Table of Contents

About

The Dynamo Primer

For Dynamo v2.0

Download the Dynamo v1.3 Primer here

Dynamo is an open source visual programming platform for designers.

Welcome

You have just opened the Dynamo Primer, a comprehensive guide to visual programming in Autodesk Dynamo. This primer is an on-going project to share the fundamentals of programming. Topics include working with computational geometry, best practices for rules-based design, cross-disciplinary programming applications, and more with the Dynamo Platform.

The power of Dynamo can be found in a wide variety of design-related activities. Dynamo enables an expanding list of readily accessible ways for you to get started:

- **Explore** visual programming for the first time
- **Connect** workflows in various software
- **Engage** an active community of users, contributors, and developers
- **Develop** an open-source platform for continued improvement

In the midst of this activity and exciting opportunity for working with Dynamo, we need a document of the same caliber, the Dynamo Primer.

This Primer includes chapters developed with Mode Lab. These chapters focus on the essentials you will need to get up and running developing your own visual programs with Dynamo and key insights on how to take Dynamo further. Here's what you can expect to learn from the primer:

- **Context** - What exactly is "Visual Programming" and what are the concepts I need to understand to dive in to Dynamo?
- **Getting Started** - How do I get Dynamo and create my first program?
- **What's in a Program** - What are the functional parts of Dynamo and how do I use them?
- **Building Blocks** - What is "Data" and what are some fundamental types I can start using in my programs?
- **Geometry for Design** - How do I work with geometric elements in Dynamo?
- **Lists, Lists, Lists** - How to do I manage and coordinate my data structures?
- **Code in Nodes** - How can I start extending Dynamo with my own code?
- **Computational BIM** - How can I use Dynamo with a Revit model?
- **Custom Nodes** - How can I create my own nodes?
- **Packages** - How can I share my tools with the community?

This is an exciting time to be learning about, working with, and developing for Dynamo. Let's get started!

Open Source

The Dynamo Primer project is open source! We're dedicated to providing quality content and appreciate any feedback you may have. If you would like to report an issue on anything at all, please post them on our GitHub issue page: https://github.com/DynamoDS/DynamoPrimer/issues

If you would like to contribute a new section, edits, or anything else to this project, check out the GitHub repo to get started: https://github.com/DynamoDS/DynamoPrimer.

The Dynamo Primer Project

The Dynamo Primer is an open source project, initiated by Matt Jezyk and the Dynamo Development team at Autodesk.

Mode Lab was commissioned to write the First Edition of the primer. We thank them for all of their efforts in establishing this valuable resource.

John Pierson of Parallax Team was commissioned to update the primer to reflect the Dynamo 2.0. revisions.

Acknowledgments

A special thanks to Ian Keough for initiating and guiding the Dynamo project.

Thank you to Matt Jezyk, Ian Keough, Zach Kron, Racel Williams and Colin McCrone for enthusiastic collaboration and the opportunity to participate on a wide array of Dynamo projects.

Software and Resources

Dynamo The current stable* release of Dynamo is Version 2.1.0

http://dynamobim.com/download/ or http://dynamobuilds.com

*Note: Starting with Revit 2020, Dynamo is bundled with Revit releases, resulting in manual installation not being required. More information is available at this blog post.

DynamoBIM The best source for additional information, learning content, and forums is the DynamoBIM website.

http://dynamobim.org

Dynamo GitHub Dynamo is an open-source development project on GitHub. To contribute, check out DynamoDS.

https://github.com/DynamoDS/Dynamo

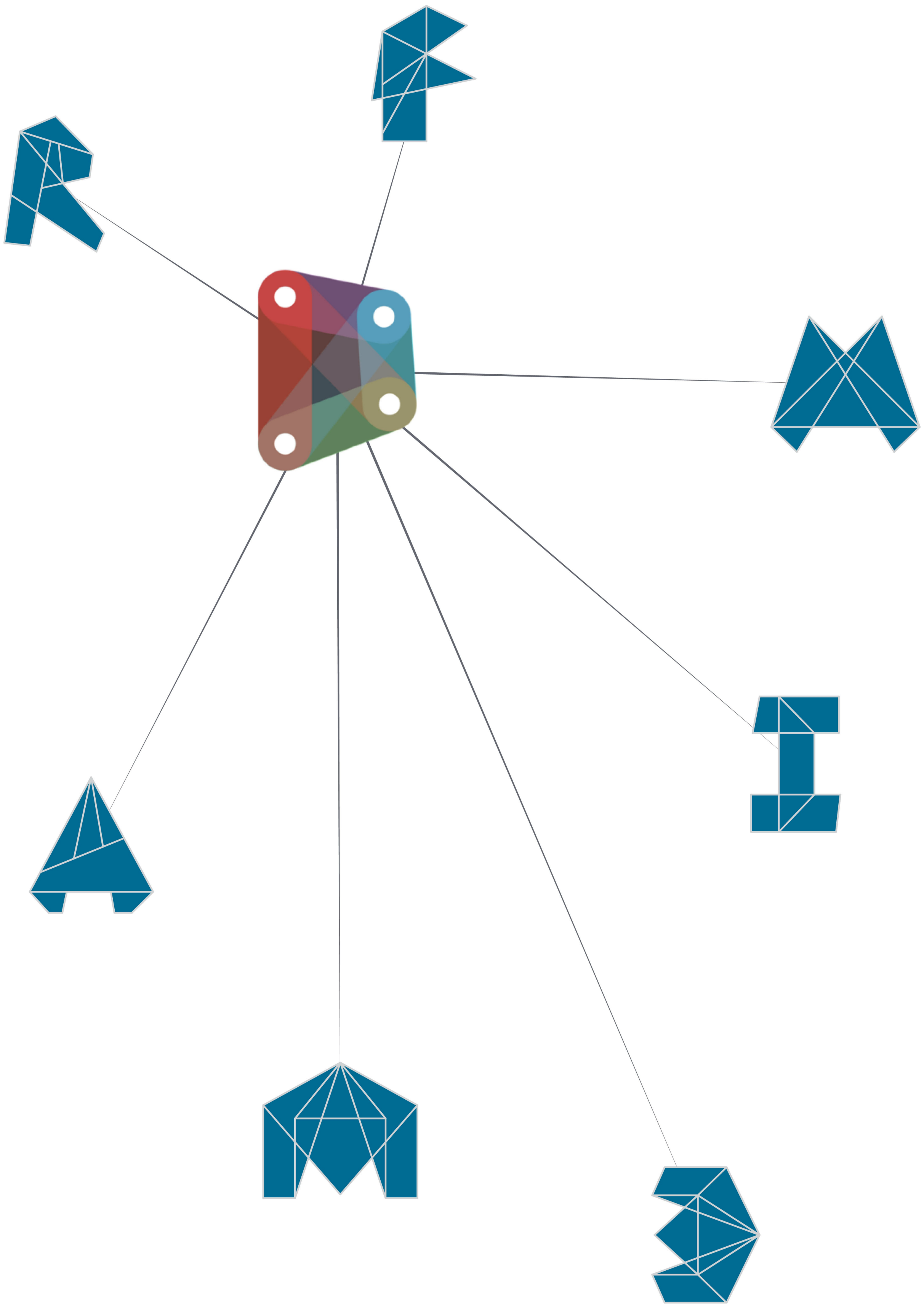
Contact Let us know about any issues with this document.

Dynamo@autodesk.com

Introduction

INTRODUCTION

From its origins as an add-on for Building Information Modeling in Revit, Dynamo has matured to become many things. Above all else it is a platform, enabling designers to explore visual programming, solve problems, and make their own tools. Let's start our journey with Dynamo by setting some context - what is it and how do I approach using it?



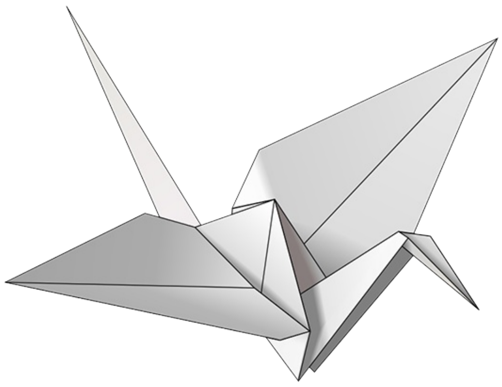
What is Visual Programming?

What is Visual Programming?

Designing frequently involves establishing visual, systemic, or geometric relationships between the parts of a design. More times than not, these relationships are developed by workflows that gets us from concept to result by way of rules. Perhaps without knowing it, we are working algorithmically - defining a step-by-step set of actions that follow a basic logic of input, processing, and output. Programming allows us to continue to work this way but by formalizing our algorithms.

Algorithms in Hand

While offering some powerful opportunities, the term **Algorithm** can carry some misconceptions with it. Algorithms can generate unexpected, wild, or cool things, but they are not magic. In fact, they are pretty plain, in and of themselves. Let's use a tangible example like an origami crane. We start with a square piece of paper (input), follow a series of folding steps (processing actions), and result in a crane (output).

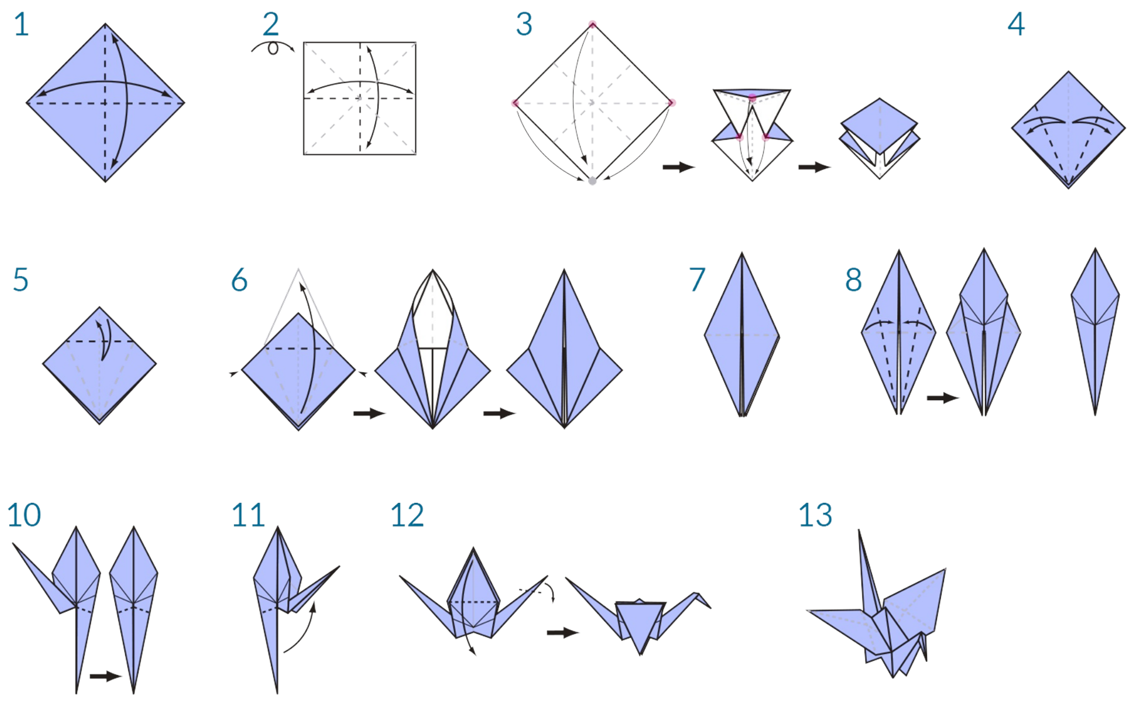


So where is the Algorithm? It is the abstract set of steps, which we can represent in a couple of ways - either textually or graphically.

Textual Instructions:

1. Start with a square piece of paper, colored side up. Fold in half and open. Then fold in half the other way.
2. Turn the paper over to the white side. Fold the paper in half, crease well and open, and then fold again in the other direction.
3. Using the creases you have made, Bring the top 3 corners of the model down to the bottom corner. Flatten model.
4. Fold top triangular flaps into the center and unfold.
5. Fold top of model downwards, crease well and unfold.
6. Open the uppermost flap of the model, bringing it upwards and pressing the sides of the model inwards at the same time. Flatten down, creasing well.
7. Turn model over and repeat Steps 4-6 on the other side.
8. Fold top flaps into the center.
9. Repeat on other side.
10. Fold both ‘legs’ of model up, crease very well, then unfold.
11. Inside Reverse Fold the “legs” along the creases you just made.
12. Inside Reverse Fold one side to make a head, then fold down the wings.
13. You now have a crane.

Graphical Instructions:



Programming Defined

Using either of these sets of instructions should result in a crane, and if you followed along yourself, you've applied an algorithm. The only difference is the way in which we read the formalization of that set of instructions and that leads us to **Programming**. Programming, frequently shortened from *Computer Programming*, is the act of formalizing the processing of a series of actions into an executable program. If we turned the above instructions for a creating crane into a format our computer can read and execute, we are Programming.

The key to and first hurdle we will find in Programming, is that we have to rely on some form of abstraction to communicate effectively with our computer. That takes the form of any number of Programming Languages, such as JavaScript, Python, or C. If we can write out a repeatable set of instructions, like for the origami crane, we only need to translate it for the computer. We are on our way to having the computer be able to make a crane or even a multitude of different cranes where each one varies slightly. This is the power of Programming - the computer will repeatedly execute whatever task, or set of tasks, we assign to it, without delay and without human error.

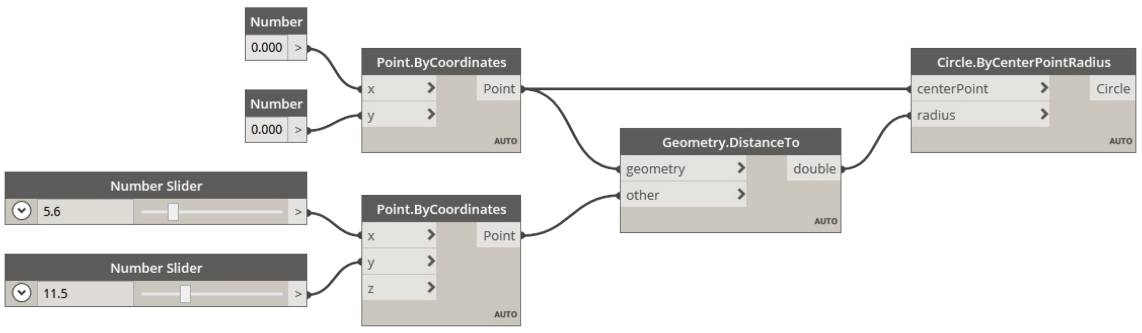
Visual Programming Defined

Download the example file that accompanies this exercise (Right click and "Save Link As..."): Visual Programming - Circle Through Point.dyn. A full list of example files can be found in the Appendix.

If you were tasked with writing instructions for folding an origami crane, how would you go about it? Would you make them with graphics, text, or some combination of the two?

If your answer contained graphics, then **Visual Programming** is definitely for you. The process is essentially the same for both Programming and Visual Programming. They utilize the same framework of formalization; however, we define the instructions and relationships of our program through a graphical (or "Visual") user interface. Instead of typing text bound by syntax, we connect pre-packaged nodes together. Here's a comparison of the same algorithm - "draw a circle through a point" - programmed with nodes versus code:

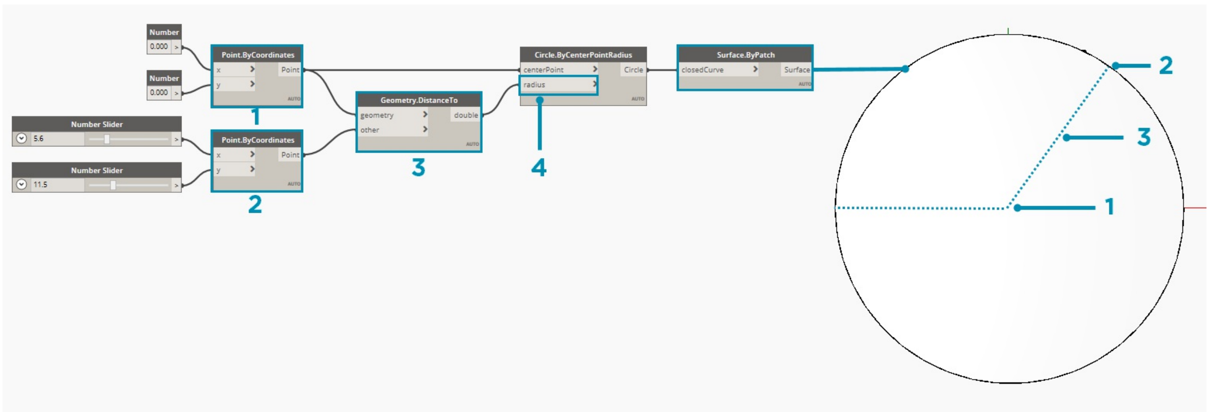
Visual Program:



Textual Program:

```
myPoint = Point.ByCoordinates(0.0,0.0,0.0);
x = 5.6;
y = 11.5;
attractorPoint = Point.ByCoordinates(x,y,0.0);
dist = myPoint.DistanceTo(attractorPoint);
myCircle = Circle.ByCenterPointRadius(myPoint,dist);
```

The results of our algorithm:



The visual characteristic to programming in such a way lowers the barrier to entry and frequently speaks to designers. Dynamo falls in the Visual Programming paradigm, but as we will see later, we can still use textual programming in the application as well.

What is Dynamo?

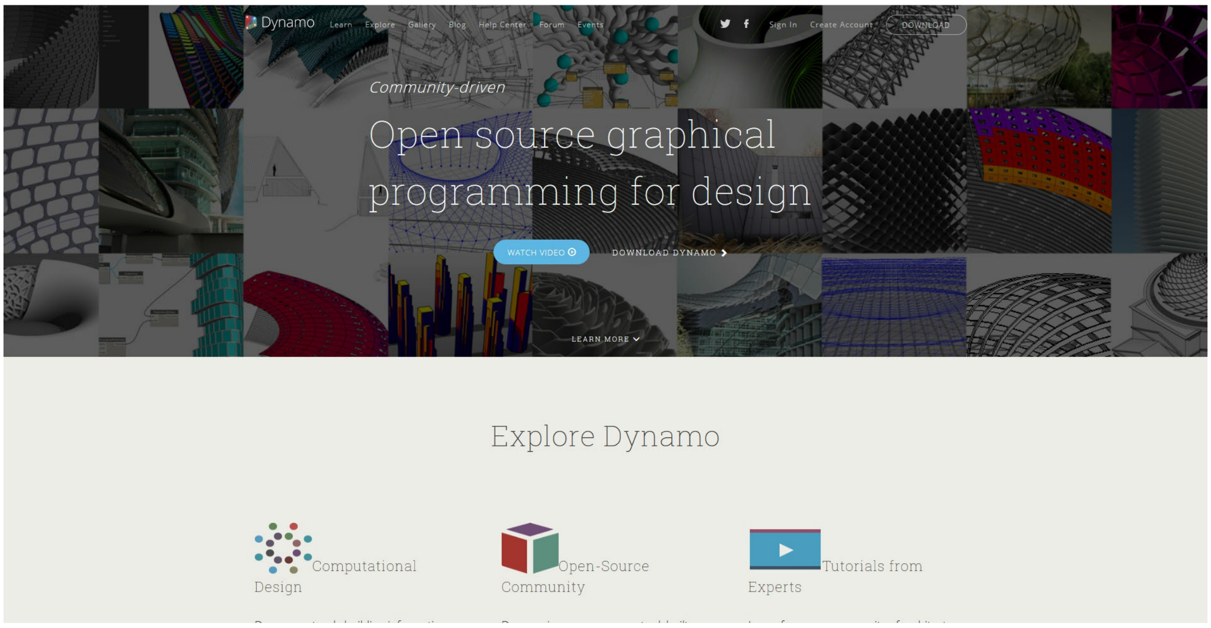
What is Dynamo?

Dynamo is, quite literally, what you make it. Working with Dynamo may include using the application, either in connection with other Autodesk software or not, engaging a Visual Programming process, or participating in a broad community of users and contributors.

The Application

Dynamo, the application, is a software that can be downloaded and run in either stand-alone "Sandbox" mode or as a plug-in for other software like Revit or Maya. It is described as:

> A visual programming tool that aims to be accessible to both non-programmers and programmers alike. It gives users the ability to visually script behavior, define custom pieces of logic, and script using various textual programming languages.



Community-driven

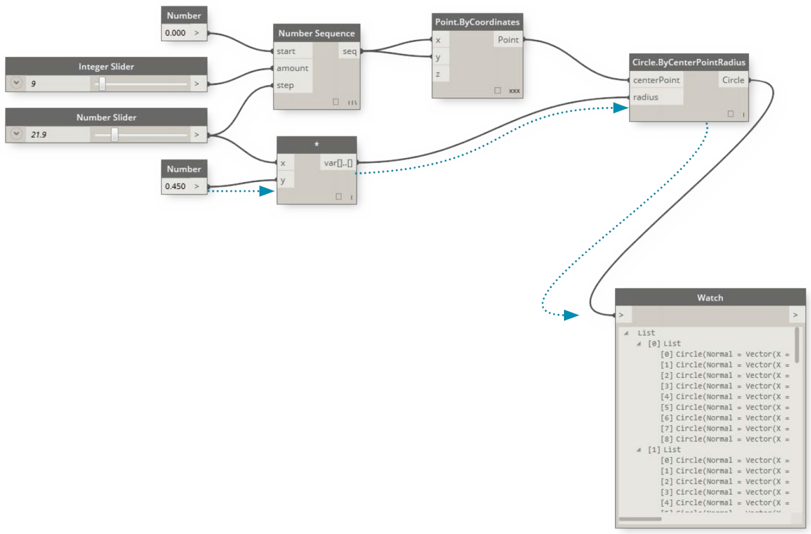
Open source graphical programming for design

Explore Dynamo

1. See Dynamo in action with Revit
2. Download the installer

The Process

Once we've installed the application, Dynamo will enable us to work within a Visual Programming process wherein we connect elements together to define the relationships and the sequences of actions that compose custom algorithms. We can use our algorithms for a wide array of applications - from processing data to generating geometry - all in real time and without writing a lick of `code`.

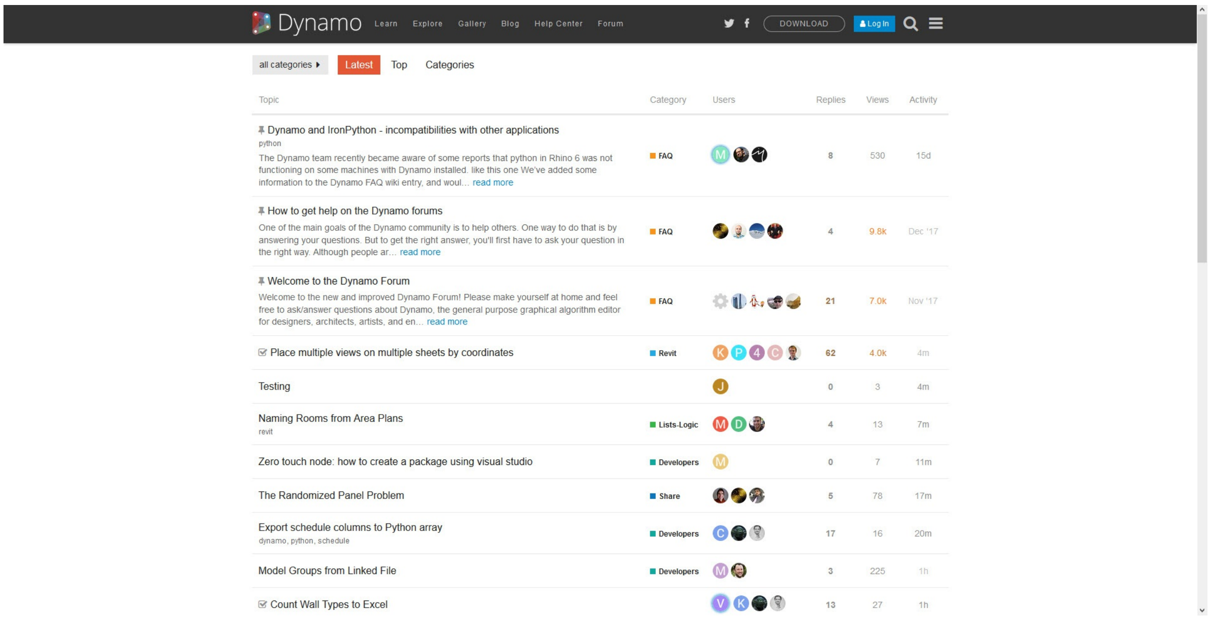


```
List
  [0] List
    [0] Circle(Normal = Vector(X =
    [1] Circle(Normal = Vector(X =
    [2] Circle(Normal = Vector(X =
    [3] Circle(Normal = Vector(X =
    [4] Circle(Normal = Vector(X =
    [5] Circle(Normal = Vector(X =
    [6] Circle(Normal = Vector(X =
    [7] Circle(Normal = Vector(X =
    [8] Circle(Normal = Vector(X =
  [1] List
    [0] Circle(Normal = Vector(X =
    [1] Circle(Normal = Vector(X =
    [2] Circle(Normal = Vector(X =
    [3] Circle(Normal = Vector(X =
    [4] Circle(Normal = Vector(X =
```

> Add elements, connect, and we are off and running with creating Visual Programs.

The Community

Dynamo wouldn't be what it is without a strong group of avid users and active contributors. Engage the community by following the Blog, adding your work to the Gallery, or discussing Dynamo in the Forum.

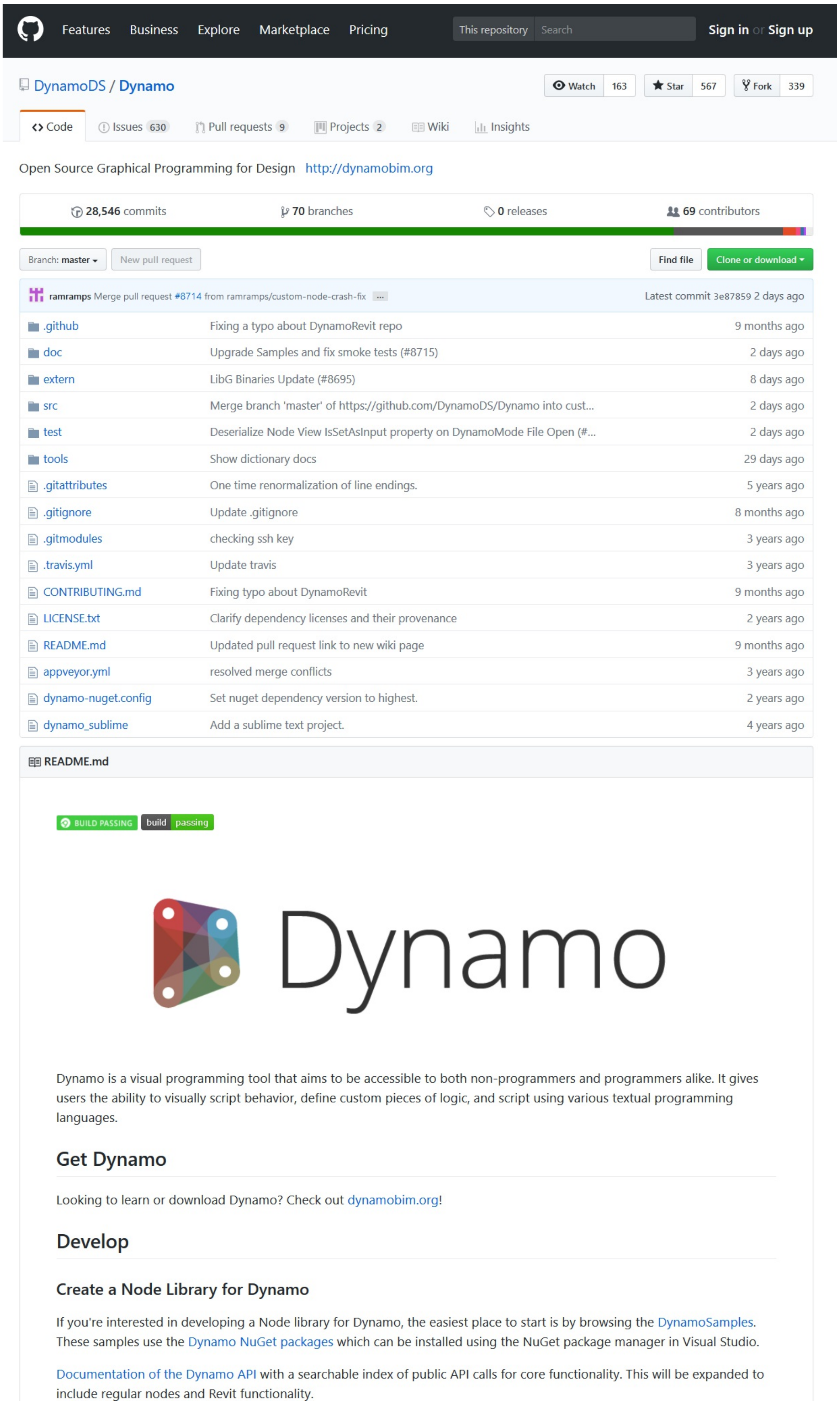


Topic	Category	Users	Replies	Views	Activity
Dynamo and IronPython - incompatibilities with other applications python The Dynamo team recently became aware of some reports that python in Rhino 6 was not functioning on some machines with Dynamo installed. like this one We've added some information to the Dynamo FAQ wiki entry, and woul… read more	FAQ	M	8	530	15d
How to get help on the Dynamo forums One of the main goals of the Dynamo community is to help others. One way to do that is by answering your questions. But to get the right answer, you'll first have to ask your question in the right way. Although people ar… read more	FAQ		4	9.8k	Dec '17
Welcome to the Dynamo Forum Welcome to the new and improved Dynamo Forum! Please make yourself at home and feel free to ask/answer questions about Dynamo, the general purpose graphical algorithm editor for designers, architects, artists, and en… read more	FAQ		21	7.0k	Nov '17
Place multiple views on multiple sheets by coordinates	Revit	K P 4 C	62	4.0k	4m
Testing		J	0	3	4m
Naming Rooms from Area Plans revit	Lists-Logic	M D	4	13	7m
Zero touch node: how to create a package using visual studio	Developers	M	0	7	11m
The Randomized Panel Problem	Share		5	78	17m
Export schedule columns to Python array dynamo, python, schedule	Developers	C	17	16	20m
Model Groups from Linked File	Developers	M	3	225	1h
Count Wall Types to Excel		V K	13	27	1h

The Platform

Dynamo is envisioned as a visual programming tool for designers, allowing us to make tools that make use of external libraries or any Autodesk product that has an API. With Dynamo Sandbox we can develop programs in a "Sandbox" style application - but the Dynamo ecosystem continues to grow.

The source code for the project is open-source, enabling us to extend its functionality to our hearts content. Check out the project on GitHub and browse the Works in Progress of users customizing Dynamo.



DynamoDS / Dynamo

Open Source Graphical Programming for Design http://dynamobim.org

28,546 commits | 70 branches | 0 releases | 69 contributors

ramramps Merge pull request #8714 from ramramps/custom-node-crash-fix ... Latest commit 3e87859 2 days ago

.github	Fixing a typo about DynamoRevit repo	9 months ago
doc	Upgrade Samples and fix smoke tests (#8715)	2 days ago
extern	LibG Binaries Update (#8695)	8 days ago
src	Merge branch 'master' of https://github.com/DynamoDS/Dynamo into cust...	2 days ago
test	Deserialize Node View IsSetAsInput property on DynamoMode File Open (#...	2 days ago
tools	Show dictionary docs	29 days ago
.gitattributes	One time renormalization of line endings.	5 years ago
.gitignore	Update .gitignore	8 months ago
.gitmodules	checking ssh key	3 years ago
.travis.yml	Update travis	3 years ago
CONTRIBUTING.md	Fixing typo about DynamoRevit	9 months ago
LICENSE.txt	Clarify dependency licenses and their provenance	2 years ago
README.md	Updated pull request link to new wiki page	9 months ago
appveyor.yml	resolved merge conflicts	3 years ago
dynamo-nuget.config	Set nuget dependency version to highest.	2 years ago
dynamo_sublime	Add a sublime text project.	4 years ago

README.md

Dynamo is a visual programming tool that aims to be accessible to both non-programmers and programmers alike. It gives users the ability to visually script behavior, define custom pieces of logic, and script using various textual programming languages.

Get Dynamo

Looking to learn or download Dynamo? Check out dynamobim.org!

Develop

Create a Node Library for Dynamo

If you're interested in developing a Node library for Dynamo, the easiest place to start is by browsing the DynamoSamples. These samples use the Dynamo NuGet packages which can be installed using the NuGet package manager in Visual Studio.

Documentation of the Dynamo API with a searchable index of public API calls for core functionality. This will be expanded to include regular nodes and Revit functionality.

Browse, Fork, and start extending Dynamo for your needs

Dynamo in Action

DYNAMO IN ACTION

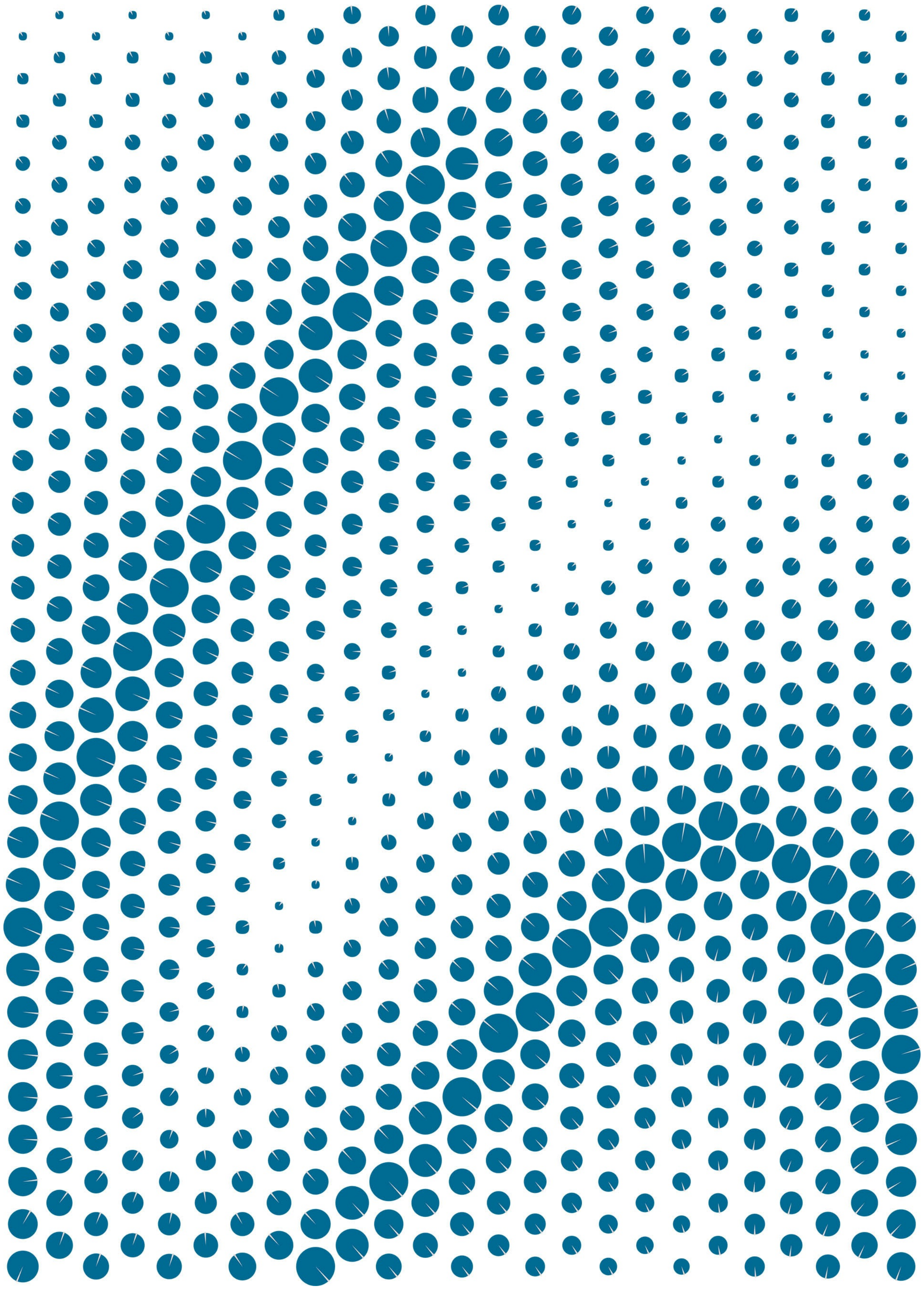
From using Visual Programming for project workflows to developing customized tools, Dynamo is an integral aspect to a wide variety of exciting applications.

Follow the Dynamo in Action board on Pinterest.

Hello Dynamo!

HELLO DYNAMO!

At its core, Dynamo is a platform for Visual Programming - it is a flexible and extensible design tool. Because it can operate as a stand-alone application or as an add-on to other design software, we can use it to develop a wide range of creative workflows. Let's install Dynamo and get started by reviewing the key features of the interface.



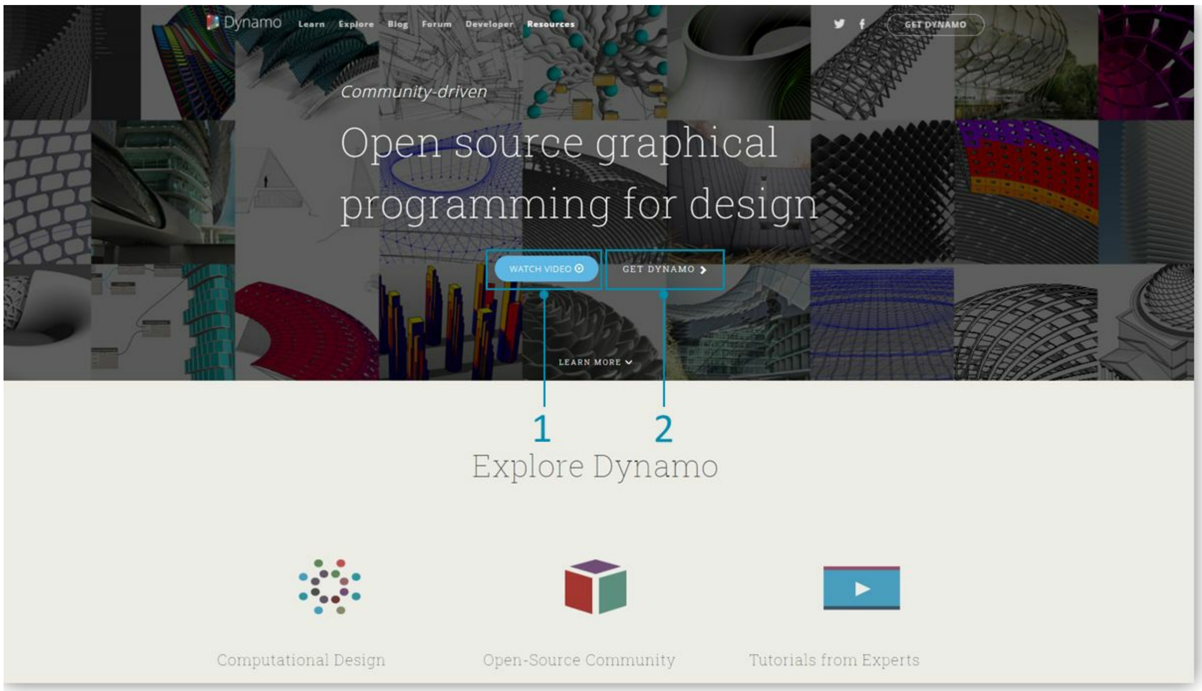
Installing and Launching Dynamo

Installing and Launching Dynamo

Dynamo is an active open-source development project with downloadable installers for both official and pre-release, i.e.. "daily build" versions. Download the official release to get started, or contribute to what Dynamo becomes through the daily builds or GitHub project.

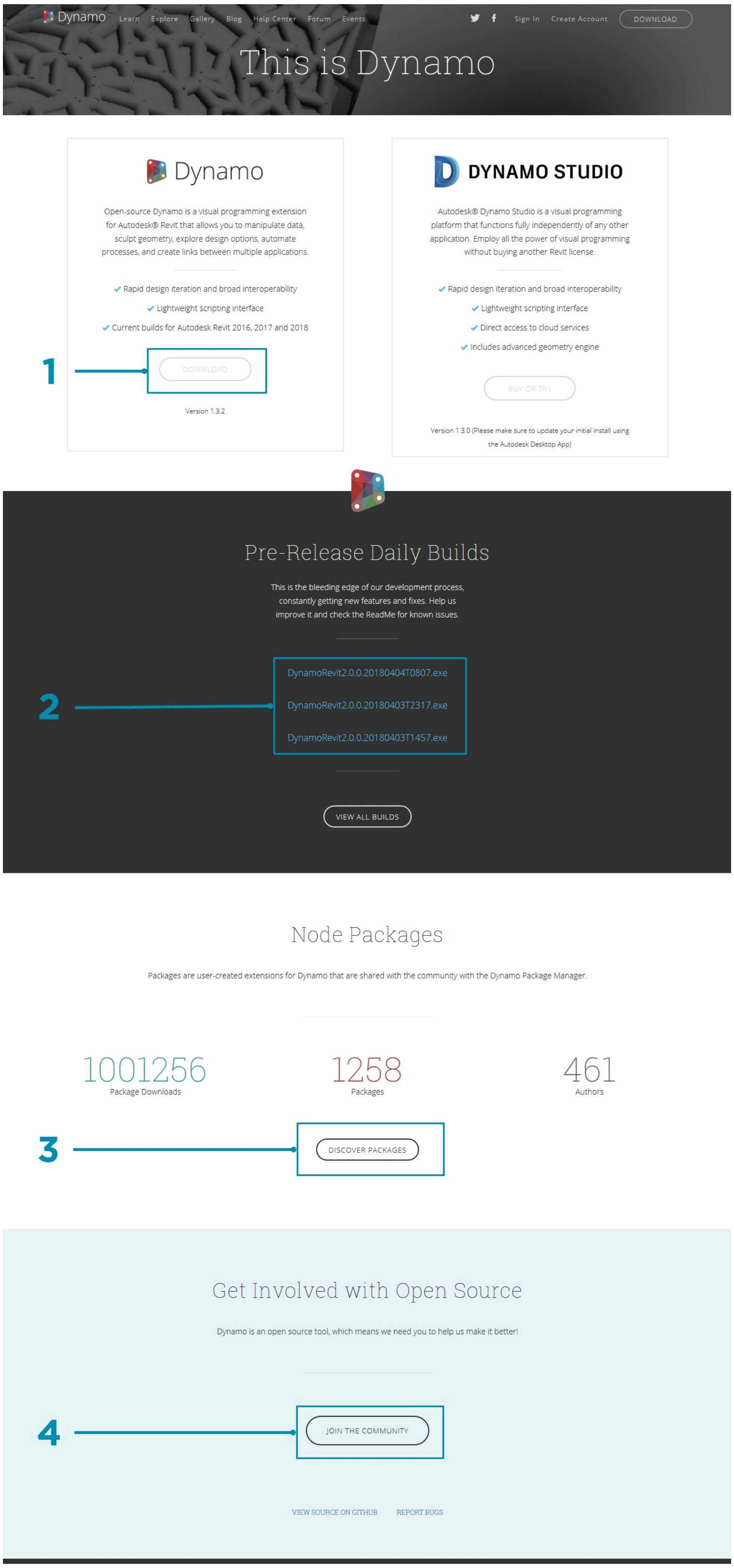
Downloading

To download the official released version of Dynamo, visit the Dynamo website. Start the download immediately by clicking from the homepage or browse to the dedicated download page.



1. Watch a video on Computational Design with Dynamo for Architecture
2. Or browse to the download page

Here you can download the "bleeding edge" development versions or go to the Dynamo Github project.



This is Dynamo

Dynamo

Open-source Dynamo is a visual programming extension for Autodesk® Revit that allows you to manipulate data, sculpt geometry, explore design options, automate processes, and create links between multiple applications.

✓ Rapid design iteration and broad interoperability
✓ Lightweight scripting interface
✓ Current builds for Autodesk Revit 2016, 2017 and 2018

Version 1.3.2

DYNAMO STUDIO

Autodesk® Dynamo Studio is a visual programming platform that functions fully independently of any other application. Employ all the power of visual programming without buying another Revit license.

✓ Rapid design iteration and broad interoperability
✓ Lightweight scripting interface
✓ Direct access to cloud services
✓ Includes advanced geometry engine

Version 1.3.0 (Please make sure to update your initial install using the Autodesk Desktop App)

Pre-Release Daily Builds

This is the bleeding edge of our development process, constantly getting new features and fixes. Help us improve it and check the ReadMe for known issues.

DynamoRevit2.0.0.20180404T0807.exe
DynamoRevit2.0.0.20180403T2317.exe
DynamoRevit2.0.0.20180403T1457.exe

Node Packages

Packages are user-created extensions for Dynamo that are shared with the community with the Dynamo Package Manager.

1001256
Package Downloads

1258
Packages

461
Authors

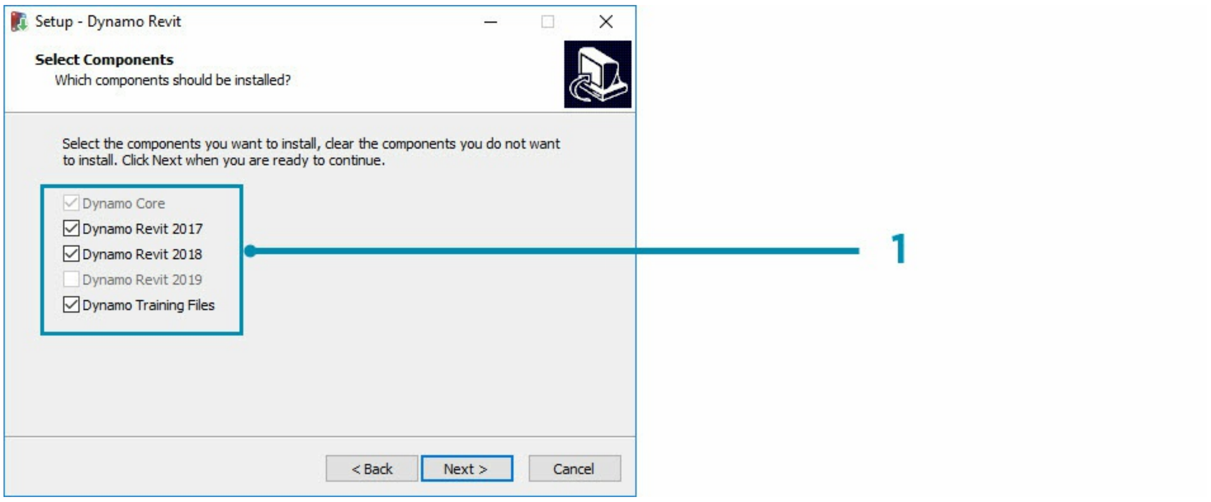
Get Involved with Open Source

Dynamo is an open source tool, which means we need you to help us make it better!

1. Download the official release installer
2. Download the daily build installers
3. Check out custom packages from a community of developers
4. Get involved in the development of Dynamo on GitHub

Installing

Browse to the directory of the downloaded installer and run the executable file. During the installation process, the setup allows you to customize the components that will be installed.

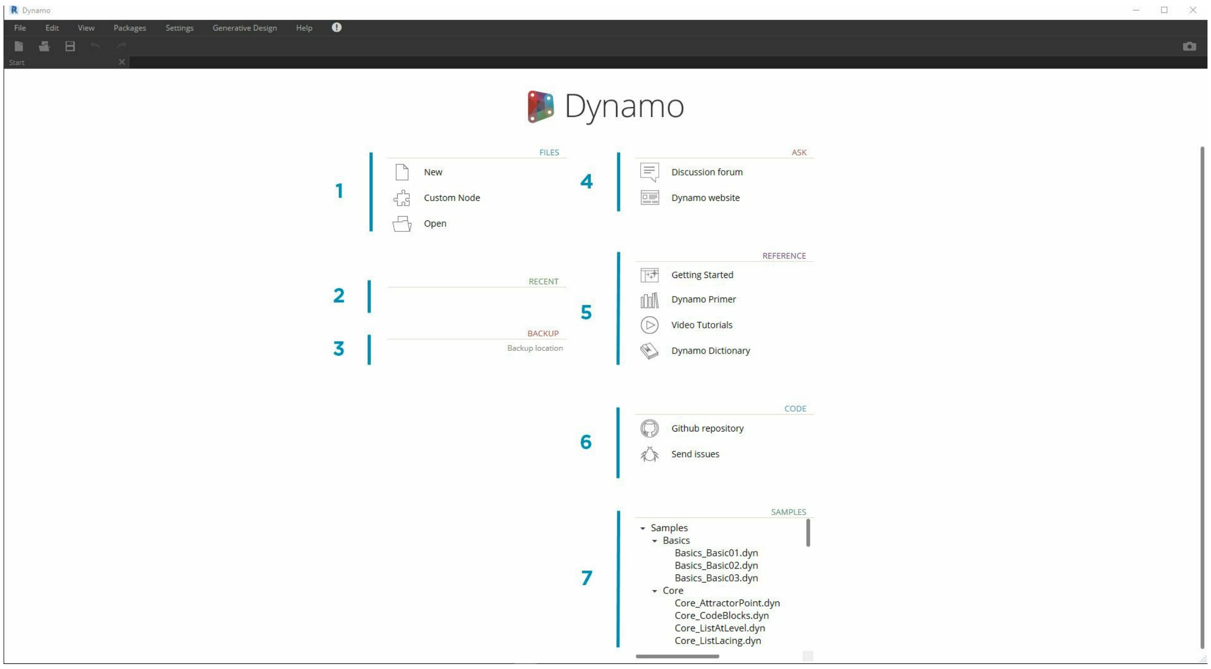

Setup - Dynamo Revit
Select Components
Which components should be installed?
Select the components you want to install, clear the components you do not want to install. Click Next when you are ready to continue.
Dynamo Core
Dynamo Revit 2017
Dynamo Revit 2018
Dynamo Revit 2019
Dynamo Training Files
1
< Back
Next >
Cancel

1. Select the Components you want to install

Here we need to decide if we want to include the components that connect Dynamo to other installed applications such as Revit. For more information on the Dynamo Platform, see **Chapter 1.2**.

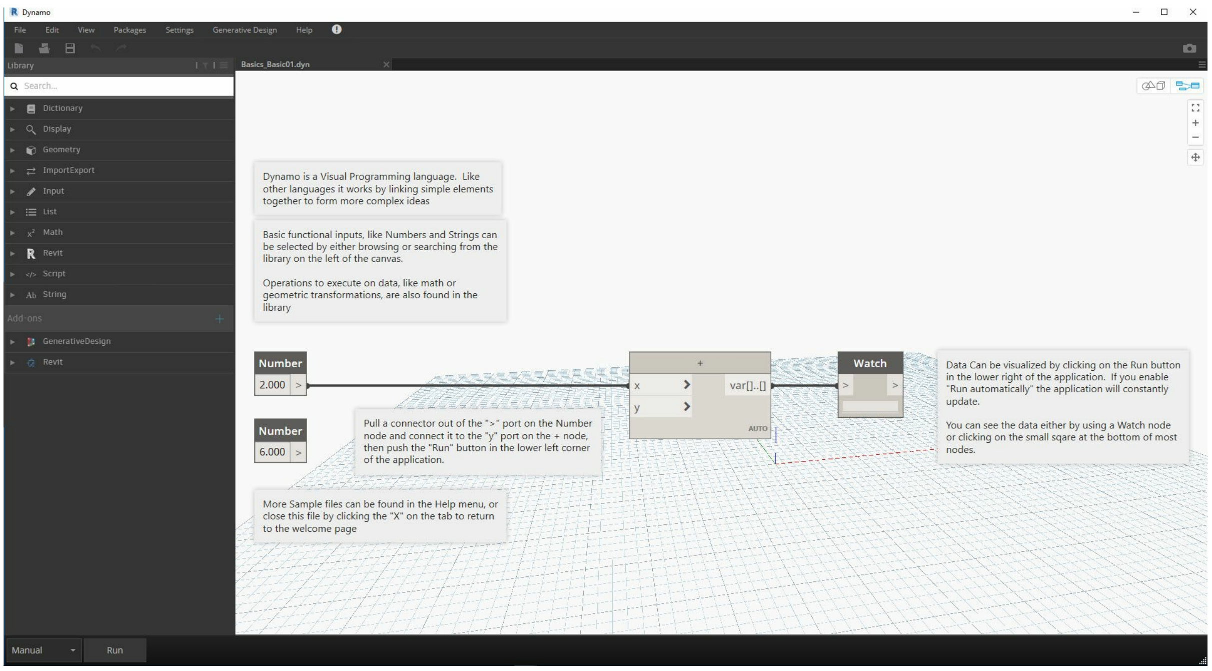
Launching

To launch Dynamo, browse to \Program Files\Dynamo\Dynamo Revit\x.y, then select DynamoSandbox.exe. This will open the stand-alone version and present Dynamo's *Start Page*. On this page, we see the standard menus and toolbar as well as a collection of shortcuts that allow us to access file functionality or access additional resources.



1. Files - Start a new file or open an existing one
2. Recent - Scroll through your recent files
3. Backup - Access to your backups
4. Ask - Get direct access to the User Forum or Dynamo Website
5. Reference - Go further with additional learning resources
6. Code - Participate in the open-source development project
7. Samples - Check out the examples that come with the installation

Open the first sample file to open your first workspace and confirm Dynamo is working correctly. Click Samples > Basics > **Basics_Basic01.dyn**.



Dynamo is a Visual Programming language. Like other languages it works by linking simple elements together to form more complex ideas

Basic functional inputs, like Numbers and Strings can be selected by either browsing or searching from the library on the left of the canvas.

Operations to execute on data, like math or geometric transformations, are also found in the library

Pull a connector out of the ">" port on the Number node and connect it to the "y" port on the + node, then push the "Run" button in the lower left corner of the application.

More Sample files can be found in the Help menu, or close this file by clicking the "X" on the tab to return to the welcome page

Data Can be visualized by clicking on the Run button in the lower right of the application. If you enable "Run automatically" the application will constantly update.

You can see the data either by using a Watch node or clicking on the small sqare at the bottom of most nodes.

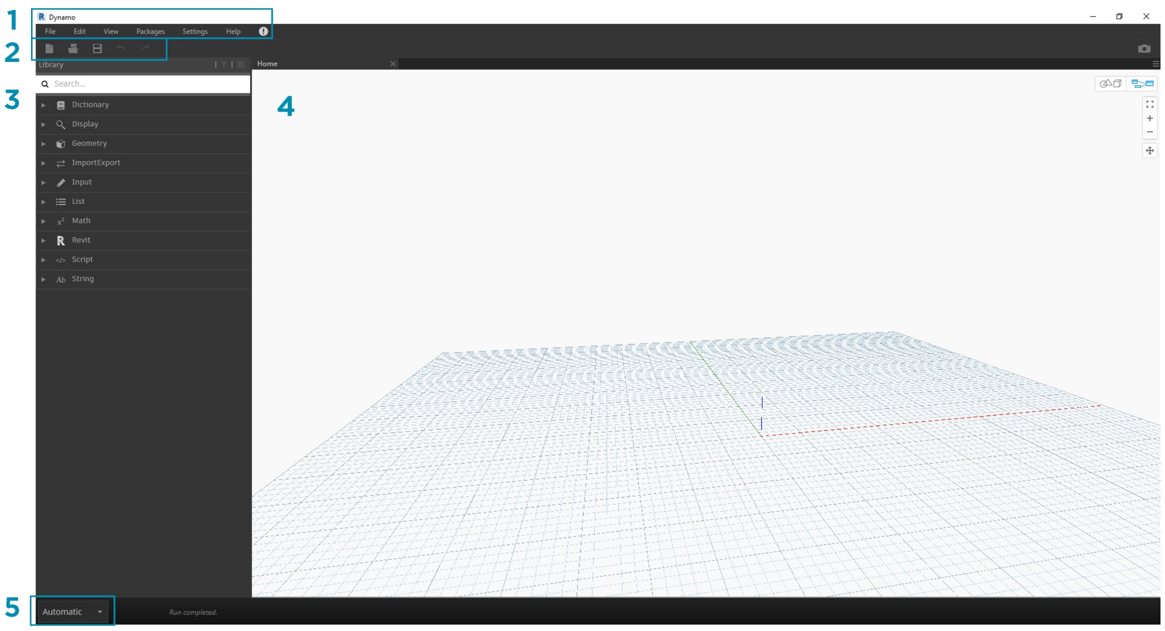
1. Confirm that the Execution Bar says "Automatic" or click Run
2. Follow the instructions and connect the **Number** Node to the **+** Node
3. Confirm that this Watch Node shows a result

If this file successfully loads, you should be able to execute your first visual program with Dynamo.

The User Interface

The Dynamo User Interface

The User Interface (UI) for Dynamo is organized into five main regions, the largest of which is the workspace where we compose our visual programs.

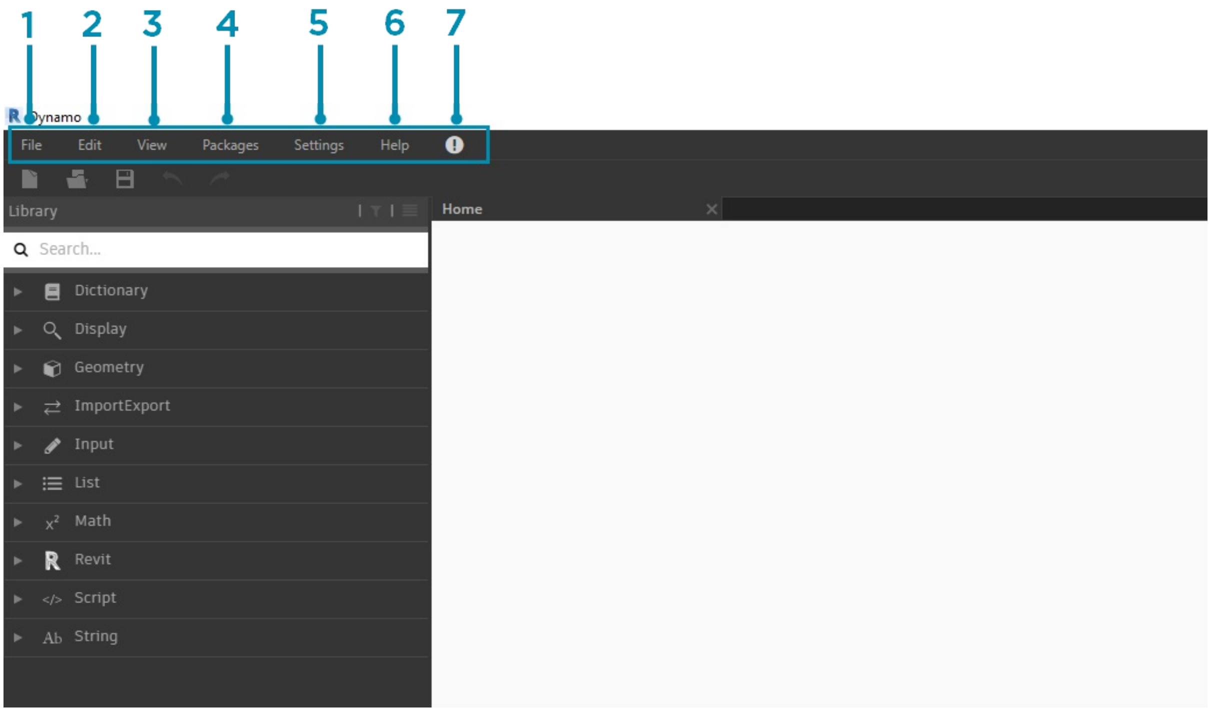


1. Menus
2. Toolbar
3. Library
4. Workspace
5. Execution Bar

Let's dive deeper into the UI and explore the functionality of each region.

Menus

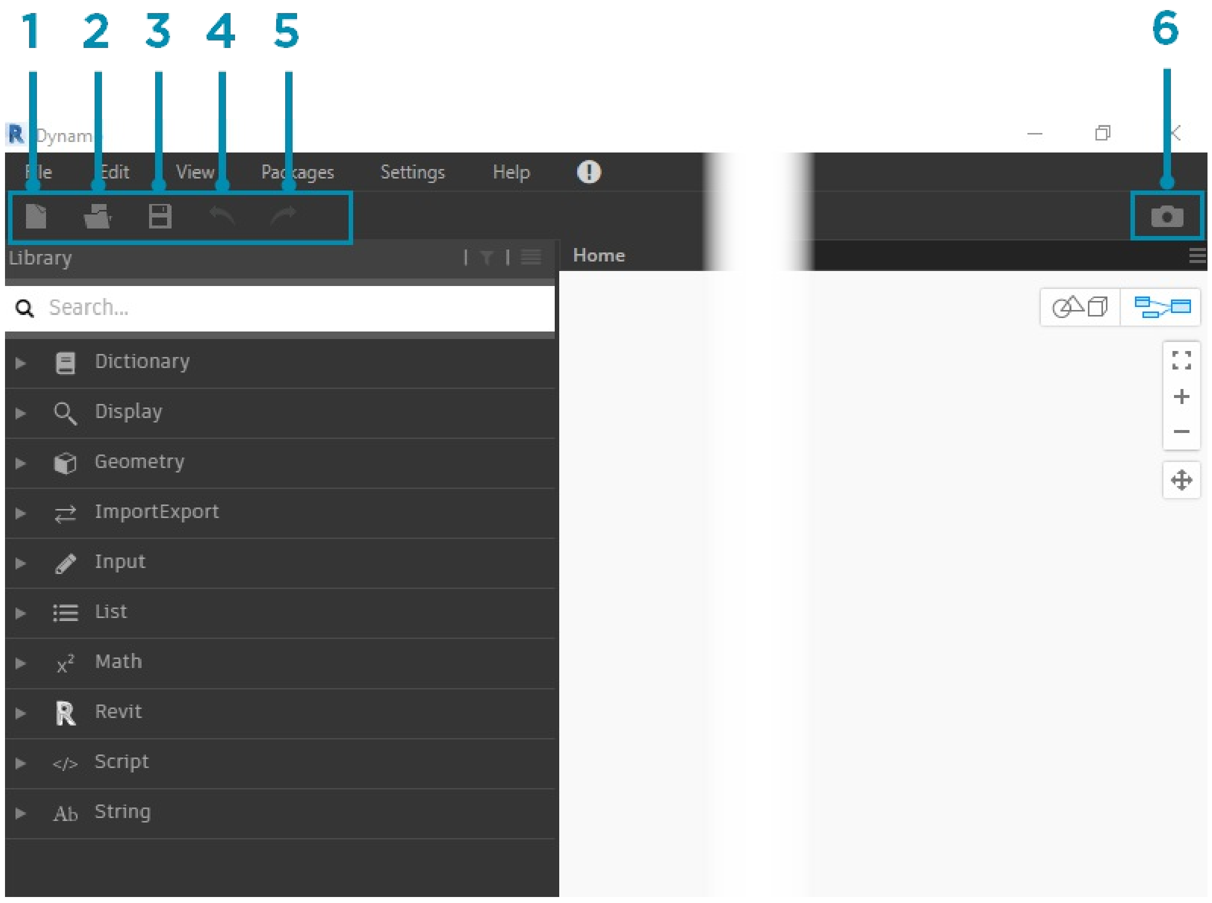
The Dropdown Menus are a great place to find some of the basic functionality of the Dynamo application. Like most Windows software, actions related to managing files and operations for selection and content editing are found in the first two menus. The remaining menus are more specific to Dynamo.



1. File
2. Edit
3. View
4. Packages
5. Settings
6. Help
7. Notifications

Toolbar

Dynamo's Toolbar contains a series of buttons for quick access to working with files as well as Undo [Ctrl + Z] and Redo [Ctrl + Y] commands. On the far right is another button that will export a snapshot of the workspace, which is extremely useful for documentation and sharing.



1. New - Create a new .dyn file
2. Open - Open an existing .dyn (workspace) or .dyf (custom node) file
3. Save/Save As - Save your active .dyn or .dyf file
4. Undo - Undo your last action
5. Redo - Redo the next action
6. Export Workspace as Image - Export the visible workspace as a PNG file

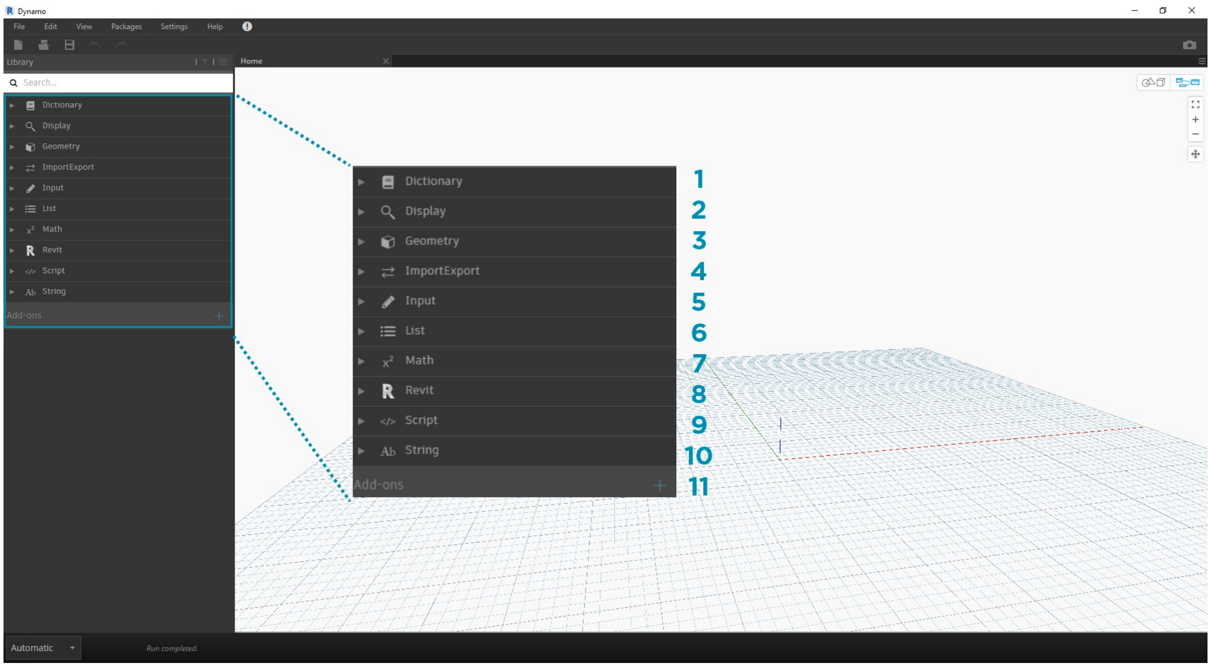
Library

The Library contains all of the loaded Nodes, including the default Nodes that come with the installation as well as any additionally loaded Custom Nodes or Packages. The Nodes in the Library are organized hierarchically within libraries, categories, and, where appropriate, sub-categories based on whether the Nodes **Create** data, execute an **Action**, or **Query** data.

Browsing

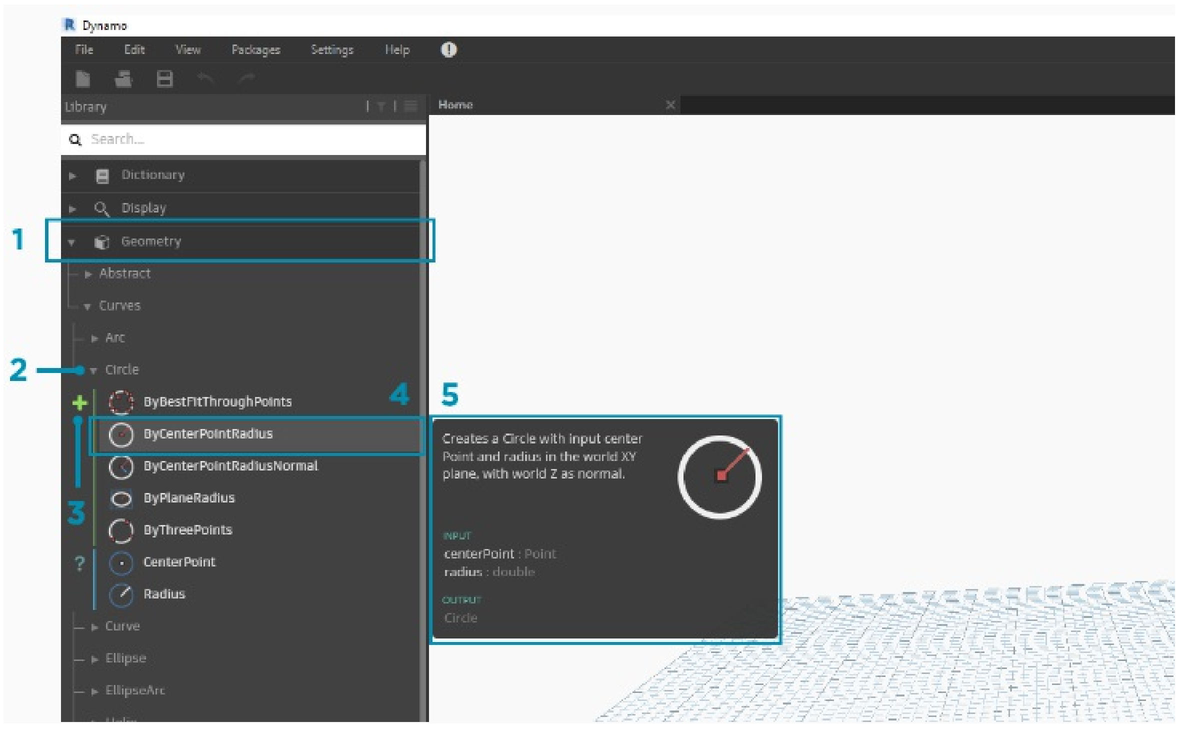
By default, the **Library** will contain eight categories of Nodes. **Core** and **Geometry** are great menus to begin exploring as they contain the largest quantity of Nodes. Browsing through these categories is the fastest way to understand the hierarchy of what we can add to our Workspace and the best way to discover new Nodes you haven't used before.

> We will focus on the default collection of Nodes now, but note that we will extend this Library with Custom Nodes, additional libraries, and the Package Manager later.



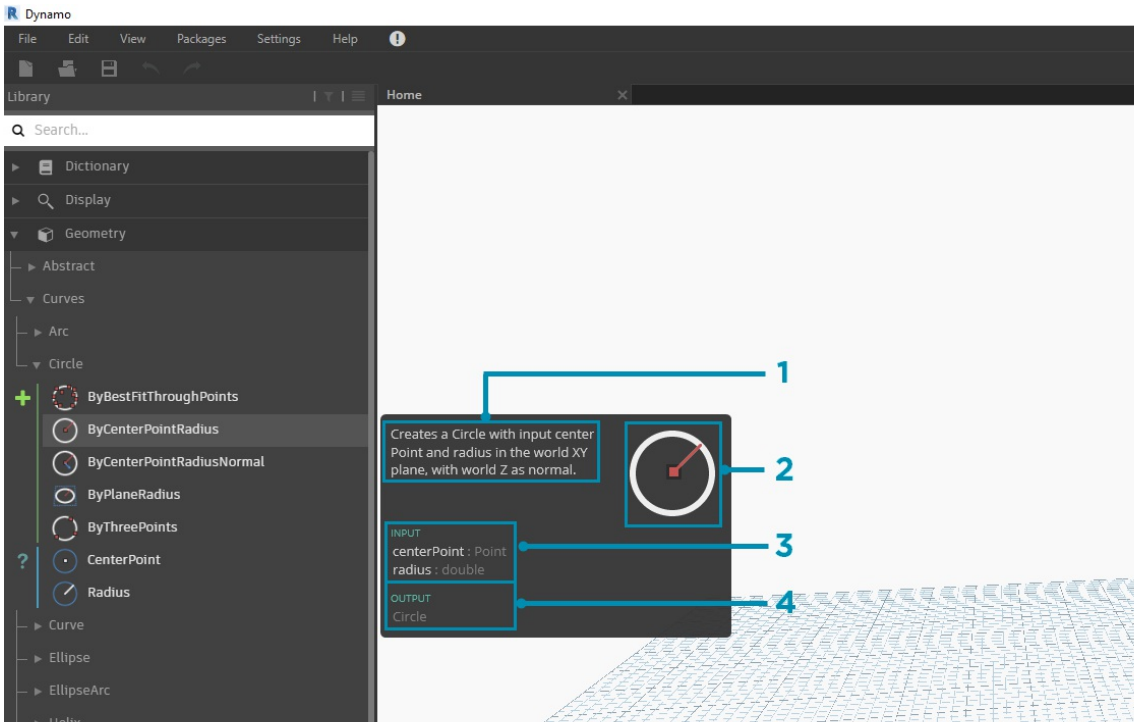
1. Dictionary
2. Display
3. Geometry
4. ImportExport
5. Input
6. List
7. Math
8. Revit
9. Script
10. String
11. Add-ons

Browse the Library by clicking through the menus. Click the Geometry > Curves > Circle. Note the new portion of the menu that is revealed and specifically the **Create** and **Query** Labels.



1. Library
2. Category
3. Subcategory: Create/Actions/Query
4. Node
5. Node Description and properties - this appears when hovering over the node icon.

From the same Circle menu, hover your mouse over **ByCenterPointRadius**. The window reveals more detailed information about the Node beyond its name and icon. This offers us a quick way to understand what the Node does, what it will require for inputs, and what it will give as an output.



Creates a Circle with input center Point and radius in the world XY plane, with world Z as normal.

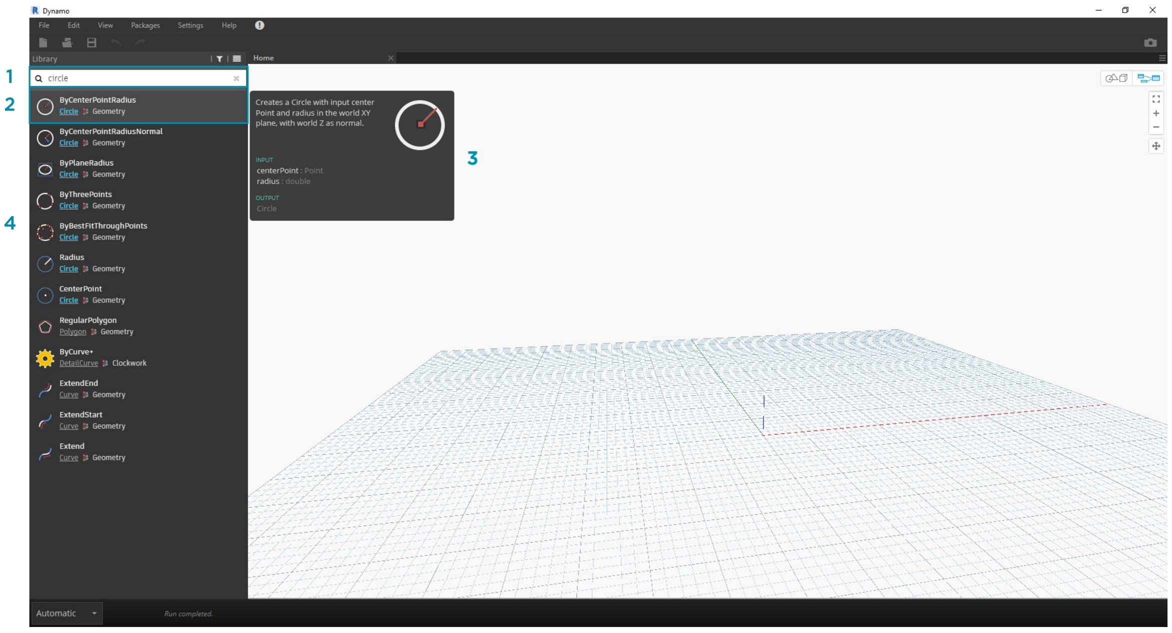
INPUT
centerPoint : Point
radius : double

OUTPUT
Circle

1. Description - plain language description of the Node
2. Icon - larger version of the icon in the Library Menu
3. Input(s) - name, data type, and data structure
4. Output(s) - data type and structure

Searching

If you know with relative specificity which Node you want to add to your Workspace, the **Search** field is your best friend. When you are not editing settings or specifying values in the Workspace, the cursor is always present in this field. If you start typing, the Dynamo Library will reveal a selected best fit match (with breadcrumbs for where it can be found in the Node categories) and a list of alternate matches to the search. When you hit Enter, or click on the item in the truncated browser, the highlighted Node is added to the center of the Workspace.



Creates a Circle with input center Point and radius in the world XY plane, with world Z as normal.

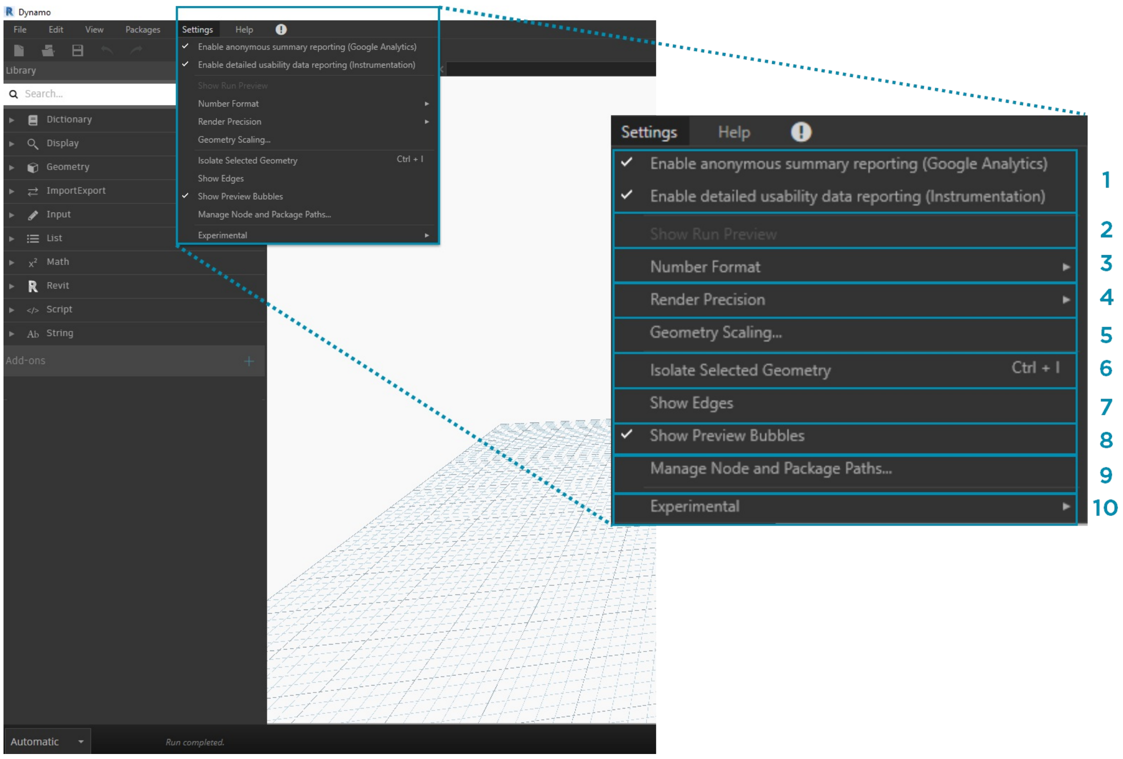
INPUT
centerPoint : Point
radius : double

OUTPUT
Circle

1. Search Field
2. Best Fit Result / Selected
3. Alternate Matches

Settings

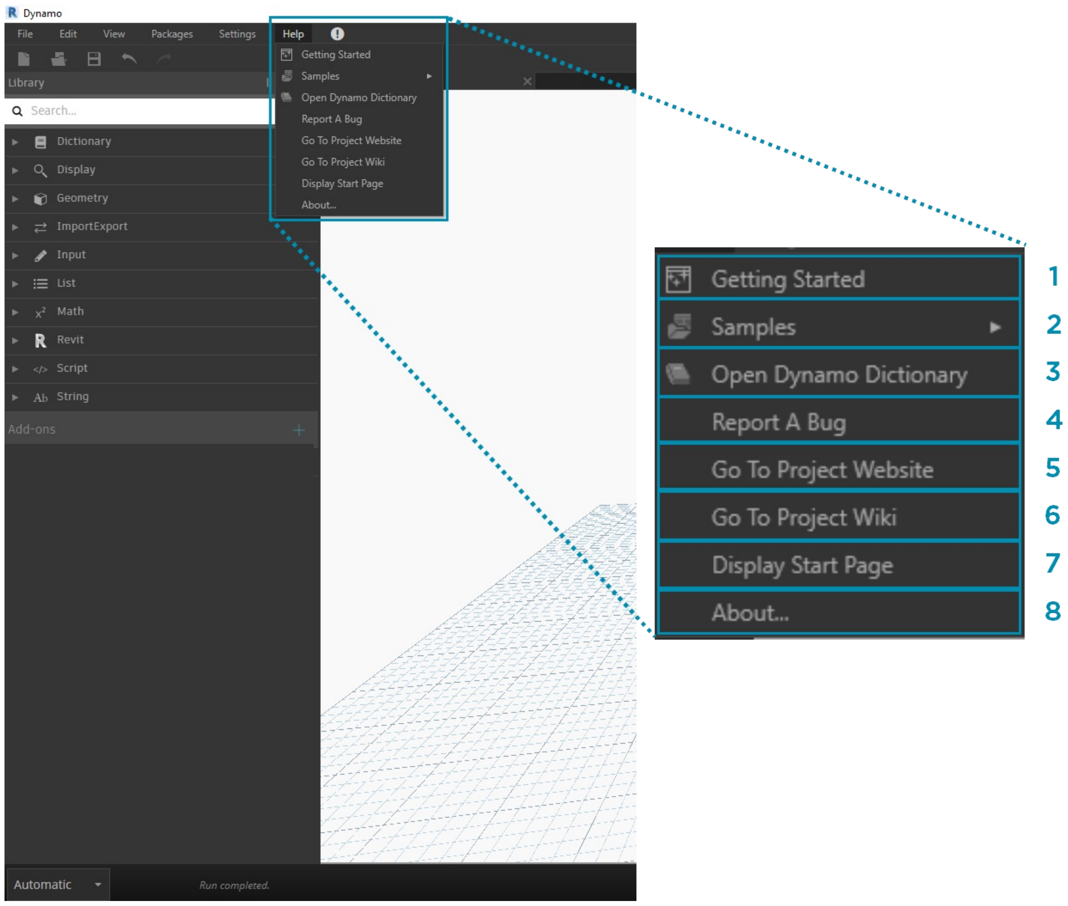
From geometric to user settings, these options can be found in the **Settings** menu. Here you can opt in or out for sharing your user data to improve Dynamo as well as define the application's decimal point precision and geometry render quality.



1. Enabling Reporting - Options for sharing user data to improve Dynamo.
2. Show Run Preview - Preview the execution state of your graph. Nodes scheduled for execution will be highlighted in your graph.
3. Number Format Options - Change the document settings for decimals.
4. Render Precision - Raise or lower the document render quality.
5. Geometry Scaling - Select range of geometry you are working on.
6. Isolate Selected Geometry - Isolated background geometry based on your node selection.
7. Show/Hide Geometry Edges - Toggle 3D geometry edges.
8. Show/Hide Preview Bubbles - Toggle data preview bubbles below nodes.
9. Manage Node and Package Paths - Manage file paths to make nodes and packages show up in the Library.
10. Enabling Experimental Features - Use beta features new in Dynamo.

Help

If you're stuck, check out the **Help** Menu. Here you can find the sample files that come with your installation as well as access one of the Dynamo reference websites through your internet browser. If you need to, check the version of Dynamo installed and whether it is up to date through the **About** option.



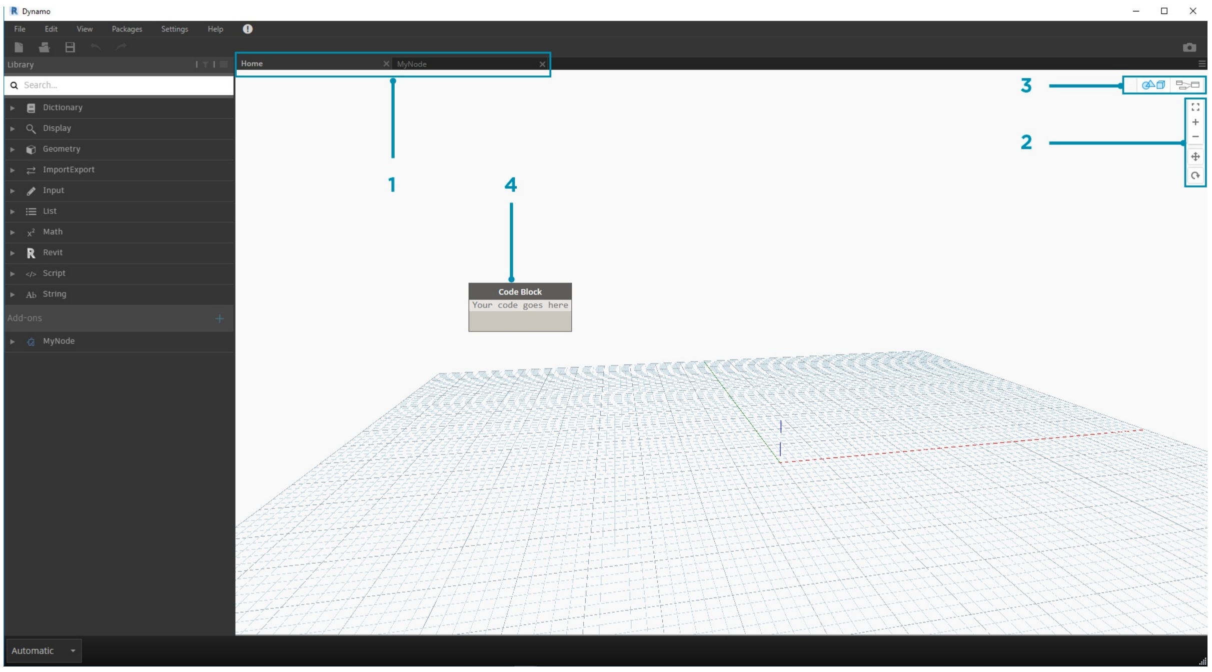
1. Getting Started - A brief introduction to using Dynamo.
2. Samples - Reference example files.
3. Open Dynamo Dictionary - Resource with documentation on all nodes.
4. Report A Bug - Open an Issue on GitHub.
5. Go To Project Website - View the Dynamo Project on GitHub.
6. Go To Project Wiki - Visit the wiki for learning about development using the Dynamo API, supporting libraries and tools.
7. Display Start Page - Return to the Dynamo start page when within a document.
8. About - Dynamo Version data.

The Workspace

The Workspace

The Dynamo **Workspace** is where we develop our visual programs, but it's also where we preview any resulting geometry. Whether we are working in a Home Workspace or a Custom Node, we can navigate with our mouse or the buttons at top right. Toggling between modes at bottom right switches which preview we navigate.

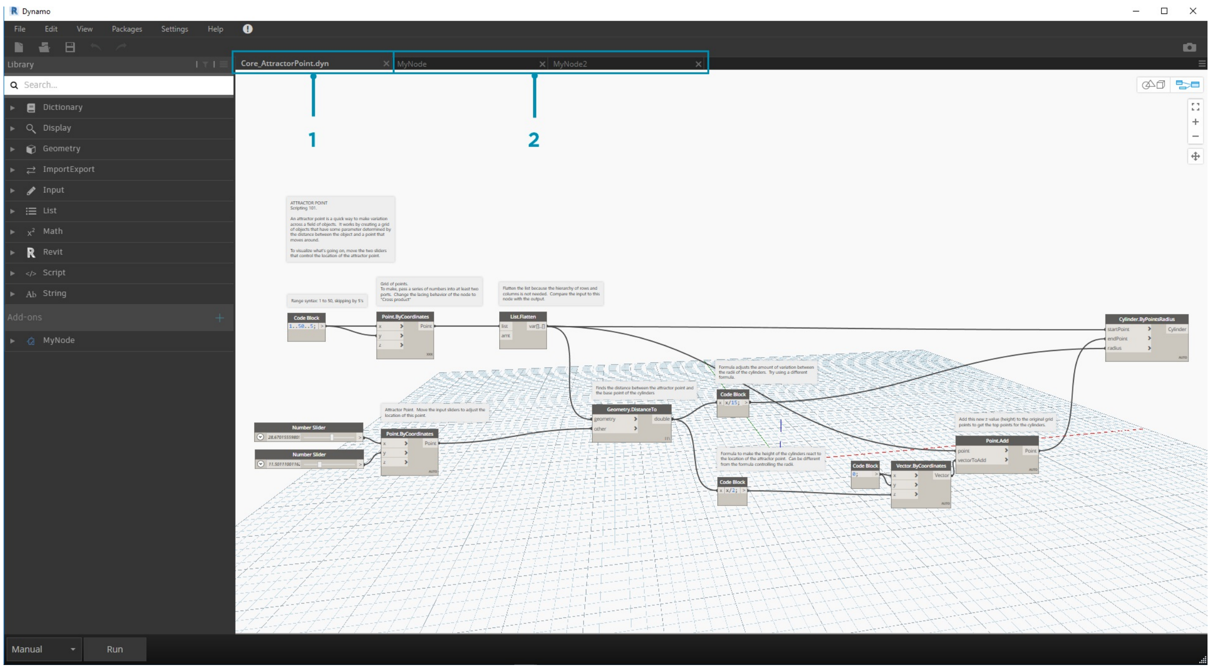
> Note: Nodes and geometry have a draw order so you may have objects rendered on top of each other. This can be confusing when adding multiple nodes in sequence as they may be rendered in the same position in the Workspace.



1. Tabs
2. Zoom/Pan Buttons
3. Preview Mode
4. Double Clicking on the Workspace

Tabs

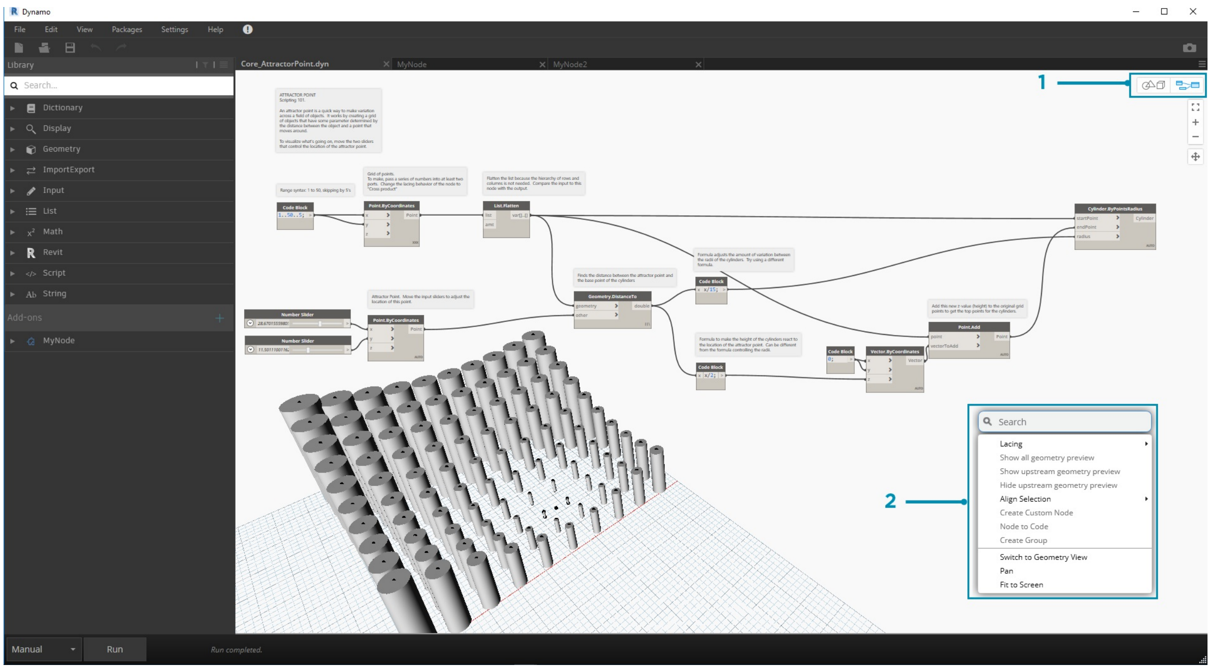
The active Workspace tab allows you to navigate and edit your program. When you open a new file, by default you are opening a new **Home** Workspace. You may also open a new **Custom Node** Workspace from the File Menu or by the *New Node by Selection* right click option when Nodes are selected (more on this functionality later).



Note: You may have only one Home Workspace open at a time; however, you may have multiple Custom Node Workspaces open in additional tabs.

Graph versus 3D Preview Navigation

In Dynamo, the Graph and the 3D results of the Graph (if we are creating geometry) are both rendered in the Workspace. By default the Graph is the active preview, so using the Navigation buttons or middle mouse button to pan and zoom will move us through the Graph. Toggling between active previews can be achieved three ways:

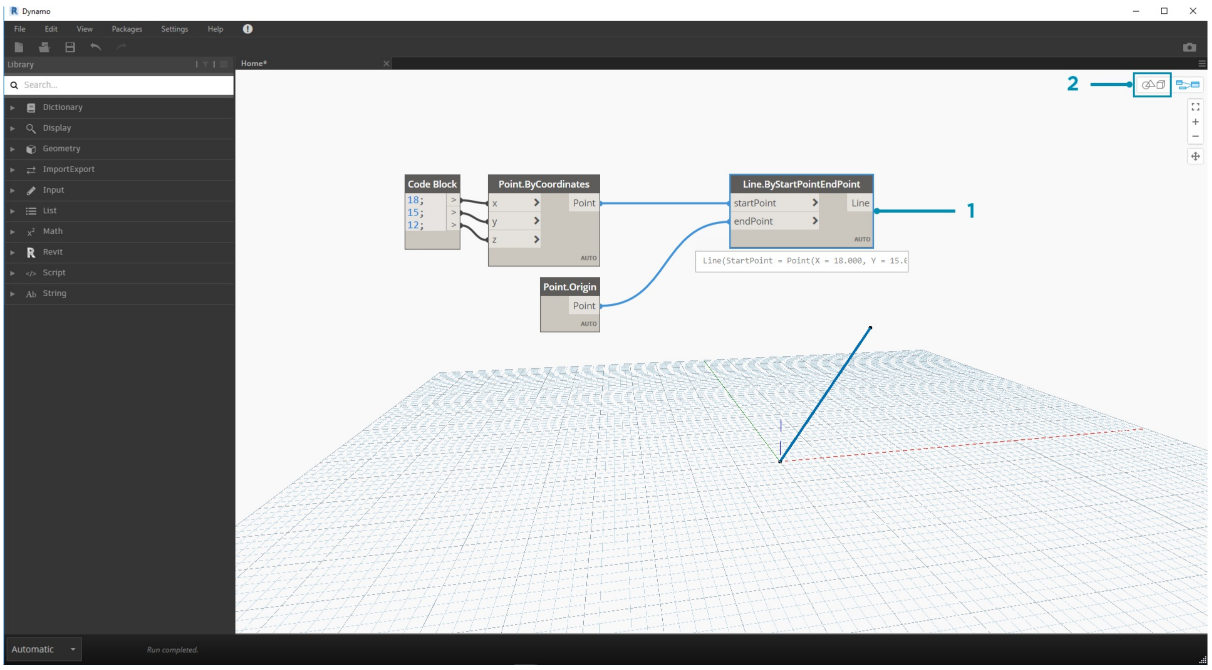


1. Preview Toggle Buttons in the Workspace
2. Right clicking in the Workspace and selecting *Switch to ... View*
3. Keyboard shortcut (Ctrl + B)

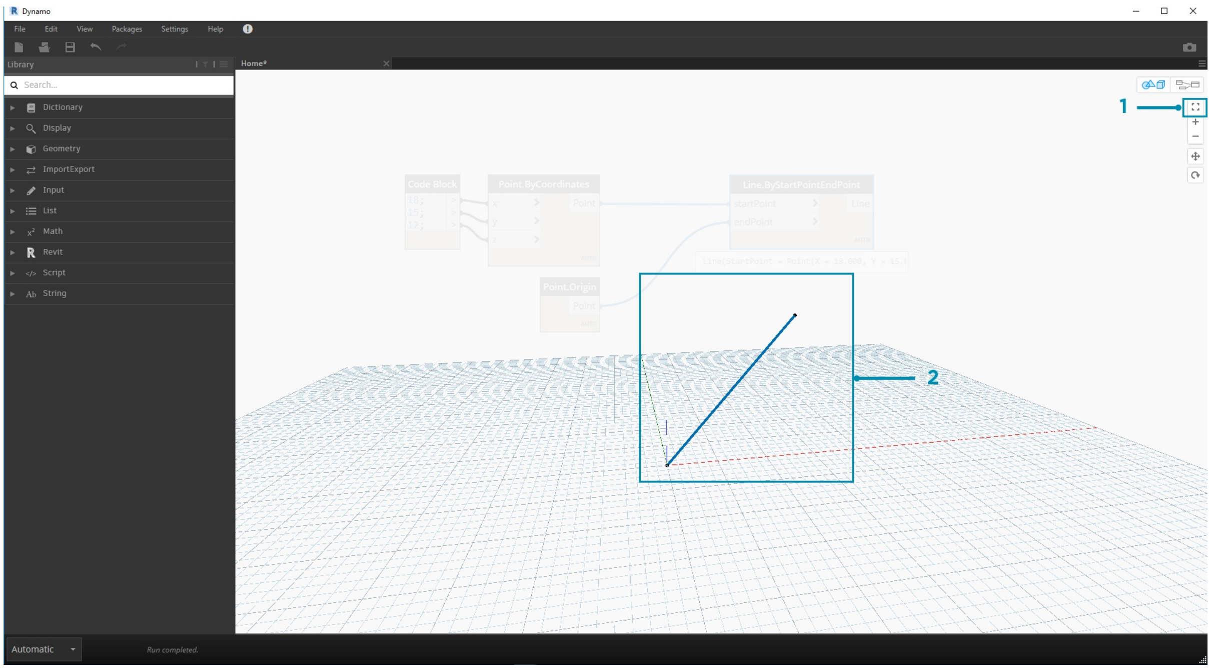
The 3D Preview Navigation mode also gives us the ability for **Direct Manipulation** of points, exemplified in Getting Started.

Zoom to Recenter

We can easily pan, zoom and rotate freely around models in 3D Preview Navigation mode. However, to zoom specifically on an object created by a geometry node, we can use the Zoom All icon with a single node selected.



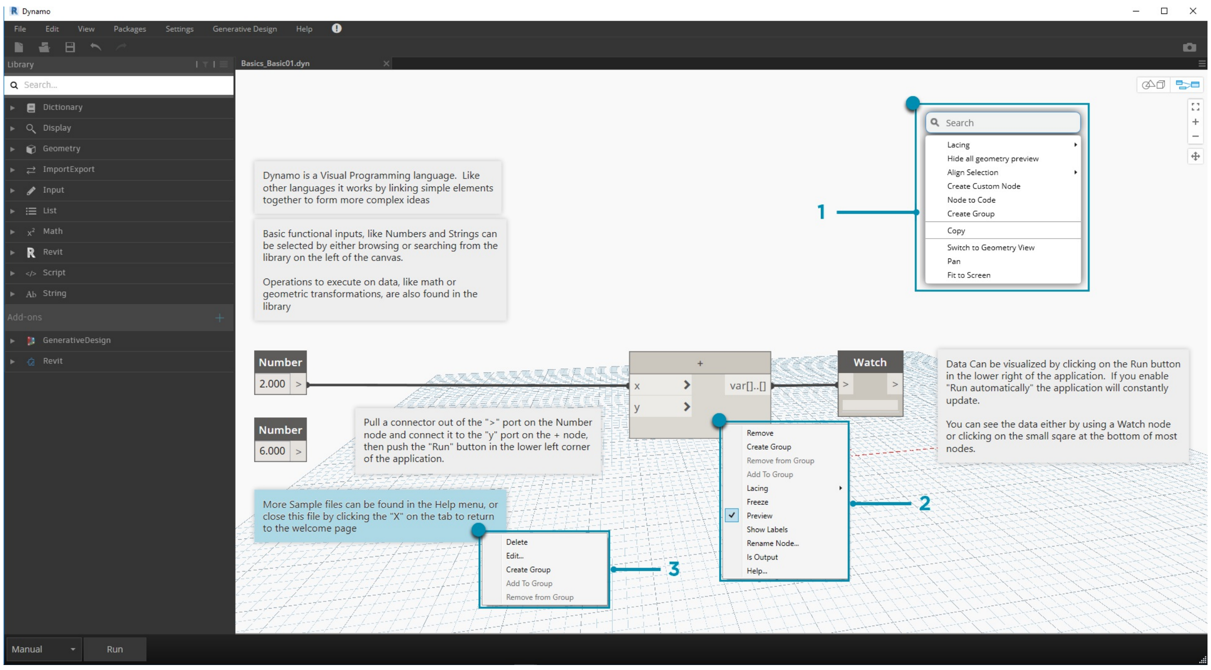
1. Select the node corresponding to the geometry that will center the view.
2. Switch to the 3D Preview Navigation.



1. Click on the Zoom All icon in the top right.
2. The selected geometry will be centered inside the view.

Hello Mouse!

Based on which Preview mode is active, your mouse buttons will act differently. In general, the left mouse click selects and specifies inputs, the right mouse click gives access to options, and the middle mouse click allows you to navigate the Workspace. The right mouse click will present us with options based on the context of where we are clicking.


Dynamo is a Visual Programming language. Like other languages it works by linking simple elements together to form more complex ideas

Basic functional inputs, like Numbers and Strings can be selected by either browsing or searching from the library on the left of the canvas.

Operations to execute on data, like math or geometric transformations, are also found in the library

Pull a connector out of the ">" port on the Number node and connect it to the "y" port on the + node, then push the "Run" button in the lower left corner of the application.

Data Can be visualized by clicking on the Run button in the lower right of the application. If you enable "Run automatically" the application will constantly update.

You can see the data either by using a Watch node or clicking on the small sqare at the bottom of most nodes.

More Sample files can be found in the Help menu, or close this file by clicking the "X" on the tab to return to the welcome page

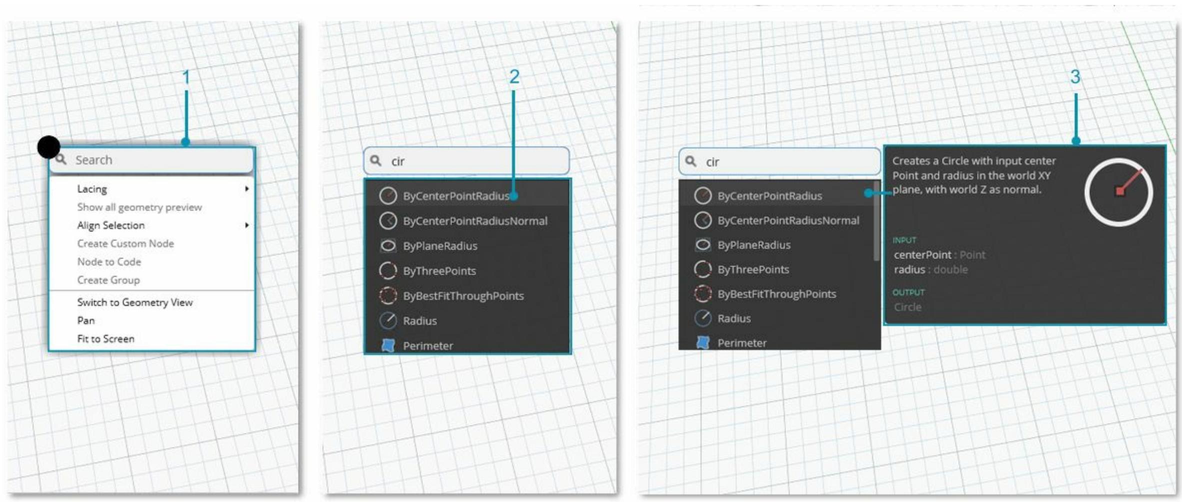
1. Right Click on the Workspace.
2. Right Click on a Node.
3. Right Click on a Note.

Here's a table of mouse interactions per Preview:

Mouse Action	Graph Preview	3D Preview
Left Click	Select	N/A
Right Click	Context Menu	Zoom Options
Middle Click	Pan	Pan
Scroll	Zoom In/Out	Zoom In/Out
Double Click	Create Code Block	N/A

In-Canvas Search

Using the "In-Canvas Search" will add some serious speed to your Dynamo work-flow by providing you access to node descriptions and tool-tips without taking you away from your place on the graph! By just right-clicking, you can access all the useful functionality of the "Library Search" from wherever you happen to be working on the canvas.

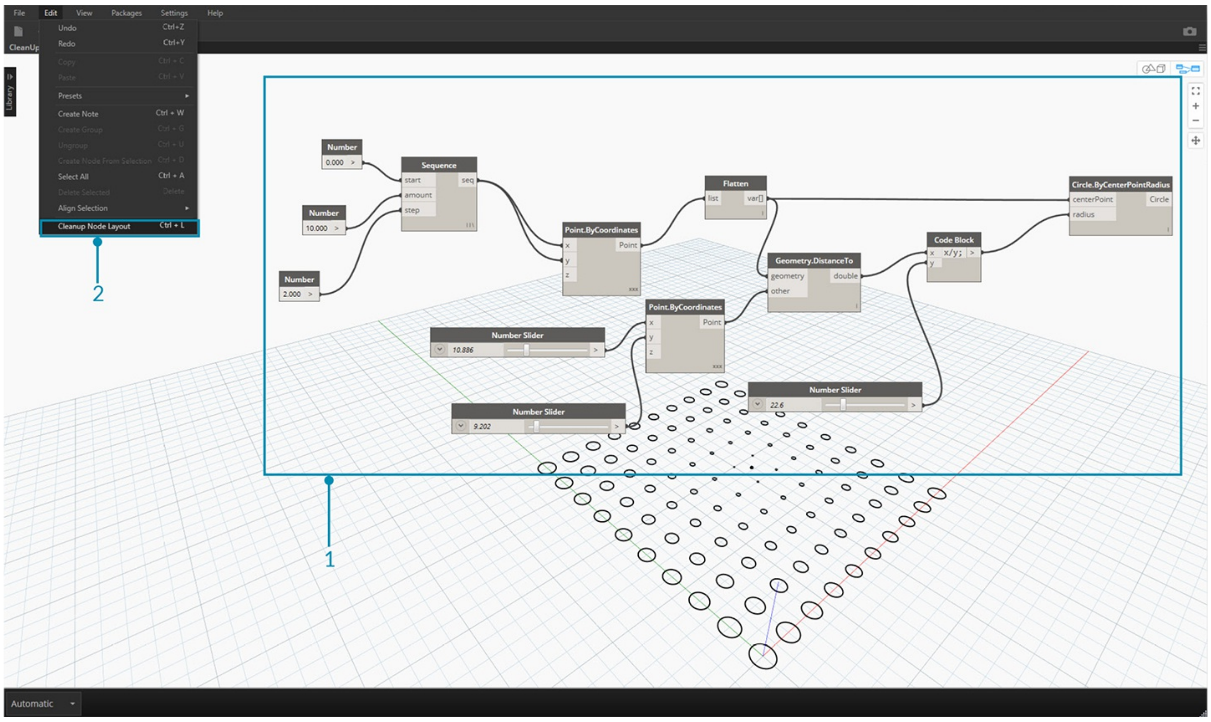


1. Right click anywhere on the canvas to bring up the search feature. While the search bar is empty, the drop-down will be a preview menu.
2. As you type into the search bar, the drop-down menu will continuously update to show the most relevant search results.
3. Hover over the search results to bring up their corresponding descriptions and tool-tips.

Clean Up Node Layout

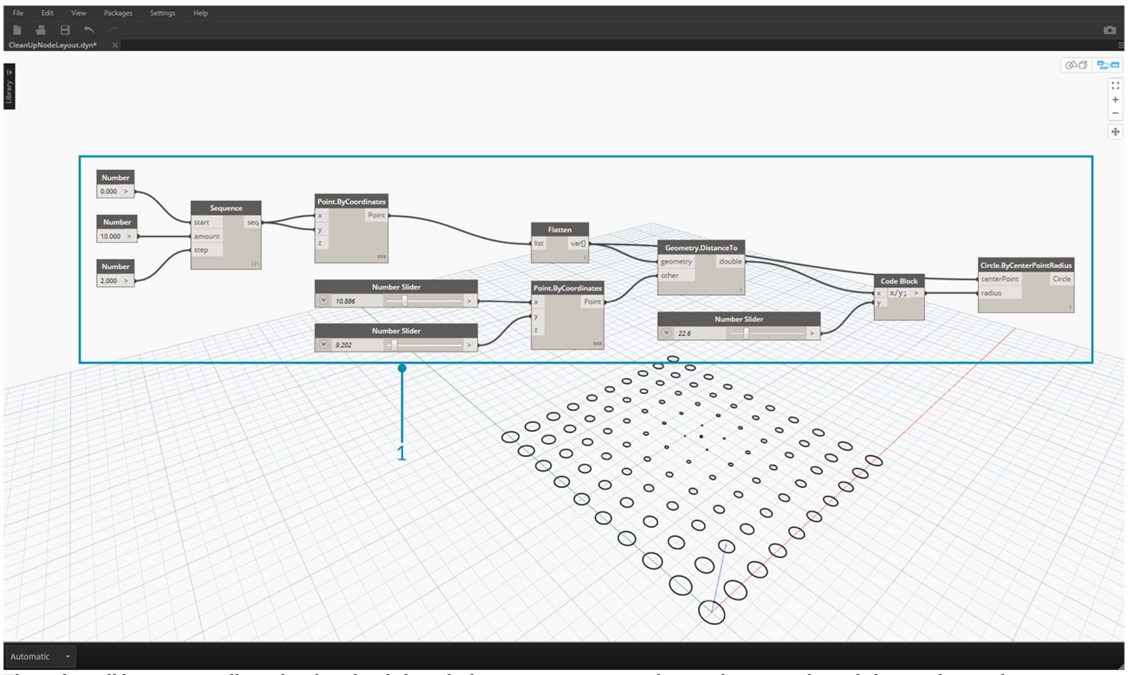
Keeping your Dynamo canvas organized becomes increasingly important as your files build in complexity. Although we have the **Align Selection** tool to work with small amounts of selected Nodes, Dynamo also features the **Cleanup Node Layout** tool to help with overall file cleanup.

Before Node Cleanup



1. Select the Nodes to be automatically organized, or leave all unselected to clean up all nodes in the file.
2. The Cleanup Node Layout feature is located under the Edit tab.

After Node Cleanup



3. The nodes will be automatically re-distributed and aligned, cleaning up any staggered or overlapping nodes and aligning them with neighboring nodes.

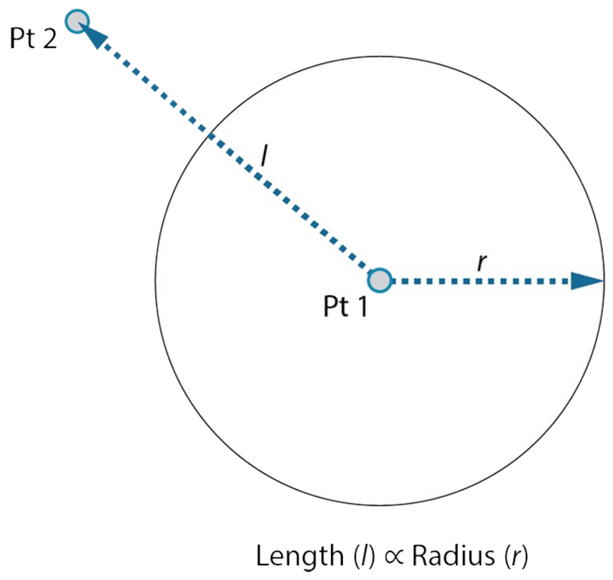
Getting Started

GETTING STARTED

Now that we have familiarized ourselves with the interface layout and navigating the Workspace, our next step is to understand the typical workflow for developing a graph in Dynamo. Let's get started by creating a dynamically sized circle and then create an array of circles with varying radii.

Defining Objectives and Relationships

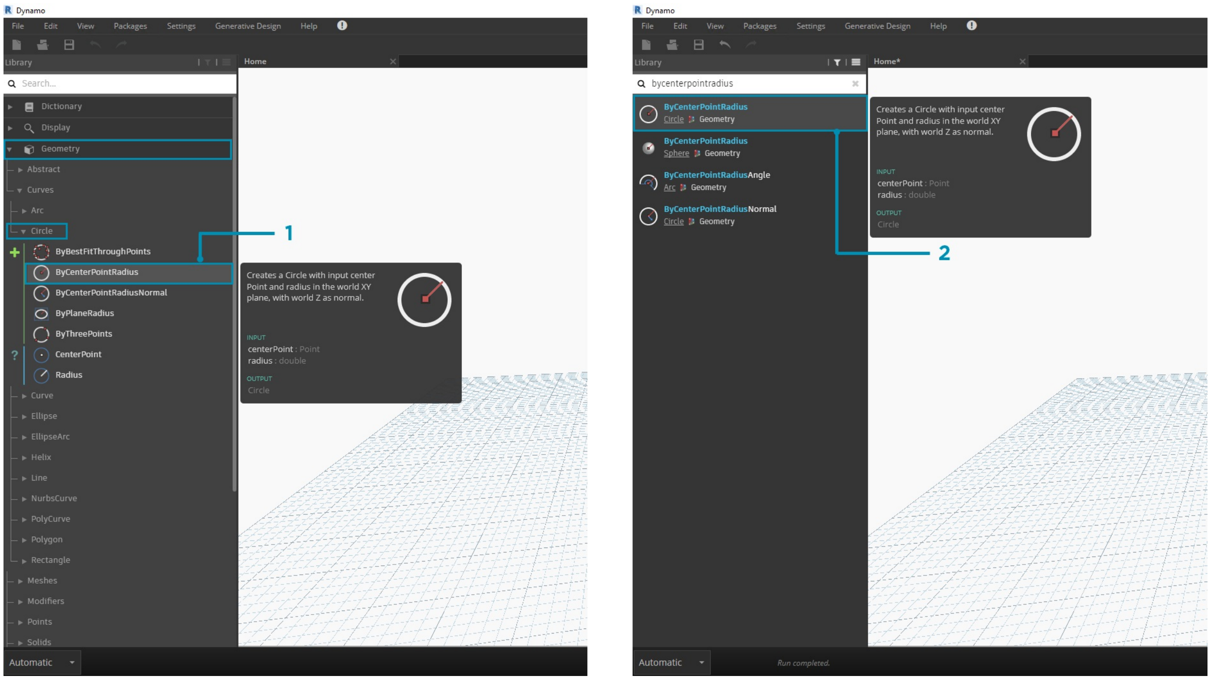
Before we add anything to the Dynamo Workspace, it is key that we have a solid understanding of what we are trying to achieve and what the significant relationships will be. Remember that anytime we are connecting two Nodes, we are creating an explicit link between them - we may change the flow of data later, but once connected we've committed to that relationship. In this exercise we want to create a circle (*Objective*) where the radius input is defined by a distance to a nearby point (*Relationship*).



A point that defines a distance-based relationship is commonly referred to as an "Attractor." Here the distance to our Attractor Point will be used to specify how big our circle should be.

Adding Nodes to the Workspace

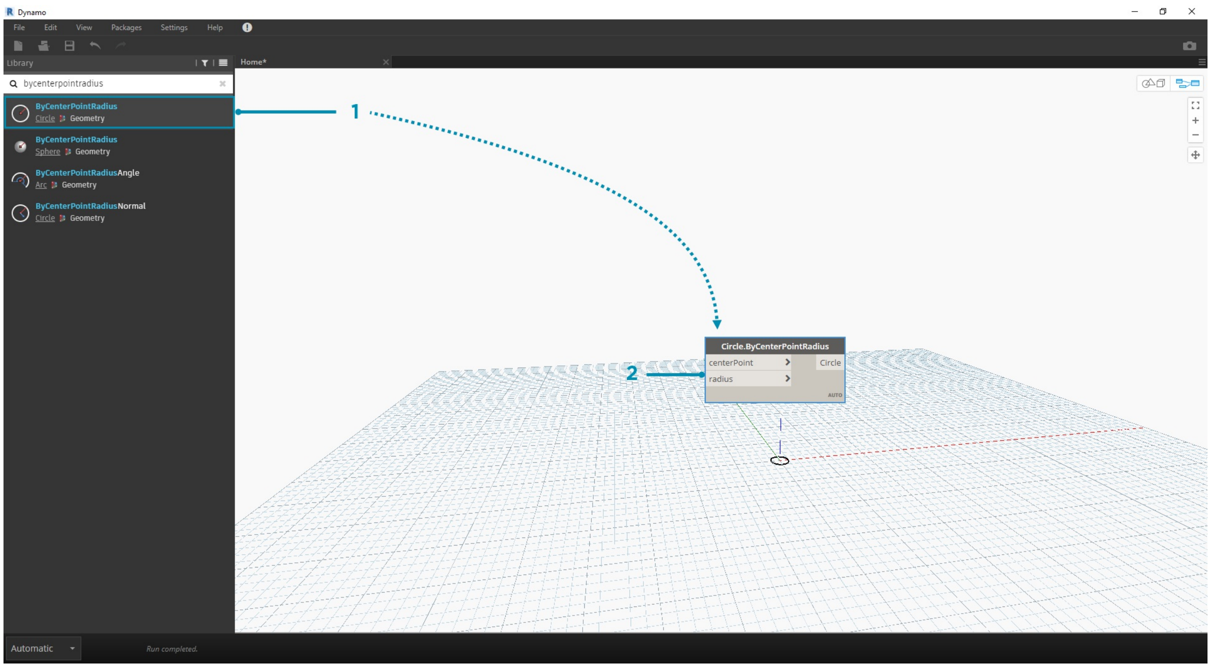
Now that we have our Objectives and Relationships sketched we can begin creating our graph. We need the Nodes that will represent the sequence of actions Dynamo will execute. Since we know we are trying to create a circle, let's start by locating a Node that does so. Using the Search field or browsing through the Library, we will find that there is more than one way to create a circle.



Creates a Circle with input center Point and radius in the world XY plane, with world Z as normal.

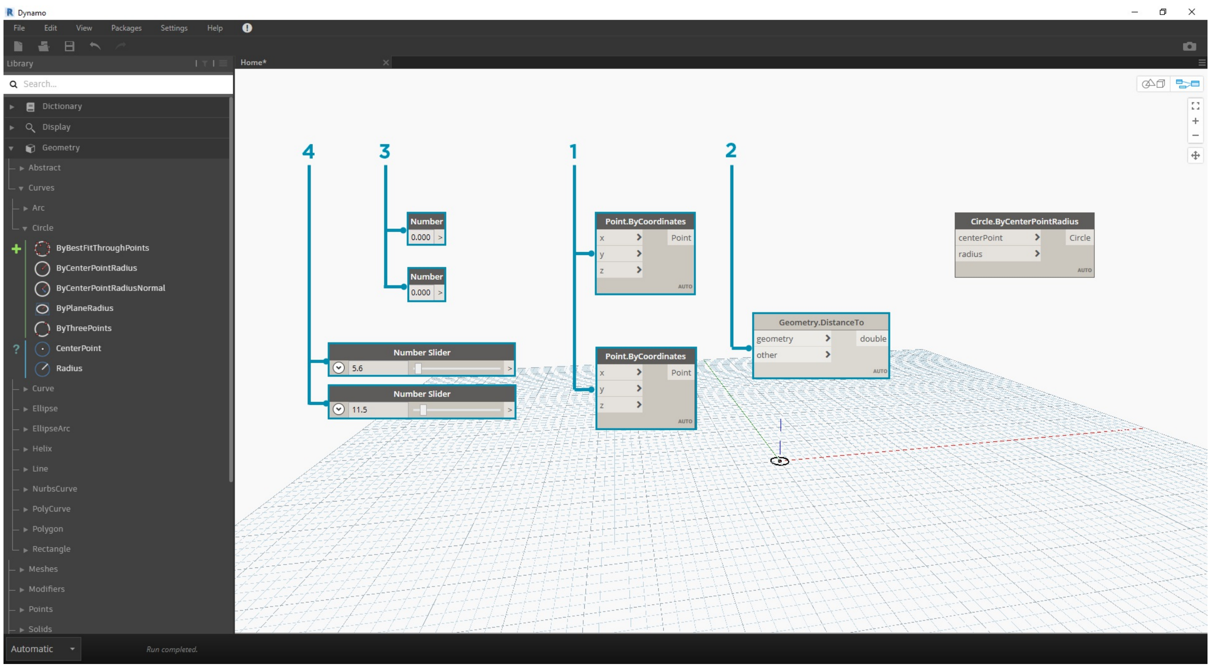
1. Browse to Geometry > Circle > **ByCenterPointRadius**
2. Search > "ByCenterPointRadius..."

Let's add the **Circle.ByCenterPointRadius** Node to the Workspace by clicking on it in the Library - this should add the Node to the center of the Workspace.



1. The Circle.ByCenterPointRadius Node in the Library
2. Clicking the Node in the Library adds it to the Workspace

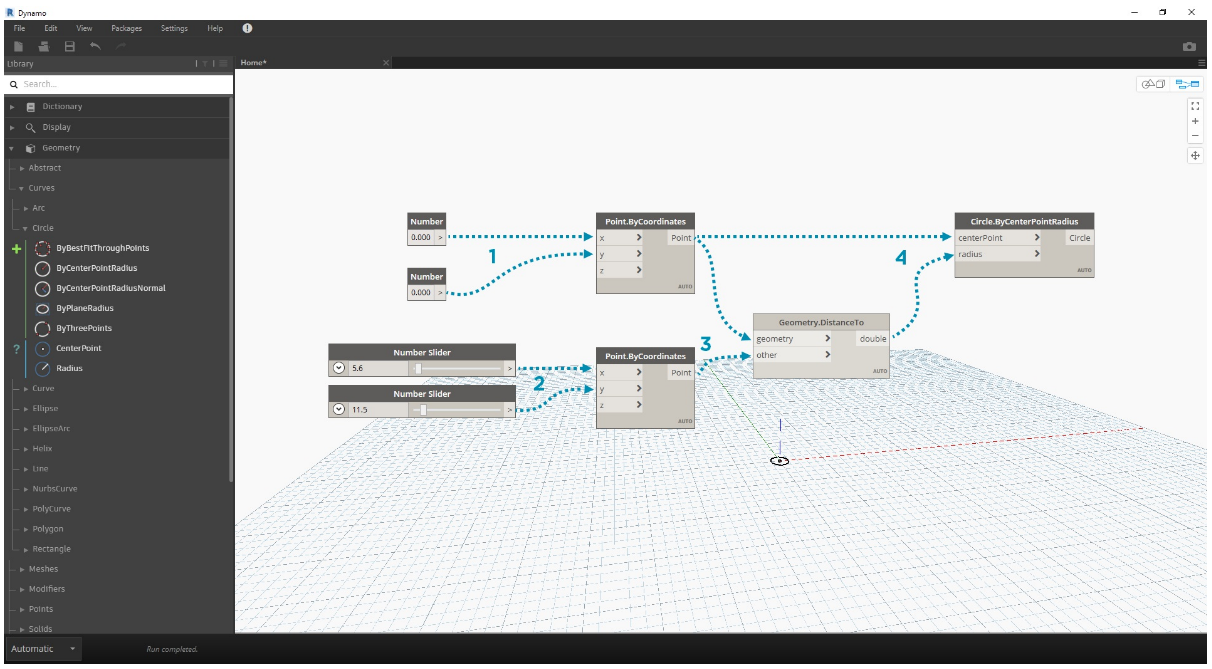
We also will need **Point.ByCoordinates**, **Number Input**, and **Number Slider** Nodes.



1. Geometry > Point > **ByCoordinates (x,y,z)**
2. Geometry > Geometry > **DistanceTo**
3. Core > Input > **Number**
4. Core > Input > **Number Slider**

Connecting Nodes with Wires

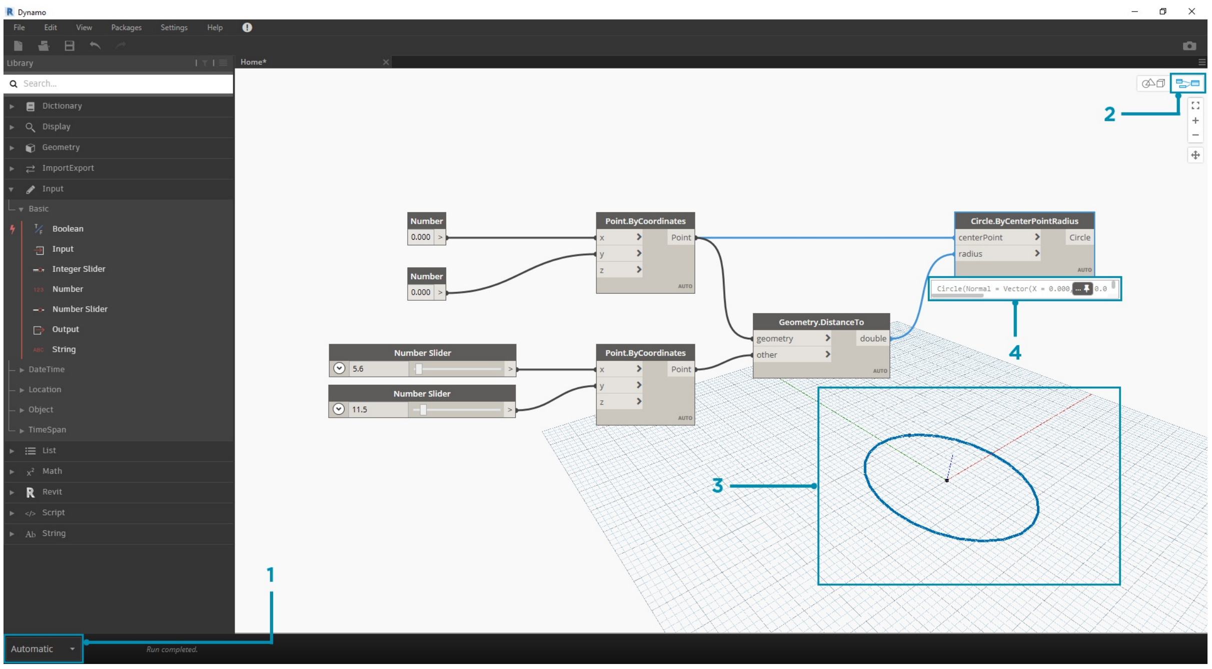
Now that we have a few Nodes, we need to connect the Ports of the Nodes with Wires. These connections will define the flow of data.



1. **Number** to **Point.ByCoordinates**
2. **Number Sliders** to **Point.ByCoordinates**
3. **Point.ByCoordinates** (2) to **DistanceTo**
4. **Point.ByCoordinates** and **DistanceTo** to **Circle.ByCenterPointRadius**

Executing the Program

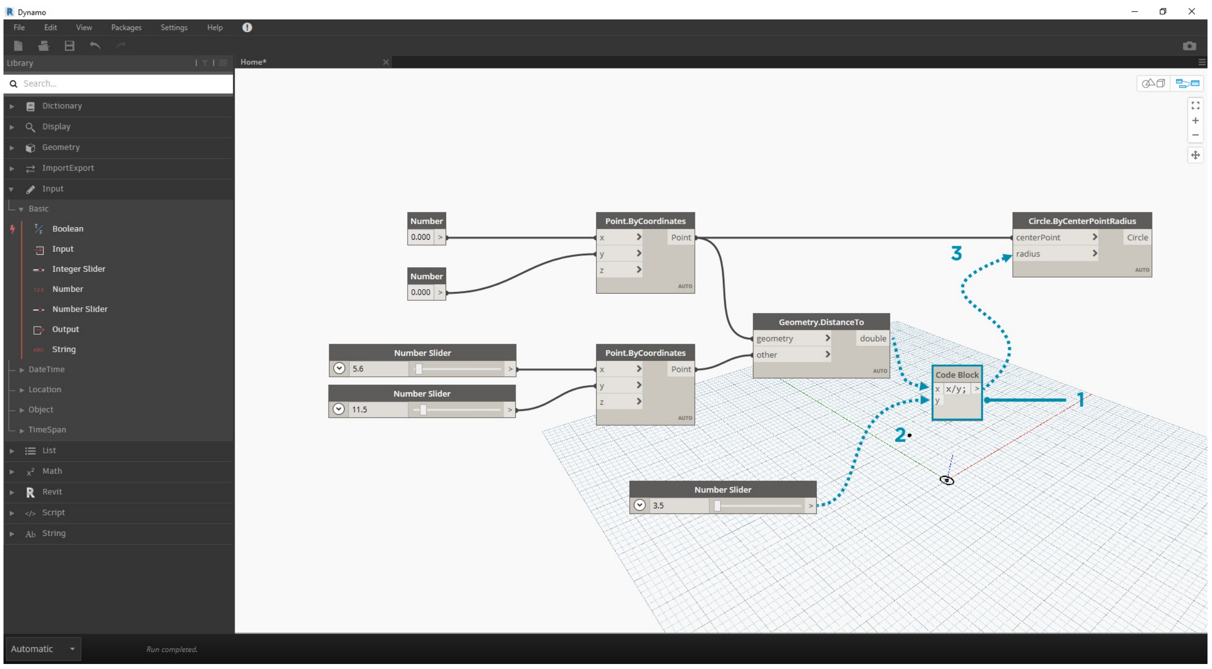
With our Program Flow defined, all we need to do is tell Dynamo to execute it. Once our program is executed (either Automatically or when we click Run in Manual Mode), data will pass through the Wires, and we should see the results in the 3d Preview.



1. (Click Run) - If the Execution Bar is in Manual Mode, we need to Click Run to execute the graph
2. Node Preview - Hovering your mouse over the box on the lower right corner of a Node will give you a pop-up of the results
3. 3D Preview - If any of our Nodes create geometry, we will see it in the 3D Preview.
4. The output geometry on the creation node.

Adding Detail

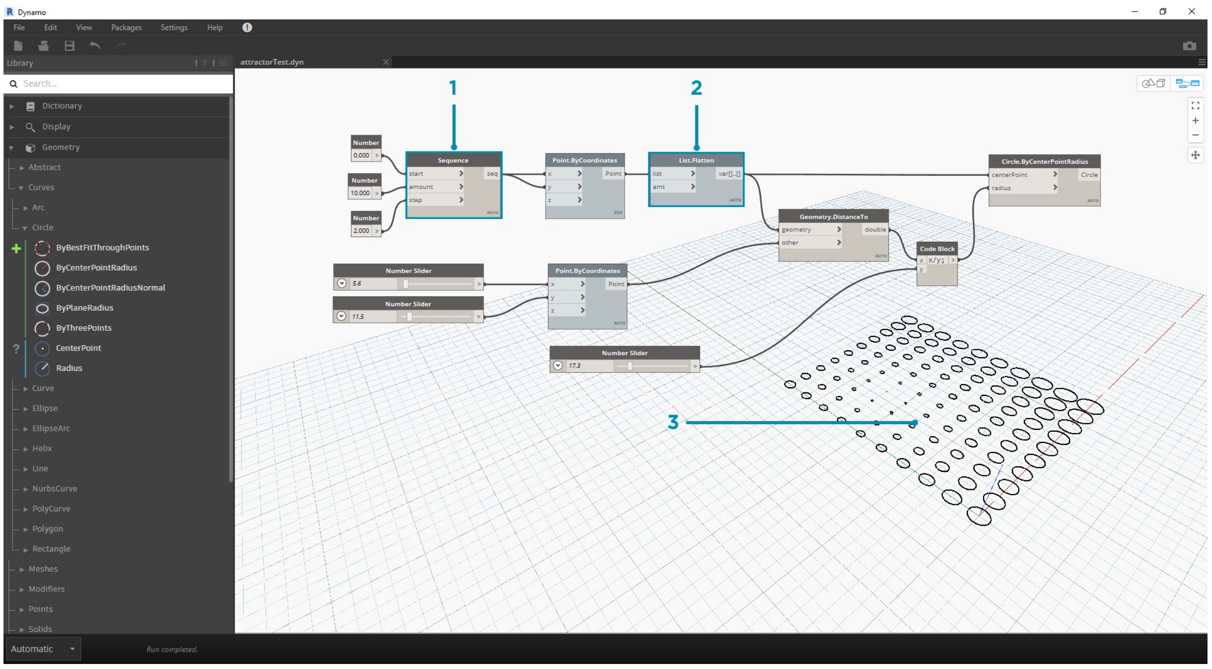
If our program is working, we should see a circle in the 3D Preview that is passing through our Attractor Point. This is great, but we may want to add more detail or more controls. Let's adjust the input to the circle Node so that we can calibrate the influence on the radius. Add another **Number Slider** to the Workspace, then double click on a blank area of the Workspace to add a **Code Block** Node. Edit the field in the Code Block, specifying `X/Y`.



1. **Code Block**
2. **DistanceTo** and **Number Slider** to **Code Block**
3. **Code Block** to **Circle.ByCenterPointRadius**

Adding complexity

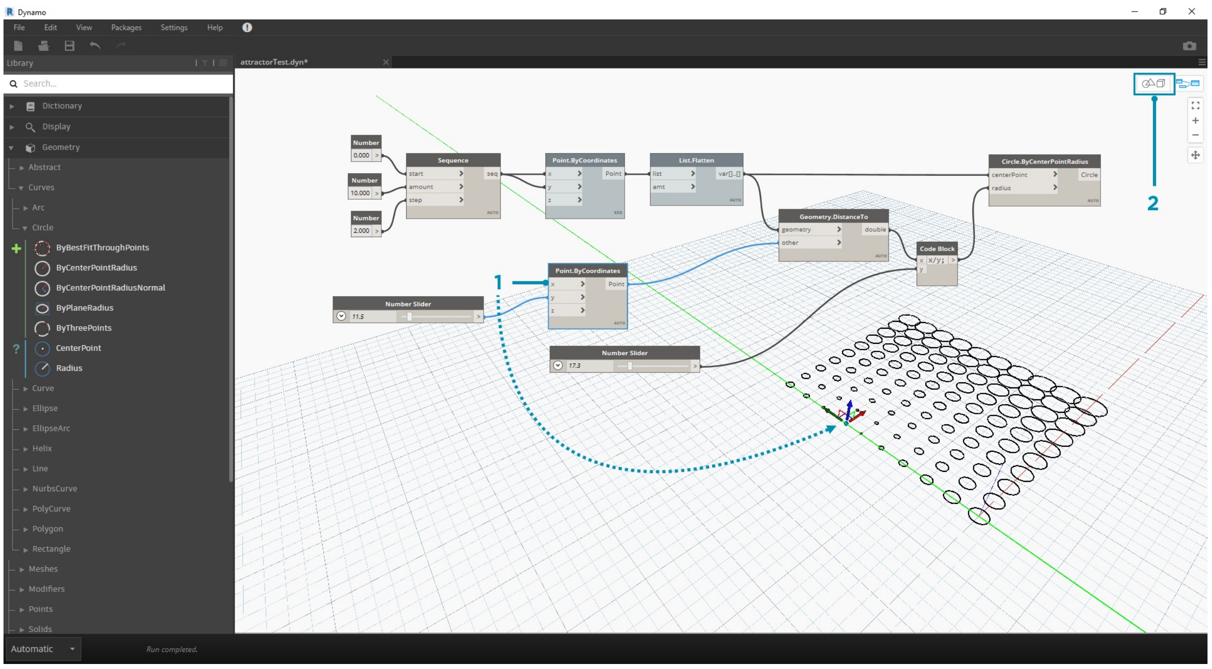
Starting simple and building complexity is an effective way to incrementally develop our program. Once it is working for one circle, let's apply the power of the program to more than one circle. Instead of one center point, if we use a grid of points and accommodate the change in the resulting data structure, our program will now create many circles - each with a unique radius value defined by the calibrated distance to the Attractor Point.



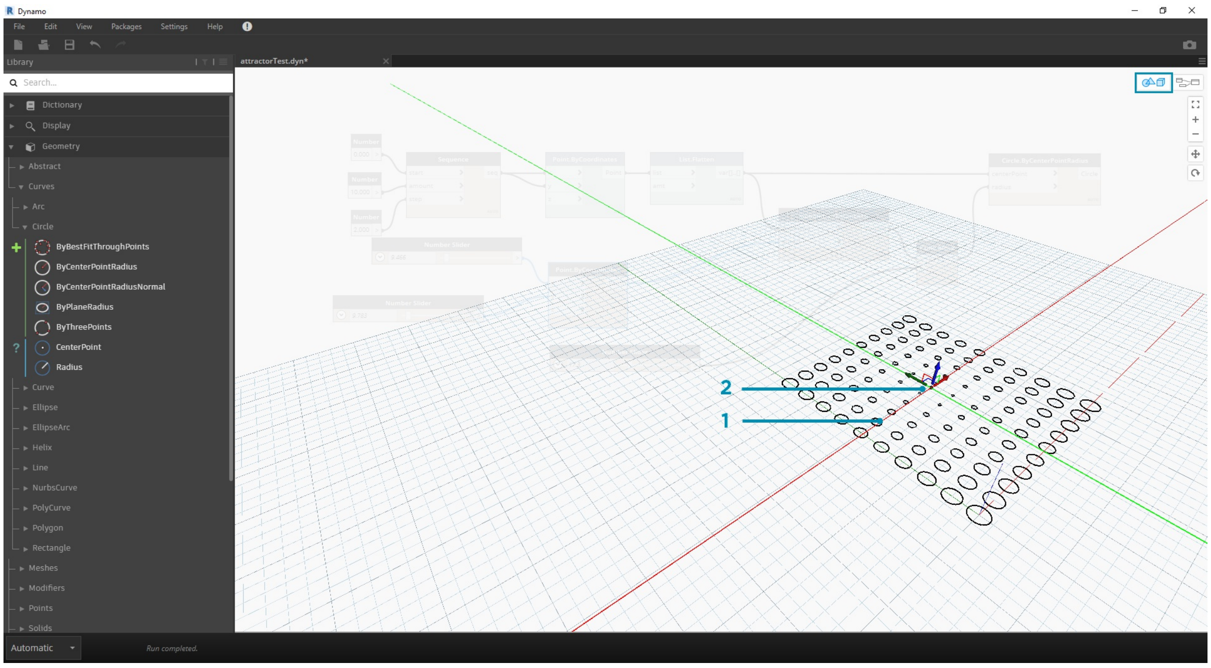
1. Add a **Number Sequence** Node and replace the inputs of **Point.ByCoordinates** - Right Click Point.ByCoordinates and select Lacing > Cross Reference
2. Add a **Flatten** Node after Point.ByCoordinates. To flatten a list completely, leave the `amt` input at the default of `-1`
3. The 3D Preview will update with a grid of circles

Adjusting with Direct Manipulation

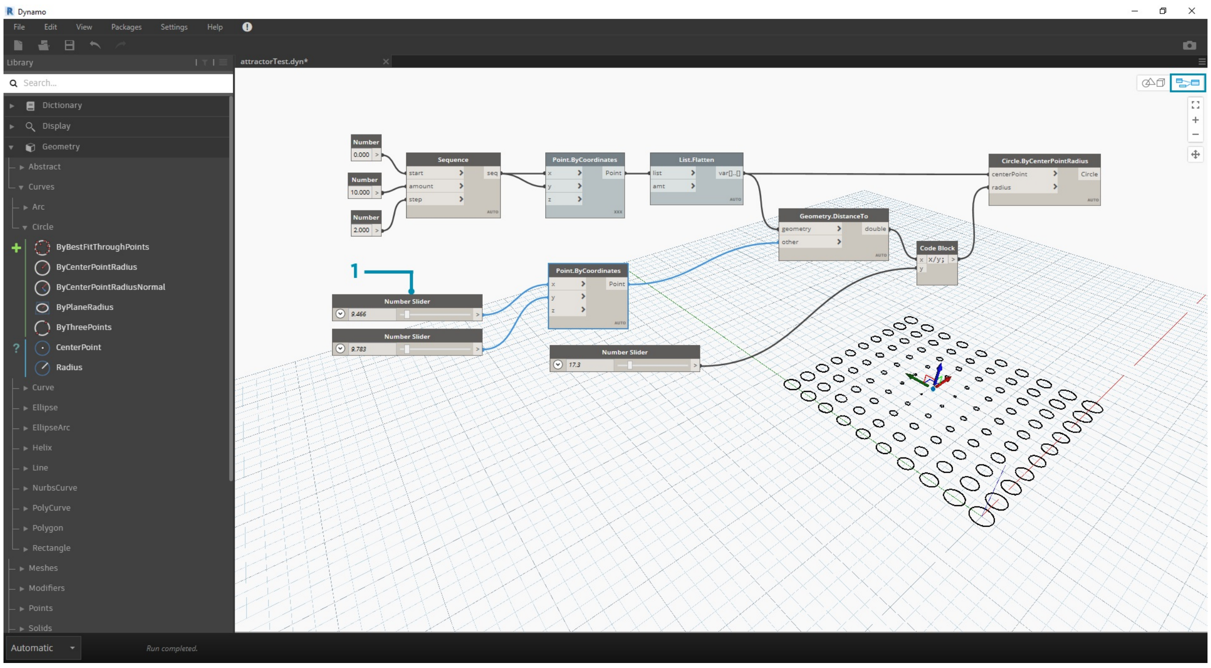
Sometimes numerical manipulation isn't the right approach. Now you can manually push and pull Point geometry when navigating in the background 3D preview. We can also control other geometry that was constructed by a point. For example, **Sphere.ByCenterPointRadius** is capable of Direct Manipulation as well. We can control the location of a point from a series of X, Y, and Z values with **Point.ByCoordinates**. With the Direct Manipulation approach, however, you are able to update the values of the sliders by manually moving the point in the **3D Preview Navigation** mode. This offers a more intuitive approach to controlling a set of discrete values that identify a point's location.



1. To use **Direct Manipulation**, select the panel of the point to be moved – arrows will appear over the point selected.
2. Switch to **3D Preview Navigation** mode.



1. Hover over the point and the X, Y, and Z axes will appear.
2. Click and drag the colored arrow to move the corresponding axis, and the **Number Slider** values will update live with the manually moved point.

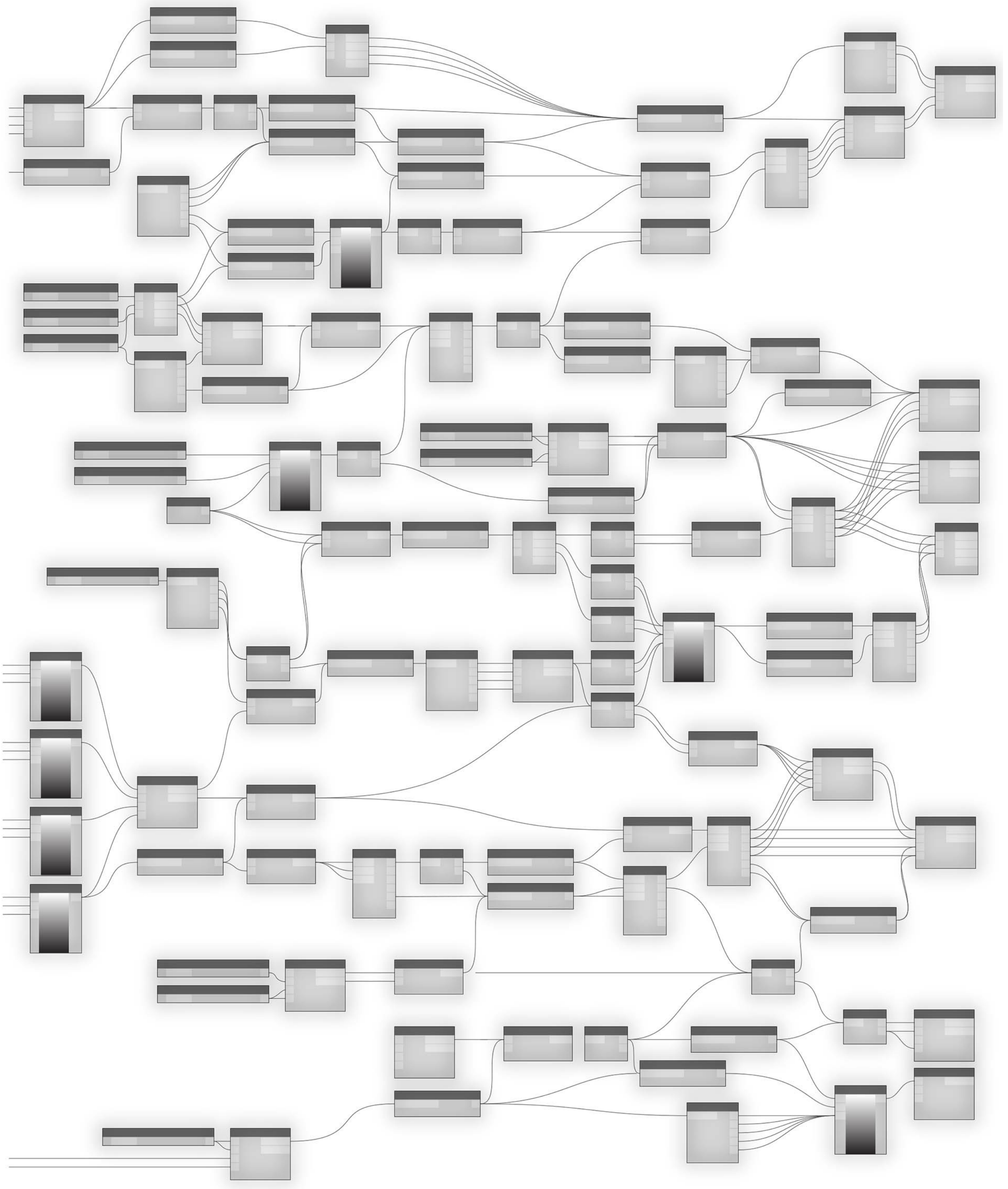


1. Note that before **Direct Manipulation** only one slider was plugged into the **Point.ByCoordinates** component. When we manually move the point in the X-direction, Dynamo will automatically generate a new **Number Slider** for the X input.

The Anatomy of a Visual Program

ANATOMY OF A VISUAL PROGRAM

Dynamo enables us to create Visual Programs in a Workspace by connecting Nodes with Wires to specify the logical flow of the resulting Visual Program. This chapter introduces the elements of Visual Programs, the organization of the Nodes available in Dynamo's Libraries, the parts and states of Nodes, and best practices for your Workspaces.



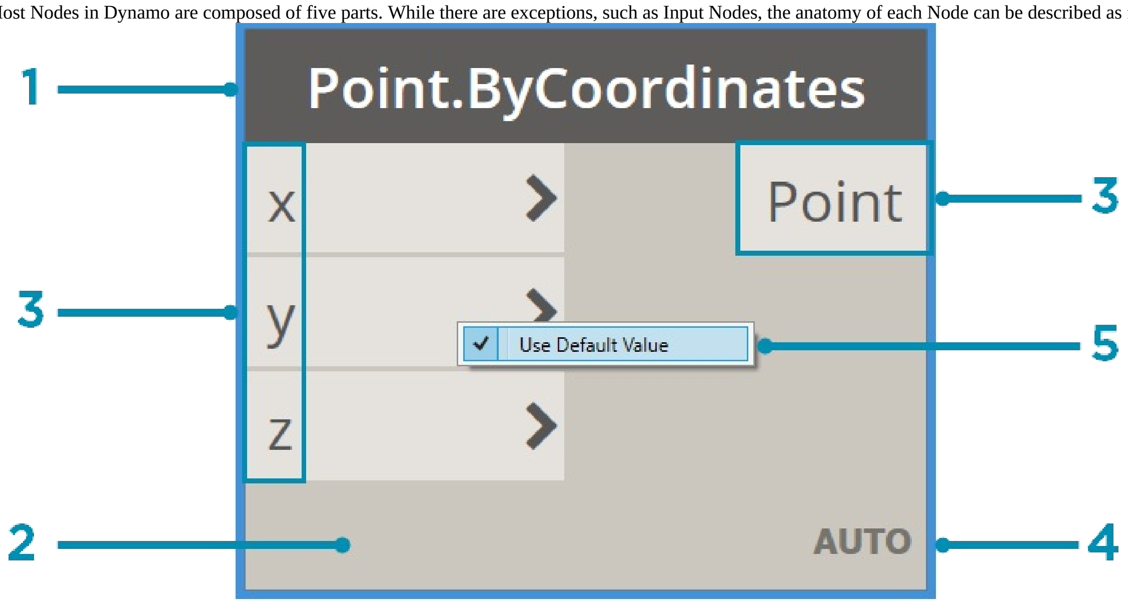
Nodes

Nodes

In Dynamo, **Nodes** are the objects you connect to form a Visual Program. Each **Node** performs an operation - sometimes that may be as simple as storing a number or it may be a more complex action such as creating or querying geometry.

Anatomy of a Node

Most Nodes in Dynamo are composed of five parts. While there are exceptions, such as Input Nodes, the anatomy of each Node can be described as follows:

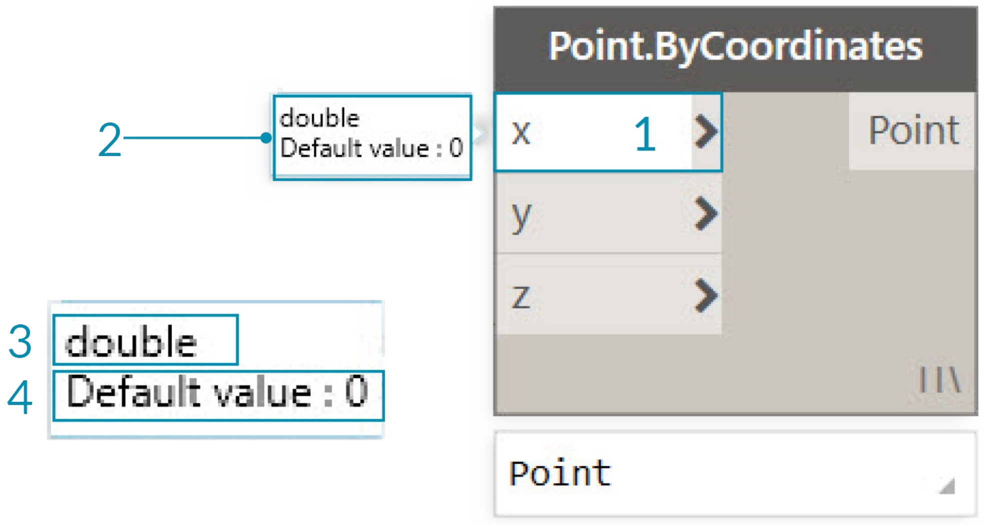


1. Name - The Name of the Node with a Category.Name naming convention
2. Main - The main body of the Node - Right-clicking here presents options at the level of the whole Node
3. Ports (In and Out) - The receptors for Wires that supply the input data to the Node as well as the results of the Node's action
4. Lacing Icon - Indicates the Lacing option specified for matching list inputs (more on that later)
5. Default Value - Right-click on an input Port - some Nodes have default values that can be used or not used.

Ports

The Inputs and Outputs for Nodes are called Ports and act as the receptors for Wires. Data comes into the Node through Ports on the left and flows out of the Node after it has executed its operation on the right. Ports expect to receive data of a certain type. For instance, connecting a number such as *2.75* to the Ports on a Point By Coordinates Node will successfully result in creating a Point; however, if we supply *"Red"* to the same Port it will result in an error.

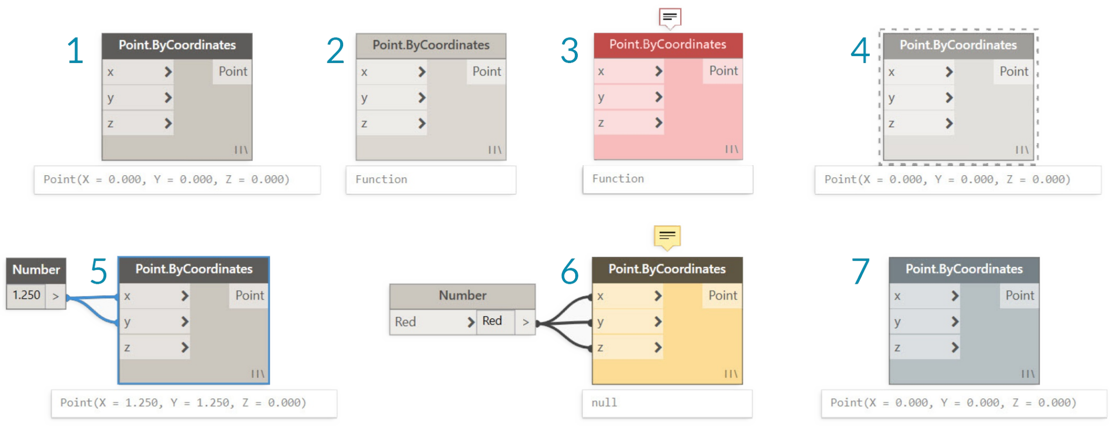
Tip: Hover over a Port to see a tooltip containing the data type expected.



1. Port Label
2. Tool Tip
3. Data Type
4. Default Value

States

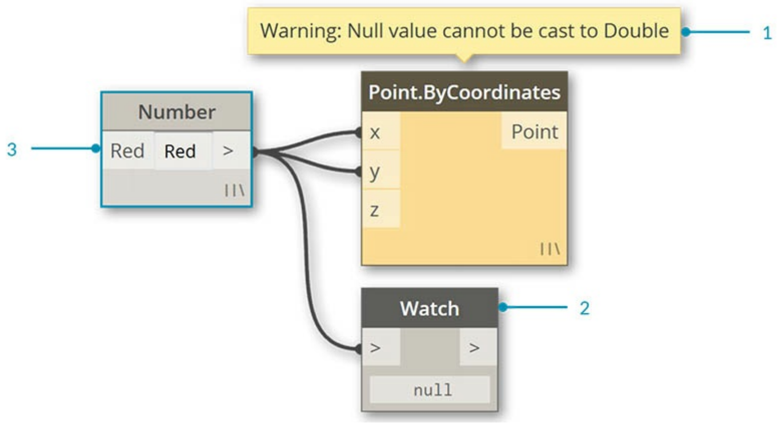
Dynamo gives an indication of the state of the execution of your Visual Program by rendering Nodes with different color schemes based on each Node's status. Furthermore, hovering or right-clicking over the Name or Ports presents additional information and options.



1. Active - Nodes with a Dark Grey Name background are well-connected and have all of their inputs successfully connected
2. Inactive - Grey Nodes are inactive and need to be connected with Wires to be part of the Program Flow in the active Workspace
3. Error State - Red indicates that the Node is in an Error State
4. Freeze - A Transparent node has Freeze turned on, suspending the execution of the node
5. Selected - Currently selected Nodes have an Aqua highlight on their border
6. Warning - Yellow Nodes are in an Warning state, meaning they may have incorrect data types
7. Background Preview - Dark Grey indicates that the geometry preview is turned off

If your Visual Program contains warning or errors, Dynamo will provide additional information about the problem. Any Node that is Yellow will also have a tooltip above the Name. Hover your mouse over the tooltip to expand it.

Tip: With this tooltip information in hand, examine the upstream Nodes to see if the data type or data structure required is in error.



1. Warning Tooltip - "Null" or no data cannot be understood as a Double ie. a number
2. Use the Watch Node to examine the input data
3. Upstream the Number Node is storing "Red" not a number

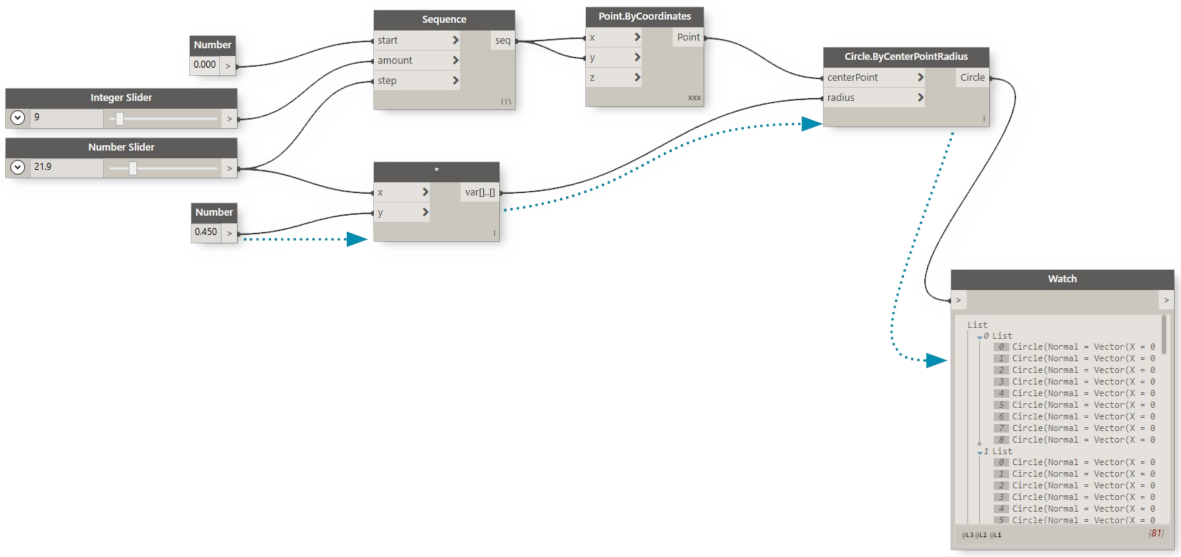
Wires

Wires

Wires connect between Nodes to create relationships and establish the Flow of our Visual Program. We can think of them literally as electrical wires that carry pulses of data from one object to the next.

Program Flow

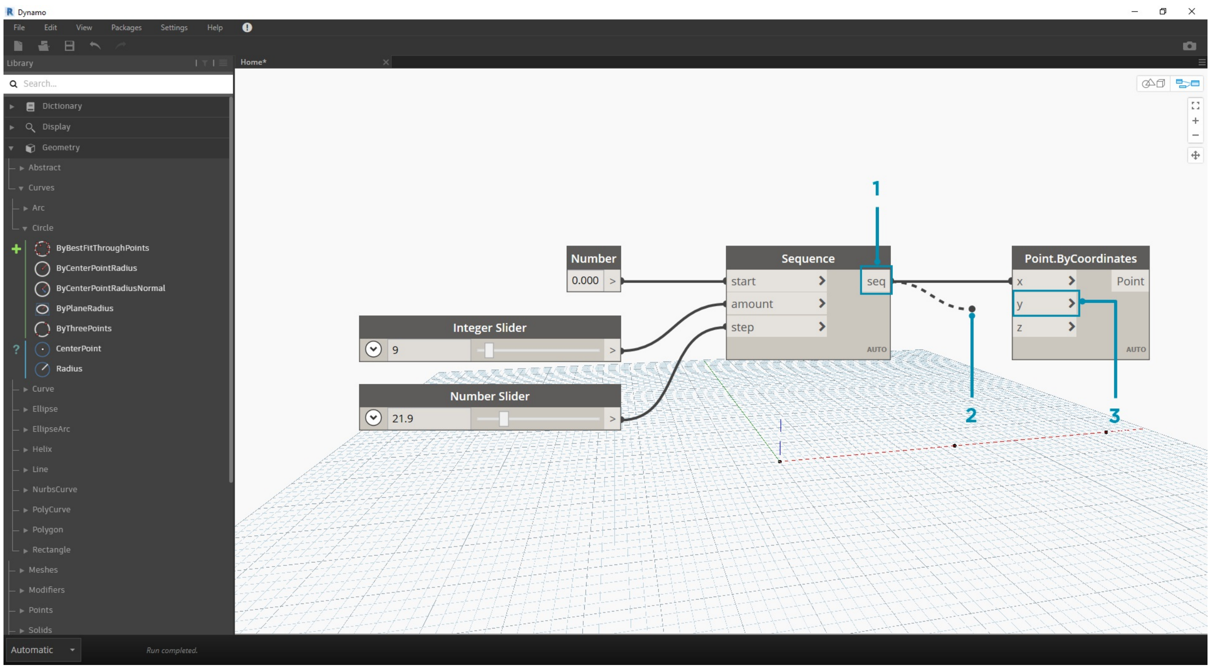
Wires connect the output Port from one Node to the input Port of another Node. This directionality establishes the **Flow of Data** in the Visual Program. Although we can arrange our Nodes however we desire in the Workspace, because the output Ports are located on the right side of Nodes and the input Ports are on the left side, we can generally say that the Program Flow moves from left to right.



Creating Wires

We create a Wire by left clicking our mouse on a Port and then left clicking on the port of another Node to create a connection. While we are in the process of making a connection, the Wire will appear dashed and will snap to become solid lines when successfully connected. The data will always flow through this Wire from output to input; however, we may create the wire in either direction in terms of the sequence of clicking on the connected Ports.

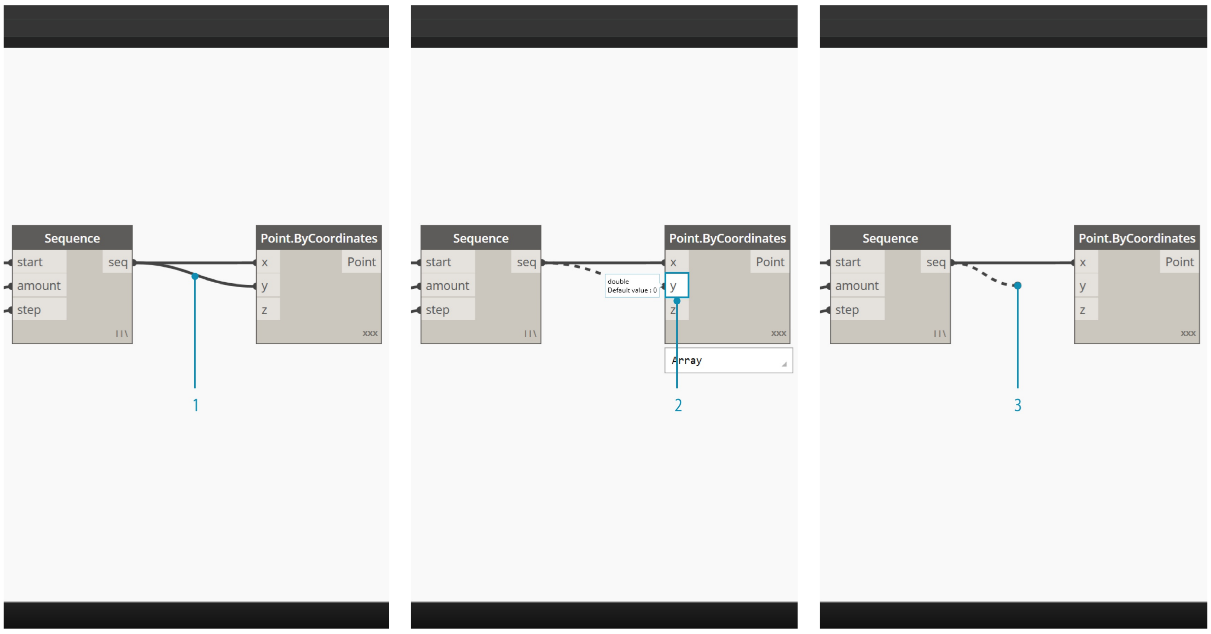
Tip: Before completing the connection with your second click, allow the Wire snap to a Port and hover your mouse there to see the Port tooltip.



1. Click on the `seq` output Port of the Number Sequence Node
2. As you are moving your mouse towards another Port, the Wire is dashed
3. Click on the `y` input Port of the Point.ByCoordinates to complete the connection

Editing Wires

Frequently we will want to adjust the Program Flow in our Visual Program by editing the connections represented by the Wires. To edit a Wire, left click on the input Port of the Node that is already connected. You now have two options:

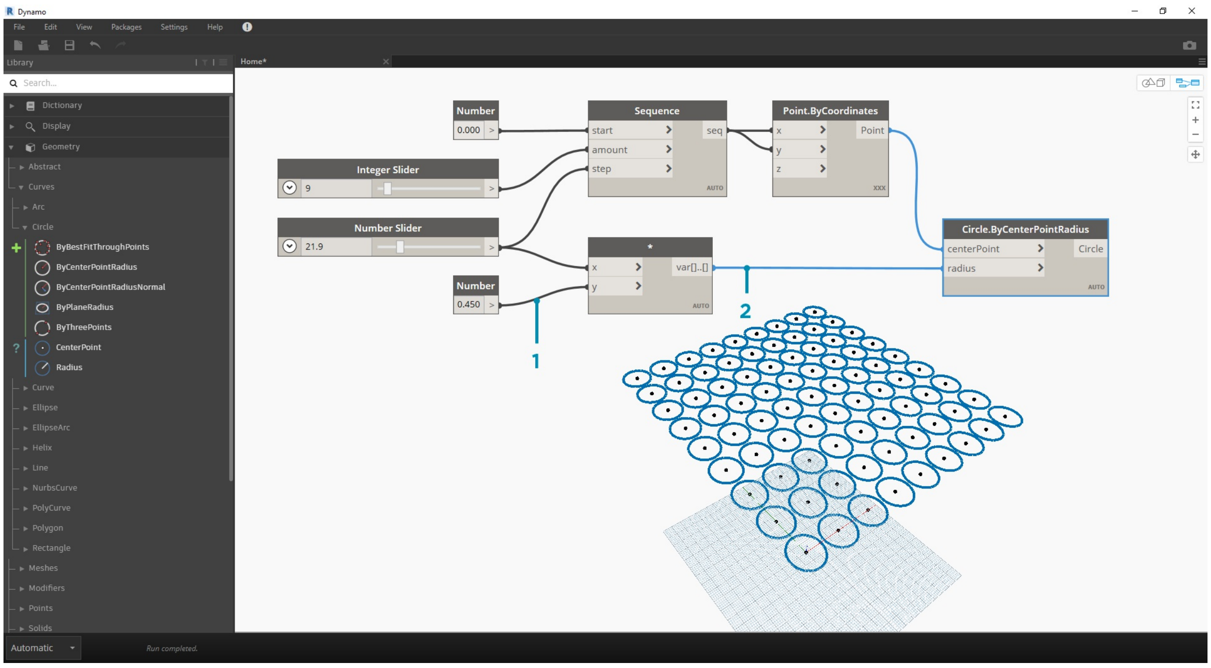


1. Existing Wire
2. To change the connection to an input Port, left click on another input Port
3. To remove the Wire, pull the Wire away and left click on the Workspace

*Note- There is additionaly functionality for moving multiple wires at once now. This is covered here http://dynamobim.org/dynamo-1-3-release/

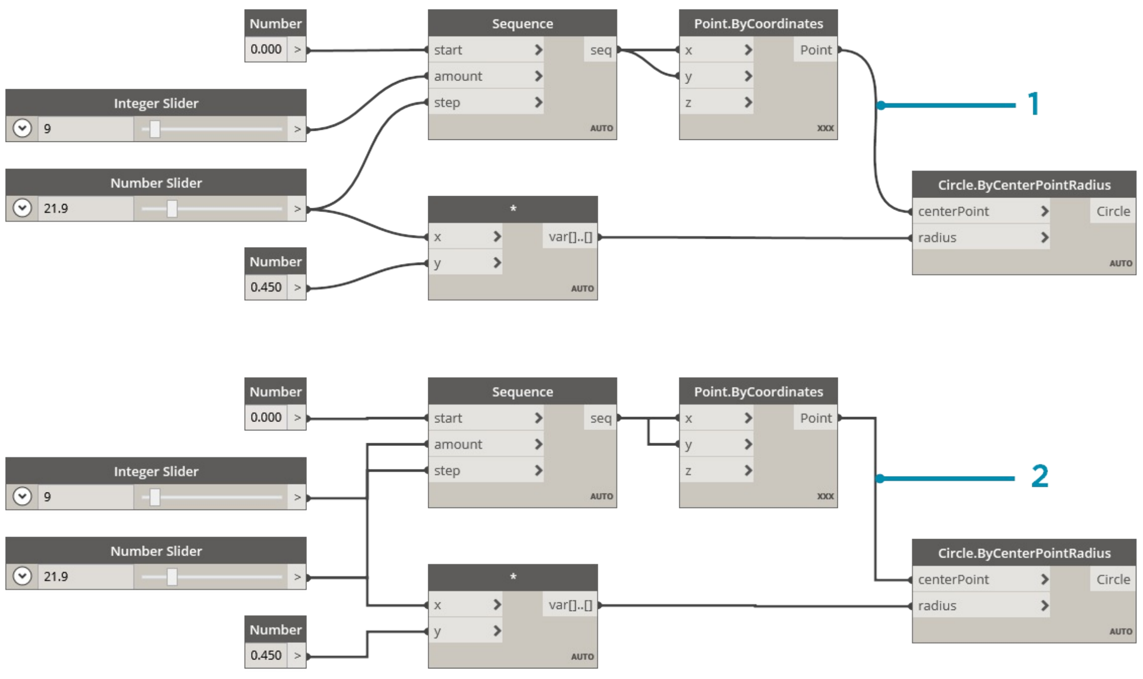
Wire Previews

By default, our Wires will be previewed with a gray stroke. When a Node is selected, it will render any connecting Wire with the same aqua highlight as the Node.



1. Default Wire
2. Highlighted Wire

Dynamo also allows us to customize how our Wires look in the Workspace through the View > Connectors Menu. Here we can toggle between Curve or Polyline Wires or turn them off all together.



1. Connector Type: Curves
2. Connector Type: Polylines

Library

Dynamo Library

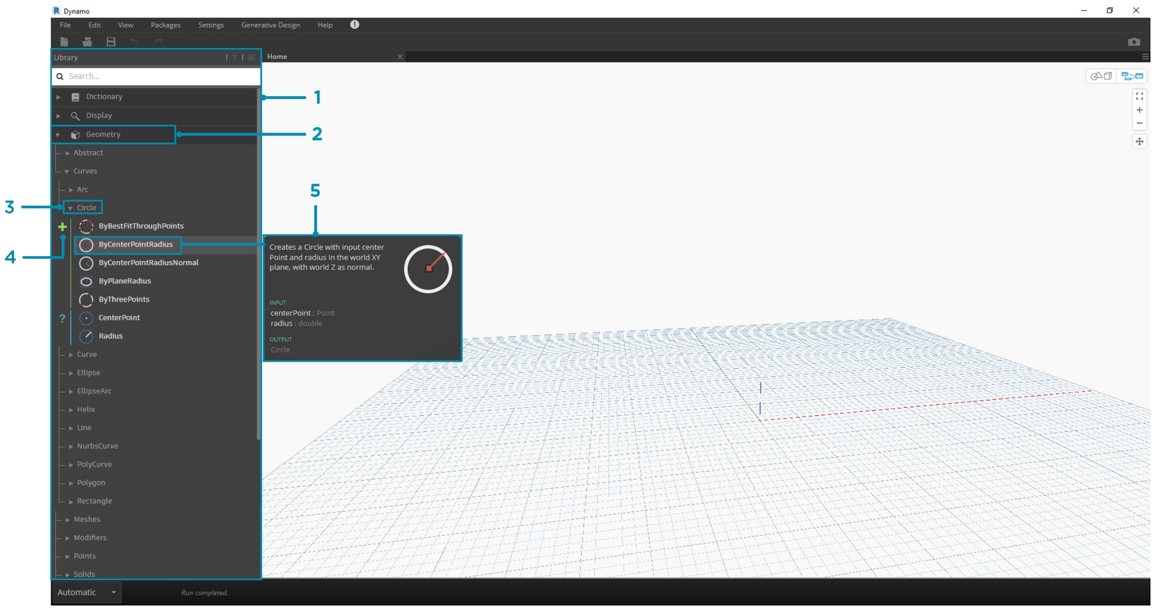
The **Dynamo Library** contains the Nodes we add to the Workspace to define Visual Programs for execution. In the Library, we can search for or browse to Nodes. The Nodes contained here - the basic Nodes installed, Custom Nodes we define, and Nodes from the Package Manager that we add to Dynamo - are organized hierachically by category. Let's review this organization and explore the key Nodes we will use frequently.

Library of Libraries

The Dynamo **Library** that we interface with in the application is actually a collection of functional libraries, each containing Nodes grouped by Category. While this may seem obtuse at first, it is a flexible framework for organizing the Nodes that come with the default installation of Dynamo - and it's even better down the road when we start extending this base functionality with Custom Nodes and additional Packages.

The Organizational Scheme

The **Library** section of the Dynamo UI is composed of hierarchically organized libraries. As we drill down into the Library, we are sequentially browsing a library, the library's categories, and the category's sub-categories to find the Node.



Creates a Circle with input center Point and radius in the world XY plane, with world Z as normal.

INPUT
centerPoint : Point
radius : double

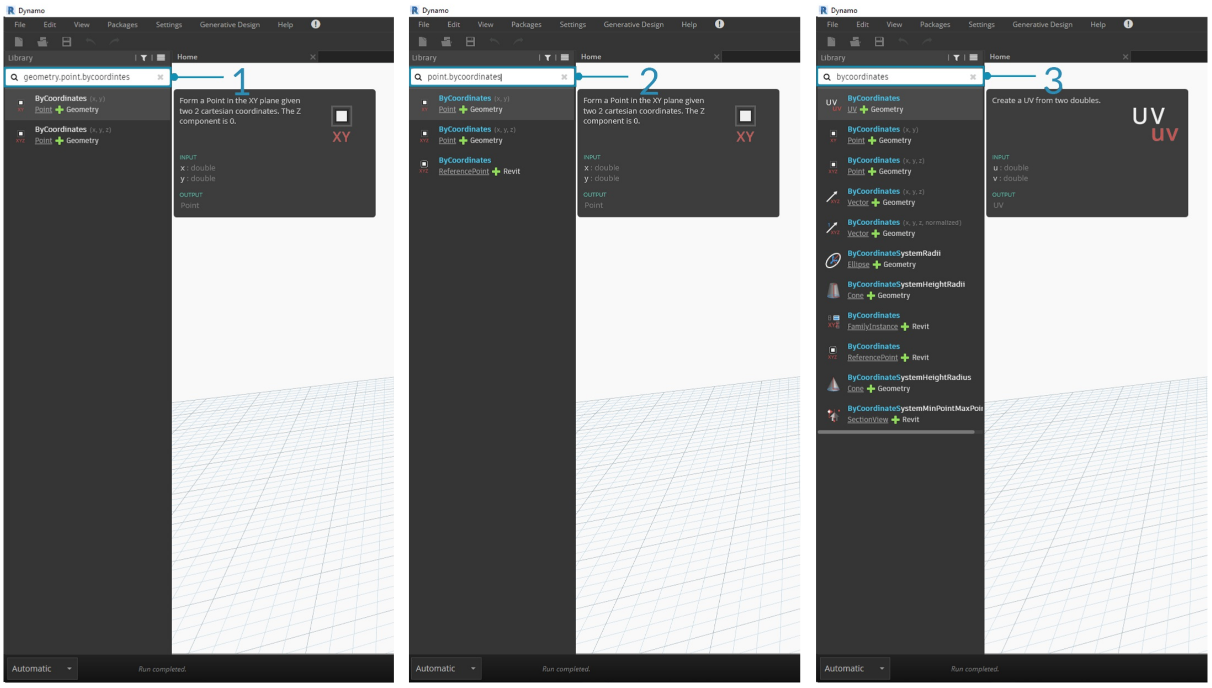
OUTPUT
Circle

1. The Library - The region of the Dynamo Interface
2. A Library - A collection of related Categories, such as **Geometry**
3. A Category - A collection of related Nodes such as everything related to **Circles**
4. A Subcategory - Breakdown of the Nodes within the Category, typically by **Create**, **Action**, or **Query**
5. A Node - The objects that are added to the Workspace to perform an action

Naming Conventions

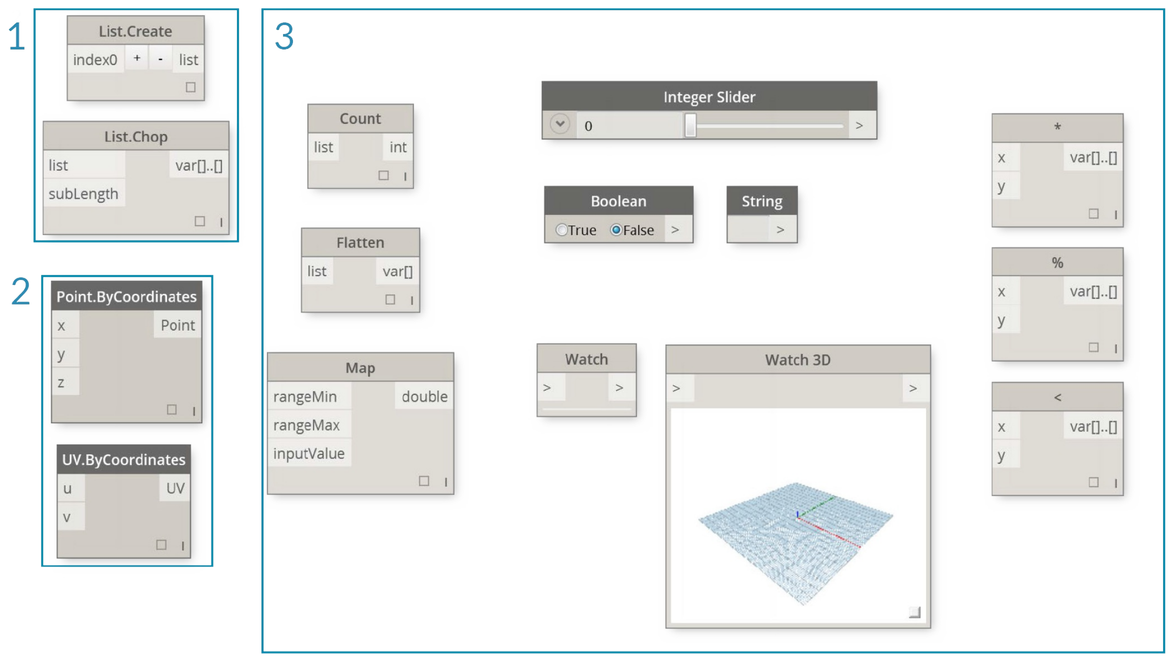
The hierarchy of each library is reflected in the Name of Nodes added to the Workspace, which we can also use in the Search Field or with Code Blocks (which use the *Dynamo textual language*). Beyond using key words to try to find Nodes, we can type the hierarchy separated with a period.

Typing in different portions of the Node's place in the Library hierarchy in the `library.category.nodeName` format returns different results:


1
2
3
geometry.point.bycoordintes
point.bycoordinates
bycoordinates
ByCoordinates
Form a Point in the XY plane given two 2 cartesian coordinates. The Z component is 0.
Create a UV from two doubles.
INPUT
x : double
y : double
u : double
v : double
OUTPUT
Point
UV
Automatic
Run completed.

1. `library.category.nodeName`
2. `category.nodeName`
3. `nodeName` or `keyword`

Typically the Name of the Node in the Workspace will be rendered in the `category.nodeName` format, with some notable exceptions particularly in the Input and View Categories. Beware of similarly named Nodes and note the category difference:



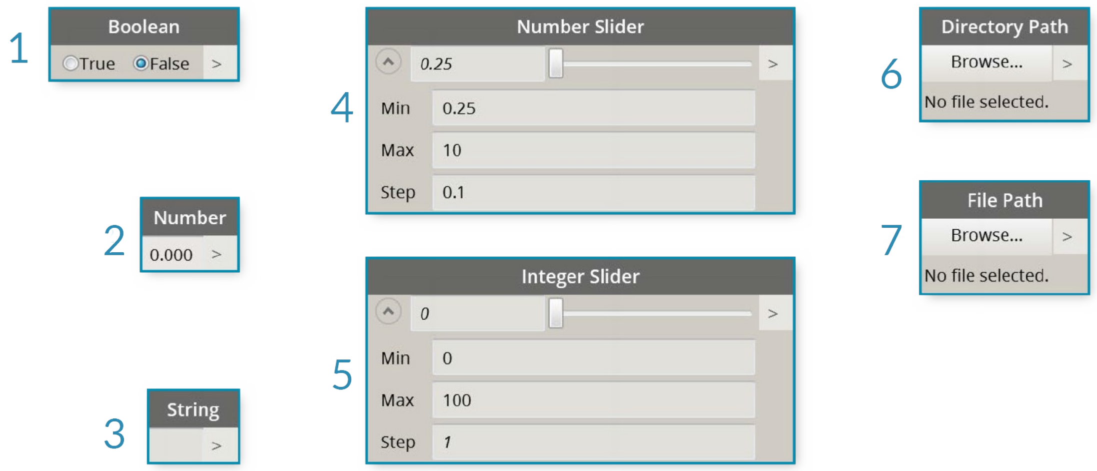
1. Nodes from most libraries will include the category format
2. `Point.ByCoordinates` and `UV.ByCoordinates` have the same Name but come from different categories
3. Notable exceptions include Built-in Functions, Core.Input, Core.View, and Operators

Frequently Used Nodes

With hundreds of Nodes included in the basic installation of Dynamo, which ones are essential for developing our Visual Programs? Let's focus on those that let us define our program's parameters (**Input**), see the results of a Node's action (**Watch**), and define inputs or functionality by way of a shortcut (**Code Block**).

Input

Input Nodes are the primary means for the User of our Visual Program - be that yourself or someone else - to interface with the key parameters. Here are the Nodes available in the Input Category of the Core Library:

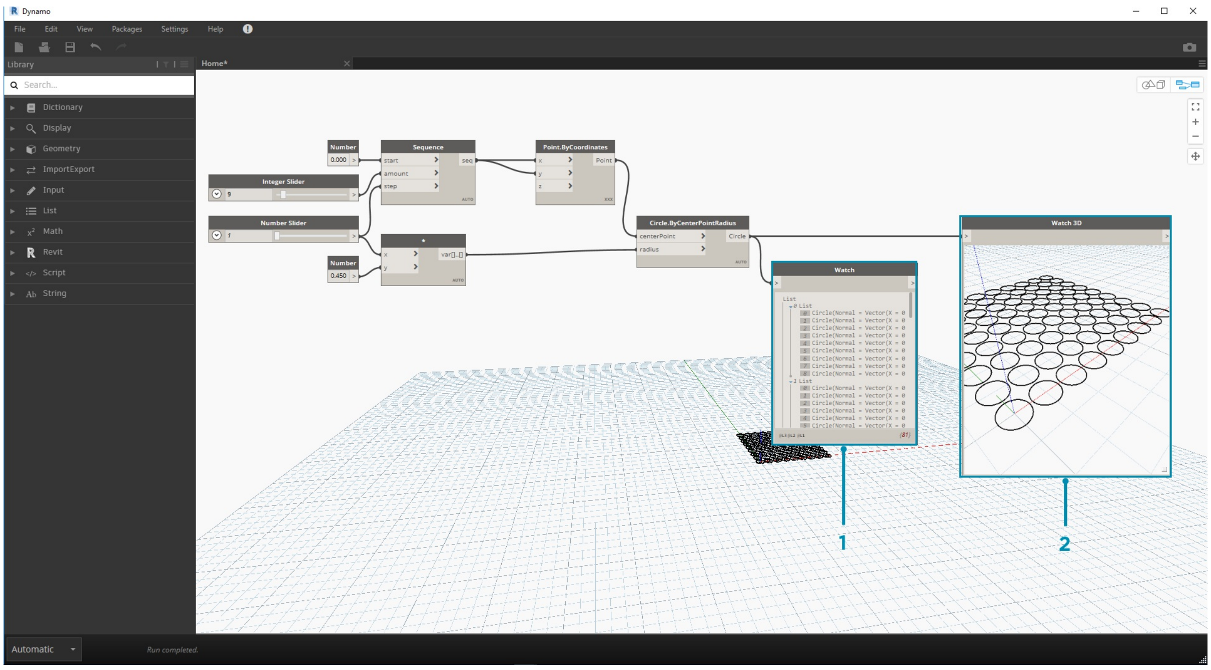


1. Boolean
2. Number
3. String
4. Number Slider
5. Integer Slider
6. Directory Path
7. File Path

Watch

The Watch Nodes are essential to managing the data that is flowing through your Visual Program. While you can view the result of a Node through the Node data preview, you may want to keep it revealed in a **Watch** Node or see the geometry results through a **Watch3D** Node. Both of these are found in the View Category in the Core Library.

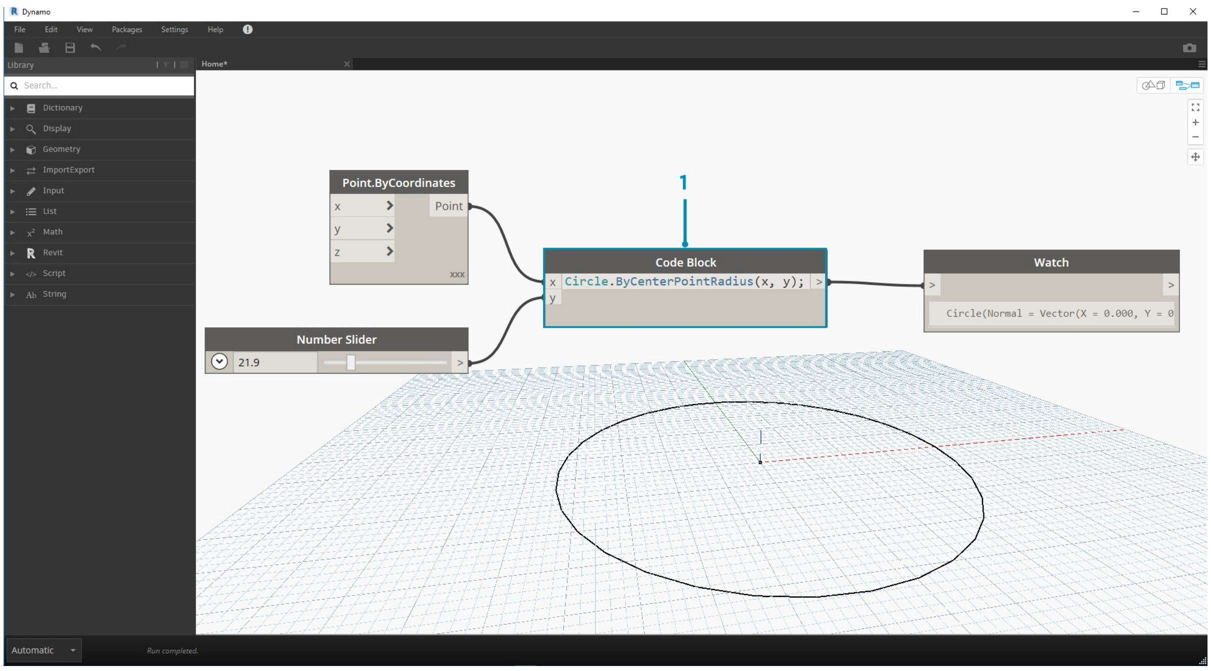
> Tip: Occasionally the 3D Preview can be distracting when your Visual Program contains a lot of Nodes. Consider unchecking the Showing Background Preview option in the Settings Menu and using a Watch3D Node to preview your geometry.



1. Watch - Note that when you select an item in the Watch Node it will be tagged in the Watch3D and 3D Previews
2. Watch3D - Grab the bottom right grip to resize and navigate with you mouse the same way you would in the 3D Preview

Code Block

Code Block Nodes can be used to define a block of code with lines separated by semi-colons. This can be as simple as `X/Y`. We can also use Code Blocks as a short cut to defining a Number Input or call to another Node's functionality. The syntax to do so follows the Naming Convention of the Dynamo textual language, DesignScript, and is covered in Section 7.2. Let's try to make a Circle with this shortcut:



```
Circle.ByCenterPointRadius(x, y);
```

```
Circle(Normal = Vector(X = 0.000, Y = 0
```

1. Double Click to create a **Code Block** Node
2. Type `Circle.ByCenterPointRadius(x,y);`
3. Clicking on the Workspace to clear the selection should add `x` and `y` inputs automatically
4. Create a **Point.ByCoordinates** Node and a **Number Slider** then connect them to the inputs of the Code Block
5. The result of executing the Visual Program should be a circle in the 3D Preview

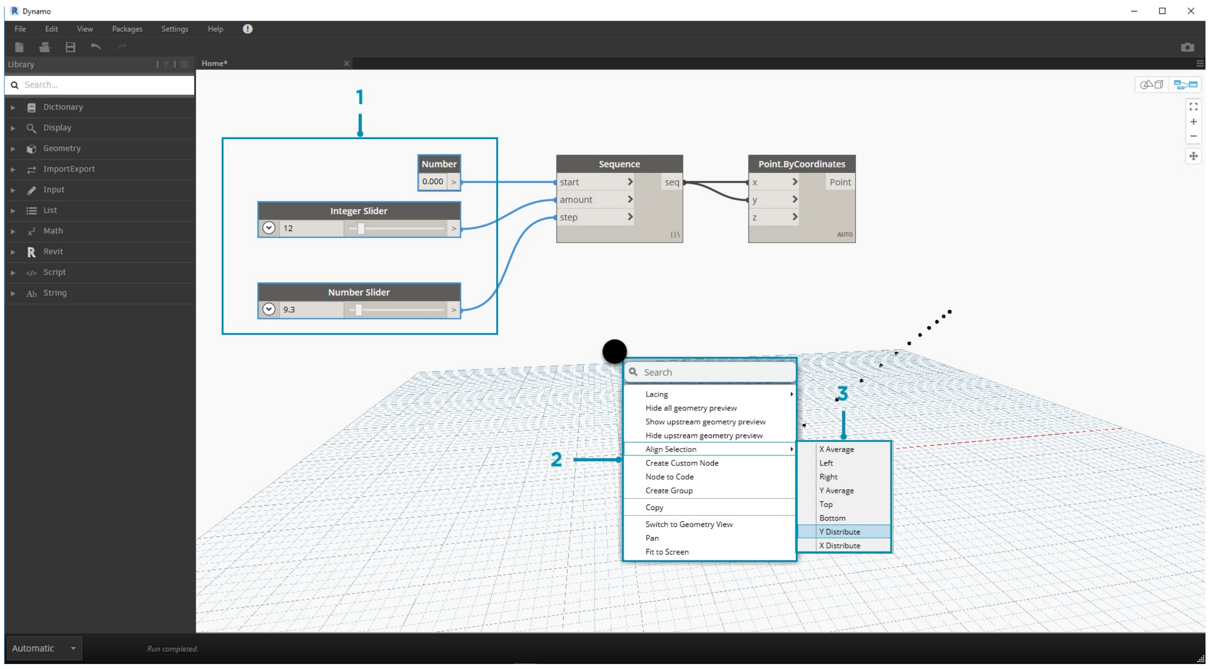
Managing Your Program

Managing Your Program

Working within a Visual Programming process can be a powerful creative activity, but very quickly the Program Flow and key user inputs can be obscured by complexity and/or layout of the Workspace. Let's review some best practices for managing your program.

Alignment

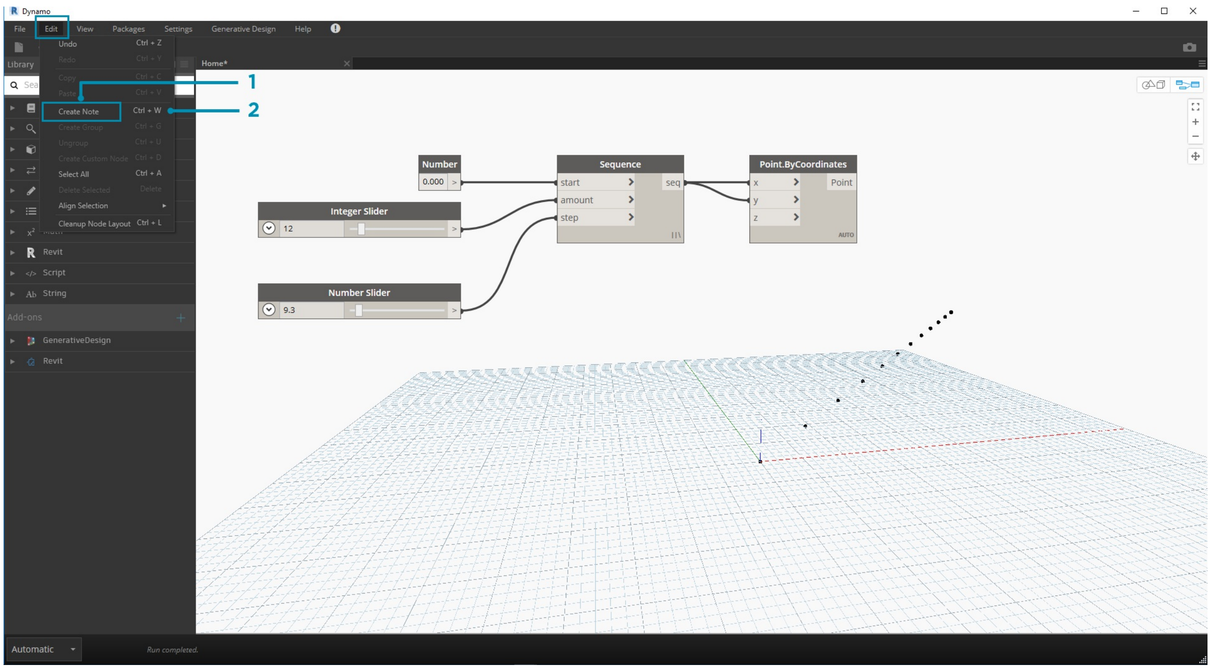
Once we have added more than a few Nodes to the Workspace, we may want to re-organize the layout of the Nodes for clarity's sake. By selecting more than one Node and right-clicking on the Workspace, the pop up window includes an **Align Selection** menu with justification and distribution options in X and Y.



1. Select more than one Node
2. Right-click on the Workspace
3. Use the **Align Selection** options

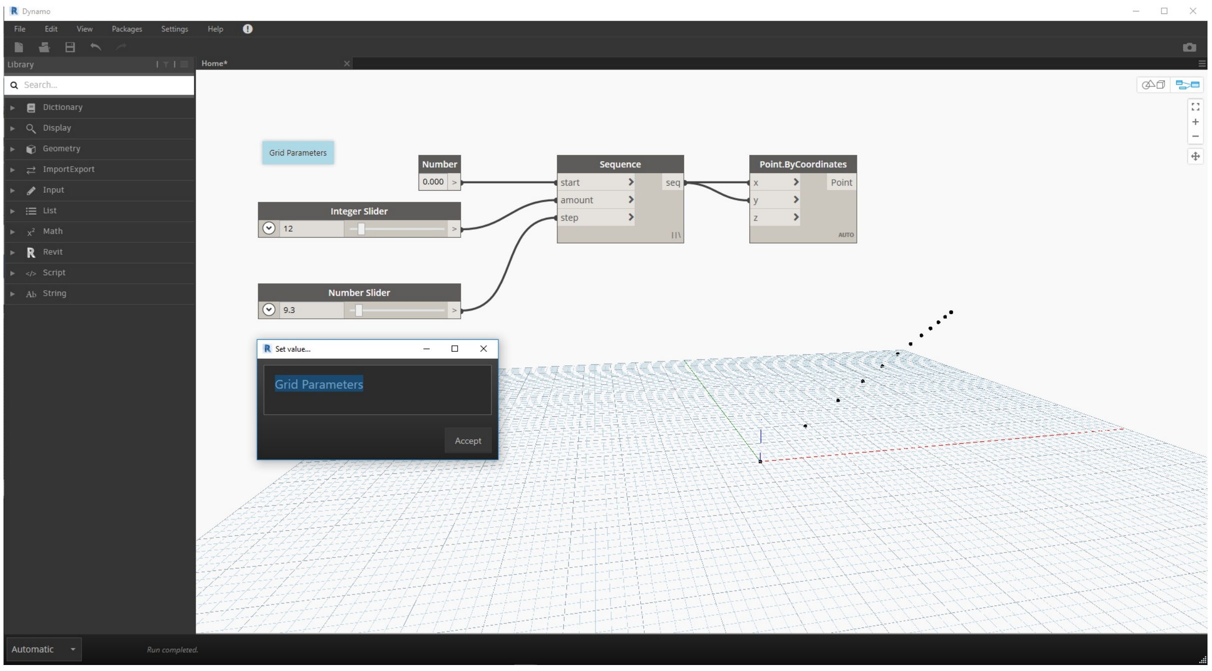
Notes

With some experience, we may be able to "read" the Visual Program by reviewing the Node Names and following the Program Flow. For users of all experience levels, it is also good practice to include plain language labels and descriptions. Dynamo has a **Notes** Node with an editable text field to do so. We can add Notes to the Workspace in two ways:



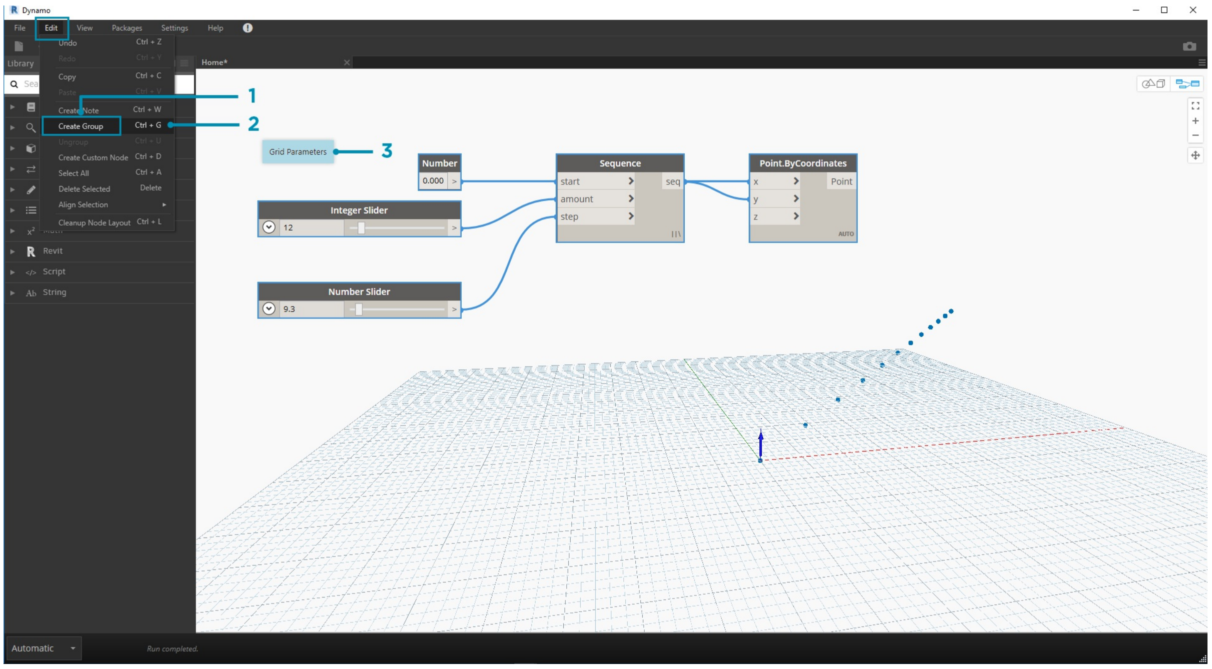
1. Browse to the menu Edit > Create Note
2. Use the keyboard shortcut Ctrl+W

Once the Note is added to the Workspace a text field will pop up allowing us to edit the text in the Note. After they are created, we can edit the Note by double-clicking or right-clicking the Note Node.



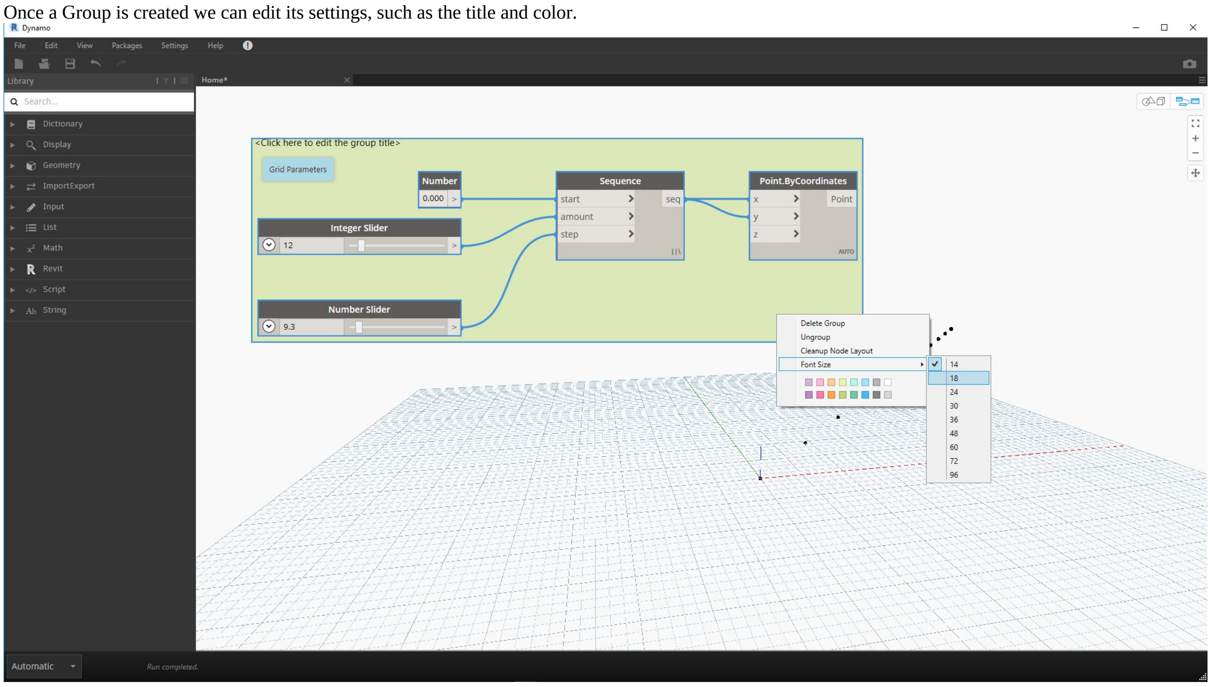
Grouping

When our Visual Program gets big, it is helpful to identify the larger steps that will be executed. We can highlight larger collections of Nodes with a **Group** to label them with a colored rectangle in the background and a title. There are three ways to make a Group with more than one Node selected:



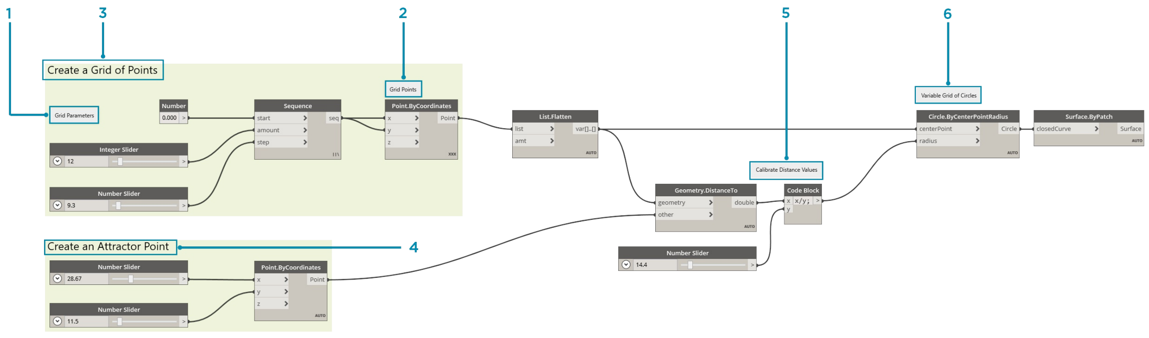
1. Browse to the menu Edit > Create Group
2. Use the keyboard shortcut Ctrl+G
3. Right-click on the Workspace and select "Create Group"

Once a Group is created we can edit its settings, such as the title and color.



Tip: Using both Notes and Groups is an effective way to annotate your file and increase readability.

Here's our program from Section 2.4 with Notes and Groups added:



1. Note: "Grid Parameters"
2. Note: "Grid Points"
3. Group: "Create a Grid of Points"
4. Group: "Create an Attractor Point"
5. Note: "Calibrate Distance Values"
6. Note: "Variable Grid of Circles"

The Building Blocks of Programs

THE BUILDING BLOCKS OF PROGRAMS

Once we are ready to dive deeper into developing Visual Programs, we will need a deeper understanding of the building blocks we will use. This chapter introduces fundamental concepts around data - the stuff that travels through the Wires of our Dynamo program.

ToString
NurbsCurve.ByPoints
Excel.ReadFromFile
Color.ByARGB
PointAnalysisData.ByPoints
Element.Name
List.Count
NewtonRootFind1DWithDeriv
Map
Math.Cosh
Topology.Faces
Cuboid.ByCorners
Color.ByARGB
TEST Recursion
Number Slider
LunchBox Domain Variables
Number Slider
Number Sequence
XYZ Array on Curve
&&
If

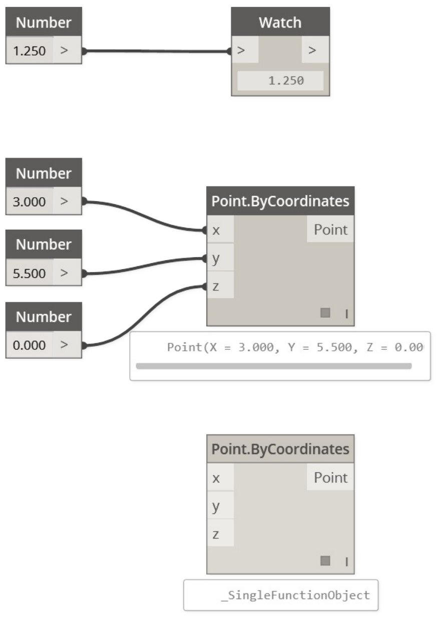
Data

Data

Data is the stuff of our programs. It travels through Wires, supplying inputs for Nodes where it gets processed into a new form of output data. Let's review the definition of data, how it's structured, and begin using it in Dynamo.

What is Data?

Data is a set of values of qualitative or quantitative variables. The simplest form of data is numbers such as `0`, `3.14`, or `17`. But data can also be of a number of different types: a variable representing changing numbers (`height`); characters (`myName`); geometry (`Circle`); or a list of data items (`1,2,3,5,8,13,...`). We need data to add to the input Ports of Dynamo's Nodes - we can have data without actions but we need data to process the actions that our Nodes represent. When we've added a Node to the Workspace, if it doesn't have any inputs supplied, the result will be a function, not the result of the action itself.



```
Point(X = 3.000, Y = 5.500, Z = 0.00
```

```
_SingleFunctionObject
```

1. Simple Data
2. Data and Action (A Node) successfully executes
3. Action (A Node) without Data Inputs returns a generic function

Beware of Nulls

The `'null'` type represents the absence of data. While this is an abstract concept, you will likely come across this while working with Visual Programming. If an action doesn't create a valid result, the Node will return a null. Testing for nulls and removing nulls from data structure is a crucial part to creating robust programs.

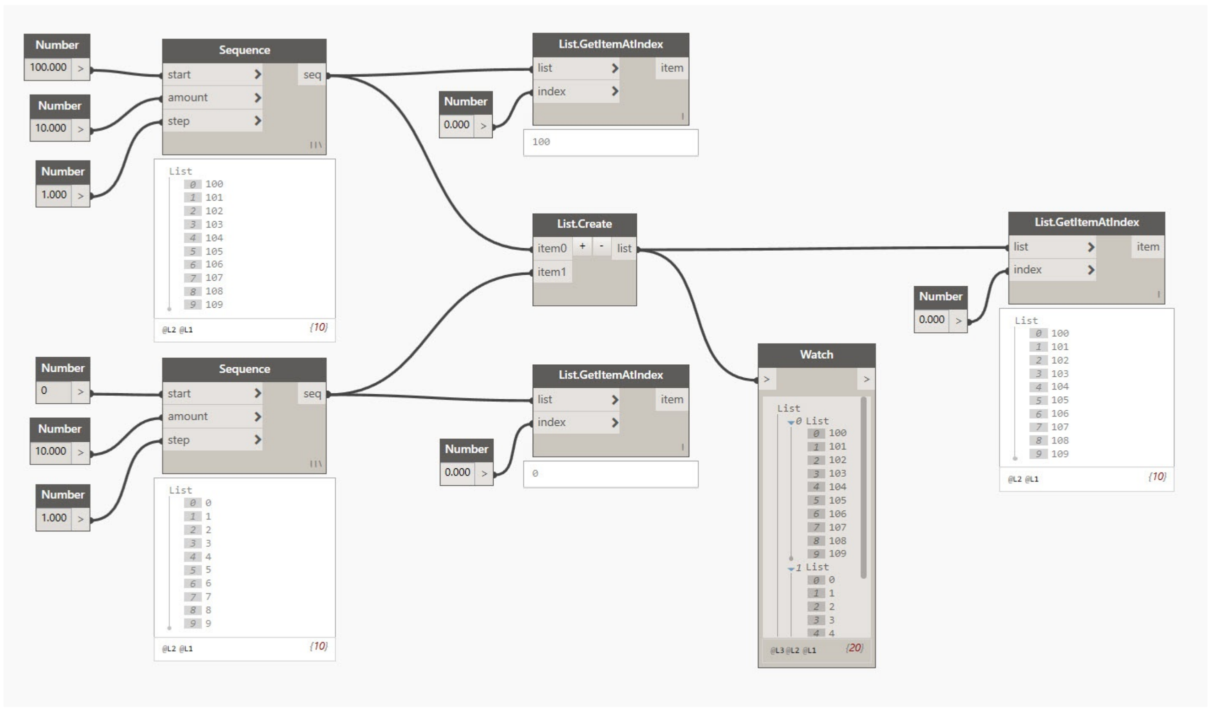
Icon	Name/Syntax	Inputs	Outputs
∅?	Object.IsNull	obj	bool

Data Structures

When we are Visual Programming, we can very quickly generate a lot of data and require a means of managing its hierarchy. This is the role of Data Structures, the organizational schemes in which we store data. The specifics of Data Structures and how to use them vary from programming language to programming language. In Dynamo, we add hierarchy to our data through Lists. We will explore this in depth in later chapters, but let's start simply:

A list represents a collection of items placed into one structure of data:

- I have five fingers (*items*) on my hand (*list*).
- There are ten houses (*items*) on my street (*list*).



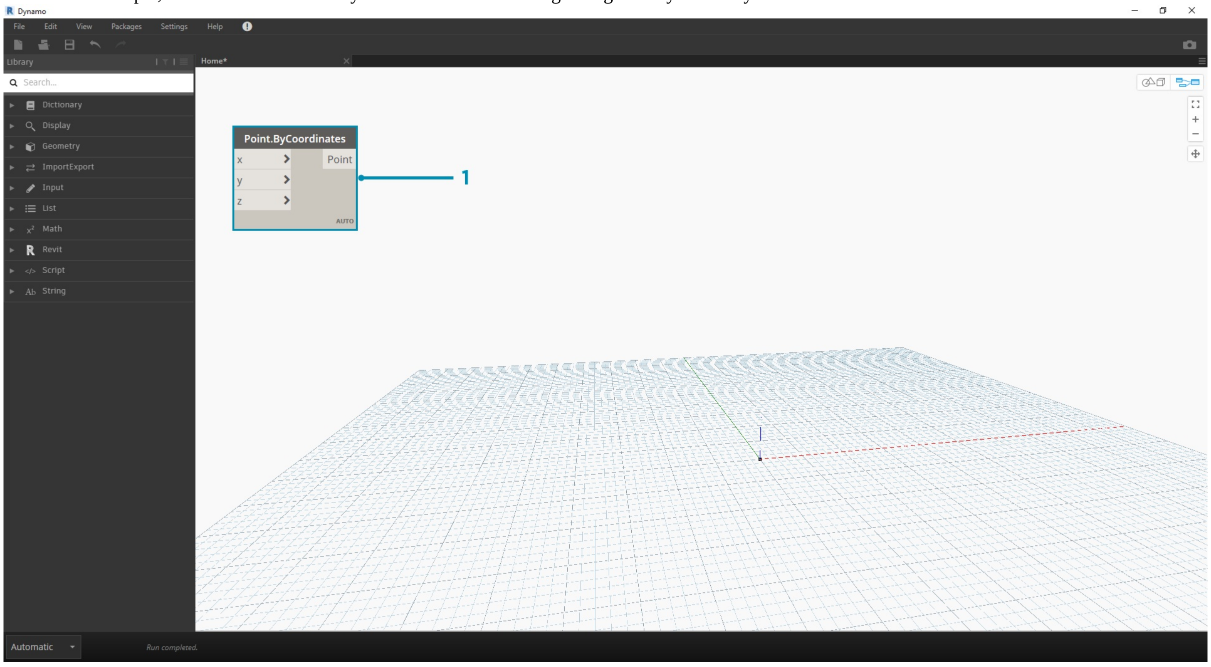
1. A **Number Sequence** node defines a list of numbers by using a *start*, *amount*, and *step* input. With these nodes, we've created two separate lists of ten numbers, one which ranges from *100-109* and another which ranges from *0-9*.
2. The **List.GetItemAtIndex** node selects an item in a list at a specific index. When choosing *0*, we get the first item in the list (*100* in this case).
3. Applying the same process to the second list, we get a value of *0*, the first item in the list.
4. Now we merge the two lists into one by using the **List.Create** node. Notice that the node creates a *list of lists.* This changes the structure of the data.
5. When using **List.GetItemAtIndex** again, with index set to *0*, we get the first list in the list of lists. This is what it means to treat a list as an item, which is somewhat different from other scripting languages. We will get more advanced with list manipulation and data structure in later chapters.

The key concept to understand about data hierarchy in Dynamo: **with respect to data structure, lists are regarded as items.** In other words, Dynamo functions with a top-down process for understanding data structures. What does this mean? Let's walk through it with an example.

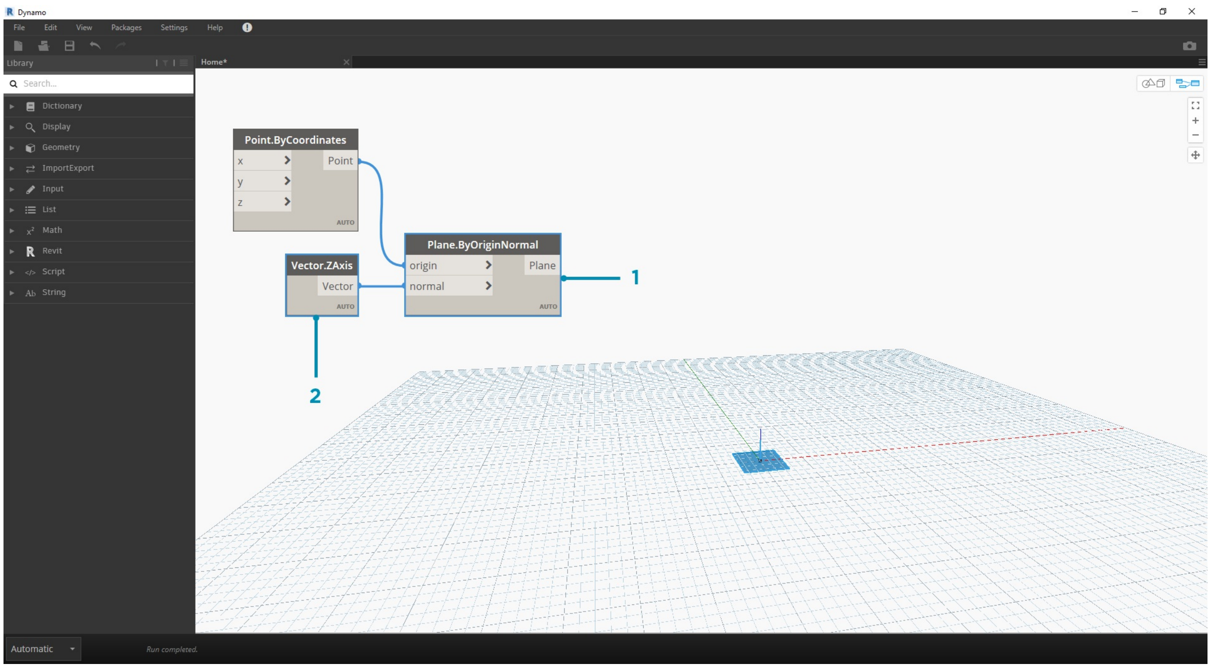
Using Data to Make a Chain of Cylinders

> Download the example file that accompanies this exercise (Right click and "Save Link As..."): Building Blocks of Programs - Data.dyn. A full list of example files can be found in the Appendix.

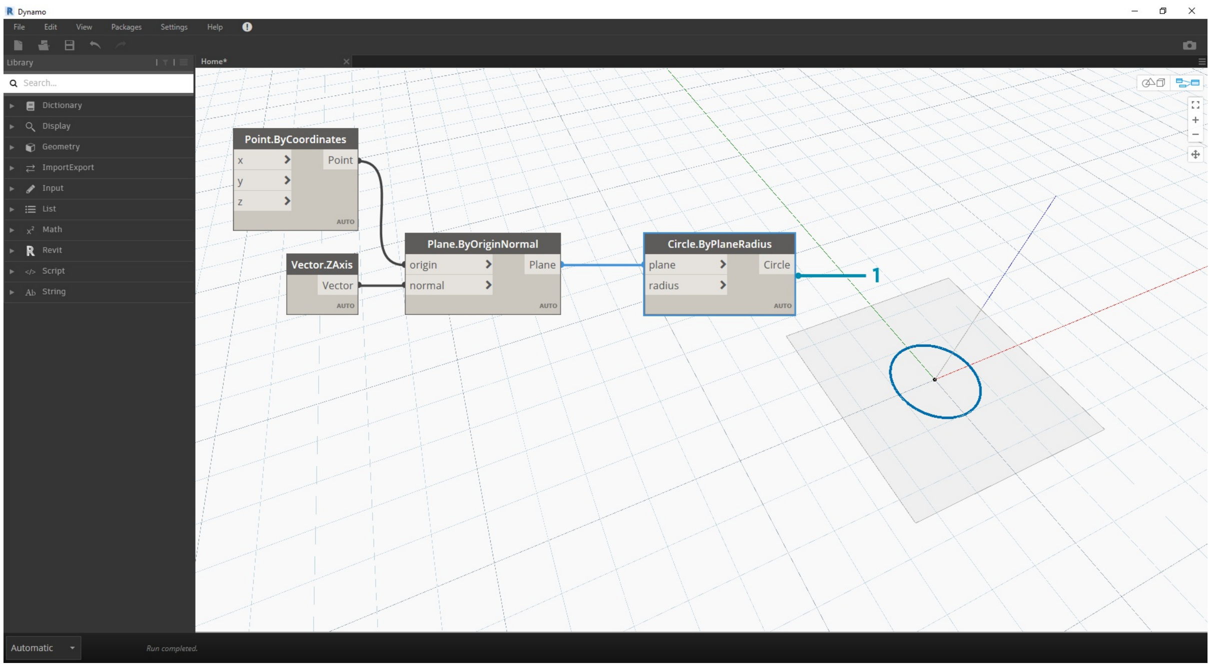
In this first example, we assemble a shelled cylinder which walks through the geometry hierarchy discussed in this section.



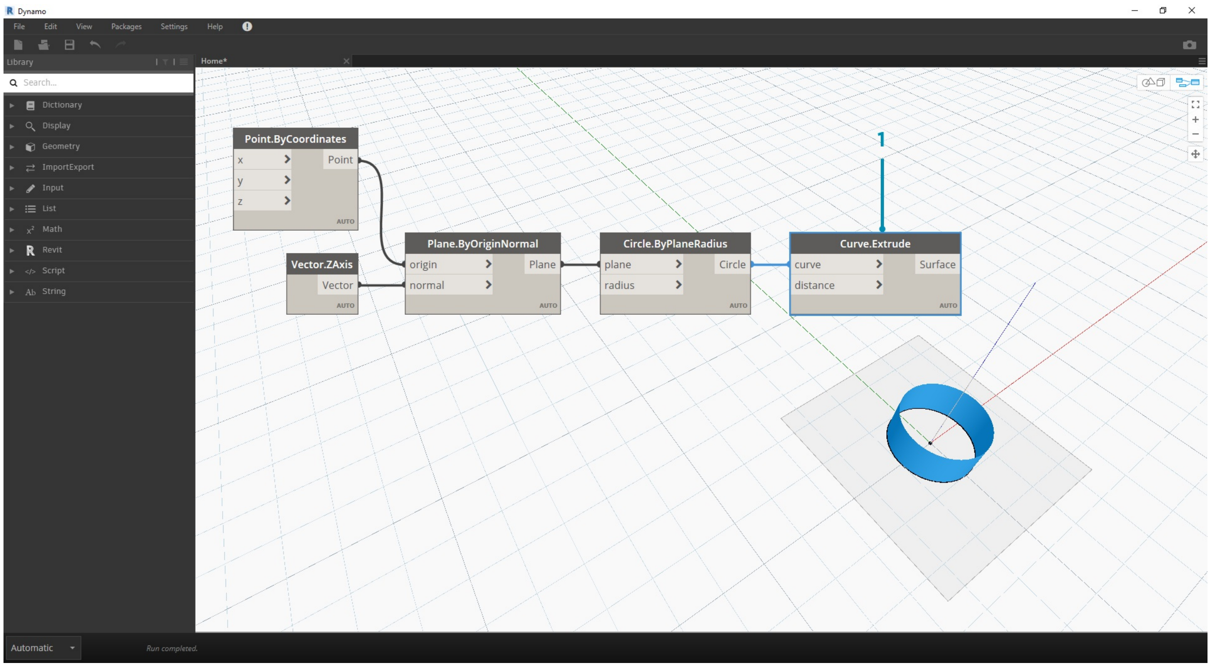
1. **Point.ByCoordinates -** after adding the node to canvas, we see a point at the origin of the Dynamo preview grid. The default values of the *x,y*, and *z* inputs are *0.0*, giving us a point at this location.



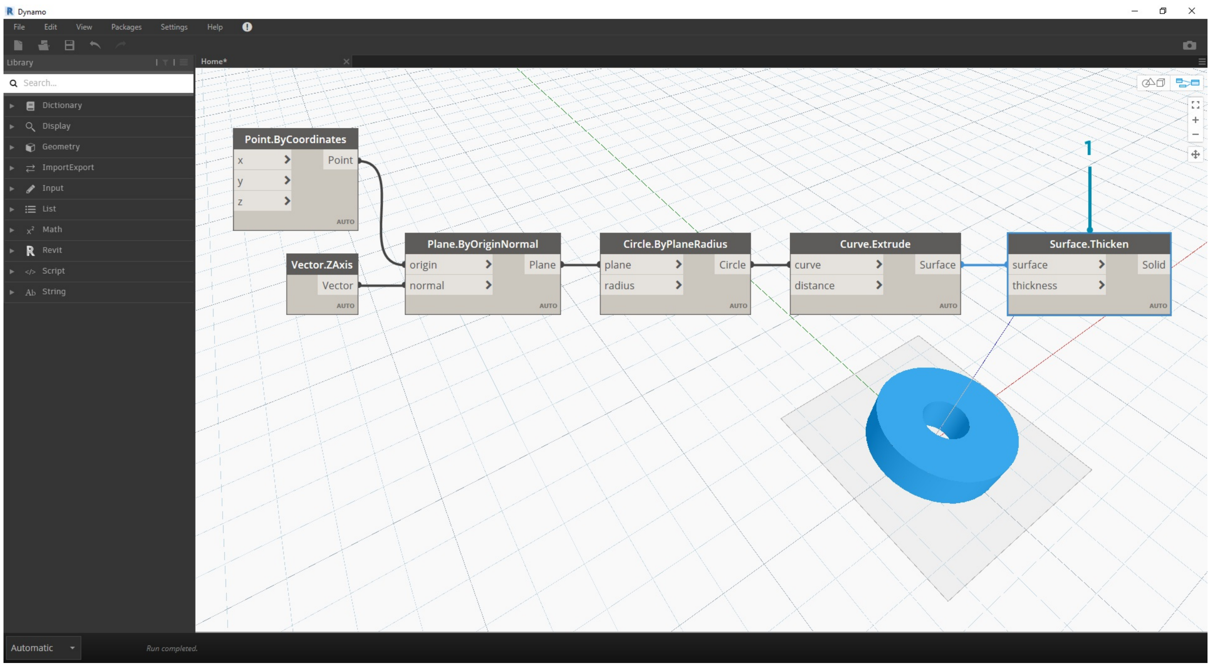
1. **Plane.ByOriginNormal -** The next step in the geometry hierarchy is a plane. There are several ways to construct a plane, and we are using an origin and normal for the input. The origin is the point node created in the previous step.
2. **Vector.ZAxis -** this is a unitized vector in the z direction. Notice there are not inputs, only a vector of [0,0,1] value. We use this as the *normal* input for the *Plane.ByOriginNormal* node. This gives us a rectangular plane in the Dynamo preview.



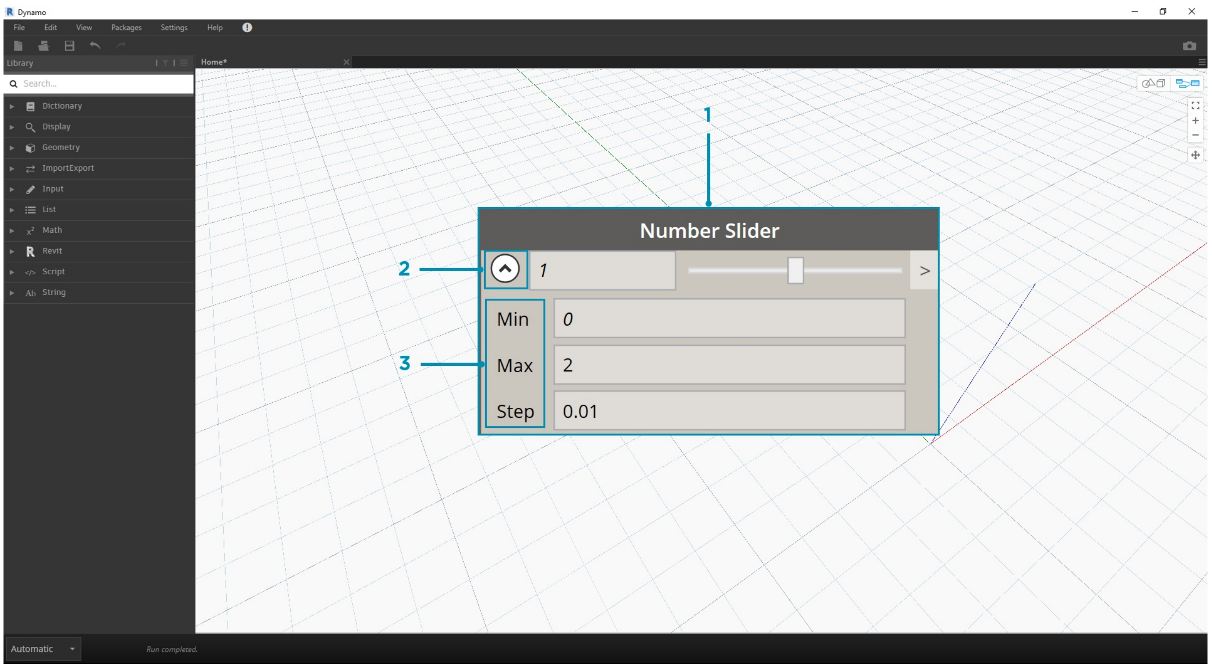
1. **Circle.ByPlaneRadius -** Stepping up the hierarchy, we now create a curve from the plane in our previous step. After plugging into the node, we get a circle at the origin. The default radius on the node is value of *1*.



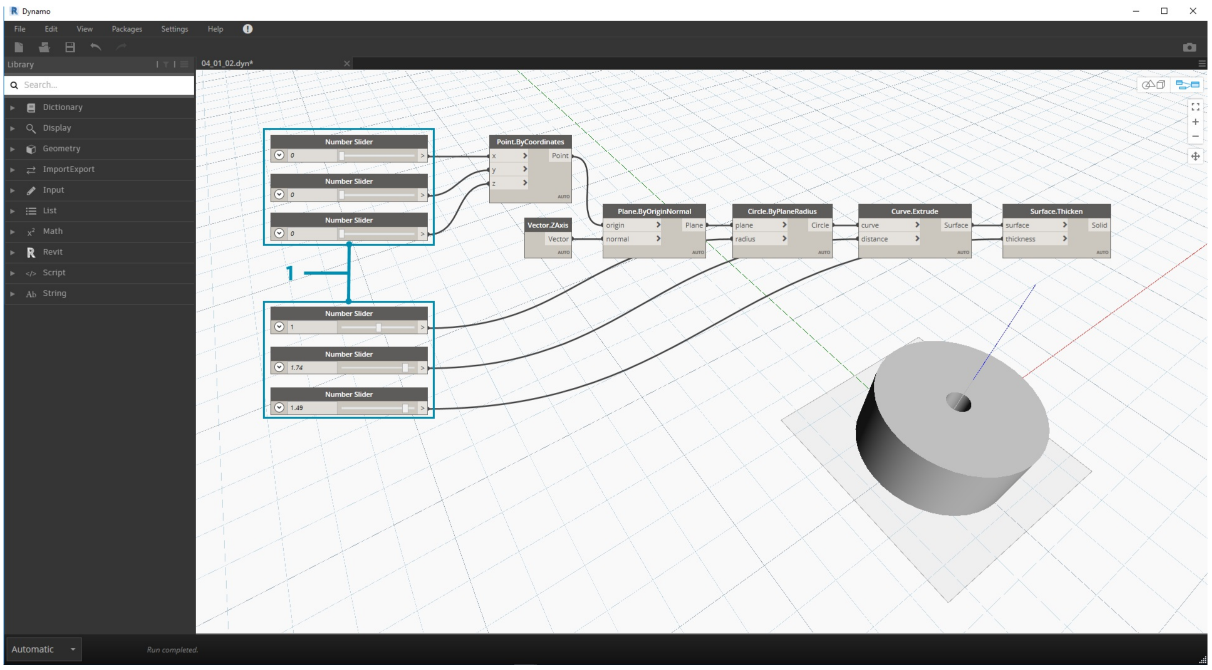
1. **Curve.Extrude -** Now we make this thing pop by giving it some depth and going in the third dimension. This node creates a surface from a curve by extruding it. The default distance on the node is *1*, and we should see a cylinder in the viewport.



1. **Surface.Thicken -** This node gives us a closed solid by offsetting the surface a given distance and closing the form. The default thickness value is *1*, and we see a shelled cylinder in the viewport in line with these values.

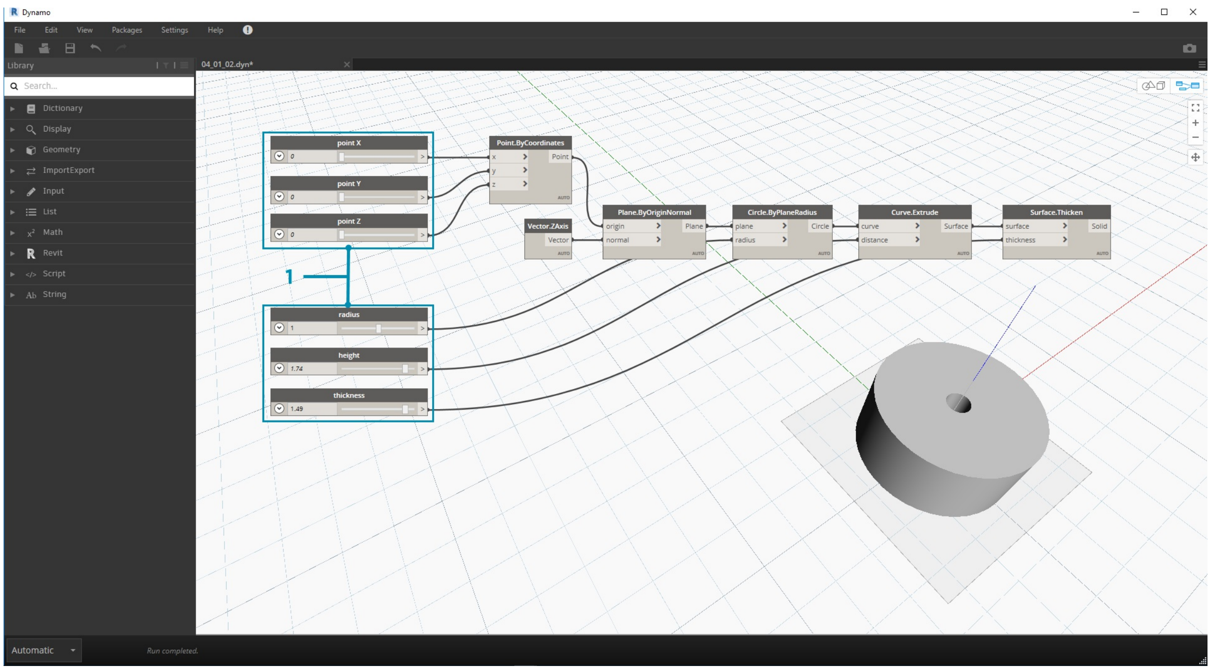


1. **Number Slider -** Rather than using the default values for all of these inputs, let's add some parametric control to the model.
2. **Domain Edit -** after adding the number slider to the canvas, click the caret in the top left to reveal the domain options.
3. **Min/Max/Step -** change the *min*, *max*, and *step* values to *0*,*2*, and *0.01* respectively. We are doing this to control the size of the overall geometry.



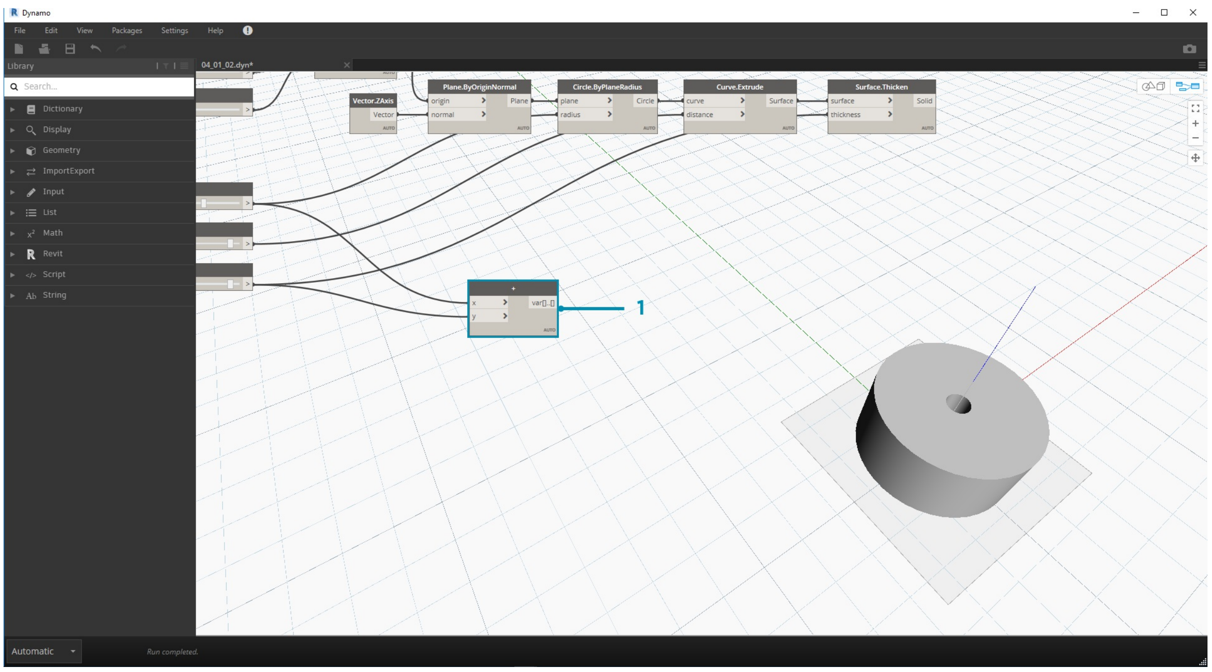
1. **Number Sliders -** In all of the default inputs, let's copy and paste this number slider (select the slider, hit Ctrl+C, then Ctrl+V) several times, until all of the inputs with defaults have a slider instead. Some of the slider values will have to be larger than zero to get the definition to work (ie: you need an extrusion depth in order to have a surface to thicken).

We've now created a parametric shelled cylinder with these sliders. Try to flex some of these parameters and see the geometry update dynamically in the Dynamo viewport.

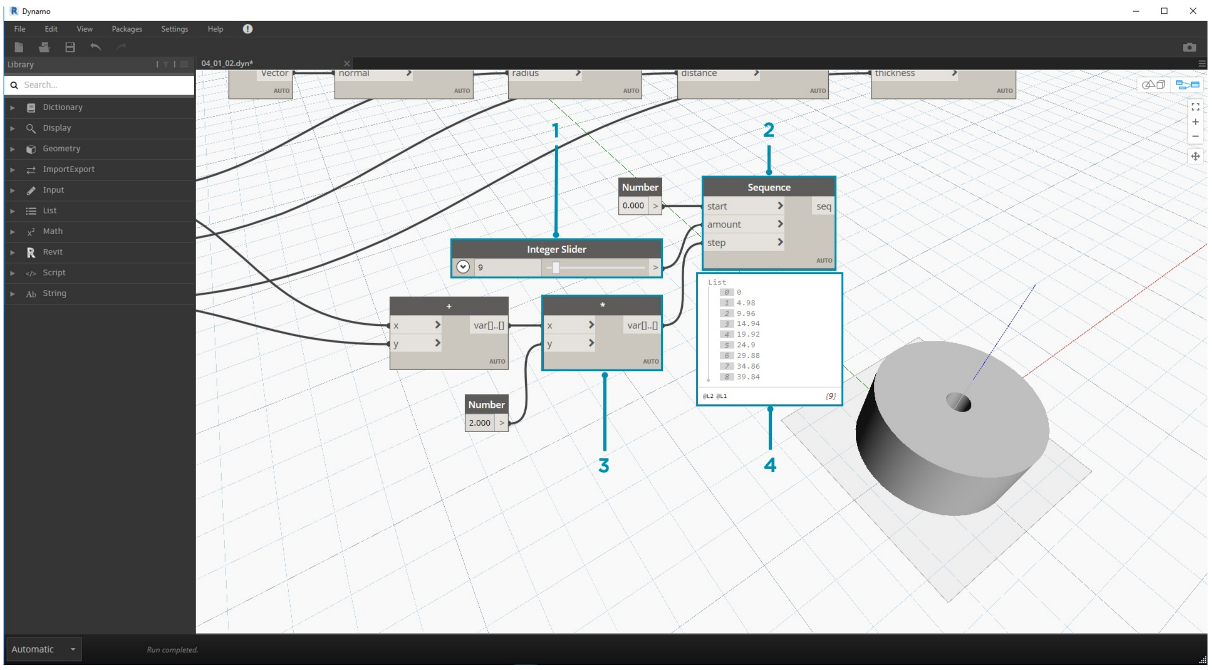


1. **Number Sliders -** taking this a step further, we've added a lot of sliders to the canvas, and need to clean up the interface of the tool we just created. Right click on one slider, select "Rename..." and change each slider to the appropriate name for its parameter. You can reference the image above for names.

At this point, we've created an awesome thickening cylinder thing. This is one object currently, let's look at how to create an array of cylinders that remains dynamically linked. To do this, we're going to create a list of cylinders, rather than working with a single item.

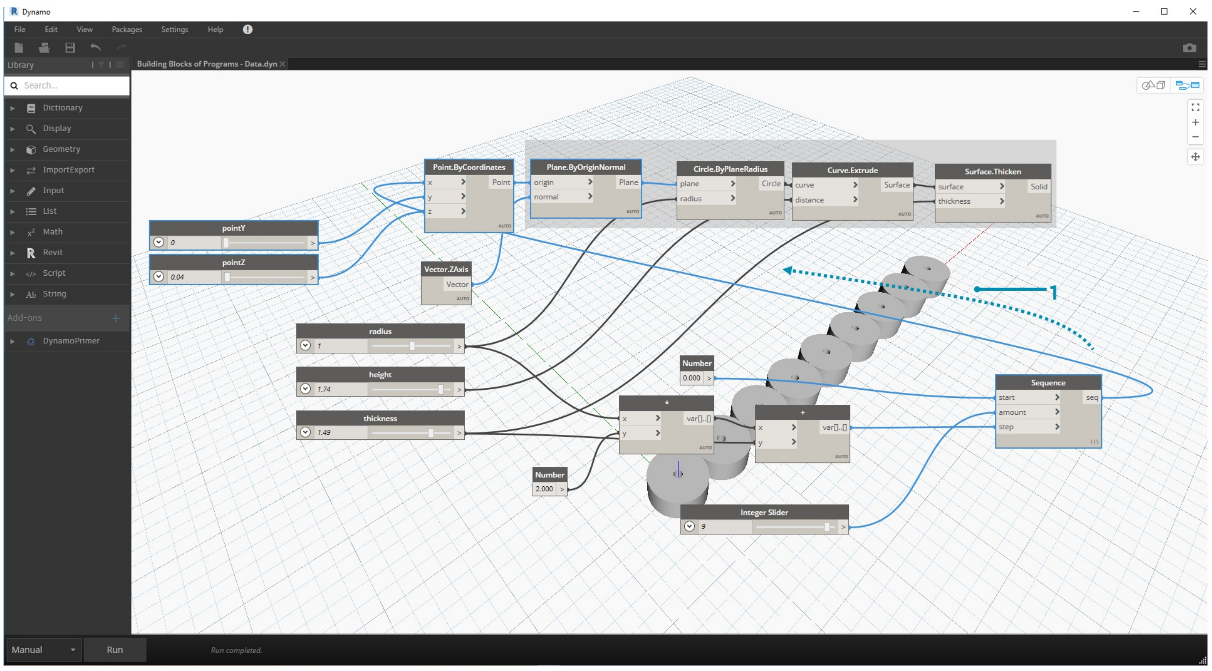


1. **Addition (+) -** Our goal is to add a row of cylinders next to the cylinder we've created. If we want to add one cylinder adjacent to the current one, we need to consider both radius of the cylinder and the thickness of its shell. We get this number by adding the two values of the sliders.

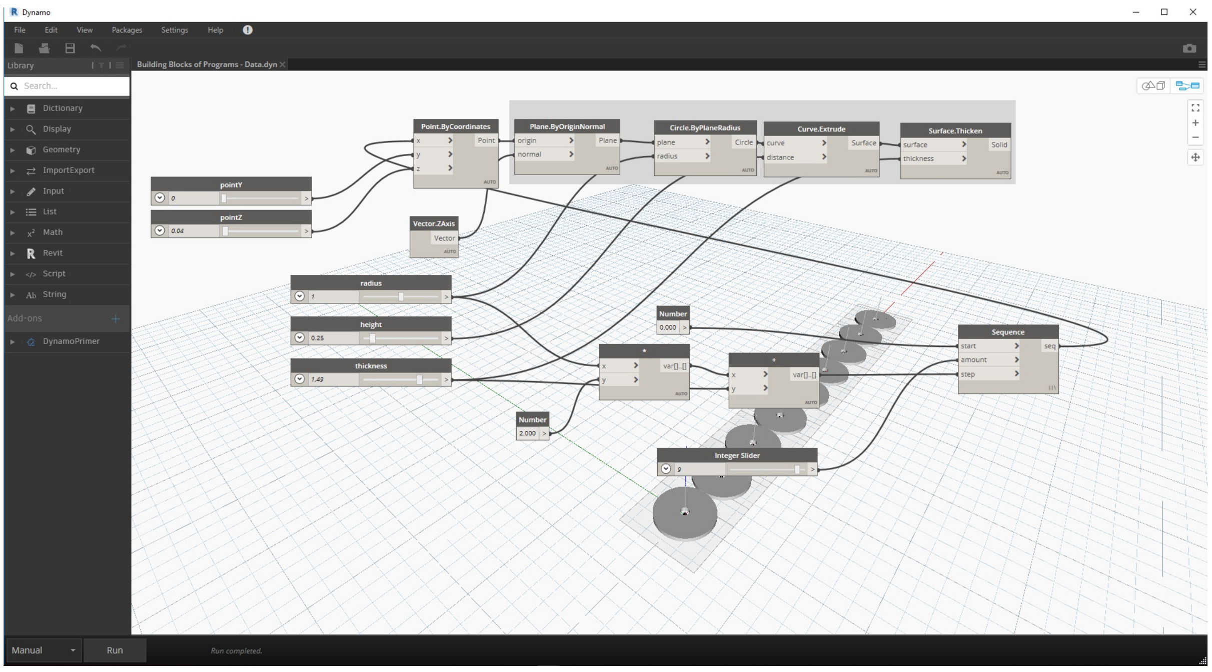


This step is more involved so let's walk through it slowly: the end goal is to create a list of numbers which define the locations of each cylinder in a row.

1. **Multiplication -** First, we want to multiply the value from the previous step by 2. The value from the previous step represents a radius, and we want to move the cylinder the full diameter.
2. **Number Sequence -** we create an array of numbers with this node. The first input is the *multiplication* node from the previous step into the *step* value. The *start* value can be set to *0.0* using a *number* node.
3. **Integer Slider -** For the *amount* value, we connect an integer slider. This will define how many cylinders are created.
4. **Output -** This list shows us the distance moved for each cylinder in the array, and is parametrically driven by the original sliders.



1. This step is simple enough - plug the sequence defined in the previous step into the *x* input of the original *Point.ByCoordinates*. This will replace the slider *pointX* which we can delete. We now see an array of cylinders in the viewport (make sure the integer slider is larger than 0).



The chain of cylinders is still dynamically linked to all of the sliders. Flex each slider to watch the definition update!

Math

Math

If the simplest form of data is numbers, the easiest way to relate those numbers is through Mathematics. From simple operators like divide to trigonometric functions, to more complex formulas, Math is a great way to start exploring numeric relationships and patterns.

Arithmetic Operators

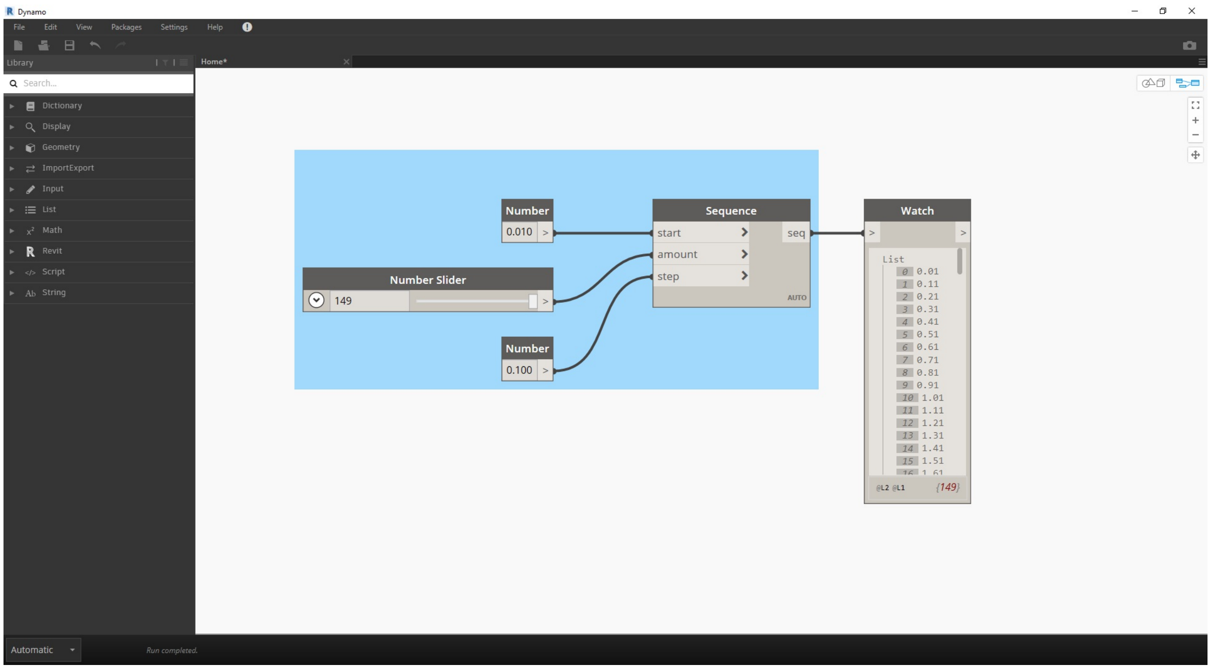
Operators are a set of components that use algebraic functions with two numeric input values, which result in one output value (addition, subtraction, multiplication, division, etc.). These can be found under Operators>Actions.

Icon	Name	Syntax	Inputs	Outputs
➕	Add	+	var[]...[], var[]...[]	var[]...[]
➖	Subtract	-	var[]...[], var[]...[]	var[]...[]
✳	Multiply	*	var[]...[], var[]...[]	var[]...[]
➗	Divide	/	var[]...[], var[]...[]	var[]...[]

Parametric Formula

> Download the example file that accompanies this exercise (Right click and "Save Link As..."): [Building Blocks of Programs - Math.dyn](). A full list of example files can be found in the Appendix.

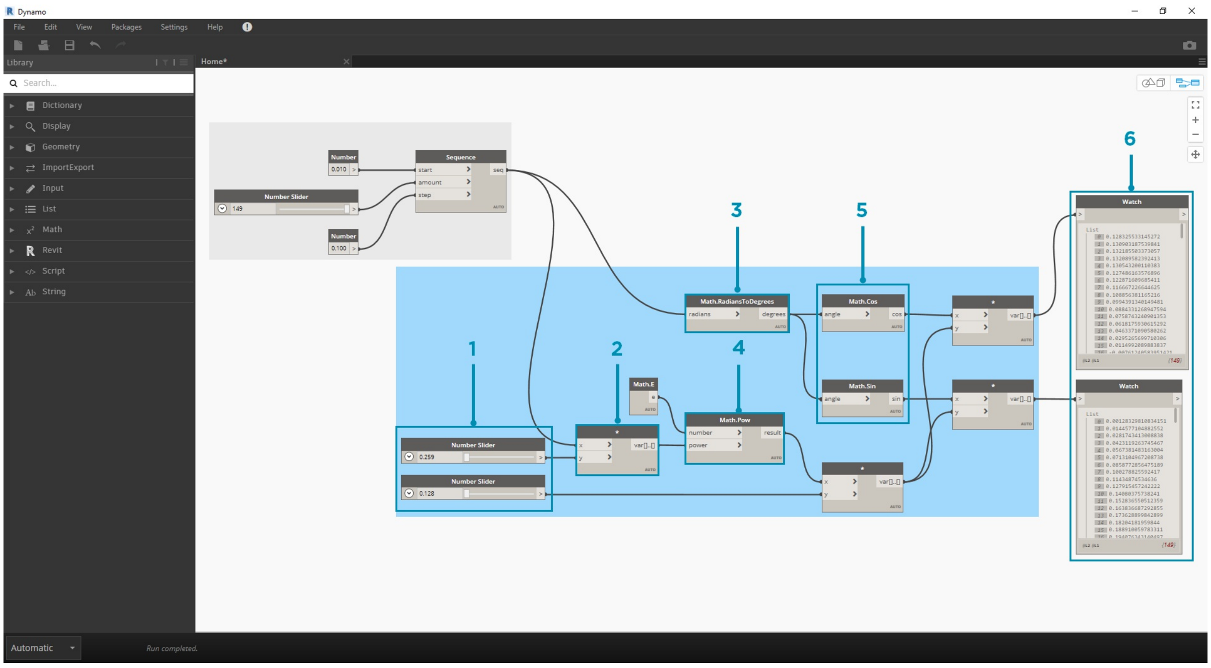
From Operators, the next logical step is to combine operators and variables to form a more complex relationship through **Formulas**. Let's make a Formula that can be controlled by input parameters, like sliders.



```
List
  0 0.01
  1 0.11
  2 0.21
  3 0.31
  4 0.41
  5 0.51
  6 0.61
  7 0.71
  8 0.81
  9 0.91
  10 1.01
  11 1.11
  12 1.21
  13 1.31
  14 1.41
  15 1.51
  16 1.61
```

1. **Number Sequence:** define a number sequence based on three inputs: *start, amount* and *step*. This sequence represents the 't' in the parametric equation, so we want to use a list that's large enough to define a spiral.

The step above has created a list of numbers to define the parametric domain. The golden spiral is defined as the equation: $x = r\cos\theta = a\cos\theta\, e^{b\theta}$ and $y = r\sin\theta = a\sin\theta\, e^{b\theta}$. The group of Nodes below represent this equation in visual programming form.

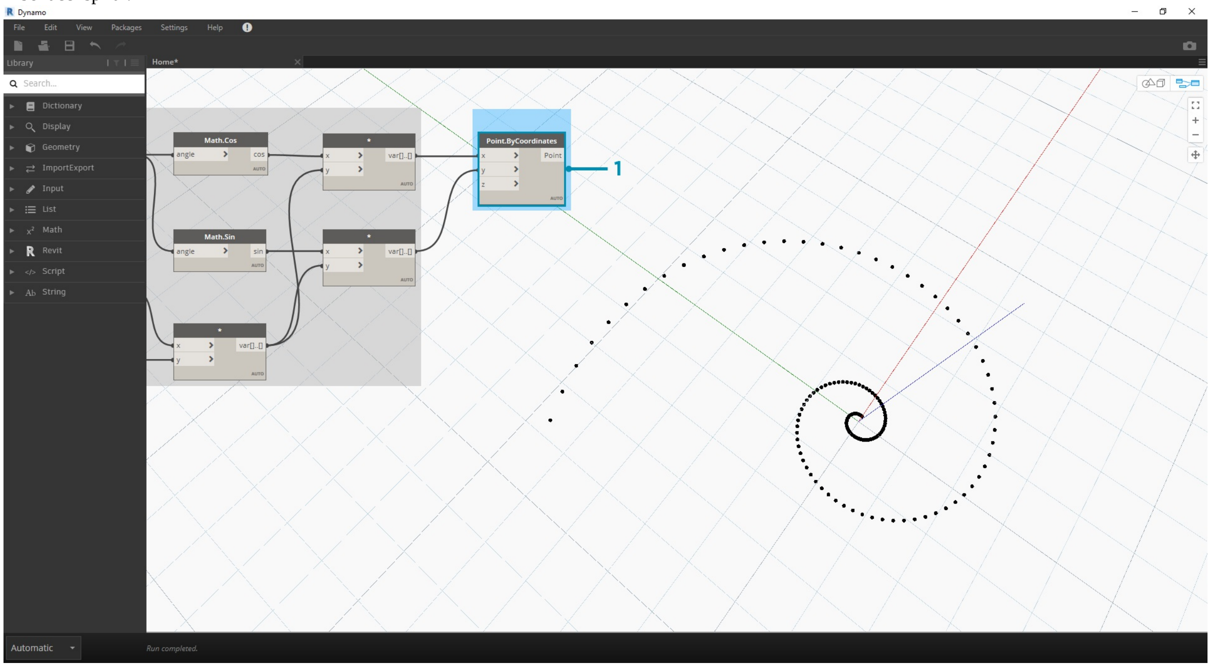


When stepping through the group of Nodes, try to pay attention to the parallel between the visual program and written equation.

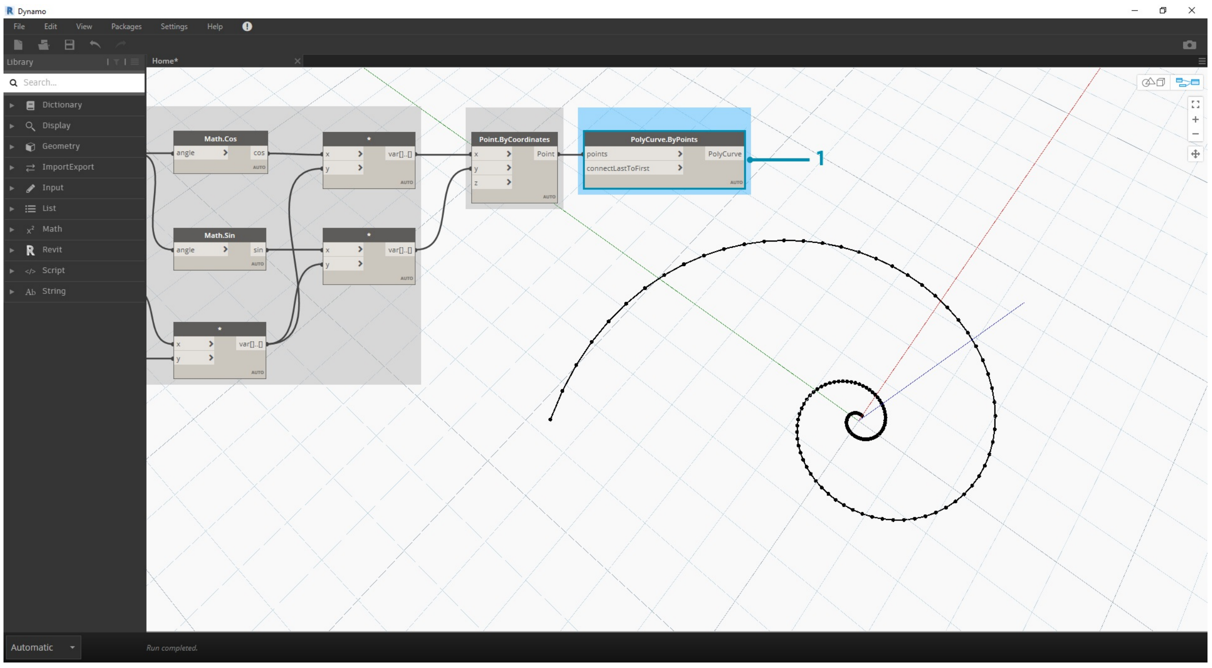
1. **Number Slider:** Add two number sliders to the canvas. These sliders will represent the *a* and the *b* variables of the parametric equation. These represent a constant which is flexible, or parameters which we can adjust towards a desired outcome.
2. *** :** The multiplication Node is represented by an asterisk. We'll use this repeatedly to connect multiplying variables
3. **Math.RadiansToDegrees:** The '*t*' values need to be translated to degrees for their evaluation in the trigonometric functions. Remember, Dynamo defaults to degrees for evaluating these functions.
4. **Math.Pow:** as a function of the '*t*' and the number '*e*' this creates the Fibonacci sequence.
5. **Math.Cos and Math.Sin:** These two trigonmetric functions will differentiate the x-coordinate and the y-coordinate, respectively, of each parametric point.
6. **Watch:** We now see that our output is two lists, these will be the *x* and *y* coordinates of the points used to generate the spiral.

From Formula to Geometry

Now, the bulk of Nodes from the previous step will work fine, but it is a lot of work. To create a more efficient workflow, have a look at **Code Blocks** (section 3.3.2.3) to define a string of Dynamo expressions into one node. In this next series of steps, we'll look at using the parametric equation to draw the Fibonacci spiral.



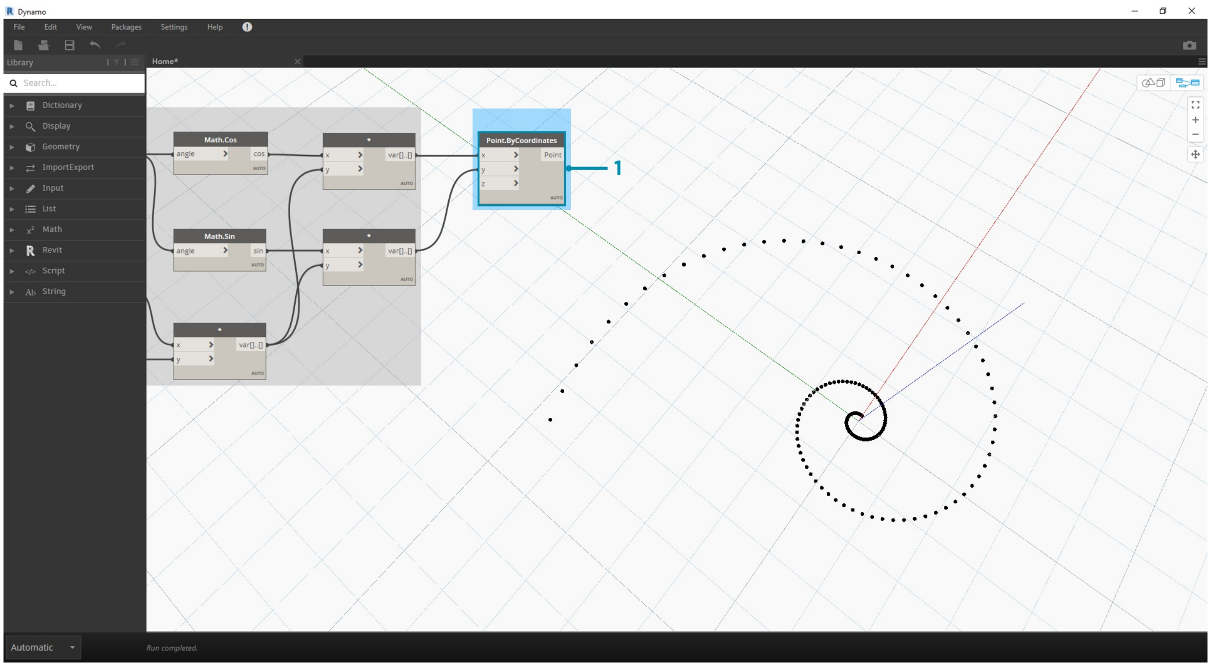
1. **Point.ByCoordinates:** Connect the upper multiplication node into the '*x*' input and the lower into the '*y*' input. We now see a parametric spiral of points on the screen.



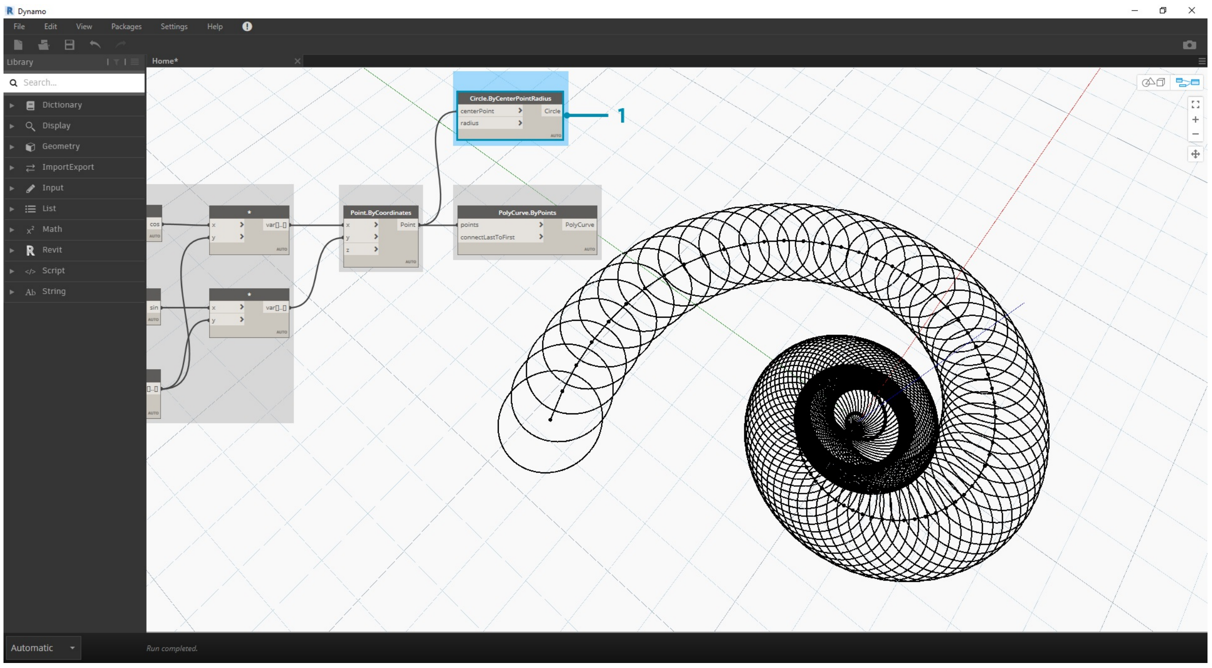
1. **Polycurve.ByPoints:** Connect Point.ByCoordinates from the previous step into *points*. We can leave *connectLastToFirst* without an input because we aren't making a closed curve. This creates a spiral which passes through each point defined in the previous step.

We've now completed the Fibonacci Spiral! Let's take this further into two separate exercises from here, which we'll call the Nautilus and the Sunflower. These are abstractions of natural systems, but the two different applications of the Fibonacci spiral will be well represented.

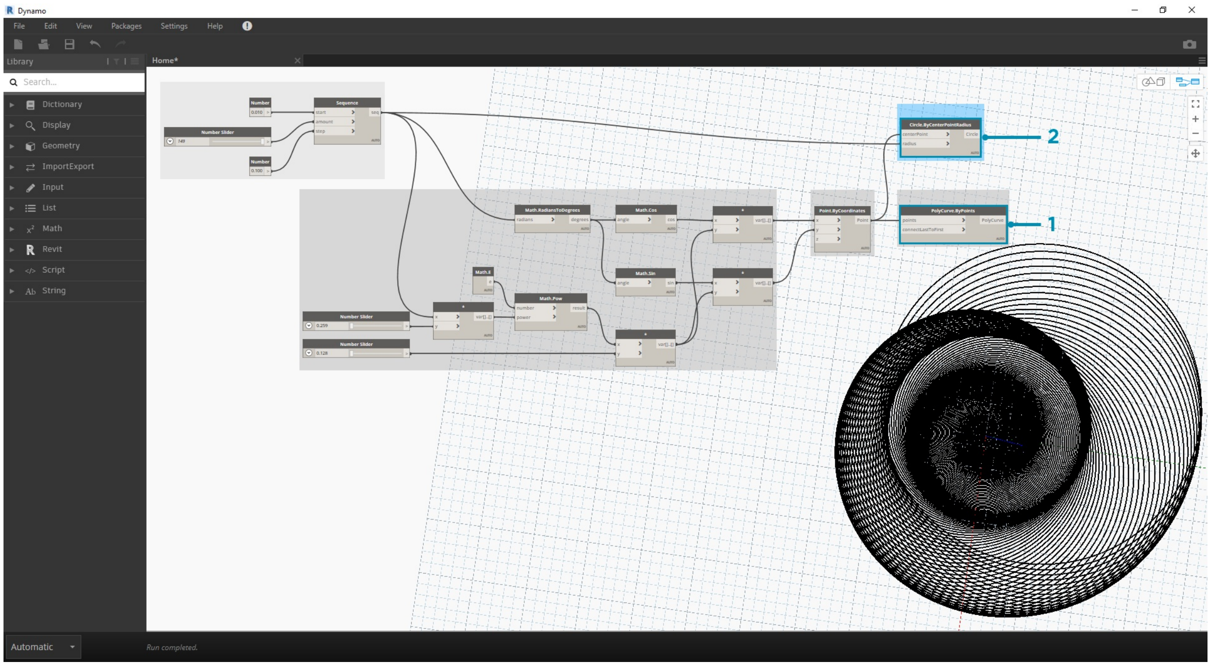
From Spiral to Nautilus



1. As a jumping-off point, let's start with the same step from the previous exercise: creating a spiral array of points with the **Point.ByCoordinates** Node.



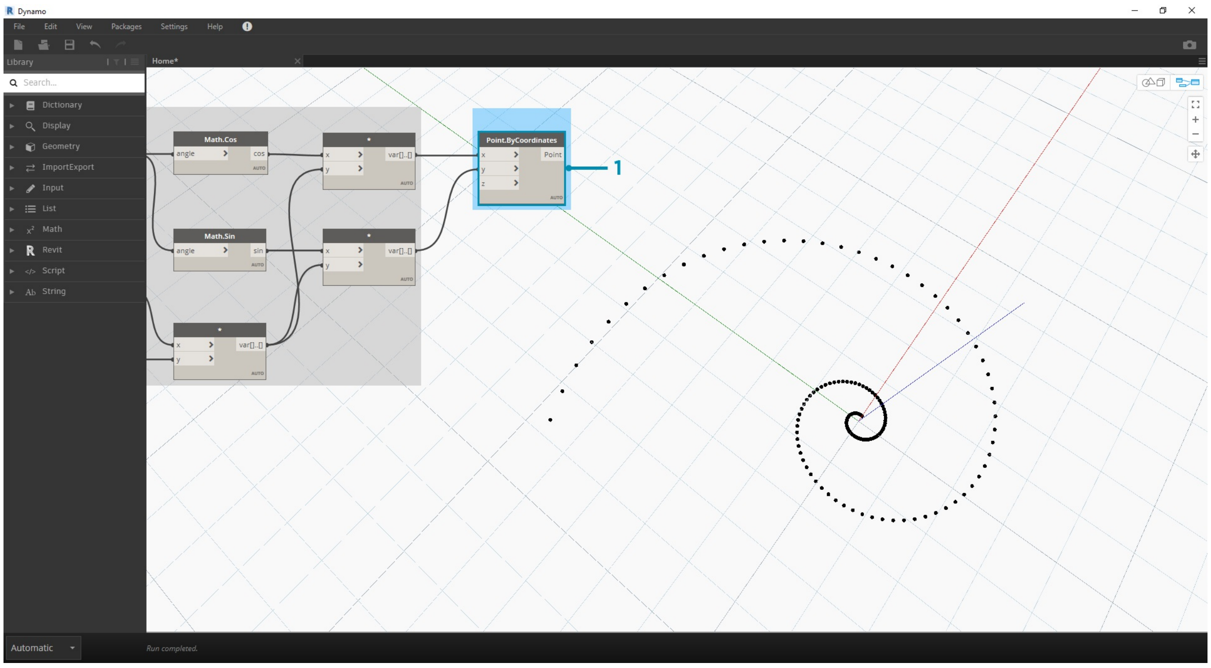
1. **Polycurve.ByPoints:** Again, this is the Node from the pervious exercise, which we'll use as a reference.
2. **Circle.ByCenterPointRadius:** We'll use a circle Node here with the same inputs as the previous step. The radius value defaults to *1.0*, so we see an immediate output of circles. It becomes immediately legible how the points diverge further from the origin.



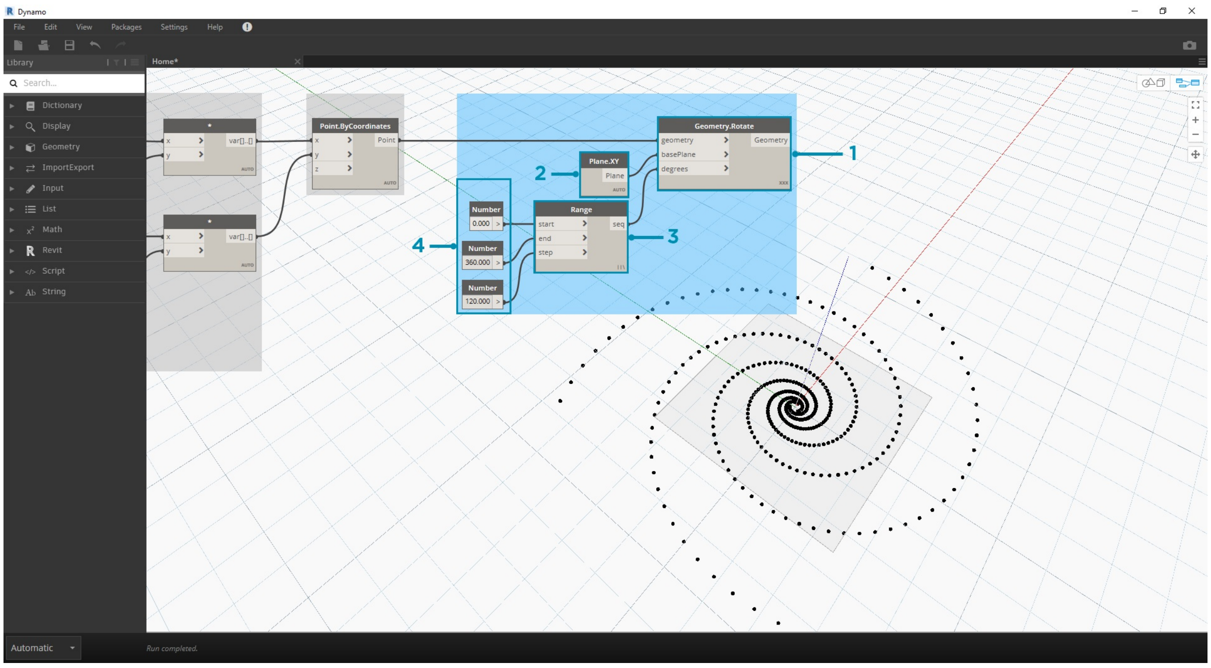
1. **Circle.ByCenterPointRadius:** To create a more dynamic array of circles, we plug the original number sequence (the '*t*' sequence) into the radius value.
2. **Number Sequence:** This is the original array of '*t*'. By plugging this into the radius value, the circle centers are still diverging further from the origin, but the radius of the circles is increasing, creating a funky Fibonacci circle graph. Bonus points if you make it 3D!

From Nautilus to Phyllotaxis Pattern

Now that we've made a circular Nautilus shell, let's jump into parametric grids. We're going to use a basic rotate on the Fibonacci Spiral to create a Fibonacci grid, and the result is modeled after the growth of sunflower seeds.

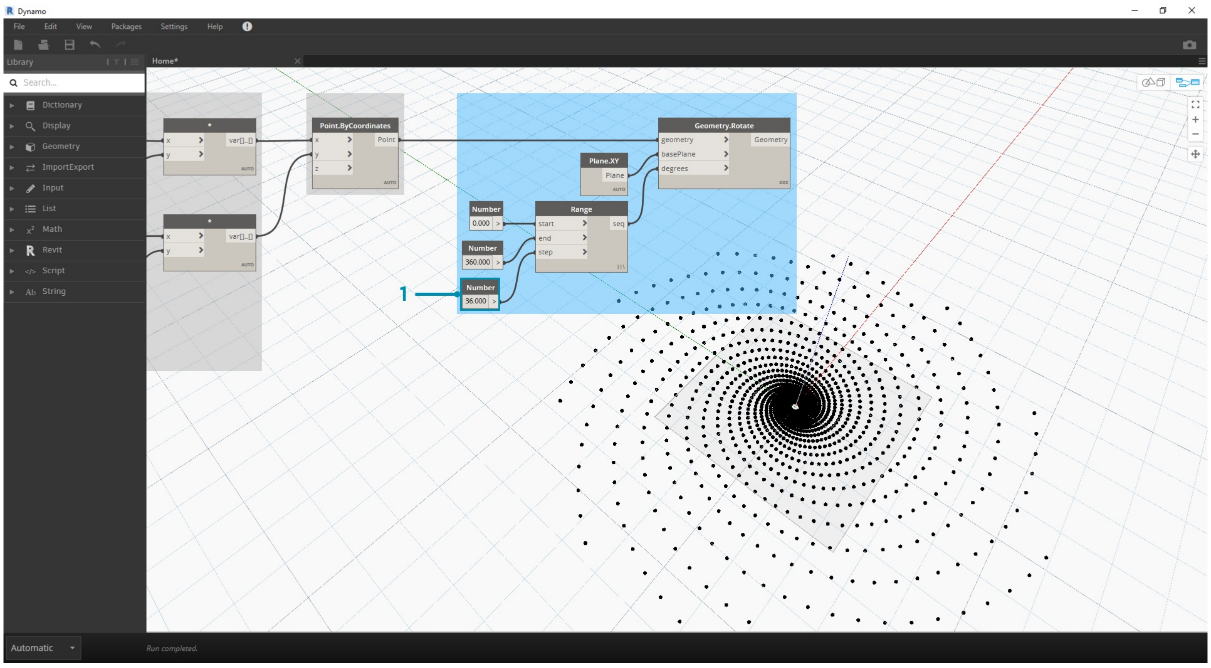


1. Again, as a jumping-off point, let's start with the same step from the previous exercise: creating a spiral array of points with the **Point.ByCoordinates** Node.

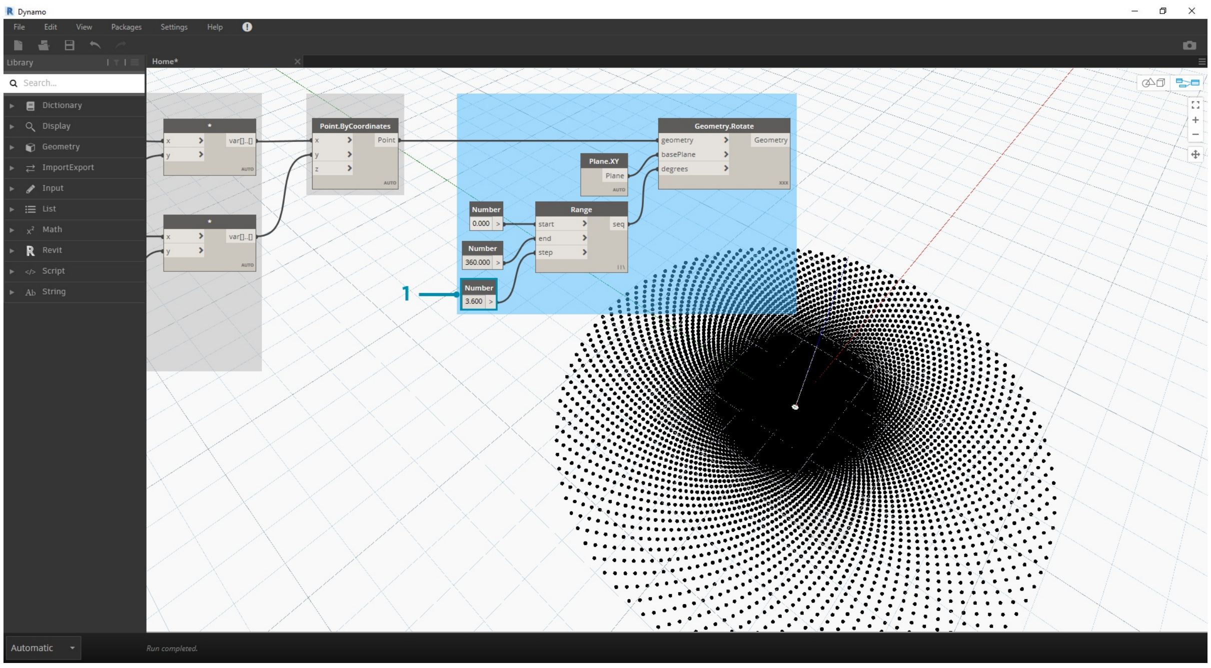


1. **Geometry.Rotate:** There are several Geometry.Rotate options; be certain you've chosen the Node with *geometry*,*basePlane*, and *degrees* as its inputs. Connect **Point.ByCoordinates** into the geometry input.
2. **Plane.XY:** Connect to the *basePlane* input. We will rotate around the origin, which is the same location as the base of the spiral.
3. **Number Range:** For our degree input, we want to create multiple rotations. We can do this quickly with a Number Range component. Connect this into the *degrees* input.
4. **Number:** And to define the range of numbers, add three number nodes to the canvas in vertical order. From top to bottom, assign values of *0.0*,*360.0,* and *120.0* respectively. These are driving the rotation of the spiral. Notice the output results from the **Number Range** node after connecting the three number nodes to the Node.

Our output is beginning to resemble a whirlpool. Let's adjust some of the **Number Range** parameters and see how the results change:



1. Change the step size of the **Number Range** node from *120.0* to *36.0*. Notice that this is creating more rotations and is therefore giving us a denser grid.



1. Change the step size of the **Number Range** node from *36.0* to *3.6*. This now gives us a much denser grid, and the directionality of the spiral is unclear. Ladies and gentlemen, we've created a sunflower.

Logic

Logic

Logic, or more specifically, **Conditional Logic**, allows us to specify an action or set of actions based on a test. After evaluating the test, we will have a Boolean value representing `True` or `False` that we can use to control the Program Flow.

Booleans

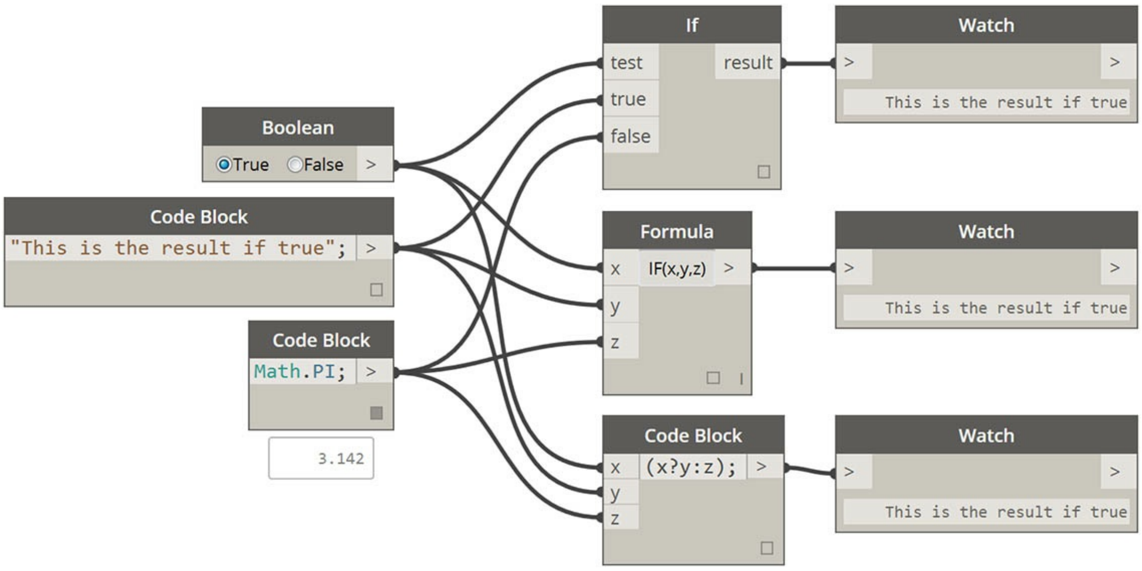
Numeric variables can store a whole range of different numbers. Boolean variables can only store two values referred to as True or False, Yes or No, 1 or 0. We rarely use booleans to perform calculations because of their limited range.

Conditional Statements

The "If" statement is a key concept in programming: "If *this* is true, then *that* happens, otherwise *something else* happens. The resulting action of the statement is driven by a boolean value. There are multiple ways to define an "If" statement in Dynamo:

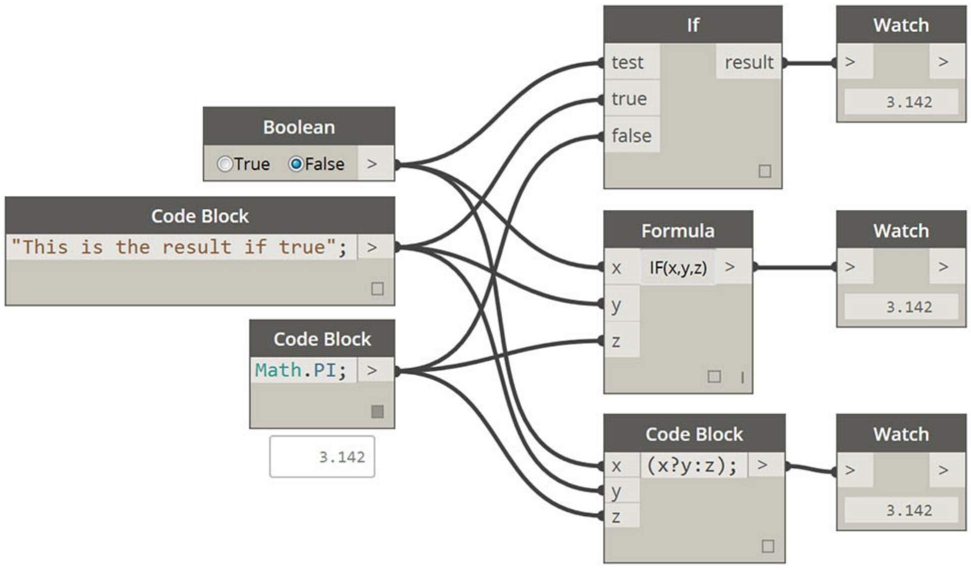
Icon	Name	Syntax	Inputs	Outputs
?	If	If	test, true, false	result
f x	Formula	IF(x,y,z)	x, y, z	result
[I]	Code Block	(x?y:z)	x, y, z	result

Let's go over a brief example on each of these three nodes in action using the conditional "If" statement:



```
"This is the result if true";
Math.PI;
(x?y:z);
This is the result if true
This is the result if true
This is the result if true
```

In this image, the *boolean* is set to *true*, which means that the result is a string reading: *"this is the result if true"*. The three Nodes creating the *If* statement are working identically here.

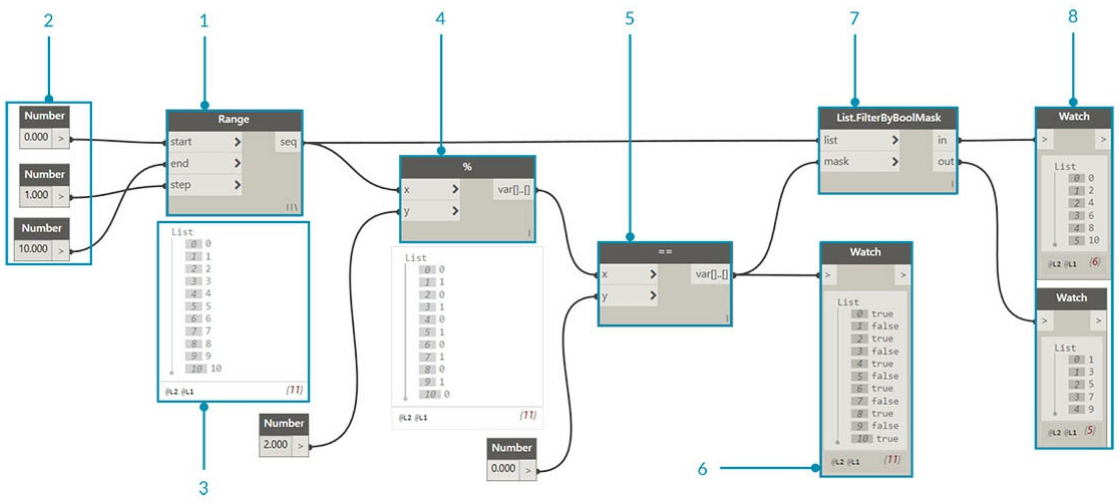


Again, the Nodes are working identically. If the *boolean* is changed to *false*, our result is the number *Pi*, as defined in the original *If* statement.

Filtering a List

Download the example file that accompanies this exercise (Right click and "Save Link As..."): [Building Blocks of Programs - Logic.dyn](). A full list of example files can be found in the Appendix.

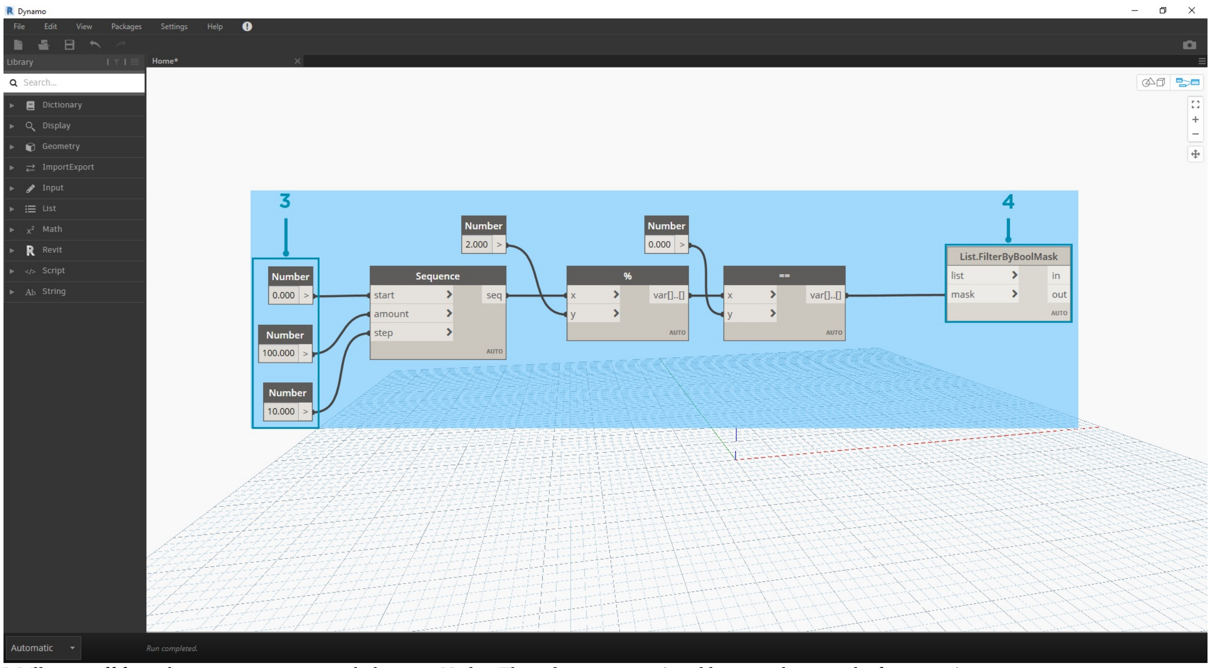
Let's use logic to separate a list of numbers into a list of even numbers and a list of odd numbers.



1. **Number Range -** add a number range to the canvas.
2. **Numbers -** add three number nodes to the canvas. The value for each number node should be: *0.0* for *start*, *10.0* for *end*, and *1.0* for *step*.
3. **Output -** our output is a list of 11 numbers ranging from 0-10.
4. **Modulo (%)-** *Number Range* into *x* and *2.0* into *y*. This calculates the remainder for each number in the list divided by 2. The output from this list gives us a list of values alternating between 0 and 1.
5. **Equality Test (==)** - add an equality test to the canvas. Plug *modulo* output into the *x* input and *0.0* into the *y* input.
6. **Watch -** The output of the equality test is a list of values alternating between true and false. These are the values used to separate the items in the list. *0* (or *true*) represents even numbers and (*1*, or *false*) represents odd numbers.
7. **List.FilterByBoolMask -** this Node will filter the values into two different lists based on the input boolean. Plug the original *number range* into the *list* input and the *equality test* output into the *mask* input. The *in* output represents true values while the *out* output represents false values.
8. **Watch -** as a result, we now have a list of even numbers and a list of odd numbers. We've used logical operators to separate lists into patterns!

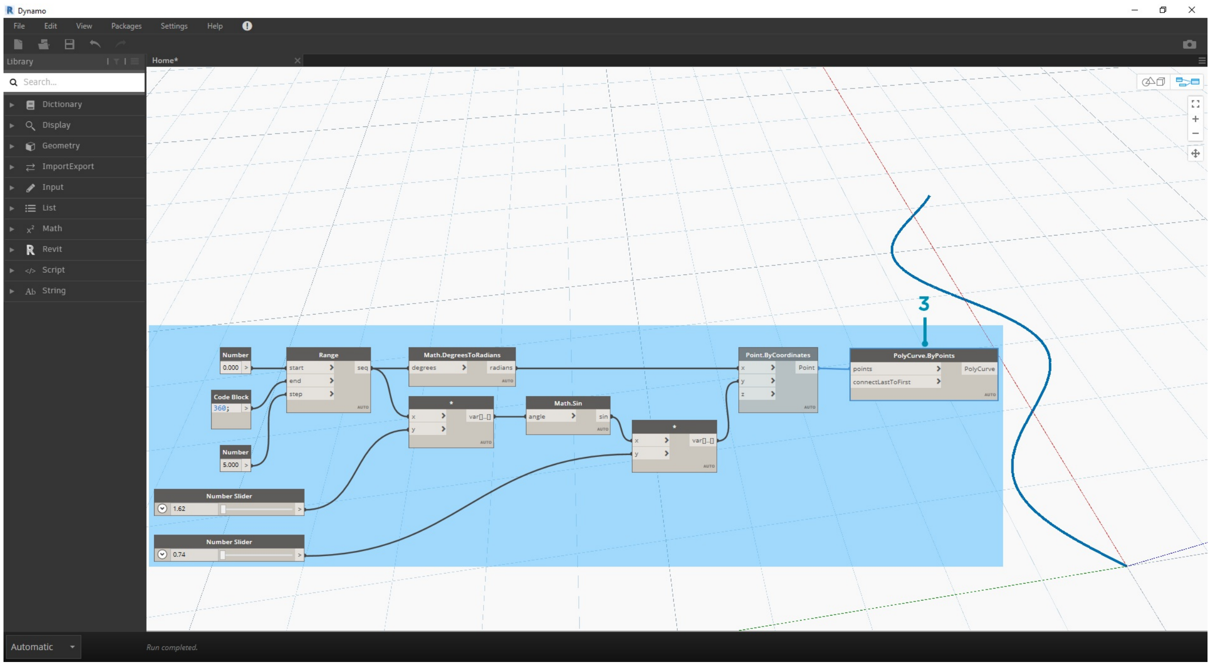
From Logic to Geometry

Building off of the logic established in the first exercise, let's apply this setup into a modeling operation.



We'll jump off from the previous exercise with the same Nodes. The only exceptions (in addition to changing the format are):

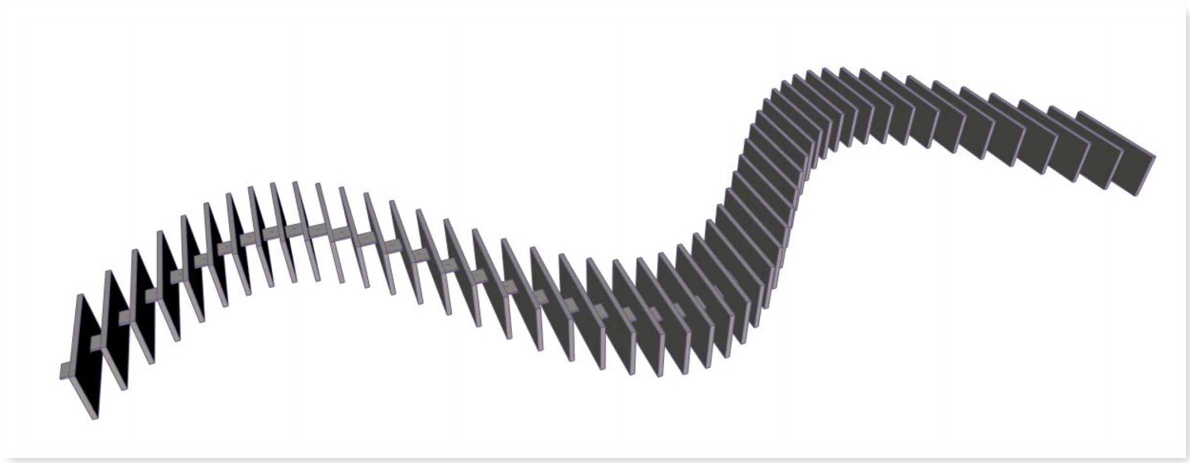
1. The input values have changed.
2. We've unplugged the in list input into *List.FilterByBoolMask*. We'll put these Nodes aside for now, but they'll come in handy later in the exercise.



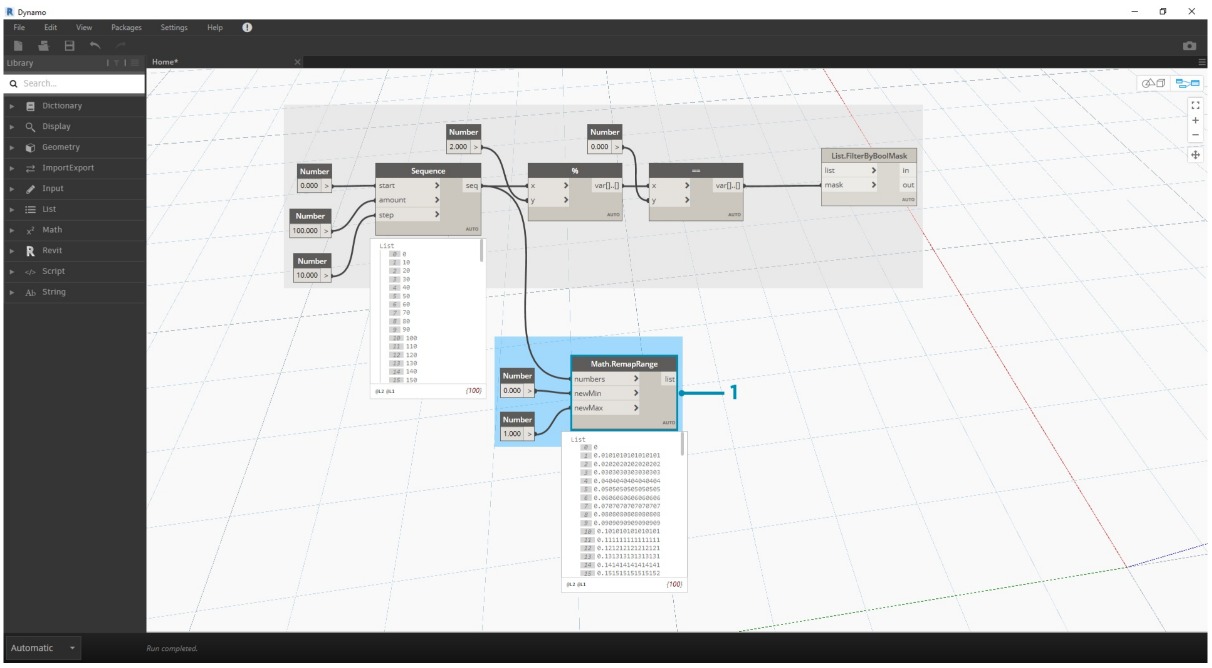
Let's begin by connecting the Nodes together as shown in the image above. This group of Nodes represents a parametric equation to define a line curve. A few notes:

1. The **first slider** should have a min of 1, a max of 4, and a step of 0.01.
2. The **second slider** should have a min of 0, a max of 1, and a step of 0.01.
3. **PolyCurve.ByPoints -** if the above Node diagram is copied, the result is a sine curve in the Dynamo Preview viewport.

The method here for the inputs: use number nodes for more static properties and number sliders on the more flexible ones. We want to keep the original number range that we're defining in the beginning of this step. However, the sine curve that we create here should have some flexibility. We can move these sliders to watch the curve update its frequency and amplitude.



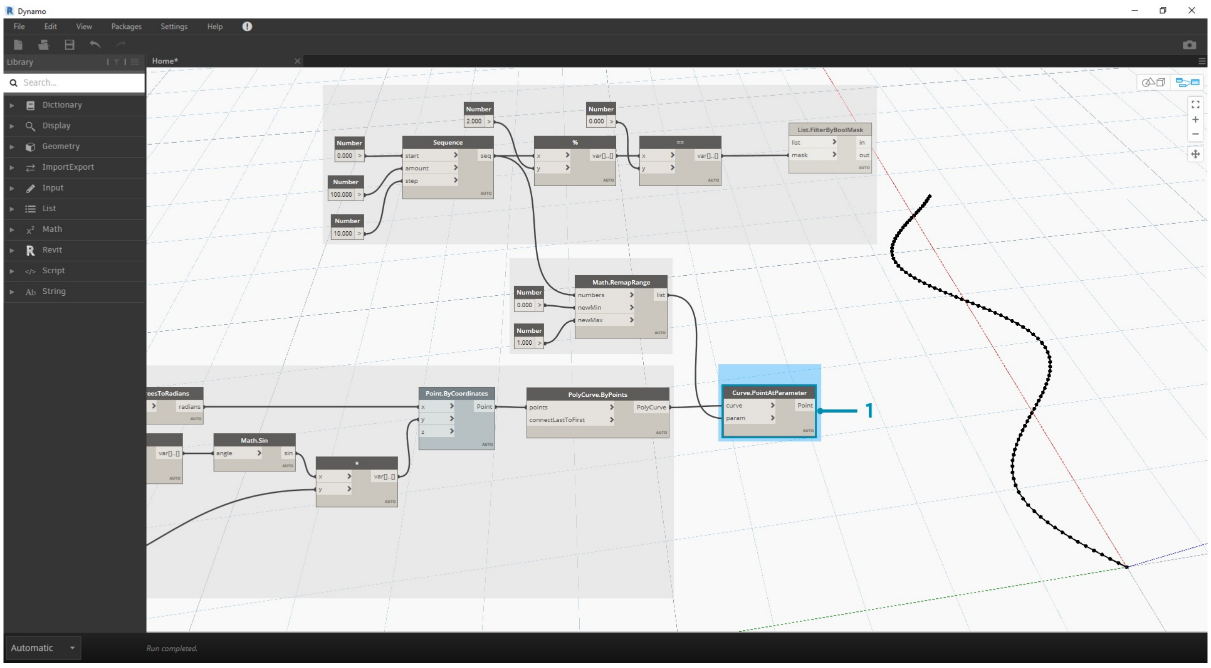
We're going to jump around a bit in the definition, so let's look at the end result so that we can reference what we're getting at. The first two steps are made separately, we now want to connect the two. We'll use the base sine curve to drive the location of the zipper components, and we'll use the true/false logic to alternate between little boxes and larger boxes.



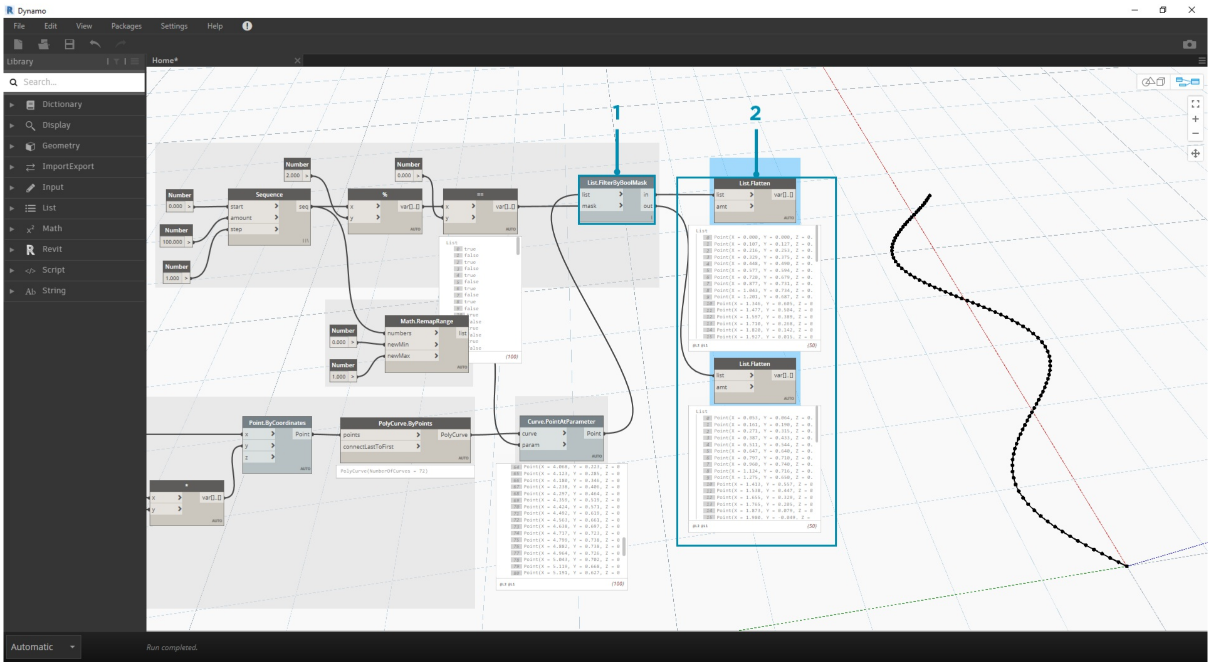
```
List
  0  0
  1  10
  2  20
  3  30
  4  40
  5  50
  6  60
  7  70
  8  80
  9  90
  10  100
  11  110
  12  120
  13  130
  14  140
  15  150
@L2 @L1                {100}
```

```
List
  0  0
  1  0.0101010101010101
  2  0.0202020202020202
  3  0.0303030303030303
  4  0.0404040404040404
  5  0.0505050505050505
  6  0.0606060606060606
  7  0.0707070707070707
  8  0.0808080808080808
  9  0.0909090909090909
  10  0.101010101010101
  11  0.111111111111111
  12  0.121212121212121
  13  0.131313131313131
  14  0.141414141414141
  15  0.151515151515152
@L2 @L1                {100}
```

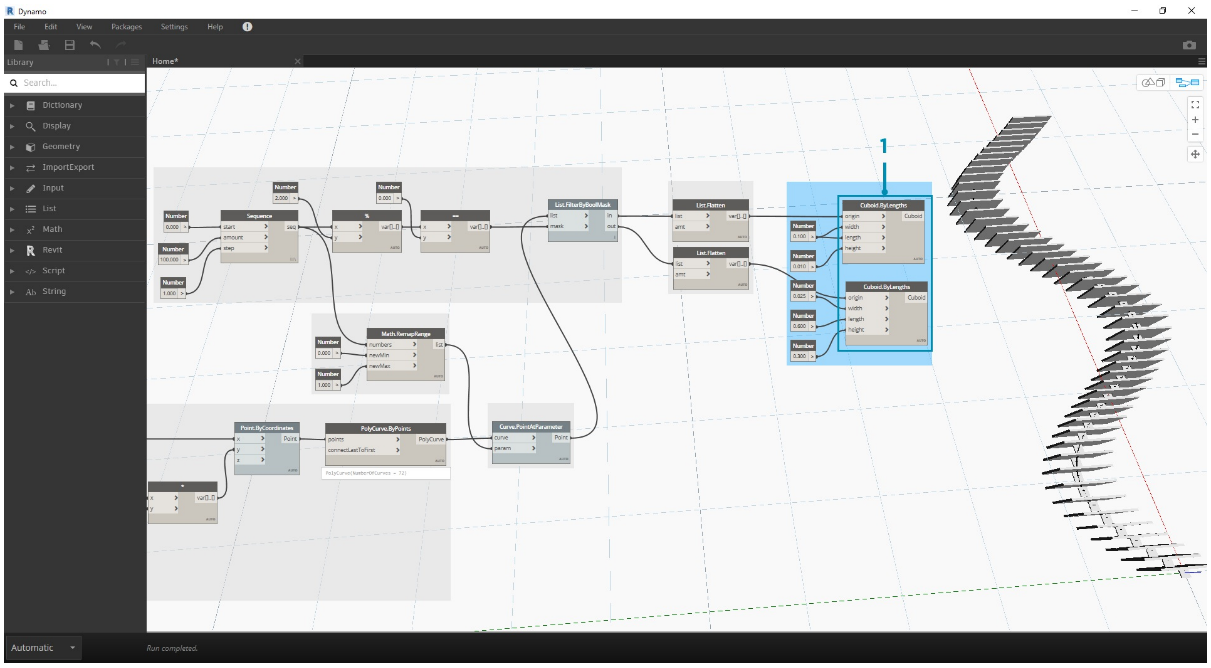
1. **Math.RemapRange -** Using the number sequence created in step 01, let's create a new series of numbers by remapping the range. The original numbers from step 01 range from 0-100. These numbers range from 0 to 1 by the *newMin* and *newMax* inputs respectively.



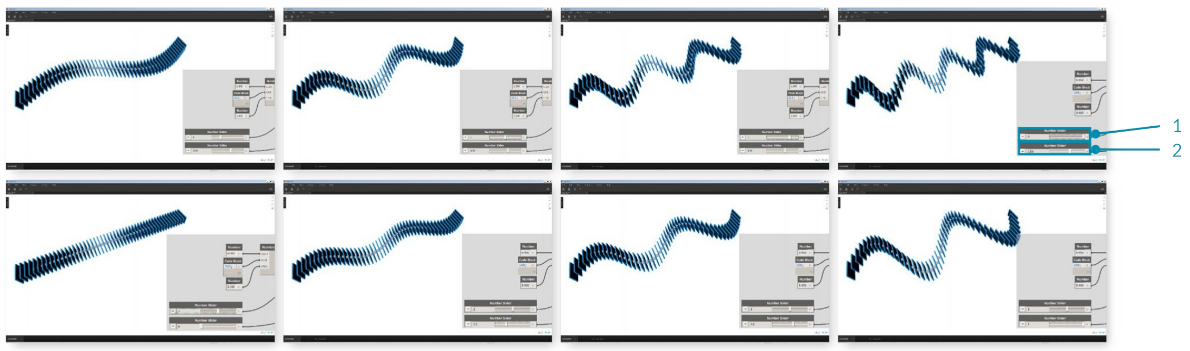
1. **Curve.PointAtParameter -** Plug *Polycurve.ByPoints* (from step 2) into *curve* and *Math.RemapRange* into *param*. This step creates points along the curve. We remapped the numbers to 0 to 1 because the input of *param* is looking for values in this range. A value of *0* represents the start point, a value of *1* represents the end points. All numbers in between evaluate within the *[0,1]* range.



1. **List.FilterByBoolMask -** Plug *Curve.PointAtParameter* from the previous step into the *list* input.
2. **Watch -** a watch node for *in* and a watch node for *out* shows that we have two lists representing even indices and odd indices. These points are ordered in the same way on the curve, which we demonstrate in the next step.



1. **Cuboid.ByLengths -** recreate the connections seen in the image above to get a zipper along the sine curve. A cuboid is just a box here, and we're defining its size based on the curve point in the center of the box. The logic of the even/odd divide should now be clear in the model.



1. **Number Slider -** stepping back to the beginning of the definition, we can flex the number slider and watch the zipper update. The top row of images represents a range values for the top number slider. This is the frequency of the wave.
2. **Number Slider -** the bottom row of images represents a range of values for the bottom slider. This is the amplitude of the wave.

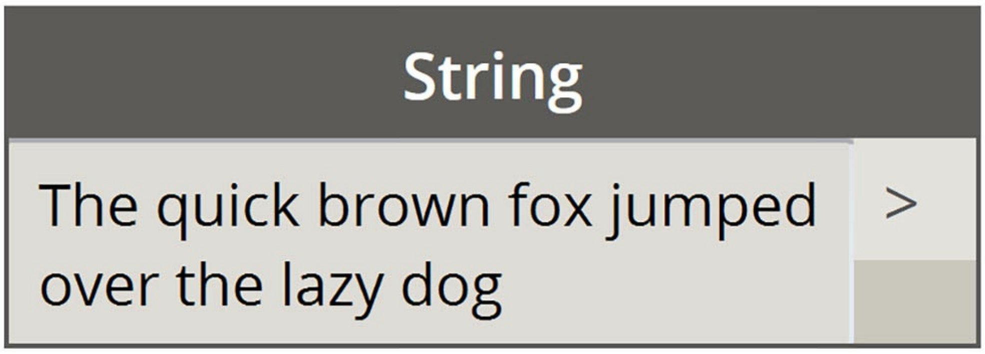
Strings

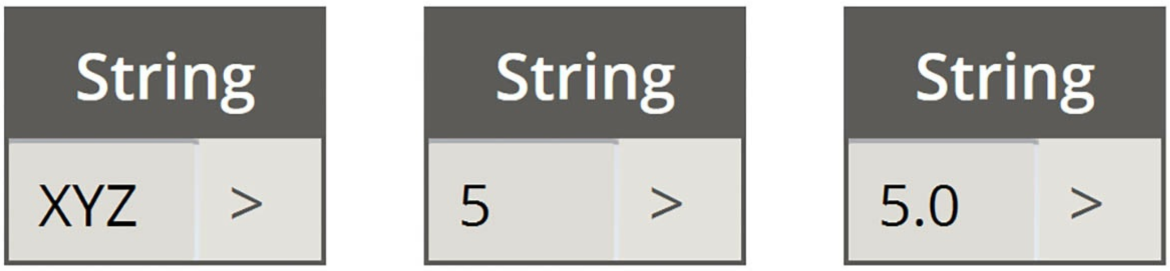
Strings

Formally, a **String** is a sequence of characters representing a literal constant or some type of variable. Informally, a string is programming lingo for text. We've worked with numbers, both integers and decimal numbers, to drive parameters and we can do the same with text.

Creating Strings

Strings can be used for a wide range of applications, including defining custom parameters, annotating documentation sets, and parsing through text-based data sets. The string Node is located in the Core>Input Category.





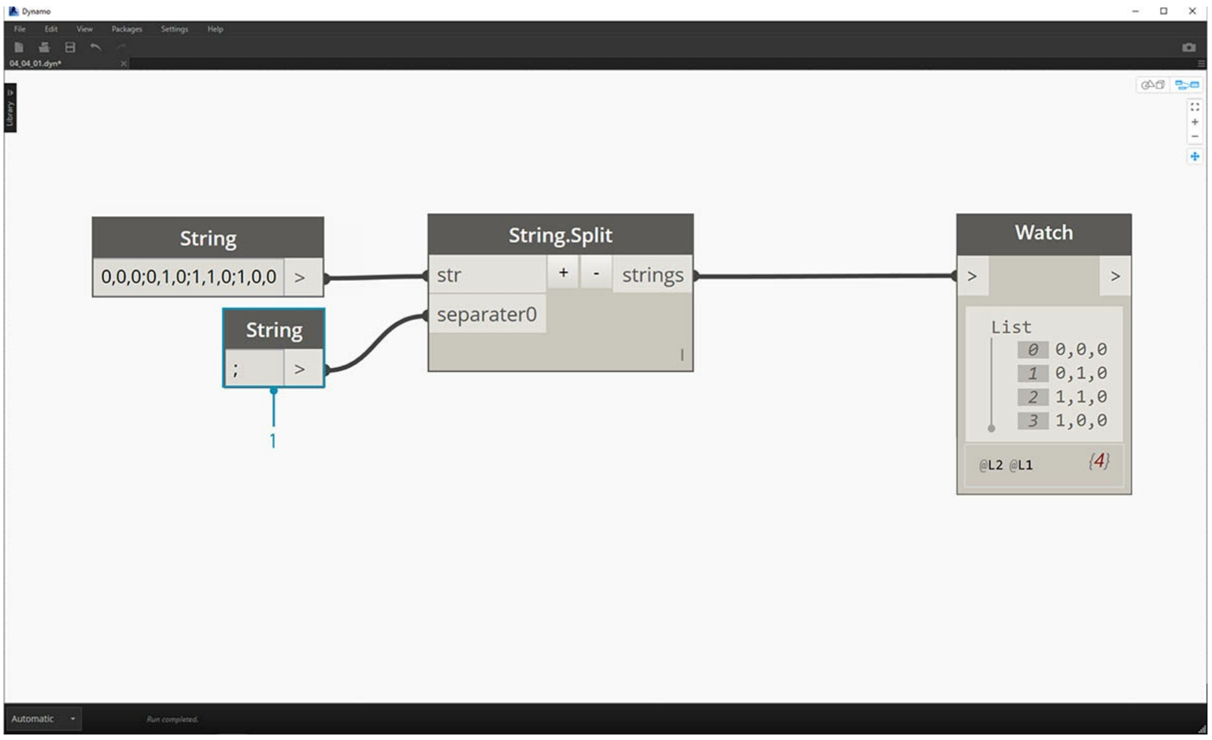
The sample Nodes above are strings. A number can be represented as a string, as can a letter, or an entire array of text.

Querying Strings

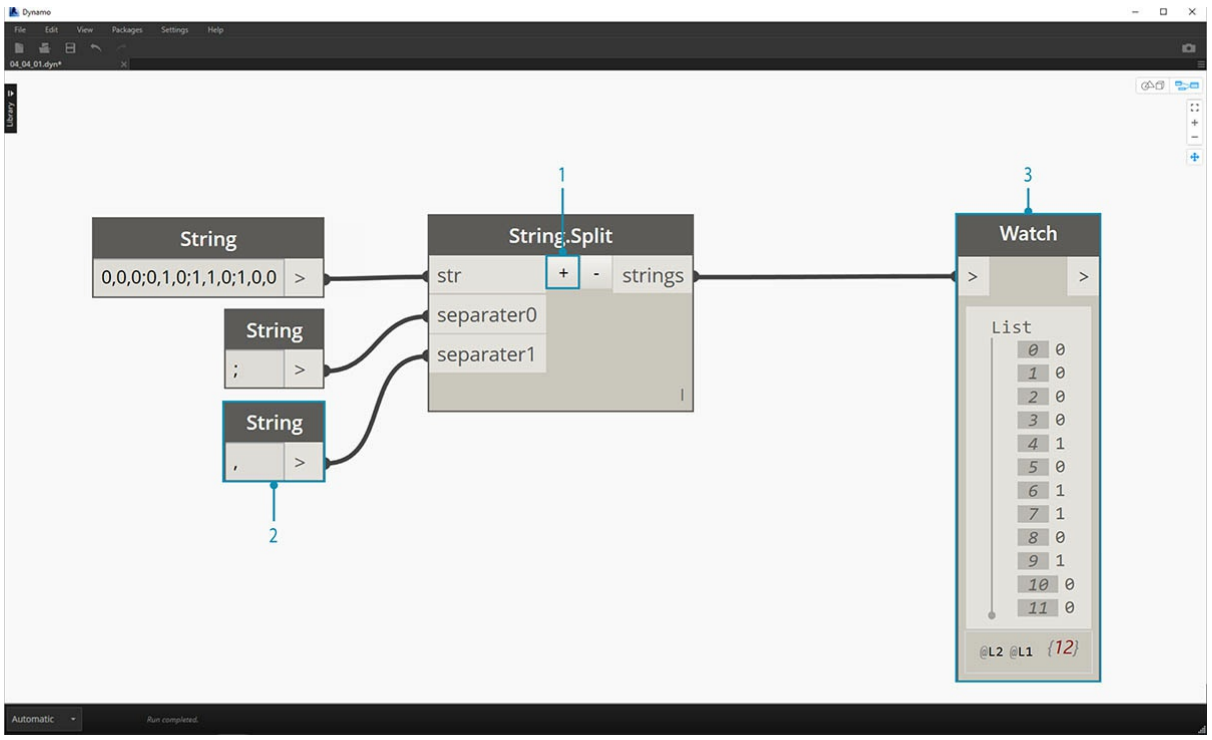
Download the example file that accompanies this exercise (Right click and "Save Link As..."): Building Blocks of Programs - Strings.dyn. A full list of example files can be found in the Appendix.

You can parse through large amounts of data quickly by querying strings. We'll talk about some basic operations which can speed up a workflow and help for software interoperability.

The image below considers a string of data coming from an external spreadsheet. The string represents the vertices of a rectangle in the XY-Plane. Let's break down some string split operations in miniature exercise:

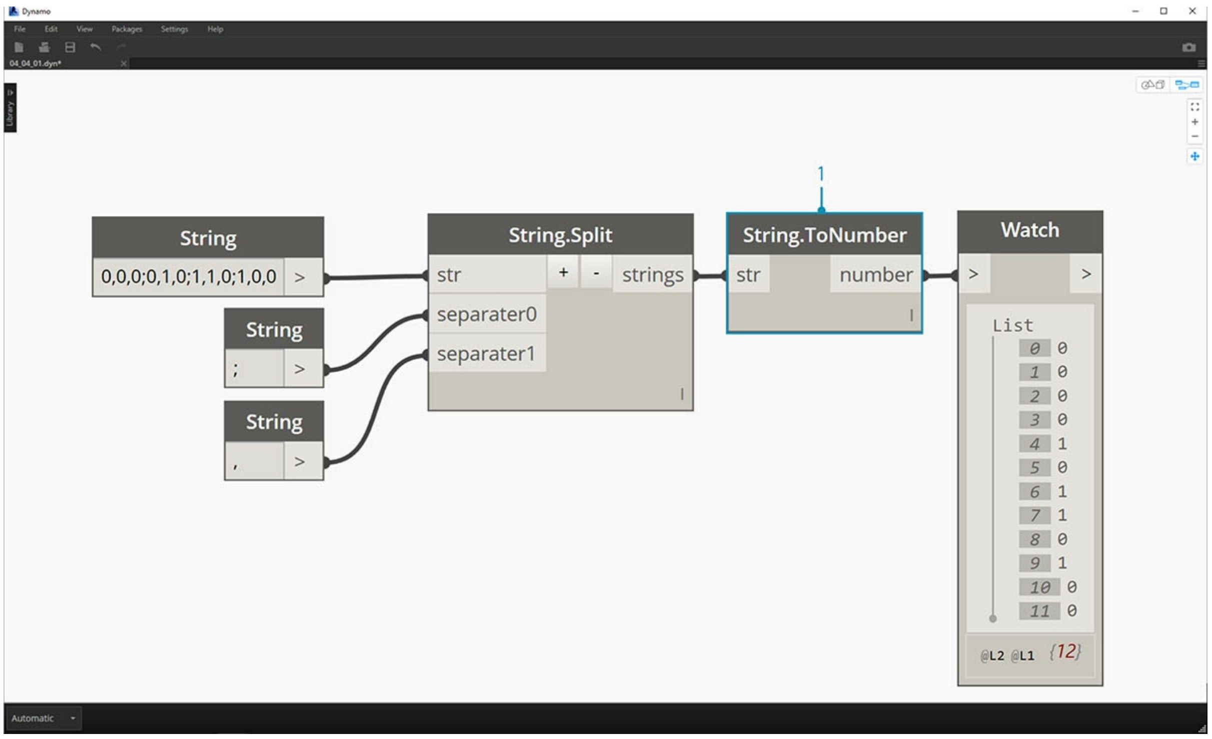


1. The ";" separator splits each vertex of the rectangle. This creates a list with 4 items for each vertex.

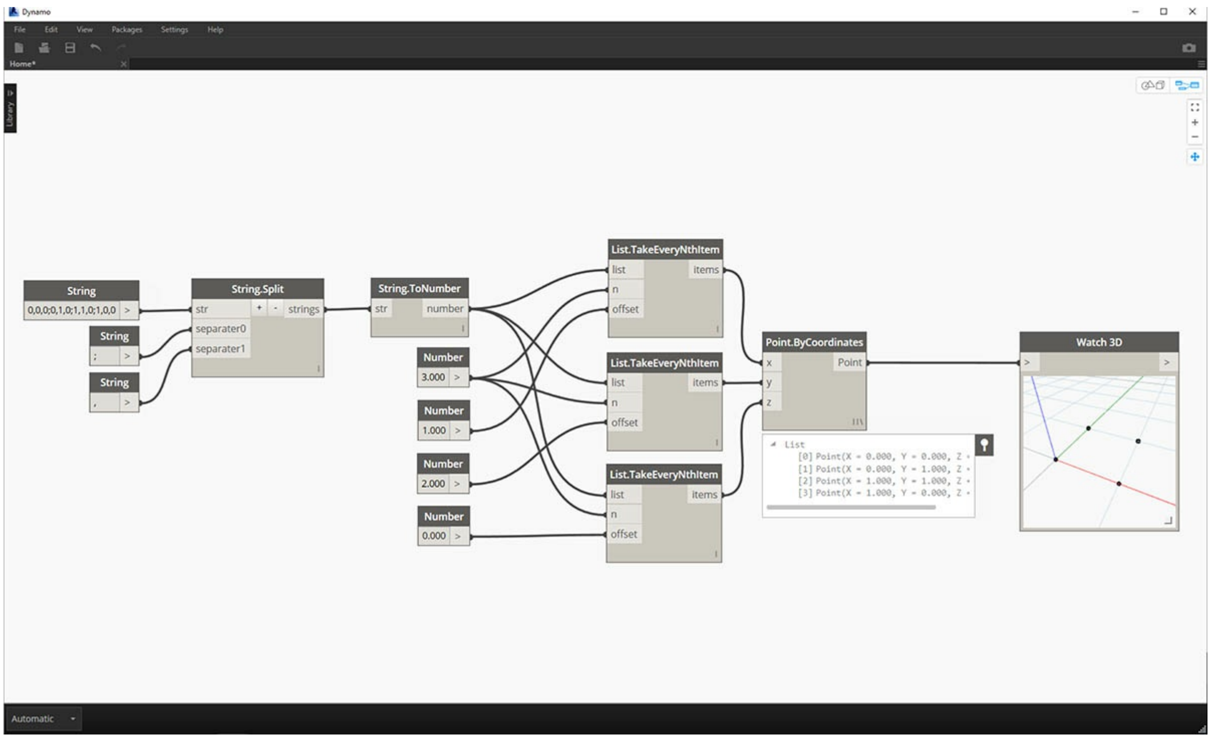


1. By hitting the "+" in the middle of the Node, we create new separator.
2. Add a "," string to the canvas and plug in to the new separator input.
3. Our result is now a list of ten items. The Node first splits based on *separator0*, then based on *separator1*.

While the list of items above may look like numbers, they are still regarded as individual strings in Dynamo. In order to create points, their data type needs to be converted from a string to a Number. This is done with the String.ToNumber Node



1. This Node is straightforward. Plug the String.Split results into the input. The output doesn't look different, but the data type is now a *number* instead of a *string*.



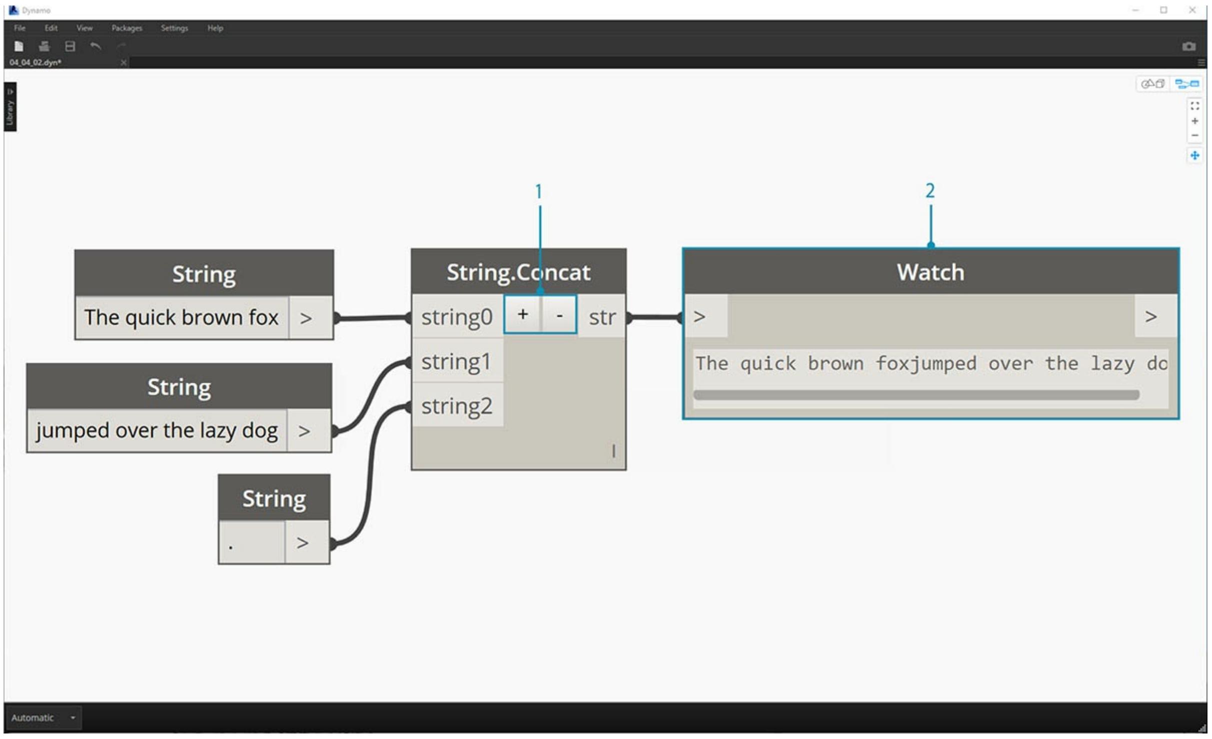
```
List
  [0] Point(X = 0.000, Y = 0.000, Z =
  [1] Point(X = 0.000, Y = 1.000, Z =
  [2] Point(X = 1.000, Y = 1.000, Z =
  [3] Point(X = 1.000, Y = 0.000, Z =
```

1. With some basic additional operations, we now have a rectangle drawn at the origin based on the original string input.

Manipulating Strings

Since a string is a generic text object, they host a wide range of applications. Let's take a look at some of the major actions in the Core>String Category in Dynamo:

This is a method of merging two strings together in order. This takes each literal string in a list and creates one merged string.

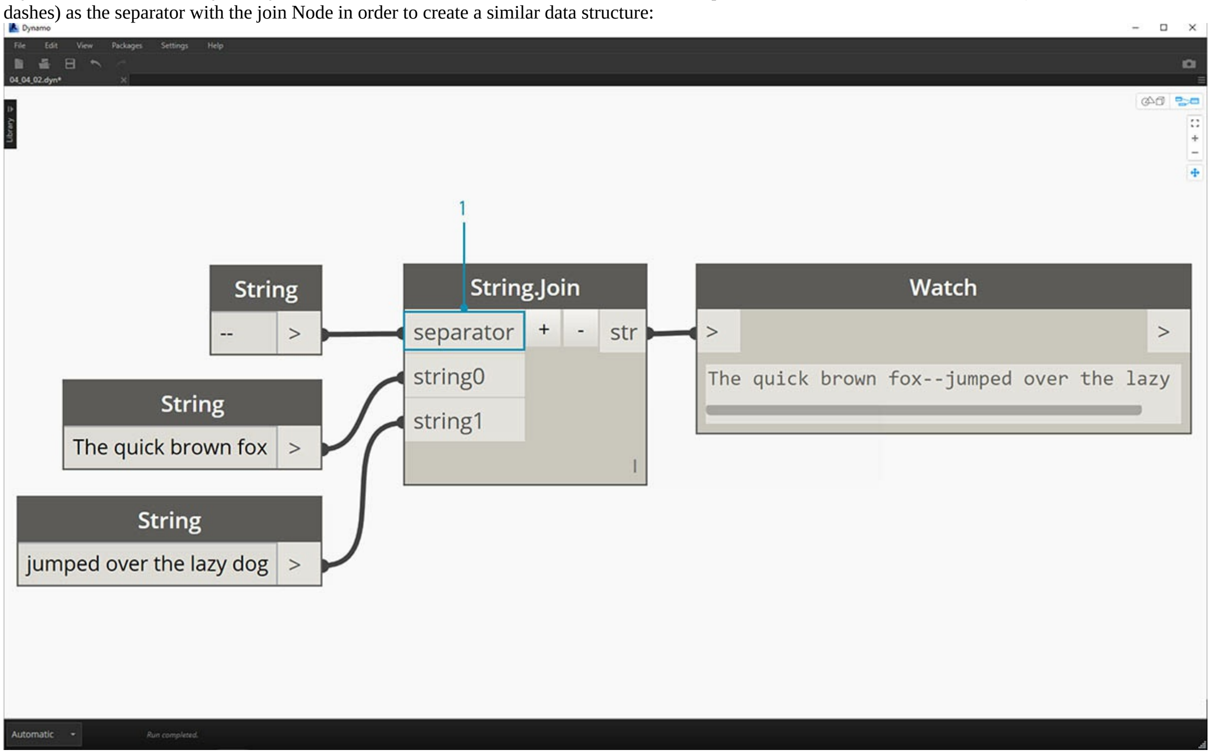


The image above represents the concatenation of three strings:

1. Add or subtract strings to the concatenation by clicking the +/- buttons in the center of the Node.
2. The output gives one concatenated string, with spaces and punctuation included.

The join method is very similar to concatenate, except it has an added layer of punctuation.

If you've worked in Excel, you may have come across a CSV file. This stands for comma-separated values. One could use a comma (or in this case, two dashes) as the separator with the join Node in order to create a similar data structure:

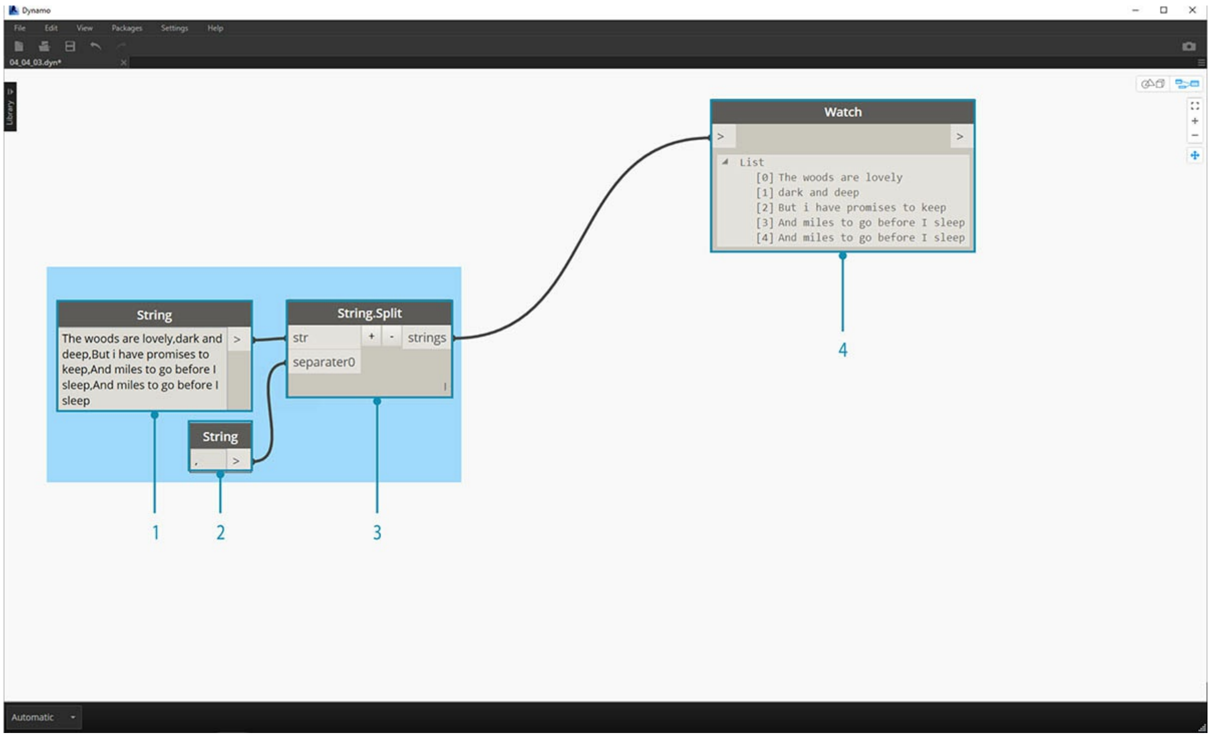


The image above represents the joining of two strings:

1. The separator input allows one to create a string which divides the joined strings.

Working with Strings

In this exercise, we're going to use methods of querying and manipulating strings to deconstruct the final stanza of Robert Frost's [Stopping By Woods on a Snowy Evening](#). Not the most practical application, but it will help us to grasp conceptual string actions as we apply them to legible lines of rhythm and rhyme.

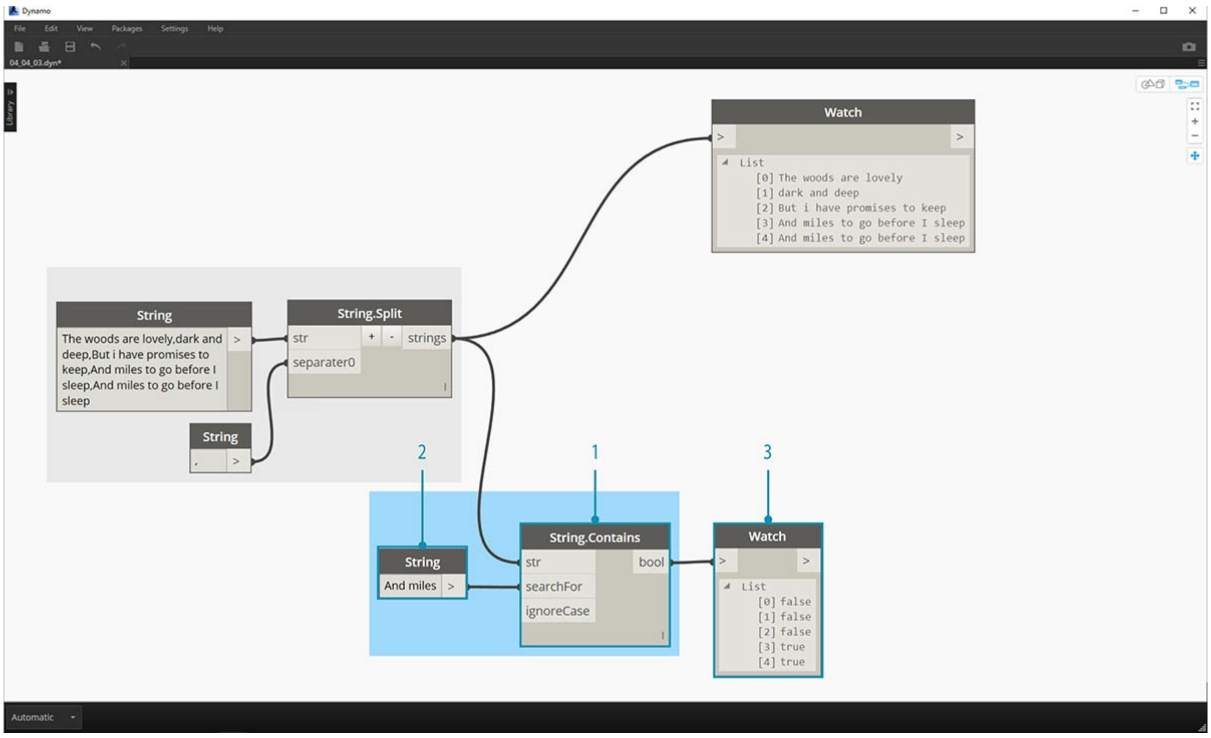


```
The woods are lovely,dark and deep,But i have promises to keep,And miles to go before I sleep,And miles to go before I sleep
```

```
List
    [0] The woods are lovely
    [1] dark and deep
    [2] But i have promises to keep
    [3] And miles to go before I sleep
    [4] And miles to go before I sleep
```

Let's begin with a basic string split of the stanza. We first notice that the writing is formatted based on commas. We'll use this format to separate each line into individual items.

1. The base string is pasted into a string node.
2. Another string node is used to denote the separator. In this case, we're using a comma.
3. A String.Split Node is added to the canvas and connected to the two strings.
4. The output shows that we've now separated the lines into individual elements.

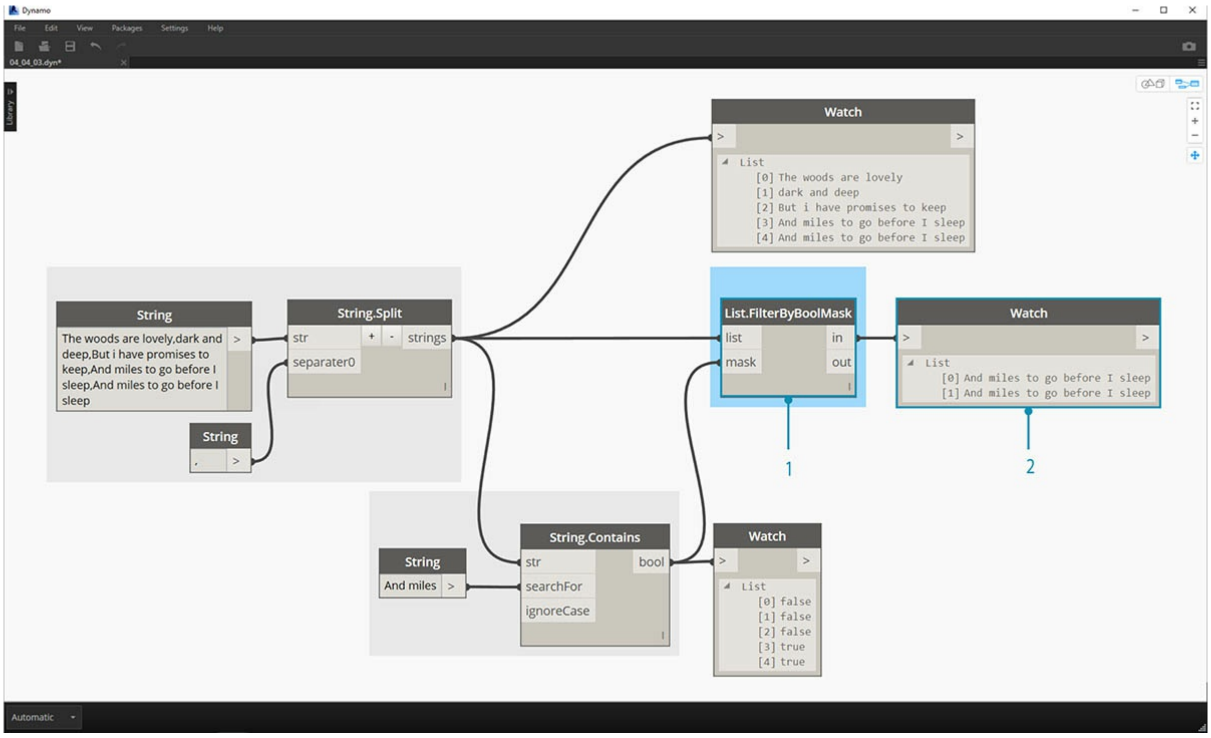


```
List
    [0] false
    [1] false
    [2] false
    [3] true
    [4] true
```

Now, let's get to the good part of the poem: the last two lines. The original stanza was one item of data. We separated this data into individual items in the first step. Now we need to do a search for the text we're looking for. And while we *can* do this by selecting the last two items of the list, if this were an entire book, we wouldn't want to read through everything and manually isolate the elements.

1. Instead of manually searching, we use a String.Contains Node to perform a search for a set of characters. This is the similar to doing the "Find" command in a word processor. In this case, we get a return of "true" or "false" if that substring is found within the item.
2. In the "searchFor" input, we define a substring that we're looking for within the stanza. Let's use a string node with the text "And miles".

3. The output gives us a list of falses and trues. We'll use this boolean logic to filter the elements in the next step.



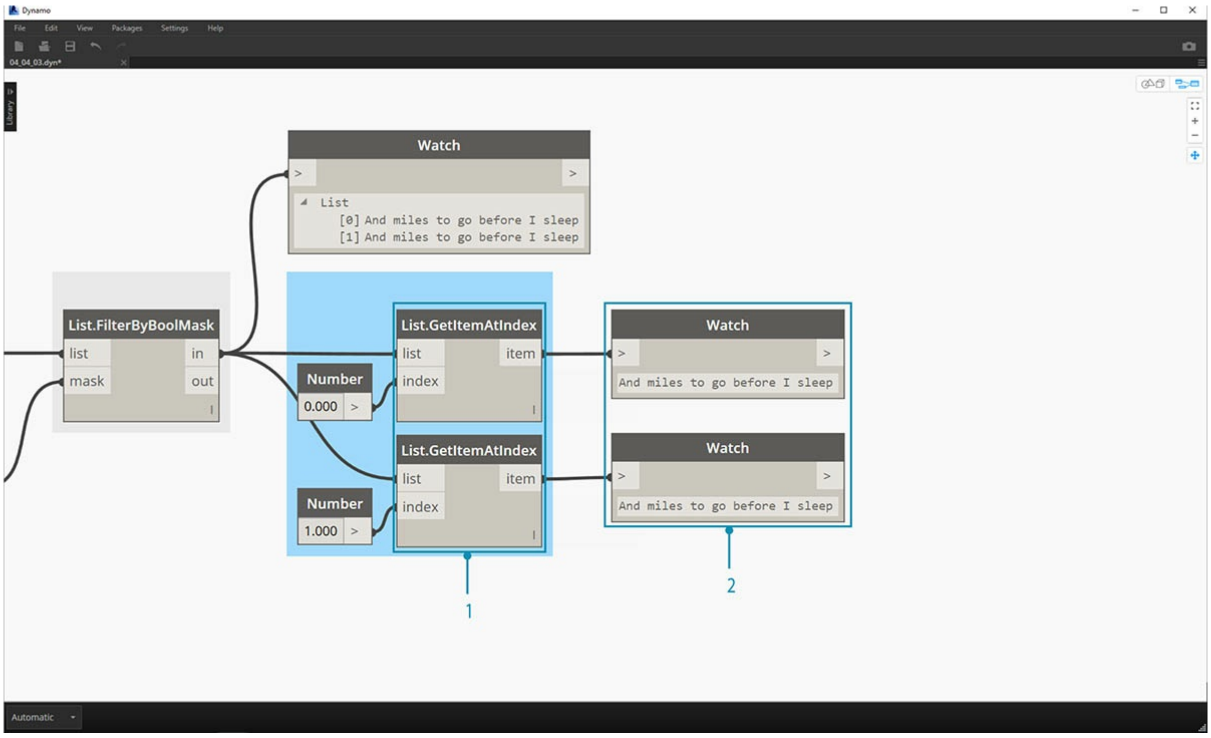
String: "The woods are lovely,dark and deep,But i have promises to keep,And miles to go before I sleep,And miles to go before I sleep"

```
List
  [0] The woods are lovely
  [1] dark and deep
  [2] But i have promises to keep
  [3] And miles to go before I sleep
  [4] And miles to go before I sleep
```

```
List
  [0] And miles to go before I sleep
  [1] And miles to go before I sleep
```

```
List
  [0] false
  [1] false
  [2] false
  [3] true
  [4] true
```

1. List.FilterByBoolMask is the Node we want to use to cull out the falses and trues. The "in" output return the statements with a "mask" input of "true, while the "out" output return those which are "false".
2. Our output from the "in" is as expected, giving us the final two lines of the stanza.



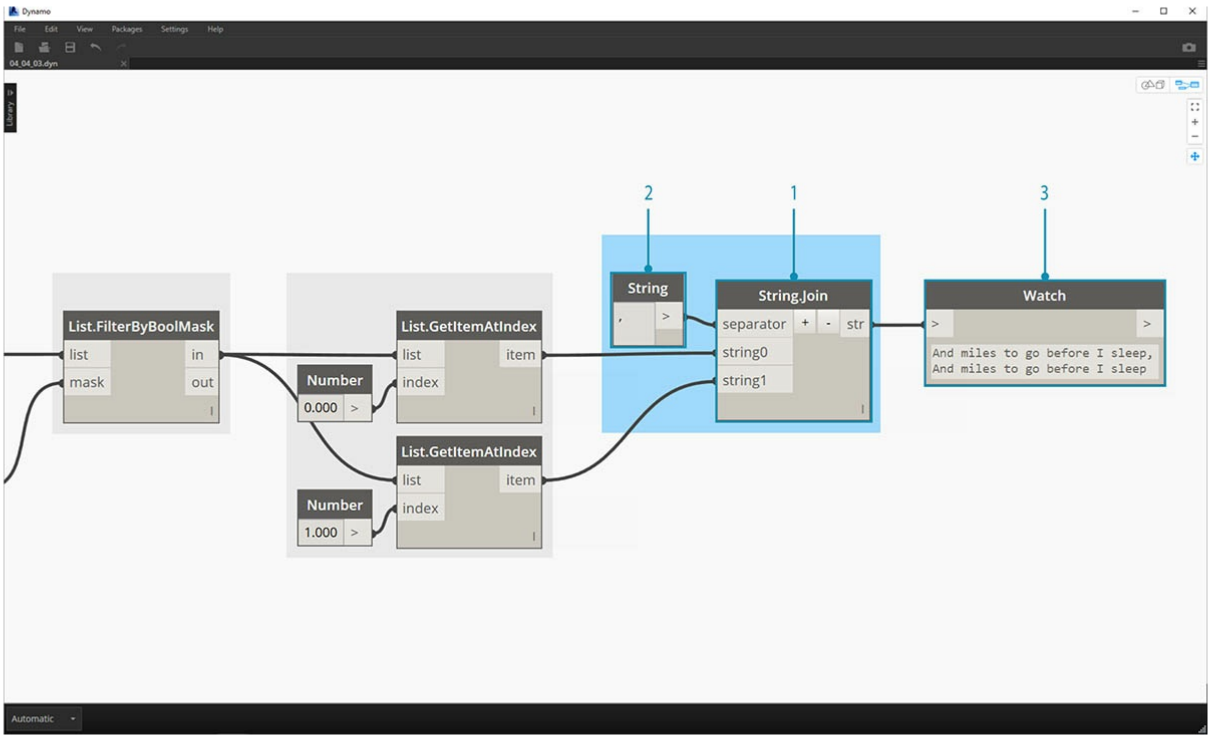
```
List
  [0] And miles to go before I sleep
  [1] And miles to go before I sleep
```

```
And miles to go before I sleep
```

```
And miles to go before I sleep
```

Now, we want to drive home the repetition of the stanza by merging the two lines together. When viewing the output of the previous step, we notice that there are two items in the list:

1. Using two List.GetItemAtIndex Nodes, we can isolate the items using the values of 0 and 1 as the index input.
2. The output for each Node gives us, in order, the final two lines.



```
And miles to go before I sleep,
And miles to go before I sleep
```

To merge these two items into one, we use the String.Join Node:

1. After adding the String.Join Node, we notice that we need a separator.
2. To create the separator, we add a string node to the canvas and type in a comma.
3. The final output has merged the last two items into one.

This may seem like a lot of work to isolate the last two lines; and it's true, string operations often require some up front work. But they are scalable, and they can be applied to large datasets with relative ease. If you are working parametrically with spreadsheets and interoperability, be sure to keep string operations in mind.

Color

Color

Color is a great data type for creating compelling visuals as well as for rendering difference in the output from your Visual Program. When working with abstract data and varying numbers, sometimes it's difficult to see what's changing and to what degree. This is a great application for colors.

Creating Colors

Colors in Dynamo are created using ARGB inputs.This corresponds to the Alpha, Red, Green, and Blue channels. The alpha represents the *transparency* of the color, while the other three are used as primary colors to generate the whole spectrum of color in concert.

Icon	Name	Syntax	Inputs	Outputs
	ARGB Color	Color.ByARGB	A,R,G,B	color

Querying Color Values

The colors in the table below query the properties used to define the color: Alpha, Red, Green, and Blue. Note that the Color.Components Node gives us all four as different outputs, which makes this Node preferable for querying the properties of a color.

Icon	Name	Syntax	Inputs	Outputs
◐	Alpha	Color.Alpha	color	A
🔴	Red	Color.Red	color	R
🟢	Green	Color.Green	color	G
🔵	Blue	Color.Blue	color	B
	Components	Color.Components	color	A,R,G,B

The colors in the table below correspond to the **HSB color space**. Dividing the color into hue, saturation, and brightness is arguably more intuitive for how we interpret color: What color should it be? How colorful should it be? And how light or dark should the color be? This is the breakdown of hue, saturation, and brightness respectively.

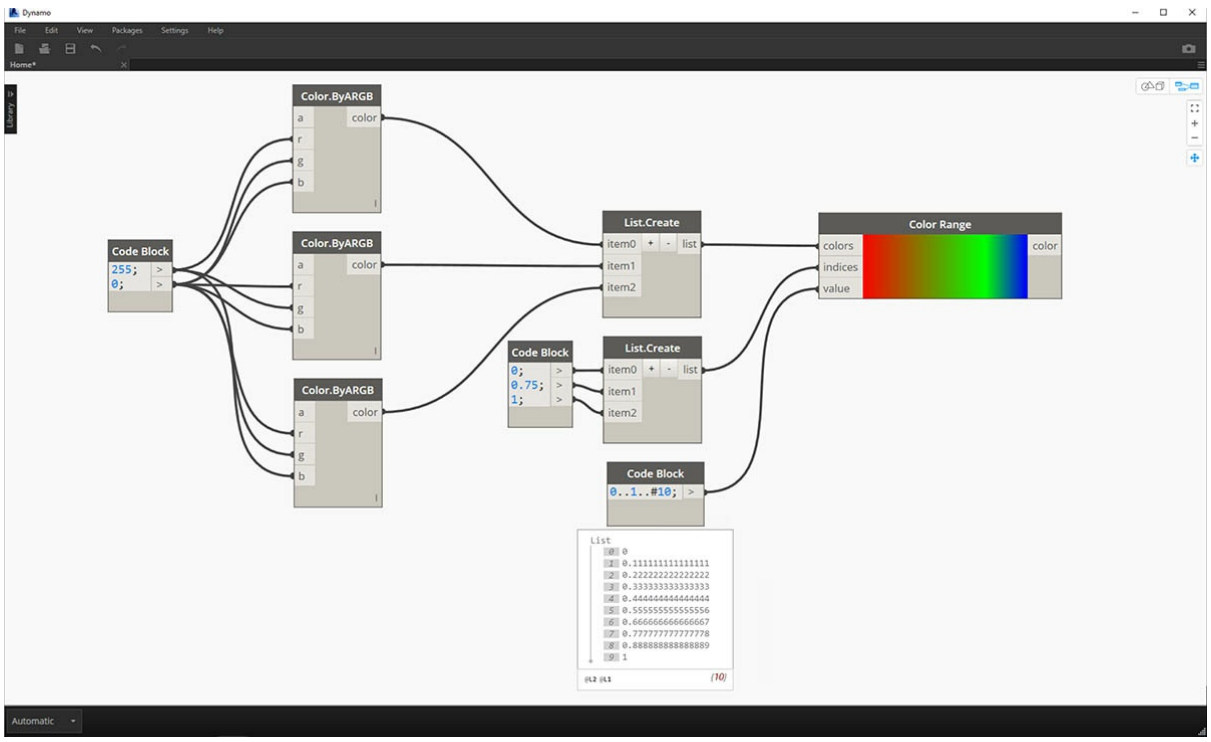
Icon	Query Name	Syntax	Inputs	Outputs
	Hue	Color.Hue	color	Hue

◐	Saturation	Color.Saturation	color	Saturation
☼	Brightness	Color.Brightness	color	Brightness

Color Range

The color range is similar to the **Remap Range** Node from section 4.2: it remaps a list of numbers into another domain. But instead of mapping to a *number* domain, it maps to a *color gradient* based on input numbers ranging from 0 to 1.

The current Node works well, but it can be a little awkward to get everything working the first time around. The best way to become familiar with the color gradient is to test it out interactively. Let's do a quick exercise to review how to setup a gradient with output colors corresponding to numbers.



```
255;
0;
```

```
0;
0.75;
1;
```

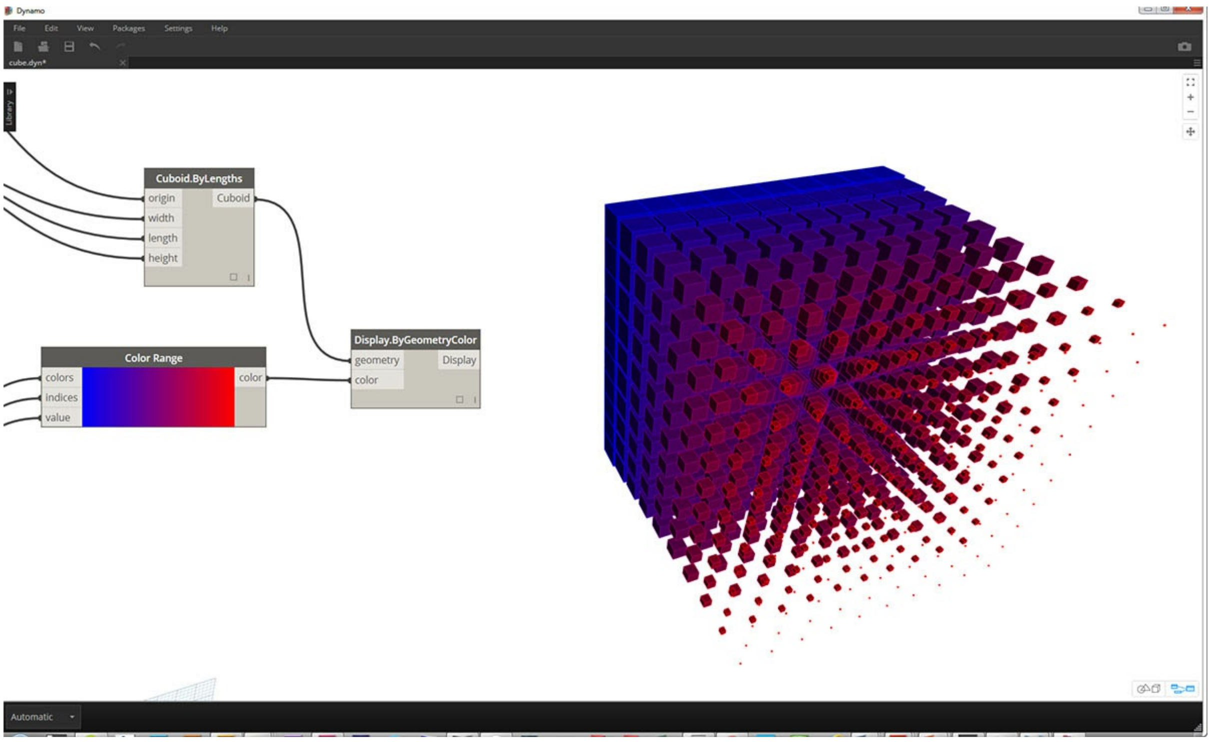
```
0..1..#10;
```

```
List
  0 0
  1 0.111111111111111
  2 0.222222222222222
  3 0.333333333333333
  4 0.444444444444444
  5 0.555555555555556
  6 0.666666666666667
  7 0.777777777777778
  8 0.888888888888889
  9 1
@L2 @L1                (10)
```

1. **Define three colors:** Using a code block node, define *red*, *green*, and *blue* by plugging in the appropriate combinations of *0* and *255*.
2. **Create list:** Merge the three colors into one list.
3. **Define Indices:** Create a list to define the grip positions of each color (ranging from 0 to 1). Notice the value of 0.75 for green. This places the green color 3/4 of the way across the horizontal gradient in the color range slider.
4. **Code Block:** Input values (between 0 and 1) to translate to colors.

Color Preview

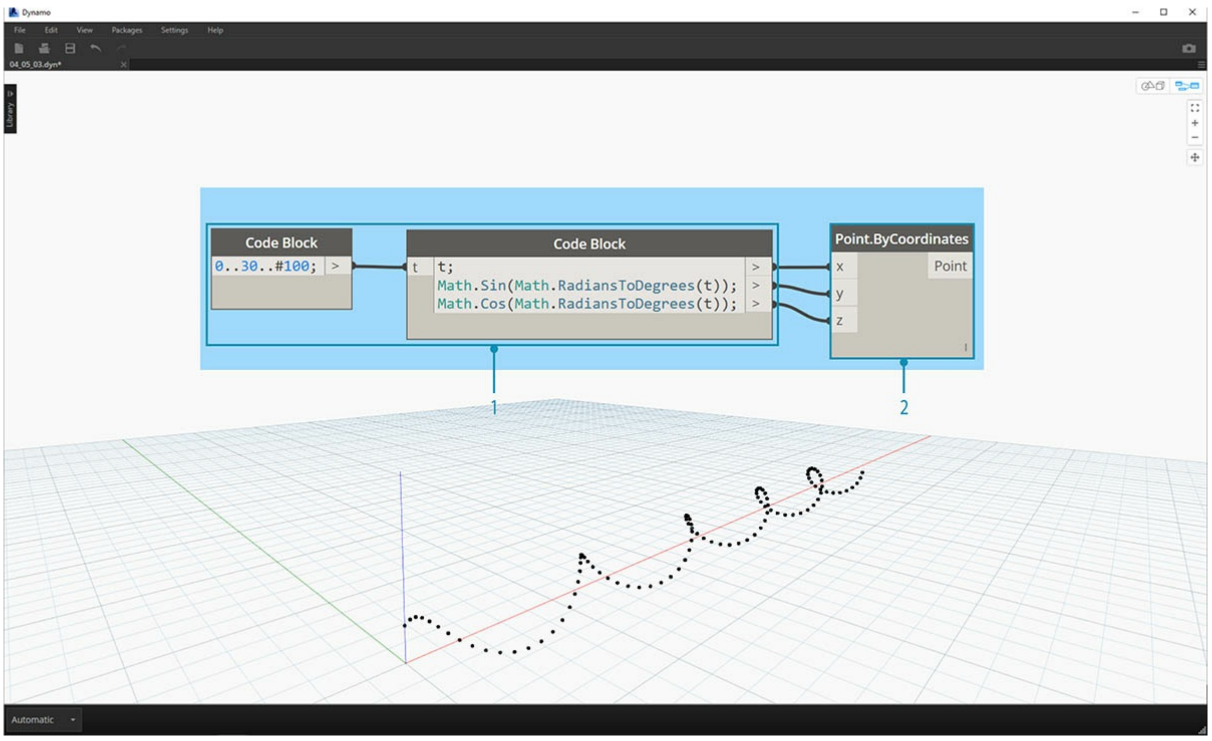
The **Display.ByGeometry** Node gives us the ability to color geometry in the Dynamo viewport. This is helpful for separating different types of geometry, demonstrating a parametric concept, or defining an analysis legend for simulation. The inputs are simple: geometry and color. To create a gradient like the image above, the color input is connected to the **color range** Node.



Color Exercise

> Download the example file that accompanies this exercise (Right click and "Save Link As..."): Building Blocks of Programs - Color.dyn. A full list of example files can be found in the Appendix.

This exercise focuses on controlling color parametrically in parallel with geometry. The geometry is a basic helix, which we define below using the **Code Block** (3.2.3). This is a quick and easy way to create a parametric function; and since our focus is on color (rather than geometry), we use the code block to efficiently create the helix without cluttering the canvas. We will use the code block more frequently as the primer moves to more advanced material.

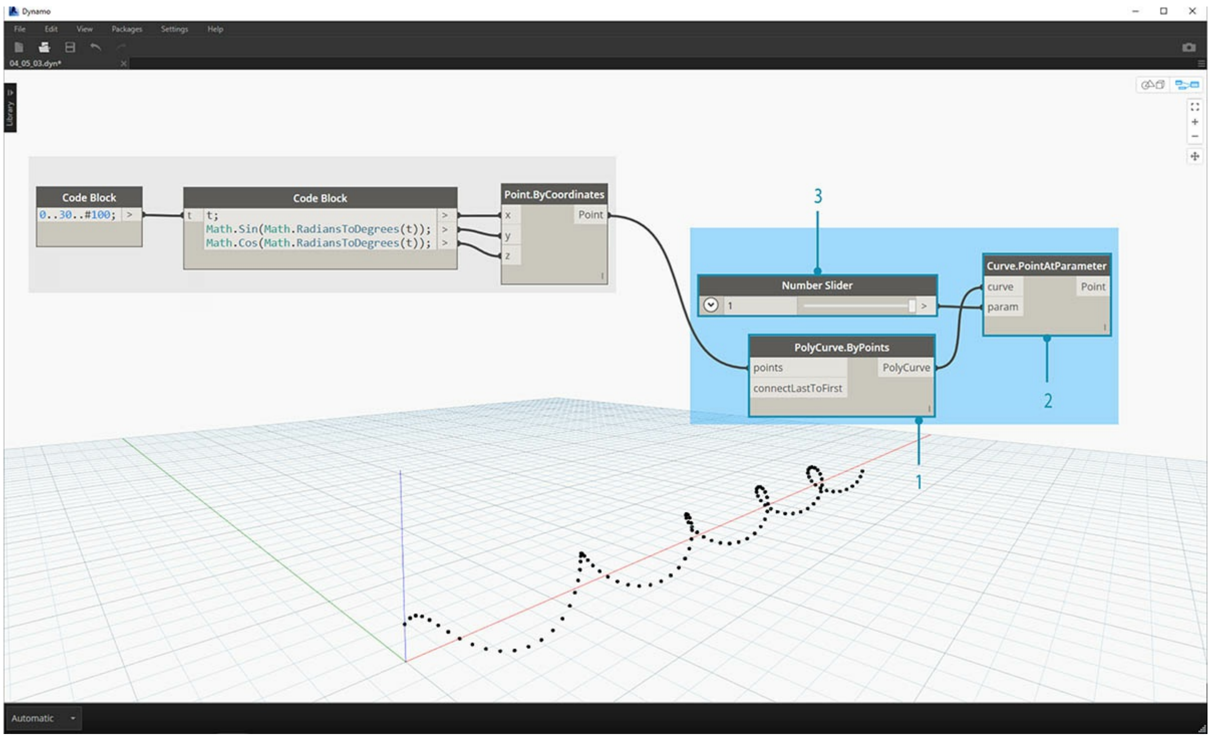


```
0..30..#100;
```

```
t;
Math.Sin(Math.RadiansToDegrees(t));
Math.Cos(Math.RadiansToDegrees(t));
```

1. **Code Block:** Define the two code blocks with the formulas above. This is a quick parametric method for creating a spiral.
2. **Point.ByCoordinates:**Plug the three outputs from the code block into the coordinates for the Node.

We now see an array of points creating a helix. The next step is to create a curve through the points so that we can visualize the helix.

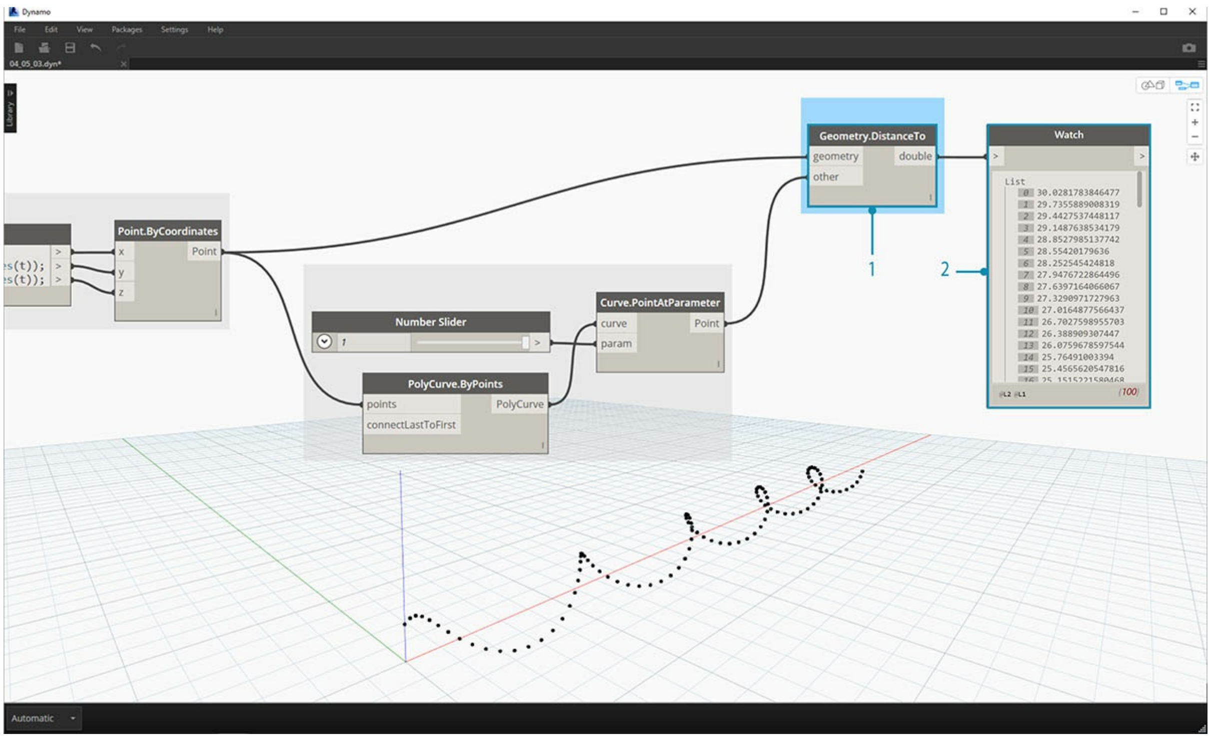


```
0..30..#100;
```

```
t;
Math.Sin(Math.RadiansToDegrees(t));
Math.Cos(Math.RadiansToDegrees(t));
```

1. **PolyCurve.ByPoints:** Connect the *Point.ByCoordinates* output into the *points* input for the Node. We get a helical curve.
2. **Curve.PointAtParameter:** Connect the *PolyCurve.ByPoints* output into the *curve* input. The purpose of this step is to create a parametric attractor point which slides along the curve. Since the curve is evaluating a point at parameter, we'll need to input a *param* value between 0 and 1.
3. **Number Slider:** After adding to the canvas, change the *min* value to *0.0*, the *max* value to *1.0*, and the *step* value to *.01*. Plug the slider output into the *param* input for *Curve.PointAtParameter*. We now see a point along the length of the helix, represented by a percentage of the slider (0 at the start point, 1 at the end point).

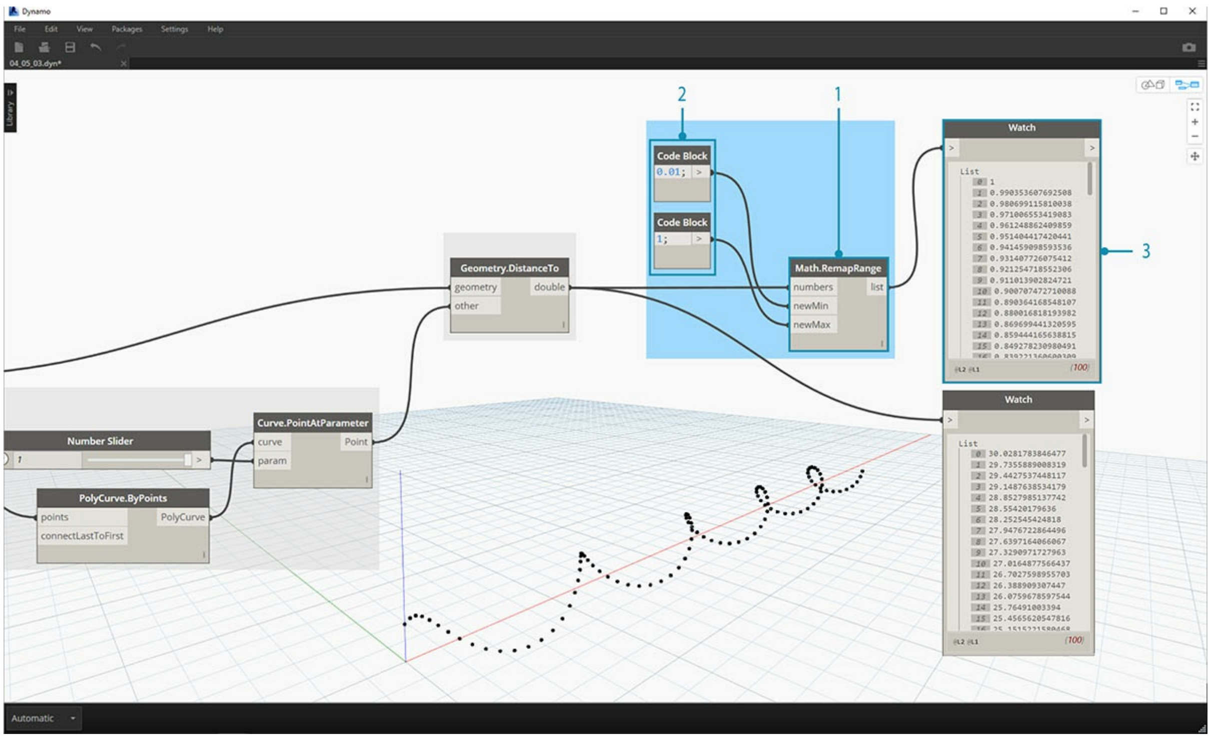
With the reference point created, we now compare the distance from the reference point to the original points defining the helix. This distance value will drive geometry as well as color.



```
List
  0 30.0281783846477
  1 29.7355889008319
  2 29.4427537448117
  3 29.1487638534179
  4 28.8527985137742
  5 28.55420179636
  6 28.252545424818
  7 27.9476722864496
  8 27.6397164066067
  9 27.3290971727963
  10 27.0164877566437
  11 26.7027598955703
  12 26.388909307447
  13 26.0759678597544
  14 25.76491003394
  15 25.4565620547816
  16 25.1515221580468
(100)
```

1. **Geometry.DistanceTo:** Connect *Curve.PointAtParameter* output into the *input*. Connect *Point.ByCoordinates* into the *geometry input.
2. **Watch:** The resultant output shows a list of distances from each helical point to the reference point along the curve.

Our next step is to drive parameters with the list of distances from the helical points to the reference point. We use these distance values to define the radii of a series of spheres along the curve. In order to keep the spheres a suitable size, we need to *remap* the values for distance.

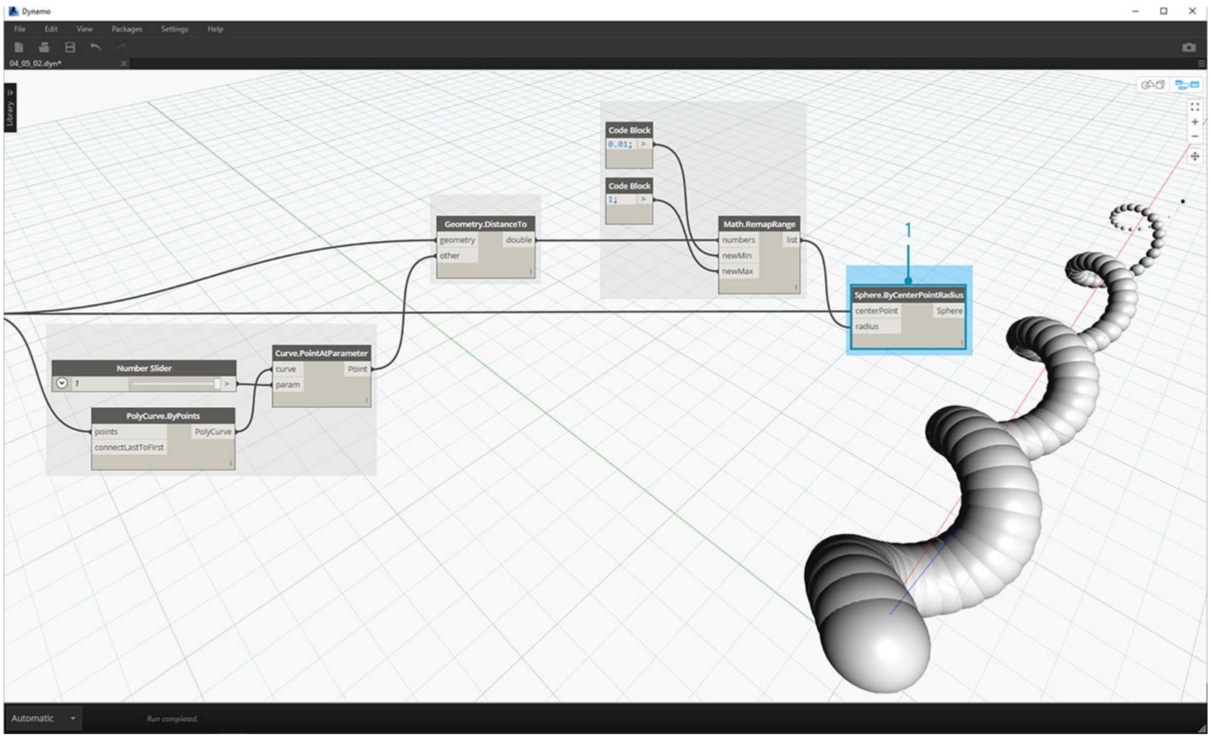


```
List
  0 1
  1 0.990353607692508
  2 0.980699115810038
  3 0.971006553419083
  4 0.961248862409859
  5 0.951404417420441
  6 0.941459098593536
  7 0.931407726075412
  8 0.921254718552306
  9 0.911013902824721
  10 0.900707472710088
  11 0.890364168548107
  12 0.880016818193982
  13 0.869699441320595
  14 0.859444165638815
  15 0.849278230980491
@L2 @L1                    (100)
```

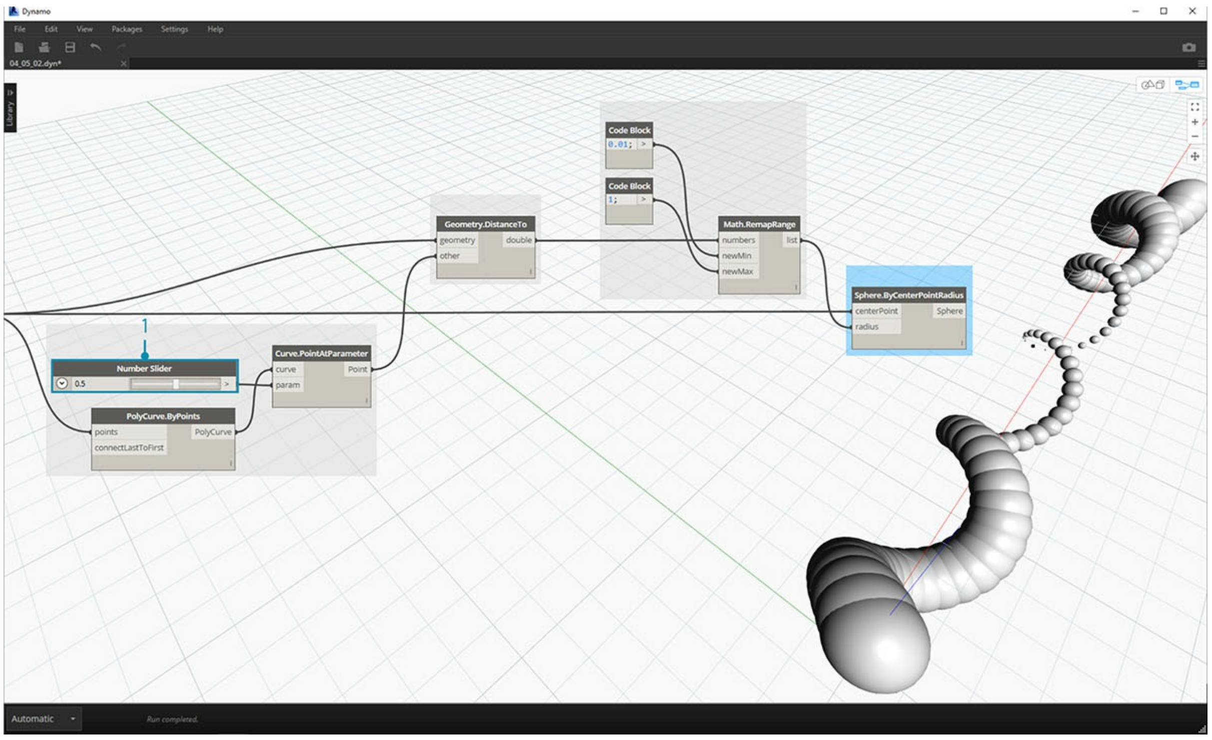
```
List
  0 30.0281783846477
  1 29.7355889008319
  2 29.4427537448117
  3 29.1487638534179
  4 28.8527985137742
  5 28.55420179636
  6 28.252545424818
  7 27.9476722864496
  8 27.6397164066067
  9 27.3290971727963
  10 27.0164877566437
  11 26.7027598955703
  12 26.388909307447
  13 26.0759678597544
  14 25.76491003394
  15 25.4565620547816
@L2 @L1                    (100)
```

1. **Math.RemapRange:** Connect *Geometry.DistanceTo* output into the numbers input.
2. **Code Block:** connect a code block with a value of *0.01* into the *newMin* input and a code block with a value of *1* into the *newMax* input.
3. **Watch:** connect the *Math.RemapRange* output into one Node and the *Geometry.DistanceTo* output into another. Compare the results.

This step has remapped the list of distance to be a smaller range. We can edit the *newMin* and *newMax* values however we see fit. The values will remap and will have the same *distribution ratio* across the domain.

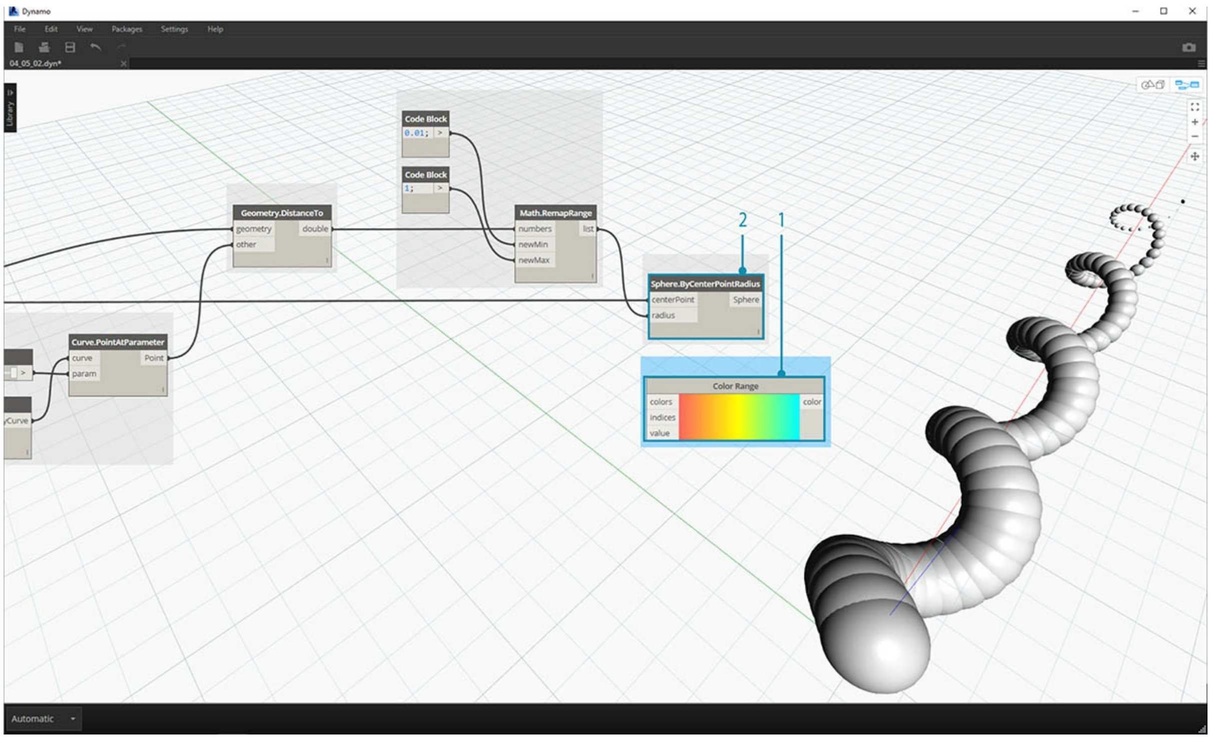


1. **Sphere.ByCenterPointRadius:** connect the *Math.RemapRange* output into the *radius* input and the original *Point.ByCoordinates* output into the *centerPoint* input.

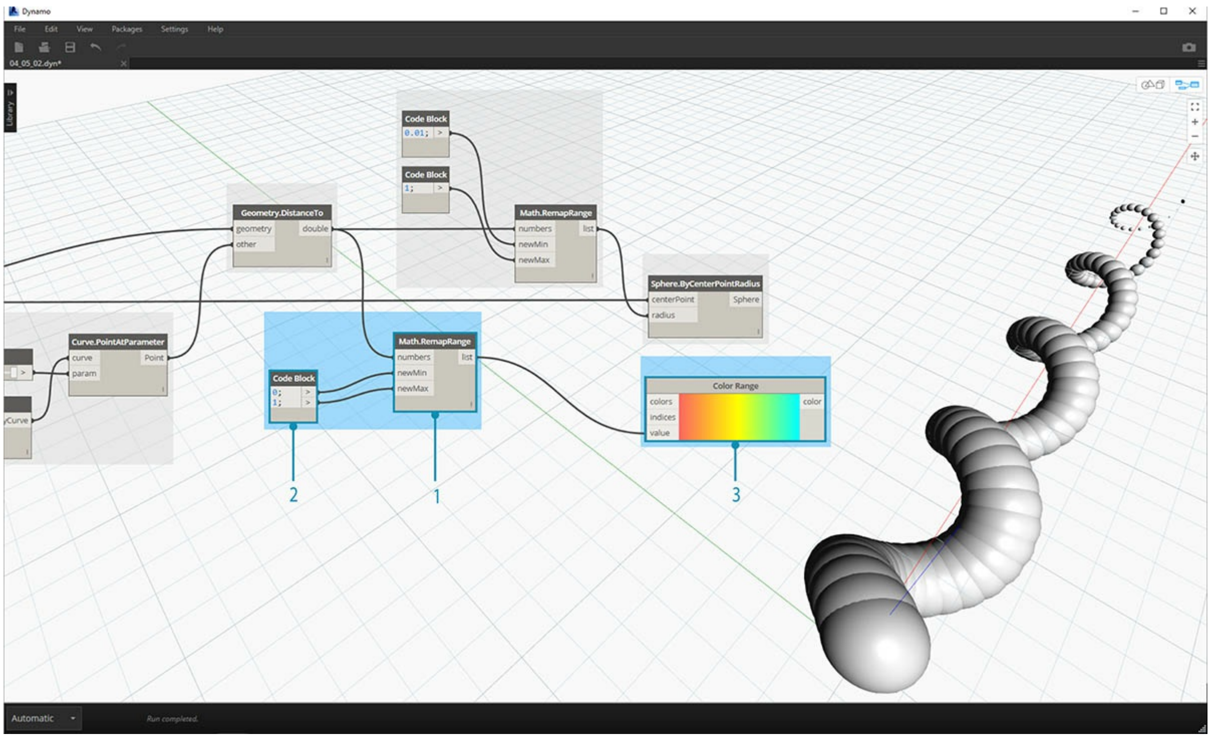


1. **Number Slider:** change the value of the number slider and watch the size of the spheres update. We now have a parametric jig.

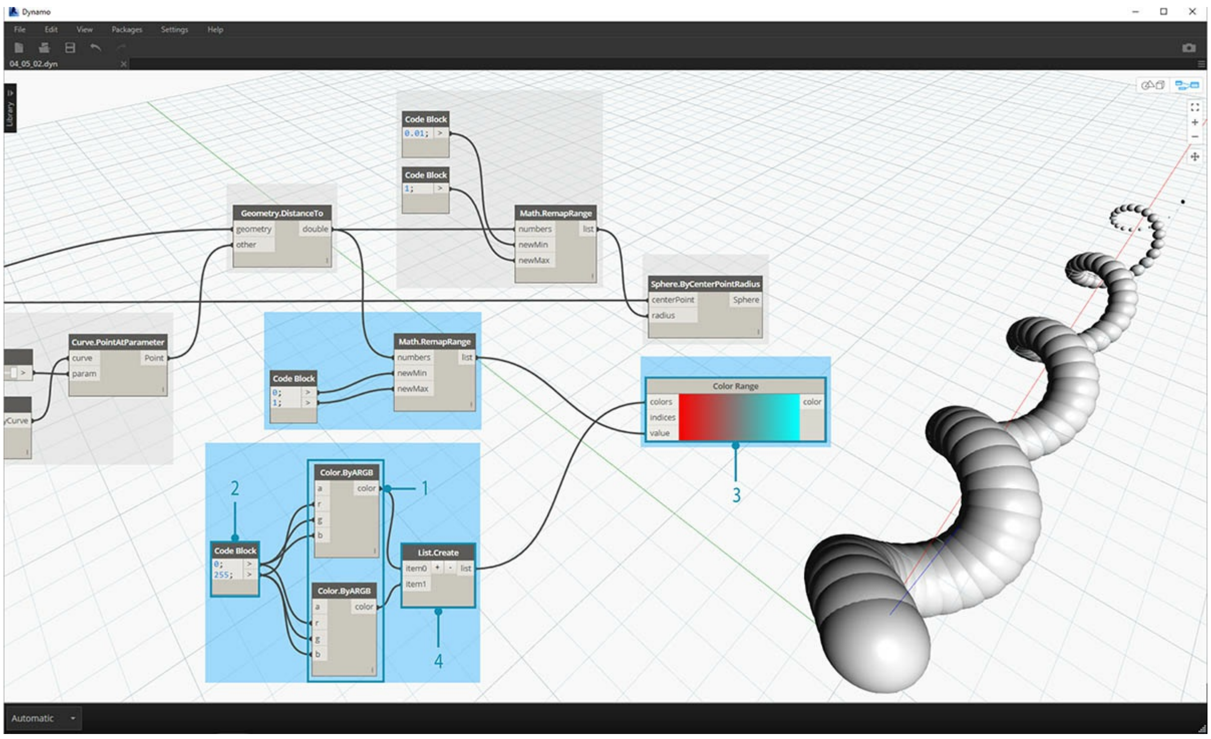
The size of the spheres demonstrates the parametric array defined by a reference point along the curve. Let's use the same concept for the sphere radius to drive their color.



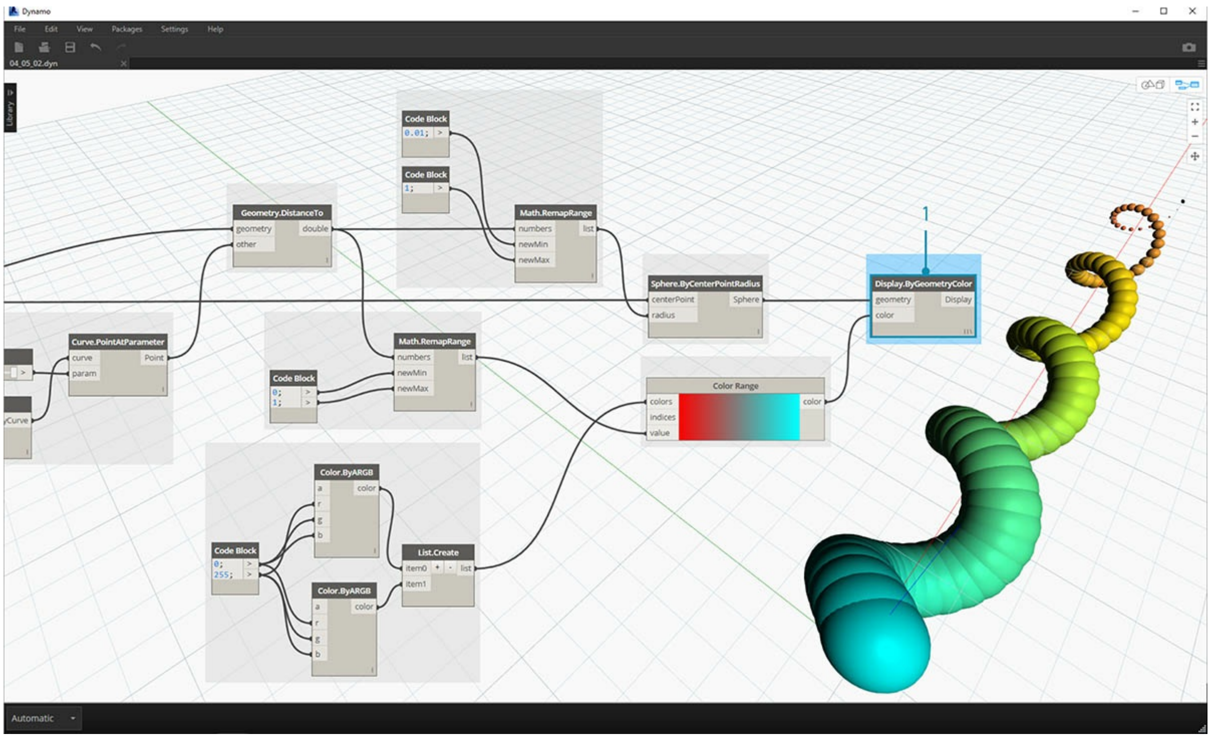
1. **Color Range:** Add top the canvas. When hovering over the *value* input, we notice that the numbers requested are between 0 and 1. We need to remap the numbers from the *Geometry.DistanceTo* output so that they are compatible with this domain.
2. **Sphere.ByCenterPointRadius:** For the time being, let's disable the preview on this Node (*Right Click > Preview*)



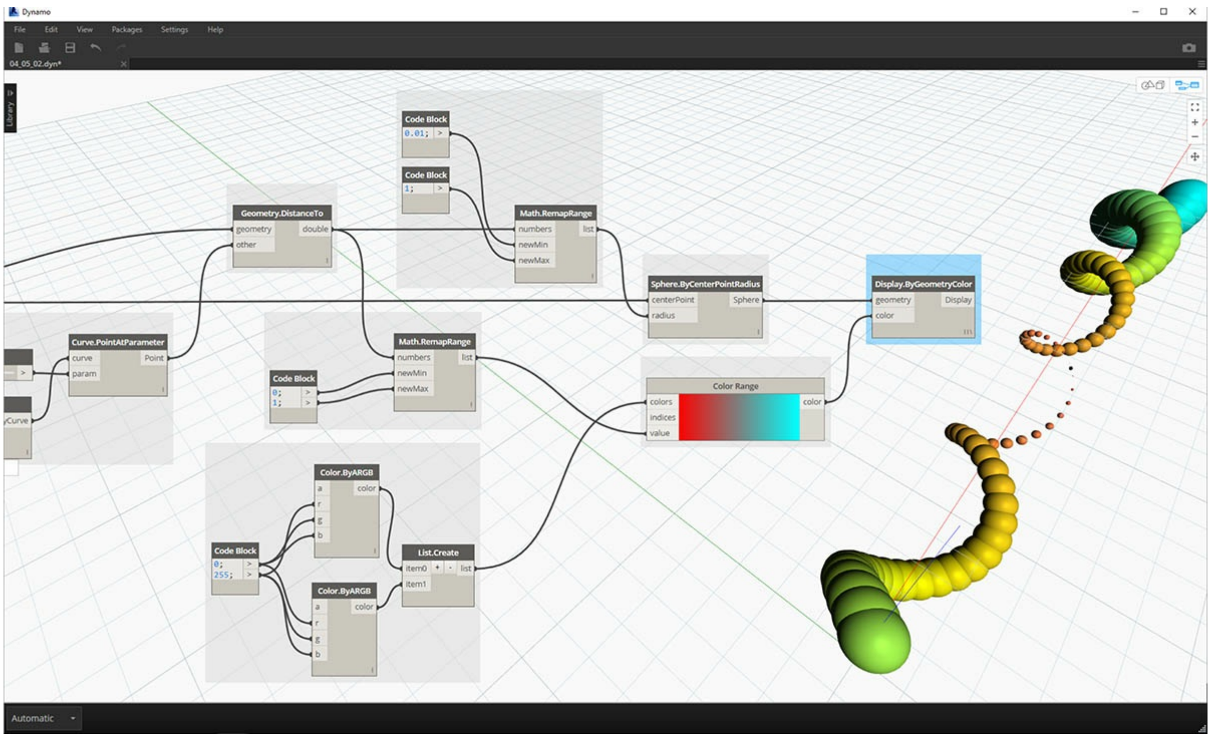
1. **Math.RemapRange:** This process should look familiar. Connect the *Geometry.DistanceTo* output into the numbers input.
2. **Code Block:** Similar to an earlier step, create a value of *0* for the *newMin* input and a value of *1* for the *newMax* input. Notice that we are able to define two outputs from one code block in this case.
3. **Color Range:** Connect the *Math.RemapRange* output into the *value* input.



1. **Color.ByARGB:** This is what we'll do to create two colors. While this process may look awkward, it's the same as RGB colors in another software, we're just using visual programming to do it.
2. **Code Block:** create two values of *0* and *255*. Plug the two outputs into the two *Color.ByARGB* inputs in agreement with the image above (or create your favorite two colors).
3. **Color Range:** The *colors* input requests a list of colors. We need to create this list from the two colors created in the previous step.
4. **List.Create:** merge the two colors into one list. Plug the output into the *colors* input for *Color Range*.



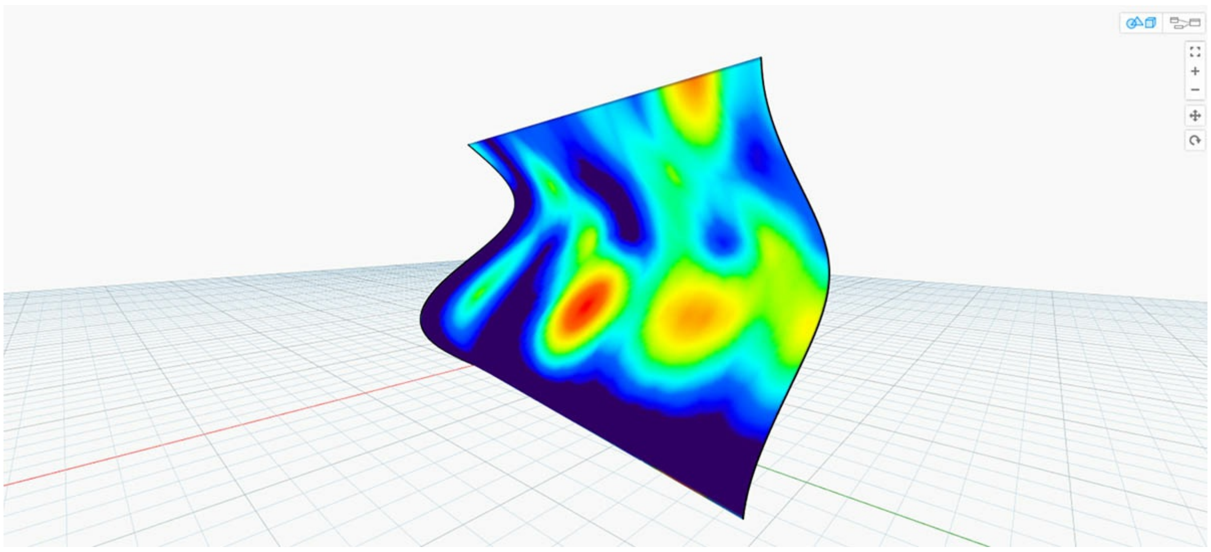
1. **Display.ByGeometryColor:** Connect *Sphere.ByCenterPointRadius* into the *geometry* input and the *Color Range* into the *color* input. We now have a smooth gradient across the domain of the curve.



If we change the value of the *number slider* from earlier in the definition, the colors and sizes update. Colors and radius size are directly related in this case: we now have a visual link between two parameters!

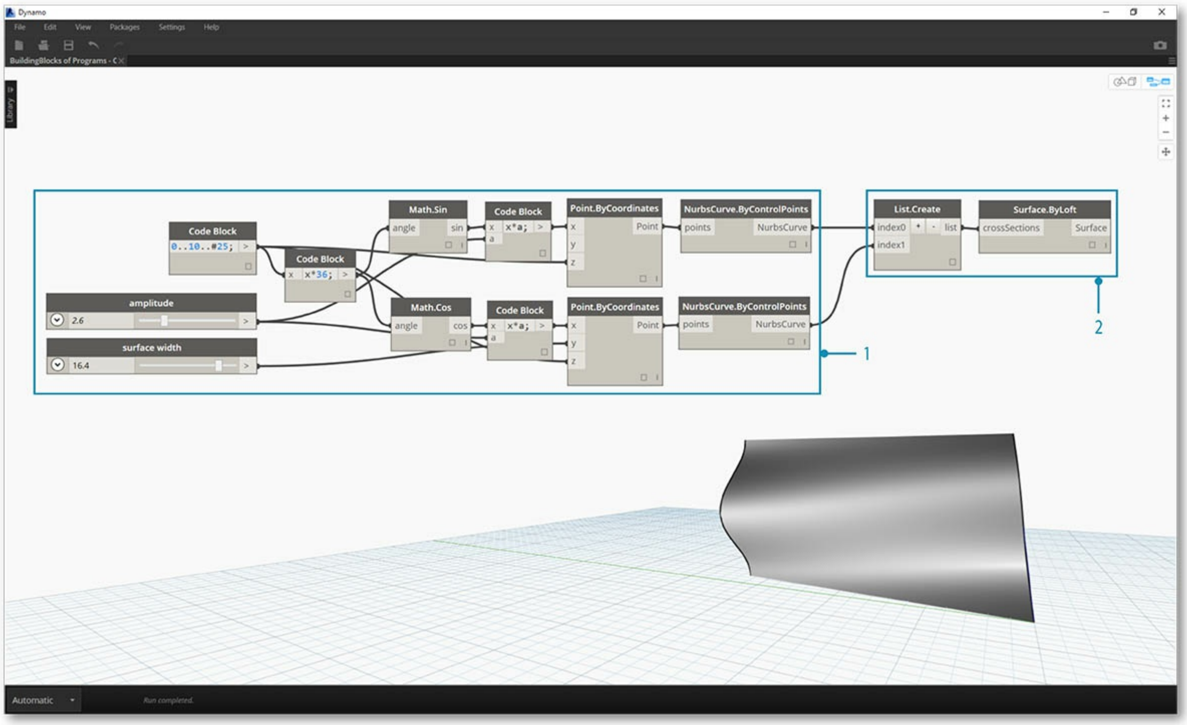
Color On Surfaces

The **Display.BySurfaceColors** node gives us the ability to map data across a surface using color! This functionality introduces some exciting possibilities for visualizing data obtained through discrete analysis like solar, energy, and proximity. Applying color to a surface in Dynamo is similar to applying a texture to a material in other CAD environments. Let's demonstrate how to use this tool in the brief exercise below.



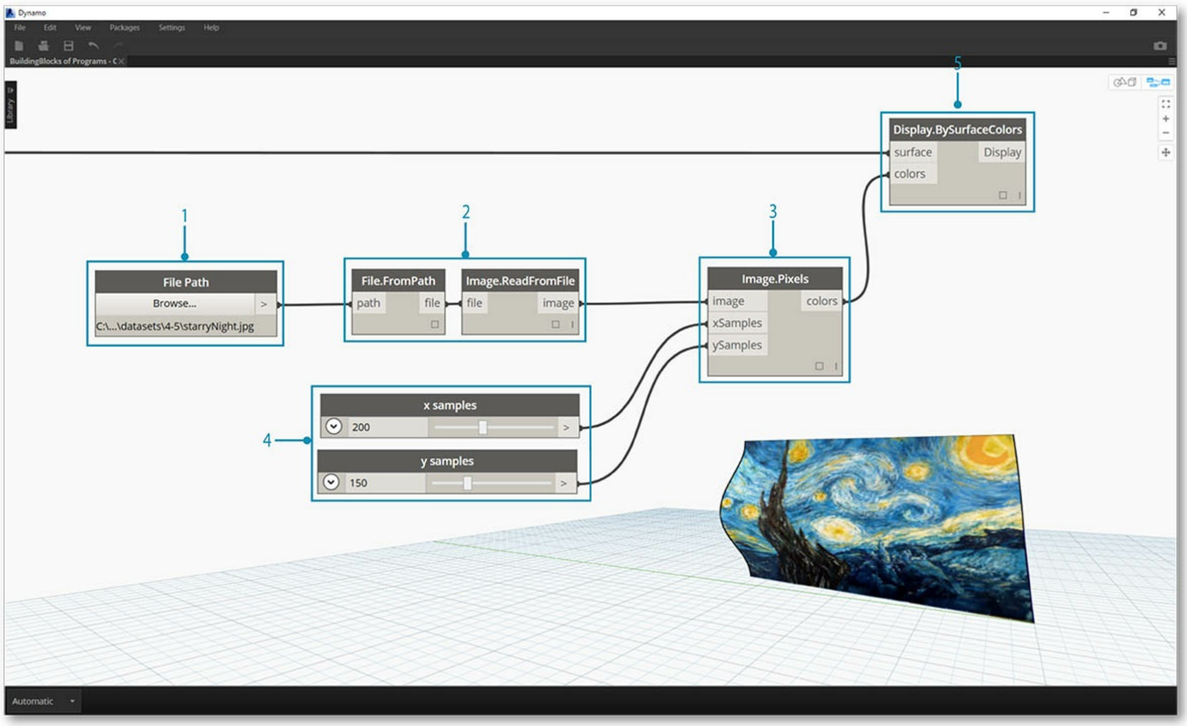
Color on Surfaces Exercise

Download the example file that accompanies this exercise (Right click and "Save Link As..."): Building Blocks of Programs - ColorOnSurface.zip. A full list of example files can be found in the Appendix.

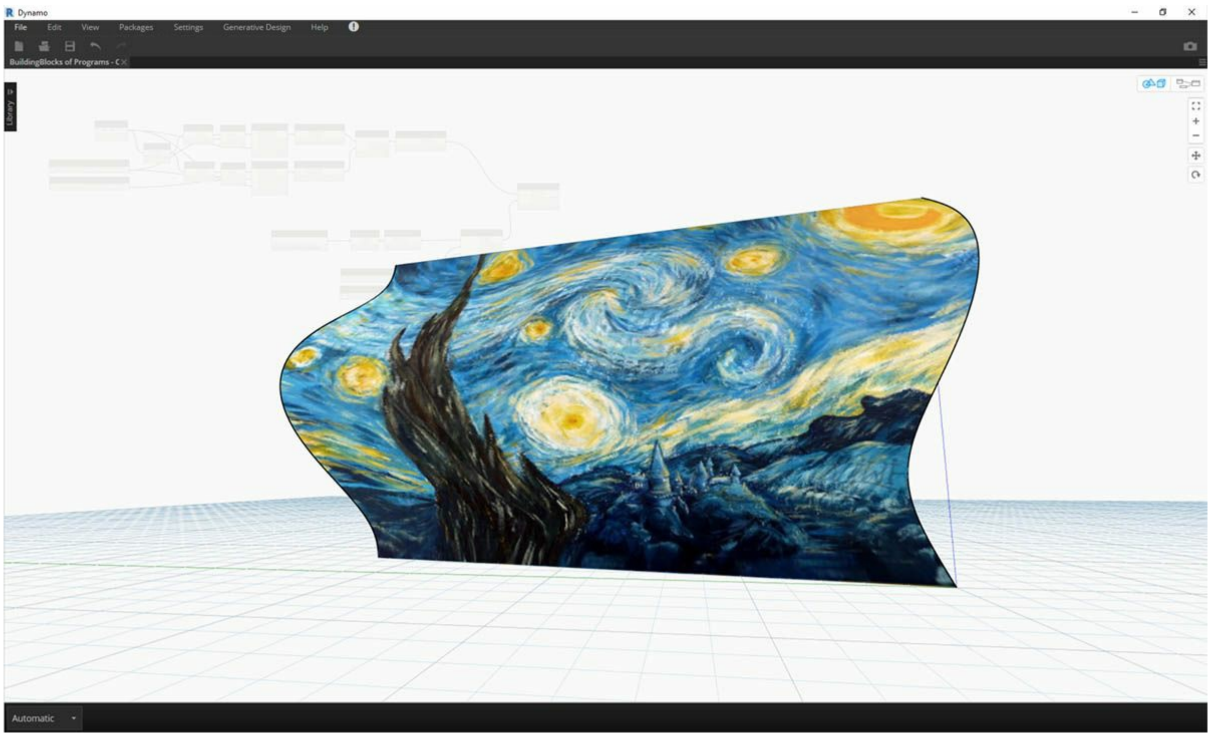


First, we need to create (or reference) a surface to use as an input for the **Display.BySurfaceColors** node. For this example we are lofting between a sine and cosine curve.

1. This **Group** of nodes is creating points along the Z-axis then displacing them based on sine and cosine functions. The two point lists are then used to generate NURBS curves.
2. **Surface.ByLoft**: generate an interpolated surface between the list of NURBS curves.



1. **File Path**: select an image file to sample for pixel data downstream
2. use **File.FromPath** to convert the file path to a file then pass into **Image.ReadFromFile** to output an image for sampling
3. **Image.Pixels**: input an image and provide a sample value to use along the x and y dimensions of the image.
4. **Slider**: provide sample values for **Image.Pixels**
5. **Display.BySurfaceColors**: map array of color values across surface along X and Y respectively

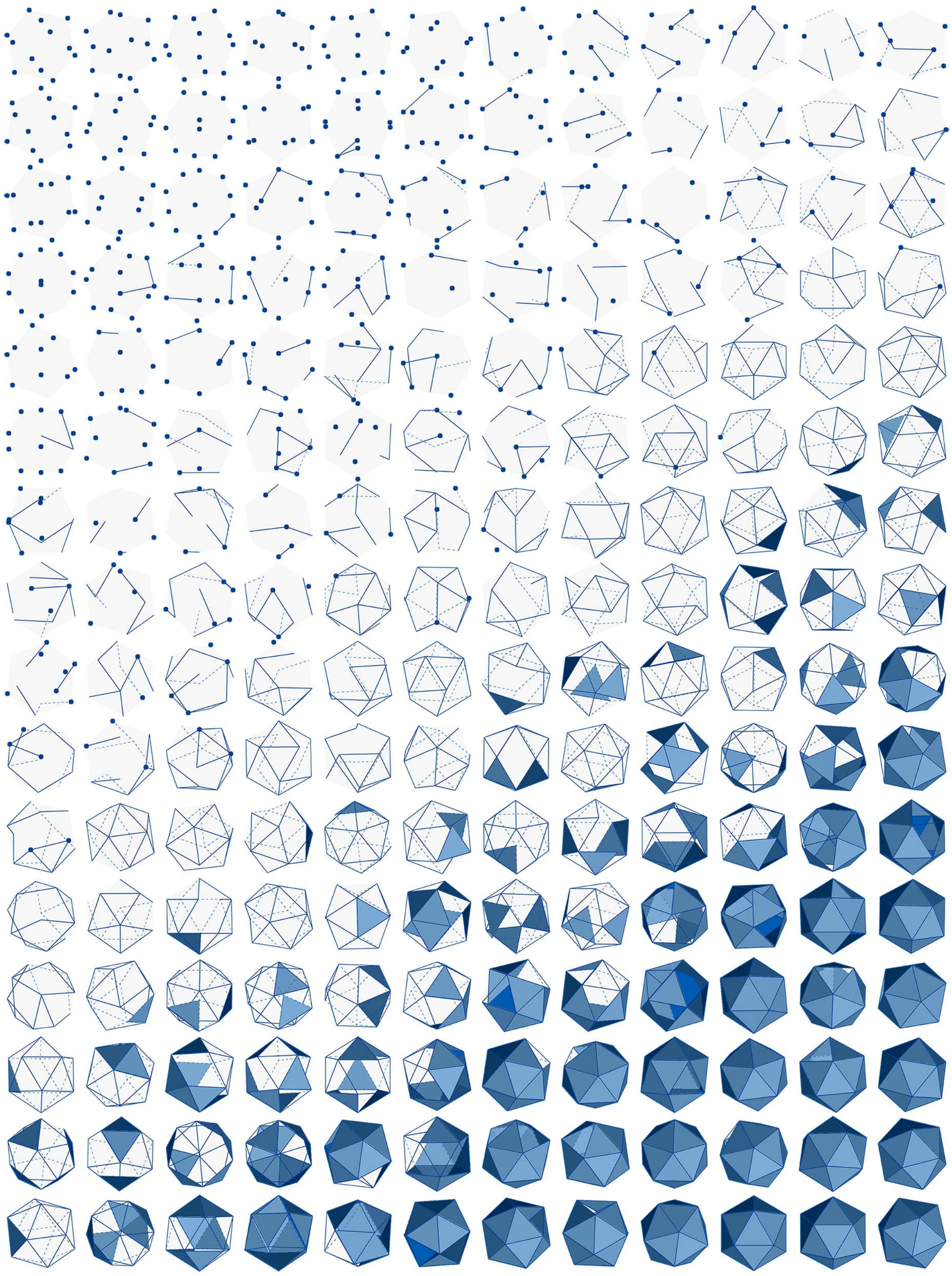


Close-up preview of the output surface with resolution of 400x300 samples

Geometry for Computational Design

GEOMETRY FOR COMPUTATIONAL DESIGN

As a Visual Programming environment, Dynamo enables you to craft the way that data is processed. Data is numbers or text, but so is Geometry. As understood by the Computer, Geometry - or sometimes called Computational Geometry - is the data we can use to create beautiful, intricate, or performance-driven models. To do so, we need to understand the ins and outs of the various types of Geometry we can use.



Geometry Overview

Geometry Overview

Geometry is the language for design. When a programming language or environment has a geometry kernel at its core, we can unlock the possibilities for designing precise and robust models, automating design routines, and generating design iterations with algorithms.

The Basics

Geometry, traditionally defined, is the study of shape, size, relative position of figures, and the properties of space. This field has a rich history going back thousands of years. With the advent and popularization of the computer, we gained a powerful tool in defining, exploring, and generating geometry. It is now so easy to calculate the result of complex geometric interactions, the fact that we are doing so is almost transparent.

If you're curious to see how diverse and complex geometry can get using the power of your computer, do a quick web search for the Stanford Bunny - a canonical model used to test algorithms.

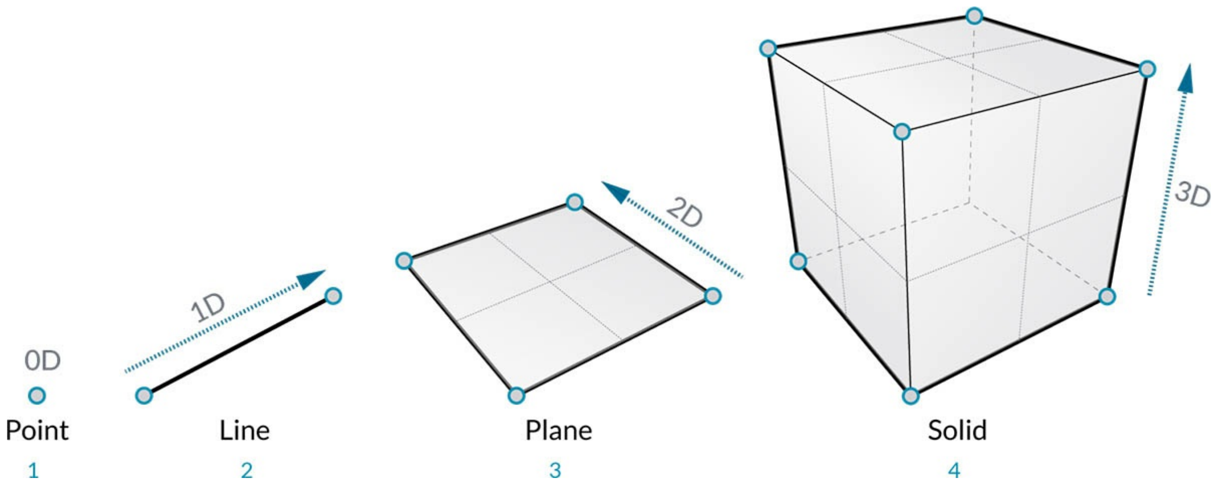
Understanding geometry in the context of algorithms, computing, and complexity, may sound daunting; however, there are a few key, and relatively simple, principles that we can establish as fundamentals to start building towards more advanced applications:

1. Geometry is **Data** - to the computer and Dynamo, a Bunny not all that different from a number.
2. Geometry relies on **Abstraction** - fundamentally, geometric elements are described by numbers, relationships, and formulas within a given spatial coordinate system
3. Geometry has a **Hierarchy** - points come together to make lines, lines come together to make surfaces, and so on
4. Geometry simultaneously describes both **the Part and the Whole** - when we have a curve, it is both the shape as well as all the possible points along it

In practice, these principles mean that we need to be aware of what we are working with (what type of geometry, how was it created, etc.) so that we can fluidly compose, decompose, and recompose different geometries as we develop more complex models.

Stepping through the Hierarchy

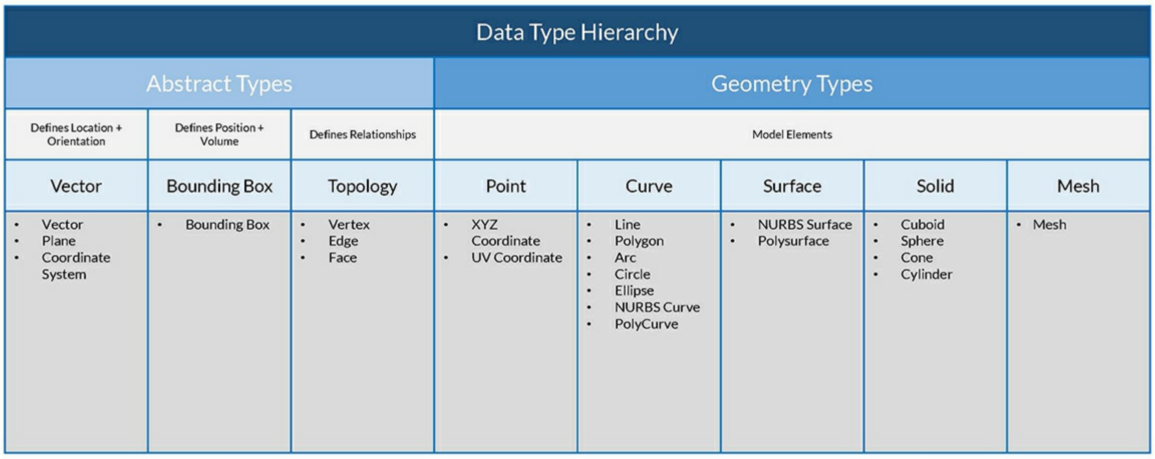
Let's take a moment to look at the relationship between the Abstract and Hierarchical descriptions of Geometry. Because these two concepts are related, but not always obvious at first, we can quickly arrive at a conceptual roadblock once we start developing deeper workflows or models. For starters, let's use dimensionality as an easy descriptor of the "stuff" we model. The number of dimensions required to describe a shape gives us a window into how Geometry is organized hierarchically.



1. A **Point** (defined by coordinates) doesn't have any dimensions to it - it's just numbers describing each coordinate
2. A **Line** (defined by two points) now has *one* dimension - we can "walk" the line either forward (positive direction) or backward (negative direction)
3. A **Plane** (defined by two lines) has *two* dimensions - walking more left or more right is now possible

4. A **Box** (defined by two planes) has *three* dimensions - we can define a position relative to up or down

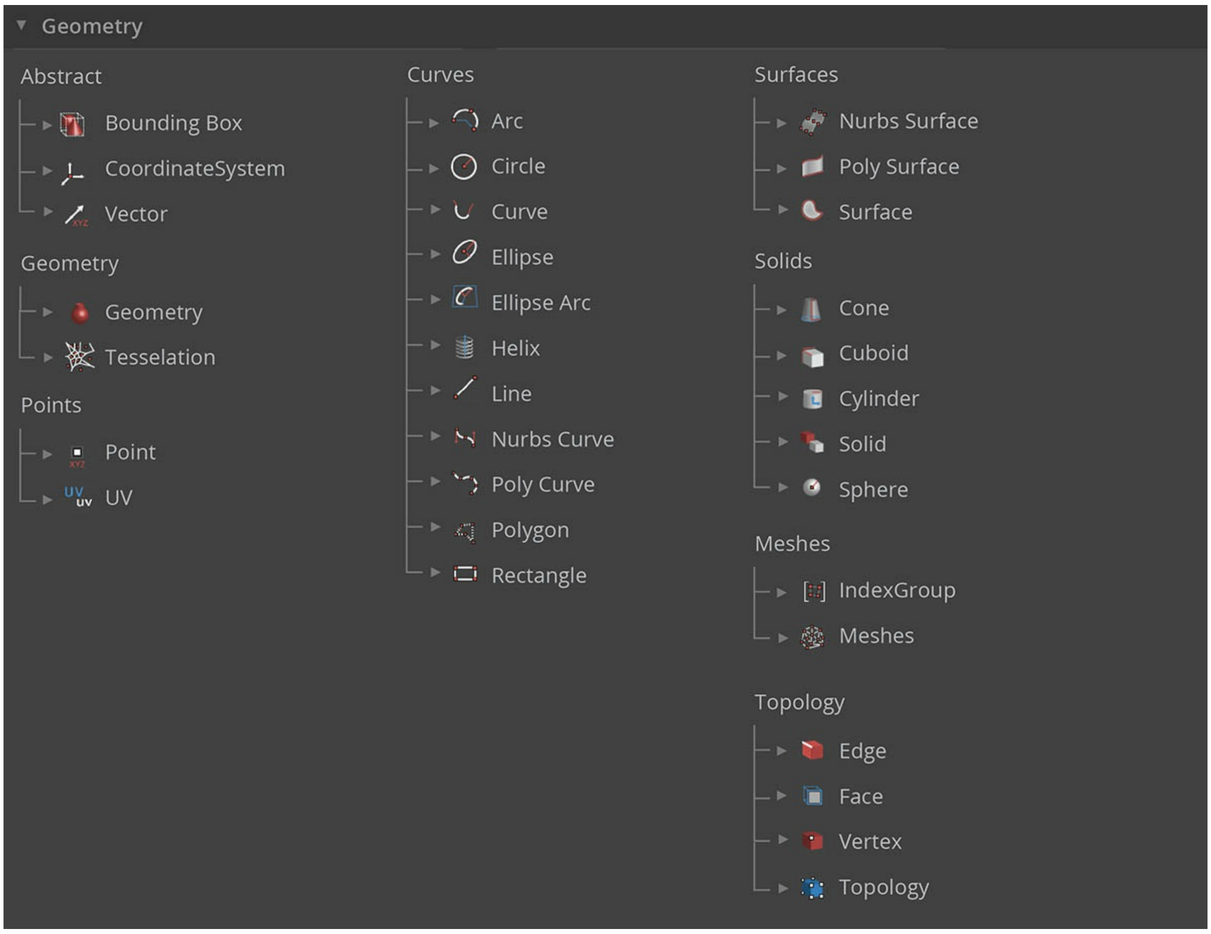
Dimensionality is a convenient way to start categorizing Geometry but it's not necessarily the best. After all, we don't model with only Points, Lines, Planes, and Boxes - what if I want something curvy? Furthermore, there is a whole other category of Geometric types that are completely abstract ie. they define properties like orientation, volume, or relationships between parts. We can't really grab a hold of a Vector so how do we define it relative to what we see in space? A more detailed categorization of the geometric hierarchy should accommodate the difference between Abstract Types or "Helpers," each of which we can group by what they help do and types that help describe the shape of model elements.



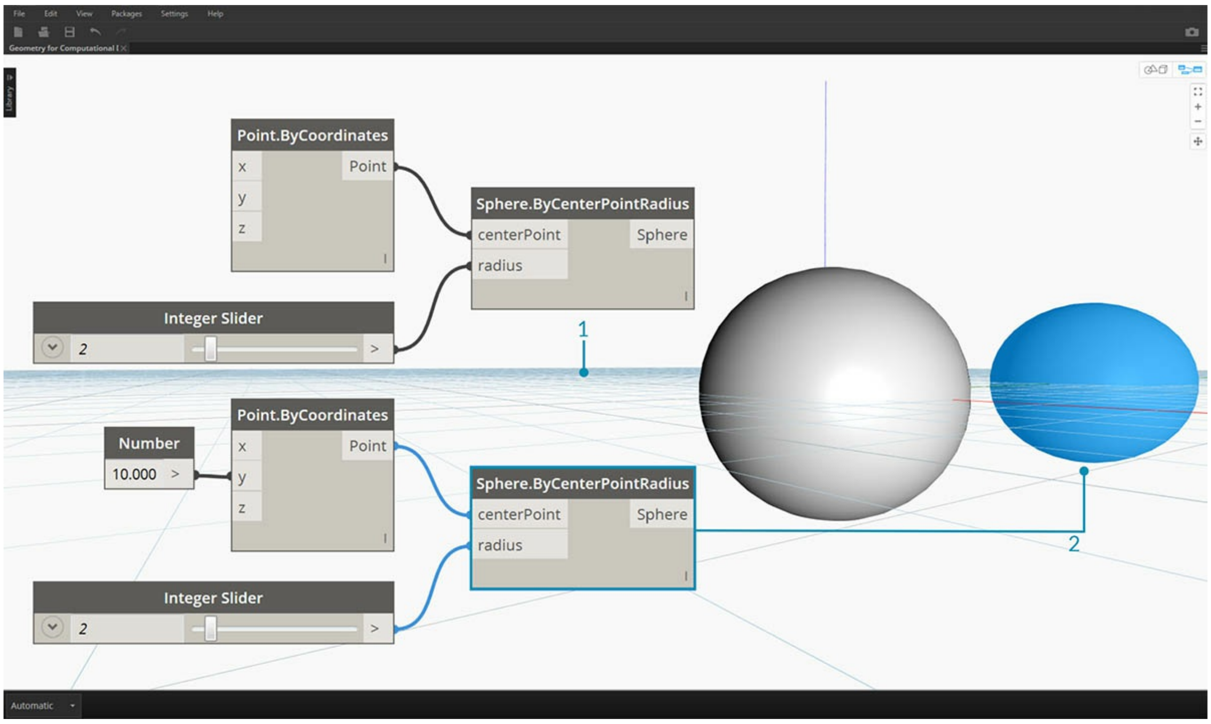
Data Type Hierarchy							
Abstract Types			Geometry Types				
Defines Location + Orientation	Defines Position + Volume	Defines Relationships	Model Elements				
Vector	Bounding Box	Topology	Point	Curve	Surface	Solid	Mesh
• Vector • Plane • Coordinate System	• Bounding Box	• Vertex • Edge • Face	• XYZ Coordinate • UV Coordinate	• Line • Polygon • Arc • Circle • Ellipse • NURBS Curve • PolyCurve	• NURBS Surface • Polysurface	• Cuboid • Sphere • Cone • Cylinder	• Mesh

Geometry in Dynamo Sandbox

So what does this mean for using Dynamo? Understanding the Geometry types and how they are related will allow us to navigate the collection of **Geometry Nodes** available to us in the Library. The Geometry Nodes are organized alphabetically as opposed to hierarchically - here they are displayed similar to their layout in the Dynamo interface.



Additionally, making models in Dynamo and connecting the preview of what we see in the Background Preview to the flow of data in our graph should become more intuitive over time.



1. Note the assumed coordinate system rendered by the grid and colored axes
2. Selected Nodes will render the corresponding geometry (if the Node creates geometry) in the background the highlight color

Download the example file that accompanies this image (Right click and "Save Link As..."): Geometry for Computational Design - Geometry Overview.dyn. A full list of example files can be found in the Appendix.

Going Further with Geometry

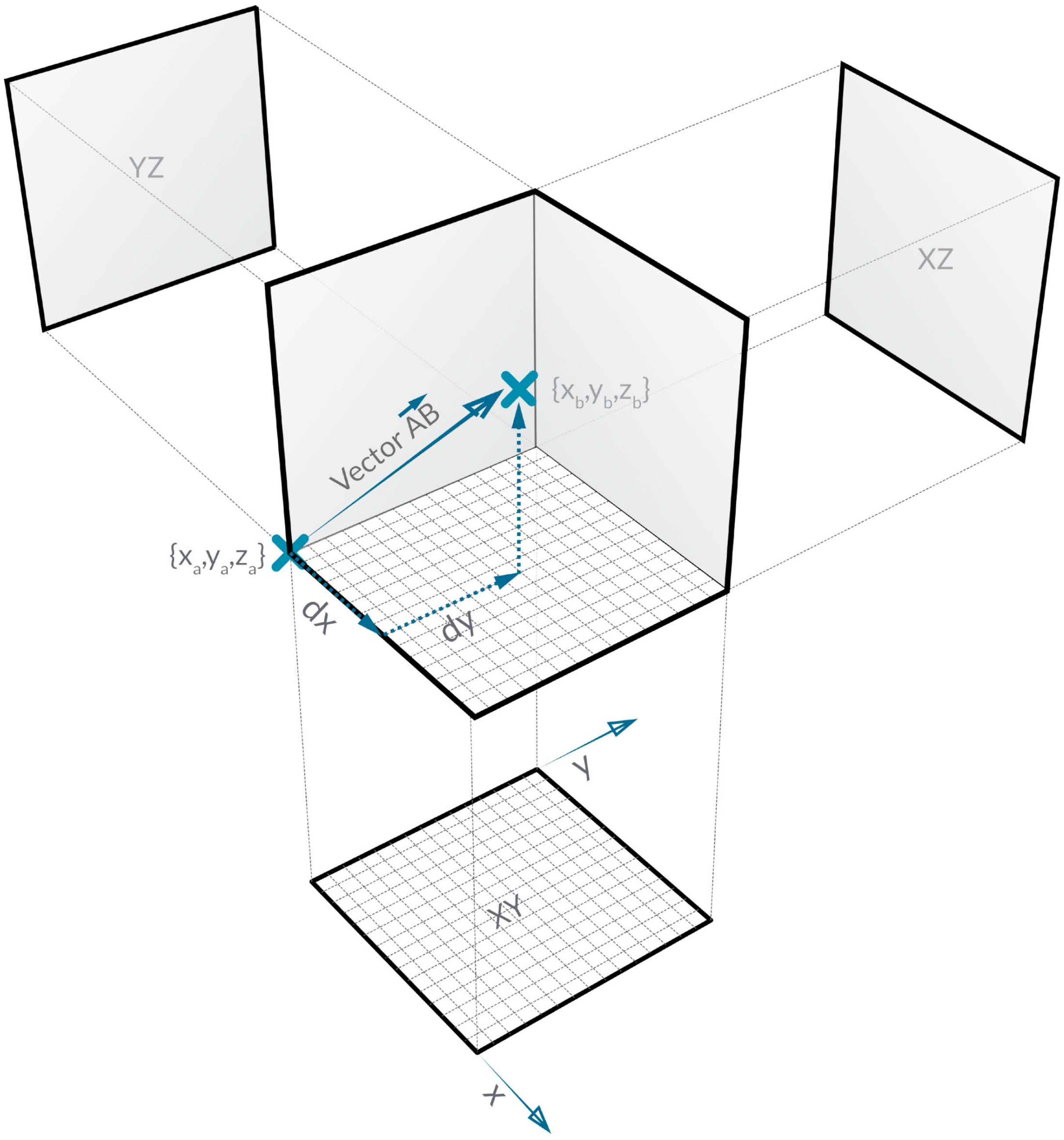
Creating models in Dynamo is not limited to what we can generate with Nodes. Here are some key ways to take your process to the next level with Geometry:

1. Dynamo allows you to import files - try using a CSV for point clouds or SAT for bringing in surfaces
2. When working with Revit, we can reference Revit elements to use in Dynamo
3. The Dynamo Package Manager offers additional functionality for extended geometry types and operations - check out the Mesh Toolkit package

Vectors

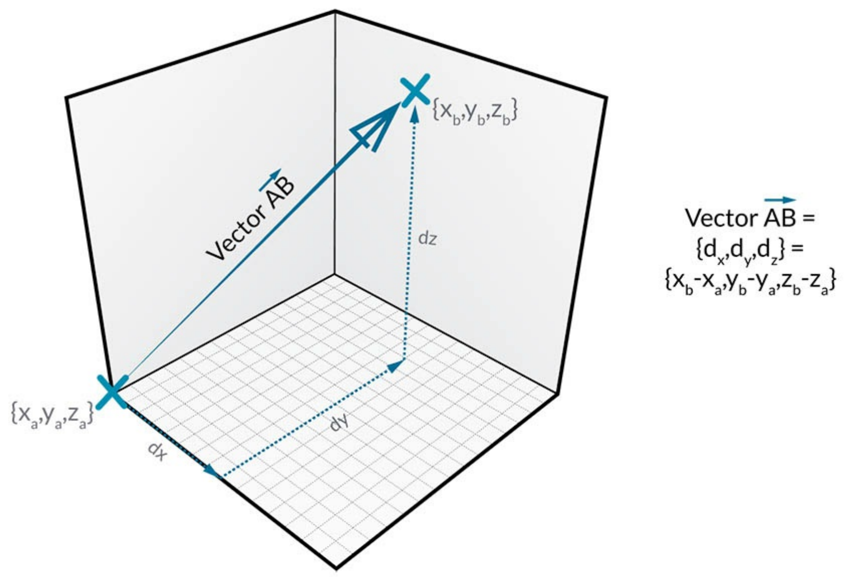
Vectors, Planes, and Coordinate Systems

Vectors, Planes, and Coordinate Systems make up the primary group of Abstract Geometry Types. They help us define location, orientation, and the spatial context for other geometry that describe shapes. If I say that I'm in New York City at 42nd Street and Broadway (Coordinate System), standing on the street level (Plane), looking North (Vector), I've just used these "Helpers" to define where I am. The same goes for a phone case product or a skyscraper - we need this context to develop our model.



What's a Vector?

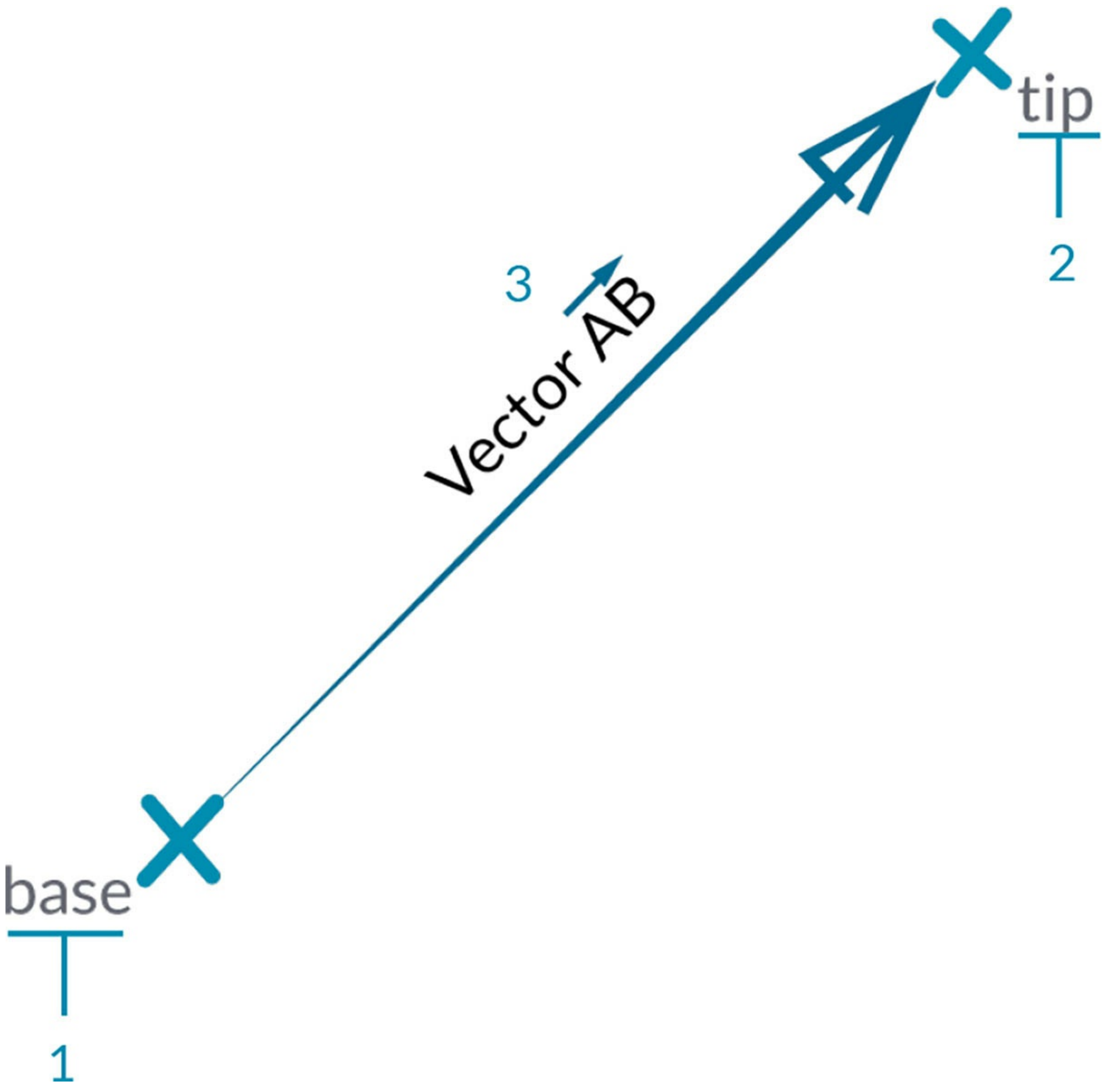
A vector is a geometric quantity describing Direction and Magnitude. Vectors are abstract; ie. they represent a quantity, not a geometrical element. Vectors can be easily confused with Points because they both are composed of a list of values. There is a key difference though: Points describe a position in a given coordinate system while Vectors describe a relative difference in position which is the same as saying "direction."



Vector $\overrightarrow{AB}$
$\{x_b,y_b,z_b\}$
dz
dy
dx
$\{x_a,y_a,z_a\}$
Vector $\overrightarrow{AB}$ =
$\{d_x,d_y,d_z\}$ =
$\{x_b-x_a,y_b-y_a,z_b-z_a\}$

If the idea of relative difference is confusing, think of the Vector AB as "I'm standing at Point A, looking toward Point B." The direction, from here (A) to there (B), is our Vector.

Breaking down Vectors further into their parts using the same AB notation:

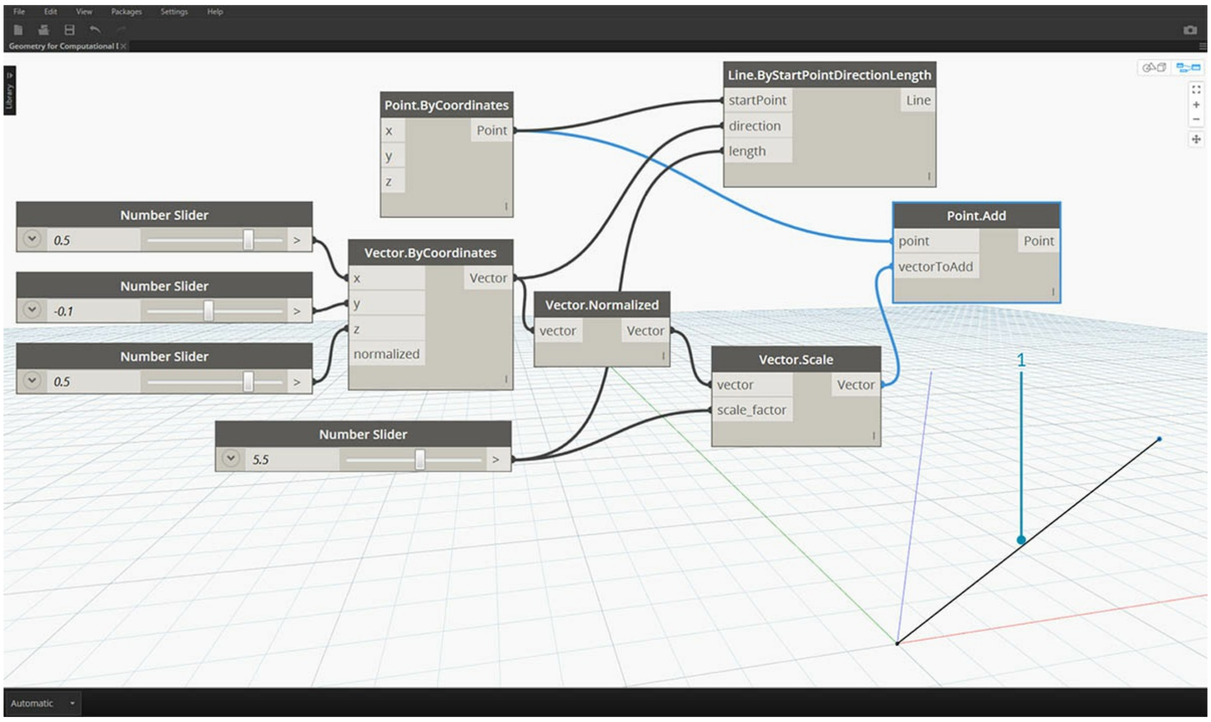


1. The **Start Point** of the Vector is called the **Base**.
2. The **End Point** of the Vector is called the **Tip** or the **Sense**.
3. Vector AB is not the same as Vector BA - that would point in the opposite direction.

If you're ever in need of comic relief regarding Vectors (and their abstract definition), watch the classic comedy Airplane and listen for the oft-quoted tongue-in cheek line:

> *Roger, Roger. What's our vector, Victor?*

Vectors are a key component to our models in Dynamo. Note that, because they are in the Abstract category of "Helpers," when we create a Vector, we won't see anything in the Background Preview.

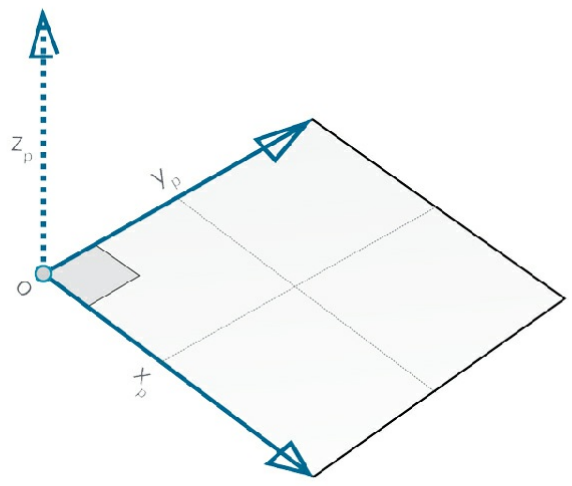


1. We can use a line as a stand in for a Vector preview.

Download the example file that accompanies this image (Right click and "Save Link As..."): Geometry for Computational Design - Vectors.dyn. A full list of example files can be found in the Appendix.

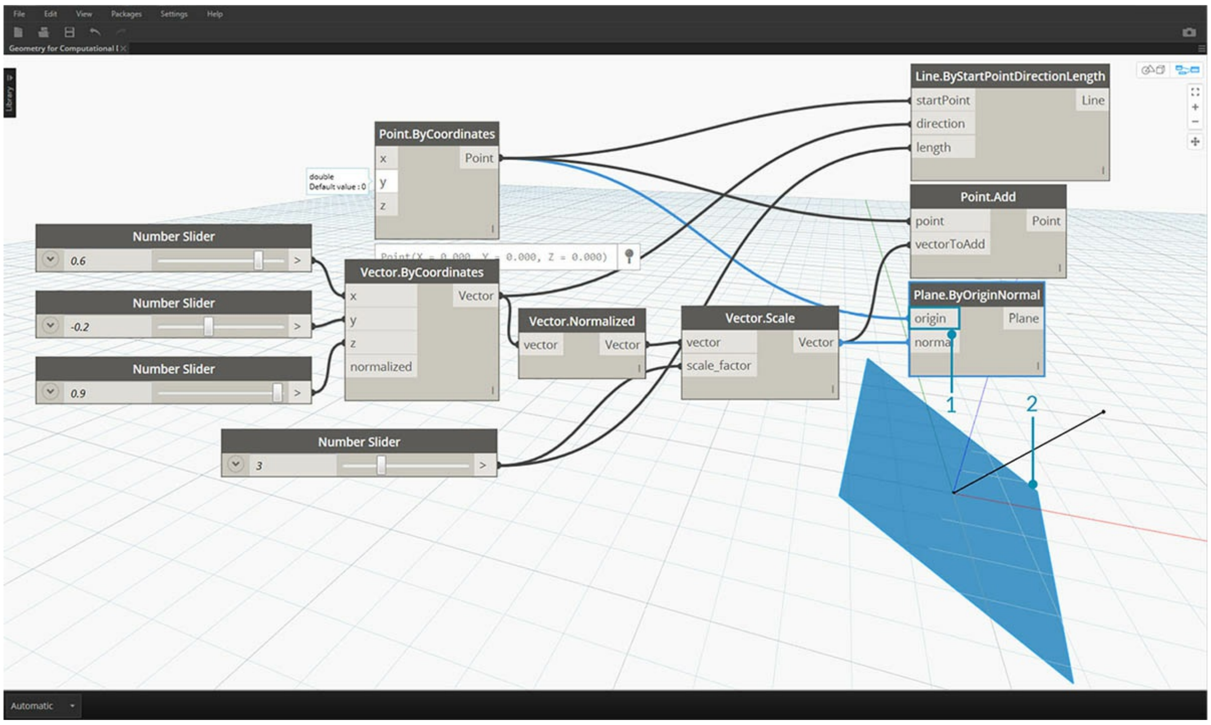
What's a Plane?

Planes are two-dimensional abstract "Helpers." More specifically, Planes are conceptually “flat,” extending infinitely in two directions. Usually they are rendered as a smaller rectangle near their origin.



You might be thinking, "Wait! Origin? That sounds like a Coordinate System... like the one I use to model in my CAD software!"

And you're correct! Most modeling software take advantage of construction planes or "levels" to define a local two-dimentional context to draft in. XY, XZ, YZ -or- North, Southeast, Plan might sound more familiar. These are all Planes, defining an infinite "flat" context. Planes don't have depth, but they do help us describe direction as well - each Plane has an Origin, X Direction, Y Direction, and a Z (Up) Direction.



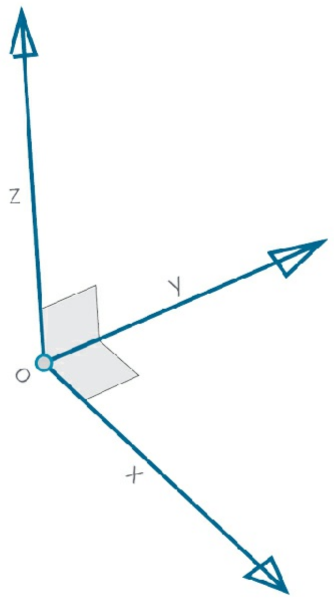
1. Although they are abstract, Planes do have an origin position so we can locate them in space.
2. In Dynamo, Planes are rendered in the Background Preview.

Download the example file that accompanies this image (Right click and "Save Link As..."): Geometry for Computational Design - Planes.dyn. A full list of example files can be found in the Appendix.

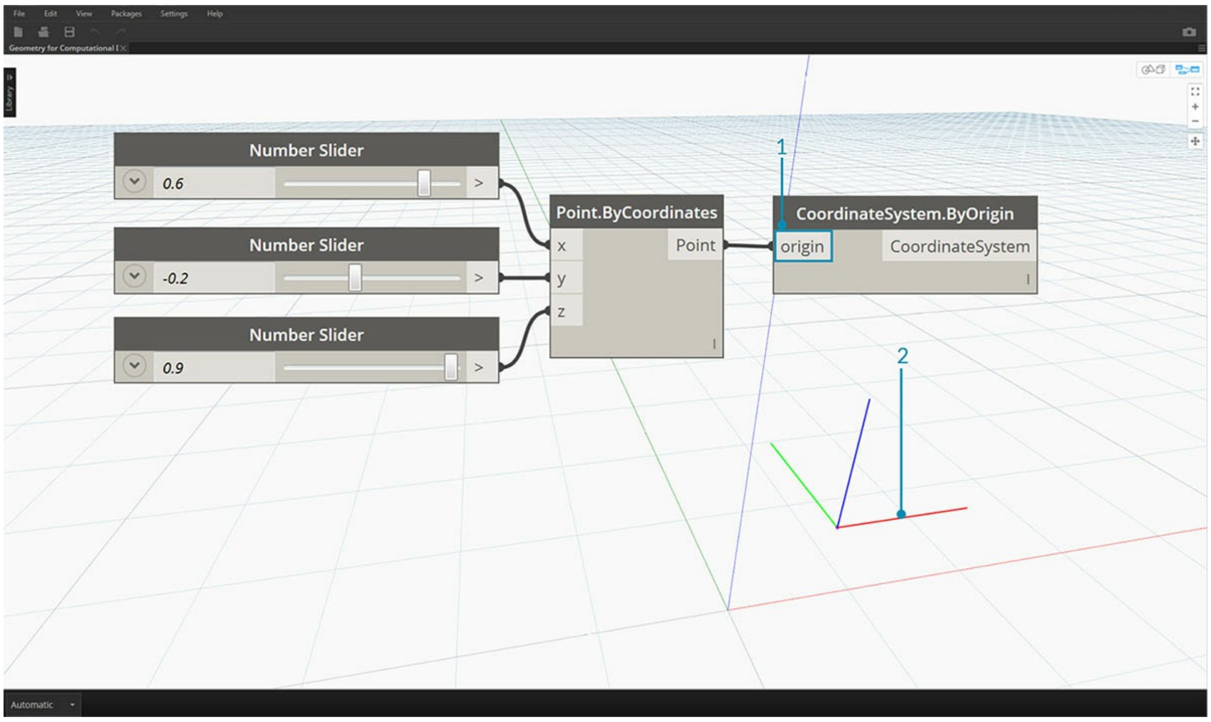
What's a Coordinate System?

If we are comfortable with Planes, we are a small step away from understanding Coordinate Systems. A Plane has all the same parts as a Coordinate System, provided it is a standard "Euclidean" or "XYZ" Coordinate System.

There are other, however, alternative Coordinate Systems such as Cylindrical or Spherical. As we will see in later sections, Coordinate Systems can also be applied to other Geometry types to define a position on that geometry.



Add alternative coordinate systems - cylindrical, spherical



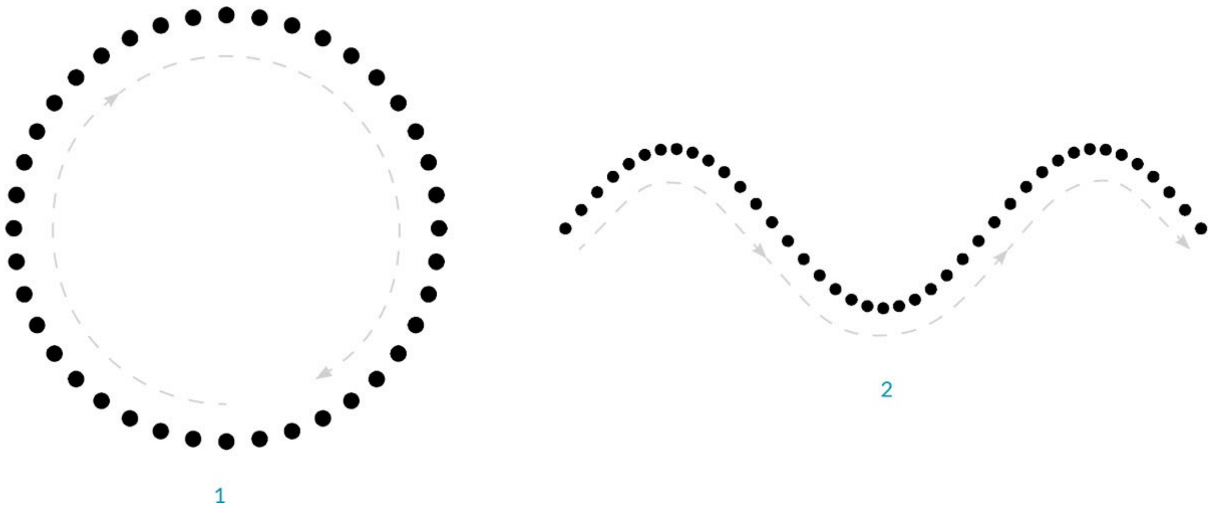
1. Although they are abstract, Coordinate Systems also have an origin position so we can locate them in space.
2. In Dynamo, Coordinate Systems are rendered in the Background Preview as a point (origin) and lines defining the axes (X is red, Y is green, and Z is blue following convention).

Download the example file that accompanies this image (Right click and "Save Link As..."): Geometry for Computational Design - Coordinate System.dyn. A full list of example files can be found in the Appendix.

Points

Points

If Geometry is the language of a model, then Points are the alphabet. Points are the foundation upon which all other geometry is created - we need at least two Points to create a Curve, we need at least three Points to make a Polygon or a Mesh Face, and so on. Defining the position, order, and relationship among Points (try a Sine Function) allows us to define higher order geometry like things we recognize as Circles or Curves.



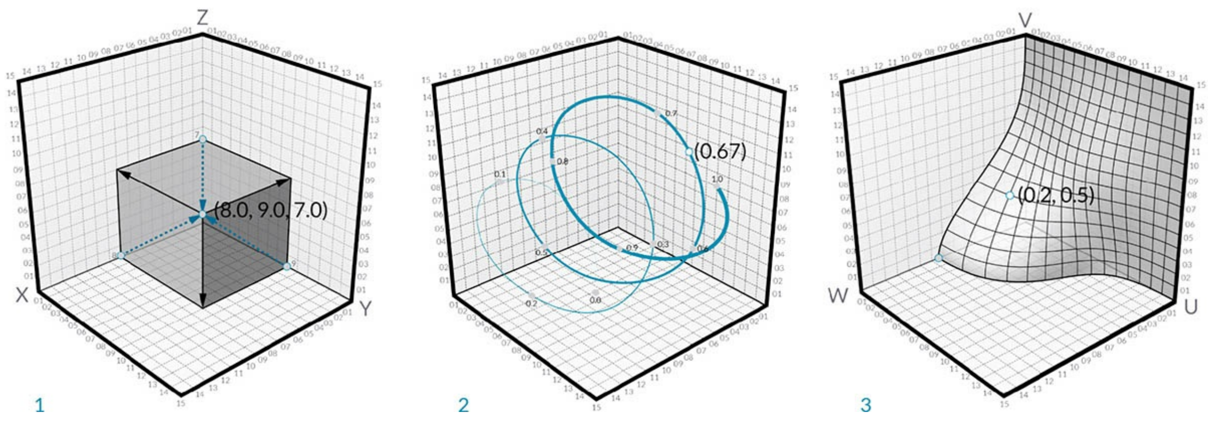
1. A Circle using the functions `x=r*cos(t)` and `y=r*sin(t)`
2. A Sine Curve using the functions `x=(t)` and `y=r*sin(t)`

What's a Point?

A Point is defined by nothing more than one or more values called coordinates. How many coordinate values we need to define the Point depends upon the Coordinate System or context in which it resides. The most common kind of Point in Dynamo exists in our three-dimensional World Coordinate System and has three coordinates [X,Y,Z].

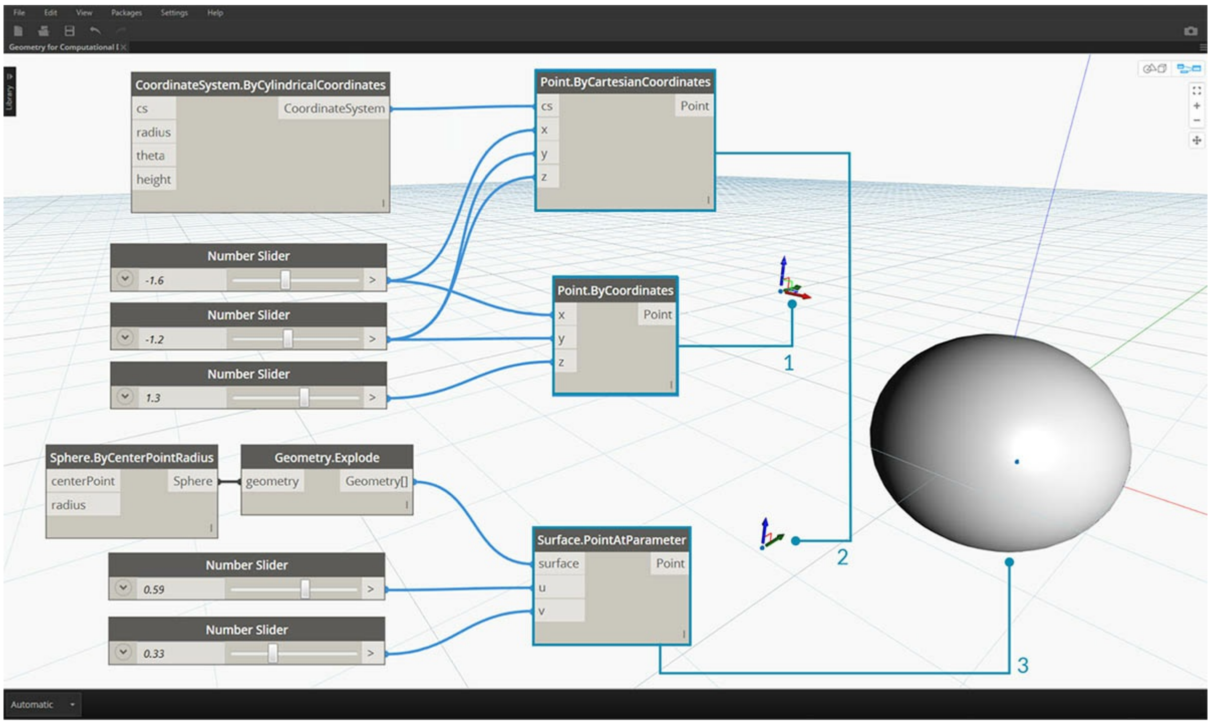
Point as Coordinates

Points can exist in a two-dimensional Coordinate System as well. Convention has different letter notation depending upon what kind of space we are working with - we might be using [X,Y] on a Plane or [U,V] if we are on a surface.



1. A Point in Euclidean Coordinate System: [X,Y,Z]
2. A Point in a Curve Parameter Coordinate System: [t]
3. A Point in a Surface Parameter Coordinate System: [U,V]

Although it might seem counter intuitive, Parameters for both Curves and Surfaces are continuous and extend beyond the edge of the given geometry. Since the shapes that define the Parameter Space reside in a three-dimensional World Coordinate System, we can always translate a Parametric Coordinate into a "World" Coordinate. The point [0.2, 0.5] on the surface for example is the same as point [1.8, 2.0, 4.1] in world coordinates.



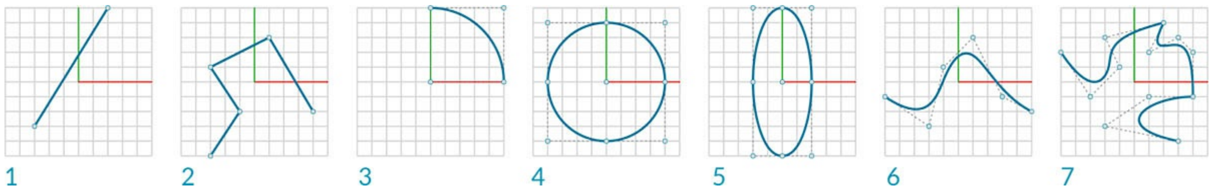
1. Point in assumed World XYZ Coordinates
2. Point relative to a given Coordinate System (Cylindrical)
3. Point as UV Coordinate on a Surface

Download the example file that accompanies this image (Right click and "Save Link As..."): Geometry for Computational Design - Points.dyn. A full list of example files can be found in the Appendix.

Curves

Curves

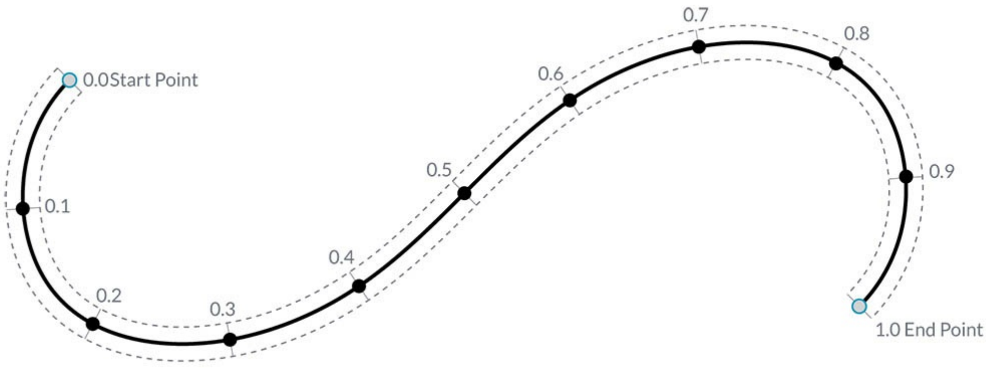
Curves are the first Geometric Data Type we've covered that have a more familiar set of shape descriptive properties - How curvey or straight? How long or short? And remember that Points are still our building blocks for defining anything from a line to a spline and all the Curve types in between.



1. Line
2. Polyline
3. Arc
4. Circle
5. Ellipse
6. NURBS Curve
7. Polycurve

What's a Curve?

The term **Curve** is generally a catch-all for all different sort of curved (even if straight) shapes. Capital "C" Curve is the parent categorization for all of those shape types - Lines, Circles, Splines, etc. More technically, a Curve describes every possible Point that can be found by inputting "t" into a collection of functions, which may range from the simple (`x = -1.26*t, y = t`) to functions involving calculus. No matter what kind of Curve we are working with, this **Parameter** called "t" is a property we can evaluate. Furthermore, regardless of the look of the shape, all Curves also have a start point and end point, which coincidentally align with the minimum and maximum t values used to create the Curve. This also helps us understand its directionality.

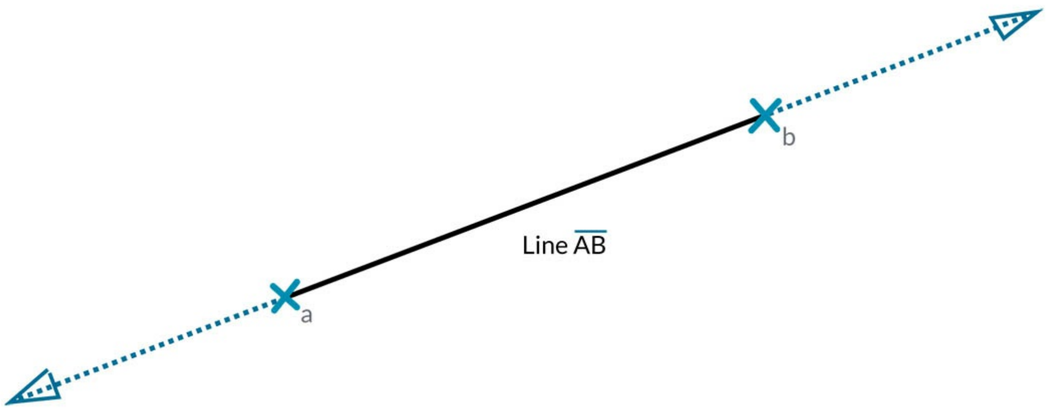


It's important to note that Dynamo assumes that the domain of "t" values for a Curve is understood to be 0.0 to 1.0.

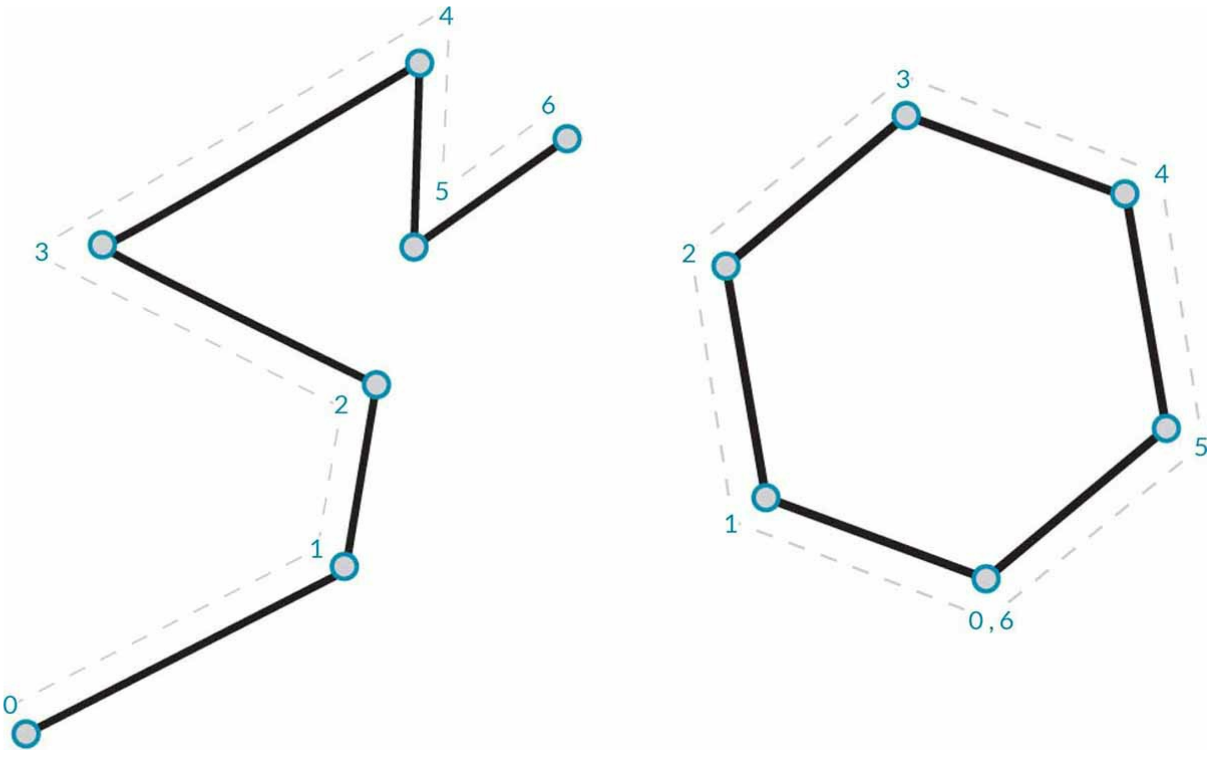
All Curves also possess a number of properties or characteristics which can be used to describe or analyze them. When the distance between the start and end points is zero, the curve is "closed." Also, every curve has a number of control-points, if all these points are located in the same plane, the curve is "planar." Some properties apply to the curve as a whole, while others only apply to specific points along the curve. For example, planarity is a global property while a tangent vector at a given t value is a local property.

Lines

Lines are the simplest form of Curves. They may not look curvy but they are in fact Curves - just without any curvature. There are a few different ways to create Lines, the most intuitive being from Point A to Point B. The shape of the Line AB will be drawn between the points but mathematically it extends infinitely in both directions.

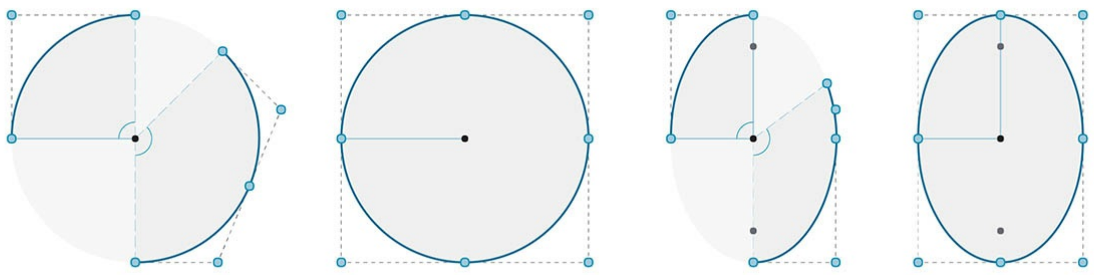


When we connect two Lines together, we have a **Polyline**. Here we have a straightforward representation of what a Control Point is. Editing any of these point locations will change the shape of the Polyline. If the Polyline is closed, we have a Polygon. If the Polygon's edge lengths are all equal, it is described as regular.



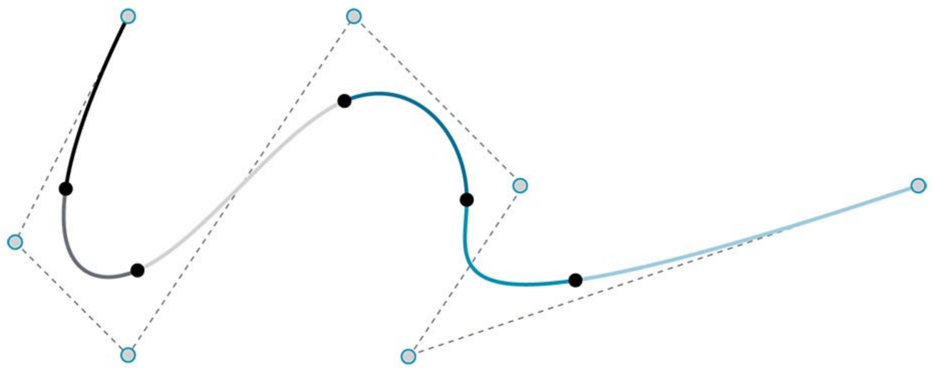
Arcs, Circles, Ellipse Arcs, and Ellipses

As we add more complexity to the Parametric Functions that define a shape, we can take one step further from a Line to create an **Arc**, **Circle**, **Ellipse Arc**, or **Ellipse** by describing one or two radii. The differences between the Arc version and the Circle or Ellipse is only whether or not the shape is closed.



NURBS + Polycurves

NURBS (Non-uniform Rational Basis Splines) are mathematical representations that can accurately model any shape from a simple two dimensional Line, Circle, Arc, or Rectangle to the most complex three-dimensional free-form organic Curve. Because of their flexibility (relatively few control points, yet smooth interpolation based on Degree settings) and precision (bound by a robust math), NURBS models can be used in any process from illustration and animation to manufacturing.

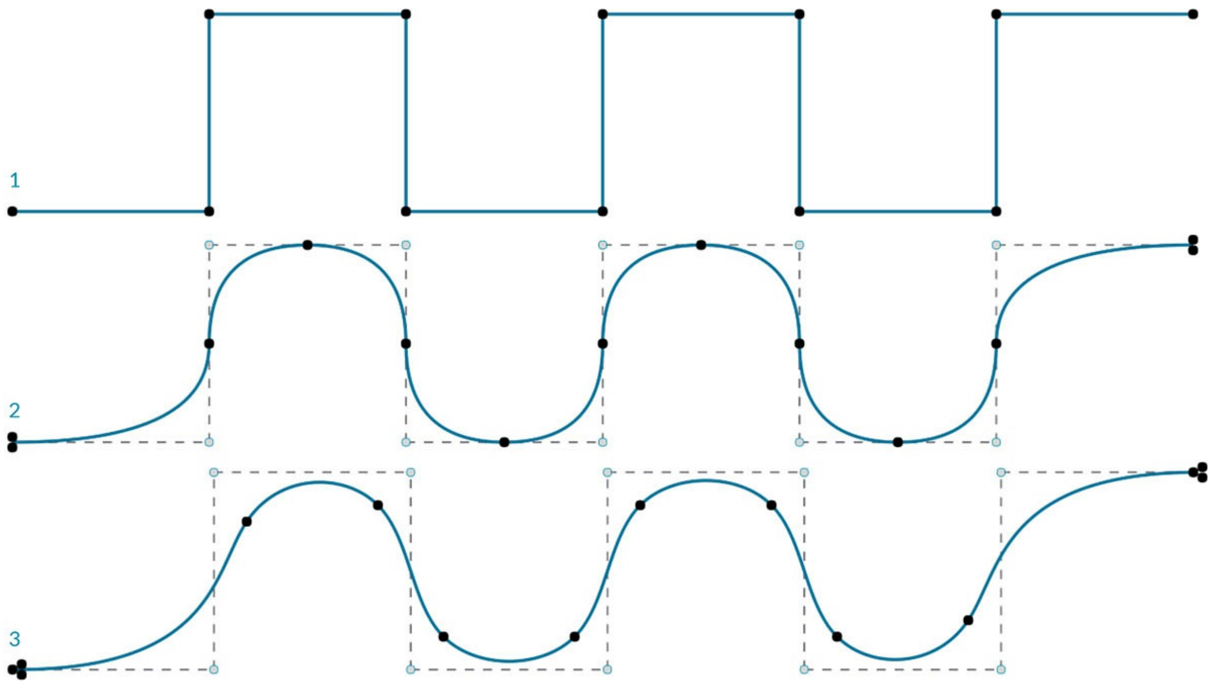


Degree: The Degree of the Curve determines the range of influence the Control Points have on a Curve; where the higher the degree, the larger the range. The Degree is a positive whole number. This number is usually 1, 2, 3 or 5, but can be any positive whole number. NURBS lines and polylines are usually Degree 1 and most free-form Curves are Degree 3 or 5.

Control Points: The Control Points are a list of at least Degree+1 Points. One of the easiest ways to change the shape of a NURBS Curve is to move its Control Points.

Weight: Control Points have an associated number called a Weight. Weights are usually positive numbers. When a Curve's Control Points all have the same weight (usually 1), the Curve is called non-rational, otherwise the Curve is called rational. Most NURBS curves are non-rational.

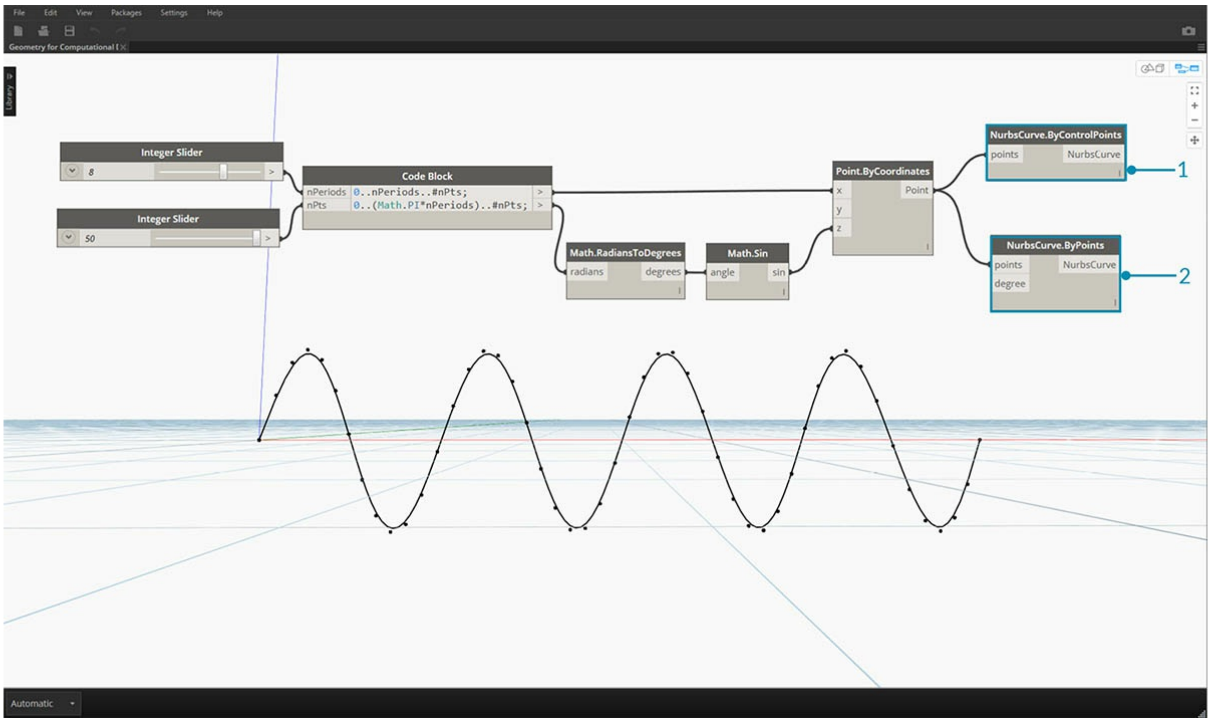
Knots: Knots are a list of (Degree+N-1) numbers, where N is the number of Control Points. The Knots are used together with the weights to control the influence of the Control Points on the resulting Curve. One use for Knots is to create kinks at certain points in the curve.



1. Degree = 1
2. Degree = 2
3. Degree = 3

Note that the higher the degree value, the more Control Points are used to interpolate the resulting Curve.

Let's make a sine curve in Dynamo using two different methods to create NURBS Curves to compare the results.



```
0..nPeriods..#nPts;
0..(Math.PI*nPeriods)..#nPts;
```

1. *NurbsCurve.ByControlPoints* uses the List of Points as Control Points
2. *NurbsCurve.ByPoints* draws a Curve through the List of Points

Download the example file that accompanies this image (Right click and "Save Link As..."): Geometry for Computational Design - Curves.dyn. A full list of example files can be found in the Appendix.

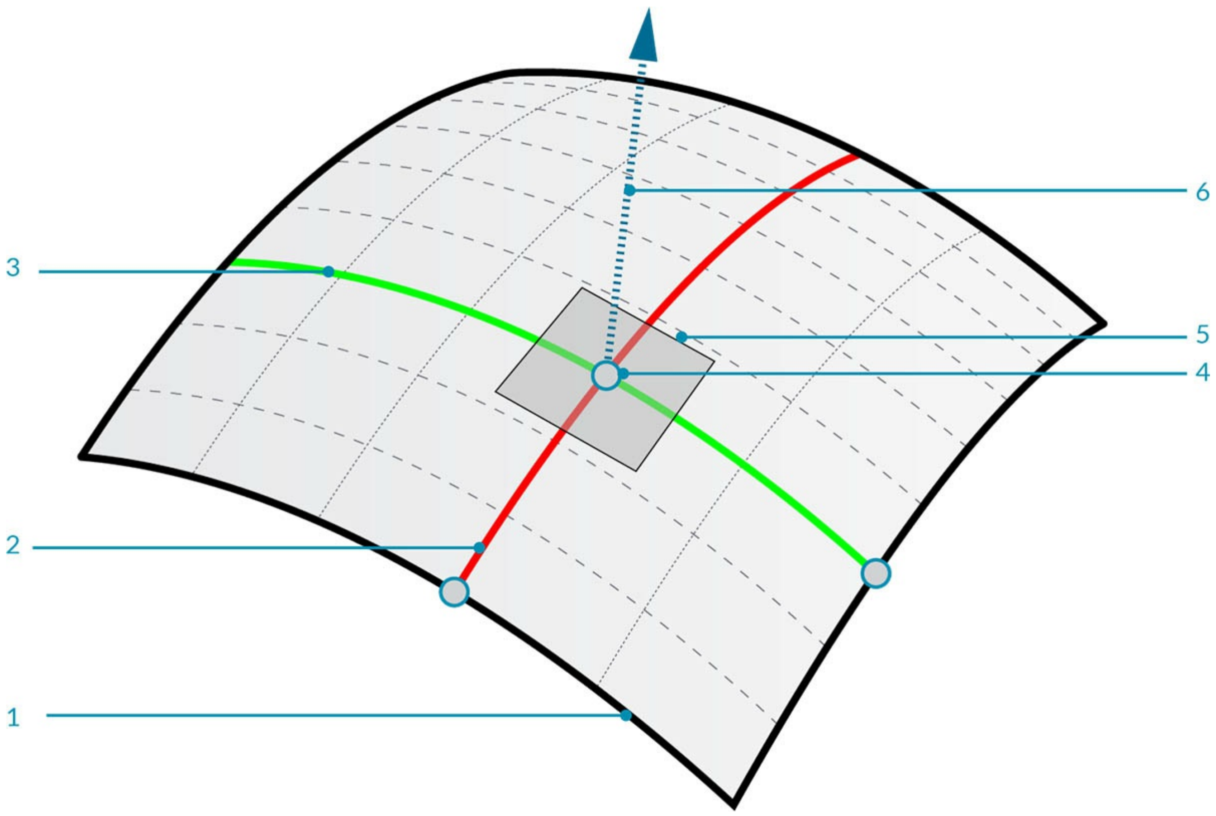
Surfaces

Surfaces

As we move from using Curves to using Surfaces in a model, we can now begin to represent objects we see in our three dimensional world. While Curves are not always planar ie. they are three dimensional, the space they define is always bound to one dimension. Surfaces give us another dimension and a collection of additional properties we can use within other modeling operations.

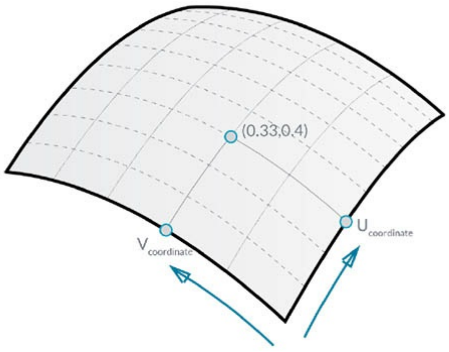
What's a Surface?

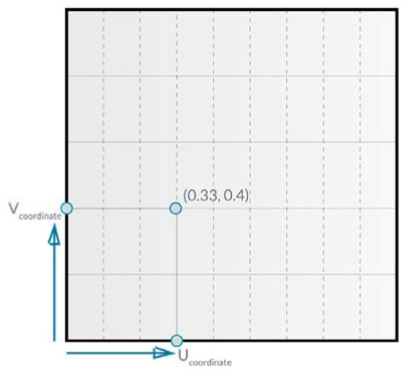
A Surface is a mathematical shape defined by a function and two parameters, Instead of `t` for Curves, we use `U` and `V` to describe the corresponding parameter space. This means we have more geometrical data to draw from when working with this type of Geometry. For example, Curves have tangent vectors and normal planes (which can rotate or twist along the curve's length), whereas Surfaces have normal vectors and tangent planes that will be consistent in their orientation.



1. Surface
2. U Isocurve
3. V Isocurve
4. UV Coordinate
5. Perpendicular Plane
6. Normal Vector

Surface Domain: A surface domain is defined as the range of (U,V) parameters that evaluate into a three dimensional point on that surface. The domain in each dimension (U or V) is usually described as two numbers (U Min to U Max) and (V Min to V Max).

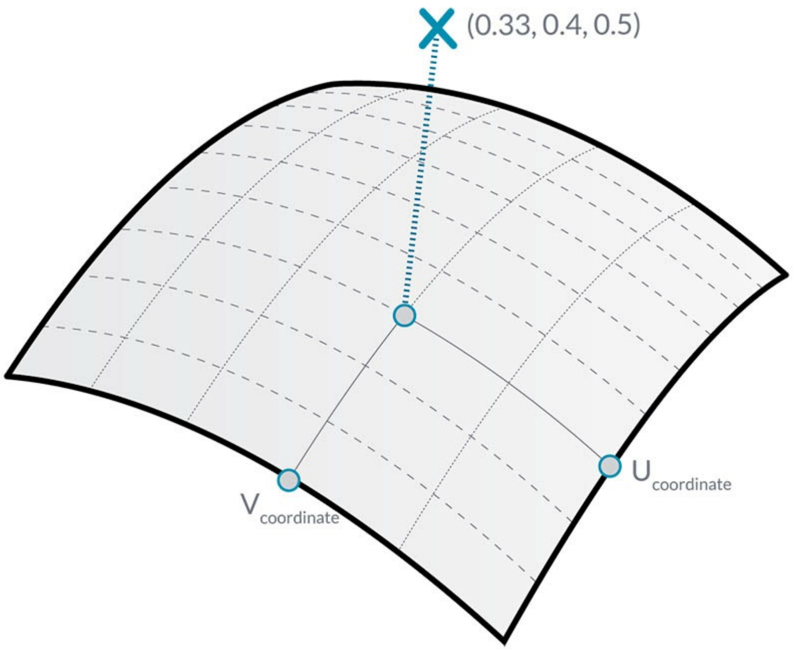




Although the shape of the Surface by not look "rectangular" and it locally may have a tighter or looser set of isocurves, the "space" defined by its domain is always two dimensional. In Dynamo, Surfaces are always understood to have a domain defined by a minimum of 0.0 and maximum of 1.0 in both U and V directions. Planar or trimmed Surfaces may have different domains.

Isocurve (or Isoparametric Curve): A curve defined by a constant U or V value on the surface and a domain of values for the corresponding other U or V direction.

UV Coordinate: The Point in UV Parameter Space defined by U, V, and sometimes W.

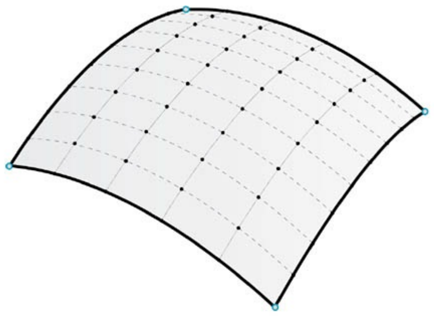


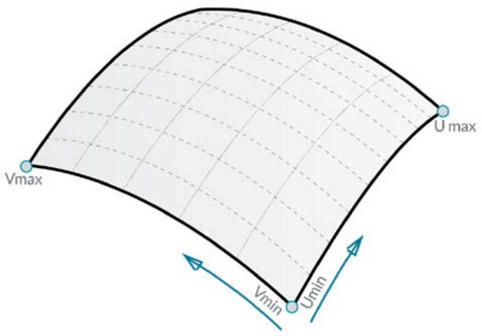
Perpendicular Plane: A Plane that is perpendicular to both U and V Isocurves at a given UV Coordinate.

Normal Vector: A Vector defining the direction of "up" relative to the Perpendicular Plane.

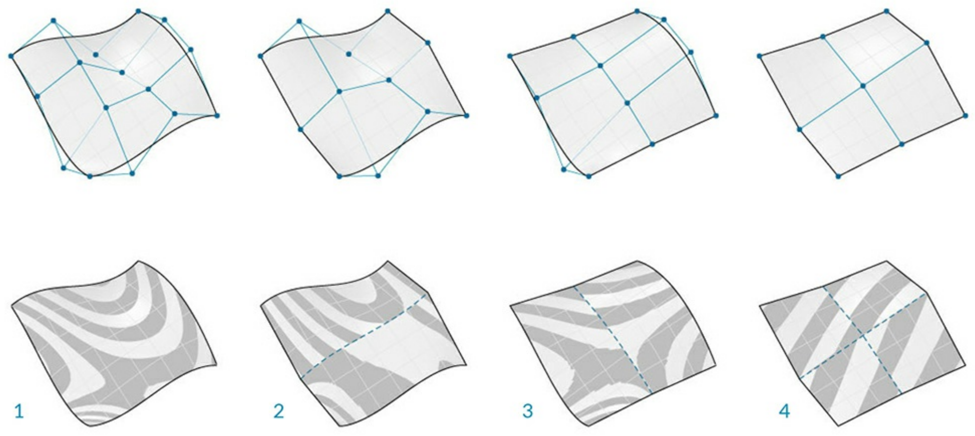
NURBS Surfaces

NURBS Surfaces are very similar to NURBS curves. You can think of NURBS Surfaces as a grid of NURBS Curves that go in two directions. The shape of a NURBS Surface is defined by a number of control points and the degree of that surface in the U and V directions. The same algorithms are used to calculate shape, normals, tangents, curvatures and other properties by way of control points, weights and degree.





In the case of NURBS surfaces, there are two directions implied by the geometry, because NURBS surfaces are, regardless of the shape we see, rectangular grids of control points. And even though these directions are often arbitrary relative to the world coordinate system, we will use them frequently to analyze our models or generate other geometry based on the Surface.

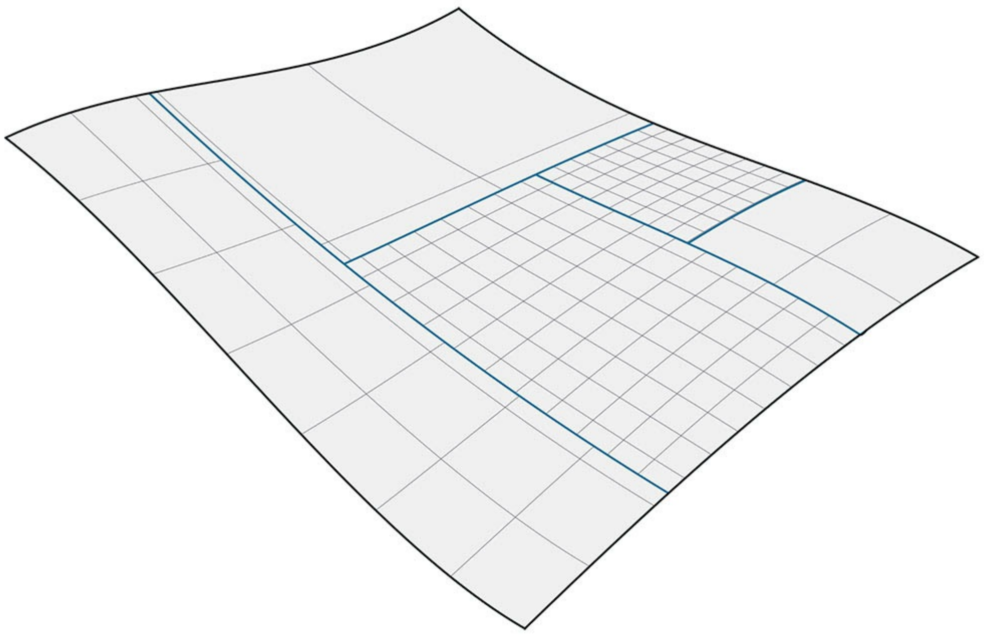


1. Degree (U,V) = (3,3)
2. Degree (U,V) = (3,1)
3. Degree (U,V) = (1,2)
4. Degree (U,V) = (1,1)

Polysurfaces

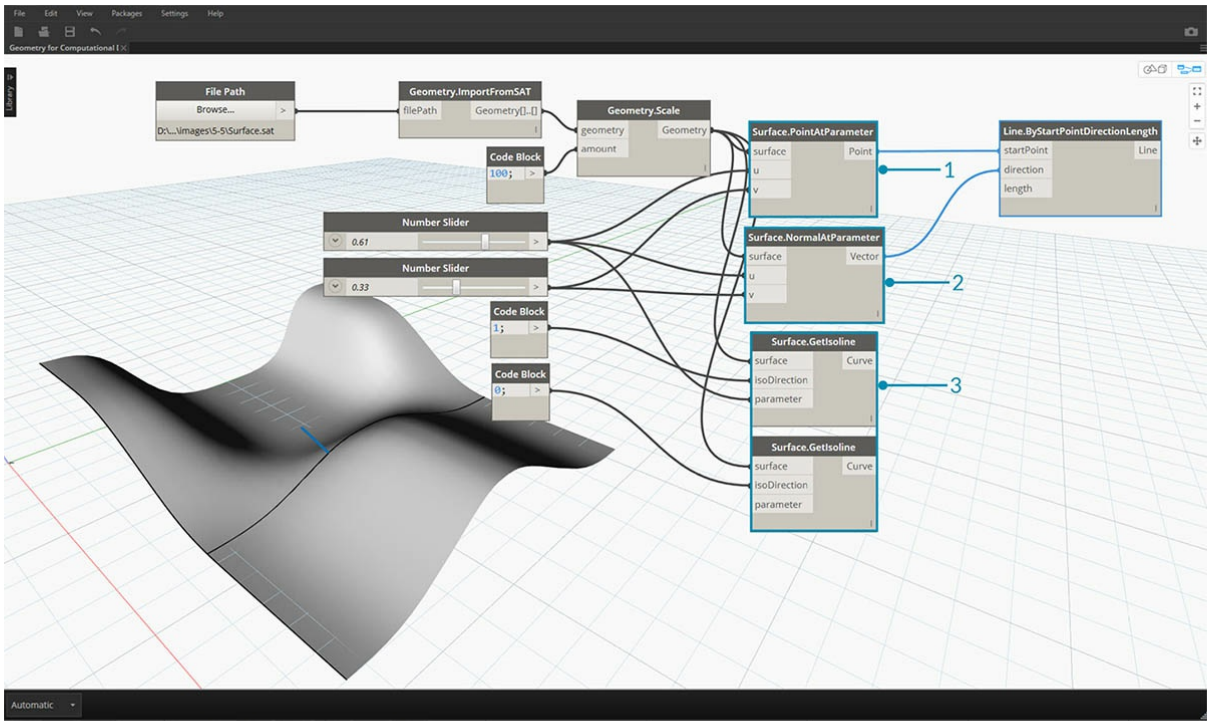
Polysurfaces are composed of Surfaces that are joined across an edge. Polysurfaces offer more than two dimensional UV definition in that we can now move through the connected shapes by way of their Topology.

> While "Topology" generally describes a concept around how parts are connected and/or related Topology in Dynamo is also a type of Geometry. Specifically it is a parent category for Surfaces, Polysurfaces, and Solids.



Sometimes called patches, joining Surfaces in this manner allows us to make more complex shapes as well as define detail across the seam. Conveniently we can apply a fillet or chamfer operation to the edges of a Polysurface.

Let's import and evaluate a Surface at a Parameter in Dynamo to see what kind of information we can extract.



1. *Surface.PointAtParameter* returns the Point at a given UV Coordinate
2. *Surface.NormalAtParameter* returns the Normal Vector at a given UV Coordinate
3. *Surface.GetIsoline* returns the Isoparametric Curve at a U or V Coordinate - note the isoDirection input.

Download the example files that accompanies this image (Right click and "Save Link As..."). A full list of example files can be found in the Appendix.

1. Geometry for Computational Design - Surfaces.dyn
2. Surface.sat

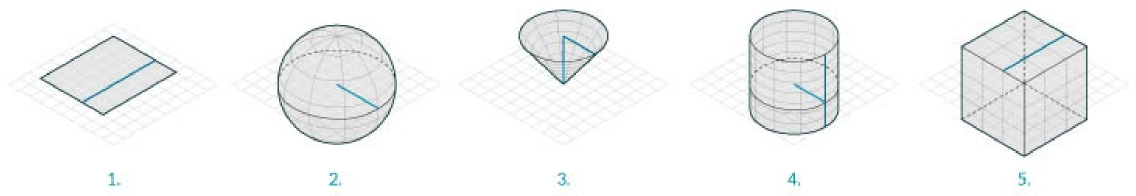
Solids

Solids

If we want to construct more complex models that cannot be created from a single surface or if we want to define an explicit volume, we must now venture into the realm of Solids (and Polysurfaces). Even a simple cube is complex enough to need six surfaces, one per face. Solids give access to two key concepts that Surfaces do not - a more refined topological description (faces, edges, vertices) and Boolean operations.

What's a Solid?

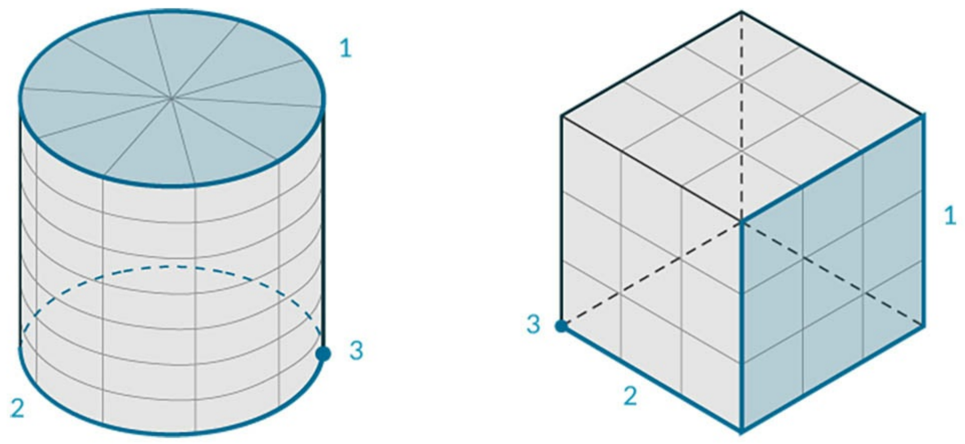
Solids consist of one or more Surfaces that contain volume by way of a closed boundary that defines "in" or "out." Regardless of how many of these Surfaces there are, they must form a "watertight" volume to be considered a Solid. Solids can be created by joining Surfaces or Polysurfaces together or by using operations such as loft, sweep, and revolve. Sphere, Cube, Cone and Cylinder primitives are also Solids. A Cube with at least one face removed counts as a Polysurface, which has some similar properties, but it is not a Solid.



1. A Plane is made of a single Surface and is not a Solid.
2. A Sphere is made of one Surface but *is* a Solid.
3. A Cone is made of two surfaces joined together to make a Solid.
4. A Cylinder is made of three surfaces joined together to make a Solid.
5. A Cube is made of six surfaces joined together to make a Solid.

Topology

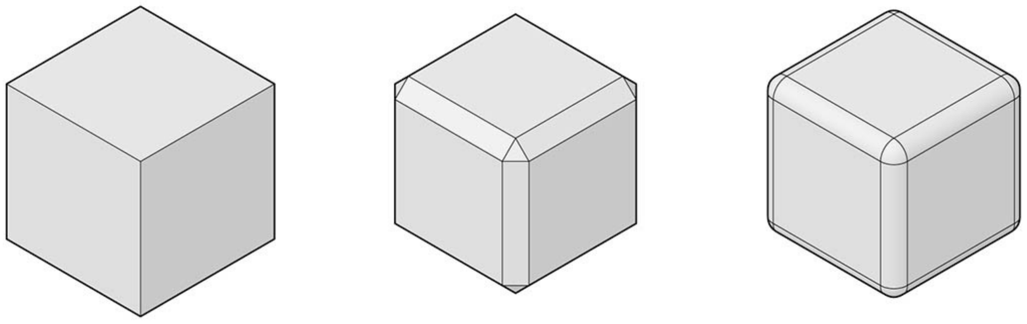
Solids are made up of three types of elements: Vertices, Edges, and Faces. Faces are the surfaces that make up the Solid. Edges are the Curves that define the connection between adjacent faces, and vertices are the start and end points of those Curves. These elements can be queried using the Topology nodes.



1. Faces
2. Edges
3. Vertices

Operations

Solids can be modified by filleting or chamfering their edges to eliminate sharp corners and angles. The chamfer operation creates a ruled surface between two faces, while a fillet blends between faces to maintain tangency.



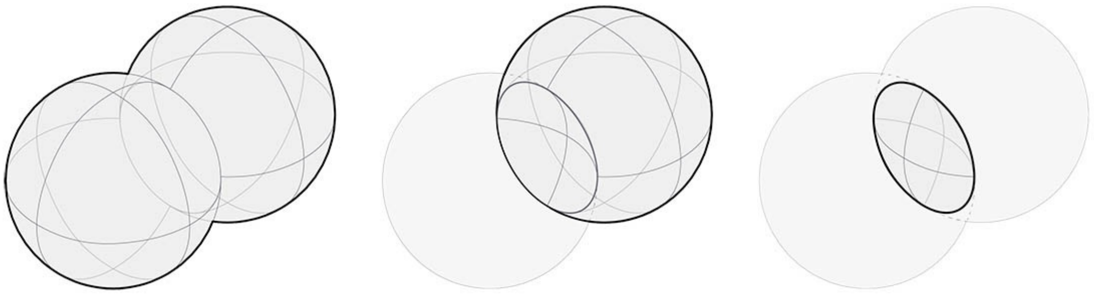
1. Solid Cube
2. Chamfered Cube
3. Filleted Cube

Boolean Operations

Solid Boolean operations are methods for combining two or more Solids. A single Boolean operation actually means performing four operations:

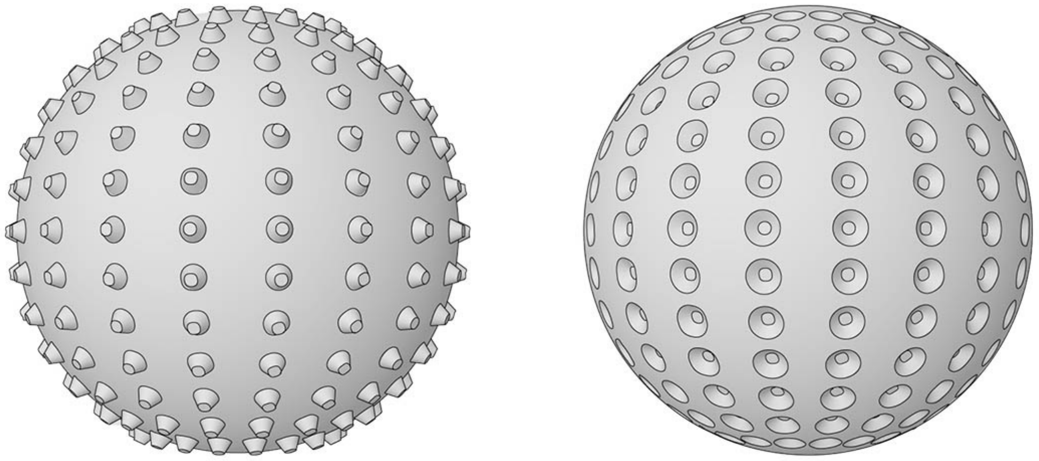
1. **Intersect** two or more objects.
2. **Split** them at the intersections.
3. **Delete** unwanted portions of the geometry.
4. **Join** everything back together.

This makes Solid Booleans a powerful time-saving process. There are three Solid Boolean operations that distinguish which parts of the geometry are kept.



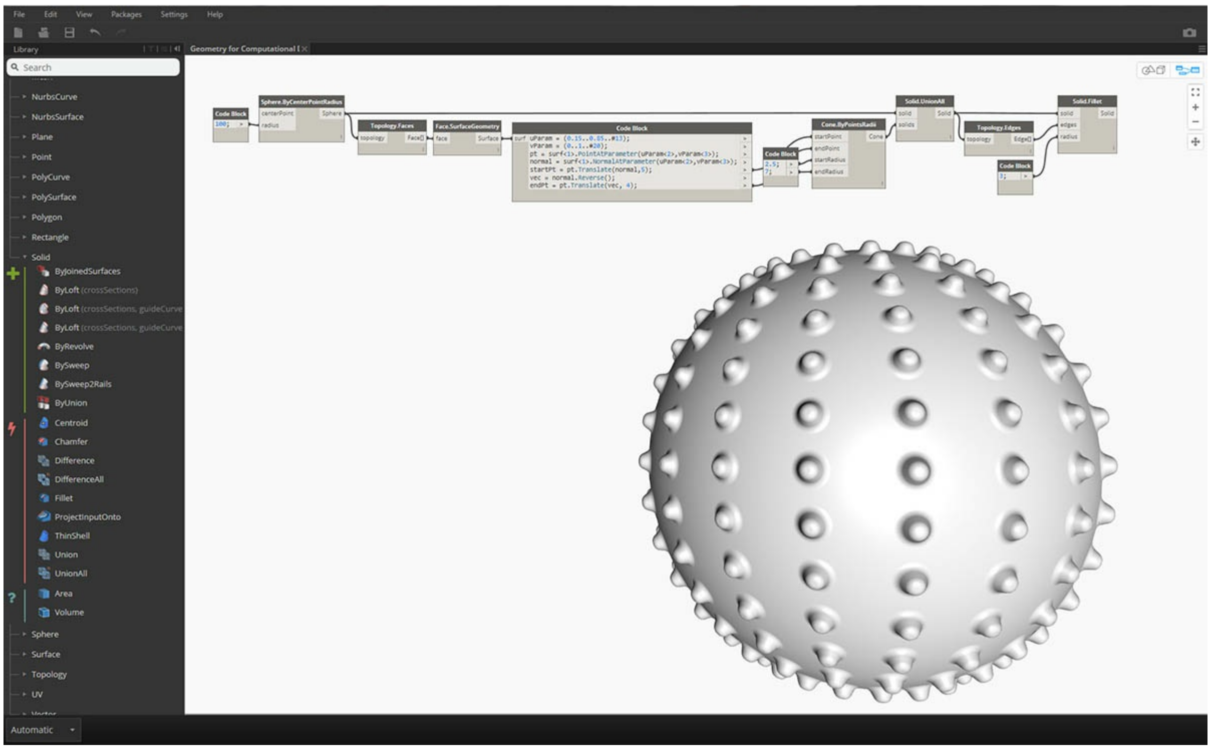
1. **Union:** Remove the overlapping portions of the Solids and join them into a single Solid.
2. **Difference:** Subtract one Solid from another. The Solid to be subtracted is referred to as a tool. Note that you could switch which Solid is the tool to keep the inverse volume.
3. **Intersection:** Keep only the intersecting volume of the two Solids.

In addition to these three operations, Dynamo has **Solid.DifferenceAll** and **Solid.UnionAll** nodes for performing difference and union operations with multiple Solids.



1. **UnionAll:** Union operation with sphere and outward-facing cones
2. **DifferenceAll:** Difference operation with sphere and inward-facing cones

Let's use a few Boolean operations to create a spiky ball.



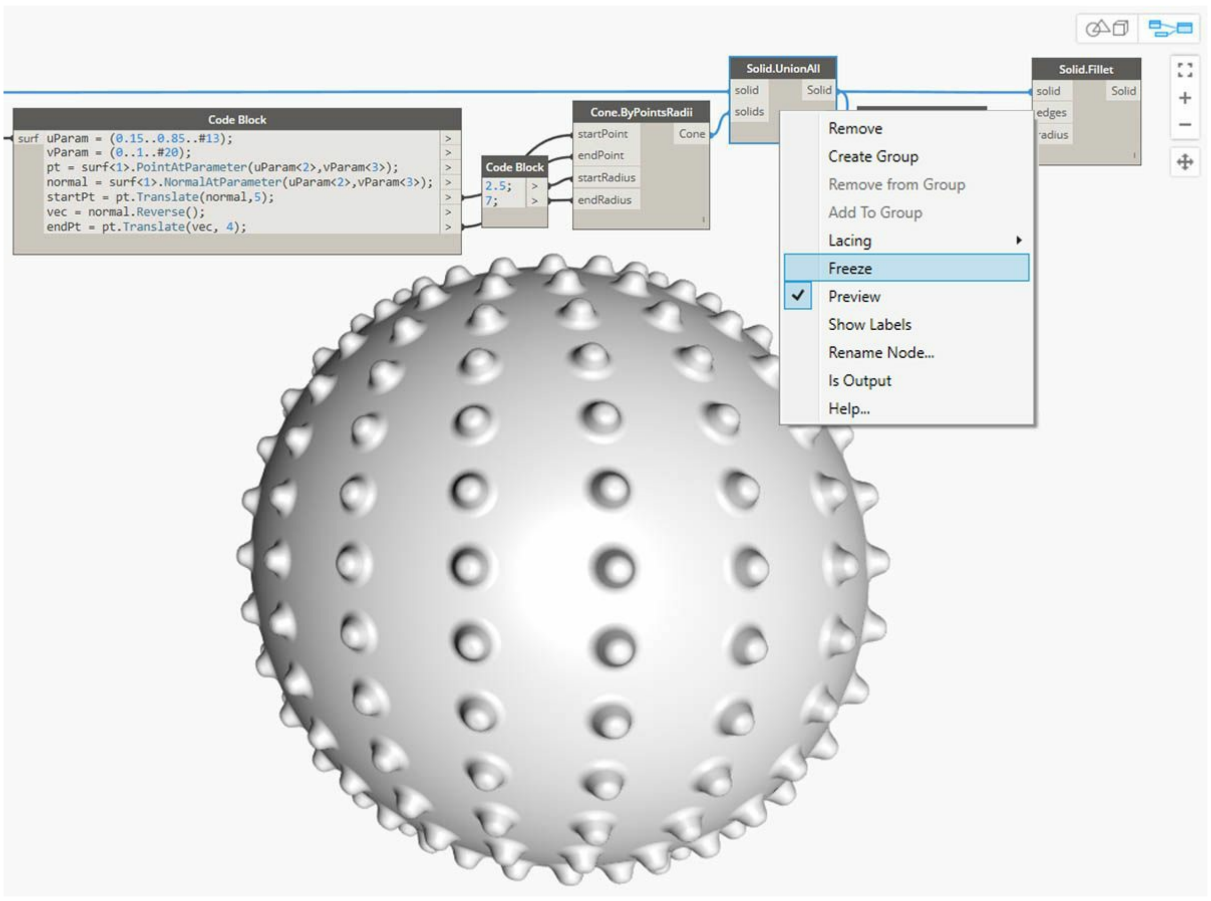
```
uParam = (0.15..0.85..#13);
vParam = (0..1..#20);
pt = surf<1>.PointAtParameter(uParam<2>,vParam<3>);
normal = surf<1>.NormalAtParameter(uParam<2>,vParam<3>);
startPt = pt.Translate(normal,5);
vec = normal.Reverse();
endPt = pt.Translate(vec, 4);
```

1. **Sphere.ByCenterPointRadius**: Create the base Solid.
2. **Topology.Faces**, **Face.SurfaceGeometry**: Query the faces of the Solid and convert to surface geometry—in this case, the Sphere has only one Face.
3. **Cone.ByPointsRadii**: Construct cones using points on the surface.
4. **Solid.UnionAll**: Union the Cones and the Sphere.
5. **Topology.Edges**: Query the edges of the new Solid
6. **Solid.Fillet**: Fillet the Edges of the spiky ball

Download the example files that accompany this image (Right click and "Save Link As..."). A full list of example files can be found in the Appendix. Geometry for Computational Design - Solids.dyn

Freezing

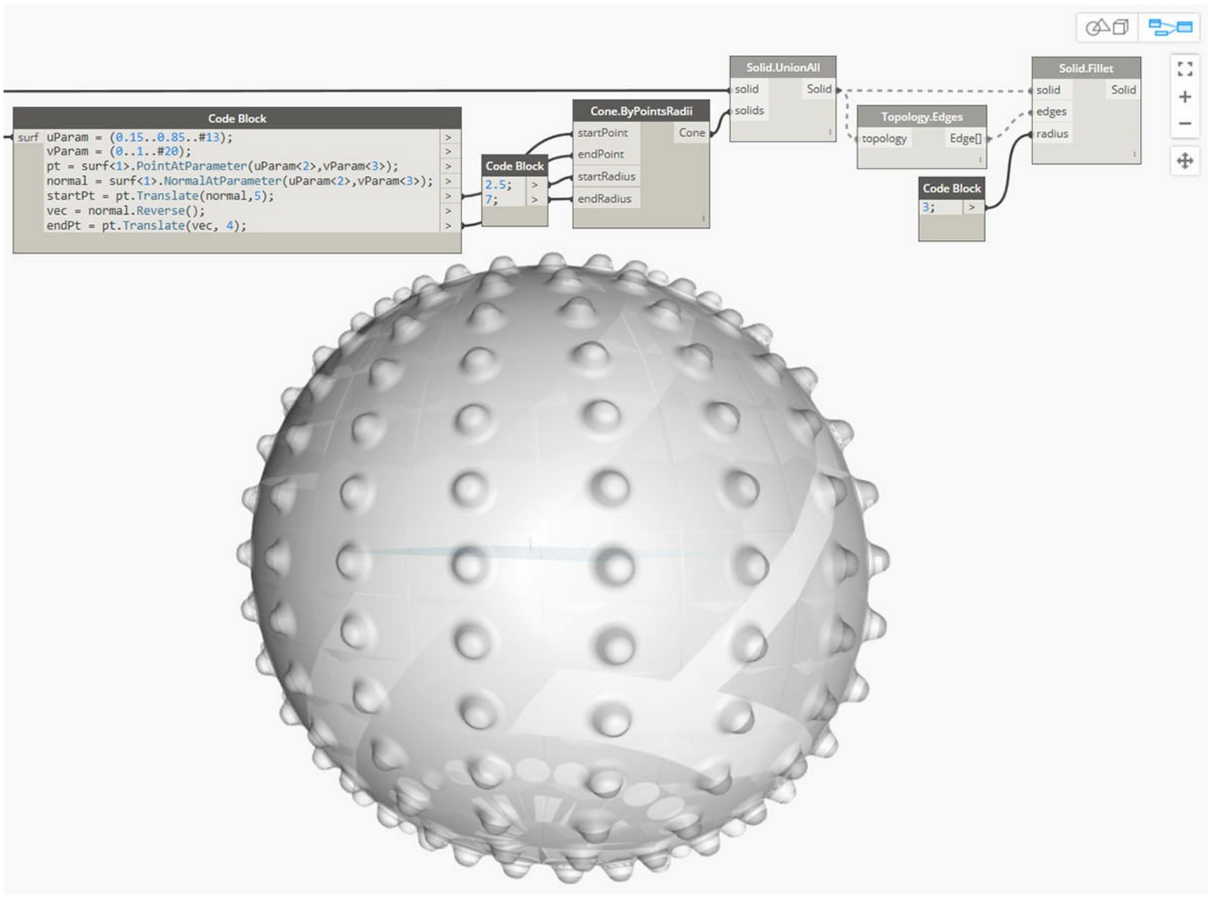
Boolean operations are complex and can be slow to calculate. Use Freeze functionality to suspend the execution of selected nodes and affected downstream nodes.



```
uParam = (0.15..0.85..#13);
vParam = (0..1..#20);
pt = surf<1>.PointAtParameter(uParam<2>,vParam<3>);
normal = surf<1>.NormalAtParameter(uParam<2>,vParam<3>);
startPt = pt.Translate(normal,5);
vec = normal.Reverse();
endPt = pt.Translate(vec, 4);
```

```
2.5;
7;
```

Use the right-click contextual menu to Freeze the Solid Union operation

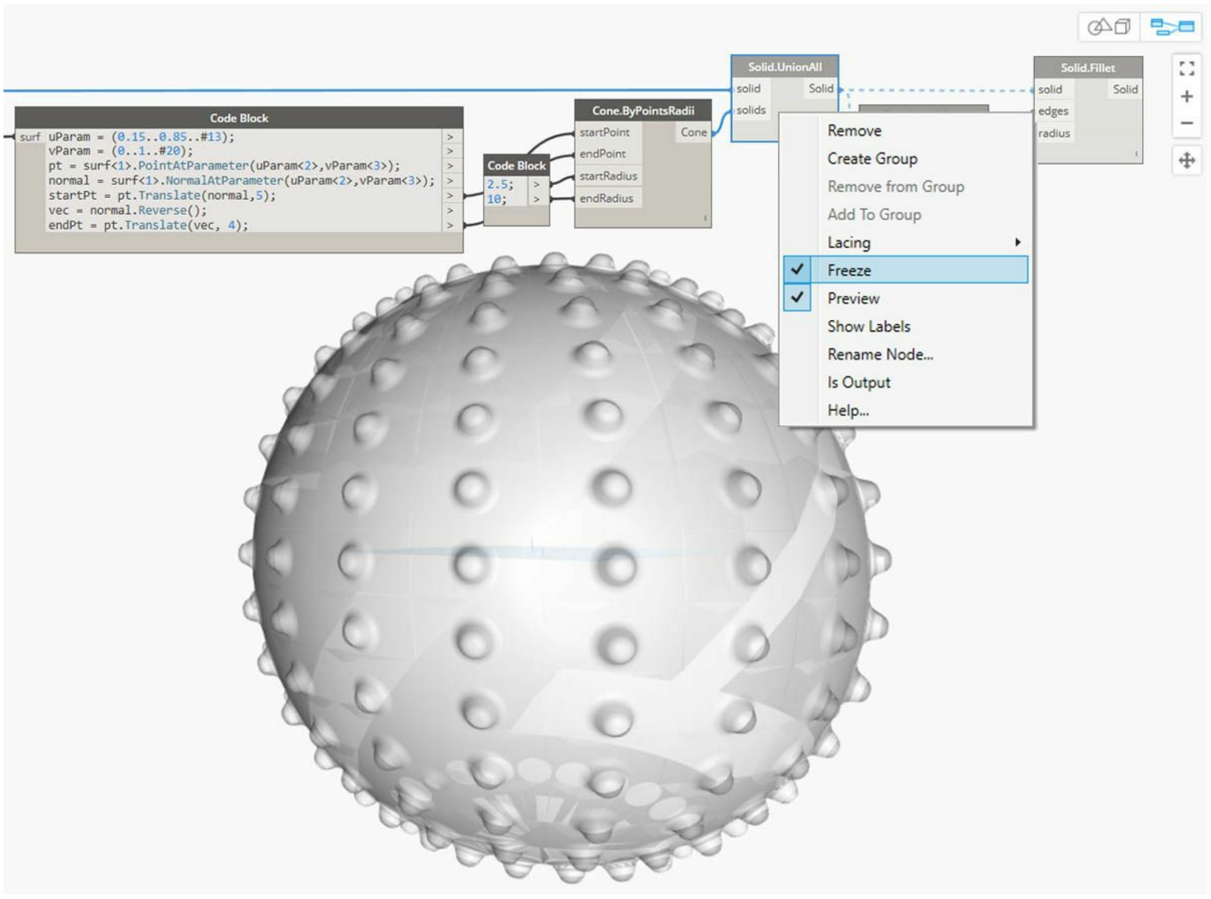


```
uParam = (0.15..0.85..#13);
vParam = (0..1..#20);
pt = surf<1>.PointAtParameter(uParam<2>,vParam<3>);
normal = surf<1>.NormalAtParameter(uParam<2>,vParam<3>);
startPt = pt.Translate(normal,5);
vec = normal.Reverse();
endPt = pt.Translate(vec, 4);
```

```
2.5;
7;
```

```
3;
```

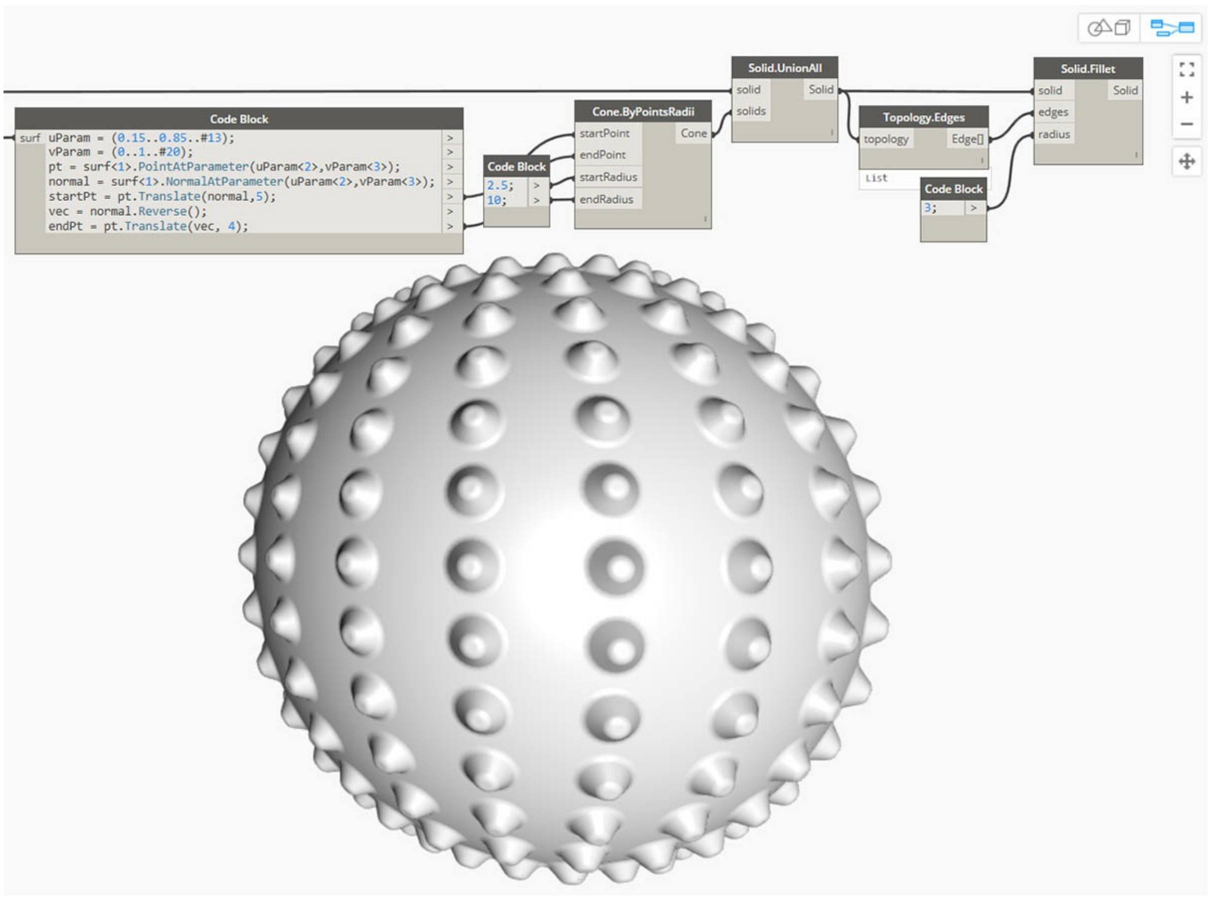
The selected node and all downstream nodes will preview in a light grey ghosted mode, and affected wires will be displayed as dashed lines. The affected geometry preview will also be ghosted. You can now change values upstream without calculating the boolean union.



```
uParam = (0.15..0.85..#13);
vParam = (0..1..#20);
pt = surf<1>.PointAtParameter(uParam<2>,vParam<3>);
normal = surf<1>.NormalAtParameter(uParam<2>,vParam<3>);
startPt = pt.Translate(normal,5);
vec = normal.Reverse();
endPt = pt.Translate(vec, 4);
```

```
2.5;
10;
```

To unfreeze the nodes, right-click and uncheck Freeze.



```
uParam = (0.15..0.85..#13);
vParam = (0..1..#20);
pt = surf<1>.PointAtParameter(uParam<2>,vParam<3>);
normal = surf<1>.NormalAtParameter(uParam<2>,vParam<3>);
startPt = pt.Translate(normal,5);
vec = normal.Reverse();
endPt = pt.Translate(vec, 4);
```

```
2.5;
10;
```

```
3;
```

All affected nodes and associated geometry previews will update and revert to the standard preview mode.

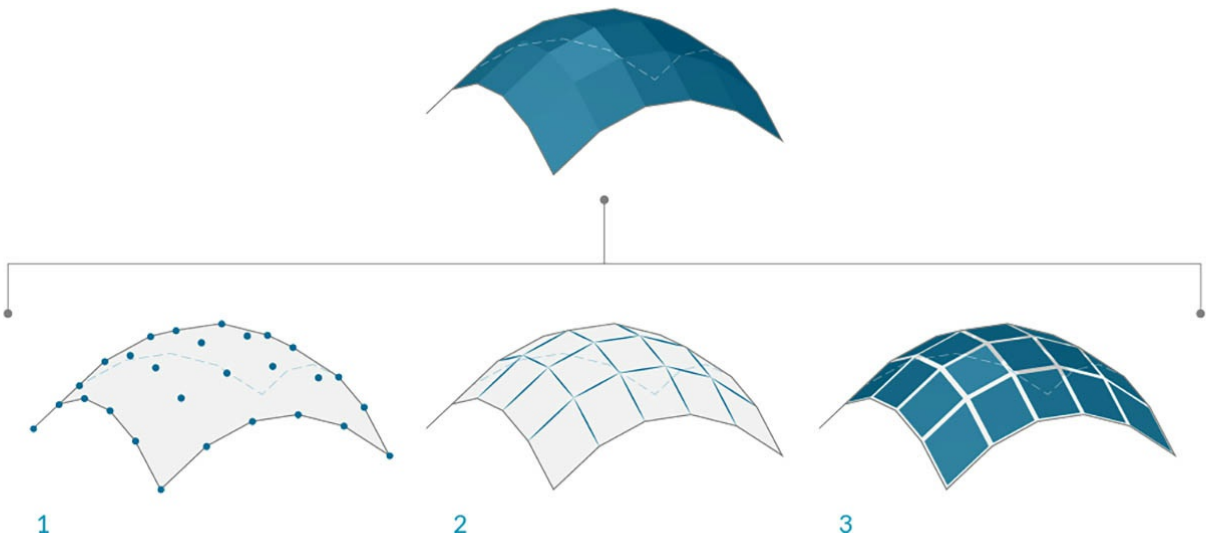
Meshes

Meshes

In the field of computational modeling, Meshes are one of the most pervasive forms of representing 3D geometry. Mesh geometry can be a light-weight and flexible alternative to working with NURBS, and Meshes are used in everything from rendering and visualizations to digital fabrication and 3D printing.

What's a Mesh?

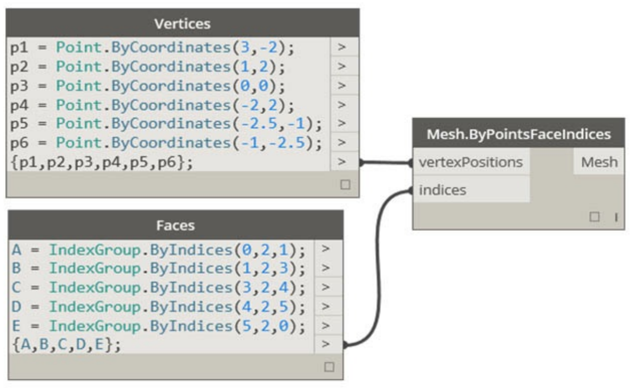
A Mesh is a collection of quadrilaterals and triangles that represents a surface or solid geometry. Like Solids, the structure of a Mesh object includes vertices, edges, and faces. There are additional properties that make Meshes unique as well, such as normals.



1. Mesh vertices
2. Mesh edges *Edges with only one adjoining face are called "Naked." All other edges are "Clothed"
3. Mesh faces

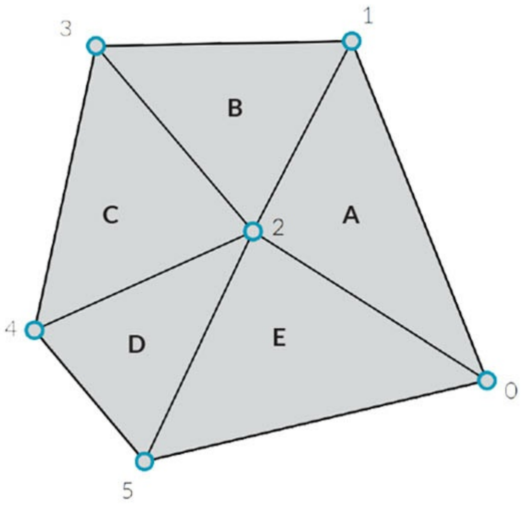
Mesh Elements

Dynamo defines Meshes using a Face-Vertex data structure. At its most basic level, this structure is simply a collection of points which are grouped into polygons. The points of a Mesh are called vertices, while the surface-like polygons are called faces. To create a Mesh we need a list of vertices and a system of grouping those vertices into faces called an index group.



```
p1 = Point.ByCoordinates(3,-2);
p2 = Point.ByCoordinates(1,2);
p3 = Point.ByCoordinates(0,0);
p4 = Point.ByCoordinates(-2,2);
p5 = Point.ByCoordinates(-2.5,-1);
p6 = Point.ByCoordinates(-1,-2.5);
{p1,p2,p3,p4,p5,p6};
```

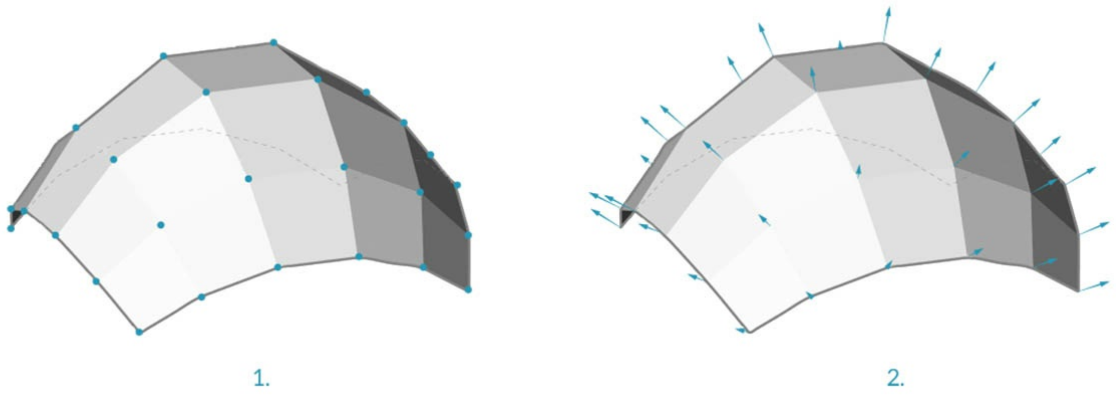
```
A = IndexGroup.ByIndices(0,2,1);
B = IndexGroup.ByIndices(1,2,3);
C = IndexGroup.ByIndices(3,2,4);
D = IndexGroup.ByIndices(4,2,5);
E = IndexGroup.ByIndices(5,2,0);
{A,B,C,D,E};
```



1. List of vertices
2. List of index groups to define faces

Vertices + Vertex Normals

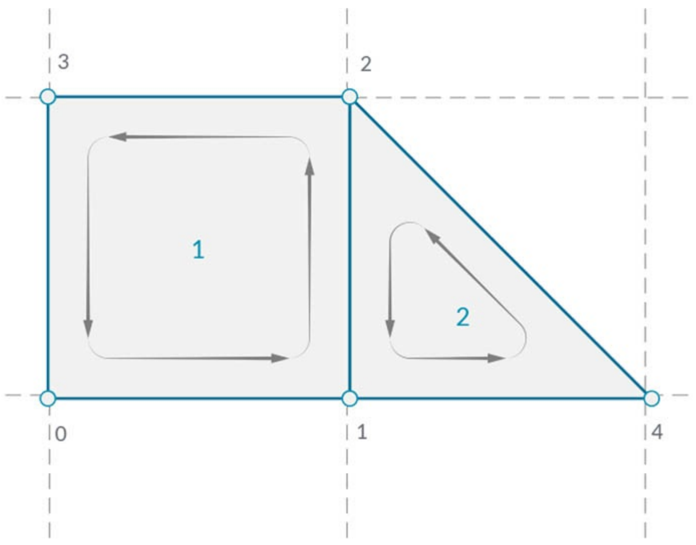
The vertices of a Mesh are simply a list of points. The index of the vertices is very important when constructing a Mesh, or getting information about the structure of a Mesh. For each vertex, there is also a corresponding vertex normal (vector) which describes the average direction of the attached faces and helps us understand the "in" and "out" orientation of the Mesh.



1. Vertices
2. Vertex Normals

Faces

A face is an ordered list of three or four vertices. The “surface” representation of a Mesh face is therefore implied according to the position of the vertices being indexed. We already have the list of vertices that make up the Mesh, so instead of providing individual points to define a face, we simply use the index of the vertices. This also allows us to use the same vertex in more than one face.



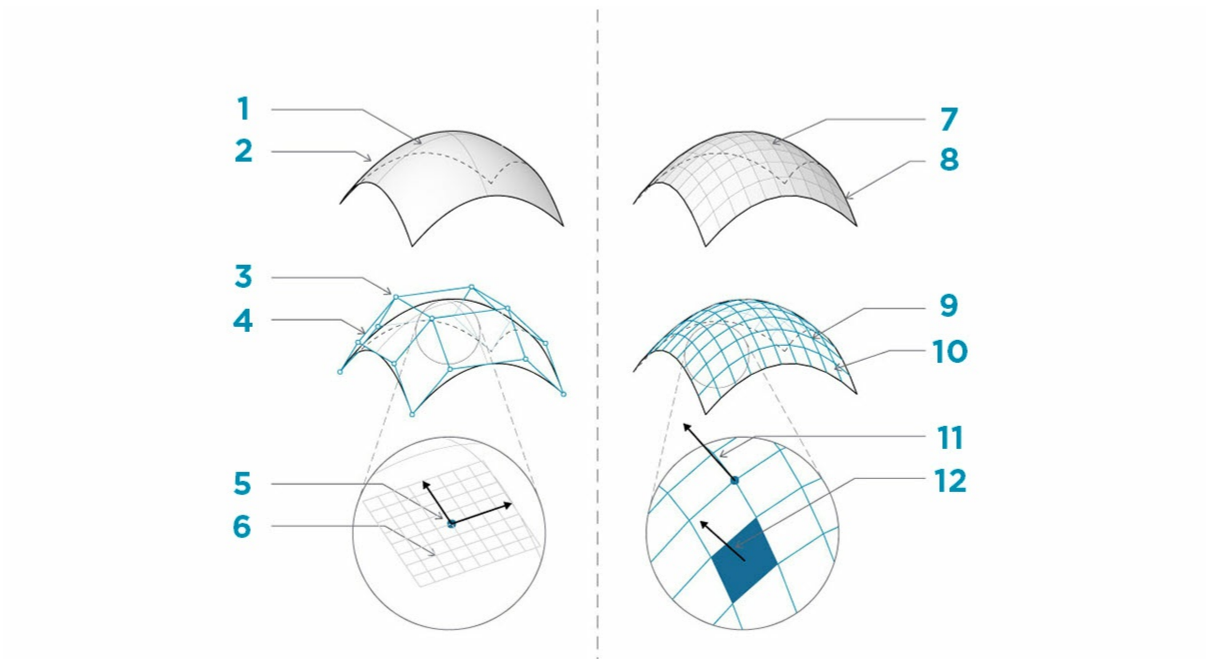
1. A quad face made with indices 0, 1, 2, and 3
2. A triangle face made with indices 1, 4, and 2 Note that the index groups can be shifted in their order - as long as the sequence is ordered in a counter-clockwise manner, the face will be defined correctly

Meshes versus NURBS Surfaces

How is Mesh geometry different from NURBS geometry? When might you want to use one instead of the other?

Parameterization

In a previous chapter, we saw that NURBS surfaces are defined by a series of NURBS curves going in two directions. These directions are labeled `U` and `V`, and allow a NURBs surface to be parameterized according to a two-dimensional surface domain. The curves themselves are stored as equations in the computer, allowing the resulting surfaces to be calculated to an arbitrarily small degree of precision. It can be difficult, however, to combine multiple NURBS surfaces together. Joining two NURBS surfaces will result in a polysurface, where different sections of the geometry will have different UV parameters and curve definitions.

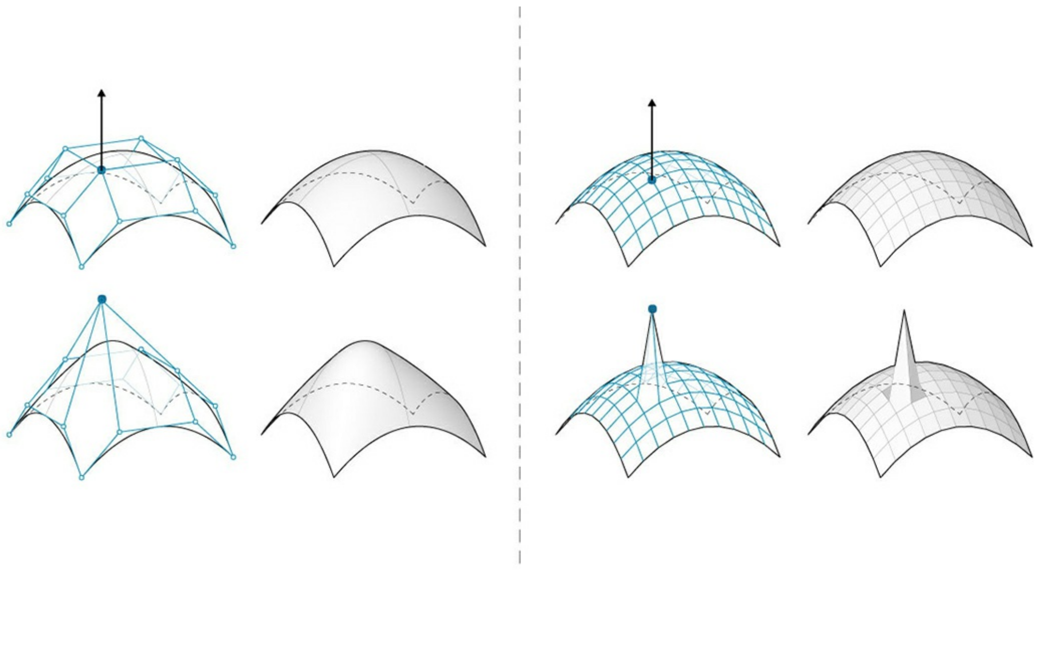


1. Surface
2. Isoparametric (Isoparm) Curve
3. Surface Control Point
4. Surface Control Polygon
5. Isoparametric Point
6. Surface Frame
7. Mesh
8. Naked Edge
9. Mesh Network
10. Mesh Edges
11. Vertex Normal
12. Mesh Face / Mesh Face Normal

Meshes, on the other hand, are comprised of a discrete number of exactly defined vertices and faces. The network of vertices generally cannot be defined by simple UV coordinates, and because the faces are discrete the amount of precision is built into the Mesh and can only be changed by refining the Mesh and adding more faces. The lack of mathematical descriptions allows Meshes to more flexibly handle complex geometry within a single Mesh.

Local versus Global Influence

Another important difference is the extent to which a local change in Mesh or NURBS geometry affects the entire form. Moving one vertex of a Mesh only affects the faces that are adjacent to that vertex. In NURBS surfaces, the extent of the influence is more complicated and depends on the degree of the surface as well as the weights and knots of the control points. In general, however, moving a single control point in a NURBS surface creates a smoother, more extensive change in geometry.



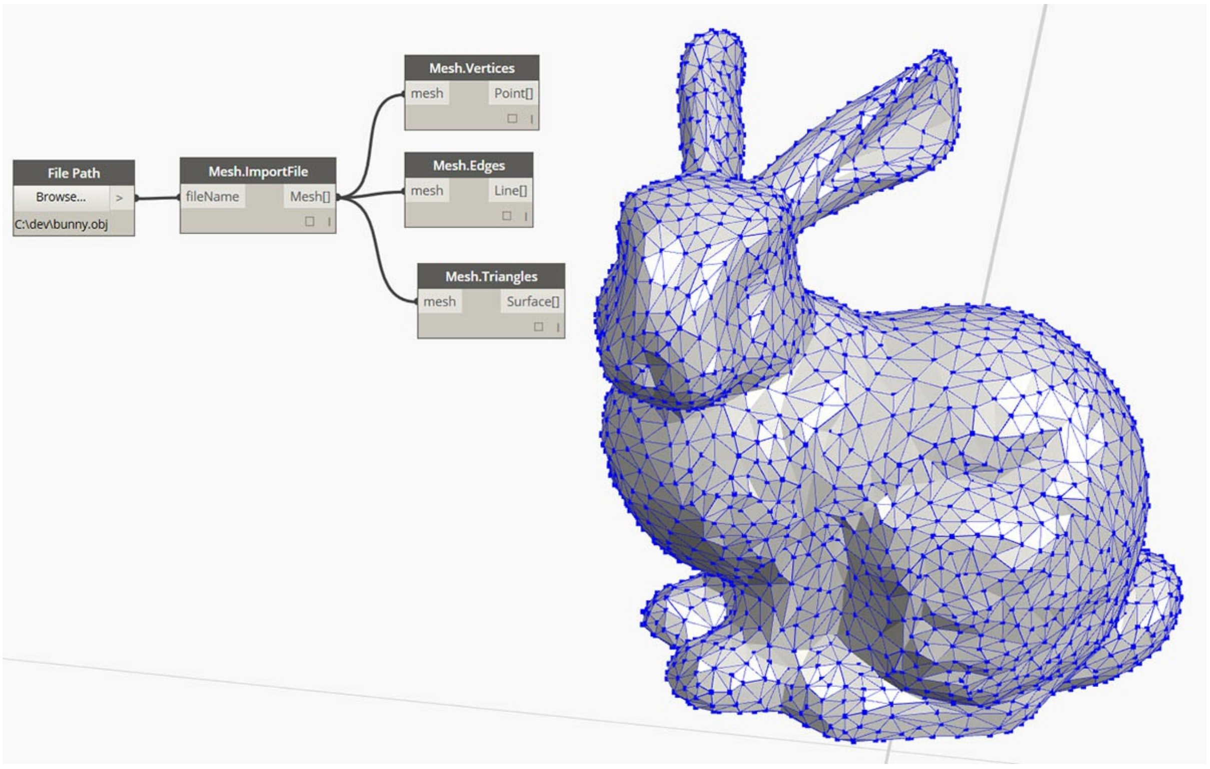
1. NURBS Surface - moving a control point has influence that extends across the shape
2. Mesh geometry - moving a vertex has influence only on adjacent elements

One analogy that can be helpful is to compare a vector image (composed of lines and curves) with a raster image (composed of individual pixels). If you zoom into a vector image, the curves remain crisp and clear, while zooming into a raster image results in seeing individual pixels become larger. In this analogy, NURBS surfaces can be compared to a vector image because there is a smooth mathematical relationship, while a Mesh behaves similarly to a raster image with a set resolution.

Mesh Toolkit

Dynamo's mesh capabilities can be extended by installing the Mesh Toolkit package. The Dynamo Mesh Toolkit provides tools to import Meshes from external file formats, create a Mesh from Dynamo geometry objects, and manually build Meshes by their vertices and indices. The library also provides tools to modify Meshes, repair Meshes, or extract horizontal slices for use in fabrication.

See chapter 10.2 for an example using Mesh Toolkit.



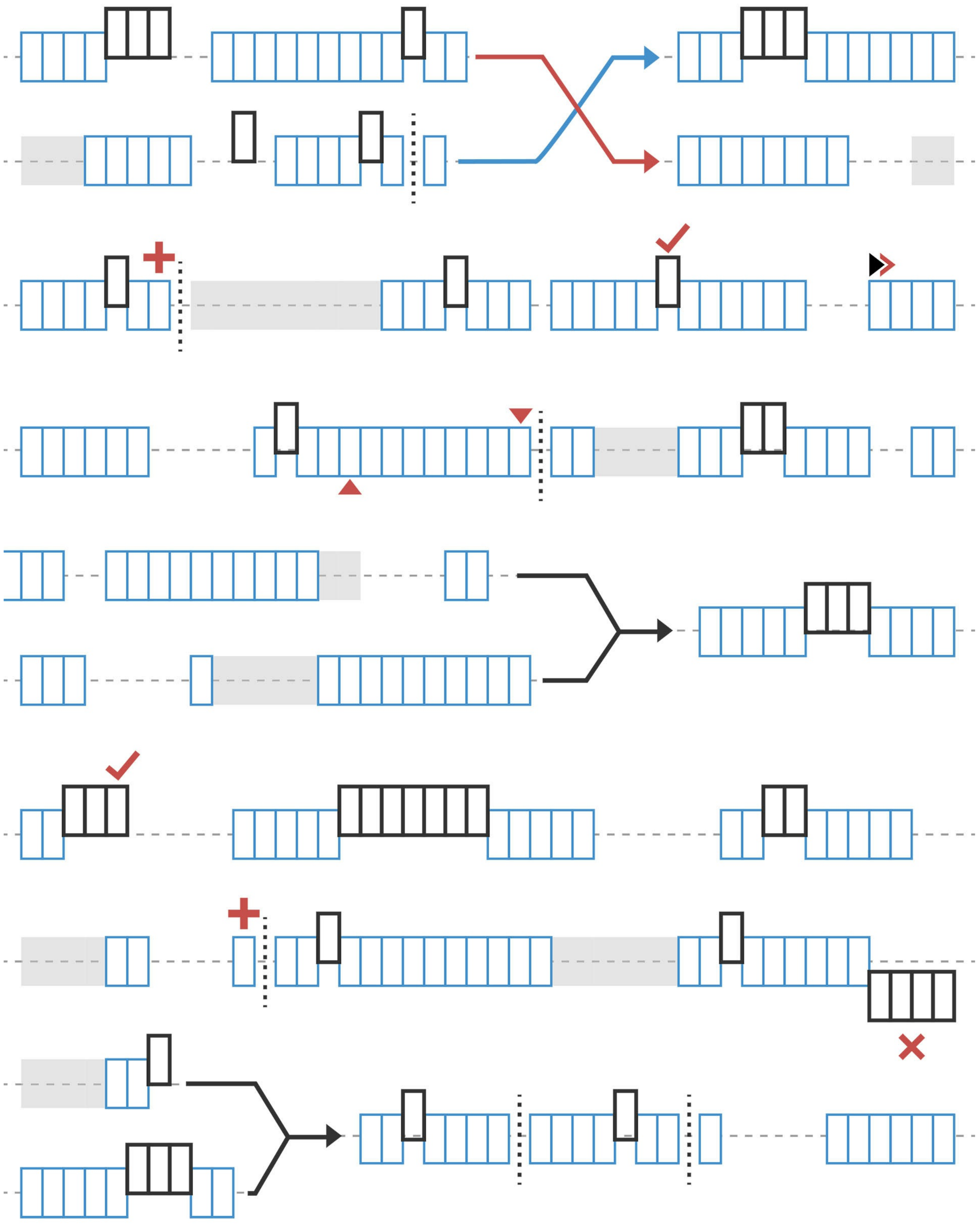
Importing Geometry

Importing Geometry

Designing with Lists

Designing with Lists

Lists are the way we organize data. On your computer's operating system, you have files and folders. In Dynamo, we can regard these as items and lists, respectively. Like your operating system, there are many ways to create, modify, and query data. In this chapter, we'll break down how lists are managed in Dynamo.



What's a List

What's a List?

A list is a collection of elements, or items. Take a bunch of bananas, for example. Each banana is an item within the list (or bunch). It's easier to pick up a bunch of bananas rather than each banana individually, and the same holds for grouping elements by parametric relationships in a data structure.

Photo by Augustus Binu.

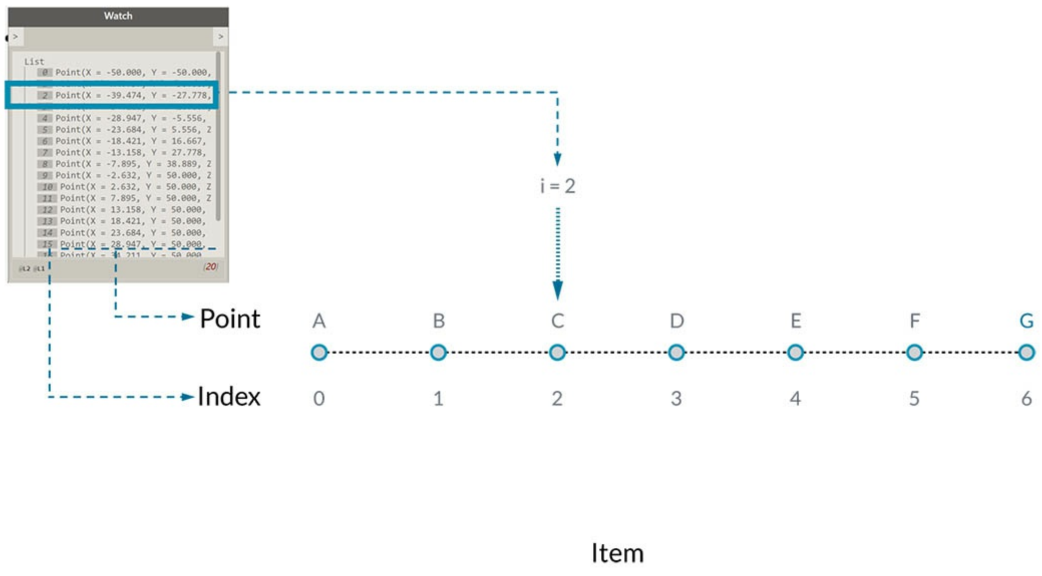
When we buy groceries, we put all of the purchased items into a bag. This bag is also a list. If we're making banana bread, we need 3 bunches of bananas (we're making a *lot* of banana bread). The bag represents a list of banana bunches and each bunch represents a list of bananas. The bag is a list of lists (two-dimensional) and the banana is a list (one-dimensional).

In Dynamo, list data is ordered, and the first item in each list has an index "0". Below, we'll discuss how lists are defined in Dynamo and how multiple lists relate to one another.

Zero-Based Indices

One thing that might seem odd at first is that the first index of a list is always 0; not 1. So, when we talk about the first item of a list, we actually mean the item that corresponds to index 0.

For example, if you were to count the number of fingers we have on our right hand, chances are that you would have counted from 1 to 5. However, if you were to put your fingers in a list, Dynamo would have given them indices from 0 to 4. While this may seem a little strange to programming beginners, the zero-based index is standard practice in most computation systems.



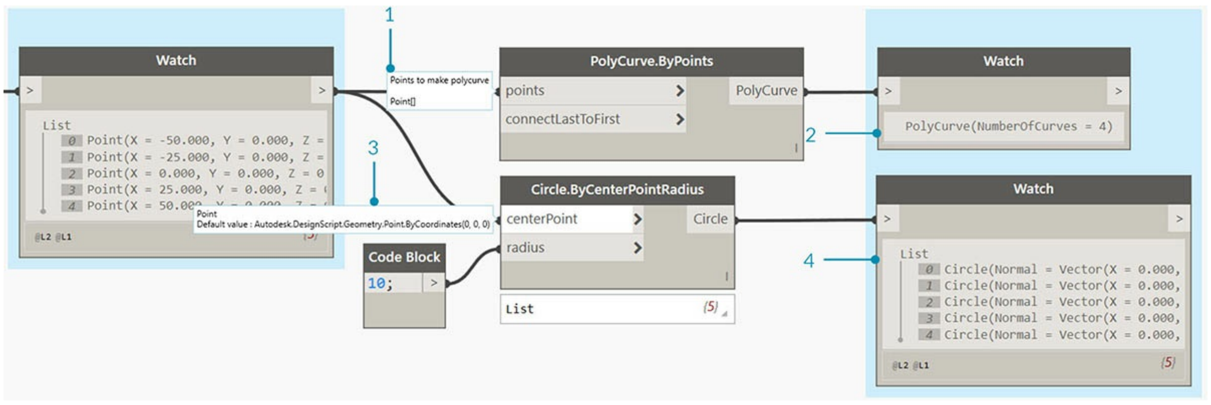
Note that we still have 5 items in the list; it's just that the list is using a zero-based counting system. And the items being stored in the list don't just have to be numbers. They can be any data type that Dynamo supports, such as points, curves, surfaces, families, etc.

Often times the easiest way to take a look at the type of data stored in a list is to connect a watch node to another node's output. By default, the watch node automatically shows all indices to the left side of the list and displays the data items on the right.

These indices are a crucial element when working with lists.

Inputs and Outputs

Pertaining to lists, inputs and outputs vary depending on the Dynamo node being used. As an example, let's use a list of 5 points and connect this output to two different Dynamo nodes: *PolyCurve.ByPoints* and *Circle.ByCenterPointRadius*:



1. The *points* input for *PolyCurve.ByPoints* is looking for *"Point[]"*. This represents a list of points.
2. The output for *PolyCurve.ByPoints* is a single PolyCurve created from a list of five points.
3. The *centerPoint* input for *Circle.ByCenterPointRadius* asks for *"Point"*.
4. The output for *Circle.ByCenterPointRadius* is a list of five circles, whose centers correspond to the original list of points.

The input data for *PolyCurve.ByPoints* and *Circle.ByCenterPointRadius* are the same, however the Polycurve node gives us one polycurve while the Circle node gives us 5 circles with centers at each point. Intuitively this makes sense: the polycurve is drawn as a curve connecting the 5 points, while the circles create a different circle at each point. So what's happening with the data?

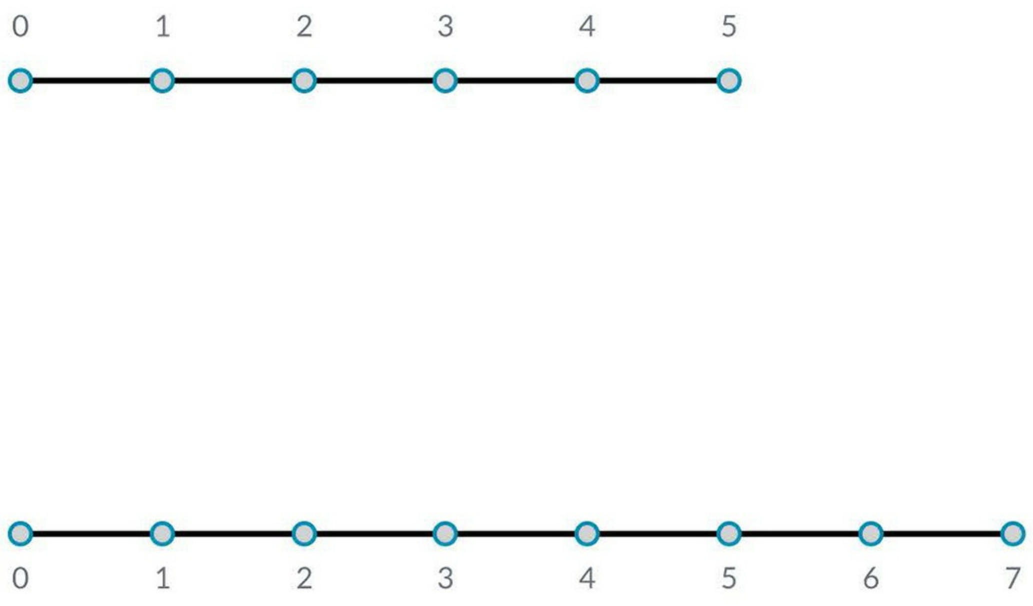
Hovering over the *points* input for *Polycurve.ByPoints*, we see that the input is looking for *"Point[]"*. Notice the brackets at the end. This represents a list of points, and to create a polycurve, the input needs to be a list for each polycurve. This node will therefore condense each list into one polycurve.

On the other hand, the *centerPoint* input for *Circle.ByCenterPointRadius* asks for *"Point"*. This node looks for one point, as an item, to define the center point of the circle. This is why we get five circles from the input data. Recognizing these difference with inputs in Dynamo helps to better understand how the nodes are operating when managing data.

Lacing

Data matching is a problem without a clean solution. It occurs when a node has access to differently sized inputs. Changing the data matching algorithm can lead to vastly different results.

Imagine a node which creates line segments between points (Line.ByStartPointEndPoint). It will have two input parameters which both supply point coordinates:

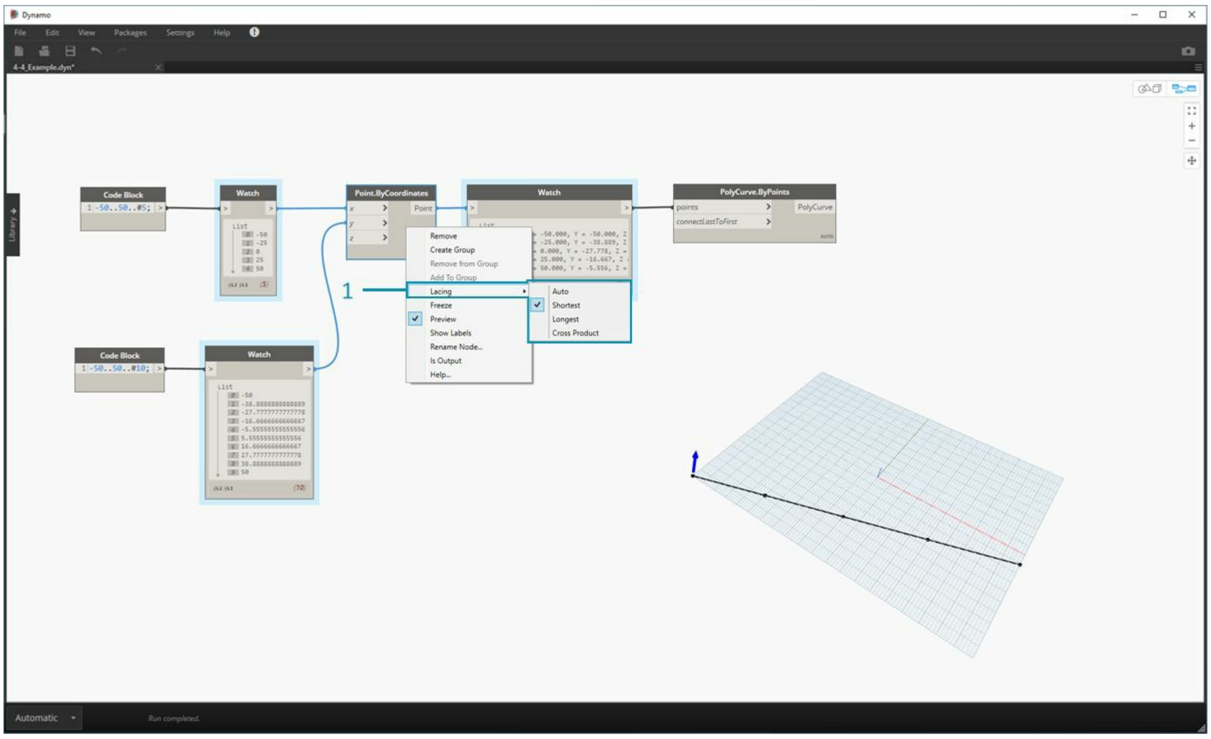


As you can see there are different ways in which we can draw lines between these sets of points. Lacing options are found by right-clicking the center of a node and choosing the "Lacing" menu.

Base File

> Download the example file that accompanies this exercise (Right click and "Save Link As..."): Lacing.dyn. A full list of example files can be found in the Appendix.

To demonstrate the lacing operations below, we'll use this base file to define shortest list, longest list, and cross product.



```
-50..50..#5;
```

```
List
  [0] -50
  [1] -25
  [2] 0
  [3] 25
  [4] 50
(5)
```

```
-50..50..#10;
```

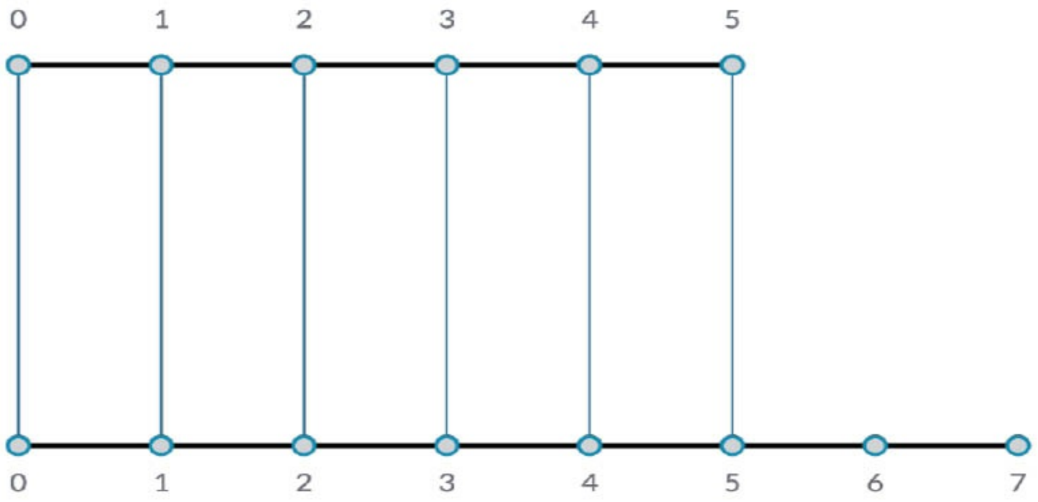
```
List
  [0] -50
  [1] -38.8888888888889
  [2] -27.7777777777778
  [3] -16.6666666666667
  [4] -5.55555555555556
  [5] 5.55555555555556
  [6] 16.6666666666667
  [7] 27.7777777777778
  [8] 38.8888888888889
  [9] 50
(10)
```

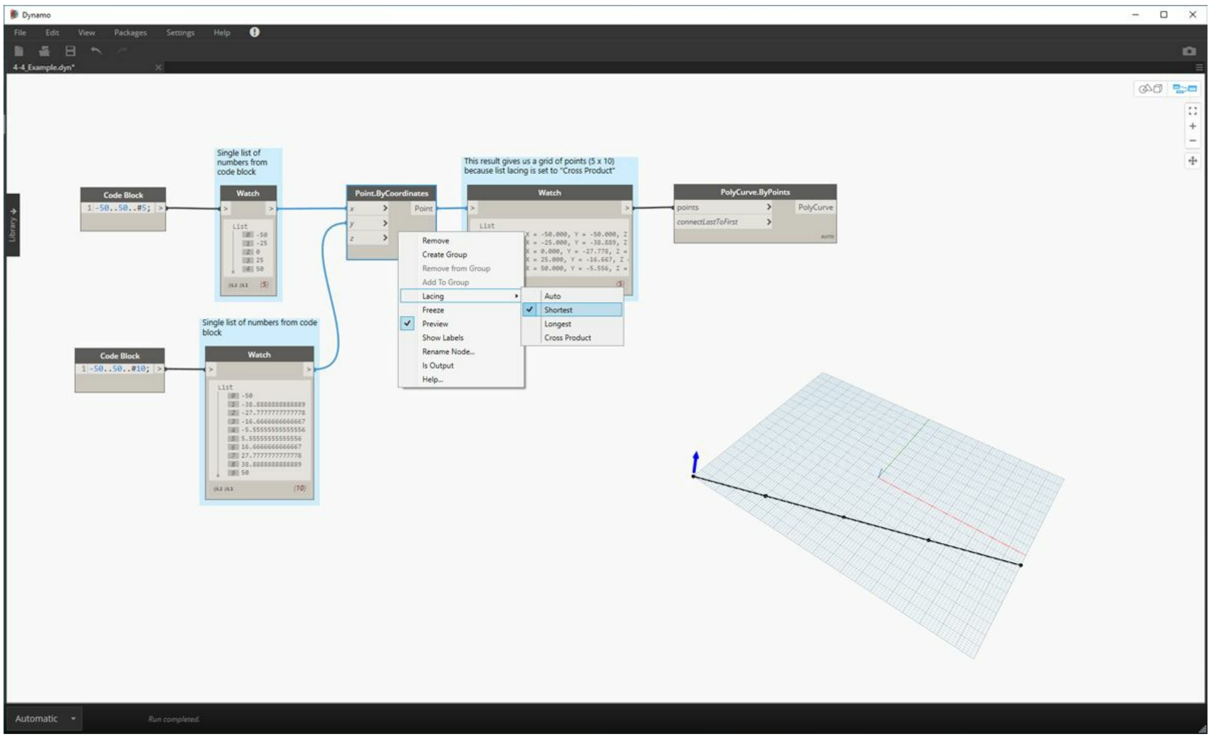
```
List
-50.000, Y = -50.000, Z
-25.000, Y = -38.889, Z
0.000, Y = -27.778, Z =
25.000, Y = -16.667, Z
50.000, Y = -5.556, Z =
```

1. We'll change the lacing on *Point.ByCoordinates*, but won't change anything else about the graph above.

Shortest List

The simplest way is to connect the inputs one-on-one until one of the streams runs dry. This is called the “Shortest List” algorithm. This is the default behavior for Dynamo nodes:





Single list of numbers from code block

This result gives us a grid of points (5 x 10) because list lacing is set to "Cross Product"

Single list of numbers from code block

```
1 -50..50..#5;
```

```
List
  [0] -50
  [1] -25
  [2] 0
  [3] 25
  [4] 50
(5)
```

```
1 -50..50..#10;
```

```
List
  [0] -50
  [1] -38.8888888888889
  [2] -27.7777777777778
  [3] -16.6666666666667
  [4] -5.55555555555556
  [5] 5.55555555555556
  [6] 16.6666666666667
  [7] 27.7777777777778
  [8] 38.8888888888889
  [9] 50
(10)
```

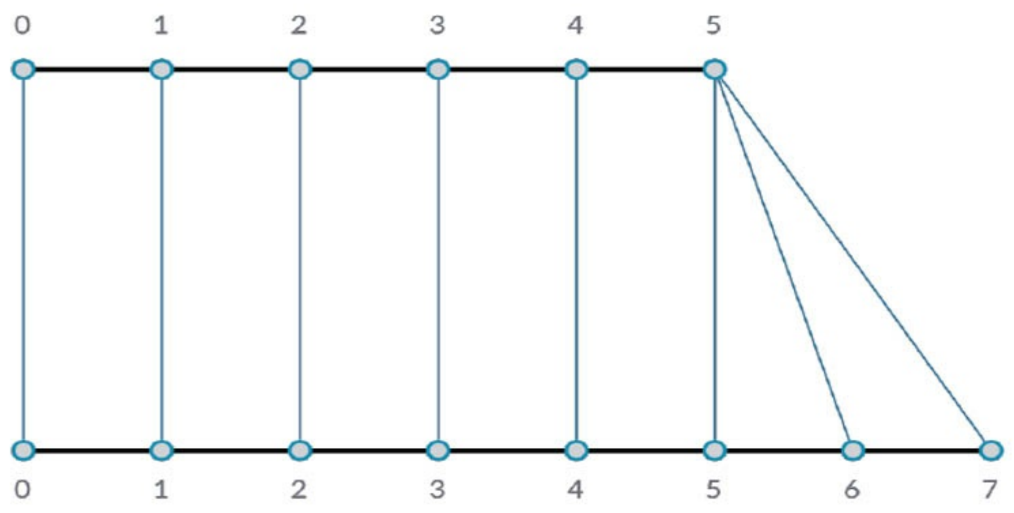
```
List
  X = -50.000, Y = -50.000, Z
  X = -25.000, Y = -38.889, Z
  X = 0.000, Y = -27.778, Z =
  X = 25.000, Y = -16.667, Z
  X = 50.000, Y = -5.556, Z =
(5)
```

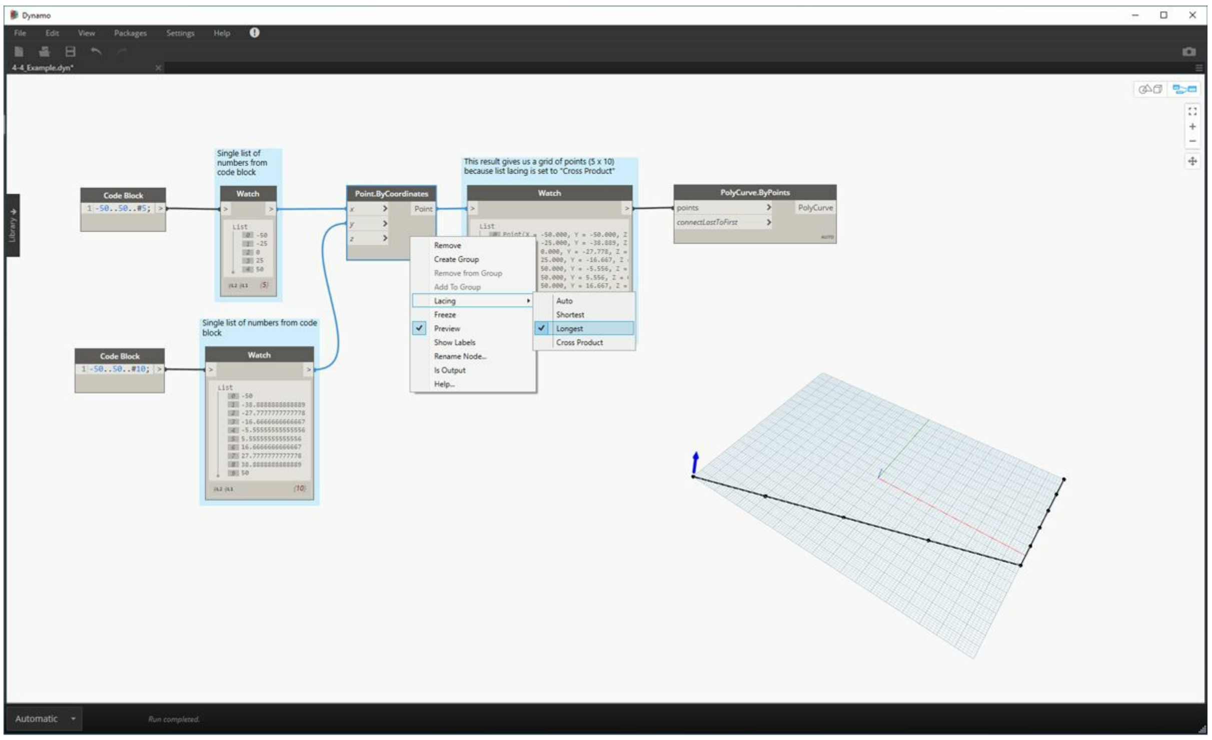
Lacing: Shortest ✓

By changing the lacing to *shortest list*, we get a basic diagonal line composed of five points. Five points is the length of the lesser list, so the shortest list lacing stops after it reaches the end of one list.

Longest List

The “Longest List” algorithm keeps connecting inputs, reusing elements, until all streams run dry:





Single list of numbers from code block

```
1 -50..50..#5;
```

```
List
  0 -50
  1 -25
  2 0
  3 25
  4 50
(5)
```

Single list of numbers from code block

```
1 -50..50..#10;
```

```
List
  0 -50
  1 -38.8888888888889
  2 -27.7777777777778
  3 -16.6666666666667
  4 -5.55555555555556
  5 5.55555555555556
  6 16.6666666666667
  7 27.7777777777778
  8 38.8888888888889
  9 50
(10)
```

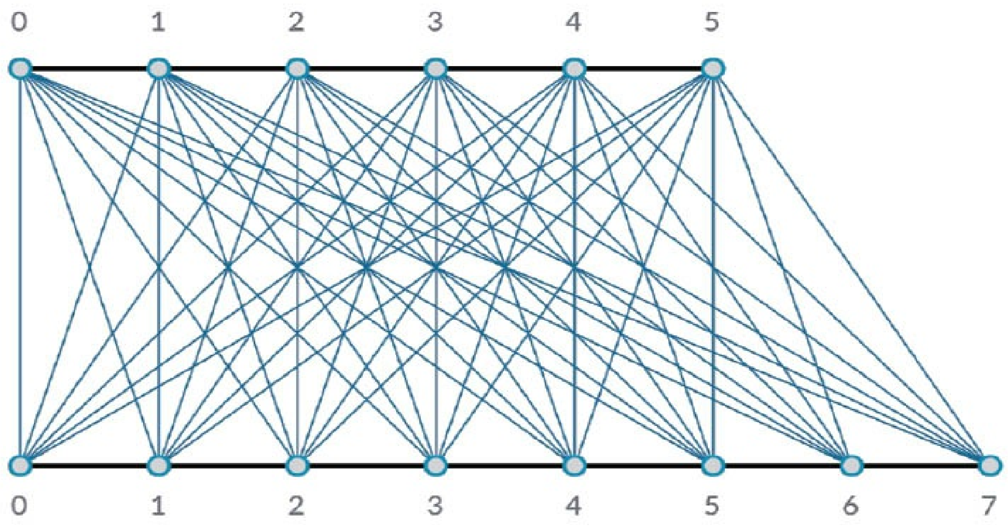
This result gives us a grid of points (5 x 10) because list lacing is set to "Cross Product"

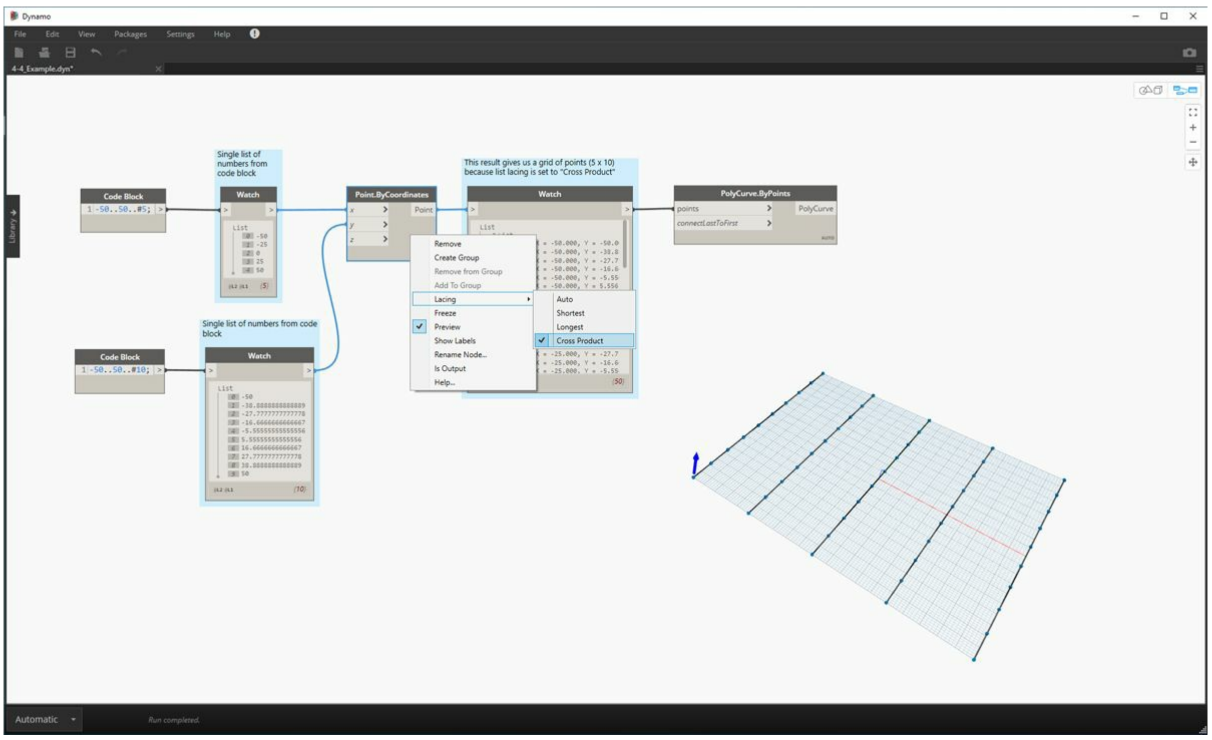
```
List
  0 Point(X = -50.000, Y = -50.000, Z
    -25.000, Y = -38.889, Z
    0.000, Y = -27.778, Z =
    25.000, Y = -16.667, Z
    50.000, Y = -5.556, Z =
    50.000, Y = 5.556, Z =
    50.000, Y = 16.667, Z =
```

By changing the lacing to *longest list*, we get a diagonal line which extends vertically. By the same method as the concept diagram, the last item in the list of 5 items will be repeated to reach the length of the longer list.

Cross Product

Finally, the “Cross Product” method makes all possible connections:





Single list of numbers from code block

This result gives us a grid of points (5 x 10) because list lacing is set to "Cross Product"

Single list of numbers from code block

```
1 -50..50..#5;
```

```
List
  0 -50
  1 -25
  2 0
  3 25
  4 50
(5)
```

```
1 -50..50..#10;
```

```
List
  0 -50
  1 -38.8888888888889
  2 -27.7777777777778
  3 -16.6666666666667
  4 -5.55555555555556
  5 5.55555555555556
  6 16.6666666666667
  7 27.7777777777778
  8 38.8888888888889
  9 50
(10)
```

```
List
X = -50.000, Y = -50.0
X = -50.000, Y = -38.8
X = -50.000, Y = -27.7
X = -50.000, Y = -16.6
X = -50.000, Y = -5.55
X = -50.000, Y = 5.556
X = -25.000, Y = -27.7
X = -25.000, Y = -16.6
X = -25.000, Y = -5.55
(50)
```

By changing the lacing to *Cross Product*, we get every combination between each list, giving us a 5x10 grid of points. This is an equivalent data structure to the cross product as shown in the concept diagram above, except our data is now a list of lists. By connecting a polycurve, we can see that each list is defined by its X-Value, giving us a row of vertical lines.

Working with Lists

Working with Lists

Now that we've established what a list is, let's talk about operations we can perform on it. Imagine a list as a deck of playing cards. A deck is the list and each playing card represents an item.

Photo by Christian Gidlöf

What **queries** can we make from the list? This accesses existing properties.

- Number of cards in the deck? 52.
- Number of suits? 4.
- Material? Paper.
- Length? 3.5" or 89mm.
- Width? 2.5" or 64mm.

What **actions** can we perform on the list? This changes the list based on a given operation.

- We can shuffle the deck.
- We can sort the deck by value.
- We can sort the deck by suit.
- We can split the deck.
- We can partition the deck by dealing out individual hands.
- We can select a specific card in the deck.

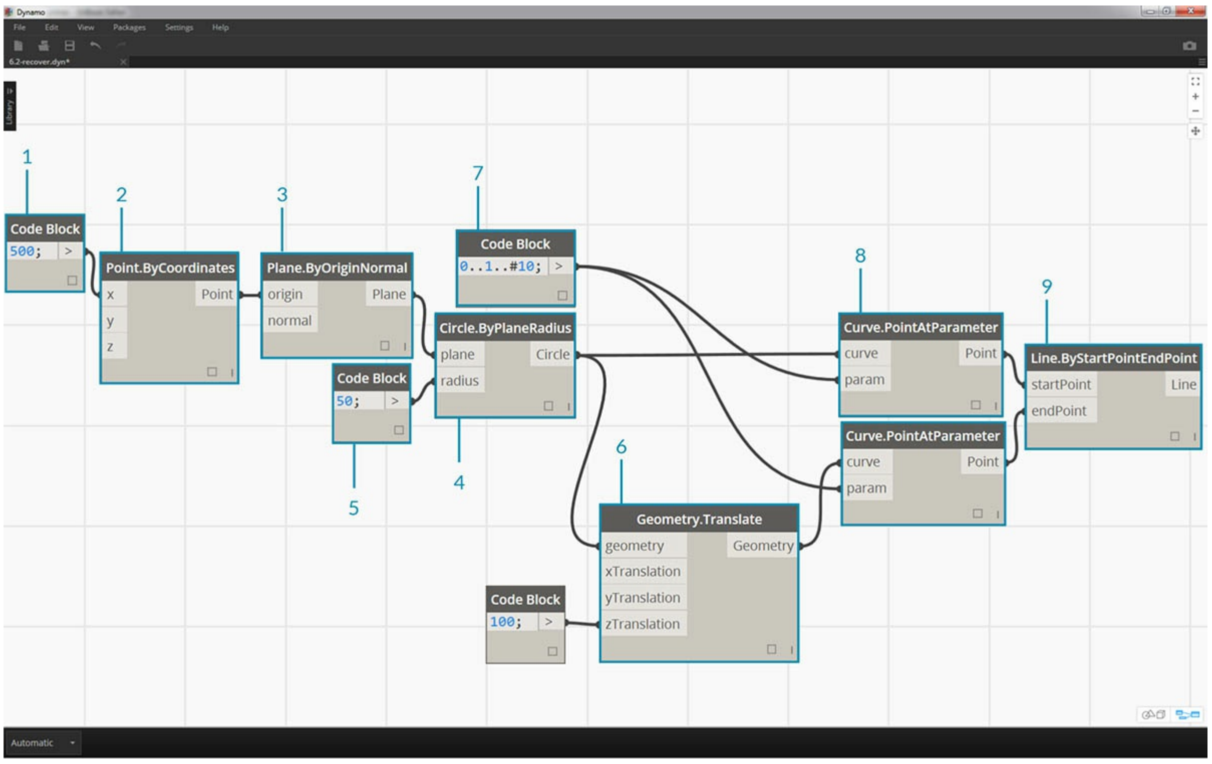
All of the operations listed above have analogous Dynamo nodes for working with lists of generic data. The lessons below will demonstrate some of the fundamental operations we can perform on lists.

List Operations

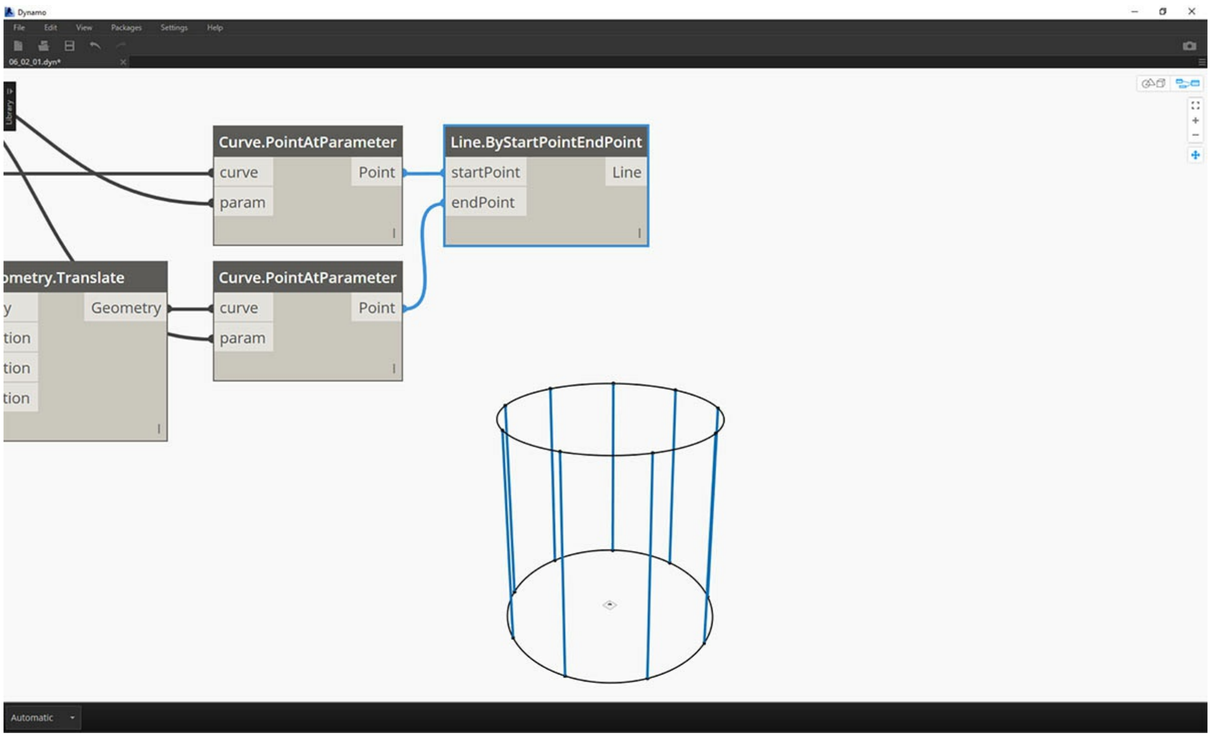
The image below is the base graph we will be using to demonstrate basic list operations. We'll explore how to manage data within a list and demonstrate the visual results.

Exercise - List Operations

Download the example file that accompanies this exercise (Right click and "Save Link As..."): List-Operations.dyn. A full list of example files can be found in the Appendix.

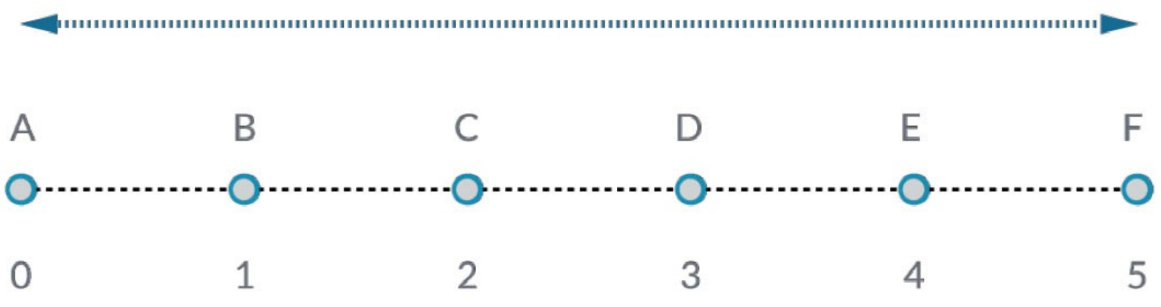


1. Begin with a *code block* with a value of `500;`
2. Plug into the *x* input of a *Point.ByCoordinates* node.
3. Plug the node from the previous step into the origin input of a *Plane.ByOriginNormal* node.
4. Using a *Circle.ByPlaneRadius* node, plug the node from the previous step into the plane input.
5. Using *code block*, designate a value of `50;` for the *radius*. This is the first circle we'll create.
6. With a *Geometry.Translate* node, move the circle up 100 units in the Z direction.
7. With a *code block* node, define a range of ten numbers between 0 and 1 with this line of code: `0..1..#10;`
8. Plug the code block from the previous step into the *param* input of two *Curve.PointAtParameter* nodes. Plug *Circle.ByPlaneRadius* into the curve input of the top node, and *Geometry.Translate* into the curve input of the node beneath it.
9. Using a *Line.ByStartPointEndPoint*, connect the two *Curve.PointAtParameter* nodes.



1. A *Watch3D* node shows the results of the *Line.ByStartPointEndPoint*. We are drawing lines between two circles to represent basic list operations and will use this base Dynamo graph to walk through the list actions below.

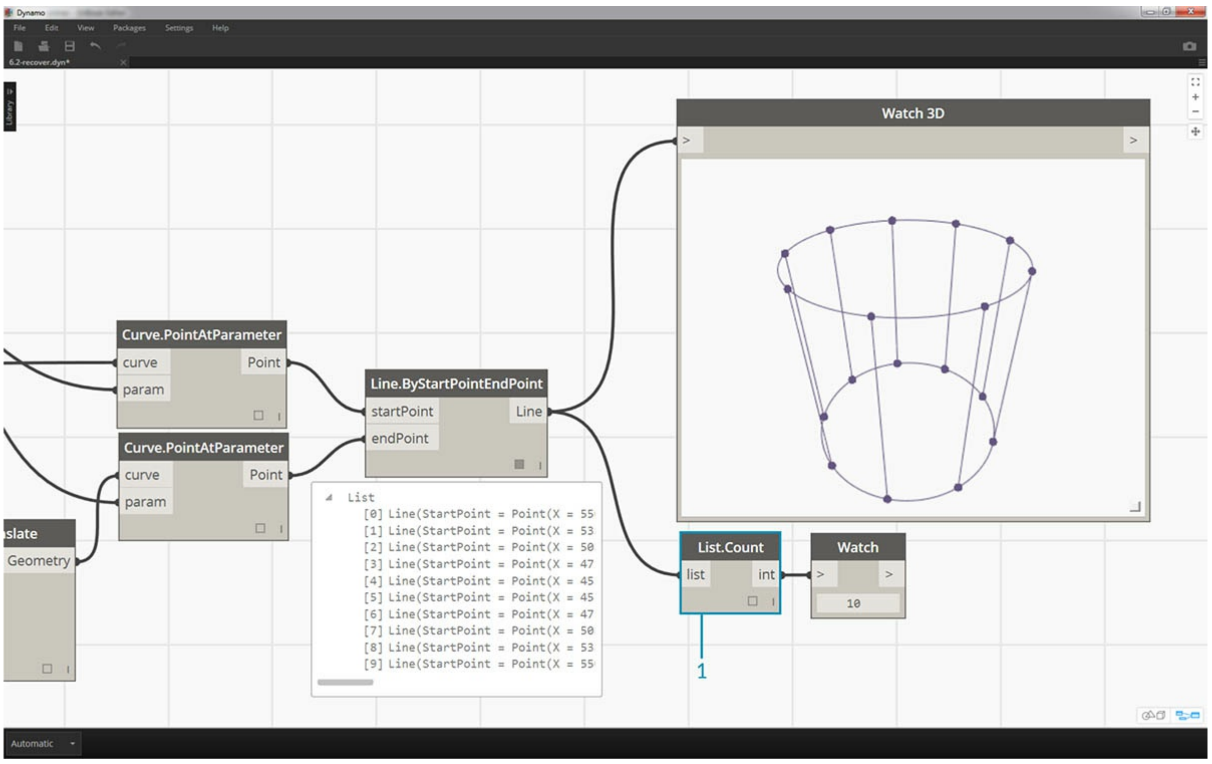
List.Count



The *List.Count* node is straightforward: it counts the number of values in a list and returns that number. This node gets more nuanced as we work with lists of lists, but we'll demonstrate that in the coming sections.

Exercise - List.Count

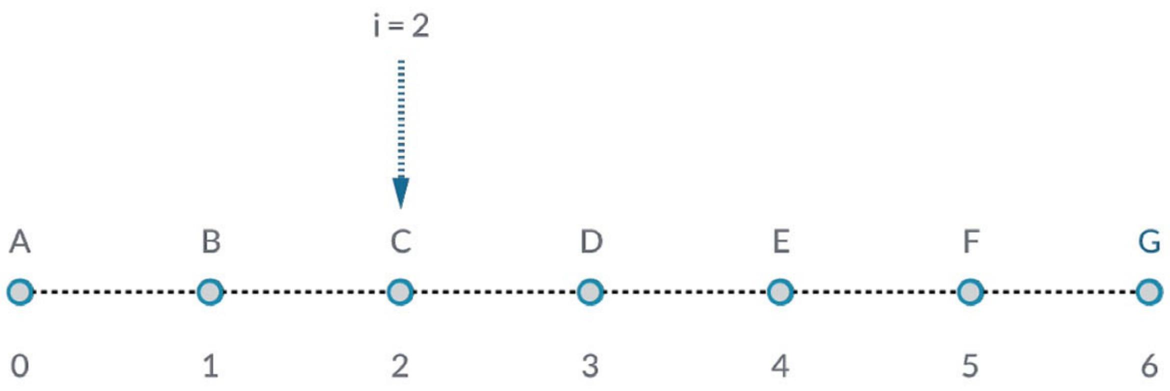
Download the example file that accompanies this exercise (Right click and "Save Link As..."): [List-Count.dyn](). A full list of example files can be found in the Appendix.



```
List
  [0] Line(StartPoint = Point(X = 55
  [1] Line(StartPoint = Point(X = 53
  [2] Line(StartPoint = Point(X = 50
  [3] Line(StartPoint = Point(X = 47
  [4] Line(StartPoint = Point(X = 45
  [5] Line(StartPoint = Point(X = 45
  [6] Line(StartPoint = Point(X = 47
  [7] Line(StartPoint = Point(X = 50
  [8] Line(StartPoint = Point(X = 53
  [9] Line(StartPoint = Point(X = 55
```

1. The *List.Count* node returns the number of lines in the *Line.ByStartPointEndPoint* node. The value is 10 in this case, which agrees with the number of points created from the original *code block* node.

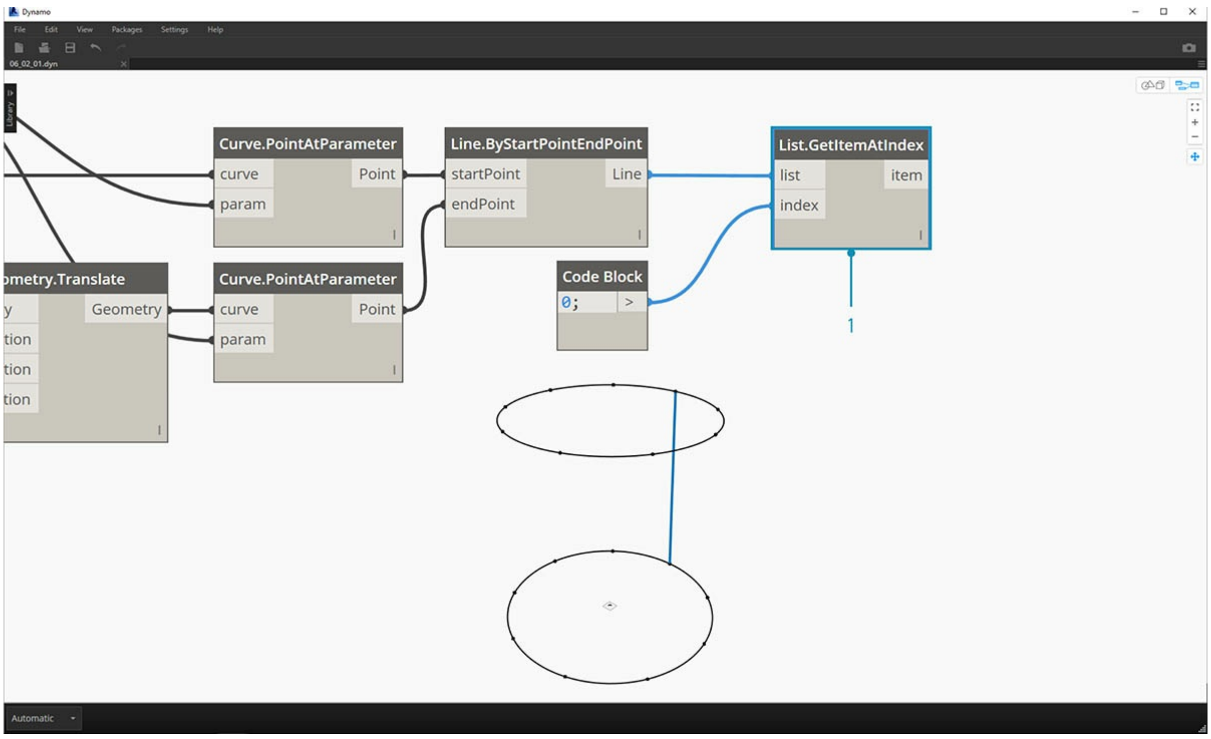
List.GetItemAtIndex



List.GetItemAtIndex is a fundamental way to query an item in the list. In the image above, we are using an index of *"2"* to query the point labeled *"C"*.

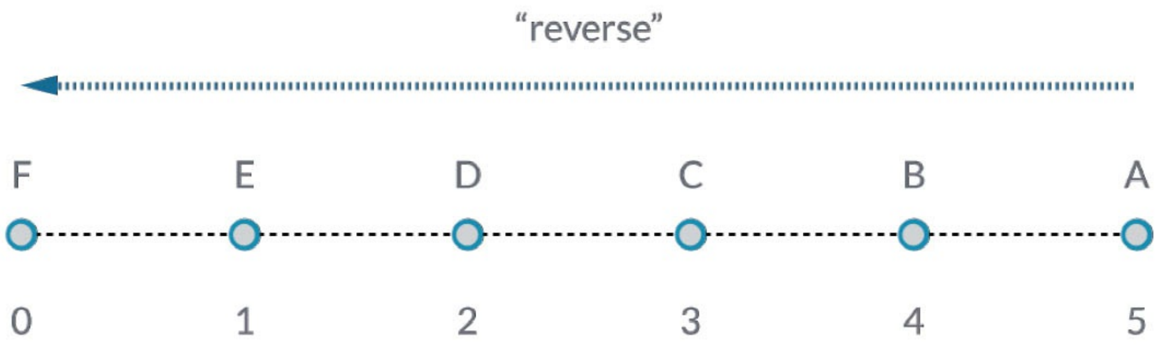
Exercise - List.GetItemAtIndex

Download the example file that accompanies this exercise (Right click and "Save Link As..."): List-GetItemAtIndex.dyn. A full list of example files can be found in the Appendix.



1. Using the *List.GetItemAtIndex* node, we are selecting index *"0"*, or the first item in the list of lines.
2. The *Watch3D* node reveals that we've selected one line. Note: to get the image above, be sure to disable the preview of *Line.ByStartPointEndPoint*.

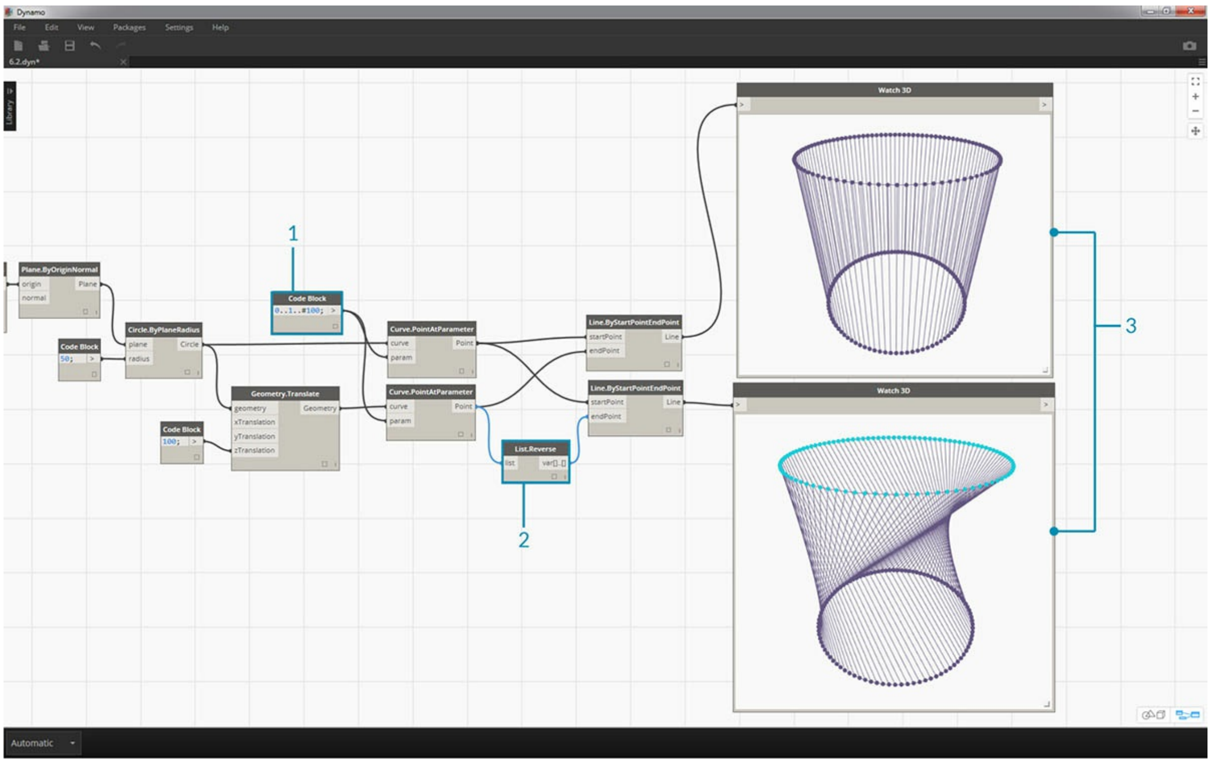
List.Reverse



List.Reverse reverses the order of all of the items in a list.

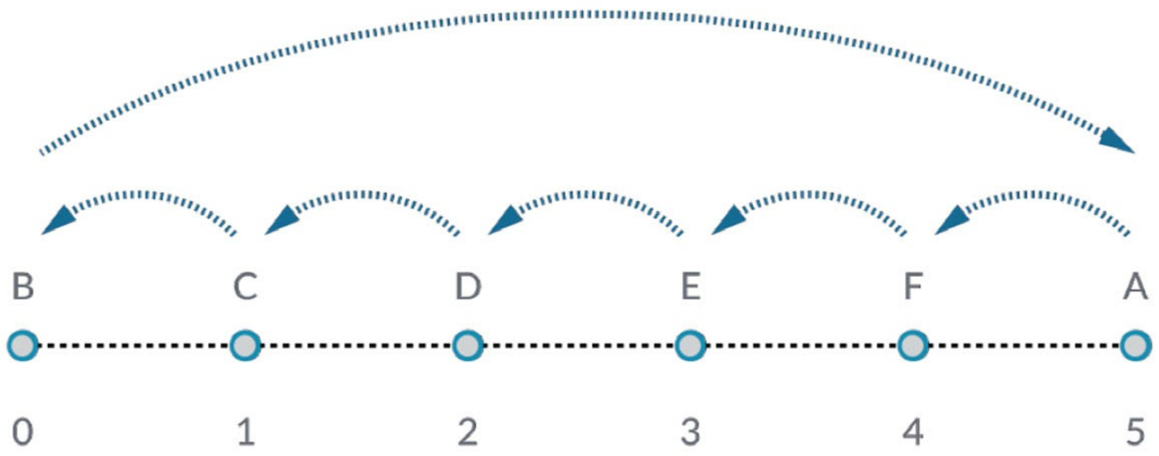
Exercise - List.Reverse

Download the example file that accompanies this exercise (Right click and "Save Link As..."): List-Reverse.dyn. A full list of example files can be found in the Appendix.



1. To properly visualize the reversed list of lines, create more lines by changing the code block to `0..1..#100;`
2. Insert a *List.Reverse* node in between *Curve.PointAtParameter* and *Line.ByStartPointEndPoint* for one of the list of points.
3. The *Watch3D* nodes show two different results. The first one shows the result without a reversed list. The lines connect vertically to neighboring points. The reversed list, however, will connect all of the points to the opposing order in the other list.

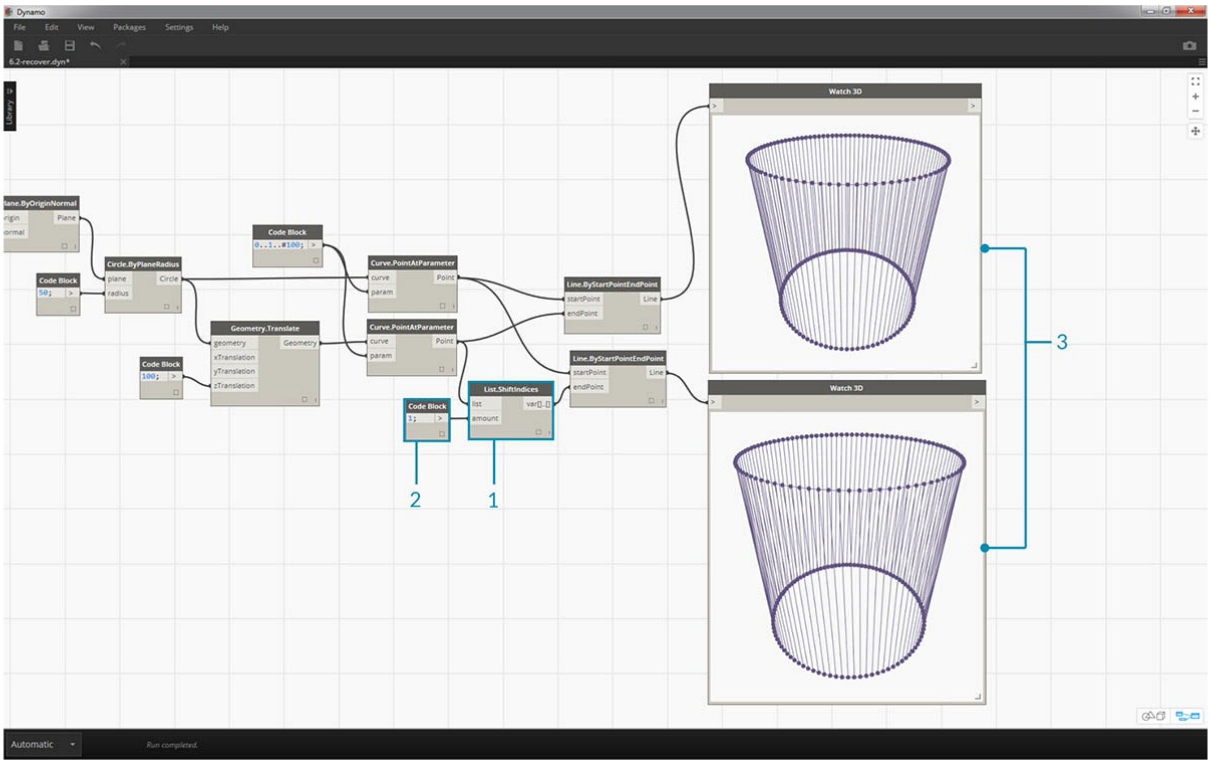
List.ShiftIndices



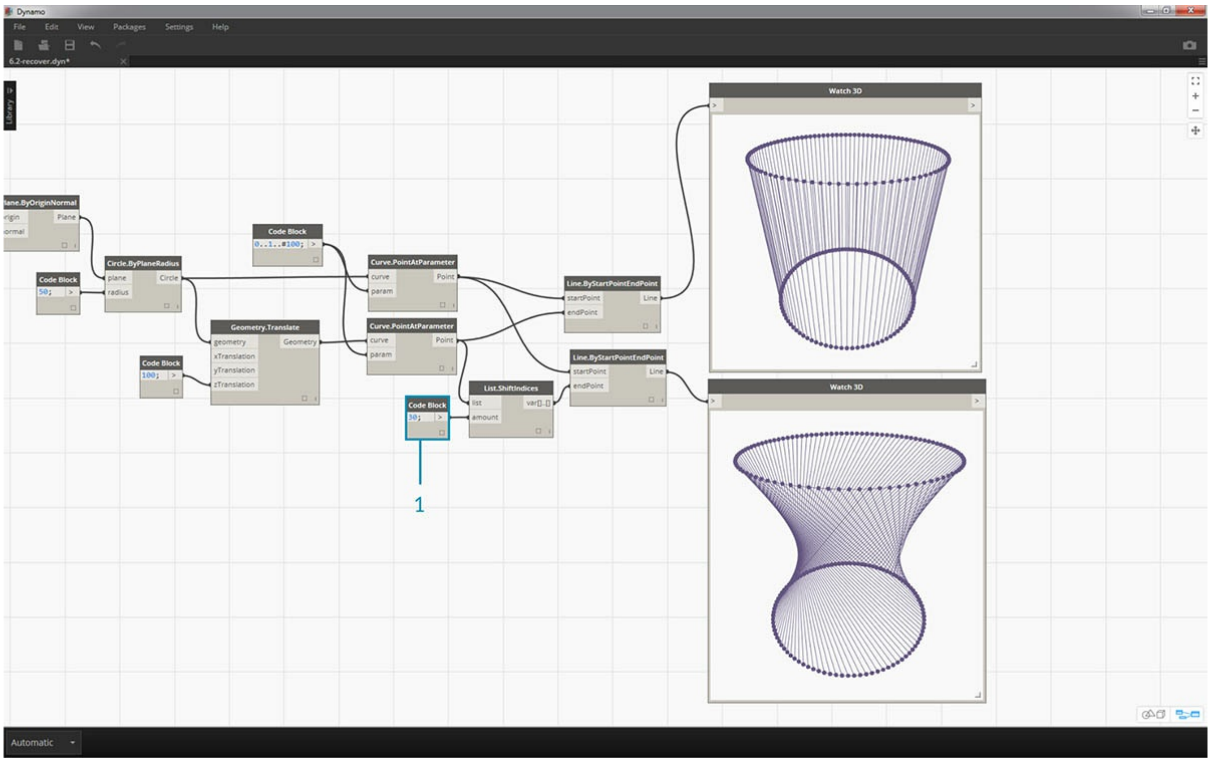
List.ShiftIndices is a good tool for creating twists or helical patterns, or any other similar data manipulation. This node shifts the items in a list a given number of indices.

Exercise - List.ShiftIndices

Download the example file that accompanies this exercise (Right click and "Save Link As..."): List-ShiftIndices.dyn. A full list of example files can be found in the Appendix.

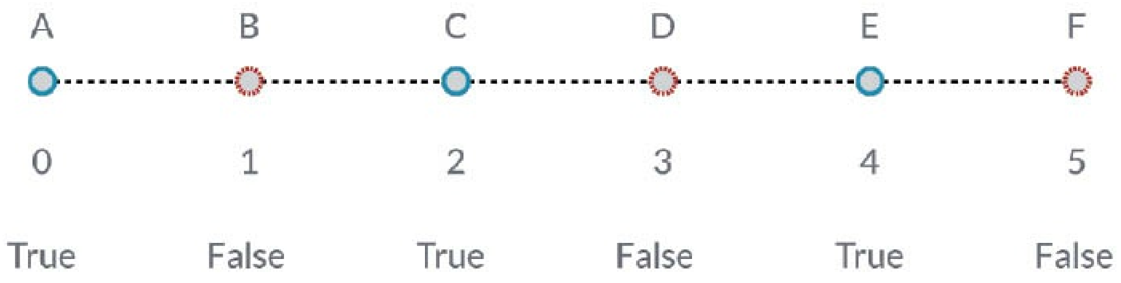


1. In the same process as the reverse list, insert a *List.ShiftIndices* into the *Curve.PointAtParameter* and *Line.ByStartPointEndPoint*.
2. Using a *code block*, designated a value of *"1"* to shift the list one index.
3. Notice that the change is subtle, but all of the lines in the lower *Watch3D* node have shifted one index when connecting to the other set of points.



1. By changing to *code block* to a larger value, *"30"* for example, we notice a significant difference in the diagonal lines. The shift is working like a camera's iris in this case, creating a twist in the original cylindrical form.

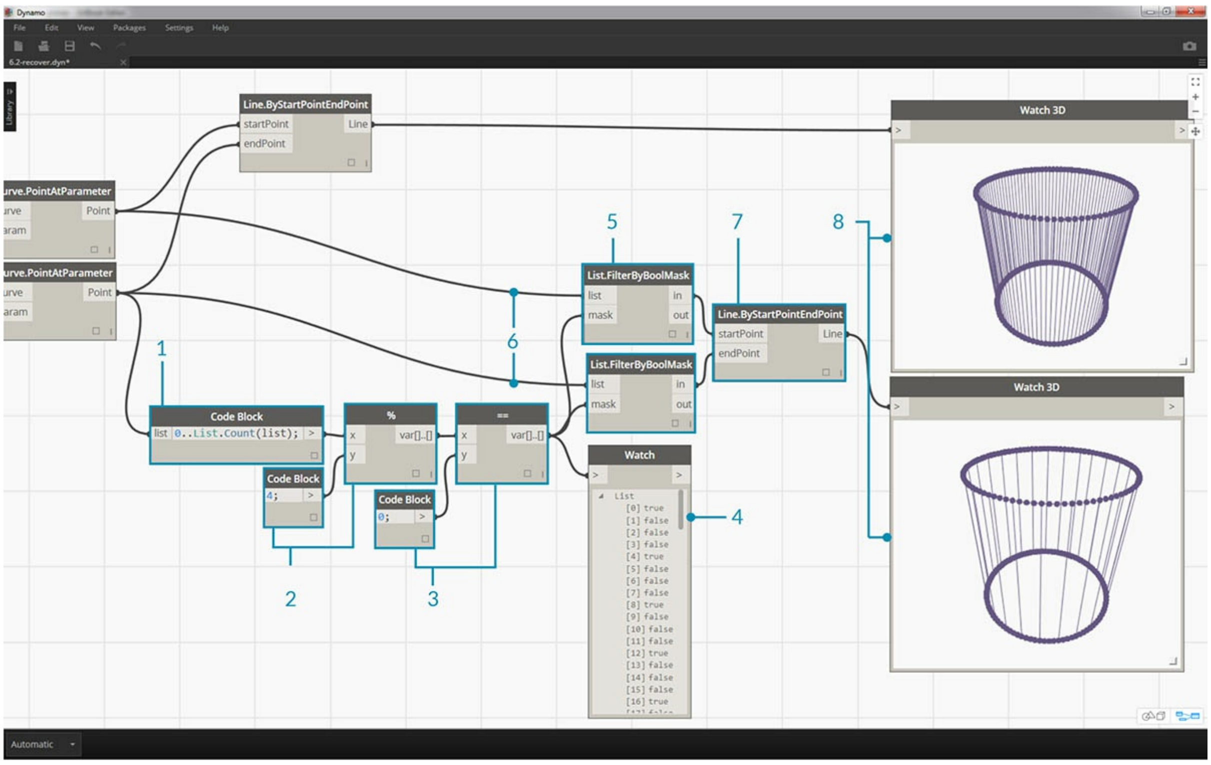
List.FilterByBooleanMask



List.FilterByBooleanMask will remove certain items based on a list of booleans, or values reading "true" or "false".

Exercise - List.FilterByBooleanMask

Download the example file that accompanies this exercise (Right click and "Save Link As..."): List-FilterByBooleanMask.dyn. A full list of example files can be found in the Appendix.



In order to create a list of values reading "true" or "false", we need to a little more work...

1. Using a *code block*, define an expression with the syntax: `0..List.Count(list);`. Connect the *Curve.PointAtParameter* node to the *list* input. We'll walk through this setup more in the code block chapter, but the line of code in this case is giving us a list representing each index of the *Curve.PointAtParameter* node.
2. Using a *"%"* (modulus) node, connect the output of the *code block* into the *x* input, and a value of *4* into the *y* input. This will give us the remainder when dividing the list of indices by 4. Modulus is a really helpful node for pattern creation. All values will read as the possible remainders of 4: 0, 1, 2, 3.
3. From the *modulus* node, we know that a value of 0 means that the index is divisible by 4 (0,4,8,etc...). By using a *"=="* node, we can test for the divisibility by testing it against a value of *"0"*.
4. The *Watch* node reveals just this: we have a true/false pattern which reads: *true,false,false,false....*
5. Using this true/false pattern, connect to the mask input of two *List.FilterByBooleanMask* nodes.
6. Connect the *Curve.PointAtParameter* node into each list input for the *List.FilterByBooleanMask*.
7. The output of *Filter.ByBooleanMask* reads *"in"* and *"out"*. *"In"* represents values which had a mask value of *"true"* while *"out"* represents values which had a value of *"false"*. By plugging the *"in"* outputs into the *startPoint* and *endPoint* inputs of a *Line.ByStartPointEndPoint* node, we've created a filtered list of lines.
8. The *Watch3D* node reveals that we have fewer lines than points. We've selected only 25% of the nodes by filtering only the true values!

Lists of Lists

Lists of Lists

Let's add one more tier to the hierarchy. If we take the deck of cards from the original example and create a box which contains multiple decks, the box now represents a list of decks, and each deck represents a list of cards. This is a list of lists. For the analogy in this section, the red box below contains a list of coin rolls, and each roll contains a list of pennies.

Photo by Dori.

What **queries** can we make from the list of lists? This accesses existing properties.

- Number of coin types? 2.
- Coin type values? $0.01 and $0.25.
- Material of quarters? 75% copper and 25% nickel.
- Material of pennies? 97.5% zinc and 2.5% copper.

What **actions** can we perform on the list of lists? This changes the list of lists based on a given operation.

- Select a specific stack of quarters or pennies.
- Select a specific quarter or penny.
- Rearrange the stacks of quarters and pennies.
- Shuffle the stacks together.

Again, Dynamo has an analagous node for each one of the operations above. Since we're working with abstract data and not physical objects, we need a set of rules to govern how we move up and down the data hierarchy.

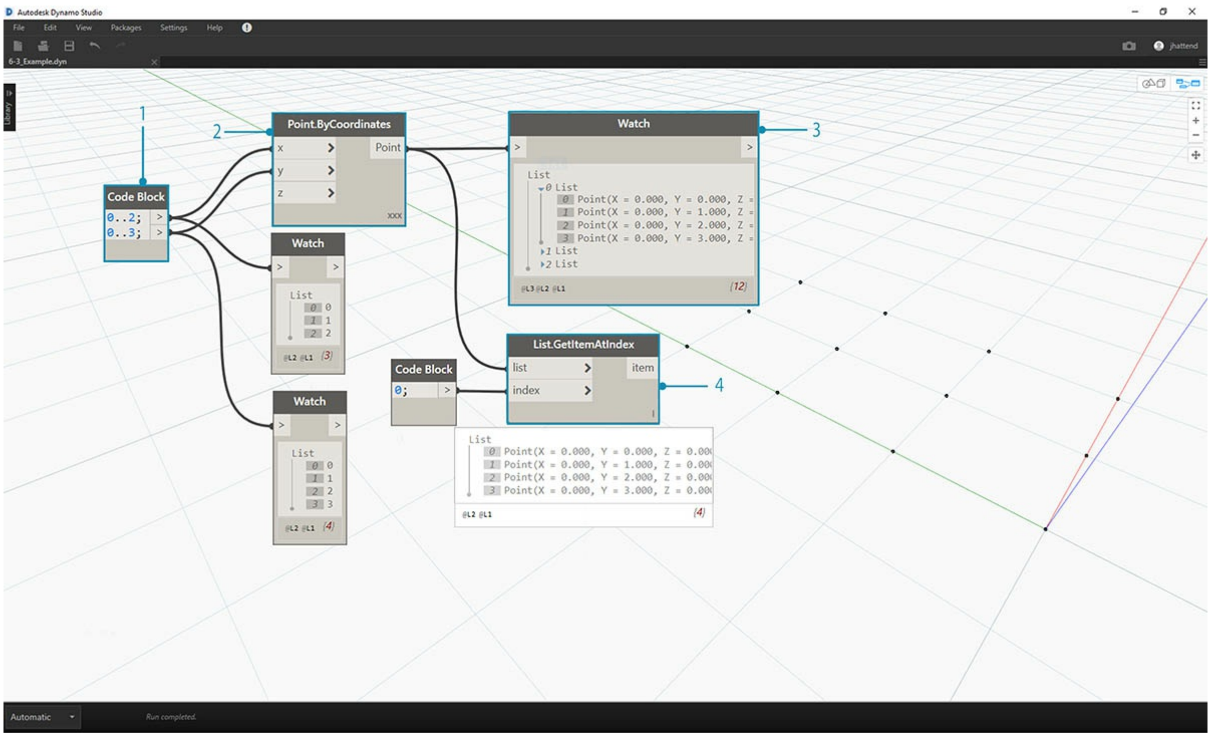
When dealing with lists of lists, the data is layered and complex, but this provides an opportunity to do some awesome parametric operations. Let's break down the fundamentals and discuss a few more operations in the lessons below.

Top-Down Hierarchy

The fundamental concept to learn from this section: **Dynamo treats lists as objects in and of themselves**. This top-down hierarchy is developed with object-oriented programming in mind. Rather than selecting sub-elements with a command like List.GetItemAtIndex, Dynamo will select that index of the main list in the data structure. And that item can be another list. Let's break it down with an example image:

Exercise - Top-Down Hierarchy

Download the example file that accompanies this exercise (Right click and "Save Link As..."): Top-Down-Hierarchy.dyn. A full list of example files can be found in the Appendix.



```
0..2;
0..3;
```

```
List
 0 List
   0 Point(X = 0.000, Y = 0.000, Z =
   1 Point(X = 0.000, Y = 1.000, Z =
   2 Point(X = 0.000, Y = 2.000, Z =
   3 Point(X = 0.000, Y = 3.000, Z =
 1 List
 2 List
@L3 @L2 @L1                        {12}
```

```
List
 0 0
 1 1
 2 2
@L2 @L1 {3}
```

```
List
 0 0
 1 1
 2 2
 3 3
@L2 @L1 {4}
```

```
0;
```

```
List
 0 Point(X = 0.000, Y = 0.000, Z = 0.00
 1 Point(X = 0.000, Y = 1.000, Z = 0.00
 2 Point(X = 0.000, Y = 2.000, Z = 0.00
 3 Point(X = 0.000, Y = 3.000, Z = 0.00
@L2 @L1                           {4}
```

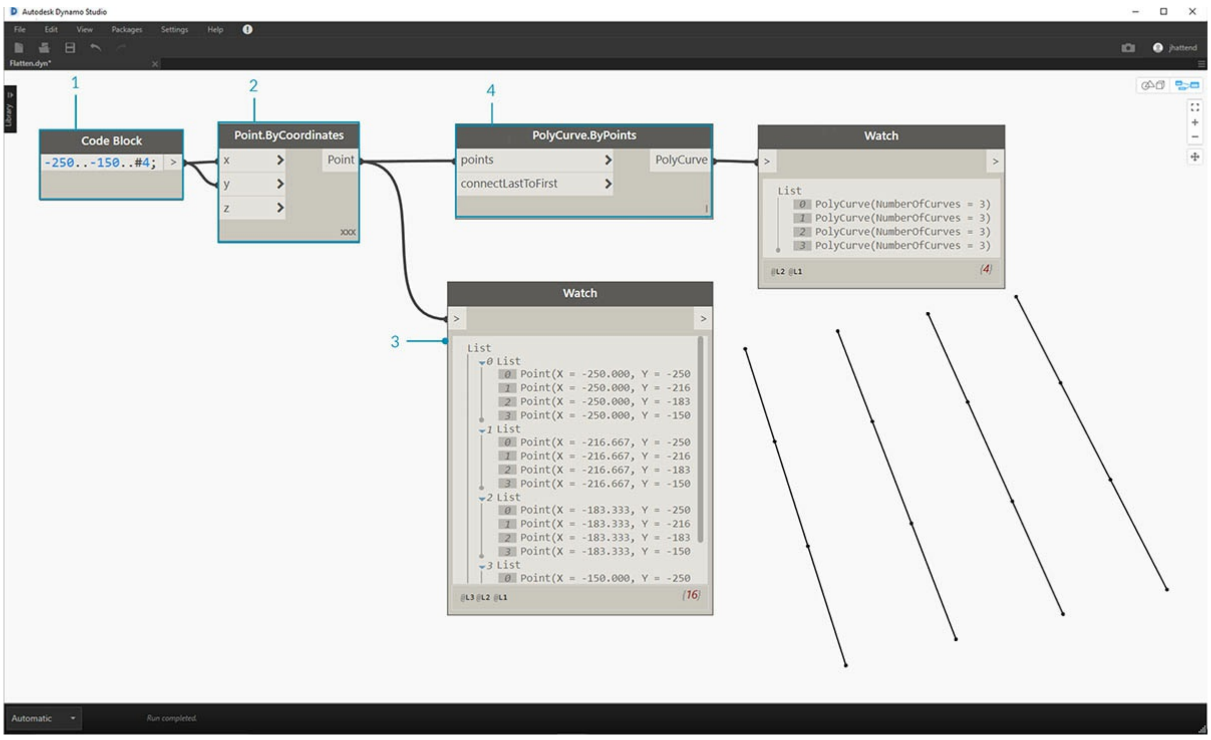
1. With *code block*, we've defined two ranges:``` 0..2; 0..3; ```
2. These ranges are connected to a *Point.ByCoordinates* node with lacing set to *"Cross Product"*. This creates a grid of points, and also returns a list of lists as an output.
3. Notice that the *Watch* node gives 3 lists with 4 items in each list.
4. When using *List.GetItemAtIndex*, with an index of 0, Dynamo selects the first list and all of its contents. Other programs may select the first item of every list in the data structure, but Dynamo employs a top-down hierarchy when dealing with data.

Flatten and List.Flatten

Flatten removes all tiers of data from a data structure. This is helpful when the data hierarchies are not necessary for your operation, but it can be risky because it removes information. The example below shows the result of flattening a list of data.

Exercise - Flatten

> Download the example file that accompanies this exercise (Right click and "Save Link As..."): Flatten.dyn. A full list of example files can be found in the Appendix.

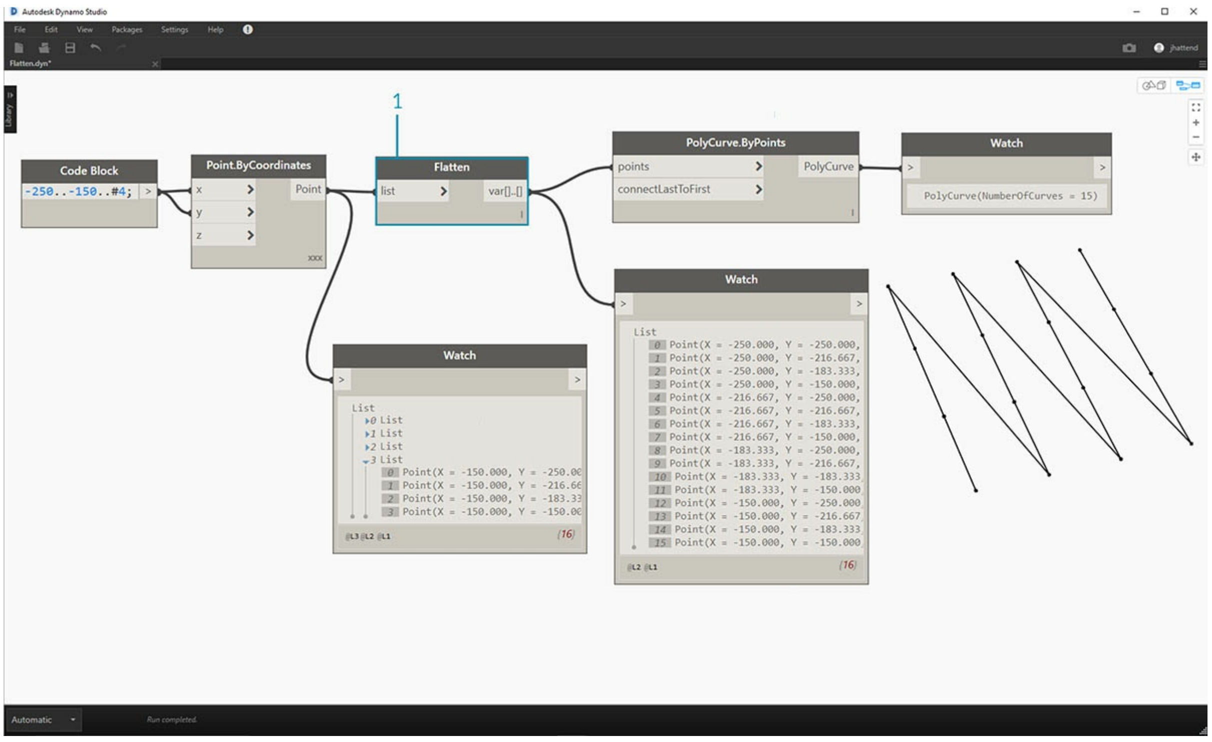


```
-250..-150..#4;
```

```
List
 0 PolyCurve(NumberOfCurves = 3)
 1 PolyCurve(NumberOfCurves = 3)
 2 PolyCurve(NumberOfCurves = 3)
 3 PolyCurve(NumberOfCurves = 3)
@L2 @L1                        {4}
```

```
List
 0 List
   0 Point(X = -250.000, Y = -250
   1 Point(X = -250.000, Y = -216
   2 Point(X = -250.000, Y = -183
   3 Point(X = -250.000, Y = -150
 1 List
   0 Point(X = -216.667, Y = -250
   1 Point(X = -216.667, Y = -216
   2 Point(X = -216.667, Y = -183
   3 Point(X = -216.667, Y = -150
 2 List
   0 Point(X = -183.333, Y = -250
   1 Point(X = -183.333, Y = -216
   2 Point(X = -183.333, Y = -183
   3 Point(X = -183.333, Y = -150
 3 List
   0 Point(X = -150.000, Y = -250
@L3 @L2 @L1                   {16}
```

1. Insert one line of code to define a range in *code block*:``` -250..-150..#4; ```
2. Plugging the *code block* into the *x* and *y* input of a *Point.ByCoordinates* node, we set the lacing to *"Cross Product"* to get a grid of points.
3. The *Watch* node shows that we have a list of lists.
4. A *PolyCurve.ByPoints* node will reference each list and create a respective polycurve. Notice in the Dynamo preview that we have four polycurve representing each row in the grid.



```
-250..-150..#4;
```

```
PolyCurve(NumberOfCurves = 15)
```

```
List
 0 List
 1 List
 2 List
 3 List
   0 Point(X = -150.000, Y = -250.00
   1 Point(X = -150.000, Y = -216.66
   2 Point(X = -150.000, Y = -183.33
   3 Point(X = -150.000, Y = -150.00
@L3 @L2 @L1                      {16}
```

```
List
  0 Point(X = -250.000, Y = -250.000,
  1 Point(X = -250.000, Y = -216.667,
  2 Point(X = -250.000, Y = -183.333,
  3 Point(X = -250.000, Y = -150.000,
  4 Point(X = -216.667, Y = -250.000,
  5 Point(X = -216.667, Y = -216.667,
  6 Point(X = -216.667, Y = -183.333,
  7 Point(X = -216.667, Y = -150.000,
  8 Point(X = -183.333, Y = -250.000,
  9 Point(X = -183.333, Y = -216.667,
 10 Point(X = -183.333, Y = -183.333
 11 Point(X = -183.333, Y = -150.000
 12 Point(X = -150.000, Y = -250.000
 13 Point(X = -150.000, Y = -216.667
 14 Point(X = -150.000, Y = -183.333
 15 Point(X = -150.000, Y = -150.000
@L2 @L1                          {16}
```

1. By inserting a *flatten* before the polycurve node, we've created one single list for all of the points. The polycurve node references a list to create one curve, and since all of the points are on one list, we get one zig-zag polycurve which runs throughout the entire list of points.

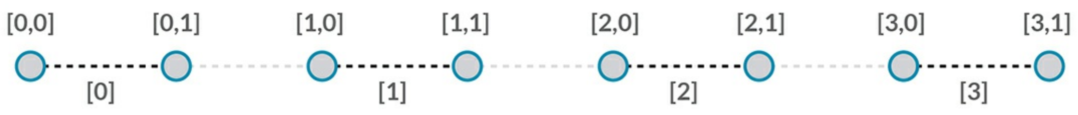
There are also options for flattening isolated tiers of data. Using the List.Flatten node, you can define a set number of data tiers to flatten from the top of the hierarchy. This is a really helpful tool if you're struggling with complex data structures which are not necessarily relevant to your workflow. And another option is to use the flatten node as a function in List.Map. We'll discuss List.Map more below.

Chop

When parametric modeling, there are also times where you'll want to add more data structure to an existing list. There are many nodes available for this as well, and chop is the most basic version. With chop, we can partition a list into sublists with a set number of items.

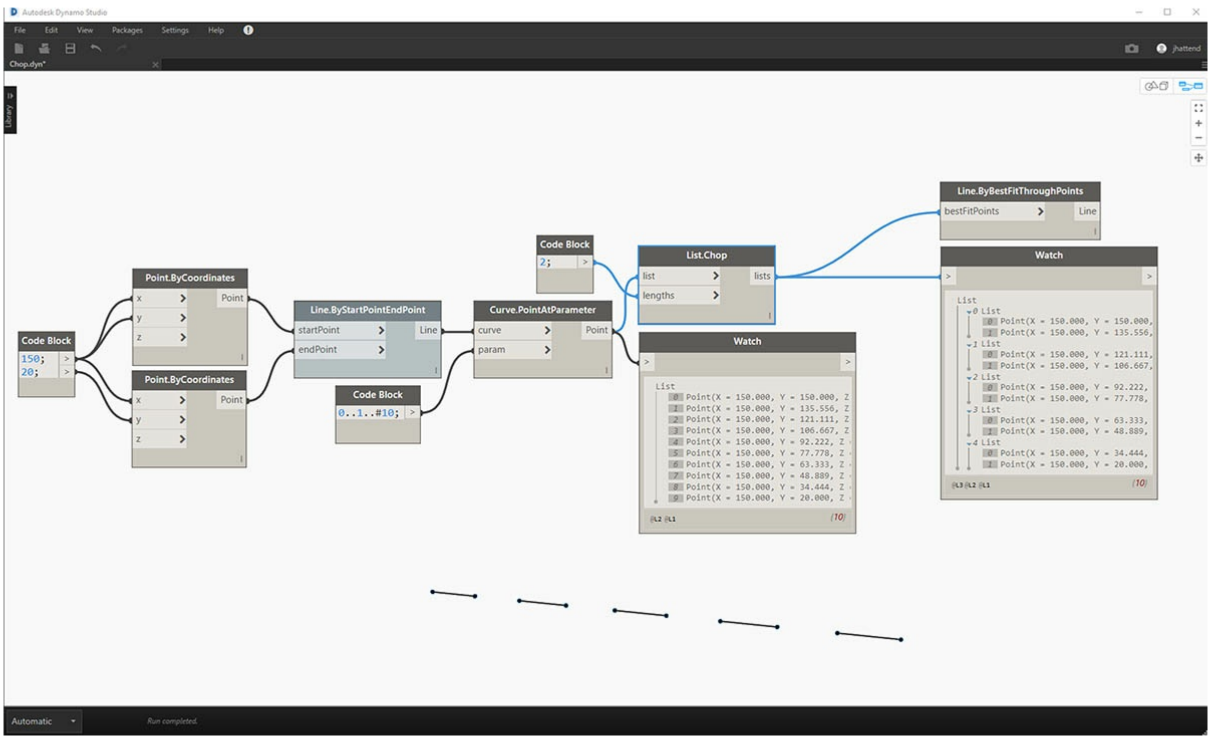
Exercise - List.Chop

> Download the example file that accompanies this exercise (Right click and "Save Link As..."): Chop.dyn. A full list of example files can be found in the Appendix.



A *List.Chop _with a _subLength* of 2 creates 4 lists with 2 items each.

The chop command divides lists based on a given list length. In some ways, chop is the opposite of flatten: rather than removing data structure, it adds new tiers to it. This is a helpful tool for geometric operations like the example below.



```
List
  0 Point(X = 150.000, Y = 150.000, Z
  1 Point(X = 150.000, Y = 135.556, Z
  2 Point(X = 150.000, Y = 121.111, Z
  3 Point(X = 150.000, Y = 106.667, Z
  4 Point(X = 150.000, Y = 92.222, Z
  5 Point(X = 150.000, Y = 77.778, Z
  6 Point(X = 150.000, Y = 63.333, Z
  7 Point(X = 150.000, Y = 48.889, Z
  8 Point(X = 150.000, Y = 34.444, Z
  9 Point(X = 150.000, Y = 20.000, Z
@L2 @L1                          {10}
```

```
List
  0 List
    0 Point(X = 150.000, Y = 150.000,
    1 Point(X = 150.000, Y = 135.556,
  1 List
    0 Point(X = 150.000, Y = 121.111,
    1 Point(X = 150.000, Y = 106.667,
  2 List
    0 Point(X = 150.000, Y = 92.222,
    1 Point(X = 150.000, Y = 77.778,
  3 List
    0 Point(X = 150.000, Y = 63.333,
    1 Point(X = 150.000, Y = 48.889,
  4 List
    0 Point(X = 150.000, Y = 34.444,
    1 Point(X = 150.000, Y = 20.000,
@L3 @L2 @L1                      {10}
```

List.Map and List.Combine

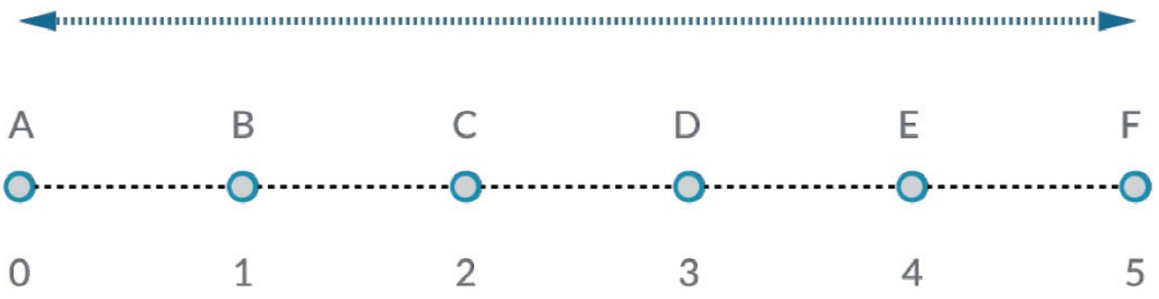
A List.Map/Combine applies a set function to an input list, but one step down in the hierarchy. Combinations are the same as Maps, except combinations can have multiple inputs corresponding to the input of a given function.

Exercise - List.Map

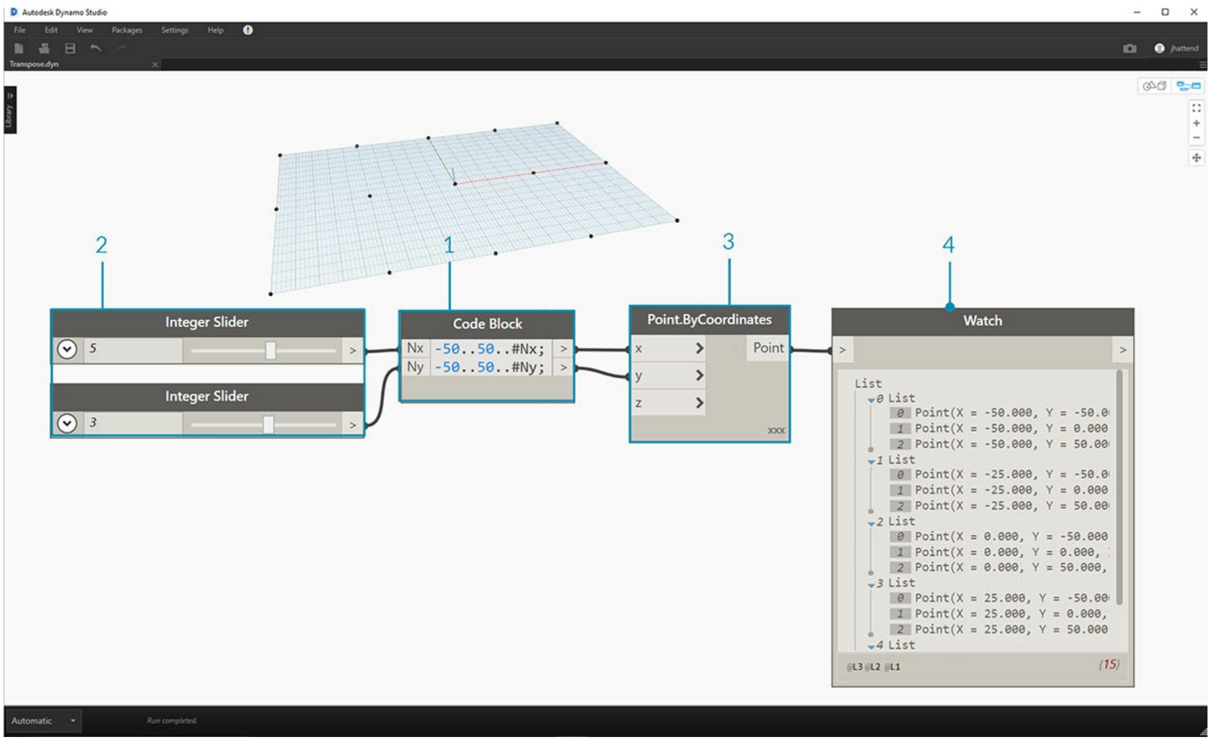
Note: This exercise was created with a previous version of Dynamo. Much of the List.Map functionality has been resolved with the addition of the List@Level feature. For more information, see List@Level below.

> Download the example file that accompanies this exercise (Right click and "Save Link As..."): Map.dyn. A full list of example files can be found in the Appendix.

As a quick introduction, let's review the List.Count node from a previous section.



> The *List.Count* node counts all of the items in a list. We'll use this to demonstrate how *List.Map* works.



```
-50..50..#Nx;
-50..50..#Ny;
```

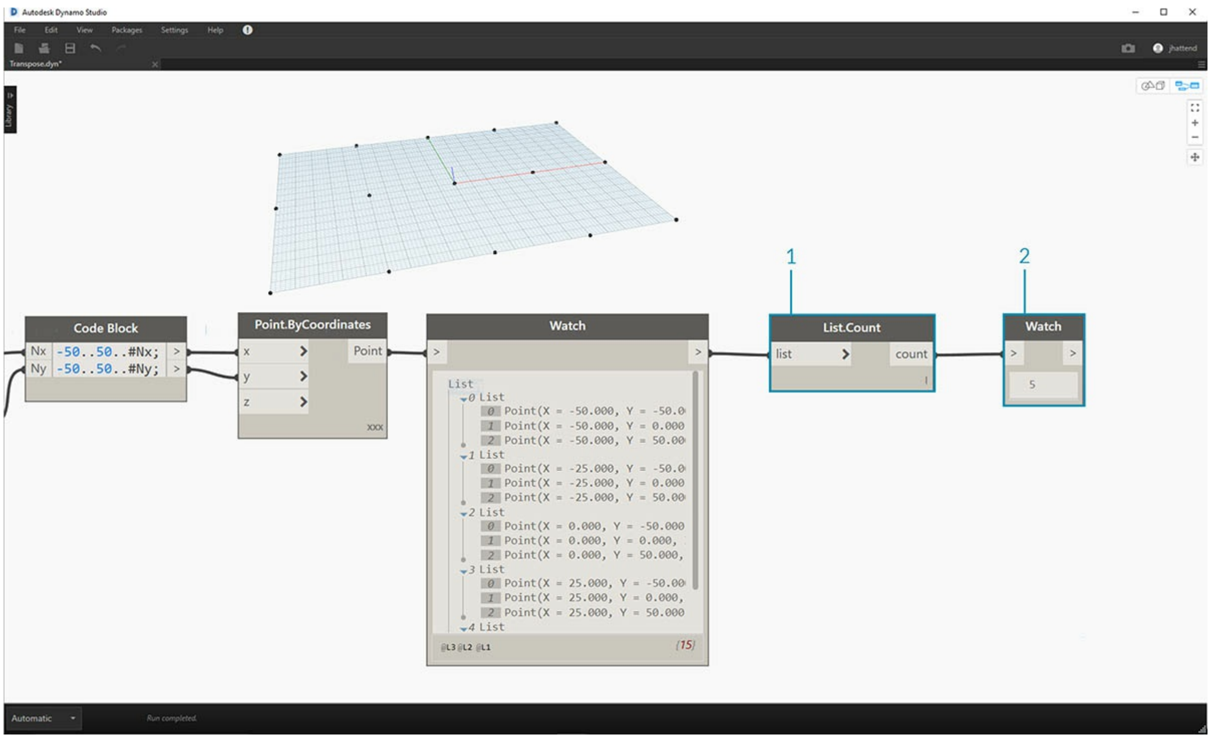
```
List
 0 List
   0 Point(X = -50.000, Y = -50.0
   1 Point(X = -50.000, Y = 0.000
   2 Point(X = -50.000, Y = 50.00
 1 List
   0 Point(X = -25.000, Y = -50.0
   1 Point(X = -25.000, Y = 0.000
   2 Point(X = -25.000, Y = 50.00
 2 List
   0 Point(X = 0.000, Y = -50.000
   1 Point(X = 0.000, Y = 0.000,
   2 Point(X = 0.000, Y = 50.000,
 3 List
   0 Point(X = 25.000, Y = -50.00
   1 Point(X = 25.000, Y = 0.000,
   2 Point(X = 25.000, Y = 50.000
 4 List
@L3 @L2 @L1                  {15}
```

1. Insert two lines of code into the *code block*:

   ```
   -50..50..#Nx;
   -50..50..#Ny;
   ```

   After typing in this code, the code block will create two inputs for Nx and Ny.

2. With two *integer sliders*, define the *Nx* and *Ny* values by connecting them to the *code block*.
3. Connect each line of the code block into the respective *X* and *Y* inputs of a *Point.ByCoordinates* node. Right click the node, select "Lacing", and choose *"Cross Product"*. This creates a grid of points. Because we defined the range from -50 to 50, we are spanning the default Dynamo grid.
4. A *Watch* node reveals the points created. Notice the data structure. We've created a list of lists. Each list represents a row of points of the grid.



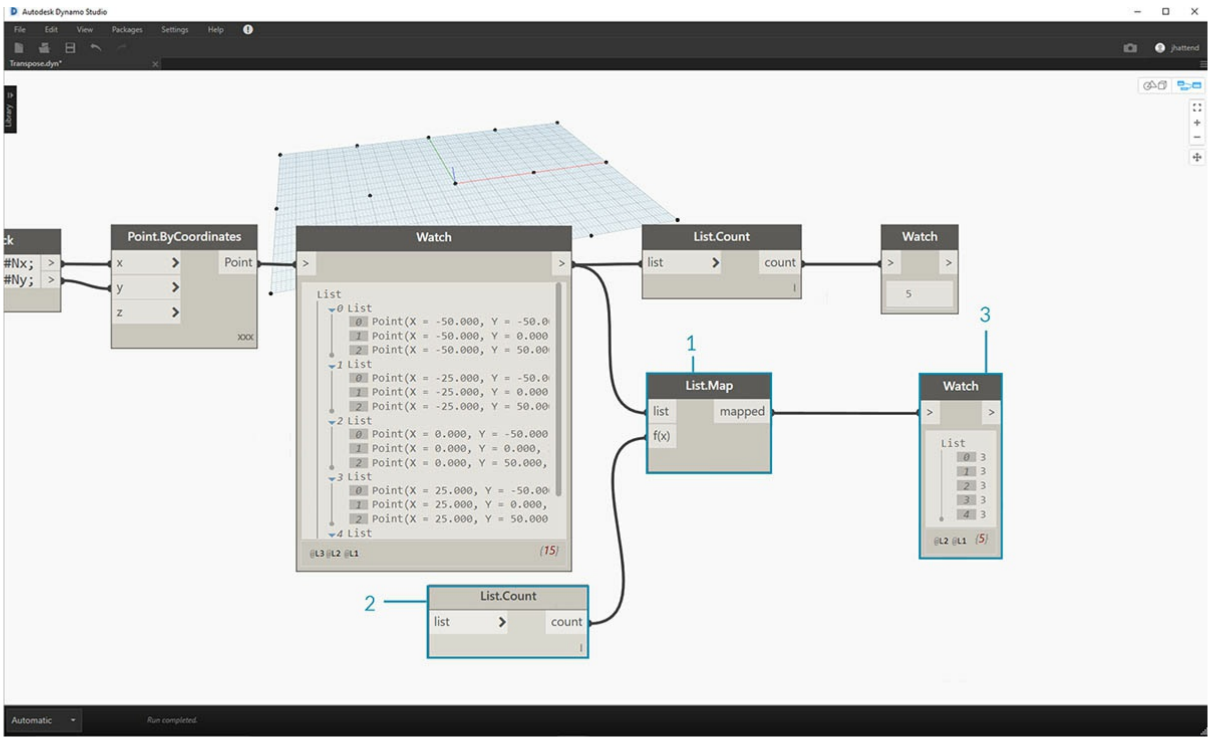
```
-50..50..#Nx;
-50..50..#Ny;
```

```
List
 0 List
   0 Point(X = -50.000, Y = -50.0
   1 Point(X = -50.000, Y = 0.000
   2 Point(X = -50.000, Y = 50.00
 1 List
   0 Point(X = -25.000, Y = -50.0
   1 Point(X = -25.000, Y = 0.000
   2 Point(X = -25.000, Y = 50.00
 2 List
   0 Point(X = 0.000, Y = -50.000
   1 Point(X = 0.000, Y = 0.000,
   2 Point(X = 0.000, Y = 50.000,
 3 List
   0 Point(X = 25.000, Y = -50.00
   1 Point(X = 25.000, Y = 0.000,
   2 Point(X = 25.000, Y = 50.000
 4 List
@L3 @L2 @L1                  {15}
```

1. Attach a *List.Count* node to the output of the watch node from the previous step.
2. Connect a *Watch* node to the List.Count output.

Notice that the List.Count node gives a value of 5. This is equal to the "Nx" variable as defined in the code block. Why is this?

- First, the Point.ByCoordinates node uses the "x" input as the primary input for creating lists. When Nx is 5 and Ny is 3, we get a list of 5 lists, each with 3 items.
- Since Dynamo treats lists as objects in and of themselves, a List.Count node is applied to the main list in the hierarchy. The result is a value of 5, or, the number of lists in the main list.



```
List
 0 List
   0 Point(X = -50.000, Y = -50.0
   1 Point(X = -50.000, Y = 0.000
   2 Point(X = -50.000, Y = 50.00
 1 List
   0 Point(X = -25.000, Y = -50.0
   1 Point(X = -25.000, Y = 0.000
   2 Point(X = -25.000, Y = 50.00
 2 List
   0 Point(X = 0.000, Y = -50.000
   1 Point(X = 0.000, Y = 0.000,
   2 Point(X = 0.000, Y = 50.000,
 3 List
   0 Point(X = 25.000, Y = -50.00
   1 Point(X = 25.000, Y = 0.000,
   2 Point(X = 25.000, Y = 50.000
 4 List
@L3 @L2 @L1                  {15}
```

```
List
 0 3
 1 3
 2 3
 3 3
 4 3
@L2 @L1 {5}
```

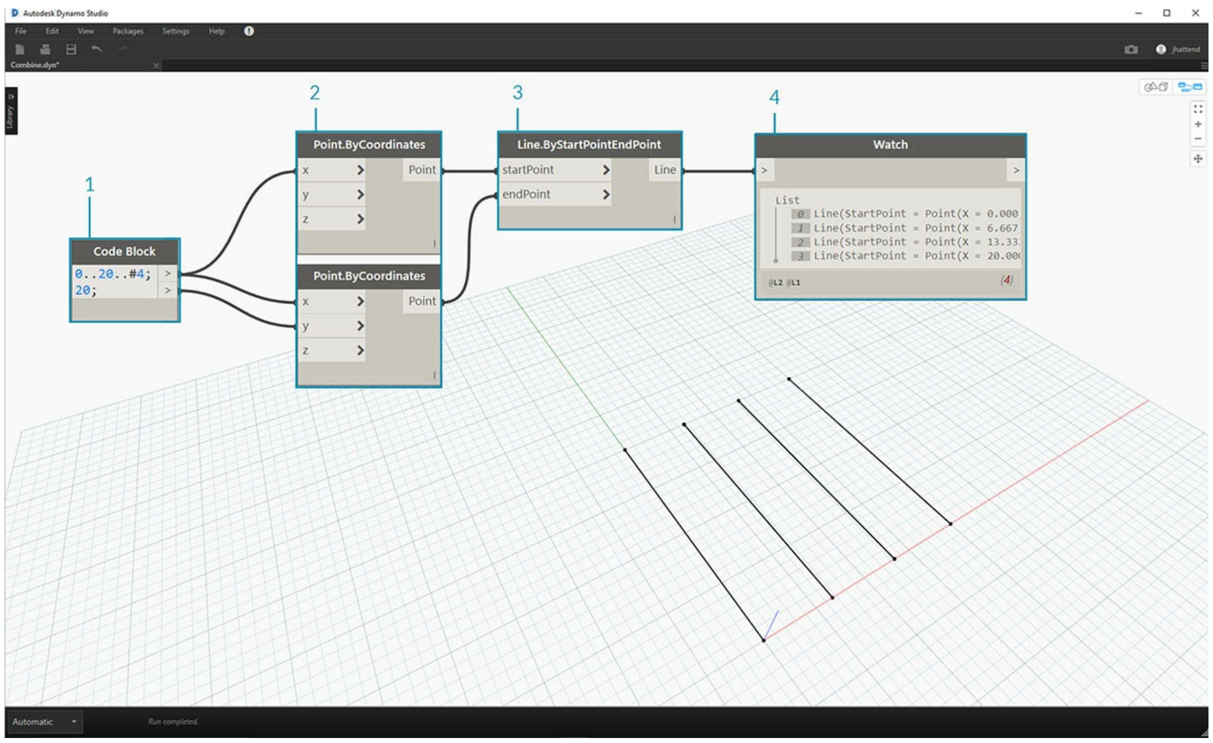
1. By using a *List.Map* node, we take a step down in the hierarchy and perform a *"function"* at this level.
2. Notice that the *List.Count* node has no input. It is being used as a function, so the *List.Count* node will be applied to every individual list one step down in the hierarchy. The blank input of *List.Count* corresponds to the list input of *List.Map*.
3. The results of *List.Count* now gives a list of 5 items, each with a value of 3. This represents the length of each sublist.

Exercise - List.Combine

Note: This exercise was created with a previous version of Dynamo. Much of the List.Combine functionality has been resolved with the addition of the List@Level feature. For more information, see List@Level below.

Download the example file that accompanies this exercise (Right click and "Save Link As..."): Combine.dyn. A full list of example files can be found in the Appendix.

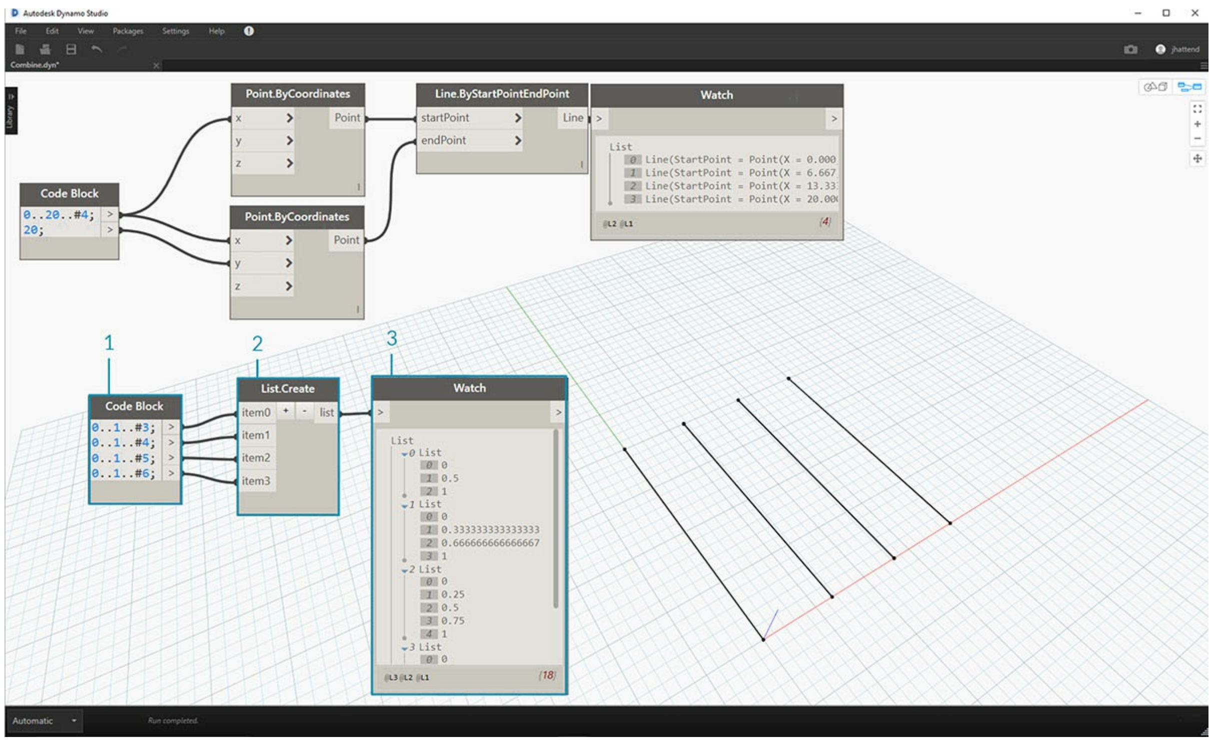
In this exercise, we'll use a similar logic to List.Map, but with multiple elements. In this case, we want to divide a list of curves by a unique number of points.



```
0..20..#4;
20;
```

```
List
  0 Line(StartPoint = Point(X = 0.000
  1 Line(StartPoint = Point(X = 6.667
  2 Line(StartPoint = Point(X = 13.33
  3 Line(StartPoint = Point(X = 20.00
```

1. Using the *code block*, define a range using the syntax: ```..20..#4; `and a value of`20; ``` below that line.
2. Connect the *code block* to two *Point.ByCoordinates* nodes.
3. Create a *Line.ByStartPointEndPoint* from the *Point.ByCoordinates* nodes.
4. The *Watch* node shows four lines.



```
0..20..#4;
20;
```

```
List
  0 Line(StartPoint = Point(X = 0.000
  1 Line(StartPoint = Point(X = 6.667
  2 Line(StartPoint = Point(X = 13.33
  3 Line(StartPoint = Point(X = 20.00
```

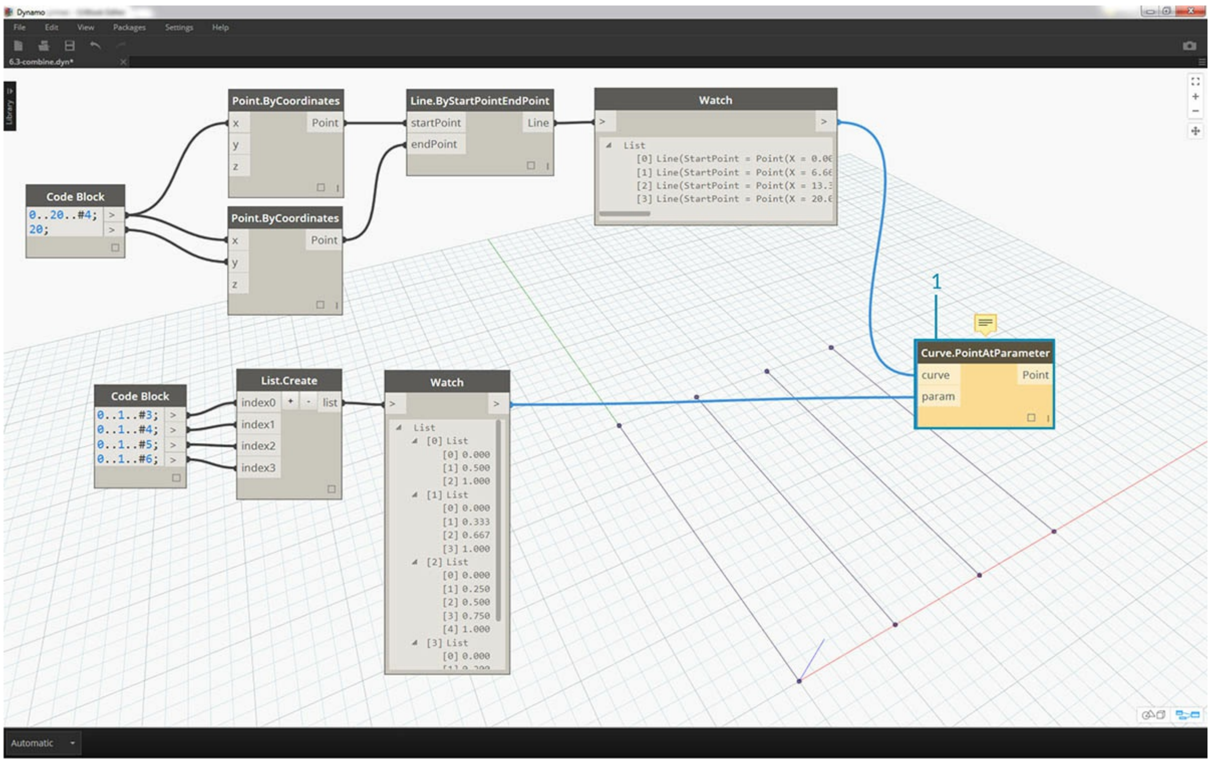
```
0..1..#3;
0..1..#4;
0..1..#5;
0..1..#6;
```

```
List
  0 List
    0 0
    1 0.5
    2 1
  1 List
    0 0
    1 0.333333333333333
    2 0.666666666666667
    3 1
  2 List
    0 0
    1 0.25
    2 0.5
    3 0.75
    4 1
  3 List
    0 0
```

1. Below the graph for line creation, we want to use _code block _to create four distinct ranges to divide the lines uniquely. We do this with the following lines of code:

   ```
   0..1..#3;
   0..1..#4;
   0..1..#5;
   0..1..#6;
   ```

2. With a *List.Create* node, we merge the four lines from the *code block* into one list.
3. The *Watch* node reveals a list of lists.



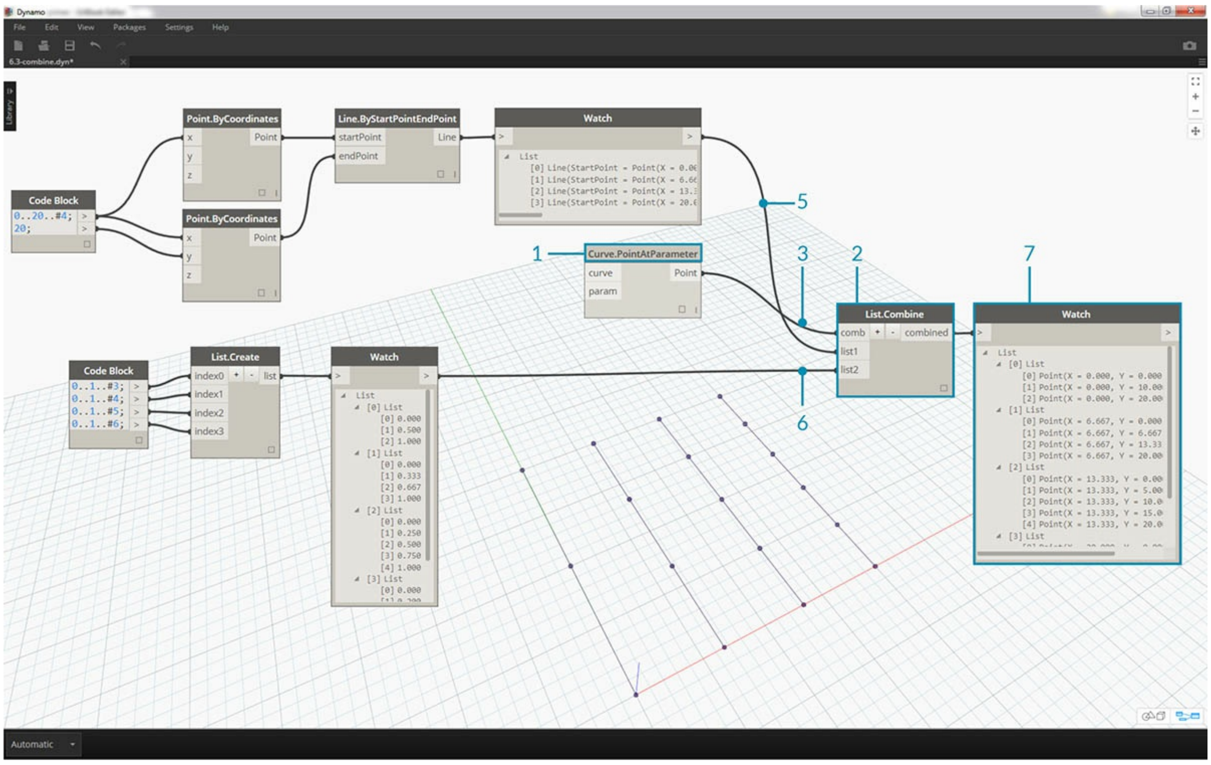
```
0..20..#4;
20;
```

```
0..1..#3;
0..1..#4;
0..1..#5;
0..1..#6;
```

```
List
   [0] Line(StartPoint = Point(X = 0.00
   [1] Line(StartPoint = Point(X = 6.66
   [2] Line(StartPoint = Point(X = 13.3
   [3] Line(StartPoint = Point(X = 20.0
```

```
List
  [0] List
     [0] 0.000
     [1] 0.500
     [2] 1.000
  [1] List
     [0] 0.000
     [1] 0.333
     [2] 0.667
     [3] 1.000
  [2] List
     [0] 0.000
     [1] 0.250
     [2] 0.500
     [3] 0.750
     [4] 1.000
  [3] List
     [0] 0.000
```

1. *Curve.PointAtParameter* will not work by connecting the lines directly into the *parameter* values. We need to step one level down on the hierarchy. For this, we'll use *List.Combine*.



```
List
  [0] List
     [0] Point(X = 0.000, Y = 0.000
     [1] Point(X = 0.000, Y = 10.00
     [2] Point(X = 0.000, Y = 20.00
  [1] List
     [0] Point(X = 6.667, Y = 0.000
     [1] Point(X = 6.667, Y = 6.667
     [2] Point(X = 6.667, Y = 13.33
     [3] Point(X = 6.667, Y = 20.00
  [2] List
     [0] Point(X = 13.333, Y = 0.00
     [1] Point(X = 13.333, Y = 5.00
     [2] Point(X = 13.333, Y = 10.0
     [3] Point(X = 13.333, Y = 15.0
     [4] Point(X = 13.333, Y = 20.0
  [3] List
```

By using *List.Combine*, we can successfully divide each line by the given ranges. This gets a little tricky, so we'll break it down in-depth.

1. First, add a *Curve.PointAtParameter* node to the canvas. This will be the *"function" _or* "combinator" *that we apply to _List.Combine* node. More on that in a second.
2. Add a *List.Combine* node to the canvas. Hit the *"+"* or *"-"* to add or subtract inputs. In this case, we'll use the default two inputs on the node.
3. We want to plug the *Curve.PointAtParameter* node into the *"comb"* input of *List.Combine*. And one more important node: be sure to right-click the *"param" _input of _Curve.PointAtParameter* and uncheck *"use default value"*. Default values in Dynamo inputs have to be removed when running a node as a function. In other words, we should consider default values as having additional nodes wired to them. Because of this, we need to remove the default values in this case.
4. We know we have two inputs, the lines and the parameters to create points. But how do we connect them to the *List.Combine* inputs and in what order?

5. The empty inputs of *Curve.PointAtParameter*, from top-to-bottom need to be filled in the combinator in the same order. So, the lines are plugged into *list1* of *List.Combine*.
6. Following suit, the parameter values are plugged into the *list2* input of *List.Combine*.
7. The *Watch* node and the Dynamo preview shows us that we have 4 lines, each divided based on the *code block* ranges.

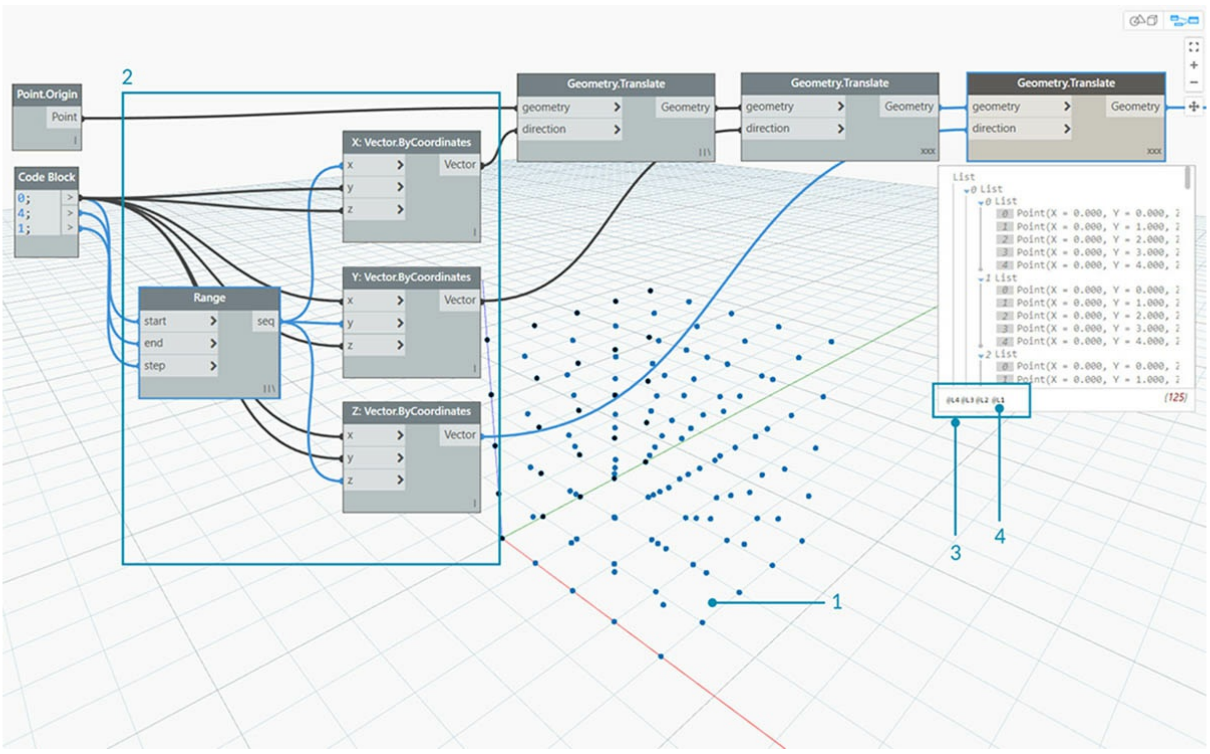
List@Level

Preferred to List.Map, the List@Level feature allows you to directly select which level of list you want to work with right at the input port of the node. This feature can be applied to any incoming input of a node and will allow you access the levels of your lists quicker and easier than other methods. Just tell the node what level of the list you want to use as the input and let the node do the rest.

List@Level Exercise

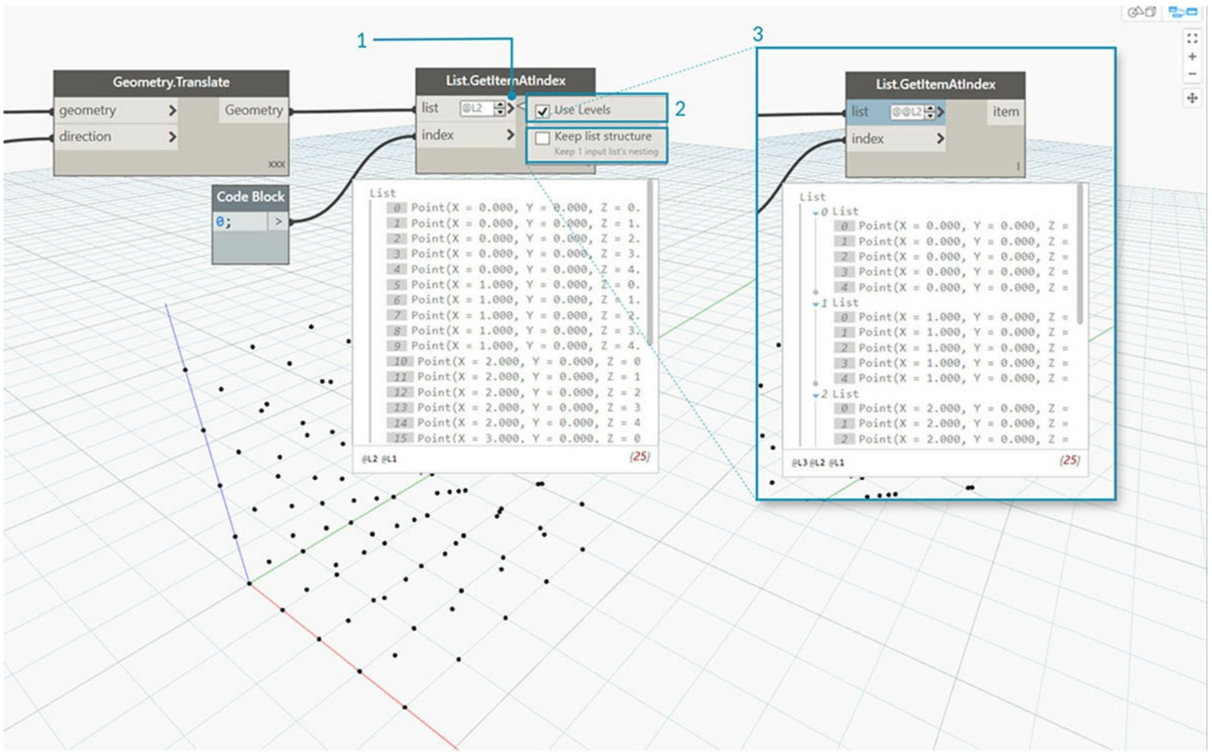
In this exercise, we will use the List@Level feature to isolate a specific level of data.

Download the example file that accompanies this exercise (Right click and "Save Link As..."): List@Level. A full list of example files can be found in the Appendix.



```
List
 0 List
  0 List
   0 Point(X = 0.000, Y = 0.000, Z
   1 Point(X = 0.000, Y = 1.000, Z
   2 Point(X = 0.000, Y = 2.000, Z
   3 Point(X = 0.000, Y = 3.000, Z
   4 Point(X = 0.000, Y = 4.000, Z
  1 List
   0 Point(X = 0.000, Y = 0.000, Z
   1 Point(X = 0.000, Y = 1.000, Z
   2 Point(X = 0.000, Y = 2.000, Z
   3 Point(X = 0.000, Y = 3.000, Z
   4 Point(X = 0.000, Y = 4.000, Z
  2 List
   0 Point(X = 0.000, Y = 0.000, Z
   1 Point(X = 0.000, Y = 1.000, Z
{125}
```

1. We will start with a simple 3D grid of points.
2. Since the grid is constructed with a Range for X, Y and Z, we know that the data is structured with 3 tiers: an X List, Y List and Z List.
3. These tiers exist at different **Levels**. The Levels are indicated at the bottom of the Preview Bubble. The list Levels columns correspond to the list data above to help identify which level to work within.
4. The List Levels are organized in reverse order so that the lowest level data is always in “L1”. This will help ensure that your graphs will work as planned, even if anything is changed upstream.

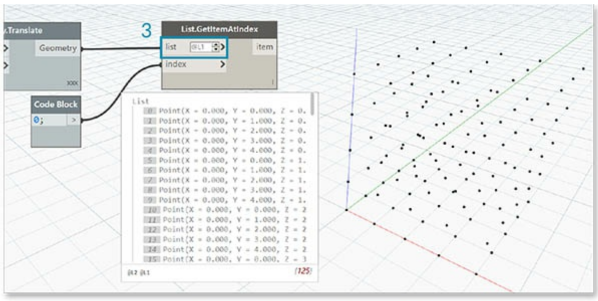


```
List
  0 Point(X = 0.000, Y = 0.000, Z = 0.
  1 Point(X = 0.000, Y = 0.000, Z = 1.
  2 Point(X = 0.000, Y = 0.000, Z = 2.
  3 Point(X = 0.000, Y = 0.000, Z = 3.
  4 Point(X = 0.000, Y = 0.000, Z = 4.
  5 Point(X = 1.000, Y = 0.000, Z = 0.
  6 Point(X = 1.000, Y = 0.000, Z = 1.
  7 Point(X = 1.000, Y = 0.000, Z = 2.
  8 Point(X = 1.000, Y = 0.000, Z = 3.
  9 Point(X = 1.000, Y = 0.000, Z = 4.
  10 Point(X = 2.000, Y = 0.000, Z = 0
  11 Point(X = 2.000, Y = 0.000, Z = 1
  12 Point(X = 2.000, Y = 0.000, Z = 2
  13 Point(X = 2.000, Y = 0.000, Z = 3
  14 Point(X = 2.000, Y = 0.000, Z = 4
  15 Point(X = 3.000, Y = 0.000, Z = 0
@L2 @L1                          {25}
```

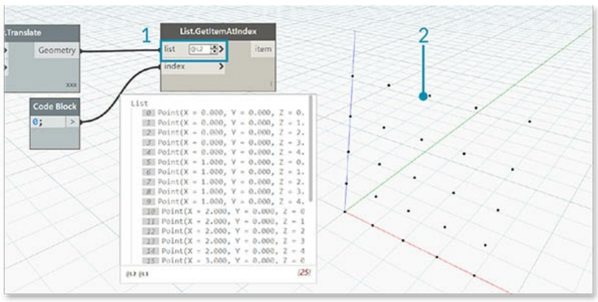
```
List
  0 List
    0 Point(X = 0.000, Y = 0.000, Z =
    1 Point(X = 0.000, Y = 0.000, Z =
    2 Point(X = 0.000, Y = 0.000, Z =
    3 Point(X = 0.000, Y = 0.000, Z =
    4 Point(X = 0.000, Y = 0.000, Z =
  1 List
    0 Point(X = 1.000, Y = 0.000, Z =
    1 Point(X = 1.000, Y = 0.000, Z =
    2 Point(X = 1.000, Y = 0.000, Z =
    3 Point(X = 1.000, Y = 0.000, Z =
    4 Point(X = 1.000, Y = 0.000, Z =
  2 List
    0 Point(X = 2.000, Y = 0.000, Z =
    1 Point(X = 2.000, Y = 0.000, Z =
    2 Point(X = 2.000, Y = 0.000, Z =
@L3 @L2 @L1                      {25}
```

1. To use the List@Level function, click '>'. Inside this menu, you will see two checkboxes.
2. **Use Levels** - This enables the List@Level functionality. After clicking on this option, you will be able to click through and select the input list levels you want the node to use. With this menu, you can quickly try out different level options by clicking up or down.
3. **Keep list structure** – If enabled, you will have the option to keep that input's level structure. Sometimes, you may have purposefully organized your data into sublists. By checking this option, you can keep your list organization intact and not lose any information.

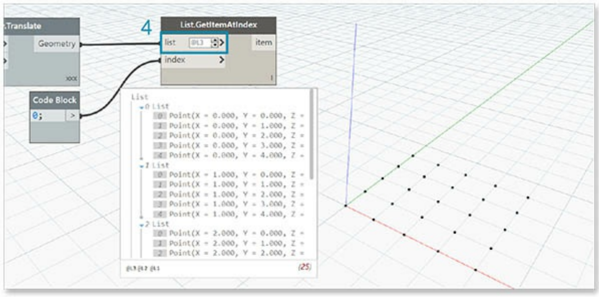
With our simple 3D grid, we can access and visualize the list structure by toggling through the List Levels. Each List Level and index combination will return a different set of points from our original 3D set.



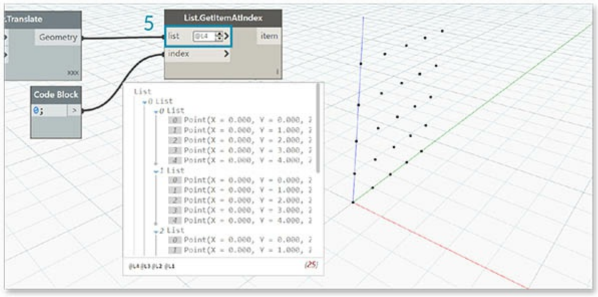
```
List
  0 Point(X = 0.000, Y = 0.000, Z = 0.
  1 Point(X = 0.000, Y = 1.000, Z = 0.
  2 Point(X = 0.000, Y = 2.000, Z = 0.
  3 Point(X = 0.000, Y = 3.000, Z = 0.
  4 Point(X = 0.000, Y = 4.000, Z = 0.
  5 Point(X = 0.000, Y = 0.000, Z = 1.
  6 Point(X = 0.000, Y = 1.000, Z = 1.
  7 Point(X = 0.000, Y = 2.000, Z = 1.
  8 Point(X = 0.000, Y = 3.000, Z = 1.
  9 Point(X = 0.000, Y = 4.000, Z = 1.
  10 Point(X = 0.000, Y = 0.000, Z = 2
  11 Point(X = 0.000, Y = 1.000, Z = 2
  12 Point(X = 0.000, Y = 2.000, Z = 2
  13 Point(X = 0.000, Y = 3.000, Z = 2
  14 Point(X = 0.000, Y = 4.000, Z = 2
  15 Point(X = 0.000, Y = 0.000, Z = 3
@L2 @L1                          {125}
```



```
List
  0 Point(X = 0.000, Y = 0.000, Z = 0.
  1 Point(X = 0.000, Y = 0.000, Z = 1.
  2 Point(X = 0.000, Y = 0.000, Z = 2.
  3 Point(X = 0.000, Y = 0.000, Z = 3.
  4 Point(X = 0.000, Y = 0.000, Z = 4.
  5 Point(X = 1.000, Y = 0.000, Z = 0.
  6 Point(X = 1.000, Y = 0.000, Z = 1.
  7 Point(X = 1.000, Y = 0.000, Z = 2.
  8 Point(X = 1.000, Y = 0.000, Z = 3.
  9 Point(X = 1.000, Y = 0.000, Z = 4.
  10 Point(X = 2.000, Y = 0.000, Z = 0
  11 Point(X = 2.000, Y = 0.000, Z = 1
  12 Point(X = 2.000, Y = 0.000, Z = 2
  13 Point(X = 2.000, Y = 0.000, Z = 3
  14 Point(X = 2.000, Y = 0.000, Z = 4
  15 Point(X = 3.000, Y = 0.000, Z = 0
@L2 @L1                          {25}
```



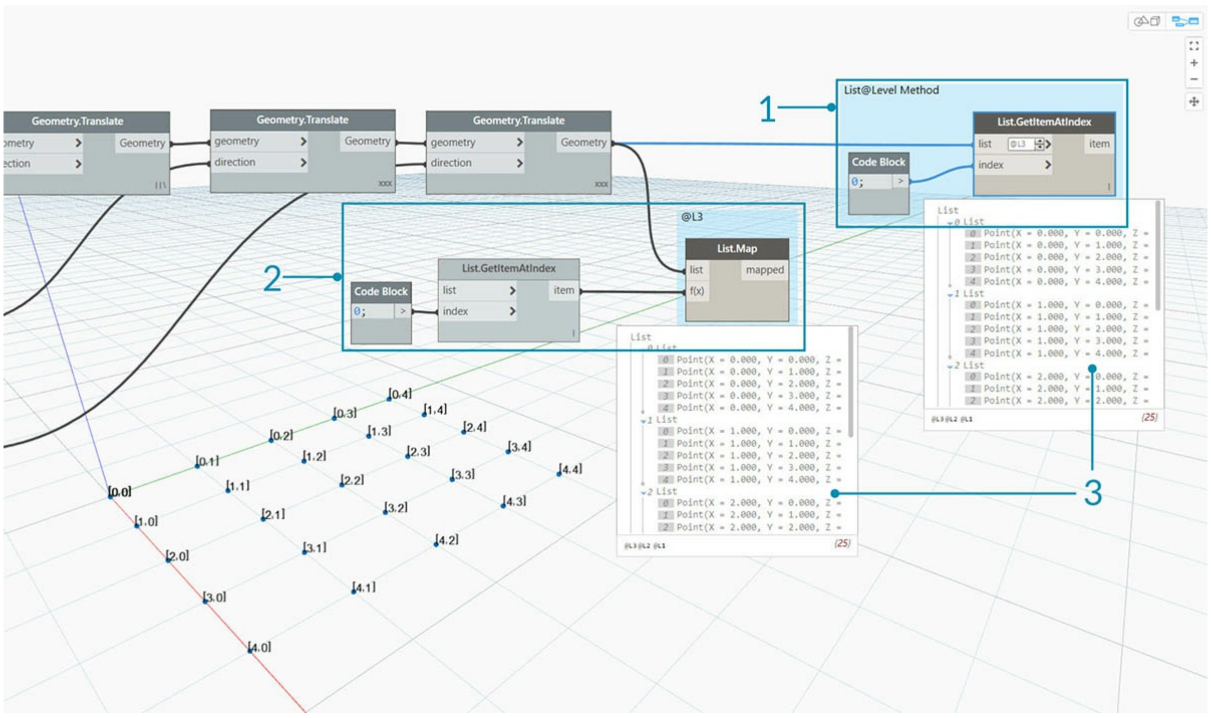
```
List
  0 List
    0 Point(X = 0.000, Y = 0.000, Z =
    1 Point(X = 0.000, Y = 1.000, Z =
    2 Point(X = 0.000, Y = 2.000, Z =
    3 Point(X = 0.000, Y = 3.000, Z =
    4 Point(X = 0.000, Y = 4.000, Z =
  1 List
    0 Point(X = 1.000, Y = 0.000, Z =
    1 Point(X = 1.000, Y = 1.000, Z =
    2 Point(X = 1.000, Y = 2.000, Z =
    3 Point(X = 1.000, Y = 3.000, Z =
    4 Point(X = 1.000, Y = 4.000, Z =
  2 List
    0 Point(X = 2.000, Y = 0.000, Z =
    1 Point(X = 2.000, Y = 1.000, Z =
    2 Point(X = 2.000, Y = 2.000, Z =
@L3 @L2 @L1                      {25}
```



```
List
  0 List
    0 List
      0 Point(X = 0.000, Y = 0.000, Z
      1 Point(X = 0.000, Y = 1.000, Z
      2 Point(X = 0.000, Y = 2.000, Z
      3 Point(X = 0.000, Y = 3.000, Z
      4 Point(X = 0.000, Y = 4.000, Z
    1 List
      0 Point(X = 0.000, Y = 0.000, Z
      1 Point(X = 0.000, Y = 1.000, Z
      2 Point(X = 0.000, Y = 2.000, Z
      3 Point(X = 0.000, Y = 3.000, Z
      4 Point(X = 0.000, Y = 4.000, Z
    2 List
      0 Point(X = 0.000, Y = 0.000, Z
      1 Point(X = 0.000, Y = 1.000, Z
@L4 @L3 @L2 @L1                  {25}
```

1. "@L2" in DesignScript allows us to select only the List at Level 2.
2. The List at Level 2 with the index 0 includes only the first set of Y points, returning only the XZ grid.
3. If we change the Level filter to "L1", we will be able to see everything in the first List Level. The List at Level 1 with the index 0 includes all of our 3D points in a flat list.
4. If we try the same for "L3" we will see only the third List Level points. The List at Level 3 with the index 0 includes only the first set of Z points, returning only an XY grid.
5. If we try the same for "L4" we will see only the third List Level points. The List at Level 4 with the index 0 includes only the first set of X points, returning only an YZ grid.

Although this particular example can also be created with List.Map, List@Level greatly simplifies the interaction, making it easy to access the node data. Take a look below at a comparison between a List.Map and List@Level methods:



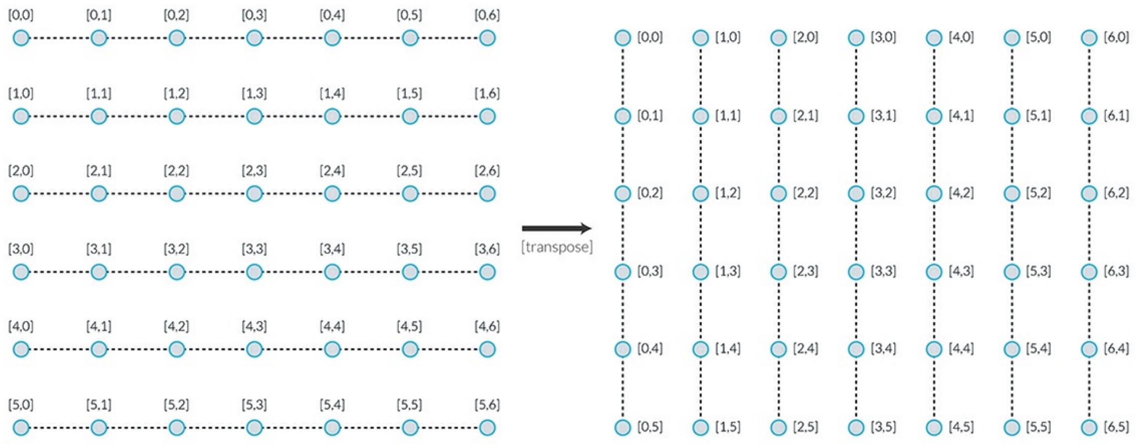
```
List
 0 List
   0 Point(X = 0.000, Y = 0.000, Z =
   1 Point(X = 0.000, Y = 1.000, Z =
   2 Point(X = 0.000, Y = 2.000, Z =
   3 Point(X = 0.000, Y = 3.000, Z =
   4 Point(X = 0.000, Y = 4.000, Z =
 1 List
   0 Point(X = 1.000, Y = 0.000, Z =
   1 Point(X = 1.000, Y = 1.000, Z =
   2 Point(X = 1.000, Y = 2.000, Z =
   3 Point(X = 1.000, Y = 3.000, Z =
   4 Point(X = 1.000, Y = 4.000, Z =
 2 List
   0 Point(X = 2.000, Y = 0.000, Z =
   1 Point(X = 2.000, Y = 1.000, Z =
   2 Point(X = 2.000, Y = 2.000, Z =
@L3 @L2 @L1                          {25}
```

```
List
 0 List
   0 Point(X = 0.000, Y = 0.000, Z =
   1 Point(X = 0.000, Y = 1.000, Z =
   2 Point(X = 0.000, Y = 2.000, Z =
   3 Point(X = 0.000, Y = 3.000, Z =
   4 Point(X = 0.000, Y = 4.000, Z =
 1 List
   0 Point(X = 1.000, Y = 0.000, Z =
   1 Point(X = 1.000, Y = 1.000, Z =
   2 Point(X = 1.000, Y = 2.000, Z =
   3 Point(X = 1.000, Y = 3.000, Z =
   4 Point(X = 1.000, Y = 4.000, Z =
 2 List
   0 Point(X = 2.000, Y = 0.000, Z =
   1 Point(X = 2.000, Y = 1.000, Z =
   2 Point(X = 2.000, Y = 2.000, Z =
@L3 @L2 @L1                          {25}
```

1. Although both methods will give us access to the same points, the List@Level method allows us to easily toggle between layers of data within a single node.
2. To access a point grid with List.Map, we will need a List.GetItemAtIndex node alongside the List.Map. For every list level that we are stepping down, we will need to use an additional List.Map node. Depending on the complexity of your lists, this could require you to add a significant amount of List.Map Nodes to your graph to access the right level of information.
3. In this example, a List.GetItemAtIndex node with a List.Map node reurns the same set of points with the same list structure as the List.GetItemAtIndex with '@L3' selected.

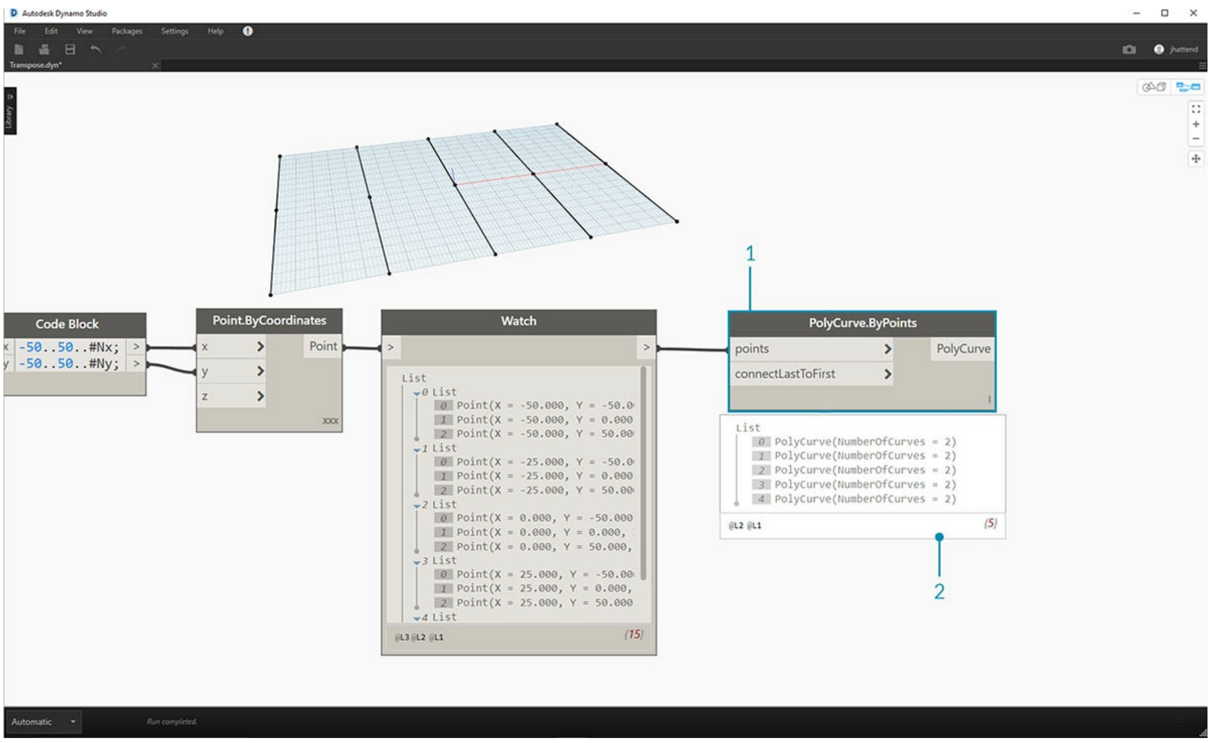
Transpose

Transpose is a fundamental function when dealing with lists of lists. Just as in spreadsheet programs, a transpose flips the columns and rows of a data structure. We'll demonstrate this with a basic matrix below, and in the following section, we'll demonstrate how a transpose can be use to create geometric relationships.



Exercise - List.Transpose

Download the example file that accompanies this exercise (Right click and "Save Link As..."): Transpose.dyn. A full list of example files can be found in the Appendix.



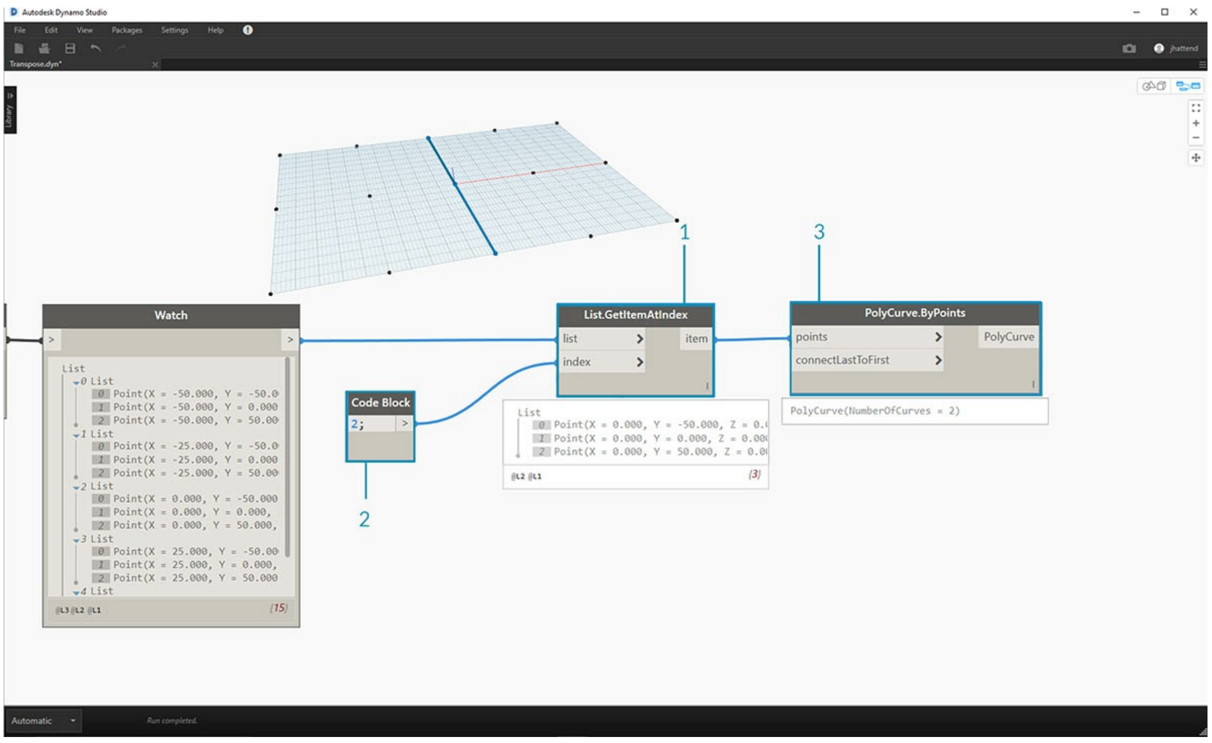
```
-50..50..#Nx;
-50..50..#Ny;
```

```
List
 0 List
   0 Point(X = -50.000, Y = -50.0
   1 Point(X = -50.000, Y = 0.000
   2 Point(X = -50.000, Y = 50.00
 1 List
   0 Point(X = -25.000, Y = -50.0
   1 Point(X = -25.000, Y = 0.000
   2 Point(X = -25.000, Y = 50.00
 2 List
   0 Point(X = 0.000, Y = -50.000
   1 Point(X = 0.000, Y = 0.000,
   2 Point(X = 0.000, Y = 50.000,
 3 List
   0 Point(X = 25.000, Y = -50.00
   1 Point(X = 25.000, Y = 0.000,
   2 Point(X = 25.000, Y = 50.000
 4 List
@L3 @L2 @L1                  {15}
```

```
List
   0 PolyCurve(NumberOfCurves = 2)
   1 PolyCurve(NumberOfCurves = 2)
   2 PolyCurve(NumberOfCurves = 2)
   3 PolyCurve(NumberOfCurves = 2)
   4 PolyCurve(NumberOfCurves = 2)
@L2 @L1                       {5}
```

Let's delete the *List.Count* nodes from the previous exercise and move on to some geometry to see how the data structured.

1. Connect a *PolyCurve.ByPoints* to the output of the watch node from *Point.ByCoordinates*.
2. The output shows 5 polycurves, and we can see the curves in our Dynamo preview. The Dynamo node is looking for a list of points (or a list of lists of points in this case) and creating a single polycurve from them. Essentially, each list has converted to a curve in the data structure.



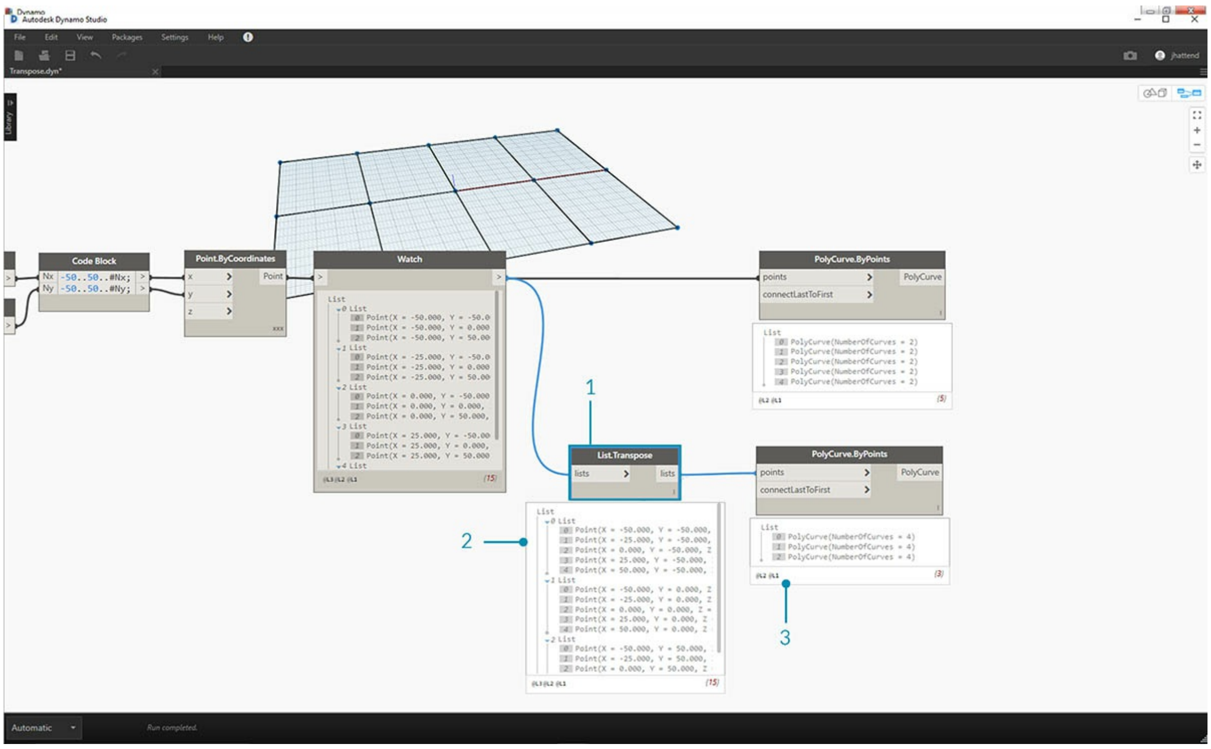
```
List
 0 List
   0 Point(X = -50.000, Y = -50.0
   1 Point(X = -50.000, Y = 0.000
   2 Point(X = -50.000, Y = 50.00
 1 List
   0 Point(X = -25.000, Y = -50.0
   1 Point(X = -25.000, Y = 0.000
   2 Point(X = -25.000, Y = 50.00
 2 List
   0 Point(X = 0.000, Y = -50.000
   1 Point(X = 0.000, Y = 0.000,
   2 Point(X = 0.000, Y = 50.000,
 3 List
   0 Point(X = 25.000, Y = -50.00
   1 Point(X = 25.000, Y = 0.000,
   2 Point(X = 25.000, Y = 50.000
 4 List
@L3 @L2 @L1                  {15}
```

```
2;
```

```
List
   0 Point(X = 0.000, Y = -50.000, Z = 0.0
   1 Point(X = 0.000, Y = 0.000, Z = 0.00
   2 Point(X = 0.000, Y = 50.000, Z = 0.0
@L2 @L1                       {3}
```

```
PolyCurve(NumberOfCurves = 2)
```

1. If we want to isolate one row of curves, we use the *List.GetItemAtIndex* node.
2. Using a *code block* value of 2, query the 3rd element in the main list.
3. The *PolyCurve.ByPoints* gives us one curve, since only one list is connected to the node.



```
Nx  -50..50..#Nx;
Ny  -50..50..#Ny;
```

```
List
 0 List
   0 Point(X = -50.000, Y = -50.0
   1 Point(X = -50.000, Y = 0.000
   2 Point(X = -50.000, Y = 50.00
 1 List
   0 Point(X = -25.000, Y = -50.0
   1 Point(X = -25.000, Y = 0.000
   2 Point(X = -25.000, Y = 50.00
 2 List
   0 Point(X = 0.000, Y = -50.000
   1 Point(X = 0.000, Y = 0.000,
   2 Point(X = 0.000, Y = 50.000,
 3 List
   0 Point(X = 25.000, Y = -50.00
   1 Point(X = 25.000, Y = 0.000,
   2 Point(X = 25.000, Y = 50.000
 4 List
@L3 @L2 @L1                {15}
```

```
List
  0 PolyCurve(NumberOfCurves = 2)
  1 PolyCurve(NumberOfCurves = 2)
  2 PolyCurve(NumberOfCurves = 2)
  3 PolyCurve(NumberOfCurves = 2)
  4 PolyCurve(NumberOfCurves = 2)
@L2 @L1                {5}
```

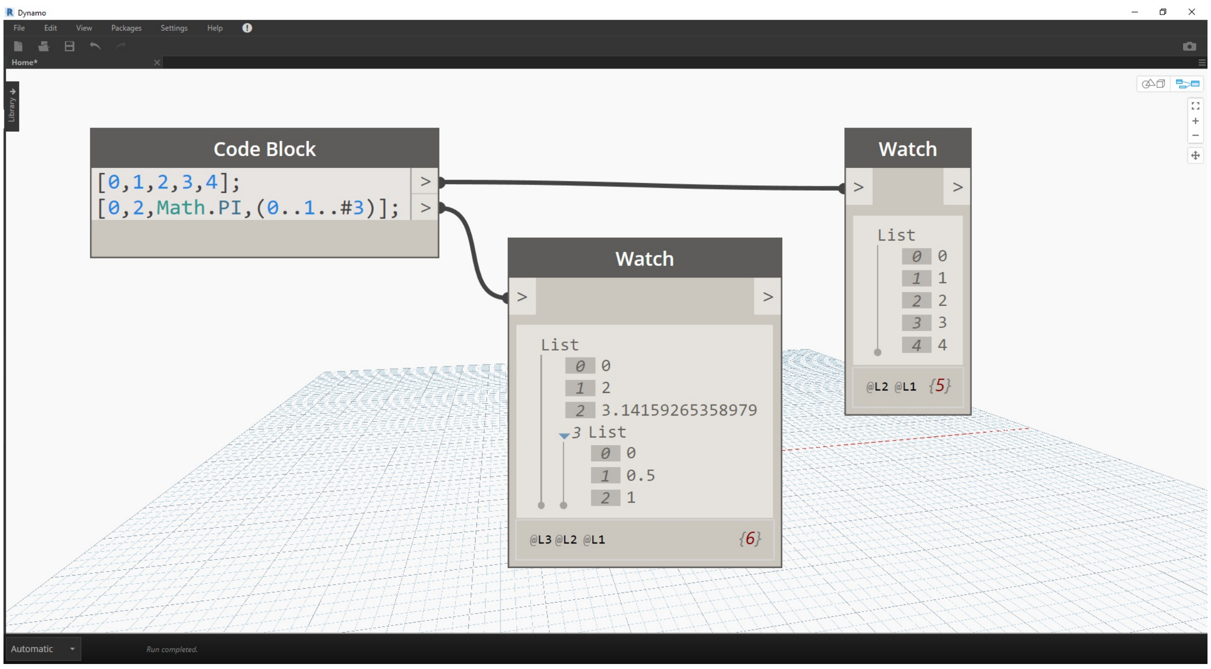
```
List
 0 List
   0 Point(X = -50.000, Y = -50.000,
   1 Point(X = -25.000, Y = -50.000,
   2 Point(X = 0.000, Y = -50.000, Z
   3 Point(X = 25.000, Y = -50.000,
   4 Point(X = 50.000, Y = -50.000,
 1 List
   0 Point(X = -50.000, Y = 0.000, Z
   1 Point(X = -25.000, Y = 0.000, Z
   2 Point(X = 0.000, Y = 0.000, Z =
   3 Point(X = 25.000, Y = 0.000, Z
   4 Point(X = 50.000, Y = 0.000, Z
 2 List
   0 Point(X = -50.000, Y = 50.000,
   1 Point(X = -25.000, Y = 50.000,
   2 Point(X = 0.000, Y = 50.000, Z
@L3 @L2 @L1                {15}
```

```
List
  0 PolyCurve(NumberOfCurves = 4)
  1 PolyCurve(NumberOfCurves = 4)
  2 PolyCurve(NumberOfCurves = 4)
@L2 @L1                {3}
```

1. A *List.Transpose* node will switch all of the items with all of the lists in a list of lists. This sounds complicated, but it's the same logic as transpose in Microsoft Excel: switching columns with rows in a data structure.
2. Notice the abstract result: the transpose changed the list structure from a 5 lists with 3 items each to 3 lists with 5 items each.
3. Notice the geometric result: using *PolyCurve.ByPoints*, we get 3 polycurves in the perpendicular direction to the original curves.

Code Block Creation

Code block shorthand uses "[]" to define a list. This is a much faster and more fluid way to create list than the List.Create node. Code block is discussed in more detail in Chapter 7. Reference the image below to note how a list with multiple expressions can be defined with code block.



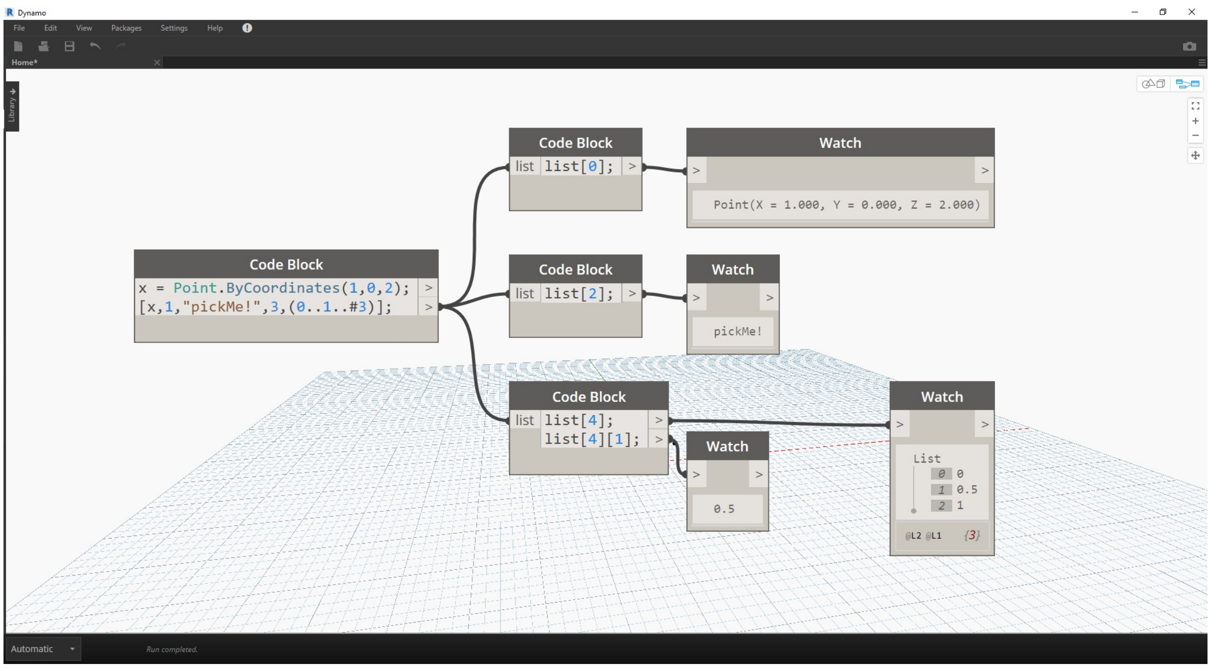
```
[0,1,2,3,4];
[0,2,Math.PI,(0..1..#3)];
```

```
List
  0 0
  1 2
  2 3.14159265358979
  3 List
    0 0
    1 0.5
    2 1
@L3 @L2 @L1      {6}
```

```
List
  0 0
  1 1
  2 2
  3 3
  4 4
@L2 @L1  {5}
```

Code Block Query

Code block shorthand uses "[]" as a quick and easy way to select specific items that you want from a complex data structure. Code blocks are discussed in more detail in Chapter 7. Reference the image below to note how a list with multiple data types can be queried with code block.



```
x = Point.ByCoordinates(1,0,2);
[x,1,"pickMe!",3,(0..1..#3)];
```

```
list[0];
```

```
Point(X = 1.000, Y = 0.000, Z = 2.000)
```

```
list[2];
```

```
pickMe!
```

```
list[4];
list[4][1];
```

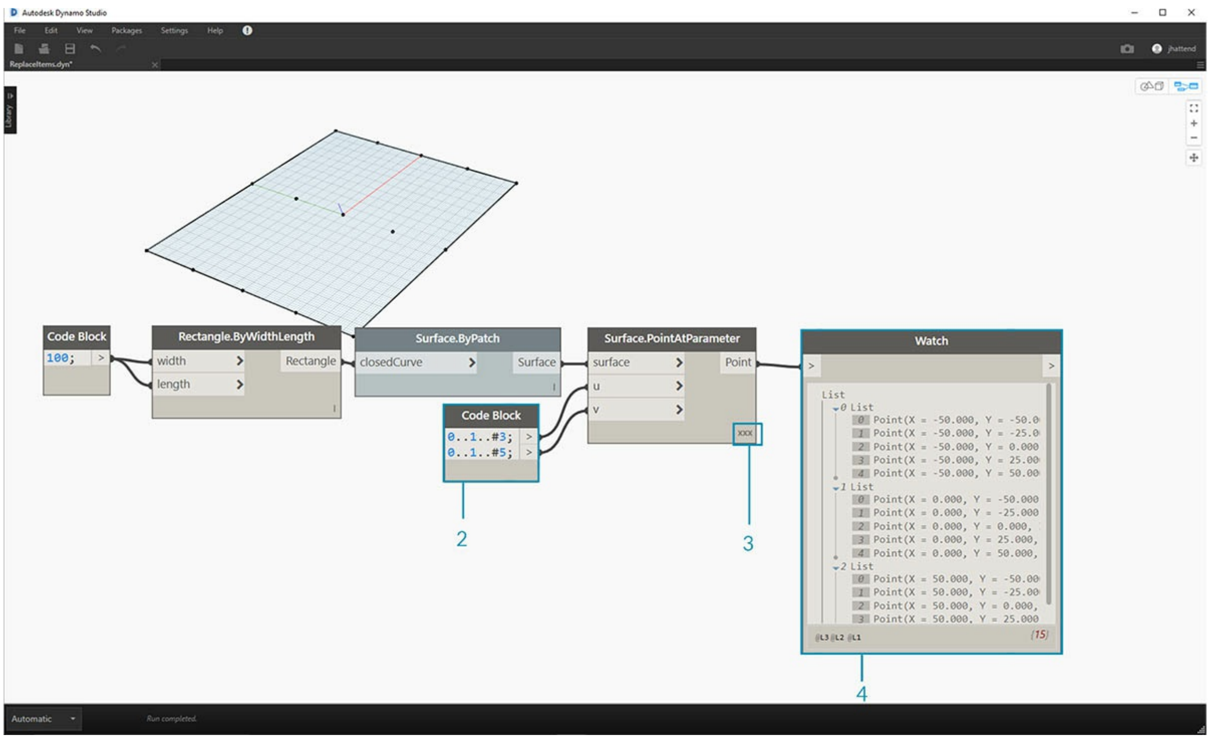
```
0.5
```

```
List
  0 0
  1 0.5
  2 1
@L2 @L1   {3}
```

Exercise - Querying and Inserting Data

Download the example file that accompanies this exercise (Right click and "Save Link As..."): ReplaceItems.dyn. A full list of example files can be found in the Appendix.

This exercise uses some of the logic established in the previous one to edit a surface. Our goal here is intuitive, but the data structure navigation will be more involved. We want to articulate a surface by moving a control point.



```
100;
```

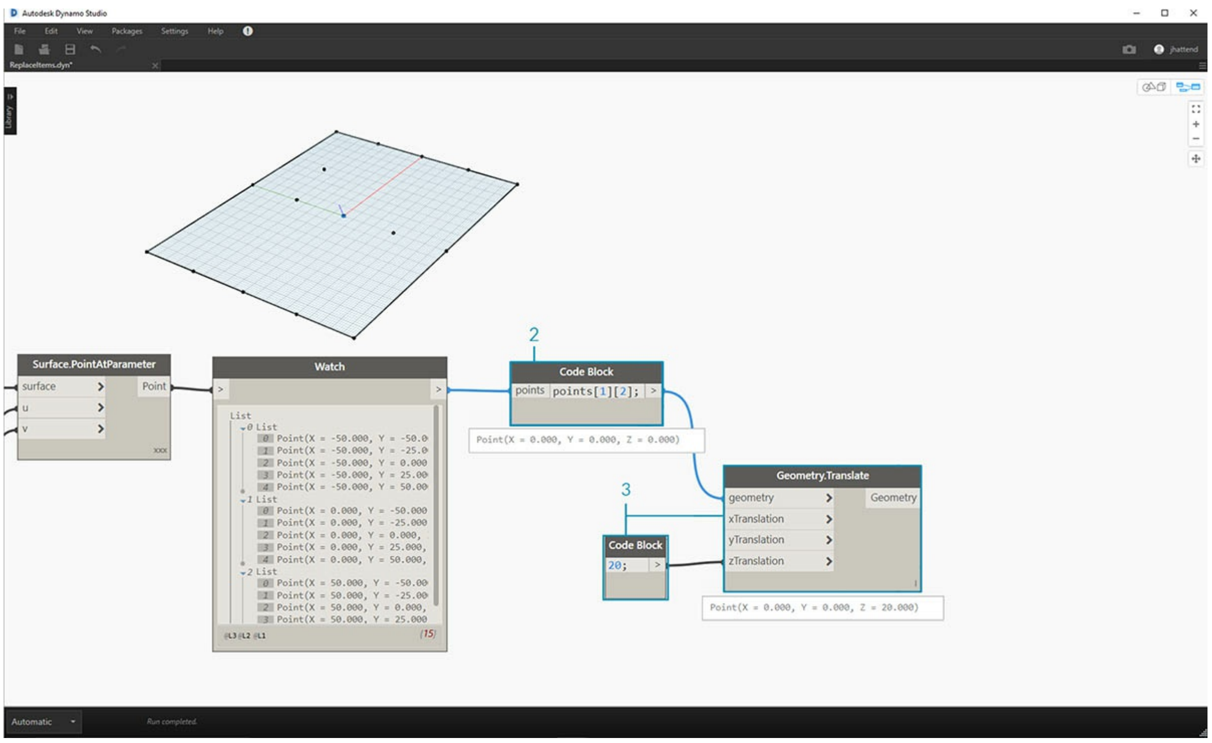
```
0..1..#3;
0..1..#5;
```

```
List
 0 List
   0 Point(X = -50.000, Y = -50.0
   1 Point(X = -50.000, Y = -25.0
   2 Point(X = -50.000, Y = 0.000
   3 Point(X = -50.000, Y = 25.00
   4 Point(X = -50.000, Y = 50.00
 1 List
   0 Point(X = 0.000, Y = -50.000
   1 Point(X = 0.000, Y = -25.000
   2 Point(X = 0.000, Y = 0.000,
   3 Point(X = 0.000, Y = 25.000,
   4 Point(X = 0.000, Y = 50.000,
 2 List
   0 Point(X = 50.000, Y = -50.00
   1 Point(X = 50.000, Y = -25.00
   2 Point(X = 50.000, Y = 0.000,
   3 Point(X = 50.000, Y = 25.000
@L3 @L2 @L1   {15}
```

1. Begin with the string of nodes above. We are creating a basic surface which spans the default Dynamo grid.
2. Using *code block*, insert these two lines of code and connect to the *u* and *v* inputs of *Surface.PointAtParameter*, respectively:

   ```
   -50..50..#3;
   -50..50..#5;
   ```

3. Be sure to set the Lacing of *Surface.PointAtParameter* to *"Cross Product"*.
4. The *Watch* node show that we have a list of 3 lists, each with 5 items.



```
points[1][2];
```

```
Point(X = 0.000, Y = 0.000, Z = 0.000)
```

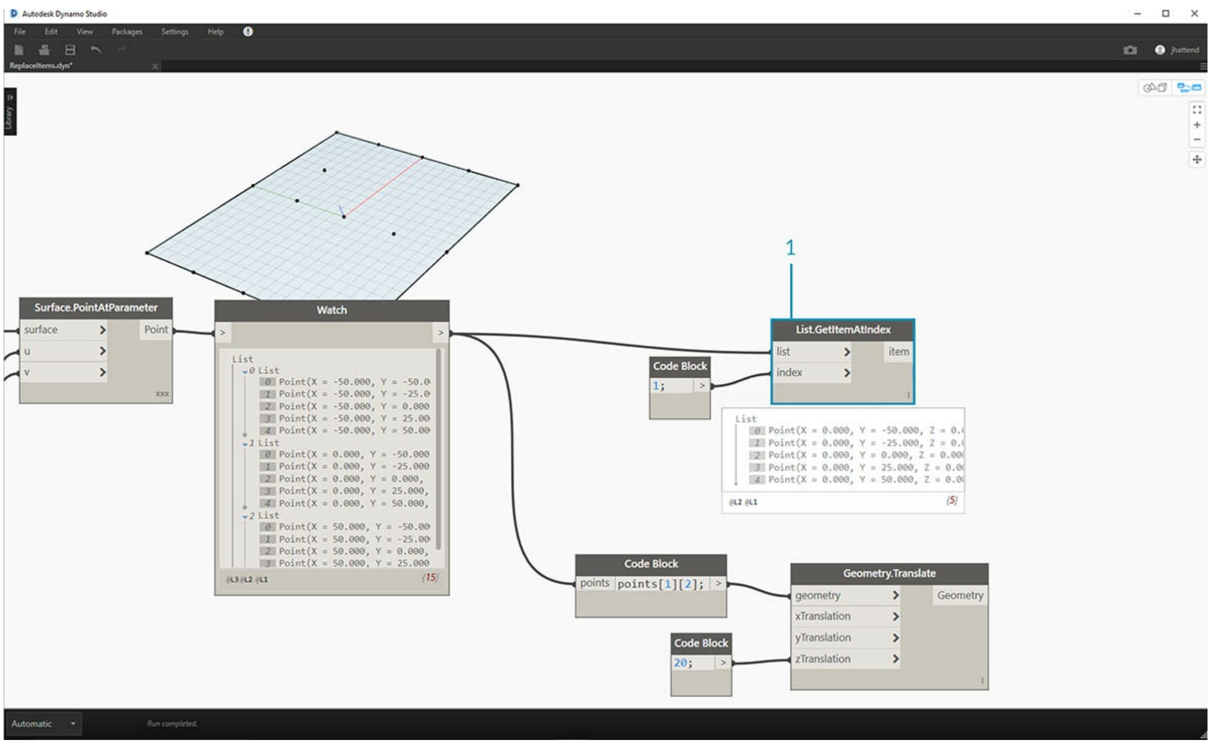
```
List
 0 List
  0 Point(X = -50.000, Y = -50.0
  1 Point(X = -50.000, Y = -25.0
  2 Point(X = -50.000, Y = 0.000
  3 Point(X = -50.000, Y = 25.00
  4 Point(X = -50.000, Y = 50.00
 1 List
  0 Point(X = 0.000, Y = -50.000
  1 Point(X = 0.000, Y = -25.000
  2 Point(X = 0.000, Y = 0.000,
  3 Point(X = 0.000, Y = 25.000,
  4 Point(X = 0.000, Y = 50.000,
 2 List
  0 Point(X = 50.000, Y = -50.00
  1 Point(X = 50.000, Y = -25.00
  2 Point(X = 50.000, Y = 0.000,
  3 Point(X = 50.000, Y = 25.000
@L3 @L2 @L1                  {15}
```

```
20;
```

```
Point(X = 0.000, Y = 0.000, Z = 20.000)
```

In this step, we want to query the central point in the grid we've created. To do this we'll select the middle point in the middle list. Makes sense, right?

1. To confirm that this is the correct point, we can also click through the watch node items to confirm that we're targeting the correct one.
2. Using *code block*, we'll write a basic line of code for querying a list of lists: `points[1][2];`
3. Using *Geometry.Translate*, we'll move the selected point up in the *Z* direction by *20* units.



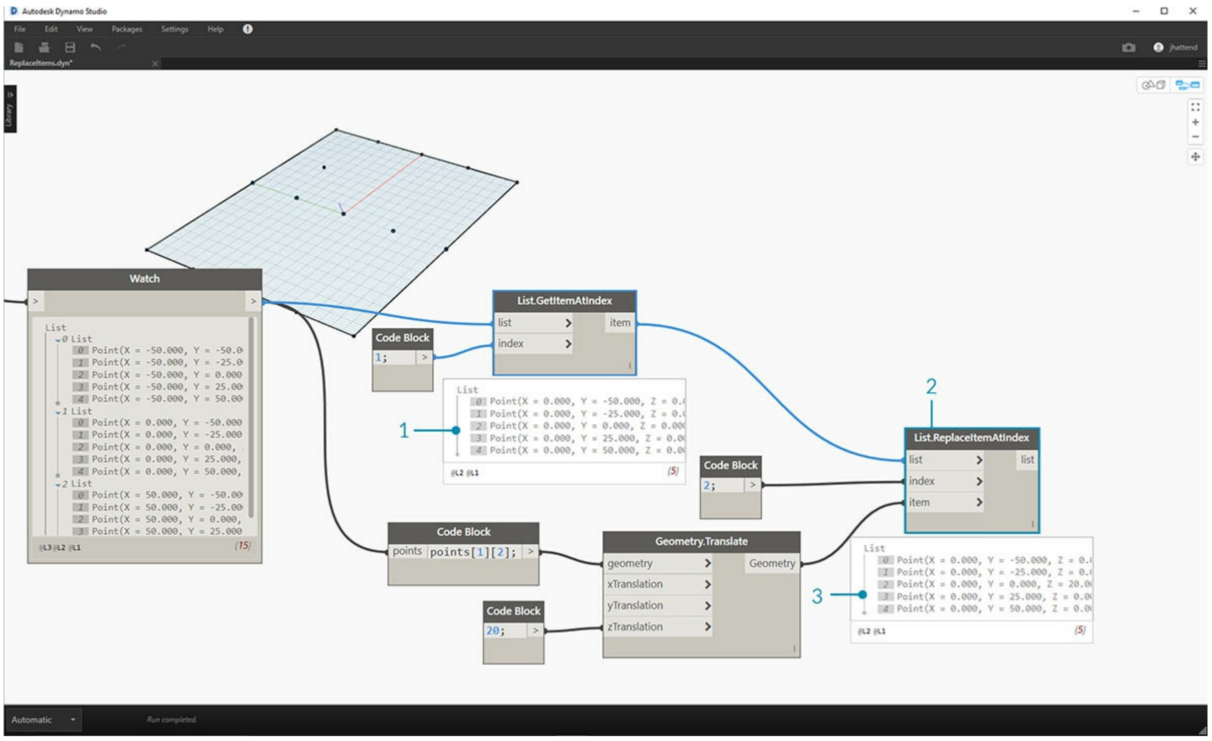
```
1;
```

```
List
  0 Point(X = 0.000, Y = -50.000, Z = 0.
  1 Point(X = 0.000, Y = -25.000, Z = 0.
  2 Point(X = 0.000, Y = 0.000, Z = 0.00
  3 Point(X = 0.000, Y = 25.000, Z = 0.0
  4 Point(X = 0.000, Y = 50.000, Z = 0.0
@L2 @L1                  {5}
```

```
points[1][2];
```

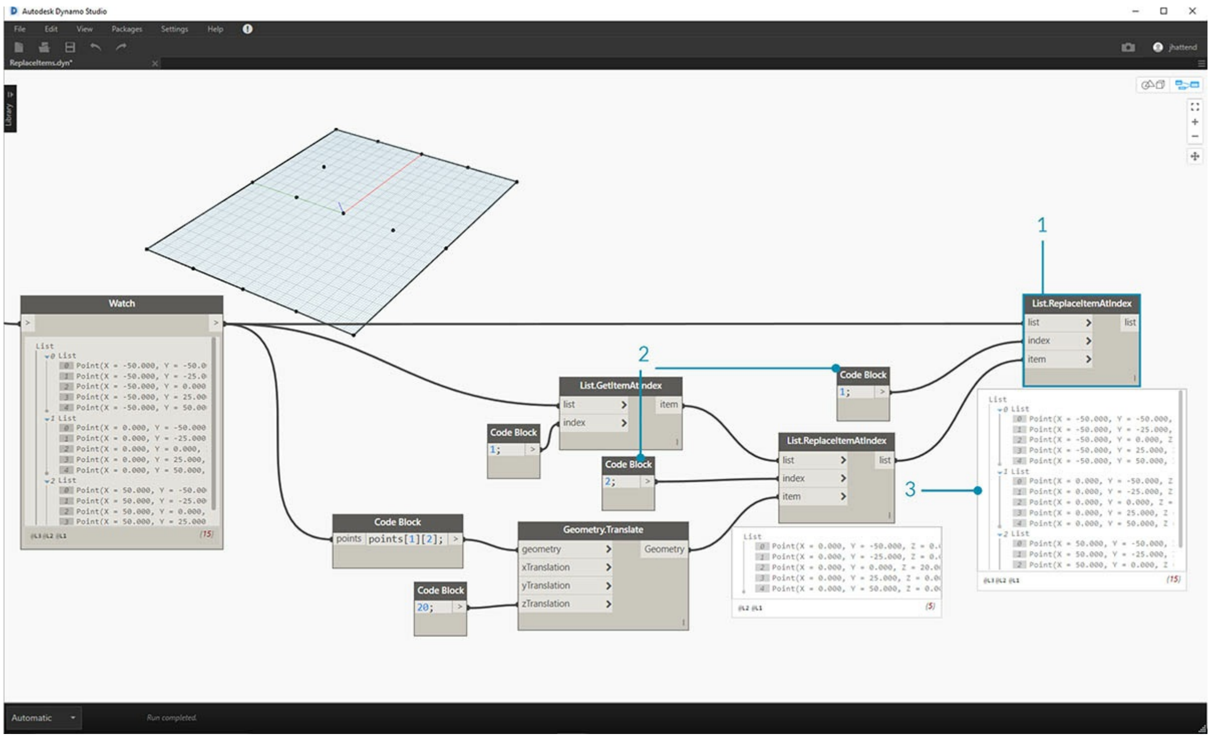
```
20;
```

1. Let's also select the middle row of points with a *List.GetItemAtIndex* node. Note: Similar to a previous step, we can also query the list with *code block*, using a line of `points[1];`



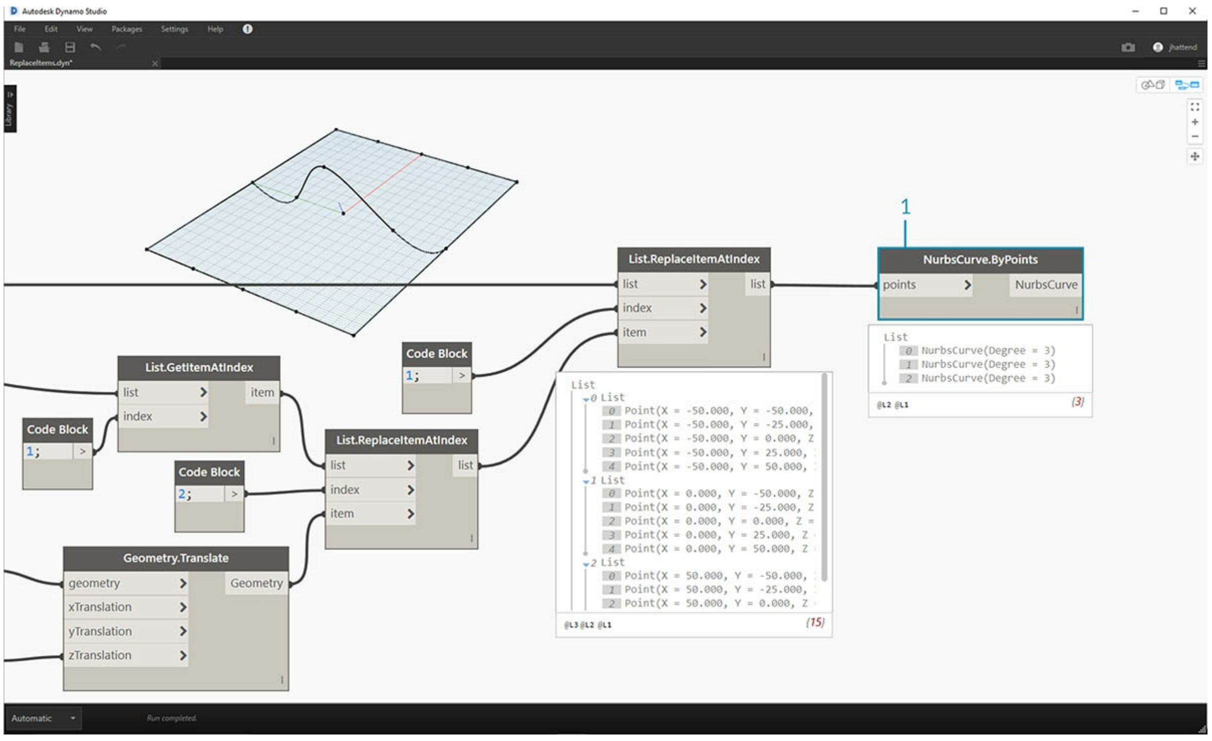
So far we've successfully queried the center point and moved it upward. Now we want need to insert this moved point back into the original data structure.

1. First, we want to replace the item of the list we isolated in a previous step.
2. Using *List.ReplaceItemAtIndex*, we'll replace the middle item by using and index of *"2"*, with the replacement item connected to the moved point (*Geometry.Translate*).
3. The output shows that we've input the moved point into the middle item of the list.



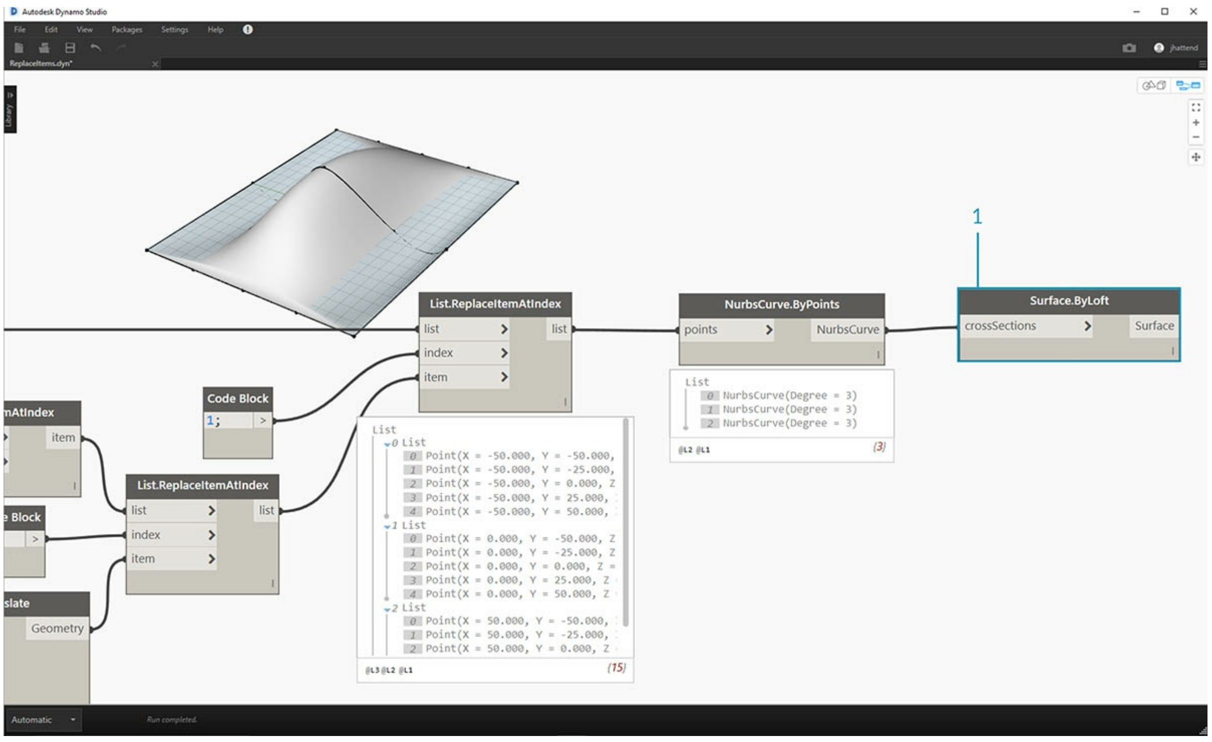
Now that we've modified the list, we need to insert this list back into the original data structure: the list of lists.

1. Following the same logic, use *List.ReplaceItemAtIndex* to replace the middle list with the our modified list.
2. Notice that the *code blocks* defining the index for these two nodes are 1 and 2, which matches the original query from the *code block* (*points[1][2]*).
3. By selecting the list at *index 1*, we see the data structure highlighted in the Dynamo preview. We successfully merged the moved point into the original data structure.



There are many ways to make a surface from this set of points. In this case, we're going to create a surface by lofting curves together.

1. Create a *NurbsCurve.ByPoints* node and connect the new data structure to create three nurbs curves.



1. Connect a *Surface.ByLoft* to the output from *NurbsCurve.ByPoints*. We now have a modified surface. We can change the original *Z* value of Geometry. Translate and watch the geometry update!

n-Dimensional Lists

n-Dimensional Lists

Further down the rabbit-hole, let's add even more tiers to hierarchy. Data structure can expand far beyond a two-dimensional list of lists. Since lists are items in and of themselves in Dynamo, we can create data with as many dimensions as possible.

The analogy we'll work with here are Russian Nesting Dolls. Each list can be regarded as one container holding multiple items. Each list has its own properties and is also regarded as its own object.

A set of Russian Nesting Dolls (Photo by Zeta) is an analogy for n-Dimensional lists. Each layer represents a list, and each list contains items within it. In Dynamo's case, each container can have multiple containers inside (representing the items of each list).

n-Dimensional lists are difficult to explain visually, but we've set up a few exercises in this chapter which focus on working with lists which venture beyond two dimensions.

Mapping and Combinations

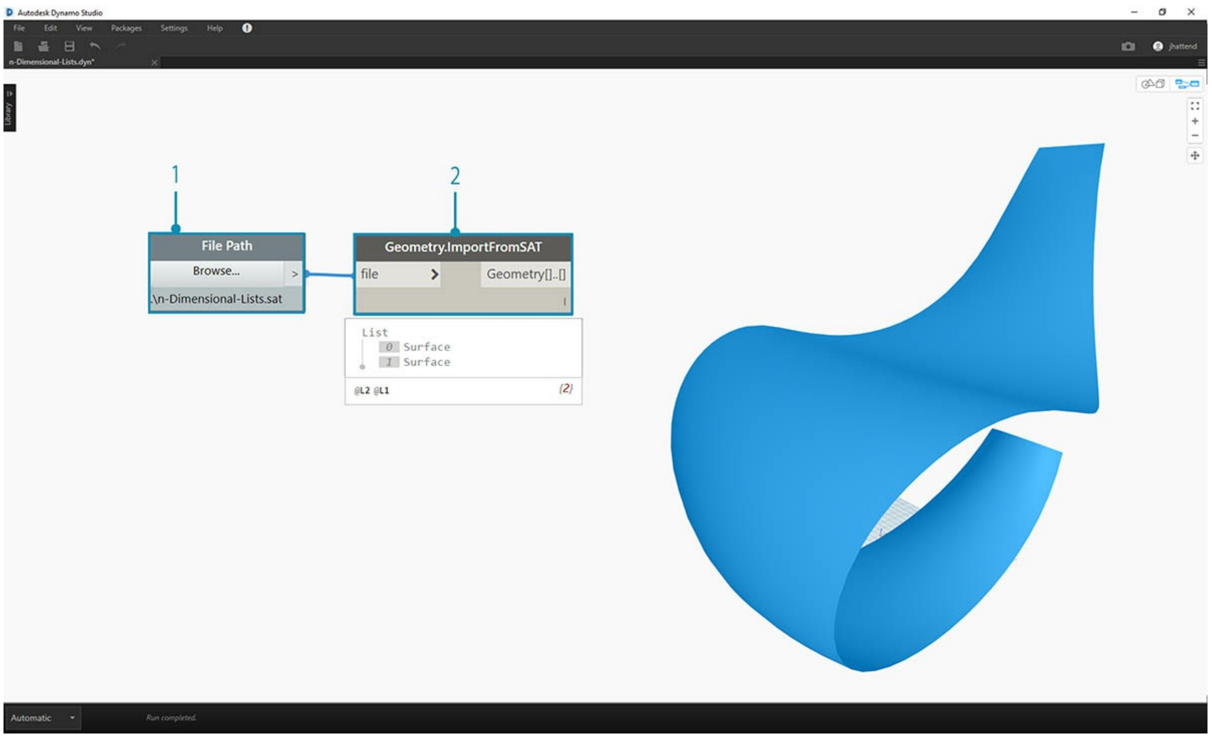
Mapping is arguably the most complex part of data management in Dynamo, and is especially relevant when working with complex hierarchies of lists. With the series of exercises below, we'll demonstrate when to use mapping and combinations as data becomes multi-dimensional.

Preliminary introductions to List.Map and List.Combine can be found in the previous section. In the last exercise below, we'll use these nodes on a complex data structure.

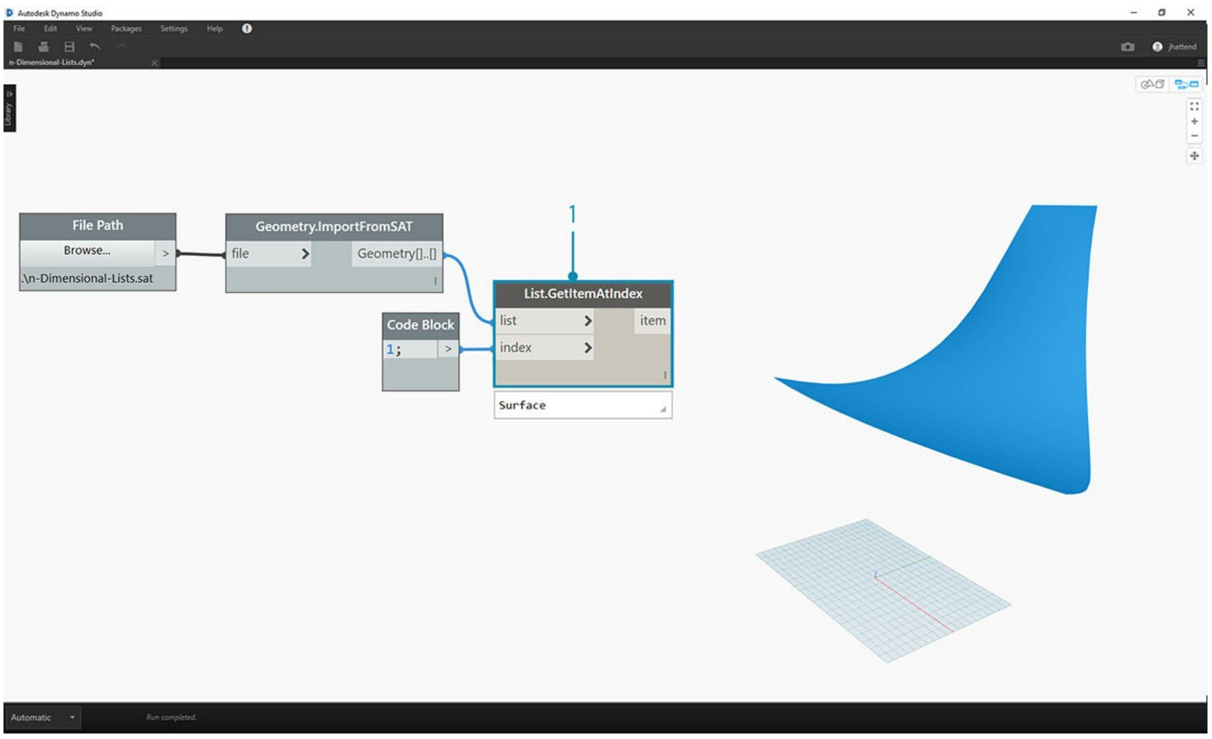
Exercise - 2D Lists - Basic

Download the example files that accompanies this exercise (Right click and "Save Link As..."). A full list of example files can be found in the Appendix. 1.n-Dimensional-Lists.dyn 2.n-Dimensional-Lists.sat

This exercise is the first in a series of three which focuses on articulating imported geometry. Each part in this series of exercises will increase in the complexity of data structure.

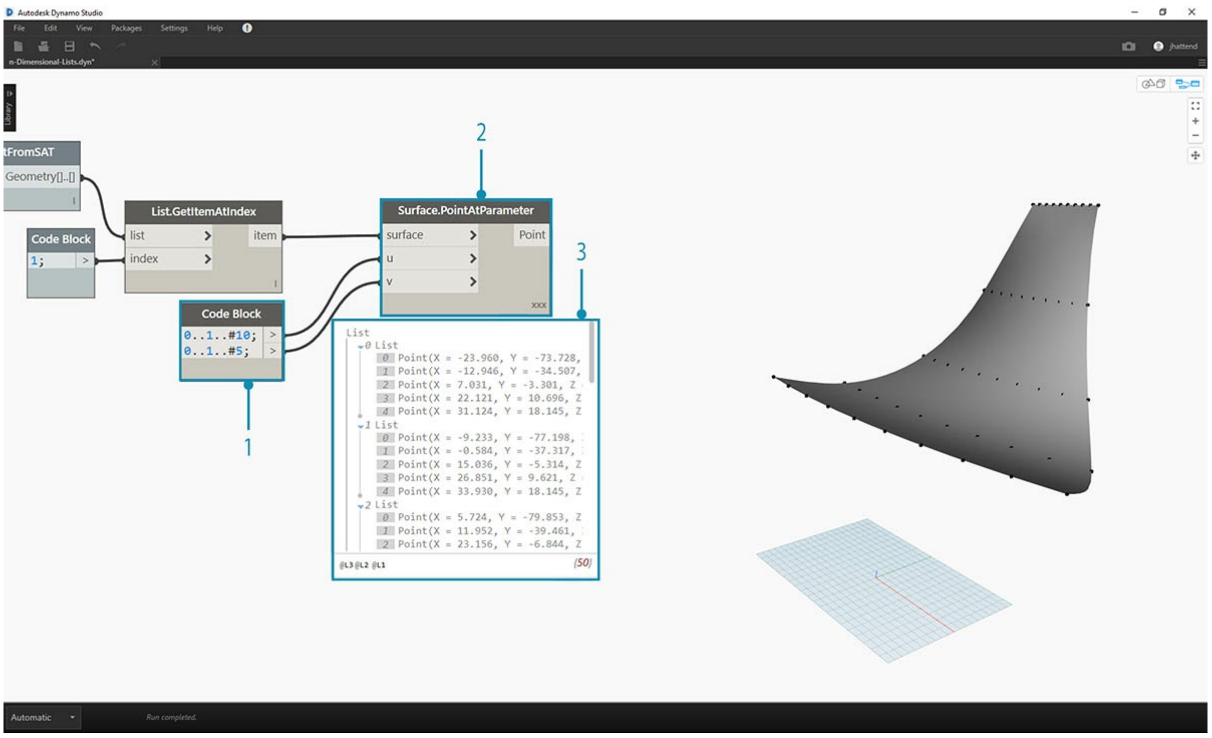


1. Let's begin with the .sat file in the exercise file folder. We can grab this file using the *File Path* node.
2. With *Geometry.ImportFromSAT*, the geometry is imported into our Dynamo preview as two surfaces.



For this exercise, we want to keep it simple and work with one of the surfaces.

1. Let's select the index of *1* to grab the upper surface. We do this with *List.GetItemAtIndex* node.



```
1;
```

```
0..1..#10;
0..1..#5;
```

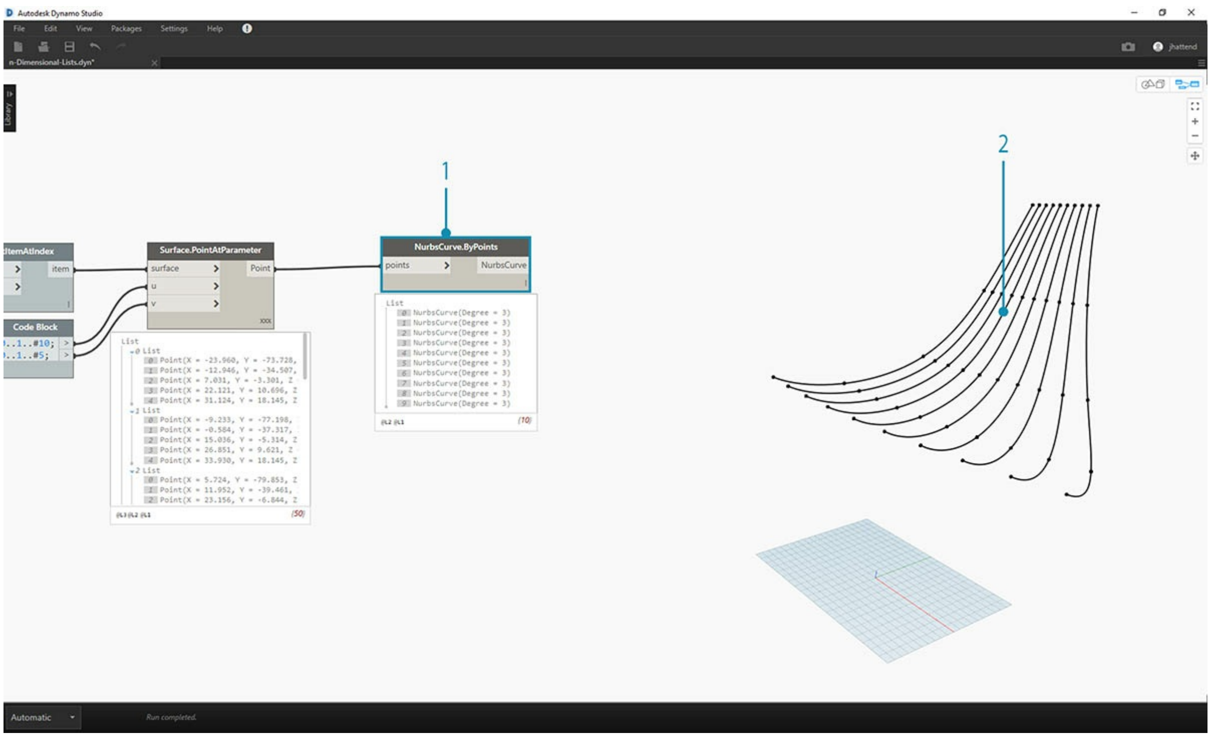
```
List
 0 List
   0 Point(X = -23.960, Y = -73.728,
   1 Point(X = -12.946, Y = -34.507,
   2 Point(X = 7.031, Y = -3.301, Z
   3 Point(X = 22.121, Y = 10.696, Z
   4 Point(X = 31.124, Y = 18.145, Z
 1 List
   0 Point(X = -9.233, Y = -77.198,
   1 Point(X = -0.584, Y = -37.317,
   2 Point(X = 15.036, Y = -5.314, Z
   3 Point(X = 26.851, Y = 9.621, Z
   4 Point(X = 33.930, Y = 18.145, Z
 2 List
   0 Point(X = 5.724, Y = -79.853, Z
   1 Point(X = 11.952, Y = -39.461,
   2 Point(X = 23.156, Y = -6.844, Z
@L3 @L2 @L1                    (50)
```

The next step is to divide the surface into a grid of points.

1. Using *code block*, insert these two lines of code:

   ```
   0..1..#10;
   0..1..#5;
   ```

2. With the *Surface.PointAtParameter*, connect the two code block values to *u* and *v*. Change the *lacing* of this node to *"Cross Product"*.
3. The output reveals the data structure, which is also visible in the Dynamo preview.

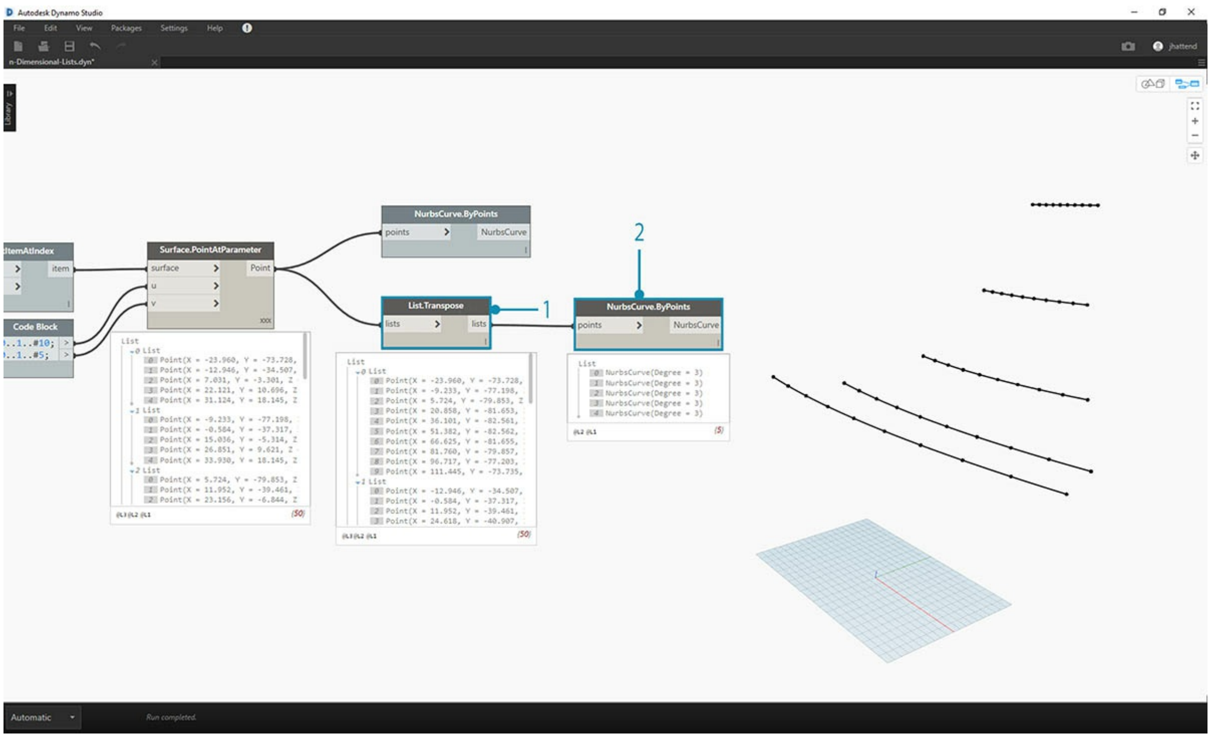


```
..1..#10;
..1..#5;
```

```
List
 0 List
   0 Point(X = -23.960, Y = -73.728,
   1 Point(X = -12.946, Y = -34.507,
   2 Point(X = 7.031, Y = -3.301, Z
   3 Point(X = 22.121, Y = 10.696, Z
   4 Point(X = 31.124, Y = 18.145, Z
 1 List
   0 Point(X = -9.233, Y = -77.198,
   1 Point(X = -0.584, Y = -37.317,
   2 Point(X = 15.036, Y = -5.314, Z
   3 Point(X = 26.851, Y = 9.621, Z
   4 Point(X = 33.930, Y = 18.145, Z
 2 List
   0 Point(X = 5.724, Y = -79.853, Z
   1 Point(X = 11.952, Y = -39.461,
   2 Point(X = 23.156, Y = -6.844, Z
@L3 @L2 @L1                    (50)
```

```
List
  0 NurbsCurve(Degree = 3)
  1 NurbsCurve(Degree = 3)
  2 NurbsCurve(Degree = 3)
  3 NurbsCurve(Degree = 3)
  4 NurbsCurve(Degree = 3)
  5 NurbsCurve(Degree = 3)
  6 NurbsCurve(Degree = 3)
  7 NurbsCurve(Degree = 3)
  8 NurbsCurve(Degree = 3)
  9 NurbsCurve(Degree = 3)
@L2 @L1                    (10)
```

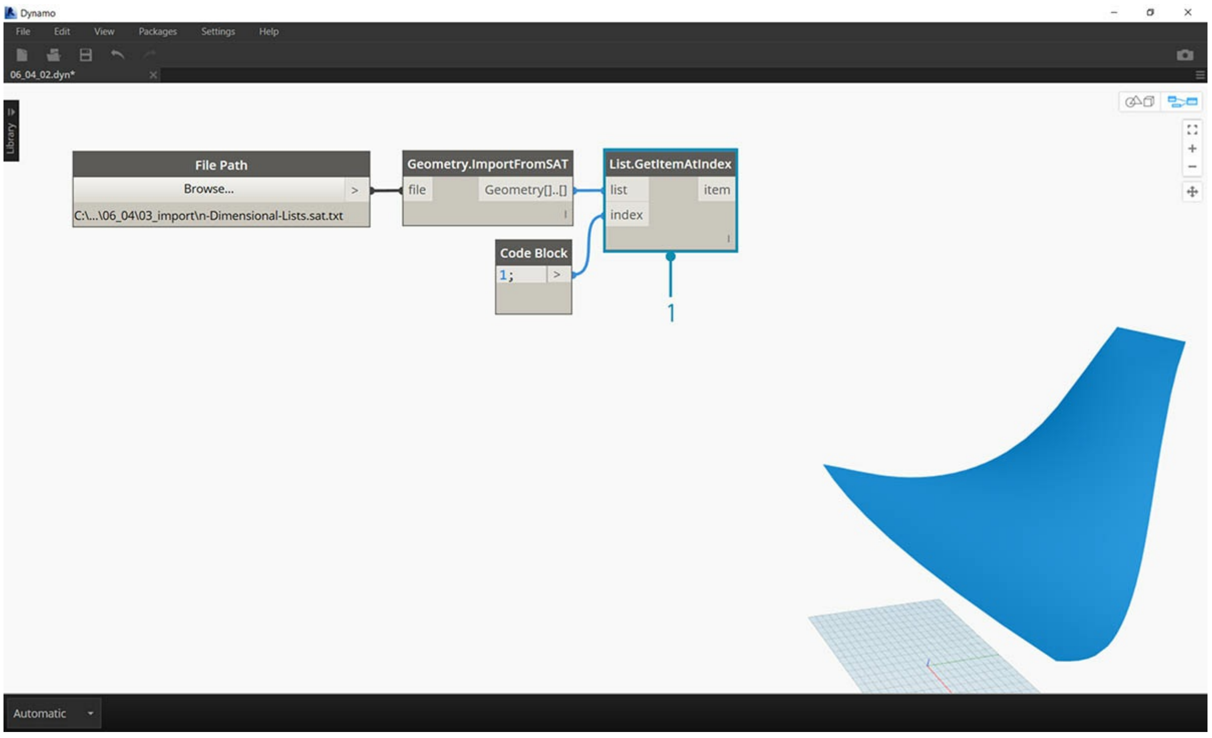
1. To get a look at how the data structure is organized, let's connect a *NurbsCurve.ByPoints* to the output of *Surface.PointAtParameter*.
2. Notice we have ten curves running vertically along the surface.



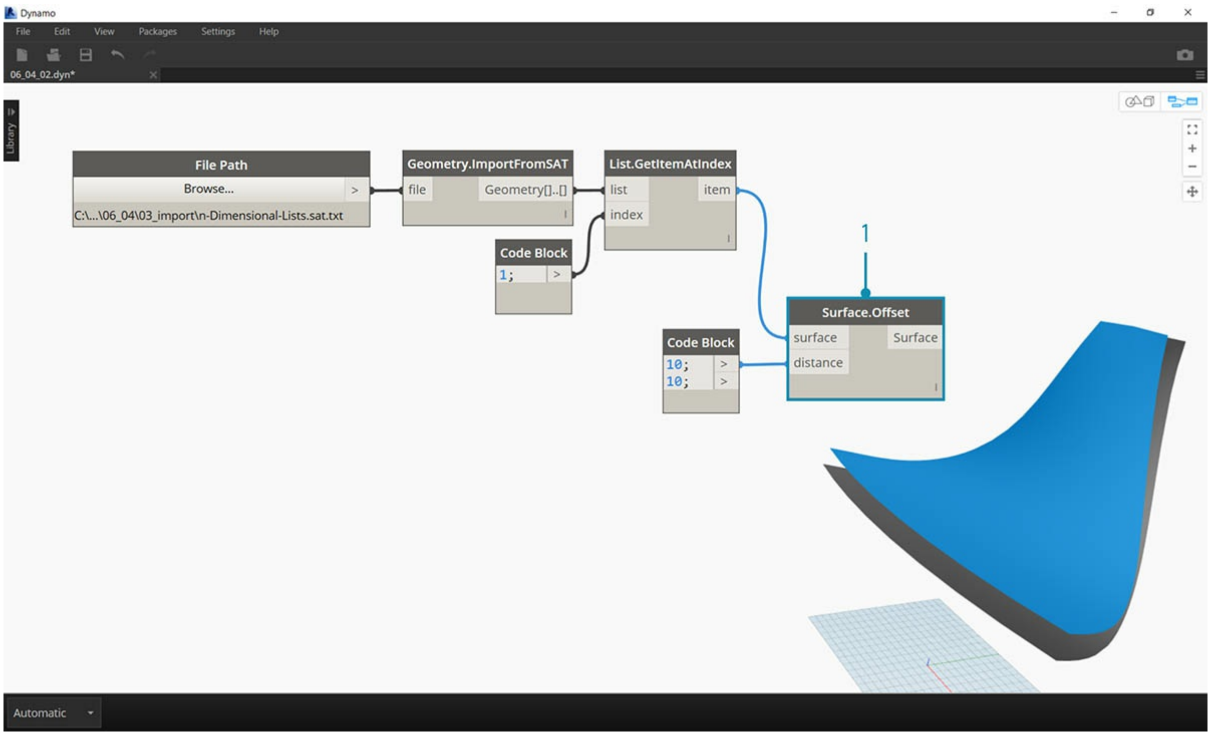
1. A basic *List.Transpose* will flip the columns and rows of a list of lists.
2. Connecting the output of *List.Transpose* to *NurbsCurve.ByPoints*, we now get five curves running horizontally across the surface.

Exercise - 2D Lists - Advanced

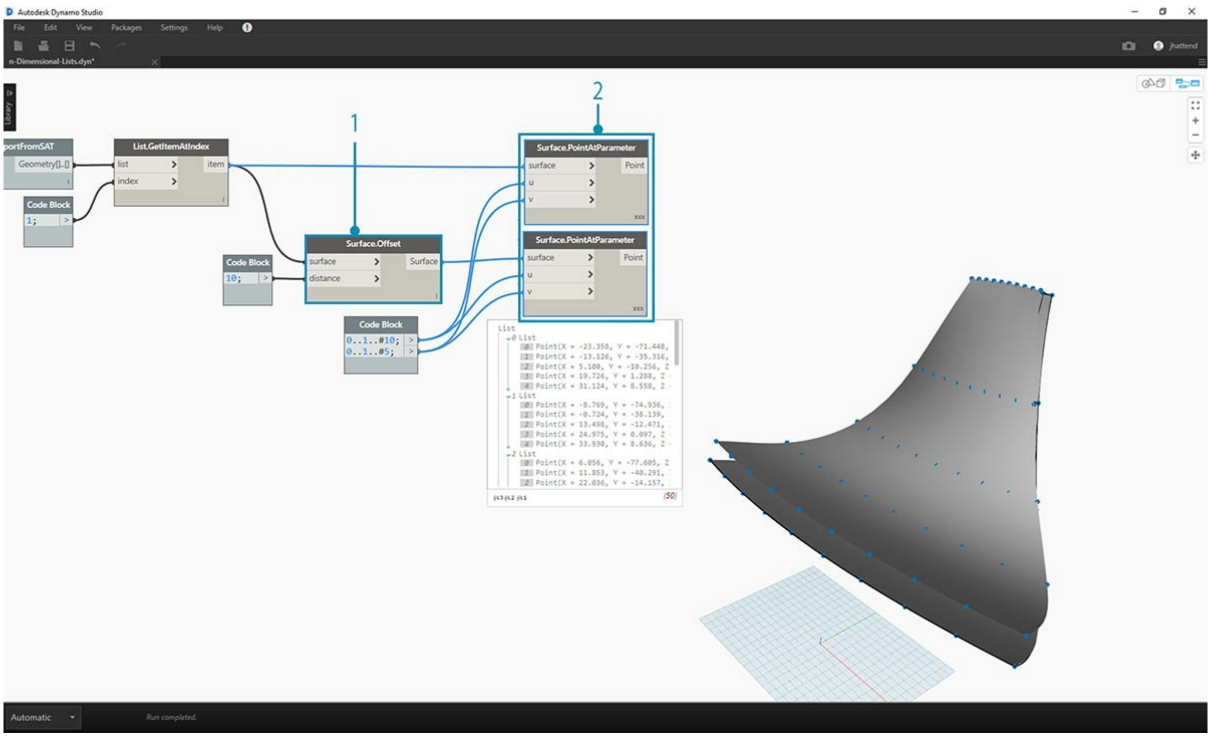
Let's increase the complexity. Suppose we wanted to perform an operation on the curves created from the previous exercise. Perhaps we want to relate these curves to another surface and loft between them. This requires more attention to data structure, but the underlying logic is the same.



1. Begin with a step from the previous exercise, isolating the upper surface of the imported geometry with the *List.GetItemAtIndex* node.



1. Using *Surface.Offset*, offset the surface by a value of *10*.

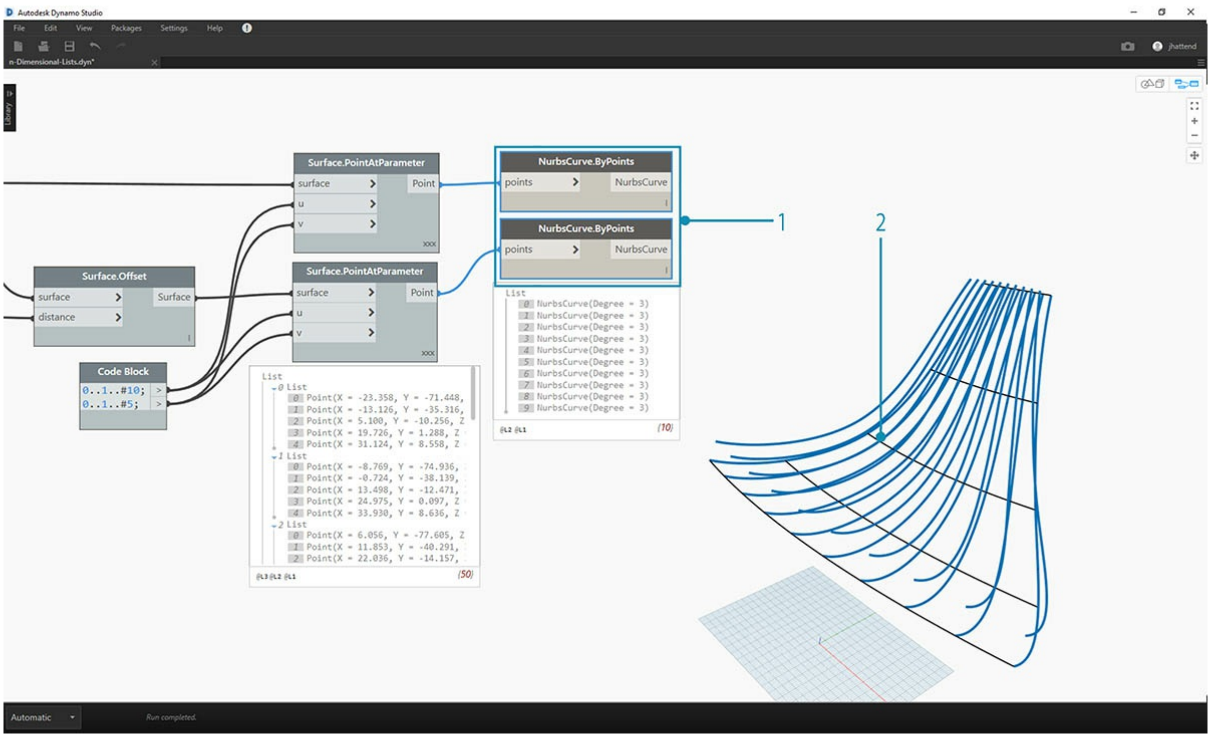


```
List
 0 List
   0 Point(X = -23.358, Y = -71.448,
   1 Point(X = -13.126, Y = -35.316,
   2 Point(X = 5.100, Y = -10.256, Z
   3 Point(X = 19.726, Y = 1.288, Z
   4 Point(X = 31.124, Y = 8.558, Z
 1 List
   0 Point(X = -8.769, Y = -74.936,
   1 Point(X = -0.724, Y = -38.139,
   2 Point(X = 13.498, Y = -12.471,
   3 Point(X = 24.975, Y = 0.097, Z
   4 Point(X = 33.930, Y = 8.636, Z
 2 List
   0 Point(X = 6.056, Y = -77.605, Z
   1 Point(X = 11.853, Y = -40.291,
   2 Point(X = 22.036, Y = -14.157,
@L3 @L2 @L1                      {50}
```

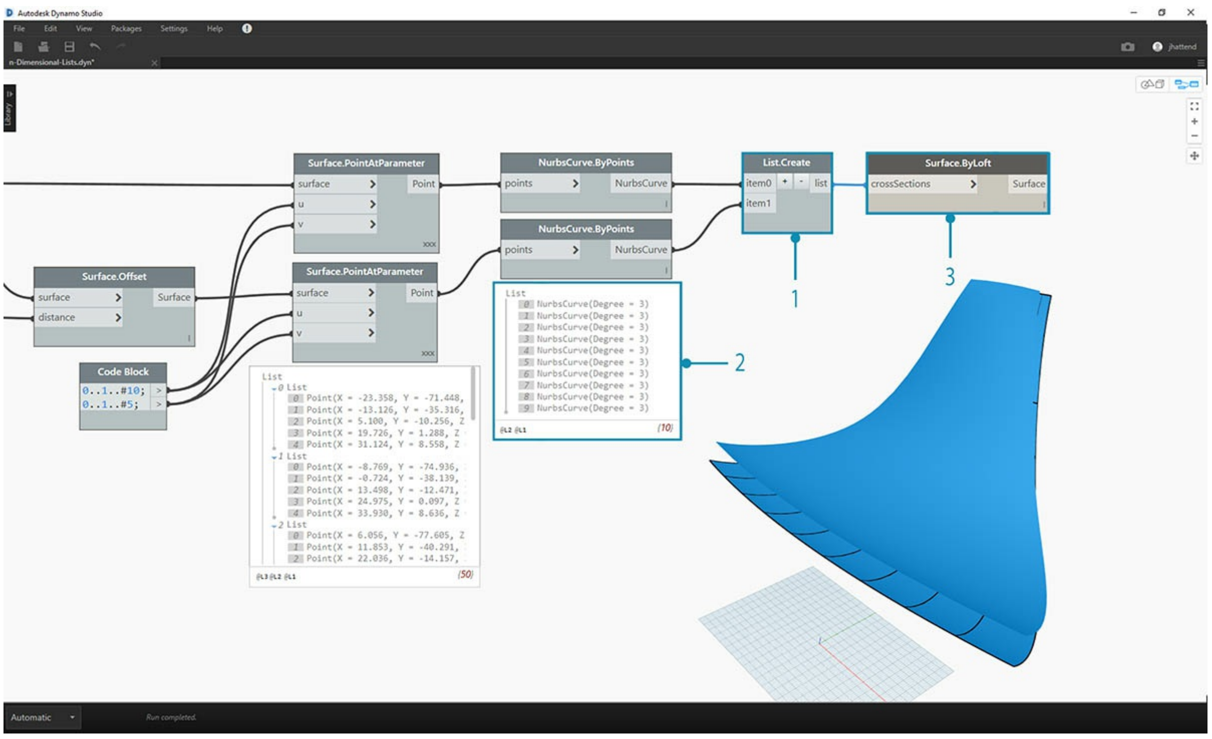
1. In the same manner as the previous exercise, define a *code block* with these two lines of code:

```
0..1..#10;
0..1..#5;
```

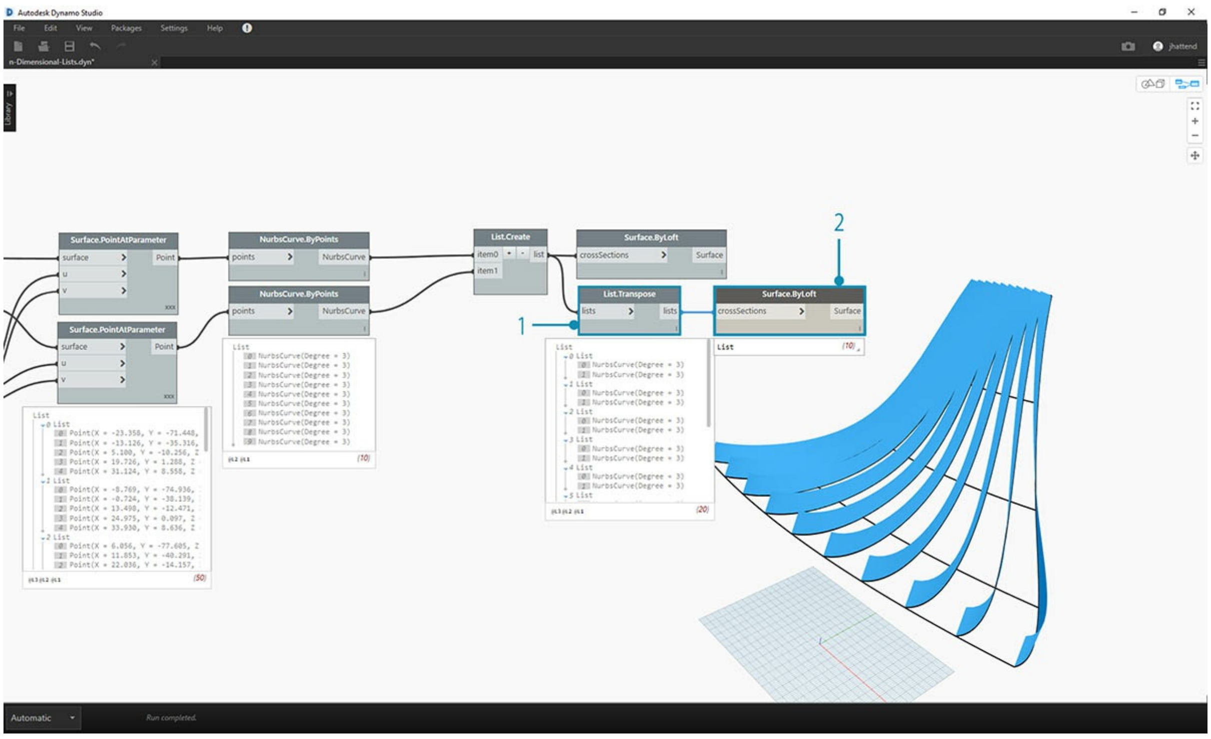
2. Connect these outputs to two *Surface.PointAtParameter* nodes, each with *lacing* set to *"Cross Product"*. One of these nodes is connected to the original surface, while the other is connected to the offset surface.



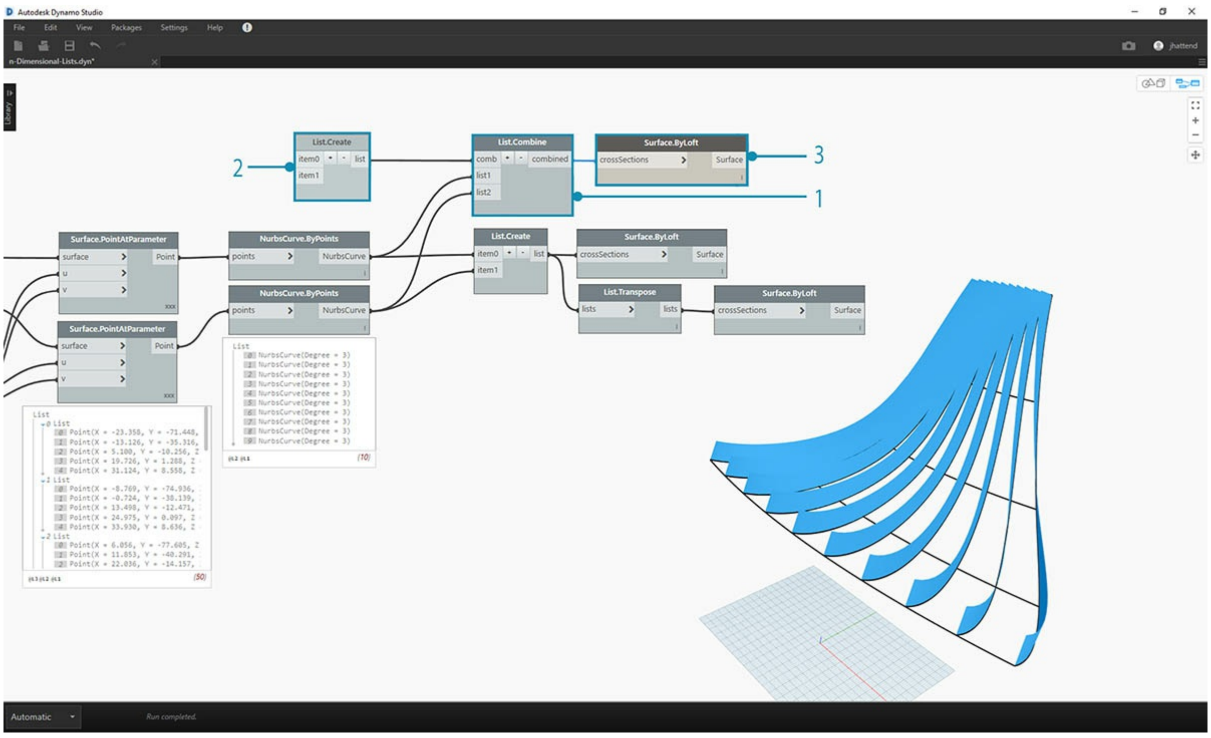
1. As in the previous exercise, connect the outputs to two *NurbsCurve.ByPoints* nodes.
2. Our Dynamo preview shows two curves, corresponding to two surfaces.



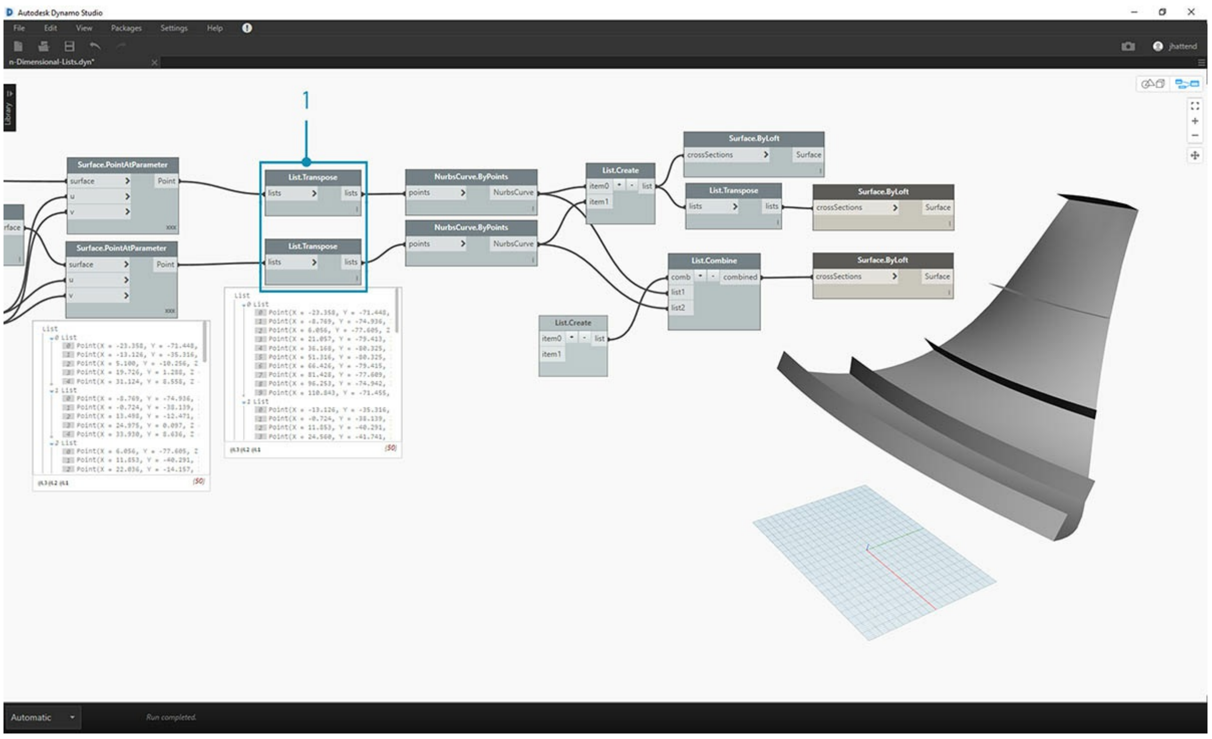
1. By using *List.Create*, we can combine the two sets of curves into one list of lists.
2. Notice from the output, we have two lists with ten items each, representing each connect set of nurbs curves.
3. By performing a *Surface.ByLoft*, we can visually make sense of this data structure. The node lofts all of the curves in each sublist.



1. By using *List.Transpose*, remember, we are flipping all of the columns and rows. This node will transfer two lists of ten curves into ten lists of two curves. We now have each nurbs curve related to the neighboring curve on the other surface.
2. Using *Surface.ByLoft*, we arrive at a ribbed structure.



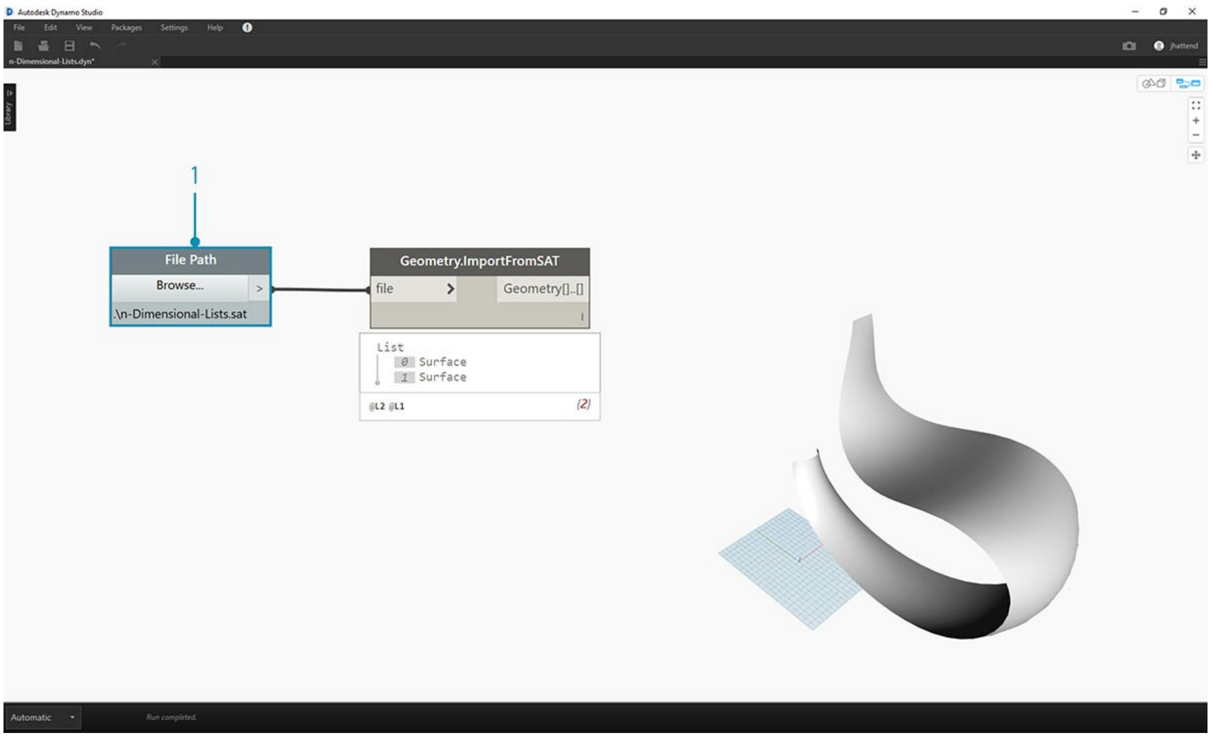
1. An alternative to *List.Transpose* uses *List.Combine*. This will operate a *"combinator"* on each sublist.
2. In this case, we're using *List.Create* as the *"combinator"*, which will create a list of each item in the sublists.
3. Using the *Surface.ByLoft* node, we get the same surfaces as in the previous step. Transpose is easier to use in this case, but when the data structure becomes even more complex, *List.Combine* is more reliable.



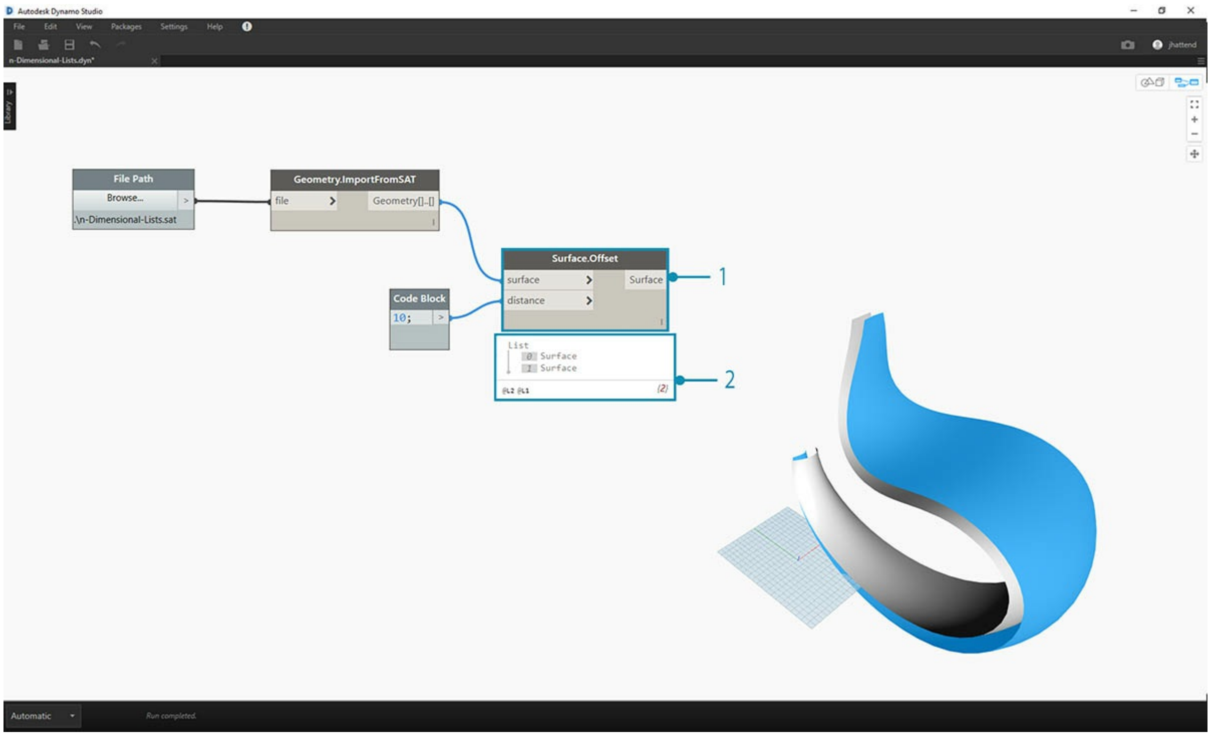
1. Stepping back a few steps, if we want to switch the orientation of the curves in the ribbed structure, we want to use a List.Transpose before connect to *NurbsCurve.ByPoints*. This will flip the columns and rows, giving us 5 horizontal ribs.

Exercise - 3D Lists

Now, we're going to go even one step further. In this exercise, we'll work with both imported surfaces, creating a complex data hierarchy. Still, our aim is to complete the same operation with the same underlying logic.

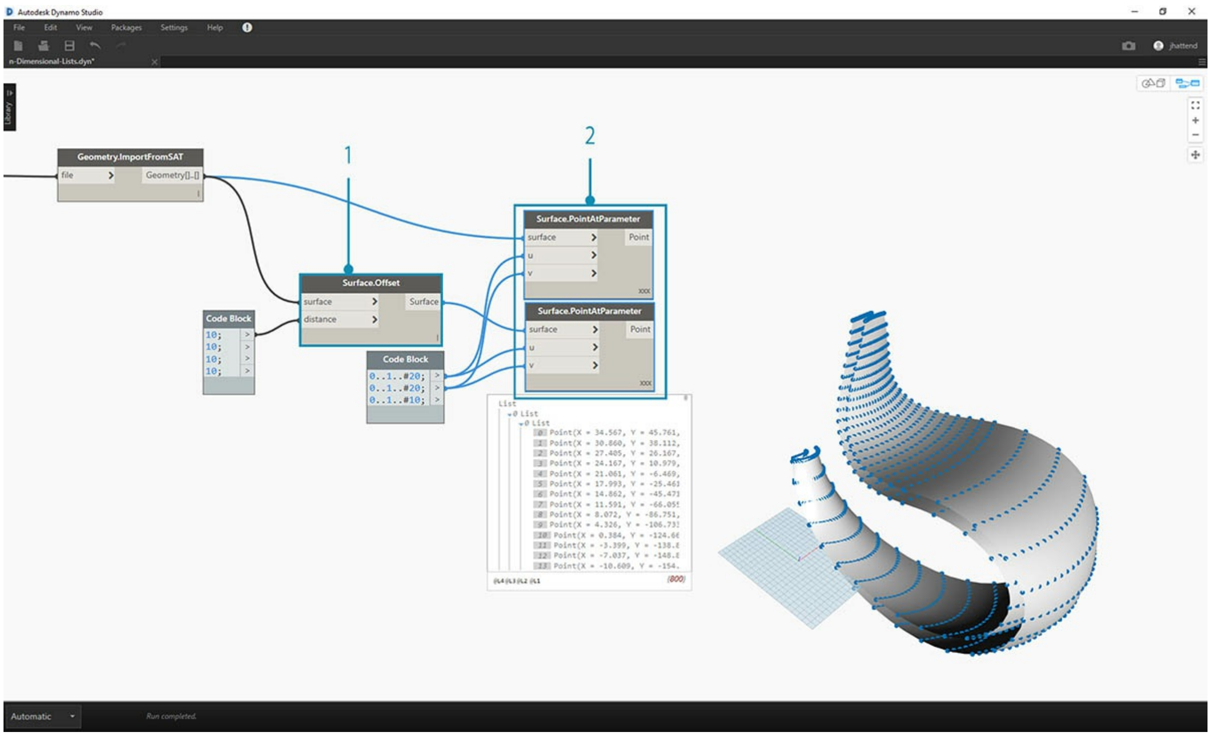


1. Begin with the imported file from previous exercise.



```
List
  0 Surface
  1 Surface
@L2 @L1                (2)
```

1. As in the previous exercise, use the *Surface.Offset* node to offset by a value of *10*.
2. Notice from the output, that we've created two surfaces with the offset node.

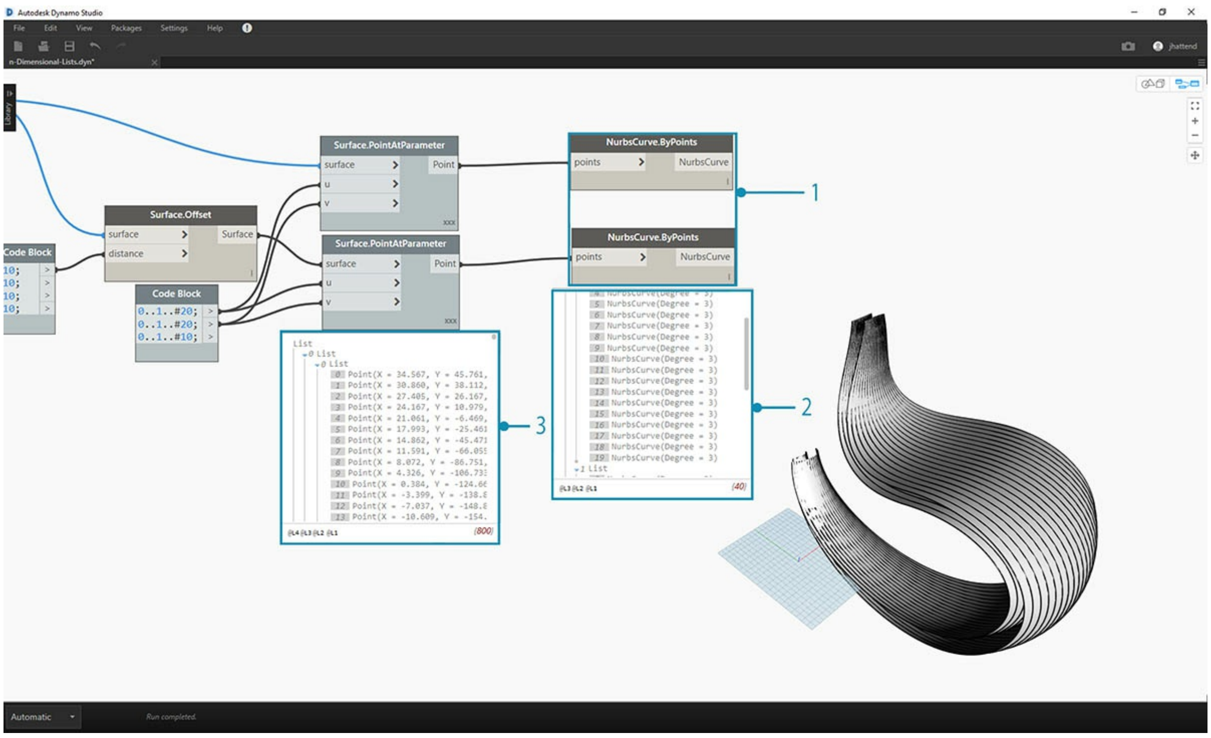


```
List
  0 List
    0 List
      0 Point(X = 34.567, Y = 45.761,
      1 Point(X = 30.860, Y = 38.112,
      2 Point(X = 27.405, Y = 26.167,
      3 Point(X = 24.167, Y = 10.979,
      4 Point(X = 21.061, Y = -6.469,
      5 Point(X = 17.993, Y = -25.461
      6 Point(X = 14.862, Y = -45.471
      7 Point(X = 11.591, Y = -66.055
      8 Point(X = 8.072, Y = -86.751,
      9 Point(X = 4.326, Y = -106.731
      10 Point(X = 0.384, Y = -124.66
      11 Point(X = -3.399, Y = -138.8
      12 Point(X = -7.037, Y = -148.8
      13 Point(X = -10.609, Y = -154.
@L4 @L3 @L2 @L1                (800)
```

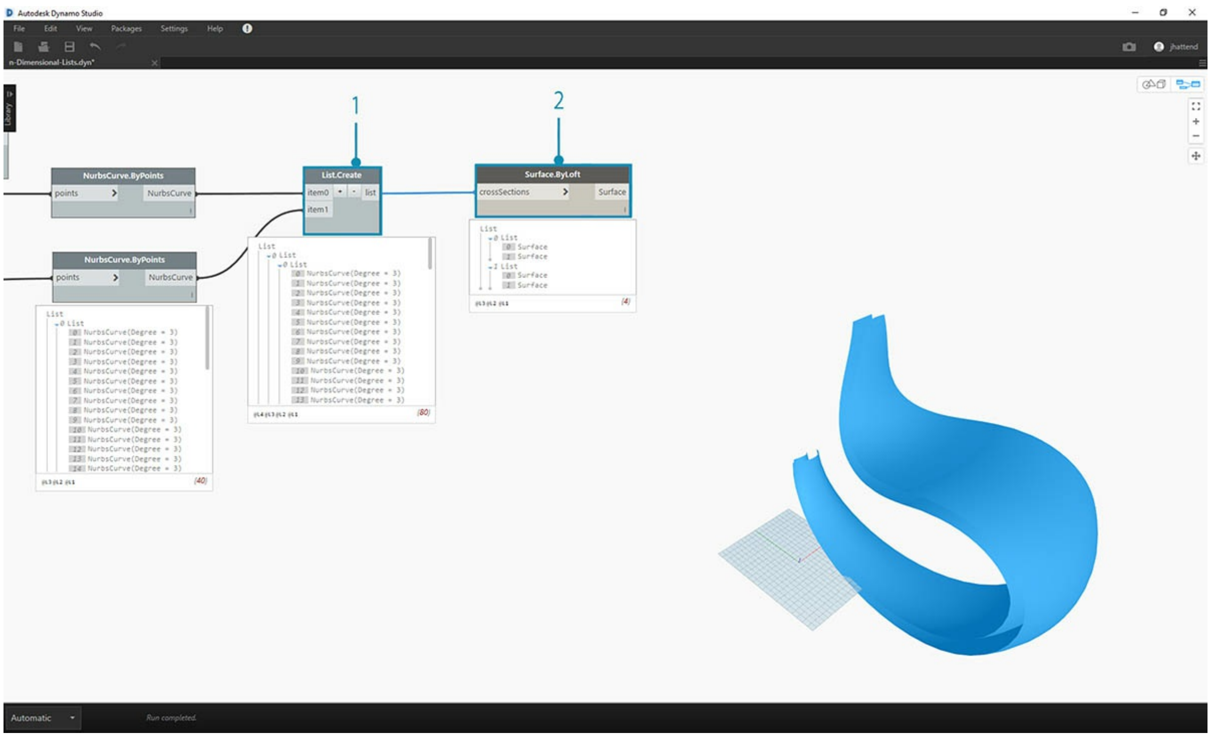
1. In the same manner as the previous exercise, define a code block with these two lines of code:

   ```
   0..1..#20;
   0..1..#10;
   ```

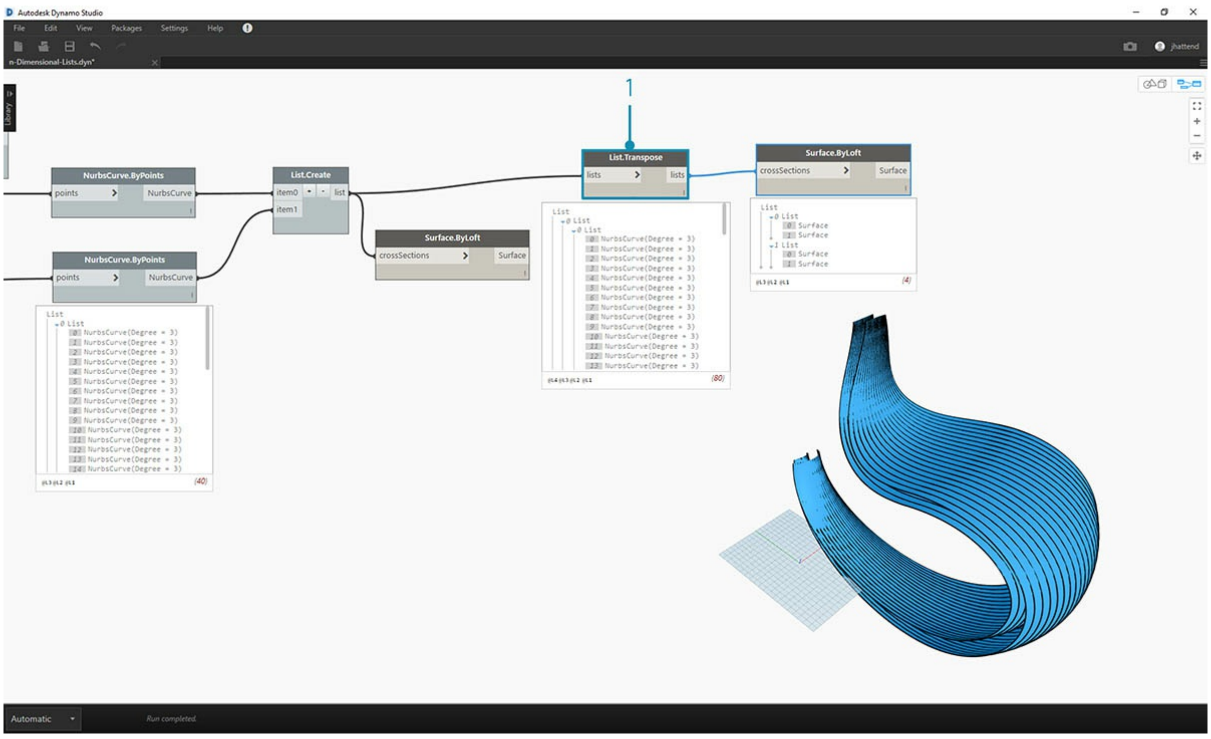
2. Connect these outputs to two *Surface.PointAtParameter* nodes, each with lacing set to *"Cross Product"*. One of these nodes is connected to the original surfaces, while the other is connected to the offset surfaces.



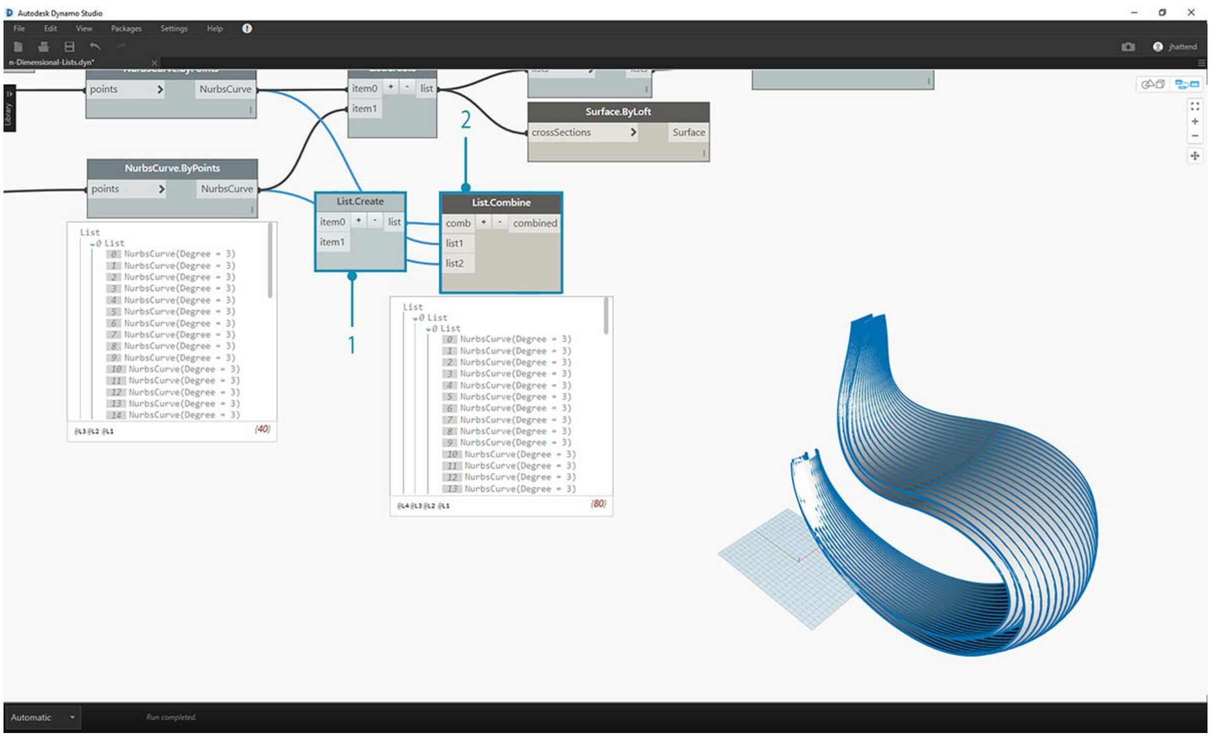
1. As in the previous exercise, connect the outputs to two *NurbsCurve.ByPoints* nodes.
2. Looking at the output of the *NurbsCurve.ByPoints*, notice that this is a list of two lists, which is more complex than the previous exercise. The data is categorized by the underlying surface, so we've added another tier to the data structured.
3. Notice that things become more complex in the *Surface.PointAtParameter* node. In this case we have a list of lists of lists.



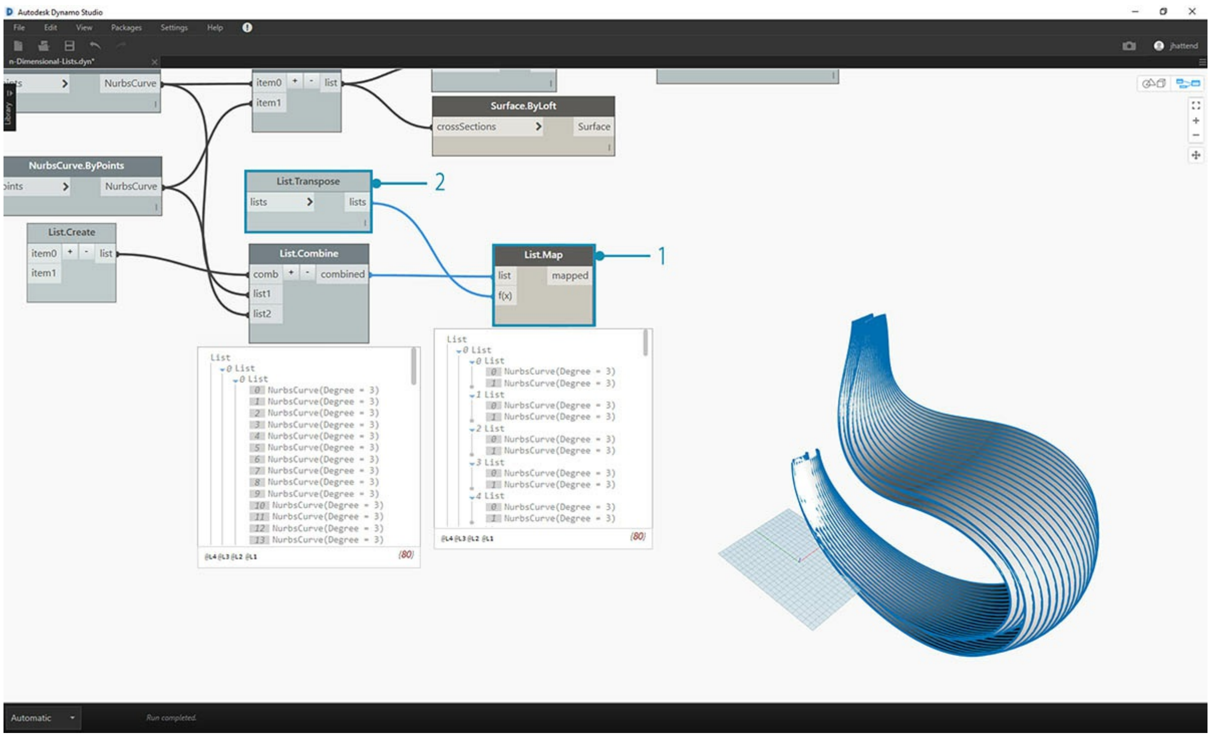
1. Using the *List.Create* node, we merge the nurbs curves into one data structure, creating a list of lists of lists.
2. By connecting a *Surface.ByLoft* node, we get a version of the original surfaces, as they each remain in their own list as created from the original data structure.



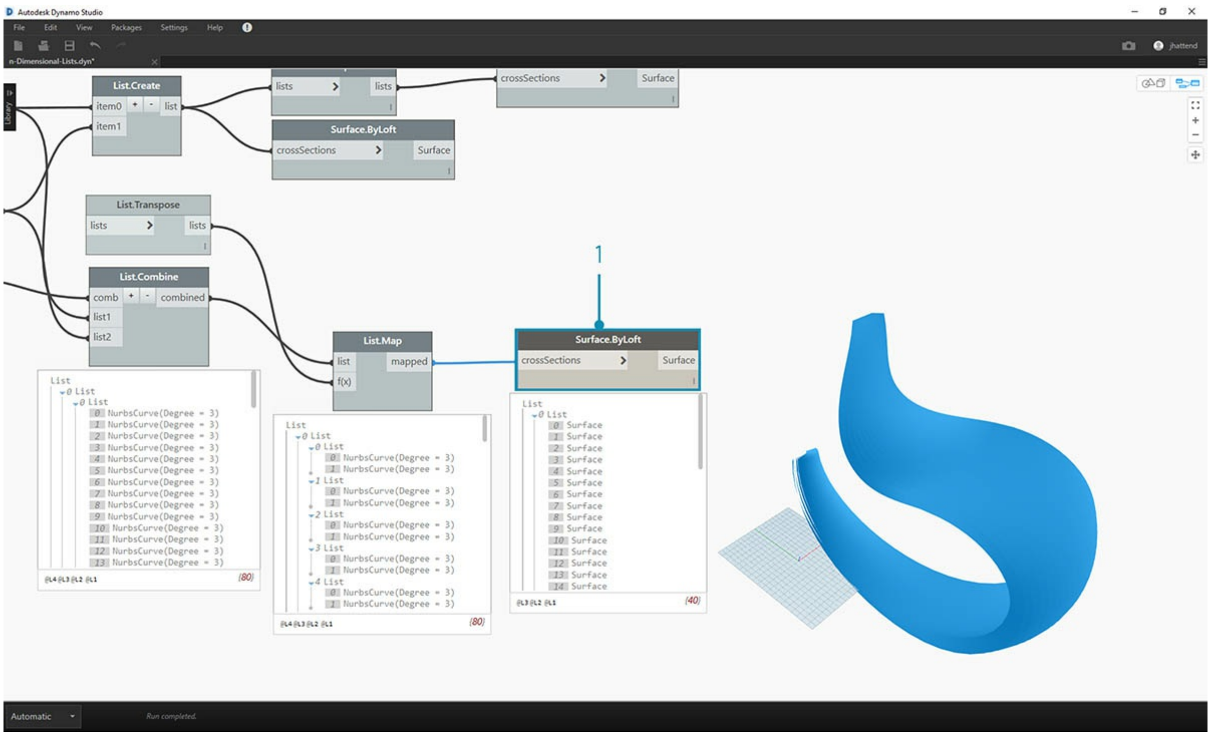
1. In the previous exercise, we were able to use a *List.Transpose* to create a ribbed structure. This won't work here. A transpose should be used on a two-dimensional list, and since we have a three-dimensional list, an operation of "flipping columns and rows" won't work as easily. Remember, lists are objects, so *List.Transpose* will flip our lists with out sublists, but won't flip the nurbs curves one list further down in the hierarchy.



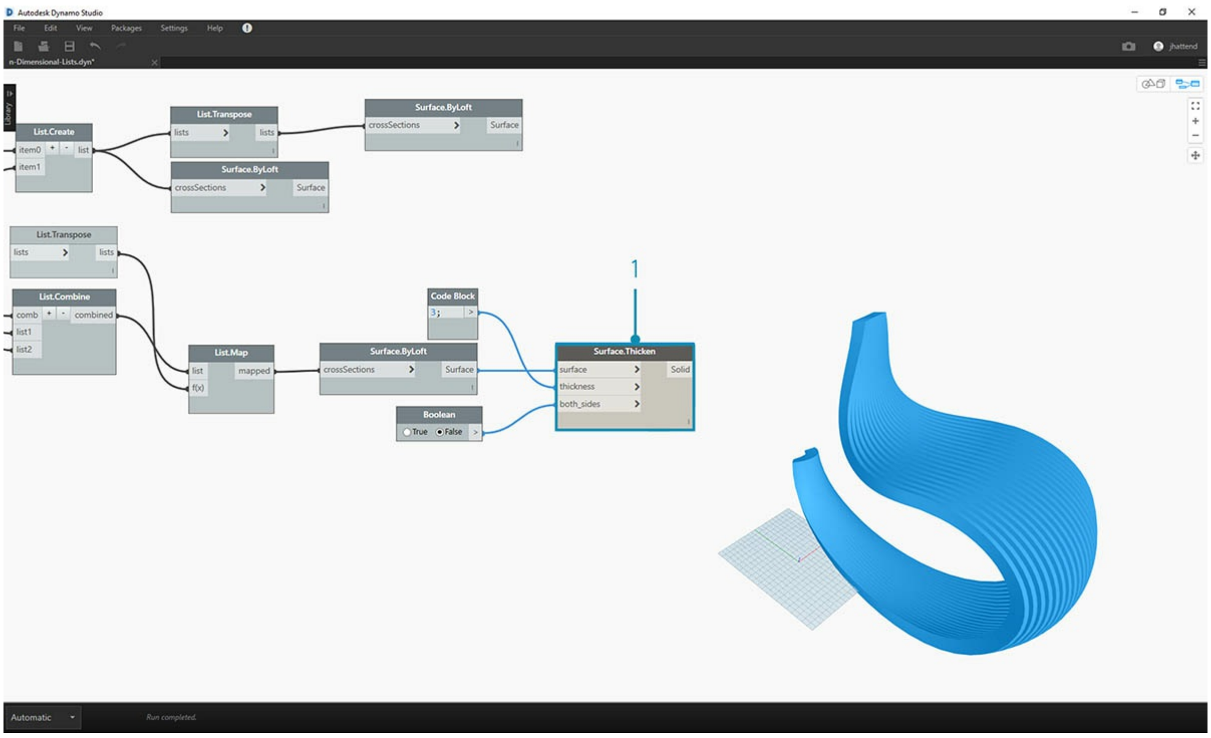
1. *List.Combine* will work better for us here. We want to use *List.Map* and *List.Combine* nodes when we get to more complex data structures.
2. Using *List.Create* as the *"combinator"*, we create a data structure that will work better for us.



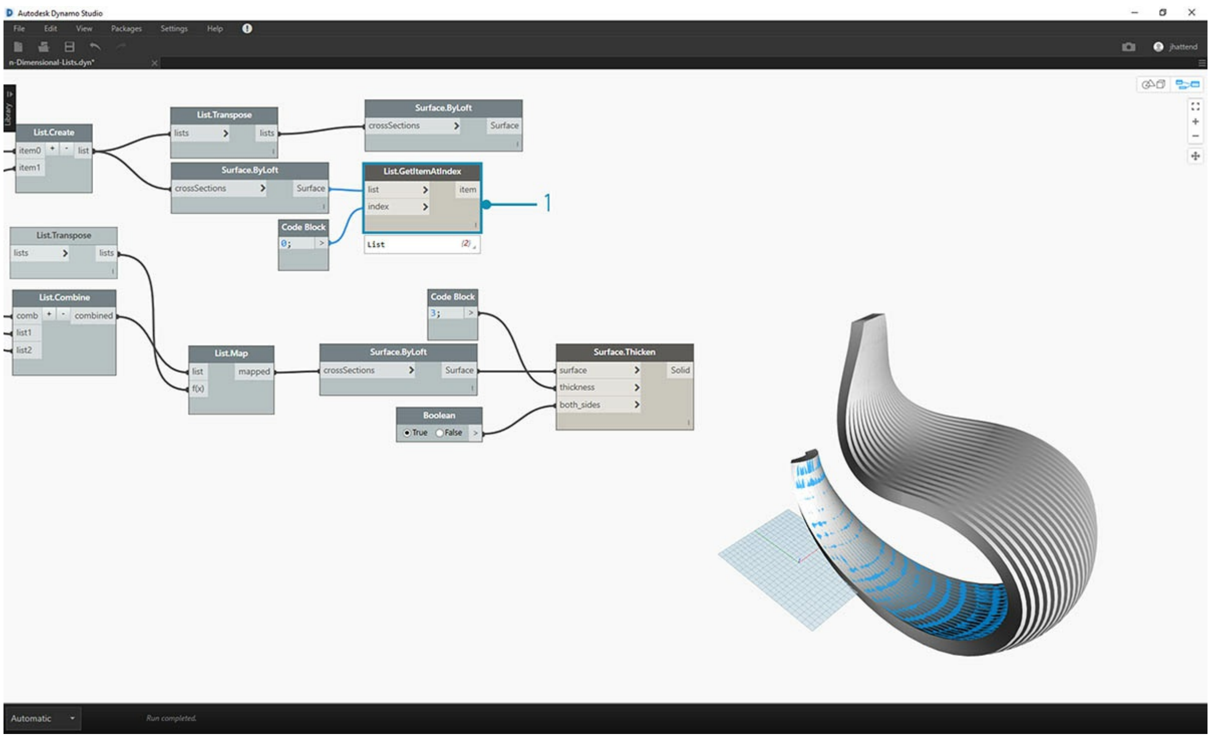
1. The data structure still needs to be transposed at one step down on the hierarchy. To do this we'll use *List.Map*. This is working like *List.Combine*, except with one input list, rather than two or more.
2. The function we'll apply to *List.Map* is *List.Transpose*, which will flip the columns and rows of the sublists within our main list.



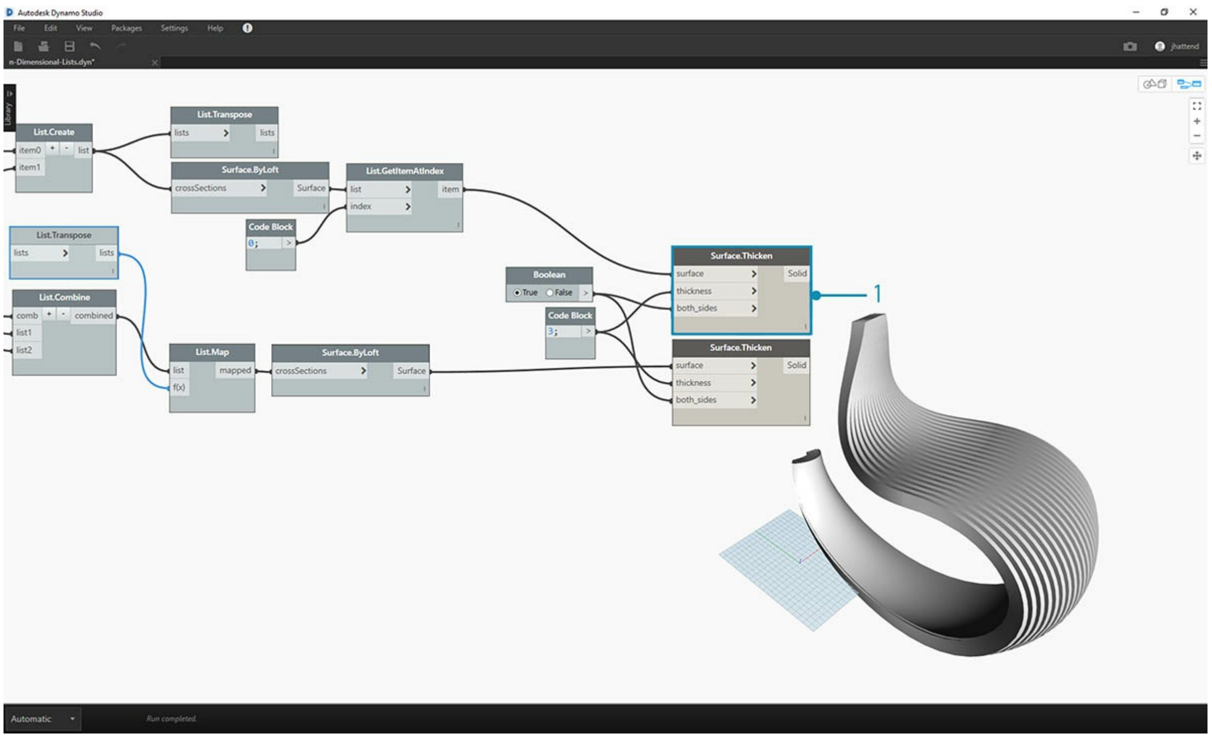
1. Finally, we can loft the nurbs curves together with a proper data hierarchy, giving us a ribbed structure.



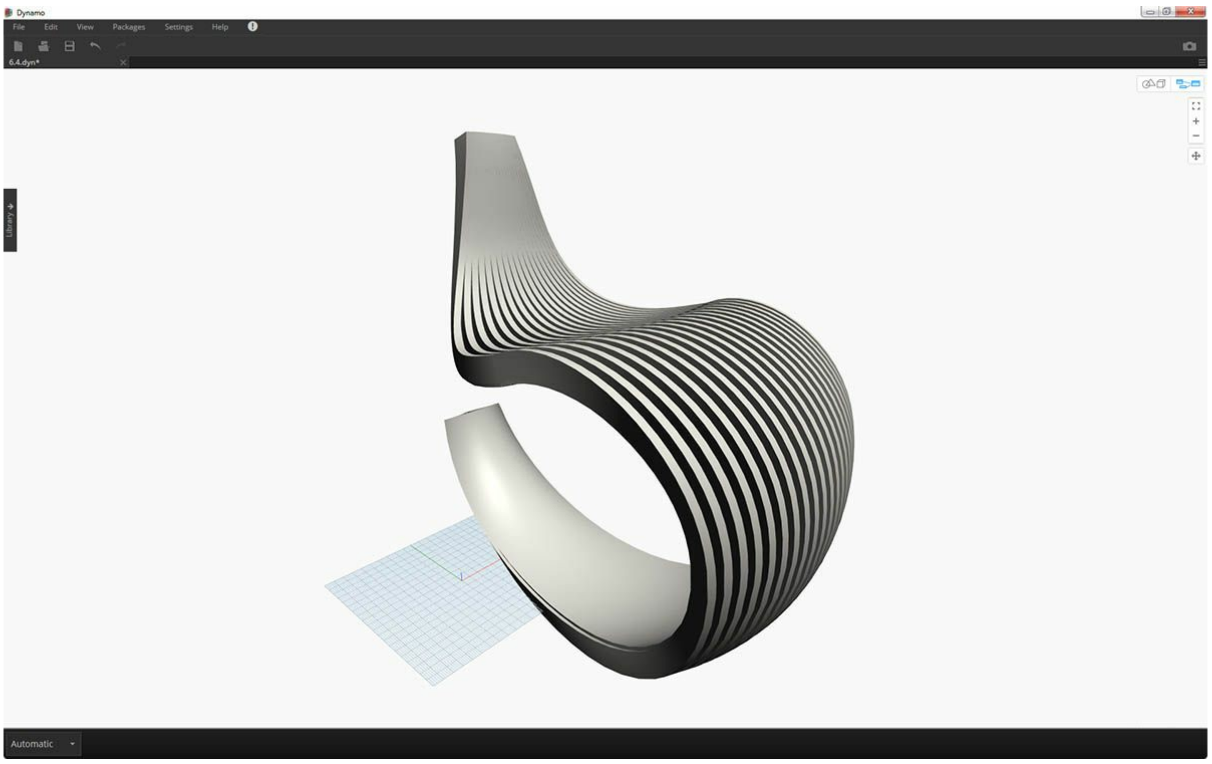
1. Let's add some depth to the geometry with a *Surface.Thicken* node.



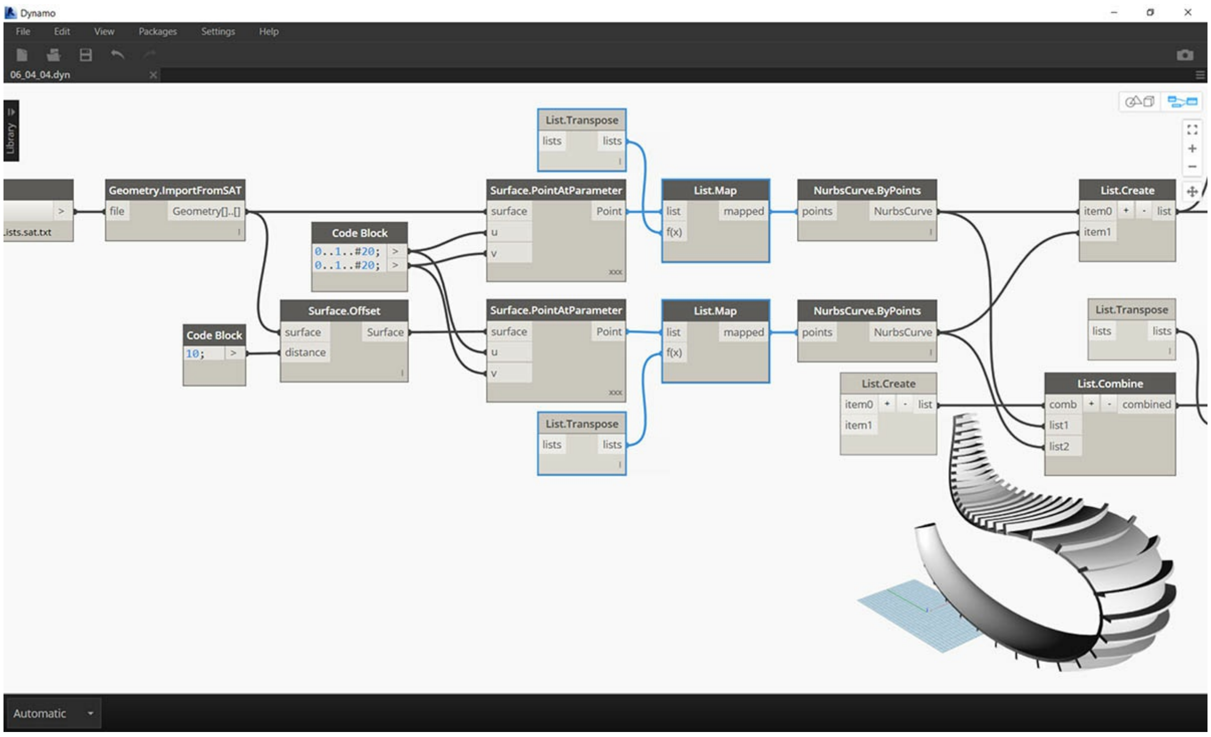
1. It'll be nice to add a surface backing two this structure, so we'll use *List.GetItemAtIndex* to select the back surface from the lofted surfaces from the previous steps.



1. And thickening these selected surfaces, our articulation is complete.

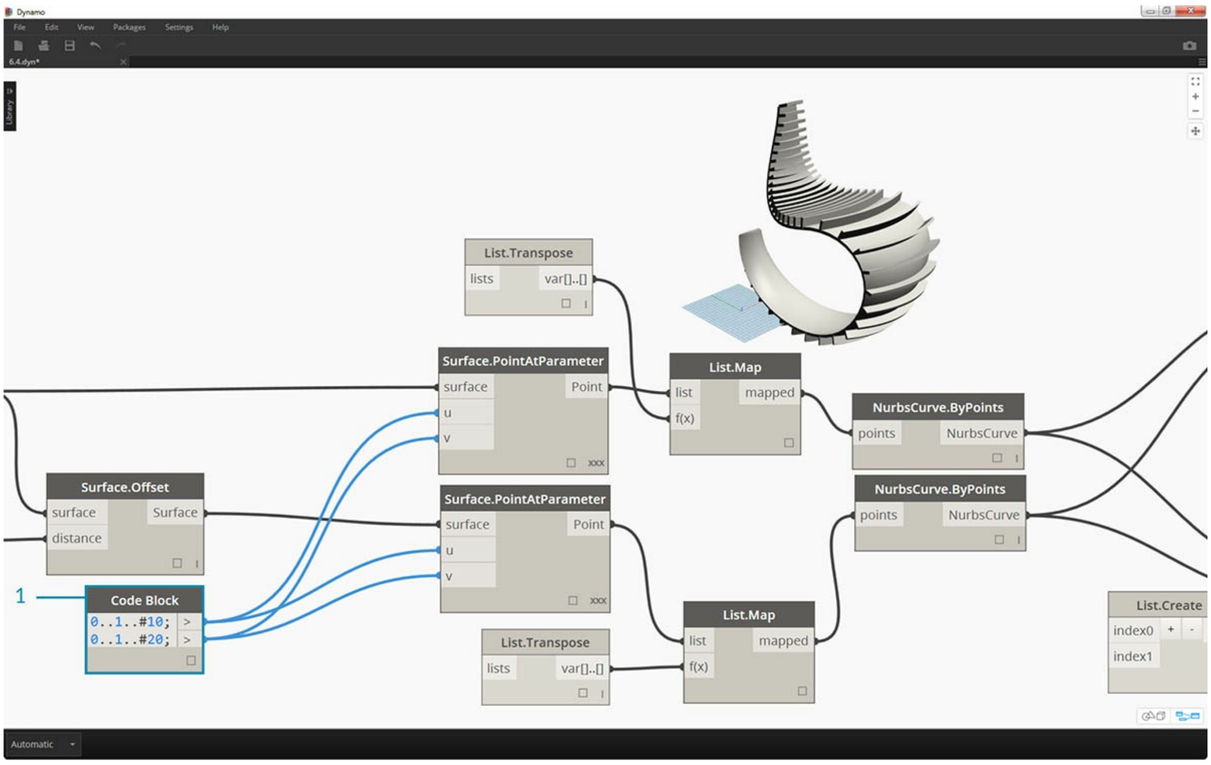


Not the most comfortable rocking chair ever, but it's got a lot of data going on.



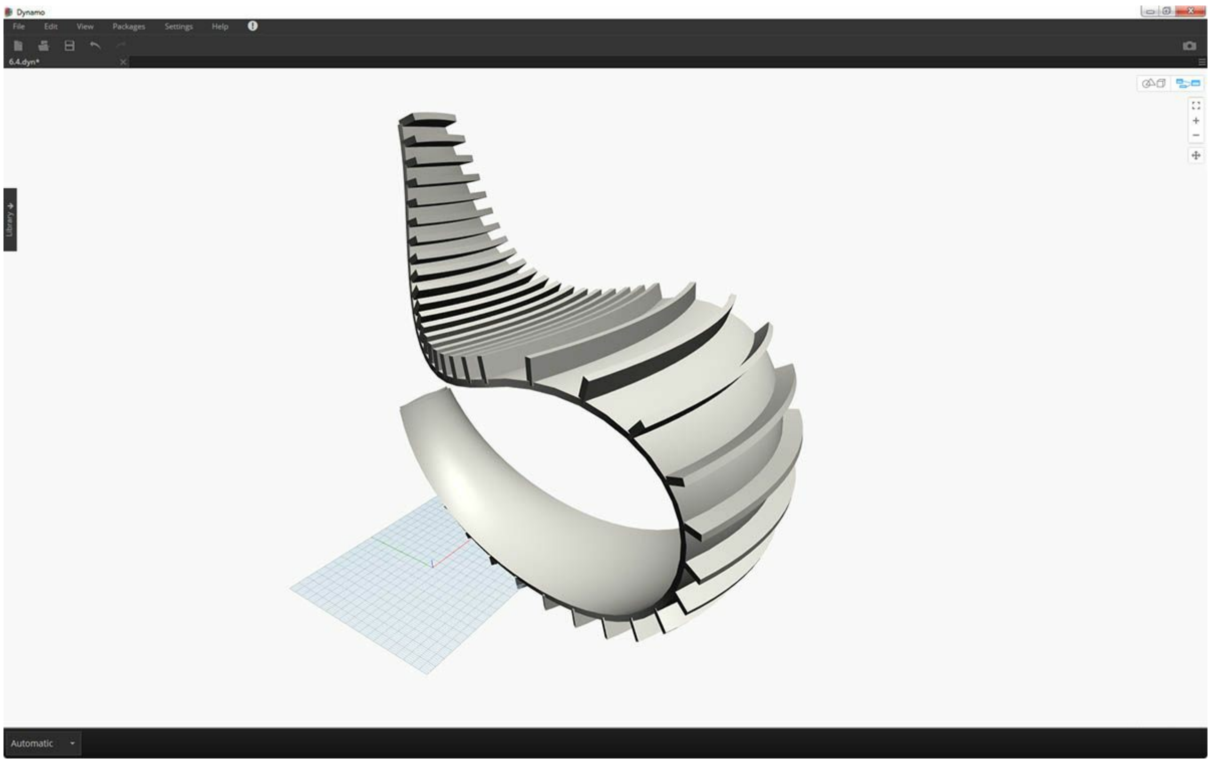
Last step, let's reverse the direction of the striated members. As we used transpose in the previous exercise, we'll do something similar here.

1. Since we have one more tier to the hierarchy, so we need to use *List.Map* with a *List.Tranpose* function to change the direction of our nurbs curves.



1. We may want to increase the number of treads, so we can change the code block to

```
0..1..#20;
0..1..#10;
```

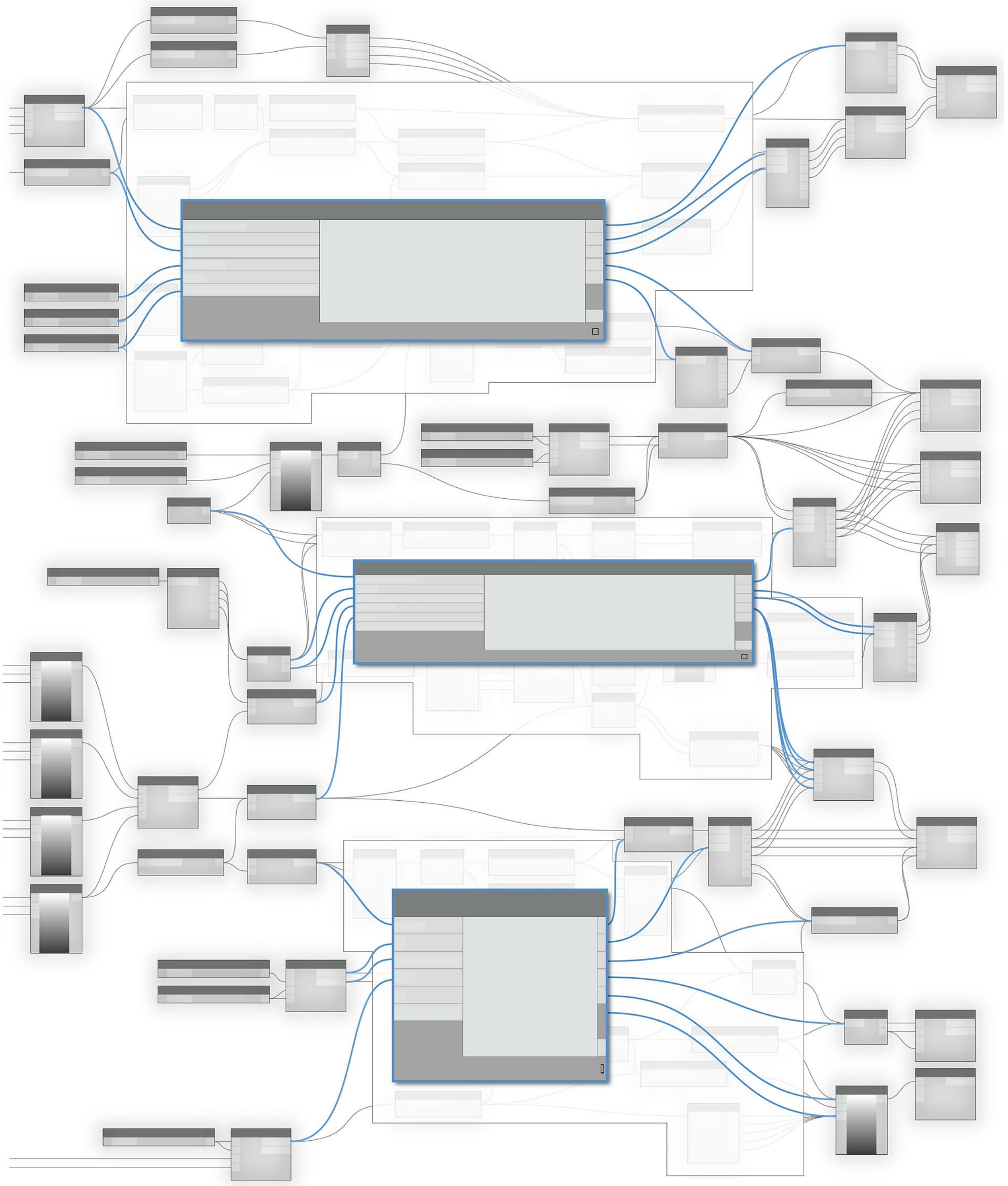


The first version of the rocking chair was sleek, so our second model offers an off-road, sport-utility version of recumbency.

Code Blocks and DesignScript

Code Blocks and DesignScript

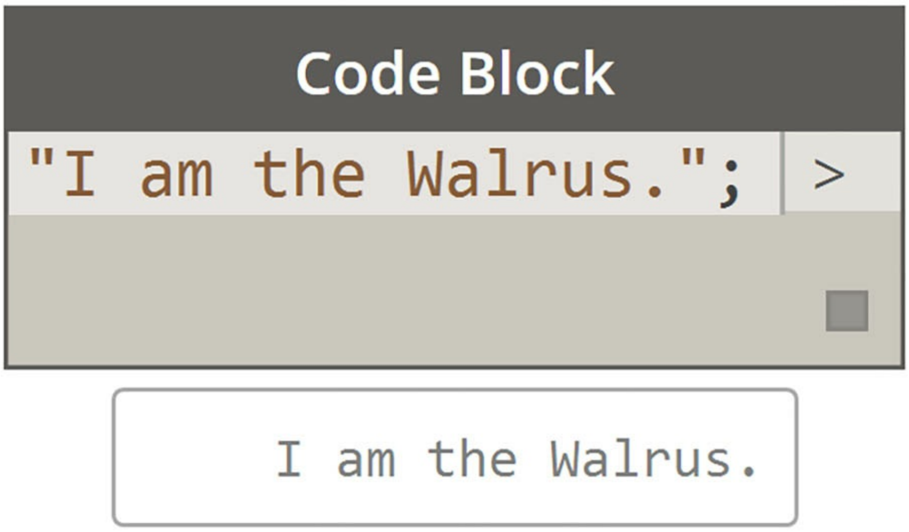
The code block is a unique feature in Dynamo that dynamically links a visual programming environment with a text-based one. The code-block has access to all of the Dynamo nodes and can define an entire graph in one node. Read this chapter closely, as the code block is a fundamental building block of Dynamo.



What's a Code Block

What's a Code Block?

Code blocks are a window deep into DesignScript, the programming language at the heart of Dynamo. Built from scratch to support exploratory design workflows, DesignScript is a readable and concise language that offers both immediate feedback to small bits of code and also scales to large and complex interactions. DesignScript also forms the backbone of the engine that drives most aspects of Dynamo “under the hood”. Because nearly all of the functionality found in Dynamo nodes and interactions have a one-to-one relationship with the scripting language, there are unique opportunities to move between node-based interactions and scripting in a fluid way.



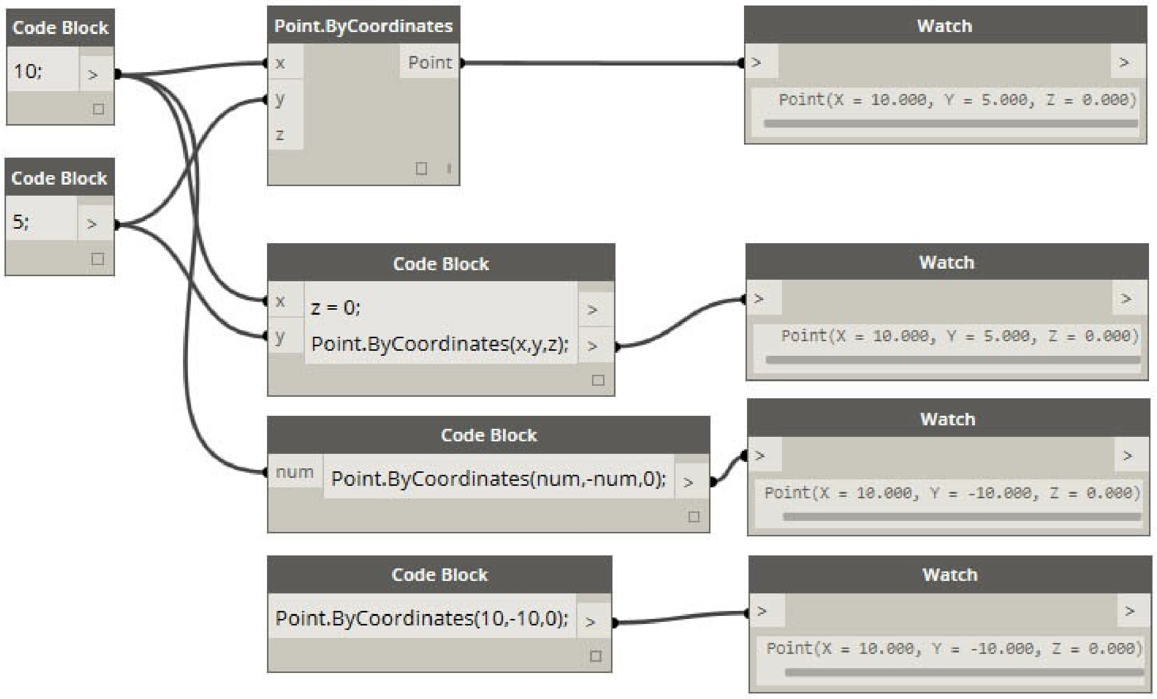
```
"I am the Walrus.";
```

For beginners, nodes can be automatically converted to text syntax to aid in learning DesignScript or simply to reduce the size of larger sections of graphs. This is done using a process called "Node to Code", which is outlined in more detail in the DesignScript Syntax section. More experienced users can use Code Blocks to create customized mashups of existing functionality and user authored relationships using many standard coding paradigms. In between the beginner and advanced user, there are a huge number of shortcuts and code snippets that will accelerate your designs. While the term 'code block' may be a little intimidating to non-programmers, it is both easy to use and robust. A beginner can use the code block efficiently with minimal coding, and an advanced user can define scripted definitions to be recalled elsewhere in a Dynamo definition.

Code Block: A brief overview

In short, code blocks are a text-scripting interface within a visual-scripting environment. They can be used as numbers, strings, formulas, and other data types. The code block is designed for Dynamo, so one can define arbitrary variables in the code block, and those variables are automatically added to the inputs of the node:

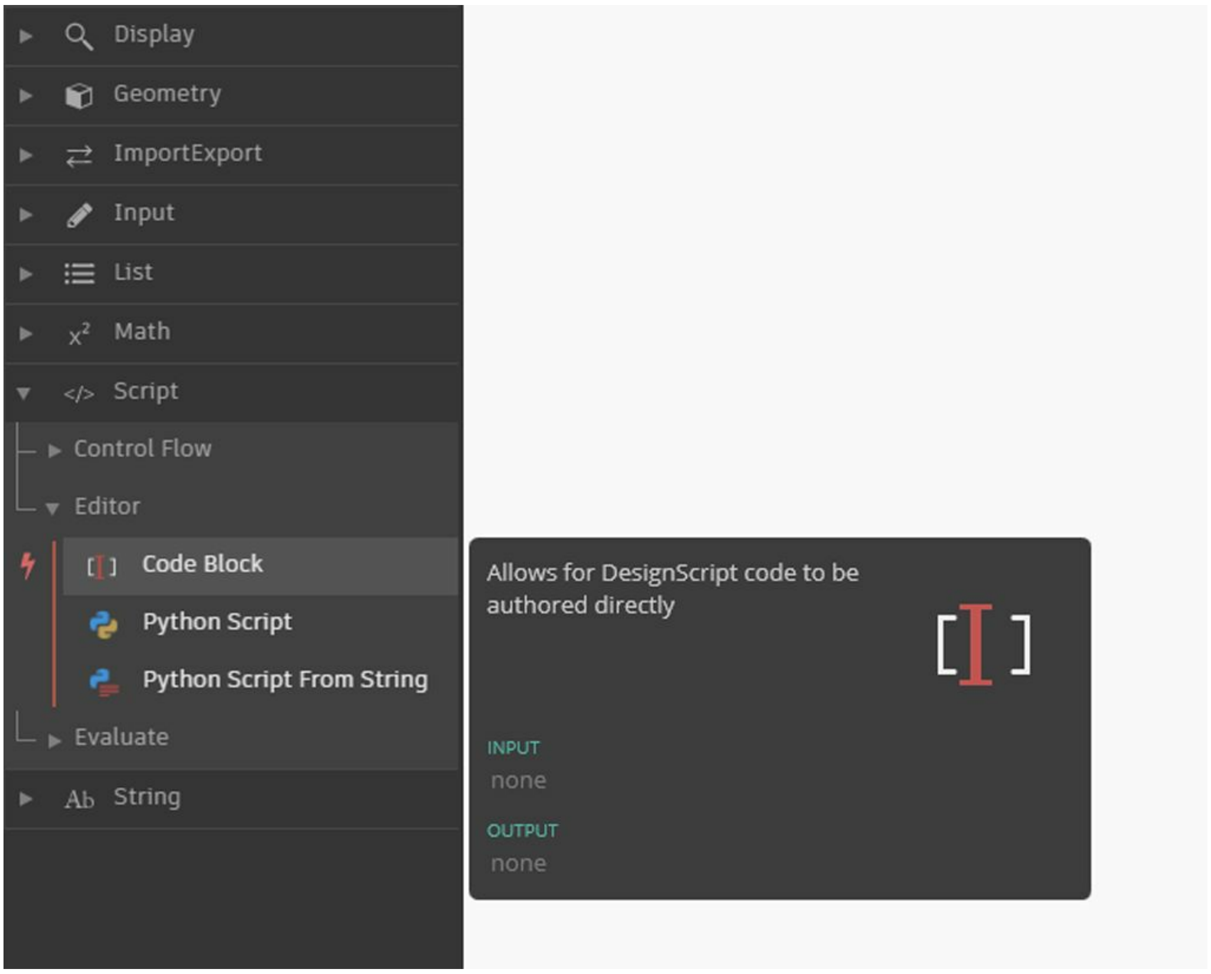
With code blocks, a user has the flexibility to decide how to specify inputs. Here are several different ways to make a basic point with coordinates *(10, 5, 0)*:


Code Block
10;
Code Block
5;
Point.ByCoordinates
x
y
z
Point
Watch
Point(X = 10.000, Y = 5.000, Z = 0.000)
Code Block
x
y
z = 0;
Point.ByCoordinates(x,y,z);
Watch
Point(X = 10.000, Y = 5.000, Z = 0.000)
Code Block
num
Point.ByCoordinates(num,-num,0);
Watch
Point(X = 10.000, Y = -10.000, Z = 0.000)
Code Block
Point.ByCoordinates(10,-10,0);
Watch
Point(X = 10.000, Y = -10.000, Z = 0.000)

As you learn more of the available functions in the library, you might even find that typing “Point.ByCoordinates” is faster than searching in the library and finding the proper node. When you type in *"Point."* for example, Dynamo will display a list of possible functions to apply to a Point. This makes the scripting more intuitive and will help with learning how to apply functions in Dynamo.

Creating Code Block Nodes

The code block can be found in *Core>Input>Actions>Code Block*. But even faster, just double click on the canvas and the code block appears. This node is used so often, it's given full double-click privileges.



Allows for DesignScript code to be authored directly

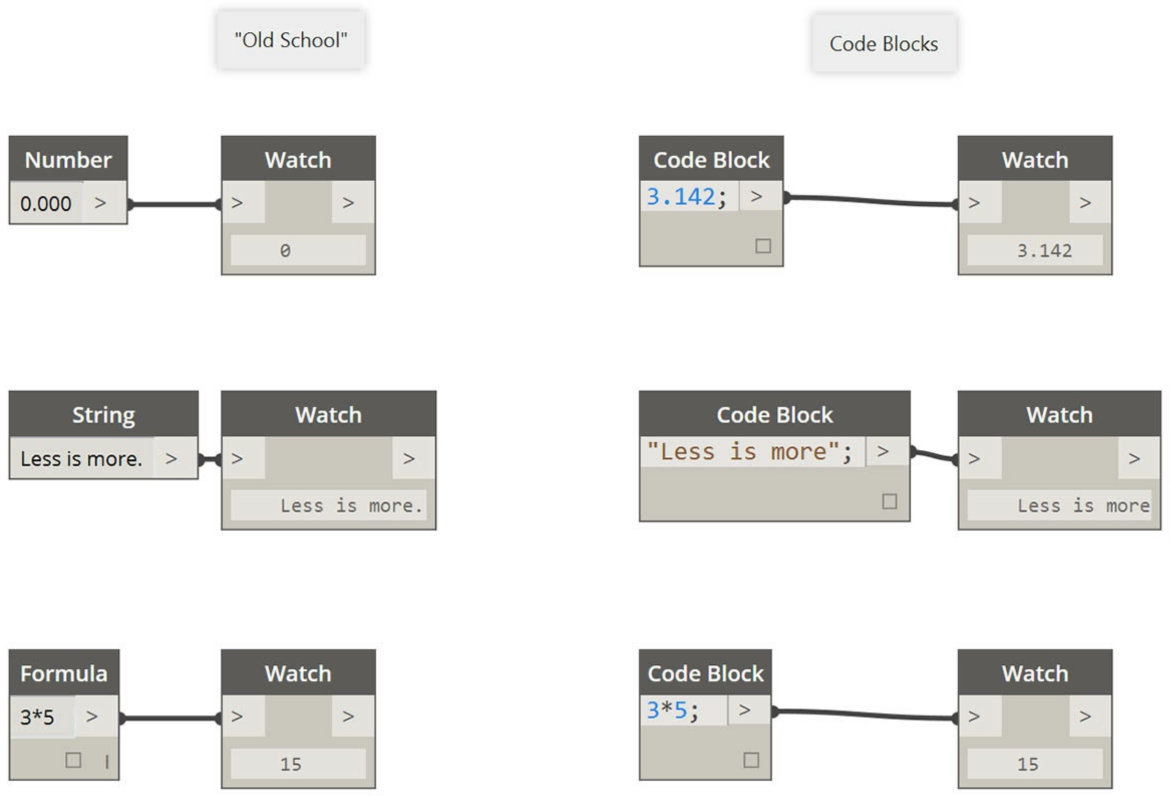
INPUT
none

OUTPUT
none

Numbers, strings, and formulas

Code blocks are also flexible towards data types. The user can quickly define numbers, strings, and formulas and the code block will deliver the desired output.

In the image below, you can see the "old school" way of doing things is a little long-winded: the user searches for the intended node in the interface, adds the node to the canvas, and then inputs the data. With code block, the user can double-click on the canvas to pull up the node, and type in the correct data type with basic syntax.



The *number*, *string*, and *formula* nodes are three examples of Dynamo nodes which are arguably obsolete in comparison to the *code block*.

DesignScript Syntax

DesignScript Syntax

You may have noticed a common theme in the names of nodes in Dynamo: each node uses a *"."* syntax without spaces. This is because the text at the top of each node respresents the actual syntax for scripting, and the *"."* (or *dot notation*) separates an element from the possible methods we can call. This creates an easy translation from visual scripting to text-based scripting.

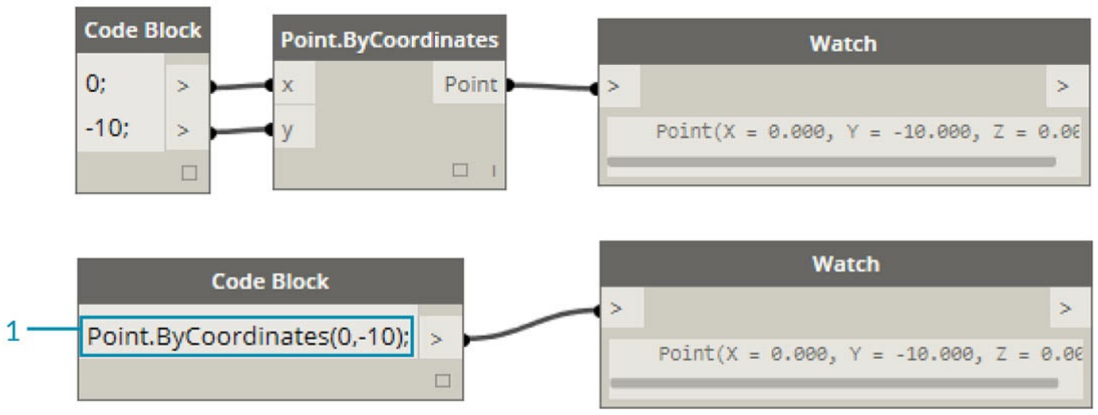
As a general analogy for the dot notation, how can we deal with a parametric apple in Dynamo? Below are a few methods we'll run on the apple before deciding to eat it. (Note: these are not actual Dynamo methods):

Humanly Readible	Dot Notation	Output
What color is the apple?	Apple.color	red
Is the apple ripe?	Apple.isRipe	true
How much does the apple weigh?	Apple.weight	6 oz.
Where did the apple come from?	Apple.parent	tree
What does the apple create?	Apple.children	seeds
Is this apple locally grown?	Apple.distanceFromOrchard	60 mi.

I don't know about you, but judging by the outputs in the table above, this looks like one tasty apple. I think I'll *Apple.eat()* it.

Dot Notation in Code Block

With the apple analogy in mind, let's look at *Point.ByCoordinates* and show how we can create a point using the code block:


Code Block
0;
-10;
Point.ByCoordinates
x
y
Point
Watch
Point(X = 0.000, Y = -10.000, Z = 0.00
1
Code Block
Point.ByCoordinates(0,-10);
Watch
Point(X = 0.000, Y = -10.000, Z = 0.00

The *code block* syntax `Point.ByCoordinates(0,10);` gives the same result as a *Point.ByCoordinates* node in Dynamo, except we're able to create a point using one node. This is more efficient than the connecting a separate node into *"X"* and *"Y"*.

1. By using *Point.ByCoordinates* in the code block, we are specifying the inputs in the same order as the out-of-the-box node *(X,Y)*.

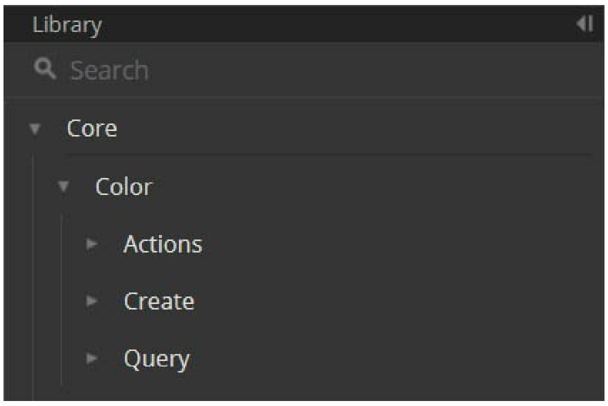
Calling Nodes

You can call any regular node in the library through a Code Block as long as the node isn't a special *"UI" node*: those with a special user interface feature. For instance, you can call *Circle.ByCenterPointRadius*, but it wouldn't make much sense to call a *Watch 3D* node.

Regular nodes (most of your library), generally come in three types:

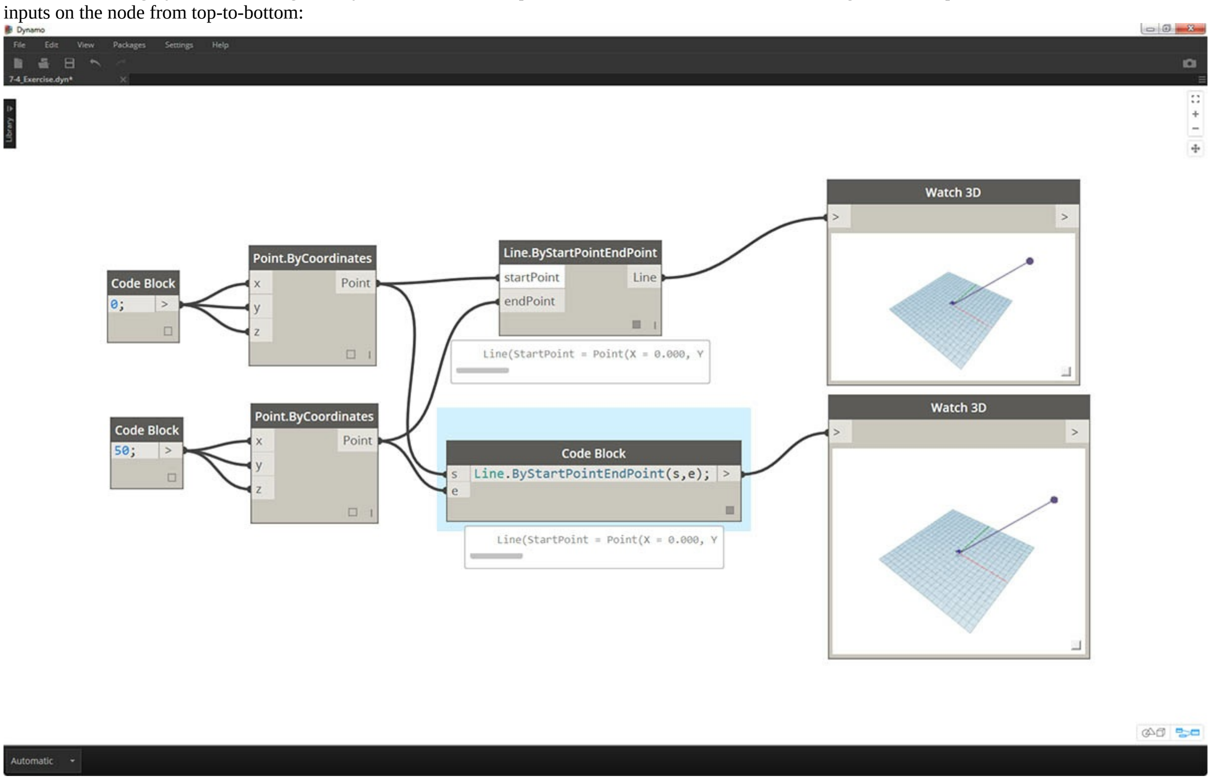
- **Create** - Create (or construct) something
- **Action** - Perform an action on something
- **Query** - Get a property of something that already exists

You’ll find that the library is organized with these categories in mind. Methods, or nodes, of these three types are treated differently when invoked within a Code Block.



Create

The "Create" category will construct geometry from scratch. We input values in the code block from left-to-right. These inputs are in the same order as the inputs on the node from top-to-bottom:

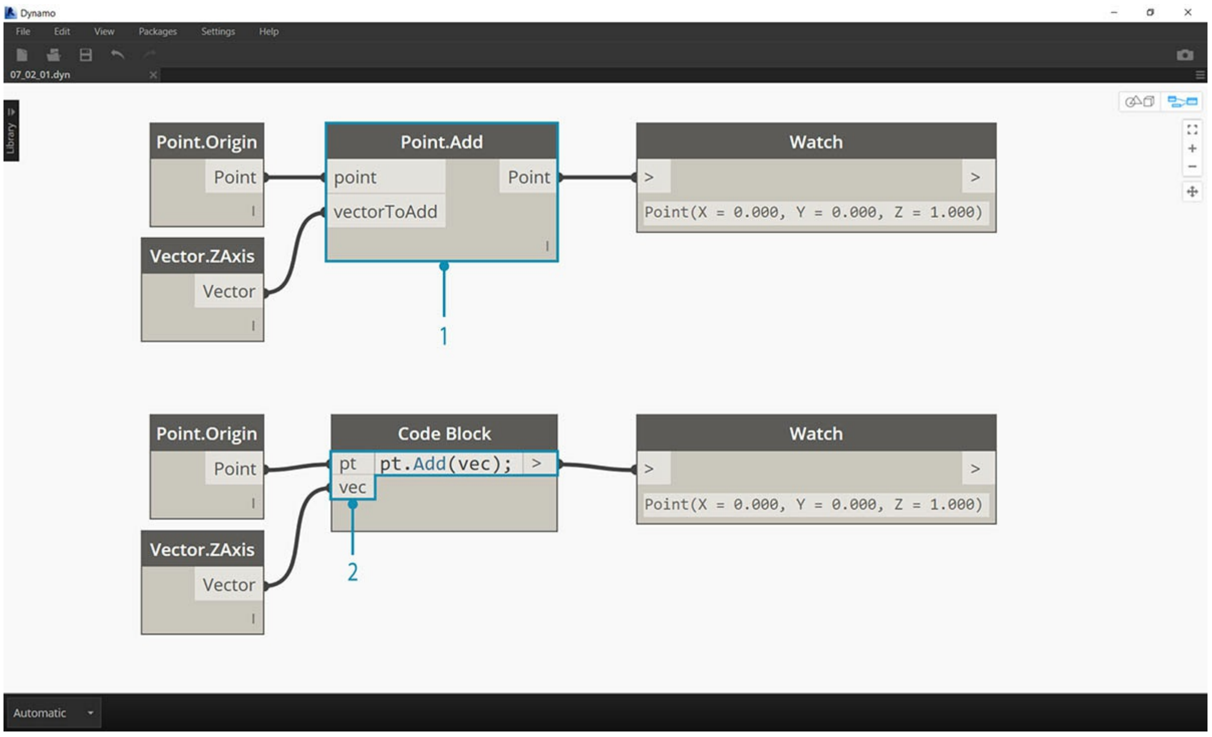


```
0;
50;
Line.ByStartPointEndPoint(s,e);
Line(StartPoint = Point(X = 0.000, Y
Line(StartPoint = Point(X = 0.000, Y
```

Comparing the *Line.ByStartPointEndPoint* node and the corresponding syntax in the code block, we get the same results.

Action

An action is something you do to an object of that type. Dynamo uses *dot notation*, common to many coding languages, to apply an action to a thing. Once you have the thing, type a dot then the name of the action. The action-type method’s input is placed in parentheses just like create-type methods, only you don’t have to specify the first input you see on the corresponding node. Instead, we specify the element upon which we are performing the action:

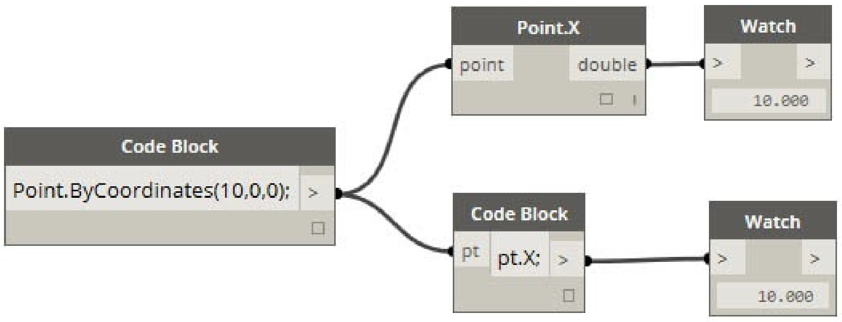


```
Point(X = 0.000, Y = 0.000, Z = 1.000)
```

1. The *Point.Add* node is an action-type node, so the syntax works a little differently.
2. The inputs are (1) the *point*, and (2) the *vector* to add to it. In a *Code Block*, we've named the point (the thing) *"pt"*. To add a vector named *"vec"* to *"pt"*, we would write *pt.Add(vec)*, or: thing, dot, action. The Add action only has one input, or all the inputs from the *Point.Add* node minus the first one. The first input for the *Point.Add* node is the point itself.

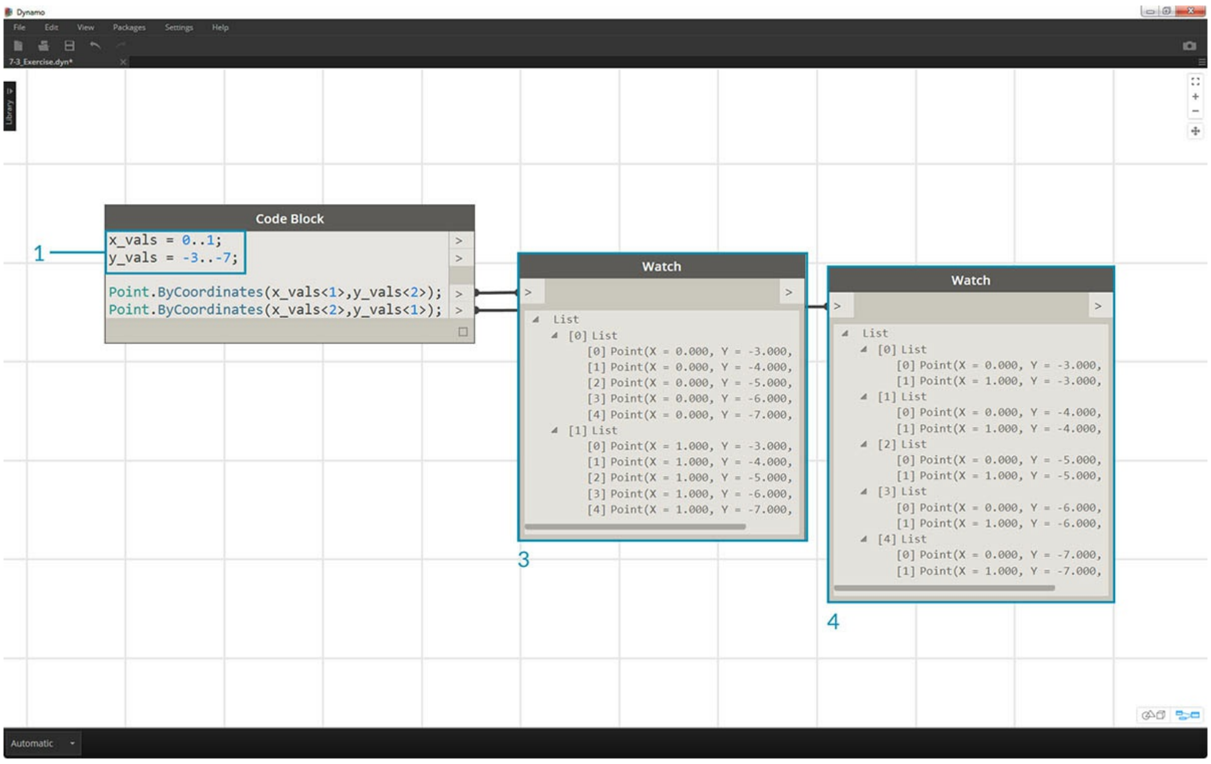
Query

Query-type methods get a property of an object. Since the object itself is the input, you don't have to specify any inputs. No parentheses required.



How About Lacing?

Lacing with nodes is somewhat different from lacing with code block. With nodes, the user right clicks on the node and selects the lacing option to perform. With code block, the user has much more control as to how the data is structured. The code block shorthand method uses *replication guides* to set how several one-dimensional lists should be paired. Numbers in angled brackets "<>" define the hierarchy of the resulting nested list: <1>,<2>,<3>, etc.



```
x_vals = 0..1;
y_vals = -3..-7;

Point.ByCoordinates(x_vals<1>,y_vals<2>);
Point.ByCoordinates(x_vals<2>,y_vals<1>);
```

```
List
  [0] List
      [0] Point(X = 0.000, Y = -3.000,
      [1] Point(X = 0.000, Y = -4.000,
      [2] Point(X = 0.000, Y = -5.000,
      [3] Point(X = 0.000, Y = -6.000,
      [4] Point(X = 0.000, Y = -7.000,
  [1] List
      [0] Point(X = 1.000, Y = -3.000,
      [1] Point(X = 1.000, Y = -4.000,
      [2] Point(X = 1.000, Y = -5.000,
      [3] Point(X = 1.000, Y = -6.000,
      [4] Point(X = 1.000, Y = -7.000,
```

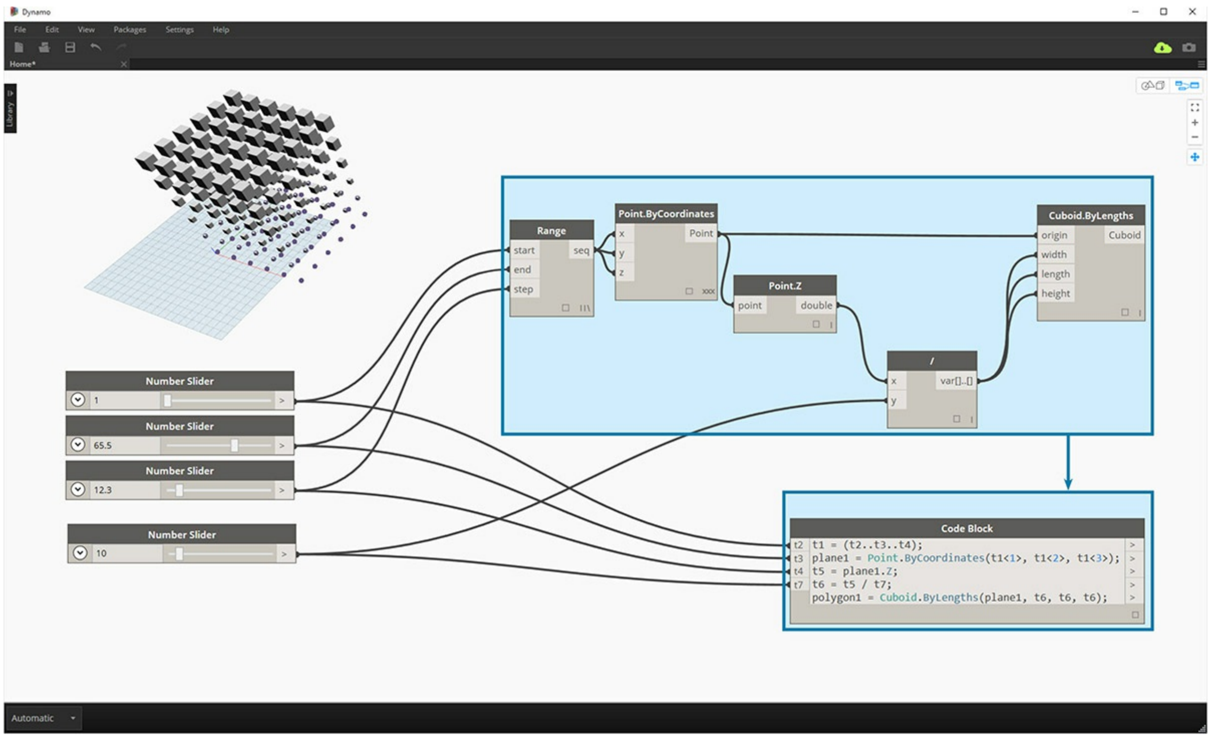
```
List
  [0] List
      [0] Point(X = 0.000, Y = -3.000,
      [1] Point(X = 1.000, Y = -3.000,
  [1] List
      [0] Point(X = 0.000, Y = -4.000,
      [1] Point(X = 1.000, Y = -4.000,
  [2] List
      [0] Point(X = 0.000, Y = -5.000,
      [1] Point(X = 1.000, Y = -5.000,
  [3] List
      [0] Point(X = 0.000, Y = -6.000,
      [1] Point(X = 1.000, Y = -6.000,
  [4] List
      [0] Point(X = 0.000, Y = -7.000,
      [1] Point(X = 1.000, Y = -7.000,
```

1. In this example, we use a shorthand to define two ranges (more on shorthand in the following section of this chapter). In short, `0..1;` is equivalent to `{0,1}` and `-3..-7`is equivalent to `{-3,-4,-5,-6,-7}`. The result gives us lists of 2 x-values and 5 y-values. If we don't use replication guides with these mismatched lists, we get a list of two points, which is the length of the shortest list. Using replication guides, we can find all of the possible combinations of 2 and 5 coordinates (or, a **Cross Product**).
2. Using the syntax `Point.ByCoordinates(x_vals<1>,y_vals<2>);` we get **two** lists with **five** items in each list.
3. Using the syntax `Point.ByCoordinates(x_vals<2>,y_vals<1>);` we get **five** lists with **two** items in each list.

With this notation, we can also specify which list will be dominant: 2 lists of 5 things or 5 lists of 2 things. In the example, changing the order of the replication guides makes the result a list of rows of points or a list of columns of points in a grid.

Node to Code

While the code block methods above may take some getting used to, there is a feature in Dynamo called "Node to Code" which will make the process easier. To use this feature, select an array of nodes in your Dynamo graph, right-click on the canvas and select "Node to Code". Dynamo condenses these nodes into a code block, with all of the inputs and outputs! Not only is this a great tool for learning code block, but it also allows you to work with a more efficient and parametric Dynamo graph. We'll conclude the exercise below by using "Node to Code", so don't miss it.

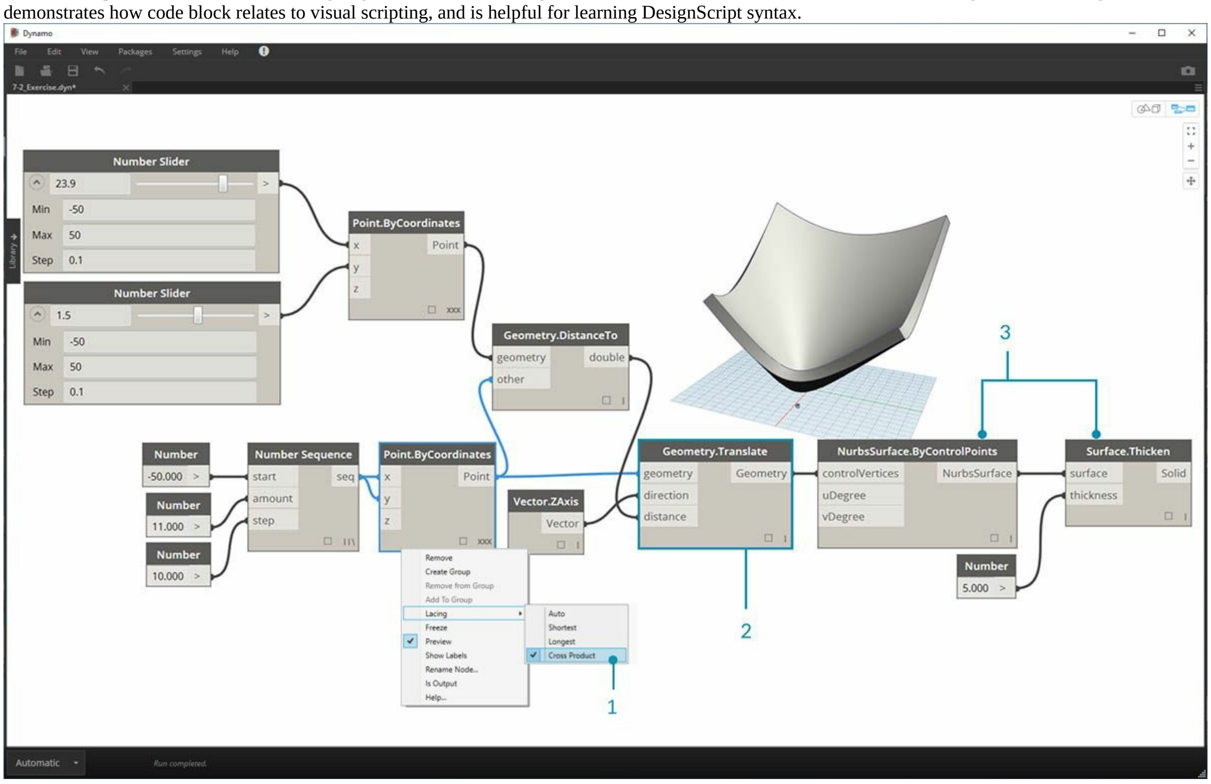


```
t1 = (t2..t3..t4);
plane1 = Point.ByCoordinates(t1<1>, t1<2>, t1<3>);
t5 = plane1.Z;
t6 = t5 / t7;
polygon1 = Cuboid.ByLengths(plane1, t6, t6, t6);
```

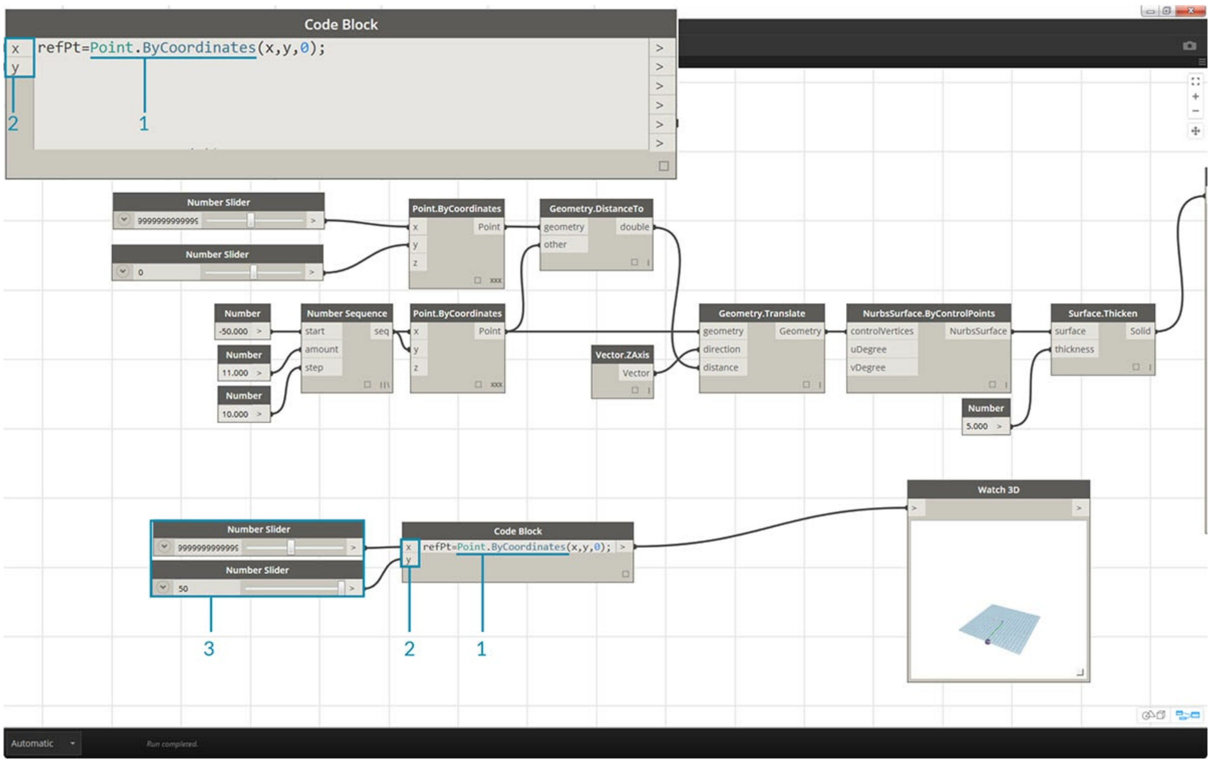
Exercise

> Download the example file that accompanies this exercise (Right click and "Save Link As..."). A full list of example files can be found in the Appendix. Dynamo-Syntax_Attractor-Surface.dyn

To show the power of code block, we are going to translate an existing attractor field definition into code block form. Working with an existing definition demonstrates how code block relates to visual scripting, and is helpful for learning DesignScript syntax.

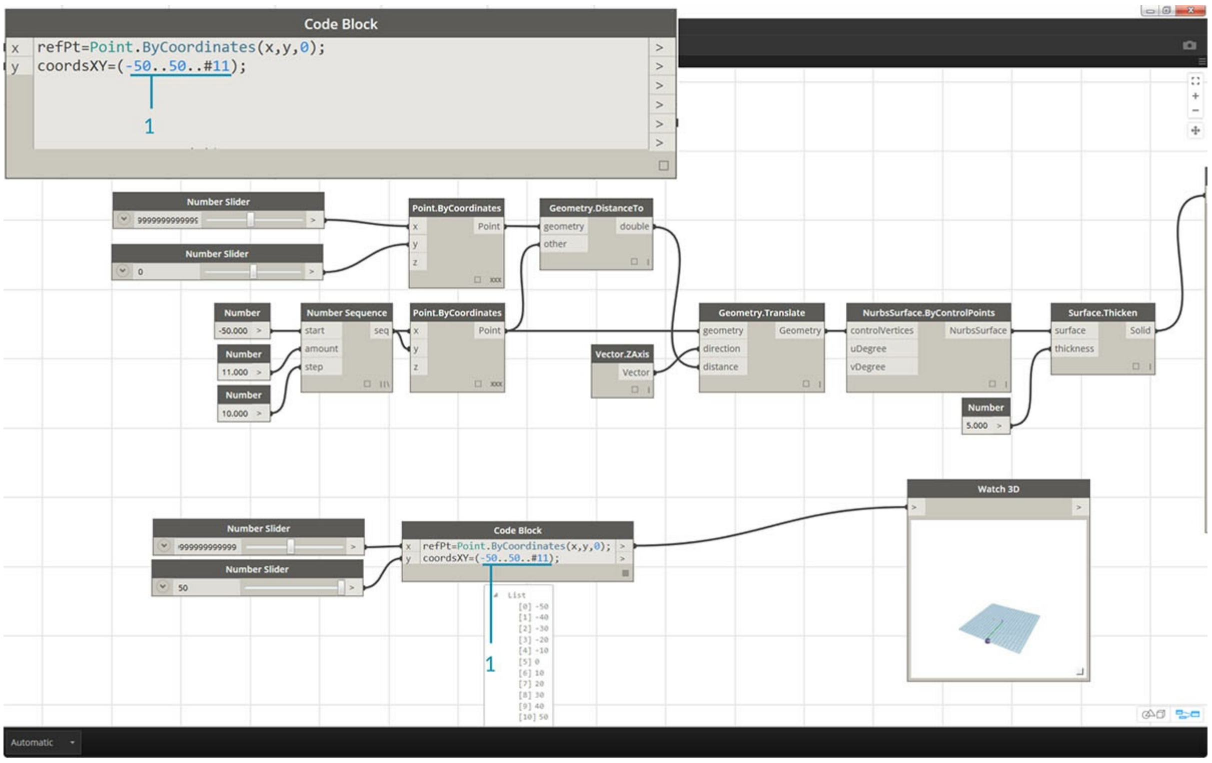


> Begin by recreating the definition in the image above (or by opening the sample file).
>
> 1. Notice that the lacing on *Point.ByCoordinates* has been set to *Cross Product*.
> 2. Each point in a grid is moved up in the Z direction based on its distance to the reference point.
> 3. A surface is recreated and thickened, creating a bulge in the geometry relative to the distance to the reference point.



```
refPt=Point.ByCoordinates(x,y,0);
```

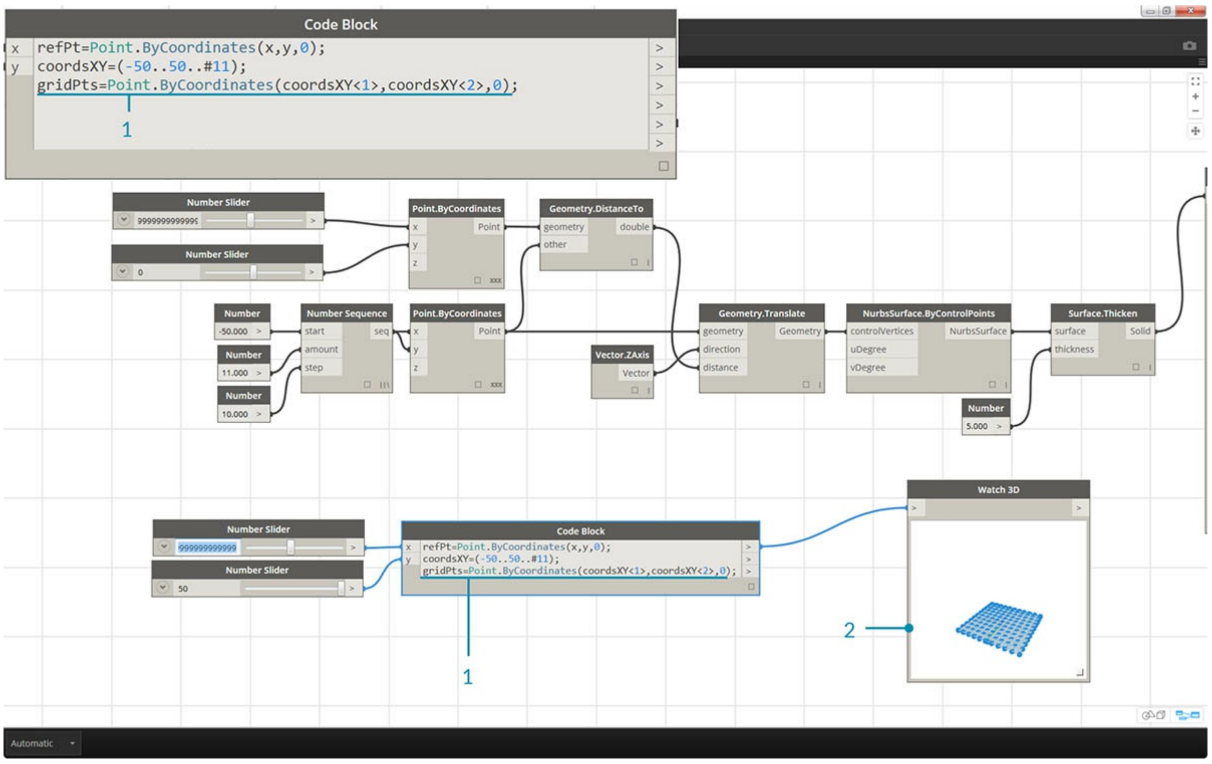
1. Starting from the beginning, let's define the reference point first: `Point.ByCoordinates(x,y,0);`. We use the same *Point.ByCoordinates* syntax as is specified on the top of the reference point node.
2. The variables *x* and *y* are inserted into the code block so that we may update these dynamically with sliders.
3. Add some *sliders* to the *code block* inputs which range from *-50* to *50*. This way, we can span across the default Dynamo grid.



```
refPt=Point.ByCoordinates(x,y,0);
coordsXY=(-50..50..#11);
```

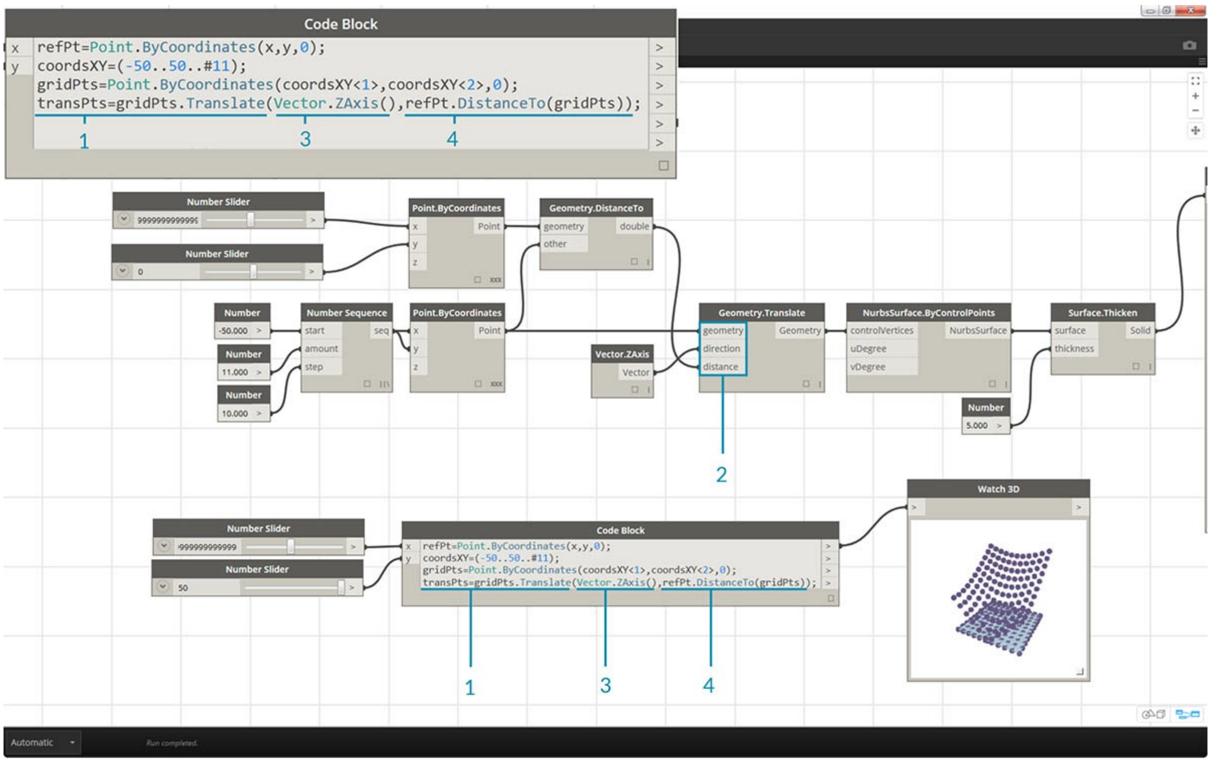
```
List
  [0] -50
  [1] -40
  [2] -30
  [3] -20
  [4] -10
  [5] 0
  [6] 10
  [7] 20
  [8] 30
  [9] 40
  [10] 50
```

1. In the second line of the *code block*, we define a shorthand to replace the number sequence node: `coordsXY = (-50..50..#11);`. We'll discuss this more in the next section. For now, notice that this shorthand is equivalent to the *Number Sequence* node in the visual script.



```
refPt=Point.ByCoordinates(x,y,0);
coordsXY=(-50..50..#11);
gridPts=Point.ByCoordinates(coordsXY<1>,coordsXY<2>,0);
```

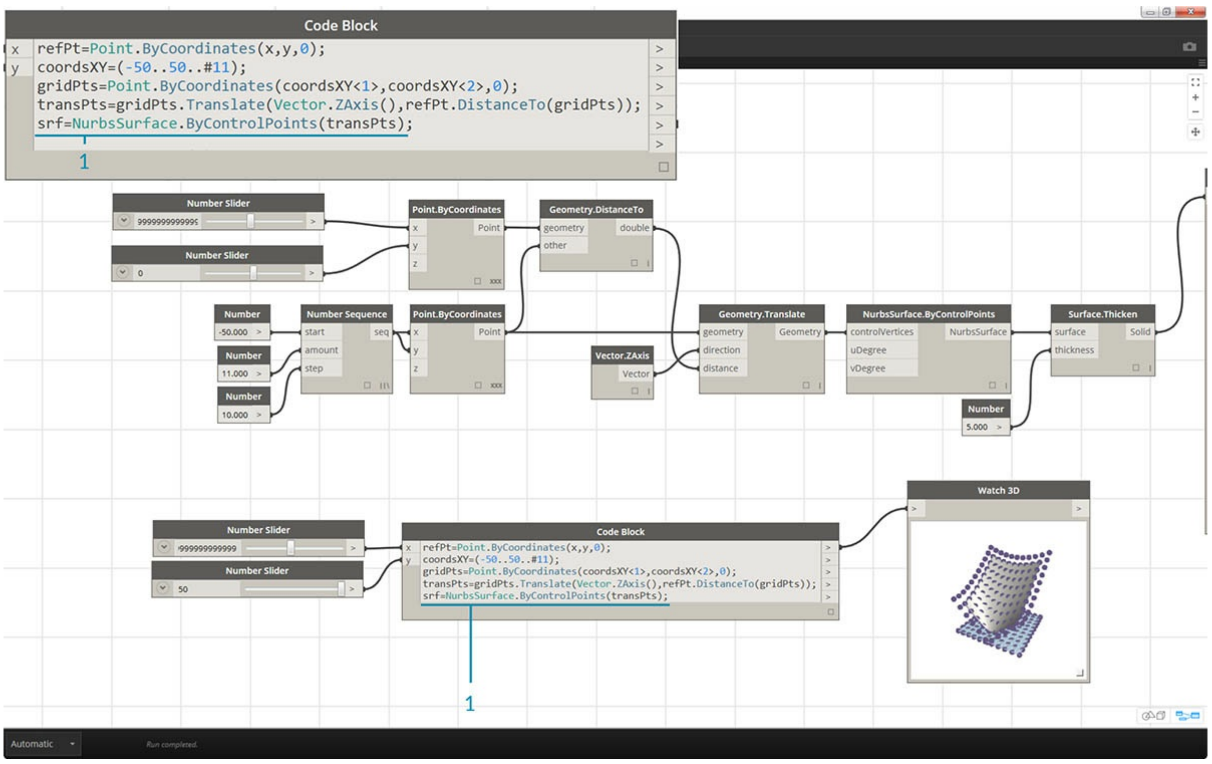
1. Now, we want to create a grid of points from the *coordsXY* sequence. To do this, we want to use the *Point.ByCoordinates* syntax, but also need to initiate a *Cross Product* of the list in the same manner that we did in the visual script. To do this, we type the line: `gridPts = Point.ByCoordinates(coordsXY<1>,coordsXY<2>,0);`. The angled brackets denote the cross product reference.
2. Notice in the *Watch3D* node that we have a grid of points across the Dynamo grid.



```
refPt=Point.ByCoordinates(x,y,0);
coordsXY=(-50..50..#11);
gridPts=Point.ByCoordinates(coordsXY<1>,coordsXY<2>,0);
transPts=gridPts.Translate(Vector.ZAxis(),refPt.DistanceTo(gridPts));
```

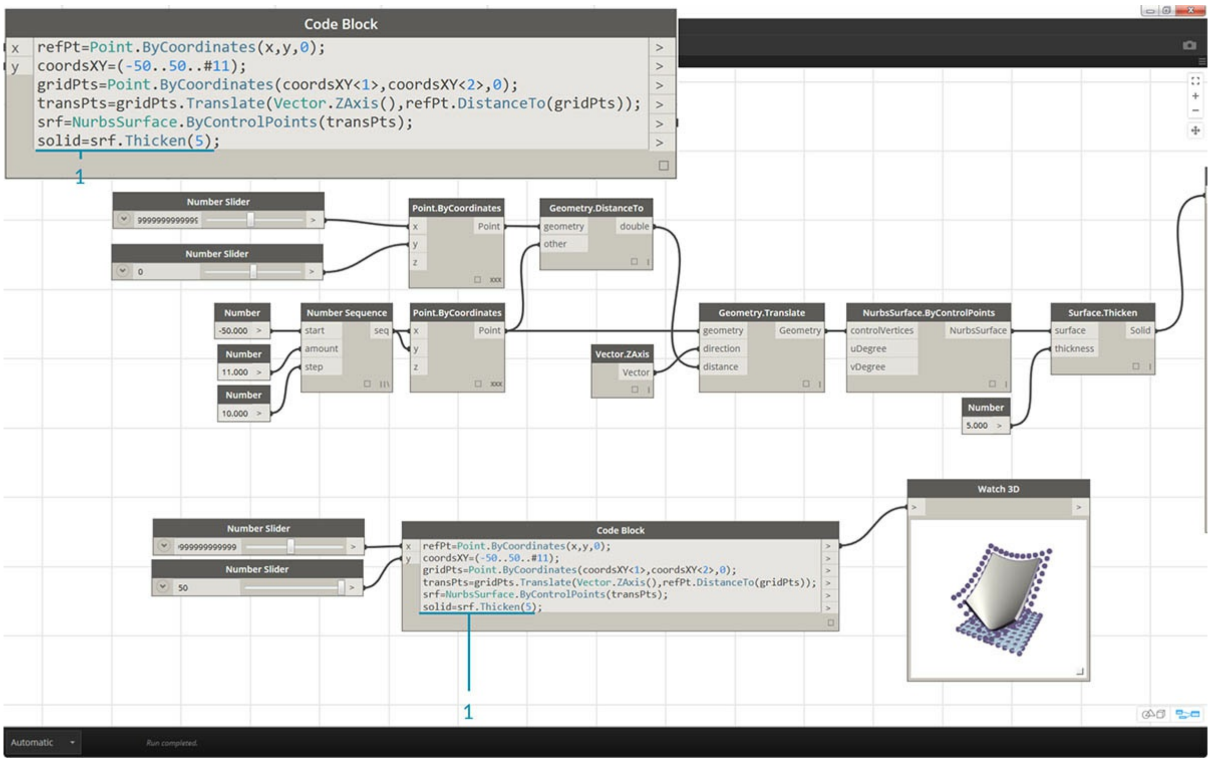
1. Now for the tricky part: We want to move the grid of points up based on their distance to the reference point. First, let's call this new set of points *transPts*. And since a translation is an action on an existing element, rather than using `Geometry.Translate...`, we use `gridPts.Translate`.
2. Reading from the actual node on the canvas, we see that there are three inputs. The geometry to translate is already declared because we are performing the action on that element (with *gridPts.Translate*). The remaining two inputs will be inserted into the parentheses of the function: *direction* and *distance*.
3. The direction is simple enough, we use a `Vector.ZAxis()` to move vertically.
4. The distance between the reference point and each grid point still needs to be calculated, so we do this as an action to the reference point in the same manner: `refPt.DistanceTo(gridPts)`

5. The final line of code gives us the translated points: `transPts = gridPts.Translate(Vector.ZAxis(),refPt.DistanceTo(gridPts));`



```
refPt=Point.ByCoordinates(x,y,0);
coordsXY=(-50..50..#11);
gridPts=Point.ByCoordinates(coordsXY<1>,coordsXY<2>,0);
transPts=gridPts.Translate(Vector.ZAxis(),refPt.DistanceTo(gridPts));
srf=NurbsSurface.ByControlPoints(transPts);
```

1. We now have a grid of points with the appropriate data structure to create a Nurbs Surface. We construct the surface using `srf = NurbsSurface.ByControlPoints(transPts);`

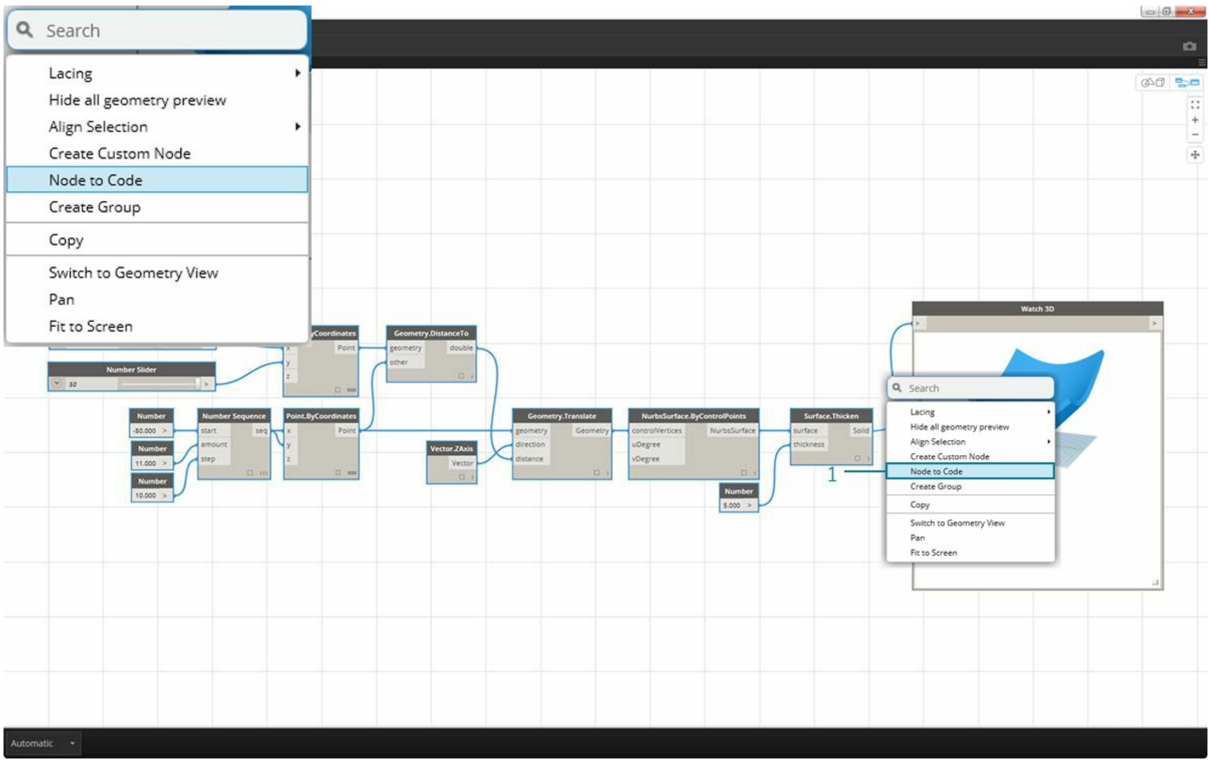


```
refPt=Point.ByCoordinates(x,y,0);
coordsXY=(-50..50..#11);
gridPts=Point.ByCoordinates(coordsXY<1>,coordsXY<2>,0);
transPts=gridPts.Translate(Vector.ZAxis(),refPt.DistanceTo(gridPts));
srf=NurbsSurface.ByControlPoints(transPts);
solid=srf.Thicken(5);
```

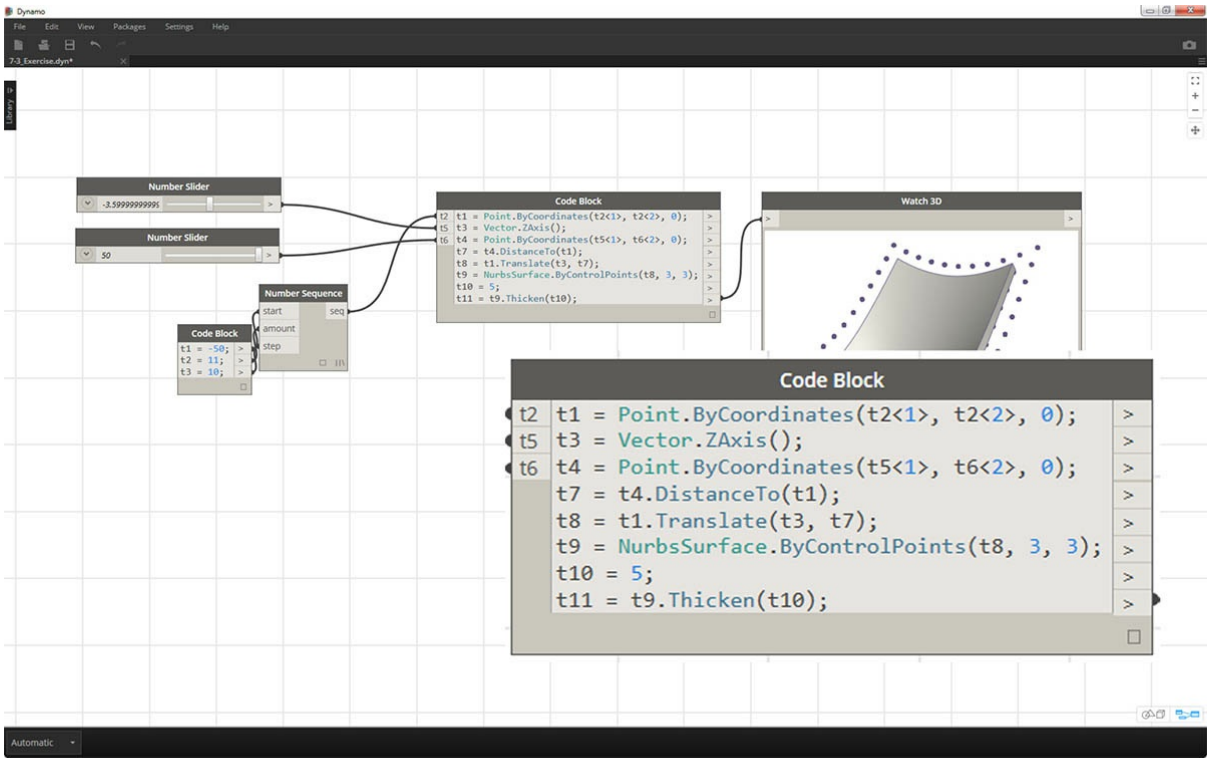
1. And finally, to add some depth to the surface, we construct a solid using `solid = srf.Thicken(5);` In this case we thickened the surface by 5 units in the code, but we could always declare this as a variable (calling it *thickness* for example) and then control that value with a slider.

Simplify the Graph with "Node to Code"

The "Node to Code" feature automates the entire exercise that we just completed with the click of a button. Not only is this powerful for creating custom definitions and reusable code blocks, but it is also a really helpful tool to learn how to script in Dynamo:



1. Start with the existing visual script from step 1 of the exercise. Select all of the nodes, right click on the canvas, and select *"Node to Code"*. Simple as that.



```
t1 = -50;
t2 = 11;
t3 = 10;
```

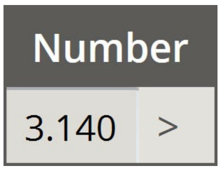
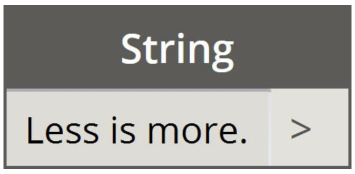
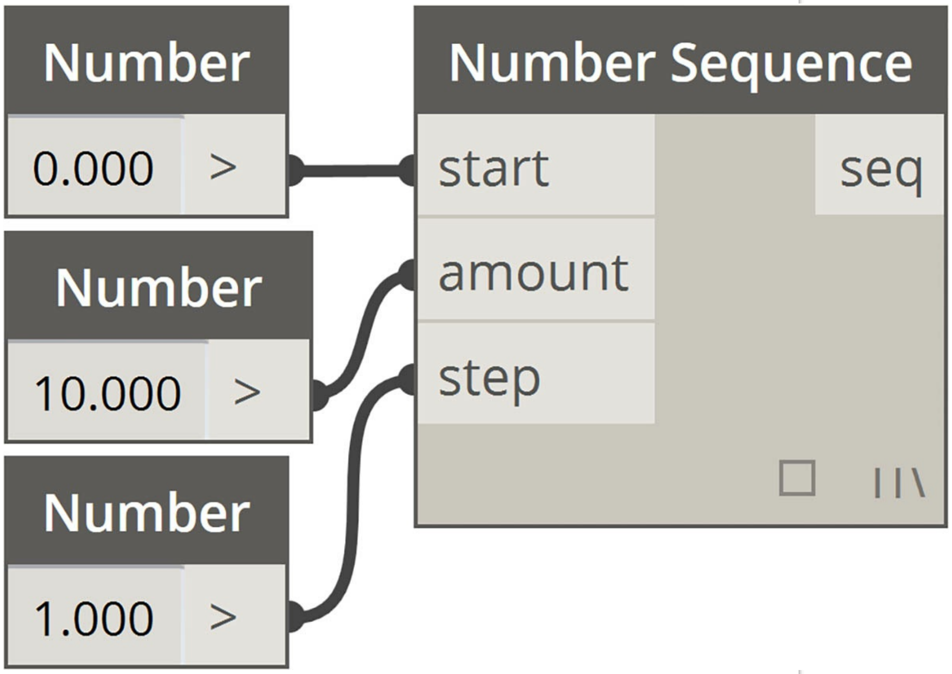
```
t1 = Point.ByCoordinates(t2<1>, t2<2>, 0);
t3 = Vector.ZAxis();
t4 = Point.ByCoordinates(t5<1>, t6<2>, 0);
t7 = t4.DistanceTo(t1);
t8 = t1.Translate(t3, t7);
t9 = NurbsSurface.ByControlPoints(t8, 3, 3);
t10 = 5;
t11 = t9.Thicken(t10);
```

Dynamo has automated a text based version of the visual graph, lacing and all. Test this out on your visual scripts and release the power of the code block!

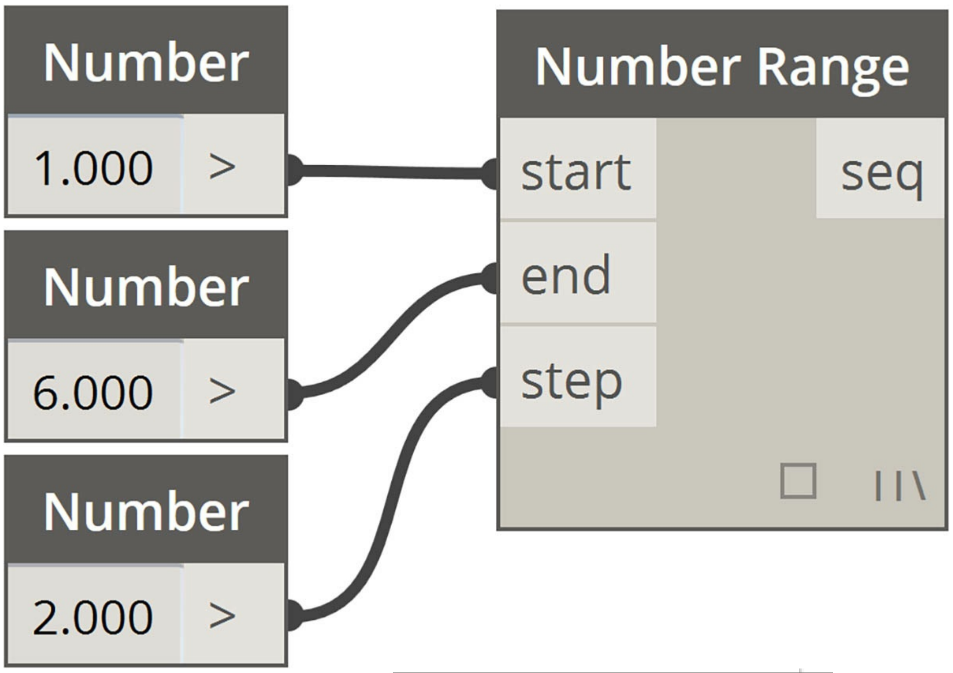
Shorthand

Shorthand

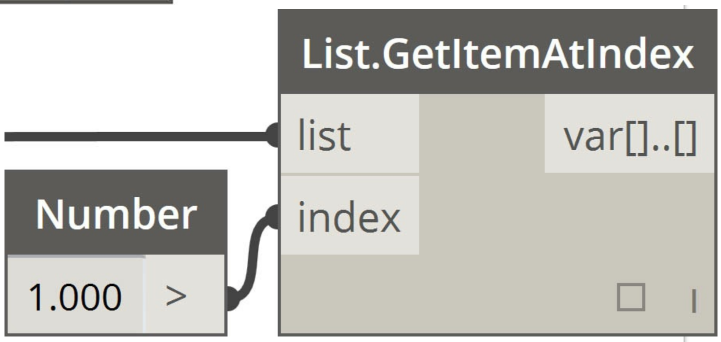
There are a few basic shorthand methods in the code block which, simply put, make data management *a lot* easier. We'll break down the basics below and discuss how this shorthand can be used both for creating and querying data.

Data Type	Standard Dynamo
Numbers	
Strings	
Sequences	 Number 0.000 Number 10.000 Number 1.000 Number Sequence start amount step seq I I \

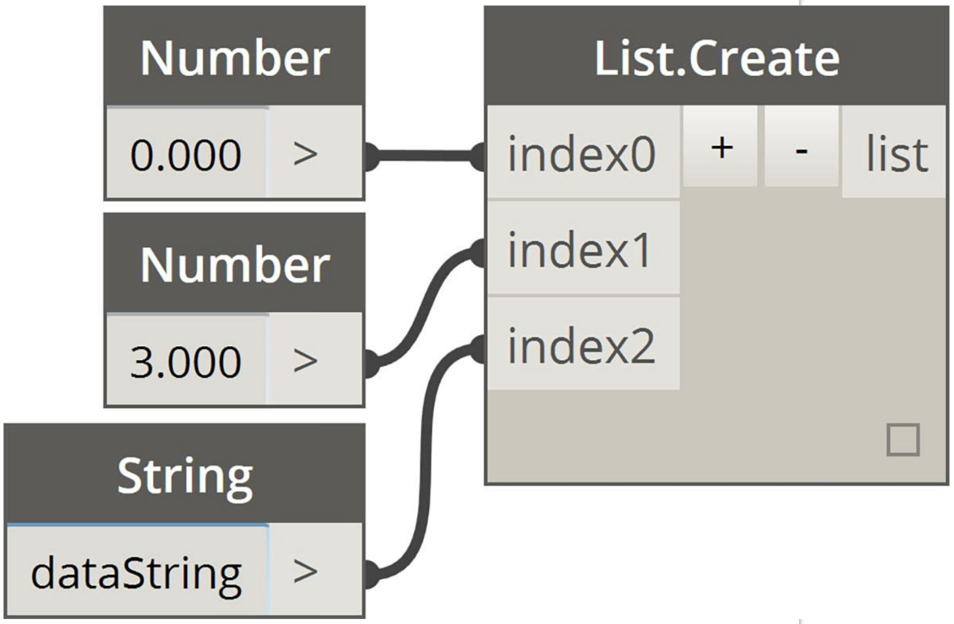
Ranges



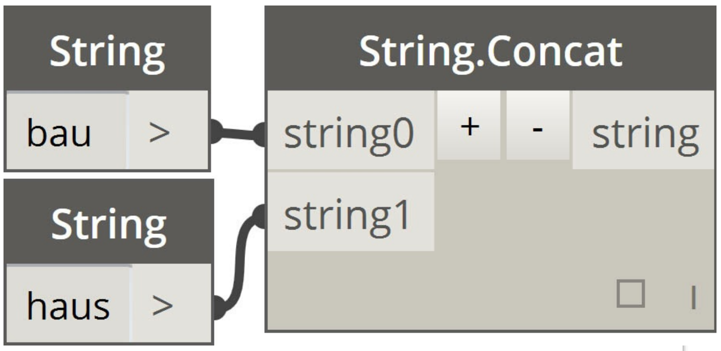
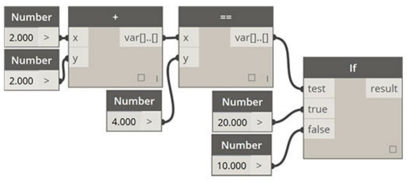
Get Item at Index



Create List



Concatenate Strings

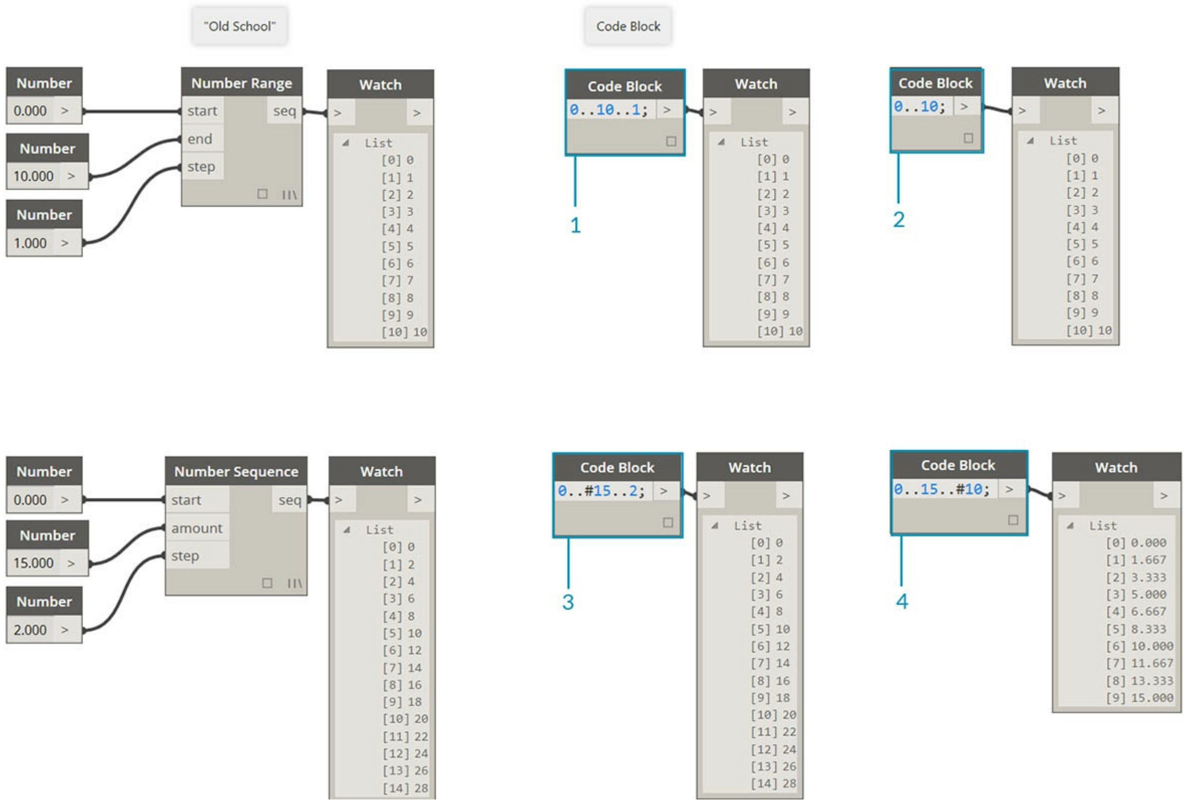
Concatenate Strings	 String bau String haus String.Concat string0 + - string string1
Conditional Statements	 Number 2.000 Number 2.000 + x y var[]..[] == Number 4.000 Number 20.000 Number 10.000 If test true false result

Additional Syntax

Node(s)	Code Block Equivalent	Note
Any operator (+, &&, >=, Not, etc.)	+, &&, >=, !, etc.	Note that "Not" becomes "!" but the node is called "Not" to distinguish from "Factorial"
Boolean True	true;	Note lower case
Boolean False	false;	Note lower case

Ranges

The method for defining ranges and sequences can be reduced to basic shorthand. Use the image below as a guide to the ".." syntax for defining a list of numerical data with code block. After getting the hang of this notation, creating numerical data is a really efficient process:



```
List
[0] 0
[1] 1
[2] 2
[3] 3
[4] 4
[5] 5
[6] 6
[7] 7
[8] 8
[9] 9
[10] 10
```

```
List
[0] 0
[1] 2
[2] 4
[3] 6
[4] 8
[5] 10
[6] 12
[7] 14
[8] 16
[9] 18
[10] 20
[11] 22
[12] 24
[13] 26
[14] 28
```

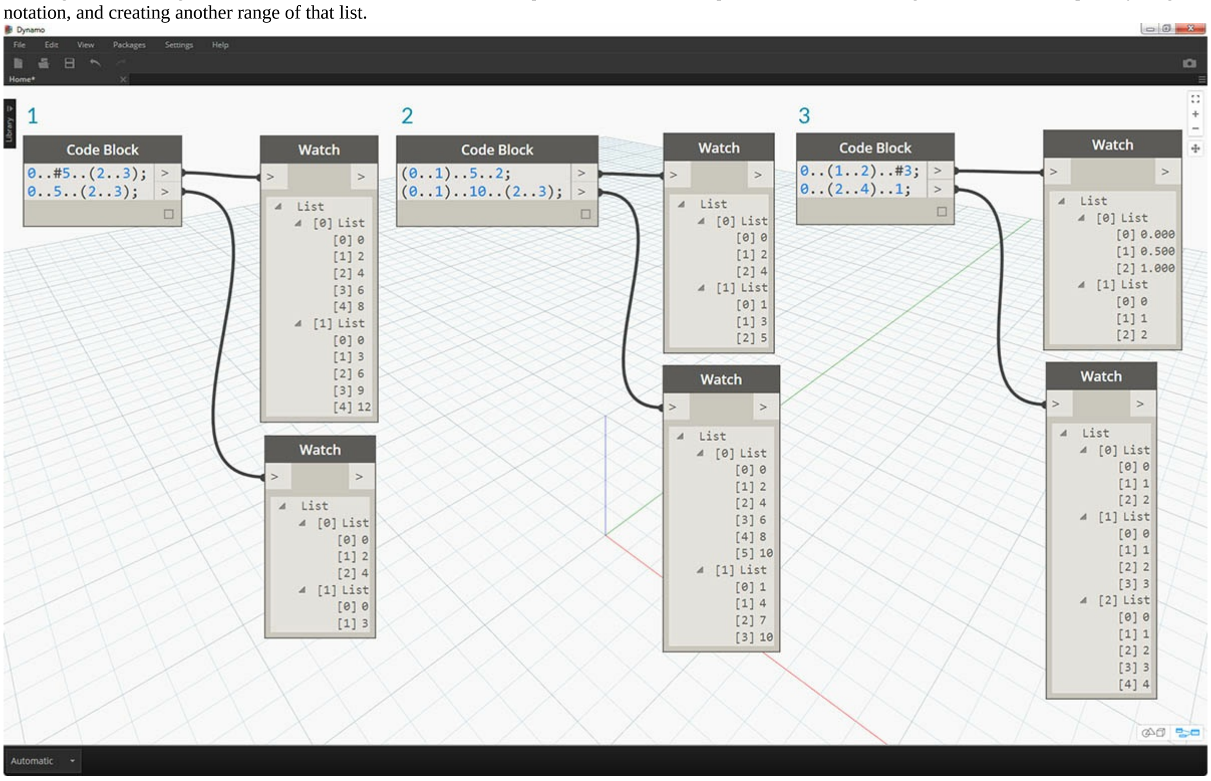
```
List
[0] 0.000
[1] 1.667
[2] 3.333
[3] 5.000
[4] 6.667
[5] 8.333
[6] 10.000
[7] 11.667
[8] 13.333
[9] 15.000
```

1. In this example, a number range is replaced by basic code block syntax defining the `beginning..end..step-size;`. Represented numerically, we get: `0..10..1;`
2. Notice that the syntax `0..10..1;` is equivalent to `0..10;`. A step-size of 1 is the default value for the shorthand notation. So `0..10;` will give a sequence from 0 to 10 with a step-size of 1.

3. The *number sequence* example is similar, except we use a *"#"* to state that we want 15 values in the list, rather than a list which goes up to 15. In this case, we are defining: `beginning..#ofSteps..step-size:`. The actual syntax for the sequence is `0..#15..2`
4. Using the *"#"* from the previous step, we now place it in the *"step-size"* portion of the syntax. Now, we have a *number range* spanning from the *"beginning"* to the *"end"* and the *"step-size"* notation evenly distributes a number of values between the two: `beginning..end..#ofSteps`

Advanced Ranges

Creating advanced ranges allows us to work with list of lists in a simple fashion. In the examples below, we're isolating a variable from the primary range notation, and creating another range of that list.



Code Block (1):
```
0..#5..(2..3);
0..5..(2..3);
```

Watch:
```
List
  [0] List
    [0] 0
    [1] 2
    [2] 4
    [3] 6
    [4] 8
  [1] List
    [0] 0
    [1] 3
    [2] 6
    [3] 9
    [4] 12
```

Watch:
```
List
  [0] List
    [0] 0
    [1] 2
    [2] 4
  [1] List
    [0] 0
    [1] 3
```

Code Block (2):
```
(0..1)..5..2;
(0..1)..10..(2..3);
```

Watch:
```
List
  [0] List
    [0] 0
    [1] 2
    [2] 4
  [1] List
    [0] 1
    [1] 3
    [2] 5
```

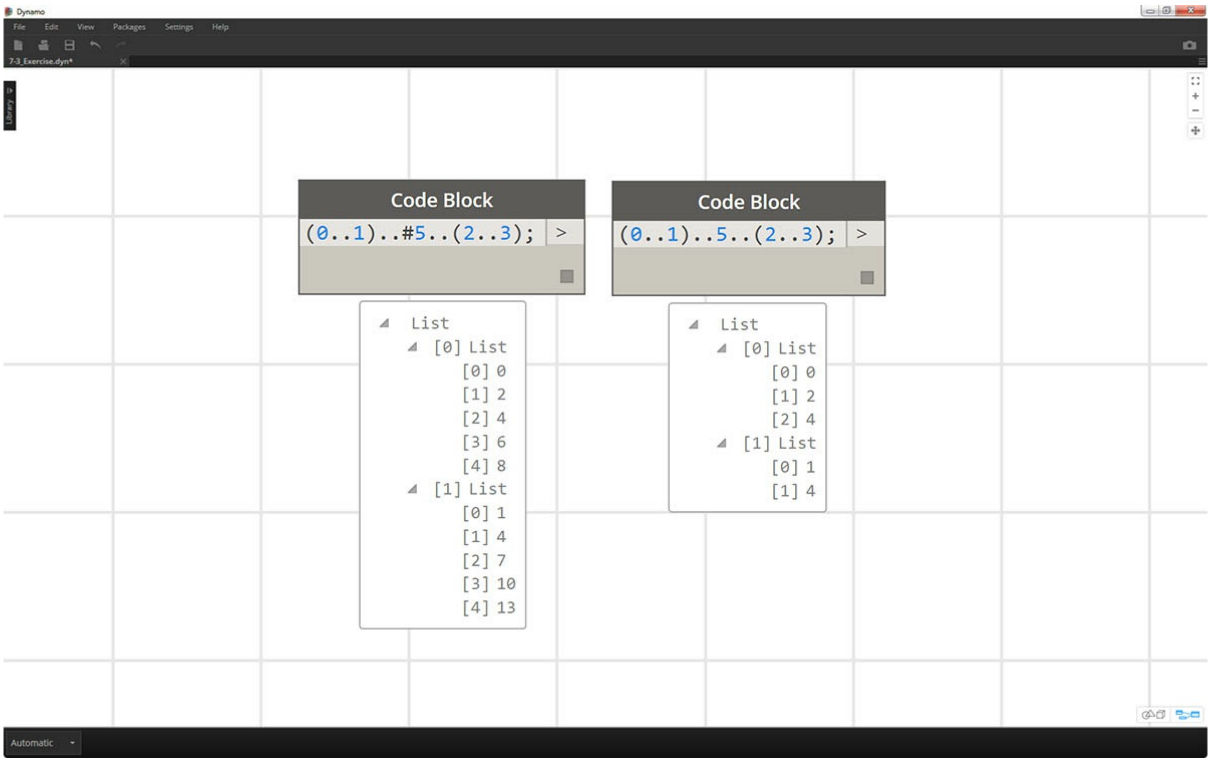
Watch:
```
List
  [0] List
    [0] 0
    [1] 2
    [2] 4
    [3] 6
    [4] 8
    [5] 10
  [1] List
    [0] 1
    [1] 4
    [2] 7
    [3] 10
```

Code Block (3):
```
0..(1..2)..#3;
0..(2..4)..1;
```

Watch:
```
List
  [0] List
    [0] 0.000
    [1] 0.500
    [2] 1.000
  [1] List
    [0] 0
    [1] 1
    [2] 2
```

Watch:
```
List
  [0] List
    [0] 0
    [1] 1
    [2] 2
  [1] List
    [0] 0
    [1] 1
    [2] 2
    [3] 3
  [2] List
    [0] 0
    [1] 1
    [2] 2
    [3] 3
    [4] 4
```

1. Creating nested ranges, compare the notation with a *"#"* vs. the notation without. The same logic applies as in basic ranges, except it gets a little more complex.
2. We can define a sub-range at any place within the primary range, and notice that we can have two sub-ranges as well.
3. By controlling the *"end"* value in a range, we create more ranges of differing lengths.



Code Block:
```
(0..1)..#5..(2..3);
```

```
List
  [0] List
    [0] 0
    [1] 2
    [2] 4
    [3] 6
    [4] 8
  [1] List
    [0] 1
    [1] 4
    [2] 7
    [3] 10
    [4] 13
```

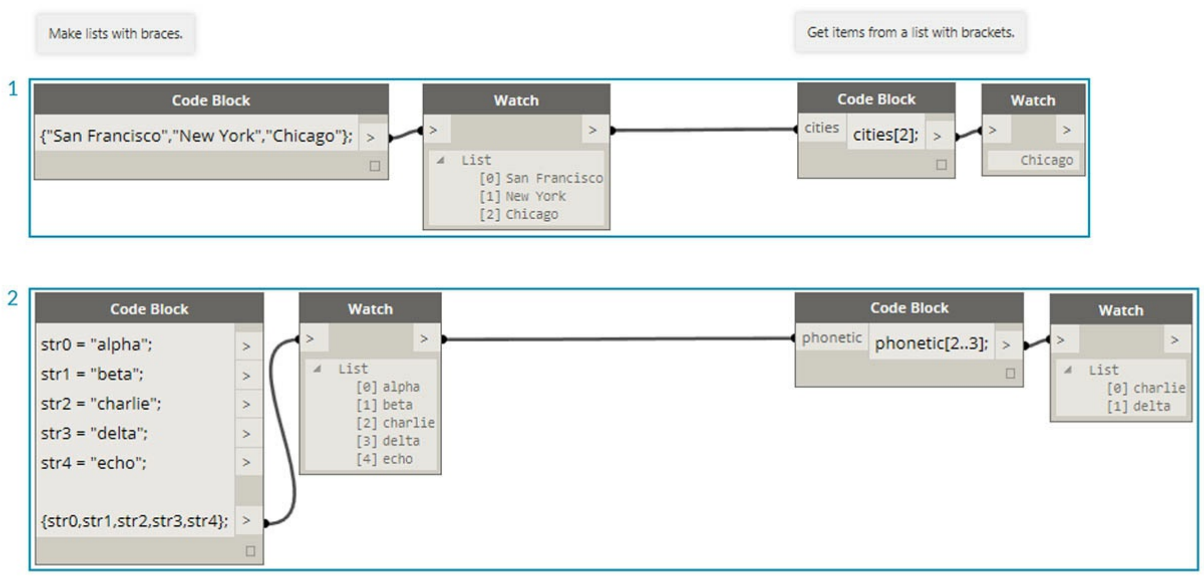
Code Block:
```
(0..1)..5..(2..3);
```

```
List
  [0] List
    [0] 0
    [1] 2
    [2] 4
  [1] List
    [0] 1
    [1] 4
```

As a logic exercise, compare the two shorthands above and try to parse through how *subranges* and the *"#"* notation drive the resultant output.

Make lists and get items from a list

In addition to making lists with shorthand, we can also create lists on the fly. These list can contain a wide range of element types and can also be queried (remember, lists are objects in themselves). To summarize, with code block you make lists with braces (a.k.a. “curly brackets”) and you query items from a list with brackets (a.k.a. “square brackets”):



```
{"San Francisco","New York","Chicago"};
```

```
List
  [0] San Francisco
  [1] New York
  [2] Chicago
```

```
cities[2];
```

```
Chicago
```

```
str0 = "alpha";
str1 = "beta";
str2 = "charlie";
str3 = "delta";
str4 = "echo";

{str0,str1,str2,str3,str4};
```

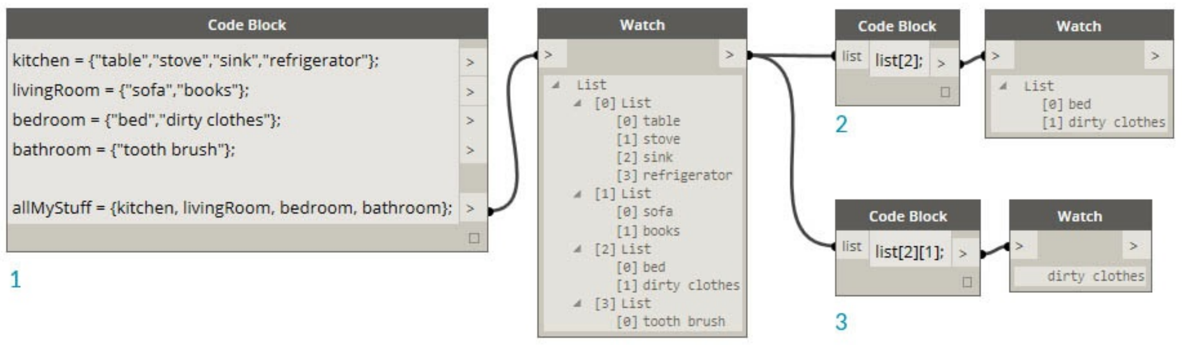
```
List
  [0] alpha
  [1] beta
  [2] charlie
  [3] delta
  [4] echo
```

```
phonetic[2..3];
```

```
List
  [0] charlie
  [1] delta
```

1. Create lists quickly with strings and query them using the item index.
2. Create lists with variables and query using the range shorthand notation.

And manging with nested lists is a similar process. Be aware of the list order and recall using multiple sets of square brackets:



```
kitchen = {"table","stove","sink","refrigerator"};
livingRoom = {"sofa","books"};
bedroom = {"bed","dirty clothes"};
bathroom = {"tooth brush"};

allMyStuff = {kitchen, livingRoom, bedroom, bathroom};
```

```
List
  [0] List
      [0] table
      [1] stove
      [2] sink
      [3] refrigerator
  [1] List
      [0] sofa
      [1] books
  [2] List
      [0] bed
      [1] dirty clothes
  [3] List
      [0] tooth brush
```

```
list[2];
```

```
List
  [0] bed
  [1] dirty clothes
```

```
list[2][1];
```

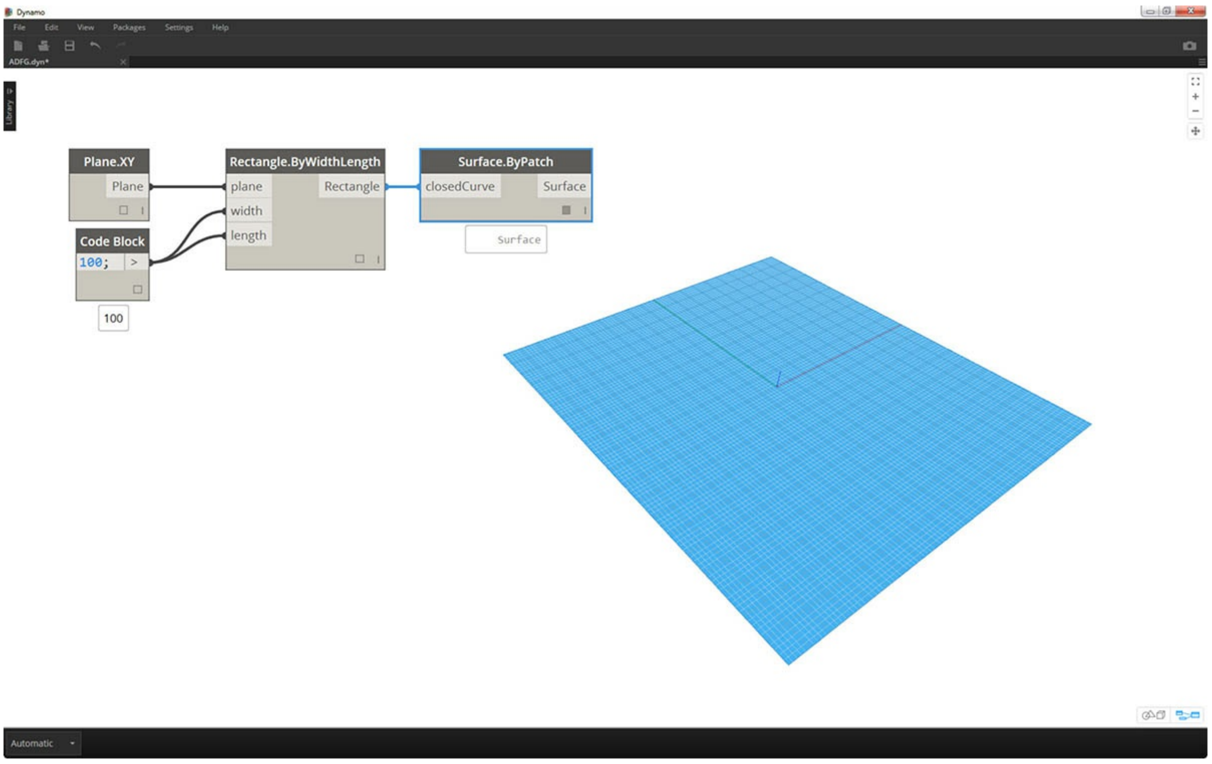
```
dirty clothes
```

1. Define a list of lists.
2. Query a list with single bracket notation.
3. Query an item with double bracket notation.

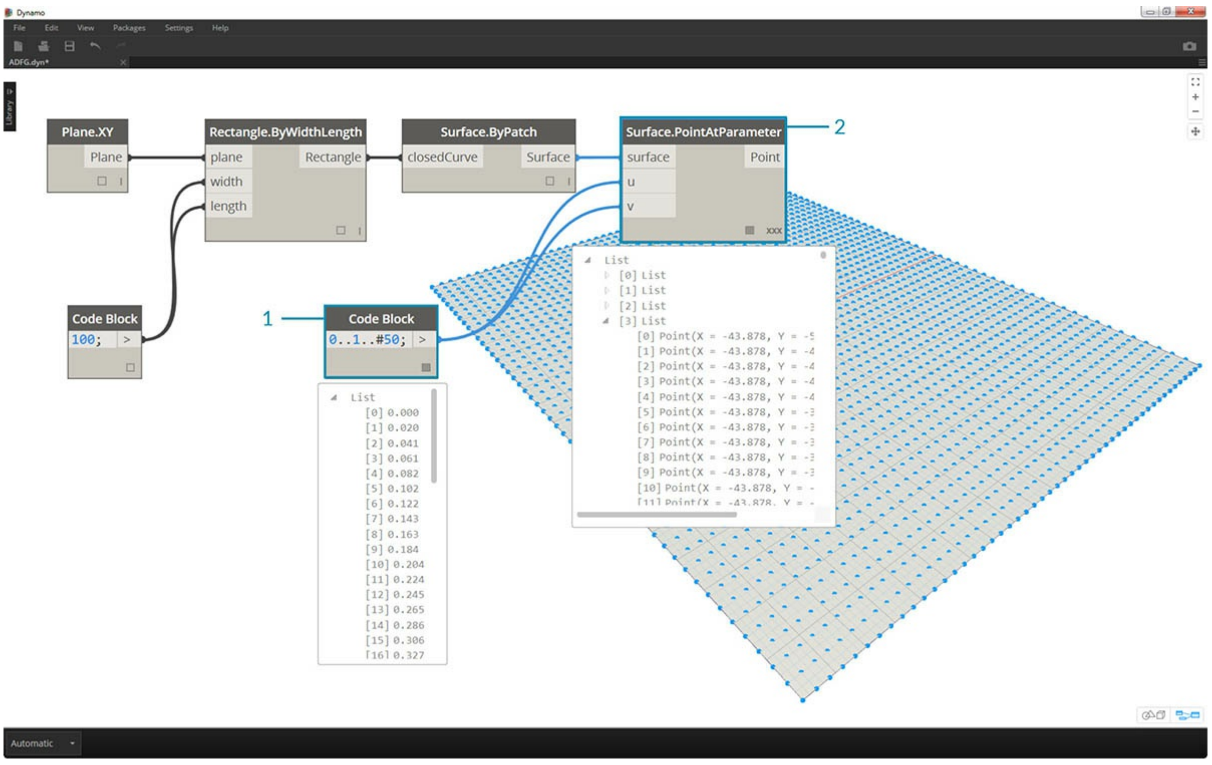
Exercise

Download the example file that accompanies this exercise (Right click and "Save Link As..."). A full list of example files can be found in the Appendix. Obsolete-Nodes_Sine-Surface.dyn

In this exercise, we will flex our new shorthand skills to create a funky-cool eggshell surface defined by ranges and formulas. During this exercise, notice how we use code block and existing Dynamo nodes in tandem: we use the code block for the heavy data lifting while the Dynamo nodes are visually laid out for legibility of the definition.



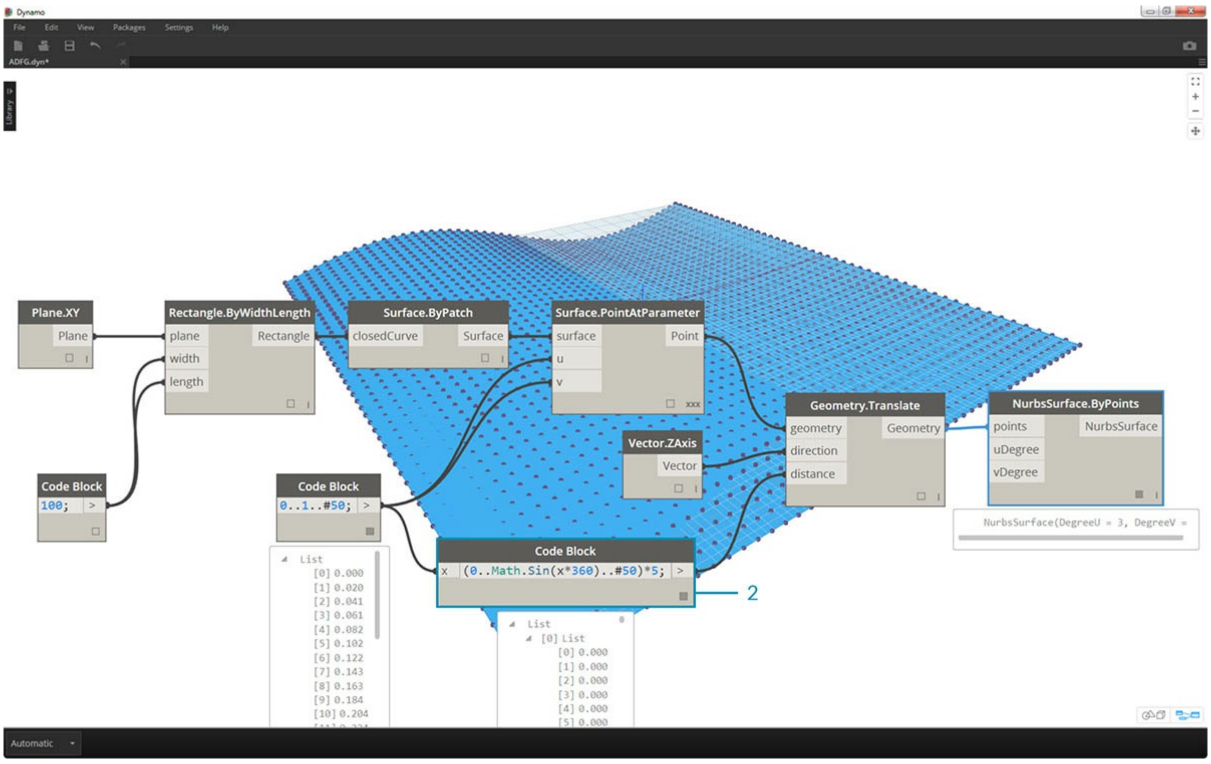
Start by creating a surface by connecting the nodes above. Instead of using a number node to define width and length, double click on the canvas and type `100;` into a code block.



```
List
  [0] List
  [1] List
  [2] List
  [3] List
    [0] Point(X = -43.878, Y = -
    [1] Point(X = -43.878, Y = -
    [2] Point(X = -43.878, Y = -
    [3] Point(X = -43.878, Y = -
    [4] Point(X = -43.878, Y = -
    [5] Point(X = -43.878, Y = -
    [6] Point(X = -43.878, Y = -
    [7] Point(X = -43.878, Y = -
    [8] Point(X = -43.878, Y = -
    [9] Point(X = -43.878, Y = -
    [10] Point(X = -43.878, Y = -
    [11] Point(X = -43.878, Y = -
```

```
List
  [0] 0.000
  [1] 0.020
  [2] 0.041
  [3] 0.061
  [4] 0.082
  [5] 0.102
  [6] 0.122
  [7] 0.143
  [8] 0.163
  [9] 0.184
  [10] 0.204
  [11] 0.224
  [12] 0.245
  [13] 0.265
  [14] 0.286
  [15] 0.306
  [16] 0.327
```

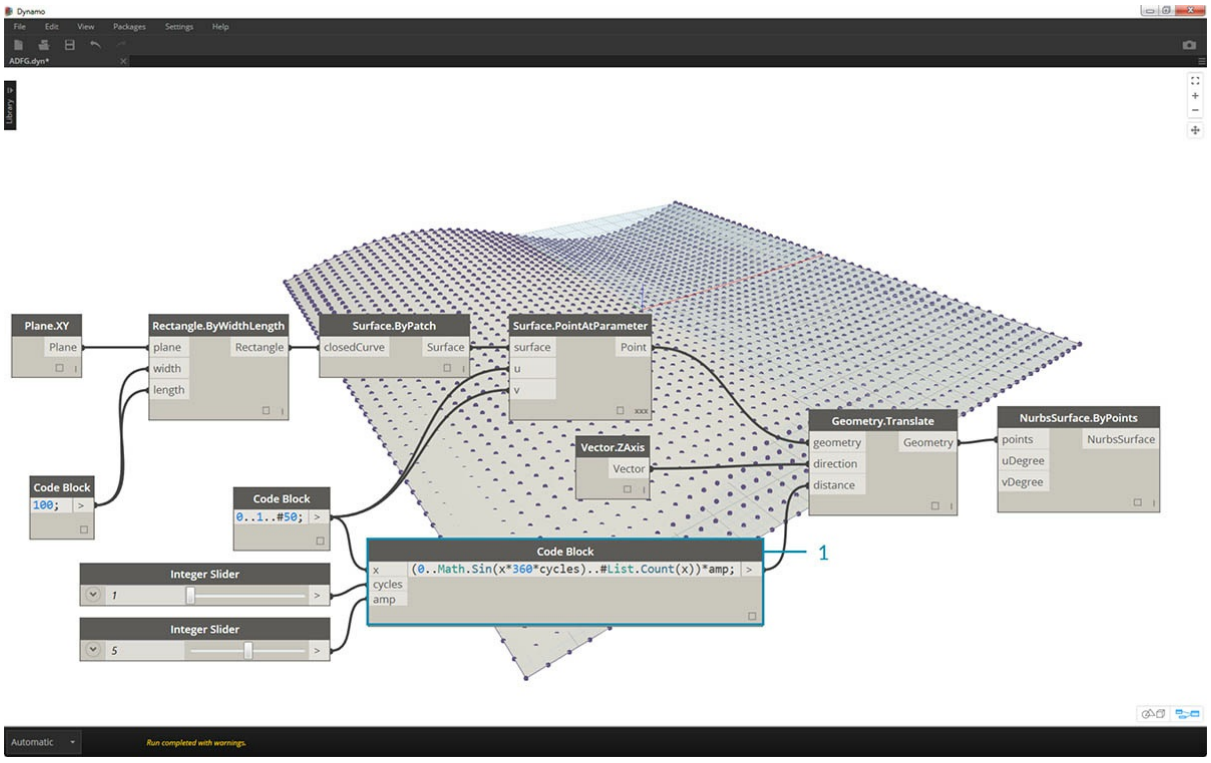
1. Define a range between 0 and 1 with 50 divisions by typing `0..1..#50` into a code block.
2. Connect the range into *Surface.PointAtParameter*, which takes *u* and *v* values between 0 and 1 across the surface. Remember to change the *Lacing* to *Cross Product* by right clicking on the *Surface.PointAtParameter* node.



```
100;
0..1..#50;
(0..Math.Sin(x*360)..#50)*5;
NurbsSurface(DegreeU = 3, DegreeV =
```

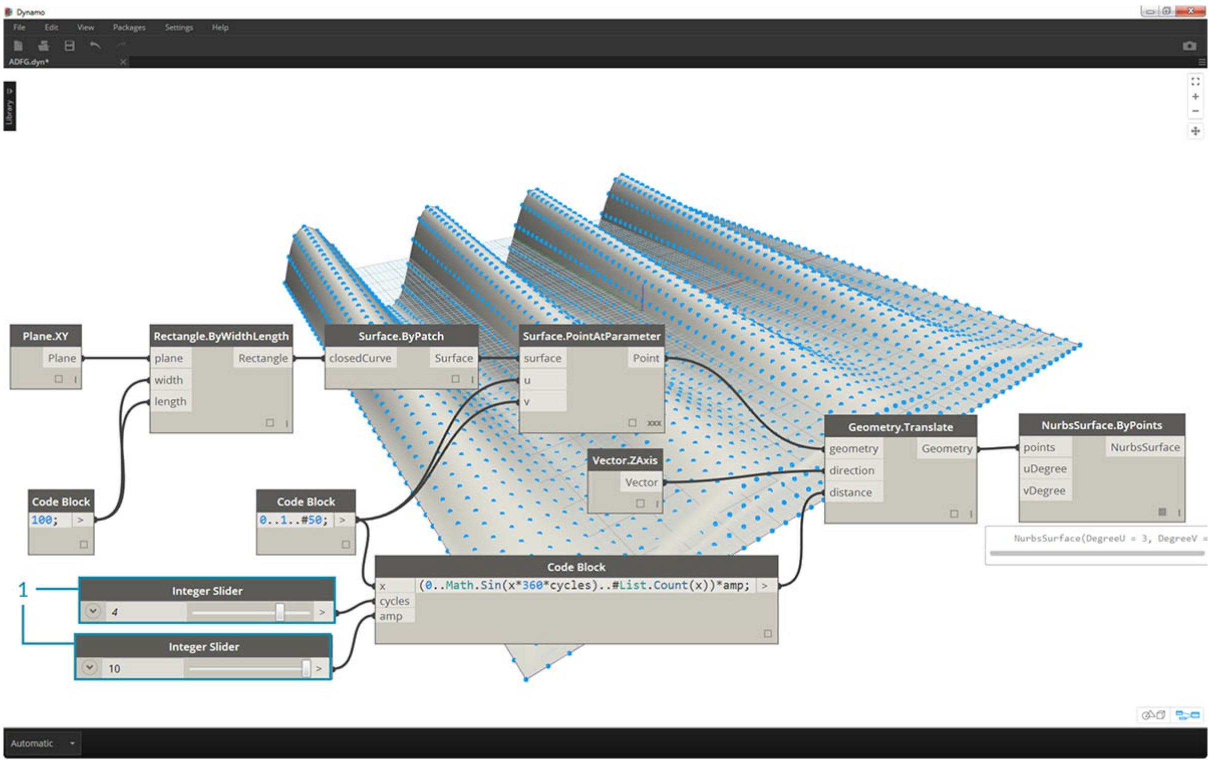
In this step, we employ our first function to move the grid of points up in the Z. This grid will drive a generated surface based on the underlying function.

1. Add the visual nodes to the canvas as shown in the image above.
2. Rather than using a formula node, we use a code block with the line: `(0..Math.Sin(x*360)..#50)*5;`. To quickly break this down, we're defining a range with a formula inside of it. This formula is the Sine function. The sine function receives degree inputs in Dynamo, so in order to get a full sine wave, we multiple our *x* values (this is the range input from 0 to 1) by *360*. Next we want the same number of divisions as control grid points for each row, so we define fifty subdivisions with *#50*. Finally, the multiplier of 5 simply increases the amplitude of translation so that we can see the effect in the Dynamo Preview.



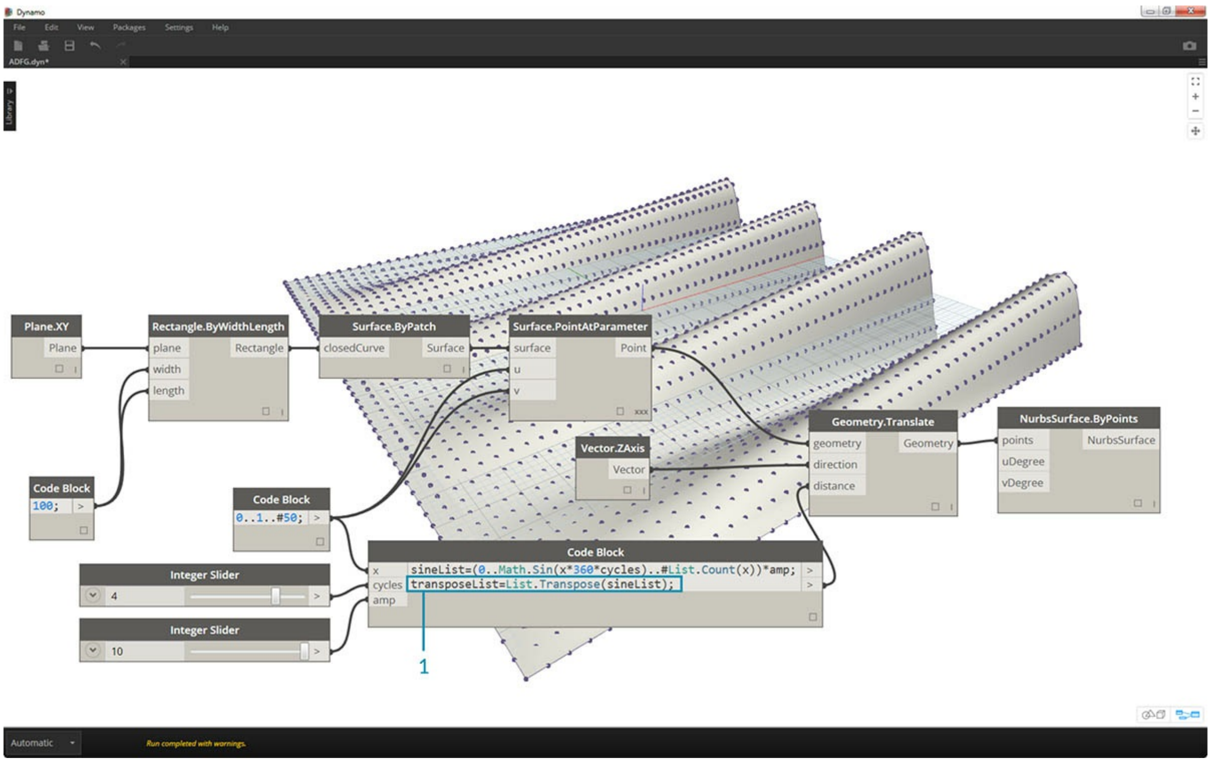
```
(0..Math.Sin(x*360*cycles)..#List.Count(x))*amp;
```

1. While the previous code block worked fine, it wasn't completely parametric. We want to dynamically drive its parameters, so we'll replace the line from the previous step with `(0..Math.Sin(x*360*cycles)..#List.Count(x))*amp;`. This gives us the ability to define these values based on inputs.



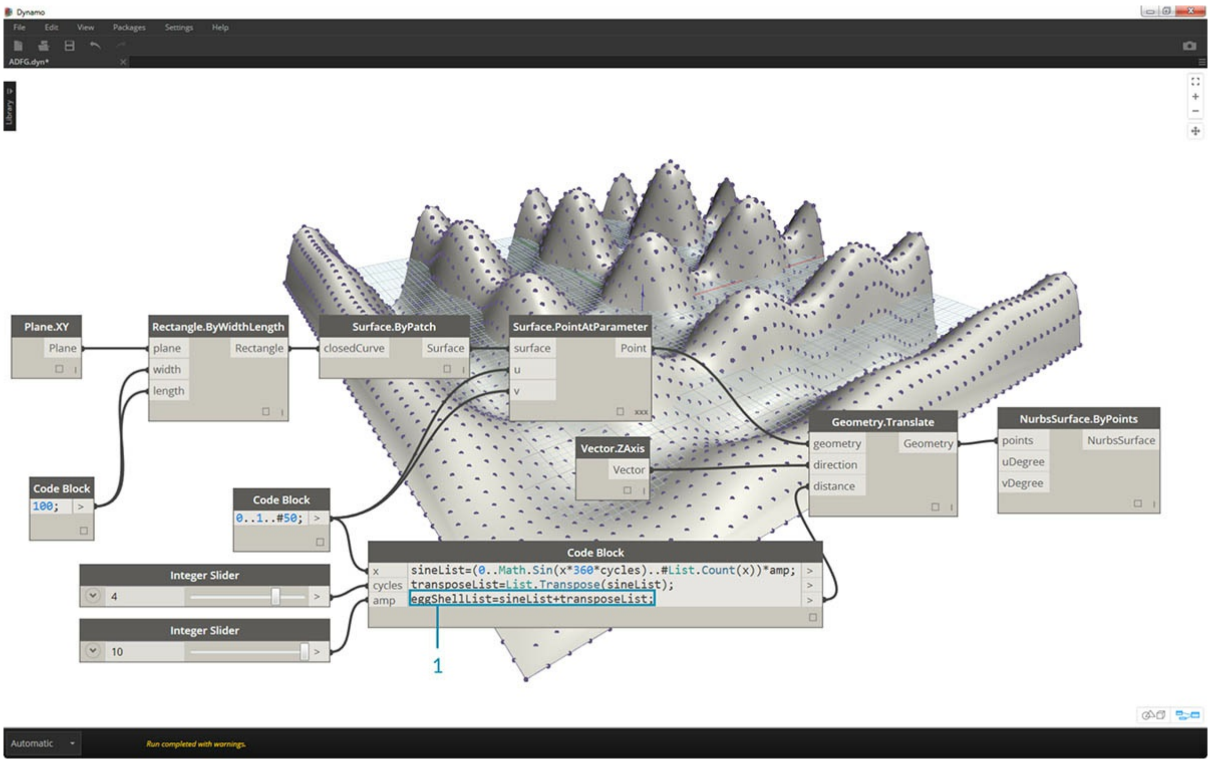
```
100;
0..1..#50;
(0..Math.Sin(x*360*cycles)..#List.Count(x))*amp;
```

1. By changing the sliders (ranging from 0 to 10), we get some interesting results.



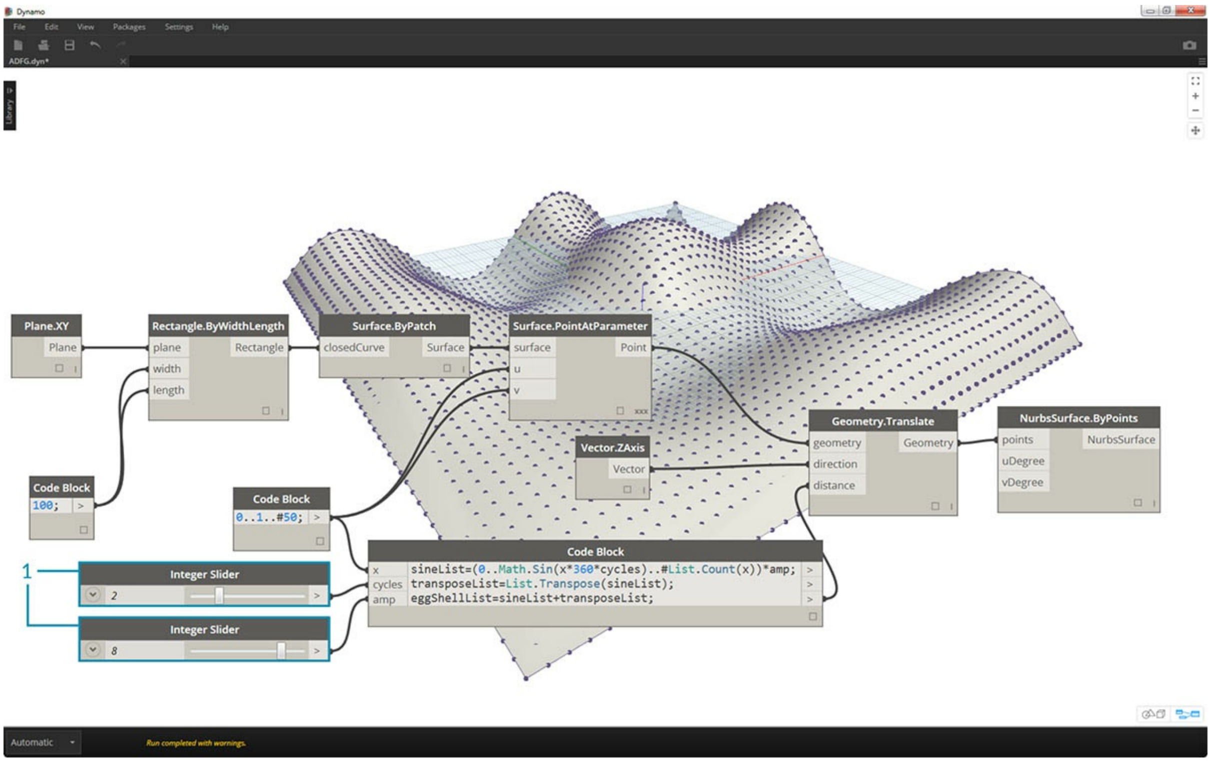
```
sineList=(0..Math.Sin(x*360*cycles)..#List.Count(x))*amp;
transposeList=List.Transpose(sineList);
```

1. By doing a transpose on the number range, we reverse the direction of the curtain wave: `transposeList = List.Transpose(sineList);`



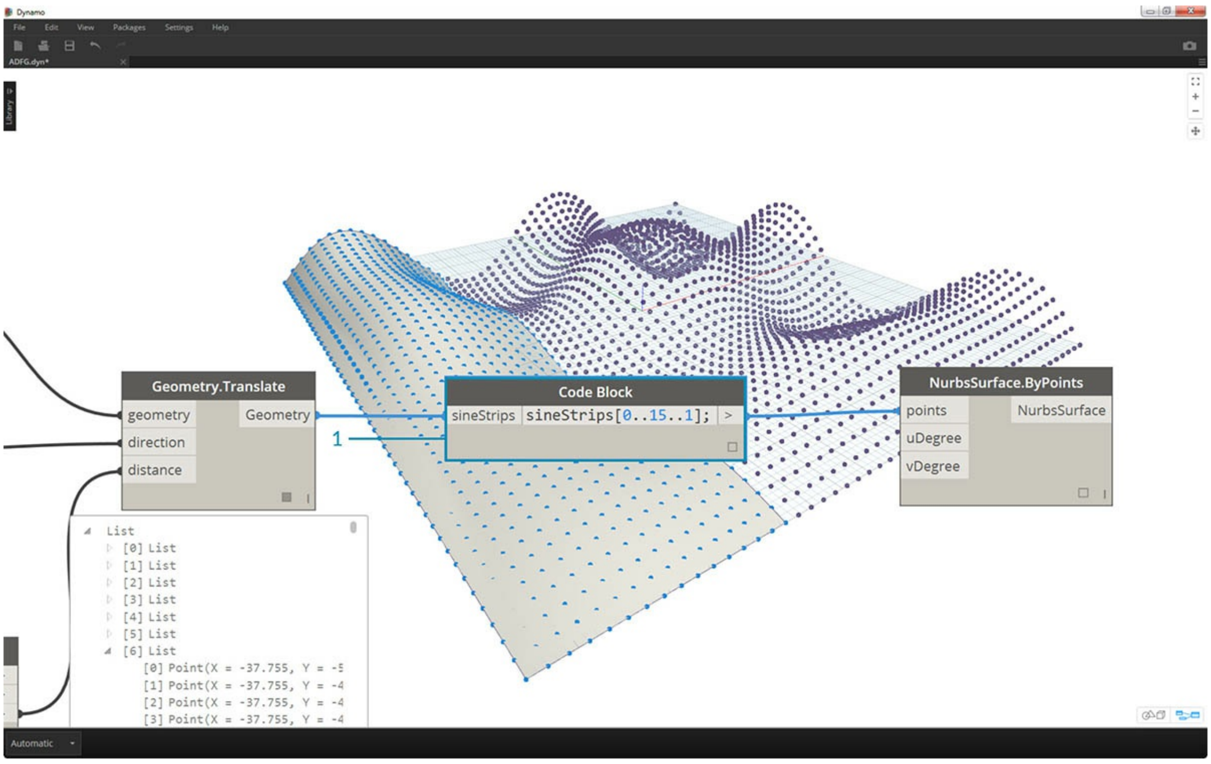
```
sineList=(0..Math.Sin(x*360*cycles)..#List.Count(x))*amp;
transposeList=List.Transpose(sineList);
eggShellList=sineList+transposeList;
```

1. We get a distorted eggshell surface when we add the sineList and the tranposeList: `eggShellList = sineList+transposeList;`



```
sineList=(0..Math.Sin(x*360*cycles)..#List.Count(x))*amp;
transposeList=List.Transpose(sineList);
eggShellList=sineList+transposeList;
```

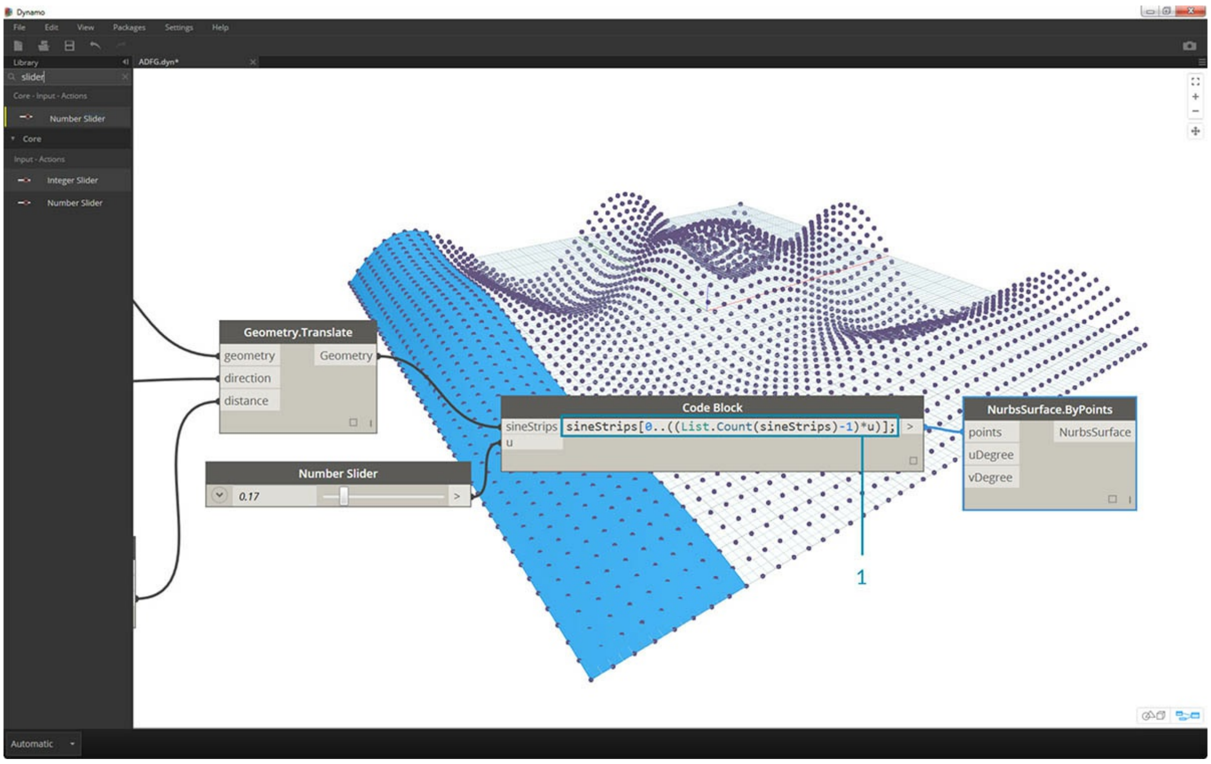
1. Changing the sliders again let's us calm the waters of this algorithm.



```
sineStrips[0..15..1];
```

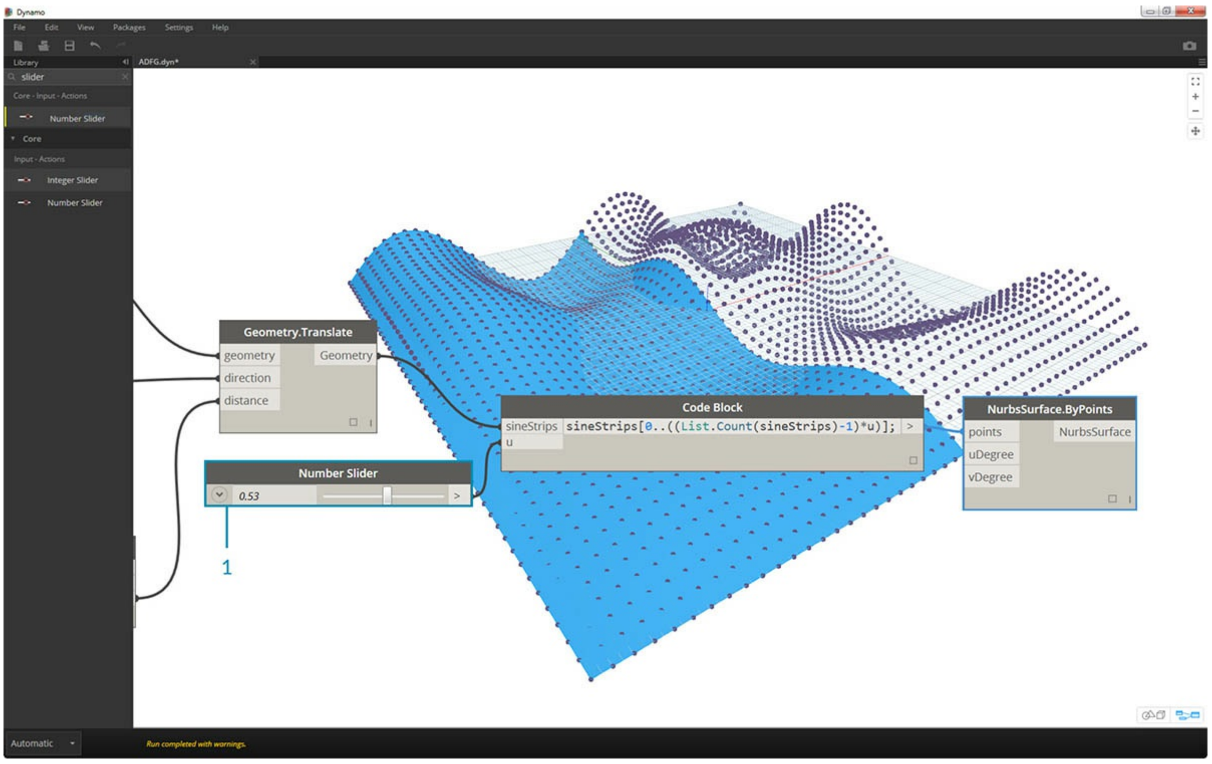
```
List
  [0] List
  [1] List
  [2] List
  [3] List
  [4] List
  [5] List
  [6] List
    [0] Point(X = -37.755, Y = -5
    [1] Point(X = -37.755, Y = -4
    [2] Point(X = -37.755, Y = -4
    [3] Point(X = -37.755, Y = -4
```

1. Last, let's query isolated parts of the data with the code block. To regenerate the surface with a specific range of points, add the code block above between the *Geometry.Translate* and *NurbsSurface.ByPoints* node. This has the line of text: `sineStrips[0..15..1];`. This will select the first 16 rows of points (out of 50). Recreating the surface, we can see that we've generated an isolated portion of the grid of points.



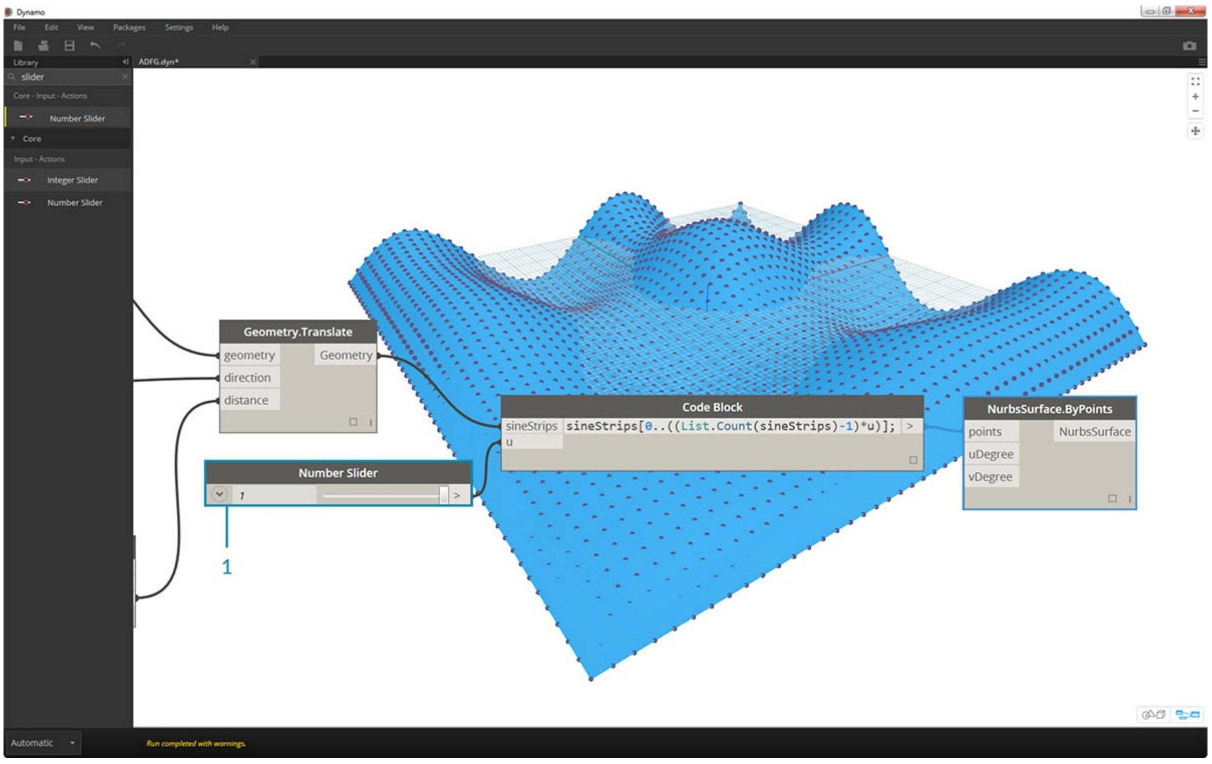
```
sineStrips[0..((List.Count(sineStrips)-1)*u)];
```

1. In the final step, to make this code block more parametric, we drive the query by using a slider ranging from 0 to 1. We do this with this line of code: `sineStrips[0..((List.Count(sineStrips)-1)*u)];`. This may seem confusing, but the line of code gives us a quick way to scale the length of the list into a multiplier between 0 and 1.



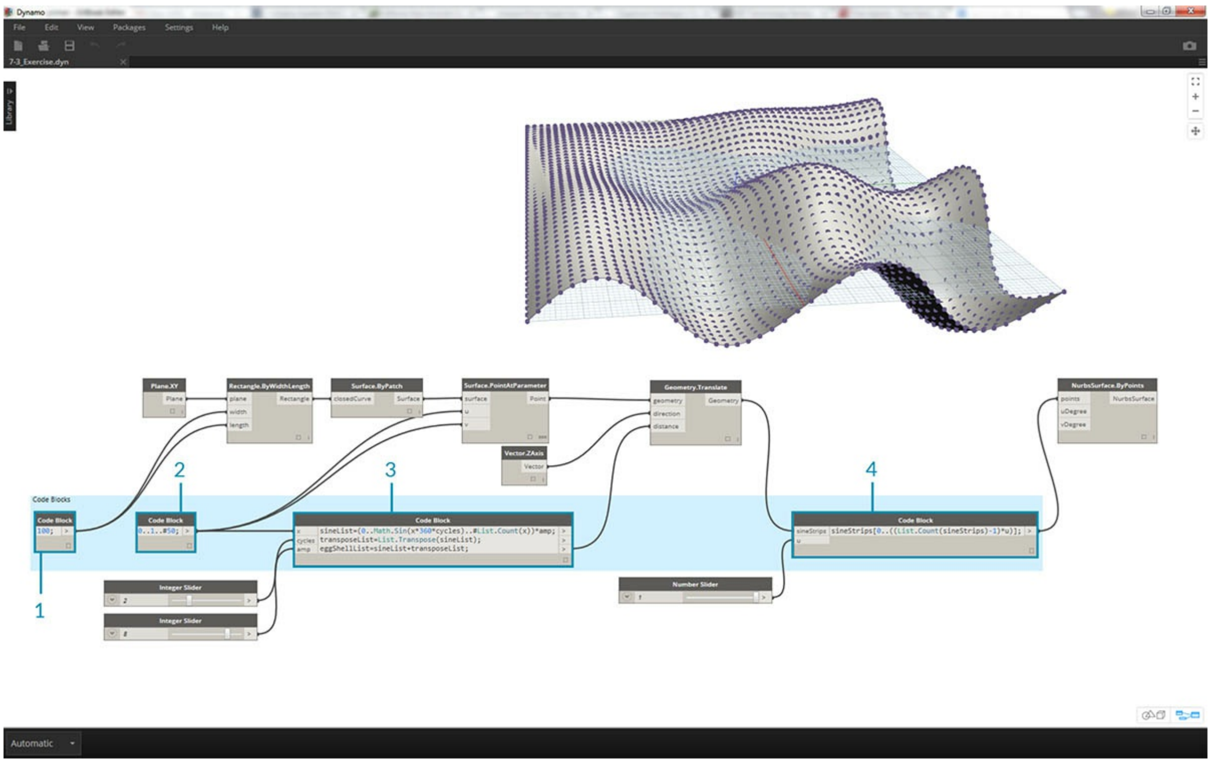
```
sineStrips[0..((List.Count(sineStrips)-1)*u)];
```

1. A value of *.53* on the slider creates a surface just past the midpoint of the grid.



```
sineStrips[0..((List.Count(sineStrips)-1)*u)];
```

1. And as expected, a slider of *1* creates a surface from the full grid of points.



```
100;
```

```
0..1..#50;
```

```
sineList=(0..Math.Sin(x*360*cycles)..#List.Count(x))*amp;
transposeList=List.Transpose(sineList);
eggShellList=sineList+transposeList;
```

```
sineStrips[0..((List.Count(sineStrips)-1)*u)];
```

Looking at the resultant visual graph, we can highlight the code blocks and see each of their functions.

1. The first code block replaces the *Number* node.
2. The second code block replaces the *Number Range* node.
3. The third code block replaces the *Formula* node (as well as *List.Transpose*, *List.Count* and *Number Range*).
4. The fourth code block queries a list of lists, replacing the *List.GetItemAtIndex* node.

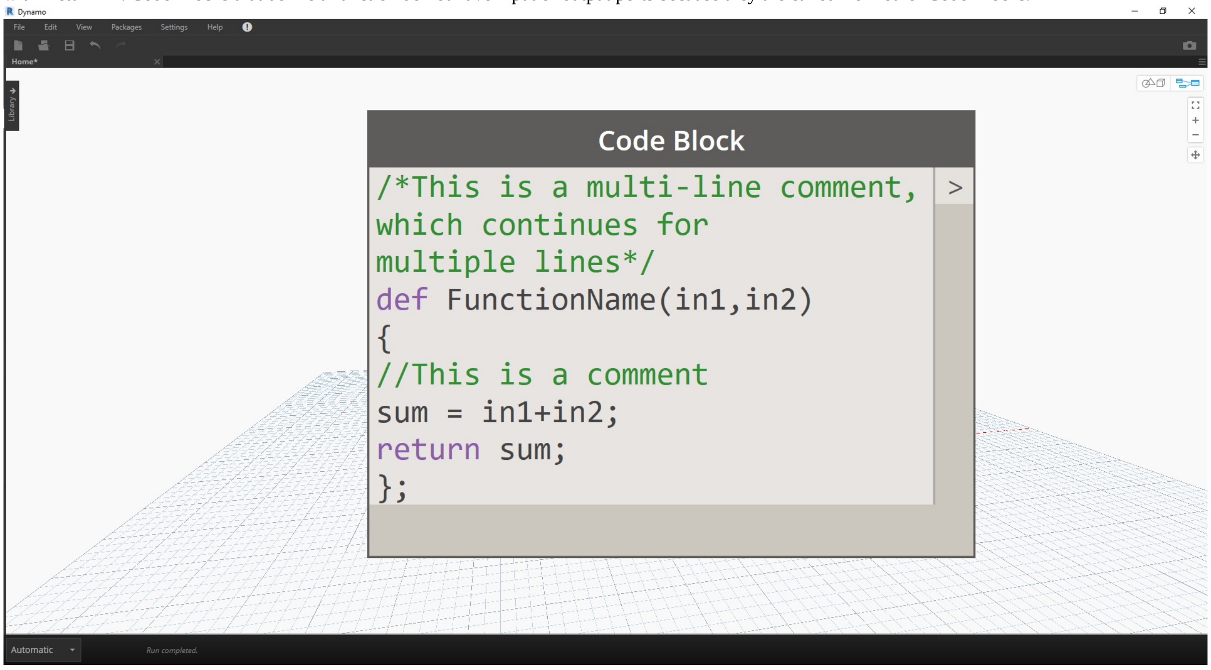
Functions

Code Block Functions

Functions can be created in a code block and recalled elsewhere in a Dynamo definition. This creates another layer of control in a parametric file, and can be viewed as a text-based version of a custom node. In this case, the "parent" code block is readily accessible and can be located anywhere on the graph. No wires needed!

Parent

The first line has the key word “def”, then the function name, then the names of inputs in parentheses. Braces define the body of the function. Return a value with “return =”. Code Blocks that define a function do not have input or output ports because they are called from other Code Blocks.

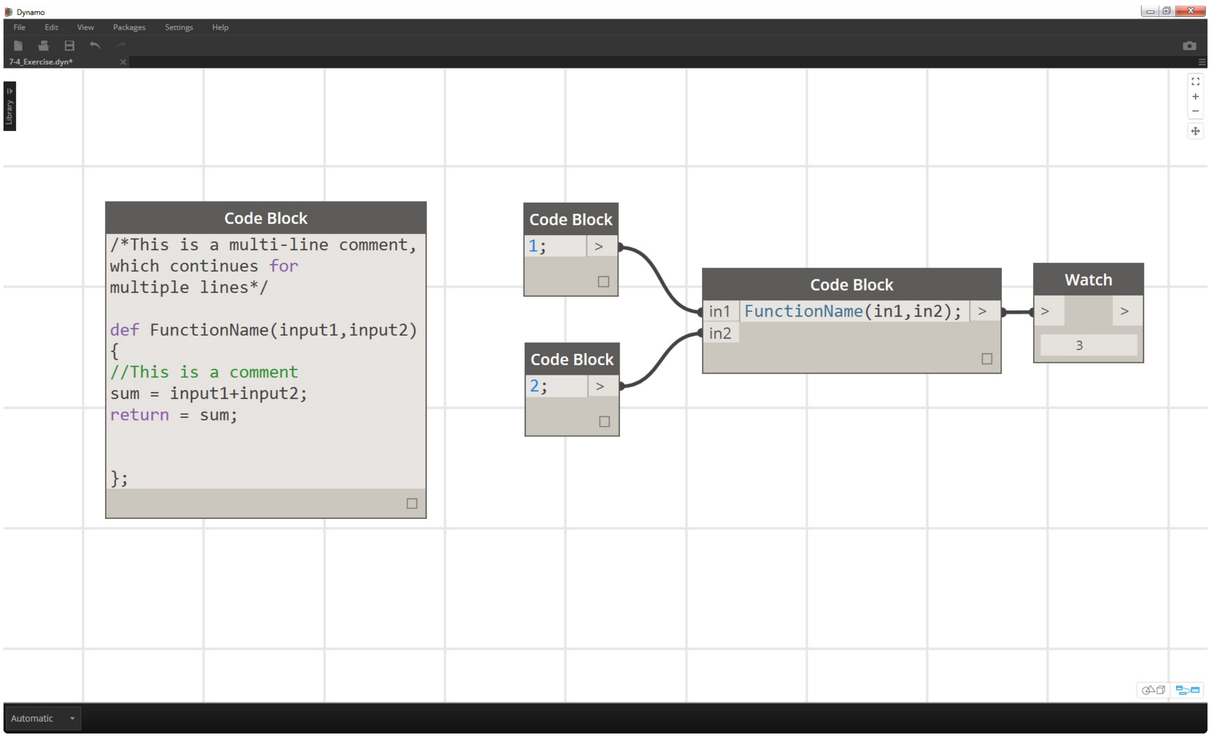


```
/*This is a multi-line comment,
which continues for
multiple lines*/
def FunctionName(in1,in2)
{
//This is a comment
sum = in1+in2;
return sum;
};
```

```
/*This is a multi-line comment,
which continues for
multiple lines*/
def FunctionName(in1,in2)
{
//This is a comment
sum = in1+in2;
return sum;
};
```

Children

Call the function with another Code Block in the same file by giving the name and the same number of arguments. It works just like the out-of-the-box nodes in your library.



```
/*This is a multi-line comment,
which continues for
multiple lines*/

def FunctionName(input1,input2)
{
//This is a comment
sum = input1+input2;
return = sum;


};
```

```
1;
```

```
2;
```

```
FunctionName(in1,in2);
```

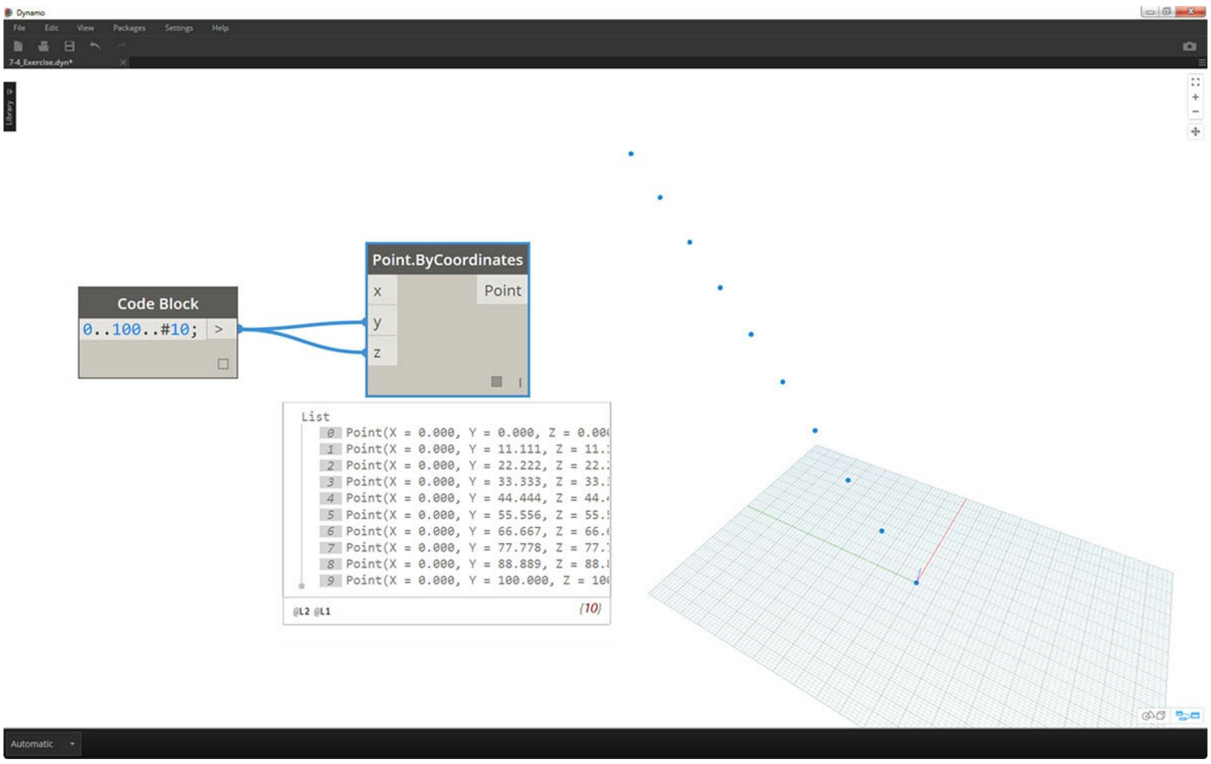
```
3
```

```
FunctionName(in1,in2);
```

Exercise

Download the example file that accompanies this exercise (Right click and "Save Link As..."). A full list of example files can be found in the Appendix. [Functions_SphereByZ.dyn]

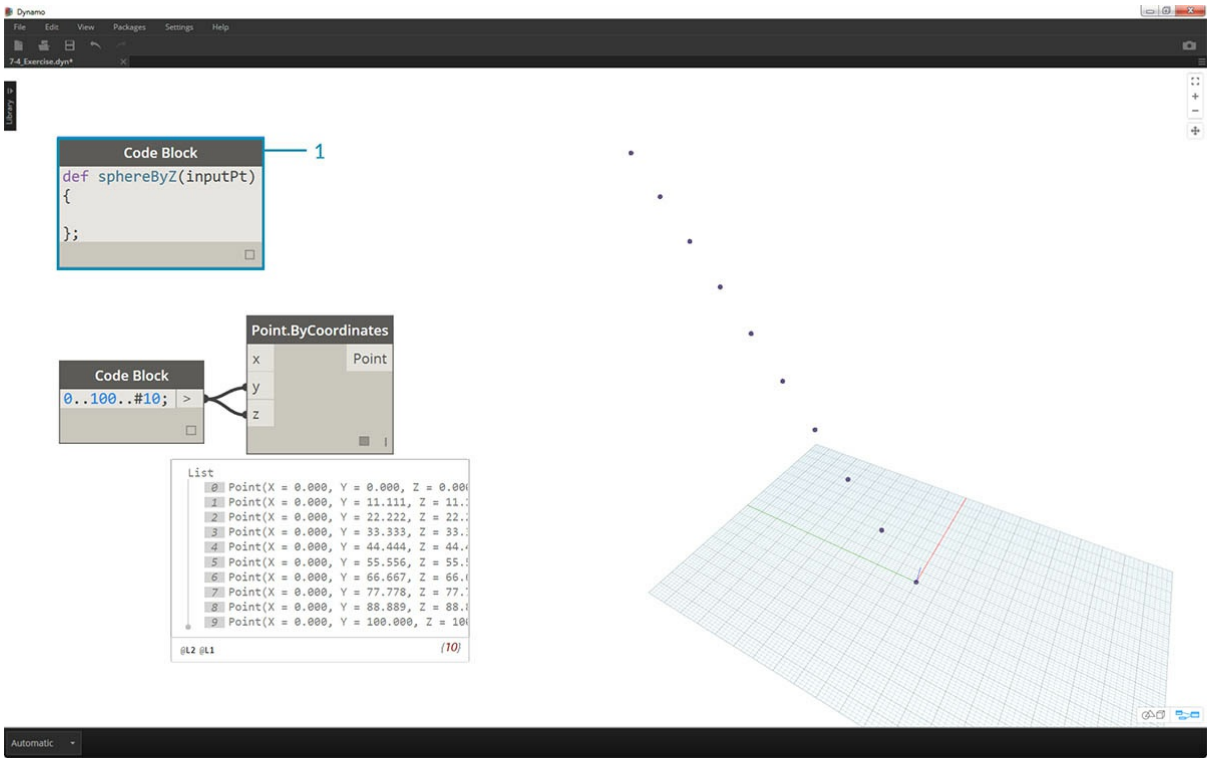
In this exercise, we will make a generic definition that will create spheres from an input list of points. The radius of these spheres are driven by the Z property of each point.



```
0..100..#10;
```

```
List
  0 Point(X = 0.000, Y = 0.000, Z = 0.000
  1 Point(X = 0.000, Y = 11.111, Z = 11.1
  2 Point(X = 0.000, Y = 22.222, Z = 22.2
  3 Point(X = 0.000, Y = 33.333, Z = 33.3
  4 Point(X = 0.000, Y = 44.444, Z = 44.4
  5 Point(X = 0.000, Y = 55.556, Z = 55.5
  6 Point(X = 0.000, Y = 66.667, Z = 66.6
  7 Point(X = 0.000, Y = 77.778, Z = 77.7
  8 Point(X = 0.000, Y = 88.889, Z = 88.8
  9 Point(X = 0.000, Y = 100.000, Z = 100
@L2 @L1                              {10}
```

Let's begin with a number range of ten values spanning from 0 to 100. Plug these into a *Point.ByCoordinates* nodes to create a diagonal line.



```
def sphereByZ(inputPt)
{

};
```

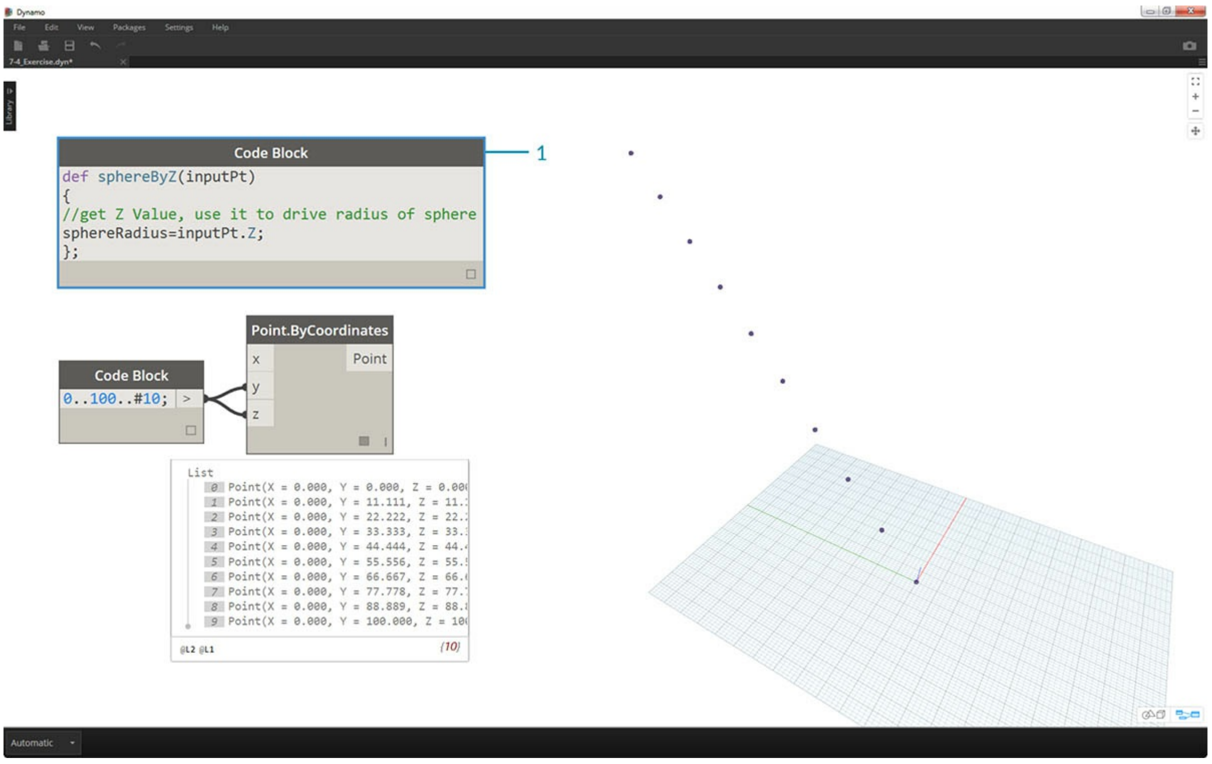
```
0..100..#10;
```

```
List
 0 Point(X = 0.000, Y = 0.000, Z = 0.00
 1 Point(X = 0.000, Y = 11.111, Z = 11.
 2 Point(X = 0.000, Y = 22.222, Z = 22.
 3 Point(X = 0.000, Y = 33.333, Z = 33.
 4 Point(X = 0.000, Y = 44.444, Z = 44.
 5 Point(X = 0.000, Y = 55.556, Z = 55.
 6 Point(X = 0.000, Y = 66.667, Z = 66.
 7 Point(X = 0.000, Y = 77.778, Z = 77.
 8 Point(X = 0.000, Y = 88.889, Z = 88.
 9 Point(X = 0.000, Y = 100.000, Z = 10
@L2 @L1                          {10}
```

1. Create a *code block* and introduce our definition by using the line of code:

   ```
   def sphereByZ(inputPt){
   };
   ```

   The *inputPt* is the name we've given to represent the points that will drive the function. As of now, the function isn't doing anything, but we'll build up this function in the steps to come.



```
def sphereByZ(inputPt)
{
//get Z Value, use it to drive radius of sphere
sphereRadius=inputPt.Z;
};
```

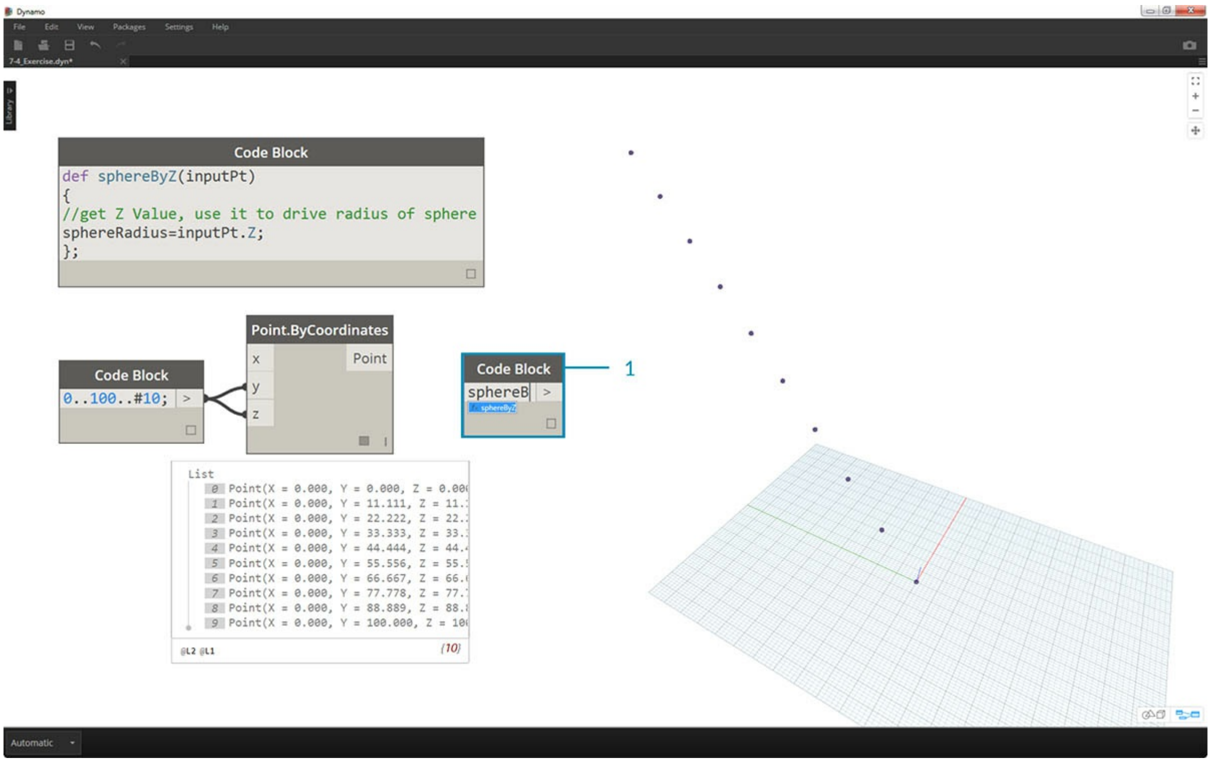
```
0..100..#10;
```

```
List
 0 Point(X = 0.000, Y = 0.000, Z = 0.00
 1 Point(X = 0.000, Y = 11.111, Z = 11.
 2 Point(X = 0.000, Y = 22.222, Z = 22.
 3 Point(X = 0.000, Y = 33.333, Z = 33.
 4 Point(X = 0.000, Y = 44.444, Z = 44.
 5 Point(X = 0.000, Y = 55.556, Z = 55.
 6 Point(X = 0.000, Y = 66.667, Z = 66.
 7 Point(X = 0.000, Y = 77.778, Z = 77.
 8 Point(X = 0.000, Y = 88.889, Z = 88.
 9 Point(X = 0.000, Y = 100.000, Z = 10
@L2 @L1                          {10}
```

1. Adding to the *code block* function, we place a comment and a *sphereRadius* variable which queries the *Z* position of each point. Remember, *inputPt.Z* does not need parenetheses as a method. This is a *query* of an existing element's properties, so no inputs are necessary:

   ```
   def sphereByZ(inputPt,radiusRatio)
   {
   //get Z Value, use it to drive radius of sphere
   ```

```
sphereRadius=inputPt.Z;
};
```



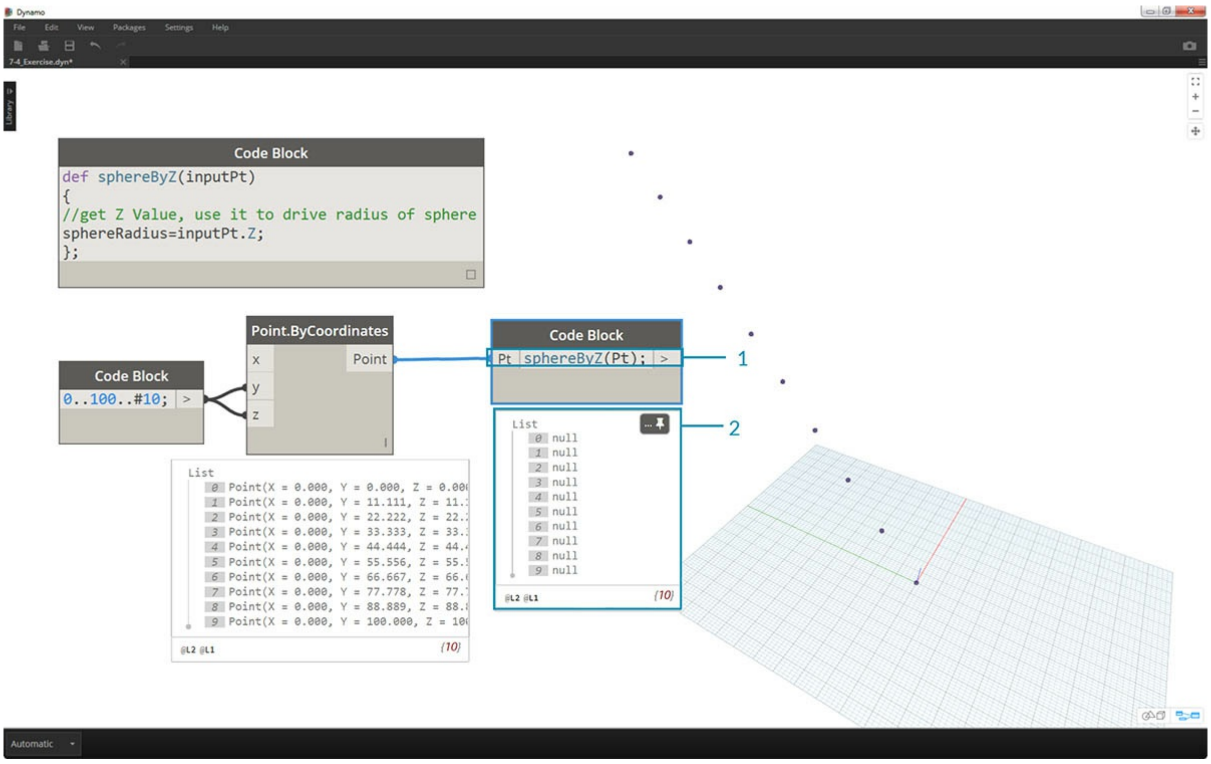
```
def sphereByZ(inputPt)
{
//get Z Value, use it to drive radius of sphere
sphereRadius=inputPt.Z;
};
```

```
0..100..#10;
```

```
sphereB
```

```
List
  0 Point(X = 0.000, Y = 0.000, Z = 0.00
  1 Point(X = 0.000, Y = 11.111, Z = 11.
  2 Point(X = 0.000, Y = 22.222, Z = 22.
  3 Point(X = 0.000, Y = 33.333, Z = 33.
  4 Point(X = 0.000, Y = 44.444, Z = 44.
  5 Point(X = 0.000, Y = 55.556, Z = 55.
  6 Point(X = 0.000, Y = 66.667, Z = 66.
  7 Point(X = 0.000, Y = 77.778, Z = 77.
  8 Point(X = 0.000, Y = 88.889, Z = 88.
  9 Point(X = 0.000, Y = 100.000, Z = 10
@L2 @L1                            {10}
```

1. Now, let's recall the function we've created in another *code block*. If we double-click on the canvas to create a new *code block*, and type in *sphereB*, we notice that Dynamo suggest the *sphereByZ* function that we've defined. Your function has been added to the intellisense library! Pretty cool.



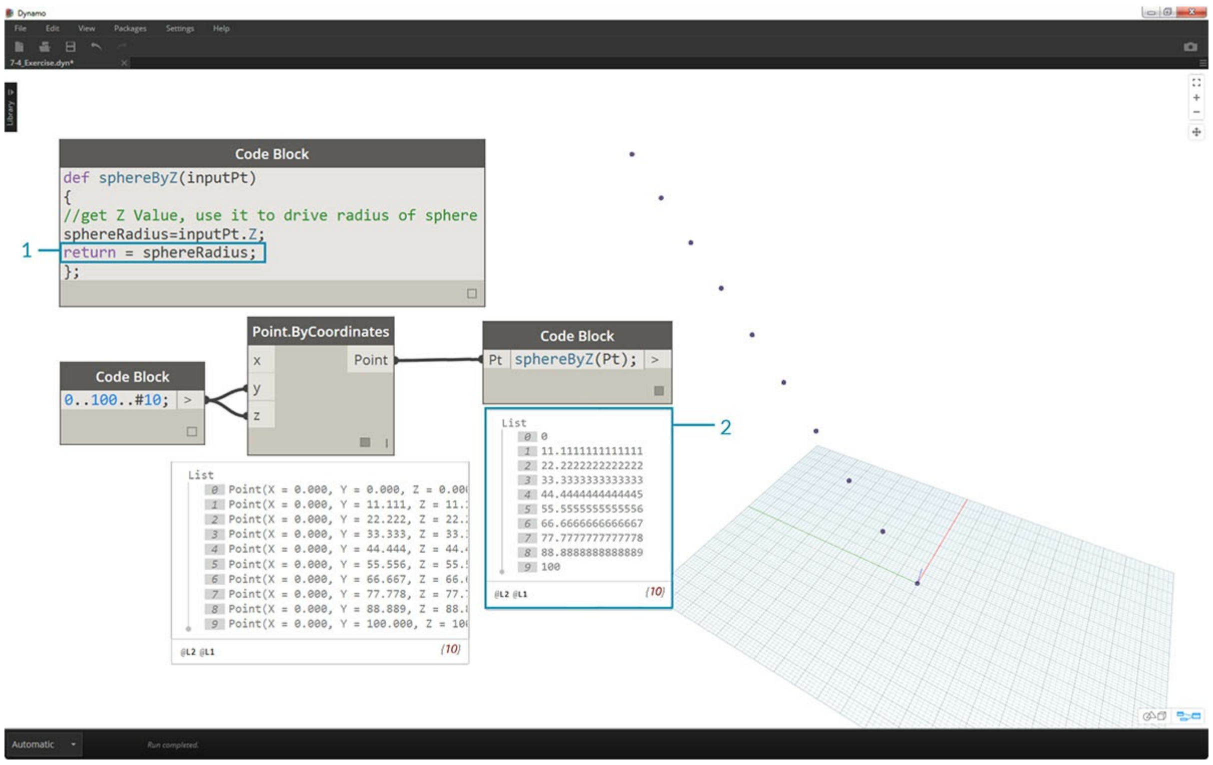
```
Pt  sphereByZ(Pt);
```

```
List
  0 null
  1 null
  2 null
  3 null
  4 null
  5 null
  6 null
  7 null
  8 null
  9 null
@L2 @L1        {10}
```

1. Now we call the function and create a variable called *Pt* to plug in the points created in the earlier steps:

   ```
   sphereByZ(Pt)
   ```

2. We notice from the output that we have all null values. Why is this? When we defined the function, we are calculating the *sphereRadius* variable, but we did not define what the function should *return* as an *output*. We can fix this in the next step.



```
def sphereByZ(inputPt)
{
//get Z Value, use it to drive radius of sphere
sphereRadius=inputPt.Z;
return = sphereRadius;
};
```

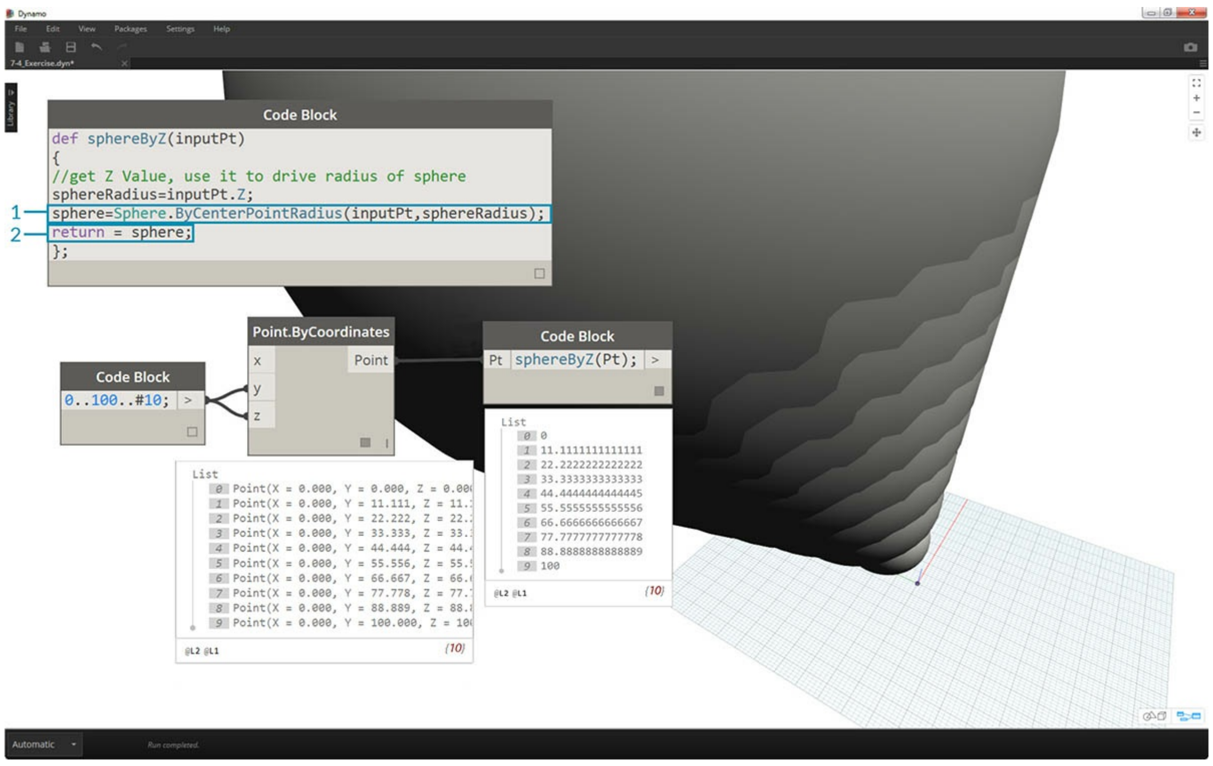
```
0..100..#10;
```

```
sphereByZ(Pt);
```

```
List
 0 Point(X = 0.000, Y = 0.000, Z = 0.00
 1 Point(X = 0.000, Y = 11.111, Z = 11.
 2 Point(X = 0.000, Y = 22.222, Z = 22.
 3 Point(X = 0.000, Y = 33.333, Z = 33.
 4 Point(X = 0.000, Y = 44.444, Z = 44.
 5 Point(X = 0.000, Y = 55.556, Z = 55.
 6 Point(X = 0.000, Y = 66.667, Z = 66.
 7 Point(X = 0.000, Y = 77.778, Z = 77.
 8 Point(X = 0.000, Y = 88.889, Z = 88.
 9 Point(X = 0.000, Y = 100.000, Z = 10
@L2 @L1                             {10}
```

```
List
 0 0
 1 11.1111111111111
 2 22.2222222222222
 3 33.3333333333333
 4 44.4444444444445
 5 55.5555555555556
 6 66.6666666666667
 7 77.7777777777778
 8 88.8888888888889
 9 100
@L2 @L1              {10}
```

1. An important step, we need to define the output of the function by adding the line `return = sphereRadius;` to the *sphereByZ* function.
2. Now we see that the output of the *code block* gives us the Z coordinates of each point.



```
def sphereByZ(inputPt)
{
//get Z Value, use it to drive radius of sphere
sphereRadius=inputPt.Z;
sphere=Sphere.ByCenterPointRadius(inputPt,sphereRadius);
return = sphere;
};
```

```
0..100..#10;
```

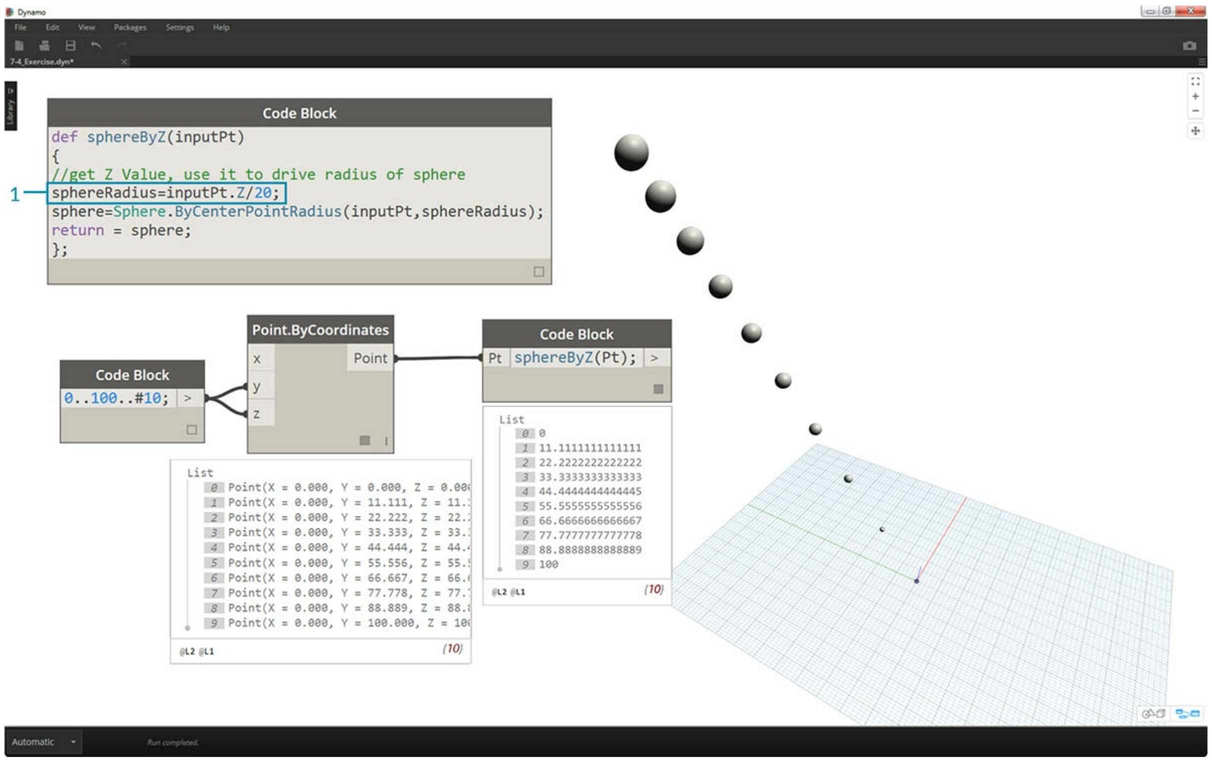
```
sphereByZ(Pt);
```

```
List
 0 Point(X = 0.000, Y = 0.000, Z = 0.00
 1 Point(X = 0.000, Y = 11.111, Z = 11.
 2 Point(X = 0.000, Y = 22.222, Z = 22.
 3 Point(X = 0.000, Y = 33.333, Z = 33.
 4 Point(X = 0.000, Y = 44.444, Z = 44.
 5 Point(X = 0.000, Y = 55.556, Z = 55.
 6 Point(X = 0.000, Y = 66.667, Z = 66.
 7 Point(X = 0.000, Y = 77.778, Z = 77.
 8 Point(X = 0.000, Y = 88.889, Z = 88.
 9 Point(X = 0.000, Y = 100.000, Z = 10
@L2 @L1                             {10}
```

```
List
 0 0
 1 11.1111111111111
 2 22.2222222222222
 3 33.3333333333333
 4 44.4444444444445
 5 55.5555555555556
 6 66.6666666666667
 7 77.7777777777778
 8 88.8888888888889
 9 100
@L2 @L1              {10}
```

Let's create actual spheres now by editing the *Parent* function.

1. We first define a sphere with the line of code: `sphere=Sphere.ByCenterPointRadius(inputPt,sphereRadius);`
2. Next, we change the return value to be the *sphere* instead of the *sphereRadius*: `return = sphere;`. This gives us some giant spheres in our Dynamo preview!



```
def sphereByZ(inputPt)
{
//get Z Value, use it to drive radius of sphere
sphereRadius=inputPt.Z/20;
sphere=Sphere.ByCenterPointRadius(inputPt,sphereRadius);
return = sphere;
};
```

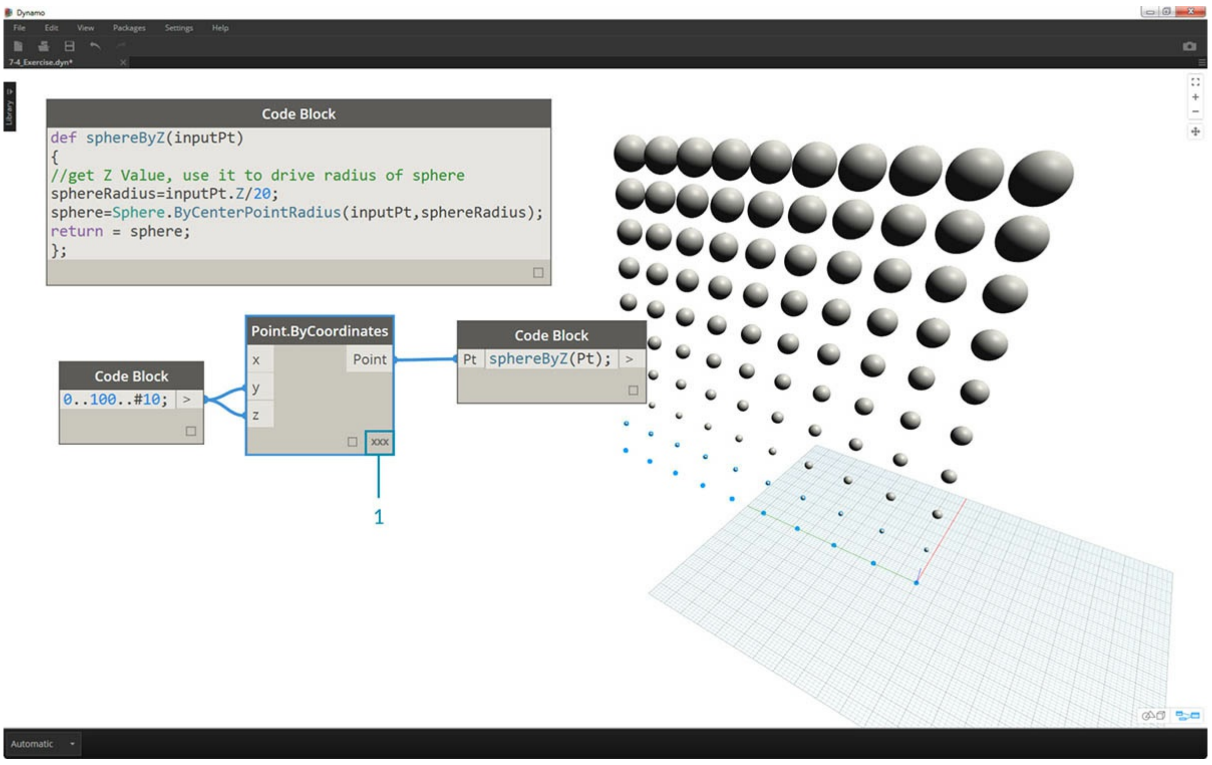
```
0..100..#10;
```

```
sphereByZ(Pt);
```

```
List
  0 Point(X = 0.000, Y = 0.000, Z = 0.00
  1 Point(X = 0.000, Y = 11.111, Z = 11.
  2 Point(X = 0.000, Y = 22.222, Z = 22.
  3 Point(X = 0.000, Y = 33.333, Z = 33.
  4 Point(X = 0.000, Y = 44.444, Z = 44.
  5 Point(X = 0.000, Y = 55.556, Z = 55.
  6 Point(X = 0.000, Y = 66.667, Z = 66.
  7 Point(X = 0.000, Y = 77.778, Z = 77.
  8 Point(X = 0.000, Y = 88.889, Z = 88.
  9 Point(X = 0.000, Y = 100.000, Z = 10
@L2 @L1                              {10}
```

```
List
  0 0
  1 11.1111111111111
  2 22.2222222222222
  3 33.3333333333333
  4 44.4444444444445
  5 55.5555555555556
  6 66.6666666666667
  7 77.7777777777778
  8 88.8888888888889
  9 100
@L2 @L1                {10}
```

1. To temper the size of these spheres, let's update the *sphereRadius* value by adding a divider: `sphereRadius = inputPt.Z/20;`. Now we can see the separate spheres and start to make sense of the relationship between radius and Z value.

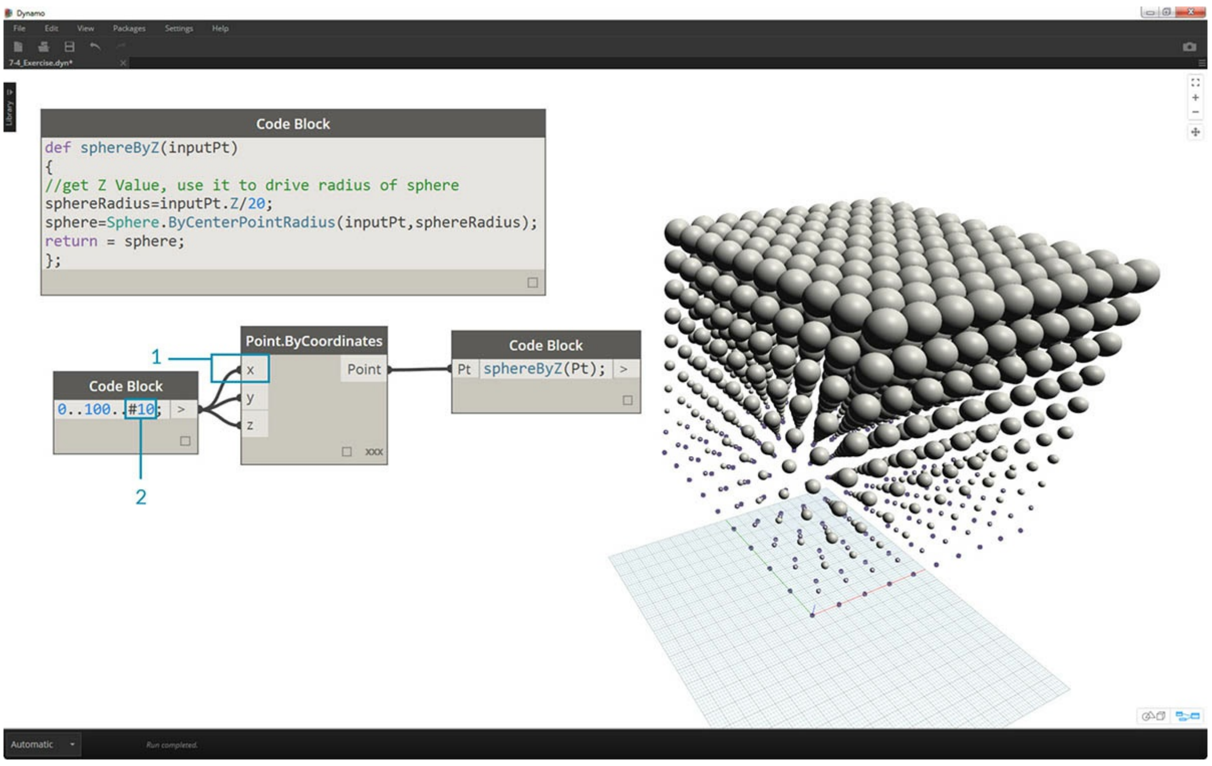


```
def sphereByZ(inputPt)
{
//get Z Value, use it to drive radius of sphere
sphereRadius=inputPt.Z/20;
sphere=Sphere.ByCenterPointRadius(inputPt,sphereRadius);
return = sphere;
};
```

```
0..100..#10;
```

```
sphereByZ(Pt);
```

1. On the *Point.ByCoordinates* node, by changing the lacing from *Shortest List* to *Cross Product*, we create a grid of points. The *sphereByZ* function is still in full effect, so the points all create spheres with radii based on Z values.

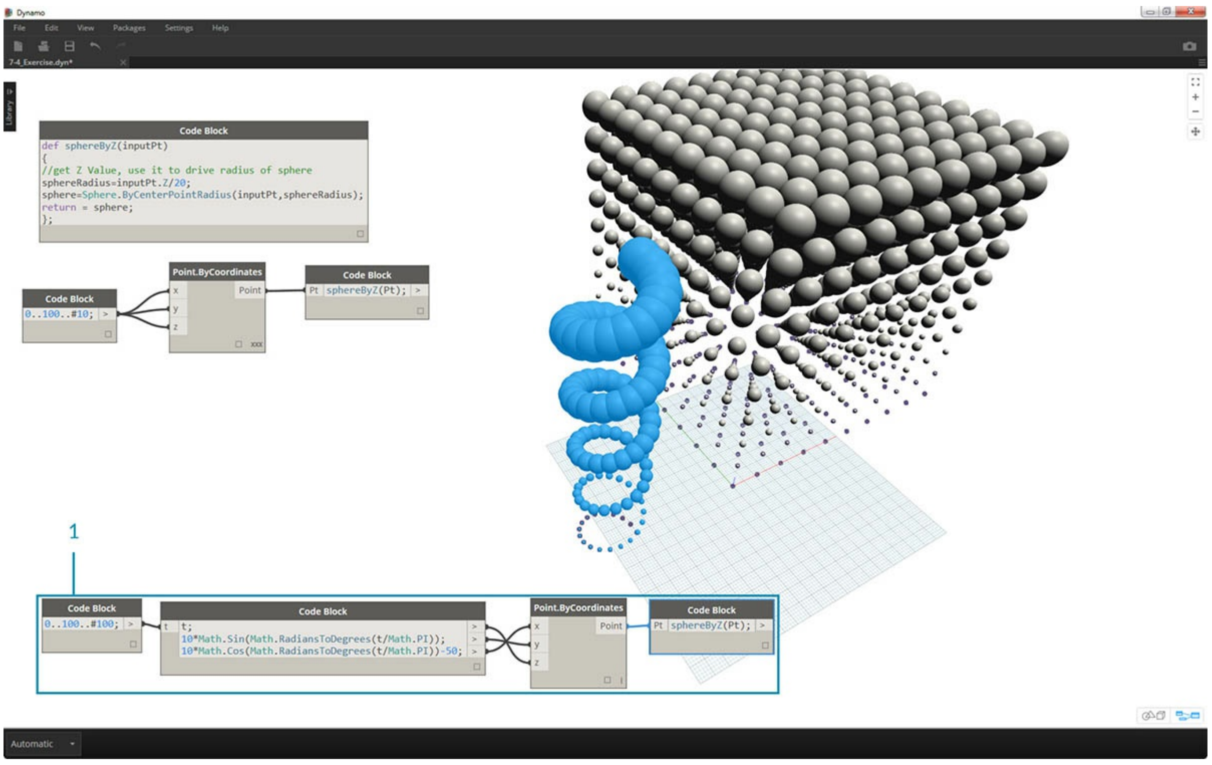


```
def sphereByZ(inputPt)
{
//get Z Value, use it to drive radius of sphere
sphereRadius=inputPt.Z/20;
sphere=Sphere.ByCenterPointRadius(inputPt,sphereRadius);
return = sphere;
};
```

```
0..100..#10;
```

```
sphereByZ(Pt);
```

1. And just to test the waters, we plug the original list of numbers into the X input for *Point.ByCoordinates*. We now have a cube of spheres.
2. Note: if this takes a long time to calculate on your computer, try to change *#10* to something like *#5*.

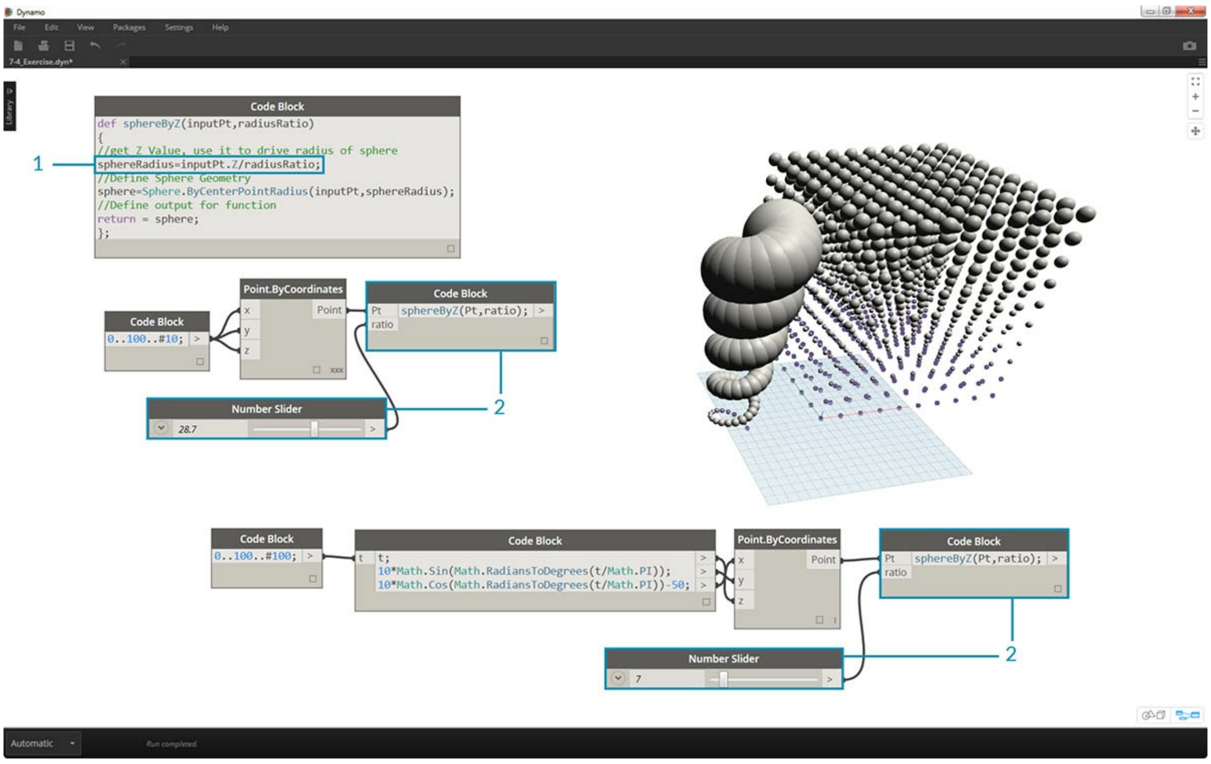


```
0..100..#100;
```

```
t;
10*Math.Sin(Math.RadiansToDegrees(t/Math.PI));
10*Math.Cos(Math.RadiansToDegrees(t/Math.PI))-50;
```

```
sphereByZ(Pt);
```

1. Remember, the *sphereByZ* function we've created is a generic function, so we can recall the helix from an earlier lesson and apply the function to it.



Code Block

```
def sphereByZ(inputPt,radiusRatio)
{
//get Z Value, use it to drive radius of sphere
sphereRadius=inputPt.Z/radiusRatio;
//Define Sphere Geometry
sphere=Sphere.ByCenterPointRadius(inputPt,sphereRadius);
//Define output for function
return = sphere;
};
```

Code Block

```
0..100..#10;
```

Point.ByCoordinates (x, y, z → Point)

Code Block (Pt, ratio)

```
sphereByZ(Pt,ratio);
```

Number Slider: 28.7

Code Block

```
0..100..#100;
```

Code Block (t)

```
t;
10*Math.Sin(Math.RadiansToDegrees(t/Math.PI));
10*Math.Cos(Math.RadiansToDegrees(t/Math.PI))-50;
```

Point.ByCoordinates (x, y, z → Point)

Code Block (Pt, ratio)

```
sphereByZ(Pt,ratio);
```

Number Slider: 7

One final step: let's drive the radius ratio with a user defined parameter. To do this, we need to create a new input for the function and also replace the *20* divider with a parameter.

1. Update the *sphereByZ* definition to:

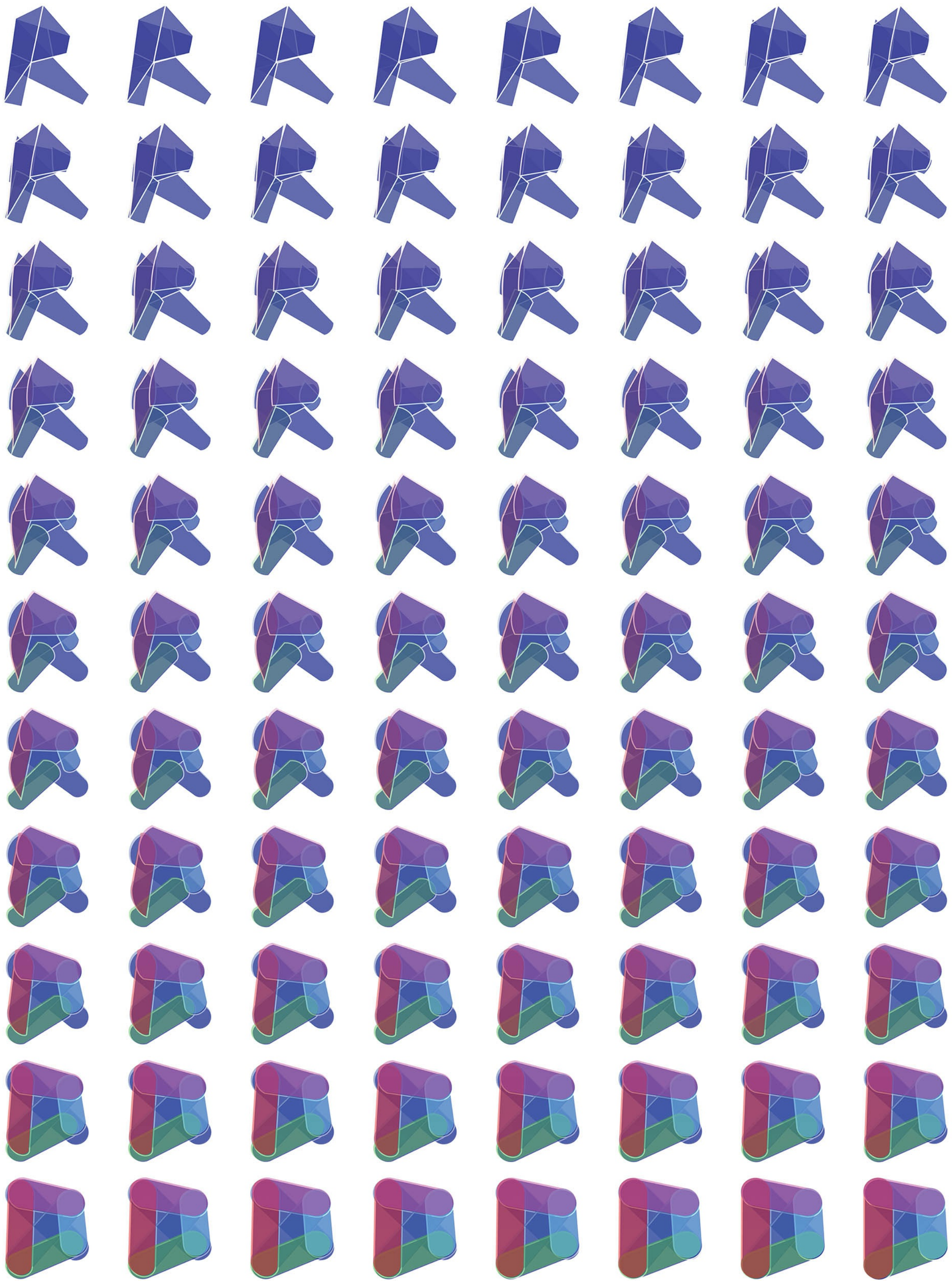
   ```
   def sphereByZ(inputPt,radiusRatio)
   {
   //get Z Value, use it to drive radius of sphere
   sphereRadius=inputPt.Z/radiusRatio;
   //Define Sphere Geometry
   sphere=Sphere.ByCenterPointRadius(inputPt,sphereRadius);
   //Define output for function
   return sphere;
   };
   ```

2. Update the children code blocks by adding a *ratio* variable to the input: `sphereByZ(Pt,ratio);` Plug a slider into the newly created code block input and vary the size of the radii based on the radius ratio.

Dynamo for Revit

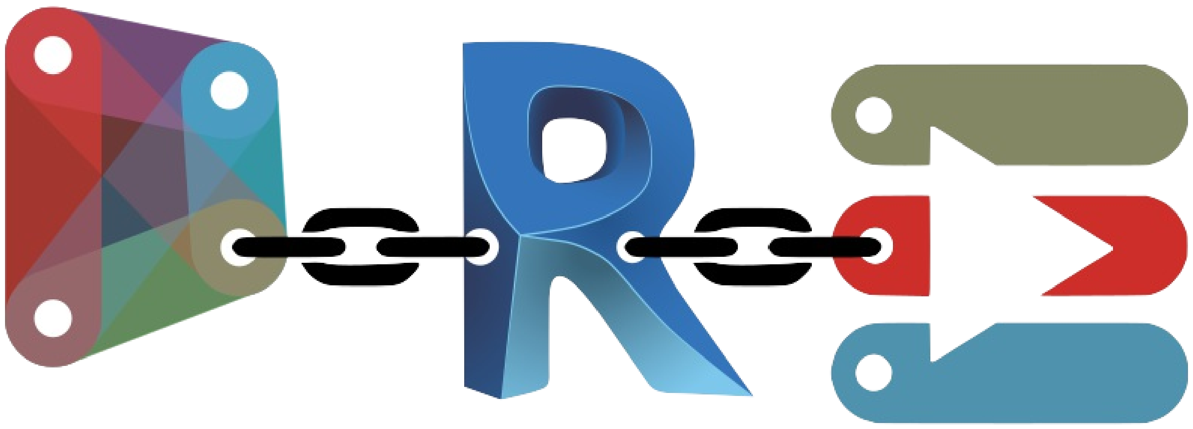
Dynamo for Revit

While Dynamo is a flexible environment, designed to port into a wide range of programs, it was originally created for use with Revit. A visual program creates robust options for a Building Information Model (BIM). Dynamo offers a whole suite of nodes specifically designed for Revit, as well as third-party libraries from a thriving AEC community. This chapter focuses on the basics of using Dynamo in Revit.



The Revit Connection

The Revit Connection



Dynamo for Revit extends buildin g information modeling with the data and logic environment of a graphical algorithm editor. Its flexibility, coupled with a robust Revit database, offers a new perspective for BIM.

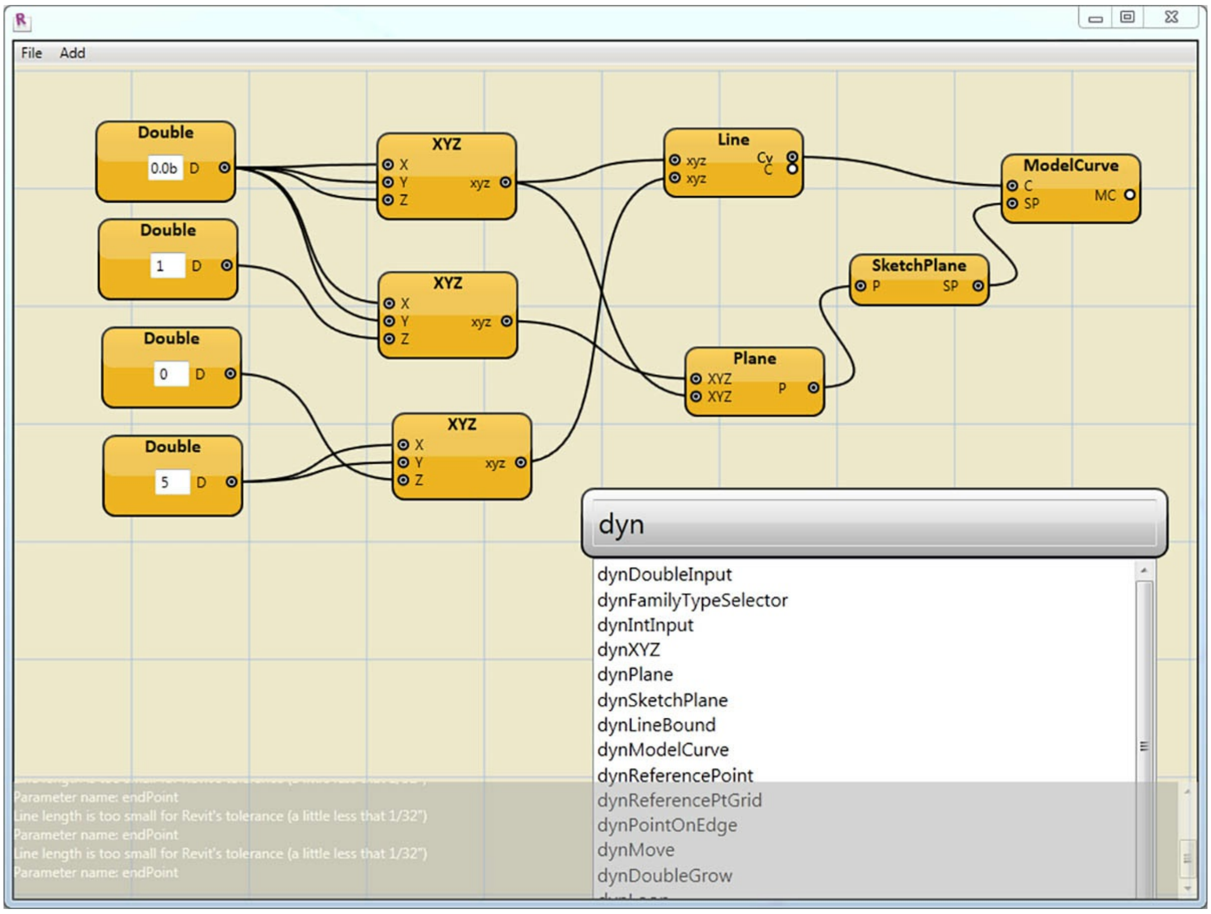
This chapter focuses on the Dynamo workflows for BIM. Sections are primarily exercise-based, since jumping right into a project is the best way to get familiar with a graphical algorithm editor for BIM. But first, let's talk about the beginnings of the program.

Revit Version Compatibility

As both Revit and Dynamo continue to evolve, you may notice that the Revit version you are working with is not compatible with the Dynamo for Revit version you have installed on your machine. Below outlines which versions of Dynamo for Revit are compatible with Revit.

Revit Version	First Stable Dynamo Version	Last Supported Dynamo for Revit Version
2013	0.6.1	0.6.3
2014	0.6.1	0.8.2
2015	0.7.1	1.2.1
2016	0.7.2	1.3.2
2017	0.9.0	1.3.4 / 2.0.3
2018	1.3.0	1.3.4 / 2.0.3
2019	1.3.3	1.3.4 / 2.0.4
2020+	2.1.0 - Revit 2020+ now includes Dynamo and receives updates as Revit does.)	N/A

History of Dynamo



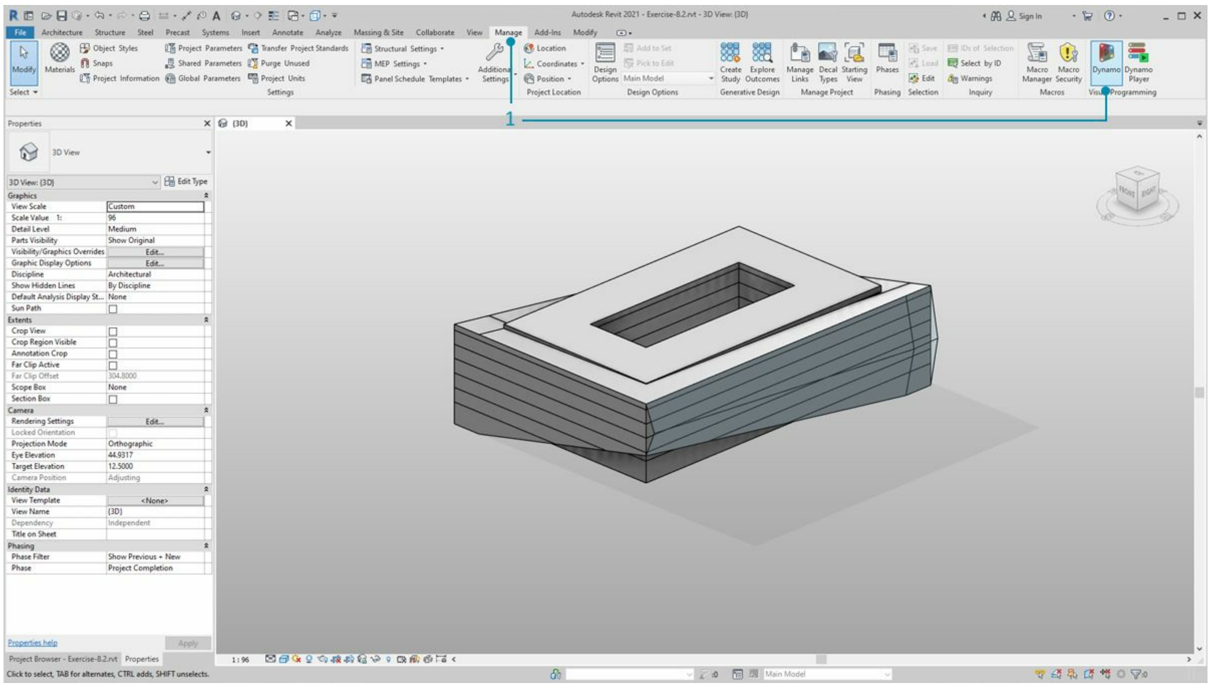
```
Parameter name: endPoint
Line length is too small for Revit's tolerance (a little less that 1/32")
Parameter name: endPoint
Line length is too small for Revit's tolerance (a little less that 1/32")
Parameter name: endPoint
```

With a dedicated team of developers and a passionate community, the project has come a long way from its humble beginnings.

Dynamo was originally created to streamline AEC workflows in Revit. While Revit creates a robust database for every project, it can be difficult for an average user to access this information outside of the constraints of the interface. Revit hosts a comprehensive API (Application Program Interface), allowing third-party developers to create custom tools. And programmers have been using this API for years, but text-based scripting isn't accessible to everyone. Dynamo seeks to democratize Revit data through an approachable graphical algorithm editor.

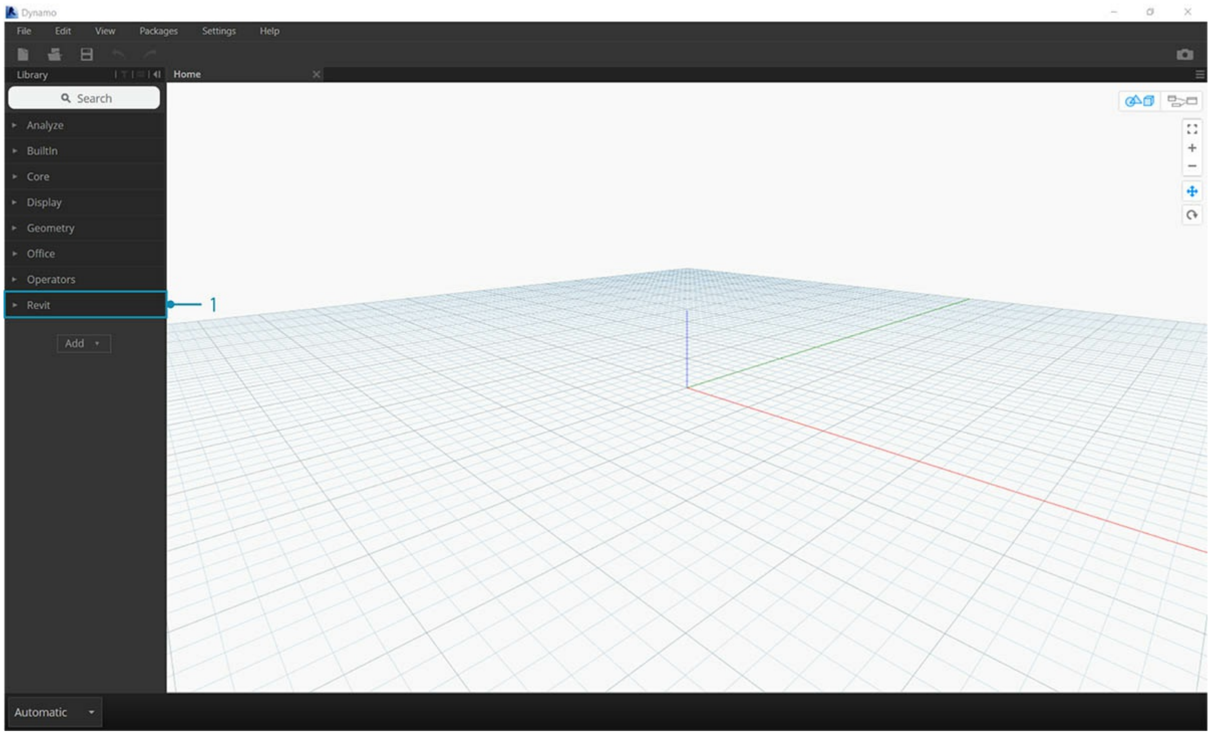
Using the core Dynamo nodes in tandem with custom Revit ones, a user can substantially expand parametric workflows for interoperability, documentation, analysis, and generation. With Dynamo, tedious workflows can be automated while design explorations can thrive.

Running Dynamo in Revit



View Scale	Custom
Scale Value 1:	96
Detail Level	Medium
Parts Visibility	Show Original
Discipline	Architectural
Show Hidden Lines	By Discipline
Default Analysis Display St...	None
Far Clip Offset	304.8000
Scope Box	None
Projection Mode	Orthographic
Eye Elevation	44.9317
Target Elevation	12.5000
Camera Position	Adjusting
View Template	<None>
View Name	{3D}
Dependency	Independent
Phase Filter	Show Previous + New
Phase	Project Completion

1. In a Revit project or family editor, navigate to Addins and click *Dynamo*. Take note: Dynamo will run only in the file in which it was opened.



1. When opening Dynamo in Revit, there is a new category called *"Revit"*. This is a comprehensive addition to the UI which offers nodes specifically catering to Revit workflows.*

**Note - By using the Revit-specific family of nodes, the Dynamo graph will only work when opening in Dynamo for Revit. If a Dynamo for Revit graph is opened in Dynamo Sandbox for example, the Revit nodes will be missing.*

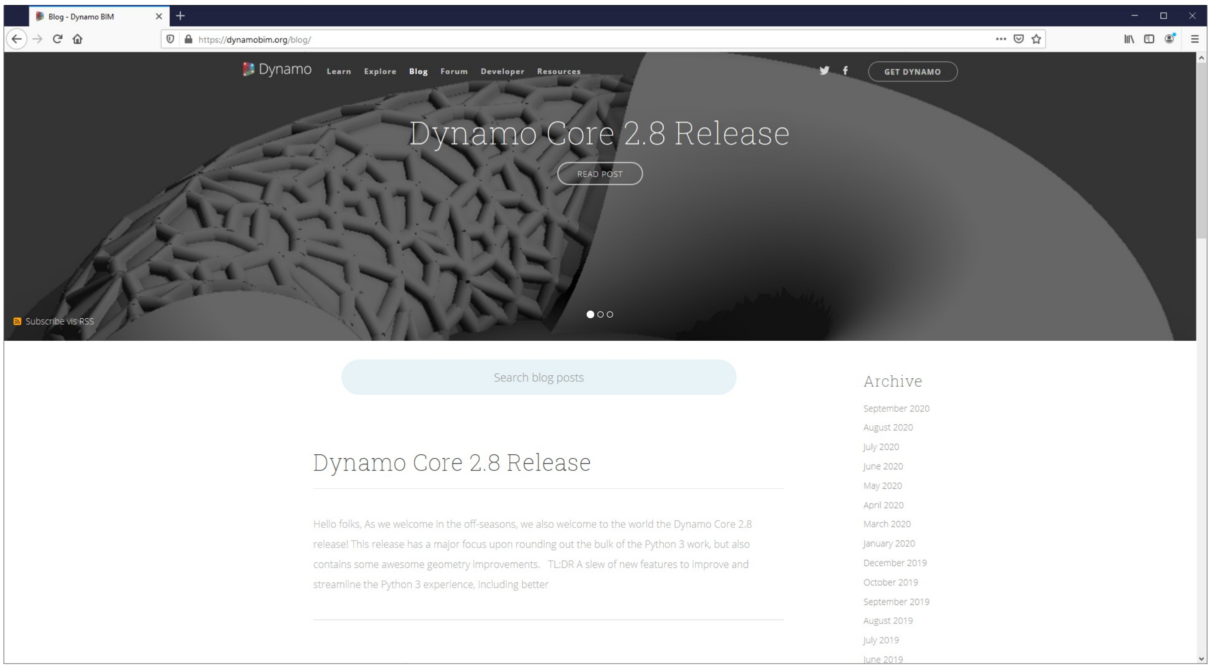
Freezing Nodes

Since Revit is a platform which provides robust project management, parametric operations in Dynamo can be complex and slow to calculate. If Dynamo is taking a long time to calculate nodes, you may want to use the "freeze" node functionality in order to pause the execution of Revit operations while you develop your graph. For more information on freezing nodes, check out the "Freezing" section in the solids chapter.

Community

Since Dynamo was originally created for AEC, its large and growing community is a great resource for learning from and connecting with experts in the industry. Dynamo’s community is made of architects, engineers, programmers, and designers who all have a passion for sharing and making.

Dynamo is an open-source project that is constantly evolving, and a lot of development is Revit-related. If you're new to the game, get on the discussion forum and start posting questions! If you're a programmer and want to get involved in Dynamo's development, check out the github page. Also, a great resource for third-party libraries is the Dynamo package manager. Many of these packages are made with AEC in mind, and we'll take a look at third-party packages for panelization in this chapter.



Dynamo Core 2.8 Release

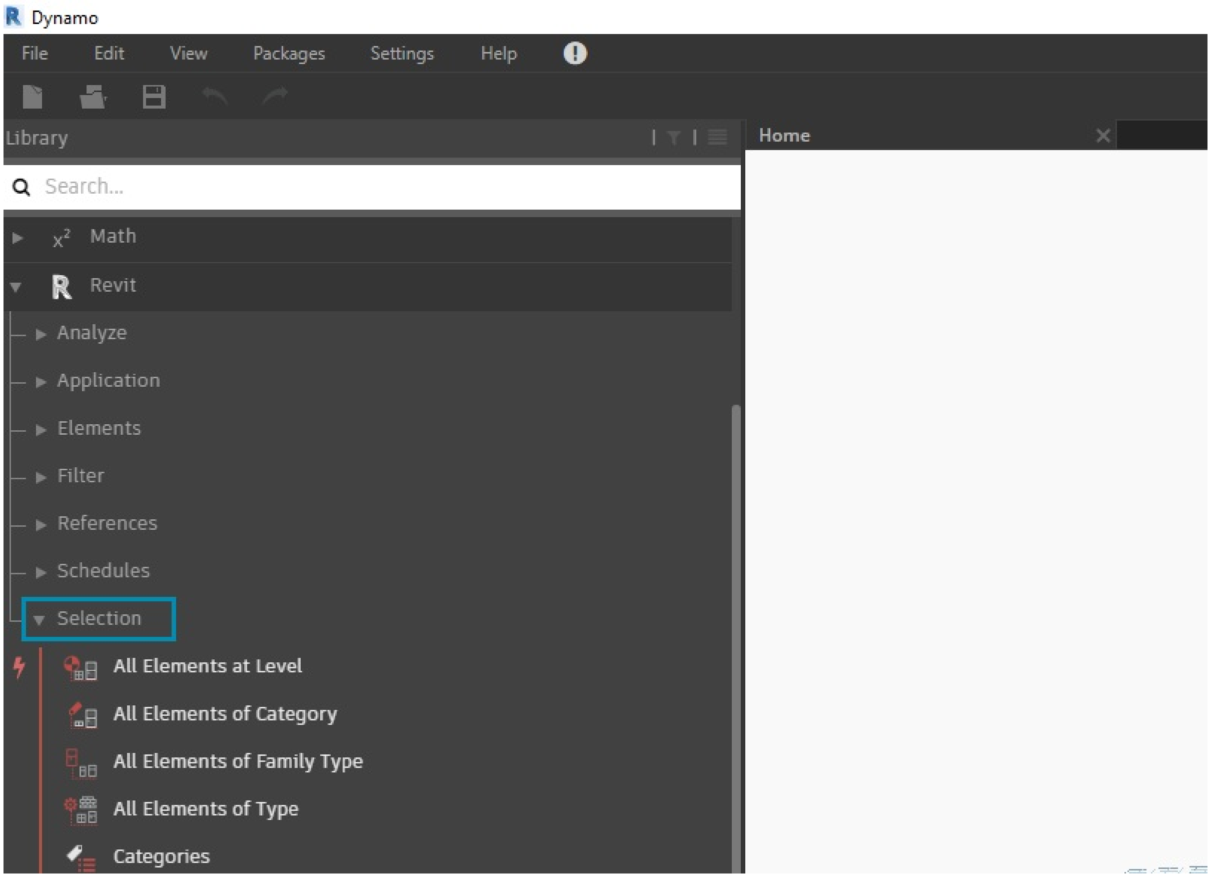
Hello folks, As we welcome in the off-seasons, we also welcome to the world the Dynamo Core 2.8 release! This release has a major focus upon rounding out the bulk of the Python 3 work, but also contains some awesome geometry improvements.   TL:DR A slew of new features to improve and streamline the Python 3 experience, including better

Dynamo also maintains an active blog. Read up on recent posts to learn about the latest developments!

Selecting

Selecting

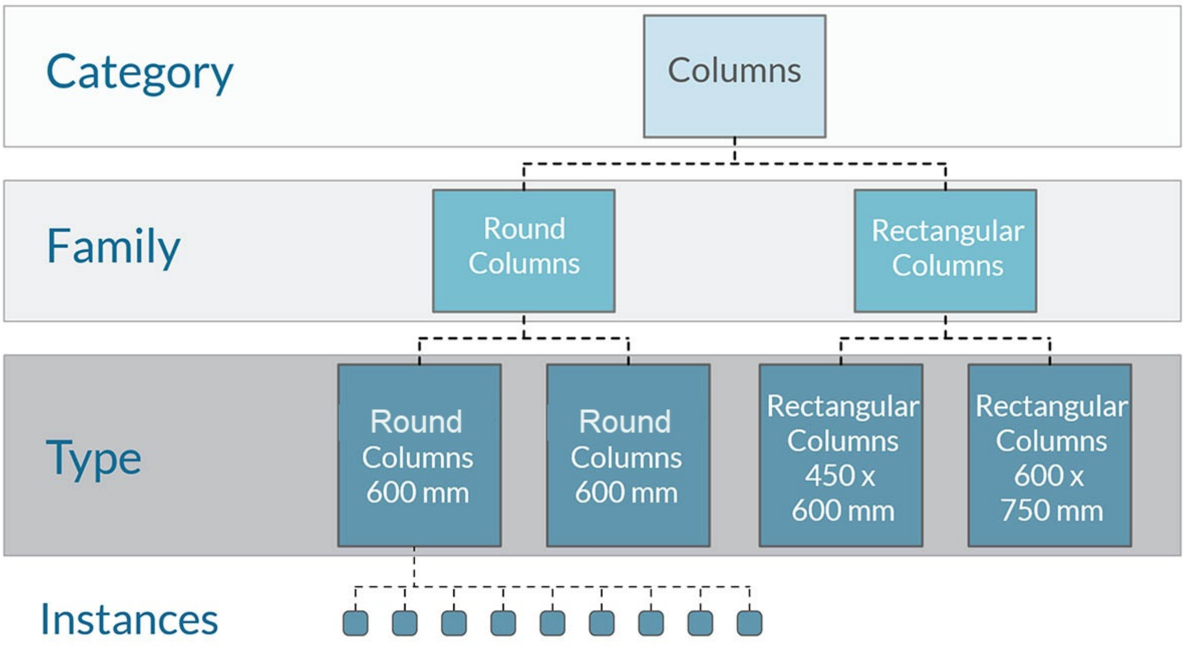
Revit is a data-rich environment. This gives us a range of selection abilities which expands far beyond "point-and-click". We can query the Revit database and dynamically link Revit elements to Dynamo geometry while performing parametric operations.



The Revit library in the UI offers a "Selection" category which enables multiple ways to select geometry.

To select Revit elements properly, it's important to have a full-understanding of the Revit element hierarchy. Want to select all the walls in a project? Select by category. Want to select every Eames chair in your mid-century modern lobby? Select by family. Before jumping into an exercise, let's do a quick review of the Revit hierarchy.

Revit Hierarchy

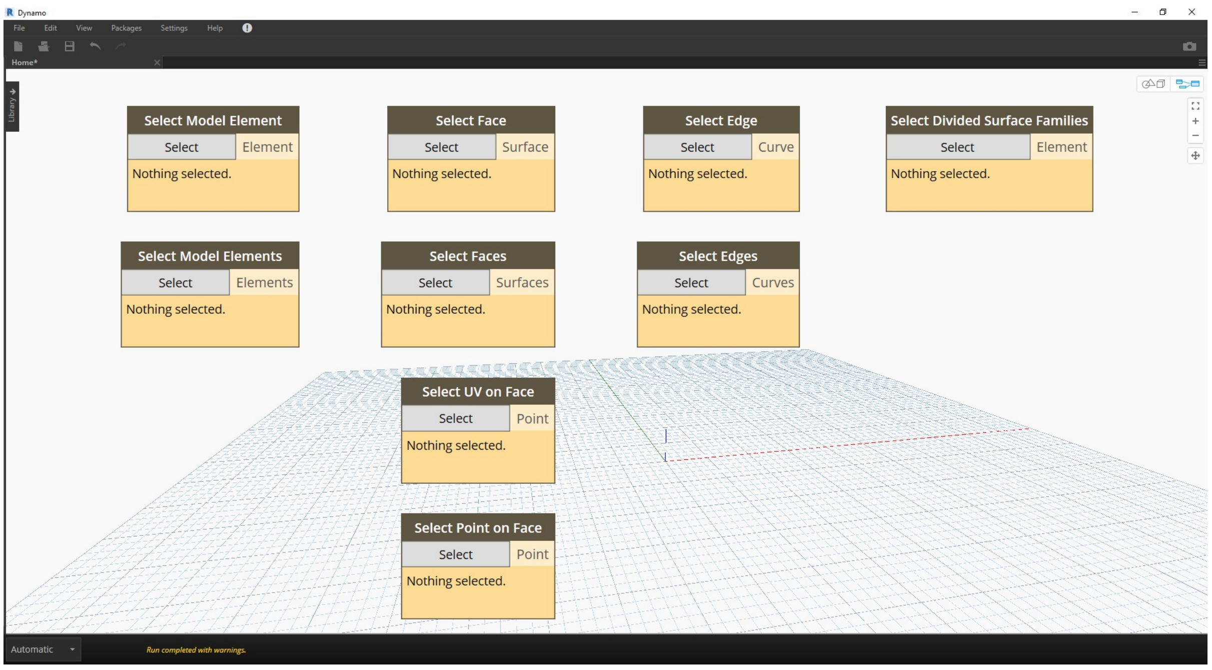


Remember the taxonomy from Biology? Kingdom, Phylum, Class, Order, Family, Genus, Species? Revit elements are categorized in a similar manner. On a basic level, the Revit hierarchy can be broken down into Categories, Families, Types*, and Instances. An instance is an individual model element (with a unique ID) while a category defines a generic group (like "walls" or "floors"). With the Revit database organized in this manner, we can select one element and choose all similar elements based on a specified level in the hierarchy.

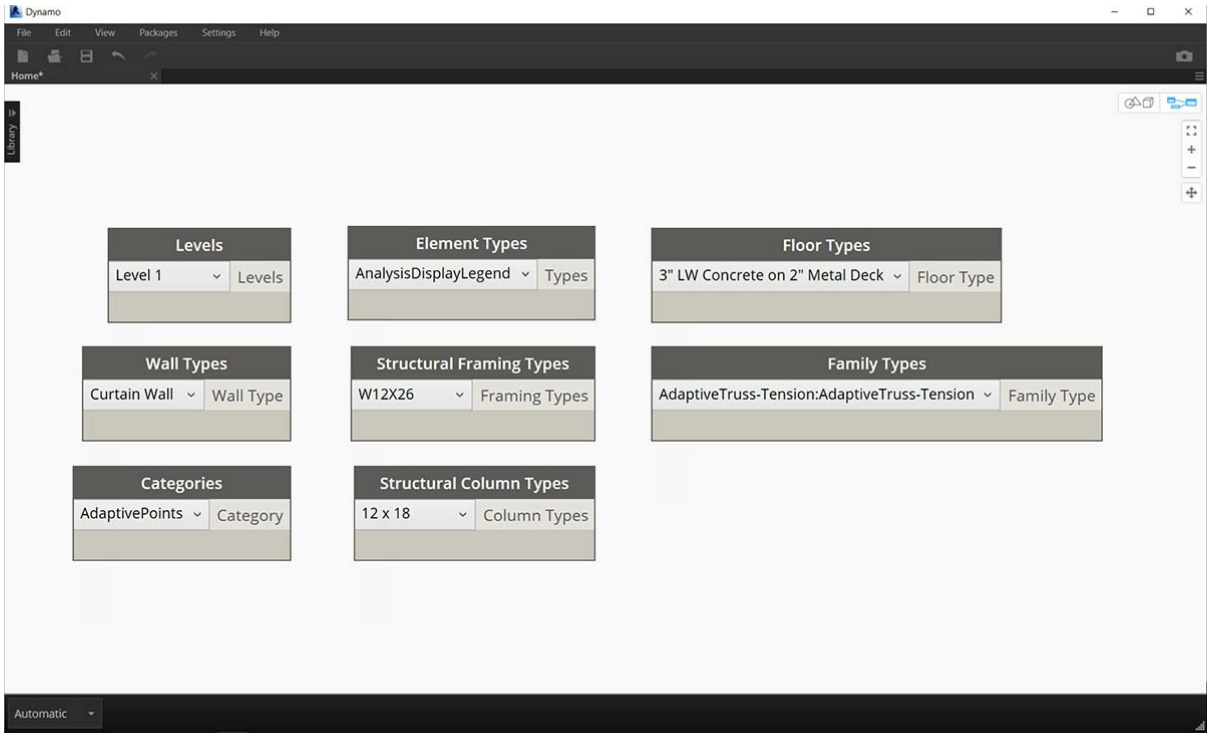
**Note - Types in Revit are defined differently from types in programming. In Revit, a type refers to a branch of the hierarchy, rather than a "data type".*

Database Navigation with Dynamo nodes

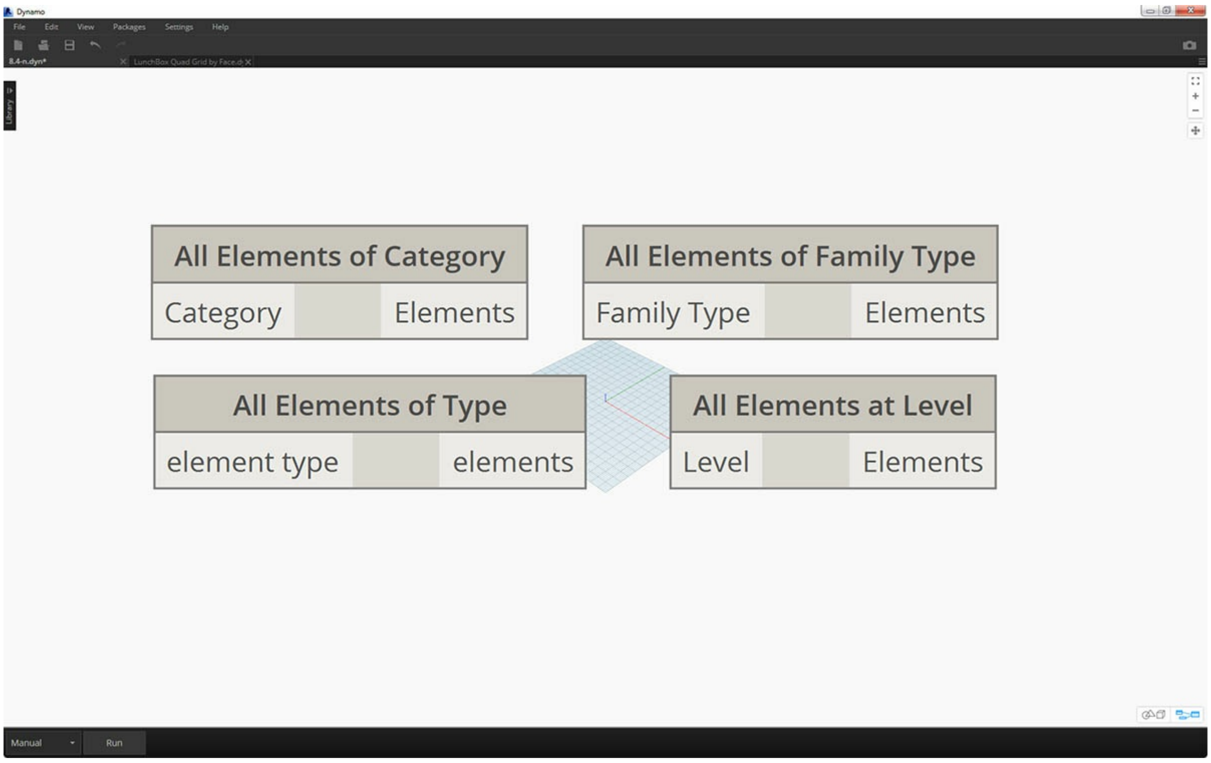
The three images below breakdown the main categories for Revit element selection in Dynamo. These are great tools to use in combination, and we'll explore some of these in the following exercises.



Point-and-click is the easiest way to directly select a Revit element. You can select a full model element, or parts of its topology (like a face or an edge). This remains dynamically linked to that Revit object, so when the Revit file updates its location or parameters, the referenced Dynamo element will update in the graph.



Dropdown menus create a list of all accessible elements in a Revit project. You can use this to reference Revit elements which are not necessarily visible in a view. This is a great tool for querying existing elements or creating new ones in a Revit project or family editor.



You can also select Revit element by specific tiers in the *Revit hierarchy*. This is a powerful option for customizing large arrays of data in preparation for documentation or generative instantiation and customization.

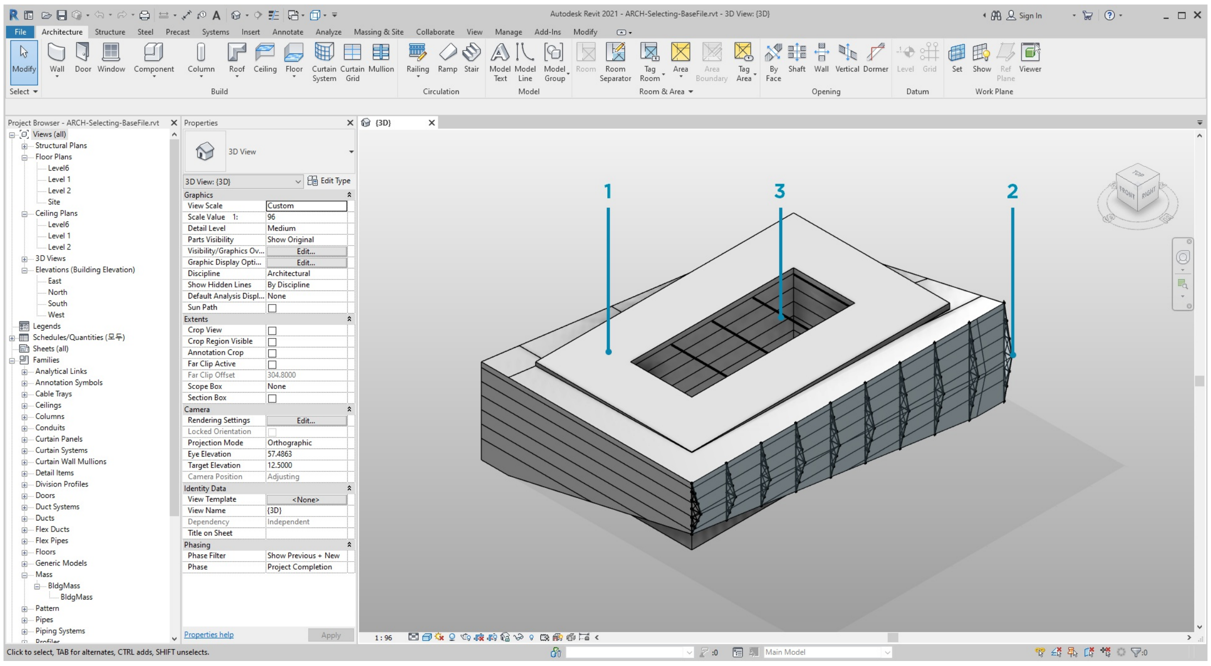
With the three images above in mind, let's dive into an exercise which selects elements from a basic Revit project in preparation for the parametric applications we'll create in the remaining sections of this chapter.

Exercise

Download the example files that accompanies this exercise (Right click and "Save Link As..."). A full list of example files can be found in the Appendix.

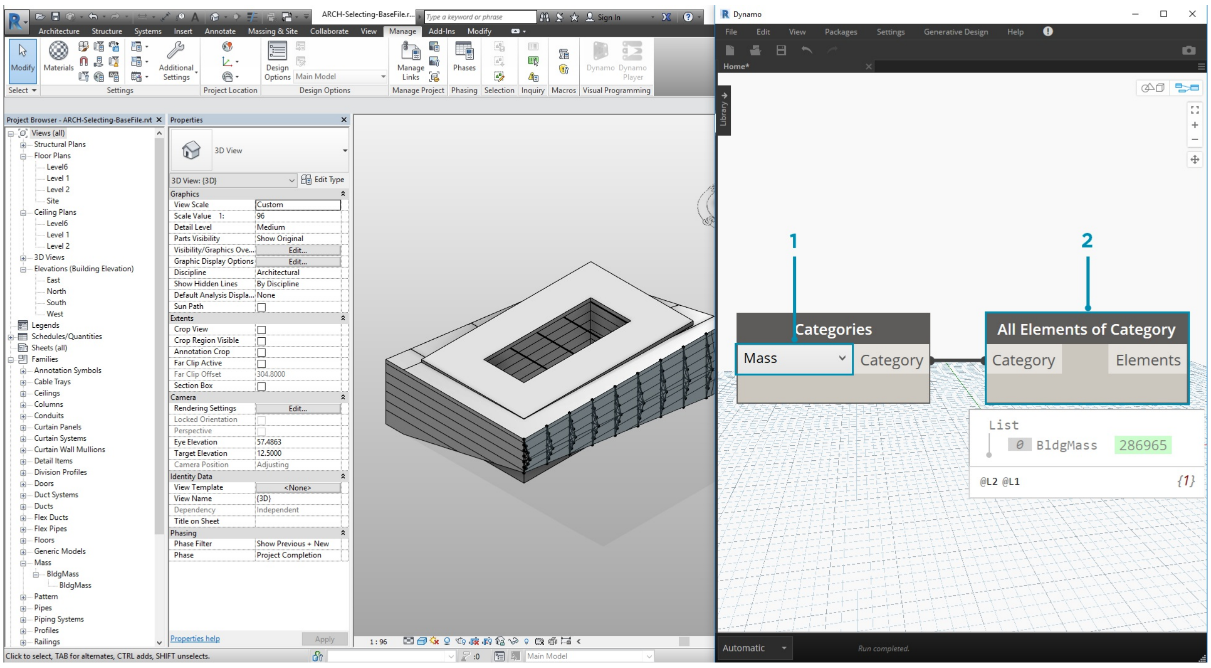
1. Selecting.dyn

2. [ARCH-Selecing-BaseFile.rvt]()



In this example Revit file, we have three element types of a simple building. We're going to use this as an example for selecting Revit elements within the context of the Revit hierarchy:

1. Building Mass
2. Trusses (Adaptive Components)
3. Beams (Structural Framing)

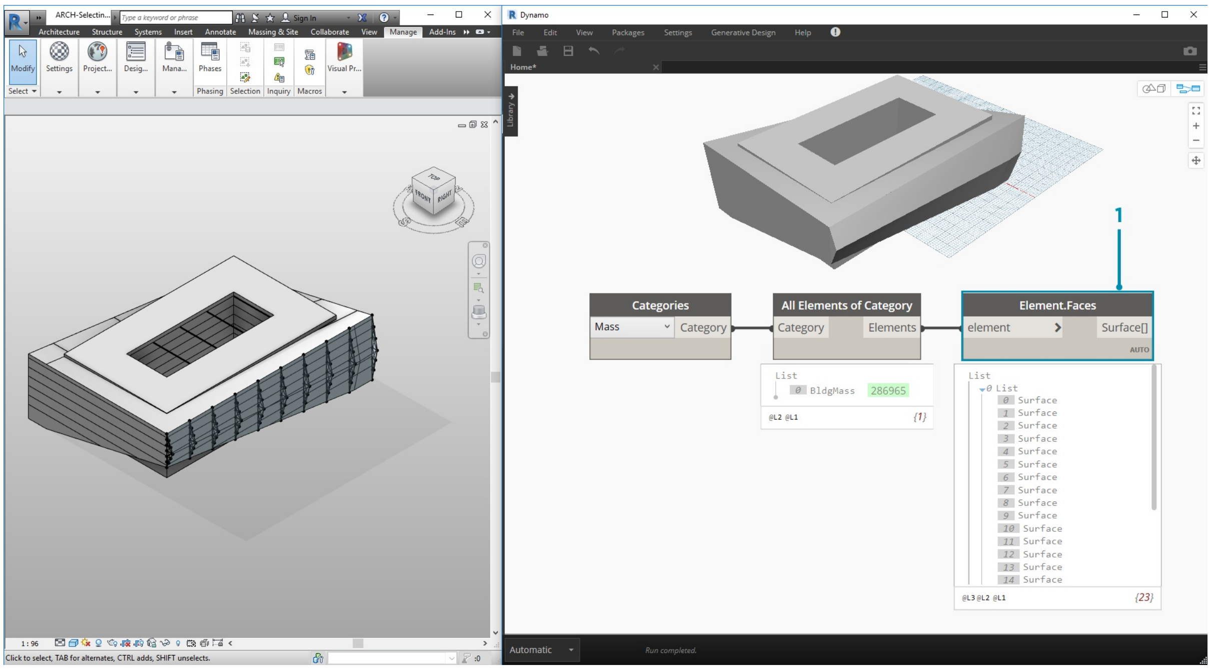


What conclusions can we draw from the elements currently in the Revit project view? And how far down the hierarchy do we need to go to select the appropriate elements? This will of course become a more complex task when working on a large project. There are a lot of options available: we can select elements by categories, levels, families, instances, etc.

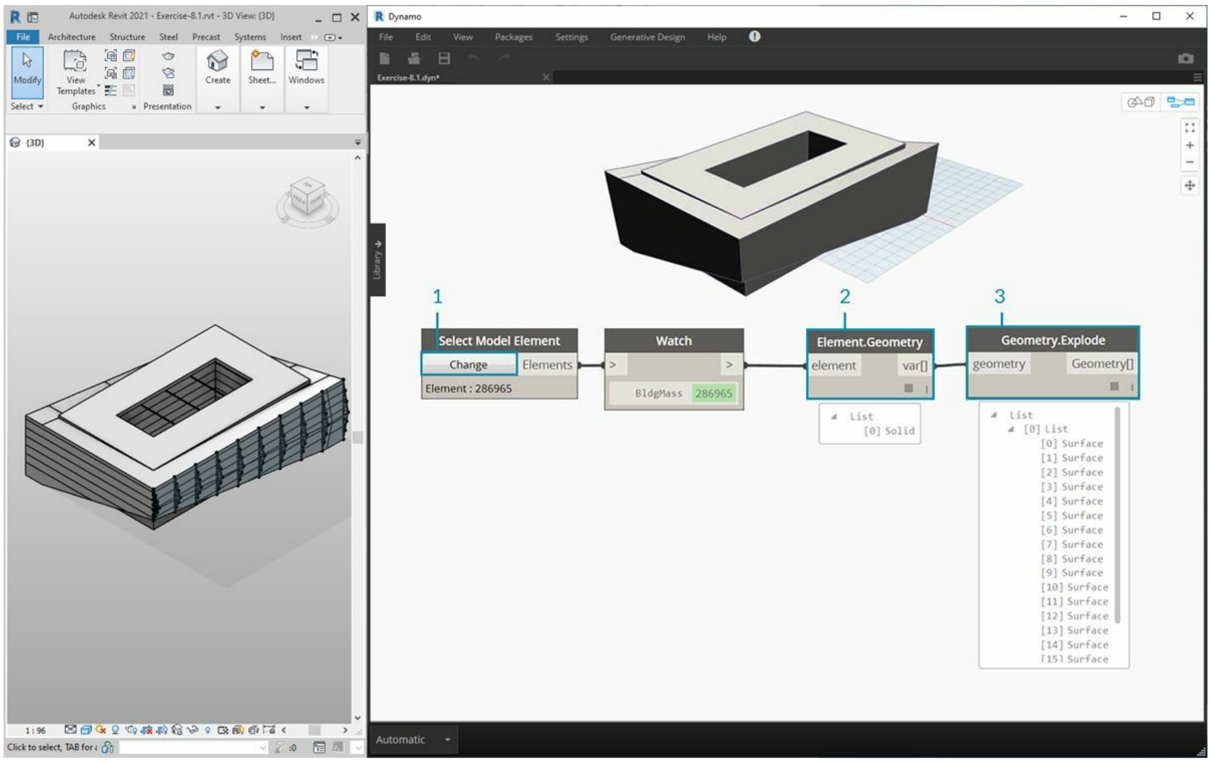
1. Since we're working with a basic setup, let's select the building mass by choosing *"Mass"* in the Categories dropdown node. This can be found in the Revit>Selection tab.
2. The output of the Mass category is just the category itself. We need to select the elements. To do this, we use the *"All Elements of Category"* node.

At this point, notice that we don't see any geometry in Dynamo. We've selected a Revit element, but have not converted the element into Dynamo geometry. This is an important separation. If you were to select a large number of elements, you don't want to preview all of them in Dynamo because this would slow everything down. Dynamo is a tool to manage a Revit project without necessarily performing geometry operations, and we'll look at that in the next section of this chapter.

In this case, we're working with simple geometry, so we want to bring the geometry into the Dynamo preview. The "BldgMass" in the watch node above has a green number* next to it. This represents the element's ID and tells us that we are dealing with a Revit element, not Dynamo geometry. The next step is to convert this Revit element into geometry in Dynamo.

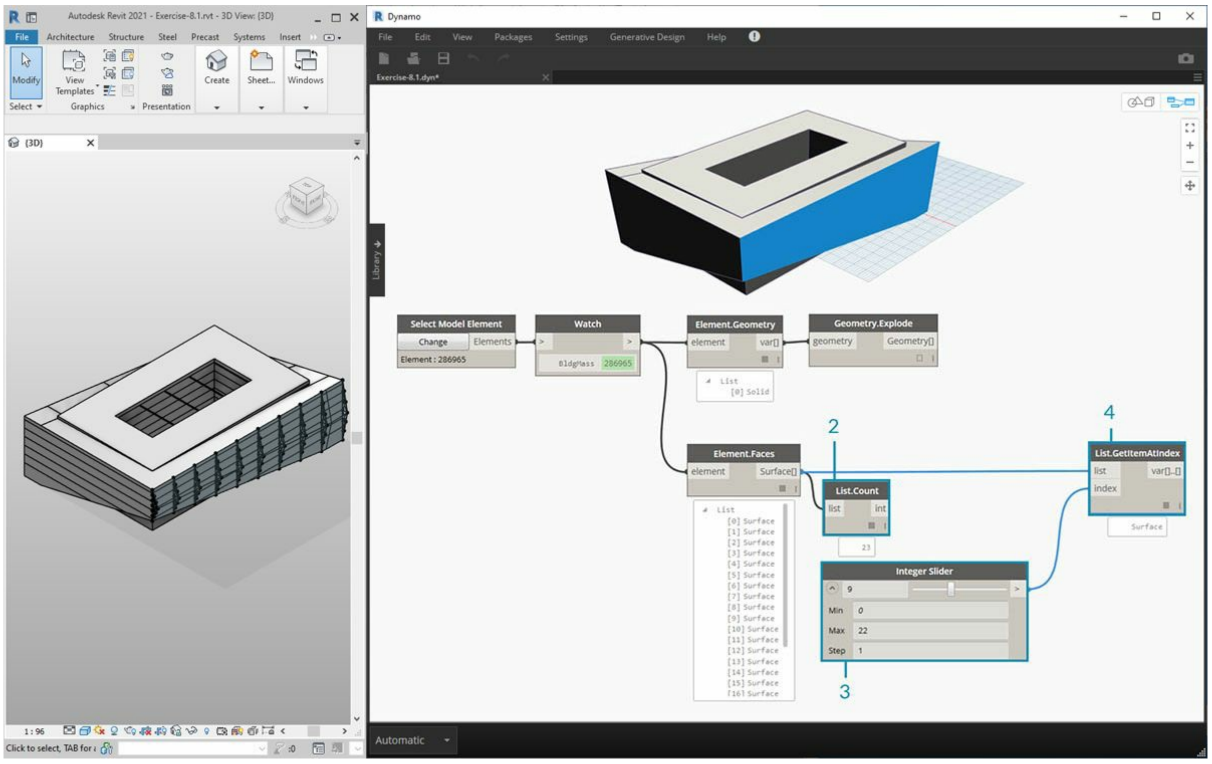


1. Using the *Element. Faces* node, we get a list of surfaces representing each face of the Revit Mass. We can now see the geometry in the Dynamo viewport and start to reference the face for parametric operations.

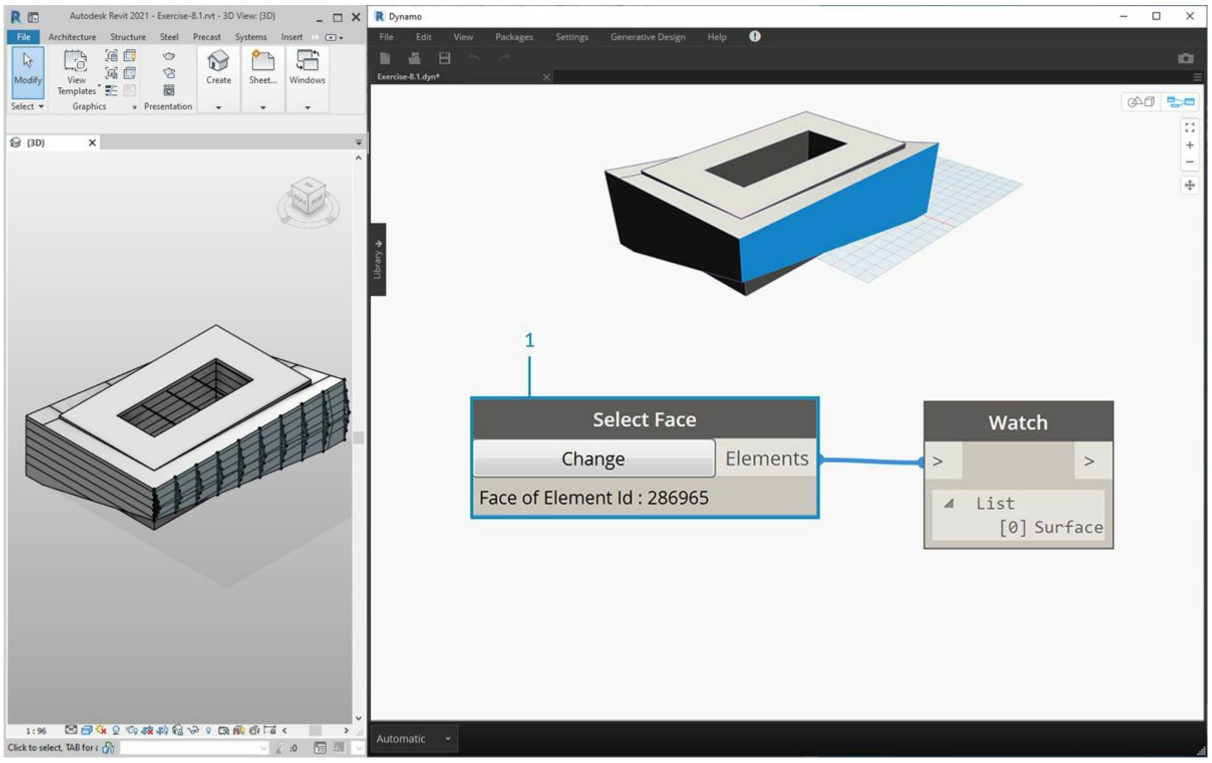


Here's an alternative method. In this case, we're stepping away from selecting via the Revit Hierarchy *("All Elements of Category")* and electing to explicitly select geometry in Revit.

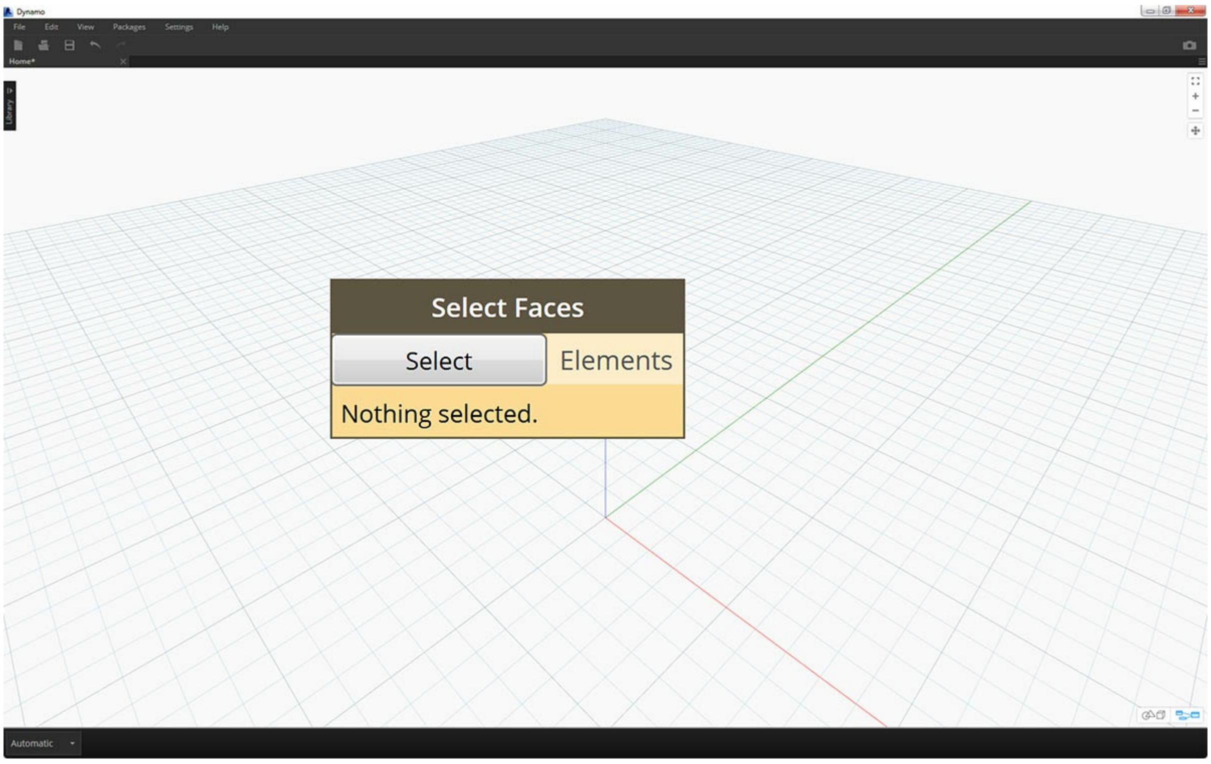
1. Using the *"Select Model Element"* node, click the *"select"* (or *"change"*) button. In the Revit viewport, select the desired element. In this case, we're selecting the building mass.
2. Rather than *Element.Faces*, we can select the full mass as one solid geometry using *Element.Geometry*. This selects all of the geometry contained within that mass.
3. Using *Geometry.Explode*, we can get the list of surfaces again. These two nodes work the same as *Element.Faces* but offer alternative options for delving into the geometry of a Revit element.



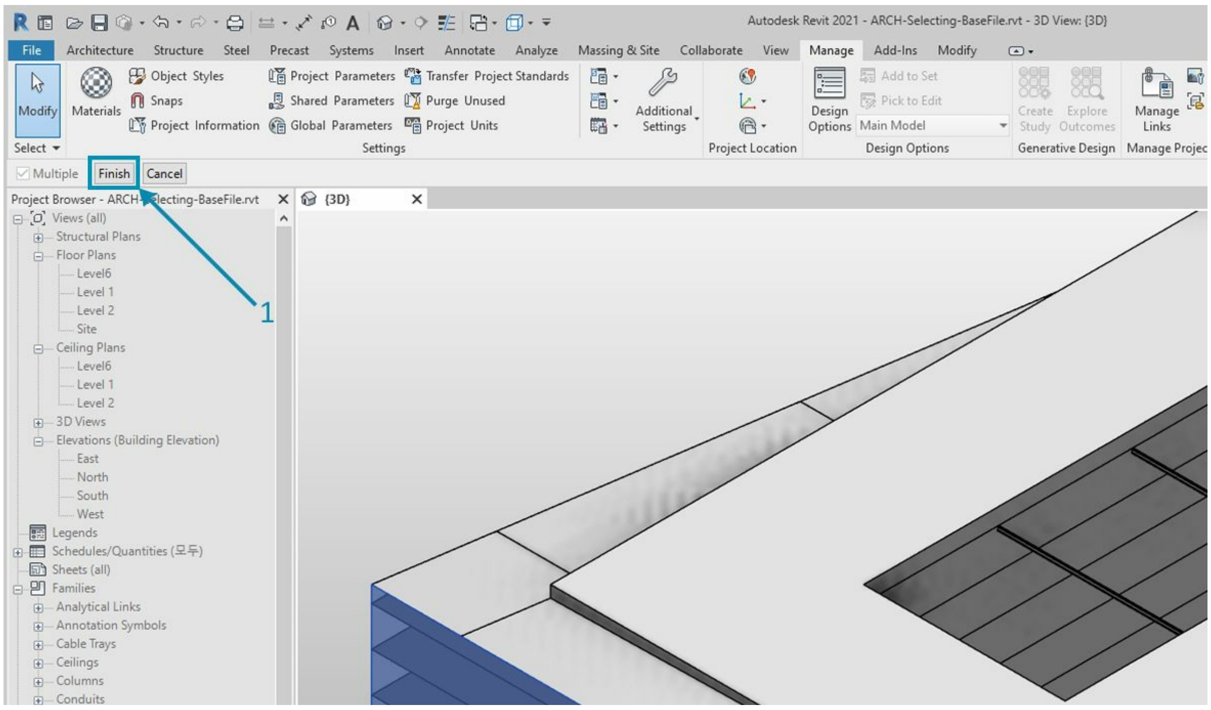
1. Using some basic list operations, we can query a face of interest.
2. First, the *List.Count* node reveals that we're working with 23 surfaces in the mass.
3. Referencing this number, we change the Maximum value of an *integer slider* to *"22"*.
4. Using *List.GetItemAtIndex*, we input the lists and the *integer slider* for the *index*. Sliding through with the selected, we stop when we get to *index 9* and have isolated the main facade hosts the trusses.



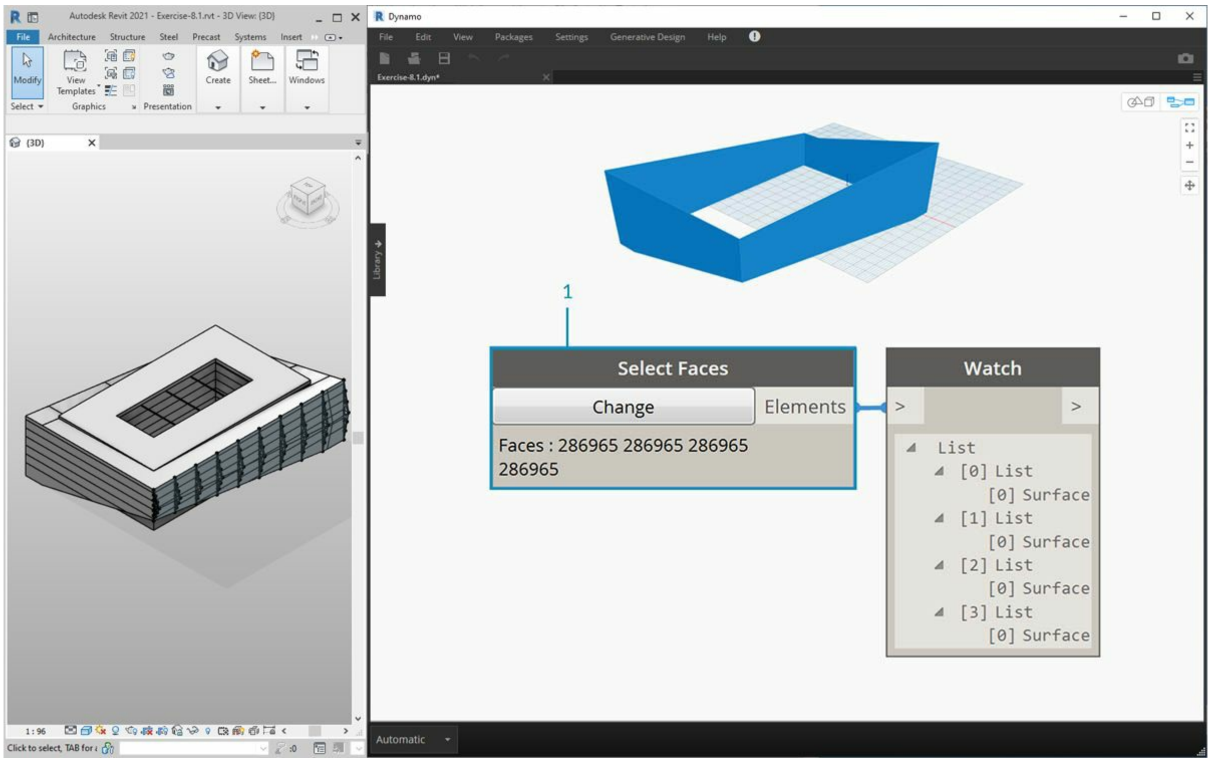
1. The previous step was a little cumbersome. We can do this much faster with the *"Select Face"* node. This allows us to isolate a face that is not an element itself in the Revit project. The same interaction applies as *"Select Model Element"*, except we select the surface rather than the full element.



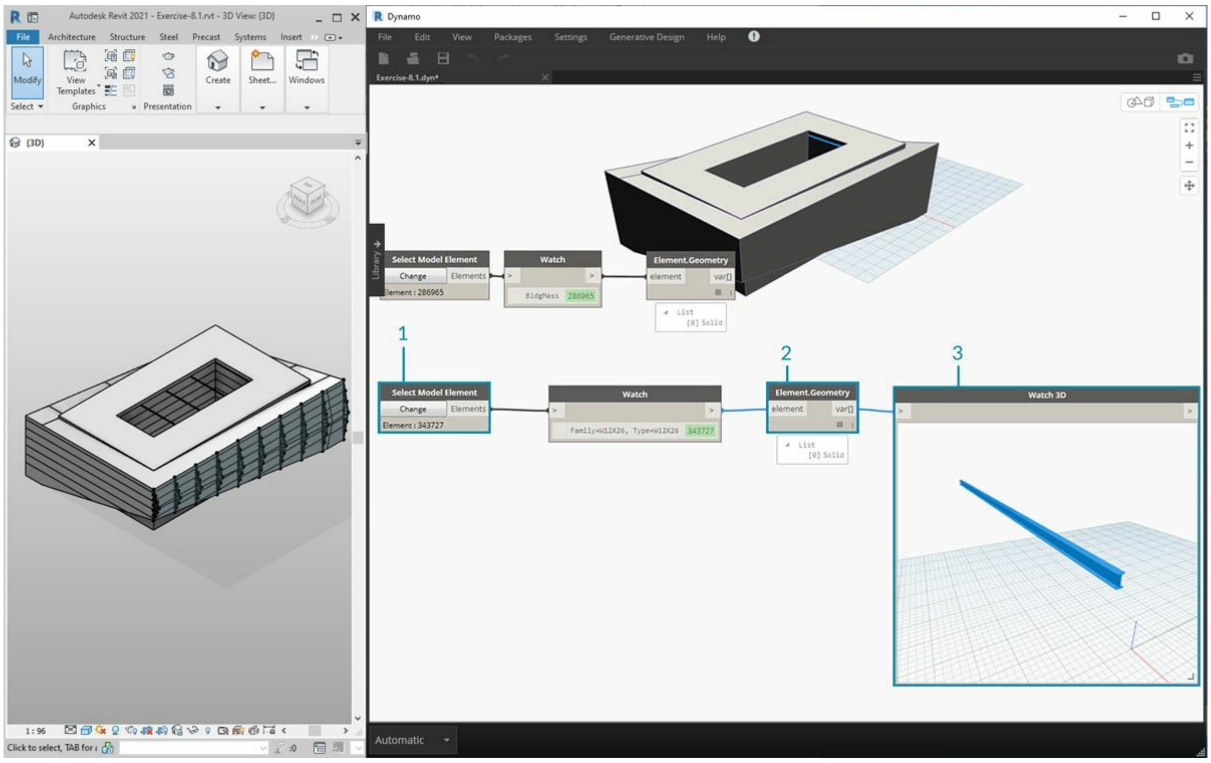
Suppose we want to isolate the main facade walls of the building. We can use the *"Select Faces"* node to do this. Click the "Select" button and then select the four main facades in Revit.


Autodesk Revit 2021 - ARCH-Selecting-BaseFile.rvt - 3D View: {3D}
Multiple
Finish
Cancel
1
Project Browser - ARCH-Selecting-BaseFile.rvt
{3D}

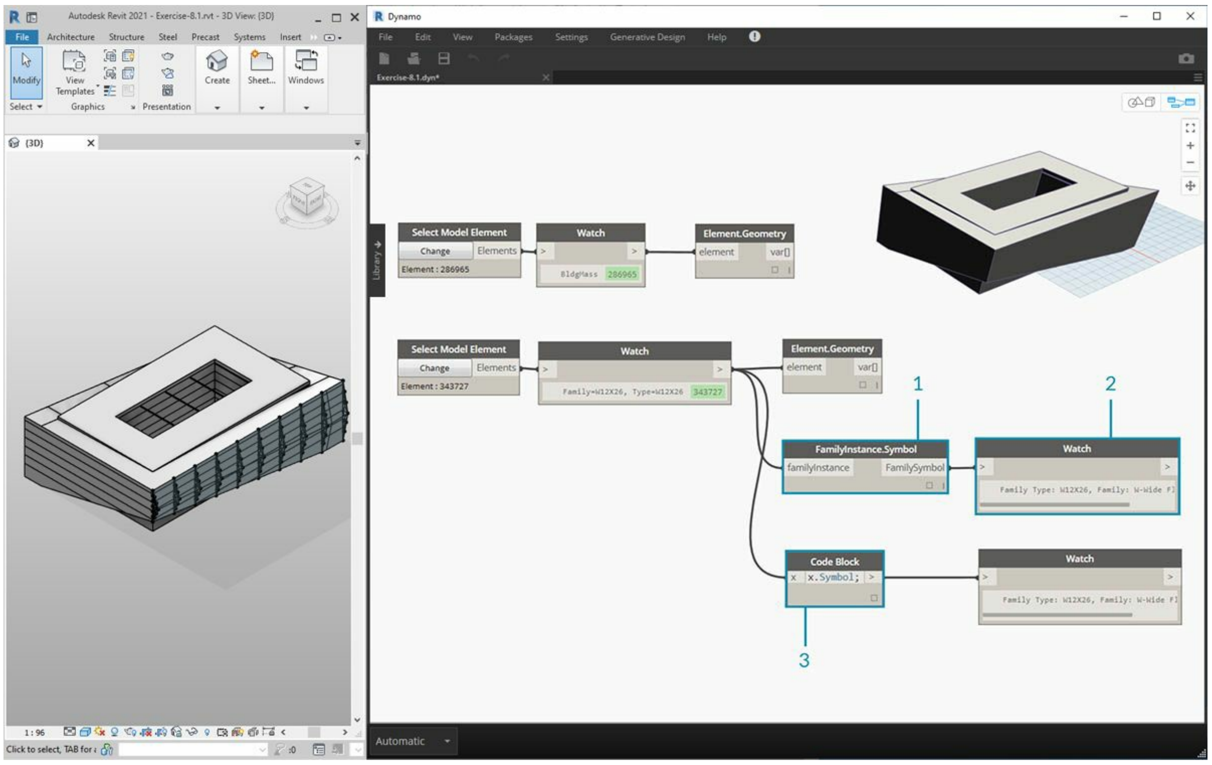
1. After selecting the four walls, make sure you click the *"Finish"* button in Revit.



1. The faces are now imported into Dynamo as surfaces.



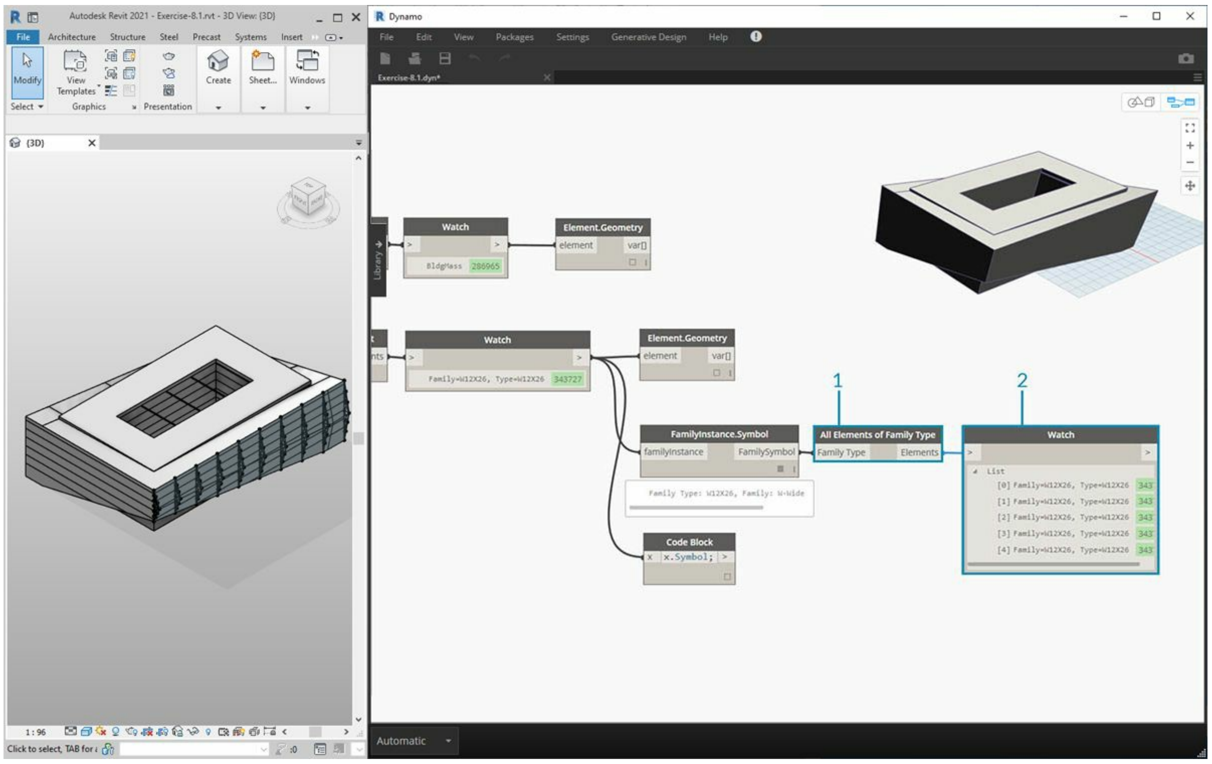
1. Now, let's take a look at the beams over the atrium. Using the *"Select Model Element"* node, select one of the beams.
2. Plug the beam element into the *Element.Geometry* node and we now have the beam in the Dynamo viewport.
3. We can zoom in on the geometry with a *Watch3D* node (if you don't see the beam in Watch 3D, right click and hit "zoom to fit").



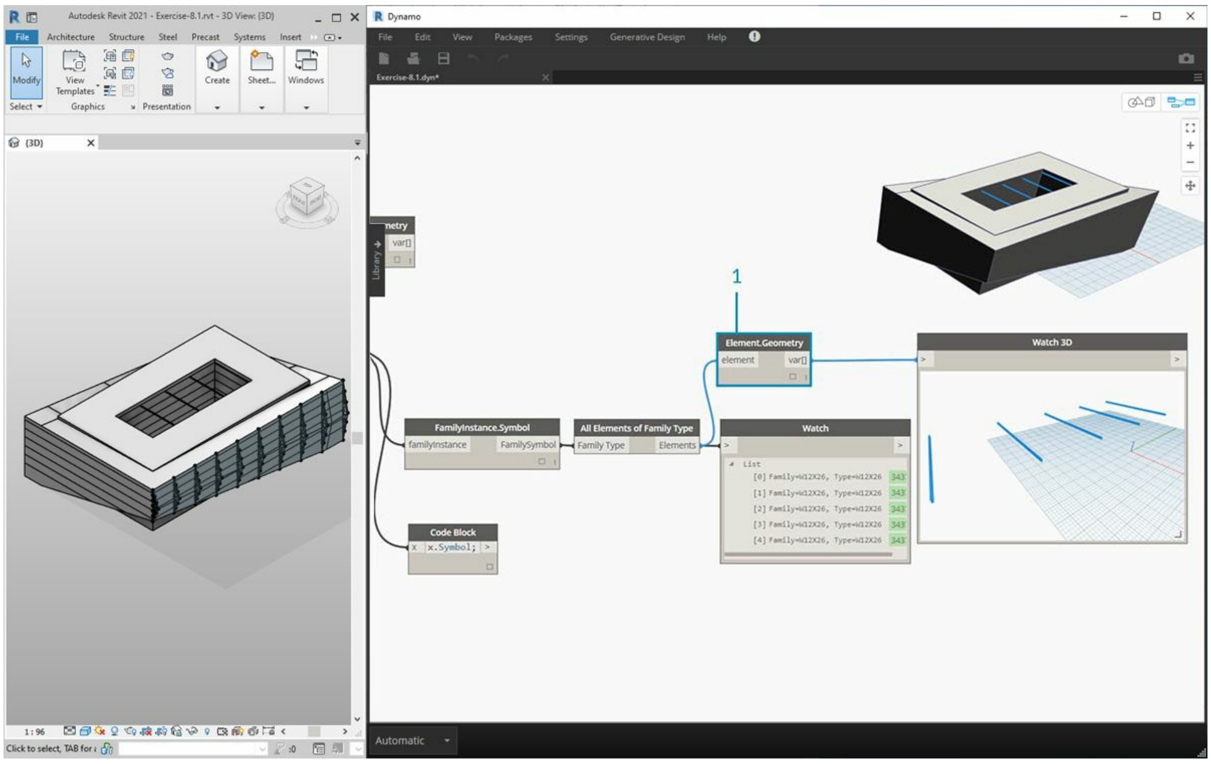
A question that may come up often in Revit/Dynamo workflows: how do I select one element and get all similar elements? Since the selected Revit element contains all of its hierarchical information, we can query its family type and select all elements of that type.

1. Plug the beam element into a *FamilyInstance.Symbol** node.
2. The *Watch* node reveals that the output is now a family symbol rather than a Revit element.
3. *FamilyInstance.Symbol* is a simple query, so we can do this in the code block just as easily with `x.Symbol;` and get the same results.

**Note - a family symbol is Revit API terminology for family type. Since this may cause some confusion, it will be updated in upcoming releases.*


Watch
BldgMass 286965
Element.Geometry
element
var[]
Family=W12X26, Type=W12X26 343727
FamilyInstance.Symbol
familyInstance
FamilySymbol
Family Type: W12X26, Family: W-Wide
All Elements of Family Type
Family Type
Elements
List
[0] Family=W12X26, Type=W12X26 343
[1] Family=W12X26, Type=W12X26 343
[2] Family=W12X26, Type=W12X26 343
[3] Family=W12X26, Type=W12X26 343
[4] Family=W12X26, Type=W12X26 343
Code Block
x.Symbol;
1
2
Automatic

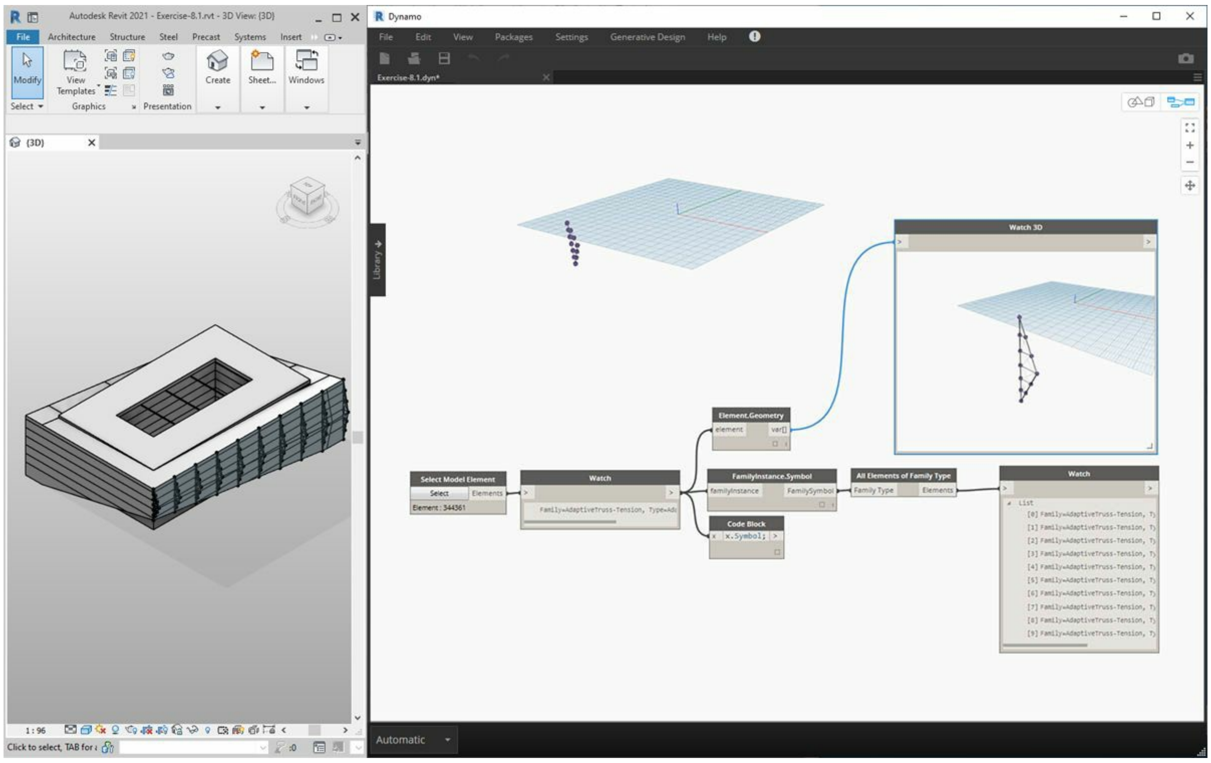
1. To select the remaining beams, we use the *"All Elements of Family Type"* node.
2. The watch node shows that we've selected five revit elements.


```
List
  [0] Family=W12X26, Type=W12X26 343
  [1] Family=W12X26, Type=W12X26 343
  [2] Family=W12X26, Type=W12X26 343
  [3] Family=W12X26, Type=W12X26 343
  [4] Family=W12X26, Type=W12X26 343
```

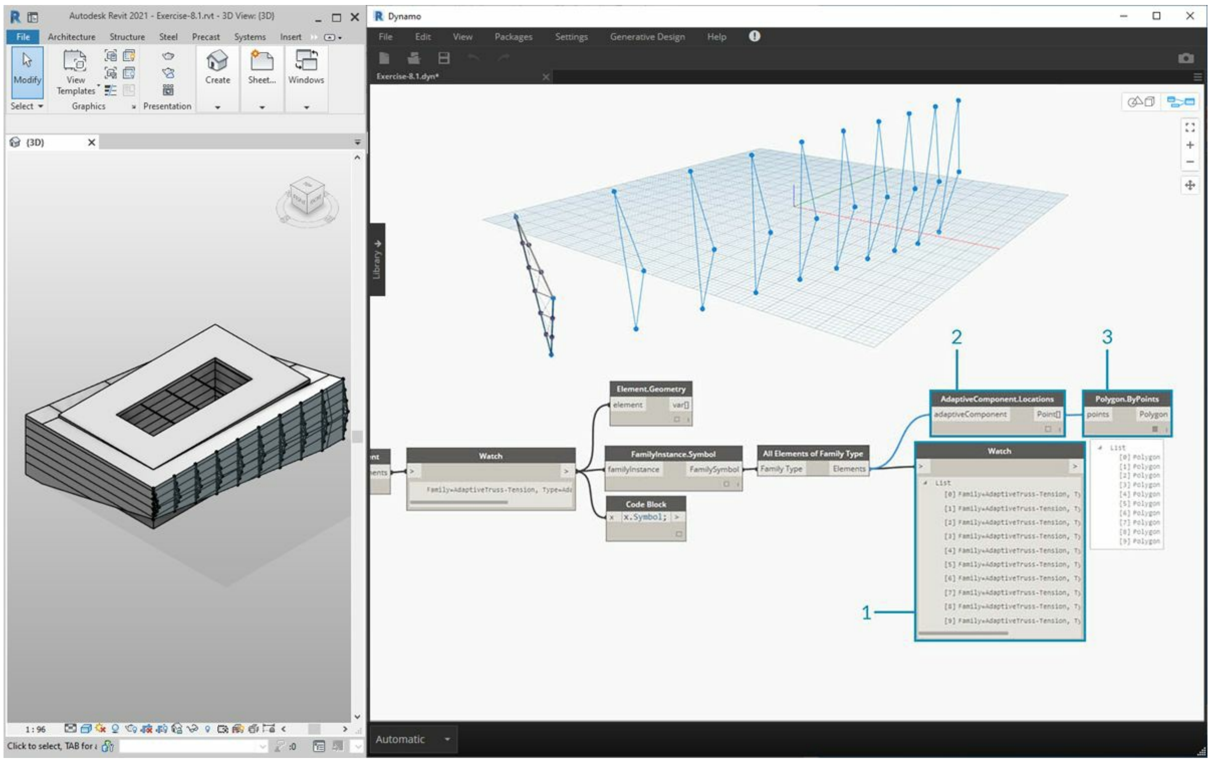
1. We can convert all of these five elements to Dynamo geometry too.

What if we had 500 beams? Converting all of these elements into Dynamo geometry would be really slow. If Dynamo is taking a long time to calculate nodes, you may want to use the "freeze" node functionality in order to pause the execution of Revit operations while you develop your graph. For more information on freezing nodes, check out the "Freezing" section in the solids chapter.

In any case, if we were to import 500 beams, do we need all of the surfaces to perform the intended parametric operation? Or can we extract basic information from the beams and perform generative tasks with fundamental geometry? This is a question that we'll keep in mind as we walk through this chapter. For example, let's take a look at the truss system:


```
Element : 344361

Family=AdaptiveTruss-Tension, Type=Ad

List
  [0] Family=AdaptiveTruss-Tension, Ty
  [1] Family=AdaptiveTruss-Tension, Ty
  [2] Family=AdaptiveTruss-Tension, Ty
  [3] Family=AdaptiveTruss-Tension, Ty
  [4] Family=AdaptiveTruss-Tension, Ty
  [5] Family=AdaptiveTruss-Tension, Ty
  [6] Family=AdaptiveTruss-Tension, Ty
  [7] Family=AdaptiveTruss-Tension, Ty
  [8] Family=AdaptiveTruss-Tension, Ty
  [9] Family=AdaptiveTruss-Tension, Ty
```

Using the same graph of nodes, select the truss element rather than the beam element. Before doing this, delete the Element.Geometry from the previous step.



1. In the *Watch* node, we can see that we have a list of adaptive components selected from Revit. We want to extract the basic information, so we're start with the adaptive points.
2. Plug the *"All Elements of Family Type"* node into the *"AdaptiveComponent.Location"* node. This gives us a list of lists, each with three points which represent the adaptive point locations.
3. Connecting a *"Polygon.ByPoints"* node returns a polycurve. We can see this in the Dynamo viewport. By this method, we've visualized the geometry of one element and abstracted the geometry of the remaining array of elements (which could be larger in number than this example).

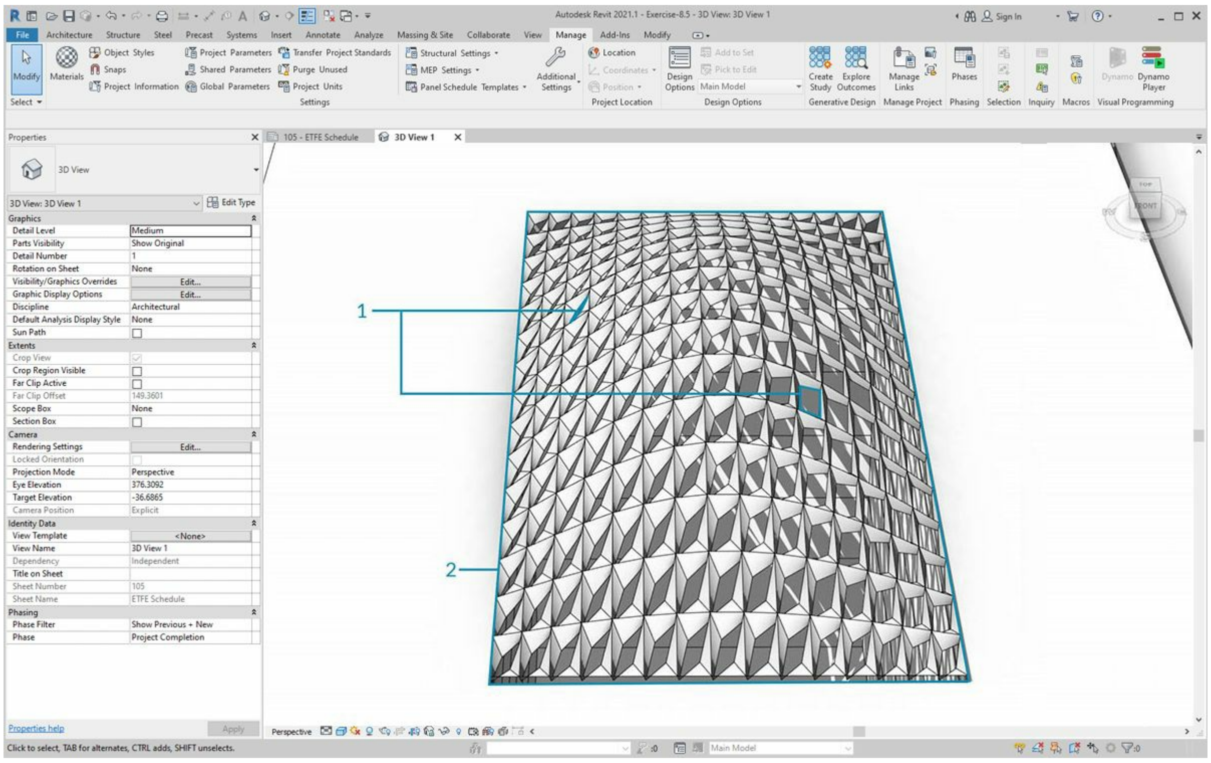
**Tip: if you click on the green number of a Revit element in Dynamo, the Revit viewport will zoom to that element.*

Editing

Editing

A powerful feature of Dynamo is that you can edit parameters on a parametric level. For example, a generative algorithm or the results of a simulation can be used to drive the parameters of an array of elements. This way, a set of instances from the same family can have custom properties in your Revit project.

Type and Instance Parameters



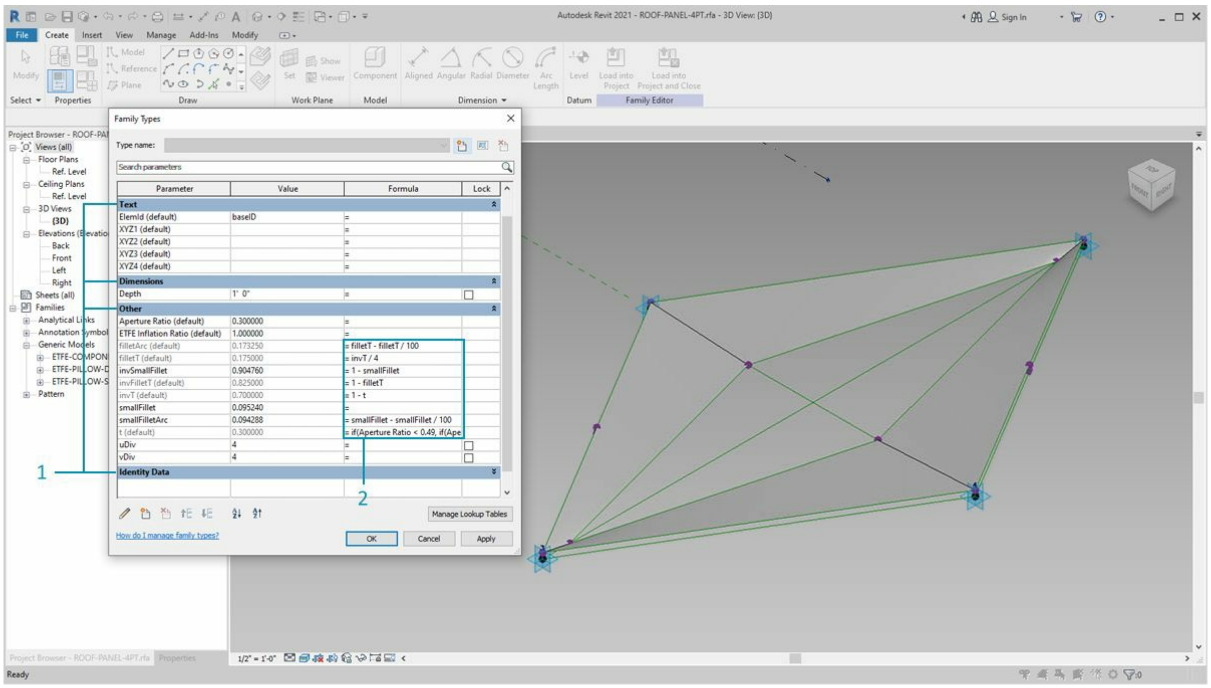
Properties — 3D View

3D View: 3D View 1

Graphics	
Detail Level	Medium
Parts Visibility	Show Original
Detail Number	1
Rotation on Sheet	None
Visibility/Graphics Overrides	Edit...
Graphic Display Options	Edit...
Discipline	Architectural
Default Analysis Display Style	None
Sun Path	☐
Extents	
Crop View	✓
Crop Region Visible	☐
Far Clip Active	☐
Far Clip Offset	149.3601
Scope Box	None
Section Box	☐
Camera	
Rendering Settings	Edit...
Locked Orientation	
Projection Mode	Perspective
Eye Elevation	376.3092
Target Elevation	-36.6865
Camera Position	Explicit
Identity Data	
View Template	<None>
View Name	3D View 1
Dependency	Independent
Title on Sheet	
Sheet Number	105
Sheet Name	ETFE Schedule
Phasing	
Phase Filter	Show Previous + New
Phase	Project Completion

Click to select, TAB for alternates, CTRL adds, SHIFT unselects.

1. Instance parameters define the aperture of the panels on the roof surface, ranging from an Aperture Ratio of 0.1 to 0.4.
2. Type-based parameters are applied to every element on the surface because they are the same family type. The material of each panel, for example, can be driven by a type-based parameter.



Family Types

Parameter	Value	Formula	Lock
Text			
ElemId (default)	baseID	=	
XYZ1 (default)		=	
XYZ2 (default)		=	
XYZ3 (default)		=	
XYZ4 (default)		=	
Dimensions			
Depth	1' 0"	=	☐
Other			
Aperture Ratio (default)	0.300000	=	
ETFE Inflation Ratio (default)	1.000000	=	
filletArc (default)	0.173250	= filletT - filletT / 100	
filletT (default)	0.175000	= invT / 4	
invSmallFillet	0.904760	= 1 - smallFillet	
invFilletT (default)	0.825000	= 1 - filletT	
invT (default)	0.700000	= 1 - t	
smallFillet	0.095240	=	
smallFilletArc	0.094288	= smallFillet - smallFillet / 100	
t (default)	0.300000	= if(Aperture Ratio < 0.49, if(Ape	
uDiv	4	=	☐
vDiv	4	=	☐
Identity Data			

1. If you've set up a Revit family before, remember that you have to assign a parameter type (string, number, dimension, etc.) Be sure to use the correct data type when assigning parameters from Dynamo.

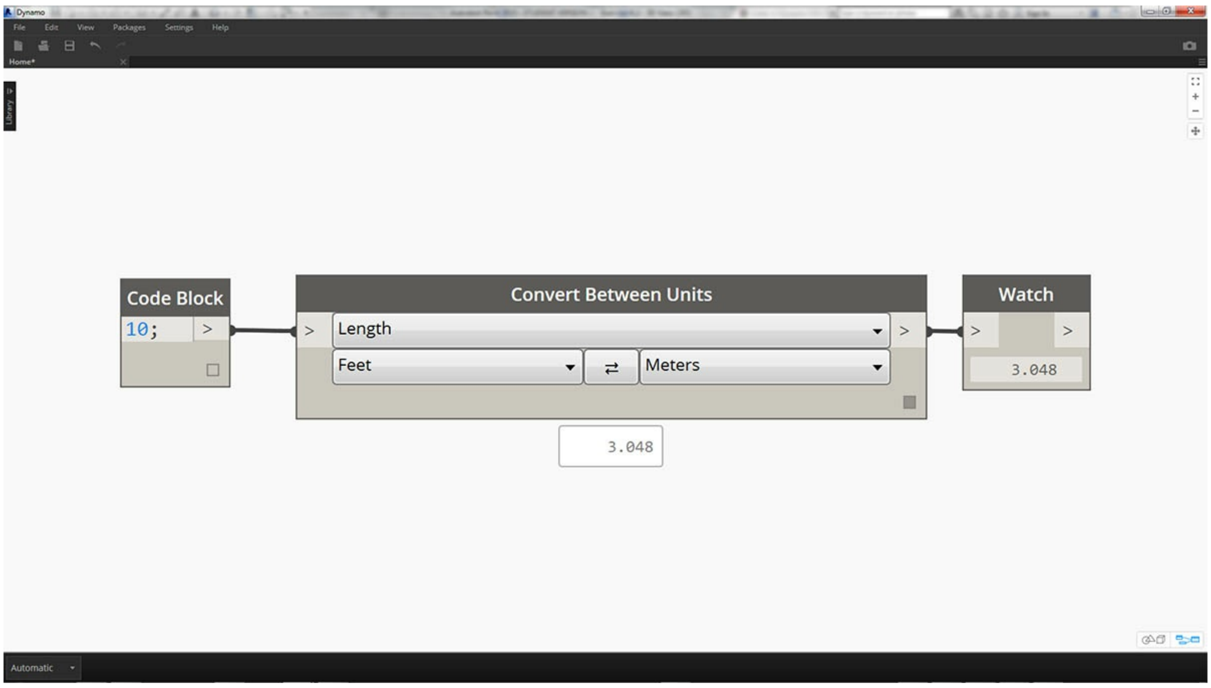
2. You can also use Dynamo in combination with parametric constraints defined in a Revit family's properties.

As a quick review of parameters in Revit, we recall that there are type parameters and instance parameters. Both can be edited from Dynamo, but we'll work with instance parameters in the exercise below.

Note: As you discover the wide-reaching application of editing parameters, you may want to edit a large quantity of elements in Revit with Dynamo. This can be a *computationally expensive* operation, meaning that it can be slow. If you're editing a large number of elements, you may want to use the "freeze" node functionality in order to pause the execution of Revit operations while you develop your graph. For more information on freezing nodes, check out the "Freezing" section in the solids chapter.

Units

As of version 0.8, Dynamo is fundamentally unitless. This allows Dynamo to remain an abstract visual programming environment. Dynamo nodes that interact with Revit dimensions will reference the Revit project's units. For example, if you are setting a length parameter in Revit from Dynamo, the number in Dynamo for the value will correspond to the default units in the Revit project. The exercise below works in meters.



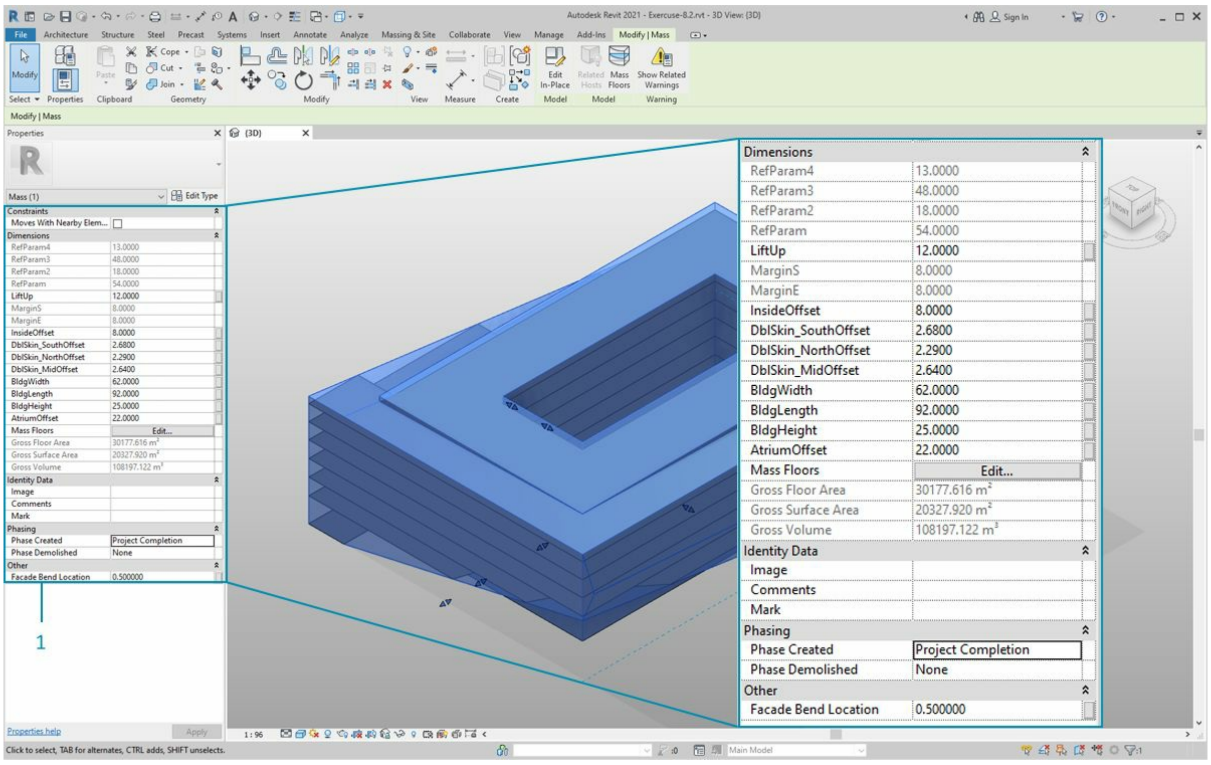
For a quick conversion of units, use the *"Convert Between Units"* node. This is a handy tool for converting Length, Area, and Volume units on the fly.

Exercise

Download the example files that accompanies this exercise (Right click and "Save Link As..."). A full list of example files can be found in the Appendix.

1. Editing.dyn
2. ARCH-Editing-BaseFile.rvt

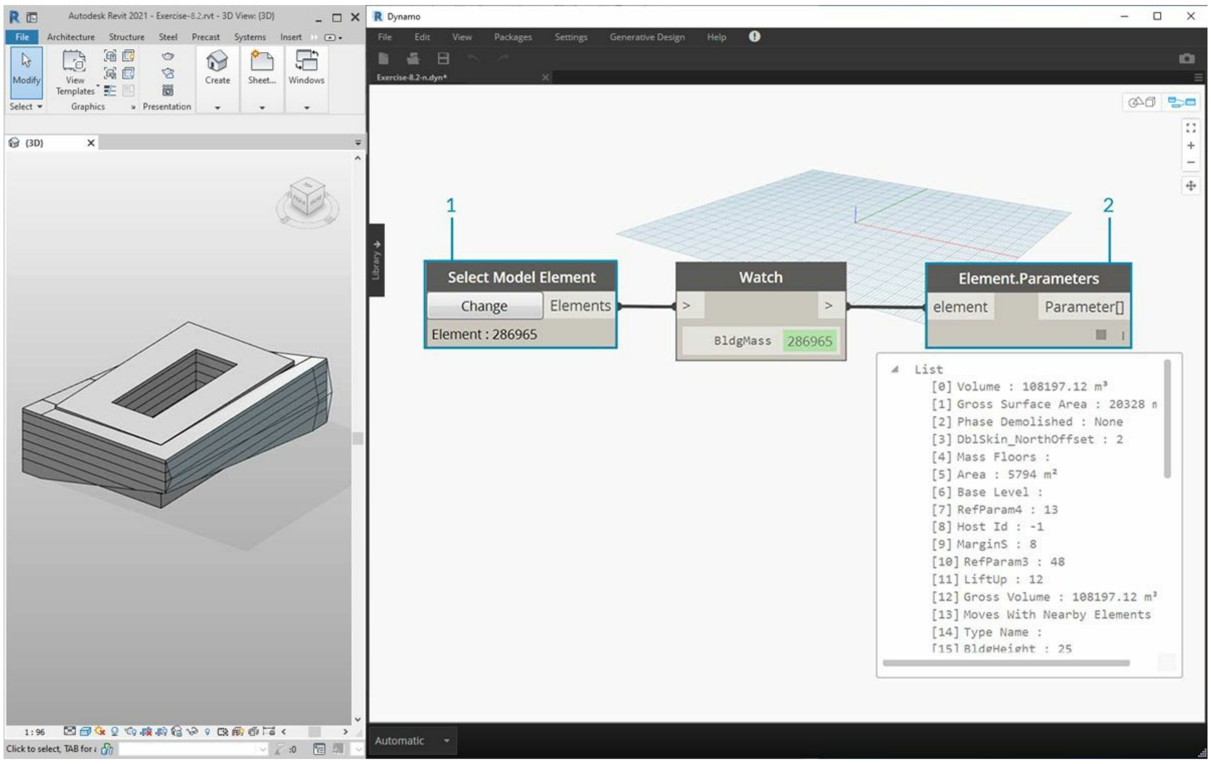
This exercise focuses on editing Revit elements without performing geometric operation in Dynamo. We're not importing Dynamo geometry here, just editing parameters in a Revit project. This exercise is basic, and to the more advanced Revit users, notice that these are instance parameters of a mass, but the same logic can be applied to an array of elements to customize on a large scale. This is all done with the "Element.SetParameterByName" node.



Dimensions	
RefParam4	13.0000
RefParam3	48.0000
RefParam2	18.0000
RefParam	54.0000
LiftUp	12.0000
MarginS	8.0000
MarginE	8.0000
InsideOffset	8.0000
DblSkin_SouthOffset	2.6800
DblSkin_NorthOffset	2.2900
DblSkin_MidOffset	2.6400
BldgWidth	62.0000
BldgLength	92.0000
BldgHeight	25.0000
AtriumOffset	22.0000
Mass Floors	Edit...
Gross Floor Area	30177.616 m²
Gross Surface Area	20327.920 m²
Gross Volume	108197.122 m³
Identity Data	
Image	
Comments	
Mark	
Phasing	
Phase Created	Project Completion
Phase Demolished	None
Other	
Facade Bend Location	0.500000

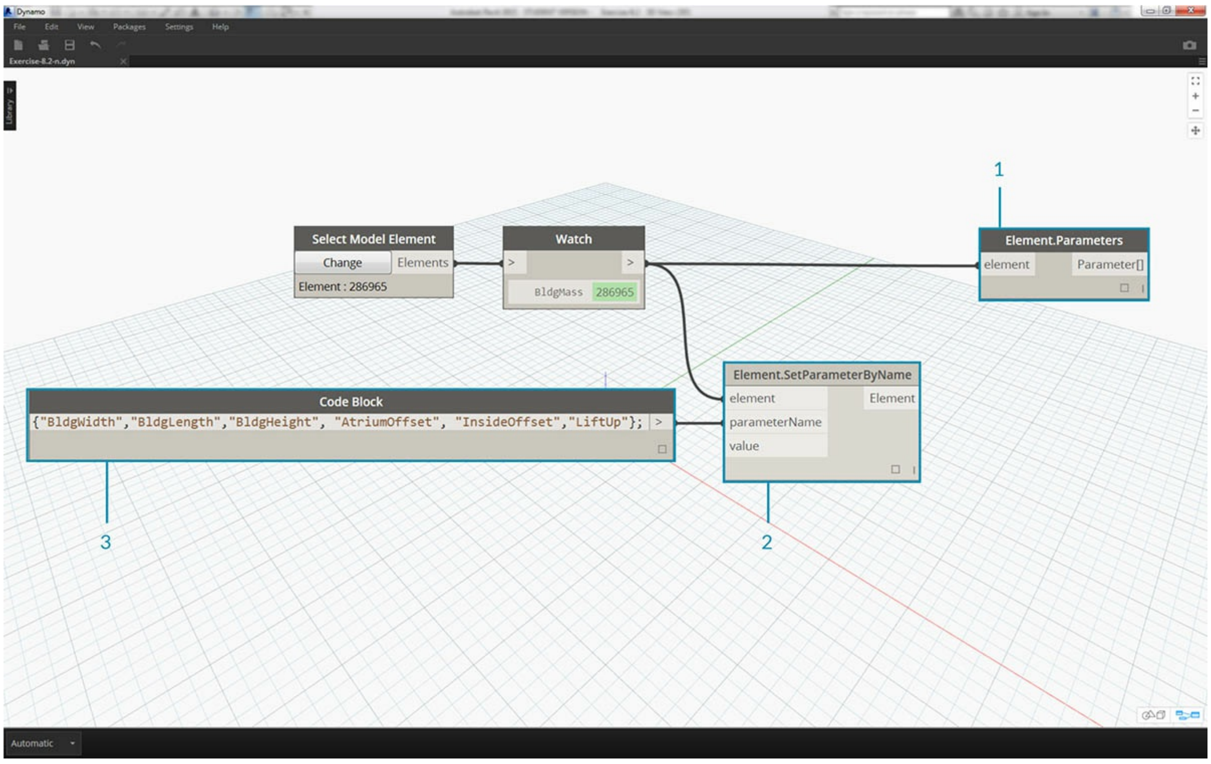
Begin with the example Revit file for this section. We've removed the structural elements and adaptive trusses from the previous section. In this exercise, we will focus on a parametric rig in Revit and manipulating in Dynamo.

1. Selecting the building in Mass in Revit, we see an array of instance parameters in the properties panel.



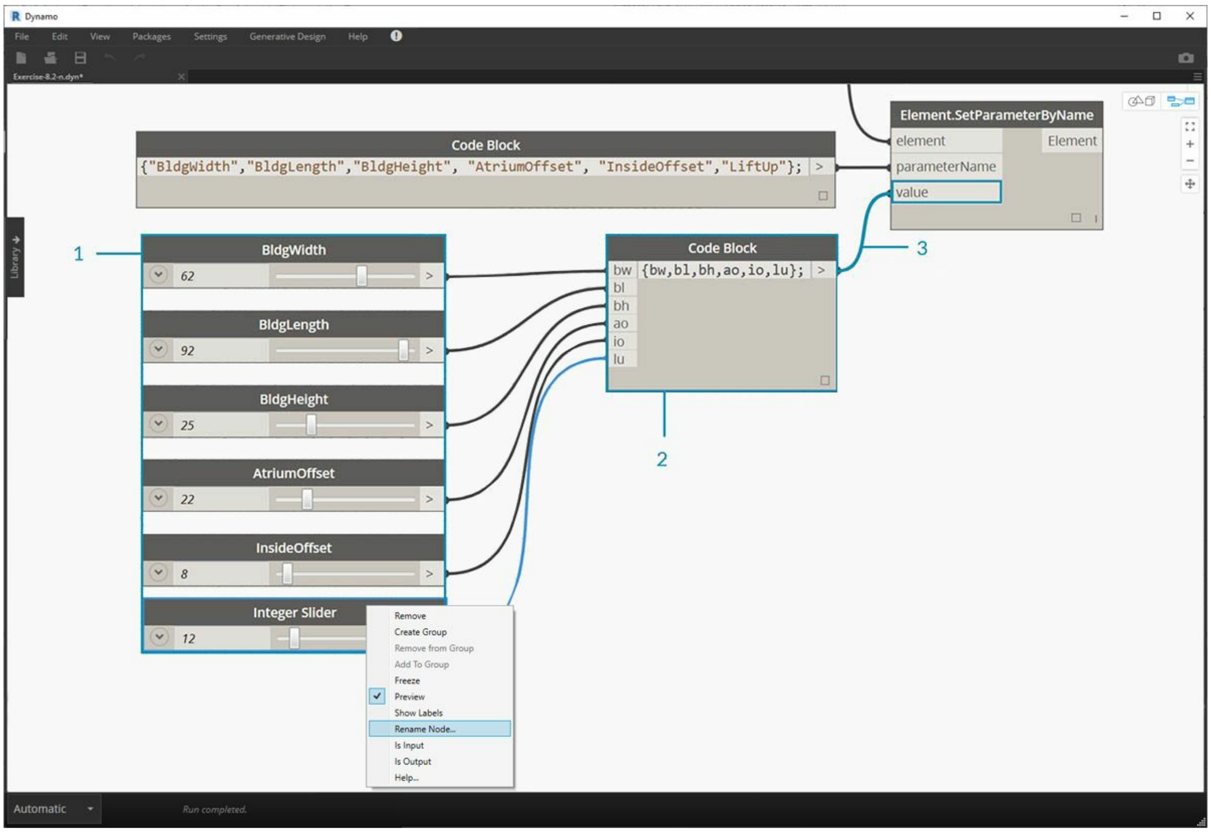
```
List
  [0] Volume : 108197.12 m³
  [1] Gross Surface Area : 20328 m
  [2] Phase Demolished : None
  [3] DblSkin_NorthOffset : 2
  [4] Mass Floors :
  [5] Area : 5794 m²
  [6] Base Level :
  [7] RefParam4 : 13
  [8] Host Id : -1
  [9] MarginS : 8
  [10] RefParam3 : 48
  [11] LiftUp : 12
  [12] Gross Volume : 108197.12 m³
  [13] Moves With Nearby Elements
  [14] Type Name :
  [15] BldgHeight : 25
```

1. Select the building mass with the *"Select Model Element"* node.
2. We can query all of the parameters of this mass with the *"Element.Parameters"* node. This includes type and instance parameters.



```
{"BldgWidth","BldgLength","BldgHeight", "AtriumOffset", "InsideOffset","LiftUp"};
```

1. Reference the *Element.Parameters* node to find target parameters. Or, we can view the properties panel from the previous step to choose which parameter names we want to edit. In this case, we are looking for the parameters which affect the large geometric moves on the building mass.
2. We will make changes to the Revit element using the *Element.SetParameterByName* node.
3. Using the *code block*, we define a list of these parameters, with quotes around each item to denote a string. We can also use the List.Create node with a series of *"string"* nodes connected to multiple inputs. Code block is simply faster and easier. Just make sure that the string matches the exact name in Revit, case-specific: `{"BldgWidth","BldgLength","BldgHeight", "AtriumOffset", "InsideOffset","LiftUp"};`

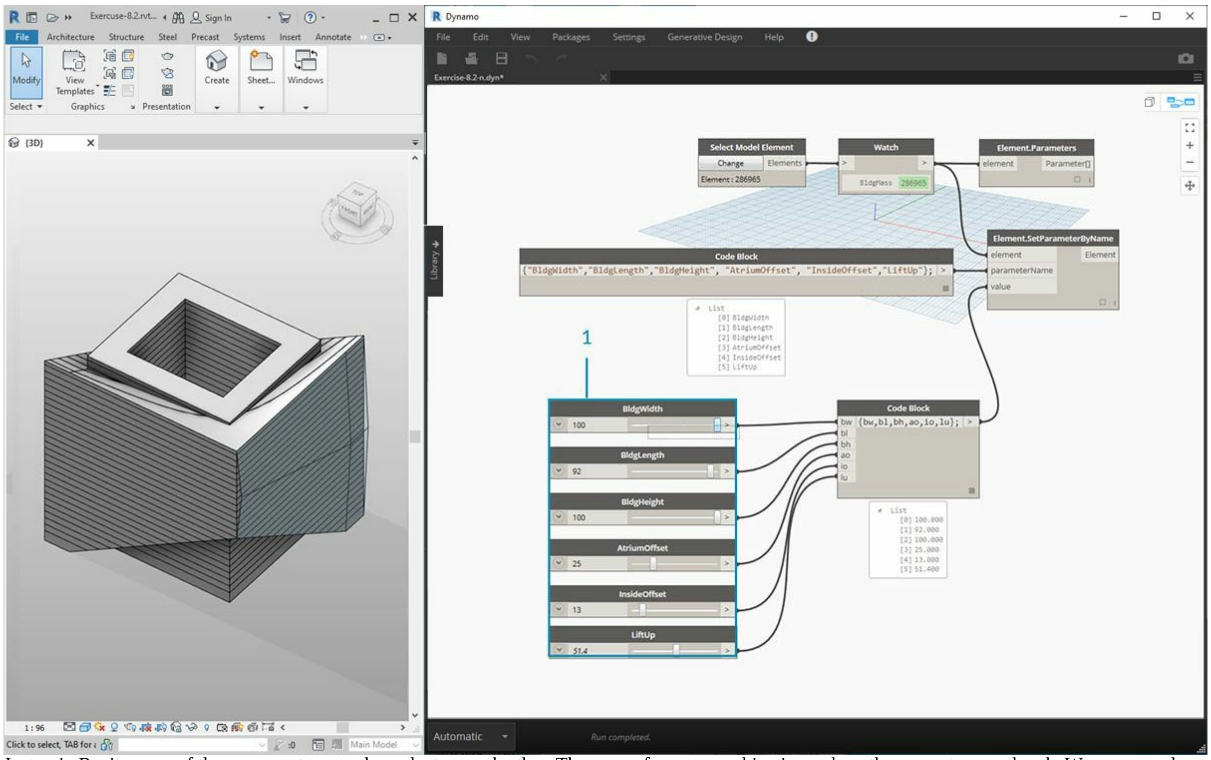


```
{"BldgWidth","BldgLength","BldgHeight", "AtriumOffset", "InsideOffset","LiftUp"};
{bw,bl,bh,ao,io,lu};
```

1. We also want to designate values for each parameter. Add six *"integer sliders"* to the canvas and rename to the corresponding parameter in the list. Also, set the values of each slider to the image above. In order from top-to-bottom: `62,92,25,22,8,12`
2. Define another *code block* with a list of the same length as the parameter names. In this case, we name variables (without quotes) which create

inputs for the *code block*. Plug the *sliders* into each respective input: `{bw,bl,bh,ao,io,lu};`
3. Connect the *code block* to the *"Element.SetParameterByName"** node. With run automatically checked, we will automatically see results.

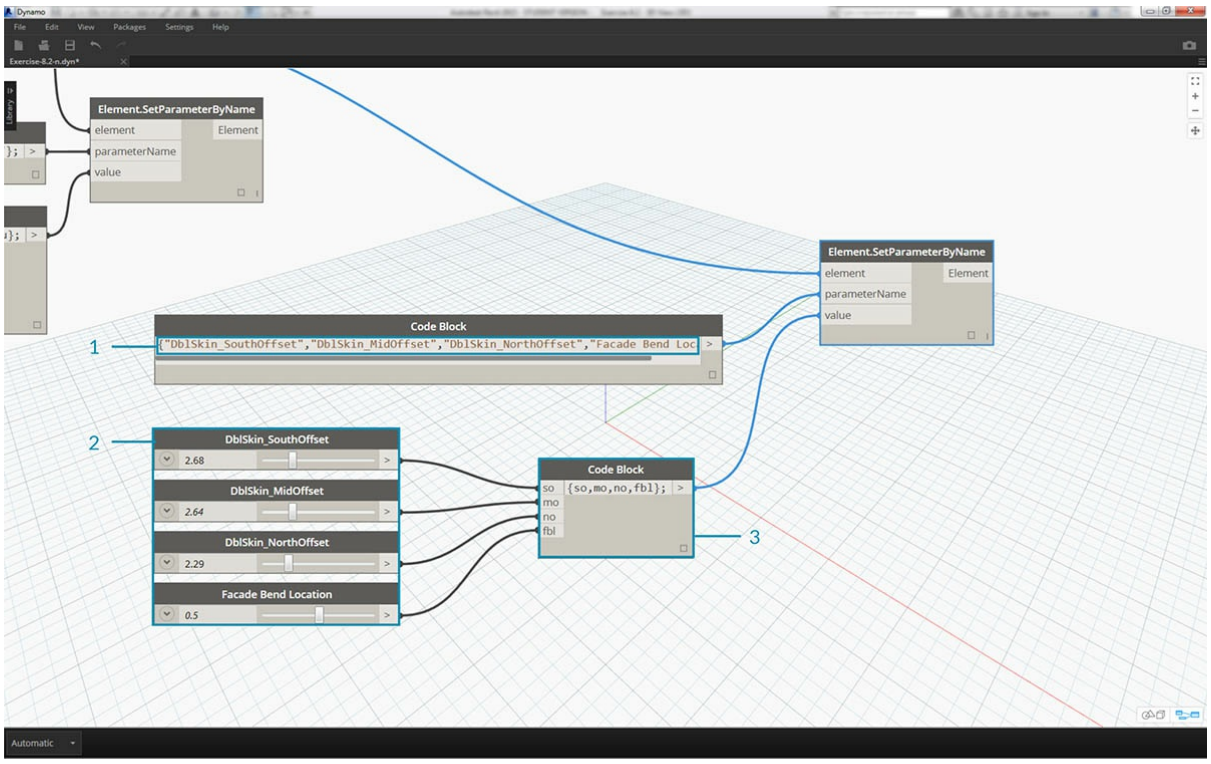
**Note - this demonstration works with instance parameters, but not type parameters.*



```
{"BldgWidth","BldgLength","BldgHeight", "AtriumOffset", "InsideOffset","LiftUp"};
{bw,bl,bh,ao,io,lu};
```

Just as in Revit, many of these parameters are dependent on each other. There are of course combinations where the geometry may break. We can remedy this issue with defined formulas in the parameter properties, or we can setup a similar logic with math operations in Dynamo (this is an additional challenge if you'd like to expand on the exercise).

1. This combination gives a funky new design to the building mass: `100,92,100,25,13,51.4`

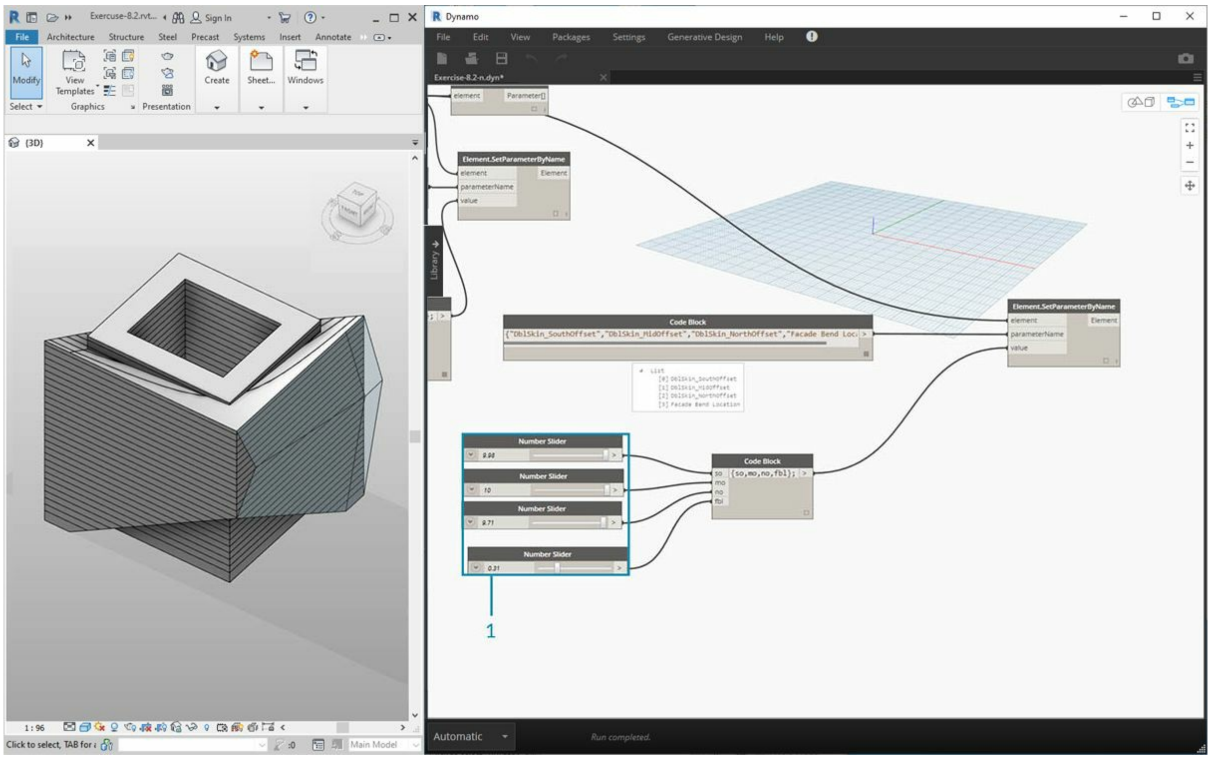


```
{"DblSkin_SouthOffset","DblSkin_MidOffset","DblSkin_NorthOffset","Facade Bend Loc
{so,mo,no,fbl};
```

1. Let's copy the graph and focus on the facade glazing which will house the truss system. We isolate four parameters in this case: `{"DblSkin_SouthOffset","DblSkin_MidOffset","DblSkin_NorthOffset","Facade Bend Location"};`
2. Additionally, we create *number sliders* and rename to the appropriate parameters. The first three sliders from top-to-bottom should be remapped to a domain of [0,10], while the final slider, *"Facade Bend Location"*, should be remapped to a domain of [0,1]. These values, from

top-to-bottom should start with these values (although they're arbitrary): `2.68,2.64,2.29,0.5`

3. Define a new *code block* and connect the sliders: `{so,mo,no,fbl};`



```
{"DblSkin_SouthOffset","DblSkin_MidOffset","DblSkin_NorthOffset","Facade Bend Loc:
```

```
List
  [0] DblSkin_SouthOffset
  [1] DblSkin_MidOffset
  [2] DblSkin_NorthOffset
  [3] Facade Bend Location
```

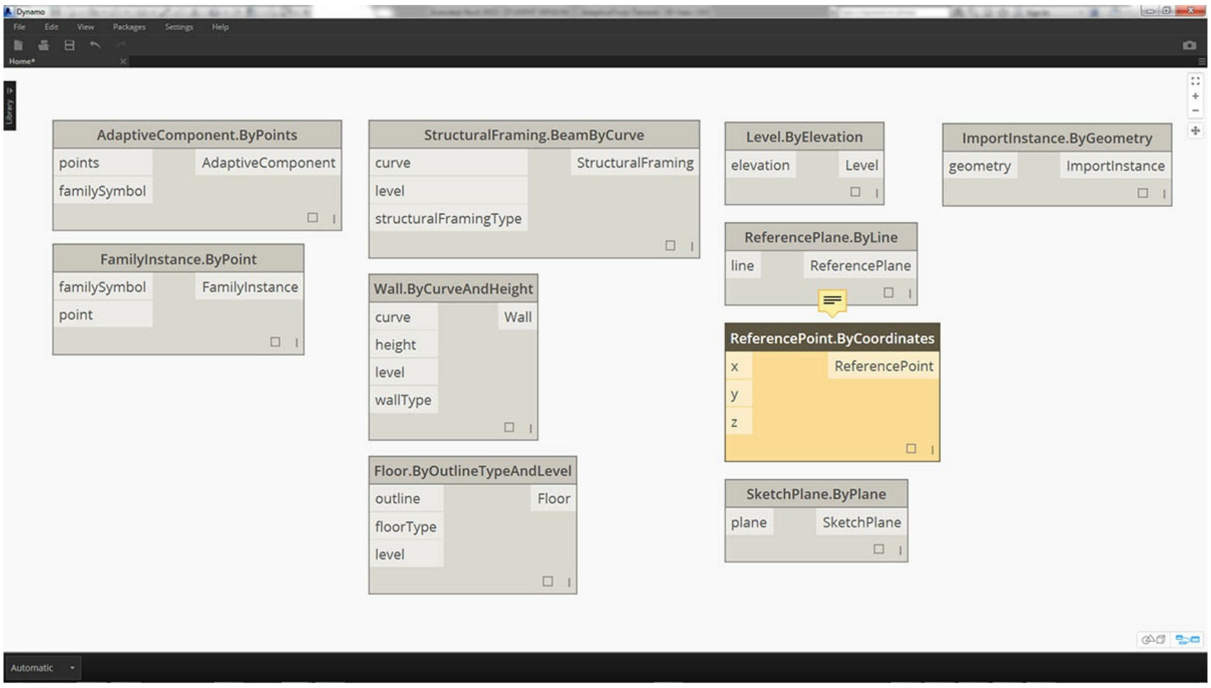
```
{so,mo,no,fbl};
```

1. By changing the *sliders* in this part of the graph, we can make the facade glazing much more substantial: `9.98,10.0,9.71,0.31`

Creating

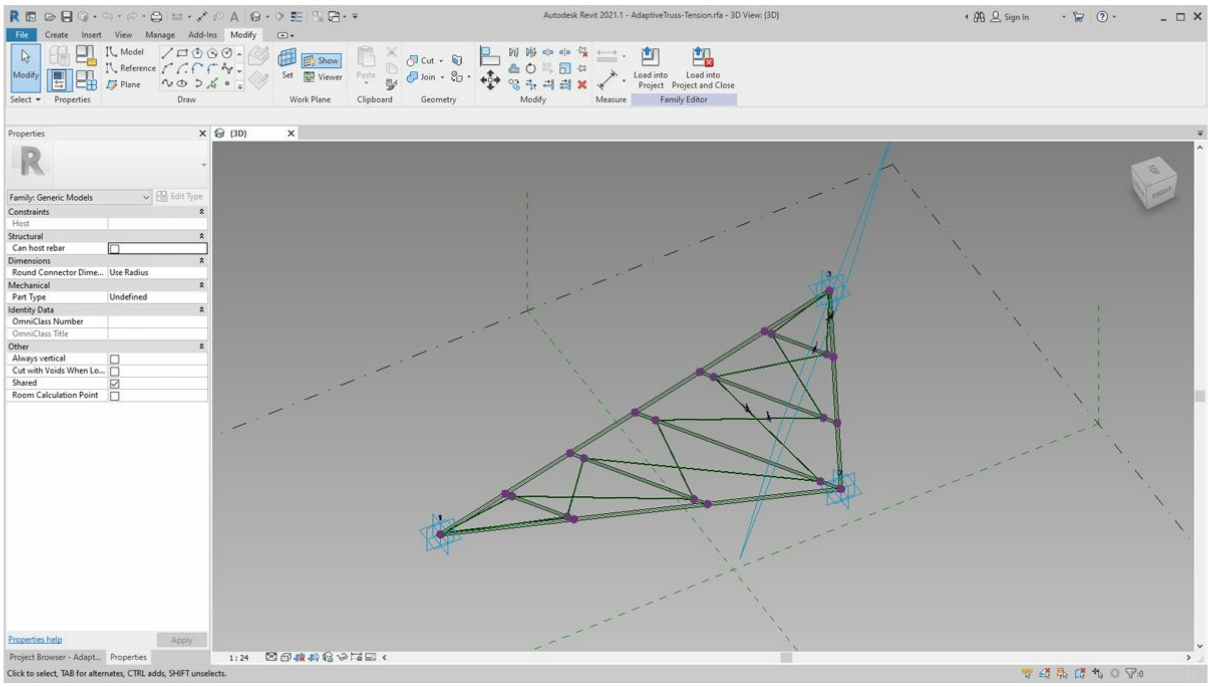
Creating

You can create an array of Revit elements in Dynamo with full parametric control. The Revit nodes in Dynamo offer the ability to import elements from generic geometries to specific category types (like walls and floors). In this section, we'll focus on importing parametrically flexible elements with adaptive components.



Adaptive Components

An adaptive component is a flexible family category which lends itself well to generative applications. Upon instantiation, you can create a complex geometric element which is driven by the fundamental location of adaptive points.


Autodesk Revit 2021.1 - AdaptiveTruss-Tension.rfa - 3D View: {3D}
Properties
Family: Generic Models
{3D}
Load into Project
Load into Project and Close
Family Editor

An example of a three-point adaptive component in the family editor. This generates a truss which is defined by the position of each adaptive point. In the exercise below, we'll use this component to generate a series of trusses across a facade.

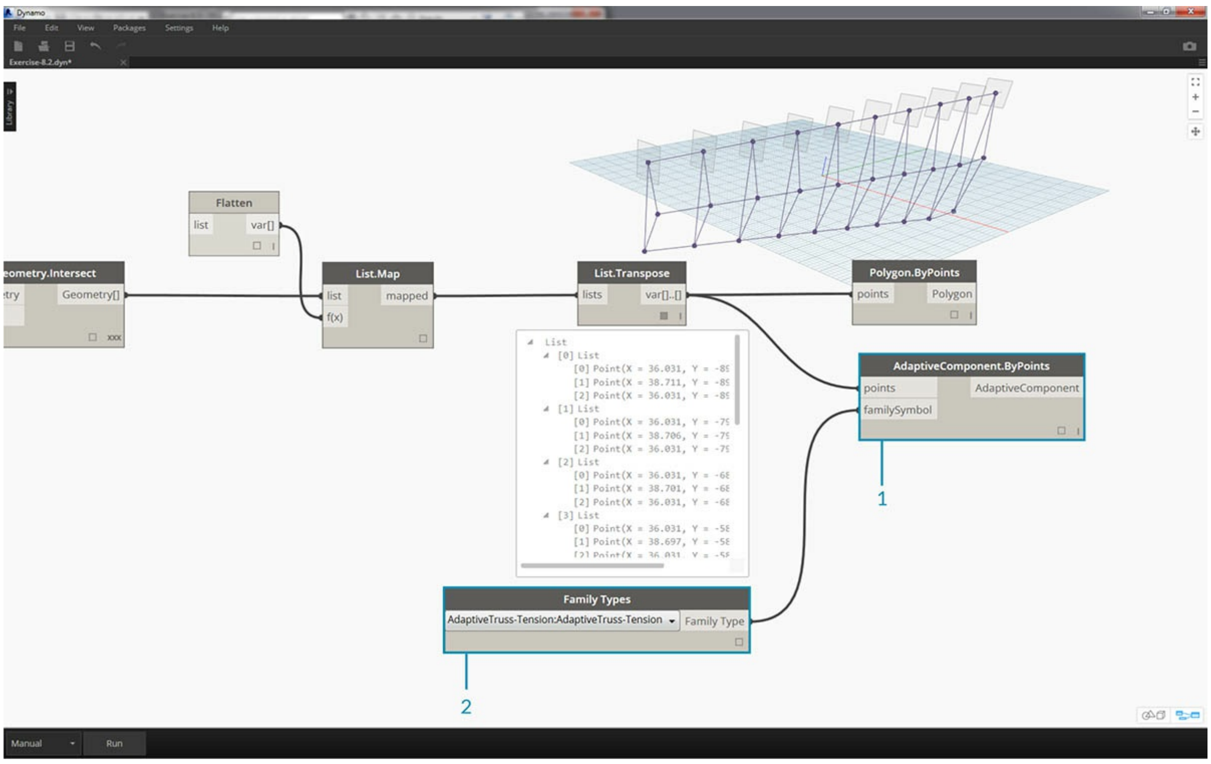
Principles of Interoperability

The adaptive component is a good example for best practices of interoperability. We can create an array of adaptive components by defining the fundamental adaptive points. And, when transferring this data to other programs, we have the ability to reduce the geometry to simple data. Importing and exporting with a program like Excel follows a similar logic.

Suppose a facade consultant wants to know the location of the truss elements without needing to parse through fully articulated geometry. In preparation for fabrication, the consultant can reference the location of adaptive points to regenerate geometry in a program like Inventor.

The workflow we'll setup in the exercise below allows us to access all of this data while creating the definition for Revit element creation. By this process, we can merge conceptualization, documentation, and fabrication into a seamless workflow. This creates a more intelligent and efficient process for interoperability.

Multiple Elements and Lists



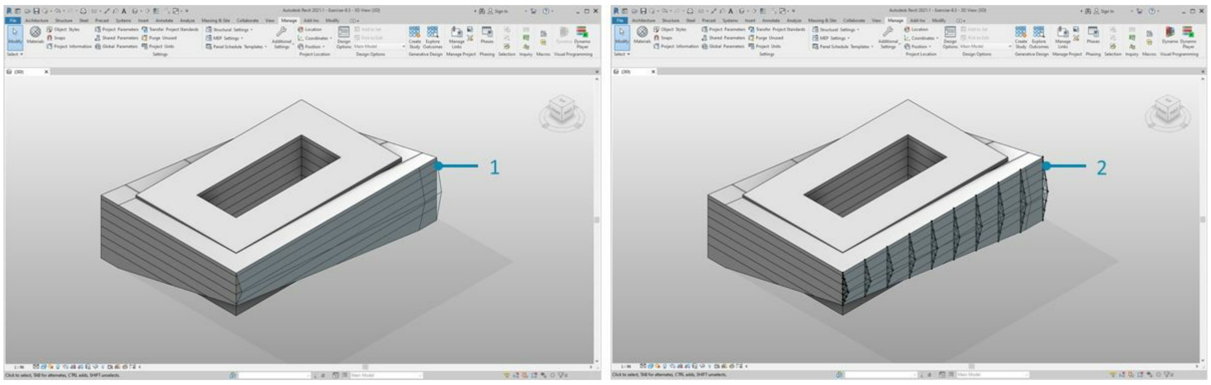
```
List
  [0] List
    [0] Point(X = 36.031, Y = -89
    [1] Point(X = 38.711, Y = -89
    [2] Point(X = 36.031, Y = -89
  [1] List
    [0] Point(X = 36.031, Y = -79
    [1] Point(X = 38.706, Y = -79
    [2] Point(X = 36.031, Y = -79
  [2] List
    [0] Point(X = 36.031, Y = -68
    [1] Point(X = 38.701, Y = -68
    [2] Point(X = 36.031, Y = -68
  [3] List
    [0] Point(X = 36.031, Y = -58
    [1] Point(X = 38.697, Y = -58
    [2] Point(X = 36.031, Y = -58
```

The exercise below will walk through how Dynamo references data for Revit element creation. To generate multiple adaptive components, we define a list of lists, where each list has three points representing each point of the adaptive component. We'll keep this in mind as we manage the data structures in Dynamo.

Exercise

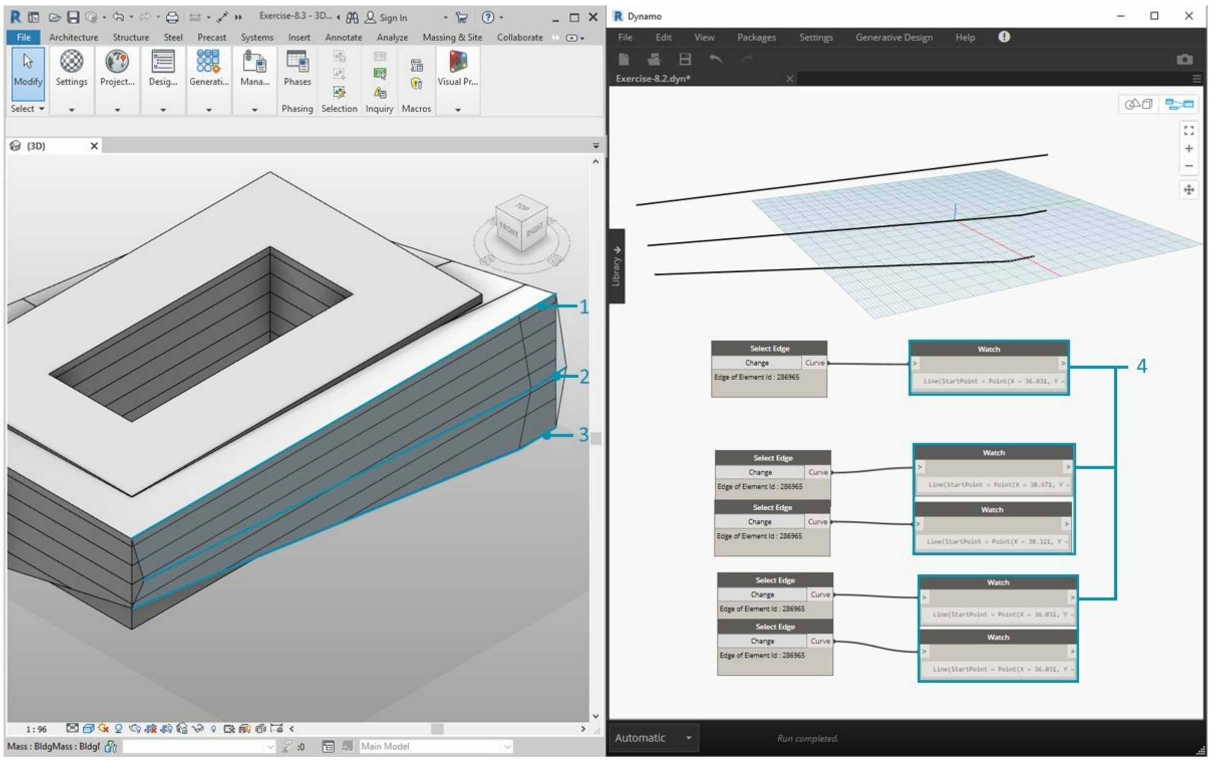
Download the example files that accompanies this exercise (Right click and "Save Link As..."). A full list of example files can be found in the Appendix.

1. Creating.dyn
2. ARCH-Creating-BaseFile.rvt



Beginning with the example file from this section (or continuing with the Revit file from the previous session), we see the same Revit mass.

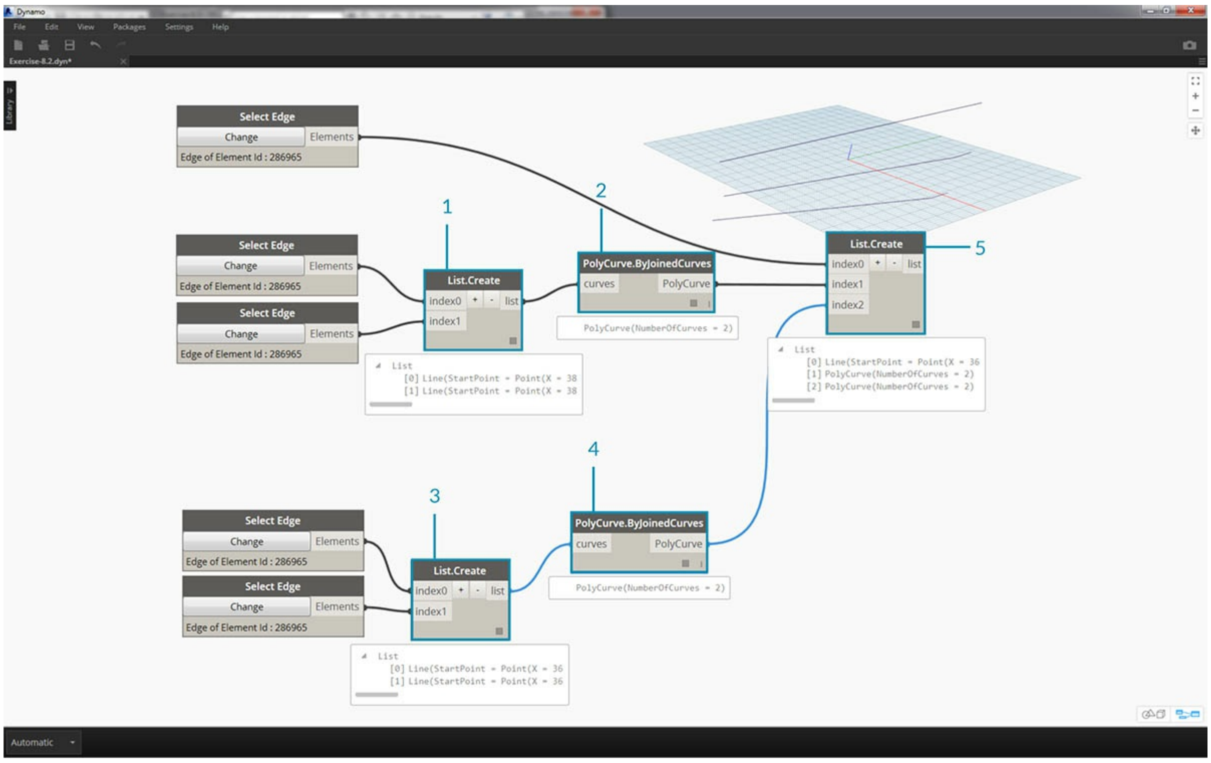
1. This is the file as opened.
2. This is the truss system we created with Dynamo, linked intelligently to the Revit mass.



We've used the *"Select Model Element"* and *"Select Face"* nodes, now we're taking one step further down in the geometry hierarchy and using *"Select Edge"*. With the Dynamo solver set to run *"Automatic"*, the graph will continually update to changes in the Revit file. The edge we are selecting is tied dynamically to the Revit element topology. As long as the topology* does not change, the connection remains linked between Revit and Dynamo.

1. Select the top most curve of the glazing facade. This spans the full length of the building. If you're having trouble selecting the edge, remember to choose the selection in Revit by hovering over the edge and hitting *"Tab"* until the desired edge is highlighted.
2. Using two *"Select Edge"* nodes, select each edge representing the cant at the middle of the facade.
3. Do the same for the bottom edges of the facade in Revit.
4. The *Watch* nodes reveal that we now have lines in Dynamo. This is automatically converted to Dynamo geometry since the edges themselves are not Revit elements. These curves are the references we'll use to instantiate adaptive trusses across the facade.

**Note - to keep a consistent topology, we're referring to a model that does not have additional faces or edges added. While parameters can change its shape, the way in which it is built remains consistent.*



```
PolyCurve(NumberOfCurves = 2)
```

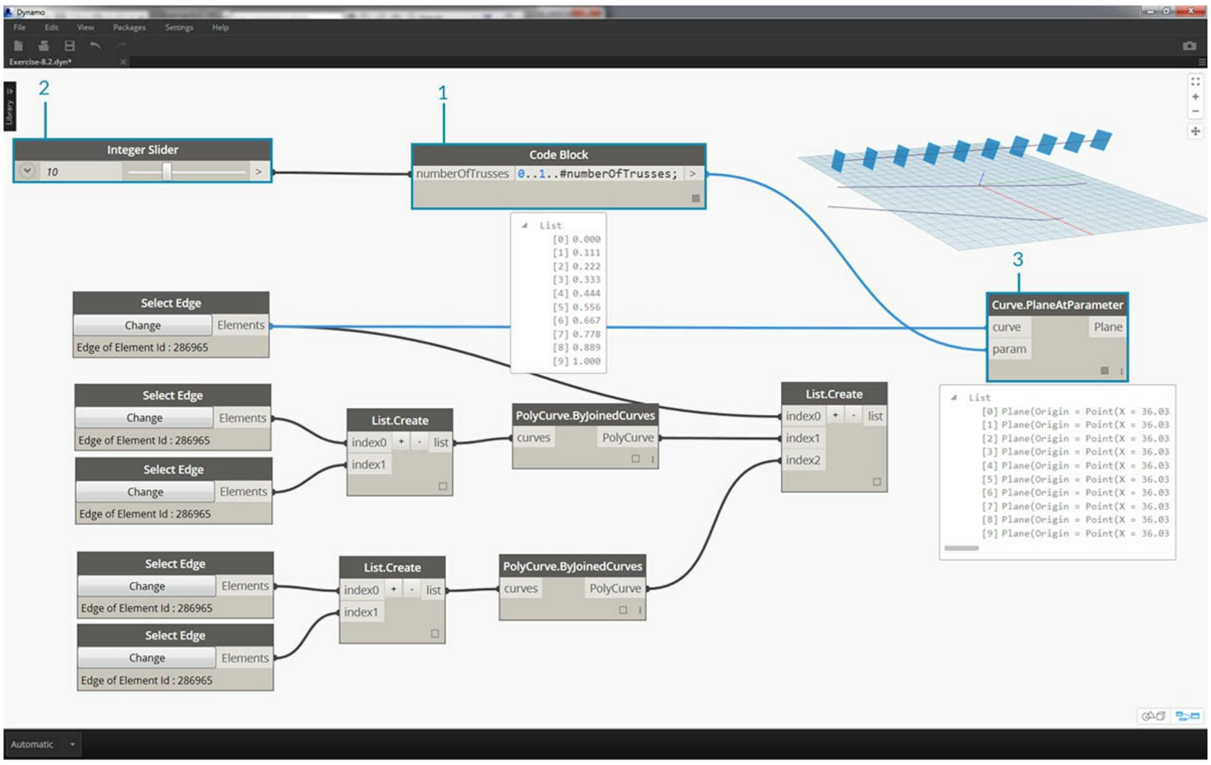
```
List
  [0] Line(StartPoint = Point(X = 38
  [1] Line(StartPoint = Point(X = 38
```

```
List
  [0] Line(StartPoint = Point(X = 36
  [1] PolyCurve(NumberOfCurves = 2)
  [2] PolyCurve(NumberOfCurves = 2)
```

```
List
  [0] Line(StartPoint = Point(X = 36
  [1] Line(StartPoint = Point(X = 36
```

We first need to join the curves and merge them into one list. This way we can *"group"* the curves to perform geometry operations.

1. Create a list for the two curves at the middle of the facade.
2. Join the two curves into a Polycurve by plugging the *List.Create* component into a *Polycurve.ByJoinedCurves* node.
3. Create a list for the two curves at the bottom of the facade.
4. Join the two curves into a Polycurve by plugging the *List.Create* component into a *Polycurve.ByJoinedCurves* node.
5. Finally, join the three main curves (one line and two polycurves) into one list.



```
0..1..#numberOfTrusses;
```

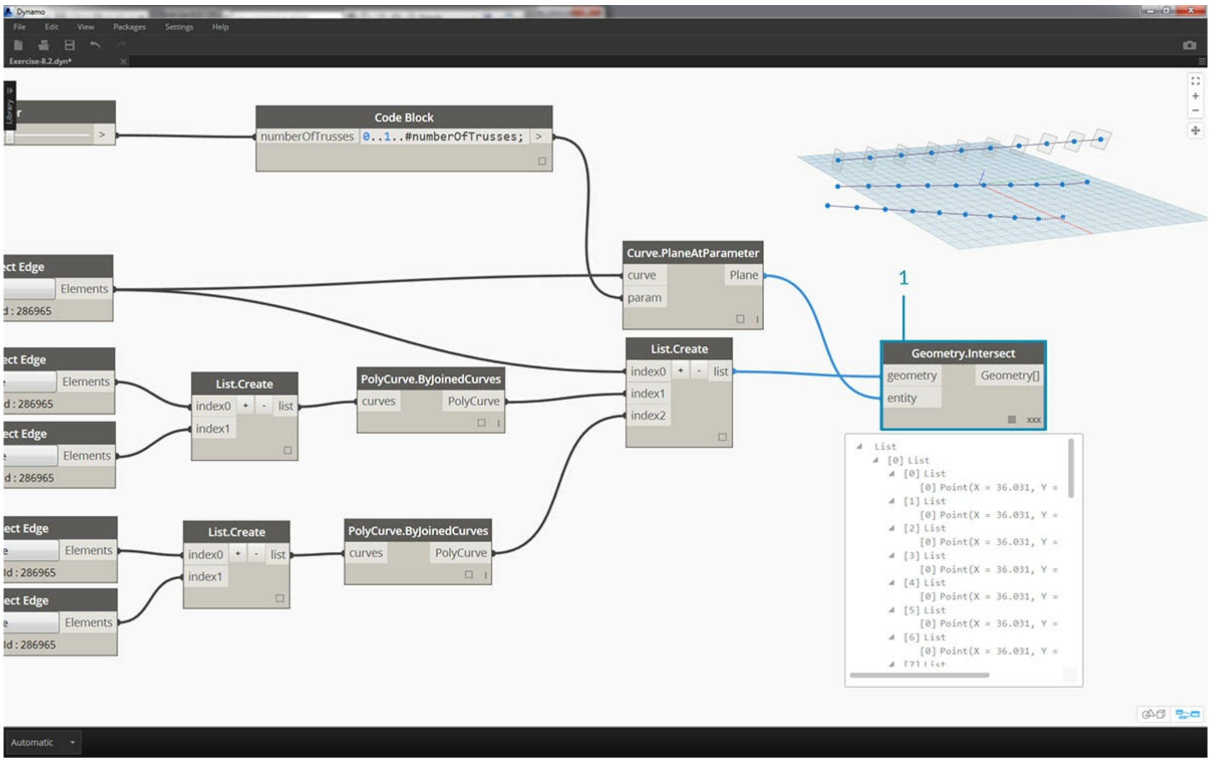
```
List
  [0] 0.000
  [1] 0.111
  [2] 0.222
  [3] 0.333
  [4] 0.444
  [5] 0.556
  [6] 0.667
  [7] 0.778
  [8] 0.889
  [9] 1.000
```

```
List
  [0] Plane(Origin = Point(X = 36.03
  [1] Plane(Origin = Point(X = 36.03
  [2] Plane(Origin = Point(X = 36.03
  [3] Plane(Origin = Point(X = 36.03
  [4] Plane(Origin = Point(X = 36.03
  [5] Plane(Origin = Point(X = 36.03
  [6] Plane(Origin = Point(X = 36.03
  [7] Plane(Origin = Point(X = 36.03
  [8] Plane(Origin = Point(X = 36.03
  [9] Plane(Origin = Point(X = 36.03
```

We want to take advantage of the top curve, which is a line, and represents the full span of the facade. We'll create planes along this line to intersect with the set of curves we've grouped together in a list.

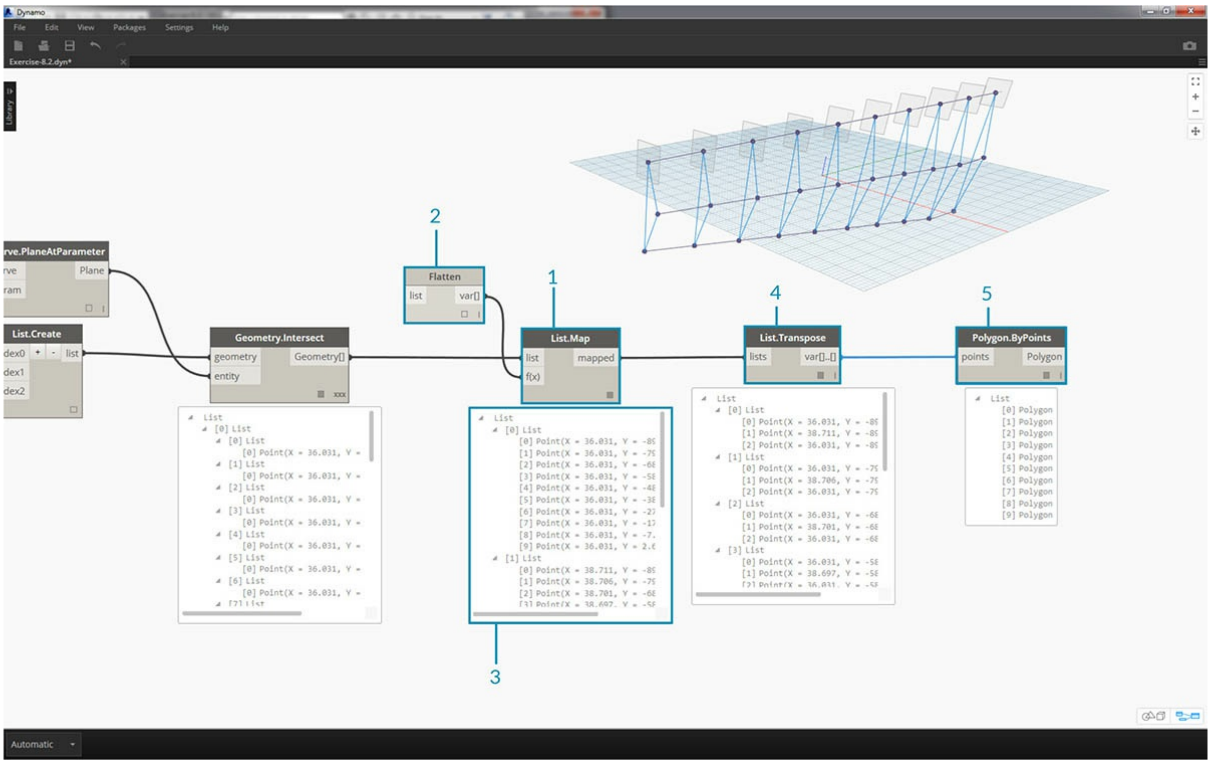
1. With a *code block*, define a range using the syntax: `0..1..#numberOfTrusses;`

1. Plug an *integer slider* into the input for the code block. As you could have guessed, this will represent the number of trusses. Notice that the slider controls the number of items in the range defined from *0* to *1*.
2. Plug the *code block* into the *param* input of a *"Curve.PlaneAtParameter"* node, and plug the top edge into the *curve* input. This will give us ten planes, evenly distributed across the span of the facade.



A plane is an abstract piece of geometry, representing a two dimensional space which is infinite. Planes are great for contouring and intersecting, as we are setting up in this step.

1. Using the *Geometry.Intersect* node (note the cross product lacing), plug the *Curve.PlaneAtParameter* into the *entity* input of the *Geometry.Intersect* node. Plug the main *List.Create* node into the *geometry* input. We now see points in the Dynamo viewport representing the intersection of each curve with the defined planes.

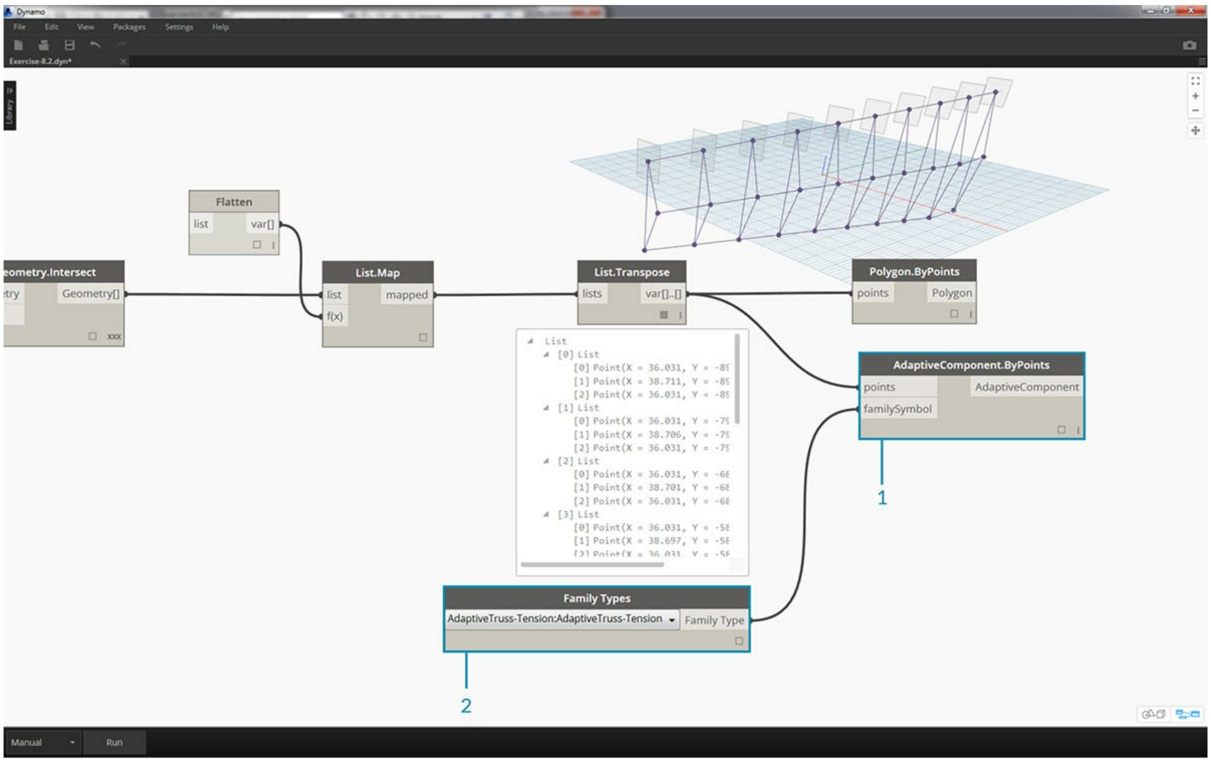


Notice the output is a list of lists of lists. Too many lists for our purposes. We want to do a partial flatten here. We need to take one step down on the list and flatten the result. To do this, we use the *List.Map* operation, as discussed in the list chapter of the primer.

1. Plug the *Geometry.Intersect* node into the list input of *List.Map*.
2. Plug a *Flatten* node into the f(x) input of *List.Map*. The results gives 3 list, each with a count equal to the number of trusses.
3. We need to change this data. If we want to instantiate the truss, we have to use the same number of adaptive points as defined in the family. This is a three point adaptive component, so instead of three lists with 10 items each (numberOfTrusses), we want 10 lists of three items each.

This way we can create 10 adaptive components.

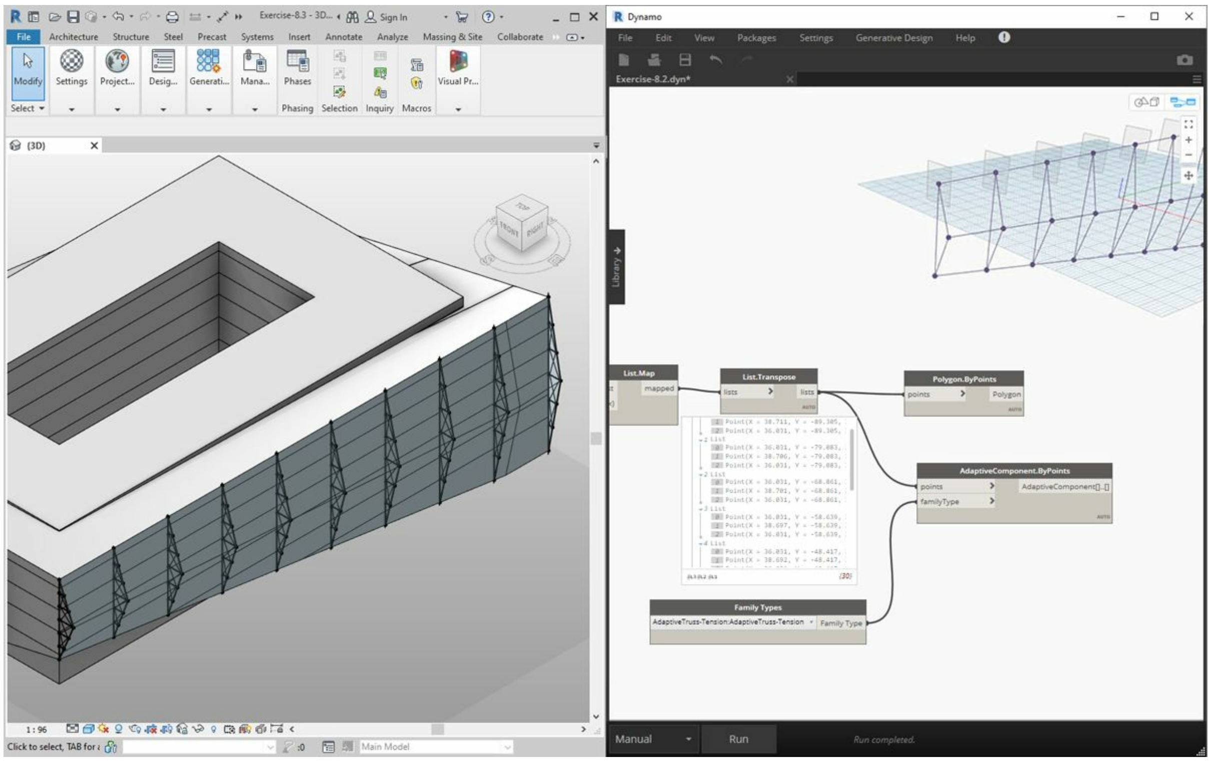
4. Plug the *List.Map* into a *List.Transpose* node. Now we have the desired data output.
5. To confirm that the data is correct, add a *Polygon.ByPoints* node to the canvas and double check with the Dynamo preview.



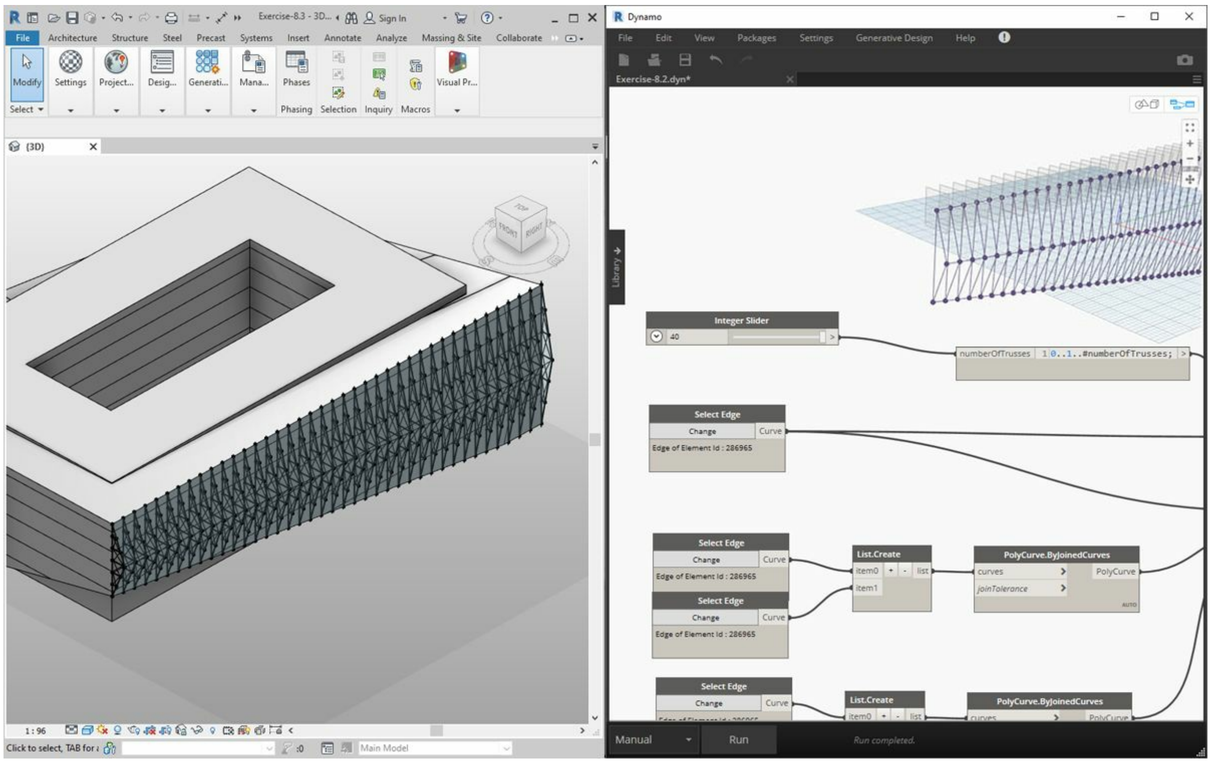
```
List
  [0] List
      [0] Point(X = 36.031, Y = -89
      [1] Point(X = 38.711, Y = -89
      [2] Point(X = 36.031, Y = -89
  [1] List
      [0] Point(X = 36.031, Y = -79
      [1] Point(X = 38.706, Y = -79
      [2] Point(X = 36.031, Y = -79
  [2] List
      [0] Point(X = 36.031, Y = -68
      [1] Point(X = 38.701, Y = -68
      [2] Point(X = 36.031, Y = -68
  [3] List
      [0] Point(X = 36.031, Y = -58
      [1] Point(X = 38.697, Y = -58
      [2] Point(X = 36.031, Y = -58
```

In the same way we created the polygons, we array the adaptive components.

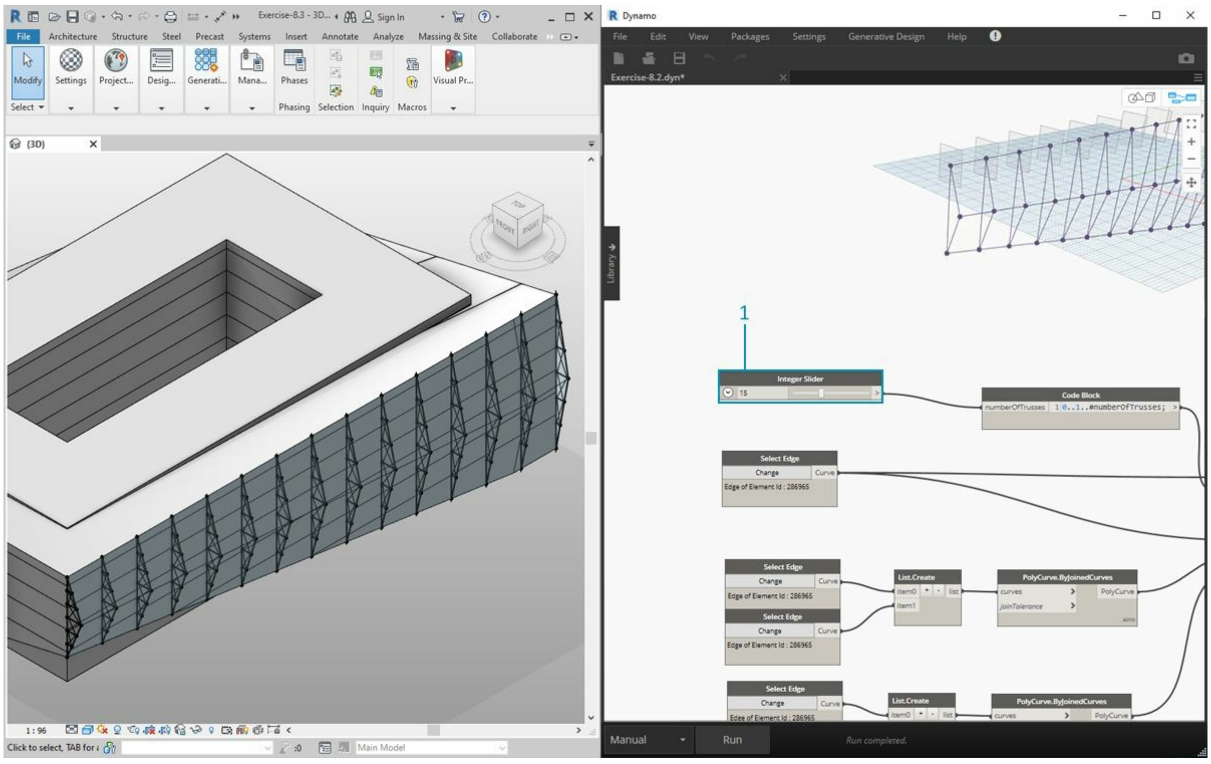
1. Add an *AdaptiveComponent.ByPoints* node to the canvas, plug the *List.Transpose* node into the *points* input.
2. Using a *Family Types* node, select the *"AdaptiveTruss"* family, and plug this into the *familySymbol* input of the *AdaptiveComponent.ByPoints* node.



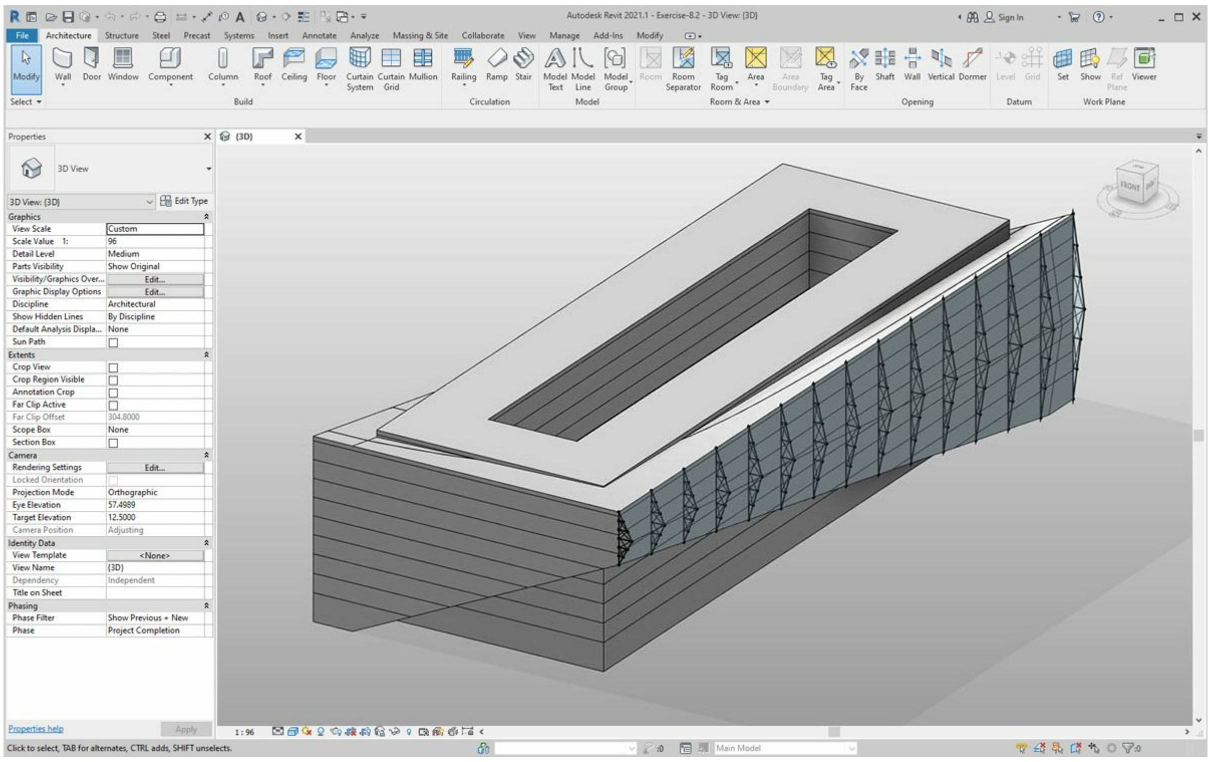
Checking in Revit, we now have the ten trusses evenly spaced across the facade!



1. "Flexing" the graph, we turn up the *numberOfTrusses* to *40* by changing the *slider*. Lots of trusses, not very realistic, but the parametric link is working.



1. Taming the truss system, let's compromise with a value of *15* for *numberOfTrusses*.


Autodesk Revit 2021.1 - Exercise-8.2 - 3D View: {3D}

Properties

3D View

3D View: {3D}

Graphics	
View Scale	Custom
Scale Value 1:	96
Detail Level	Medium
Parts Visibility	Show Original
Visibility/Graphics Over...	Edit...
Graphic Display Options	Edit...
Discipline	Architectural
Show Hidden Lines	By Discipline
Default Analysis Displa...	None
Sun Path	☐
Extents	
Crop View	☐
Crop Region Visible	☐
Annotation Crop	☐
Far Clip Active	☐
Far Clip Offset	304.8000
Scope Box	None
Section Box	☐
Camera	
Rendering Settings	Edit...
Locked Orientation	☐
Projection Mode	Orthographic
Eye Elevation	57.4989
Target Elevation	12.5000
Camera Position	Adjusting
Identity Data	
View Template	<None>
View Name	{3D}
Dependency	Independent
Title on Sheet	
Phasing	
Phase Filter	Show Previous + New
Phase	Project Completion

Click to select, TAB for alternates, CTRL adds, SHIFT unselects.

And for the final test, by selecting the mass in Revit and editing instance parameters, we can change the form of the building and watch the truss follow suit. Remember, this Dynamo graph has to be open in order to see this update, and the link will be broken as soon as it's closed.

DirectShape Elements

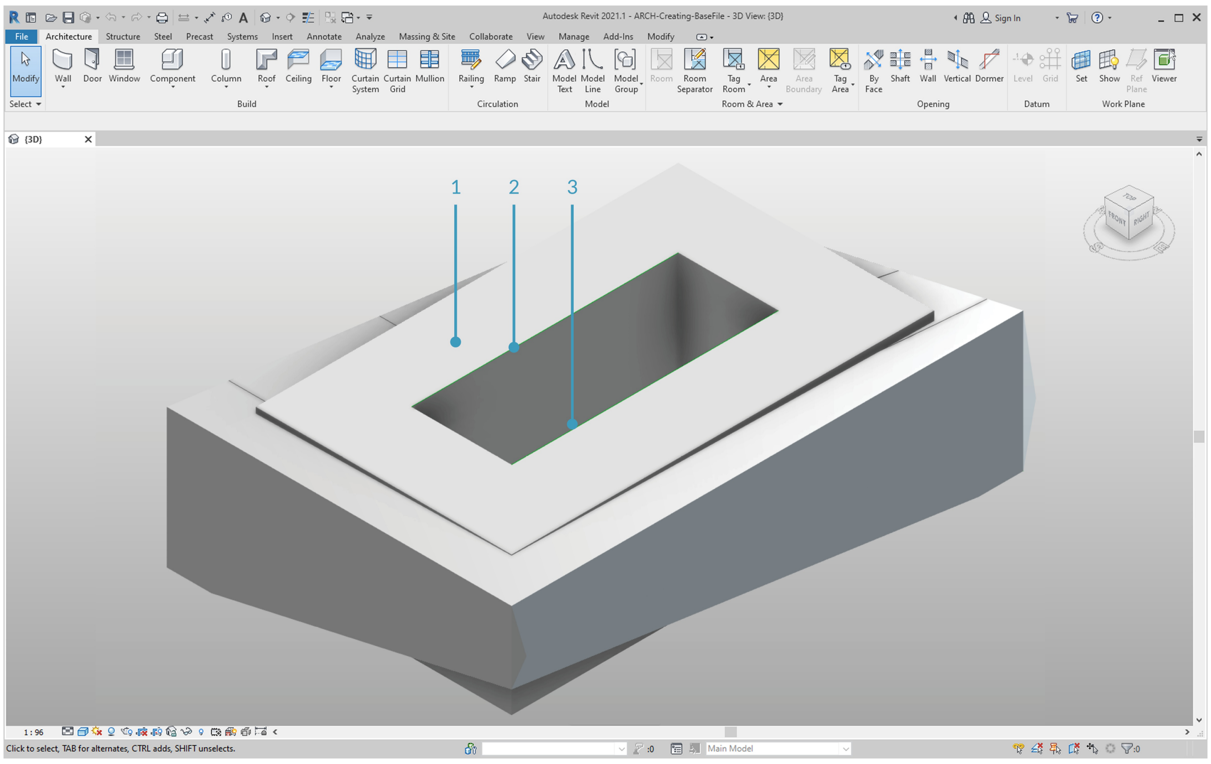
Another method for importing parametric Dynamo geometry into Revit is with DirectShape. In summary, the DirectShape element and related classes support the ability to store externally created geometric shapes in a Revit document. The geometry can include closed solids or meshes. DirectShape is primarily intended for importing shapes from other data formats such as IFC or STEP where not enough information is available to create a "real" Revit element. Like the IFC and STEP workflow, the DirectShape functionality works well with importing Dynamo created geometries into Revit projects as real elements.

Let's walk through and exercise for importing Dynamo geometry as a DirectShape into our Revit project. Using this method, we can assign an imported geometry's category, material, and name - all while maintaining a parametric link to our Dynamo graph.

Exercise

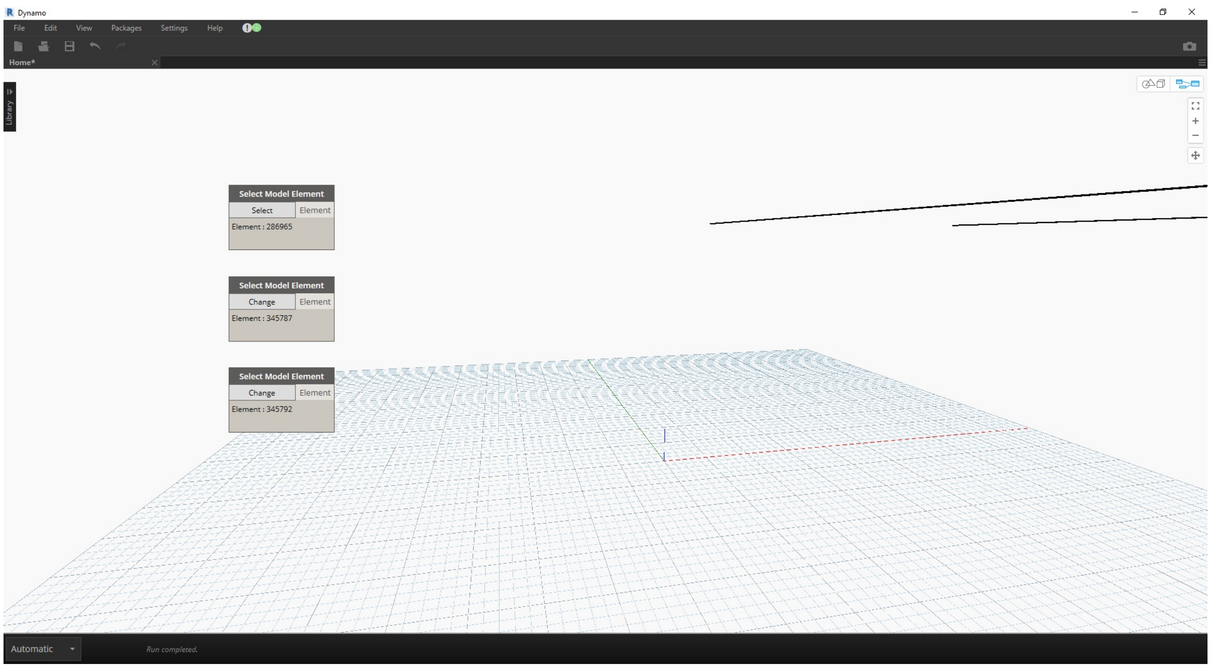
Download the example files that accompanies this exercise (Right click and "Save Link As..."). A full list of example files can be found in the Appendix.

1. DirectShape.dyn
2. ARCH-DirectShape-BaseFile.rvt

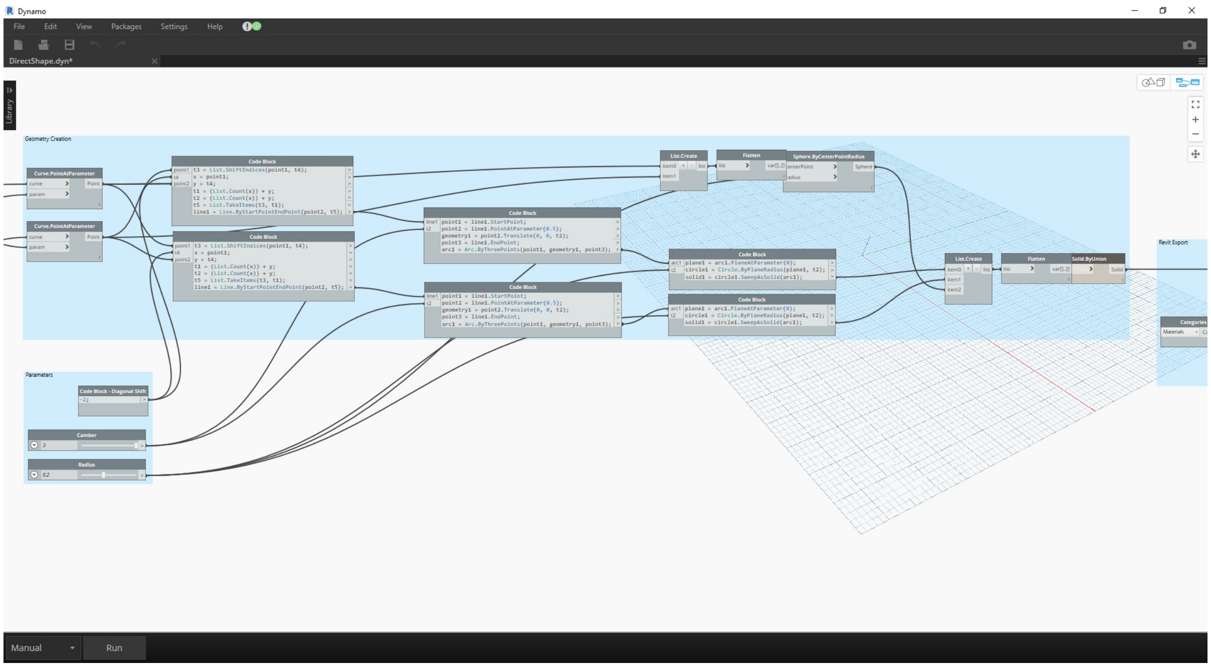


Begin by opening the sample file for this lesson - ARCH-DirectShape-BaseFile.rvt.

1. In the 3D view, we see our building mass from the previous lesson.
2. Along the edge of the atrium is one reference curve, we'll use this as a curve to reference in Dynamo.
3. Along the opposing edge of the atrium is another reference curve which we'll reference in Dynamo as well.



1. To reference our geometry in Dynamo, we'll use *Select Model Element* for each member in Revit. Select the mass in Revit and import the geometry into Dynamo by Using *Element.Faces* - the mass should now be visible in your Dynamo preview.
2. Import one reference curve into Dynamo by using *Select Model Element* and *CurveElement.Curve*.
3. Import the other reference curve into Dynamo by using *Select Model Element* and *CurveElement.Curve*.



```
t3 = List.ShiftIndices(point1, t4);
x = point1;
y = t4;
t1 = (List.Count(x)) + y;
t2 = (List.Count(x)) + y;
t5 = List.TakeItems(t3, t1);
line1 = Line.ByStartPointEndPoint(point2, t5);
```

```
t3 = List.ShiftIndices(point1, t4);
x = point1;
y = t4;
t1 = (List.Count(x)) + y;
t2 = (List.Count(x)) + y;
t5 = List.TakeItems(t3, t1);
line1 = Line.ByStartPointEndPoint(point2, t5);
```

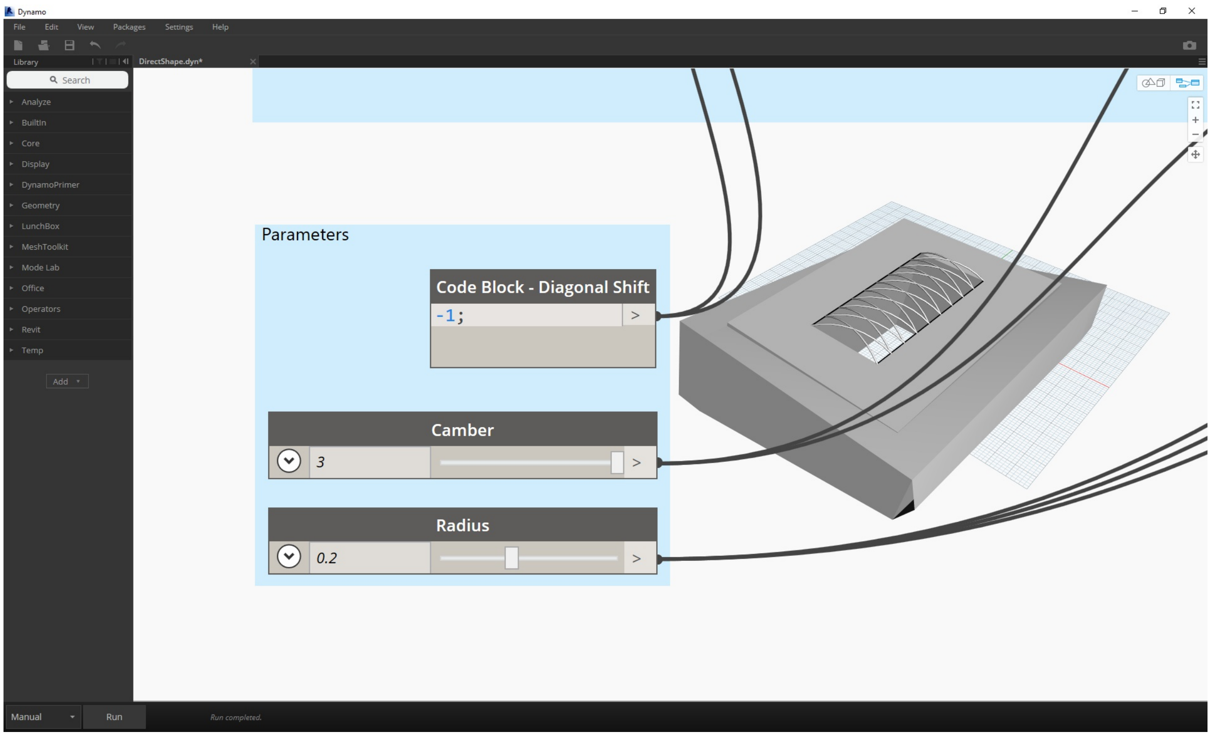
```
point1 = line1.StartPoint;
point2 = line1.PointAtParameter(0.5);
geometry1 = point2.Translate(0, 0, t2);
point3 = line1.EndPoint;
arc1 = Arc.ByThreePoints(point1, geometry1, point3);
```

```
point1 = line1.StartPoint;
point2 = line1.PointAtParameter(0.5);
geometry1 = point2.Translate(0, 0, t2);
point3 = line1.EndPoint;
arc1 = Arc.ByThreePoints(point1, geometry1, point3);
```

```
plane1 = arc1.PlaneAtParameter(0);
circle1 = Circle.ByPlaneRadius(plane1, t2);
solid1 = circle1.SweepAsSolid(arc1);
```

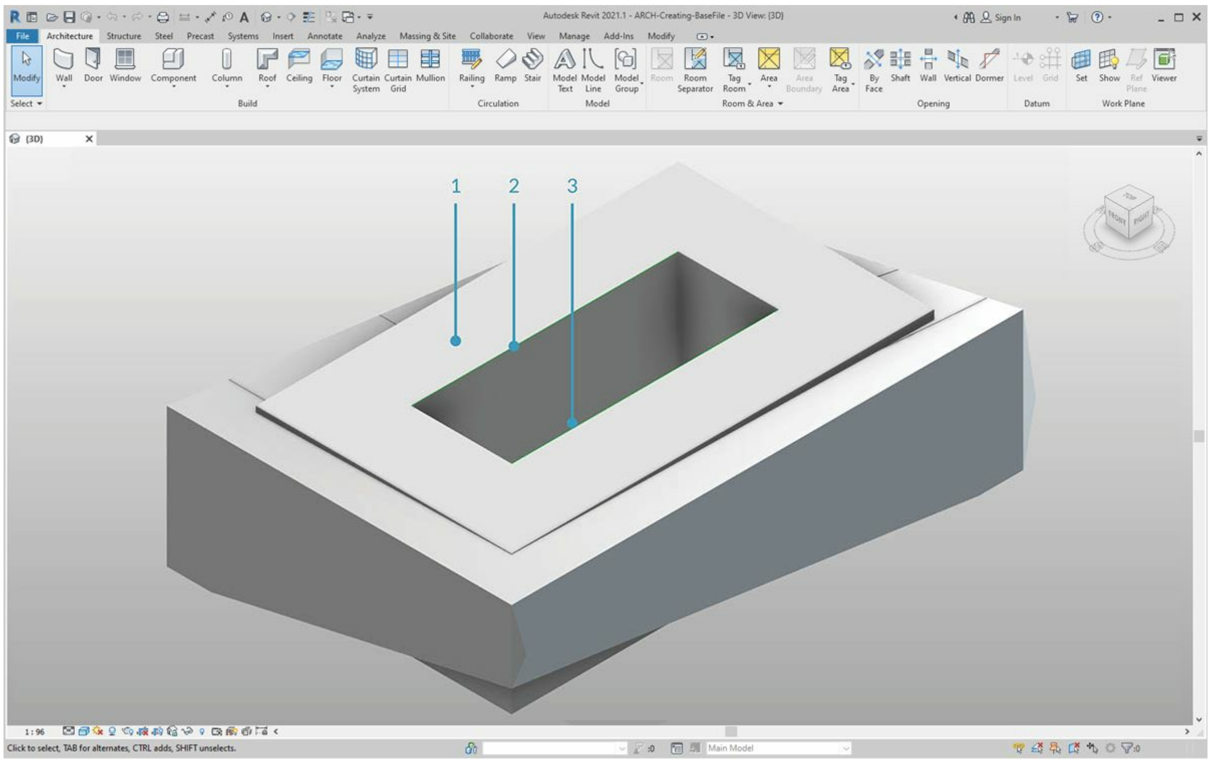
```
plane1 = arc1.PlaneAtParameter(0);
circle1 = Circle.ByPlaneRadius(plane1, t2);
solid1 = circle1.SweepAsSolid(arc1);
```

1. Zooming out and panning to the right in the sample graph, we see a large group of nodes - these are geometric operations which generate the trellis roof structure visible in the Dynamo preview. These nodes are generating using the *Node to Code* functionality as discussed in the code block section of the primer.
2. The structure is driven by three major parameters - Diagonal Shift, Camber, and Radius.

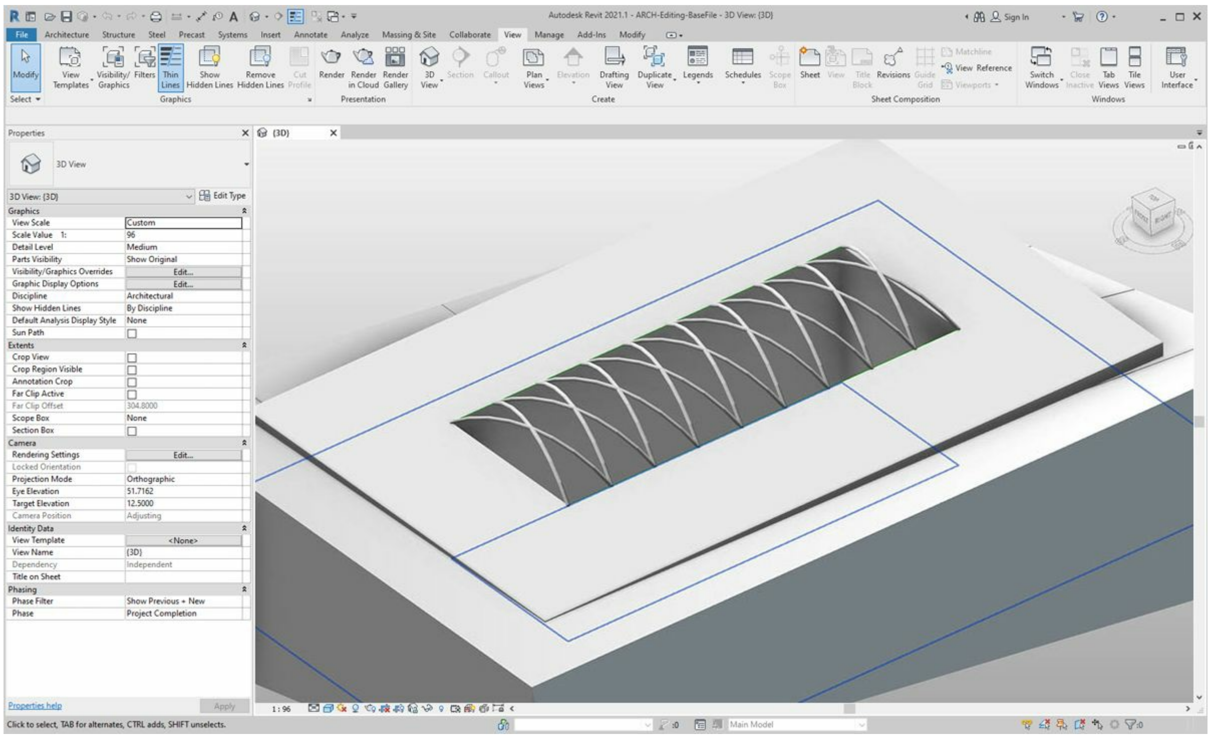


```
-1;
```

Zooming a close-up look of the parameters for this graph. We can flex these to get different geometry outputs.



1. Dropping the *DirectShape.ByGeometry* node onto the canvas, we see that it has four inputs: **geometry, category, material**, and **name**.
2. Geometry will be the solid created from the geometry creation portion of the graph
3. The category input is chosen using the dropdown *Categories* node. In this case we'll use "Structural Framing".
4. The material input is selected through the array of nodes above - although it can be more simply defined as "Default" in this case.



After running Dynamo, back in Revit, we have the imported geometry on the roof in our project. This is a structural framing element, rather than a generic model. The parametric link to Dynamo remains intact.

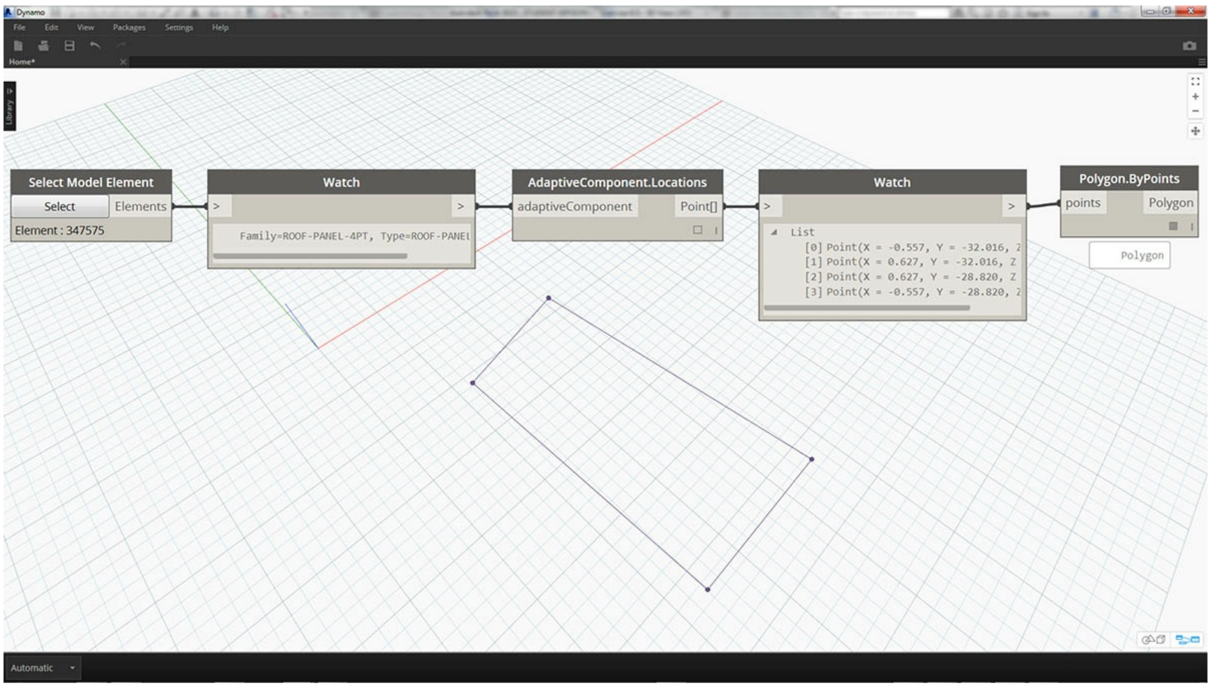
Customizing

Customizing

While we previously looked at editing a basic building mass, we want to dive deeper into the Dynamo/Revit link by editing a large number of elements in one go. Customizing on a large scale becomes more complex as data structures require more advanced list operations. However, the underlying principles behind their execution is fundamentally the same. Let's study some opportunities for analysis from a set of adaptive components.

Point Location

Suppose we've created a range of adaptive components and want to edit parameters based on their point locations. The points, for example, could drive a thickness parameter which is related to the area of the element. Or, they could drive an opacity parameter related to solar exposure throughout the year. Dynamo allows the connection of analysis to parameters in a few easy steps, and we'll explore a basic version in the exercise below.



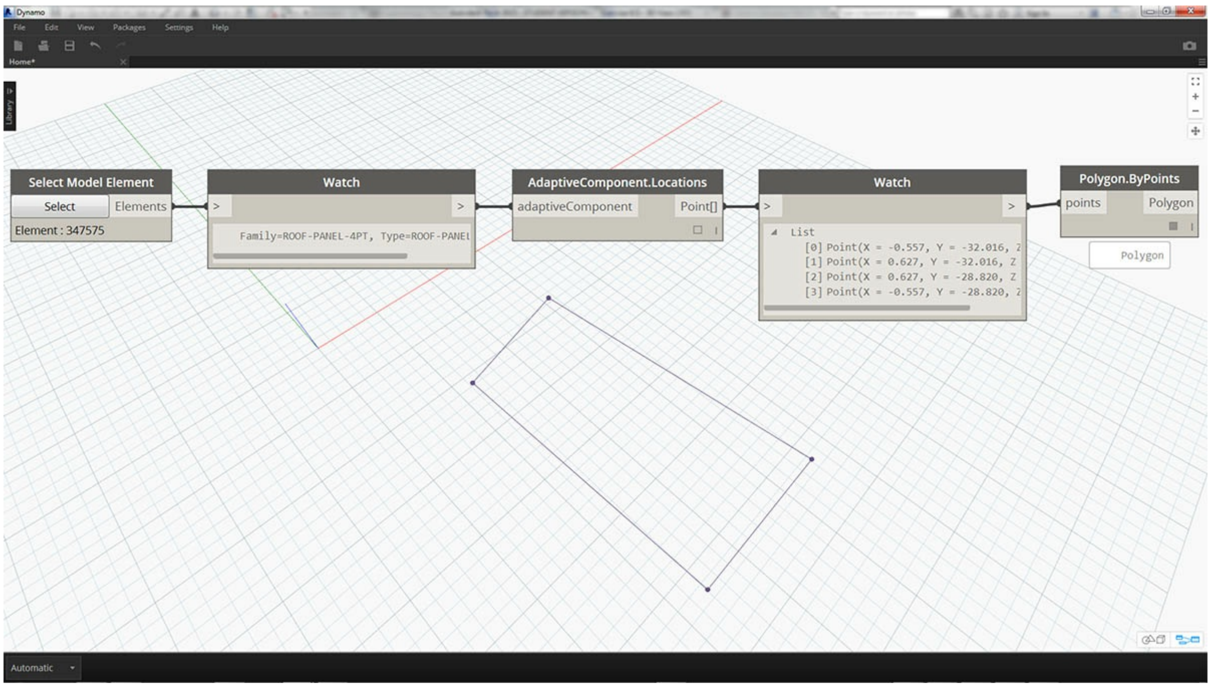
```
Family=ROOF-PANEL-4PT, Type=ROOF-PANEL
```

```
List
    [0] Point(X = -0.557, Y = -32.016, Z
    [1] Point(X = 0.627, Y = -32.016, Z
    [2] Point(X = 0.627, Y = -28.820, Z
    [3] Point(X = -0.557, Y = -28.820, Z
```

Query the adaptive points of a selected adaptive component by using the *AdaptiveComponent.Locations* node. This allows us to work with an abstracted version of a Revit element for analysis.

By extracting the point location of adaptive components, we can run a range of analysis for that element. A four-point adaptive component will allow you to study the deviation from plane for a given panel for example.

Solar Orientation Analysis



Select Model Element
Select
Elements
Element : 347575
Watch
Family=ROOF-PANEL-4PT, Type=ROOF-PANEL
AdaptiveComponent.Locations
adaptiveComponent
Point[]
Watch
List
[0] Point(X = -0.557, Y = -32.016, Z
[1] Point(X = 0.627, Y = -32.016, Z
[2] Point(X = 0.627, Y = -28.820, Z
[3] Point(X = -0.557, Y = -28.820, Z
Polygon.ByPoints
points
Polygon
Automatic

Use remapping to map a set of a data into a parameter range. This is fundamental tool used in a parametric model, and we'll demonstrate it in the exercise below.

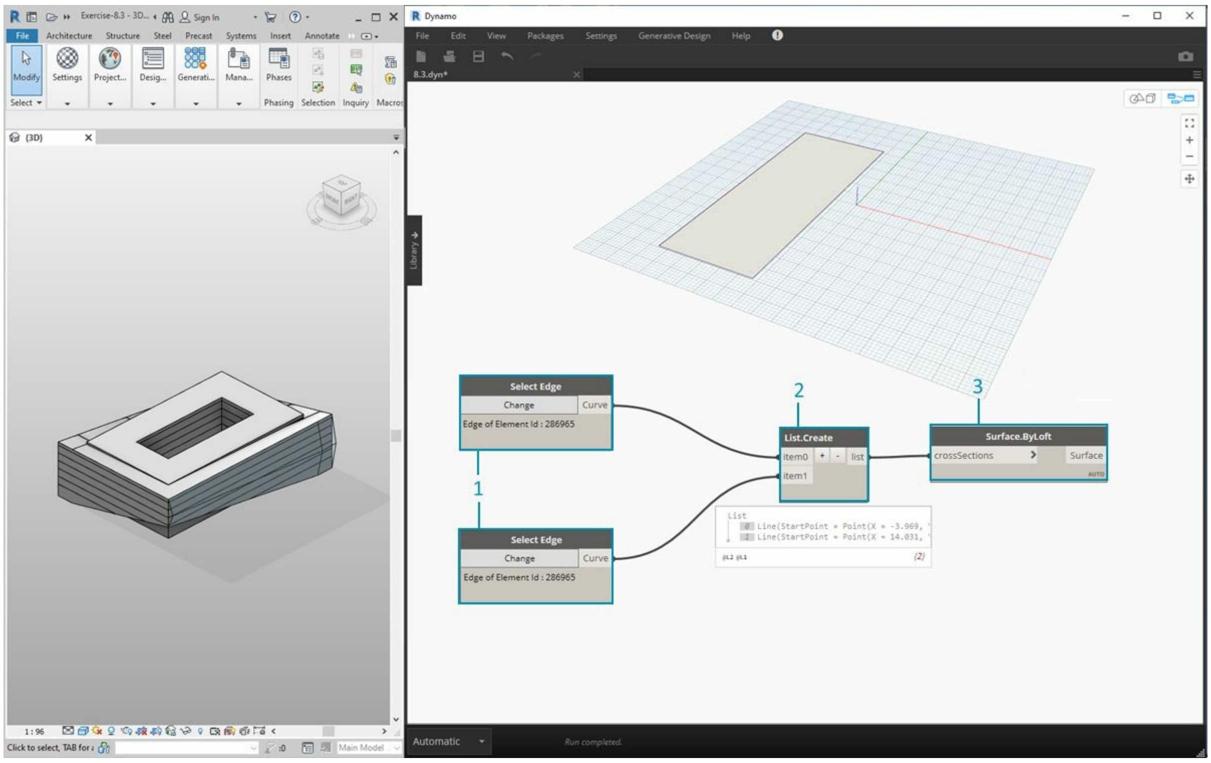
Using Dynamo, the point locations of adaptive components can be used to create a best-fit plane each element. We can also query the sun position in the Revit file and study the plane's relative orientation to the sun in comparison to other adaptive components. Let's set that up in the exercise below by creating an algorithmic roofscape.

Exercise

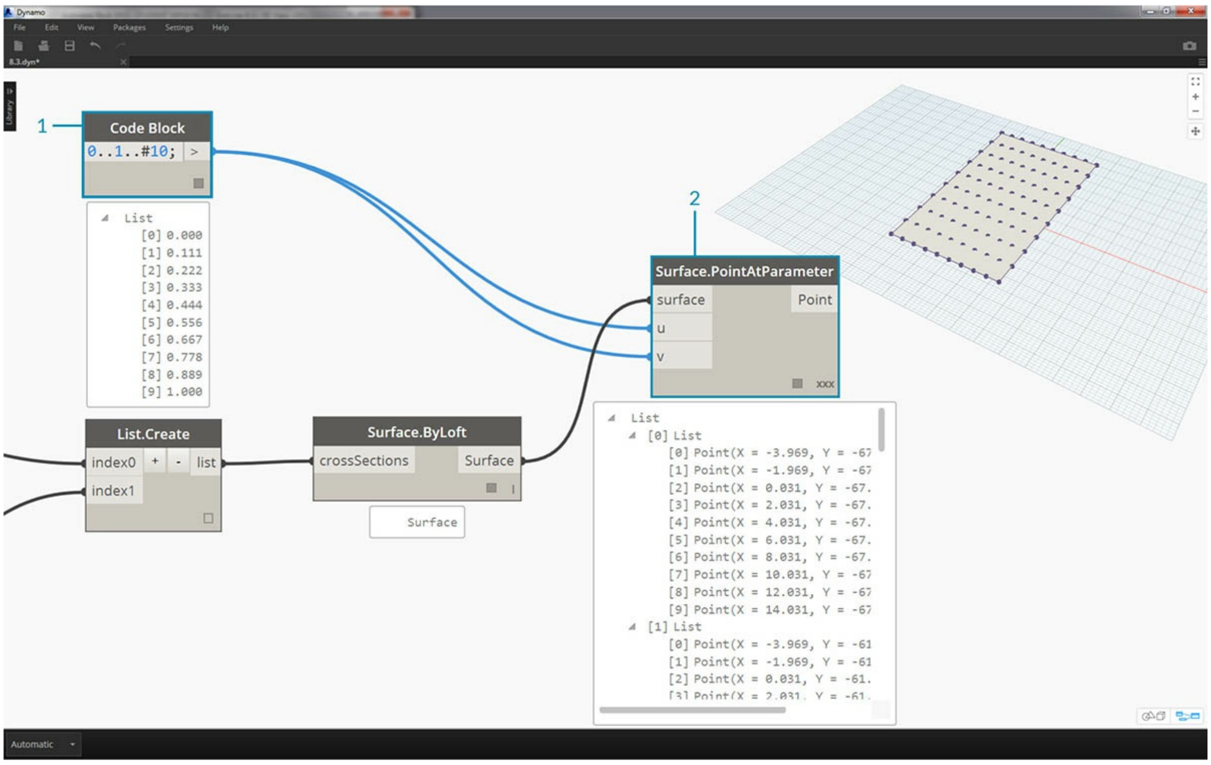
Download the example files that accompanies this exercise (Right click and "Save Link As..."). A full list of example files can be found in the Appendix.

1. Customizing.dyn
2. ARCH-Customizing-BaseFile.rvt

This exercise will expand on the techniques demonstrated in the previous section. In this case, we are defining a parametric surface from Revit elements, instantiating four-point adaptive components and then editing them based on orientation to the sun.



1. Beginning by selecting two edges with the *"Select Edge"* node. The two edges are the long spans of the atrium.
2. Combine the two edges into one list with the *List.Create* node.
3. Create a surface between the two edges with a *Surface.ByLoft*.



```
0..1..#10;
```

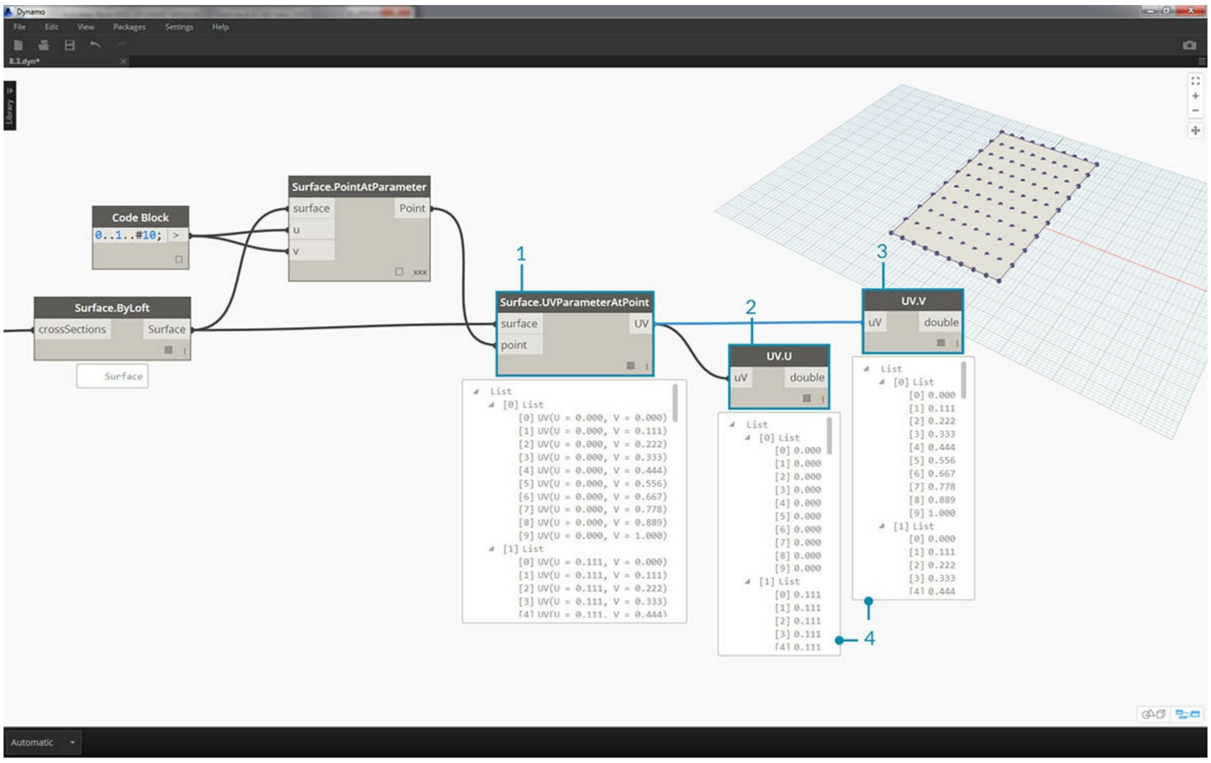
```
List
  [0] 0.000
  [1] 0.111
  [2] 0.222
  [3] 0.333
  [4] 0.444
  [5] 0.556
  [6] 0.667
  [7] 0.778
  [8] 0.889
  [9] 1.000
```

```
Surface
```

```
List
  [0] List
    [0] Point(X = -3.969, Y = -67
    [1] Point(X = -1.969, Y = -67
    [2] Point(X = 0.031, Y = -67.
    [3] Point(X = 2.031, Y = -67.
    [4] Point(X = 4.031, Y = -67.
    [5] Point(X = 6.031, Y = -67.
    [6] Point(X = 8.031, Y = -67.
    [7] Point(X = 10.031, Y = -67
    [8] Point(X = 12.031, Y = -67
    [9] Point(X = 14.031, Y = -67
  [1] List
    [0] Point(X = -3.969, Y = -61
    [1] Point(X = -1.969, Y = -61
    [2] Point(X = 0.031, Y = -61.
    [3] Point(X = 2.031, Y = -61.
```

1. Using *code block*, define a range from 0 to 1 with 10 evenly spaced values: `0..1..#10;`
2. Plug the *code block* into the *u* and *v* inputs of a *Surface.PointAtParameter* node, and plug the *Surface.ByLoft* node into the *surface* input. Right click the node and change the *lacing* to *Cross Product*. This will give a grid of points on the surface.

This grid of points serves as the control points for a parametrically defined surface. We want to extract the u and v positions of each one of these points so that we can plug them into a parametric formula and keep the same data structure. We can do this by querying the parameter locations of the points we just created.



```
Surface
```

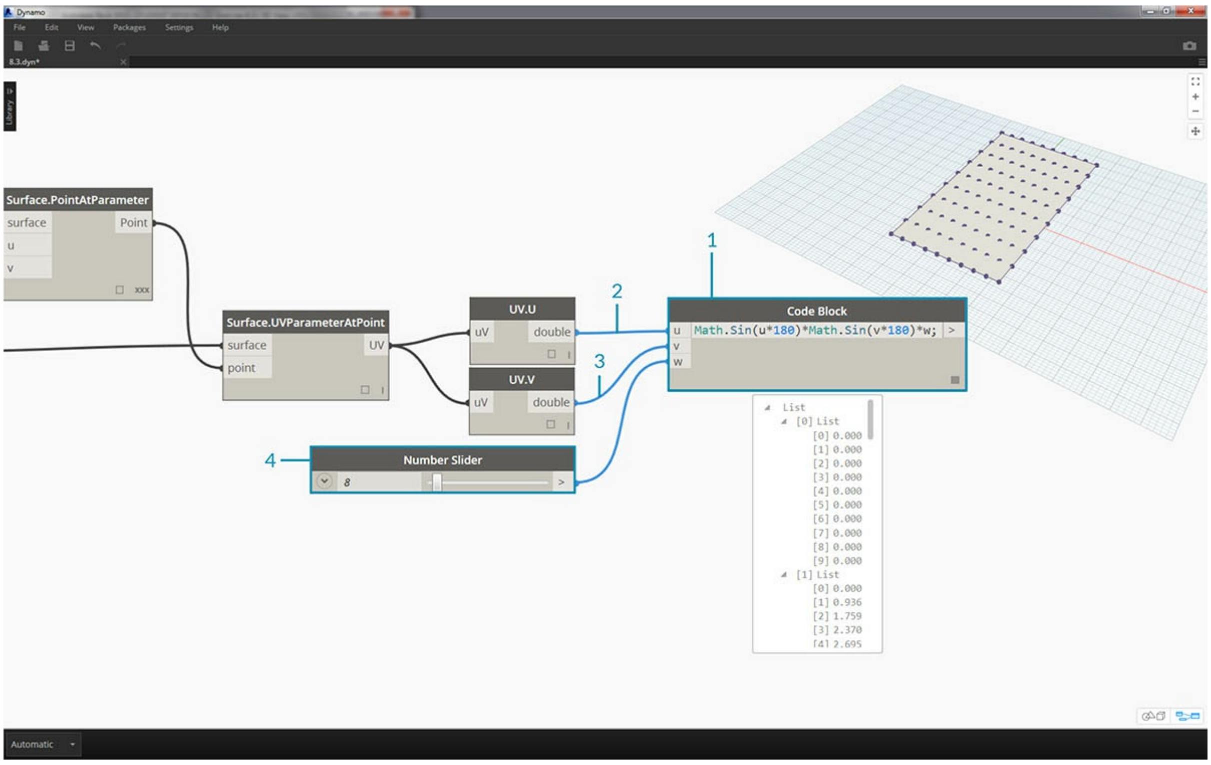
```
List
  [0] List
    [0] UV(U = 0.000, V = 0.000)
    [1] UV(U = 0.000, V = 0.111)
    [2] UV(U = 0.000, V = 0.222)
    [3] UV(U = 0.000, V = 0.333)
    [4] UV(U = 0.000, V = 0.444)
    [5] UV(U = 0.000, V = 0.556)
    [6] UV(U = 0.000, V = 0.667)
    [7] UV(U = 0.000, V = 0.778)
    [8] UV(U = 0.000, V = 0.889)
    [9] UV(U = 0.000, V = 1.000)
  [1] List
    [0] UV(U = 0.111, V = 0.000)
    [1] UV(U = 0.111, V = 0.111)
    [2] UV(U = 0.111, V = 0.222)
    [3] UV(U = 0.111, V = 0.333)
    [4] UV(U = 0.111, V = 0.444)
```

```
List
  [0] List
    [0] 0.000
    [1] 0.000
    [2] 0.000
    [3] 0.000
    [4] 0.000
    [5] 0.000
    [6] 0.000
    [7] 0.000
    [8] 0.000
    [9] 0.000
  [1] List
    [0] 0.111
    [1] 0.111
    [2] 0.111
    [3] 0.111
    [4] 0.111
```

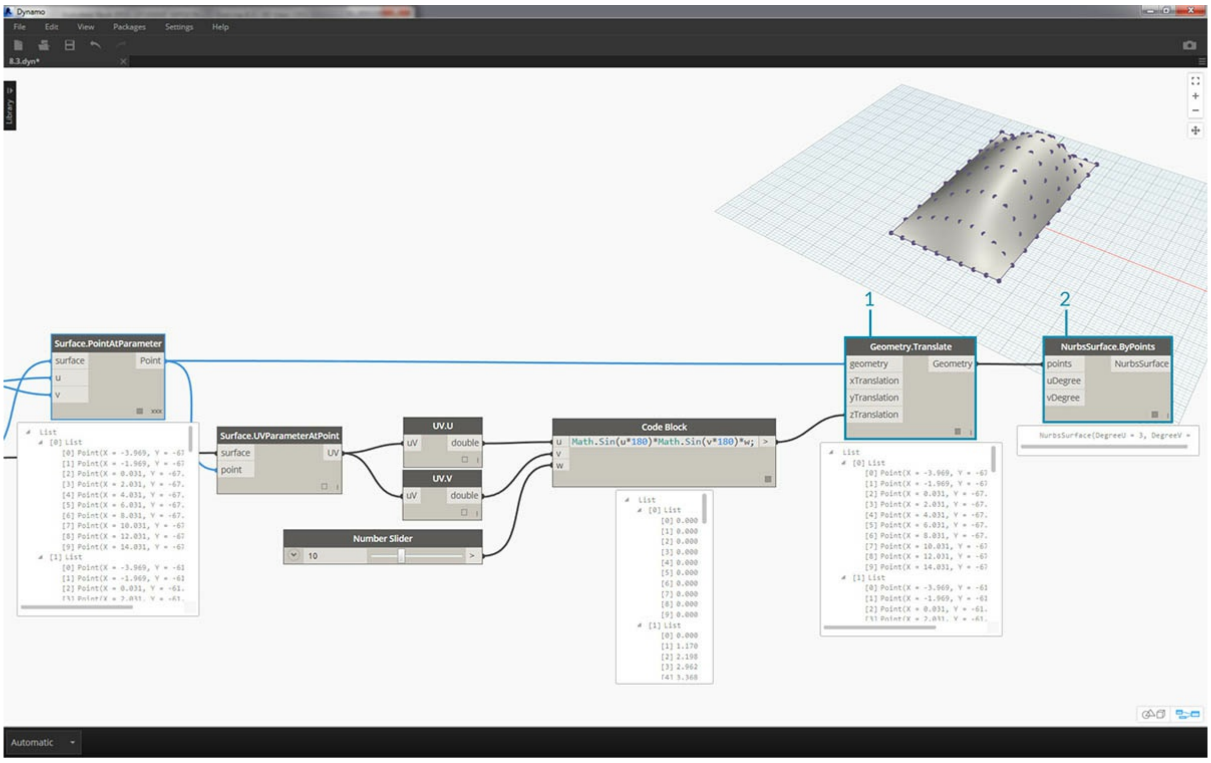
```
List
  [0] List
    [0] 0.000
    [1] 0.111
    [2] 0.222
    [3] 0.333
    [4] 0.444
    [5] 0.556
    [6] 0.667
    [7] 0.778
    [8] 0.889
    [9] 1.000
  [1] List
    [0] 0.000
    [1] 0.111
    [2] 0.222
    [3] 0.333
    [4] 0.444
```

1. Add a *Surface.ParameterAtPoint* node to the canvas, connect the inputs as shown above.
2. Query the *u* values of these parameters with the *UV.U* node.

3. Query the *v* values of these parameters with the *UV.V* node.
4. The outputs show the corresponding *u* and *v* values for every point of the surface. We now have a range from *0* to *1* for each value, in the proper data structure, so we're ready to apply a parametric algorithm.


Surface.PointAtParameter
Surface.UVParameterAtPoint
UV.U
UV.V
Code Block
Math.Sin(u*180)*Math.Sin(v*180)*w;
Number Slider
1
2
3
4

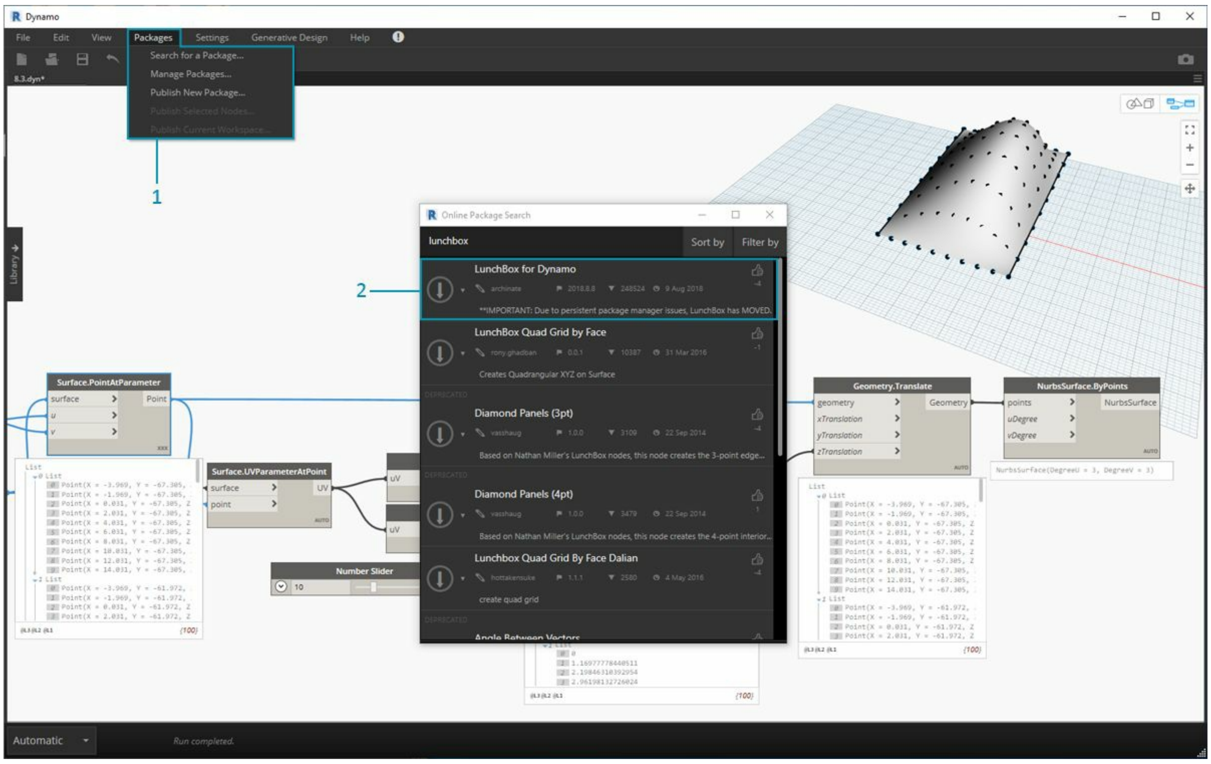
1. Add a *code block* to the canvas and enter the code: `Math.Sin(u*180)*Math.Sin(v*180)*w;` This is a parametric function which creates a sine mound from a flat surface.
2. The *u* input connects to *UV.U*.
3. The *v* input connects to *UV.V*.
4. The *w* input represents the *amplitude* of the shape, so we attach a *number slider* to it.



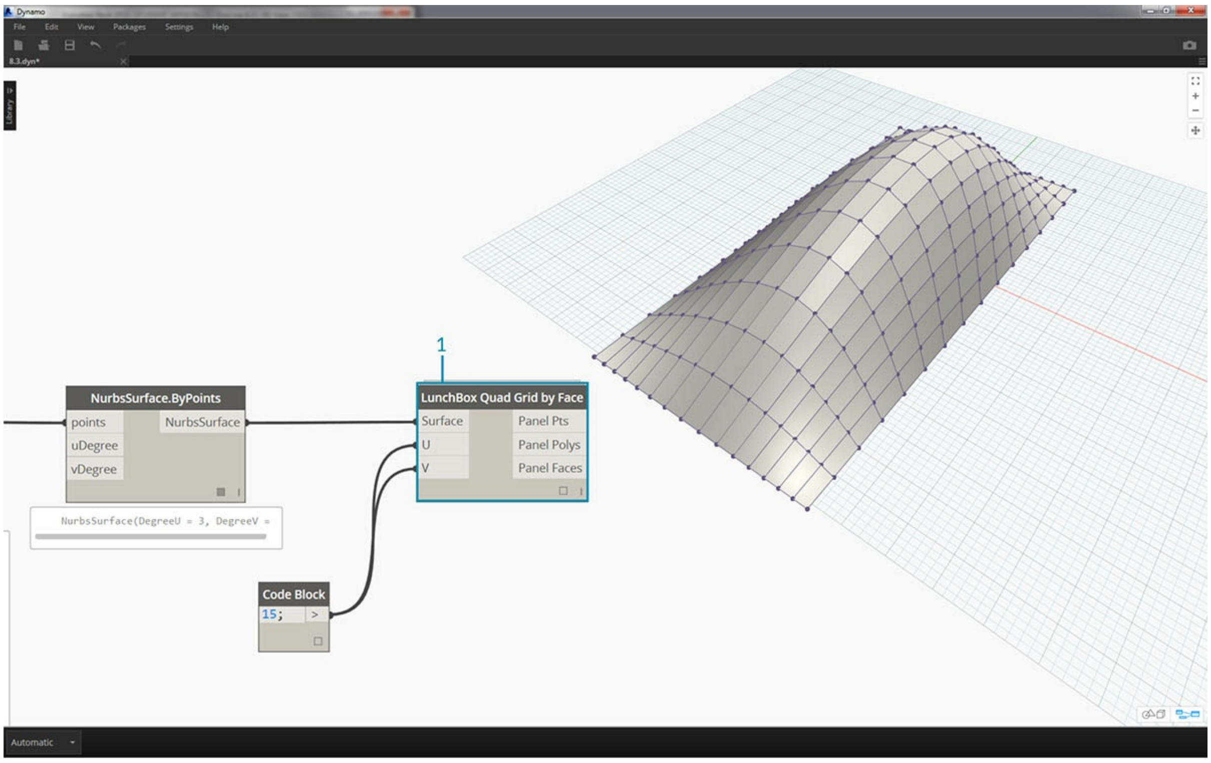
1. Now, we have a list of values as defined by the algorithm. Let's use this list of values to move the points up in the *+Z* direction. Using *Geometry.Translate*, plug the *code block* into *zTranslation* and the *Surface.PointAtParameter* into the *geometry* input. You should see the new points displayed in the Dynamo preview.
2. Finally, we create a surface with the *NurbsSurface.ByPoints* node, plugging the node from the previous step into the points input. We have

ourselves a parametric surface. Feel free to drag the slider to watch the mound shrink and grow.

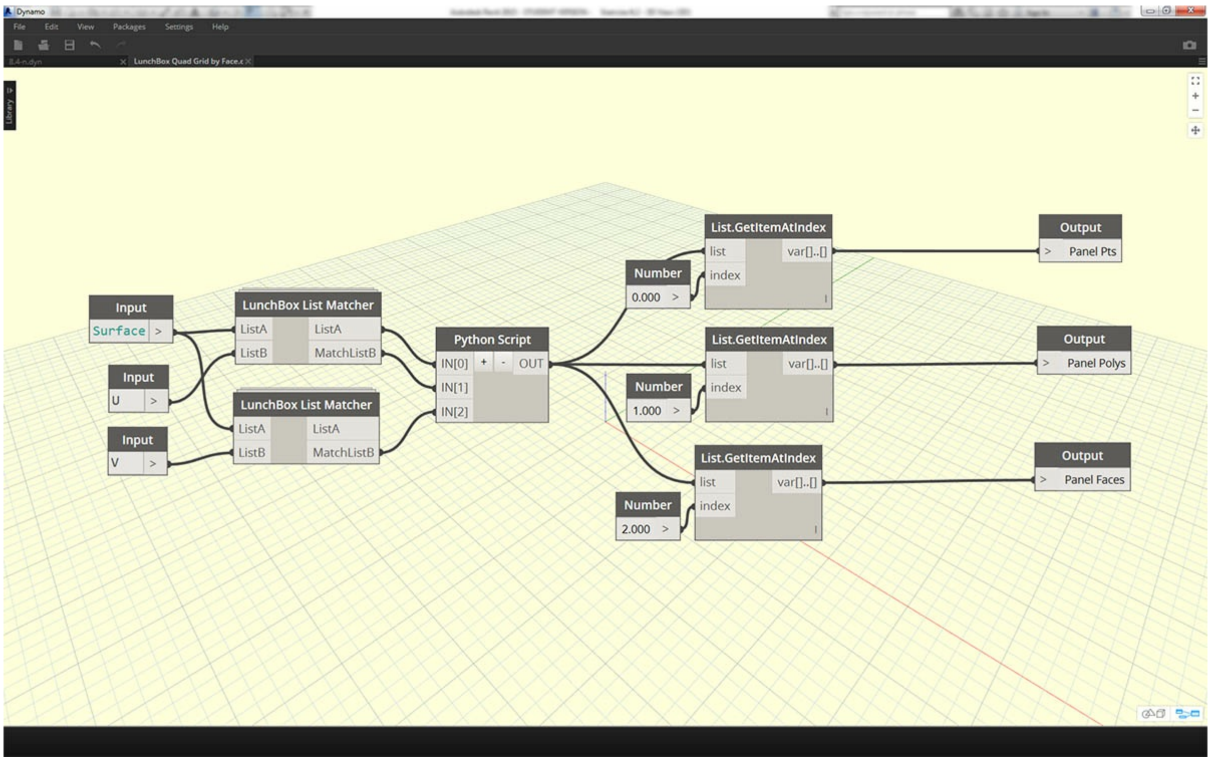
With the parametric surface, we want to define a way to panelize it in order to array four-point adaptive components. Dynamo does not have out-of-the-box functionality for surface panelization, so we can look to the community for helpful Dynamo packages.



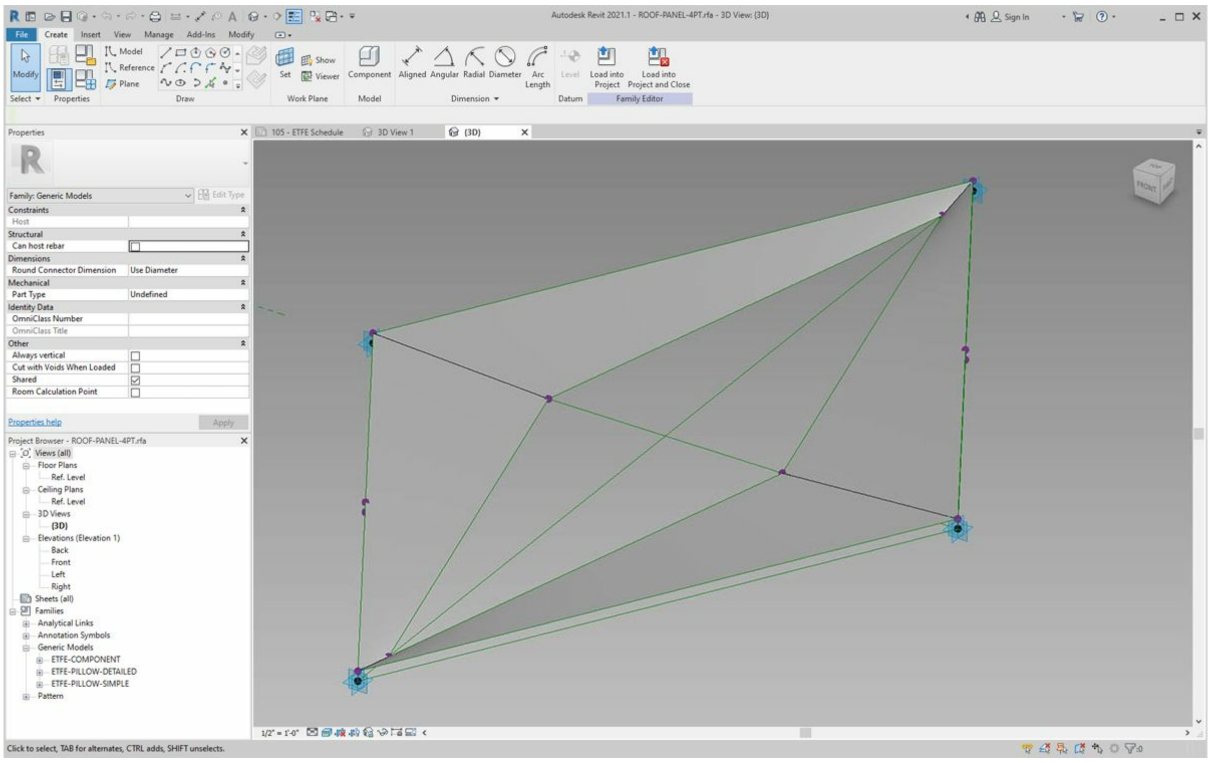
1. Go to *Packages>Search for a Package...*
2. Search for *"LunchBox"* and download *"LunchBox for Dynamo"*. This is a really helpful set of tools for geometry operations such as this.



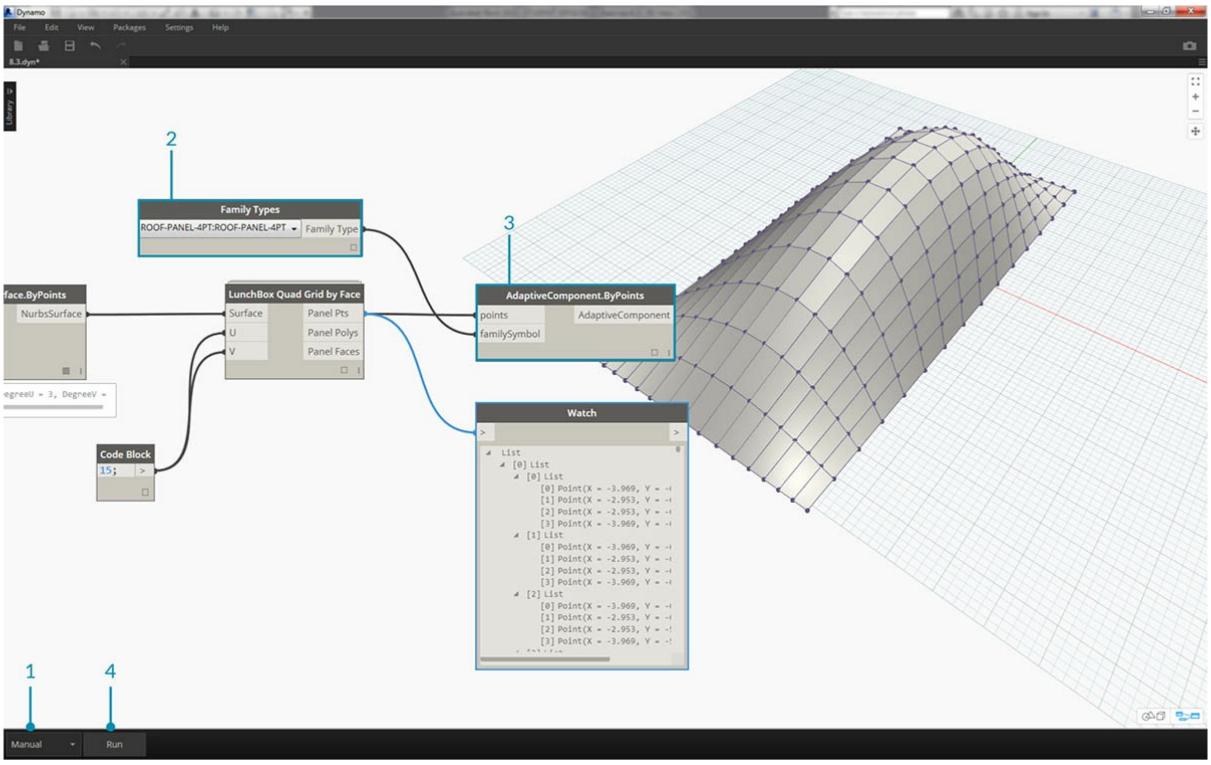
1. After downloading, you now have full access to the LunchBox suite. Search for *"Quad Grid"* and select *"LunchBox Quad Grid By Face"*. Plug the parametric surface into the *surface* input and set the *U* and *V* divisions to *15*. You should see a quad-paneled surface in your Dynamo preview.



If you're curious about its setup, you can double click on the *Lunch Box* node and see how it's made.

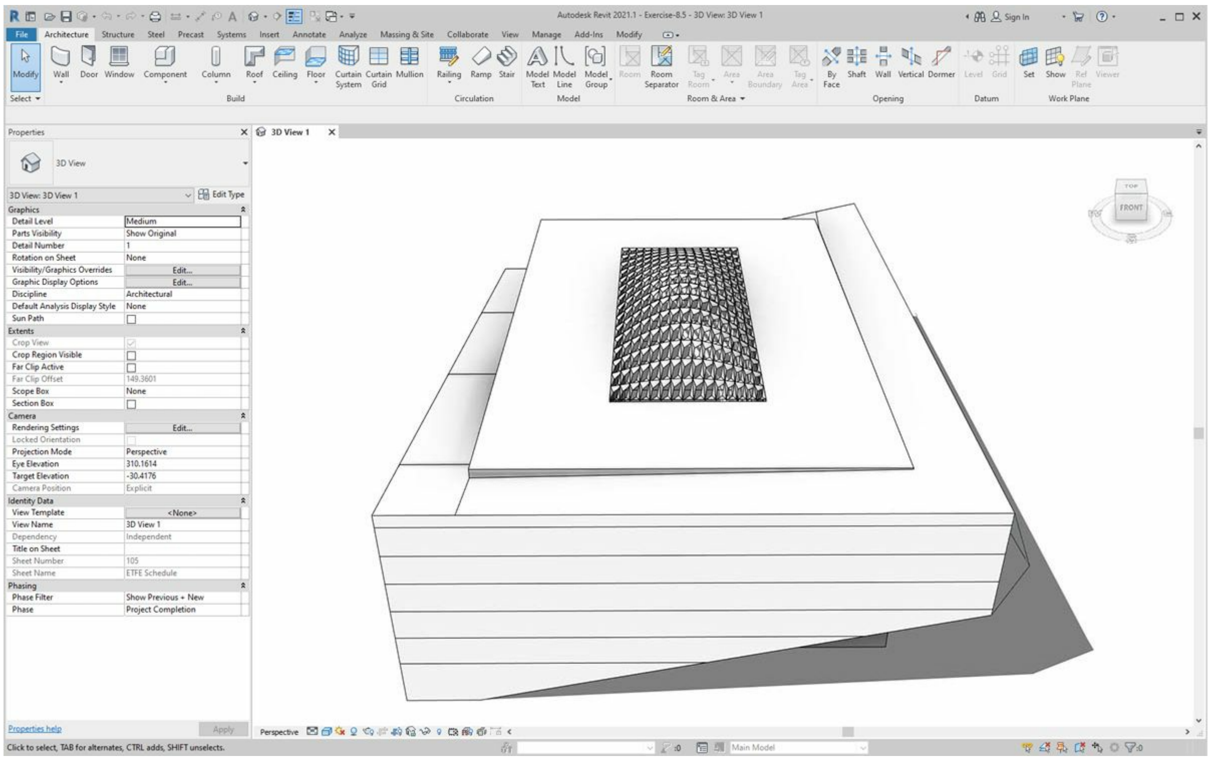

Autodesk Revit 2021.1 - ROOF-PANEL-4PT.rfa - 3D View: {3D}
Properties
Family: Generic Models
Project Browser - ROOF-PANEL-4PT.rfa
ETFE-COMPONENT
ETFE-PILLOW-DETAILED
ETFE-PILLOW-SIMPLE

Back in Revit, let's take a quick look at the adaptive component we're using here. No need to follow along, but this is the roof panel we're going to instantiate. It is a four-point adaptive component which is a crude representation of an ETFE system. The aperture of the center void is on a parameter called *"ApertureRatio"*.

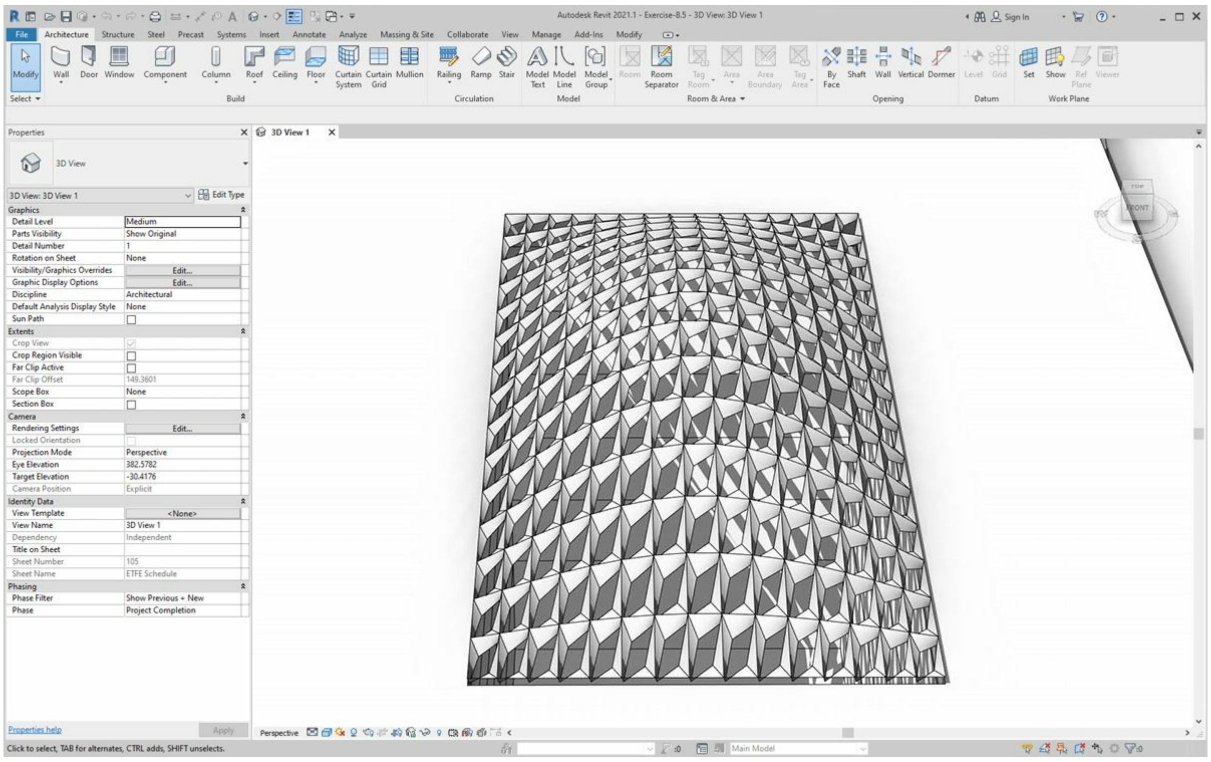


1. We're about to instantiate a lot of geometry in Revit, so make sure to turn the Dynamo solver to *"Manual"*.
2. Add a *Family Types* node to the canvas and select *"ROOF-PANEL-4PT"*.
3. Add an *AdaptiveComponent.ByPoints* node to the canvas, connect *Panel Pts* from the *"LunchBox Quad Grid by Face"* output into the *points* input. Connect the *Family Types* node to the *familySymbol* input.
4. Hit *Run*. Revit will have to *think* for a bit while the geometry is being created. If it takes too long, reduce the *code block's '15'* to a lower number. This will reduce the number of panels on the roof.

Note: If Dynamo is taking a long time to calculate nodes, you may want to use the "freeze" node functionality in order to pause the execution of Revit operations while you develop your graph. For more information on freezing nodes, check out the "Freezing" section in the solids chapter.



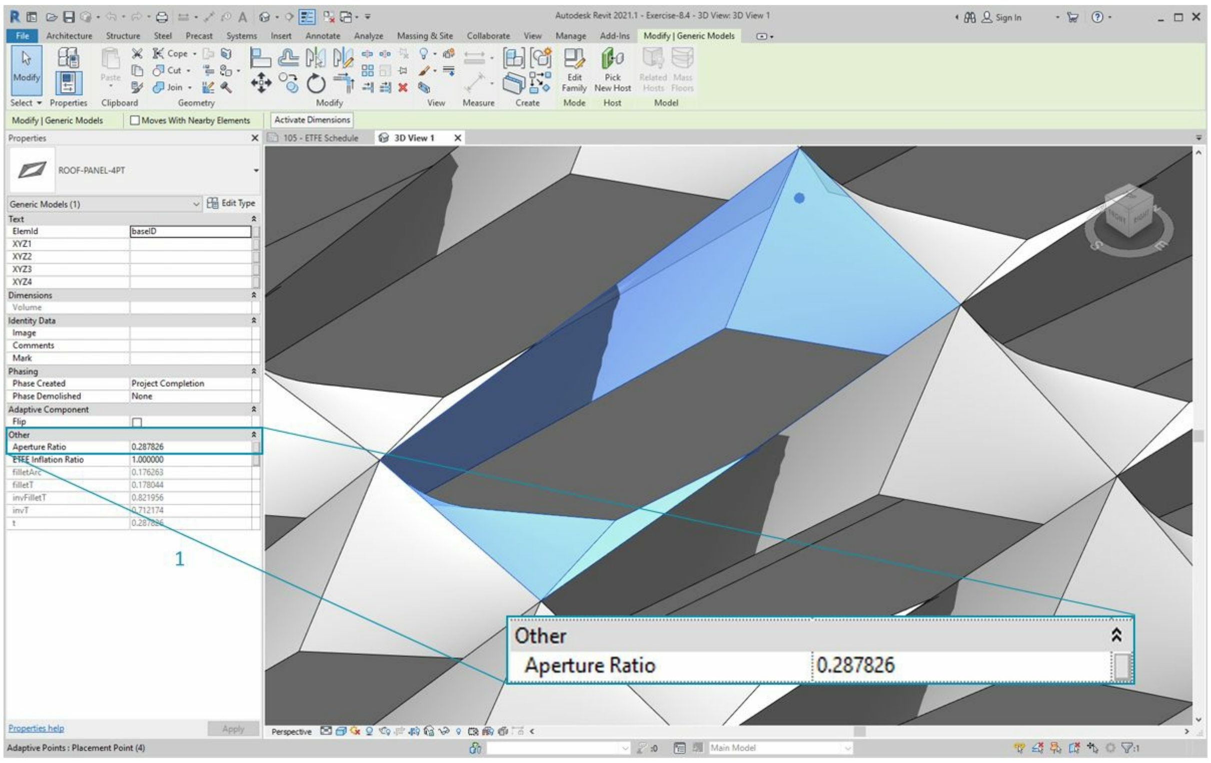
Back in Revit, we have the array of panels on the roof.



Properties	
3D View	
3D View: 3D View 1	Edit Type
Graphics	
Detail Level	Medium
Parts Visibility	Show Original
Detail Number	1
Rotation on Sheet	None
Visibility/Graphics Overrides	Edit...
Graphic Display Options	Edit...
Discipline	Architectural
Default Analysis Display Style	None
Sun Path	☐
Extents	
Crop View	✓
Crop Region Visible	☐
Far Clip Active	☐
Far Clip Offset	149.3601
Scope Box	None
Section Box	☐
Camera	
Rendering Settings	Edit...
Locked Orientation	☐
Projection Mode	Perspective
Eye Elevation	382.5782
Target Elevation	-30.4176
Camera Position	Explicit
Identity Data	
View Template	<None>
View Name	3D View 1
Dependency	Independent
Title on Sheet	
Sheet Number	105
Sheet Name	ETFE Schedule
Phasing	
Phase Filter	Show Previous + New
Phase	Project Completion

Zooming in, we can get a closer look at their surface qualities.

Analysis

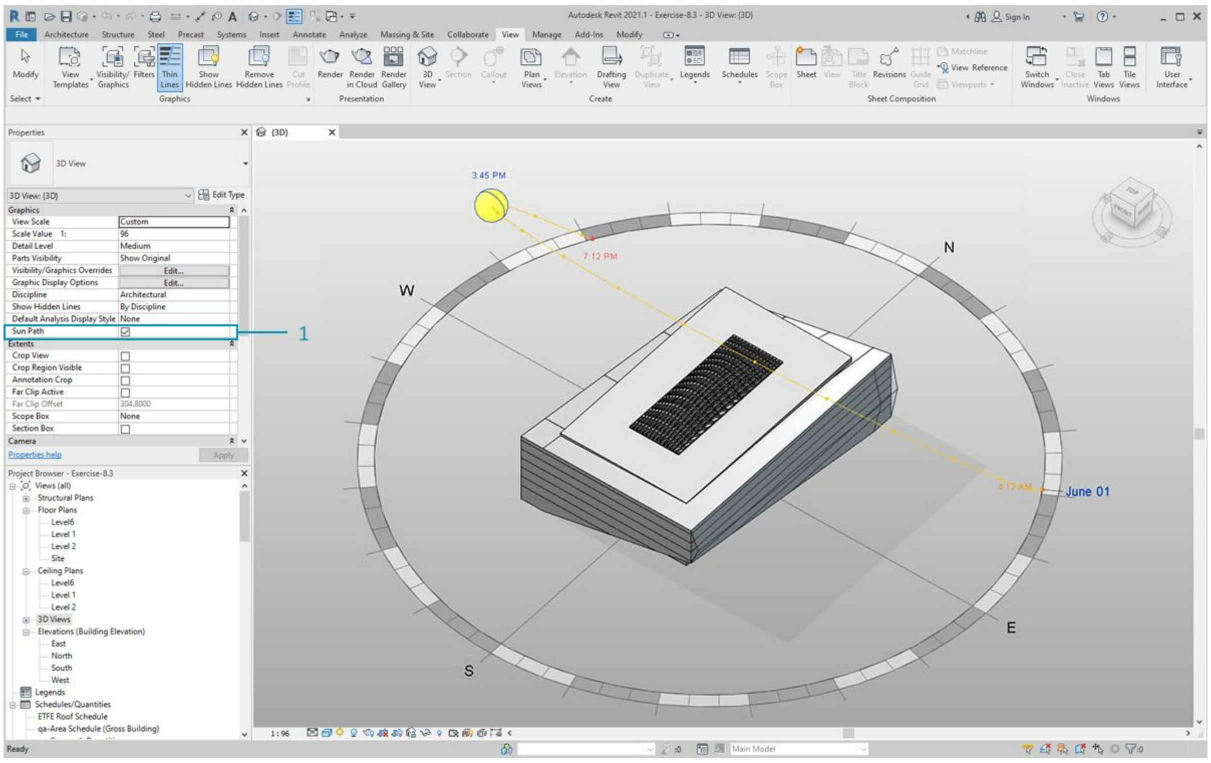


Properties	
ROOF-PANEL-4PT	
Generic Models (1)	Edit Type
Text	
ElemId	baseID
XYZ1	
XYZ2	
XYZ3	
XYZ4	
Dimensions	
Volume	
Identity Data	
Image	
Comments	
Mark	
Phasing	
Phase Created	Project Completion
Phase Demolished	None
Adaptive Component	
Flip	☐
Other	
Aperture Ratio	0.287826
ETFE Inflation Ratio	1.000000
filletArc	0.176263
filletT	0.178044
invFilletT	0.821956
invT	0.712174
t	0.287826

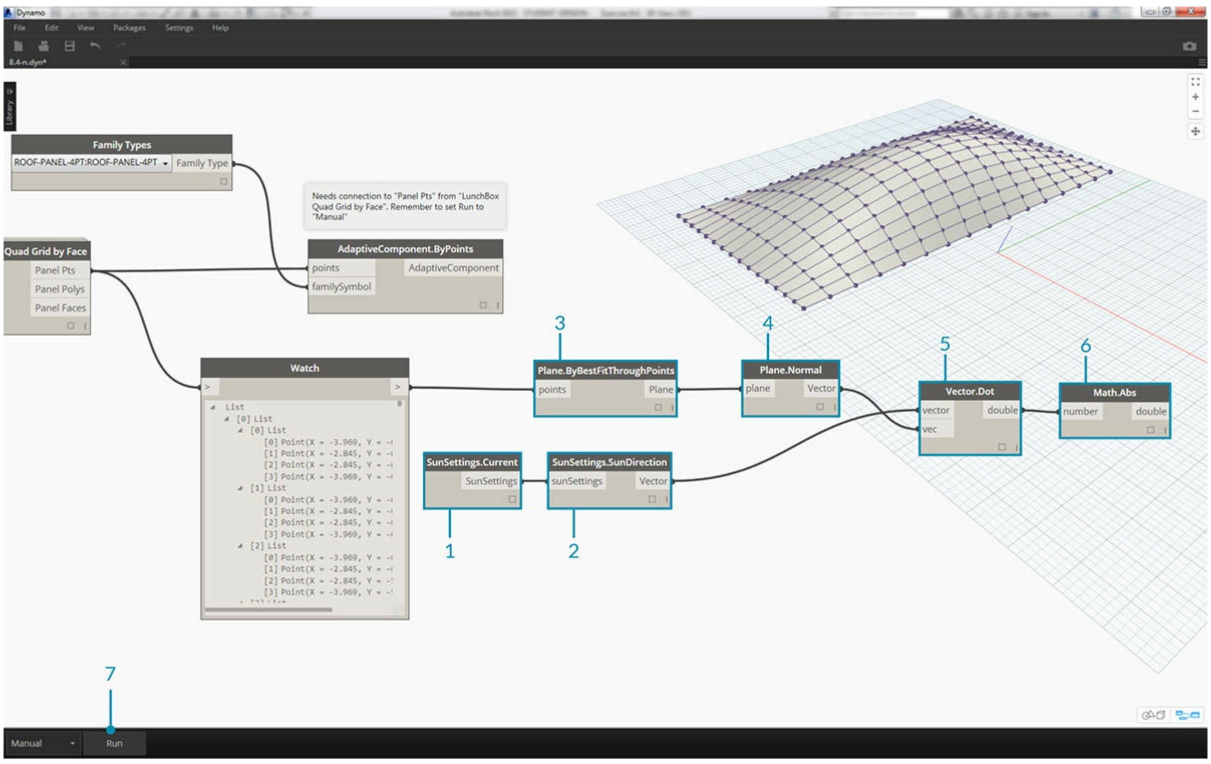
Other

Aperture Ratio 0.287826

1. Continuing from the previous step, let's go further and drive the aperture of each panel based on its exposure to the sun. Zooming into Revit and select one panel, we see in the properties bar that there is a parameter called *"Aperture Ratio"*. The family is setup so that the aperture ranges, roughly, from *0.05* to *0.45*.

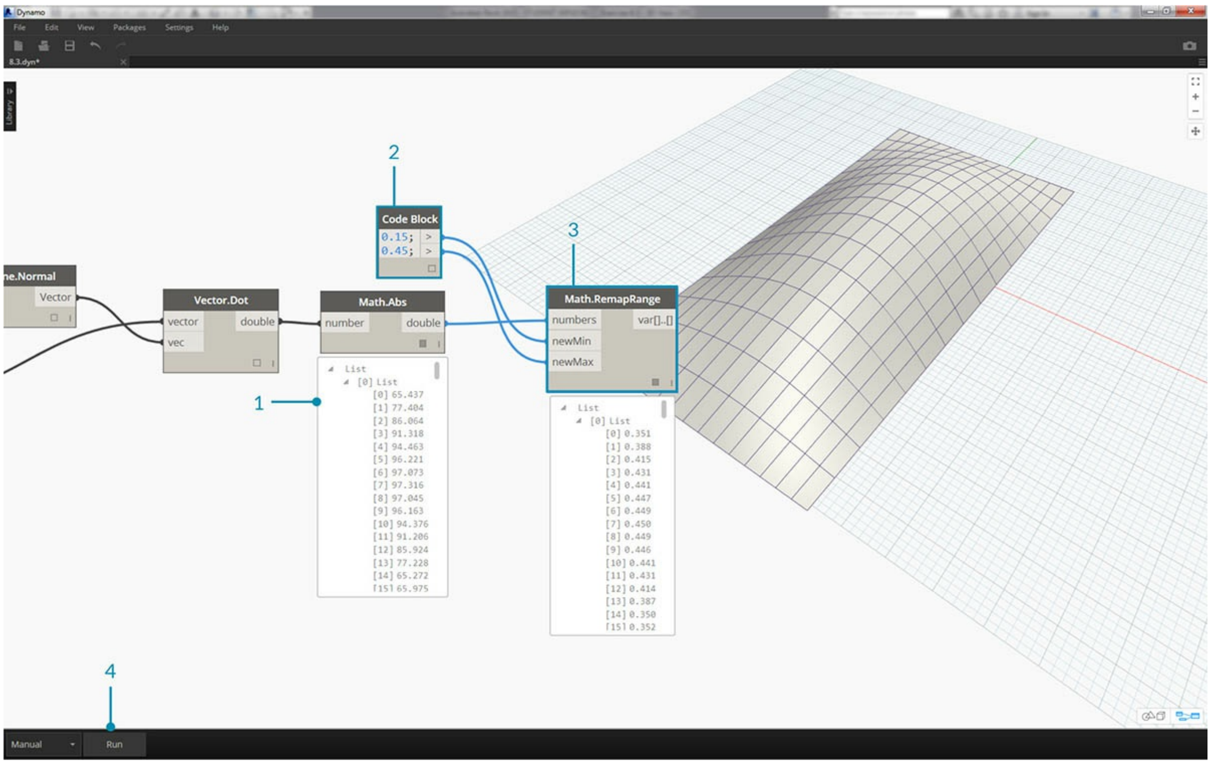


1. If we turn on the solar path, we can see the current sun location in Revit.



```
List
  [0] List
    [0] List
      [0] Point(X = -3.969, Y = -
      [1] Point(X = -2.845, Y = -
      [2] Point(X = -2.845, Y = -
      [3] Point(X = -3.969, Y = -
    [1] List
      [0] Point(X = -3.969, Y = -
      [1] Point(X = -2.845, Y = -
      [2] Point(X = -2.845, Y = -
      [3] Point(X = -3.969, Y = -
    [2] List
      [0] Point(X = -3.969, Y = -
      [1] Point(X = -2.845, Y = -
      [2] Point(X = -2.845, Y = -
      [3] Point(X = -3.969, Y = -
```

1. We can reference this sun location using the *SunSettings.Current* node.
2. Plug the Sun settings into *Sunsetting.SunDirection* to get the solar vector.
3. From the *Panel Pts* used to create the adaptive components, use *Plane.ByBestFitThroughPoints* to approximate a plane for the component.
4. Query the *normal* of this plane.
5. Use the *dot product* to calculate solar orientation. The dot product is a formula which determines how parallel or anti-parallel two vectors may be. So we're taking the plane normal of each adaptive component and comparing it to the solar vector to roughly simulate solar orientation.
6. Take the *absolute value* of the result. This ensures that the dot product is accurate if the plane normal is facing the reverse direction.
7. Hit *Run*.

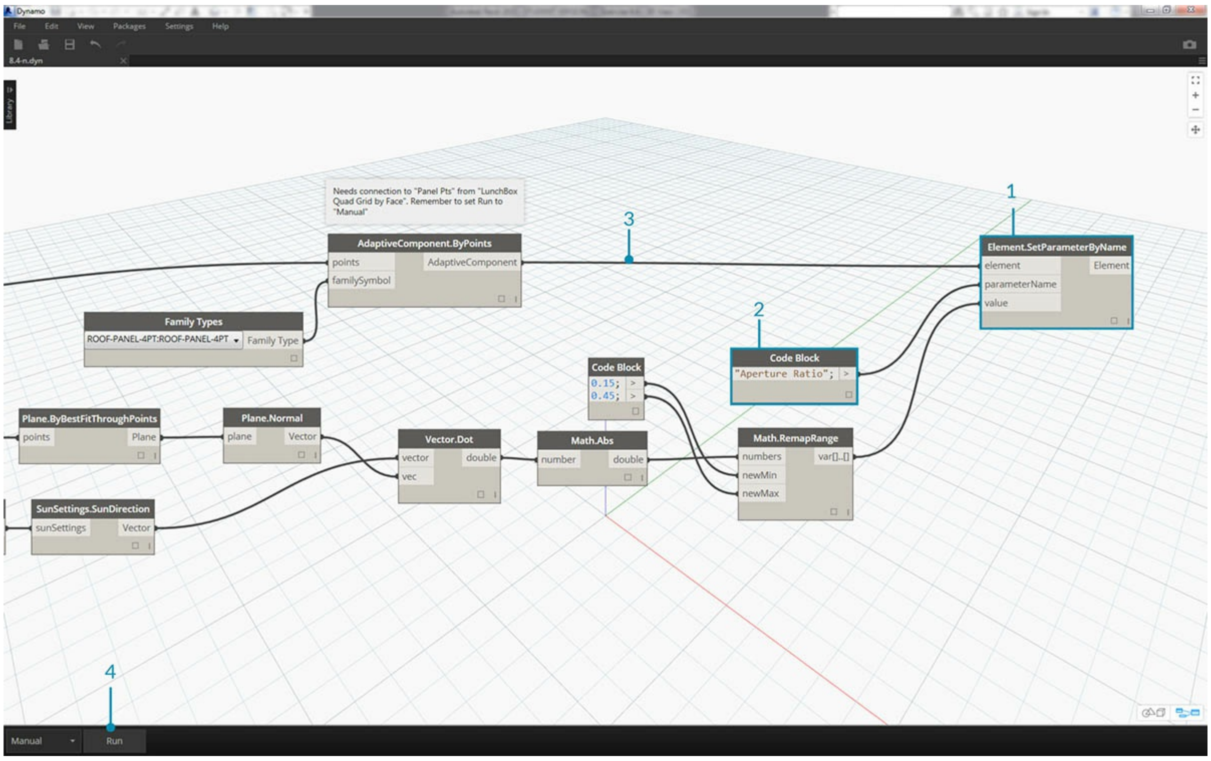


```
0.15;
0.45;
```

```
List
  [0] List
    [0] 65.437
    [1] 77.404
    [2] 86.064
    [3] 91.318
    [4] 94.463
    [5] 96.221
    [6] 97.073
    [7] 97.316
    [8] 97.045
    [9] 96.163
    [10] 94.376
    [11] 91.206
    [12] 85.924
    [13] 77.228
    [14] 65.272
    [15] 65.975
```

```
List
  [0] List
    [0] 0.351
    [1] 0.388
    [2] 0.415
    [3] 0.431
    [4] 0.441
    [5] 0.447
    [6] 0.449
    [7] 0.450
    [8] 0.449
    [9] 0.446
    [10] 0.441
    [11] 0.431
    [12] 0.414
    [13] 0.387
    [14] 0.350
    [15] 0.352
```

1. Looking at the *dot product*, we have a wide range of numbers. We want to use their relative distribution, but we need to condense the numbers into the appropriate range of the *"Aperture Ratio"* parameter we plan to edit.
2. The *Math.RemapRange* is a great tool for this. It takes an input list and remaps its bounds into two target values.
3. Define the target values as *0.15* and *0.45* in a *code block*.
4. Hit *Run*.

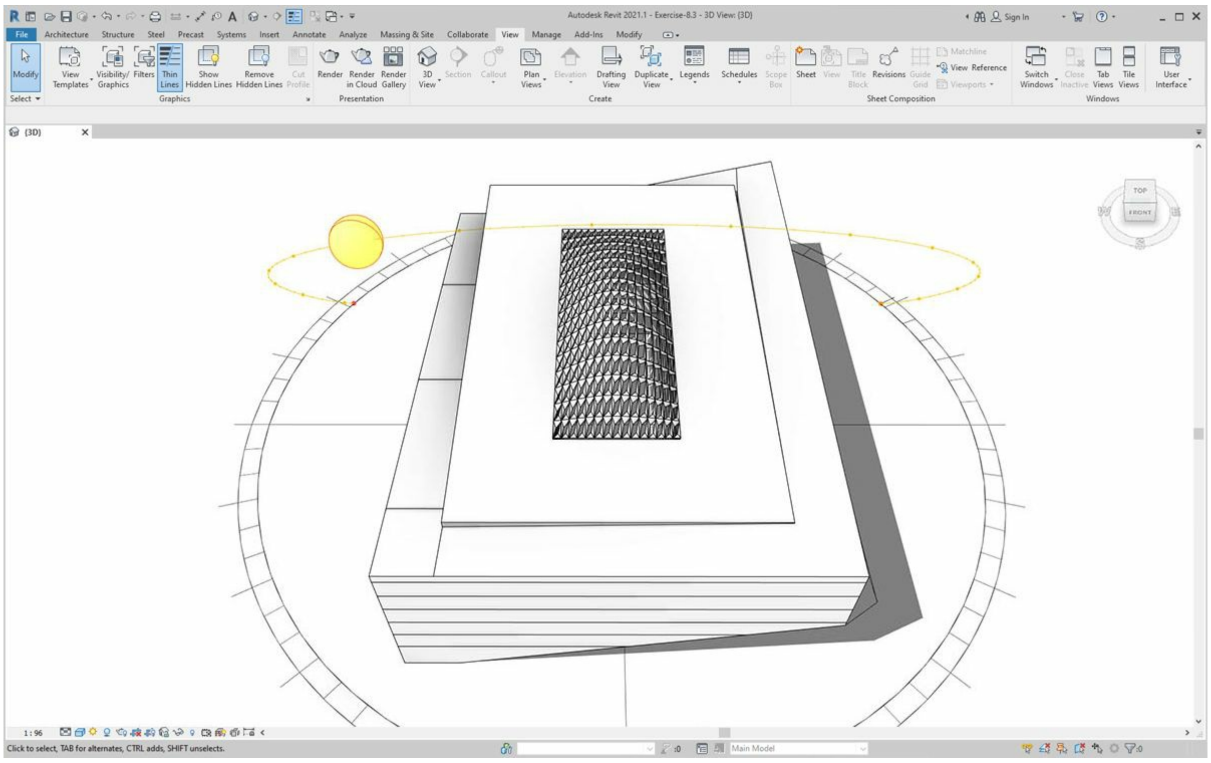


Needs connection to "Panel Pts" from "LunchBox Quad Grid by Face". Remember to set Run to "Manual"

```
0.15;
0.45;
```

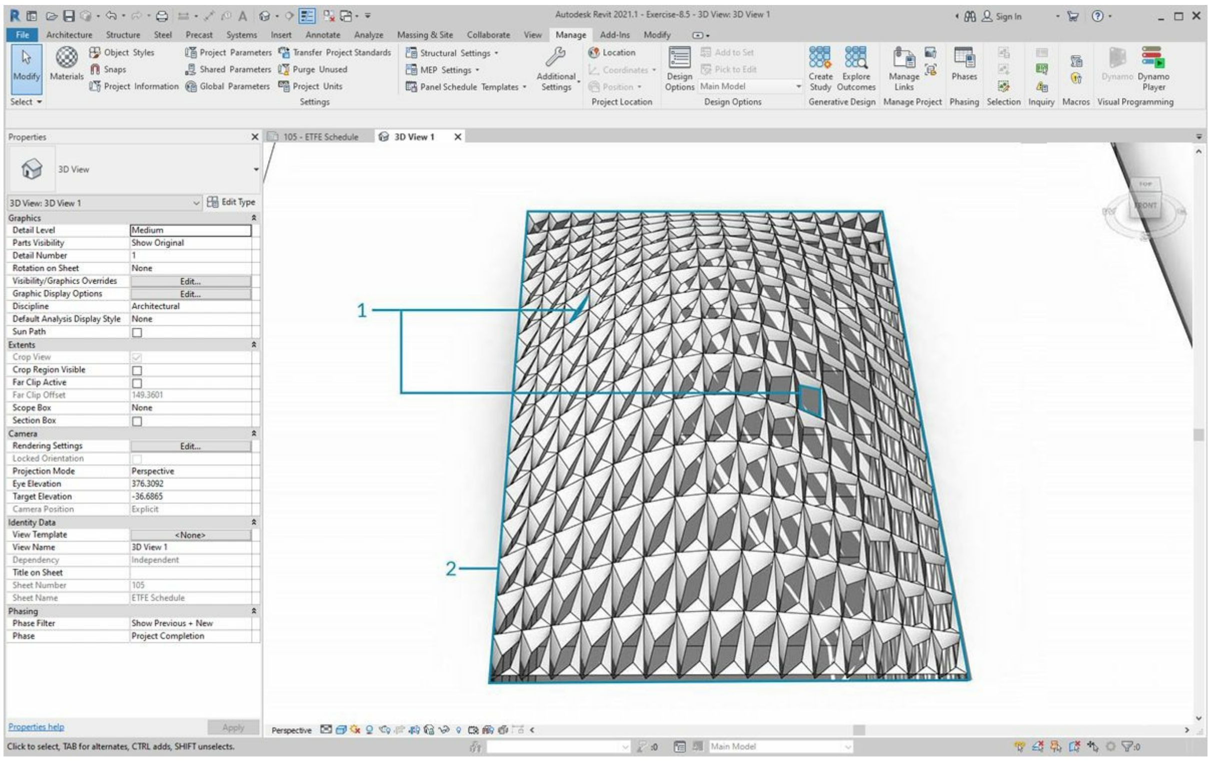
```
"Aperture Ratio";
```

1. Connect the remapped values into a *Element.SetParameterByName* node.
2. Connect the string *"Aperture Ratio"* into the *parameterName* input.
3. Connect the *adaptive components* into the *element* input.
4. Hit *Run*.



Click to select, TAB for alternates, CTRL adds, SHIFT unselects.

Back in Revit, from a distance we can make out the affect of the solar orientation on the aperture of the ETFE panels.



Properties

3D View

3D View: 3D View 1 | Edit Type

Graphics	
Detail Level	Medium
Parts Visibility	Show Original
Detail Number	1
Rotation on Sheet	None
Visibility/Graphics Overrides	Edit...
Graphic Display Options	Edit...
Discipline	Architectural
Default Analysis Display Style	None
Sun Path	☐
Extents	
Crop View	✓
Crop Region Visible	☐
Far Clip Active	☐
Far Clip Offset	149.3601
Scope Box	None
Section Box	☐
Camera	
Rendering Settings	Edit...
Locked Orientation	☐
Projection Mode	Perspective
Eye Elevation	376.3092
Target Elevation	-36.6865
Camera Position	Explicit
Identity Data	
View Template	<None>
View Name	3D View 1
Dependency	Independent
Title on Sheet	
Sheet Number	105
Sheet Name	ETFE Schedule
Phasing	
Phase Filter	Show Previous + New
Phase	Project Completion

Properties help

Click to select, TAB for alternates, CTRL adds, SHIFT unselects.

Zooming in, we see that the ETFE panels are more closed as the face the sun. Our target here is to reduce overheating from solar exposure. If we wanted to let in more light based on solar exposure, we just have to switch the domain on *Math.RemapRange*.

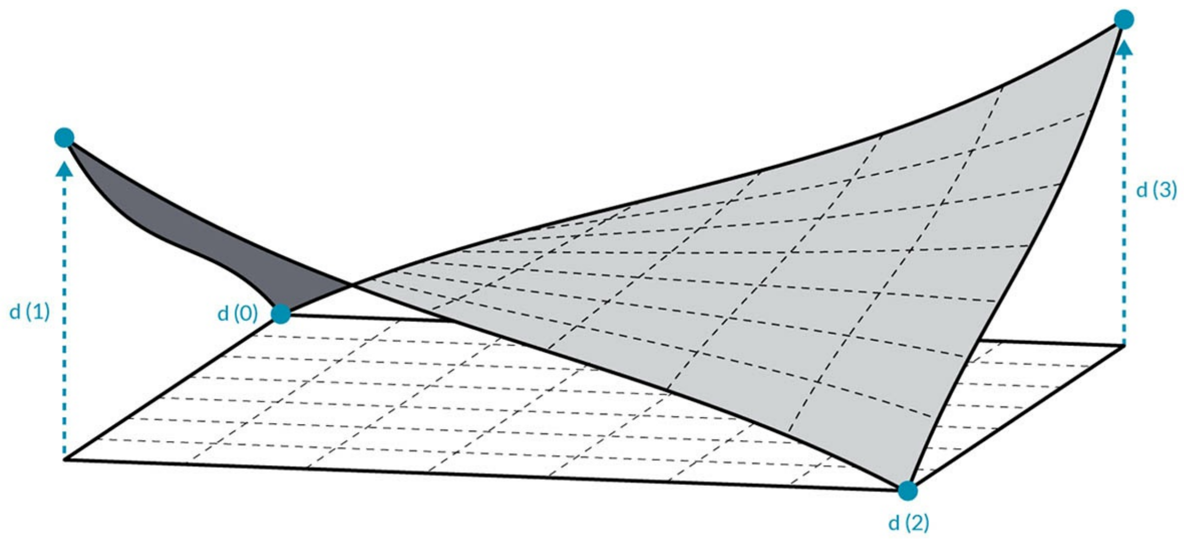
Documenting

Documenting

Editing parameters for documentation follows suit with the lessons learned in prior sections. In this section, we'll look at editing parameters which don't affect the geometric properties of an element, but instead prepare a Revit file for documentation.

Deviation

In the exercise below, we'll use a basic deviation from plane node to create a Revit sheet for documentation. Each panel on our parametrically defined roof structure has a different value for deviation, and we want to call out the range of values using color and by scheduling out the adaptive points to hand off to a facade consultant, engineer, or contractor.

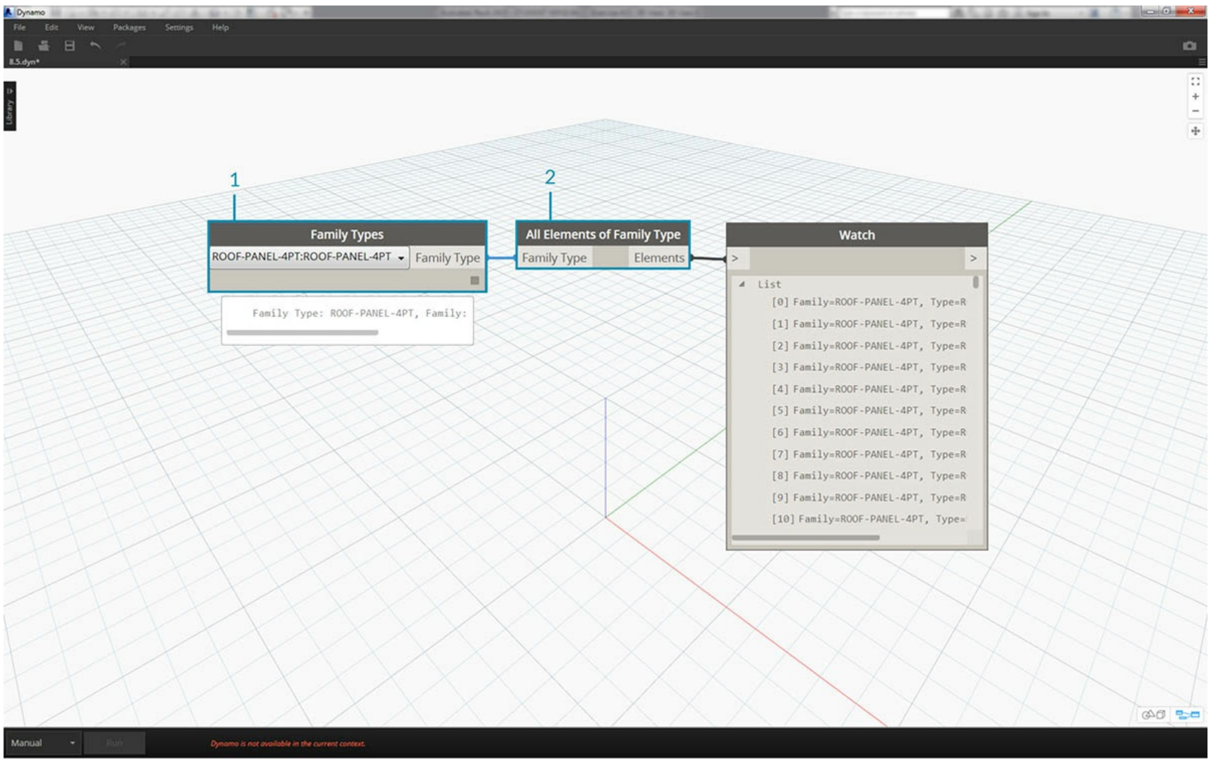


> The deviation from plane node will calculate the distance that the set of four points varies from the best-fit plane between them. This is a quick and easy way to study constructability.

Exercise

> Download the example files that accompanies this exercise (Right click and "Save Link As..."). A full list of example files can be found in the Appendix.
>
> 1. Documenting.dyn
> 2. ARCH-Documenting-BaseFile.rvt

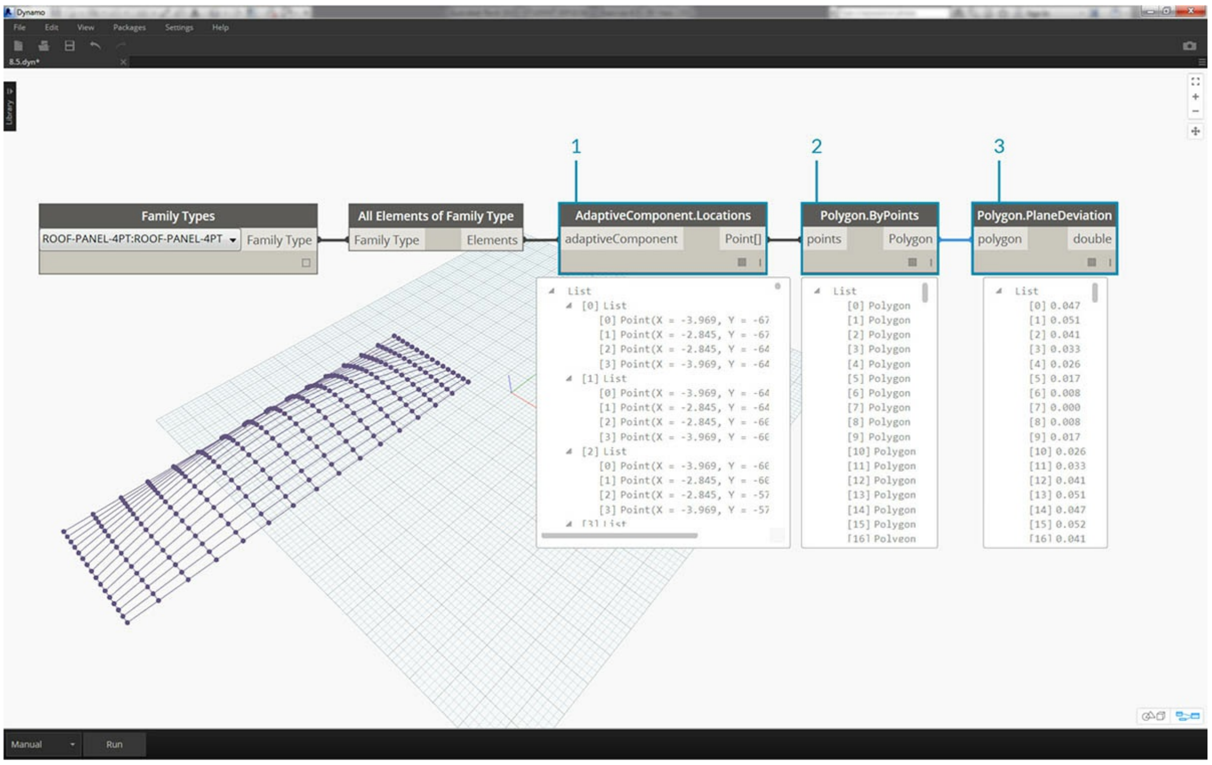
Start with the Revit file for this section (or continue from the previous section). This file has an array of ETFE panels on the roof. We'll reference these panels for this exercise.


1
2
Family Types
ROOF-PANEL-4PT:ROOF-PANEL-4PT
Family Type
All Elements of Family Type
Family Type
Elements
Watch
Manual
Run
Dynamo is not available in the current context.

```
Family Type: ROOF-PANEL-4PT, Family:
```

```
List
    [0] Family=ROOF-PANEL-4PT, Type=R
    [1] Family=ROOF-PANEL-4PT, Type=R
    [2] Family=ROOF-PANEL-4PT, Type=R
    [3] Family=ROOF-PANEL-4PT, Type=R
    [4] Family=ROOF-PANEL-4PT, Type=R
    [5] Family=ROOF-PANEL-4PT, Type=R
    [6] Family=ROOF-PANEL-4PT, Type=R
    [7] Family=ROOF-PANEL-4PT, Type=R
    [8] Family=ROOF-PANEL-4PT, Type=R
    [9] Family=ROOF-PANEL-4PT, Type=R
    [10] Family=ROOF-PANEL-4PT, Type=
```

1. Add a *Family Types* node to the canvas and choose *"ROOF-PANEL-4PT"*.
2. Plug this node into a Select *All Elements of Family Type* node to get all of the elements from Revit into Dynamo.

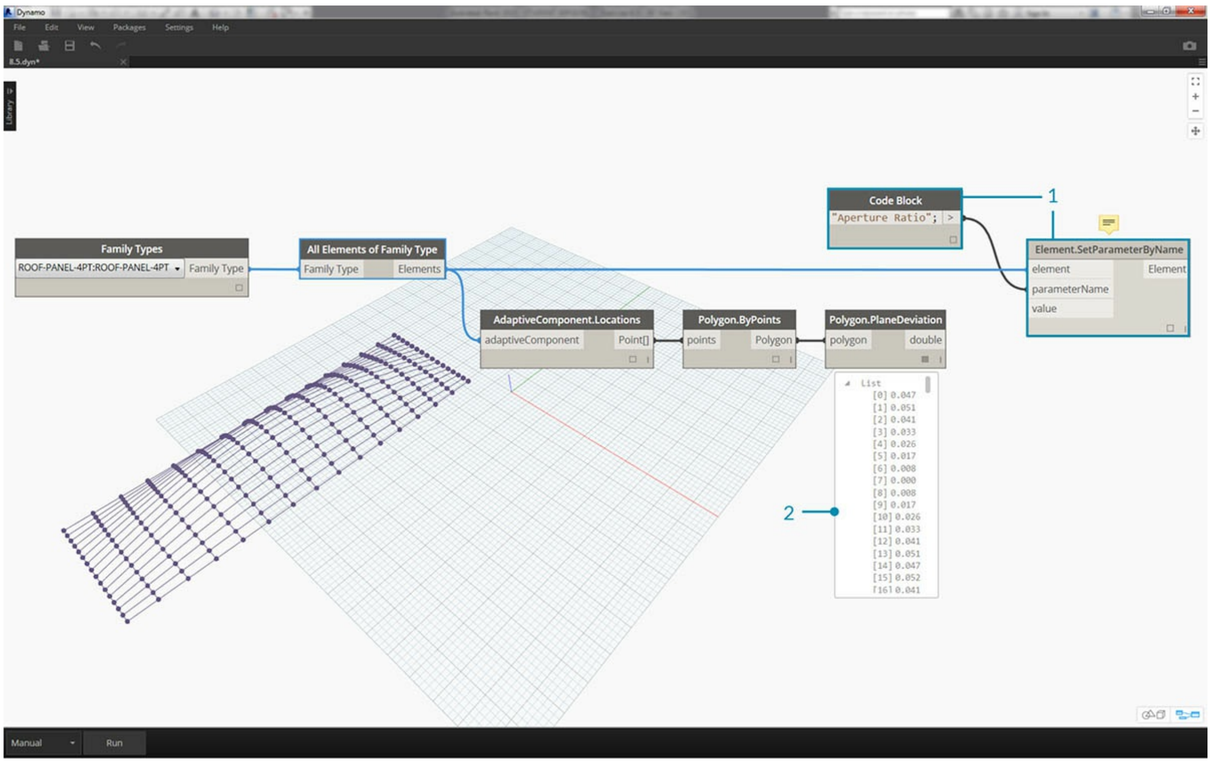


```
List
  [0] List
      [0] Point(X = -3.969, Y = -67
      [1] Point(X = -2.845, Y = -67
      [2] Point(X = -2.845, Y = -64
      [3] Point(X = -3.969, Y = -64
  [1] List
      [0] Point(X = -3.969, Y = -64
      [1] Point(X = -2.845, Y = -64
      [2] Point(X = -2.845, Y = -66
      [3] Point(X = -3.969, Y = -66
  [2] List
      [0] Point(X = -3.969, Y = -66
      [1] Point(X = -2.845, Y = -66
      [2] Point(X = -2.845, Y = -57
      [3] Point(X = -3.969, Y = -57
  [3] List
```

```
List
    [0] Polygon
    [1] Polygon
    [2] Polygon
    [3] Polygon
    [4] Polygon
    [5] Polygon
    [6] Polygon
    [7] Polygon
    [8] Polygon
    [9] Polygon
    [10] Polygon
    [11] Polygon
    [12] Polygon
    [13] Polygon
    [14] Polygon
    [15] Polygon
    [16] Polygon
```

```
List
    [0] 0.047
    [1] 0.051
    [2] 0.041
    [3] 0.033
    [4] 0.026
    [5] 0.017
    [6] 0.008
    [7] 0.000
    [8] 0.008
    [9] 0.017
    [10] 0.026
    [11] 0.033
    [12] 0.041
    [13] 0.051
    [14] 0.047
    [15] 0.052
    [16] 0.041
```

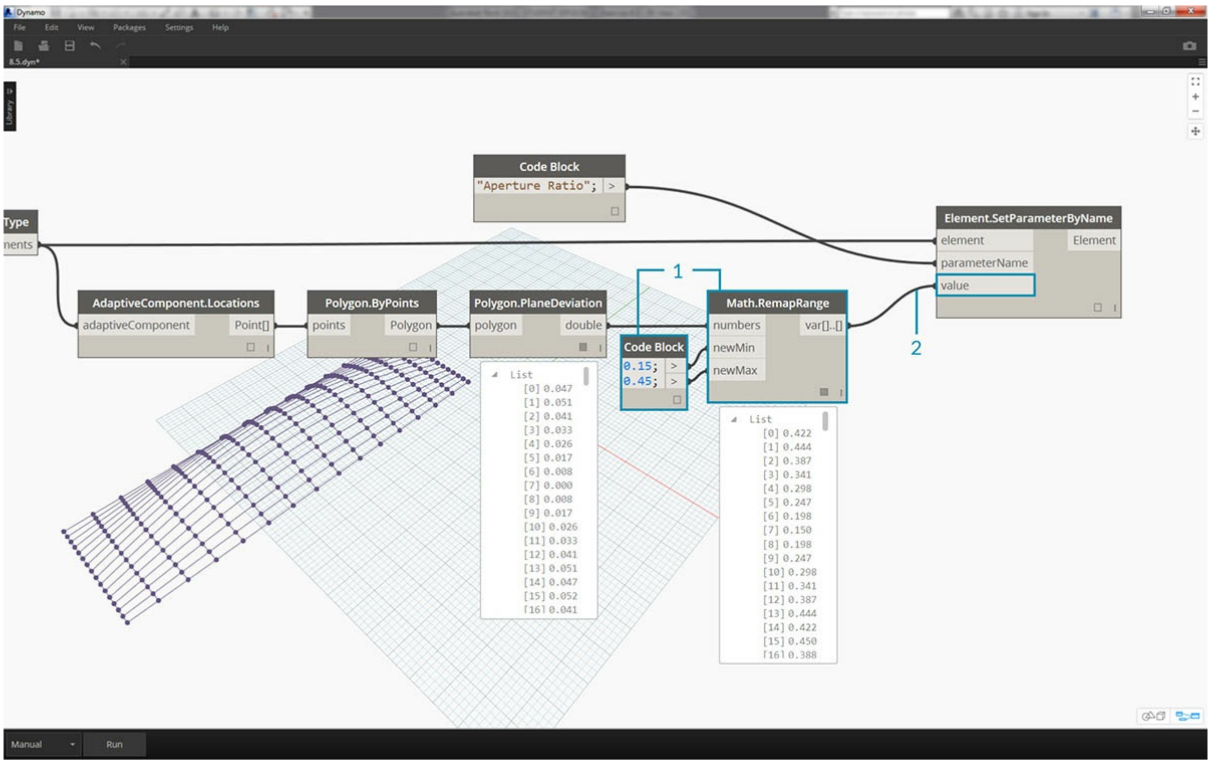
1. Query the location of adaptive points for each element with the *AdaptiveComponent.Locations* node.
2. Create a polygon from these four points with the *Polygon.ByPoints* node. Notice we now have an abstract version of the paneled system in Dynamo without having to import the full geometry of the Revit element.
3. Calculate planar deviation with the *Polygon.PlaneDeviation* node.



```
List
[0] 0.047
[1] 0.051
[2] 0.041
[3] 0.033
[4] 0.026
[5] 0.017
[6] 0.008
[7] 0.000
[8] 0.008
[9] 0.017
[10] 0.026
[11] 0.033
[12] 0.041
[13] 0.051
[14] 0.047
[15] 0.052
[16] 0.041
```

Just for kicks, like the previous exercise, let's set the *aperture ratio* of each panel based on its planar deviation.

1. Add an *Element.SetParameterByName* node to the canvas and connect the adaptive components to the *element* input. Connect a *code block* reading *"Aperture Ratio"* into the *parameterName* input.
2. We cannot directly connect the deviation results into the value input because we need to remap the values to the parameter range.

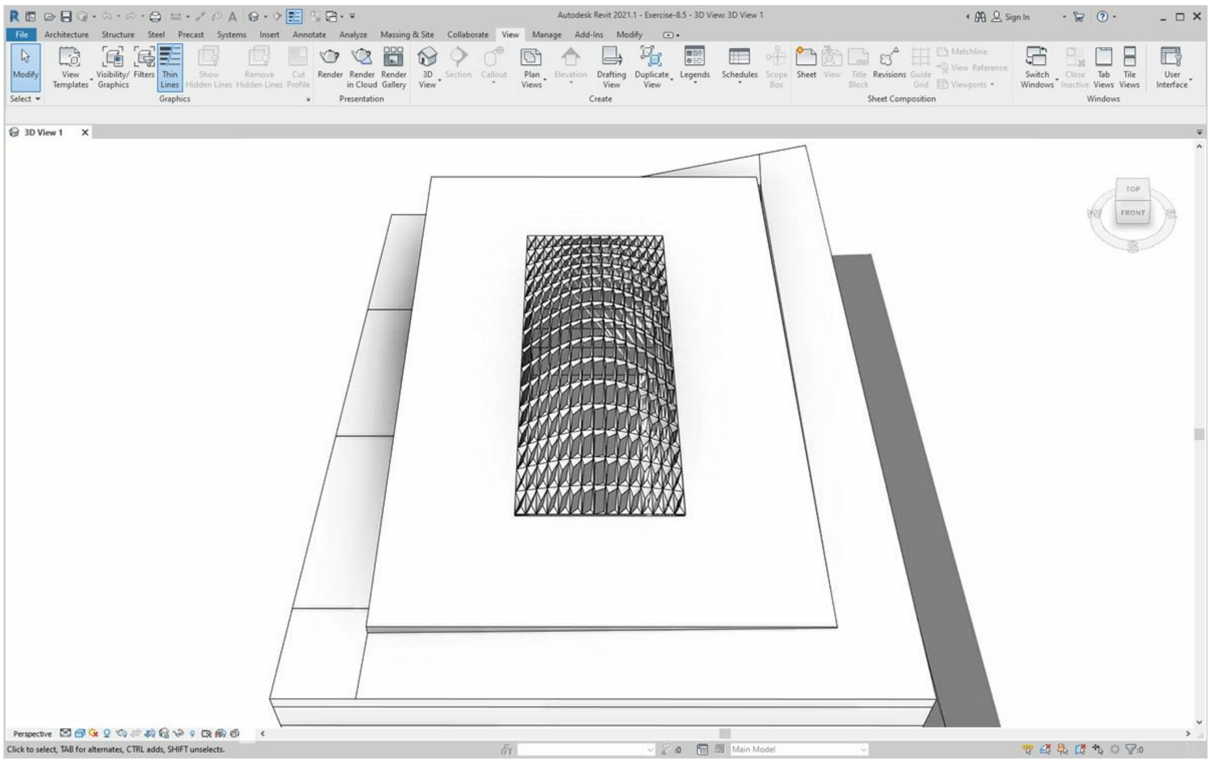


```
0.15;
0.45;
```

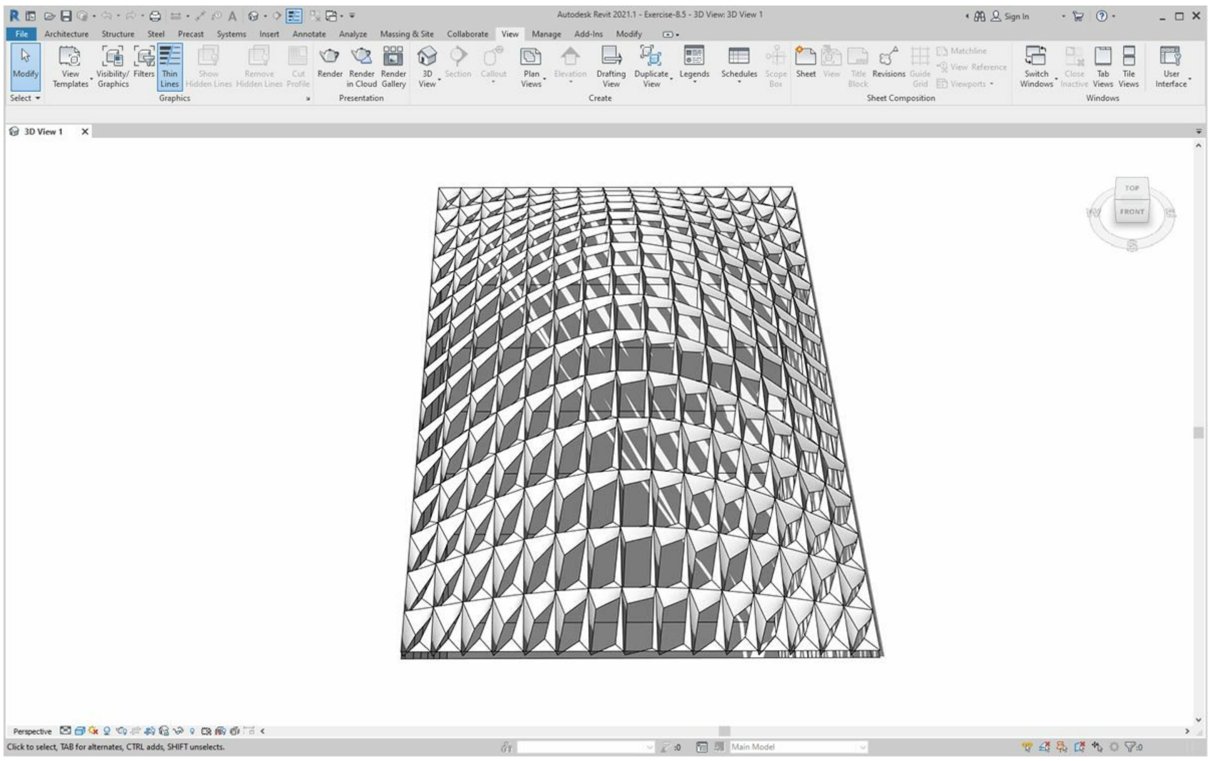
```
List
[0] 0.047
[1] 0.051
[2] 0.041
[3] 0.033
[4] 0.026
[5] 0.017
[6] 0.008
[7] 0.000
[8] 0.008
[9] 0.017
[10] 0.026
[11] 0.033
[12] 0.041
[13] 0.051
[14] 0.047
[15] 0.052
[16] 0.041
```

```
List
[0] 0.422
[1] 0.444
[2] 0.387
[3] 0.341
[4] 0.298
[5] 0.247
[6] 0.198
[7] 0.150
[8] 0.198
[9] 0.247
[10] 0.298
[11] 0.341
[12] 0.387
[13] 0.444
[14] 0.422
[15] 0.450
[16] 0.388
```

1. Using *Math.RemapRange*, remap the deviation values to a domain between *.15* and *.45*.
2. Plug these results into the value input for *Element.SetParameterByName*.


Autodesk Revit 2021.1 - Exercise-8.5 - 3D View: 3D View 1
3D View 1
Click to select, TAB for alternates, CTRL adds, SHIFT unselects.
Main Model

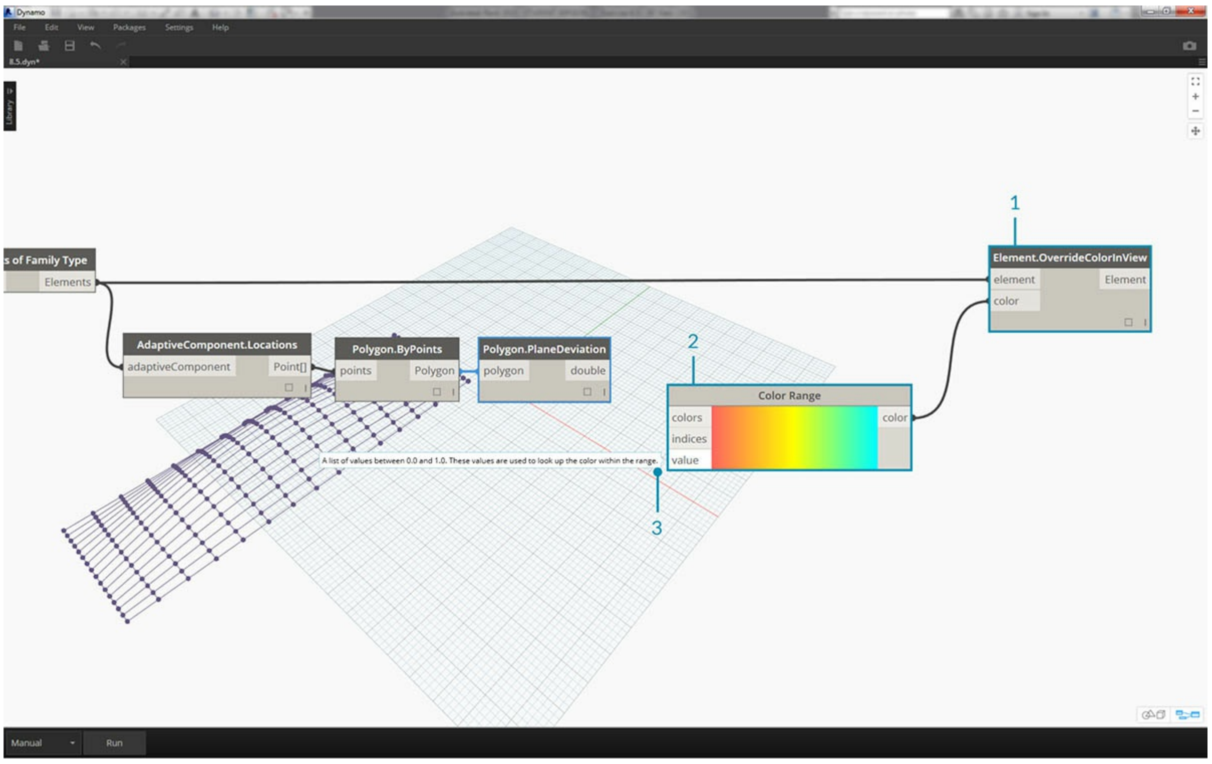
Back in Revit we can *kind of* make sense of the change in aperture across the surface.


Autodesk Revit 2021.1 - Exercise-8.5 - 3D View: 3D View 1
3D View 1
Click to select, TAB for alternates, CTRL adds, SHIFT unselects.
Main Model

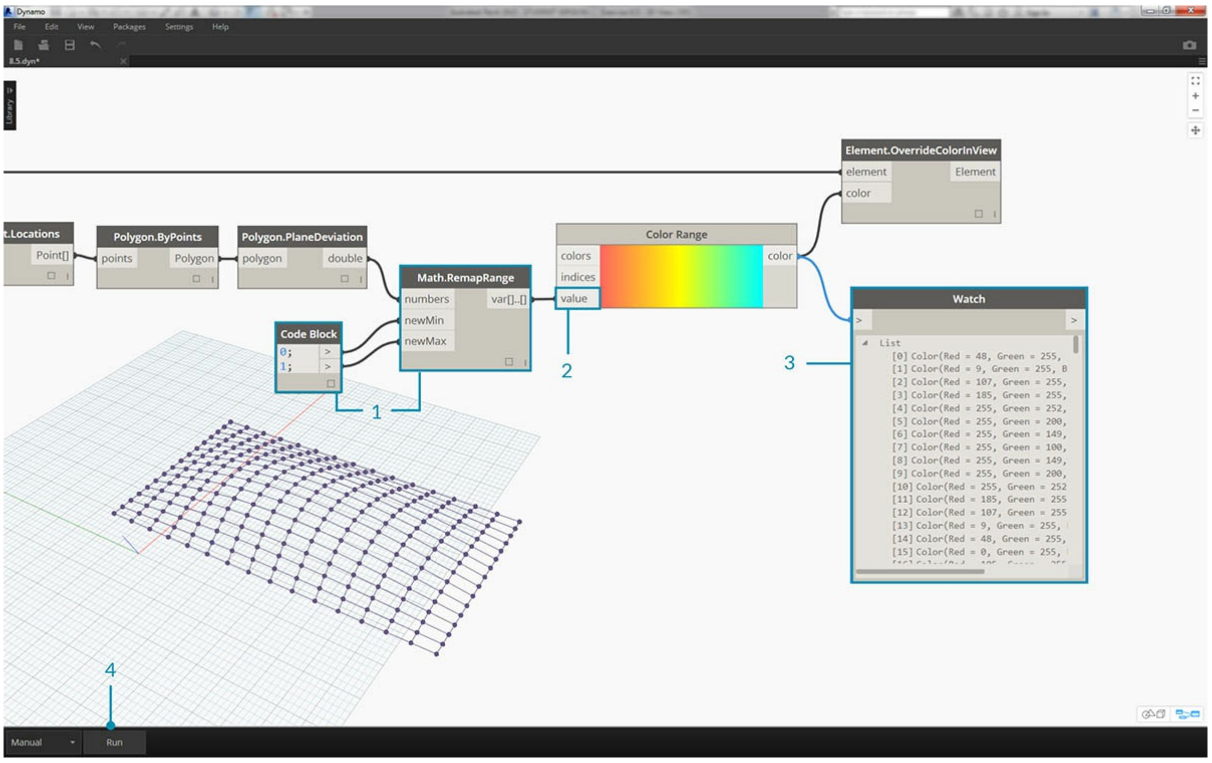
Zooming in, it becomes more clear that the closed panels are weighted towards the corners of the surface. The open corners are towards the top. The corners represent areas of larger deviation while bulge has minimal curvature, so this makes sense.

Color and Documentation

Setting the Aperture Ratio doesn't clearly demonstrate the deviation of panels on the roof, and we're also changing the geometry of the actual element. Suppose we just want to study the deviation from the standpoint of fabrication feasibility. It would be helpful to color the panels based on deviation range for our documentation. We can do that with the series of steps below, and in a very similar process to the steps above.

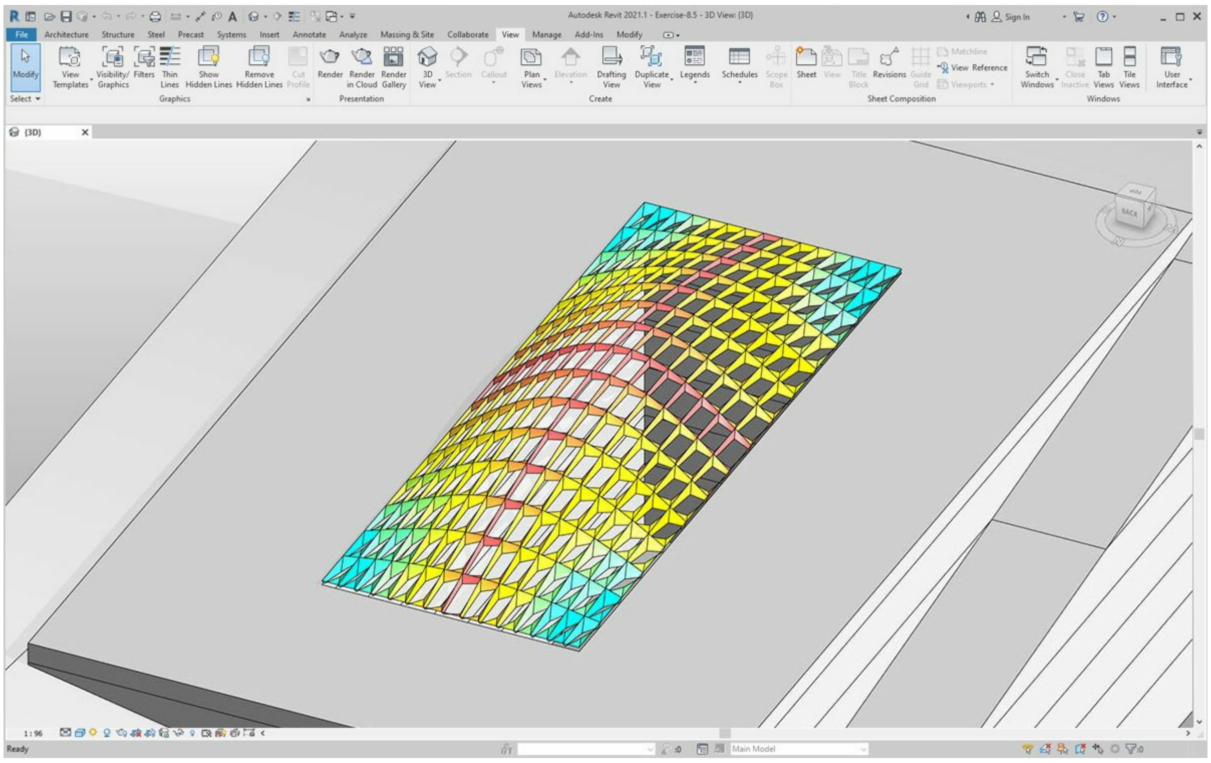


1. Remove the *Element.SetParameterByName* nodes and add *Element.OverrideColorInView*.
2. Add a *Color Range* node to the canvas and plug into the color input of *Element.OverrideColorInView*. We still have to connect the deviation values to the color range in order to create the gradient.
3. Hovering over the *value* input, we can see that the values for the input must be between *0* and *1* in order to map a color to each value. We need to remap the deviation values to this range.

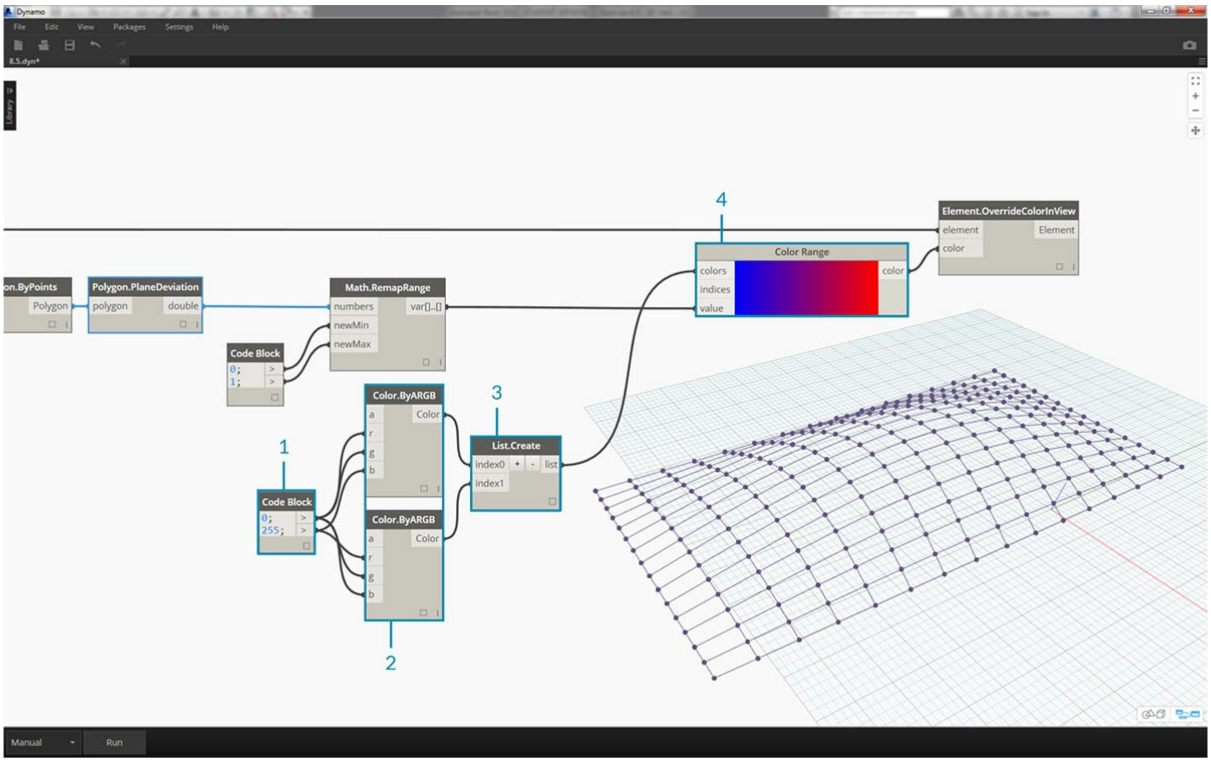


```
List
  [0] Color(Red = 48, Green = 255,
  [1] Color(Red = 9, Green = 255, B
  [2] Color(Red = 107, Green = 255,
  [3] Color(Red = 185, Green = 255,
  [4] Color(Red = 255, Green = 252,
  [5] Color(Red = 255, Green = 200,
  [6] Color(Red = 255, Green = 149,
  [7] Color(Red = 255, Green = 100,
  [8] Color(Red = 255, Green = 149,
  [9] Color(Red = 255, Green = 200,
  [10] Color(Red = 255, Green = 252
  [11] Color(Red = 185, Green = 255
  [12] Color(Red = 107, Green = 255
  [13] Color(Red = 9, Green = 255,
  [14] Color(Red = 48, Green = 255,
  [15] Color(Red = 0, Green = 255,
```

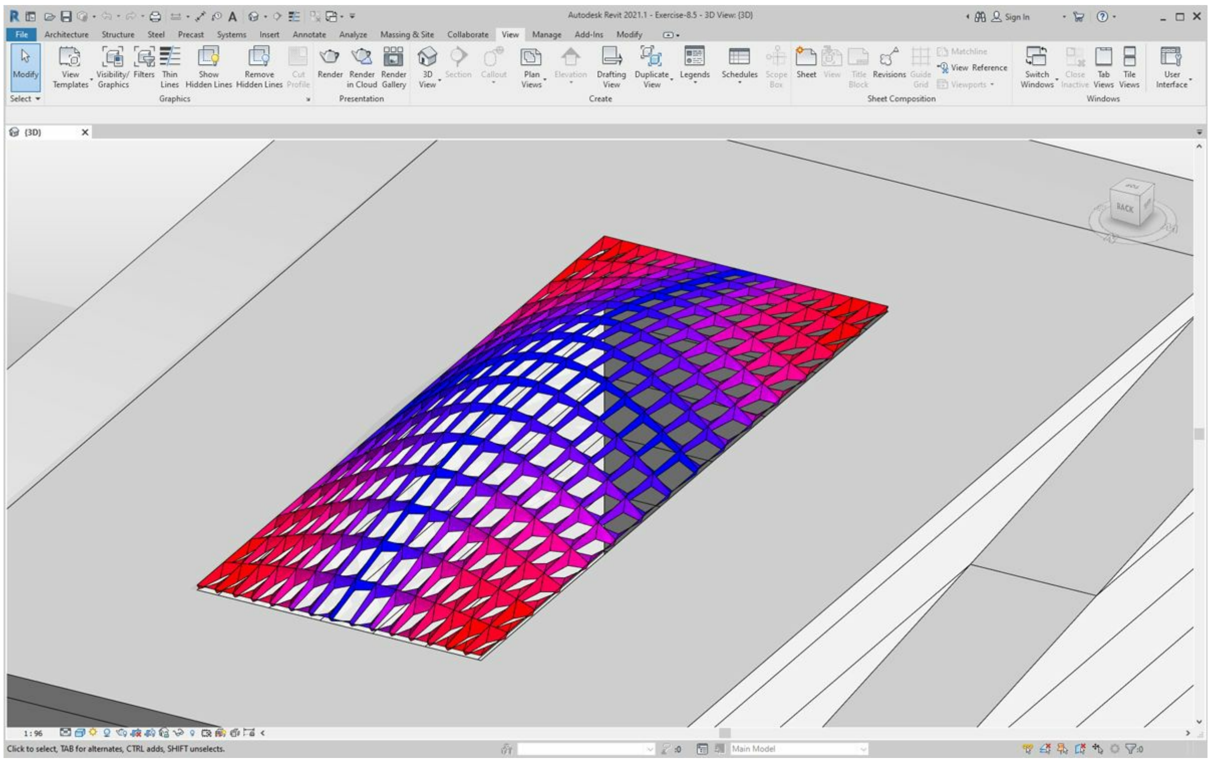
1. Using *Math.RemapRange*, remap the planar deviation values to a range between *0* and *1* (note: you can use the *"MapTo"* node to define a source domain as well).
2. Plug the results into a *Color Range* node.
3. Notice our output is a range of colors instead of a range of numbers.
4. If you're set to Manual, hit *Run*. You should be able to get away with being set to Automatic from this point forward.



Back in Revit, we see a much more legible gradient which is representative of planar deviation based on our color range. But what if we want to customize the colors? Notice that the minimum deviation values are represented in red, which seems to be the opposite of what we'd expect. We want to have maximum deviation to be red, with minimum deviation represented by a calmer color. Let's go back to Dynamo and fix this.

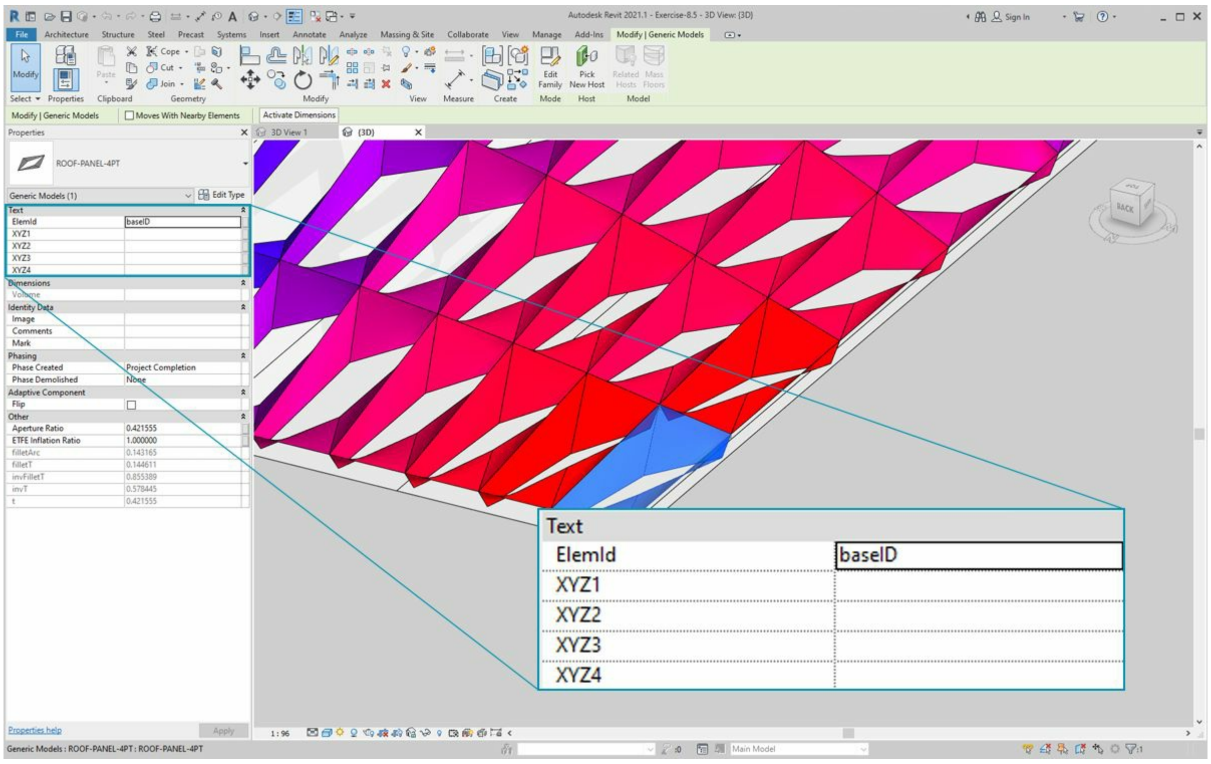


1. Using a *code block*, add two numbers on two different lines: `0;` and `255;`.
2. Create a red and blue color by plugging the appropriate values into two *Color.ByARGB* nodes.
3. Create a list from these two colors.
4. Plug this list into the *colors* input of the *Color Range*, and watch the custom color range update.


Autodesk Revit 2021.1 - Exercise-8.5 - 3D View: {3D}
{3D}
Click to select, TAB for alternates, CTRL adds, SHIFT unselects.
Main Model

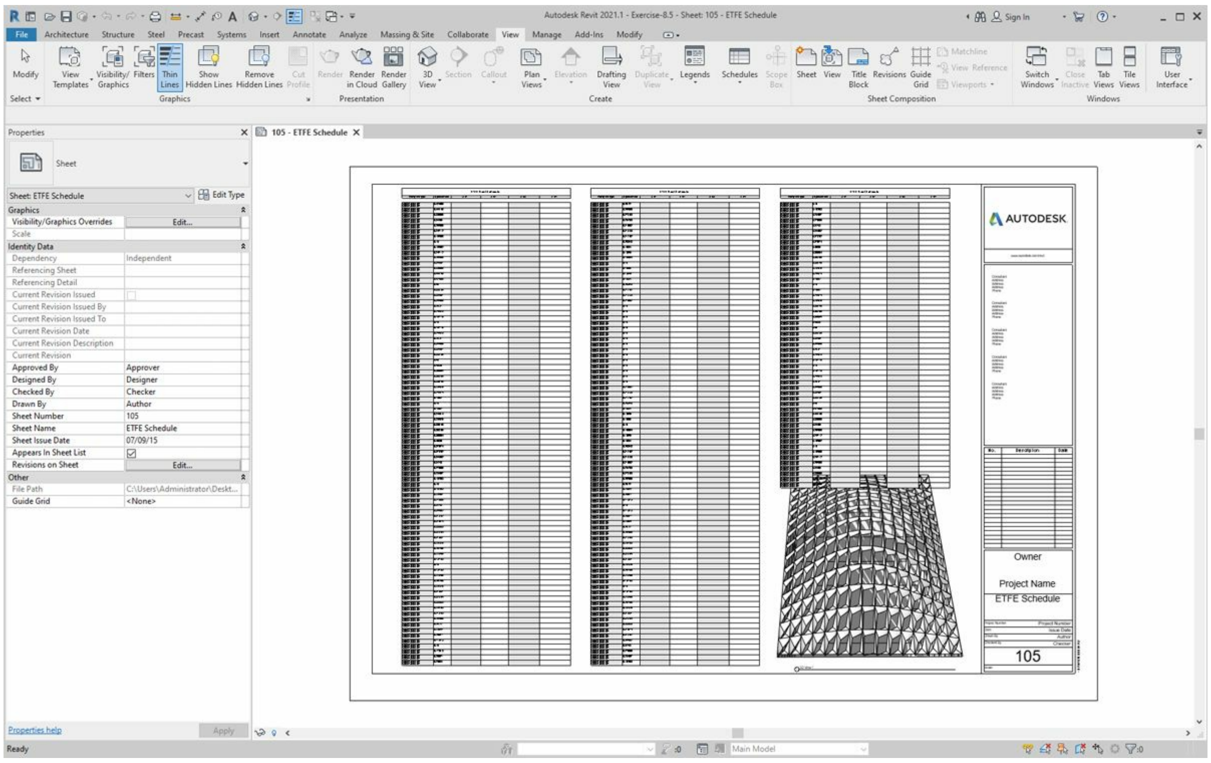
Back in Revit, we can now make better sense of areas of maximum deviation in the corners. Remember, this node is for overriding a color in a view, so it can be really helpful if we had a particular sheet in the set of drawings which focuses on a particular type of analysis.

Scheduling


Autodesk Revit 2021.1 - Exercise-8.5 - 3D View: {3D}
Modify | Generic Models
Moves With Nearby Elements
Activate Dimensions
Properties
ROOF-PANEL-4PT
Generic Models (1)
Edit Type
Generic Models : ROOF-PANEL-4PT : ROOF-PANEL-4PT

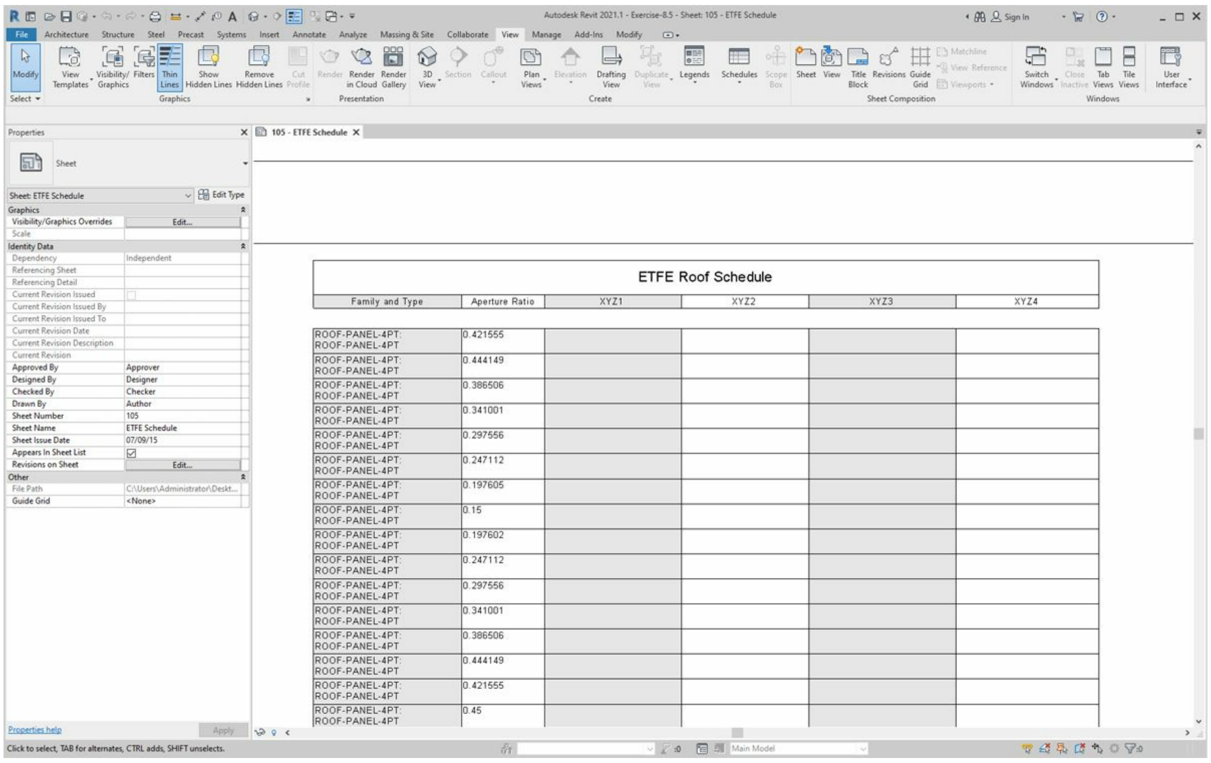
Text	
ElemId	baseID
XYZ1	
XYZ2	
XYZ3	
XYZ4	
Dimensions	
Volume	
Identity Data	
Image	
Comments	
Mark	
Phasing	
Phase Created	Project Completion
Phase Demolished	None
Adaptive Component	
Flip	☐
Other	
Aperture Ratio	0.421555
ETFE Inflation Ratio	1.000000
filletArc	0.143165
filletT	0.144611
invFilletT	0.855389
invT	0.578445
t	0.421555

1. Selecting one ETFE panel in Revit, we see that there are four instance parameters, *XYZ1*, *XYZ2*, *XYZ3*, and *XYZ4*. These are all blank after they're created. These are text-based parameters and need values. We'll use Dynamo to write the adaptive point locations to each parameter. This helps interoperability if the geometry needs to be sent to an engineer of facade consultant.



Sheet: ETFE Schedule	Edit Type
Graphics	
Visibility/Graphics Overrides	Edit...
Scale	
Identity Data	
Dependency	Independent
Referencing Sheet	
Referencing Detail	
Current Revision Issued	☐
Current Revision Issued By	
Current Revision Issued To	
Current Revision Date	
Current Revision Description	
Current Revision	
Approved By	Approver
Designed By	Designer
Checked By	Checker
Drawn By	Author
Sheet Number	105
Sheet Name	ETFE Schedule
Sheet Issue Date	07/09/15
Appears In Sheet List	✓
Revisions on Sheet	Edit...
Other	
File Path	C:\Users\Administrator\Deskt...
Guide Grid	<None>

In a sample sheet, we have a large, empty schedule. The XYZ parameters are shared parameters in the Revit file, which allows us to add them to the schedule.

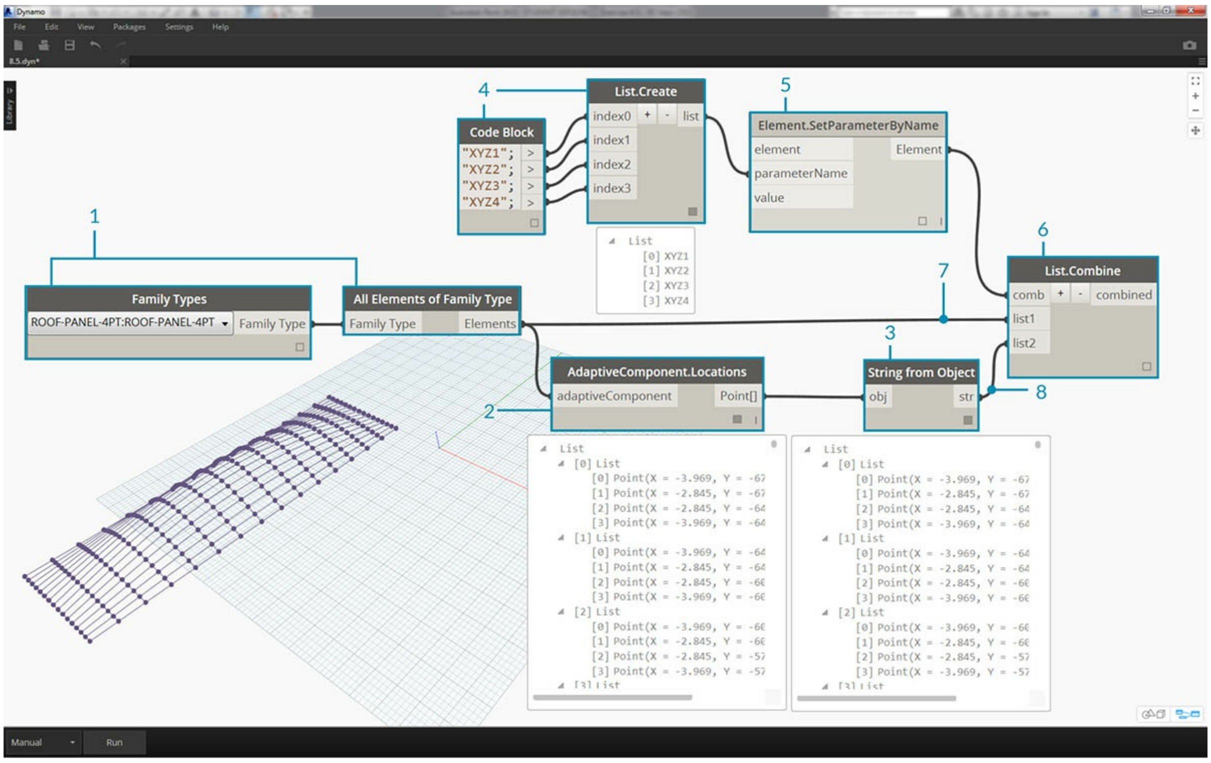


Sheet: ETFE Schedule	Edit Type
Graphics	
Visibility/Graphics Overrides	Edit...
Scale	
Identity Data	
Dependency	Independent
Referencing Sheet	
Referencing Detail	
Current Revision Issued	☐
Current Revision Issued By	
Current Revision Issued To	
Current Revision Date	
Current Revision Description	
Current Revision	
Approved By	Approver
Designed By	Designer
Checked By	Checker
Drawn By	Author
Sheet Number	105
Sheet Name	ETFE Schedule
Sheet Issue Date	07/09/15
Appears In Sheet List	✓
Revisions on Sheet	Edit...
Other	
File Path	C:\Users\Administrator\Deskt...
Guide Grid	<None>

ETFE Roof Schedule

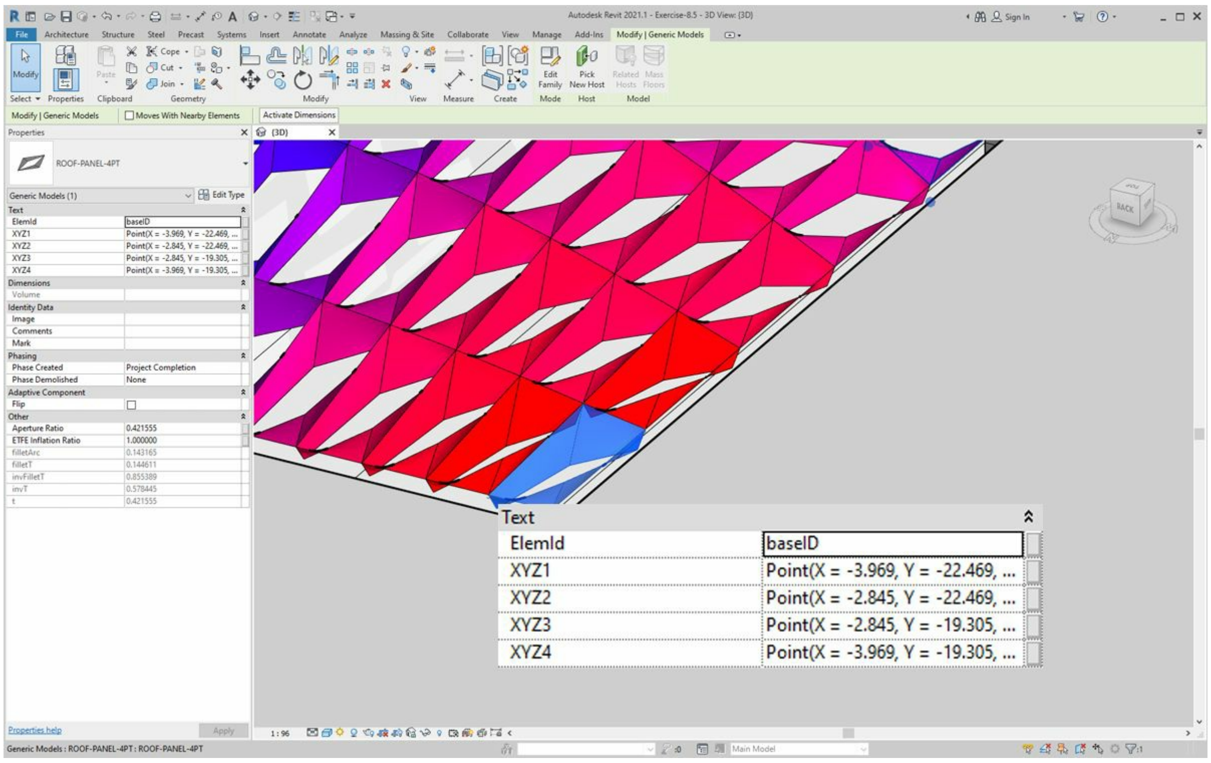
Family and Type	Aperture Ratio	XYZ1	XYZ2	XYZ3	XYZ4
ROOF-PANEL-4PT: ROOF-PANEL-4PT	0.421555				
ROOF-PANEL-4PT: ROOF-PANEL-4PT	0.444149				
ROOF-PANEL-4PT: ROOF-PANEL-4PT	0.386506				
ROOF-PANEL-4PT: ROOF-PANEL-4PT	0.341001				
ROOF-PANEL-4PT: ROOF-PANEL-4PT	0.297556				
ROOF-PANEL-4PT: ROOF-PANEL-4PT	0.247112				
ROOF-PANEL-4PT: ROOF-PANEL-4PT	0.197605				
ROOF-PANEL-4PT: ROOF-PANEL-4PT	0.15				
ROOF-PANEL-4PT: ROOF-PANEL-4PT	0.197602				
ROOF-PANEL-4PT: ROOF-PANEL-4PT	0.247112				
ROOF-PANEL-4PT: ROOF-PANEL-4PT	0.297556				
ROOF-PANEL-4PT: ROOF-PANEL-4PT	0.341001				
ROOF-PANEL-4PT: ROOF-PANEL-4PT	0.386506				
ROOF-PANEL-4PT: ROOF-PANEL-4PT	0.444149				
ROOF-PANEL-4PT: ROOF-PANEL-4PT	0.421555				
ROOF-PANEL-4PT: ROOF-PANEL-4PT	0.45				

Zooming in, the XYZ parameters are yet to be filled in. The first two parameters are taken care of by Revit.


1
2
3
4
5
6
7
8
Code Block
"XYZ1";
"XYZ2";
"XYZ3";
"XYZ4";
List.Create
Element.SetParameterByName
List.Combine
Family Types
ROOF-PANEL-4PT:ROOF-PANEL-4PT
All Elements of Family Type
AdaptiveComponent.Locations
String from Object
Manual
Run

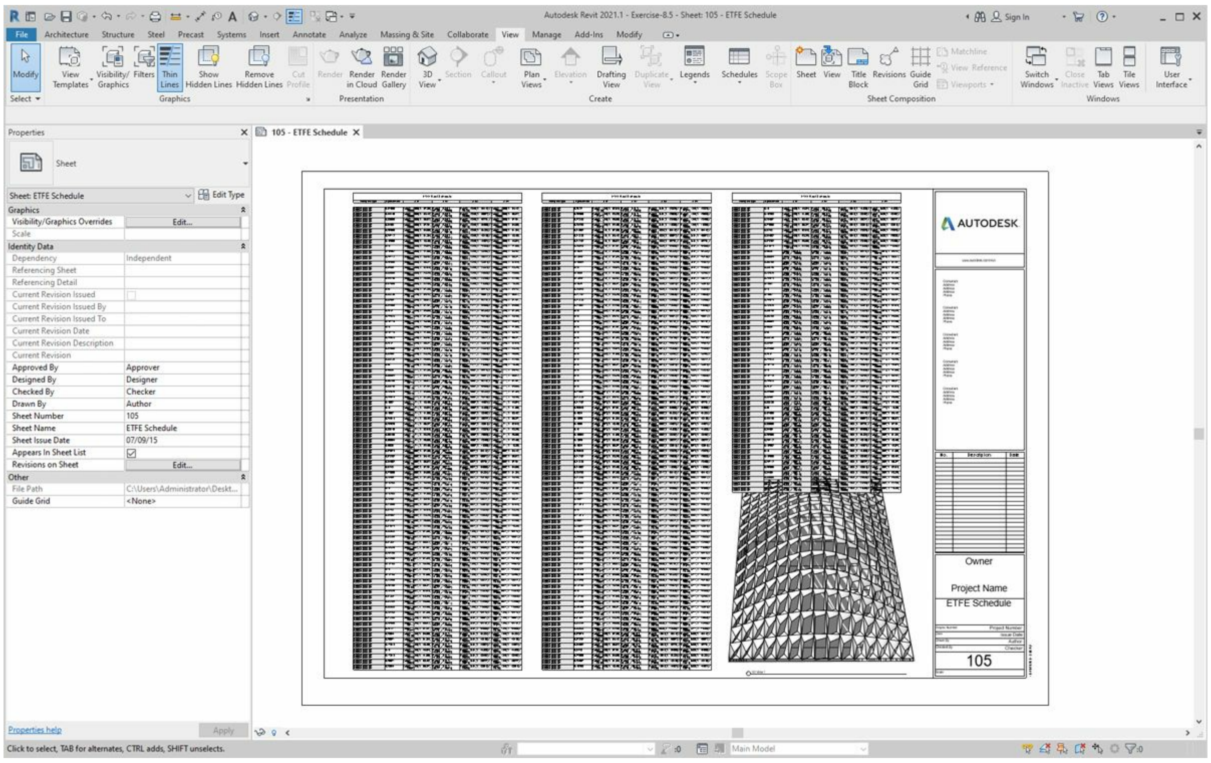
To write in these values, we'll do a complex list operation. The graph itself is simple, but the concepts build heavily from the list mapping as discussed in the list chapter.

1. Select all the adaptive components with two nodes.
2. Extract the location of each point with *AdaptiveComponent.Locations*.
3. Convert these points to strings. Remember, the parameter is text-based so we need to input the correct data type.
4. Create a list of the four strings which define the parameters to change: *XYZ1*, *XYZ2*, *XYZ3*, and *XYZ4*.
5. Plug this list into the *parameterName* input of *Element.SetParameterByName*.
6. Connect *Element.SetParameterByName* into the the *combinator* input of *List.Combine*.
7. Connect the *adaptive components* into *list1*.
8. Connect *String* from Object into *list2*.
9. We are list mapping here, because we are writing four values for each element, which creates a complex data structure. The *List.Combine* node defines an operation one step down in the data hierarchy. This is why element and value inputs are left blank. *List.Combine* is connecting the sublists of its inputs into the empty inputs of *List.SetParameterByName*, based on the order in which they are connected.



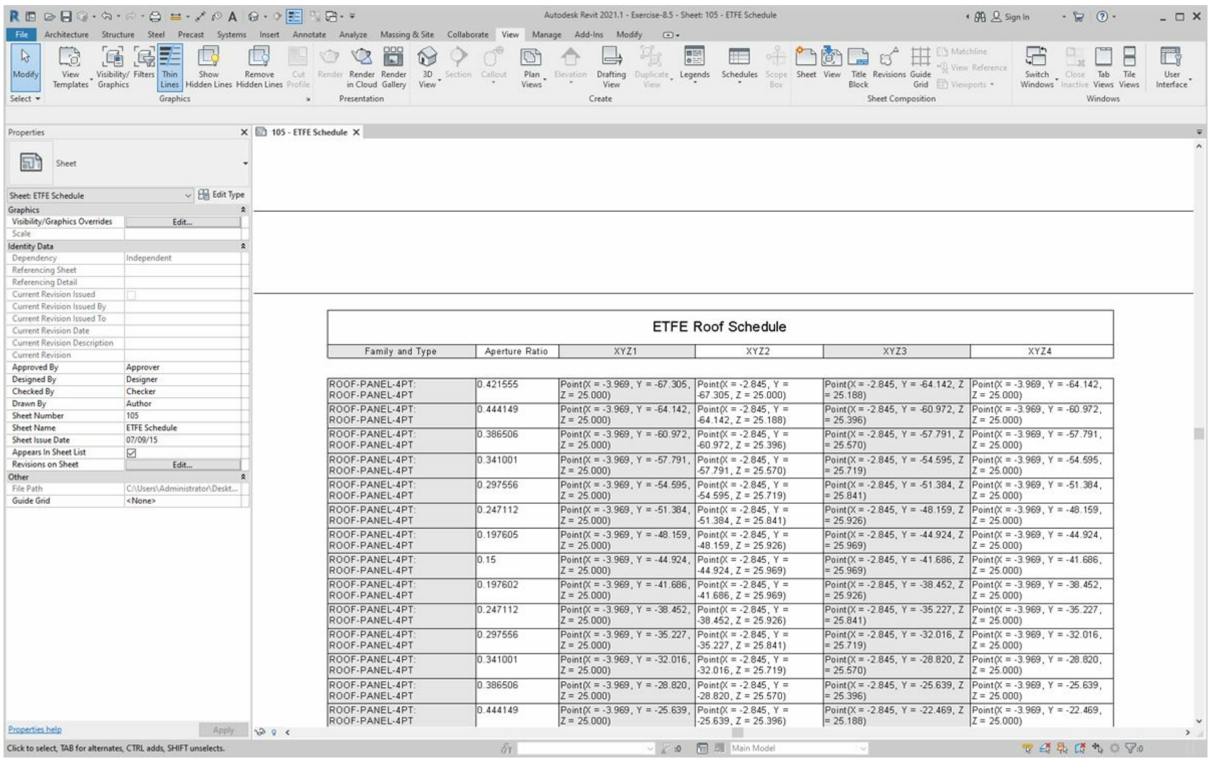
Text	
ElemId	baseID
XYZ1	Point(X = -3.969, Y = -22.469, ...
XYZ2	Point(X = -2.845, Y = -22.469, ...
XYZ3	Point(X = -2.845, Y = -19.305, ...
XYZ4	Point(X = -3.969, Y = -19.305, ...

Selecting a panel in Revit, we see now that we have string values for each parameter. Realistically, we would create a simpler format to write a point (X,Y,Z). This can be done with string operations in Dynamo, but we're bypassing that here to stay within the scope of this chapter.



Sheet: ETFE Schedule	
Visibility/Graphics Overrides	Edit...
Dependency	Independent
Approved By	Approver
Designed By	Designer
Checked By	Checker
Drawn By	Author
Sheet Number	105
Sheet Name	ETFE Schedule
Sheet Issue Date	07/09/15
Appears In Sheet List	✓
Revisions on Sheet	Edit...
File Path	C:\Users\Administrator\Deskt...
Guide Grid	<None>

A view of the sample schedule with parameters filled in.



ETFE Roof Schedule

Family and Type	Aperture Ratio	XYZ1	XYZ2	XYZ3	XYZ4
ROOF-PANEL-4PT: ROOF-PANEL-4PT	0.421555	Point(X = -3.969, Y = -67.305, Z = 25.000)	Point(X = -2.845, Y = -67.305, Z = 25.000)	Point(X = -2.845, Y = -64.142, Z = 25.188)	Point(X = -3.969, Y = -64.142, Z = 25.000)
ROOF-PANEL-4PT: ROOF-PANEL-4PT	0.444149	Point(X = -3.969, Y = -64.142, Z = 25.000)	Point(X = -2.845, Y = -64.142, Z = 25.188)	Point(X = -2.845, Y = -60.972, Z = 25.396)	Point(X = -3.969, Y = -60.972, Z = 25.000)
ROOF-PANEL-4PT: ROOF-PANEL-4PT	0.386506	Point(X = -3.969, Y = -60.972, Z = 25.000)	Point(X = -2.845, Y = -60.972, Z = 25.396)	Point(X = -2.845, Y = -57.791, Z = 25.570)	Point(X = -3.969, Y = -57.791, Z = 25.000)
ROOF-PANEL-4PT: ROOF-PANEL-4PT	0.341001	Point(X = -3.969, Y = -57.791, Z = 25.000)	Point(X = -2.845, Y = -57.791, Z = 25.570)	Point(X = -2.845, Y = -54.595, Z = 25.719)	Point(X = -3.969, Y = -54.595, Z = 25.000)
ROOF-PANEL-4PT: ROOF-PANEL-4PT	0.297556	Point(X = -3.969, Y = -54.595, Z = 25.000)	Point(X = -2.845, Y = -54.595, Z = 25.719)	Point(X = -2.845, Y = -51.384, Z = 25.841)	Point(X = -3.969, Y = -51.384, Z = 25.000)
ROOF-PANEL-4PT: ROOF-PANEL-4PT	0.247112	Point(X = -3.969, Y = -51.384, Z = 25.000)	Point(X = -2.845, Y = -51.384, Z = 25.841)	Point(X = -2.845, Y = -48.159, Z = 25.926)	Point(X = -3.969, Y = -48.159, Z = 25.000)
ROOF-PANEL-4PT: ROOF-PANEL-4PT	0.197605	Point(X = -3.969, Y = -48.159, Z = 25.000)	Point(X = -2.845, Y = -48.159, Z = 25.926)	Point(X = -2.845, Y = -44.924, Z = 25.969)	Point(X = -3.969, Y = -44.924, Z = 25.000)
ROOF-PANEL-4PT: ROOF-PANEL-4PT	0.15	Point(X = -3.969, Y = -44.924, Z = 25.000)	Point(X = -2.845, Y = -44.924, Z = 25.969)	Point(X = -2.845, Y = -41.686, Z = 25.969)	Point(X = -3.969, Y = -41.686, Z = 25.000)
ROOF-PANEL-4PT: ROOF-PANEL-4PT	0.197602	Point(X = -3.969, Y = -41.686, Z = 25.000)	Point(X = -2.845, Y = -41.686, Z = 25.969)	Point(X = -2.845, Y = -38.452, Z = 25.926)	Point(X = -3.969, Y = -38.452, Z = 25.000)
ROOF-PANEL-4PT: ROOF-PANEL-4PT	0.247112	Point(X = -3.969, Y = -38.452, Z = 25.000)	Point(X = -2.845, Y = -38.452, Z = 25.926)	Point(X = -2.845, Y = -35.227, Z = 25.841)	Point(X = -3.969, Y = -35.227, Z = 25.000)
ROOF-PANEL-4PT: ROOF-PANEL-4PT	0.297556	Point(X = -3.969, Y = -35.227, Z = 25.000)	Point(X = -2.845, Y = -35.227, Z = 25.841)	Point(X = -2.845, Y = -32.016, Z = 25.719)	Point(X = -3.969, Y = -32.016, Z = 25.000)
ROOF-PANEL-4PT: ROOF-PANEL-4PT	0.341001	Point(X = -3.969, Y = -32.016, Z = 25.000)	Point(X = -2.845, Y = -32.016, Z = 25.719)	Point(X = -2.845, Y = -28.820, Z = 25.570)	Point(X = -3.969, Y = -28.820, Z = 25.000)
ROOF-PANEL-4PT: ROOF-PANEL-4PT	0.386506	Point(X = -3.969, Y = -28.820, Z = 25.000)	Point(X = -2.845, Y = -28.820, Z = 25.570)	Point(X = -2.845, Y = -25.639, Z = 25.396)	Point(X = -3.969, Y = -25.639, Z = 25.000)
ROOF-PANEL-4PT: ROOF-PANEL-4PT	0.444149	Point(X = -3.969, Y = -25.639, Z = 25.000)	Point(X = -2.845, Y = -25.639, Z = 25.396)	Point(X = -2.845, Y = -22.469, Z = 25.188)	Point(X = -3.969, Y = -22.469, Z = 25.000)

Each ETFE panel now has the XYZ coordinates written for each adaptive point, representing the corners of each panel for fabrication.

Dictionaries in Dynamo

Dictionaries in Dynamo

Dictionaries represent a collection of data that is related to another piece of data known as a key. Dictionaries expose the ability to search for, delete and insert data into a collection.

Essentially, we can think of a dictionary as a really smart way to look something up.

While dictionary functionality has been available in Dynamo for some time, Dynamo 2.0 introduces a new way of managing this data type.

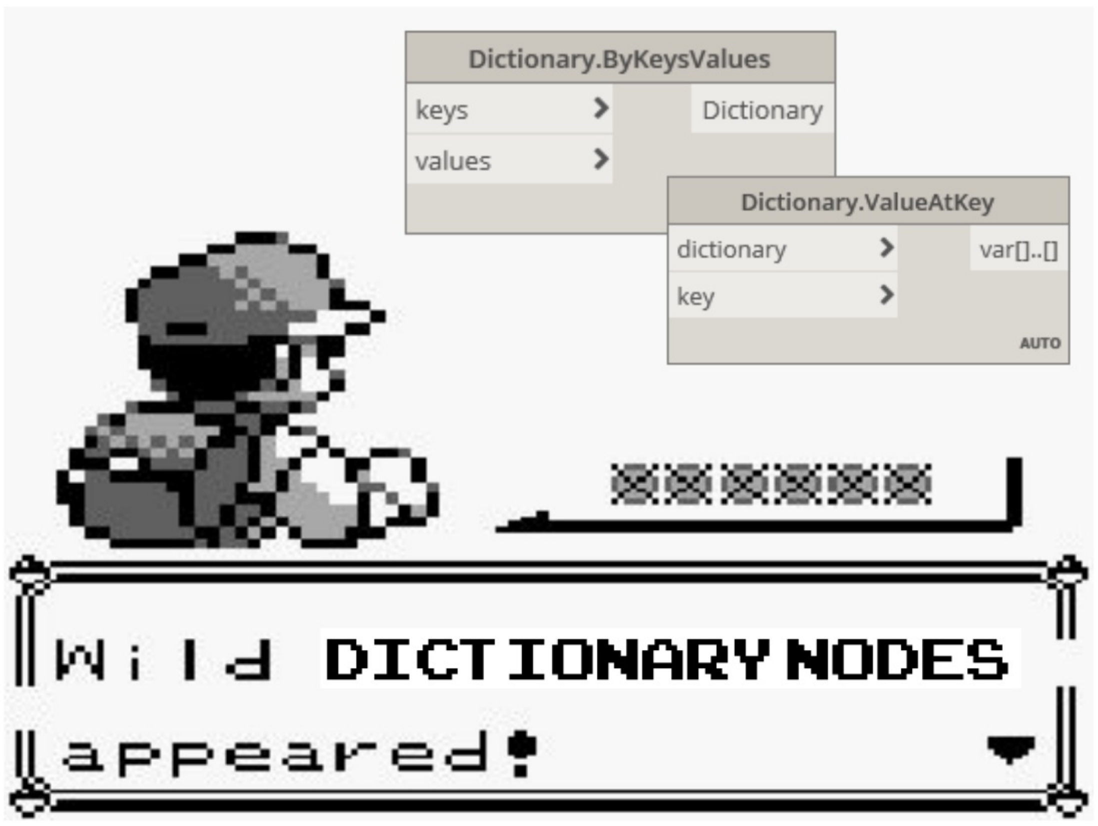


Image Courtesy of [*sixtysecondrevit.com*]

What is a Dictionary

Dictionaries

Dynamo 2.0 introduces the concept of separating the dictionary data type from the list data type. This change can pose some significant changes to how you create and work with data in your workflows. Prior to 2.0, dictionaries and lists were combined as a data type. In short, lists were actually dictionaries with integer keys.

- **What is a dictionary?**

  A dictionary is a data type composed of a collection of key-value pairs where each key is unique in each collection. A dictionary has no order and basically you can "look things up" using a key instead of an index value like in a list. *In Dynamo 2.0, keys can only be strings.*

- **What is a list?**

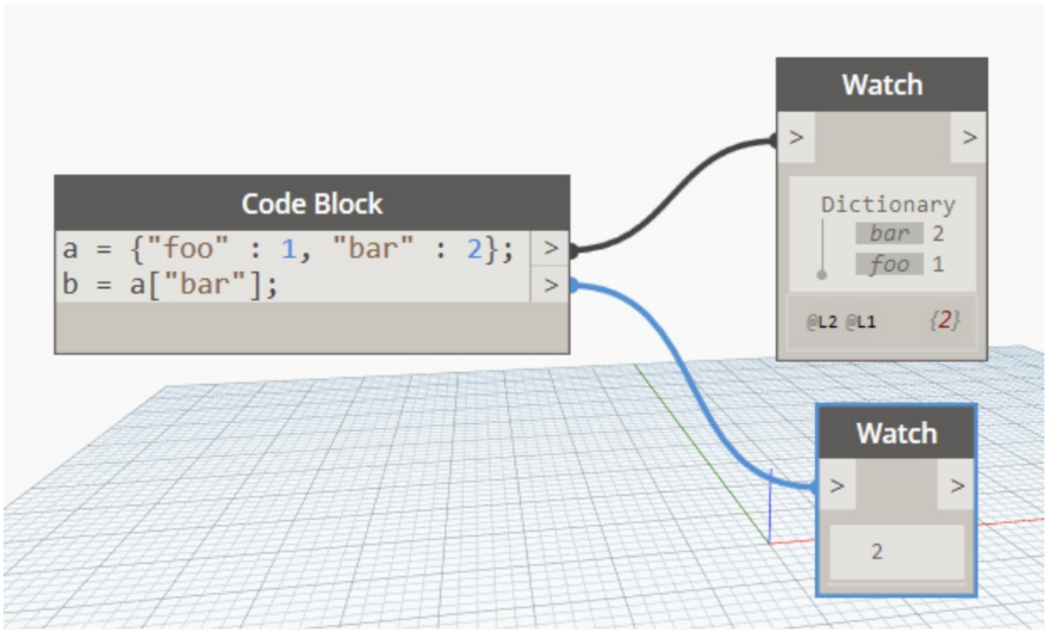
  A list is a data type composed of a collection of ordered values. In Dynamo, lists use integers as index values.

- **Why was this change made and why should I care?**

  The separation of dictionaries from lists introduces dictionaries as a first-class citizen that you can use to quickly and easily store and lookup values without needing to remember an index value or maintain a strict list structure throughout your workflow. During user testing, we saw a significant reduction in graph size when dictionaries were utilized instead of several `GetItemAtIndex` nodes.

- **What are the changes?**

  - *Syntax* changes have occurred that change how you will initialize and work with dictionaries and lists in code blocks.
    - Dictionaries use the following syntax `{key:value}`
    - Lists use the following syntax `[value,value,value]`
  - *New nodes* have been introduced to the library to help you create, modify, and query dictionaries.
  - Lists created in 1.x code blocks will automatically migrated on load of the script to the new list syntax that uses square brackets `[ ]` instead of curly brackets `{ }`

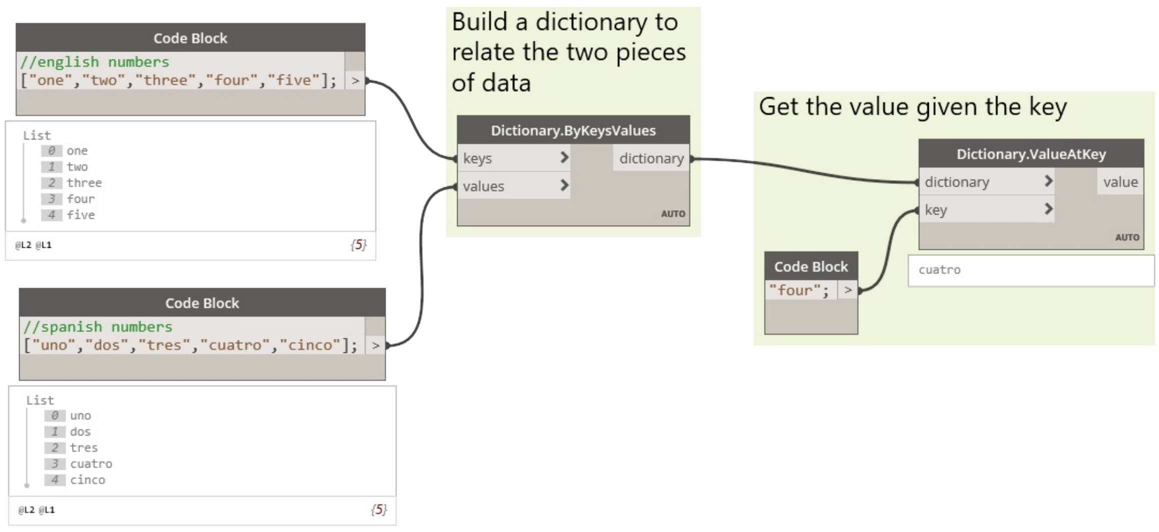


```
a = {"foo" : 1, "bar" : 2};
b = a["bar"];
```

- **Why should I care? What would you use these for?**

  In computer science, Dictionaries - like lists- are collections of objects. While lists are in a specific order, dictionaries are *unordered* collections. They are not reliant on sequential numbers (indices), instead, they utilize *keys.*

In the image below we demonstrate a potential use case of a dictionary. Often times dictionaries are used to relate two pieces of data that might not have a direct correlation. In our case, we are connecting the Spanish version of a word to the English version for later lookup.



```
//english numbers
["one","two","three","four","five"];
```

```
List
  0 one
  1 two
  2 three
  3 four
  4 five
@L2 @L1                {5}
```

```
//spanish numbers
["uno","dos","tres","cuatro","cinco"];
```

```
List
  0 uno
  1 dos
  2 tres
  3 cuatro
  4 cinco
@L2 @L1                {5}
```

Build a dictionary to relate the two pieces of data

Get the value given the key

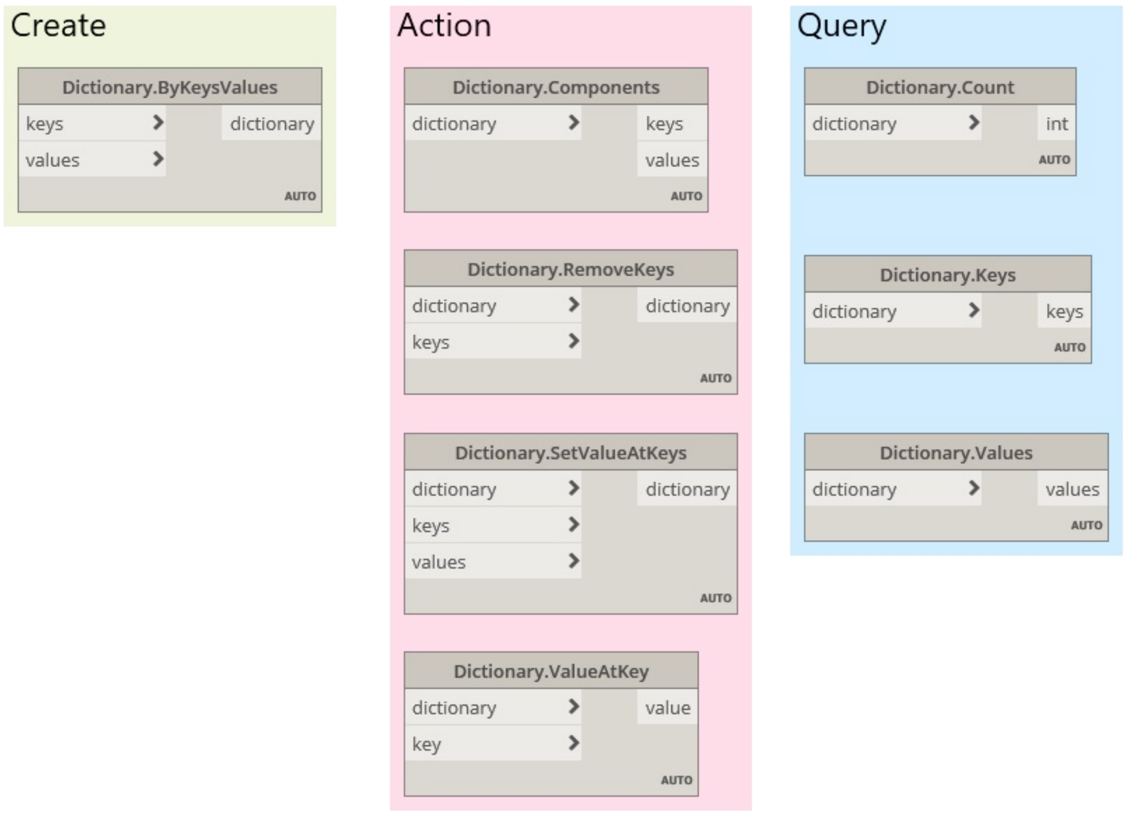
```
"four";
```

```
cuatro
```

Node Uses

Dictionary Nodes

Dynamo 2.0 exposes a variety of Dictionary nodes for our use. This includes *create, action, and query* nodes.



- `Dictionary.ByKeysValues` will create a dictionary with the supplied values and keys. *(The number of entries will be whatever the shortest list input is)*
- `Dictionary.Components` will produce the components of the input dictionary. *(This is the reverse of the create node.)*
- `Dictionary.RemoveKeys` will produce a new dictionary object with the input keys removed.
- `Dictionary.SetValueAtKeys` will produce a new dictionary based on the input keys and the values to replace the current value at the corresponding keys.
- `Dictionary.ValueAtKey` will return the value at the input key.
- `Dictionary.Count` will tell you how many key value pairs are in the dictionary.
- `Dictionary.Keys` will return what keys are currently stored in the dictionary.
- `Dictionary.Values` will return what values are currently stored in the dictionary.

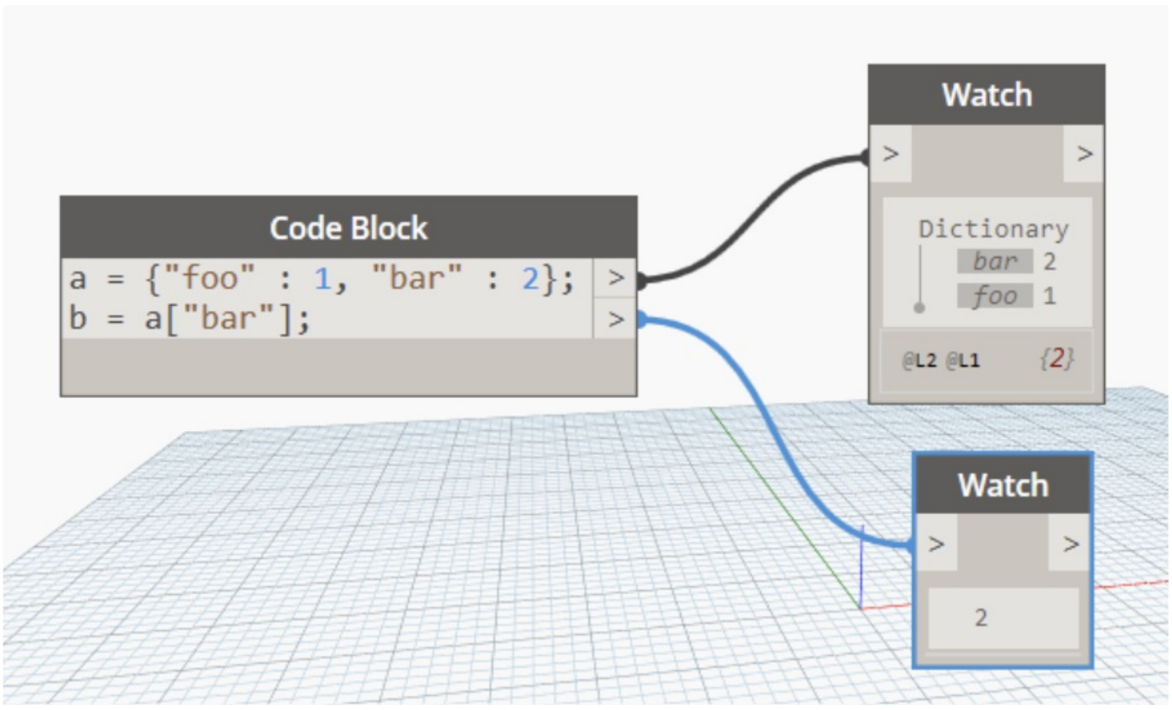
Overall relating data with dictionaries is a magnificent alternative to the old method of working with indices and lists.

Code Block Uses

Dictionaries in Code Blocks

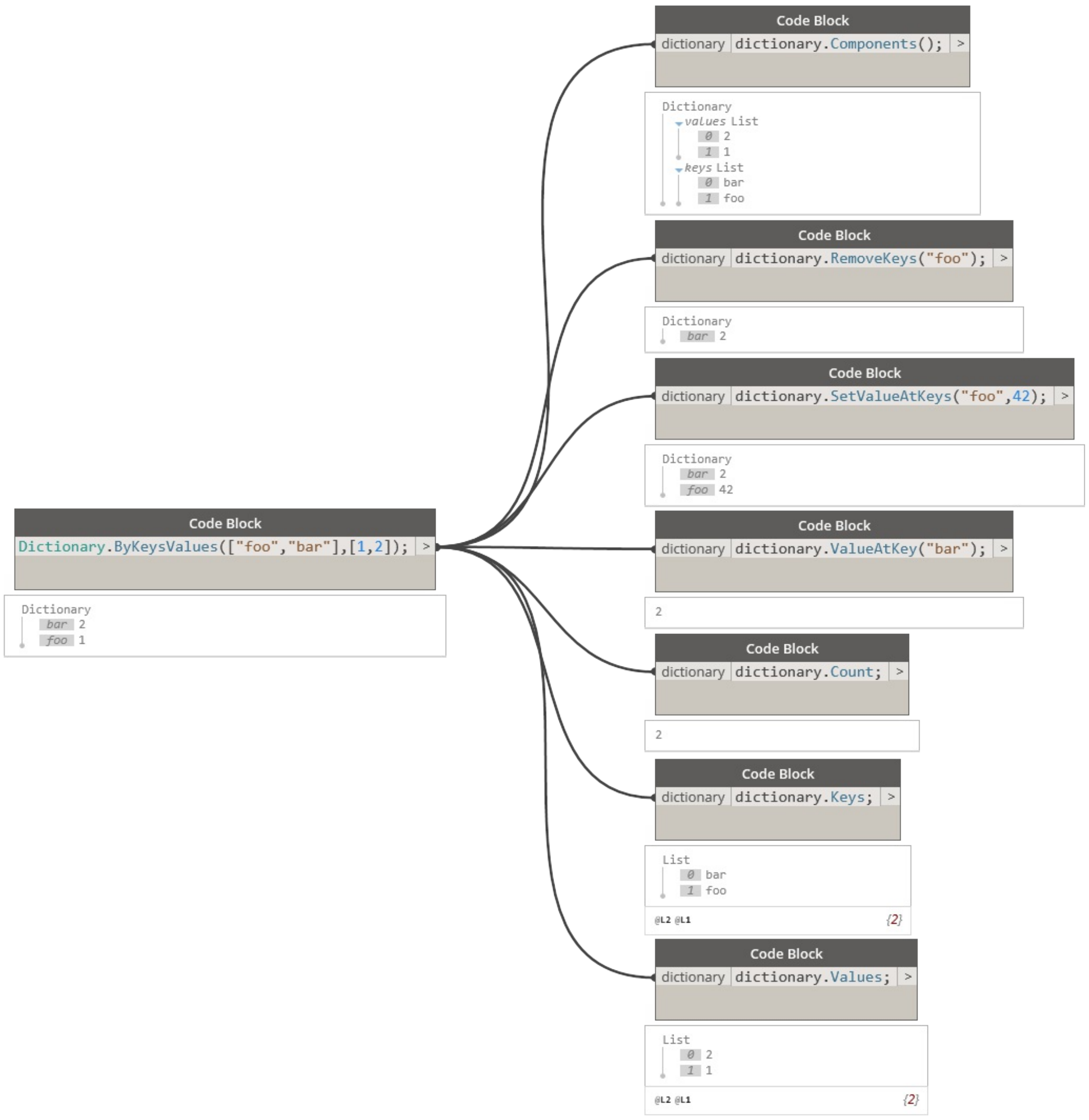
Not only does Dynamo 2.0 introduce the nodes previously discussed for dictionaries, there is new functionality in code blocks for this as well!

You can use syntax like below or DesignScript-based representations of the nodes.



```
a = {"foo" : 1, "bar" : 2};
b = a["bar"];
```

Since a dictionary is an object type in Dynamo we can commit the following actions upon it.


```
Code Block
Dictionary.ByKeysValues(["foo","bar"],[1,2]);
Dictionary
  bar 2
  foo 1

Code Block
dictionary  dictionary.Components();
Dictionary
  values List
    0 2
    1 1
  keys List
    0 bar
    1 foo

Code Block
dictionary  dictionary.RemoveKeys("foo");
Dictionary
  bar 2

Code Block
dictionary  dictionary.SetValueAtKeys("foo",42);
Dictionary
  bar 2
  foo 42

Code Block
dictionary  dictionary.ValueAtKey("bar");
2

Code Block
dictionary  dictionary.Count;
2

Code Block
dictionary  dictionary.Keys;
List
  0 bar
  1 foo
@L2 @L1                {2}

Code Block
dictionary  dictionary.Values;
List
  0 2
  1 1
@L2 @L1                {2}
```

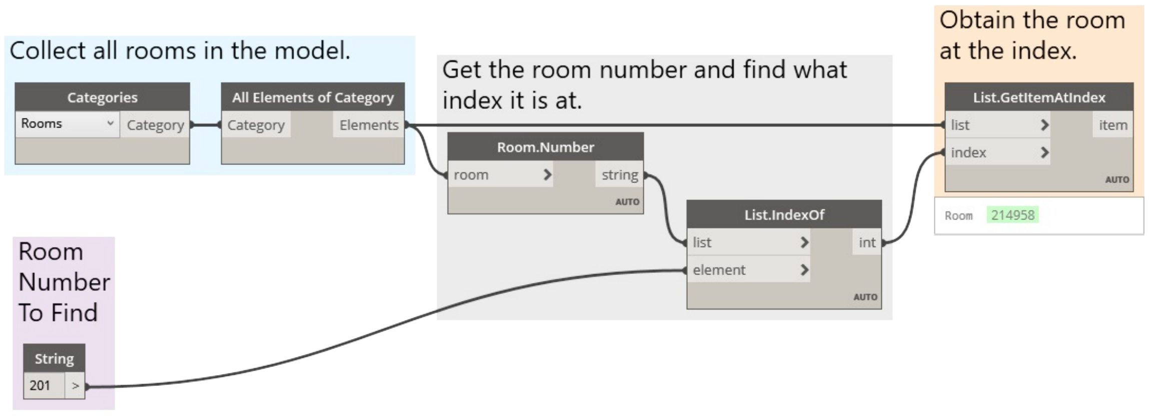
Maintaining these sort of interactions becomes especially useful when relating Revit data to strings. Next, we will look at some Revit use-cases.

Use-Cases

Dictionaries - Revit Use-Cases

Have you ever wanted to look up something in Revit by a piece of data that it has?

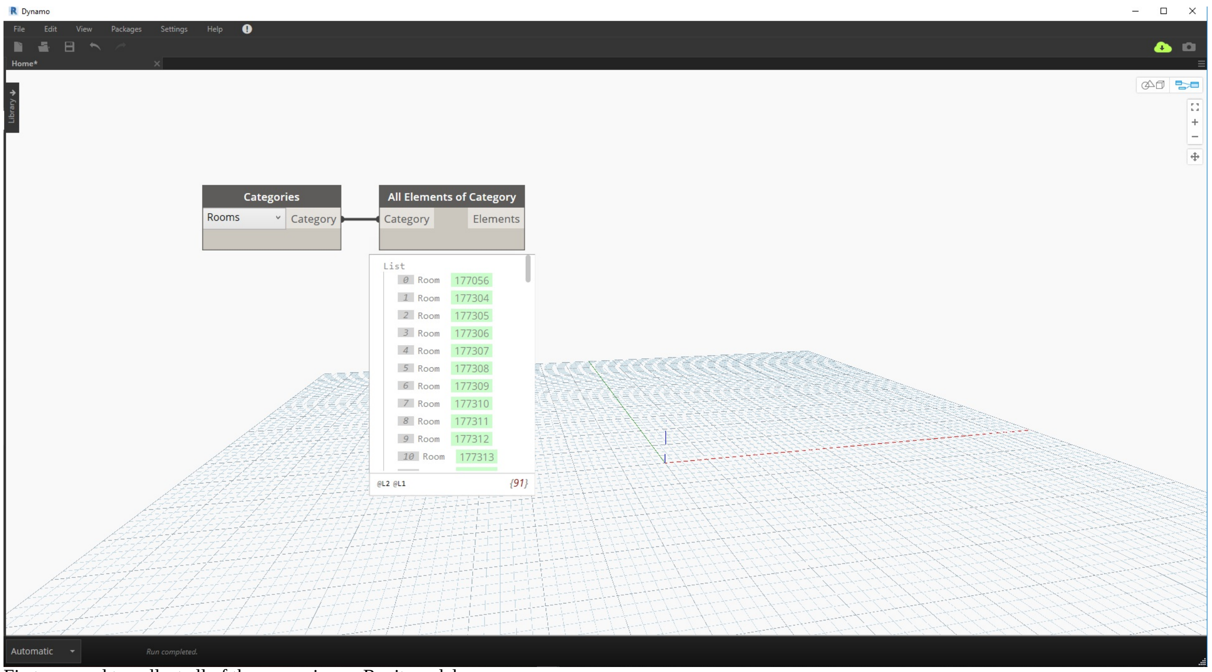
Chances are if you have you've done something like this:



In the image above we are collecting all of the rooms in the Revit model, getting the index of the room we want (by room number), and finally grabbing the room at the index.

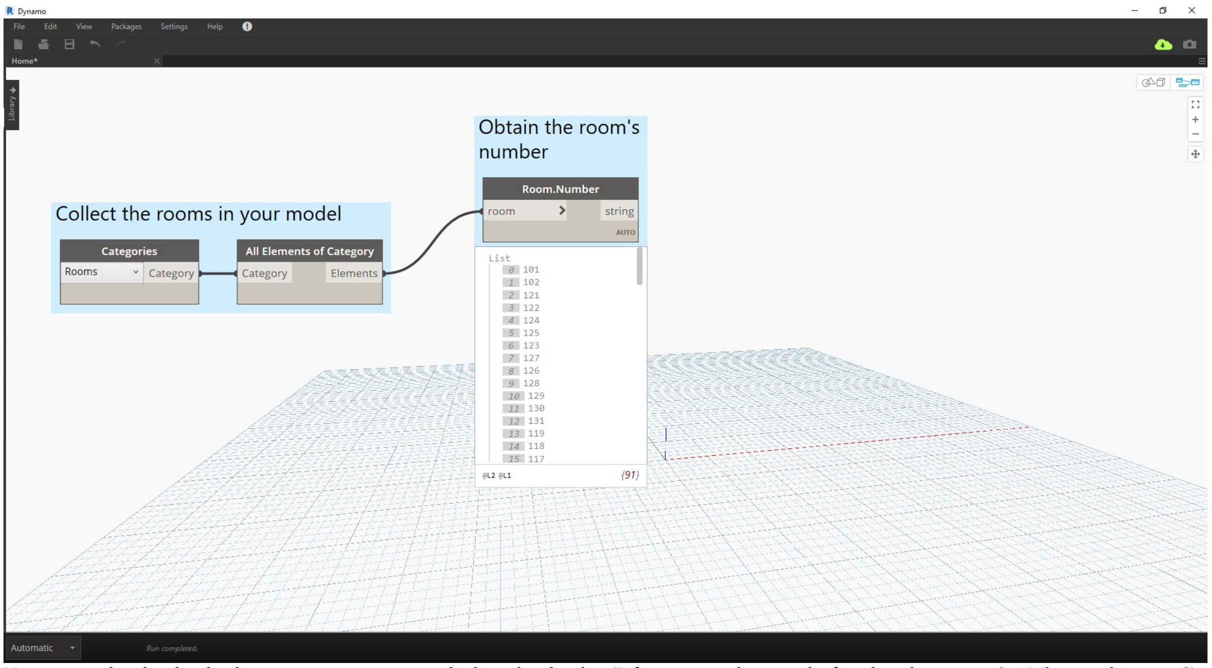
Now let's recreate this idea using dictionaries.

Download the example file that accompanies this exercise (Right click and "Save Link As..."): RoomDictionary.dyn. A full list of example files can be found in the Appendix.



First we need to collect all of the rooms in our Revit model.

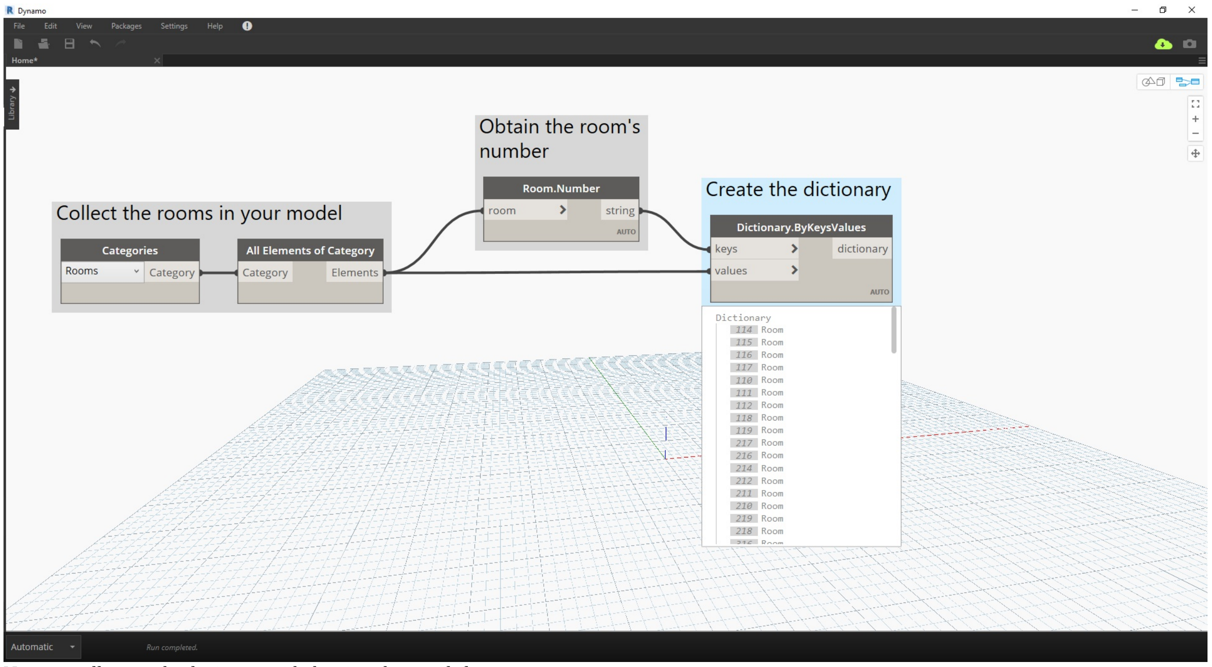
- We choose the Revit category we want to work with, (In this case, we are working with rooms).
- We tell Dynamo to collect all of those elements



```
List
 0 101
 1 102
 2 121
 3 122
 4 124
 5 125
 6 123
 7 127
 8 126
 9 128
 10 129
 11 130
 12 131
 13 119
 14 118
 15 117
@L2 @L1                {91}
```

Next, we need to decide what keys we are going to use to look up this data by. (Information on keys can be found on the section, 9-1 What is a dictionary?).

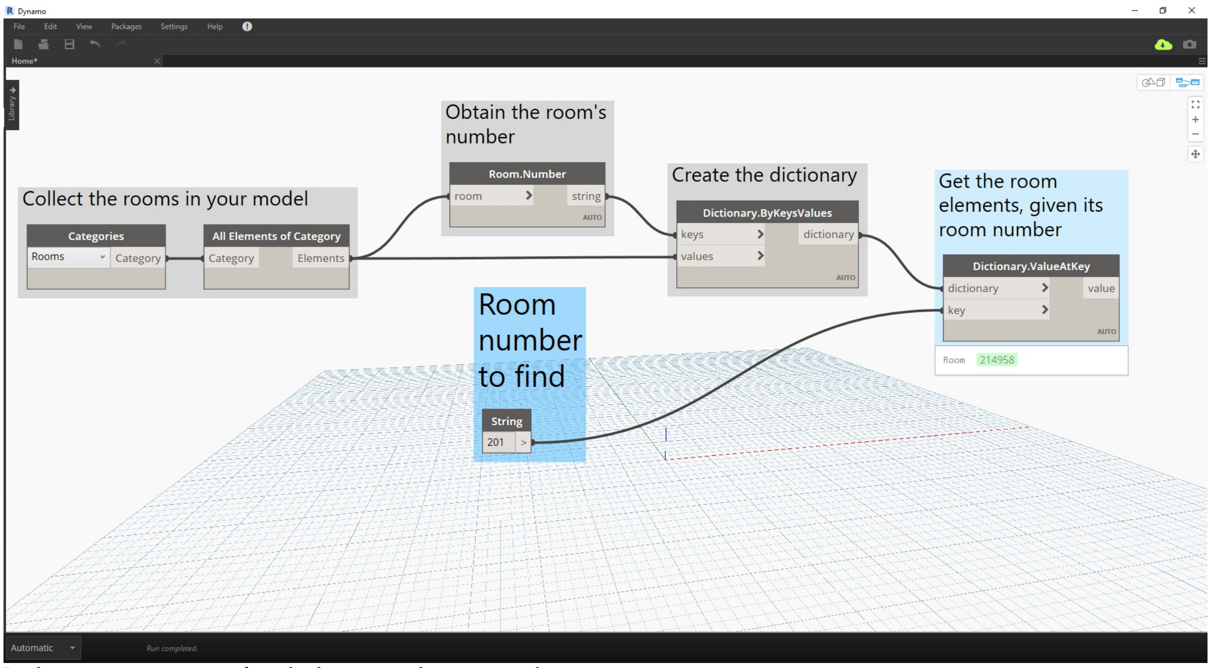
- The data that we will use is the room number.



```
Dictionary
 114 Room
 115 Room
 116 Room
 117 Room
 110 Room
 111 Room
 112 Room
 118 Room
 119 Room
 217 Room
 216 Room
 214 Room
 212 Room
 211 Room
 210 Room
 219 Room
 218 Room
```

Now we will create the dictionary with the given keys and elements.

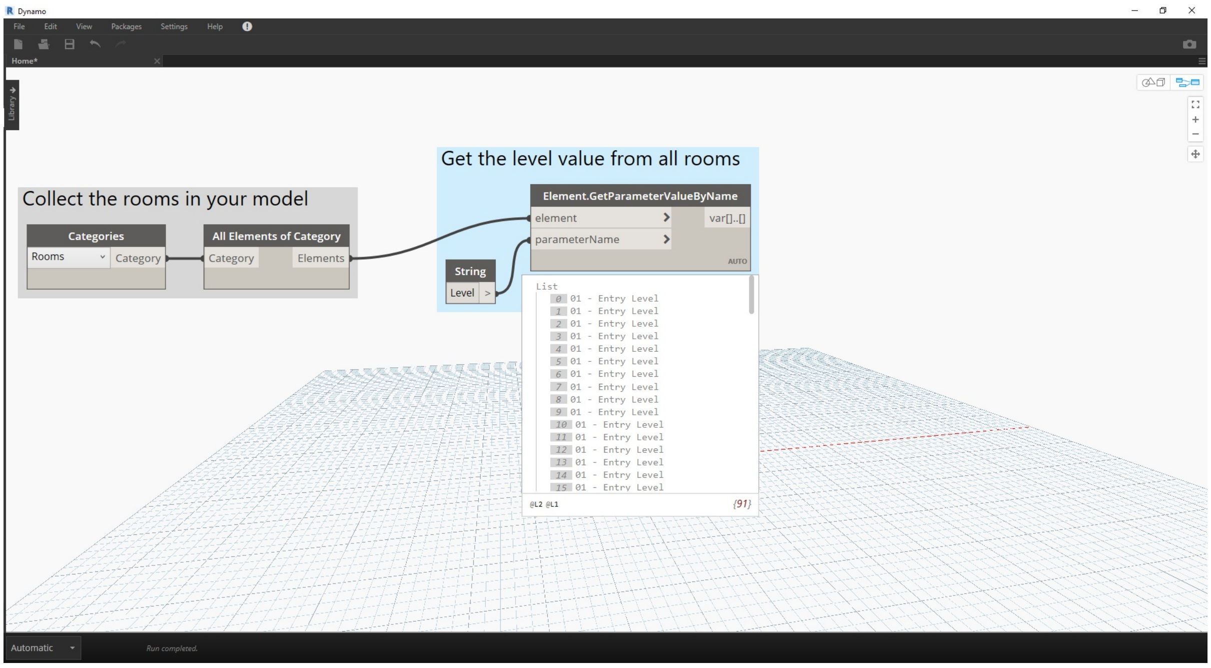
- The node, `Dictionary.ByKeysValues` will create a dictionary given the appropriate inputs.
- `Keys` need to be a string, while `values` can be a variety of object types.



Lastly, we can retrieve a room from the dictionary with its room number now.

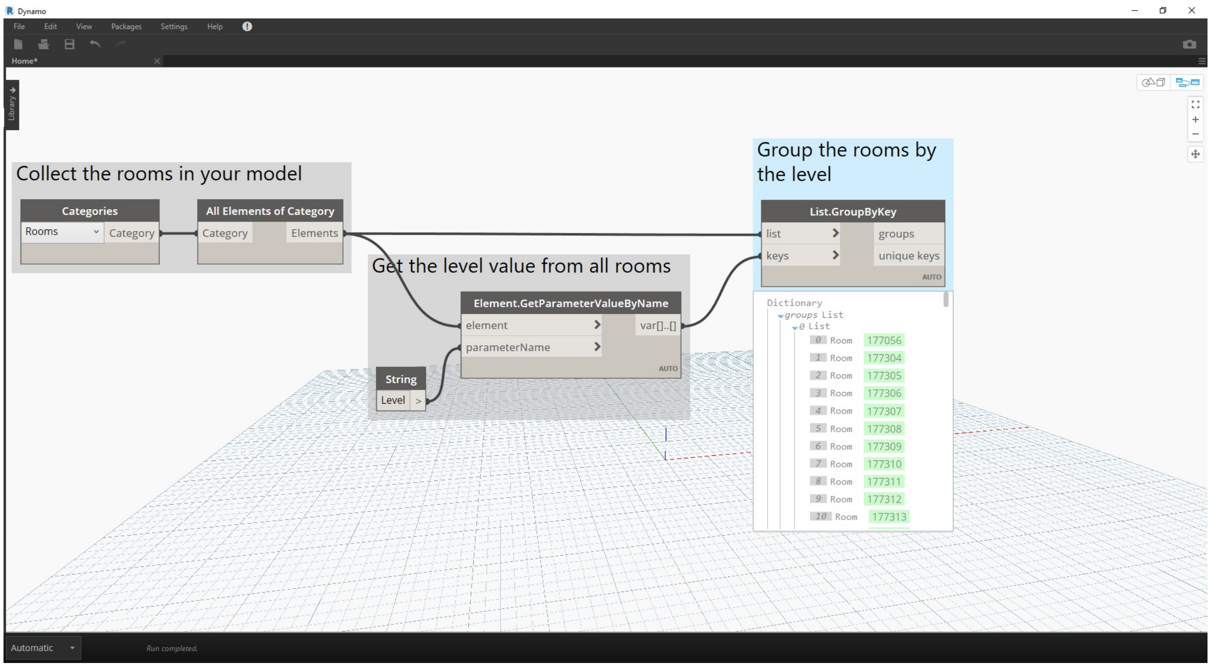
- `String` will be the key that we are using to look up an object from the dictionary.
- `Dictionary.ValueAtKey` will obtain the object from the dictionary now.

Using this same dictionary logic, we can create dictionaries with grouped objects as well. If we wanted to look up all rooms at a given level we can modify the above graph as follows.



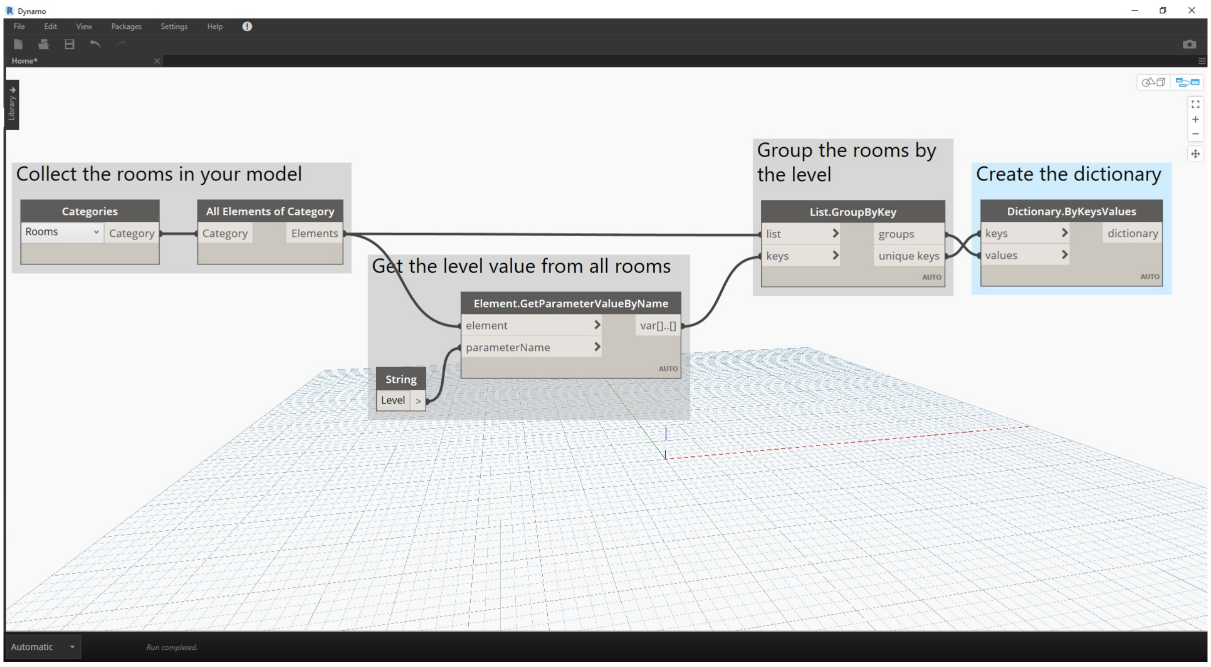
```
List
  0 01 - Entry Level
  1 01 - Entry Level
  2 01 - Entry Level
  3 01 - Entry Level
  4 01 - Entry Level
  5 01 - Entry Level
  6 01 - Entry Level
  7 01 - Entry Level
  8 01 - Entry Level
  9 01 - Entry Level
  10 01 - Entry Level
  11 01 - Entry Level
  12 01 - Entry Level
  13 01 - Entry Level
  14 01 - Entry Level
  15 01 - Entry Level
@L2 @L1                {91}
```

- Rather than using the room number as the key, we can now use a parameter value, (in this case we will use level).

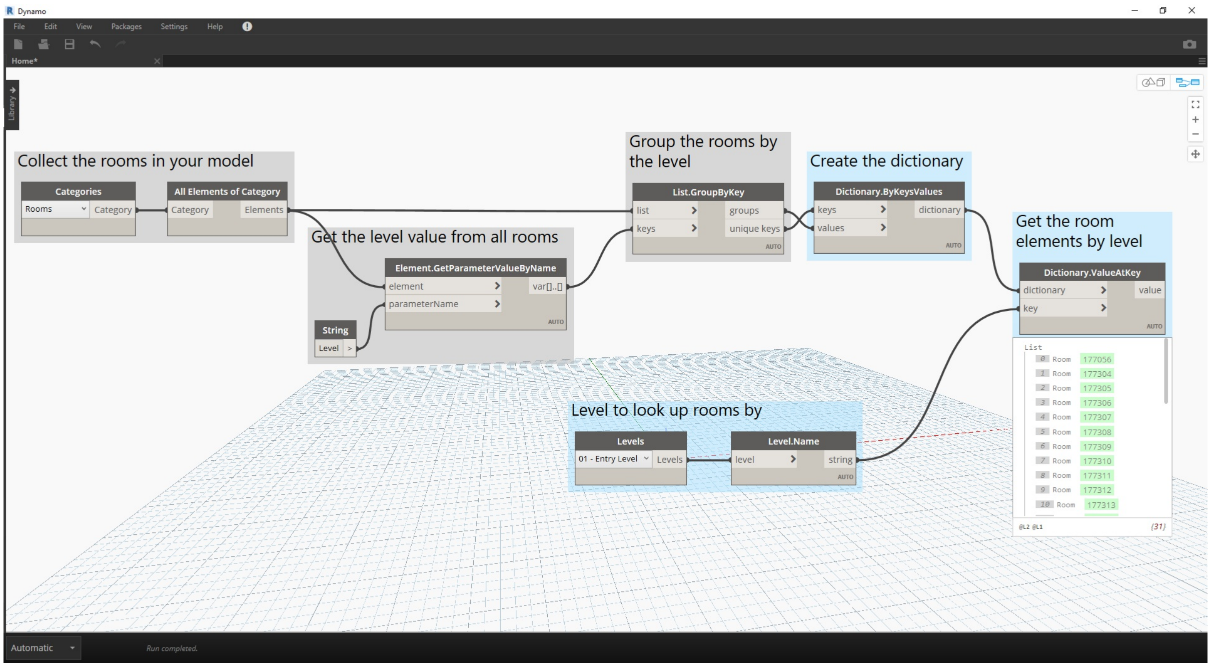


```
Dictionary
  groups List
    0 List
      0 Room 177056
      1 Room 177304
      2 Room 177305
      3 Room 177306
      4 Room 177307
      5 Room 177308
      6 Room 177309
      7 Room 177310
      8 Room 177311
      9 Room 177312
      10 Room 177313
```

- Now, we can group the rooms by the level that they reside on.



- With the elements grouped by the level, we can now use the shared keys (unique keys) as our key for our dictionary, and the lists of rooms as the elements.



```
List
  0 Room 177056
  1 Room 177304
  2 Room 177305
  3 Room 177306
  4 Room 177307
  5 Room 177308
  6 Room 177309
  7 Room 177310
  8 Room 177311
  9 Room 177312
  10 Room 177313
@L2 @L1                {31}
```

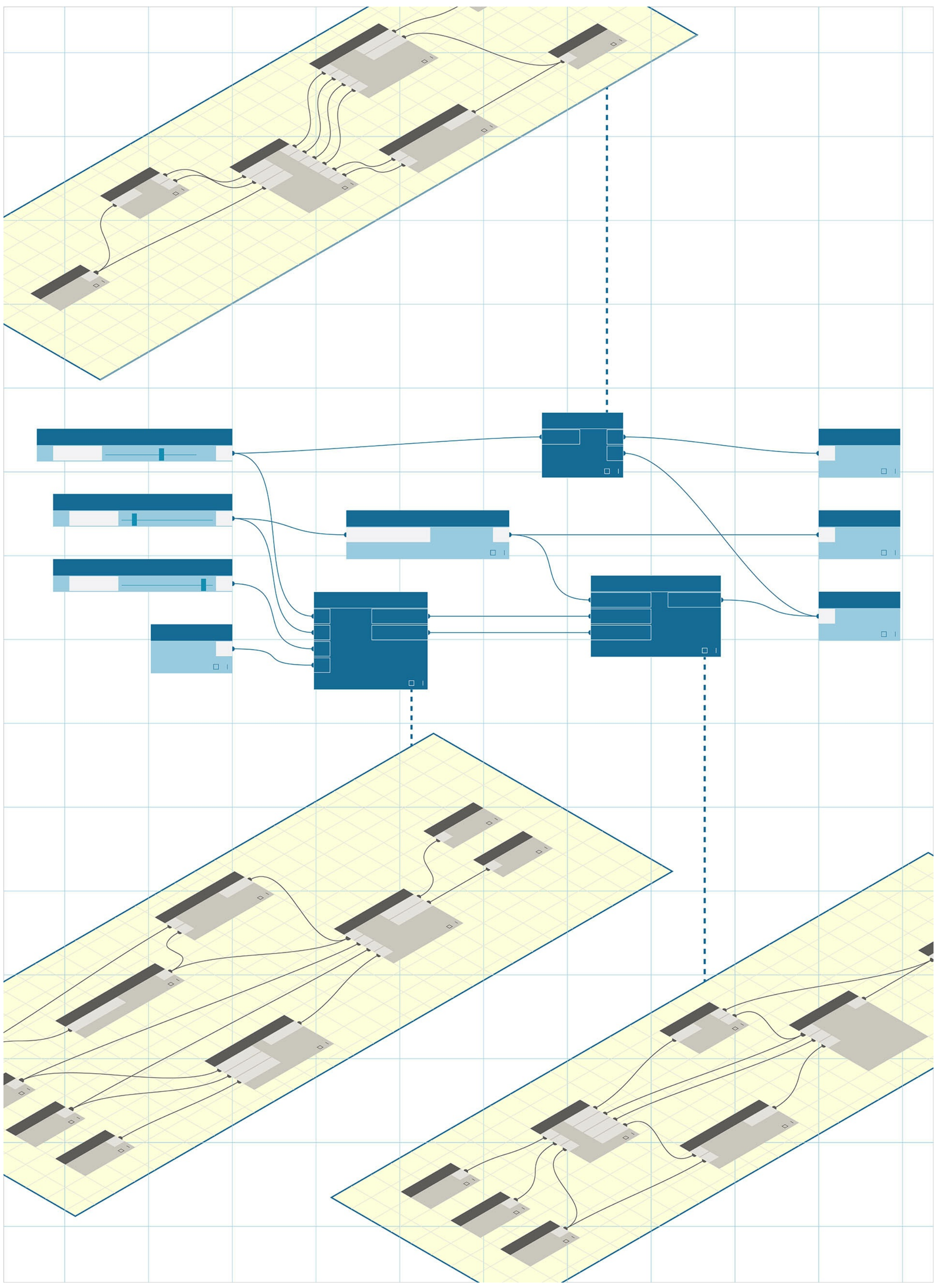
- Lastly, using the levels in the Revit model, we can look up which rooms reside on that level in the dictionary. `Dictionary.ValueAtKey` will take the level name and return the room objects at that level.

The opportunities for Dictionary use are really endless. The ability to relate your BIM data in Revit to the element itself poses a variety of use cases.

Custom Nodes

Custom Nodes

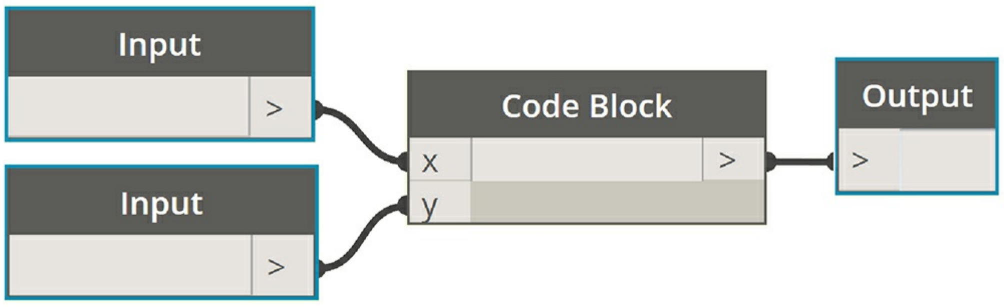
Out of the box, Dynamo has a lot of functionality stored in its Library of Nodes. For those frequently used routines or that special graph you want to share with the community, Custom Nodes are a great way to extend Dynamo even further.



Custom Node Introduction

Custom Nodes

Dynamo offers many core nodes for a wide-range of visual programming tasks. Sometimes a quicker, more elegant, or more easily shared solution is to build your own nodes. These can be reused among different projects, they make your graph clearer and cleaner, and they can be pushed to the package manager and shared with the global Dynamo community.



Cleaning up Your Graph

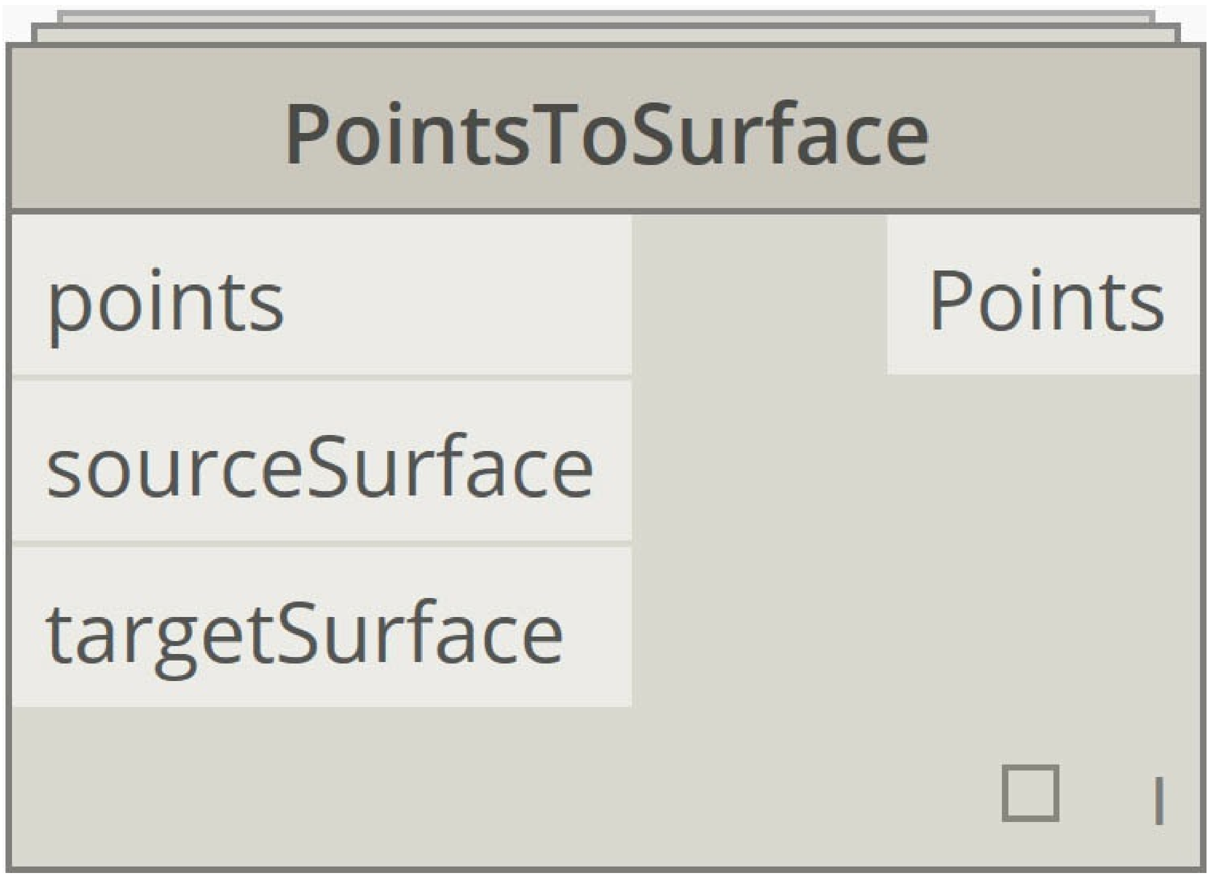
Custom Nodes are constructed by nesting other nodes and custom nodes inside of a "Dynamo Custom Node," which we can think of conceptually as a container. When this container node is executed in your graph, everything inside it will be executed to allow you to reuse and share a useful combination of nodes.

Adapting to Change

When you have multiple copies of a custom node in your graph, you can update all of them by editing the base custom node. This allows you to update your graph seamlessly by adapting to any changes that may occur in workflow or design.

Work Sharing

Arguably the best feature of custom nodes is their work sharing capabilities. If a "power user" creates a complex Dynamo graph and hands it off to a designer who is new to Dynamo, he/she can condense the graph to the bare essentials for design interaction. The custom node can be opened to edit the internal graph, but the "container" can be kept simple. With this process, custom nodes allow Dynamo users to design a graph that is clean and intuitive.

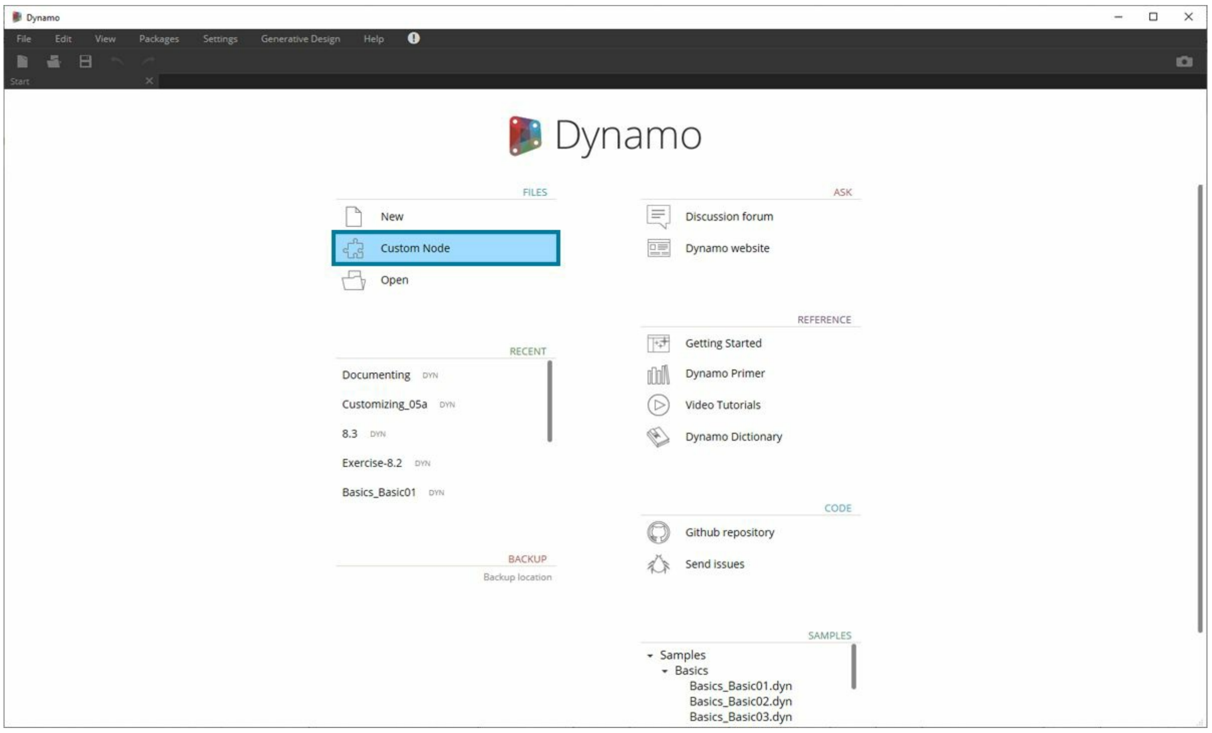


Many Ways to Build a Node

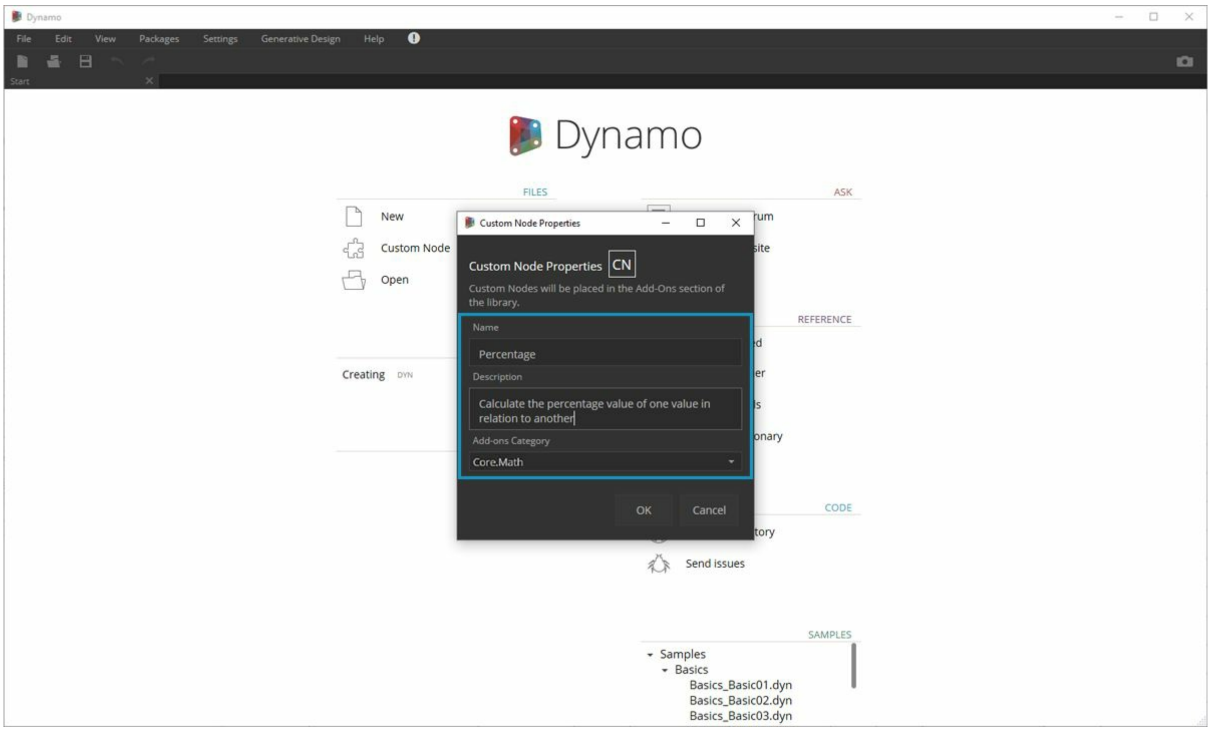
There are a wide variety of ways to build custom nodes in Dynamo. In the examples in this chapter, we'll create custom nodes directly from the Dynamo UI. If you are a programmer and you are interested in C# or Zero-Touch formatting, you can reference this page on the Dynamo Wiki for a more in-depth review.

Custom Node Environment

Let's jump into the custom node environment and make a simple node to calculate a percentage. The custom node environment is different from the Dynamo graph environment, but the interaction is fundamentally the same. With that said, let's create our first custom node!



To create a Custom Node from scratch, Launch Dynamo and select Custom Node, or type Ctrl + Shift + N from the canvas.



Custom Node Properties CN

Custom Nodes will be placed in the Add-Ons section of the library.

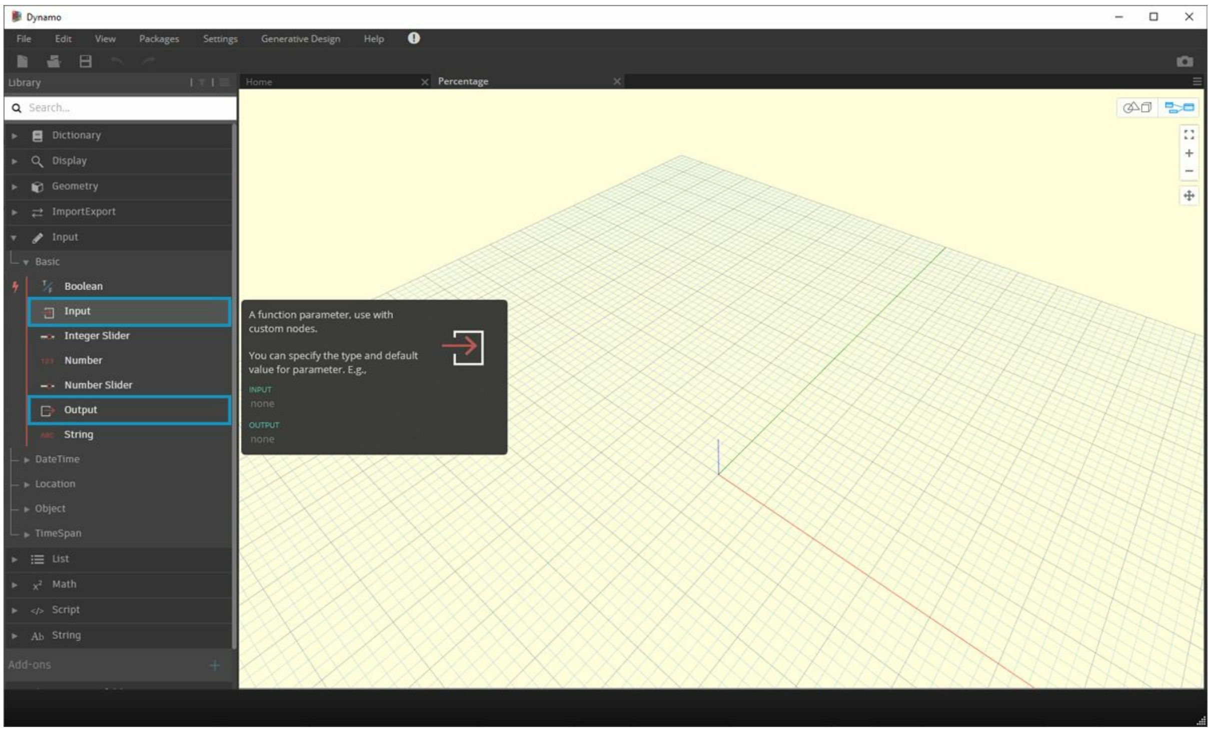
Name: Percentage

Description: Calculate the percentage value of one value in relation to another

Add-ons Category: Core.Math

Assign a name, description, and category in the Custom Node Properties dialog.

1. **Name:** Percentage
2. **Description**: Calculate the percentage of one value in relation to another.
3. **Category:** Core.Math



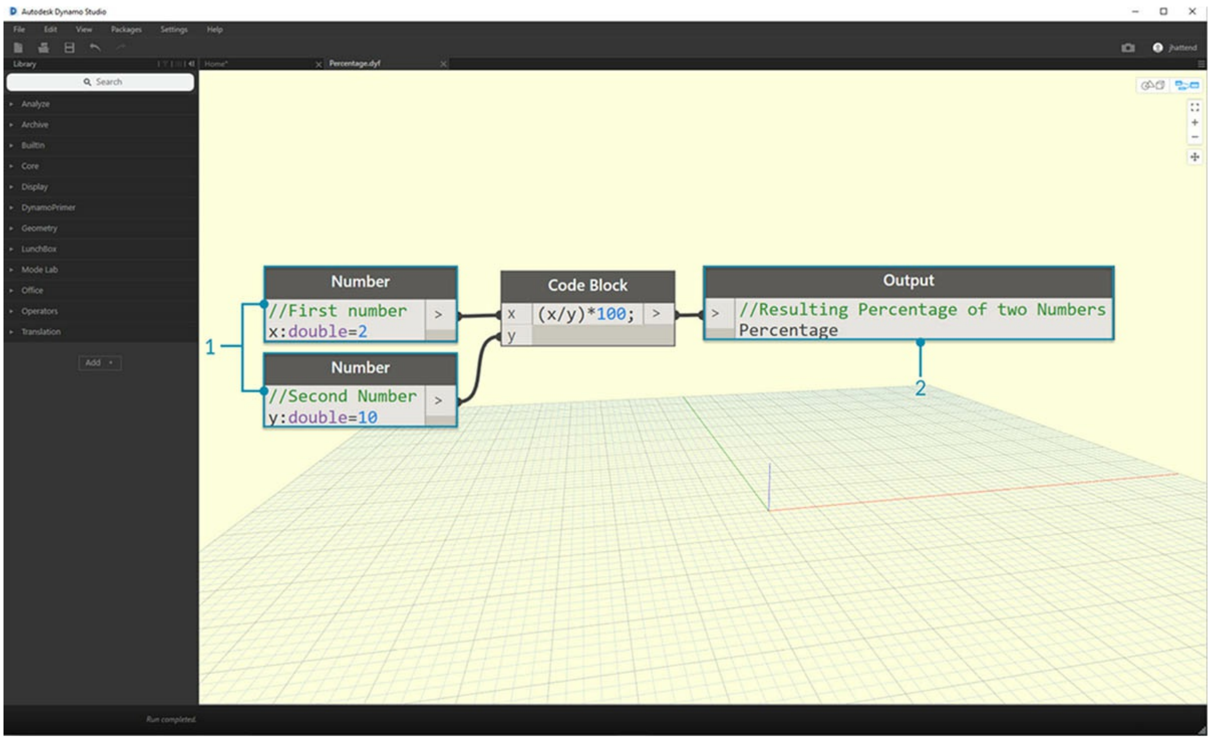
A function parameter, use with custom nodes.

You can specify the type and default value for parameter. E.g.,

INPUT
none

OUTPUT
none

This will open a canvas with a yellow background, indicating that you are working inside a custom node. In this canvas you have access to all of the core Dynamo nodes, as well as the **Input** and **Output** nodes, which label the data flowing into and out of the custom node. They can be found in *Core>Input*.



```
//First number
x:double=2
```

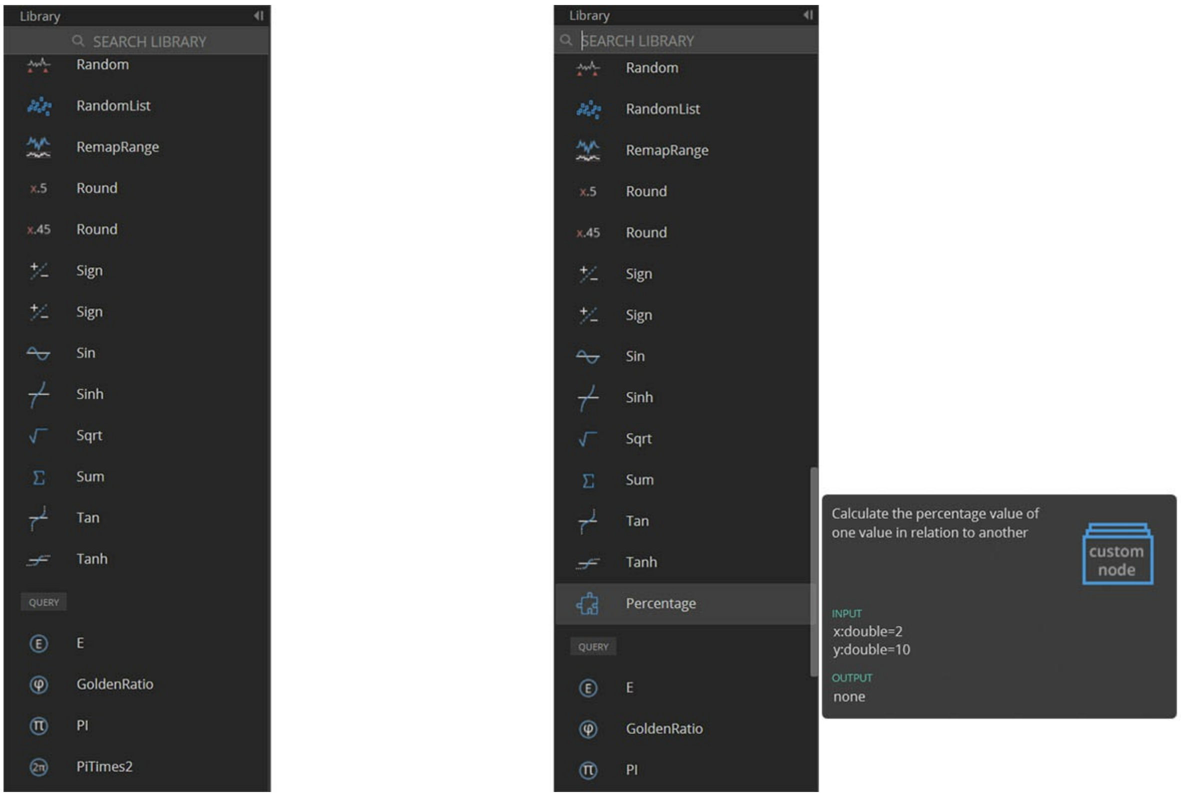
```
//Second Number
y:double=10
```

```
(x/y)*100;
```

```
//Resulting Percentage of two Numbers
Percentage
```

1. **Inputs:** input nodes create input ports on the custom node. The syntax for an input node is *input_name : datatype = default_value(optional).*
2. **Outputs:** Similar to inputs, these will create and name output ports on the custom node. Consider adding a **Custom Comment** to your Input and Output ports to hint at the Input and Output types. This is discussed in more detail in the Creating Custom Nodes section.

You can save this custom node as a .dyf (as opposed to the standard .dyn) file and it will automatically be added to your session and future sessions. You will find the custom node in your library in the category that is specified in the custom node's properties.



Calculate the percentage value of one value in relation to another

INPUT
x:double=2
y:double=10

OUTPUT
none

Left: The Core > Math category of the default library Right: Core > Math with the new custom node

Moving Forward

Now that we've created our first custom node, the next sections will dive deeper into custom node functionality and how to publish generic workflows. In the following section, we'll look at developing a custom node that transfers geometry from one surface to another.

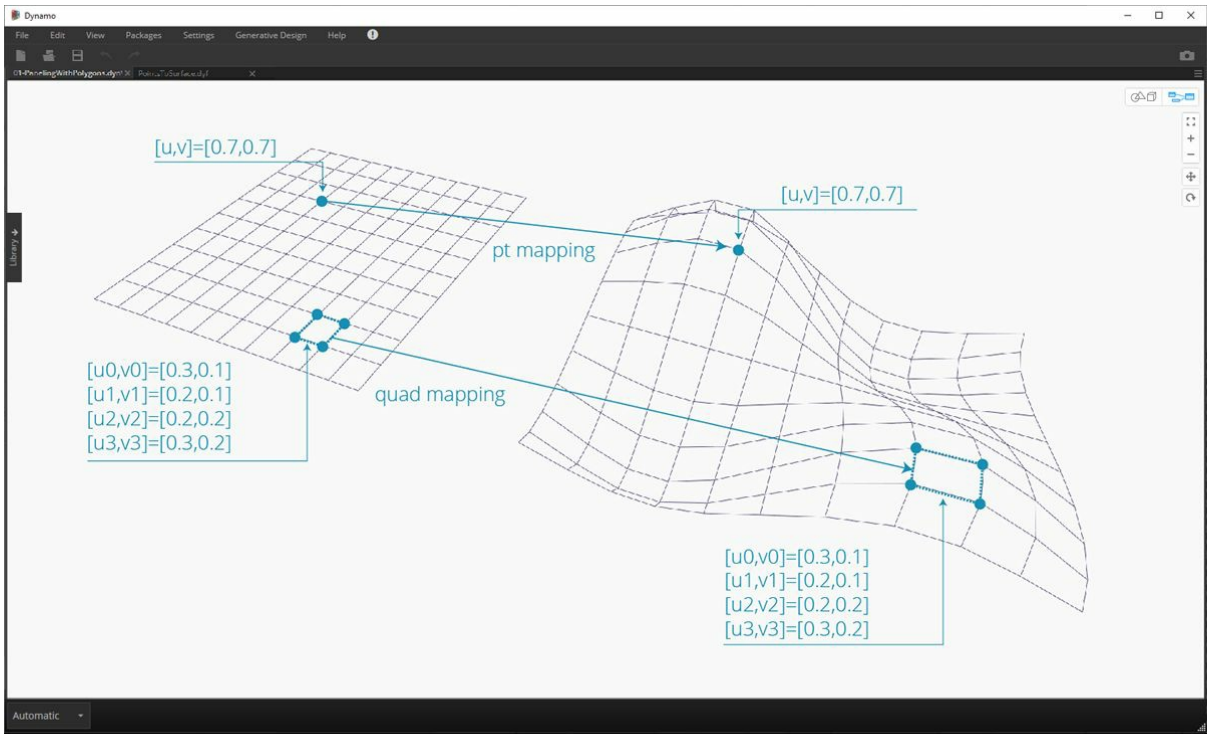
Creating a Custom Node

Creating a Custom Node

Dynamo offers several different methods for creating custom nodes. You can build custom nodes from scratch, from an existing graph, or explicitly in C#. In this section we will cover building a custom node in the Dynamo UI from an existing graph. This method is ideal for cleaning up the workspace, as well as packaging a sequence of nodes to reuse elsewhere.

Custom Nodes for UV Mapping

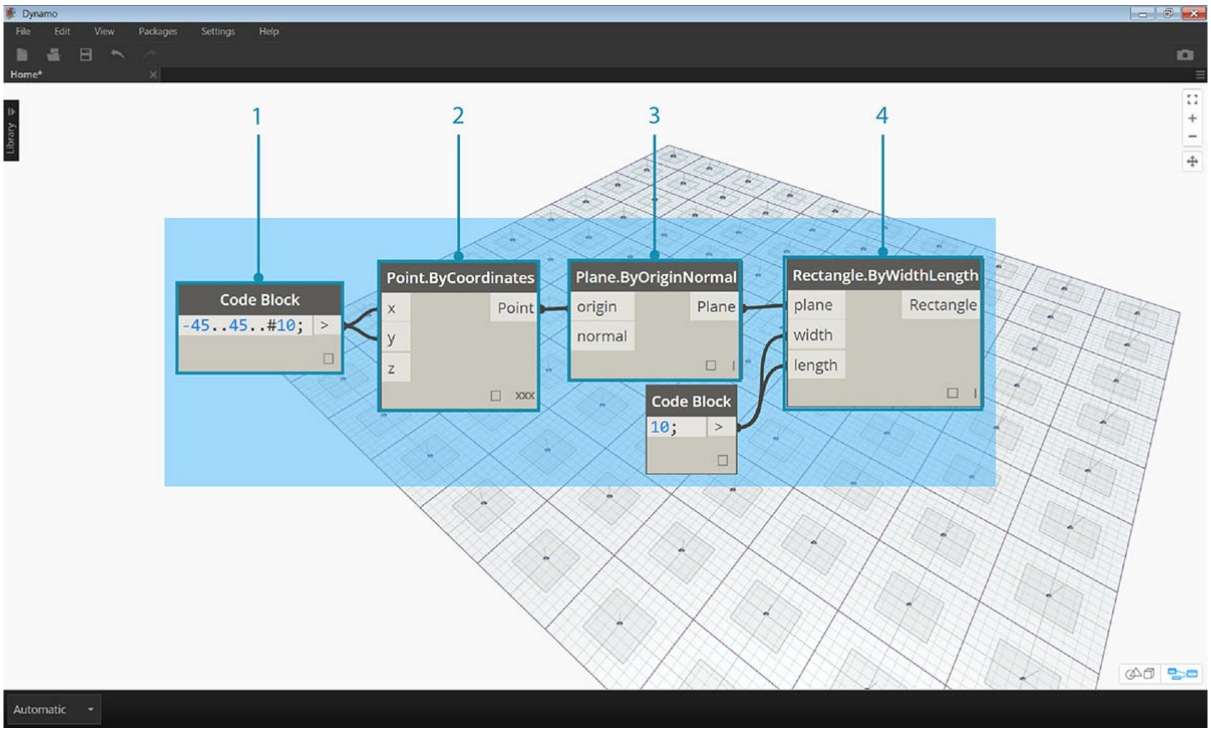
In the image below, we map a point from one surface to another using UV coordinates. We'll use this concept to create a panelized surface which references curves in the XY plane. We'll create quad panels for our panelization here, but using the same logic, we can create a wide variety of panels with UV mapping. This is a great opportunity for custom node development because we will be able to repeat a similar process more easily in this graph or in other Dynamo workflows.



Creating a Custom Node from an Existing Graph

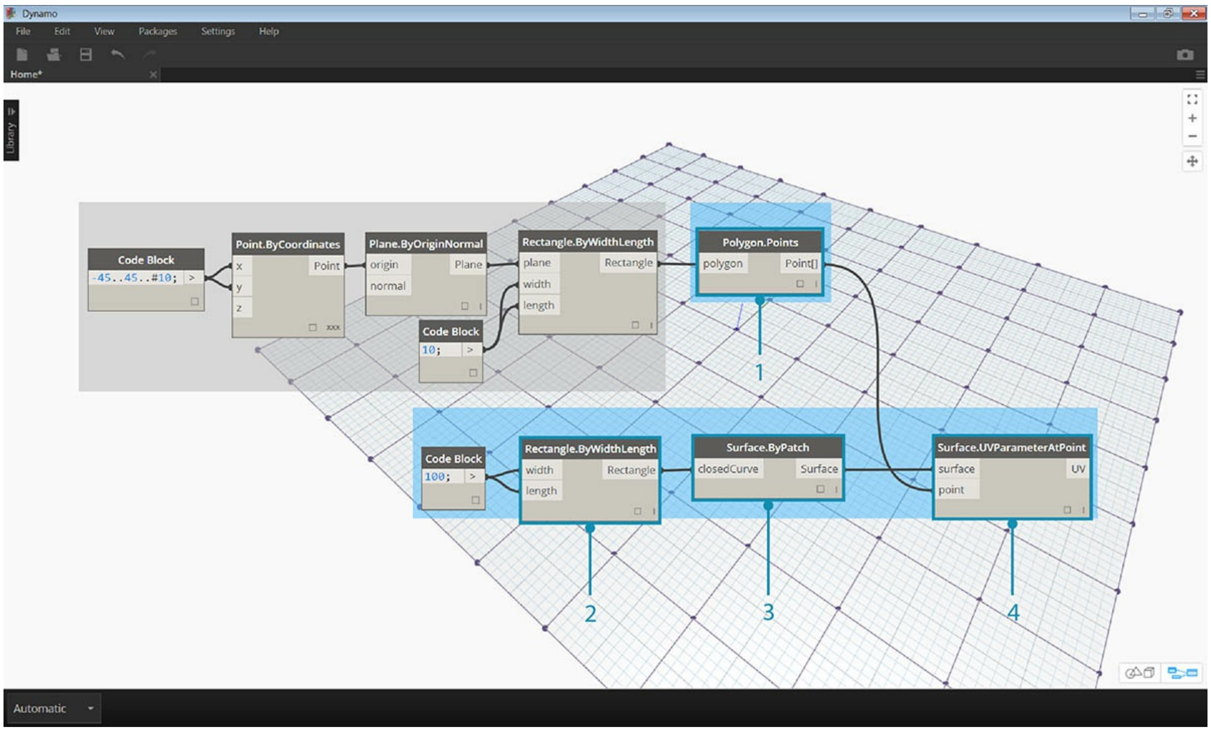
> Download and unzip the example files for this exercise (Right click and choose "Save Link As..."). A full list of example files can be found in the Appendix. UV-CustomNode.zip

Let’s start by creating a graph that we want to nest into a custom node. In this example, we will create a graph that maps polygons from a base surface to a target surface, using UV coordinates. This UV mapping process is something we use frequently, making it a good candidate for a custom node. For more information on surfaces and UV space, see section 5.5. The complete graph is *UVmapping_Custom-Node.dyn* from the .zip file downloaded above.



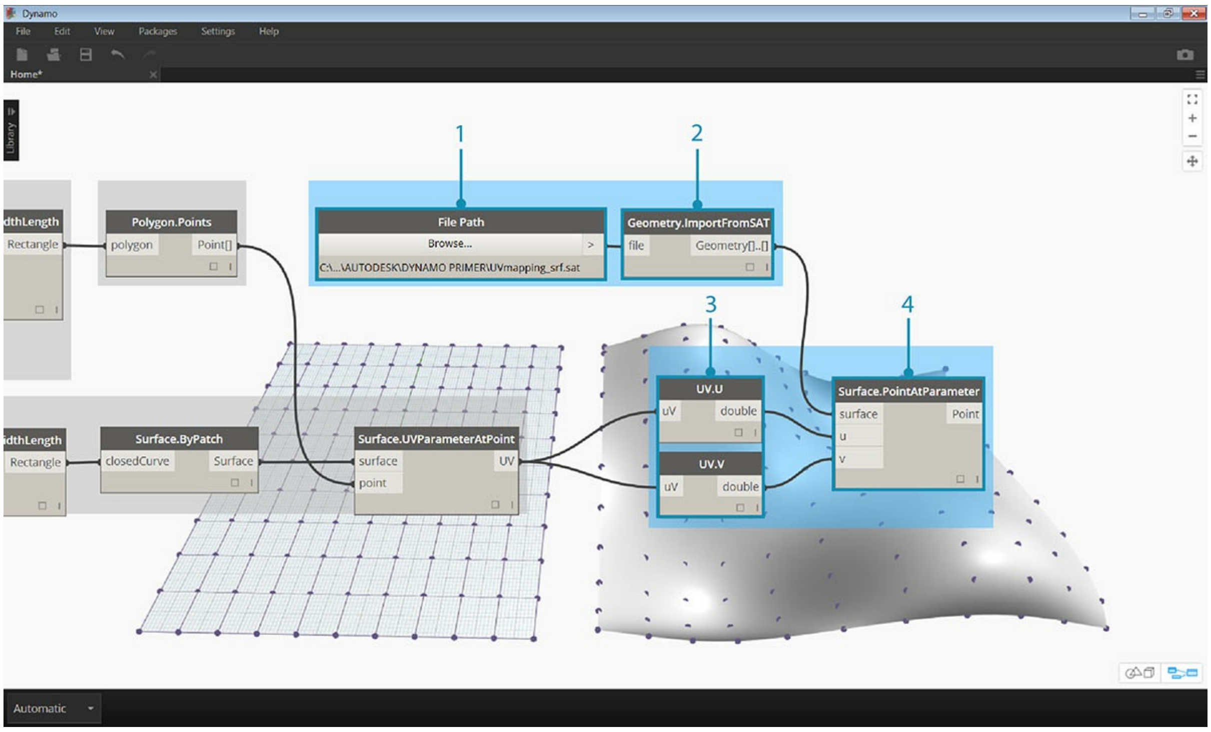
1. **Code Block:** Create a range of 10 numbers between 45 and negative 45 using a code block.
2. **Point.ByCoordinates:** Connect the output of the Code Block to the ‘x’ and ‘y’ inputs and set the lacing to cross-reference. You should now have a grid of points.
3. **Plane.ByOriginNormal:** Connect the *‘Point’* output to the *‘origin’* input to create a plane at each of the points. The default normal vector of (0,0,1) will be used.
4. **Rectangle.ByWidthLength:** Connect the planes from the previous step into the *‘plane’* input, and use a Code Block with a value of *10* to specify the width and length.

You should now see a grid of rectangles. Let’s map these rectangles to a target surface using UV coordinates.



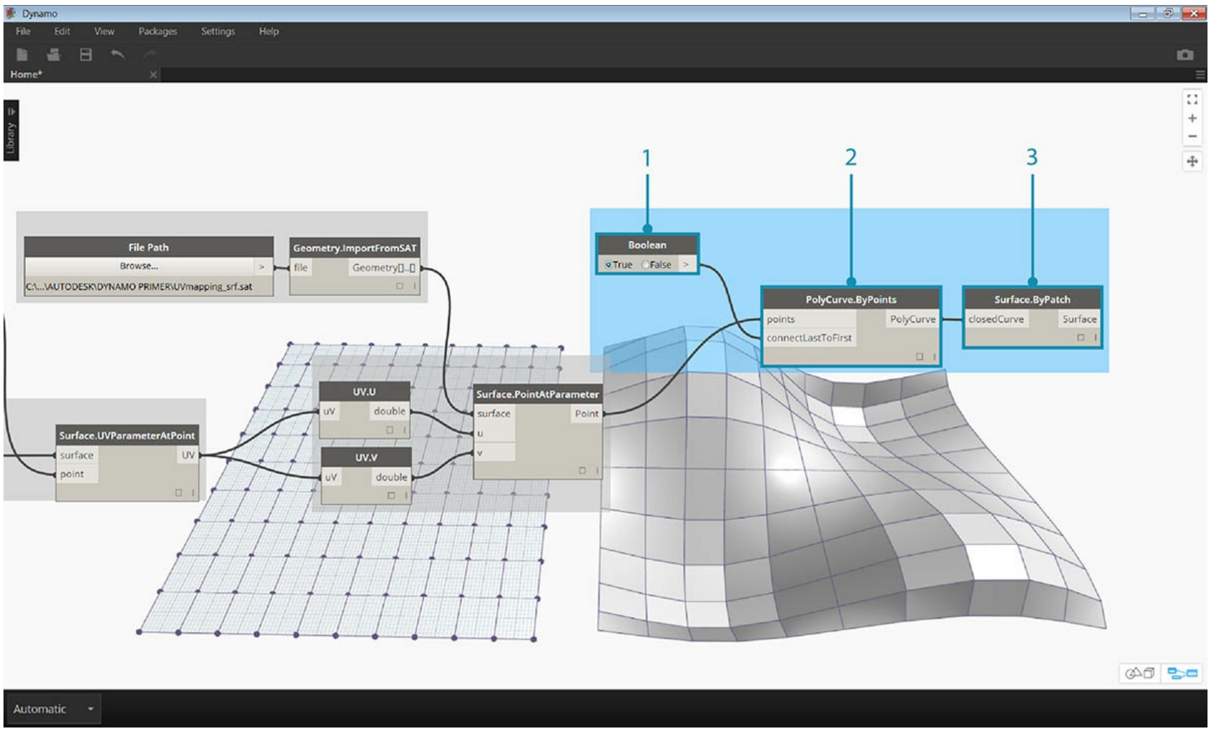
1. **Polygon.Points:** Connect the Rectangle output from the previous step to the *‘polygon’* input to extract the corner points of each rectangle. These are the points that we will map to the target surface.
2. **Rectangle.ByWidthLength:** Use a Code Block with a value of *100* to specify the width and length of a rectangle. This will be the boundary of our base surface.
3. **Surface.ByPatch:** Connect the Rectangle from the previous step to the *‘closedCurve’* input to create a base surface.
4. **Surface.UVParameterAtPoint:** Connect the *‘Point’* output of the *Polygon.Points* node and the *‘Surface’* output of the *Surface.ByPatch* node to return the UV parameter at each point.

Now that we have a base surface and a set of UV coordinates, we can import a target surface and map the points between surfaces.



1. **File Path:** Select the file path of the surface you want to import. The file type should be .SAT. Click the *"Browse..."* button and navigate to the *UVmapping_srf.sat* file from the .zip file downloaded above.
2. **Geometry.ImportFromSAT:** Connect the file path to import the surface. You should see the imported surface in the geometry preview.
3. **UV:** Connect the UV parameter output to a *UV.U* and a *UV.V* node.
4. **Surface.PointAtParameter:** Connect the imported surface as well as the u and v coordinates. You should now see a grid of 3D points on the target surface.

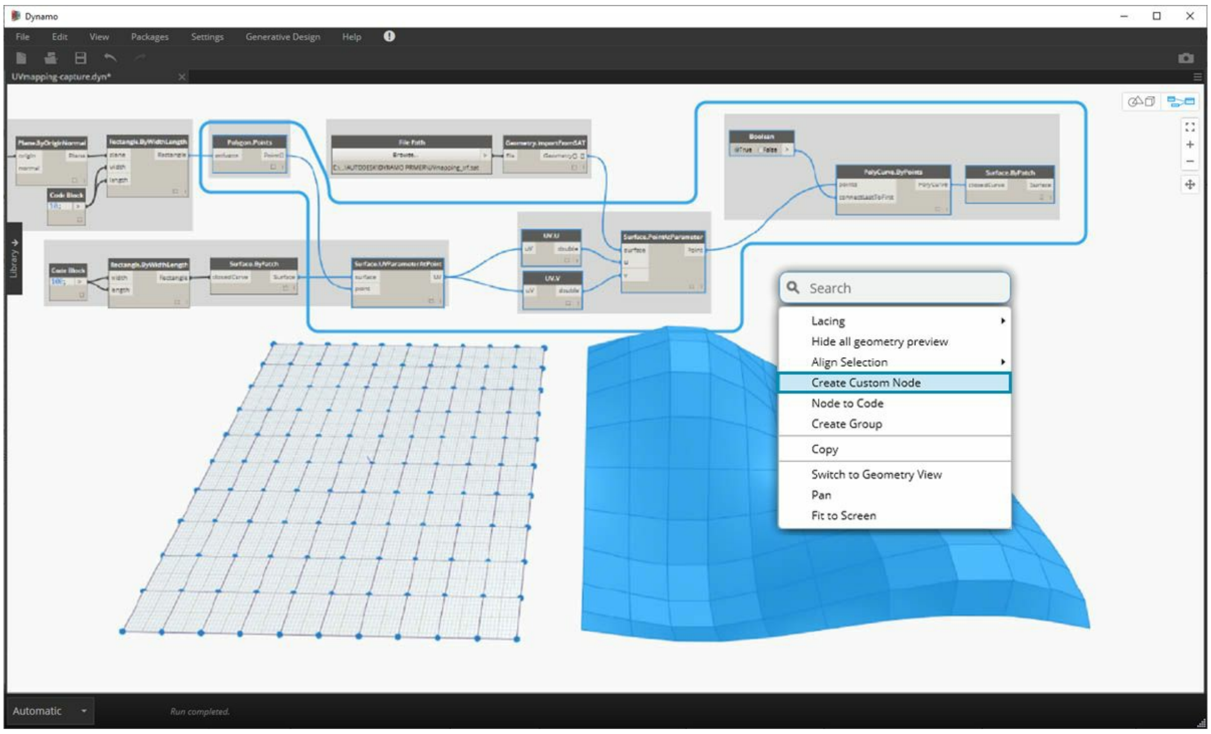
The final step is to use the 3D points to construct rectangular surface patches.



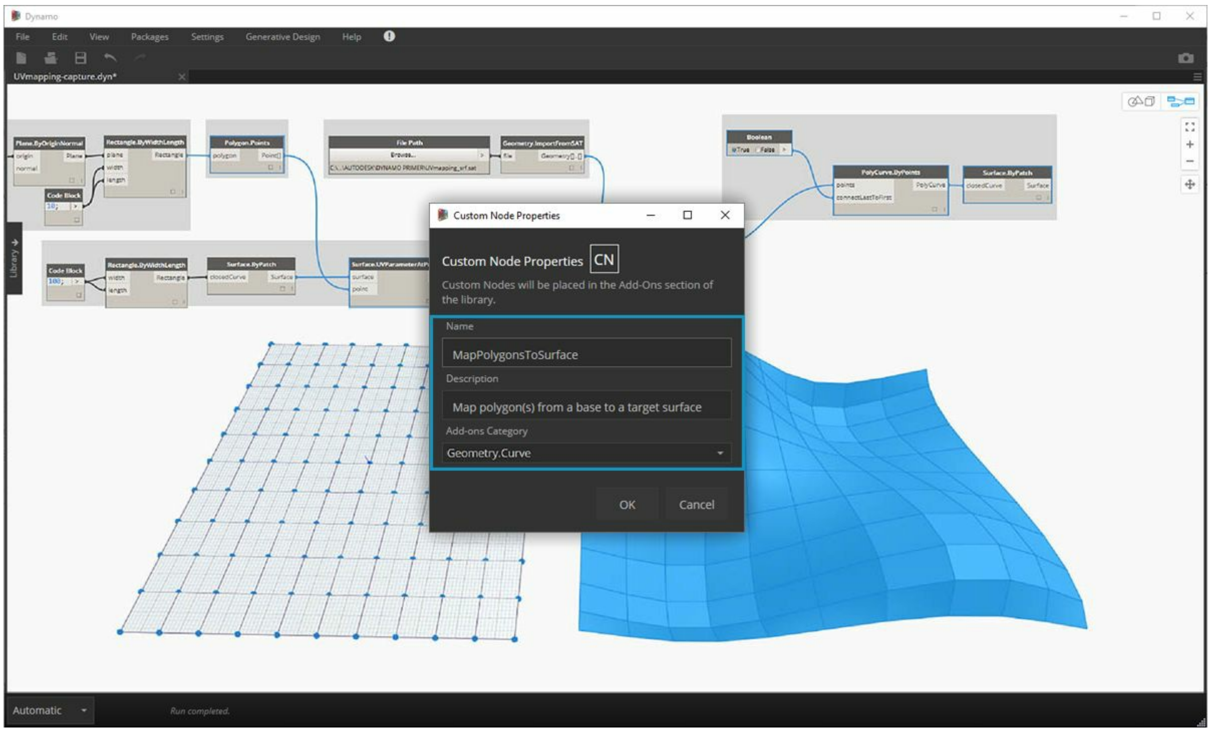
1. **PolyCurve.ByPoints:** Connect the points on the surface to construct a polycurve through the points.
2. **Boolean:** Add a Boolean to the workspace and connect it to the *'connectLastToFirst'* input and toggle to True to close the polycurves. You should now see rectangles mapped to the surface.
3. **Surface.ByPatch:** Connect the polycurves to the *'closedCurve'* input to construct surface patches.

Now let's select the nodes that we want to nest into a Custom Node, thinking about what we want to be the inputs and outputs of our node. We want our

Custom Node to be as flexible as possible, so it should be able to map any polygons, not just rectangles.



Select the above nodes (beginning with *Polygon.Points*), right click on the workspace and select *'node from selection'*.


Custom Node Properties CN

Custom Nodes will be placed in the Add-Ons section of the library.

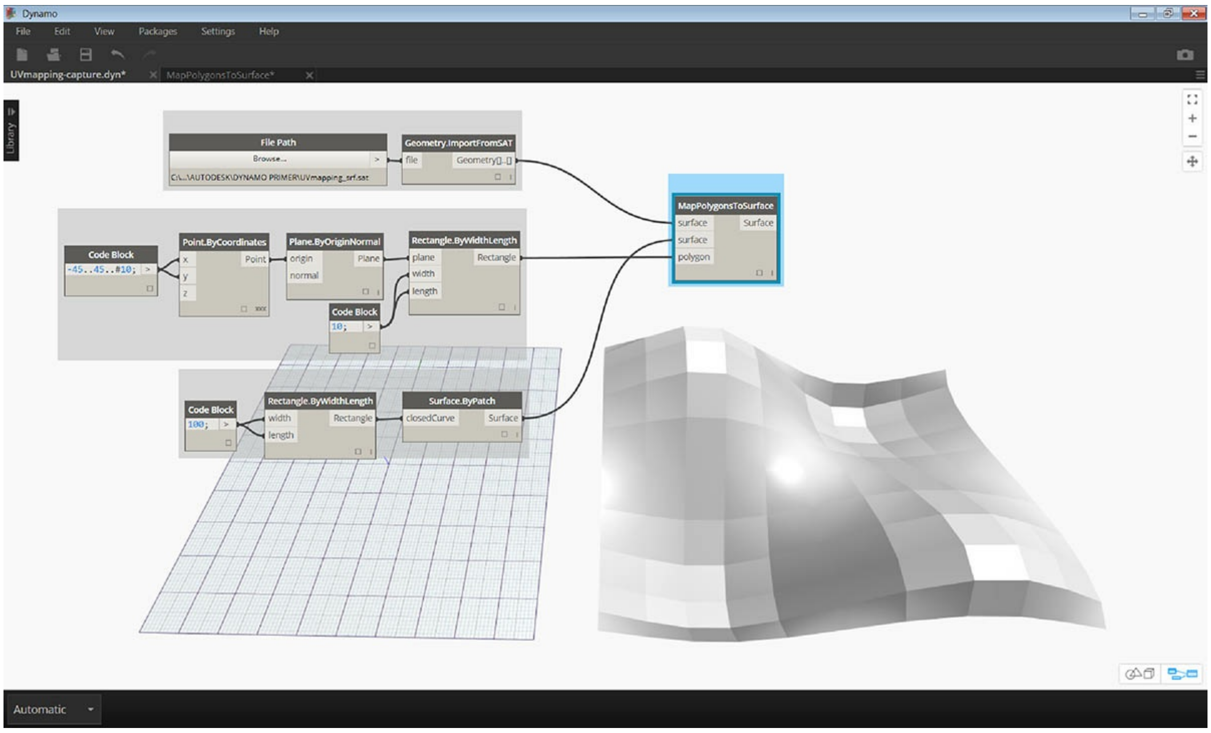
Name: MapPolygonsToSurface

Description: Map polygon(s) from a base to a target surface

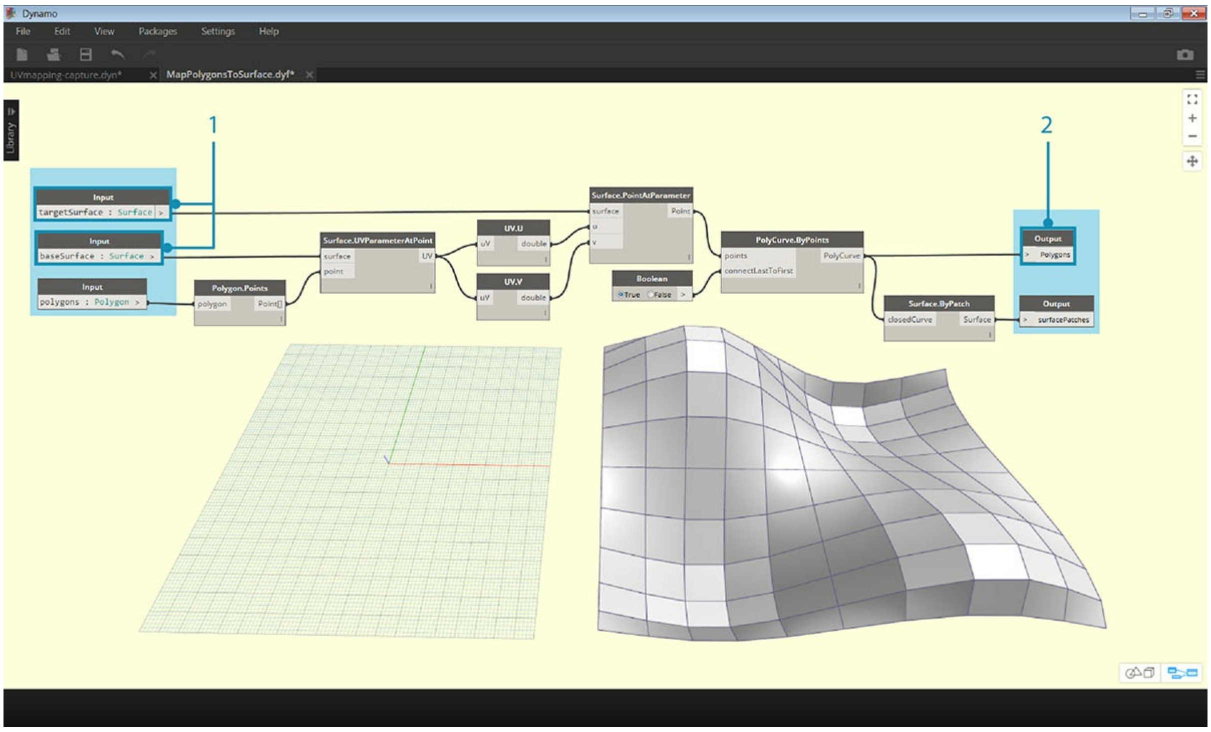
Add-ons Category: Geometry.Curve

OK Cancel

In the Custom Node Properties dialog, assign a name, description, and category to the Custom Node.



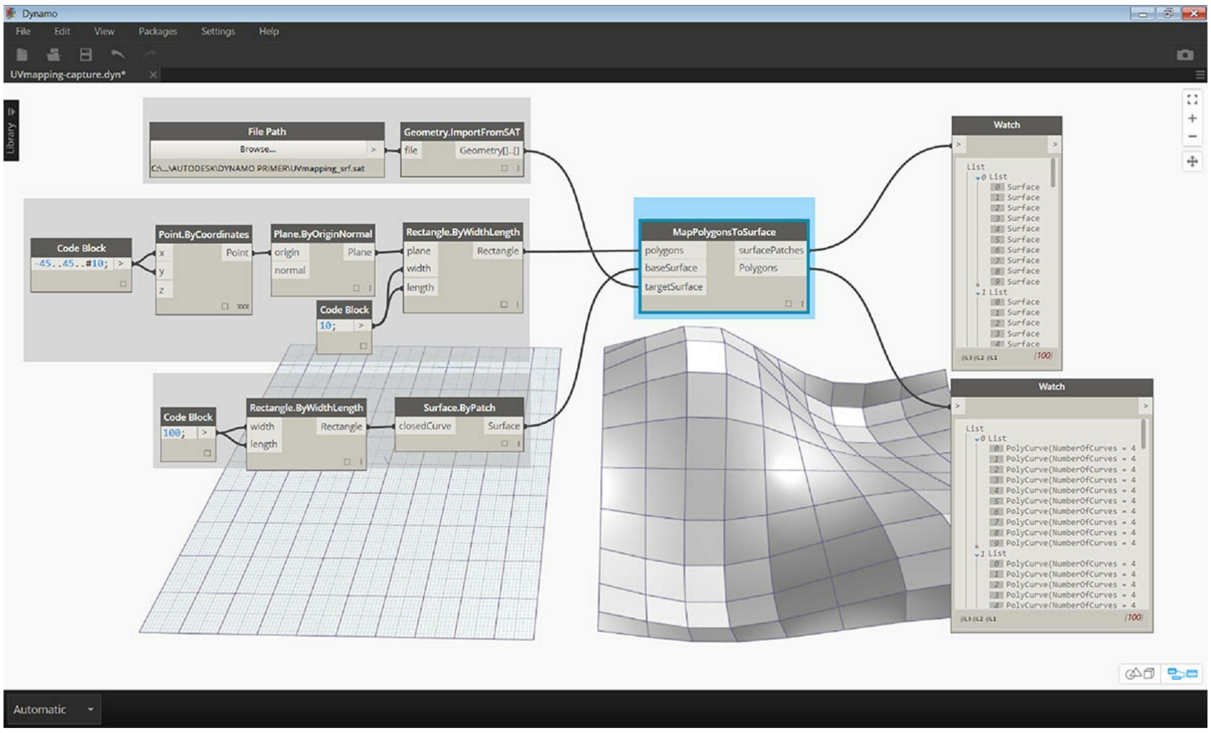
The Custom Node has considerably cleaned up the workspace. Notice that the inputs and outputs have been named based on the original nodes. Let’s edit the Custom Node to make the names more descriptive.



Double click the Custom Node to edit it. This will open a workspace with a yellow background representing the inside of the node.

1. **Inputs:** Change the input names to *baseSurface* and *targetSurface*.
2. **Outputs:** Add an additional output for the mapped polygons.

Save the custom node and return to the home workspace.



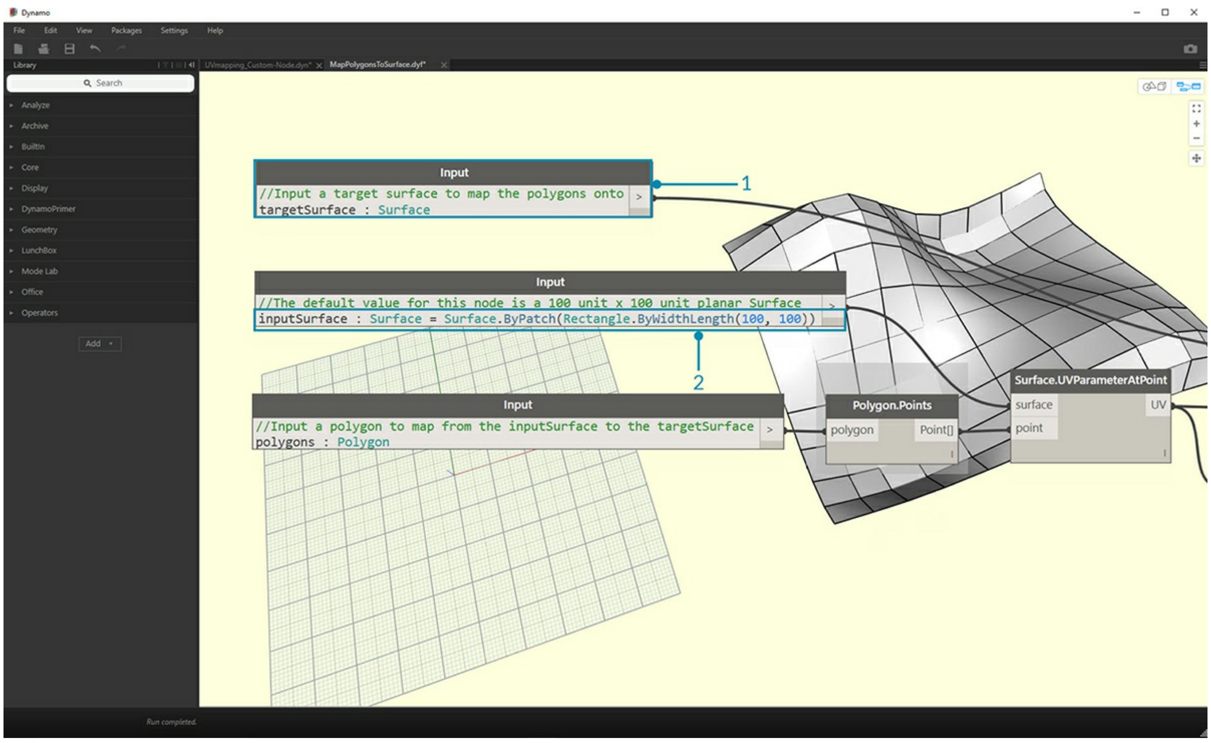
```
-45..45..#10;
```

```
10;
```

```
100;
```

The **MapPolygonsToSurface** node reflects the changes we just made.

We can also add to the robustness of the Custom Node by adding in **Custom Comments**. Comments can help to hint at the input and output types or explain the functionality of the node. Comments will appear when the user hovers over an input or output of a Custom Node.


Dynamo
Library
UVmapping_Custom-Node.dyn*
MapPolygonsToSurface.dyf*
Input
1
2
Polygon.Points
Surface.UVParameterAtPoint
Run completed.

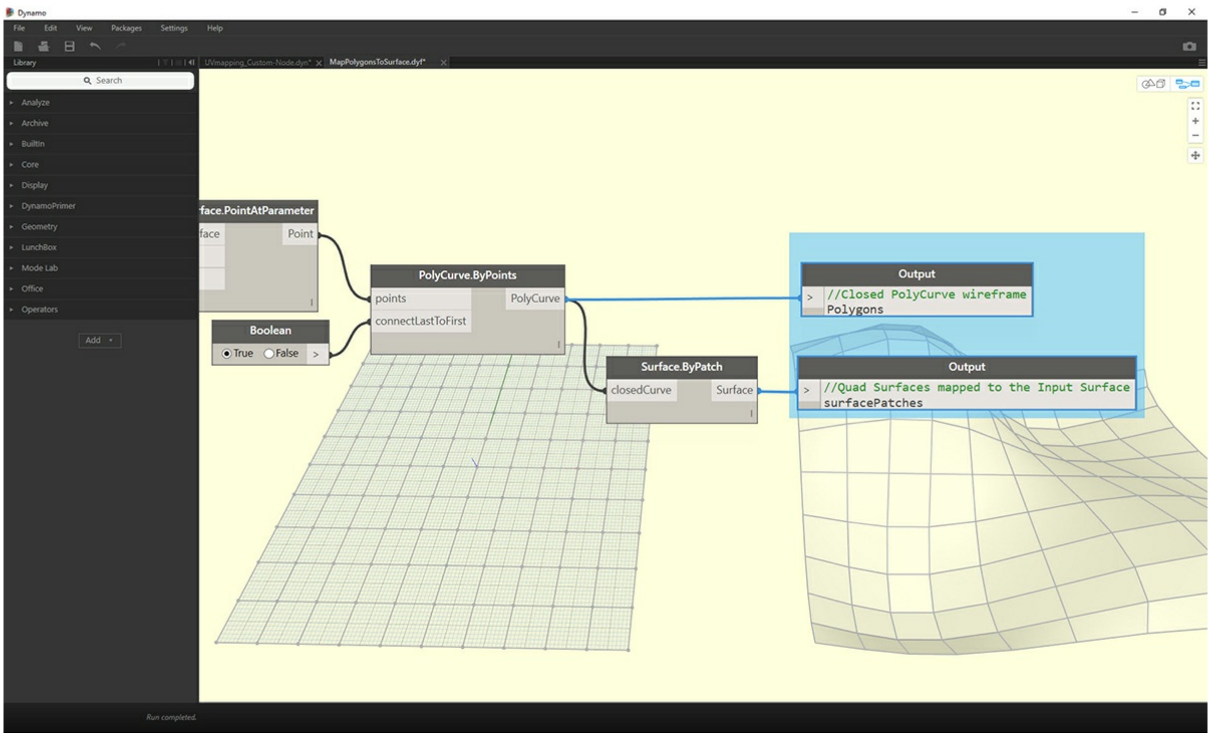
```
//Input a target surface to map the polygons onto
targetSurface : Surface
```

```
//The default value for this node is a 100 unit x 100 unit planar Surface
inputSurface : Surface = Surface.ByPatch(Rectangle.ByWidthLength(100, 100))
```

```
//Input a polygon to map from the inputSurface to the targetSurface
polygons : Polygon
```

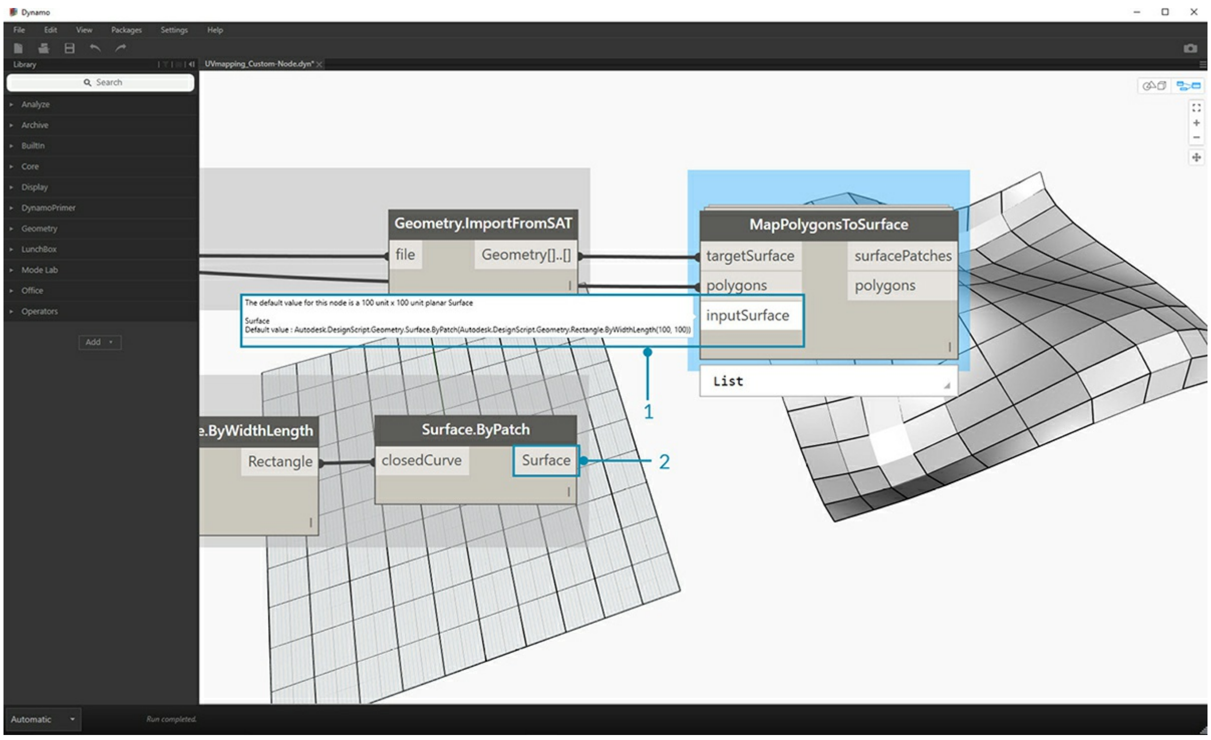
Double click the Custom Node to edit it. This will re-open the yellow background workspace.

1. Begin editing the Input Code Block. To start a Comment, type "//" followed by the comment text. Type anything that may help to clarify the Node - Here we will describe the *targetSurface*.
2. Let's also set the default value for the *inputSurface* by setting the input type equal to a value. Here, we will set the default value to the original Surface.ByPatch set.


```
//Closed PolyCurve wireframe
Polygons
```

```
//Quad Surfaces mapped to the Input Surface
surfacePatches
```

Comments can also be applied to the Outputs. Begin editing the text in the Output Code Block. To start a Comment, type "//" followed by the comment text. Here we will clarify the *Polygons* and the *surfacePatches* Outputs by adding a more in-depth description.


The default value for this node is a 100 unit x 100 unit planar Surface

Surface
Default value : Autodesk.DesignScript.Geometry.Surface.ByPatch(Autodesk.DesignScript.Geometry.Rectangle.ByWidthLength(100, 100))

>

1. Hover over the Custom Node Inputs to see the Comments.
2. With the default value set on our *inputSurface*, we can also run the definition without a surface input.

Publishing to Your Library

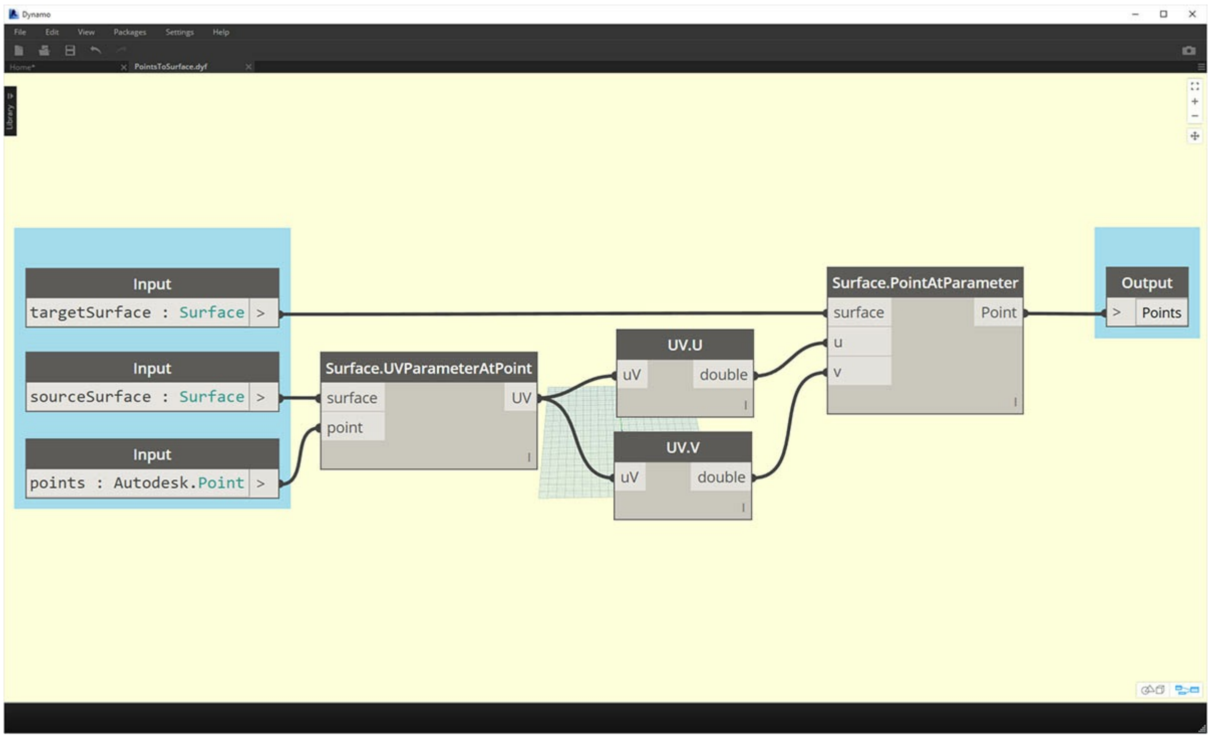
Adding to Your Library

We've just created a custom node and applied it to a specific process in our Dynamo graph. And we like this node so much, we want to keep it in our Dynamo library to reference in other graphs. To do this, we'll publish the node locally. This is a similar process to publishing a package, which we'll walk through in more detail in the next chapter.

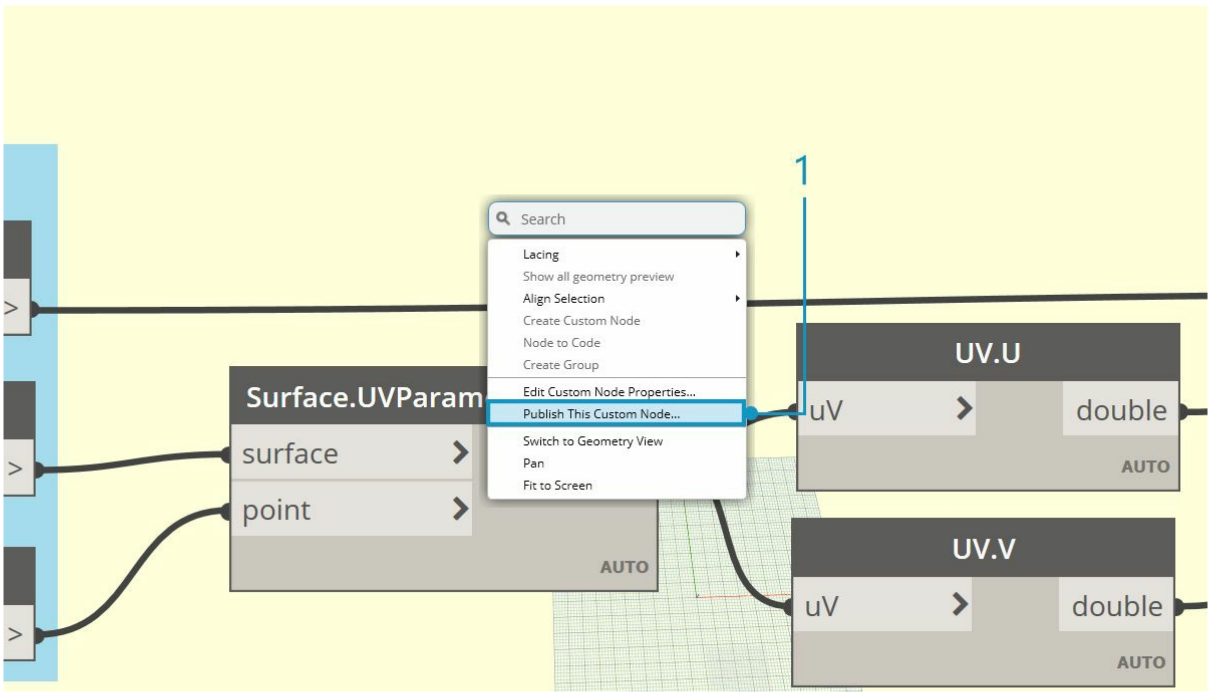
Publishing a Custom Node Locally

Let's move forward with the custom node that we created in the previous section. By publishing a node locally, the node will be accessible in your Dynamo library when you open a new session. Without publishing a node, a Dynamo graph which references a custom node must also have that custom node in its folder (or the custom node must be imported into Dynamo using *File>Import Library*).

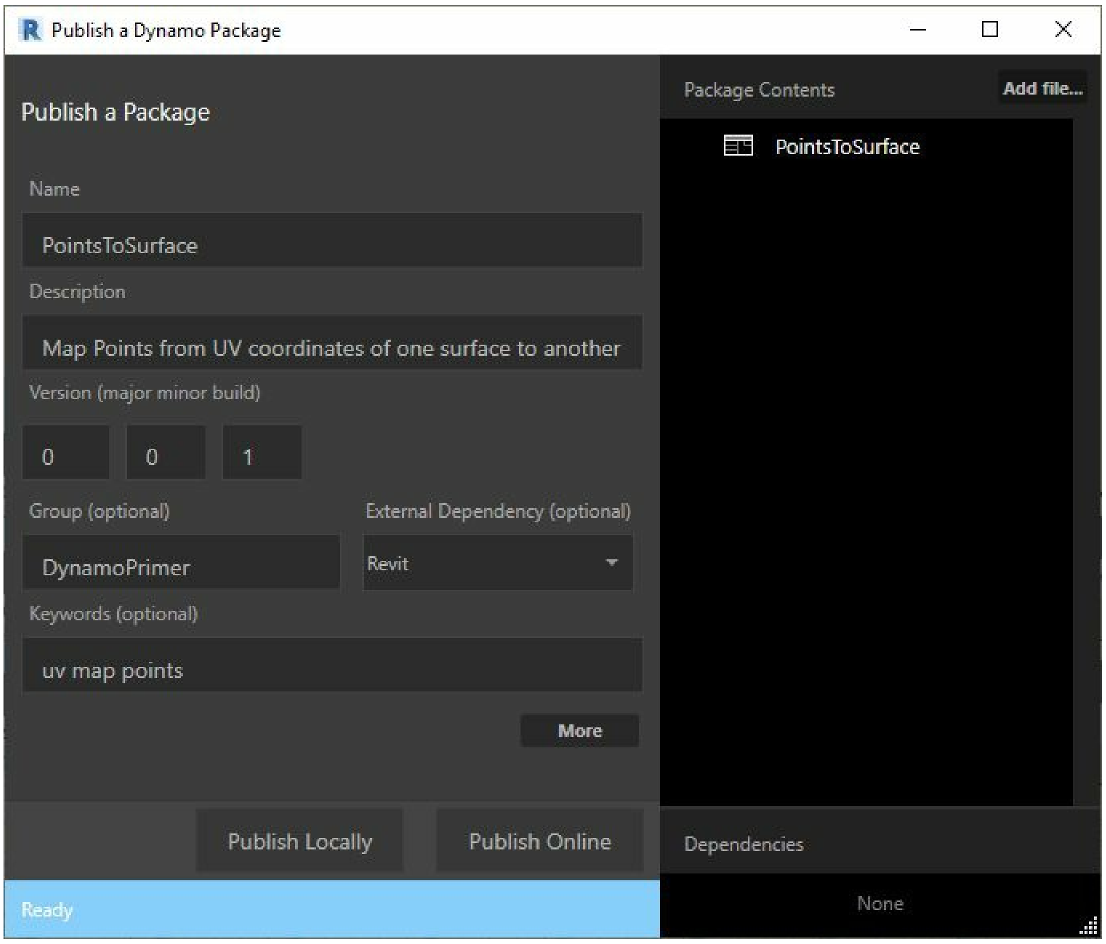
> Download the example file that accompanies this exercise (Right click and "Save Link As..."). A full list of example files can be found in the Appendix. PointsToSurface.dyf



> After opening the PointsToSurface custom node, we see the graph above in the Dynamo Custom Node Editor. You can also open up a custom node by double clicking on it in the Dynamo Graph Editor.



1. To Publish a custom node locally, simply right click on the canvas and select *"Publish This Custom Node..."*


Publish a Dynamo Package

Publish a Package

Name: PointsToSurface

Description: Map Points from UV coordinates of one surface to another

Version (major minor build): 0 0 1

Group (optional): DynamoPrimer

External Dependency (optional): Revit

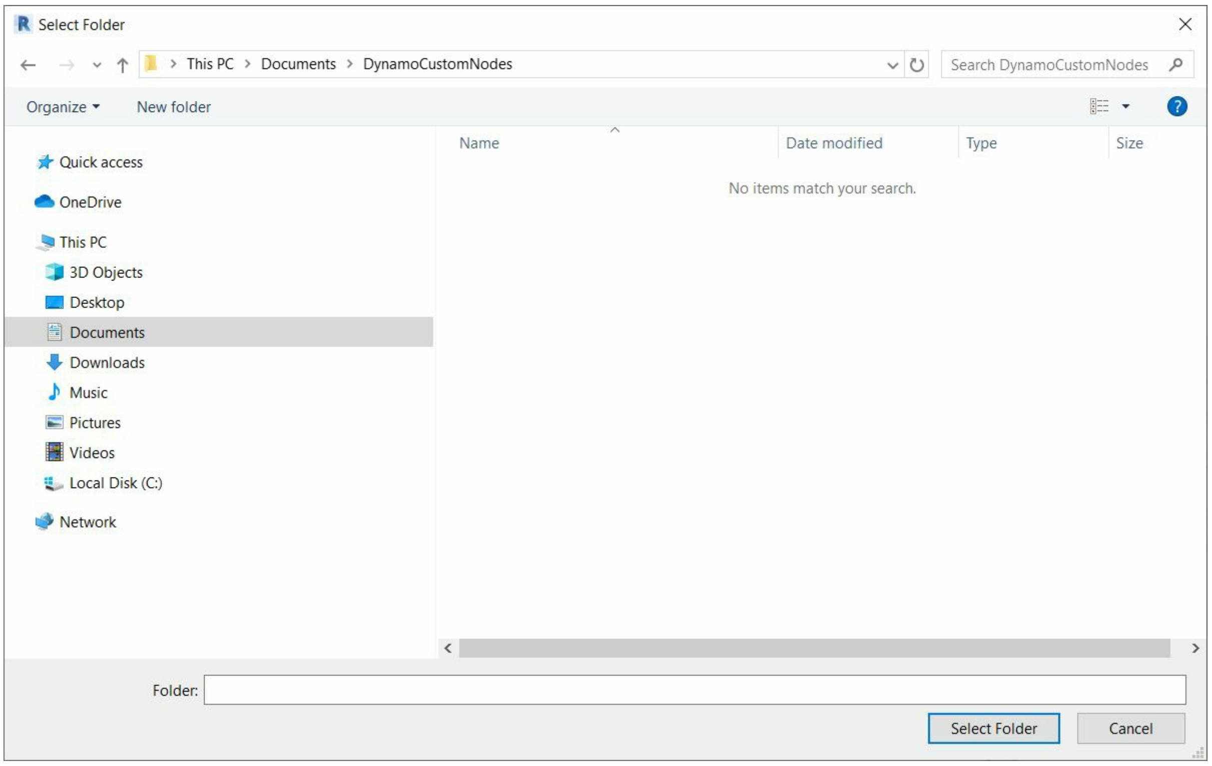
Keywords (optional): uv map points

Package Contents: PointsToSurface

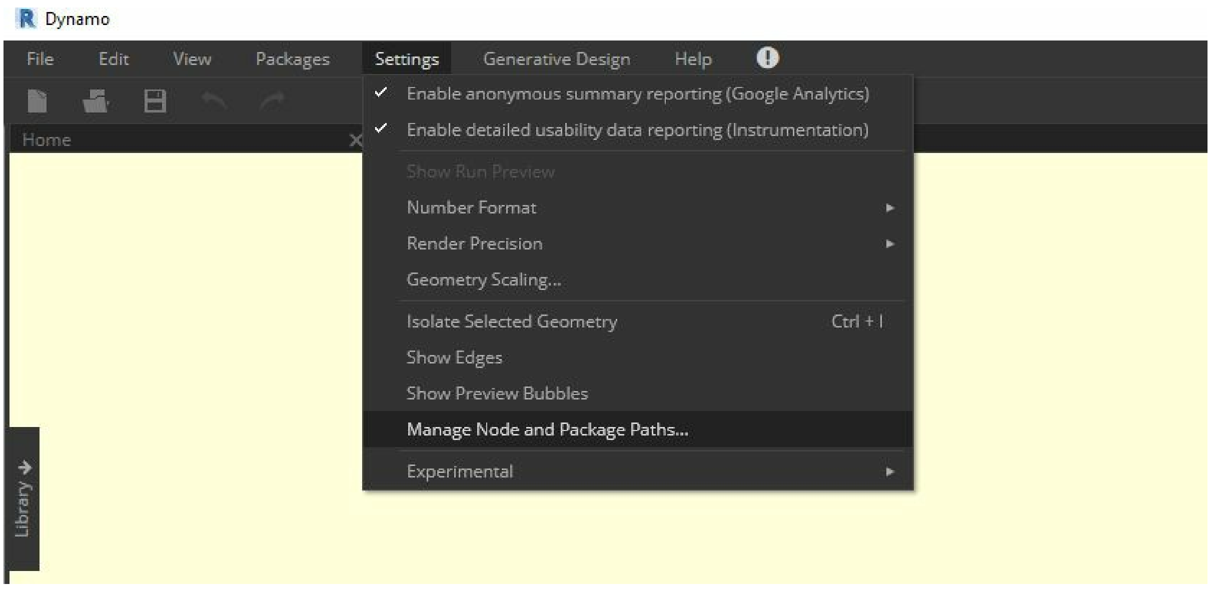
Dependencies: None

Ready

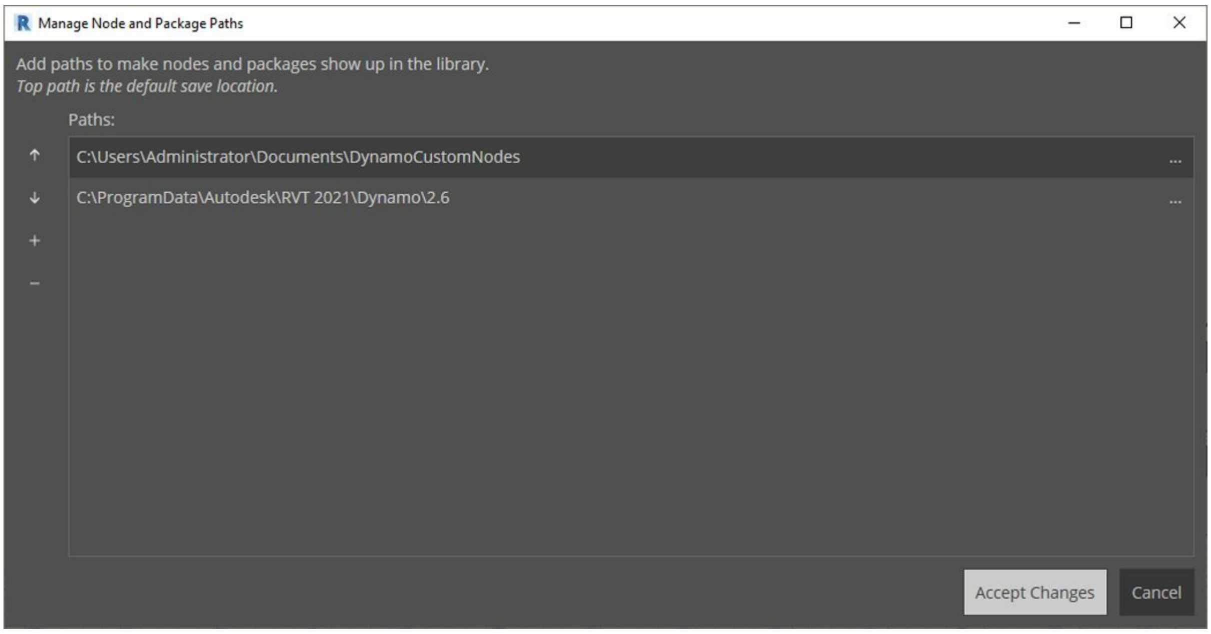
Fill out the relevant information similar to the image above and select *"Publish Locally"*.. Note that the Group field defines the main element accessible from the Dynamo menu.



Choose a folder to house all of the custom nodes that you plan on publishing locally. Dynamo will check this folder each time it loads, so make sure the folder is in a permanent place. Navigate to this folder and choose *"Select Folder"*. Your Dynamo node is now published locally, and will remain in your Dynamo Toolbar each time you load the program!



1. To check on the custom node folder location, go to *Settings > Manage Node and Package Paths...*


Manage Node and Package Paths

Add paths to make nodes and packages show up in the library.
Top path is the default save location.

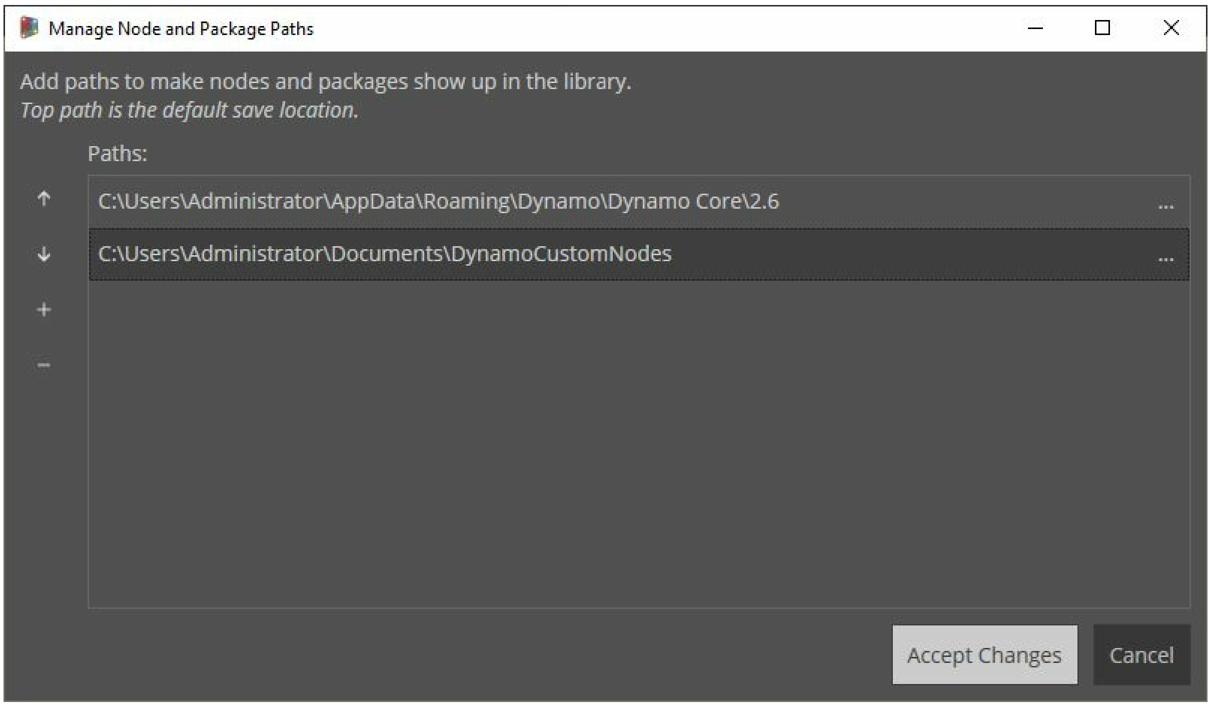
Paths:

C:\Users\Administrator\Documents\DynamoCustomNodes

C:\ProgramData\Autodesk\RVT 2021\Dynamo\2.6

In this window we see two paths: *AppData\Roaming\Dynamo*... refers to the default location of Dynamo Packages installed online. *Documents\DynamoCustomNodes*... refers to the location of custom nodes we've published locally. *

1. You may want to move your local folder path down in the list order above (by selecting the folder path and clicking on the down arrow to the left of the path names). The top folder is the default path for package installs. So by keeping the default Dynamo package install path as the default folder, online packages will be separated from your locally published nodes.*


Manage Node and Package Paths

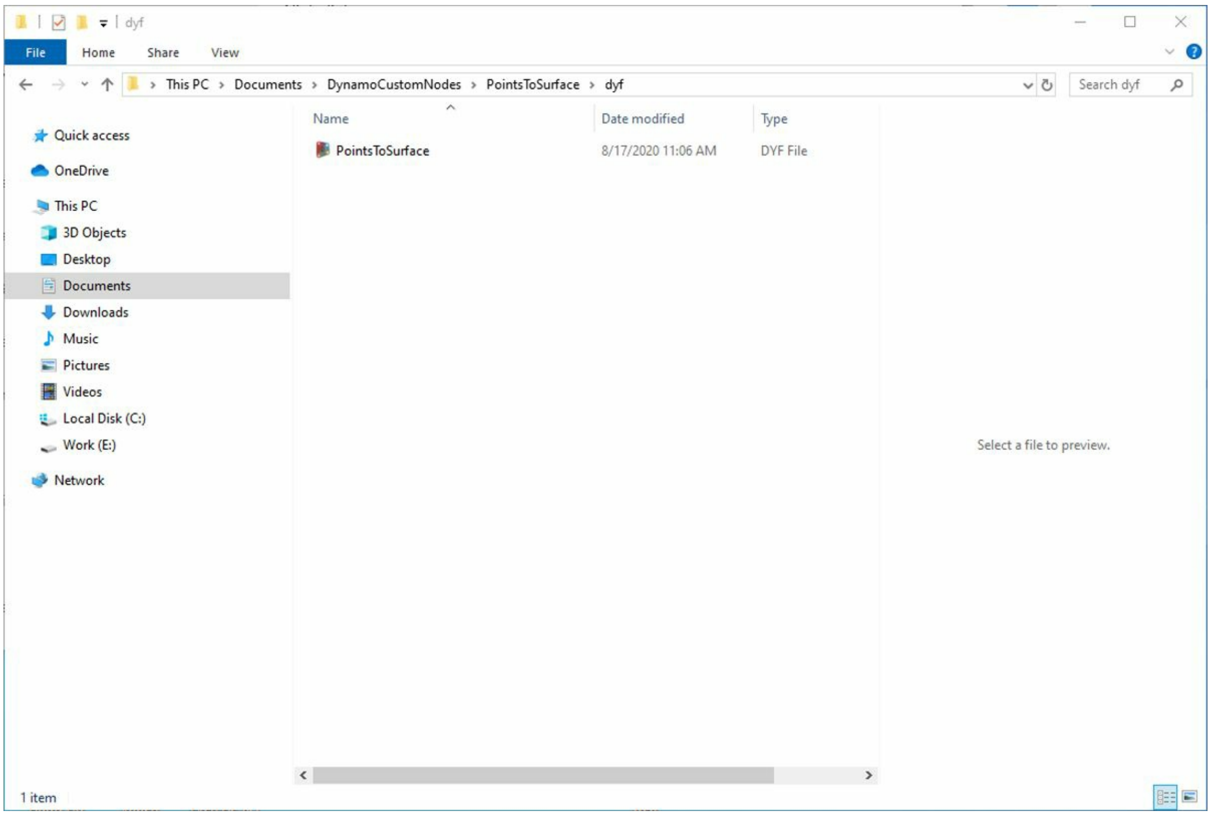
Add paths to make nodes and packages show up in the library.
Top path is the default save location.

Paths:

C:\Users\Administrator\AppData\Roaming\Dynamo\Dynamo Core\2.6

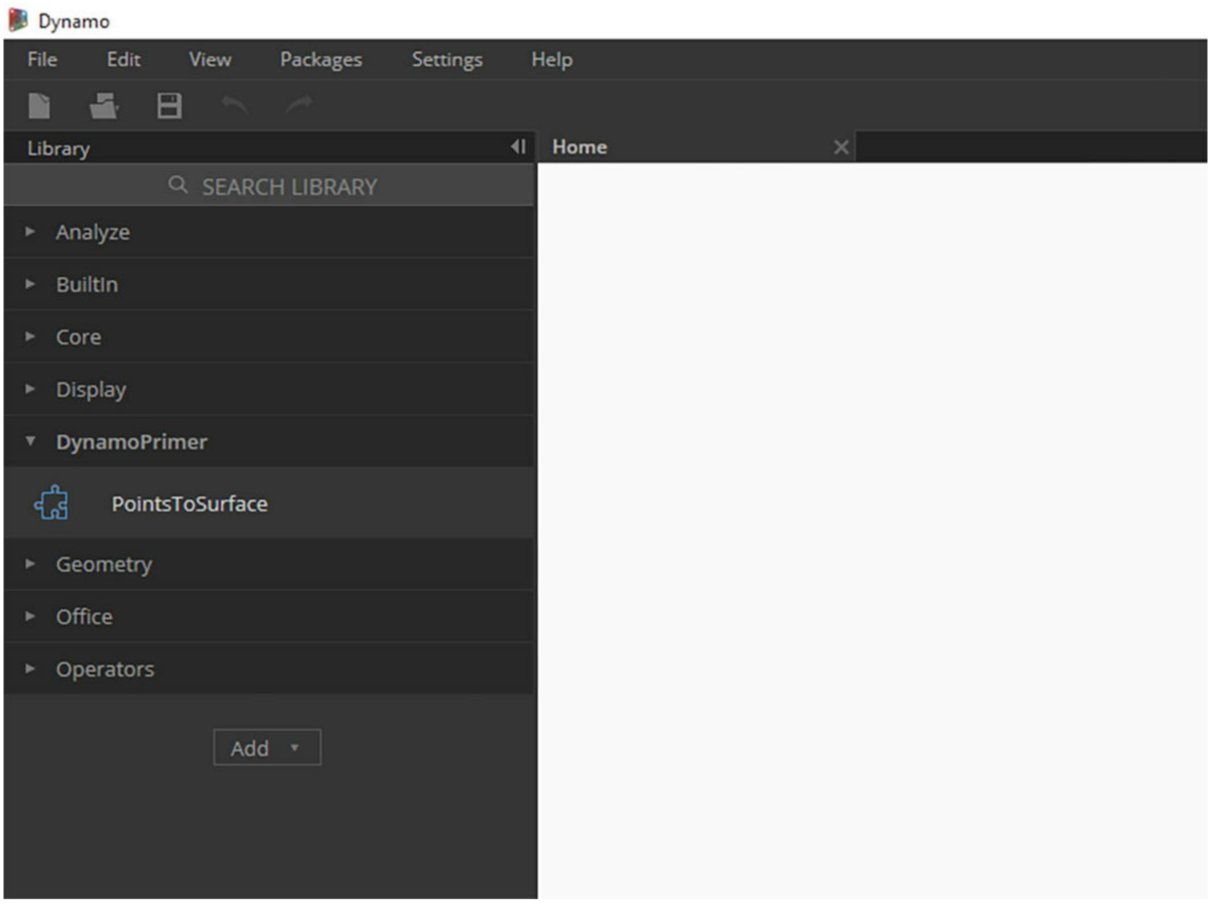
C:\Users\Administrator\Documents\DynamoCustomNodes

We switched the order of the path names in order to have Dynamo's default path as the package install location.



Name	Date modified	Type
PointsToSurface	8/17/2020 11:06 AM	DYF File

Navigating to this local folder, we can find the original custom node in the *".dyf"* folder, which is the extension for a Dynamo Custom Node file. We can edit the file in this folder and the node will update in the UI. We can also add more nodes to the main *DynamoCustomNode* folder and Dynamo will add them to your library at restart!



Dynamo will now load each time with *"PointsToSurface"* in the *"DynamoPrimer"* group of your Dynamo library.

Python Nodes

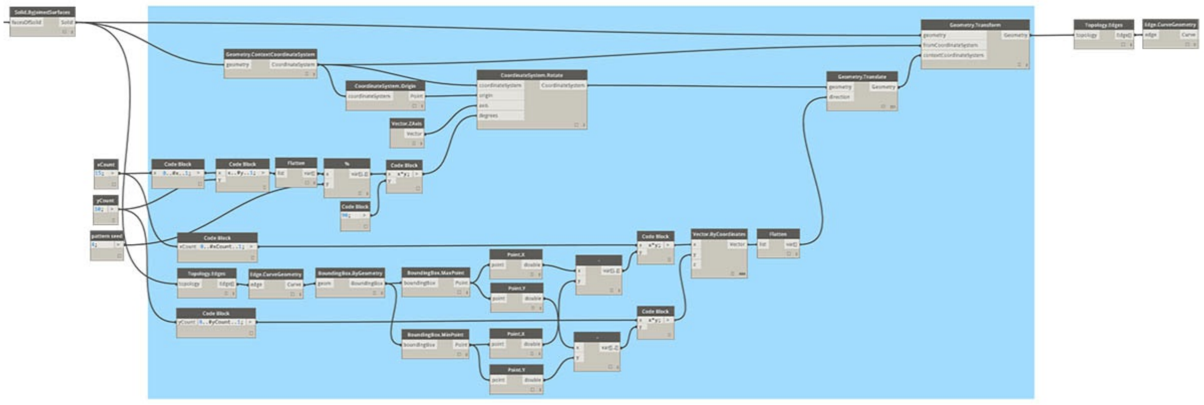
Python

Python is a widely used programming language whose popularity has a lot to do with its style of syntax. It's highly readable, which makes it easier to learn than many other languages. Python supports modules and packages, and can be embedded into existing applications. The examples in this section assume a basic familiarity with Python. For information about how to get up and running with Python, a good resource is the "Getting Started" page on Python.org.

Visual vs. Textual Programming

Why would you use textual programming in Dynamo's visual programming environment? As we discussed in chapter 1.1, visual programming has many advantages. It allows you to create programs without learning special syntax in an intuitive visual interface. However, a visual program can become cluttered, and can at times fall short in functionality. For example, Python offers much more achieveable methods for writing conditional statements (if/then) and looping. Python is a powerful tool that can extend the capabilities of Dynamo and allow you to replace many nodes with a few concise lines of code.

Visual Program:

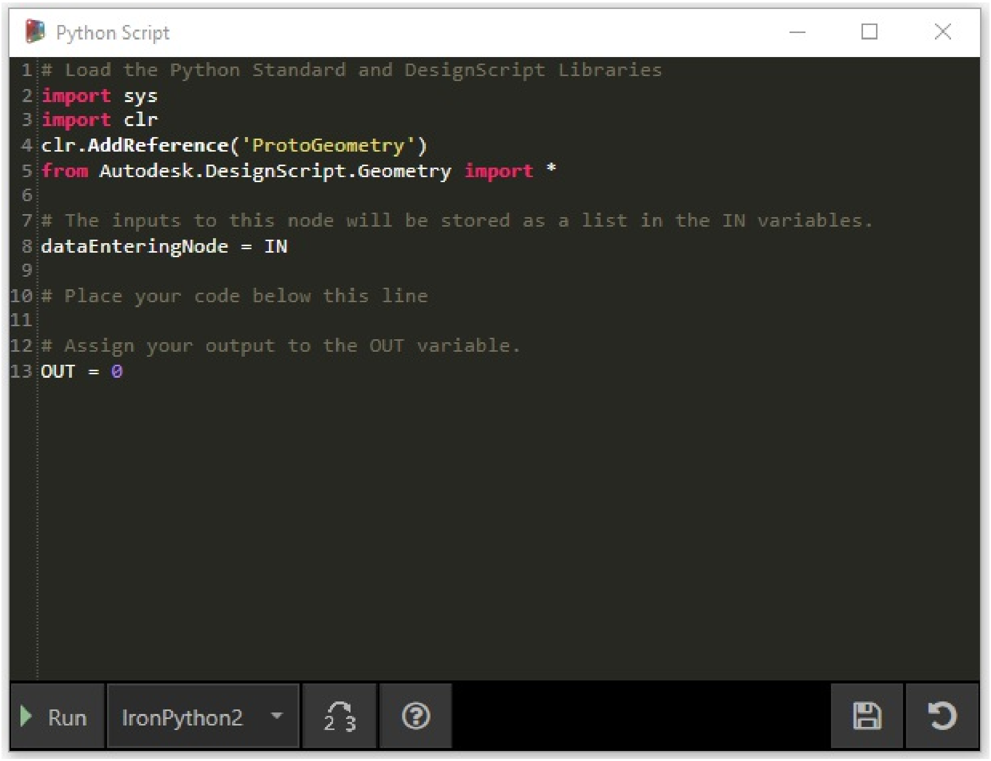


Textual Program:

```
import clr
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *

solid = IN[0]
seed = IN[1]
xCount = IN[2]
yCount = IN[3]

solids = []

yDist = solid.BoundingBox.MaxPoint.Y-solid.BoundingBox.MinPoint.Y
xDist = solid.BoundingBox.MaxPoint.X-solid.BoundingBox.MinPoint.X

for i in xRange:
    for j in yRange:
        fromCoord = solid.ContextCoordinateSystem
        toCoord = fromCoord.Rotate(solid.ContextCoordinateSystem.Origin,Vector.ByCoordinates(0,0,1),(90*(
        vec = Vector.ByCoordinates((xDist*i),(yDist*j),0)
        toCoord = toCoord.Translate(vec)
        solids.append(solid.Transform(fromCoord,toCoord))

OUT = solids
```

The Python Node

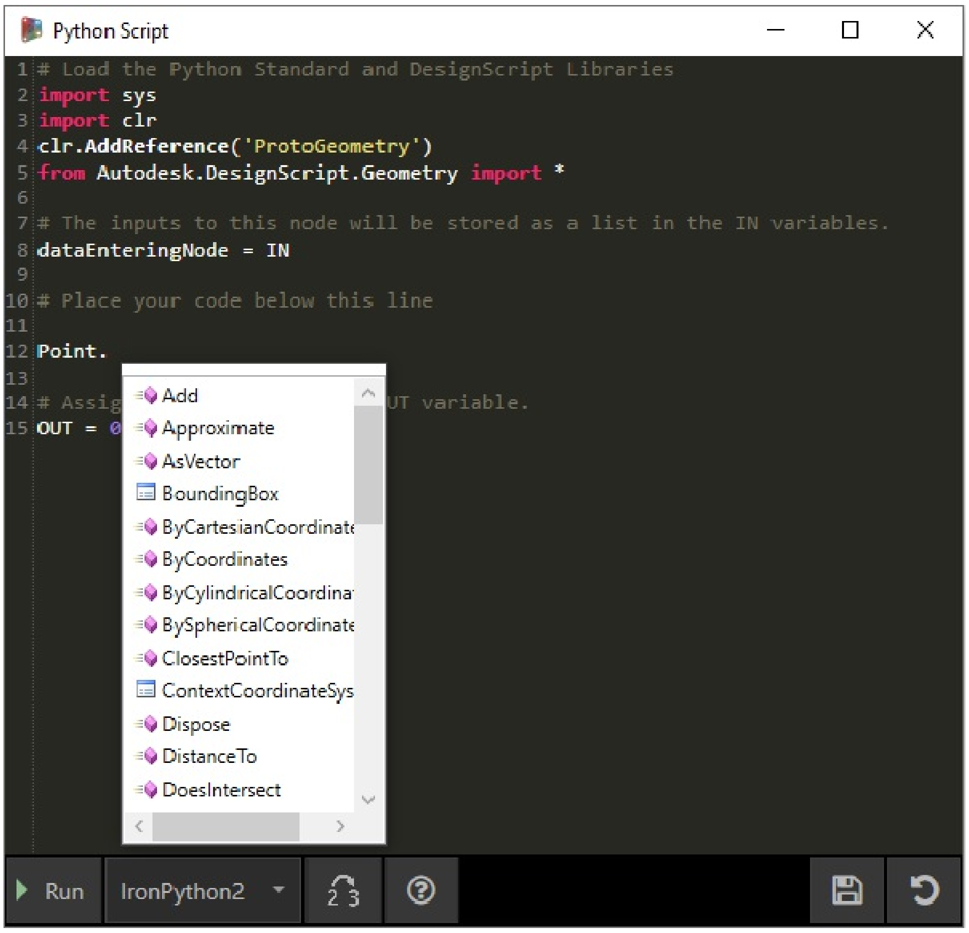
Like code blocks, Python nodes are a scripting interface within a visual programming environment. The Python node can be found under *Core>Scripting* in the library. Double clicking the node opens the python script editor (you can also right click on the node and select *Edit...*).



```
# Load the Python Standard and DesignScript Libraries
import sys
import clr
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *

# The inputs to this node will be stored as a list in the IN variables.
dataEnteringNode = IN

# Place your code below this line

# Assign your output to the OUT variable.
OUT = 0
```

You'll notice some boilerplate text at the top, which is meant to help you reference the libraries you'll need. Inputs are stored in the IN array. Values are returned to Dynamo by assigning them to the OUT variable.

The Autodesk.DesignScript.Geometry library allows you to use dot notation similar to Code Blocks. For more information on Dynamo syntax, refer to chapter 7.2 as well as the DesignScript Guide. Typing a geometry type such as 'Point.' will bring up a list of methods for creating and querying points.



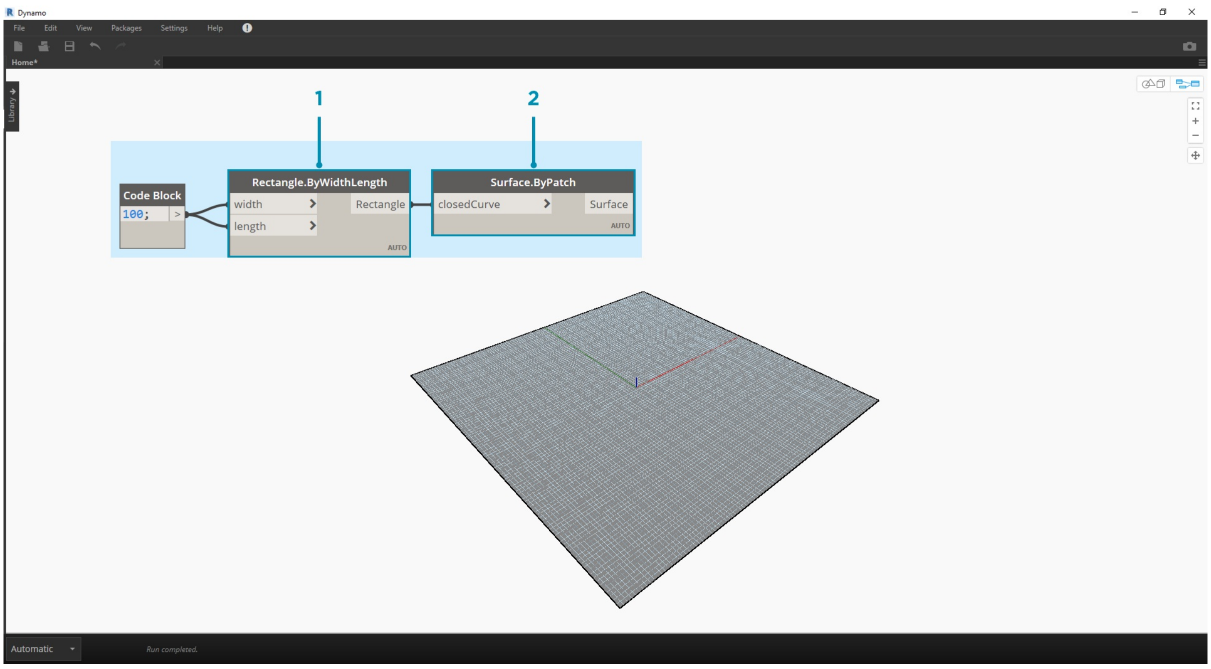
```
# Load the Python Standard and DesignScript Libraries
import sys
import clr
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *

# The inputs to this node will be stored as a list in the IN variables.
dataEnteringNode = IN

# Place your code below this line

Point.

# Assig                                  UT variable.
OUT = 0
```

Methods include constructors such as *ByCoordinates*, actions like *Add*, and queries like *X*, *Y* and *Z* coordinates.

Exercise

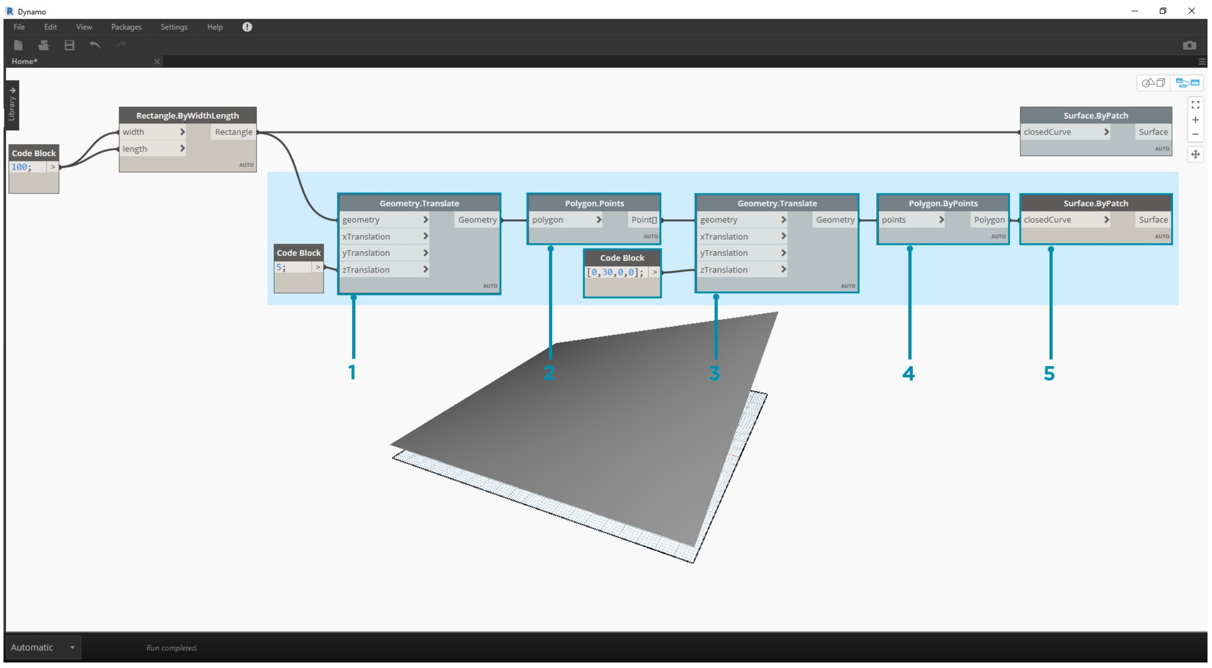
Download the example file that accompanies this exercise (Right click and "Save Link As..."). A full list of example files can be found in the Appendix. Python_Custom-Node.dyn

In this example, we will write a python script that creates patterns from a solid module, and turn it into a custom node. First, let’s create our solid module using Dynamo nodes.



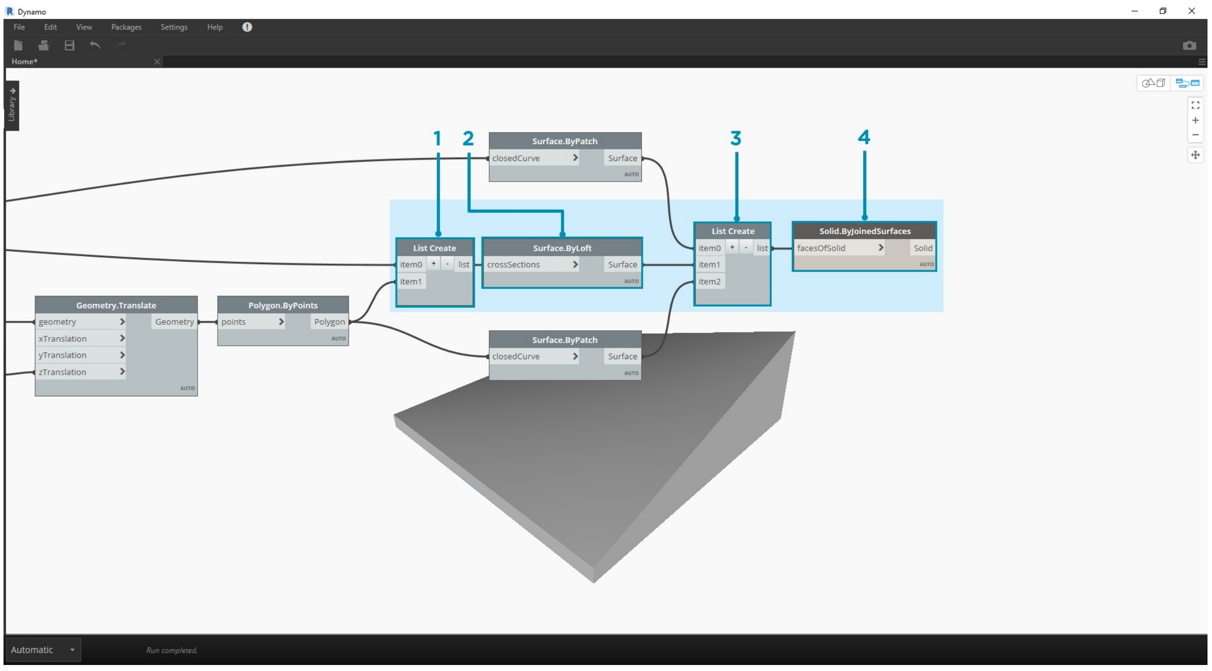
1. **Rectangle.ByWidthLength:** Create a rectangle that will be the base of our solid.

2. **Surface.ByPatch:** Connect the rectangle to the '*closedCurve*' input to create the bottom surface.



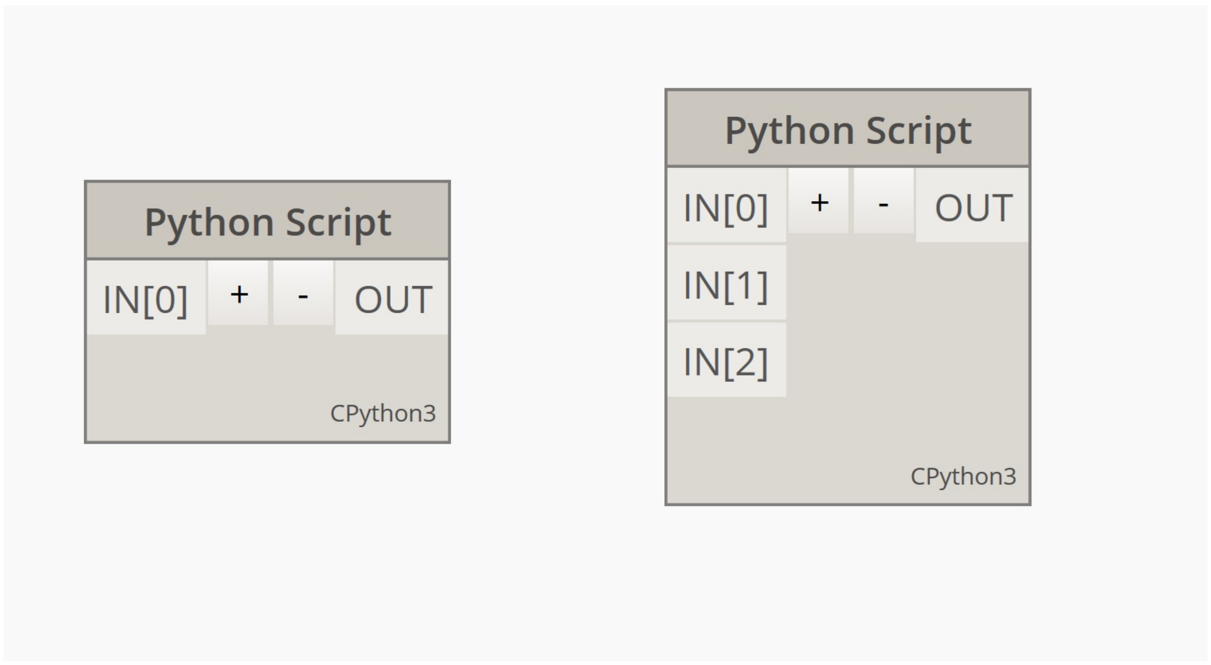
1. **Geometry.Translate:** Connect the rectangle to the '*geometry*' input to move it up, using a code block to specify the base thickness of our solid.
2. **Polygon.Points:** Query the translated rectangle to extract the corner points.
3. **Geometry.Translate:** Use a code block to create a list of four values corresponding to the four points, translating one corner of the solid up.
4. **Polygon.ByPoints:** Use the translated points to reconstruct the top polygon.
5. **Surface.ByPatch:** Connect the polygon to create the top surface.

Now that we have our top and bottom surfaces, let's loft between the two profiles to create the sides of the solid.



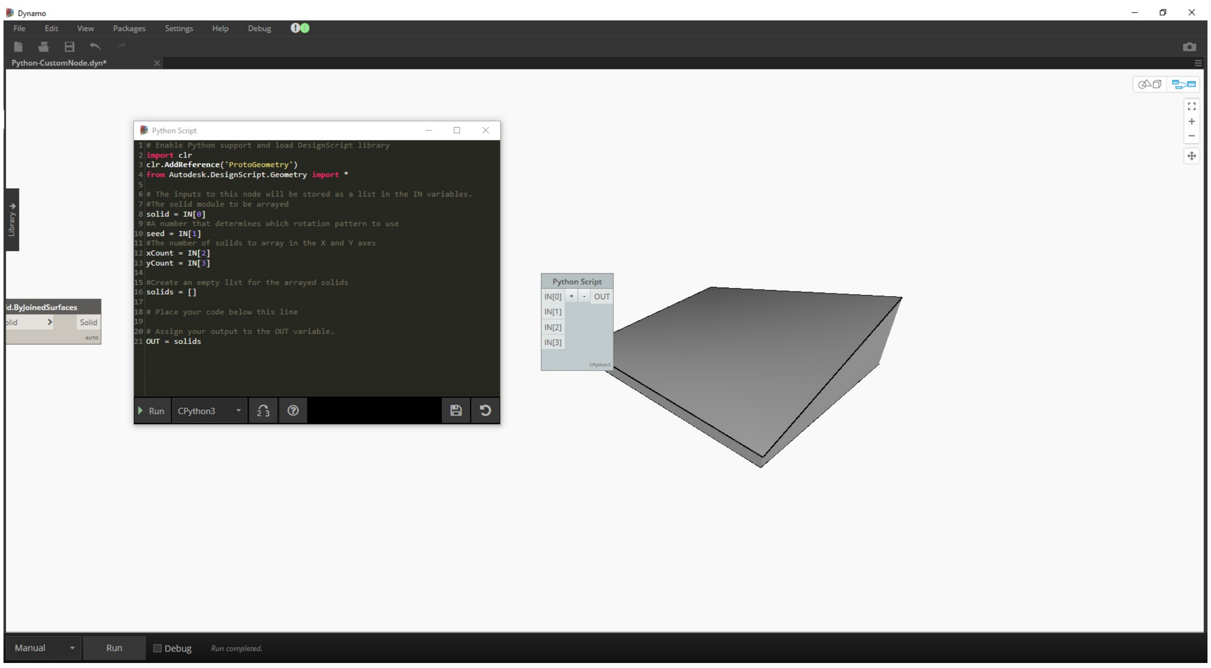
1. **List.Create:** Connect the bottom rectangle and the top polygon to the index inputs.
2. **Surface.ByLoft:** Loft the two profiles to create the sides of the solid.
3. **List.Create:** Connect the top, side, and bottom surfaces to the index inputs to create a list of surfaces.
4. **Solid.ByJoinedSurfaces:** Join the surfaces to create the solid module.

Now that we have our solid, let's drop a Python Script node onto the workspace.



To add additional inputs to the node, close the editor and click the + icon on the node. The inputs are named IN[0], IN[1], etc. to indicate that they represent items in a list.

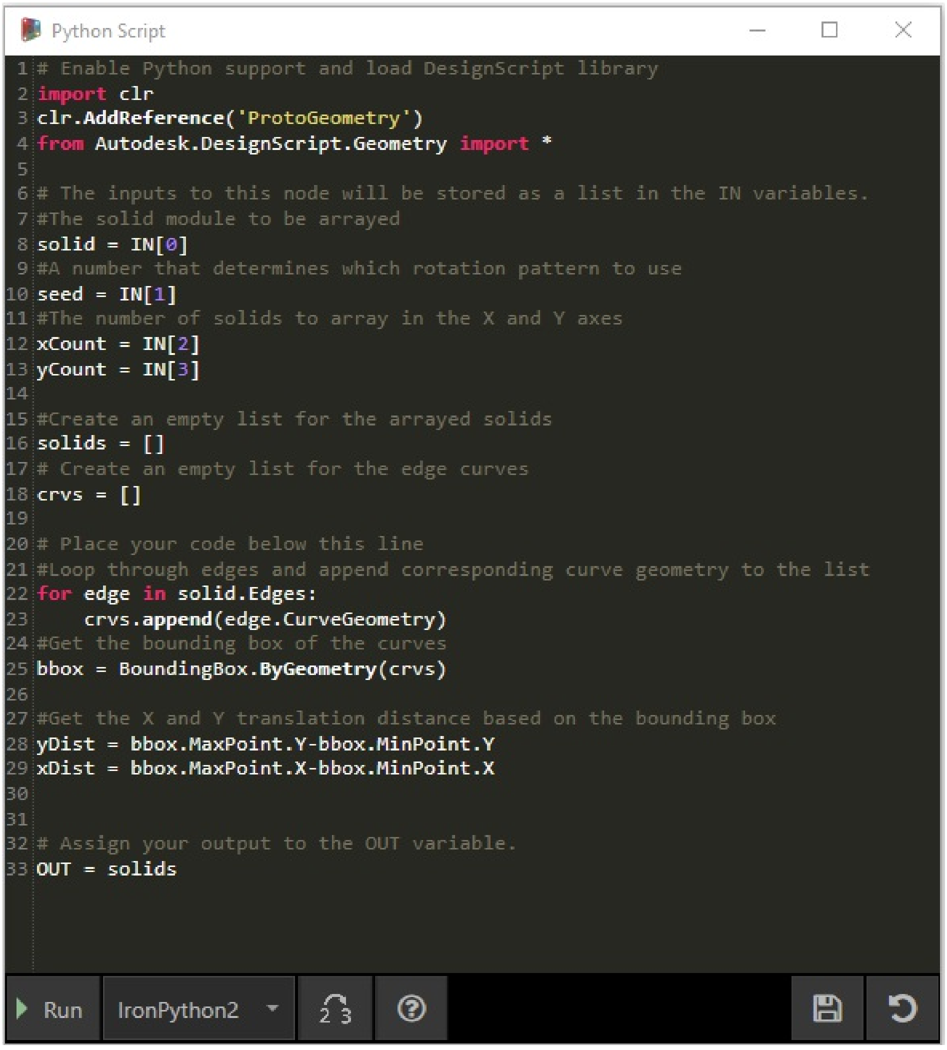
Let’s start by defining our inputs and output. Double click the node to open the python editor.



```
# Enable Python support and load DesignScript library
import clr
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *

# The inputs to this node will be stored as a list in the IN variables.
#The solid module to be arrayed
solid = IN[0]
#A number that determines which rotation pattern to use
seed = IN[1]
#The number of solids to array in the X and Y axes
xCount = IN[2]
yCount = IN[3]

#Create an empty list for the arrayed solids
solids = []

# Place your code below this line

# Assign your output to the OUT variable.
OUT = solids
```

```
# Enable Python support and load DesignScript library
import clr
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *

# The inputs to this node will be stored as a list in the IN variables.
#The solid module to be arrayed
solid = IN[0]
#A number that determines which rotation pattern to use
seed = IN[1]
#The number of solids to array in the X and Y axes
xCount = IN[2]
yCount = IN[3]

#Create an empty list for the arrayed solids
solids = []

# Place your code below this line
# Assign your output to the OUT variable.
OUT = solids
```

This code will make more sense as we progress in the exercise. Next we need to think about what information is required in order to array our solid module. First, we will need to know the dimensions of the solid to determine the translation distance. Due to a bounding box bug, we will have to use the edge curve geometry to create a bounding box.



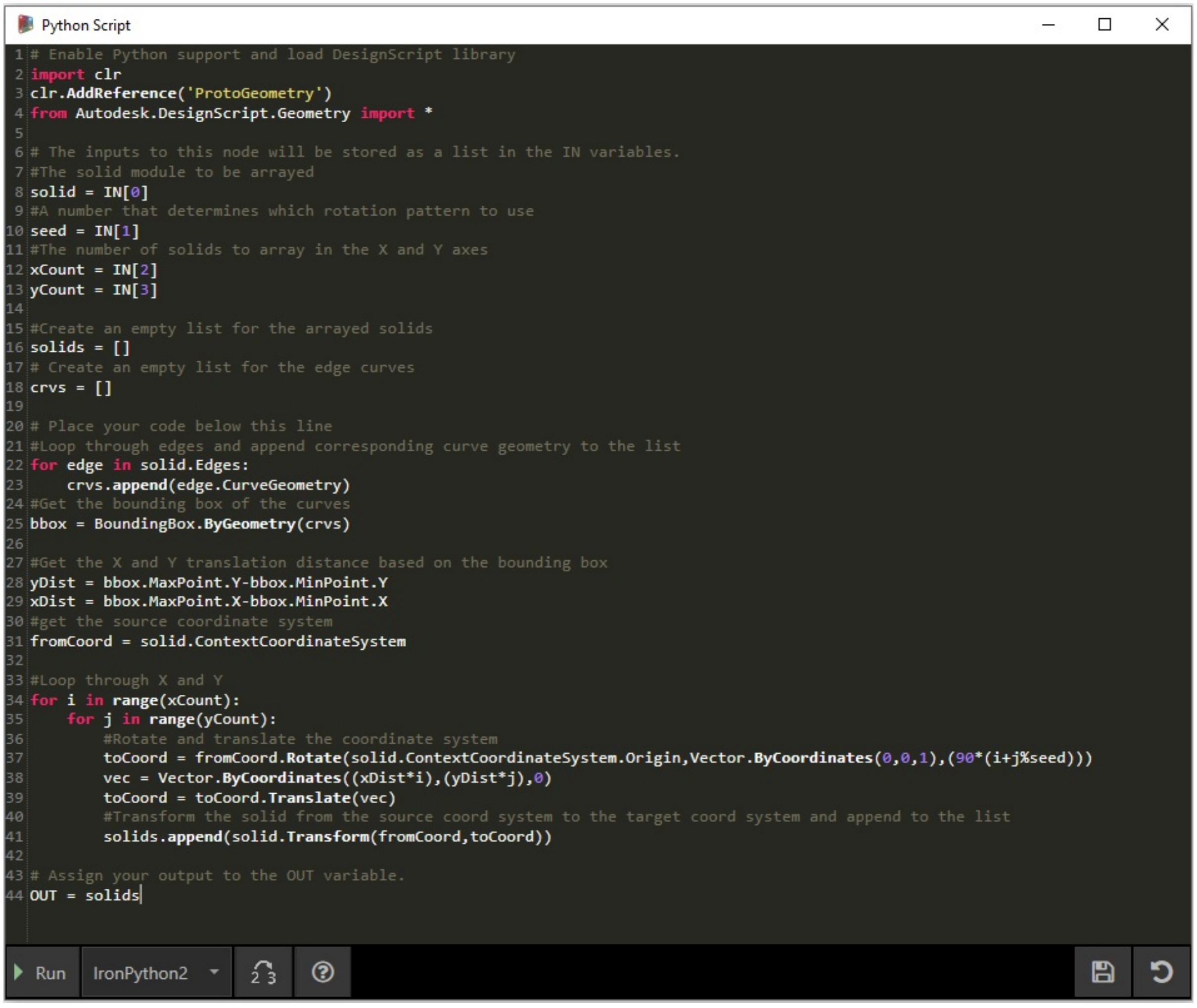
```
# Enable Python support and load DesignScript library
import clr
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *

# The inputs to this node will be stored as a list in the IN variables.
#The solid module to be arrayed
solid = IN[0]
#A number that determines which rotation pattern to use
seed = IN[1]
#The number of solids to array in the X and Y axes
xCount = IN[2]
yCount = IN[3]

#Create an empty list for the arrayed solids
solids = []
# Create an empty list for the edge curves
crvs = []

# Place your code below this line
#Loop through edges and append corresponding curve geometry to the list
for edge in solid.Edges:
    crvs.append(edge.CurveGeometry)
#Get the bounding box of the curves
bbox = BoundingBox.ByGeometry(crvs)

#Get the X and Y translation distance based on the bounding box
yDist = bbox.MaxPoint.Y-bbox.MinPoint.Y
xDist = bbox.MaxPoint.X-bbox.MinPoint.X


# Assign your output to the OUT variable.
OUT = solids
```

A look at the Python node in Dynamo. Notice that we're using the same syntax as we see in the titles of the nodes in Dynamo. The commented code is below.

```
# Enable Python support and load DesignScript library
import clr
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *

# The inputs to this node will be stored as a list in the IN variables.
#The solid module to be arrayed
solid = IN[0]
#A number that determines which rotation pattern to use
seed = IN[1]
#The number of solids to array in the X and Y axes
xCount = IN[2]
yCount = IN[3]

#Create an empty list for the arrayed solids
solids = []
# Create an empty list for the edge curves
crvs = []

# Place your code below this line
#Loop through edges and append corresponding curve geometry to the list
for edge in solid.Edges:
    crvs.append(edge.CurveGeometry)
#Get the bounding box of the curves
bbox = BoundingBox.ByGeometry(crvs)
```

```
#Get the X and Y translation distance based on the bounding box
yDist = bbox.MaxPoint.Y-bbox.MinPoint.Y
xDist = bbox.MaxPoint.X-bbox.MinPoint.X


# Assign your output to the OUT variable.
OUT = solids
```

Since we will be both translating and rotating the solid modules, let’s use the Geometry.Transform operation. By looking at the Geometry.Transform node, we know that we will need a source coordinate system and a target coordinate system to transform the solid. The source is the context coordinate system of our solid, while the target will be a different coordinate system for each arrayed module. That means we will have to loop through the x and y values to transform the coordinate system differently each time.

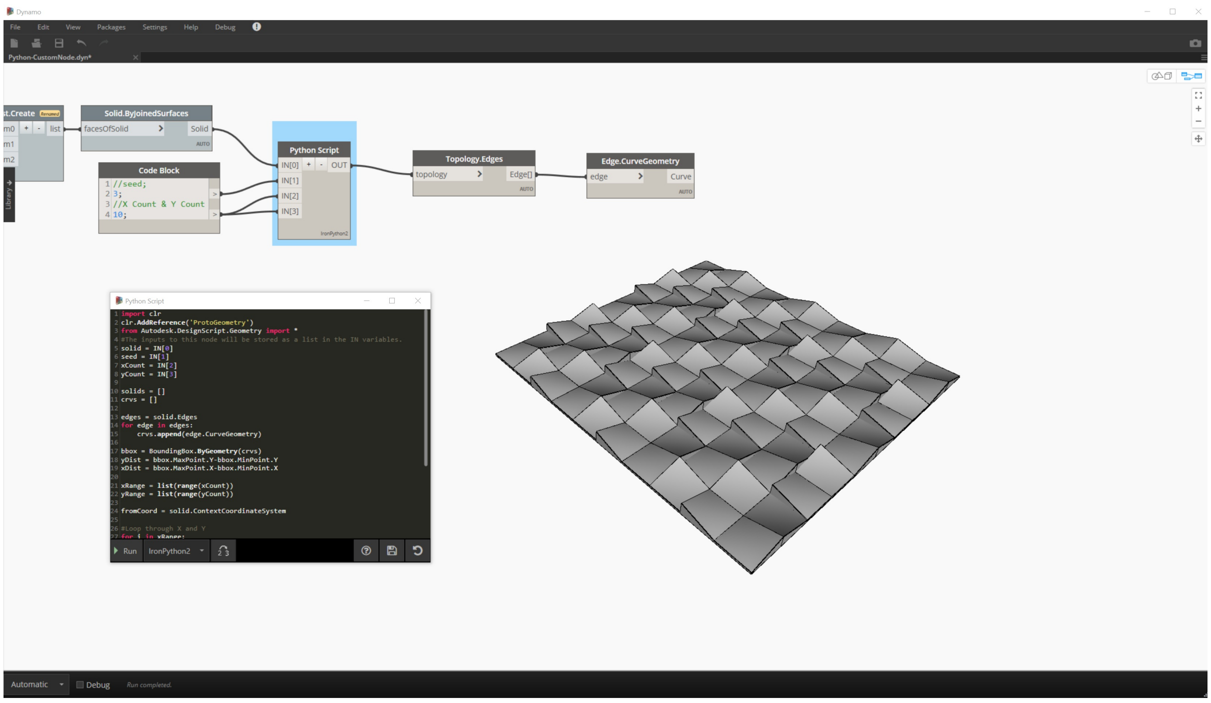


```
# Enable Python support and load DesignScript library
import clr
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *

# The inputs to this node will be stored as a list in the IN variables.
#The solid module to be arrayed
solid = IN[0]
#A number that determines which rotation pattern to use
seed = IN[1]
#The number of solids to array in the X and Y axes
xCount = IN[2]
yCount = IN[3]

#Create an empty list for the arrayed solids
solids = []
# Create an empty list for the edge curves
crvs = []

# Place your code below this line
#Loop through edges and append corresponding curve geometry to the list
for edge in solid.Edges:
    crvs.append(edge.CurveGeometry)
#Get the bounding box of the curves
bbox = BoundingBox.ByGeometry(crvs)

#Get the X and Y translation distance based on the bounding box
yDist = bbox.MaxPoint.Y-bbox.MinPoint.Y
xDist = bbox.MaxPoint.X-bbox.MinPoint.X
#get the source coordinate system
fromCoord = solid.ContextCoordinateSystem

#Loop through X and Y
for i in range(xCount):
    for j in range(yCount):
        #Rotate and translate the coordinate system
        toCoord = fromCoord.Rotate(solid.ContextCoordinateSystem.Origin,Vector.ByCoordinates(0,0,1),(90*(i+j%seed)))
        vec = Vector.ByCoordinates((xDist*i),(yDist*j),0)
        toCoord = toCoord.Translate(vec)
        #Transform the solid from the source coord system to the target coord system and append to the list
        solids.append(solid.Transform(fromCoord,toCoord))

# Assign your output to the OUT variable.
OUT = solids
```

A look at the Python node in Dynamo. The commented code is below.

```
# Enable Python support and load DesignScript library
import clr
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *

# The inputs to this node will be stored as a list in the IN variables.
#The solid module to be arrayed
solid = IN[0]
#A number that determines which rotation pattern to use
seed = IN[1]
#The number of solids to array in the X and Y axes
xCount = IN[2]
yCount = IN[3]

#Create an empty list for the arrayed solids
solids = []
# Create an empty list for the edge curves
crvs = []

# Place your code below this line
#Loop through edges and append corresponding curve geometry to the list
```

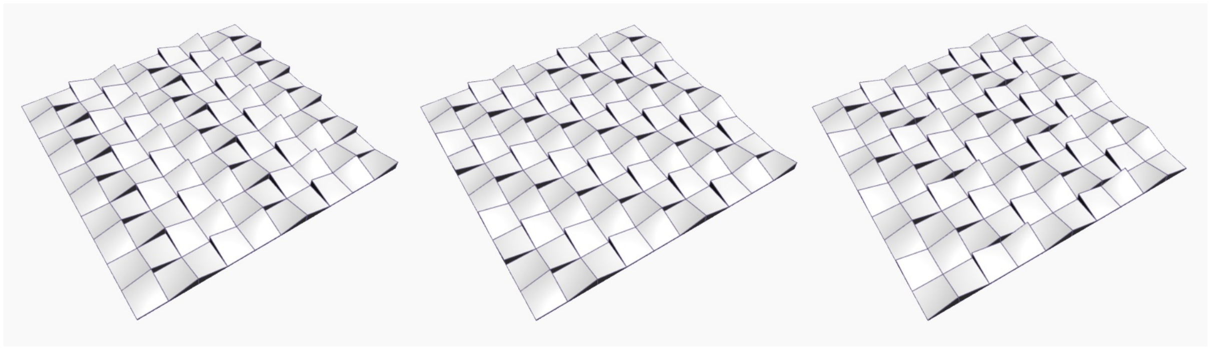
```
for edge in solid.Edges:
    crvs.append(edge.CurveGeometry)
#Get the bounding box of the curves
bbox = BoundingBox.ByGeometry(crvs)

#Get the X and Y translation distance based on the bounding box
yDist = bbox.MaxPoint.Y-bbox.MinPoint.Y
xDist = bbox.MaxPoint.X-bbox.MinPoint.X
#get the source coordinate system
fromCoord = solid.ContextCoordinateSystem

#Loop through X and Y
for i in range(xCount):
    for j in range(yCount):
        #Rotate and translate the coordinate system
        toCoord = fromCoord.Rotate(solid.ContextCoordinateSystem.Origin,Vector.ByCoordinates(0,0,1),(90*(
        vec = Vector.ByCoordinates((xDist*i),(yDist*j),0)
        toCoord = toCoord.Translate(vec)
        #Transform the solid from the source coord system to the target coord system and append to the li
        solids.append(solid.Transform(fromCoord,toCoord))

# Assign your output to the OUT variable.
OUT = solids
```



```
//seed;
3;
//X Count & Y Count
10;
```

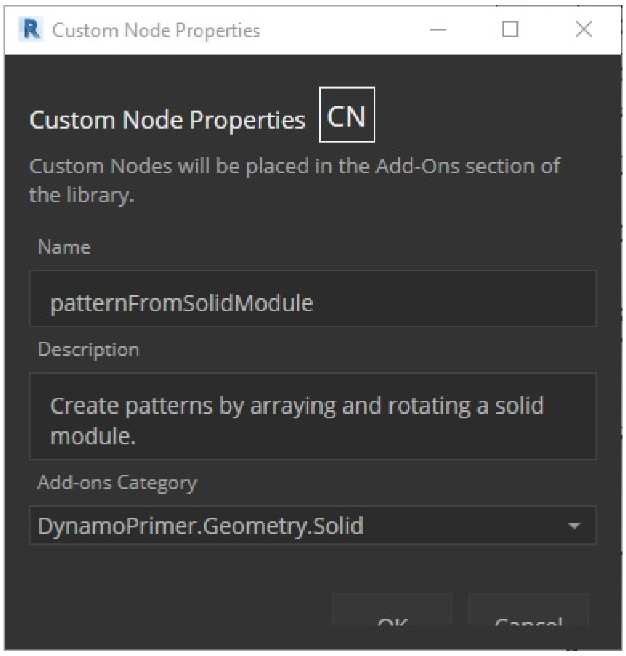
```
import clr
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *
#The inputs to this node will be stored as a list in the IN variables.
solid = IN[0]
seed = IN[1]
xCount = IN[2]
yCount = IN[3]

solids = []
crvs = []

edges = solid.Edges
for edge in edges:
    crvs.append(edge.CurveGeometry)

bbox = BoundingBox.ByGeometry(crvs)
yDist = bbox.MaxPoint.Y-bbox.MinPoint.Y
xDist = bbox.MaxPoint.X-bbox.MinPoint.X

xRange = list(range(xCount))
yRange = list(range(yCount))

fromCoord = solid.ContextCoordinateSystem

#Loop through X and Y
for i in xRange:
```

Clicking run on the python node will allow our code to execute.



Try changing the seed value to create different patterns. You can also change the parameters of the solid module itself for different effects. In Dynamo 2.0 you can simply change the seed and click run without closing the Python window.

Now that we have created a useful python script, let's save it as a custom node. Select the python script node, right-click and select 'New Node From Selection.'


Custom Node Properties

Custom Node Properties CN

Custom Nodes will be placed in the Add-Ons section of the library.

Name

patternFromSolidModule

Description

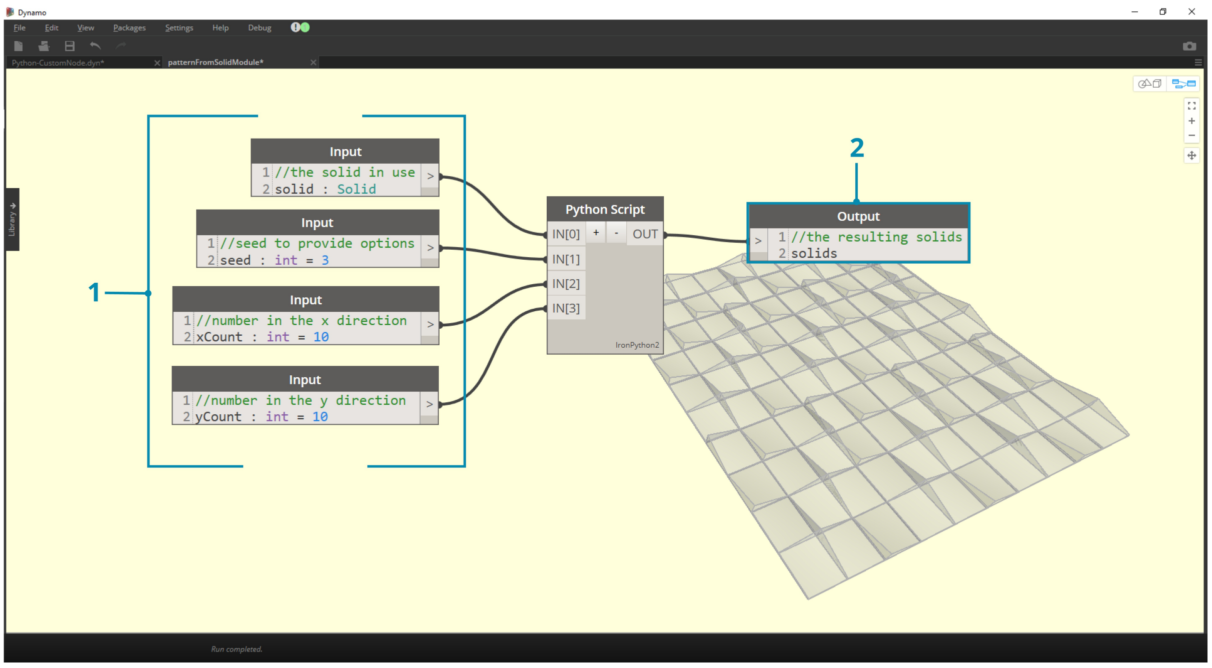
Create patterns by arraying and rotating a solid module.

Add-ons Category

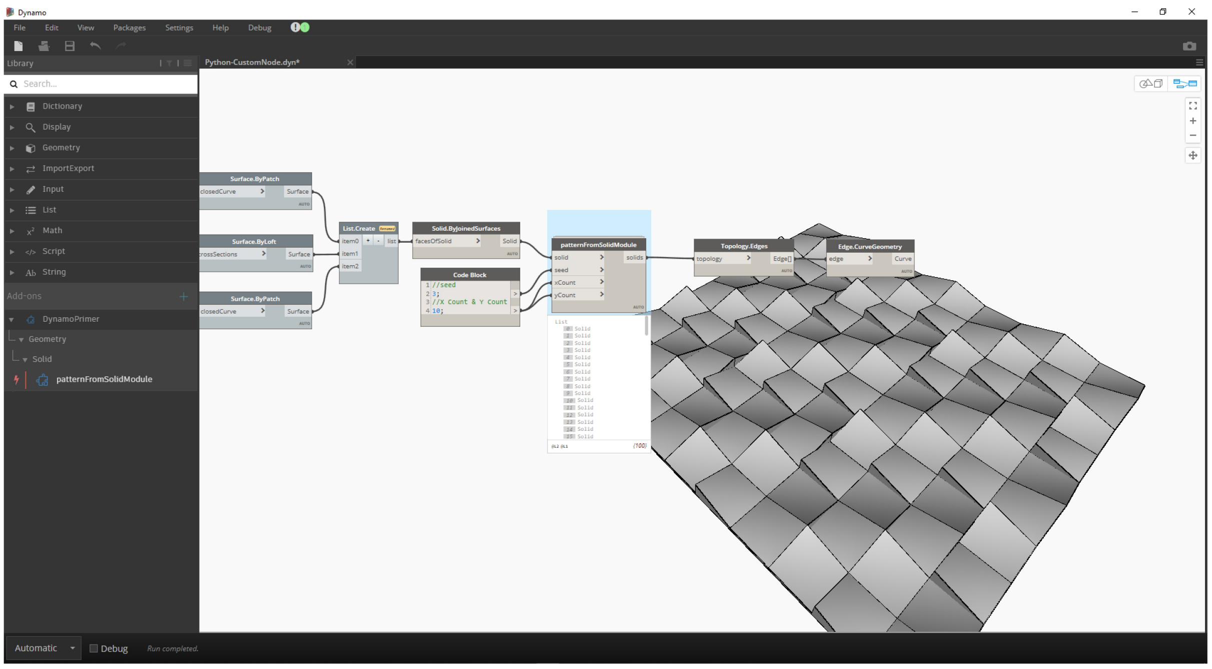
DynamoPrimer.Geometry.Solid

Assign a name, description, and category.

This will open a new workspace in which to edit the custom node.



1. **Inputs:** Change the input names to be more descriptive and add data types and default values.
2. **Output:** Change the output name Save the node as a .dyf file.



```
//seed
3;
//X Count & Y Count
10;
```

```
List
  0 Solid
  1 Solid
  2 Solid
  3 Solid
  4 Solid
  5 Solid
  6 Solid
  7 Solid
  8 Solid
  9 Solid
  10 Solid
  11 Solid
  12 Solid
  13 Solid
  14 Solid
  15 Solid
@L2 @L1                {100}
```

The custom node reflects the changes we just made.

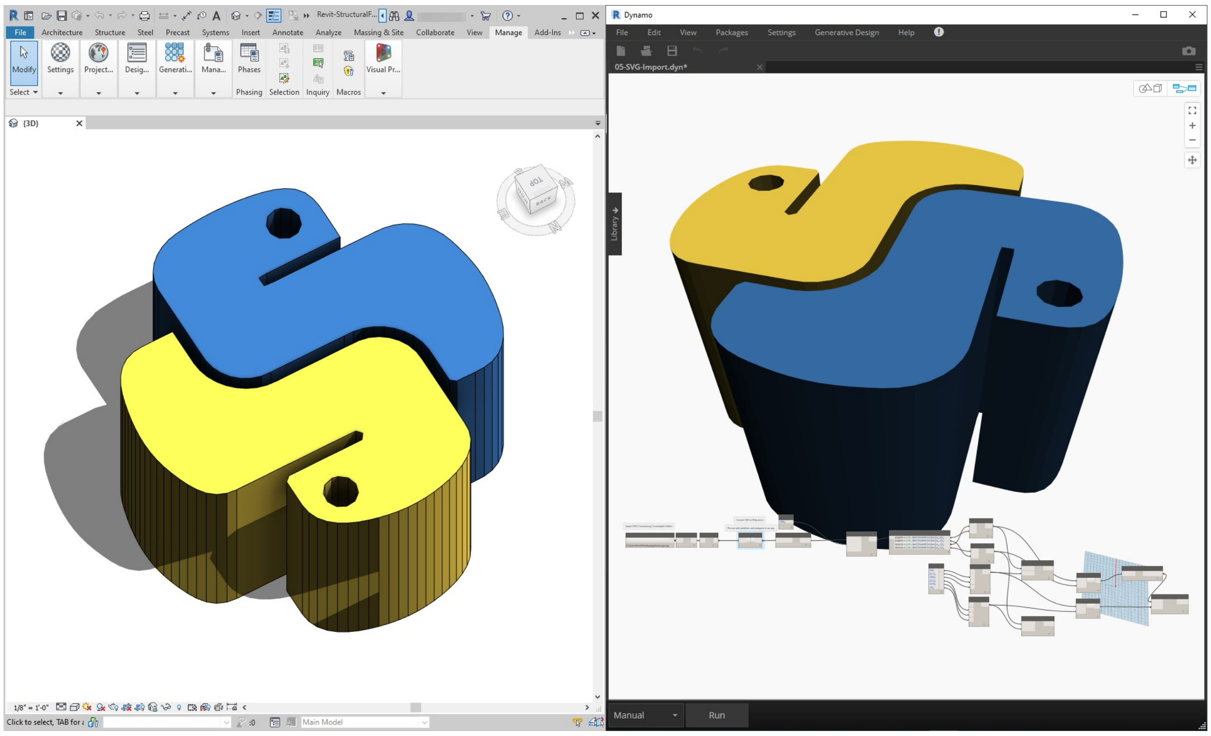
Python and Revit

Python and Revit

Now that we've demonstrated how to use Python scripts in Dynamo, let's take a look at connecting Revit libraries into the scripting environment. Remember, we imported our Dynamo core nodes with the first three lines in the block of code below. To import the Revit nodes, Revit elements, and the Revit document manager, we only have to add a few more lines:

```
import clr
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *

# Import RevitNodes
clr.AddReference("RevitNodes")
import Revit

# Import Revit elements
from Revit.Elements import *

# Import DocumentManager
clr.AddReference("RevitServices")
import RevitServices
from RevitServices.Persistence import DocumentManager

import System
```

This gives us access to the Revit API and offers custom scripting for any Revit task. By combining the process of visual programming with Revit API scripting, collaboration and tool development improve significantly. For example, a BIM manager and a schematic designer can work together on the same graph. In this collaboration, they can improve design and execution of the model.



Platform Specific APIs

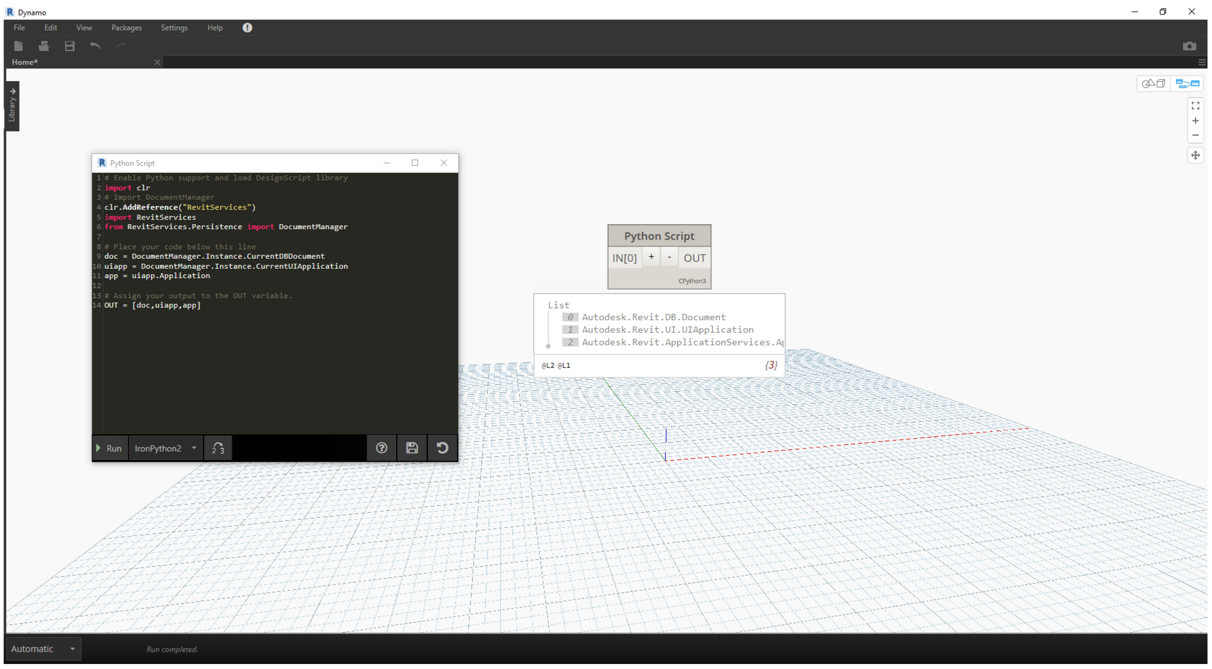
The plan behind the Dynamo Project is to widen the scope of platform implementation. As Dynamo adds more programs to the docket, users will gain access to platform-specific APIs from the Python scripting environment. While Revit is the case study for this section, we can anticipate more chapters in the future which offer comprehensive tutorials on scripting in other platforms. Additionally, there are many IronPython libraries accessible now which can be imported into Dynamo!

The examples below demonstrate ways to implement Revit-specific operations from Dynamo using Python. For a more detailed review on Python's relationship to Dynamo and Revit, refer to the Dynamo Wiki page. Another useful resource for Python and Revit is the Revit Python Shell Project.

Exercise 01

> Create a new Revit Project. Download the example file that accompanies this exercise (Right click and "Save Link As..."). A full list of example files can be found in the Appendix. Revit-Doc.dyn

In these exercises, we'll explore elementary Python scripts in Dynamo for Revit. The exercise will focus on dealing with Revit files and elements, as well as the communication between Revit and Dynamo.

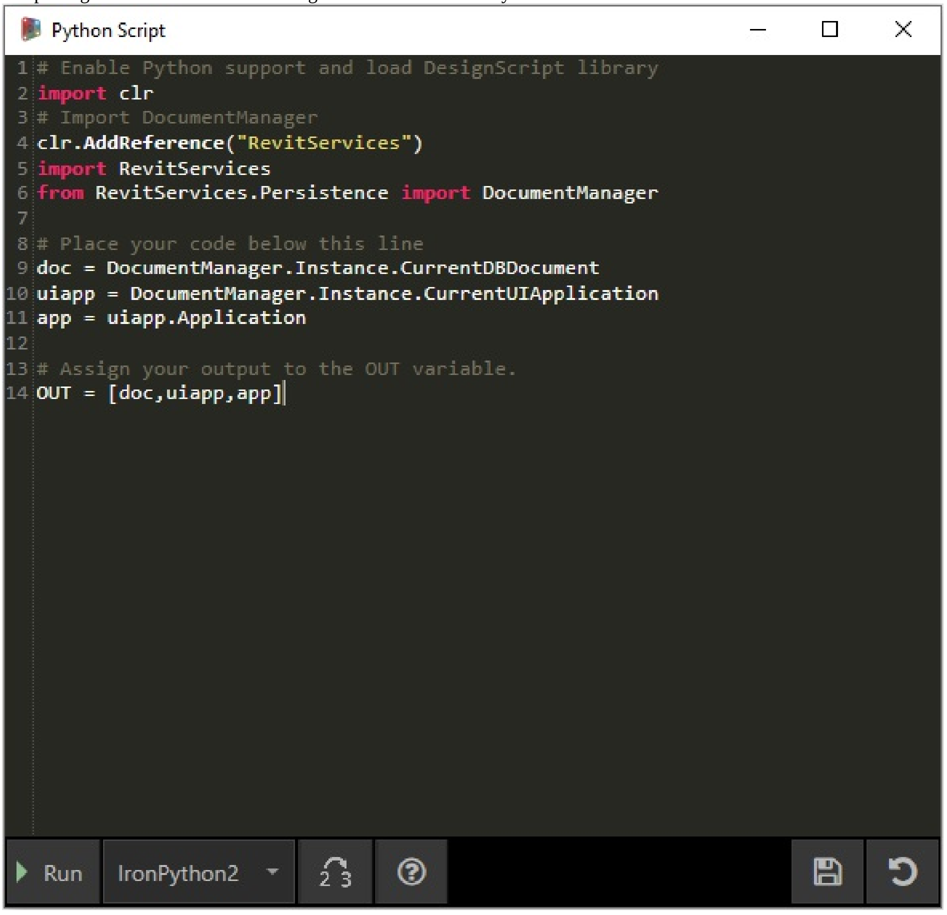


```
# Enable Python support and load DesignScript library
import clr
# Import DocumentManager
clr.AddReference("RevitServices")
import RevitServices
from RevitServices.Persistence import DocumentManager

# Place your code below this line
doc = DocumentManager.Instance.CurrentDBDocument
uiapp = DocumentManager.Instance.CurrentUIApplication
app = uiapp.Application

# Assign your output to the OUT variable.
OUT = [doc,uiapp,app]
```

```
List
  0 Autodesk.Revit.DB.Document
  1 Autodesk.Revit.UI.UIApplication
  2 Autodesk.Revit.ApplicationServices.A
@L2 @L1                              {3}
```

This is a cut and dry method for retrieving the *doc*, *uiapp*, and *app* of the Revit file linked to your Dynamo sesson. Programmers who have worked in the Revit API before may notice the items in the watch list. If these items do not look familiar, that's okay; we'll be using other examples in the exercises below.

Here is how we're importing Revit Services and retrieving the document data in Dynamo:



```
# Enable Python support and load DesignScript library
import clr
# Import DocumentManager
clr.AddReference("RevitServices")
import RevitServices
from RevitServices.Persistence import DocumentManager

# Place your code below this line
doc = DocumentManager.Instance.CurrentDBDocument
uiapp = DocumentManager.Instance.CurrentUIApplication
app = uiapp.Application

# Assign your output to the OUT variable.
OUT = [doc,uiapp,app]
```

A look at the Python node in Dynamo. The commented code is below.

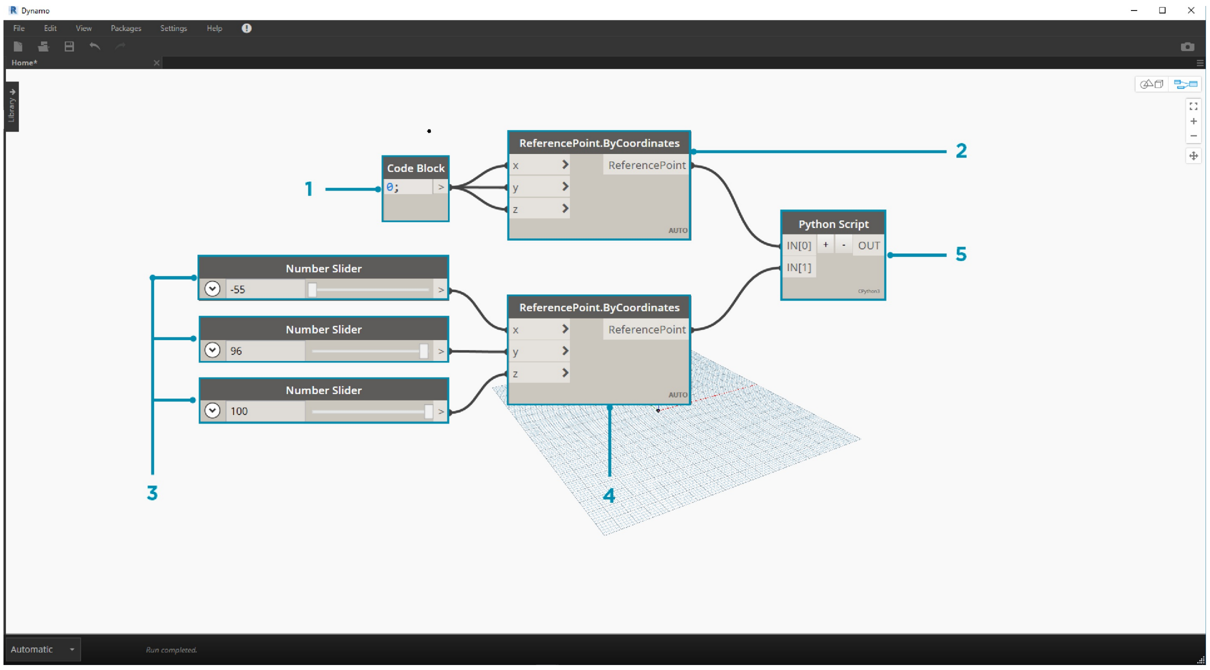
```
# Enable Python support and load DesignScript library
import clr
# Import DocumentManager
clr.AddReference("RevitServices")
```

```
import RevitServices
from RevitServices.Persistence import DocumentManager

# Place your code below this line
doc = DocumentManager.Instance.CurrentDBDocument
uiapp = DocumentManager.Instance.CurrentUIApplication
app = uiapp.Application

# Assign your output to the OUT variable.
OUT = [doc,uiapp,app]
```

Exercise 02

> Download the example files that accompanies this exercise (Right click and "Save Link As..."). A full list of example files can be found in the Appendix. Revit-ReferenceCurve.dyn

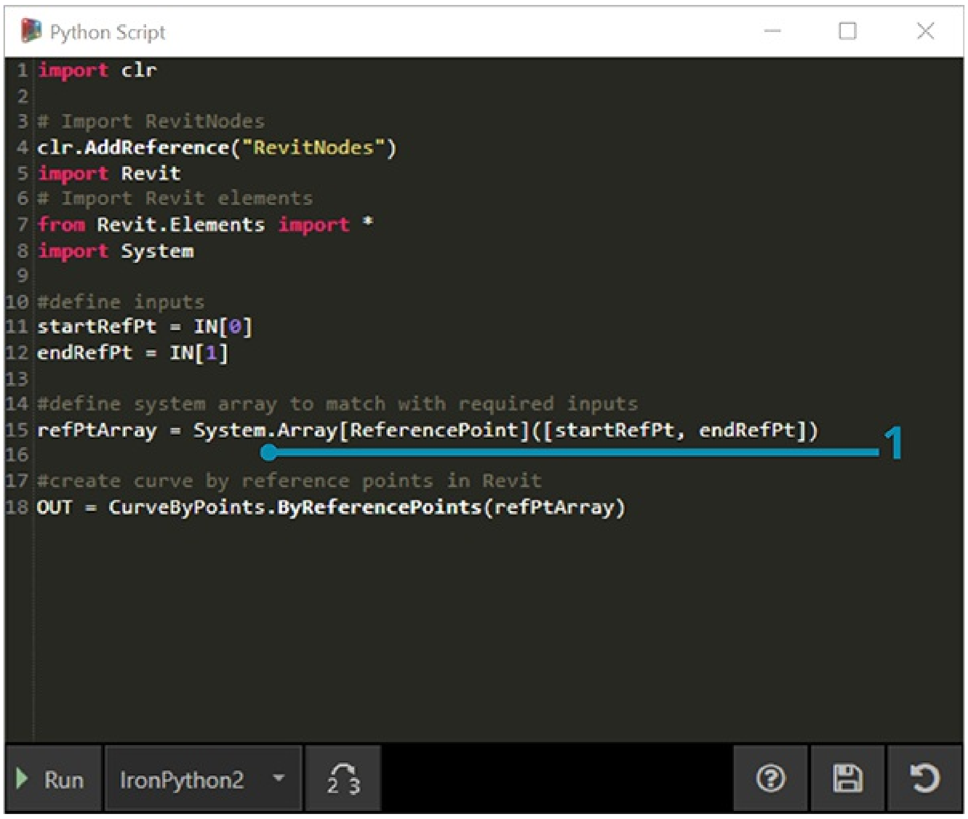
In this exercise, we'll make a simple Model Curve in Revit using the Dynamo Python node.



Begin with the set of nodes in the image above. We'll first create two reference points in Revit from Dynamo nodes.

Begin by creating a new Conceptual Mass family in Revit. Launch Dynamo and create the set of nodes in the image above. We'll first create two reference point in Revit from Dynamo nodes.

1. Create a code block and give it a value of "0;".
2. Plug this value into a ReferencePoint.ByCoordinates node for X,Y, and Z inputs.
3. Create three sliders, ranging from -100 to 100 with a step size of 1.
4. Connect each slider to a ReferencePoint.ByCoordinates node.
5. Add a Python node to the workspace, click the "+" button on the node to add another input and plug the two references points into each input. Open the Python node.

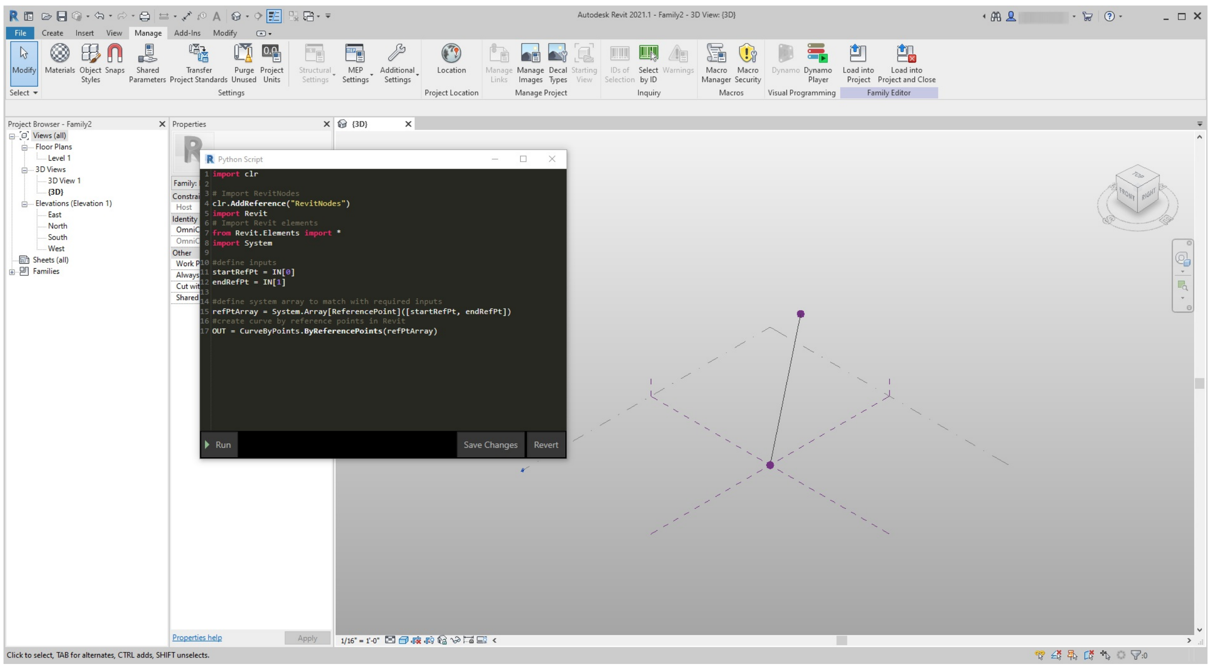


```
import clr

# Import RevitNodes
clr.AddReference("RevitNodes")
import Revit
# Import Revit elements
from Revit.Elements import *
import System

#define inputs
startRefPt = IN[0]
endRefPt = IN[1]

#define system array to match with required inputs
refPtArray = System.Array[ReferencePoint]([startRefPt, endRefPt])

#create curve by reference points in Revit
OUT = CurveByPoints.ByReferencePoints(refPtArray)
```

A look at the Python node in Dynamo. The commented code is below.

1. **System.Array:** Revit needs a System Array as an input (rather than a Python list). This is just one more line of code, but paying attention to argument types will facilitate Python programming in Revit.

```
import clr

# Import RevitNodes
clr.AddReference("RevitNodes")
import Revit
# Import Revit elements
from Revit.Elements import *
import System

#define inputs
startRefPt = IN[0]
endRefPt = IN[1]

#define system array to match with required inputs
refPtArray = System.Array[ReferencePoint]([startRefPt, endRefPt])
#create curve by reference points in Revit
OUT = CurveByPoints.ByReferencePoints(refPtArray)
```



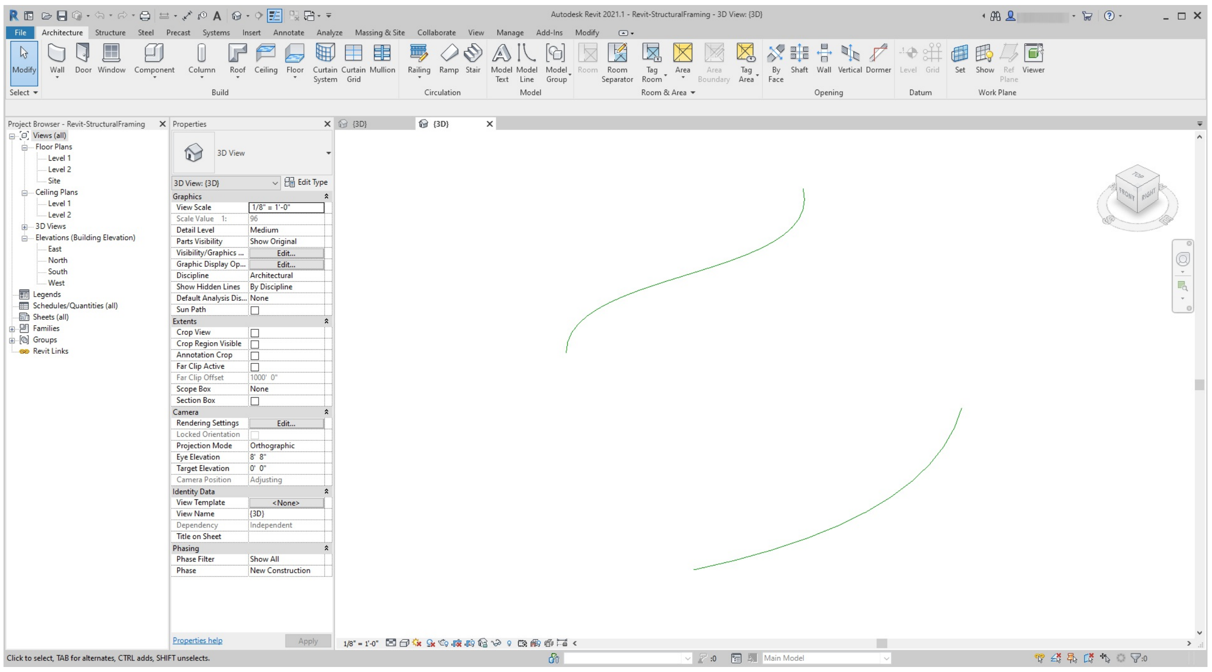
```
import clr

# Import RevitNodes
clr.AddReference("RevitNodes")
import Revit
# Import Revit elements
from Revit.Elements import *
import System

#define inputs
startRefPt = IN[0]
endRefPt = IN[1]

#define system array to match with required inputs
refPtArray = System.Array[ReferencePoint]([startRefPt, endRefPt])
#create curve by reference points in Revit
OUT = CurveByPoints.ByReferencePoints(refPtArray)
```

From Dynamo, we've created two reference points with a line connecting them using Python. Let's take this a little further in the next exercise.

Exercise 03

Download and unzip the example files that accompany this exercise (Right click and "Save Link As..."). A full list of example files can be found in the Appendix. Revit-StructuralFraming.zip

This exercise keeps it simple, but drives home the topics of connecting data and geometry from Revit to Dynamo and back. Let's begin by opening Revit-StructuralFraming.rvt. Once opened, load Dynamo and open the file Revit-StructuralFraming.dyn.



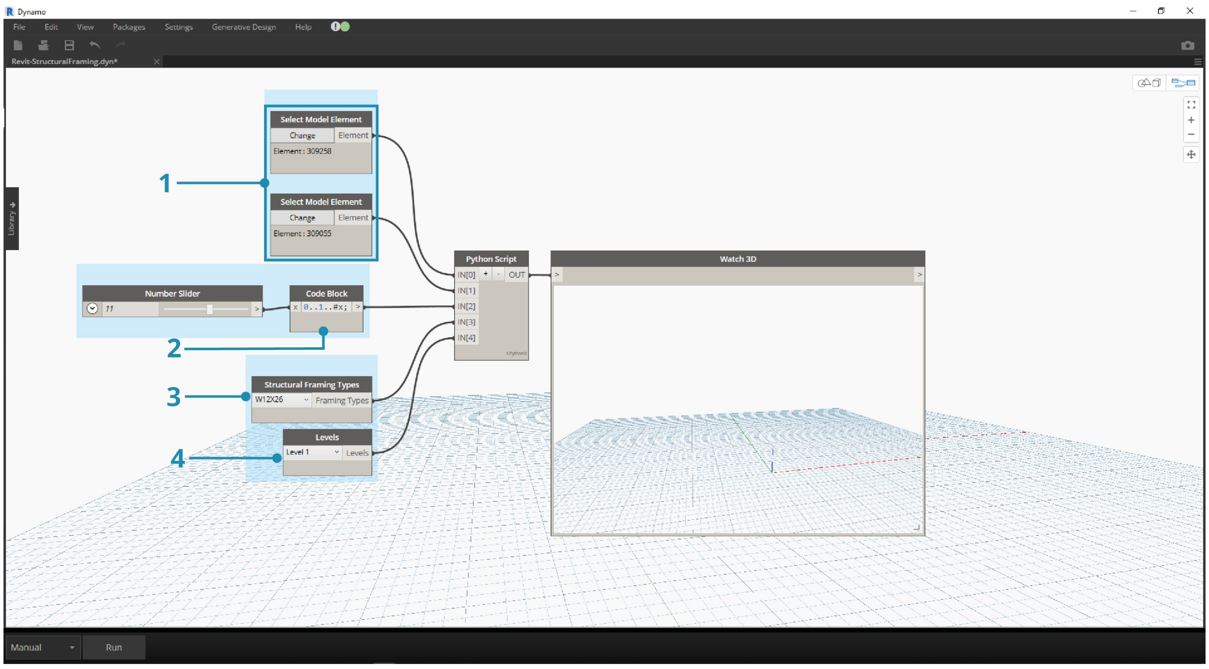
Properties

3D View

3D View: {3D} Edit Type

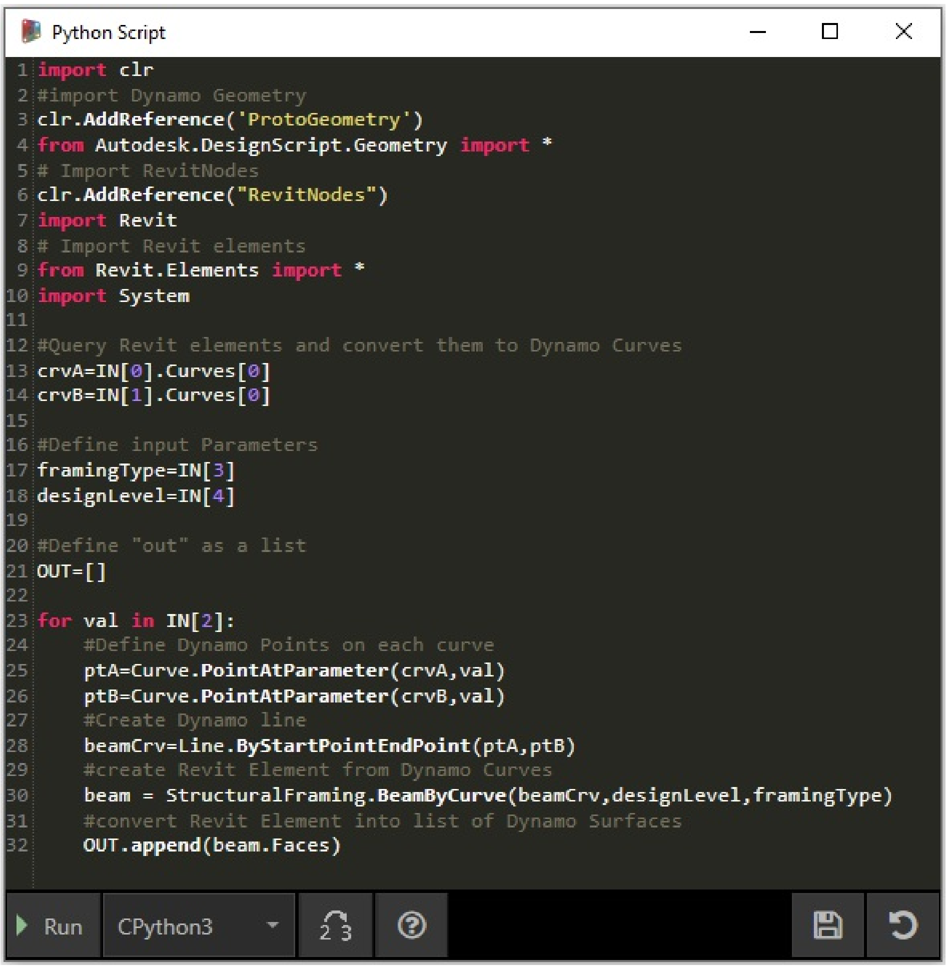
Graphics	
View Scale	1/8" = 1'-0"
Scale Value 1:	96
Detail Level	Medium
Parts Visibility	Show Original
Visibility/Graphics ...	Edit...
Graphic Display Op...	Edit...
Discipline	Architectural
Show Hidden Lines	By Discipline
Default Analysis Dis...	None
Sun Path	☐
Extents	
Crop View	☐
Crop Region Visible	☐
Annotation Crop	☐
Far Clip Active	☐
Far Clip Offset	1000' 0"
Scope Box	None
Section Box	☐
Camera	
Rendering Settings	Edit...
Locked Orientation	☐
Projection Mode	Orthographic
Eye Elevation	8' 8"
Target Elevation	0' 0"
Camera Position	Adjusting
Identity Data	
View Template	<None>
View Name	{3D}
Dependency	Independent
Title on Sheet	
Phasing	
Phase Filter	Show All
Phase	New Construction

This Revit file is about as basic as it gets. Two reference curves: one drawn on Level 1 and the other drawn on Level 2. We want to get these curves into Dynamo and maintain a live link.



In this file we have a set of nodes plugging into five inputs of a Python node.

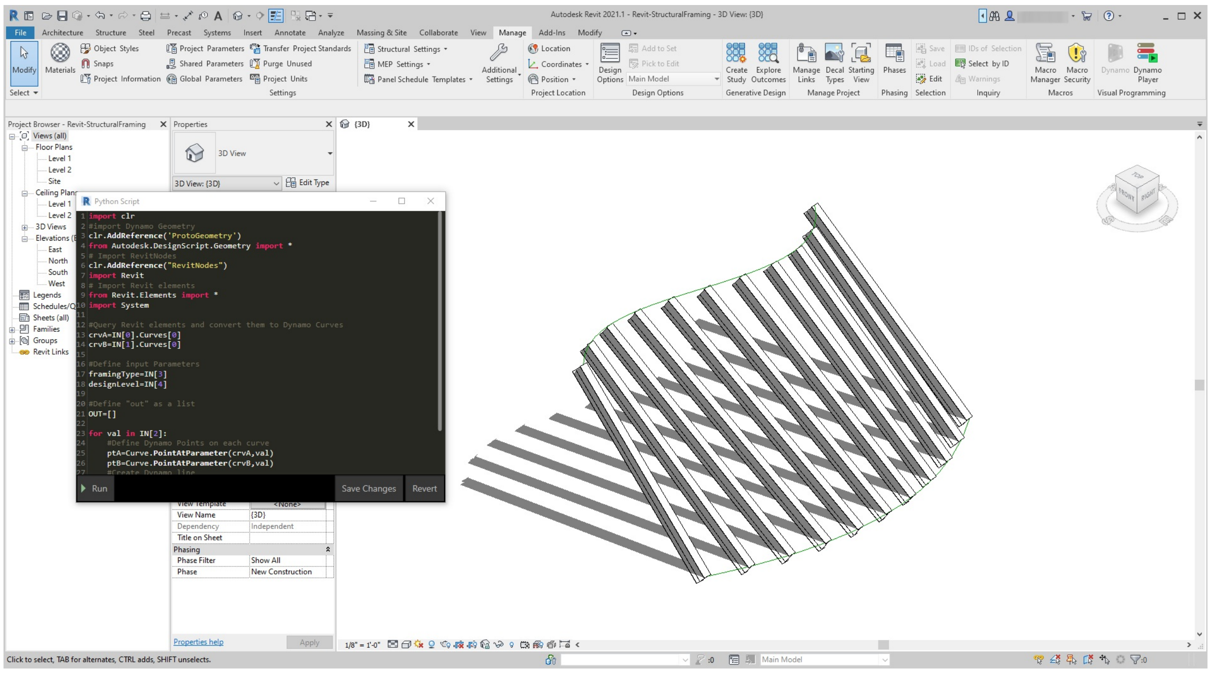
1. **Select Model Element Nodes:** Hit the select button for each and select a corresponding curve in Revit.
2. **Code Block:** using the syntax *"0..1..#x;"*, connect an integer slider ranging from 0 to 20 into the *x* input. This designates the number of beams to draw between the two curves.
3. **Structural Framing Types:** We'll choose the default W12x26 beam here from the dropdown menu.
4. **Levels:** select "Level 1".



```python
import clr
#import Dynamo Geometry
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *
# Import RevitNodes
clr.AddReference("RevitNodes")
import Revit
# Import Revit elements
from Revit.Elements import *
import System

#Query Revit elements and convert them to Dynamo Curves
crvA=IN[0].Curves[0]
crvB=IN[1].Curves[0]

#Define input Parameters
framingType=IN[3]
designLevel=IN[4]

#Define "out" as a list
OUT=[]

for val in IN[2]:
    #Define Dynamo Points on each curve
    ptA=Curve.PointAtParameter(crvA,val)
    ptB=Curve.PointAtParameter(crvB,val)
    #Create Dynamo line
    beamCrv=Line.ByStartPointEndPoint(ptA,ptB)
    #create Revit Element from Dynamo Curves
    beam = StructuralFraming.BeamByCurve(beamCrv,designLevel,framingType)
    #convert Revit Element into list of Dynamo Surfaces
    OUT.append(beam.Faces)
```

This code in Python is a little more dense, but the comments within the code describe what's happening in the process:

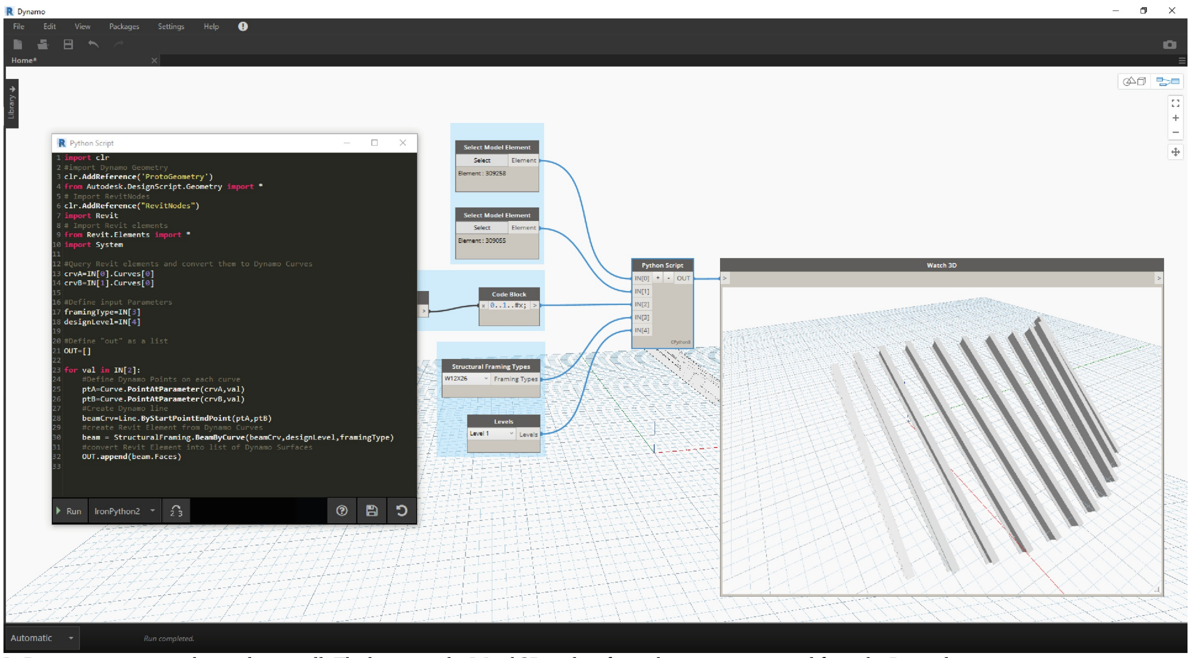
```
import clr
#import Dynamo Geometry
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *
# Import RevitNodes
clr.AddReference("RevitNodes")
import Revit
# Import Revit elements
from Revit.Elements import *
import System

#Query Revit elements and convert them to Dynamo Curves
crvA=IN[0].Curves[0]
crvB=IN[1].Curves[0]

#Define input Parameters
framingType=IN[3]
designLevel=IN[4]

#Define "out" as a list
OUT=[]

for val in IN[2]:
    #Define Dynamo Points on each curve
    ptA=Curve.PointAtParameter(crvA,val)
    ptB=Curve.PointAtParameter(crvB,val)
    #Create Dynamo line
    beamCrv=Line.ByStartPointEndPoint(ptA,ptB)
    #create Revit Element from Dynamo Curves
    beam = StructuralFraming.BeamByCurve(beamCrv,designLevel,framingType)
    #convert Revit Element into list of Dynamo Surfaces
    OUT.append(beam.Faces)
```



```
import clr
#import Dynamo Geometry
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *
# Import RevitNodes
clr.AddReference("RevitNodes")
import Revit
# Import Revit elements
from Revit.Elements import *
import System

#Query Revit elements and convert them to Dynamo Curves
crvA=IN[0].Curves[0]
crvB=IN[1].Curves[0]

#Define input Parameters
framingType=IN[3]
designLevel=IN[4]

#Define "out" as a list
OUT=[]

for val in IN[2]:
    #Define Dynamo Points on each curve
    ptA=Curve.PointAtParameter(crvA,val)
    ptB=Curve.PointAtParameter(crvB,val)
```

View Name	{3D}
Dependency	Independent
Title on Sheet	
Phase Filter	Show All
Phase	New Construction

In Revit, we have an array of beams spanning the two curves as structural elements. Note: this isn't a realistic example...the structural elements are used as an example for native Revit instances created from Dynamo.



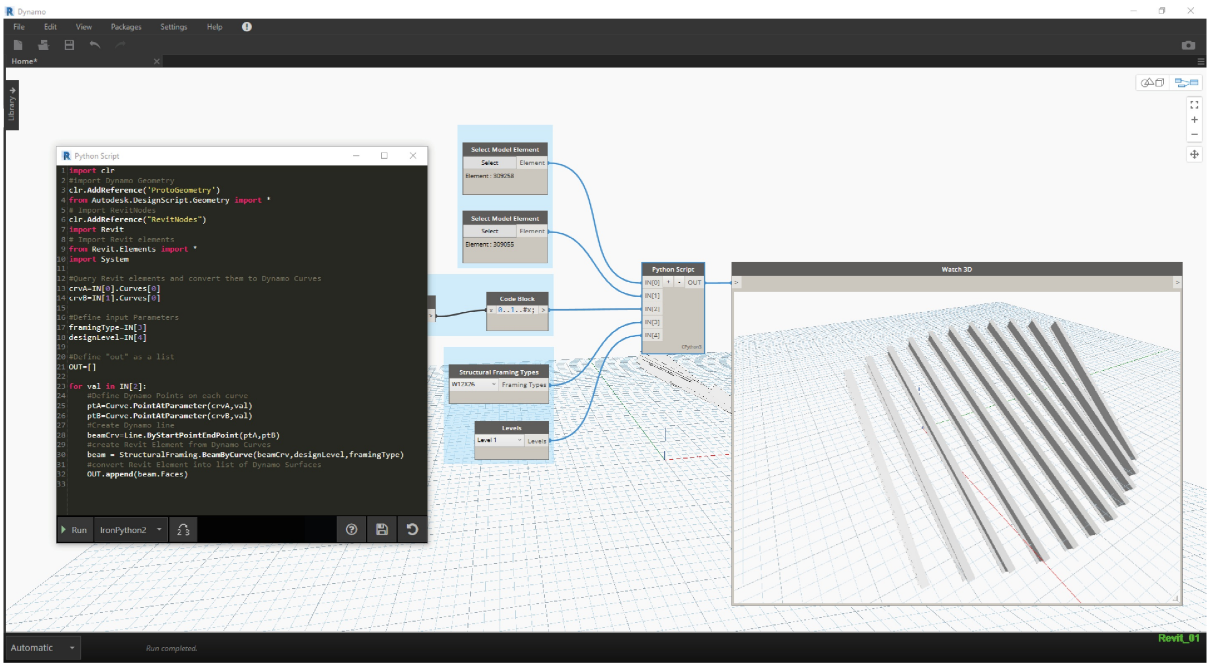
```python
import clr
#import Dynamo Geometry
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *
# Import RevitNodes
clr.AddReference("RevitNodes")
import Revit
# Import Revit elements
from Revit.Elements import *
import System

#Query Revit elements and convert them to Dynamo Curves
crvA=IN[0].Curves[0]
crvB=IN[1].Curves[0]

#Define input Parameters
framingType=IN[3]
designLevel=IN[4]

#Define "out" as a list
OUT=[]

for val in IN[2]:
    #Define Dynamo Points on each curve
    ptA=Curve.PointAtParameter(crvA,val)
    ptB=Curve.PointAtParameter(crvB,val)
    #Create Dynamo line
    beamCrv=Line.ByStartPointEndPoint(ptA,ptB)
    #create Revit Element from Dynamo Curves
    beam = StructuralFraming.BeamByCurve(beamCrv,designLevel,framingType)
    #convert Revit Element into list of Dynamo Surfaces
    OUT.append(beam.Faces)
```

In Dynamo, we can see the results as well. The beams in the Watch3D node refer to the geometry queried from the Revit elements.

Notice that we have a continuous process of translating data from the Revit Environment to the Dynamo Environment. In summary, here's how the process plays out:

1. Select Revit element
2. Convert Revit element to Dynamo Curve
3. Divide Dynamo curve into a series of Dynamo points
4. Use the Dynamo points between two curves to create Dynamo lines
5. Create Revit beams by referencing Dynamo lines
6. Output Dynamo surfaces by querying the geometry of Revit beams

This may sound a little heavy handed, but the script makes it as simple as editing the curve in Revit and re-running the solver (although you may have to delete the previous beams when doing so). *This is due to the fact that we are placing beams in python, thus breaking the association that OOTB nodes have.*



```python
import clr
#import Dynamo Geometry
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *
# Import RevitNodes
clr.AddReference("RevitNodes")
import Revit
# Import Revit elements
from Revit.Elements import *
import System

#Query Revit elements and convert them to Dynamo Curves
crvA=IN[0].Curves[0]
crvB=IN[1].Curves[0]

#Define input Parameters
framingType=IN[3]
designLevel=IN[4]

#Define "out" as a list
OUT=[]

for val in IN[2]:
    #Define Dynamo Points on each curve
    ptA=Curve.PointAtParameter(crvA,val)
    ptB=Curve.PointAtParameter(crvB,val)
    #Create Dynamo line
    beamCrv=Line.ByStartPointEndPoint(ptA,ptB)
    #create Revit Element from Dynamo Curves
    beam = StructuralFraming.BeamByCurve(beamCrv,designLevel,framingType)
    #convert Revit Element into list of Dynamo Surfaces
    OUT.append(beam.Faces)

```

With an update to the reference curves in Revit, we get a new array of beams.

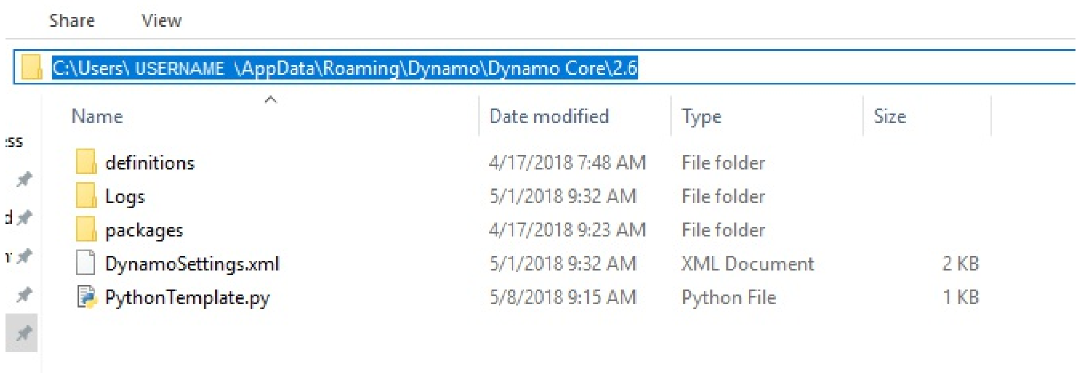
Python Templates in Dynamo 2.0

Python Templates

With Dynamo 2.0 we have the ability to specify a default template (`.py extension`) to use when opening the python window for the first time. This has been a long-desired request as this expedites the usage of Python within Dynamo. Having the ability to use a template allows us to have default imports ready to go when we want to develop a custom Python script.

The location for this template is in the `APPDATA` location for your Dynamo install.

This is typically as follows (`%appdata%\Dynamo\Dynamo Core\{version}\`).



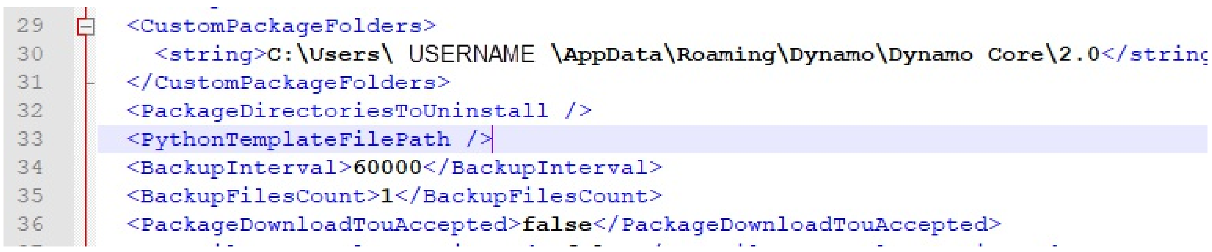
Share View

C:\Users\ USERNAME \AppData\Roaming\Dynamo\Dynamo Core\2.6

Name	Date modified	Type	Size
definitions	4/17/2018 7:48 AM	File folder	
Logs	5/1/2018 9:32 AM	File folder	
packages	4/17/2018 9:23 AM	File folder	
DynamoSettings.xml	5/1/2018 9:32 AM	XML Document	2 KB
PythonTemplate.py	5/8/2018 9:15 AM	Python File	1 KB

Setting Up The Template

In order to utilize this functionality we need to add the following line in our `DynamoSettings.xml` file. *(Edit in notepad)*



```
<CustomPackageFolders>
  <string>C:\Users\ USERNAME \AppData\Roaming\Dynamo\Dynamo Core\2.0</string
</CustomPackageFolders>
<PackageDirectoriesToUninstall />
<PythonTemplateFilePath />
<BackupInterval>60000</BackupInterval>
<BackupFilesCount>1</BackupFilesCount>
<PackageDownloadTouAccepted>false</PackageDownloadTouAccepted>
```

Where we see `<PythonTemplateFilePath />`, we can simply replace this with the following:

```
<PythonTemplateFilePath>
<string>C:\Users\CURRENTUSER\AppData\Roaming\Dynamo\Dynamo Core\2.0\PythonTemplate.py</string>
</PythonTemplateFilePath>
```

Note: replace CURRENTUSER with your username

Next we need to build a template with the functionality that we want to use built-in. In our case lets embed the Revit related imports and some of the other typical items when working with Revit.

You can start a blank notepad document and paste the following code inside:

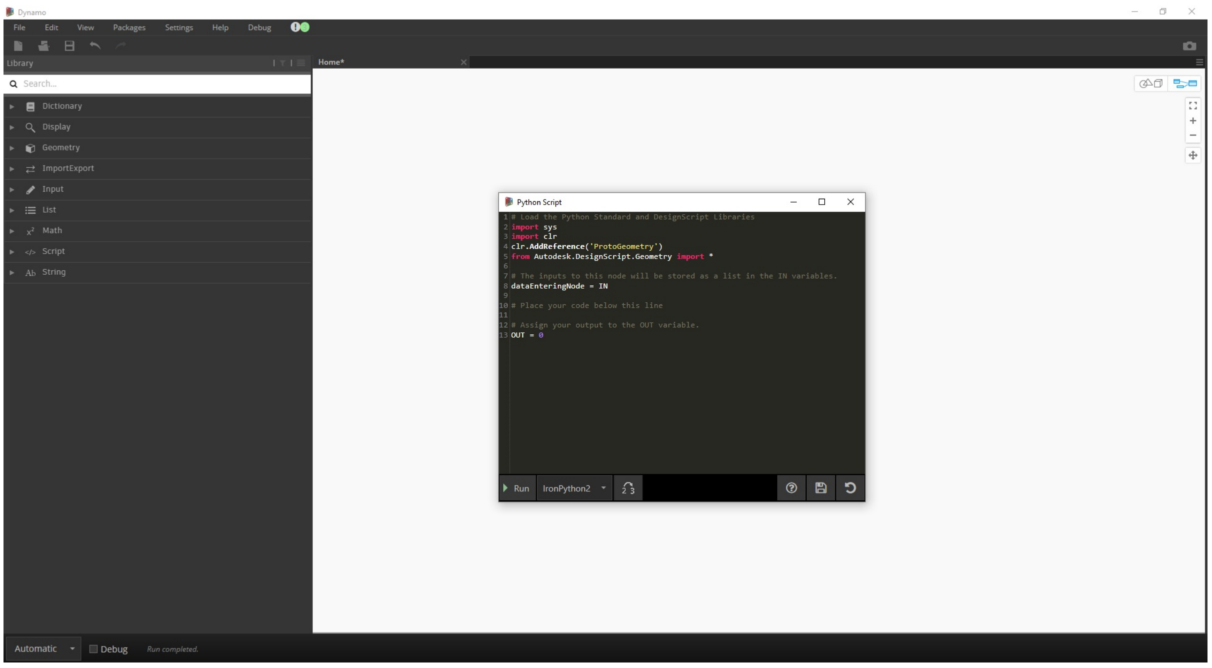
```
import clr

clr.AddReference('RevitAPI')
from Autodesk.Revit.DB import *
from Autodesk.Revit.DB.Structure import *

clr.AddReference('RevitAPIUI')
from Autodesk.Revit.UI import *

clr.AddReference('System')
from System.Collections.Generic import List

clr.AddReference('RevitNodes')
import Revit
clr.ImportExtensions(Revit.GeometryConversion)
clr.ImportExtensions(Revit.Elements)

clr.AddReference('RevitServices')
import RevitServices
from RevitServices.Persistence import DocumentManager
from RevitServices.Transactions import TransactionManager
```

```
doc = DocumentManager.Instance.CurrentDBDocument
uidoc=DocumentManager.Instance.CurrentUIApplication.ActiveUIDocument

#Preparing input from dynamo to revit
element = UnwrapElement(IN[0])

#Do some action in a Transaction
TransactionManager.Instance.EnsureInTransaction(doc)

TransactionManager.Instance.TransactionTaskDone()

OUT = element
```

Once that is done, save this file as `PythonTemplate.py` in the `APPDATA` location.

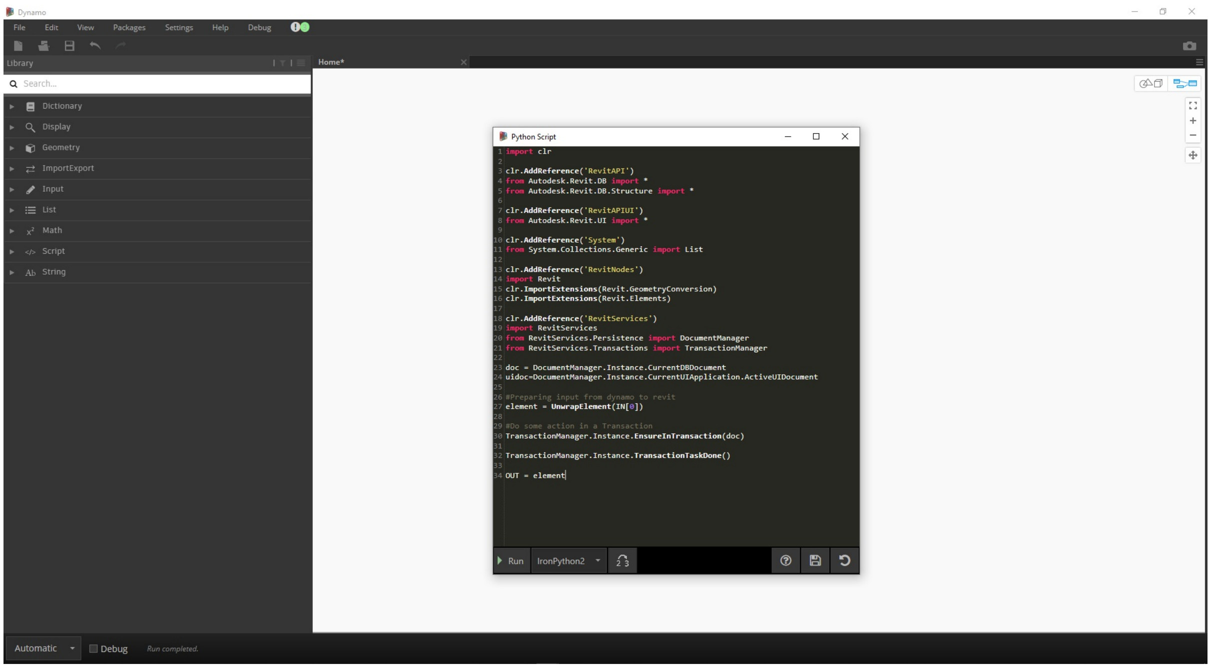
Python Script Behavior After

Ater the python template is defined, Dynamo will look for this each time a Python node is placed. If it is not found it will look like the default Python window.



```
# Load the Python Standard and DesignScript Libraries
import sys
import clr
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *

# The inputs to this node will be stored as a list in the IN variables.
dataEnteringNode = IN

# Place your code below this line

# Assign your output to the OUT variable.
OUT = 0
```

If the Python template is found (like our Revit one for example) you will see all of the default items you built in.



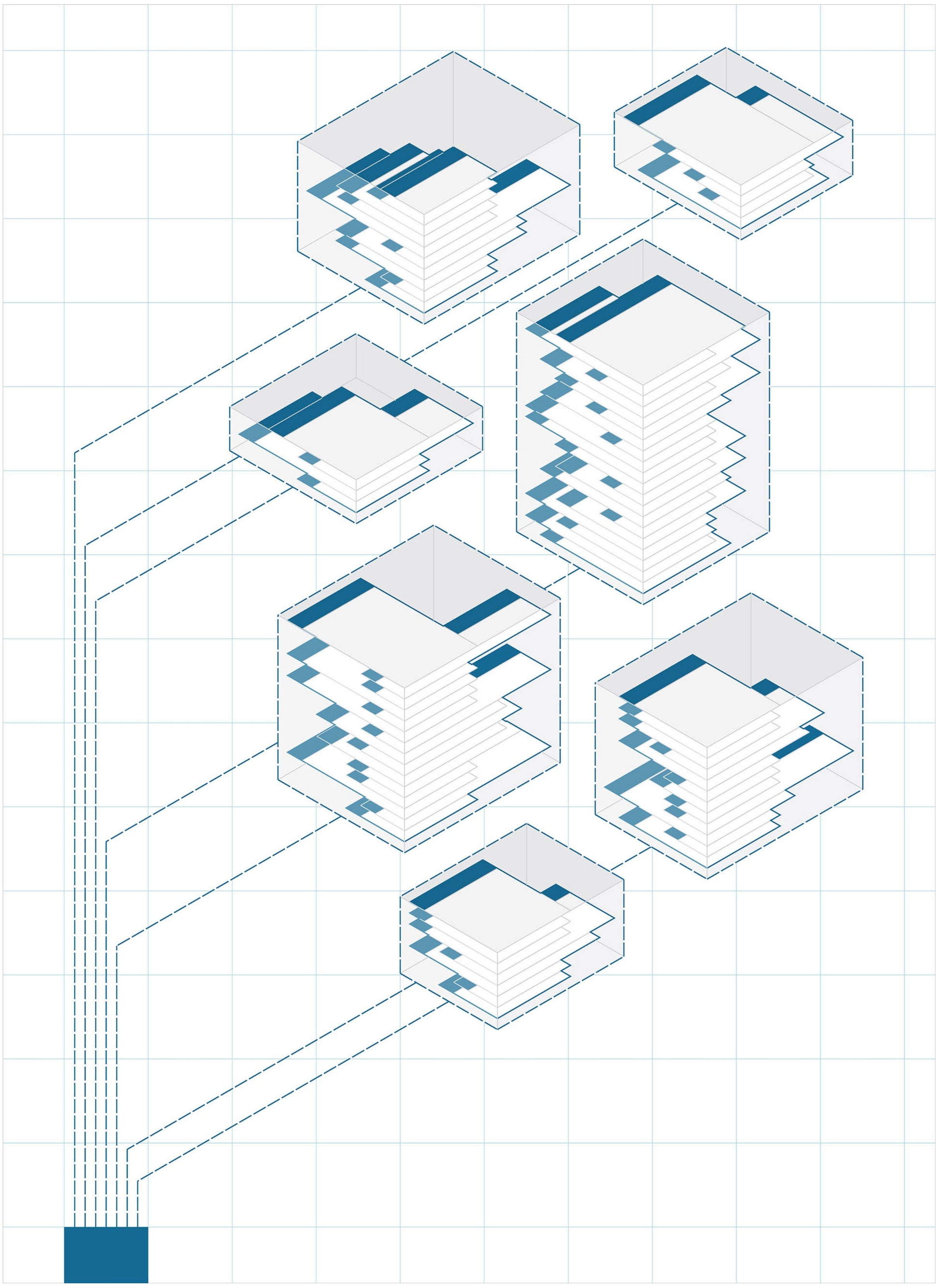
```
import clr

clr.AddReference('RevitAPI')
from Autodesk.Revit.DB import *
from Autodesk.Revit.DB.Structure import *

clr.AddReference('RevitAPIUI')
from Autodesk.Revit.UI import *

clr.AddReference('System')
from System.Collections.Generic import List

clr.AddReference('RevitNodes')
import Revit
clr.ImportExtensions(Revit.GeometryConversion)
clr.ImportExtensions(Revit.Elements)

clr.AddReference('RevitServices')
import RevitServices
from RevitServices.Persistence import DocumentManager
from RevitServices.Transactions import TransactionManager

doc = DocumentManager.Instance.CurrentDBDocument
uidoc=DocumentManager.Instance.CurrentUIApplication.ActiveUIDocument

#Preparing input from dynamo to revit
element = UnwrapElement(IN[0])

#Do some action in a Transaction
TransactionManager.Instance.EnsureInTransaction(doc)

TransactionManager.Instance.TransactionTaskDone()

OUT = element
```

Additional information regarding this great addition (by Radu Gidei) can be found here. https://github.com/DynamoDS/Dynamo/pull/8122

Packages

Packages

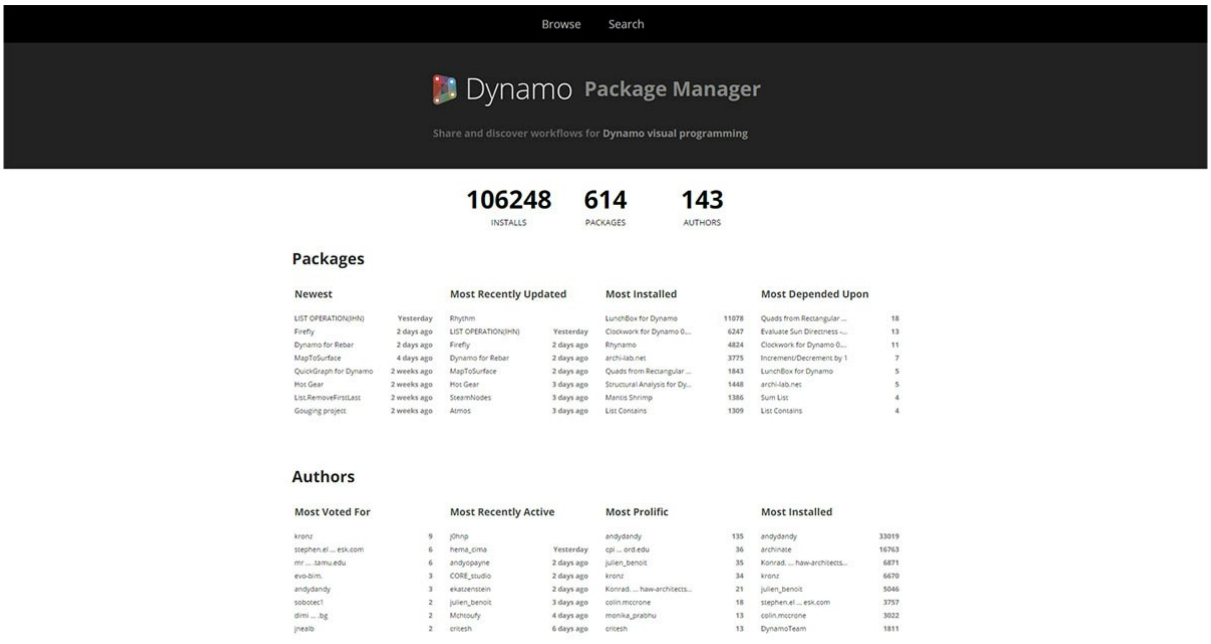
Once you have created a few Custom Nodes the very next step is to begin organizing and publishing them by way of Packages - a convenient way to store and share your nodes with the Dynamo Community.



Package Introduction

Packages

In short, a Package is a collection of Custom Nodes. The Dynamo Package Manager is a portal for the community to download any package which has been published online. These toolsets are developed by third parties in order to extend Dynamo's core functionality, accessible to all, and ready to download at the click of the button.


Browse Search

Dynamo Package Manager

Share and discover workflows for Dynamo visual programming

106248 INSTALLS 614 PACKAGES 143 AUTHORS

Packages

Newest		Most Recently Updated		Most Installed		Most Depended Upon	
LIST OPERATION(IHN)	Yesterday	Rhythm		LunchBox for Dynamo	11078	Quads from Rectangular ...	18
Firefly	2 days ago	LIST OPERATION(IHN)	Yesterday	Clockwork for Dynamo 0...	6247	Evaluate Sun Directness -...	13
Dynamo for Rebar	2 days ago	Firefly	2 days ago	Rhynamo	4824	Clockwork for Dynamo 0...	11
MapToSurface	4 days ago	Dynamo for Rebar	2 days ago	archi-lab.net	3775	Increment/Decrement by 1	7
QuickGraph for Dynamo	2 weeks ago	MapToSurface	2 days ago	Quads from Rectangular ...	1843	LunchBox for Dynamo	5
Hot Gear	2 weeks ago	Hot Gear	3 days ago	Structural Analysis for Dy...	1448	archi-lab.net	5
List.RemoveFirstLast	2 weeks ago	SteamNodes	3 days ago	Mantis Shrimp	1386	Sum List	4
Gouging project	2 weeks ago	Atmos	3 days ago	List Contains	1309	List Contains	4

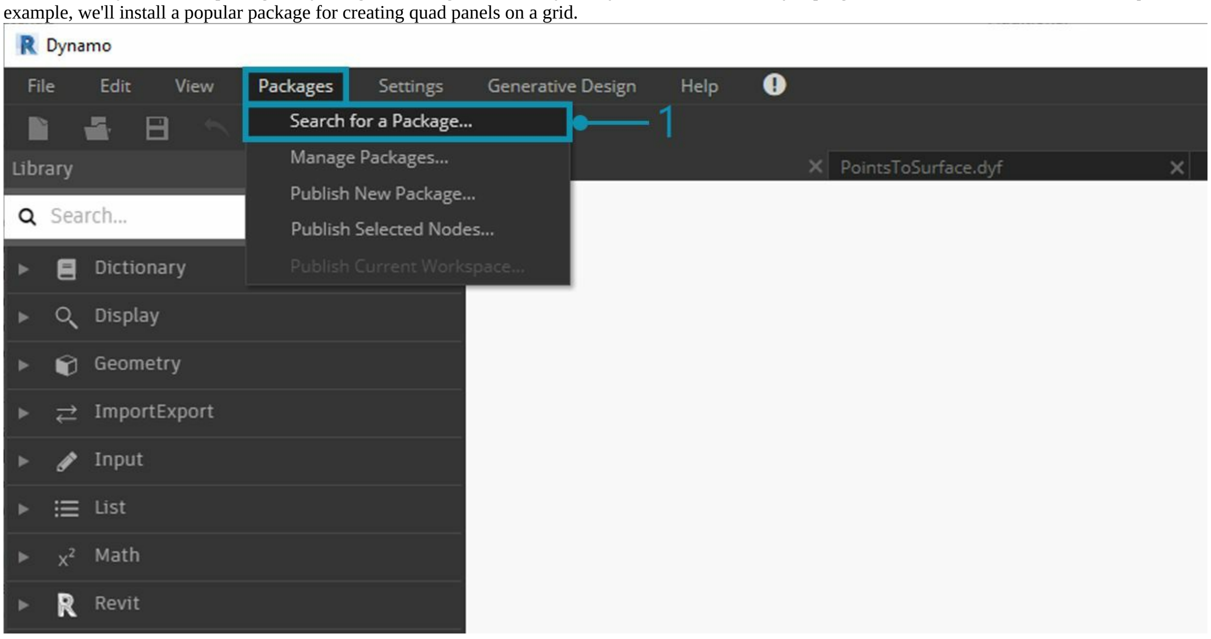
Authors

Most Voted For		Most Recently Active		Most Prolific		Most Installed	
kronz	9	j0hnp		andydandy	135	andydandy	33019
stephen.el ... esk.com	6	hema_cima	Yesterday	cpi ... ord.edu	36	archinate	16763
mr ... .tamu.edu	6	andyopayne	2 days ago	julien_benoit	35	Konrad. ... haw-architects...	6871
evo-bim.	3	CORE_studio	2 days ago	kronz	34	kronz	6670
andydandy	3	ekatzenstein	2 days ago	Konrad. ... haw-architects...	21	julien_benoit	5046
sobotec1	2	julien_benoit	3 days ago	colin.mccrone	18	stephen.el ... esk.com	3757
dimi ... .bg	2	Mchtoufy	4 days ago	monika_prabhu	13	colin.mccrone	3022
jnealb	2	critesh	6 days ago	critesh	13	DynamoTeam	1811

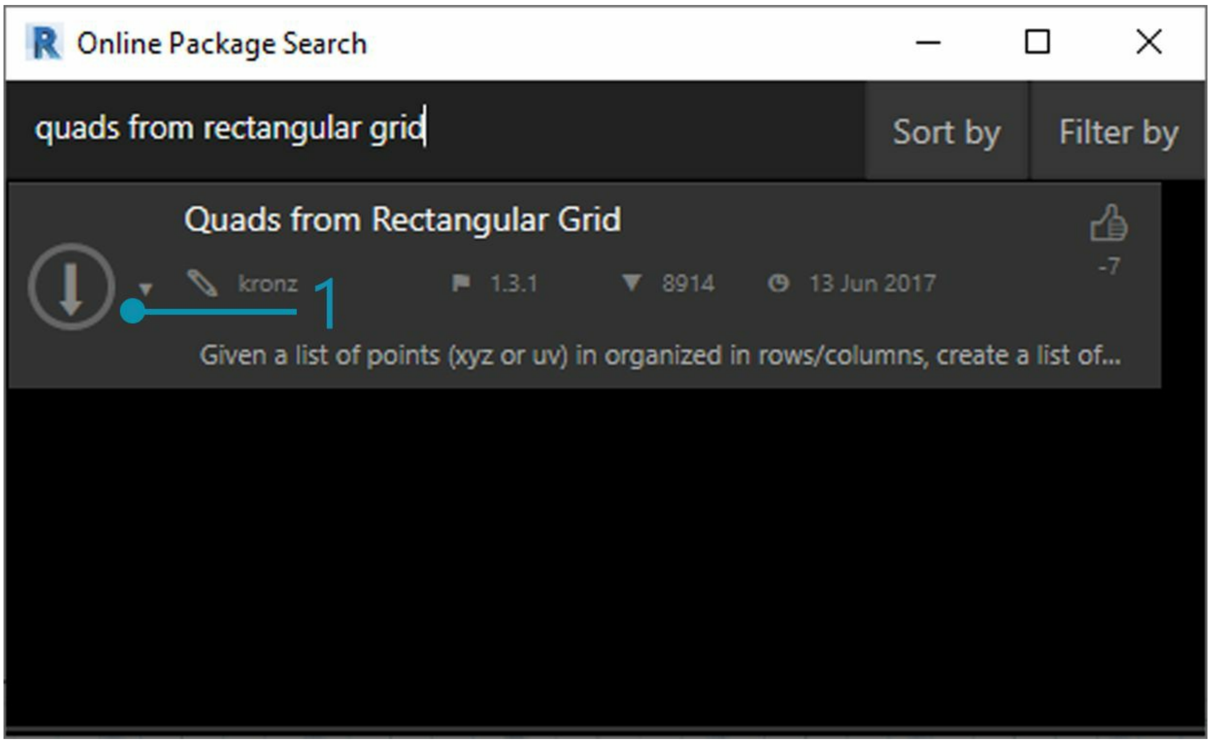
An open-source project such as Dynamo thrives on this type of community involvement. With dedicated third party developers, Dynamo is able to extend its reach to workflows across a range of industries. For this reason, the Dynamo team has made concerted efforts to streamline package development and publishing (which will be discussed in more detail in the following sections).

Installing a Package

The easiest way to install a package is by using the Packages toolbar in your Dynamo interface. Let's jump right into it and install one now. In this quick example, we'll install a popular package for creating quad panels on a grid.

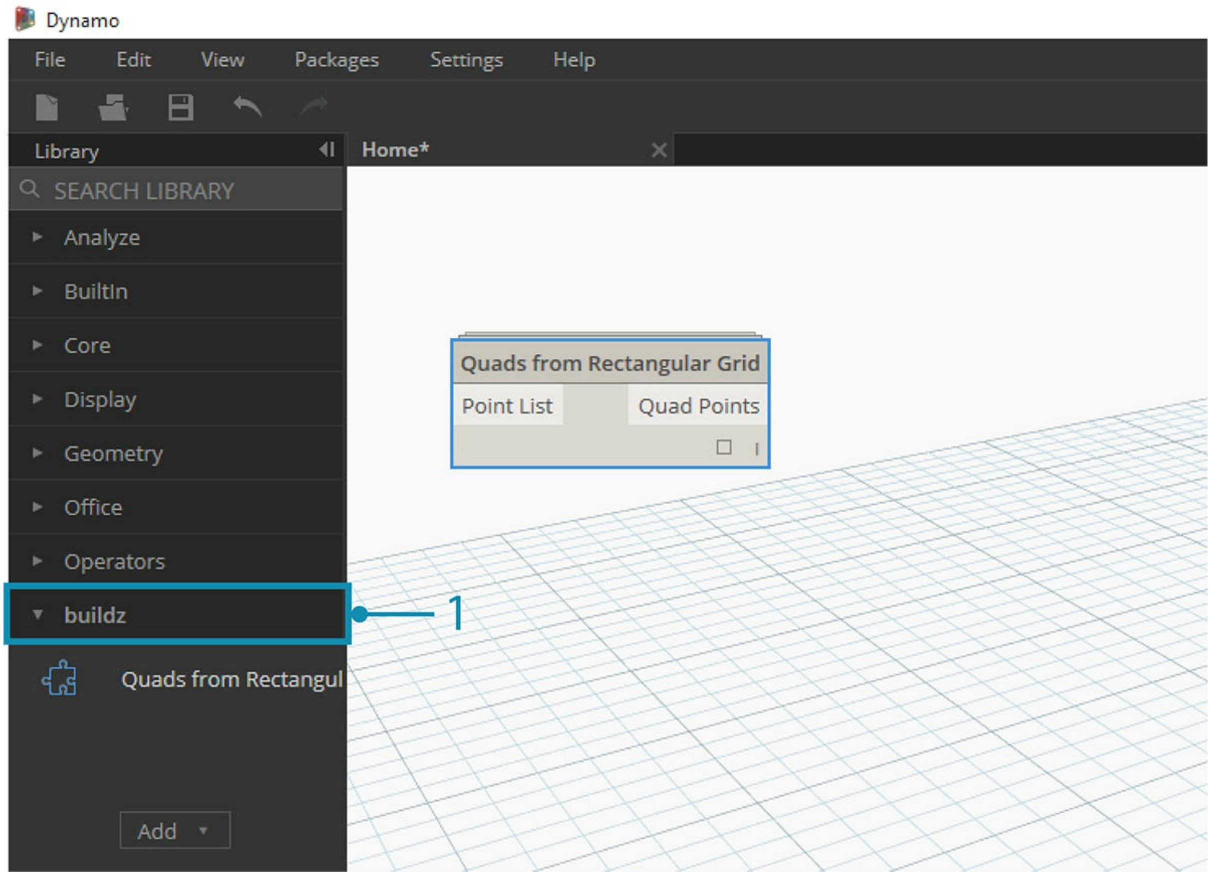


1. In Dynamo, go to *Packages>Search For a Package...*

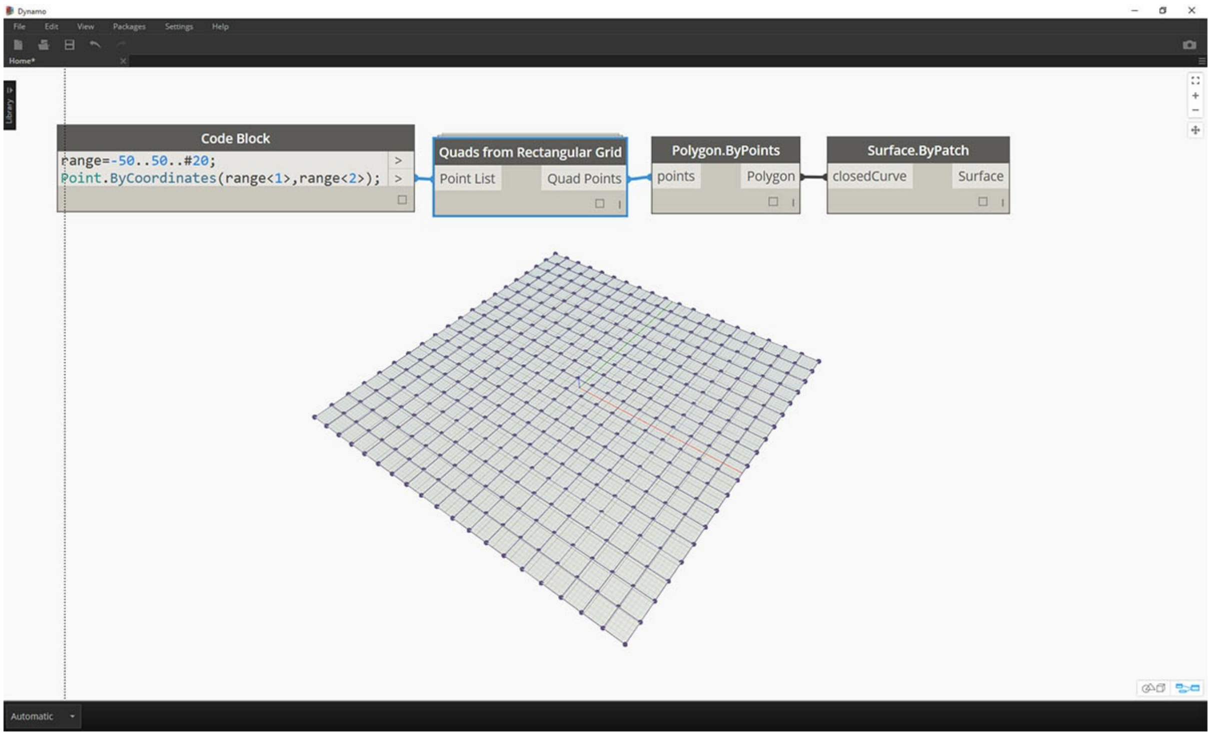

Online Package Search
quads from rectangular grid
Sort by
Filter by
Quads from Rectangular Grid
kronz
1
1.3.1
8914
13 Jun 2017
-7
Given a list of points (xyz or uv) in organized in rows/columns, create a list of...

In the search bar, let's search for "quads from rectangular grid". After a few moments, you should see all of the packages which match this search query. We want to select the first package with the matching name.

1. Click on the download arrow to the left of the package name and the package will install. Done!



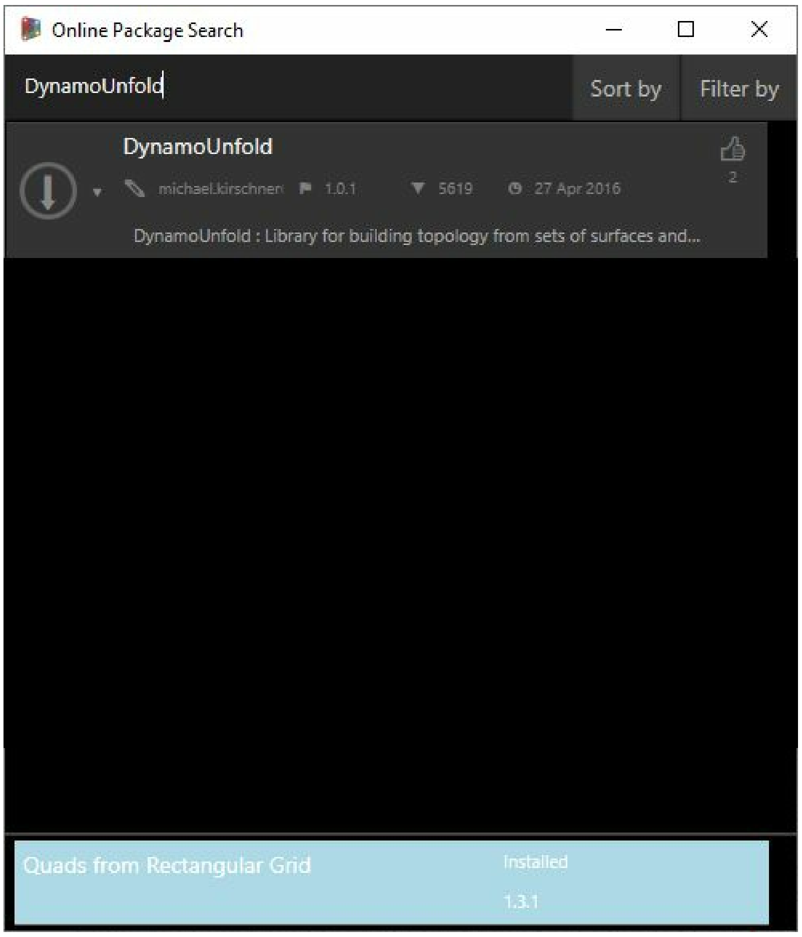
1. Notice that we now have another group in our Dynamo library called *"buildz"*. This name refers to the developer of the package, and the custom node is placed in this group. We can begin to use this right away.


```
range=-50..50..#20;
Point.ByCoordinates(range<1>,range<2>);
```

With a quick code block operation to define a rectangular grid, we've create a list of rectangular panels.

Package Folders

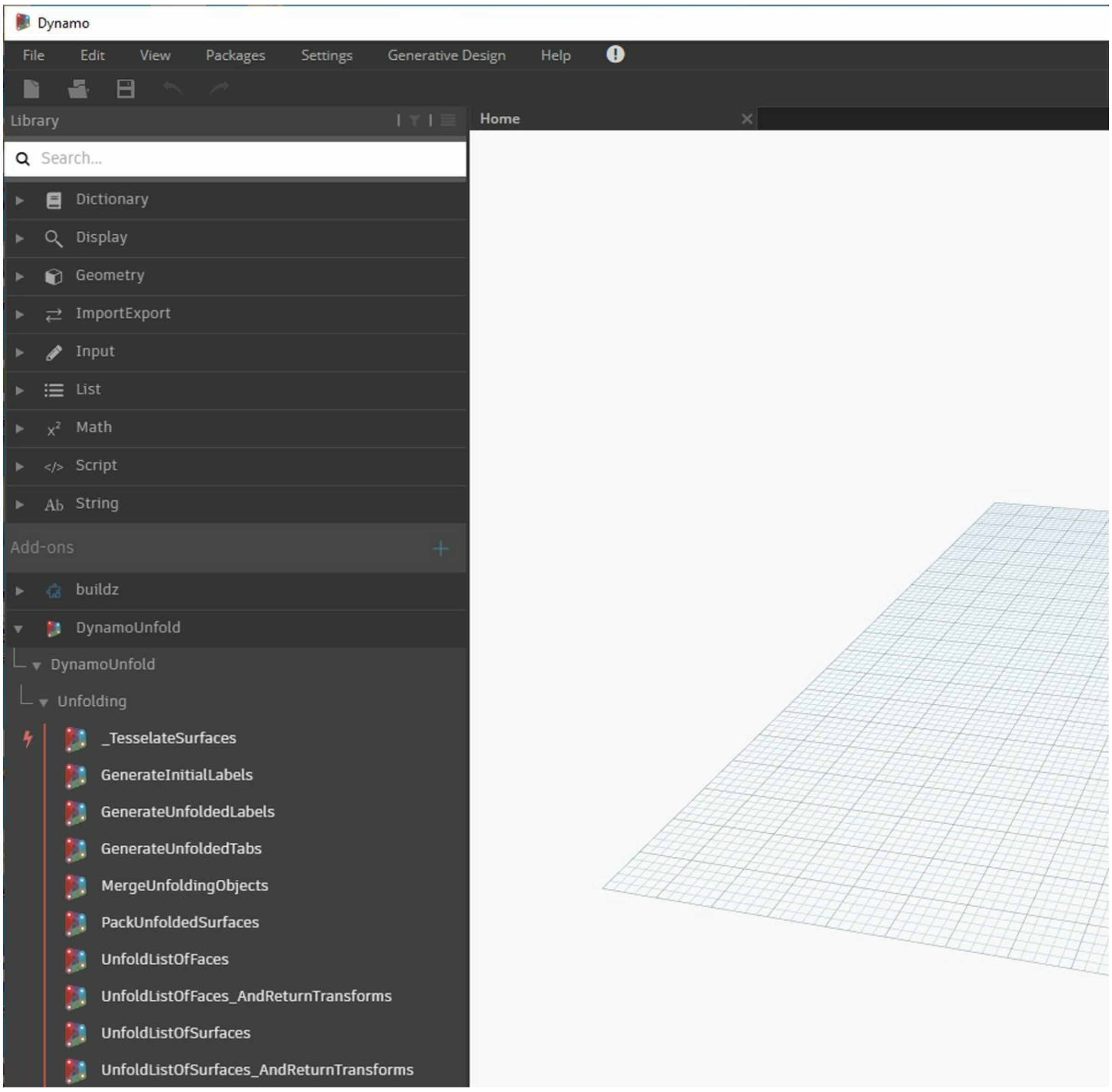
The example above focuses on a package with one custom node, but you use the same process for downloading packages with several custom nodes and supporting data files. Let's demonstrate that now with a more comprehensive package: Dynamo Unfold.



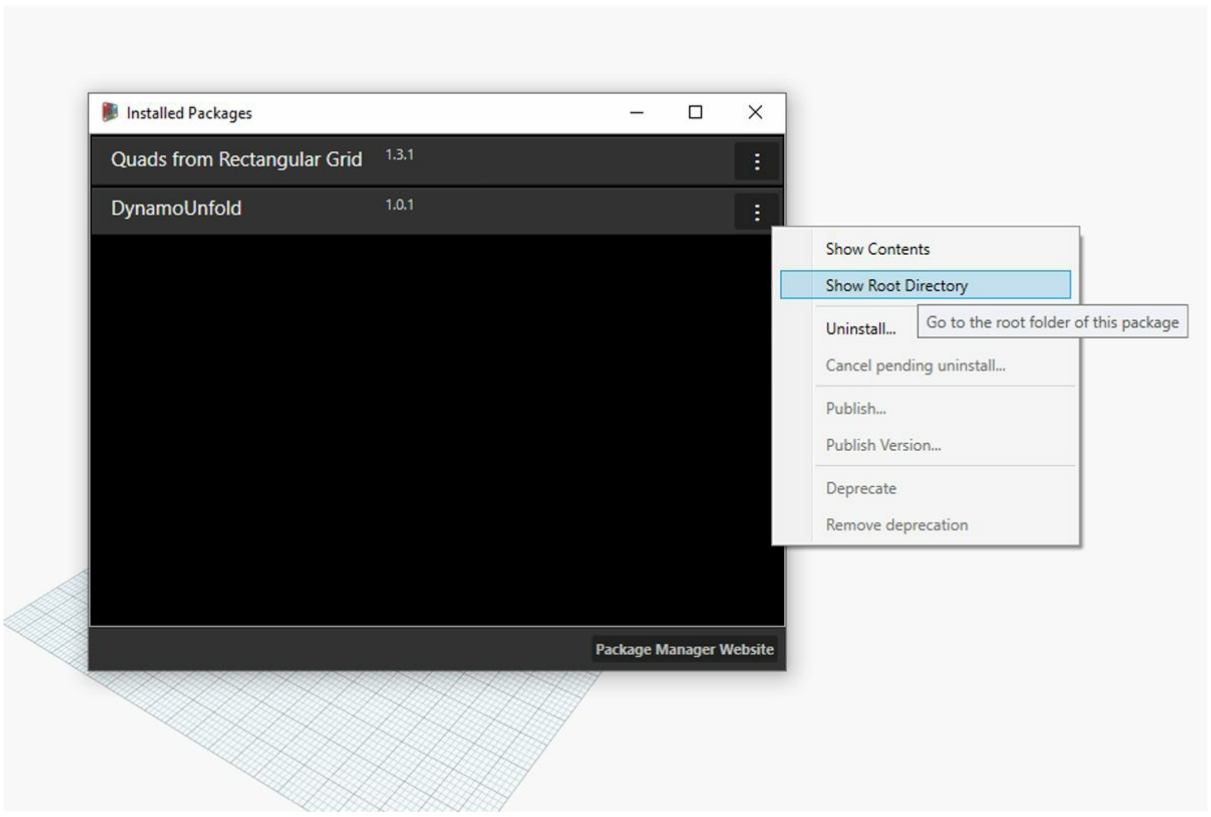
DynamoUnfold : Library for building topology from sets of surfaces and...

Quads from Rectangular Grid Installed 1.3.1

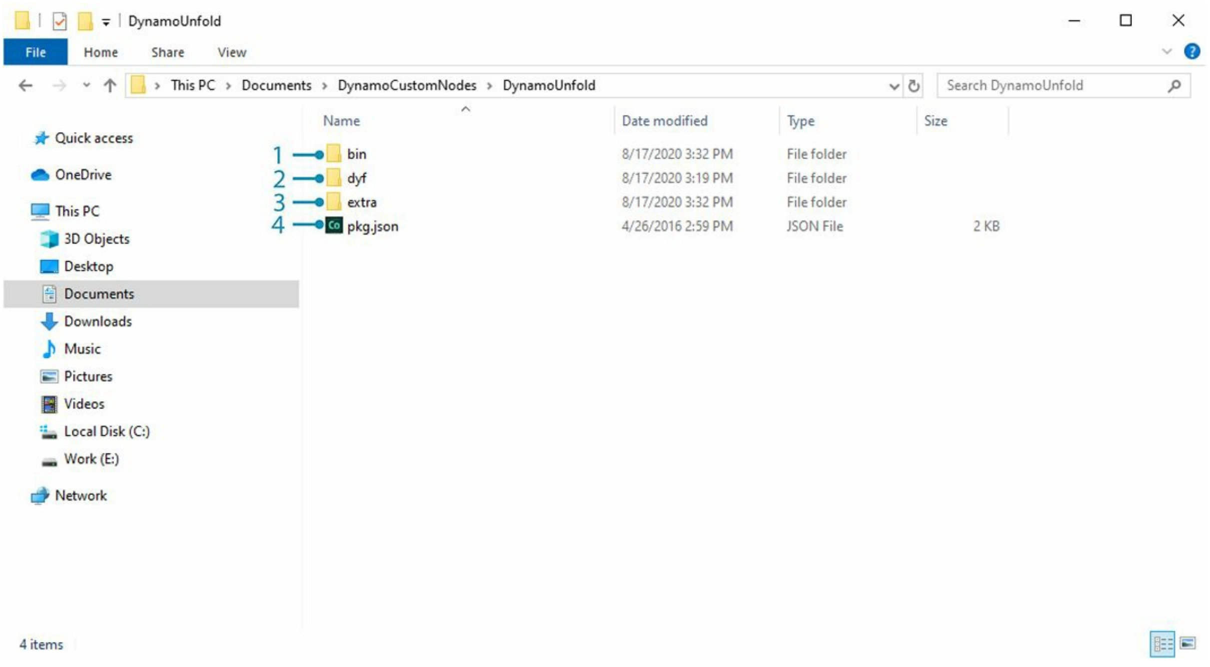
As in the example above, begin by selecting *Packages>Search for a Package...*. This time, we'll search for *"DynamoUnfold"*, one word, minding the caps. When we see the packages, download by clicking the arrow to the left of the package name. Dynamo Unfold will now be installed in your Dynamo Library.



In the Dynamo Library, we have a *DynamoUnfold* Group with multiple categories and custom nodes.



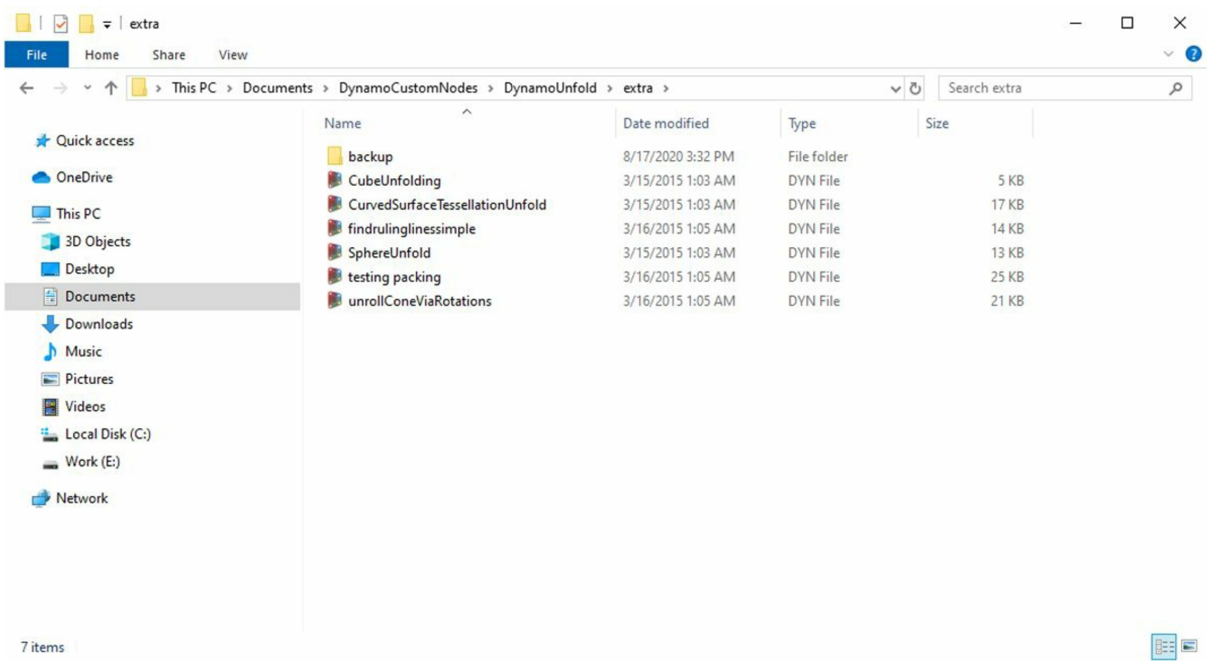
Now, let's take a look at the package's file structure. Select *"Packages>Manage Packages..."* in Dynamo. We'll see the window above with the two libraries we've installed. Click the button on the right of *DynamoUnfold* and select *"Show Root Directory"*.



Name	Date modified	Type	Size
1 bin	8/17/2020 3:32 PM	File folder	
2 dyf	8/17/2020 3:19 PM	File folder	
3 extra	8/17/2020 3:32 PM	File folder	
4 pkg.json	4/26/2016 2:59 PM	JSON File	2 KB

This will take us to the package's root directory. Notice that we have 3 folders and a file.

1. The *bin* folder houses .dll files. This Dynamo package was developed using Zero-Touch, so the custom nodes are held in this folder.
2. The *dyf* folder houses the custom nodes. This package was not developed using Dynamo custom nodes, so this folder is empty for this package.
3. The extra folder houses all additional files, including our example files.
4. The pkg file is a basic text file defining the package settings. We can ignore this for now.


extra

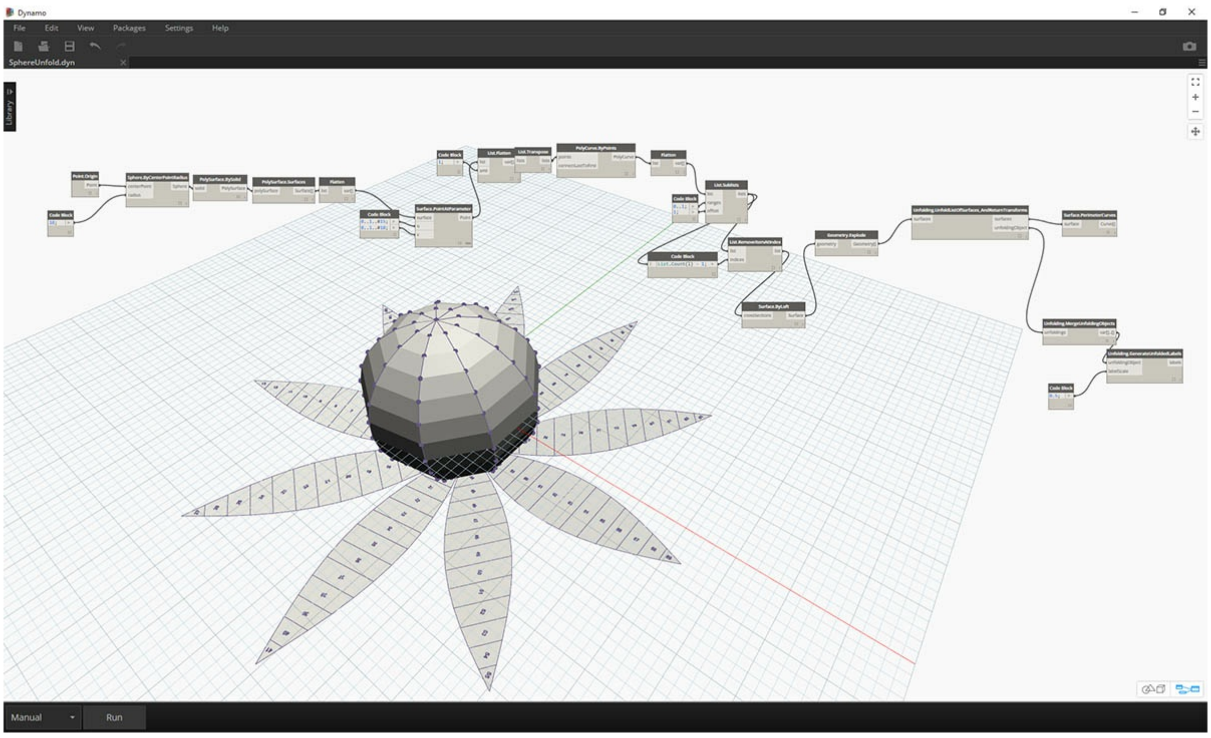
File Home Share View

This PC › Documents › DynamoCustomNodes › DynamoUnfold › extra ›

Search extra

Name	Date modified	Type	Size
backup	8/17/2020 3:32 PM	File folder	
CubeUnfolding	3/15/2015 1:03 AM	DYN File	5 KB
CurvedSurfaceTessellationUnfold	3/15/2015 1:03 AM	DYN File	17 KB
findrulinglinessimple	3/16/2015 1:05 AM	DYN File	14 KB
SphereUnfold	3/15/2015 1:03 AM	DYN File	13 KB
testing packing	3/16/2015 1:05 AM	DYN File	25 KB
unrollConeViaRotations	3/16/2015 1:05 AM	DYN File	21 KB

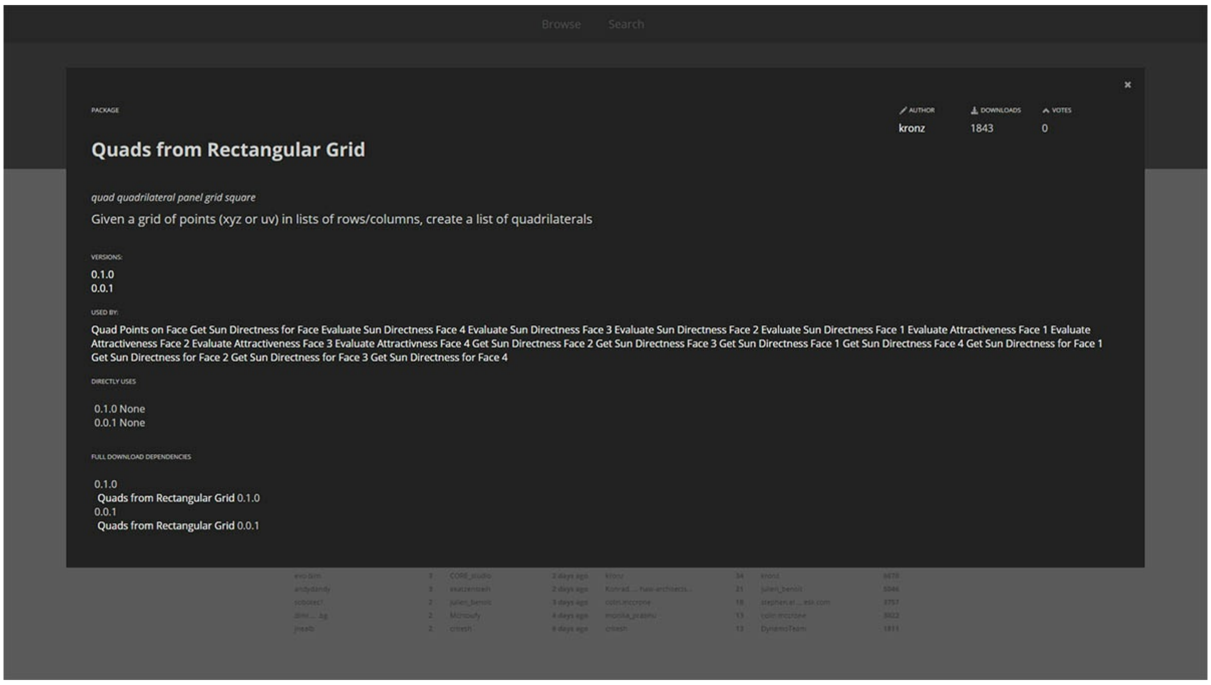
Opening the *"extra"* folder, we see a bunch of example files that were downloaded with the install. Not all packages have example files, but this is where you can find them if they are part of a package. Let's open up *"SphereUnfold"*.



After opening the file and hitting *"Run"* on the solver, we have an unfolded sphere! Example files like these are helpful for learning how to work with a new Dynamo package.

Dynamo Package Manager

Another way to discover Dynamo packages is to explore the [Dynamo Package Manager]() online. This is a good way to browse for packages, since the repository sorts packages in order of download count and popularity. Also, it's an easy way to gather information on recent updates for packages, as some Dynamo packages are subjected to versioning and dependencies of Dynamo builds.



PACKAGE

Quads from Rectangular Grid

AUTHOR: kronz
DOWNLOADS: 1843
VOTES: 0

quad quadrilateral panel grid square

Given a grid of points (xyz or uv) in lists of rows/columns, create a list of quadrilaterals

VERSIONS:

0.1.0
0.0.1

USED BY:

Quad Points on Face Get Sun Directness for Face Evaluate Sun Directness Face 4 Evaluate Sun Directness Face 3 Evaluate Sun Directness Face 2 Evaluate Sun Directness Face 1 Evaluate Attractiveness Face 1 Evaluate Attractiveness Face 2 Evaluate Attractiveness Face 3 Evaluate Attractivness Face 4 Get Sun Directness Face 2 Get Sun Directness Face 3 Get Sun Directness Face 1 Get Sun Directness Face 4 Get Sun Directness for Face 1 Get Sun Directness for Face 2 Get Sun Directness for Face 3 Get Sun Directness for Face 4

DIRECTLY USES

0.1.0 None
0.0.1 None

FULL DOWNLOAD DEPENDENCIES

0.1.0
Quads from Rectangular Grid 0.1.0
0.0.1
Quads from Rectangular Grid 0.0.1

By clicking on *"Quads from Rectangular Grid"* in the Dynamo Package Manager, you can see its descriptions, versions, the developer, and possible dependencies.

You can also download the package files from the Dynamo Package Manager, but doing so directly from Dynamo is a more seamless process.

Where are Files Stored Locally?

If you do download files from the Dynamo package manager, or if you would like to see where all of your package files are kept, click on *Settings>Manage Node and Package Paths...*. By clicking on the ellipsis next to the folder directory, you can copy the root folder and delve into the package in your explorer window. By default, packages are installed in a location similar to this folder path: *C:/Users/[username]/AppData/Roaming/Dynamo/[Dynamo Version]*.

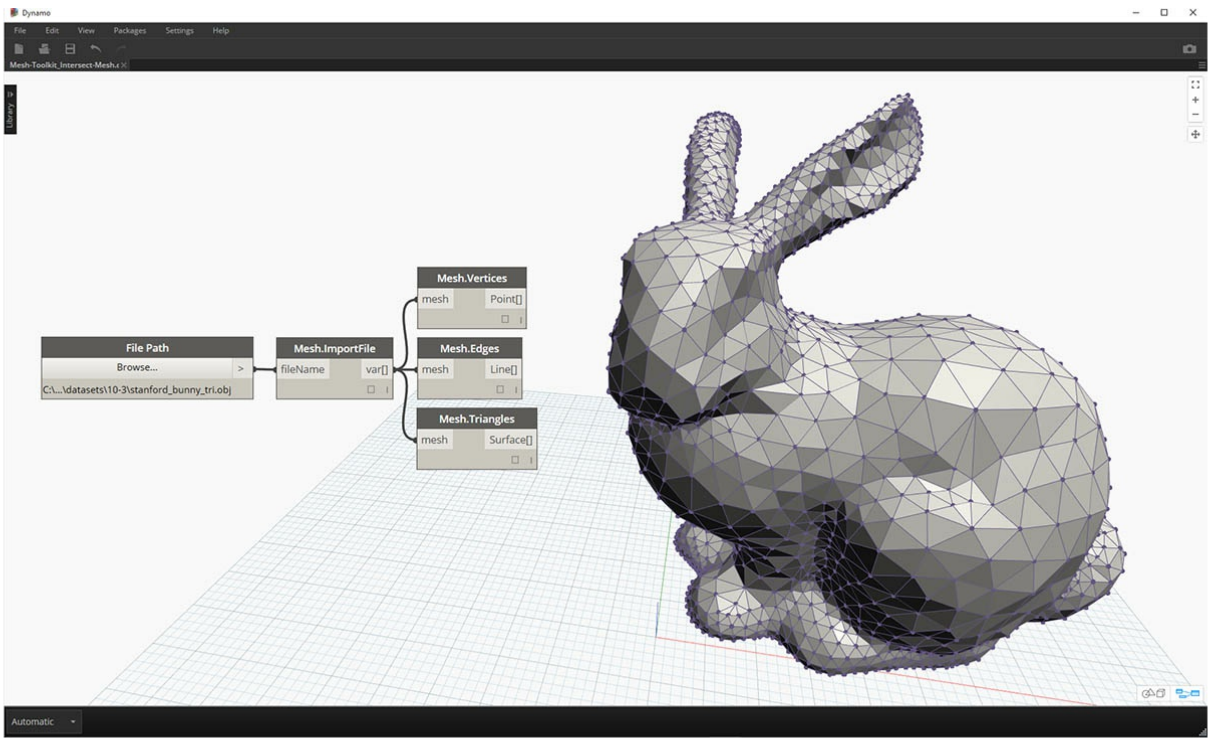
Going Further with Packages

The Dynamo community is constantly growing and evolving. By exploring the Dynamo Package Manager from time to time, you'll find some exciting new developments. In the following sections, we'll take a more in-depth look at packages, from the end-user perspective to authorship of your own Dynamo Package.

Package Case Study - Mesh Toolkit

Package Case Study – Mesh Toolkit

The Dynamo Mesh Toolkit provides tools to import meshes from external file formats, create a mesh from Dynamo geometry objects, and manually build meshes by their vertices and indices. The library also provides tools to modify meshes, repair meshes, or extract horizontal slices for use in fabrication.

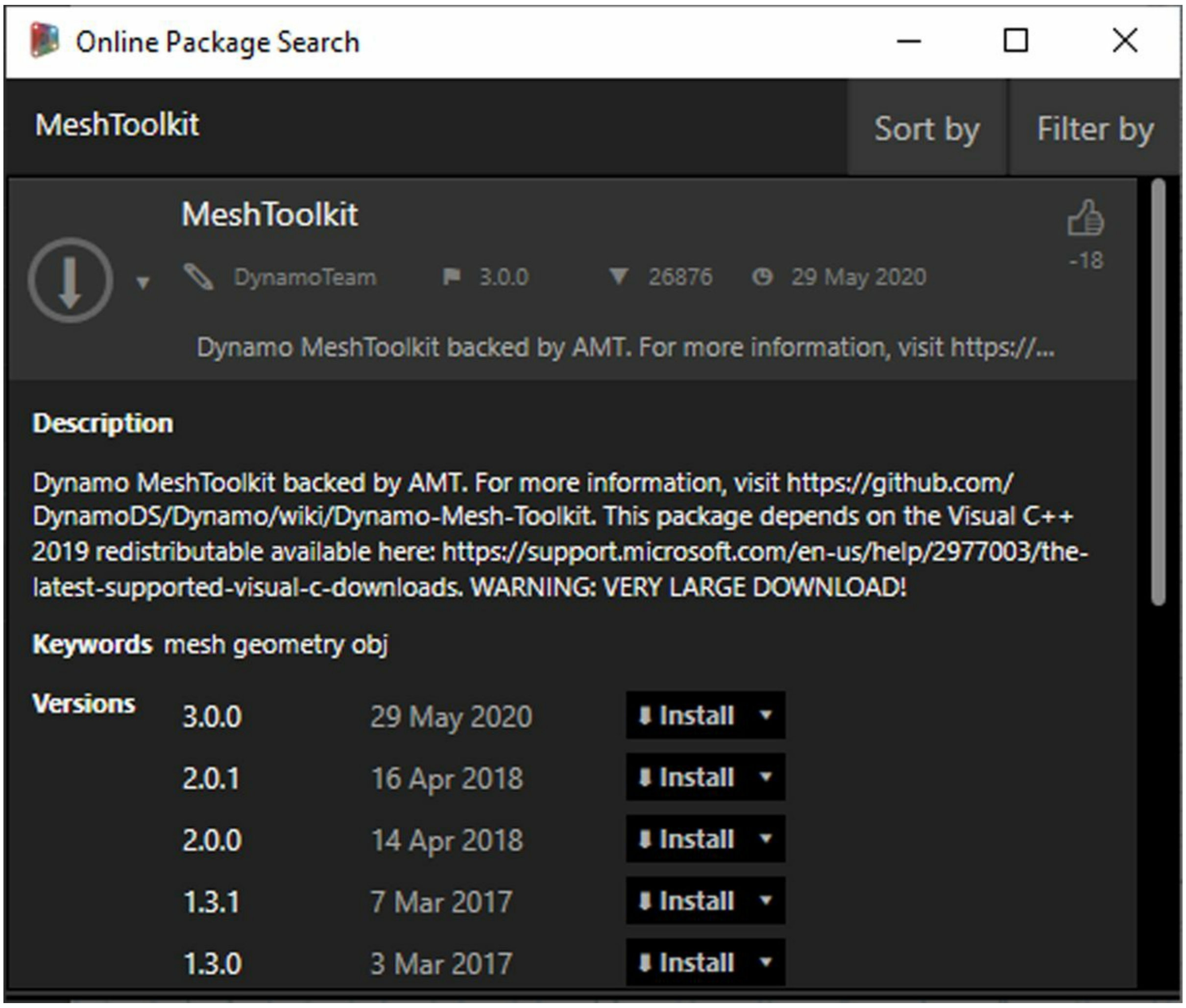


The Dynamo Mesh Toolkit is part of Autodesk's ongoing mesh research, and as such will continue to grow over the coming years. Expect new methods to appear on the toolkit frequently, and feel free to reach out to the Dynamo team with comments, bugs, and suggestions for new features.

Meshes vs. Solids

The exercise below demonstrates some basic mesh operations using the Mesh Toolkit. In the exercise, we intersect a mesh with a series of planes, which can be computationally expensive using solids. Unlike a solid, a mesh has a set "resolution" and is not defined mathematically, but topologically, and we can define this resolution based on the task at hand. For more details on mesh to solid relationships, you can reference the Geometry For Computation Design chapter in this primer. For a more thorough examination of Mesh Toolkit, you can reference the Dynamo Wiki page. Let's jump into the package in the exercise below.

Install Mesh Toolkit



Description

Dynamo MeshToolkit backed by AMT. For more information, visit https://github.com/DynamoDS/Dynamo/wiki/Dynamo-Mesh-Toolkit. This package depends on the Visual C++ 2019 redistributable available here: https://support.microsoft.com/en-us/help/2977003/the-latest-supported-visual-c-downloads. WARNING: VERY LARGE DOWNLOAD!

Keywords mesh geometry obj

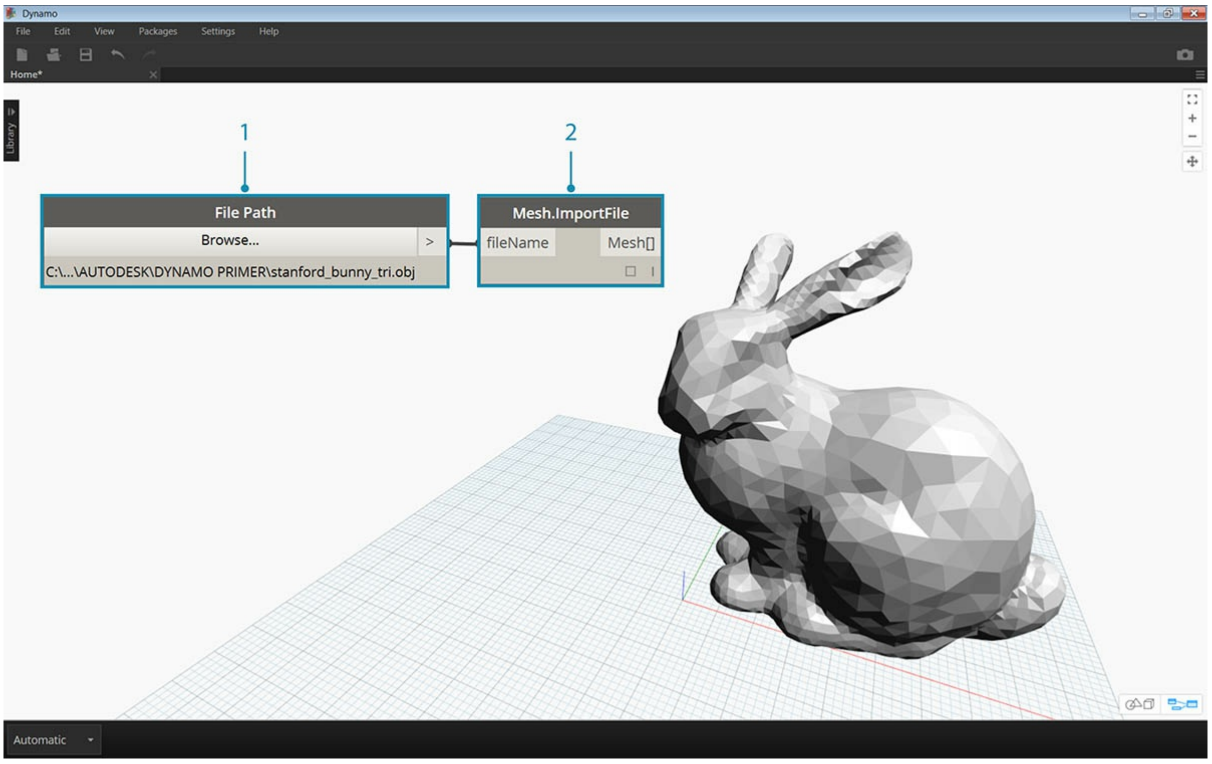
Versions			
	3.0.0	29 May 2020	Install
	2.0.1	16 Apr 2018	Install
	2.0.0	14 Apr 2018	Install
	1.3.1	7 Mar 2017	Install
	1.3.0	3 Mar 2017	Install

In Dynamo, go to *Packages* > *Search for Packages...* in the top menu bar. In the search field, type *"MeshToolkit"*, all one word, minding the caps. Click the download arrow for the appropriate package for your version of Dynamo. Simple as that!

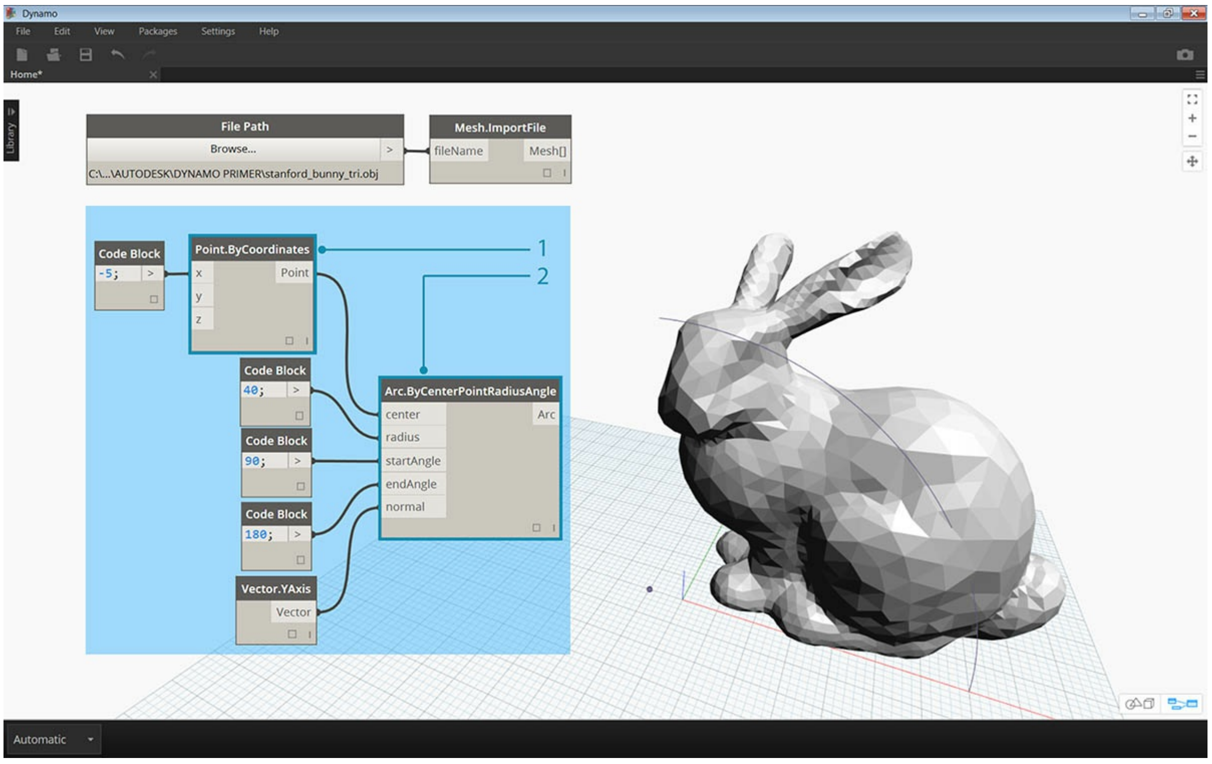
Exercise

Download and unzip the example files for this exercise (Right click and "Save Link As..."). A full list of example files can be found in the Appendix. MeshToolkit.zip

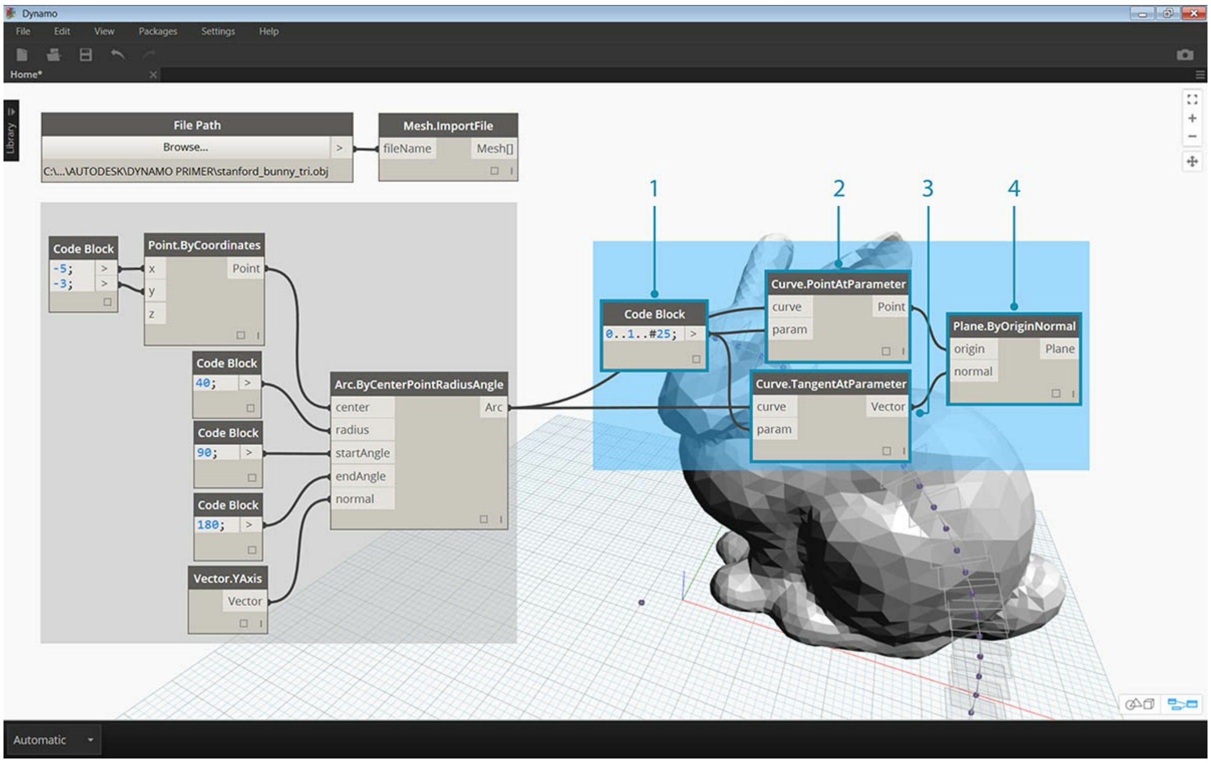
Begin by opening *Mesh-Toolkit_Intersect-Mesh.dyn in Dynamo.* In this example, we will look at the Intersect node in the mesh toolkit. We will import a mesh and intersect it with a series of input planes to create slices. This is the starting point for preparing the model for fabrication on a laser cutter, waterjet cutter, or CNC mill.



1. **File Path:** Locate the mesh file to import (*stanford_bunny_tri.obj*). Supported file types are .mix and .obj
2. **Mesh.ImportFile:** Connect the file path to import the mesh

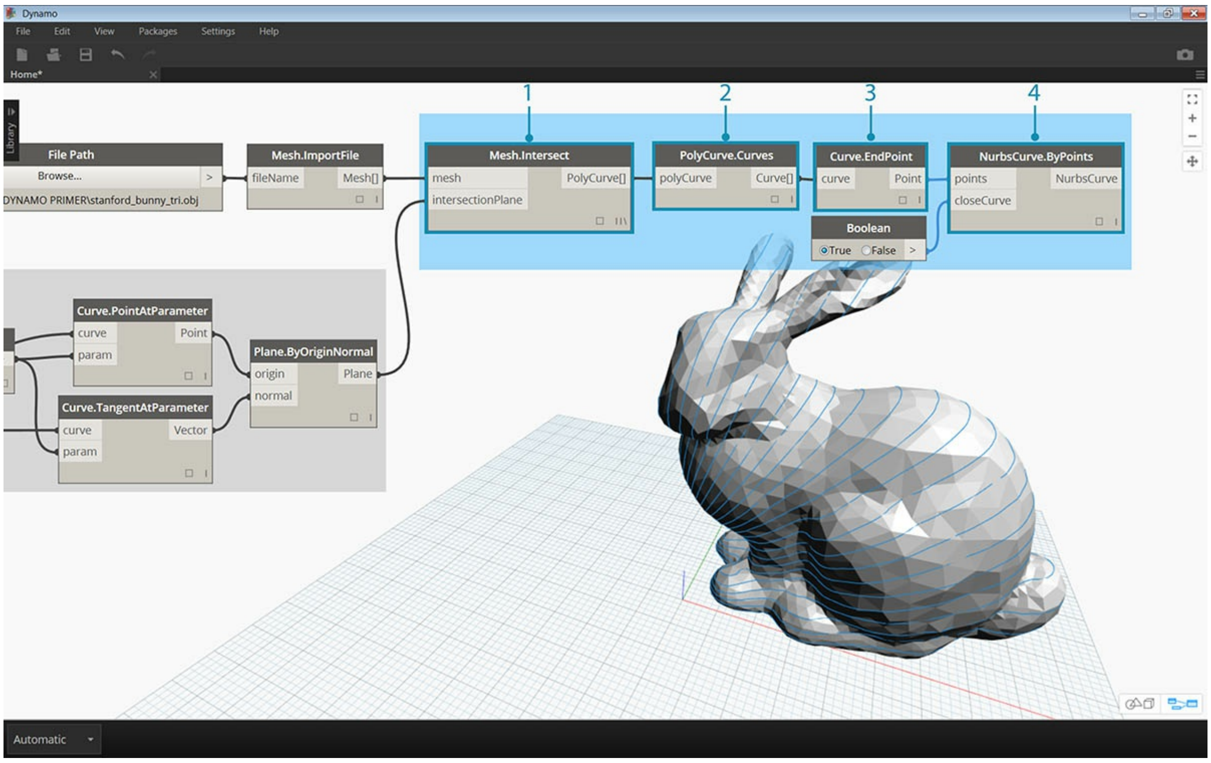


1. **Point.ByCoordinates:** Construct a point – this will be the center of an arc.
2. **Arc.ByCenterPointRadiusAngle:** Construct an arc around the point. This curve will be used to position a series of planes.

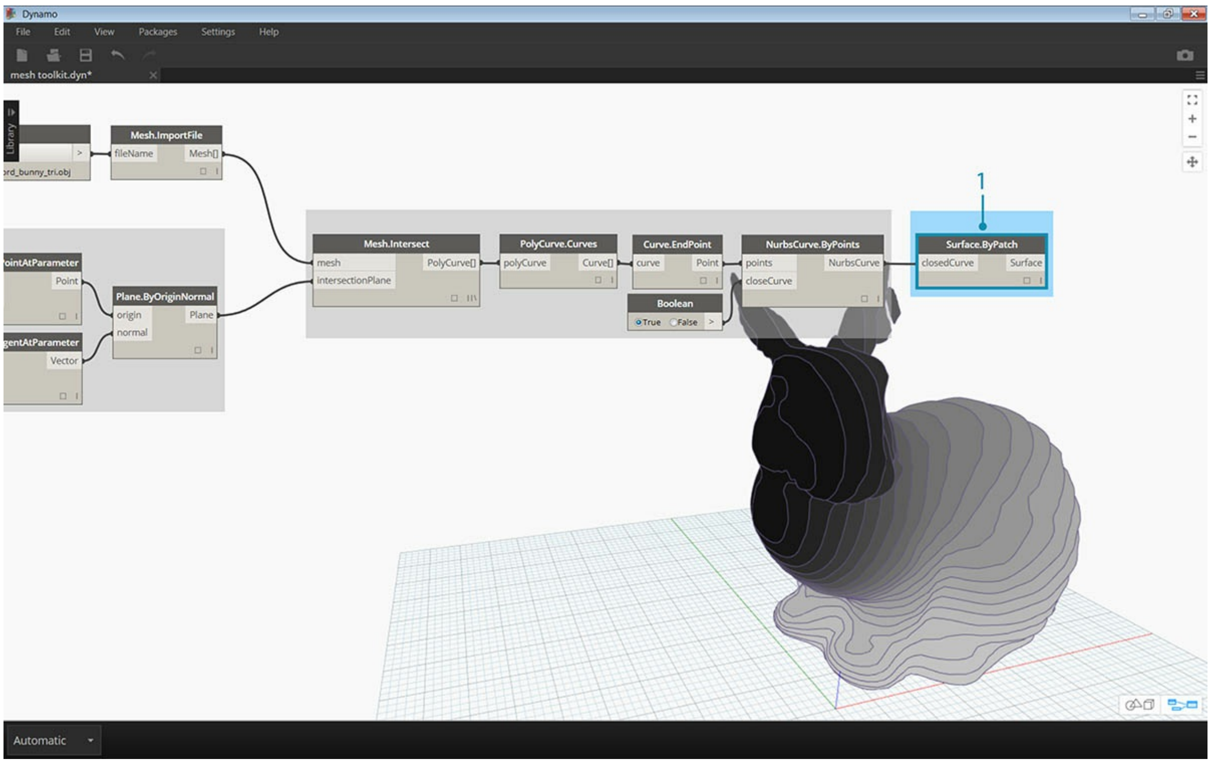


1. Code Block: Create a range of numbers between zero and one.
2. **Curve.PointAtParameter:** Connect the arc to the *'curve'* input and the code block output to the *'param'* input to extract a series of points along the curve.
3. **Curve.TangentAtParameter:** Connect the same inputs as the previous node.
4. **Plane.ByOriginNormal:** Connect the points to the *'origin'* input and the vectors to the *'normal'* input to create a series of planes at each point.

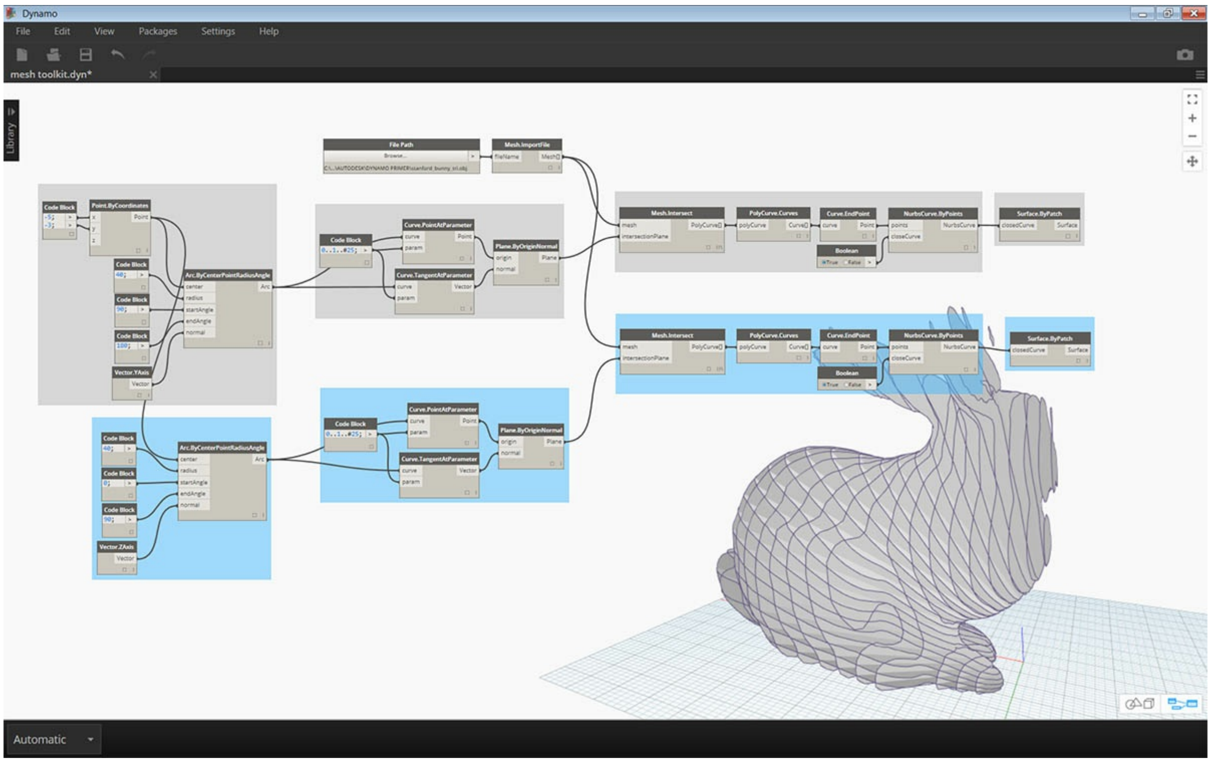
You should now see a series of planes oriented along the arc. Next, we will use these planes to intersect the mesh.



1. **Mesh.Intersect:** Intersect the planes with the imported mesh, creating a series of polycurve contours.
2. **PolyCurve.Curves:** Break the polycurves into their curve fragments.
3. **Curve.EndPoint:** Extract the end points of each curve.
4. **NurbsCurve.ByPoints:** Use the points to construct a nurbs curve. Use a Boolean node set to *True* to close the curves.



1. **Surface.ByPatch:** Construct surface patches for each contour to create “slices” of the mesh.



Add a second set of slices for a waffle/egg-crate effect.

You may have noticed that the intersection operations calculate faster with a mesh vs. a comparable solid. Workflows such as the one demonstrated in this exercise lend themselves well to working with meshes.

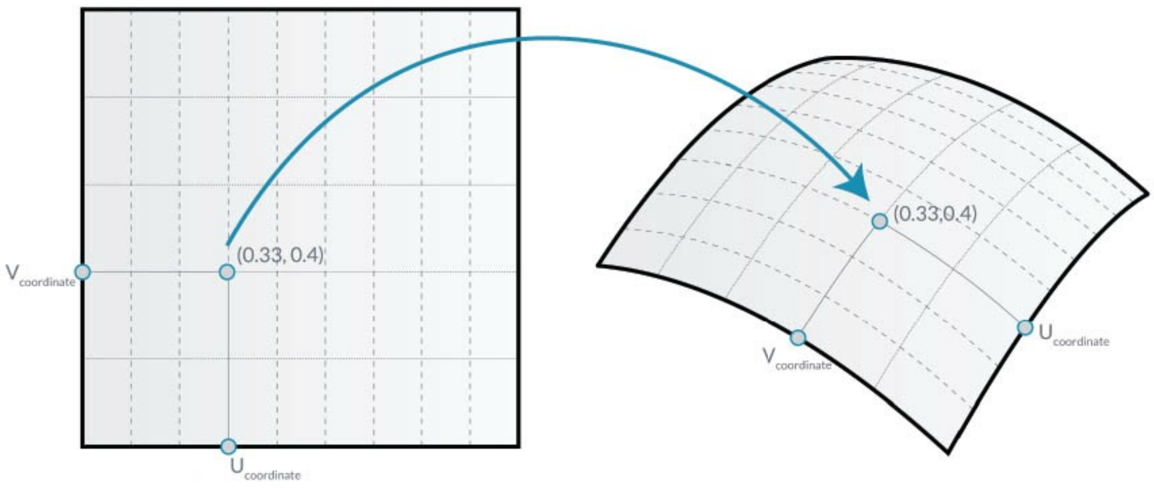
Developing a Package

Developing a Package

Dynamo offers a variety of ways to create a package for your personal use or for sharing with the Dynamo community. In the case study below, we'll walk through how a package is set up by deconstructing an existing one. This case study builds on lessons from the previous chapter, providing a set of custom nodes for mapping geometry, by UV coordinates, from one Dynamo surface to another.

MapToSurface

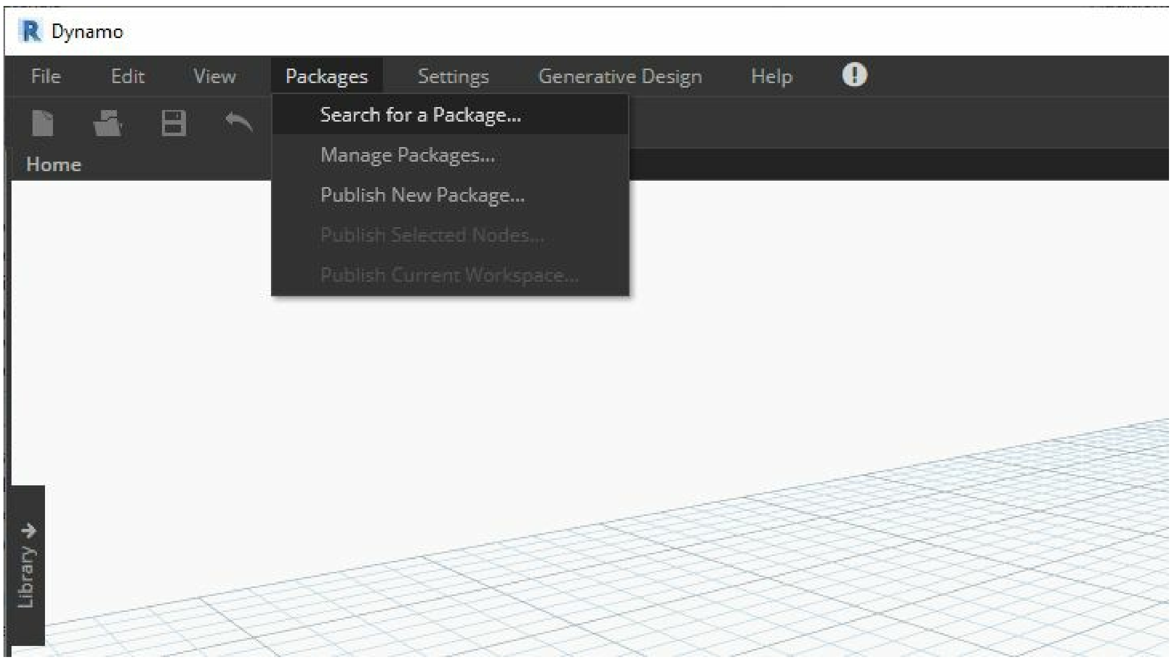
We're going to work with a sample package which demonstrates the UV mapping of points from one surface to another. We've already built the fundamentals of the tool in the Creating a Custom Node section of this primer. The files below demonstrate how we can take the concept of UV Mapping and develop a set of tools for a publishable library.



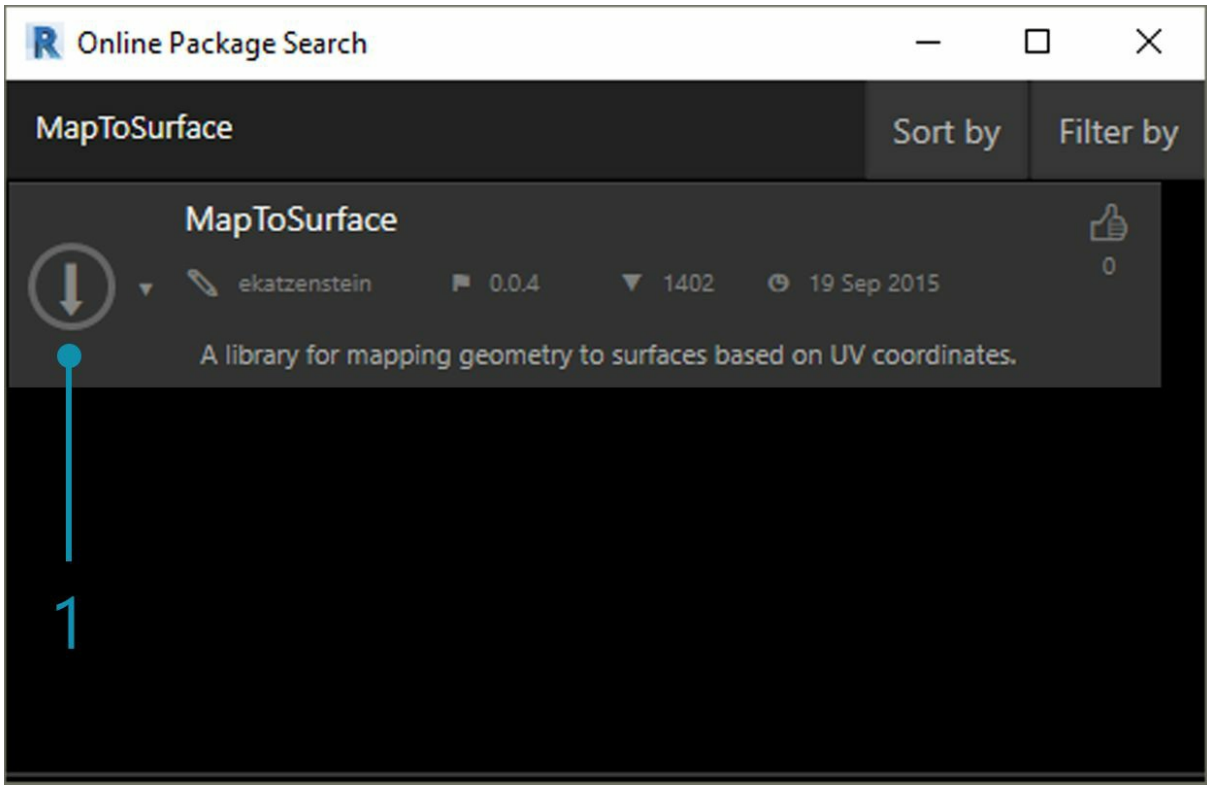
In this image, we map a point from one surface to another using UV coordinates. The package is based on this concept, but with more complex geometry.

Installing the Package

In the previous chapter, we explored ways for panelizing a surface in Dynamo based on curves defined in the XY plane. This case study extends these concepts for more dimensions of geometry. We're going to install this package as built in order to demonstrate how it was developed. In the next section, we'll demonstrate how this package was published.

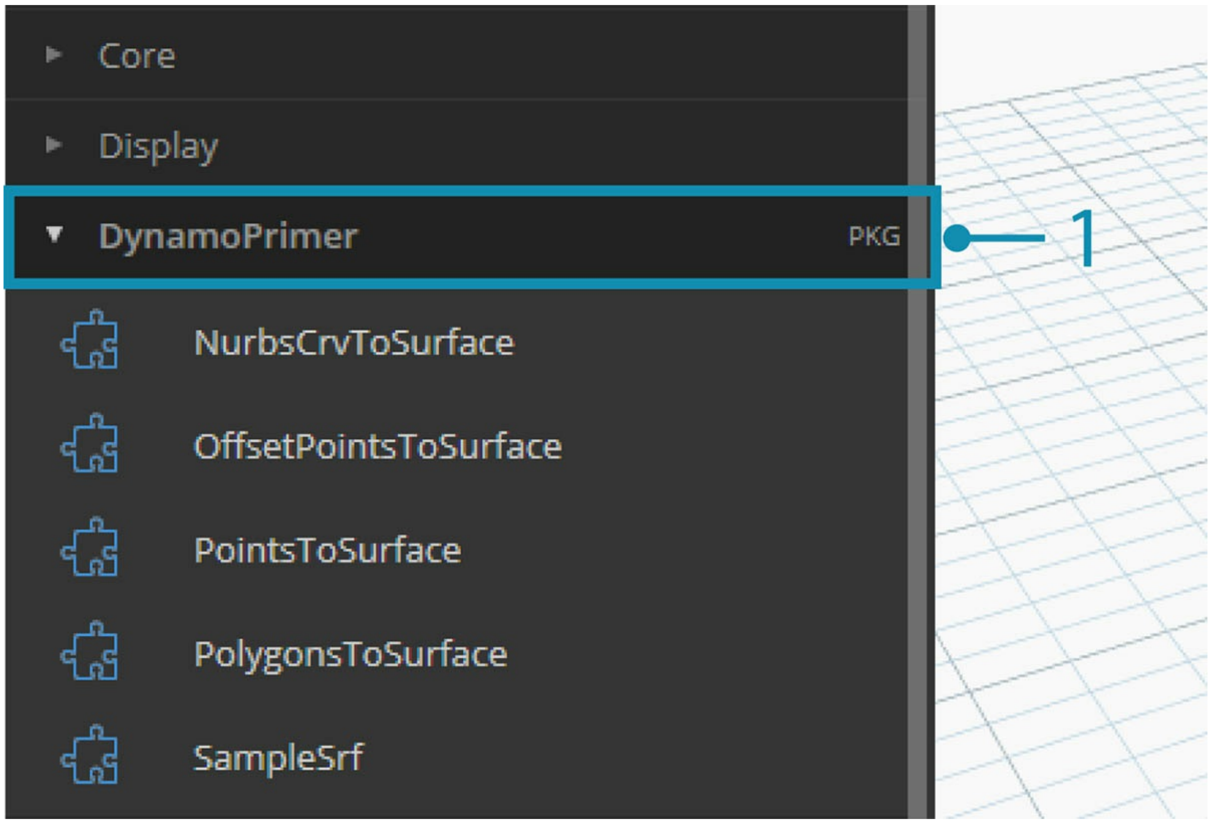


This is the easy part. In Dynamo, navigate to *"Packages>Search for a Package..."*



Search for the package *"MapToSurface"* (all one word).

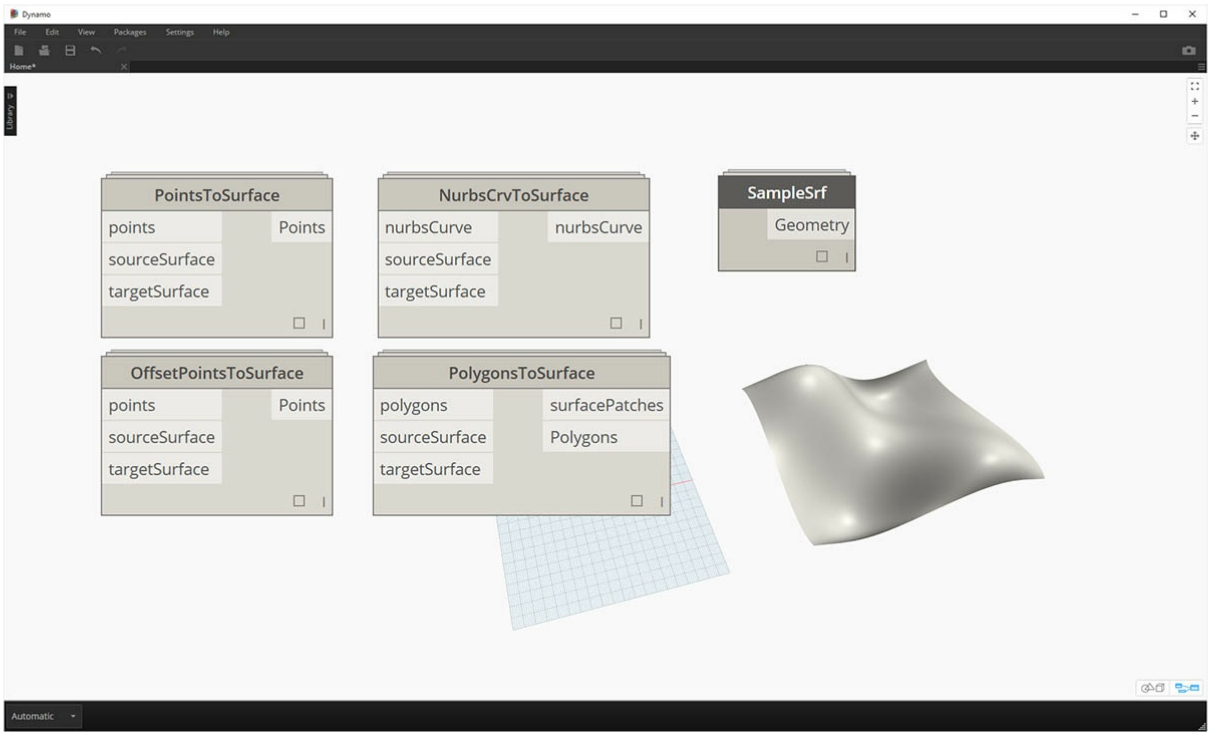
1. When the package is found, click on the big download arrow to the left of the package name. This will install the package into Dynamo.



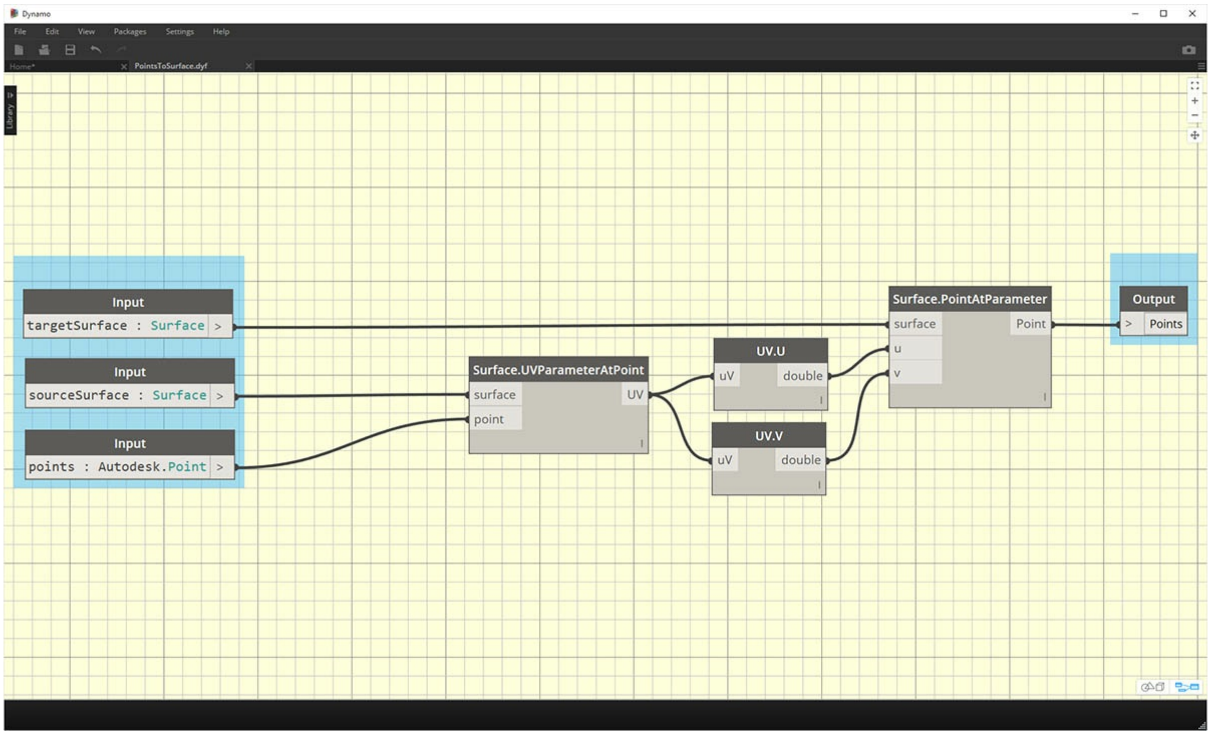
1. After installing, the custom nodes should be available under the "DynamoPrimer" group or your Dynamo Library. With the package now installed, let's walk through how it's set up.

Custom Nodes

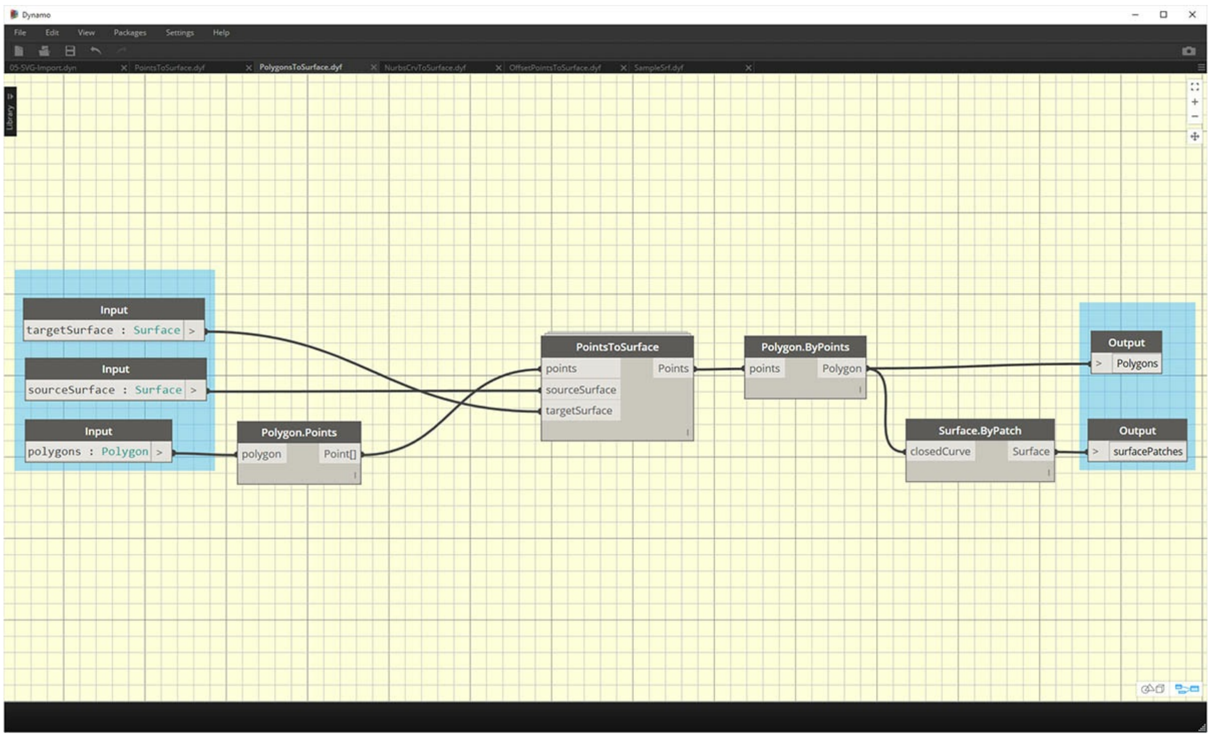
The package we're creating uses five custom nodes that we've built for reference. Let's walk through what each node does below. Some custom nodes build off of other custom nodes, and the graphs have a layout for other users to understand in a straightforward manner.



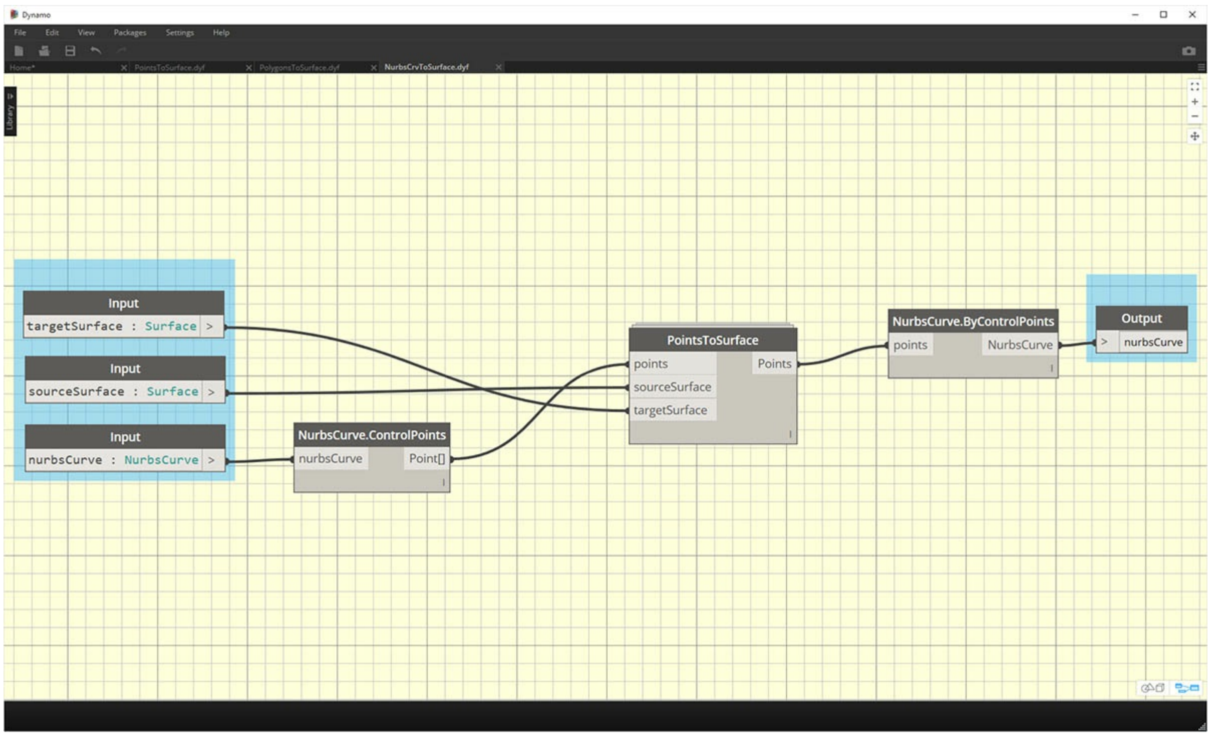
This is a simple package with five custom nodes. In the steps below, we'll briefly talk about each custom node's setup.



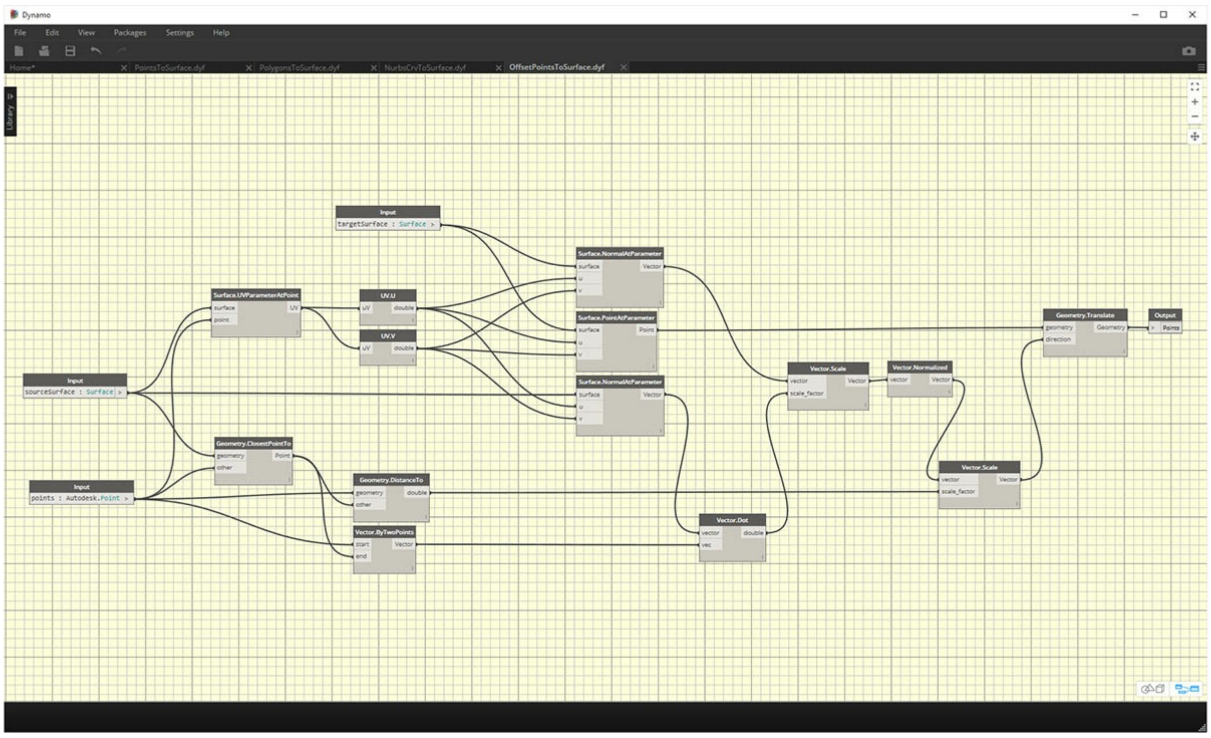
PointsToSurface: This is a basic custom node, and one from which all of the other mapping nodes are based. Simply put, the node maps a point from a source surface UV coordinate to the location of the target surface UV coordinate. And since points are the most primitive geometry, from which more complex geometry is built, we can use this logic to map 2D, and even 3D geometry from one surface to another.



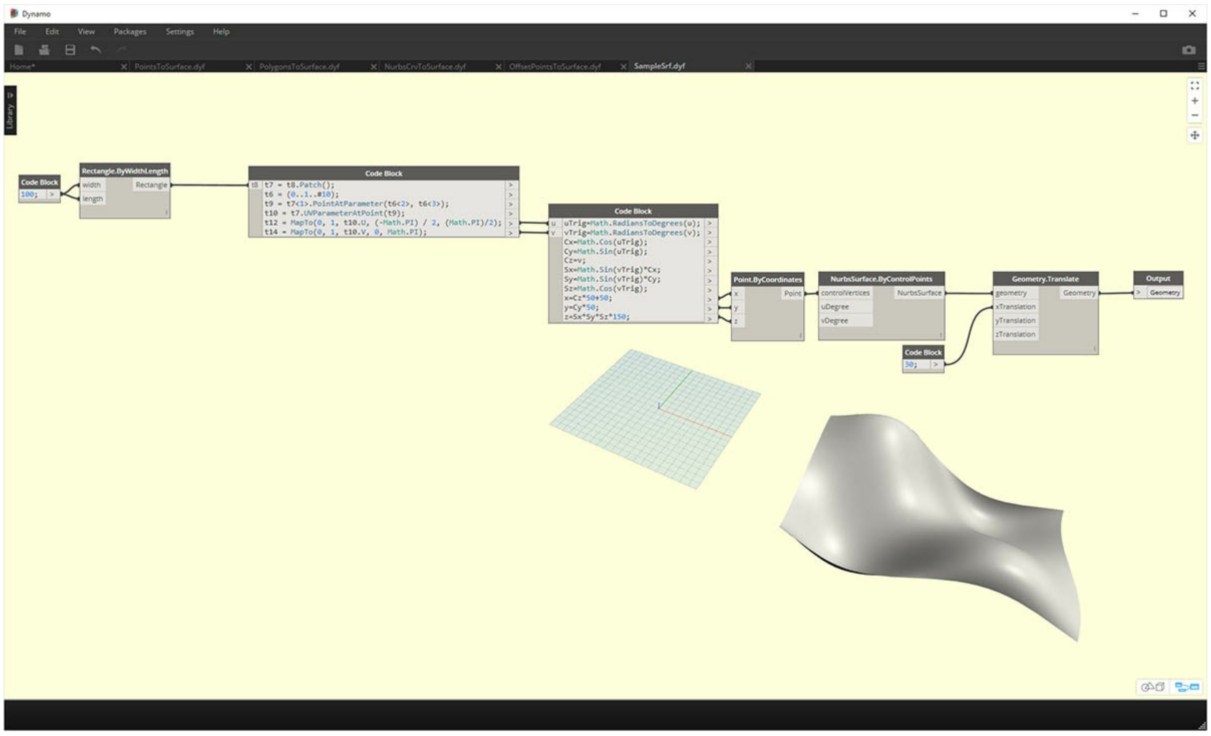
PolygonsToSurface: the logic of extending mapped points from 1D geometry to 2D geometry is demonstrated simply with polygons here. Notice that we have nested the *"PointsToSurface"* node into this custom node. This way we can map the points of each polygon to the surface, and then regenerate the polygon from those mapped points. By maintaining the proper data structure (a list of lists of points), we're able to keep the polygons separate after they're reduced to a set of points.



NurbsCrvtoSurface: The same logic applies here as in the *"PolygonsToSurface"* node. But instead of mapping polygonal points, we're mapping control points of a nurbs curve.



OffsetPointsToSurface: This node gets a little more complex, but the concept is simple: like the *"PointsToSurface"* node, this node maps points from one surface to another. However, it also considers points which are not on the original source surface, gets their distance to the closest UV parameter, and maps this distance to the target surface normal at the corresponding UV coordinate. This will make more sense when looking at the example files.



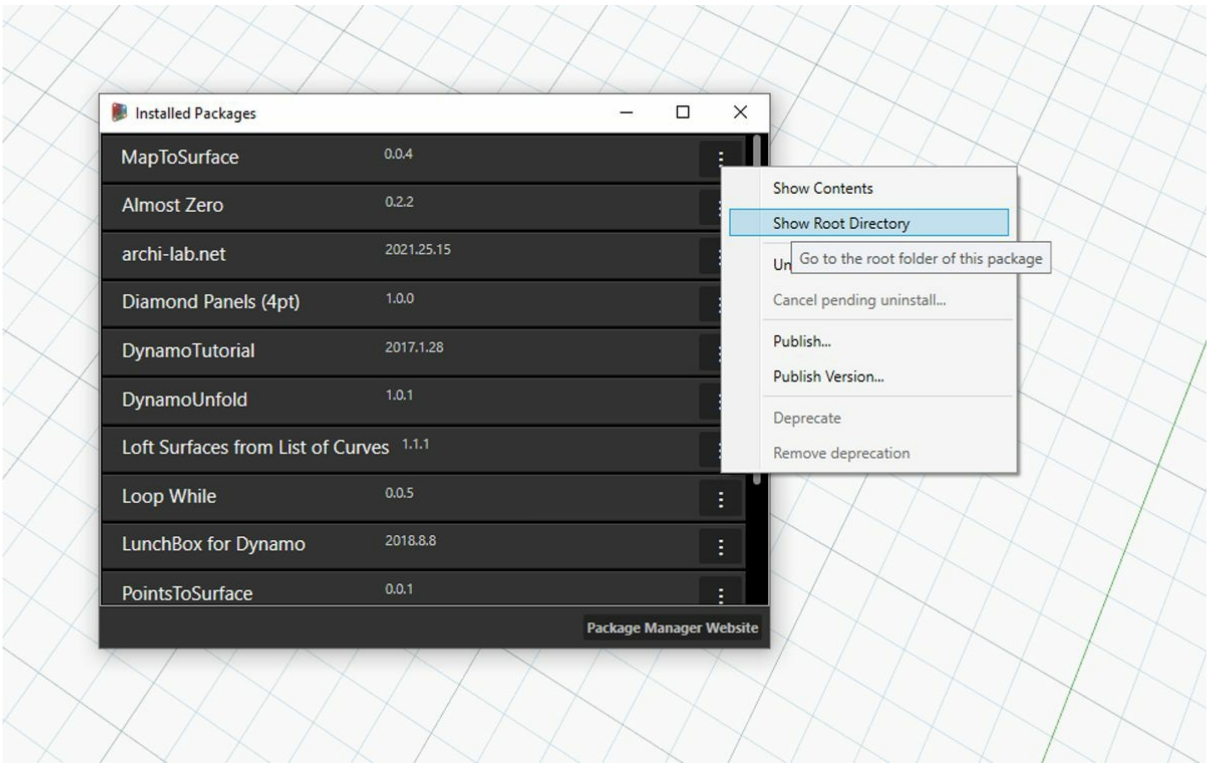
```
t7 = t8.Patch();
t6 = (0..1..#10);
t9 = t7<1>.PointAtParameter(t6<2>, t6<3>);
t10 = t7.UVParameterAtPoint(t9);
t12 = MapTo(0, 1, t10.U, (-Math.PI) / 2, (Math.PI)/2);
t14 = MapTo(0, 1, t10.V, 0, Math.PI);
```

```
uTrig=Math.RadiansToDegrees(u);
vTrig=Math.RadiansToDegrees(v);
Cx=Math.Cos(uTrig);
Cy=Math.Sin(uTrig);
Cz=v;
Sx=Math.Sin(vTrig)*Cx;
Sy=Math.Sin(vTrig)*Cy;
Sz=Math.Cos(vTrig);
x=Cz*50+50;
y=Cy*50;
z=Sx*Sy*Sz*150;
```

SampleSrf: This is a simple node which creates a parametric surface to map from the source grid to an undulating surface in the example files.

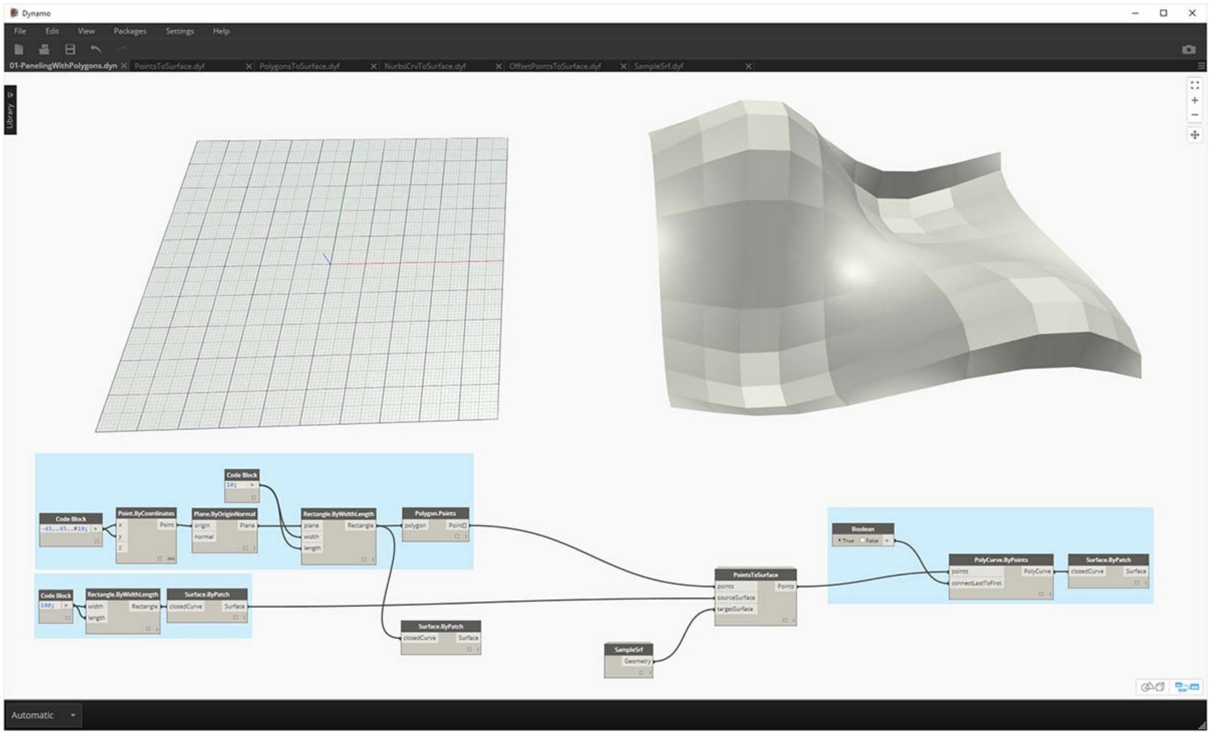
Example Files

The example files can be found in the package's root folder (In Dynamo, navigate to this folder by going to *Packages>Manage Packages...*).

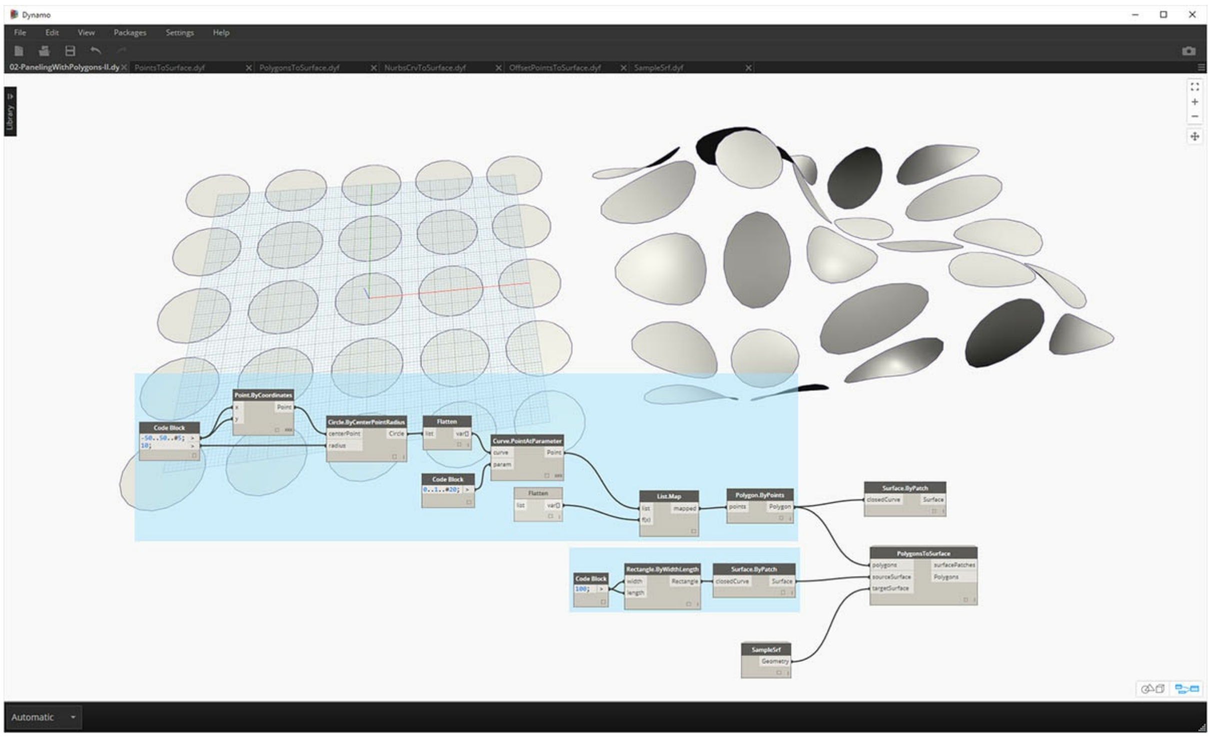


In the manage packages window, click on the three vertical dots to the right of *"MapToSurface"* and choose *"Show Root Directory"*.

With the root directory open, navigate to the *"extra"* folder, which houses all of the files in the package which are not custom nodes. This is where examples files (if they exist) are stored for Dynamo packages. The screenshots below discuss the concepts demonstrated in each example file.



01-PanelingWithPolygons: This example file demonstrates how *"PointsToSurface"* may be used to panelize a surface based on a grid of rectangles. This should look familiar, as we demonstrated a similar workflow in the previous chapter.

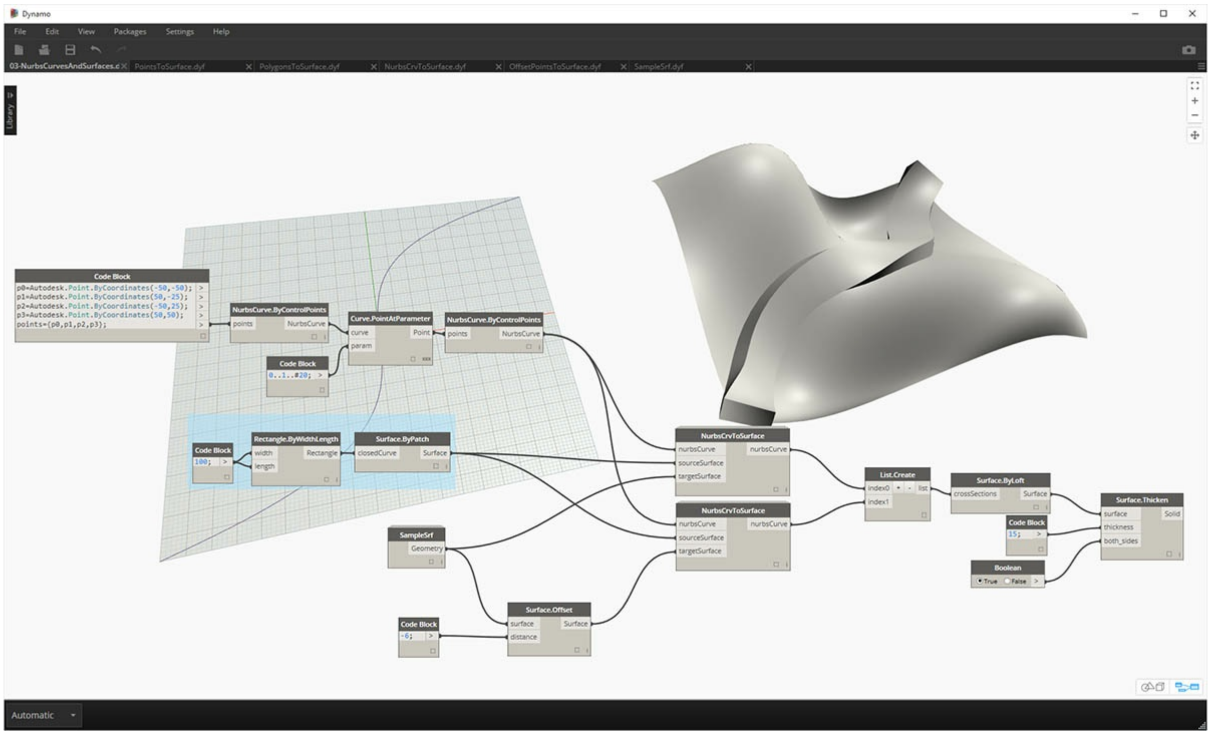


```
-50..50..#5;
10;
```

```
0..1..#20;
```

```
100;
```

02-PanelingWithPolygons-II: Using a similar workflow, this exercise file shows a setup for mapping circles (or polygons representing circles) from one surface to another. This uses the *"PolygonsToSurface"* node.



```
p0=Autodesk.Point.ByCoordinates(-50,-50);
p1=Autodesk.Point.ByCoordinates(50,-25);
p2=Autodesk.Point.ByCoordinates(-50,25);
p3=Autodesk.Point.ByCoordinates(50,50);
points=(p0,p1,p2,p3);
```

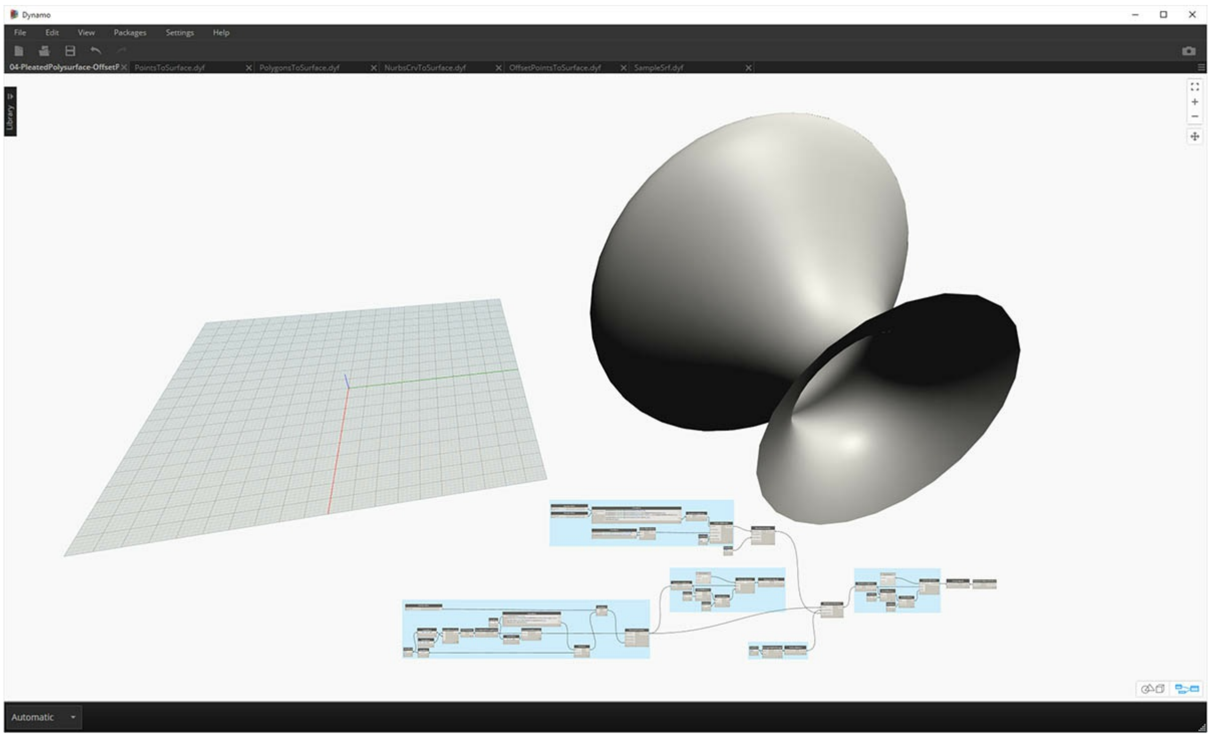
```
0..1..#20;
```

```
100;
```

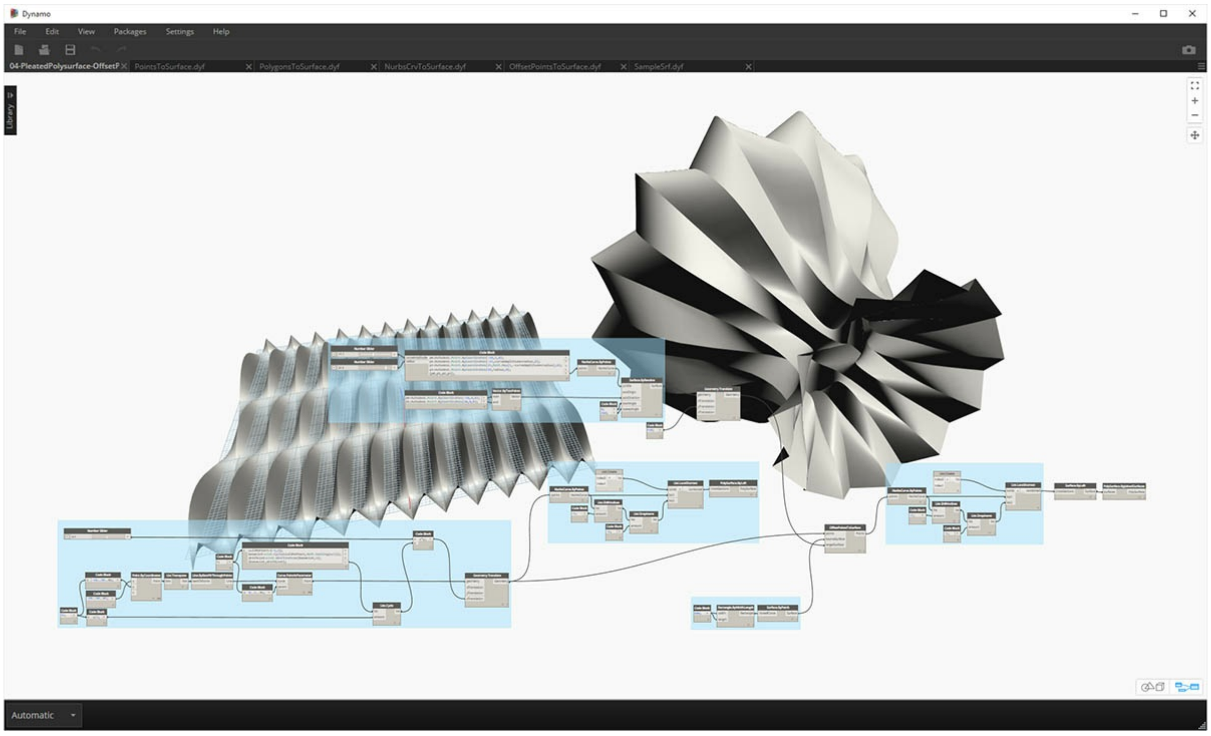
```
15;
```

```
-6;
```

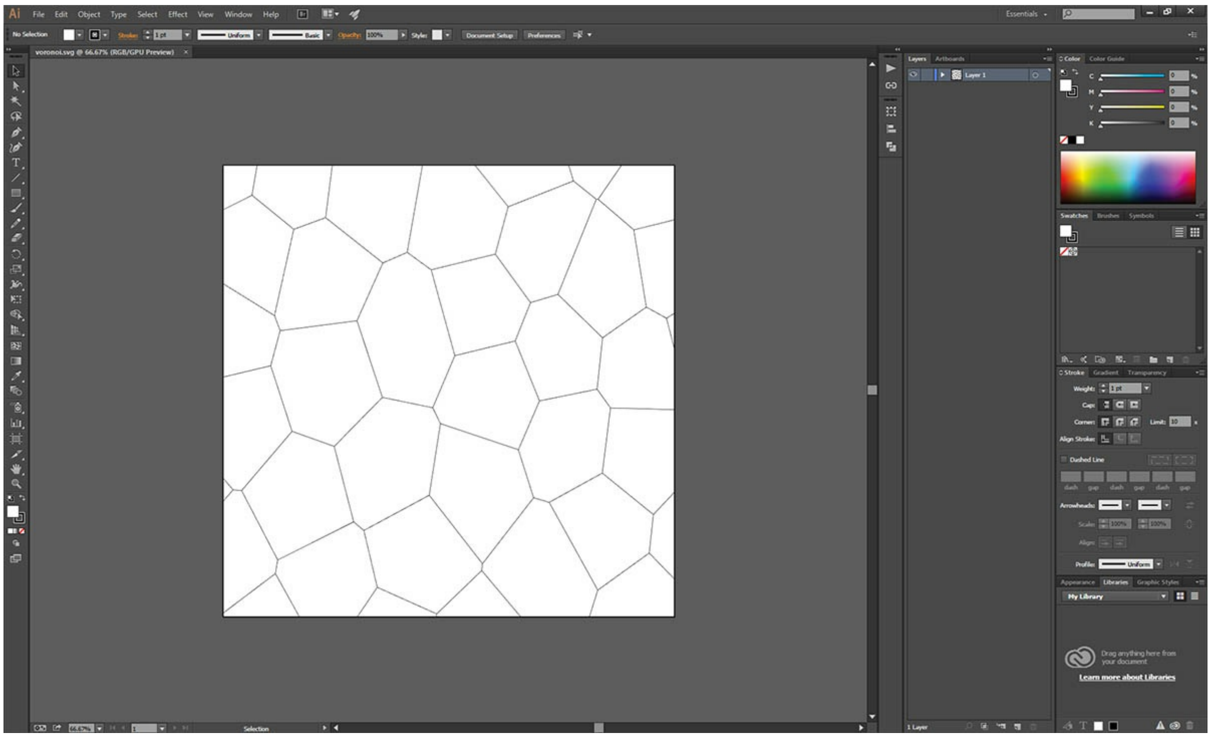
03-NurbsCrvsAndSurface: This example file adds some complexity by working with the "NurbsCrvToSurface" node. The target surface is offset a given distance and the nurbs curve is mapped to the original target surface and the offset surface. From there, the two mapped curves are lofted to create a surface, which is then thickened. This resulting solid has an undulation that is representative of the target surface normals.



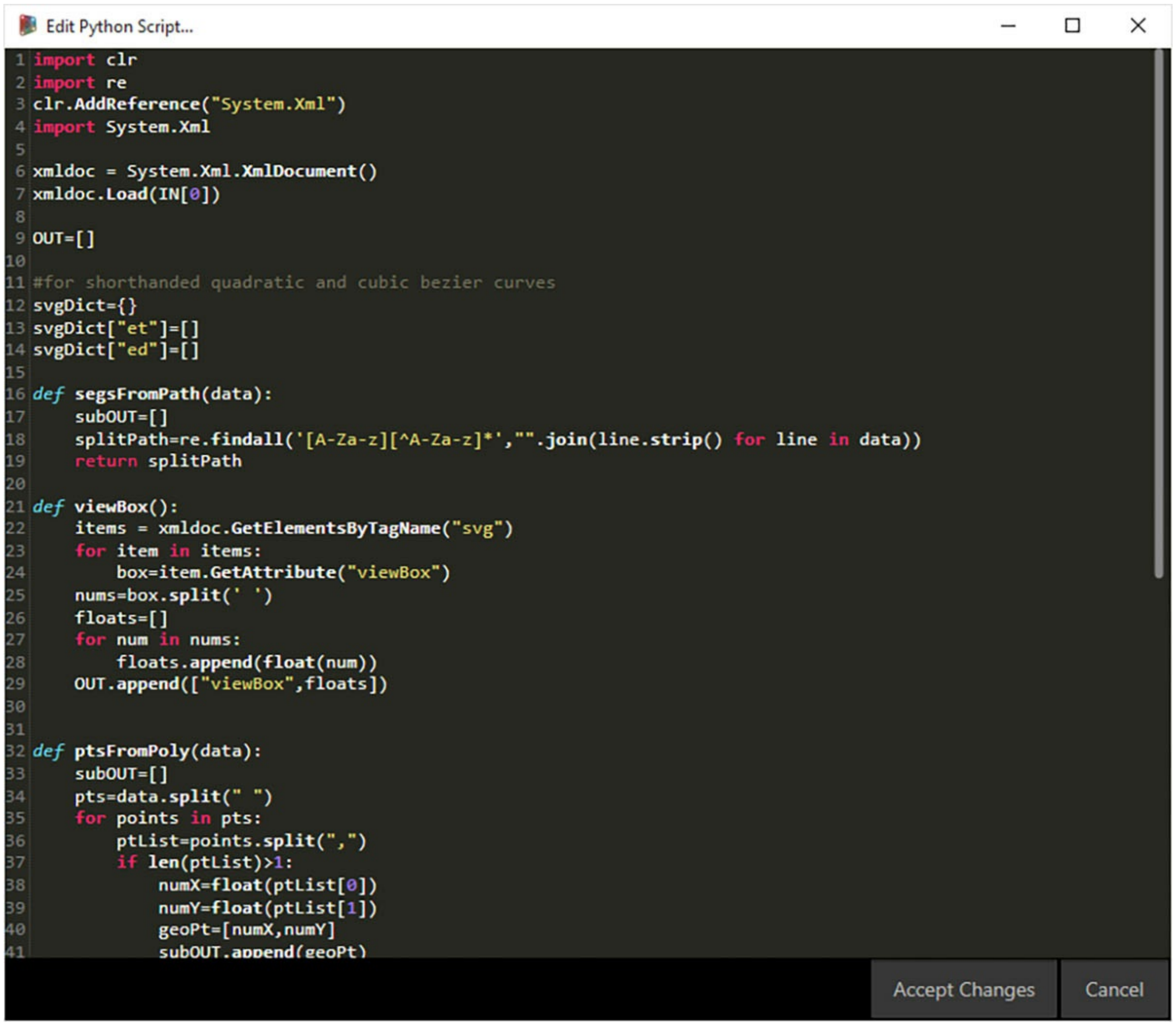
04-PleatedPolysurface-OffsetPoints: This example file demonstrates how to map a pleated polysurface from a source surface to a target surface. The source and target surface are a rectangular surface spanning the grid and a revolved surface, respectively.



04-PleatedPolysurface-OffsetPoints: The source polysurface mapped from the source surface to the target surface.

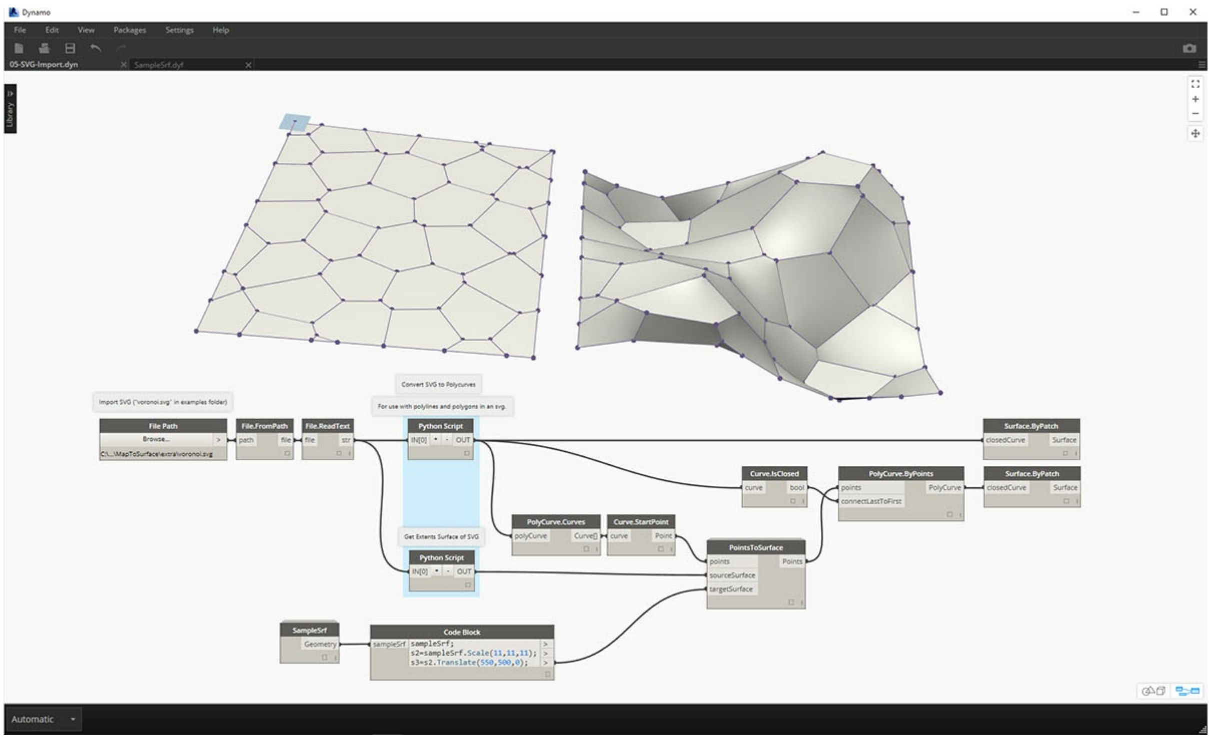


05-SVG-Import: Since the custom nodes are able to map different types of curves, this last file references an SVG file exported from Illustrator and maps the imported curves to a target surface.



```
import clr
import re
clr.AddReference("System.Xml")
import System.Xml

xmldoc = System.Xml.XmlDocument()
xmldoc.Load(IN[0])

OUT=[]

#for shorthanded quadratic and cubic bezier curves
svgDict={}
svgDict["et"]=[]
svgDict["ed"]=[]

def segsFromPath(data):
    subOUT=[]
    splitPath=re.findall('[A-Za-z][^A-Za-z]*',"".join(line.strip() for line in data))
    return splitPath

def viewBox():
    items = xmldoc.GetElementsByTagName("svg")
    for item in items:
        box=item.GetAttribute("viewBox")
    nums=box.split(' ')
    floats=[]
    for num in nums:
        floats.append(float(num))
    OUT.append(["viewBox",floats])


def ptsFromPoly(data):
    subOUT=[]
    pts=data.split(" ")
    for points in pts:
        ptList=points.split(",")
        if len(ptList)>1:
            numX=float(ptList[0])
            numY=float(ptList[1])
            geoPt=[numX,numY]
            subOUT.append(geoPt)
```

05-SVG-Import: By parsing through the syntax of a .svg file, curves are translated from .xml format to Dynamo polycurves.



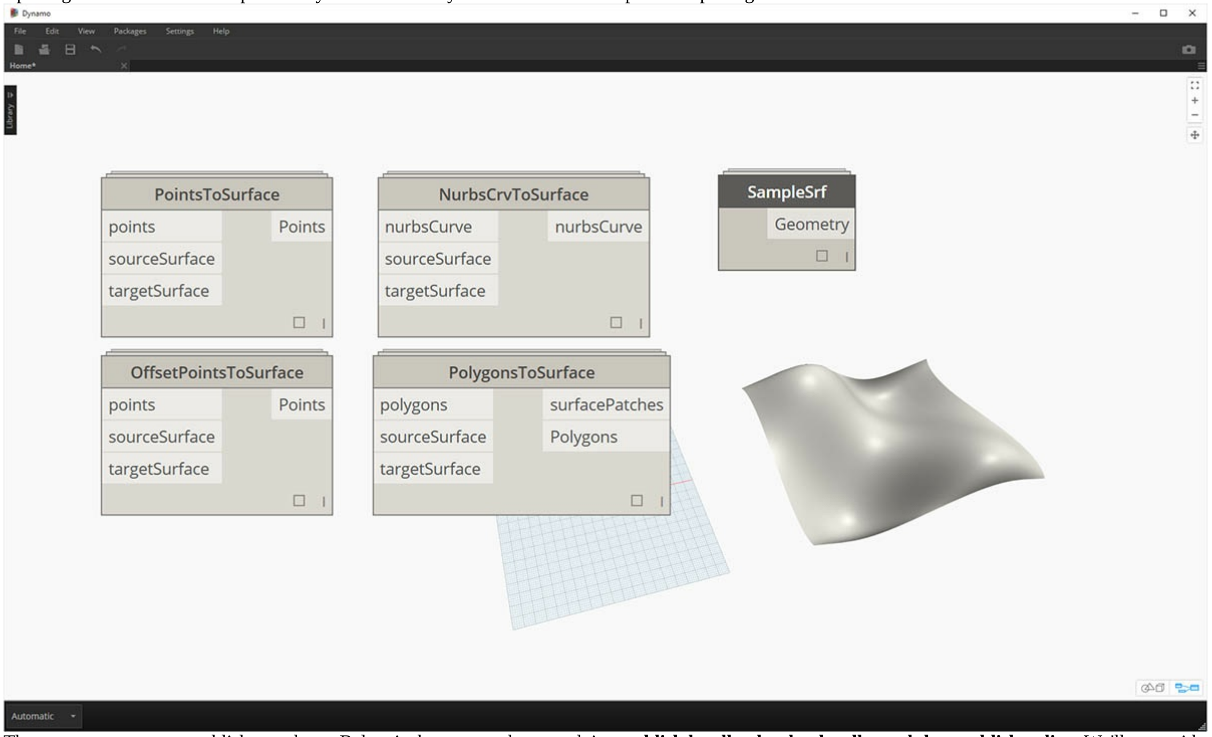
```
sampleSrf;
s2=sampleSrf.Scale(11,11,11);
s3=s2.Translate(550,500,0);
```

05-SVG-Import: The imported curves are mapped to a target surface. This allows us to explicitly (point-and-click) design a panelization in Illustrator, import into Dynamo, and apply to a target surface.

Publishing a Package

Publishing a Package

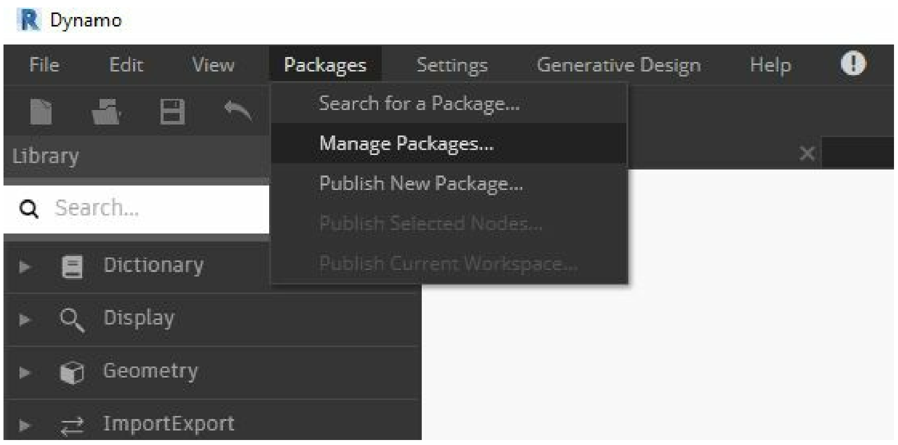
In the previous sections, we dove into the details of how our *MapToSurface* package is set up with custom nodes and example files. But how do we publish a package that has been developed locally? This case study demonstrates how to publish a package from a set of files in a local folder.



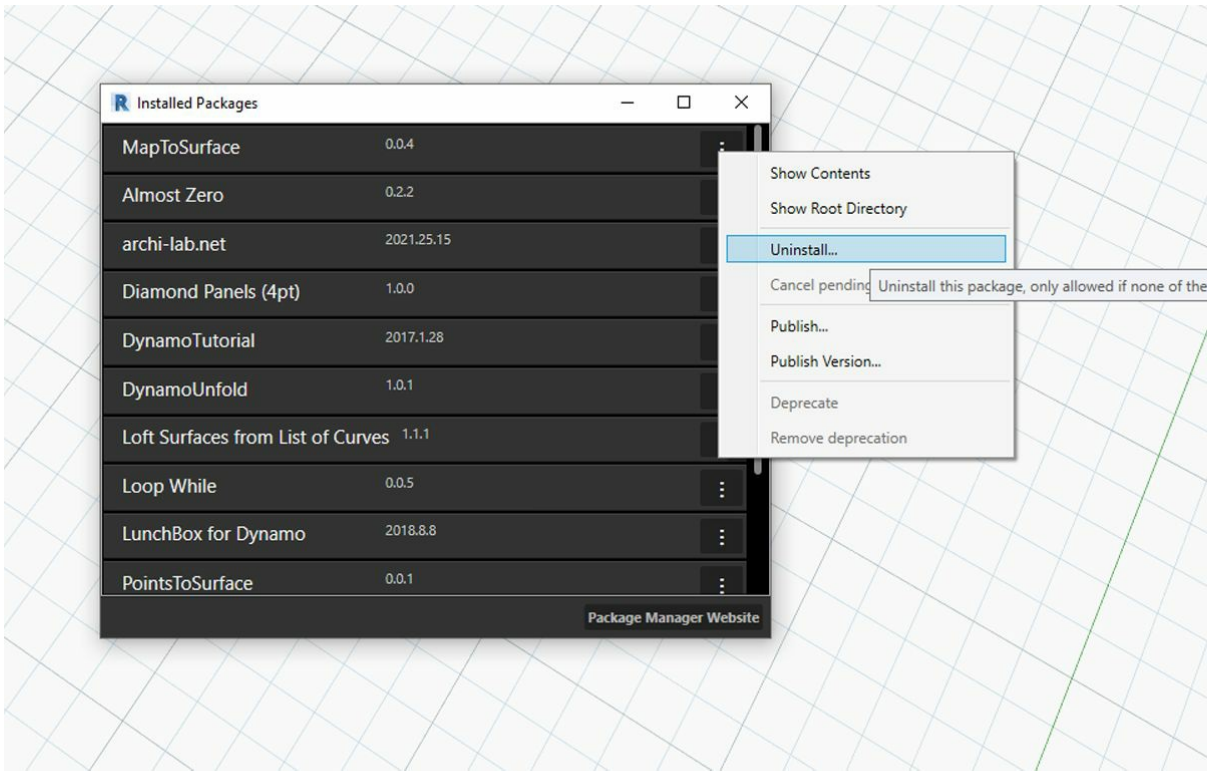
There are many ways to publish a package. Below is the process that we advise: **publish locally, develop locally, and then publish online**. We'll start with a folder containing all of the files in the package.

Uninstalling a Package

Before we jump into publishing the MapToSurface package, if you installed the package from the previous lesson, uninstall it so that you're not working with identical packages.



Begin by going to *Packages>Manage Packages...*

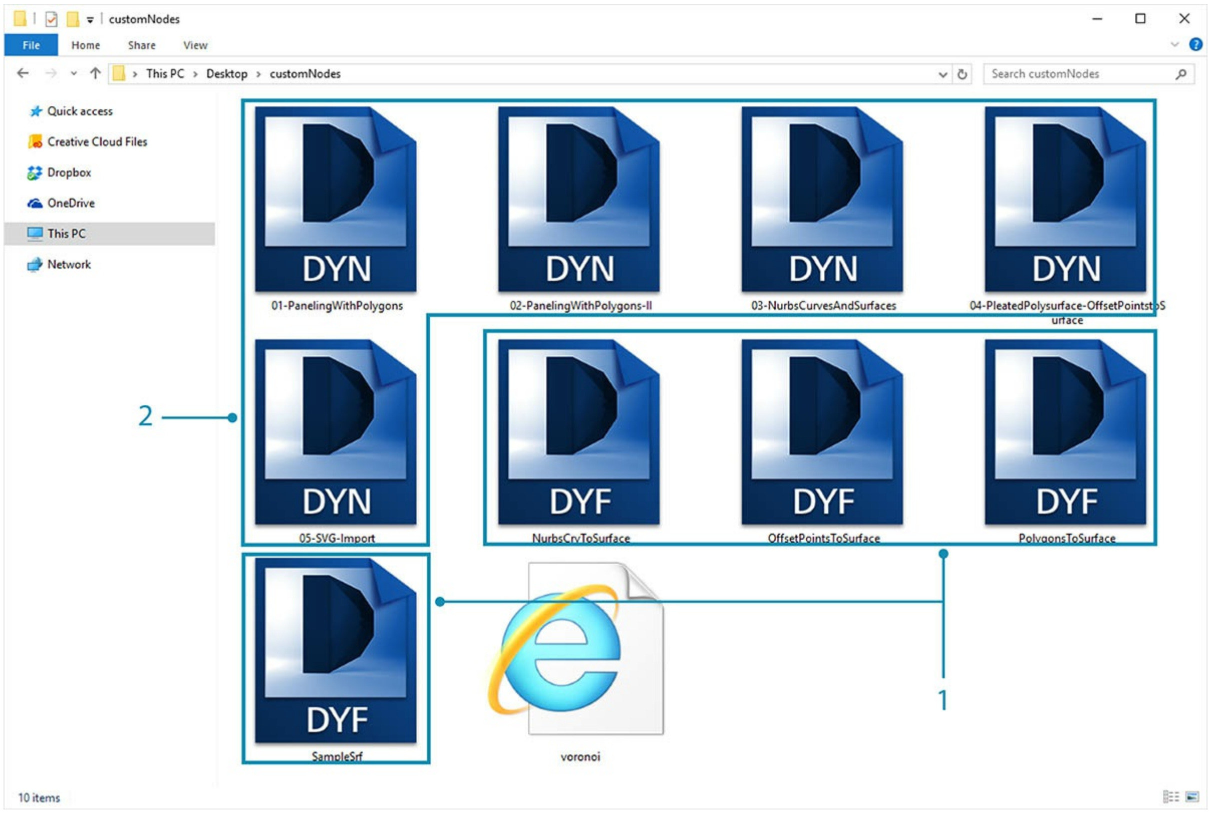


Select the button corresponding to *"MapToSurface"* and select *"Uninstall..."*. Then restart Dynamo. When reopening, when you check the *"Manage Packages"* window, the *MapToSurface* should no longer be there. Now we're ready to start from the beginning!

Publishing a Package Locally

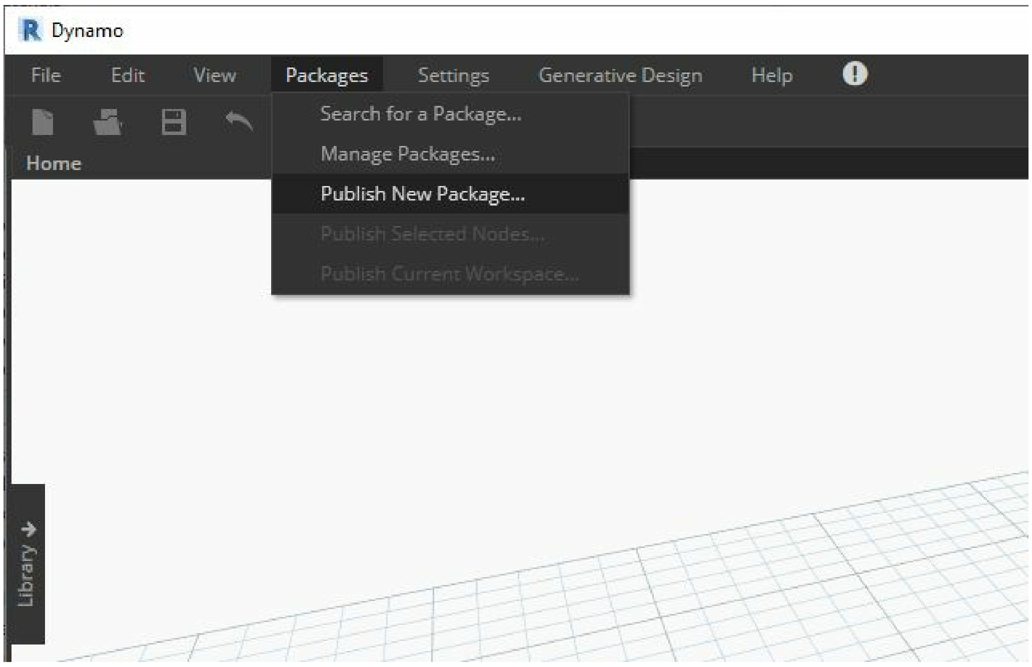
Note: As of writing this, Dynamo package publication is only enabled in Dynamo for Revit and Dynamo for Civil 3d. Dynamo Sandbox does not have publishing functionality.

Download and unzip the example files that accompany this package case study (Right click and "Save Link As..."). A full list of example files can be found in the Appendix. [MapToSurface.zip]

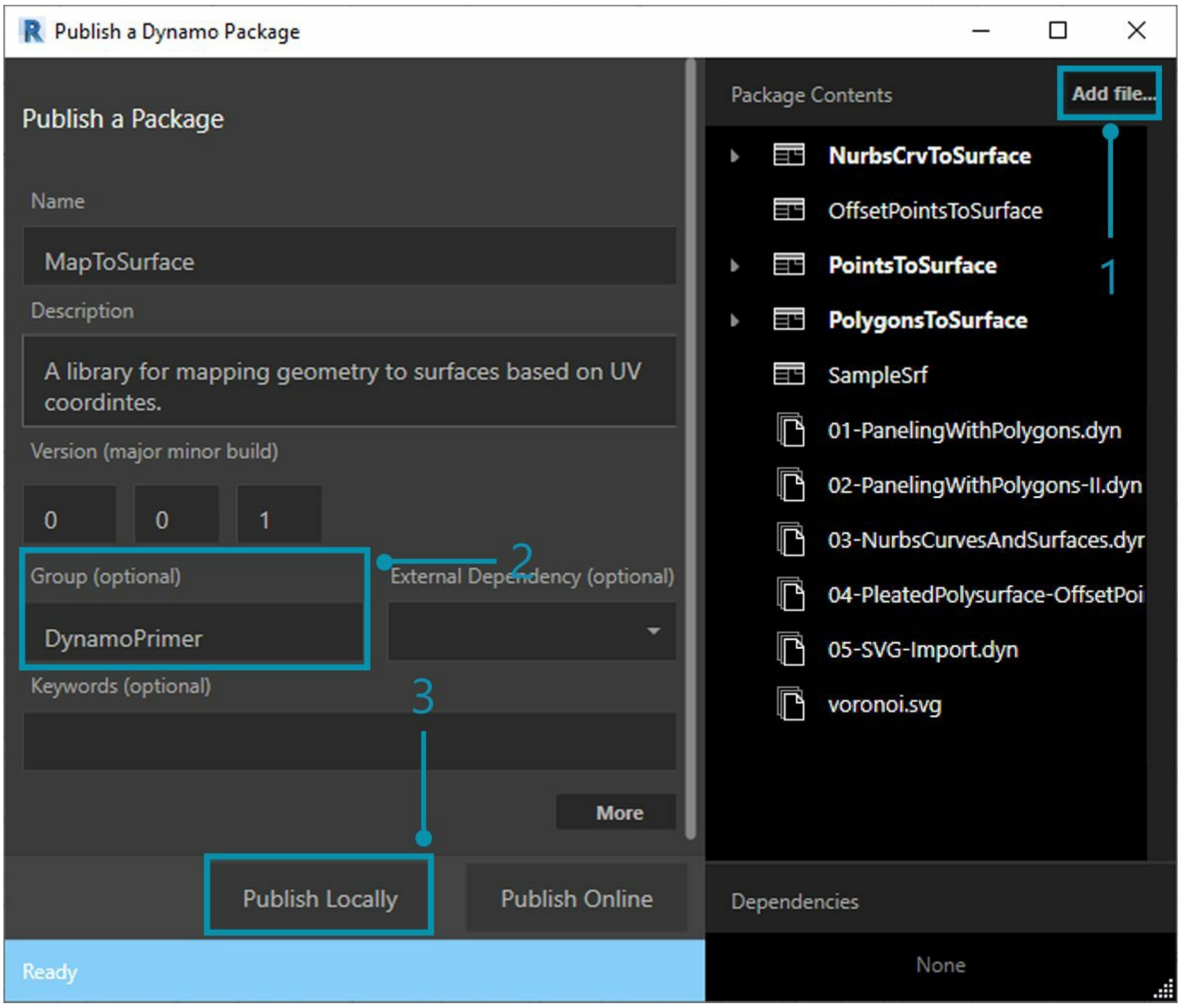


This is the first submission for our package, and we've placed all of the example files and custom nodes into one folder. With this folder prepared, we're ready to upload to the Dynamo Package Manager.

1. This folder contains five custom nodes (.dyf).
2. This folder also contains five example files (.dyn) and one imported vector file (.svg). These files will serve as introductory exercises to show the user how to work with the custom nodes.



In Dynamo, begin by clicking *Packages>Publish New Package...*


Publish a Dynamo Package

Publish a Package

Name: MapToSurface

Description: A library for mapping geometry to surfaces based on UV coordintes.

Version (major minor build): 0 0 1

Group (optional): DynamoPrimer

External Dependency (optional):

Keywords (optional):

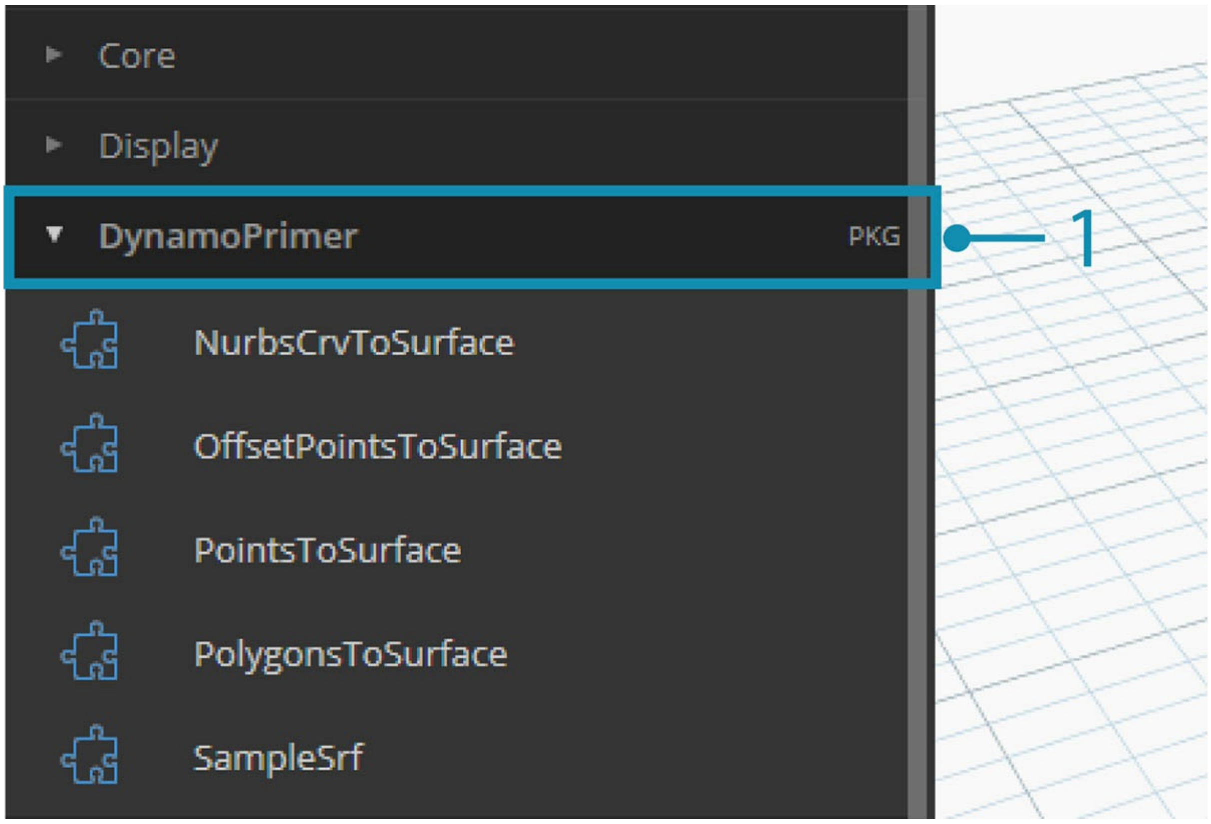
Package Contents

- NurbsCrvToSurface
- OffsetPointsToSurface
- PointsToSurface
- PolygonsToSurface
- SampleSrf
- 01-PanelingWithPolygons.dyn
- 02-PanelingWithPolygons-II.dyn
- 03-NurbsCurvesAndSurfaces.dyr
- 04-PleatedPolysurface-OffsetPoi
- 05-SVG-Import.dyn
- voronoi.svg

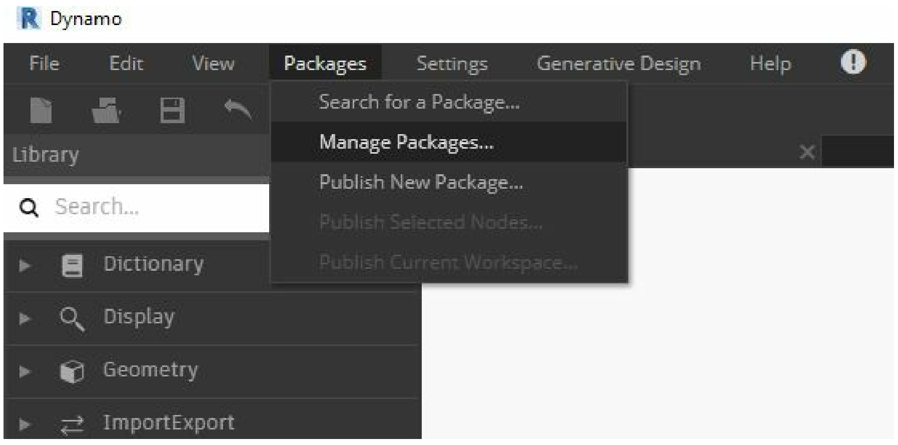
Dependencies: None

In the *"Publish a Dynamo Package"* window, we've filled out the relevant forms on the left of the window.

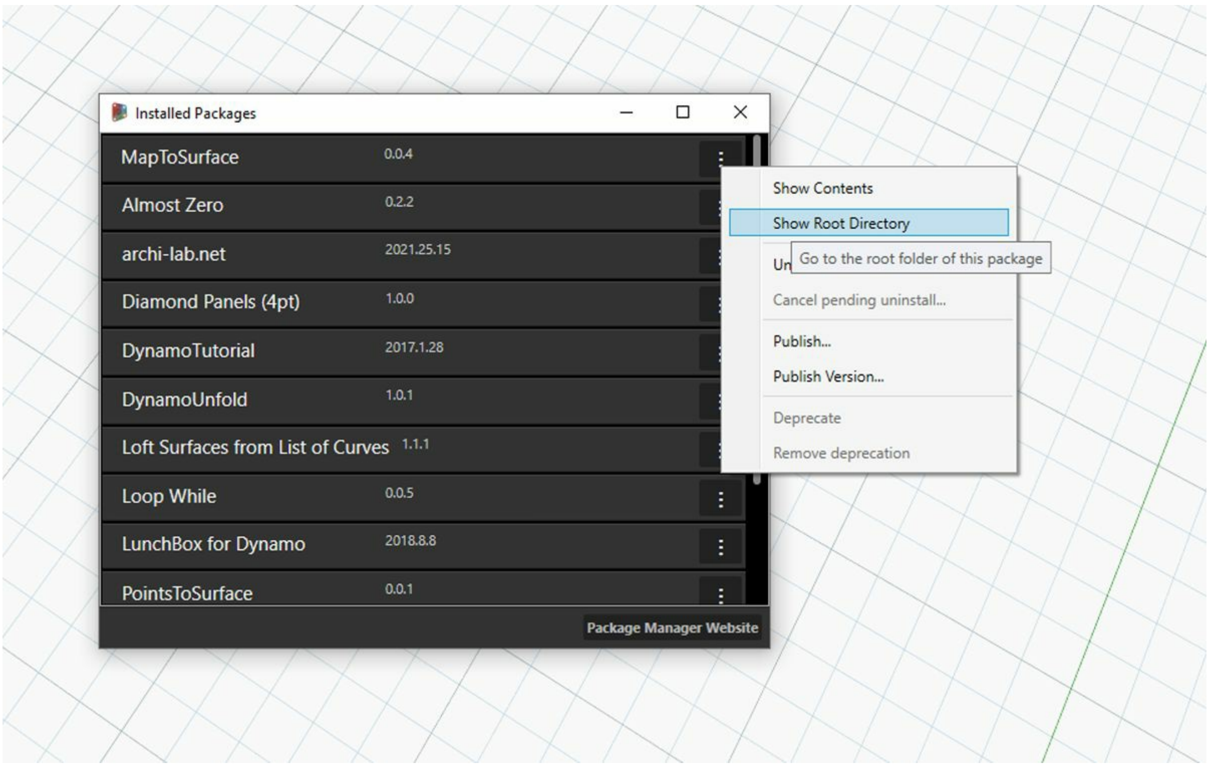
1. By clicking *"Add File"*, we've also added the files from the folder structure on the right side of the screen (to add files which are not .dyf files, be sure to change your file type in the browser window to **"All Files(.)"**. Notice that we've added every file, custom node (.dyf) or example file (.dyn), indiscriminately. Dynamo will categories these items when we publish the package.
2. The "Group" field defines which group to find the custom nodes in the Dynamo UI.
3. Publish by clicking "Publish Locally". If you're following along, be certain to click *"Publish Locally"* and **not** *"Publish Online"*; we don't want a bunch of duplicate packages on the Package Manager.



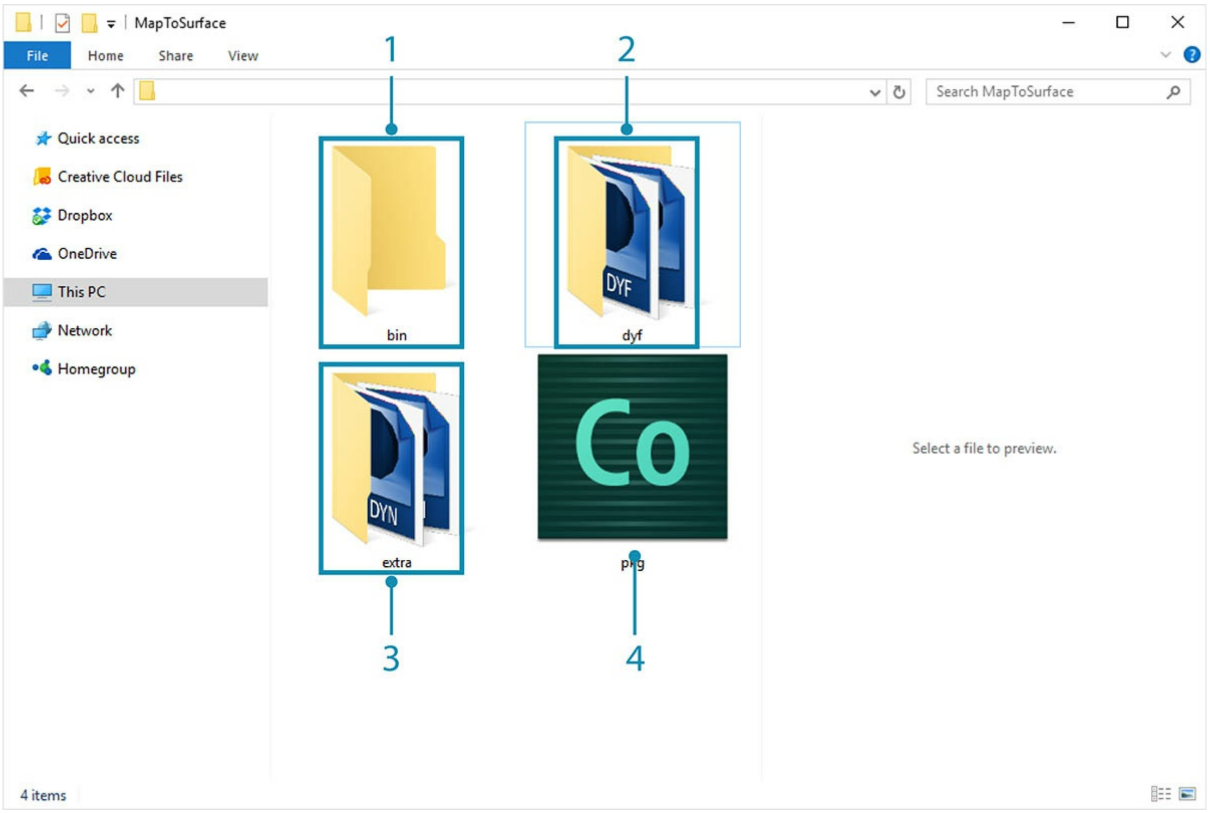
1. After publishing, the custom nodes should be available under the "DynamoPrimer" group or your Dynamo Library.



Now let's look at the root directory to see how Dynamo has formatted the package we just created. Do this by clicking *Packages>Manage Packages...*



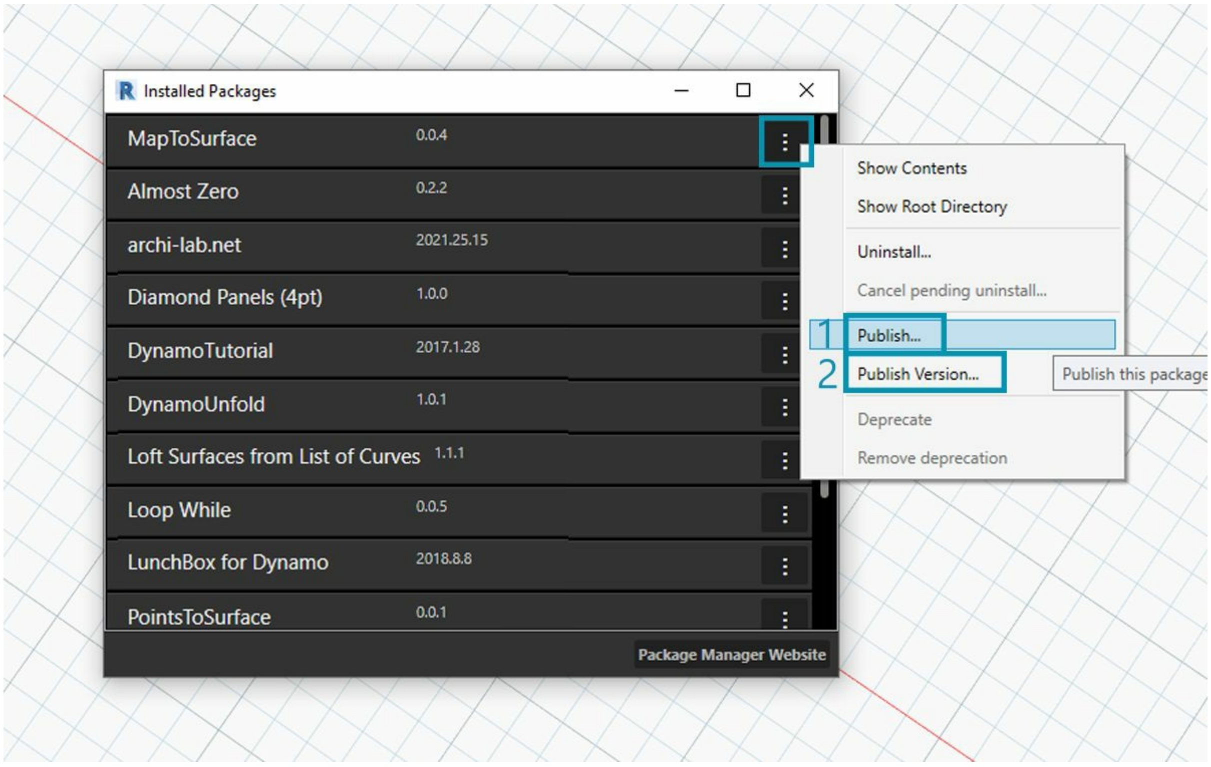
In the manage packages window, click on the three vertical dots to the right of *"MapToSurface"* and choose *"Show Root Directory"*.



Notice that the root directory is in the local location of your package (remember, we published the package "locally"). Dynamo is currently referencing this folder to read custom nodes. It's therefore important to locally publish the directory to a permanent folder location (ie: not your desktop). Here is the Dynamo package folder breakdown:

1. The *bin* folder houses .dll files created with C# or Zero-Touch libraries. We don't have any for this package so this folder is blank for this example.
2. The *dyf* folder houses the custom nodes. Opening this will reveal all of the custom nodes (.dyf files) for this package.
3. The extra folder houses all additional files. These files are likely to be Dynamo Files (.dyn) or any additional files required (.svg, .xls, .jpeg, .sat, etc.).
4. The pkg file is a basic text file defining the package settings. This is automated in Dynamo, but can be edited if you want to get into the details.

Publishing a Package Online



Note: please do not follow along with this step unless you are actually publishing a package of your own!

1. When you're ready to publish, in the "Manage Packages" window, select the button the right of MapToSurface and choose *Publish...*
2. If you're updating a package that has already been published, choose "Publish Version" and Dynamo will update your package online based on the new files in that package's root directory. Simple as that!

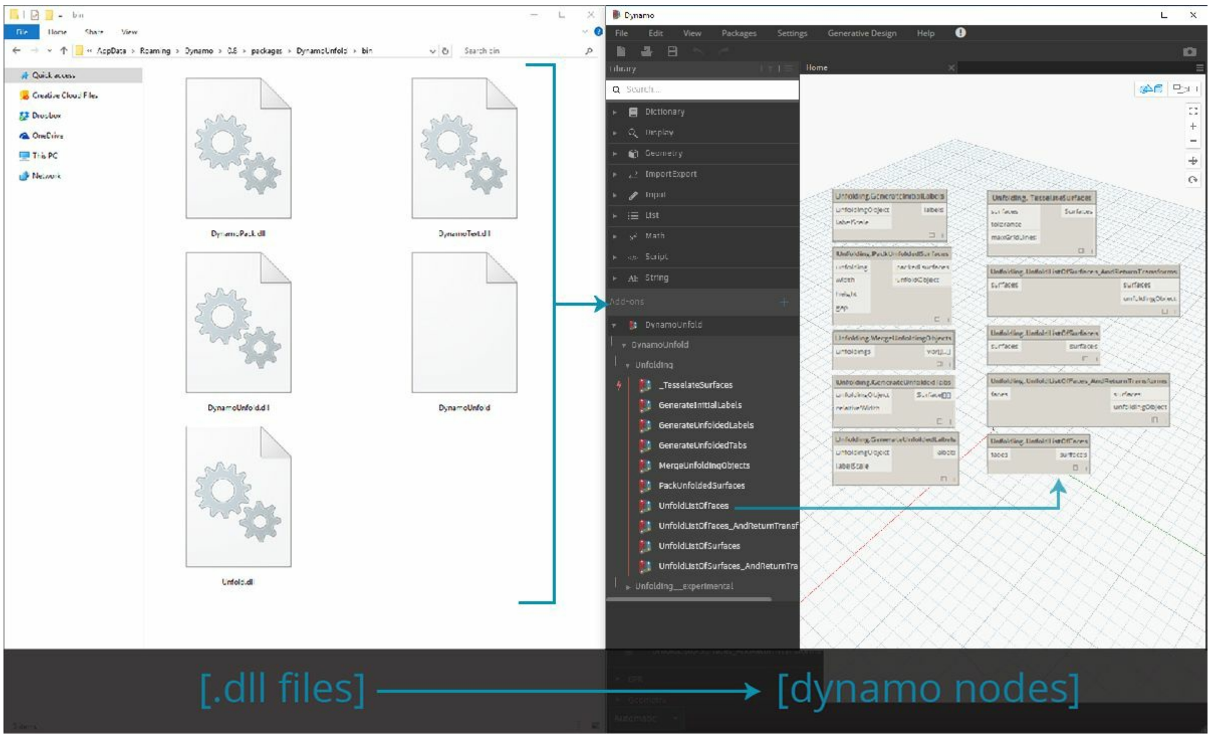
Publish Version...

When you update the files in your published package's root folder, you can publish a new version of the package by selecting *"Publish Version..."* in the *Manage Packages* window. This is a seamless way to make necessary updates to your content and share with the community. *Publish Version* will only work if you're the maintainer of the package.

Zero-Touch Importing

What is Zero-Touch?

Zero-Touch Importing refers to a simple point-and-click method for importing C# libraries. Dynamo will read the public methods of a *.dll* file and convert them to Dynamo nodes. You can use Zero-Touch to develop your own custom nodes and packages, and to import external libraries into the Dynamo environment.



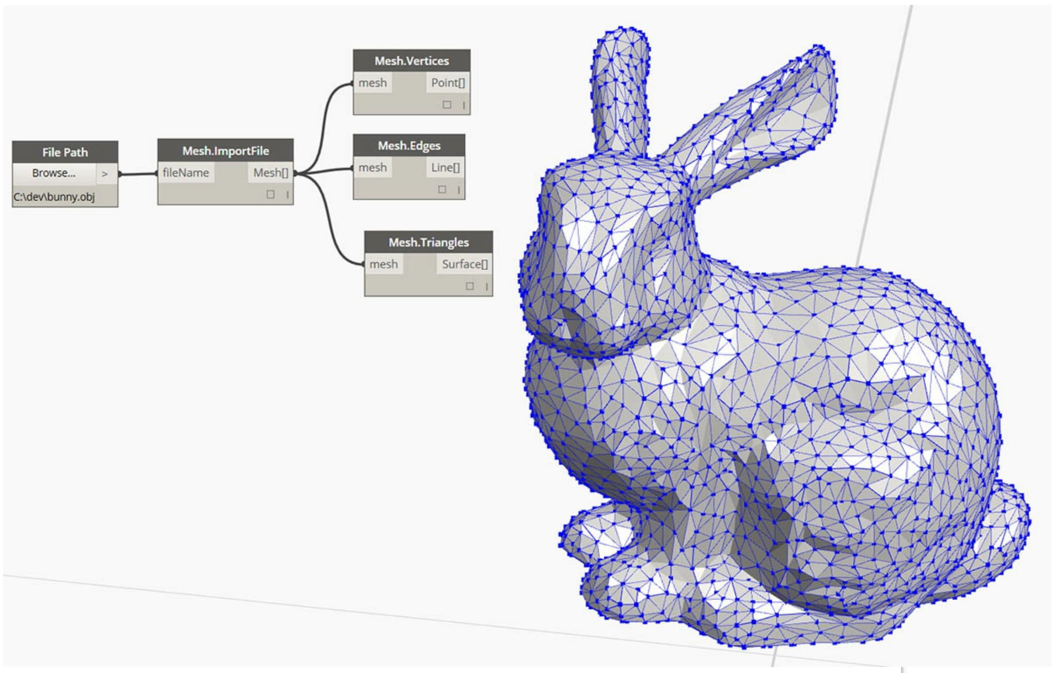
With Zero-Touch, you can actually import a library which was not necessarily developed for Dynamo and create a suite of new nodes. The current Zero-Touch functionality demonstrates the cross-platform mentality of the Dynamo Project.

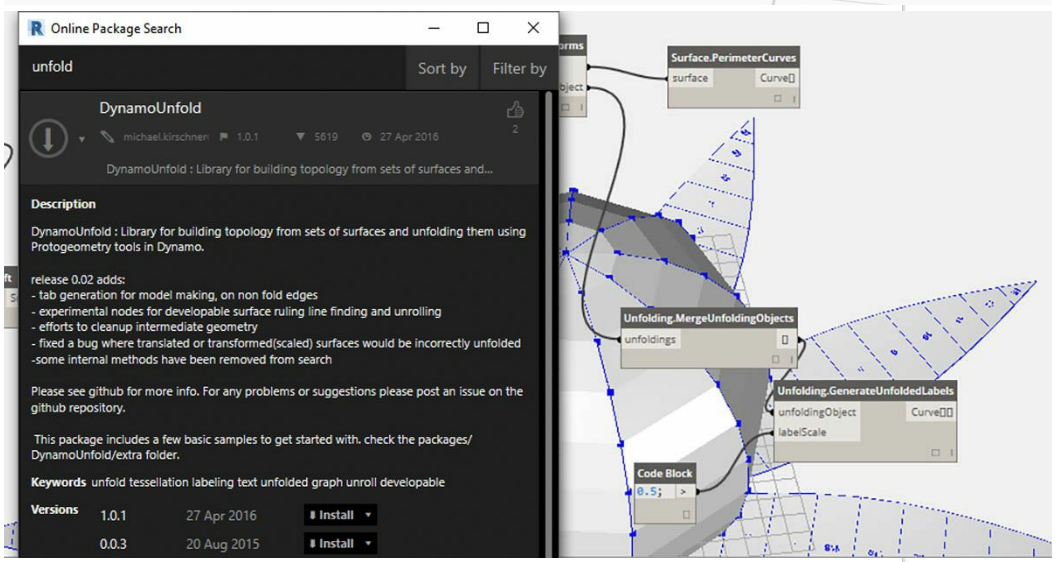
This section demonstrates how to use Zero-Touch to import a third party library. For information on developing your own Zero-Touch Library, reference the Dynamo wiki page.

Zero-Touch Packages

Zero-touch packages are a good complement to user-defined custom nodes. A few packages which use C# libraries are listed in the table below. For more detailed information on packages, visit the Packages section in the Appendix.

Logo/Image





Online Package Search

unfold

Sort by Filter by

DynamoUnfold

michael.kirschner 1.0.1 5619 27 Apr 2016

DynamoUnfold : Library for building topology from sets of surfaces and...

Description

DynamoUnfold : Library for building topology from sets of surfaces and unfolding them using Protogeometry tools in Dynamo.

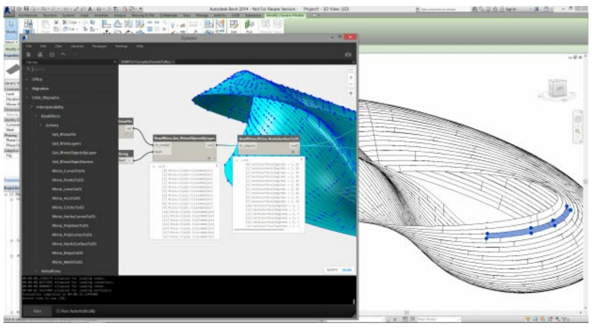
release 0.02 adds:
- tab generation for model making, on non fold edges
- experimental nodes for developable surface ruling line finding and unrolling
- efforts to cleanup intermediate geometry
- fixed a bug where translated or transformed(scaled) surfaces would be incorrectly unfolded
-some internal methods have been removed from search

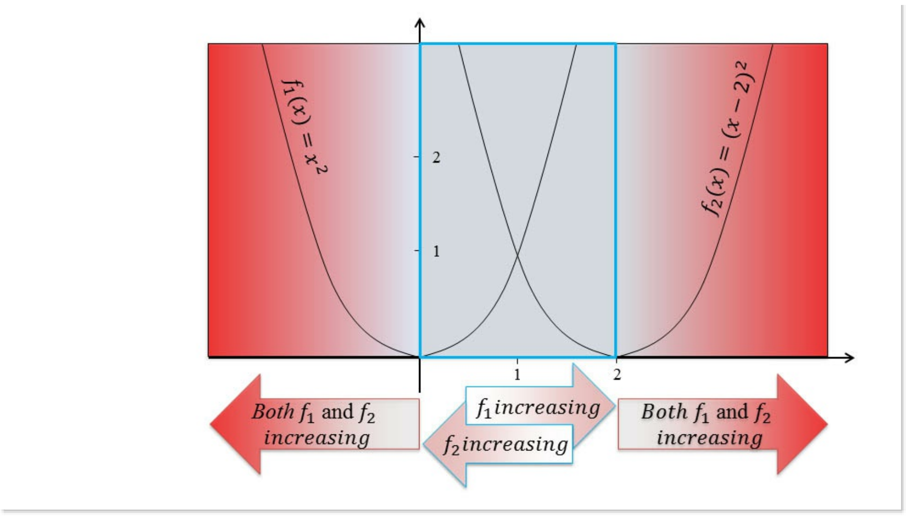
Please see github for more info. For any problems or suggestions please post an issue on the github repository.

This package includes a few basic samples to get started with. check the packages/DynamoUnfold/extra folder.

Keywords unfold tessellation labeling text unfolded graph unroll developable

Versions			
	1.0.1	27 Apr 2016	Install
	0.0.3	20 Aug 2015	Install



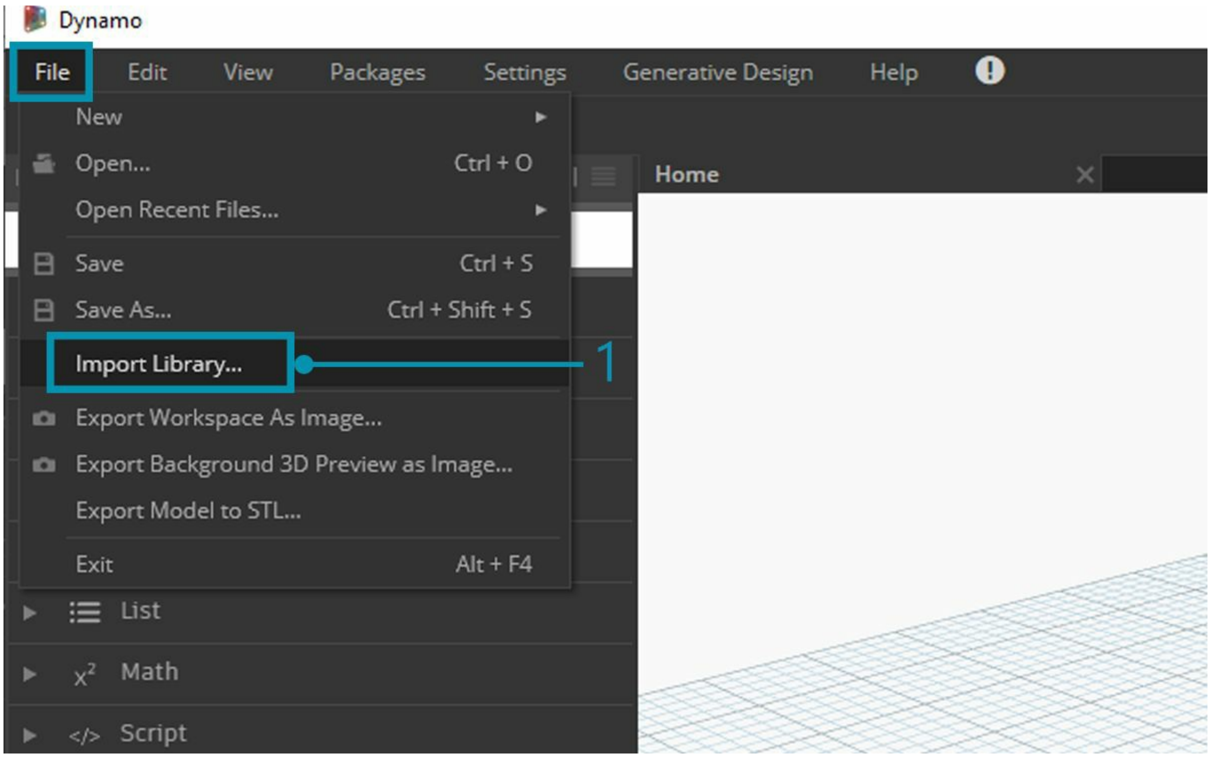


Case Study - Importing AForge

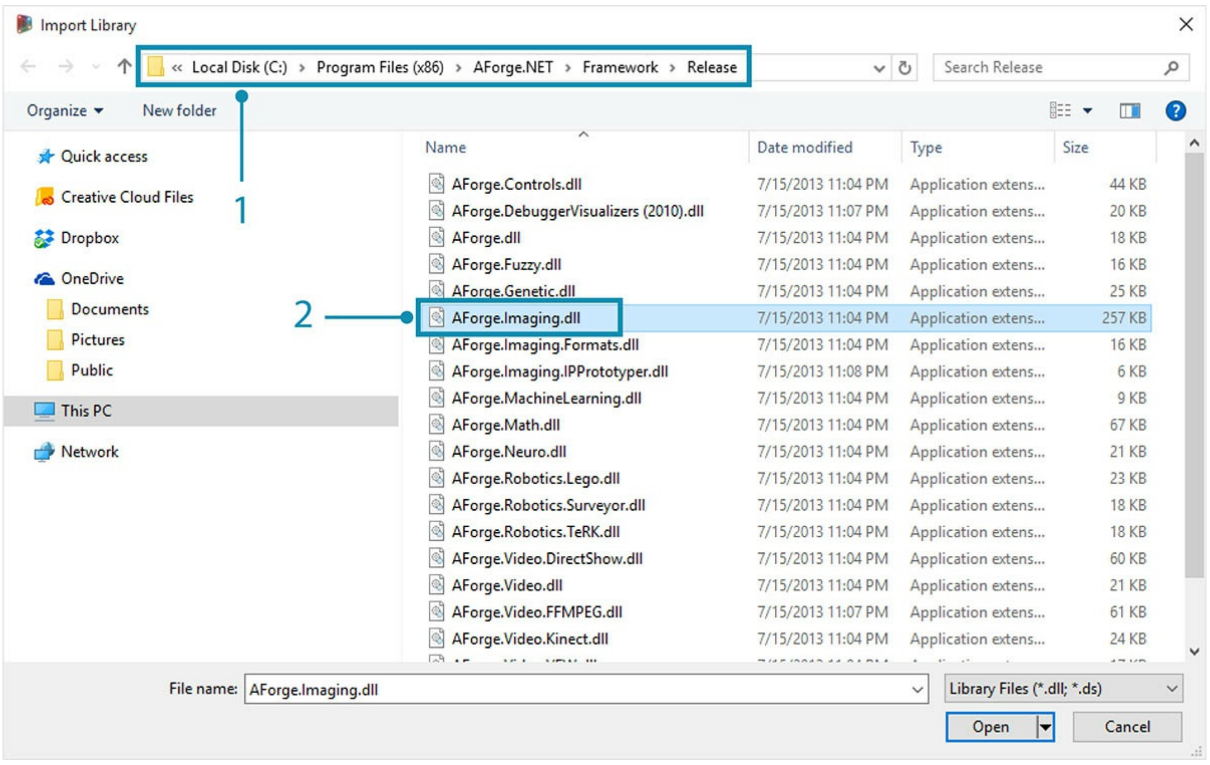
In this case study, we'll show how to import the AForge external *.dll* library. AForge is a robust library which offers a range of functionality from image processing to artificial intelligence. We'll reference the imaging class in AForge to do a few image processing exercises below.

Download and unzip the example files that accompany this package case study (Right click and "Save Link As..."). A full list of example files can be found in the Appendix. Zero-Touch-Examples.zip.

1. Let's beging by downloading AForge. On the AForge download page, select *[Download Installer]* and install after download has completed.

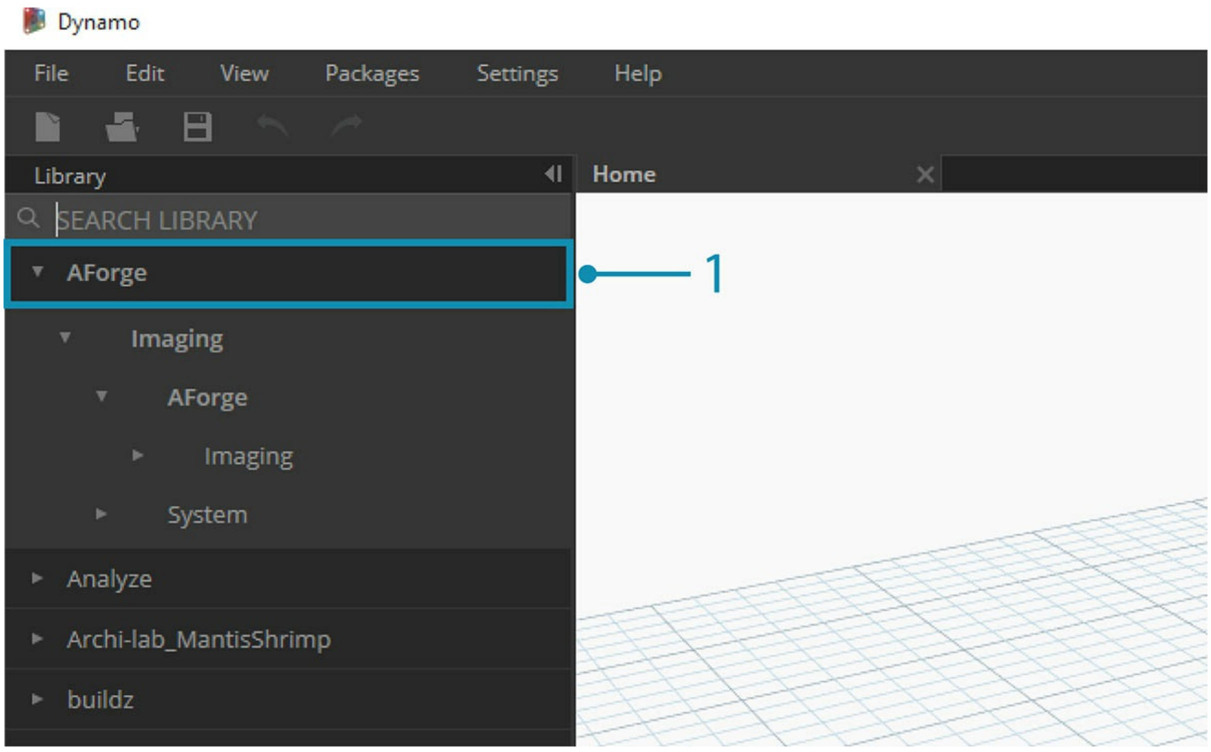


1. In Dynamo, create a new file and select *File > Import Library...*



Name	Date modified	Type	Size
AForge.Controls.dll	7/15/2013 11:04 PM	Application extens...	44 KB
AForge.DebuggerVisualizers (2010).dll	7/15/2013 11:07 PM	Application extens...	20 KB
AForge.dll	7/15/2013 11:04 PM	Application extens...	18 KB
AForge.Fuzzy.dll	7/15/2013 11:04 PM	Application extens...	16 KB
AForge.Genetic.dll	7/15/2013 11:04 PM	Application extens...	25 KB
AForge.Imaging.dll	7/15/2013 11:04 PM	Application extens...	257 KB
AForge.Imaging.Formats.dll	7/15/2013 11:04 PM	Application extens...	16 KB
AForge.Imaging.IPPrototyper.dll	7/15/2013 11:08 PM	Application extens...	6 KB
AForge.MachineLearning.dll	7/15/2013 11:04 PM	Application extens...	9 KB
AForge.Math.dll	7/15/2013 11:04 PM	Application extens...	67 KB
AForge.Neuro.dll	7/15/2013 11:04 PM	Application extens...	21 KB
AForge.Robotics.Lego.dll	7/15/2013 11:04 PM	Application extens...	23 KB
AForge.Robotics.Surveyor.dll	7/15/2013 11:04 PM	Application extens...	18 KB
AForge.Robotics.TeRK.dll	7/15/2013 11:04 PM	Application extens...	18 KB
AForge.Video.DirectShow.dll	7/15/2013 11:04 PM	Application extens...	60 KB
AForge.Video.dll	7/15/2013 11:04 PM	Application extens...	21 KB
AForge.Video.FFMPEG.dll	7/15/2013 11:07 PM	Application extens...	61 KB
AForge.Video.Kinect.dll	7/15/2013 11:04 PM	Application extens...	24 KB

1. In the pop-up window, navigate to the release folder in your AForge install. This will likely be in a folder similar to this one: *C:\Program Files (x86)\AForge.NET\Framework\Release*.
2. **AForge.Imaging.dll:** We only want to use this one file from the AForge library for this case study. Select this *.dll* and hit *"Open"*.



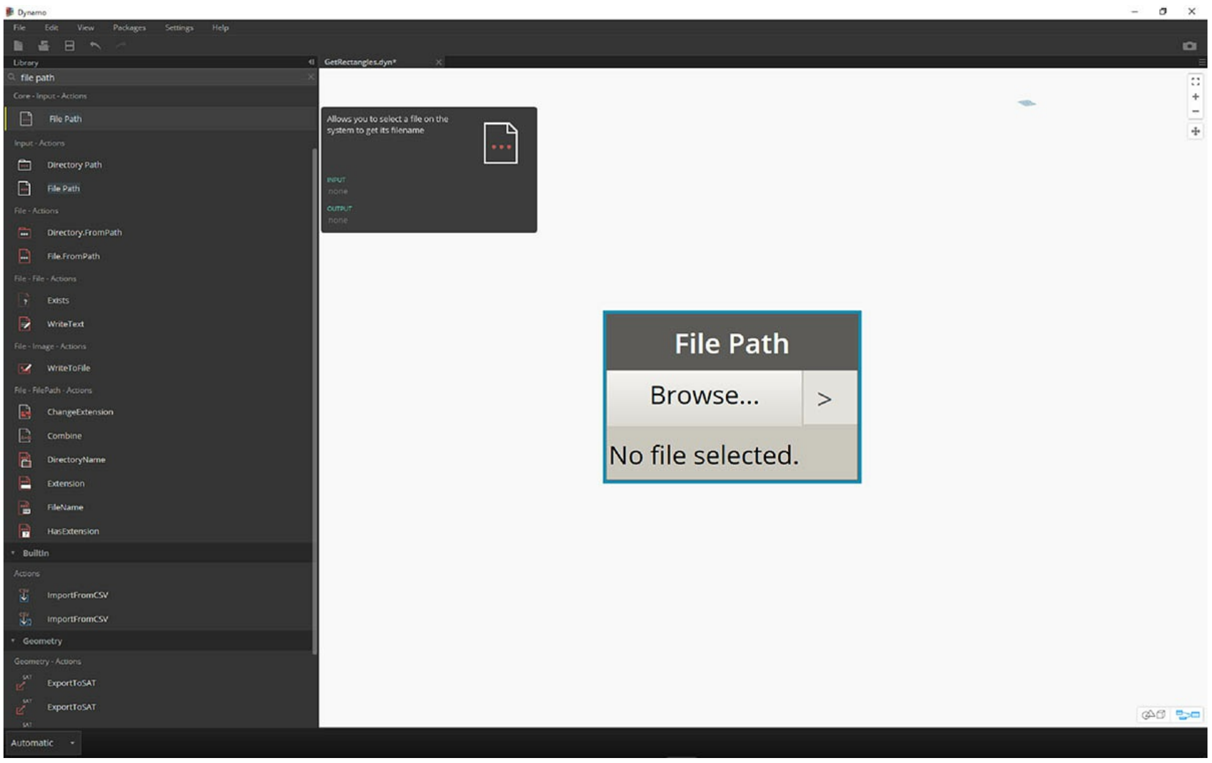
1. Back in Dynamo, you should see an *"AForge"* group of nodes added to your Library Toolbar. We now have access to the AForge imaging library from our visual program!

Exercise 1 - Edge Detection

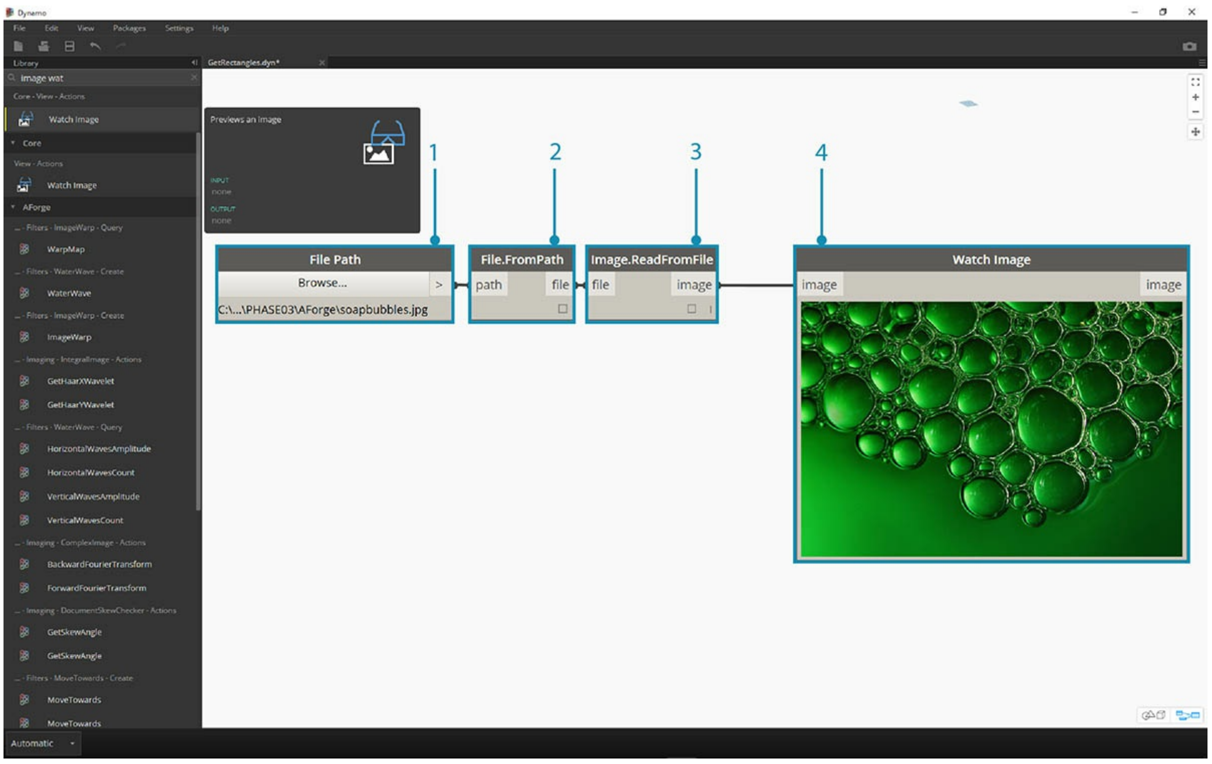
Now that the library's imported, we'll start off simple with this first exercise. We'll do some basic image processing on a sample image to show how AForge image filters. We'll use the *"Watch Image"* node to show our results and apply filters in Dynamo similar to those in Photoshop.

Download and unzip the example files that accompany this package case study (Right click and "Save Link As..."). A full list of example files can be found in the Appendix. ZeroTouchImages.zip

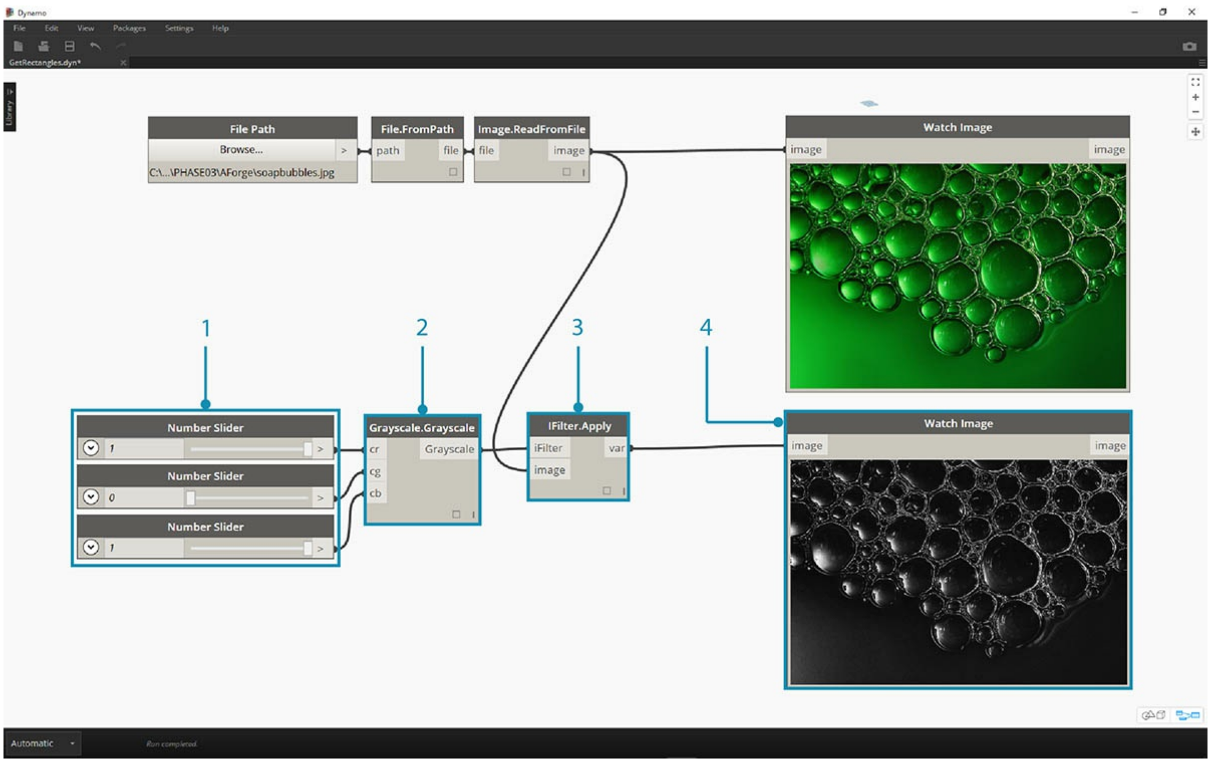
Now that the library's imported, we'll start off simple with this first exercise (*01-EdgeDetection.dyn*). We'll do some basic image processing on a sample image to show how AForge image filters. We'll use the *"Watch Image"* node to show our results and apply filters in Dynamo similar to those in Photoshop


File Path
Browse...
>
No file selected.
Allows you to select a file on the system to get its filename

First, we want to import an image to work with. Add a *File Path* node to the canvas and select "soapbubbles.jpg" from the downloaded exercise folder (photo cred: flickr).

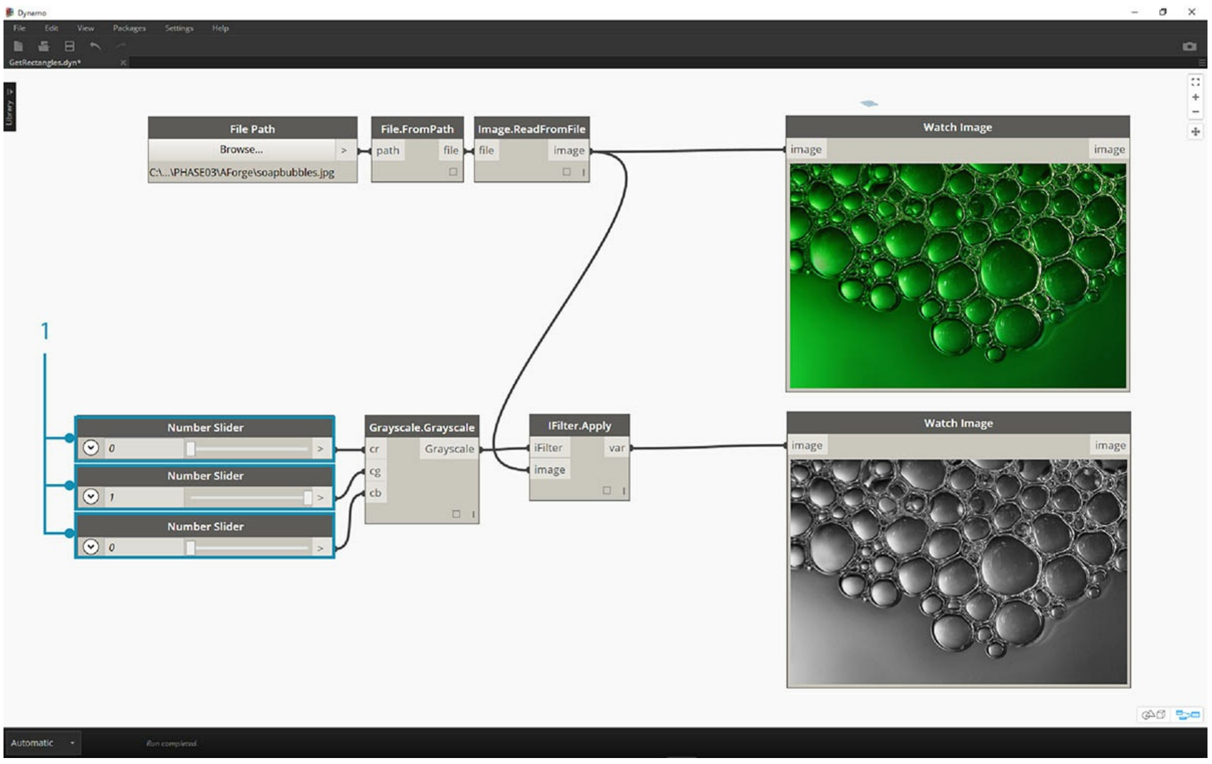


1. The File Path node simply provides a String of the path to the image we've selected. We need to convert this File Path to an image in the Dynamo environment.
2. Connect the File Path node to the File.FromPath node.
3. To convert this File into an Image, we'll use the Image.ReadFromFile node.
4. Last, let's see the result! Drop a Watch Image node onto the canvas and connect to Image.ReadFromFile. We haven't used AForge yet, but we've successfully imported an image into Dynamo.



Under AForge.Imaging.AForge.Filters (in the navigation menu), you'll notice that there is a wide array of filters available. We're going to use one of these filters now to desaturate an image based on threshold values.

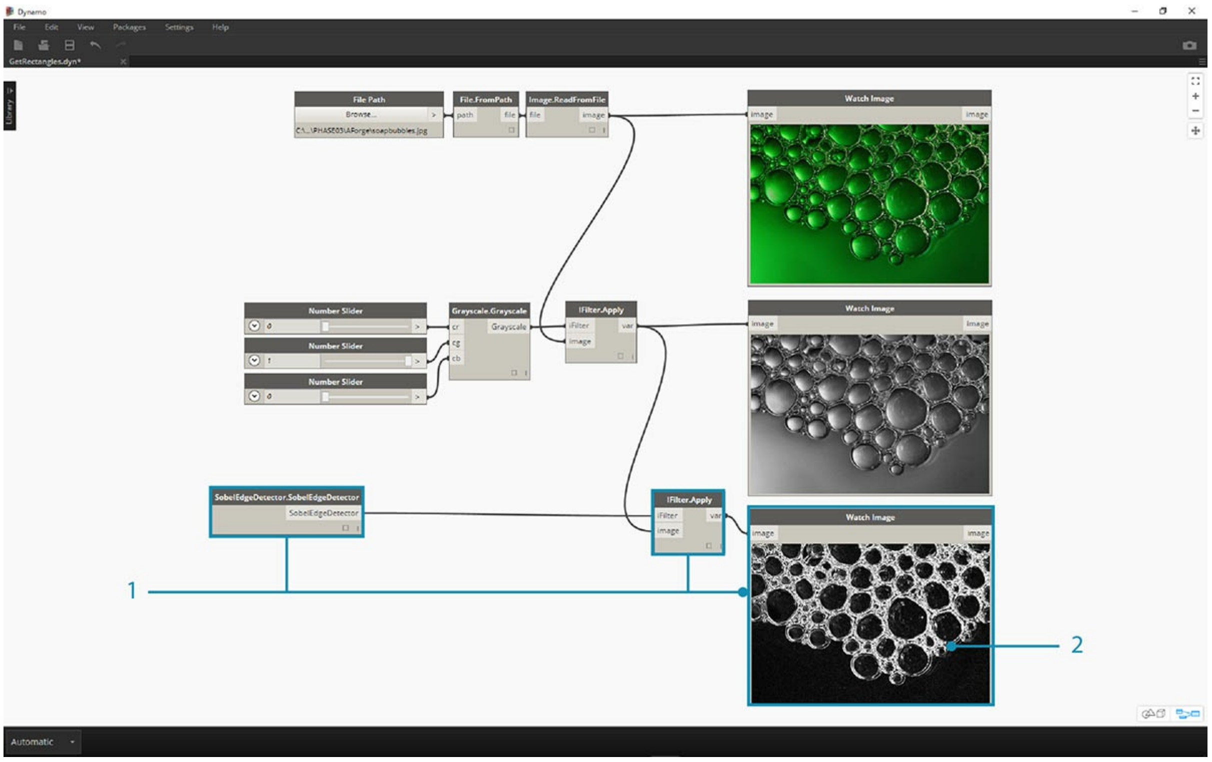
1. Drop three sliders onto the canvas, change their ranges to be from 0 to 1 and their step values to be 0.01.
2. Add the Grayscale.Grayscale node to the canvas. This is an AForge filter which applies a grayscale filter to an image. Connect the three sliders from step 1 into cr, cg, and cb. Change the top and bottom sliders to have a value of 1 and the middle slider to have a value of 0.
3. In order to apply the Grayscale filter, we need an action to perform on our image. For this, we use IFilter.Apply. Connect the image into the image input and Grayscale.Grayscale into the iFilter input.
4. Plugging into a Watch Image node, we get a desaturated image.



We can have control over how to desaturate this image based on threshold values for red, green, and blue. These are defined by the inputs to the Grayscale.Grayscale node. Notice that the image looks pretty dim - this is because the green value is set to 0 from our slider.

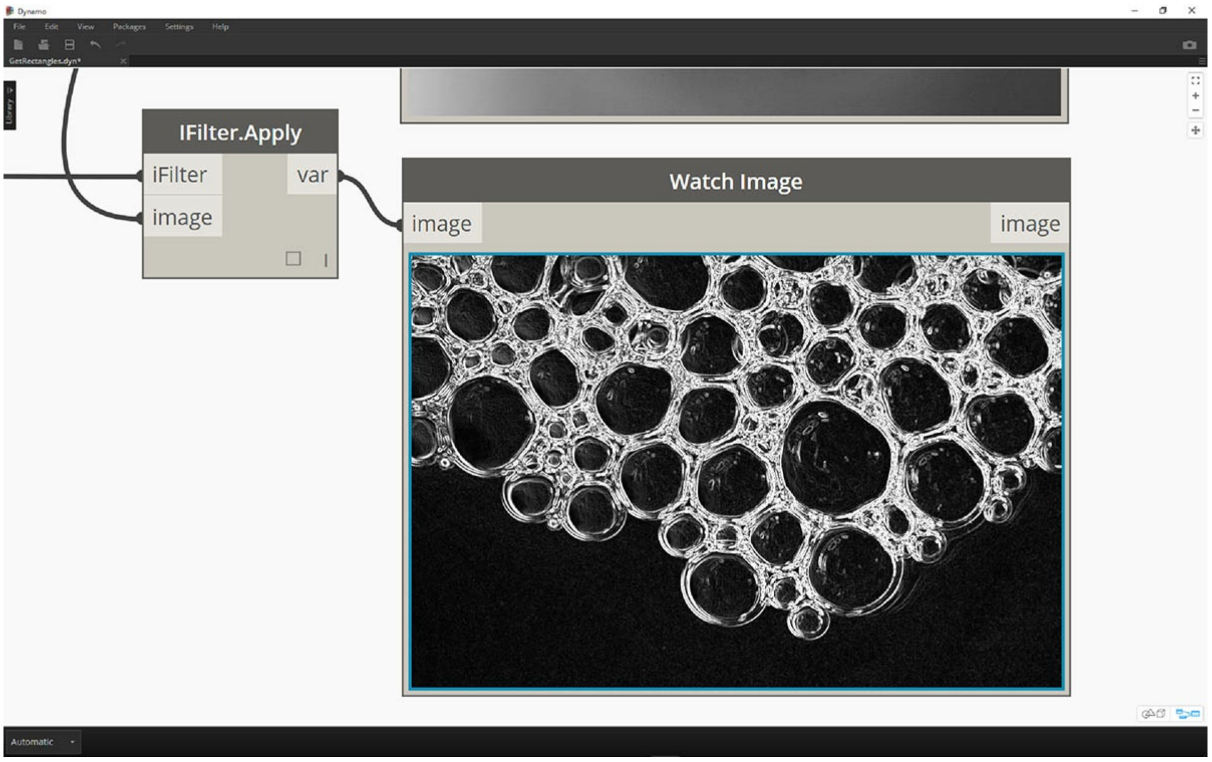
1. Change the top and bottom sliders to have a value of 0 and the middle slider to have a value of 1. This way we get a more legible desaturated

image.



Let's use the desaturated image, and apply another filter on top of it. The desaturated image has some contrast, so we we're going to test some edge detection.

1. Add a SobelEdgeDetector.SobelEdgeDetector node to the canvas. Connect this as the IFilter to a new IFilter node, and connect the desaturated image to the image input of the IFilter node.
2. The Sobel Edge Detector has highlighted the edges in a new image.

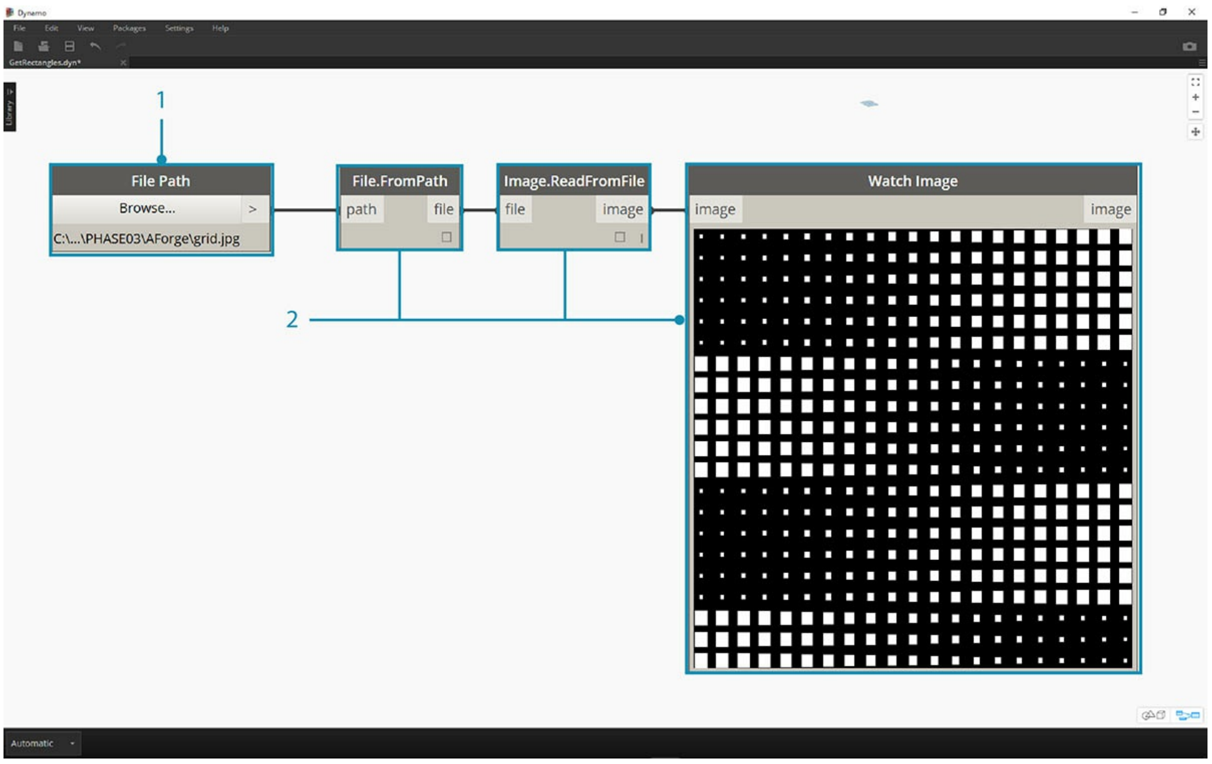


Zooming in, the edge detector has called out the outlines of the bubbles with pixels. The AForge library has tools to take results like this and create Dynamo geometry. We'll explore that in the next exercise.

Exercise 2 - Rectangle Creation

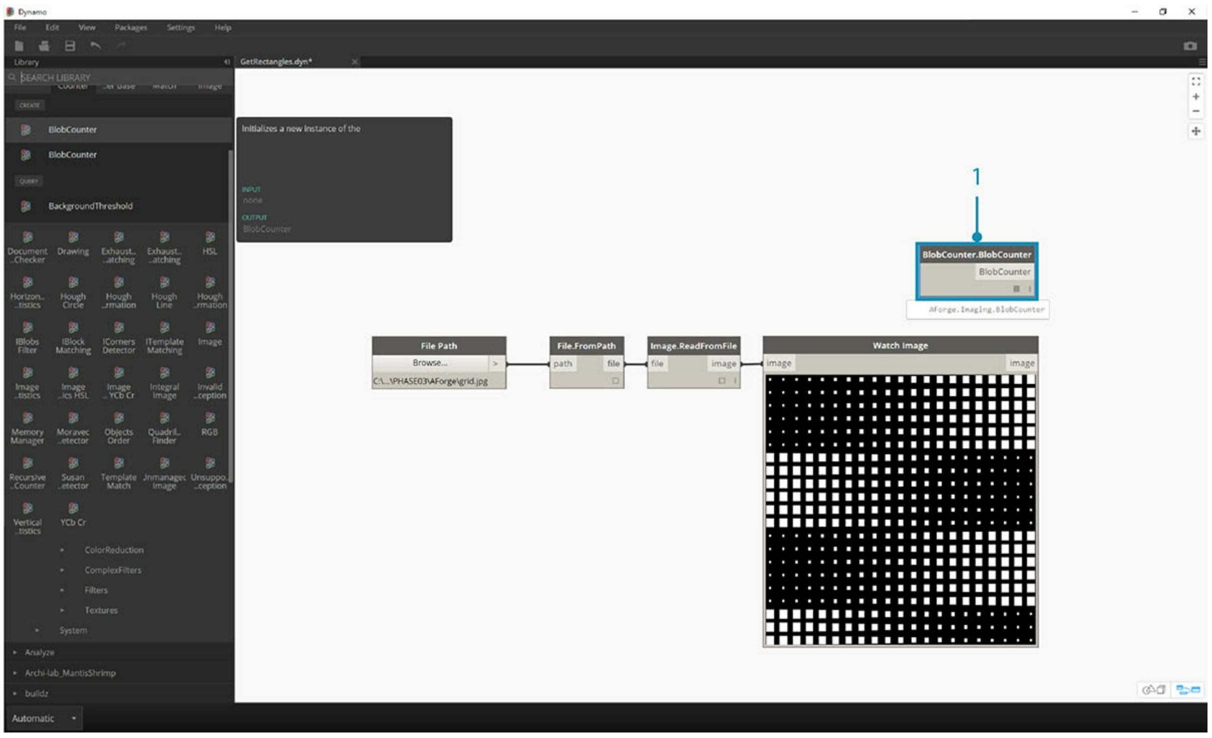
Download and unzip the example files that accompany this package case study (Right click and "Save Link As..."). A full list of example files can be found in the Appendix. ZeroTouchImages.zip

Now that we're introduced to some basic image processing, let's use an image to drive Dynamo geometry! On an elementary level, in this exercise we're aiming to do a *"Live Trace"* of an image using AForge and Dynamo. We're going to keep it simple and extract rectangles from a reference image, but there are tools available in AForge for more complex operations. We'll be working with *02-RectangleCreation.dyn* from the downloaded exercise files.

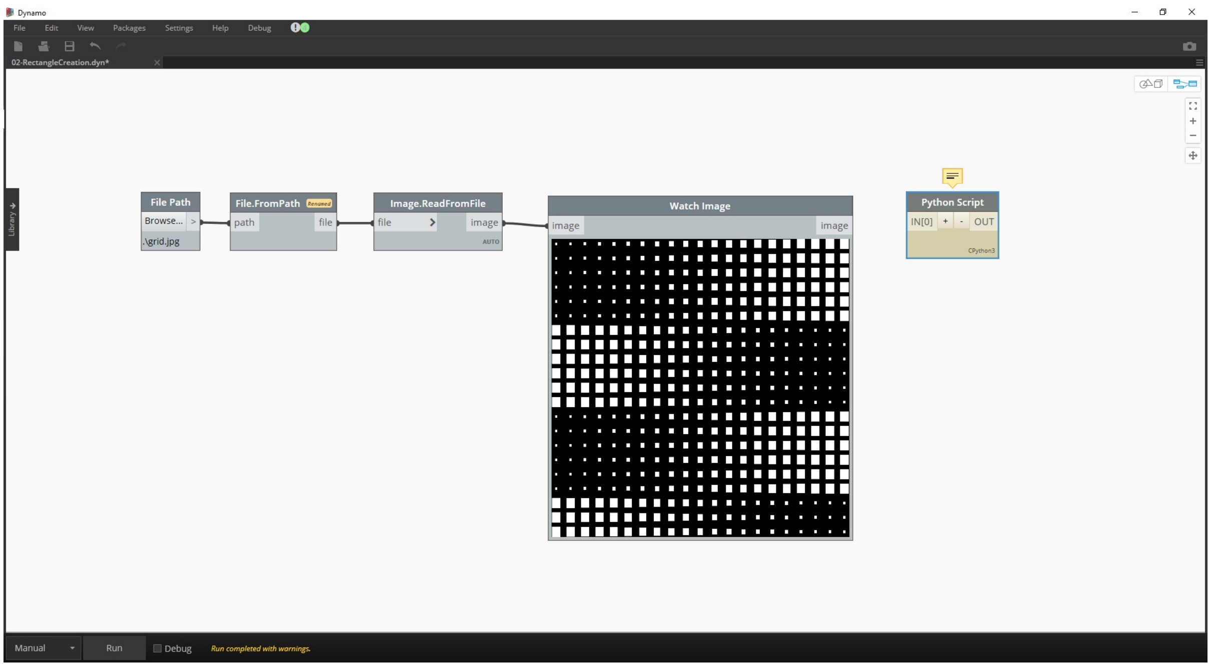


1. With the File Path node, navigate to grid.jpg in the exercise folder.
2. Connect the remaining series of nodes above to reveal a course parametric grid.

In this next step, we want to reference the white squares in the image and convert them to actual Dynamo geometry. AForge has a lot of powerful Computer Vision tools, and here we're going to use a particularly important one for the library called BlobCounter.



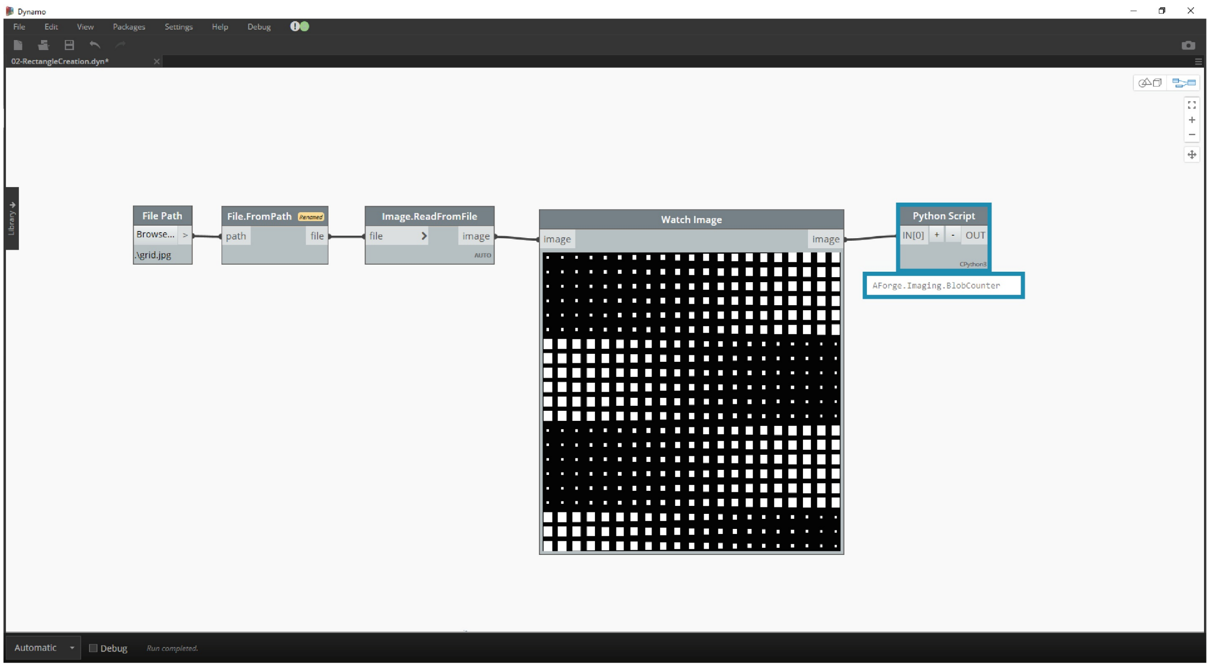
1. After adding a BlobCounter to the canvas, we need a way to process the image (similar to the IFilter tool in the previous exercise). Unfortunately the "Process Image" node is not immediately visible in the Dynamo library. This is because the function may not be visible in the AForge source code. In order to fix this, we'll need to find a work-around.



1. Add a Python node to the canvas.

```
import clr
clr.AddReference('AForge.Imaging')
from AForge.Imaging import *

bc= BlobCounter()
bc.ProcessImage(IN[0])
OUT=bc
```

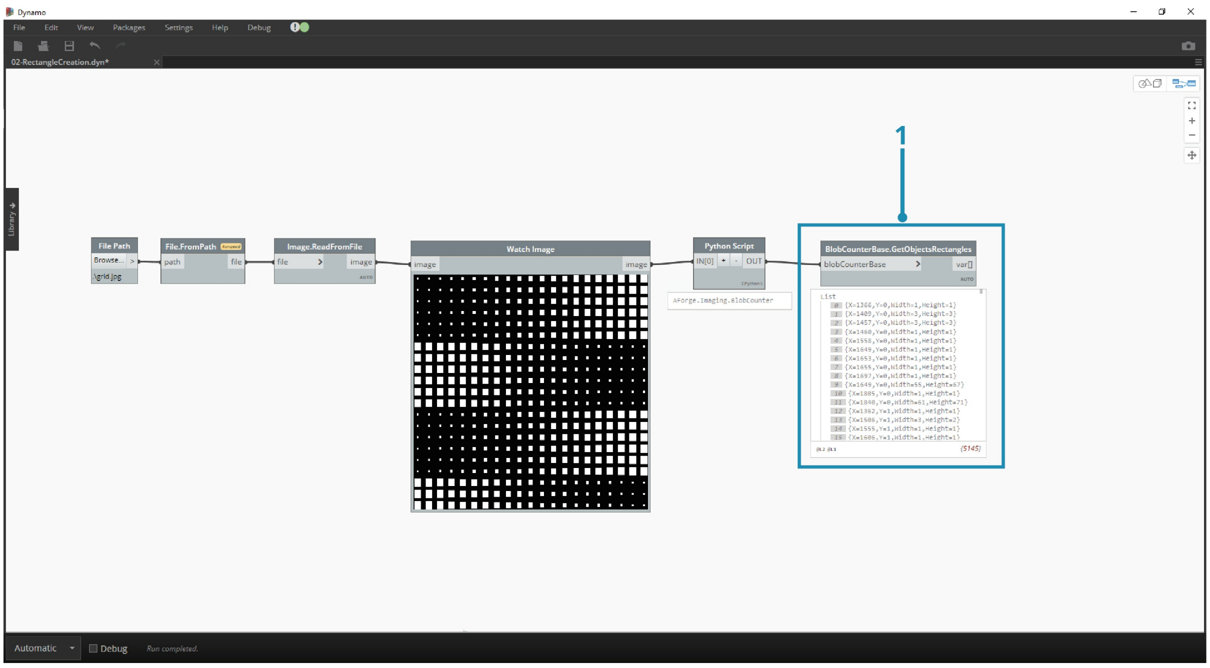
Add the code above to the Python node. This code imports the AForge library and then processes the imported image.



```
AForge.Imaging.BlobCounter
```

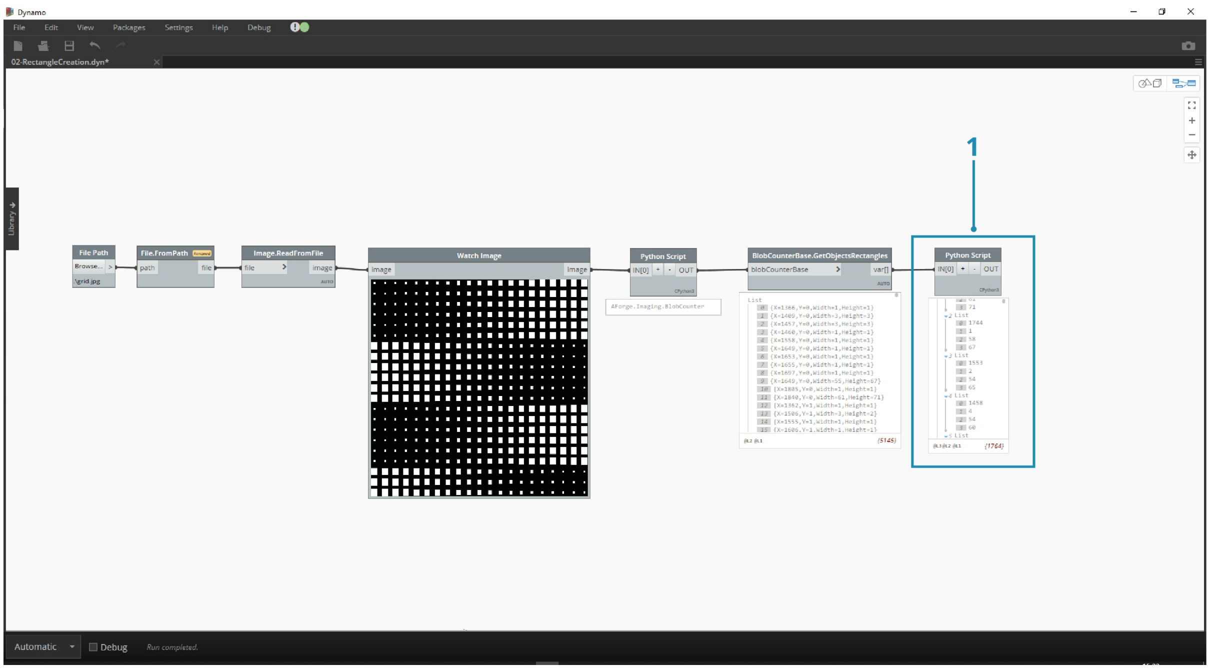
Connecting the image output to the Python node input, we get an AForge.Imaging.BlobCounter result from the Python node.

The next steps will do some tricks that demonstrate familiarity with the AForge Imaging API. It's not necessary to learn all of this for Dynamo work. This is more of a demonstration of working with external libraries within the flexibility of the Dynamo environment.



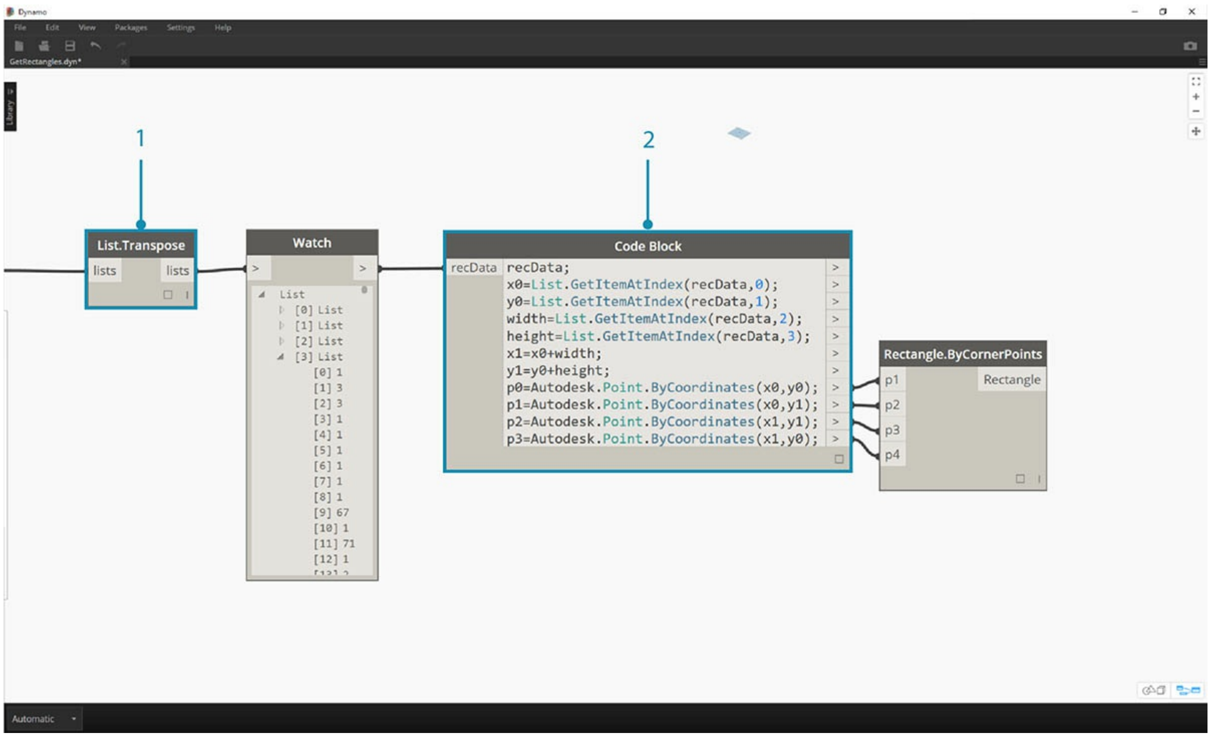
```
List
  0 {X=1366,Y=0,Width=1,Height=1}
  1 {X=1409,Y=0,Width=3,Height=3}
  2 {X=1457,Y=0,Width=3,Height=3}
  3 {X=1460,Y=0,Width=1,Height=1}
  4 {X=1558,Y=0,Width=1,Height=1}
  5 {X=1649,Y=0,Width=1,Height=1}
  6 {X=1653,Y=0,Width=1,Height=1}
  7 {X=1655,Y=0,Width=1,Height=1}
  8 {X=1697,Y=0,Width=1,Height=1}
  9 {X=1649,Y=0,Width=55,Height=67}
  10 {X=1805,Y=0,Width=1,Height=1}
  11 {X=1840,Y=0,Width=61,Height=71}
  12 {X=1362,Y=1,Width=1,Height=1}
  13 {X=1506,Y=1,Width=3,Height=2}
  14 {X=1555,Y=1,Width=1,Height=1}
  15 {X=1606,Y=1,Width=1,Height=1}
@L2 @L1                              {5145}
```

1. Connect the output of the Python script to BlobCounterBase.GetObjectRectangles. This reads objects in an image, based on a threshold value, and extracts quantified rectangles from the pixel space.



1. Adding another Python node to the canvas, connect to the GetObjectRectangles, and input the code below. This will create an organized list of Dynamo objects.

```
OUT = []
for rec in IN[0]:
    subOUT=[]
    subOUT.append(rec.X)
    subOUT.append(rec.Y)
    subOUT.append(rec.Width)
    subOUT.append(rec.Height)
    OUT.append(subOUT)
```



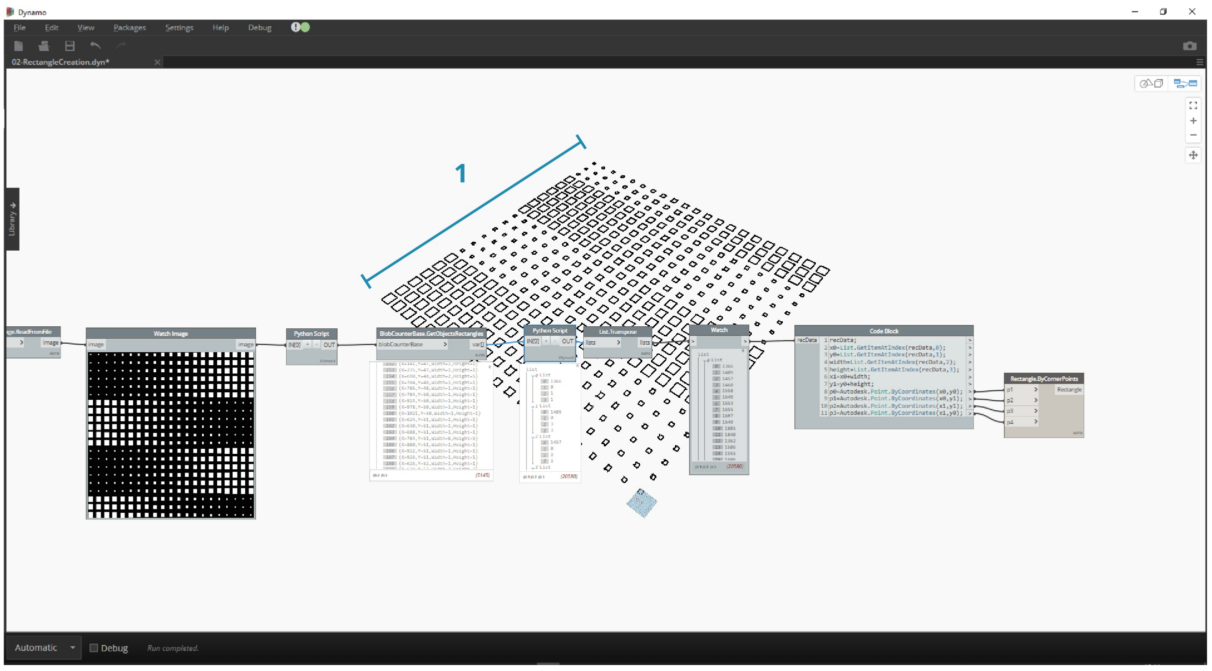
```
List
  [0] List
  [1] List
  [2] List
  [3] List
    [0] 1
    [1] 3
    [2] 3
    [3] 1
    [4] 1
    [5] 1
    [6] 1
    [7] 1
    [8] 1
    [9] 67
    [10] 1
    [11] 71
    [12] 1
```

```
recData;
x0=List.GetItemAtIndex(recData,0);
y0=List.GetItemAtIndex(recData,1);
width=List.GetItemAtIndex(recData,2);
height=List.GetItemAtIndex(recData,3);
x1=x0+width;
y1=y0+height;
p0=Autodesk.Point.ByCoordinates(x0,y0);
p1=Autodesk.Point.ByCoordinates(x0,y1);
p2=Autodesk.Point.ByCoordinates(x1,y1);
p3=Autodesk.Point.ByCoordinates(x1,y0);
```

1. Transpose the output of the Python node from the previous step. This creates 4 lists, each representing X,Y, Width, and Height for each rectangle.
2. Using Code Block, we organize the data into a structure that accommodates the Rectangle.ByCornerPoints node (code below).

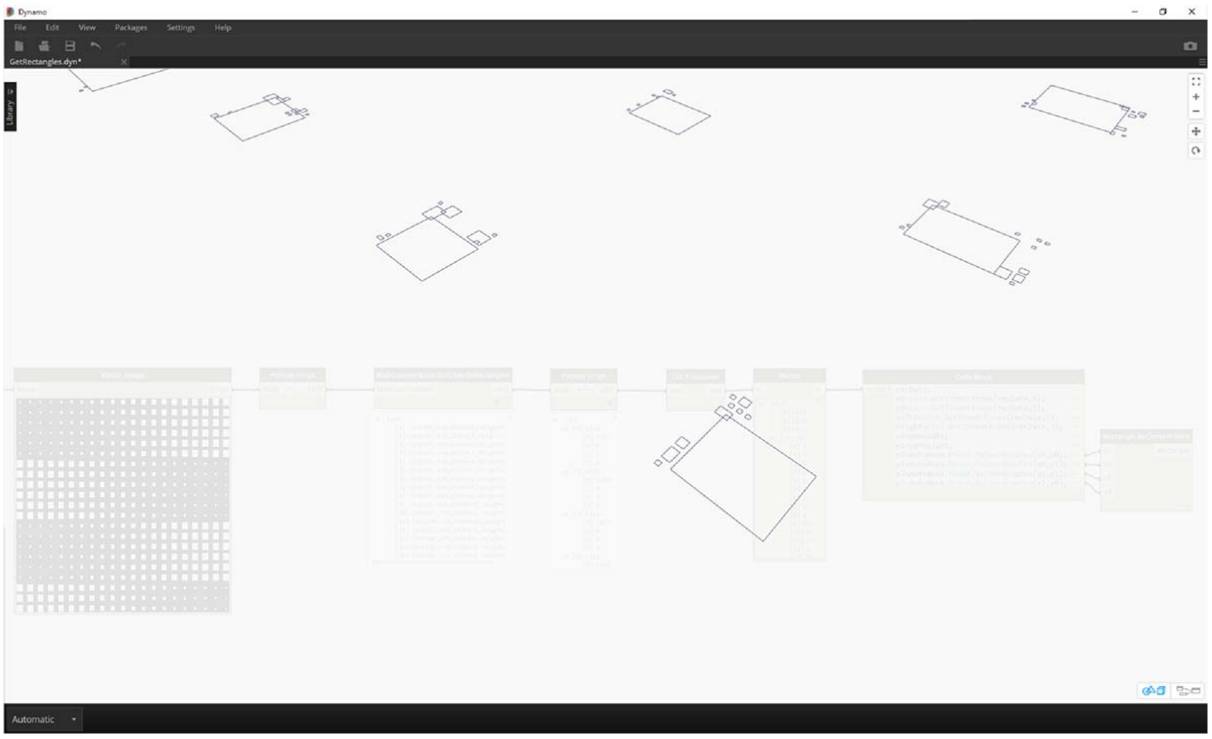
```
recData;
x0=List.GetItemAtIndex(recData,0);
y0=List.GetItemAtIndex(recData,1);
width=List.GetItemAtIndex(recData,2);
```

```
height=List.GetItemAtIndex(recData,3);
x1=x0+width;
y1=y0+height;
p0=Autodesk.Point.ByCoordinates(x0,y0);
p1=Autodesk.Point.ByCoordinates(x0,y1);
p2=Autodesk.Point.ByCoordinates(x1,y1);
p3=Autodesk.Point.ByCoordinates(x1,y0);
```

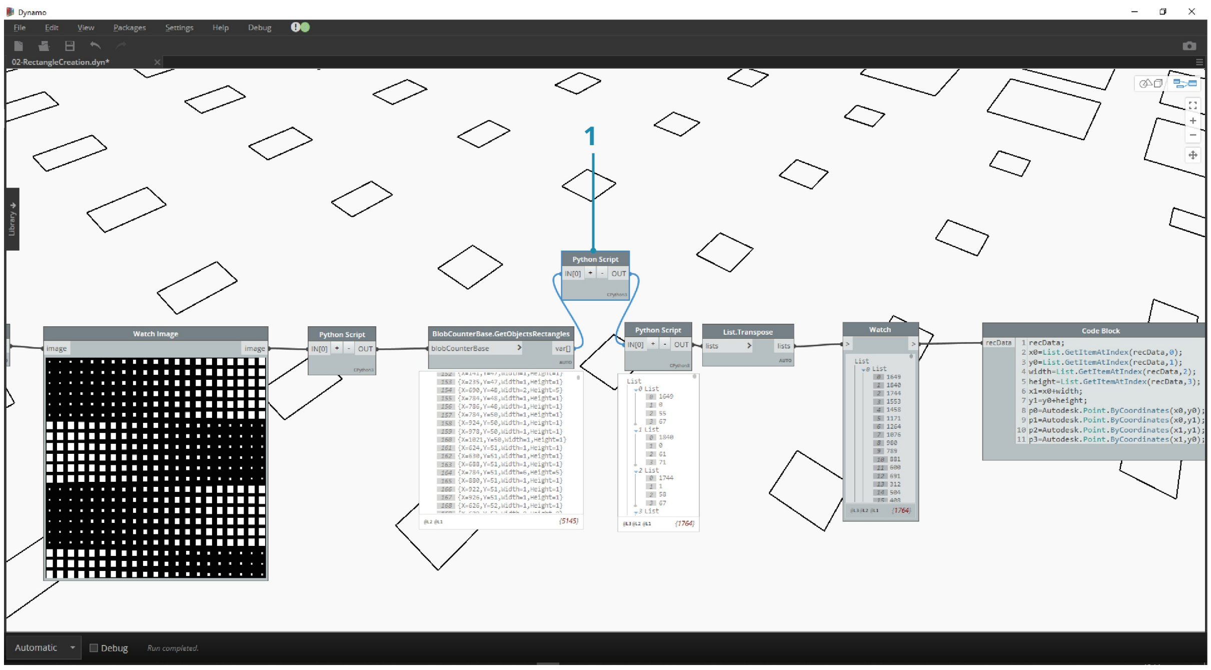


```
recData;
x0=List.GetItemAtIndex(recData,0);
y0=List.GetItemAtIndex(recData,1);
width=List.GetItemAtIndex(recData,2);
height=List.GetItemAtIndex(recData,3);
x1=x0+width;
y1=y0+height;
p0=Autodesk.Point.ByCoordinates(x0,y0);
p1=Autodesk.Point.ByCoordinates(x0,y1);
p2=Autodesk.Point.ByCoordinates(x1,y1);
p3=Autodesk.Point.ByCoordinates(x1,y0);
```

Zooming out, we have an array of rectangles representing the white squares in the image. Through programming, we've done something (roughly) similar to a live trace in Illustrator!



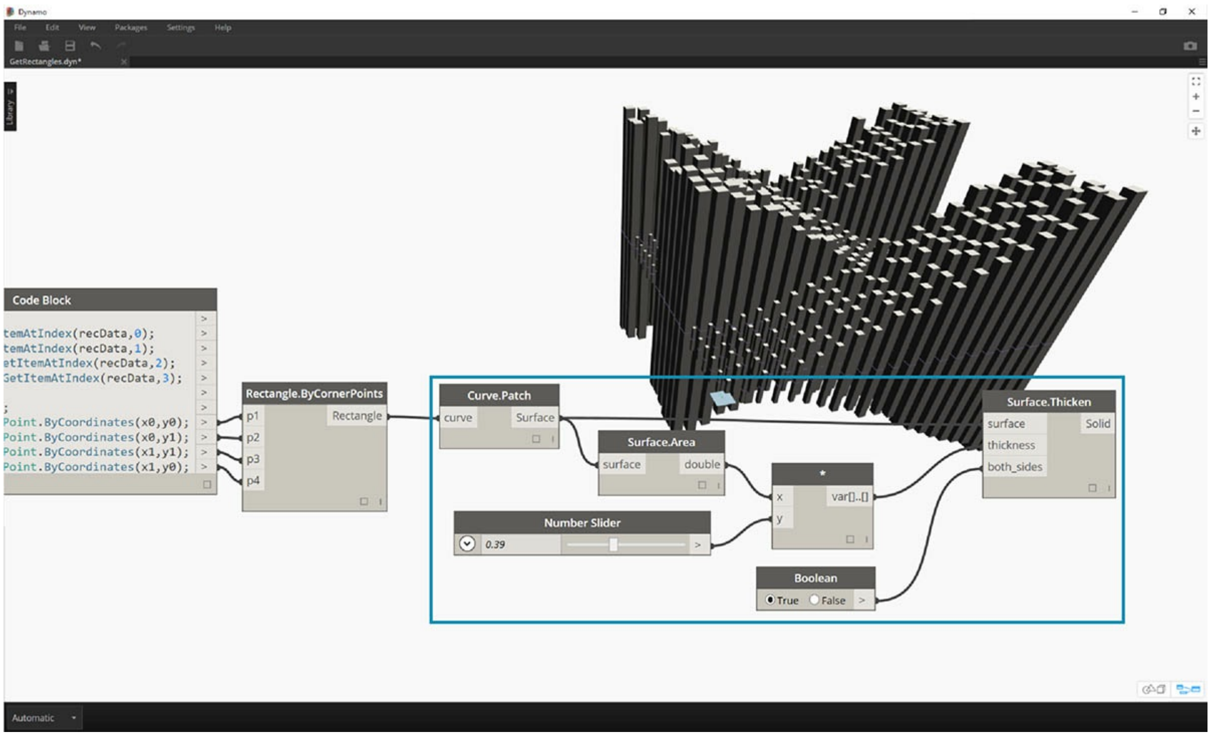
We still need some cleanup, however. Zooming in, we can see that we have a bunch of small, unwanted rectangles.



```
recData;
x0=List.GetItemAtIndex(recData,0);
y0=List.GetItemAtIndex(recData,1);
width=List.GetItemAtIndex(recData,2);
height=List.GetItemAtIndex(recData,3);
x1=x0+width;
y1=y0+height;
p0=Autodesk.Point.ByCoordinates(x0,y0);
p1=Autodesk.Point.ByCoordinates(x0,y1);
p2=Autodesk.Point.ByCoordinates(x1,y1);
p3=Autodesk.Point.ByCoordinates(x1,y0);
```

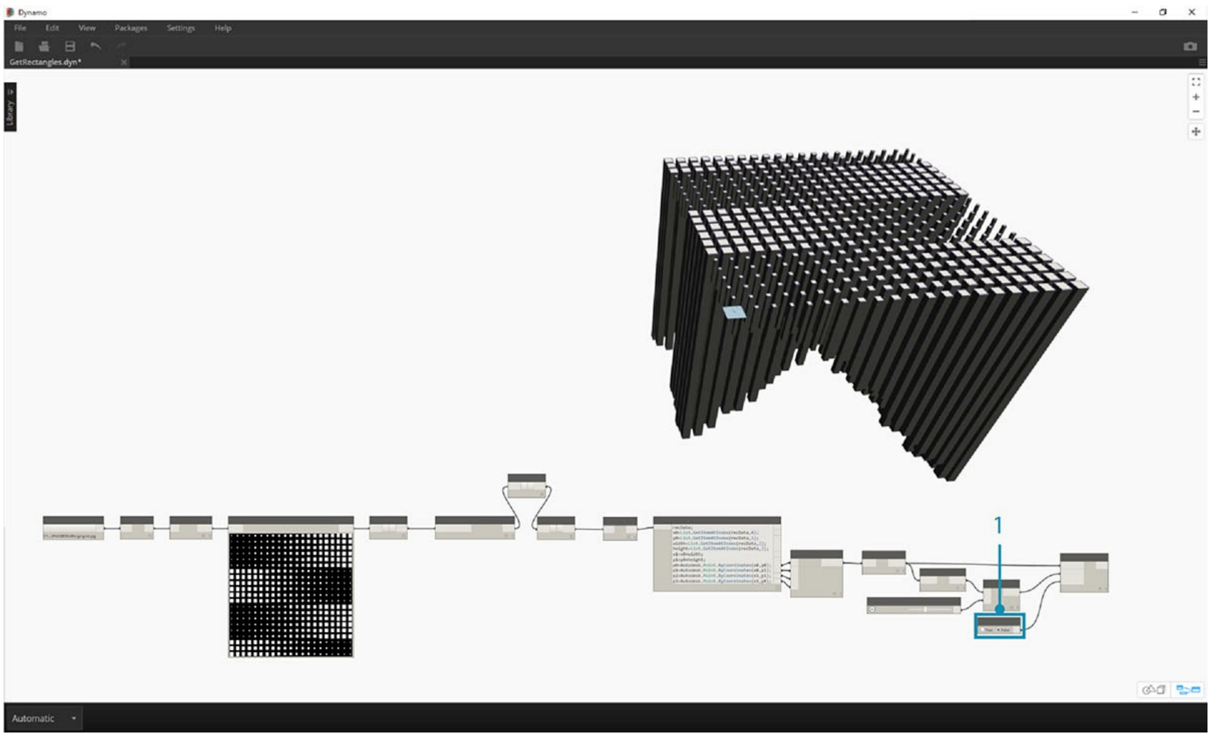
1. We get rid of the unwanted rectangles by inserting a Python node in between the GetObjectRectangles node and another Python node. The node's code is below, and removes all rectangles which are below a given size.

```
rectangles=IN[0]
OUT=[]
for rec in rectangles:
 if rec.Width>8 and rec.Height>8:
  OUT.append(rec)
```



```
temAtIndex(recData,0);
temAtIndex(recData,1);
etItemAtIndex(recData,2);
GetItemAtIndex(recData,3);

;
Point.ByCoordinates(x0,y0);
Point.ByCoordinates(x0,y1);
Point.ByCoordinates(x1,y1);
Point.ByCoordinates(x1,y0);
```

With the superfluous rectangles gone, just for kicks, let's create a surface from these rectangles and extrude them by a distance based on their areas.



1. Last, change the both_sides input to false and we get an extrusion in one direction. Dip this baby in resin and you've got yourself one super nerdy table.

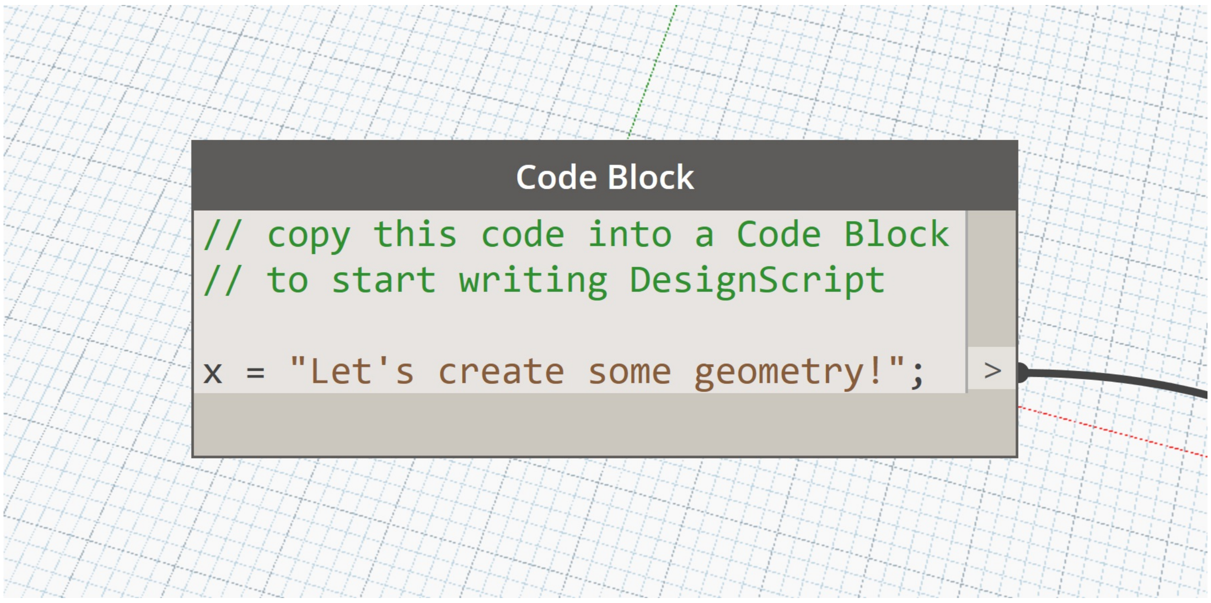
These are basic examples, but the concepts outlined here are transferable to exciting real-world applications. Computer vision can be used for a whole host of processes. To name a few: barcode readers, perspective matching, projection mapping, and augmented reality. For more advanced topics with AForge related to this exercise, have a read through this article.

Geometry with DesignScript

Geometry with DesignScript

In this section, you will find a series of lessons on the creation of geometry with DesignScript. Follow along by copying the example DesignScript into Dynamo Code Blocks.

```
// copy this code into a Code Block
// to start writing DesignScript

x = "Let's create some geometry!";
```

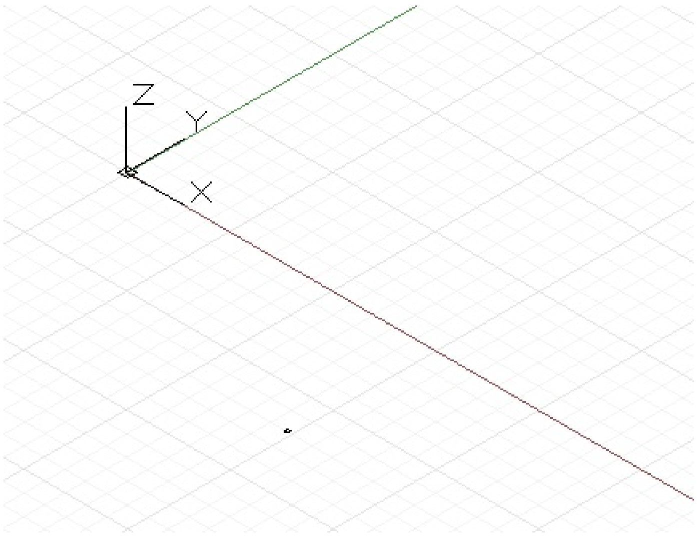


```
// copy this code into a Code Block
// to start writing DesignScript

x = "Let's create some geometry!";
```

DesignScript Geometry Basics

DesignScript Geometry Basics

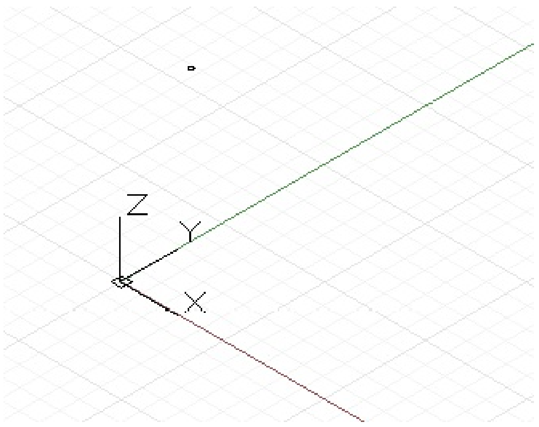
The simplest geometrical object in the Dynamo standard geometry library is a point. All geometry is created using special functions called constructors, which each return a new instance of that particular geometry type. In Dynamo, constructors begin with the name of the object's type, in this case Point, followed by the method of construction. To create a three dimensional point specified by x, y, and z Cartesian coordinates, use the *ByCoordinates* constructor:



```
// create a point with the following x, y, and z
// coordinates:
x = 10;
y = 2.5;
z = -6;

p = Point.ByCoordinates(x, y, z);
```

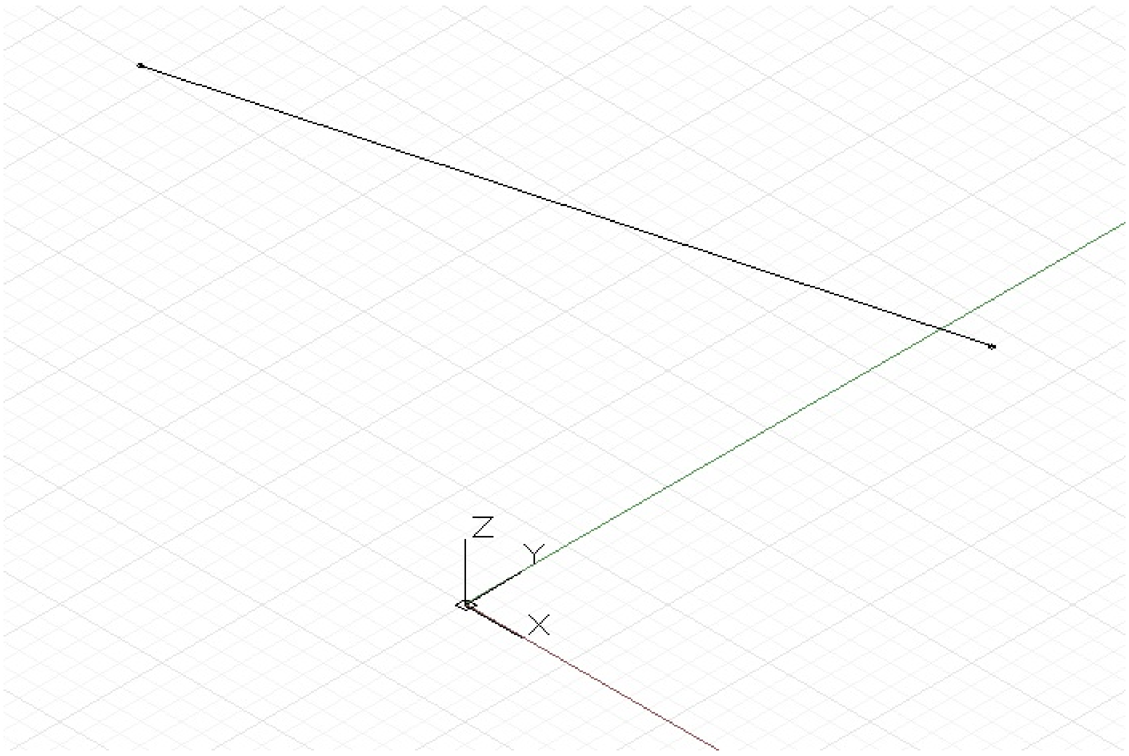
Constructors in Dynamo are typically designated with the "*By*" prefix, and invoking these functions returns a newly created object of that type. This newly created object is stored in the variable named on the left side of the equal sign.

Most objects have many different constructors, and we can use the *BySphericalCoordinates* constructor to create a point lying on a sphere, specified by the sphere's radius, a first rotation angle, and a second rotation angle (specified in degrees):



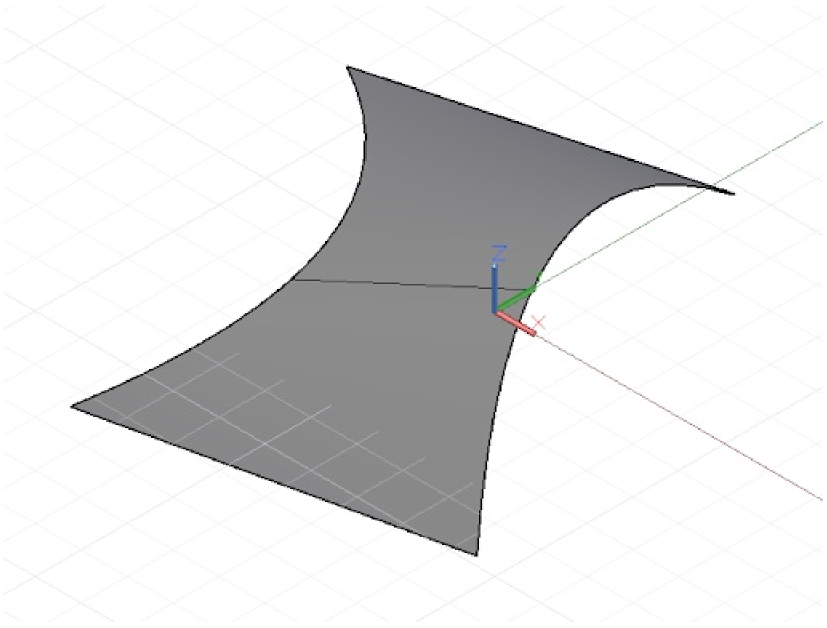
```
// create a point on a sphere with the following radius,
// theta, and phi rotation angles (specified in degrees)
radius = 5;
theta = 75.5;
phi = 120.3;
cs = CoordinateSystem.Identity();

p = Point.BySphericalCoordinates(cs, radius, theta,
    phi);
```

Points can be used to construct higher dimensional geometry such as lines. We can use the *ByStartPointEndPoint* constructor to create a Line object between two points:



```
// create two points:
p1 = Point.ByCoordinates(3, 10, 2);
p2 = Point.ByCoordinates(-15, 7, 0.5);

// construct a line between p1 and p2
l = Line.ByStartPointEndPoint(p1, p2);
```

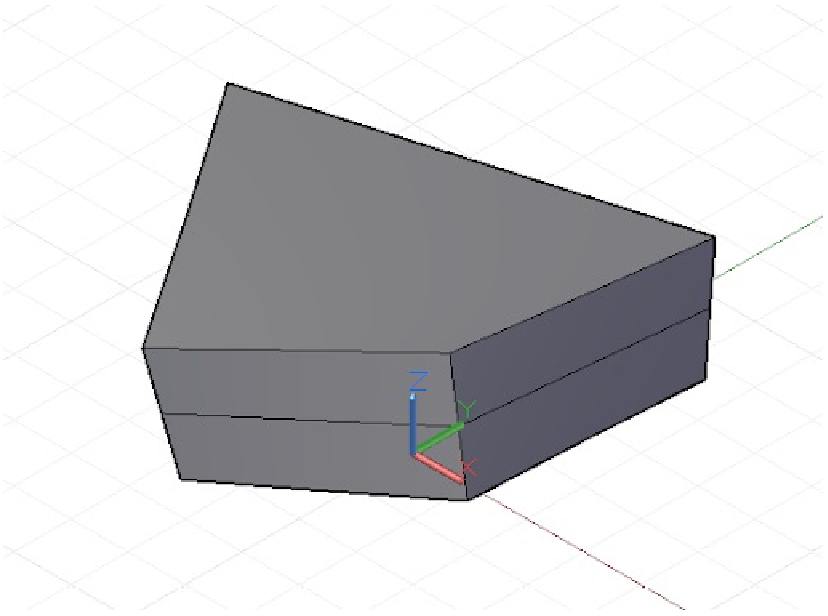
Similarly, lines can be used to create higher dimensional surface geometry, for instance using the *Loft* constructor, which takes a series of lines or curves and interpolates a surface between them.



```
// create points:
p1 = Point.ByCoordinates(3, 10, 2);
p2 = Point.ByCoordinates(-15, 7, 0.5);

p3 = Point.ByCoordinates(5, -3, 5);
p4 = Point.ByCoordinates(-5, -6, 2);

p5 = Point.ByCoordinates(9, -10, -2);
p6 = Point.ByCoordinates(-11, -12, -4);

// create lines:
```

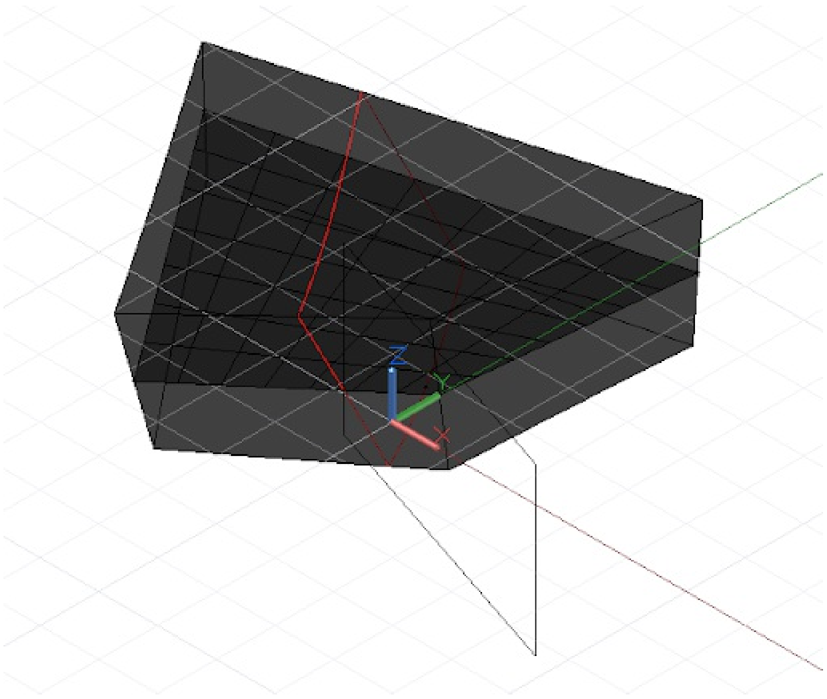
```
l1 = Line.ByStartPointEndPoint(p1, p2);
l2 = Line.ByStartPointEndPoint(p3, p4);
l3 = Line.ByStartPointEndPoint(p5, p6);

// loft between cross section lines:
surf = Surface.ByLoft([l1, l2, l3]);
```

Surfaces too can be used to create higher dimensional solid geometry, for instance by thickening the surface by a specified distance. Many objects have functions attached to them, called methods, allowing the programmer to perform commands on that particular object. Methods common to all pieces of geometry include *Translate* and *Rotate*, which respectively translate (move) and rotate the geometry by a specified amount. Surfaces have a *Thicken* method, which take a single input, a number specifying the new thickness of the surface.



```
p1 = Point.ByCoordinates(3, 10, 2);
p2 = Point.ByCoordinates(-15, 7, 0.5);

p3 = Point.ByCoordinates(5, -3, 5);
p4 = Point.ByCoordinates(-5, -6, 2);

l1 = Line.ByStartPointEndPoint(p1, p2);
l2 = Line.ByStartPointEndPoint(p3, p4);

surf = Surface.ByLoft([l1, l2]);

// true indicates to thicken both sides of the Surface:
solid = surf.Thicken(4.75, true);
```

Intersection commands can extract lower dimensional geometry from higher dimensional objects. This extracted lower dimensional geometry can form the basis for higher dimensional geometry, in a cyclic process of geometrical creation, extraction, and recreation. In this example, we use the generated Solid to create a Surface, and use the Surface to create a Curve.



```
p1 = Point.ByCoordinates(3, 10, 2);
p2 = Point.ByCoordinates(-15, 7, 0.5);

p3 = Point.ByCoordinates(5, -3, 5);
p4 = Point.ByCoordinates(-5, -6, 2);

l1 = Line.ByStartPointEndPoint(p1, p2);
l2 = Line.ByStartPointEndPoint(p3, p4);

surf = Surface.ByLoft([l1, l2]);

solid = surf.Thicken(4.75, true);

p = Plane.ByOriginNormal(Point.ByCoordinates(2, 0, 0),
    Vector.ByCoordinates(1, 1, 1));

int_surf = solid.Intersect(p);

int_line = int_surf.Intersect(Plane.ByOriginNormal(
    Point.ByCoordinates(0, 0, 0),
    Vector.ByCoordinates(1, 0, 0)));
```

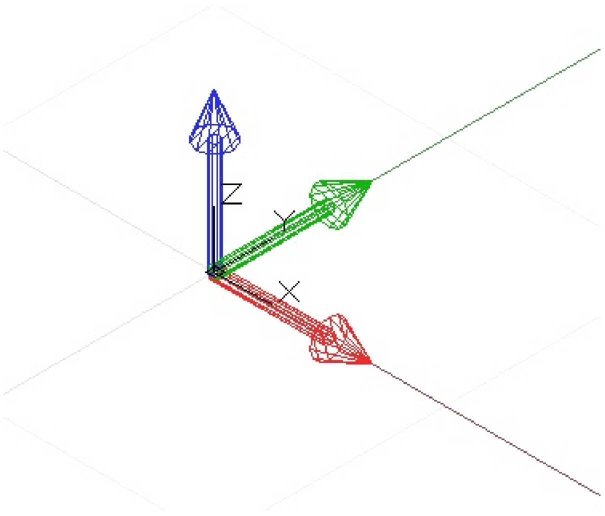
Geometric Primitives

Geometric Primitives

While Dynamo is capable of creating a variety of complex geometric forms, simple geometric primitives form the backbone of any computational design: either directly expressed in the final designed form, or used as scaffolding off of which more complex geometry is generated.

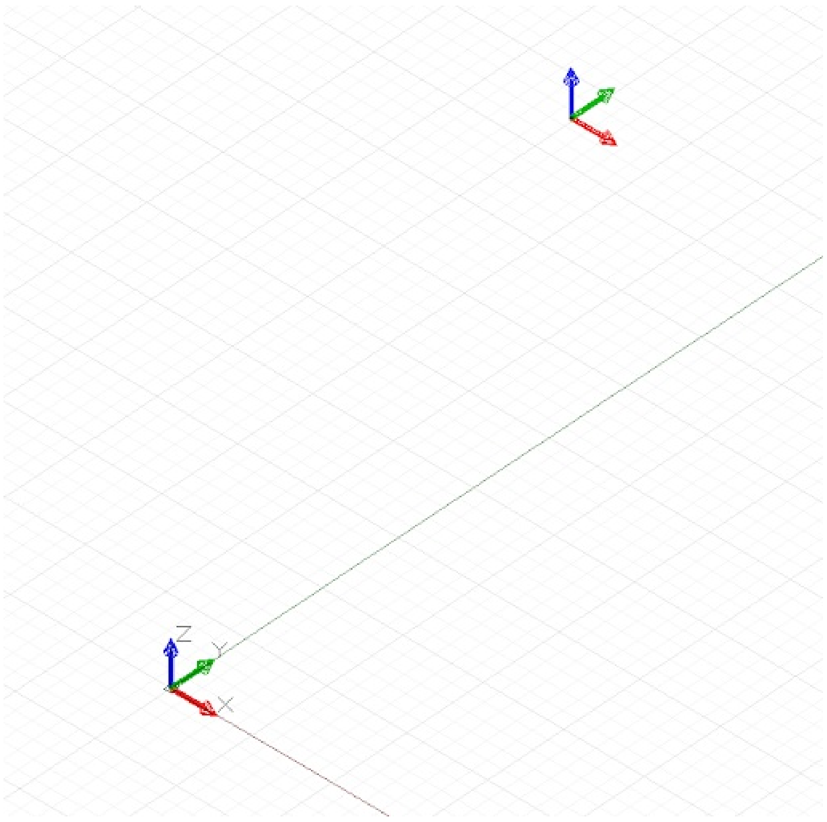
While not strictly a piece of geometry, the CoordinateSystem is an important tool for constructing geometry. A CoordinateSystem object keeps track of both position and geometric transformations such as rotation, sheer, and scaling.

Creating a CoordinateSystem centered at a point with x = 0, y = 0, z = 0, with no rotations, scaling, or sheering transformations, simply requires calling the Identity constructor:



```
// create a CoordinateSystem at x = 0, y = 0, z = 0,
// no rotations, scaling, or sheering transformations

cs = CoordinateSystem.Identity();
```

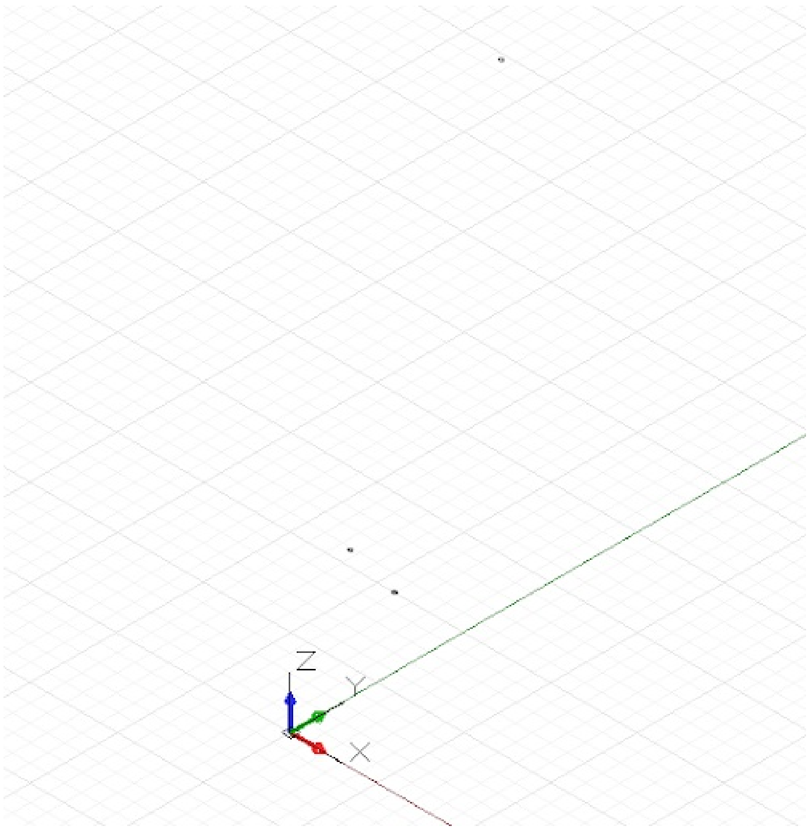
CoordinateSystems with geometric transformations are beyond the scope of this chapter, though another constructor allows you to create a coordinate system at a specific point, *CoordinateSystem.ByOriginVectors*:



```
// create a CoordinateSystem at a specific location,
// no rotations, scaling, or sheering transformations
x_pos = 3.6;
y_pos = 9.4;
z_pos = 13.0;

origin = Point.ByCoordinates(x_pos, y_pos, z_pos);
identity = CoordinateSystem.Identity();

cs = CoordinateSystem.ByOriginVectors(origin,
    identity.XAxis, identity.YAxis, identity.ZAxis);
```

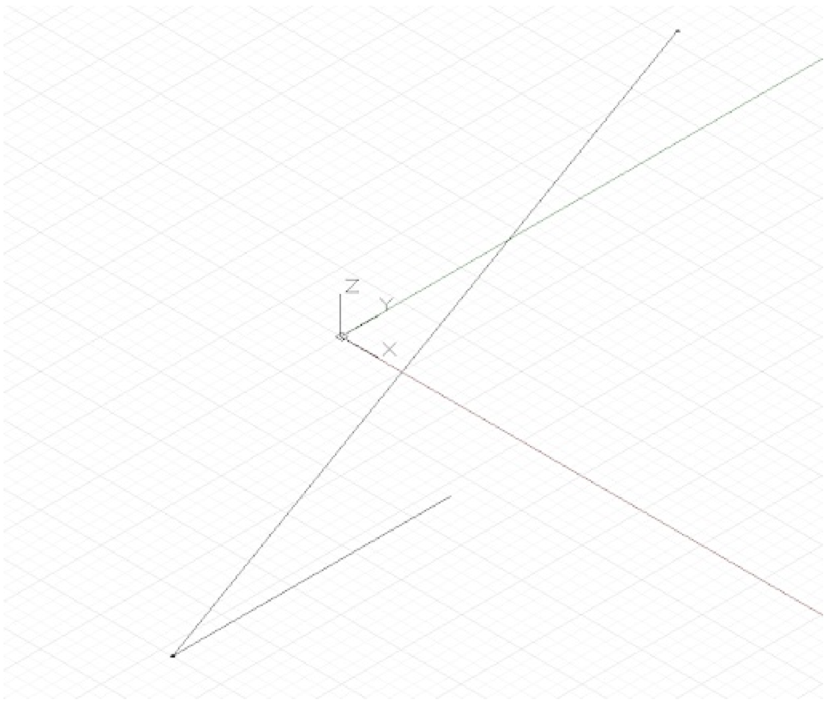
The simplest geometric primitive is a Point, representing a zero-dimensional location in three-dimensional space. As mentioned earlier there are several different ways to create a point in a particular coordinate system: *Point.ByCoordinates* creates a point with specified x, y, and z coordinates; *Point.ByCartesianCoordinates* creates a point with a specified x, y, and z coordinates in a specific coordinate system; *Point.ByCylindricalCoordinates* creates a point lying on a cylinder with radius, rotation angle, and height; and *Point.BySphericalCoordinates* creates a point lying on a sphere with radius and two rotation angle.

This example shows points created at various coordinate systems:



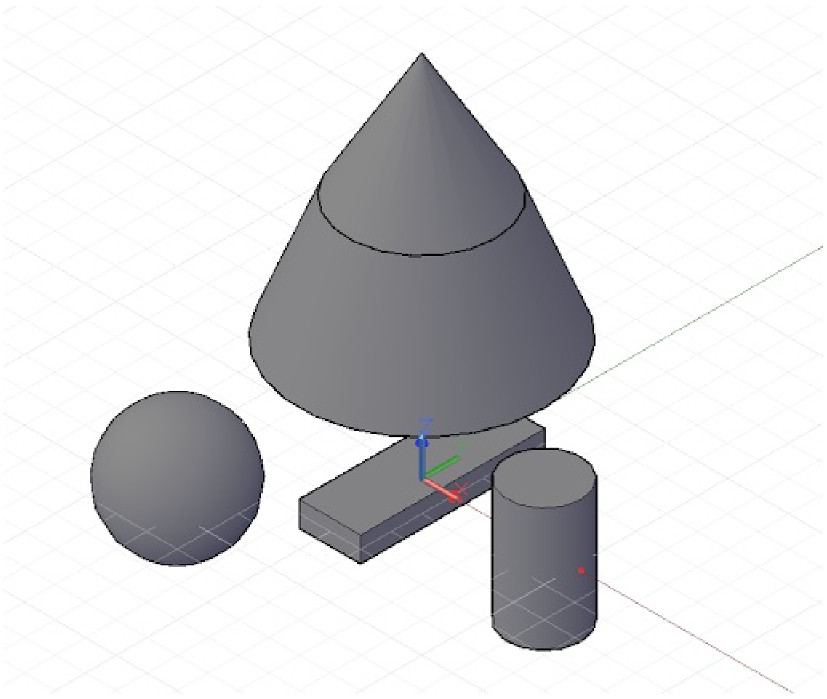
```
// create a point with x, y, and z coordinates
x_pos = 1;
y_pos = 2;
z_pos = 3;

pCoord = Point.ByCoordinates(x_pos, y_pos, z_pos);

// create a point in a specific coordinate system
cs = CoordinateSystem.Identity();
pCoordSystem = Point.ByCartesianCoordinates(cs, x_pos,
    y_pos, z_pos);

// create a point on a cylinder with the following
// radius and height
radius = 5;
height = 15;
theta = 75.5;

pCyl = Point.ByCylindricalCoordinates(cs, radius, theta,
    height);

// create a point on a sphere with radius and two angles

phi = 120.3;

pSphere = Point.BySphericalCoordinates(cs, radius,
    theta, phi);
```

The next higher dimensional Dynamo primitive is a line segment, representing an infinite number of points between two end points. Lines can be created by explicitly stating the two boundary points with the constructor *Line.ByStartPointEndPoint*, or by specifying a start point, direction, and length in that direction, *Line.ByStartPointDirectionLength*.



```
p1 = Point.ByCoordinates(-2, -5, -10);
p2 = Point.ByCoordinates(6, 8, 10);

// a line segment between two points
l2pts = Line.ByStartPointEndPoint(p1, p2);

// a line segment at p1 in direction 1, 1, 1 with
// length 10
lDir = Line.ByStartPointDirectionLength(p1,
    Vector.ByCoordinates(1, 1, 1), 10);
```

Dynamo has objects representing the most basic types of geometric primitives in three dimensions: Cuboids, created with *Cuboid.ByLengths*; Cones, created with *Cone.ByPointsRadius* and *Cone.ByPointsRadii*; Cylinders, created with *Cylinder.ByRadiusHeight*; and Spheres, created with *Sphere.ByCenterPointRadius*.



```
// create a cuboid with specified lengths
cs = CoordinateSystem.Identity();

cub = Cuboid.ByLengths(cs, 5, 15, 2);
```

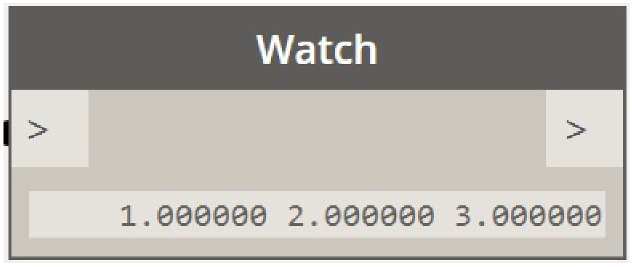
```
// create several cones
p1 = Point.ByCoordinates(0, 0, 10);
p2 = Point.ByCoordinates(0, 0, 20);
p3 = Point.ByCoordinates(0, 0, 30);

cone1 = Cone.ByPointsRadii(p1, p2, 10, 6);
cone2 = Cone.ByPointsRadii(p2, p3, 6, 0);

// make a cylinder
cylCS = cs.Translate(10, 0, 0);

cyl = Cylinder.ByRadiusHeight(cylCS, 3, 10);

// make a sphere
centerP = Point.ByCoordinates(-10, -10, 0);

sph = Sphere.ByCenterPointRadius(centerP, 5);
```

Vector Math

Vector Math

Objects in computational designs are rarely created explicitly in their final position and form, and are most often translated,rotated, and otherwise positioned based off of existing geometry. Vector math serves as a kind-of geometric scaffolding to give direction and orientation to geometry, as well as to conceptualize movements through 3D space without visual representation.

At its most basic, a vector represents a position in 3D space, and is often times thought of as the endpoint of an arrow from the position (0, 0, 0) to that position. Vectors can be created with the *ByCoordinates* constructor, taking the x, y, and z position of the newly created Vector object. Note that Vector objects are not geometric objects, and don’t appear in the Dynamo window. However, information about a newly created or modified vector can be printed in the console window:

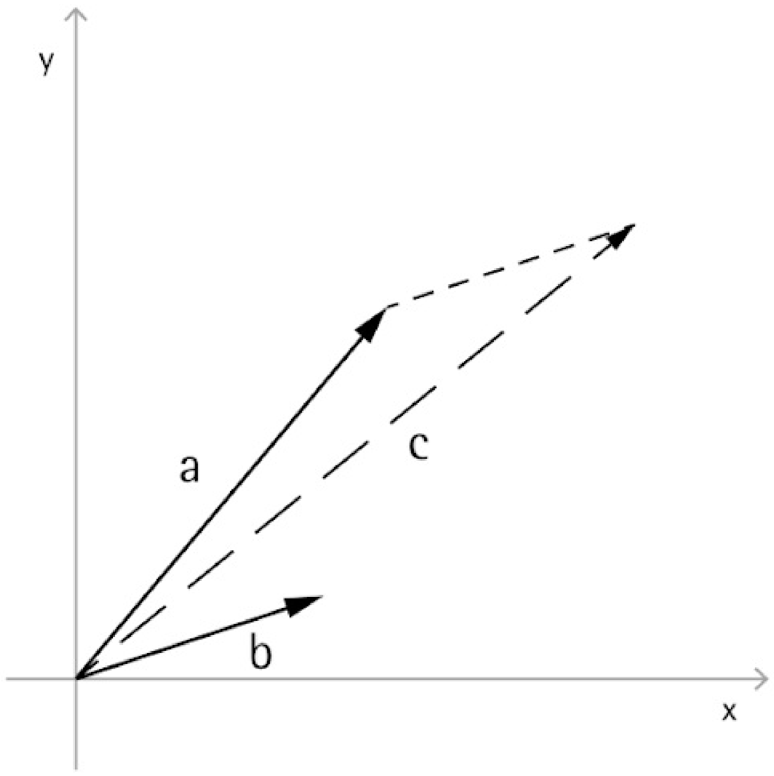


```
1.000000 2.000000 3.000000
```

```
// construct a Vector object
v = Vector.ByCoordinates(1, 2, 3);

s = v.X + " " + v.Y + " " + v.Z;
```

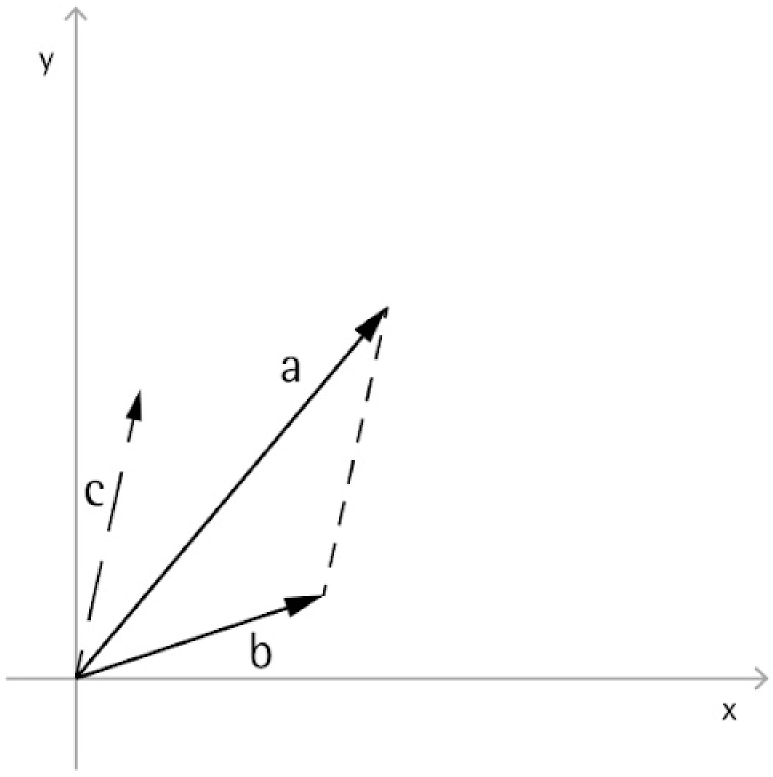
A set of mathematical operations are defined on Vector objects, allowing you to add, subtract, multiply, and otherwise move objects in 3D space as you would move real numbers in 1D space on a number line.

Vector addition is defined as the sum of the components of two vectors, and can be thought of as the resulting vector if the two component vector arrows are placed “tip to tail.” Vector addition is performed with the *Add* method, and is represented by the diagram on the left.



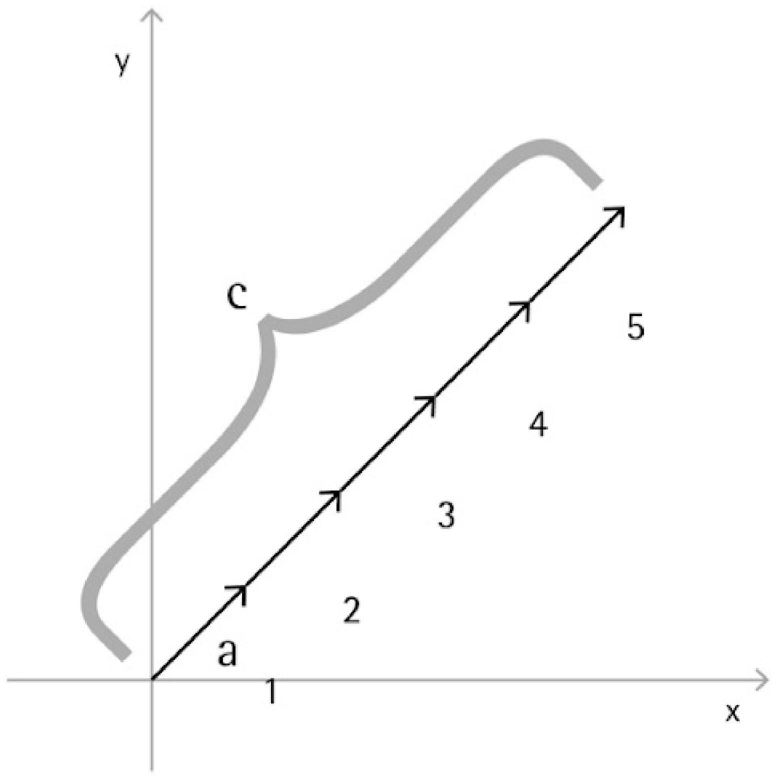
```
a = Vector.ByCoordinates(5, 5, 0);
b = Vector.ByCoordinates(4, 1, 0);

// c has value x = 9, y = 6, z = 0
c = a.Add(b);
```

Similarly, two Vector objects can be subtracted from each other with the *Subtract* method. Vector subtraction can be thought of as the direction from first vector to the second vector.



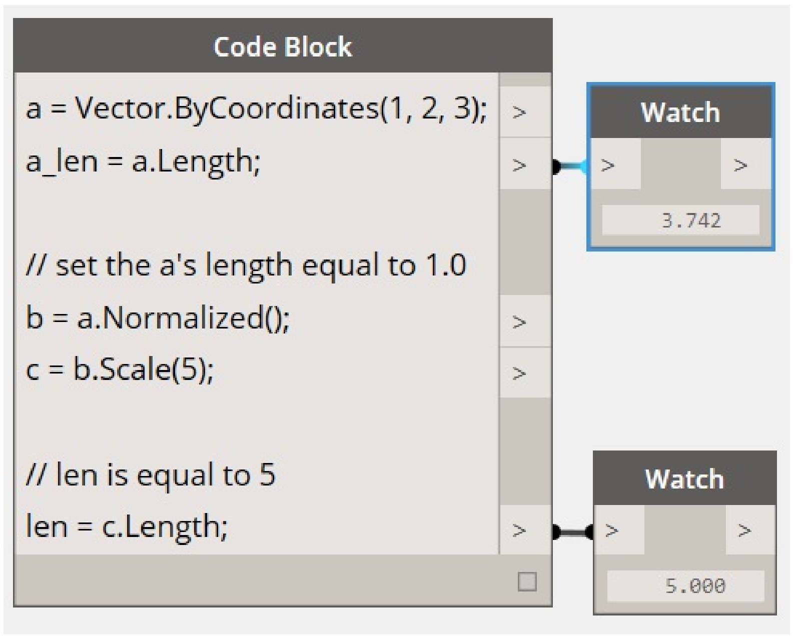
```
a = Vector.ByCoordinates(5, 5, 0);
b = Vector.ByCoordinates(4, 1, 0);

// c has value x = 1, y = 4, z = 0
c = a.Subtract(b);
```

Vector multiplication can be thought of as moving the endpoint of a vector in its own direction by a given scale factor.



```
a = Vector.ByCoordinates(4, 4, 0);

// c has value x = 20, y = 20, z = 0
c = a.Scale(5);
```

Often it’s desired when scaling a vector to have the resulting vector’s length exactly equal to the scaled amount. This is easily achieved by first normalizing a vector, in other words setting the vector’s length exactly equal to one.



```
a = Vector.ByCoordinates(1, 2, 3);
a_len = a.Length;

// set the a's length equal to 1.0
b = a.Normalized();
c = b.Scale(5);

// len is equal to 5
len = c.Length;
```

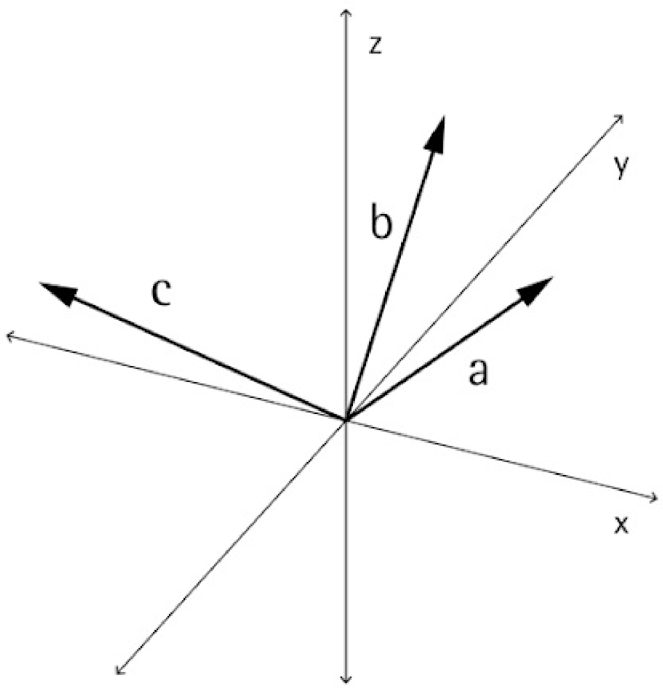
```
3.742
```

```
5.000
```

```
a = Vector.ByCoordinates(1, 2, 3);
a_len = a.Length;

// set the a's length equal to 1.0
b = a.Normalized();
c = b.Scale(5);

// len is equal to 5
len = c.Length;
```

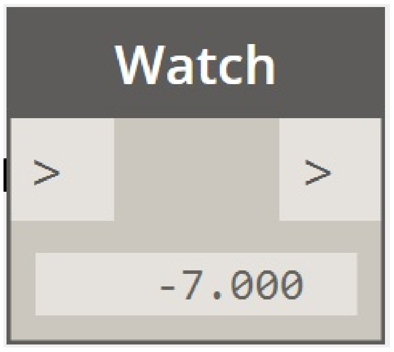
c still points in the same direction as a (1, 2, 3), though now it has length exactly equal to 5.

Two additional methods exist in vector math which don't have clear parallels with 1D math, the cross product and dot product. The cross product is a means of generating a Vector which is orthogonal (at 90 degrees to) to two existing Vectors. For example, the cross product of the x and y axes is the z axis, though the two input Vectors don't need to be orthogonal to each other. A cross product vector is calculated with the *Cross* method.



```
a = Vector.ByCoordinates(1, 0, 1);
b = Vector.ByCoordinates(0, 1, 1);

// c has value x = -1, y = -1, z = 1
c = a.Cross(b);
```

An additional, though somewhat more advanced function of vector math is the dot product. The dot product between two vectors is a real number (not a Vector object) that relates to, but is not exactly, the angle between two vectors. One useful properties of the dot product is that the dot product between two vectors will be 0 if and only if they are perpendicular. The dot product is calculated with the *Dot* method.



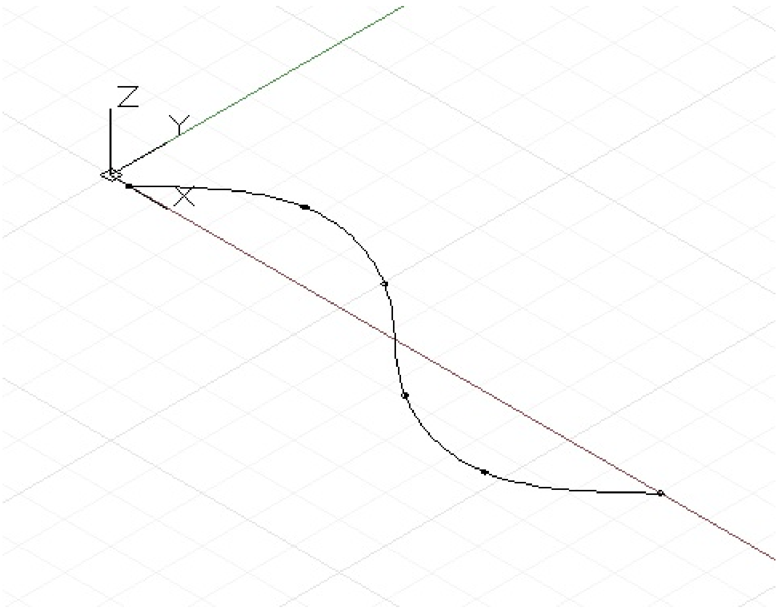
```
a = Vector.ByCoordinates(1, 2, 1);
b = Vector.ByCoordinates(5, -8, 4);

// d has value -7
d = a.Dot(b);
```

Curves: Interpolated and Control Points

Curves: Interpolated and Control Points

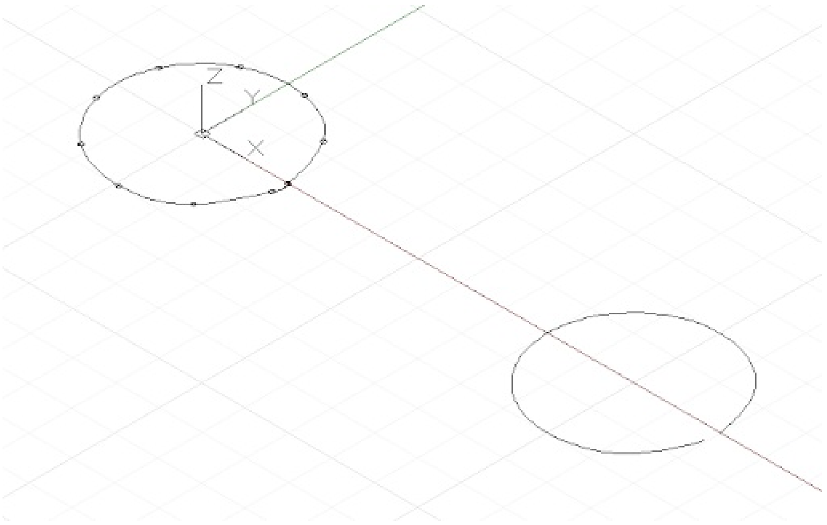
There are two fundamental ways to create free-form curves in Dynamo: specifying a collection of Points and having Dynamo interpolate a smooth curve between them, or a more low-level method by specifying the underlying control points of a curve of a certain degree. Interpolated curves are useful when a designer knows exactly the form a line should take, or if the design has specific constraints for where the curve can and cannot pass through. Curves specified via control points are in essence a series of straight line segments which an algorithm smooths into a final curve form. Specifying a curve via control points can be useful for explorations of curve forms with varying degrees of smoothing, or when a smooth continuity between curve segments is required.

To create an interpolated curve, simply pass in a collection of Points to the *NurbsCurve.ByPoints* method.



```
num_pts = 6;

s = Math.Sin(0..360..#num_pts) * 4;

pts = Point.ByCoordinates(1..30..#num_pts, s, 0);

int_curve = NurbsCurve.ByPoints(pts);
```

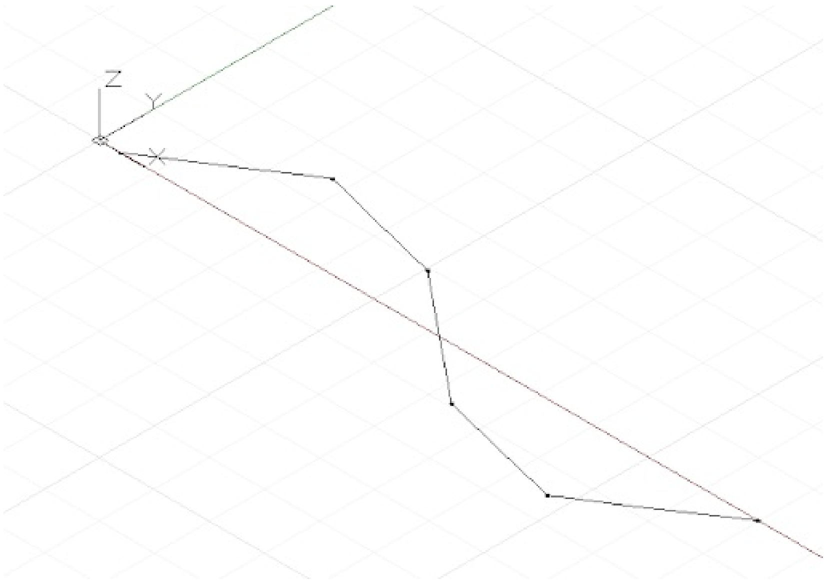
The generated curve intersects each of the input points, beginning and ending at the first and last point in the collection, respectively. An optional periodic parameter can be used to create a periodic curve which is closed. Dynamo will automatically fill in the missing segment, so a duplicate end point (identical to the start point) isn't needed.



```
pts = Point.ByCoordinates(Math.Cos(0..350..#10),
    Math.Sin(0..350..#10), 0);
```

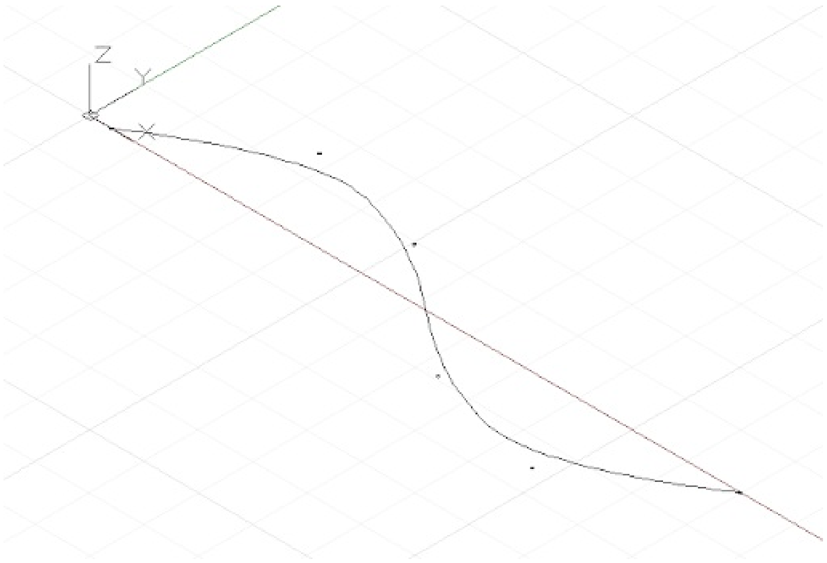
```
// create an closed curve
crv = NurbsCurve.ByPoints(pts, true);

// the same curve, if left open:
crv2 = NurbsCurve.ByPoints(pts.Translate(5, 0, 0),
    false);
```

NurbsCurves are generated in much the same way, with input points represent the endpoints of a straight line segment, and a second parameter specifying the amount and type of smoothing the curve undergoes, called the degree.* A curve with degree 1 has no smoothing; it is a polyline.



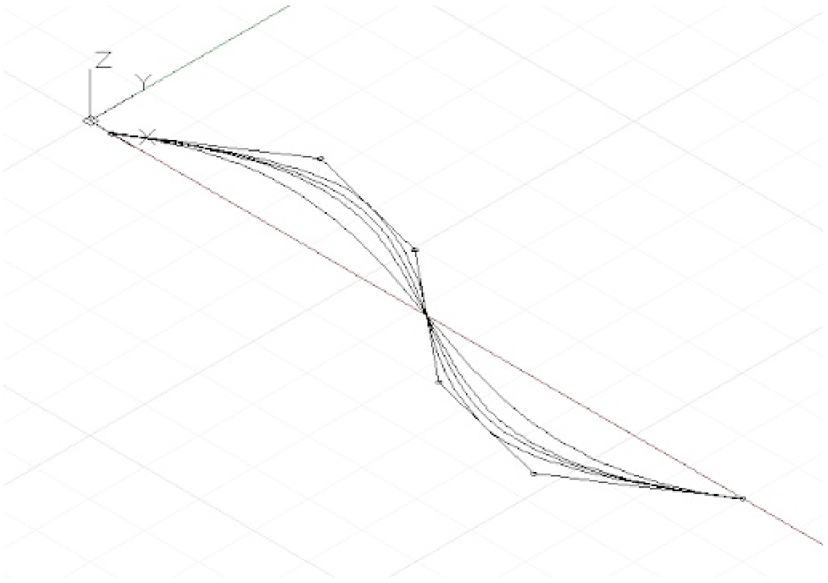
```
num_pts = 6;

pts = Point.ByCoordinates(1..30..#num_pts,
    Math.Sin(0..360..#num_pts) * 4, 0);

// a B-Spline curve with degree 1 is a polyline
ctrl_curve = NurbsCurve.ByControlPoints(pts, 1);
```

A curve with degree 2 is smoothed such that the curve intersects and is tangent to the midpoint of the polyline segments:



```
num_pts = 6;

pts = Point.ByCoordinates(1..30..#num_pts,
    Math.Sin(0..360..#num_pts) * 4, 0);

// a B-Spline curve with degree 2 is smooth
ctrl_curve = NurbsCurve.ByControlPoints(pts, 2);
```

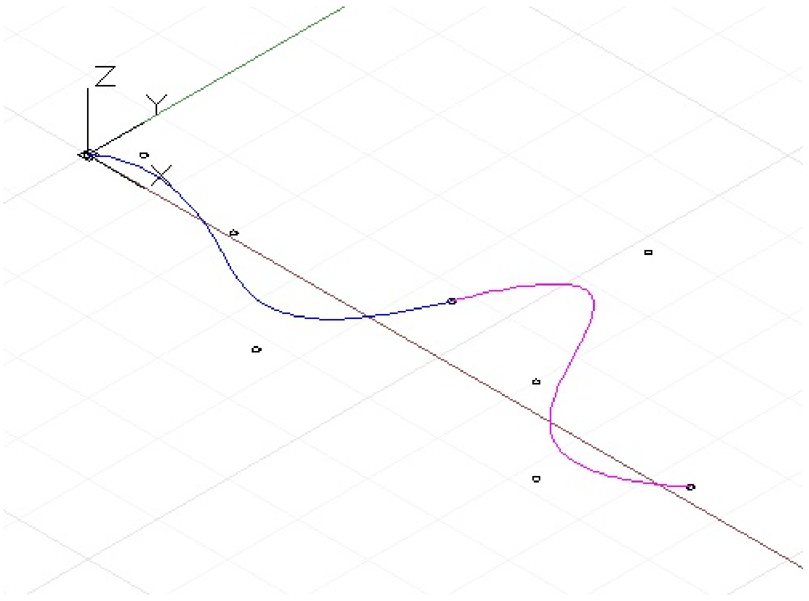
Dynamo supports NURBS (Non-uniform rational B-spline) curves up to degree 20, and the following script illustrates the effect increasing levels of smoothing has on the shape of a curve:



```
num_pts = 6;

pts = Point.ByCoordinates(1..30..#num_pts,
    Math.Sin(0..360..#num_pts) * 4, 0);

def create_curve(pts : Point[], degree : int)
{
    return = NurbsCurve.ByControlPoints(pts,
        degree);
}

ctrl_crvs = create_curve(pts, 1..11);
```

Note that you must have at least one more control point than the degree of the curve.

Another benefit of constructing curves by control vertices is the ability to maintain tangency between individual curve segments. This is done by extracting the direction between the last two control points, and continuing this direction with the first two control points of the following curve. The following example creates two separate NURBS curves which are nevertheless as smooth as one curve:



```
pts_1 = {};

pts_1[0] = Point.ByCoordinates(0, 0, 0);
pts_1[1] = Point.ByCoordinates(1, 1, 0);
pts_1[2] = Point.ByCoordinates(5, 0.2, 0);
pts_1[3] = Point.ByCoordinates(9, -3, 0);
pts_1[4] = Point.ByCoordinates(11, 2, 0);

crv_1 = NurbsCurve.ByControlPoints(pts_1, 3);

pts_2 = {};
```

```
pts_2[0] = pts_1[4];
end_dir = pts_1[4].Subtract(pts_1[3].AsVector());

pts_2[1] = Point.ByCoordinates(pts_2[0].X + end_dir.X,
    pts_2[0].Y + end_dir.Y, pts_2[0].Z + end_dir.Z);

pts_2[2] = Point.ByCoordinates(15, 1, 0);
pts_2[3] = Point.ByCoordinates(18, -2, 0);
pts_2[4] = Point.ByCoordinates(21, 0.5, 0);

crv_2 = NurbsCurve.ByControlPoints(pts_2, 3);
```

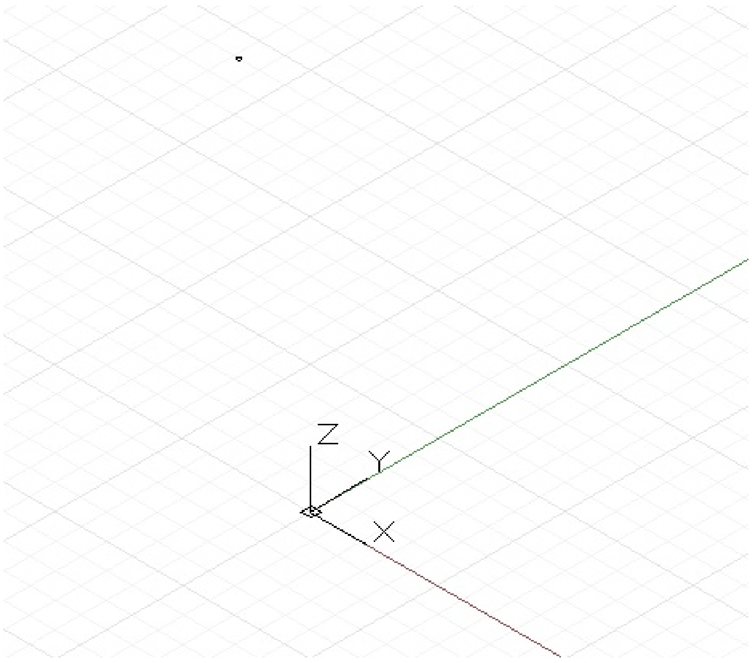
- This is a very simplified description of NURBS curve geometry, for a more accurate and detailed discussion see Pottmann, et al, 2007, in the references.

Translation, Rotation, and Other Transformations

Translation, Rotation, and Other Transformations

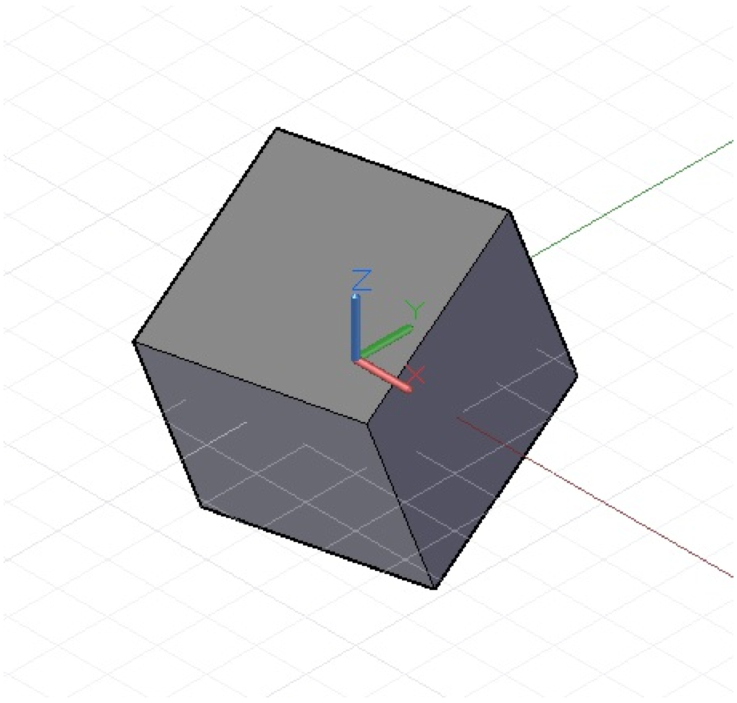
Certain geometry objects can be created by explicitly stating x, y, and z coordinates in three-dimensional space. More often, however, geometry is moved into its final position using geometric transformations on the object itself or on its underlying CoordinateSystem.

The simplest geometric transformation is a translation, which moves an object a specified number of units in the x, y, and z directions.



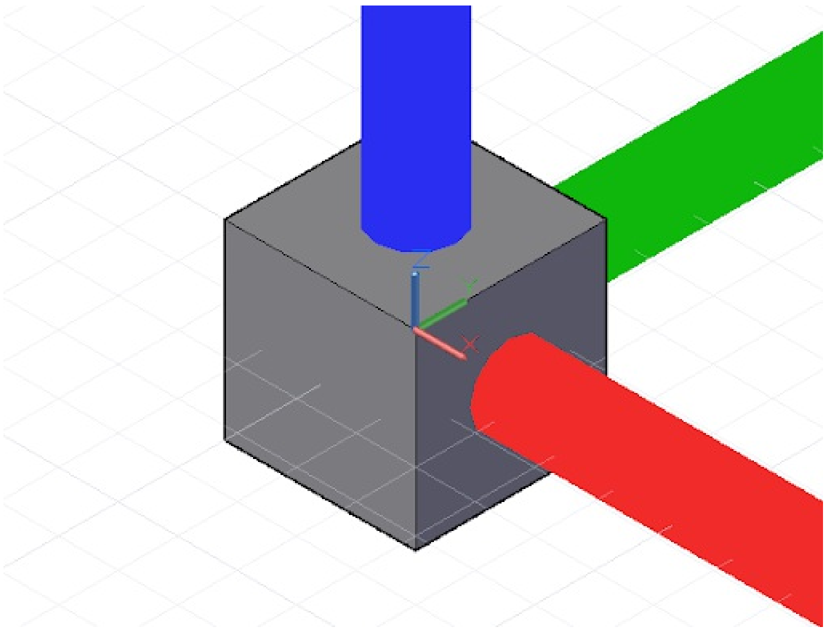
```
// create a point at x = 1, y = 2, z = 3
p = Point.ByCoordinates(1, 2, 3);

// translate the point 10 units in the x direction,
// -20 in y, and 50 in z
// p2's new position is x = 11, y = -18, z = 53
p2 = p.Translate(10, -20, 50);
```

While all objects in Dynamo can be translated by appending the *.Translate* method to the end of the object's name, more complex transformations require transforming the object from one underlying CoordinateSystem to a new CoordinateSystem. For instance, to rotate an object 45 degrees around the x axis, we would transform the object from its existing CoordinateSystem with no rotation, to a CoordinateSystem which had been rotated 45 degrees around the x axis with the *.Transform* method:



```
cube = Cuboid.ByLengths(CoordinateSystem.Identity(),
    10, 10, 10);

new_cs = CoordinateSystem.Identity();
new_cs2 = new_cs.Rotate(Point.ByCoordinates(0, 0),
    Vector.ByCoordinates(1,0,0.5), 25);

// get the existing coordinate system of the cube
old_cs = CoordinateSystem.Identity();

cube2 = cube.Transform(old_cs, new_cs2);
```

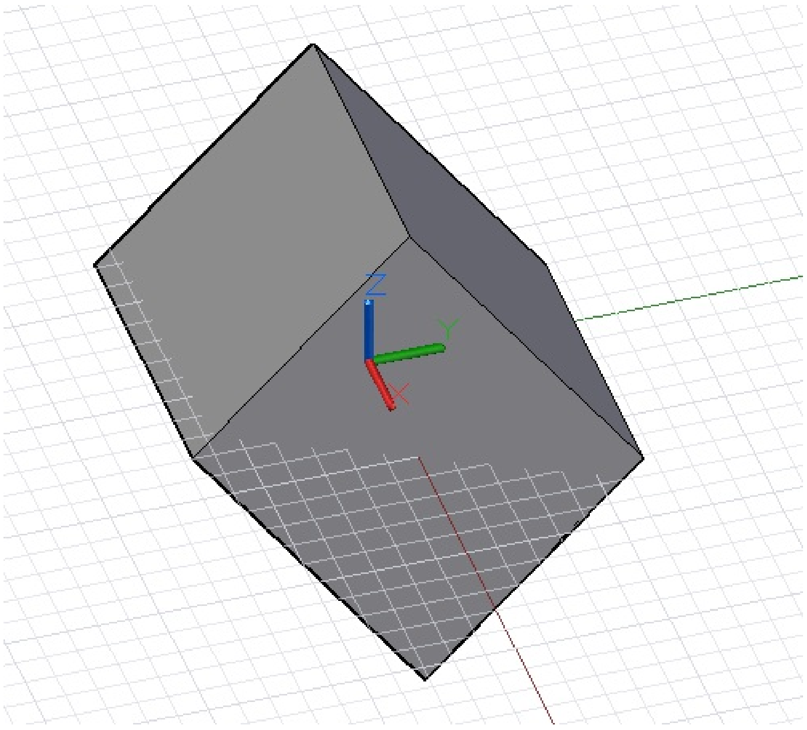
In addition to being translated and rotated, CoordinateSystems can also be created scaled or sheared. A CoordinateSystem can be scaled with the *.Scale* method:



```
cube = Cuboid.ByLengths(CoordinateSystem.Identity(),
    10, 10, 10);

new_cs = CoordinateSystem.Identity();
new_cs2 = new_cs.Scale(20);

old_cs = CoordinateSystem.Identity();
```

```
cube2 = cube.Transform(old_cs, new_cs2);
```

Sheared CoordinateSystems are created by inputting non-orthogonal vectors into the CoordinateSystem constructor.



```
new_cs = CoordinateSystem.ByOriginVectors(
    Point.ByCoordinates(0, 0, 0),
    Vector.ByCoordinates(-1, -1, 1),
    Vector.ByCoordinates(-0.4, 0, 0));

old_cs = CoordinateSystem.Identity();

cube = Cuboid.ByLengths(CoordinateSystem.Identity(),
    5, 5, 5);

new_curves = cube.Transform(old_cs, new_cs);
```

Scaling and shearing are comparatively more complex geometric transformations than rotation and translation, so not every Dynamo object can undergo these transformations. The following table outlines which Dynamo objects can have non-uniformly scaled CoordinateSystems, and sheared CoordinateSystems.

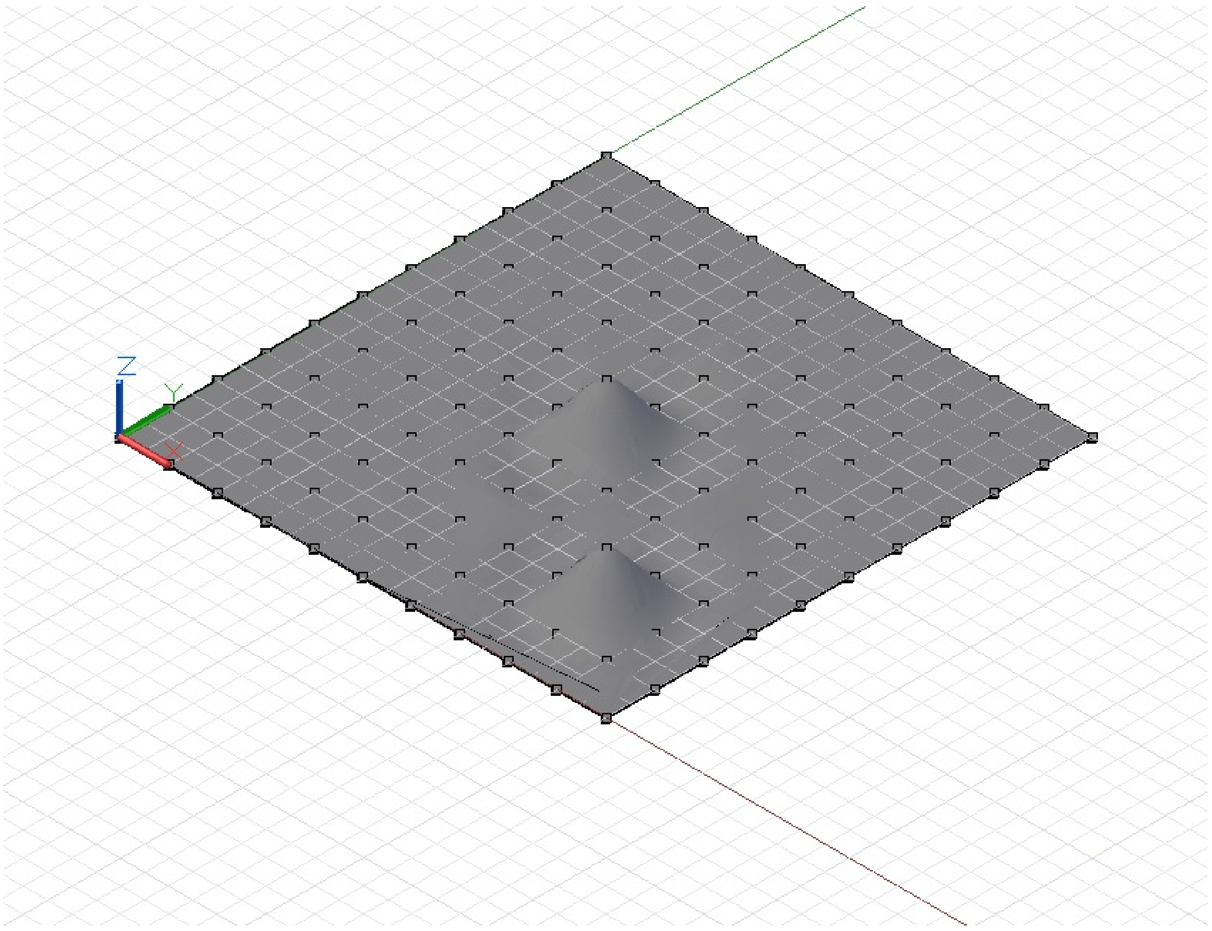
Class	Non-Uniformly Scaled CoordinateSystem	Sheared CoordinateSystem
Arc	No	No
NurbsCurve	Yes	Yes
NurbsSurface	No	No
Circle	No	No
Line	Yes	Yes
Plane	No	No
Point	Yes	Yes
Polygon	No	No
Solid	No	No
Surface	No	No
Text	No	No

Surfaces: Interpolated, Control Points, Loft, Revolve

Surfaces: Interpolated, Control Points, Loft, Revolve

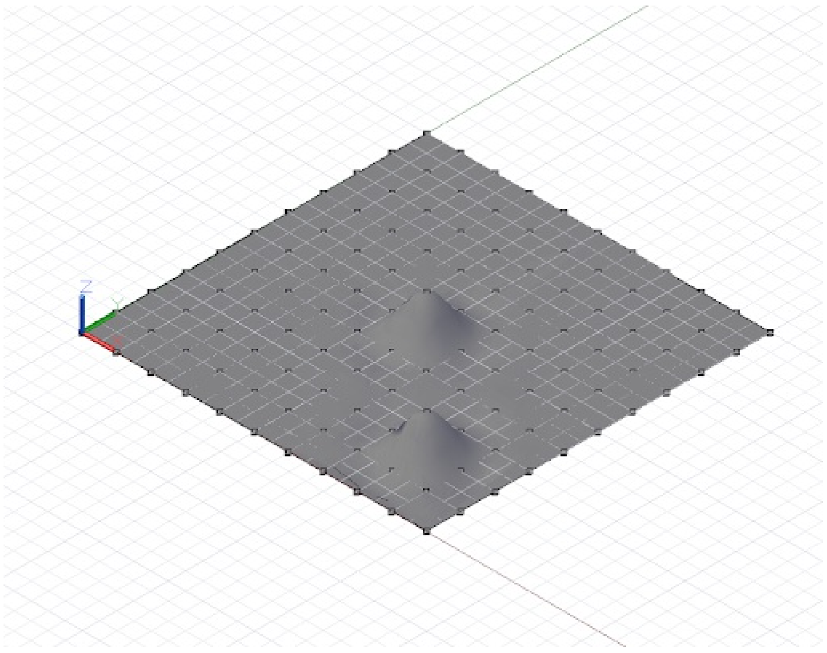
The two-dimensional analog to a NurbsCurve is the NurbsSurface, and like the freeform NurbsCurve, NurbsSurfaces can be constructed with two basic methods: inputting a set of base points and having Dynamo interpolate between them, and explicitly specifying the control points of the surface. Also like freeform curves, interpolated surfaces are useful when a designer knows precisely the shape a surface needs to take, or if a design requires the surface to pass through constraint points. On the other hand, Surfaces created by control points can be more useful for exploratory designs across various smoothing levels.

To create an interpolated surface, simply generate a two-dimensional collection of points approximating the shape of a surface. The collection must be rectangular, that is, not jagged. The method *NurbsSurface.ByPoints* constructs a surface from these points.



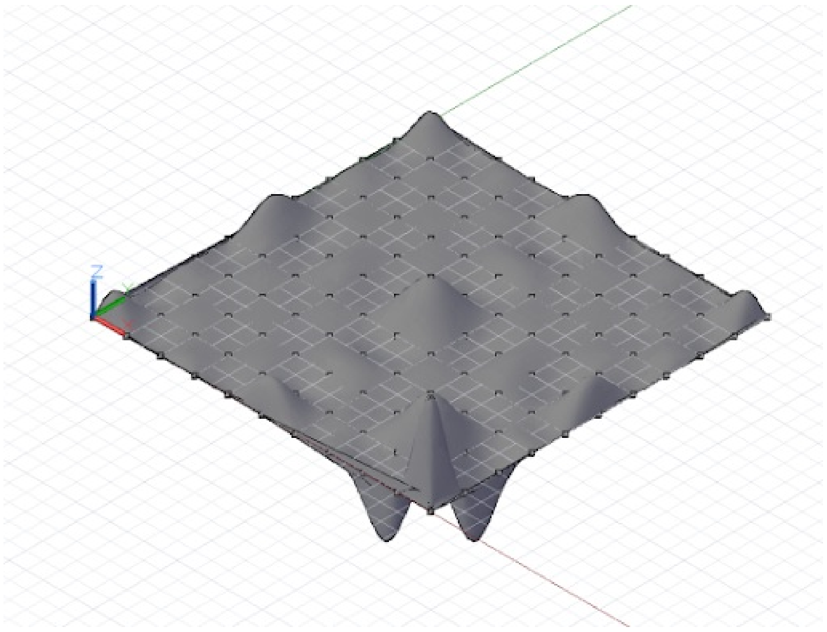
```
// python_points_1 is a set of Points generated with
// a Python script found in Chapter 12, Section 10

surf = NurbsSurface.ByPoints(python_points_1);
```

Freeform NurbsSurfaces can also be created by specifying underlying control points of a surface. Like NurbsCurves, the control points can be thought of as representing a quadrilateral mesh with straight segments, which, depending on the degree of the surface, is smoothed into the final surface form. To create a NurbsSurface by control points, include two additional parameters to *NurbsSurface.ByPoints*, indicating the degrees of the underlying curves in both directions of the surface.



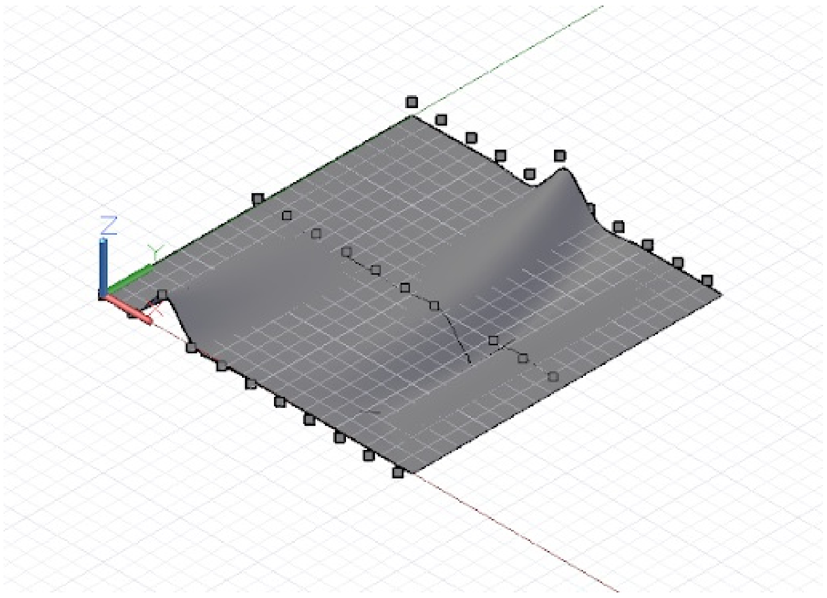
```
// python_points_1 is a set of Points generated with
// a Python script found in Chapter 12, Section 10

// create a surface of degree 2 with smooth segments
surf = NurbsSurface.ByPoints(python_points_1, 2, 2);
```

We can increase the degree of the NurbsSurface to change the resulting surface geometry:



```
// python_points_1 is a set of Points generated with
// a Python script found in Chapter 12, Section 10

// create a surface of degree 6
surf = NurbsSurface.ByPoints(python_points_1, 6, 6);
```

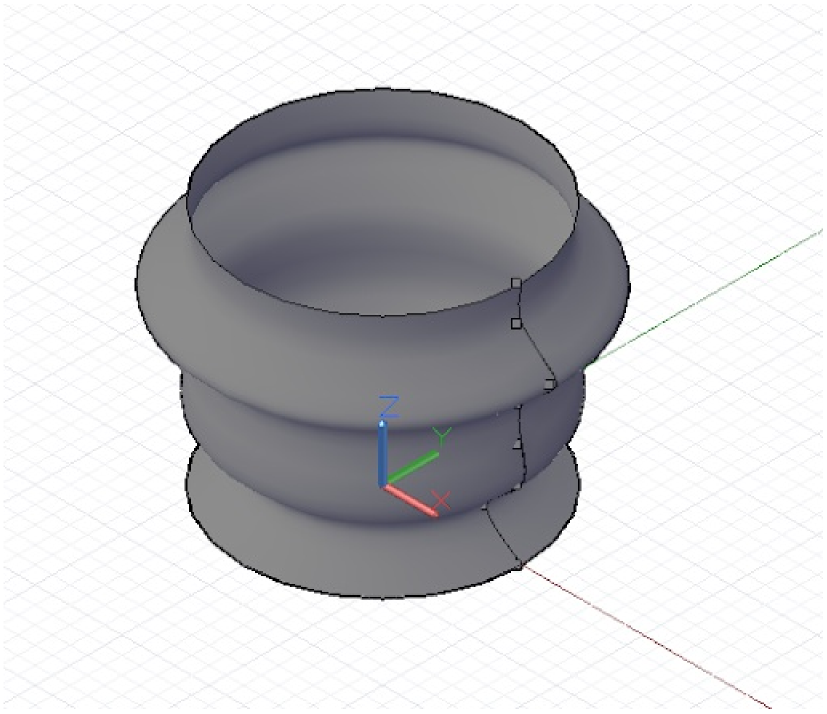
Just as Surfaces can be created by interpolating between a set of input points, they can be created by interpolating between a set of base curves. This is called lofting. A lofted curve is created using the *Surface.ByLoft* constructor, with a collection of input curves as the only parameter.



```
// python_points_2, 3, and 4 are generated with
// Python scripts found in Chapter 12, Section 10

c1 = NurbsCurve.ByPoints(python_points_2);
c2 = NurbsCurve.ByPoints(python_points_3);
c3 = NurbsCurve.ByPoints(python_points_4);

loft = Surface.ByLoft([c1, c2, c3]);
```

Surfaces of revolution are an additional type of surface created by sweeping a base curve around a central axis. If interpolated surfaces are the two-dimensional analog to interpolated curves, then surfaces of revolution are the two-dimensional analog to circles and arcs.

Surfaces of revolution are specified by a base curve, representing the “edge” of the surface; an axis origin, the base point of the surface; an axis direction, the central “core” direction; a sweep start angle; and a sweep end angle. These are used as the input to the *Surface.Revolve* constructor.



```
pts = {};
pts[0] = Point.ByCoordinates(4, 0, 0);
pts[1] = Point.ByCoordinates(3, 0, 1);
pts[2] = Point.ByCoordinates(4, 0, 2);
pts[3] = Point.ByCoordinates(4, 0, 3);
pts[4] = Point.ByCoordinates(4, 0, 4);
pts[5] = Point.ByCoordinates(5, 0, 5);
pts[6] = Point.ByCoordinates(4, 0, 6);
pts[7] = Point.ByCoordinates(4, 0, 7);
```

```
crv = NurbsCurve.ByPoints(pts);

axis_origin = Point.ByCoordinates(0, 0, 0);
axis = Vector.ByCoordinates(0, 0, 1);

surf = Surface.ByRevolve(crv, axis_origin, axis, 0,
    360);
```

Geometric Parameterization

Geometric Parameterization

In computational designs, curves and surfaces are frequently used as the underlying scaffold to construct subsequent geometry. In order for this early geometry to be used as a foundation for later geometry, the script must be able to extract qualities such as position and orientation across the entire area of the object. Both curves and surfaces support this extraction, and it is called parameterization.

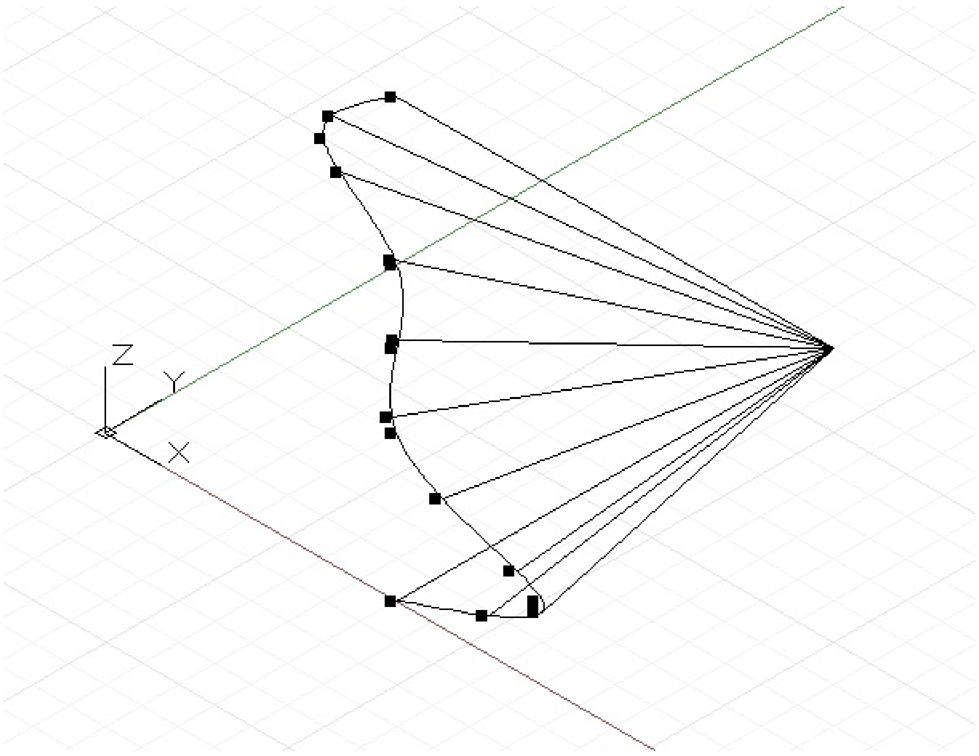
All of the points on a curve can be thought of as having a unique parameter ranging from 0 to 1. If we were to create a NurbsCurve based off of several control or interpolated points, the first point would have the parameter 0, and the last point would have the parameter 1. It’s impossible to know in advance what the exact parameter is any intermediate point is, which may sound like a severe limitation though is mitigated by a series of utility functions. Surfaces have a similar parameterization as curves, though with two parameters instead of one, called u and v. If we were to create a surface with the following points:

```
pts = [ [p1, p2, p3],
        [p4, p5, p6],
        [p7, p8, p9] ];
```

p1 would have parameter u = 0 v = 0, while p9 would have parameters u = 1 v = 1.

Parameterization isn’t particularly useful when determining points used to generate curves, its main use is to determine the locations if intermediate points generated by NurbsCurve and NurbsSurface constructors.

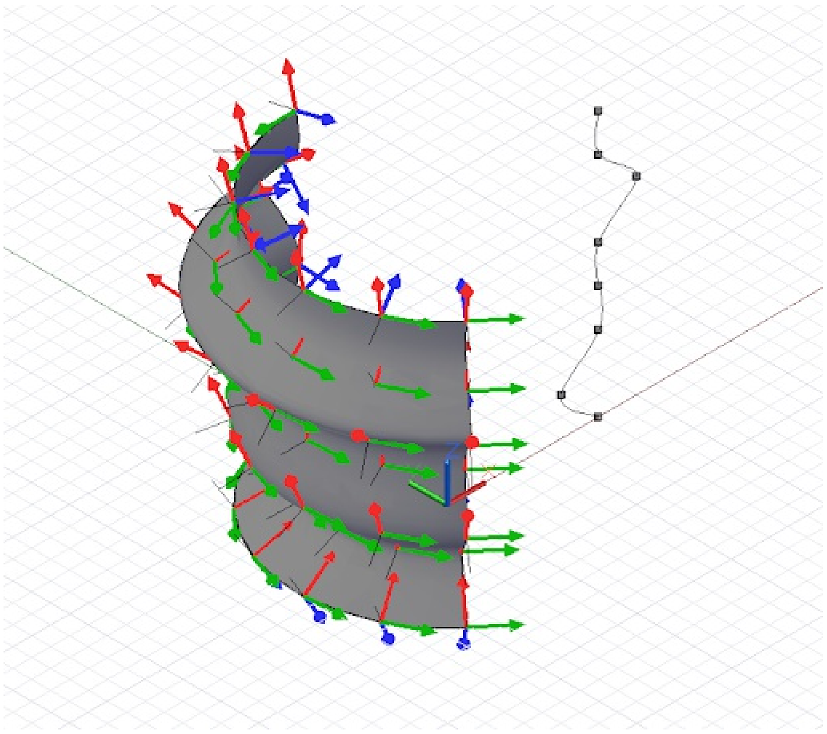
Curves have a method *PointAtParameter*, which takes a single double argument between 0 and 1, and returns the Point object at that parameter. For instance, this script finds the Points at parameters 0, .1, .2, .3, .4, .5, .6, .7, .8, .9, and 1:



```
pts = {};
pts[0] = Point.ByCoordinates(4, 0, 0);
pts[1] = Point.ByCoordinates(6, 0, 1);
pts[2] = Point.ByCoordinates(4, 0, 2);
pts[3] = Point.ByCoordinates(4, 0, 3);
pts[4] = Point.ByCoordinates(4, 0, 4);
pts[5] = Point.ByCoordinates(3, 0, 5);
pts[6] = Point.ByCoordinates(4, 0, 6);

crv = NurbsCurve.ByPoints(pts);

pts_at_param = crv.PointAtParameter(0..1..#11);

// draw Lines to help visualize the points
lines = Line.ByStartPointEndPoint(pts_at_param,
    Point.ByCoordinates(4, 6, 0));
```

Similarly, Surfaces have a method *PointAtParameter* which takes two arguments, the u and v parameter of the generated Point.

While extracting individual points on a curve and surface can be useful, scripts often require knowing the particular geometric characteristics at a parameter, such as what direction the Curve or Surface is facing. The method *CoordinateSystemAtParameter* finds not only the position but an oriented CoordinateSystem at the parameter of a Curve or Surface. For instance, the following script extracts oriented CoordinateSystems along a revolved Surface, and uses the orientation of the CoordinateSystems to generate lines which are sticking off normal to the surface:



```
pts = {};
pts[0] = Point.ByCoordinates(4, 0, 0);
pts[1] = Point.ByCoordinates(3, 0, 1);
pts[2] = Point.ByCoordinates(4, 0, 2);
pts[3] = Point.ByCoordinates(4, 0, 3);
pts[4] = Point.ByCoordinates(4, 0, 4);
pts[5] = Point.ByCoordinates(5, 0, 5);
pts[6] = Point.ByCoordinates(4, 0, 6);
pts[7] = Point.ByCoordinates(4, 0, 7);

crv = NurbsCurve.ByPoints(pts);

axis_origin = Point.ByCoordinates(0, 0, 0);
axis = Vector.ByCoordinates(0, 0, 1);

surf = Surface.ByRevolve(crv, axis_origin, axis, 90,
    140);

cs_array = surf.CoordinateSystemAtParameter(
    (0..1..#7)<1>, (0..1..#7)<2>);

def make_line(cs : CoordinateSystem) {
    lines_start = cs.Origin;
    lines_end = cs.Origin.Translate(cs.ZAxis, -0.75);

    return = Line.ByStartPointEndPoint(lines_start,
        lines_end);
}

lines = make_line(Flatten(cs_array));
```

As mentioned earlier, parameterization is not always uniform across the length of a Curve or a Surface, meaning that the parameter 0.5 doesn’t always correspond to the midpoint, and 0.25 doesn’t always correspond to the point one quarter along a curve or surface. To get around this limitation, Curves have an additional set of parameterization commands which allow you to find a point at specific lengths along a Curve.

Intersection and Trim

Intersection and Trim

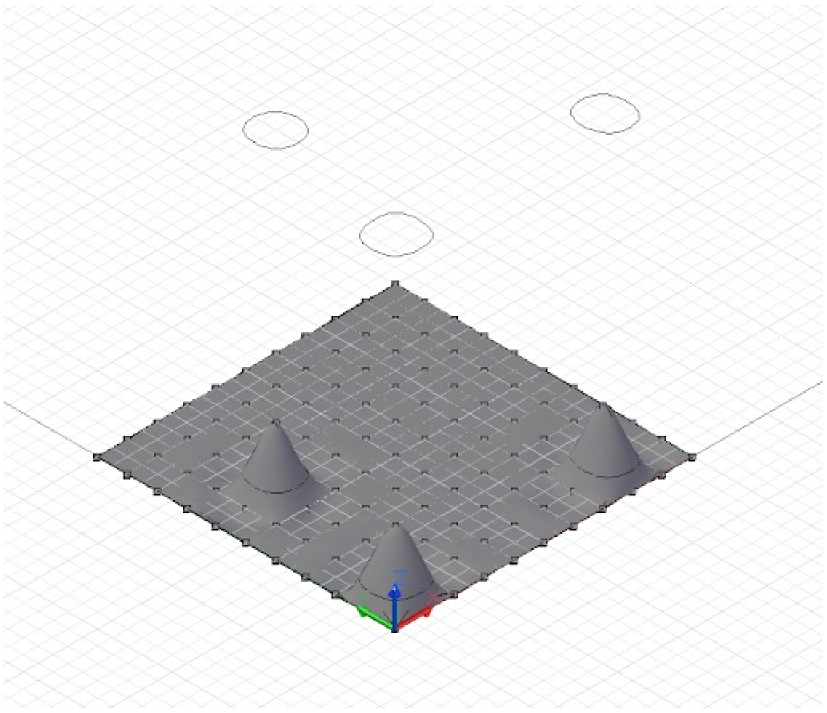
Many of the examples so far have focused on the construction of higher dimensional geometry from lower dimensional objects. Intersection methods allow this higher dimensional geometry to generate lower dimensional objects, while the trim and select trim commands allow script to heavily modify geometric forms after they've been created.

The *Intersect* method is defined on all pieces of geometry in Dynamo, meaning that in theory any piece of geometry can be intersected with any other piece of geometry. Naturally some intersections are meaningless, such as intersections involving Points, as the resulting object will always be the input Point itself. The other possible combinations of intersections between objects are outlined in the following chart. The following chart outlines the result of various intersection operations:

Intersect

With:	**Surface**	**Curve**	**Plane**	**Solid**
Surface	Curve	Point	Point, Curve	Surface
Curve	Point	Point	Point	Curve
Plane	Curve	Point	Curve	Curve
Solid	Surface	Curve	Curve	Solid

The following very simple example demonstrates the intersection of a plane with a NurbsSurface. The intersection generates a NurbsCurve array, which can be used like any other NurbsCurve.



```
// python_points_5 is a set of Points generated with
// a Python script found in Chapter 12, Section 10

surf = NurbsSurface.ByPoints(python_points_5, 3, 3);

WCS = CoordinateSystem.Identity();

pl = Plane.ByOriginNormal(WCS.Origin.Translate(0, 0,
    0.5), WCS.ZAxis);

// intersect surface, generating three closed curves
crvs = surf.Intersect(pl);

crvs_moved = crvs.Translate(0, 0, 10);
```

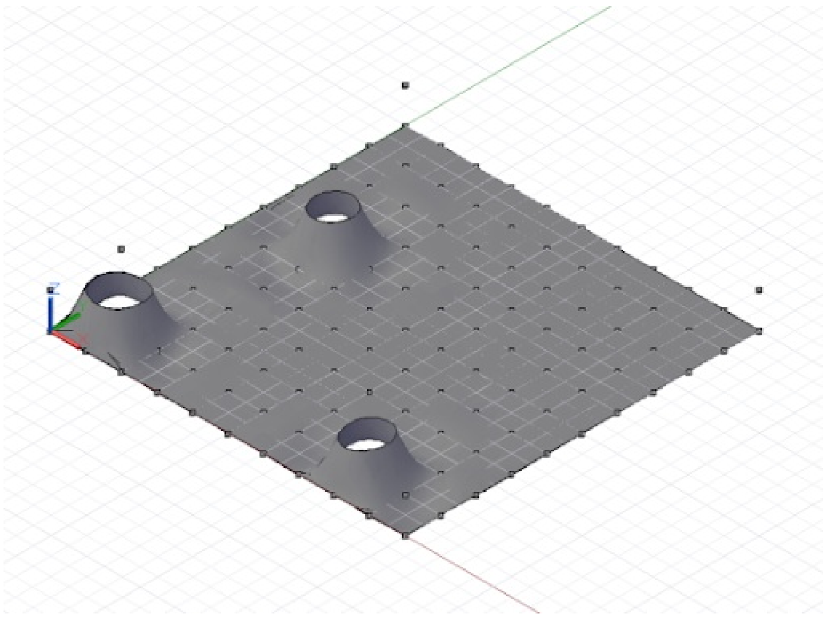
The *Trim* method is very similar to the Intersect method, in that it is defined for almost every piece of geometry. However, there are far more limitations on *Trim* than on *Intersect*.

Trim

Using:	**Point**	**Curve**	**Plane**	**Surface**	**Solid**
On: Curve	Yes	No	No	No	No

Polygon	-	No	Yes	No	No
Surface	-	Yes	Yes	Yes	Yes
Solid	-	-	Yes	Yes	Yes

Something to note about *Trim* methods is the requirement of a “select” point, a point which determines which geometry to discard, and which pieces to keep. Dynamo finds and discards the trimmed geometry closest to the select point.



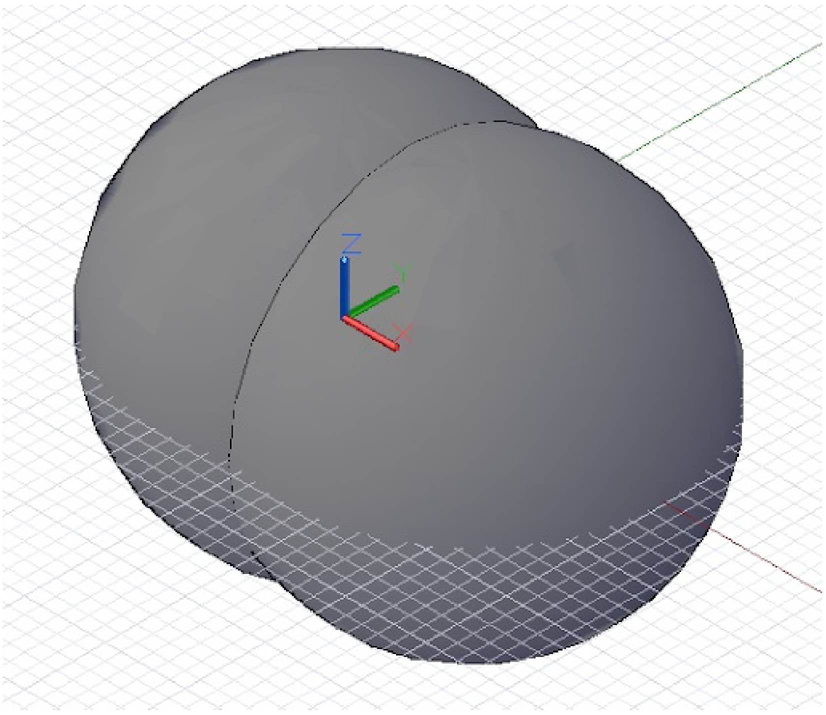
```
// python_points_5 is a set of Points generated with
// a Python script found in Chapter 12, Section 10

surf = NurbsSurface.ByPoints(python_points_5, 3, 3);

tool_pts = Point.ByCoordinates((-10..20..10)<1>,
    (-10..20..10)<2>, 1);

tool = NurbsSurface.ByPoints(tool_pts);

pick_point = Point.ByCoordinates(8, 1, 3);

result = surf.Trim(tool, pick_point);
```

Geometric Booleans

Geometric Booleans

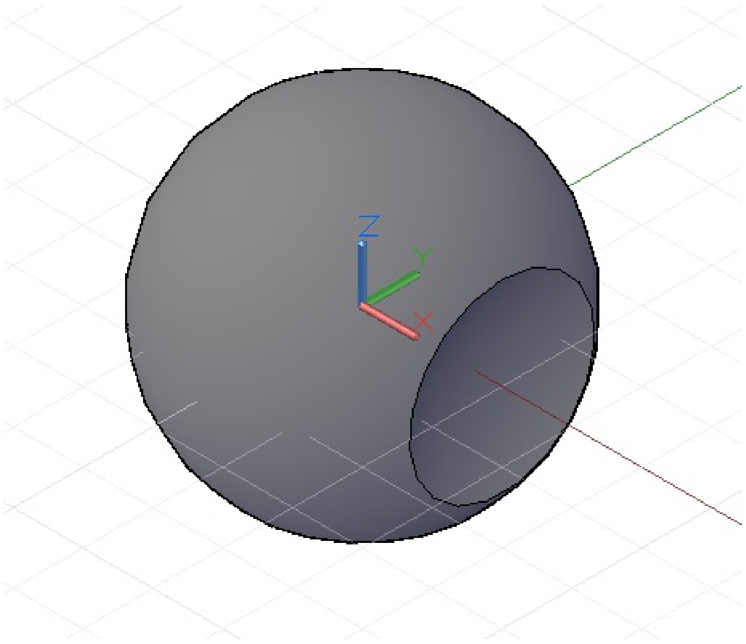
Intersect, *Trim*, and *SelectTrim* are primarily used on lower-dimensional geometry such as Points, Curves, and Surfaces. Solid geometry on the other hand, has an additional set of methods for modifying form after their construction, both by subtracting material in a manner similar to *Trim* and combining elements together to form a larger whole.

The *Union* method takes two solid objects and creates a single solid object out of the space covered by both objects. The overlapping space between objects is combined into the final form. This example combines a Sphere and a Cuboid into a single solid Sphere-Cube shape:



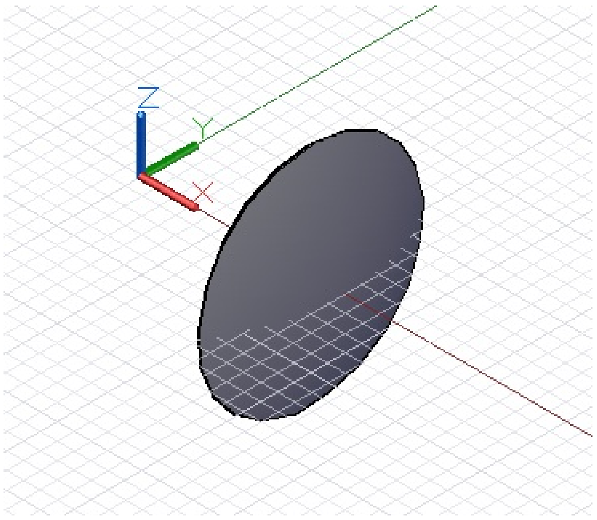
```
s1 = Sphere.ByCenterPointRadius(
    CoordinateSystem.Identity().Origin, 6);

s2 = Sphere.ByCenterPointRadius(
    CoordinateSystem.Identity().Origin.Translate(4, 0,
    0), 6);

combined = s1.Union(s2);
```

The *Difference* method, like *Trim*, subtracts away the contents of the input tool solid from the base solid. In this example we carve out a small indentation out of a sphere:



```
s = Sphere.ByCenterPointRadius(
    CoordinateSystem.Identity().Origin, 6);

tool = Sphere.ByCenterPointRadius(
    CoordinateSystem.Identity().Origin.Translate(10, 0,
    0), 6);

result = s.Difference(tool);
```

The *Intersect* method returns the overlapping Solid between two solid Inputs. In the following example, *Difference* has been changed to *Intersect*, and the resulting Solid is the missing void initially carved out:

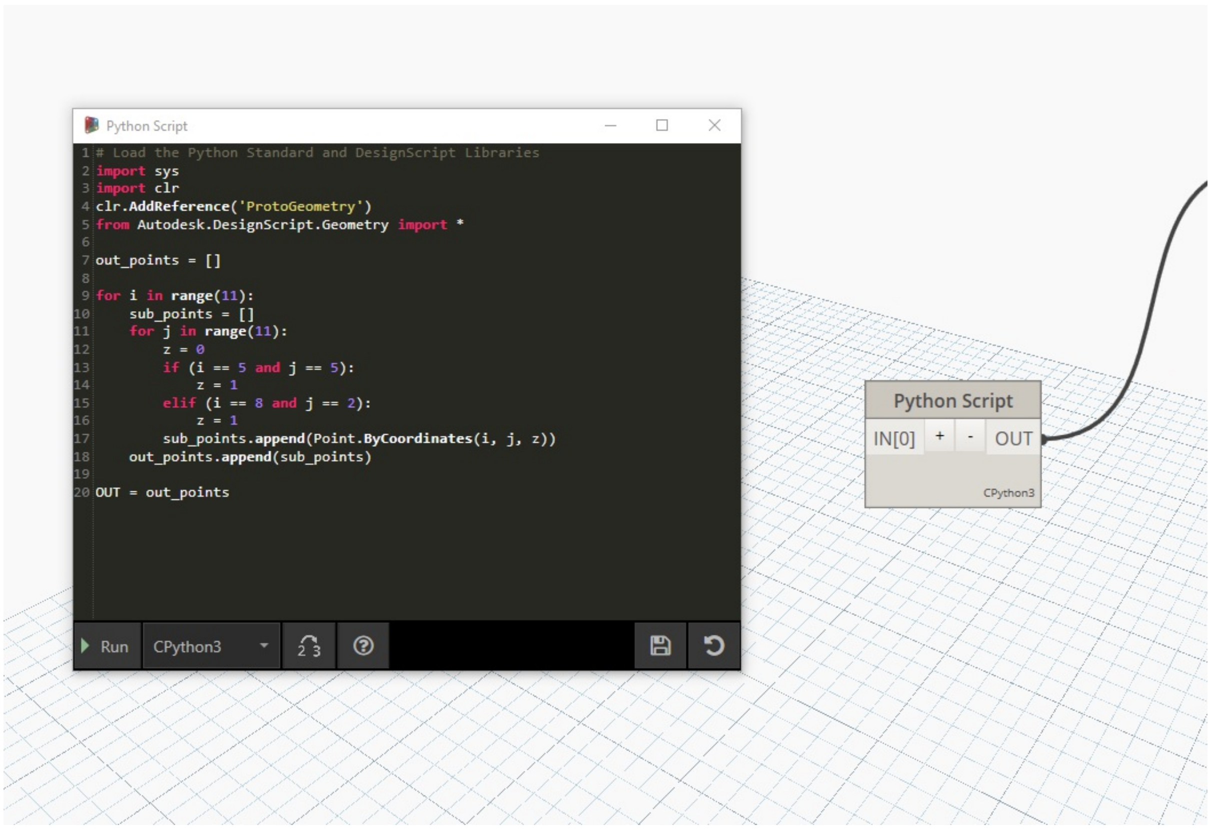


```
s = Sphere.ByCenterPointRadius(
    CoordinateSystem.Identity().Origin, 6);

tool = Sphere.ByCenterPointRadius(
    CoordinateSystem.Identity().Origin.Translate(10, 0,
    0), 6);

result = s.Intersect(tool);
```

Python Point Generators

Python Point Generators

The following Python scripts generate point arrays for several examples. They should be pasted into a Python Script node as follows:



```
# Load the Python Standard and DesignScript Libraries
import sys
import clr
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *

out_points = []

for i in range(11):
    sub_points = []
    for j in range(11):
        z = 0
        if (i == 5 and j == 5):
            z = 1
        elif (i == 8 and j == 2):
            z = 1
        sub_points.append(Point.ByCoordinates(i, j, z))
    out_points.append(sub_points)

OUT = out_points
```

python_points_1

```
out_points = []

for i in range(11):
    sub_points = []
    for j in range(11):
        z = 0
        if (i == 5 and j == 5):
            z = 1
        elif (i == 8 and j == 2):
            z = 1
        sub_points.append(Point.ByCoordinates(i, j, z))
    out_points.append(sub_points)

OUT = out_points
```

python_points_2

```
out_points = []

for i in range(11):
    z = 0
    if (i == 2):
        z = 1
    out_points.append(Point.ByCoordinates(i, 0, z))

OUT = out_points
```

python_points_3

```
out_points = []

for i in range(11):
    z = 0
    if (i == 7):
        z = -1
    out_points.append(Point.ByCoordinates(i, 5, z))

OUT = out_points
```

python_points_4

```
out_points = []

for i in range(11):
    z = 0
    if (i == 5):
        z = 1
    out_points.append(Point.ByCoordinates(i, 10, z))

OUT = out_points
```

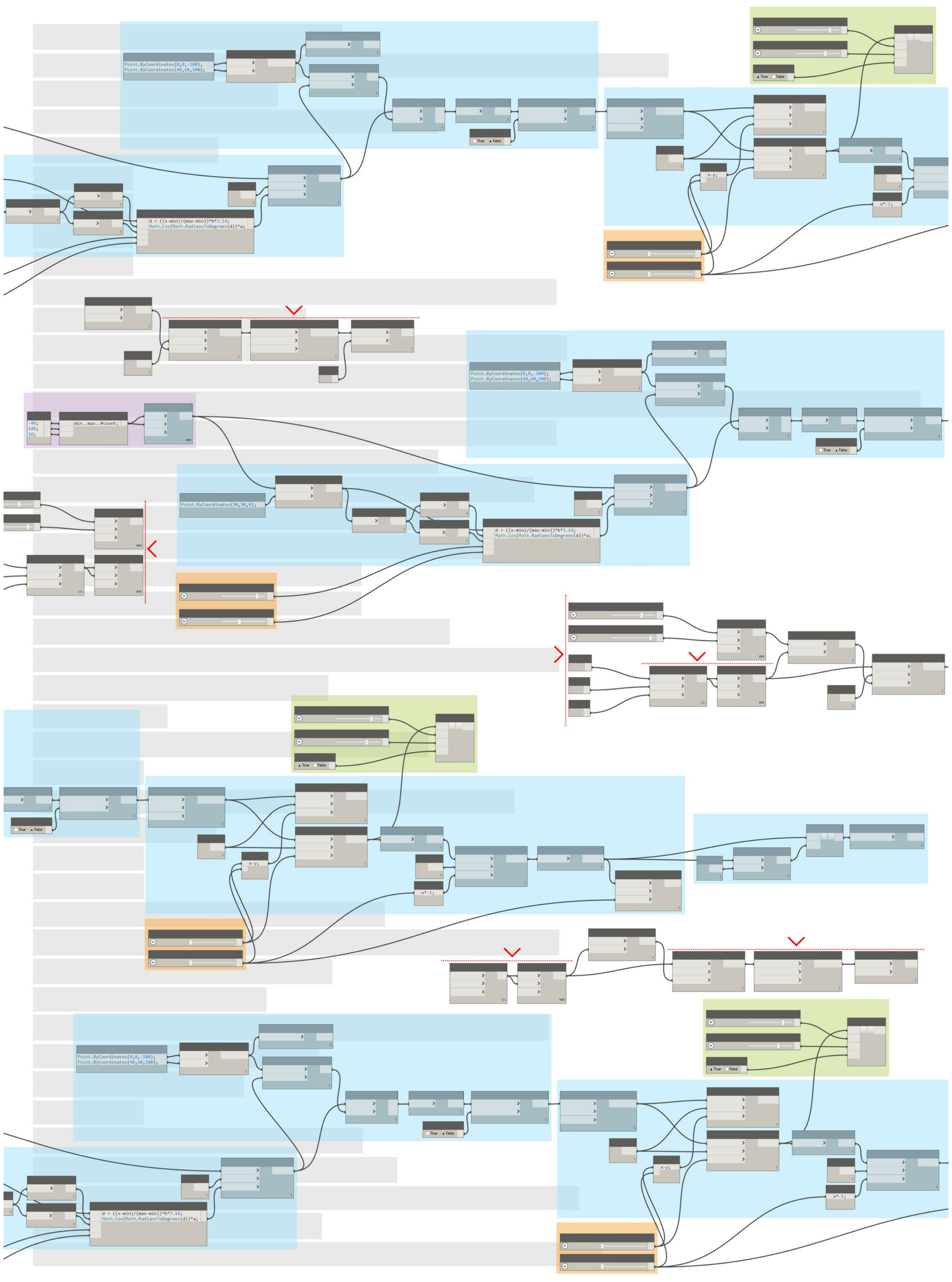
python_points_5

```
out_points = []

for i in range(11):
    sub_points = []
    for j in range(11):
        z = 0
        if (i == 1 and j == 1):
            z = 2
        elif (i == 8 and j == 1):
            z = 2
        elif (i == 2 and j == 6):
            z = 2
        sub_points.append(Point.ByCoordinates(i, j, z))
    out_points.append(sub_points)

OUT = out_points
```

Best Practices

Best Practices

This part of the primer is organized in the spirit of a journal of “best practices”. It sheds light on several strategies that we have learned, through experience and research, to be most conducive to quality parametric workflows. As designers and programmers, our metric for quality is primarily concerned with the maintainability, dependability, usability, and efficiency of our tools. While these best practices have specific examples for either visual or text based scripting, the principles are applicable to all programming environments and can inform many computational workflows.



```
Point.ByCoordinates(0,0,-100);
Point.ByCoordinates(40,60,100);
```

```
d = ((x-min)/(max-min))*b*3.14;
Math.Cos(Math.RadiansToDegrees(d))*a;
```

```
-40;
140;
10;
```

```
min..max..#count;
```

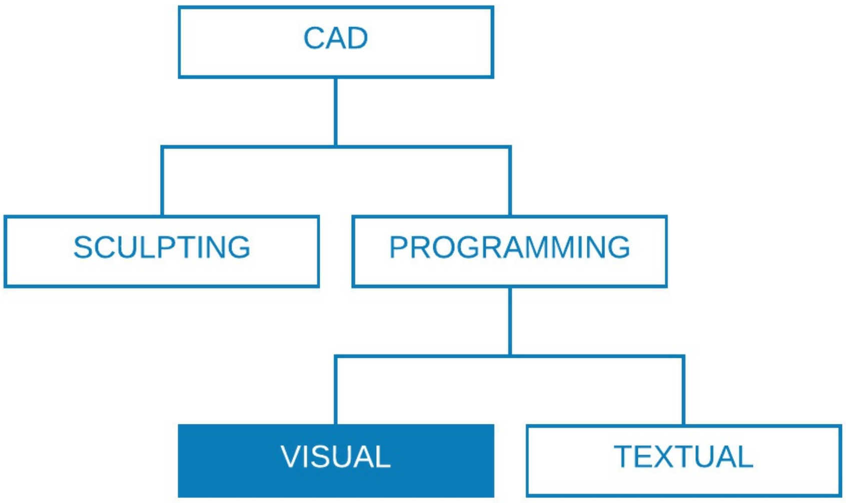
```
Point.ByCoordinates(0,0,100);
Point.ByCoordinates(40,60,100);
```

```
Point.ByCoordinates(90,90,0);
```

Graph Strategies

Graph Strategies

Prior to this chapter, the Primer has covered how to implement the powerful visual-scripting capabilities of Dynamo. A good understanding of these capabilities is a solid foundation and the first step in building robust visual programs. When we use our visual programs in the field, share them with colleagues, troubleshoot errors, or test limits we have additional issues to deal with. If someone else will be using your program or you are expecting to open it six months from now, it needs to have an immediate graphic and logical clarity. Dynamo has many tools to manage the complexity of your program, and this chapter will give guidelines on when to use them.

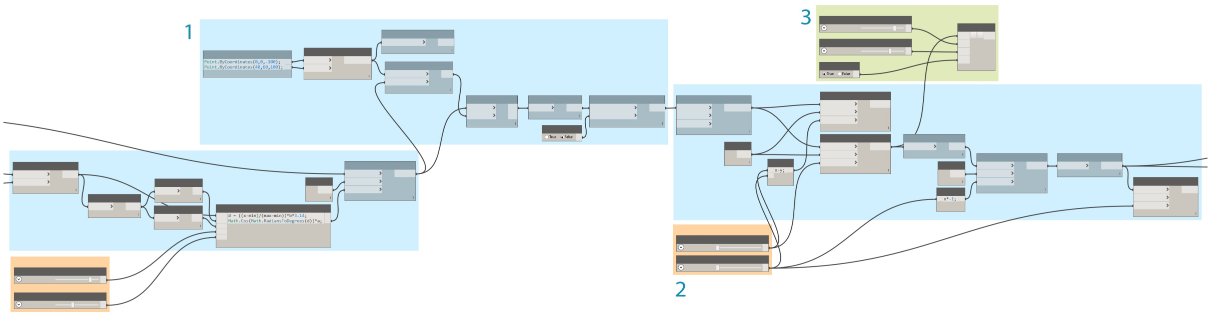


Reduce Complexity

As you develop your Dynamo graph and test ideas, it can quickly grow in size and complexity. While it is important that you create a functioning program, it is equally important to do it as simply as possible. Not only will your graph run faster and more predictably, you along with other users will understand its logic later on. The following are several ways that will help you clarify the logic of your graph.

Modularize with Groups

- Groups allow you to **create functionally distinct parts** as you build a program
- Groups allow you to **move large parts of the program** around while maintaining modularity and alignment
- You can change the **color of the group to differentiate** what Groups are doing (inputs vs functions)
- You can use groups to start **organizing your graph to streamline Custom Node creation**



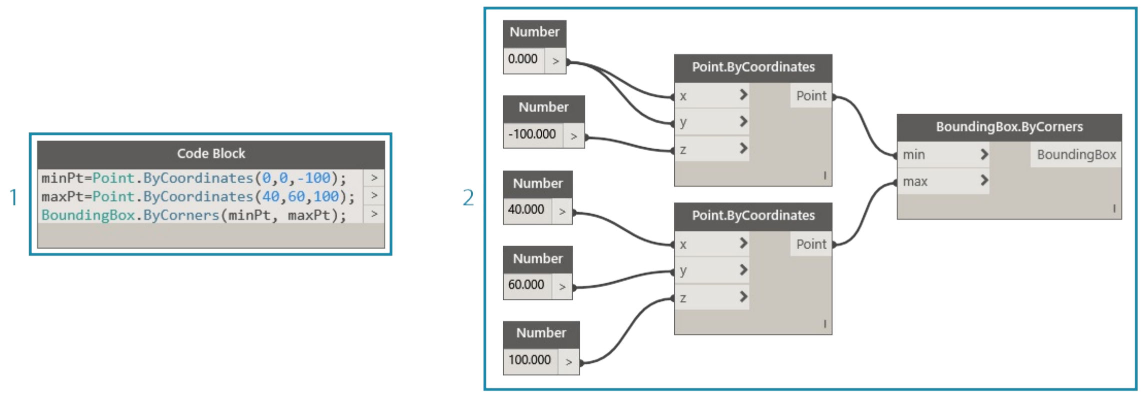
The colors in this program identify the purpose of each group. This strategy can be used to create hierarchy in any graphic standards or templates you develop.

1. Function group (blue)
2. Input group (orange)
3. Script group (green)

For how to use Groups, refer to Managing Your Program.

Develop efficiently with Code Blocks

- At times, you can use a Code Block to **type a number or node method faster than searching** (Point.ByCoordinates, Number, String, Formula)
- Code Blocks are useful **when you want to define custom functions in DesignScript to reduce the number of nodes in a graph**



```
minPt=Point.ByCoordinates(0,0,-100);
maxPt=Point.ByCoordinates(40,60,100);
BoundingBox.ByCorners(minPt, maxPt);
```

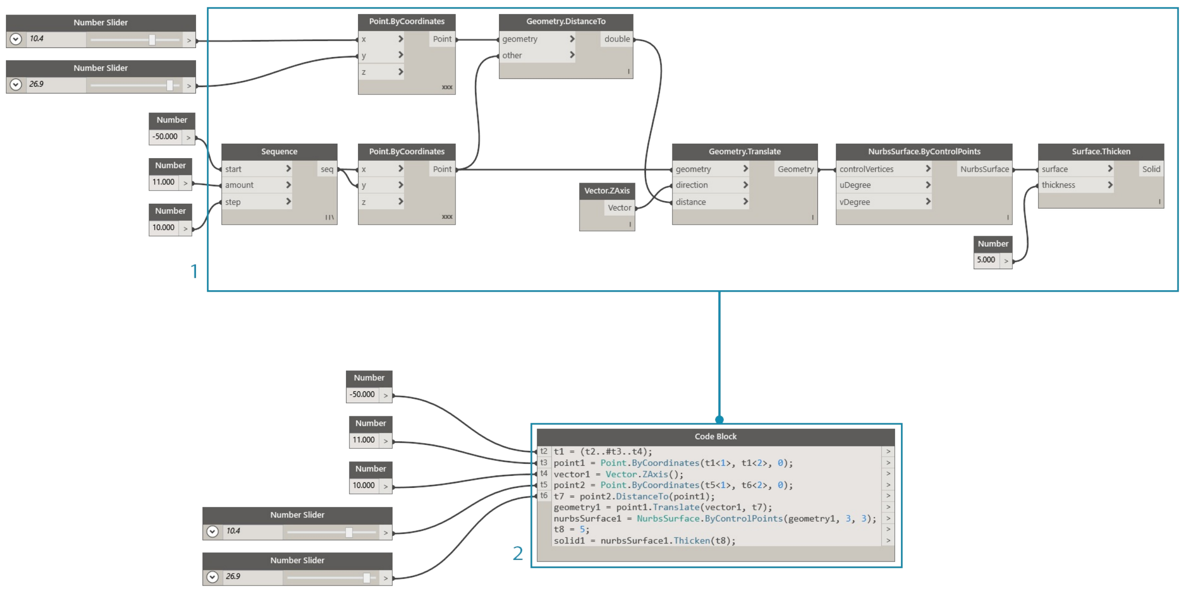
Both 1 and 2 perform the same function. It was much faster to write a few lines of code than it was to search for and add each node individually. The code block is also far more concise.

1. Design Script written in Code Block
2. Equivalent program in nodes

For how to use Code Block, refer to What's a Code Block.

Condense with Node to Code

- You can **reduce the complexity of a graph by using Node to Code** which will take a collection of simple nodes and write their corresponding DesignScript in a single Code Block
- Node to Code can **condense code without eliminating the program's clarity**
- The following are the **pros** of using Node to Code:
  - Easily condenses code into one component that is still editable
  - Can simplify a significant portion of the graph
  - Useful if the 'mini-program' will not often be edited
  - Useful for incorporating other code block functionality, like functions
- The following are the **cons** of using Node to Code:
  - Generic naming makes it less legible
  - More difficult to understand for other users
  - No easy way to return to the visual programming version



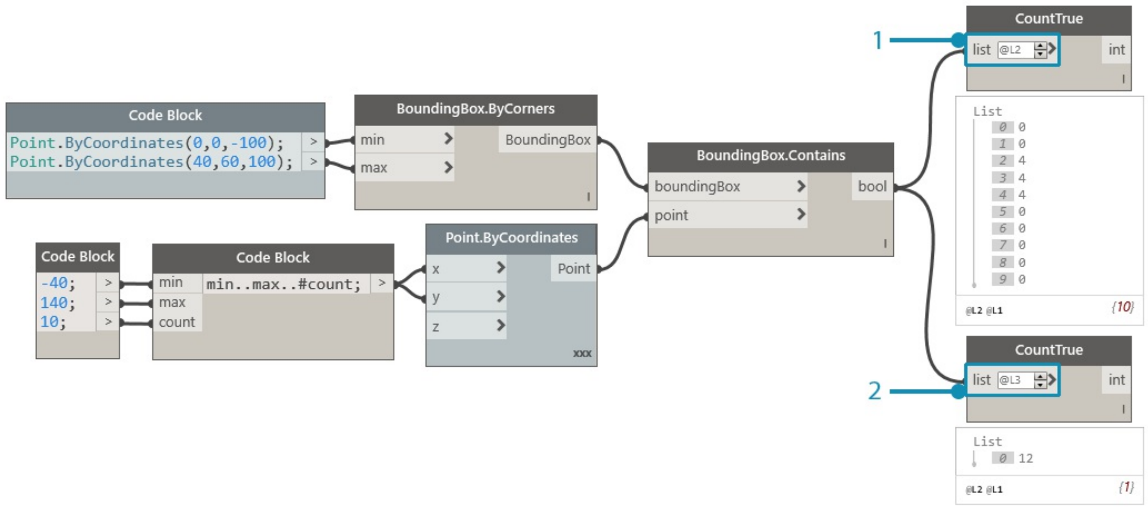
```
t1 = (t2..#t3..t4);
point1 = Point.ByCoordinates(t1<1>, t1<2>, 0);
vector1 = Vector.ZAxis();
point2 = Point.ByCoordinates(t5<1>, t6<2>, 0);
t7 = point2.DistanceTo(point1);
geometry1 = point1.Translate(vector1, t7);
nurbsSurface1 = NurbsSurface.ByControlPoints(geometry1, 3, 3);
t8 = 5;
solid1 = nurbsSurface1.Thicken(t8);
```

1. Existing program
2. Code Block created from Node to Code

For how to use Node to Code, refer to Design Script Syntax.

Access data flexibly with List@Level

- Using List@Level can help you **reduce the complexity of your graph by replacing List.Map and List.Combine nodes** which might occupy a considerable amount of canvas space
- List@Level provides you with a **quicker way than List.Map/List.Combine to construct node logic** by allowing you to access data at any level in a list right from the input port of a node



```
Point.ByCoordinates(0,0,-100);
Point.ByCoordinates(40,60,100);
```

```
-40;
140;
10;
```

```
min..max..#count;
```

```
List
  0 0
  1 0
  2 4
  3 4
  4 4
  5 0
  6 0
  7 0
  8 0
  9 0
@L2 @L1          {10}
```

```
List
  0 12
@L2 @L1          {1}
```

We can verify how many True values BoundingBox.Contains is returning and in which lists by activating List@Level for CountTrue's "list" input. List@Level allows the user to determine at which level the input will take data from. Using List@Level is flexible, efficient, and highly encouraged over other methods involving List.Map and List.Combine.

1. Counting true values at List Level 2
2. Counting true values at List Level 3

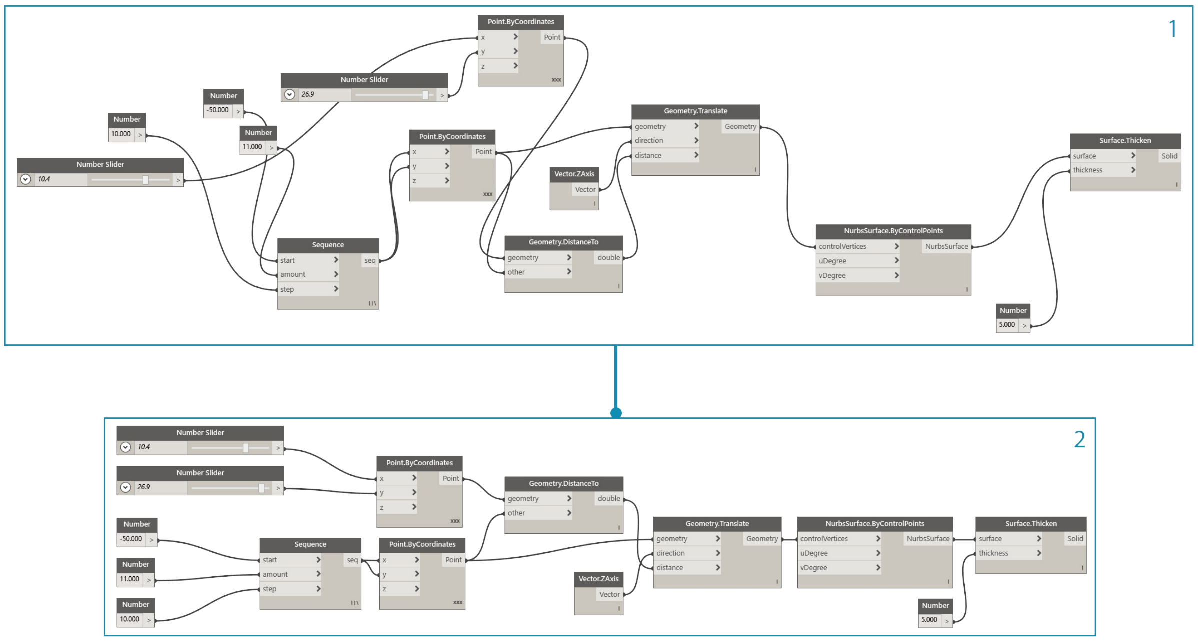
For how to use List@Level, refer to Lists of Lists.

Maintain Readability

In addition to making your graph as simple and efficient as possible, strive for graphic clarity. Despite your best efforts to make your graph intuitive with logical groupings, relationships might not be readily apparent. A simple Note inside of a Group or renaming a slider can save you or another user from unnecessary confusion or panning across the graph. The following are several ways that will help you apply graphic consistency within and across your graphs.

Visual continuity with Node Alignment

- To reduce your work after you finished building your graph, you should try to ensure the node layout is legible by **aligning nodes often and as you go**
- If others are going to be working with your graph, you should **ensure that your node-wire layout flows easily before shipping**
- To help you with alignment, **use the "Cleanup Node Layout" feature to automatically align** your graph, though less precisely than doing it yourself

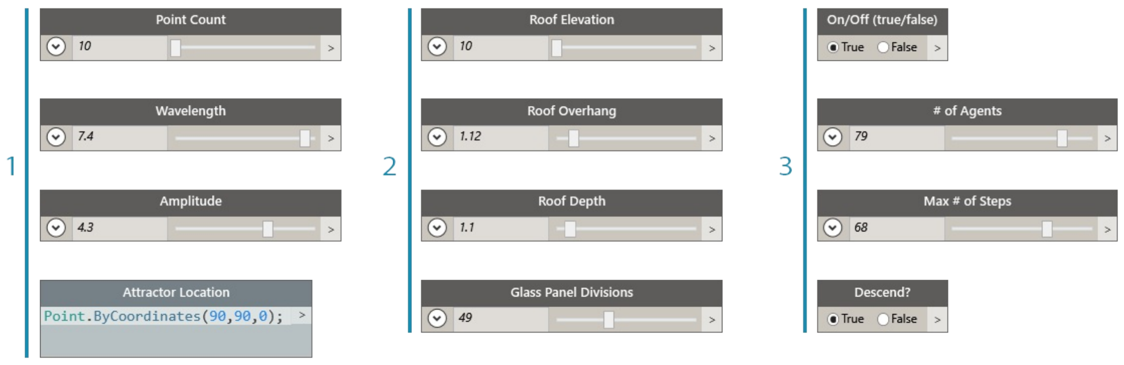


1. Unorganized graph
2. Aligned graph

For how to use Node Alignment, refer to [Managing Your Program](#).

Descriptive labeling by renaming

- Renaming inputs can help others easily understand your graph, **especially if what they plug into will be off the screen**
- **Be wary of renaming nodes other than inputs.** An alternative to this is creating a custom node from a node cluster and renaming that; it will be understood that it contains something else

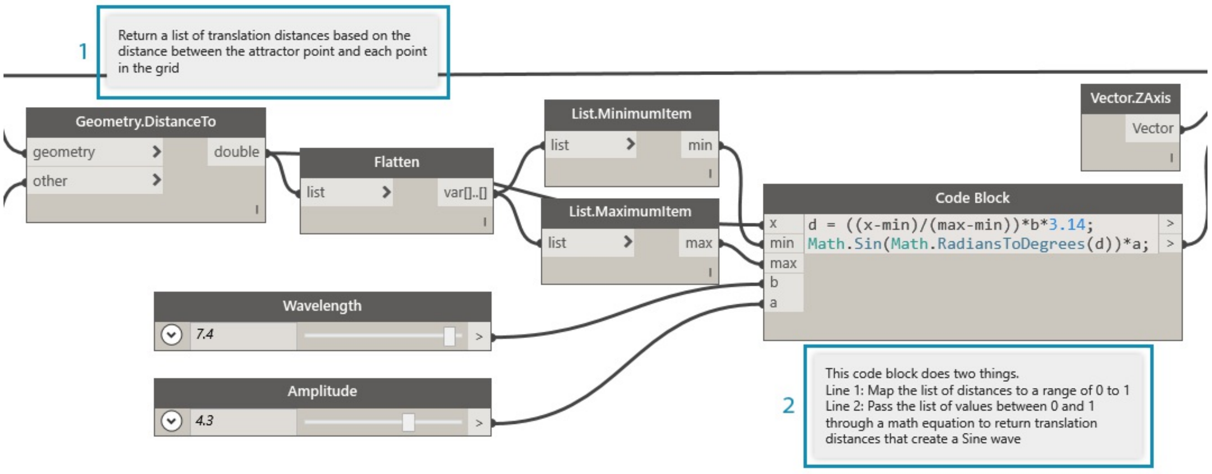


1. Inputs for surface manipulation
2. Inputs for architectural parameters
3. Inputs for drainage simulation script

To rename a node, right click on its name and choose "Rename Node...".

Explain with Notes

- You should add a Note if something in the **graph requires a plain language explanation** that the nodes can not express
- You should add a Note if a collection of **nodes or a Group is too large or complex and can't be easily understood right away**



Return a list of translation distances based on the distance between the attractor point and each point in the grid

```
d = ((x-min)/(max-min))*b*3.14;
Math.Sin(Math.RadiansToDegrees(d))*a;
```

This code block does two things.
Line 1: Map the list of distances to a range of 0 to 1
Line 2: Pass the list of values between 0 and 1 through a math equation to return translation distances that create a Sine wave

1. A Note describing the portion of the program that returns raw translation distances
2. A Note describing the code that maps those values to a Sine wave

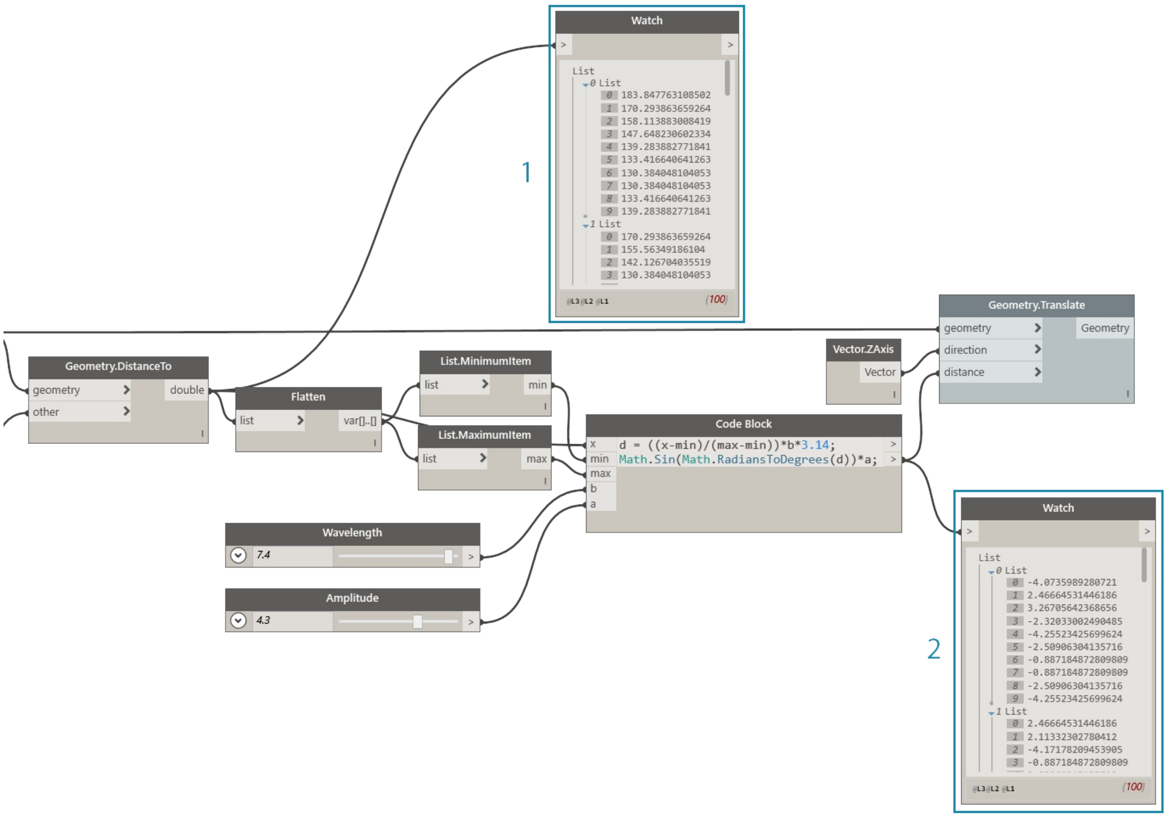
For how to add a Note, refer to Managing Your Program.

Flex Continuously

While building your visual-script, it is important to verify that what is being returned is what you expected. Not all errors or issues will cause the program to fail immediately, especially null or zero values that could affect something far downstream. This strategy is also discussed in the context of text-scripting in Scripting Strategies. The following practice will help ensure that you are getting what you expected.

Monitor data with Watch and Preview Bubbles

- Use Watch or Preview Bubbles as you build the program to **verify that key outputs are returning what you expected**



```
List
 0 List
   0 183.847763108502
   1 170.293863659264
   2 158.113883008419
   3 147.648230602334
   4 139.283882771841
   5 133.416640641263
   6 130.384048104053
   7 130.384048104053
   8 133.416640641263
   9 139.283882771841
 1 List
   0 170.293863659264
   1 155.56349186104
   2 142.126704035519
   3 130.384048104053
@L3 @L2 @L1                {100}
```

```
d = ((x-min)/(max-min))*b*3.14;
Math.Sin(Math.RadiansToDegrees(d))*a;
```

```
List
 0 List
   0 -4.0735989280721
   1 2.46664531446186
   2 3.26705642368656
   3 -2.32033002490485
   4 -4.25523425699624
   5 -2.50906304135716
   6 -0.887184872809809
   7 -0.887184872809809
   8 -2.50906304135716
   9 -4.25523425699624
 1 List
   0 2.46664531446186
   1 2.11332302780412
   2 -4.17178209453905
   3 -0.887184872809809
@L3 @L2 @L1                {100}
```

The Watch nodes are being used to compare:

1. The raw translation distances
2. The values passed through the Sine equation

For how to use Watch, refer to Library.

Ensure Reusability

It is highly likely that someone else will be opening your program at some point, even if you are working independently. They should be able to quickly understand what the program needs and produces from its inputs and outputs. This is especially important when developing a Custom Node to be shared with the Dynamo community and used in someone else's program. These practices lead to robust, reusable programs and nodes.

Manage the I/O

- To ensure legibility and scalability, you should try and **minimize inputs and outputs as much as possible**
- You should try to **strategize how you are going to build the logic by first creating a rough outline** of how the logic could work before you even add a single node to the canvas. As you develop the rough outline, you should keep track of which inputs and outputs will go into scripts

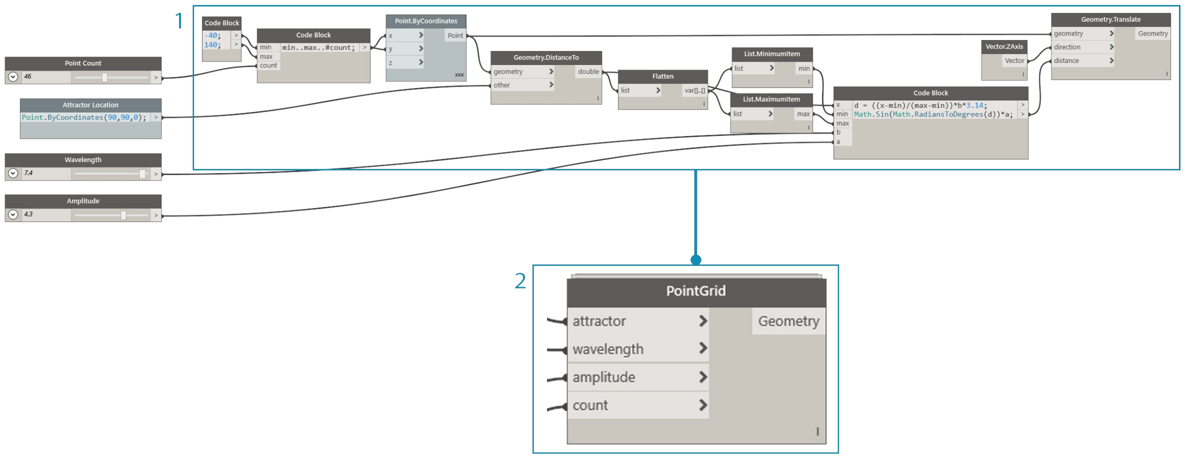
Use Presets to embed input values

- If there are **particular options or conditions that you want embedded in the graph**, you should use Presets for quick access
- You can also use Presets to **reduce complexity by caching specific slider values** in a graph with long run times

For how to use Presets, refer to Managing Your Data with Presets.

Contain programs with Custom Nodes

- You should use a Custom Node if your **program can be collected into a single container**
- You should use a a Custom Node **when a portion of the graph will be reused often** in other programs
- You should use a Custom Node if you want to **share a functionality with the Dynamo Community**



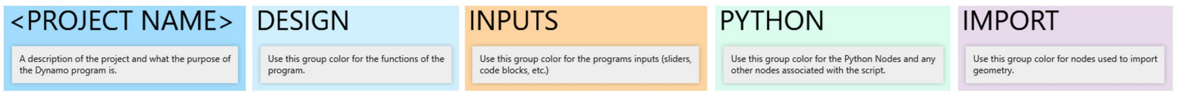
Collecting the point translation program into a Custom Node makes a robust, unique program portable and far easier to understand. Well named input ports will help other users understand how to use the node. Remember to add descriptions and required data types for each input.

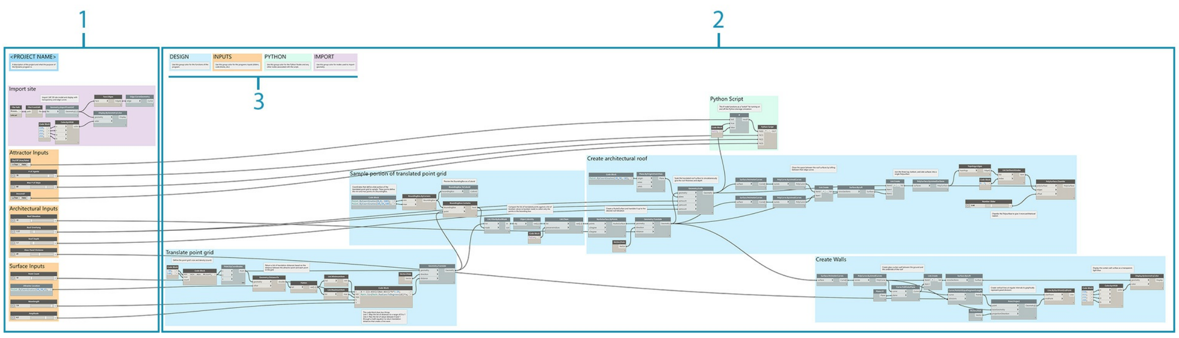
1. Existing attractor program
2. Custom Node that collects this program, PointGrid

For how to use Custom Nodes, refer to Custom Node Introduction.

Build templates

- You can build templates to **establish graphic standards across your visual graphs to ensure collaborators have a standardized way of understanding graph**
- When building a template, you can standardize **group colors and font sizes** to categorize types of workflows or data actions.
- When building a template, you can even standardize how you want to **label, color, or style the difference between front-end and back-end workflows** in your graph.


<PROJECT NAME>
A description of the project and what the purpose of the Dynamo program is.
DESIGN
Use this group color for the functions of the program.
INPUTS
Use this group color for the programs inputs (sliders, code blocks, etc.)
PYTHON
Use this group color for the Python Nodes and any other nodes associated with the script.
IMPORT
Use this group color for nodes used to import geometry.

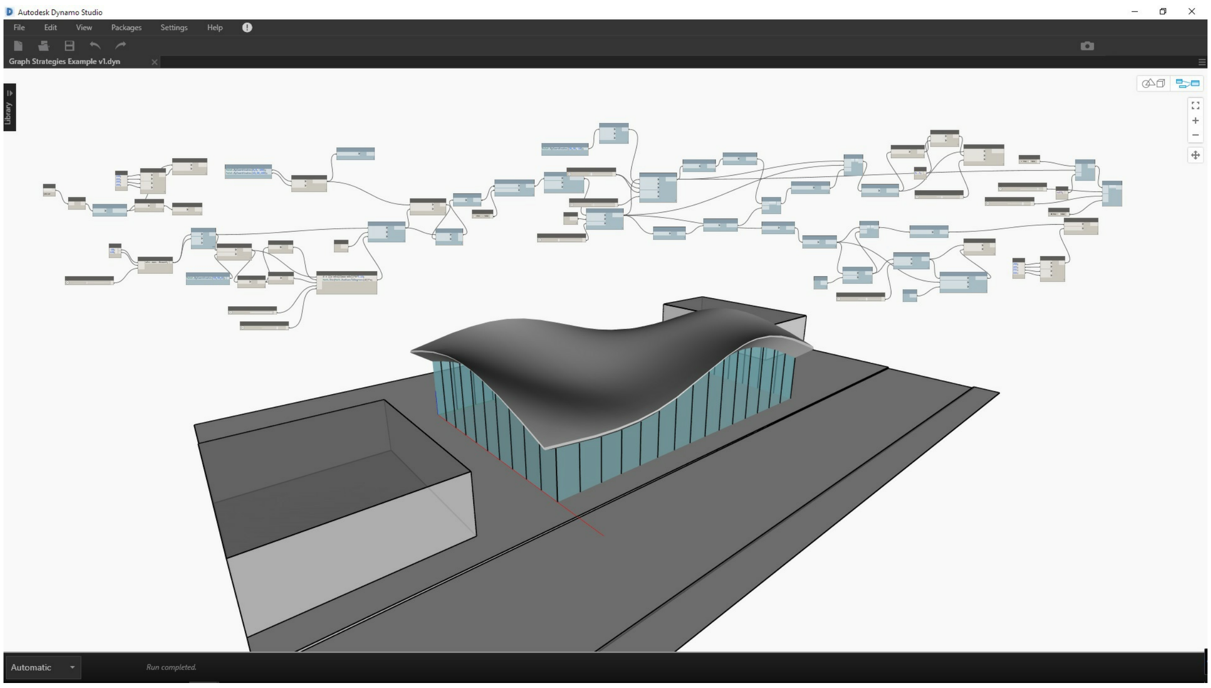


1. The UI, or front-end, of the program includes a project name, input sliders, and import geometry.
2. The back-end of the program.
3. Group color categories (the general design, inputs, Python scripting, imported geometry).

Exercise - Architectural Roof

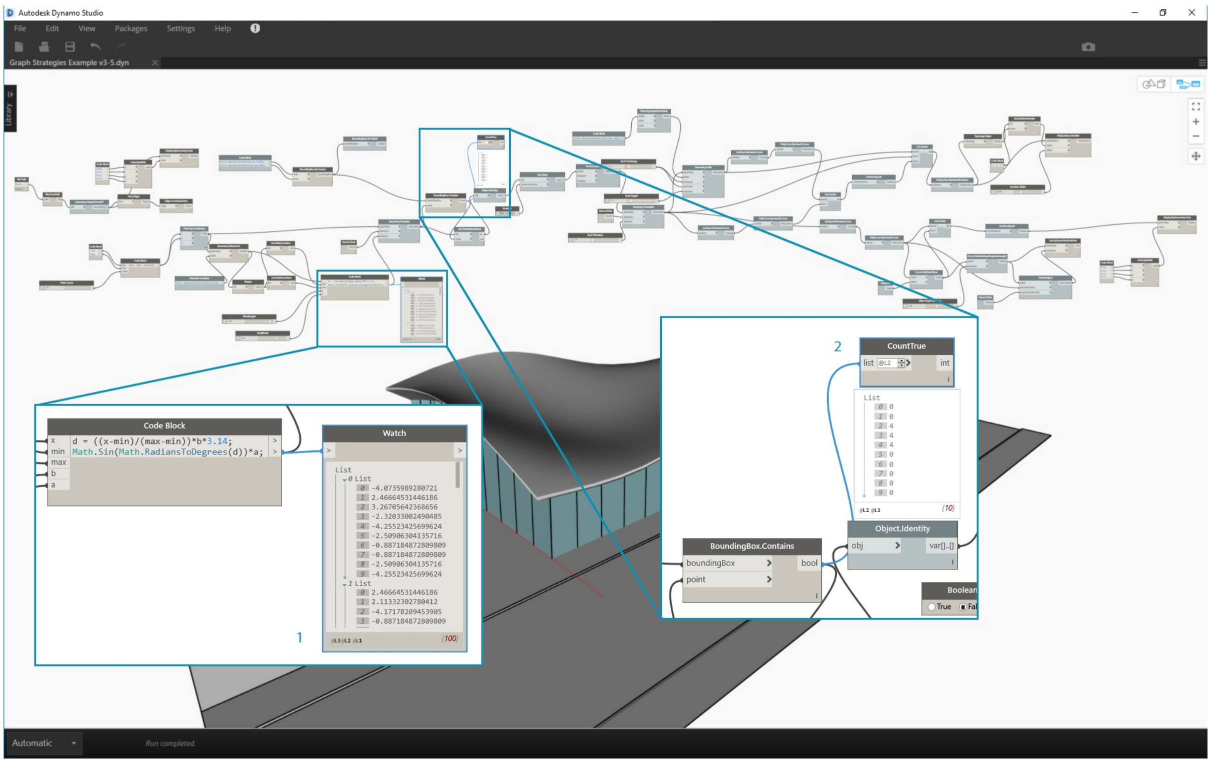
Download the example file that accompanies this exercise (Right click and "Save Link As..."). A full list of example files can be found in the Appendix. RoofDrainageSim.zip

Now that we have established several best practices, let’s apply them to a program that was put together quickly. Though the program succeeds in generating the roof, the state of the graph is a "mind-map" of the author. It lacks any organization or description of its use. We will walk through our best practices to organize, describe, and analyze the program so other users can understand how to use it.



The program is functioning, but the graph is disorganized.

Let's start by determining the data and geometry returned by the program.



```
d = ((x-min)/(max-min))*b*3.14;
Math.Sin(Math.RadiansToDegrees(d))*a;
```

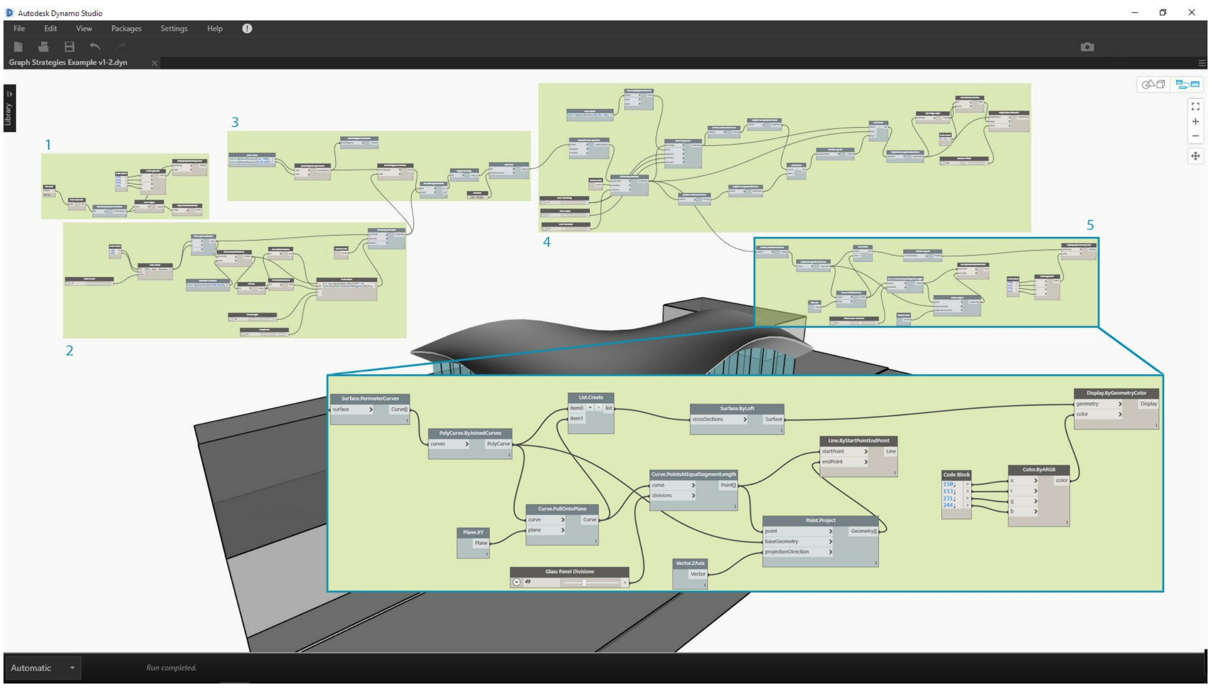
```
List
  0 List
    0 -4.0735989280721
    1 2.46664531446186
    2 3.26705642368656
    3 -2.32033002490485
    4 -4.25523425699624
    5 -2.50906304135716
    6 -0.887184872809809
    7 -0.887184872809809
    8 -2.50906304135716
    9 -4.25523425699624
  1 List
    0 2.46664531446186
    1 2.11332302780412
    2 -4.17178209453905
    3 -0.887184872809809
@L3 @L2 @L1                {100}
```

```
List
  0 0
  1 0
  2 4
  3 4
  4 4
  5 0
  6 0
  7 0
  8 0
  9 0
@L2 @L1                {10}
```

Understanding when major changes to the data occur is crucial to establishing logical divisions, or modularity. Try inspecting the rest of the program with Watch nodes to see if you can determine groups before moving on to the next step.

1. This Code Block with a math equation looks like a crucial piece of the program. A Watch node displays that it is returning lists of translation distances.
2. The purpose of this area isn't readily obvious. The arrangement of True values at list level L2 from BoundingBox.Contains and the presence of List.FilterByBoolMask suggests we are sampling a portion of the point grid.

Once we understand the elemental parts of the program, let's put them in Groups.

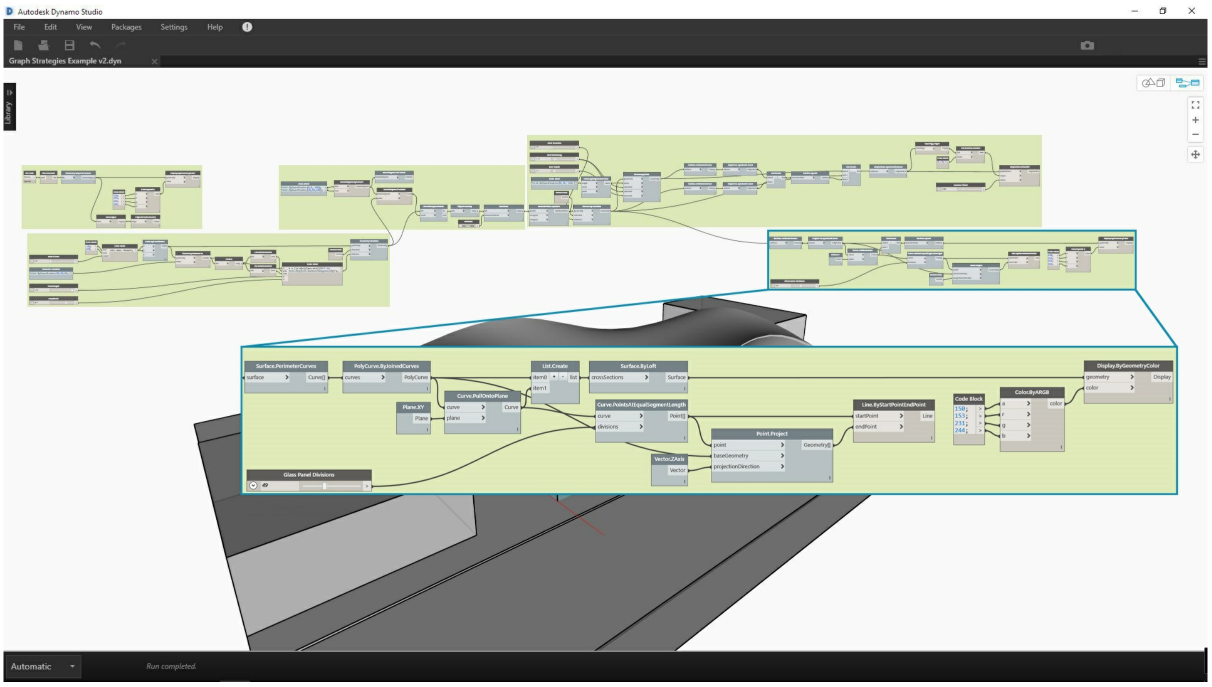


```
150;
153;
231;
244;
```

Groups allow the user to visually differentiate the parts of the program.

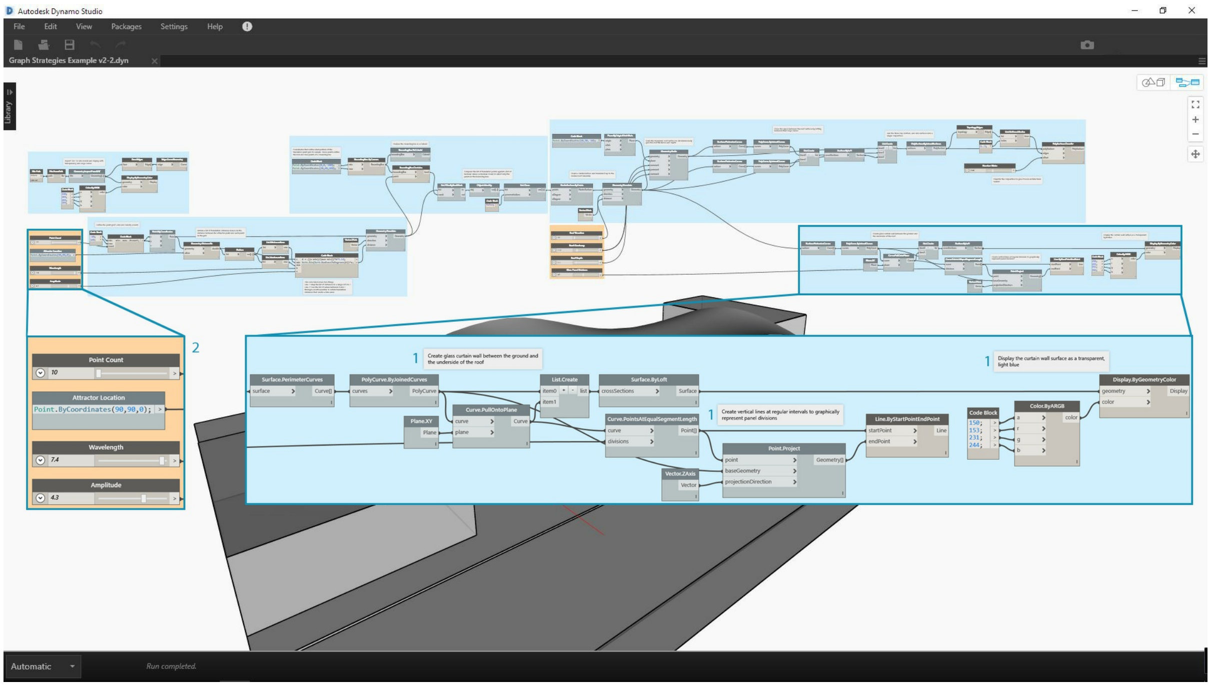
1. Import 3D site model
2. Translate point grid based on Sine equation
3. Sample portion of point grid
4. Create architectural roof surface
5. Create glass curtain wall

With Groups established, align the nodes to create visual continuity across the graph.



Visual continuity helps the user to see the program flow and implicit relationships between nodes.

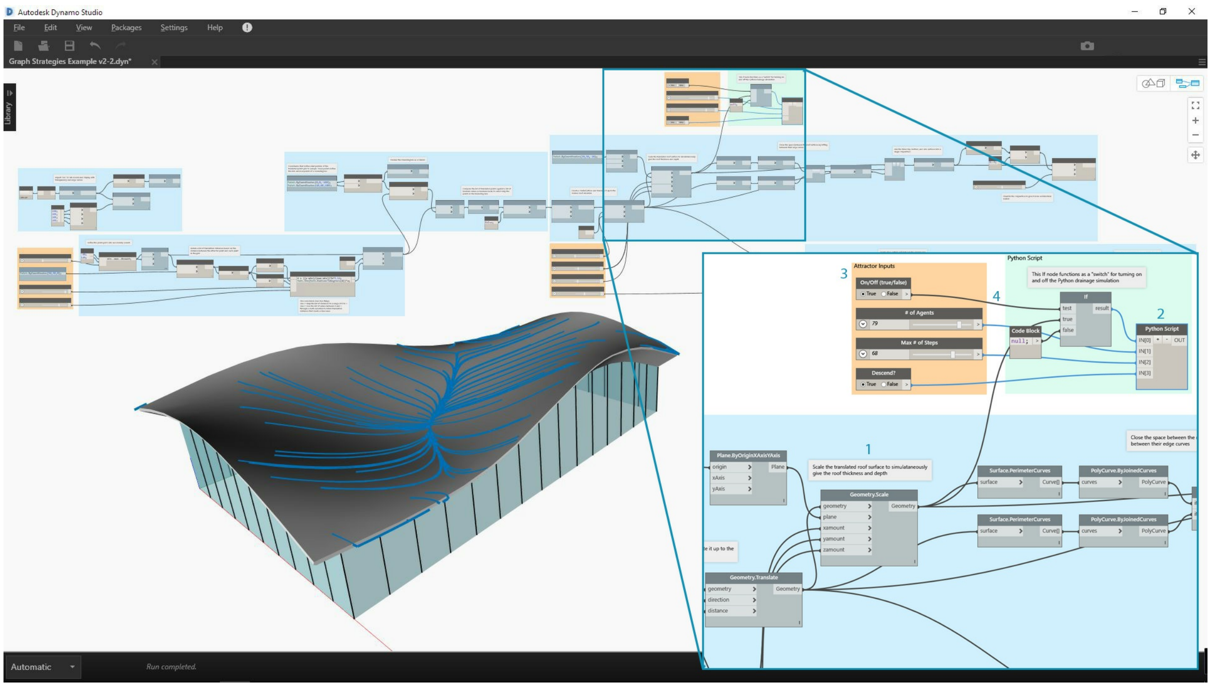
Make the program more accessible by adding another layer of graphic improvements. Add notes to describe how a specific area of the program works, give inputs custom names, and assign colors to different types of groups.


Autodesk Dynamo Studio
File
Edit
View
Packages
Settings
Help
Graph Strategies Example v2-2.dyn
Library
2
Point Count
Attractor Location
Point.ByCoordinates(90,90,0);
Wavelength
Amplitude
1
Create glass curtain wall between the ground and the underside of the roof
1
Display the curtain wall surface as a transparent, light blue
1
Create vertical lines at regular intervals to graphically represent panel divisions
Surface.PerimeterCurves
PolyCurve.ByJoinedCurves
List.Create
Surface.ByLoft
Display.ByGeometryColor
Curve.PullOntoPlane
Plane.XY
Curve.PointsAtEqualSegmentLength
Line.ByStartPointEndPoint
Code Block
Color.ByARGB
Point.Project
Vector.ZAxis
Automatic
Run completed.

These graphic improvements tell the user more about what the program is doing. The different group colors help to distinguish inputs from functions.

1. Notes
2. Inputs with descriptive names

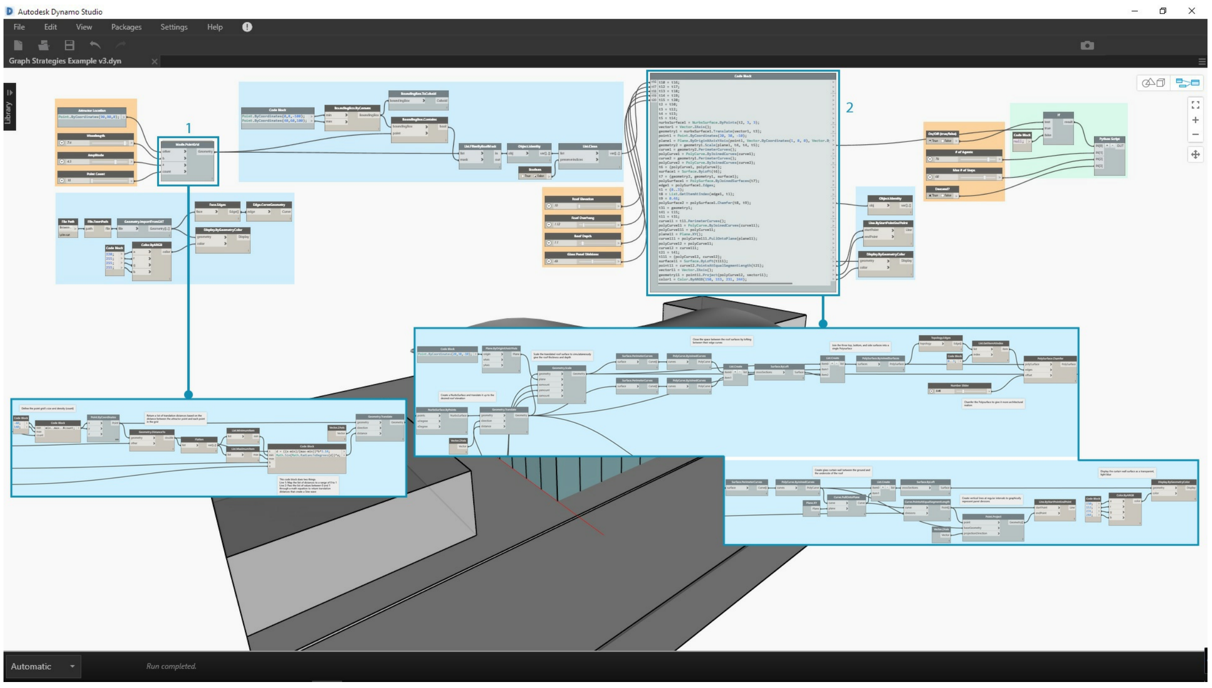
Before we start to condense the program, let's find a strategic location to introduce the Python script drainage simulator. Plug the output of the first scaled roof surface into the respective scripting input.



We've chosen to integrate scripting at this point in the program so the drainage simulation can be run on the original, single roof surface. That specific surface is not being previewed, but it saves us from having to choose the top surface of the chamfered Polysurface.

1. Source geometry for script input
2. Python node
3. Input sliders
4. On/off "switch"

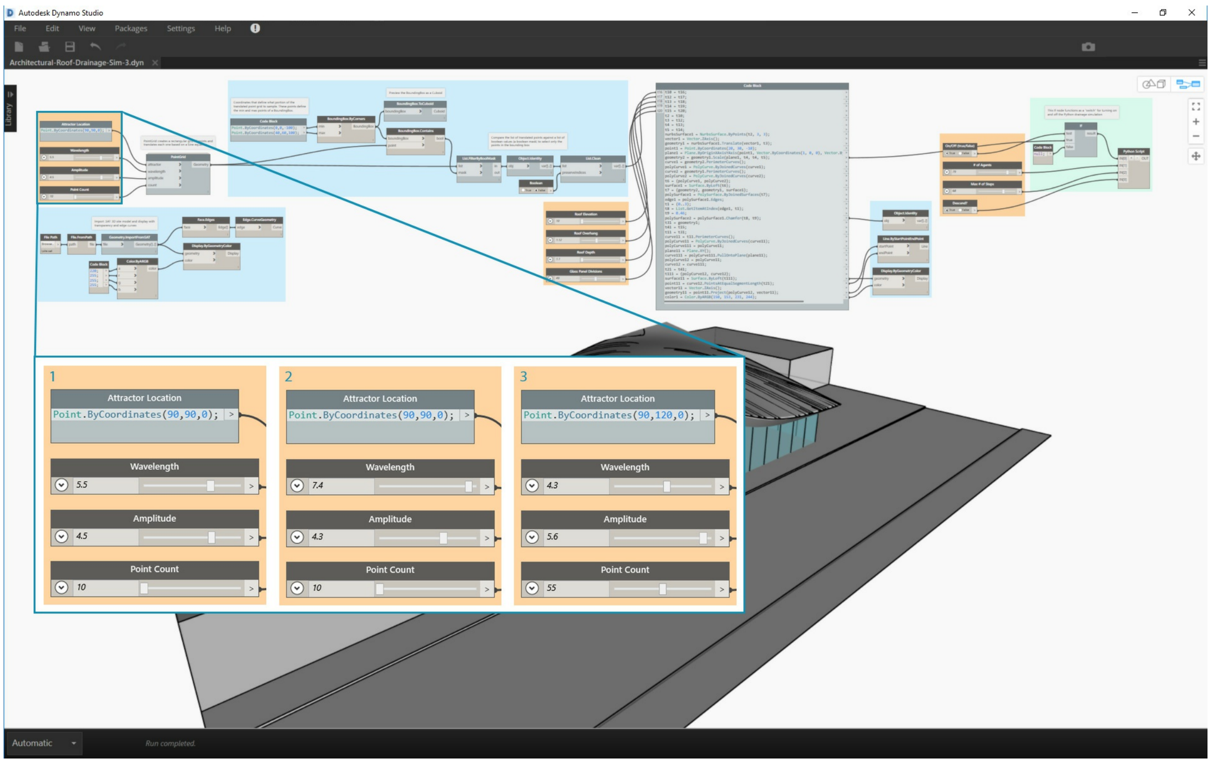
Let's simplify the graph now that everything is in place.



Condensing our program with Node to Code and Custom Node has greatly reduced the size of the graph. The groups that create the roof surface and walls have been converted to code since they are very specific to this program. The point translation group is contained in a Custom Node as it could be used in another program. In the example file, create your own custom node from the translate points group.

1. Custom Node to contain the "translate point grid" group
2. Node to Code to condense the "create architectural roof surface and curtain wall" groups

As a final step, create presets for exemplary roof forms.



1

Attractor Location
`Point.ByCoordinates(90,90,0);`
Wavelength 5.5
Amplitude 4.5
Point Count 10

2

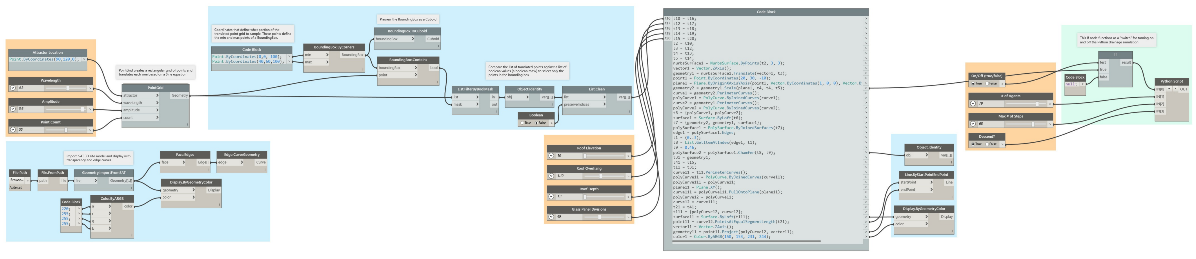
Attractor Location
`Point.ByCoordinates(90,90,0);`
Wavelength 7.4
Amplitude 4.3
Point Count 10

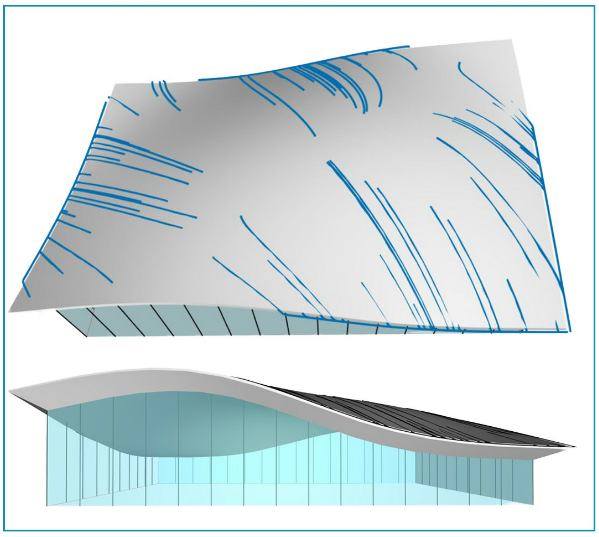
3

Attractor Location
`Point.ByCoordinates(90,120,0);`
Wavelength 4.3
Amplitude 5.6
Point Count 55

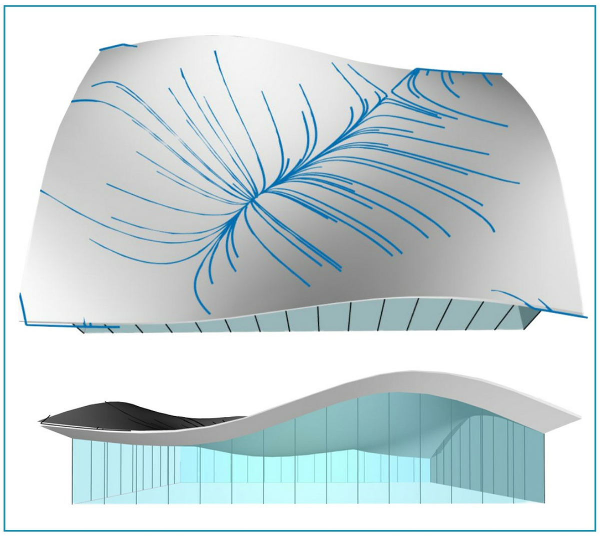
These inputs are the primary drivers of the roof form and will help users see the potential of the program.

Our program with views of two presets.





1



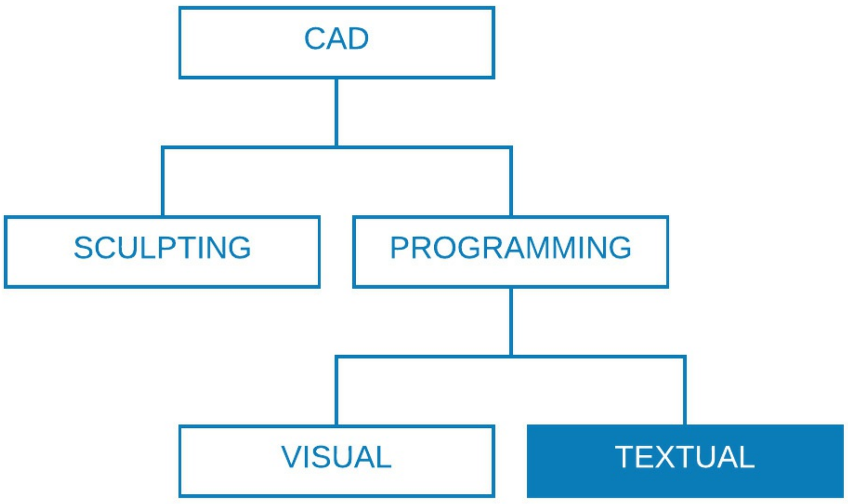
2

The roof drainage patterns give the user an analytical view of the respective presets.

Scripting Strategies

Scripting Strategies

Text-based scripting within the visual-scripting environment enables powerful and visual relationships using DesignScript, Python, and ZeroTouch (C#). The user can expose elements such as input sliders, condense large operations into DesignScript, and access powerful tools and libraries through Python or C# all within the same workspace. If managed effectively, combining these strategies can lend a great deal of customization, clarity, and efficiency to the overall program. The following are a set of guidelines to help you augment your visual-script with text-script.



Know When to Script

Text-scripting can establish relationships of a higher complexity than visual programming, yet their capabilities also overlap significantly. This makes sense because nodes are effectively pre-packaged code, and we could probably write an entire Dynamo program in DesignScript or Python. However, we use visual-scripting because the interface of nodes and wires creates an intuitive flow of graphic information. Knowing where text-scripting's capabilities go beyond visual-scripting will give you major clues to when it should be used without foregoing the intuitive nature of nodes and wires. The following are guidelines on when to script and which language to use.

Use text-scripting for:

- Looping
- Recursion
- Accessing external libraries

Choose a language:

	Looping	Recursion	Condense Nodes	Ext. Libraries	Shorthand
DesignScript	Yes	Yes	Yes	No	Yes
Python	Yes	Yes	Partially	Yes	No
ZeroTouch (C#)	No	No	No	Yes	No

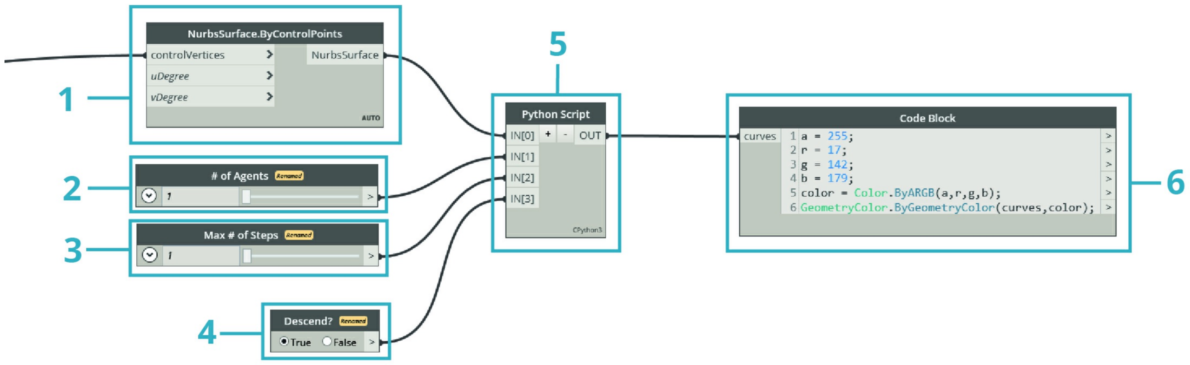
Refer to Scripting Reference for a list of what each Dynamo library gives you access to.

Think Parametrically

When scripting in Dynamo, an inevitably parametric environment, it is wise to structure your code relative to the framework of nodes and wires it will be living in. Consider the node containing your text-script as though it is any other node in the program with a few specific inputs, a function, and an expected output. This immediately gives your code inside the node a small set of variables from which to work, the key to a clean parametric system. Here are some guidelines for better integrating code into a visual program.

Identify the external variables:

- Try to determine the given parameters in your design problem so that you can construct a model that directly builds off that data.
- Before writing code, identify the variables:
    - A minimal set of inputs
    - The intended output
    - Constants



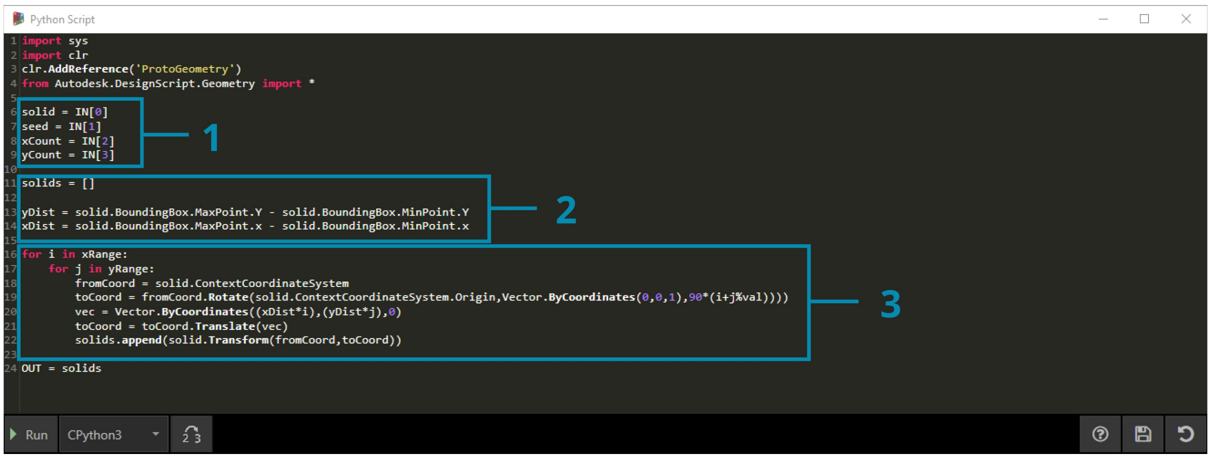
```
a = 255;
r = 17;
g = 142;
b = 179;
color = Color.ByARGB(a,r,g,b);
GeometryColor.ByGeometryColor(curves,color);
```

Several variables have been established prior to writing code.

1. The surface we will simulate rainfall on.
2. The number of rain drops (agents) we want.
3. How far we want the rain drops to travel.
4. Toggle between descending the steepest path versus traversing the surface.
5. Python Node with the respective number of inputs.
6. A Code Block to make the returned curves blue.

Design the internal relationships:

- Parametricism allows for certain parameters or variables to be edited in order to manipulate or alter the end result of an equation or system.
- Whenever entities in your script are logically related, aim to define them as functions of each other. This way when one is modified, the other can update proportionally.
- Minimize number of inputs by only exposing key parameters:
    - If a set of parameters can be derived from more parent parameters, only expose the parent parameters as script inputs. This increases the usability of your script by reducing the complexity of its interface.



```
import sys
import clr
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *

solid = IN[0]
seed = IN[1]
xCount = IN[2]
yCount = IN[3]

solids = []

yDist = solid.BoundingBox.MaxPoint.Y - solid.BoundingBox.MinPoint.Y
xDist = solid.BoundingBox.MaxPoint.x - solid.BoundingBox.MinPoint.x

for i in xRange:
    for j in yRange:
        fromCoord = solid.ContextCoordinateSystem
        toCoord = fromCoord.Rotate(solid.ContextCoordinateSystem.Origin,Vector.ByCoordinates(0,0,1),90*(i+j%val)))
        vec = Vector.ByCoordinates((xDist*i),(yDist*j),0)
        toCoord = toCoord.Translate(vec)
        solids.append(solid.Transform(fromCoord,toCoord))

OUT = solids
```

The code "modules" from the example in Python Node.

1. Inputs.
2. Variables internal to the script.
3. A loop that uses these inputs and variables to perform its function.

Tip: Place as much emphasis on the process as you do on the solution.

Don't repeat yourself (the DRY principle):

- When you have multiple ways to express the same thing in your script, at some point the duplicate representations will fall out of sync which can lead to maintenance nightmares, poor factoring, and internal contradictions.
- The DRY principle is stated as "Every piece of knowledge must have a single, unambiguous, authoritative representation within a system":
  - When this principle is successfully applied, all the related elements in your script change predictably and uniformly and all the unrelated elements do not have logical consequences on each other.

```
### BAD
for i in range(4):
  for j in range(4):
    point = Point.ByCoordinates(3*i, 3*j, 0)
    points.append(point)

### GOOD
count = IN[0]
pDist = IN[1]

for i in range(count):
  for j in range(count):
    point = Point.ByCoordinates(pDist*i, pDist*j, 0)
    points.append(point)
```

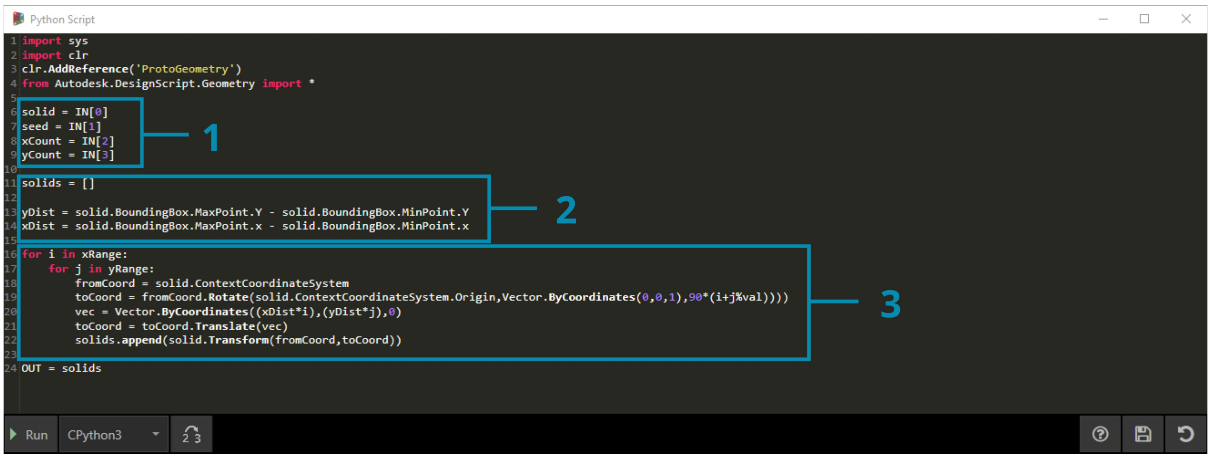
Tip: Before duplicating entities in your script (such as constant in the example above), ask yourself if you can link to the source instead.

Structure Modularly

As your code gets longer and more complex the “big idea”, or overarching algorithm becomes increasingly illegible. It also becomes more difficult to keep track of what (and where) specific things happen, find bugs when things go wrong, integrate other code, and assign development tasks. To avoid these headaches it’s wise to write code in modules, an organizational strategy that breaks up code based on the task it executes. Here are some tips for making your scripts more manageable by way of modularization.

Write code in modules:

- A "module" is a group of code that performs a specific task, similar to a Dynamo Node in the workspace.
- This can be anything that should be visually separated from adjacent code (a function, a class, a group of inputs, or the libraries you are importing).
- Developing code in modules harnesses the visual, intuitive quality of Nodes as well as the complex relationships that only text-scripting can achieve.



```
import sys
import clr
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *

solid = IN[0]
seed = IN[1]
xCount = IN[2]
yCount = IN[3]

solids = []

yDist = solid.BoundingBox.MaxPoint.Y - solid.BoundingBox.MinPoint.Y
xDist = solid.BoundingBox.MaxPoint.x - solid.BoundingBox.MinPoint.x

for i in xRange:
    for j in yRange:
        fromCoord = solid.ContextCoordinateSystem
        toCoord = fromCoord.Rotate(solid.ContextCoordinateSystem.Origin,Vector.ByCoordinates(0,0,1),90*(i+j%val)))
        vec = Vector.ByCoordinates((xDist*i),(yDist*j),0)
        toCoord = toCoord.Translate(vec)
        solids.append(solid.Transform(fromCoord,toCoord))

OUT = solids
```

These loops call a class named "agent" that we will develop in the exercise.

1. A code module that defines the start point of each agent.
2. A code module that updates the agent.
3. A code module that draws a trail for the agent's path.

Spotting code re-use:

- If you find that your code does the same (or very similar) thing in more than once place, find ways to cluster it into a function that can be called.
- "Manager" functions control program flow and primarily contain calls to "Worker" functions that handle low-level details, like moving data between structures.

This example creates spheres with radii and color based on the Z value of the center points.

1. Two "worker" parent functions: one that creates spheres with radii and display colors based the centerpoint's Z value.
2. A "manager" parent function that combines the two worker functions. Calling this will call both functions inside it.

Only show what needs to be seen:

- A module interface expresses the elements that are provided and required by the module.
- Once the interfaces between the units have been defined, the detailed design of each unit can proceed separately.

Separability/Replaceability:

- Modules don’t know or care about each other.

General forms of modularization:

- Code Grouping:

  ```
  # IMPORT LIBRARIES
  import random
  import math
  import clr
  clr.AddReference('ProtoGeometry')
  from Autodesk.DesignScript.Geometry import *

  # DEFINE PARAMETER INPUTS
  surfIn = IN[0]
  maxSteps = IN[1]
  ```

- Functions:

  ```
  def get_step_size():
    area = surfIn.Area
    stepSize = math.sqrt(area)/100
    return stepSize

  stepSize = get_step_size()
  ```

- Classes:

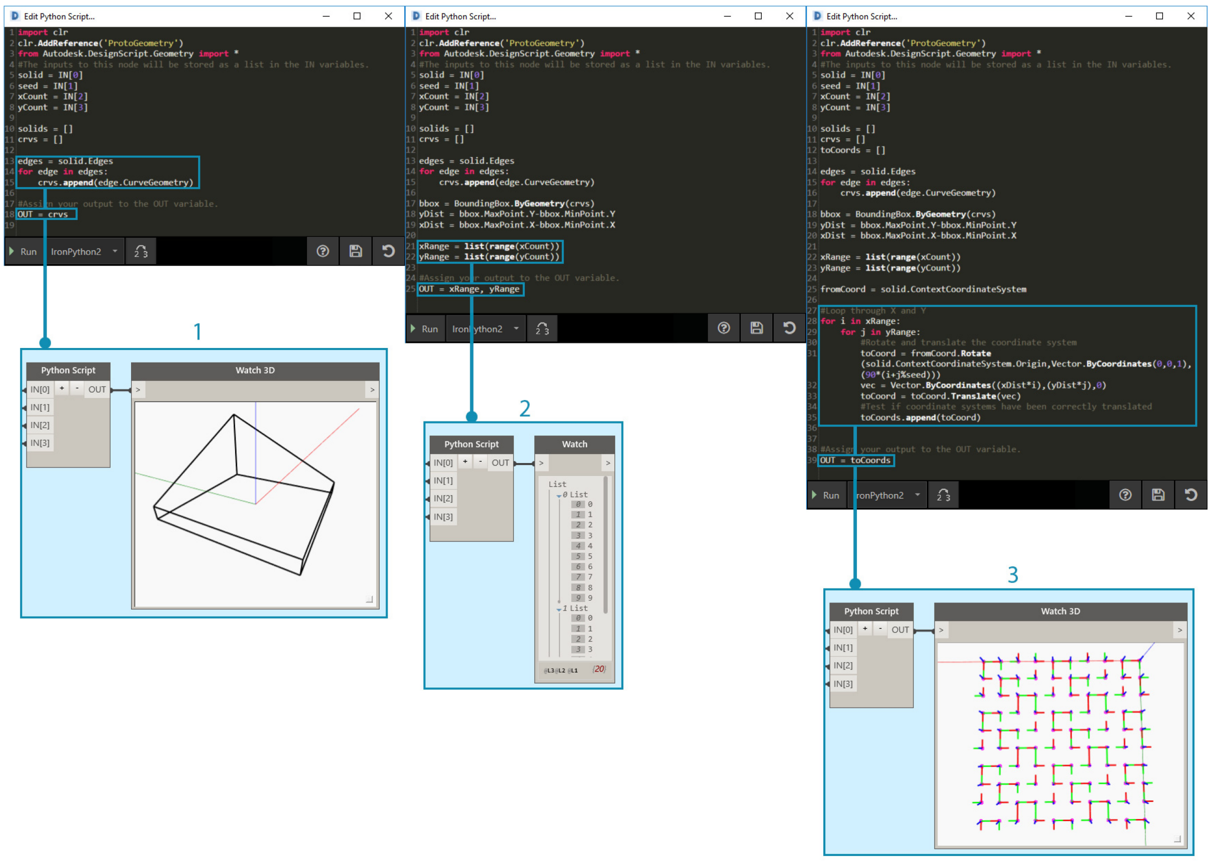
  ```
  class MyClass:
    i = 12345

    def f(self):
      return 'hello world'

  numbers = MyClass.i
  greeting = MyClass.f
  ```

Flex Continuously

While developing text-scripts in Dynamo, it is wise to constantly make sure that what is actually being created is in line with what you are expecting. This will ensure that unforeseen events-- syntax errors, logical discrepancies, value inaccuracies, anomalous outputs etc.-- are quickly discovered and dealt with as they surface rather than all at once at the end. Because text-scripts live inside nodes on the canvas, they are already integrated into the data flow of your visual program. This makes the successive monitoring of your script as simple as assigning data to be outputted, running the program, and evaluating what flows out of the script using a Watch Node. Here are some tips for continuously inspecting your scripts as you construct them.

Test as you go:

- Whenever you complete a cluster of functionality:
  - Step back and inspect your code.
  - Be critical. Could a collaborator understand what this is doing? Do I need to do this? Can this function be done more efficiently? Am I creating unnecessary duplicates or dependencies?
  - Quickly test to make sure it is returning data that “makes sense”.
- Assign the most recent data you are working with in your script as the output so that the node is always outputting relevant data when the script updates:



```
import clr
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *
#The inputs to this node will be stored as a list in the IN variables.
solid = IN[0]
seed = IN[1]
xCount = IN[2]
yCount = IN[3]

solids = []
crvs = []

edges = solid.Edges
for edge in edges:
    crvs.append(edge.CurveGeometry)

#Assign your output to the OUT variable.
OUT = crvs
```

```
import clr
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *
#The inputs to this node will be stored as a list in the IN variables.
solid = IN[0]
seed = IN[1]
xCount = IN[2]
yCount = IN[3]

solids = []
crvs = []

edges = solid.Edges
for edge in edges:
    crvs.append(edge.CurveGeometry)

bbox = BoundingBox.ByGeometry(crvs)
yDist = bbox.MaxPoint.Y-bbox.MinPoint.Y
xDist = bbox.MaxPoint.X-bbox.MinPoint.X

xRange = list(range(xCount))
yRange = list(range(yCount))

#Assign your output to the OUT variable.
OUT = xRange, yRange
```

```
List
 0 List
   0 0
   1 1
   2 2
   3 3
   4 4
   5 5
   6 6
   7 7
   8 8
   9 9
 1 List
   0 0
   1 1
   2 2
   3 3
@L3 @L2 @L1   {20}
```

```
import clr
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *
#The inputs to this node will be stored as a list in the IN variables.
solid = IN[0]
seed = IN[1]
xCount = IN[2]
yCount = IN[3]

solids = []
crvs = []
toCoords = []

edges = solid.Edges
for edge in edges:
    crvs.append(edge.CurveGeometry)

bbox = BoundingBox.ByGeometry(crvs)
yDist = bbox.MaxPoint.Y-bbox.MinPoint.Y
xDist = bbox.MaxPoint.X-bbox.MinPoint.X

xRange = list(range(xCount))
yRange = list(range(yCount))

fromCoord = solid.ContextCoordinateSystem

#Loop through X and Y
for i in xRange:
    for j in yRange:
        #Rotate and translate the coordinate system
        toCoord = fromCoord.Rotate
        (solid.ContextCoordinateSystem.Origin,Vector.ByCoordinates(0,0,1),
        (90*(i+j%seed)))
        vec = Vector.ByCoordinates((xDist*i),(yDist*j),0)
        toCoord = toCoord.Translate(vec)
        #Test if coordinate systems have been correctly translated
        toCoords.append(toCoord)


#Assign your output to the OUT variable.
OUT = toCoords
```

Flexing the example code from Python Node.

1. Check that all edges of the solid are being returned as curves to create a bounding box around.
2. Check that our Count inputs are successfully being converted to Ranges.
3. Check that coordinate systems have been properly translated and rotated in this loop.

Anticipate “edge cases”:

- While scripting, crank your input parameters to the minimum and maximum values of their allotted domain to check if the program still functions under extreme conditions.
- Even if the program is functioning at its extremes, check if it is returning unintended null/empty/zero values.
- Sometimes bugs and errors that reveal some underlying problem with your script will only surface during these edge cases.
    - Understand what is causing the error and then decide if it needs to be fixed internally or if a parameter domain needs to be redefined to avoid the problem.

Tip: Always assume the that the user will use every combination of every input value that has been exposed to him/her. This will help eliminate unwanted surprises.

Debug Efficiently

Debugging is the process of eliminating "bugs" from your script. Bugs can be errors, inefficiencies, inaccuracies, or any unintended results. Addressing a bug can be as simple as correcting a misspelled variable name to more pervasive, structural problems with your script. Ideally, flexing your script as you build it will help to catch these potential issues early, though this is no guarantee of it being bug-free. The following is a review of several best practices from above to help you address bugs systematically.

Use the watch bubble:

- Check the data returned at different places in the code by assigning it to the OUT variable, similar to the concept of flexing the program.

Write meaningful comments:

- A module of code will be much easier to debug if its intended outcome is clearly described.

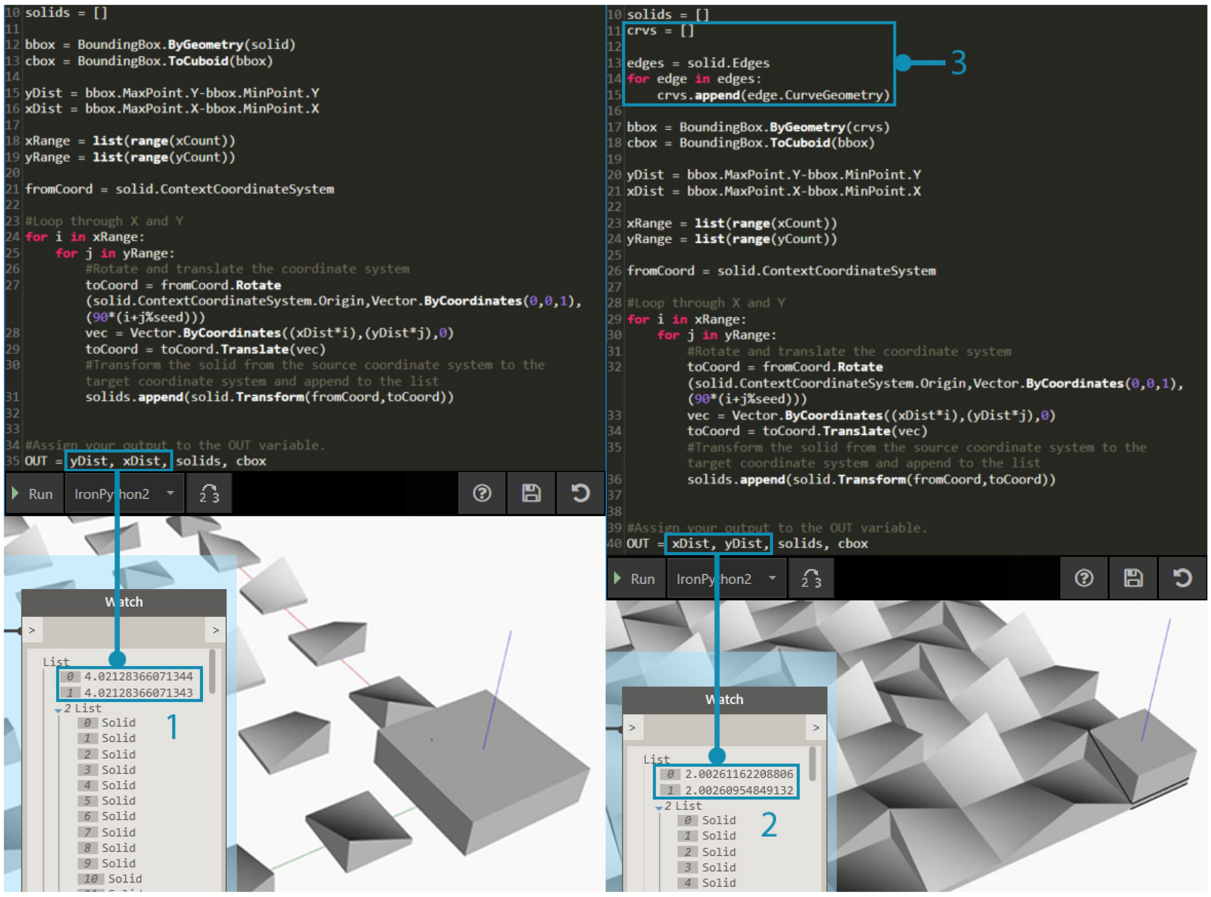
```
# Loop through X and Y
for i in range(xCount):
  for j in range(yCount):

    # Rotate and translate the coordinate system
    toCoord = fromCoord.Rotate(solid.ContextCoordinateSystem.Origin,Vector.ByCoordinates(0,0,1),(90*(i+j%
    vec = Vector.ByCoordinates((xDist*i),(yDist*j),0)
    toCoord = toCoord.Translate(vec)
```

```
# Transform the solid from the source coord system to the target coord system and append to the list
solids.append(solid.Transform(fromCoord,toCoord))
```

Normally this would be an excessive amount of commenting and blank lines, but when debugging it can be useful to break things down into manageable pieces.

Leverage the code's modularity:

- The source of an issue can be isolated to certain modules.
- Once the faulty module has been identified, fixing the problem is considerably simpler.
- When a program must be modified, code that has been developed in modules will be much easier to change:
    - You can insert new or debugged modules into an existing program with the confidence that the rest of the program will not change.



```
solids = []

bbox = BoundingBox.ByGeometry(solid)
cbox = BoundingBox.ToCuboid(bbox)

yDist = bbox.MaxPoint.Y-bbox.MinPoint.Y
xDist = bbox.MaxPoint.X-bbox.MinPoint.X

xRange = list(range(xCount))
yRange = list(range(yCount))

fromCoord = solid.ContextCoordinateSystem

#Loop through X and Y
for i in xRange:
    for j in yRange:
        #Rotate and translate the coordinate system
        toCoord = fromCoord.Rotate
        (solid.ContextCoordinateSystem.Origin,Vector.ByCoordinates(0,0,1),
        (90*(i+j%seed)))
        vec = Vector.ByCoordinates((xDist*i),(yDist*j),0)
        toCoord = toCoord.Translate(vec)
        #Transform the solid from the source coordinate system to the
        target coordinate system and append to the list
        solids.append(solid.Transform(fromCoord,toCoord))


#Assign your output to the OUT variable.
OUT = yDist, xDist, solids, cbox
```

```
List
  0 4.02128366071344
  1 4.02128366071343
  2 List
    0 Solid
    1 Solid
    2 Solid
    3 Solid
    4 Solid
    5 Solid
    6 Solid
    7 Solid
    8 Solid
    9 Solid
    10 Solid
```

```
solids = []
crvs = []

edges = solid.Edges
for edge in edges:
    crvs.append(edge.CurveGeometry)

bbox = BoundingBox.ByGeometry(crvs)
cbox = BoundingBox.ToCuboid(bbox)

yDist = bbox.MaxPoint.Y-bbox.MinPoint.Y
xDist = bbox.MaxPoint.X-bbox.MinPoint.X

xRange = list(range(xCount))
yRange = list(range(yCount))

fromCoord = solid.ContextCoordinateSystem

#Loop through X and Y
for i in xRange:
    for j in yRange:
        #Rotate and translate the coordinate system
        toCoord = fromCoord.Rotate
        (solid.ContextCoordinateSystem.Origin,Vector.ByCoordinates(0,0,1),
        (90*(i+j%seed)))
        vec = Vector.ByCoordinates((xDist*i),(yDist*j),0)
        toCoord = toCoord.Translate(vec)
        #Transform the solid from the source coordinate system to the
        target coordinate system and append to the list
        solids.append(solid.Transform(fromCoord,toCoord))


#Assign your output to the OUT variable.
OUT = xDist, yDist, solids, cbox
```

```
List
  0 2.00261162208806
  1 2.00260954849132
  2 List
    0 Solid
    1 Solid
    2 Solid
    3 Solid
    4 Solid
```

Debugging the example file from Python Node.

1. The input geometry is returning a bounding box larger that itself, as we can see from assigning xDist and yDist to OUT.
2. The edge curves of the input geometry return an appropriate bounding box with correct distances for xDist and yDist.
3. The code "module" we've inserted to address the xDist and yDist value issue.

Exercise - Steepest Path

Download the example file that accompanies this exercise (Right click and "Save Link As..."). A full list of example files can be found in the Appendix. SteepestPath.dyn

With our best practices for text-scripting in mind, let's write a rain simulation script. While we were able to apply best practices to a disorganized visual program in Graph Strategies, it is far more difficult to do that with text-scripting. Logical relationships established in text-scripting are less visible and can be almost impossible to untangle in messy code. With the power of text-scripting comes a larger responsibility in organization. We will walk through each step and apply best practices along the way.

Our script applied to an attractor-deformed surface.

The first thing we need to do is import the necessary Dynamo libraries. Doing this first will give global access to Dynamo functionality in Python.

All the libraries we intend on using need to be imported here.

Next we need to define the script's inputs and output, which will display as input ports on the node. These external inputs are the foundation for our script and the key to establishing a parametric environment.

We need to define inputs that correspond to variables in the Python script and determine a desired output:

1. The surface we want to walk down.
2. The number of agents we want to walk.
3. The maximum number of steps the agents are allowed to take.
4. An option to take the shortest path down the surface or traverse it.
5. The Python Node with input identifiers that correspond to inputs in the script (IN[0], IN[1]).
6. Output curves that can be displayed with a different color.

Now let's employ the practice of modularity and create the body of our script. Simulating the shortest path down a surface for multiple start points is a significant task that will require several functions. Rather than call the different functions throughout the script, we can modularize our code by collecting them into a single class, our agent. The different functions of this class or "module" can be called with different variables or even reused in another script.

We will need to define a class, or blueprint, for an agent with the intention of walking down a surface by choosing to travel in the steepest possible direction each time it takes a step:

1. Name.
2. Global attributes that all the agents share.
3. Instance attributes that are unique to each agent.
4. A function for taking a step.
5. A function for cataloging the position of each step to a trail list.

Let's initialize the agents by defining their start location. This is a good opportunity to flex our script and make sure the agent class is working.

We will need to instantiate all the agents we want to observe walk down the surface and define their initial attributes:

1. A new empty trail list.
2. Where they will start their journey on the surface.
3. We've assigned the agents list as the output to check what the script is returning here. The correct number of agents is being returned, but we'll need to flex the script again later on to verify the geometry it returns.

Update each agent at each step.

We will then need to enter a nested loop where for each agent and for each step, we update and record their position into their trail list. At each step we will also make sure the agent hasn’t reached a point on the surface where it cannot take another step which will allow it to descend. If that condition is met, we will end that agent's trip.

Now that our agents have been fully updated, let's return geometry that represents them.

After all the agents have either reached their limit of descent or their maximum number of steps we will create a polycurve through the points in their trail list and output the polycurve trails.

Our script for finding the steepest paths.

1. A preset that simulates rainfall on the underlying surface.
2. Rather than finding the steepest path, the agents can be toggled to traverse the underlying surface.

The full Python text-script.

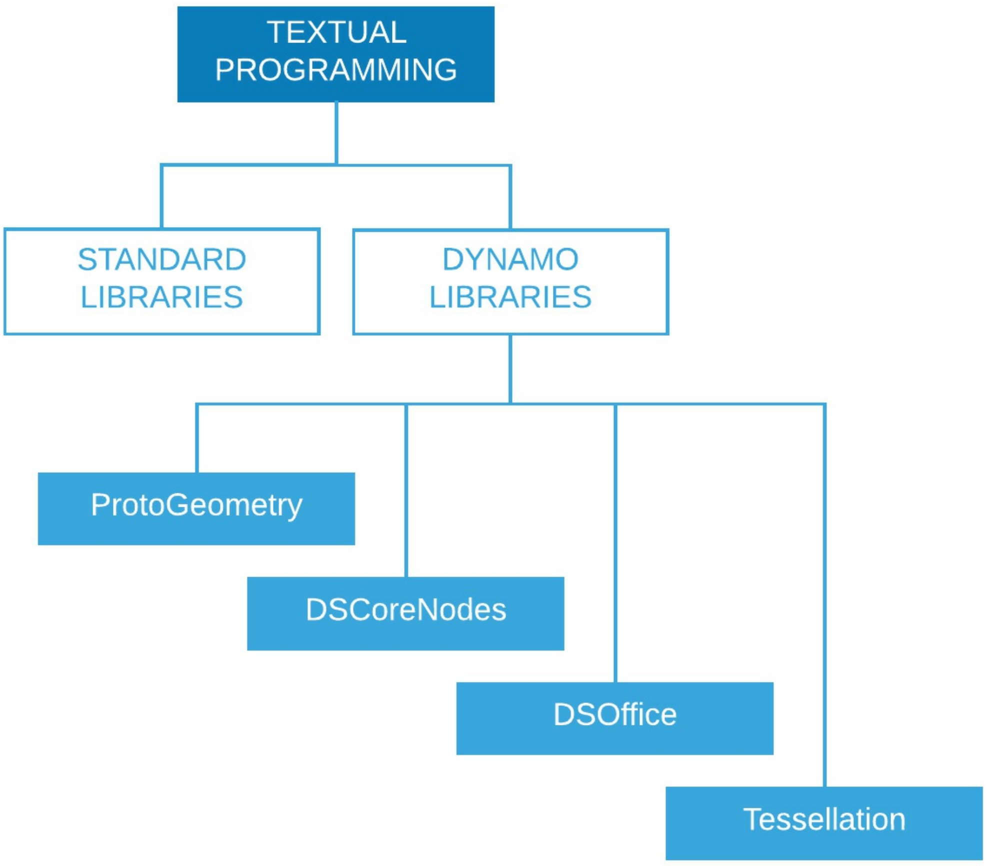
Scripting Reference

Scripting Reference

This reference page extends the best practices covered in Scripting Strategies with greater detail on code libraries, labeling, and styling. We will be using Python to illustrate the concepts below, but the same principles would apply in Python and C#(Zerotouch) but in different syntax.

Which Libraries to Use

Standard libraries are external to Dynamo and are present in the programming languages Python and C# (Zerotouch). Dynamo also has its own set of libraries that directly correspond to it's node hierarchy, enabling the user to build anything in code that could be made with nodes and wires. The following is a guide for what each Dynamo library gives access to and when to use a standard one.



Standard Libraries and Dynamo Libraries

- Standard libraries from Python and C# can be used to build advanced data and flow structures in the Dynamo environment.
- Dynamo libraries directly correspond to the node hierarchy for creating geometry and other Dynamo objects.

Dynamo Libraries

- ProtoGeometry
  - Functionality: Arc, Bounding Box, Circle, Cone, Coordinate System, Cuboid, Curve, Cylinder, Edge, Ellipse, Ellipse Arc ,Face, Geometry, Helix, Index Group, Line, Mesh, Nurbs Curve, Nurbs Surface, Plane, Point, Polygon, Rectangle, Solid, Sphere, Surface, Topology, TSpline, UV, Vector, Vertex.
  - How to import: `import Autodesk.DesignScript.Geometry`
  - **Note when using ProtoGeometry through Python or C#**, you are creating unmanaged objects, which need have their memory managed manually - please see section below: **Unmanaged Objects**, for more info.
- DSCoreNodes
  - Functionality: Color, Color Range 2D, Date Time, Time Span, IO, Formula, Logic, List, Math, Quadtree, String, Thread.
  - How to import: `import DSCore`
- Tessellation
  - Functionality: Convex Hull, Delaunay, Voronoi.
  - How to import: `import Tessellation`
- DSOffice
  - Functionality: Excel.

- How to import: `import DSOffice`

Label Carefully

While scripting, we are constantly using identifiers to denote things like variables, types, functions, and other entities. Through this system of symbolic notation, while building algorithms we can conveniently refer to information by way of labels --usually made up of a sequence of characters. Naming things well plays a significant role in writing code that can be easily read and understood by others as well as your future self! Here are some tips to keep in mind while naming things in your script:

It´s OK to use abbreviations, but explain the abbreviation with a comment:

```
### BAD
csfX = 1.6
csfY= 1.3
csfZ = 1.0
```

```
### GOOD
# column scale factor (csf)
csfX = 1.6
csfY= 1.3
csfZ = 1.0
```

Avoid redundant labeling:

```
### BAD
import car
seat = car.CarSeat()
tire = car.CarTire()
```

```
### GOOD
import car
seat = car.Seat()
tire = car.Tire()
```

Use positive logic for your variable names instead of negative logic:

```
### BAD
if 'mystring' not in text:
    print 'not found'
else:
    print 'found'
    print 'processing'
```

```
### GOOD
if 'mystring' in text:
    print 'found'
    print 'processing'
else:
    print 'not found'
```

Prefer “reverse notation”:

```
### BAD
agents = …
active_agents = …
dead_agents ...
```

```
### GOOD
agents = …
agents_active = …
agents_dead = ...
```

It’s more sensible, in structural terms.

Aliases should be used to shorten overly long and often repeated chains:

```
### BAD
from RevitServices.Persistence import DocumentManager

DocumentManager = DM

doc = DM.Instance.CurrentDBDocument
uiapp = DM.Instance.CurrentUIApplication
```

```
### GOOD
from RevitServices.Persistence import DocumentManager as DM

doc = DM.Instance.CurrentDBDocument
uiapp = DM.Instance.CurrentUIApplication
```

Aliasing can quickly lead to very confusing and non-standard programs.

Only use necessary words:

```
### BAD
```

```
rotateToCoord = rotateFromCoord.Rotate(solid.ContextCoordinateSystem.Origin,Vector.ByCoordinates(0,0,1),5

### GOOD
toCoord = fromCoord.Rotate(solid.ContextCoordinateSystem.Origin,Vector.ByCoordinates(0,0,1),5)
```

“Everything should be made as simple as possible, but not simpler.” – Albert Einstein

Style Consistently

Generally speaking there is more than one way to program just about anything, therefore your “personal style” of scripting is the result of the countless small decisions you choose to make (or not make) along the way. That said, the readability and maintainability of your code is a direct result of its internal consistency as well as its adherence to general stylistic conventions. As a rule of thumb, code that looks the same in two places should work the same, too. Here are a few tips for writing clear and consistent code.

Naming conventions: (Choose one of the conventions below for each type of entity in your code and stick to it!)

- Variables, functions, methods, packages, modules:
  `lower_case_with_underscores`
- Classes and Exceptions:
  `CapWords`
- Protected methods and internal functions:
  `_single_leading_underscore(self, ...)`
- Private methods:
  `__double_leading_underscore(self, ...)`
- Constants:
  `ALL_CAPS_WITH_UNDERSCORES`

Tip: Avoid one-letter variables (esp. l, O, I) except in very short blocks, when the meaning is clearly visible from the immediate context.

Use of blank lines:

- Surround top-level function and class definitions with two blank lines.
  - Method definitions inside a class are surrounded by a single blank line.
  - Extra blank lines may be used (sparingly) to separate groups of related functions.

Avoid extraneous whitespace:

- Immediately inside parentheses, brackets or braces:

  ```
  ### BAD
  function( apples[ 1 ], { oranges: 2 } )

  ### GOOD:
  function(apples[1], {oranges: 2})
  ```

- Immediately before a comma, semicolon, or colon:

  ```
  ### BAD
   if x == 2 : print x , y ; x , y = y , x

  ### GOOD
    if x == 2: print x, y; x, y = y, x
  ```

- Immediately before the open parenthesis that starts the argument list of a function call:

  ```
  ### BAD
  function (1)

  ### GOOD
  function(1)
  ```

- Immediately before the open parenthesis that starts an indexing or slicing:

  ```
  ### BAD
  dict ['key'] = list [index]

  ### GOOD
  dict['key'] = list[index]
  ```

- Always surround these binary operators with a single space on either side:

  ```
  assignment ( = )
  augmented assignment ( += , -= etc.)
  comparisons ( == , < , > , != , <> , <= , >= , in , not in , is , is not )
  Booleans ( and , or , not )
  ```

Watch line length:

- Don't stress over it ~ 79 characters.
- Limiting the required editor window width makes it possible to have several files open side-by-side, and works well when using code review tools

that present the two versions in adjacent columns.

- Long lines can be broken over multiple lines by wrapping expressions in parentheses:

Avoid obvious and redundant comments:

- Sometimes fewer comments makes for more readable code. Especially if it forces you to use meaningful symbol names instead.
- Adopting good coding habits reduces dependence on comments:

```
### BAD
  # get the country code
  country_code = get_country_code(address)

  # if country code is US
  if (country_code == 'US'):
    # display the form input for state
    print form_input_state()

### GOOD
  # display state selection for US users
  country_code = get_country_code(address)
  if (country_code == 'US'):
    print form_input_state()
```

Tip: Comments tell you why, Code tells you how.

Check out open source code:

- Open Source projects are built on the collaborative efforts of many developers. These projects need to maintain a high level of code readability so that the team can work together as efficiently as possible. Therefore, it is a good idea to browse through the source code of these projects to observe what these developers are doing.
- Improve your conventions:
  - Question whether or not each convention is working for the needs at hand.
  - Is functionality/efficiency being compromised?

C# (Zerotouch) Standards

Check out these wiki pages for guidance on writing C# for Zerotouch and contributing to Dynamo:

- This wiki covers some general coding standards for documenting and testing your code: https://github.com/DynamoDS/Dynamo/wiki/Coding-Standards
- This wiki specifically covers naming standards for libraries, categories, node names, port names, and abbreviations: https://github.com/DynamoDS/Dynamo/wiki/Naming-Standards

  Unmanaged Objects:

  When using Dynamo's Geometry library *(ProtoGeometry)* from Python or C# geometry objects that you create will not be managed by the virtual machine, and the memory of many of these objects will need to be cleaned up manually. To cleanup native or unmanaged objects you can use the **Dispose** method or the **using** keyword. See this wiki entry for an overview: https://github.com/DynamoDS/Dynamo/wiki/Zero-Touch-Plugin-Development#dispose--using-statement.

  You only need to dispose unmanaged resources that you don't return into the graph or store a reference to. For the rest of this section, we'll refer to these objects as *intermediate geometry*. You can see an example of this class of object in the code example below. This zero touch C# function **singleCube** returns a single cube, but creates 10000 extra cubes during its execution. We can pretend this other geometry was used as some intermediate construction geometry.

  This zero touch function will most likely crash Dynamo. Since we created 10000 solids, but only stored one of them, and only returned that one. We should instead, dispose all of our intermediate cubes, except the one that we return. We don't want to dipose what we return, as it will be propogated into the graph and used by other nodes.

```
public Cuboid singleCube(){

var output = Cuboid.ByLengths(1,1,1);

for(int i = 0; i<10000;i++){
  output = Cuboid.ByLengths(1,1,1);
}
return output;
}
```

The fixed code would look something like:

```
public Cuboid singleCube(){

  var output = Cuboid.ByLengths(1,1,1);
  var toDispose = new List<Geometry>();

  for(int i = 0; i<10000;i++){
    toDispose.Add(Cuboid.ByLengths(1,1,1));
  }
```

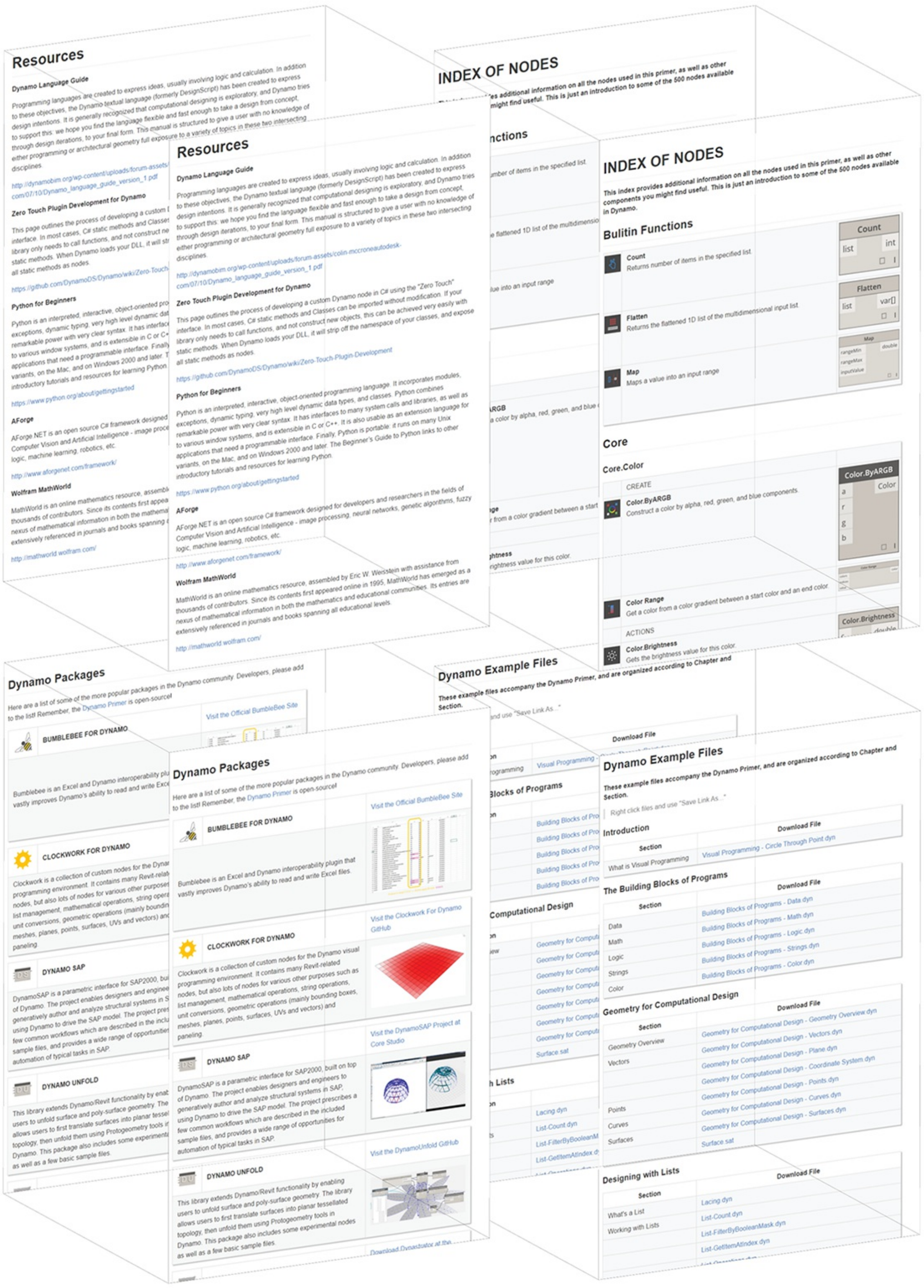
```
    foreach(IDisposable item in toDispose ){
      item.Dispose();
    }

    return output;
  }
```

In general you only need to dispose geometry like `Surfaces`, `Curves`, and `Solids`. To be safe though, you can dispose all geometry types (`Vectors`, `Points`, `CoordinateSystems`).

Appendix

Appendix A: Resources

In this section, you can find additional resources for taking your Dynamo game one step further. We've also added an index of important nodes, a collection of useful packages, and a repository of the example files in this primer. Please feel free to add to this section...remember, the Dynamo Primer is open source!



Resources

Dynamo Language Guide

Programming languages are created to express ideas, usually involving logic and calculation. In addition to these objectives, the Dynamo textual language (formerly DesignScript) has been created to express design intentions. It is generally recognized that computational designing is exploratory, and Dynamo tries to support this: we hope you find the language flexible and fast enough to take a design from concept, through design iterations, to your final form. This manual is structured to give a user with no knowledge of either programming or architectural geometry full exposure to a variety of topics in these two intersecting disciplines.

http://dynamobim.org/wp-content/uploads/forum-assets/colin-mccroneautodesk-com/07/10/Dynamo_language_guide_version_1.pdf

Zero Touch Plugin Development for Dynamo

This page outlines the process of developing a custom Dynamo node in C# using the "Zero Touch" interface. In most cases, C# static methods and Classes can be imported without modification. If your library only needs to call functions, and not construct new objects, this can be achieved very easily with static methods. When Dynamo loads your DLL, it will strip off the namespace of your classes, and expose all static methods as nodes.

https://github.com/DynamoDS/Dynamo/wiki/Zero-Touch-Plugin-Development

Python for Beginners

Python is an interpreted, interactive, object-oriented programming language. It incorporates modules, exceptions, dynamic typing, very high level dynamic data types, and classes. Python combines remarkable power with very clear syntax. It has interfaces to many system calls and libraries, as well as to various window systems, and is extensible in C or C++. It is also usable as an extension language for applications that need a programmable interface. Finally, Python is portable: it runs on many Unix variants, on the Mac, and on Windows 2000 and later. The Beginner's Guide to Python links to other introductory tutorials and resources for learning Python.

https://www.python.org/about/gettingstarted

AForge

AForge.NET is an open source C# framework designed for developers and researchers in the fields of Computer Vision and Artificial Intelligence - image processing, neural networks, genetic algorithms, fuzzy logic, machine learning, robotics, etc.

http://www.aforgenet.com/framework/

Wolfram MathWorld

MathWorld is an online mathematics resource, assembled by Eric W. Weisstein with assistance from thousands of contributors. Since its contents first appeared online in 1995, MathWorld has emerged as a nexus of mathematical information in both the mathematics and educational communities. Its entries are extensively referenced in journals and books spanning all educational levels.

http://mathworld.wolfram.com/

INDEX OF NODES

This index provides additional information on all the nodes used in this primer, as well as other components you might find useful. This is just an introduction to some of the 500 nodes available in Dynamo.

Bulitin Functions

Count
Returns number of items in the specified list.

Flatten
Returns the flattened 1D list of the multidimensional input list.

Map
Maps a value into an input range

Core

Core.Color

CREATE

Color.ByARGB
Construct a color by alpha, red, green, and blue components.

Color Range
Get a color from a color gradient between a start color and an end color.

ACTIONS

Color.Brightness
Gets the brightness value for this color.

Dynamo Packages

Here are a list of some of the more popular packages in the Dynamo community. Developers, please add to the list! Remember, the Dynamo Primer is open-source!

BUMBLEBEE FOR DYNAMO

Visit the Official BumbleBee Site

Bumblebee is an Excel and Dynamo interoperability plugin that vastly improves Dynamo's ability to read and write Excel files.

CLOCKWORK FOR DYNAMO

Visit the Clockwork For Dynamo GitHub

Clockwork is a collection of custom nodes for the Dynamo visual programming environment. It contains many Revit-related nodes, but also lots of nodes for various other purposes such as list management, mathematical operations, string operations, unit conversions, geometric operations (mainly bounding boxes, meshes, planes, points, surfaces, UVs and vectors) and paneling.

DYNAMO SAP

Visit the DynamoSAP Project at Core Studio

DynamoSAP is a parametric interface for SAP2000, built on top of Dynamo. The project enables designers and engineers to generatively author and analyze structural systems in SAP, using Dynamo to drive the SAP model. The project prescribes a few common workflows which are described in the included sample files, and provides a wide range of opportunities for automation of typical tasks in SAP.

DYNAMO UNFOLD

Visit the DynamoUnfold GitHub

This library extends Dynamo/Revit functionality by enabling users to unfold surface and poly-surface geometry. The library allows users to first translate surfaces into planar tessellated topology, then unfold them using Protogeometry tools in Dynamo. This package also includes some experimental nodes as well as a few basic sample files.

Download Dynastrator at the

Dynamo Example Files

These example files accompany the Dynamo Primer, and are organized according to Chapter and Section.

Right click files and use "Save Link As..."

Introduction

Section	Download File
What is Visual Programming	Visual Programming - Circle Through Point.dyn

The Building Blocks of Programs

Section	Download File
Data	Building Blocks of Programs - Data.dyn
Math	Building Blocks of Programs - Math.dyn
Logic	Building Blocks of Programs - Logic.dyn
Strings	Building Blocks of Programs - Strings.dyn
Color	Building Blocks of Programs - Color.dyn

Geometry for Computational Design

Section	Download File
Geometry Overview	Geometry for Computational Design - Geometry Overview.dyn
Vectors	Geometry for Computational Design - Vectors.dyn
	Geometry for Computational Design - Plane.dyn
	Geometry for Computational Design - Coordinate System.dyn
Points	Geometry for Computational Design - Points.dyn
Curves	Geometry for Computational Design - Curves.dyn
Surfaces	Geometry for Computational Design - Surfaces.dyn
	Surface.sat

Designing with Lists

Section	Download File
What's a List	Lacing.dyn
Working with Lists	List-Count.dyn
	List-FilterByBooleanMask.dyn
	List-GetItemAtIndex.dyn

Resources

Resources

Dynamo Wiki

"This wiki is for learning about development using the Dynamo API, supporting libraries and tools."

https://github.com/DynamoDS/Dynamo/wiki

Dynamo Blog

This blog is the most up-to-date collection of articles from the Dynamo team, discussing new features, workflows, and all things Dynamo.

http://dynamobim.com/blog/

DesignScript Guide

Programming languages are created to express ideas, usually involving logic and calculation. In addition to these objectives, the Dynamo textual language (formerly DesignScript) has been created to express design intentions. It is generally recognized that computational designing is exploratory, and Dynamo tries to support this: we hope you find the language flexible and fast enough to take a design from concept, through design iterations, to your final form. This manual is structured to give a user with no knowledge of either programming or architectural geometry full exposure to a variety of topics in these two intersecting disciplines.

http://dynamobim.org/wp-content/links/DesignScriptGuide.pdf

The Dynamo Primer Project

The Dynamo Primer is an open source project, initiated by Matt Jezyk and the Dynamo Development team at Autodesk. The first version of the primer was developed by Mode Lab. To contribute, fork the repo, add your content, and submit a pull request.

https://github.com/DynamoDS/DynamoPrimer

Zero Touch Plugin Development for Dynamo

This page outlines the process of developing a custom Dynamo node in C# using the "Zero Touch" interface. In most cases, C# static methods and Classes can be imported without modification. If your library only needs to call functions, and not construct new objects, this can be achieved very easily with static methods. When Dynamo loads your DLL, it will strip off the namespace of your classes, and expose all static methods as nodes.

https://github.com/DynamoDS/Dynamo/wiki/Zero-Touch-Plugin-Development

Python for Beginners

Python is an interpreted, interactive, object-oriented programming language. It incorporates modules, exceptions, dynamic typing, very high level dynamic data types, and classes. Python combines remarkable power with very clear syntax. It has interfaces to many system calls and libraries, as well as to various window systems, and is extensible in C or C++. It is also usable as an extension language for applications that need a programmable interface. Finally, Python is portable: it runs on many Unix variants, on the Mac, and on Windows 2000 and later. The Beginner's Guide to Python links to other introductory tutorials and resources for learning Python.

https://www.python.org/about/gettingstarted

AForge

AForge.NET is an open source C# framework designed for developers and researchers in the fields of Computer Vision and Artificial Intelligence - image processing, neural networks, genetic algorithms, fuzzy logic, machine learning, robotics, etc.

http://www.aforgenet.com/framework/

Wolfram MathWorld

MathWorld is an online mathematics resource, assembled by Eric W. Weisstein with assistance from thousands of contributors. Since its contents first appeared online in 1995, MathWorld has emerged as a nexus of mathematical information in both the mathematics and educational communities. Its entries are extensively referenced in journals and books spanning all educational levels.

http://mathworld.wolfram.com/

Revit Resources

buildz

"These posts are mainly about the Revit platform, with recommendations on how to enjoy it."

http://buildz.blogspot.com/

Nathan's Revit API Notebook

"This notebook attempts to remedy a few 'resource deficiencies' in learning and applying the Revit API in the context of a design workflow"

http://wiki.theprovingground.org/revit-api

Revit Python Shell

"The RevitPythonShell adds an IronPython interpreter to Autodesk Revit and Vasari." This project pre-dates Dynamo and is a great reference for Python development. RPS Project: https://github.com/architecture-building-systems/revitpythonshell Developer's Blog: http://darenatwork.blogspot.com/

The Building Coder

A robust catalogue of Revit API workflows from one of the leading experts in BIM.

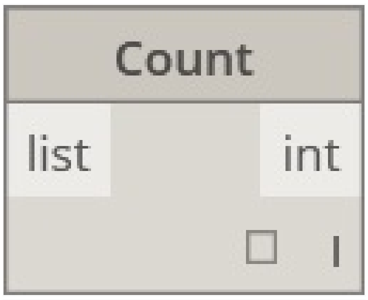
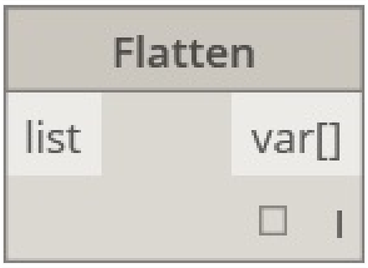
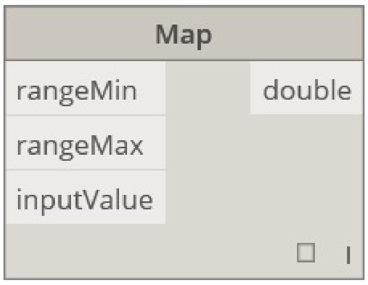
http://thebuildingcoder.typepad.com/

Index of Nodes

INDEX OF NODES

This index provides additional information on all the nodes used in this primer, as well as other components you might find useful. This is just an introduction to some of the 500 nodes available in Dynamo.

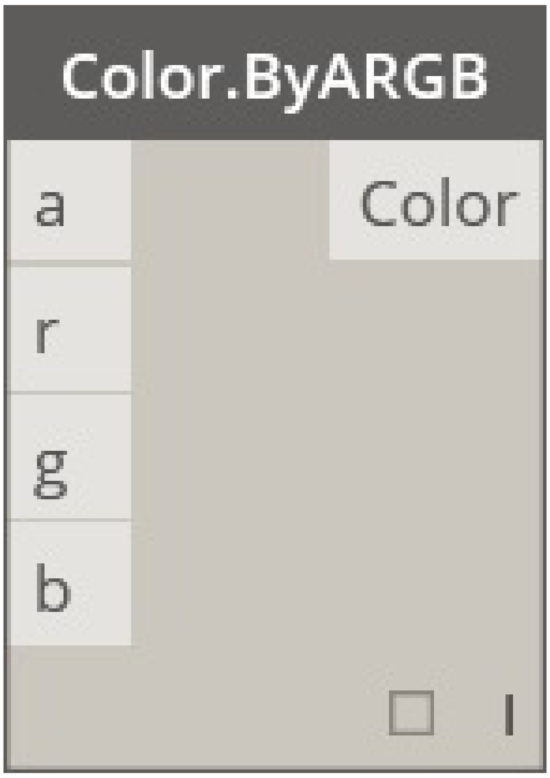
Bulitin Functions

	Count Returns number of items in the specified list.	
	Flatten Returns the flattened 1D list of the multidimensional input list.	
	Map Maps a value into an input range	

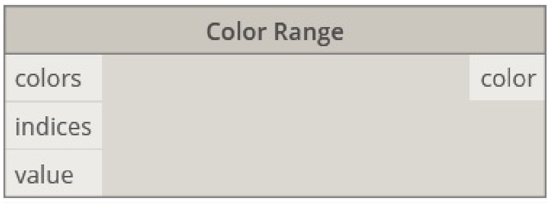
Core

Core.Color

	CREATE	
	Color.ByARGB Construct a color by alpha, red, green, and blue components.	

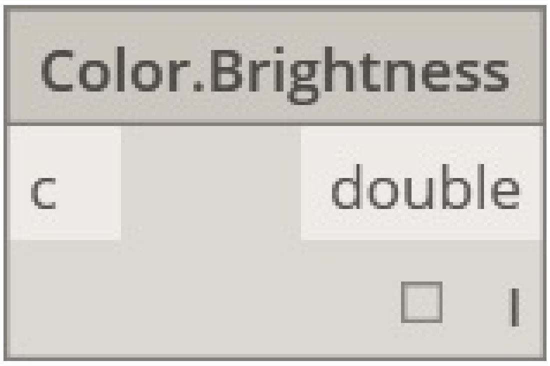


Color Range
Get a color from a color gradient between a start color and an end color.

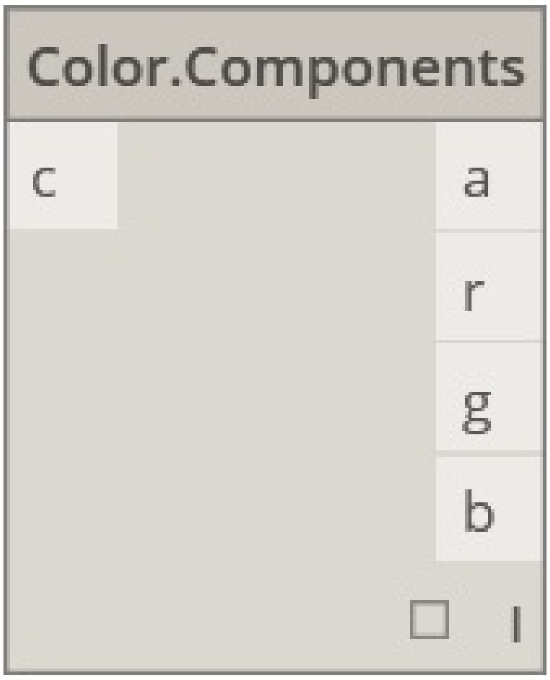


ACTIONS

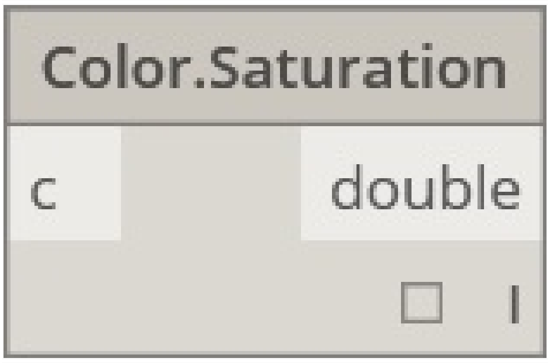
Color.Brightness
Gets the brightness value for this color.



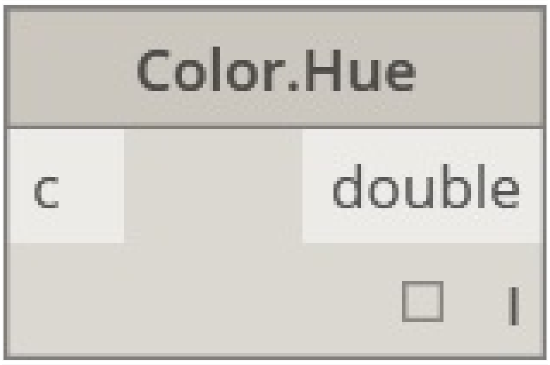
Color.Components
Lists the components for the color in the order: alpha, red, green, blue.



Color.Saturation
Gets the saturation value for this color

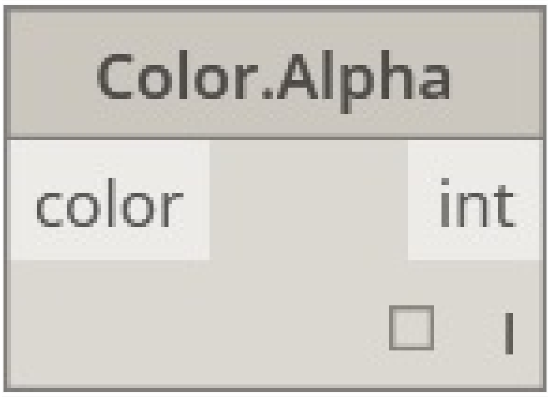


Color.Hue
Gets the hue value for this color.

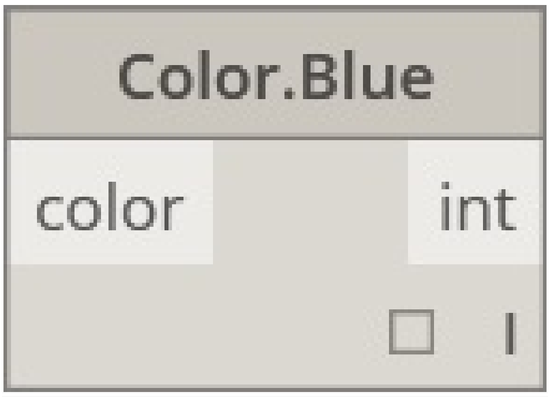
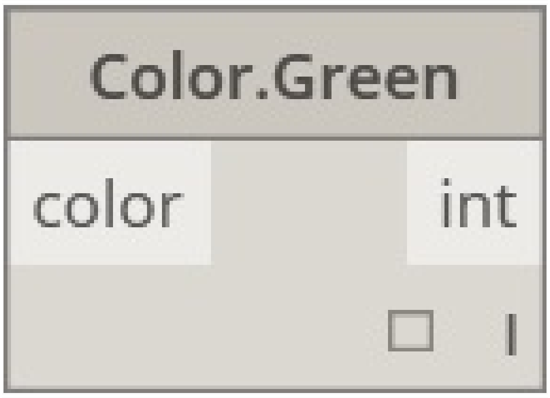
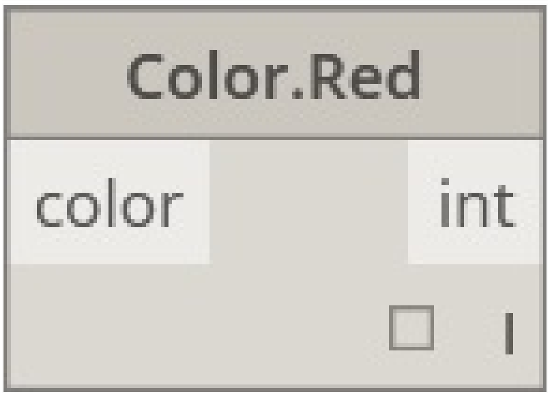


QUERY

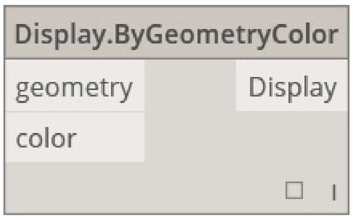
Color.Alpha
Find the alpha component of a color, 0 to 255.



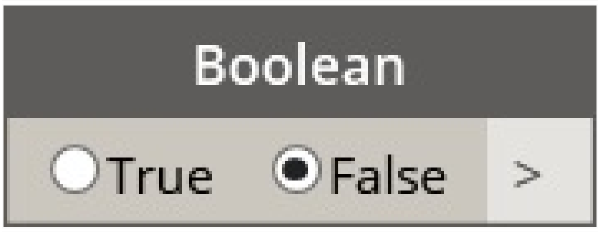
Color.Blue
Find the blue component of a color, 0 to 255.

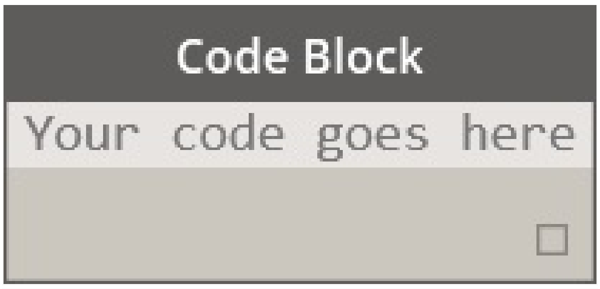
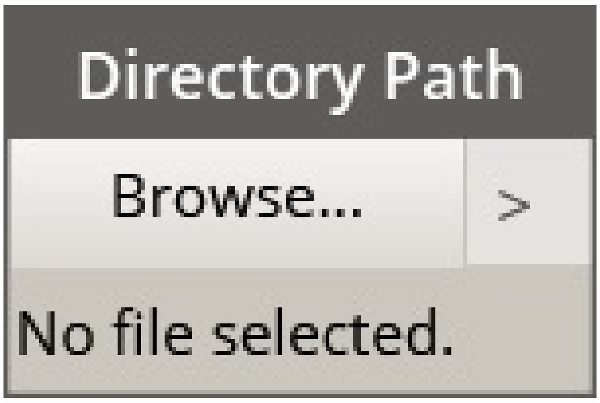
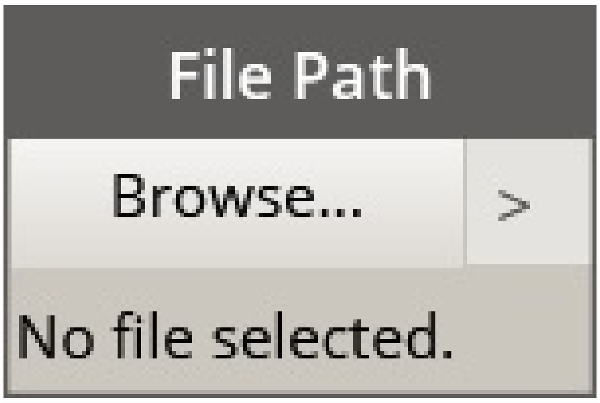
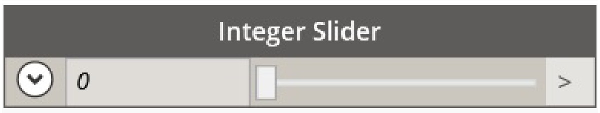
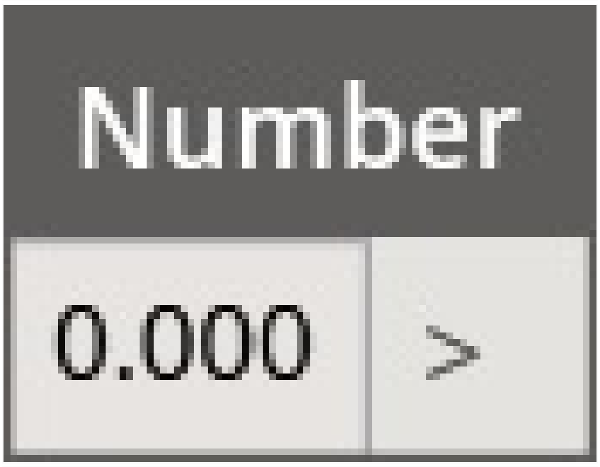
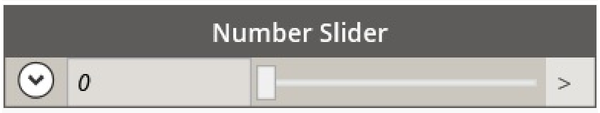
●		
●	**Color.Green** Find the green component of a color, 0 to 255.	
●	**Color.Red** Find the red component of a color, 0 to 255.	

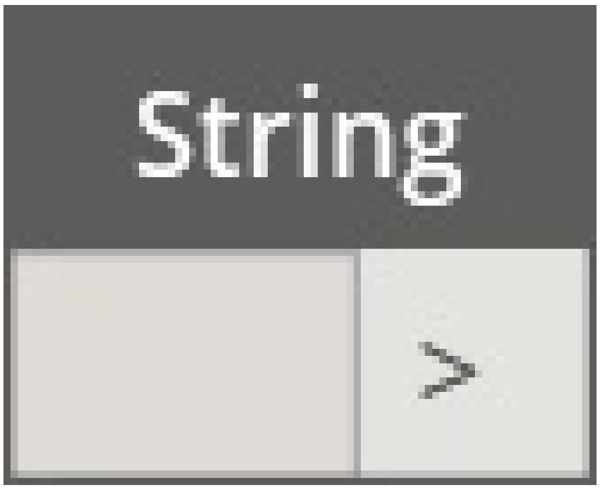
Core.Display

	CREATE	
	Display.ByGeometryColor Displays geometry using a color.	

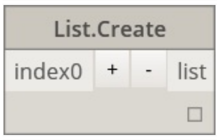
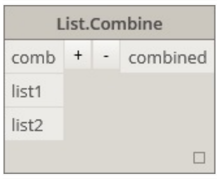
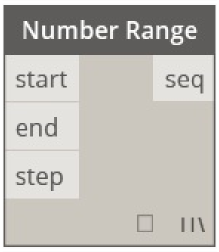
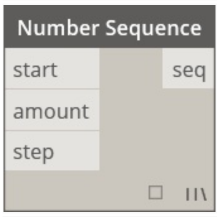
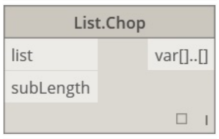
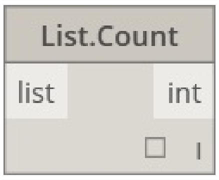
Core.Input

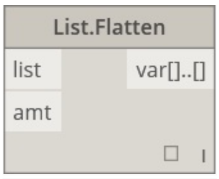
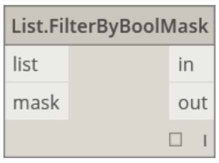
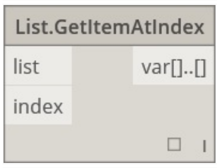
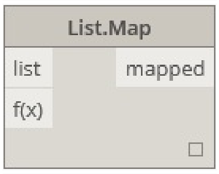
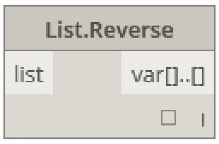
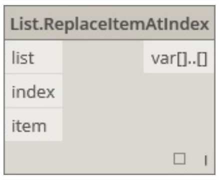
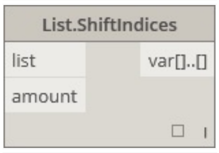
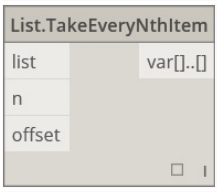
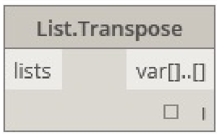
	ACTIONS	
	Boolean Selection between a true and false.	
	Code Block	

	Allows for DesignScript code to be authored directly.	
	Directory Path Allows you to select a directory on the system to get its path	
	File Path Allows you to select a file on the system to get its filename.	
	Integer Slider A slider that produces integer values.	
	Number Creates a number.	
	Number Slider A slider that produces numeric values.	

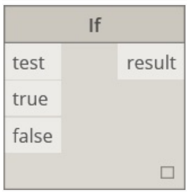
	String Creates a string.	

Core.List

	CREATE	
	List.Create Makes a new list out of the given inputs.	
	List.Combine Applies a combinator to each element in two sequences	 List.Combine comb + - combined list1 list2
	Number Range Creates a sequence of numbers in the specified range.	
	Number Sequence Creates a sequence of numbers.	
	ACTIONS	
	List.Chop Chop a list into a set of lists each containing the given amount of items.	
	List.Count Gets the number of items stored in the given list.	
	List.Flatten Flattens a nested list of lists by a certain amount.	

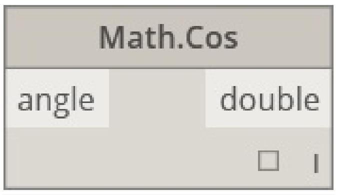
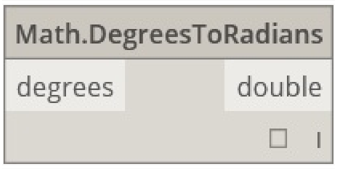
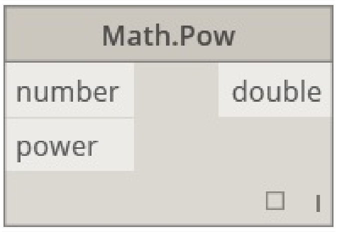
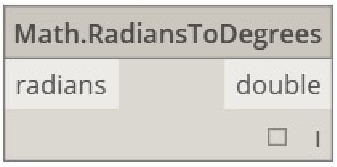
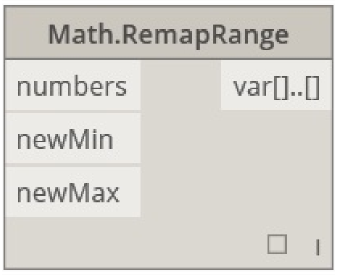
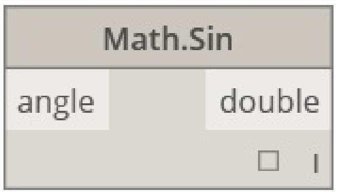
		 List.Flatten list amt var[]..[]
	List.FilterByBoolMask Filters a sequence by looking up corresponding indices in a separate list of booleans.	 List.FilterByBoolMask list mask in out
	List.GetItemAtIndex Gets an item from the given list that's located at the specified index.	 List.GetItemAtIndex list index var[]..[]
	List.Map Applies a function over all elements of a list, generating a new list from the results	 List.Map list f(x) mapped
	List.Reverse Creates a new list containing the items of the given list but in reverse order	 List.Reverse list var[]..[]
	List.ReplaceItemAtIndex Replace an item from the given list that's located at the specified index	 List.ReplaceItemAtIndex list index item var[]..[]
	List.ShiftIndices Shifts indices in the list to the right by the given amount	 List.ShiftIndices list amount var[]..[]
	List.TakeEveryNthItem Fetches items from the given list at indices that are multiples of the given value, after the given offset.	 List.TakeEveryNthItem list n offset var[]..[]
	List.Transpose Swaps rows and columns in a list of lists. If there are some rows that are shorter than others, null values are inserted as place holders in the resultant array such that it is always rectangular	 List.Transpose lists var[]..[]

Core.Logic

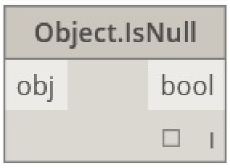
	ACTIONS	
	If Conditional statement. Checks the boolean value of the test input. If the test input is true, the result outputs the true input, otherwise the result outputs the false input.	 If test true false result

Core.Math

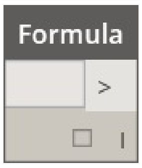
	ACTIONS	

	Math.Cos Fines the cosine of an angle.	
	Math.DegreesToRadians Converts an angle in degrees to an angle in radians.	
	Math.Pow Raises a number to the specified power.	
	Math.RadiansToDegrees Converts an angle in radians to an angle in degrees.	
	Math.RemapRange Adjusts the range of a list of numbers while preserving the distribution ratio.	
	Math.Sin Finds the sine of an angle.	

Core.Object

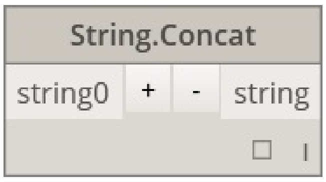
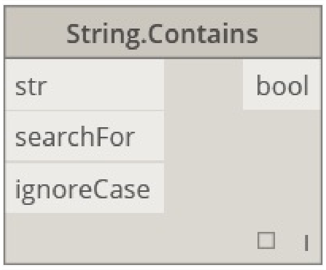
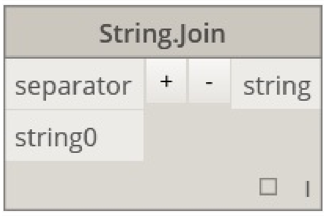
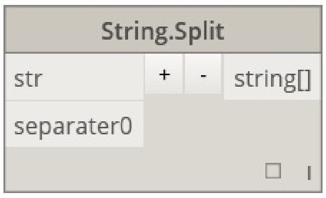
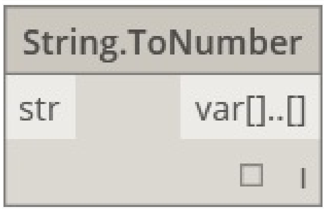
	ACTIONS	
	Object.IsNull Determines if the given object is null.	

Core.Scripting

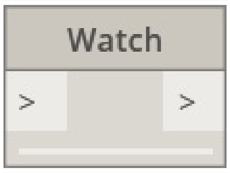
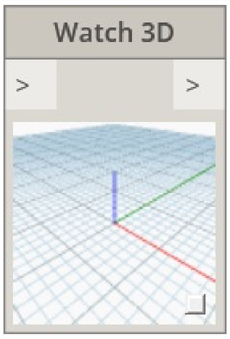
	ACTIONS	
	Formula Evaluates mathematical formulas. Uses NCalc for evaluation. See http://ncalc.codeplex.com	

Core.String

	ACTIONS	

	String.Concat Concatenates multiple strings into a single string.	
	String.Contains Determines if the given string contains the given substring.	
	String.Join Concatenates multiple strings into a single string, inserting the given separator between each joined string.	
	String.Split Divides a single string into a list of strings, with divisions determined by the given separater strings.	
	String.ToNumber Converts a string to an integer or a double.	

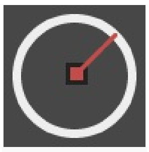
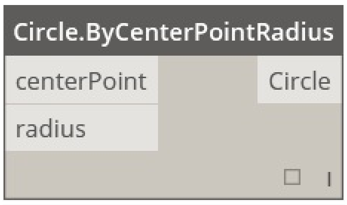
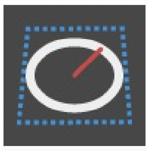
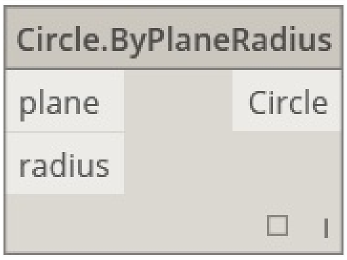
Core.View

	ACTIONS	
	View.Watch Visualize the output of node.	
	View.Watch 3D Shows a dynamic preview of geometry.	

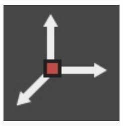
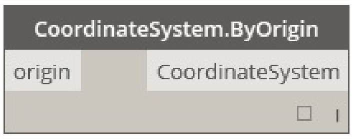
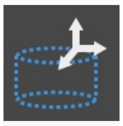
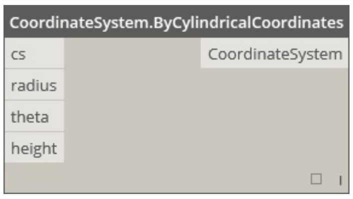
Geometry

Geometry.Circle

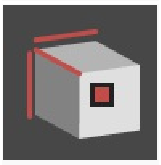
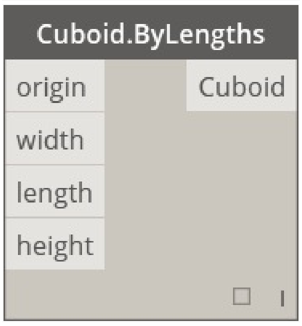
	CREATE	
	Circle.ByCenterPointRadius	

	Creates a Circle with input center Point and radius in the world XY plane, with world Z as normal.	 Circle.ByCenterPointRadius centerPoint radius Circle
	**Circle.ByPlaneRadius** Create a Circle centered at the input Plane origin (root), lying in the input Plane, with given radius.	 Circle.ByPlaneRadius plane radius Circle

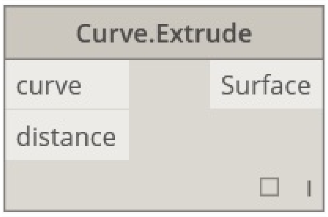
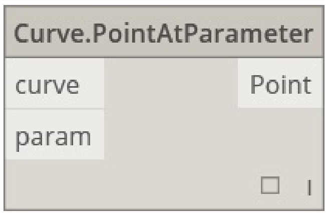
Geometry.CoordinateSystem

	CREATE	
	**CoordinateSystem.ByOrigin** Create a CoordinateSystem with origin at input Point, with X and Y Axes set as WCS X and Y axes	 CoordinateSystem.ByOrigin origin CoordinateSystem
	**CoordinateSystem.ByCyclindricalCoordinates** Creates a CoordinateSystem at the specified cylindrical coordinate parameters with respet to the specified coordinate system	 CoordinateSystem.ByCylindricalCoordinates cs radius theta height CoordinateSystem

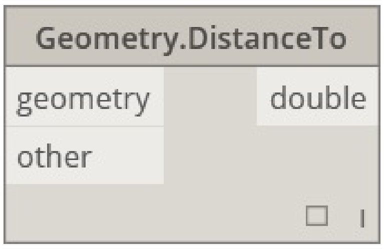
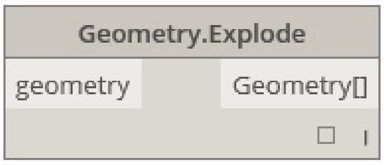
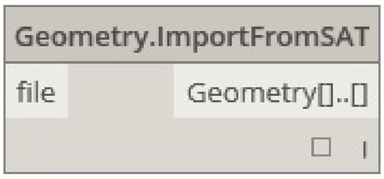
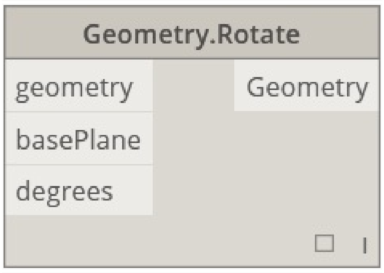
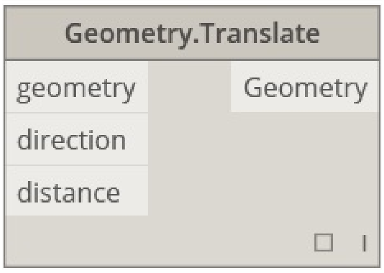
Geometry.Cuboid

	CREATE	
	**Cuboid.ByLengths** (origin) Create a Cuboid centered at input Point, with specified width, length, and height.	 Cuboid.ByLengths origin width length height Cuboid

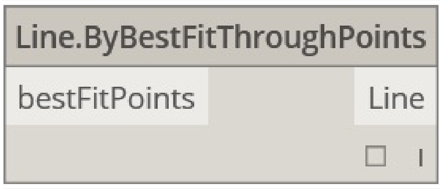
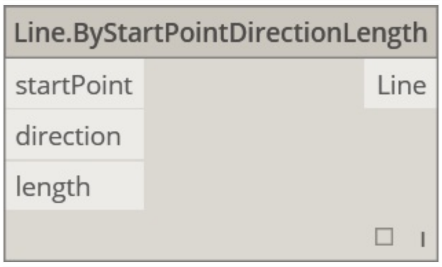
Geometry.Curve

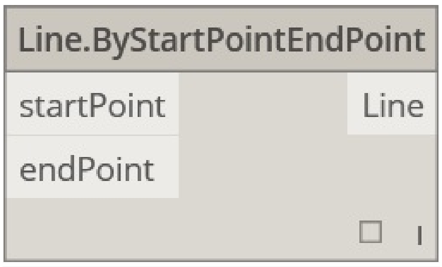
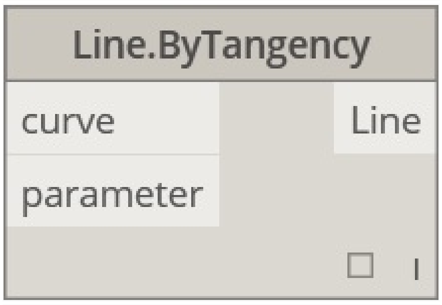
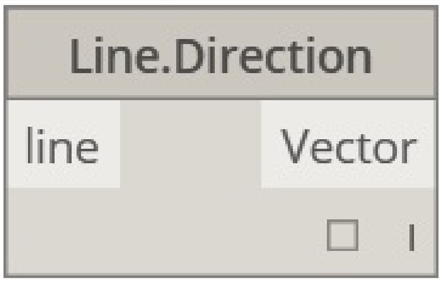
	ACTIONS	
	Curve.Extrude (distance) Extrudes a Curve in the normal Vector direction.	 Curve.Extrude curve distance Surface
	Curve.PointAtParameter Get a Point on the Curve at a specified parameter between StartParameter() and EndParameter().	 Curve.PointAtParameter curve param Point

Geometry.Geometry

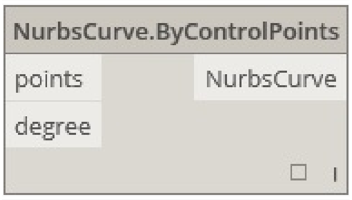
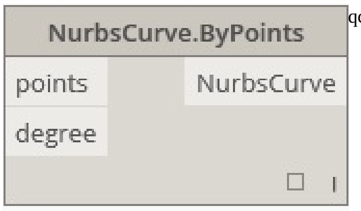
	ACTIONS	
	Geometry.DistanceTo Obtain the distance from this Geometry to another.	 Geometry.DistanceTo geometry, other → double
	Geometry.Explode Separates compound or non-separated elements into their component parts	 Geometry.Explode geometry → Geometry[]
	Geometry.ImportFromSAT List of imported geometries	 Geometry.ImportFromSAT file → Geometry[]..[]
	Geometry.Rotate (basePlane) Rotates an object around the Plane origin and normal by a specified degree.	 Geometry.Rotate geometry, basePlane, degrees → Geometry
	Geometry.Translate Translates any geometry type by the given distance in the given direction.	 Geometry.Translate geometry, direction, distance → Geometry

Geometry.Line

	CREATE	
	Line.ByBestFitThroughPoints Creates a Line best approximating a scatter plot of Points.	 Line.ByBestFitThroughPoints bestFitPoints → Line
	Line.ByStartPointDirectionLength Create a straight Line starting at Point, extending in Vector direction by specified length.	 Line.ByStartPointDirectionLength startPoint, direction, length → Line
	Line.ByStartPointEndPoint Creates a straight Line between two input Points.	

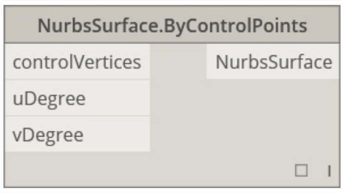
		 Line.ByStartPointEndPoint startPoint endPoint Line
	Line.ByTangency Create a Line tangent to the input Curve, positioned at the parameter Point of the input Curve.	 Line.ByTangency curve parameter Line
	QUERY	
	Line.Direction The direction of the Curve.	 Line.Direction line Vector

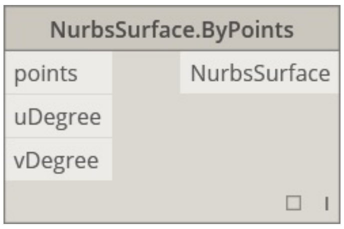
Geometry.NurbsCurve

	Create	
	NurbsCurve.ByControlPoints Create a BSplineCurve by using explicit control points.	 NurbsCurve.ByControlPoints points degree NurbsCurve
	NurbsCurve.ByPoints Create a BSplineCurve by interpolating between points	 NurbsCurve.ByPoints points degree NurbsCurve

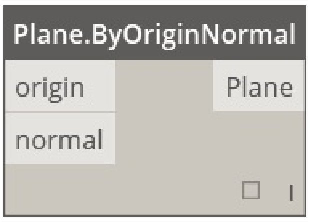
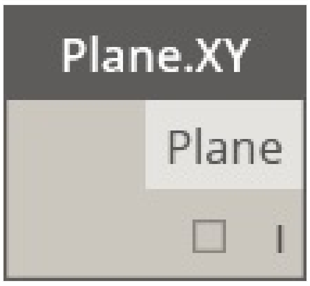
qcomm

Geometry.NurbsSurface

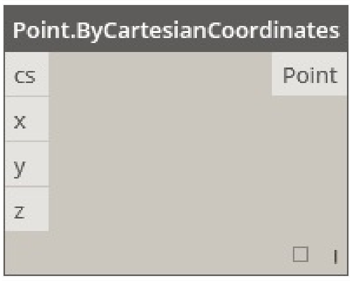
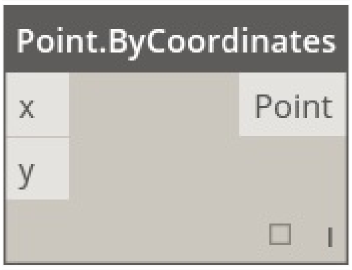
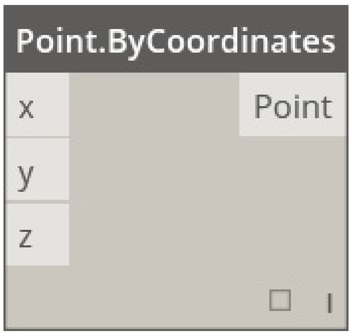
	Create	
	NurbsSurface.ByControlPoints Create a NurbsSurface by using explicit control Points with specified U and V degrees.	 NurbsSurface.ByControlPoints controlVertices uDegree vDegree NurbsSurface
	NurbsSurface.ByPoints Creates a NurbsSurface with specified interpolated points and U and V degrees. The resultant surface will pass through all of the points.	

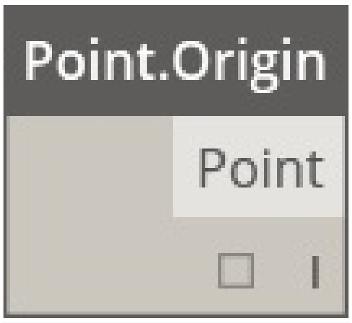
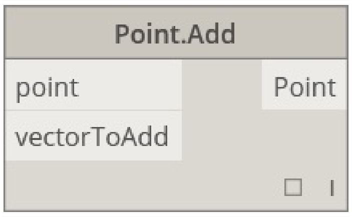
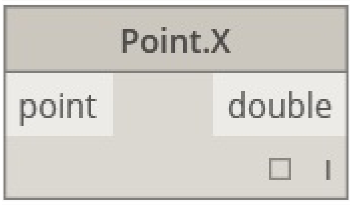
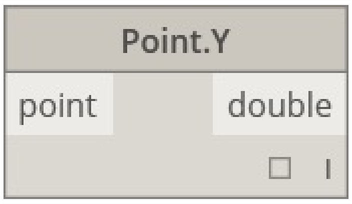
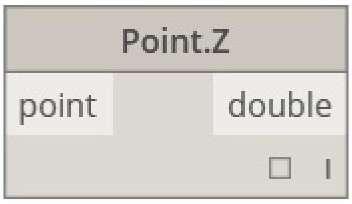
		 NurbsSurface.ByPoints points uDegree vDegree NurbsSurface

Geometry.Plane

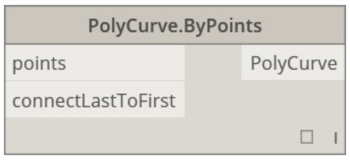
	CREATE	
	Plane.ByOriginNormal Create a Plane centered at root Point, with input normal Vector.	
	Plane.XY Creates a plane in the world XY	

Geometry.Point

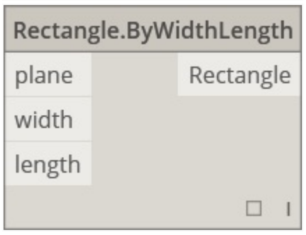
	CREATE	
	Point.ByCartesianCoordinates Form a Point in th egiven coordinate system with 3 cartesian coordinates	
	Point.ByCoordinates (2d) Form a Point in the XY plane given two 2 Cartesian coordinates. The Z component is 0.	
	Point.ByCoordinates (3d) Form a Point given 3 Cartesian coordinates.	
	Point.Origin Get the Origin point (0,0,0)	

		Point.Origin Point
	ACTIONS	
	Point.Add Add a vector to a point. The same as Translate (Vector).	Point.Add point vectorToAdd Point
	QUERY	
	Point.X Get the X component of a point	Point.X point double
	Point.Y Get the Y component of a point	Point.Y point double
	Point.Z Get the Z component of a point	Point.Z point double

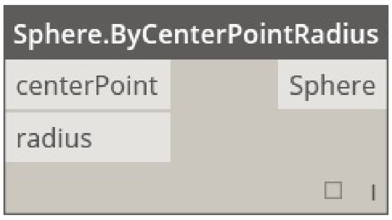
Geometry.Polycurve

	CREATE	
	Polycurve.ByPoints Make PolyCurve from sequence of lines connecting points. For closed curve last point should be in the same location as the start point.	PolyCurve.ByPoints points connectLastToFirst PolyCurve

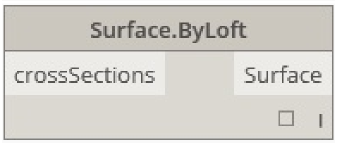
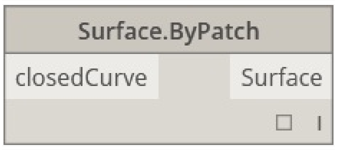
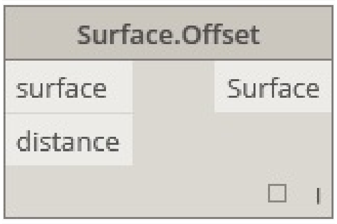
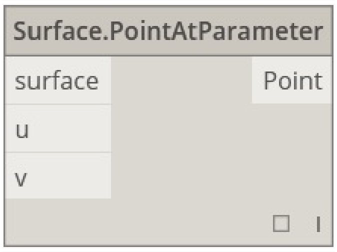
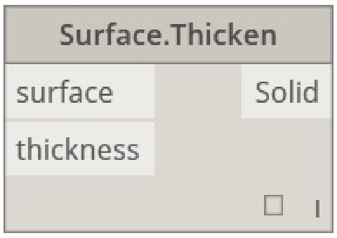
Geometry.Rectangle

	CREATE	
	Rectangle.ByWidthLength (Plane) Create a Rectangle centered at input Plane root, with input width (Plane X axis length) and (Plane Y axis length).	Rectangle.ByWidthLength plane width length Rectangle

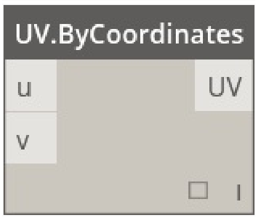
Geometry.Sphere

	CREATE	
	Sphere.ByCenterPointRadius Create a Solid Sphere centered at the input Point, with given radius.	 Sphere.ByCenterPointRadius centerPoint Sphere radius

Geometry.Surface

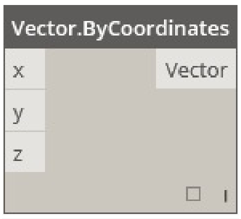
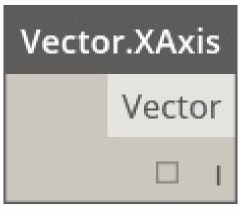
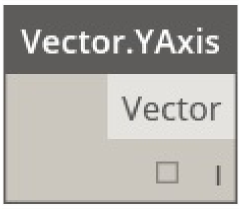
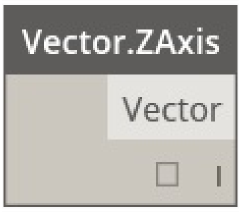
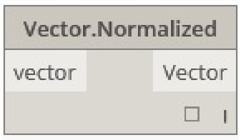
	CREATE	
	Surface.ByLoft Create a Surface by lofting between input cross section Curves	
	Surface.ByPatch Create a Surface by filling in the interior of a closed boundary defined by input Curves.	
	ACTIONS	
	Surface.Offset Offset Surface in direction of Surface normal by specified distance	
	Surface.PointAtParameter Return the Point at a specified U and V parameters.	
	Surface.Thicken Thicken Surface into a Solid, extruding in the direction of Surface normals on both sides of the Surface.	

Geometry.UV

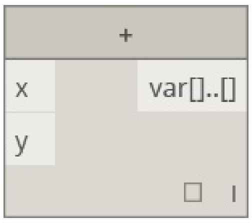
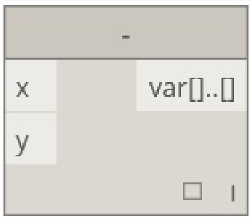
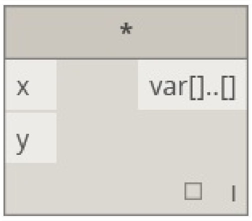
	CREATE	
	UV.ByCoordinates Create a UV from two doubles.	

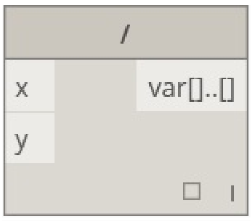
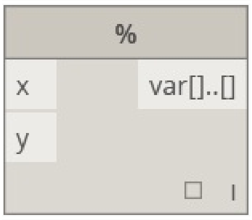
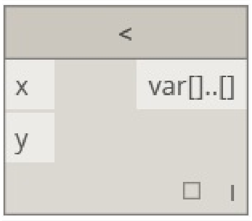
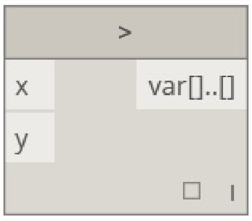
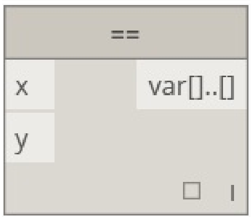
Geometry.Vector

	CREATE	
	Vector.ByCoordinates Form a Vector by 3 Euclidean coordinates	

		
	Vector.XAxis Gets the canonical X axis Vector (1,0,0)	
	Vector.YAxis Gets the canonical Y axis Vector (0,1,0)	
	Vector.ZAxis Gets the canonical Z axis Vector (0,0,1)	
	ACTIONS	
	Vector.Normalized Get the normalized version of a vector	

Operators

	+ Addition	
	- Subtraction	
	* Multiplication	
	/ Division	

		
	% Modular Division finds the remainder of the first input after dividing by the second input	
	< Less Than	
	> Greater Than	
	== Equality tests for equality between two values.	

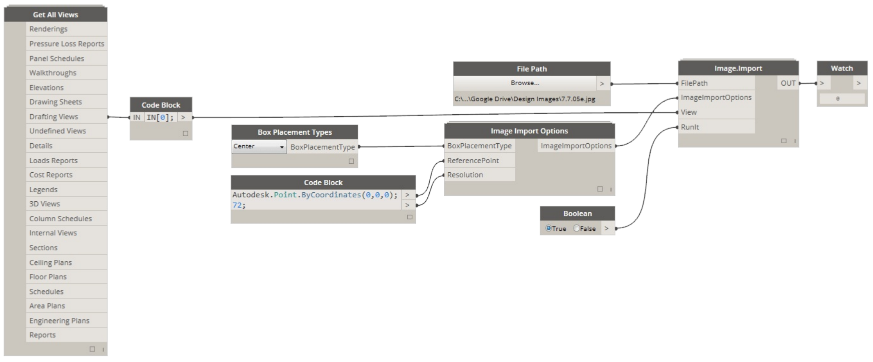
Useful Packages

Dynamo Packages

Here are a list of some of the more popular packages in the Dynamo community. Developers, please add to the list! Remember, the Dynamo Primer is open-source!

ARCHI-LAB Visit the Official archi-lab Site

archi-lab is a collection of over 50+ custom packages that vastly extend Dynamo's ability to interact with Revit. Nodes contained in archi-lab package vary from basic list operations to advanced Analysis Visualization Framework nodes for Revit. archi-lab is available on the package manager



```
IN[0];
```

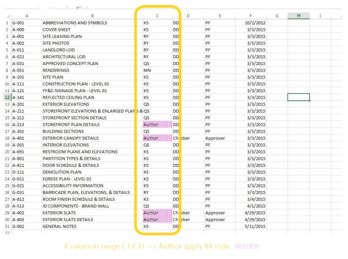
C:\...\Google Drive\Design Images\7.7.05e.jpg

```
Autodesk.Point.ByCoordinates(0,0,0);
72;
```

BimorphNodes is a versatile collection of powerful utility nodes. The package highlights include ultra-efficient clash detection and geometry intersection no
ImportInstance (CAD) curve conversion nodes, and linked element collectors that resolve limitations in the Revit API. To learn about the full range of nodes
visit the BimorphNodes dictionary. BimorphNodes is available on the package manager.

BUMBLEBEE FOR DYNAMO [Visit the Official BumbleBee Site]

Bumblebee is an Excel and Dynamo interoperability plugin that vastly improves Dynamo's ability to read and write Excel files.

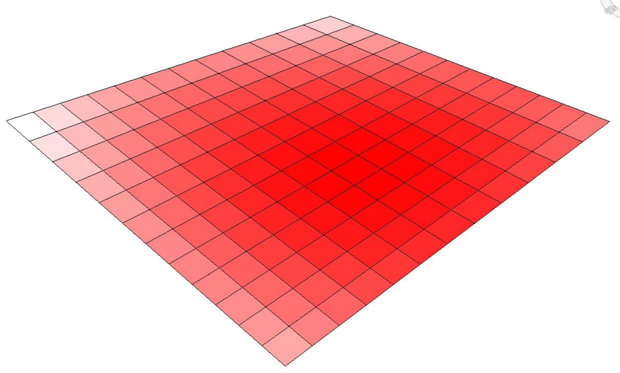


	A	B	C	D	E	F	G	H	I
1	G-001	ABBREVIATIONS AND SYMBOLS	KS	DD	PF	10/2/2012			
2	A-000	COVER SHEET	KS	DD	PF	3/3/2015			
3	A-001	SITE LEASING PLAN	RY	DD	PF	3/3/2015			
4	A-002	SITE PHOTOS	RY	DD	PF	3/3/2015			
5	A-011	LANDLORD LOD	RY	DD	PF	3/3/2015			
6	A-021	ARCHITECTURAL LOD	RY	DD	PF	3/3/2015			
7	A-031	APPROVED CONCEPT PLAN	QS	DD	PF	3/3/2015			
8	A-051	RENDERINGS	MN	DD	PF	3/3/2015			
9	A-101	SITE PLAN	KS	DD	PF	3/3/2015			
10	A-111	CONSTRUCTION PLAN - LEVEL 01	KS	DD	PF	3/3/2015			
11	A-121	FF&E-SIGNAGE PLAN - LEVEL 01	KS	DD	PF	3/3/2015			
12	A-141	REFLECTED CEILING PLAN	KS	DD	PF	3/3/2015			
13	A-201	EXTERIOR ELEVATIONS	QS	DD	PF	3/3/2015			
14	A-211	STOREFRONT ELEVATIONS & ENLARGED PLA S &	QS	DD	PF	3/3/2015			
15	A-212	STOREFRONT SECTION DETIALS	QS	DD	PF	3/3/2015			
16	A-213	STOREFRONT PLAN DETAILS	Author	DD	PF	3/3/2015			
17	A-301	BUILDING SECTIONS	QS	DD	PF	3/3/2015			
18	A-401	EXTERIOR CANOPY DETAILS	Author	Ch cker	Approver	3/3/2015			
19	A-501	INTERIOR ELEVATIONS	QS	DD	PF	3/3/2015			
20	A-691	RESTROOM PLANS AND ELEVATIONS	KS	DD	PF	3/3/2015			
21	A-801	PARTITION TYPES & DETAILS	KS	DD	PF	3/3/2015			
22	A-811	DOOR SCHEDULE & DETAILS	KS	DD	PF	3/3/2015			
23	D-111	DEMOLITION PLAN	KS	DD	PF	3/3/2015			
24	G-011	EGRESS PLAN - LEVEL 01	KS	DD	PF	3/3/2015			
25	G-021	ACCESSIBILITY INFORMATION	KS	DD	PF	3/3/2015			
26	G-031	BARRICADE PLAN, ELEVATIONS, & DETAILS	RY	DD	PF	3/3/2015			
27	A-812	ROOM FINISH SCHEDULE & DETAILS	KS	DD	PF	3/4/2015			
28	A-513	ID COMPONENTS - BRAND WALL	QS	DD	PF	4/1/2015			
29	A-402	EXTERIOR SLATS	Author	Ch cker	Approver	4/29/2015			
30	A-403	EXTERIOR SLATS DETAILS	Author	Ch cker	Approver	4/29/2015			
31	G-002	GENERAL NOTES	KS	DD	PF	5/11/2015			
32									

-

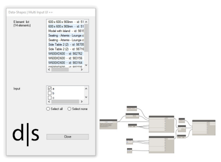
CLOCKWORK FOR DYNAMO [Visit the Clockwork For Dynamo GitHub]

Clockwork is a collection of custom nodes for the Dynamo visual programming environment. It contains many Revit-related nodes, but also lots of nodes for various other purposes such as list management, mathematical operations, string operations, unit conversions, geometric operations (mainly bounding boxes, meshes, planes, points, surfaces, UVs and vectors) and paneling.



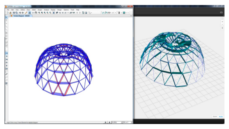
DATA|SHAPES [Visit Data|Shapes on GitHub]

DataShapes is a package that aims to extend the user functionality of Dynamo scripts. This has a heavy focus on adding greater functionality to Dynamo player. For more infor visit https://data-shapes.net/. Want to create awesome Dynamo player workflows? Use this package.



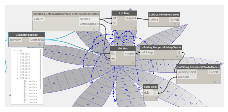
DYNAMO SAP Visit the DynamoSAP Project at Core Studio

DynamoSAP is a parametric interface for SAP2000, built on top of Dynamo. The project enables designers and engineers to generatively author and analyze structural systems in SAP, using Dynamo to drive the SAP model. The project prescribes a few common workflows which are described in the included sample files, and provides a wide range of opportunities for automation of typical tasks in SAP.



DYNAMO UNFOLD [Visit the DynamoUnfold GitHub]

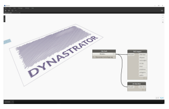
This library extends Dynamo/Revit functionality by enabling users to unfold surface and poly-surface geometry. The library allows users to first translate surfaces into planar tessellated topology, then unfold them using Protogeometry tools in Dynamo. This package also includes some experimental nodes as well as a few basic sample files.



[Download]

DYNASTRATOR Dynastrator at the Package Manager

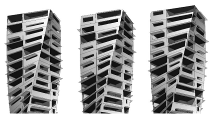
Import vector art from Illustrator or the web using .svg. This allows you to import manually created drawings into Dynamo for parametric operations.



ENERGY ANALYSIS Visit the Energy Analysis

FOR DYNAMO for Dynamo Project on GitHub

Energy Analysis for Dynamo allows for parametric energy modeling and whole-building energy analysis workflows in Dynamo 0.8. Energy Analysis for Dynamo allows the user to configure the energy model from Autodesk Revit, submit to Green Building Studio for DOE2 energy analysis, and dig into the results returned from the analysis. The package is being developed by Thornton Tomasetti's CORE studio.

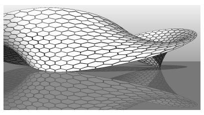


Firefly is a collection of nodes which enable dynamo to talk to input/output devices, like the Arduino micro controller. Because the data flow happens “live”
opens up many opportunities for interactive prototyping between the digital and physical worlds through web cams, mobile phones, game controllers, senso

LUNCHBOX FOR DYNAMO

Lunchbox is provided in the primer as a resource.

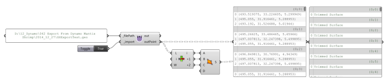
LunchBox is a collection of reusable geometry and data management nodes. The tool includes nodes for surface paneling, geometry, Revit data collection, and more! You can read all about Lunchbox at Proving Ground.



MANTIS SHRIMP

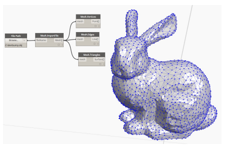
Visit the official Mantis Shrimp site.

Mantis Shrimp is an interoperability project that allows you to easily import Grasshopper and/or Rhino geometry into Dynamo.



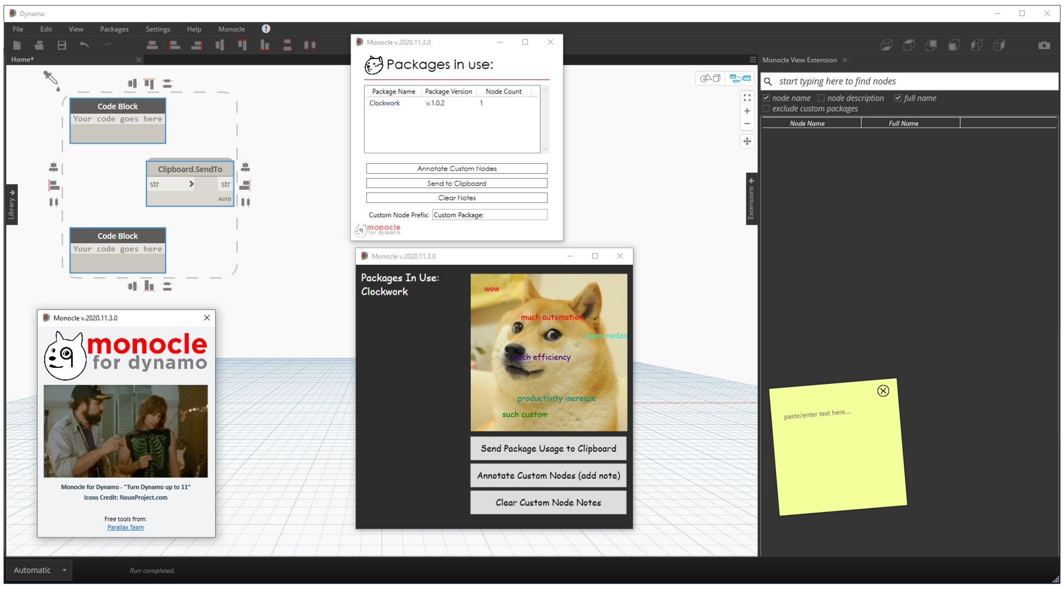
MESH TOOLKIT [Visit the Dynamo Mesh Toolkit GitHub]

The Dynamo Mesh Toolkit provides many useful tools for working with mesh geometry. The functionality of this package includes the ability to import meshes from external file formats, generate meshes from pre-existing Dynamo geometry objects, and manually build meshes through vertices and connectivity information. Additionally, this toolkit includes tools to modify and repair mesh geometry.

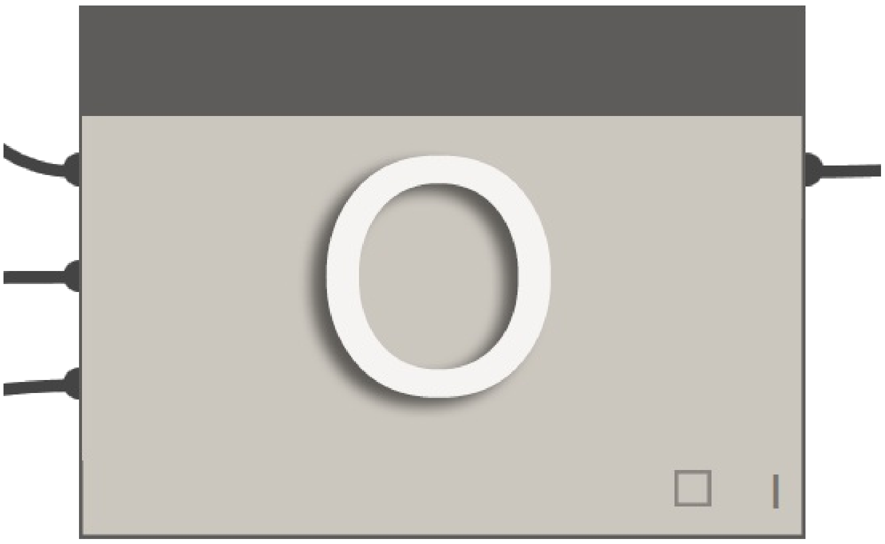


MONOCLE [Visit the Monocle GitHub]

Monocle is a View Extension for Dynamo 2.0.x. Monocle contains a set of useful tools for package identification, graph cleanup and more! Monocle aims to add functionality to the Dynamo UI in a seemless way that leaves you thinking, *"is this built into dynamo?"*. Monocle is available on the package manager.

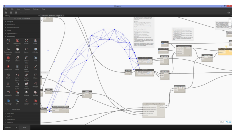


Package Name	Package Version	Node Count
Clockwork	v.1.0.2	1

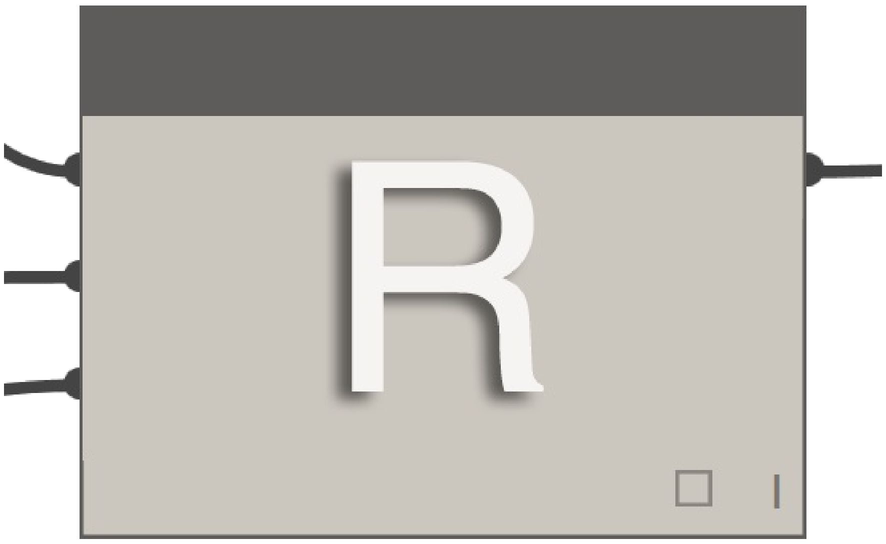


OPTIMO Visit the Optimo GitHub

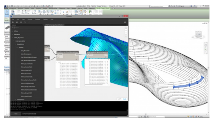
Optimo provides dynamo users with the capability to optimize self-defined design problems by using various evolutionary algorithms. Users can define the problem objective or set of objectives as well as specific fitness functions.



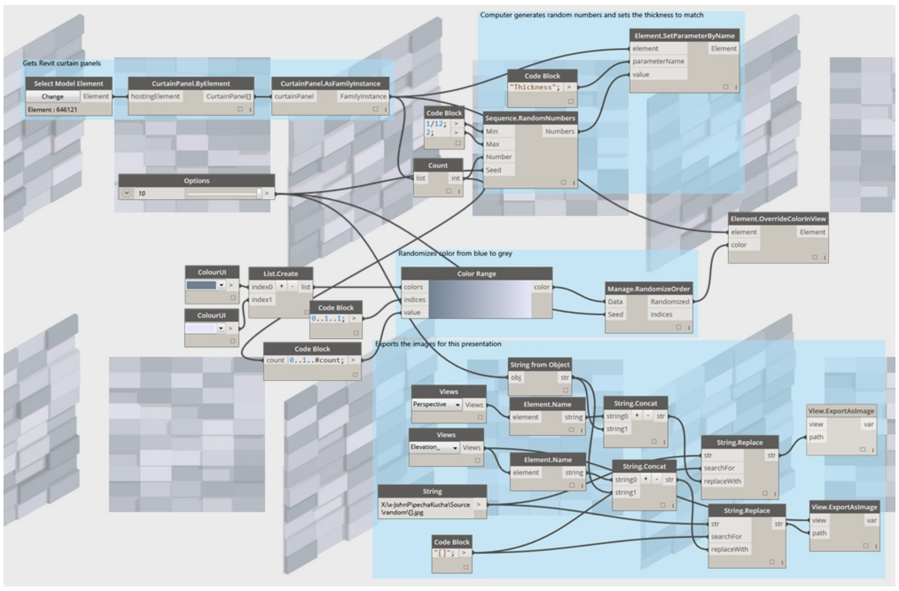
RHYNAMO [Visit the Rhynamo Bitbucket]



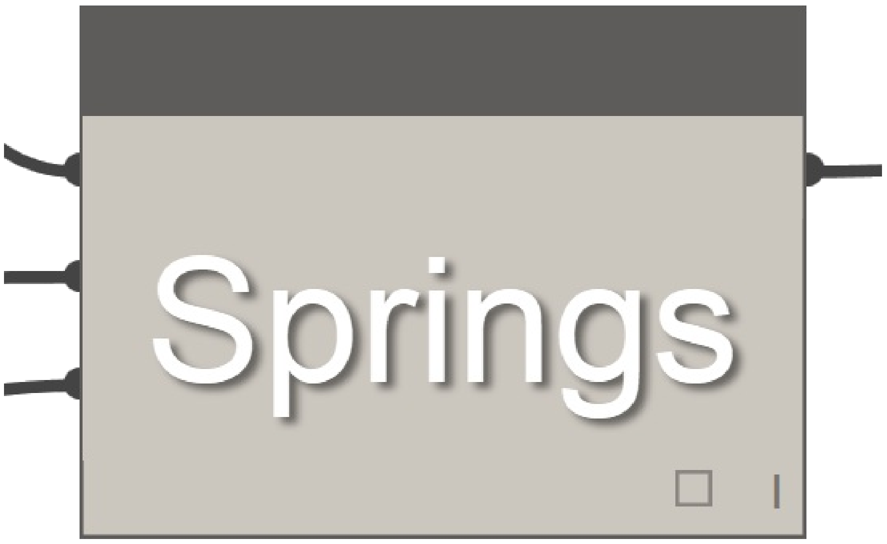
The Rhynamo node library provides users with the ability to read and write Rhino 3DM files from within Dynamo. Rhynamo translates Rhino geometry into usable Dynamo geometry by using McNeel's OpenNURBS library allowing for new workflows that can exchange geometry and data fluidly between Rhino and Revit. This package also contains some experimental nodes that allow for "live" access to the Rhino command line.



RHYTHM [Visit Rhythm on GitHub]



Rhythm is a set of useful nodes to help your Revit project maintain a good rhythm with Dynamo. basically it does some pretty okay stuff. Rhythm is open source and primarily built in C#, and adds Revit nodes, core nodes and a view extension to your Dynamo. Rhythm is available on the package manager.



Spring Nodes [Visit Spring Nodes on GitHub]

Spring nodes main focus is to improve Dynamo's interaction with Revit. The wider goal is to explore any and all means that can help accelerate BIM focused work-flows. Many of the nodes use either IronPython or DesignScript and can be a good starting point for learning the specific syntax and finer points of both. Spring nodes is available on the package manager.

Example Files

Dynamo Example Files

These example files accompany the Dynamo Primer, and are organized according to Chapter and Section.

Right click files and use "Save Link As..."

Introduction

Section	Download File
What is Visual Programming	Visual Programming - Circle Through Point.dyn

Anatomy of a Dynamo Definition

Section	Download File
Presets	Presets.dyn

The Building Blocks of Programs

Section	Download File
Data	Building Blocks of Programs - Data.dyn
Math	Building Blocks of Programs - Math.dyn
Logic	Building Blocks of Programs - Logic.dyn
Strings	Building Blocks of Programs - Strings.dyn
Color	Building Blocks of Programs - Color.dyn

Geometry for Computational Design

Section	Download File
Geometry Overview	Geometry for Computational Design - Geometry Overview.dyn
Vectors	Geometry for Computational Design - Vectors.dyn
	Geometry for Computational Design - Plane.dyn
	Geometry for Computational Design - Coordinate System.dyn
Points	Geometry for Computational Design - Points.dyn
Curves	Geometry for Computational Design - Curves.dyn
Surfaces	Geometry for Computational Design - Surfaces.dyn
	Surface.sat

Designing with Lists

Section	Download File
What's a List	Lacing.dyn
Working with Lists	List-Count.dyn
	List-FilterByBooleanMask.dyn
	List-GetItemAtIndex.dyn
	List-Operations.dyn
	List-Reverse.dyn
	List-ShiftIndices.dyn
Lists of Lists	Chop.dyn
	Combine.dyn
	Flatten.dyn
	Map.dyn
	ReplaceItems.dyn
	Top-Down-Hierarchy.dyn
	Transpose.dyn
n-Dimensional Lists	n-Dimensional-Lists.dyn
	n-Dimensional-Lists.sat

Code Blocks and DesignScript

Section	Download File
DesignScript Syntax	Dynamo-Syntax_Attractor-Surface.dyn
Shorthand	Obsolete-Nodes_Sine-Surface.dyn
Functions	Functions_SphereByZ.dyn

Dynamo for Revit

Section	Download File
Selecting	Selecting.dyn
	ARCH-Selecing-BaseFile.rvt
Editing	Editing.dyn
	ARCH-Editing-BaseFile.rvt
Creating	Creating.dyn
	ARCH-Creating-BaseFile.rvt
Customizing	Customizing.dyn
	ARCH-Customizing-BaseFile.rvt
Documenting	Documenting.dyn
	ARCH-Documenting-BaseFile.rvt

Dictionaries in Dynamo

Section	Download File
Room Dictionary	RoomDictionary.dyn

Custom Nodes

Section	Download File
Creating a Custom Node	UV-CustomNode.zip
Publishing to Your Library	PointsToSurface.dyf
Python Nodes	Python-CustomNode.dyn
Python and Revit	Revit-Doc.dyn
Python and Revit	Revit-ReferenceCurve.dyn
Python and Revit	Revit-StructuralFraming.zip

Packages

Section	Download File
Package Case Study - Mesh Toolkit	MeshToolkit.zip
Publishing a Package	MapToSurface.zip
Zero-Touch Importing	ZeroTouchImages.zip