

# Table of Contents

1. [概要](#) 0
2. [はじめに](#) 1
  1. [ビジュアル プログラミングの概要](#) 1.1
  2. [Dynamo とは](#) 1.2
  3. [DYNAMO IN ACTION](#) 1.3
3. [Dynamo の概要](#) 2
  1. [Dynamo のインストールと起動](#) 2.1
  2. [Dynamo ユーザ インタフェース](#) 2.2
  3. [ワークスペース](#) 2.3
  4. [作業を開始する](#) 2.4
4. [ビジュアル プログラムの構造](#) 3
  1. [ノード](#) 3.1
  2. [ワイヤ](#) 3.2
  3. [Dynamo ライブリ](#) 3.3
  4. [プログラムを管理する](#) 3.4
5. [プログラムの構成要素](#) 4
  1. [データ](#) 4.1
  2. [数学的方法](#) 4.2
  3. [ロジック](#) 4.3
  4. [文字列](#) 4.4
  5. [色](#) 4.5
6. [計算設計用のジオメトリ](#) 5
  1. [ジオメトリの概要](#) 5.1
  2. [ベクトル、平面、座標系](#) 5.2
  3. [点](#) 5.3
  4. [曲線](#) 5.4
  5. [サーフェス](#) 5.5
  6. [ソリッド](#) 5.6
  7. [メッシュ](#) 5.7
  8. [ジオメトリの読み込み](#) 5.8
7. [リストを使用した設計](#) 6
  1. [リストの概要](#) 6.1
  2. [リストの操作](#) 6.2
  3. [リストのリスト](#) 6.3
  4. [N 次元のリスト](#) 6.4
8. [コード ブロックと DesignScript](#) 7
  1. [コード ブロックとは](#) 7.1
  2. [DesignScript 構文](#) 7.2
  3. [省略表記](#) 7.3
  4. [コード ブロック関数](#) 7.4
9. [Revit で Dynamo を使用する](#) 8
  1. [Revit との関係](#) 8.1
  2. [選択](#) 8.2
  3. [編集](#) 8.3
  4. [作成](#) 8.4
  5. [カスタマイズ](#) 8.5
  6. [設計図書の作成](#) 8.6
10. [Dynamo のディクショナリ](#) 9
  1. [ディクショナリ](#) 9.1
  2. [\[Dictionary\] カテゴリのノード](#) 9.2
  3. [コード ブロックにおけるディクショナリ](#) 9.3
  4. [ディクショナリ - Revit での使用例](#) 9.4
11. [カスタム ノード](#) 10
  1. [カスタム ノード](#) 10.1
  2. [カスタム ノードを作成する](#) 10.2
  3. [ライブラリへの追加](#) 10.3
  4. [Python](#) 10.4
  5. [Python と Revit](#) 10.5
  6. [Python テンプレート](#) 10.6
12. [パッケージ](#) 11
  1. [パッケージ](#) 11.1
  2. [パッケージのケーススタディ - Mesh Toolkit](#) 11.2

- 3. [パッケージを開発する](#) 11.3
  - 4. [パッケージをパブリッシュする](#) 11.4
  - 5. [Zero-Touch の概要](#) 11.5
- 13. [Dynamo の Web エクスペリエンス](#) 12
    - 1. [Web に送信\(Dynamo Studio を使用\)](#) 12.1
    - 2. [\[カスタマイザ\]ビュー](#) 12.2
  - 14. [ベストプラクティス](#) 13
    - 1. [見やすいプログラムを作成するためのガイドライン](#) 13.1
    - 2. [スクリプト作成のガイドライン](#) 13.2
    - 3. [スクリプトリファレンス](#) 13.3
  - 15. [付録](#) 14
    - 1. [リソース](#) 14.1
    - 2. [ノードの索引](#) 14.2
    - 3. [Dynamo パッケージ](#) 14.3
    - 4. [Dynamo のサンプル ファイル](#) 14.4

# 概要

## Dynamo Primer

### Dynamo v2.0

Dynamo v1.3 Primer の手引きは[こちらからダウンロード](#)できます。



Dynamo は、設計者向けのオープンソースのビジュアルプログラミング プラットフォームです。

#### ようこそ

この Dynamo Primer は、Autodesk Dynamo Studio を使用したビジュアルプログラミングの総合ガイドです。この手引は、ビジュアルプログラミングの基本情報を共有することを目的としており、継続的に更新されています。この手引きには、ジオメトリの計算設計、規則に基づく設計のベストプラティクス、複数の専門分野をまたがるプログラミングの応用方法など、Dynamo プラットフォームに関するさまざまな情報が記載されています。

Dynamo は、広範囲にわたる設計関連ワークフローにおいてその真価を発揮します。Dynamo は、さまざまな方法で使用することができます。

- ビジュアルプログラミングをはじめて体験する
- さまざまなソフトウェアのワークフローを接続する
- さまざまなユーザや開発者が積極的に意見を交換するコミュニティに参加する
- 継続的な改善を目指してオープンソースのプラットフォームを開発する

こうして Dynamo の開発を続けていくうちに、適切な資料が必要になりました。その過程で作成されたのが、この Dynamo Primer です。

この手引きは、Mode Lab で開発された各章から構成されています。各章では、Dynamo を使用してビジュアルプログラミング開発を行うために必要な基本的知識と、その知識を応用するための重要な情報を紹介しています。この手引には、次のトピックが記載されています。

- コンテキスト - ビジュアルプログラミングの定義と、Dynamo を使用するために理解しておく必要がある概念
- 作業の開始 - Dynamo の入手方法と、最初のプログラムの作成方法
- プログラムの内容 - Dynamo の各部の機能とその使用方法
- 構成要素 - データの定義と、プログラムで使用できる基本的なタイプ
- ジオメトリによる設計 - Dynamo でジオメトリ要素を操作する方法
- リストの使用方法 - データ構造の管理方法と調整方法
- ノード内のコード - 独自のコードを使用して Dynamo を拡張する方法
- 計算に基づく BIM - Dynamo と Revit モデルを組み合わせて使用する方法
- カスタムノード - 独自のノードを作成する方法
- パッケージ - 自作のツールをコミュニティで共有する方法

この手引で Dynamo について学習し、実際に使用して、開発プロジェクトに参加してください。

---

#### オープンソース

Dynamo Primer は、オープンソースプロジェクトです。高品質なコンテンツを提供するには、ユーザの皆様からのフィードバックが必要です。問題を発見した場合は、GitHub のバグレポートページに投稿してください。<https://github.com/DynamoDS/DynamoPrimer/issues>

この Dynamo Primer に新しいセクションを追加する場合や、既存の内容を編集する場合は、GitHub のリポジトリにアクセスしてください。<https://github.com/DynamoDS/DynamoPrimer>

---

## Dynamo Primer プロジェクトの概要

Dynamo Primer は、オートデスクの Matt Jezyk 氏と Dynamo 開発チームによって開始されたオープンソースプロジェクトです。

Dynamo Primer の初版は、Mode Lab によって作成されました。このような貴重なリソースの確立に取り組んでいただいたすべての方に感謝いたします。



この手引きは、Dynamo 2.0 の改訂を反映するため、Parallax Team の John Pierson 氏によって更新されました。



## 謝辞

Dynamo プロジェクトの創立と指揮に携わった Ian Keough 氏に、深く感謝いたします。

さまざまな Dynamo プロジェクトで積極的に協力していただいた Matt Jezyk 氏、Ian Keough 氏、Zach Kron 氏、Racel Williams 氏、Colin McCrone 氏にも感謝を申し上げます。

## ソフトウェアとリソース

**Dynamo:** 現在の公式リリースのバージョンは 2.0 です。

<http://dynamobim.com/download/>、または <http://dynamobuilds.com> (英語)

**DynamoBIM:** 追加情報、ラーニング コンテンツ、フォーラムについては、DynamoBIM の Web サイトを参照してください。

<http://dynamobim.org> (英語)

**Dynamo GitHub:** Dynamo は、GitHub 上で開発されたオープンソースプロジェクトです。DynamoDS を確認して、開発プロジェクトに参加してください。

<https://github.com/DynamoDS/Dynamo> (英語)

お問い合わせ先: このドキュメントに関する問題については、次の窓口にご連絡ください。

Dynamo@autodesk.com

## License

Copyright 2018 Autodesk

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at

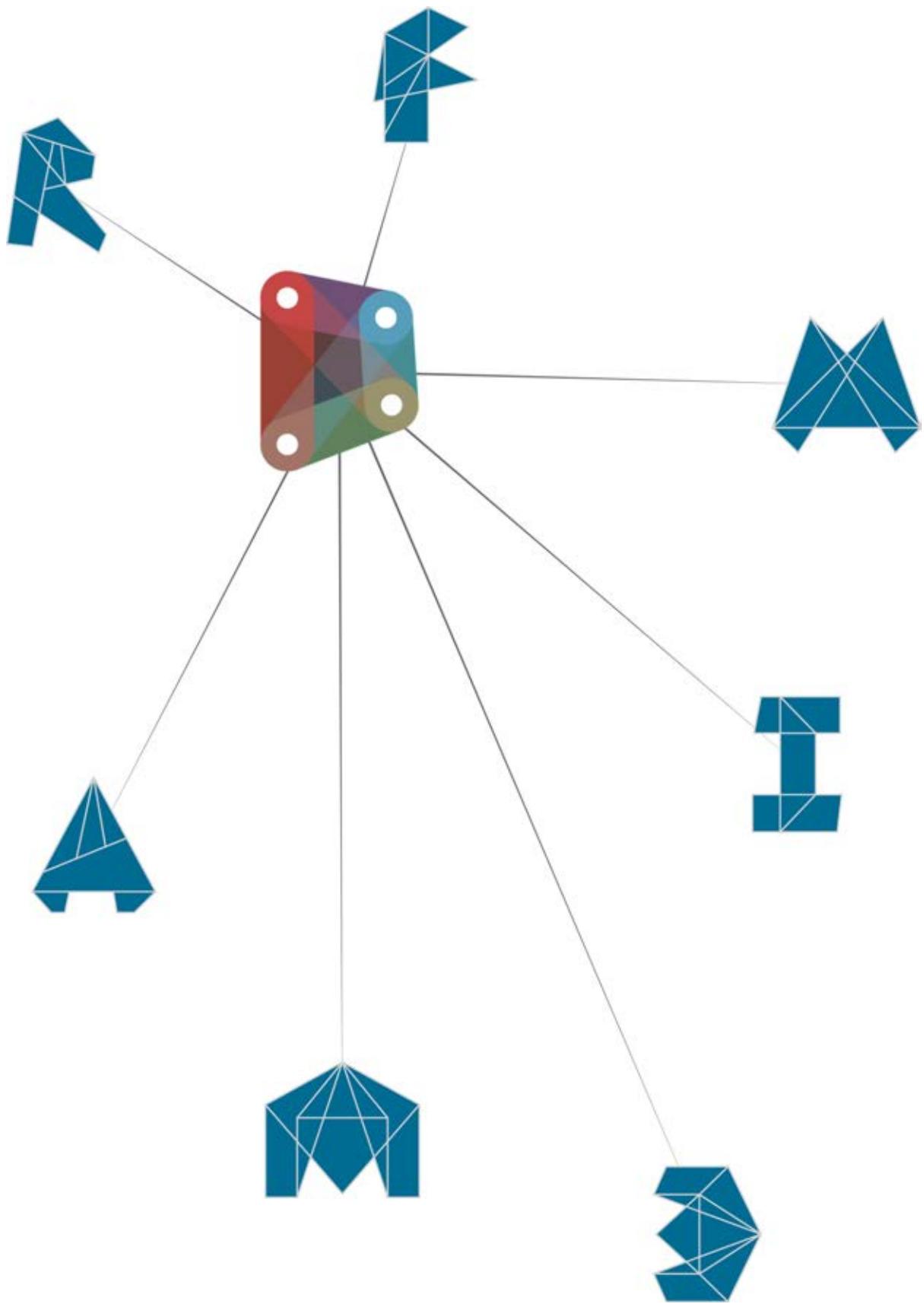
<http://www.apache.org/licenses/LICENSE-2.0>

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

## はじめに

### はじめに

Revit のビルディング インフォメーション モデリング(BIM)用アドオンとして開発が始まった Dynamo は、さまざまな機能を持つソフトウェアに成長しました。設計者は、開発プラットフォームである Dynamo を使用して問題を解決し、独自のツールを作成することができます。では、コンテキストを設定して、Dynamo を使用してみましょう。このセクションでは、Dynamo の概要と使用方法について説明します。



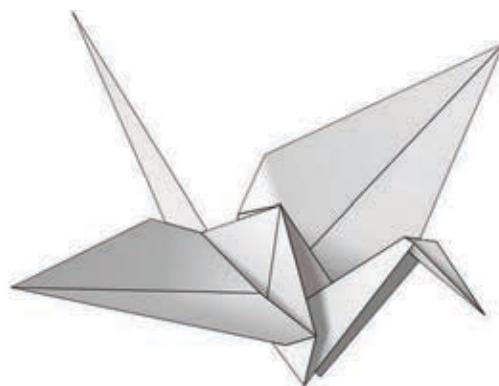
# ビジュアル プログラミングの概要

## ビジュアル プログラミングの概要

設計作業では、多くの場合、設計要素間の視覚的な関係、システム的な関係、幾何学的な関係を決定するという作業が発生します。通常、こうした関係は、ルールに従って概念から具体的な成果を生み出すワークフローによって構築されていきます。私たちは、アルゴリズムを意識することなく、アルゴリズムに従って毎日の業務を行っています。たとえば、入力、処理、出力という基本的なロジックに従い、段階的に業務を行っています。プログラミングでアルゴリズムを定式化することにより、こうした方法で業務を継続的に行うことができるようになります。

## 折り鶴で確かめるアルゴリズム

アルゴリズムは非常に便利な手法ですが、誤って理解される場合もあります。アルゴリズムは、予想もできない素晴らしいものを生み出す可能性を秘めていますが、決して魔法ではありません。実際に、アルゴリズム自体は非常に単純なものです。具体的な例として、折り鶴について考えてみましょう。最初に正方形の紙を用意します。これを一連の手順に従って折っていくと、折り鶴が出来上がります。この場合、紙が「入力」、折る手順が「処理」、出来上がった折り鶴が「出力」になります。

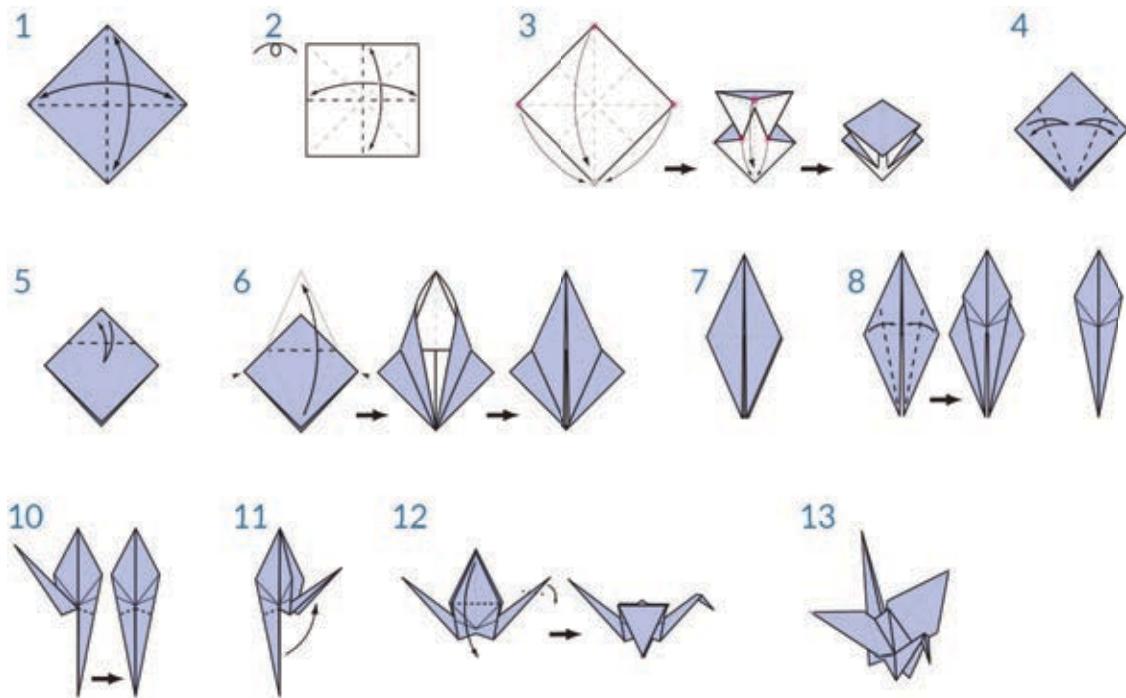


では、折り鶴のどこにアルゴリズムがあるのでしょうか。アルゴリズムは一連の抽象的な手順であり、テキストとグラフィックスのどちらでも表現することができます。

### テキストによる手順の説明:

1. 正方形の紙を用意し、色が付いている面を表にして置きます。対角線で2つ折りにしてから元に戻します。次に、もう一方の対角線で2つ折りにしてから元に戻します。
2. 紙を裏返し、白い面を表にして置きます。正方形の両辺を2つ折りにして、折筋をつけてから元に戻します。次に、別の方に向に2つ折りにしてから元に戻します。
3. ここまでにつけた折筋を使って、紙の上部の3つの角を下の角に合わせ、正方形に折りつぶします。
4. 手前の2つの三角形の両側下辺を中心線に合わせて折り、元に戻します。
5. 上の角を中央に合わせて、折筋をつけてから元に戻します。
6. 手前の袋の部分を開いて持ち上げながら、両端を中央へ向かって押しつぶし、折筋をつけます。
7. 紙を裏返して、手順4～6を繰り返します。
8. 上の頂点を中央に合わせて折ってから元に戻します。
9. 反対側でも同じ手順を繰り返します。
10. 左右の「脚」を上に向かって折り、折筋をつけてから元に戻します。
11. 上の手順でつけた折筋に沿って、左右の「脚」を中割り折ります。
12. 一方を中割り折りして鶴の頭を作り、羽の部分を引き下げて開きます。
13. これで折り鶴が出来上がりました。

### グラフィックスによる手順の説明



## プログラミングとは

どちらの手順に従って折っても、折り鶴が出来上がります。実際に折ってみると、アルゴリズムに従って折り鶴を作ったことになります。唯一異なるのは、定式化された一連の手順を読む方法です。これが、プログラミングを行う理由になります。コンピュータ プログラミング(通常は、省略してプログラミングといいます)とは、一連の処理を実行可能なプログラムとして定式化する行為のことです。上記の鶴の折り方をコンピュータで読み取って実行できる形式に変換した場合、プログラミングを行っていることになります。

プログラミングで最初に直面する重要な課題は、コンピュータと効率的なコミュニケーションを行うために、特定の抽象化形式を使用しなければならないということです。その抽象化の形式として、JavaScript、Python、Cなどのプログラミング言語が使用されています。先ほどの鶴の折り方のように反復可能な手順を作成できれば、後はそれをコンピュータ用に翻訳するだけで済みます。近いうちに、コンピュータで折り鶴を作るだけでなく、それぞれ微妙に異なる多数の折り鶴を作ることもできるようになるでしょう。これがプログラミングの能力です。どんなタスクをコンピュータに割り当てても、遅延やヒューマン エラーを発生させることなくタスクを繰り返し実行することができます。

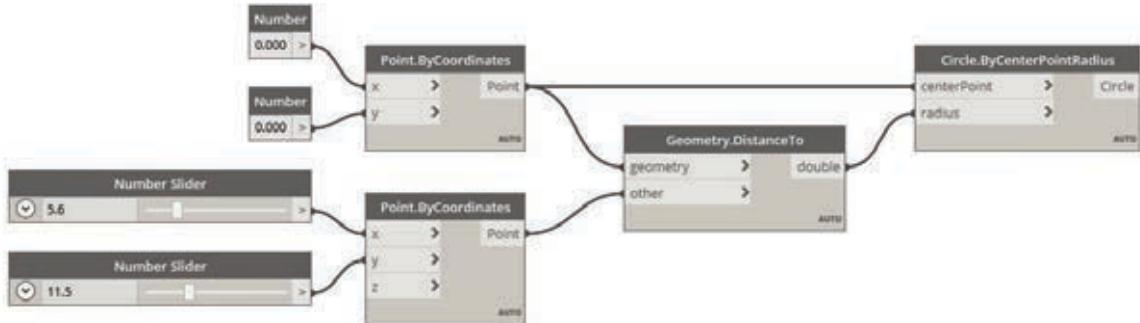
## ビジュアル プログラミングとは

この演習に付属しているサンプル ファイル [Visual Programming - Circle Through Point.dyn](#) をダウンロードしてください(右クリックして[名前を付けてリンク先を保存]を選択)。すべてのサンプルファイルの一覧については、付録を参照してください。

折り鶴の作り方を記述するタスクを割り当られたとしたら、どのようにしてそのタスクを実行しますか。この場合、グラフィックスで手順を記述する方法、テキストで手順を記述する方法、グラフィックスとテキストを組み合わせて手順を記述する方法が考えられます。

グラフィックスで手順を記述する場合、またはグラフィックスとテキストを組み合わせて手順を記述する場合、ビジュアル プログラミングが最適な方法です。通常のプログラミングとビジュアル プログラミングのどちらの場合も、基本的なプロセスは同じです。どちらも同じ定式化の枠組みを使用しますが、ビジュアル プログラミングの場合は、視覚的なグラフィカル ユーザ インタフェースを使用して、プログラムの指示や関係を定義します。この場合、構文に従ってテキストを入力するのではなく、事前にパッケージ化されたノードを相互に接続します。ここで、同じアルゴリズムを使用して、ビジュアル プログラミングとテキストによるプログラミングを比較してみましょう。「指定された点を通過する円弧を描画する」という命令を、ノードを使用してプログラミングする場合の例と、コードを使用してプログラミングする場合の例を示します。

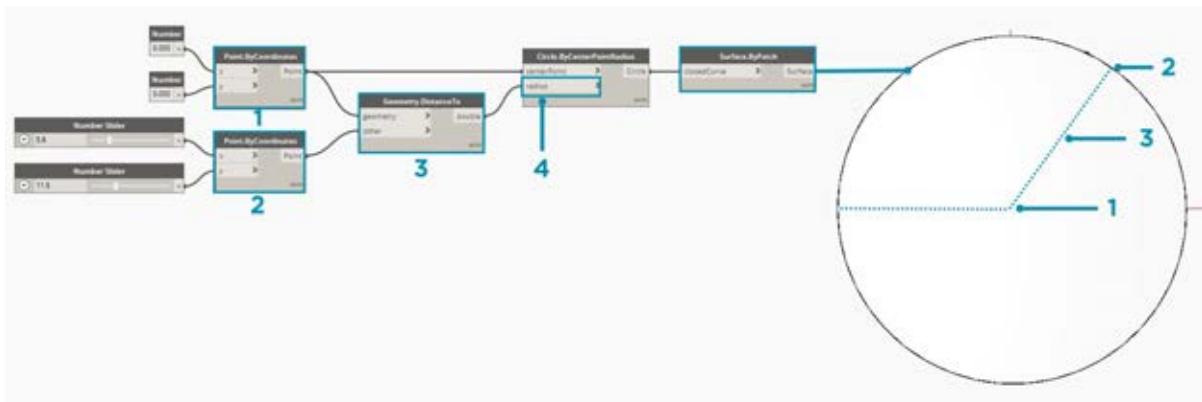
## ビジュアル プログラミング:



テキストによるプログラミング:

```
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);
```

このアルゴリズムの結果は、次のようにになります。



このように、ビジュアル プログラミングには、視覚的に理解しやすいという特徴があります。そのため、最初から簡単にプログラミングを実行することができ、設計者にとっても便利な仕様になっています。Dynamo はビジュアル プログラミングの範疇に分類されますが、アプリケーション内でテキストを使用してプログラミングを行うこともできます。これについては、後で説明します。

# Dynamo とは

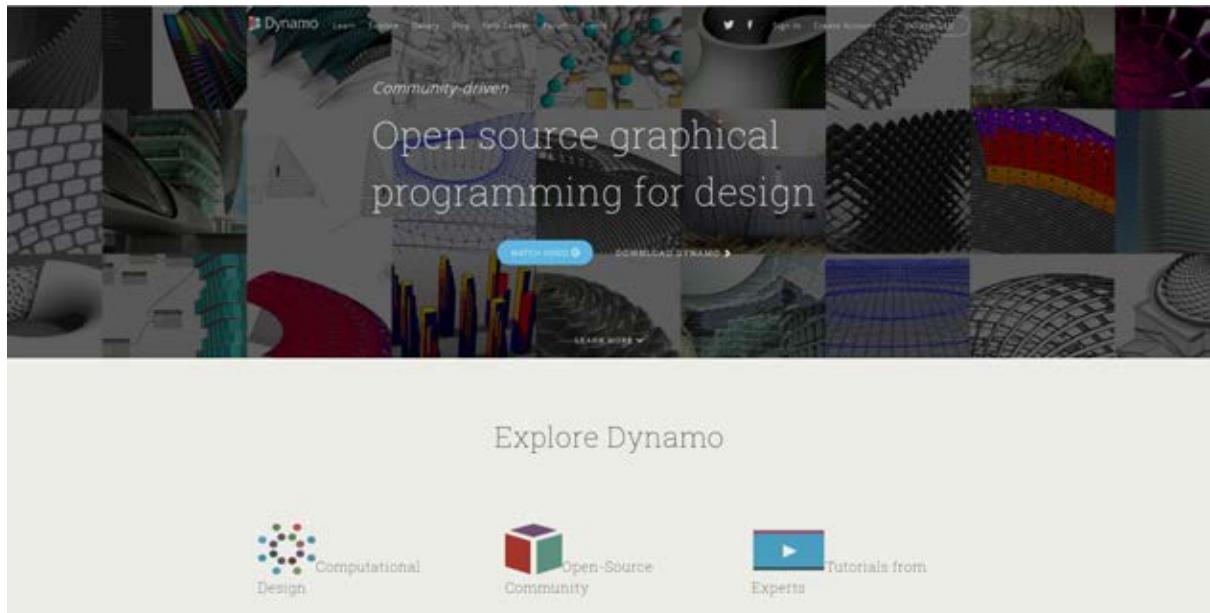
## Dynamo とは

Dynamo は「発電機」という意味ですが、Dynamo はその名前のとおり、プログラミング作業の動力となるツールです。Dynamo で作業する場合、アプリケーションを単独で使用する場合もあれば、オートデスクの他のソフトウェアと組み合わせて使用する場合もあります。また、ビジュアル プログラミングのプロセスを実行する場合もあれば、国際的なユーザや開発者のコミュニティに参加する場合もあります。

### アプリケーション

Dynamo は、スタンドアロンの「サンドボックス」モードで実行可能なソフトウェアであり、Revit や Maya などの他のソフトウェアのプラグインとしても使用することができます。簡単に説明すると、次のようにになります。

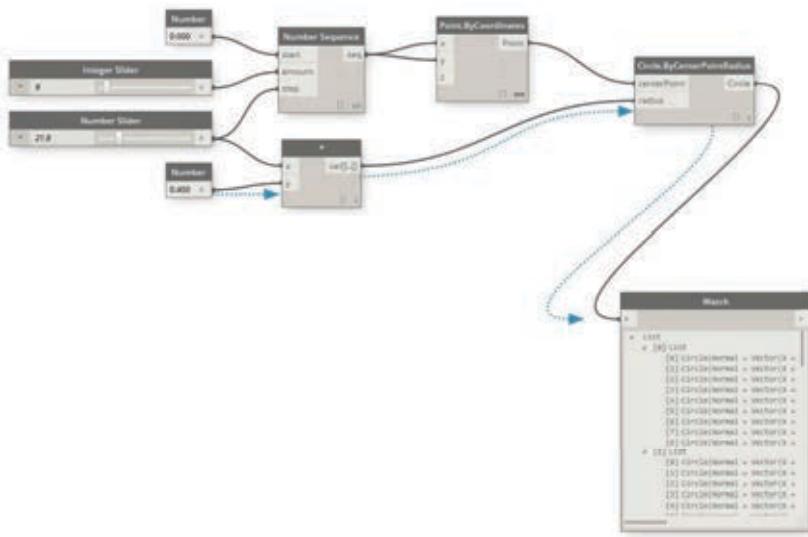
Dynamo は、プログラミング経験の有無にかかわらず、だれでも簡単に使用できることを目的として開発されたビジュアル プログラミング ツールです。このツールを使用すると、アプリケーションの動作を視覚的にスクリプト化したり、カスタムのロジックを定義したり、さまざまなテキストベースのプログラミング言語をスクリプト化することができます。



1. Dynamo を Revit と組み合わせて使用する場合について紹介しています。
2. インストーラをダウンロードすることができます。

## プロセス

Dynamo をインストールすると、ビジュアル プログラミングを行いながら、さまざまな要素を互いに接続し、動作の関係と順序を定義して、カスタマイズされたアルゴリズムを構築できるようになります。構築したアルゴリズムは、データの処理からジオメトリの生成まで、さまざまな用途で使用することができます。コードを 1 行も記述する必要がなく、すべてがリアルタイムで実行されます。



ビジュアル プログラミングを開始するには、最初に各要素を追加して接続する必要があります。

## コミュニティ

現在の Dynamo があるのは、熱心なユーザと開発者のコミュニティによって支えられてきたからです。開発チームのブログを読む、作品をギャラリーに投稿する、フォーラムで Dynamo について議論するなどの方法で、ぜひこのコミュニティに参加してください。

The screenshot shows the homepage of the Dynamo Forum. At the top, there's a navigation bar with links for Learn, Support, Gallery, Shop, Help Center, Forum, and a search bar. Below the navigation, there's a section for 'all categories' with a 'Listed' button, followed by 'Top' and 'Categories' links. The main content area is a table listing forum topics. The columns are: Topic, Category, User, Replies, Views, and Activity. The topics listed include:

- # Dynamo and IronPython - incompatibilities with other applications (Category: FAQ, User: 6, Replies: 230, Views: 119)
- # How to get help on the Dynamo forums (Category: FAQ, User: 4, Replies: 926, Views: 117)
- # Welcome to the Dynamo Forum (Category: FAQ, User: 21, Replies: 734, Views: 117)
- # Place multiple views on multiple sheets by coordinate (Category: Best, User: 82, Replies: 406, Views: 116)

Below the table, there's a section for 'Testing' with topics like 'Naming Rooms from Area Plans' and 'Zero touch node: how to create a package using visual studio'. There's also a section for 'The Randomized Panel Problem' and 'Export schedule columns to Python array'. At the bottom, there's a section for 'Model Groups from Linked File' and 'Count Wall Types to Excel'.

## プラットフォーム

Dynamo は、設計者向けのビジュアル プログラミング ツールとして設計されており、外部ライブラリを使用するツールや、API をサポートしているオートデスク製品を使用するツールを作成することができます。Dynamo Studio では、「サンドボックス」スタイルでプログラムを開発することができます。Dynamo を取り巻くユーザや開発者のコミュニティは拡大を続けています。

プロジェクトのソースコードはオープンソースとして公開されているため、個々のニーズに合わせて機能を拡張することができます。GitHub のプロジェクトにアクセスして、ユーザによる Dynamo のカスタマイズ作業の進行状況を確認してください。

S Features Business Explore Marketplace Pricing This repository Search Sign in Sign up

DynamoDS / Dynamo

Watch 163 Star 567 Fork 339

Code Issues 630 Pull requests 9 Projects 2 Wiki Insights

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

28,546 commits 70 branches 0 releases 69 contributors

Branch: master New pull request Find file Close or download

ramramps Merge pull request #8714 from ramramps/custom-node-crash-fix 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

BUILD PASSING build passing



# Dynamo

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!](http://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.

プロジェクトにアクセスしてプロジェクトをフォークし、Dynamo をニーズに合わせて拡張することができます。

## DYNAMO IN ACTION

### DYNAMO IN ACTION

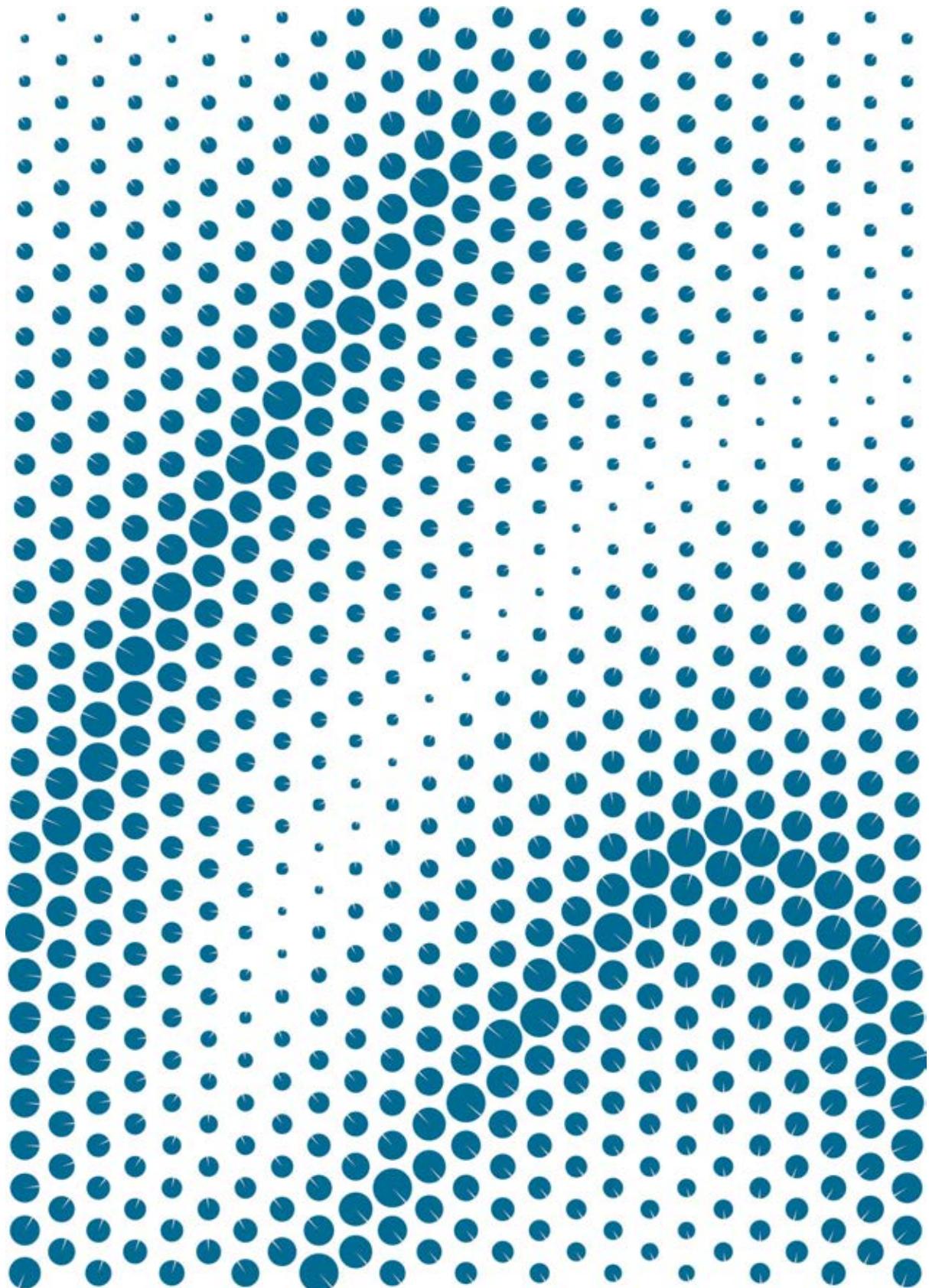
プロジェクトのワークフローにおけるビジュアル プログラミングの活用から、カスタマイズされたツールの開発まで、幅広い用途で使用できる Dynamo は、アプリケーションの開発に欠かせないツールです。

[Pinterest で Dynamo in Action のボードをフォローしてください。](#)

## **Dynamo の概要**

### **Dynamo の概要**

Dynamo は、ビジュアル プログラミング向けのプラットフォームであり、拡張性に優れた柔軟な設計ツールです。Dynamo は、スタンドアロンのアプリケーションとしても、他の設計ソフトウェアのアドオンとしても使用できるため、クリエイティブな各種ワークフローを開発することができます。Dynamo をインストールしたら、最初にインターフェースの重要な特徴から確認していきましょう。



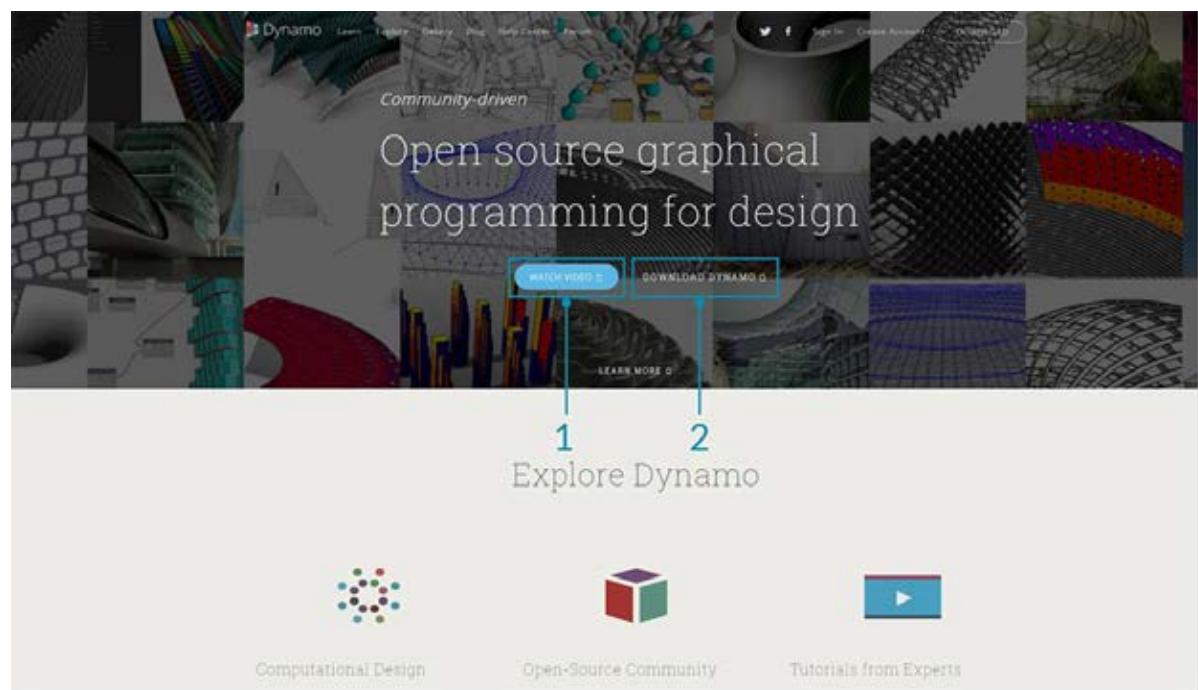
# Dynamo のインストールと起動

## Dynamo のインストールと起動

Dynamo は、オープンソースのプロジェクトとして活発に開発が進められているため、ダウンロードできるインストーラとしては、公式リリースと毎日更新されるプレリースの 2 種類が用意されています。Dynamo を初めて使用する場合は、公式リリースをダウンロードしてください。また、プレリースや GitHub プロジェクトを通じて、ぜひ Dynamo の開発にご協力ください。

### ダウンロード

Dynamo の公式リリースをダウンロードするには、[Dynamo の Web サイト](#)にアクセスします。トップページかダウンロード ページにアクセスしてダウンロード ボタンをクリックすると、ダウンロードがすぐに開始されます。



1. Dynamo を活用した建築意匠の計算設計に関するビデオを見ることができます。
2. ダウンロード ページにアクセスできます。

[Dynamo Github](#) プロジェクトにアクセスすると、プレリースをダウンロードして開発中の最新機能を確認することができます。



**Dynamo**

Open-source Dynamo is a visual programming extension for Autodesk® Revit™ that allows you to manipulate data, select 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

**1**

Lesson 1.32

**DYNAMO STUDIO**

Autodesk® Dynamo Studio is a visual programming platform that functions fully independently of any other application. Embrace 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

**2**

Version 1.0.0 (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.

**2**

DynamoRevit2.0.0.20180404170007.exe  
Dynamostudio2.0.0.20180404170007.exe  
Dynamomodel2.0.0.20180404170007.exe

**3**

## 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

**3**

**Get Involved with Open Source**

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

**4**

1001256 package downloads | 1258 packages | 461 authors

1. 公式リリースのインストーラをダウンロード
2. プレリリースのインストーラをダウンロード
3. 開発者のコミュニティからカスタム パッケージを確認してください。
4. GitHub で Dynamo の開発プロジェクトに参加できます。

## インストール

ダウンロードしたインストーラの保存フォルダを開き、.exe ファイルを実行します。インストール時に、[Setup]ダイアログでインストールするコンポーネントをカスタマイズすることができます。

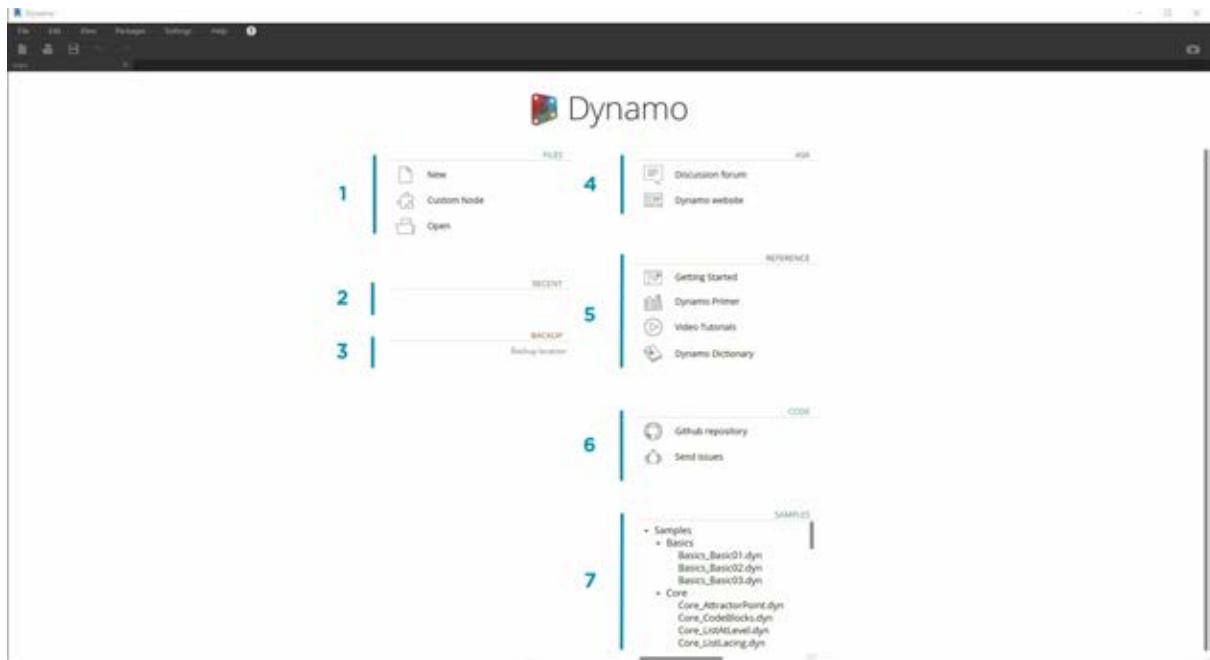


1. インストールするコンポーネントを選択します。

ここで、Revit などの他のインストール済みアプリケーションに Dynamo を接続するコンポーネントをインストールするかどうかを決定する必要があります。Dynamo プラットフォームの詳細については、第 1 章 2 節を参照してください。

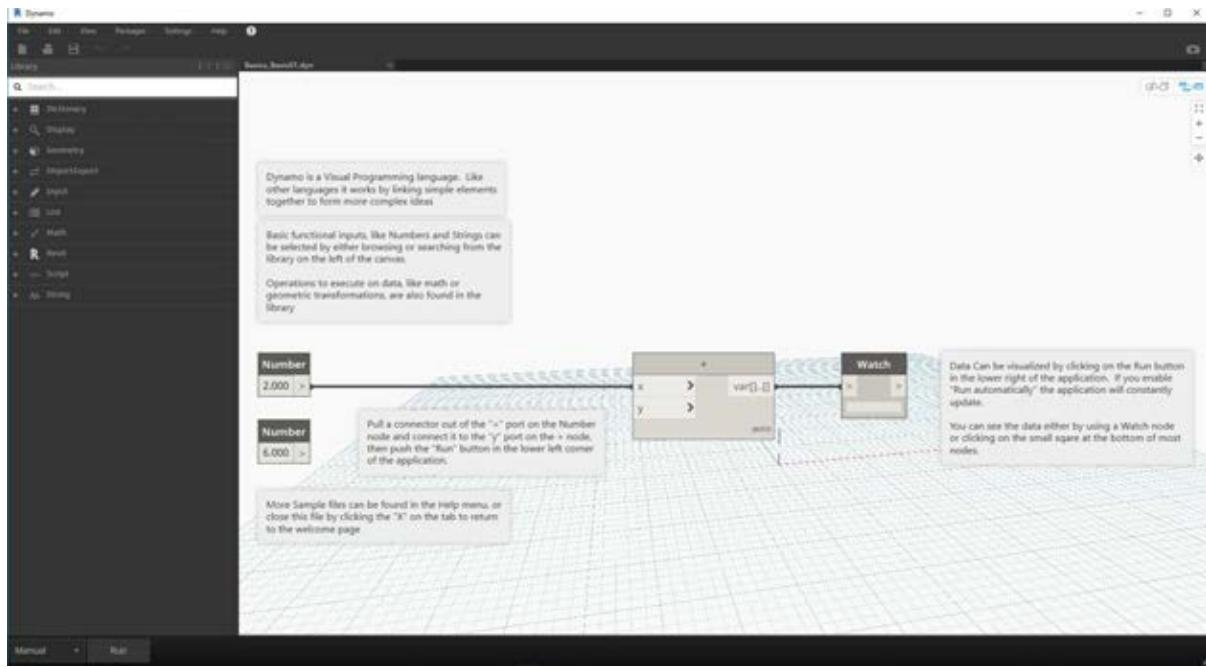
## 起動

Dynamo を起動するには、¥Program Files¥Dynamo¥Dynamo Revit¥x.y にアクセスして、DynamoSandbox.exe を選択します。これにより、Dynamo のスタンドアロン版が起動し、スタートページが表示されます。このページには、標準のメニューとツールバーの他に、ファイル、機能、追加リソースにアクセスするためのショートカットの一覧が表示されます。



1. [ファイル] - 新しいファイルを作成したり、既存のファイルを開くことができます。
2. [最近使用したファイル] - 最近使用したファイルが一覧表示されます。
3. [バックアップ] - バックアップにアクセスできます。
4. [確認] - ユーザ フォーラムと Dynamo Web サイトに直接アクセスできます。
5. [参照] - 追加の学習リソースを参照することができます。
6. [コード] - オープンソースの開発プロジェクトに参加できます。
7. [サンプル] - インストールに付属するサンプルを確認できます。

最初のサンプルファイルを開いてワークスペースを作成し、Dynamo が正しく動作することを確認します。[サンプル] > [基本機能] > [Basics\_Basic01.dyn] をクリックします。



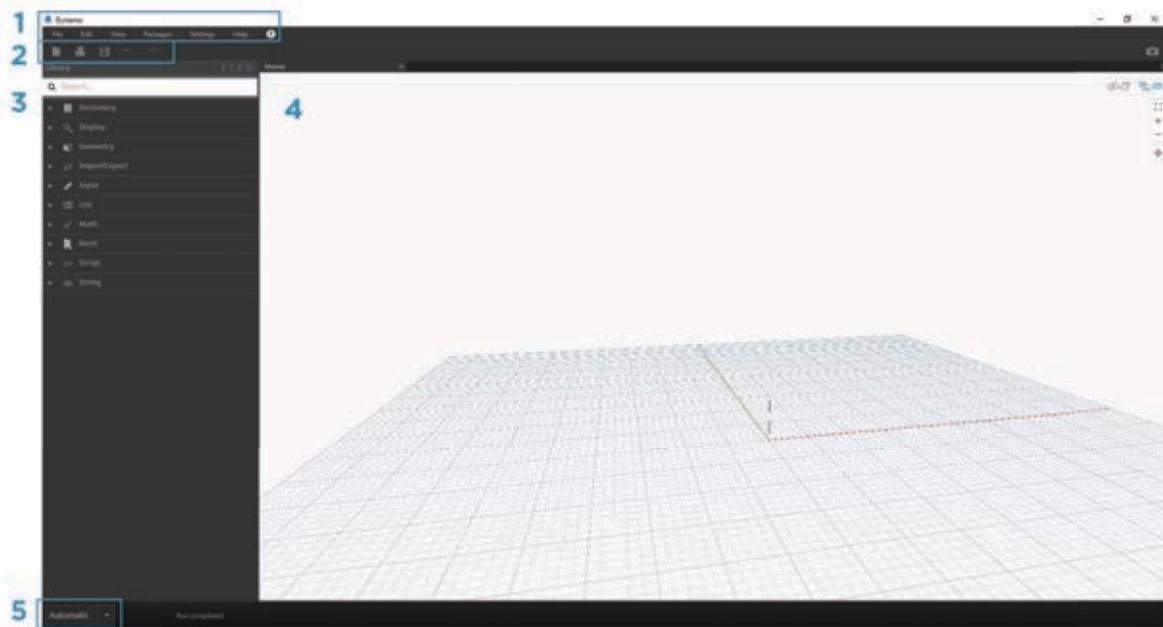
1. 拡張バーに[自動]と表示されていることを確認します。表示されていない場合は、[実行]をクリックします。
2. 画面上の指示に従って、Number ノードを [+] ノードに接続します。
3. Watch ノードに結果が表示されることを確認します。

このファイルが正常にロードされていれば、Dynamo でビジュアル プログラミングを開始することができます。

# Dynamo ユーザ インタフェース

## Dynamo ユーザ インタフェース

Dynamo のユーザ インタフェースは、5 つの主要な領域で構成されています。そのうち最も大きな領域は、ビジュアル プログラミングの構成に使用されるワークスペースです。

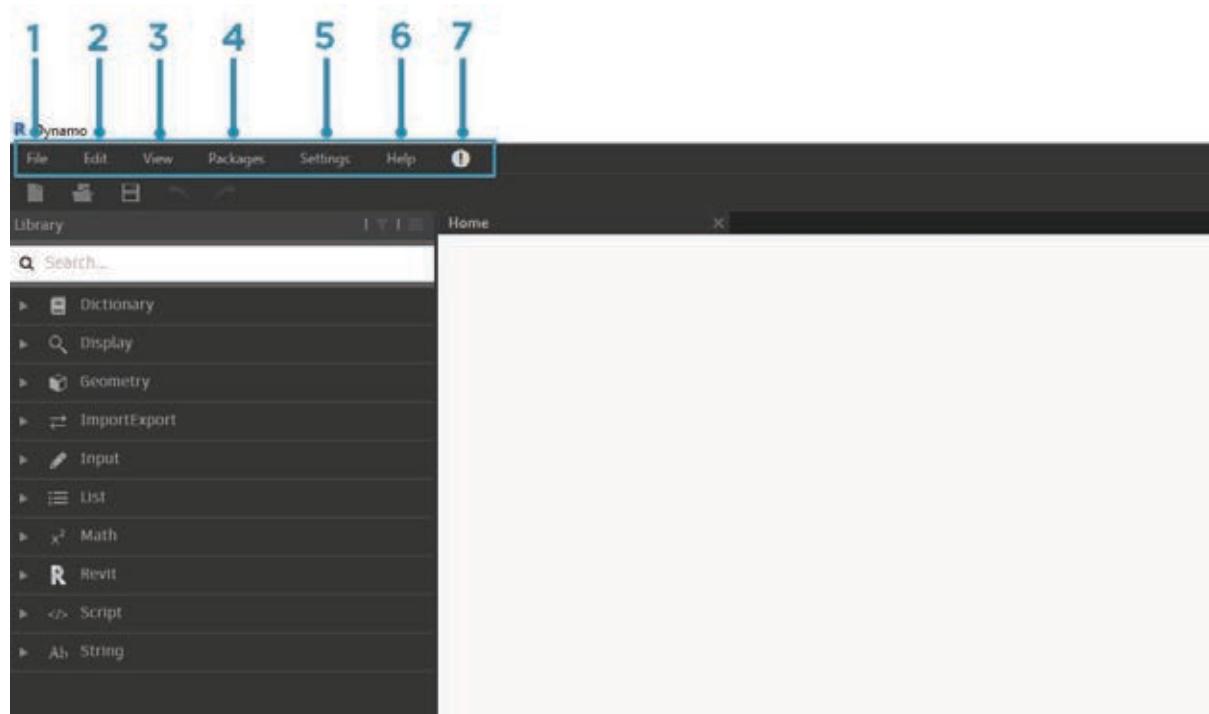


1. メニュー
2. ツールバー
3. ライブラリ
4. ワークスペース
5. 実行バー

ここからは、Dynamo の UI を確認しながら、各領域の機能を詳しく見ていきます。

#### メニュー

ドロップダウン メニューでは、Dynamo アプリケーションの基本的な機能の一部にアクセスすることができます。多くの Windows ソフトウェアと同様に、最初の 2 つのメニュー項目(左端のメニュー項目とその横のメニュー項目)で、ファイルの管理に関する操作や、選択とコンテンツ編集に関する操作を行うことができます。他のメニュー項目は、Dynamo 固有のメニュー項目です。

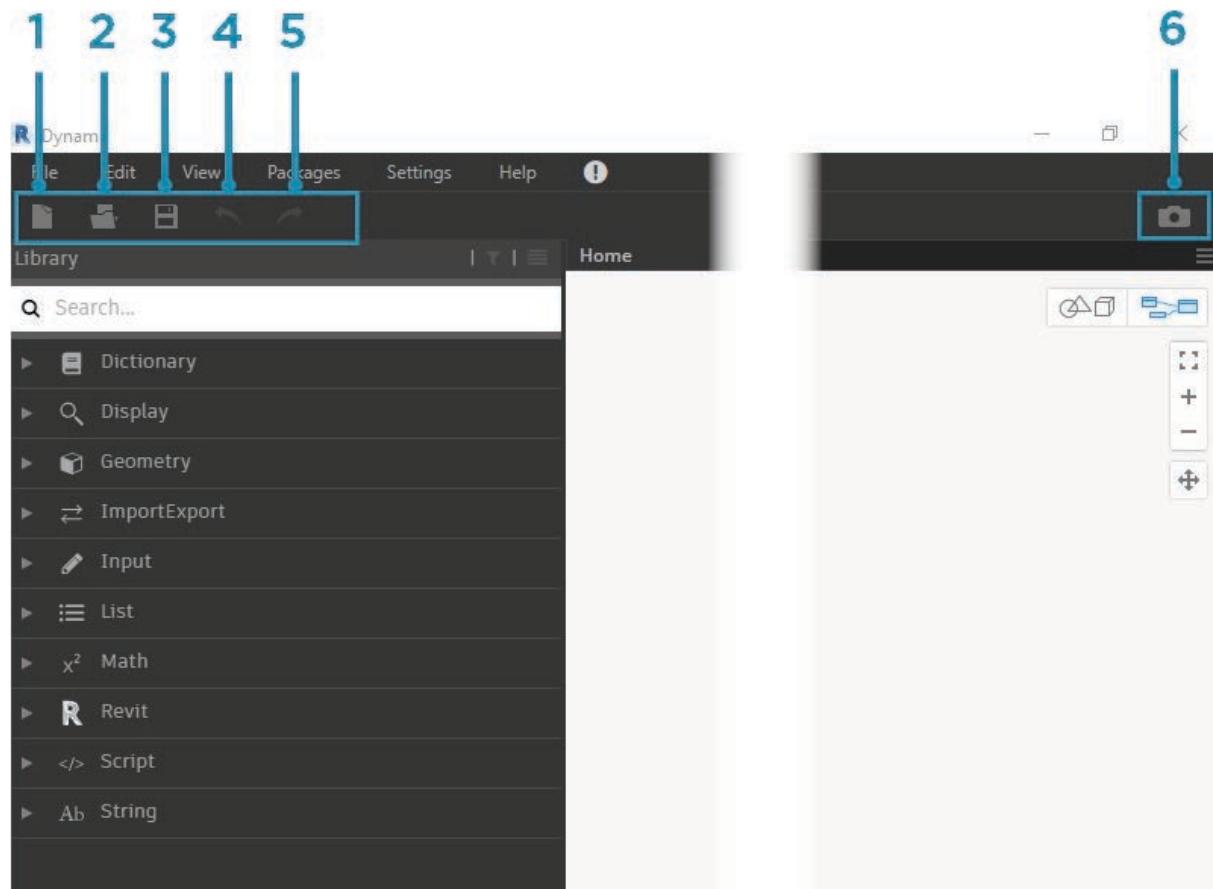


1. ファイル
2. 編集

3. ビュー
4. パッケージ
5. 設定
6. ヘルプ
7. 通知

### ツールバー

Dynamo ツールバーには、[元に戻す][Ctrl + Z]コマンドと[やり直し][Ctrl + Y]コマンドの他に、ファイルに関する作業に役立つ一連のクイック アクセス ボタンが用意されています。右端にあるボタンを使用すると、ワークスペースのスナップショットを書き出すことができます。このボタンは、ドキュメントの作成や共有を行う場合に特に便利です。



1. [新規] - 新しい .dyn ファイルを作成する場合に使用します。
2. [開く] - 既存の .dyn ファイル(ワークスペース)または .dyf ファイル(カスタム ノード)を開く場合に使用します。
3. [保存]/[名前を付けて保存] - アクティブな .dyn ファイルや .dyf ファイルを保存する場合に使用します。
4. [元に戻す] - 最後の操作を元に戻す場合に使用します。
5. [やり直し] - 次の操作をやり直す場合に使用します。
6. [ワークスペースをイメージとして書き出す] - 表示されているワークスペースを PNG ファイルに書き出す場合に使用します。

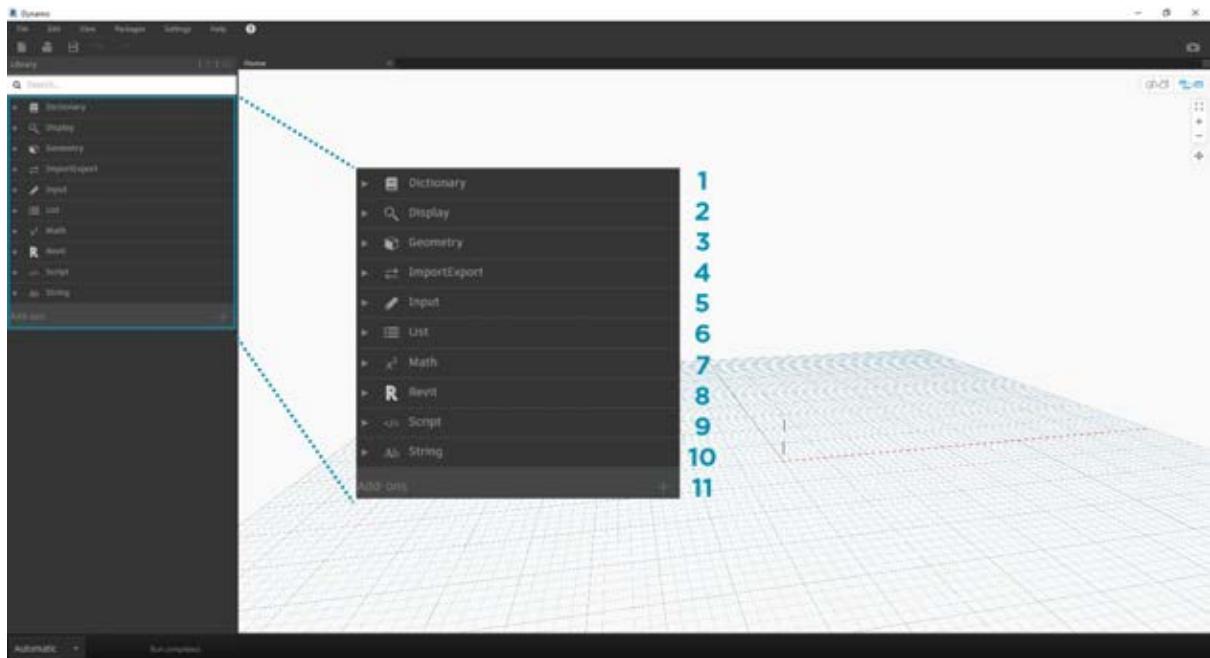
### ライブラリ

ライブラリには、付属の既定ノード、追加でロードされるカスタム ノードやパッケージなど、ロードされたすべてのノードが格納されます。ライブラリ内のノードは、そのノードがデータを作成するのか、アクションを実行するのか、データをクエリーするのかという区別に基づいて、ライブラリやカテゴリ内部で、(あるいは該当する場合)サブカテゴリ内部で、階層別に分類されます。

#### カテゴリを参照する

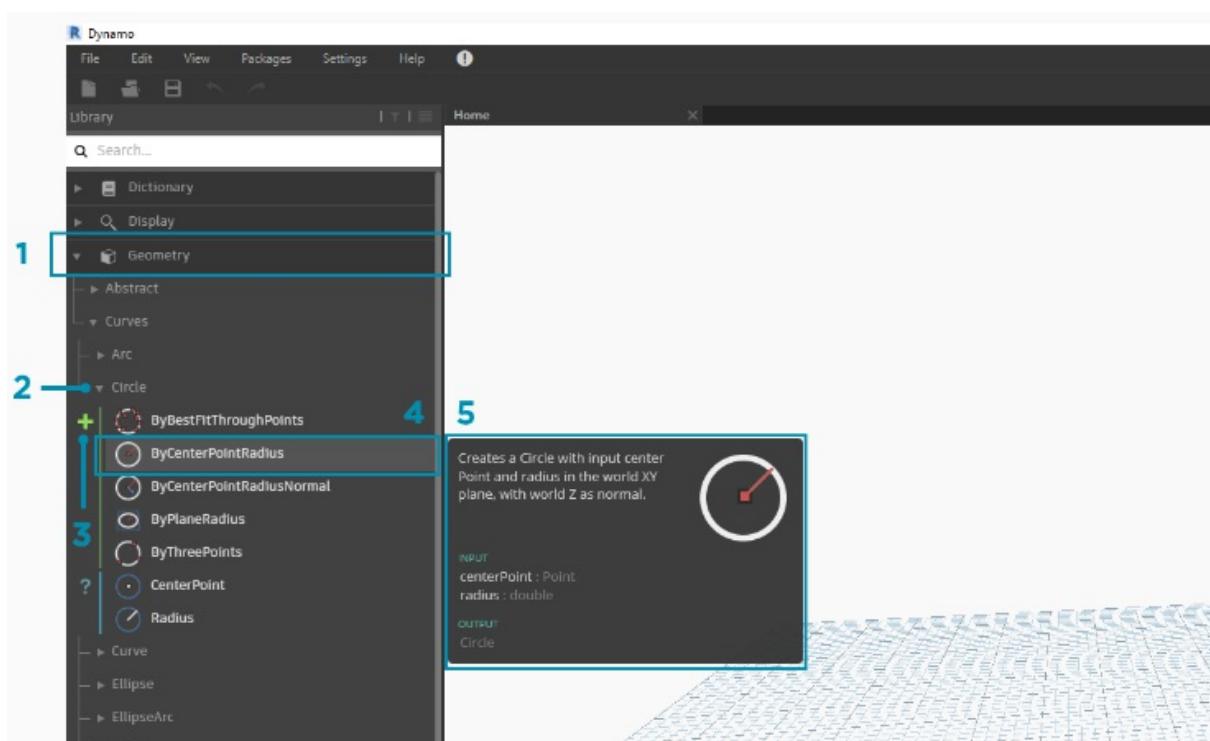
既定では、ライブラリにはノードのカテゴリが 8 つ存在します。[Core]と[Geometry]には最も多くのノードが含まれているため、ノードを探す場合は最初にこれらのカテゴリを使用することをお勧めします。これらのカテゴリを参照すると、ワークスペースに追加できるノードの階層や、これまでに使用したことがない新しいノードを探す最適な方法について、簡単に理解することができます。

ここでは、既定のノードを中心に確認しますが、後で、カスタムノード、追加のライブラリ、Package Manager を使用して、このライブラリを拡張していきます。



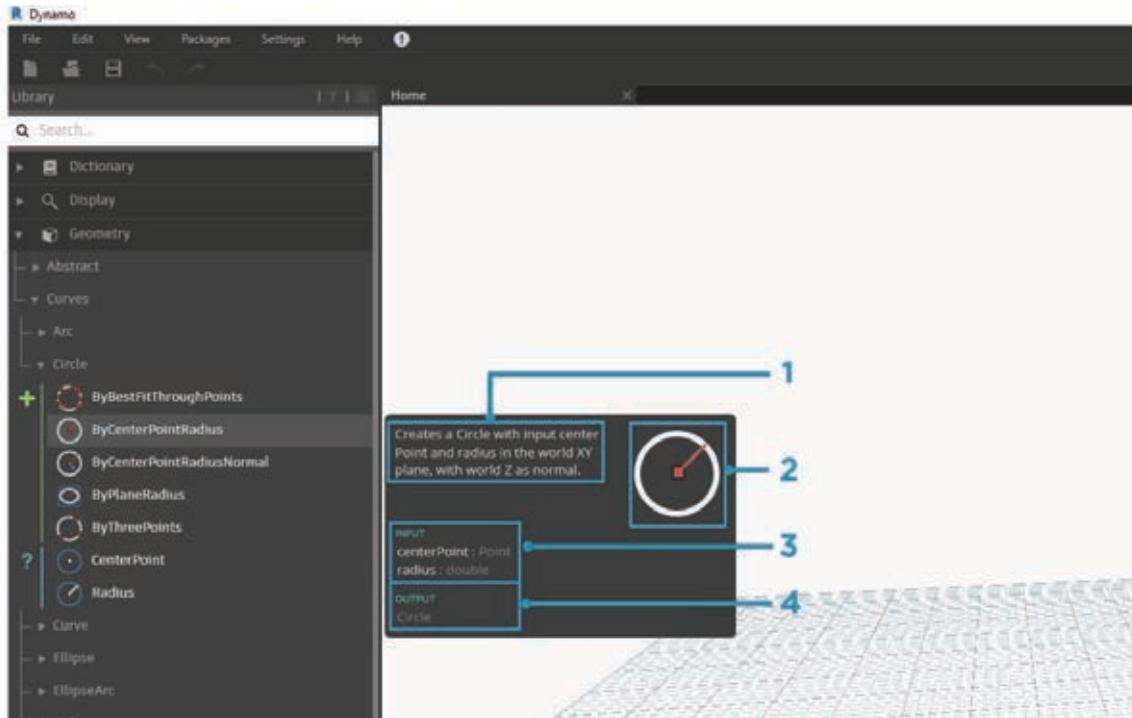
1. Dictionary
2. Display
3. Geometry
4. ImportExport
5. Input
6. List
7. Matches
8. Revit
9. Script
10. String
11. Add-ons

メニューで[Library]をクリックしてライブラリを参照し、[Geometry] > [Curves] > [Circle]の順にクリックします。新しいメニュー項目(具体的には、作成用の「+」ラベルとクエリー用の「?」ラベル)が表示されることを確認してください。



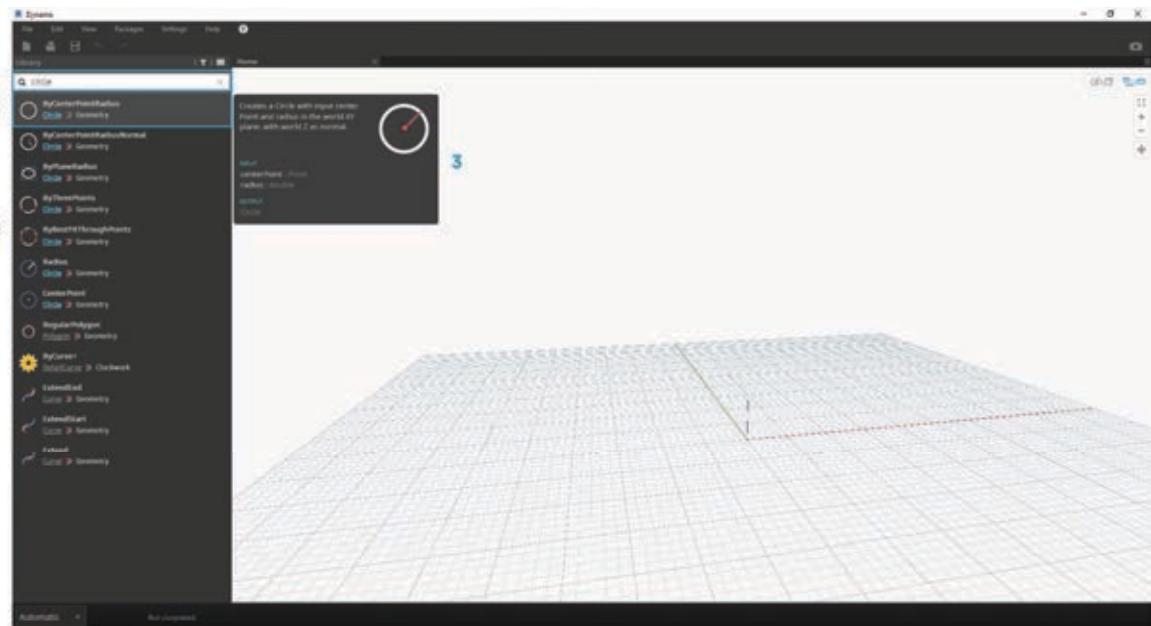
1. ライブラリ
2. カテゴリ
3. サブカテゴリ: 作成/アクション/クエリー
4. ノード
5. ノードの説明とプロパティ(ノードのアイコンの上にカーソルを置くと表示されます)

同じ[Circle]メニューで、**ByCenterPointRadius** の上にカーソルを置きます。ウインドウに、ノード名とアイコン以外の詳細情報が表示されます。これにより、そのノードの機能、必要な入力、生成される出力について、すばやく確認することができます。



#### ノードを検索する

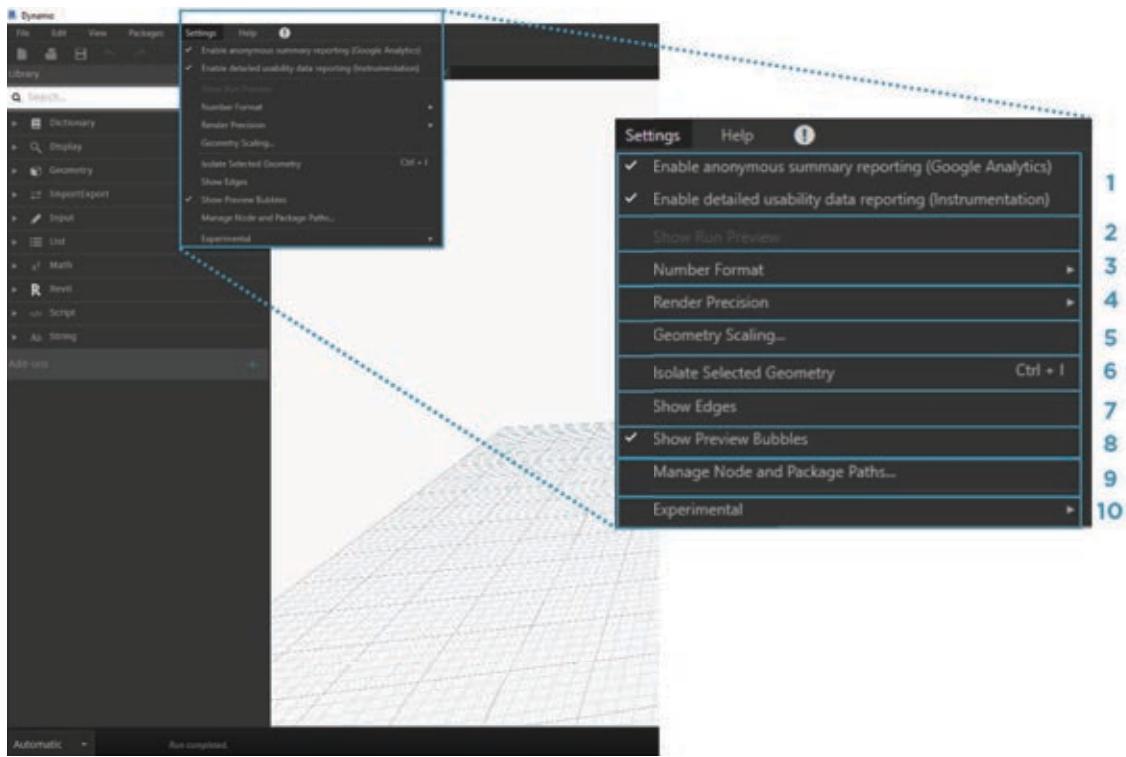
ワークスペースに追加するノードが具体的に決まっている場合は、[検索]フィールドを使用すると便利です。ワークスペースで設定の編集や値の指定を行っているとき以外は、このフィールドにカーソルが常に表示されます。検索する文字列を入力すると、その文字列に最も近いノードと他のノードのリストが、そのノードが格納されているカテゴリとともに、Dynamo ライブラリに表示されます。ブラウザで[Enter]キーを押すか対象の項目をクリックすると、ハイライト表示されたノードがワークスペースの中央に追加されます。



1. 検索フィールド
  2. 検索文字列に最も近いノード/選択されたノード
  3. 代替ノード

設定

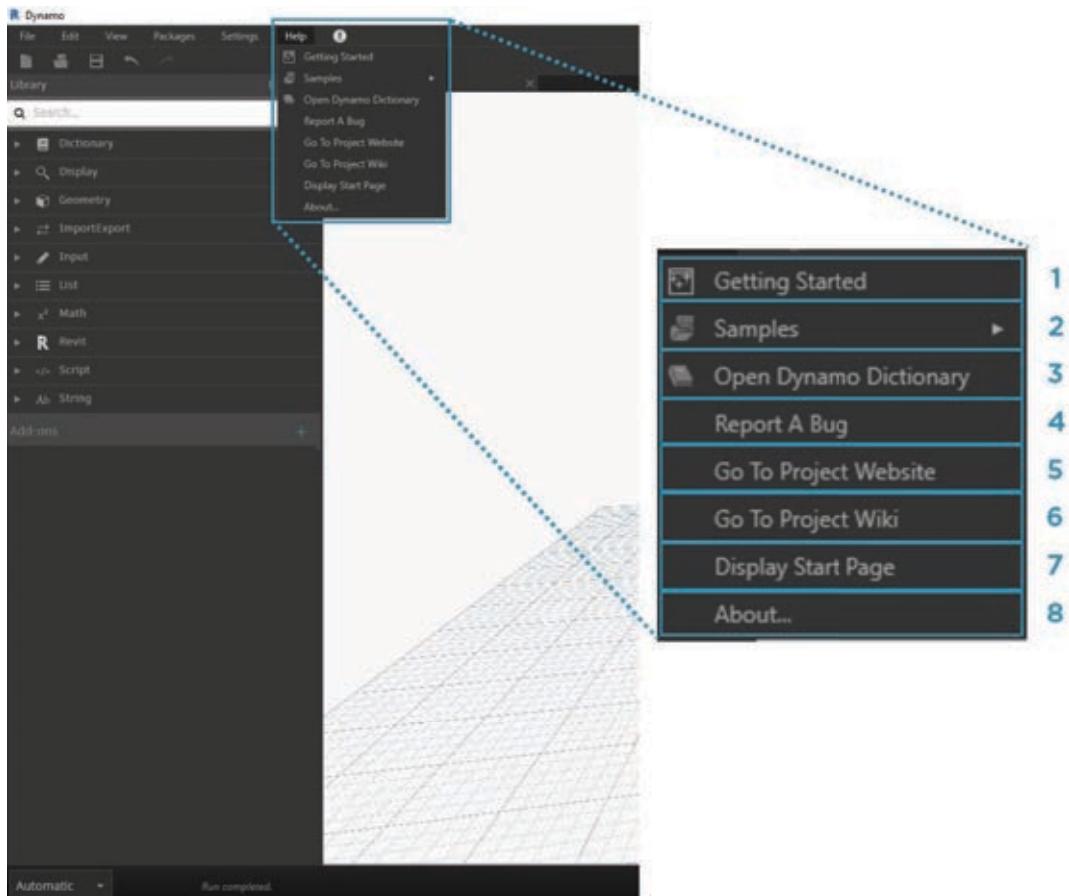
ジオメトリの設定オプションやユーザ設定オプションは、[設定]メニューに用意されています。このメニューを使用して、Dynamoを改善するためのユーザデータを共有したり、アプリケーションの小数点の精度とジオメトリのレンダリング品質を設定することができます。



1. [レポートを有効にする] - Dynamo の向上に使用されるユーザデータを送信するオプションです。
2. [実行のプレビューを表示] - グラフの実行状態をプレビューします。実行するようにスケジュール設定されているノードは、グラフ内でハイライト表示されます。
3. [数値形式] - 浮動小数点型の数値に関するドキュメント設定を変更します。
4. [レンダリング精度] - ドキュメントのレンダリング品質を調整します。
5. [ジオメトリのスケーリング] - 操作するジオメトリの範囲を選択します。
6. [選択したジオメトリを選択表示] - ノードの選択に基づいて背景のジオメトリを選択表示します。
7. [エッジを表示] - 3D ジオメトリのエッジの表示設定を切り替えます。
8. [プレビュー バルーンを非表示] - ノードの下に表示されるデータプレビュー バルーンを非表示にします。
9. [ノードとパッケージのパスを管理] - ノードとパッケージをライブラリに表示するのに使用されるファイルパスを管理します。
10. [サンプルの実験的機能を有効にする] - Dynamo に試験的に導入されているベータ版の機能を使用します。

## ヘルプ

不明な点については、[ヘルプ]メニューを確認します。このメニューを使用して、付属のサンプル ファイルを入手したり、既定のインターネット ブラウザで Dynamo の参照用 Web サイトにアクセスすることができます。必要な場合は、[バージョン情報]オプションを用いて、インストールされている Dynamo のバージョンを確認したり、Dynamo が最新の状態に更新されているかどうかを確認することができます。



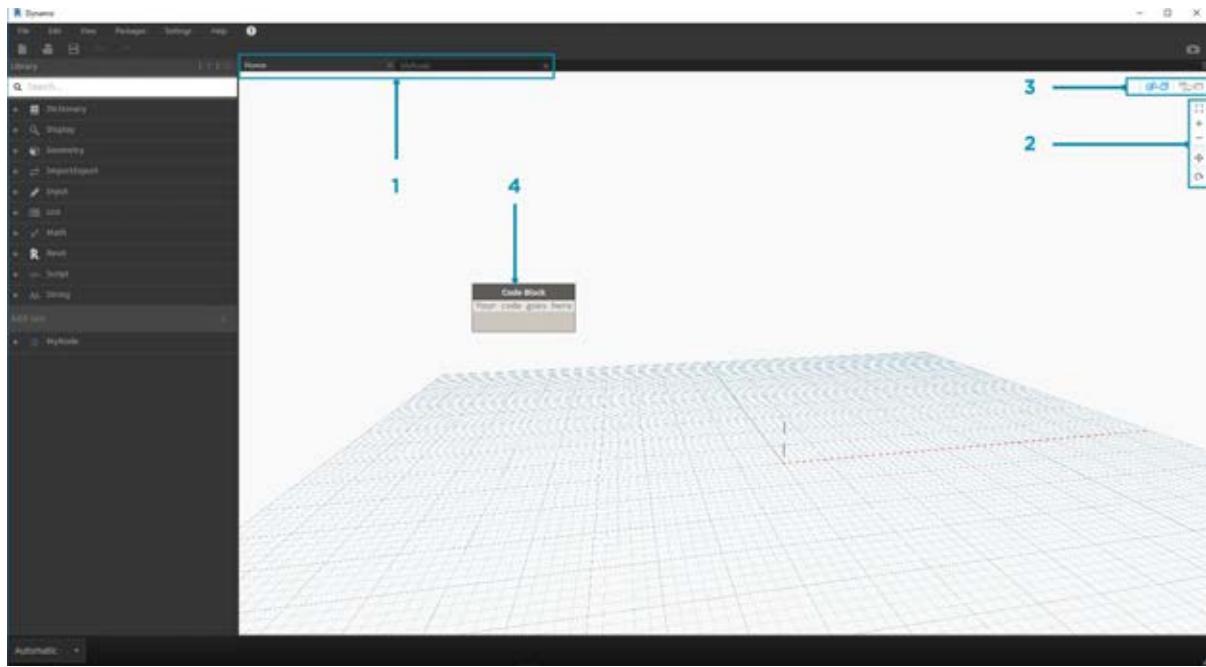
1. [スタートアップ ガイド] - Dynamo の使用方法を簡単に紹介しています。
2. [サンプル] - 参照用のサンプル ファイルです。
3. [Dynamo ディクショナリを開く] - すべてのノードについての設計図書作成で参照するリソースです。
4. [バグをレポート] - GitHub 上に Issue (イシュー)を作成します。
5. [プロジェクトの Web サイトに移動] - GitHub 上の Dynamo プロジェクトを表示します。
6. [プロジェクトの Wiki に移動] - Wiki にアクセスして、Dynamo API を使用する開発方法やライブラリとツールのサポートに関する情報を表示します。
7. [開始ページを表示] - ドキュメント内の Dynamo 開始ページに戻ります。
8. [バージョン情報...] - Dynamo のバージョン情報を確認します。

# ワークスペース

## ワークスペース

Dynamo のワークスペースでは、ビジュアル プログラミングを行うだけでなく、出力されるジオメトリのプレビューを確認することもできます。ホーム ワークスペースとカスタム ノードのどちらで作業しているかにかかわらず、マウスか画面右上のコントロール ボタンを使用してナビゲートすることができます。ナビゲート中にプレビュー モードを切り替えるには、画面右上のスイッチを使用します。

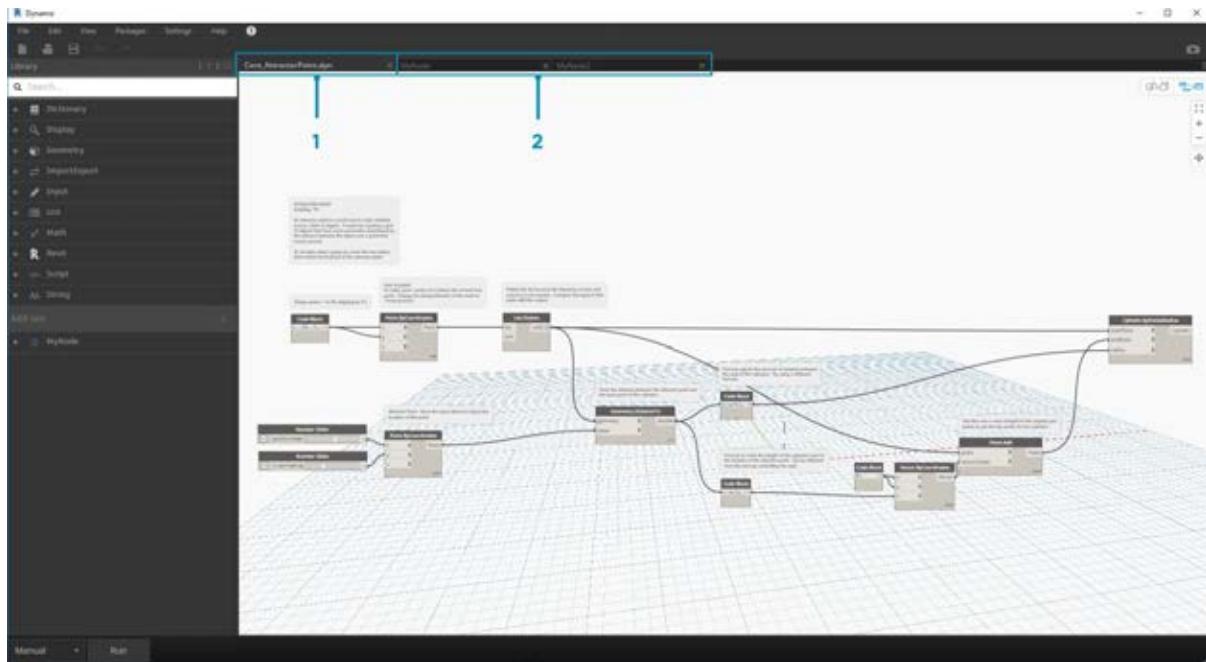
注: ノードとジオメトリは順序にしたがって描画されるため、複数のオブジェクトが互いに折り重なるようにレンダリングされる場合があります。そのため、複数のノードをまとめて追加すると、複数のオブジェクトがワークスペース内の同じ位置にレンダリングされ、見にくくなることがあります。



1. タブ
2. ズーム/画面移動ボタン
3. プレビュー モード
4. ワークスペースをダブルクリック

## タブ

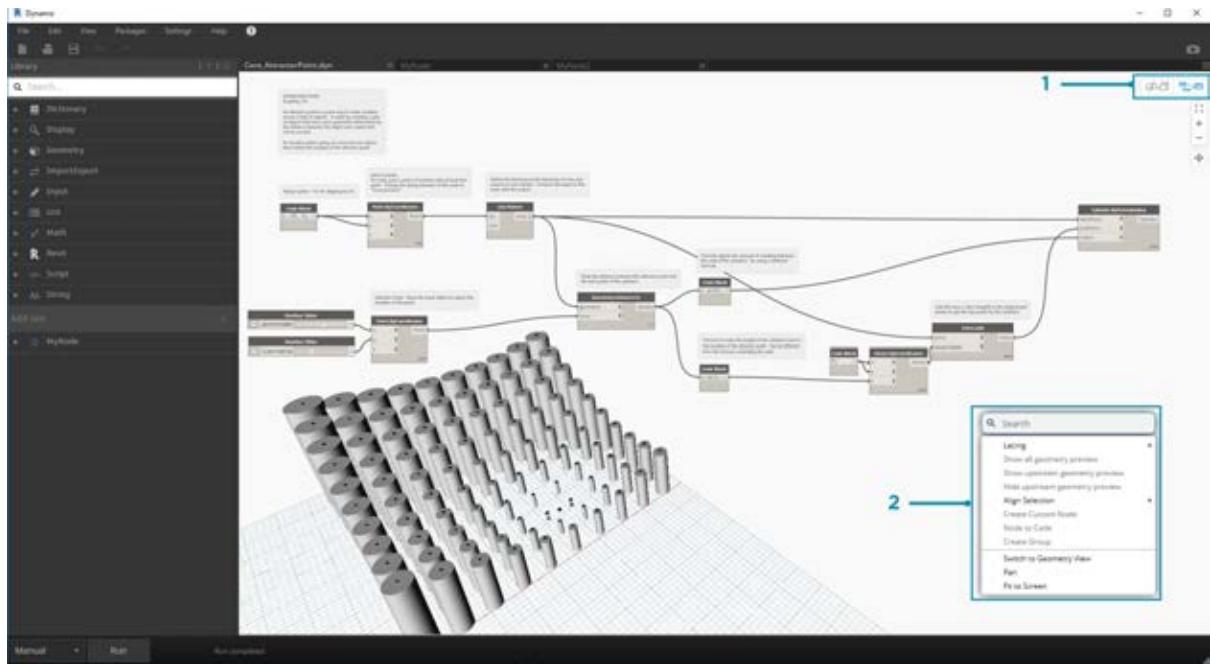
ワークスペースのアクティブなタブ上では、プログラムのナビゲートや編集を行うことができます。新しいファイルを開くと、既定で新しいホームワークスペースが表示されます。または、複数のノードを選択し、右クリックで[選択から新規ノードを作成]オプションを選択して、新しいカスタムノードワークスペースを開くこともできます。これについては、後で詳しく説明します([ファイル]メニューから開くこともできます)。



注: 同時に開くことができるホーム ワークスペースは 1 つだけです。ただし、カスタム ノード ワークスペースについては、タブを追加していくことで複数のワークスペースを開くことができます。

### グラフと 3D プレビュー ナビゲーション

Dynamo では、グラフとそのグラフの 3D 出力結果の両方が、ワークスペース内にレンダリングされます(ただし、3D の出力結果がレンダリングされるのは、ジオメトリを作成する場合にだけです)。既定では、グラフがアクティブなプレビューとして設定されています。そのため、ナビゲーション ボタンまたはマウスの中央ホイールを使用して、グラフ上で画面移動やズームを行うことになります。アクティブなビューを切り替えるには、次の 3 つの方法があります。

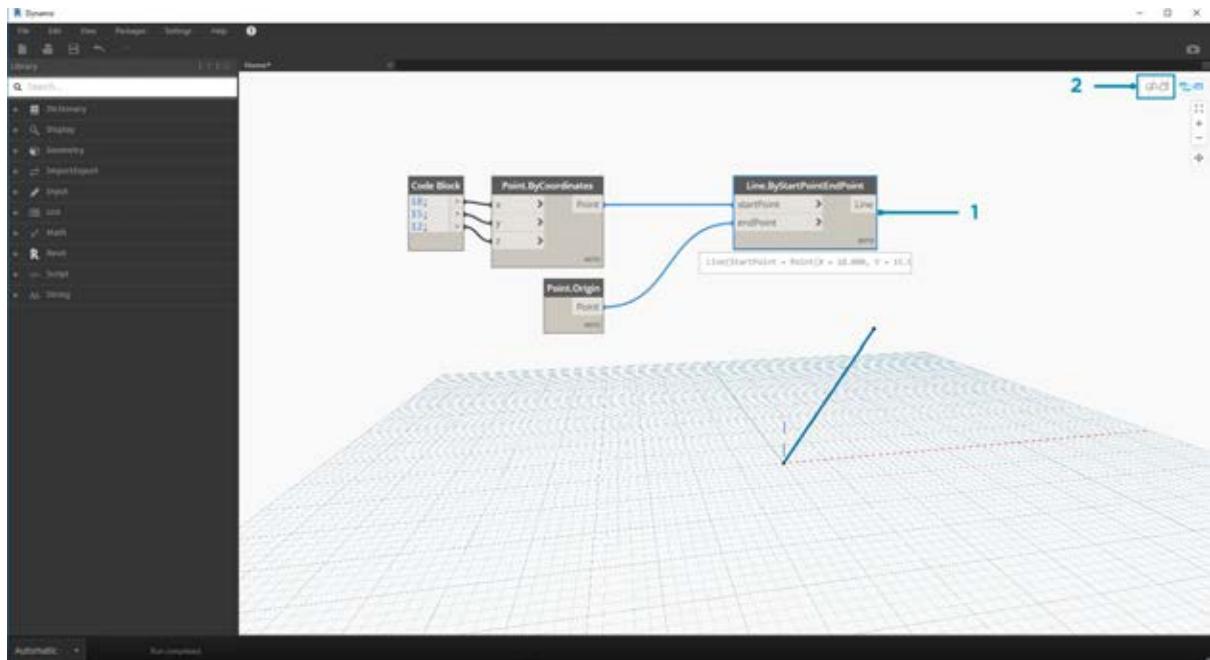


1. ワークスペース内のプレビューアイコンを使用する。
2. ワークスペース内で右クリックして、[ジオメトリビューに切り替える]または[ノードビューに切り替える]を選択する。
3. キーボードショートカット([Ctrl] + [B])を使用する。

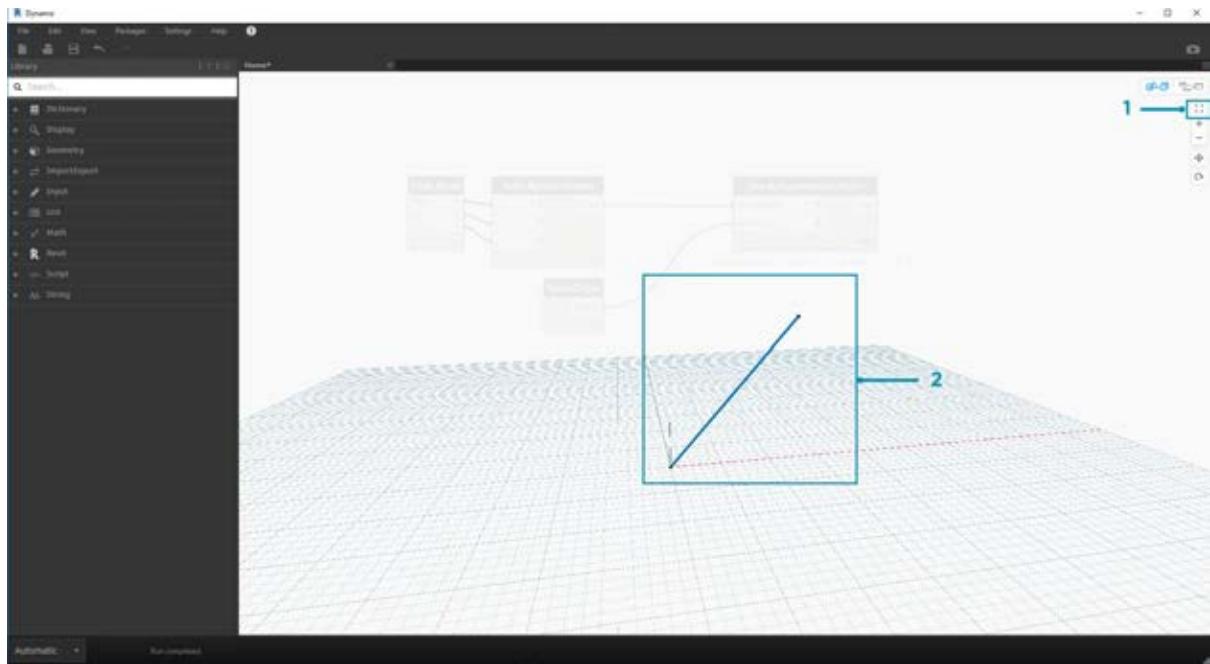
3D プレビュー ナビゲーション モードを使用すると、[スタートアップ ガイド](#)で紹介したように、点の直接操作を行うことができます。

#### ズームして中心位置を変更する

3D プレビュー ナビゲーション モードでは、モデルの画面移動、ズーム、回転を簡単に行うことができます。ただし、ジオメトリノード上で作成されたオブジェクトのズームを行うには、特定のノードを 1 つだけ選択して[全体表示]アイコンを使用します。



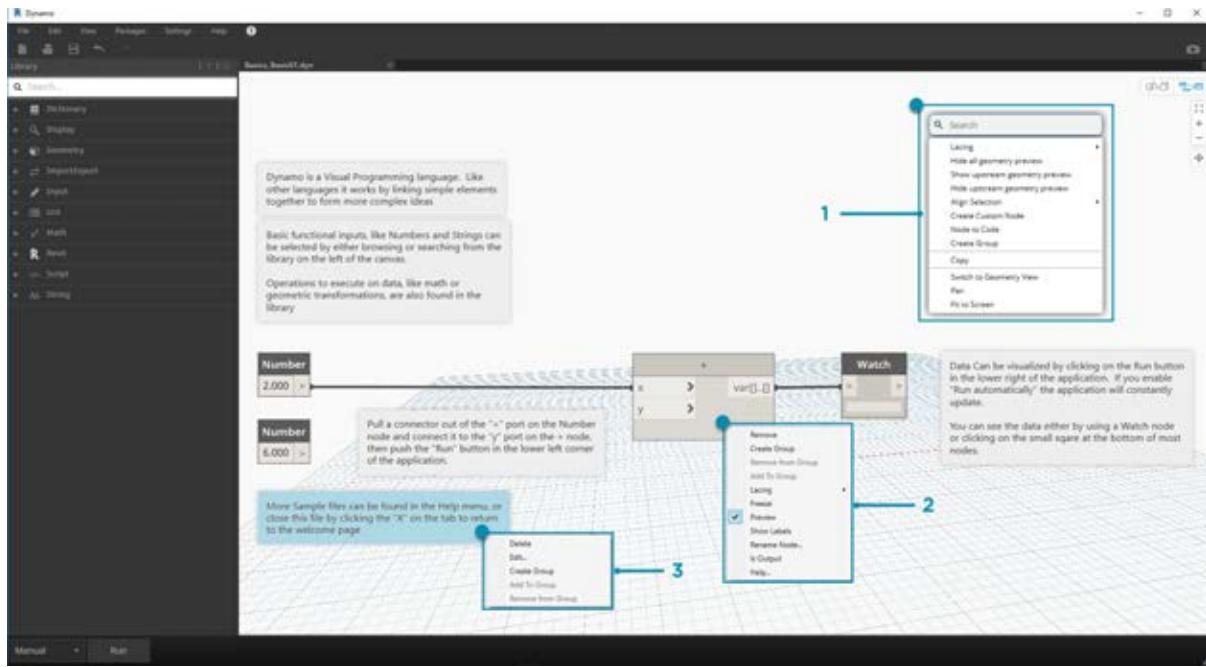
1. ビューの中心となるジオメトリに対応するノードを選択します。
2. 3D プレビュー ナビゲーションに切り替えます。



1. 画面右上の[全体表示]アイコンをクリックします。
2. 選択したジオメトリがビューの中心位置になります。

## マウス操作の基本

プレビューモードがアクティブになっているかどうかによって、マウスボタンの動作が異なります。通常、マウスの左クリックで要素の選択や入力値の指定を行い、マウスの右クリックでオプションを表示します。マウスの中央ホイールをクリックすると、ワークスペース内をナビゲートすることができます。マウスの右ボタンをクリックすると、クリックした場所に応じて異なるオプションが表示されます。



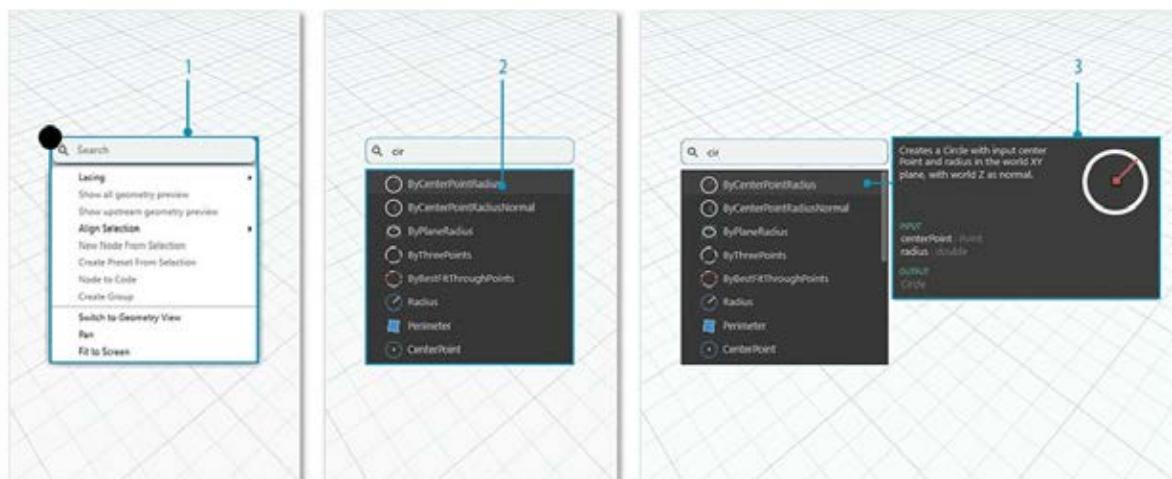
1. ワークスペースを右クリックします。
2. ノードを右クリックします。
3. 注釈を右クリックします。

次の表に、プレビュー時のマウス操作を示します。

マウスの操作	グラフ プレビュー	3D プレビュー
左クリック	選択	(動作なし)
右クリック	コンテキストメニュー	ズーム オプション
ホイール クリック	画面移動	画面移動
ホイール スクロール	ズームイン/ズームアウト	ズームイン/ズームアウト
ダブルクリック	コード ブロックを作成	(動作なし)

## キャンバス内検索

キャンバス内検索機能を使用すると、作業中のグラフ位置から離れることなくノードの説明やツールチップにアクセスできるため、Dynamo ワークフローをすばやく進めることができます。キャンバスのどの位置で作業していても、右クリックするだけで便利なライブラリ検索機能を使用することができます。



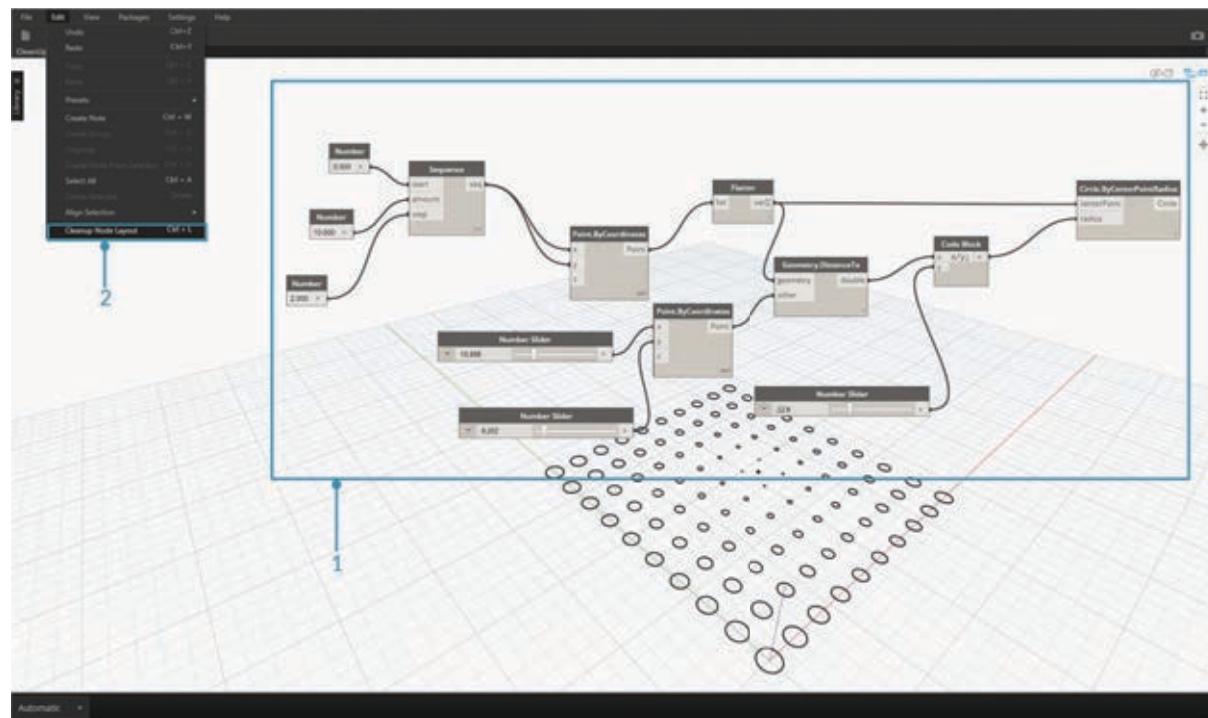
1. 検索機能を使用するには、キャンバス上の任意の場所を右クリックします。検索バーに何も入力されていない場合は、ドロップダウンメニューがプレビュー表示されます。

2. 検索バーに入力された文字列に応じて、その入力内容に最も関係がある検索結果がドロップダウンメニューに表示されます。この検索結果は、検索バーに文字を入力するたびに更新されます。
3. マウス カーソルを検索結果の上に置くと、対応する説明とツールチップが表示されます。

## ノードのレイアウトをクリーンアップ

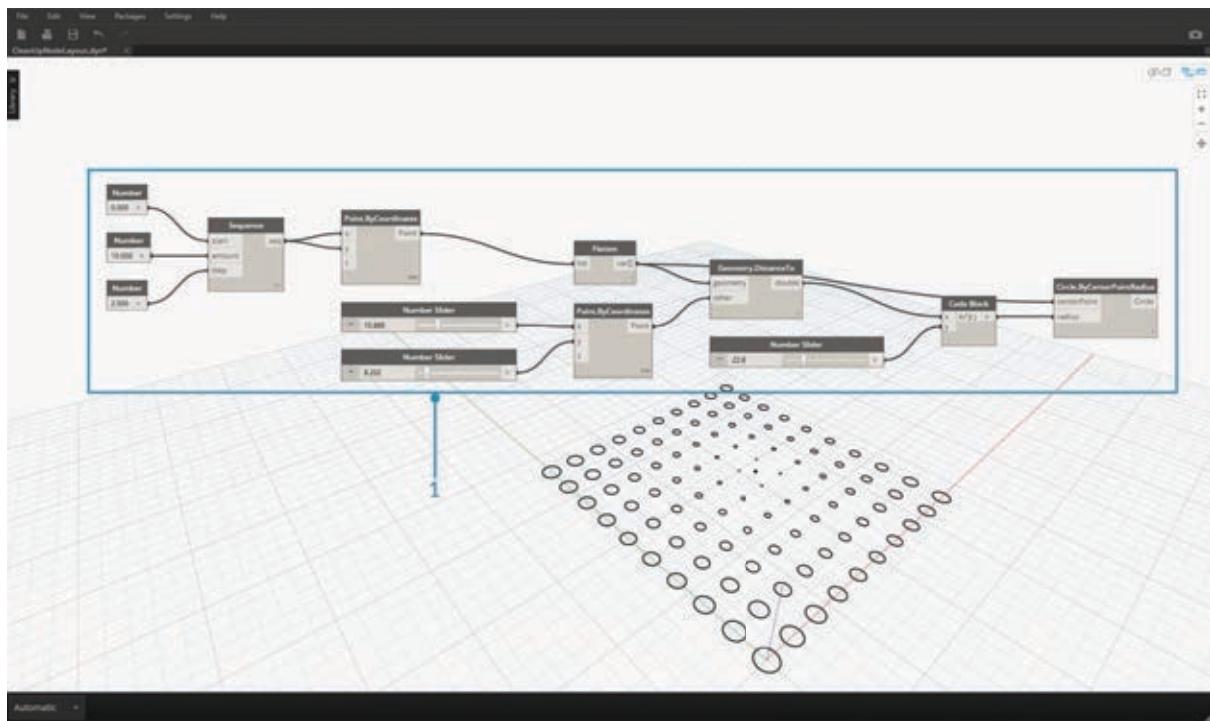
開発が進んでファイル構成が複雑になると、Dynamo のキャンバスを整理することが重要になります。小数のノードを選択して整理する場合は、[選択を位置合わせ]ツールを使用しますが、ファイル全体をクリーンアップする場合は、[ノードのレイアウトをクリーンアップ]ツールを使用すると便利です。

クリーンアップ前のノード



1. 自動的に整理するノードを選択します。ファイル内のすべてのノードをクリーンアップする場合は、すべてのノードを未選択の状態にしておきます。
2. [編集]タブで[ノードのレイアウトをクリーンアップ]を選択します。

クリーンアップ後のノード



1. ノードの再配置と位置合わせが自動的に実行されます。ノードのズレや重なりが解消され、隣り合うノードに合わせて位置が調整されます。

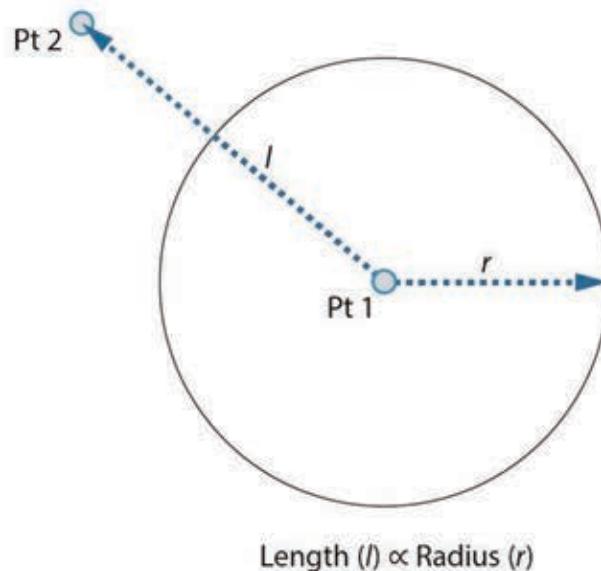
# 作業を開始する

## 作業を開始する

ここまででは、インターフェースのレイアウトとワークスペースのナビゲートについて見てきました。ここからは、Dynamo でグラフを作成するための一般的なワークフローについて見ていきましょう。最初に、動的にサイズを変更できる円を作成してから、さまざまな半径を持つ円の配列を作成します。

### 目的と関係を定義する

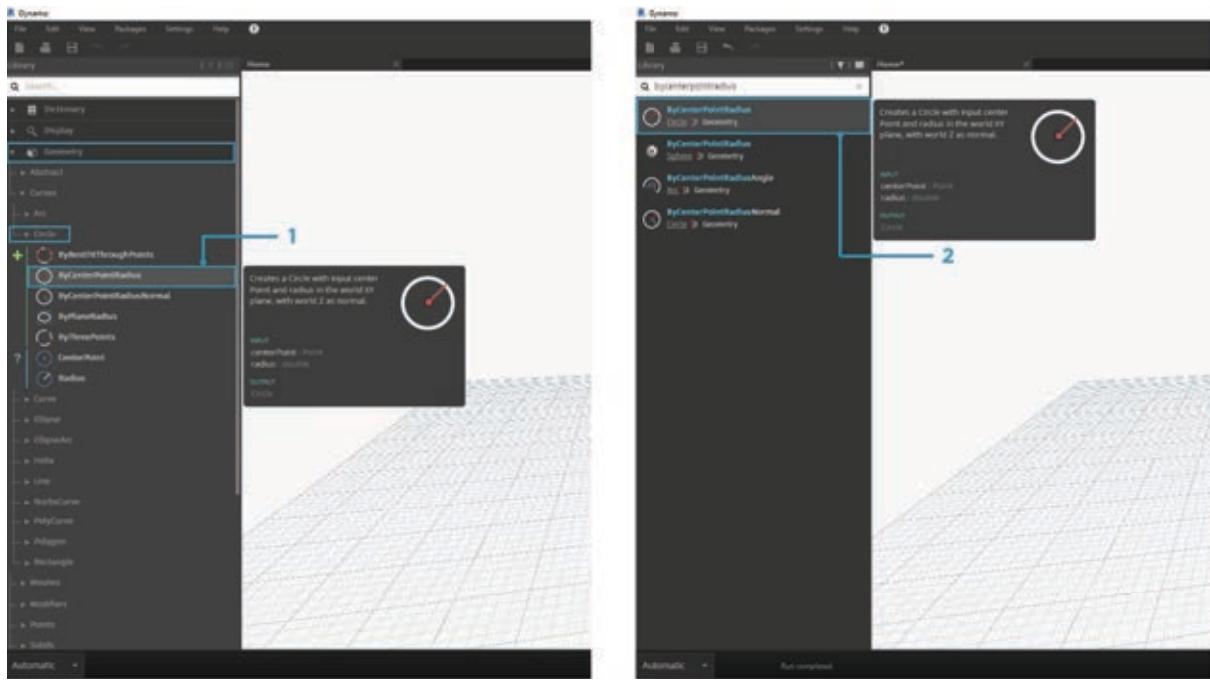
オブジェクトを Dynamo ワークスペースに追加する前に、作業の目的と、どのような関係が重要であるかを理解しておく必要があります。2 つのノードを接続すると、それらのノード間に明示的なリンクが作成されます。データのフローは後で変更できますが、ノードを一度接続すると、そのノード間の関係が確定します。この演習では円を作成し、近接する点までの距離により、円の半径の入力値を定義します。この場合、円を作成することが「目的」で、近接する点までの距離が「関係」になります。



距離ベースの関係を定義する点は、通常「アトラクタ」と呼ばれます。ここでは、アトラクタ点までの距離を使用して、円の大きさを指定します。

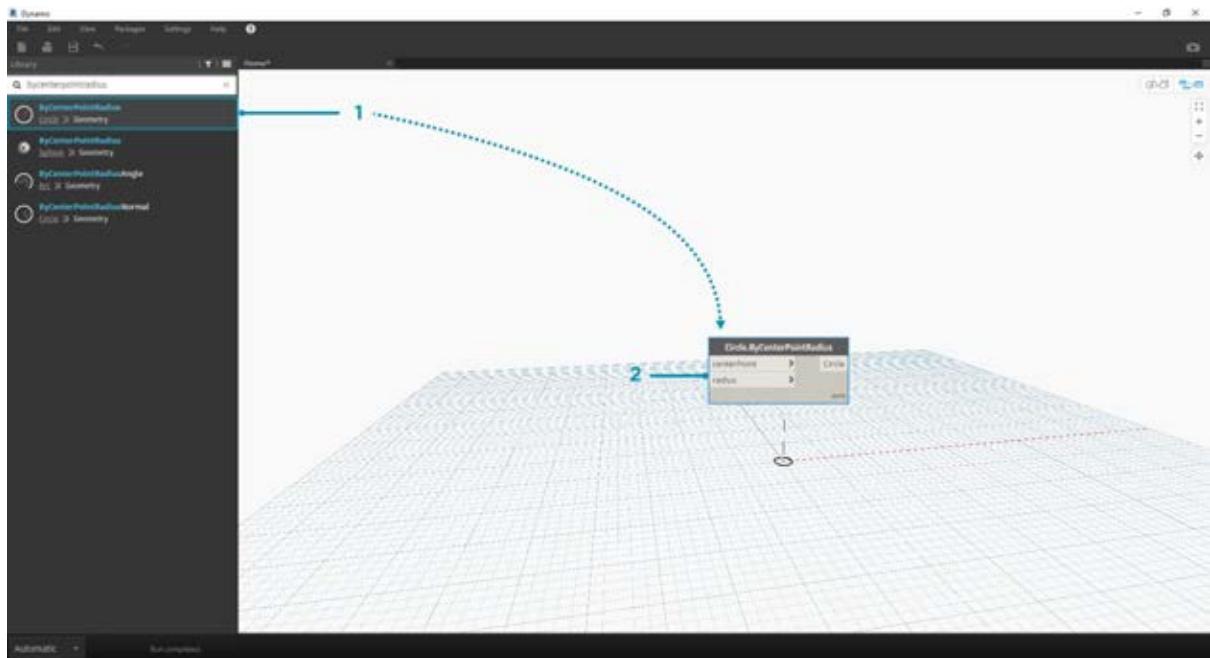
### ワークスペースにノードを追加する

目的と関係を確認したところで、グラフの作成を開始します。グラフを作成するには、Dynamo が実行するアクションの順番を表すノードが必要になります。ここでは円を作成することになるため、最初に、円を作成するノードを探しましょう。[検索] フィールドを使用するかライブラリを参照すると、複数の方法で円を作成できることがわかります。



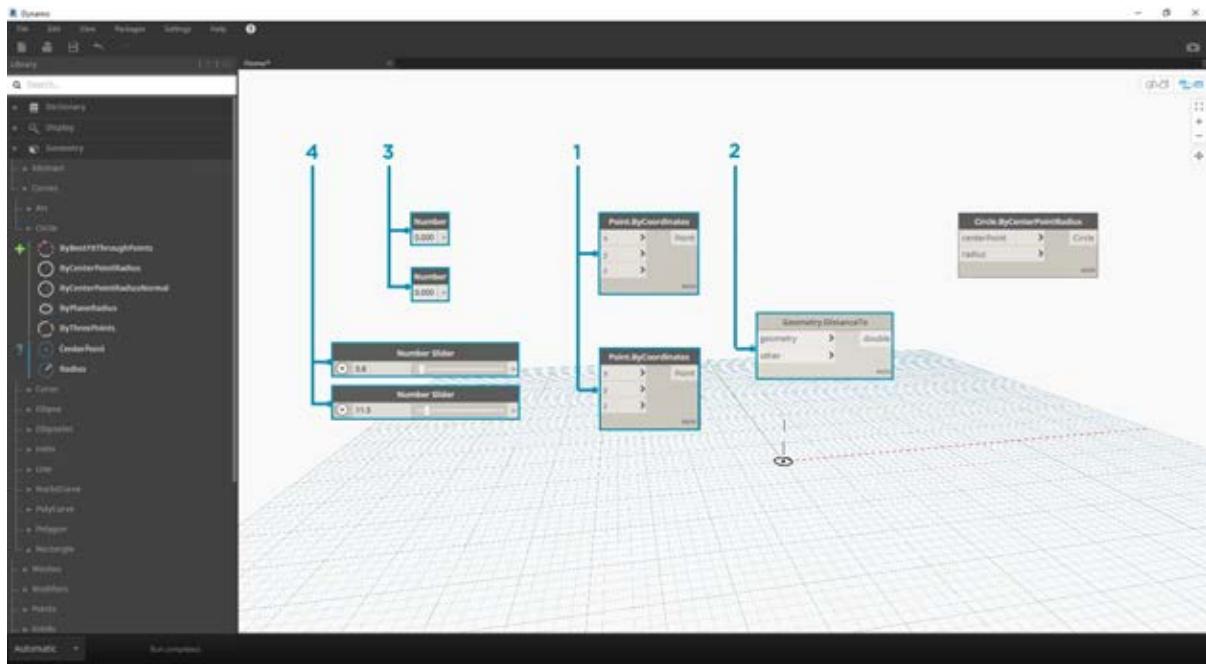
1. [Geometry] > [Curves] > [Circle] > [Circle.ByPointRadius]に移動します。
2. [検索]フィールドで「ByCenterPointRadius...」と入力します。

ライブラリで Circle.ByPointRadius ノードをクリックして、このノードをワークスペースの中心に追加します。



1. ライブラリ内の Circle.ByPointandRadius ノード
2. ライブラリで上記のノードをクリックして、このノードをワークスペースに追加します。

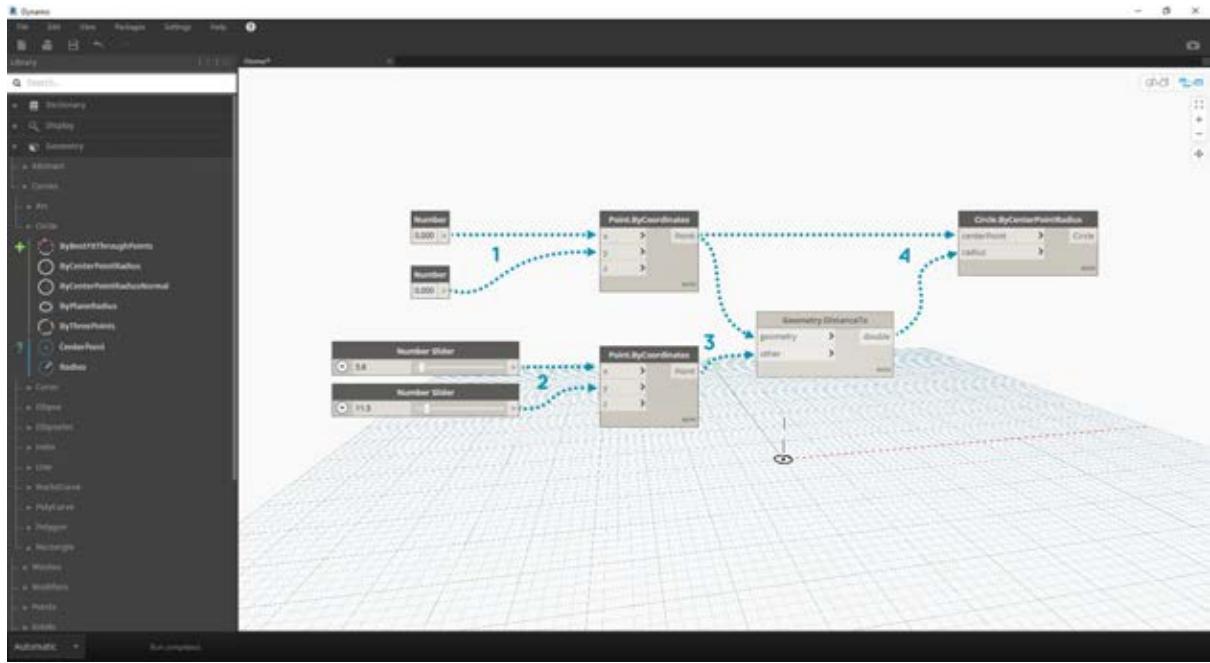
さらに、Point.ByCoordinates ノード、Number Input ノード、Number Slider ノードも必要になります。



1. [Geometry] > [Points] > [Point] > [Point.ByCoordinates]を選択します。
2. [Geometry] > [Geometry] > [DistanceTo]を選択します。
3. [Input] > [Basic] > [Number]を選択します。
4. [Input] > [Basic] > [Number Slider]を選択します。

#### ノードをワイヤで接続する

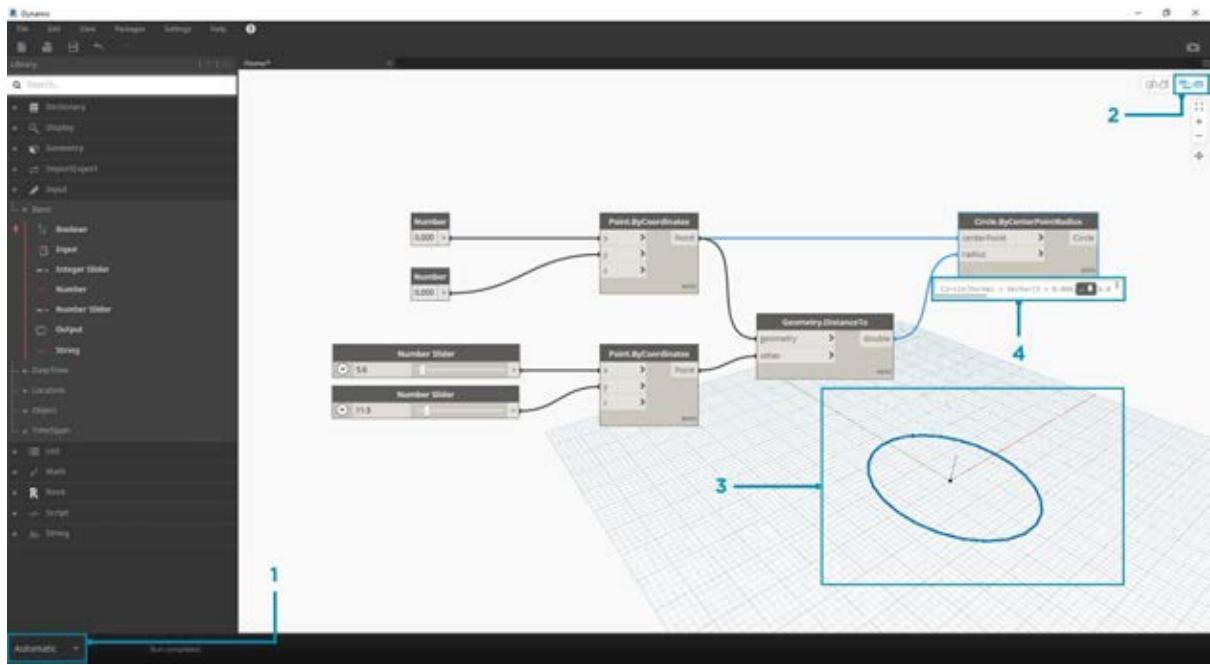
ノードを追加したら、そのノードのポートをワイヤで接続する必要があります。この接続により、データのフローが定義されます。



1. Number ノードを Point.ByCoordinates ノードに接続します。
2. Number Sliders ノードを Point.ByCoordinates ノードに接続します。
3. Point.ByCoordinates (2)ノードを DistanceTo ノードに接続します。
4. Point.ByCoordinates ノードと DistanceTo ノードを Circle.ByCenterPointRadius ノードに接続します。

### プログラムを実行する

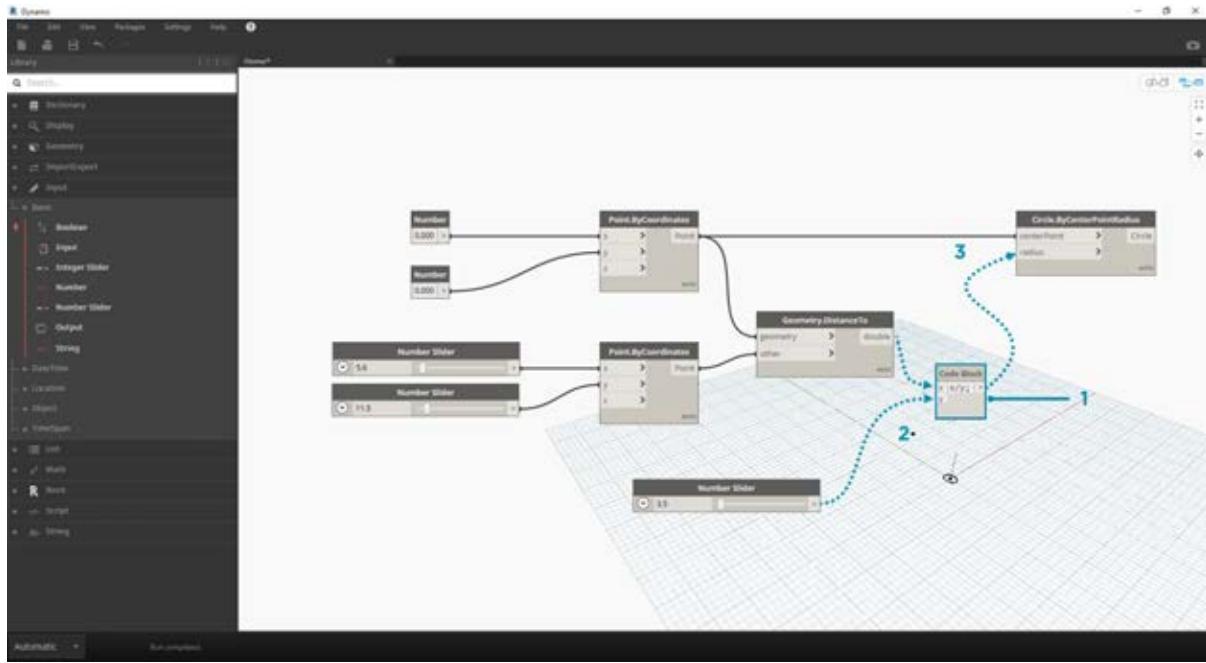
プログラム フローの定義が完了したら、後はプログラム フローの実行を Dynamo に対して指示するだけです。プログラムを実行すると、自動的に実行したか手動モードで[実行]をクリックして実行したかに関係なく、データがワイヤ経由で送信され、結果が 3D プレビューに表示されます。



1. [実行]をクリック - 実行バーが手動モードになっている場合は、[実行]をクリックしてグラフを実行する必要があります。
2. ノードのプレビュー - ノードの右下隅にあるボックス上にカーソルを合わせると、ポップアップに結果が表示されます。
3. 3D プレビュー - ノードを使用してジオメトリを作成した場合は、そのジオメトリが 3D プレビューに表示されます。
4. 作成ノードに関する出力ジオメトリが表示されます。

### 詳細を追加する

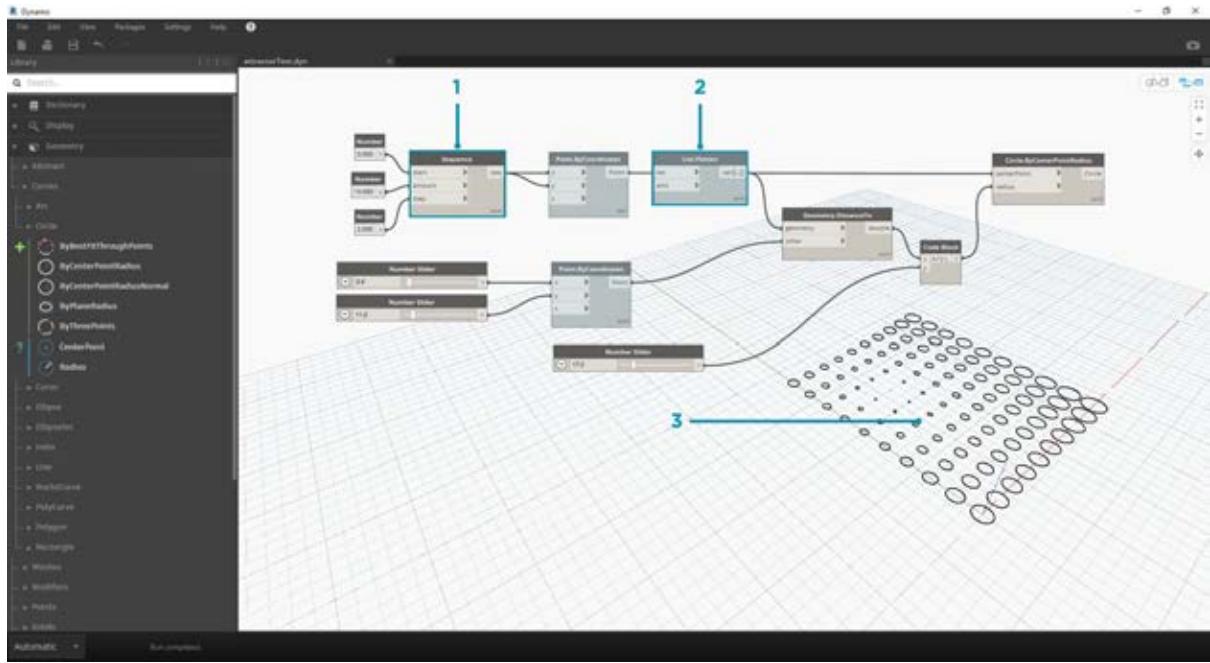
プログラムが稼働している場合、アトラクタ点を通過する円が 3D プレビューに表示されます。この状態で、さらに詳細やコントロールを追加することができます。ここでは、半径に対する影響を調整できるように、Circle ノードの入力を調整しましょう。別の Number Slider ノードをワークスペースに追加してから、ワークスペースの空白領域をダブルクリックして Code Block ノードを追加します。次に、Code Block ノードのフィールドで「X/Y」を指定します。



1. **Code Block** ノード
2. **DistanceTo** ノードと **Number Slider** ノードを **Code Block** ノードに接続します。
3. **Code Block** ノードを **Circle.ByCenterPointRadius** ノードに接続します。

## 複雑なプログラムにする

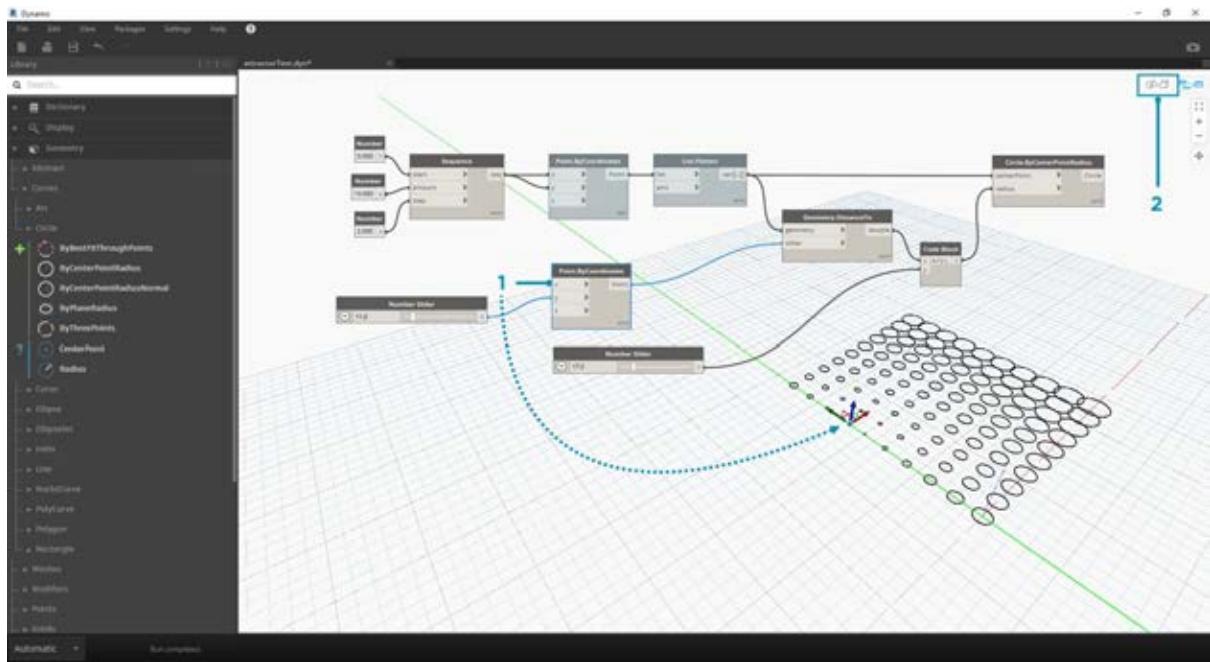
単純なプログラムを作成して次第に複雑なものにしていく方法は、プログラムを段階的に開発するための効果的な方法です。1つの円で機能するプログラムを作成したら、複数の円でも機能するようにプログラムの性能を拡張してみましょう。1つの中心点の代わりに点のグリッドを使用し、生成されるデータ構造の変更を適用すると、プログラムによって多くの円が作成されます。各円の半径の値は、アトラクタ点までの調整された距離によって定義されたユニークな値になります。



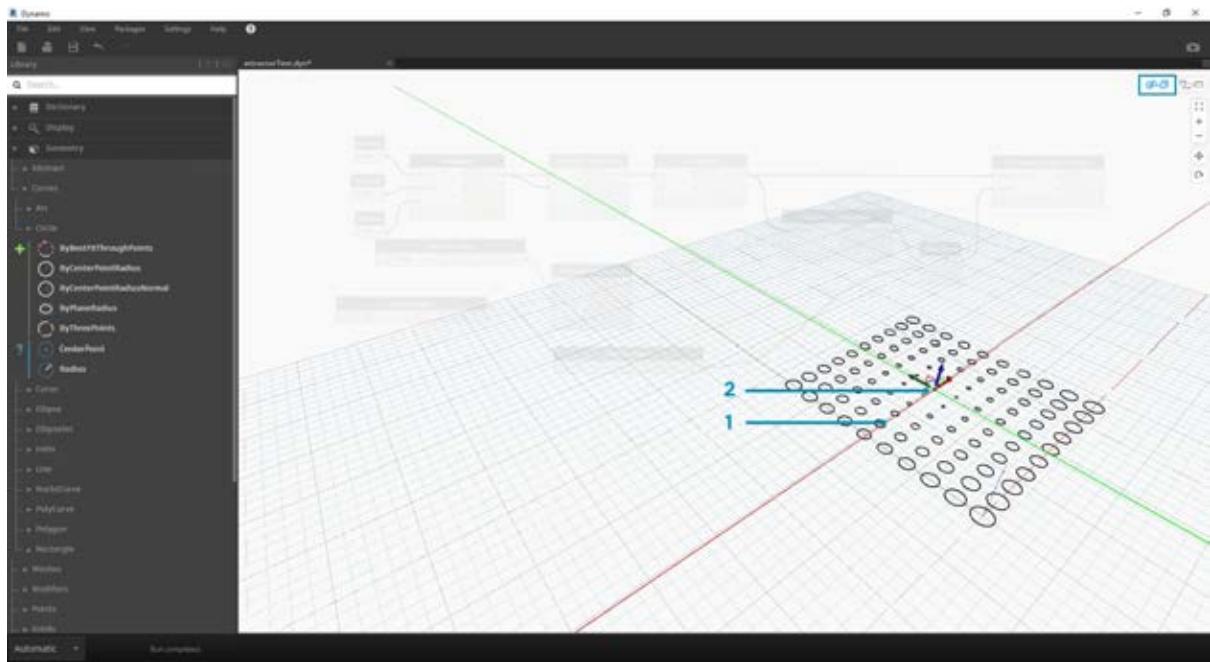
1. Number Sequence ノードを追加して Point.ByCoordinates ノードの入力を置き換え、Point.ByCoordinates ノードを右クリックして[レーシング] > [外積]を選択します。
2. Point.ByCoordinates ノードの後に Flatten ノードを追加します。リストを完全にフラット化するには、amt 入力を既定の -1 のままにします。
3. 3D プレビューが円のグリッドによって更新されます。

### 直接操作で値を調整する

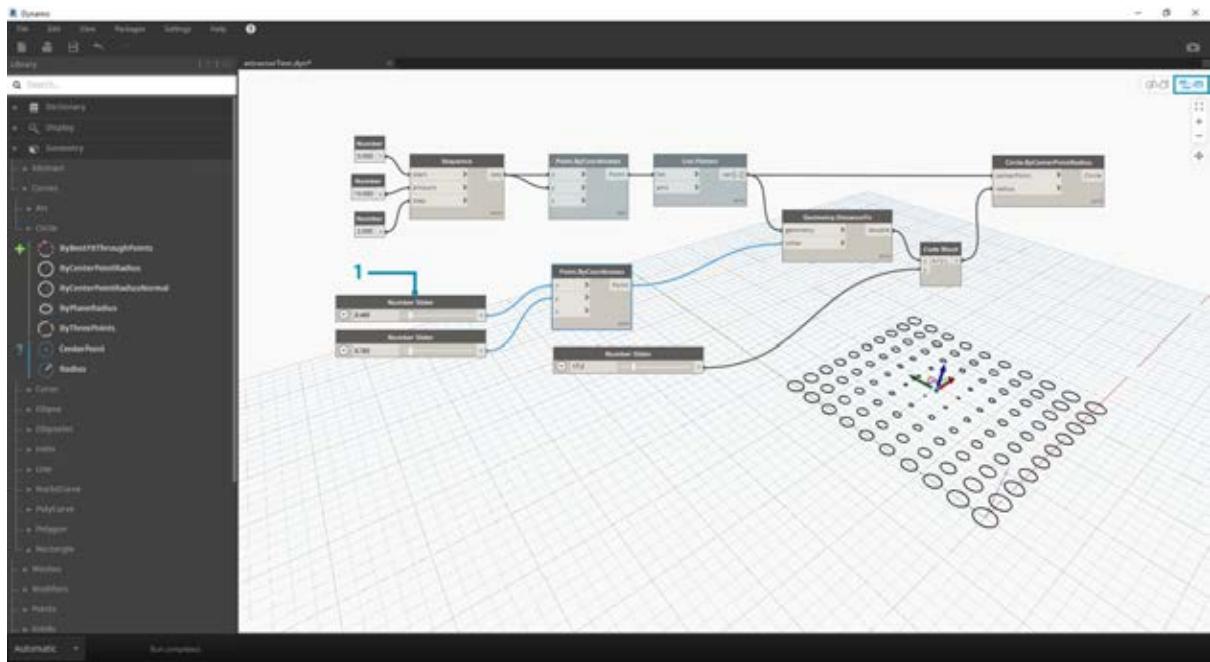
場合によっては、数値を操作しない方がいいことがあります。このような場合、背景の 3D プレビューをナビゲートする際に、点ジオメトリを手動で操作することができます。また、点によって作成された他のジオメトリをコントロールすることもできます。たとえば、Sphere.ByCenterPointRadius ノードの場合、直接操作を行うこともできます。点の位置は、Point.ByCoordinates ノードで X、Y、Z の値を使用してコントロールすることができます。ただし、直接操作の場合、3D プレビュー ナビゲーション モードで点を手動で移動することにより、スライダの値を更新することができます。この方法により、点の場所を識別する個別の値セットを直感的にコントロールすることができます。



1. 直接操作を使用するには、移動する点のパネルを選択します。選択した点の上に矢印が表示されます。
2. 3D プレビュー ナビゲーション モードに切り替えます。



1. 点の上にカーソルを合わせると、X 軸、Y 軸、Z 軸が表示されます。
  2. 色付きの矢印をクリックして対応する軸にドラッグすると、手動で移動した点に合わせて **Number Slider** ノードの値が更新されます。

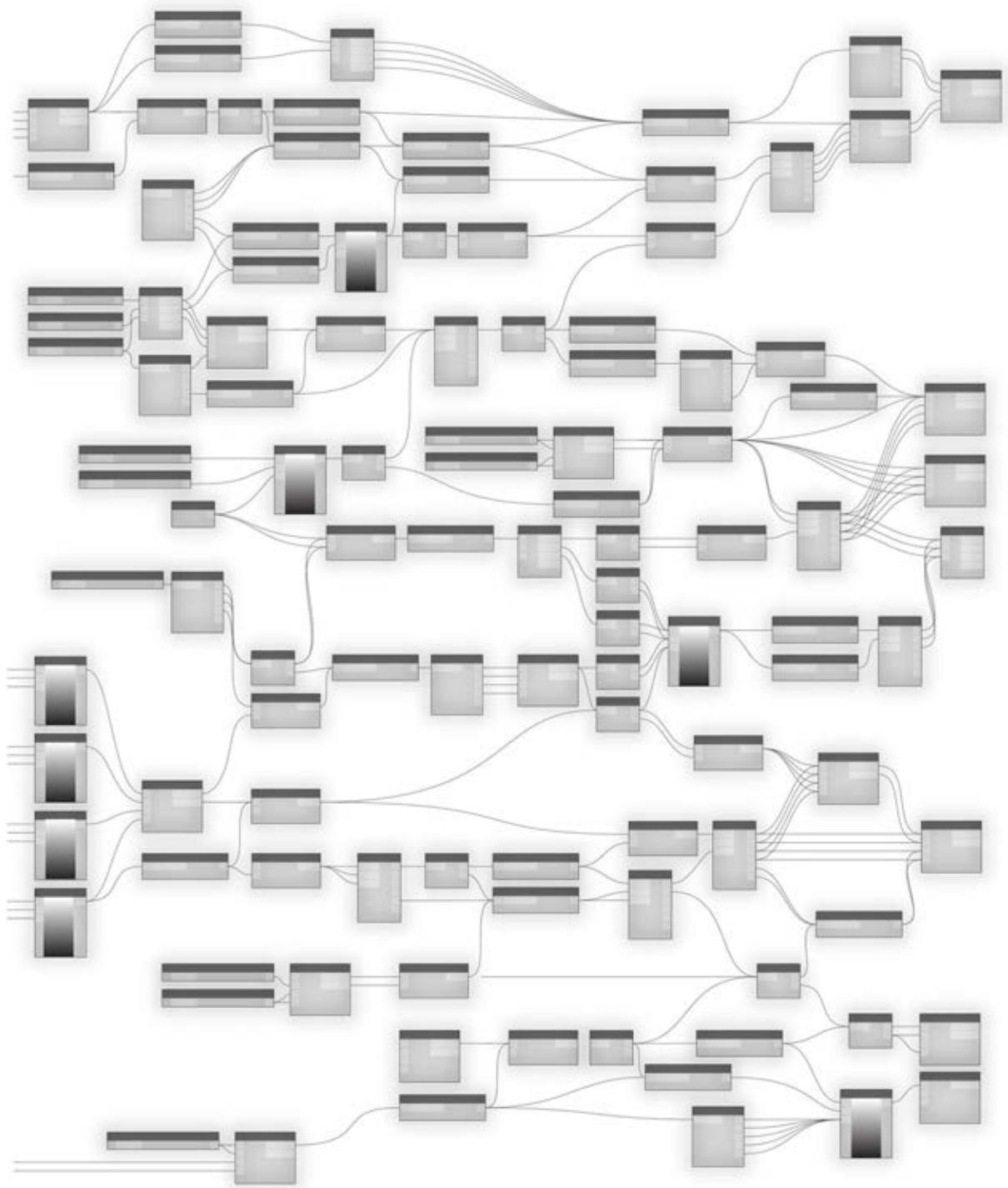


1. 直接操作を実行する前は、1 つのスライダだけが **Point.ByCoordinates** コンポーネントに接続されていたことに注意してください。点を X 方向に手動で移動すると、X 入力用として新しい **Number Slider** ノードが自動的に生成されます。

## ビジュアル プログラムの構造

### ビジュアル プログラムの構造

Dynamo を使用すると、ノードをワイヤで接続してビジュアル プログラムをワークスペース内に作成し、そのビジュアル プログラムの論理フローを指定できます。この章では、ビジュアル プログラムの要素、Dynamo のライブラリで使用可能なノードの編成、ノードのパラメータと状態、ワークスペースのベスト プラクティスについて紹介します。



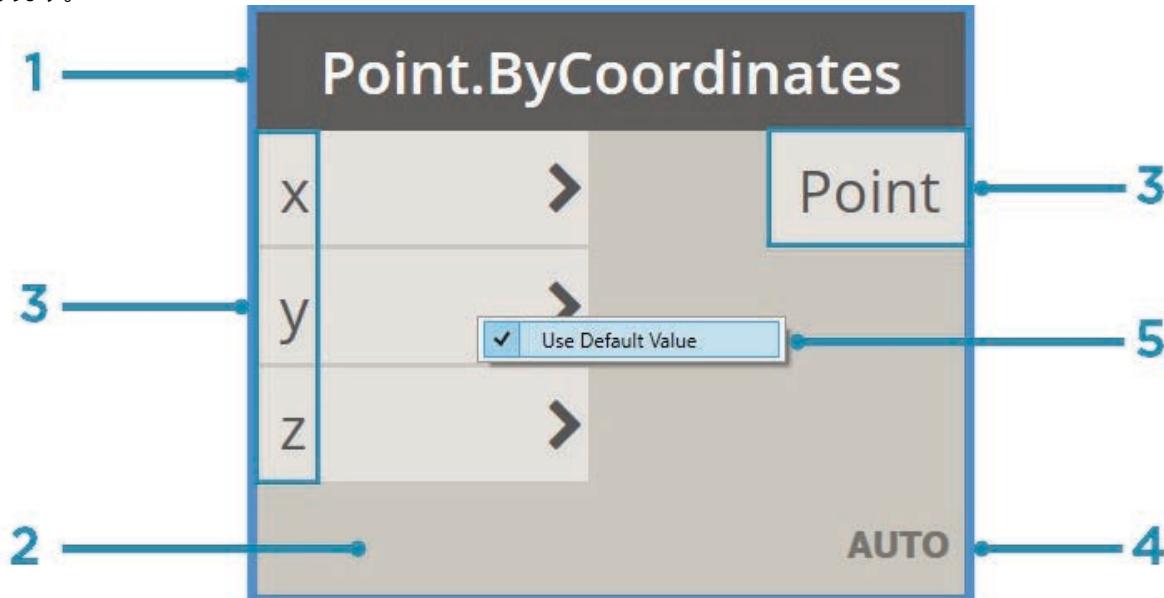
# ノード

## ノード

Dynamo のノードは、ビジュアル プログラムを形成するために接続するオブジェクトです。各ノードは、特定の操作を実行します。操作には、数値の保存といった単純なものもあれば、ジオメトリの作成やクエリーの実行といった複雑なものもあります。

### ノードの構造

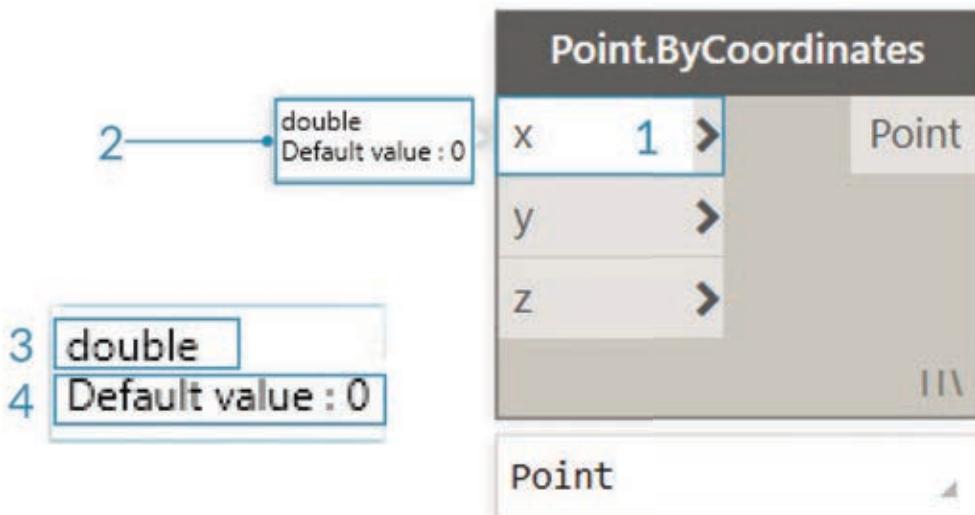
Dynamo のほとんどのノードは、5 つのパートで構成されています。入力ノードなどの例外はありますが、各ノードの構造は次のようになります。



### ポート

ノードの入力と出力はポートと呼ばれ、ワイヤの出入り口として機能します。データは左側のポートを経由して入力され、操作が実行されるとノードの右側から出力されます。ポートは、特定のタイプのデータを受信するように設定されています。たとえば、2.75 という数値を Point.ByCoordinates ノードのポートに接続すると、点が正常に作成されます。ただし、同じポートに対して「Red」を指定すると、エラーが発生します。

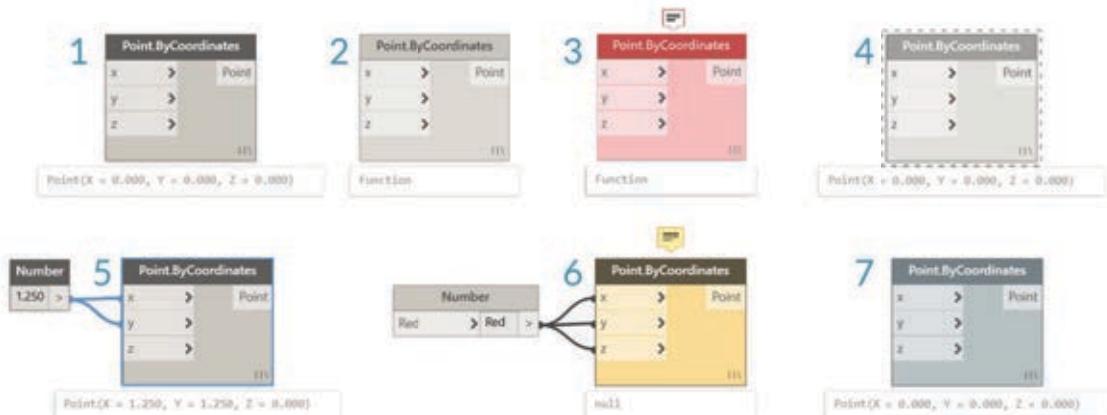
ヒント: ポートにカーソルを合わせると、そのポートの正しいデータ タイプを示すツールチップが表示されます。



1. ポートのラベル
2. タールチップ
3. データタイプ
4. 既定値

## 状態

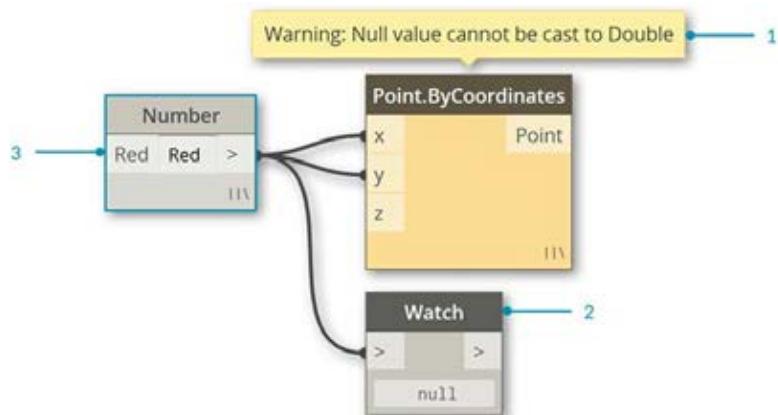
Dynamo は、各ノードのステータスに基づいて異なるカラー スキームでノードをレンダリングすることにより、ビジュアル プログラムの実行状態を示します。また、名前やポートにカーソルを合わせるか右クリックすると、追加の情報やオプションが表示されます。



1. アクティブ - ノード名の背景がダークグレーになっているノードは正常に接続されており、すべての入力が正常に接続されています。
2. 非アクティブ - グレーのノードは非アクティブになっているため、ワイヤで接続してアクティブなワークスペースのプログラムフローに追加する必要があります。
3. エラー状態 - エラー状態のノードは赤色で表示されます。
4. フリーズ - フリーズ状態のノードは半透明で表示されます。こうしたノードの実行は中止されています。
5. 選択済み - 現在選択されているノードは、境界が水色でハイライト表示されます。
6. 警告 - 警告状態のノードは黄色で表示されます。これは正しくないデータタイプが入力されていることを表します。
7. 背景プレビュー - ダークグレーは、ジオメトリのプレビューがオフになっていることを表します。

ビジュアルプログラムに警告やエラーが含まれている場合、その問題に関する追加情報が表示されます。また、黄色のノードでも、ノード名の上にツールチップが表示されます。このツールチップにカーソルを合わせると、ツールチップの内容が展開表示されます。

ヒント: このツールチップ情報を使用して上流のノードを検査することにより、必要なデータタイプまたはデータ構造にエラーがないかどうかを確認することができます。



1. 警告ツールチップ - 「Null」を指定した場合、またはデータを指定しなかった場合、倍精度浮動小数点数(数値)として認識されません。
2. Watchノードを使用して入力データを検査します。
3. 上流の Number ノードに数値ではなく「Red」が指定されています。

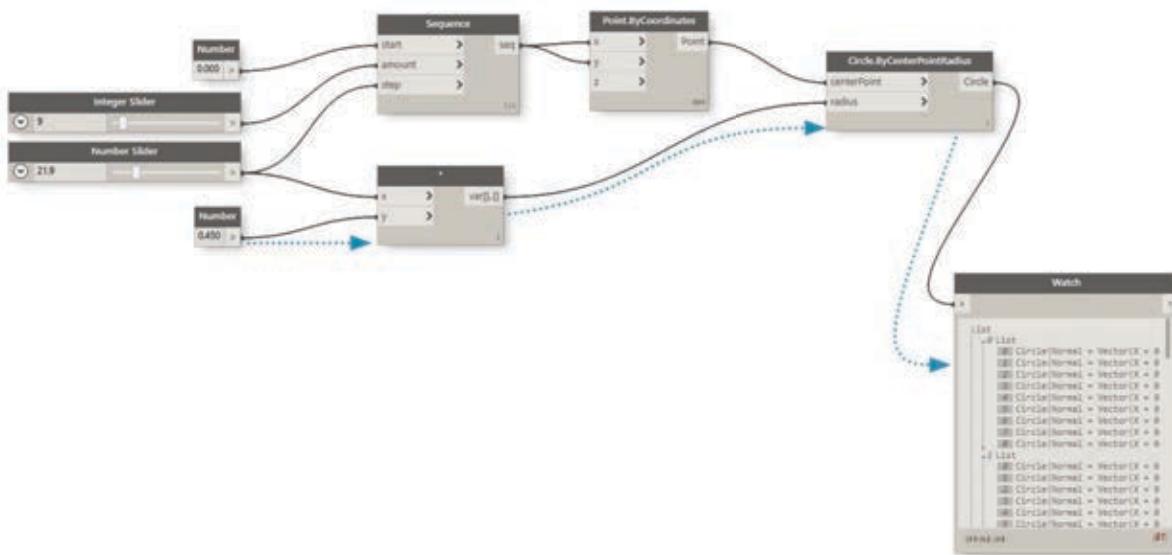
# ワイヤ

## ワイヤ

ワイヤは各ノードを接続してノード間の関係を作成し、ビジュアル プログラムのフローを確立します。ワイヤはその名前のとおり、特定のオブジェクトから次のオブジェクトにデータ パルスを送信するための電線と考えることができます。

### プログラム フロー

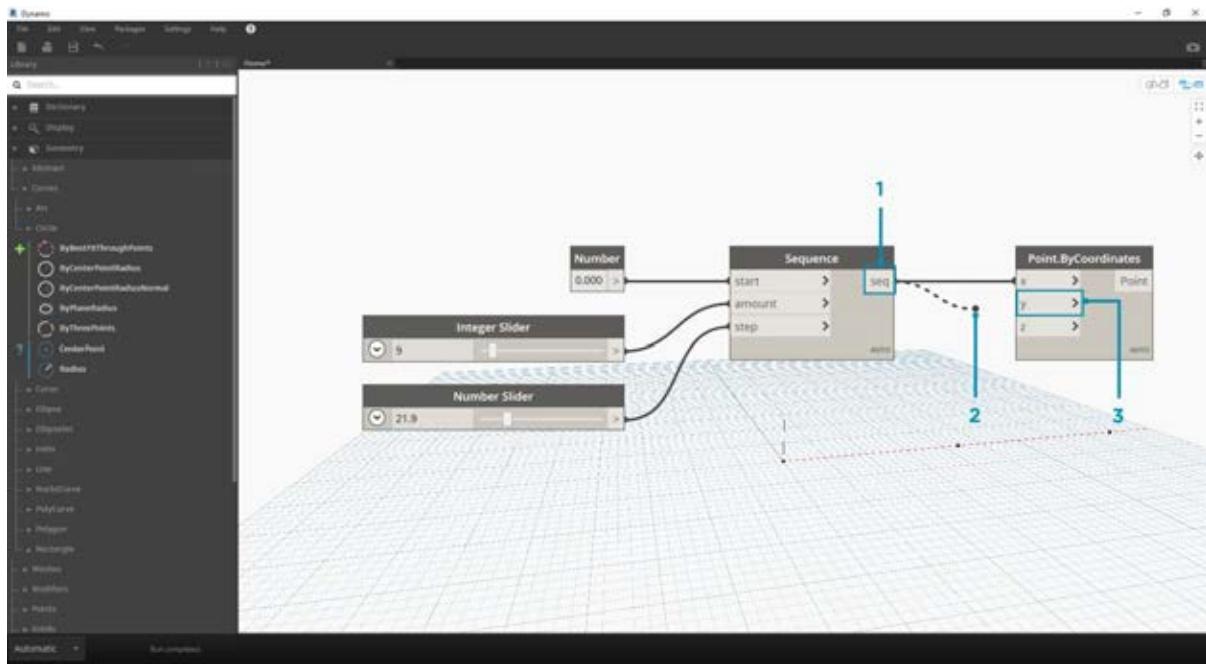
ワイヤは、特定のノードの出力ポートを別のノードの入力ポートに接続します。この接続により、ビジュアル プログラムのデータ フローが確立します。ワークスペースではノードを自由に配置できますが、出力ポートはノードの右側、入力ポートはノードの左側に配置されているため、プログラム フローの方向は、通常、左から右になります。



### ワイヤを作成する

ワイヤを作成するには、特定のポート上でマウスを左クリックし、次に別のノードのポート上で左クリックします。この操作により、これらのポート間の接続が作成されます。接続の作成中、ワイヤは破線で表示され、正常に接続されると実線に変わります。データは常にこのワイヤを経由して出力から入力へと移動します。ただし、接続されている一連のポートをクリックすると、どちらの方向にもワイヤを作成できます。

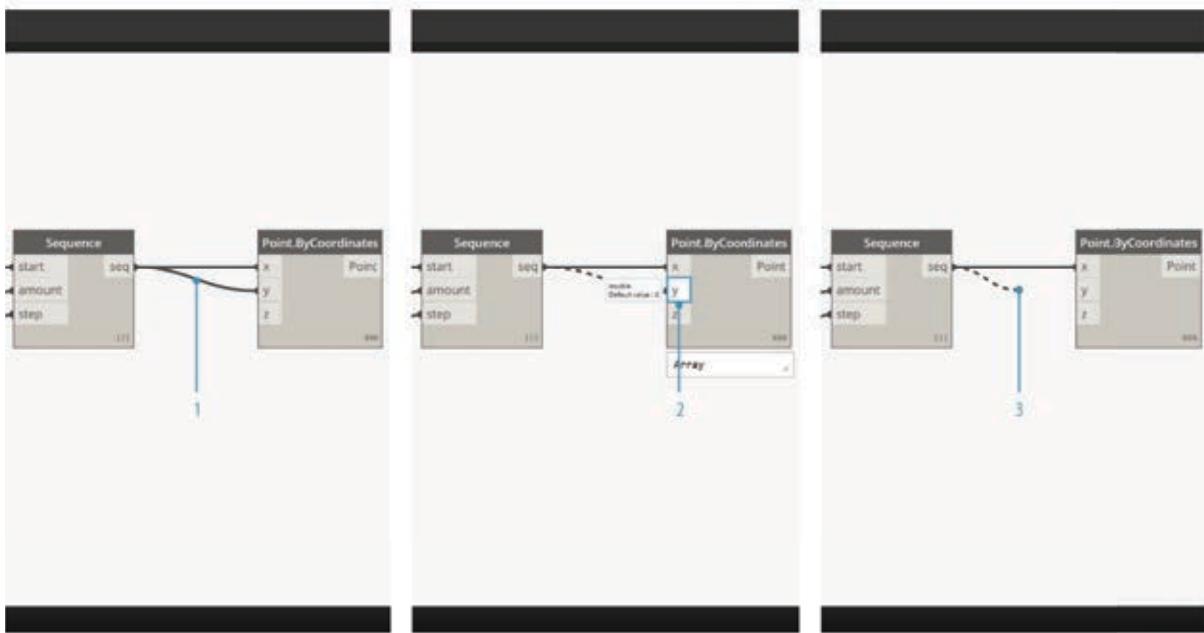
ヒント: 2 回目のクリックで接続を完了する前に、ワイヤをポートにスナップしてカーソルを合わせると、そのポートのツールチップが表示されます。



1. Sequence ノードの seq 出力ポートをクリックします。
2. マウスを別のポートに移動させる間、ワイヤは破線で表示されます。
3. Point.ByCoordinates ノードの y 入力ポートをクリックして接続を完了します。

### ワイヤを編集する

多くの場合、ワイヤで表示されている接続を編集して、ビジュアル プログラムのプログラム フローを調整する必要が生じます。ワイヤを編集するには、既に接続されているノードの入力ポートを左クリックします。ワイヤを編集する場合の方法は 2 つあります。

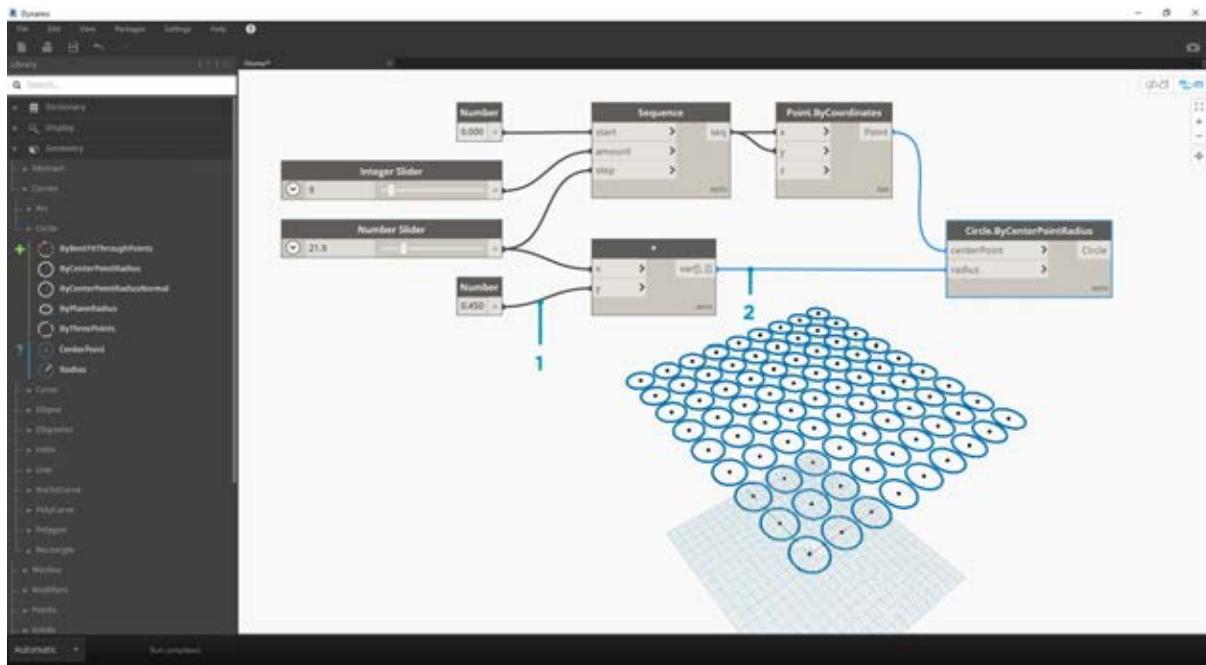


- 既存のワイヤを選択します。
- 入力ポートへの接続を変更するには、別の入力ポートを左クリックします。
- ワイヤを削除するには、ワイヤを入力ポートから離してワークスペースを左クリックします。

\*注: 複数のワイヤを一度に移動する追加機能も備わりました。これについては <http://dynamobim.org/dynamo-1-3-release/> で説明しています。

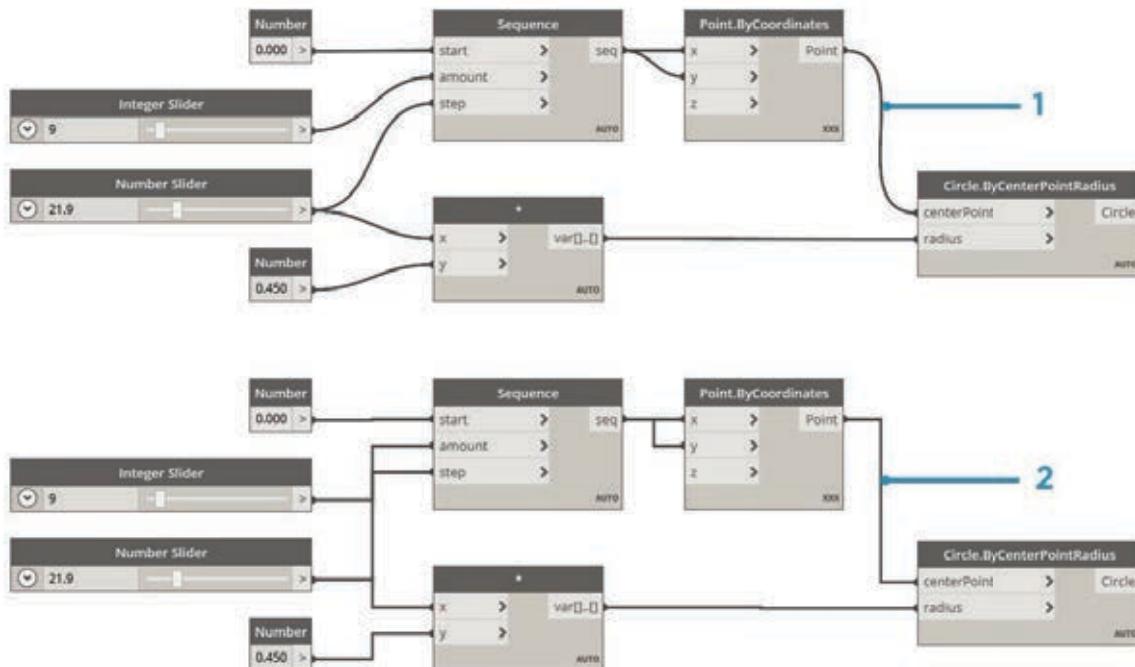
#### ワイヤのプレビュー

既定では、ワイヤはグレーのストロークでプレビュー表示されます。特定のノードを選択すると、接続されているワイヤがそのノードと同じ水色でハイライト表示されます。



1. 既定のワイヤ
2. ハイライト表示されているワイヤ

また、[ビュー] > [コネクタ]メニューを使用することにより、ワークスペース内でワイヤを表示する方法をカスタマイズできます。曲線やポリラインのワイヤを切り替えたり、すべてのワイヤをまとめてオフにすることができます。



1. コネクタ タイプ: 曲線
2. コネクタ タイプ: ポリライン

# Dynamo ライブラリ

## Dynamo ライブラリ

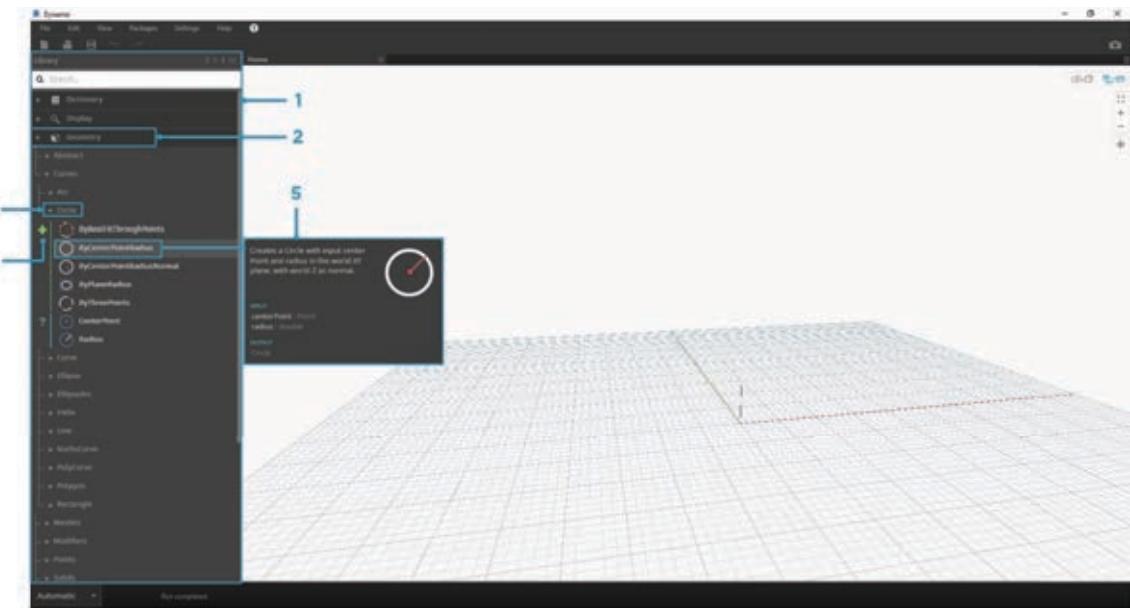
Dynamo ライブラリには、ワークスペースに追加するノードが含まれています。これらのノードを使用して、実行するビジュアル プログラムを定義します。ライブラリでは、ノードの検索と参照を行うことができます。ライブラリには、インストール済みの基本ノード、ユーザが定義したカスタム ノード、Dynamo に追加された Package Manager のノードが格納されます。これらのノードは、カテゴリによって階層的に編成されます。ここでは、この編成を確認して、頻繁に使用する主要なノードを見ていきます。

### ライブラリのライブラリ

アプリケーションで使用する Dynamo ライブラリは、実際には関数ライブラリの集合です。各ライブラリには、カテゴリ別にグループ化されたノードが含まれています。最初は、この仕組みを不便だと感じるかもしれません、Dynamo の既定のインストールに付属しているノードを整理するための柔軟な仕組みです。また、この基本機能をカスタム ノードや追加パッケージを使用して拡張すると、さらに便利になります。

### 編成スキーム

Dynamo UI の[ライブラリ]セクションは、階層構造のライブラリで構成されています。ライブラリの階層は、ライブラリ、ライブラリのカテゴリ、カテゴリのサブカテゴリ、ノードという順序で構成されています。

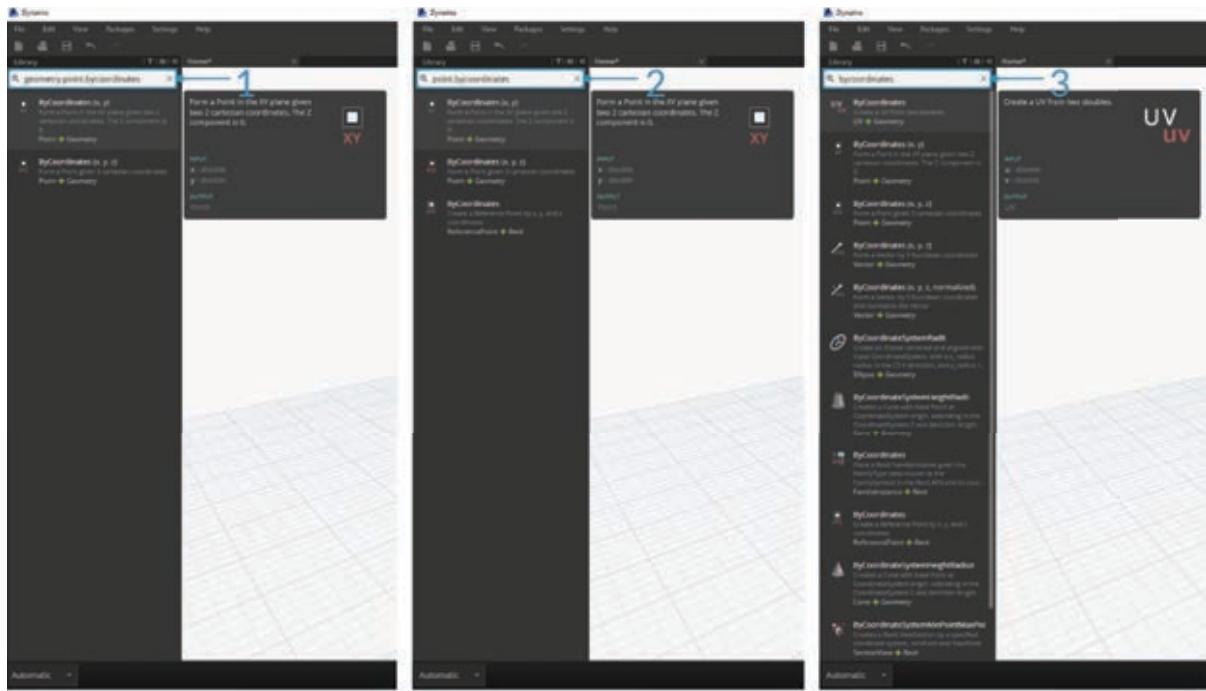


1. ライブラリ - Dynamo のインターフェース領域
2. ライブラリ - Geometry などの関連カテゴリの集合
3. カテゴリ - Circle に関するすべてのノードなど、関連ノードの集合
4. サブカテゴリ - カテゴリ内のノードの内容(通常、Create、Action、または Query)
5. ノード - アクションを実行するための、ワークスペースに追加されたオブジェクト

## 命名規則

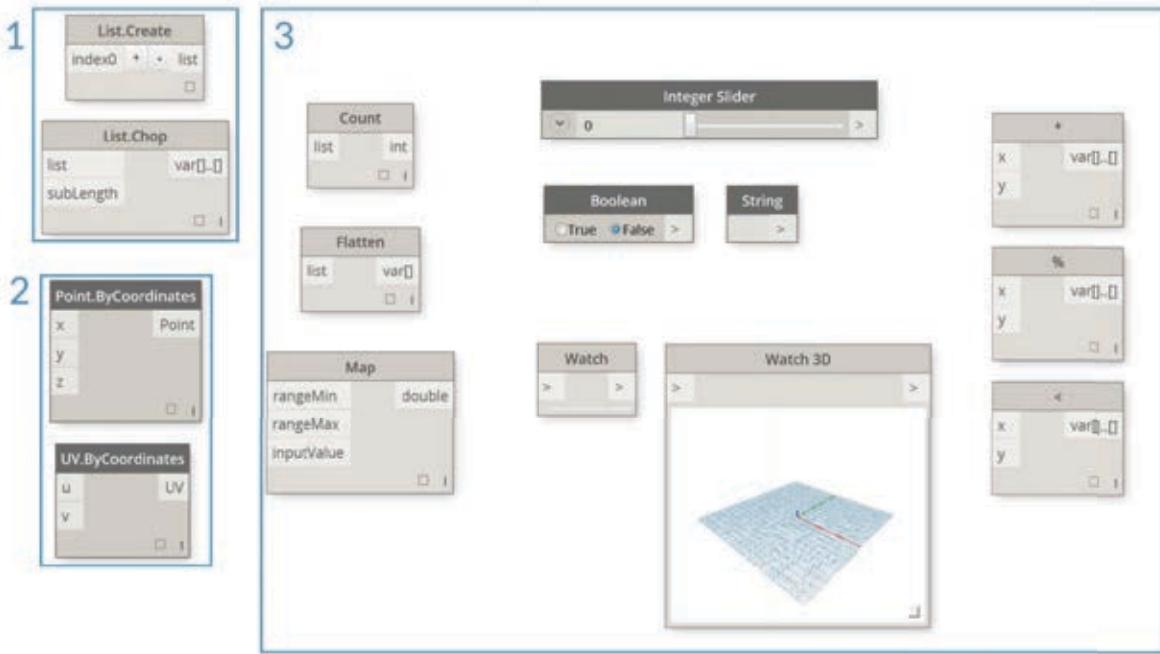
各ライブラリの階層は、ワークスペースに追加されたノードの名前に反映されます。この階層は、[検索]フィールドまたは(Dynamo テキスト言語を使用する)コードブロックで使用することもできます。ノードを検索する場合は、キーワードを使用するだけでなく、ピリオドで区切った階層名を入力して検索することもできます。

ライブラリ階層内のノードの場所の任意の部分を `library.category.nodeName` という形式で入力すると、次のように異なる結果が返されます。



1. `library.category.nodeName`
2. `category.nodeName`
3. `nodeName` または `keyword`

通常、ワークスペース内のノードの名前は `category.nodeName` という形式で表示されますが、いくつかの例外があります。特に注意が必要な例外は、Input カテゴリと View カテゴリです。ノードの名前は同じですが、カテゴリが異なっていることに注意してください。



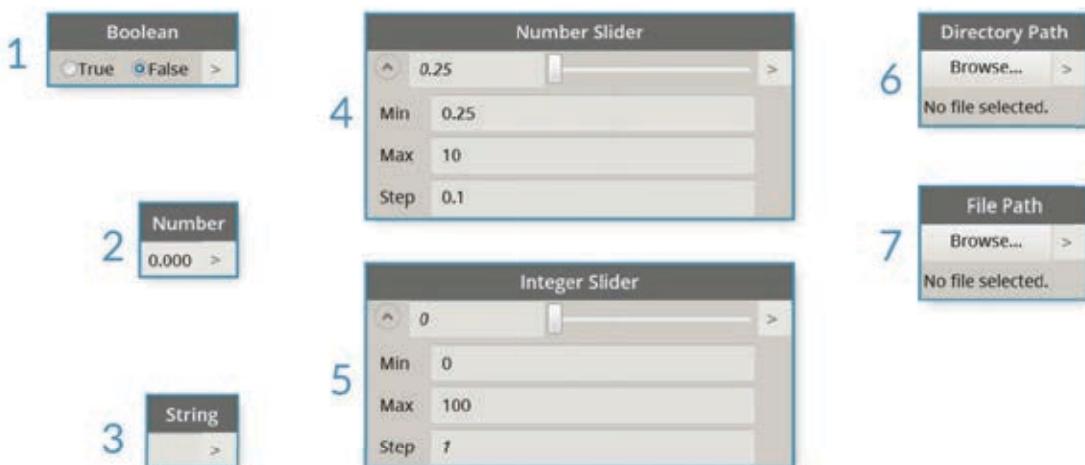
1. Point.ByCoordinates ノードと UV.ByCoordinates ノードは、名前は同じですがカテゴリが異なっています。
2. ほとんどのライブラリのノードには、カテゴリ形式が含まれています。
3. 注意する例外としては、Built-in Functions、Core.Input、Core.View、Operators などがあります。

### 頻繁に使用されるノード

Dynamo の既定のインストールには、数百個のノードが付属しています。では、ビジュアル プログラムを作成するために必要なノードはどれでしょうか。ここでは、プログラムのパラメータを定義するノード(Input)に注目し、ノードのアクション(Watch)の結果を確認して、ショートカット(Code Block)を使用して入力や機能を定義してみましょう。

#### Input ノード

Input ノードは、ビジュアル プログラムのユーザが重要なパラメータを使用する場合の主要な手段です。次の図は、Core ライブラリの Input カテゴリで使用可能なノードを示しています。

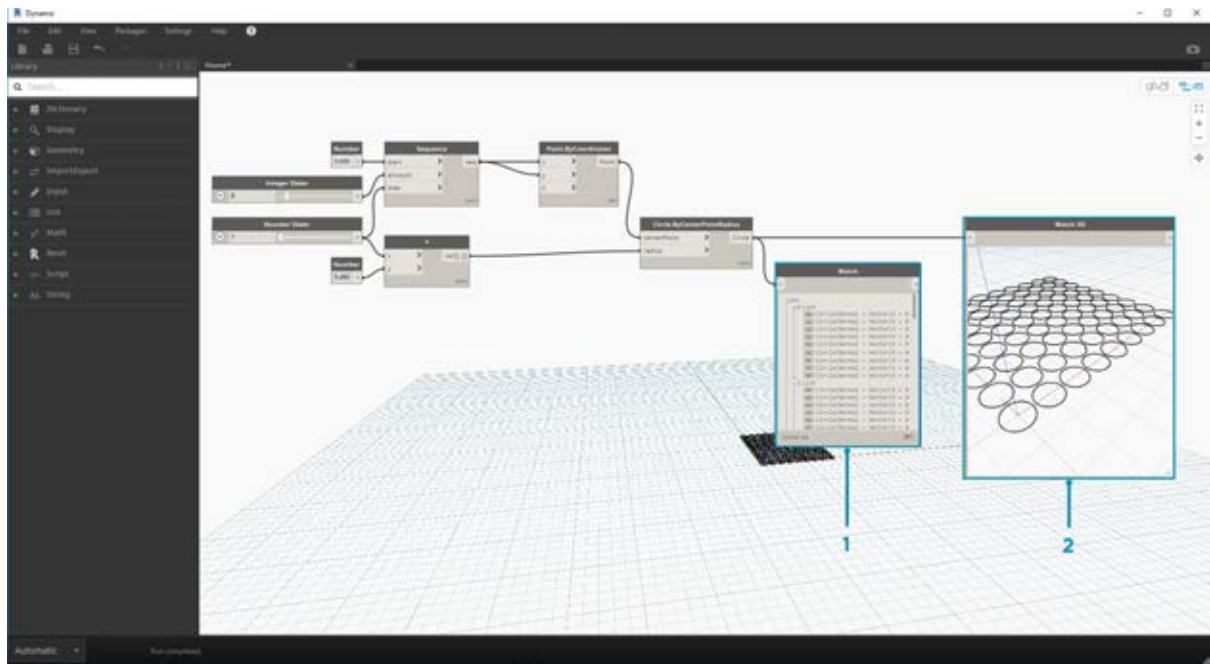


1. Boolean
2. Number
3. String
4. Number Slider
5. Integer Slider
6. Directory Path
7. File Path

#### Watch ノード

Watch ノードは、ビジュアル プログラムを経由してやり取りされるデータを管理するために必要なノードです。ノードの結果はノードデータのプレビューで表示できますが、Watch ノードで継続的に表示するか、Watch3D ノードでジオメトリの結果を表示することをお勧めします。これらのノードは、どちらも Core ライブラリの View カテゴリに含まれています。

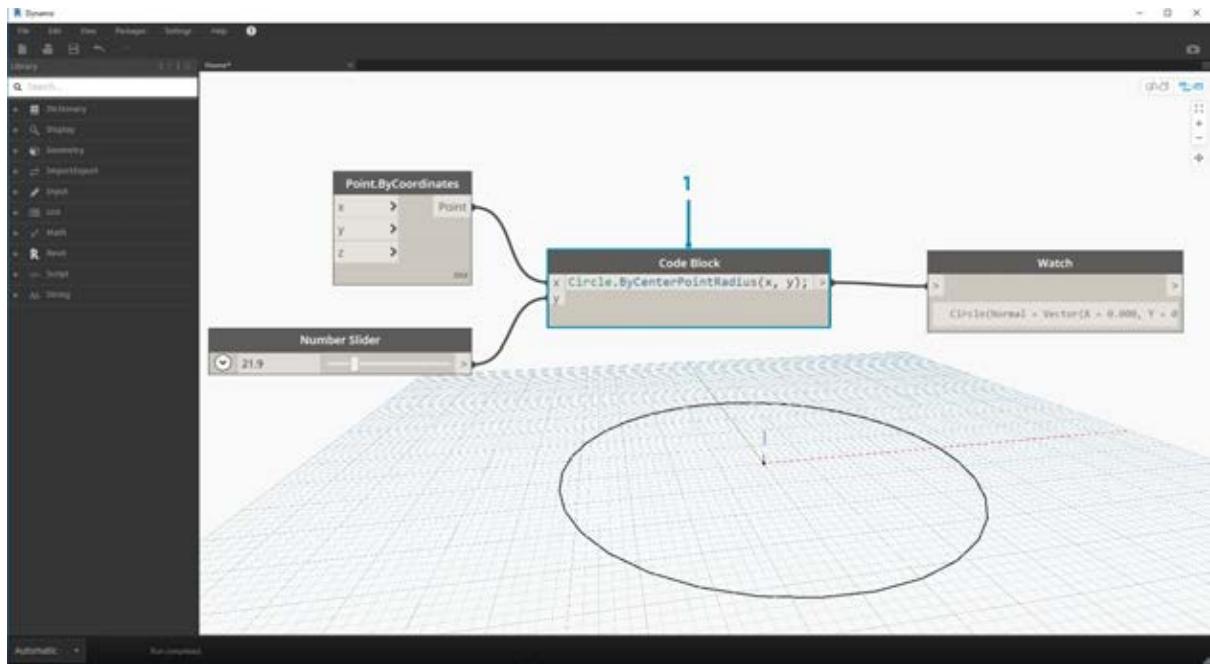
**ヒント:** ビジュアル プログラムに多数のノードが含まれている場合、3D プレビューの表示が見にくくなることがあります。その場合は、[設定]メニューの[背景 3D プレビューの表示]オプションの選択を解除し、Watch3D ノードを使用してジオメトリをプレビューすることをお勧めします。



1. Watch ノードで項目を選択すると、Watch3D プレビューと 3D プレビューでその項目がタグ付けされます。
2. Watch3D ノードの右下のグリップをグラブすると、3D プレビューの場合と同様にマウスを使用してサイズの変更とナビゲートを行うことができます。

#### Code Block ノード

Code Block ノードでセミコロン区切りの行を使用して、コード ブロックを定義することができます。これは、X/Y を使用する場合と同じくらいに簡単です。また、Code Block ノードをショートカットとして使用して Number Input ノードを定義したり、別のノードの機能を呼び出すこともできます。これを実行するための構文は、7.2 で紹介している Dynamo テキスト言語 DesignScript の命名規則に準拠します。では、このショートカットを使用して円を作成してみましょう。



1. ダブルクリックして **Code Block** ノードを作成します。
2. `Circle.ByCenterPointRadius(x, y);` と入力します。
3. ワークスペースをクリックして選択項目をクリアすると、`x` 入力と `y` 入力が自動的に追加されます。
4. **Point.ByCoordinates** ノードと **Number Slider** ノードを作成して **Code Block** の入力に接続します。
5. ビジュアルプログラムを実行すると、3D プレビューに円が表示されます。

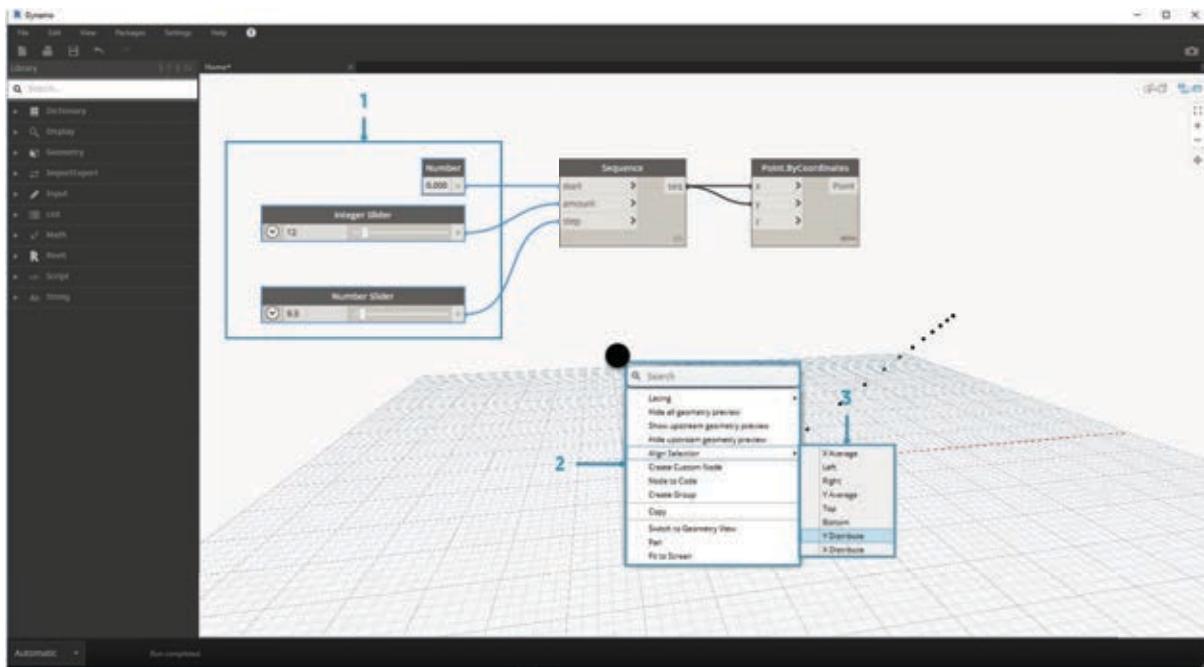
# プログラムを管理する

## プログラムを管理する

ビジュアル プログラミングは非常にクリエイティブな作業ですが、ワークスペースの複雑性やレイアウトの関係で、プログラム フローや主要なユーザ入力がすぐにわかりにくくなる場合があります。ここでは、プログラムの管理に関するベスト プラクティスをいくつか確認してみましょう。

### 位置合わせ

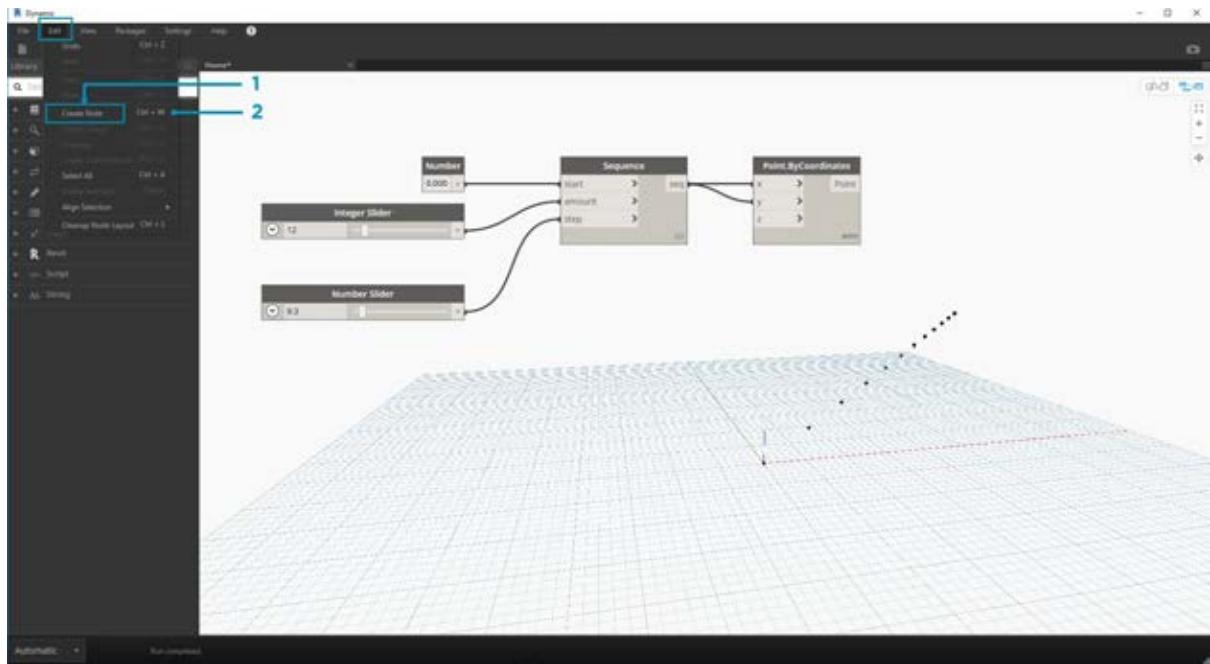
ワークスペースにある程度の数のノードを追加した場合は、画面を見やすくするためにノードのレイアウトを再編成することをお勧めします。複数のノードを選択してワークスペースを右クリックすると、X と Y で位置合わせオプションや分配オプションを指定するための [選択を位置合わせ] メニューがポップアップ ウィンドウに表示されます。



1. 複数のノードを選択します。
2. ワークスペースを右クリックします。
3. [選択を位置合わせ]オプションを使用します。

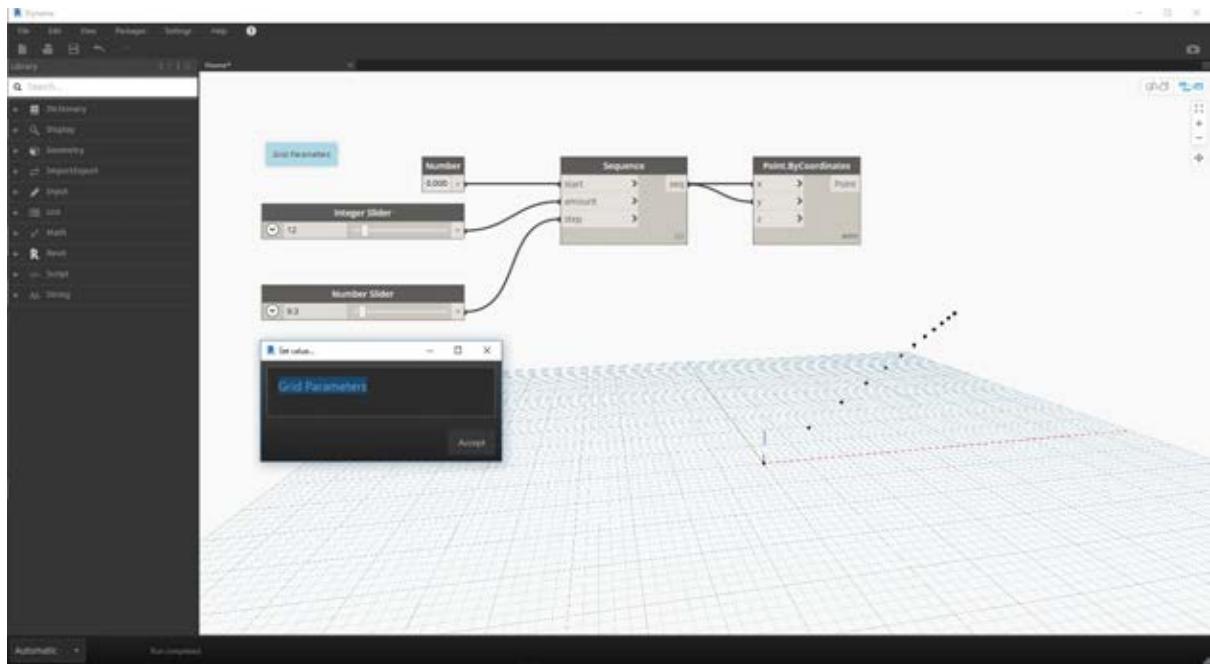
## ノート

Dynamo の操作に慣れてくると、ノード名とプログラムフローを確認することにより、ビジュアルプログラムを「読む」ことができるようになります。経験のレベルを問わず、すべてのユーザーにとって、わかりやすい名前と説明を入力することが重要になります。そのため、Dynamo には、編集可能なテキストフィールドが含まれているノートノードが用意されています。ノートをワークスペースに追加する方法は 2 つあります。



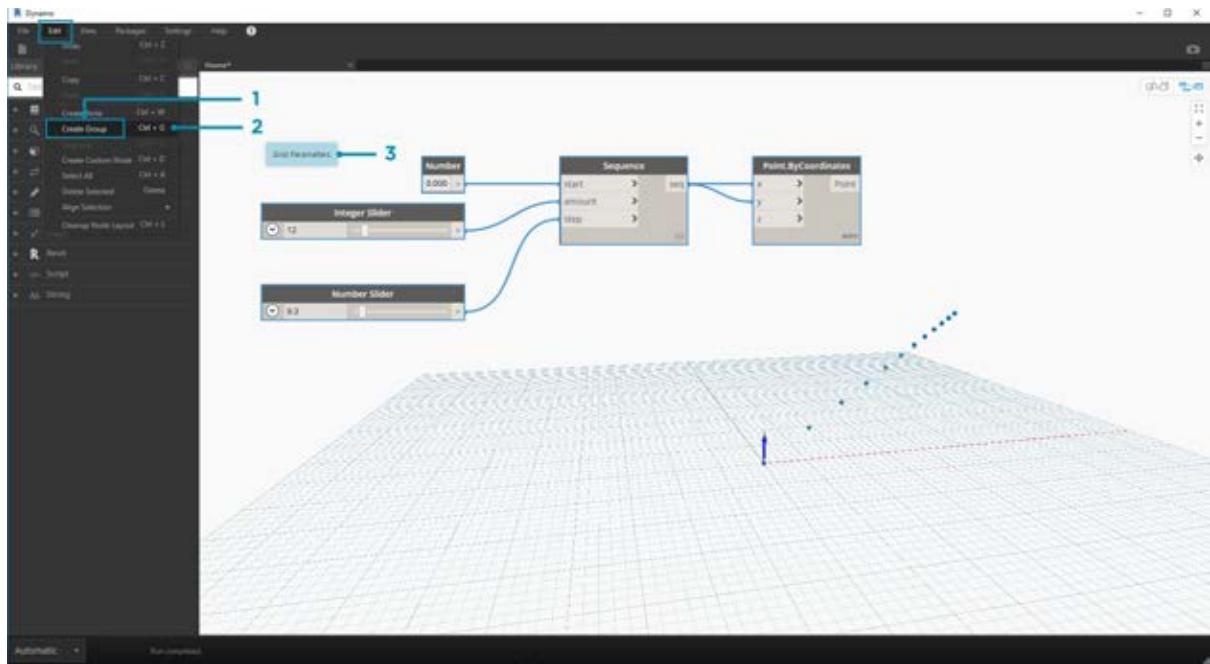
1. [編集] > [ノートを作成]メニューを参照します。
2. キーボード ショートカット[Ctrl]+[W]を使用します。

ワークスペースにノートが追加されると、ノート内のテキストを編集するためのテキスト フィールドがポップアップ表示されます。ノートが作成されると、ノート ノードをダブルクリックするか右クリックしてノートを編集できるようになります。



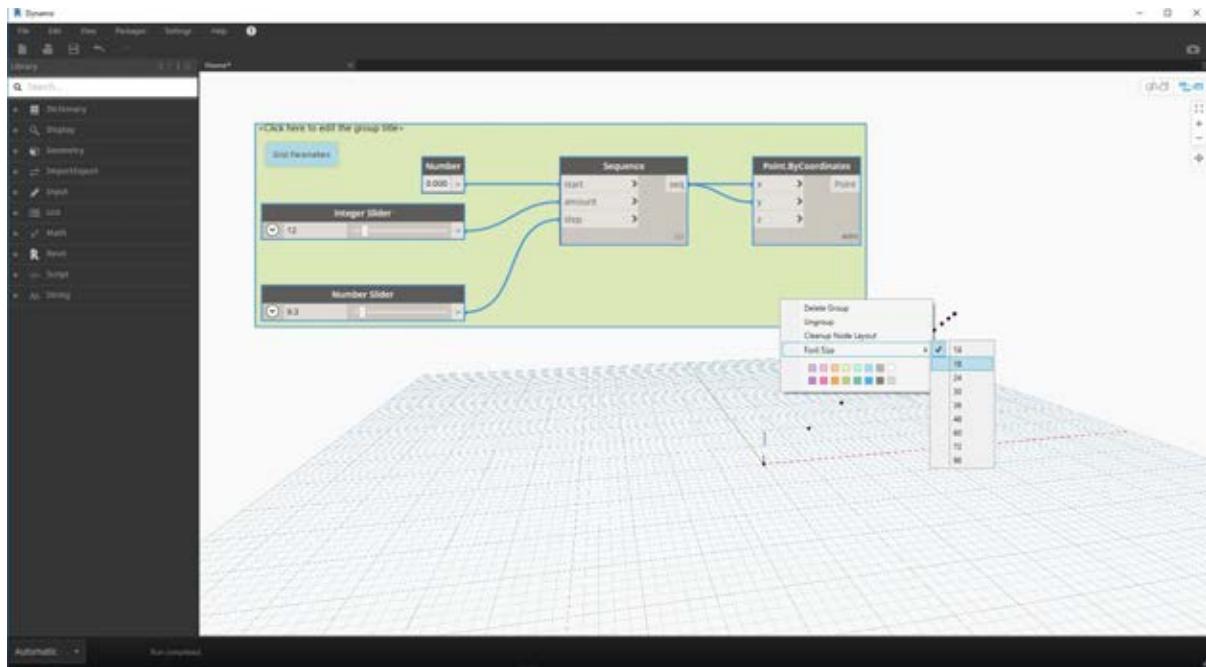
## グループ化

ビジュアル プログラムのサイズが大きい場合、大まかな実行手順が特定できると便利です。グループを使用してノードの集合をハイライト表示し、色付きの長方形にタイトルが表示されたラベルを付けることができます。複数のノードを選択してグループを作成する方法は 3 つあります。



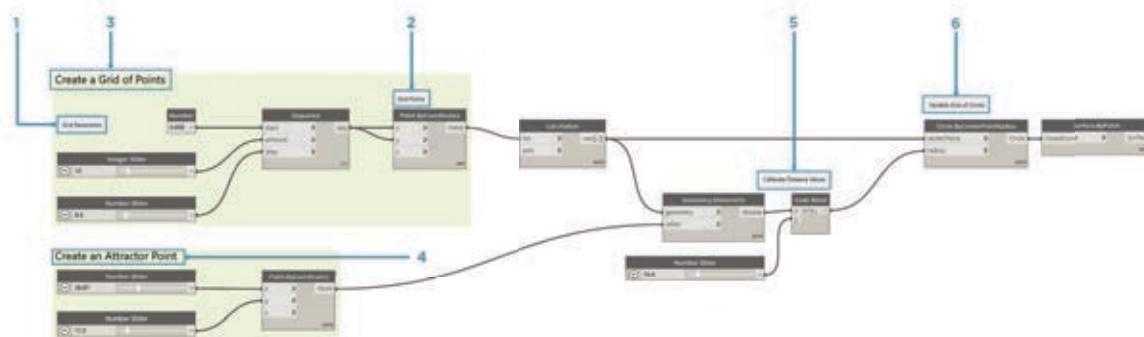
1. [編集] > [グループを作成]メニューを参照します。
2. キーボード ショートカット[Ctrl]+[G]を使用します。
3. ワークスペースを右クリックし、[グループを作成]を選択します。

作成したグループについては、タイトルやカラーなどの設定を編集できます。



ヒント: ノードとグループの両方を使用すると、わかりやすい注釈をファイルに簡単に追加することができます。

これはセクション 2.4 で使用したプログラムですが、ノートとグループが追加されています。



1. ノート: 「Grid Parameters」
2. ノート: 「Grid Points」
3. グループ: 「Create a Grid of Points」
4. グループ: 「Create an Attractor Point」
5. ノート: 「Calibrate Distance Values」
6. ノート: 「Variable Grid of Circles」

## プログラムの構成要素

### プログラムの構成要素

ここまで説明で、ビジュアル プログラミングの基本的な仕組みについて確認しました。ここからは、ビジュアル プログラミングで使用する構成要素について詳しく見ていきましょう。この章では、Dynamo 内で受け渡されるデータの基本的な概念について説明します。



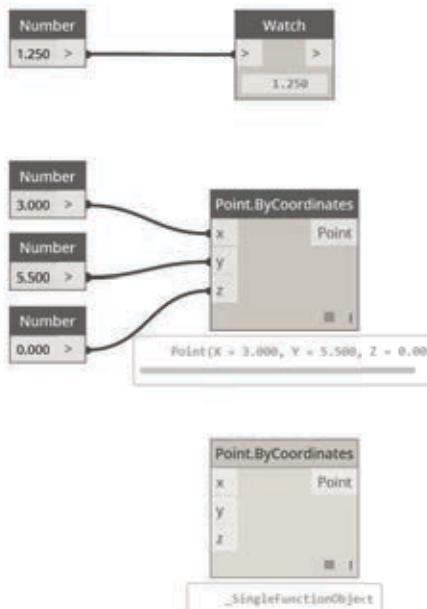
# データ

## データ

データは、各種プログラムの構成要素です。データはワイヤ経由でやり取りされ、そのデータを処理するノードに入力情報を提供します。処理されたデータは、新しい形式の出力データになります。ここでは、データの定義と構造について確認し、Dynamoで実際にデータを使用してみます。

### データの概要

データとは、定性的な変数や定量的な変数の集まりのことです。最も単純な形式のデータは、0, 3, 14, 17などの数値です。この他にも、変化する数値を表す変数(height)、文字(myName)、ジオメトリ(Circle)、データ項目のリスト(1, 2, 3, 5, 8, 13, ...)など、さまざまな種類のデータがあります。Dynamoのノードの入力ポートには、データを追加する必要があります。アクションが設定されていないデータを使用することはできますが、ノードが表すアクションを処理する場合はデータが必要になります。ノードをワークスペースに追加する際に、そのノードに対して入力データを指定しなかった場合、結果として関数が返されます。アクション自体の結果は返されません。



1. 単純なデータ
2. データとアクション(ノード)が正常に実行される
3. 入力データが指定されていないアクション(ノード)から一般的な関数が返される

### NULLに関する注意事項

'null'というタイプは、データが存在しないことを表します。これは抽象的な概念ですが、ビジュアルプログラミングを使用しているとよく登場します。たとえば、アクションを実行して正しい結果が生成されなかった場合に、ノードから NULL が返されます。安定したプログラムを作成するには、NULL をテストし、データ構造から NULL を削除することが重要になります。

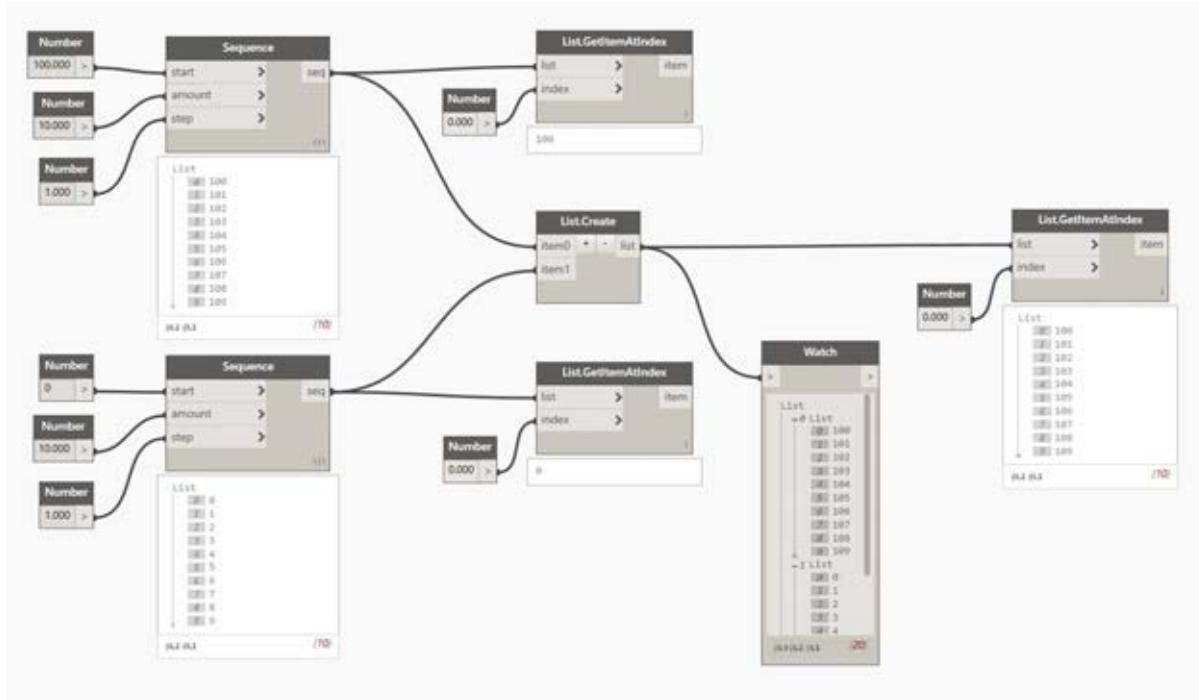


## データ構造

ビジュアル プログラミングを行うと、大量のデータを非常に短時間で生成できるため、データ階層を管理するための手段が必要になります。これを行うには、データ構造を使用します。データ構造とは、データを格納するための構造化されたスキームのことです。データ構造の仕様と使用方法は、プログラミング言語によって異なります。Dynamo の場合、リストを使用してデータに階層を追加します。これについては、これ以降の章で詳しく説明します。ここでは、簡単な操作から始めてみましょう。

リストは、1 つのデータ構造内に配置された項目の集合を表します。リストと項目の関係は次のようにになります。

- 私の手(リスト)には 5 本の指(項目)がある。
- 私の家の前の通り(リスト)には 10 軒の家(項目)が建っている。



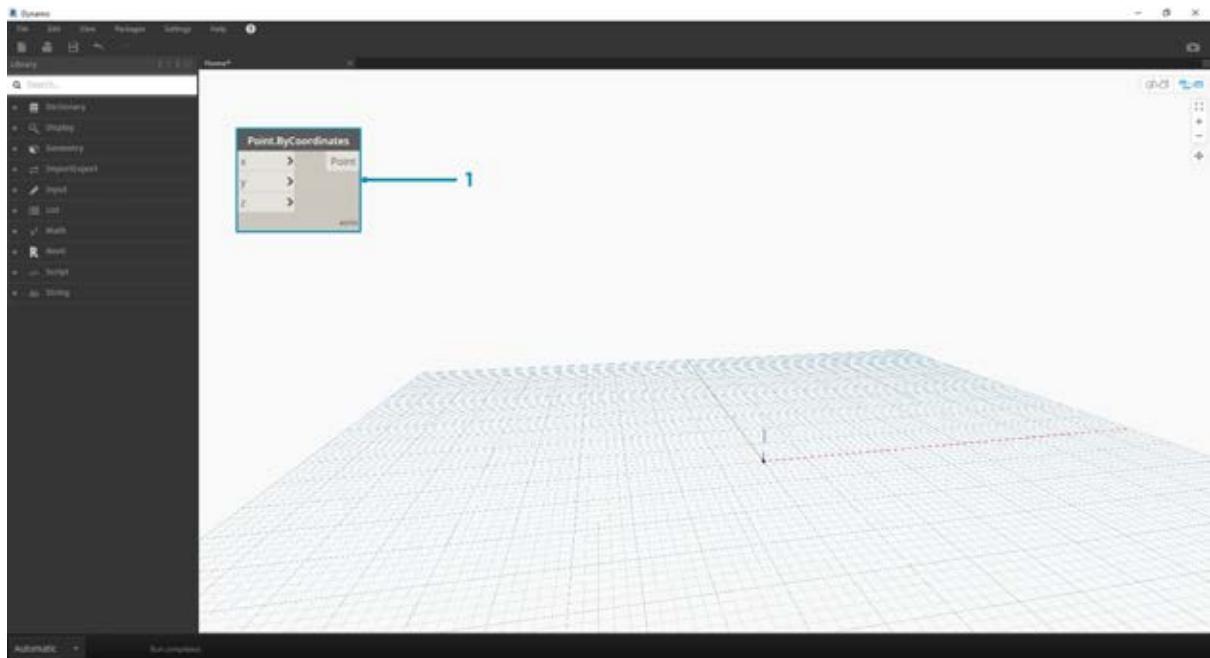
1. Number Sequence ノードは、*start* 入力、*amount* 入力、*step* 入力を使用して数値のリストを定義します。上の図の 2 つのノードを使用して、10 個の数値を格納する 2 つのリストを作成しました。一方のリストには 100 から 109 までの数値が格納され、もう一方のリストには 0 から 9 までの数値が格納されています。
2. List.GetItemAtIndex ノードは、特定のインデックスを使用して、リスト内の特定の項目を選択します。*0* を選択すると、リスト内の最初の項目(この場合は 100)が取得されます。
3. もう一方のリストでも同じ手順を実行すると、リスト内の最初の項目である 0 が取得されます。
4. 次に、List.Create ノードを使用して、2 つのリストをマージして 1 つのリストを作成します。このノードによって「リストのリスト」が作成されることに注意してください。これにより、データの構造が変化します。
5. インデックスを 0 に設定してもう一度 List.GetItemAtIndex ノードを使用すると、「リストのリスト」内の最初の項目が取得されます。この場合、リストが項目として処理されます。この動作は、他のスクリプト言語とは多少異なっています。リストの操作とデータ構造については、これ以降の章でさらに詳しく説明します。

Dynamo のデータ階層を理解するには、「データ構造については、リストが項目として認識される」という概念を理解する必要があります。つまり、Dynamo は、トップダウン プロセスを使用してデータ構造を認識するということです。こうした考え方について、例を使用して詳しく説明します。

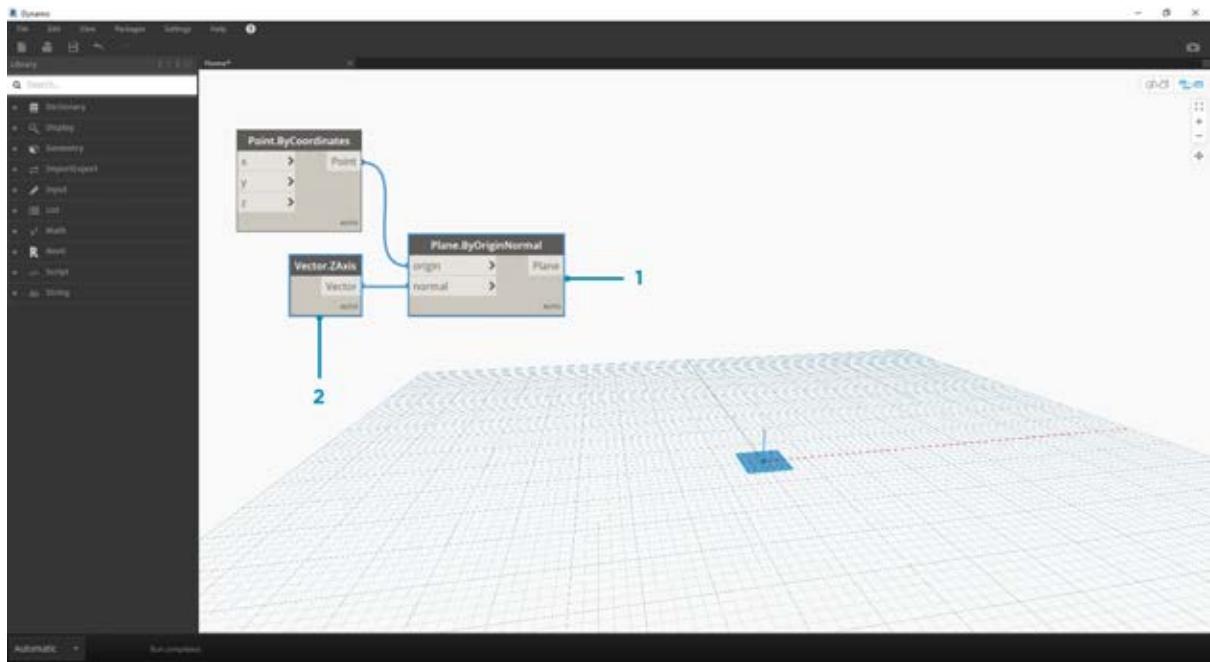
### データを使用して円柱の配列を作成する

この演習に付属しているサンプル ファイルをダウンロードしてください(右クリックして[名前を付けてリンク先を保存...]を選択): [Building Blocks of Programs - Data.dyn](#)。すべてのサンプル ファイルの一覧については、付録を参照してください。

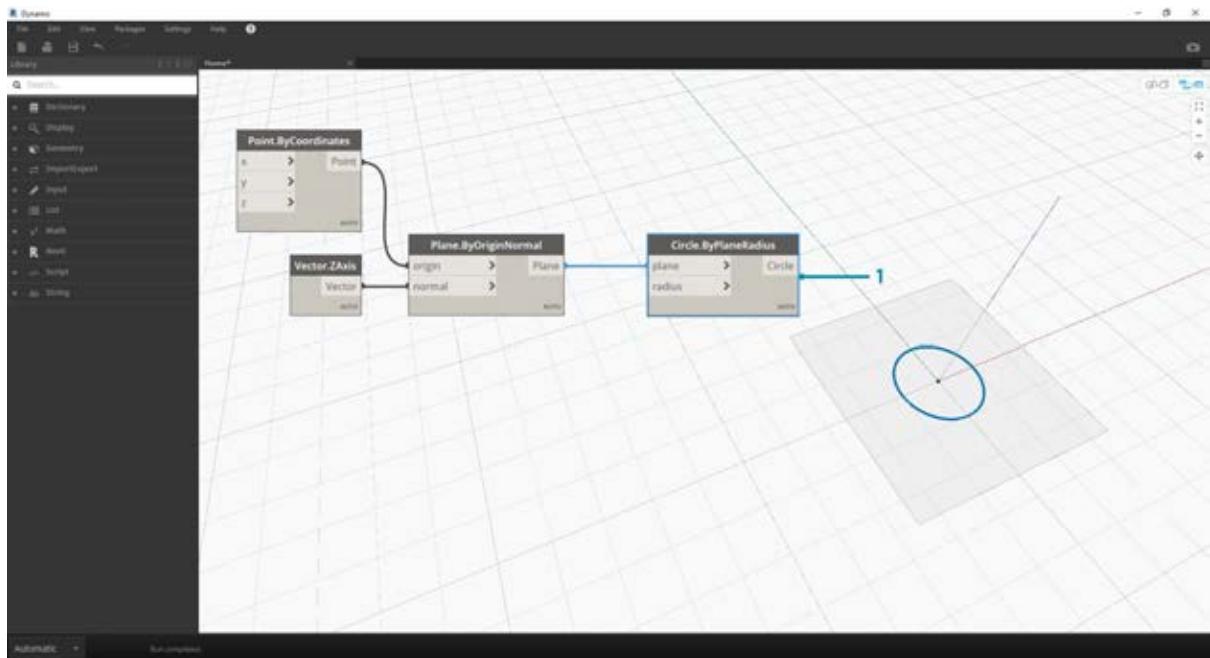
この最初の例では、ジオメトリ階層全体にわたるシエル化された円柱を作成します。ジオメトリ階層については、このセクションで説明します。



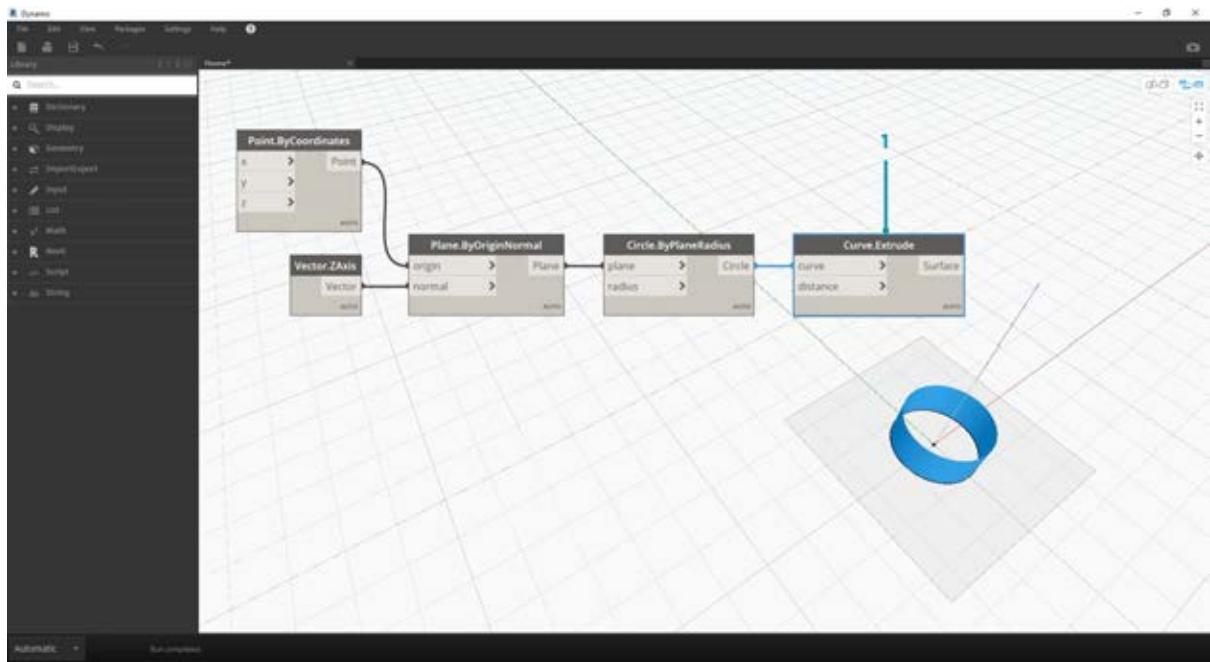
1. **Point.ByCoordinates** - このノードをキャンバスに追加すると、Dynamo プレビュー グリッドの基準点に点が表示されます。x,y 入力と z 入力の既定値は 0.0 です。この場合、Dynamo プレビュー グリッドの基準点に点が表示されます。



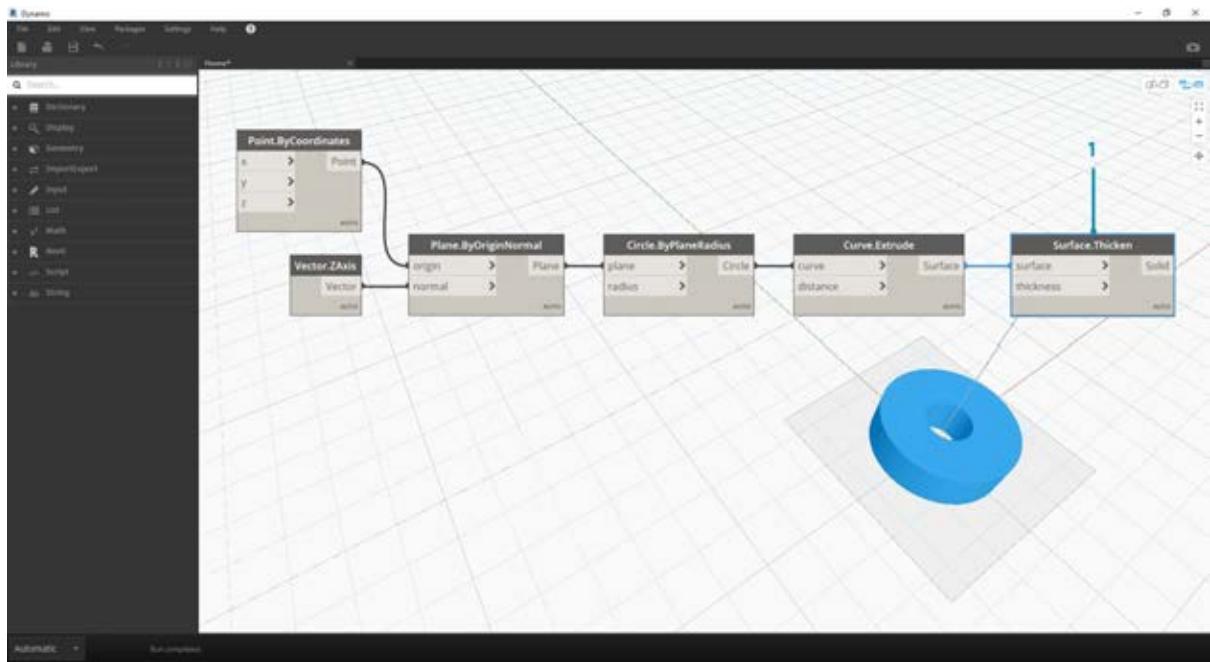
1. **Plane.ByOriginNormal** - ジオメトリ階層内の次の段階は、平面です。平面を作成する方法はいくつかありますが、ここでは、基準点と法線を入力して作成します。ここで使用する基準点は、前の手順で作成した点ノードです。
2. **Vector.ZAxis** - このノードは、Z 方向の単位ベクトルです。このノードには入力はありません、 $[0,0,1]$ という値のベクトルしかないように注意してください。このノードは、*Plane.ByOriginNormal* ノードの *normal* の入力として使用します。これにより、Dynamo のプレビューに長方形の平面が表示されます。



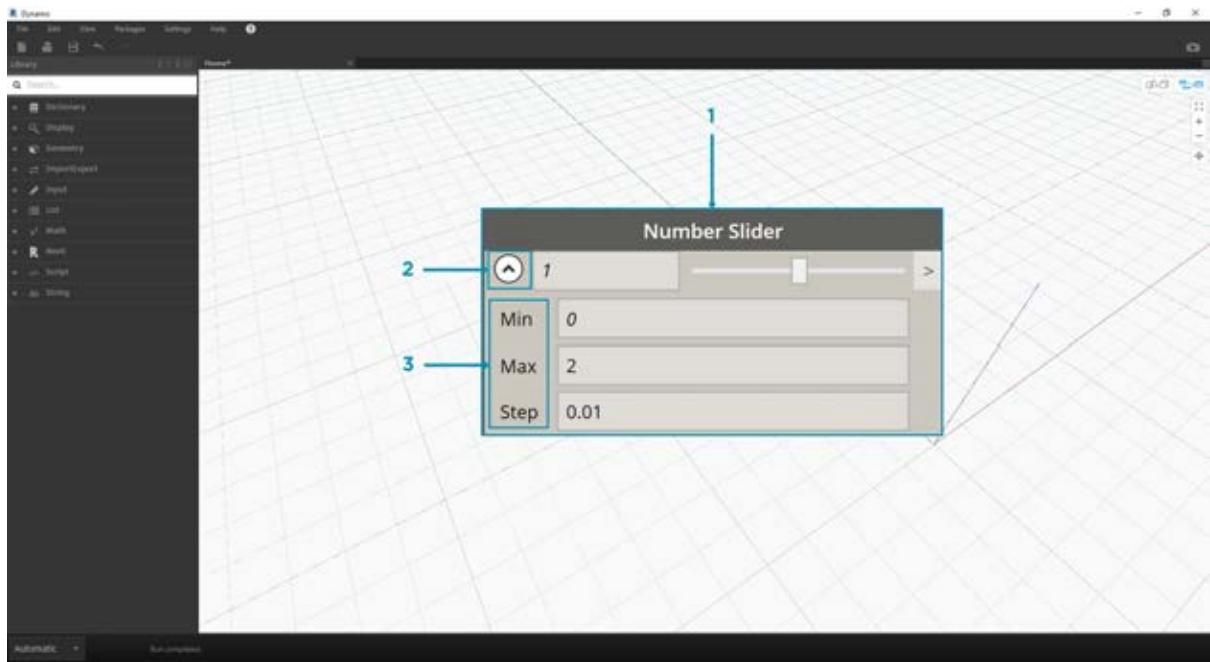
1. **Circle.ByPlaneRadius** - 次に、前の手順で作成した平面から曲線を作成して階層を拡張します。このノードに接続すると、基準点に円が表示されます。このノードの半径の既定値は 1 です。



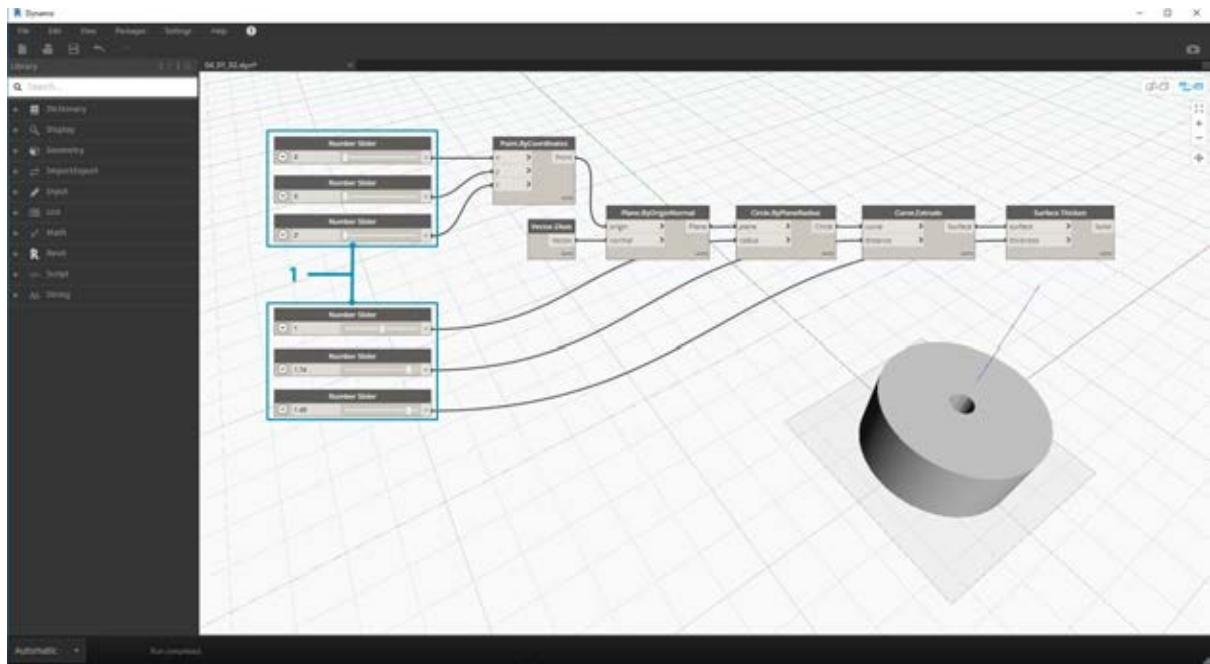
1. **Curve.Extrude** - このノードに深さを設定することにより、基準点に表示されている円を立体的な 3 次元の円柱にします。このノードは、曲線を押し出してサーフェスを作成します。このノードの距離の既定値は 1 です。この場合、ビューポートに円柱が表示されます。



1. **Surface.Thicken** - このノードは、指定の距離でサーフェスをオフセットして形状を閉じることにより、閉じたソリッドを作成します。厚さの既定値は 1 です。この場合、その値に従って、シェル化された円柱がビューポートに表示されます。

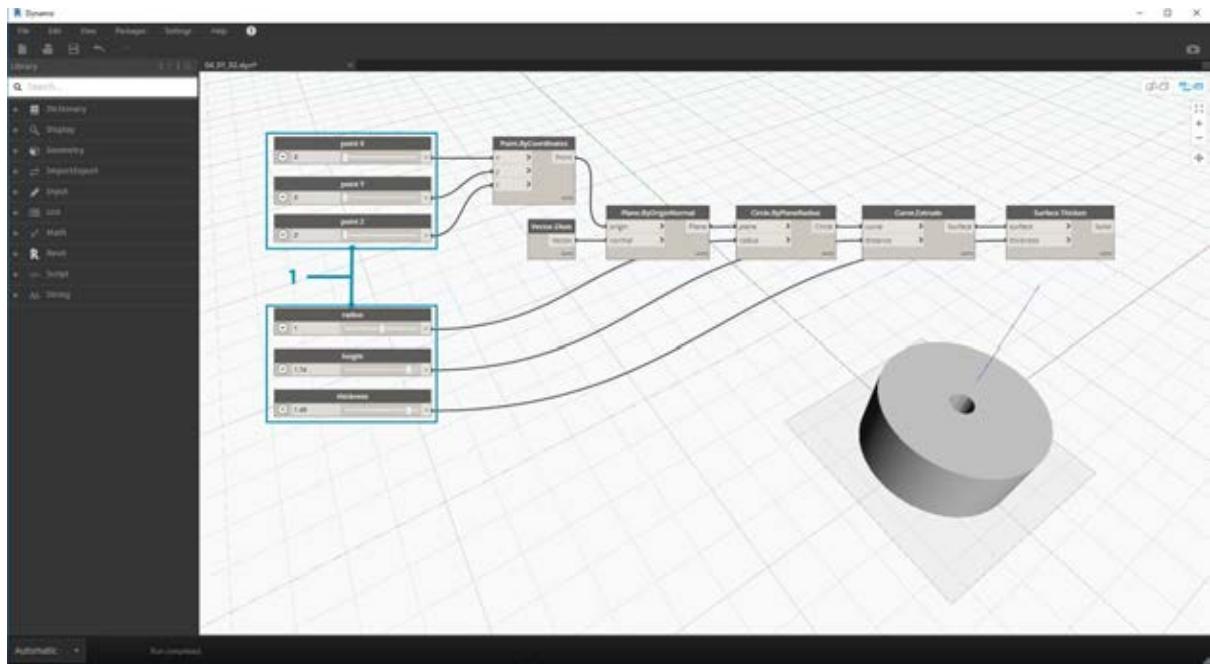


1. **Number Slider** - ここでは、すべての入力に対して既定値を使用するのではなく、パラメータ制御をモデルに追加してみましょう。
2. 範囲編集用キャレット - 数値スライダをキャンバスに追加したら、左上のキャレットをクリックして範囲オプションを表示します。
3. **Min/Max/Step** - *Min*, *Max*, *Step*の値を、それぞれ 0, 2, 0.01 に変更します。これにより、ジオメトリ全体のサイズをコントロールします。



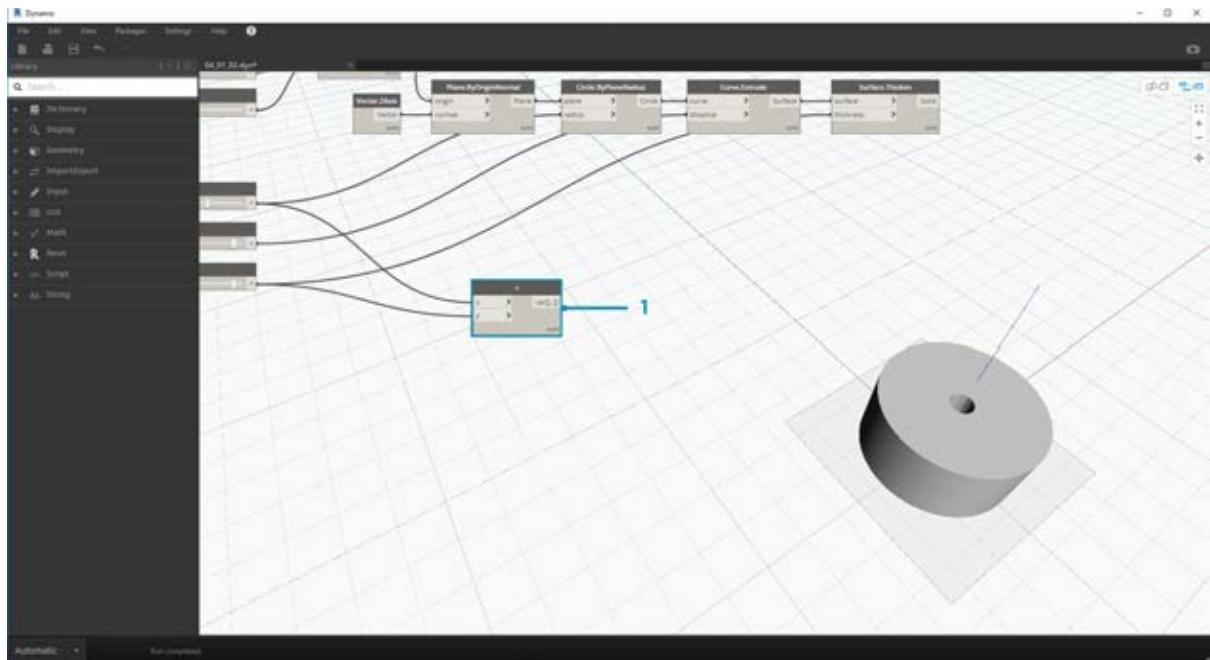
1. **Number Slider** - すべての既定の入力で、この数値スライダのコピー アンド ペースト操作を繰り返します(Number Slider ノードを選択して[Ctrl]+[C]を押し、次に[Ctrl]+[V]を押す)。この操作は、既定値が設定されているすべての入力に Number Slider ノードが接続されるまで繰り返します。定義を有効にするには、一部のスライダ値をゼロより大きな値に設定する必要があります。たとえば、サーフェスに厚みを付ける場合は、押し出しの深さをゼロより大きな値に指定する必要があります。

これで、数値スライダを使用したパラメータ制御のシェル化された円柱が作成されました。いくつかのパラメータの値を変更して、Dynamo のビューポート内でジオメトリが動的に更新されることを確認してください。

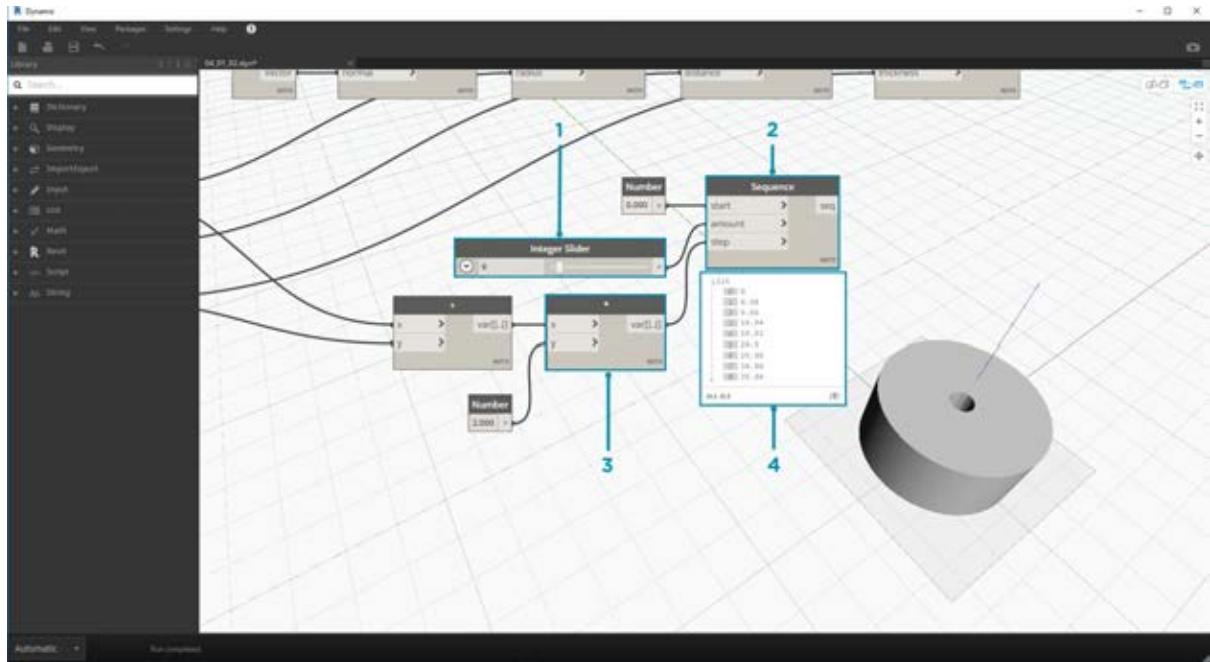


1. Number Slider - これまでの手順で、多くのスライダをキャンバスに追加してきました。そのため、作成したツールのインターフェースをクリーンアップする必要があります。各スライダを右クリックして[Rename...]を選択し、そのスライダのパラメータに適した名前に変更します。どのような名前が適しているかについては、上の図を参照してください。

ここまで手順で、厚みのある円柱が作成されました。この時点では、この円柱はまだ 1 つのオブジェクトにすぎません。ここからは、動的にリンクされた円柱の配列を作成していきます。そのためには、項目を 1 つずつ操作するのではなく、円柱のリストを作成します。

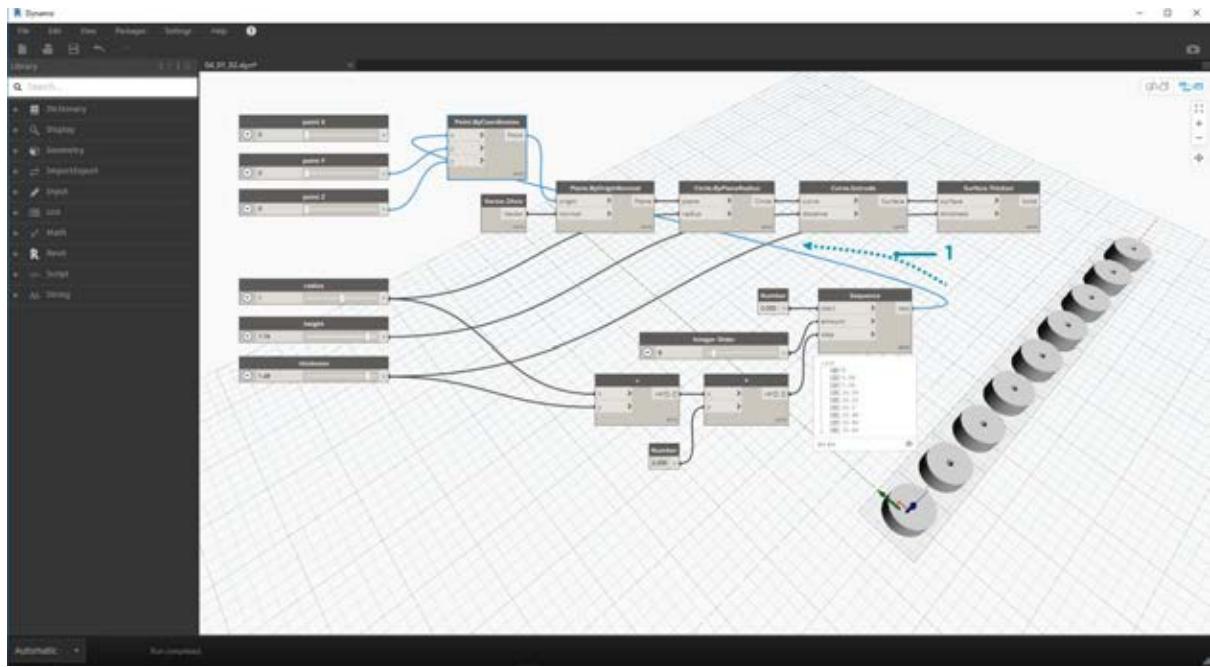


1. 追加用ノード (+) - ここでの目的は、前の手順で作成した円柱の横に、一連の円柱を追加することです。現在の円柱の横に別の円柱を追加するには、円柱の半径とシェルの厚さの両方を考慮する必要があります。ここでは、2つのスライダの値を追加して、円柱の半径とシェルの厚さを設定します。

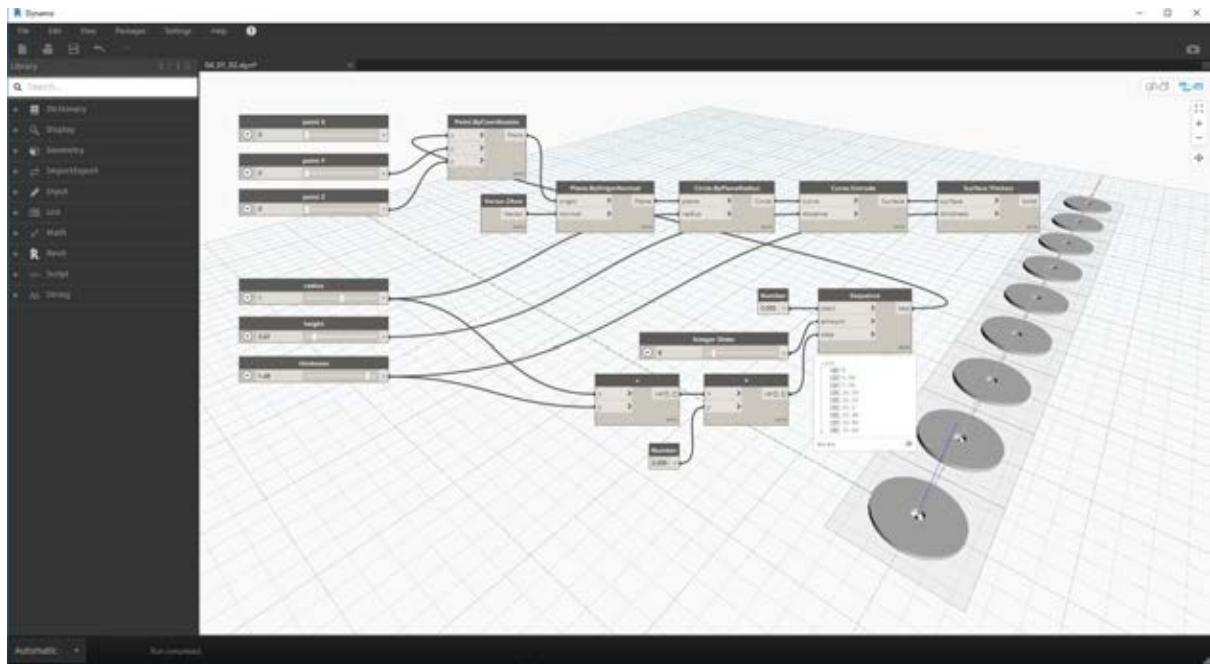


この操作は少し複雑であるため、手順を追って詳しく説明します。最終的な目標は、連続する各円柱の位置を定義する数値のリストを作成することです。

1. 乗算用ノード (\*) - 最初に、前の手順で指定した値を 2 倍にします。前の手順で指定した値は半径でしたが、ここでは円柱の直径を指定します。
2. Sequence - 次に、このノードを使用して、数値の配列を作成します。最初の入力は、前の手順の乗算用ノード() の step の値です。start の値は、Number ノードを使用して  $0.0*$  に設定します。
3. Integer Slider - Integer Slider ノードを接続して amount の値を設定します。これにより、作成される円柱の数が決まります。
4. 出力のリスト - このリストには、配列内の各円柱の移動距離が表示されます。このリストは、元のスライダのパラメータによって制御されます。



1. この手順は簡単です。前の手順で指定したシーケンスを、元の *Point.ByCoordinates* ノードの *x* 入力に接続するだけです。この操作により、*pointX* スライダが置き換えられます。このスライダは、削除してかまいません。これで、ビューポートに円柱の配列が表示されます(Integer Slider ノードで 0 よりも大きな値を設定してください)。



円柱の配列は、すべてのスライダに動的にリンクされたままの状態になっています。各スライダの値を変更して、円柱の定義がどのように更新されるかを確認してください。

# 数学的方法

## 数学的方法

データの最も単純な形式が数値だとするならば、数値を関連づける最も簡単な方法は数学的方法を活用することです。割り算のような単純な演算子から、三角関数などの複雑な計算式まで、数学的方法は数値の関係とパターンを調べるために非常に便利な方法です。

### 算術演算子

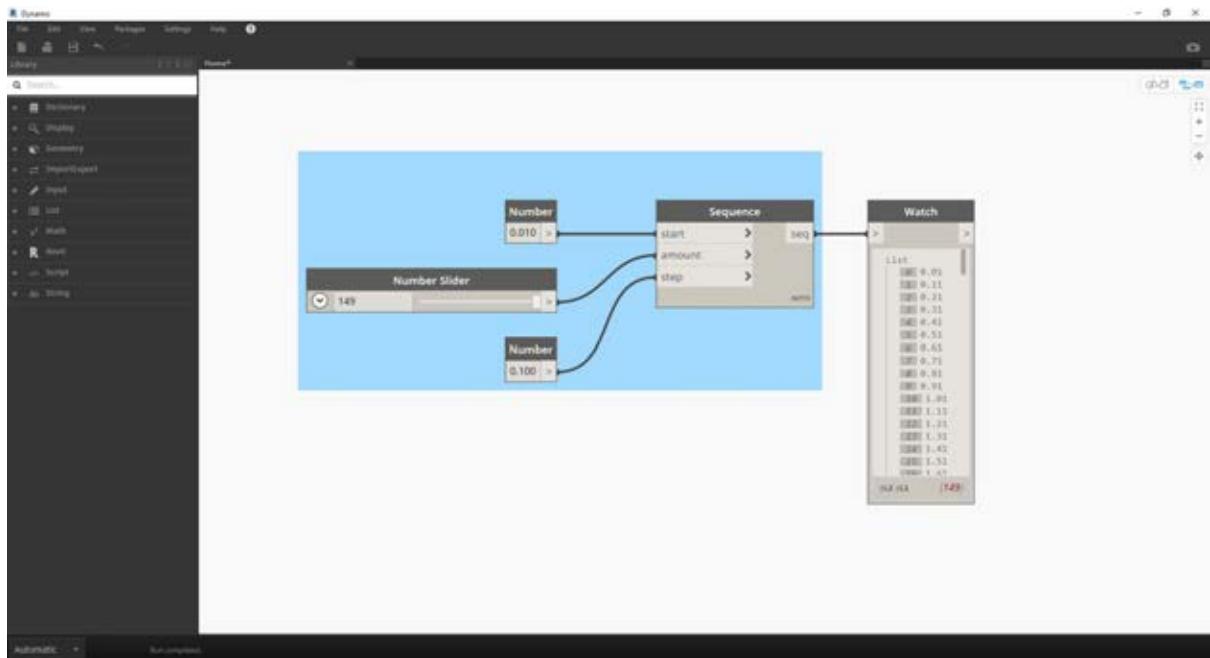
演算子は、代数関数と2つの入力値を組み合わせて使用する一連のコンポーネントであり、1つの出力値を生成します(加算、減産、乗算、除算など)。演算子は、[Operators] > [Actions]で使用できます。

アイコン	名前	構文	入力	出力
	Add	+	var[]...[], var[]...[] var[]...[]	
	Subtract	-	var[]...[], var[]...[] var[]...[]	
	Multiply	*	var[]...[], var[]...[] var[]...[]	
	Divide	/	var[]...[], var[]...[] var[]...[]	

### パラメータを使用する計算式

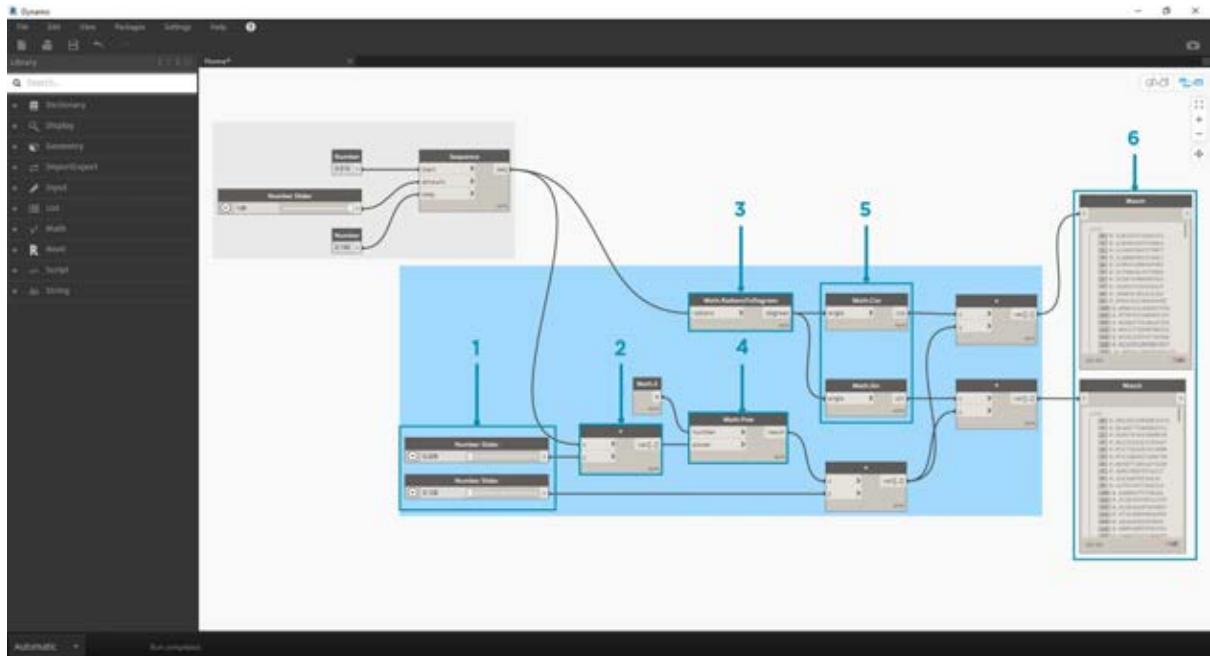
この演習に付属しているサンプル ファイルをダウンロードしてください(右クリックして[名前を付けてリンク先を保存...]を選択): [Building Blocks of Programs - Math.dyn](#)。すべてのサンプルファイルの一覧については、付録を参照してください。

[Operators]で演算子と変数を組み合わせて計算式を作成することにより、複雑な関係を定義することができます。ここでは、スライダなどの入力パラメータによってコントロールできる計算式を作成してみましょう。



1. Number Sequence ノードで、*start*、*amount*、\*\**step* という 3 つの入力に基づいて数列を定義します。この数列は、パラメータ式内の「t」を表しています。ここでは、サイズの大きなリストを使用して、らせん構造を定義します。

上記の手順により、パラメータ領域を定義するための数値のリストが作成されます。黄金螺旋は、 $x = r \cos \theta = a \cos \theta e^{b\theta}$  と  $y = r \sin \theta = a \sin \theta e^{b\theta}$  の等式として定義されます。下図のノードグループは、この等式をビジュアルプログラミングの形式で表現しています。

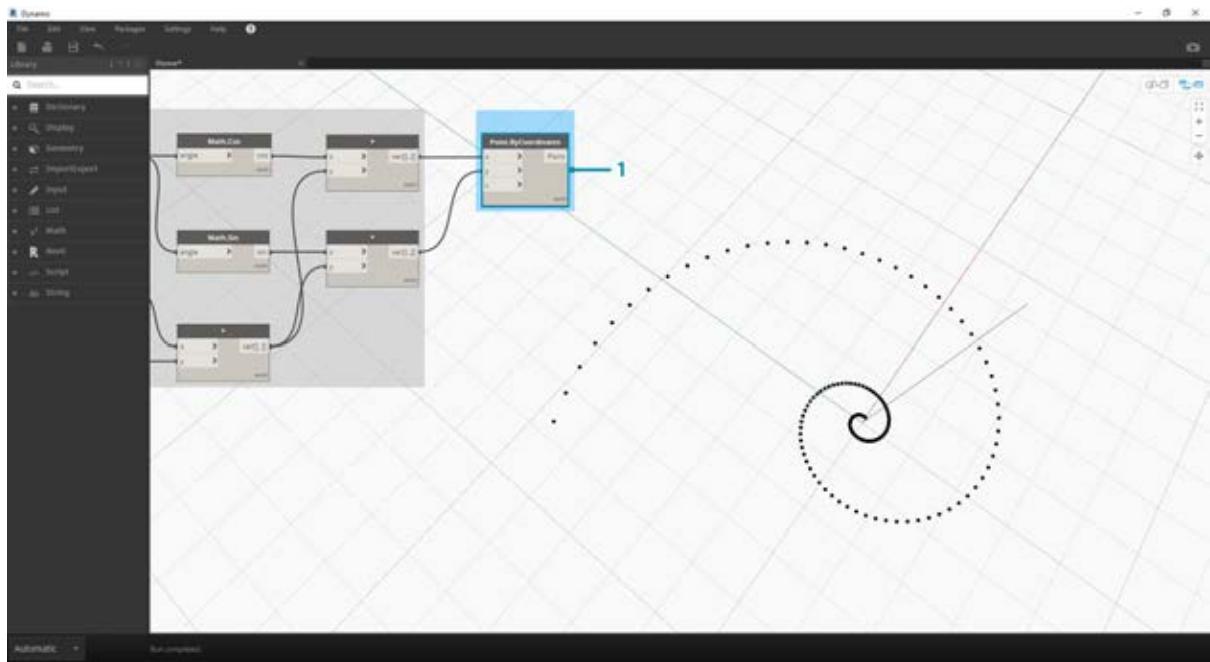


ここからは、順を追ってノード グループを確認していきます。ビジュアル プログラムと記述形式の等式との相違点に注意してください。

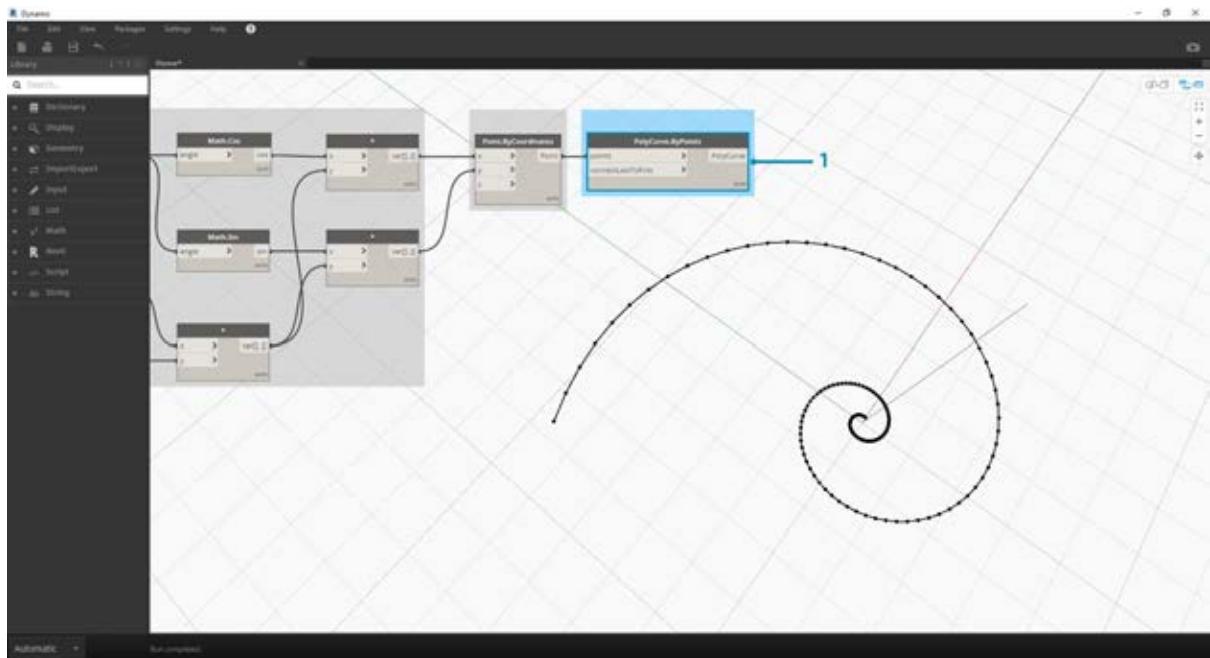
1. Number Slider ノードを使用して、キャンバスに 2 つの数値スライダを追加します。これらのスライダは、パラメータ式の変数である「 $a$ 」と「 $b$ 」を表します。これらの変数は、柔軟に変化する定数や、必要な結果に合わせて調整可能なパラメータを表します。
2. 「\*」ノードは、乗算を表します。このノードを繰り返し使用して、乗算変数を接続します。
3. Math.RadiansToDegrees ノードで「 $t$ 」の値を角度に変換して、三角関数内でその角度を評価します。Dynamo では、三角関数の評価には角度が既定で使用されます。
4. Math.Pow ノードは、「 $t$ 」および数値「 $e$ 」の関数として動作し、フィボナッチ数列を作成します。
5. Math.Cos と Math.Sin ノードは、三角関数です。パラメータ指定の各点の X 座標と Y 座標を識別します。
6. Watch ノードで、出力値として 2 つのリストが生成されたことを確認します。これらのリストが、らせんの生成に使用される点群の x 座標と y 座標になります。

#### 計算式からジオメトリへ

上記の手順でも問題なく機能しますが、多数のノードを使用するため、手間がかかります。より効率的なワークフローを作成するには、セクション 3.3.2.3 のコード ブロックに関する説明で、Dynamo 表現の文字列を 1 つのノード内に定義する方法を参照してください。次の手順では、パラメータ式を使用してフィボナッチ曲線を描画する方法について確認します。



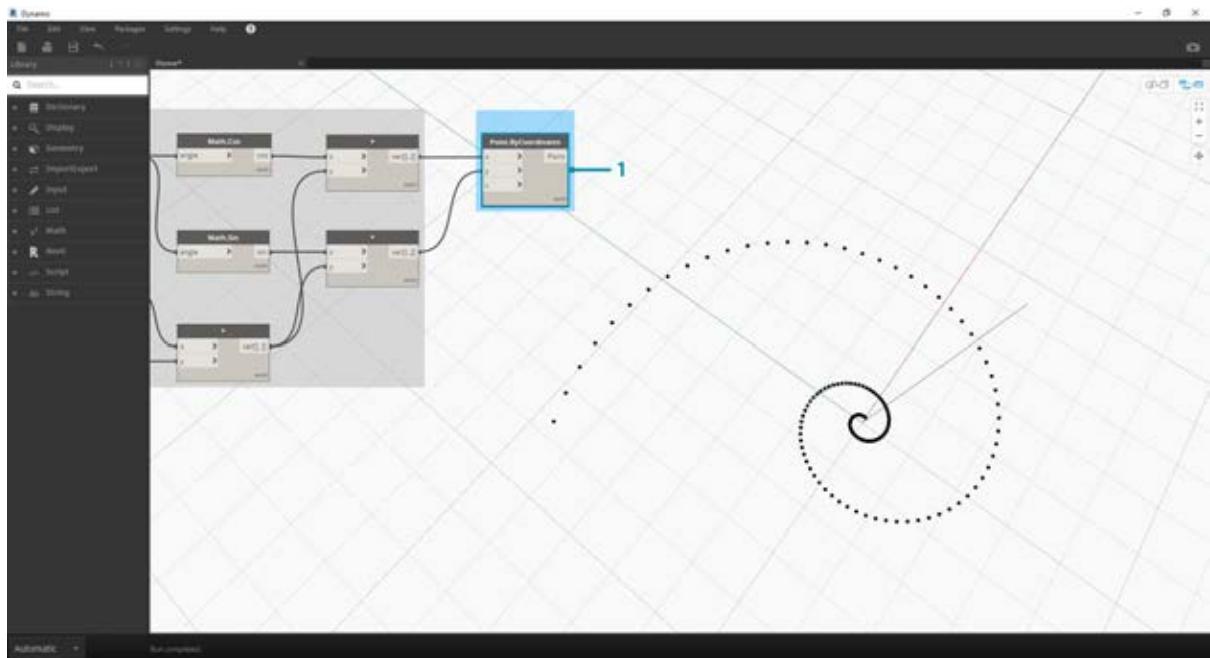
1. Point.ByCoordinates ノードの  $x$  入力に上部の乗算ノードを接続し、 $y$  入力に下部の乗算ノードを接続します。この操作により、パラメータで制御された点群のらせん構造が画面上に表示されます。



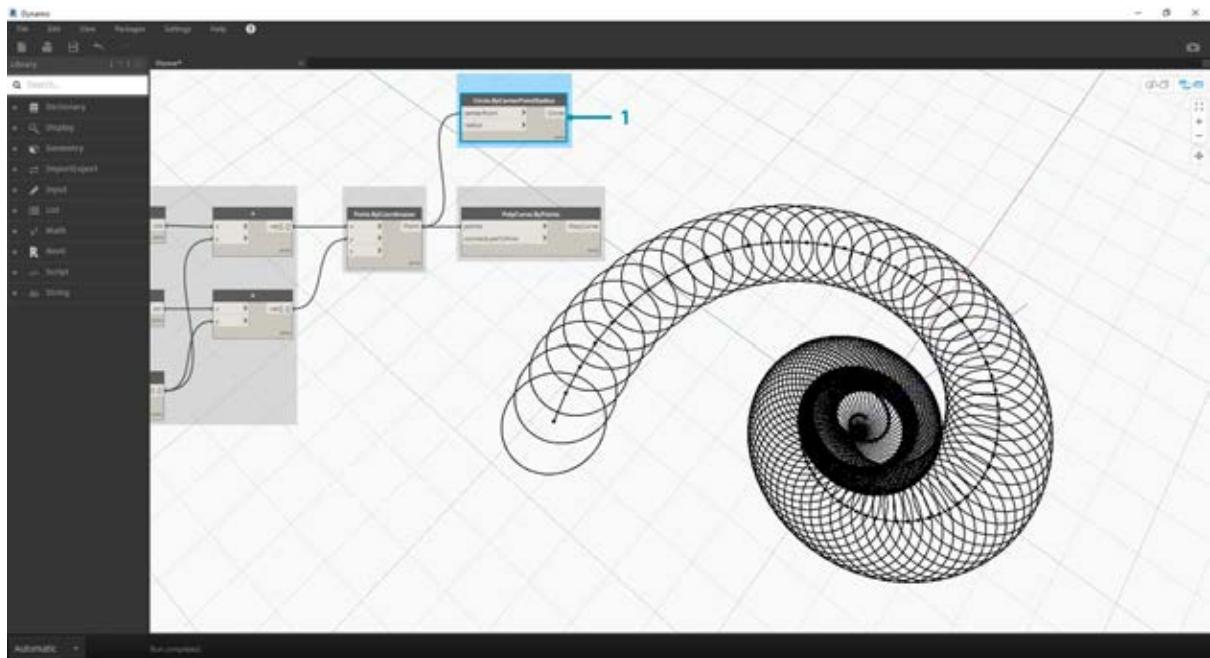
1. **Polycurve.ByPoints** ノードの *points* 入力に **Point.ByCoordinates** ノードの出力を接続します。ここでは閉曲線は作成しないため、*connectLastToFirst* 入力には何も接続しなくてかまいません。この操作により、前の手順で定義した各点を通過するらせん構造が作成されます。

これで、フィボナッチ曲線が作成されました。ここからは、2つの演習で「オウムガイ」と「ヒマワリ」を作成してみましょう。これらは自然界にみられる形狀を抽象的に表しているばかりではなく、フィボナッチ曲線の2つの異なる適用例を的確に表現しています。

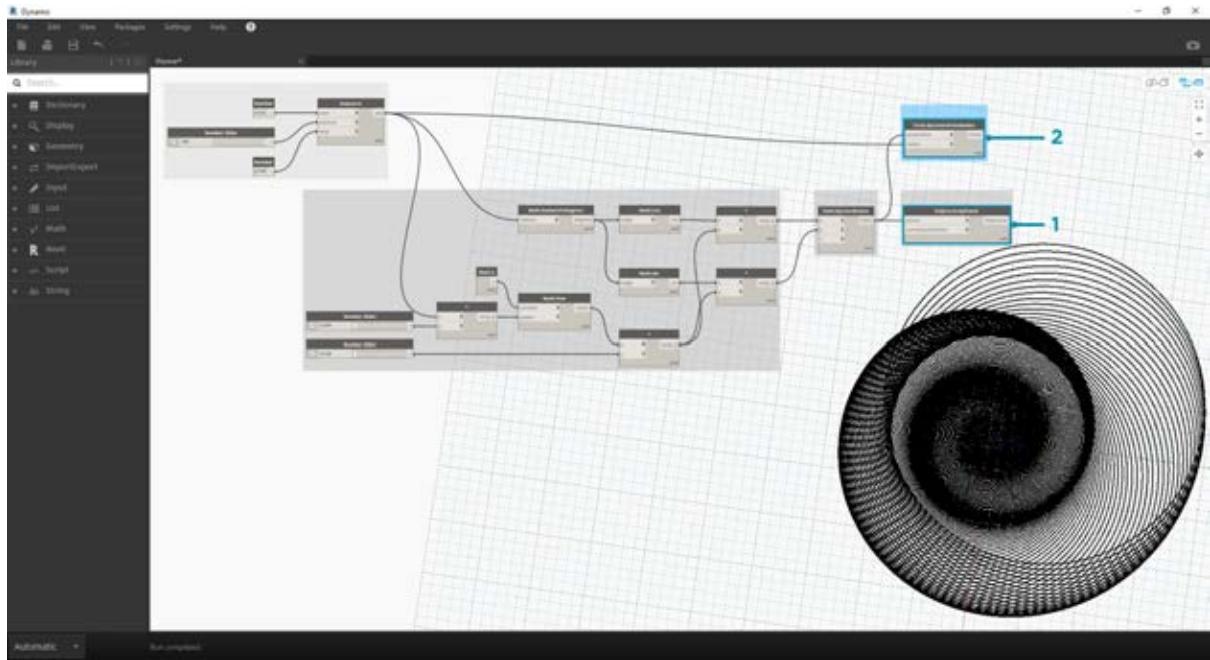
らせん構造からオウムガイへ



- 最初に、前の演習と同じ手順を実行します。**Point.ByCoordinates** ノードを使用して、点群をらせん状に並べた配列を作成します。



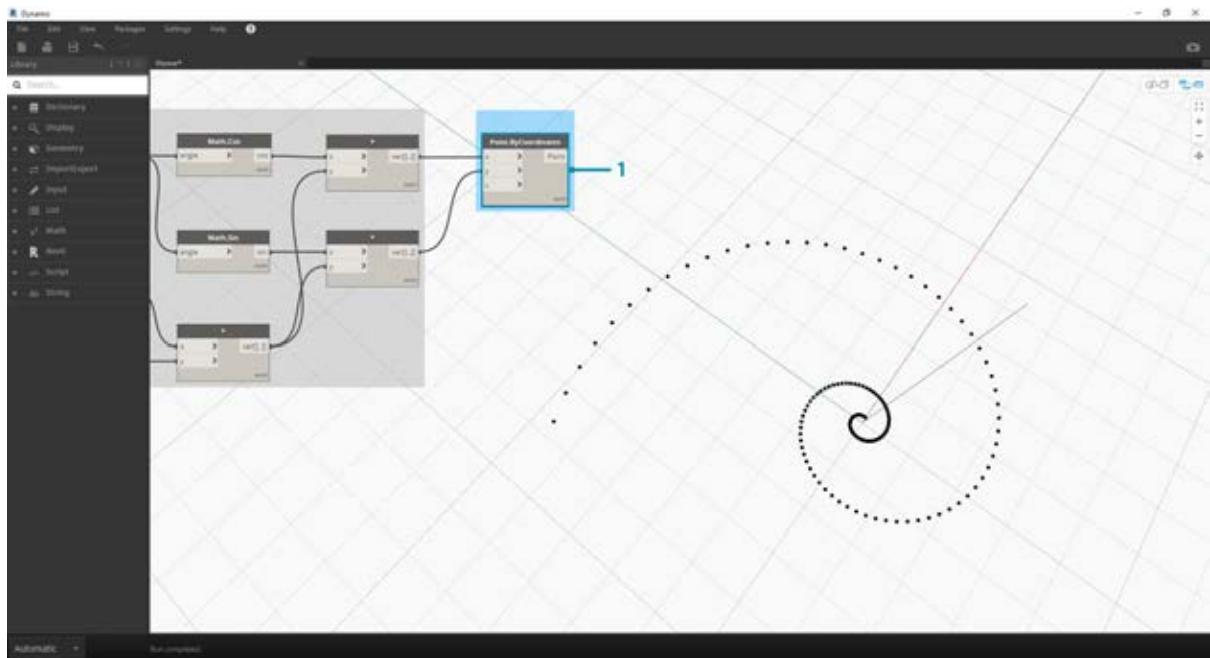
1. 前の演習で使用した **Polycurve.ByPoints** ノードを、参照用ノードとして使用します。
2. 円弧を作成するための **Circle.ByCenterPointRadius** ノードの入力ポートに、前の手順と同じ値を接続します。  
radius 入力の値は既定で 1.0 に設定されているため、出力として円弧が即座に表示されます。点群が基準点から遠ざかっていく様子がよくわかります。



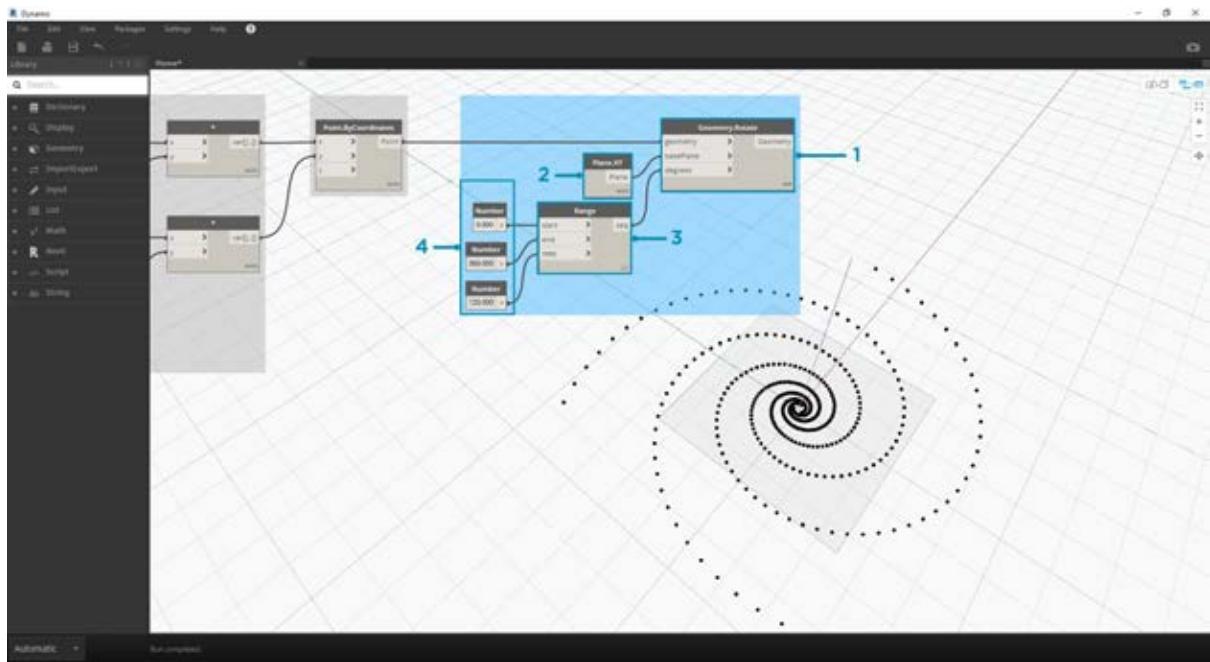
1. `Circle.ByCenterPointRadius` ノードの radius 入力に元の数値(「 $t$ 」の数列)を接続し、より動的な円弧の配列を作成します。
2. 数列のノード グループは「 $t$ 」の元の配列です。これを radius 入力に接続すると、円弧の中心は基準点から離れていくますが、円の半径は次第に長くなっていき、特長的な形状のらせん構造が作成されます。余裕があれば、これを 3D に変換してみてください。

#### オウムガイから螺旋葉序パターンへ

ここまで手順では、円弧を重ねたオウムガイのような形状を作成しました。次に、パラメータ制御のグリッドを使用してみましょう。ここでは、フィボナッチ曲線の基本的ならせんを使用して、フィボナッチ グリッドを作成していきます。結果として、[ヒマワリ](#)のようならせん形状が作成されます。

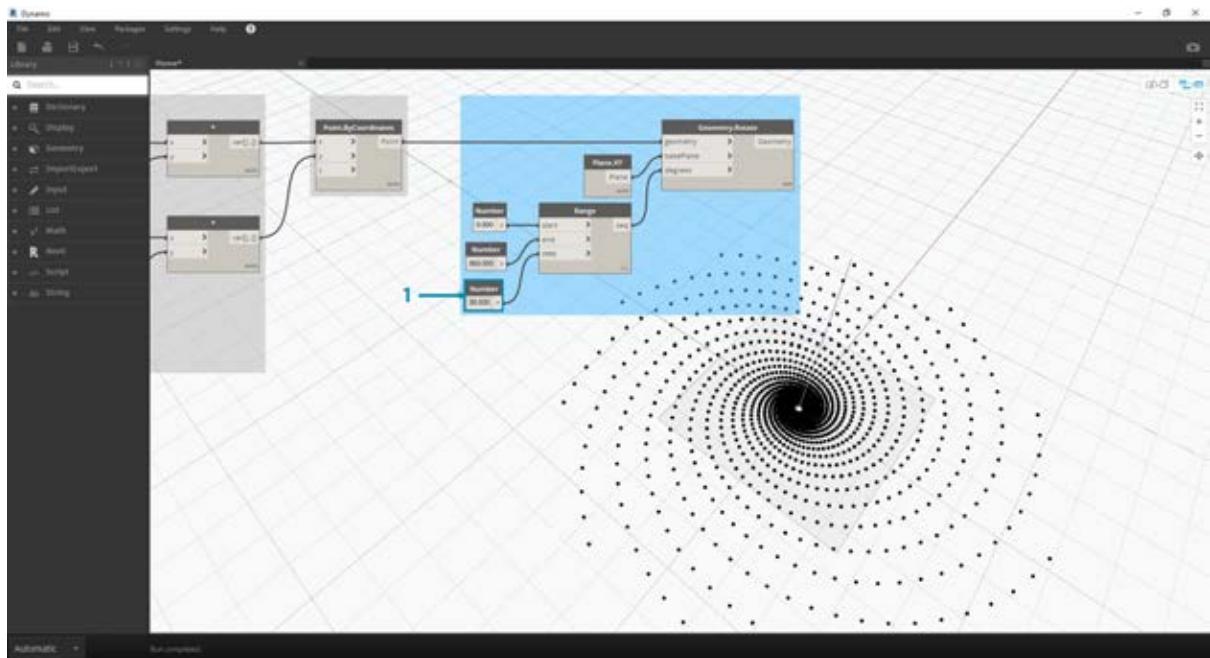


1. ここでも、前の演習と同じ手順から開始します。Point.ByCoordinates ノードを使用して、点群をらせん状に並べた配列を作成します。

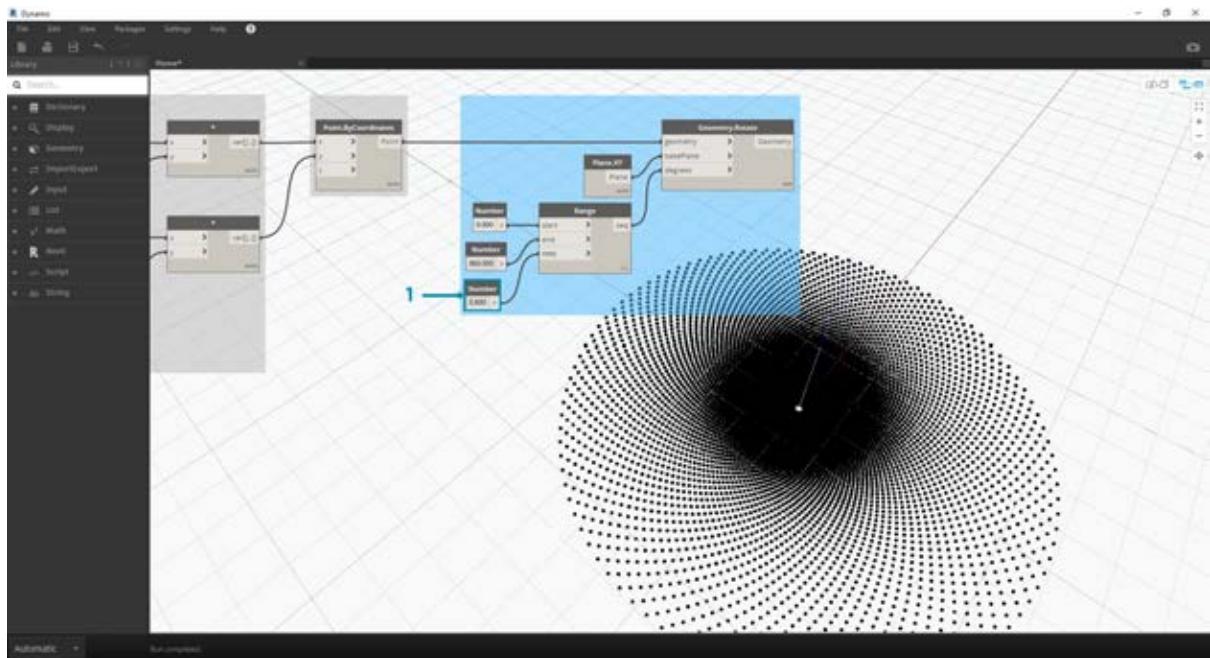


1. **Geometry.Rotate** ノードには、同じ名前のノードが複数存在します。geometry、basePlane、degrees という入力を持つ Geometry.Rotate ノードを選択してください。geometry 入力に Point.ByCoordinates ノードを接続します。
2. Plane.XY ノードを basePlane 入力に接続します。これにより、基準点に向かって回り込んでいくように点が描画されます。基準点の位置は、らせんの基準の位置と同じです。
3. Number Range ノードで角度を入力することにより、複数の巻き筋を作成します。Number Range コンポーネントを使用すると、この操作をすばやく実行することができます。このコンポーネントを degrees 入力に接続します。
4. Number ノードをキャンバスに 3 つ追加して縦に並べ、数値の範囲を定義します。上から下に、それぞれ 0.0、360.0、\*\*120.0 の値を割り当てます。これらの値により、らせんの巻き筋が制御されます。Number Range ノードに 3 つの Number ノードを接続すると、結果が出力されます。

出力結果が渦の流れのようになってきました。Number Range パラメータの一部を調整して、出力結果がどのように変わるか確認してみましょう。



1. Number Range ノードの step の入力値を 120.0 から 36.0 に変更します。この操作により、巻き筋の数が増え、グリッドの密度が上がります



1. Range ノードの step 入力の値を 36.0 から 3.6 に変更します。この操作により、グリッドの密度が大幅に上がり、螺旋の方向が不明瞭になります。これで、ヒマワリが完成しました。

# ロジック

## ロジック

ロジック(正確には条件付きロジック)により、テストに基づく単一のアクションや一連のアクションを指定することができます。テストの評価を行うと、True または False のブール値が返されます。このブール値を使用して、プログラム フローをコントロールすることができます。

### ブール値

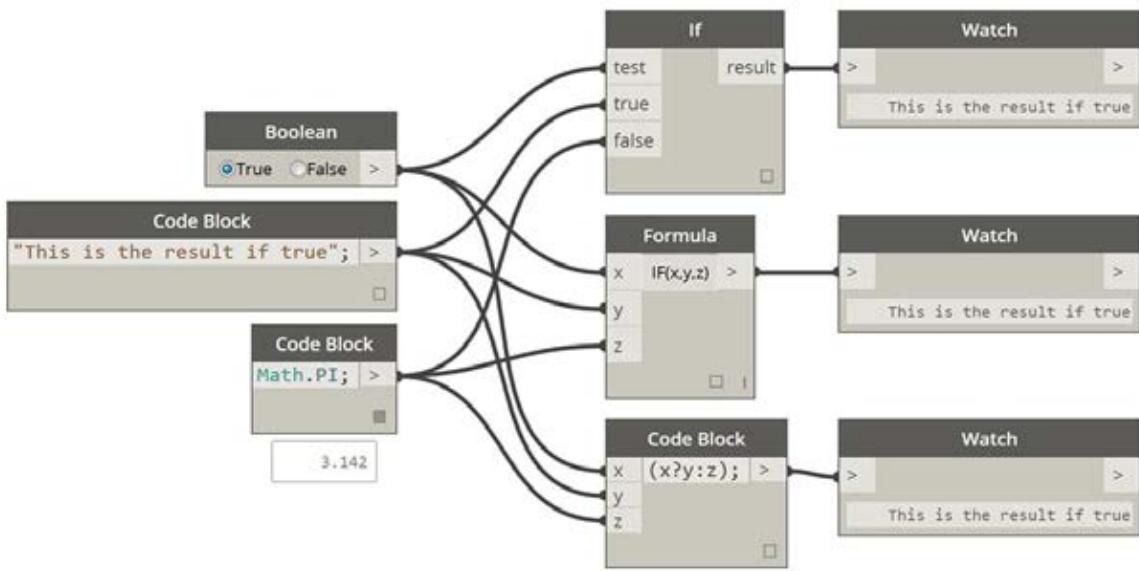
数値変数には、さまざまな数値の範囲全体を格納することができます。ブール変数には、True または False、Yes または No、1 または 0 など、2 つの値のみを格納することができます。このようにブール値の範囲は限られているため、ブール値を使用して計算を行うことはほとんどありません。

### 条件ステートメント

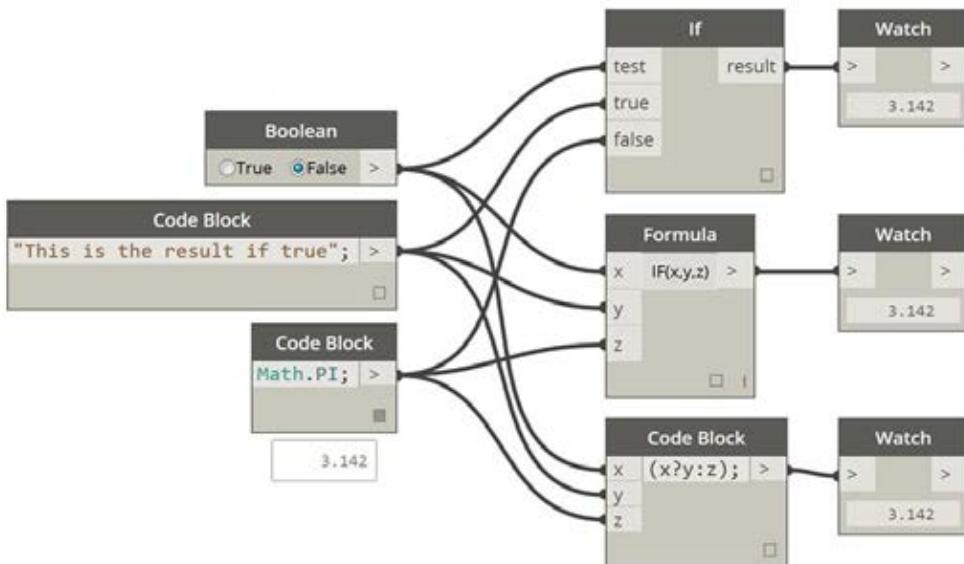
If ステートメントは、プログラミングにおける重要な概念です。このステートメントは、「この条件が真である場合はこの処理を実行し、偽である場合は別の処理を実行する」という形式で記述されます。このステートメントの結果として出力されるアクションは、ブール値によって制御されます。Dynamo で If ステートメントを定義する場合、いくつかの方法があります。

アイコン	名前	構文	入力	出力
	If	If	test, true, false	result
	Formula	IF(x,y,z)	x, y, z	result
	Code Block (x?y:z)		x, y, z	result

ここでは簡単な例を使用して、If 条件ステートメントを使用する 3 つのノードの動作を確認していきます。



上の図では、Boolean ノードの出力値が *True* に設定されているため、その結果として "this is the result if true" という文が表示されます。この場合、If ステートメントを構成する 3 つのノードの動作は同じになります。

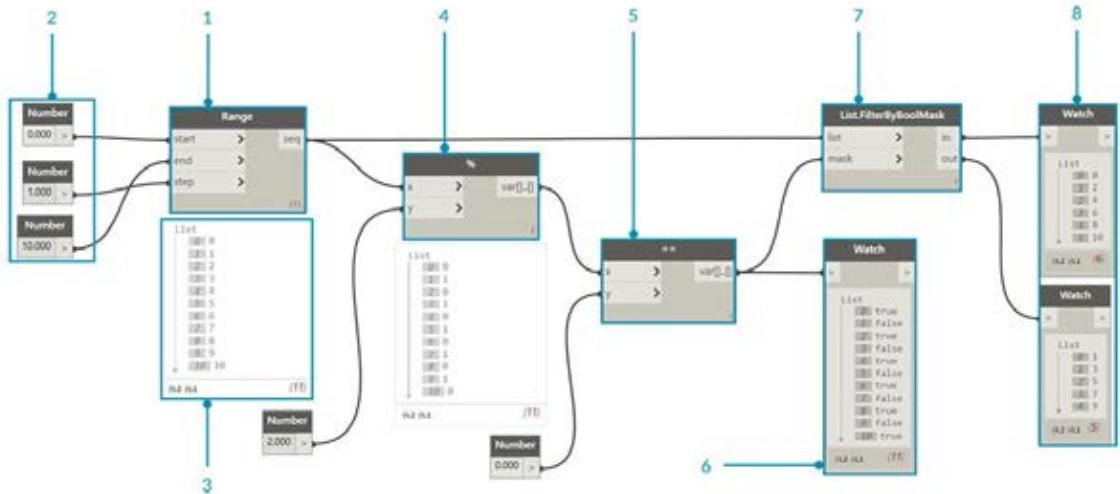


繰り返しますが、これら 3 つのノードの動作はすべて同じになります。布尔値を *False* に変更すると、元の If ステートメントで定義されているとおりに、出力結果の値が *Pi* になります。

### リストをフィルタする

この演習に付属しているサンプル ファイルをダウンロードしてください(右クリックして[名前を付けてリンク先を保存...]を選択): [Building Blocks of Programs - Logic.dyn](#)。すべてのサンプルファイルの一覧については、付録を参照してください。

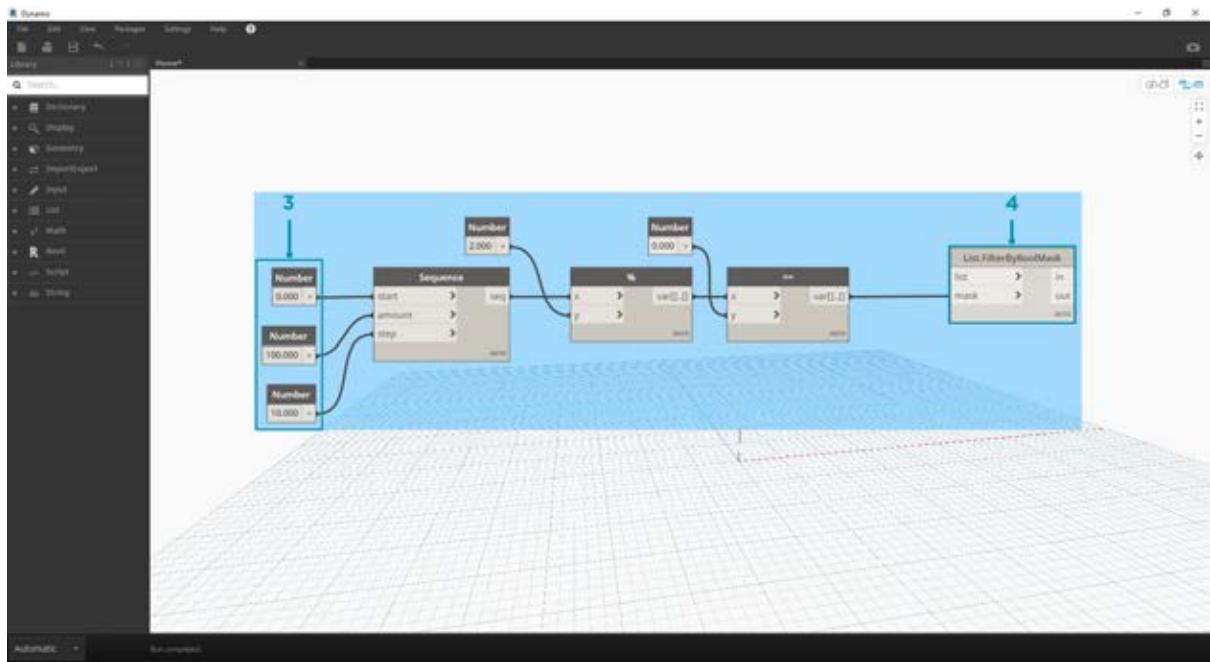
ここでは、ロジックを使用して、数値のリストを偶数のリストと奇数のリストに分割してみましょう。



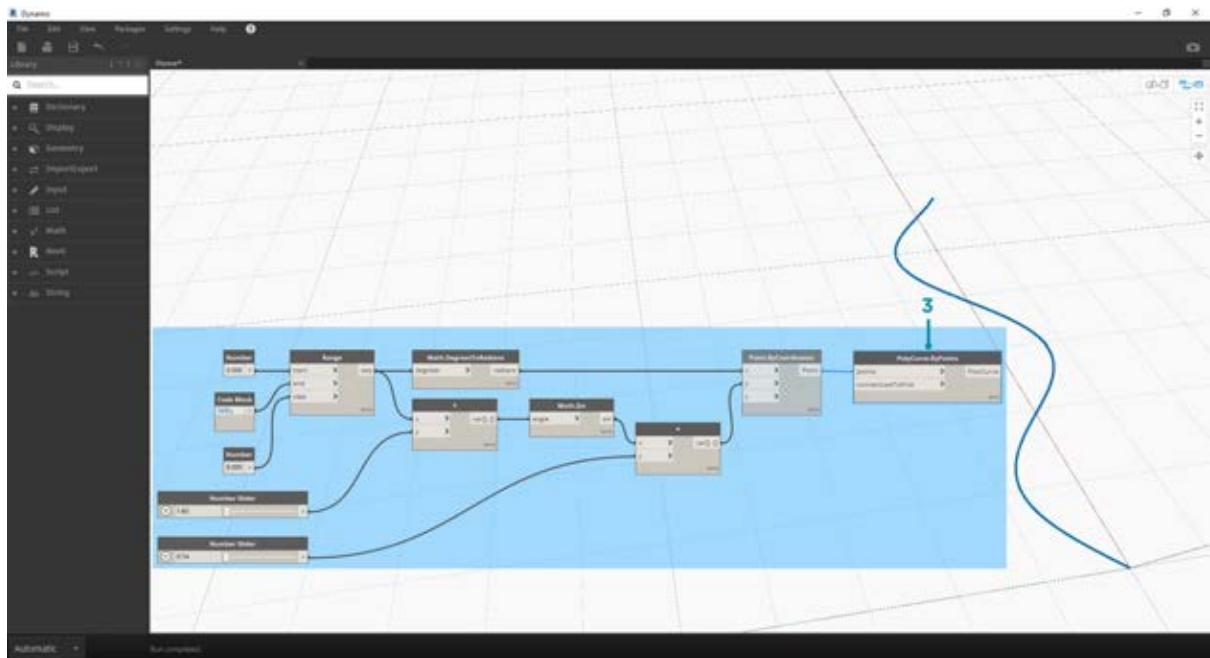
1. Number Range ノードを使用して、数値の範囲をキャンバスに追加します。
2. Number ノードを使用して、3 つの Number ノードをキャンバスに追加します。各 Number ノードで、次のように値を指定します。start 入力: 0.0, end 入力: 10.0, step 入力: 1.0。
3. 出力結果として、0 から 10 までの範囲にわたる 11 個の数値のリストが生成されます。
4. 「%(Modulo)」ノード(モジュロ演算ノード)の x 入力に Number Range ノードを接続し、y 入力で 2.0 を指定します。この操作により、リスト内の各数値を 2 で除算した場合の余りが算出されます。このリストの出力値は、0 と 1 が交互に現れる数値のリストになります。
5. 「(==)」ノード(等価テストノード)を使用して、キャンバスに等価テストを追加します。「(%)」ノードの出力を「(==)」ノードの x 入力に接続し、Number ノードの 0.000 出力を「(==)」ノードの y 入力に接続します。
6. Watch ノードを使用して、等価テストの出力が true と false の値を交互に繰り返すリストになっていることを確認します。これらの値を使用して、リスト内の項目が区別されます。0 (または true) は偶数を表し、1 (または false) は奇数を表します。
7. List.FilterByBoolMask ノードは、ブール値の入力に基づいて数値をフィルタし、2 つの異なるリストに分割します。元の数値の範囲を list 入力に接続し、「(==)」ノードの出力を mask 入力に接続します。in 出力は true の値を表し、out 出力は false の値を表します。
8. Watch ノードを使用して、偶数のリストと奇数のリストが生成されたことを確認します。これで、論理演算子を使用して、リストがパターン別に分類されました。

## ロジックからジオメトリへ

ここでは、最初の演習で作成したロジックを変更してモデリング操作に適用してみましょう。



1. ここでも、前の演習と同じノードを使用します。ただし、次のように、いくつか違いがあります。
2. ノードの形式が変更されています。
3. 入力値が変更されています。
4. *List.FilterByBoolMask* ノードの *list* 入力に対する接続が解除されています。これらのノードは、この演習の後半で使用します。



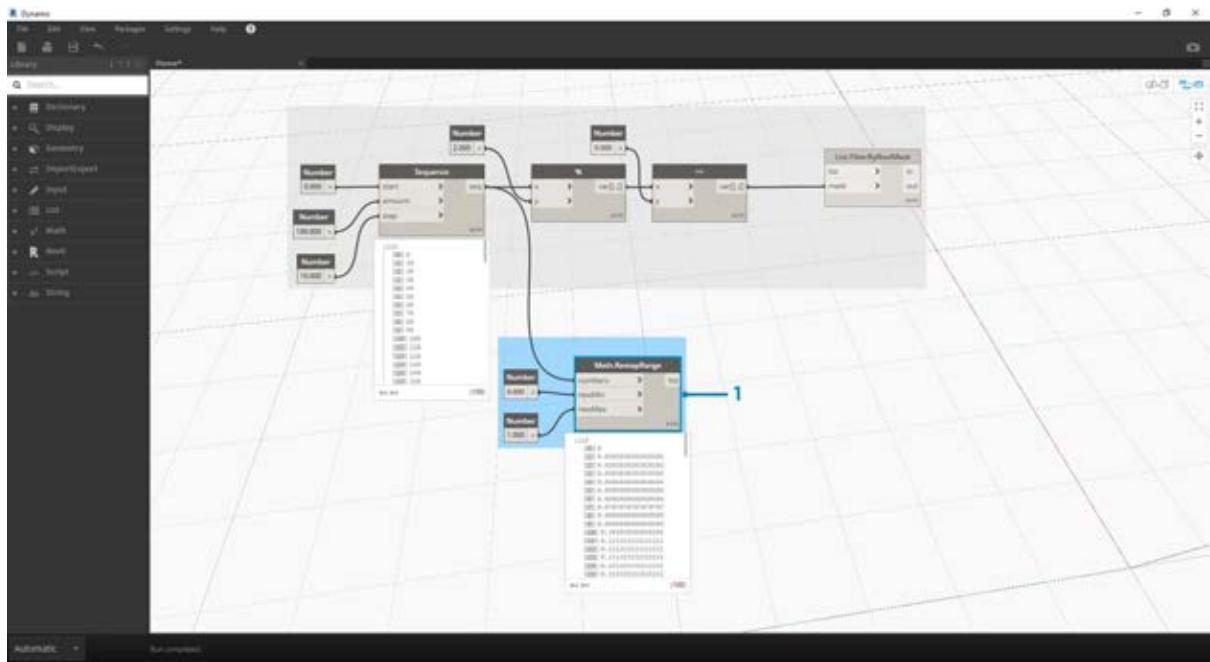
最初に、上の図のように各ノードを接続します。このノード グループは、曲線を定義するためのパラメータ制御式を表しています。ここで、次の点に注意する必要があります。

1. 1 番目の Number Slider ノードでは、最小値 1、最大値 4、ステップ値 0.01 に設定する必要があります。
2. 2 番目の Number Slider ノードでは、最小値 0、最大値 1、ステップ値 0.01 に設定する必要があります。
3. PolyCurve.ByPoints ノードにより、上図のとおりにノード ダイアグラムを構成した時点で、Dynamo のプレビューに正弦曲線が output されます。

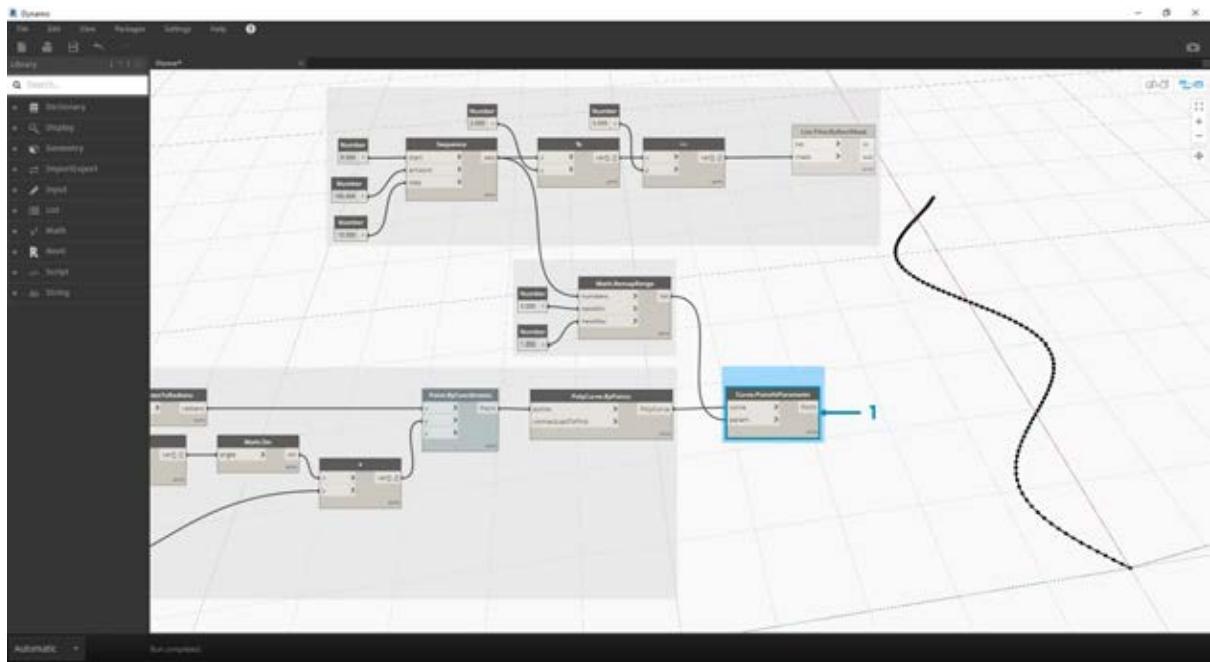
ここでは、静的なプロパティで Number ノードを使用し、動的なプロパティで Number Slider ノードを使用して入力を行います。この手順の最初で定義した元の数値の範囲をそのまま使用してもかまいませんが、ここで作成する正弦曲線に対して、ある程度の柔軟性を設定しておく必要があります。Number Slider ノードの値を変更して、曲線の周波数と振幅がどのように変化するかを確認してください。



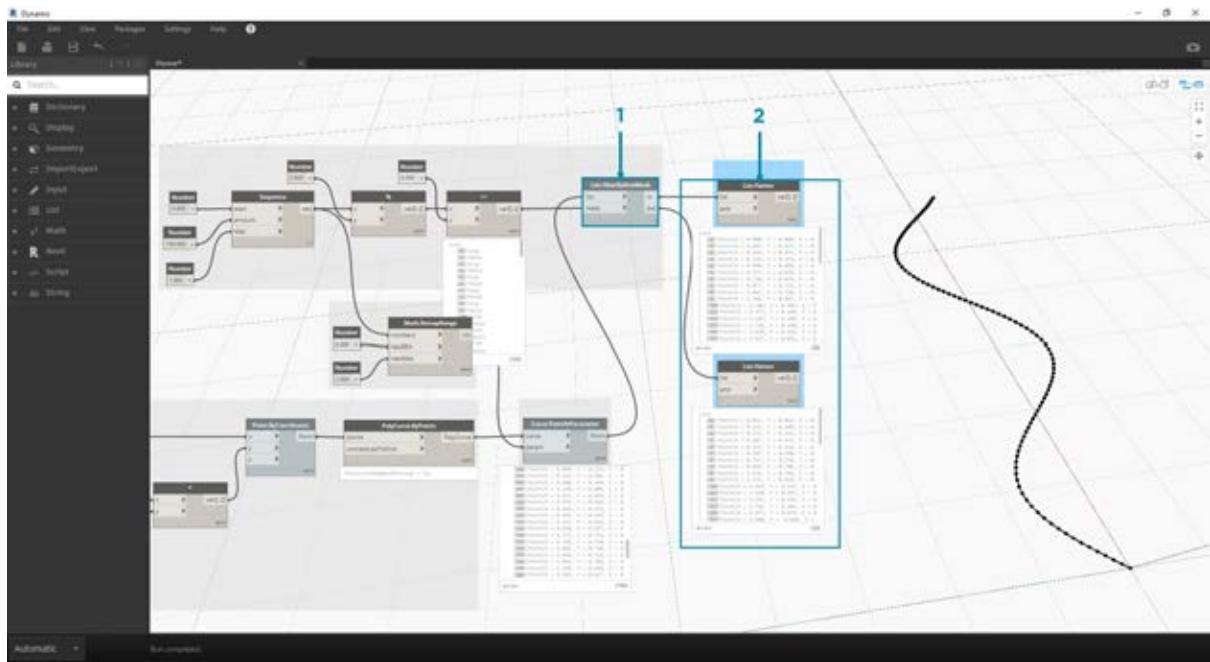
ここで少し先回りをして、最終的な結果を確認しましょう。個別に作成した最初の 2 つのステップを接続する必要があります。基本の正弦曲線を使用して、ジッパー状のコンポーネントの位置をコントロールし、真偽判定のロジックを使用して、小さなボックスと大きなボックスを交互に配置します。



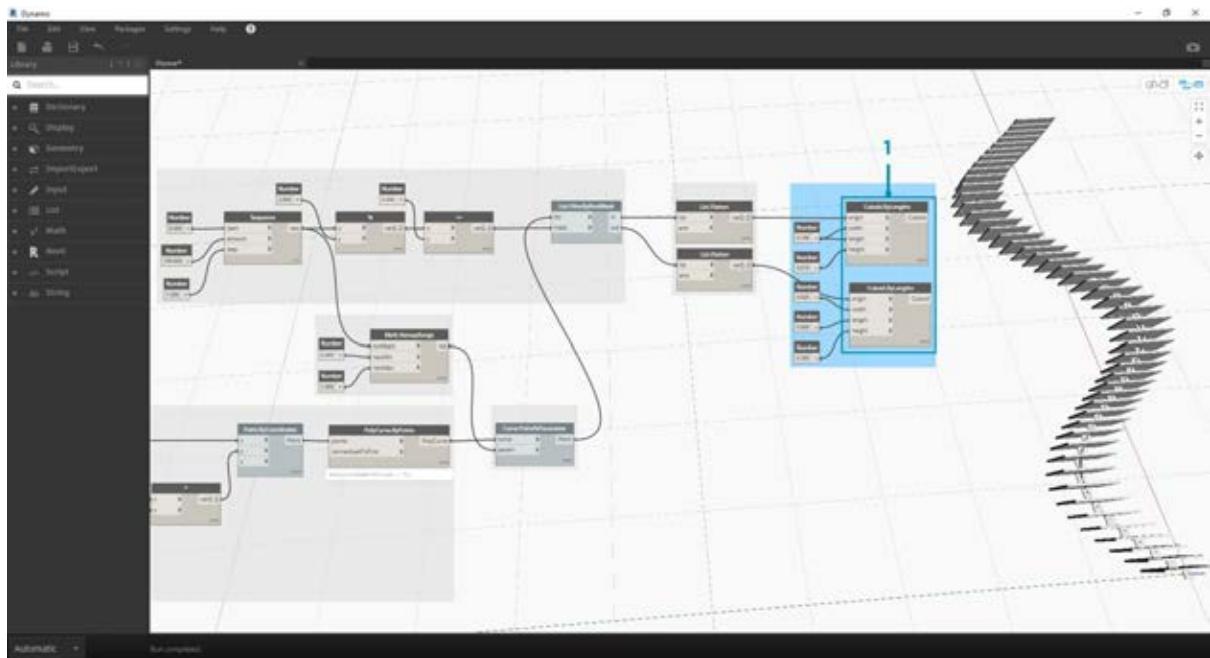
- Sequence ノードの step 入力の値 1 で作成された数列の範囲を Math.RemapRange ノードで再マッピングして、新しい一連の数値を作成します。元の数値の範囲は、0 から 100 までになります(step 値 1)。新しい数値の範囲は、0 から 1 になります。最小値の 0 を newMin 入力に接続し、最大値の 1 を newMax 入力に接続します。



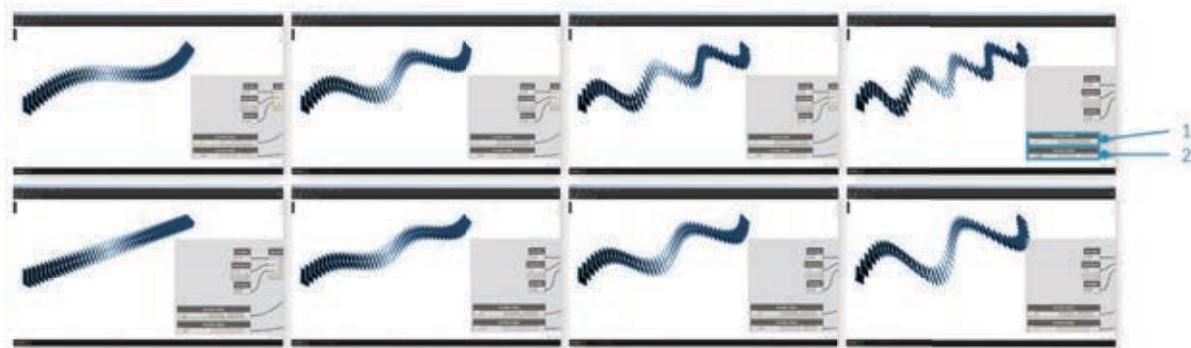
1. **Curve.PointAtParameter** ノードの *curve* 入力に *Polycurve.ByPoints* ノード(セクション 4.2 を参照)を接続し、*param* 入力に *Math.RemapRange* ノードを接続します。この操作により、曲線に沿って点群が作成されます。ここでは数値の範囲を 0 から 1 までの範囲に再マッピングしましたが、その理由は、*param* 入力に 0 から 1 までの範囲の値を指定する必要があるためです。*0* の値は始点を表し、*1* の値は終点を表します。すべての中間値は、 $[0,1]$  の範囲内で評価されます。



1. *List.FilterByBoolMask* ノードの *list* 入力に、前の手順で使用した *Curve.PointAtParameter* ノードを接続します。
2. *Watch* ノードを *in* 出力と *out* 出力に 1 つずつ接続し、偶数値のインデックスを表すリストと奇数値のインデックスを表すリストが生成されたことを確認します。これらの点群は、曲線上に同じ方法で配置されます。これについては、次の手順で確認します。



1. **Cuboid.ByLengths** ノードを上の図のように接続し、正弦曲線に沿ったジッパー構造を作成します。ノード名の「Cuboid」とは、直方体という意味です。ここでは、直方体の中央に位置する曲線上の点を基準として、直方体のサイズを定義します。この操作により、偶数と奇数の除算ロジックがモデル内に明示的に定義されます。



1. Number Slider ノードの数値スライダを使用して最初の定義からやり直すと、ジッパー構造がどのように変化するかを確認することができます。上の図の上列の画像は、上部の Number Slider ノードの値を調整した場合の変化を示しています。これが、波の周波数になります。
2. 下列の画像は、下部の Number Slider ノードの値を調整した場合の変化を示しています。これが、波の振幅になります。

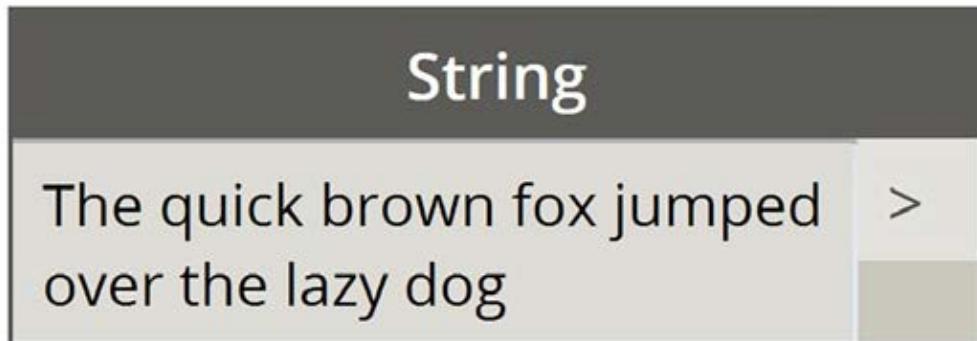
# 文字列

## 文字列

文字列とは、正確には、リテラル定数や何らかのタイプの変数を表す一連の文字の集合のことです。広い意味では、文字列とは、テキストを意味するプログラミング用語です。ここまで手順では、整数と小数の両方を使用してパラメータをコントロールしていましたが、テキストについても同じ処理を行うことができます。

### 文字列を作成する

文字列は、カスタム パラメータの定義、ドキュメント セットの注釈付け、テキストベースのデータ セットの解析など、さまざまな目的で使用することができます。String ノードは、[Core] > [Input] カテゴリで使用することができます。



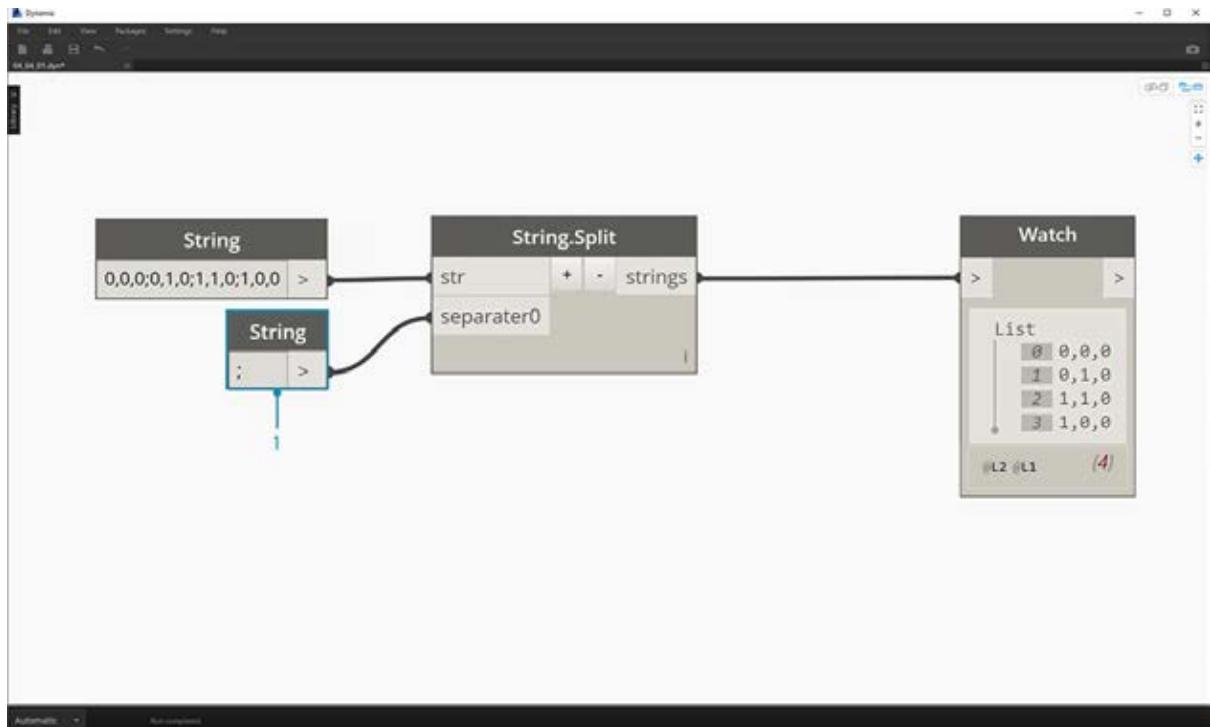
上記のサンプル ノードは、すべて文字列用のノードです。数値は文字と同様に、文字列として表すことも、テキスト配列全体として表すこともできます。

### 文字列のクエリーを実行する

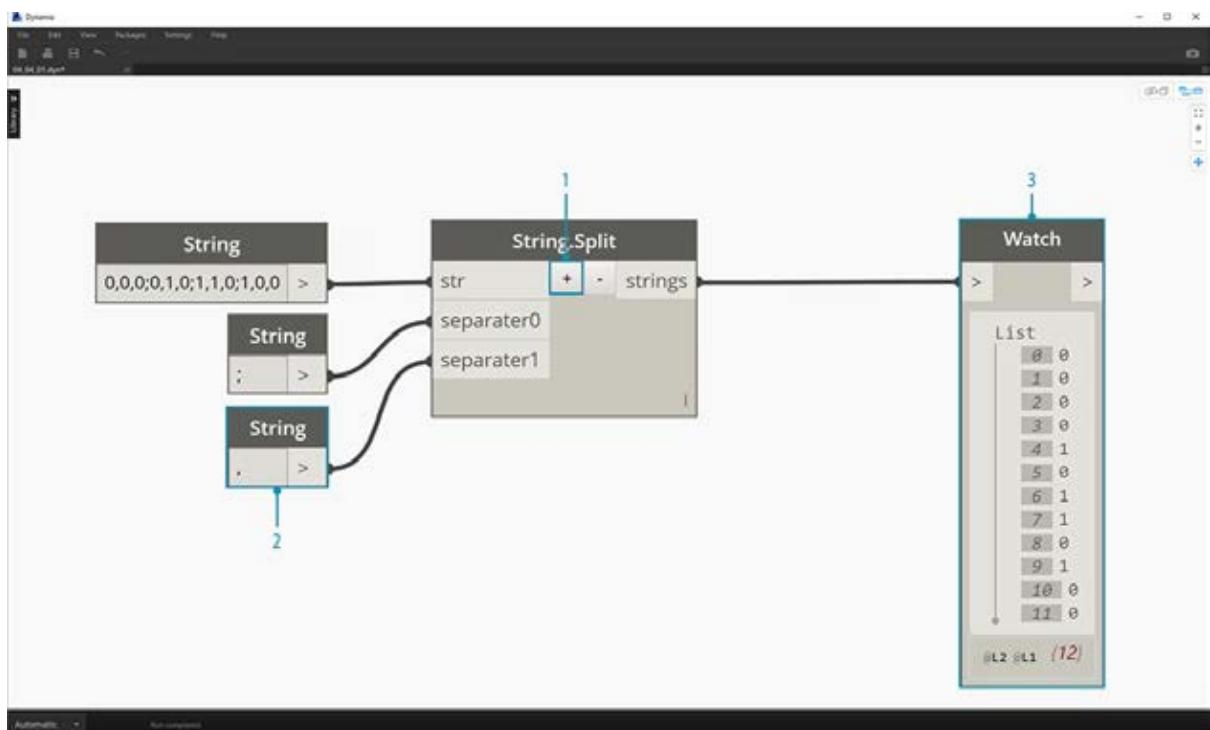
この演習に付属しているサンプル ファイルをダウンロードしてください(右クリックして[名前を付けてリンク先を保存...]を選択): [Building Blocks of Programs - Strings.dyn](#)。すべてのサンプルファイルの一覧については、付録を参照してください。

文字列のクエリーを実行することにより、大量のデータをすばやく解析することができます。ここでは、ワークフローを促進してソフトウェアの相互運用性を向上させるための基本的な操作について説明します。

次の図は、外部のスプレッドシートから取得したデータの文字列を示しています。この文字列は、XY 面上の長方形の頂点を表しています。簡単な演習として、文字列の分割操作を確認してみましょう。

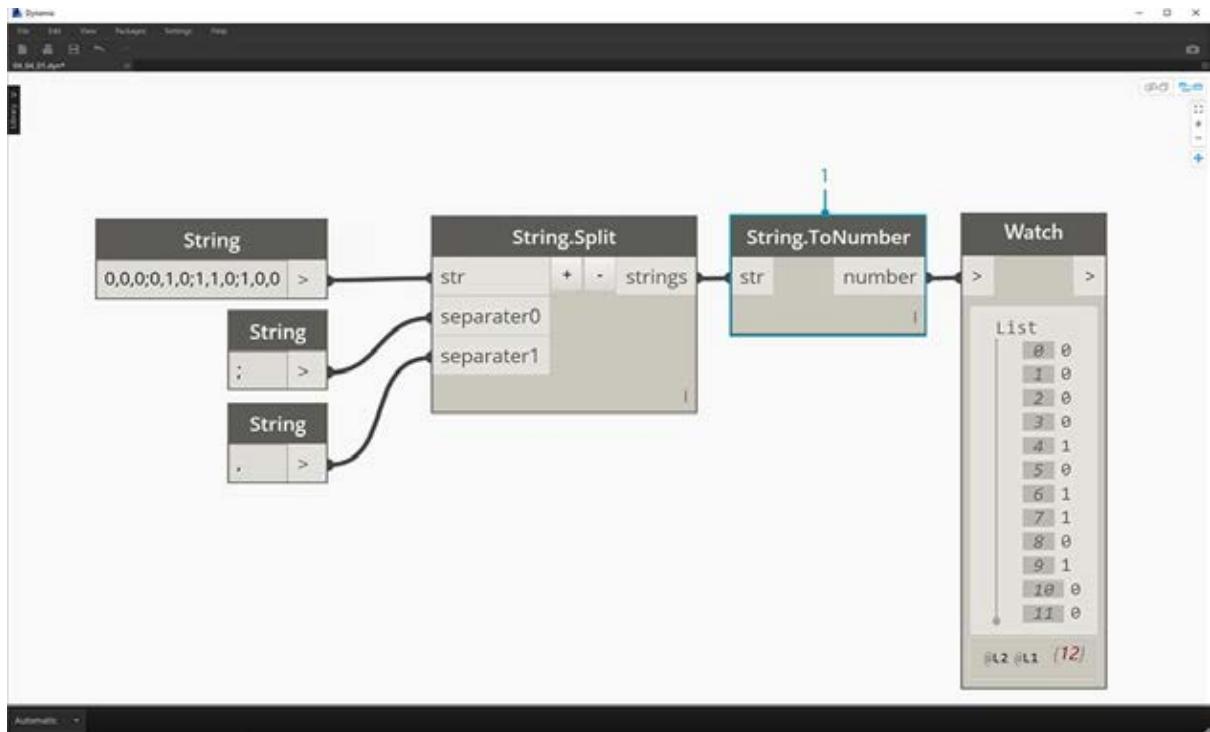


- 「;」セパレータは、長方形の各頂点を分割します。これにより、各頂点の 4 つの項目が含まれているリストが作成されます。

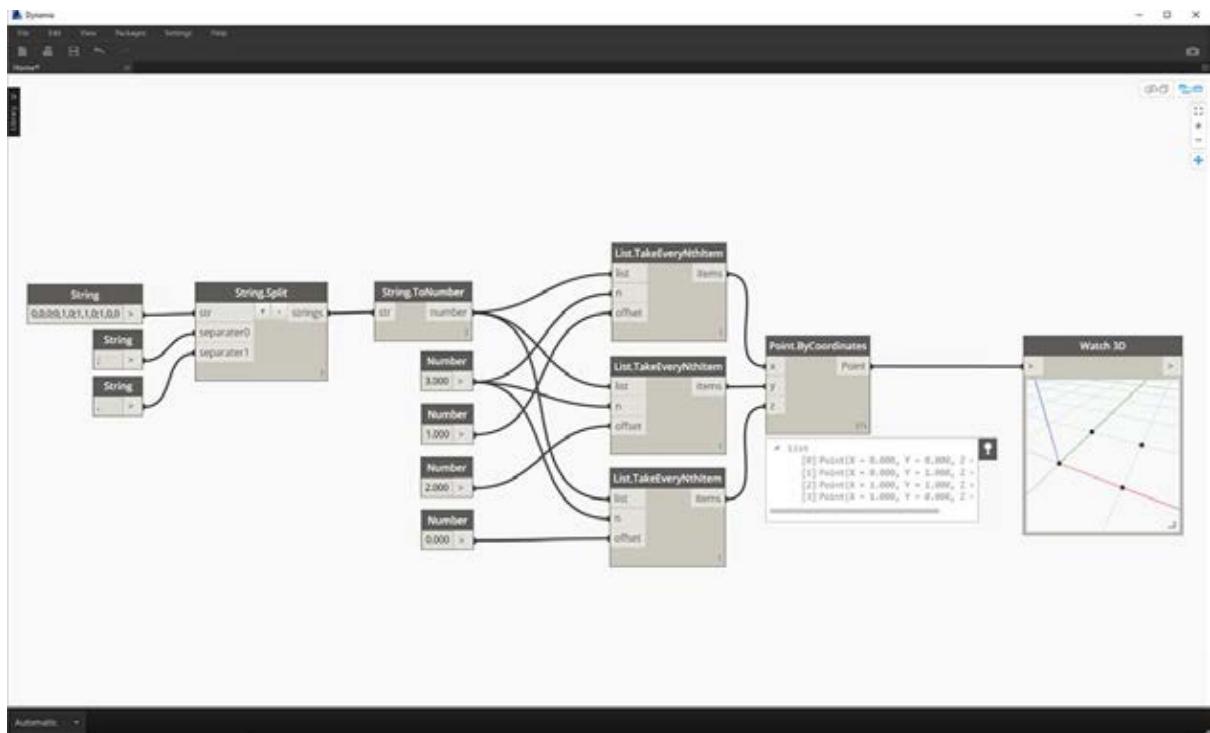


1. String.Split ノードの中央にある[+]をクリックして、新しいセパレータを作成します。
2. 「;」という文字をキャンバスに追加し、separator1 入力に接続します。
3. リストの項目が 10 個になります。String.Split ノードは、separator0 に基づいて分割を行ってから、separator1 に基づいて分割を行います。

上記のリストの項目は数値に見えますが、Dynamo 内では個別の文字列として認識されます。点群を作成するには、データ タイプを文字列から数値に変換する必要があります。これを行うには String.ToNumber ノードを使用します。



1. String.ToNumber ノードは、単純なノードです。String.Split ノードの出力を String.ToNumber ノードの入力に接続します。出力の表示は変わりませんが、データタイプは文字列から数値に変換されています。

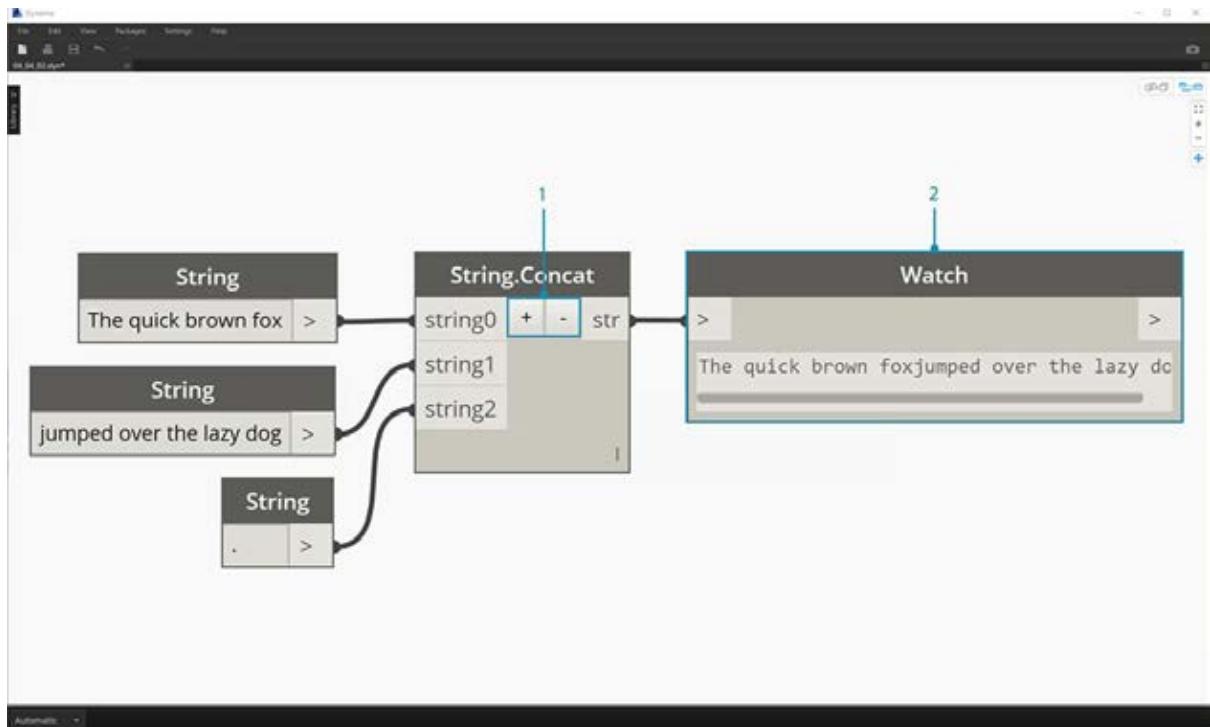


1. いくつかの基本的な操作を実行すると、元の文字列の入力に基づいて基準点に長方形が描画されます。

## 文字列を操作する

文字列は一般的なテキストオブジェクトであるため、さまざまな目的で使用することができます。ここでは、[Core] > [String]カテゴリの主要なアクションをいくつか見てみましょう。

このカテゴリには、複数の文字列を順に結合する方法が用意されています。リストから各リテラル文字列を取得して、1つの文字列に結合します。

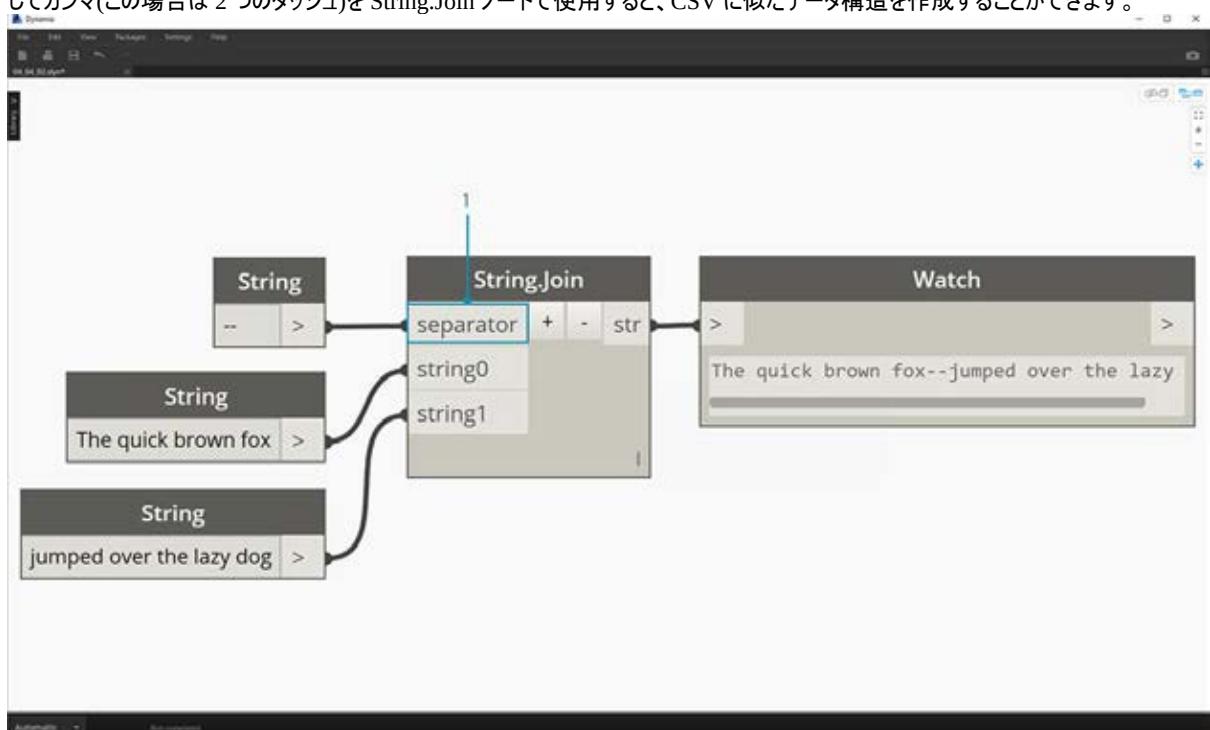


上の図は、3つの文字列を連結する方法を示しています。

1. **String.Concat** ノード中央の[+]/[-]ボタンをクリックして、連結する文字列の追加と除外を行います。
2. 出力として、スペースと句読点を含む1つの連結された文字列が生成されます。

結合操作は連結操作に非常によく似ていますが、句読点のレイヤーが追加される点が異なります。

Excelを使用したことがありますれば、CSVファイルについても知っているでしょう。CSVとは、カンマ区切り値という意味です。セパレータとしてカンマ(この場合は2つのダッシュ)を **String.Join** ノードで使用すると、CSVに似たデータ構造を作成することができます。

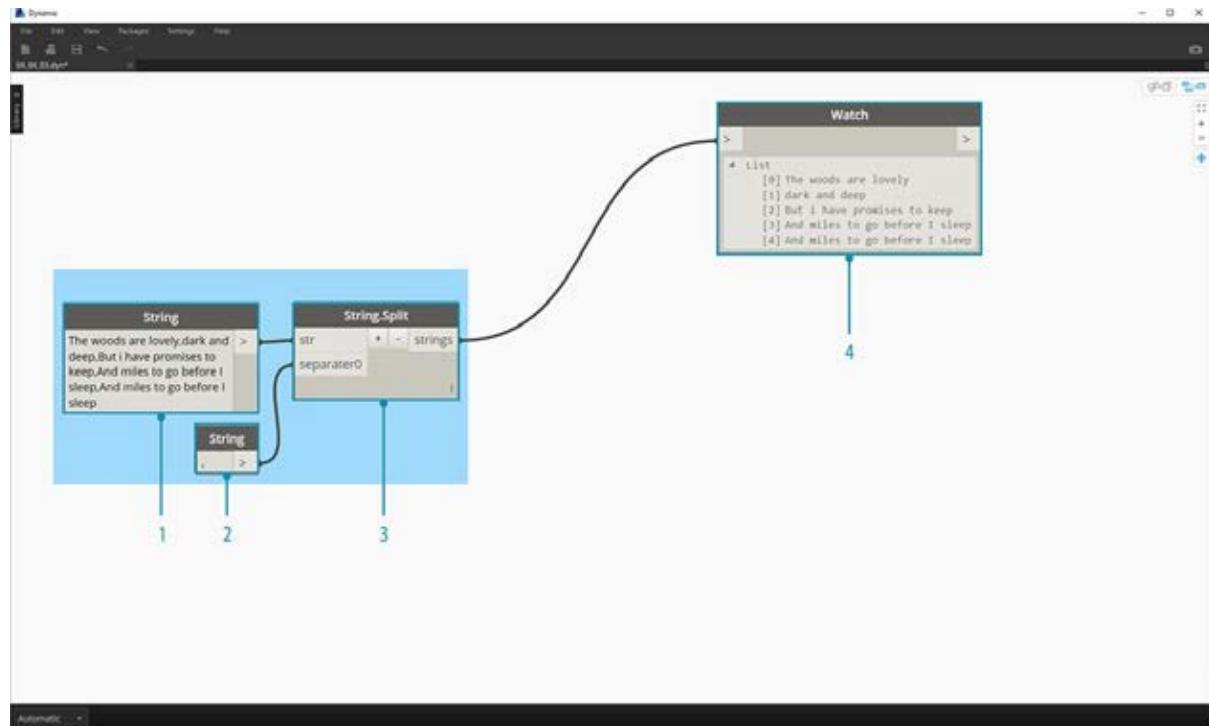


上の図は、2つの文字列を結合する方法を示しています。

1. **separator** 入力を使用して、結合された文字列の区切り文字を指定することができます。

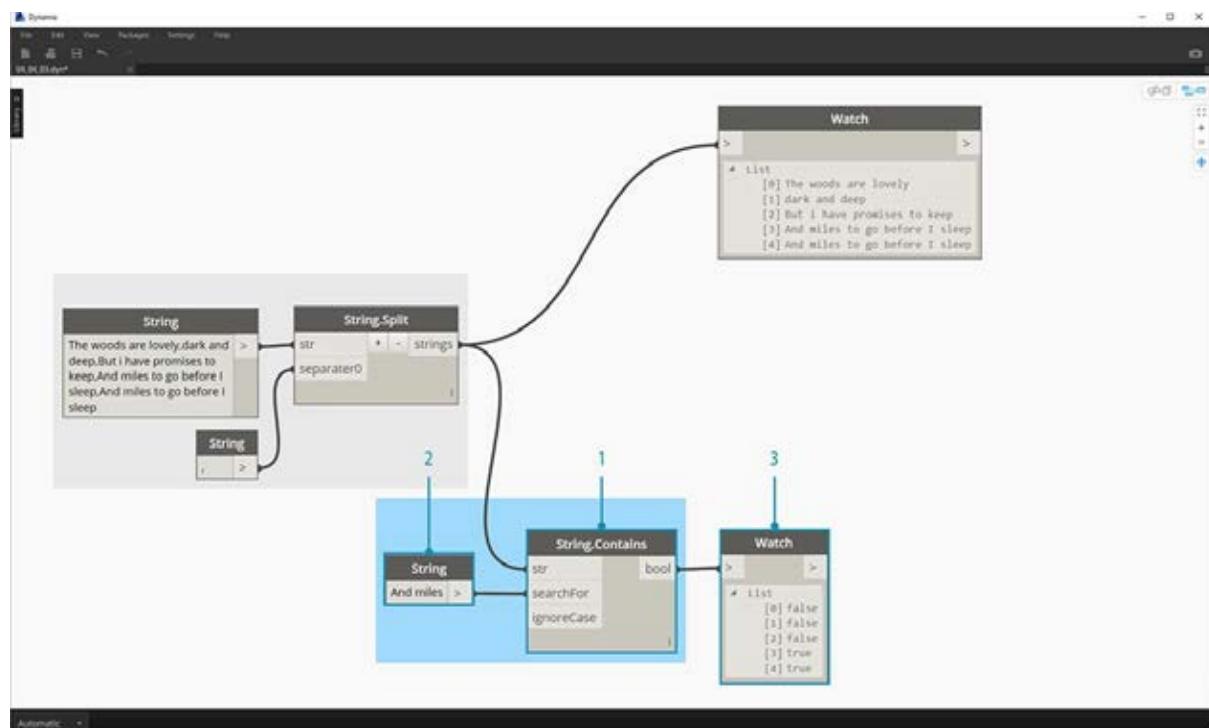
## 文字列を操作する

この演習では、文字列のクエリーと文字列の操作を実行して、Robert Frost の「[Stopping By Woods on a Snowy Evening](#)」という詩の最後の節を分解してみましょう。実際の開発作業で詩の文字列を操作することはできませんが、文字列に対する操作を理解するのに役立ちます。



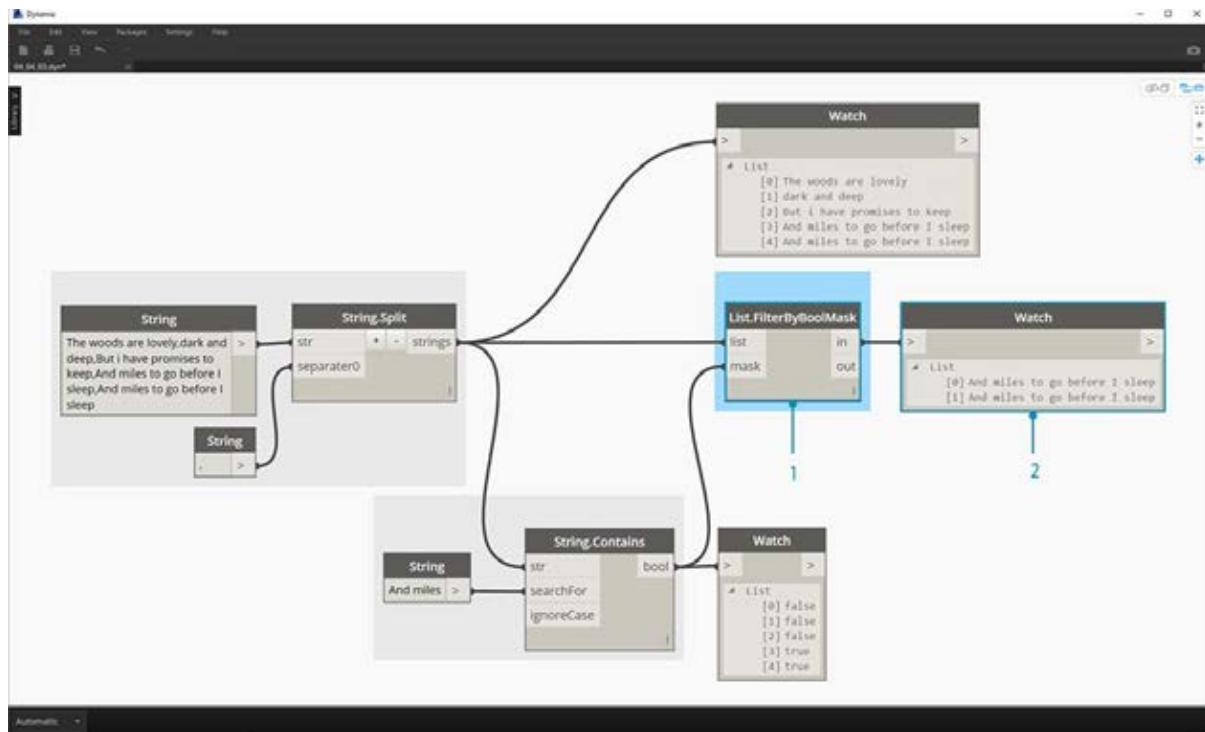
最初に、節の文字列を分解する基本的な操作を実行します。文字はカンマに基づいて書式設定されていることがわかります。この書式を使用して、各行を個々の項目に分割します。

1. ベースとなる文字列を String ノードに貼り付けます。
2. 別の String ノードを使用して、セパレータを指定します。ここでは、カンマをセパレータとして使用します。
3. String.Split ノードをキャンバスに追加し、2 つの String ノードに接続します。
4. 出力として、各行が個別の要素に分割されます。

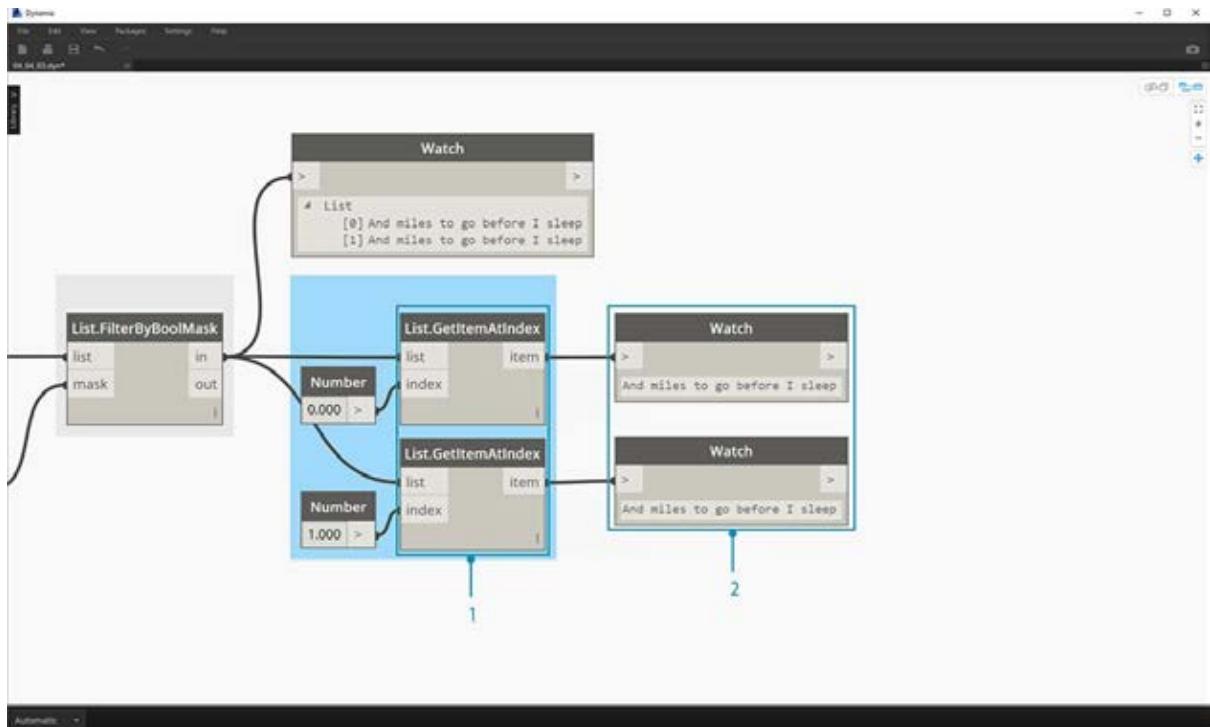


次に、詩のサビとなる最後の 2 行を処理しましょう。元の節は 1 つのデータ項目でした。最初の手順で、このデータを個別の項目に分割しました。そのため、必要な文字列を検索する必要があります。この例では、リストの最後にある 2 つの項目を選択することは難しくありませんが\*\*、書籍全体が処理対象になるような場合、内容をすべて読んでから各要素を手作業で区別するのは大変な作業になります。

1. String.Contains ノードを使用して文字セットを検索すれば、こうした手作業を行う必要はありません。これは、文書ファイルで「検索」コマンドを実行するのと似ています。この場合、対象のサブストリングが項目内で見つかったかどうかに応じて、「true」または「false」が返されます。
2. String.Contains ノードの searchFor 入力で、節内で検索するサブストリングを定義します。ここでは、String ノードで「And miles」と指定します。
3. 出力として、false と true のリストが表示されます。次の手順では、このブール値ロジックを使用して各要素をフィルタします。

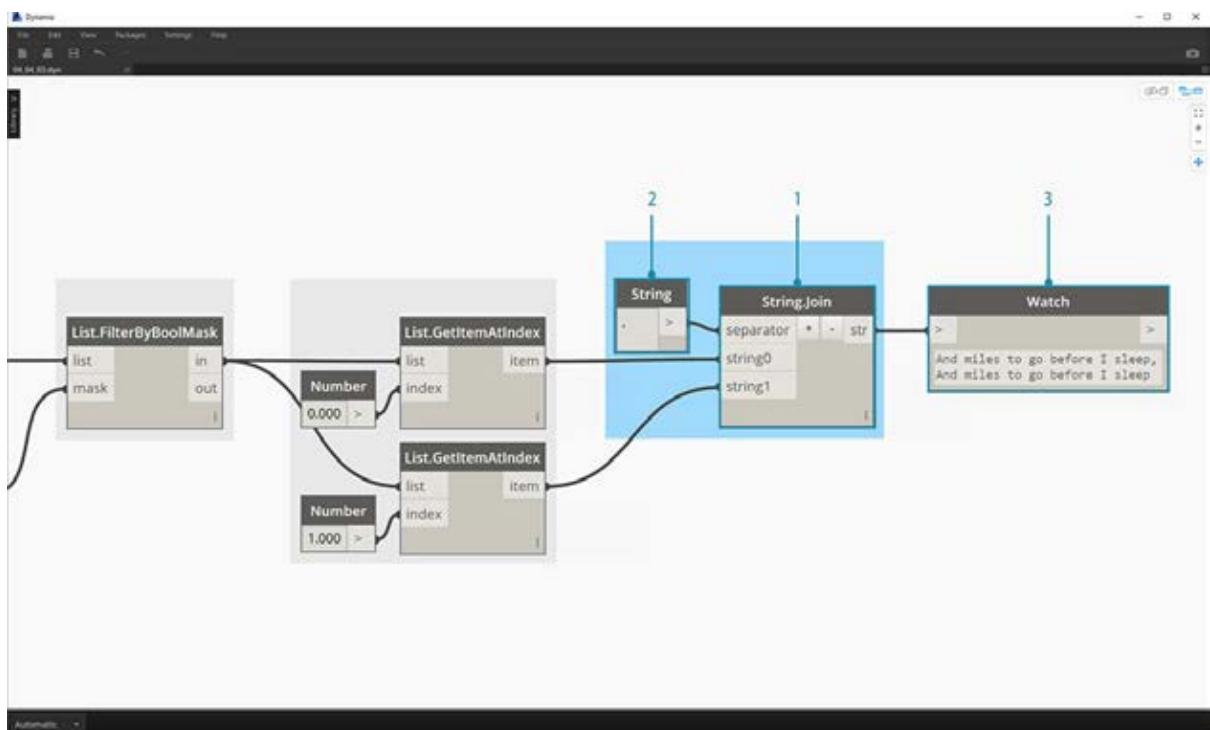


1. List.FilterByBoolMask ノードを使用して false と true を抽出します。in 出力は、mask 入力が true になっているステートメントを返し、out 出力は、mask 入力が false になっているステートメントを返します。
2. in 出力により、節の最後の 2 行が返されます。この 2 行には、「And miles」という文字列が含まれています。



次に、最後の 2 行を結合して節の繰り返しを強調します。前の手順の出力を確認すると、リスト内に 2 つの項目が含まれていることがわかります。

1. 2 つの List.GetItemAtIndex ノードを使用し、0 と 1 を index 入力の値として指定することにより、2 つの項目を分離することができます。
2. 各ノードの出力として、最後の 2 行が順に表示されます。



これらの 2 つの項目を 1 つに結合するには、String.Join ノードを使用します。

1. String.Join ノードを追加したら、セパレータを指定する必要があります。
2. セパレータを指定するには、キャンバスに String ノードを追加してカンマを入力します。
3. 最終的な出力として、最後の 2 行が 1 行に結合されます。

このように、最後の 2 行を分割する作業は手間がかかります。文字列の操作では、多くの場合、事前の作業が必要になります。ただし、文字列の操作は拡張可能であるため、大規模なデータセットにも比較的簡単に適用することができます。スプレッドシートでパラメータを使用して対話式に操作する場合は、文字列の操作を実行すると便利です。

# 色

## 色

色は、効果的なビジュアルを作成するためだけではなく、ビジュアル プログラムの出力で差異をレンダリングするために重要なデータ タイプです。抽象的なデータや変化する数値を操作する場合、何がどの程度変化するのかを確認するのが難しいことがあります。Dynamo は、色の処理に優れたアプリケーションです。

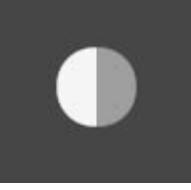
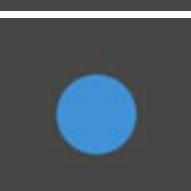
### 色を作成する

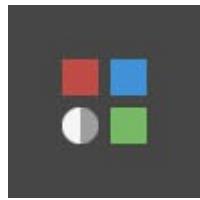
Dynamo では、ARGB 入力を使用して色を作成します。これは、アルファ、赤、緑、青の各チャネルに対応しています。アルファは色の透明度を表し、他の 3 つのチャネルは、色のスペクトラル全体を生成するための原色として組み合わせて使用されます。

アイコン	名前	構文	入力	出力
	ARGB Color	Color.ByARGB A, R, G, B color		

### 色の値のクエリー

次の表に記載されている各ノードにより、色を定義するアルファ、赤、緑、青の各プロパティのクエリーが実行されます。Color.Components ノードは、これら 4 つのプロパティをそれぞれ異なる出力として生成します。そのため、色のプロパティのクエリーを実行する場合は、このノードを使用すると便利です。

アイコン	名前	構文	入力	出力
	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

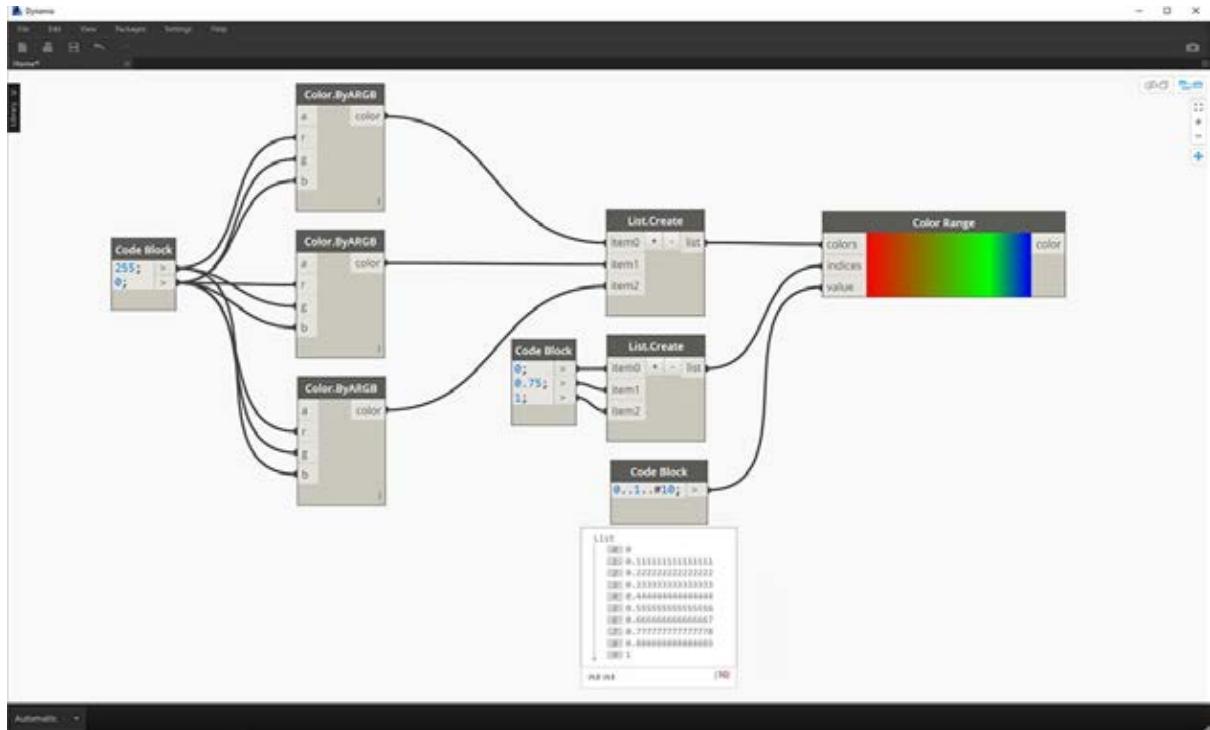
次の表に記載されている色は、**HSB** 色空間に対応しています。色を、色相、彩度、明るさに分割すると、より直感的に色を解釈することができます。たとえば、最初に処理する色を決め、次にその色の彩度と明るさを設定します。このように、色相、彩度、明るさをそれぞれ個別に設定していきます。

アイコン	クエリー名	構文	入力	出力
	Hue	Color.Hue	color	Hue
	Saturation	Color.Saturation	color	Saturation
	Brightness	Color.Brightness	color	Brightness

## 色範囲

色範囲は、セクション 4.2 で説明した、数値のリストを別の範囲に再マッピングする **Remap Range** ノードに似ています。ただし、色範囲は、数値の範囲にマッピングされるのではなく、入力された 0 から 1 までの数値に基づいて色のグラデーションにマッピングされます。

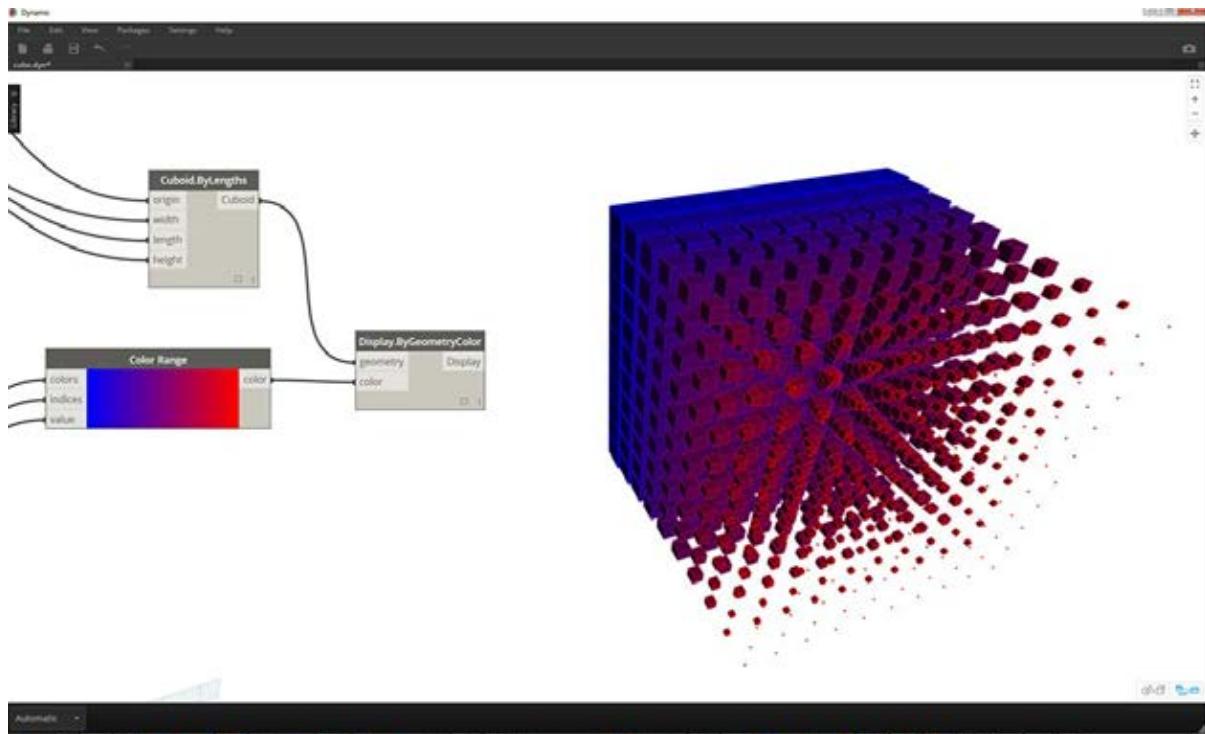
現在のノードは正しく機能しますが、最初からすべてを正しく機能させるのは少し大変です。色のグラデーションを理解するための最適な方法は、色のグラデーションを対話式に試す方法です。ここでは、簡単な演習を行い、数値に対応する色の出力を使用してグラデーションを設定する方法を確認します。



1. 3つの色を定義する: Code Block ノードを使用して 0 から 255 までの適切な数値の組み合わせに接続することにより、赤、緑、\*\*青\*\*を定義します。
2. リストを作成する: 3つの色を 1 つのリストにマージします。
3. インデックスを定義する: 0 から 1 までの範囲で、各色のグリップ位置を定義するリストを作成します。緑の値が 0.75 になっていることに注意してください。これにより、色範囲スライダの水平方向のグラデーションの 4 分の 3 が緑色になります。
4. Code Block ノードを設定する: 0 から 1 までの値を入力することで、グラデーションを色に変換します。

### 色のプレビュー

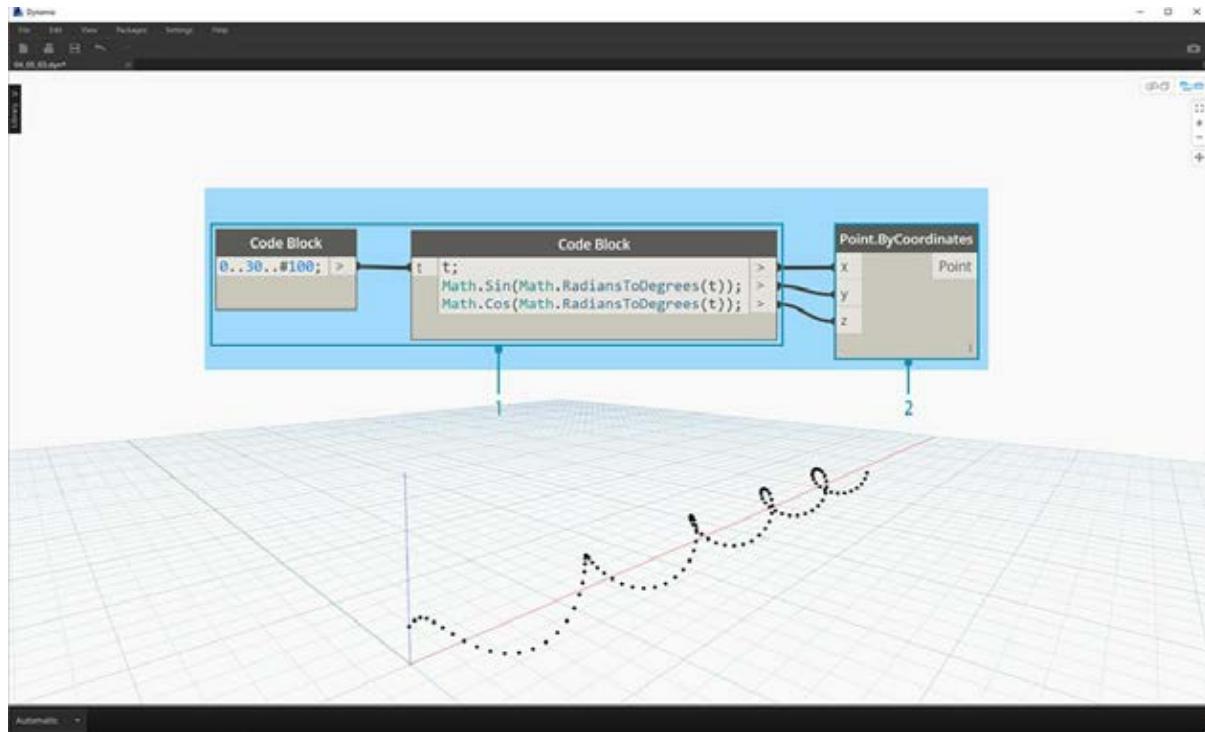
**Display.ByGeometry** ノードを使用すると、Dynamo のビューポート内でジオメトリに色を付けることができます。この機能は、ジオメトリの各種タイプを区別する場合、パラメータの概念を表現する場合、シミュレーション用の解析凡例を定義する場合に便利です。この場合の入力は単純で、ジオメトリと色だけです。上の図のようなグラデーションを作成するには、color 入力を Color Range ノードに接続します。



## 色の操作に関する演習

この演習に付属しているサンプル ファイルをダウンロードしてください(右クリックして[名前を付けてリンク先を保存...]を選択):  
[Building Blocks of Programs - Color.dyn](#)。すべてのサンプル ファイルの一覧については、付録を参照してください。

この演習では、パラメータを使用して、ジオメトリと並行して色の管理を行います。この演習で使用するジオメトリは、単純ならせん構造です。このらせん構造は、Code Block ノード (3.2.3) を使用して定義します。これは、パラメータを使用する関数をすばやく簡単に作成するための方法です。この演習の目的は、ジオメトリではなく色を操作することであるため、コード ブロックを使用してキャンバスを見やすい状態に保ったまま、らせん構造を効率的に作成します。この手引ではより高度なマテリアルを取り上げるようになるため、ここからは、コード ブロックを頻繁に使用することになります。

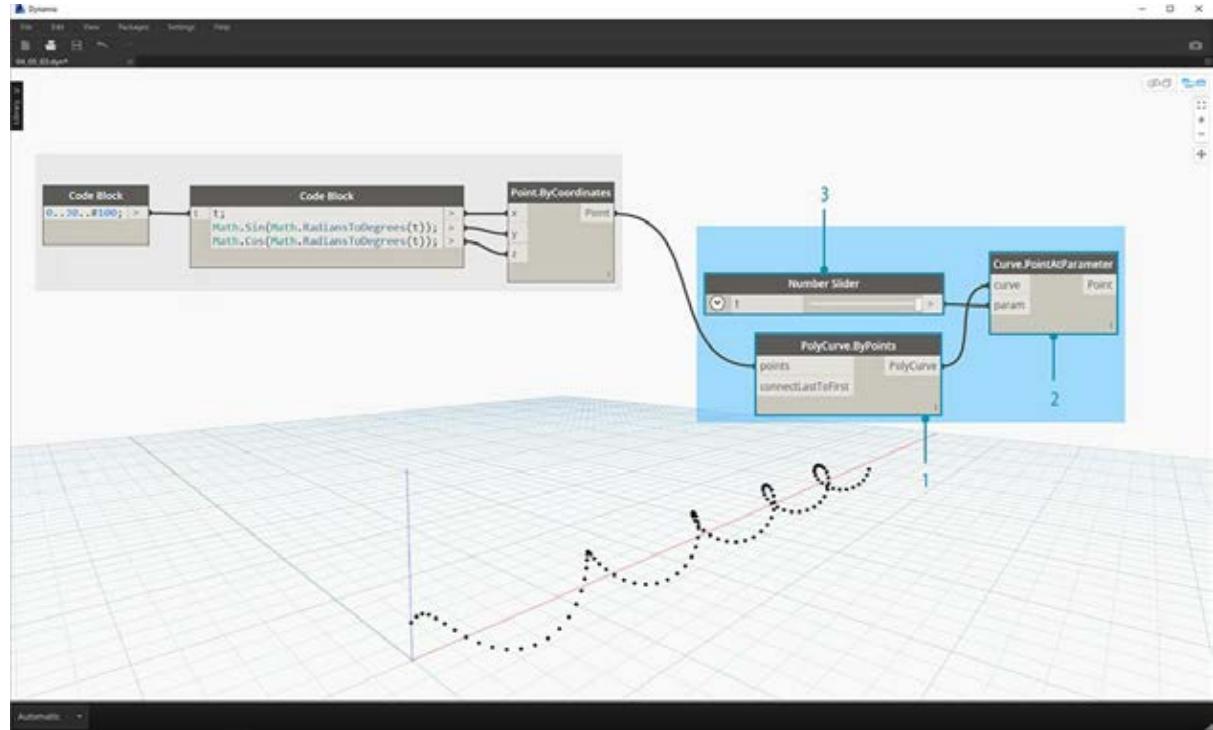


1. Code Block ノードを使用して、上図に示す式を持つ 2 つのコード ブロックを定義します。これは、パラメータを使用し

らせん構造をすばやく作成するための方法です。

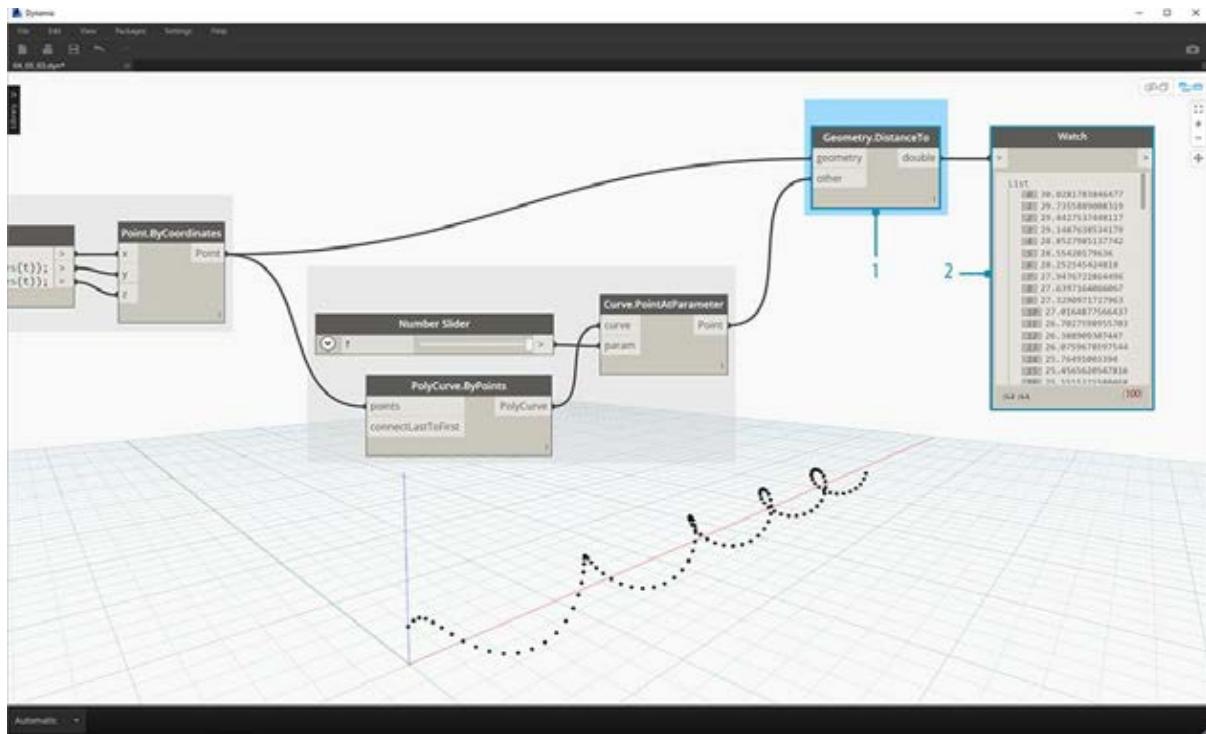
2. **Point.ByCoordinates** ノードの座標(x,y,z)に Code Block ノードの 3 つの出力を接続します。

これで、らせん構造を形成する点の配列が表示されます。次の手順では、このらせん構造の点群から曲線を作成して、完全ならせん構造を作成します。



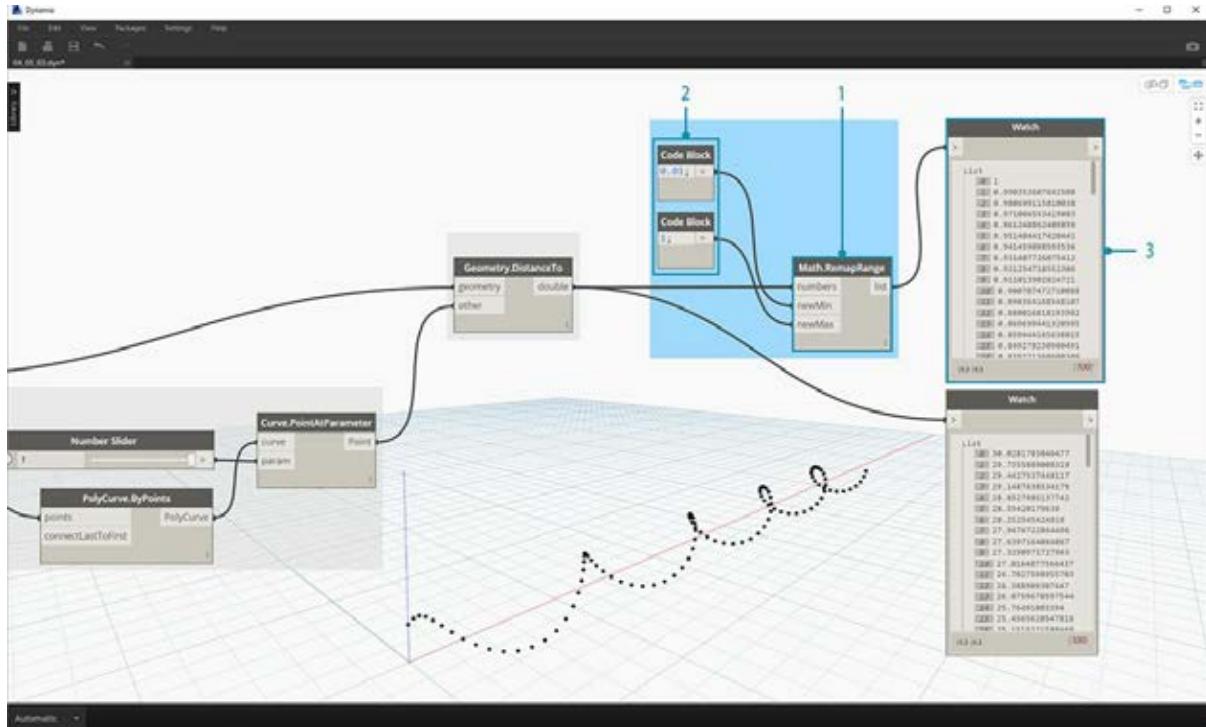
1. **PolyCurve.ByPoints** ノードの *points* 入力に *Point.ByCoordinates* ノードの出力を接続します。これにより、らせん状の曲線が作成されます。
2. **Curve.PointAtParameter** ノードの *curve* 入力に *PolyCurve.ByPoints* ノードの出力を接続します。この手順の目的は、曲線に沿ってスライドするパラメータのアトラクタ点を作成することです。この曲線によってパラメータの点が評価されるため、0 から 1 の範囲で *param* の値を入力する必要があります。
3. **Number Slider** ノードをキャンバスに追加したら、*Min* の値を 0.0、*Max* の値を 1.0、*Step* の値を .01 に変更します。次に、*Number Slider* ノードの出力を *Curve.PointAtParameter* ノードの *param* 入力に接続します。らせん構造全体に沿って表示されている点を、スライダのパーセンテージとして表すようになりました(開始点は 0、終点は 1)。

これで、参照点が作成されました。次に、この参照点から、らせん構造を定義する元の点までの距離を比較します。この距離により、色とジオメトリをコントロールします。



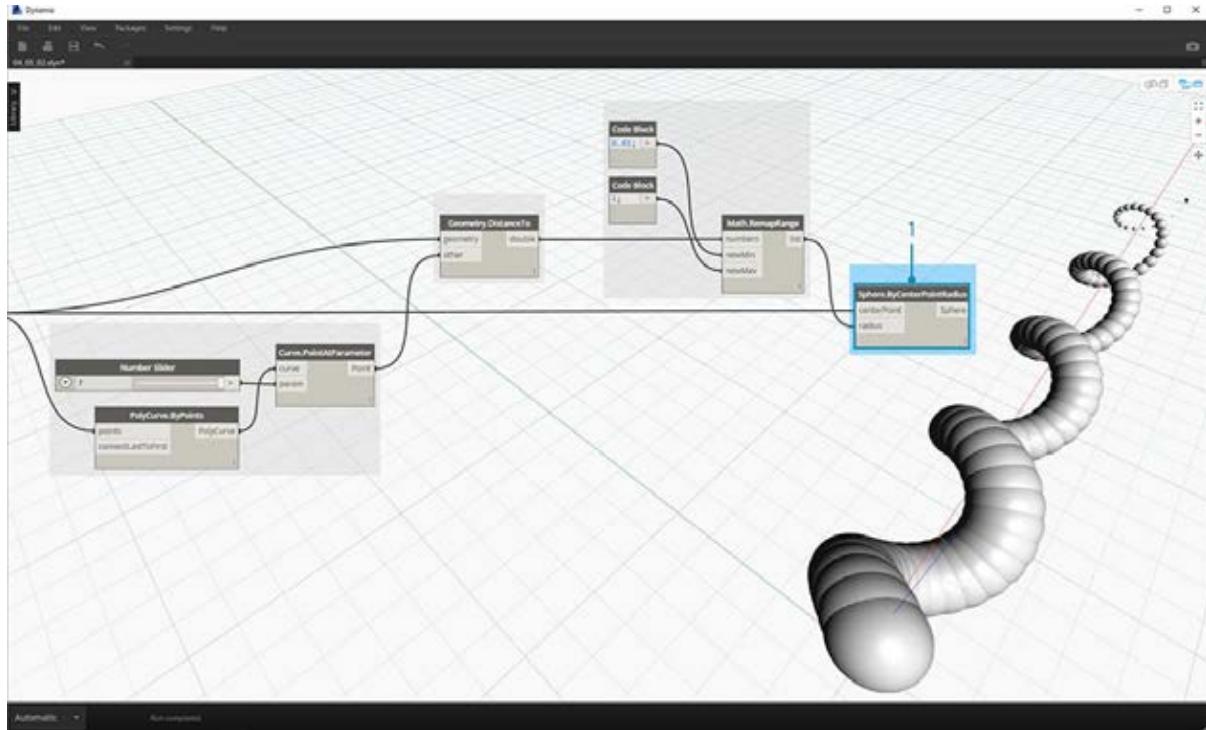
1. **Geometry.DistanceTo** ノードの **other** 入力に **Curve.PointAtParameter** ノードの出力を接続します。次に、**Point.ByCoordinates** ノードを **Geometry.DistanceTo** ノードの **geometry** 入力に接続します。
2. **Watch** ノードに、らせん構造の曲線を構成するそれぞれの点から参照点までの距離のリストが表示されます。

次の手順では、らせん構造の各点から参照点までの距離のリストを使用して、パラメータを設定します。また、これらの距離の値を使用して、曲線に沿った一連の球形の半径を定義します。これらの球形を適切なサイズに保つには、距離の値を再マッピングする必要があります。

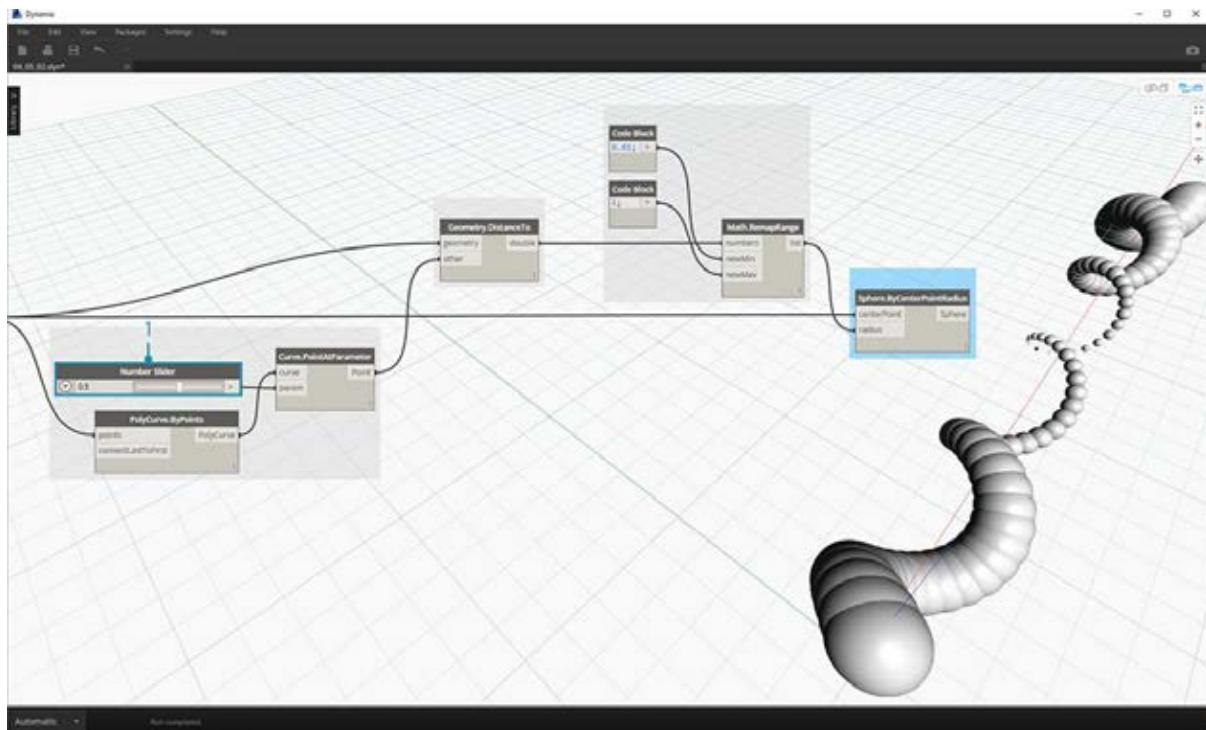


1. **Math.RemapRange** ノードの **numbers** 入力に **Geometry.DistanceTo** ノードの出力を接続します。
2. 値が **0.01** の **Code Block** ノードを **Math.RemapRange** ノードの **newMin** 入力に接続し、値が **1** の **Code Block** ノードを **Math.RemapRange** ノードの **newMax** 入力に接続します。
3. いずれかの **Watch** ノードに **Math.RemapRange** ノードの出力を接続し、もう一方の **Watch** ノードに **Geometry.DistanceTo** ノードの出力を接続します。次に、結果を比較します。

この手順により、距離のリストがより狭い範囲に再マッピングされます。再マッピングの結果が適切な場合であっても、*newMin* と *newMax* の値を編集することができます。これらの値は、範囲全体で分布比率を保持したまま再マッピングされます。

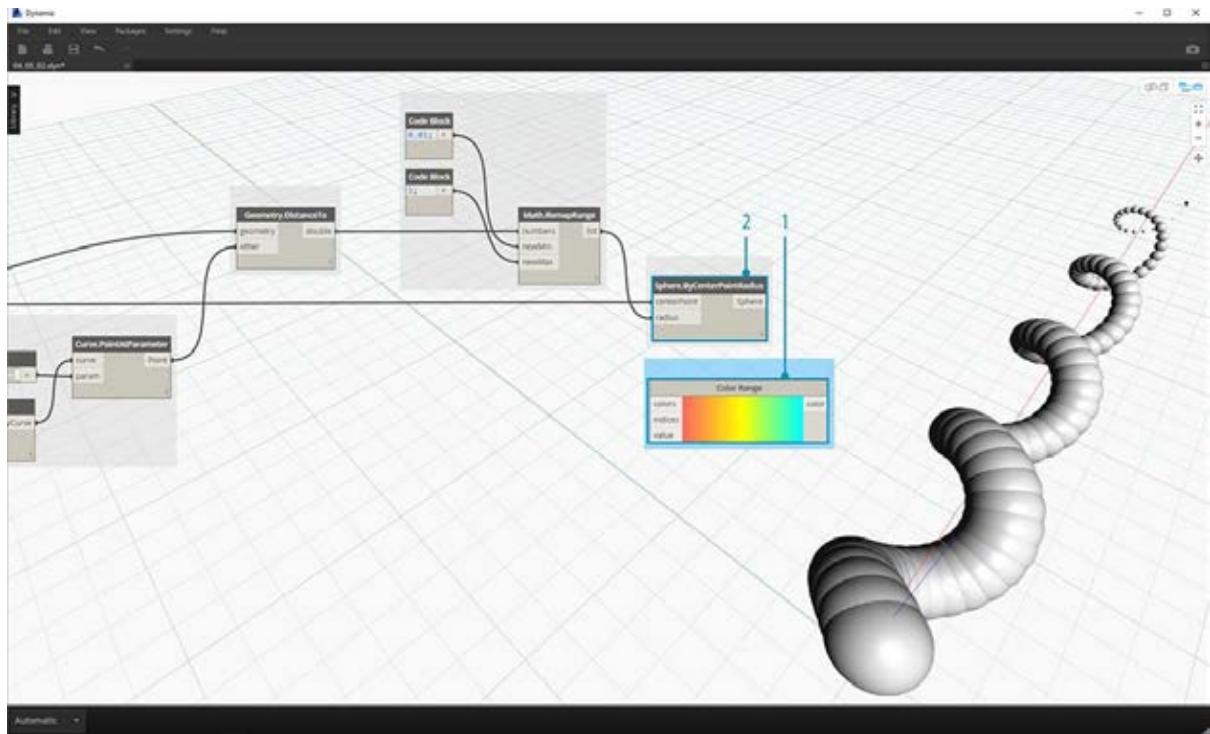


1. **Sphere.ByCenterPointRadius** ノードの *radius* 入力に *Math.RemapRange* ノードの出力を接続し、元の *Point.ByCoordinates* ノードの出力を *Sphere.ByCenterPointRadius* ノードの *centerPoint* 入力に接続します。

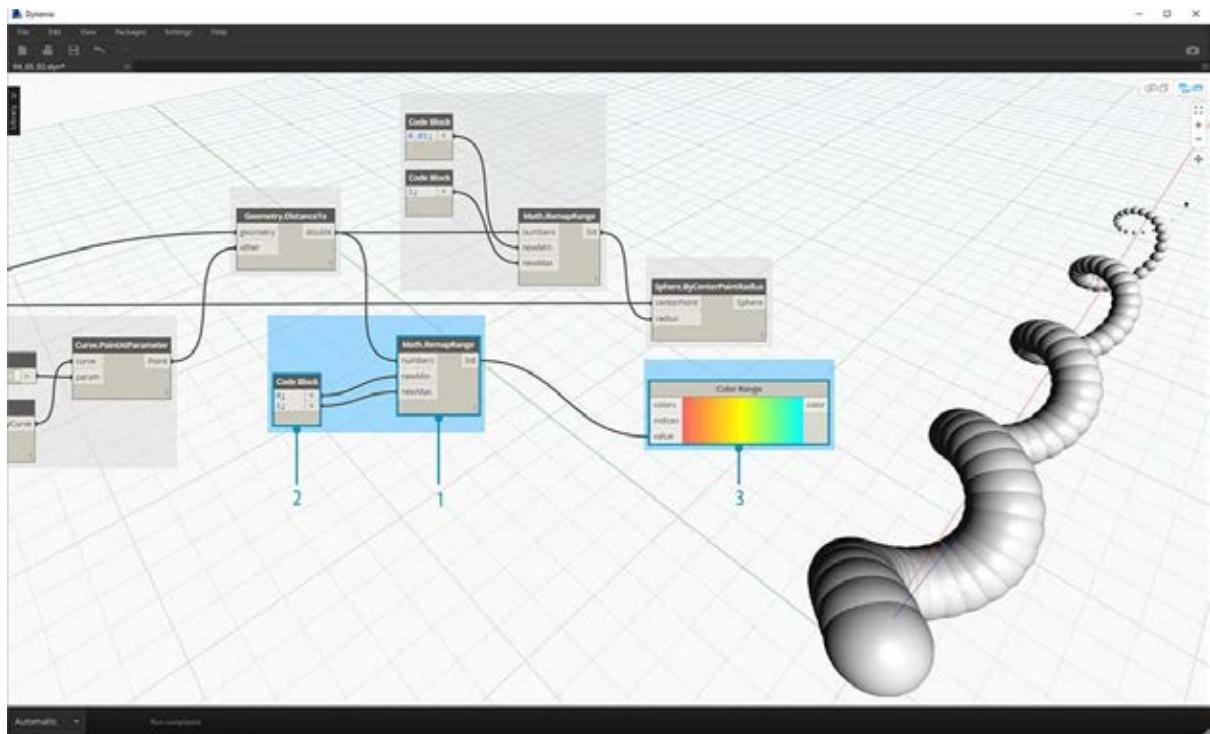


1. **Number Slider** ノードの値を変更し、球形のサイズが更新されることを確認します。ここで、パラメータツールを使用します。

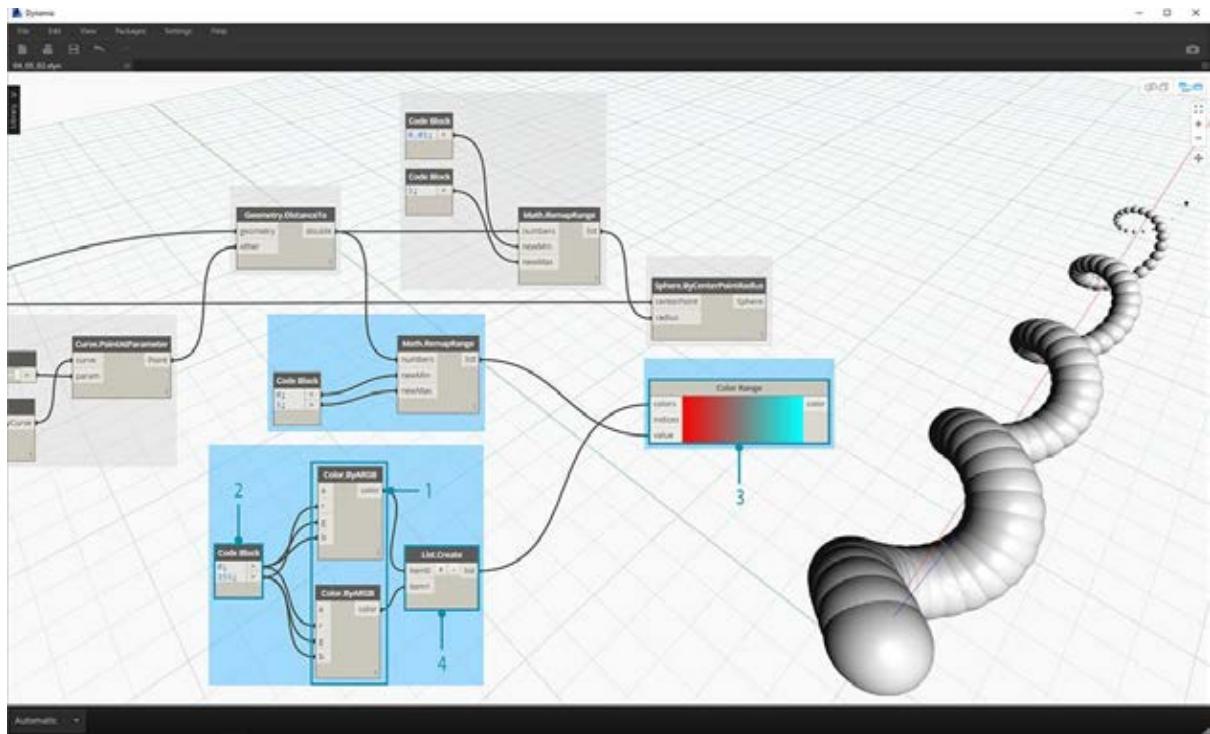
球形のサイズは、曲線に沿った参照点によって定義されるパラメータ配列を示しています。ここでは、球形の半径と同じ考え方で、球形の色を操作してみます。



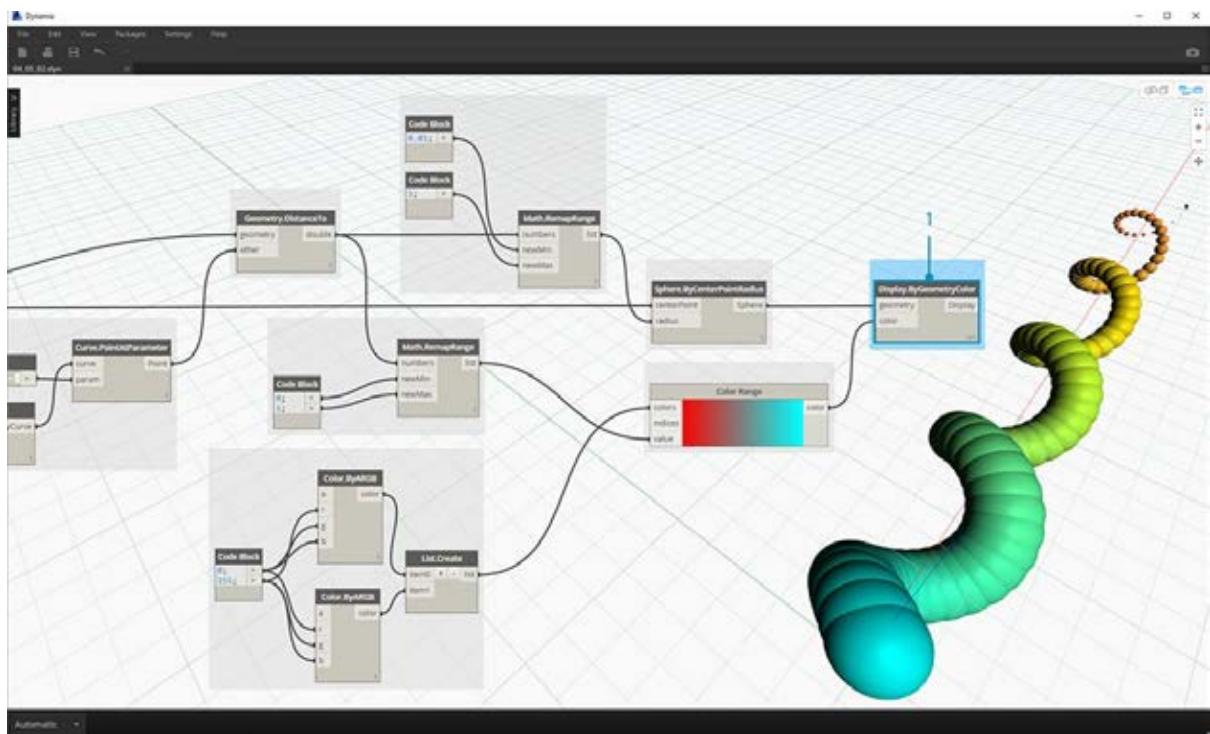
1. **Color Range** ノードをキャンバス上に追加します。value 入力にマウス ポイントを置くと、0 から 1 までの範囲で数値を指定する必要があることがわかります。ここでは、*Geometry.DistanceTo* ノードの出力から数値を再マッピングして、それらの数値をこの範囲に対応させる必要があります。
2. **Sphere.ByCenterPointRadius** ノードでのプレビューを一時的に無効にします(右クリックして[プレビュー]を選択)。



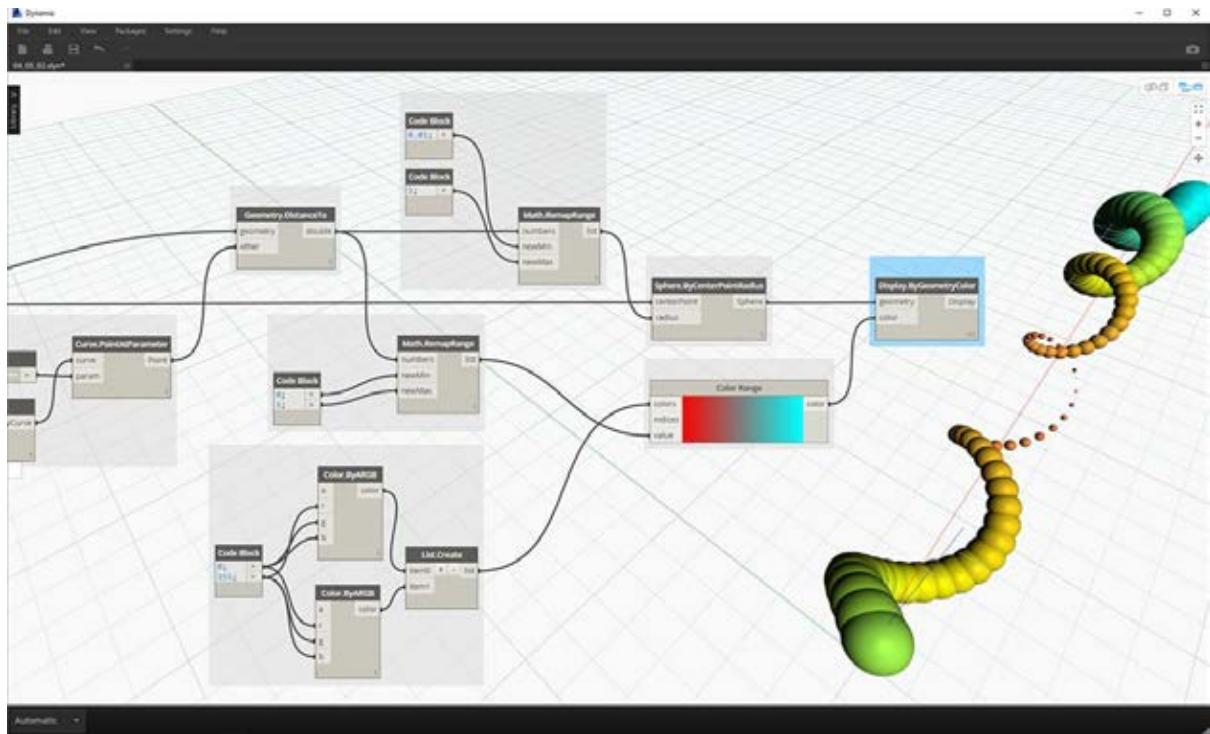
1. **Math.RemapRange** ノードの numbers 入力にも同様に *Geometry.DistanceTo* ノードの出力を接続します。
2. **Code Block** ノードを使用して、**Math.RemapRange** ノードの newMin 入力の値として 0 を設定し、newMax 入力の値として 1 を設定します。この場合、1 つのコード ブロックで 2 つの出力を定義できることに注意してください。
3. **Color Range** ノードの value 入力に **Math.RemapRange** 出力を接続します。



1. **Color.ByARGB** ノードを使用して、2つの色を作成します。この手順は複雑そうに感じるかもしれません、他のソフトウェアで使用する RGB カラーの場合と同じです。ここでは、ビジュアル プログラミングを使用して色を処理するということだけのことです。
2. **Code Block** ノードを使用して、0 と 255 という 2 つの値を作成します。次に、上の図のように、この 2 つの値を **Color.ByARGB** の各入力に接続します(または、任意の色を 2 つ作成します)。
3. **Color Range** ノードの *colors* 入力では、色のリストが必要になります。上の手順で作成した 2 つの色を使用して、色のリストを作成する必要があります。
4. **List.Create** ノードを使用して、2 つの色を 1 つのリストにマージします。次に、このノードの出力を **Color Range** ノードの *colors* 入力に接続します。



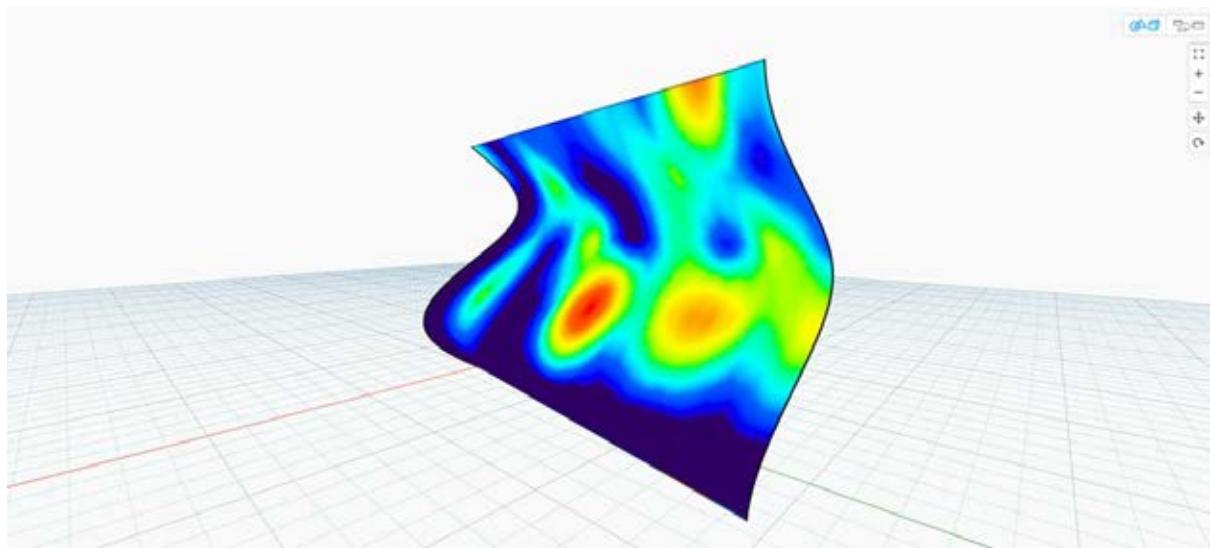
1. **Display.ByGeometryColor** ノードの *geometry* 入力に **Sphere.ByCenterPointRadius** ノードを接続し、**Color Range** ノードを **Display.ByGeometryColor** ノードの *color* 入力に接続します。これで、曲線領域全体にスムーズな色のグラデーションが適用されます。



前の手順の *Number Slider* ノードの値を定義内で変更すると、色とサイズが変更されます。この場合、色と半径のサイズは相互に直接関係しています。これで、2つのパラメータ間の関係を視覚的に確認することができます。

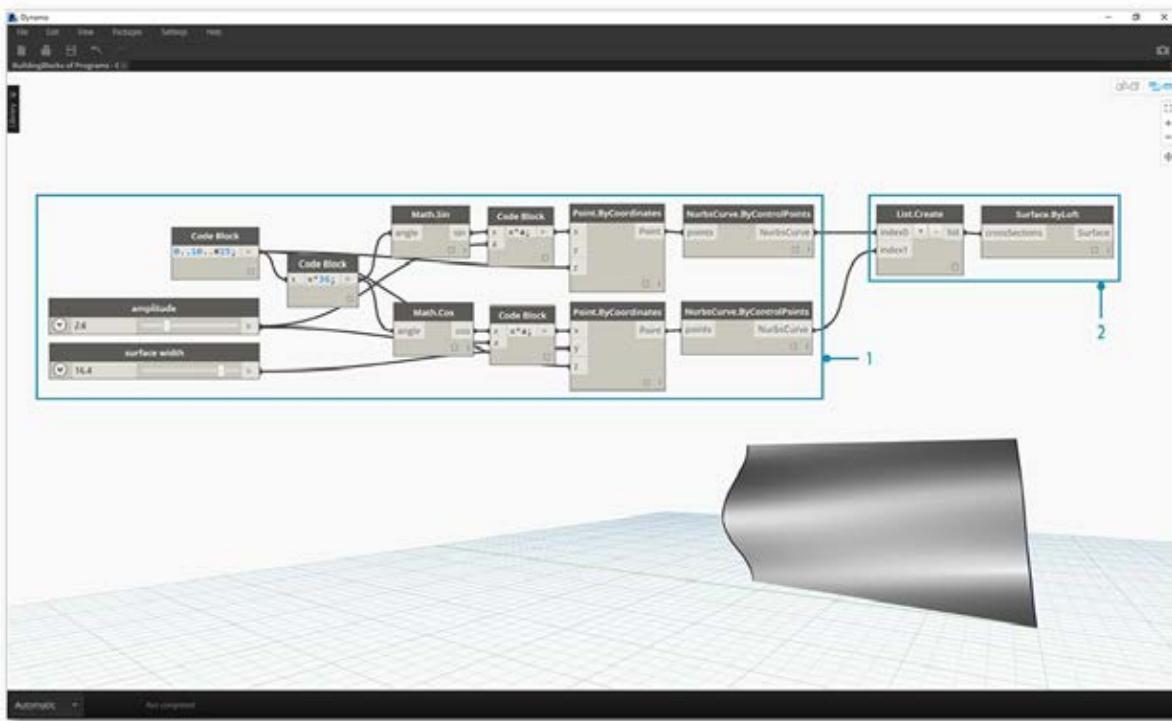
### サーフェス上の色

*Display.BySurfaceColors* ノードを使用すると、サーフェス全体にデータを色でマッピングすることができます。この機能により、日照解析、エネルギー解析、近接度解析など、各種の解析で取得したデータを視覚化することができます。Dynamo では、他の CAD 環境でマテリアルにテクスチャを適用する場合と同様に、サーフェスに色を適用することができます。次の簡単な演習で、このツールの使用方法を確認します。



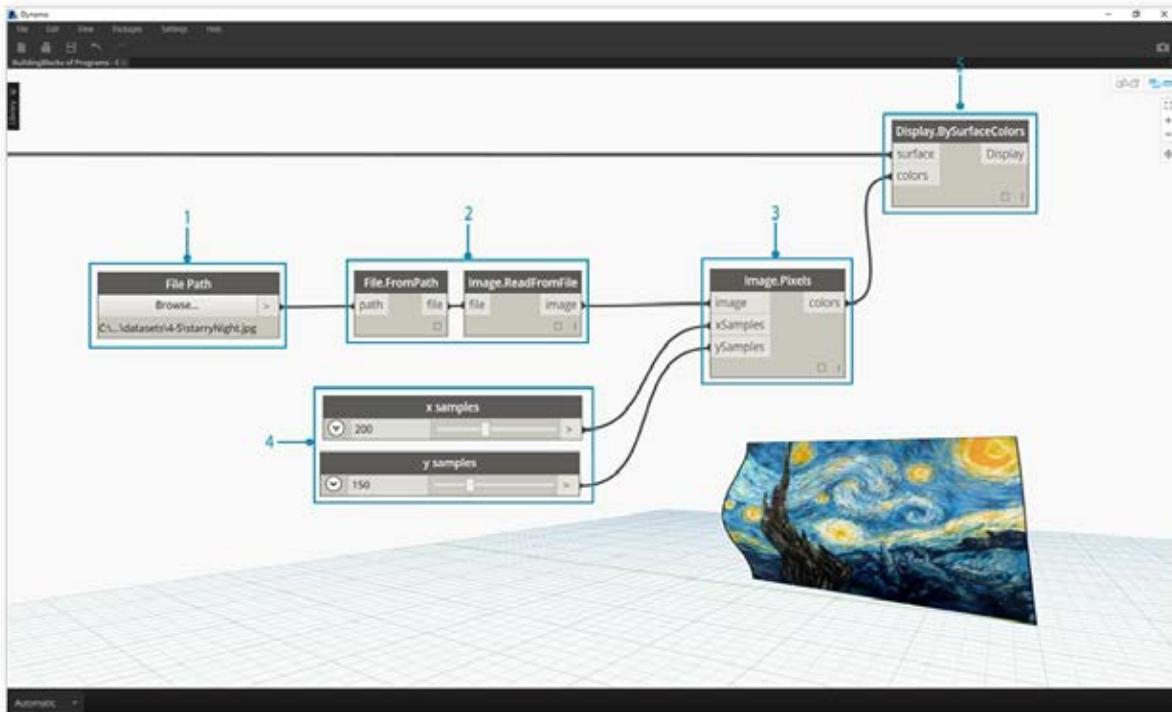
### サーフェスで色を処理するための演習

この演習に付属しているサンプル ファイルをダウンロードしてください(右クリックして[名前を付けてリンク先を保存...]を選択): [Building Blocks of Programs - ColorOnSurface.zip](#)。すべてのサンプル ファイルの一覧については、付録を参照してください。



最初に、**Display.BySurfaceColors** ノードの入力として使用するサーフェスを作成(または参照)する必要があります。この例では、正弦曲線と余弦曲線間をロフトします。

1. このノード グループは、Z 軸に沿って点を作成してから、正弦関数と余弦関数に基づいてそれらの点の位置を変更します。その後、2 つの点リストを使用して NURBS 曲線が生成されます。
2. **Surface.ByLoft** ノードを使用して、NURBS 曲線のリスト間に、補間されたサーフェスを生成します。



1. **File Path** ノードを使用して、下流のピクセルデータのサンプリングを行うためのイメージ ファイルを選択します。
2. **File.FromPath** ノードを使用してファイルパスをファイルに変換し、そのファイルを **Image.ReadFromFile** ノードに渡してサンプリング用のイメージを出力します。
3. **Image.Pixels** ノードを使用してイメージを入力し、そのイメージの X、Y 座標に対応して使用されるサンプル値を指定します。

4. Slider ノードを使用して、Image.Pixels ノードのサンプル値を指定します。
5. Display.BySurfaceColors ノードを使用して、色の値の配列を、X、Y 座標に対応してサーフェス全体にマッピングします。

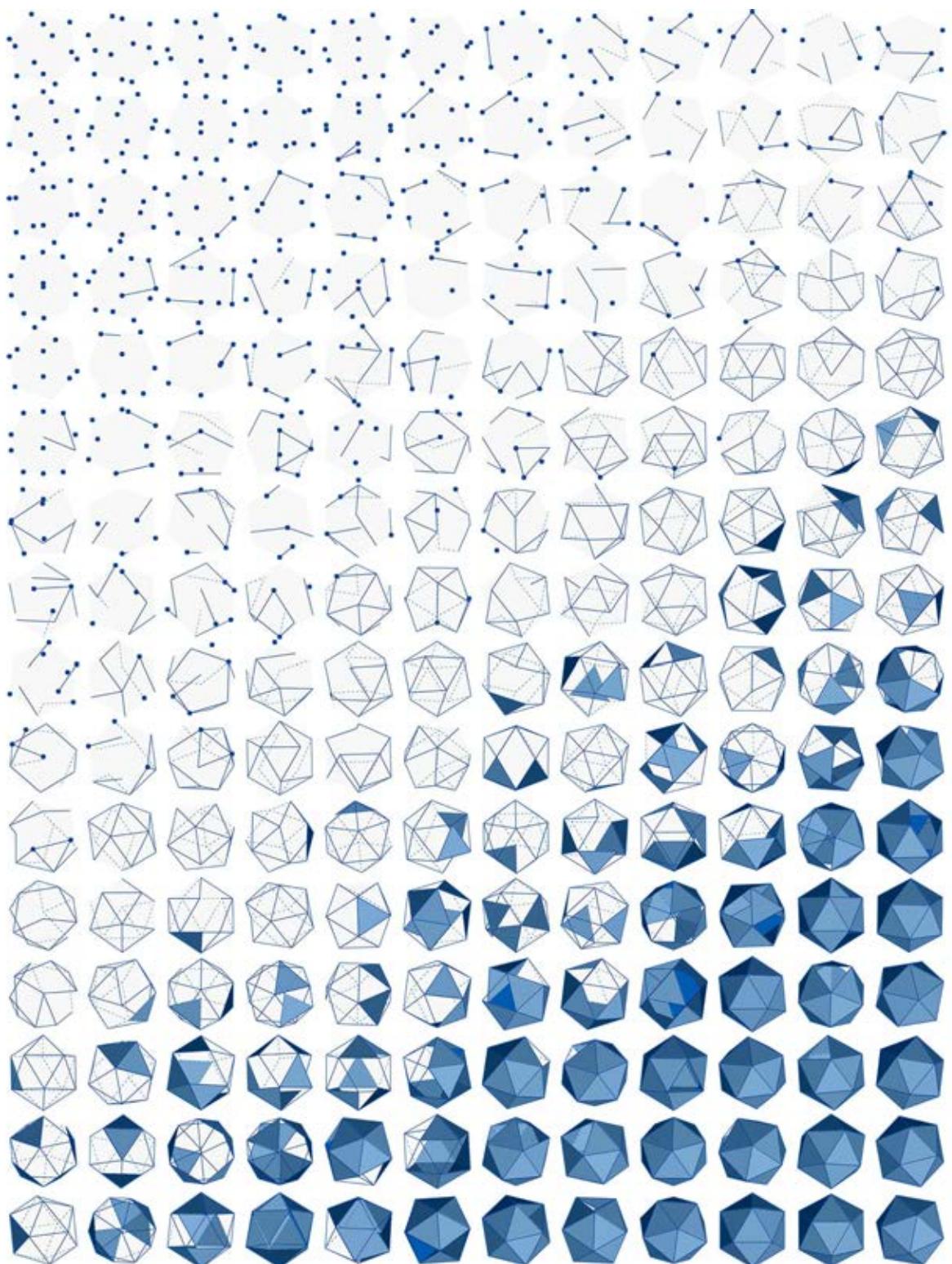


400x300 のサンプル解像度で、出力サーフェスのプレビューを拡大します。

## 計算設計用のジオメトリ

## 計算設計用のジオメトリ

ビジュアル プログラミング環境の Dynamo では、データの処理方法を構築することができます。数値やテキストだけでなく、ジオメトリもデータです。コンピュータが認識できるジオメトリ(計算設計用のジオメトリと呼ぶ場合もあります)は、美しいモデル、複雑なモデル、パフォーマンス重視のモデルを作成するためのデータです。ジオメトリを使用するには、そのジオメトリのさまざまな入力と出力を理解する必要があります。



# ジオメトリの概要

## ジオメトリの概要

ジオメトリは、設計用の言語です。プログラミング言語やプログラミング環境の核心部分でジオメトリ カーネルを使用している場合は、設計ルーチンを自動化し、アルゴリズムを使用して設計の繰り返し部分を生成することにより、正確で安定したモデルを設計することができます。

### 基本的な概念

ジオメトリとは、古典的な定義によれば、形状の外形、サイズ、相対的位置、空間上の特性に関する研究(幾何学)のことです。この分野には、数千年にもおよぶ豊かな歴史があります。コンピュータの出現と普及により、私たちは、ジオメトリの定義、研究、生成を行うための強力な手段を手に入れました。現在では、ジオメトリの複雑な相互作用の結果を簡単に計算できるようになりました。



コンピュータを利用してどれほど多様で複雑なジオメトリを作成できるかということを確かめるには、「Stanford Bunny」という文字列で Web を検索してみてください。「Stanford Bunny」とは、アルゴリズムのテストで使用されている標準モデルのことです。

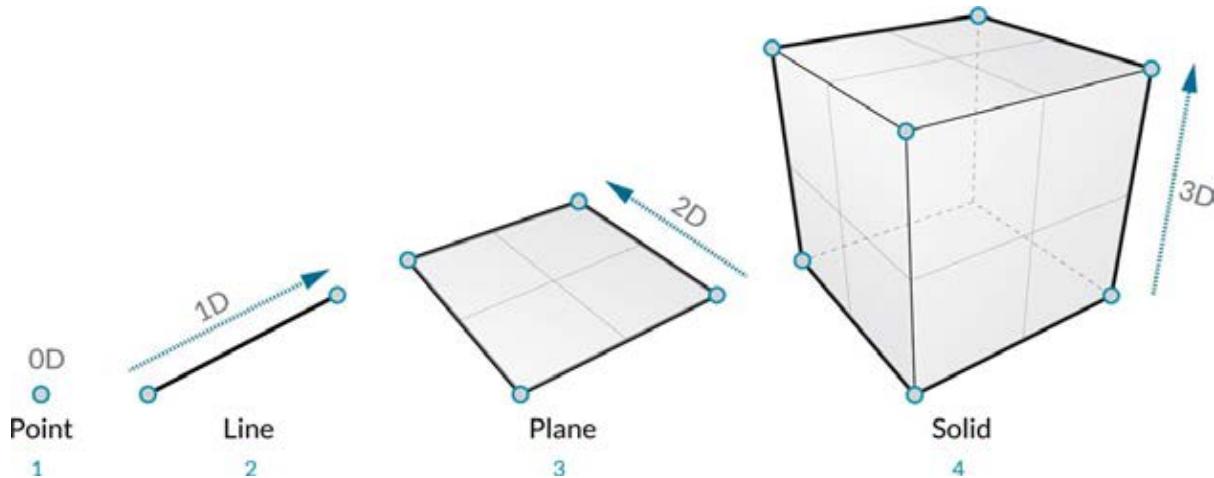
アルゴリズム、コンピューティング、複雑性という観点からジオメトリを理解するのは、難しそうだと感じるかもしれません。しかし、比較的単純ないくつかの重要な原則を理解すれば、それをベースとしてより高度なケースに適用することができます。

1. ジオメトリはデータです。コンピュータや Dynamo にとって、Stanford Bunny と数値に大きな違いはありません。
2. ジオメトリは、抽象化に依存します。基本的に、ジオメトリの要素は、特定の空間座標系内で、数値、関係、計算式によって記述されます。
3. ジオメトリには階層があります。点が集まって線が構成され、線が集まって面が構成されます。
4. ジオメトリは、細部と全体の両方を同時に記述します。たとえば曲線が存在する場合、その曲線は曲線の形状を表すと同時に、その曲線を構成する点群も表します。

実際の作業では、複雑なモデルの開発においてさまざまなジオメトリの構成、解体、再構成を柔軟に実行できるように、作業の内容(ジオメトリのタイプやジオメトリの構成など)について理解しておく必要があります。

### 階層とは

ここで、ジオメトリの抽象と階層との関係について簡単に説明します。これら 2 つの概念は相互に関係していますが、この関係を最初から明確に理解するのは必ずしも簡単なことではありません。そのため、複雑なワークフローやモデルの開発を開始すると、すぐに理解の壁に直面することがあります。ここでは、入門者向けに次元という概念を使用して、モデリングについて簡単に説明します。1 つの形状を記述するためにいくつの次元が必要になるかを考えると、ジオメトリの階層構成を理解する手がかりになります。



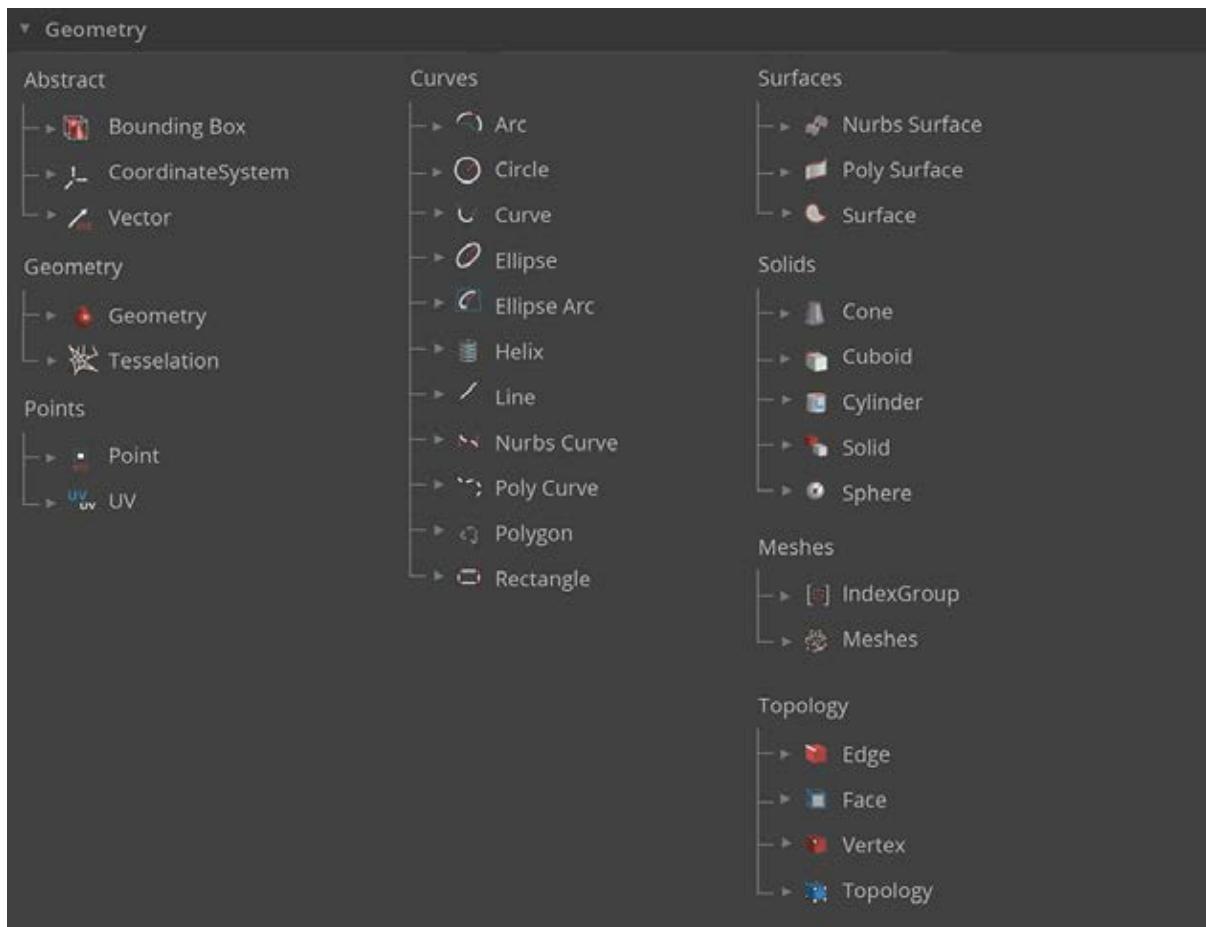
- 座標によって定義される点には、次元は存在しません。点は、各座標を示す単なる数字に過ぎません。
- 2つの点によって定義される線には、1つの次元が存在します。線の上では、前方(正の方向)または後方(負の方向)に向かって移動することができます。
- 複数の線によって定義される面には、2つの次元が存在します。面の上では、前後だけでなく左右に移動することもできます。
- 複数の面によって定義される直方体には、3つの次元が存在します。ボックスの中では、前後左右に加えて、高低の位置関係を定義することができます。

ジオメトリを分類する場合、次元という概念は役に立ちますが、必ずしもそれが最適な概念というわけではありません。なぜなら、点、線、面、直方体だけを使用してモデルを作成することはほとんどないからです。曲線や曲面を使用する場合を考えてみれば、それは明らかです。また、方向、体積、ペーパー間の関係などを定義する、完全に抽象的なジオメトリタイプのまったく異なるカテゴリもあります。ベクトルを実際にこの手でつかむことはできません。では、空間内に見えているものに対してベクトルを定義するにはどうすればいいでしょうか。ジオメトリの階層をさらに細分化すると、抽象的なジオメトリタイプを「補助的なジオメトリタイプ」とみなすべきだということがわかります。各タイプは、その機能と、モデル要素の形状を記述するタイプを基準として、グループ化することができます。

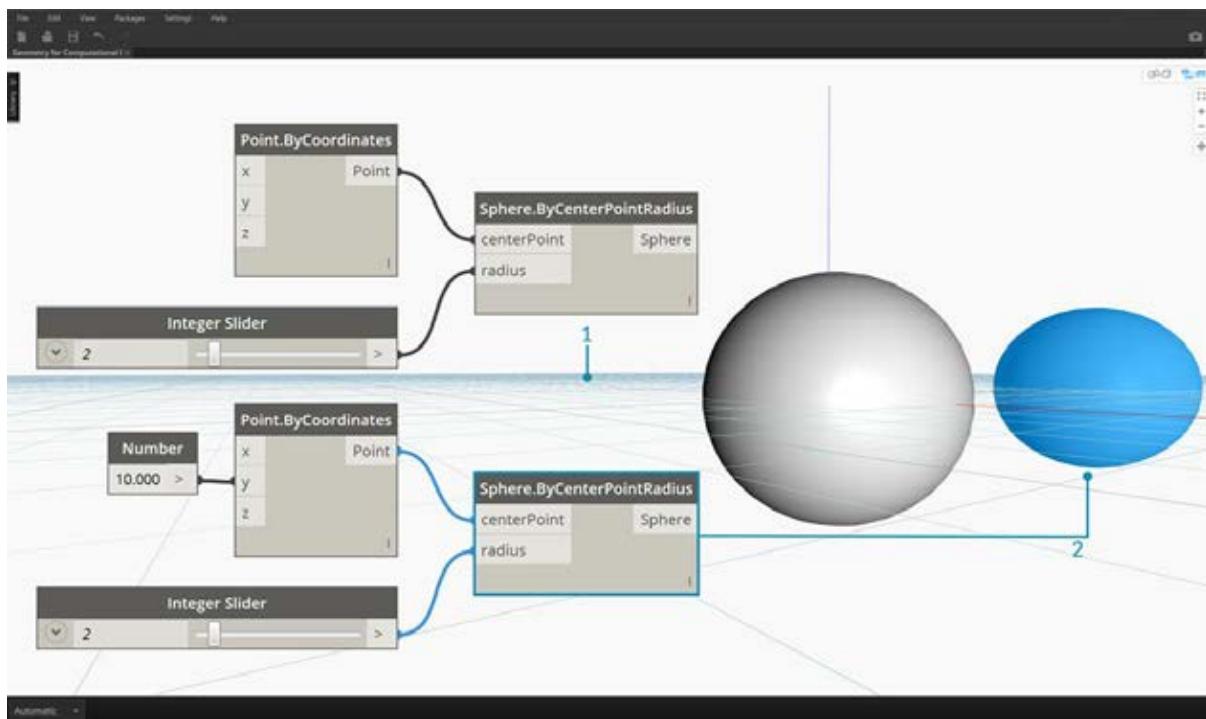
Data Type Hierarchy							
Abstract Types			Geometry Types				
Defines Location + Orientation	Defines Position + Volume	Defines Relationships	Model Elements				
Vector	Bounding Box	Topology	Point	Curve	Surface	Solid	Mesh
<ul style="list-style-type: none"> <li>Vector</li> <li>Plane</li> <li>Coordinate System</li> </ul>	<ul style="list-style-type: none"> <li>Bounding Box</li> </ul>	<ul style="list-style-type: none"> <li>Vertex</li> <li>Edge</li> <li>Face</li> </ul>	<ul style="list-style-type: none"> <li>XYZ Coordinate</li> <li>UV Coordinate</li> </ul>	<ul style="list-style-type: none"> <li>Line</li> <li>Polygon</li> <li>Arc</li> <li>Circle</li> <li>Ellipse</li> <li>NURBS Curve</li> <li>PolyCurve</li> </ul>	<ul style="list-style-type: none"> <li>NURBS Surface</li> <li>PolySurface</li> </ul>	<ul style="list-style-type: none"> <li>Cuboid</li> <li>Sphere</li> <li>Cone</li> <li>Cylinder</li> </ul>	<ul style="list-style-type: none"> <li>Mesh</li> </ul>

## Dynamo Studio におけるジオメトリ

ジオメトリの階層は、Dynamo の使用に関してどのような意味を持つでしょうか。ジオメトリのタイプとその相互関係を理解すれば、ライブラリ内で使用できるジオメトリノードのグループをナビゲートできるようになります。ジオメトリノードは、階層別ではなくアルファベット順に整理されています。Dynamo のユーザインターフェース内のレイアウトに似たイメージで表示されます。



Dynamoでモデルを作成し、[背景プレビュー]でプレビュー表示される内容をグラフ内のデータフローに接続する作業は、時間の経過に伴ってより直感的に行うことができるようになります。



1. 仮の座標系が色付きのグリッドラインによってレンダリングされています。
2. ノードを選択すると、そのノードによってジオメトリが作成される場合は、対応するジオメトリが背景内でハイライト表示されます。この画像に付属しているサンプル ファイルをダウンロードしてください(右クリックして[名前を付けてリンク先を

[保存...]を選択): [Geometry for Computational Design - Geometry Overview.dyn](#)。すべてのサンプルファイルの一覧については、付録を参照してください。

## ジオメトリの詳細を確認する

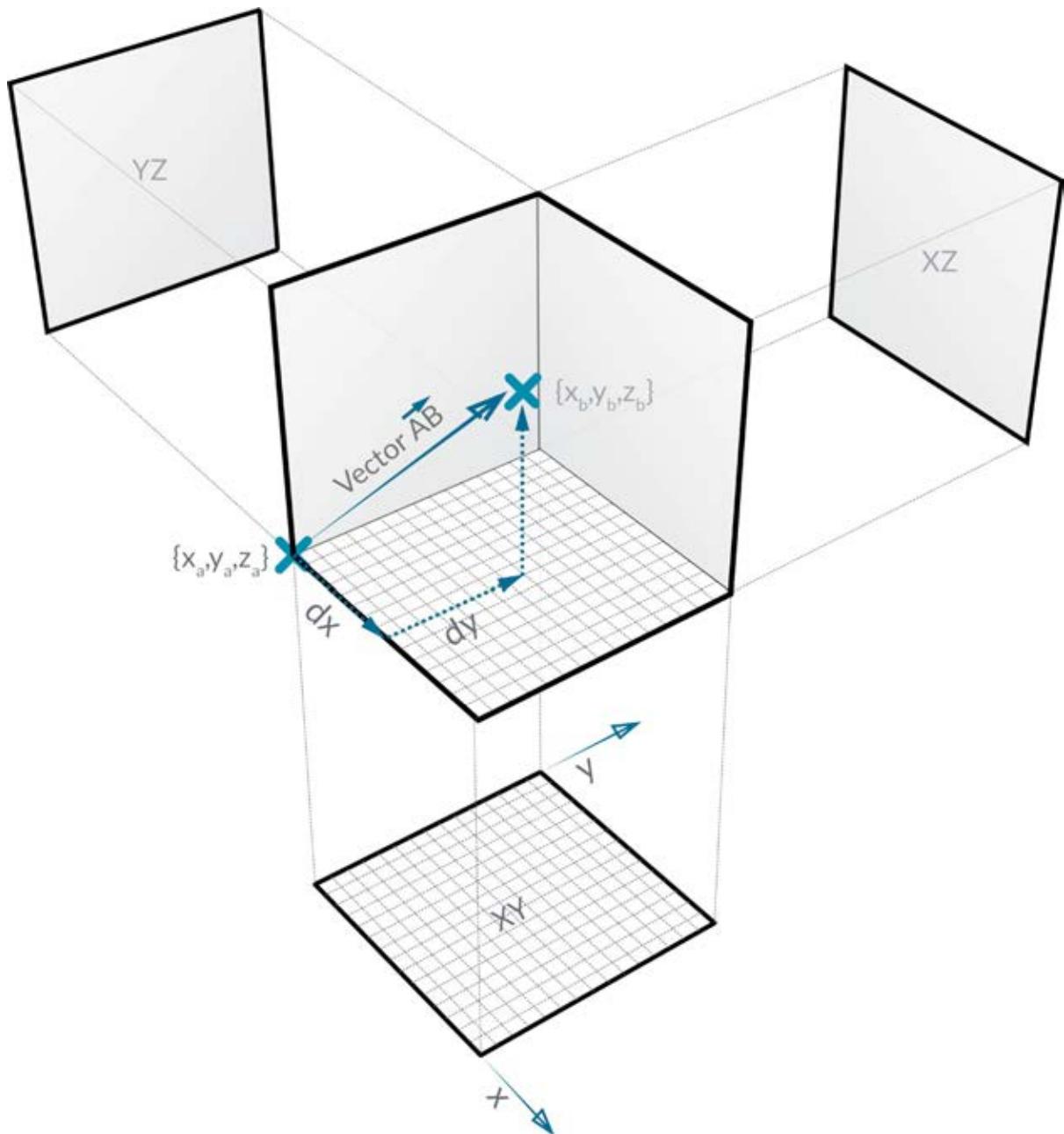
Dynamo で作成するモデルは、ノードを使用して生成されるモデルだけではありません。ジオメトリでの作業を次のレベルに進めるには、キーとなる方法がいくつかあります。

1. Dynamo では、CSV ファイルを使用して点群を読み込んだり、SAT ファイルを読み込んでサーフェスを作成するなど、ファイルを読み込んでさまざまな操作を行うことができます。
2. Dynamo を Revit と組み合わせると、Revit の要素を参照して Dynamo で使用することができます。
3. Dynamo Package Manager には、ジオメトリのタイプや操作を拡張するための追加の機能が用意されています。[Mesh Toolkit](#) パッケージを確認してください。

# ベクトル、平面、座標系

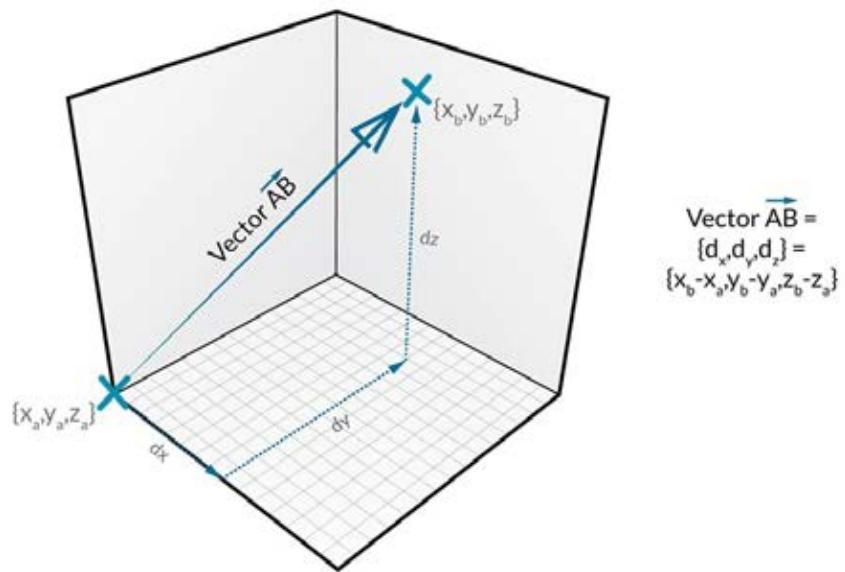
## ベクトル、平面、座標系

抽象的なジオメトリタイプの主要なグループは、ベクトル、平面、座標系により構成されています。ベクトル、平面、座標系により、形状を表すその他のジオメトリの位置、方向、空間コンテキストを定義することができます。たとえば、ニューヨーク市 42 丁目のブロードウェイ(座標系)の路上(平面)に立って北(ベクトル)を向いている場合、ベクトル、平面、座標系という[Helper]カテゴリの情報を使用して自分の現在の居場所を定義することになります。電話ケース製品や高層ビルについても、同じことが言えます。モデルを開発するには、このコンテキストが必要です。



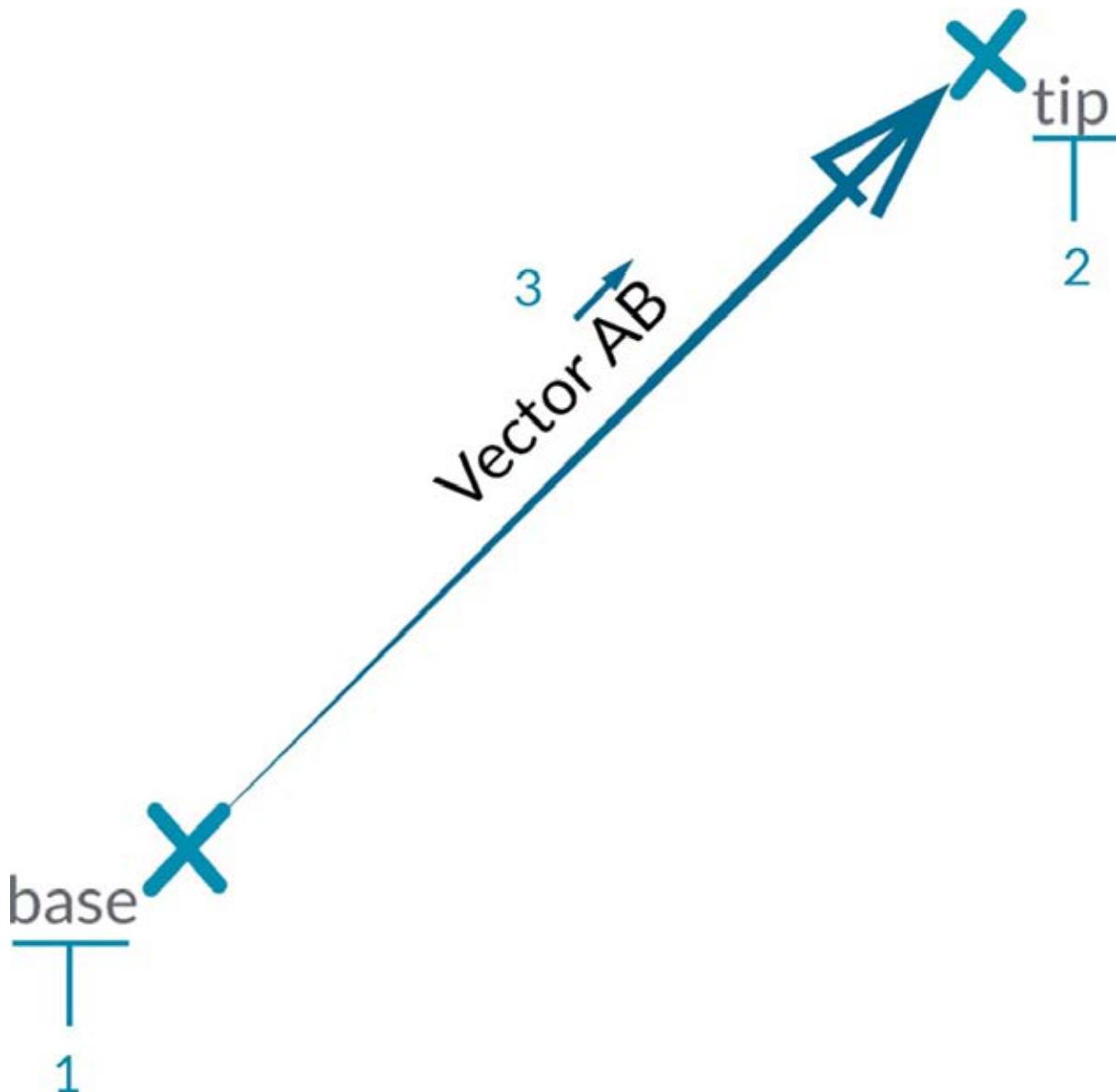
## ベクトルの概要

ベクトルとは、方向と大きさを表すジオメトリの量です。ベクトルは、特定のジオメトリ要素ではなく量を表す抽象的な概念です。ベクトルは点と同様に値のリストで構成されているため、点とベクトルを混同しないようにする必要があります。ただし、点とベクトルには大きな違いがあります。点が特定の座標系における位置を表すのに対して、ベクトルは位置における相対的な差異を表します。これは、「方向」と言い換えることもできます。



「相対的な差異」という概念がわかりにくい場合は、ベクトル AB を「点 A に立って点 B の方向を向いている」と考えてみてください。現在地 A から目的地 B に対する方向が、ベクトルです。

ここでは、同じ AB 表記を使用して、ベクトルを構成する要素について説明します。

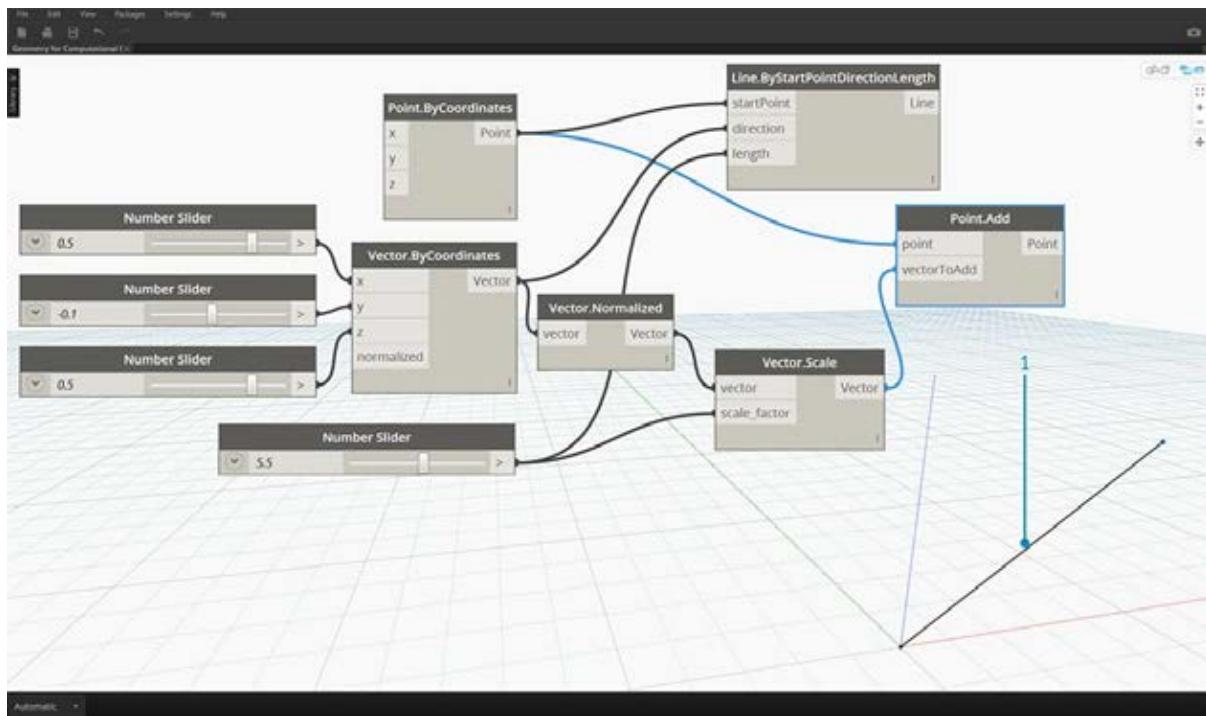


- ベクトルの開始点は、起点と呼ばれます。
- ベクトルの終了点は、先端または向きと呼ばれます。
- ベクトル  $AB$  とベクトル  $BA$  は違います。これらは、反対方向のベクトルです。

ベクトルとその定義に関するジョークとして、古典的なコメディである Airplane (邦題: フライングハイ)の有名なジョークがあります。

*Roger, Roger. What's our vector, Victor?* (訳注: ロジャー、了解だ(発音は「ロジヤー」、ラジヤー)。ビクター、機首の向き(発音は「ベクター」、ベクトル)は?)

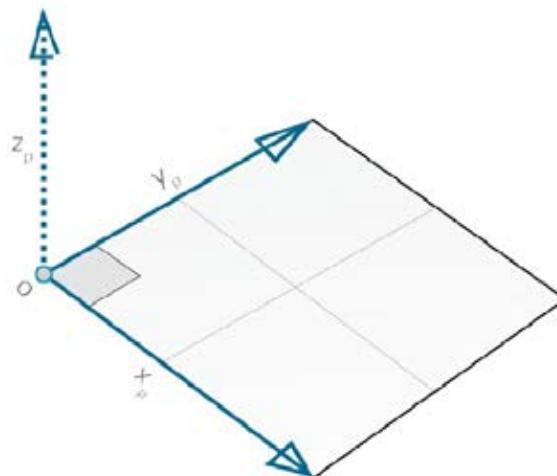
ベクトルは、Dynamo のモデルにおける主要なコンポーネントです。ベクトルは[Helper]という抽象的なカテゴリに分類されるため、ベクトルを作成しても背景プレビューには何も表示されないように注意してください。



- ベクトル プレビューの代わりに線分を使用することができます。この画像に付属しているサンプル ファイルをダウンロードしてください(右クリックして[名前を付けてリンク先を保存...]を選択): [Geometry for Computational Design - Vectors.dyn](#)。すべてのサンプルファイルの一覧については、付録を参照してください。

## 平面の概要

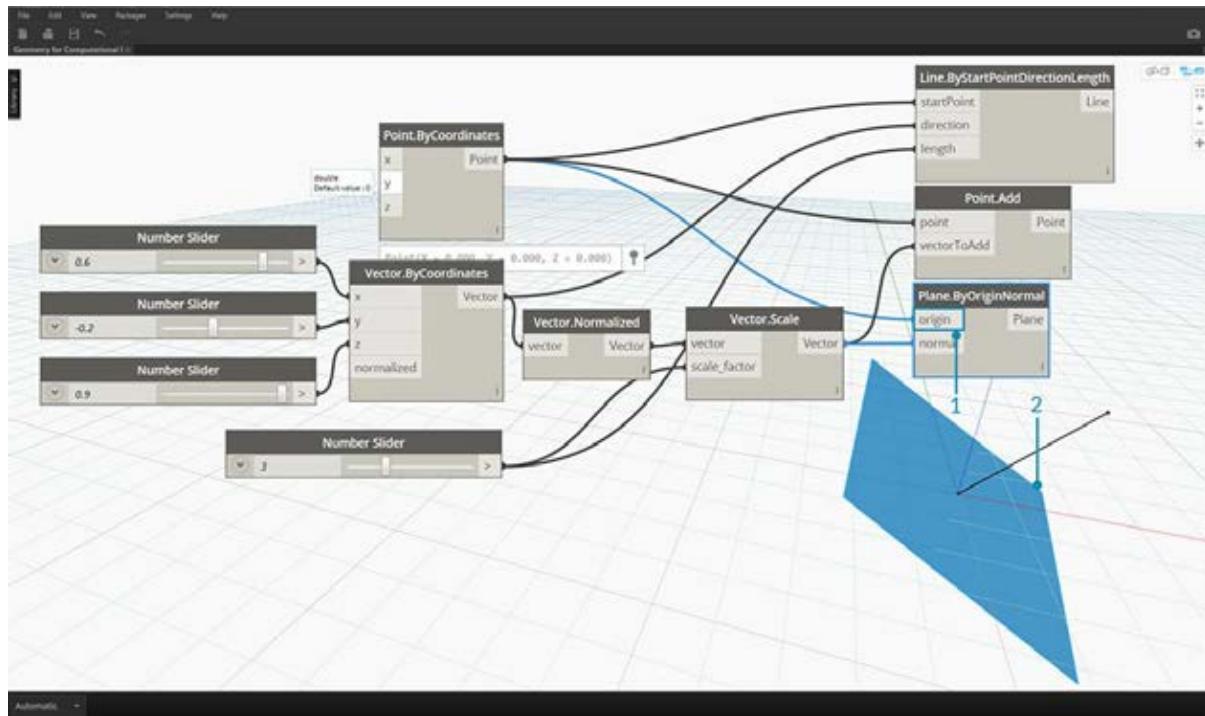
平面は、2 次元の抽象的な Helper です。より厳密に定義すると、平面とは 2 つの方向に無限に延びる概念的に「平らな」面のことです。平面は、通常、小さな長方形として基準点の近くにレンダリングされます。



ここで、「基準点とは、CAD ソフトウェアでモデルを作成する場合に使用する、座標系に関する用語ではないだろうか」と思いつく人がいるかもしれません。

そのとおりです。多くのモデル作成ソフトウェアは、構築面(「レベル」)を使用してローカルな 2 次元のコンテキストを定義し、その面上に図面を作成します。XY 平面、XZ 平面、YZ 平面や、北、南、東などの用語の方がなじみがあるかもしれません。これらはすべて、無限の平らなコンテキストを定義する平面です。平面に厚みはありませんが、方向を表す場合に役立ちます。平面には、基準

点、X 方向、Y 方向、Z(上下)方向があります。

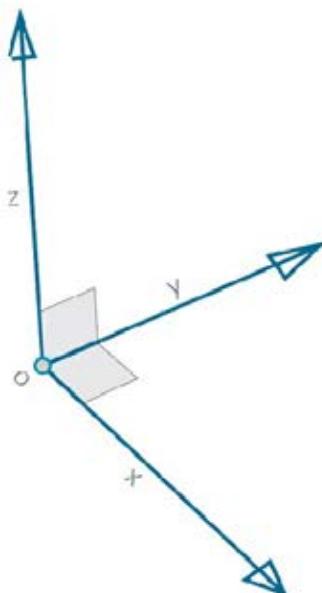


1. 平面は抽象的な概念ですが、平面には基準点があるため、空間内で平面の場所を特定することができます。
2. Dynamo では、平面は背景プレビューにレンダリングされます。この画像に付属しているサンプル ファイルをダウンロードしてください(右クリックして[名前を付けてリンク先を保存...]を選択): [Geometry for Computational Design - Planes.dyn](#)。すべてのサンプルファイルの一覧については、付録を参照してください。

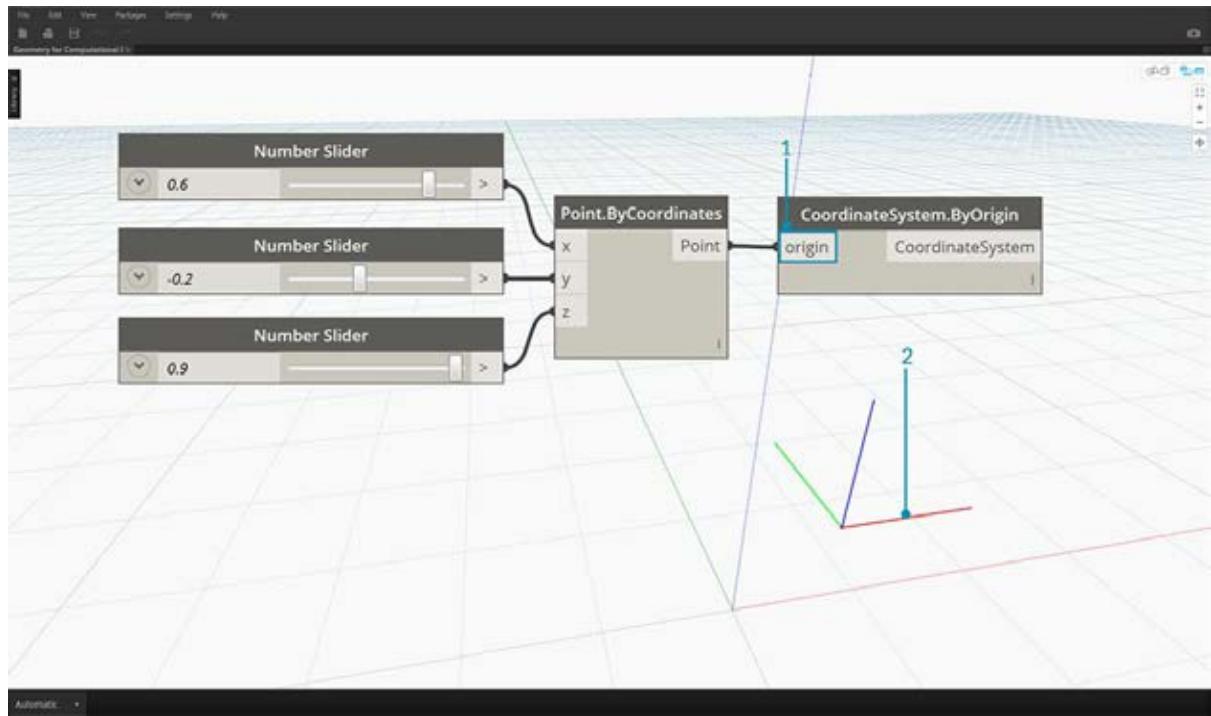
## 座標系の概要

平面を理解できれば、座標系もすぐに理解することができます。平面のすべての要素は、標準的なユークリッド座標系または XYZ 座標系の要素と同じです。

ただし、円柱や球体など、他の座標系もあります。平面の要素は、これらの座標系の要素とは異なります。他のジオメトリ タイプに座標系を適用し、そのジオメトリ上の位置を定義することもできます。これについては、これ以降のセクションで説明します。



別の座標系(円柱、球体)に関する説明をここに追加

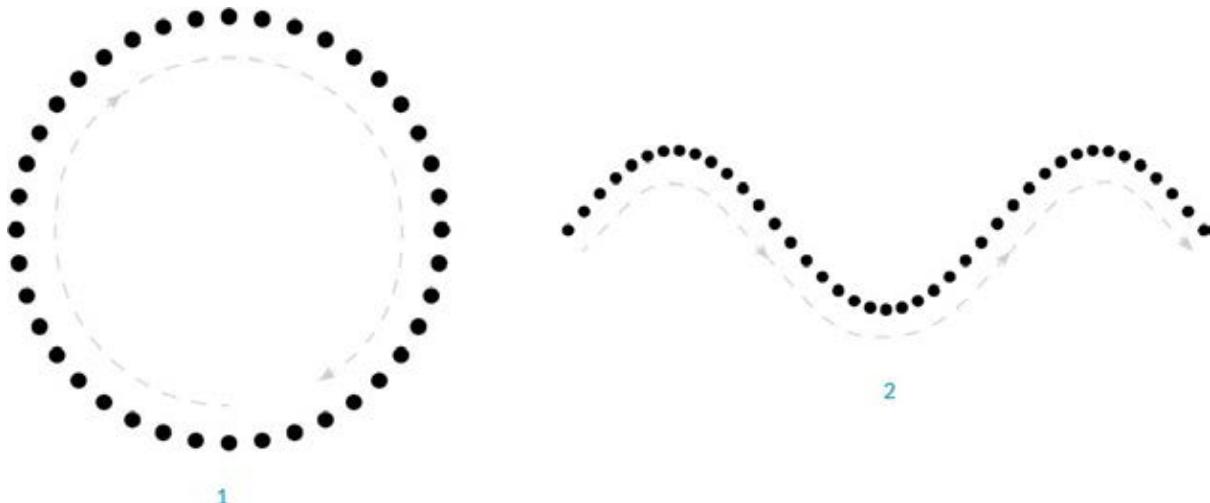


1. 座標系は抽象的な概念ですが、座標系には基準点があるため、空間内で座標系の場所を特定することができます。
2. Dynamo の座標系は、点(基準点)と 3 つの軸を定義する線分(X は赤、Y は緑、Z は青で表示)として背景プレビューにレンダリングされます。この画像に付属しているサンプル ファイルをダウンロードしてください(右クリックして[名前を付けてリンク先を保存...]を選択): [Geometry for Computational Design - Coordinate System.dyn](#)。すべてのサンプルファイルの一覧については、付録を参照してください。

# 点

## 点

ジオメトリがモデルの言語だと仮定すると、点はアルファベットにあたります。点は、点以外のすべてのジオメトリを作成するための基礎になります。たとえば、1 本の曲線を作成するには、少なくとも 2 つの点が必要です。また、1 つのポリゴンまたはメッシュ面を作成するには、少なくとも 3 つの点が必要です。正弦関数を使用して点群の位置、順序、関係を定義すると、円や曲線など、高次元のジオメトリを定義することができます。



1.  $x=r*\cos(t)$  関数と  $y=r*\sin(t)$  関数を使用すると、円が作成されます。
2.  $x=(t)$  関数と  $y=r*\sin(t)$  関数を使用すると、正弦曲線が作成されます。

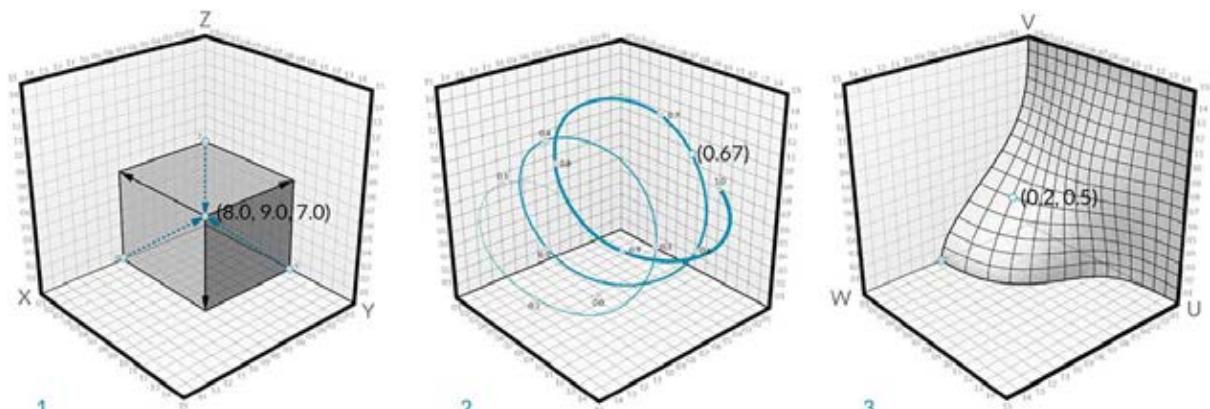
## 点の概要

点は、座標と呼ばれる 1 つまたは複数の値によって定義されます。点を定義するために必要な座標値の数は、その点が存在する座標系やコンテキストによって異なります。Dynamo で使用される最も一般的な種類の点は、3 次元のワールド座標系内に存在します。それらの点には、[X,Y,Z] の 3 つの座標があります。



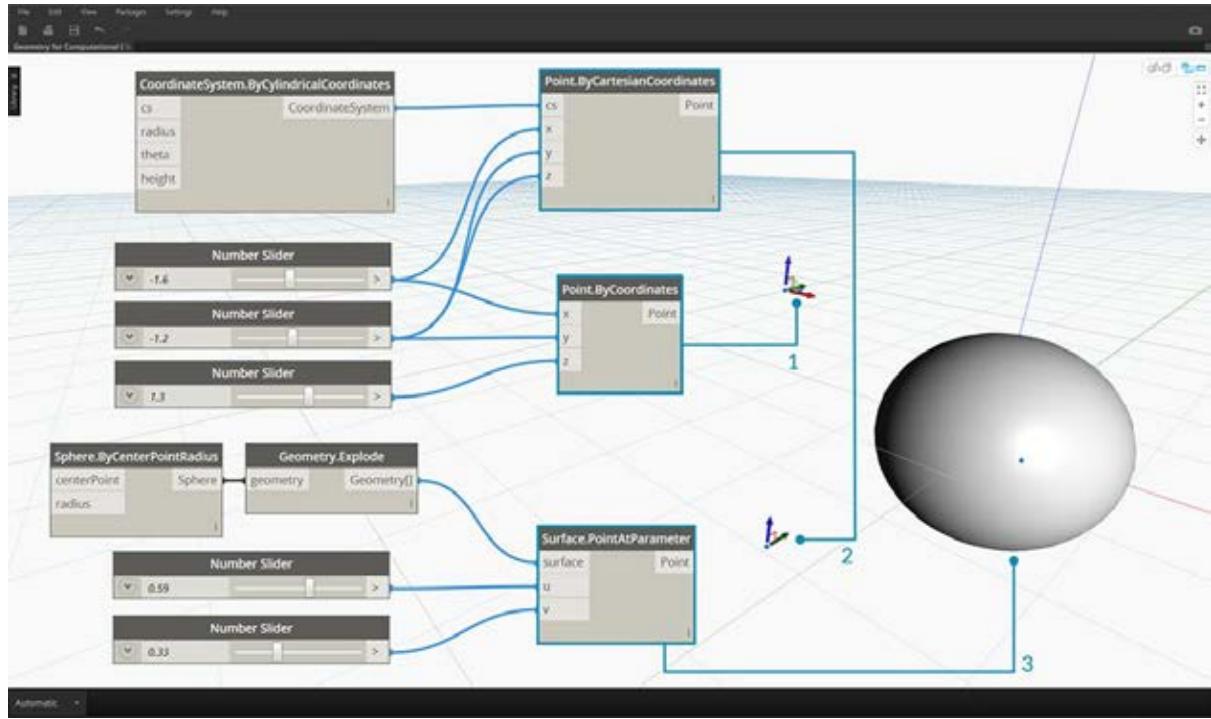
## 座標としての点

点は、2 次元の座標系内にも存在することができます。通常は、使用する空間に応じて異なる文字表記を使用します。この手引では、平面上では[X,Y]を使用し、サーフェス上では[U,V]を使用します。



1. ユークリッド座標系上の点: [X,Y,Z]
2. 曲線パラメータの座標系上の点: [t]
3. サーフェス パラメータの座標系上の点: [U,V]

直感的に理解できないかもしれません、曲線とサーフェスのパラメータは連続的で、指定されたジオメトリのエッジの外側にも存在します。パラメタ空間を定義する形状は3次元のワールド座標系内に存在しているため、パラメタ座標をいつでも「ワールド」座標に変換することができます。たとえば、サーフェス上の点[0.2, 0.5]は、ワールド座標系の点[1.8, 2.0, 4.1]と同じです。

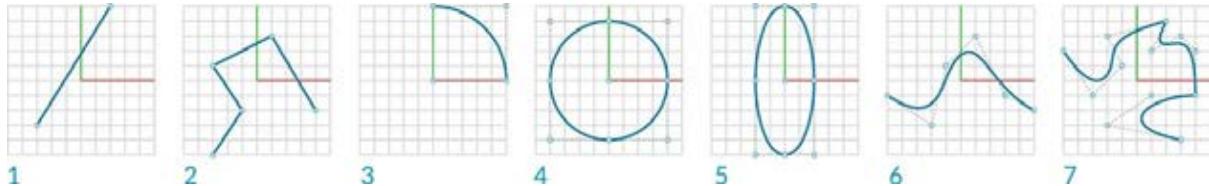


1. ワールド XYZ 座標とみなされる座標で表された点
2. 指定された座標系(円柱座標)で表された点
3. サーフェス上の UV 座標で表された点 この画像に付属しているサンプル ファイルをダウンロードしてください(右クリックして[名前を付けてリンク先を保存]を選択): [Geometry for Computational Design - Points.dyn](#)。すべてのサンプル ファイルの一覧については、付録を参照してください。

# 曲線

## 曲線

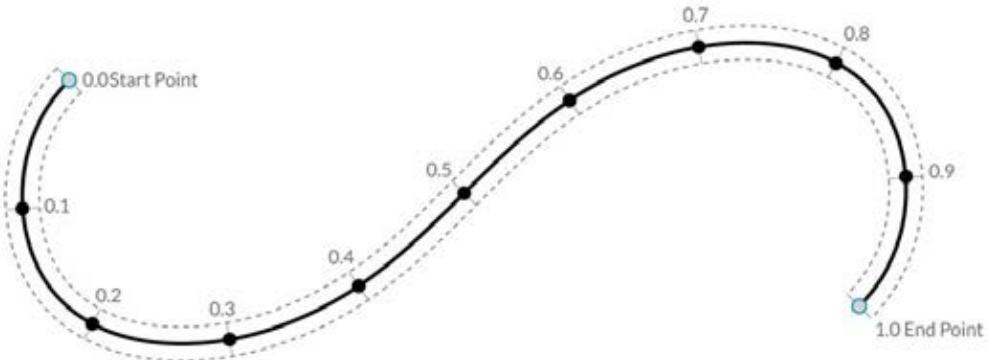
ジオメトリデータタイプには複数のタイプがありますが、最初に曲線について説明します。曲線には、曲がっているか、まっすぐか、長いか、短いかなど、その形状を表す一連のプロパティが存在します。線分からスプライン曲線まで、すべての曲線タイプは点という構成要素で定義されることに注意してください。



1. 線分
2. ポリライン
3. 円弧
4. 円
5. 橢円
6. NURBS 曲線
7. ポリカーブ

## 曲線の概要

曲線という用語は通常、すべての曲がった形状を指します(直線を含みます)。大文字の「C」で始まる「Curve」は、直線、円、スプライン曲線など、すべての形状タイプの親カテゴリです。より厳密に定義すると、曲線とは、 $(x=-1.26*t, y=t)$ などの単純な関数から微積分を使用する複雑な関数まで、さまざまな関数の集合に「t」を代入して指定することができるすべての点のことです。操作する曲線の種類に関係なく、この「t」というパラメータが評価対象のプロパティになります。また、形状の外観にかかわらず、すべての曲線には開始点と終了点があり、この開始点と終了点が、曲線の作成に使用される  $t$  の最小値と最大値に対応します。これを理解すると、曲線の方向についても理解できます。



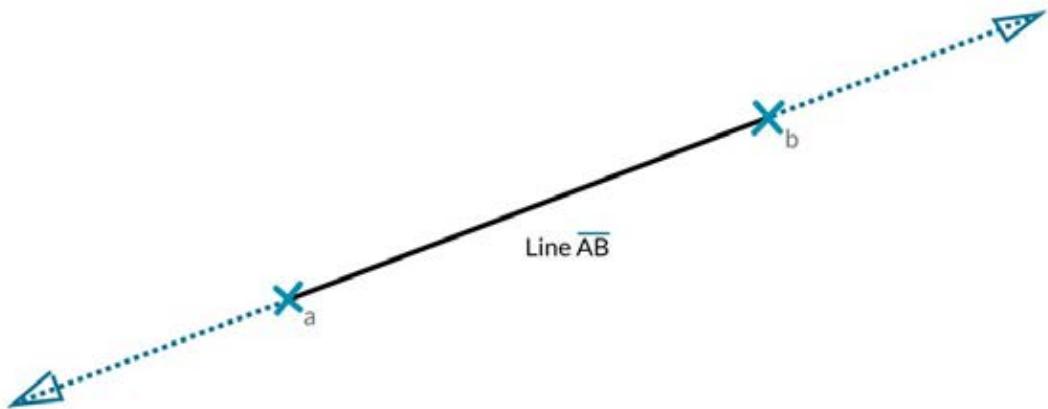
Dynamo では、曲線に対する  $t$  の値の範囲は 0.0 ~ 1.0 になります。これは、覚えておく必要があります。

すべての曲線には、曲線の記述や解析で使用されるさまざまなプロパティと性質があります。開始点と終了点との間の距離がゼロの場合は、「閉じた」曲線になります。また、すべての曲線には複数の制御点があります。これらの点がすべて一つの平面上に配置されている場合、その曲線は「平らな」曲線になります。一部のプロパティは曲線全体に適用されますが、曲線上の特定の点にのみ適用されるプロパティもあります。たとえば平面性はグローバルプロパティですが、特定の  $t$  値における接線ベクトルはローカルプロパ

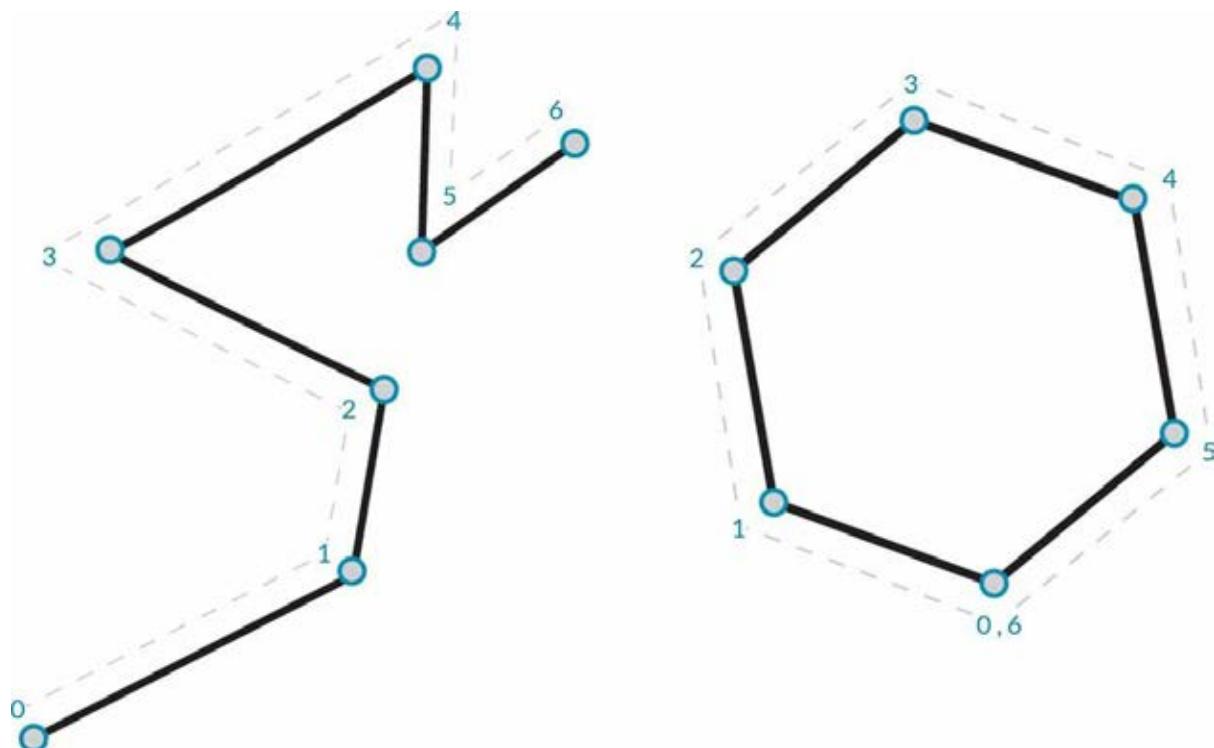
テイです。

## 線分

線分は、最も簡単な形状の曲線です。直線は曲線には見えないかもしれません、曲率がゼロであるというだけで、実際には曲線です。線分を作成する方法はいくつかあります。最も直感的な方法は、点 A から点 B までの線分を作成する方法です。この場合、線分 AB の形状が点間に描画されますが、数学的には、直線が両方向に無限に延びている状態になります。

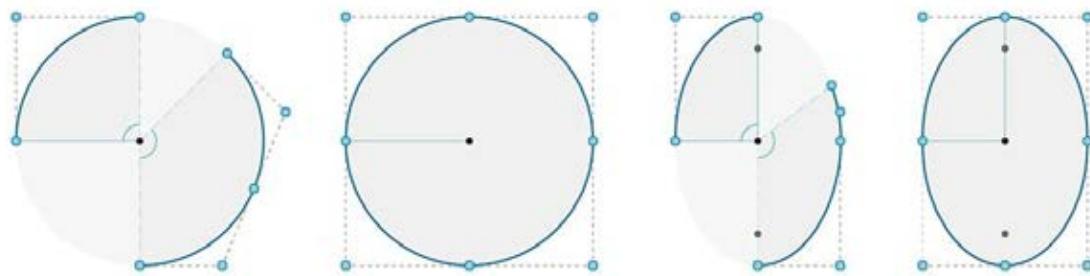


2つの線分を接続すると、ポリラインが作成されます。ここで、制御点について簡単に説明します。どの制御点の位置を編集しても、ポリラインの形状が変化します。ポリラインが閉じている場合は、ポリゴンが作成されます。ポリゴンの辺の長さがすべて同じである場合、このポリゴンは正多角形になります。



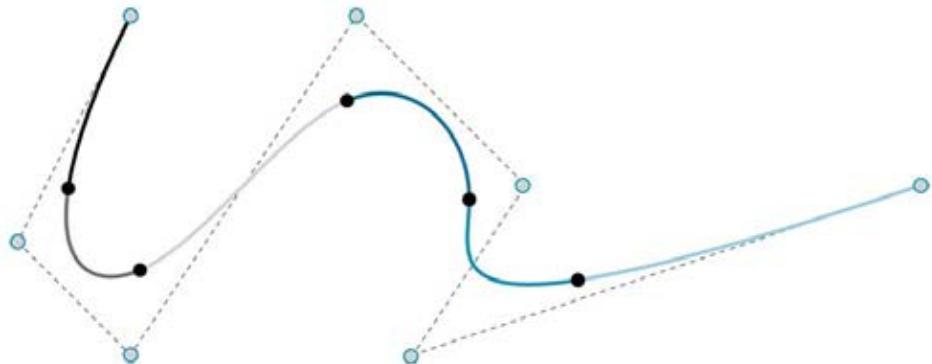
円弧、円、橢円弧、橢円

ここからは、形状を定義するための複雑なパラメトリック関数を見ていきます。これまで線分について説明しましたが、ここでは 1 つまたは 2 つの半径を設定することにより、円弧、円、橢円弧、橢円を作成します。円弧と円または橢円の違いは、形状が閉じているかどうかだけです。



## NURBS 曲線とポリカーブ

NURBS(非一様有理スプライン)は、単純な 2 次元の線分、円、円弧、長方形の形状から、複雑な 3 次元フリーフォームの有機的な曲線まで、あらゆる形状を正確にモデル化することができる数学的表現です。その柔軟性(比較的少ない制御点で、次数の設定に基づいたスムーズな補間が可能)と精度(堅牢な数学演算による形状指定)により、イラストレーションやアニメーションから製造にわたる幅広いプロセスで使用することができます。

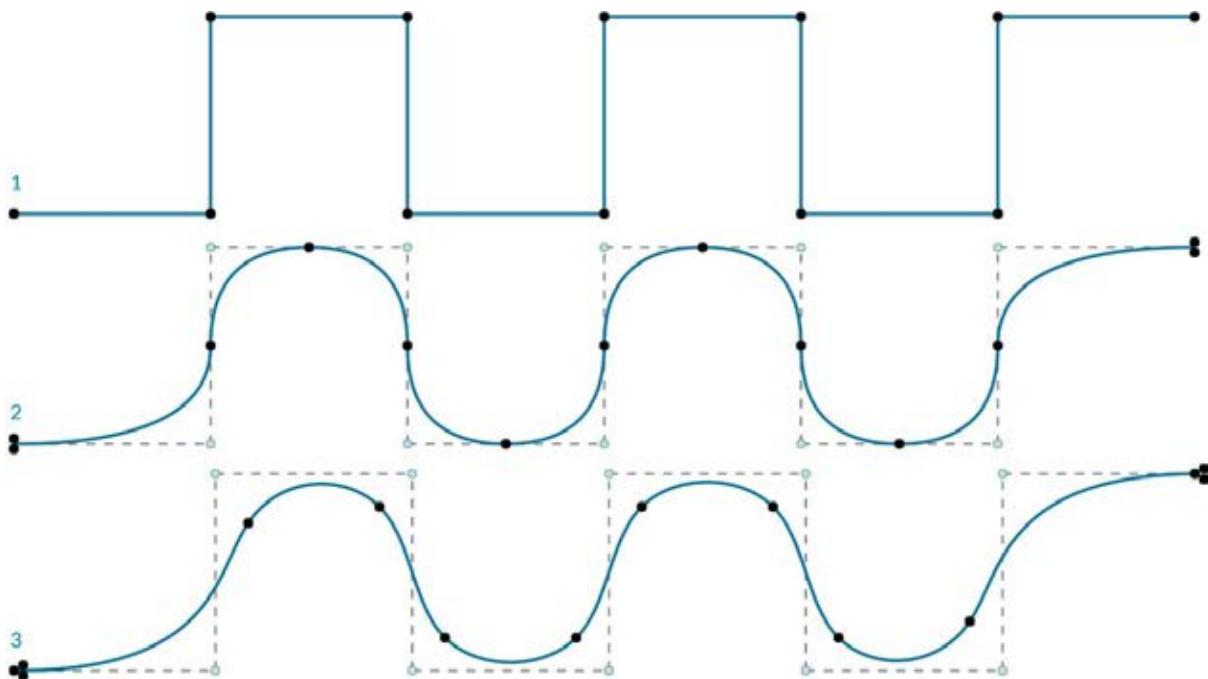


**次数:** 曲線の次数により、制御点が曲線に与える影響力の範囲が決まります。次数が大きいほど、影響力の範囲も大きくなります。次数は正の整数です。この数値は通常 1、2、3、または 5 ですが、任意の正の整数にすることができます。NURBS 線分とポリラインの次数は、通常は 1 です。ほとんどのフリーフォーム曲線の次数は、3 または 5 です。

**制御点:** 制御点は、次数 + 1 個以上の点を含むリストです。NURBS 曲線の形状を変更するための最も簡単な方法は、その制御点を移動する方法です。

**ウェイト:** 制御点には、ウェイトと呼ばれる数値が関連付けられています。ウェイトは、通常は正の数値です。曲線の制御点のすべてのウェイトの値が同じである場合(通常は 1)、その曲線は非有理曲線と呼ばれます。それ以外の場合は、有理曲線と呼ばれます。ほとんどの NURBS 曲線は、非有理曲線です。

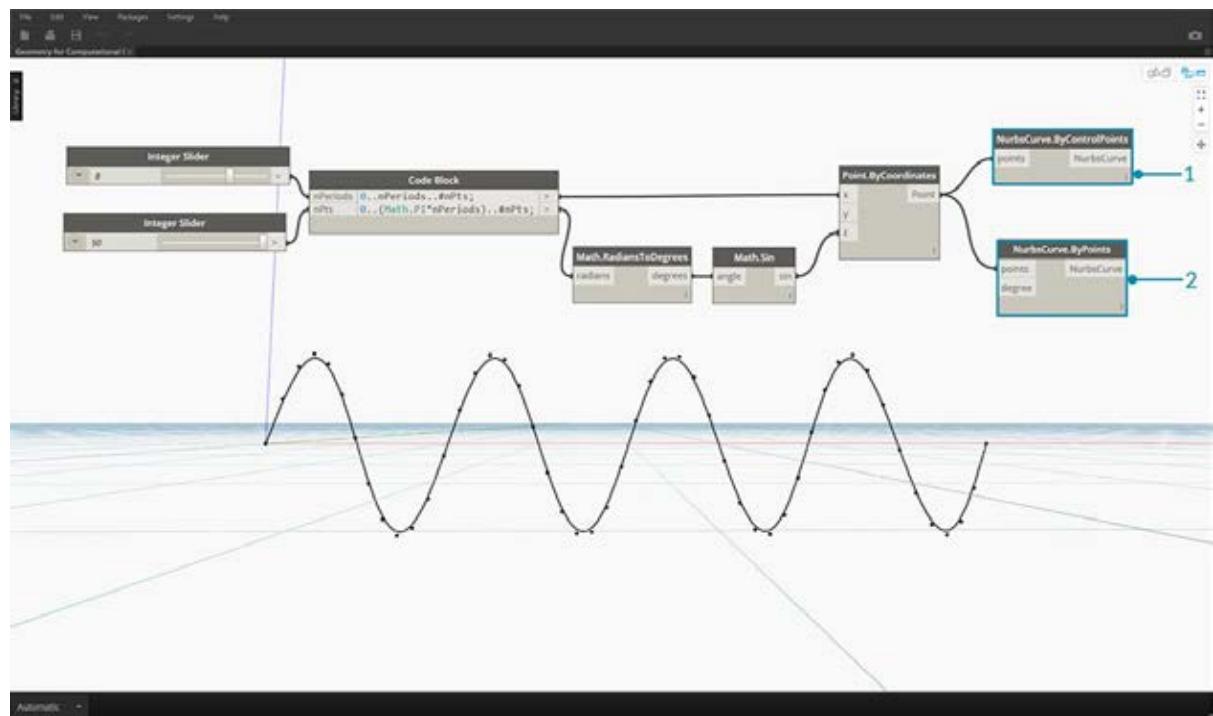
**ノット:** ノットは、N を制御点の数としたとき、「次数 + N - 1」個の数値のリストとして表されます。ノットはウェイトとともに使用され、作成される曲線上の制御点の影響力をコントロールします。ノットは、曲線上の特定の点でねじれを作成する場合などに使用します。



1. 次数 = 1
2. 次数 = 2
3. 次数 = 3

次数の値が大きいほど、作成される曲線を補間するための制御点の数が多くなります。

次に、NURBS 曲線を作成する 2 つの方法を Dynamo で使用して正弦曲線を作成し、結果を比較してみましょう。



1. *NurbsCurve.ByControlPoints* ノードは、点のリストを制御点として使用します。
2. *NurbsCurve.ByPoints* ノードは、点のリストを使用して曲線を描画します。この画像に付属しているサンプル ファイルをダウンロードしてください(右クリックして[名前を付けてリンク先を保存...]を選択): [Geometry for Computational Design - Curves.dyn](#)。すべてのサンプルファイルの一覧については、付録を参照してください。

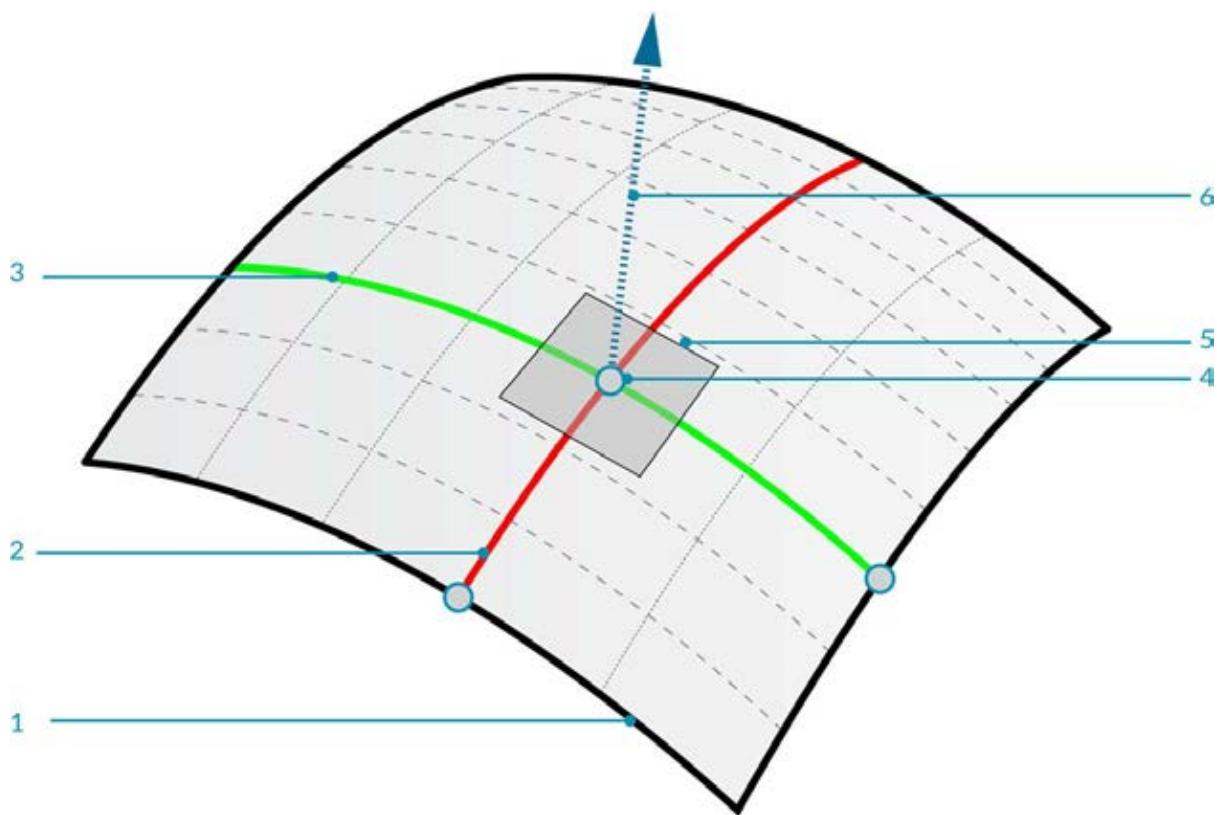
# サーフェス

## サーフェス

ここまで、モデル内で曲線を使用するケースを見てきました。ここからは、モデル内でサーフェスを使用するケースを見ていきます。サーフェスを使用することにより、オブジェクトを3次元の世界で表現できるようになります。曲線は常に平らというわけではありません。曲線は3次元のオブジェクトですが、曲線が定義する空間は必ず1次元になります。サーフェスを定義すると、次元をもう1つ増やし、別のプロパティの集合を他のモデリング操作で使用できるようになります。

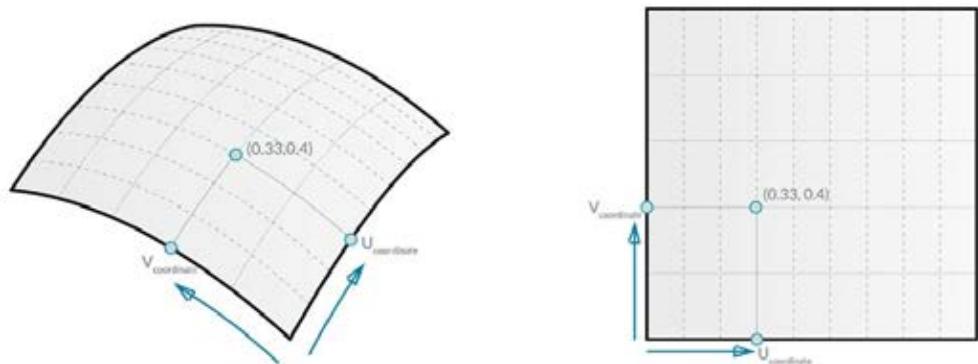
### サーフェスの概要

サーフェスは、1つの関数と2つのパラメータによって定義される数学的な形状です。曲線を定義する  $t$  ではなく、サーフェスでは  $U$  と  $V$  を使用してパラメータ空間を定義します。そのため、このタイプのジオメトリを使用すると、より多くのジオメトリデータを描画することができます。たとえば、曲線には法線ベクトルと接平面があり、接平面は曲線に沿って回転させることができます。一方、サーフェスの法線ベクトルと接平面は、方向が固定されています。



1. サーフェス
2. U アイソカーブ
3. V アイソカーブ
4. UV 座標
5. 接平面
6. 法線ベクトル

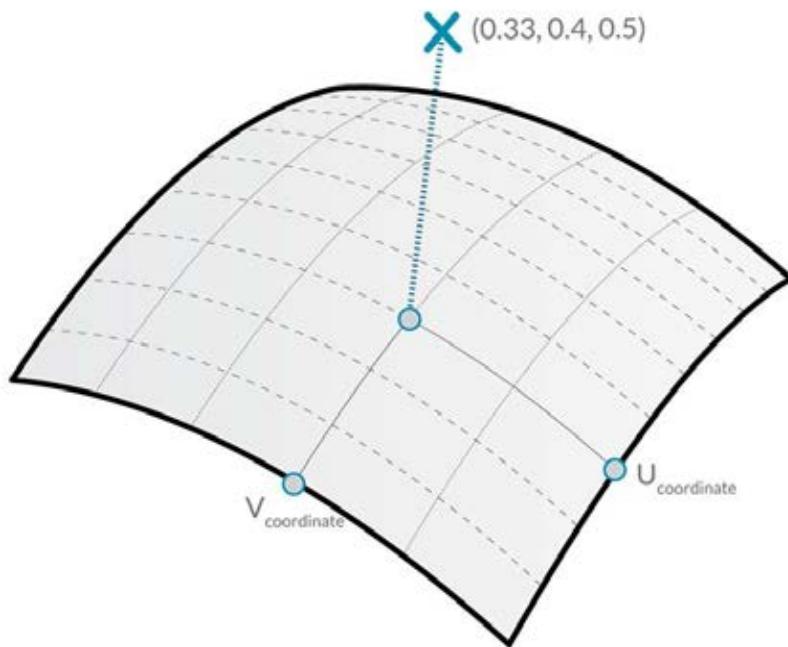
**サーフェスの範囲:** サーフェスの範囲は、そのサーフェス上の3次元の点を指定する( $U, V$ )パラメータの範囲として定義されます。各次元( $U$ または $V$ )の範囲は、通常2つの数値( $U$ の最小値から $U$ の最大値までの数値と、 $V$ の最小値から $V$ の最大値までの数値)で表されます。



サーフェスの形状が「長方形」に見えない場合や、サーフェスのローカルのアイソカーブの密度が高い場合や低い場合がありますが、サーフェスの範囲によって定義される「空間」は常に 2 次元になります。Dynamo におけるサーフェスの範囲は、U 方向と V 方向の両方向で、最小値 0.0 から最大値 1.0 までの範囲内で定義する必要があります。ただし、平らなサーフェスやトリム サーフェスについては、範囲が異なる場合があります。

**アイソカーブ(アイソパラメトリック曲線):** サーフェス上の U 定数値または V 定数値と、対応するもう一方の U 方向または V 方向の値の範囲によって定義される曲線です。

**UV 座標:** U と V (場合によってはさらに W)を使用して定義される、UV パラメータ空間内の点です。

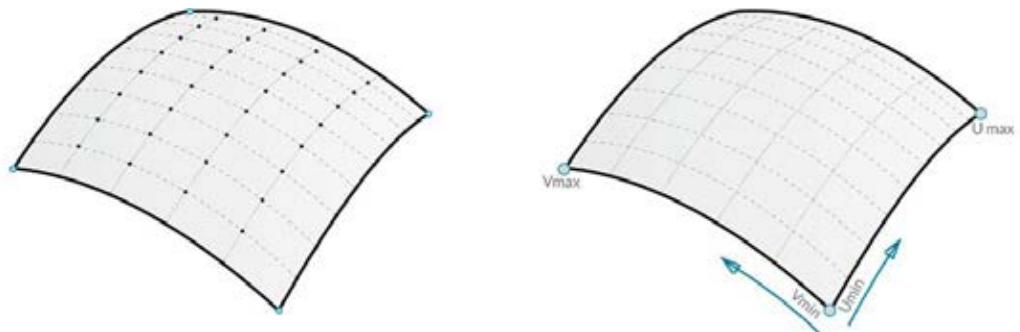


**接平面:** 特定の UV 座標において、U アイソカーブと V アイソカーブの両方に接する平面です。

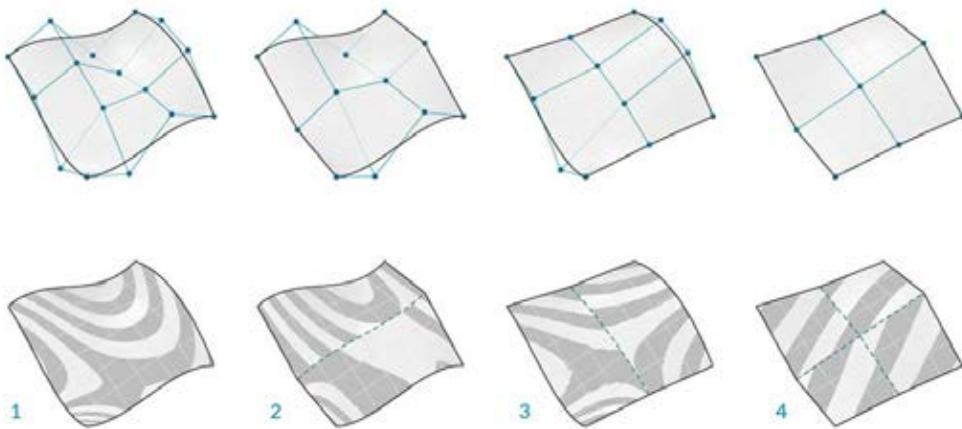
**法線ベクトル:** 接平面に対して相対的に「上」方向を定義するベクトルです。

### NURBS サーフェス

NURBS サーフェスは、NURBS 曲線によく似ています。NURBS サーフェスは、2 つの方向に向かう NURBS 曲線のグリッドとして考えることができます。NURBS サーフェスの形状は、制御点の数と、そのサーフェスの U 方向および V 方向の次数によって定義されます。制御点、ウェイト、次元を使用して、形状、法線、接線、曲率などのプロパティを計算する場合にも、同じアルゴリズムが使用されます。



NURBS サーフェスの場合、ジオメトリによって 2 つの方向が暗黙的に定義されます。これは、表示される形状に関係なく、NURBS サーフェスは制御点から構成される長方形のグリッドであるためです。これらの方向は、多くの場合、ワールド座標系に対して任意の方向が相対的に設定されています。このチュートリアルでは、これらの方向を頻繁に使用し、サーフェスに基づいてモデルの解析や他のジオメトリの生成を行います。

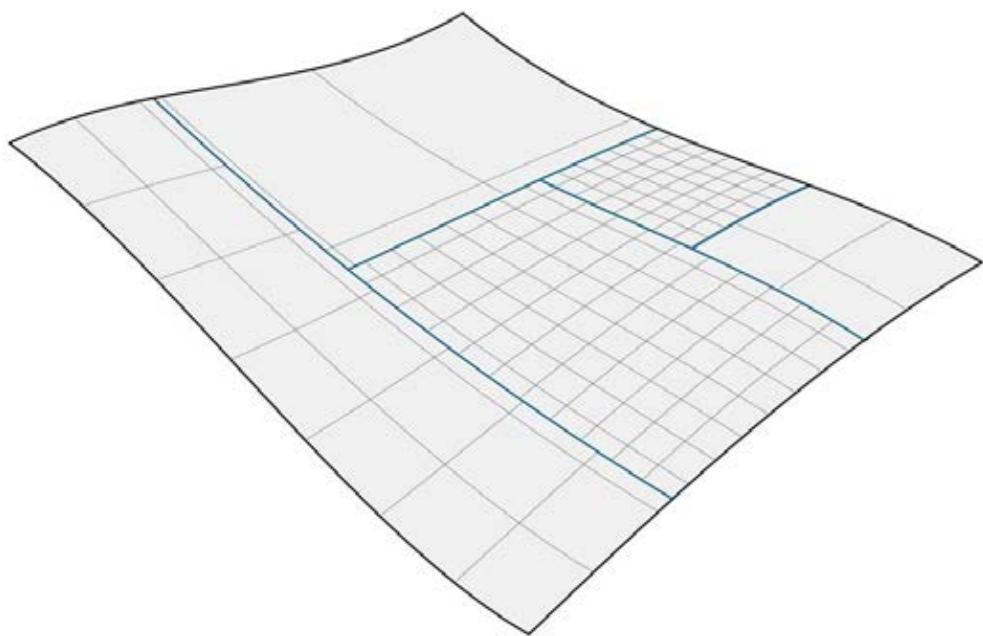


1. Degree (U,V) = (3,3)
2. Degree (U,V) = (3,1)
3. Degree (U,V) = (1,2)
4. Degree (U,V) = (1,1)

## ポリサーフェス

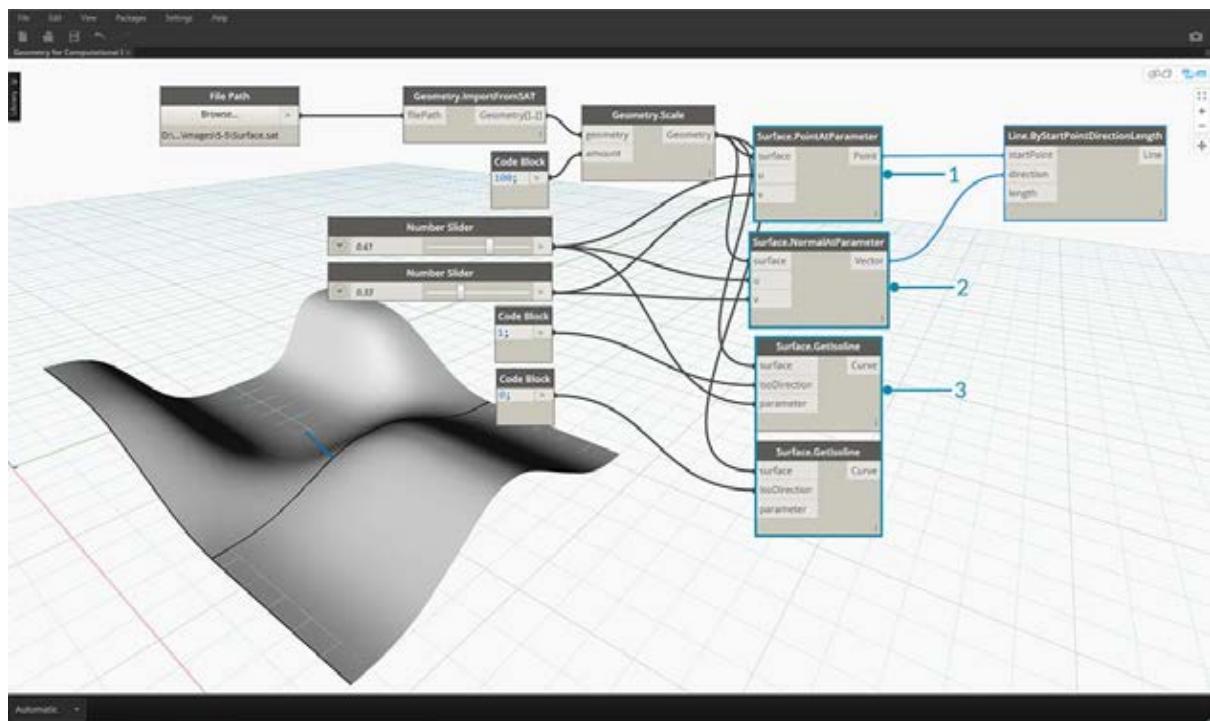
ポリサーフェスは、エッジで結合されているサーフェスによって構成されます。ポリサーフェスのトポロジを経由して結合された形状間を移動できるため、ポリサーフェスは 2 次元の UV 定義よりも多くの機能を持っています。

一般的にトポロジとは、パートの結合や関連についての概念を指しますが、Dynamo におけるトポロジとは、一種のジオメトリであります。具体的には、トポロジは、サーフェス、ポリサーフェス、ソリッドの親カテゴリにあたります。



このような方法でサーフェスを結合することにより、複雑な形状を作成したり、継ぎ目の形状の詳細を定義することができます。このように結合されたサーフェスは、パッチと呼ばれる場合もあります。ポリサーフェスのエッジに対して、フィレット操作や面取り操作を簡単に適用することができます。

Dynamo のパラメータにサーフェスを読み込んで評価し、どのような情報を抽出できるかを確認してみましょう。



1. *Surface.PointAtParameter* ノードは、指定された UV 座標における点を返します。
2. *Surface.NormalAtParameter* ノードは、指定された UV 座標における法線ベクトルを返します。
3. *Surface.GetIsoline* ノードは、U 座標または V 座標におけるアイソパラメトリック曲線を返します。入力が isoDirection であることに注意してください。この画像に付属しているサンプル ファイルをダウンロードしてください(右クリックして[名前を付けてリンク先を保存]を選択)。すべてのサンプルファイルの一覧については、付録を参照してください。

い。

4. [Geometry for Computational Design - Surfaces.dyn](#)
5. [Surface.sat](#)

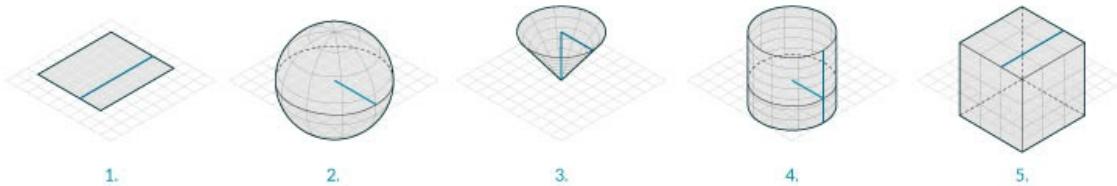
# ソリッド

## ソリッド

単一のサーフェスから作成することのできない複雑なモデルを構築する場合や、明示的な体積を定義する場合は、ソリッド(およびポリサーフェス)を使用する必要があります。単純な立方体でさえ、全部で 6 つのサーフェスが必要になる複雑な構造をしています。ソリッドには、サーフェスには存在しない 2 つの重要な概念があります。それは、高度な位相幾何学的な概念(面、辺、頂点)と、布尔演算という概念です。

### ソリッドの概要

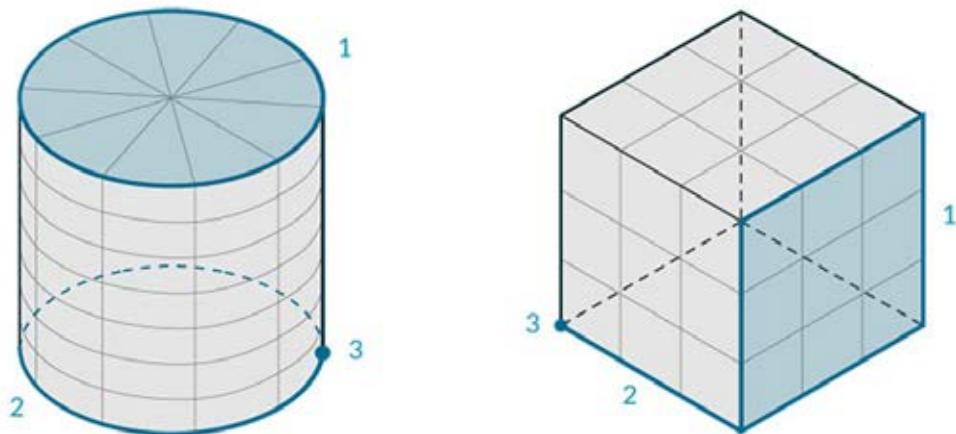
ソリッドは 1 つまたは複数のサーフェスから構成され、「内部」と「外部」を定義する閉じた境界によって体積が定義されます。ソリッドとして認識される条件は、サーフェスの数に関係なく、全体が完全に閉じた形状になっているということです。ソリッドは、サーフェスまたはポリサーフェスを結合して作成することも、ロフト、スイープ、回転などの操作を使用して作成することもできます。球体、立方体、円錐、円柱プリミティブなどもソリッドです。立方体から 1 つまたは複数の面を取り除いた場合、その形状はポリサーフェスとして認識されます。ポリサーフェスのプロパティはソリッドのプロパティと似ていますが、このポリサーフェスはソリッドではありません。



1. 単一のサーフェスで構成される平面は、ソリッドではありません。
2. 単一のサーフェスで構成される球体は、ソリッドです。
3. 2 つの結合されたサーフェスで構成される円錐は、ソリッドです。
4. 3 つの結合されたサーフェスで構成される円柱は、ソリッドです。
5. 6 つの結合されたサーフェスで構成される立方体は、ソリッドです。

## トポロジ

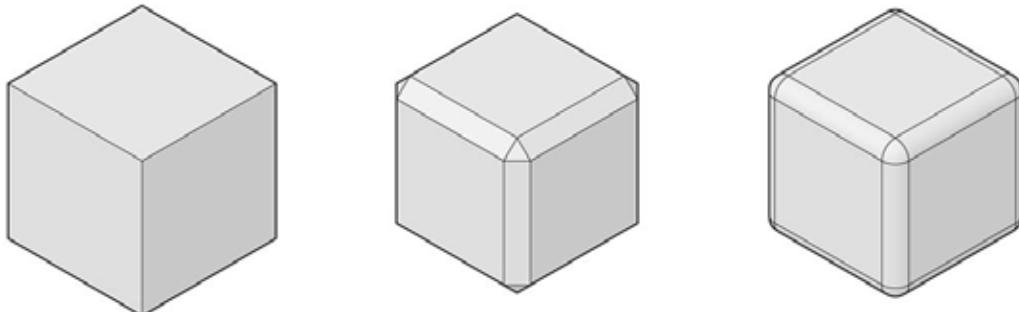
ソリッドは、頂点、辺、面という 3 種類の要素で構成されます。面は、ソリッドを構成するサーフェスです。辺は、隣接する面の接続を定義する曲線です。頂点は、これらの曲線の開始点と終了点です。Topology ノードを使用すると、これらの要素についてクリーを実行することができます。



1. 面
2. 辺
3. 頂点

## 操作

ソリッドの辺をフィレット操作や面取り操作で変更することにより、角のとがりや出っ張りを取り除くことができます。面取り操作を実行すると、2つの面の間に直線的なサーフェスが作成され、フィレット操作を実行すると、2つの面がなめらかに接合されます。



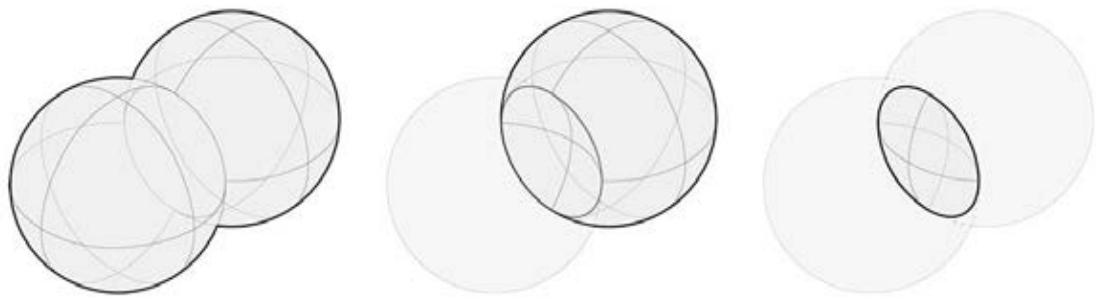
1. ソリッド立方体
2. 面取りされた立方体
3. フィレットされた立方体

## ブール演算

ソリッドのブール演算は、2つ以上のソリッドを組み合わせるための方法です。ブール演算を1回実行すると、実際には次に示す4つの操作が実行されます。

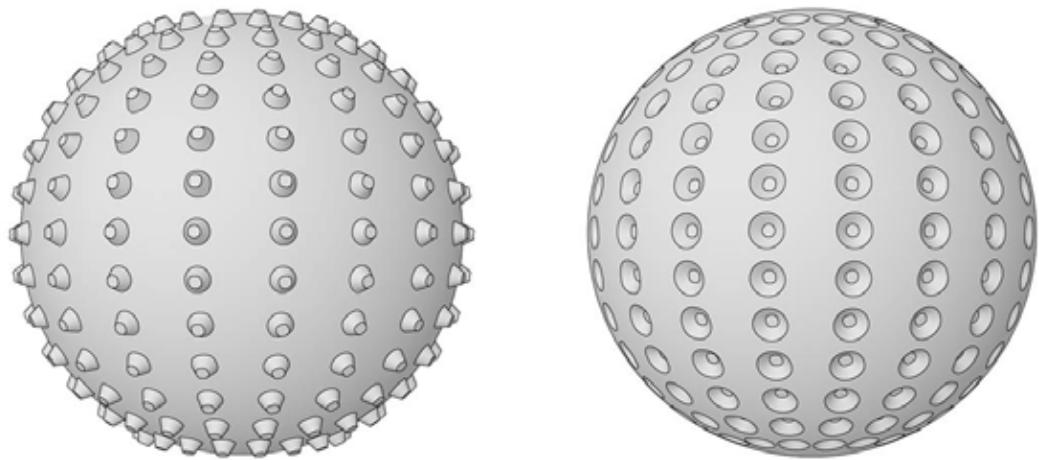
1. 2つ以上のオブジェクトを交差させる。
2. すべてのオブジェクトを交点で分割する。
3. ジオメトリの不要な部分を削除する。
4. すべてのオブジェクトを1つに結合する。

このように、ソリッドのブール演算は、作業時間を節約するための強力な機能です。ソリッドのブール演算には、ジオメトリのどの部分を保持するかを区別する3つの操作があります。



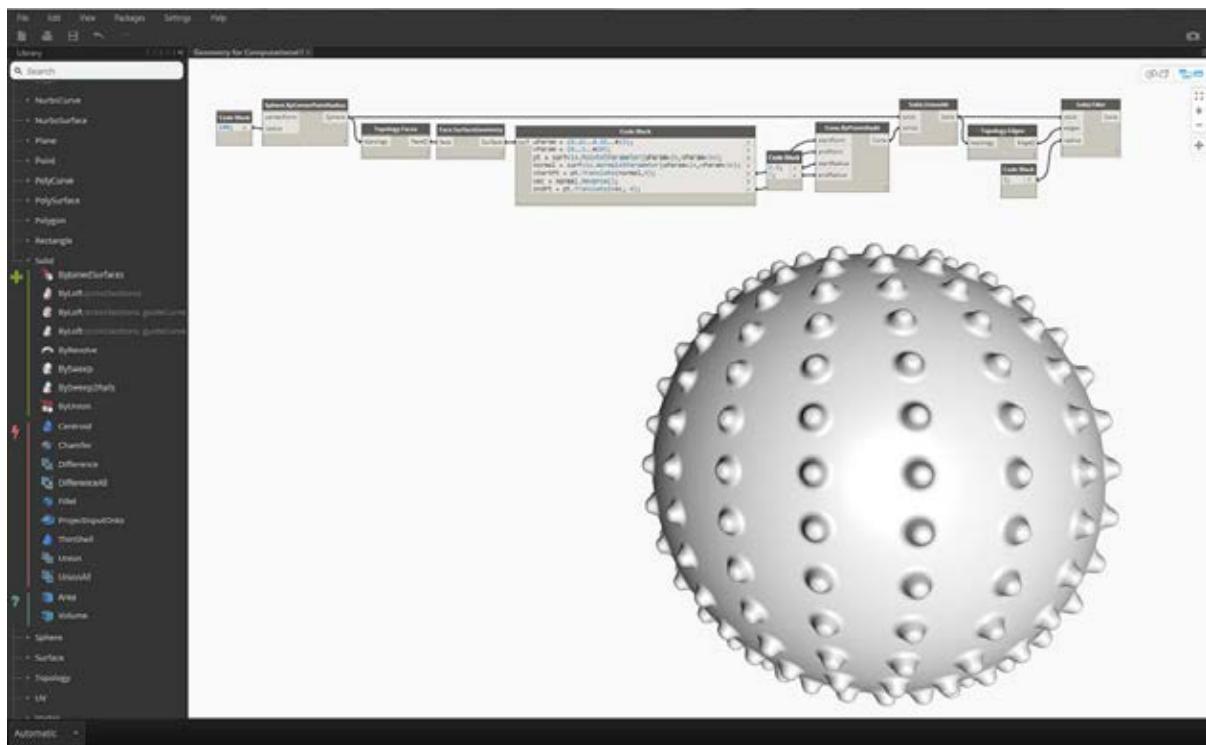
1. 和: ソリッドの重複部分を削除して 1 つのソリッドに結合します。
2. 差: 一方のソリッドから別のソリッドを取り除きます。取り除く側のソリッドは、ツールと呼ばれます。ツールとなるソリッドを切り替えて、逆のボリュームを作成することができます。
3. 積: 2 つのソリッドが交差している部分だけを保持します。

これら 3 つの演算に加えて、Dynamo には、さまざまなソリッドの差演算と和演算を実行するための **Solid.DifferenceAll** ノードと **Solid.UnionAll** ノードが用意されています。



1. **UnionAll** ノードは、球体と外側を向いた円錐の和演算を行います。
2. **DifferenceAll** ノードは、球体と内側を向いた円錐の差演算を行います。

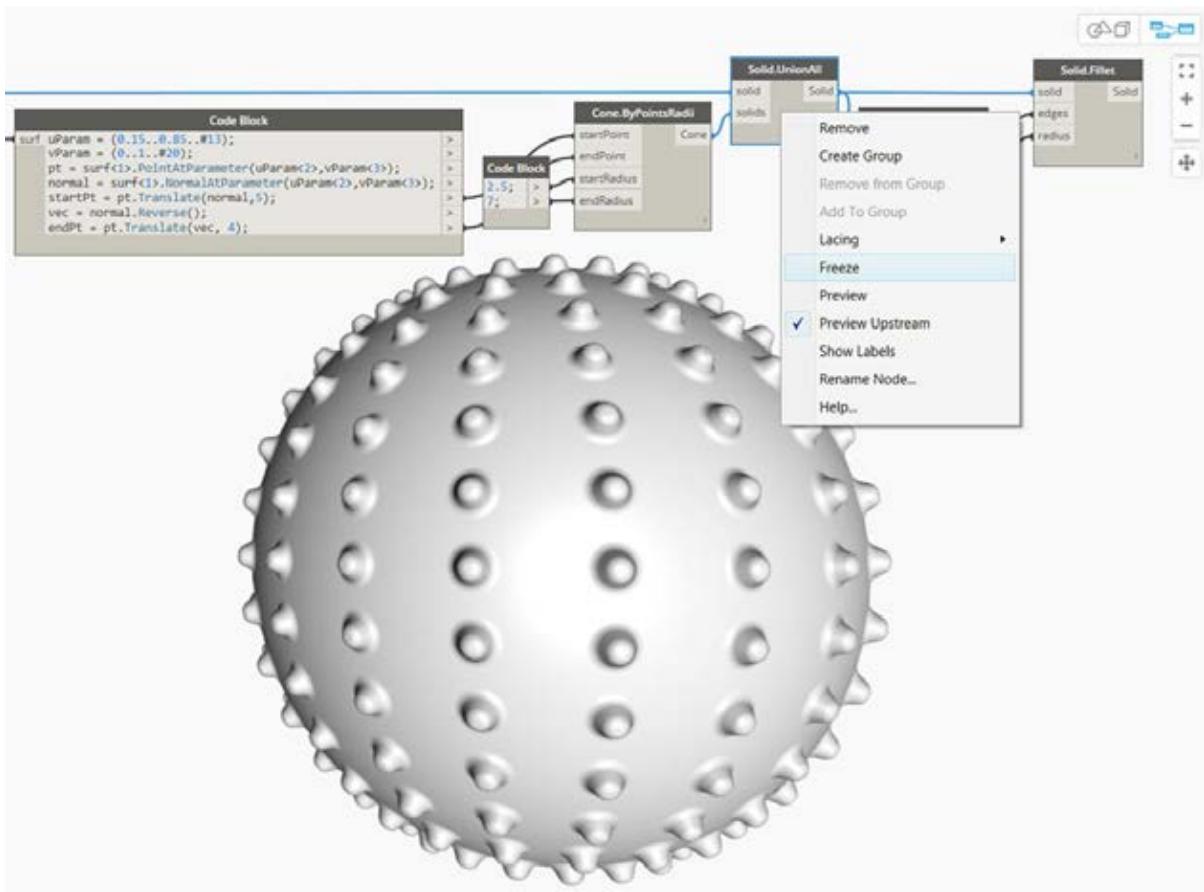
いくつかのブール演算を使用して、スパイク状のボールを作成してみましょう。



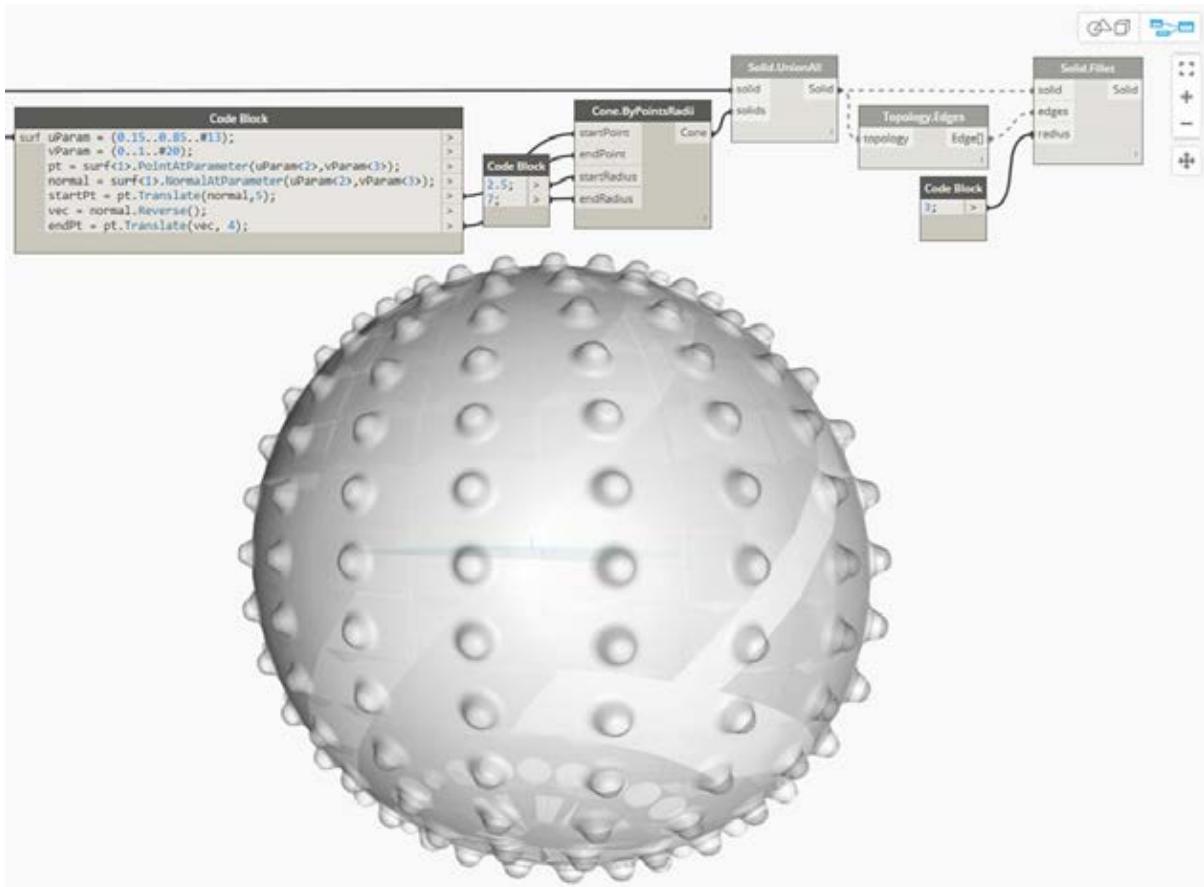
1. **Sphere.ByCenterPointRadius** ノードを使用して、ベースとなるソリッドを作成します。
2. **Topology.Faces** ノードと **Face.SurfaceGeometry** ノードを使用してソリッドの面のクエリーを実行し、サーフェス ジオメトリに変換します。この場合、球体には 1 つの面しかありません。
3. **Cone.ByPointsRadii** ノードで、サーフェス上の点を使用して円錐を作成します。
4. **Solid.UnionAll** ノードを使用して、円錐と球体との和演算を行います。
5. **Topology.Edges** ノードを使用して、新しいソリッドの辺のクエリーを実行します。
6. **Solid.Fillet** ノードを使用して、スパイク状の球体のエッジの面取りを行います。この画像に付属しているサンプル ファイルをダウンロードしてください(右クリックして[名前を付けてリンク先を保存]を選択)。すべてのサンプルファイルの一覧については、付録を参照してください。[Geometry for Computational Design - Solids.dyn](#)

## フリーズ

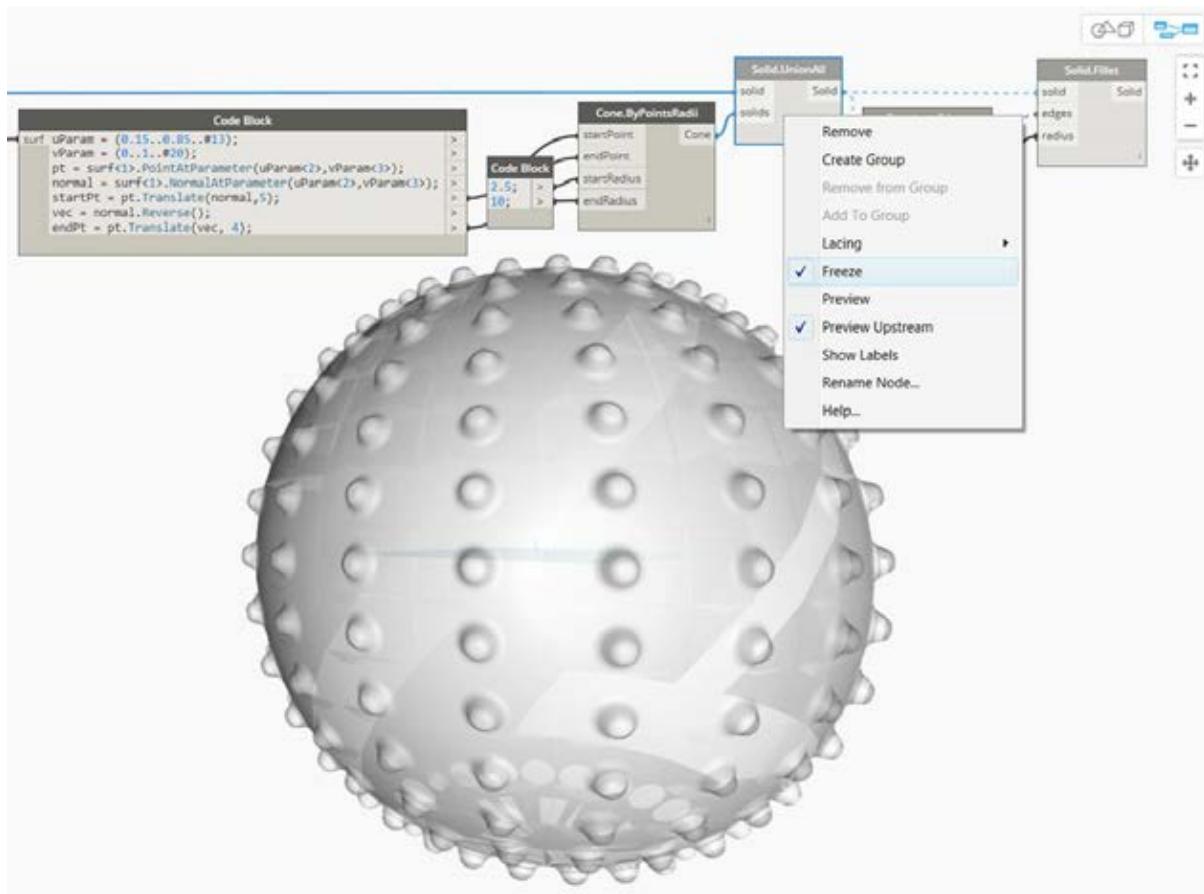
ブール演算は複雑なため、計算に時間がかかります。選択したノードとその影響を受ける下流ノードの実行を中止するには、フリーズ機能を使用します。



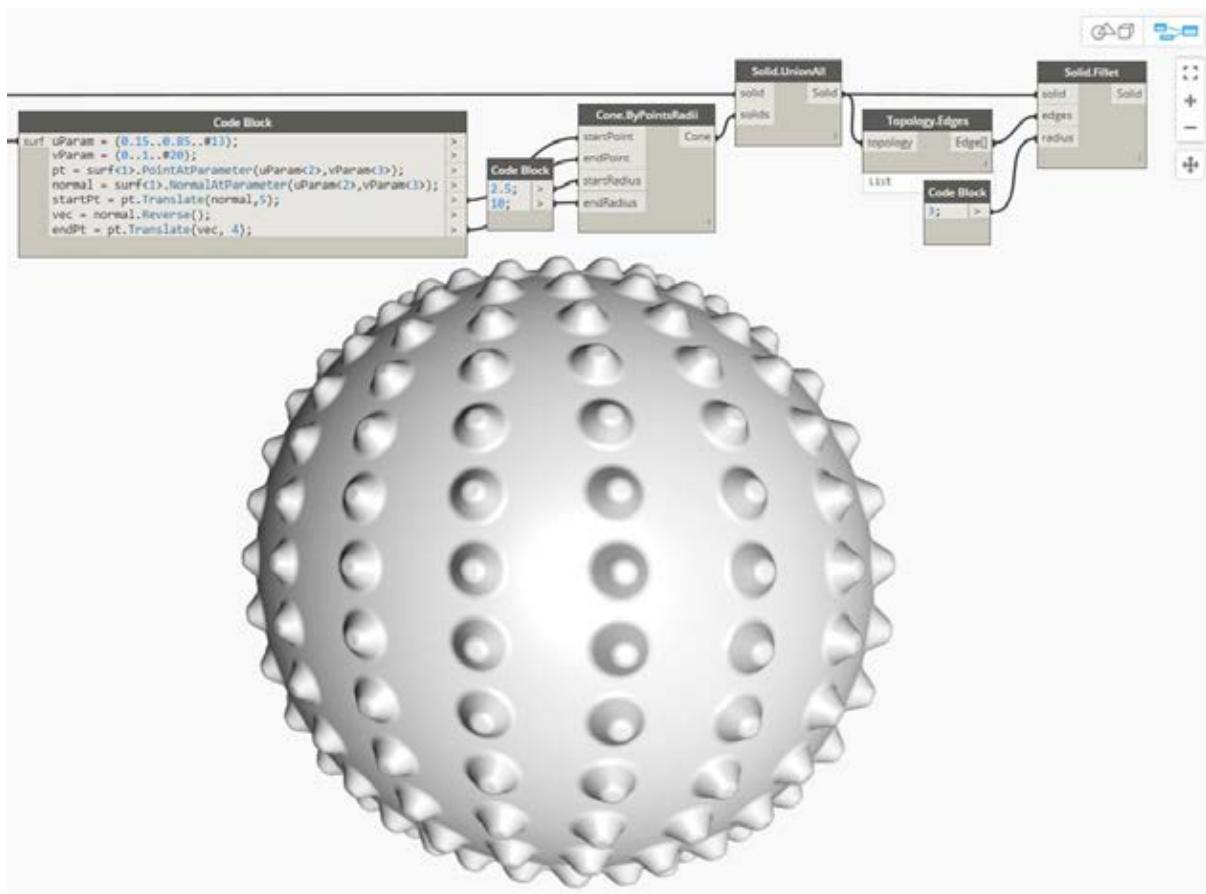
Solid.UnionAll ノードの和演算をフリーズするには、右クリックしてコンテキストメニューを使用します。



選択したノードとすべての下流ノードがライトグレーのゴーストモードでプレビュー表示され、影響を受けるワイヤが破線で表示されます。影響を受けるジオメトリのプレビューも、ゴーストモードになります。これで、 Boolean論理和を計算することなく、上流で値を変更することができます。



ノードのフリーズを解除するには、ノードを右クリックして[フリーズ]の選択を解除します。



影響を受けるすべてのノードとそれに関連するジオメトリのプレビューが更新され、標準プレビュー モードに戻ります。

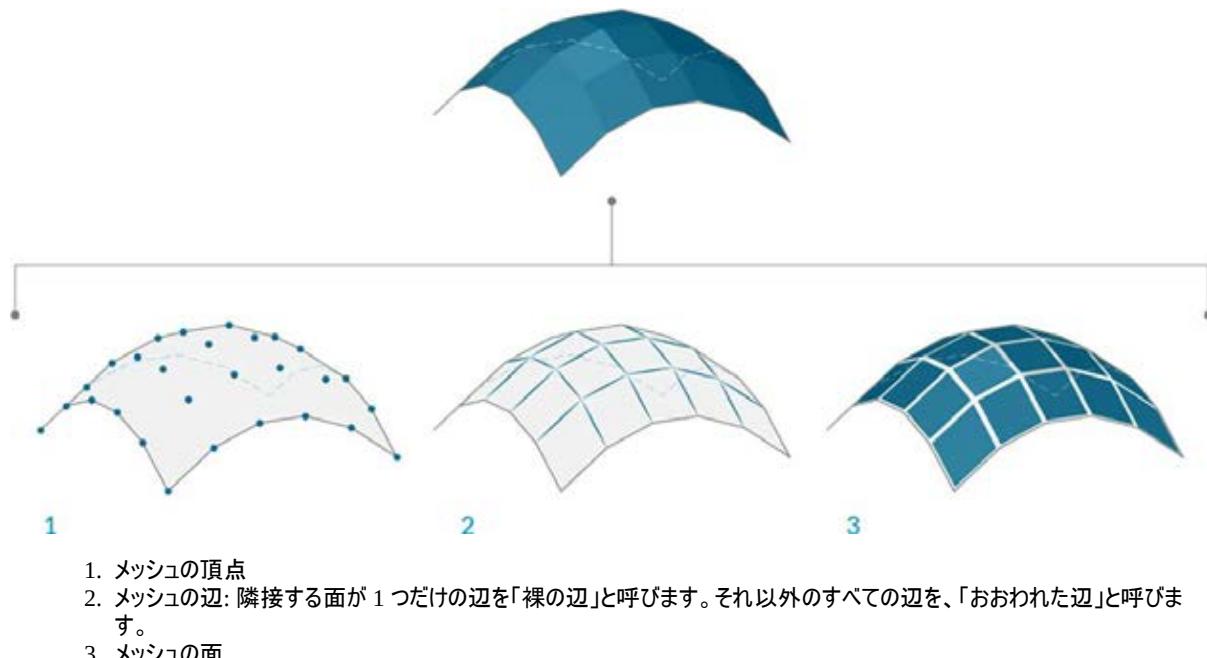
# メッシュ

## メッシュ

コンピュータ モデリングの分野では、3D ジオメトリを表現する形式としてメッシュが広く普及しています。メッシュ ジオメトリは、NURBS に代わる軽量かつ柔軟なジオメトリです。また、メッシュは、レンダリング、視覚化、デジタル ファブリケーション、3D 印刷など幅広い用途に使用することができます。

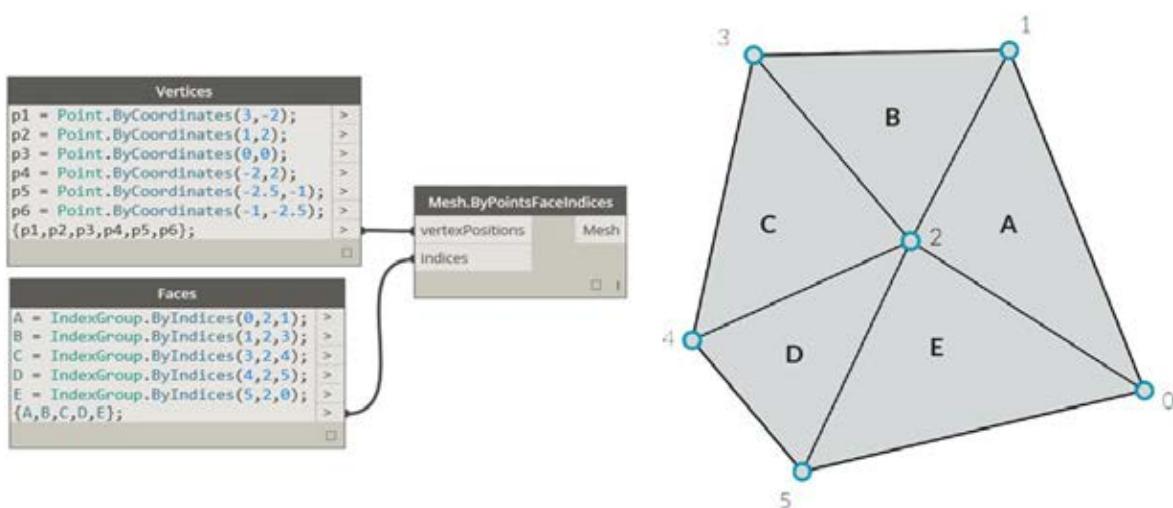
### メッシュの概要

メッシュは、サーフェスまたはソリッド ジオメトリを表す四角形と三角形の集まりです。メッシュ オブジェクトは、ソリッドと同様に、頂点、辺、面から構成されます。メッシュには、標準のプロパティの他に、法線のような独自のプロパティもあります。



### メッシュの要素

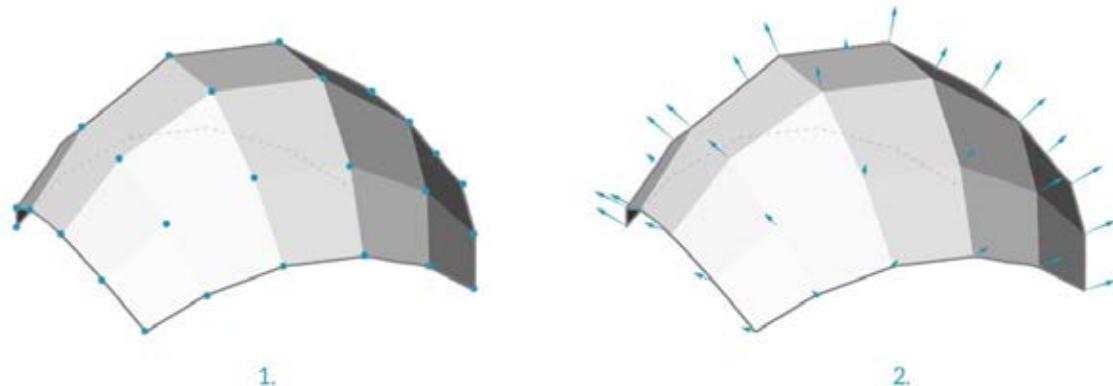
Dynamo では、面と頂点のデータ構造を使用してメッシュを定義します。最も基本的なレベルでは、この構造はポリゴンにグループ化された単なる点の集合です。メッシュの点を頂点と呼び、サーフェスのような形状のポリゴンを面と呼びます。メッシュを作成するには、頂点のリストと、それらの頂点をインデックス グループと呼ばれる面にグループ化するための仕組みが必要です。



1. 頂点のリスト
2. 面を定義するためのインデックス グループのリスト

### 頂点と頂点法線

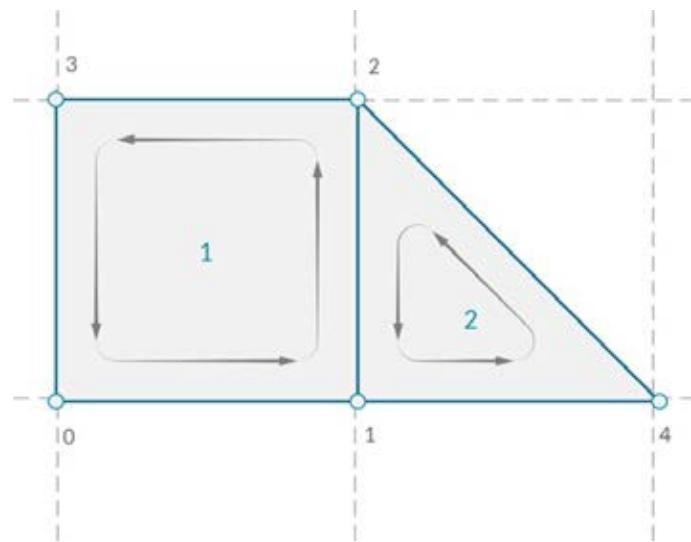
メッシュの頂点は、単純な点のリストです。メッシュを作成する場合や、メッシュの構造に関する情報を取得する場合、頂点のインデックスは非常に重要です。各頂点には、対応する頂点法線(ベクトル)があります。この頂点法線は、頂点において隣接する面の方向の平均を表すため、メッシュが「内向き」か「外向き」かを判断する場合に役立ちます。



1. 頂点
2. 頂点法線

### 面

面は、3つまたは4つの頂点の順番付きリストです。そのため、メッシュ面の「サーフェス」としての表示方法は、インデックス化された頂点の位置によって決まります。メッシュを構成する頂点のリストは既に作成されているため、ここでは個々の点を指定して面を定義するのではなく、頂点のインデックスをそのまま使用します。これにより、複数の面で同じ頂点を使用することができます。



1. 四角形の面は、0、1、2、3 のインデックスから構成されます。
2. 三角形の面は、1、4、2 のインデックスから構成されます。インデックス グループの順序は、変更することができます。ただし、反時計回りに並んでいる必要があります。これにより、面が正しく定義されます。

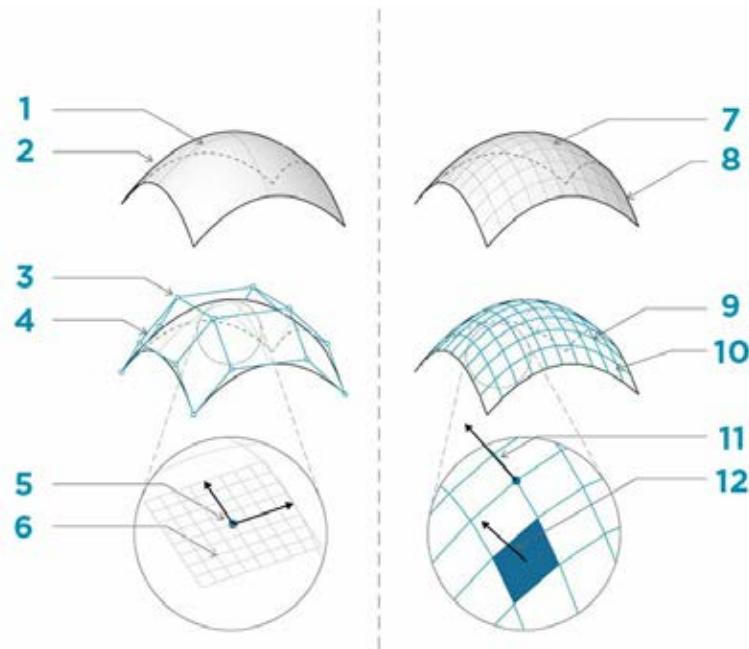
### メッシュと NURBS サーフェスとの比較

メッシュ ジオメトリと NURBS ジオメトリとの違いは何でしょうか。どのような場合にどちらのジオメトリを使用したらよいのでしょうか。

#### パラメータ化

前の章で、NURBS サーフェスは2つの方向に向かう一連の NURBS 曲線によって定義されるということを説明しました。これらの方

向には、U と V というラベルが付けられます。これにより、2 次元サーフェスの範囲に応じて、NURBS サーフェスをパラメータ化することができます。曲線自体は、計算式としてコンピュータに格納されます。これにより、生成されるサーフェスを任意の精度で計算することができます。ただし、複数の NURBS サーフェスを結合するのは難しい場合があります。2 つの NURBS サーフェスを結合すると、ポリサーフェスが作成されます。ジオメトリの異なる部分には、異なる UV パラメータと曲線がそれぞれ定義されます。

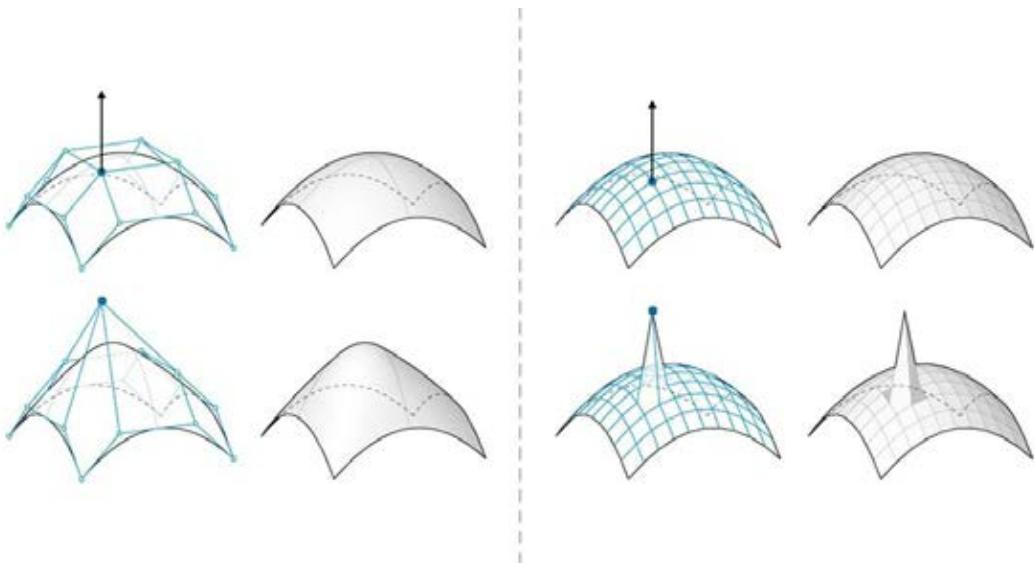


1. サーフェス
2. アイソパラメトリック(Isoparm)曲線
3. サーフェス制御点
4. サーフェス制御ポリゴン
5. パラメトリック点
6. サーフェス フレーム
7. メッシュ
8. 裸の辺
9. メッシュ ネットワーク
10. メッシュの辺
11. 頂点法線
12. メッシュ面、メッシュ面の法線

NURBS サーフェスとは異なり、メッシュは、正確に定義された複数の不連続な頂点と面で構成されます。頂点のネットワークは、通常、単純な UV 座標で定義することはできません。面は互いに連続していないため、精度はメッシュ内で定義されます。精度を高めるには、メッシュを変更し、面の数を増やす必要があります。メッシュには数学的な表現方法がないため、1 つのメッシュ内で複雑なジオメトリをより柔軟に処理することができます。

### ローカルな影響とグローバルな影響

メッシュと NURBS サーフェスのもう一つの重要な違いは、メッシュ内または NURBS ジオメトリ内のローカルの変更が形状全体に与える影響の度合いです。メッシュで 1 つの頂点を移動すると、その頂点に隣接する面だけが影響を受けます。NURBS サーフェスの場合、影響の範囲はより複雑で、サーフェスの次数、制御点のウェイト、制御点のノットによって影響の範囲が異なります。ただし、一般的には、NURBS サーフェスで 1 つの制御点を移動した場合の方が、ジオメトリの変更はより広範囲で滑らかなものになります。



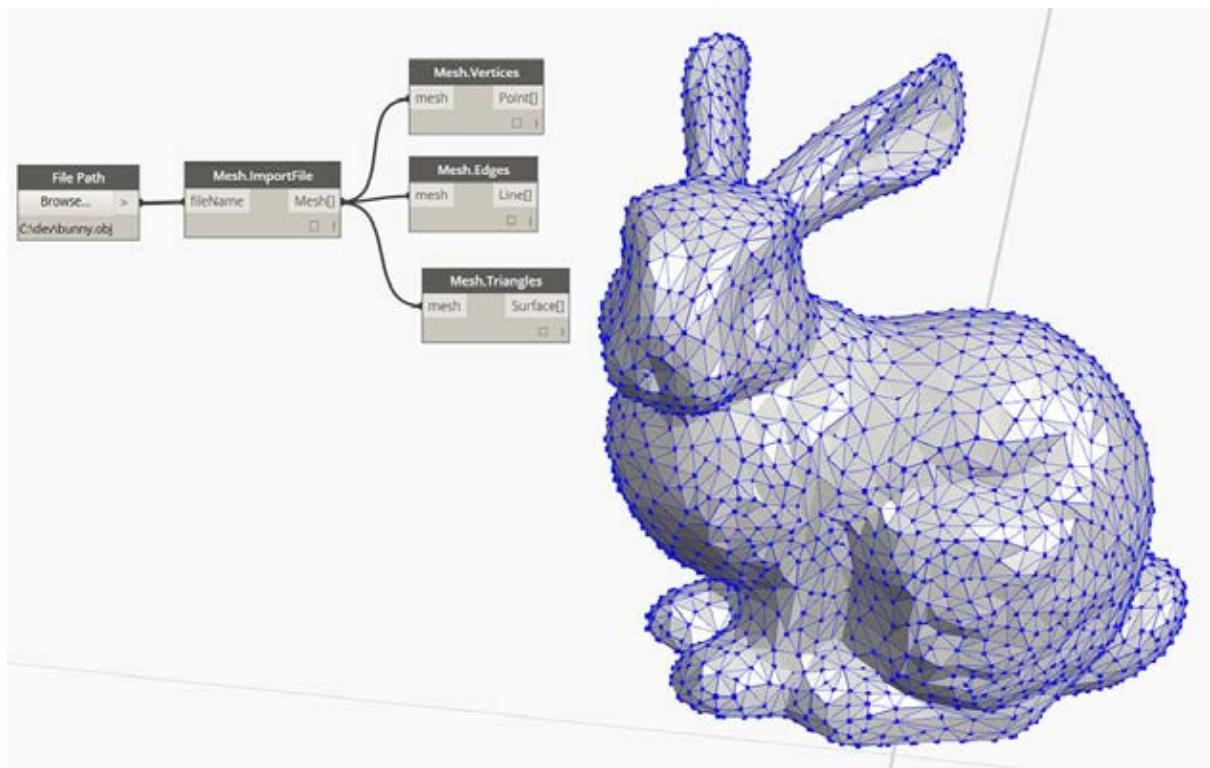
1. NURBS サーフェスで 1 つの制御点を移動すると、形状全体に影響が及びます。
2. メッシュ ジオメトリで 1 つの頂点を移動すると、隣接する要素にのみ影響が及びます。

この対比関係は、直線と曲線で構成されるベクター イメージと、個々のピクセルで構成されるラスター イメージとの関係に似ています。ベクター イメージを拡大表示しても曲線はくっきりと表示されるのに対して、ラスター イメージを拡大表示すると個々のピクセルが拡大されて表示されます。つまり、NURBS サーフェスは、数学的に滑らかな関係があるという点でベクター イメージに似ています。一方メッシュは、一定の解像度を持つという点でラスター イメージに似ています。

## Mesh Toolkit

Dynamo のメッシュ機能は、[Mesh Toolkit](#) パッケージをインストールすることによって拡張することができます。Dynamo Mesh Toolkit は、外部ファイル形式からメッシュを読み込む機能、Dynamo のジオメトリオブジェクトからメッシュを作成する機能、頂点とインデックスからメッシュを手動で作成する機能を提供するライブラリです。このライブラリには、メッシュの変更や修復を行うためのツールや、製造処理で使用する水平方向のスライスを抽出するためのツールも用意されています。

Mesh Toolkit の使用例については、第 10.2 章を参照してください。



# ジオメトリの読み込み

## ジオメトリの読み込み

いくつかの方法で、Dynamo にジオメトリを読み込むことができます。前のセクションでは、*Mesh Toolkit* を使用してメッシュを読み込む方法を説明しました。同様に、ソリッドモデルを .SAT ファイルから読み込むこともできます。これらのプロセスでは、別のプラットフォームでジオメトリを開発し、そのジオメトリを Dynamo に読み込み、ビジュアル プログラミングを使用してパラメトリック操作を適用します。

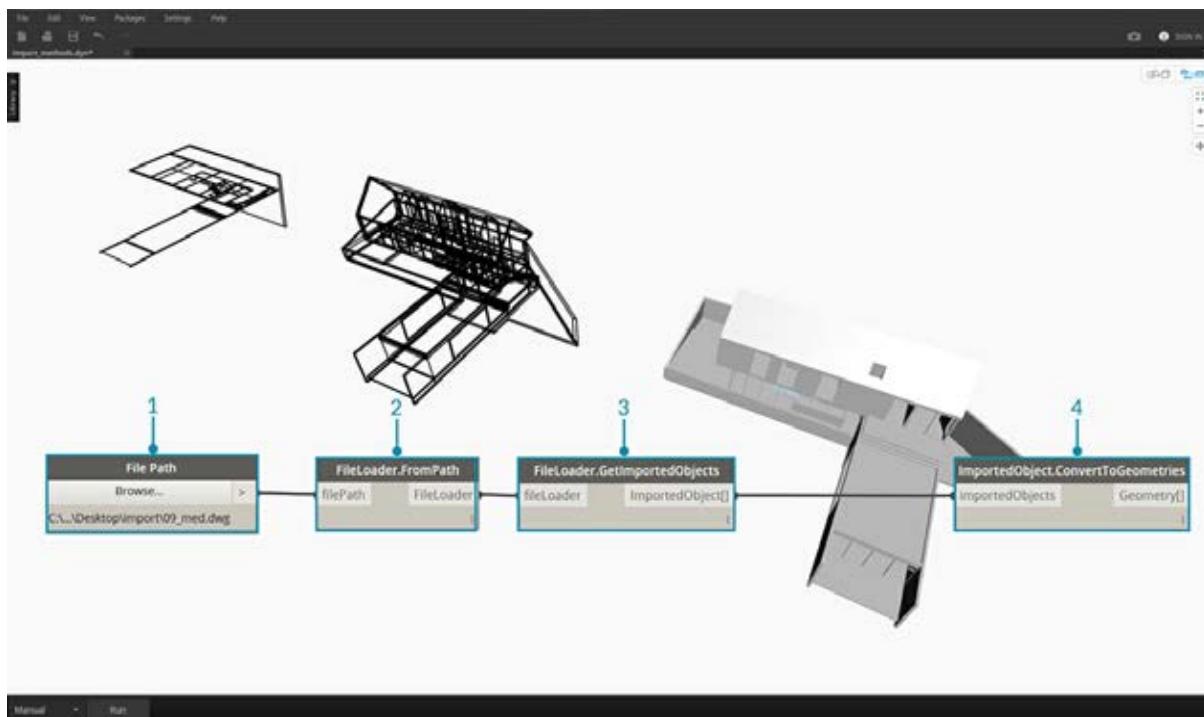
ジオメトリは、ATF 変換と呼ばれる処理を使用して読み込むこともできます。ATF 変換では、ジオメトリだけでなく、ファイルの構造も読み込むことができます。たとえば、モデル全体を読み込むのではなく、読み込む .DWG のレイヤを選択することができます。この操作については、これ以降で詳しく説明します。

### DWG ファイルからジオメトリを読み込む

DWG を Dynamo 環境に読み込むためのノードは、*Translation* カテゴリにあります(注: これらのノードは、[Dynamo Studio](#) でのみ利用できます)。次の例は、ファイルを参照してその内容を読み込み、使用可能な Dynamo ジオメトリに変換するための一連のコンポーネントを示しています。Dynamo には、DWG ファイルの特定のオブジェクトをフィルタまたは選択して読み込むための機能が用意されています。この機能については、これ以降で詳しく説明します。DWG ファイルからのジオメトリの読み込みについて詳しくは、Ben Goh の[ブログ記事](#)を参照してください。

### 読み込まれたオブジェクトの取得

Dynamo Studio に DWG を読み込むための最も簡単な方法は、ワークスペースにファイル全体を読み込む方法です。

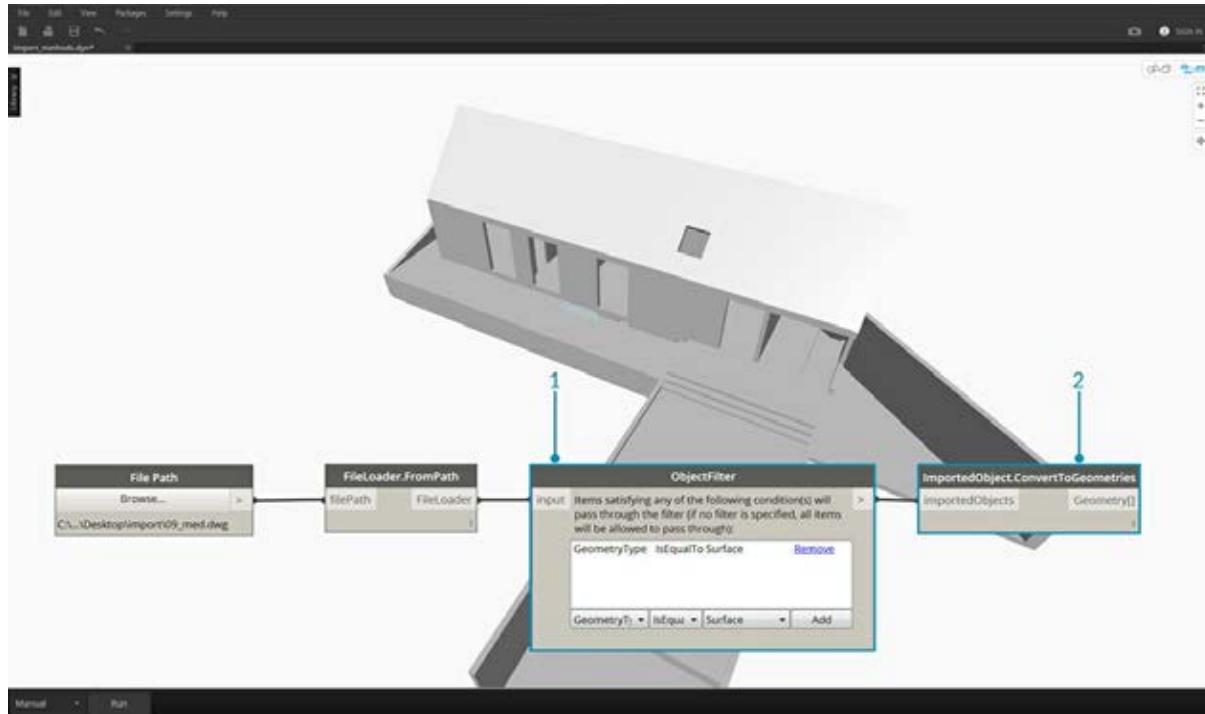


上の図のように、DWG ファイル内のすべてのジオメトリタイプ(サーフェス、メッシュ、曲線、線分)が Dynamo に読み込まれます。

### オブジェクト フィルタ

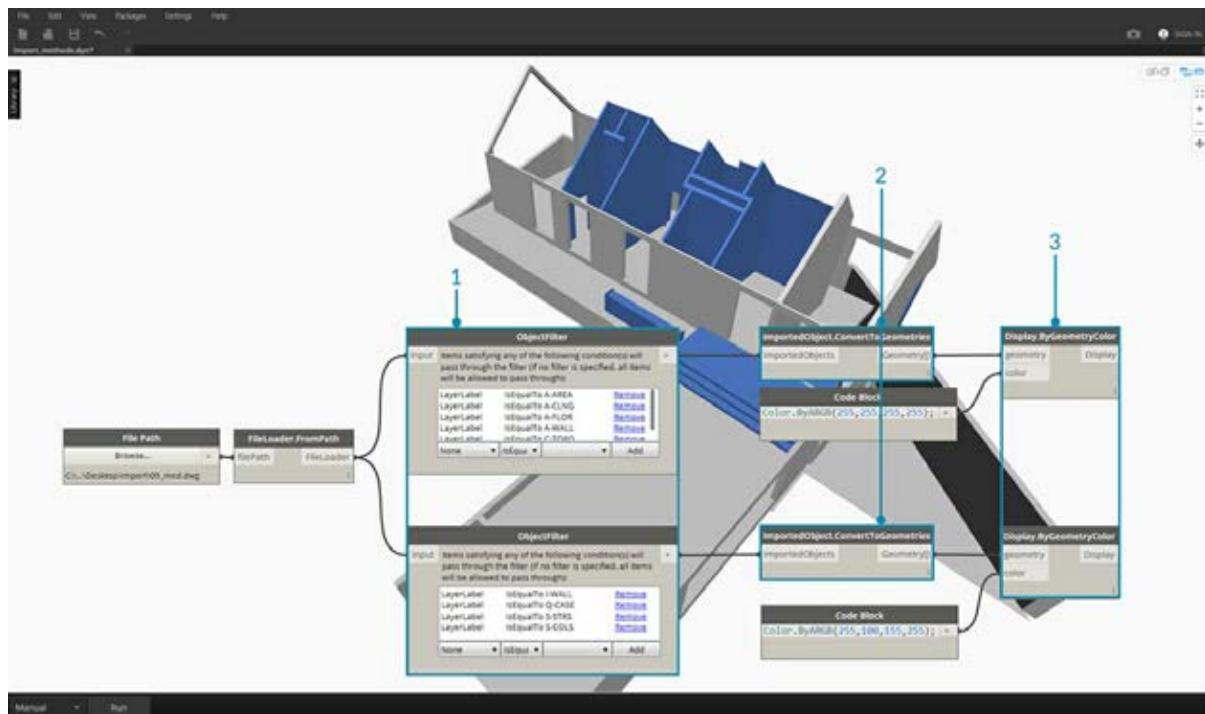
DWG ファイルから読み込むジオメトリを指定するには、ObjectFilter ノードを定義に追加します。ObjectFilter ノードは、Fileloader ノードと ImportedObject ノードのリストと互換性があり、ImportedObject のリストを出力します。

次の図は、各 **ObjectFilter** ノード内の条件ステートメントを示しています。指定された条件を満たすすべての **ImportedObject** がフィルタを通過します。レイヤのラベル(レイヤ名)、ジオメトリタイプ、拡散色などに基づいてフィルタすることができます。別のフィルタと組み合わせて、選択肢を絞り込むこともできます。



1. **FileLoader.GetImportedObjects** ノードの代わりに **ObjectFilter** ノードを使用して、DWG ファイルで特定の条件に基づく検索を行います。この場合、サーフェス ジオメトリのみが読み込まれ、前の図で表示されていたすべての曲線と線分のジオメトリが削除されます。
2. **ObjectFilter** ノードを **ImportedObject.ConvertToGeometries** ノードに接続し、フィルタ後の中のジオメトリを読み込みます。

異なる条件ステートメントを持つ 2 つのフィルタを追加することにより、ジオメトリのリストを複数のストリームに分割することができます。



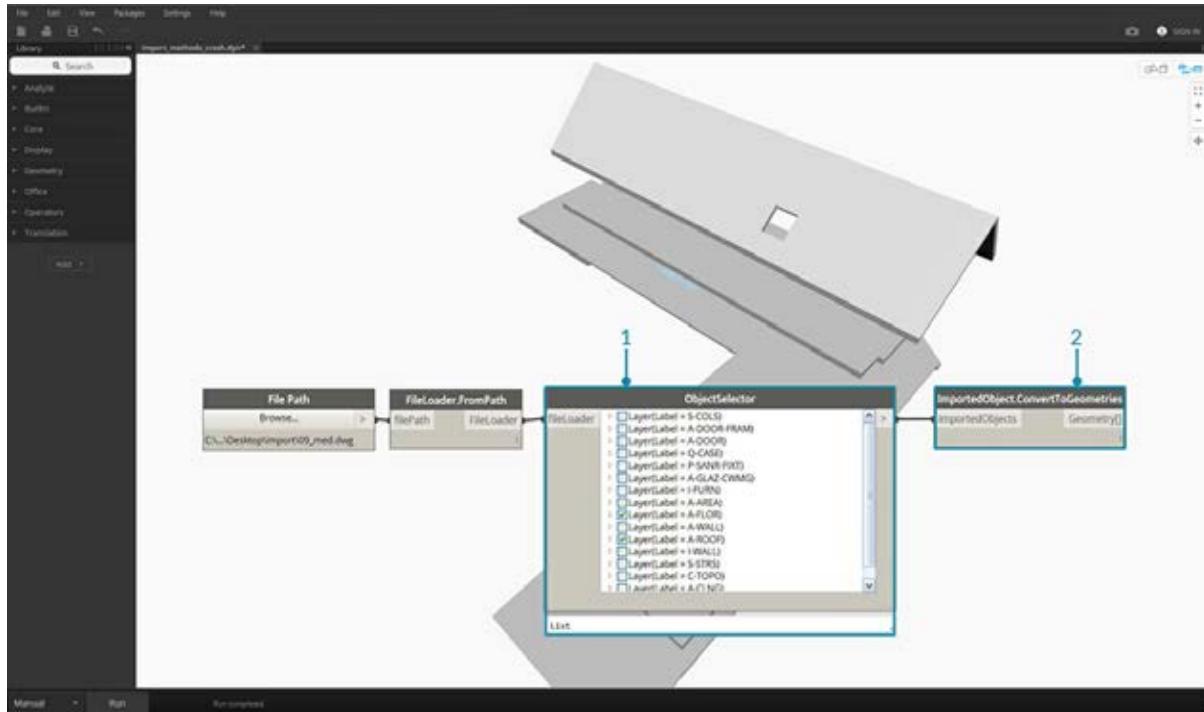
1. **FileLoader.GetImportedObjects** ノードの代わりに、別の条件ステートメントを持つ 2 つの **ObjectFilter** ノードを

使用します。こうすると、1つのファイルのジオメトリが2つの異なるストリームに分割されます。

2. ObjectFilter ノードを ImportedObject.ConvertToGeometries ノードに接続し、filtrタ後のジオメトリを読み込みます。
3. ImportedObject.ConvertToGeometries ノードを Display.ByGeometryColor ノードに接続し、各ストリームを異なる色で表示します。

### 明示的なオブジェクト選択

別 の方法として、ObjectSelector ノードを使用して、DWG ファイルからオブジェクトを読み込むこともできます。この方法では、フィルタを使用することなく、Dynamo に読み込むオブジェクトとレイヤを指定することができます。

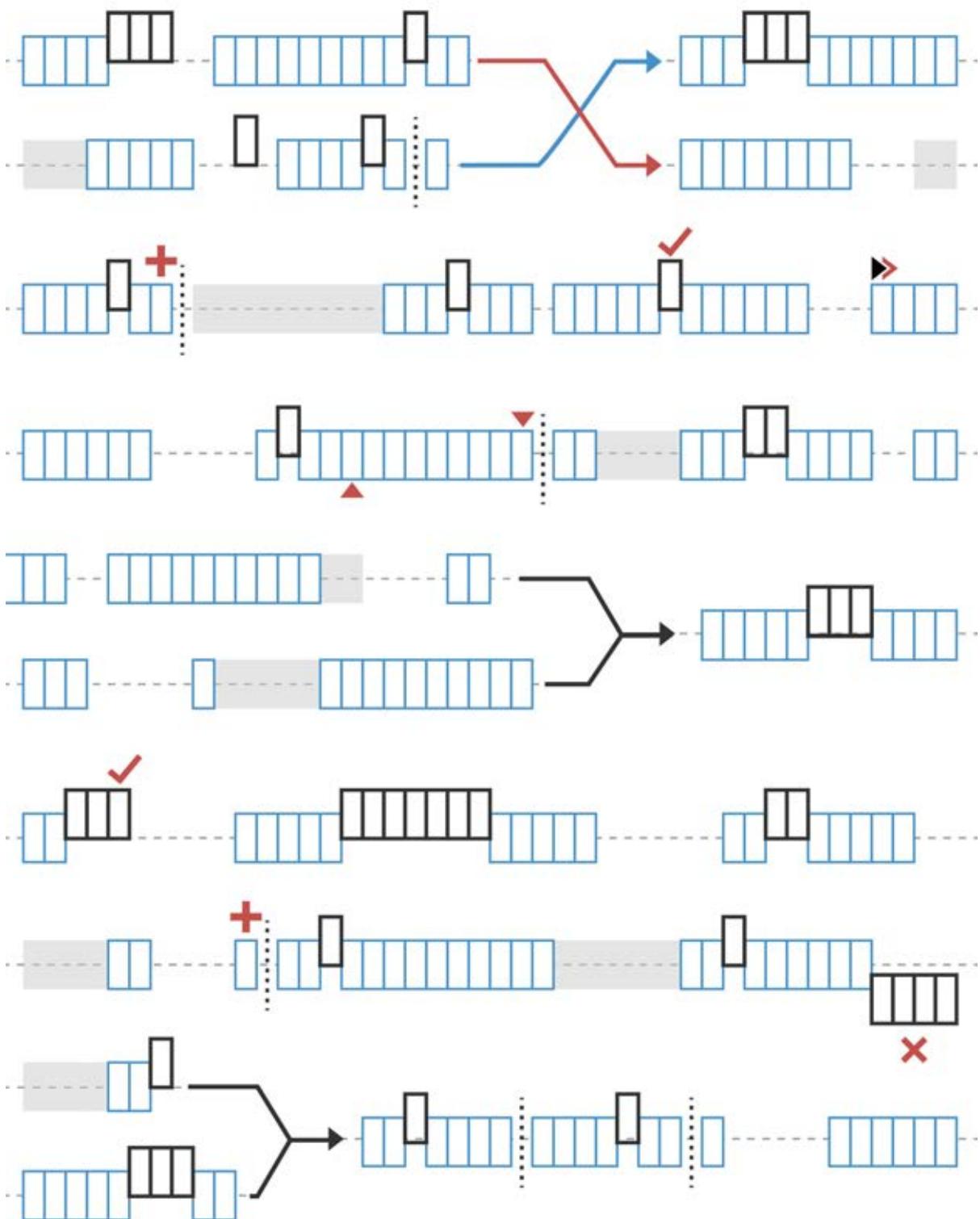


1. FileLoader.GetImportedObjects ノードの代わりに ObjectSelector ノードを使用して、DWG ファイル内の特定のレイヤとオブジェクトを指定します。
2. ObjectSelector ノードを ImportedObject.ConvertToGeometries ノードに接続します。

## **リストを使用した設計**

### **リストを使用した設計**

リストを使用して、データを整理することができます。コンピュータのオペレーティング システム上にはファイルとフォルダがありますが、Dynamo では、項目とリストが、それぞれファイルとフォルダに対応します。オペレーティング システムのように、さまざまな方法でデータの作成、変更、クリアを実行することができます。この章では、Dynamo でリストを管理する方法を詳しく説明します。



# リストの概要

## リストの概要

リストとは、要素(項目)の集合です。例として、1 房のバナナを考えてみましょう。1 本のバナナが、リスト(房)内の 1 つの項目になります。それぞれのバナナをばらばらに取り上げるより、房をまとめて持ち上げる方が簡単です。同じことが、データ構造内のパラメータに基づく関係によって各要素をグループ化する場合についても当てはまります。



写真: [AugustusBinu](#)

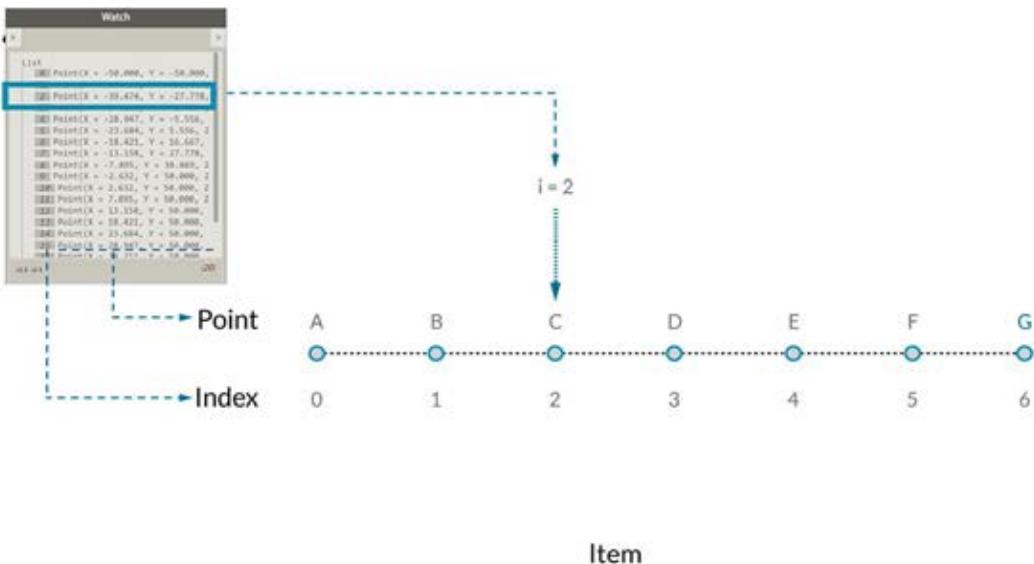
日用品を購入する場合、購入した商品をすべて買い物カゴに入れます。このカゴも、リストと考えることができます。たとえば、3 房のバナナを使用して多くのバナナパンを作るします。\*\* この場合、すべてのバナナが入っている袋がバナナの房のリストで、それぞれの房がバナナのリストになります。この袋は 2 次元の「リストのリスト」で、バナナは 1 次元の「リスト」です。

Dynamo では、リスト データには順序が付けられ、各リストの最初の項目にはインデックス「0」が割り当てられます。次のセクションでは、Dynamo におけるリストの定義の仕組みと、複数のリストの相互関係について説明します。

### ゼロで始まるインデックス

リストのインデックスは、1 ではなく必ず 0 から始まります。これは、最初は不思議に感じるかもしれません。今後、リストの最初の項目と言う場合は、インデックス 0 の付いた項目のことを意味します。

たとえば、右手の指を使って数を数える場合、1 から 5 の順序で数えます。0 から数え始めることはほとんどないはずです。しかし、右手の指を Dynamo のリストだと仮定すると、それぞれの指に 0 から 4 のインデックスが割り当てられることになります。プログラミングの初心者にとっては少し不思議に感じられるかもしれません、ほとんどのコンピュータ システムでは、インデックスの値はゼロで始まるのが普通です。



Item

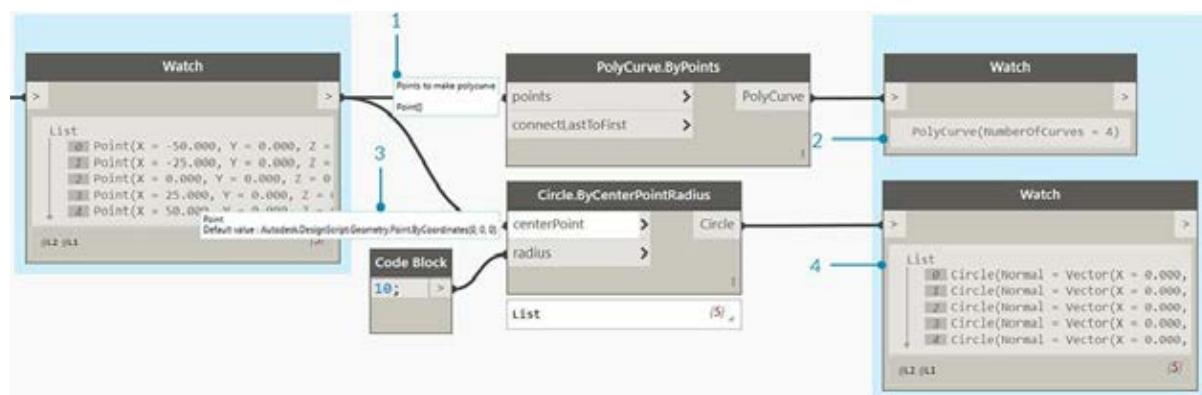
ここでも、引き続きリスト内に 5 つの項目が含まれています。ただし、最初の項目に 0 のインデックス値が割り当てられているものと仮定して説明を続けます。リストに格納できる項目は、数値だけではありません。Dynamo でサポートされているデータタイプであれば、点、曲線、サーフェス、ファミリなど、任意のデータタイプをリスト内に格納することができます。

多くの場合、リストに格納されているデータのタイプを確認する最も簡単な方法は、ノードの出力を Watch ノードに接続する方法です。既定では、Watch ノードにはリストの左側のすべてのインデックスと、右側のすべてのデータ項目が自動的に表示されます。

これらのインデックスは、リストを操作する場合の重要な要素です。

## 入力と出力

リストの入力と出力は、使用する Dynamo ノードによって異なります。例として、5 つの点を持つリストのノードの出力を PolyCurve.ByPoints ノードと Circle.ByCenterPointRadius ノードに接続してみましょう。



1. PolyCurve.ByPoints ノードの *points* 入力には、*Point[]* が必要です。これは、点のリストを表しています。
2. PolyCurve.ByPoints ノードの出力は、5 つの点を持つリストから作成された 1 つのポリカーブです。
3. Circle.ByCenterPointRadius ノードの *CenterPoint* 入力には、*Point* が必要です。
4. Circle.ByCenterPointRadius ノードの出力は、5 つの円を持つリストです。それぞれの円の中心が、元のリスト内の点に対応しています。

PolyCurve.ByPoints ノードと Circle.ByCenterPointRadius ノードの入力データは同じですが、PolyCurve.ByPoints ノードは 1 つのポリカーブを返し、Circle.ByCenterPointRadius ノードは各点を中心とする 5 つの円を返します。この場合、5 つの点をつなぐ曲線としてポリカーブが描画され、Circle.ByCenterPointRadius ノードは各点を中心とする個別の円を描画します。これは、直感的に理解することができます。では、データには何が起きているのでしょうか。

Polycurve.ByPoints ノードの *points* 入力の上にカーソルを置くと、この入力に *Point[]* が必要であることがわかります。最後の角括弧に注意してください。これは、点のリストであることを表しています。ポリカーブを作成するには、ポリカーブごとにリストを入力する必要があります。入力された各リストは、Polycurve.ByPoints ノードによって 1 つのポリカーブに集約されます。

一方、*Circle.ByCenterPointRadius* ノードの *centerPoint* 入力には、*Point* が必要です。このノードはリストの項目である 1 つの点を取得し、それを円の中心として定義します。入力されたデータから 5 つの円が生成されるのは、このためです。Dynamo の入力に関するこれらの違いを認識しておくと、ノードによるデータ処理の仕組みを正しく理解することができます。

## レーシング

データの一一致は、明確に解決することができない問題です。このような問題は、異なるサイズの入力をノードに渡すときに発生します。データ一一致アルゴリズムを変更すると、結果が大きく変わってしまうことがあります。

例として、2 点の間に直線セグメントを作成するノード(*Line.ByStartPointEndPoint*)を考えてみます。このノードは、点の座標を指定する 2 つの入力パラメータを使用します。

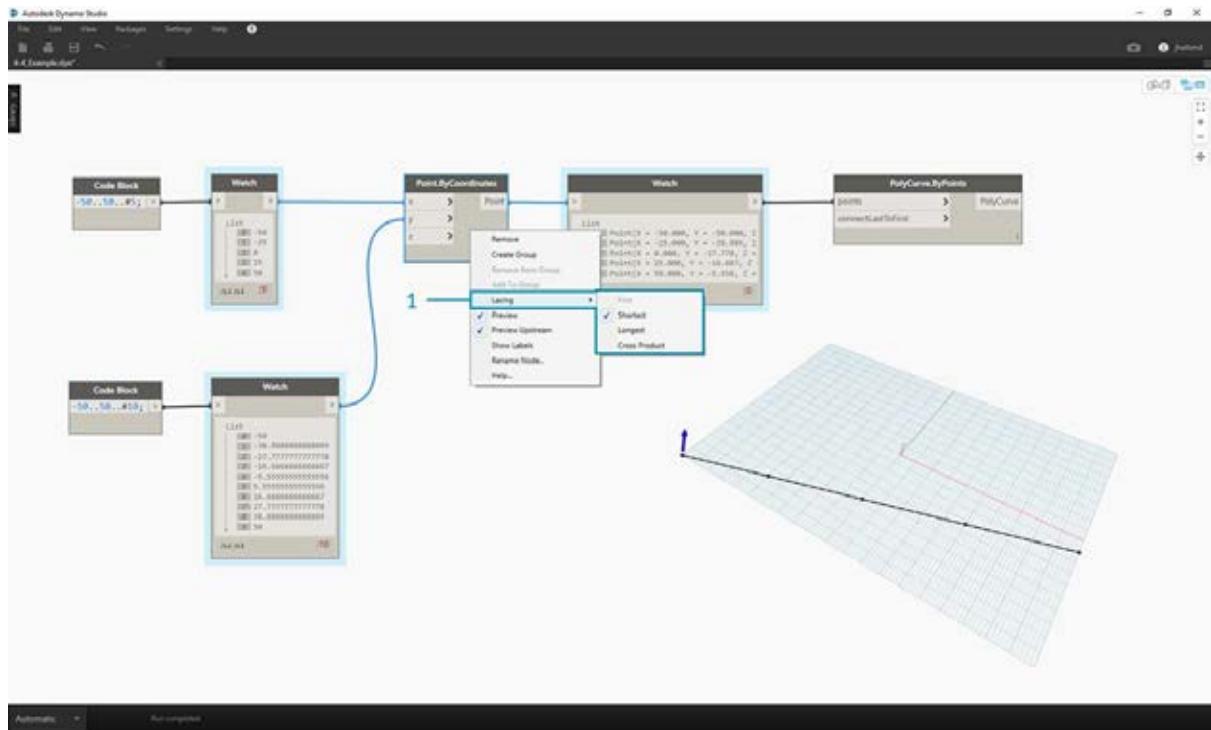


これらの点の集まりの間に直線を描画する場合、いくつかの方法があります。レーシング オプションを使用するには、ノードの中心を右クリックして[レーシング]メニューを選択します。

## 基準ファイル

この演習に付属しているサンプル ファイルをダウンロードしてください(右クリックして[名前を付けてリンク先を保存...]を選択): [Lacing.dyn](#)。すべてのサンプルファイルの一覧については、付録を参照してください。

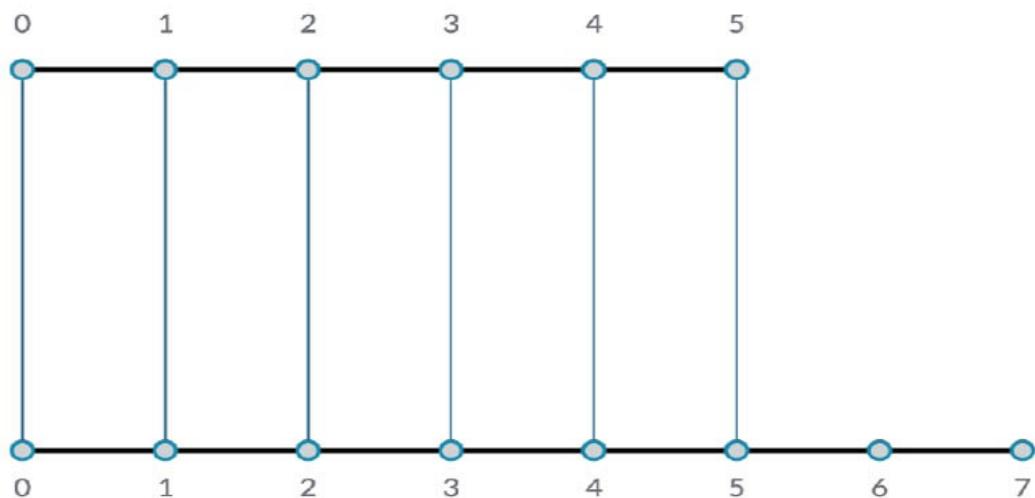
次の図を使用して、レーシング操作について説明します。ここでは、上記の基準ファイルを使用して、最短リスト、最長リスト、外積を定義します。

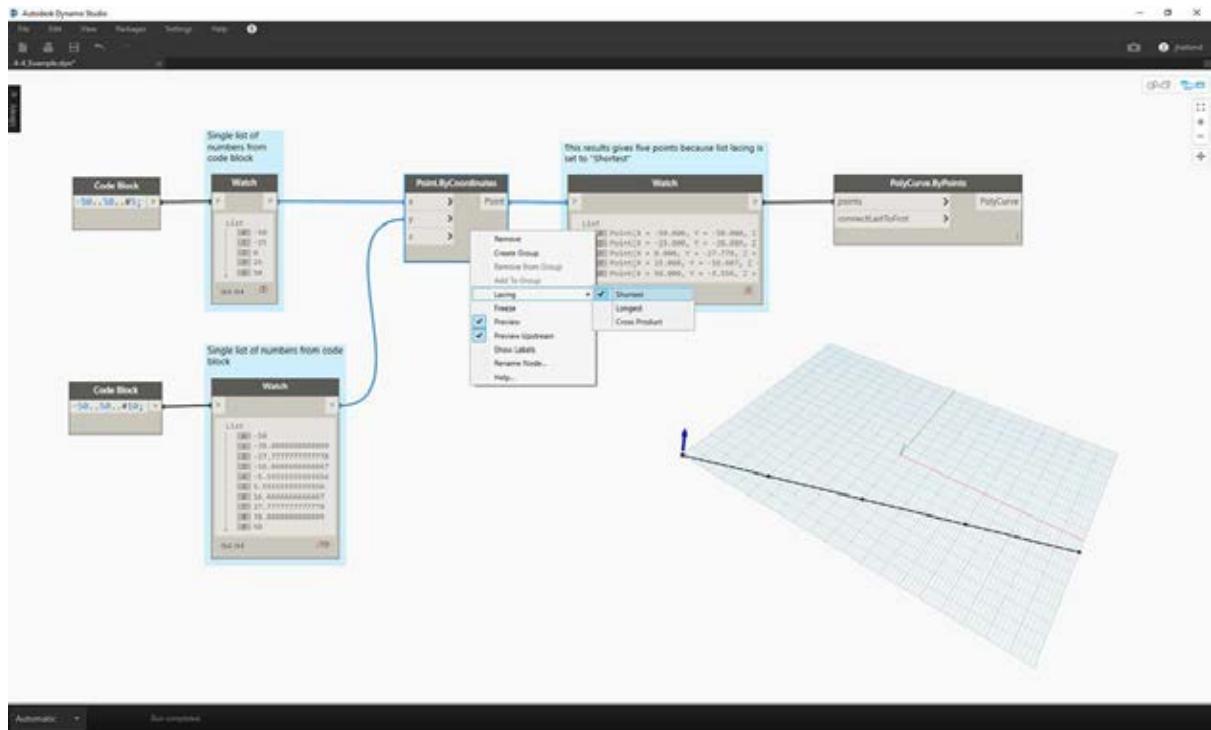


1. *Point.ByCoordinates* ノードでレーシングを変更しますが、上図のグラフについては何も変更しません。

#### 最短リスト

最も単純な方法は、一方のリストが終了するまで、入力された点を 1 対 1 で接続していく方法です。これは、「最短リスト」アルゴリズムと呼ばれます。Dynamo ノードの既定の動作です。

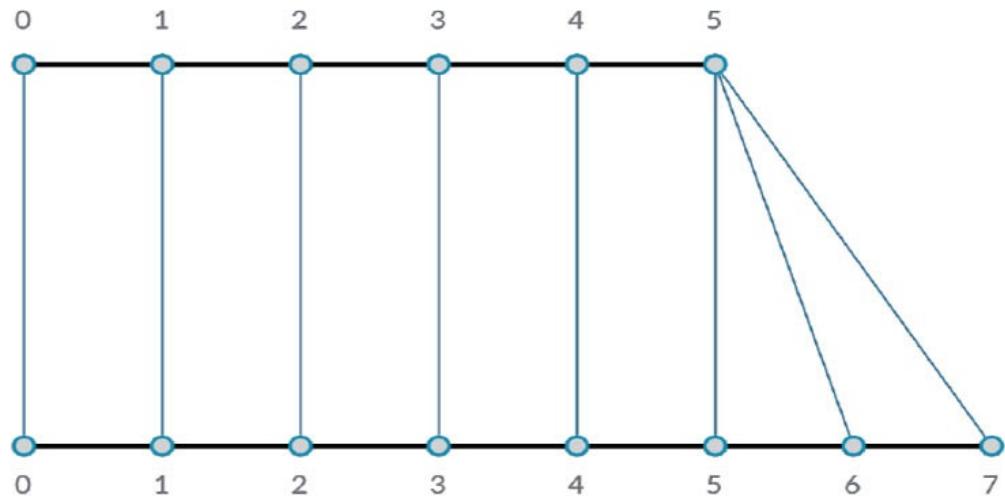


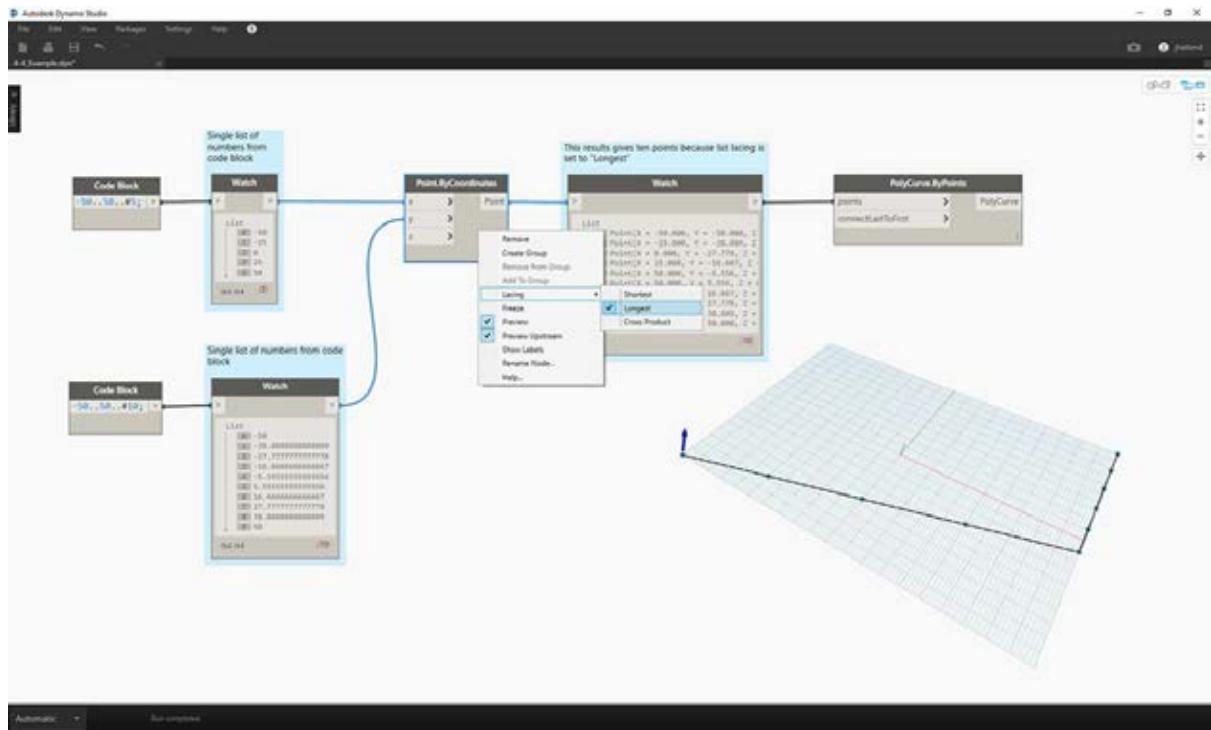


レーシングを[最短リスト]に変更すると、5つの点で構成される基本的な対角線が生成されます。最短リストのレーシングは、2つのリストのうち短い方のリストの最後の項目に達した場合に動作が終了します。そのためこの例では、5つの点が含まれているリストの最後に達すると、レーシング動作が停止します。

### 最長リスト

「最長リスト」アルゴリズムの場合、すべてのリストの最後の項目に達するまで、同じ要素を繰り返し使用して入力が接続されたままの状態になります。

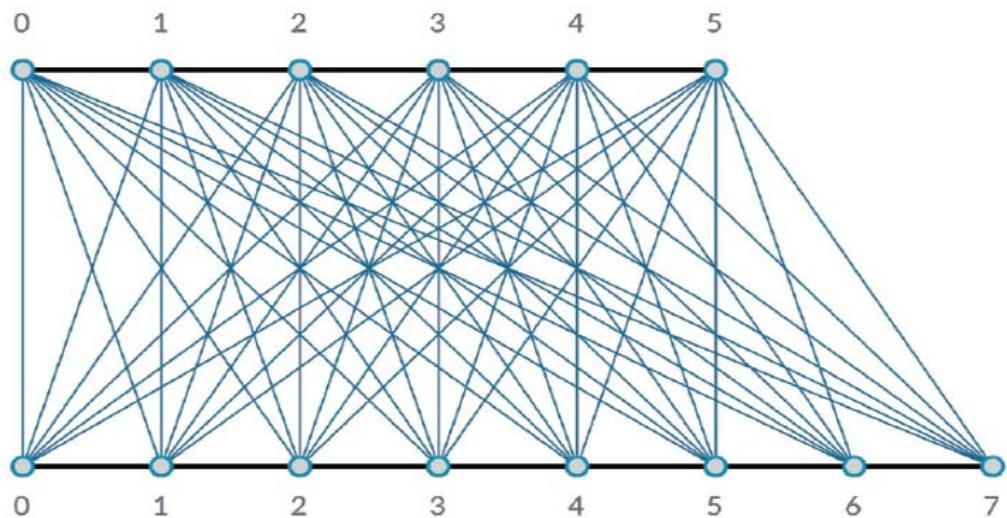


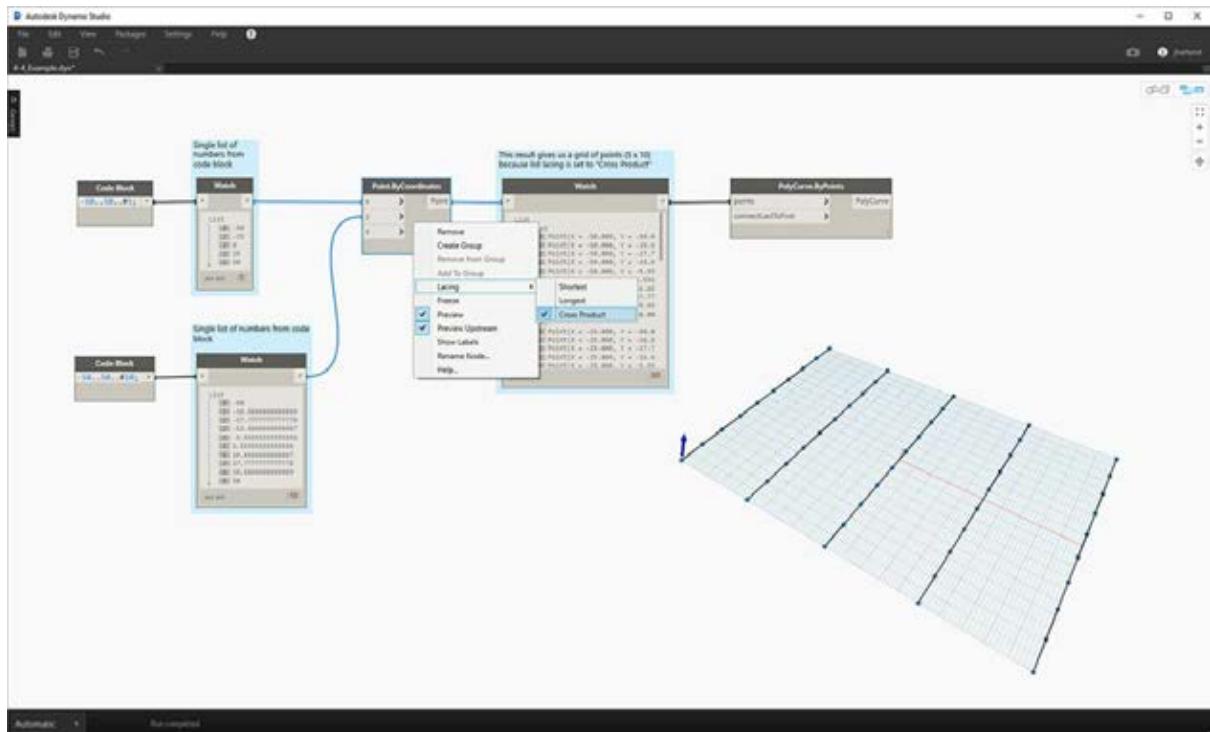


レーシングを[最長リスト]に変更すると、垂直方向に伸びる対角線が生成されます。上の概念図のように、長い方のリストの最後の項目に達するまで、短い方のリスト(項目数が 5 のリスト)の最後の項目が繰り返し使用されます。

## 外積

最後に、「外積」方式について説明します。この方式では、考えられる接続がすべて生成されます。





レーシングを[外積]に変更すると、各リスト間で考えられるすべての組み合わせの 5x10 の点のグリッドが生成されます。これは、上の概念図に示す外積に該当するデータ構造です。ただし、ここで使用しているデータは、「リストのリスト」です。ポリカーブを接続すると、各リストは X 値によって定義され、垂直な直線の列が生成されていることがわかります。

# リストの操作

## リストの操作

リストとは何かということを理解したところで、ここからは、リストに対して実行できる操作について見ていきます。ここでは、リストをトランプのカードだと考えてください。1組のトランプがリストで、1枚1枚のカードが1つの項目となります。



写真: [Christian Gidlöf](#)

リストに関する質問(クエリー)を作成する場合、次のような質問が考えられます。これにより、リストの特性(プロパティ)がわかります。

- Q: カードの数は?52.
- Q: スートの数は?4.
- Q: 素材は?A: 紙
- Q: 長さは?A: 89 ミリ
- Q: 幅は?A: 64 ミリ

リストに対して実行できる操作(アクション)としては、次のような操作が考えられます。この場合、実行する操作に応じてリストが変化します。

- カードをシャッフルする。
- 数字の順にカードを並べ替える。
- スート別にカードを並べ替える。
- カード全体をいくつかに分割する。
- 各プレーヤにカードを配ることにより、カード全体をいくつかに分割する。
- デッキから特定のカードを選ぶ。

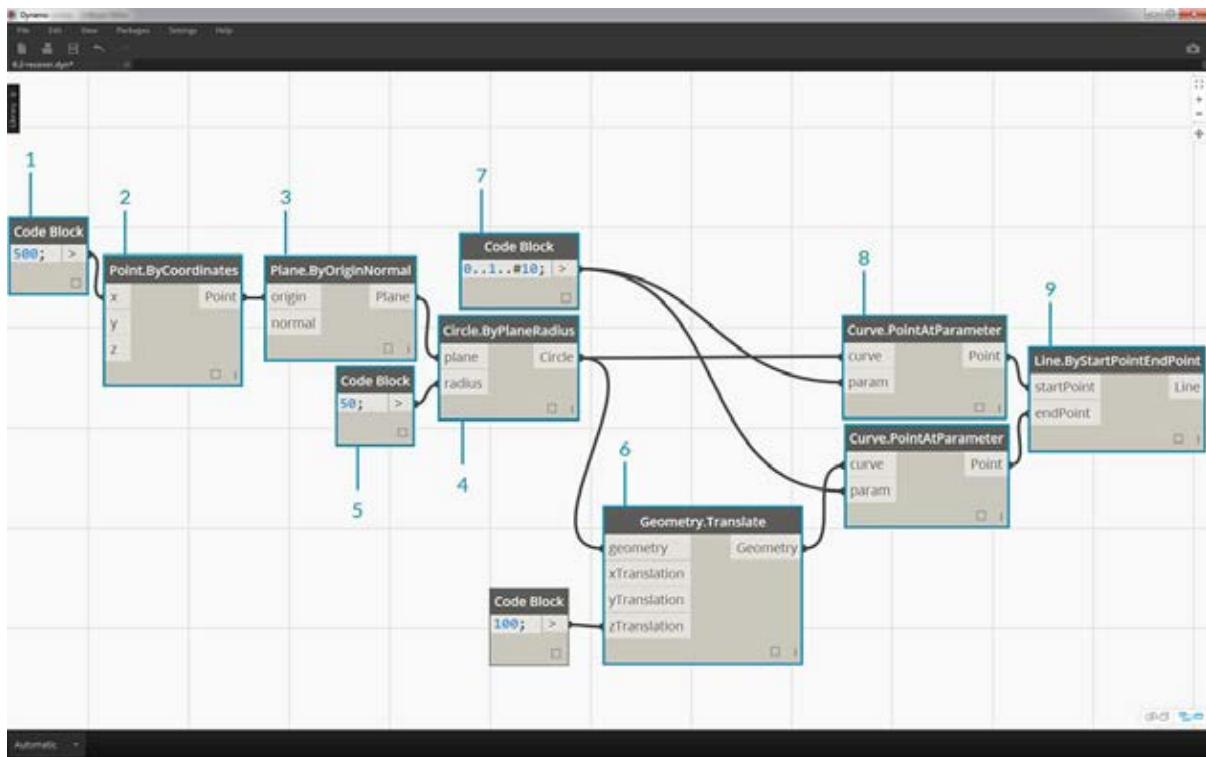
Dynamo では、これらの操作に似た操作を実行することができます。その場合、Dynamo の各ノードを使用して、一般的なデータのリストを操作することになります。これ以降の各演習では、リストに対して実行できる基本的な操作をいくつか見ていきます。

## リストの操作

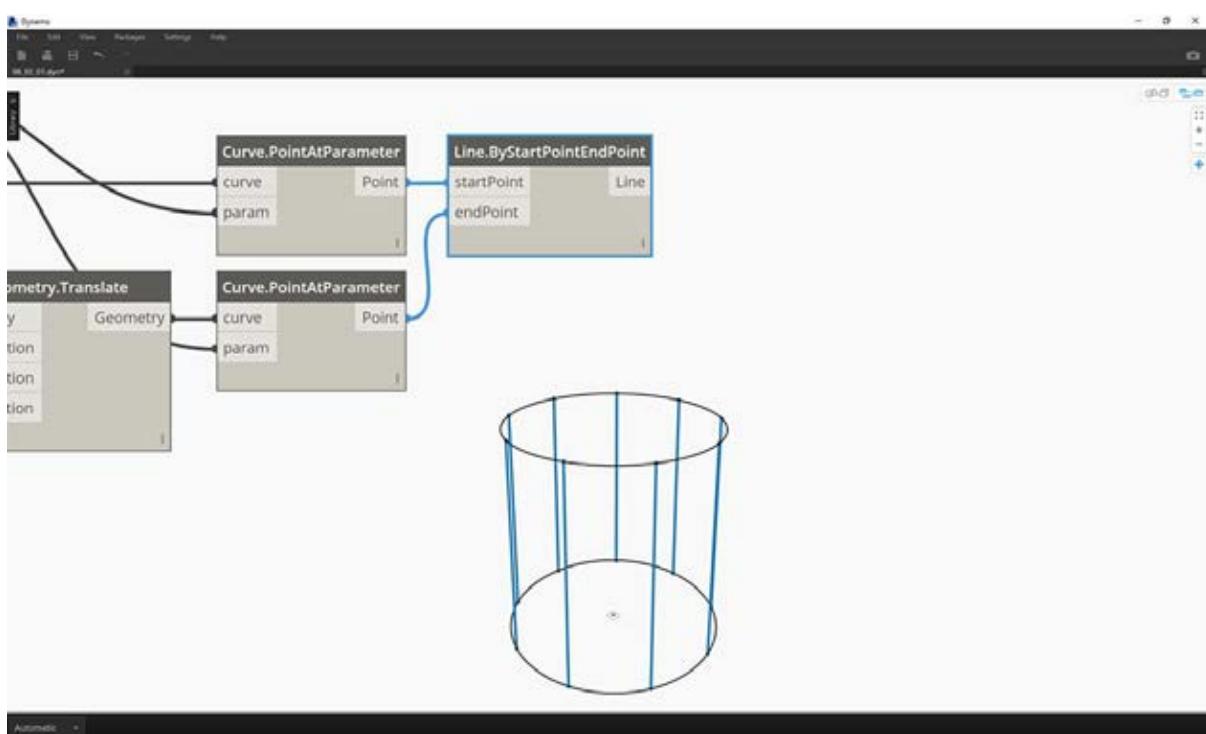
次の図は、基本的なリスト操作を説明するための基本的なグラフを示しています。リスト内のデータを管理する方法と、視覚的な結果を表示する方法について説明します。

### 演習 - リストの操作

この演習に付属しているサンプル ファイルをダウンロードしてください(右クリックして[名前を付けてリンク先を保存...]を選択):  
[List-Operations.dyn](#)。すべてのサンプル ファイルの一覧については、付録を参照してください。

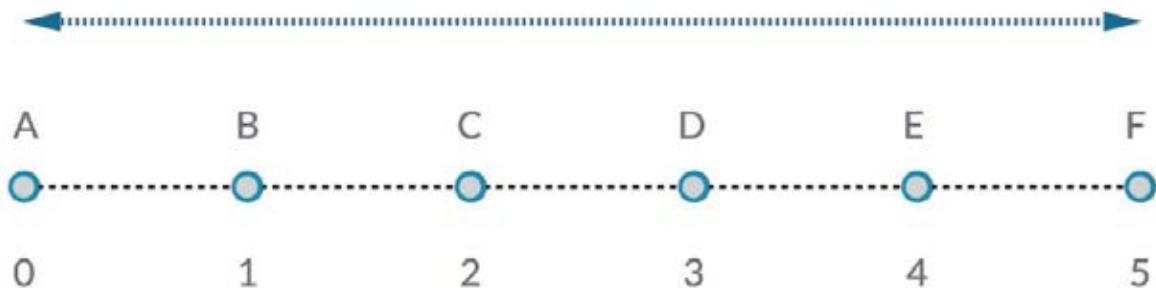


1. 最初に、500; という値が表示されている *Code Block* ノードを使用します。
2. 上記の *Code Block* ノードを *Point.ByCoordinates* ノードの *x* 入力に接続します。
3. 上記のノードを *Plane.ByOriginNormal* ノードの *origin* 入力に接続します。
4. *Plane.ByOriginNormal* ノードを *Circle.ByPlaneRadius* ノードの *plane* 入力に接続します。
5. *Code Block* ノードを使用して、*radius* の値を 50; に設定します。これは、最初に作成する円の半径です。
6. *Geometry.Translate* ノードを使用して、上記の円を Z の正の向きに 100 単位移動します。
7. *Code Block* ノードで 0..1..#10; というコードを指定して、0 から 1 までの範囲で 10 個の数字を定義します。
8. 上記の *Code Block* ノードを 2 つの *Curve.PointAtParameter* ノードの *param* 入力に接続します。次に、上部に配置されている方の *Curve.PointAtParameter* ノードの *curve* 入力に *Circle.ByPlaneRadius* ノードを接続し、下部に配置されている方の *Curve.PointAtParameter* ノードの *curve* 入力に *Geometry.Translate* ノードを接続します。
9. *Line.ByStartPointEndPoint* ノードを使用して、2 つの *Curve.PointAtParameter* ノードを接続します。



- Watch3D ノードに、Line.ByStartPointEndPoint ノードの結果が表示されます。ここからは、2つの円の間に線分を描画して、リストの基本的な操作を実行するための基本的な Dynamo グラフを作成します。また、このグラフを使用して、リストに対する操作を詳しく見ていきます。

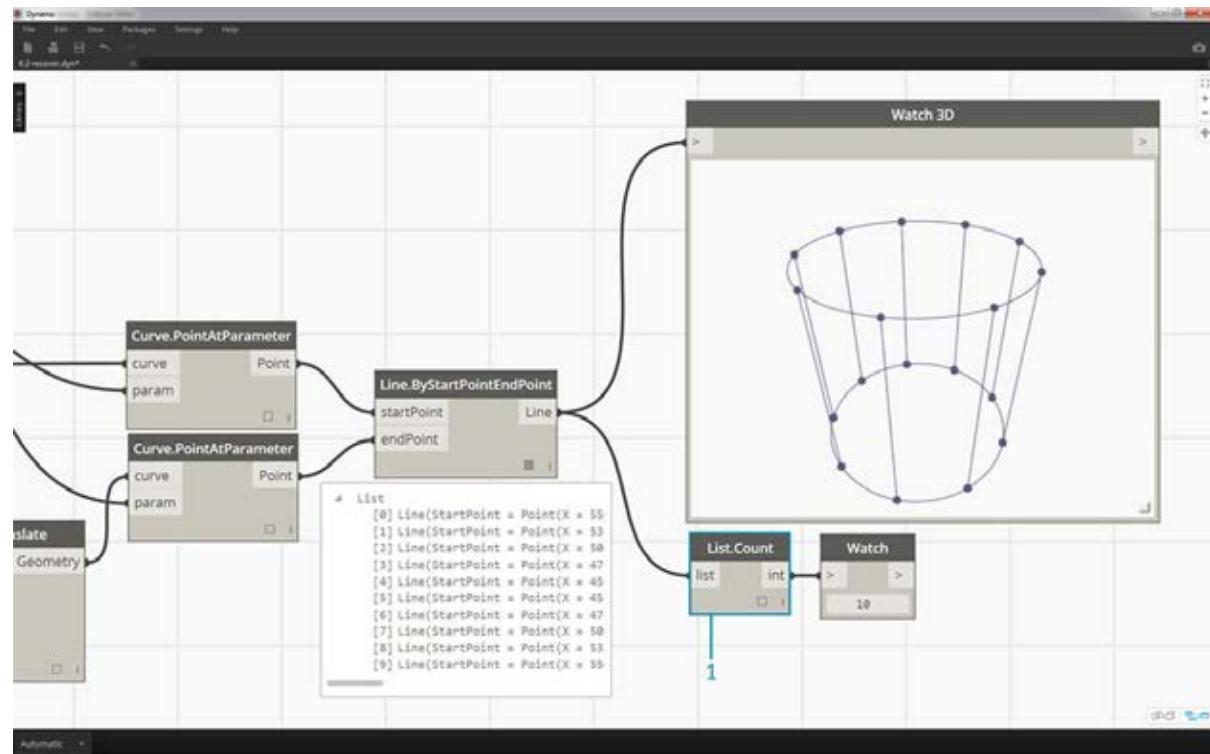
## List.Count



**List.Count** ノードは、リスト内の値の数をカウントしてその数を返すという単純なノードです。「リストのリスト」を操作する場合は、このノードの使用方法も多少複雑になりますが、それについてはこれ以降のセクションで説明します。

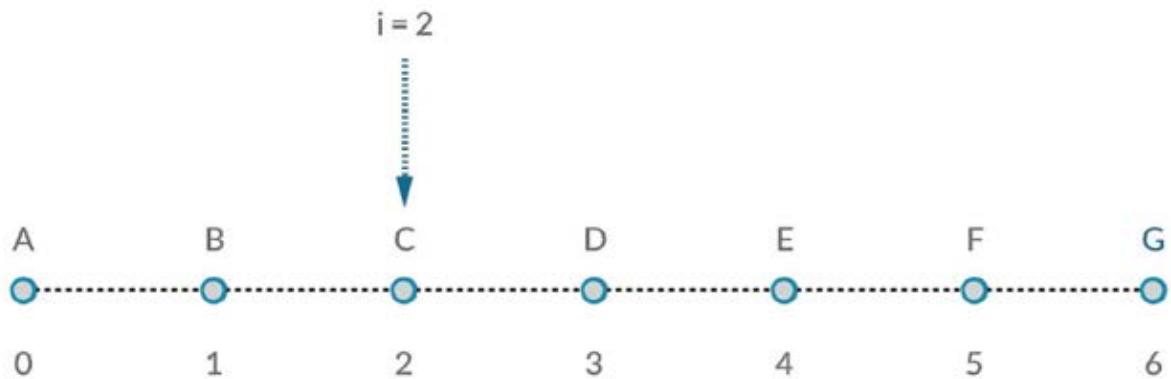
### 演習 - List.Count ノード

この演習に付属しているサンプルファイルをダウンロードしてください(右クリックして[名前を付けてリンク先を保存...]を選択): [List-Count.dyn](#)。すべてのサンプルファイルの一覧については、付録を参照してください。



1. **List.Count** ノードは、**Line.ByStartPointEndPoint** ノード内の線分の数を返します。この場合は 10 という値が返されますが、これは、元の **Code Block** ノードで作成した点の数に対応しています。

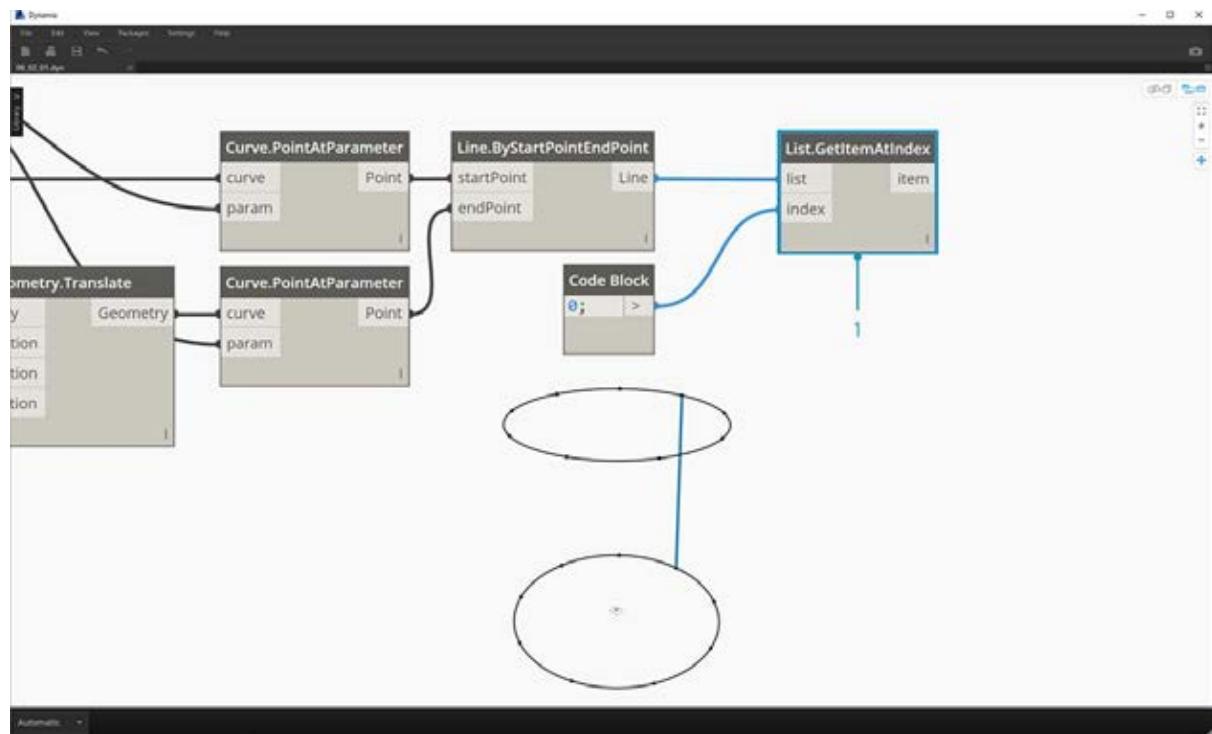
## List.GetItemAtIndex



*List.GetItemAtIndex* ノードは、リスト内の項目のクエリーを実行するための基本的な方法です。上の図の場合、「2」というインデックス値を使用して、「C」というラベルが付いている点のクエリーを実行します。

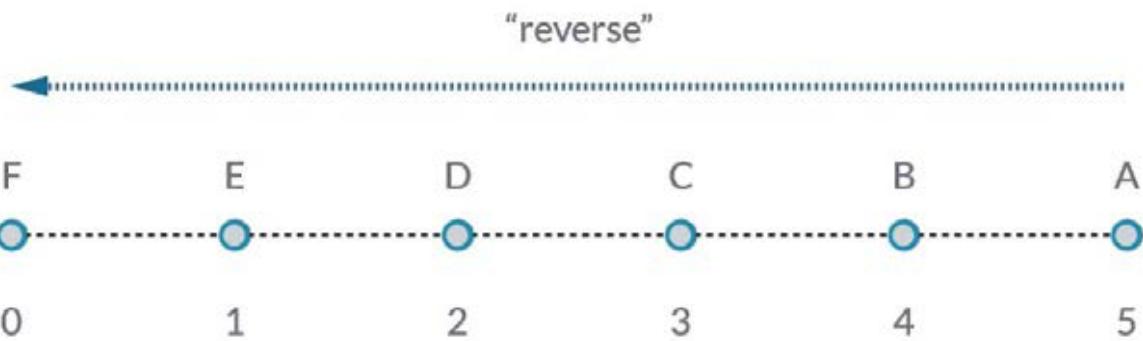
#### 演習 - List.GetItemAtIndex ノード

この演習に付属しているサンプル ファイルをダウンロードしてください(右クリックして[名前を付けてリンク先を保存...]を選択):  
[List-GetItemAtIndex.dyn](#)。すべてのサンプル ファイルの一覧については、付録を参照してください。



1. *List.GetItemAtIndex* ノードを使用して、インデックス値「0」を選択するか、線分のリスト内の先頭の項目を選択します。
2. *Watch3D* ノードに、上記で選択した 1 本の線分が表示されます。上の図のように表示するには、*Line.ByStartPointEndPoint* ノードのプレビューを無効にする必要があります。

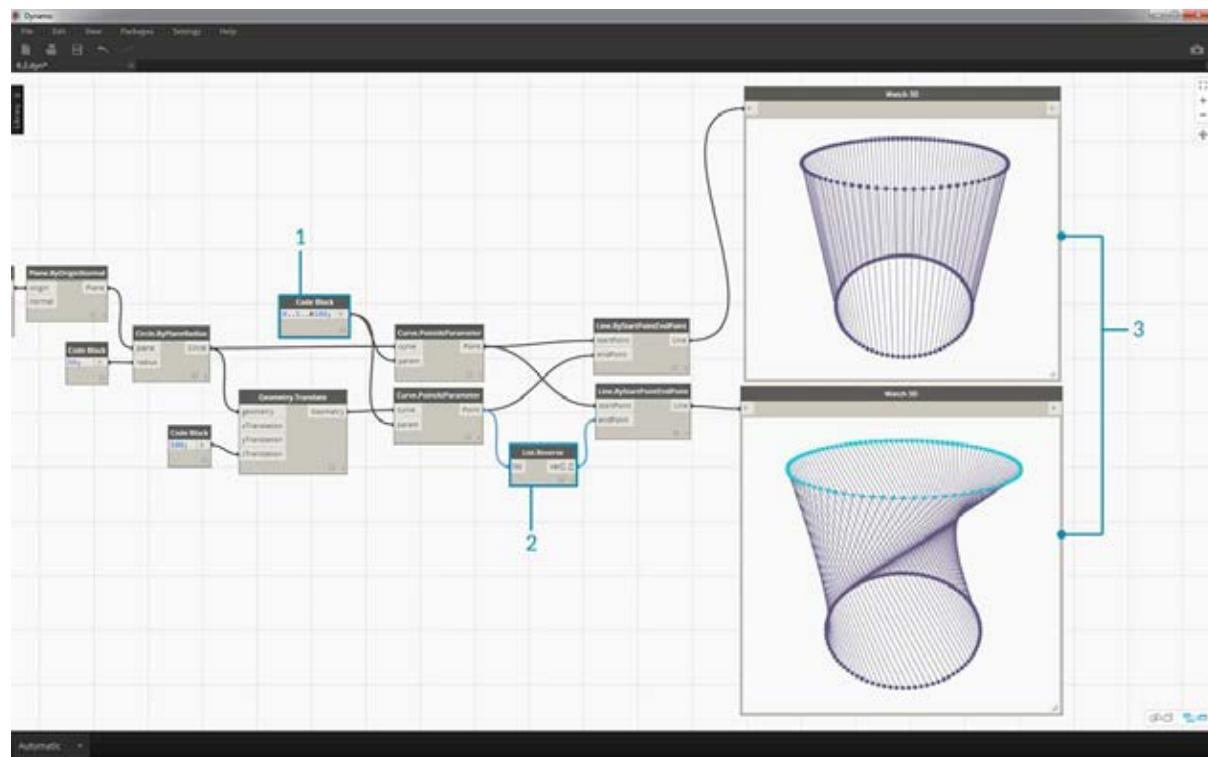
#### List.Reverse ノード



*List.Reverse* ノードは、リスト内のすべての項目の順序を逆にします。

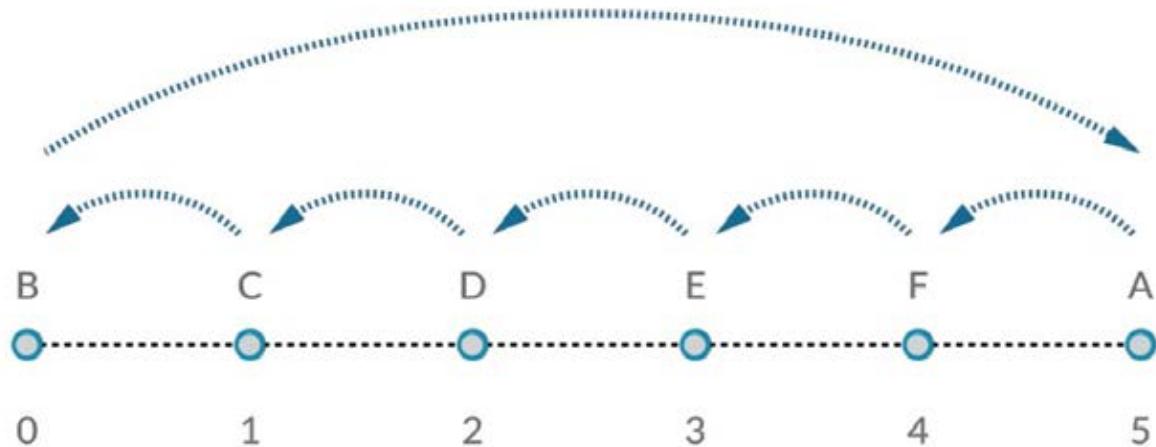
#### 演習 - **List.Reverse** ノード

この演習に付属しているサンプル ファイルをダウンロードしてください(右クリックして[名前を付けてリンク先を保存...]を選択): [List-Reverse.dyn](#)。すべてのサンプル ファイルの一覧については、付録を参照してください。



- 順序が逆になった線分のリストをわかりやすく表示するために、コード ブロックを `0..1..#100;` に変更して線分の数を増やします。
- いずれかの点のリストで、*Curve.PointAtParameter* ノードと *Line.ByStartPointEndPoint* ノードの間に *List.Reverse* ノードを挿入します。
- Watch3D* ノードに、2 つの異なる結果が表示されます。一方のノードには、リストが反転されていない結果が表示されます。それぞれの線分が、対応する点に対して垂直に接続されています。もう一方のノードには、反転していないリストとは逆の順序で、すべての点が接続されます。

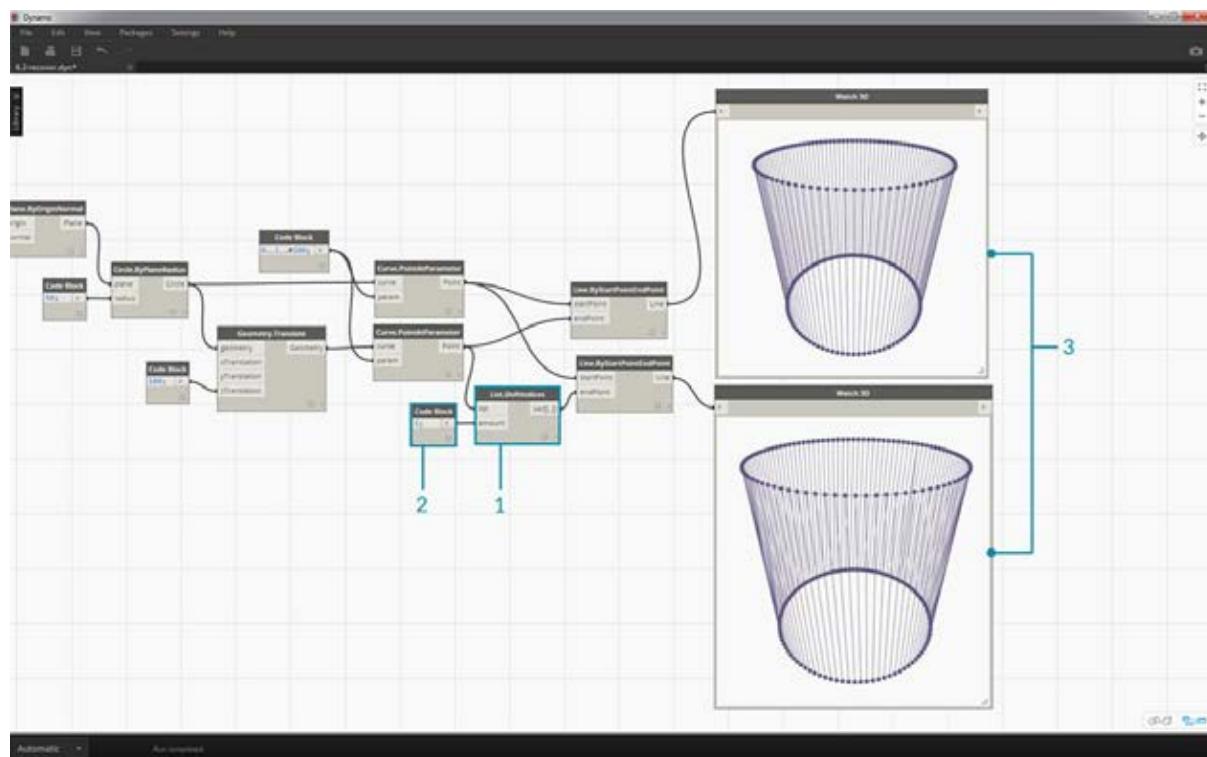
#### **List.ShiftIndices** ノード



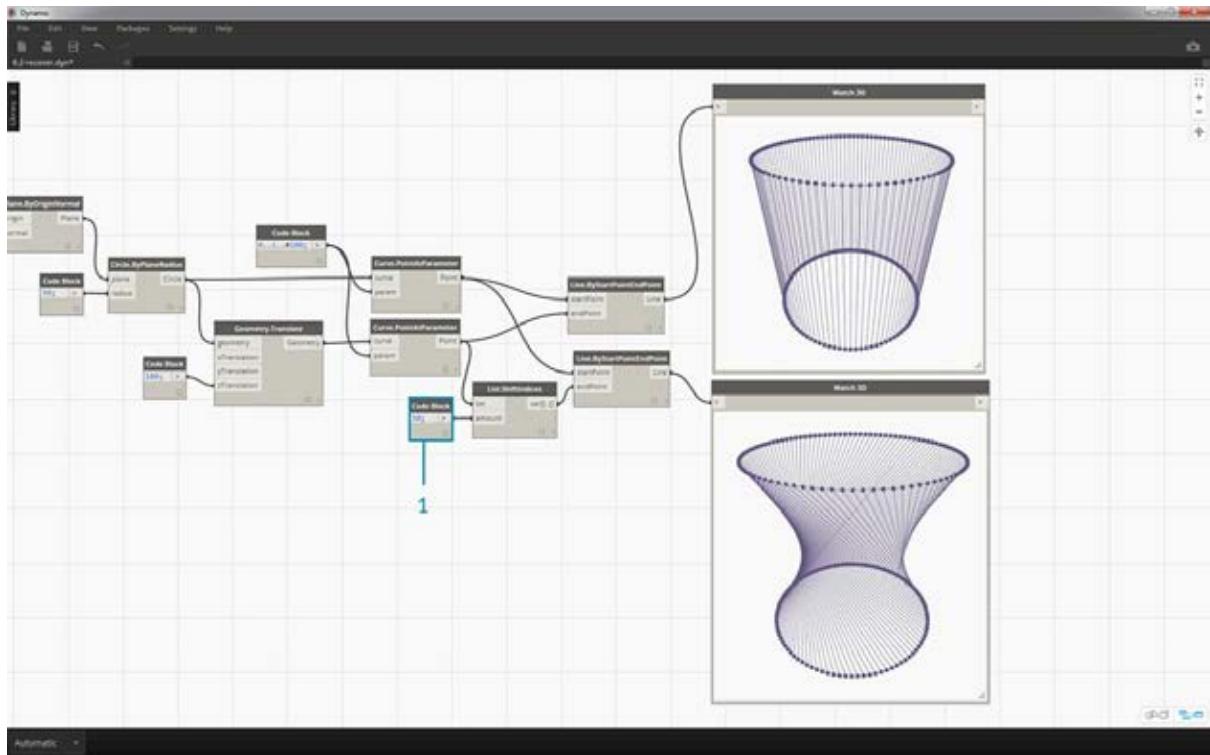
*List.ShiftIndices* ノードは、ねじれパターンやらせんパターンなどを作成する場合に便利なノードです。このノードは、指定されたインデックス値の分だけ、リスト内の項目を移動します。

#### 演習 - List.ShiftIndices ノード

この演習に付属しているサンプル ファイルをダウンロードしてください(右クリックして[名前を付けてリンク先を保存...]を選択):  
[List-ShiftIndices.dyn](#)。すべてのサンプル ファイルの一覧については、付録を参照してください。

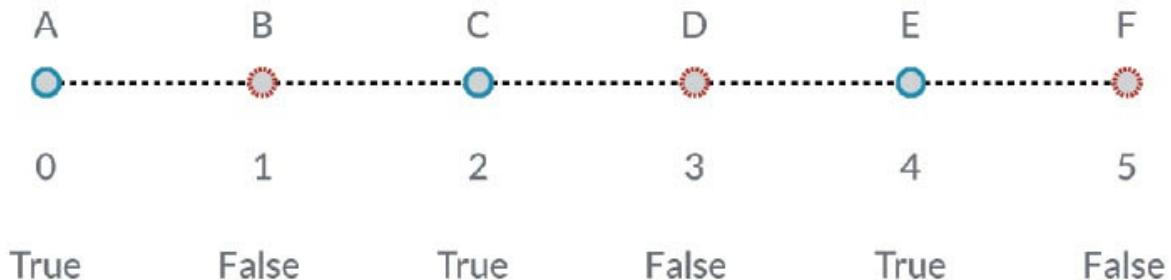


1. List.Reverse ノードの場合と同様に、List.ShiftIndices ノードを Curve.PointAtParameter ノードと Line.ByStartPointEndPoint ノードの間に挿入します。
2. Code Block ノードを使用して、リストを移動するインデックス値として「1」を指定します。
3. この場合の変化はわずかですが、下部に表示されている方の Watch3D ノード内のすべての線分が、インデックス 1 つ分だけ横にずれた点に接続されています。



1. *Code Block* ノードの値を「30」などの大きな値に変更すると、すべての線分が大きく変化することがわかります。元の円柱形状がこのように変化する動作は、カメラの絞りによく似ています。

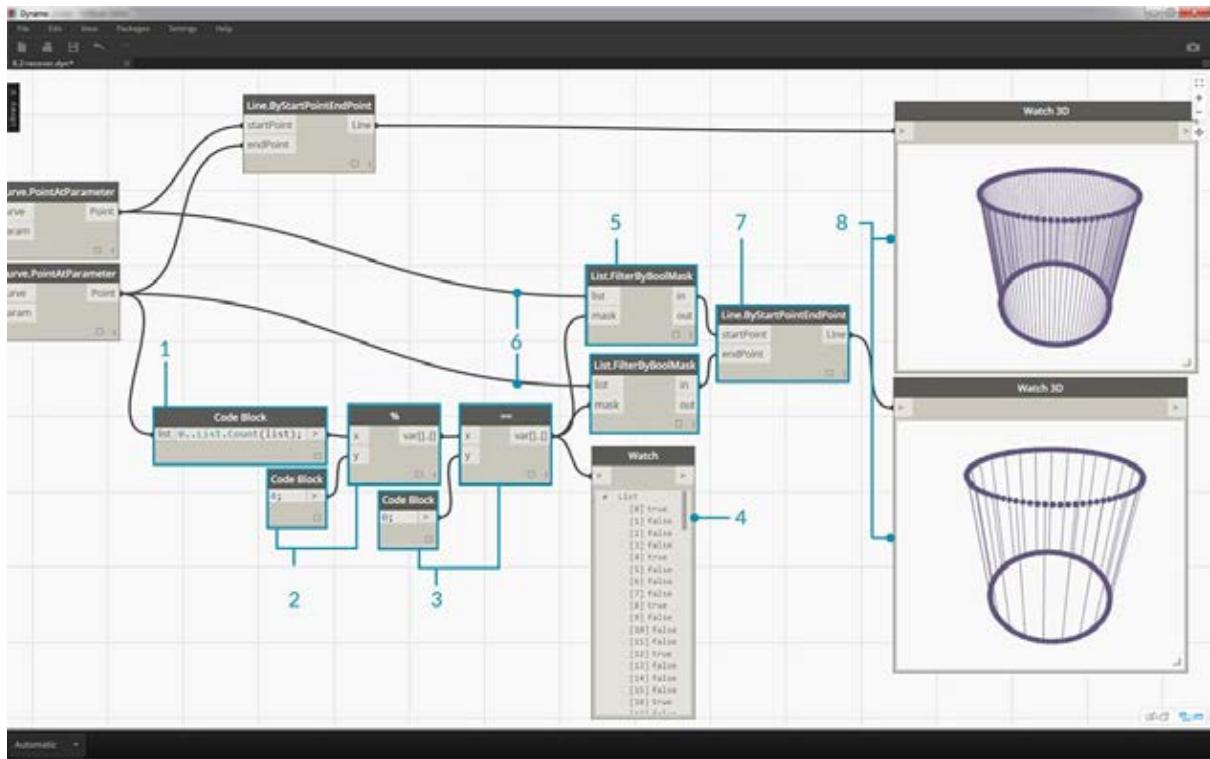
#### List.FilterByBooleanMask ノード



*List.FilterByBooleanMask* ノードは、ブール値のリストに基づいて、特定の項目を削除します。ブール値とは、「True」または「False」を読み取る値のことです。

#### 演習 - List.FilterByBooleanMask ノード

この演習に付属しているサンプル ファイルをダウンロードしてください(右クリックして[名前を付けてリンク先を保存...]を選択):  
[List-FilterByBooleanMask.dyn](#)。すべてのサンプル ファイルの一覧については、付録を参照してください。



「true」または「false」を読み取る値のリストを作成するには、次の手順を実行します。

1. *Code Block* ノードを使用して、`0..List.Count(list);` という構文の式を作成します。次に、*Curve.PointAtParameter* ノードを *list* 入力に接続します。この設定については、コード ブロックの章で詳しく説明します。ここでは、上記の構文を使用すると、*Curve.PointAtParameter* ノードの各インデックス値を表すリストが作成されるということだけを覚えてください。
2. 剰余演算を処理する「%」ノードを使用して、*Code Block* ノードの出力を *x* 入力に接続し、4 という値を *y* 入力に接続します。この操作により、インデックス値のリストを 4 で割った場合の余りが取得されます。剰余演算を処理する「%」ノードは、パターンを作成する場合に非常に便利なノードです。たとえば、整数を 4 で割った場合の余りは、0、1、2、3 のいずれかになります。
3. 「%」ノードの値が 0 になっている場合、インデックス値は 4 で割り切れる数値(0、4、8 など)であるということがわかります。「==」ノードを使用して、インデックス値を「0」という値に対してテストすると、そのインデックス値が割り切れる数値であるかどうかを判断することができます。
4. *Watch* ノードには、「true, false, false, false...」というパターンでテストの結果が表示されます。
5. この true/false のパターンを使用して、2 つの *List.FilterByBooleanMask* ノードの *mask* 入力に接続します。
6. *Curve.PointAtParameter* ノードを、2 つの *List.FilterByBooleanMask* ノードの *list* 入力に接続します。
7. *List.FilterByBooleanMask* の出力が *in* と *out* から読み取られます。*in* は、「true」というマスク値を持つ値を表し、*out* は、「false」というマスク値を持つ値を表します。*in* 出力を *Line.ByStartPointEndPoint* ノードの *startPoint* 入力と *endPoint* 入力に接続すると、フィルタされた線分のリストが作成されます。
8. 下部に表示されている方の *Watch3D* ノードに、線分の数が点の数よりも少ない円柱が表示されます。フィルタ処理で true の値だけに絞り込んだため、ノードの 25% だけが選択されました。

# リストのリスト

## リストのリスト

ここでは、階層にもう1つの層を追加してみましょう。「リストの操作」の章で例として取り上げたトランプを、ここでも例として使用します。たとえば、複数のデッキ(デッキとは1組のトランプのこと)が含まれたボックスを作成したとすると、そのボックスはデッキを格納するリストで、各デッキはカードのリストということになります。これが、「リストのリスト」という考え方です。たとえば次の図では、赤いボックスに硬貨の山のリストが入っていて、それぞれの山には硬貨のリストが含まれています。



写真: [Dori](#)

リストのリストに関する質問(クエリー)を作成する場合、次のような質問が考えられます。これにより、リストの特性(プロパティ)がわかります。

- Q: 硬貨は何種類? A: 4種類
- Q: 各硬貨の価値は? A: \$0.01 と \$0.25
- Q: 25セント硬貨の材質は? A: 銅 75%、ニッケル 25%
- Q: 1セント硬貨の材質は? A: 亜鉛 97.5%、銅 2.5%

リストのリストに対して実行できる操作(アクション)としては、次のような操作が考えられます。この場合、実行する操作に応じてリストのリストが変化します。

- 25セント硬貨または1セント硬貨の特定の山を選ぶ。
- 特定の25セント硬貨または1セント硬貨を選ぶ。
- 25セント硬貨の山と1セント硬貨の山の配置を変える。
- 25セント硬貨の山と1セント硬貨の山をすべて入れ替える。

前の章でも説明したように、Dynamoの各ノードを使用して、上記の操作に似た操作を実行することができます。ここでは、物理的な物体ではなく抽象的なデータを操作するため、データ階層を上下に移動する方法をコントロールするための一連のルールが必要になります。

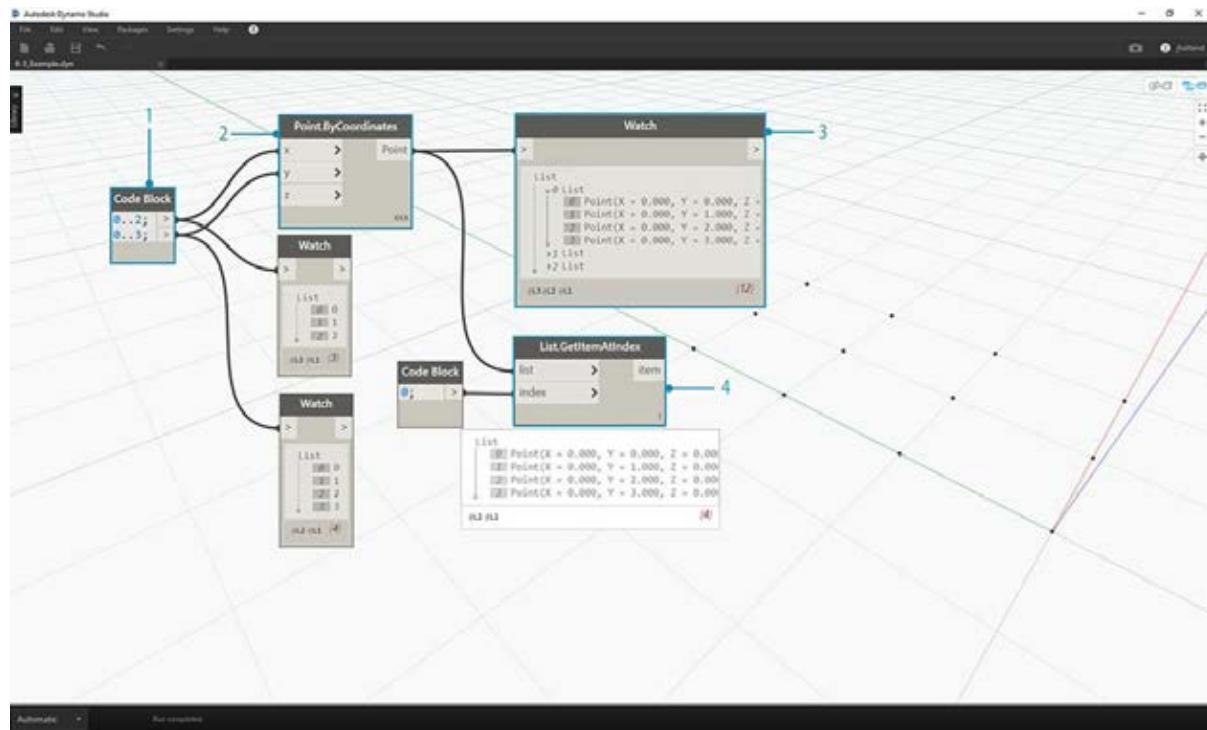
リストのリストを操作する場合、データは複雑な階層構造になっていますが、この複雑な構造により、非常に高度なパラメータ操作を実行することができます。これ以降の各演習では、基礎的な概念といつかの操作方法を確認していきます。

## トップダウン階層

このセクションで説明する基本的な概念は、「Dynamo は、リストをオブジェクトとして処理する」ということです。トップダウン階層は、オブジェクト指向プログラミングをベースとして開発されています。Dynamo は、List.GetItemAtIndexなどのコマンドでサブ要素を選択するのではなく、データ構造内でメインリストのインデックスを選択します。リスト中で選択したその項目が、別のリストである場合もあります。ここからは、サンプルの図を使用して詳しく説明していきます。

### 演習 - トップダウン階層

この演習に付属しているサンプル ファイルをダウンロードしてください(右クリックして[名前を付けてリンク先を保存...]を選択):  
[Top-Down-Hierarchy.dyn](#)。すべてのサンプル ファイルの一覧については、付録を参照してください。



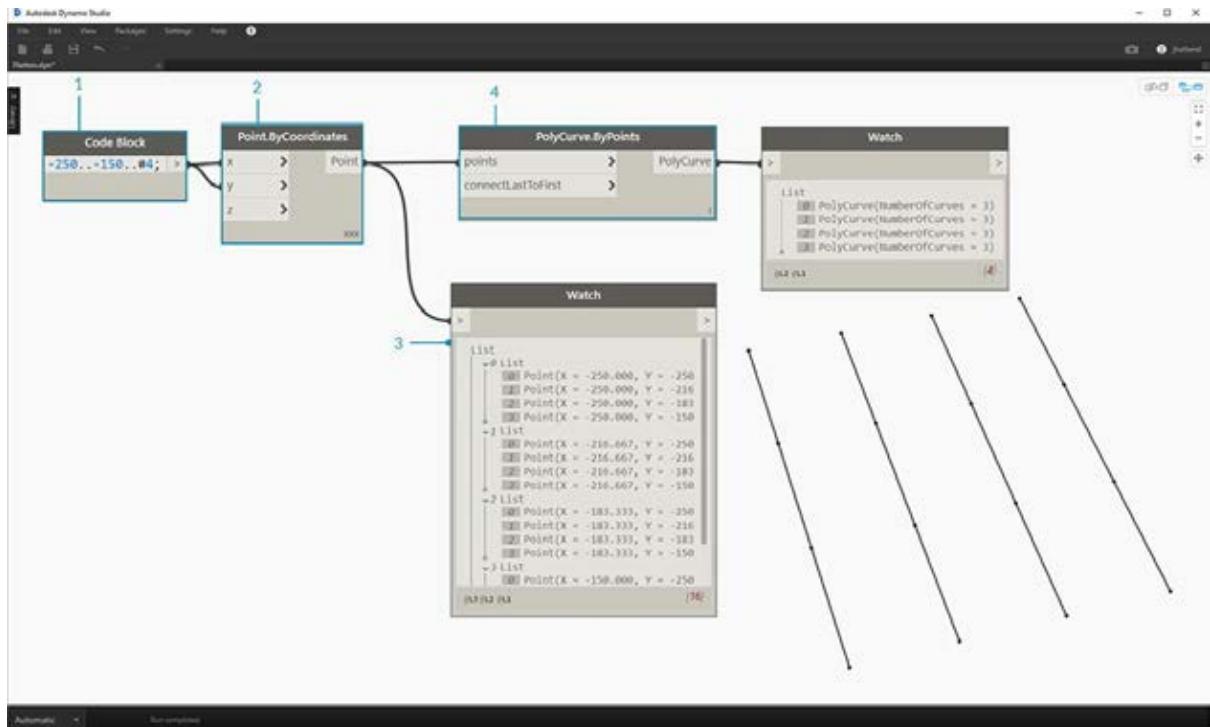
1. *Code Block* ノードを使用して、0..2; 0..3; という 2 つの範囲が定義されています。
2. これらの範囲は、レーシングが「外積」に設定された *Point.ByCoordinates* ノードに接続されています。これにより、点のグリッドが作成され、リストのリストが 出力として返されます。
3. *Watch* ノードの各リストで、4 つの項目を持つ 3 つのリストが指定されていることに注意してください。
4. インデックス値 0 を指定して *List.GetItemAtIndex* ノードを使用した場合、Dynamo は、最初のリストとそのリスト内のすべてのコンテンツを選択します。他のプログラムでは、データ構造内のすべてのリストの最初の項目だけが選択される場合がありますが、Dynamo は、トップダウン階層を使用してデータの処理を行います。

### フラット化と *List.Flatten* ノード

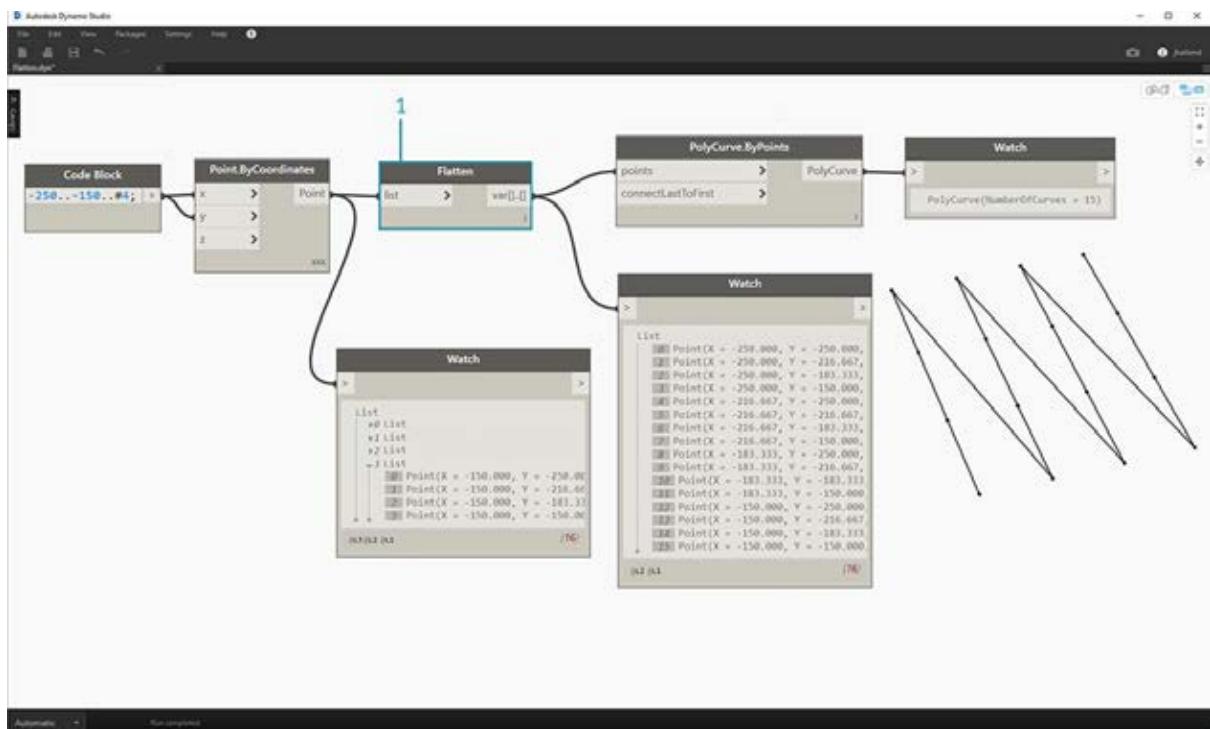
フラット化を行うと、データ構造からすべてのデータ層が削除されます。この機能は、操作にデータ階層が必要ない場合に便利ですが、情報が削除される可能性があります。次の図は、データのリストをフラット化した結果を示しています。

### 演習 - フラット化

この演習に付属しているサンプル ファイルをダウンロードしてください(右クリックして[名前を付けてリンク先を保存...]を選択):  
[Flatten.dyn](#)。すべてのサンプル ファイルの一覧については、付録を参照してください。



1. *Code Block* ノードで `-250..-150..#4;` というコード行を挿入して範囲を定義します。
2. *Code Block* ノードを *Point.ByCoordinates* ノードの *x* 入力と *y* 入力に接続します。これにより、点のグリッドを取得するための「外積」に対するレーシングが設定されます。
3. *Watch* ノードに、リストのリストが表示されます。
4. *PolyCurve.ByPoints* ノードは、各リストを参照して個別のポリカーブを作成します。Dynamo のプレビューに 4 つのポリカーブが表示されます。それぞれのポリカーブが、グリッド内の各行を表しています。



1. *PolyCurve.ByPoints* ノードの前に *Flatten* ノードを挿入して、すべての点を含む単一のリストを作成します。  
*PolyCurve.ByPoints* ノードは、リストを参照して 1 つのカーブを作成しますが、すべての点が 1 つのリスト上に存在しているため、点のリスト全体を通過するジグザグのポリカーブが 1 つだけ作成されます。

独立したデータ層をフラット化する方法も用意されています。List.Flatten ノードを使用すると、データ層の数を定義して、最上位の階層からデータ層をフラット化することができます。このノードは、現在のワークフローに必ずしも関係しない複雑なデータ構造を操作

する場合に特に便利です。別の方法として、List.Flatten ノードを関数として List.Map ノードで使用することもできます。List.Map ノードについては、これ以降のセクションで詳しく説明します。

## 分割

パラメトリックモデリングを行う際に、既存のリストにデータ構造を追加したい場合があります。その場合、さまざまなノードを使用できますが、List.Chop ノードが最も基本的なノードです。List.Chop ノードで設定された数の項目を使用すると、1つのリストを複数のサブリストに分割することができます。

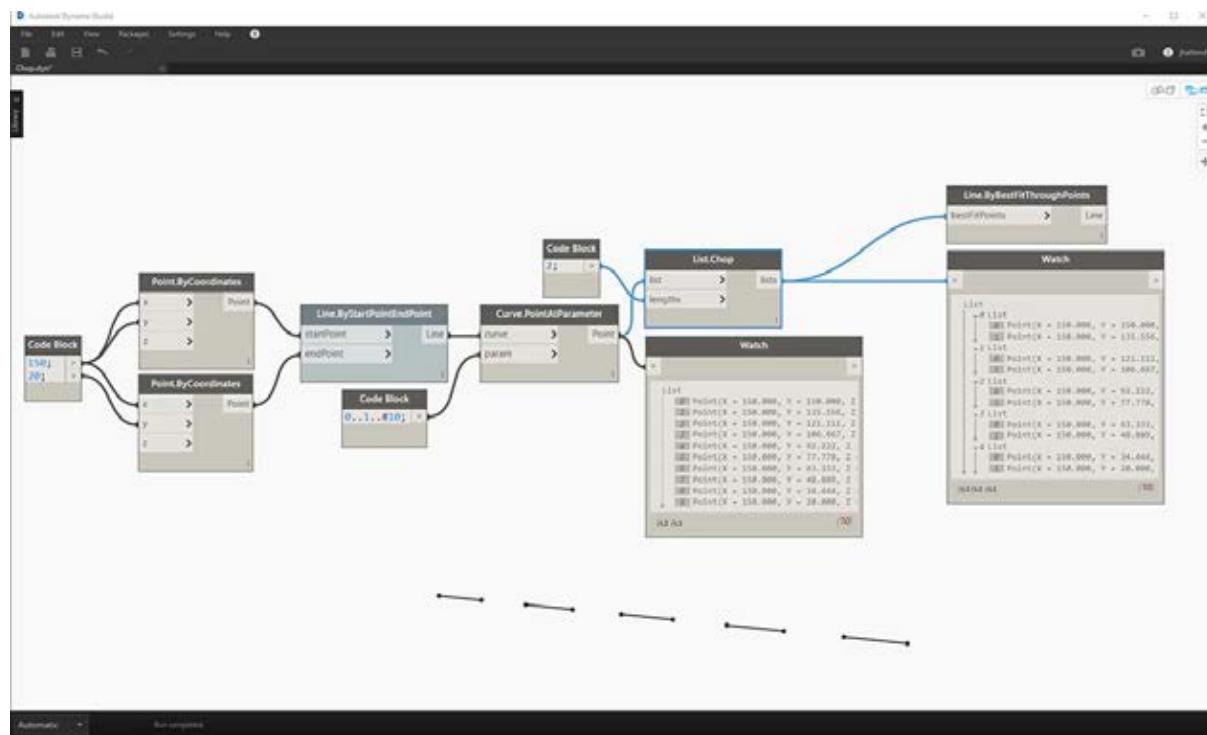
### 演習 - List.Chop ノード

この演習に付属しているサンプルファイルをダウンロードしてください(右クリックして[名前を付けてリンク先を保存...]を選択): [Chop.dyn](#)。すべてのサンプルファイルの一覧については、[付録](#)を参照してください。



List.Chop ノードの *subLength* の値を 2 に設定すると、2 つの項目を持つ 4 つのリストが作成されます。

Chop コマンドを実行すると、指定されたリストの長さに従ってリストが分割されます。List.Chop ノードは、データ構造を削除するではなく新しいデータ層をデータ構造に追加するなど、いくつかの点で List.Flatten ノードとは逆の動作を行います。List.Chop ノードは、次の例のようなジオメトリに関する操作を実行する場合に便利です。



### List.Map ノードと List.Combine ノード

List.Map ノードと List.Combine ノードは、階層内の 1 段階下の層で、設定された関数を入力リストに適用します。リストの組み合わせリストのマップはほとんど同じ操作ですが、組み合わせの場合は、特定の関数の入力に対応する複数の入力を使用できる点が、マップとは異なっています。

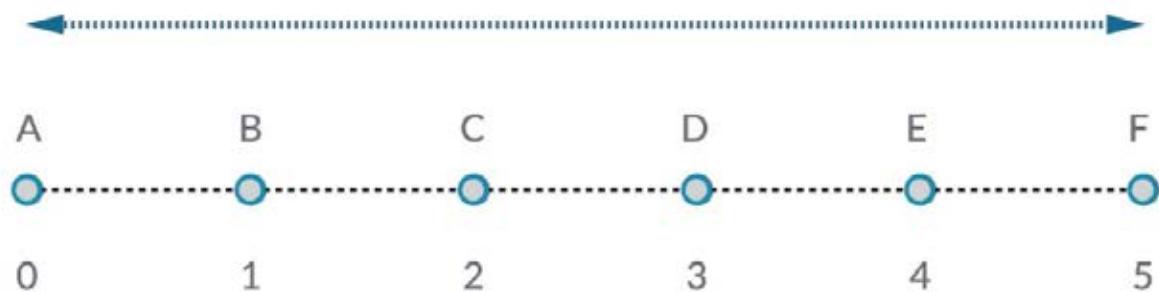
### 演習 - List.Map ノード

注: この演習は前バージョンの *Dynamo* を使用して作成されました。List.Map の機能に関する問題の大半は、List@Level 機能の追加によって解決されました。詳細については、後述の [List@Level](#) を参照してください。

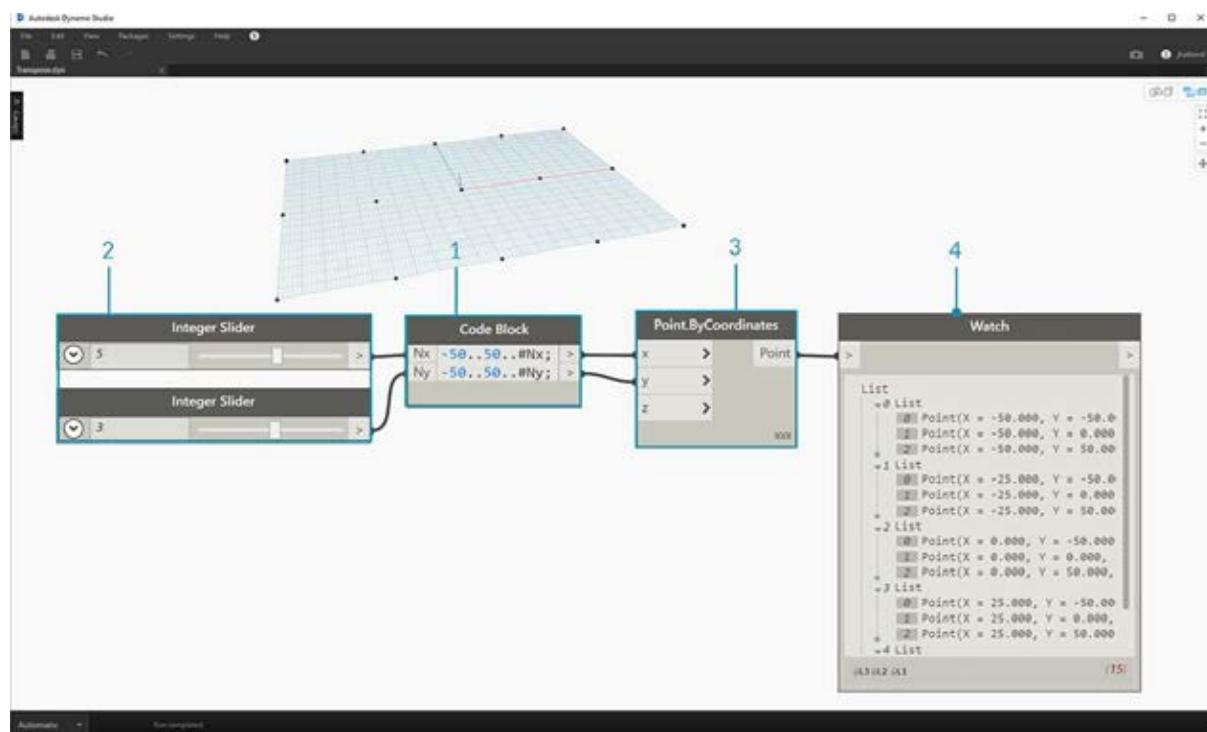
この演習に付属しているサンプルファイルをダウンロードしてください(右クリックして[名前を付けてリンク先を保存...]を選択):

[Map.dyn](#)。すべてのサンプル ファイルの一覧については、[付録](#)を参照してください。

簡単な例として、前のセクションで説明した List.Count ノードをもう一度確認してみましょう。



List.Count ノードは、リスト内のすべての項目をカウントします。ここでは、List.Count ノードを使用して、List.Map ノードの仕組みについて説明します。

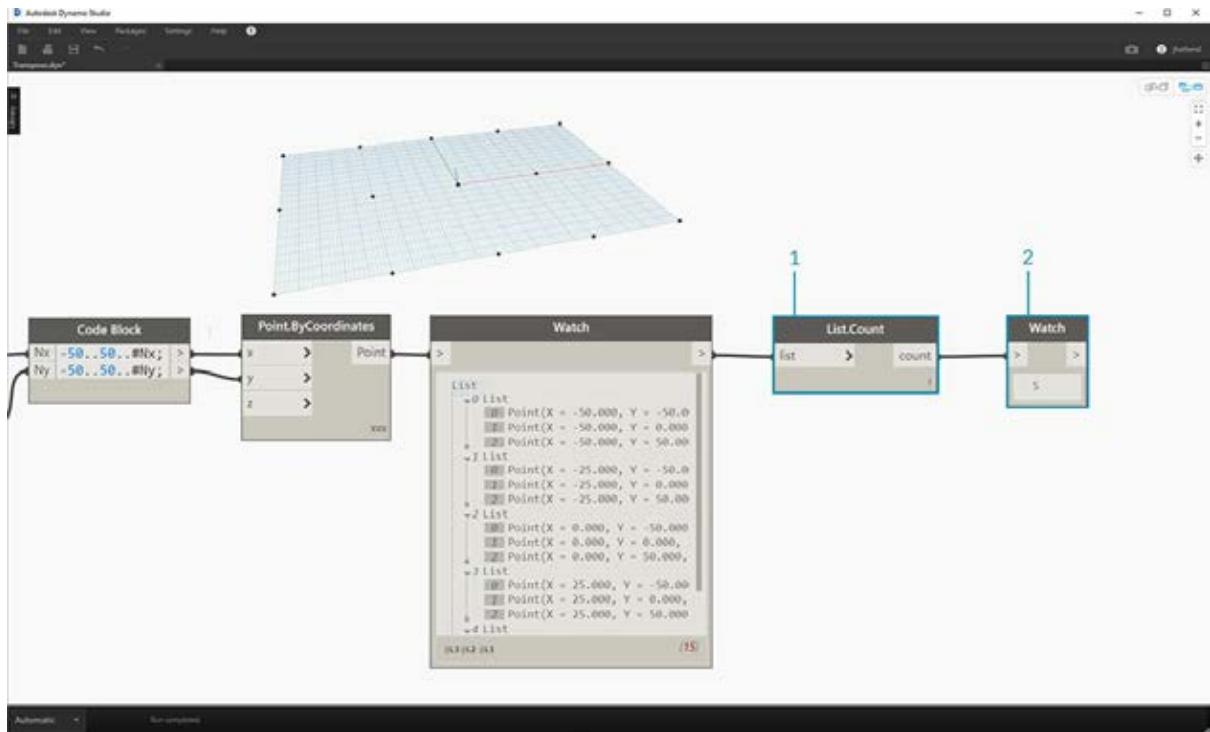


1. Code Block ノードで、次の 2 行を挿入します。

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

このコードを入力すると、Code Block ノードに Nx と Ny という 2 つの入力が作成されます。

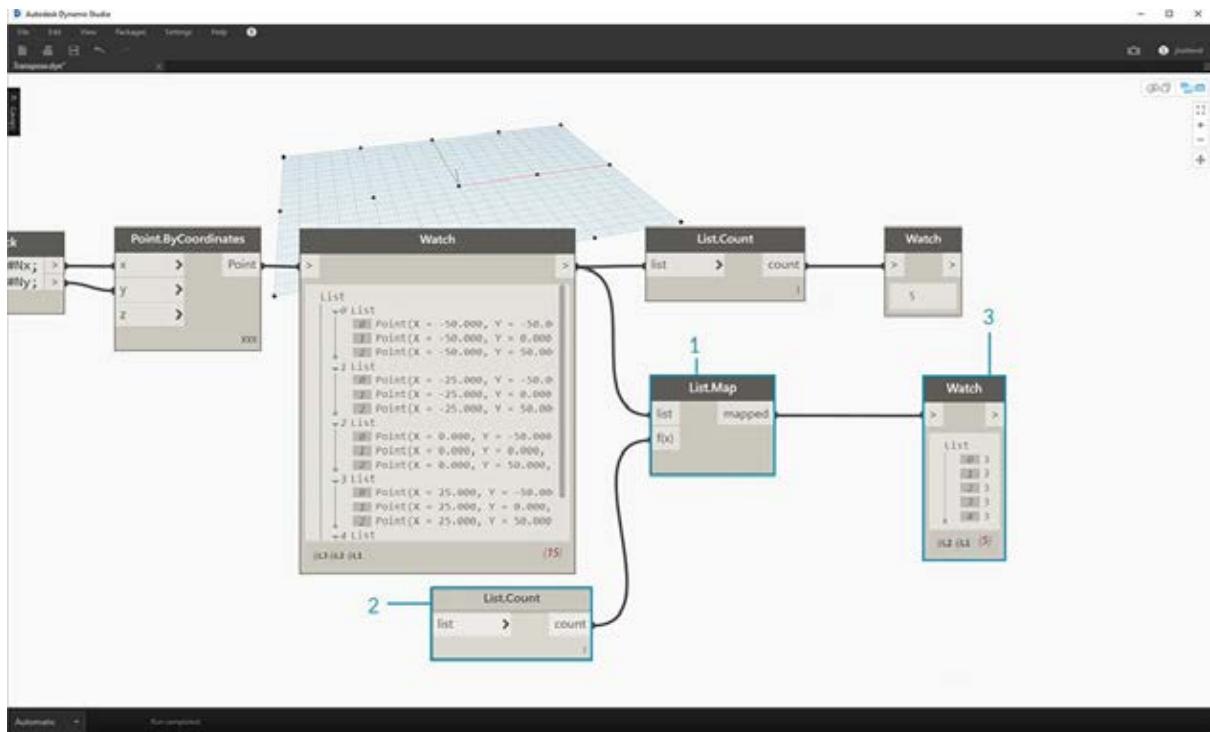
1. 2 つの Integer Slider ノードを使用して Nx の値と Ny の値を Code Block ノードに接続すると、これらの値が定義されます。
2. 次に、Code Block ノードの各行を Point.ByCoordinates ノードの x 入力と y 入力に接続します。このノードを右クリックして「レーシング」と「外積」を続けて選択します。この操作により、点のグリッドが作成されます。-50 から 50 までの範囲が定義されているため、ここでは既定の Dynamo グリッド全体を使用しています。
3. Watch ノードに、作成された点が表示されます。データ構造を確認すると、リストのリストが作成されていることがわかります。それぞれのリストが、グリッドの点の 1 行を表しています。



1. 前の手順の Watch ノードの出力に List.Count ノードを接続します。
2. 別の Watch ノードを List.Count ノードの int 出力に接続します。

List.Count ノードによって 5 という値が生成されていることに注意してください。これは、Code Block ノードで定義されている「Nx」変数と同じ値です。こうなる理由は次のとおりです。

- Point.ByCoordinates ノードは、リストを作成するための主要な入力として「x」入力を使用します。Nx の値が 5 で Ny の値が 3 の場合、3 つの項目を持つ 5 つのリストが含まれた 1 つのリストが作成されます。
- Dynamo は、リストをオブジェクトとして処理するため、List.Count ノードは、階層内のメインのリストに適用されます。その結果、メインのリストに含まれているリストの数である 5 という値が生成されることになります。



1. List.Map ノードを使用して、階層内の 1 段階下の層で「関数」を実行します。
2. List.Count ノードには入力がないことに注意してください。List.Count ノードは関数として使用されるため、このノード

は階層内の 1 段階下の層にあるすべてのリストに適用されます。List.Count ノードの空の入力は、List.Map ノードの list 入力に対応しています。

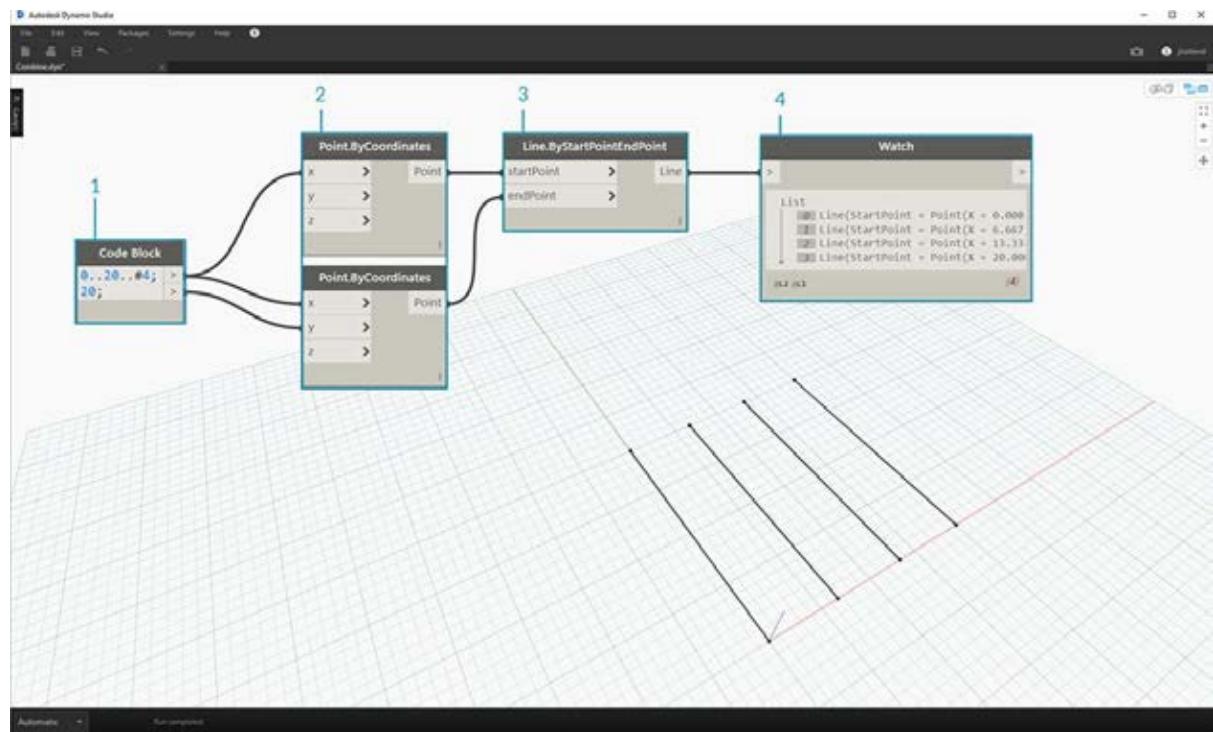
3. List.Count ノードにより、5 つの項目を持つリストが 1 つ作成されます。それぞれの項目の値は 3 になります。この値は、各サブリストの長さを表しています。

#### 演習 - List.Combine ノード

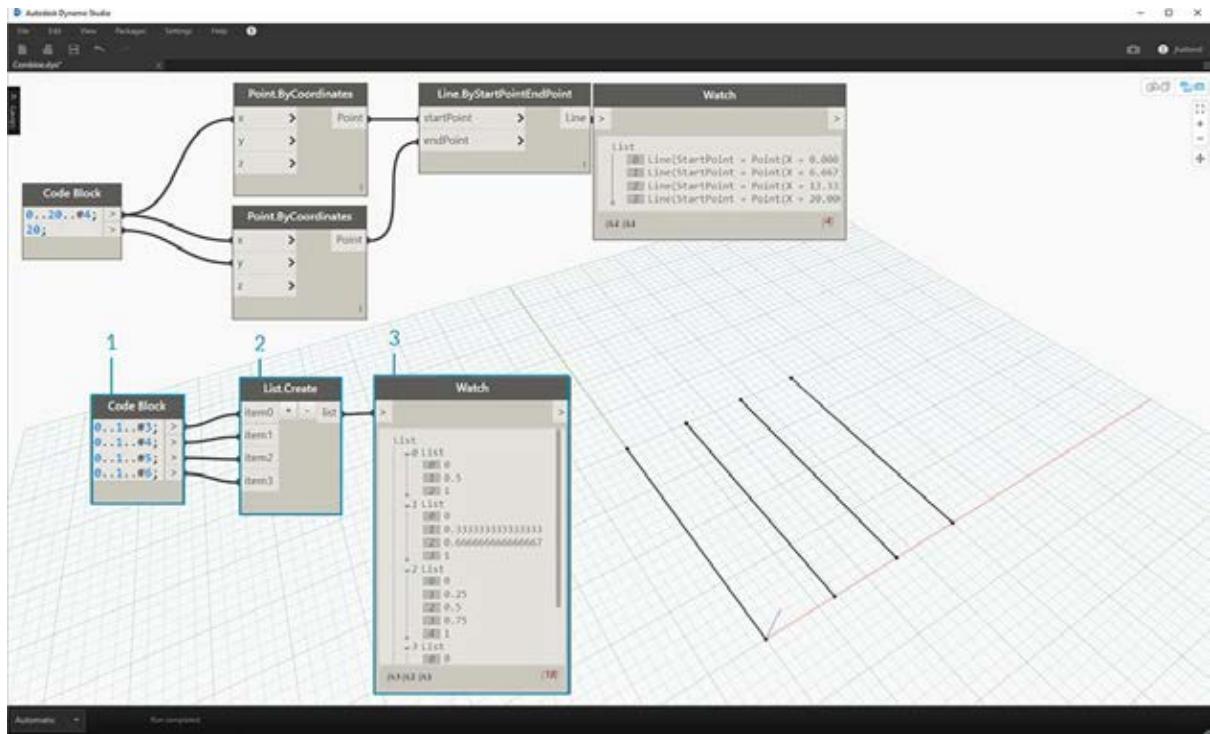
注: この演習は前バージョンの Dynamo を使用して作成されました。List.Combine の機能に関する問題の大半は、List@Level 機能の追加によって解決されました。詳細については、後述の List@Level を参照してください。\*\*\*\*

この演習に付属しているサンプルファイルをダウンロードしてください(右クリックして[名前を付けてリンク先を保存……]を選択): [Combine.dyn](#)。すべてのサンプルファイルの一覧については、[付録](#)を参照してください。

この演習では、List.Map ノードの場合と似た方法で、複数の要素を処理します。具体的には、点の固有の番号を使用して、曲線のリストを操作します。



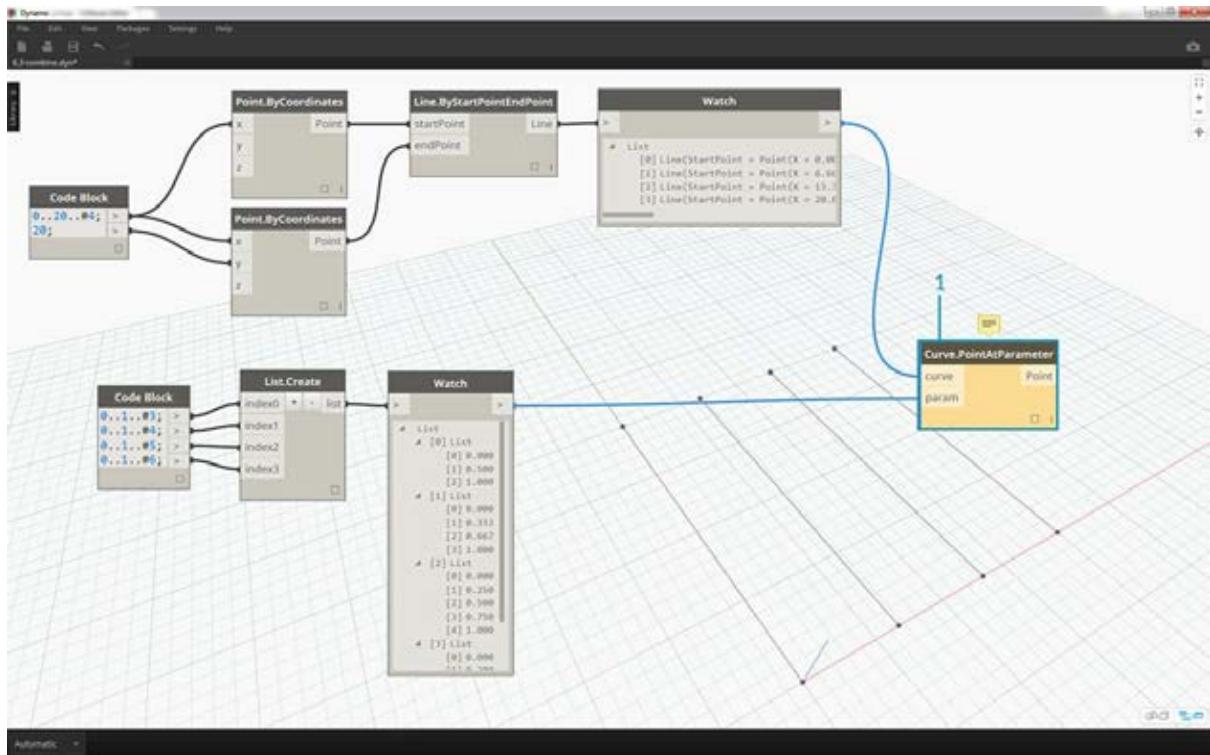
1. Code Block ノードで ..20..#4; という構文を使用し、このコード行の下で 20; という値を指定して範囲を定義します。
2. Code Block ノードを 2 つの Point.ByCoordinates ノードに接続します。
3. 2 つの Point.ByCoordinates ノードから Line.ByStartPointEndPoint ノードを作成します。
4. Watch ノードに 4 つの行が表示されます。



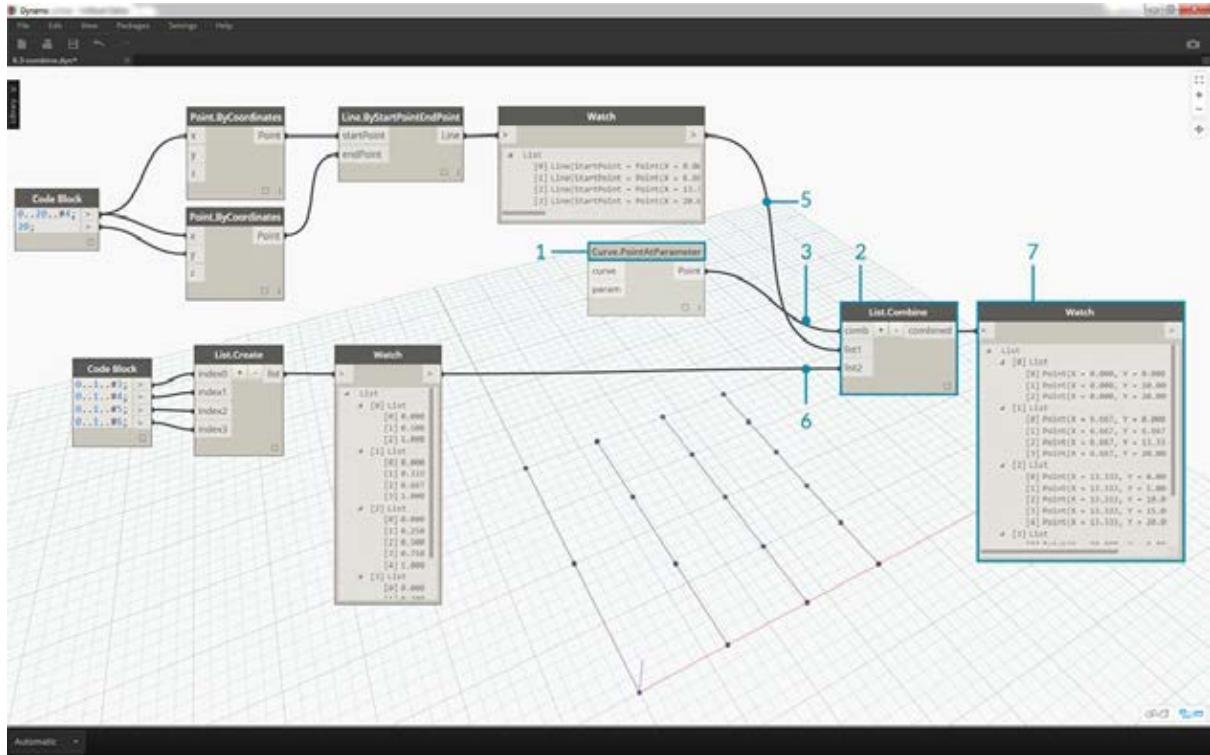
- 線分作成用グラフの下で Code Block ノードを使用して、各線分を分割するための 4 つの異なる範囲を作成します。これを行うには、次のコード行を入力します。

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

2. List.Create ノードを使用して、Code Block ノードの 4 つのコード行を 1 つのリストにマージします。
3. Watch ノードに、リストのリストが表示されます。



- 線分を直接 *parameter* の値に接続しても、*Curve.PointAtParameter* ノードは機能しません。階層内の 1 段階下の層で操作を行う必要があります。この場合、*List.Combine* ノードを使用します。



*List.Combine* ノードを使用すると、指定した範囲で各線分を正しく分割することができます。この操作は少し複雑であるため、手順を追って詳しく説明します。

- 最初に、*Curve.PointAtParameter* ノードをキャンバスに追加します。このノードが、*List.Combine* ノードに適用される \*\* 関数またはコンビネータになります。詳細については、これ以降で説明します。
- 次に、*List.Combine* ノードをキャンバスに追加し、「+」符号または「-」符号をクリックして加算または減算を実行します。ここでは、既定の 2 つの入力を *List.Combine* ノードで使用します。
- 次に、*Curve.PointAtParameter* ノードを *List.Combine* ノードの *comb* 入力に接続します。その際、*Curve.PointAtParameter* ノードの *param* 入力を右クリックし、[既定値を使用]オプションの選択を解除してください。ノードを関数として実行する場合は、Dynamo 入力の既定値を削除する必要があります。つまり、既定値とは、入力ポートに追加のノードが接続されているものと考えてください。そのため、ここでは既定値を削除する必要があります。
- 点を作成するには 2 つの入力(線分とパラメータ)が必要ですが、それらの入力をどのように方法と順序で *List.Combine* の入力に接続すればよいでしょうか。
- Curve.PointAtParameter* の空の入力を、上から下の順序でコンビネータに入力する必要があります。この操作により、各線分が *List.Combine* ノードの *list1* 入力に接続されます。
- 同様に、各パラメータの値が *List.Combine* ノードの *list2* 入力に接続されます。
- Watch ノードと Dynamo のプレビューに、*Code Block* ノードで指定した範囲に基づいて分割された 4 つの線分が表示されます。

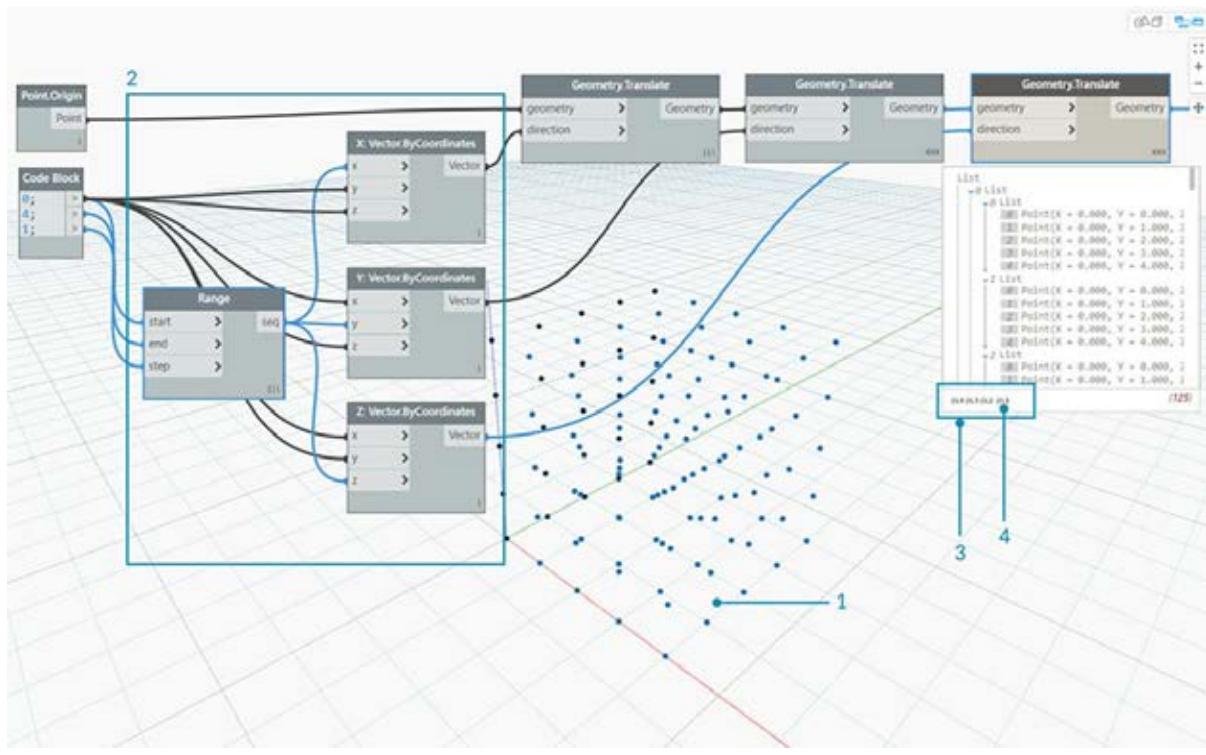
## List@Level

*List.Map* ではなく *List@Level* 機能を使用すると、ノードの入力ポートに使用するリストのレベルを直接選択することができます。この機能は、入力ポートに指定されたあらゆる入力値に使用できます。使用すると、もっともすばやく簡単にリストのレベルにアクセスすることができます。ノードの入力として使用するリストのレベルを指定するだけで、後はすべてをノードに任せることができます。

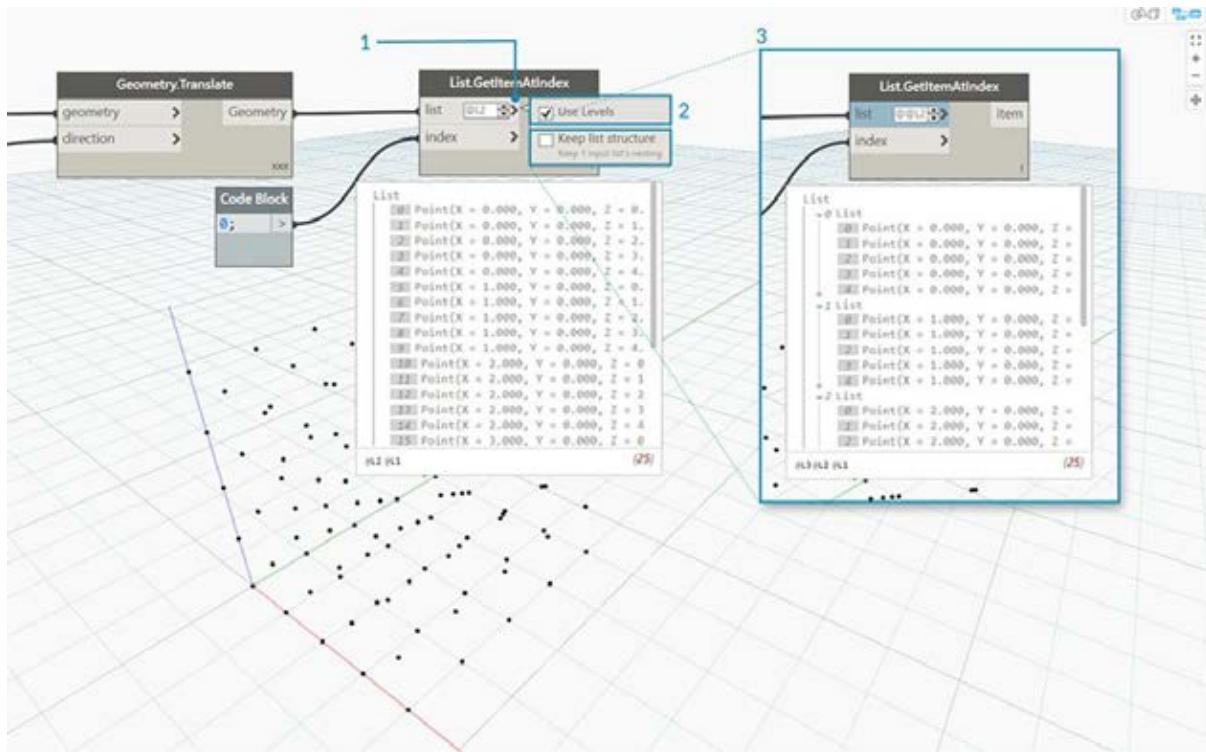
### List@Level の演習

この演習では、*List@Level* 機能を使用して特定のデータ レベルを分離してみましょう。

この演習に付属しているサンプル ファイルをダウンロードしてください(右クリックして[名前を付けてリンク先を保存...]を選択): [List@Level](#)。すべてのサンプルファイルの一覧については、[付録](#)を参照してください。



1. 点によって構成される単純な 3D グリッドから始めることにします。
2. X、Y、Z に Range ノードを接続することでグリッドが構成されています。したがって、データの構造は X リスト、Y リスト、Z リストの 3 層から成ることがわかります。
3. これらの層は、異なるレベルに存在します。レベルは、プレビュー バルーンの下に表示されます。リストレベルの列はリストデータに対応しています。ここで、作業中のレベルを確認することができます。
4. リストレベルは逆順で階層化されているので、最低レベルのデータは常に「L1」にあります。これにより、上流で何かが変更されたとしても、グラフを意図したとおりに動作させることができます。

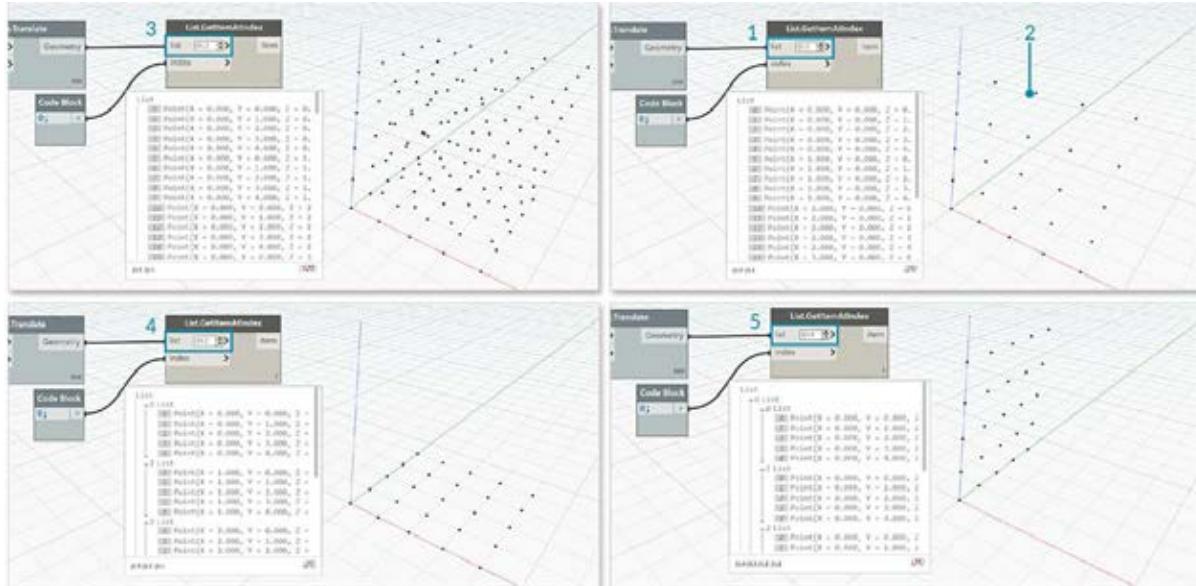


1. List@Level 機能を使用するには、「>」をクリックします。このメニュー内部では、2 つのチェックボックスが表示されます。
2. レベルを使用 - このオプションをオンにすると、List@Level の機能が有効になります。このオプションをクリックした後で、ノードで使用する入力リストレベルをすべてクリックして選択することができます。このメニューで上矢印か下矢印を

クリックすることにより、さまざまなレベル オプションを試すことができます。

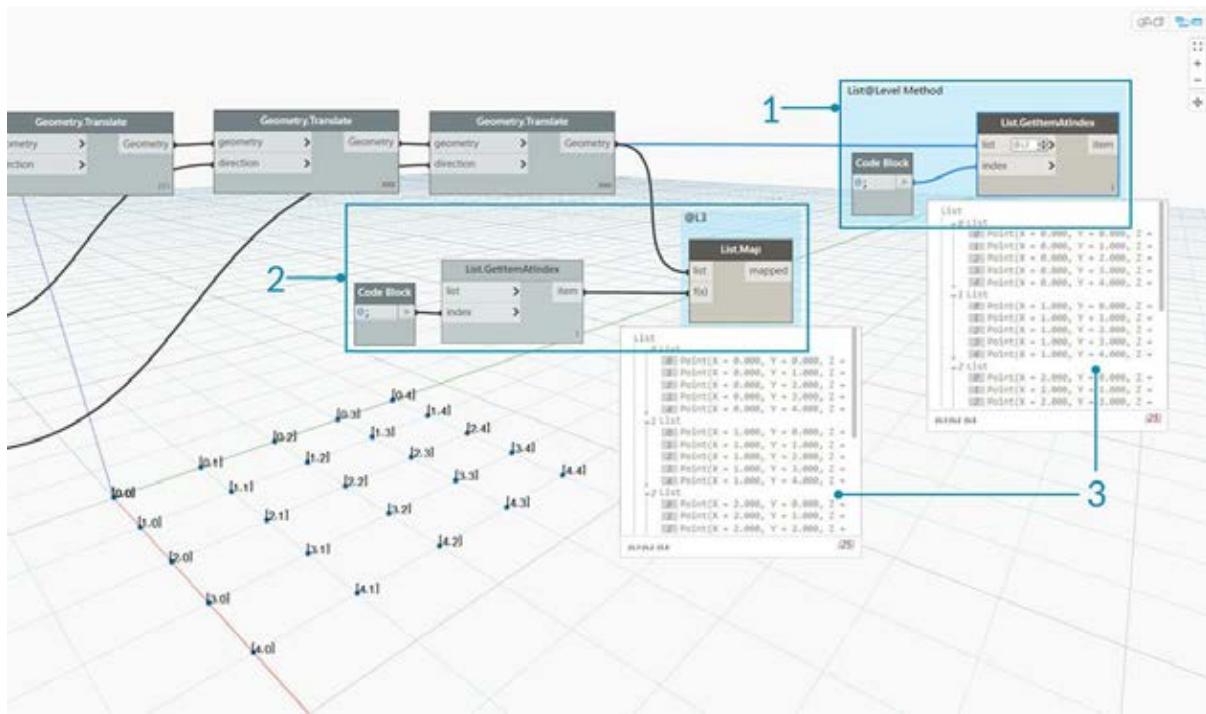
3. リスト構造を保持 - オンにしていると、その入力のレベル構造を保持することができます。意図的にデータを複数のサブリストに整理している場合は、このオプションをオンにしておくと、いかなる情報も失わずにリスト構造をそのまま保持しておくことができます。

リストレベル全体を切り替えることにより、単純な 3D グリッド上でリスト構造を視覚的に確認することができます。リストレベルとインデックスの組み合わせを変更すると、元の 3D セットとは異なる点のセットが返されます。



1. DesignScript で「@L2」を選択すると、レベル 2 のリストのみを選択することができます。
2. index に 0 が入力されているとき、レベル 2 のリストには Y を 0 とする点のセットのみが含まれているので、XZ のグリッド面のみが返されます。
3. レベル フィルタを「@L1」に変更すると、リストレベル 1 に含まれるすべての点を確認することができます。index に 0 が入力されているとき、レベル 1 のリストにはフラットリストの 3D ポイントがすべて含まれています。
4. 同様に「@L3」に変更すると、リストレベル 3 の点のみが表示されます。index に 0 が入力されているとき、レベル 3 のリストには Z を 0 とする点のセットのみが含まれているので、XY のグリッド面のみが返されます。
5. 同様に「@L4」に変更すると、リストレベル 4 の点のみが表示されます。index に 0 が入力されているとき、レベル 4 のリストには X を 0 とする点のセットのみが含まれているので、YZ のグリッド面のみが返されます。

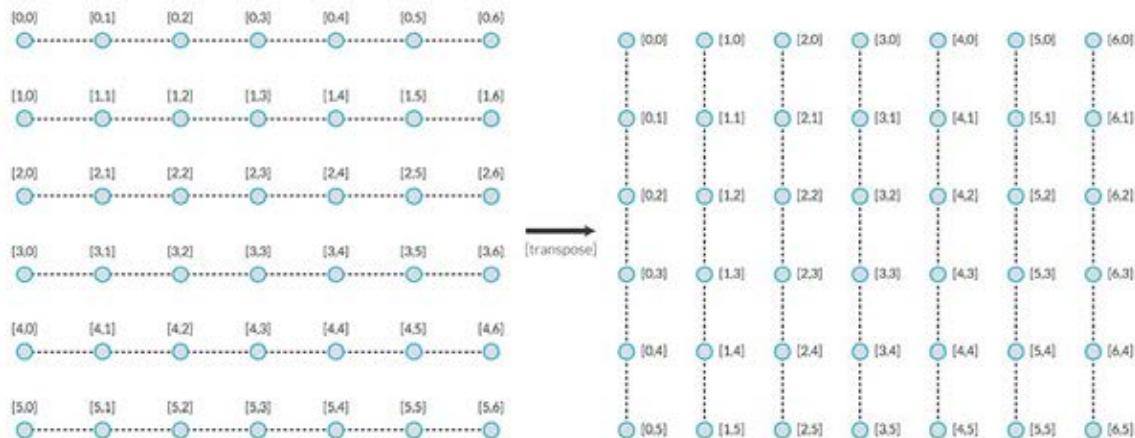
今回の例は List.Map でも作成できますが、List@Level を使用すると、動作を単純化して簡単にノード データにアクセスできるようになります。次に List.Map と List@Level の比較をご覧ください。



- どちらの方法でも同じ点にアクセスすることができますが、List@Level を使用するとなつた 1 個のノードだけでデータのレイヤを簡単に切り替えることができます。
- List.Map を使用して点構成のグリッドにアクセスするには、List.Map の隣に List.GetItemAtIndex ノードを配置する必要があります。レベルの階層を下りていくたびに、List.Map ノードを追加していく必要があります。リストが複雑になればなるほど、目的の情報レベルにアクセスするには、多数の List.Map ノードをグラフに追加しなければなりません。
- この例では、List.GetItemAtIndex ノードを List.Map ノードに接続することにより、List.GetItemAtIndex で「@L3」を選択したときと同じリスト構造を持つ同じ点のセットを取得しています。

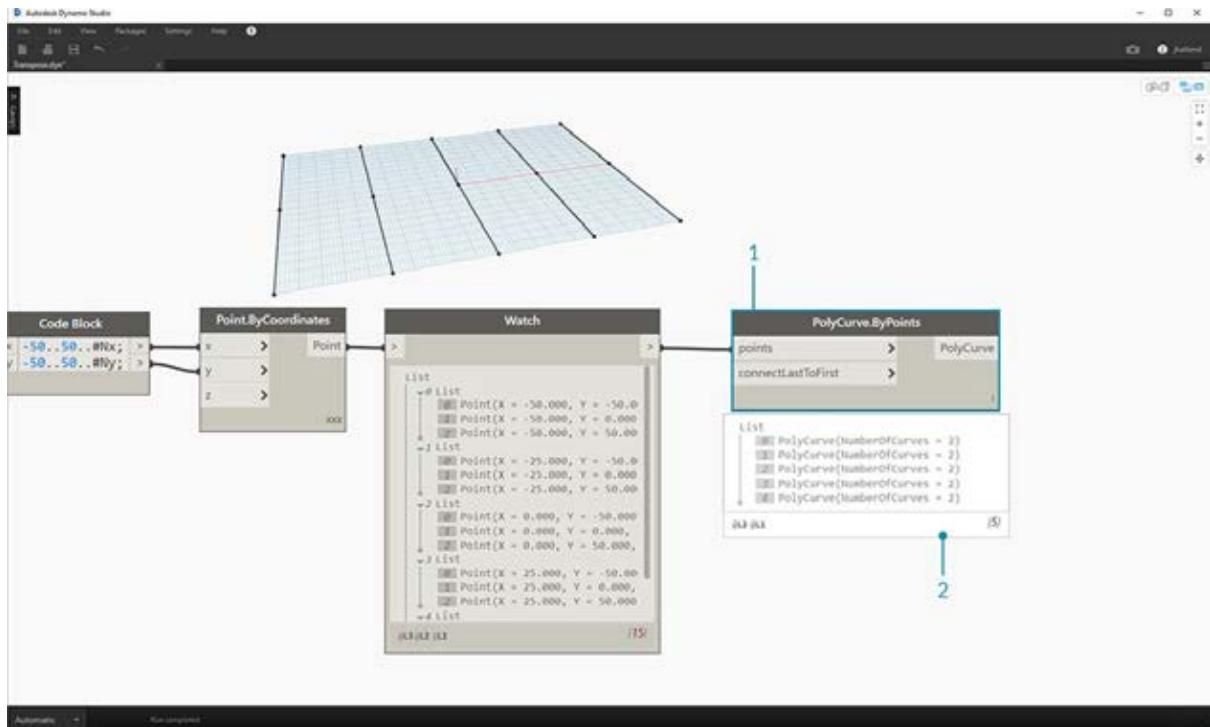
## 転置

転置は、リストのリストを操作する場合の基本的な機能です。転置を実行すると、スプレッドシートプログラムの場合と同様に、データ構造内の列と行の位置が反転します。次の基本的な行列を使用して、転置の仕組みを説明します。また、次のセクションでは、転置機能を使用して幾何学的な関係を作成する方法について説明します。



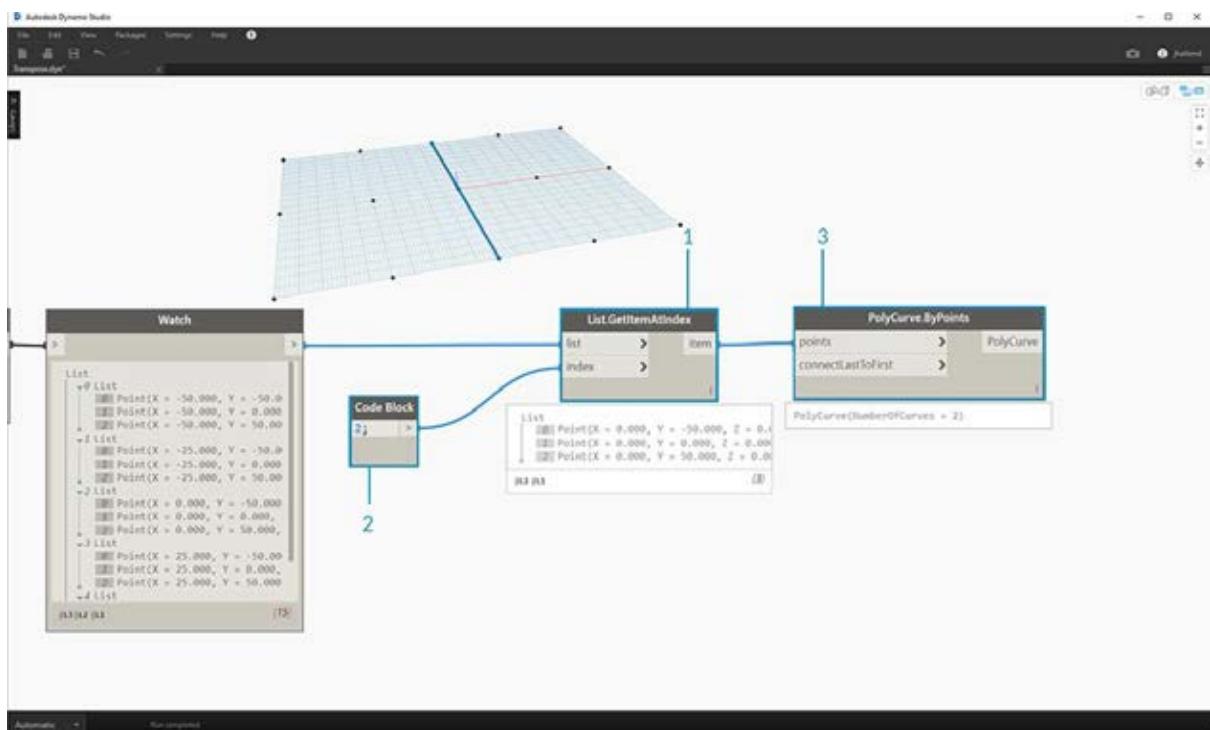
## 演習 - List.Transpose ノード

この演習に付属しているサンプルファイルをダウンロードしてください(右クリックして[名前を付けてリンク先を保存...]を選択): [Transpose.dyn](#)。すべてのサンプルファイルの一覧については、付録を参照してください。

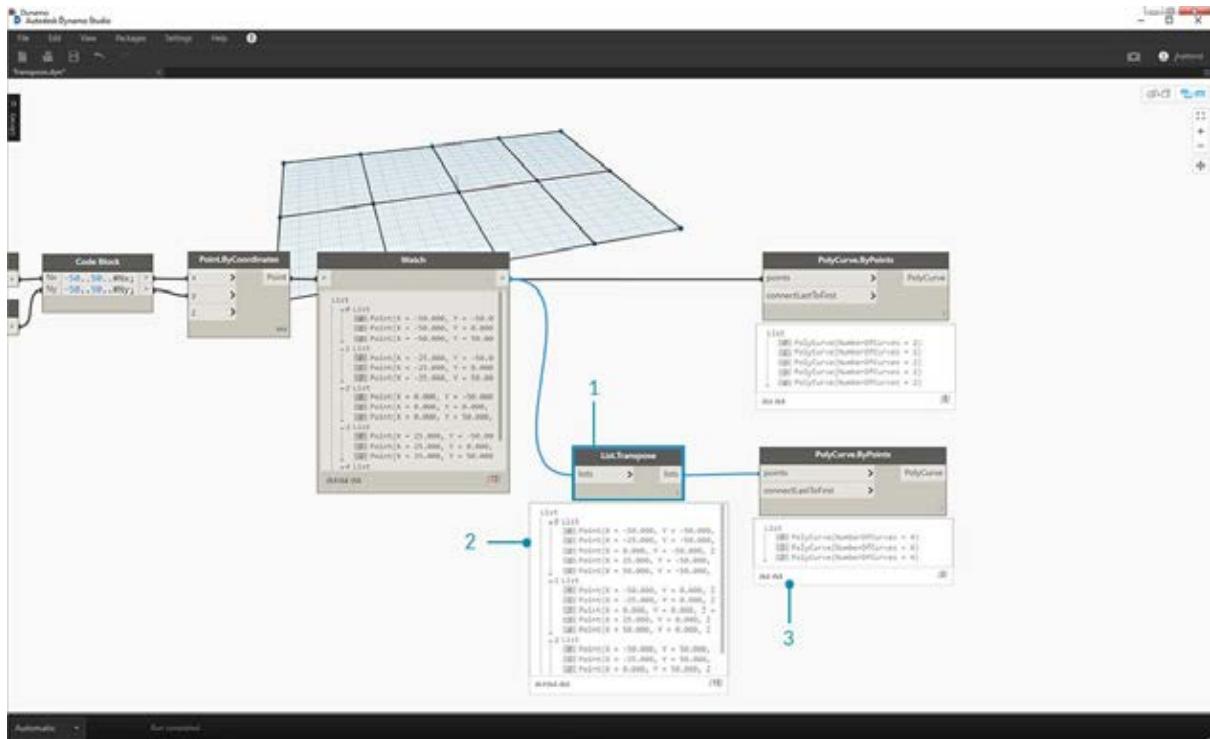


最初に、前の演習で使用した *List.Count* ノードを削除して別のジオメトリに移動し、データ構造がどのようにになっているかを確認します。

1. *Point.ByCoordinates* ノードから続く *Watch* ノードの出力に、*PolyCurve.ByPoints* ノードを接続します。
2. 出力として 5 つのポリカーブが作成され、Dynamo のプレビューにそれらのポリカーブが表示されます。Dynamo ノードは点のリスト(この場合は、点のリストのリスト)を検索し、そのリストから 1 つのポリカーブを作成します。原則として、各リストはデータ構造内で曲線に変換されます。



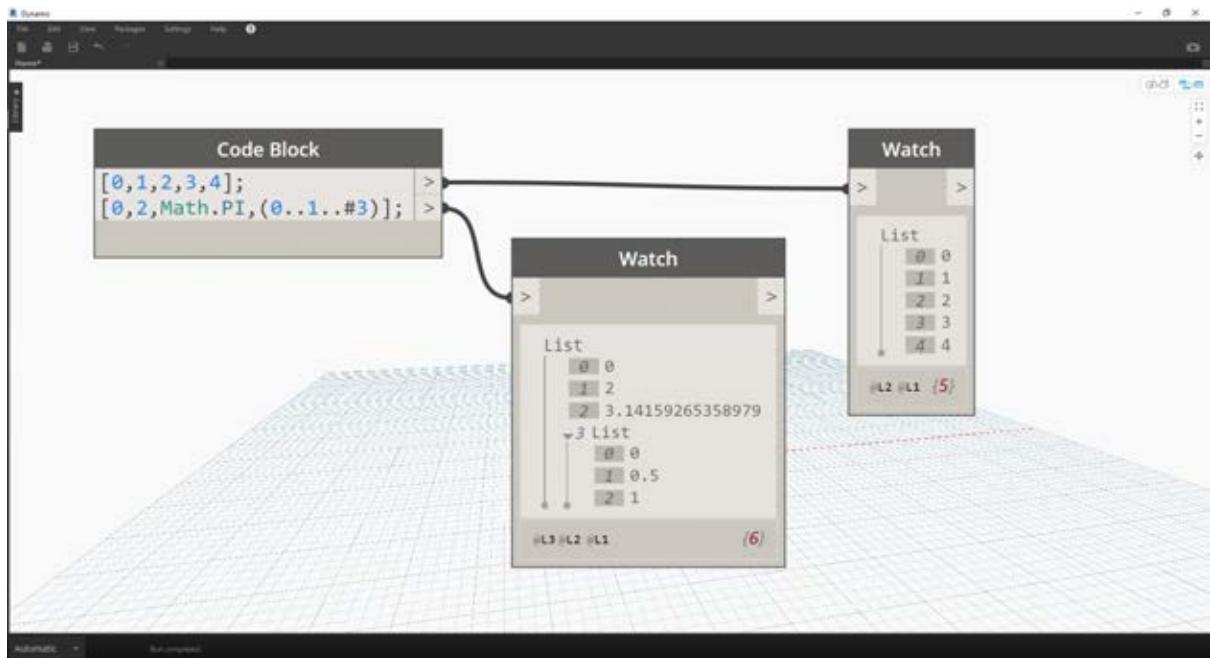
1. 曲線のいずれか 1 つの行を分離するには、*List.GetItemAtIndex* ノードを使用します。
2. *Code Block* ノードで値 2 を指定して、メインリスト内の 3 番目の要素をクエリーします。
3. *PolyCurve.ByPoints* ノードによって曲線が 1 つだけ作成されます。これは、このノードに接続されているリストが 1 つしかないためです。



1. **List.Transpose** ノードは、すべての項目をメインリスト内のすべてのリストと切り替えます。この動作は少し複雑に感じるかもしれませんねが、データ構造内の列と行を入れ替える Microsoft Excel と同じ仕組みです。
2. 転置により、リストの構造が変更されていることを確認します。転置前は 3 つの項目を持つ 5 つのリストだったのに対して、転置後は 5 つの項目を持つ 3 つのリストに変わっています。
3. **PolyCurve.ByPoints** ノードにより、元の曲線に対して垂直方向に 3 つのポリカーブが作成されます。

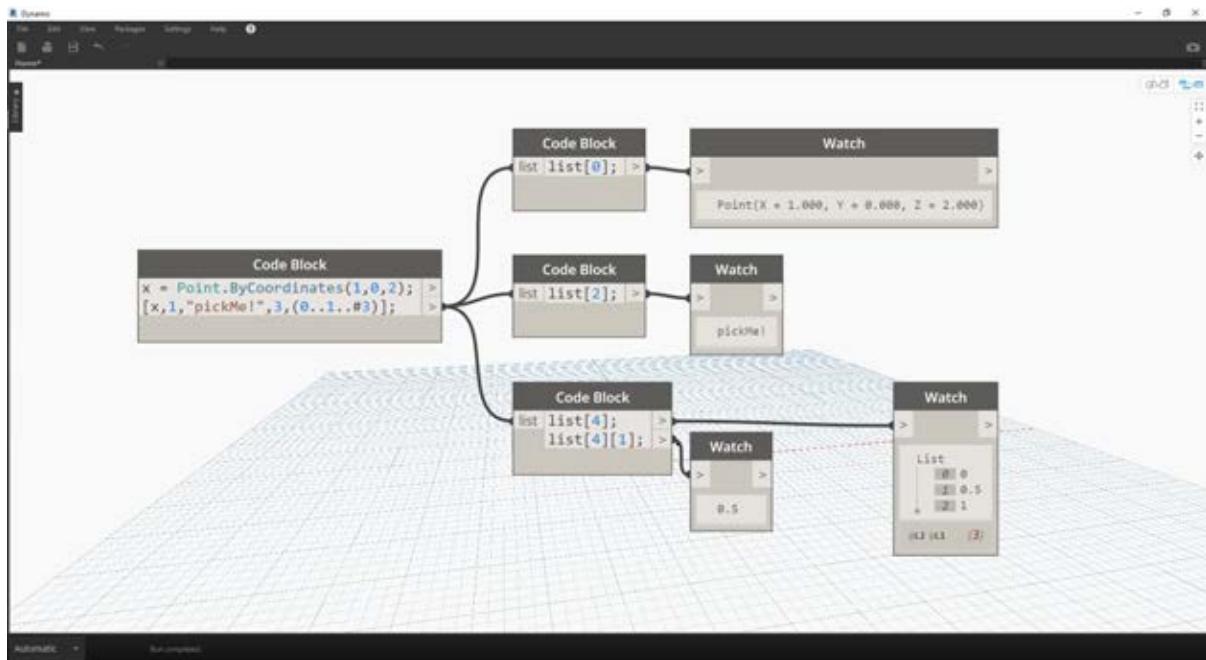
## コード ブロックの作成

コード ブロックでは、「[]」という省略表記を使用してリストを定義します。この方法を使用すると、List.Create ノードを使用するよりも素早く柔軟にリストを作成することができます。コード ブロックについては、第 7 章で詳しく説明します。次の図は、コード ブロックを使用して、複数の式を持つリストを作成する方法を示しています。



### コード ブロックのクエリー

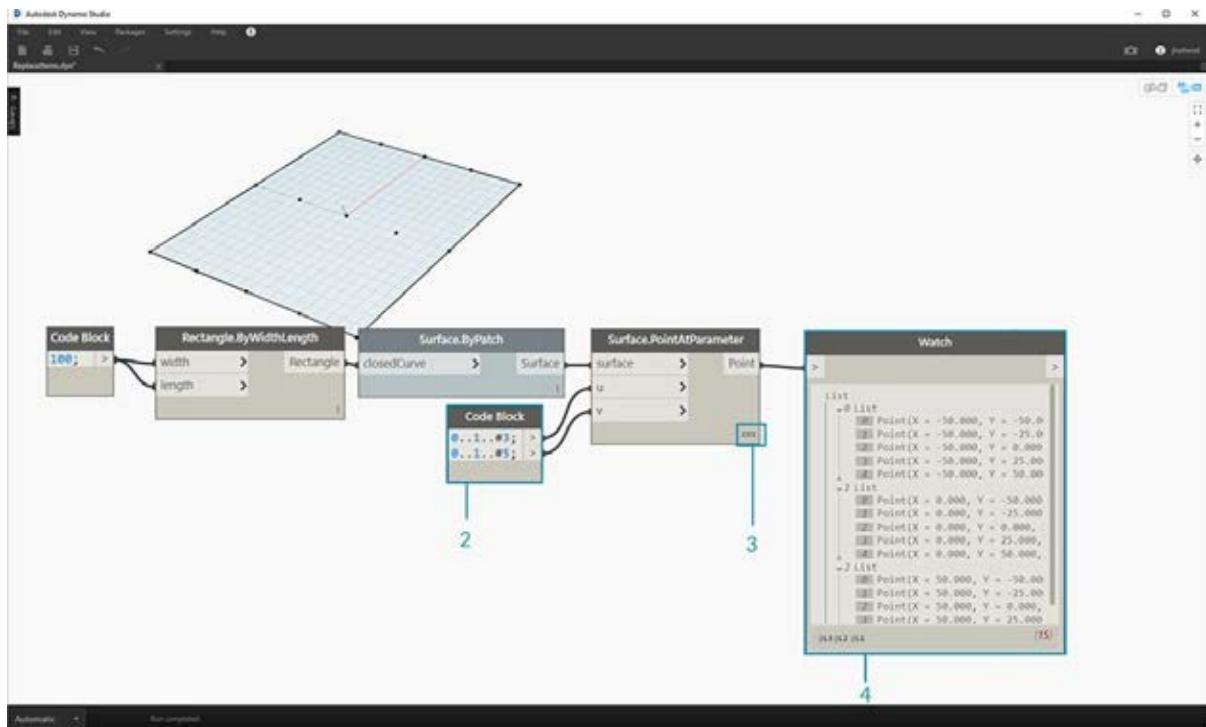
コード ブロックで「[]」という省略表記を使用すると、複雑なデータ構造で特定の項目をすばやく簡単に選択することができます。コード ブロックについては、第 7 章で詳しく説明します。次の図は、コード ブロックを使用して、複数のデータ タイプが存在するリストのクエリーを実行する方法を示しています。



## 演習 - データのクエリーと挿入

この演習に付属しているサンプル ファイルをダウンロードしてください(右クリックして[名前を付けてリンク先を保存...]を選択):  
[ReplaceItems.dyn](#)。すべてのサンプル ファイルの一覧については、付録を参照してください。

この演習では、前の演習で作成したロジックを使用してサーフェスを編集します。この演習の目的は直感的な操作を行うことです  
が、データ構造のナビゲーションは少し複雑な操作です。ここでは、制御点を移動することにより、サーフェスの編集を行います。

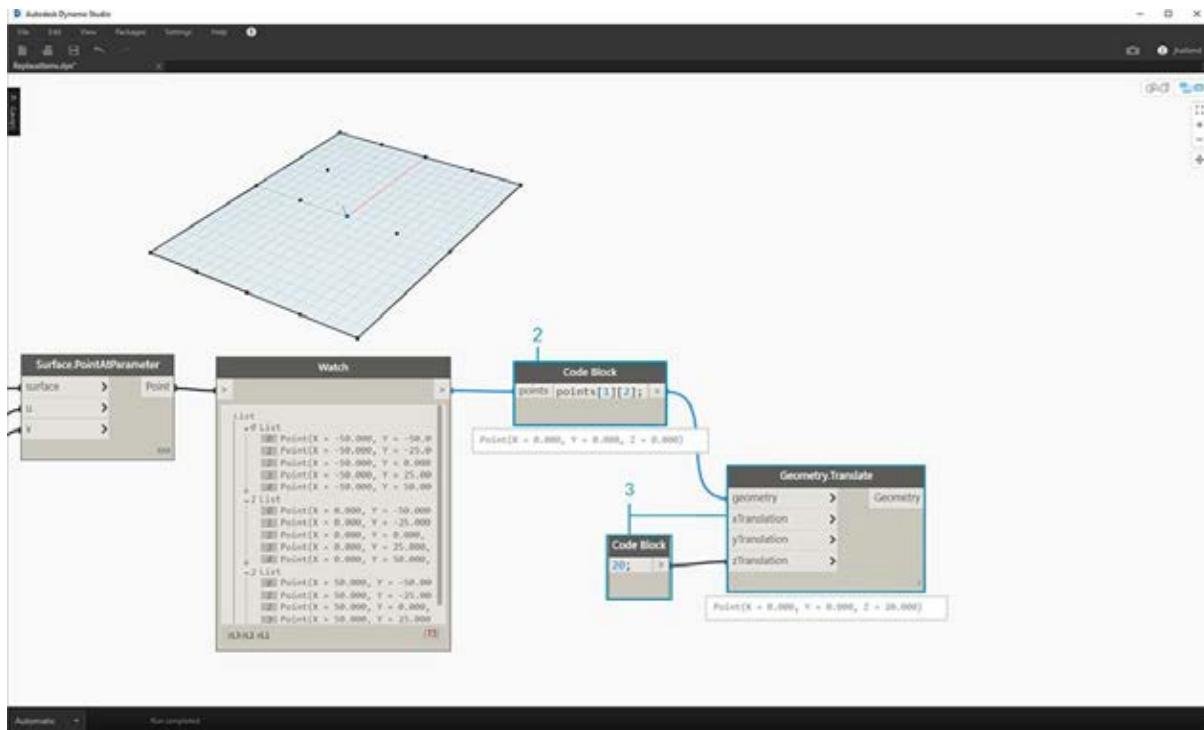


- 最初に、上図の各ノードの文字列を操作します。既定の Dynamo グリッド全体にわたる基本的なサーフェスを作成します。
- Code Block* ノードを使用して次の 2 つのコード行を入力し、*Surface.PointAtParameter* ノードの *u* 入力と *v* 入力にそれぞれのコード行を接続します。

-50..50..#3;

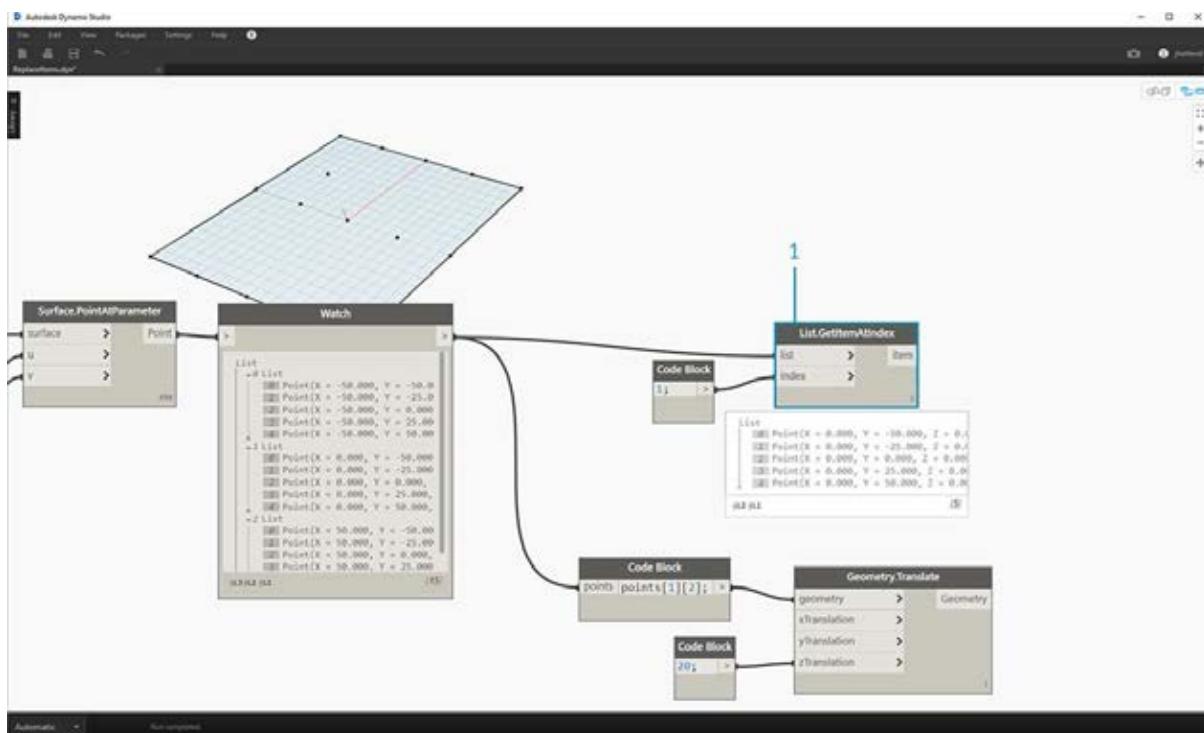
-50..50..#5;

3. *Surface.PointAtParameter* ノードのレーシングを「外積」に設定します。
4. *Watch* ノードに、5 つの項目を持つ 3 つのリストが含まれた 1 つのリストが表示されます。

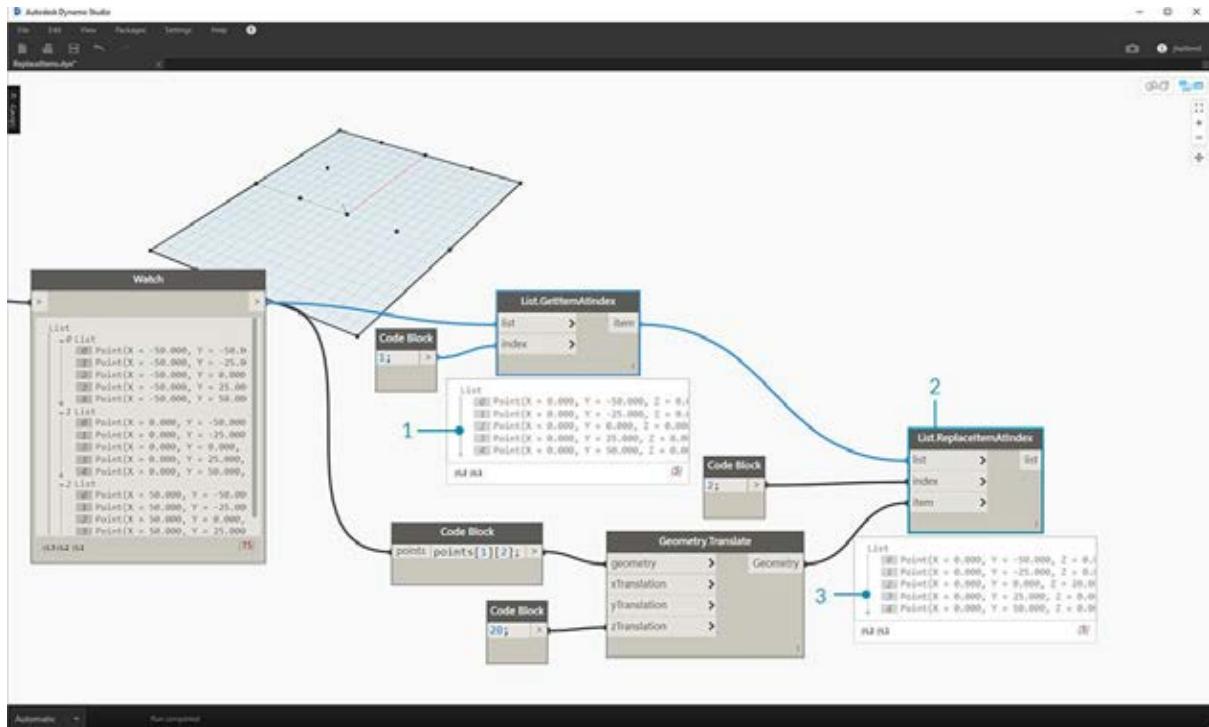


この演習では、作成したグリッドの中心点のクエリーを実行します。これを行うには、中央のリストの中心点を選択します。

1. 正しい点を選択するには、*Watch* ノードの各項目をクリックして、正しい点が選択されているかどうかを確認します。
2. *Code Block* ノードを使用して、リストのリストをクエリーするための基礎となるコード行を次の形式で指定します。  
`points[1][2];`
3. *Geometry.Translate* ノードを使用して、選択した点を Z の正の向きに 20 単位移動します。

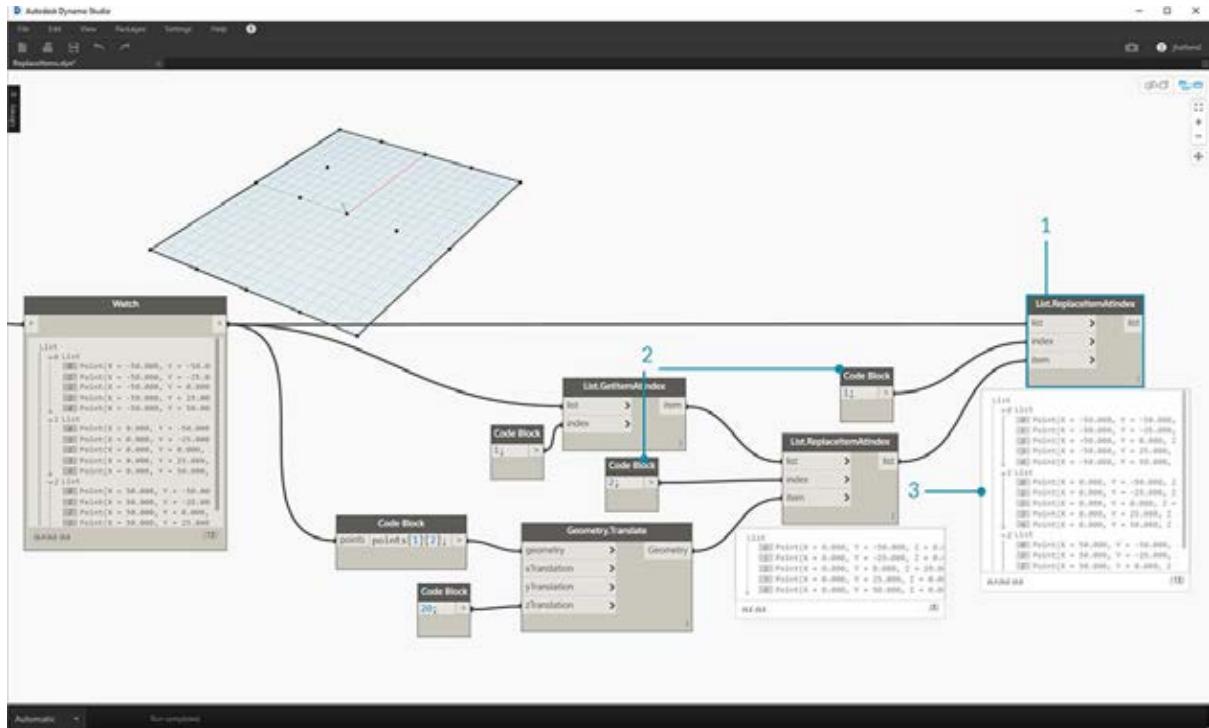


1. *List.GetItemAtIndex* ノードを使用して、点の中央の行を選択します。前の手順と同様に、*Code Block* ノードで *points[1]*; というコード行を指定してクエリーを実行することもできます。



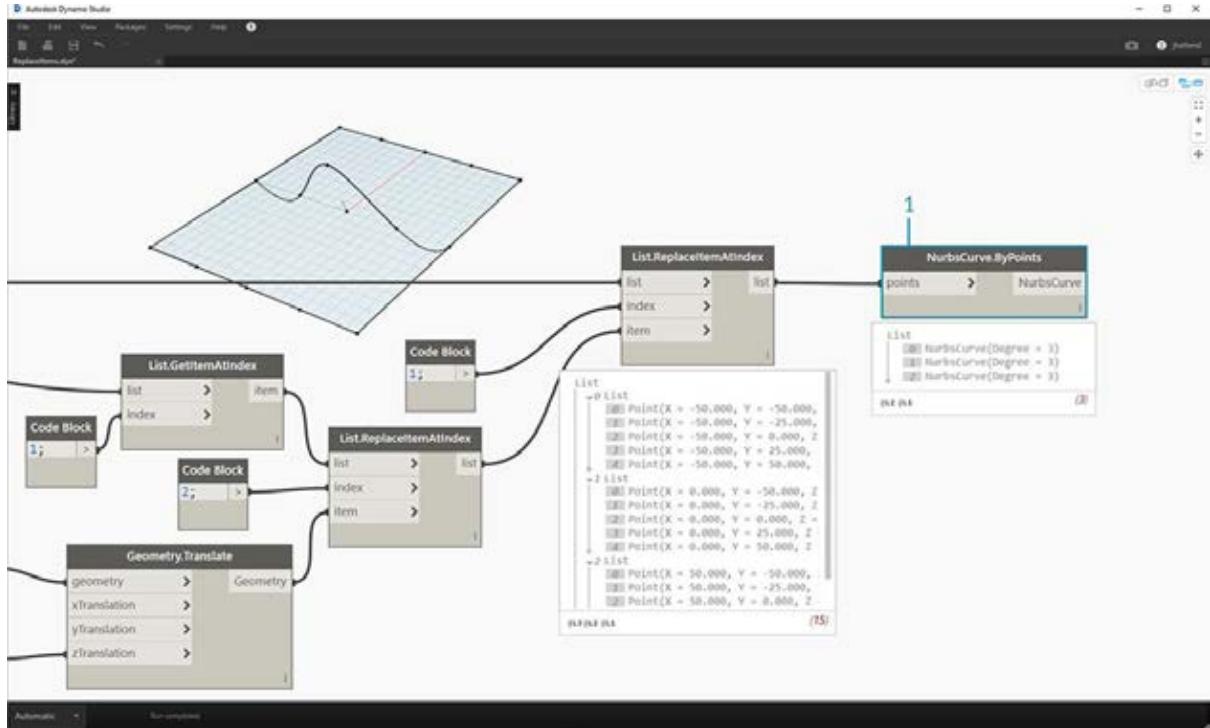
ここまで手順で、中心点のクエリーを実行し、中心点を上方向に移動しました。ここでは、この中心点を元のデータ構造に戻します。

1. 最初に、前の手順で分離したリストの項目を置き換えます。
2. *List.ReplaceItemAtIndex* ノードで「2」というインデックス値を使用し、置き換える項目を対象となる点 (*Geometry.Translate* ノードの *Geometry* 出力)に接続して、中心点を置き換えます。
3. 対象の点がリストの中央の項目に入力されたことが出力として表示されます。



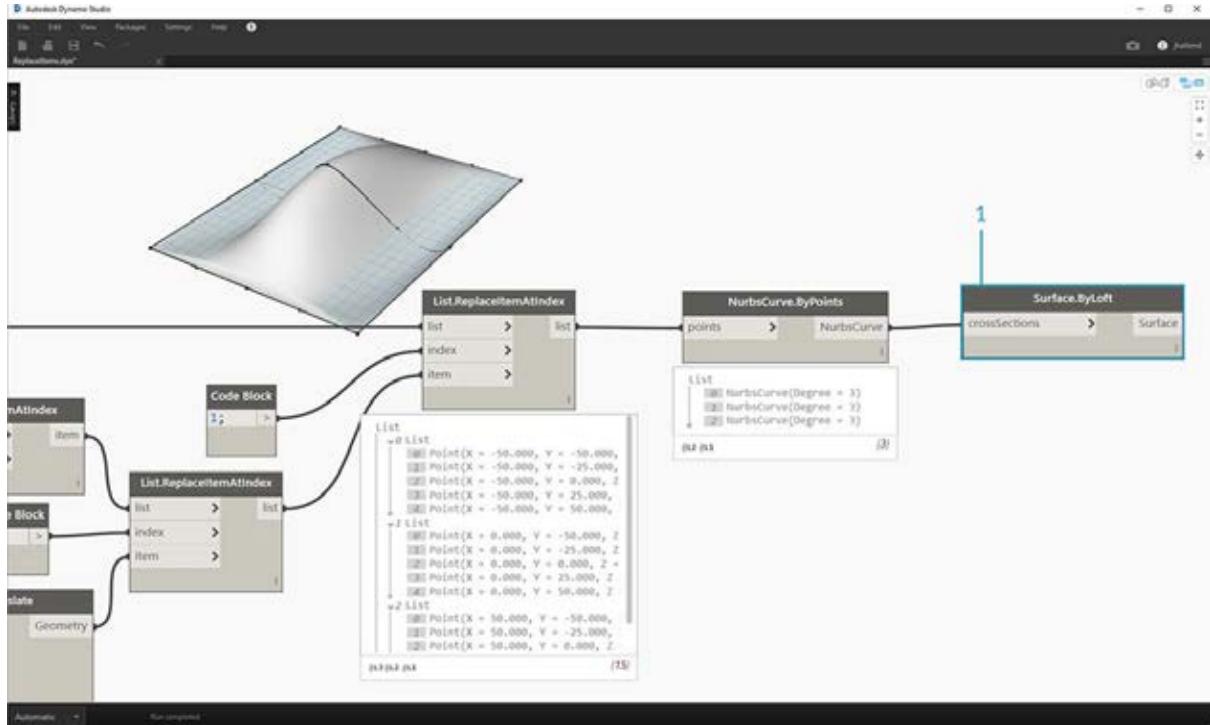
ここでは、前の演習で変更したリストを、元のデータ構造であるリストのリストに戻します。

- 前の演習と同様に、*List.ReplaceItemAtIndex* ノードを使用して、中央のリストを変更したリストで置き換えます。
- これら 2 つのノードのインデックスを定義する *Code Block* ノードの値は、それぞれ 1 と 2 であることに注意してください。これらの値は、*Code Block* ノードで指定した元のクエリー(*points[1]/[2]*)に対応しています。
- インデックス 1 の位置でリストを選択すると、Dynamo のプレビューでデータ構造がハイライト表示されます。これで、対象の点が元のデータ構造に正しく移動されたことになります。



上図の点のセットからサーフェスを作成する方法はいくつかあります。ここでは、複数の曲線をまとめてロフトしてサーフェスを作成してみましょう。

- NurbsCurve.ByPoints* ノードを作成し、新しいデータ構造を接続して、3 つの NURBS 曲線を作成します。



- Surface.ByLoft* ノードを *NurbsCurve.ByPoints* ノードの出力に接続します。これで、サーフェスが変更されました。ジオメトリの元の Z 値を変更することができます。この値を変更して、ジオメトリがどのように変化するかを確認してください



# N 次元のリスト

## N 次元のリスト

ここでは、データ階層にさらに層を追加して、より高度なリストの操作方法について見ていきます。データ構造を拡張して、2次元を超える多次元のリストのリストを作成することができます。Dynamo のリストは項目として処理されるため、必要な数の次元を持つデータを作成することができます。

多次元のリストの構造は、ロシアのマトリョーシカ人形に似ています。各リストは、複数の項目を格納するコンテナとして考えることができます。各リストには独自のプロパティが存在し、それぞれのリストは独自のオブジェクトとして見なされます。



ロシアのマトリョーシカ人形(写真: [Zeta](#))は、N 次元のリストの構造に似ています。それぞれの層が 1 つのリストを表し、各リスト内に項目が格納されています。Dynamo では、1 つのリストに複数のリストを格納することができます。この場合、格納されているそれぞれのリストが、格納元のリストの項目になります。

N 次元のリストを視覚的に説明するのは難しいため、この章ではいくつかの演習を使用して、2 次元を超える多次元のリストを操作する方法について見ていきます。

## マッピングと組み合わせ

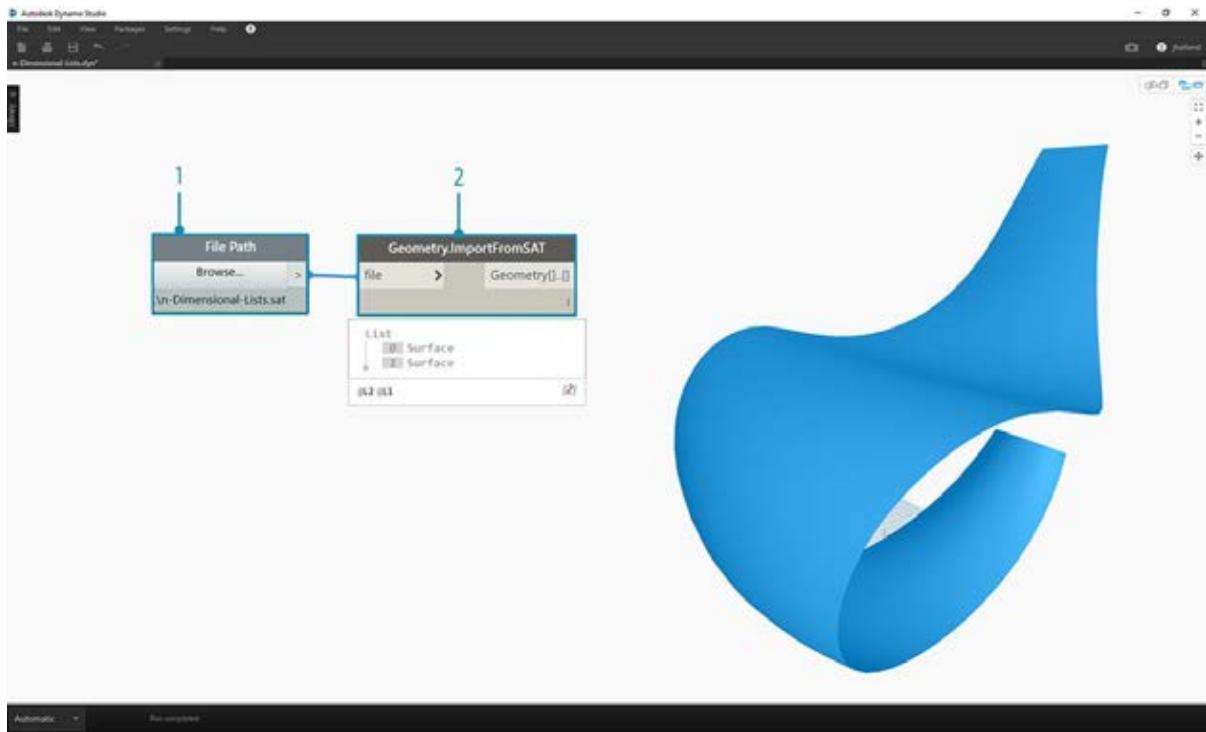
Dynamo のデータ管理において最も複雑な概念はマッピングですが、複雑なリスト階層を操作する場合は特に、マッピングが重要になります。この章の演習では、多次元のデータを操作する際に、どのようなケースでマッピングと組み合わせ機能を使用するかについて説明します。

List.Map ノードと List.Combine ノードの概要については、前の章で説明しました。この章の最後の演習でこれらのノードを使用して、複雑なデータ構造の操作を行います。

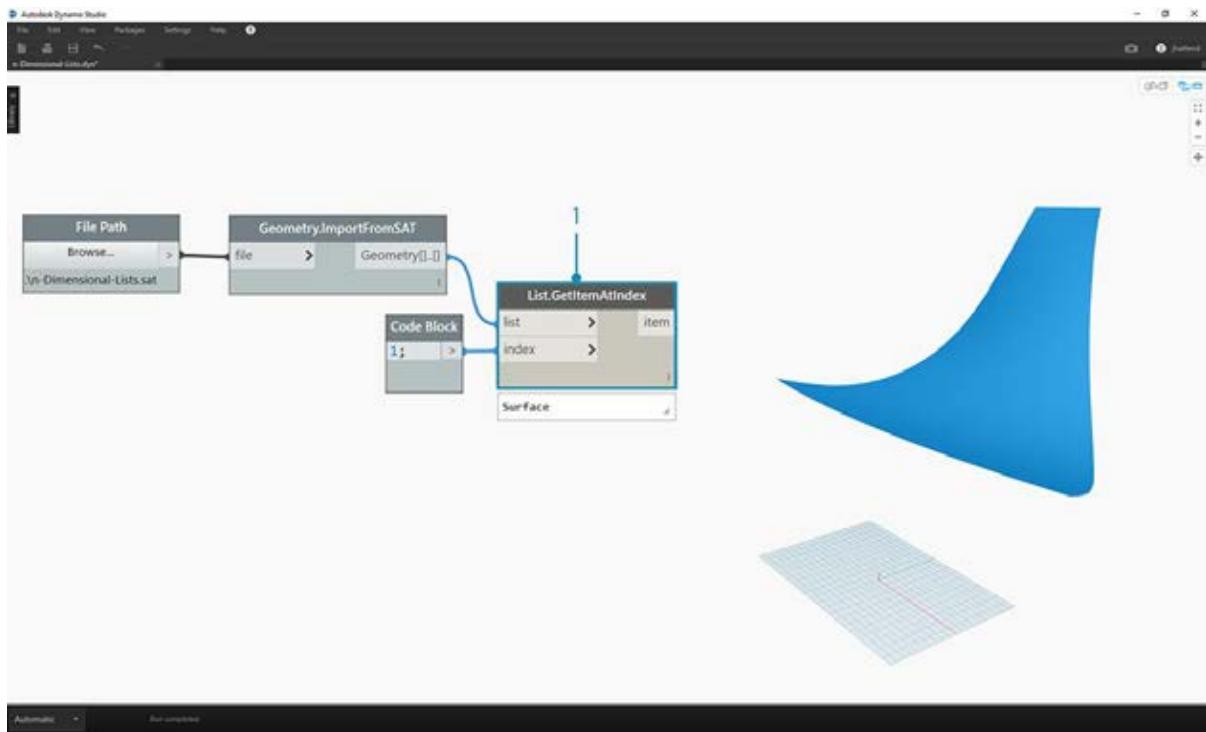
### 演習 - 2 次元のリスト - 基本的な操作

この演習に付属しているサンプルファイルをダウンロードしてください(右クリックして[名前を付けてリンク先を保存]を選択)。すべてのサンプルファイルの一覧については、付録を参照してください。1.[n-Dimensional-Lists.dyn](#) 2.[n-Dimensional-Lists.sat](#)

この章では、読み込んだジオメトリを操作するための演習を 3 つ見ていきます。この演習はその最初の演習です。演習を進めていくにつれて、より複雑なデータ構造を扱います。

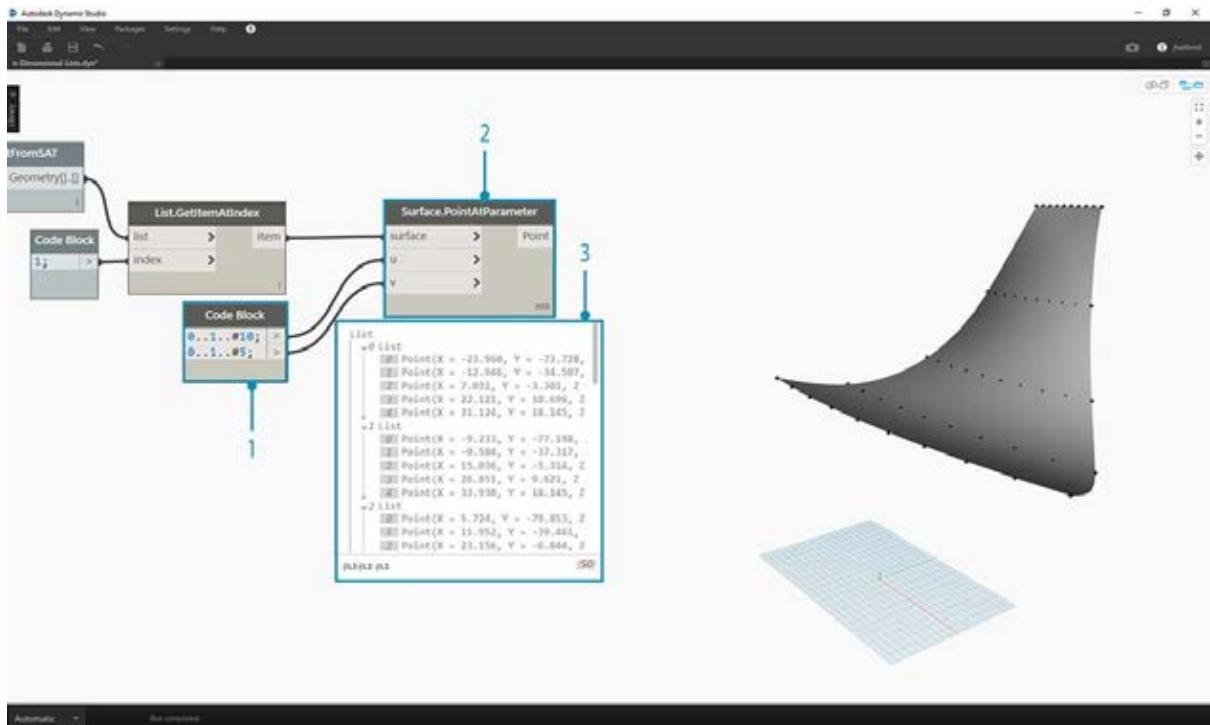


- 最初に、演習ファイル フォルダ内の .sat ファイルを使用します。このファイルを取得するには、*File Path* ノードを使用します。
- Geometry.ImportFromSAT* ノードを使用すると、ジオメトリが 2 つのサーフェスとして Dynamo のプレビューに読み込まれます。



説明を簡単にするため、この演習では 1 つのサーフェスだけを使用します。

- インデックス値として 1 を選択して、上部のサーフェスをグラブします。これを行うには、*List.GetItemAtIndex* ノードを使用します。

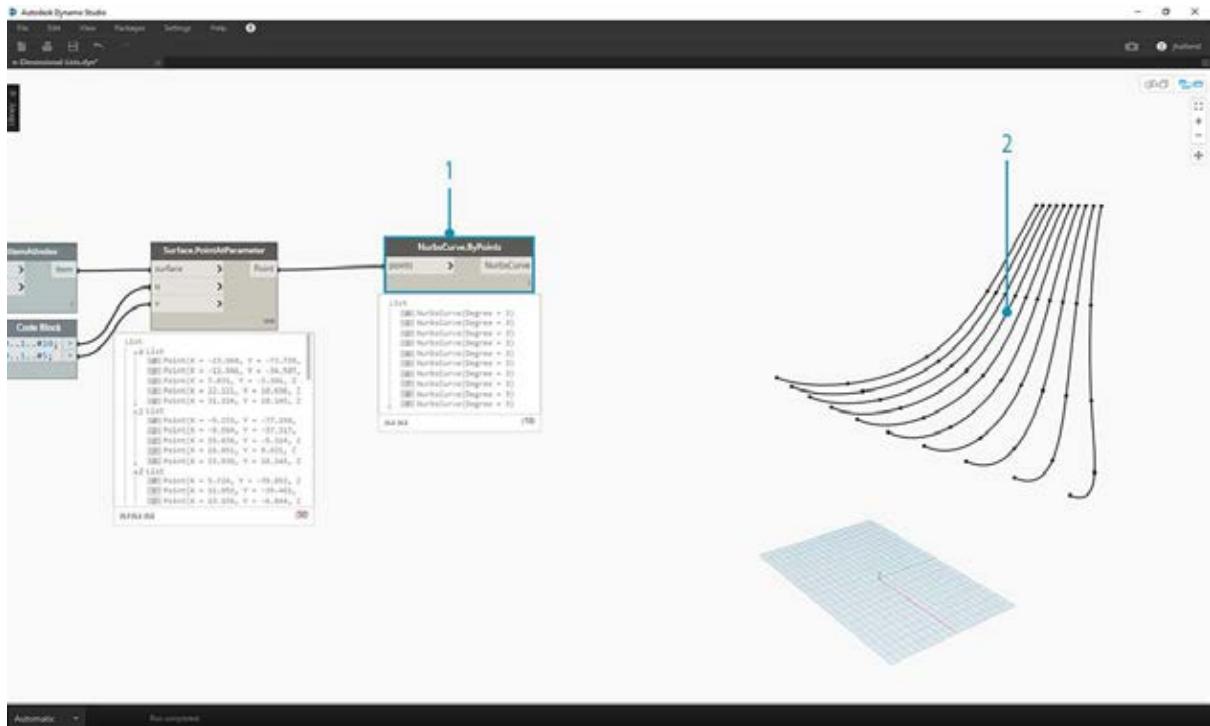


次に、グラフしたサーフェスを点のグリッドに分割します。

1. *Code Block* ノードを使用して、次の 2 つのコード行を入力します。

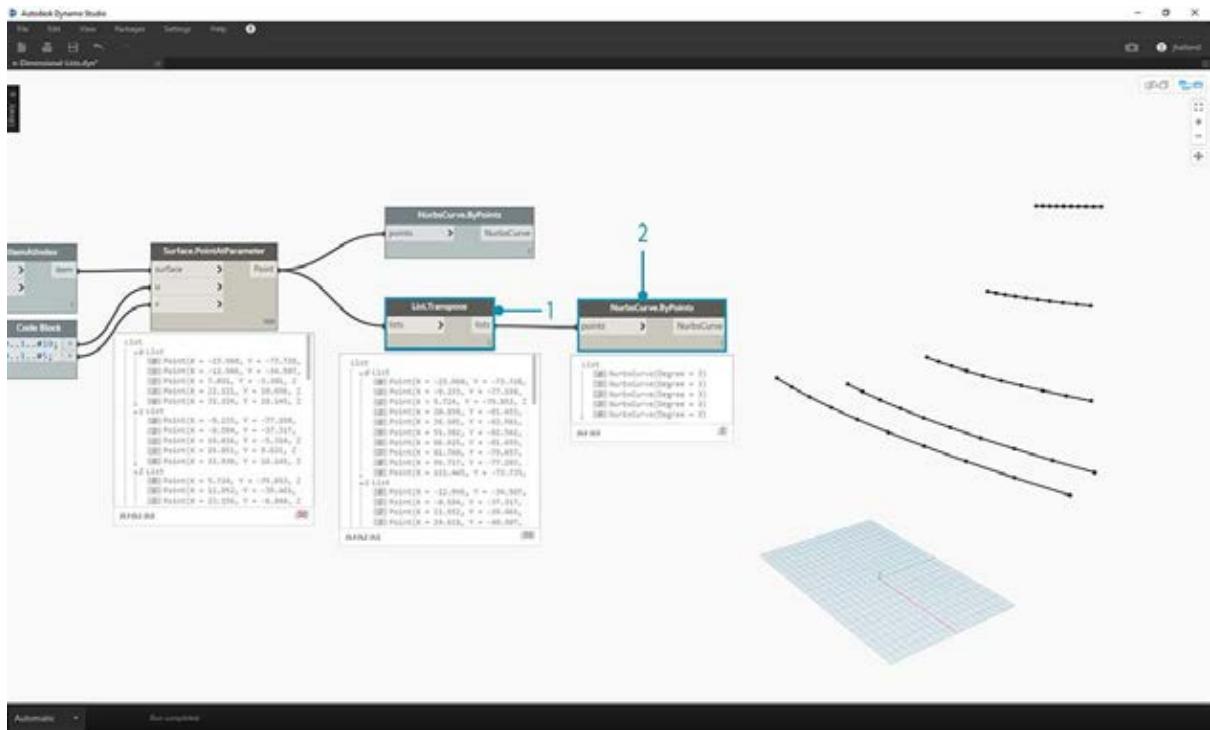
```
0..1..#10;
0..1..#5;
```

1. 上記のコード行を、*Surface.PointAtParameter* ノードの *u* 入力と *v* 入力に接続します。次に、このノードのレーシングを「外積」に変更します。
2. 出力としてデータ構造が表示されます。このデータ構造は、Dynamo のプレビューで表示することもできます。



1. *NurbsCurve.ByPoints* ノードを *Surface.PointAtParameter* ノードの出力に接続して、データ構造の内容を確認します。

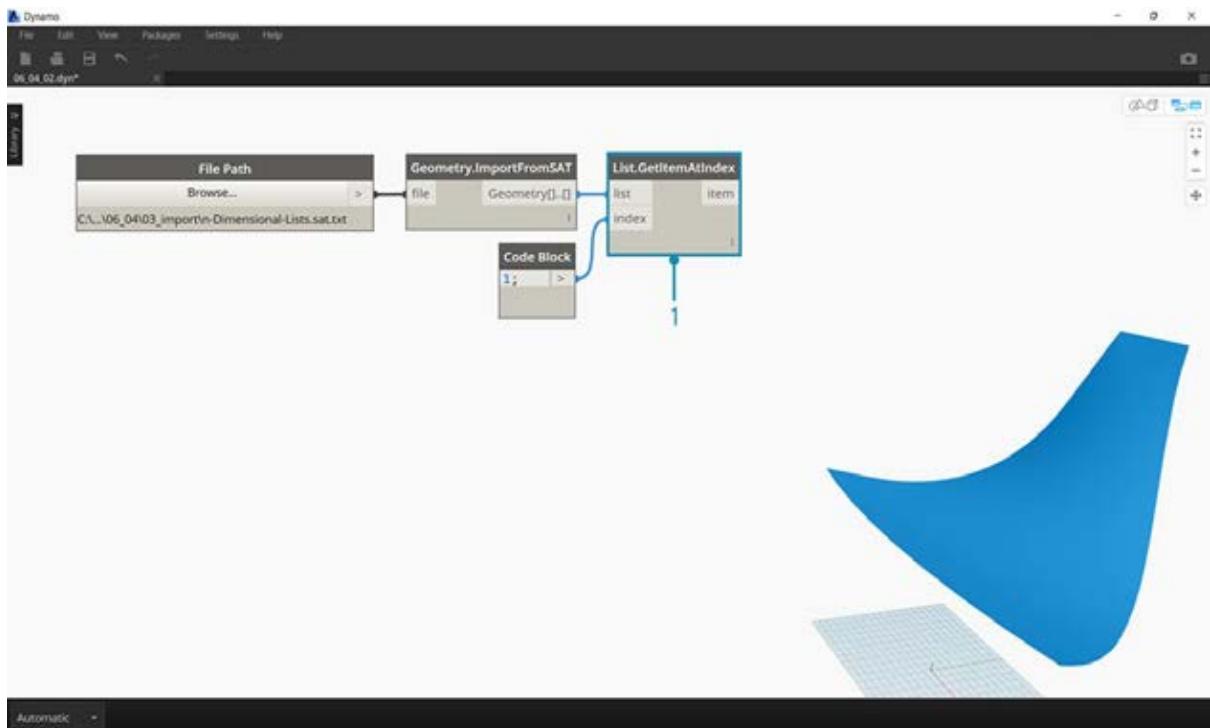
2. サーフェスに沿って垂直方向に 10 本の曲線が表示されます。



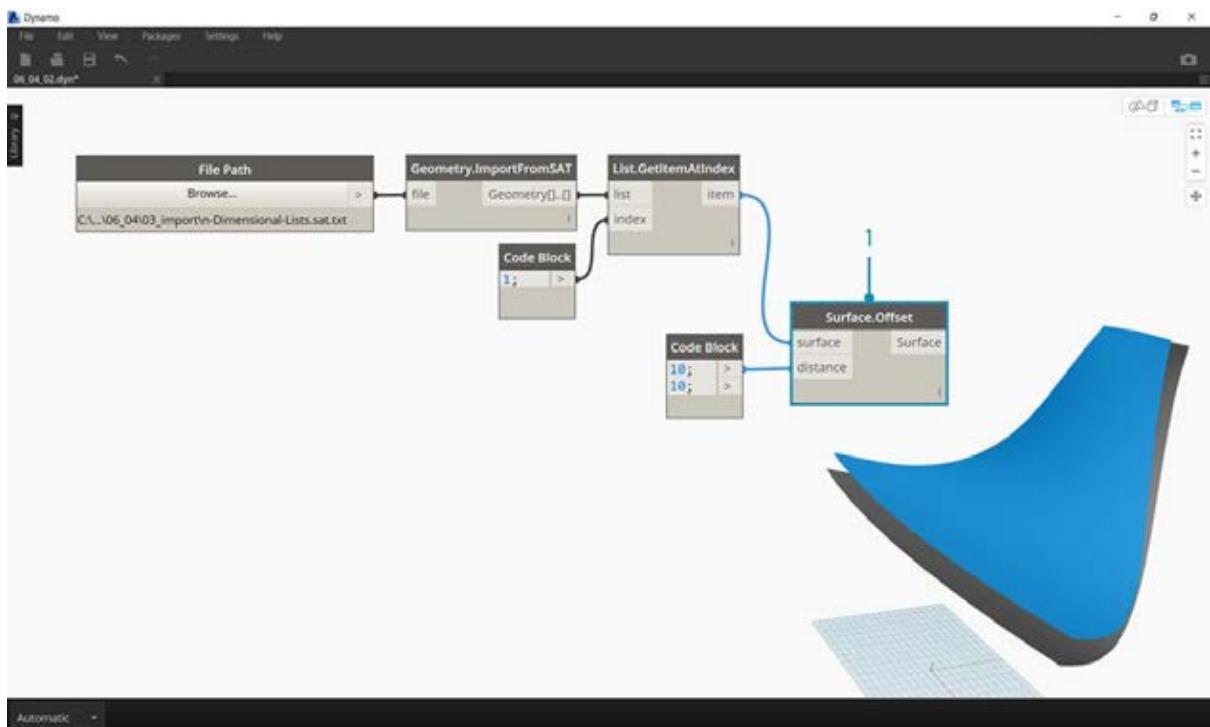
1. 基本的な *List.Transpose* ノードにより、リストのリストの列と行が反転します。
2. *List.Transpose* ノードの出力を *NurbsCurve.ByPoints* ノードに接続すると、サーフェスに沿って水平方向に 5 本の曲線が作成されます。

## 演習 - 2 次元のリスト - 高度な操作

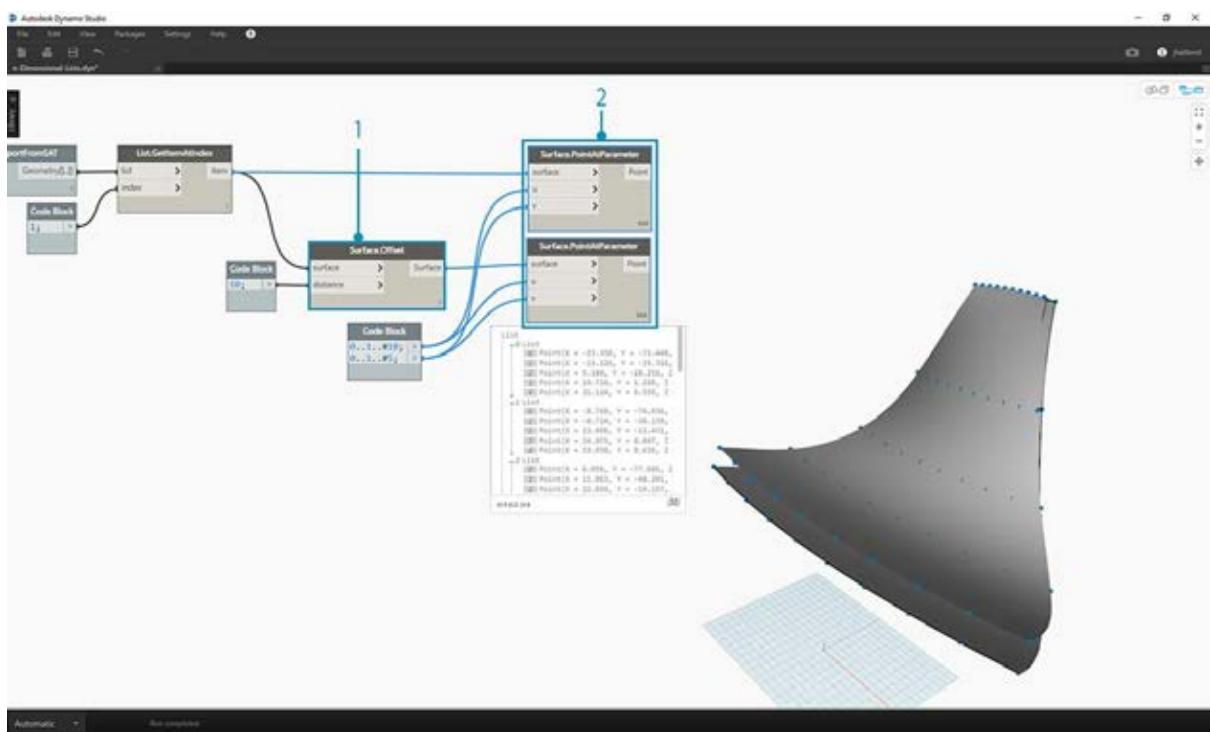
この演習では、少し複雑な操作を実行してみましょう。ここでは、前の演習で作成した曲線に対して操作を実行します。具体的には、これらの曲線を別のサーフェスに関連付けて、2 つのサーフェス間で曲線をロフトします。この操作を実行する場合、データ構造の処理が少し複雑になりますが、基本的な考え方はこれまでと同じです。



1. 最初に、*List.GetItemAtIndex* ノードを使用して、前の演習で読み込んだジオメトリの上部サーフェスを分離します。



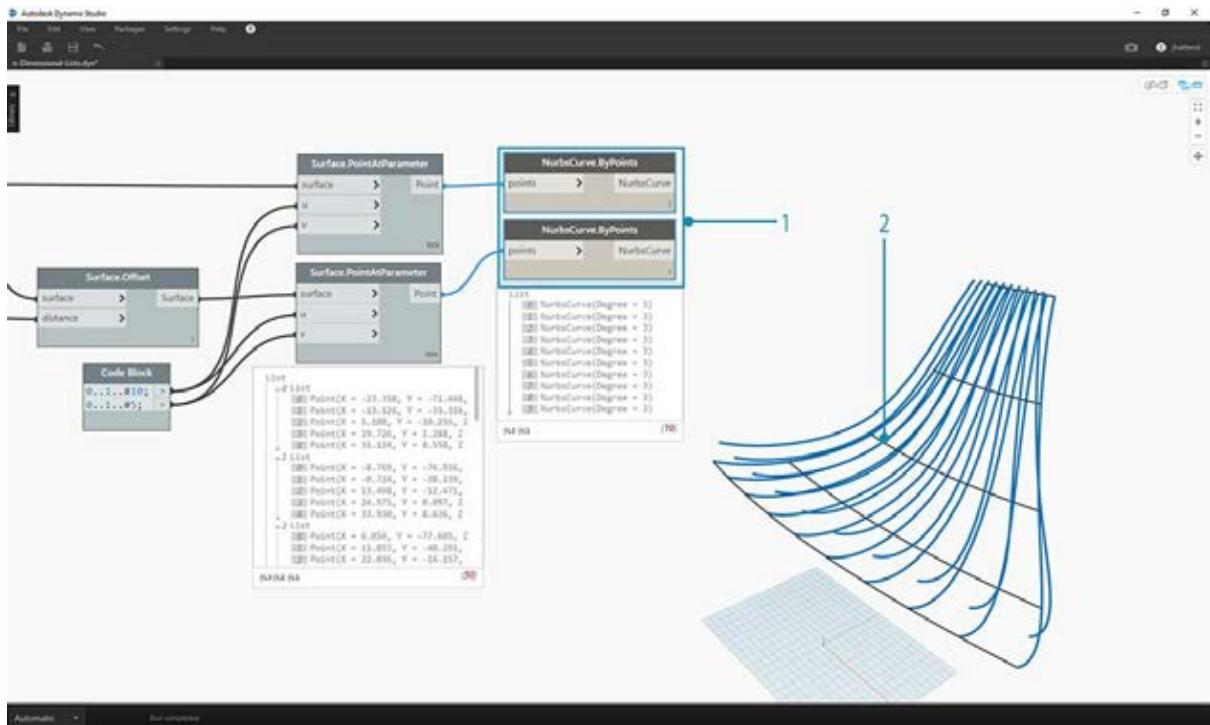
1. *Surface.Offset* ノードで 10 という値を指定して、サーフェスをオフセットします。



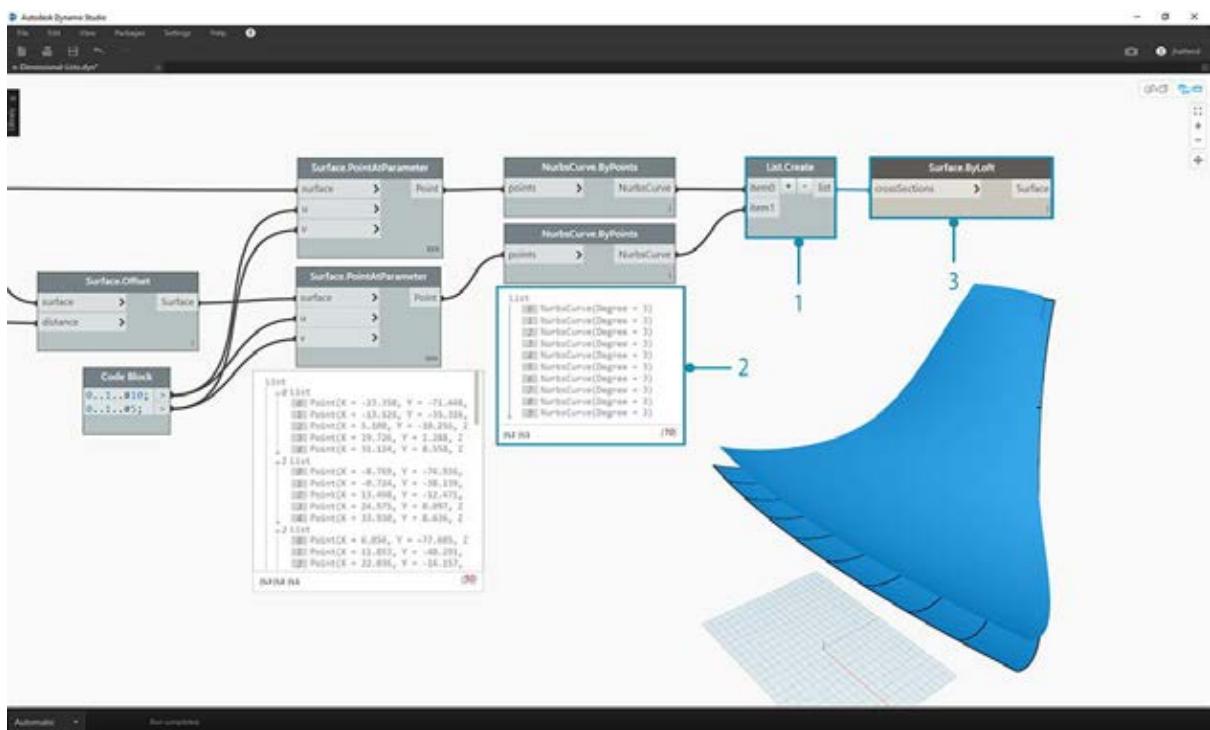
1. 前の演習と同様に、*Code Block* ノードで次の 2 つのコード行を入力します。

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

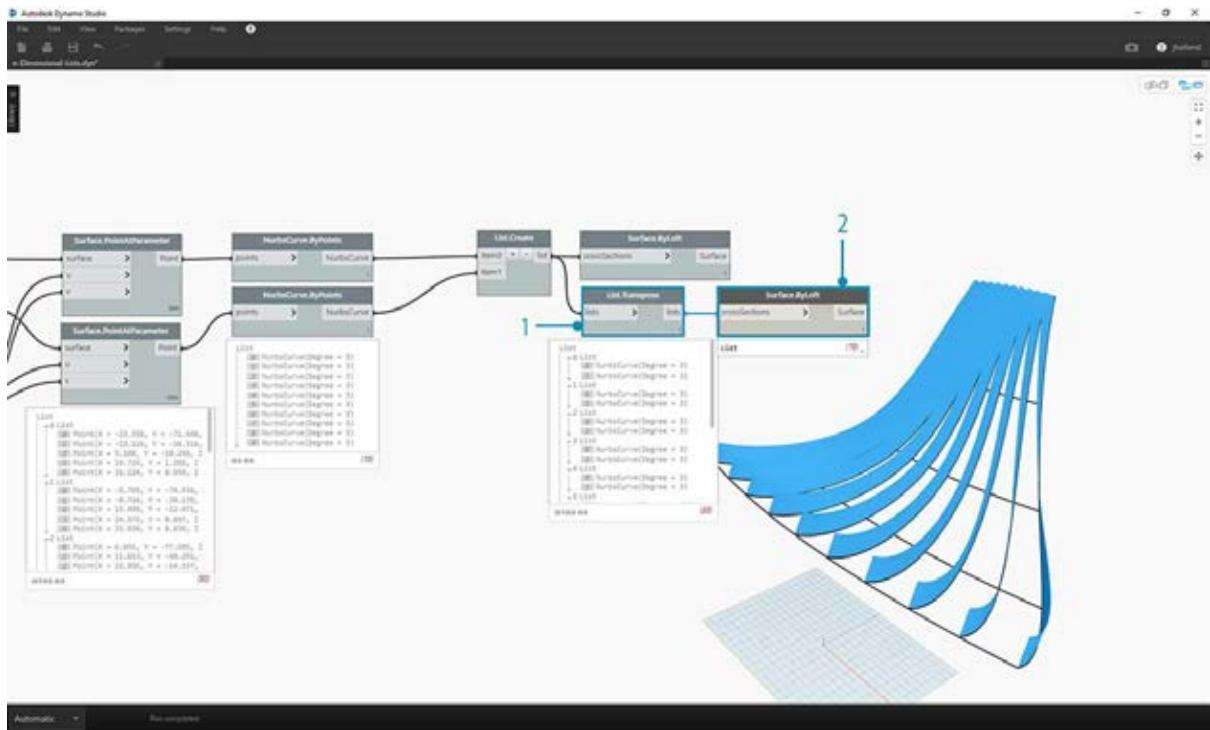
1. 上記のコード行を 2 つの *Surface.PointAtParameter* ノードに接続し、各ノードのレーシングを「外積」に設定します。いずれか一方のノードが元のサーフェスに接続され、もう一方のノードがオフセットされたサーフェスに接続されます。

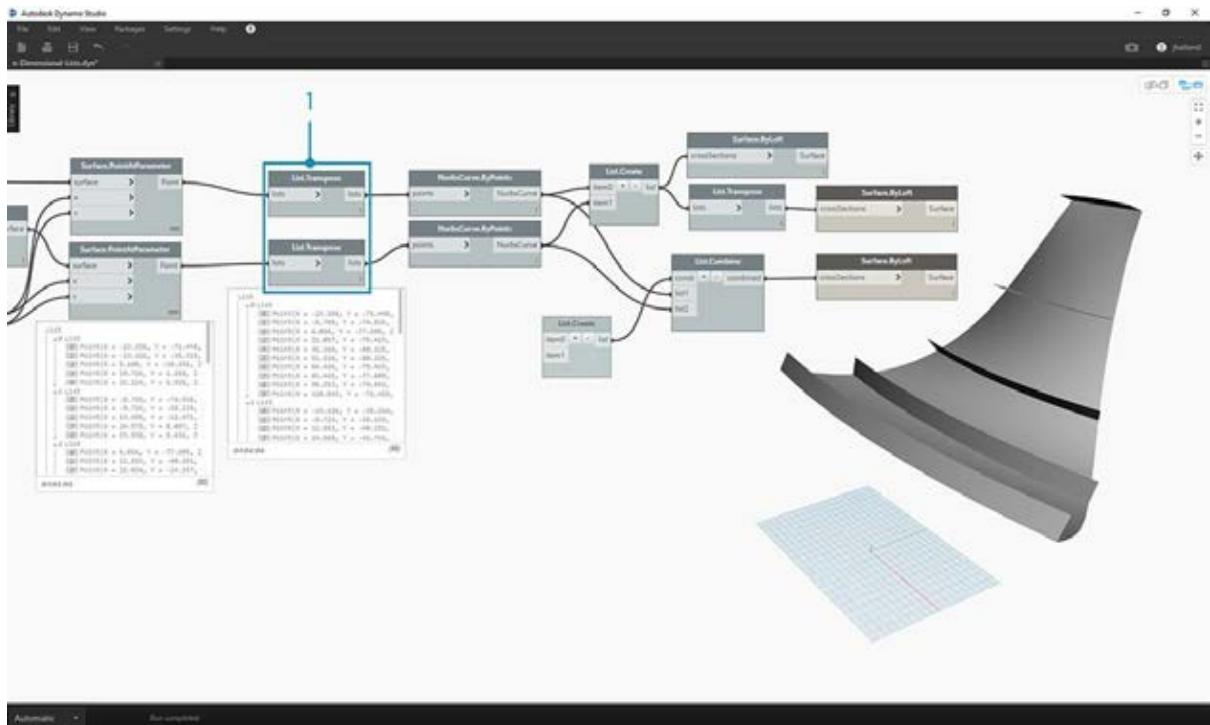


1. 前の演習と同様に、**Surface.PointAtParameter** ノードの出力を 2 つの **NurbsCurve.ByPoints** ノードに接続します。
2. Dynamo のプレビューに、2 つのサーフェスに対応する 2 組の曲線が表示されます。



1. **List.Create** ノードを使用して、2 組の曲線を 1 つのリストのリストに結合します。
2. 10 個の項目を持つ 2 つのリストが output として表示されます。各リストが、NURBS 曲線の各接続セットを表しています。
3. **Surface.ByLoft** ノードを実行すると、このデータ構造を視覚的に理解することができます。このノードは、各サブリスト内のすべての曲線をロフトします。

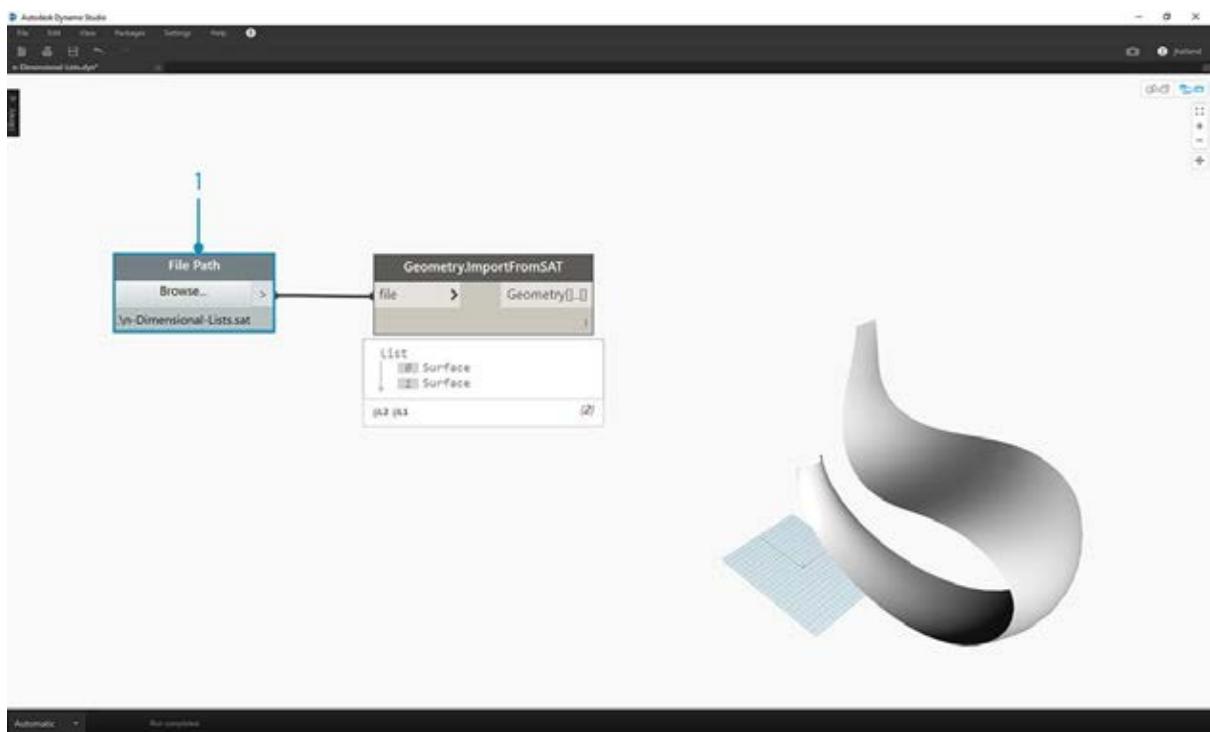




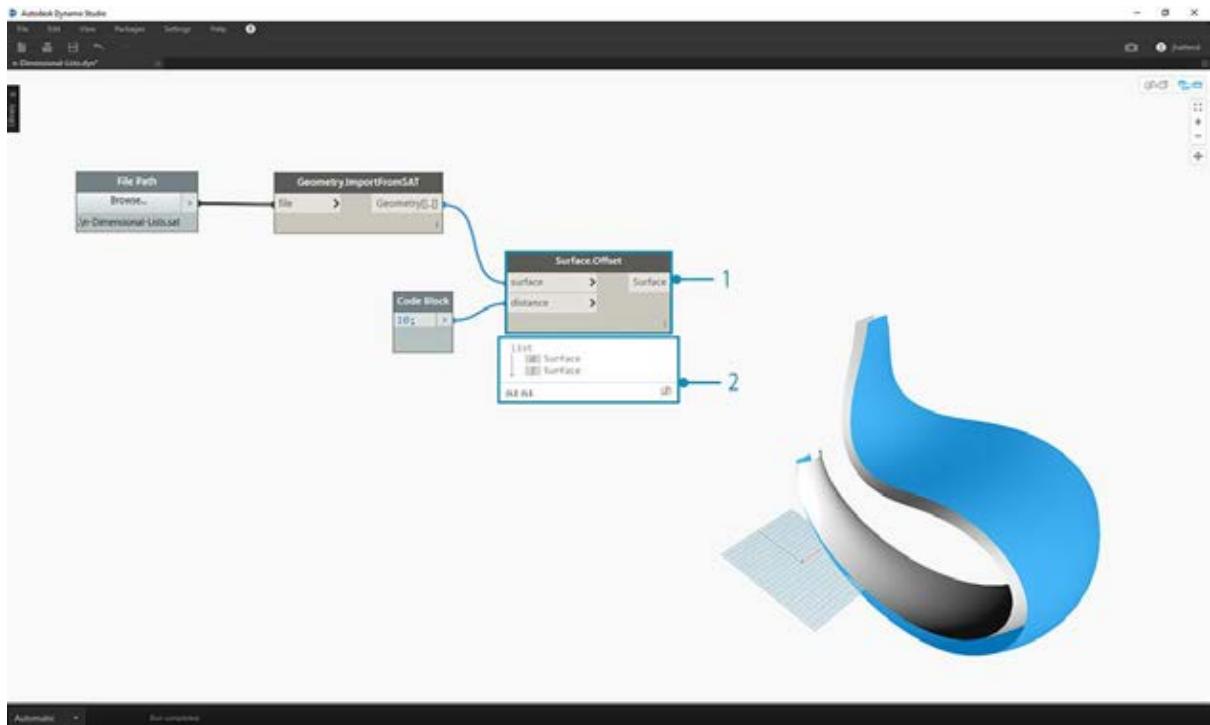
- リブ形状の構造物内の曲線の方向を切り替える場合は、手順をいくつか戻り、NurbsCurve.ByPoints ノードに接続する前に List.Transpose ノードを使用します。これにより、リスト内の列と行が反転し、水平方向のリブ形状が 5 つ作成されます。

### 演習 - 3 次元のリスト

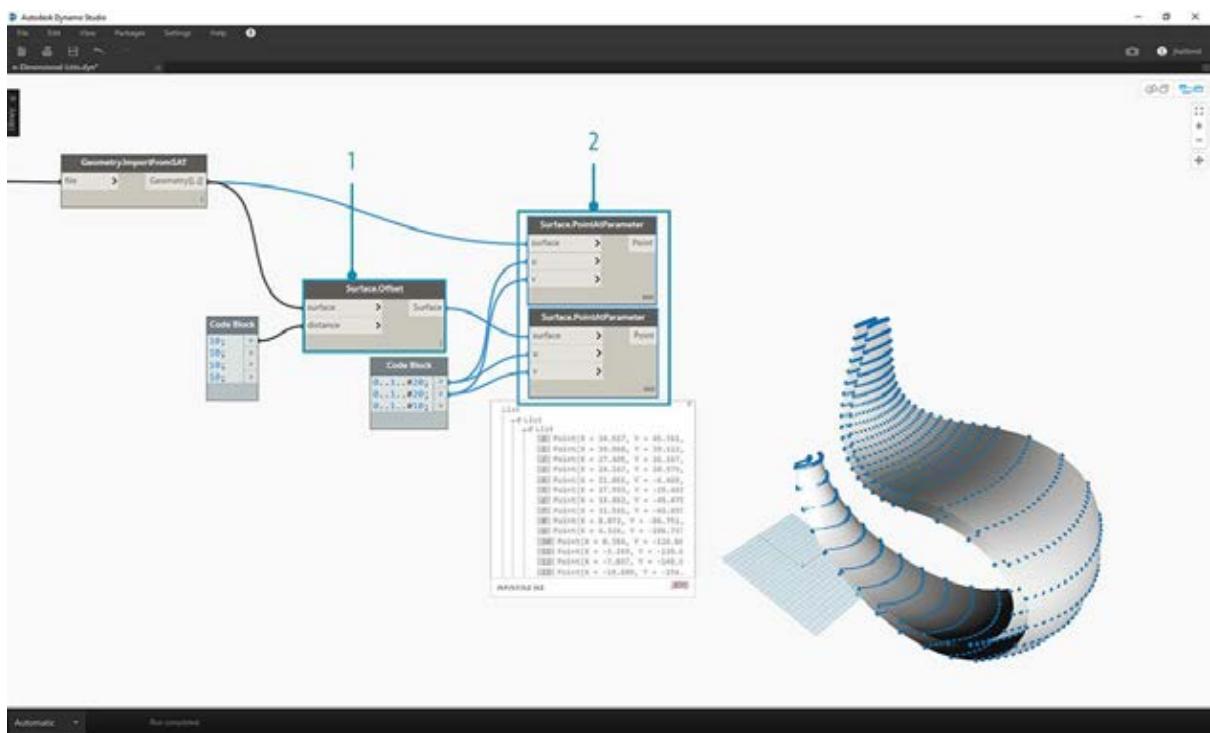
ここからは、さらに高度な操作を実行してみましょう。この演習では、読み込んだ 2 つのサーフェスを両方とも使用して、複雑なデータ階層を作成します。ただし、基本的な考え方はこれまでと同じで、同じ操作を実行することになります。



- 最初に、前の演習で読み込んだファイルを使用します。



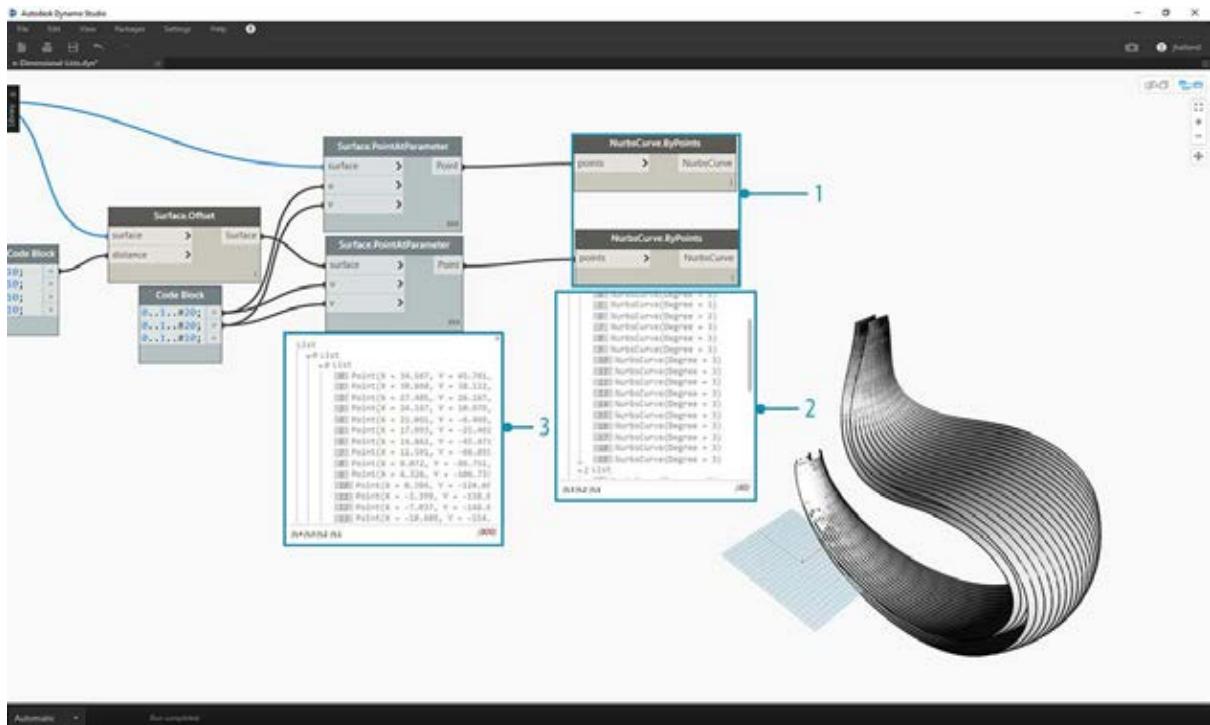
1. 前の演習と同様に *Surface.Offset* ノードを使用し、サーフェスをオフセットする値として *10* を指定します。
2. オフセットされたノードによって 2 つのサーフェスが作成されたことが出力として表示されます。



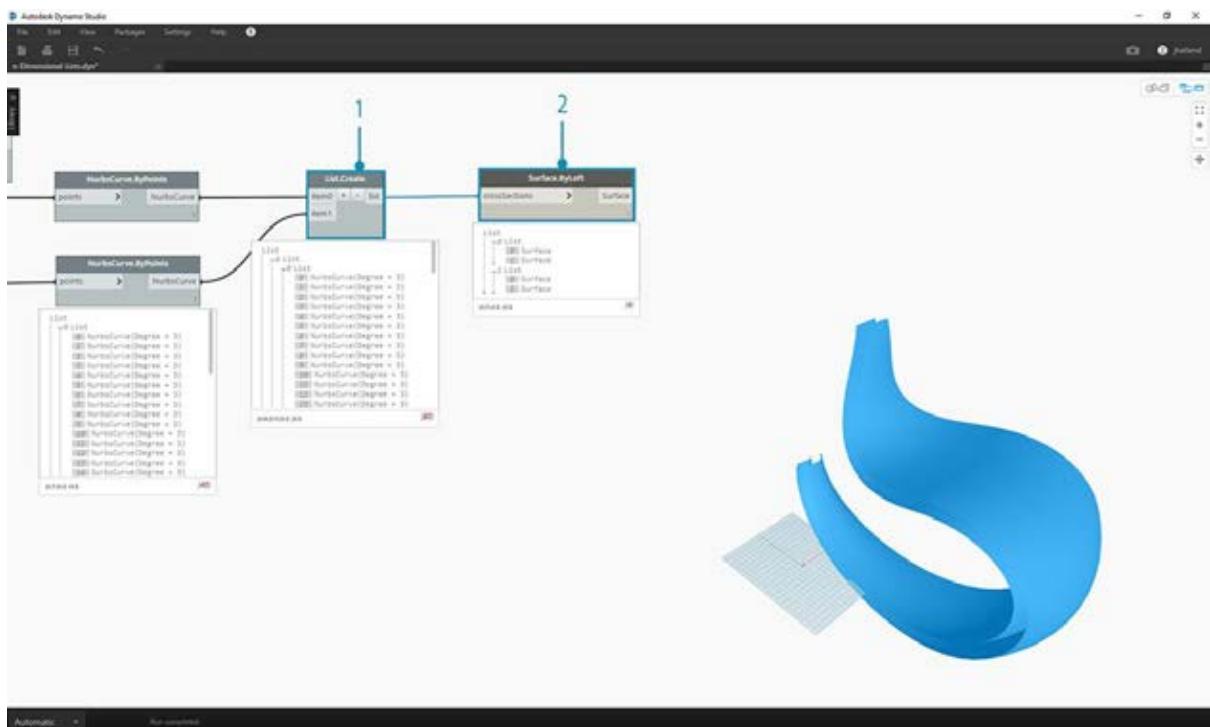
1. 前の演習と同様に、Code Block で次の 2 つのコード行を入力します。

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

1. 上記のコード行を 2 つの *Surface.PointAtParameter* ノードに接続し、各ノードのレーシングを「外積」に設定します。いずれか一方のノードが元のサーフェスに接続され、もう一方のノードがオフセットされたサーフェスに接続されます。



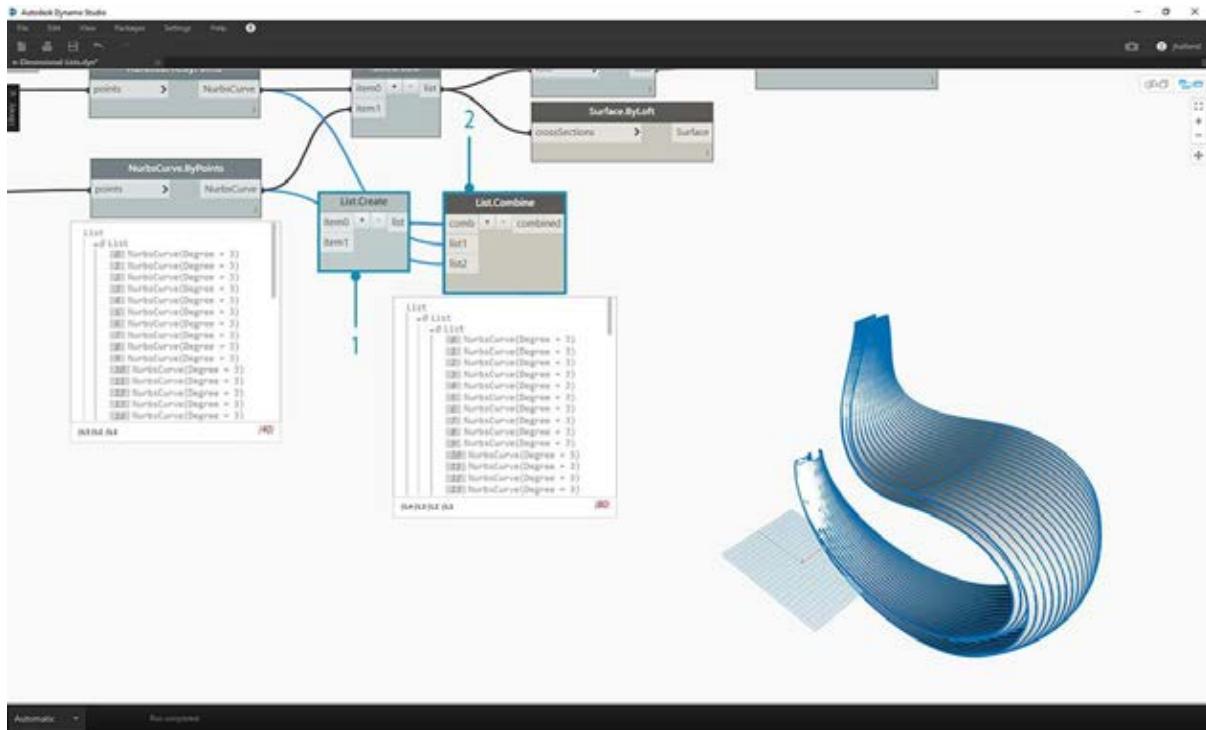
- 前の演習と同様に、Surface.PointAtParameter ノードの出力を 2 つの NurbsCurve.ByPoints ノードに接続します。
- NurbsCurve.ByPoints ノードの出力を確認すると、この出力が 2 つのリストを持つリストであることがわかります。これは、前の演習で扱ったリストよりも複雑なデータ構造です。データは基礎となるサーフェスによって分類されるため、構造化されたデータに別の層を追加します。
- Surface.PointAtParameter ノードでは、データ構造がさらに複雑になっていることがわかります。このノードでは、「リストのリストのリスト」が作成されます。



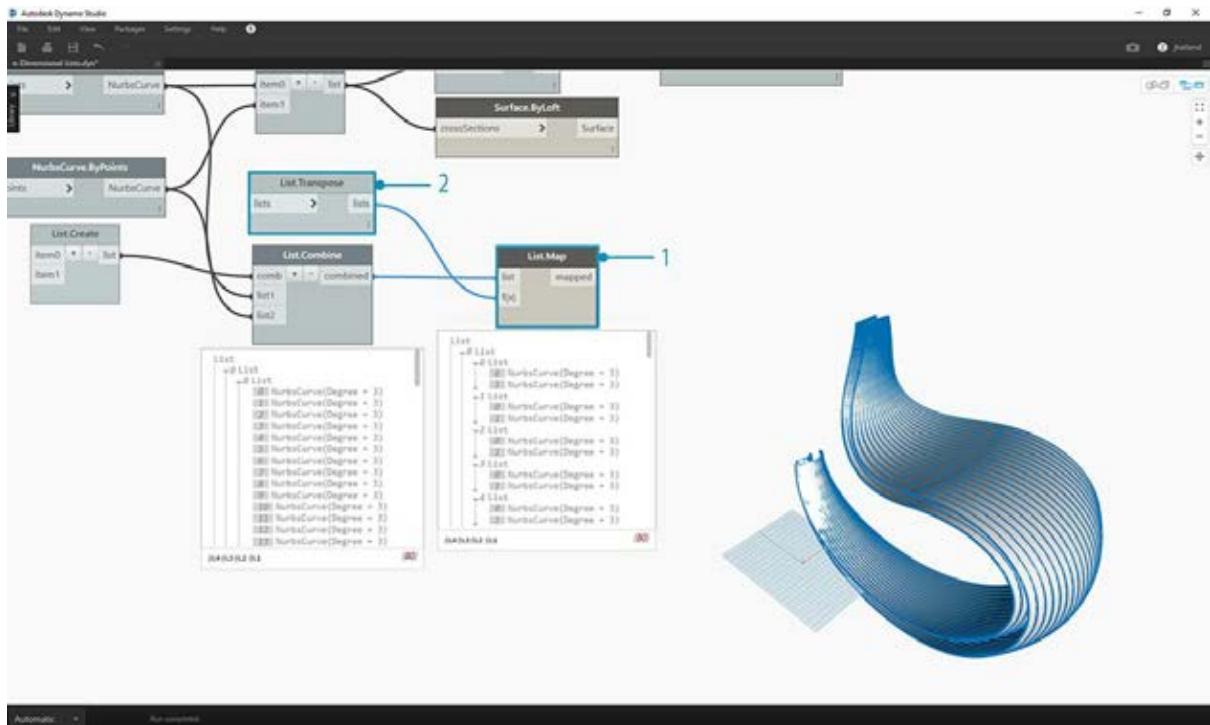
- List.Create ノードを使用して、NURBS 曲線を 1 つのデータ構造にマージします。これにより、「リストのリストのリスト」が作成されます。
- Surface.ByLoft ノードを接続すると、元のサーフェスが取得されます。これは、元のデータ構造から作成された独自のリスト内にサーフェスがそのまま残っているためです。



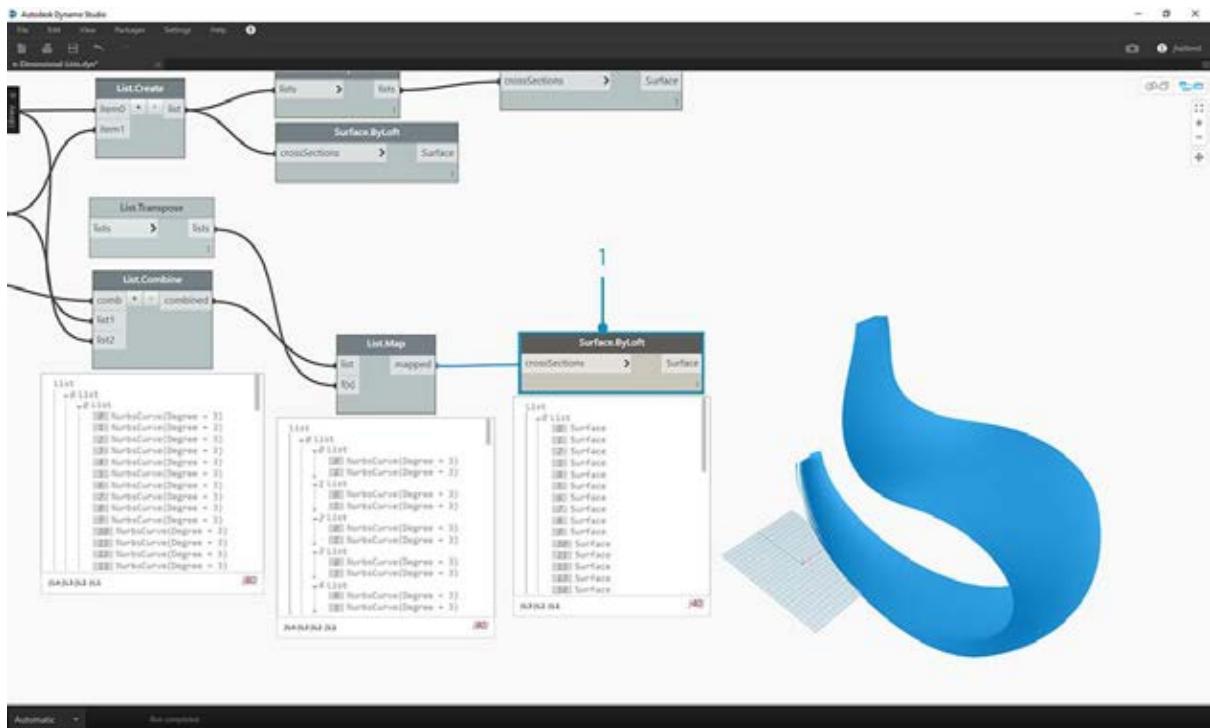
- 前の演習では、*List.Transpose* ノードを使用してリブ形状の構造物を作成しましたが、ここでは同じ操作を行うことはできません。*List.Transpose* ノードを使用できるのは 2 次元のリストの場合ですが、ここでは 3 次元のリストを操作するため、リスト内の列と行を反転する操作を簡単に実行することはできません。これまでに説明したように、リストはオブジェクトとして処理されるため、サブリストが含まれていないリストを *List.Transpose* ノードを使用して反転することはできますが、階層内の 1 段階下の層でリストの NURBS 曲線を反転することはできません。



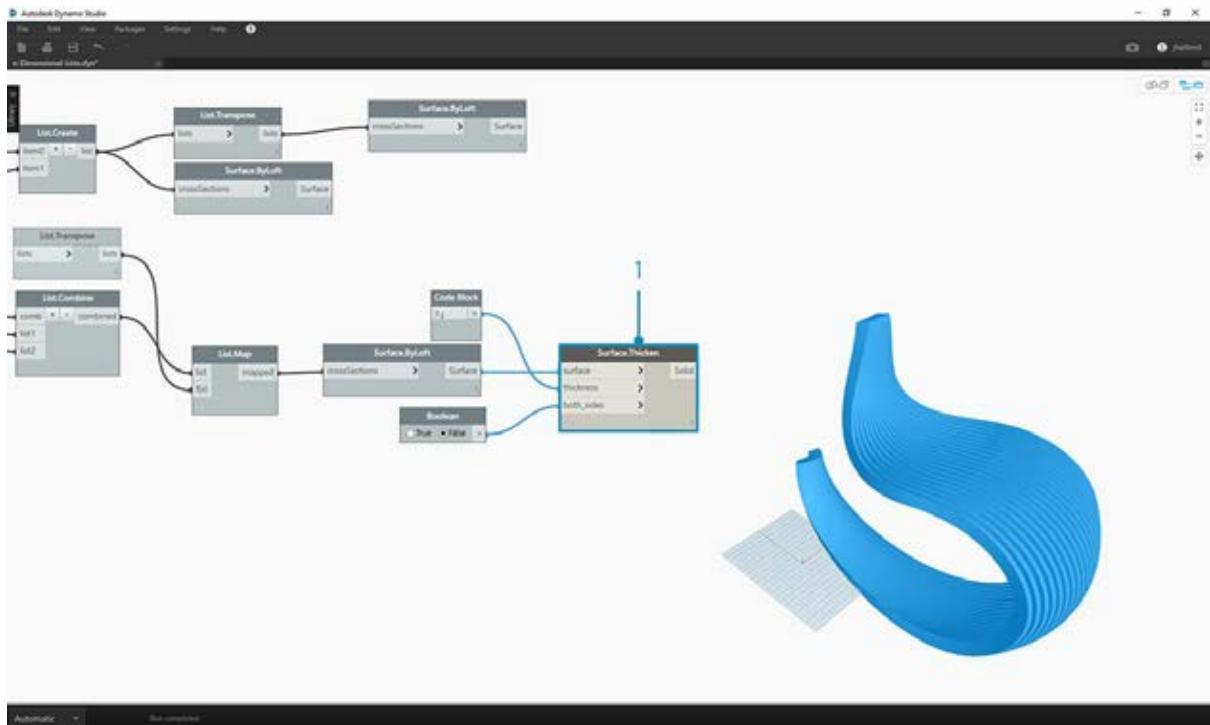
- ここでは、*List.Combine* ノードを使用します。さらに複雑なデータ構造を処理する場合は、*List.Map* ノードと *List.Combine* ノードを使用します。
- List.Create* ノードを「コンビネータ」として使用して、この操作に適したデータ構造を作成します。



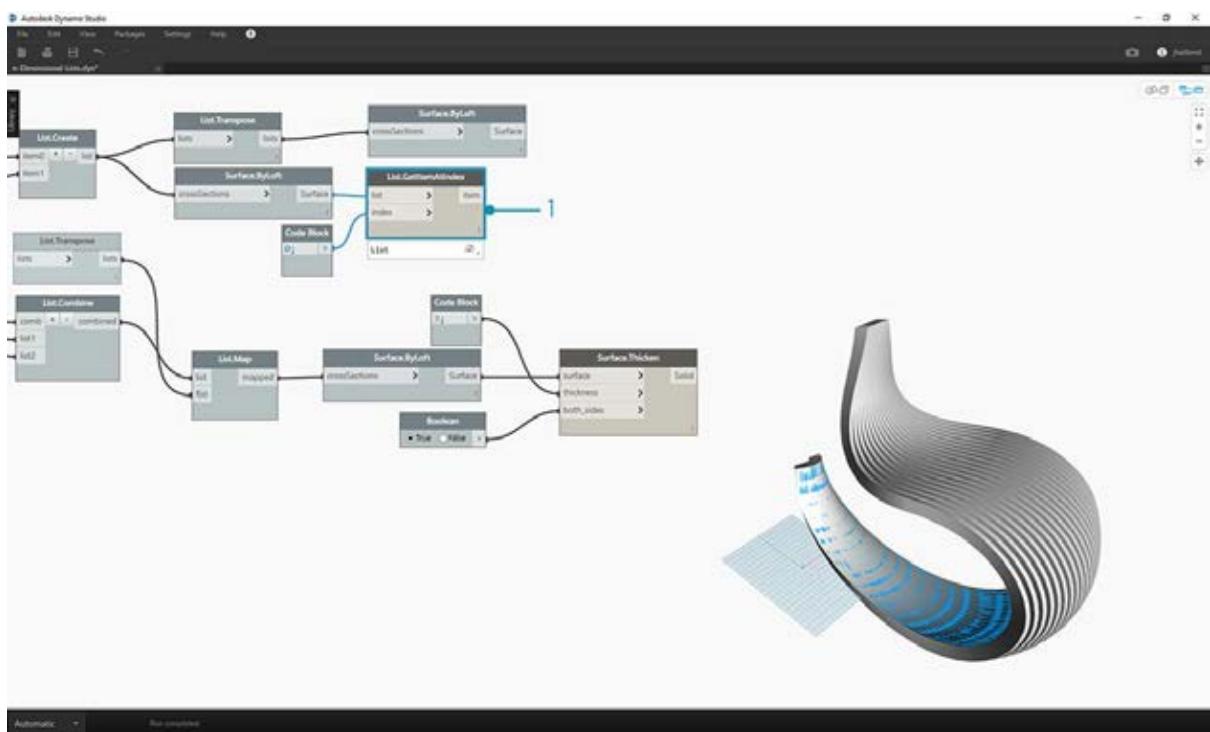
- 階層内の 1 段階下の層で、データ構造を転置する必要があります。これを行うには、*List.Map* ノードを使用します。このノードは *List.Combine* ノードに似ていますが、複数の入力リストではなく 1 つの入力リストだけを使用する点が異なっています。
- List.Map* ノードに *List.Transpose* ノードを適用します。これにより、メインリスト内のサブリストの列と行が反転します。



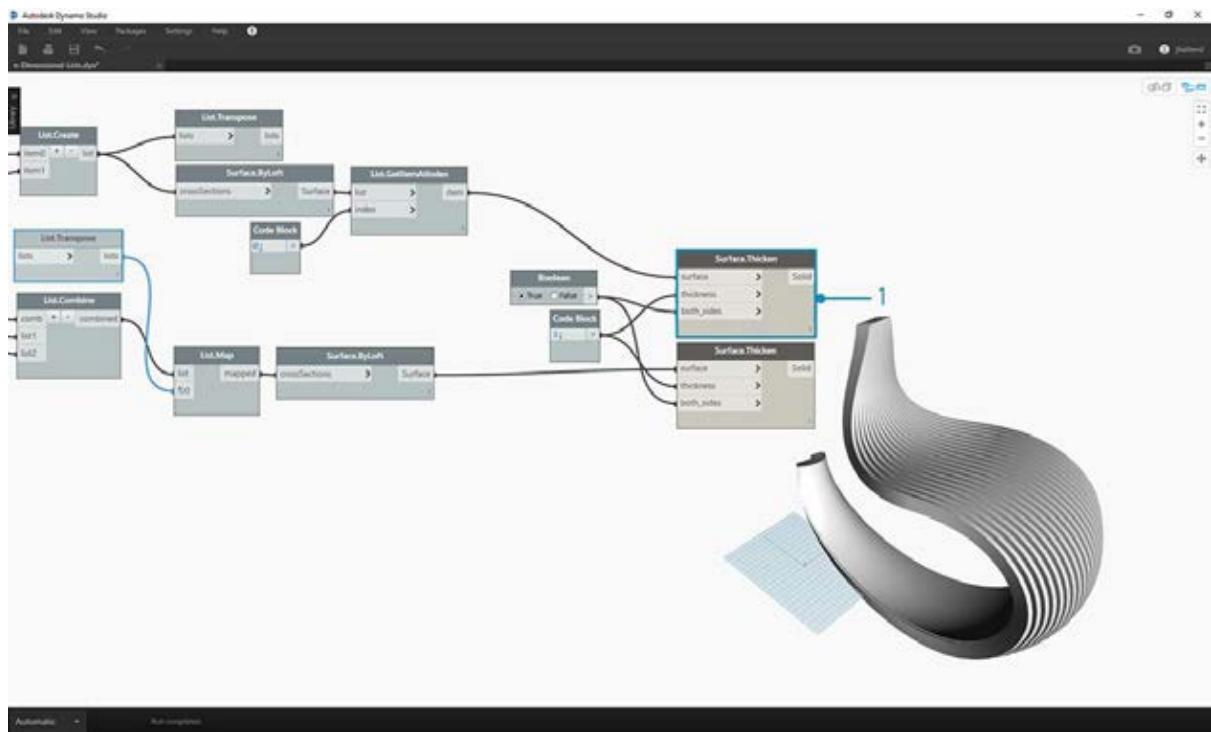
- これで、適切なデータ階層を持つ NURBS 曲線をまとめてロフトし、リブ形状の構造物を作成できるようになりました。



1. ここでは、*Surface.Thicken* ノードを使用して、ジオメトリに深さを設定してみましょう。



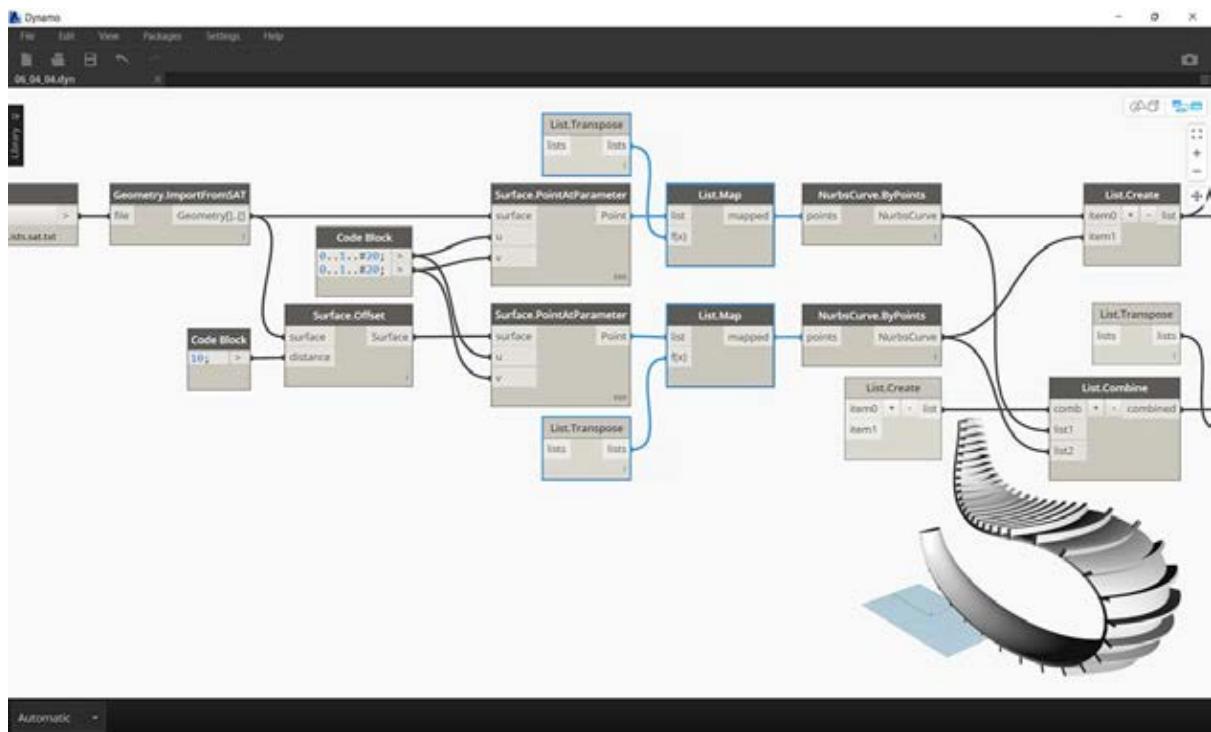
1. この構造物にサーフェスを追加して補強してみましょう。List.GetItemAtIndex ノードを使用して、前の手順でロフトしたサーフェスのうち、背面サーフェスを選択します。



1. 選択したサーフェスに厚みをつければ、操作は完了です。

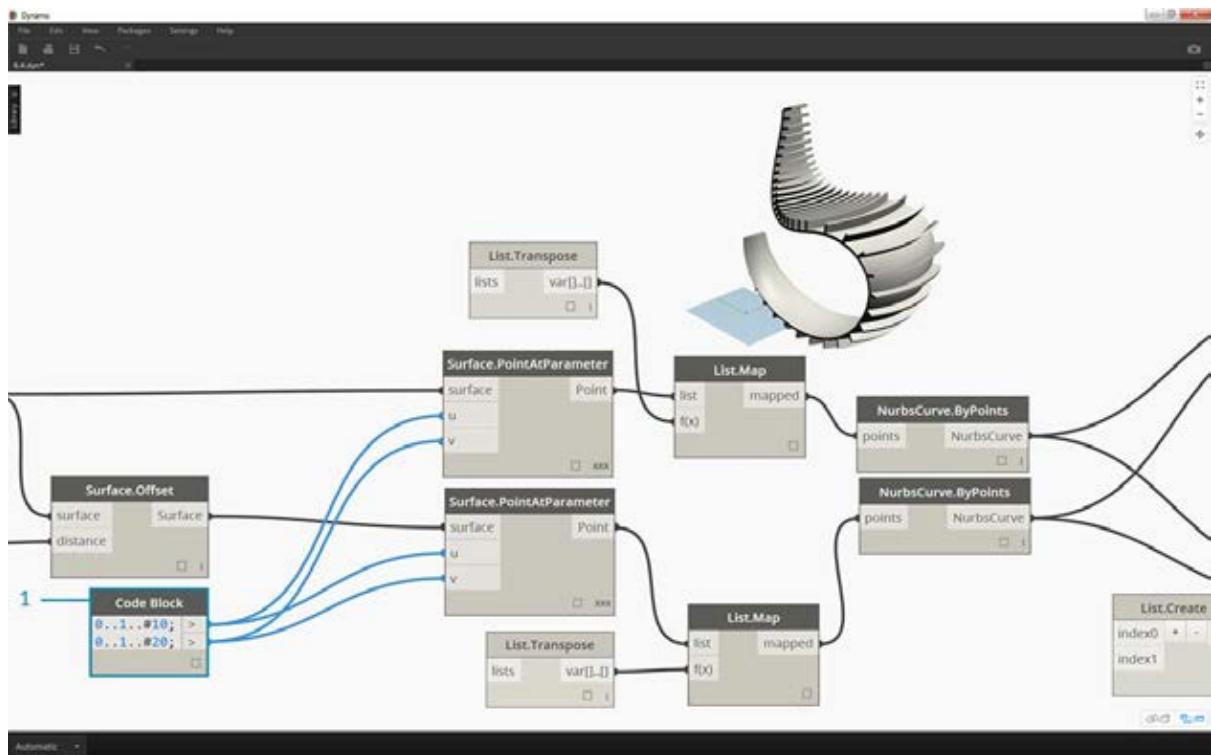


これで、さまざまなデータを使用したロッキング チェアが完成しました。



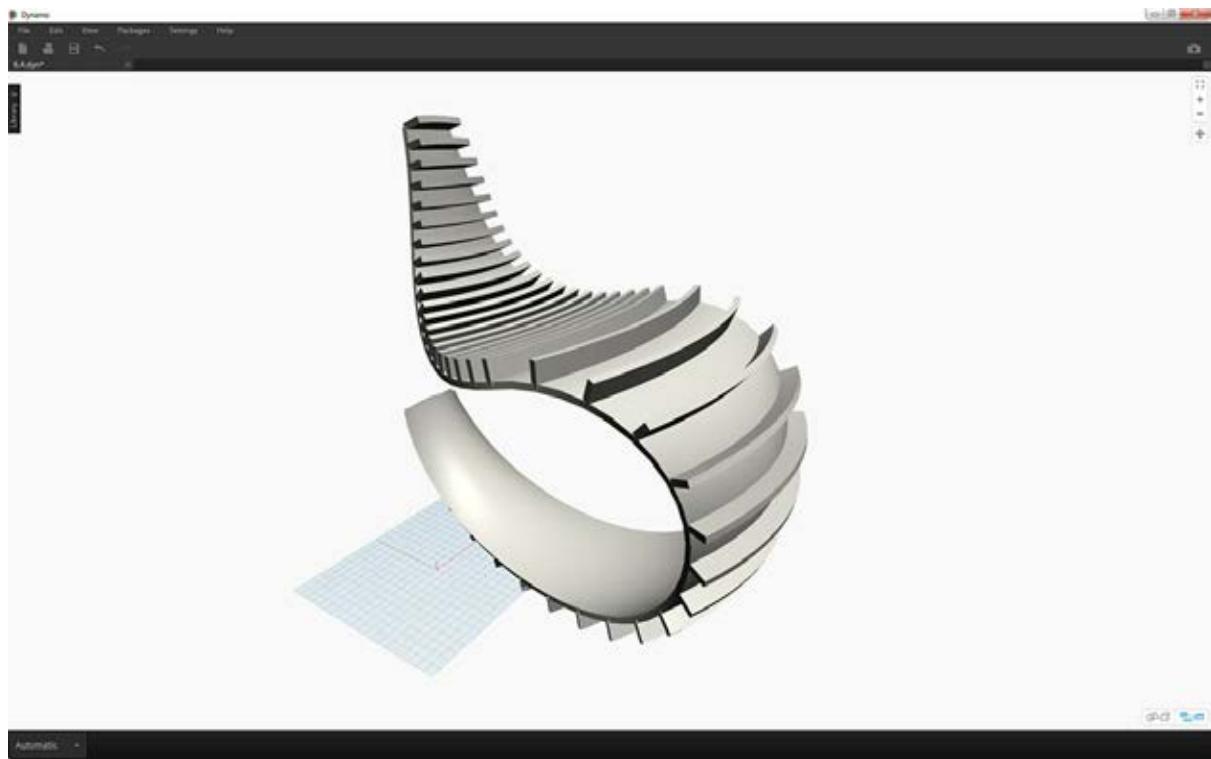
最後に、ロッキング チェアの溝の方向を変えてみましょう。前の演習では転置用のノードを使用しましたが、ここでも同じような方法で操作を行います。

1. データ階層にもう 1 つ深い層があるため、List.Map ノードと List.Transpose ノードを使用して溝の方向を変更します。



1. Code Block ノードのコード行を次のように変更して、溝の数を増やします。

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

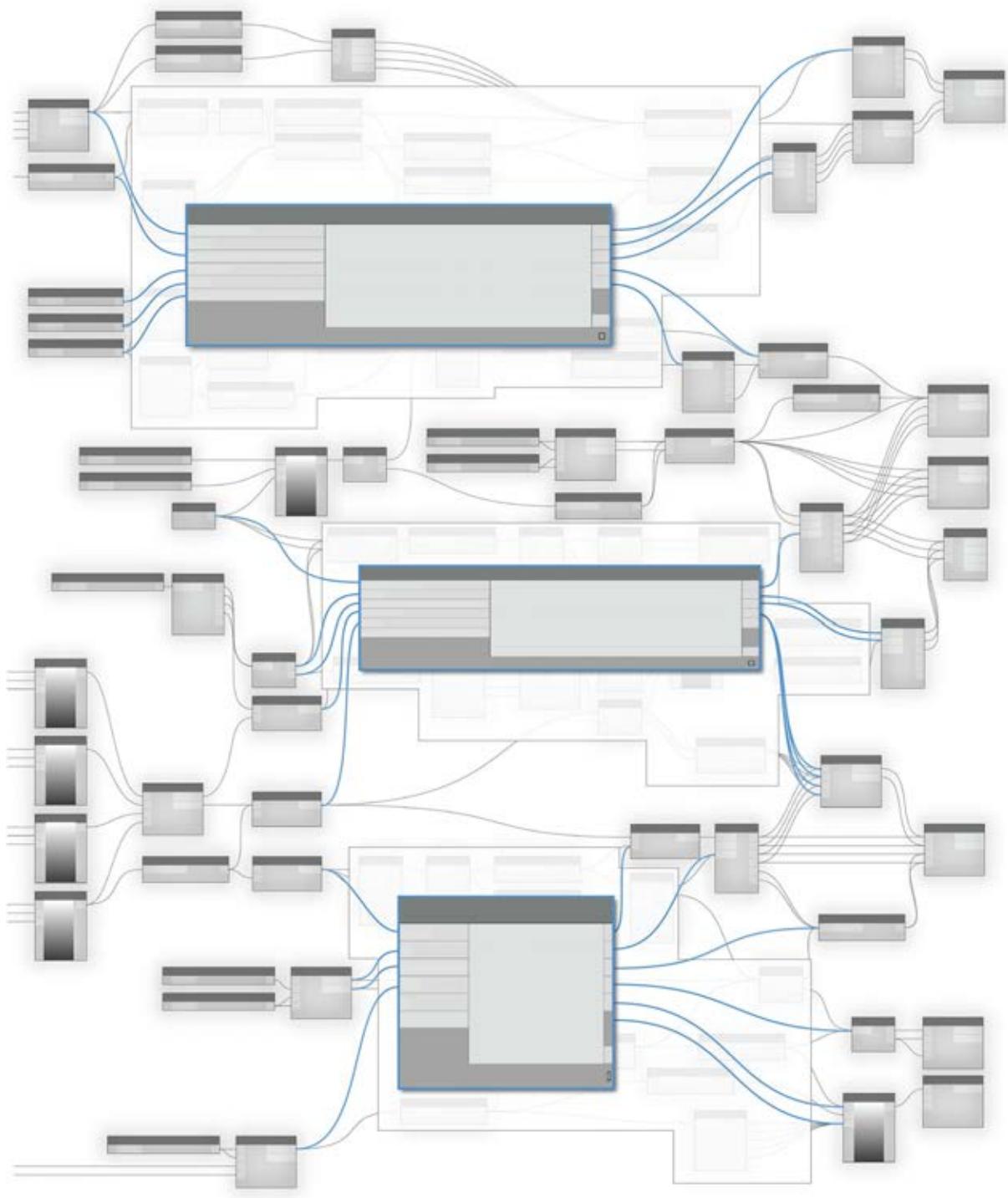


最初に作成したロッキング チェアは滑らかな形状でしたが、このロッキング チェアはまったく異なる形狀になりました。

## コード ブロックと DesignScript

### コード ブロックと DesignScript

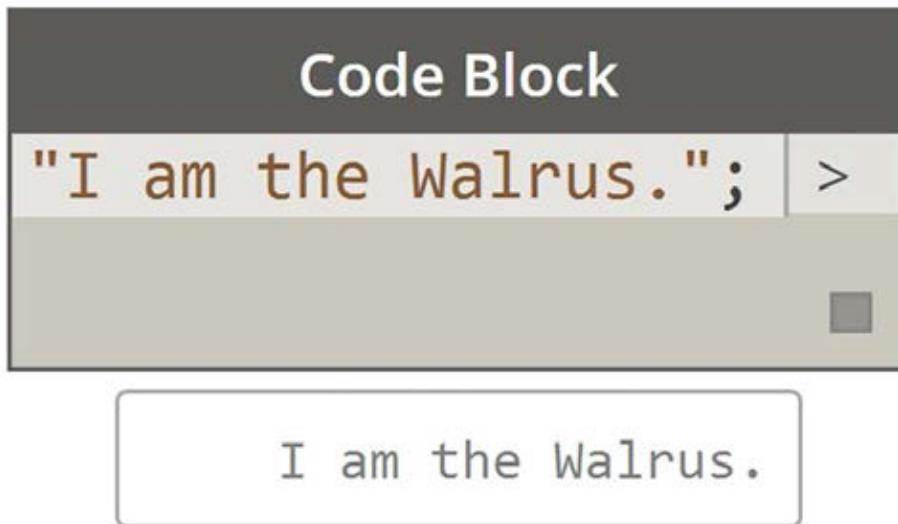
コード ブロックは、ビジュアル プログラミング環境をテキストベースの環境に動的にリンクさせる、Dyanamo 独自の機能です。コード ブロックでは、1 つのノードですべての Dynamo ノードにアクセスし、グラフ全体を定義することができます。コード ブロックは Dynamo の基本的な要素です。この章をよく読んで、コード ブロックを理解してください。



## コード ブロックとは

### コード ブロックとは

コード ブロックを使用すると、Dynamo の核となる DesignScript というプログラミング言語を詳しく理解することができます。予備的な設計ワークフローをサポートするためにまったく新しく開発された DesignScript は、簡潔で理解しやすい言語です。短いコードを迅速に変更することも、大規模で複雑な相互作用に対応することもできます。DesignScript は、Dynamo 内部のほとんどの要素を駆動するエンジンの中核でもあります。Dynamo ノード内のほぼすべての機能と相互作用について、スクリプト言語に対する 1 対 1 の関係が存在するため、ノードベースの相互作用とスクリプトを柔軟に切り替えることができます。これは、Dynamo 独自の機能です。

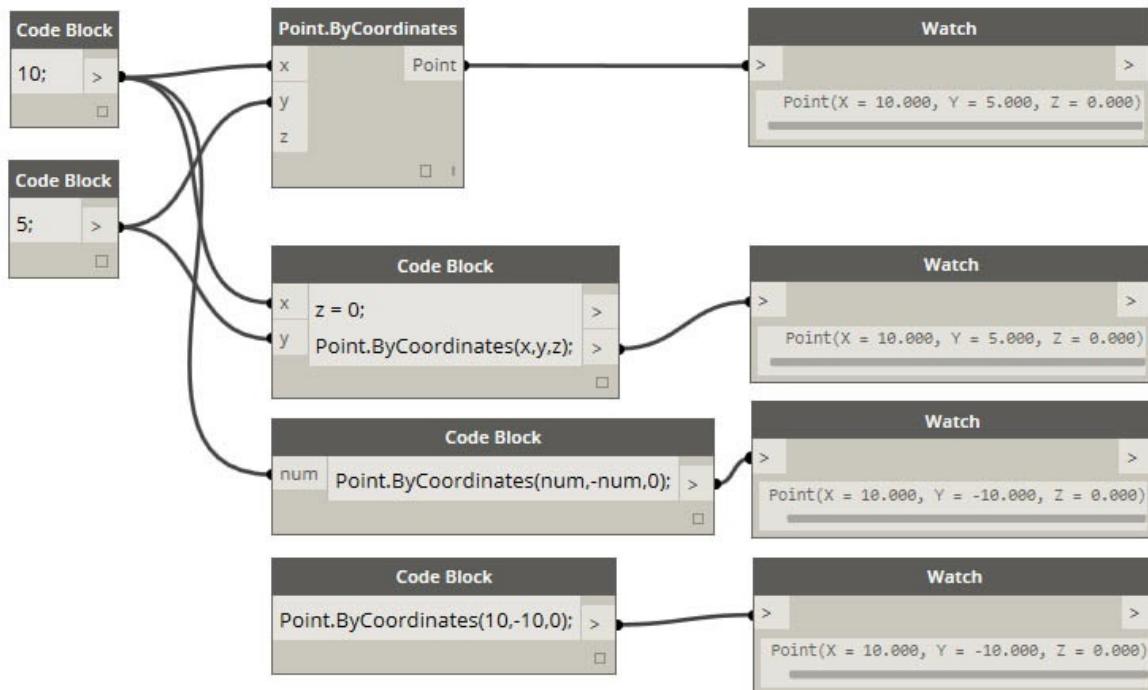


操作に慣れていない初級ユーザのために、ノードを自動的にテキスト構文に変換する機能が用意されています。これにより、DesignScript を簡単に理解したり、グラフの大きな部分のサイズを小さくすることができます。この操作は、「ノードをコード化」という処理を使用して実行されます。この処理については、[DesignScript 構文](#)のセクションで詳しく説明します。操作に慣れている上級ユーザは、コード ブロックで各種の標準的なコードのパラダイムを使用して、既存の機能とユーザ定義の関係を組み合わせてカスタマイズすることができます。初心者ユーザと上級ユーザの間に位置する中級ユーザについては、設計作業をすばやく行うためのさまざまなショートカットやコード スニペットが用意されています。プログラミング経験のないユーザにとって、「コード ブロック」という言葉は難しそうに感じるかもしれません、使用方法も簡単で、安定して動作します。初級ユーザは、最小限のコードを記述してコード ブロックを効率的に使用でき、上級ユーザは、Dynamo 定義内の任意の場所で呼び出すことのできるスクリプト定義を作成することができます。

### コード ブロックの概要

コード ブロックとは、一言でいうと、ビジュアル スクリプティング環境内で使用するテキスト スクリプティング インタフェースです。コード ブロックは、数値、文字列、式などのデータ タイプとして使用することができます。コード ブロックは Dynamo 専用に設計されているため、コード ブロック内で任意の変数を定義することができます。定義した変数は、自動的にノードの入力に追加されます。

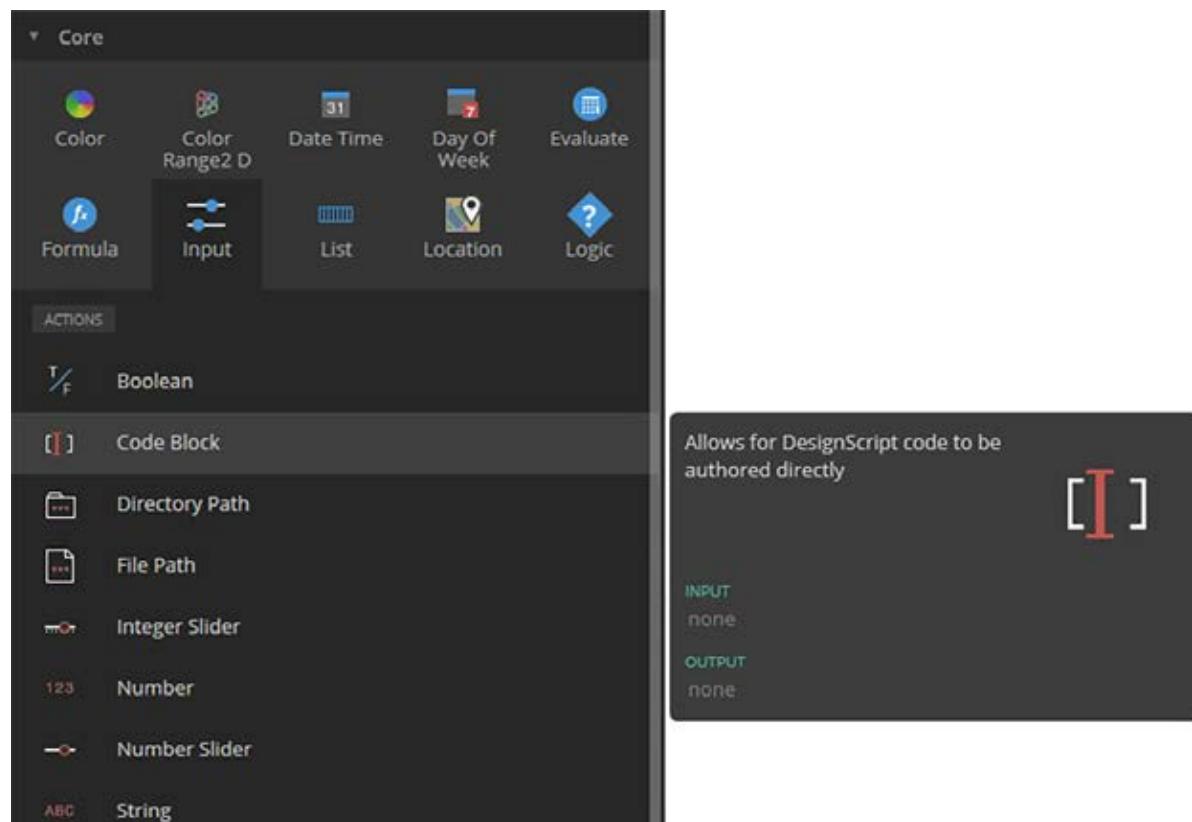
コード ブロックを使用すると、入力値の指定方法を柔軟に決定することができます。 $(10, 5, 0)$  という座標の基本的な点を作成するには、次のようにいくつかの方法があります。



ライブラリ内の使用可能な各種の関数に慣れてくると、「Point.ByCoordinates」と入力する方が、ライブラリ内を検索して目的のノードを探すよりも早いということがわかるようになります。たとえば「Point」と入力すると、点に対して適用できる関数のリストが表示されます。これにより、直感的にスクリプトを作成することができ、Dynamoで関数を適用する方法を理解することができます。

### Code Block ノードを作成する

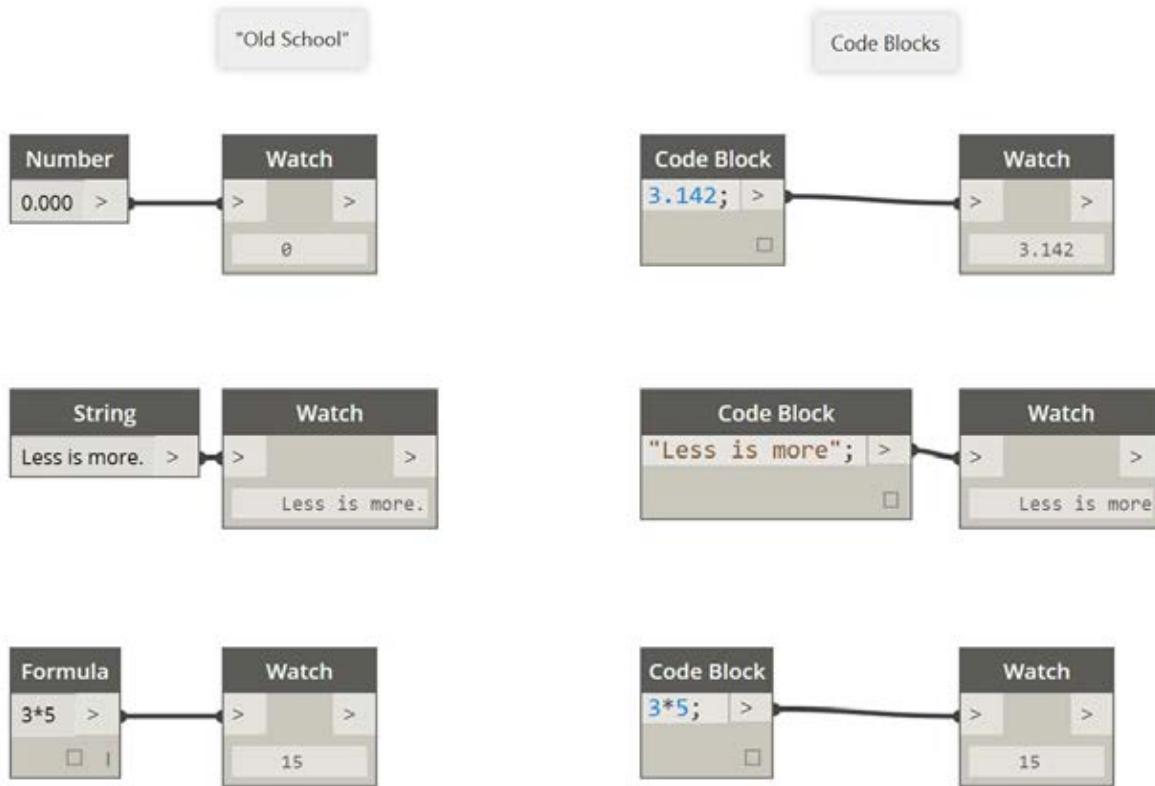
Code Block ノードは、[Core] > [Input] > [Actions] > [Code Block]で使用することができます。または、キャンバス上でダブルクリックするだけで Code Block ノードが表示されます。この方が簡単です。Code Block ノードは頻繁に使用されるため、キャンバス上でダブルクリックするだけで Code Block ノードが表示されるようになっています。



## 数値、文字列、式

Code Block ノードを使用すると、データ タイプについても柔軟に操作することができます。Code Block ノードを使用して、数値、文字列、式をすばやく定義し、目的の値を出力することができます。

次の図を見ると、値を定義するための従来の方法(左側の方法)は少し冗長であることがわかります。この方法では、インターフェース内で目的のノードを検索してキャンバスに追加し、データを入力する必要があります。Code Block ノードを使用すると、キャンバス上でダブルクリックするだけで Code Block ノードを表示し、基本的な構文で正しいデータ タイプを入力することができます。



Code Block ノードと比べた場合、Number、String、Formula の各ノードは旧式のノードということができます。

# DesignScript 構文

## DesignScript 構文

Dynamo のノード名に共通の形式があることにお気付きでしょうか。各ノードはスペースなしの「.」構文を使用しています。各ノードの最上部の文字列がスクリプトの実際の構文を表しており、「.」(ドット表記)が呼び出し可能なメソッドと要素を区切っています。これにより、ビジュアル スクリプトから文字ベースのスクリプトに簡単に変換することができます。



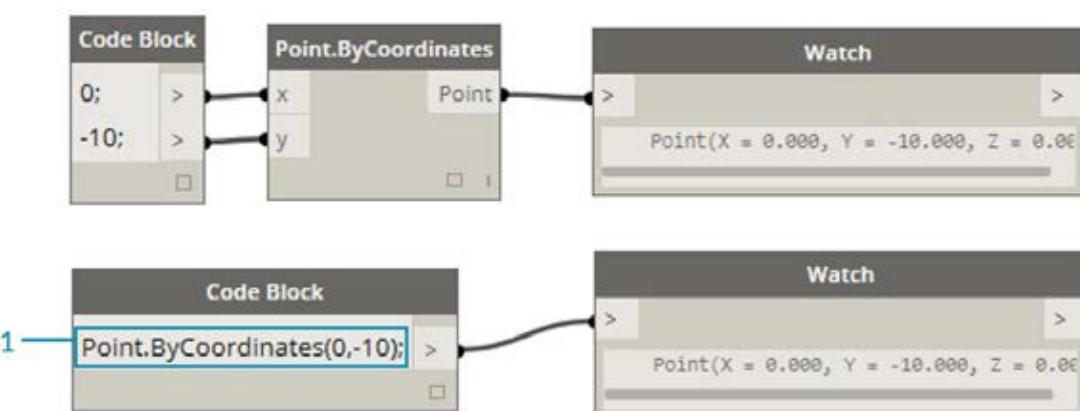
ドット表記の一般的な使用例として、apple (リンゴ)というパラメータが Dynamo でどのように処理されるか見てみましょう。以下は、リンゴを食べる前にリンゴに対して実行するいくつかのメソッドを示しています。(注意: これらは Dynamo の実際のメソッドではありません)。

説明	ドット表記	出力
リンゴは何色ですか?	Apple.color	red
リンゴは熟していますか?	Apple.isRipe	true
リンゴの重さはどのくらいですか?	Apple.weight	6 oz
リンゴはどこで生まれましたか?	Apple.parent	tree(木)
リンゴは何を作りますか?	Apple.children	seeds(種)
このリンゴは地元で育てられましたか?	Apple.distanceFromOrchard	60 mi.

読者がどう思われるかはわかりませんが、上記の表の出力から判断すると、私にはおいしいリンゴのように見えます。Apple.eat() を実行しようと思います。

### コード ブロックのドット表記

リンゴの例を念頭に置きながら、Point.ByCoordinates ノードでコード ブロックを使用して点を作成する方法を見てみましょう。



*Code Block* ノードの構文 `Point.ByCoordinates(0, 10);` は、Dynamo の `Point.ByCoordinates` ノードと同じ結果を生成します。ただし、1 つのノードを使用して点を作成できるという違いがあります。`x` 入力と `y` 入力に異なるノードを接続する必要がないため、より効率的です。

1. コード ブロックで `Point.ByCoordinates` を使用することにより、初期設定のままのノード(`x,y`)と同じ順番で入力を指定します。

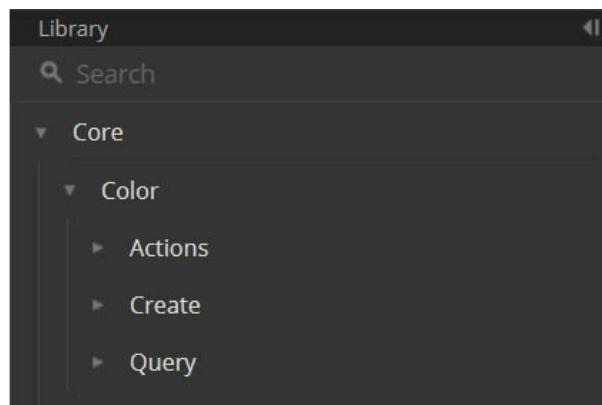
## ノードを呼び出す

ライブラリ内の通常のノードは、*Code Block* を通じて呼び出すことができます。ただし、特別なユーザ インタフェース機能を持つ特別な UI ノードは呼び出しができません。たとえば、`Circle.ByCenterPointRadius` を呼び出すことはできますが、`Watch 3D` ノードを呼び出すことは意味がありません。

通常の(ライブラリのほとんどの)ノードは一般的に 3 つのタイプに分けられます。

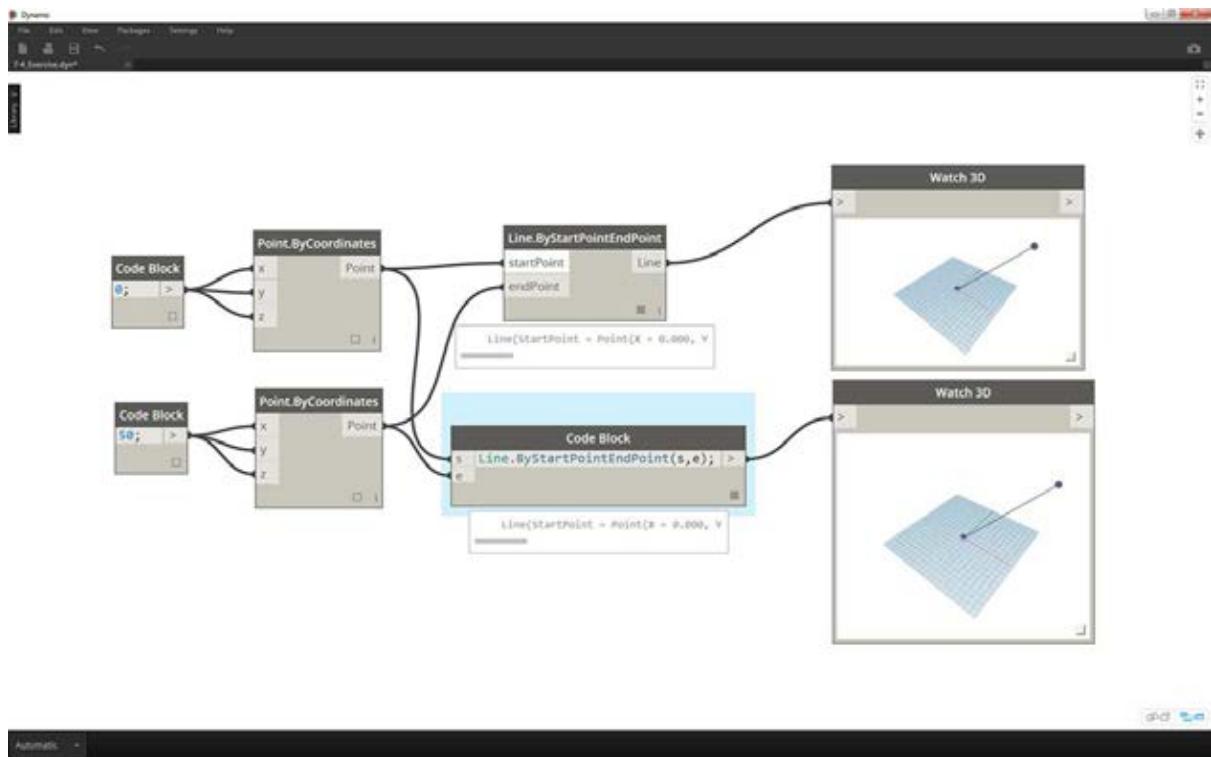
- **Create** - 何かを作成(または構築)します
- **Action** - 何かに対してアクションを実行します
- **Query** - 既に存在する何かのプロパティを取得します

ライブラリはこれらのカテゴリに基づいて編成されています。これらの 3 つのタイプのメソッド(ノード)では、*Code Block* で呼び出されるときの処理がそれぞれ異なります。



### Create

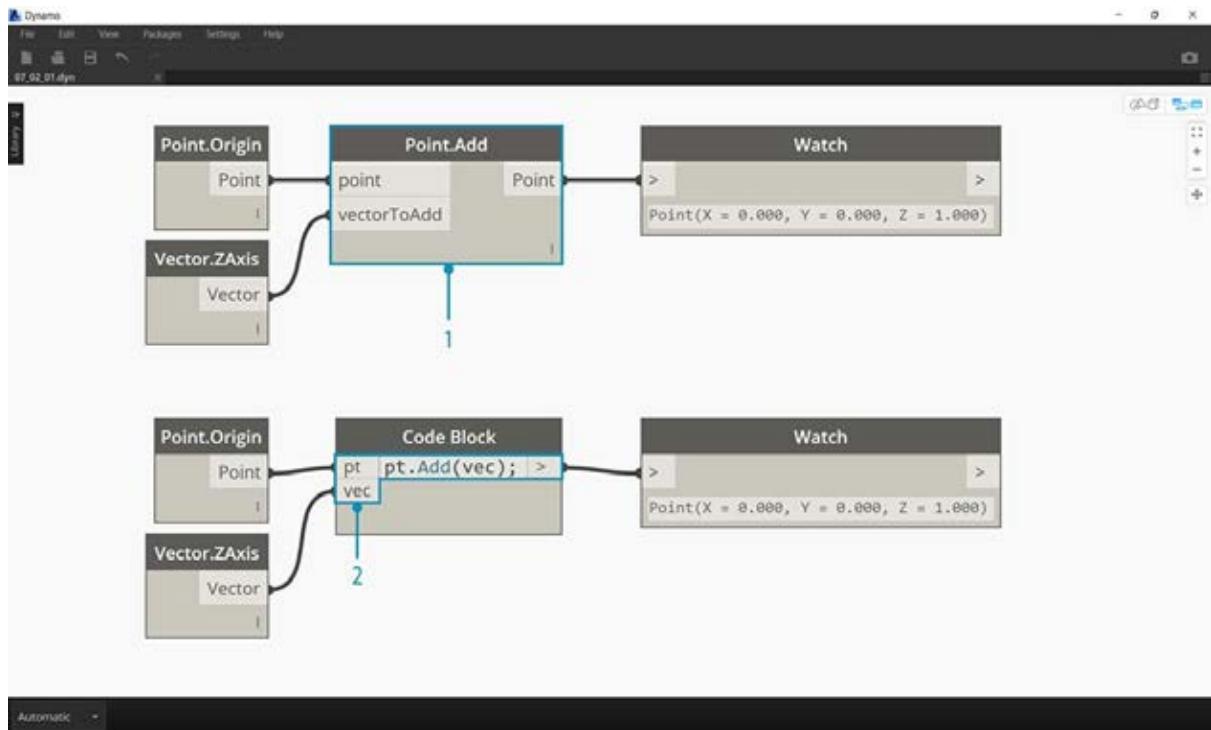
[Create] カテゴリはジオメトリを一から構築します。コード ブロックには値を左から右に入力します。これらの入力の順番は、上から下に入力するノードの入力の順序と同じです。



*Line.ByStartPointEndPoint* ノードと、コード ブロック内に対応する構文とでは、同じ結果が生成されます。

### Action

このタイプのオブジェクトに対してはアクションを行います。Dynamo は(多くのコーディング言語で一般的な)ドット表記を使用して、対象に対してアクションを適用します。対象とするオブジェクトを決めたら、ドットに続けてアクション名を入力します。Action タイプのメソッドの入力は、Create タイプのメソッドと同様、括弧で囲んで指定されます。ただし、対応するノードの最初の入力を指定する必要はありません。代わりに、アクションの実行対象の要素を指定します。

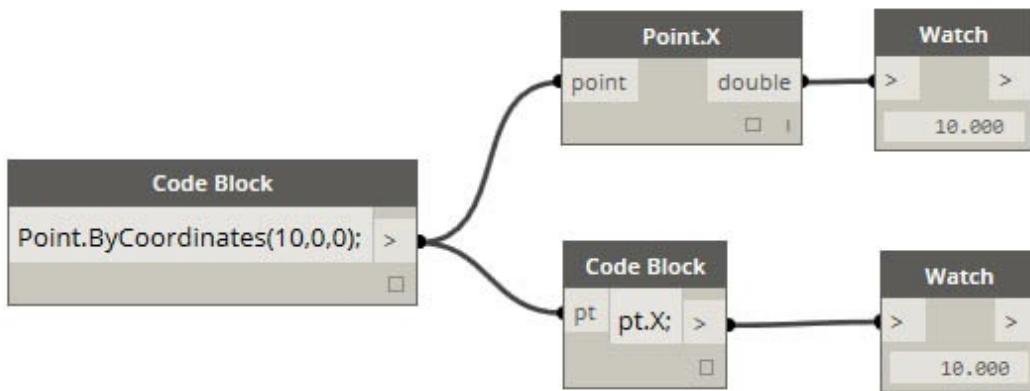


1. *Point.Add* ノードは Action タイプのノードであるため、構文の動作が多少異なります。
2. 入力は(1) *point* と、(2)その点に追加する *vector* の 2 つです。Code Block では、点(対象)に「*pt*」という名前を付け

ています。「`vec`」という名前のベクトルを「`pt`」に追加するには、`pt.Add(vec)`、つまり「対象、ドット、アクション」の形式で書き込みます。Add アクションの入力は 1 つのみであるか、または `Point.Add` ノードの最初の入力を除くすべての入力になります。`Point.Add` ノードの最初の入力は点自体です。

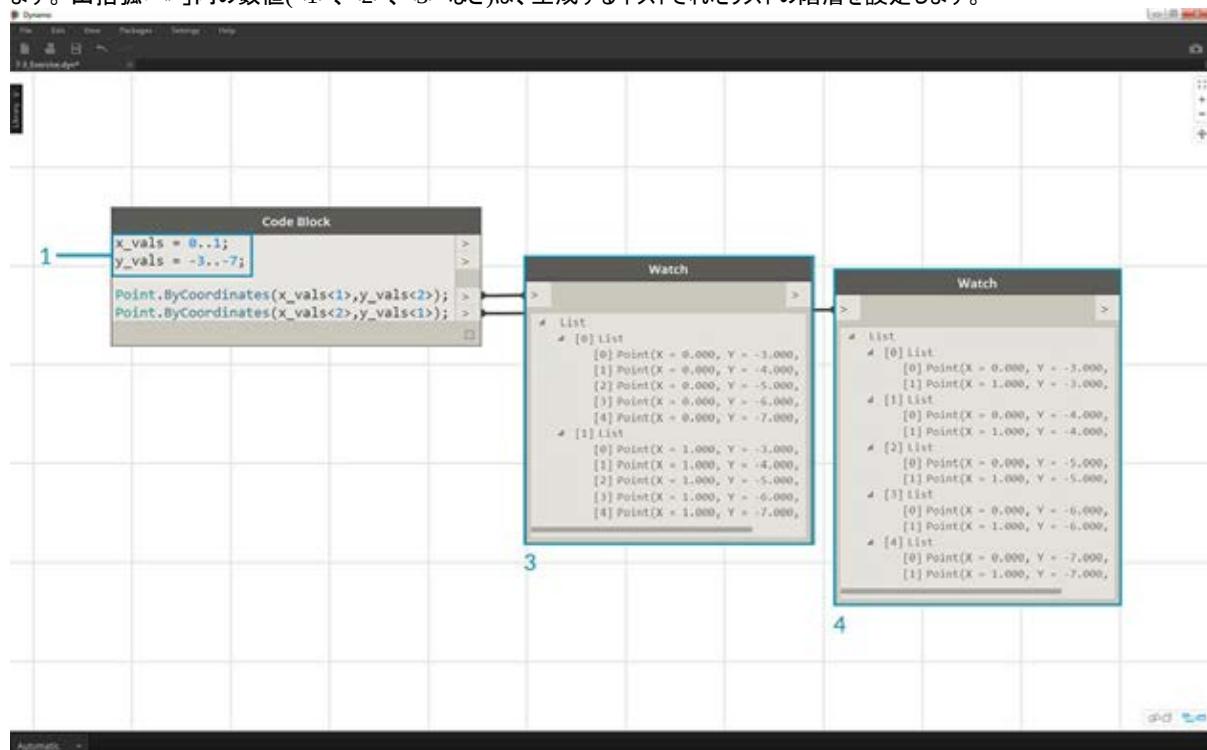
## Query

Query タイプのメソッドはオブジェクトのプロパティを取得します。オブジェクト自体が入力であるため、入力を指定する必要はありません。括弧は必要ありません。



## レーシングを適用する

ノードを使用するレーシングと、コード ブロックを使用するレーシングとは少々異なります。ノードを使用する場合、ユーザはノードを右クリックして、実行するレーシング オプションを選択します。コード ブロックを使用する場合、ユーザはデータを構築する方法をより詳しくコントロールできます。コード ブロックの省略表記では、複製ガイドを使用して、複数の 1 次元リストをペアリングする方法を設定します。山括弧「`<>`」内の数値(`<1>`、`<2>`、`<3>`など)は、生成するネストされたリストの階層を設定します。



1. この例では、省略表記で 2 つの範囲を設定します(省略表記については、この章の次のセクションで詳しく説明します)。簡単に説明すると、`0..1` は `{0, 1}` に相当し、`-3..-7` は `{-3, -4, -5, -6, -7}` に相当します。この結果、2 つの `x` 値と 5 つの `y` 値のリストが返されます。このような範囲が一致していないリストを持つ複製ガイドを使用しない場合は、2 つの点が含まれるリストが返され、リストの長さは短い方の長さになります。複製ガイドを使用すると、2 つの値と 5 つの値を組み合わせたすべての座標(または外積)を表示できます。
2. 構文 `Point.ByCoordinates(x_vals<1>,y_vals<2>);` を使用すると、5 つの項目を含む 2 つのリスト

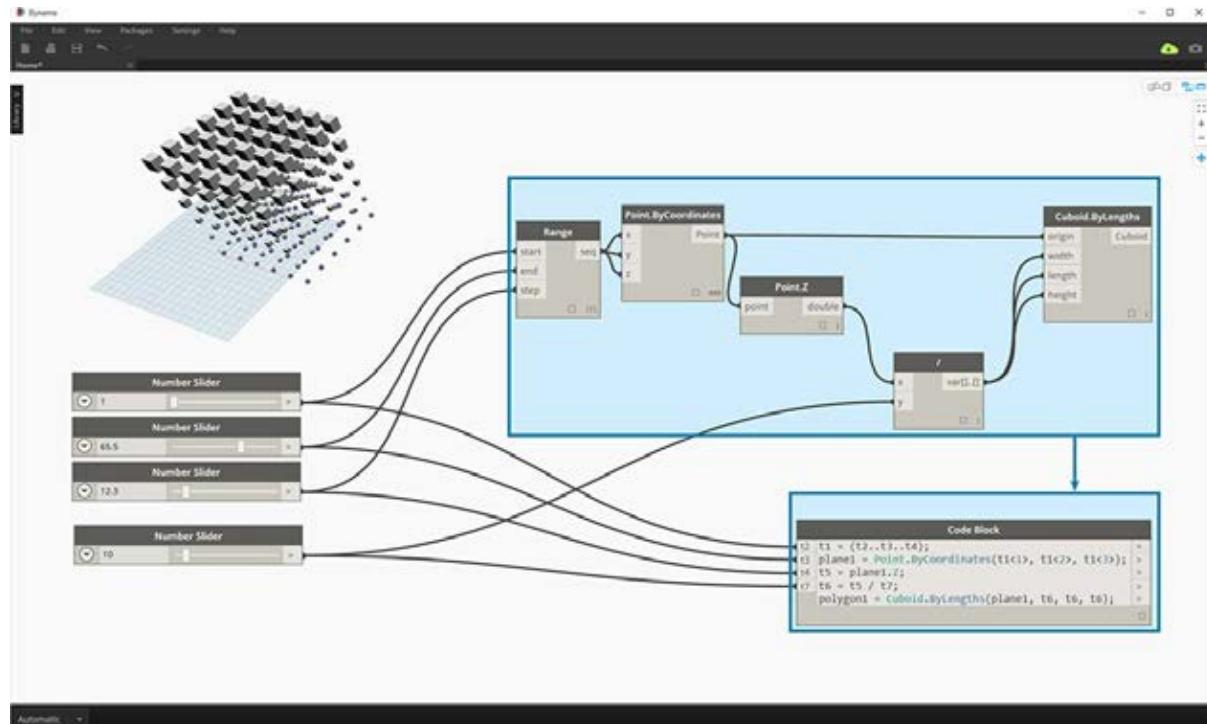
が生成されます。

- 構文 `Point.ByCoordinates(x_vals<2>, y_vals<1>);` を使用すると、2つの項目を含む5つのリストが生成されます。

この表記では、5つの項目を持つ2つのリストと2つの項目を持つ5つのリストのどちらを優先リストにするか指定することもできます。例では、複製ガイドの順番を変更することにより、グリッドの点の行のリストまたは列のリストを生成できます。

## ノードをコード化

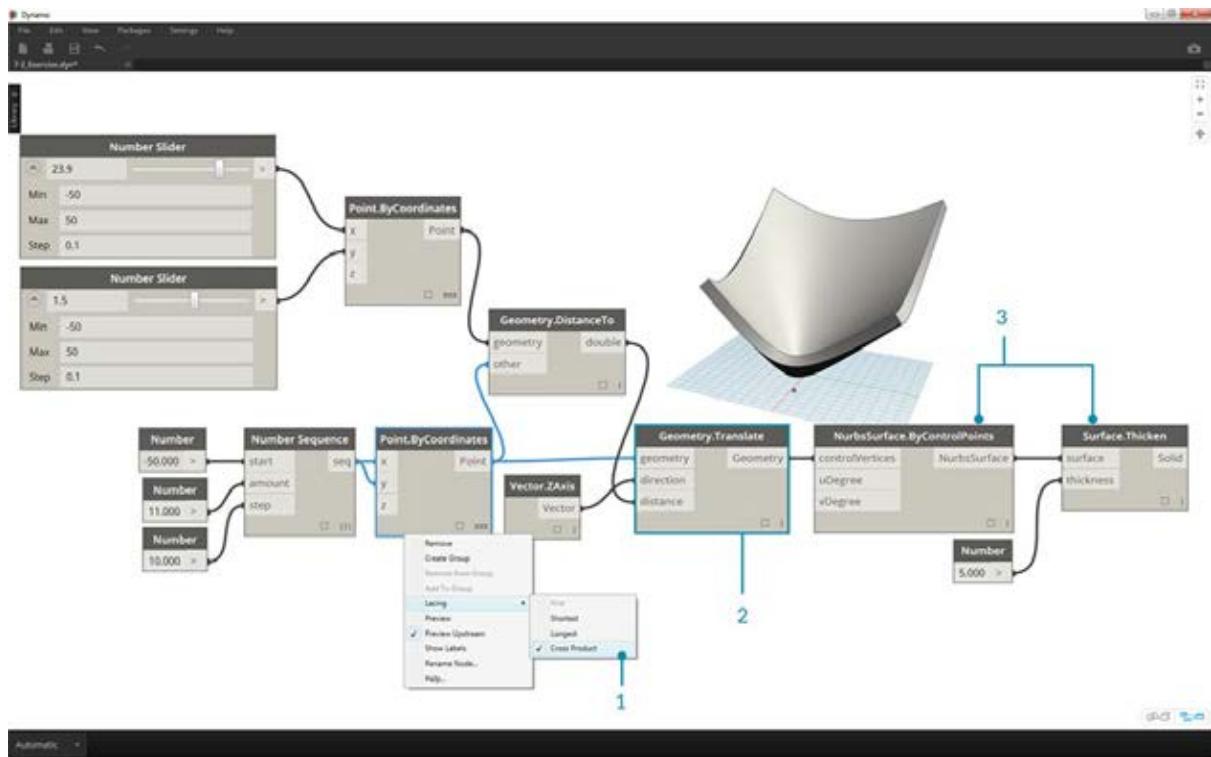
コード ブロックを使用する上記の方法を使用するには多少の慣れが必要ですが、Dynamo には処理を容易にする[ノードをコード化]という機能があります。この機能を使用するには、Dynamo グラフでノードの配列を選択し、キャンバスを右クリックして、[ノードをコード化]を選択します。Dynamo はこれらのノードを、すべての入力と出力を含め、1つのコード ブロックに統合します。このツールはコード ブロックの学習に役立つだけでなく、より効率的にパラメトリックな Dynamo グラフの使用を可能にします。次の演習では最後に[ノードをコード化]を使用します。



## 演習

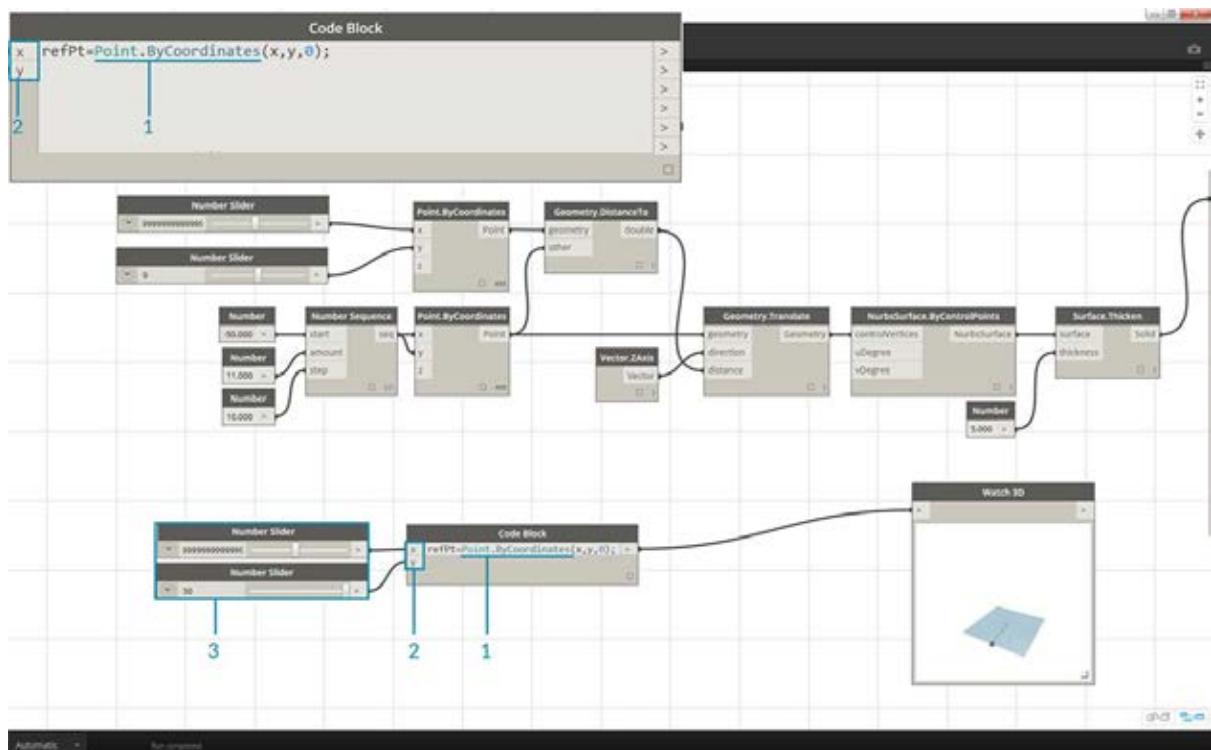
この演習に付属しているサンプル ファイルをダウンロードしてください(右クリックして[名前を付けてリンク先を保存]を選択)。すべてのサンプル ファイルの一覧については、付録を参照してください。[Dynamo-Syntax\\_Attractor-Surface.dyn](#)

コード ブロックの性能を紹介するため、既存のアトラクタ フィールドの設定をコード ブロックの形式に変換します。既存の設定を使用することにより、コード ブロックがビジュアル スクリプトとどのように関連付けられているかを確認し、DesignScript 構文について学習します。

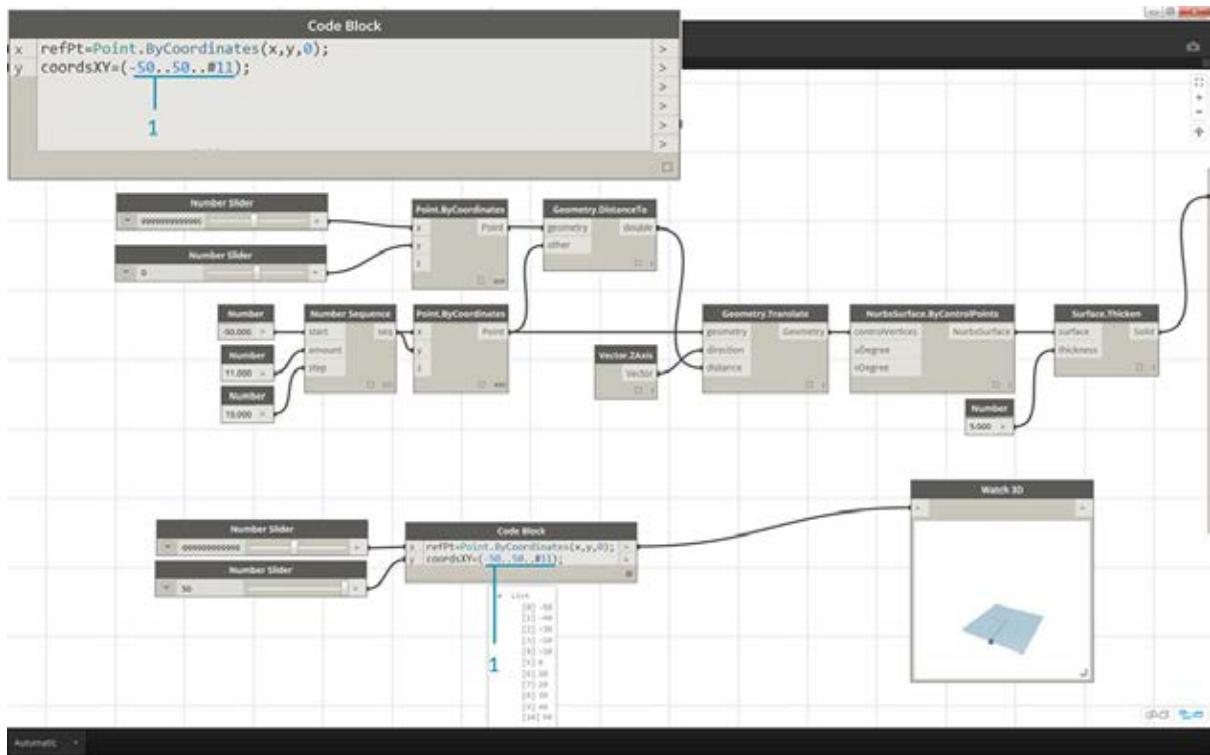


まず、上記の画像の設定を再作成します(またはサンプル ファイルを開きます)。

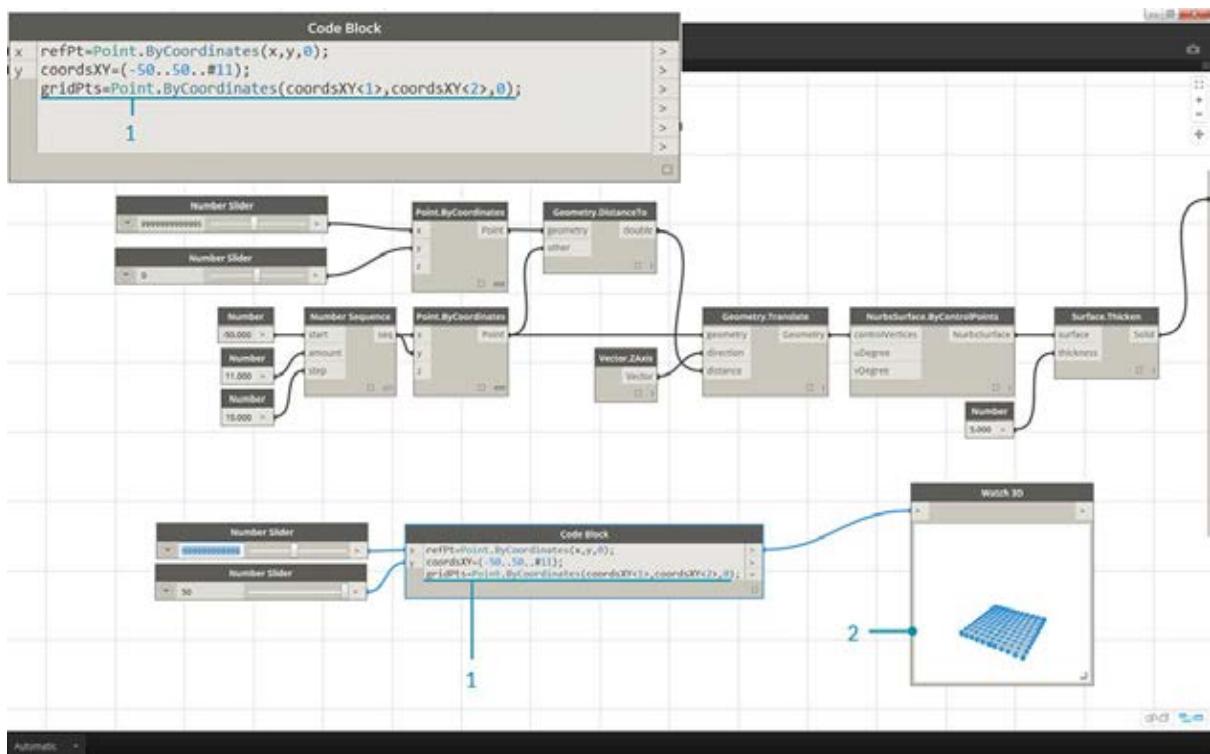
1. *Point.ByCoordinates* ノードのレーシングが外積に設定されていることに注目してください。
2. グリッド内の各点は、参照点までの距離に基づいて Z の正の向きに移動します。
3. サーフェスが再作成されて厚みが付けられ、参照点までの距離を基準にしてジオメトリ内にふくらみが作成されます。



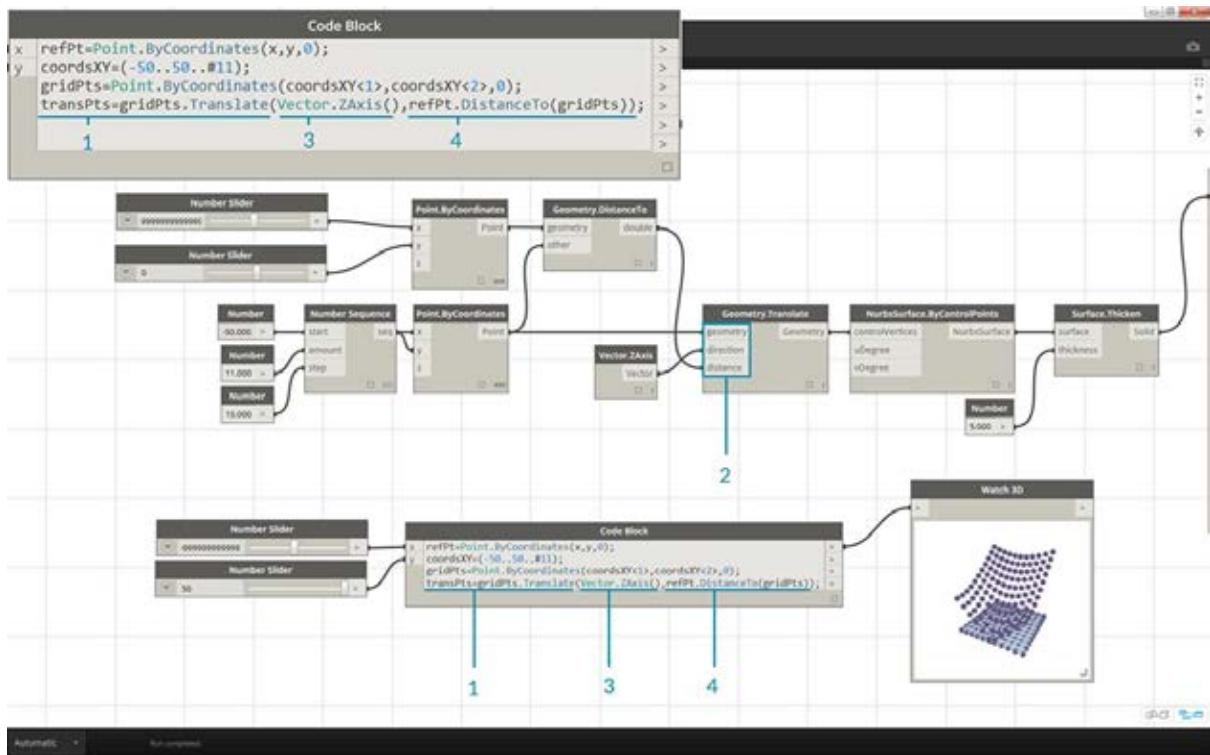
1. 初めからやり直して、まず参照点 *Point.ByCoordinates(x, y, 0)* を設定しましょう。参照点ノードの最上部で指定したものと同じ *Point.ByCoordinates* 構文を使用します。
2. コード ブロックに変数 *x* と *y* が挿入され、スライダを使用してこれらを動的に更新できます。
3. -50から50までの範囲の *Number Slider* ノードをいくつか *Code Block* ノードの入力に追加します。これにより、既定の Dynamo グリッド全体を使用できます。



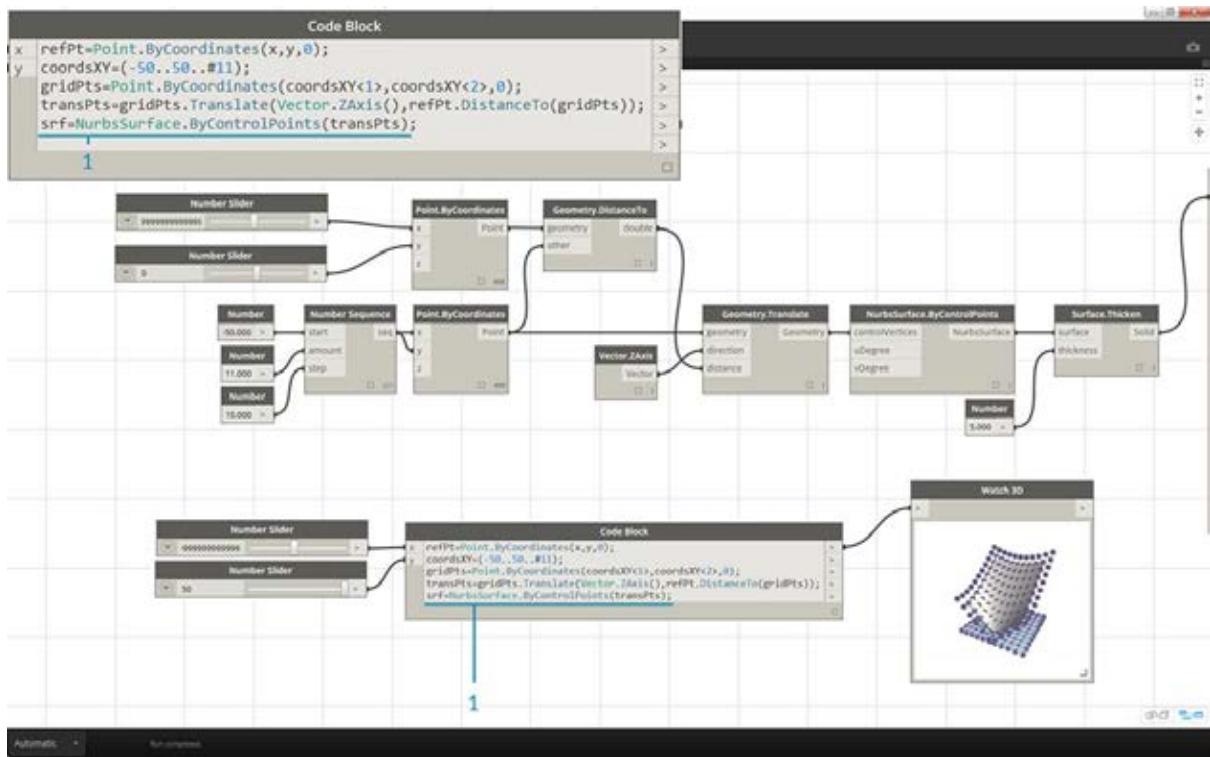
1. *Code Block* ノードの 2 行目では、数値シーケンスノードを置き換える省略表記を設定します: `coordsXY = (-50..50..#11);`。次のセクションではこの方法について詳しく説明します。ここでは、この省略表記がビジュアルスクリプトの *Number Sequence* ノードに相当していることを確認してください。



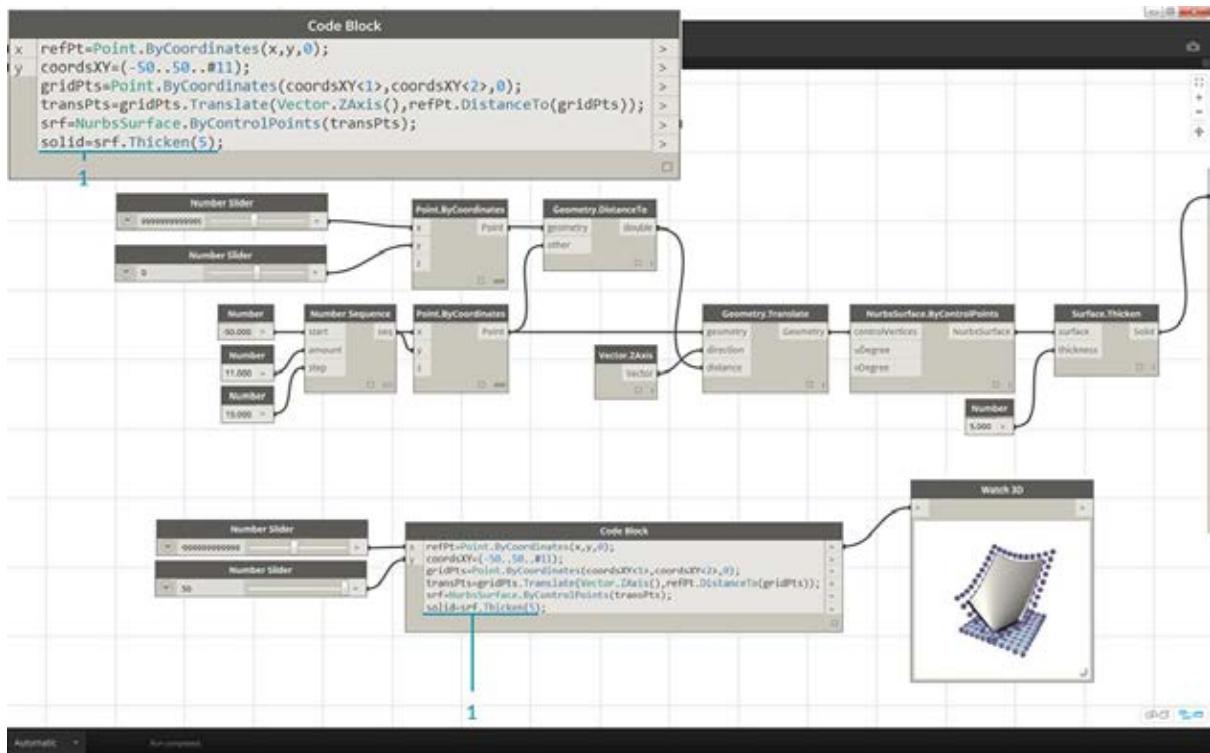
1. 次に、*coordsXY* シーケンスから点のグリッドを作成します。これを行うには *Point.ByCoordinates* 構文を使用します。ただし、ビジュアルスクリプトで実行した場合と同様、リストにある外積を作成する必要があります。これを行うには、次の行を入力します: `gridPts = Point.ByCoordinates(coordsXY<1>,coordsXY<2>,0);`。山括弧は外積参照を意味します。
2. *Watch3D* ノードには Dynamo のグリッド全体に点のグリッドが表示されます。



1. 少し難しくなってきますが、ここで、点のグリッドを参照点までの距離に基づいて上方向に移動する必要があります。まず新しい点のセット `transPts` を呼び出しましょう。変換は既存の要素に対するアクションであるため、`Geometry.Translate...` を使用する代わりに `gridPts.Translate` を使用します。
2. キャンバス上の実際のノードから、3つの入力があることが読み取れます。変換するジオメトリは既に宣言されています。この要素に対して、`gridPts.Translate` を使用してアクションを実行しているためです。残りの 2 つの入力は関数 `direction` と `distance` の括弧内に插入されます。
3. 方向の指定は簡単です。`Vector.ZAxis()` を使用して垂直方向に移動します。
4. 参照点と各グリッド間の距離は計算する必要があります。これは同様に、参照点に対するアクションとして実行します：`refPt.DistanceTo(gridPts)`。
5. コードの最終行 `transPts = gridPts.Translate(Vector.ZAxis(), refPt.DistanceTo(gridPts));` によって変換された点が返されます。



- これで適切なデータ構造を持つ点のグリッドを使用してNURBSサーフェスを作成できます。サーフェスは `srf = NurbsSurface.ByControlPoints(transPts);` を使用して構築します。

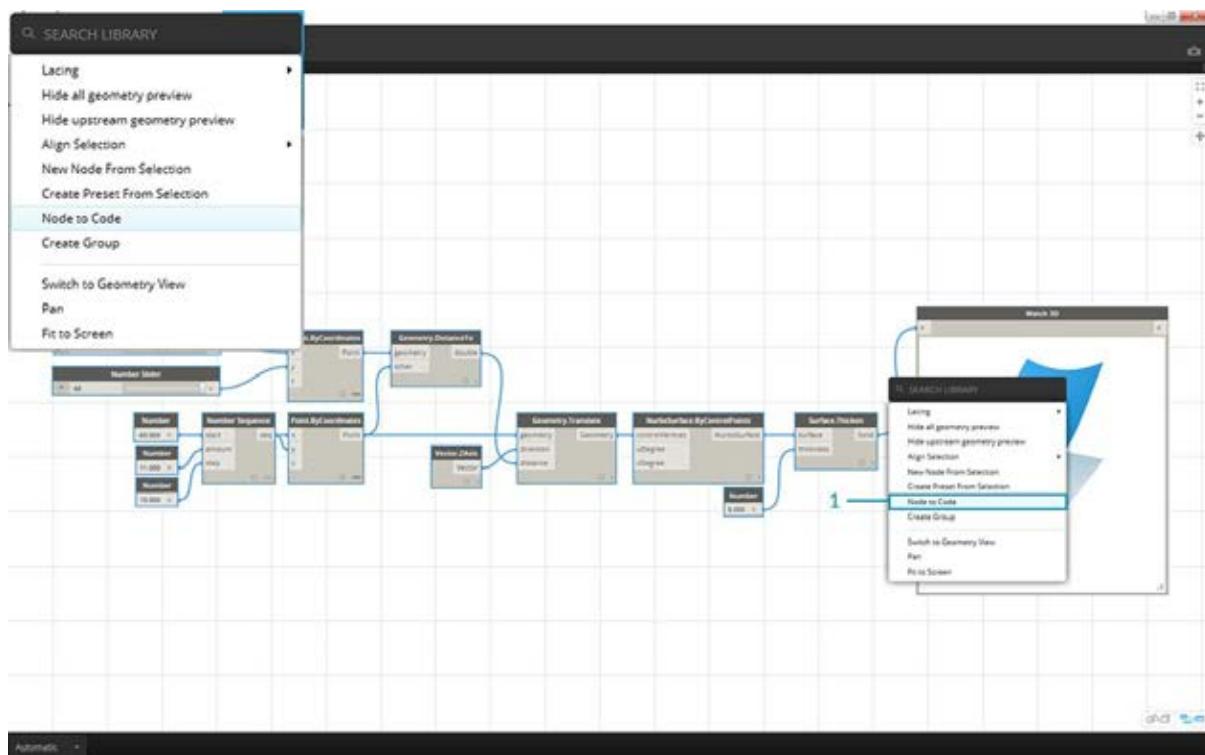


- 最後に、サーフェスに深さを追加するため、`solid = srf.Thicken(5);` を使用してソリッドを構築します。ここではコードで 5 単位を指定して厚みを付けますが、これを変数として(thickness などの名前を付けて)宣言し、スライダを使用して値をコントロールすることも可能です。

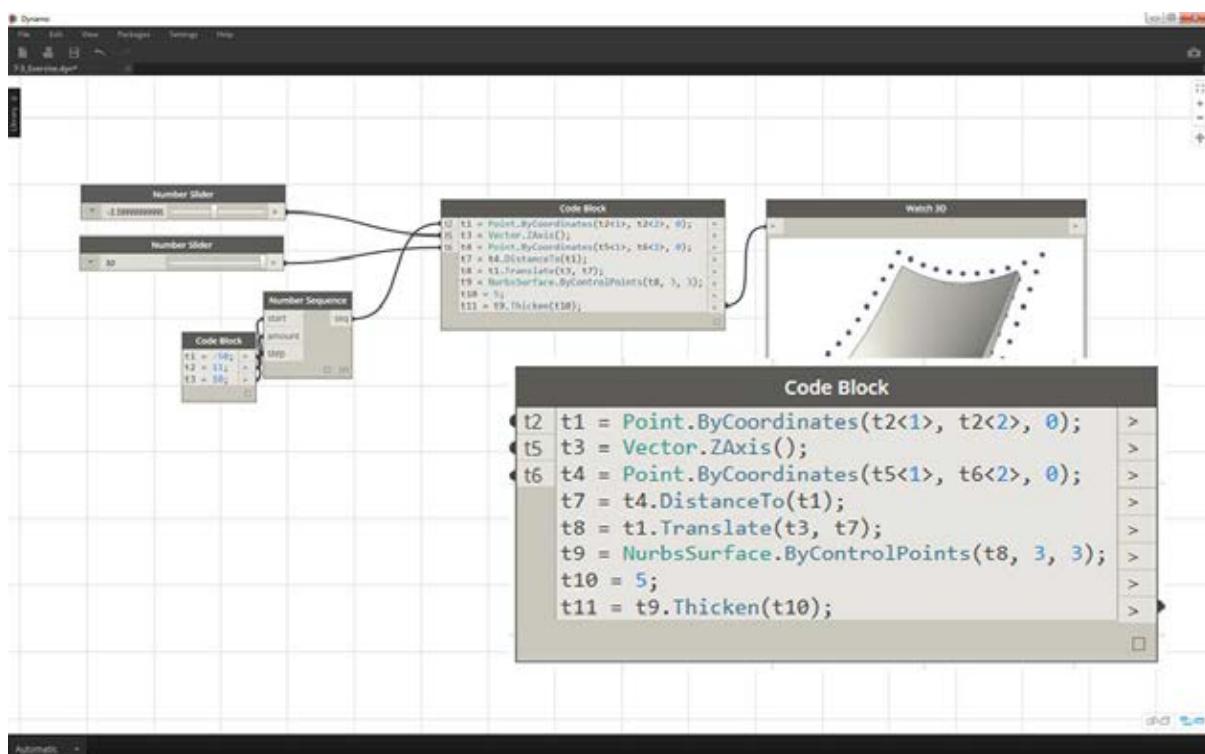
[ノードをコード化]を使用してグラフを単純化する

[ノードをコード化]機能を使用すると、先ほど終了した演習全体を、ボタンをクリックするだけで自動化できます。このツールはカスタ

ム設定や再利用可能なコード ブロックの作成に優れているだけでなく、Dynamo でスクリプトを作成する方法を学習するのにも役立ちます。



- まず、この演習の手順 1 で使用したビジュアル スクリプトを操作してみます。すべてのノードを選択し、キャンバスを右クリックして、[ノードをコード化]を選択します。非常に簡単です。



Dynamo はビジュアル グラフ、レーシングなどの文字ベースのバージョンを自動化しています。これをビジュアル スクリプトでテストして、コード ブロックの性能をお試しください。

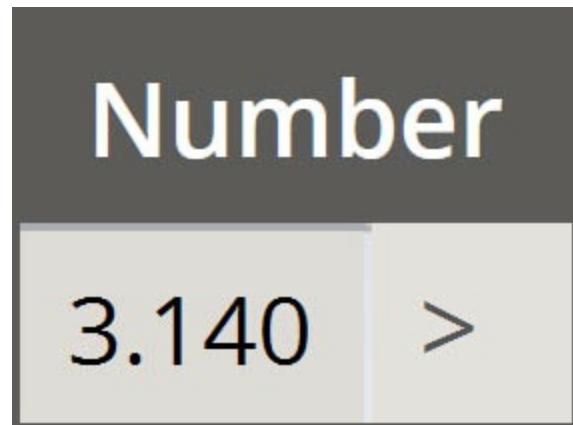
## 省略表記

### 省略表記

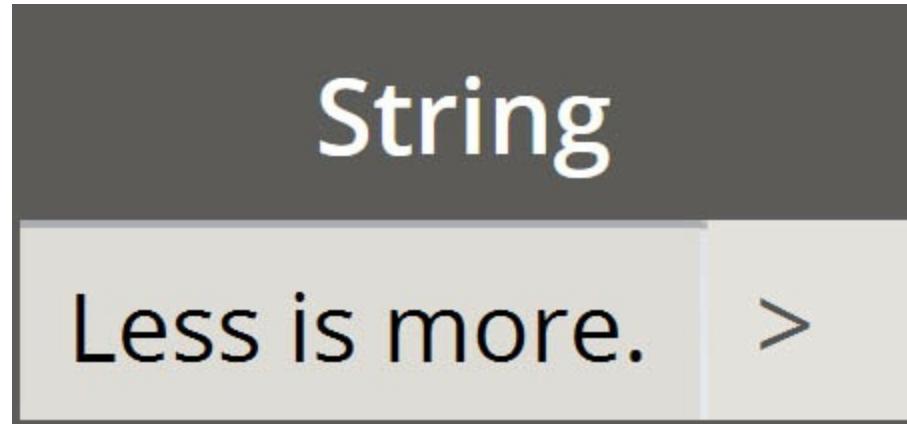
コード ブロックにはデータ管理を大幅に容易にする基本的な省略表記方法がいくつかあります。ここでは基本の概要を示し、この省略表記をデータの作成とクエリーの両方に使用する方法を説明します。

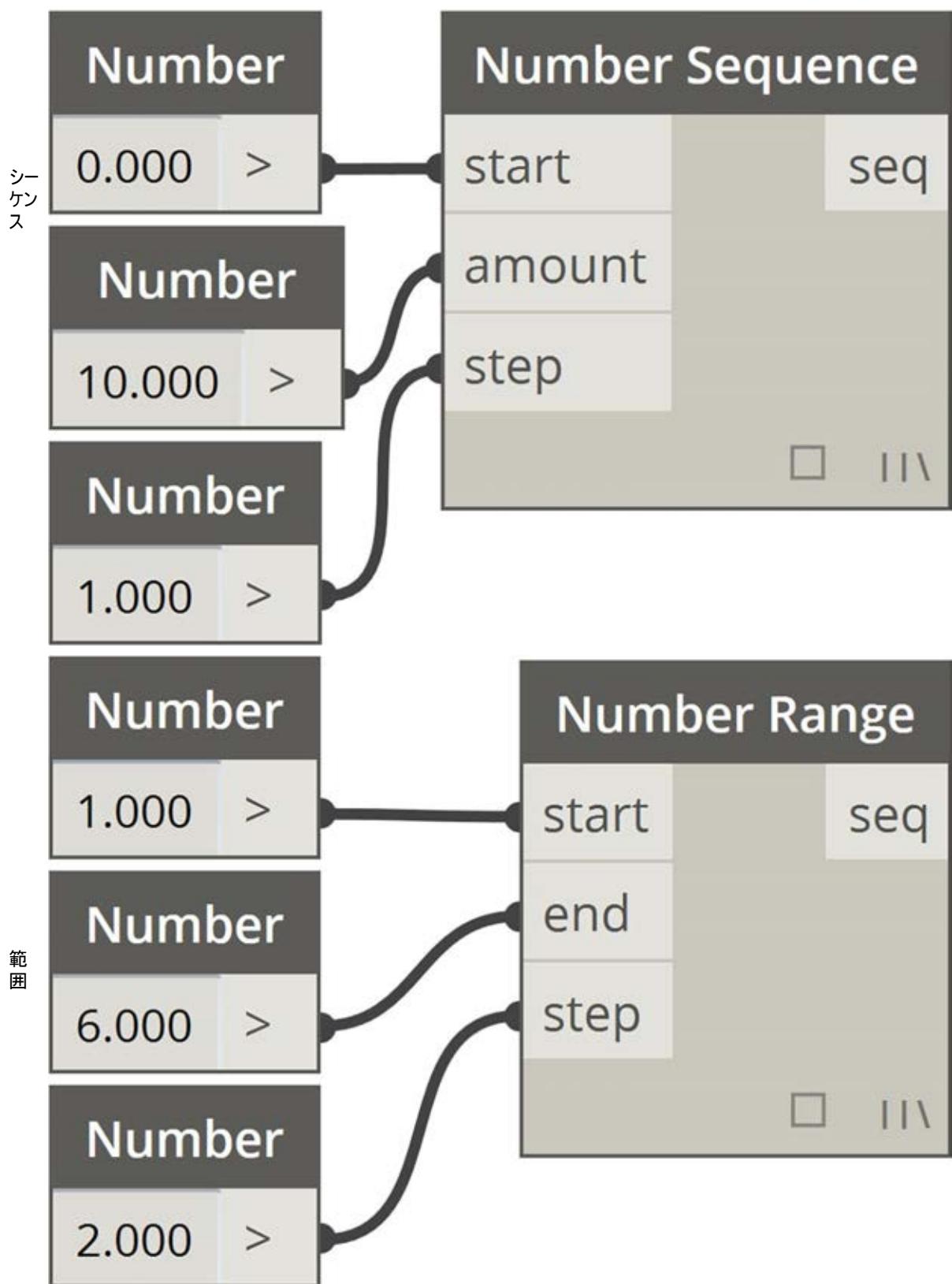
データ  
タタ  
イプ

標準 Dynamo

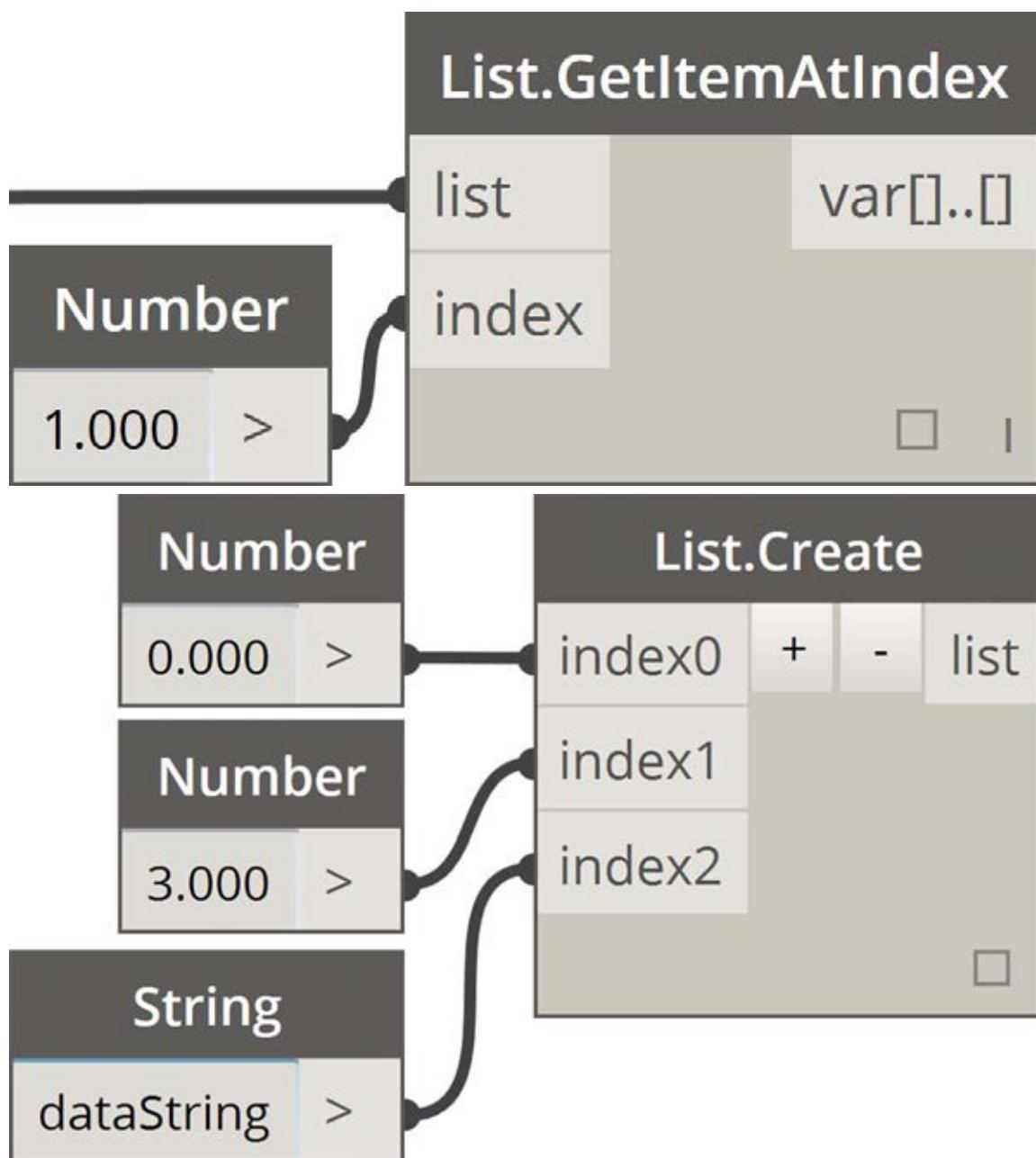


文字  
列





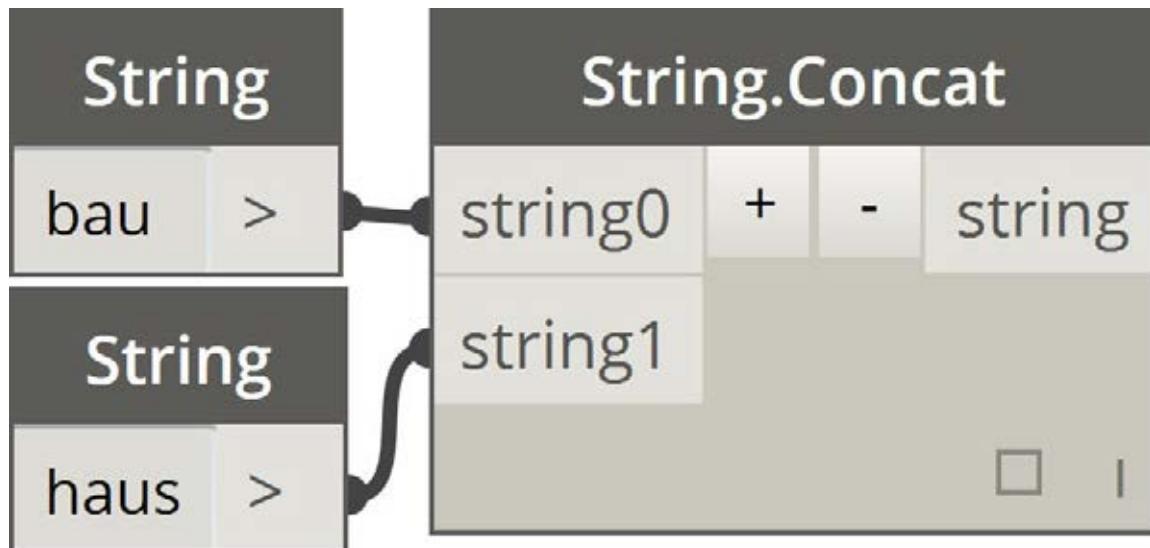
イン  
デッ  
クス  
での  
項  
目  
の  
取  
得



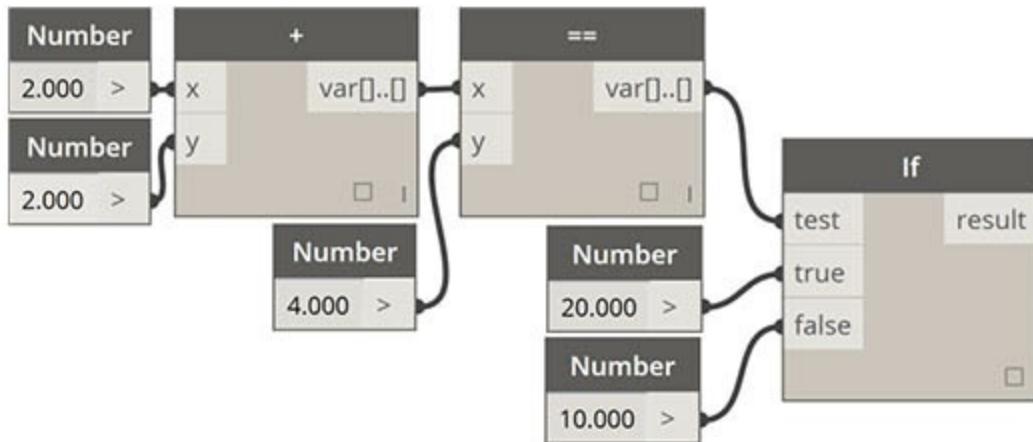
リス  
トの  
作  
成

文  
字

列の連結



条件ステートメント



## その他の構文

ノード	コード ブロックの同等表記	注意
すべての演算子(+、&&、>=、Notなど)	+、&&、>=、!など	「Not」は「!」になりますが、「Factorial」(階乗)と区別するためノードは「Not」と呼ばれます
ブールの True	true;	小文字を使用します
ブールの False	false;	小文字を使用します

## 範囲

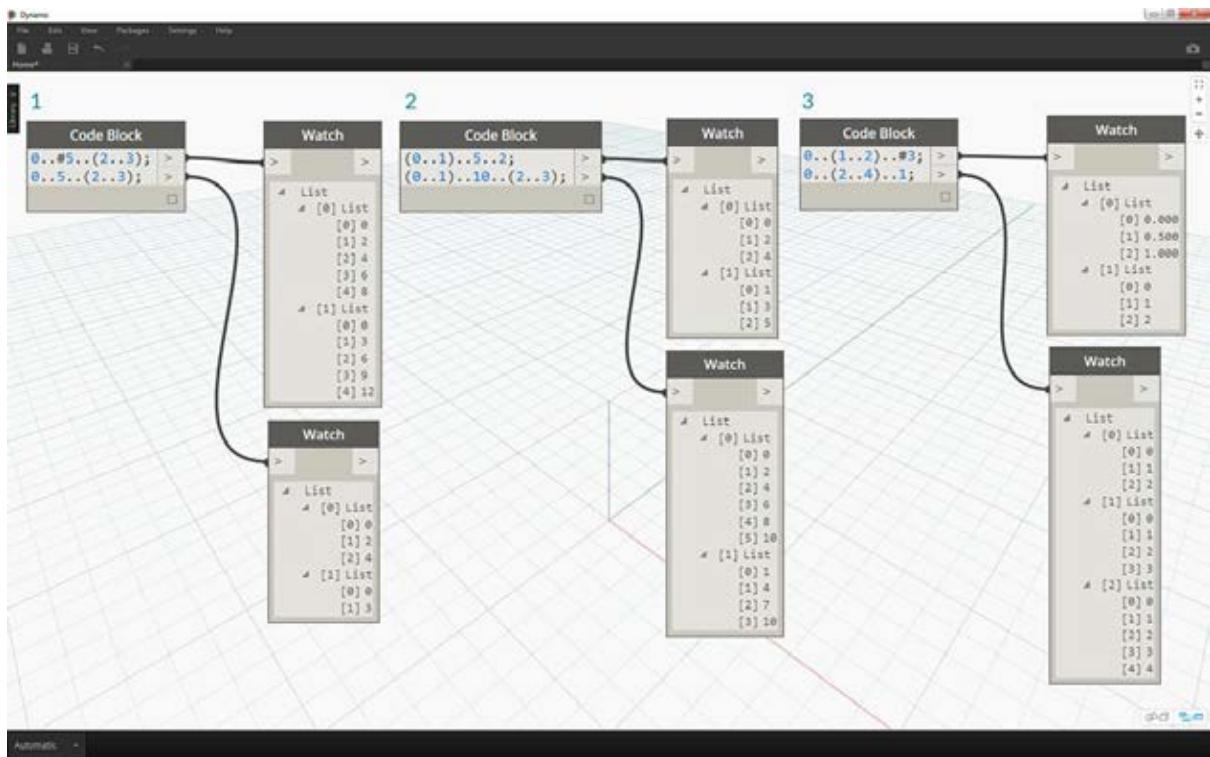
基本的な省略表記を組み合わせることで、範囲とシーケンスを設定することができます。下記の画像を「..」構文のガイドとして参照し、コード ブロックを使用して数値データのリストを設定してみましょう。この表記に慣れると、数値データを効率的に作成できるようになります。



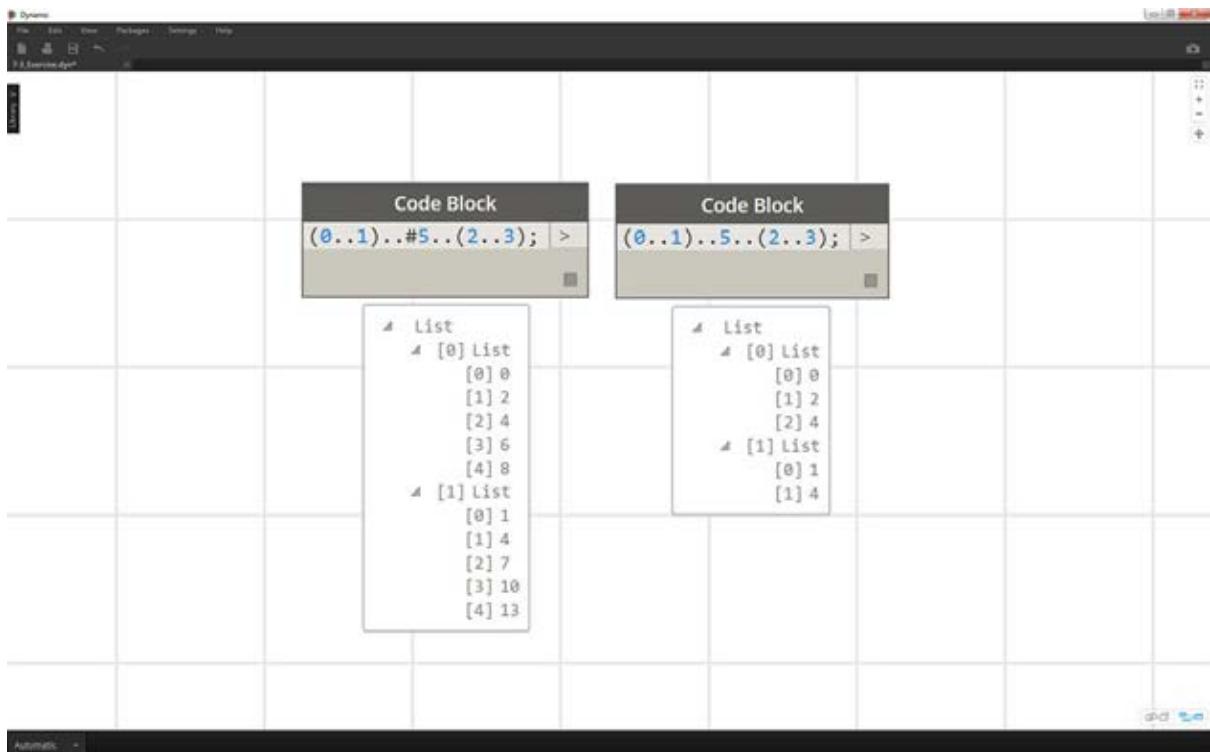
- この例では、数値範囲を `beginning..end..step-size;` の基本的なコード ブロック構文で置き換えて設定します。数値で表すと、`0.. 10.. 1;` になります。
- 構文 `0..10..1;` は `0..10;` と同じ意味になります。step-size の 1 は省略表記の既定値です。つまり、`0..10;` は間隔の大きさが 1 であるシーケンス 0 から 10 を表しています。
- 数値のシーケンスの例も同様です。ただし、「#」を使用して、15までの値を含むリストではなく、15 個の値を含むリストを指定しています。この場合、`beginning..#ofSteps..step-size:` を設定しています。シーケンスの実際の構文は `0..#15..2` になります。
- 今度は前の手順の「#」を構文の step-size 部分に配置してみましょう。これで、数値範囲は `beginning` から `end` に設定され、これらの 2 つの間の値が step-size 表記に指定された値で均等に分割されます:  
`beginning..end..#ofSteps.`

## 高度な範囲

高度な範囲を作成すると、リストのリストを簡単な方法で使用できます。次の例では、メイン範囲の表記から変数を分離して、このリストに別の範囲を作成します。



1. ネストされた範囲を作成して、「#」が指定されている表記と指定されていない表記とを比較してみましょう。ロジックは基本的な範囲と同じですが、多少複雑になります。
2. サブ範囲はメイン範囲内の任意の場所に設定できます。また、2つのサブ範囲を設定することもできます。
3. 範囲内の「end」値をコントロールすることにより、長さが異なる範囲を追加で作成できます。

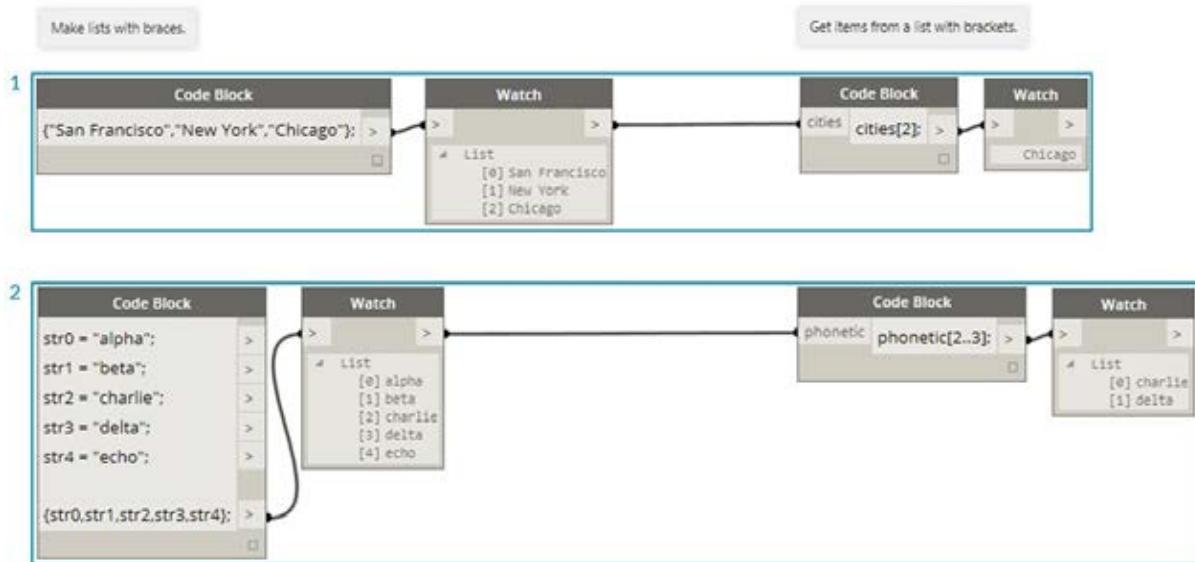


ロジックの演習として上記の2つの省略表記を比較し、サブ範囲と「#」表記が出力をどのようにコントロールしているかを読み解いてください。

### リストを作成してリストから項目を取得する

リストは省略表記を使用して作成できる他、すばやく作成することも可能です。これらのリストには幅広い要素タイプを含めることができます。

でき、クエリーを実行することも可能です(リストはリスト自体がオブジェクトです)。簡単に言うと、コード ブロックでプレース(中括弧)を使用してリストを作成し、ブラケット(角括弧)を使用してリスト内の項目のクエリーを実行します。



1. 文字列を使用してリストをすばやく作成し、項目のインデックスを使用してクエリーを実行します。
2. 変数を使用してリストを作成し、範囲の省略表記を使用してクエリーを実行します。

ネストされたリストを管理するプロセスは同様です。リストの順番に配慮し、複数の角括弧のセットを使用します。

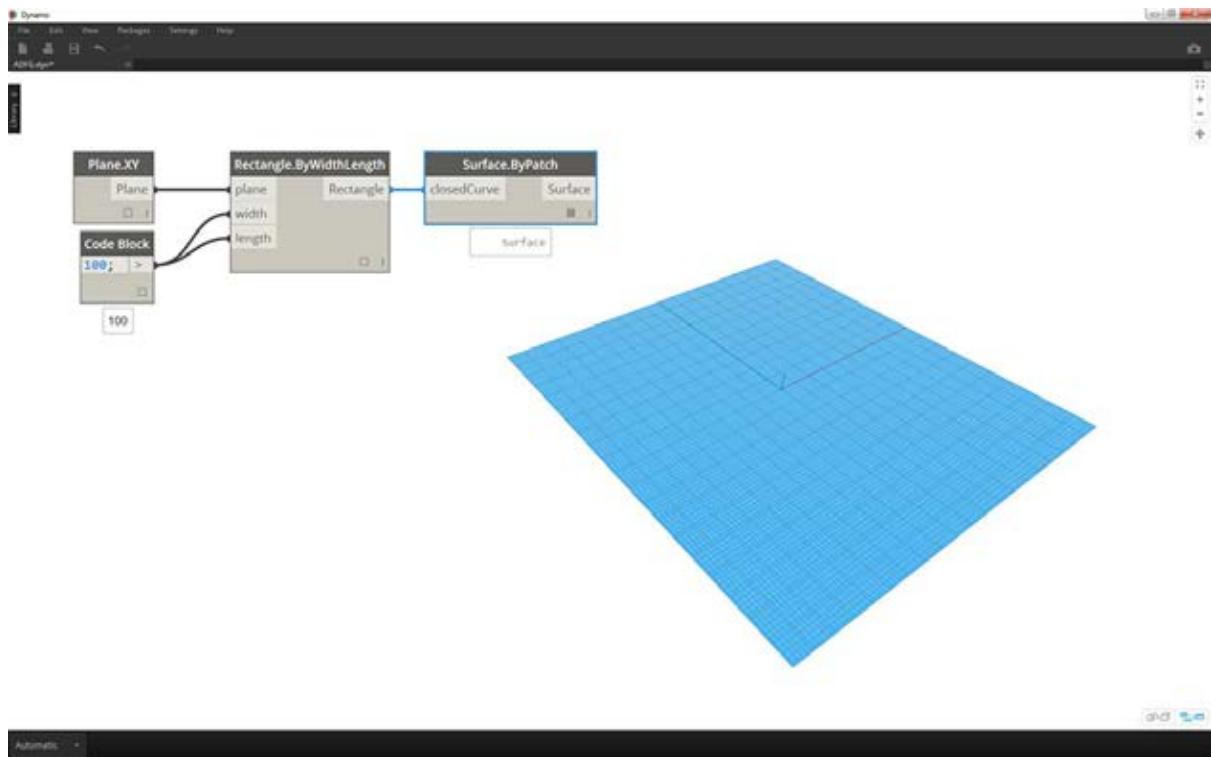


1. リストのリストを設定します。
2. 1 つの角括弧の表記を使用してクエリーを実行します。
3. 2 つの角括弧の表記を使用して項目のクエリーを実行します。

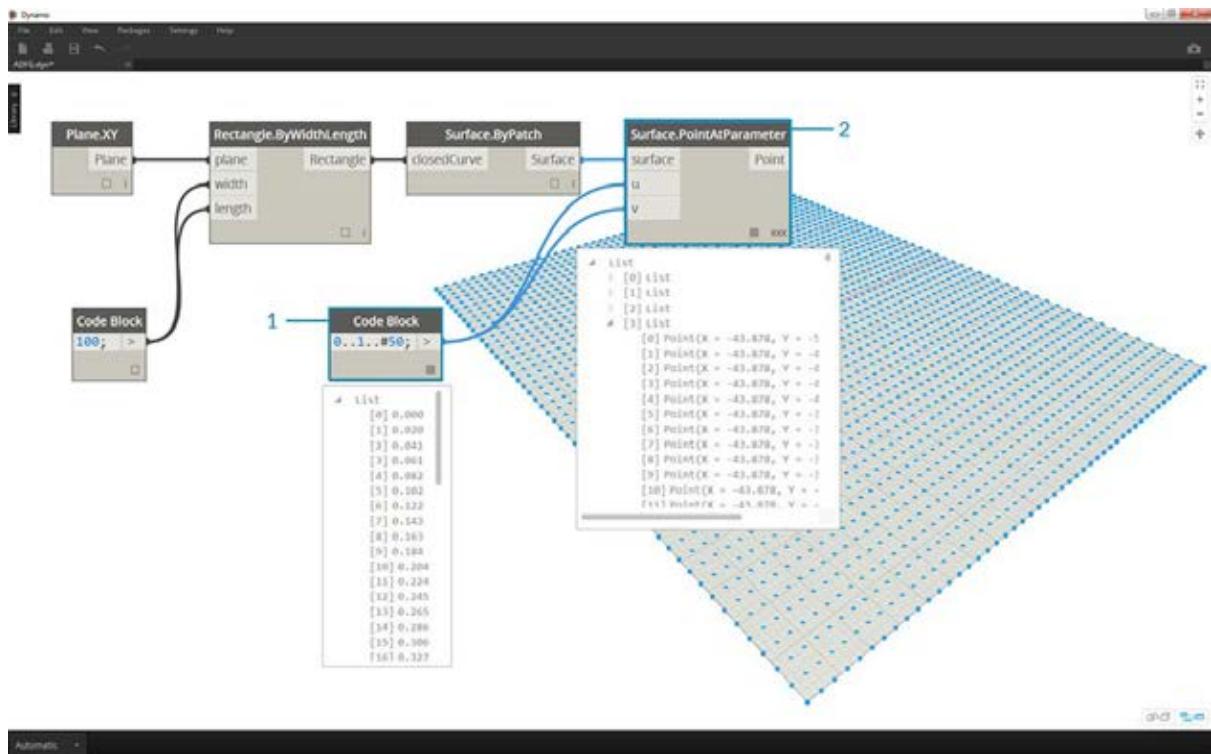
## 演習

この演習に付属しているサンプル ファイルをダウンロードしてください(右クリックして[名前を付けてリンク先を保存]を選択)。すべてのサンプル ファイルの一覧については、付録を参照してください。[Obsolete-Nodes\\_Sine-Surface.dyn](#)

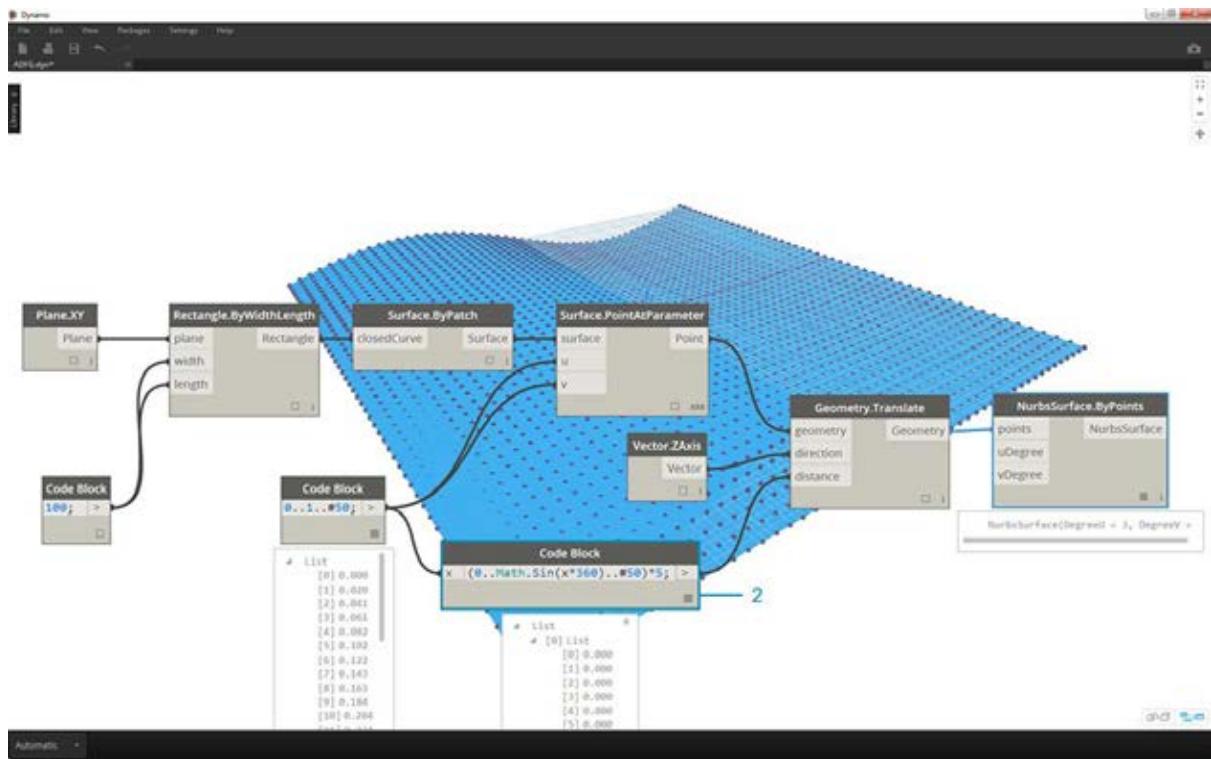
この演習では、新しい省略表記のスキルを使用して範囲と式を設定し、一風変わった卵型のサーフェスを作成します。この演習では、コード ブロックと既存の Dynamo ノードを並行して使用する方法を学習します。Dynamo ノードを視覚的に配置して設定を確認しながら、コード ブロックを使用して大きなデータを処理します。



まず上記のノードを接続してサーフェスを作成します。数値ノードを使用して幅と長さを設定する代わりに、キャンバスをダブルクリックして、コード ブロックに `100;` と入力します。

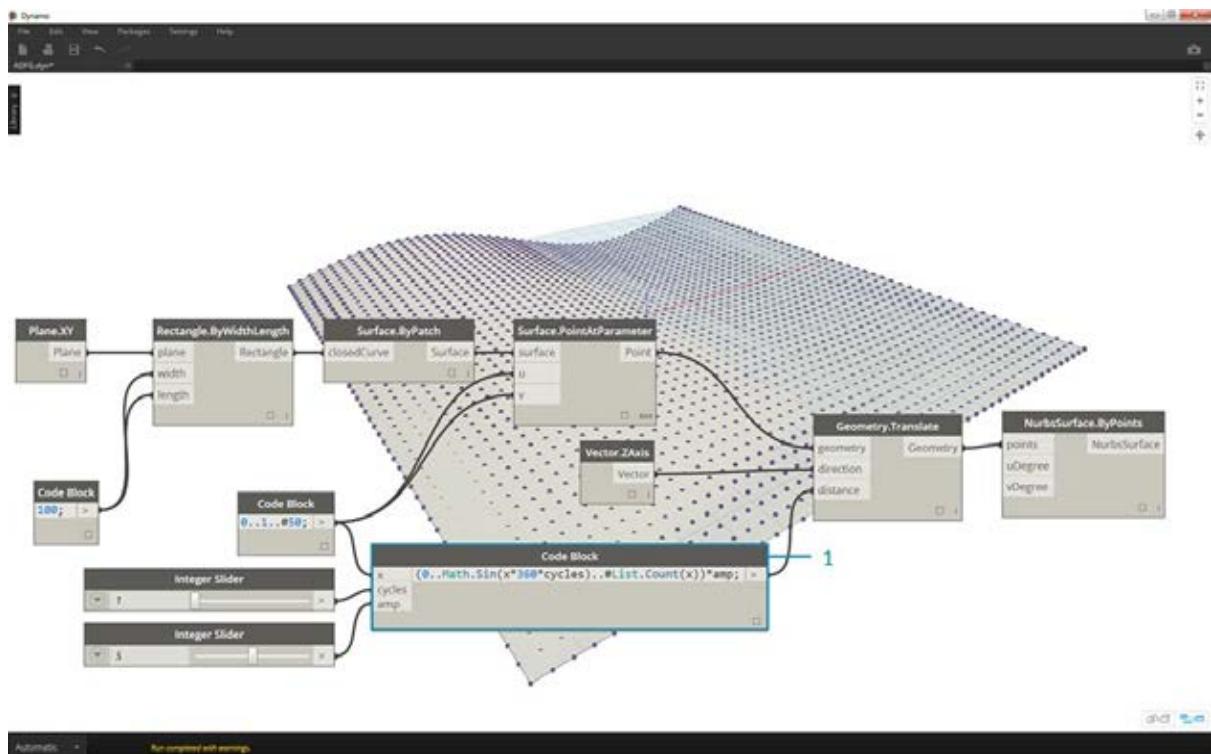


1. Code Block に `0..1..#50` と入力して、0 から 1 の範囲を 50 個に分割するように設定します。
2. 範囲を `Surface.PointAtParameter` ノードに接続します。これは 0 から 1 の範囲内にある  $u$  と  $v$  の値を取得してサーフェス全体に設定します。`Surface.PointAtParameter` ノードを右クリックし、レーシングを外積に変更します。

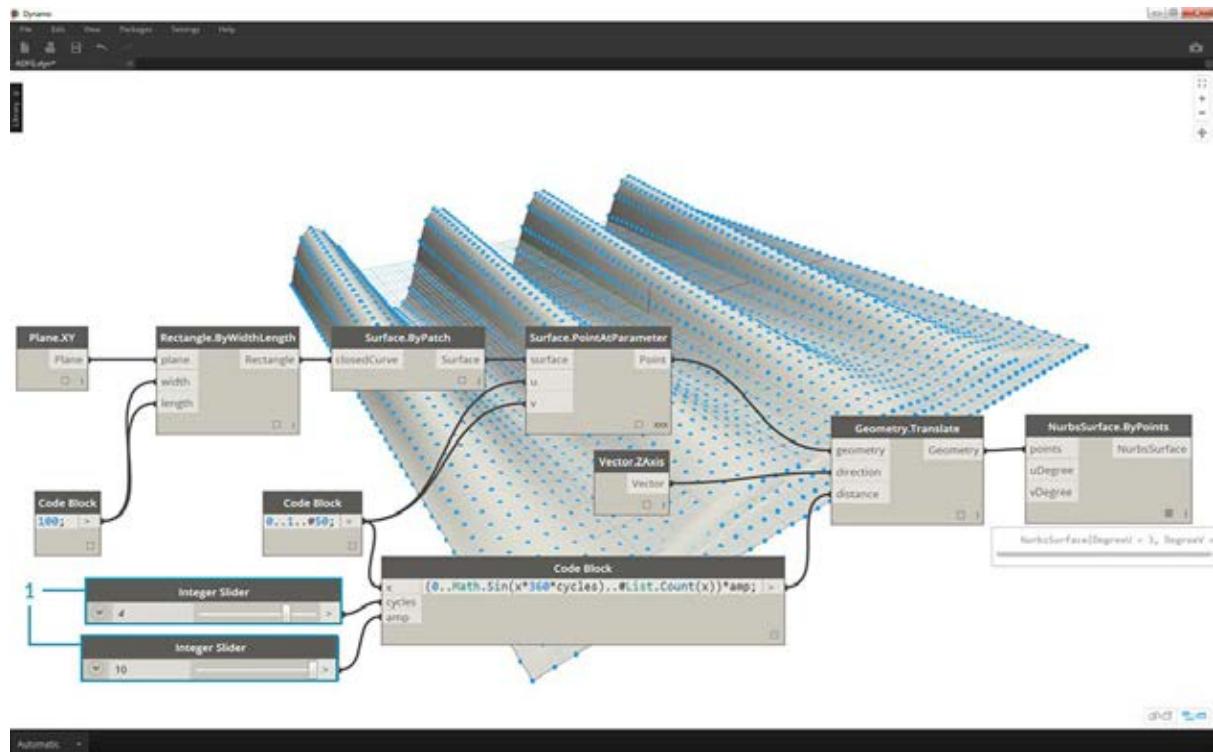


この手順では、最初の関数を適用して、グリッドを Z の正の向きに移動します。このグリッドは基盤となる関数に基づいて生成されるサーフェスをコントロールします。

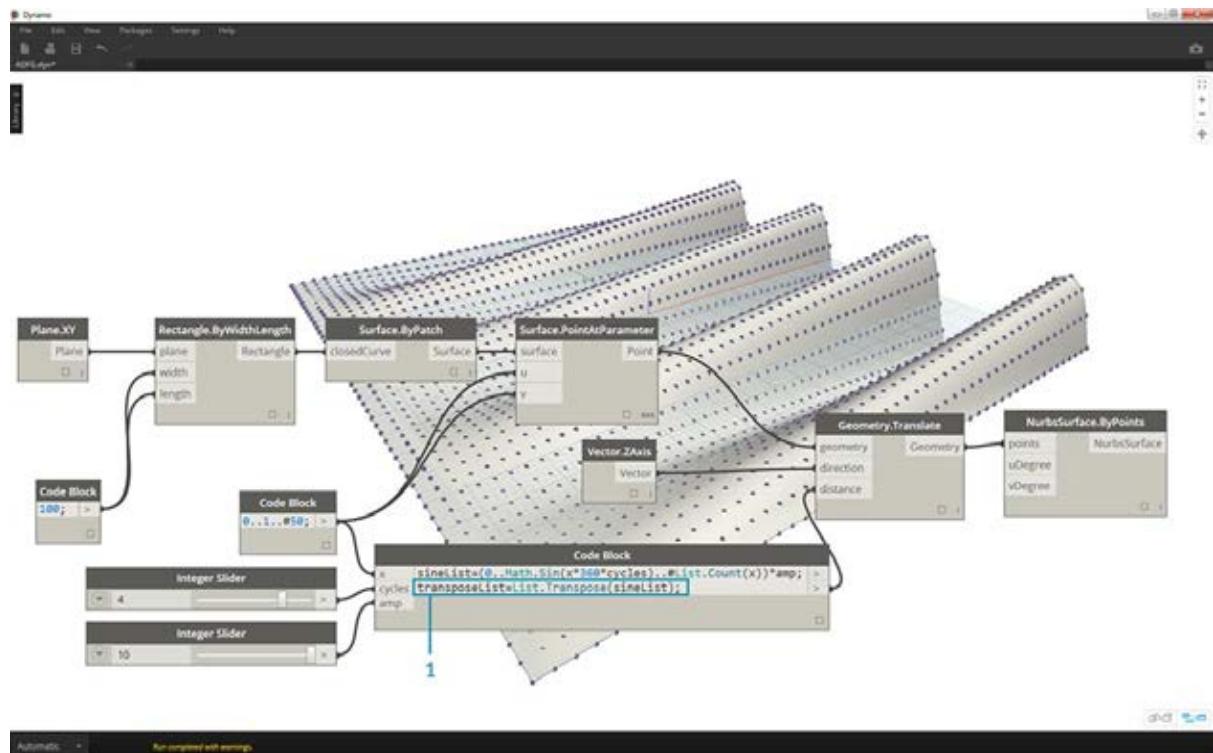
1. 上記の画像に示すように、ビジュアルノードをキャンバスに追加します。
2. 式ノードを使用する代わりに、Code Block を使用して  $(0..Math.Sin(x*360)..#50)*5;$  を指定します。この式を簡単に分割して説明するために、式内に範囲を設定します。この式は正弦関数です。正弦関数は Dynamo で角度(度)入力を受け取ります。このため、完全な正弦波を得るには、x 値(0 から 1 までの入力範囲)を 360 で乗算します。次に、各行のコントロールグリッドの点と同じ数だけ分割するため、#50 を指定して 50 個のサブディビジョンを設定します。最後に、Dynamo プレビューで効果を確認できるようにするために、累乗の指数に 5 を指定して変換の振幅を大きくします。



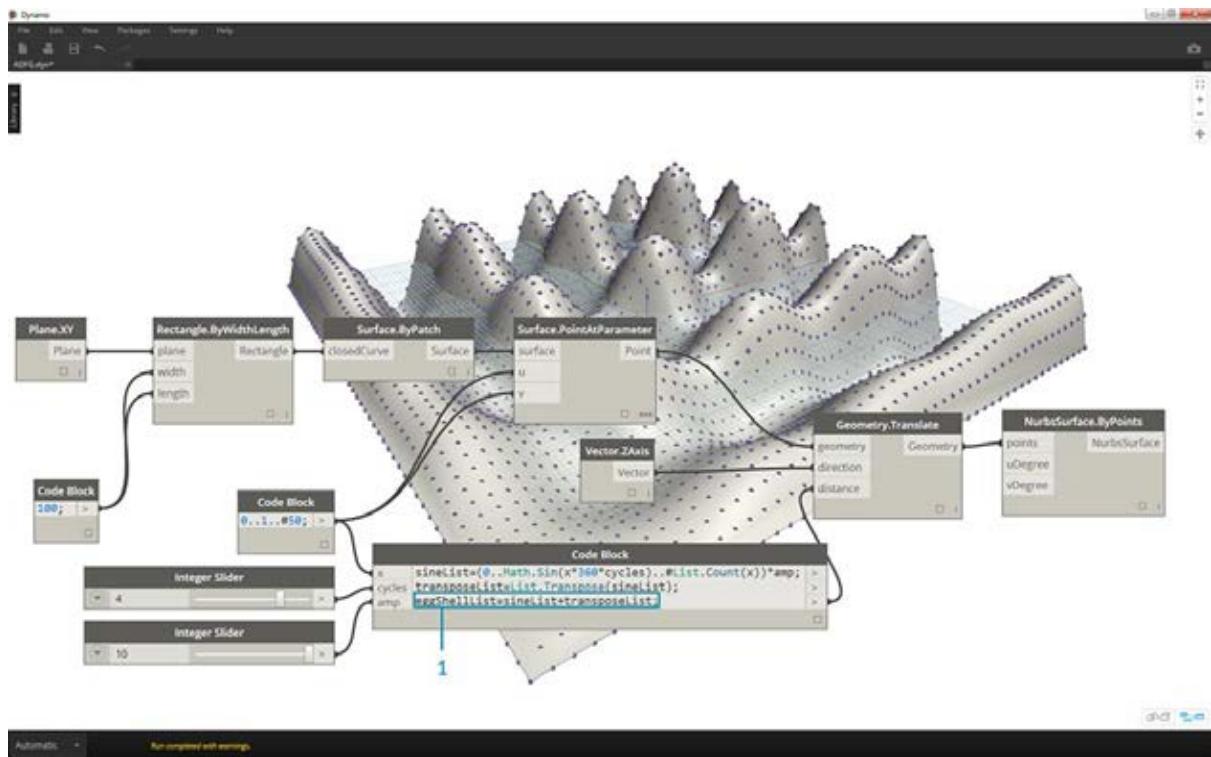
- 以前のコード ブロックは正常に動作しましたが、完全にパラメトリックではありませんでした。パラメータを動的にコントロールするため、前の手順で使用した行を  
`(0..Math.Sin(x*360*cycles)..#List.Count(x))*amp;` で置き換えます。こうすることで、これらの値を入力に基づいて設定できるようになります。



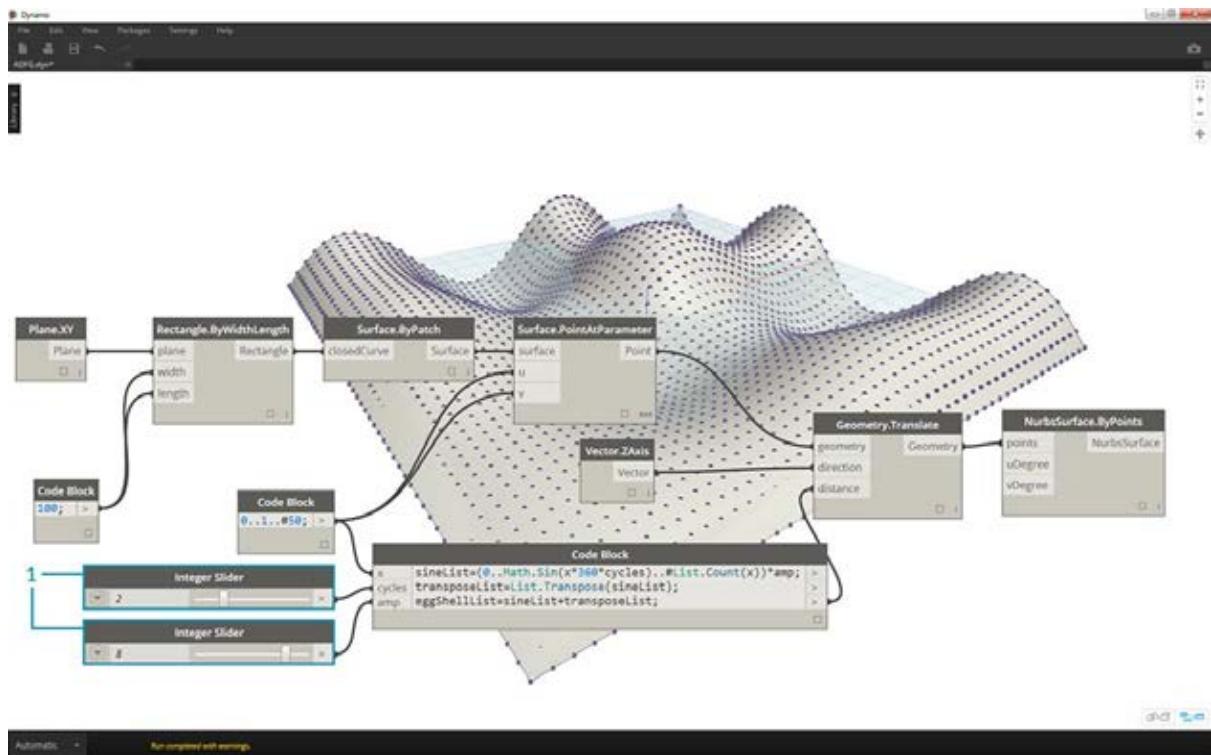
- スライダ(範囲 0 から 10)を変更して、どのような結果が生じるか確認します。



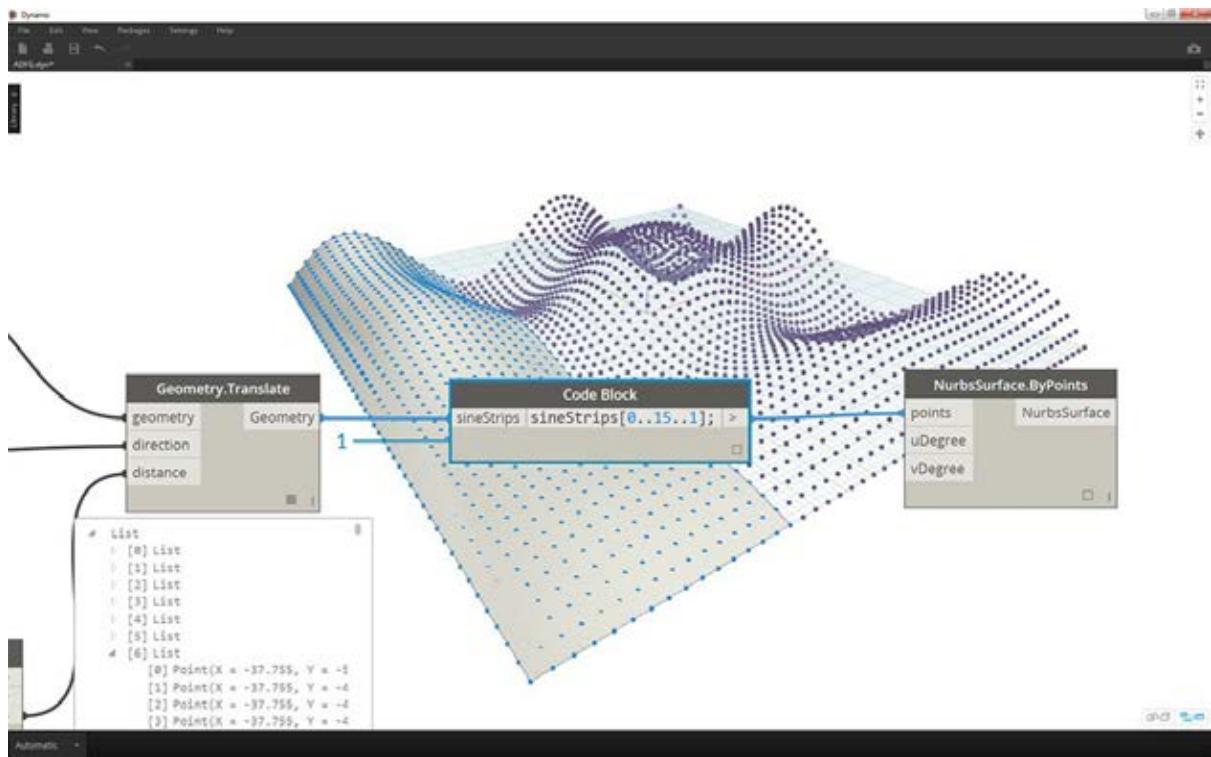
- 数値範囲を転置することにより、カーテン ウェーブの方向を反転します: `transposeList = List.Transpose(sineList);`



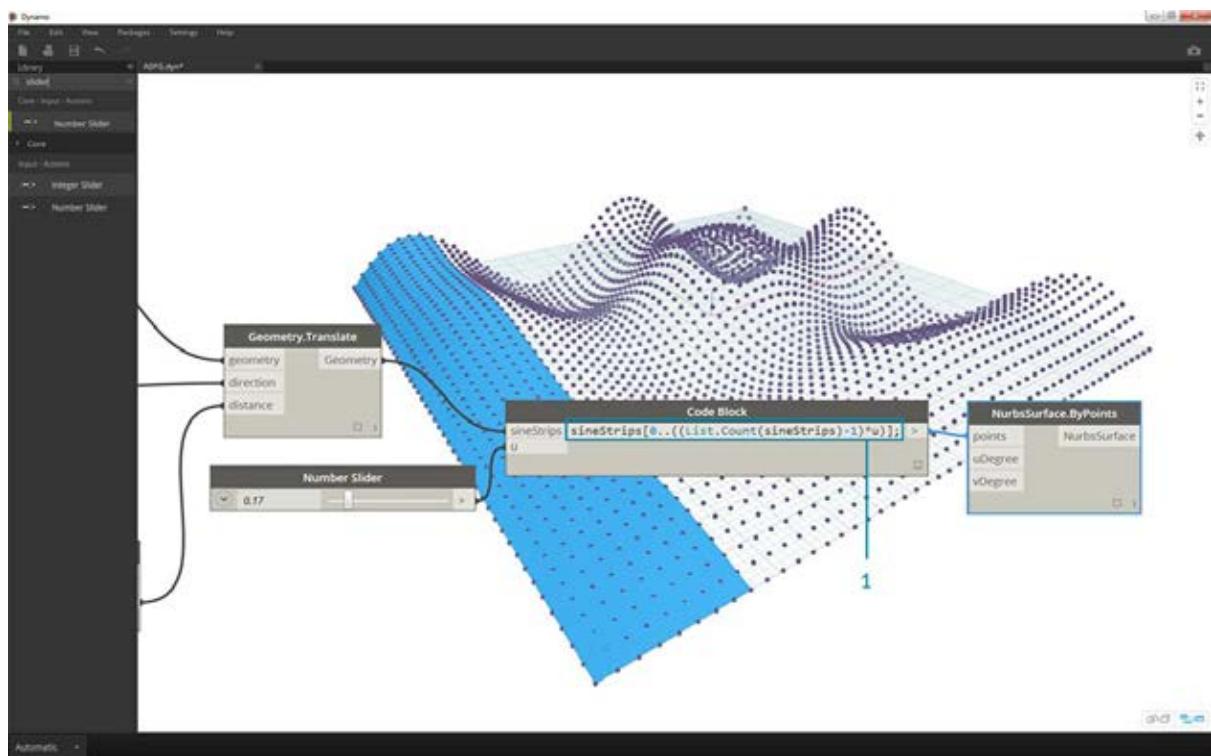
1. sineList と transposeList を追加すると、歪曲した卵型のサーフェスが生成されます: eggShellList = sineList+transposeList;。



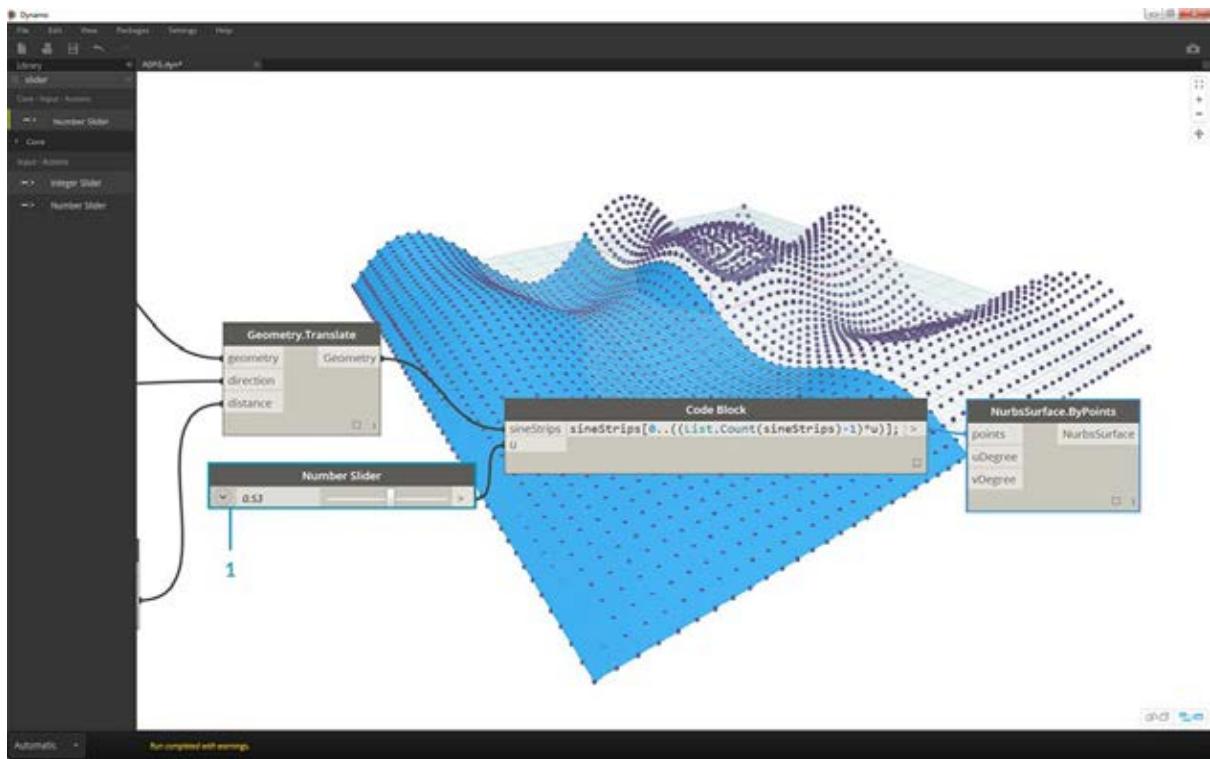
1. もう一度スライダを変更して、このアルゴリズムの生成結果をなだらかになるように調整しましょう。



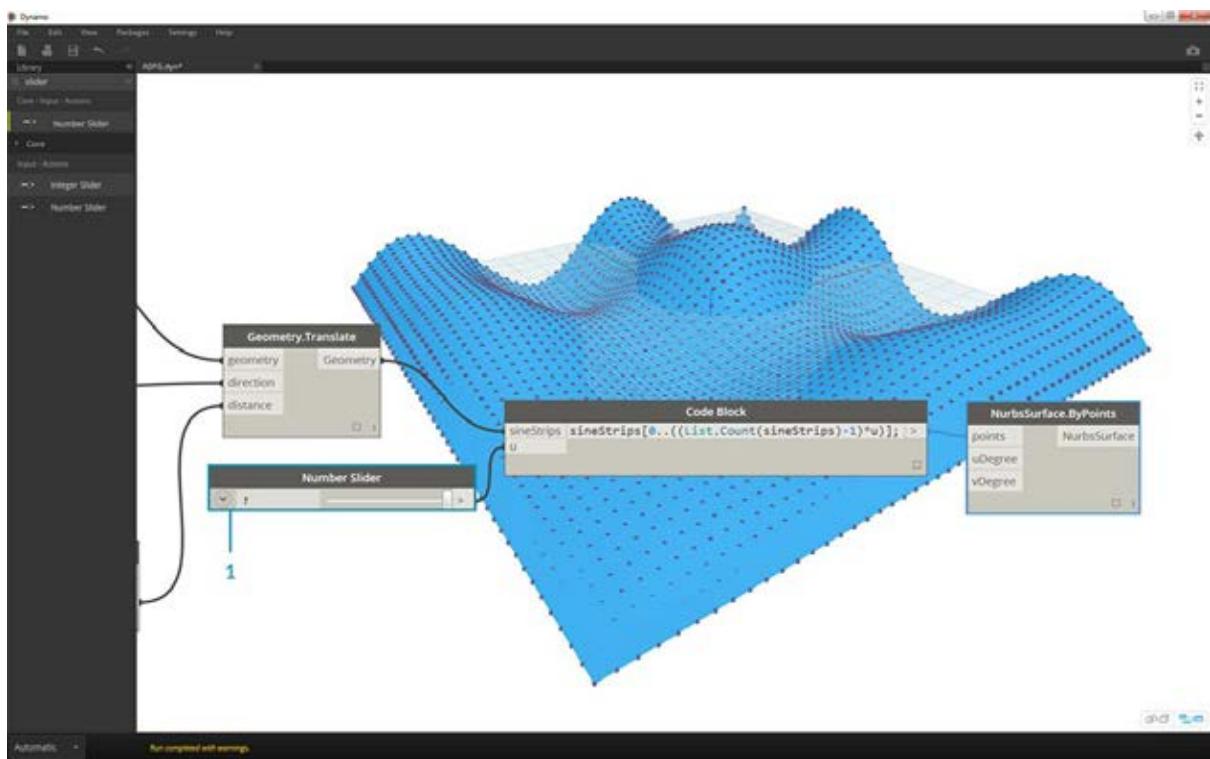
- 最後に、コードブロックを使用して、データの一部のクエリーを実行しましょう。特定の範囲の点を指定してサーフェスを再生成するには、Geometry.Translate ノードと NurbsSurface.ByPoints ノードの間に上記の Code Block ノードを追加します。sineStrips... 15 [0.. 1] ; が指定されています。これにより、50 行の最初の 16 行の点が選択されます。サーフェスを作成すると、点のグリッドの一部が分離されて生成されていることがわかります。



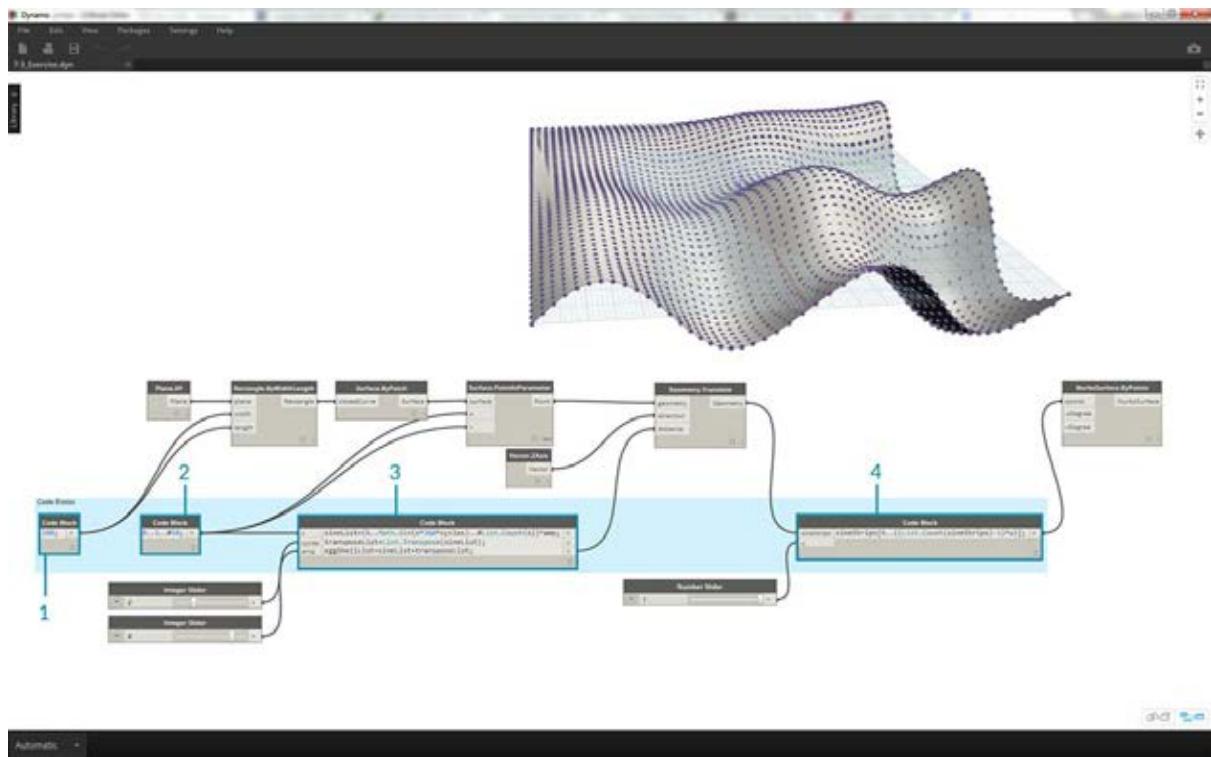
- 最後の手順では、このコードブロックをよりパラメトリックなものにするため、範囲 0 から 1 のスライダを使用してクエリーをコントロールします。これを行うには、コード行 `sineStrips[0..((List.Count(sineStrips)-1)*u)];` を使用します。わかりにくいかもしれません、このコード行により、リストの長さを乗数 0 から 1 の値を使用してすばやくスケールできます。



1. スライダに .53 の値を設定すると、グリッドの中央をわずかに超えるサーフェスが作成されます。



1. おわかりのように、スライダを 1 に設定すると、すべての点のグリッドを使用してサーフェスが作成されます。



生成されたビジュアル グラフを参照する際、コード ブロックをハイライト表示して Code Block ノードの各関数を確認できます。

1. 最初の Code Block ノードは Number ノードを置き換えます。
2. 2 番目の Code Block ノードは Number Range ノードを置き換えます。
3. 3 番目の Code Block ノードは Formula ノード(および List.Transpose、List.Count、Number Range の各ノード)を置き換えます。
4. 4 番目の Code Block ノードはリストのリストのクエリーを実行し、List.GetItemAtIndex ノードを置き換えます。

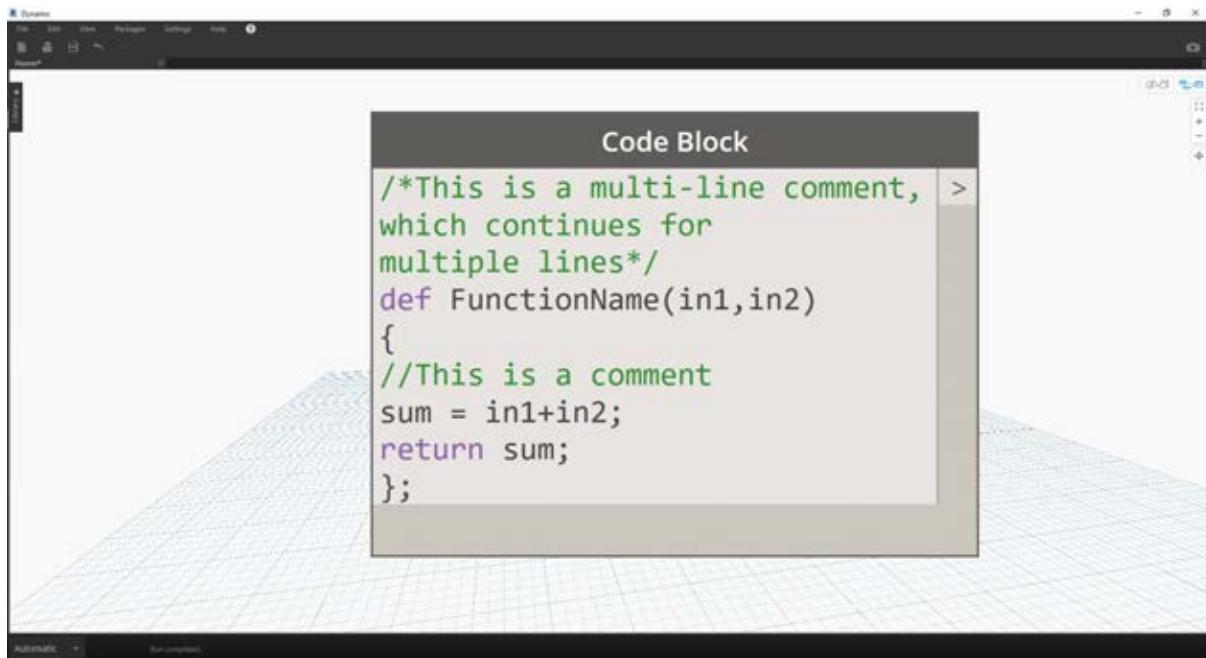
# コード ブロック関数

## コード ブロック関数

関数は Code Block ノード内で作成することができ、Dynamo 定義の任意の場所から呼び出すことができます。これにより、パラメトリック ファイル内に新しいコントロール レイヤーが追加されます。Code Block ノードは、テキストベースのカスタム ノードとしてみなすことができます。この場合、「親」コード ブロックに簡単にアクセスすることができます。親コード ブロックは、グラフ上の任意の場所に配置することができます。ワイヤは必要ありません。

### 親

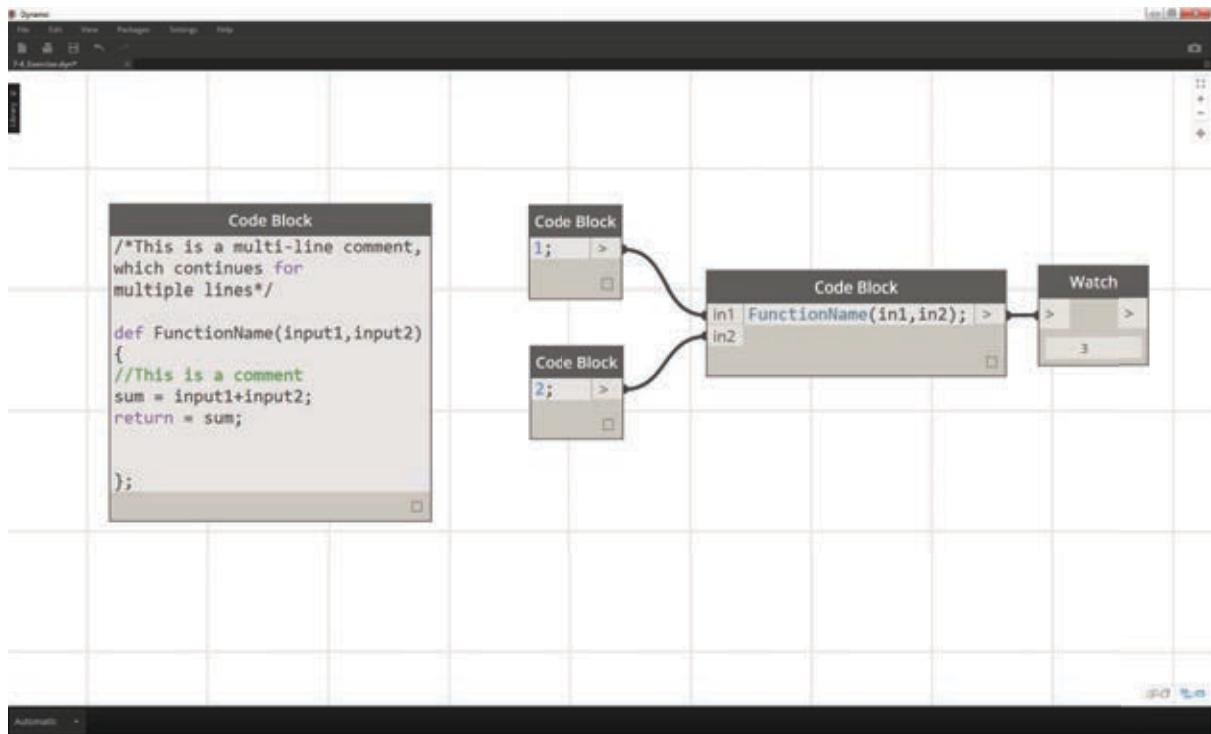
最初の行に、キーワードの「def」と関数名を続けて入力し、入力データの名前を括弧で囲んで記述します。関数の本文を定義する場合は、波括弧 {} を使用します。値を返す場合は、「return =」を指定します。関数を定義する Code Block ノードは他の Code Block ノードから呼び出されるため、入力ポートと出力ポートはありません。



```
/*This is a multi-line comment,
which continues for
multiple lines*/
def FunctionName(in1,in2)
{
//This is a comment
sum = in1+in2;
return sum;
};
```

## 子

関数を呼び出すには、同じファイル内で別の Code Block ノードを使用して、呼び出す関数の名前と、親 Code Block ノードで定義されているものと同じ数の引数を指定します。これは、ライブラリ内に用意されているノードと同様に動作します。

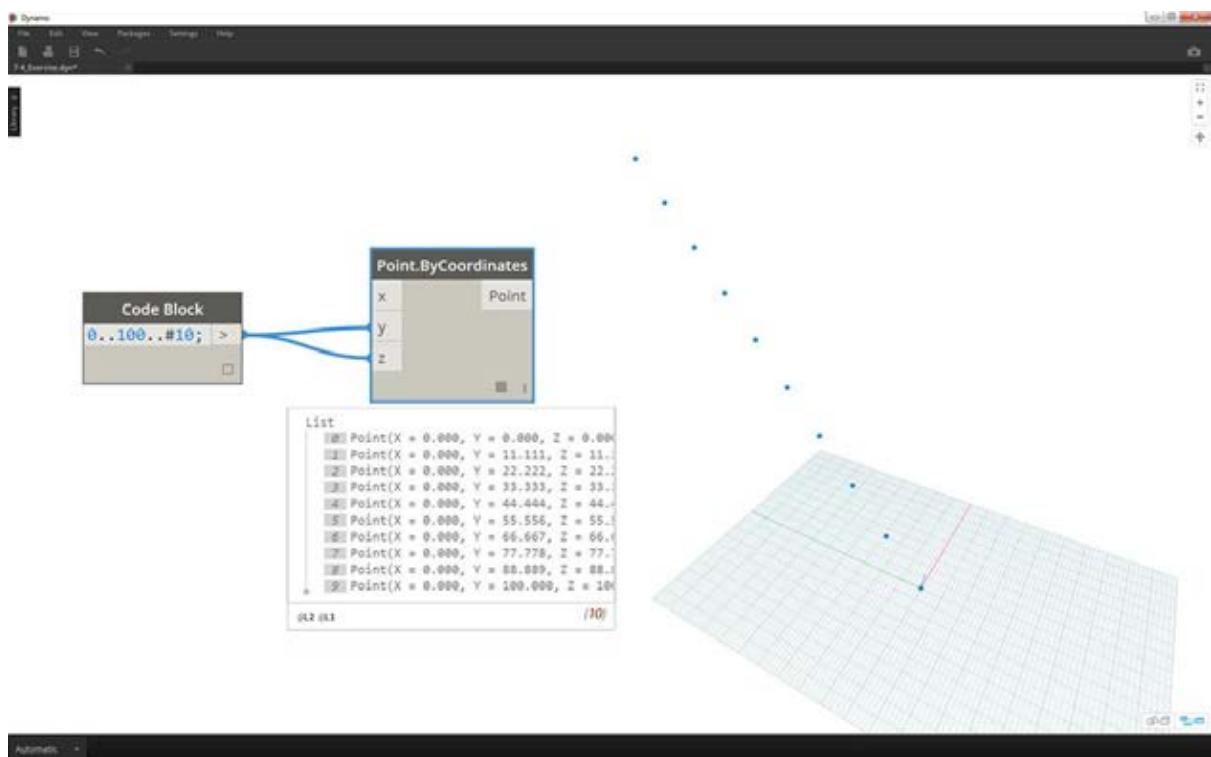


FunctionName(in1,in2);

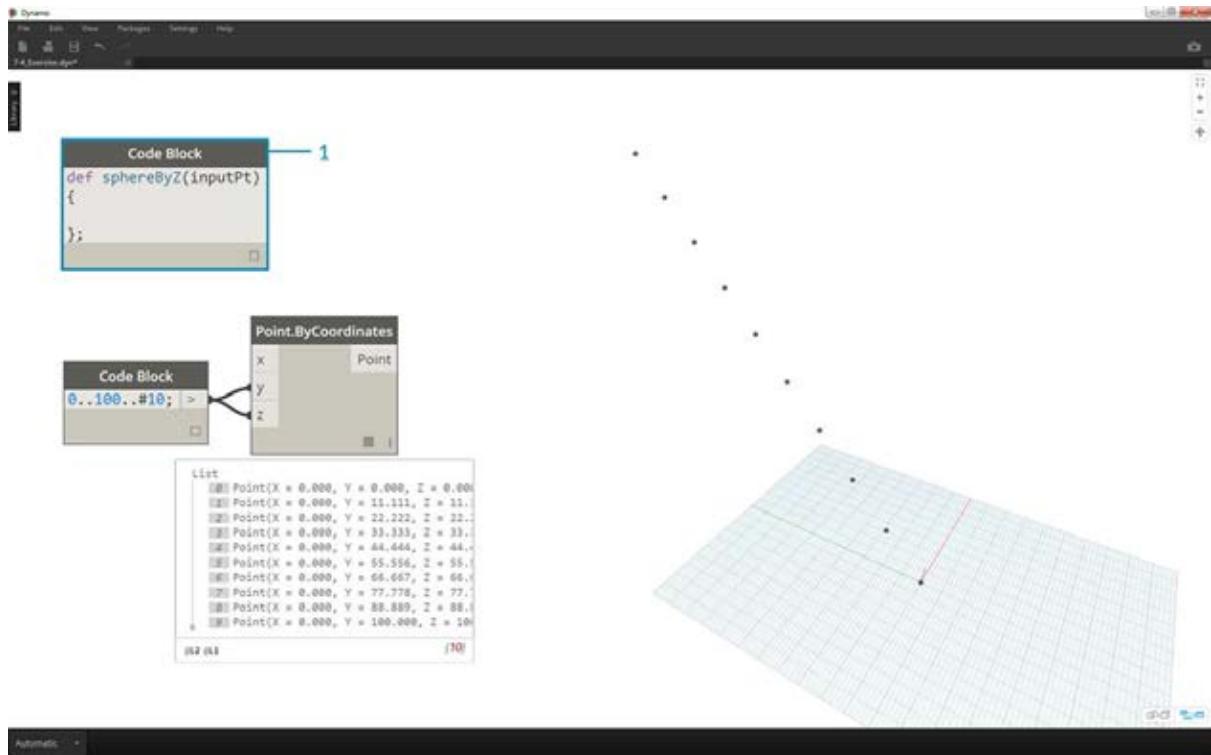
## 演習

この演習に付属しているサンプル ファイルをダウンロードしてください(右クリックして[名前を付けてリンク先を保存]を選択)。すべてのサンプルファイルの一覧については、付録を参照してください。[Functions\\_SphereByZ.dyn](#)

この演習では、点の入力リストから球体を生成する一般的な定義を作成します。これらの球体の半径は、各点の Z プロパティによってコントロールされます。



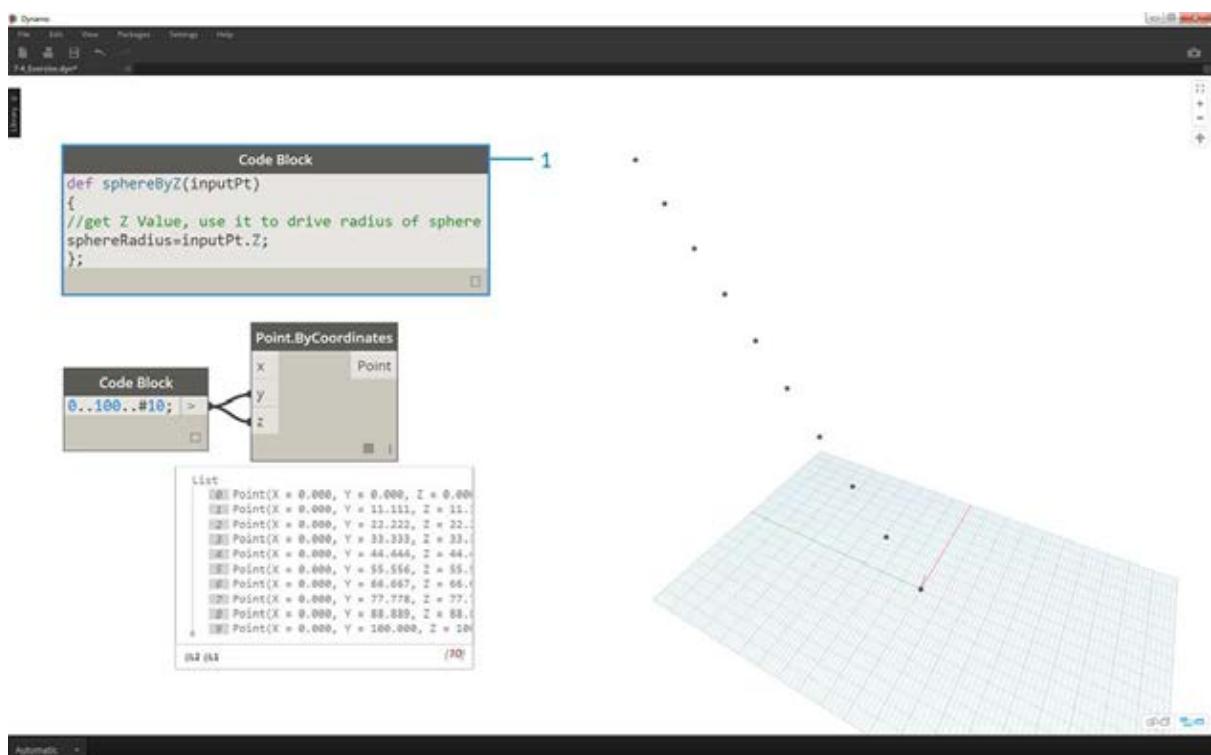
最初に、0 から 100 までの範囲内で 10 個の数値を作成しましょう。これらの数値を Point.ByCoordinates ノードに接続し、斜線を作成します。



1. *Code Block* ノードを作成し、コード行を使用して定義を開始します。

```
def sphereByZ(inputPt){  
};
```

*inputPt* は、関数をコントロールする点を表す名前です。この時点では、この関数はまだ機能しませんが、これ以降の手順でこの関数を設定していきます。

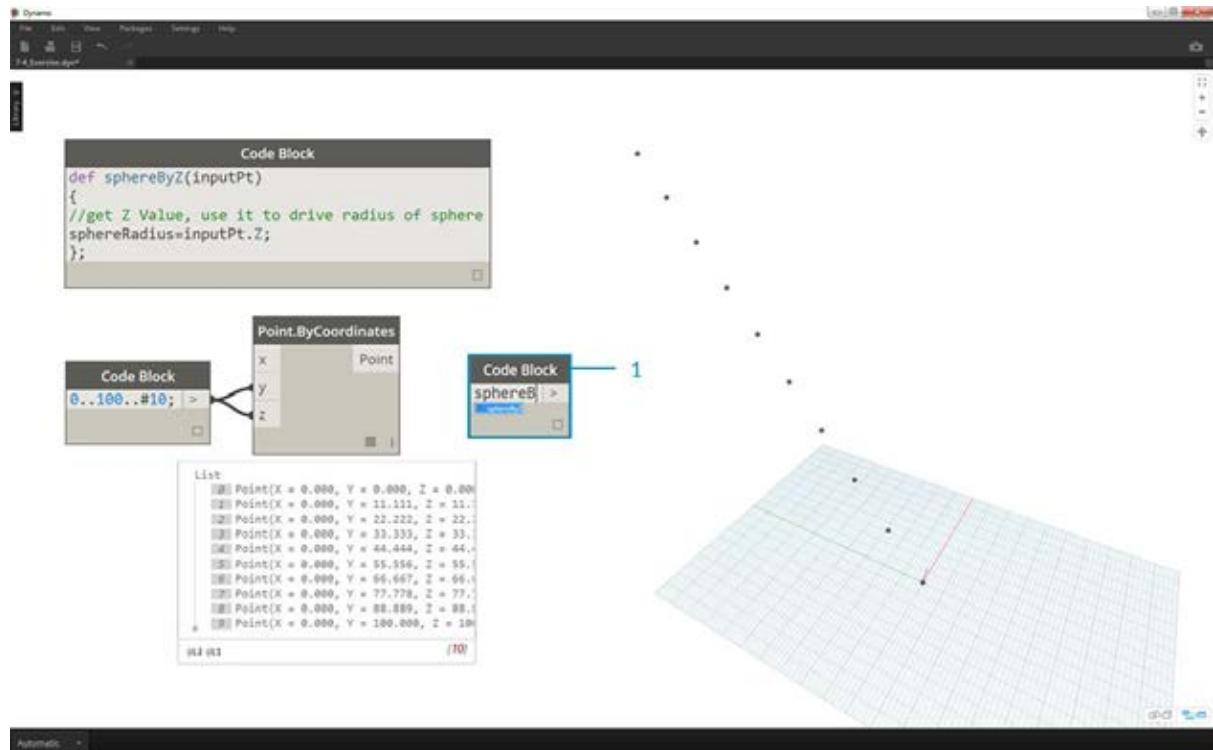


1. *Code Block* ノードを使用して、この関数にコメントと *sphereRadius* 変数を入力します。この変数により、各点の Z 位置のクエリーが実行されます。*inputPt.Z* はメソッドであるため、引数を指定するための括弧は必要ありません。これは既存の要素のプロパティのクエリーであるため、入力は必要ありません。

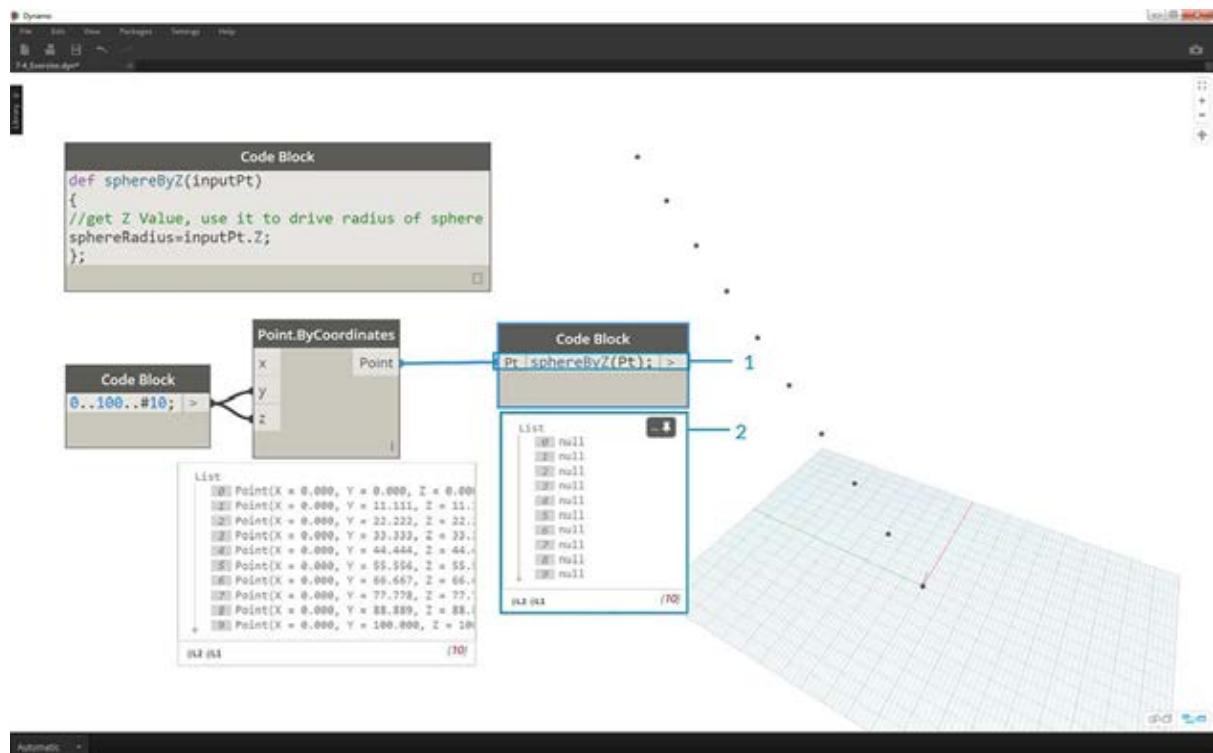
```

def sphereByZ(inputPt, radiusRatio)
{
    //get Z Value, use it to drive radius of sphere
    sphereRadius=inputPt.Z;
};

```



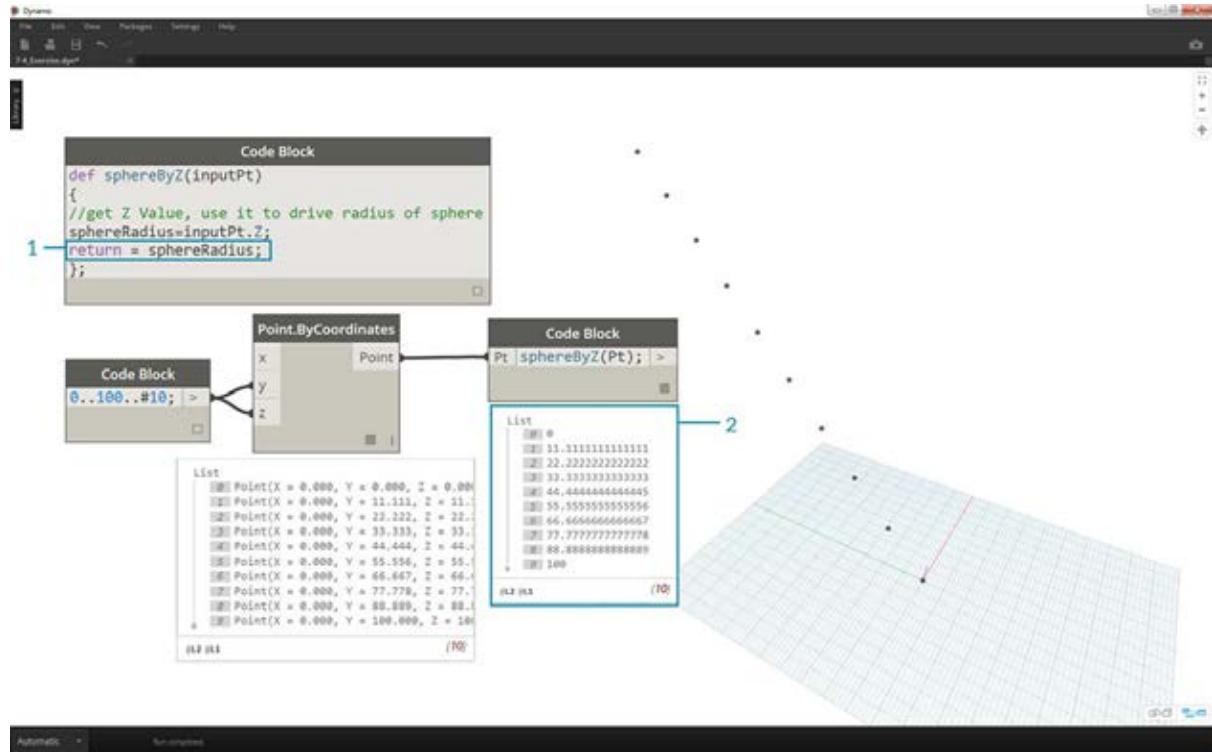
- ここで、別の Code Block ノードで作成した関数を呼び出してみましょう。キャンバスをダブルクリックして新しい Code Block ノードを作成し、sphereB と入力すると、既に定義されている sphereByZ 関数が候補として表示されます。これにより、前の手順で作成した関数が IntelliSense ライブリパリに追加されていることがわかります。



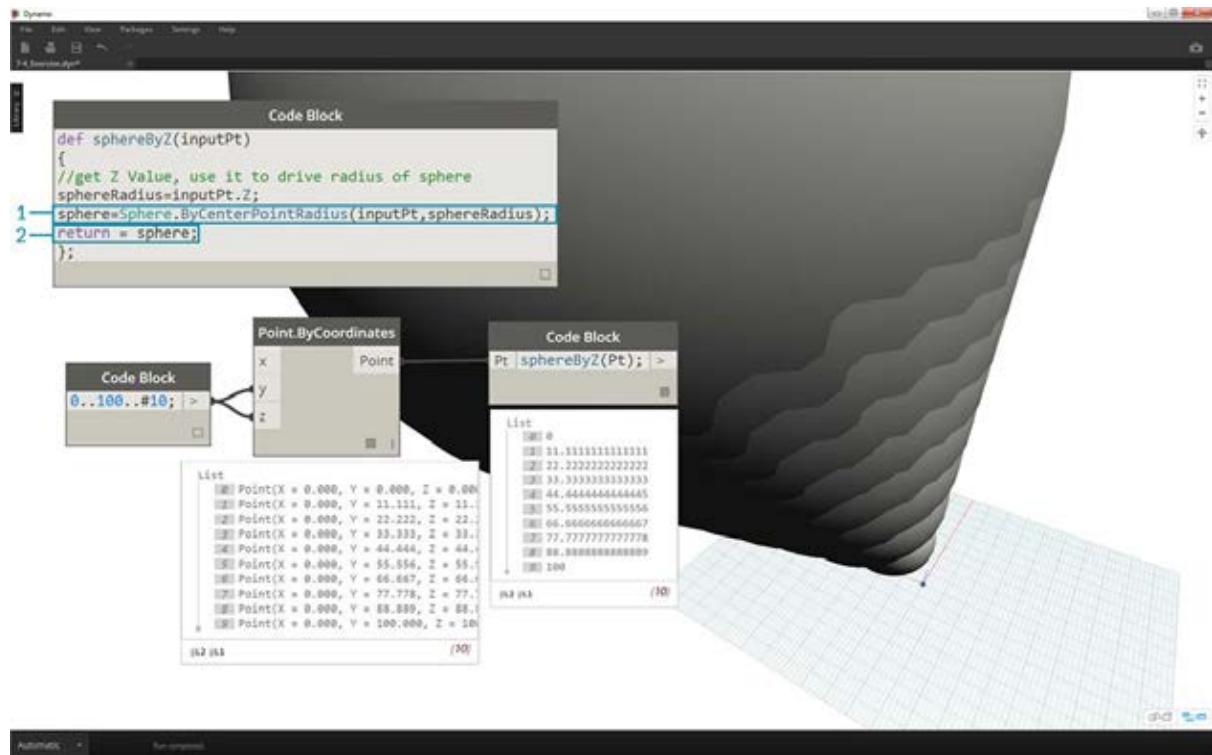
- 関数を呼び出して Pt という変数を作成して、前の手順で作成した点を接続します。

## sphereByZ(Pt)

- 出力されたリストを確認すると、値がすべて NULL になっていることがわかります。なぜでしょうか。これは、この関数を定義するときに *sphereRadius* 変数は計算しましたが、この関数が何を出力として返すのかを定義しなかったためです。\*\* これについては、次の手順で修正します。

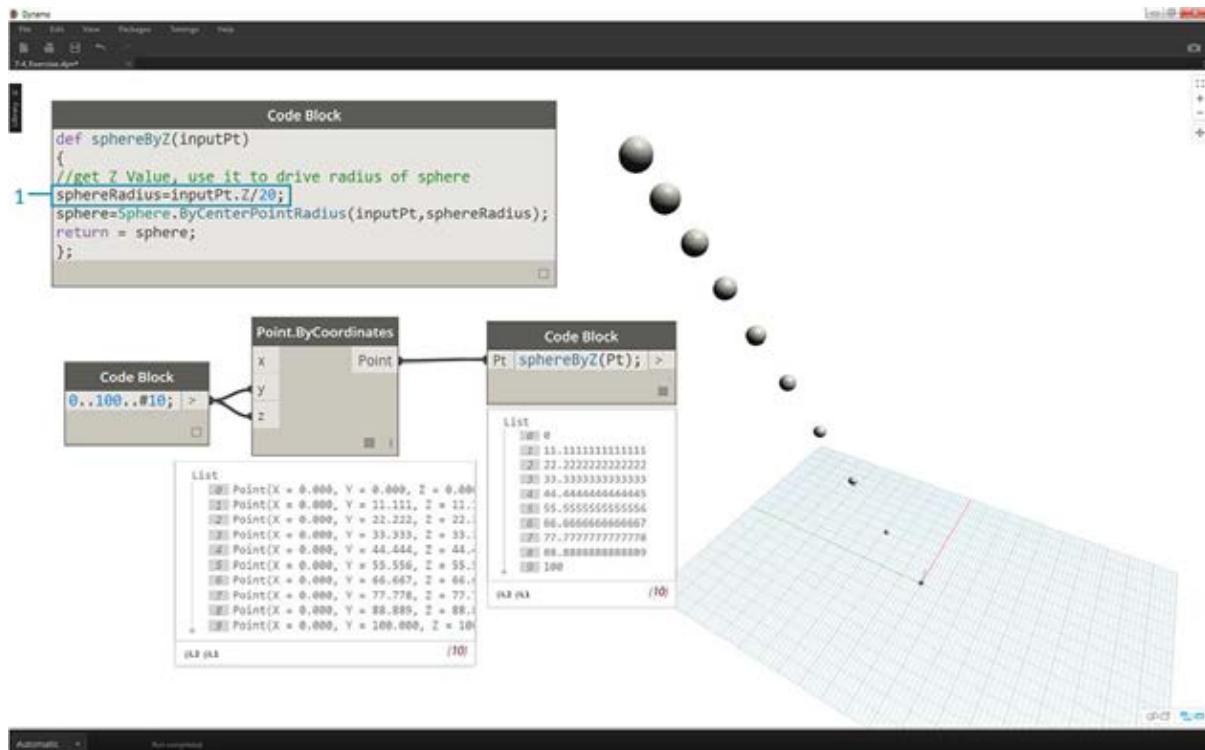


- sphereByZ* 関数に `return = sphereRadius` というコード行を追加して、関数の出力を定義する必要があります。これは重要な手順です。
- Code Block* ノードの出力は、各点の Z 座標です。

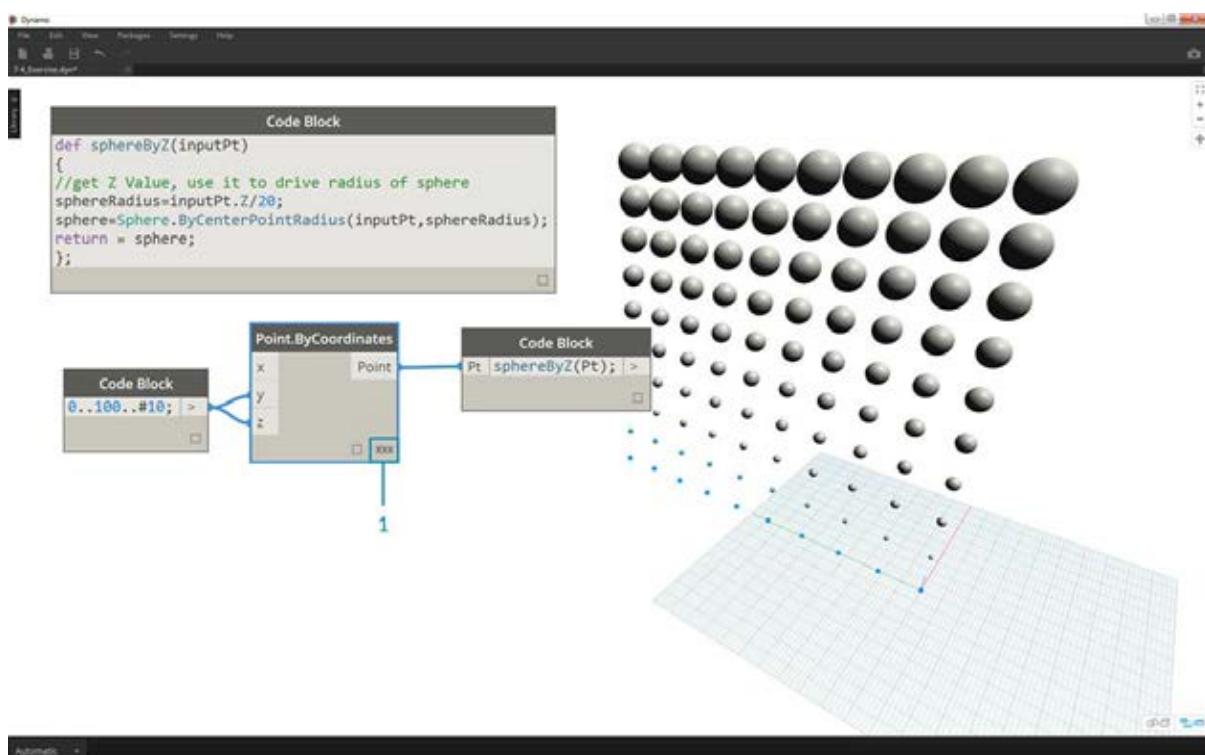


親関数を編集して、実際に球体を作成してみましょう。

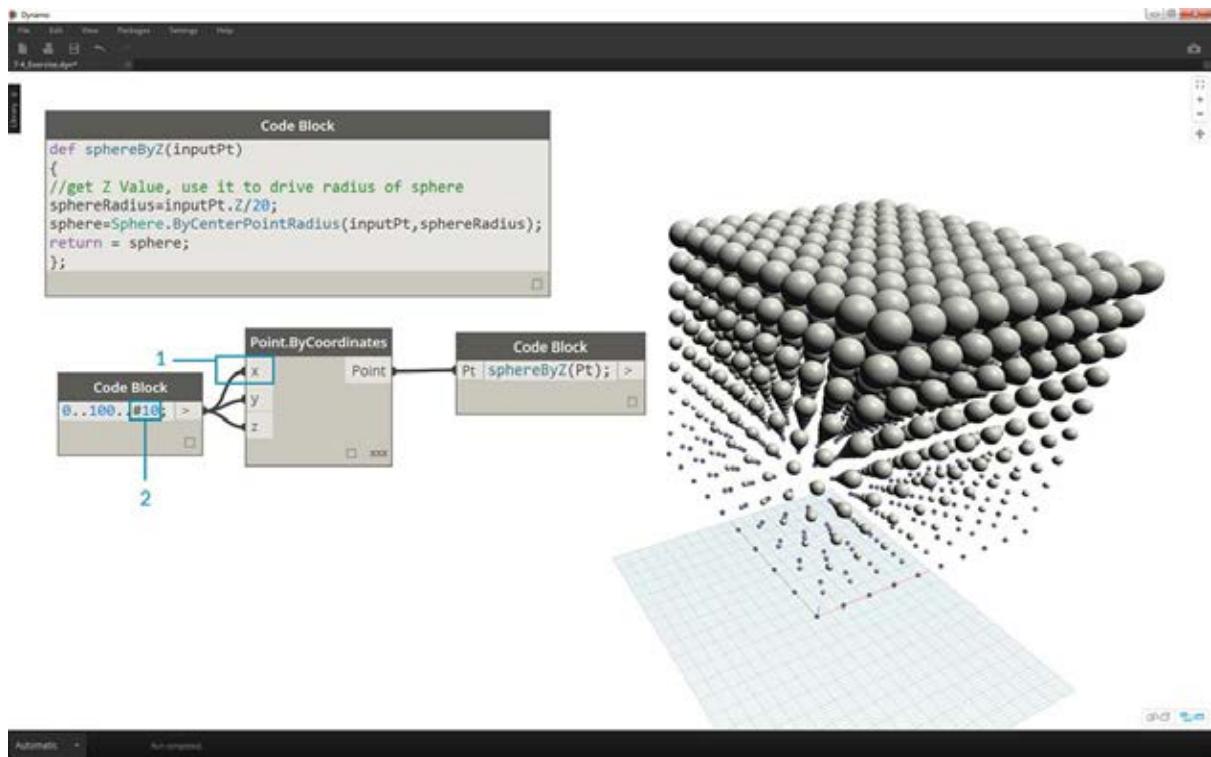
- 最初に、`sphere=Sphere.ByCenterPointRadius(inputPt, sphereRadius);`というコード行で球体を定義します。
- 次に、戻り値が `sphereRadius` から `sphere` となるように、`return = sphere;`と記述します。これで、Dynamo プレビューに非常に大きな球体が表示されます。



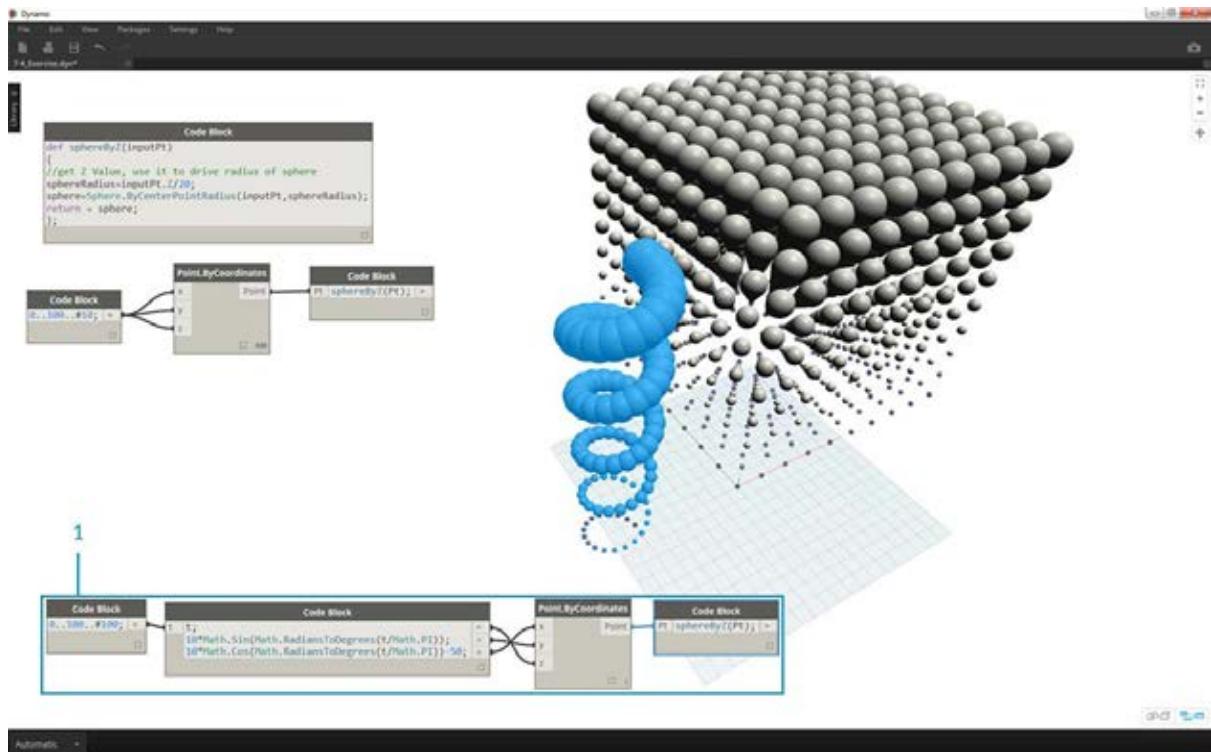
- `sphereRadius=inputPt.Z/20;`という除数を追加して `sphereRadius` の値を更新し、球体のサイズを調整します。これで、各球体が離れて表示され、半径と Z 値との関係がわかるようになります。



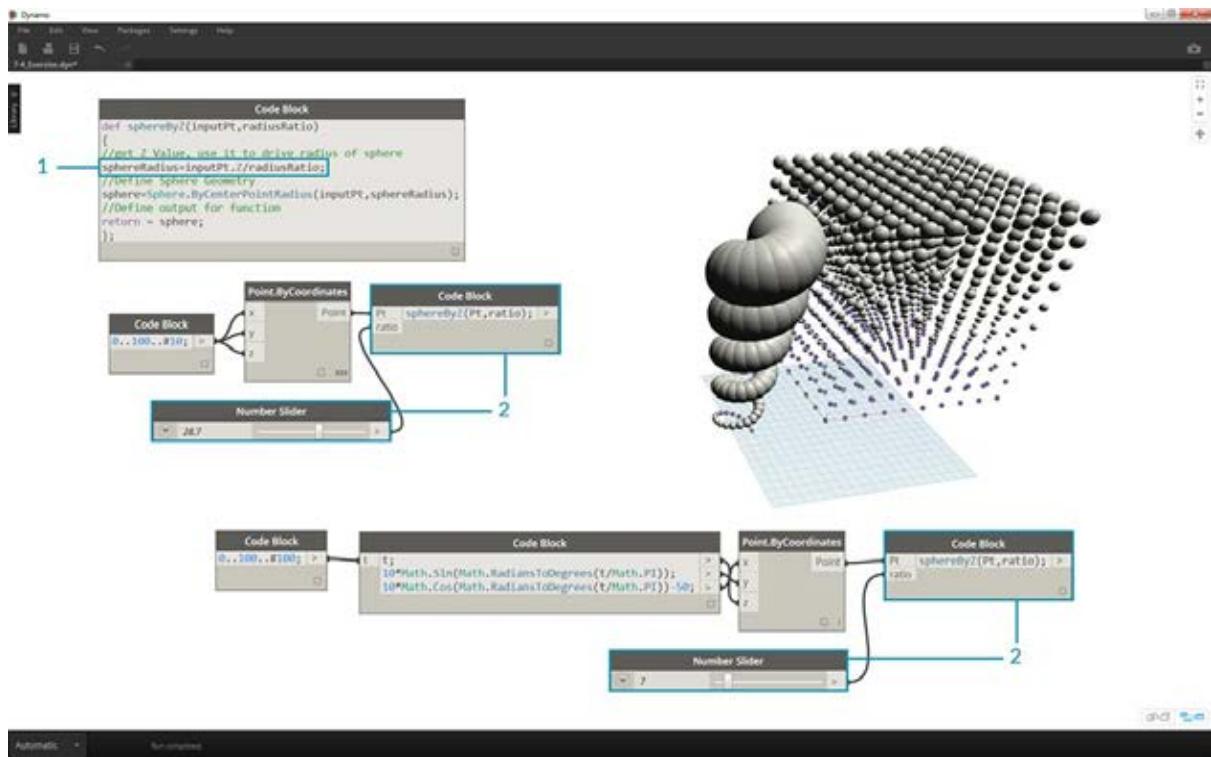
- Point.ByCoordinates** ノードでレーシングを[最短リスト]から[外積]に変更して、点のグリッドを作成します。`sphereByZ` 関数により、すべての点で Z 値に基づいた半径の円が作成されます。



1. 次に、**Point.ByCoordinates** ノードの X 入力に元の数値のリストを接続してみます。この操作により、球体によって構成される立方体が作成されます。
2. 注: この処理に時間がかかる場合は、#10 を #5 などの値に変更してください。



1. ここで作成した **sphereByZ** 関数は汎用的な関数であるため、前の演習で作成したらせん構造を呼び出して、この関数を適用することができます。



最後に、半径の比をユーザ設定のパラメータでコントロールしてみましょう。これを行うには、関数に対して新しい入力を作成し、除数の 20 をパラメータで置き換える必要があります。

1. `sphereByZ` 関数の定義を、次のように更新します。

```
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;
};
```

1. `sphereByZ(Pt, ratio);` のように、子の Code Block ノードの入力に `ratio` 変数を追加して更新します。次に、新しく作成した Code Block ノードの入力に Number Slider ノードを接続し、半径の比に基づいて半径のサイズを変更します。

## **Revit で Dynamo を使用する**

### **Revit で Dynamo を使用する**

Dynamo は、さまざまなプログラムで活用できるように設計された柔軟なプログラミング環境ですが、もともとは Revit と組み合わせて使用するために開発されました。ビジュアル プログラミングにより、ビルディング インフォメーション モデリング(BIM)の堅牢な設計案を作成することができます。Dynamo には、Revit 専用に設計されたさまざまなノードが用意されています。さらに建設業者のコミュニティでも、各種のサードパーティ製ライブラリが開発されています。この章では、Dynamo を Revit に組み込んで使用する際の基本的な操作方法について説明します。



# Revit との関係

## Revit との関係



Dynamo を Revit に組み込んで使用すると、Revit のビルディング インフォメーション モデリング(BIM)機能を、データとロジックに基づく Dynamo の視覚的なアルゴリズム編集環境によって拡張することができます。Dynamo の柔軟性を Revit の堅牢なデータベース機能と組み合わせることにより、BIM の新しい可能性が広がります。

この章では、Dynamo を使用した BIM ワークフローについて説明します。この章の各セクションでは、演習を行ながら BIM を確認していきます。BIM の視覚的なアルゴリズム編集機能の仕組みを理解するには、サンプルのプロジェクトで実際に操作するのが最適な方法です。ただしその前に、Dynamo の歴史を簡単に説明します。

#

### Revit のバージョンの互換性

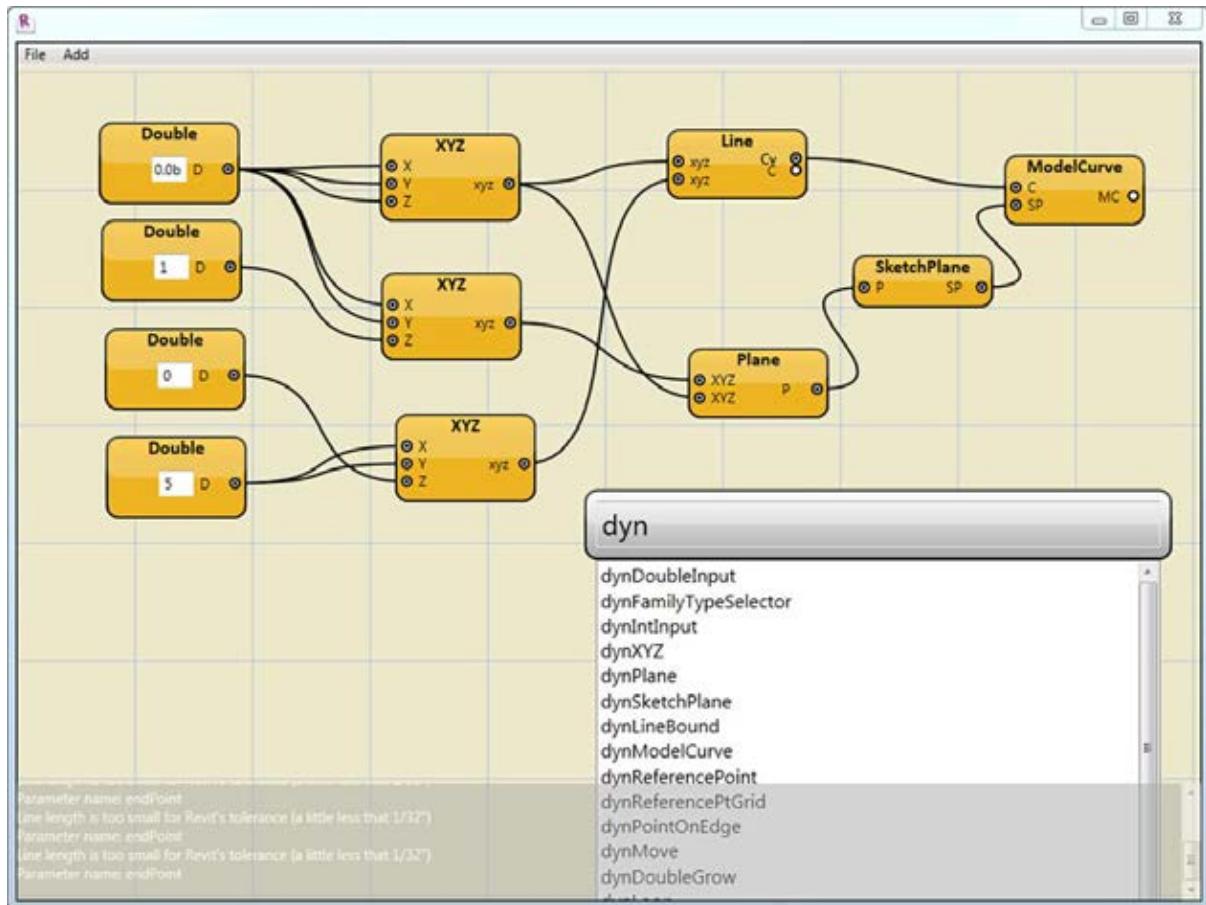
Revit と Dynamo はどちらも進化し続けているため、作業中の Revit のバージョンが、マシンにインストールされている Dynamo for Revit のバージョンと互換性がない場合があります。次の一覧は Revit と互換性のある Dynamo for Revit のバージョンの概要です。

#### Revit のバージョン 最も安定する Dynamo のバージョン サポートされる最も古い Dynamo for Revit のバージョン

2013	<a href="#">0.6.1</a>	<a href="#">0.6.3</a>
2014	<a href="#">0.6.1</a>	<a href="#">0.8.2</a>
2015	<a href="#">0.7.1</a>	<a href="#">1.2.1</a>
2016	<a href="#">0.7.2</a>	<a href="#">1.3.2</a>
2017	<a href="#">0.9.0</a>	<a href="#">最新のプレリース</a>
2018	<a href="#">1.3.0</a>	<a href="#">最新のプレリース</a>
2019	<a href="#">1.3.3</a>	<a href="#">最新のプレリース</a>

#

### Dynamo の歴史

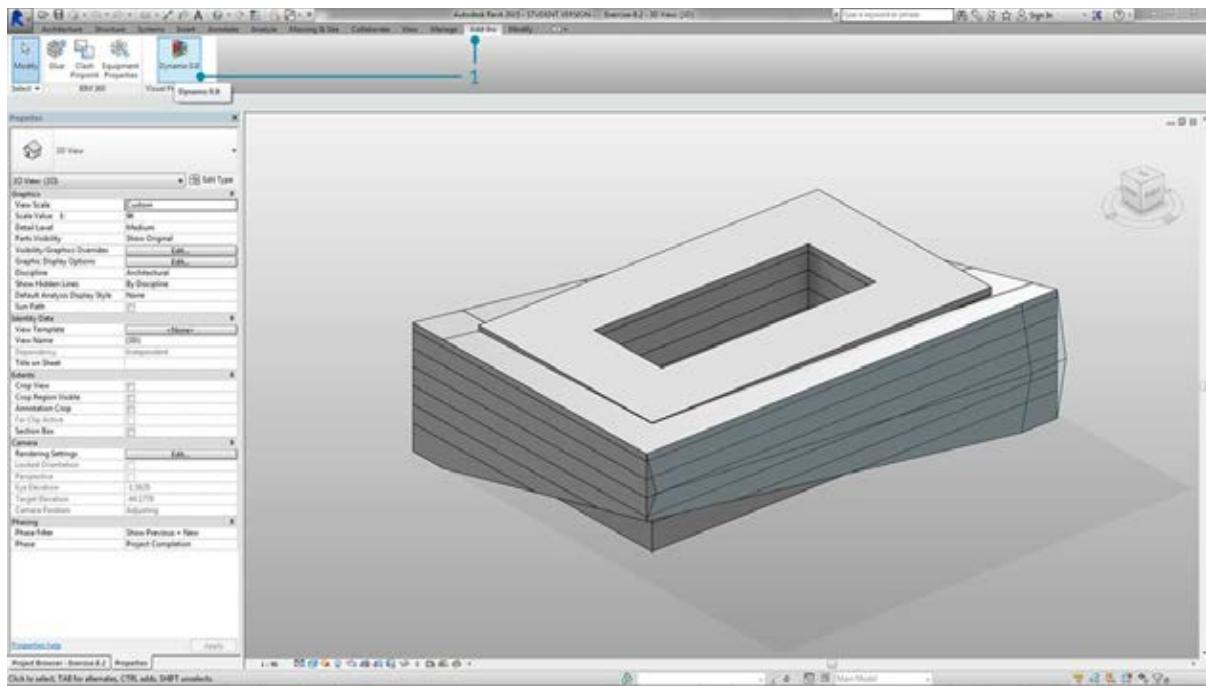


Dynamo プロジェクトは、開発チームとコミュニティの積極的なサポートによってここまで発展しましたが、最初の目標は小さなものでした。

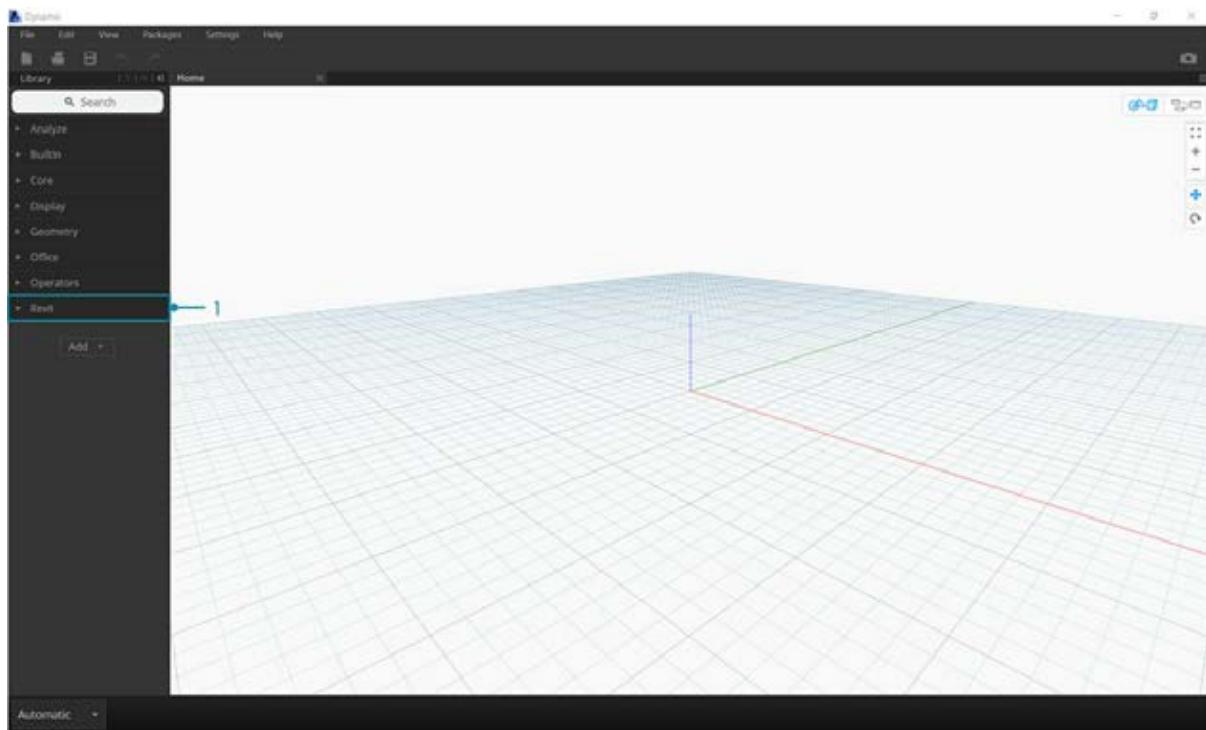
Dynamo は、もともと Revit の設計ワークフローを合理化するために開発されました。Revit では、プロジェクトごとに堅牢なデータベースが作成されますが、ユーザインターフェースの制約を受けることなくこの情報にアクセスすることは、一般的なユーザにとっては難しい場合があります。Revit には包括的な API (アプリケーションプログラムインターフェース) が用意されているため、サードパーティの開発者はこれらの API を使用して、カスタマイズされたツールを作成することができます。プログラマーであればこの API には慣れていますが、プログラミングの経験がないユーザにとっては、テキストベースのスクリプトを記述することは簡単なことではありません。Dynamo の開発チームは、わかりやすい視覚的なアルゴリズムエディタを提供することにより、Revit のデータを簡単に操作できるようにすることを目指しています。

カスタマイズされた Revit と Dynamo の主要なノードを組み合わせて使用すると、相互運用性、設計図書作成、解析、モデル生成などにおいて、パラメータ制御によるワークフローの範囲を大きく広げることができます。Dynamo を使用すれば、面倒なワークフロー作業を自動化し、設計作業に集中することができます。

## Revit で Dynamo を実行する



1. Revit プロジェクトやファミリ エディタで、[アドイン]タブから[Dynamo]をクリックします。Dynamo は、Dynamo を起動したファイル内でのみ実行されることに注意してください。



1. Revit で Dynamo を起動すると、Dynamo のライブラリ内に[Revit]という新しいカテゴリが表示されます。この新しいカテゴリから、Revit ワークフロー専用のノードにアクセスすることができます。

\*注: Dynamo グラフで Revit 固有のファミリを扱うノードを使用する場合、そのグラフは Revit で稼働している Dynamo から開いたときにのみ正常に動作します。たとえば、Revit で稼働している Dynamo のグラフを Dynamo Sandbox で開くと、Revit ノードが失われます。

### ノードのフリーズ

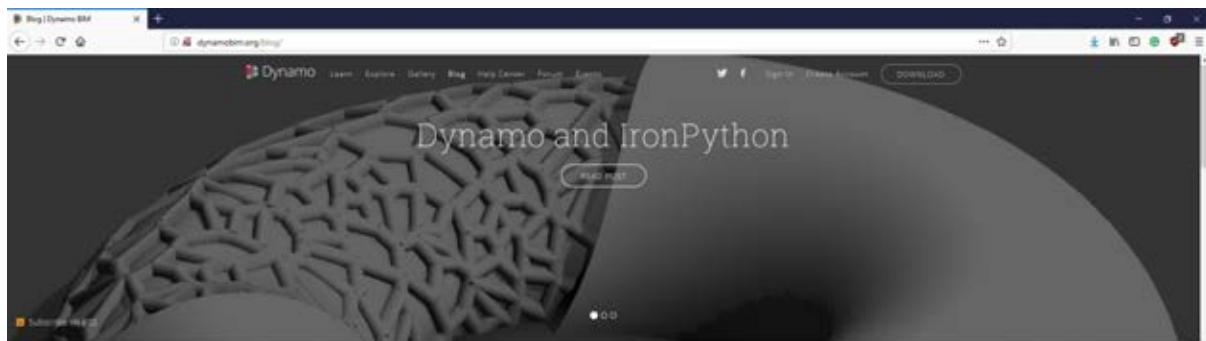
Revit は堅牢なプロジェクト管理を提供するプラットフォームであるため、場合によっては Dynamo のパラメータ操作が複雑になり、計算速度が低下することがあります。Dynamo によるノード計算で時間がかかる場合は、ノードを「フリーズ」する機能を使用して、

グラフの開発中に Revit 関連操作の実行を停止することができます。ノードをフリーズする操作の詳細については、「ソリッド」の章の「フリーズ」セクションを参照してください。

## コミュニティ

Dynamo は、もともと建築設計者や構造設計者向けに開発されたツールです。Dynamo のコミュニティは、建設業界の専門家と交流しながら学ぶことができる場所として、現在も成長を続けています。Dynamo のコミュニティは、情報の共有と開発プロジェクトに積極的に参加する建築設計者、構造設計者、プログラマ、デザイナーによって構成されています。

Dynamo は、継続的に進化していくオープンソース プロジェクトであり、その開発の大部分は Revit に関係しています。ディスカッション フォーラムにアクセスし、[質問を投稿してみてください](#)。プログラマとして Dynamo プロジェクトに参加する場合は、次のリンクを参照してください: [GitHub の Dynamo ページ](#) また、[Dynamo Package Manager](#) では、さまざまなサードパーティ製ライブラリが提供されています。提供されているパッケージの多くは、建設業界のワークフローで使用することを目的として作成されています。実際に、パネル作成用のサードパーティ製パッケージを使用してみましょう。



## Dynamo and IronPython

The Dynamo team recently became aware of some reports that python in Windows was not functioning on some machines with Dynamo installed. We've added some information to the [Dynamo FAQ] will attempt, and would like to share it here as well in case you run into a similar problem. Q: How do I get [Dynamsoft Python] functioning to work again, and use it?

### Archive

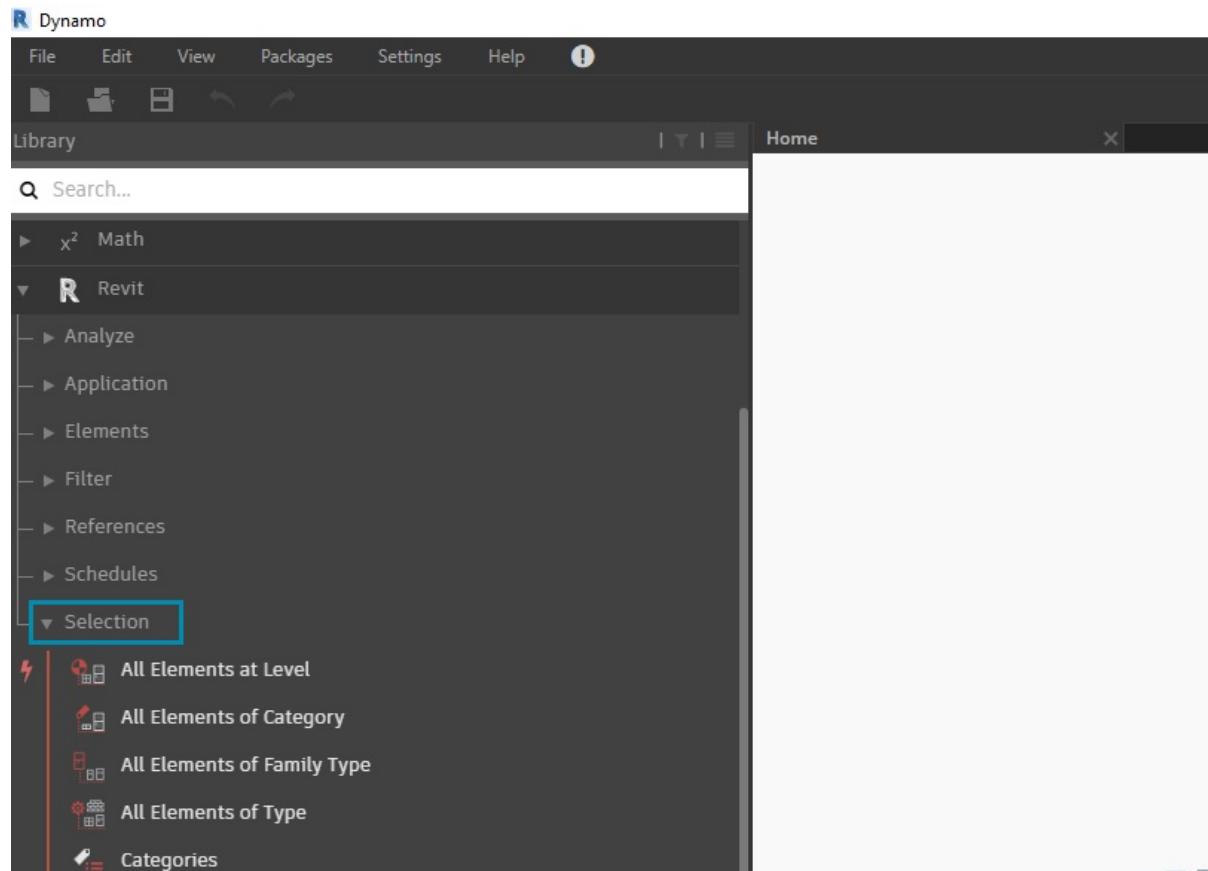
MARCH 2018  
January 2018  
December 2017  
November 2017  
October 2017  
September 2017  
August 2017  
July 2017  
April 2017  
March 2017  
January 2017  
December 2016  
November 2016  
October 2016

Dynamo 開発チームは、[ブログ](#)を頻繁に更新しています。最近の記事を確認し、最新の開発情報を入手してください。

# 選択

## 選択

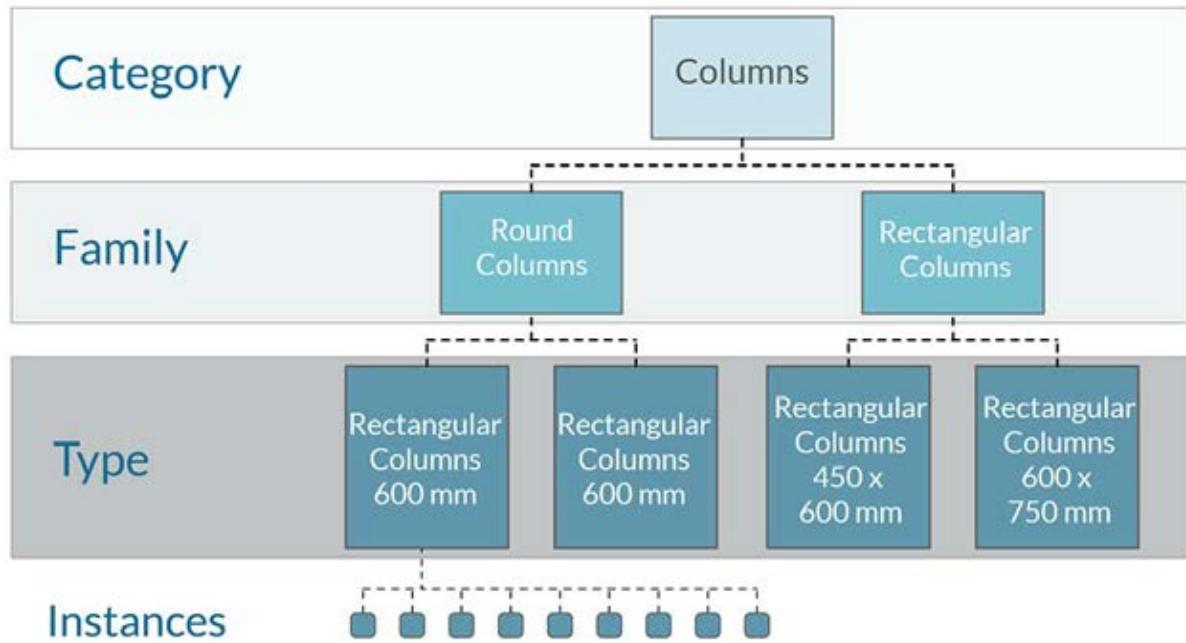
Revit は、非常に豊富なデータを扱う環境です。このため、選択機能が「ポイント アンド クリック」をはるかに超えた範囲にまで拡張されています。パラメトリックな操作の実行中に、Revit のデータベースにクエリーを行い、Revit の要素を Dynamo のジオメトリに動的にリンクすることができます。



[Revit]ライブラリの[選択]カテゴリから、さまざまな方法でジオメトリを選択することができます。

Revit の要素を選択するには、Revit の要素の階層構造について十分に理解しておくことが大切です。プロジェクト内のすべての壁を選択するには、カテゴリ単位で選択します。ミッドセンチュリー モダン スタイルのロビーに配置したイームズ チェアをすべて選択するには、ファミリ単位で選択します。演習に入る前に、Revit の階層について簡単におさらいしておきましょう。

## Revit の階層

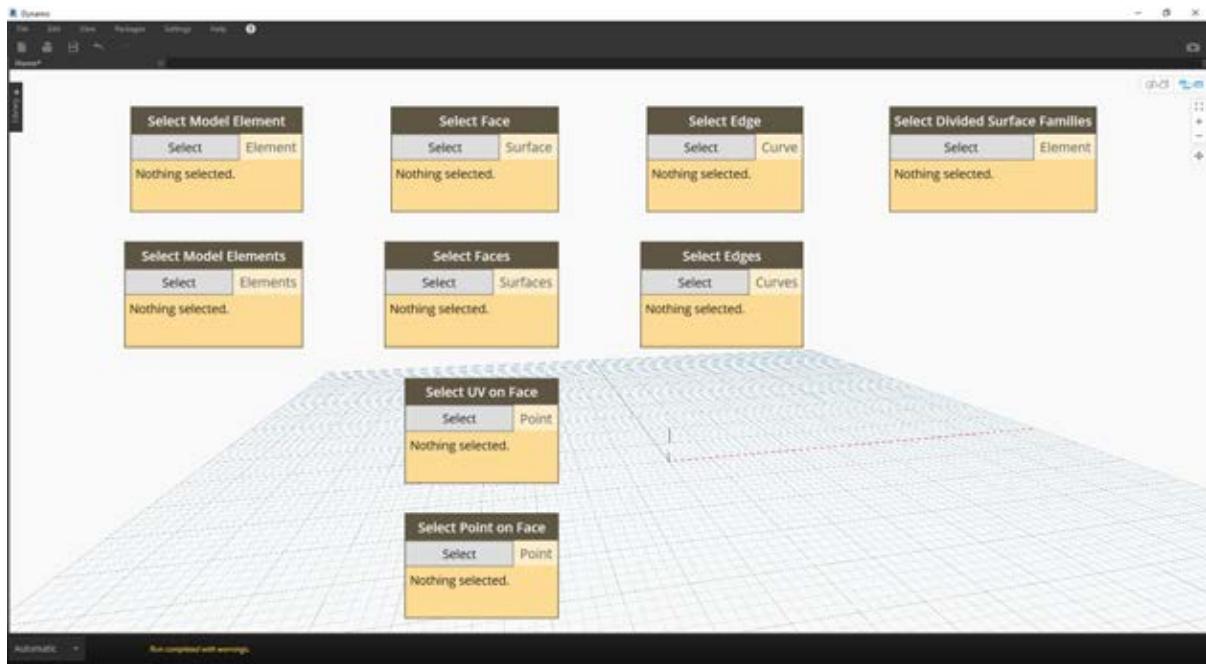


生物学ではあらゆる生物が体系的に分類されており、その分類法は、上位から下位にかけて、界、門、綱、目、科、属、種という階層構造から成り立っています。Revitにおける要素の分類法はこれに似ています。基本的には、Revitの階層構造は、上位から下位にかけて、カテゴリ、ファミリ、タイプ、インスタンスに分かれています。インスタンスは(ユニークなIDを持つ)個別のモデル要素です。カテゴリは、「壁」や「床」などの一般的なグループのことです。このようにして構成されているRevitのデータベースを使用して、1つの要素を選択したり、階層構造の中の指定したレベルに基づいて同種の要素をすべて選択することができます。

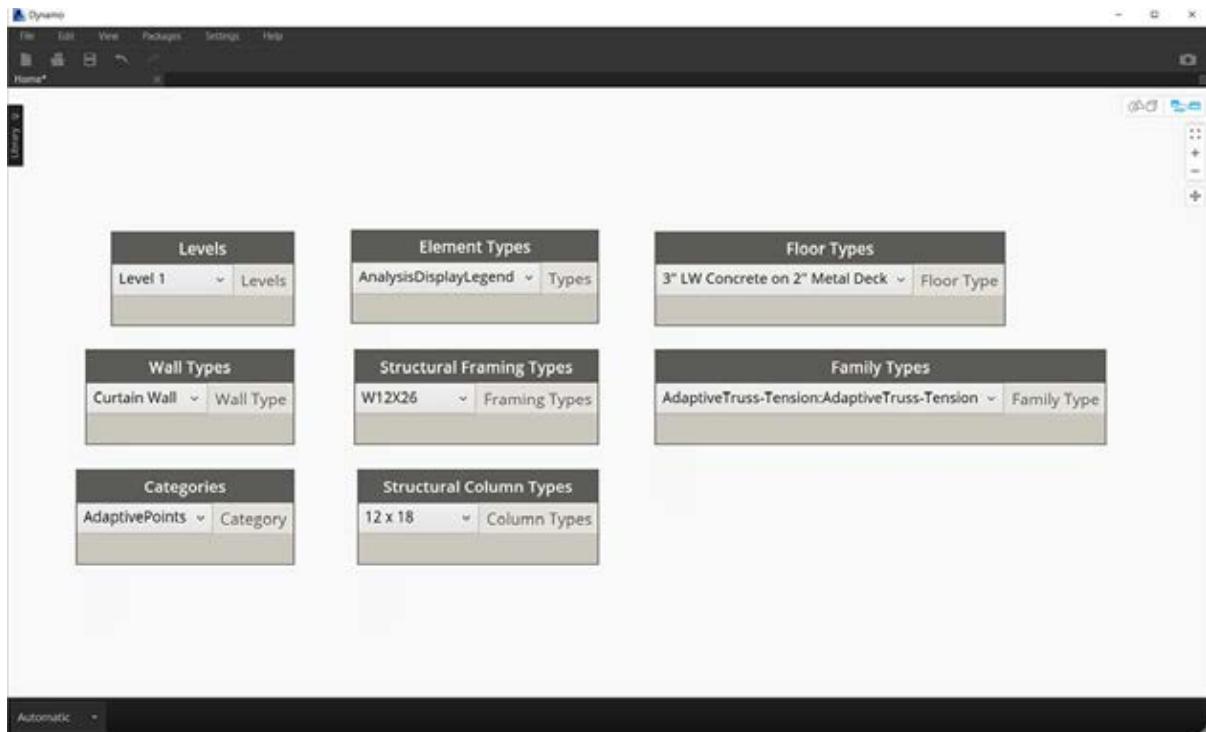
**注:** Revitにおける「タイプ」の定義は、プログラミングでいう「型」とは異なります。Revitでいう「タイプ」は、いわゆる「データ タイプ」ではなく、分類階層における1つの枝を指します。

#### Dynamo のノードを使用したデータベース ナビゲーション

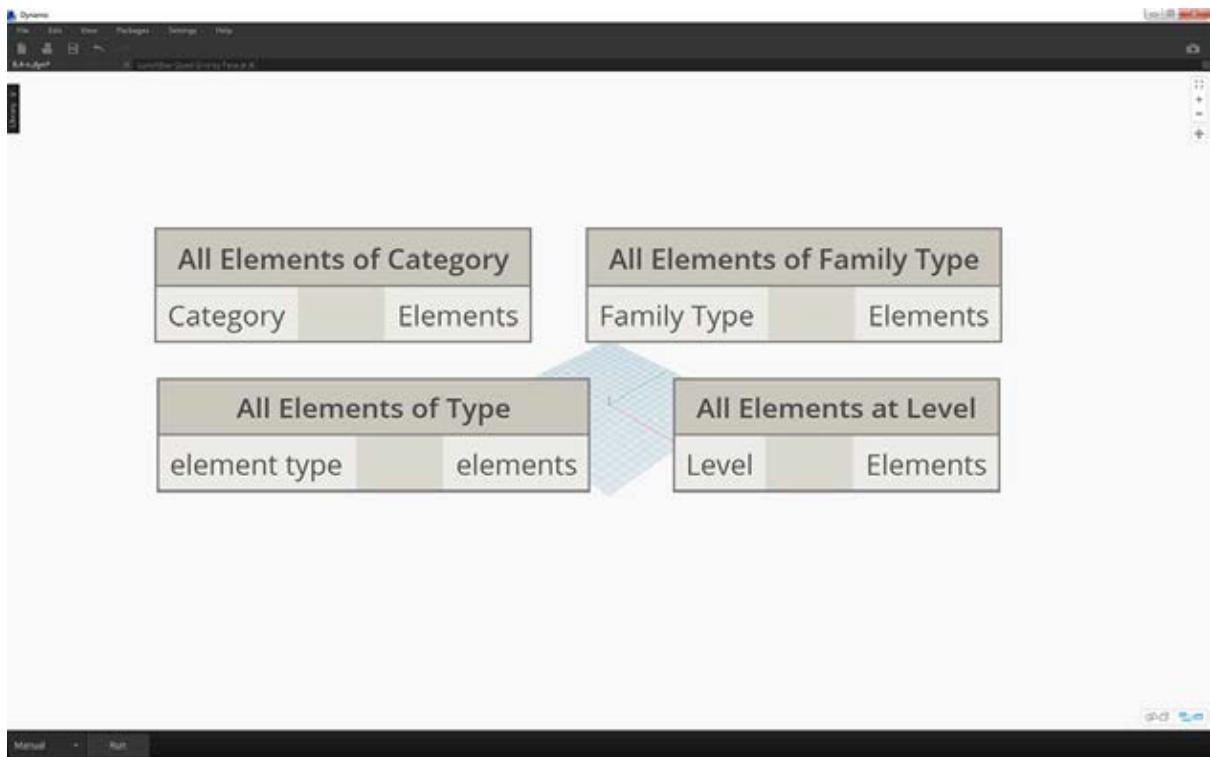
次の3つの画像では、Revitの要素を Dynamo で選択するいくつかの主だった方法を紹介しています。これらのツールを組み合わせて使用すると便利です。その一部をこれ以降の演習で実際に使用してみましょう。



ポイントアンドクリックは、Revit の要素を直接選択する最も簡単な方法です。モデル要素全体であれ、そのトポロジの一部（たとえば 1 つの面や 1 つのエッジ）であれ、選択することができます。この方法では Revit オブジェクトへの動的なリンクが維持されるので、Revit ファイルの場所やパラメータが変更されると、グラフ内で参照されている Dynamo の要素が更新されます。



ドロップダウンメニューで、Revit プロジェクト内のアクセス可能なすべての要素のリストが作成されます。ビューで確認できない Revit の要素も含めて、これで参照することができます。Revit プロジェクトやファミリ エディタで既存の要素をクエリーしたり、新しい要素を作成するには、このツールが役に立ちます。



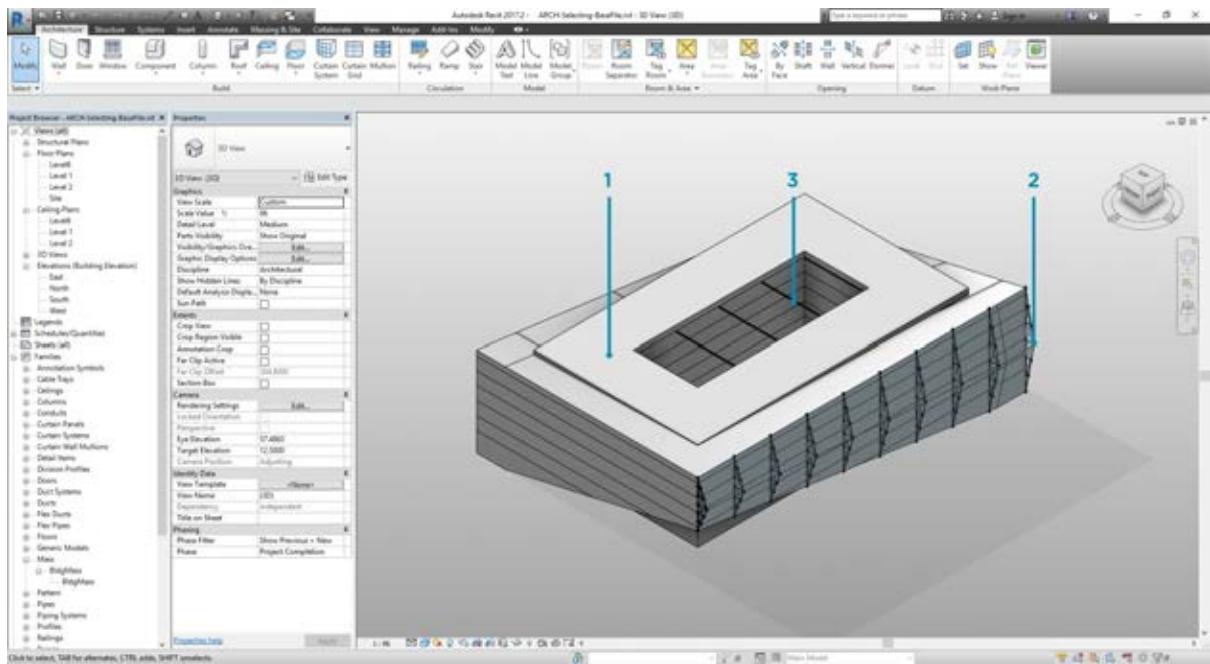
Revit の要素の選択方法としては、他に Revit の階層構造から特定の階層を指定する方法もあります。この方法は、設計図書作成、インスタンスの生成とカスタマイズなどのために準備されている大規模なデータ配列をカスタマイズするのにとても役立ちます。

上記の 3 点の画像に留意して、この章の続きで説明するパラメトリック アプリケーションの作成に備えて、基本的な Revit プロジェクトから要素を選択する演習を開始しましょう。

## 演習

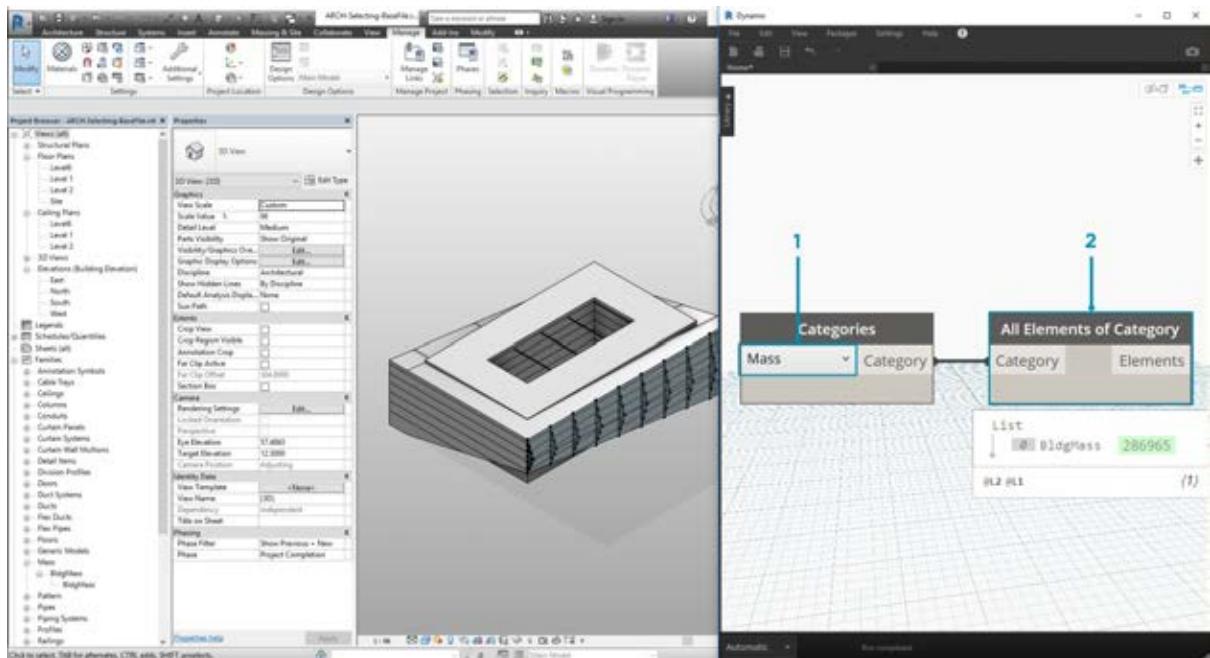
この演習に付属しているサンプル ファイルをダウンロードしてください(右クリックして[名前を付けてリンク先を保存]を選択)。すべてのサンプル ファイルの一覧については、付録を参照してください。

1. [Selecting.dyn](#)
2. [ARCH-Selecting-BaseFile.rvt](#)



この Revit ファイルのサンプルには、3 つの要素タイプから成る 1 つの単純な建物モデルが収録されています。それでは、このモデルを見本として使用して、次に挙げる Revit の階層構造の中で Revit の要素を選択してみましょう。

1. 建物のマス(基本形状)
2. ト拉斯(アダプティブ コンポーネント)
3. 梁(構造フレーム)

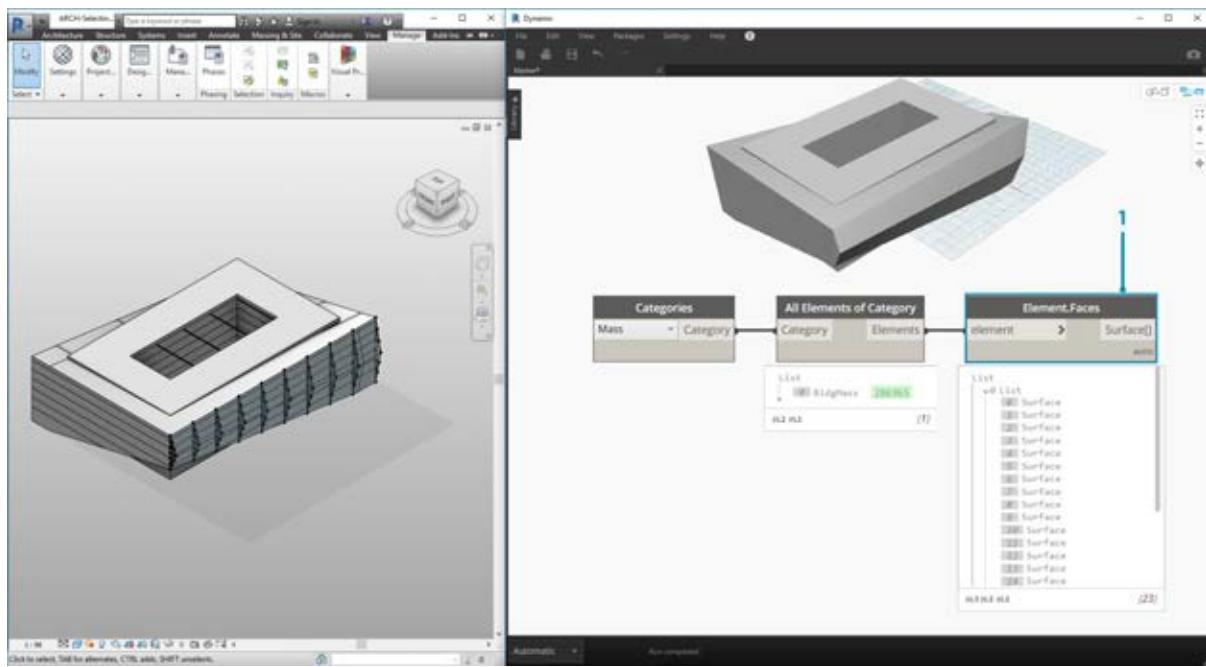


Revit のプロジェクトビューに表示されている要素から、どのような結論を導き出すことができますか。また、適切な要素を選択するには、階層をいくつ降りていく必要があるでしょうか。言うまでもありませんが、大規模なプロジェクトで作業する際にはこの問題はもっと複雑になります。使用できるオプションはたくさんあり、カテゴリ別、レベル別、ファミリ別、インスタンス別に要素を選択することができます。

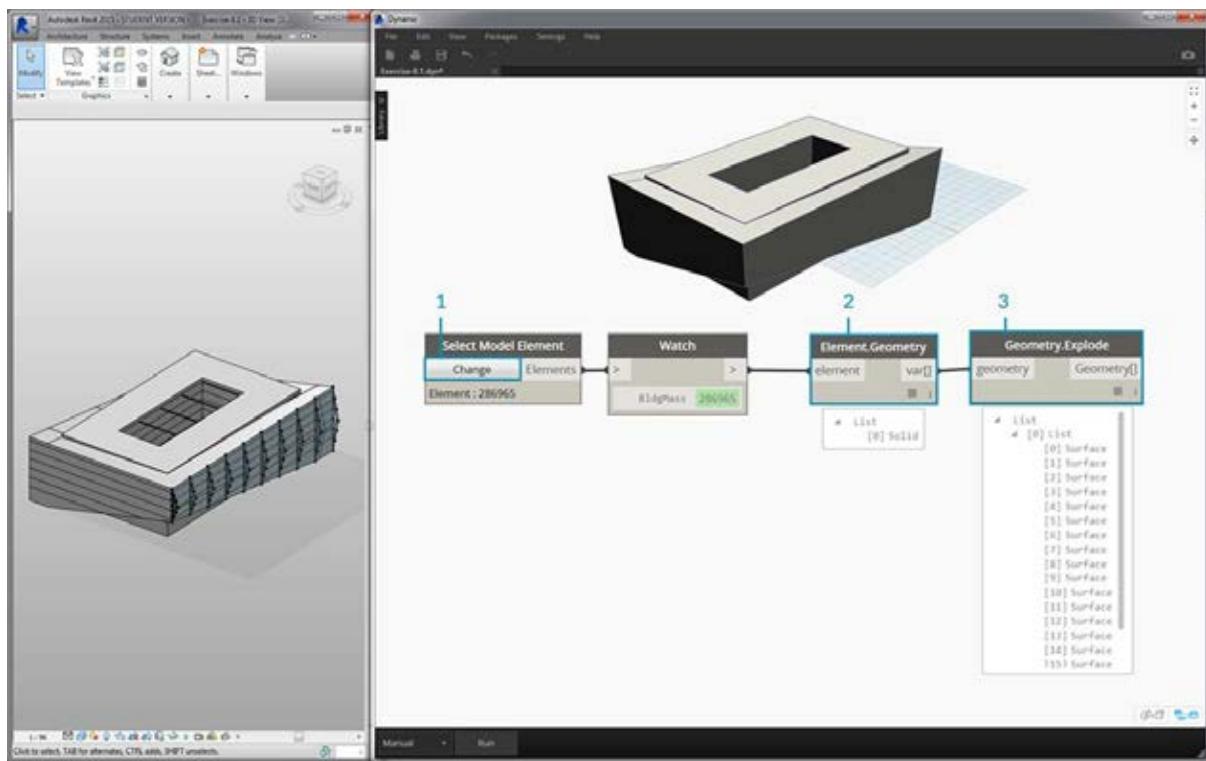
1. ここでは基本的な構造を扱っているので、Categories ドロップダウン ノードで[Mass]を選択して、建物のマスを選択してみましょう。これは、[Revit] > [選択]タブにあります。
2. [Mass]カテゴリのノードでは、単純にカテゴリ自体が出力されるので、要素を選択する必要があります。これを実行するには、All Elements of Category ノードを使用します。

この時点では、Dynamo にジオメトリが表示されないことに注意してください。Revit の要素が既に選択されていますが、まだ Dynamo ジオメトリに変換されていません。この区別は重要です。多数の要素を選択する場合は、すべての動作が非常に遅くなることがあるため、すべての要素を Dynamo でプレビューすることは好ましくありません。Dynamo は、ジオメトリ操作の実行を必要とせずに Revit のプロジェクトを管理することのできるツールです。そのことについては、この章の次のセクションで説明します。

ここでは単純なジオメトリを使用して演習を行っているので、Dynamo のプレビューでジオメトリを表示できるようにします。上記の Watch ノード中の「BldgMass」の隣に、緑の背景色付きで数値が表示されています。これは要素の ID を表しており、この ID からユーザーが Dynamo ジオメトリではなく Revit 要素を扱っていることがわかります。次の手順で、この Revit 要素を Dynamo ジオメトリに変換してみましょう。

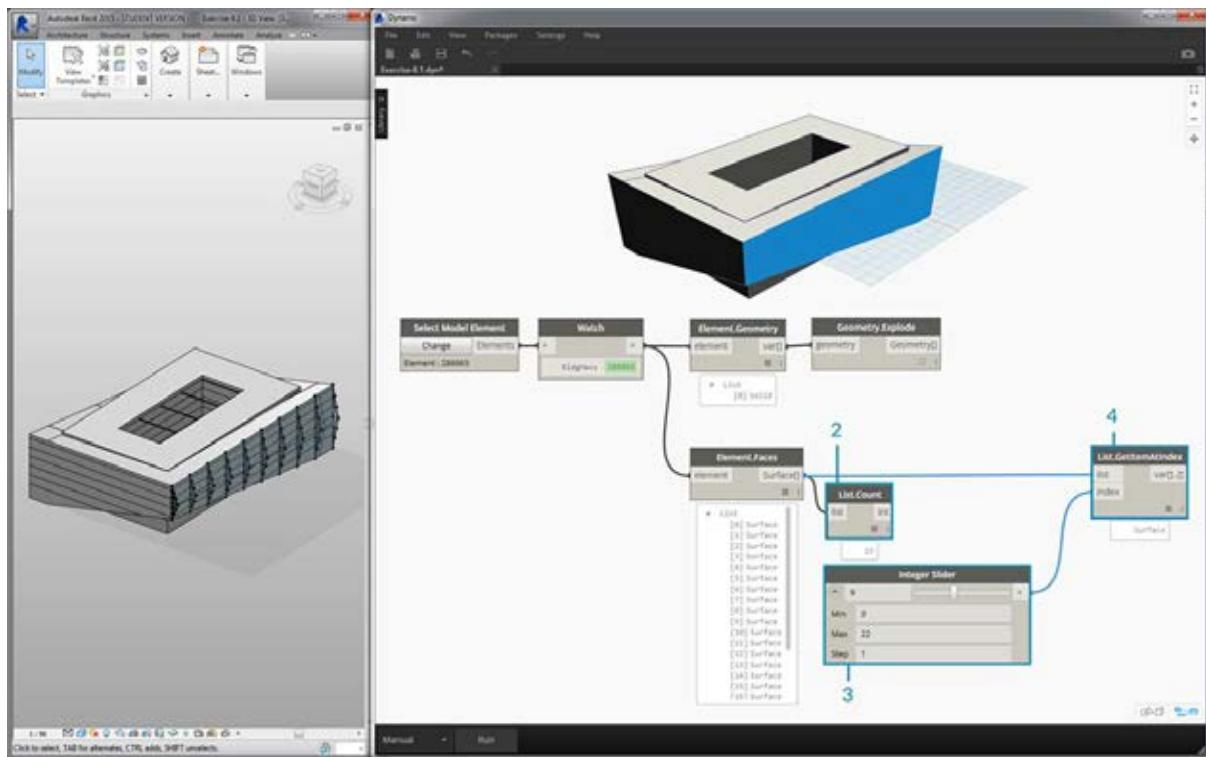


1. *Element.Faces* ノードを使用して、Revit のマスの各面を表すサーフェスのリストを取得します。これで、Dynamo のビューポートでジオメトリを表示し、その面をパラメトリック操作の際に参照できるようになりました。

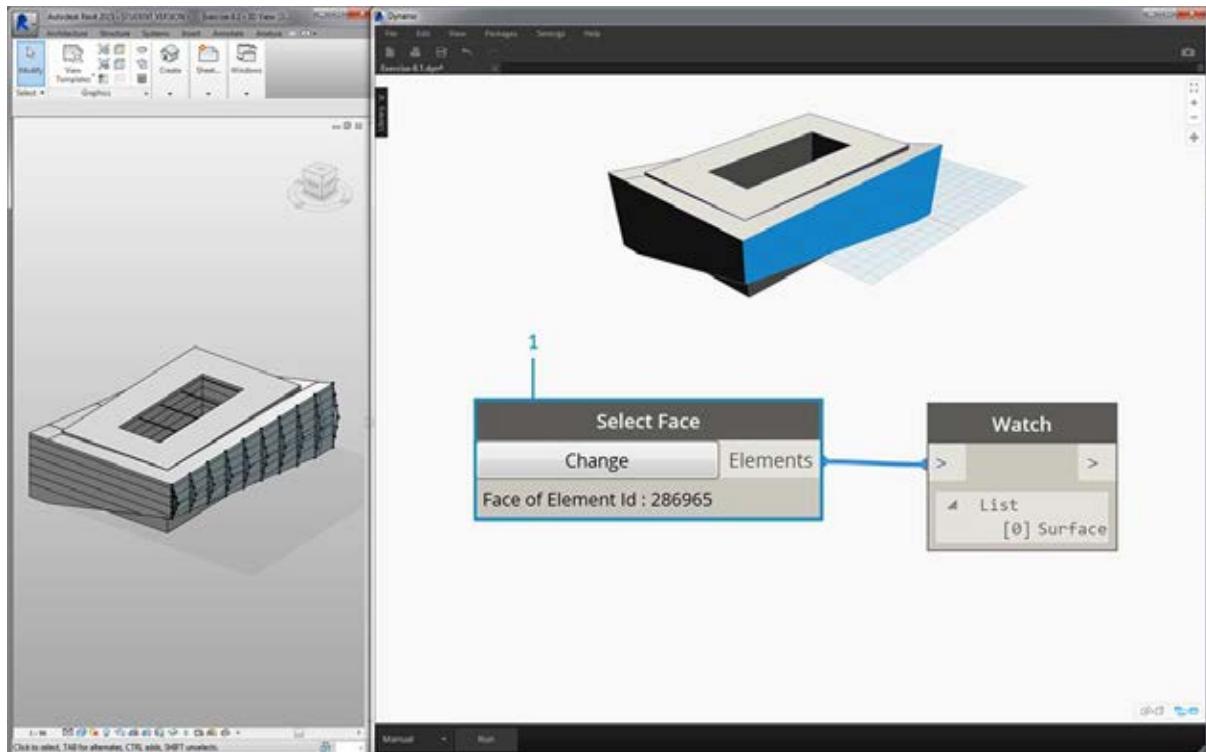


別の方法を紹介します。この場合は、(All Elements of Category ノードを使用して) Revit の階層構造から選択を行い、Revit のジオメトリを指定して明示的に選択するという方法を探りません。

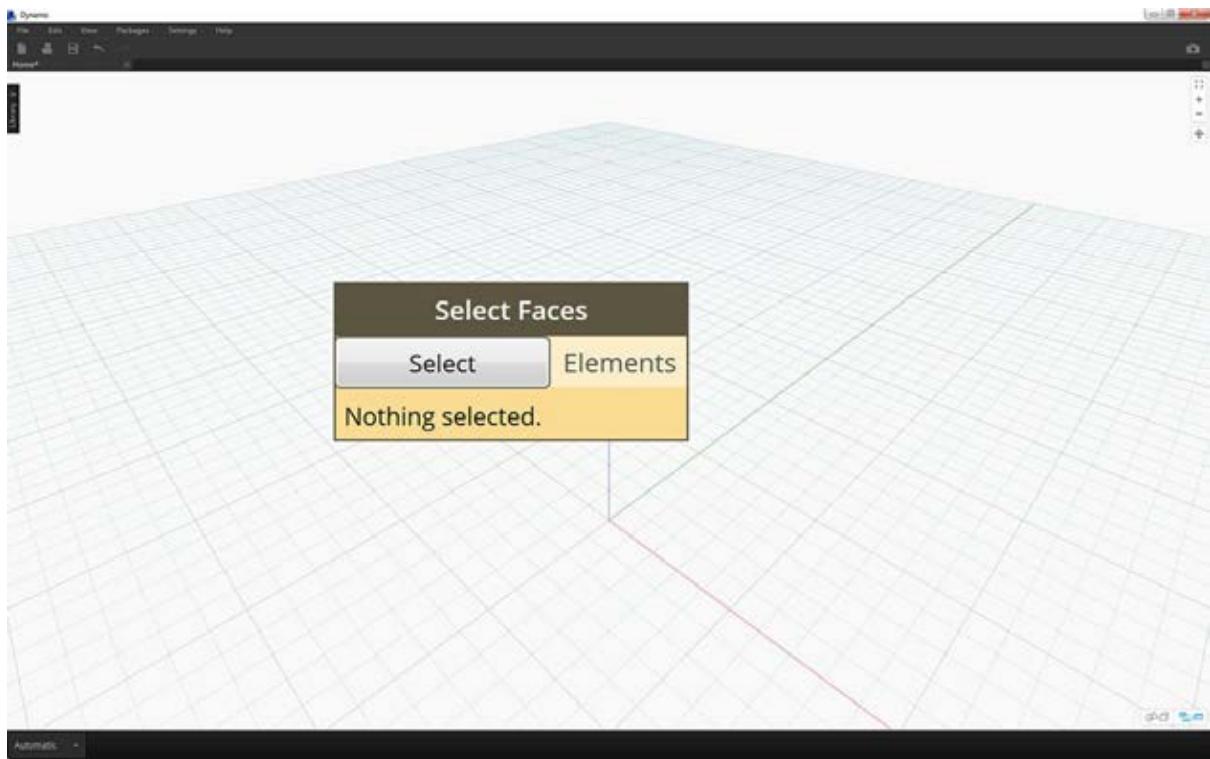
1. *Select Model Element* ノードを使用して、[選択] (または[変更])ボタンをクリックします。Revit のビューポートで、目的の要素を選択します。この場合は、建物のマスを選択することにします。
2. *Element.Faces* ノードではなく *Element.Geometry* ノードを使用すると、マス全体を 1 つのソリッド ジオメトリとして選択することができます。このノードでは、そのマスの内部に含まれるすべてのジオメトリが選択されます。
3. *Geometry.Explode* ノードを使用すると、やはりサーフェスのリストを取得することができます。これら 2 つのノードは、*Element.Faces* と同様の機能に加えて、Revit 要素のジオメトリを掘り下げるのに役立つオプションを提供します。



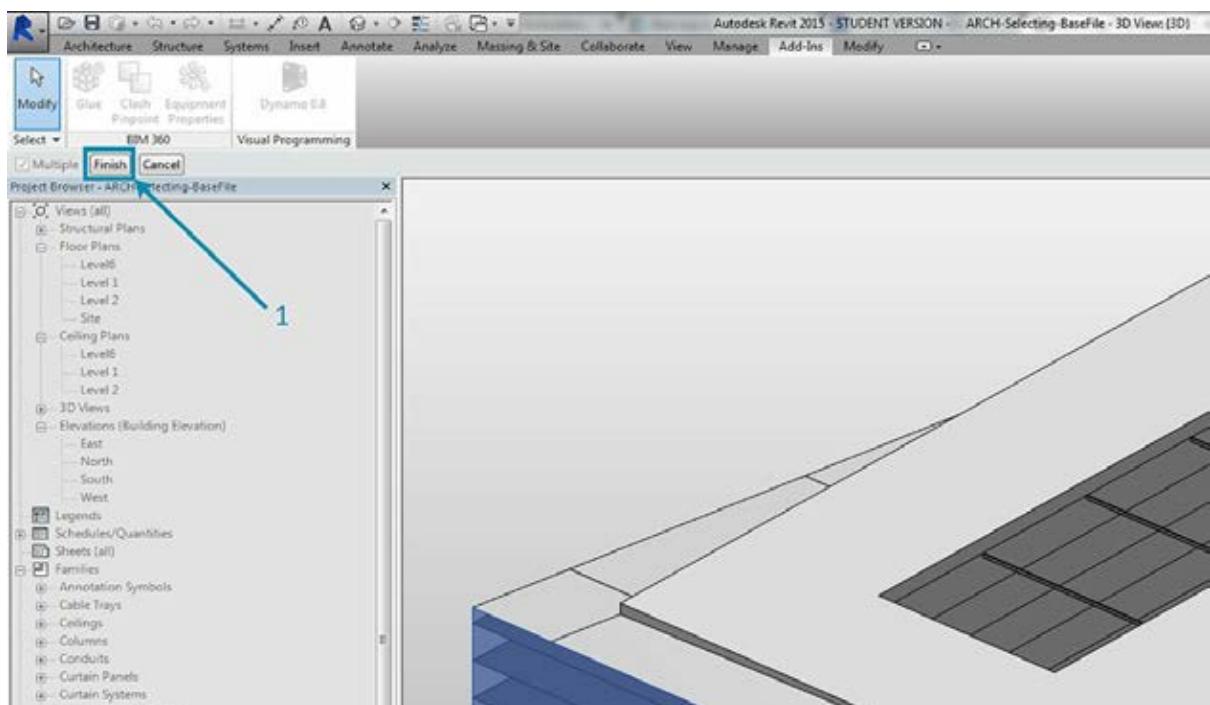
1. リストの基本的 操作をいくつか行うことで、対象の面のクエリーを実行できます。
2. まず“List.Count”ノードで、作業中のマスに 23 のサーフェスが含まれていることを確認します。
3. この数値を基準として、Integer Slider ノードの最大値を「22」に変更します。
4. List.GetItemAtIndex ノードを使用して、リストと index に使用する Integer Slider ノードを入力します。選択したスライダを index 9 まで動かして、トラスをホストしているメイン ファサードが選択表示された段階で停止します。



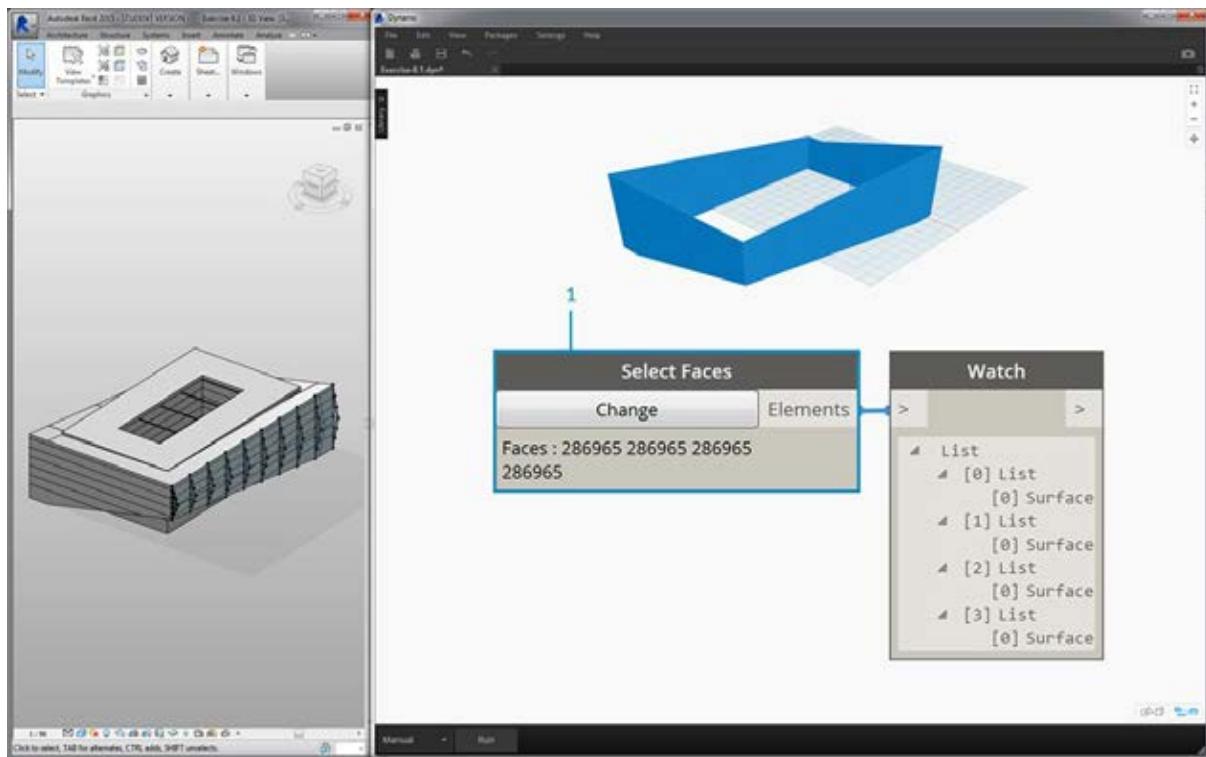
1. 前の手順は少しばかり面倒でした。Select Face ノードを使用すると、同じことをもっとすばやく実行することができます。これにより、Revit プロジェクト内ではそれ自体で 1 つの要素として扱われない面も選択できます。Select Model Element でも同様の操作を行うことができます。ただし、こちらでは要素全体ではなくサーフェスを選択します。



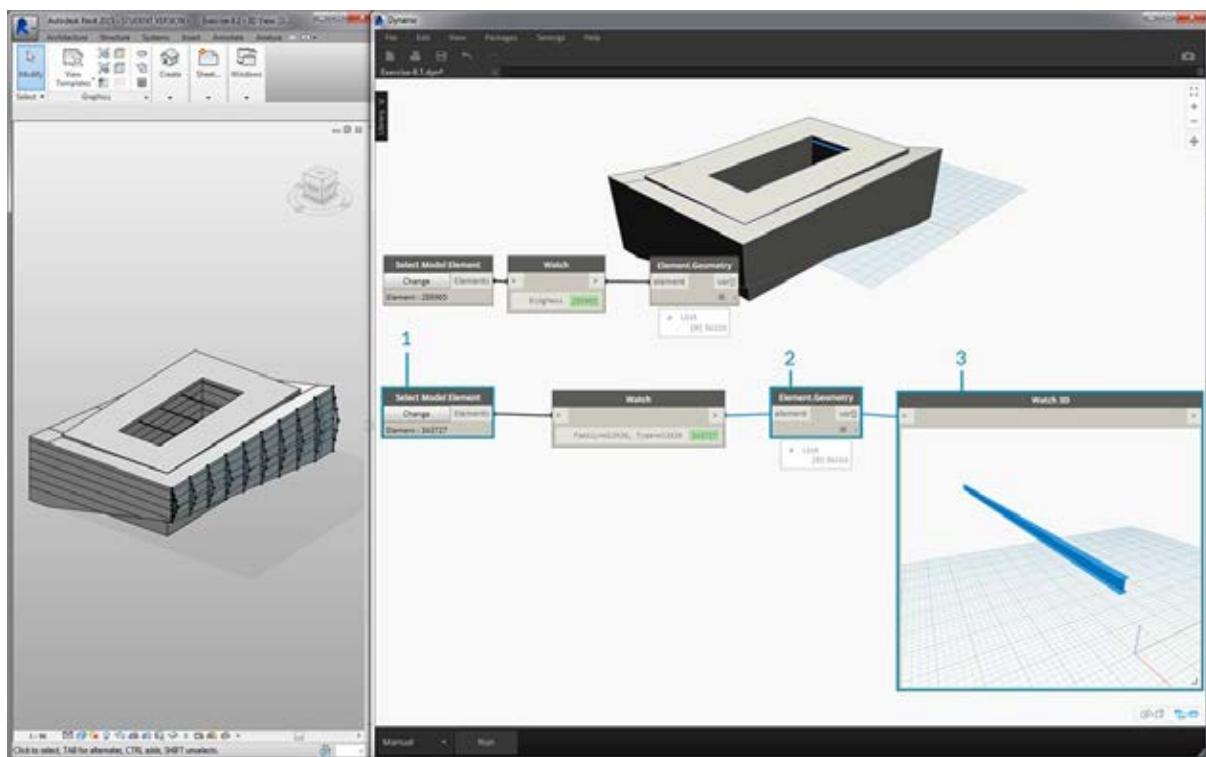
建物のメインファサードの壁を選択してみましょう。Select Faces ノードを使用してこれを行うことができます。[選択]ボタンをクリックし、Revit の 4 つのメインファサードを選択します。



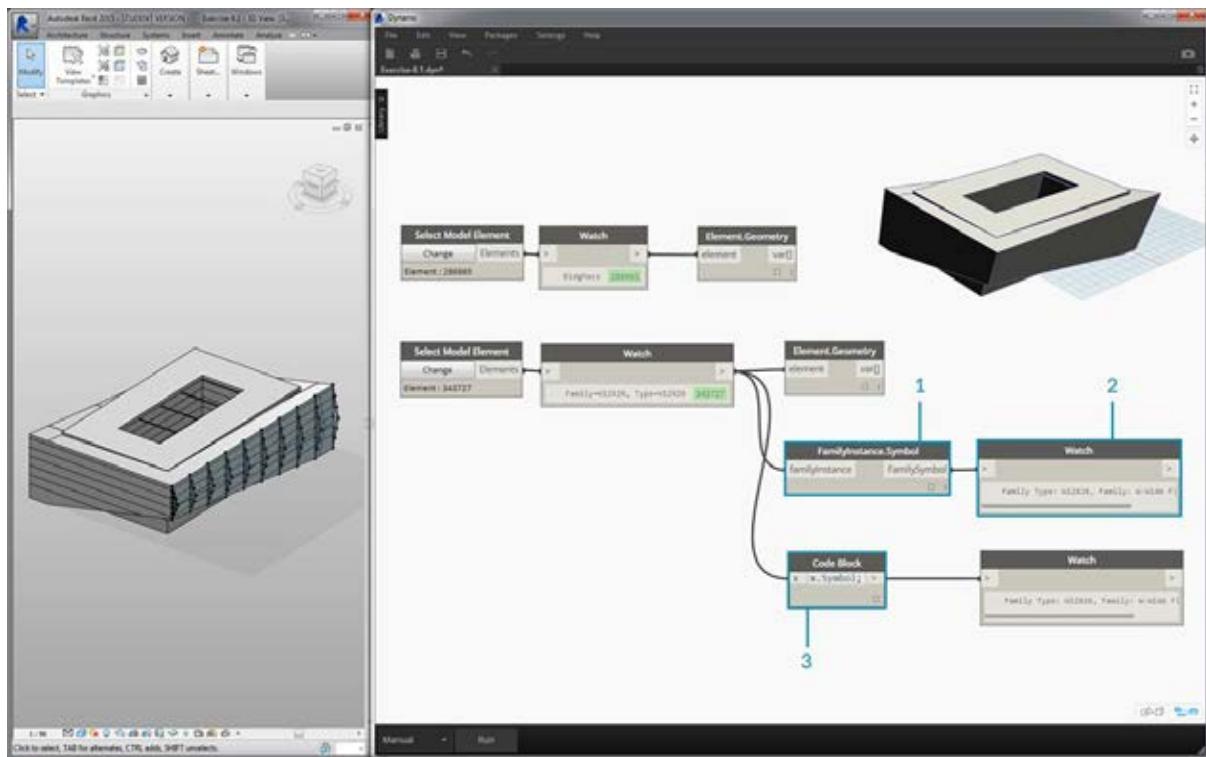
1. 4 つの壁面を選択した後で、必ず Revit で[終了]ボタンをクリックしてください。



1. これで、面が Dynamo にサーフェスとして読み込まれました。



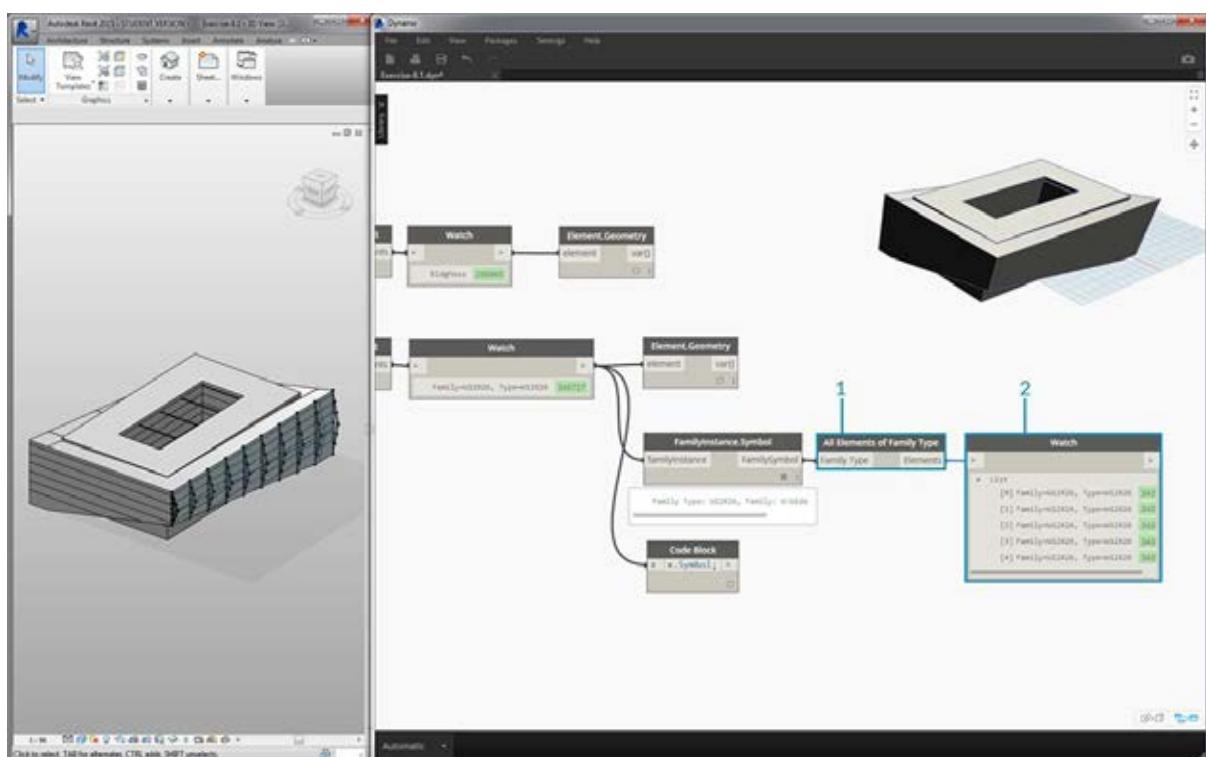
1. さて、アトリウムの上の梁を見てみましょう。Select Model Element ノードを使用して、梁のうち 1 つを選択します。
2. 梁の要素を Element.Geometry ノードに接続すると、Dynamo のビューポートで梁が表示されるようになります。
3. Watch3D ノードを使用してジオメトリを拡大表示することができます(Watch 3D で梁が表示されない場合は、右クリックしてから[全体表示]を選択します)。



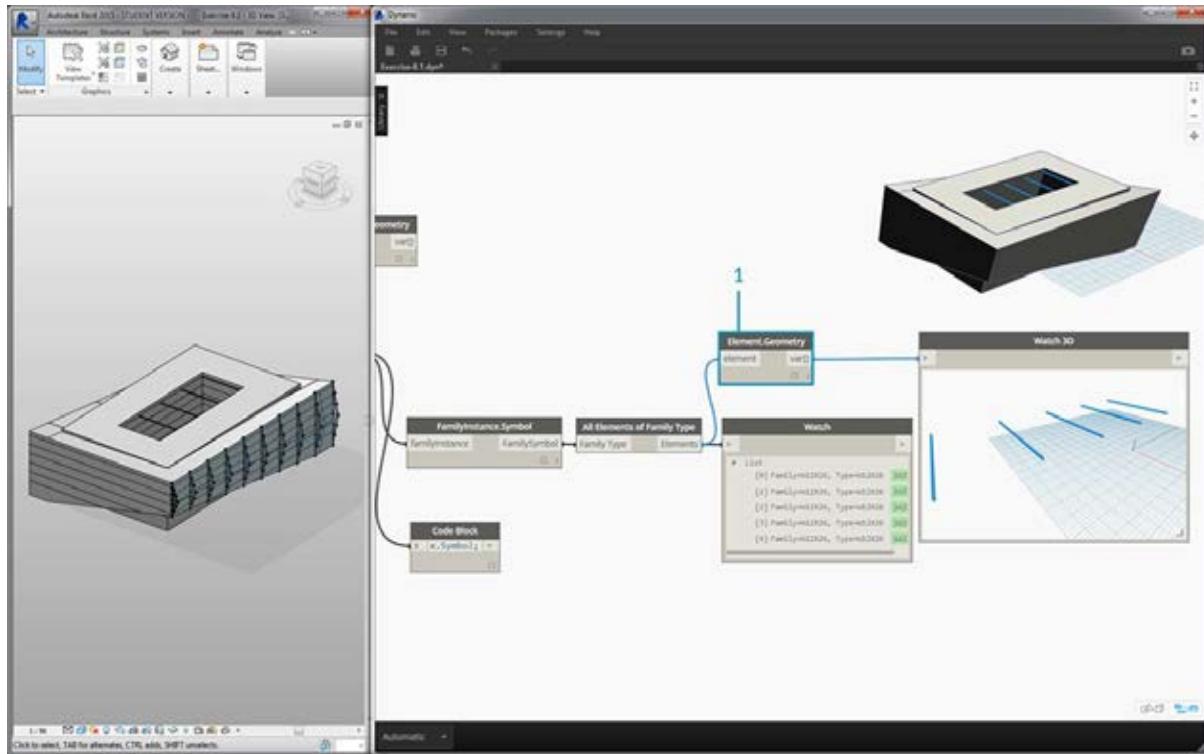
Revit と Dynamo の使用中にしばしば生じる疑問として、1 つの要素を選択して、それに類似するすべての要素を取得するにはどうすればいいのか、というものがあります。選択した Revit 要素にはその要素の階層に関する情報がすべて含まれているので、そのファミリ タイプをクエリーして、同じタイプの要素をすべて選択することができます。

1. 梁の要素を *FamilyInstance.Symbol* ノードに接続します。
2. *Watch* ノードで、出力が Revit 要素ではなくファミリ記号になっていることが確認できます。
3. *FamilyInstance.Symbol* は単純なクエリーですから、コード ブロック内で簡単に *x.Symbol* と入力して同様の結果を得ることができます。

注 - ファミリ記号は **Revit API** でファミリ タイプを指す用語です。この用語は混乱を引き起こしかねないため、今後のリリースで変更される可能性があります。



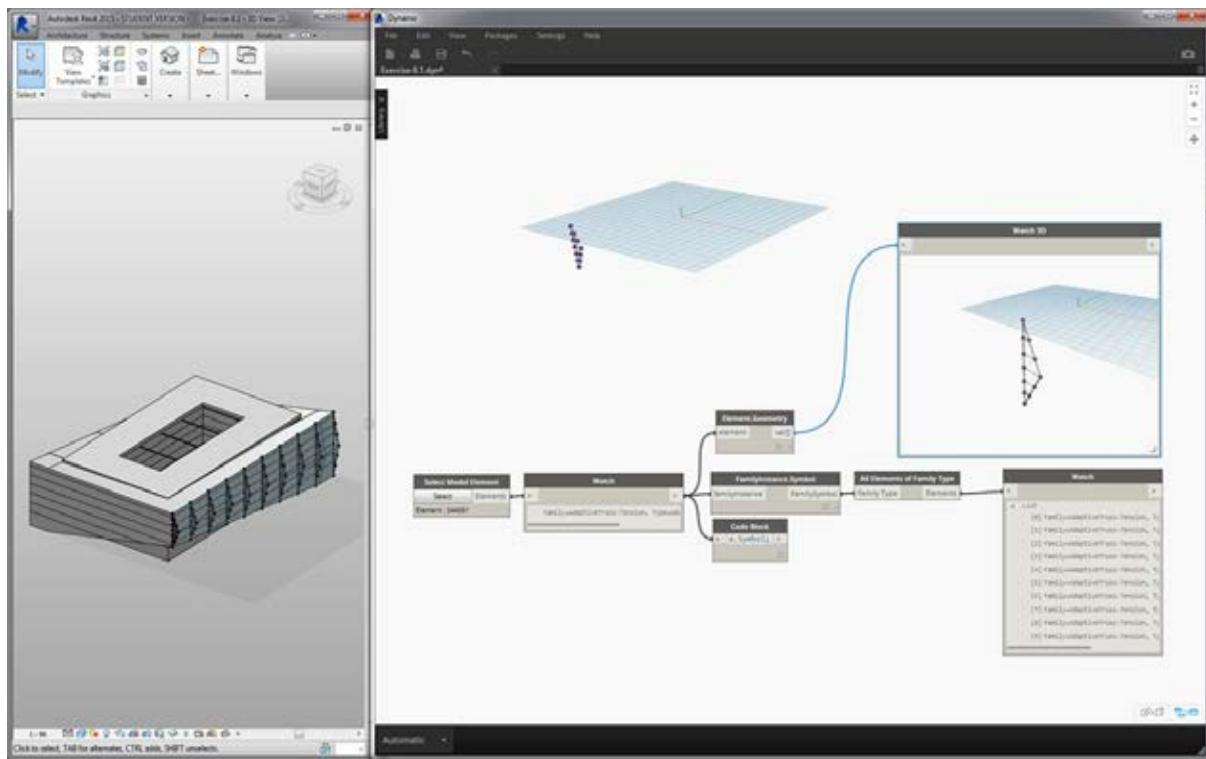
1. 残りの梁を選択するには、All Elements of Family Type ノードを使用します。
2. Watch ノードで、5 つの Revit 要素が選択されていることを確認します。



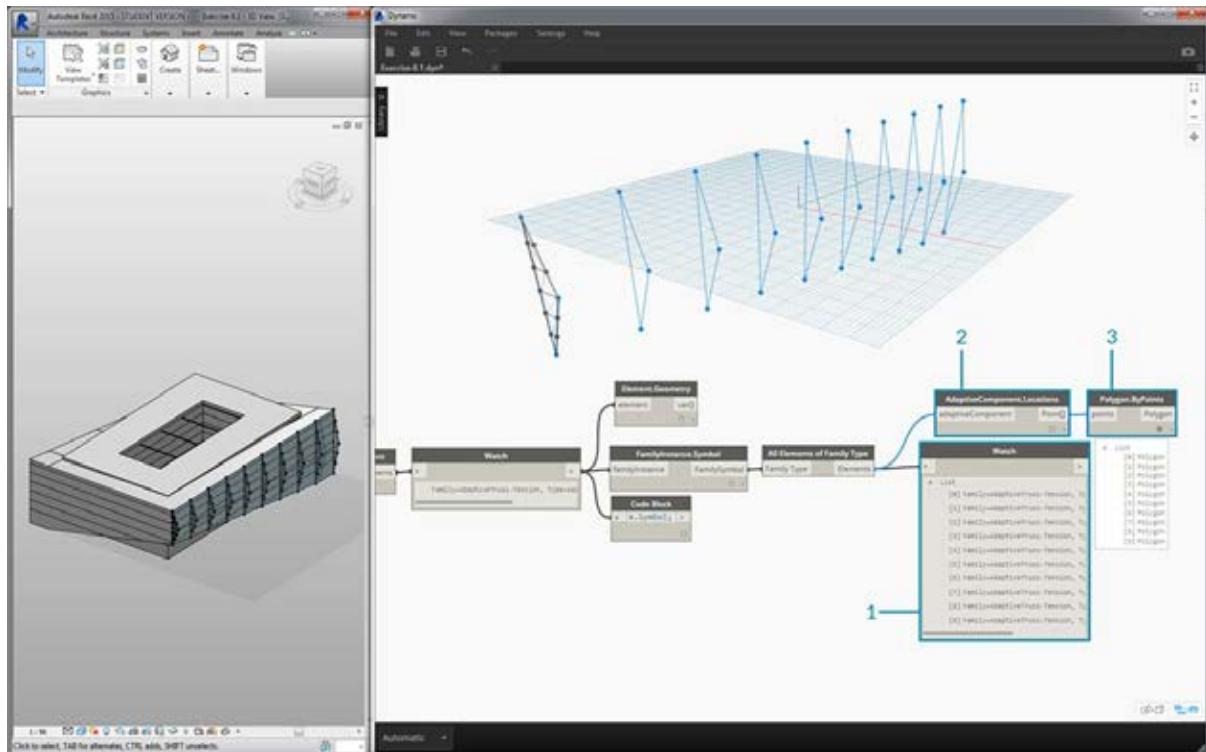
1. これら 5 つの要素すべてを Dynamo のジオメトリに変換することもできます。

しかし、500 本の梁を変換しなければならないとしたら、どうでしょうか。すべての要素を Dynamo ジオメトリに変換するにはたいへんな時間がかかるでしょう。Dynamo でノードの計算に膨大な時間がかかる場合は、ノードを「フリーズ」する機能を使用して、グラフの開発中に Revit 関連操作の実行を停止することができます。ノードをフリーズする操作の詳細については、「ソリッド」の章の「フリーズ」セクションを参照してください。

いずれにせよ、500 本の梁を読み込まなければならない場合、目的のパラメータ操作を実行するのにすべてのサーフェスは必要あるでしょうか。それとも、梁から基本情報を抽出して、基本的なジオメトリを使用して生成タスクを実行すればよいでしょうか。この問い合わせを念頭に置きながら、この章での演習を進めていくことにしましょう。たとえば、トラスシステムのことを考えてみてください。



同じノードのグラフを使用して、梁要素ではなくトラス要素を選択します。この操作を行う前に、ここまで手順で使用した Element.Geometry を削除してください。



1. *Watch* ノードで、Revit から選択されたアダプティブコンポーネントのリストを取得していることを確認できます。基本情報を抽出するために、まずはアダプティブ点からとります。
2. *All Elements of Family Type* ノードを *AdaptiveComponent.Location* ノードに接続します。これによってリストのリストが 1 つ作成されます。各リストは、アダプティブ点の場所を表す 3 つの点から構成されています。
3. *Polygon.ByPoints* ノードを接続すると、ポリカーブが返されます。これは Dynamo のビューポートで確認できます。この方法により、1 つの要素のジオメトリを表示し、残りの要素配列のジオメトリを抽出しました(なお、ここで扱った例よりも多くの要素を抽出することが可能です)。

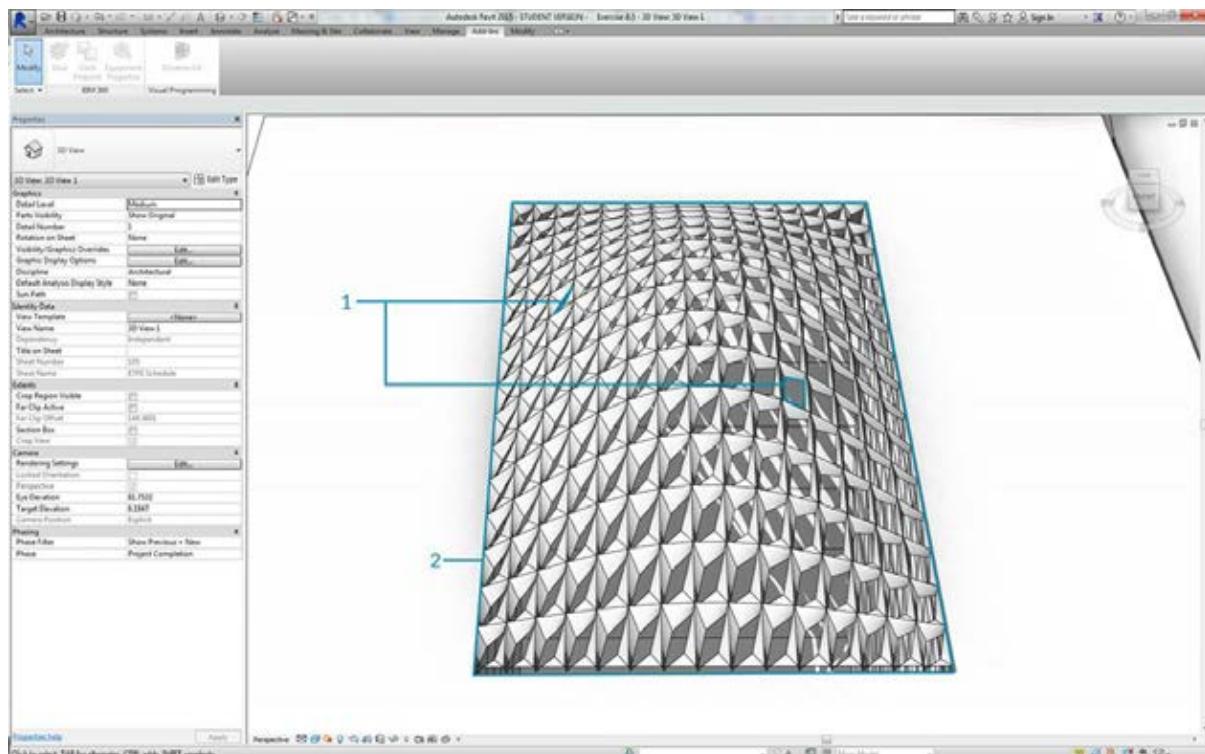
\*ヒント: *Dynamo* で、*Revit* 要素の緑の背景色付きで表示されている数字をクリックすると、*Revit* のビューポートでその要素が拡大表示されます。

# 編集

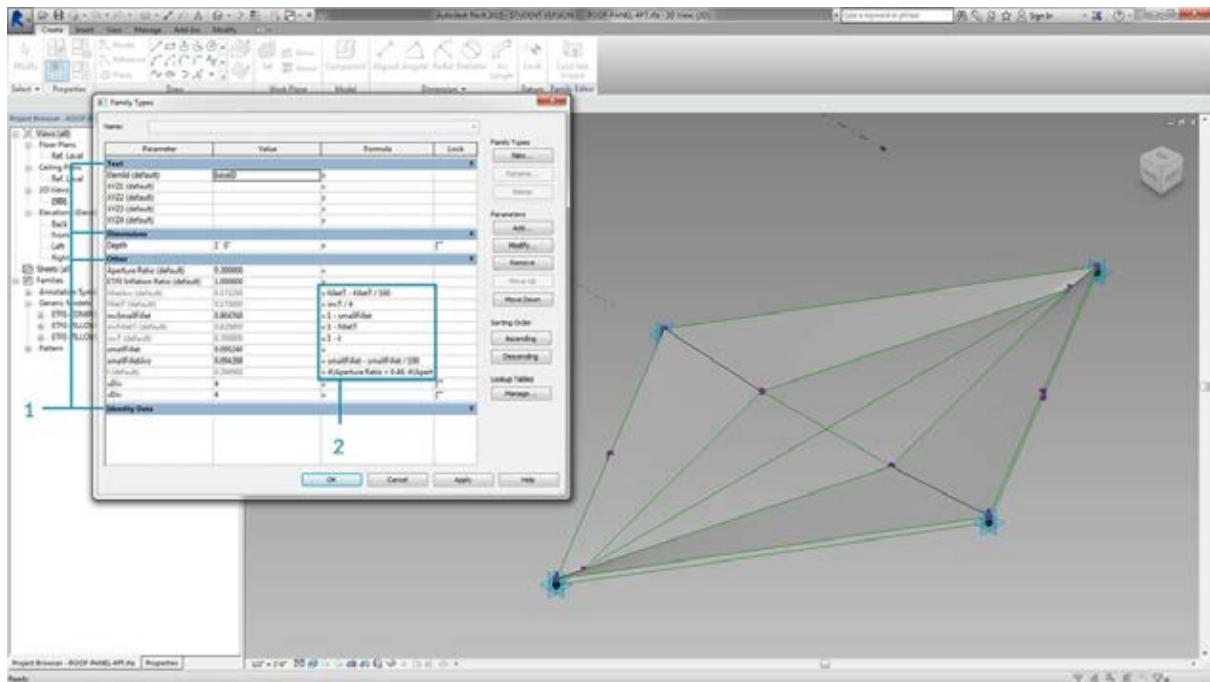
## 編集

Dynamo には、パラメトリック レベルでパラメータを編集するための強力な機能が用意されています。たとえば、生成アルゴリズムやシミュレーションの結果を使用して、要素の配列のパラメータをコントロールすることができます。この方法で、同じファミリのインスタンスの集合に、Revit プロジェクトのカスタム プロパティを設定することができます。

### タイプ パラメータとインスタンス パラメータ



1. インスタンス パラメータは、0.1 ~ 0.4 の開口率で、屋根サーフェス上のパネルの開口部を定義します。
2. タイプベースのパラメータは、サーフェス上のすべての要素に適用されます。これらの要素は同じファミリ タイプであるためです。たとえば、各パネルのマテリアルをタイプベースのパラメータによってコントロールすることができます。



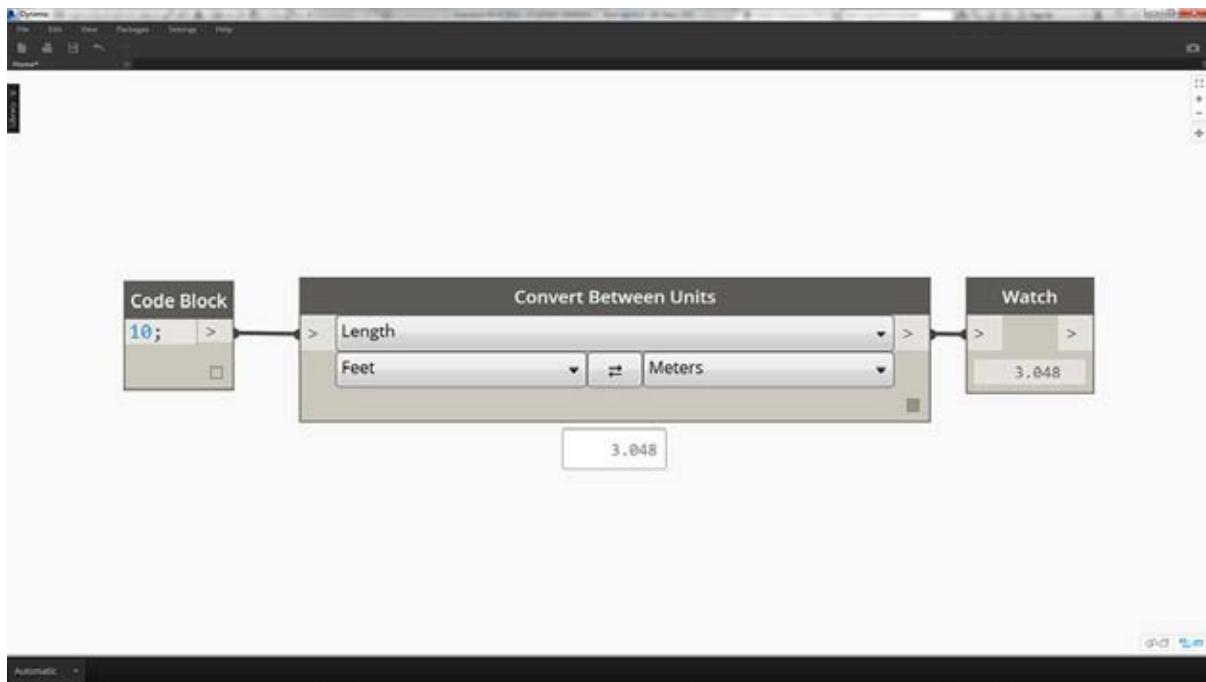
- これまでに Revit ファミリを使用したことがあれば、文字列、数値、寸法などのパラメータ タイプを指定する必要があることはわかるでしょう。Dynamo からパラメータを割り当てる場合は、必ず正しいデータ タイプを使用してください。
- また、Revit ファミリのプロパティで定義されたパラメトリック拘束と組み合わせて Dynamo を使用することもできます。

Revit のパラメータの簡単な復習として、タイプ パラメータとインスタンス パラメータの 2 種類があることを思い出してください。両方とも Dynamo で編集できますが、次の演習ではインスタンス パラメータを使用します。

**注:** 編集するパラメータの用途は幅広いため、Revit と Dynamo を併用すると、多くの要素を編集することができます。これは計算量が多い演算であるため、速度が低下することがあります。多くの要素を編集する場合は、ノードを「フリーズ」する機能を使用して、グラフの開発中に Revit に関連する操作の実行を停止することをお勧めします。ノードをフリーズする操作の詳細については、[「ソリッド」の章](#)の「フリーズ」セクションを参照してください。

## 単位

バージョン 0.8 以降、基本的に Dynamo では単位が使用されなくなりました。これにより、抽象的なビジュアル プログラミング環境が実現します。Revit の寸法を使用する Dynamo のノードは、Revit プロジェクトの単位を参照します。たとえば、Revit の長さパラメータを Dynamo で設定する場合、Dynamo の数値は Revit プロジェクトの既定の単位に対応します。次の演習では、メートル単位の数値を操作します。



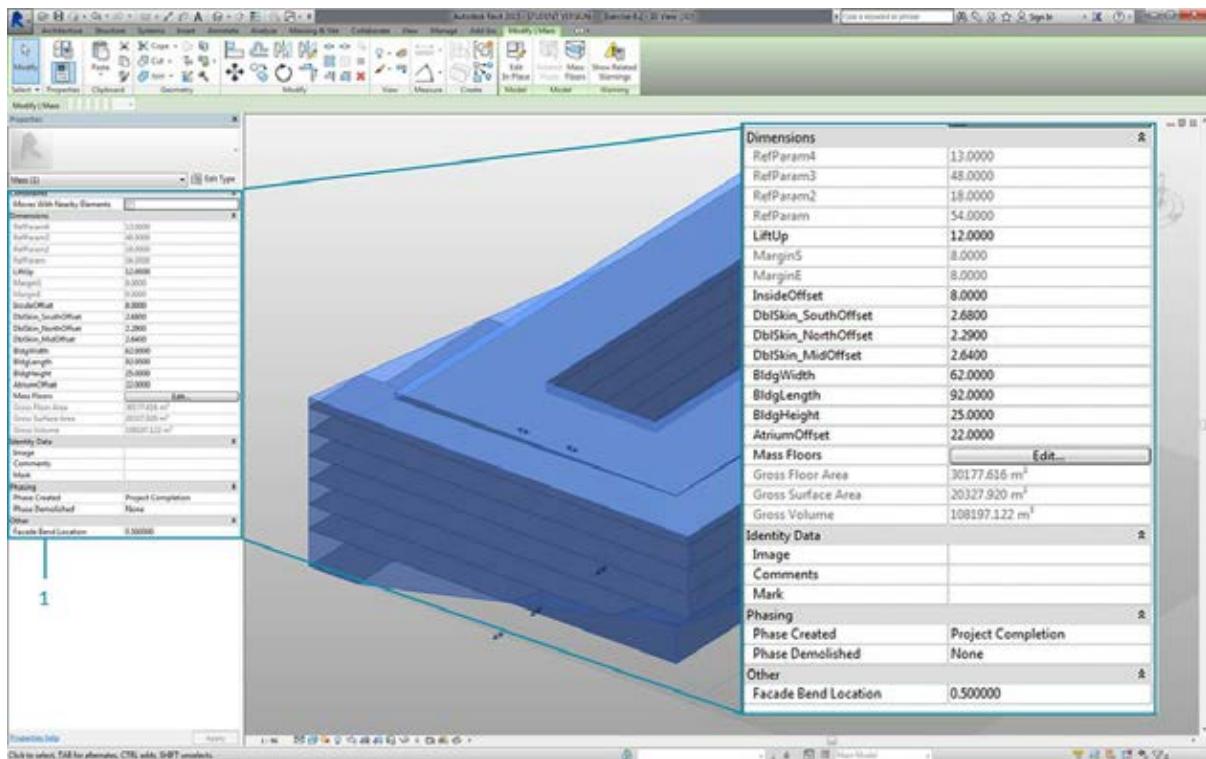
単位を素早く変換するには、*Convert Between Units* ノードを使用します。このノードは、長さ、面積、体積の単位をその場で変換できる便利なツールです。

## 演習

この演習に付属しているサンプル ファイルをダウンロードしてください(右クリックして[名前を付けてリンク先を保存]を選択)。すべてのサンプルファイルの一覧については、付録を参照してください。

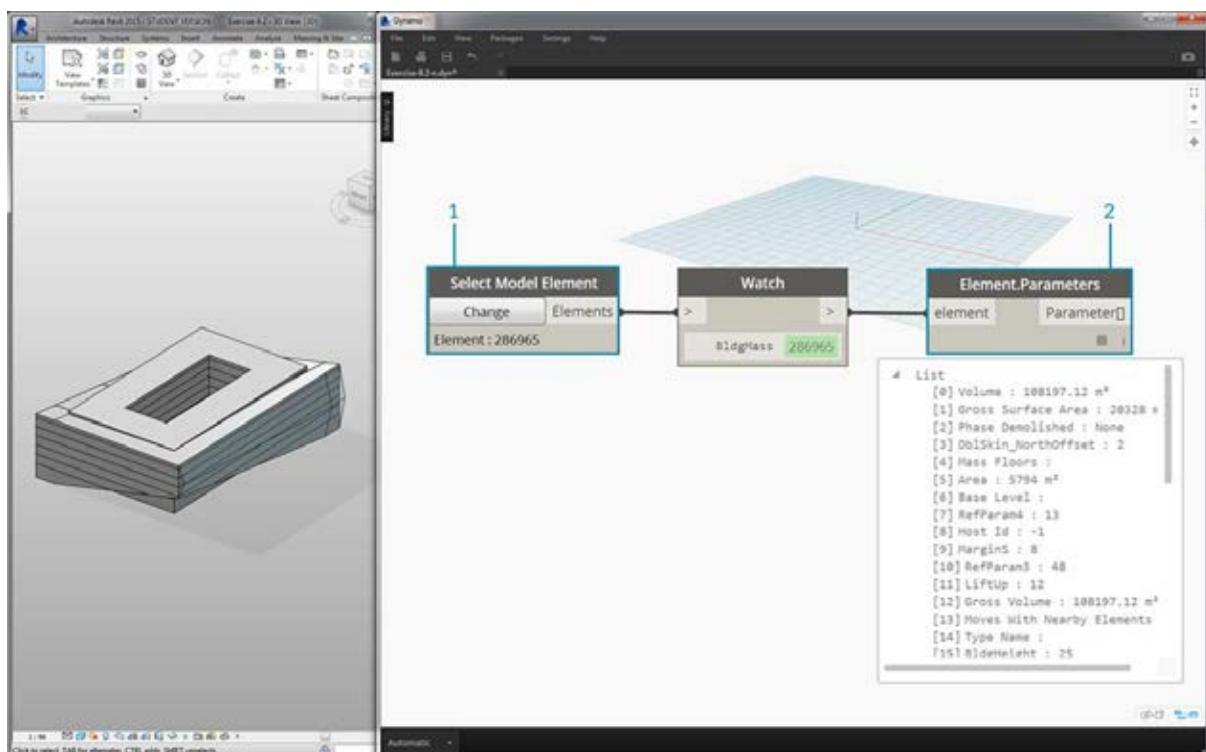
1. [Editing.dyn](#)
2. [ARCH-Editing-BaseFile.rvt](#)

この演習では、Dynamo でジオメトリ操作を実行することなく Revit 要素を編集します。Dynamo ジオメトリを読み込みます、Revit プロジェクトで直接パラメータを編集します。これは基本的な演習です。Revit の上級ユーザであれば、次の図のパラメータはマスのインスタンス パラメータであることがわかるでしょう。同じロジックを要素の配列に適用し、大規模なカスタマイズを行うことができます。これは、すべて Element.SetParameterByName ノードで行います。

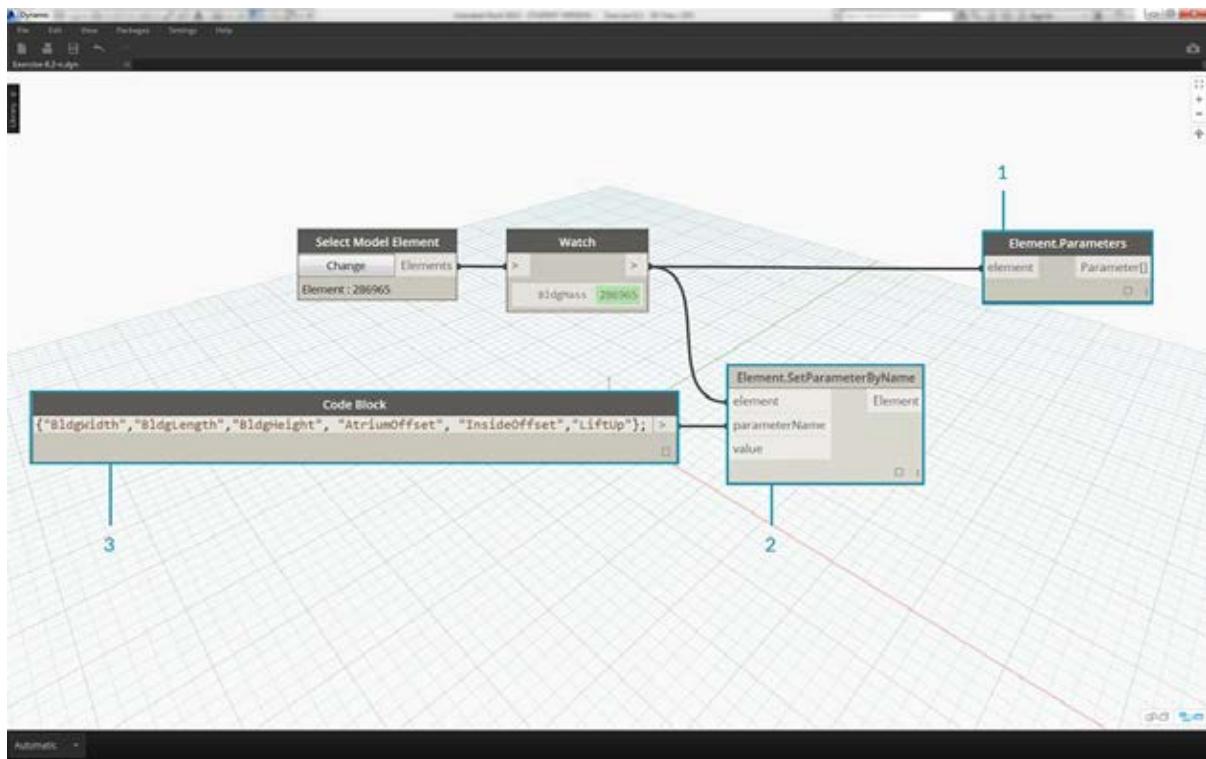


このセクションでは、最初にサンプルの Revit ファイルを使用します。構造要素とアダプティブ ト拉斯は、前のセクションで削除されました。この演習では、Revit のパラメータ機能を確認しながら、Dynamo で操作を行います。

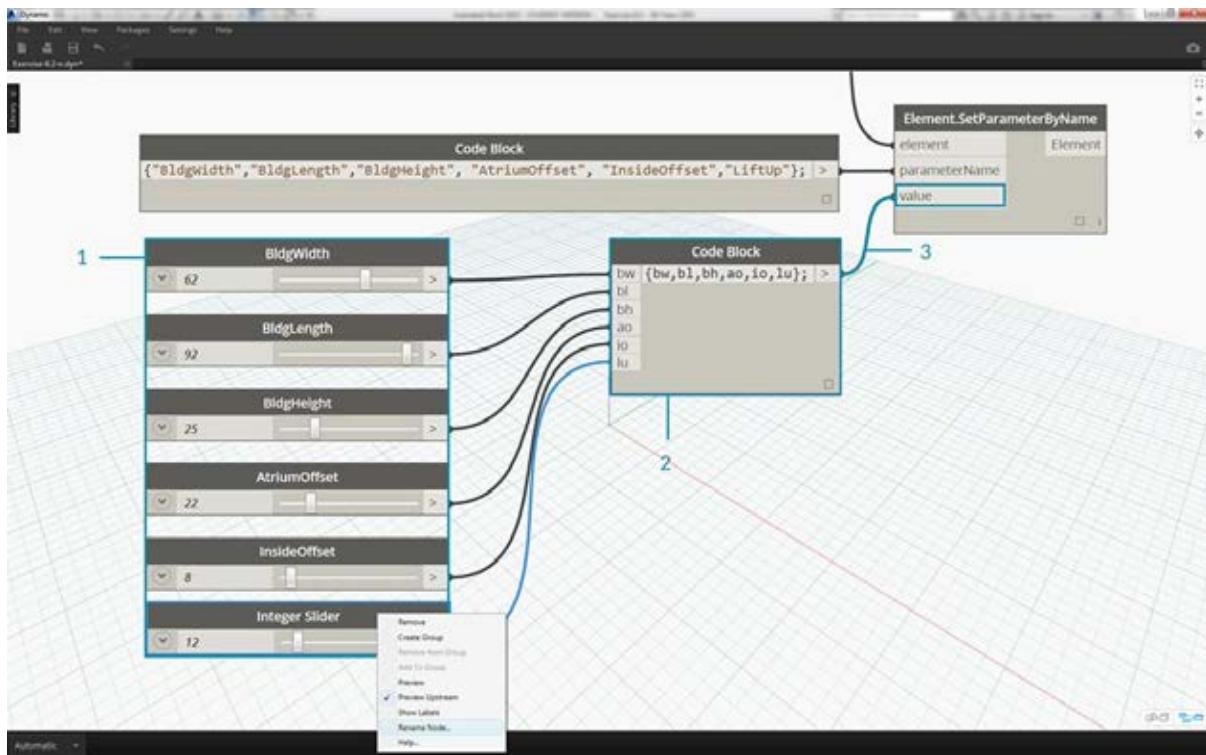
1. Revit で建物のマスを選択すると、プロパティ パネルにインスタンス パラメータの配列が表示されます。



1. *Select Model Element* ノードを使用して、建物のマスを選択します。
2. *Element.Parameters* ノードを使用して、このマスのすべてのパラメータのクエリーを実行することができます。パラメータには、タイプ パラメータとインスタンス パラメータがあります。



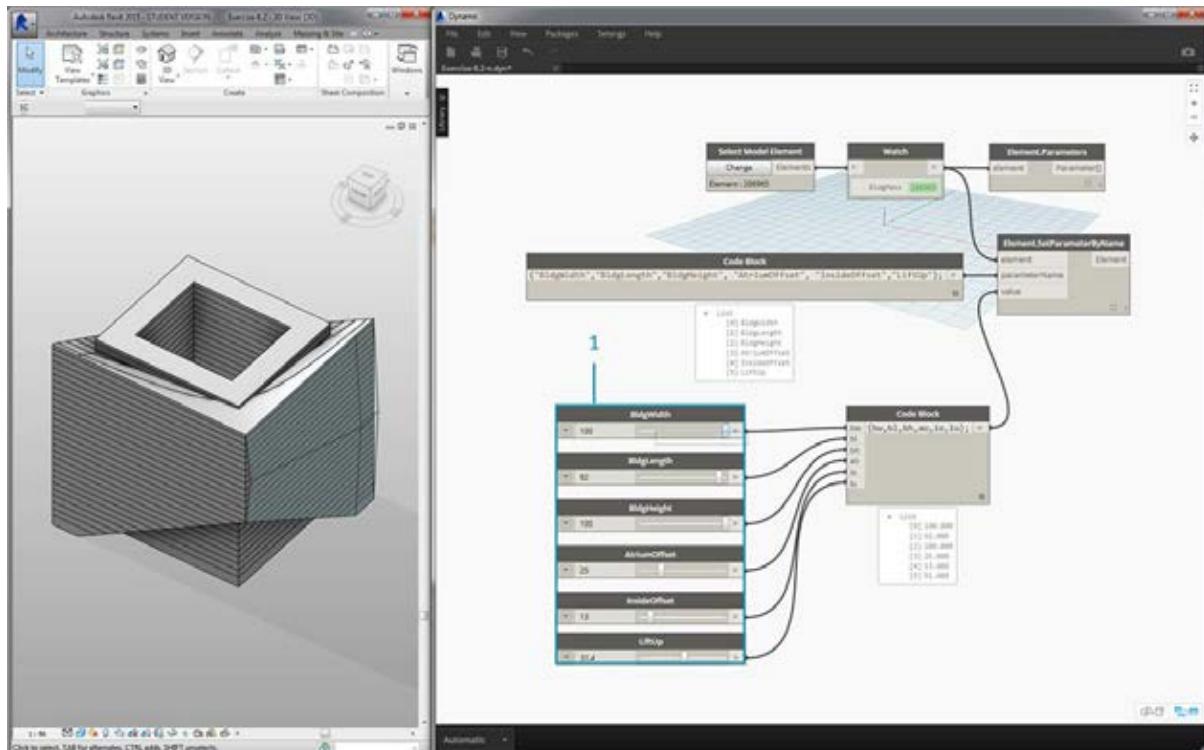
1. *Element.Parameters* ノードを参照して、対象のパラメータを探します。または、前の手順のプロパティパネルを表示して、編集するパラメータ名を選択することもできます。この場合、建物のマスの形状に大きく影響するパラメータを探す必要があります。
2. *Element.SetParameterByName* ノードを使用して、Revit 要素に変更を加えます。
3. *Code Block* ノードを使用して、対象のパラメータのリストを定義します。その際、各項目を引用符で囲み、その項目が文字列であることを指定します。List.Create ノードの複数の入力に string ノードを接続して使用することもできますが、Code Block ノードを使用する方が簡単で、時間もかかりません。文字列が、次のように大文字小文字を含めて Revit の正確な名前と一致していることを確認します。`{"BldgWidth", "BldgLength", "BldgHeight", "AtriumOffset", "InsideOffset", "LiftUp"}`



1. 各パラメータの値を指定します。*Integer Slider* ノードを 6 つキャンバスに追加し、リスト内のパラメータに合わせて名前

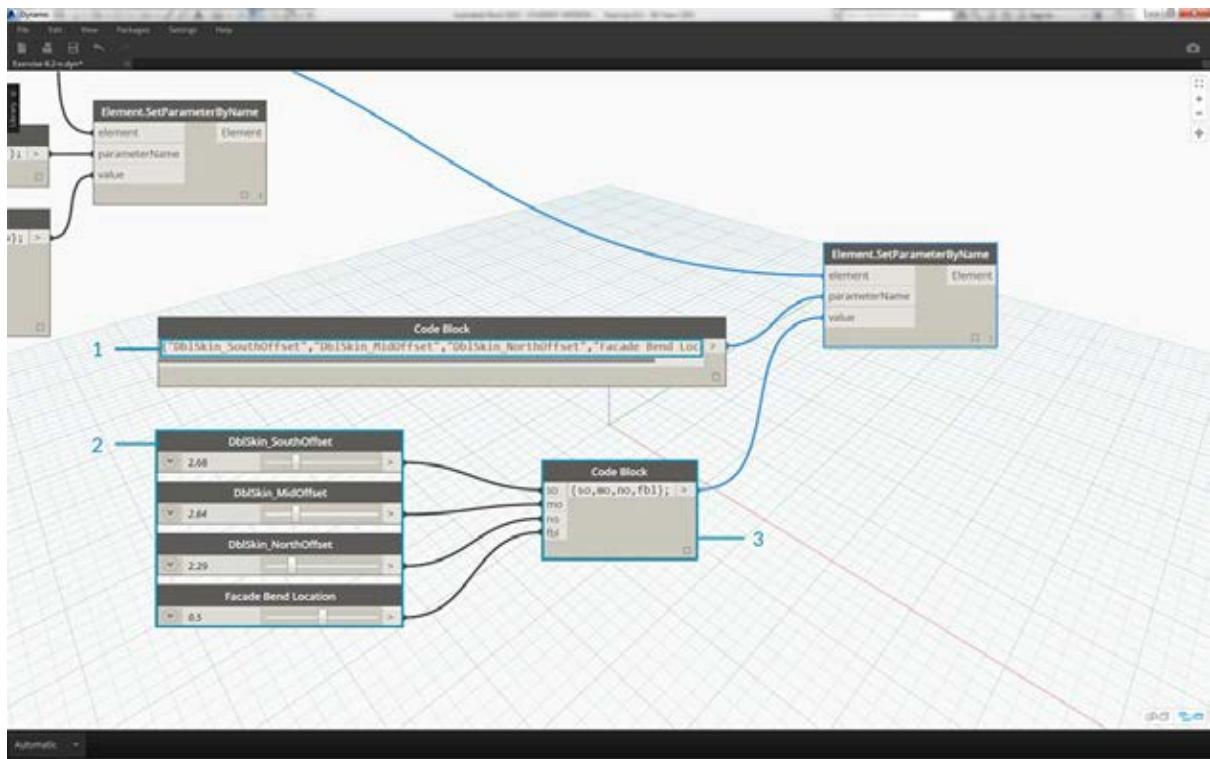
- を変更します。また、各スライダの値を上の図のように設定します。上から下に、62、92、25、22、8、12 の順で設定します。
2. パラメータ名の数と同じ長さのリストを使用して、別の *Code Block* ノードを定義します。その際、*Code Block* ノードの入力を作成する変数の名前を、引用符を使用せずに入力します。*\*\*Integer Slider* ノードを `{bw, bl, bh, ao, io, lu}`; の各入力に接続します。
  3. *Code Block* ノードを *Element.SetParameterByName* ノードに接続します。[自動実行]をオンにすると、結果が自動的に表示されます。

注: このデモンストレーションは、インスタンス パラメータには対応していますが、タイプ パラメータには対応していません。

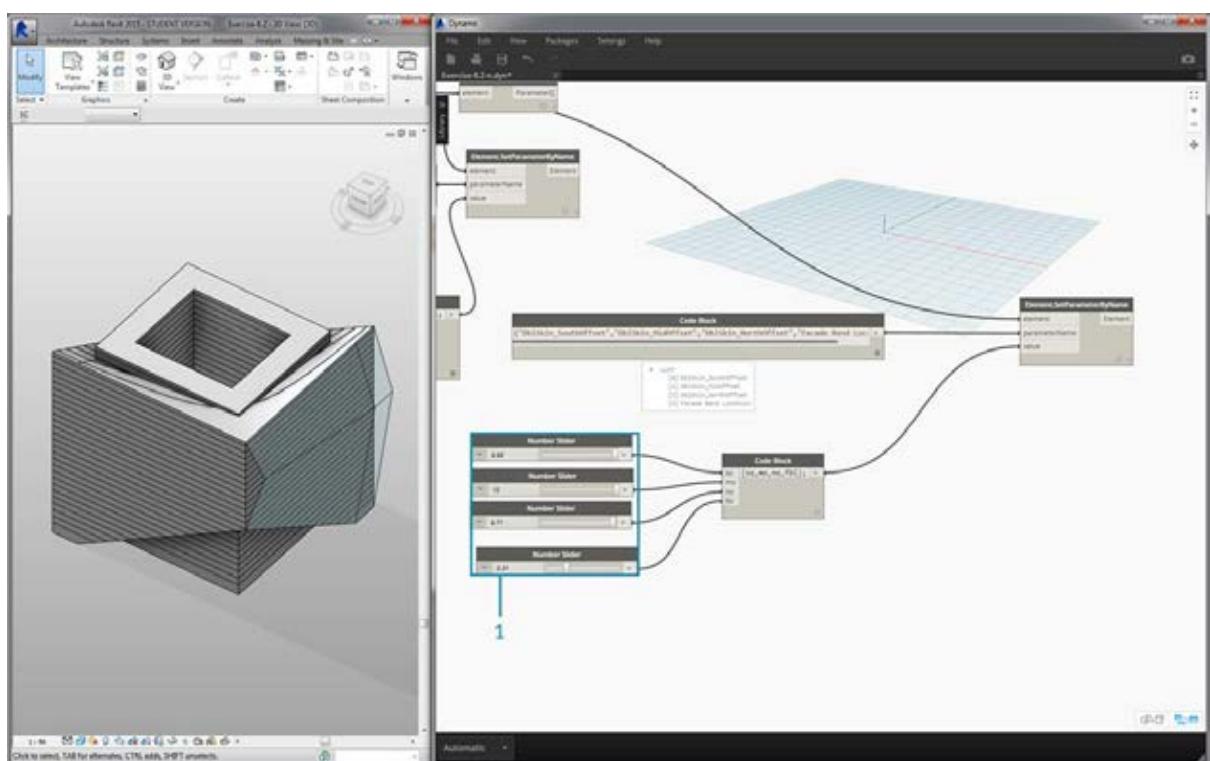


Revit と同様に、これらのパラメータの多くは相互に依存しています。これらの中には、ジオメトリが壊れてしまう組み合わせもあります。この問題を解決するには、定義済みの式をパラメータ プロパティで使用するか、Dynamo の数値演算で同様のロジックを設定します。余裕があれば、この演習の追加の課題として取り組んでみてください。

1. 100、92、100、25、13、51.4 という組み合わせにより、特徴的な新しいデザインが建物のマスに追加されます。



1. グラフをコピーして、トラスシステムを格納するファサードガラスを確認してみましょう。この場合、次の4つのパラメータを使用します。`{"DblSkin_SouthOffset", "DblSkin_MidOffset", "DblSkin_NorthOffset", "Facade Bend Location"}`;
2. また、Number Sliderノードを使用して、対応するパラメータに合わせて名前を変更します。上から3つのスライダは[0,10]の範囲に再マップし、一番下のスライダ(Facade Bend Location)は[0,1]の範囲に再マップします。次に、これらの値を2.68、2.64、2.29、0.5にそれぞれ設定します。
3. 新しいCode Blockノードを定義し、`{ so, mo, no, fbl }`；というコード行を指定して各スライダを結合します。



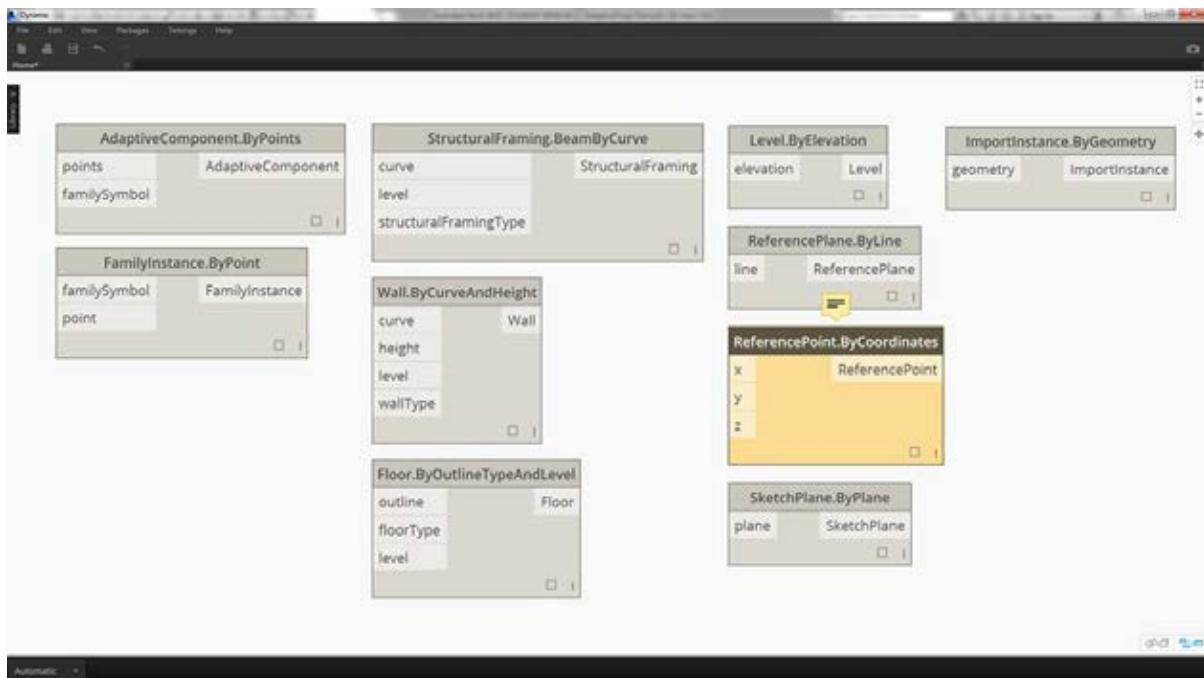
1. グラフのこの部分の各Number Sliderノードの値をそれぞれ9.98、10.0、9.71、0.31に変更すると、ファサード

ガラスがさらにがっしりとした形状になります。

# 作成

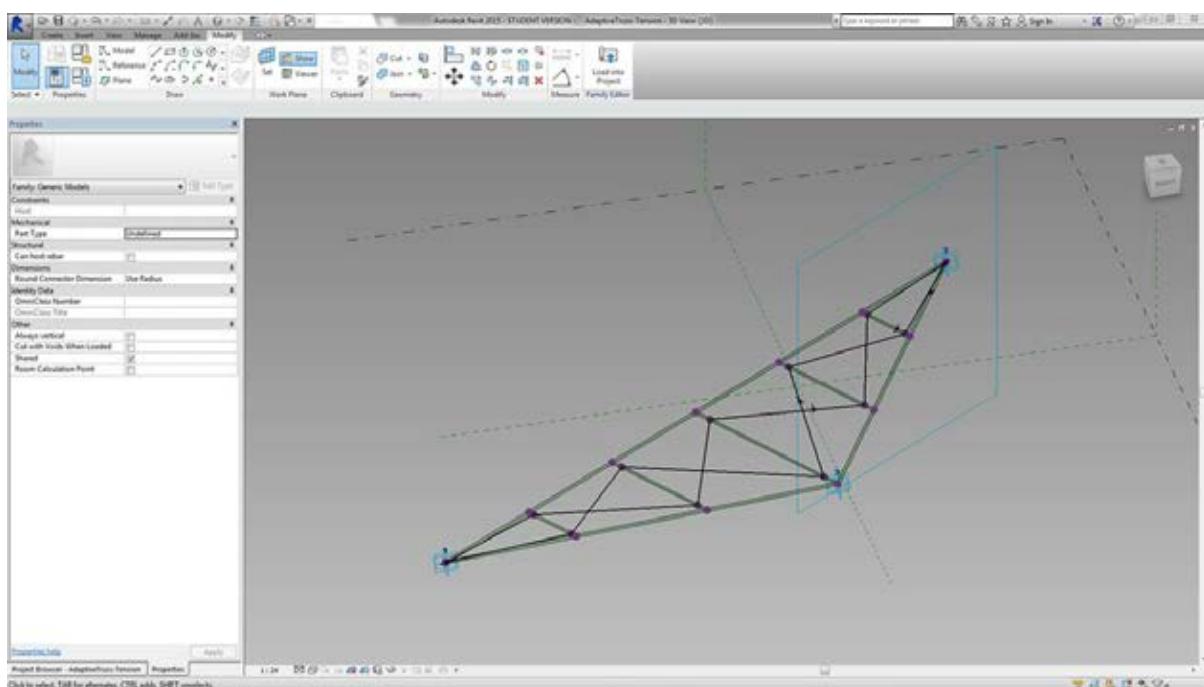
## 作成

Dynamo では、完全なパラメトリック コントロールを使用して Revit 要素の配列を作成できます。Dynamo の Revit ノードは、一般的なジオメトリから特定のカテゴリ タイプ(壁、床など)まで、さまざまな要素を読み込む機能を提供します。このセクションでは、パラメータを使用してアダプティブ コンポーネントを含む柔軟性が高い要素を読み込みます。



## アダプティブ コンポーネント

アダプティブ コンポーネントはオブジェクトの生成に役立つ柔軟性の高いファミリ カテゴリです。インスタンス化すると、アダプティブ 点の基本的な位置でコントロールされる複雑なジオメトリ要素を作成することができます。



これは、ファミリエディタ上で 3 つのアダプティブ点により構成されているアダプティブコンポーネントです。これにより生成されるトラスは、各アダプティブ点の位置によって設定されます。次の演習では、このコンポーネントを使用して、ファサード全体に一連のトラスを生成します。

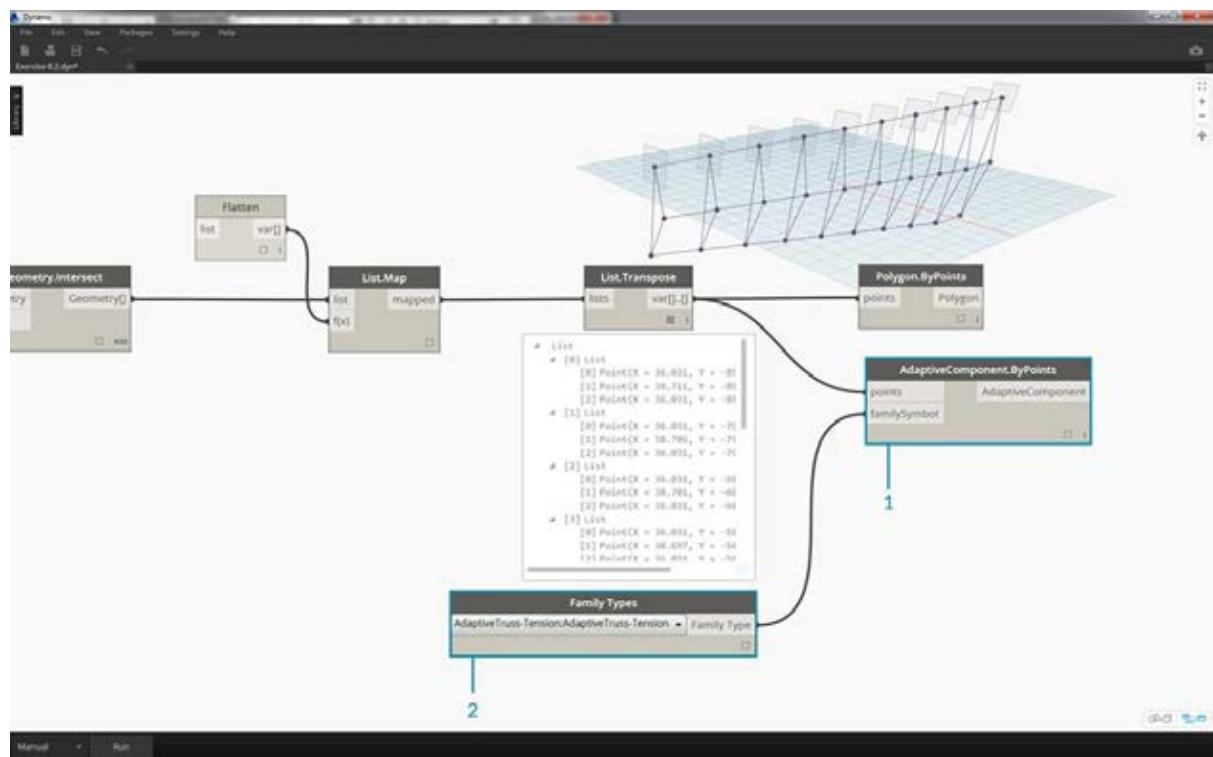
## 相互運用性の原則

アダプティブコンポーネントは相互運用性のベストプラクティスの好例です。基本的なアダプティブ点を設定することにより、アダプティブコンポーネントの配列を作成できます。また、このデータを他のプログラムに転送すると、ジオメトリを単純なデータに変換できます。Excel などのプログラムに読み込んだり書き出す場合も同様です。

ファサード設計の監修者が、完全なジオメトリを詳細に解析する必要なく、トラス要素の位置を確認する必要があるとします。製造の準備段階で、コンサルタントはアダプティブ点の位置を参照することにより、Inventor などのプログラムでジオメトリを再生成できます。

次の演習のワークフローでは、このようなデータすべてにアクセスしながら、Revit 要素を作成するための設定を行います。このプロセスにより、概念化、ドキュメント作成、製造を、1 つのシームレスなワークフローに統合できます。これにより、相互運用性を実現するためのよりインテリジェントで効率的なプロセスを作成できます。

## 複数の要素リスト

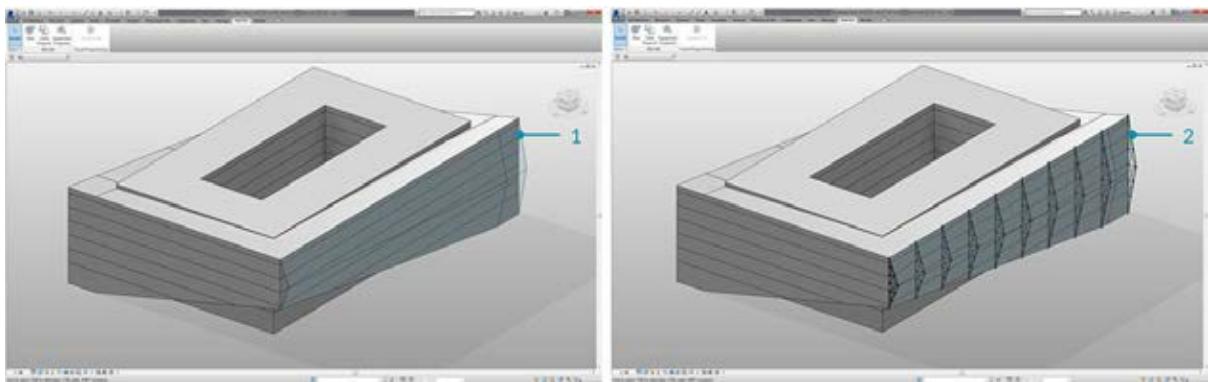


次の演習では、Revit 要素を作成するためのデータを Dynamo が参照する仕組みについて学習します。複数のアダプティブコンポーネントを生成するには、リストのリストを設定します。各リストには、アダプティブコンポーネントの各点を表す 3 つの点が含まれています。Dynamo でデータ構造を管理する際は、このことを考慮します。

## 演習

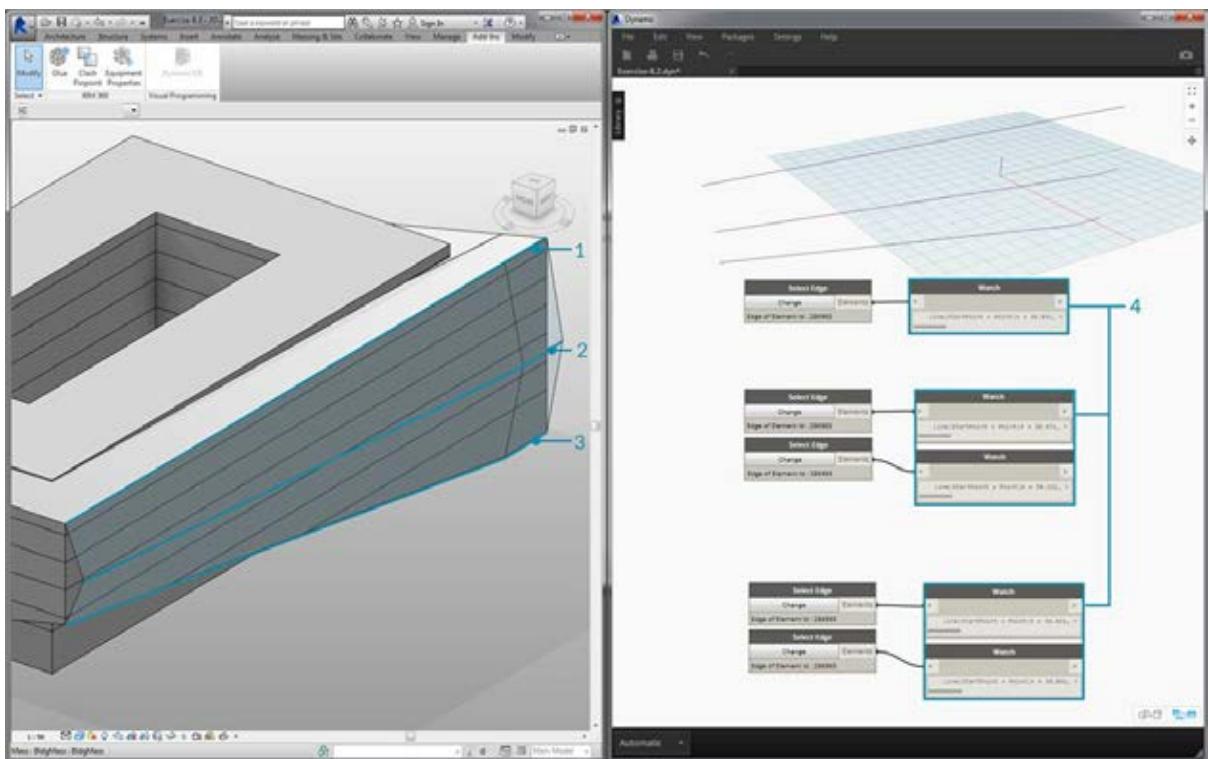
この演習に付属しているサンプルファイルをダウンロードしてください(右クリックして[名前を付けてリンク先を保存]を選択)。すべてのサンプルファイルの一覧については、付録を参照してください。

1. [Creating.dyn](#)
2. [ARCH-Creating-BaseFile.rvt](#)



このセクションでサンプル ファイルの使用を開始した場合(または、前のセッションの Revit ファイルを継続して使用した場合)は、同じ Revit のマスが表示されます。

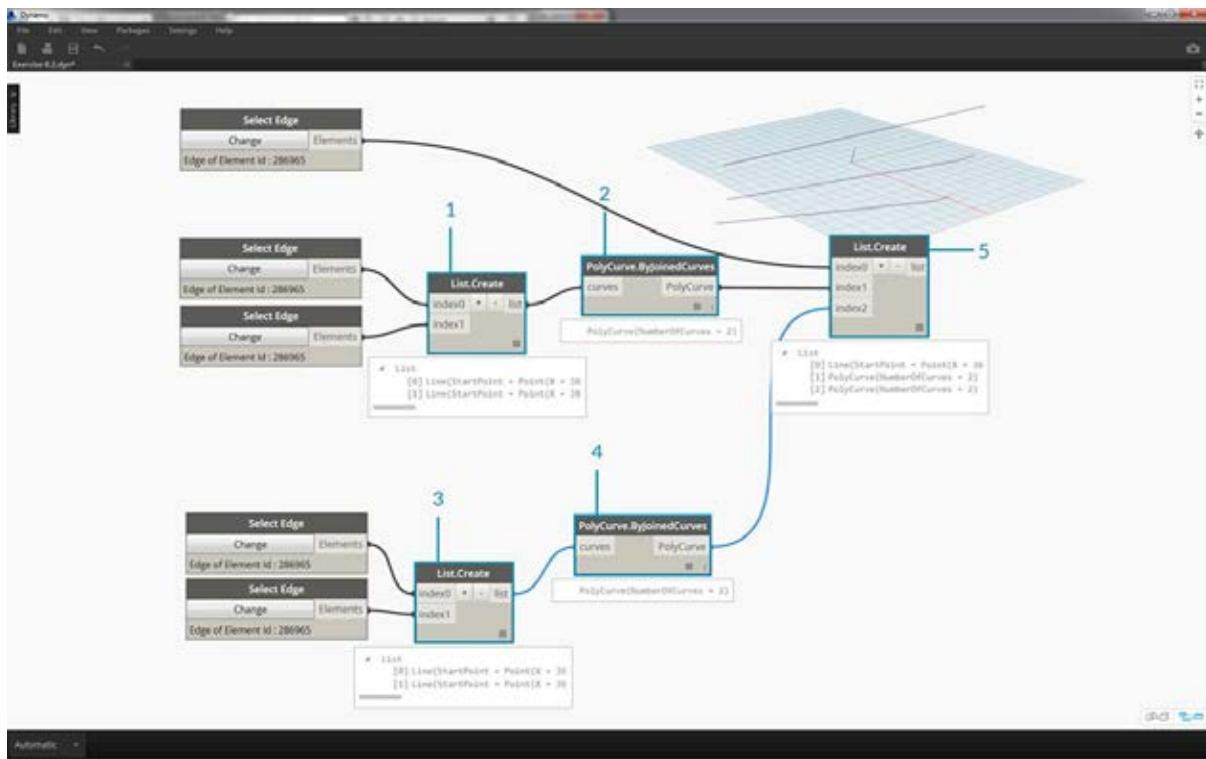
1. これはファイルを開いた状態です。
2. これは、Dynamo で作成したトラス システムです。高度な方法で Revit のマスにリンクされています。



これまで *Select Model Element* ノードと *Select Face* ノードを使用しました。ここでは、ジオメトリ階層の 1 段階下の層で *Select Edge* ノードを使用します。Dynamo ソルバを[自動]で実行するように設定すると、グラフは Revit ファイルの変更に応じて継続的に更新されます。選択したエッジは Revit 要素トポロジに動的に関連付けられます。トポロジが変更されない限り、Revit と Dynamo 間の接続はリンクされ続けます。

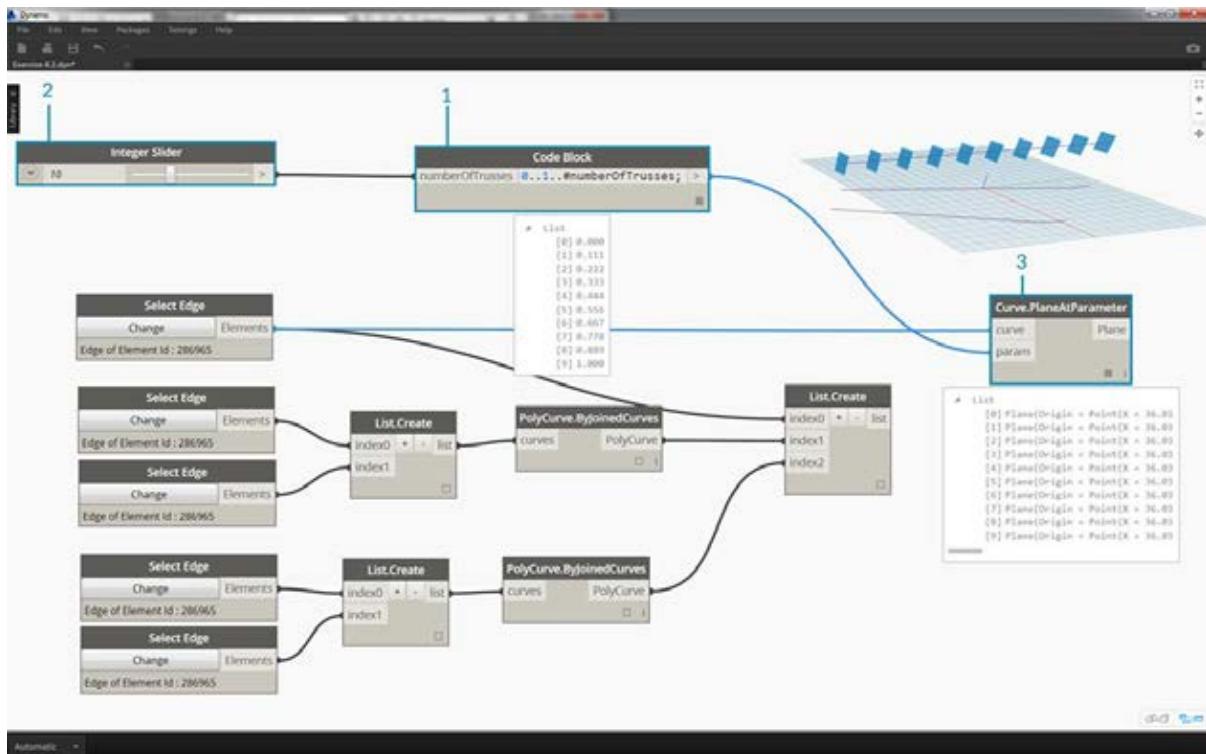
1. グレーディング フサードの最上部の曲線を選択します。これは建物の全体の長さに及びます。Revit でエッジを選択するには、エッジにカーソルを置いて、目的のエッジがハイライト表示されるまで [Tab] を押し続けます。
2. 2 つの *Select Edge* ノードを使用して、フサード中央の傾斜を示す各エッジを選択します。
3. Revit でフサードの最下部のエッジに対して同じ操作を行います。
4. Dynamo に線が設定されたことが *Watch* ノードによって示されます。エッジ自体は Revit 要素ではないため、自動的に Dynamo ジオメトリに変換されます。これらの曲線は、フサード全体にわたるアダプティブ トラスをインスタンス化する際に使用する参照です。

\*注意: トポロジの一貫性を保持するため、面やエッジが追加されないモデルを参照します。パラメータを使用して形状を変更することはできますが、トポロジの作成方法を変更することはできません。



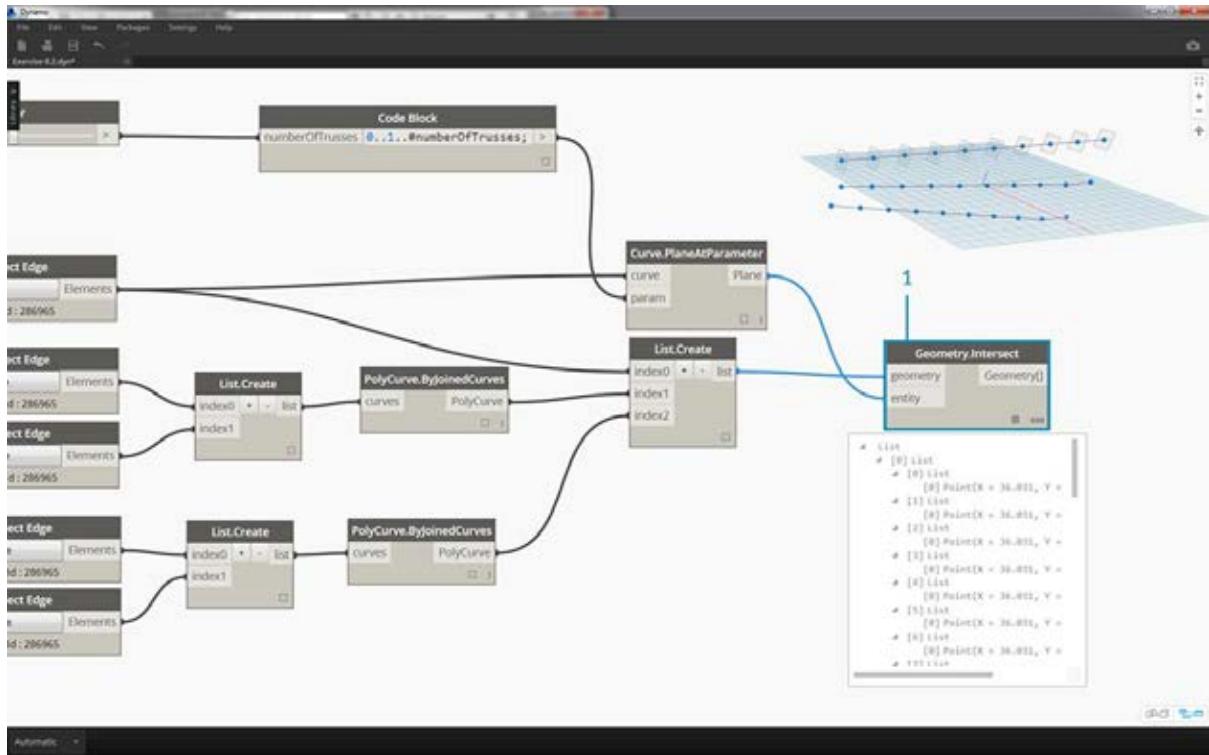
まず曲線を結合して 1 つのリストに統合する必要があります。これにより、曲線を「グループ化」して、ジオメトリ操作を実行できます。

1. ファサードの中央にある 2 つの曲線のリストを作成します。
2. *List.Create* コンポーネントを *Polycurve.ByJoinedCurves* ノードに接続して、2 つの曲線を 1 つのポリカーブに結合します。
3. ファサードの最下部にある 2 つの曲線のリストを作成します。
4. *List.Create* コンポーネントを *Polycurve.ByJoinedCurves* ノードに接続して、2 つの曲線を 1 つのポリカーブに結合します。
5. 最後に、3 つの主要な曲線(1 つの直線と 2 つのポリカーブ)を 1 つのリストに結合します。



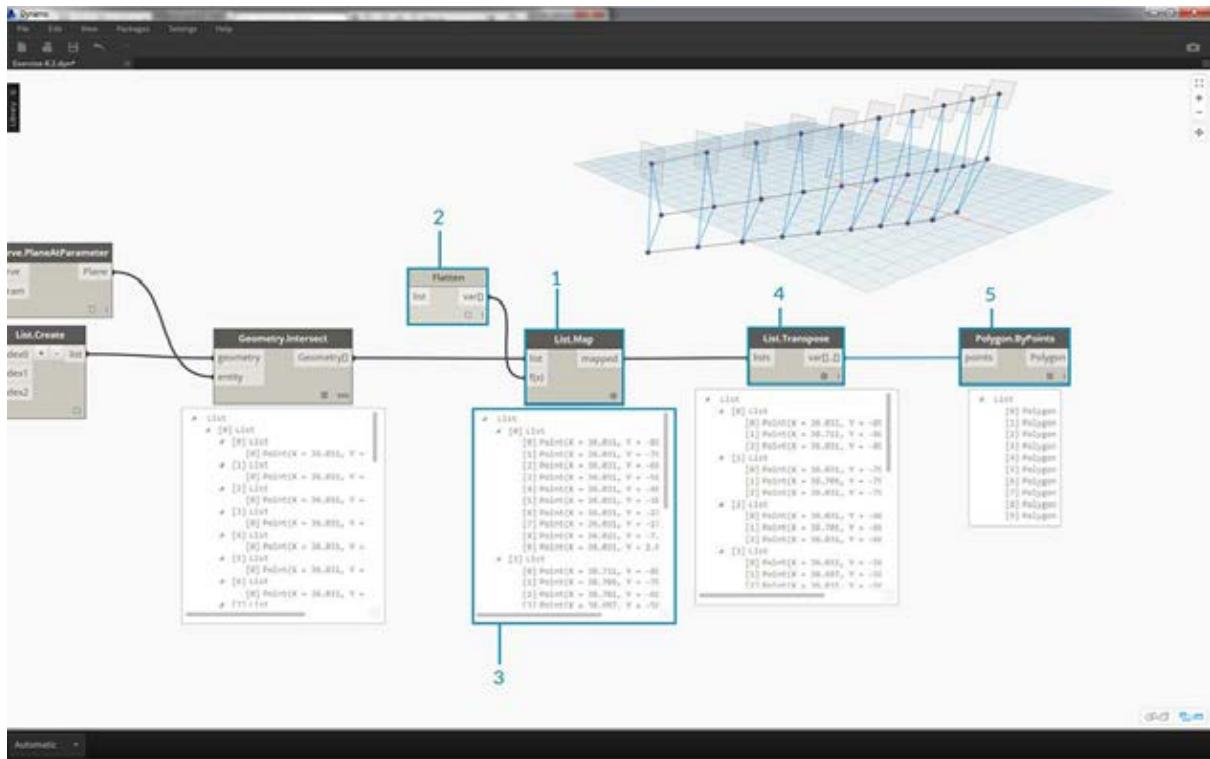
直線になっている最上部の線を使用して、ファサードのスパン全体を表します。この線に沿って平面を作成し、リストでグループ化した曲線のセットと交差させます。

1. *Code Block* ノードで、構文 `0..1..#numberOfTrusses;` を使用して範囲を設定します。
2. *Integer Slider* ノードを *Code Block* ノードの入力に接続します。おわかりのとおり、これはトラスの数を表します。スライダーは項目の数を 0 から 1 の範囲でコントロールします。
3. *Code Block* ノードを *Curve.PlaneAtParameter* ノードの *param* 入力に接続し、最上部のエッジを *curve* 入力に接続します。これにより、10 個の平面がファサードのスパン全体にわたって均等に配置されます。



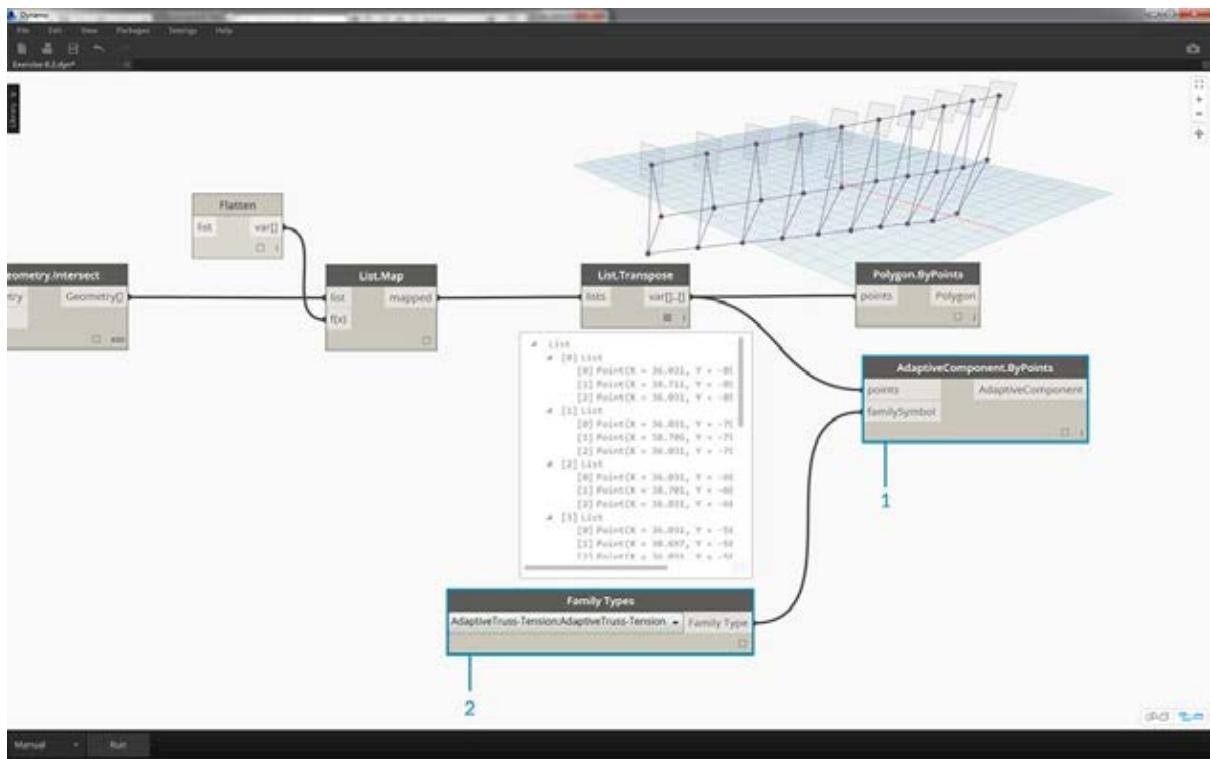
平面はジオメトリの抽象的なピースであり、無限の 2 次元空間を表します。平面は輪郭や交差の作成に適しています。実際に行ってみましょう。

1. *Geometry.Intersect* ノードを使用して、*Curve.PlaneAtParameter* を *Geometry.Intersect* ノードの *entity* 入力に接続します。メインの *List.Create* ノードを *geometry* 入力に接続します。Dynamo のビューポートには、設定した平面と各曲線の交点が表示されます。



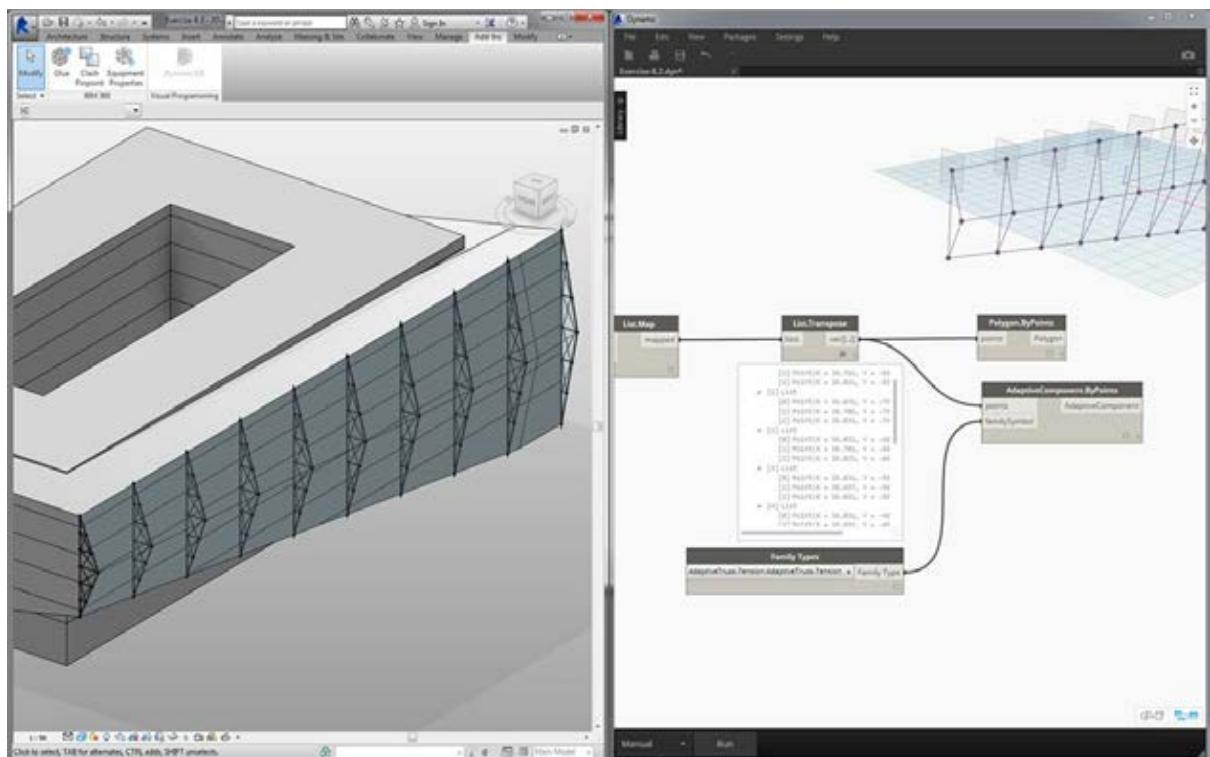
出力にはリストのリストのリストが表示されます。操作目的に対してリストの数が多すぎます。リストの一部をフラットにしましょう。リストの 1 段階下の層で結果をフラットにします。これを行うには、手引のリストに関する章で説明したように、*List.Map* 操作を使用します。

1. *Geometry.Intersect* ノードを *List.Map* の list 入力に接続します。
2. *Flatten* ノードを *List.Map* ノードの f(x) 入力に接続します。この結果、リストは 3 個になり、各リストにはトラスと同じ数の項目が含まれます。
3. このデータは変更する必要があります。トラスをインスタンス化する場合は、ファミリで設定されているアダプティブ点と同じ数を使用する必要があります。これは 3 つの点で構成されているアダプティブコンポーネントです。このため、それぞれ 10 個の項目(numberOfTrusses)が含まれている 3 個のリストではなく、それぞれ 3 個の項目が含まれている 10 個のリストが必要になります。これにより、10 個のアダプティブコンポーネントを作成できます。
4. *List.Map* ノードを *List.Transpose* ノードに接続します。これで目的的データ出力を得ることができます。
5. データが正しいことを確認するには、*Polygon.ByPoints* ノードをキャンバスに追加して、Dynamo プレビューで再確認します。

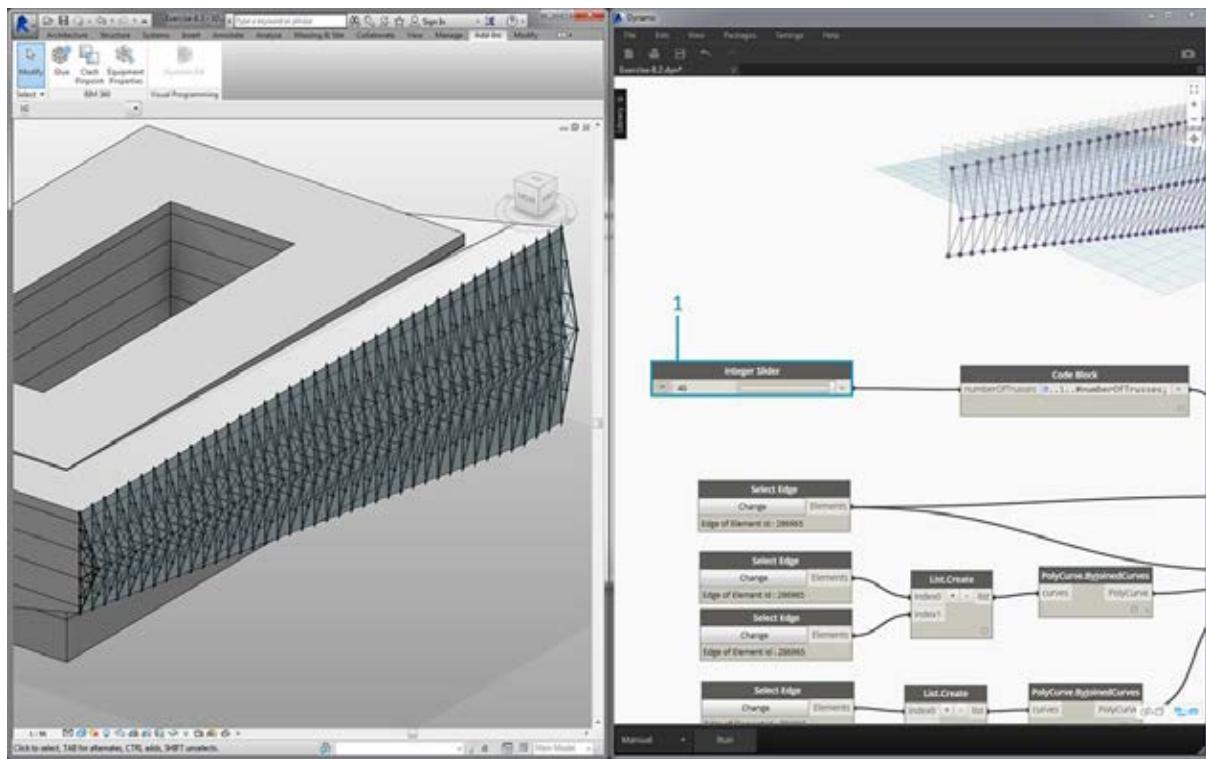


ポリゴンを作成するのと同じ方法で、アダプティブコンポーネントを配列します。

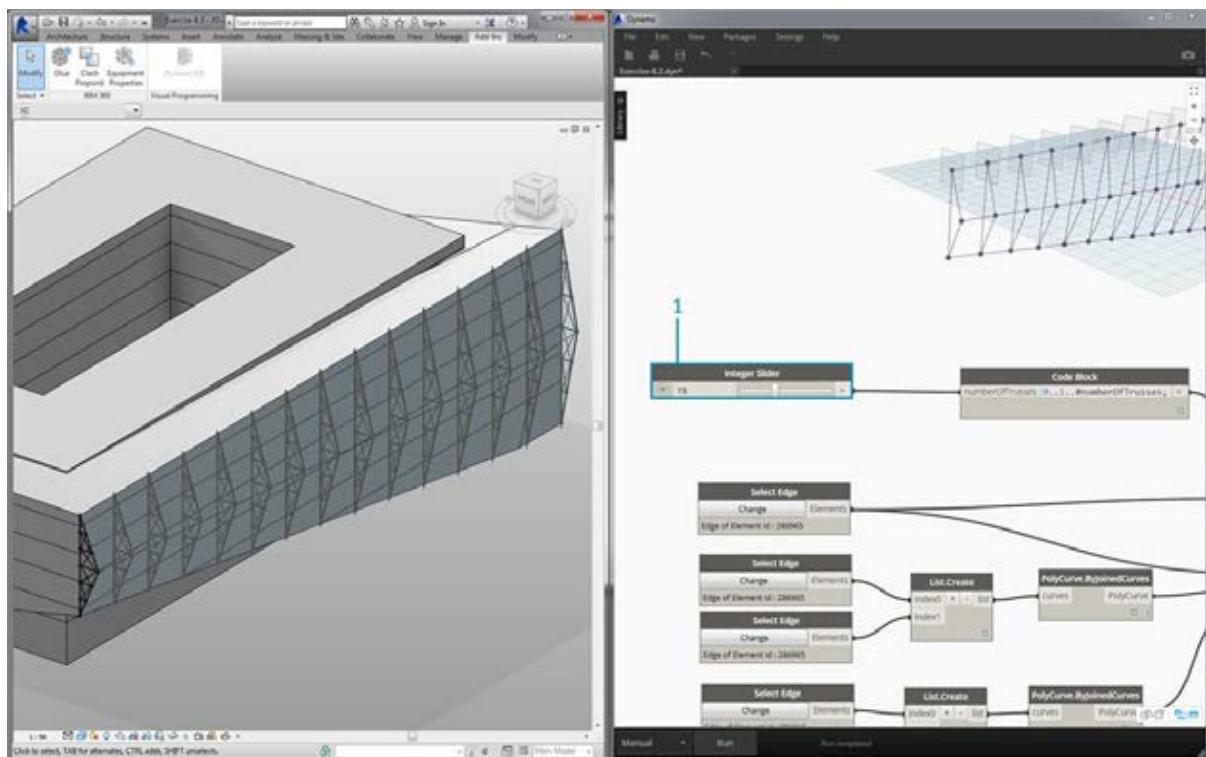
1. *AdaptiveComponent.ByPoints* ノードをキャンバスに追加し、*List.Transpose* ノードを *points* 入力に接続します。
2. *Family Types* ノードを使用して、*AdaptiveTruss* ファミリを選択し、これを *AdaptiveComponent.ByPoints* ノードの *familySymbol* 入力に接続します。



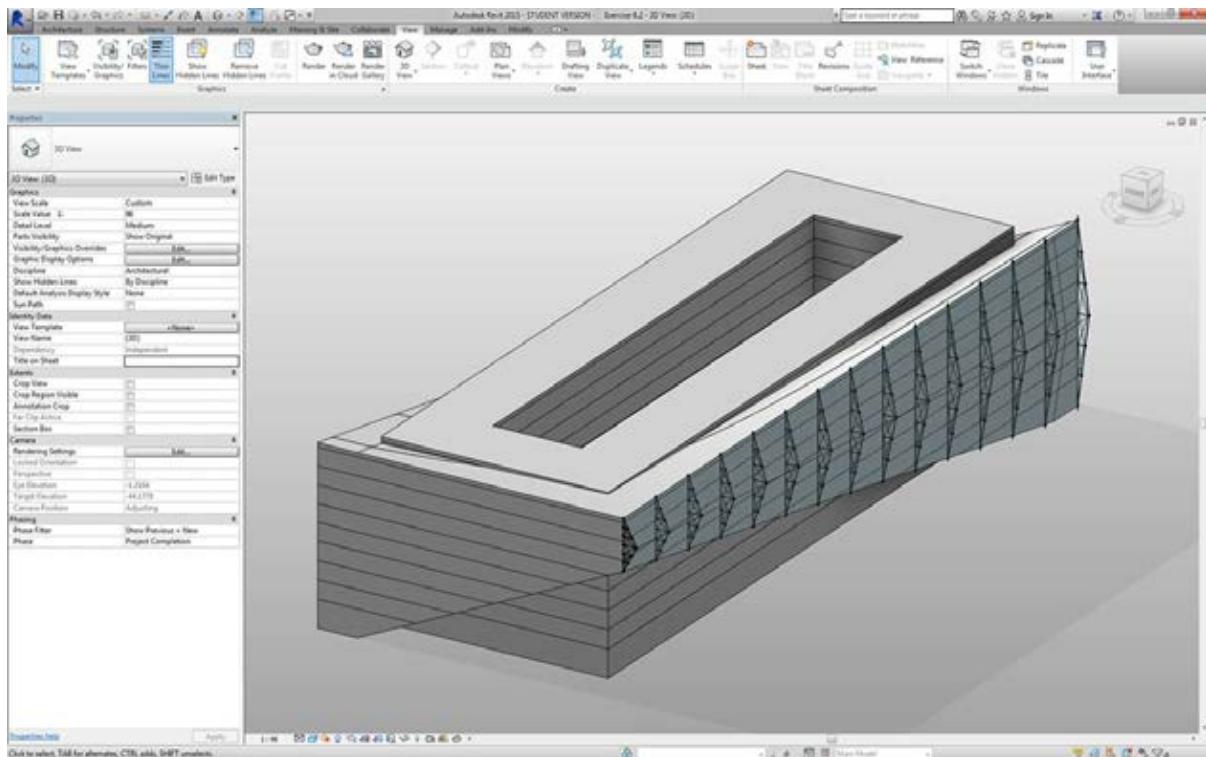
Revit で確認すると、10 個のトラスがファサード全体にわたって均等に配置されています。



1. グラフを柔軟に調整することができます。Integer Slider ノードを変更して `numberOfTrusses` を「40」にします。トラスの数が尋常でなく増えますが、パラメトリックリンクは機能しています。



1. トラス システムに慣れてきたら、`numberOfTrusses` に値 15 を設定してみましょう。



最後の確認として、Revit でマスを選択してインスタンス パラメータを編集することにより建物の形状を変更して、トラスがこれに従って変更されるかを確認します。この更新を確認するには、この Dynamo グラフを開いておく必要があります。閉じた場合、リンクはすぐに切断されます。

## DirectShape 要素

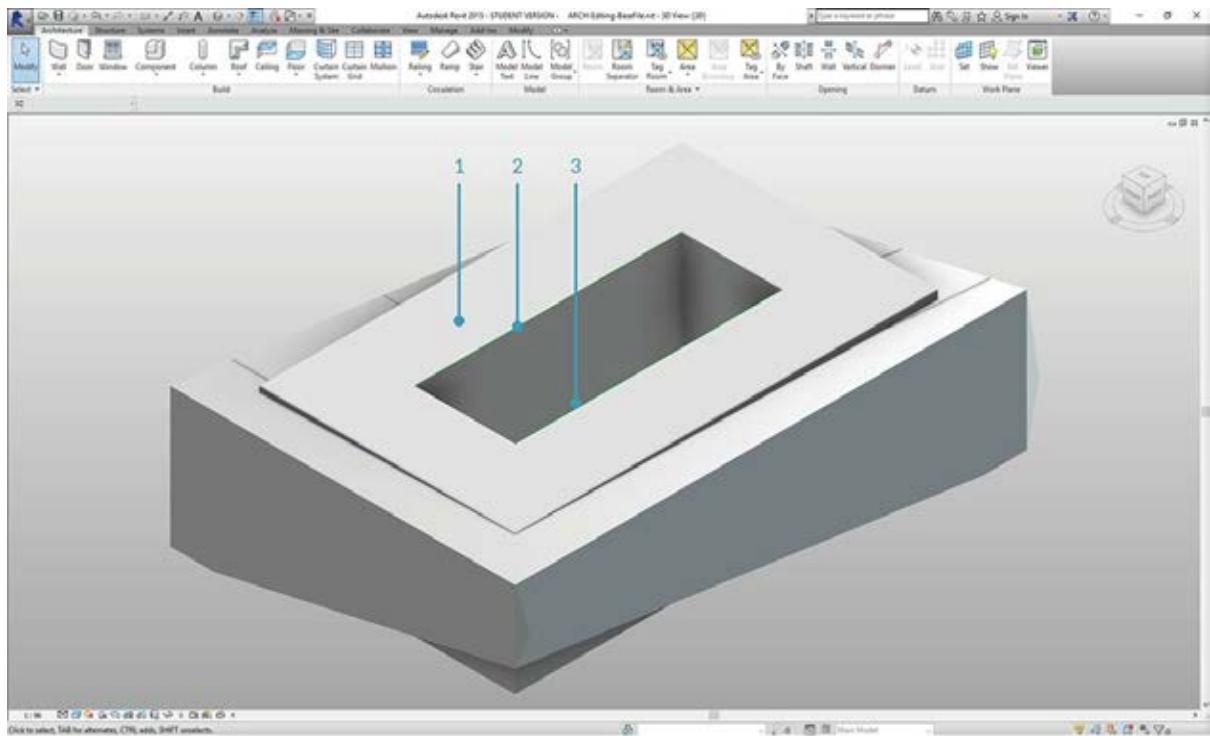
Dynamo のパラメトリック ジオメトリを Revit に読み込む別の方法として、DirectShape を使用する方法があります。つまり、DirectShape 要素と関連クラスは、外部で作成されたジオメトリ形状を Revit ドキュメントに保存する機能をサポートしています。ジオメトリには閉じたソリッドやメッシュを含めることができます。DirectShape の主な目的は、「実際」の Revit 要素を作成するための情報が不足している IFC や STEP などの他のデータ形式の形状を読み込むことにあります。DirectShape 機能は、IFC や STEP のワークフローのように、Dynamo で作成されたジオメトリを Revit プロジェクトに実際の要素として読み込むことに優れています。

Dynamo ジオメトリを DirectShape として Revit プロジェクトに読み込む方法を学習し、演習を行いましょう。この方法を使用すると、Dynamo グラフへのパラメトリックリンクを維持しつつ、読み込んだジオメトリのカテゴリ、マテリアル、名前を割り当てることができます。

## 演習

この演習に付属しているサンプル ファイルをダウンロードしてください(右クリックして[名前を付けてリンク先を保存]を選択)。すべてのサンプル ファイルの一覧については、付録を参照してください。

1. [DirectShape.dyn](#)
2. [ARCH-DirectShape-BaseFile.rvt](#)



このレッスンのサンプル ファイル ARCH-DirectShape-BaseFile.rvt を開きます。

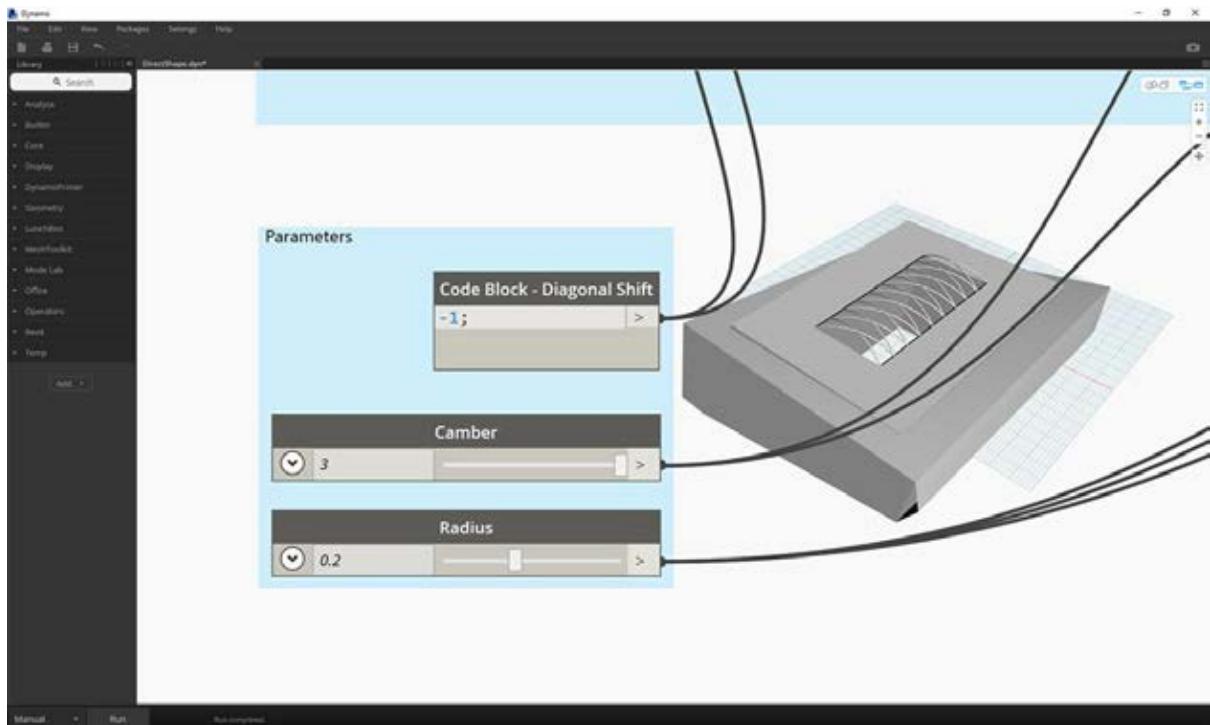
1. 3D ビューには、前の演習で使用した建物のマスが表示されます。
2. アトリウムのエッジに沿って見えるのは 1 つの参照曲線です。これは Dynamo で参照する曲線として使用します。
3. アトリウムの反対側のエッジに見えるのは別の参照曲線です。これも Dynamo で参照します。



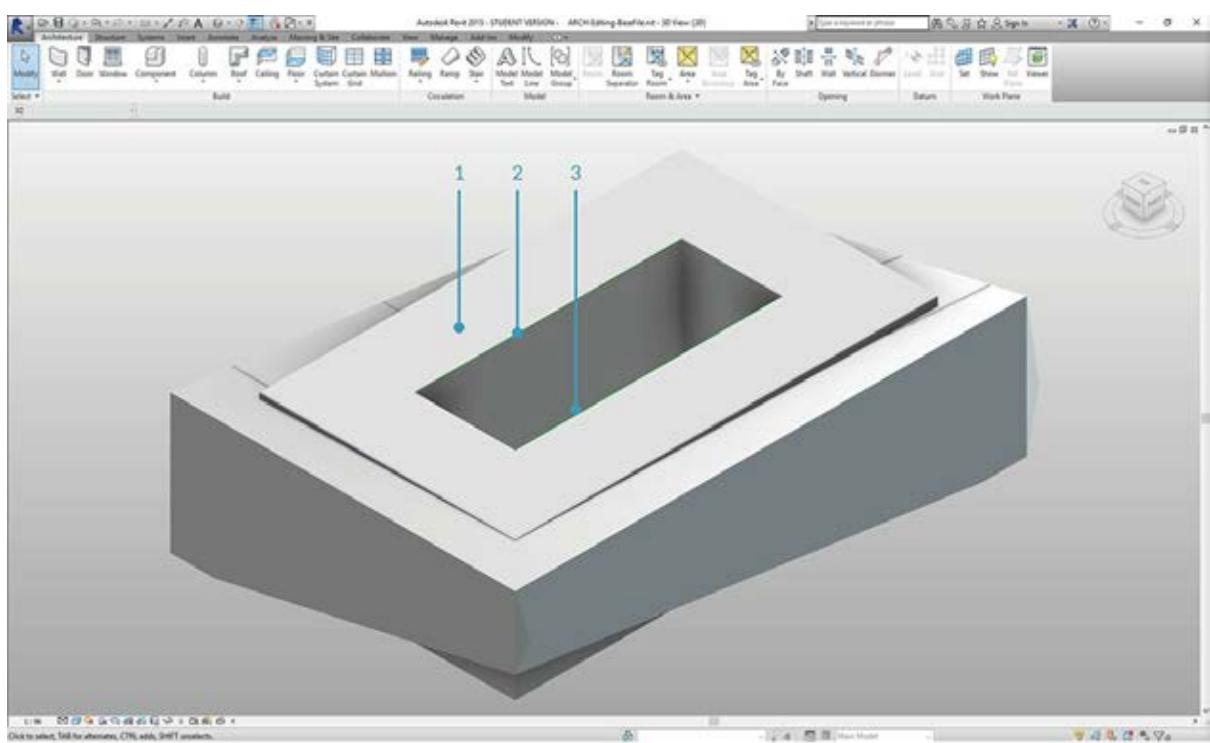
1. Dynamo でジオメトリを参照するには、Revit の各要素に対して *Select Model Element* ノードを使用します。Revit でマスを選択し、*Element.Faces* ノードを使用して Dynamo にジオメトリを読み込みます。マスが Dynamo プレビューに表示されます。
2. *Select Model Element* ノードと *CurveElement.Curve* ノードを使用して、一方の参照曲線を Dynamo に読み込みます。
3. *Select Model Element* ノードと *CurveElement.Curve* ノードを使用して、もう一方の参照曲線を Dynamo に読み込みます。



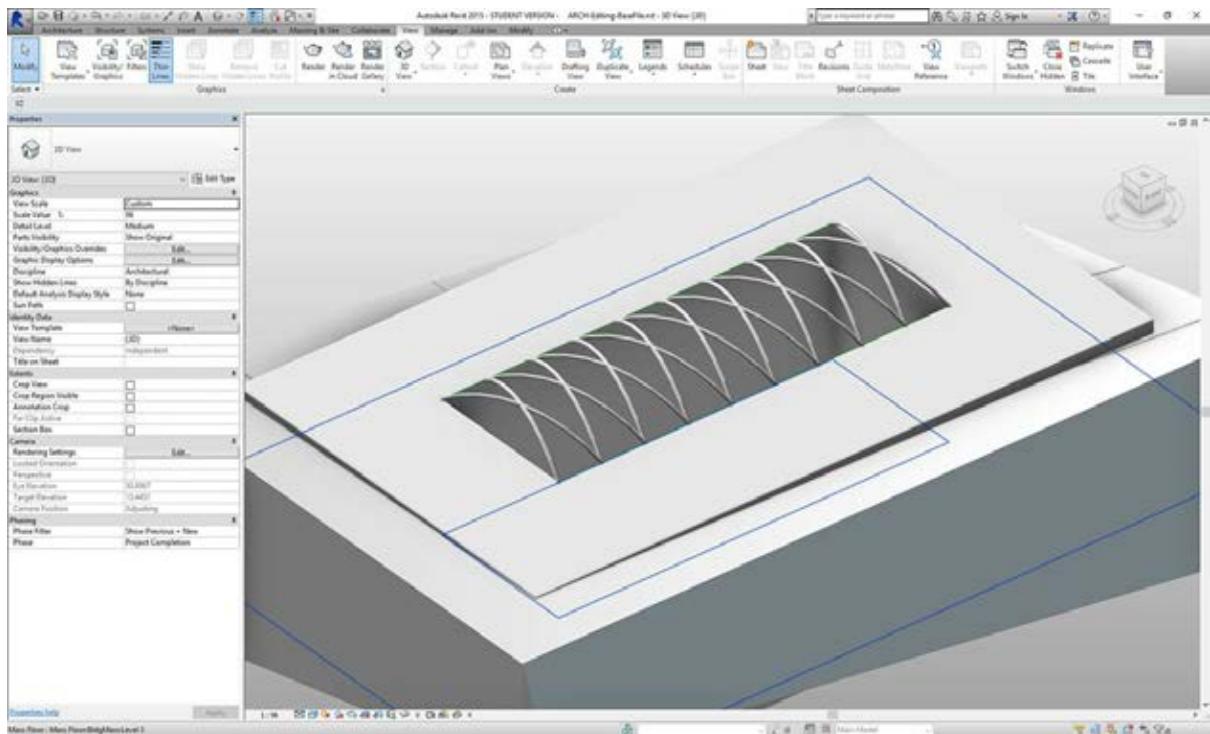
1. 縮小して画面をサンプル グラフの右に移動すると、大きなノードのグループが見えます。これらはジオメトリを操作し、Dynamo プレビューで表示される格子状の屋根構造を生成します。これらのノードは、手引の「[コード ブロック](#)」セクションで説明されている[ノードをコード化]機能を使用して生成されます。
2. この構造は、Diagonal Shift、Camber、Radius という 3 つの主要なパラメータでコントロールされます。



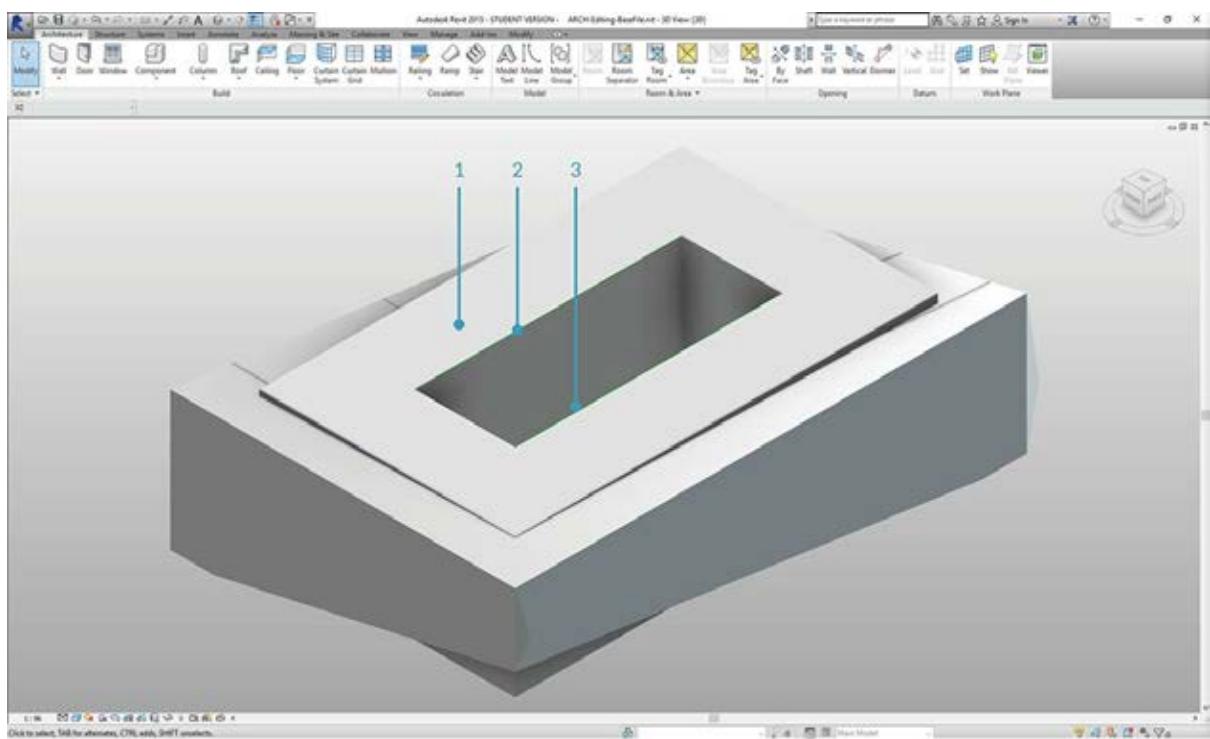
このグラフのパラメータをクローズアップします。これらを調整して、さまざまなジオメトリを出力することができます。



1. *DirectShape.ByGeometry* ノードをキャンバス上にドロップすると、**geometry**、**category**、**material**、**\*\*name\*\*** という 4 つの入力が表示されます。
2. ジオメトリは、グラフのジオメトリ作成部分から作成されるソリッドになります。
3. category 入力は、ドロップダウン *Categories* ノードを使用して選択されます。ここでは、[Structural Framing]を使用します。
4. 上記のノードの配列から material 入力が選択されます。この場合は、より単純に「既定値」として設定できます。



Dynamo を実行した後に Revit に戻ると、プロジェクト内の屋根に読み込まれたジオメトリが表示されます。これは生成モデルではなく構造フレーム要素です。Dynamo へのパラメトリックリンクは維持されます。



1. 「Diagonal Shift」パラメータを[-2]に変更して Dynamo グラフを調整した場合は、Dynamo を再度実行して、新しく読み込んだ DirectShape を取得します。

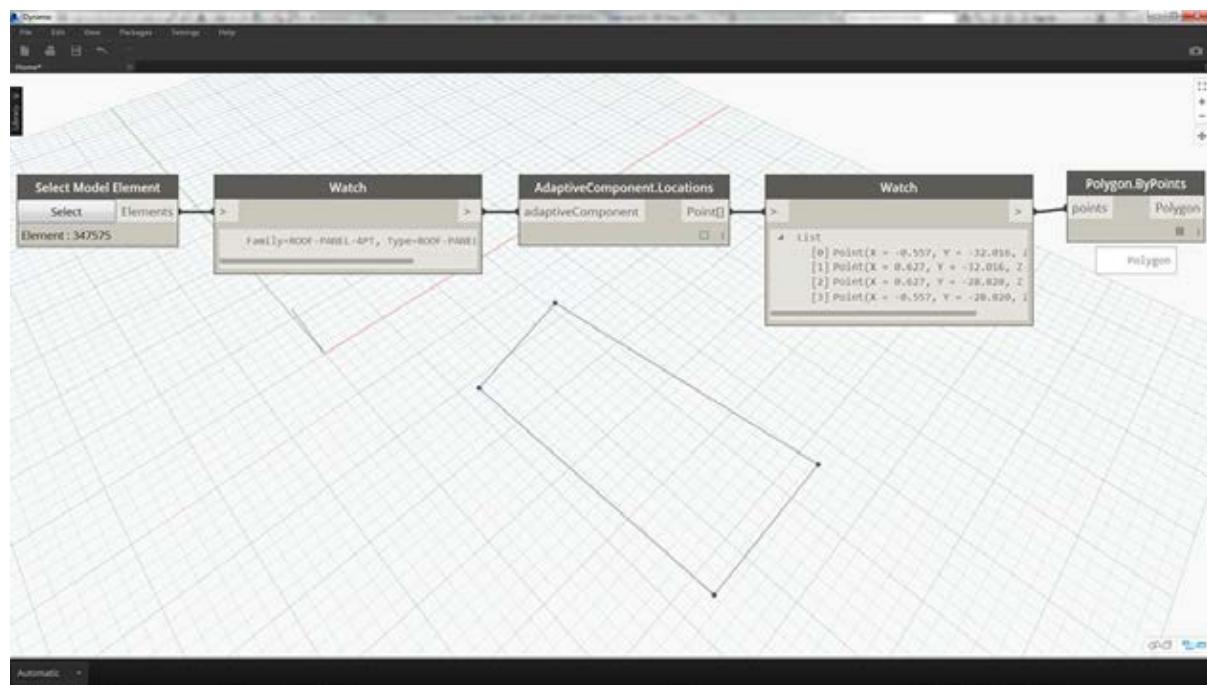
# カスタマイズ

## カスタマイズ

ここまで、基本的な建物のマスを編集する方法について紹介してきました。ここからは、多数の要素を一度に編集することで Dynamo と Revit のリンクについてより深く掘り下げていきましょう。カスタマイズする対象の規模が拡大すると、リストのデータ構造においてより高度な操作が要求されるので、カスタマイズの操作がより複雑になります。ただし、それを実行する背景で駆動している原理原則は、根本的にはこれまでと変わりありません。検討のために、アダプティブコンポーネントのセットからいくつかの事例を取り上げてみましょう。

### 点の位置

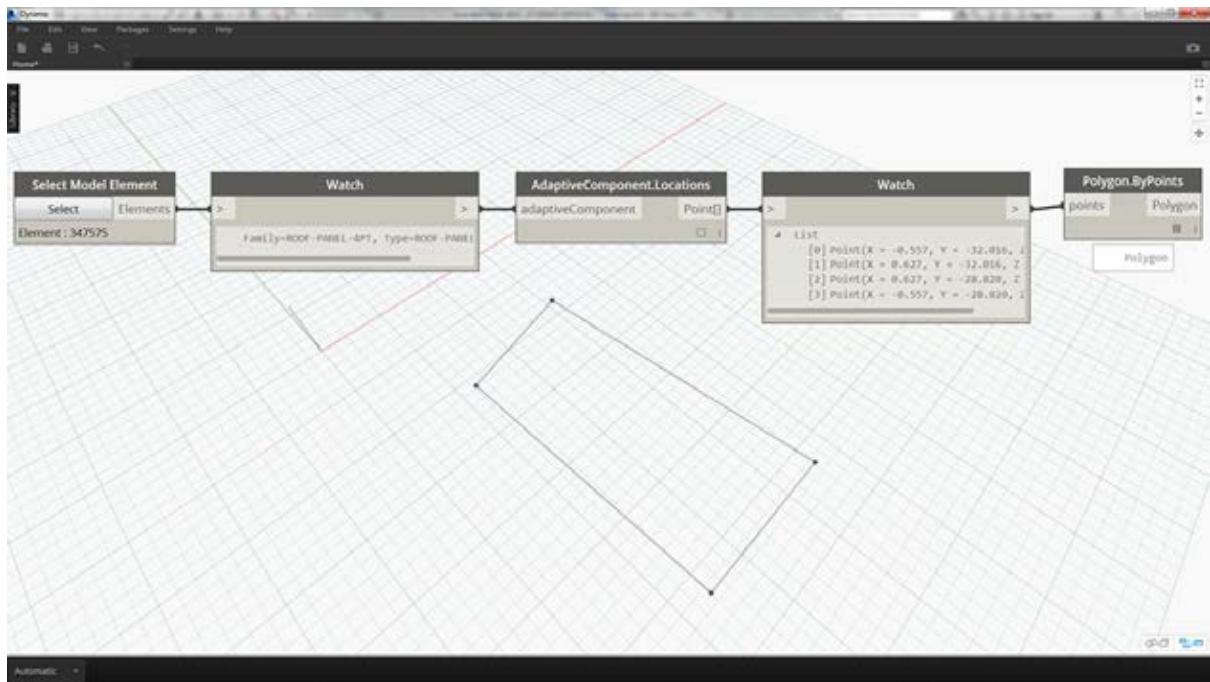
アダプティブコンポーネントを既に作成したという前提で、これからその点群の位置に基づいてパラメータを編集していくことにします。点群により、たとえば、要素の領域にかかる厚みのパラメータをコントロールすることができます。また、太陽光の年間露光量にかかる透過性のパラメータをコントロールすることもできます。Dynamo では、少ない手順で簡単に解析結果をパラメータに渡すことができます。次の演習でその基本的な手順を実践してみましょう。



*AdaptiveComponent.Locations* ノードを使用して、選択したアダプティブコンポーネントのアダプティブ点のクエリーを実行します。これにより、Revit の要素を解析用に抽出したバージョンを使用して作業することができます。

アダプティブコンポーネントを構成する点の位置を抽出することで、その要素に関するさまざまな解析を行うことができます。4 点構成のアダプティブコンポーネントにより、たとえば指定したパネルにおける水平面からの偏差を検討することができます。

### 太陽の向きの解析



再マッピング機能を使用すると、一連のデータセットを一定のパラメータ範囲にマッピングすることができます。これはパラメトリックモデリングで使用する基本的なツールです。これ以降の演習で実際に扱ってみることにします。

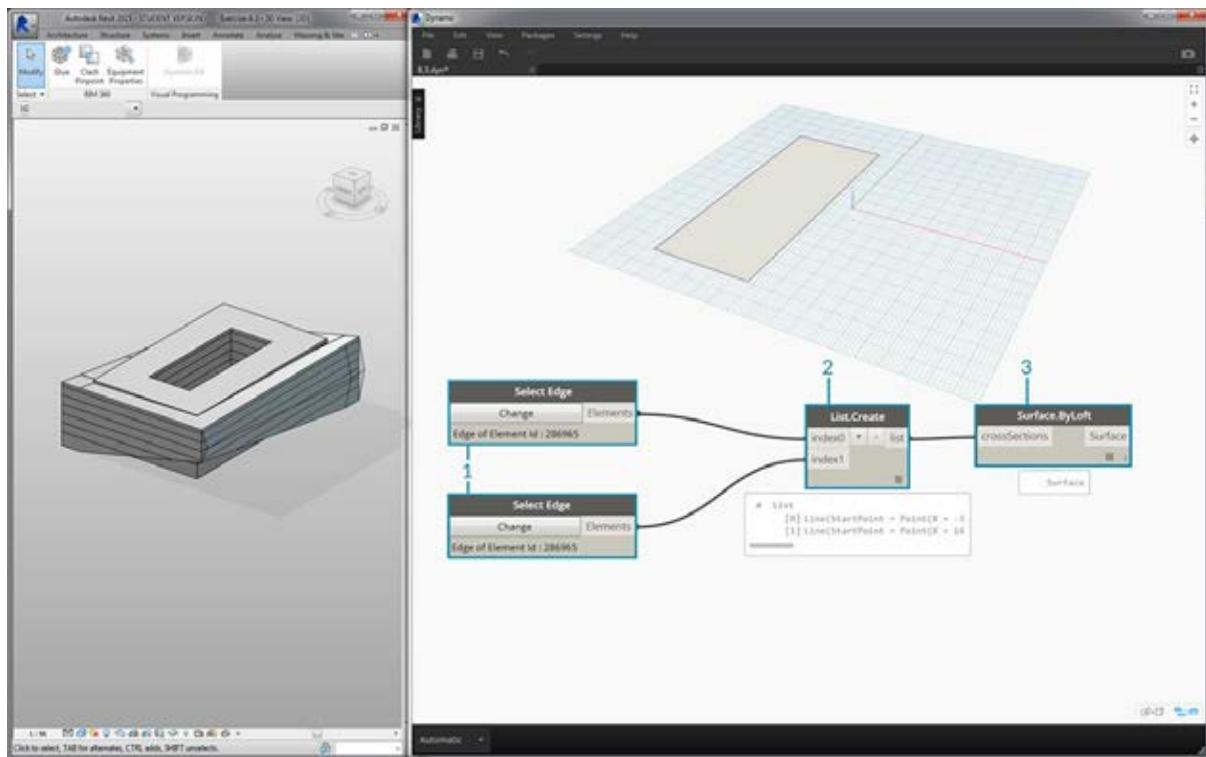
Dynamo を使用すると、アダプティブコンポーネントを構成する点群の位置から、要素ごとに最も適した平面を作成することができます。さらに Revit ファイル内の太陽の位置を参照して、太陽に対するその平面の向きを、他のアダプティブコンポーネントと比較しながら検討することもできます。これ以降の演習で、アルゴリズムに基づいて屋根の形状を生成することで、その設定を行っていきましょう。

## 演習

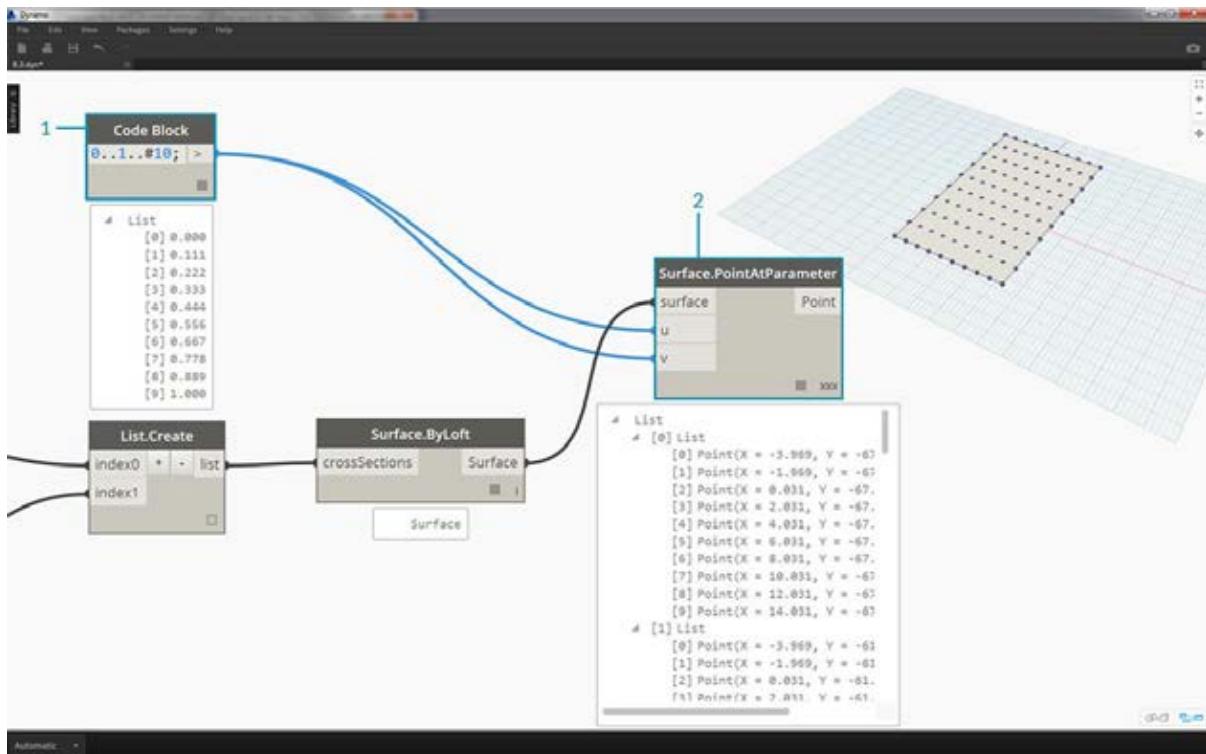
この演習に付属しているサンプルファイルをダウンロードしてください(右クリックして[名前を付けてリンク先を保存]を選択)。すべてのサンプルファイルの一覧については、付録を参照してください。

1. [Customizing.dyn](#)
2. [ARCH-Customizing-BaseFile.rvt](#)

この演習では、前のセクションで紹介したテクニックについて詳しく説明します。このケースでは、Revit の要素からパラメトリック サーフェスを設定して、4 点構成のアダプティブコンポーネントをインスタンス化し、太陽に対する向きに基づいて編集します。



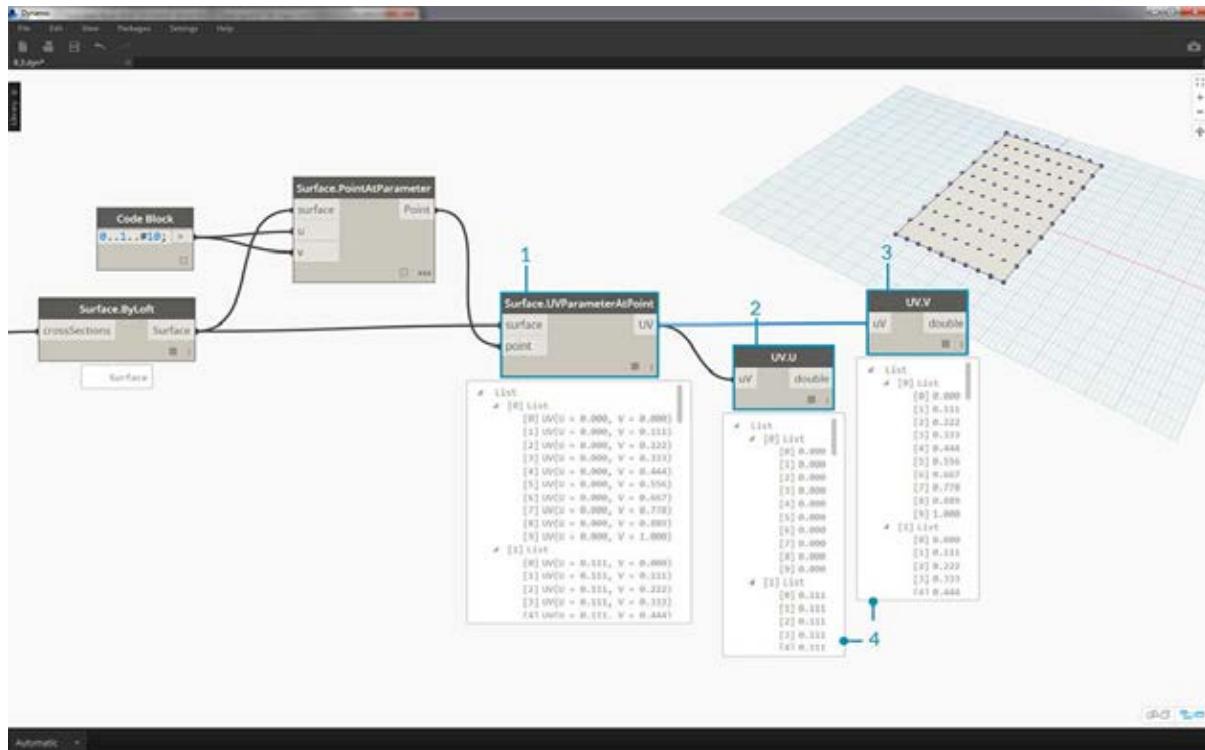
1. まず、Select Edge ノードを使用して 2 本のエッジを選択します。2 本のエッジはアトリウムの長辺です。
2. List.Create ノードを使用して、2 本のエッジを組み合わせて 1 つのリストを作成します。
3. Surface.ByLoft ノードを使用して、2 本のエッジの間に 1 つのサーフェスを作成します。



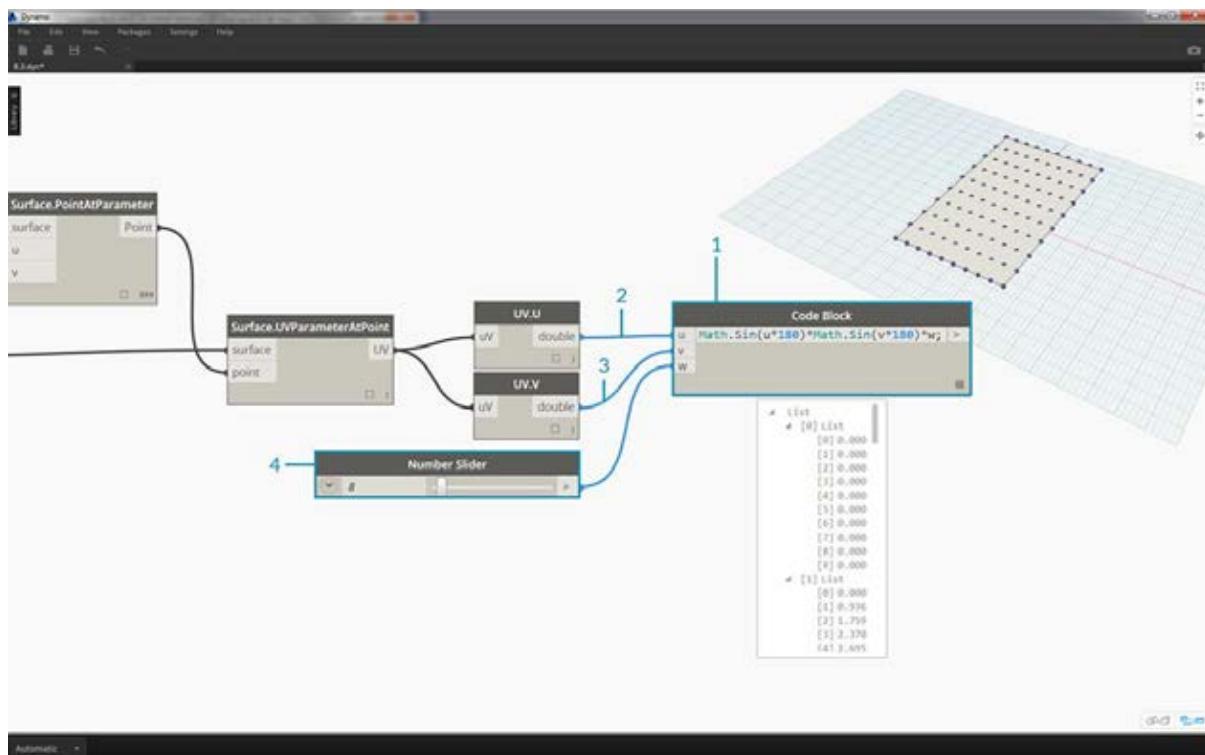
1. Code Block ノードを使用して、0 から 1 までの範囲を 10 等分した値 `0..1..#10;` に指定します。
2. Code Block ノードから Surface.PointAtParameter ノードの `u` 入力と `v` 入力に接続し、Surface.ByLoft ノードを `surface` 入力に接続します。ノードを右クリックして、レーシングを外積に変更します。これで、サーフェス上の点群から構成されるグリッドを取得できるようになります。

この点群によるグリッドは、パラメータに基づいて設定されたサーフェスの制御点として機能します。これら各点の位置を `u` と `v` の値として抽出することで、その値をパラメータの式に代入し、同一のデータ構造を保持できます。これを行うには、先ほど作成した点群の

位置のパラメータをクエリーする必要があります。

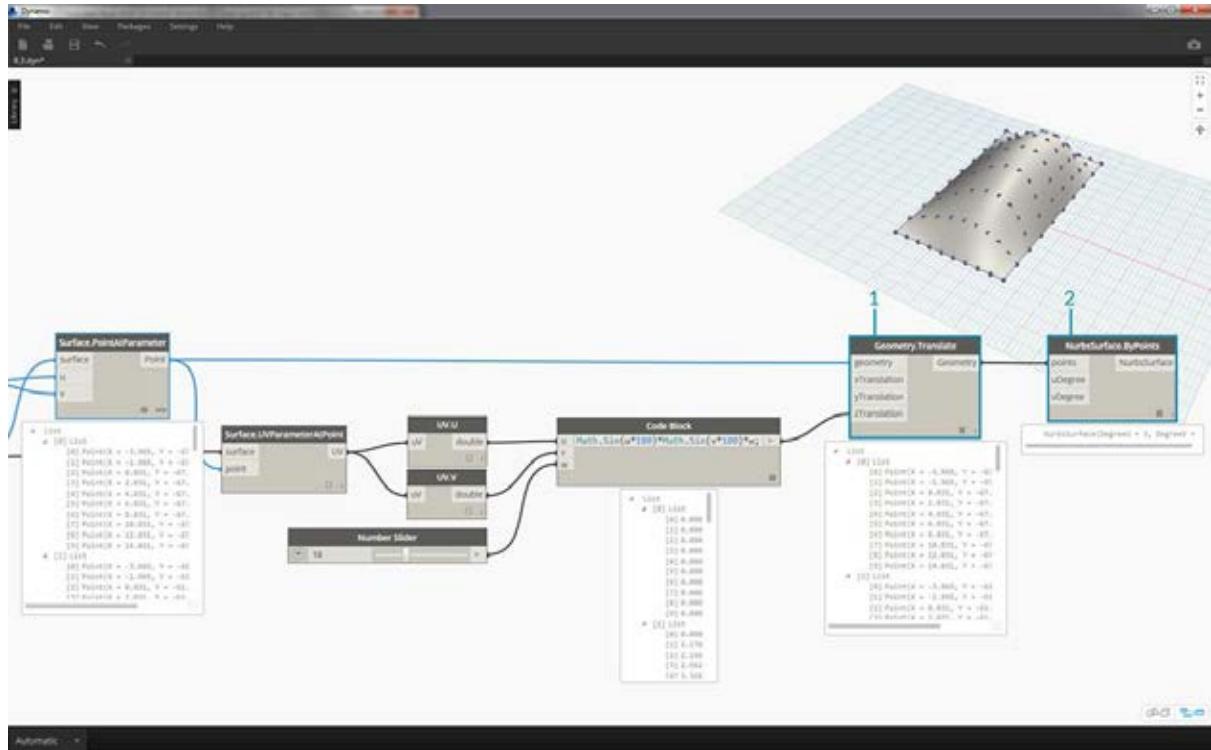


1. *Surface.ParameterAtPoint* ノードをキャンバスに追加し、その入力を上図のように接続します。
2. *UV.U* ノードを使用して、上記のパラメータの *u* の値をクエリーします。
3. *UV.V* ノードを使用して、上記のパラメータの *v* の値をクエリーします。
4. サーフェス上のすべての点に対応する *u* と *v* の値が列出されます。これで、適切なデータ構造で 0 から 1 までの範囲で各値を取得したので、パラメトリックアルゴリズムを適用する準備ができました。



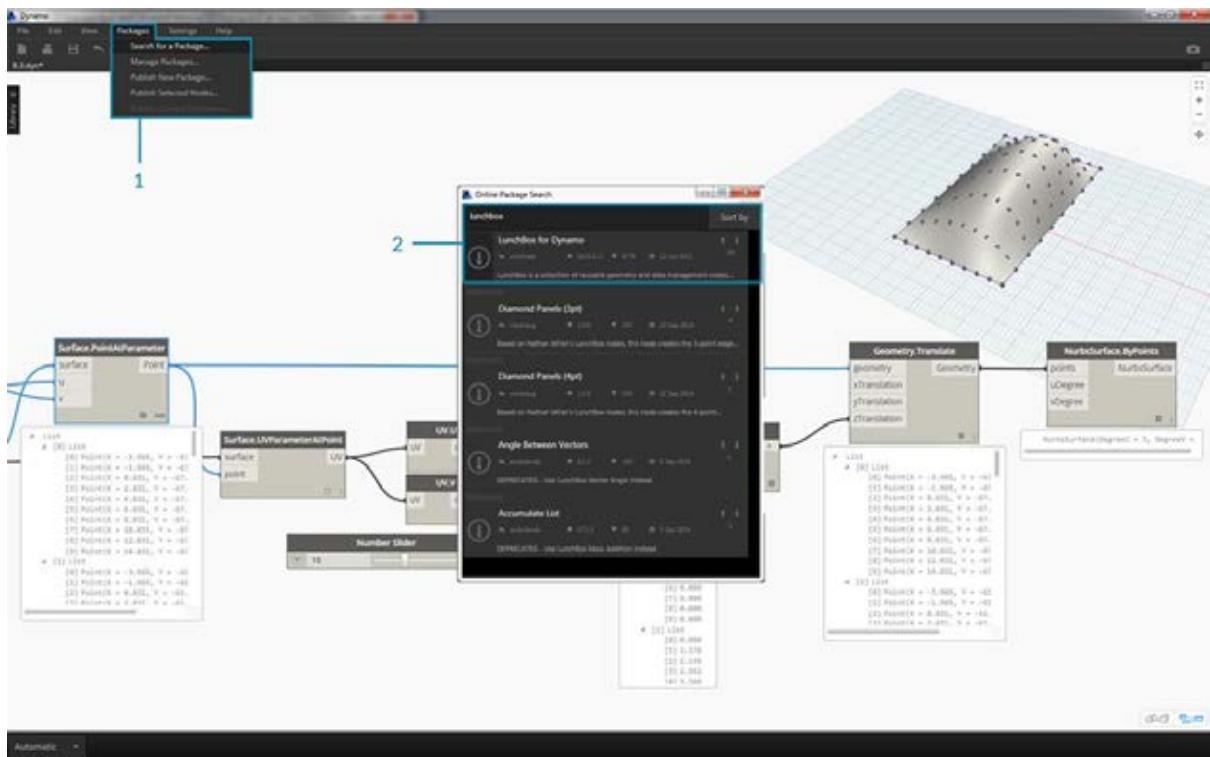
1. キャンバスに *Code Block* ノードを追加して、次のコードを入力します。 $\text{Math.Sin}(u*180)*\text{Math.Sin}(v*180)*w;$  これは、平坦なサーフェスから正弦波状の隆起を作成するパラメトリック関数です。

2. *u* 入力に、*UV.U* ノードからの出力を接続します。
3. *v* 入力に、*UV.V* ノードからの出力を接続します。
4. *w* 入力は形状の振幅を表します。そのため、ここには *Number Slider* を接続します。

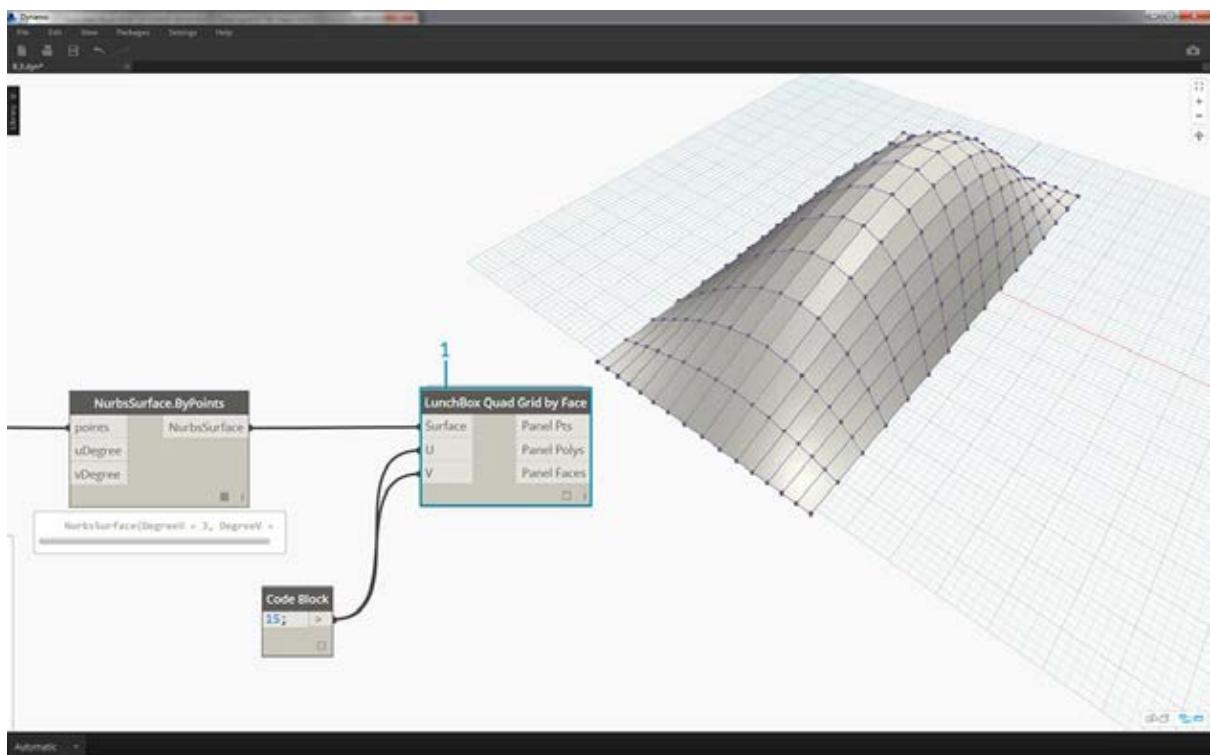


1. ここまで手順で、アルゴリズムによって定義された値のリストを取得することができました。この値のリストを使用して、点群を Z の正の向きに動かしましょう。*Code Block* ノードを *Geometry.Translate* ノードの *zTranslation* 入力に接続し、*Surface.PointAtParameter* ノードをやはり *Geometry.Translate* ノードの *geometry* 入力に接続します。Dynamo のプレビューに新しい点群が表示されるはずです。
2. 最後に、前の手順から *NurbsSurface.ByPoints* ノードの *points* 入力に接続することで、サーフェスを作成します。こうしてパラメトリック サーフェスができあがりました。スライダを自由に動かして、隆起面が上下するのを確認してください。

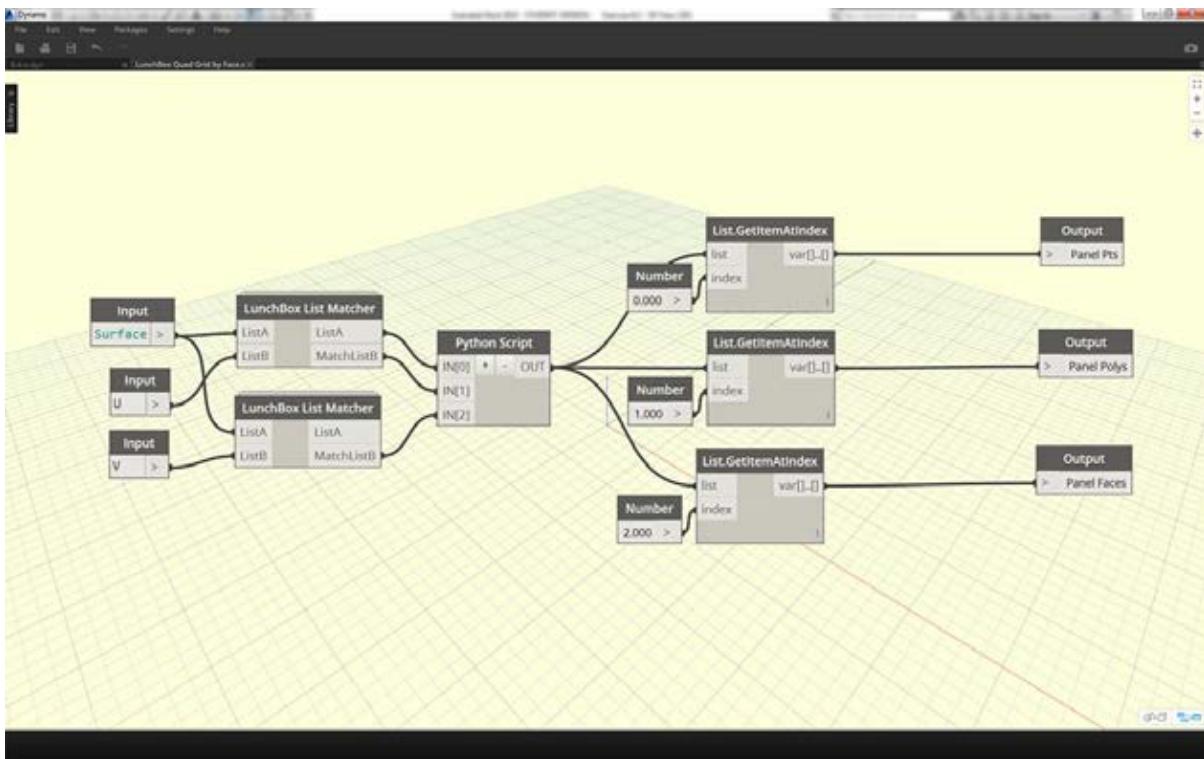
パラメトリック サーフェスを使用して、その曲面を多数の小さなパネルから成る構造に変換する方法を設定し、4 点構成のアダプティブ コンポーネントを配列していきましょう。Dynamo で提供されている既定のノードには、サーフェスを多面構造に変換する機能をもつものはありません。そこで、コミュニティにアクセスして便利な Dynamo パッケージ入手しましょう。



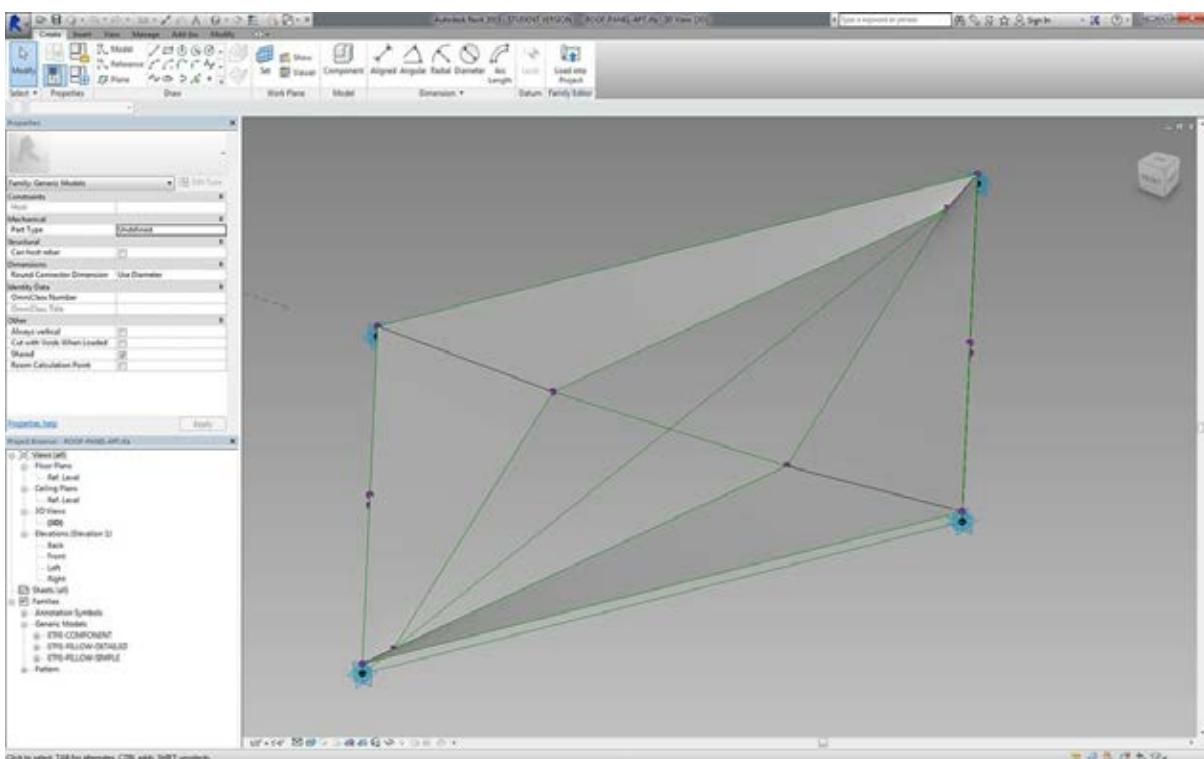
1. [パッケージ] > [パッケージの検索]に進みます。
2. 「LunchBox」という文字列を検索して、「LunchBox for Dynamo」をダウンロードします。このパッケージは、この種のジオメトリ操作にじつに役に立つツール セットです。



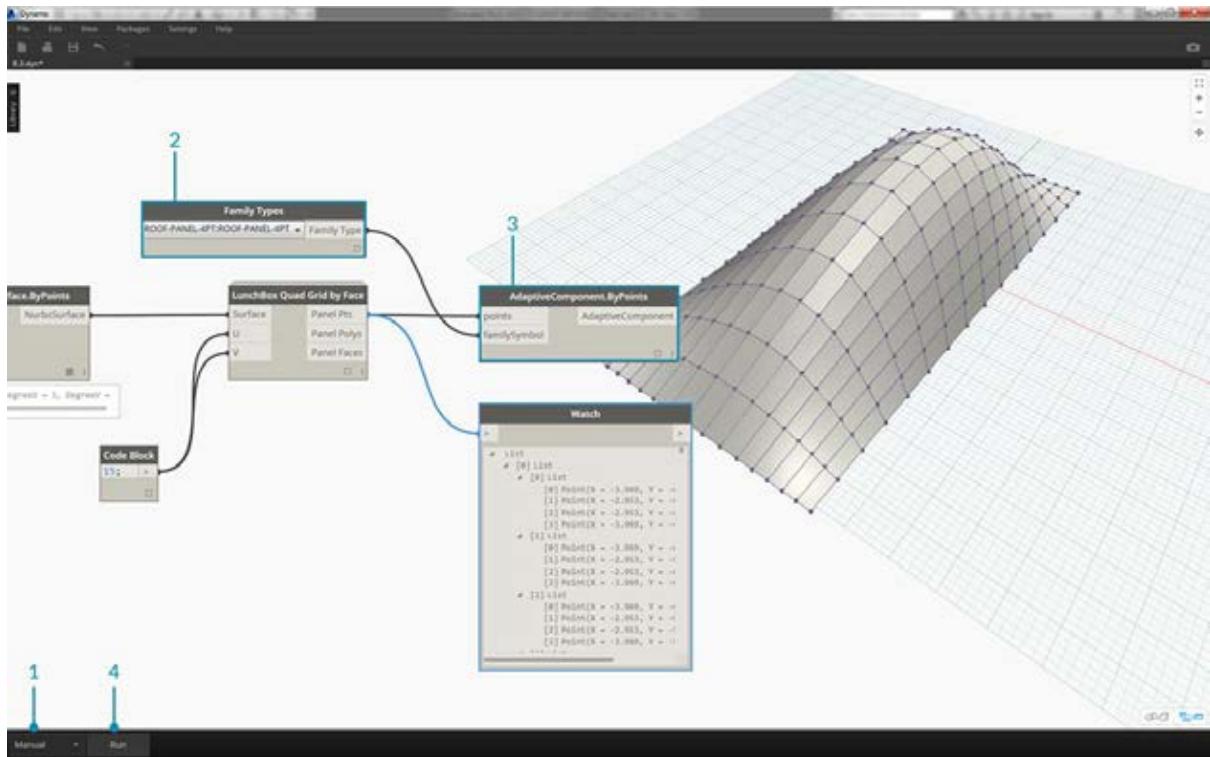
1. ダウンロードすると、LunchBox スイートに完全にアクセスできるようになります。「Quad Grid」を検索し、LunchBox Quad Grid By Face ノードを選択します。このノードの surface 入力にパラメトリック サーフェスを接続し、U 区分と V 区分を 15 に設定します。複数の長方形のパネルから成るサーフェスが Dynamo のプレビューに表示されます。



構成の詳細については、*Lunch Box* ノードをダブルクリックして確認してください。

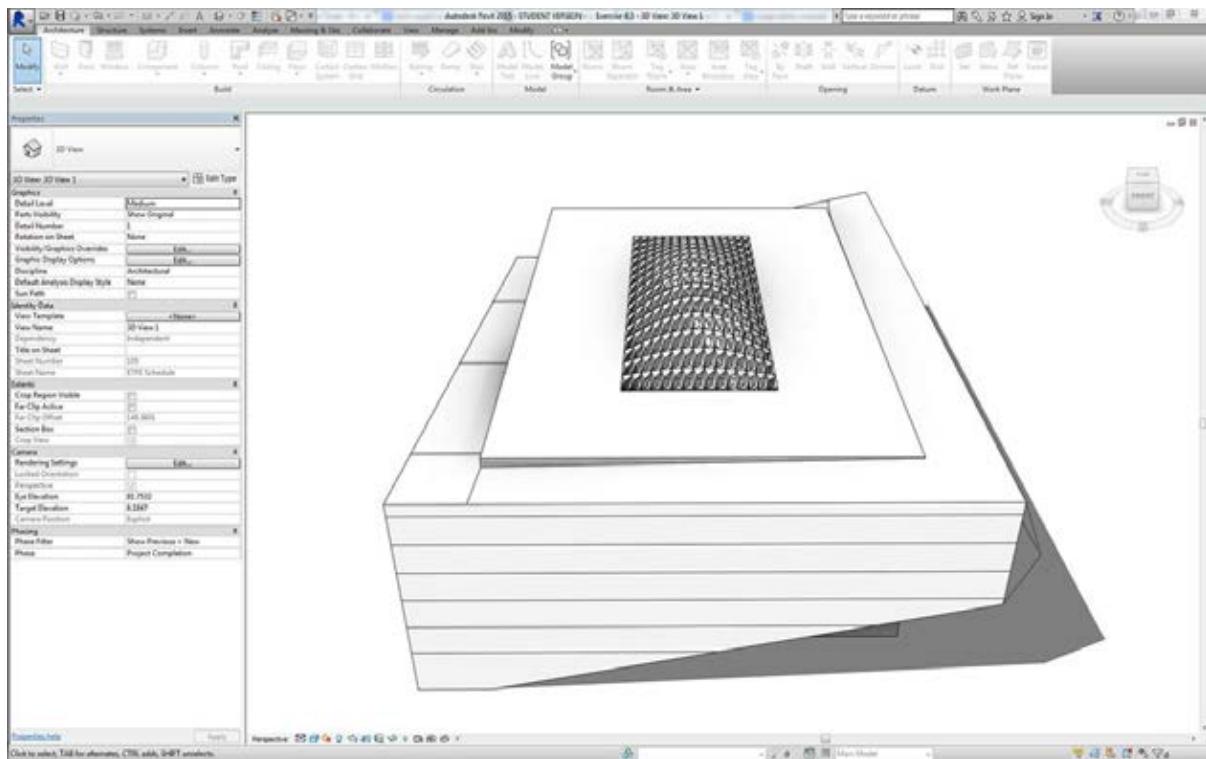


Revitに戻って、ここで使用しているアダプティブコンポーネントを簡単に確認しておきましょう。必ずしも実際に確認する必要はありませんが、いずれにせよこれからインスタンス化する対象であるこのコンポーネントは屋根のパネルです。この4点構成のアダプティブコンポーネントは、ETFEシステムをおおまかに表現しています。中央の開口部は *ApertureRatio* というパラメータによってコントロールされています。

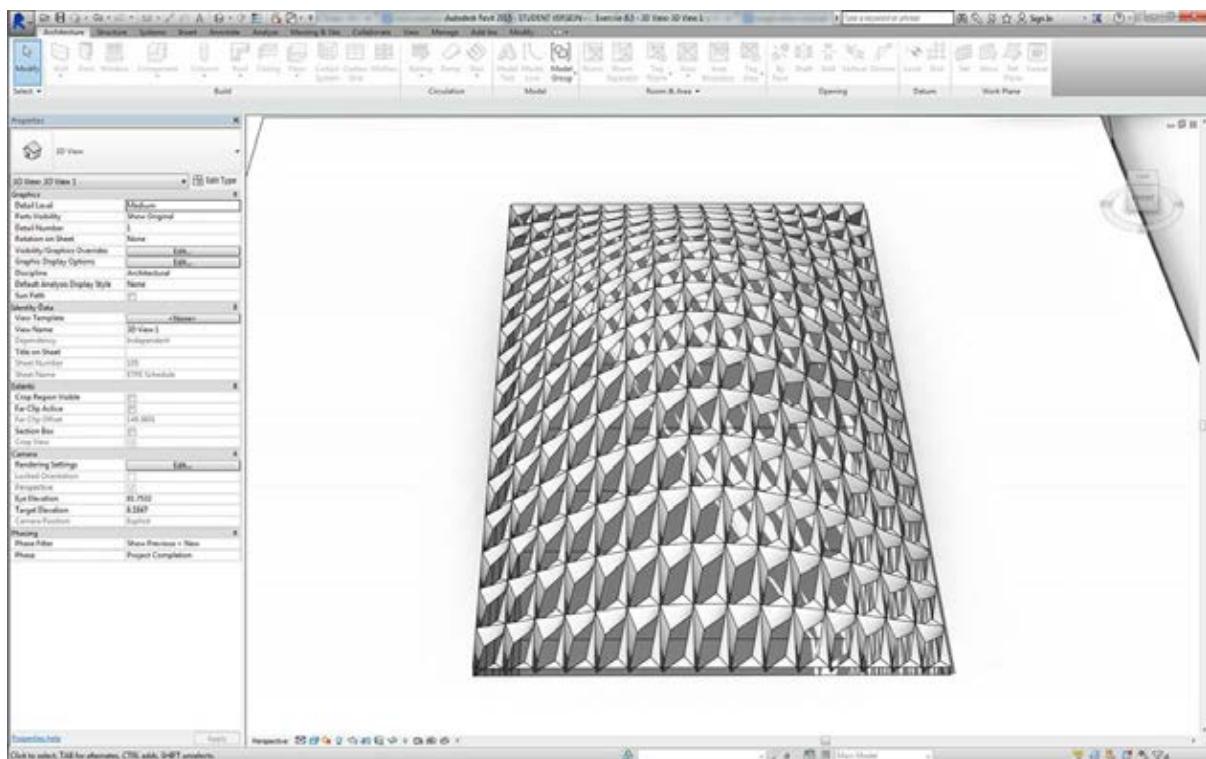


1. これから Revit 内の多数のジオメトリをインスタンス化するので、必ず事前に Dynamo のソルバを[手動]に切り替えてください。
2. *Family Types* ノードをキャンバスに追加し、[ROOF-PANEL-4PT]を選択します。
3. *AdaptiveComponent.ByPoints* ノードをキャンバスに追加し、その *points* 入力に *LunchBox Quad Grid by Face* ノードの *Panel Pts* 出力を接続します。*familySymbol* 入力に *Family Types* ノードを接続します。
4. [実行]をクリックします。Revit はジオメトリの作成中に計算に少々時間をかける必要があります。あまりにも時間がかかりすぎている場合、*Code Block* ノードの「15」の値をより小さな数に減らしてください。これを行うと、屋根の部分に使用されるパネルの数が減少します。

注: Dynamo でノードの計算に膨大な時間がかかる場合は、ノードをフリーズする機能を使用して、グラフの開発中に Revit 関連操作の実行を停止することができます。ノードをフリーズする操作の詳細については、[「ソリッド」の章の「フリーズ」セクション](#)を参照してください。

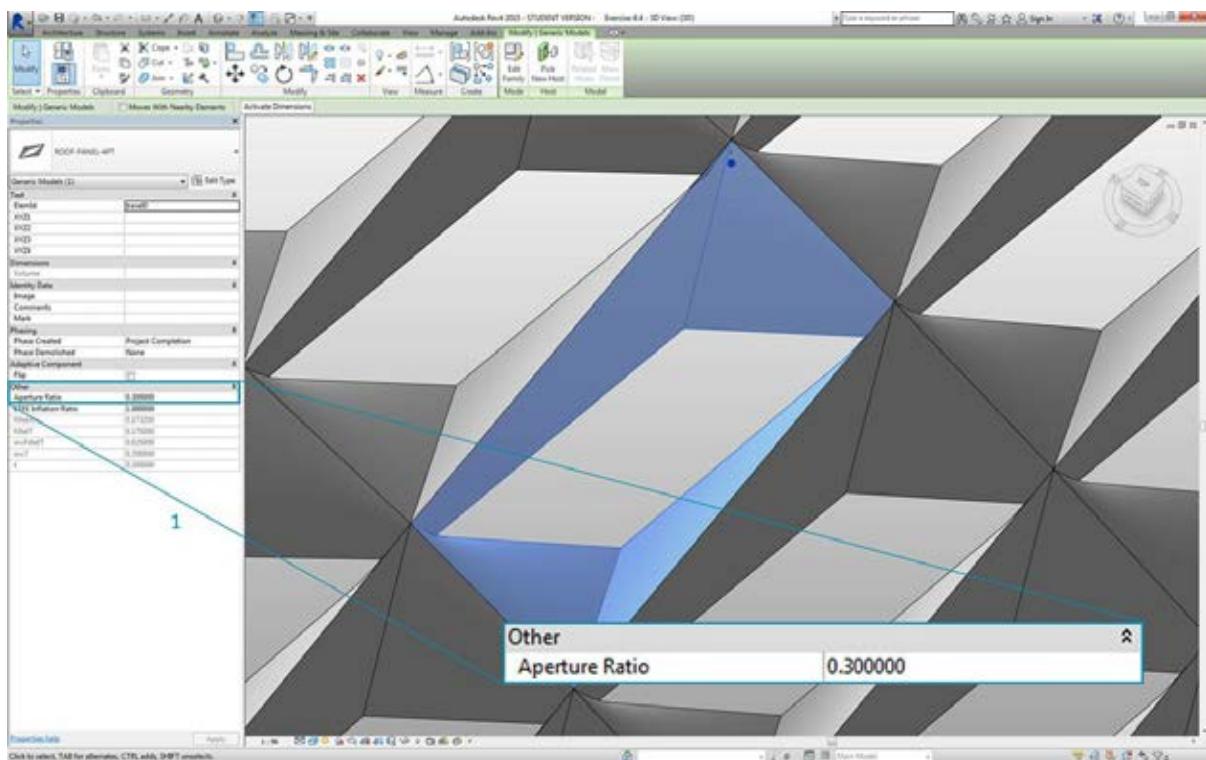


Revitに戻ると、屋根の上にパネルの配列が出現しています。

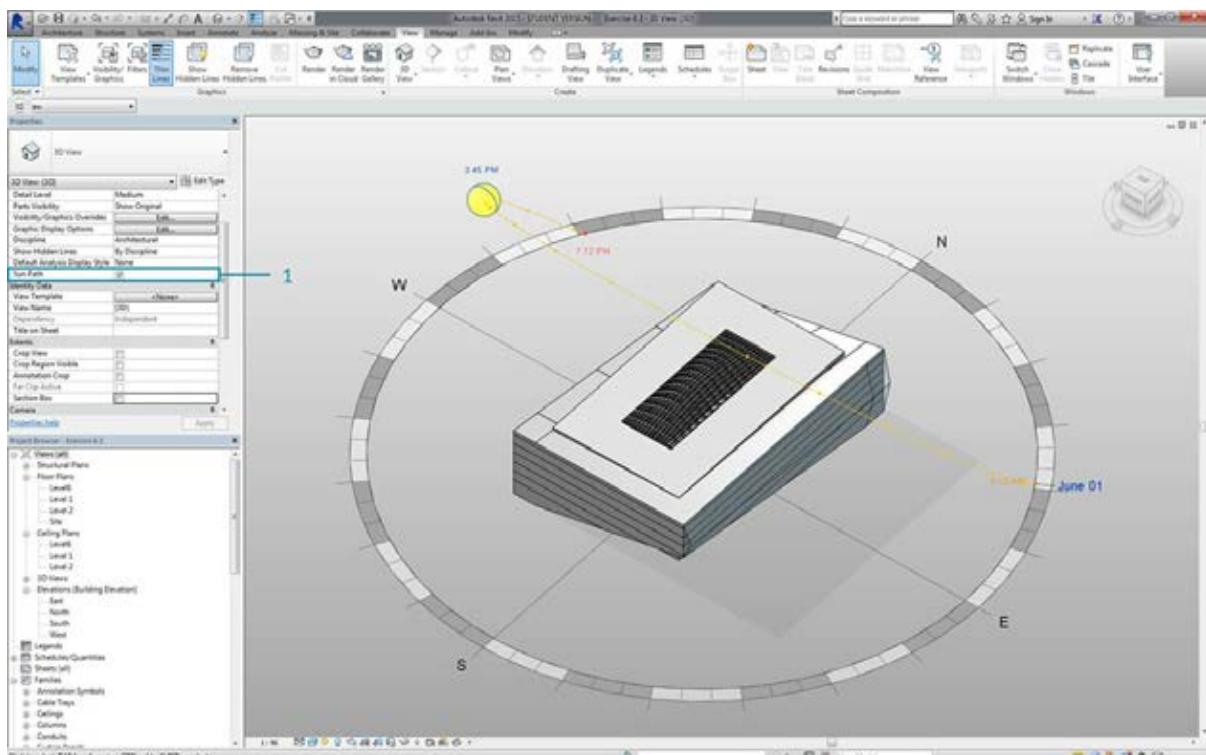


拡大表示すると、サーフェスの品質を詳細に確認できます。

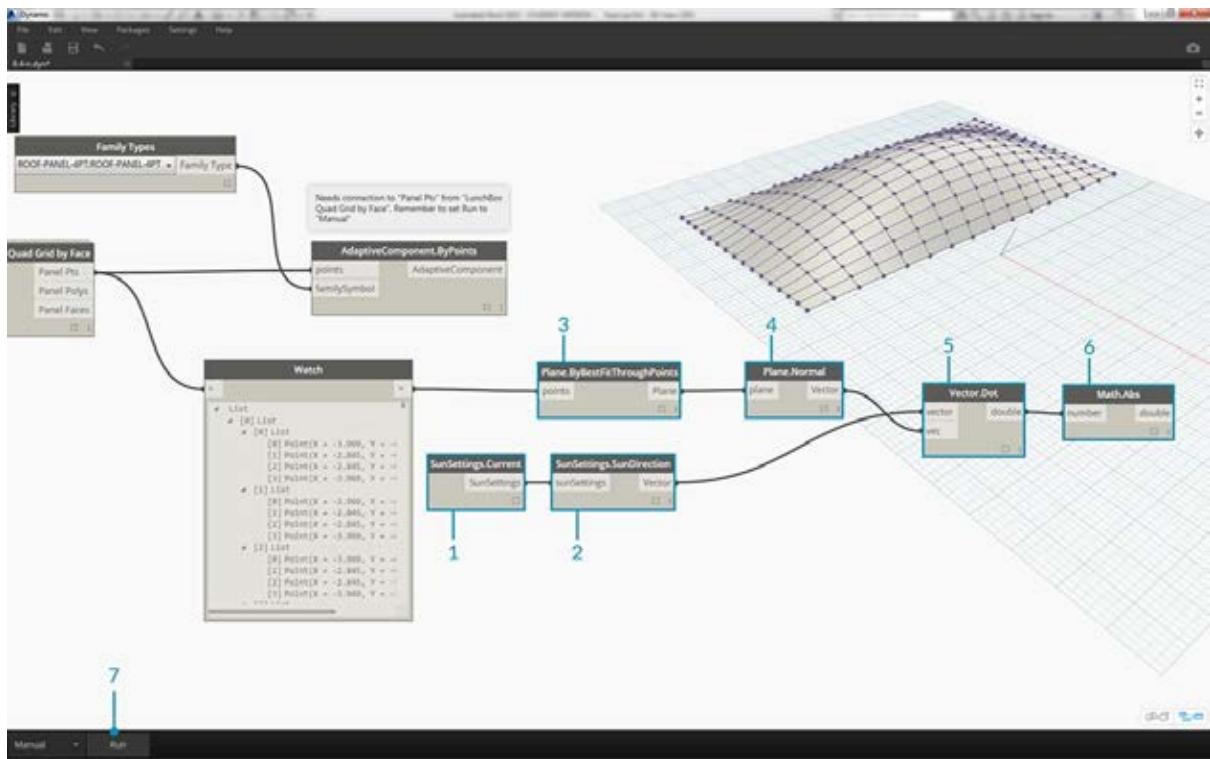
## 解析



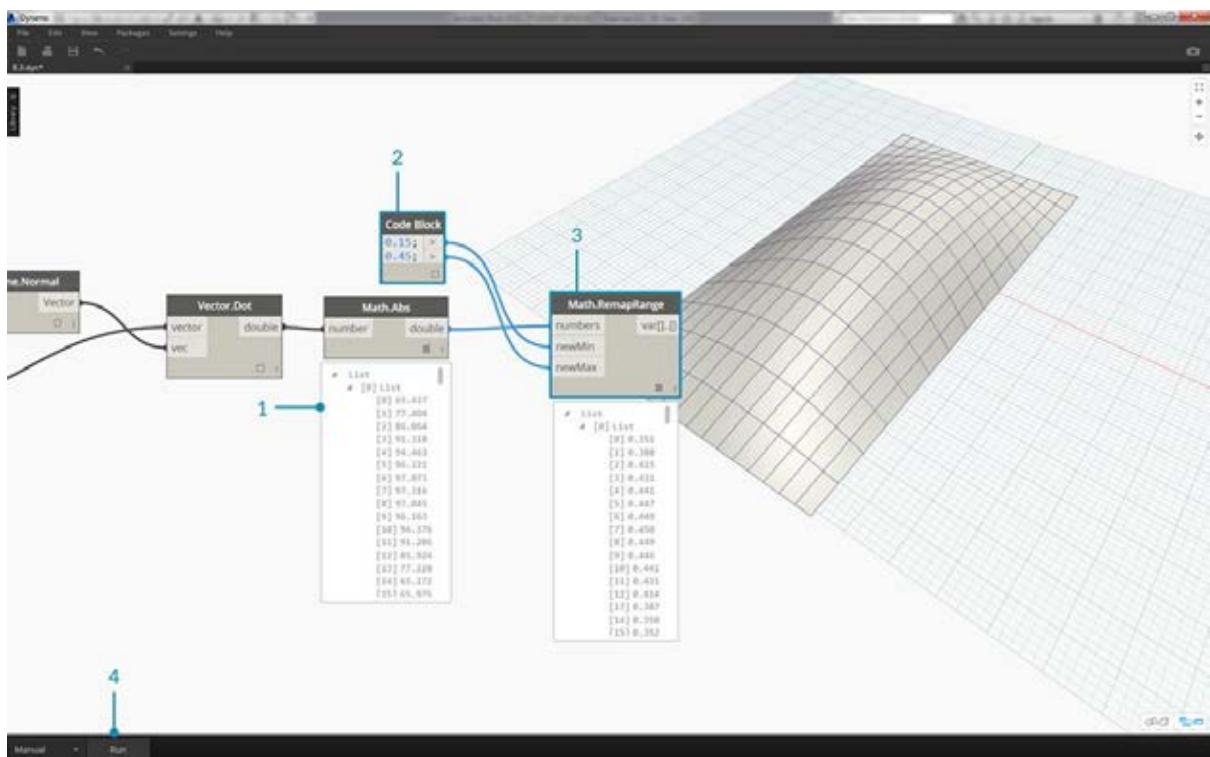
- 前の手順からさらに先へと進み、各パネルが太陽光を浴びている量に基づいてそれぞれのパネルの開き方をコントロールしてみましょう。Revitでビューを拡大表示して1つのパネルを選択すると、プロパティバーに[開口率]というパラメータが表示されます。ファミリは、開口率の範囲が0.05～0.45になるように設定されています。



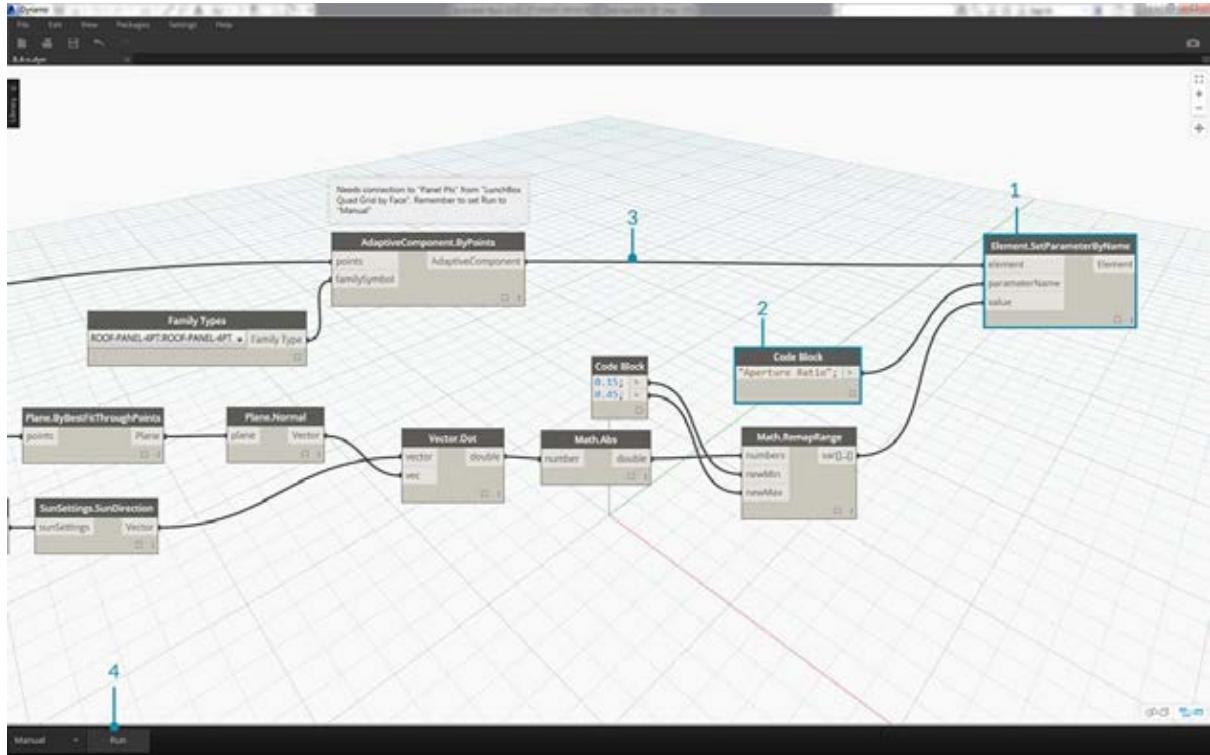
- 太陽の軌道の表示をオンにすると、Revitで現在の太陽の位置を確認することができます。



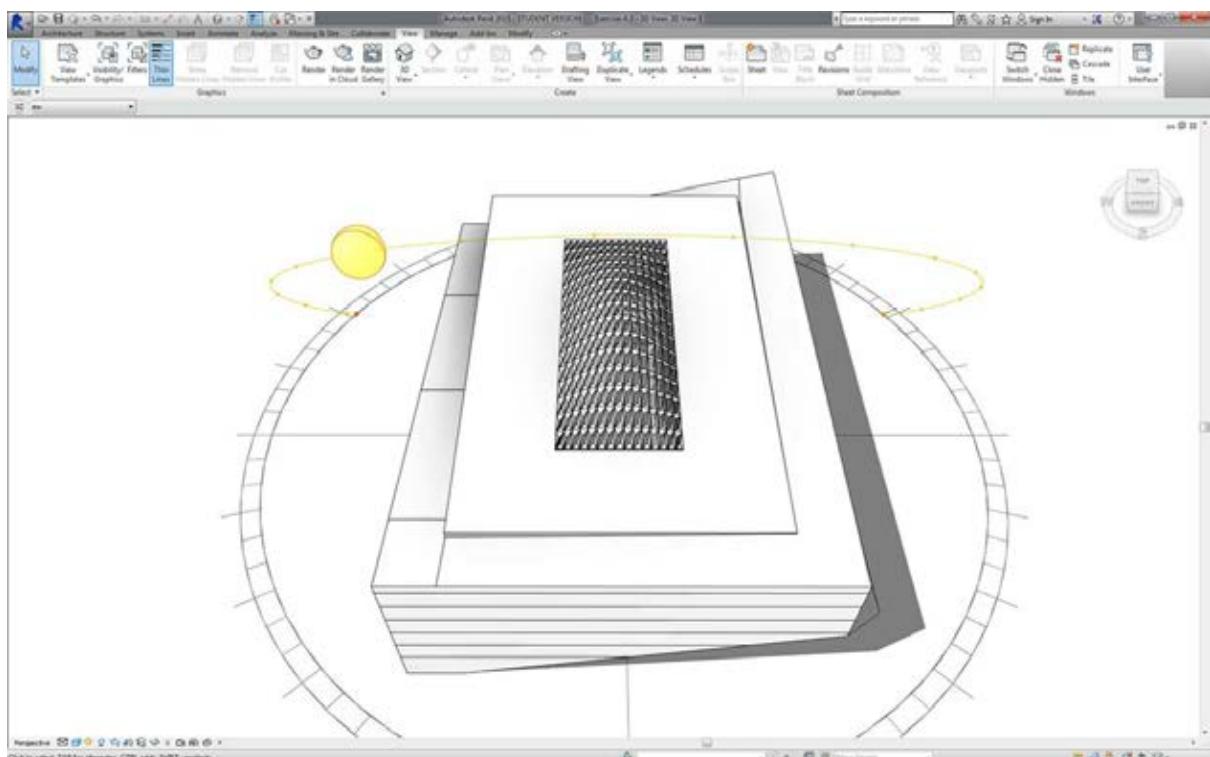
1. *SunSettings.Current* ノードを使用すると、この太陽の位置を参照することができます。
2. そのノードの *SunSettings* の出力を *Sunsetting.SunDirection* ノードの入力に接続し、太陽光のベクトルを取得します。
3. アダプティブコンポーネントの作成に使用した *Panel Pts* からの出力を、*Plane.ByBestFitThroughPoints* ノードを使用して、そのコンポーネントのために平面に近づけます。
4. この平面の法線のエクスポートを実行します。
5. 内積を使用して太陽の向きを計算します。内積に基づいて、2つのベクトルが平行であるかどうかを判定することができます。つまり、各アダプティブコンポーネントの平面法線を取得し、それと太陽光のベクトルを比較することで、太陽の向きをおおまかにシミュレートします。
6. 結果の絶対値を取得します。これにより、平面法線が逆方向を向いている場合に正確な内積が算出されます。
7. [実行]をクリックします。



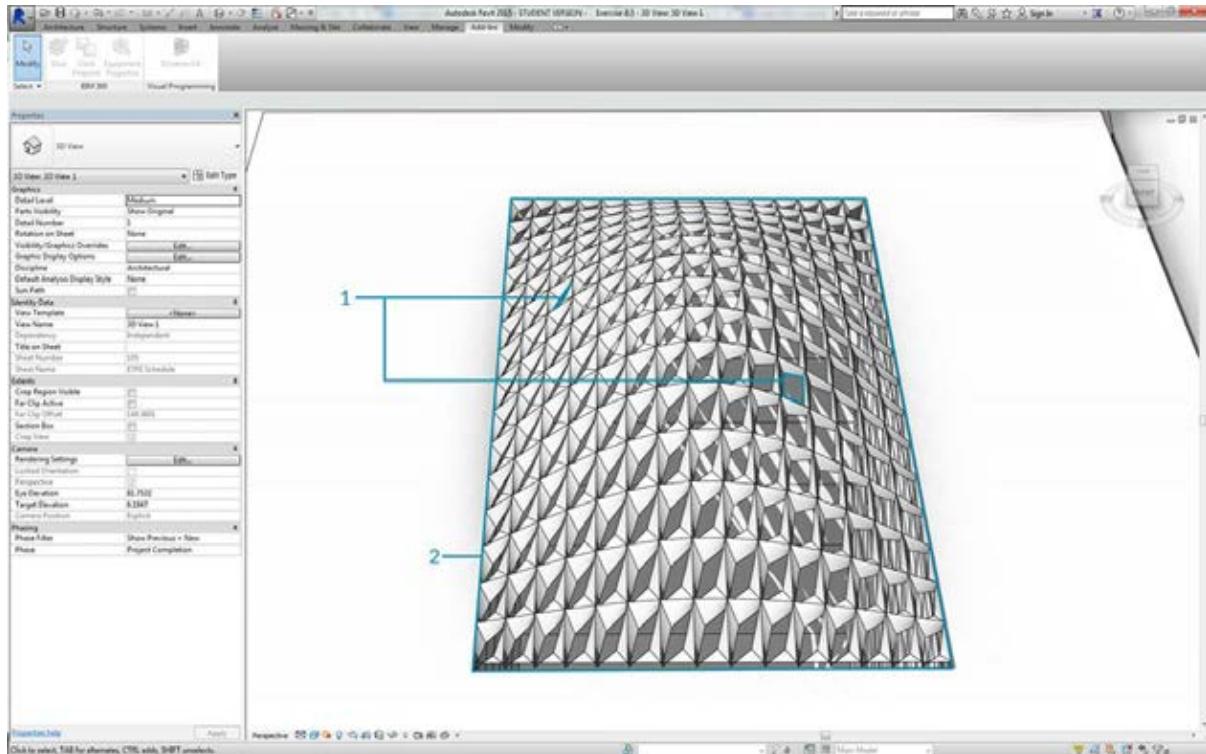
- 内積を確認すると、複数の数値が広範囲にわたって取得されています。これからその相対分布を使用するのですが、しかしそれらの数値を集約して[開口率]の適切な範囲に収めなければなりません。
- これにたいへん役立つツールが *Math.RemapRange* ノードです。そのノードにリストを入力し、その分布範囲を 2 つの目標値にマッピングし直します。
- 目標値を *Code Block* ノードで 0.15 と 0.45 として定義します。
- [実行]をクリックします。



- マッピングし直した値を *Element.SetParameterByName* ノードに接続します。
- そのノードの *parameterName* 入力に、「Aperture Ratio」という文字列を接続します。
- 同じノードの *element* 入力に、*AdaptiveComponent.ByPoints* ノードの *Adaptive Components* 出力を接続します。
- [実行]をクリックします。



Revitに戻って建物のマスを遠くから見てみると、ETFE パネルの開き方が太陽の向きによって変化していることが確認できます。



拡大表示すると、太陽により向き合っているパネルほどより閉じていることがわかります。太陽光の照射による過熱を抑えることがねらいです。太陽光をたくさん浴びている面ほど多く採光するように設定するには、ただ *Math.RemapRange* ノードで範囲を逆に切り替えるだけで済みます。

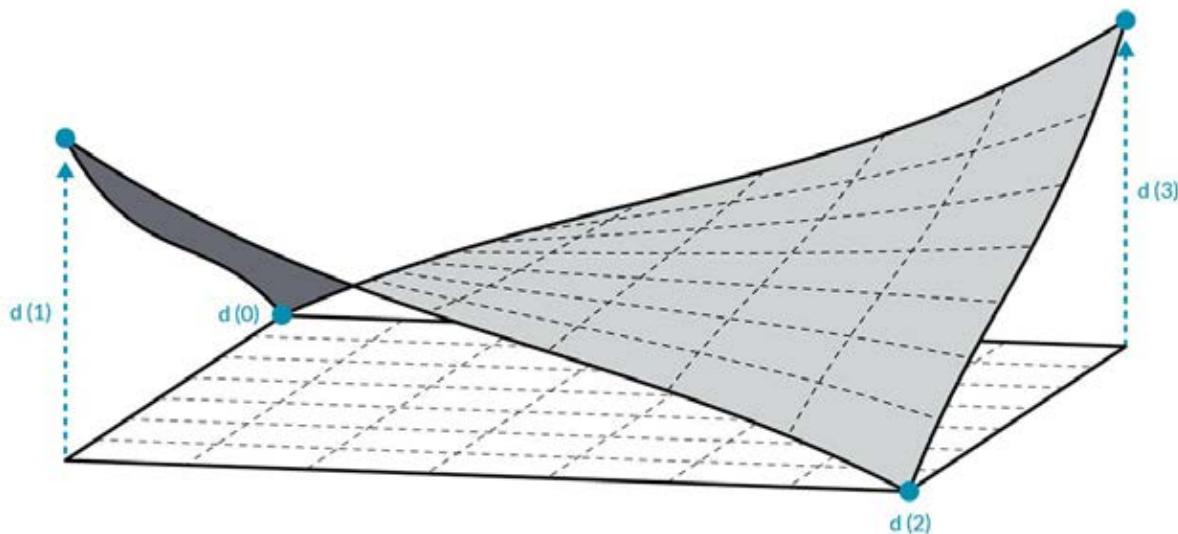
# 設計図書の作成

## 設計図書の作成

ここまでセクションの演習に続けて、パラメータを編集して設計図書を作成してみましょう。このセクションでは、パラメータを編集することで、要素のジオメトリの特性を左右するのではなく、Revit ファイルから設計図書を作成できるようにする方法について紹介します。

### 偏差

以降の演習では、水平面からの基本的な偏差を使用して、設計図書作成用に Revit のシートを作成します。パラメータで定義した屋根構造上のパネルにはそれぞれ異なる偏差の値が与えられています。そこで、色分けによって値の範囲をわかりやすく表示し、アダプティブ点を集計表に書き出してファサード設計の監修者、設計者、または施工業者に渡すことができるようになります。



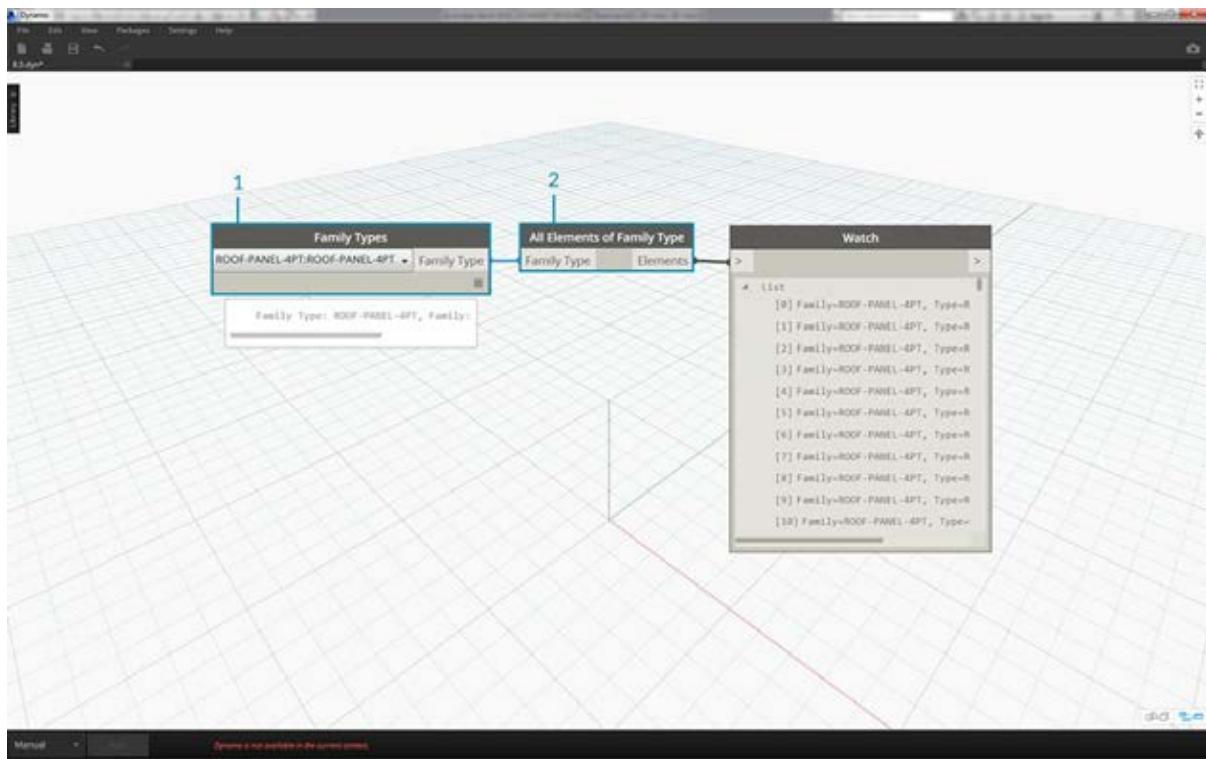
水平面からの偏差を取得するノードにより、4つの点群と最適な水平面の間の距離が計算されます。これで施工性をすばやく簡単に検討することができます。

### 演習

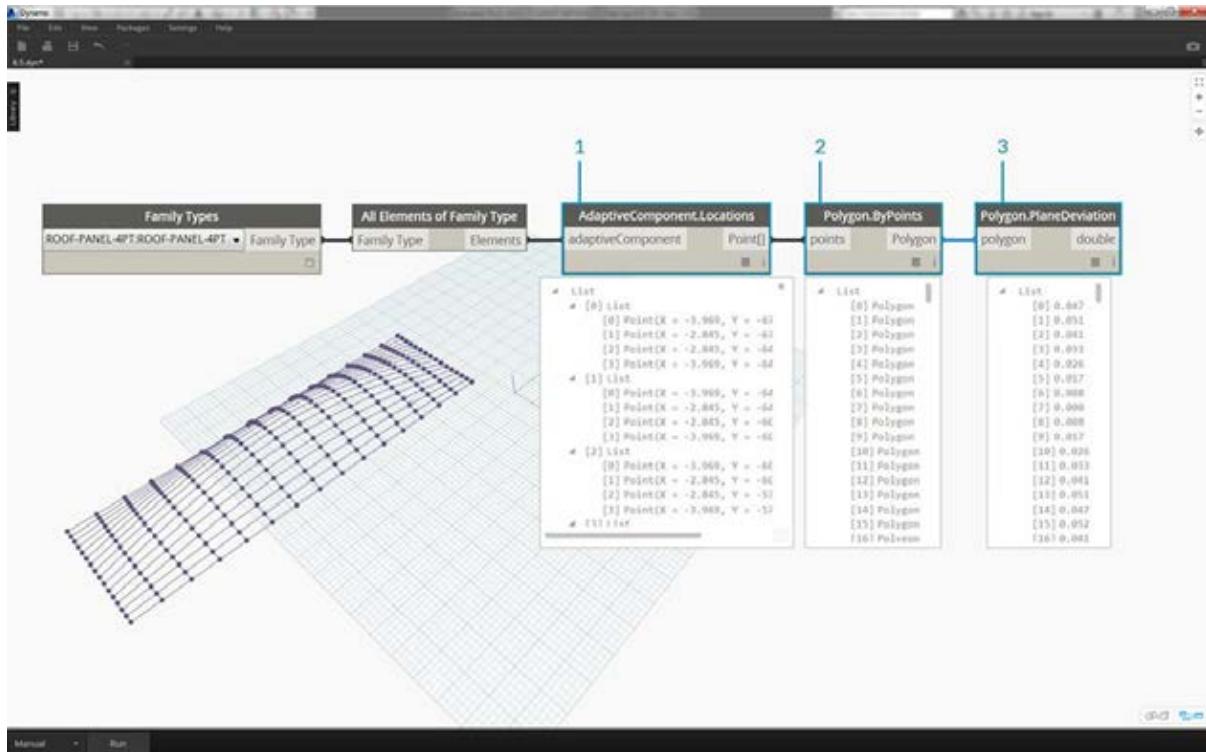
この演習に付属しているサンプル ファイルをダウンロードしてください(右クリックして[名前を付けてリンク先を保存]を選択)。すべてのサンプル ファイルの一覧については、付録を参照してください。

1. [Documenting.dyn](#)
2. [ARCH-Documenting-BaseFile.rvt](#)

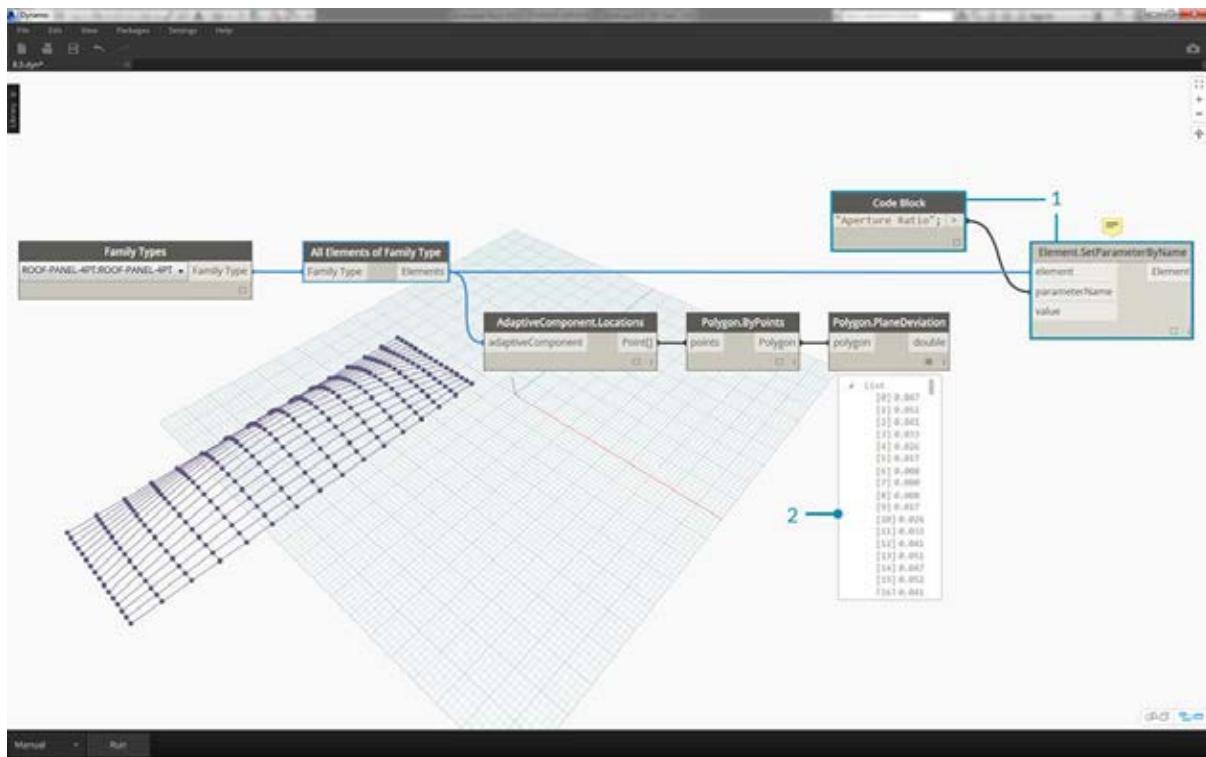
このセクション用の Revit ファイルを使用して(または前のセクションからの続きとして)演習を開始しましょう。このファイルには、屋根上の ETFE パネルの配列が収録されています。以降の演習でこれらのパネルを参照します。



1. *Family Types* ノードをキャンバスに追加し、[ROOF-PANEL-4PT]を選択します。
2. このノードを *All Elements of Family Type* ノードに接続することで、すべての要素を Revit から Dynamo に取得します。

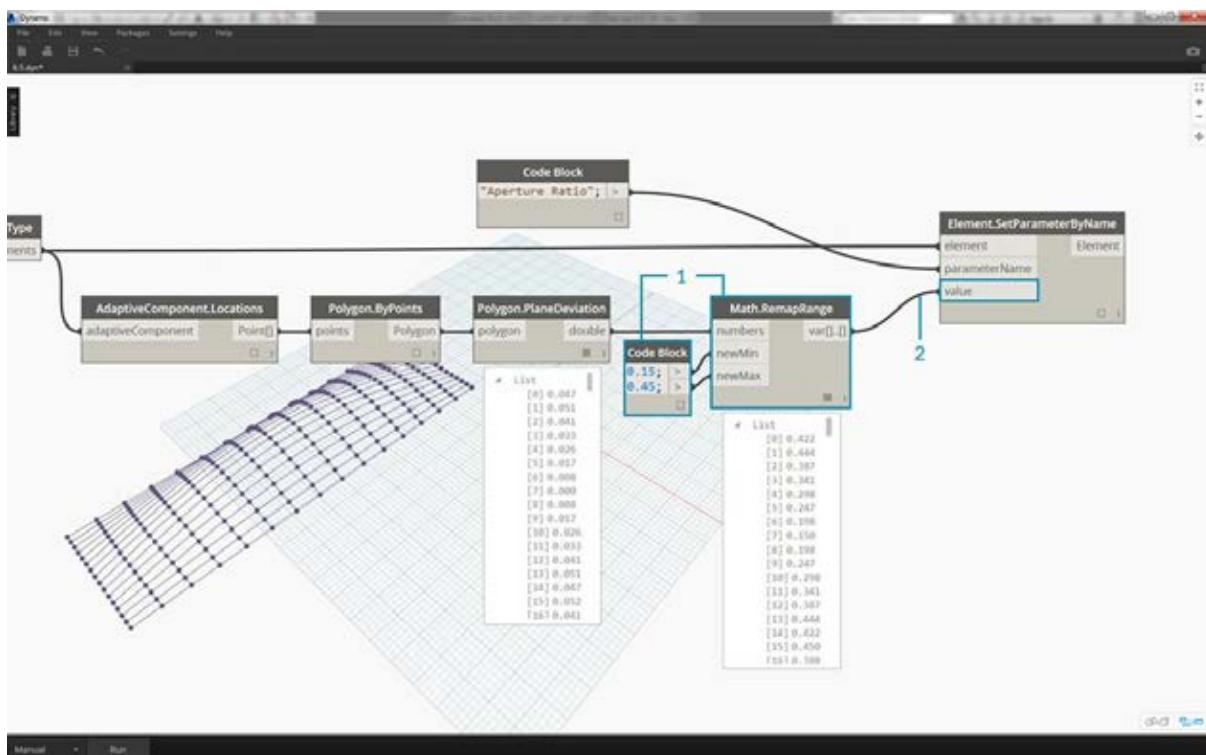


1. *AdaptiveComponent.Locations* ノードにより、各要素のアダプティブ点の位置をクエリします。
2. *Polygon.ByPoints* ノードを使用して、これら 4 点から 1 つのポリゴンを作成します。これにより、Revit 要素のジオメトリをすべて読み込むことなく、パネル システムの抽象化されたバージョンを Dynamo で取得することができます。
3. *Polygon.PlanarDeviation* ノードを使用して、水平面からの偏差を計算します。

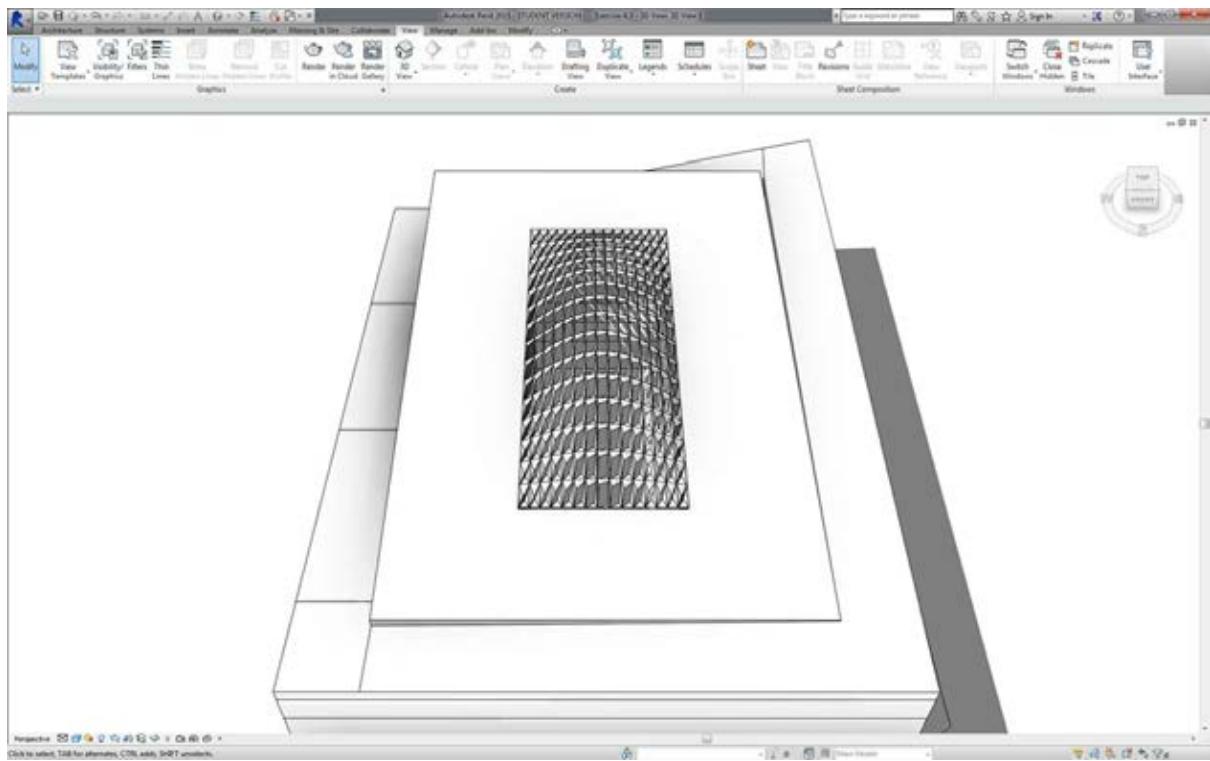


前の演習と同様に、各パネルの開口率を水平面からの偏差に基づいて設定してみましょう。

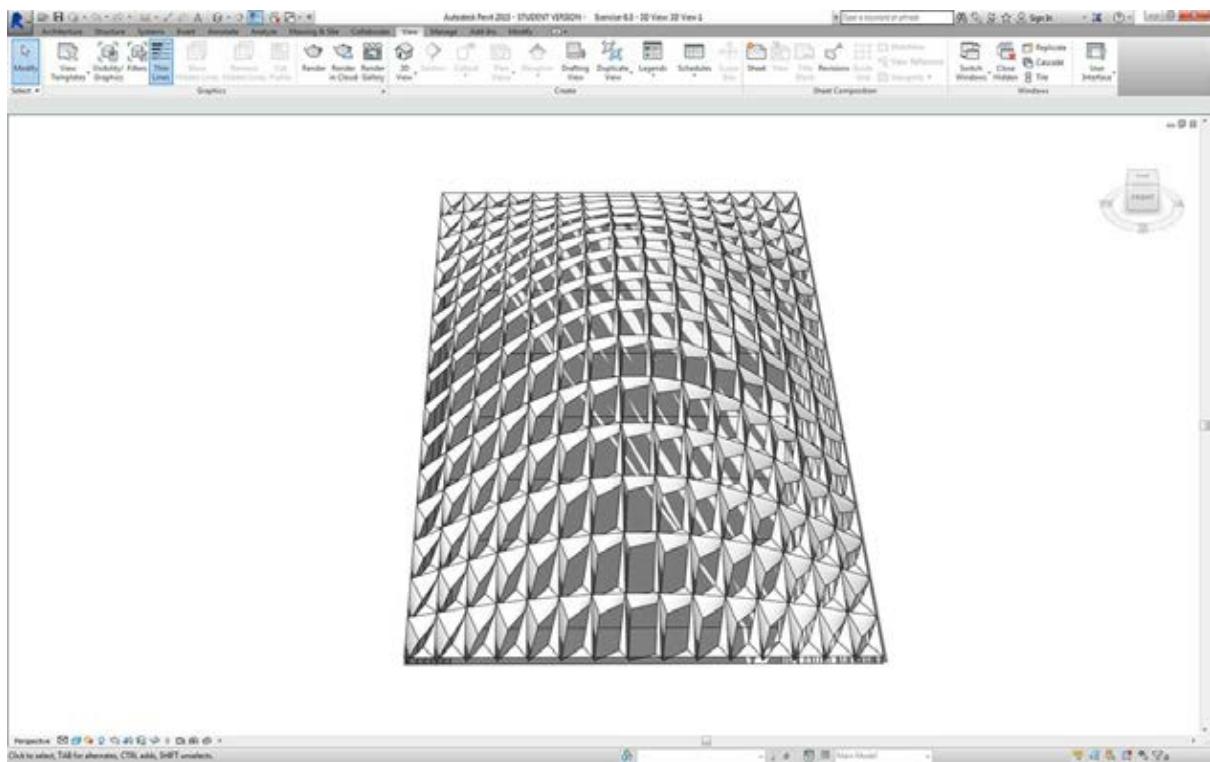
1. *Element.SetParameterByName* ノードをキャンバスに追加して、その *element* 入力にアダプティブコンポーネントを接続します。[開口率]を読み取っている *Code Block* ノードを、*parameterName* 入力に接続します。
2. 偏差の出力を直接 *value* 入力に接続することはできません。なぜなら、複数の値をパラメータ範囲にマッピングし直す必要があるからです。



1. \**Math.RemapRange* ノードを使用して、偏差の値を .15 から .45\* までの範囲にマッピングし直します。
2. そのノードの出力を *Element.SetParameterByName* の *value* 入力に接続します。



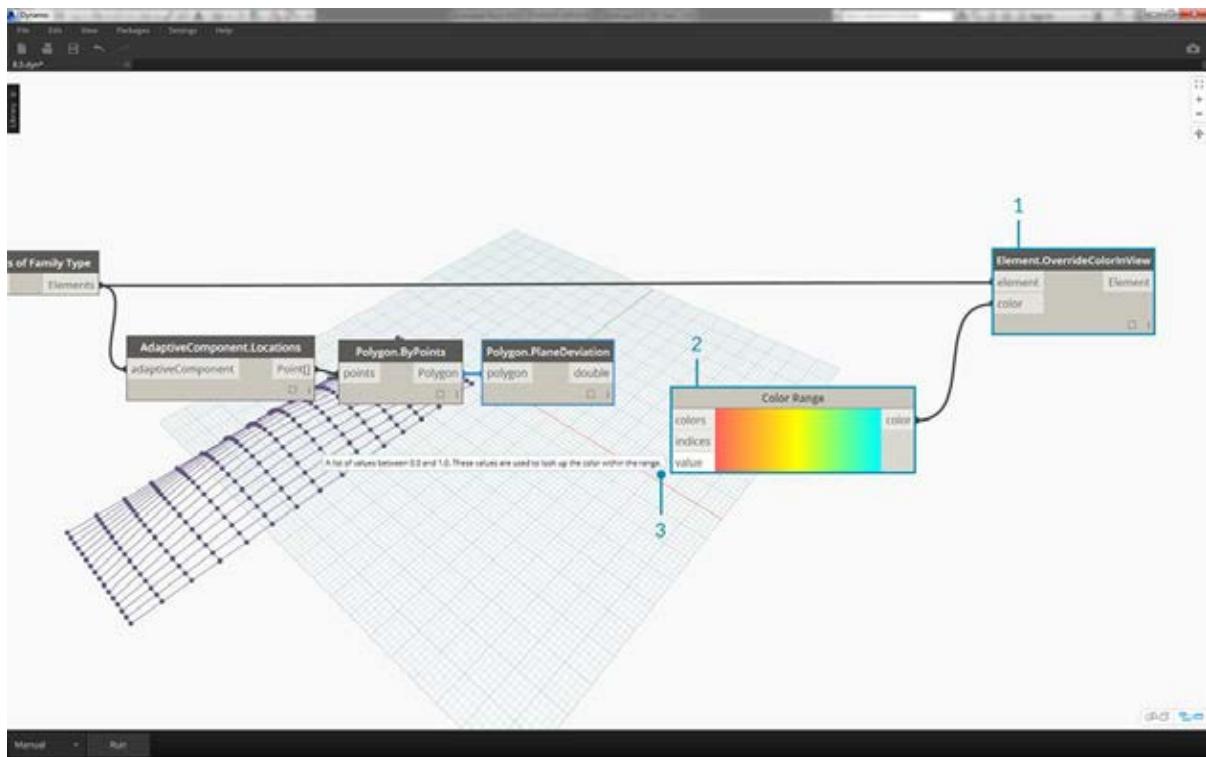
Revitに戻ると、サーフェス全体の開口率が多少変化したことがわかります。



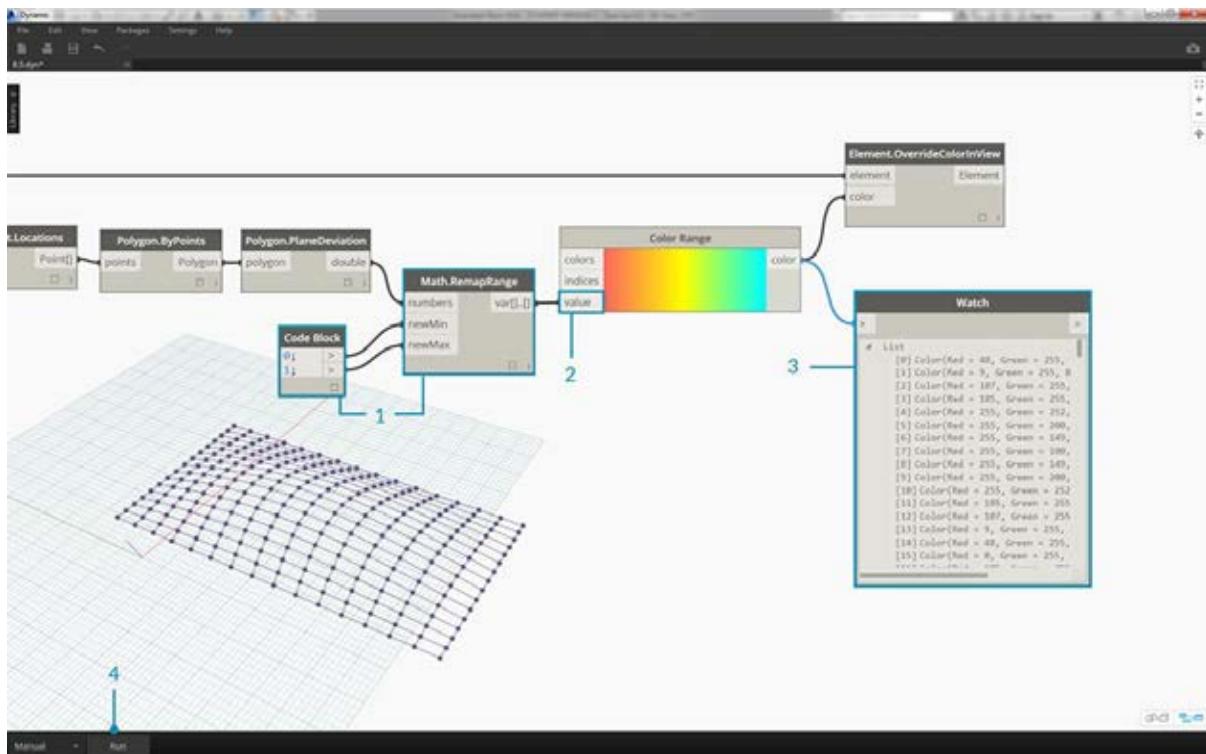
拡大表示するとはっきりわかるように、サーフェスの四隅に近付くほどパネルが閉じていく傾向にあり、また隆起の頂点へ近付くほどパネルが開いていく傾向にあります。これは、四隅のあたりでは水平面からの偏差が大きく、ふくらみの部分では水平に近くになっているためです。

### 色分けと設計図書作成

[開口率]の設定では、屋根上のパネルの偏差があまりよくわかりません。また、実際の要素のジオメトリが変更されてしまいます。単に製造性の観点から偏差を検討するだけであれば、設計図書作成の際に、偏差の範囲に基づいてパネルを色分けするとよいでしょう。下記の一連の手順によってそのような色分けを行うことができます。これは上記の手順にとてもよく似ています。

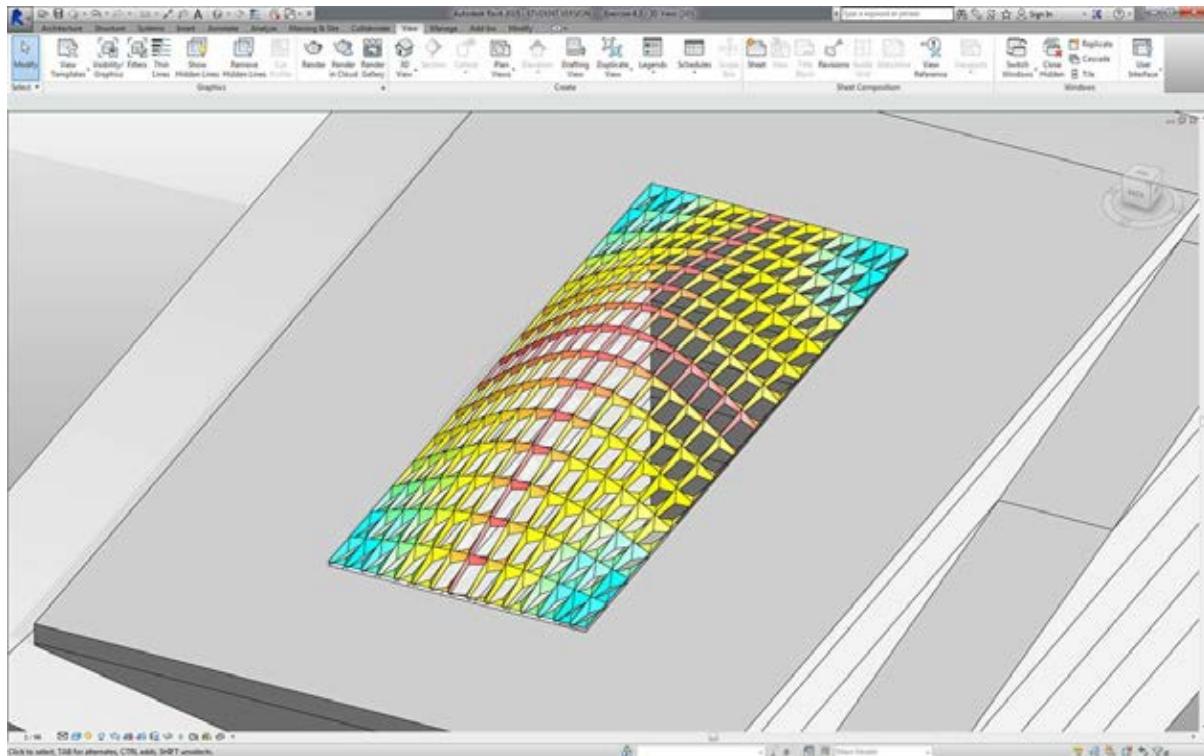


1. *Element.SetParameterByName* ノードを削除し、*Element.OverrideColorInView* ノードを追加します。
2. *Color Range* ノードをキャンバスに追加して、そのノードを *Element.OverrideColorInView* の *color* 入力に接続します。さらに、グラデーションを作成するために偏差の値を *Color Range* ノードに接続する必要があります。
3. *value* 入力にカーソルを合わせると、その入力の値が 0 から 1 までの範囲で表示されます。この値は、値ごとに色をマッピングするのに使用されます。偏差の値をこの範囲にマッピングし直す必要があります。

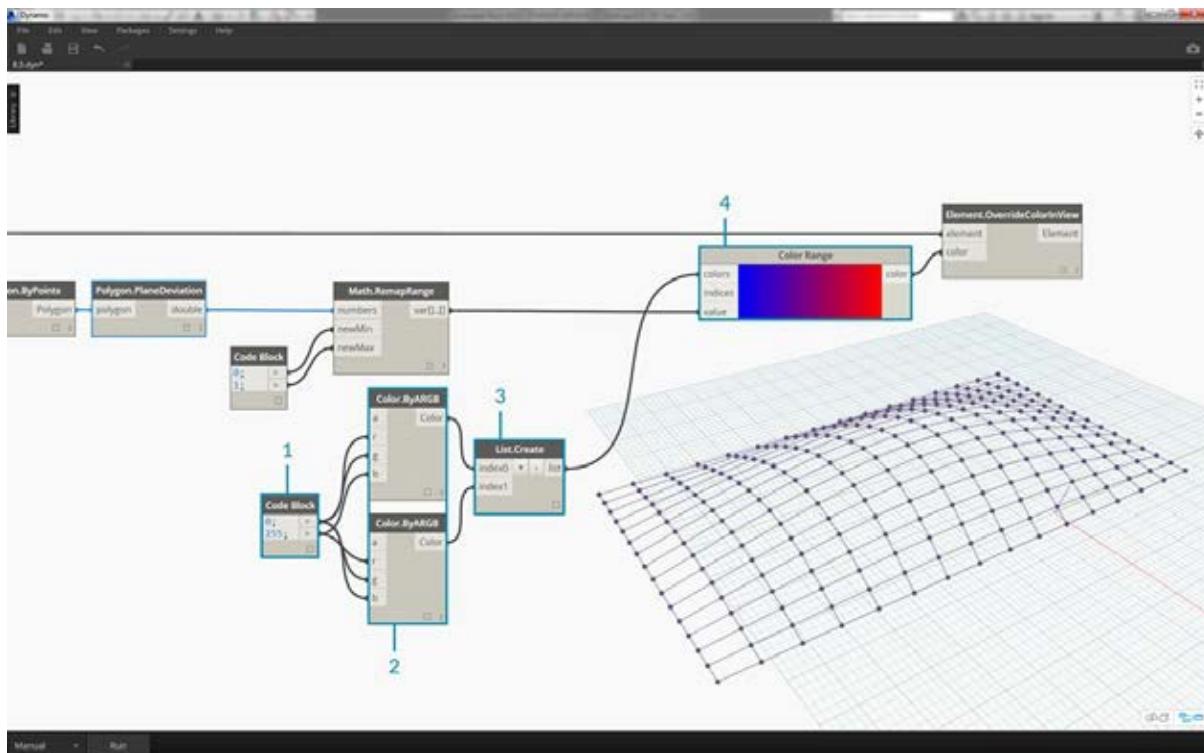


1. *Math.RemapRange* を使用して、水平面からの偏差を 0 から 1 までの範囲にマッピングし直します(注記: なお、*MapTo* ノードを使用してソースの範囲を設定することもできます)。
2. その出力結果を *Color Range* ノードに接続します。
3. ここでの出力は、数値の範囲ではなく、色の範囲です。
4. [手動]に設定している場合は[実行]をクリックします。これ以降の手順では、[自動]に設定しないように注意してください

い。

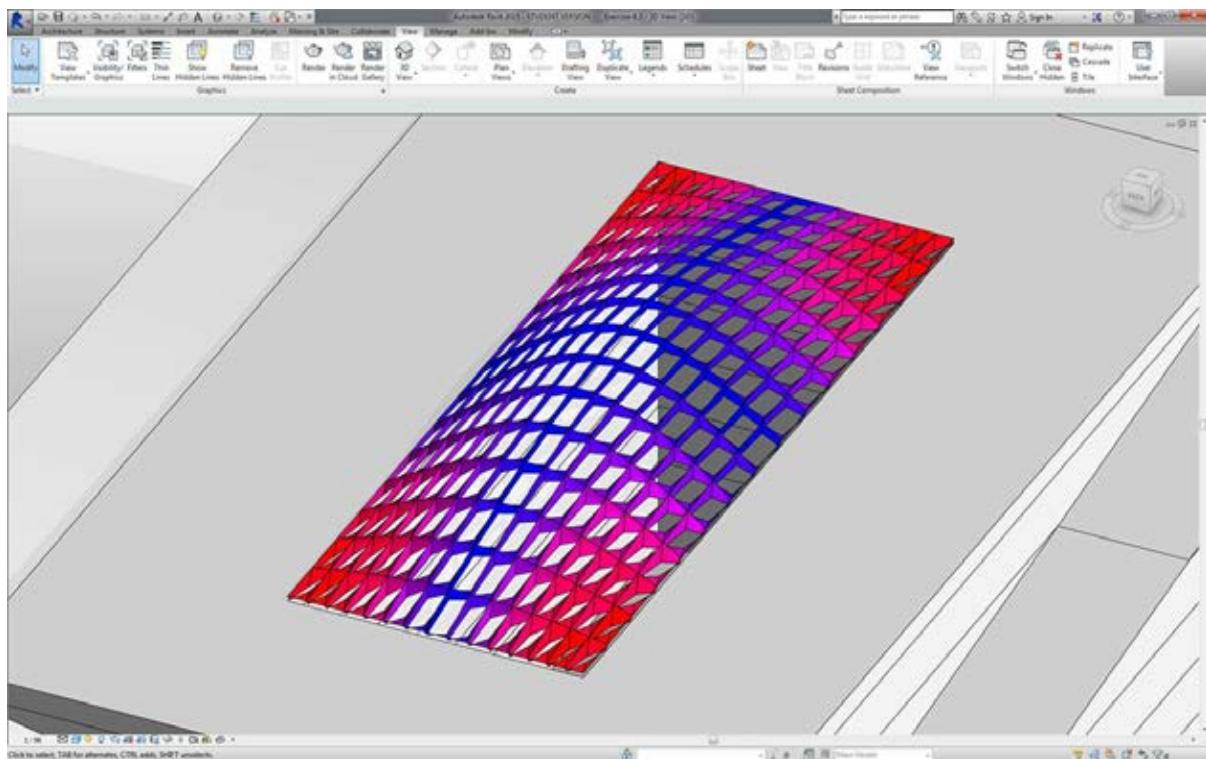


Revitに戻ると、かなり見やすいグラデーションが表示されます。これは、ユーザが指定した色の範囲に基づいて、水平面からの偏差を表しています。色分けをカスタマイズするには、どうすればよいでしょうか。いま偏差の最小値は赤色で表示されていますが、これとは逆の色分けに変更してみましょう。つまり、偏差の最大値を赤色に、偏差の最小値をもっと落ちついた色に設定することにします。Dynamoに戻ってこの修正を行ってみましょう。



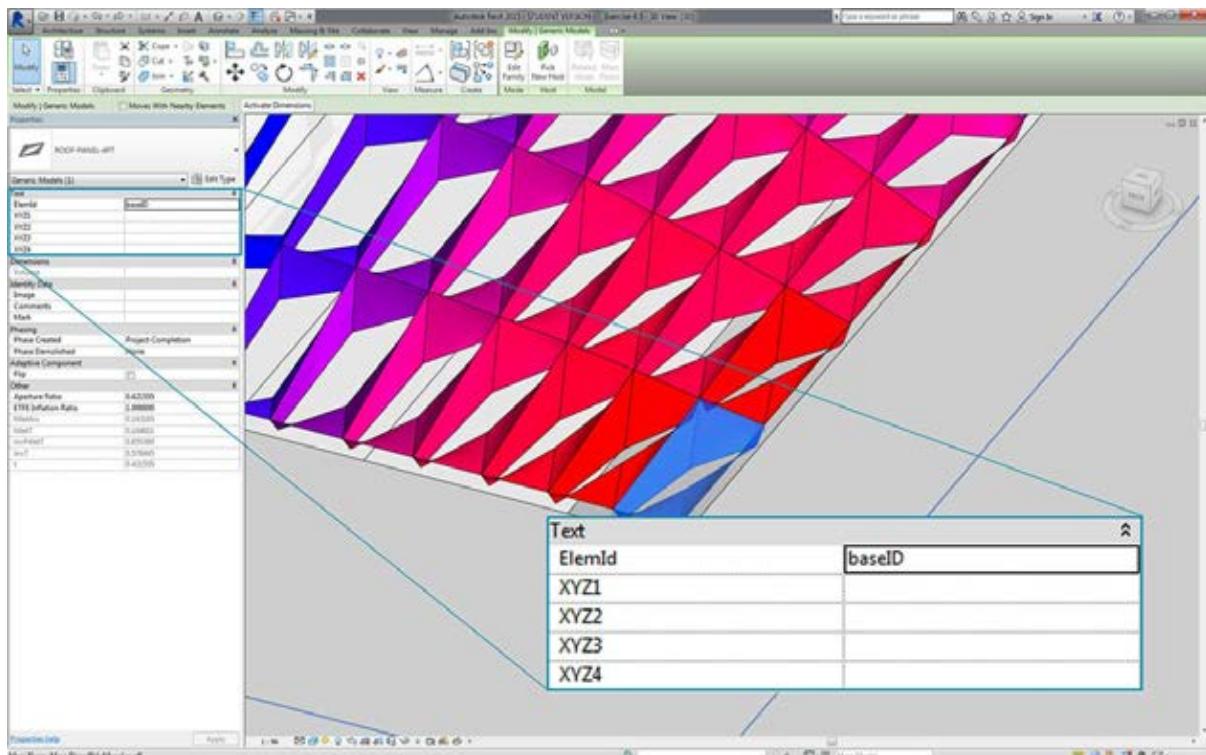
1. *Code Block* ノードを使用して、0; と 255; という 2 つの数値を、2 行に分けて追加します。
2. 2 つの *Color.ByARGB* ノードに適切な値を接続することで、赤色と青色を作成します。
3. これらの 2 色から 1 つのリストを作成します。
4. このリストを *Color Range* ノードの *colors* 入力に接続し、カスタマイズした色の範囲が更新されていることを確認します。

す。



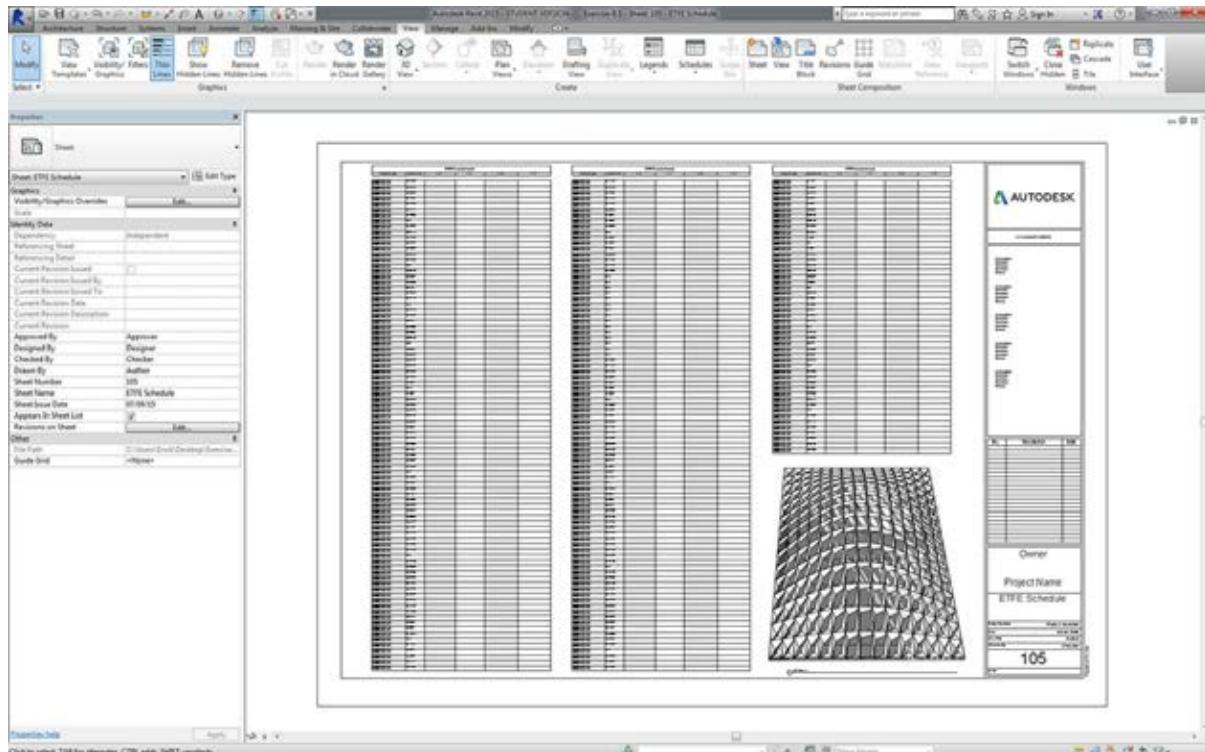
Revitに戻ると、水平面からの偏差が四隅の領域で最大になっていることがよりはっきり確認できます。なお、このノードはビュー内の色の優先設定に使用されます。したがって、一連の図面のなかで特定のシートが特定のタイプの解析を目的としている場合に、とても役に立ちます。

## 集計表の作成

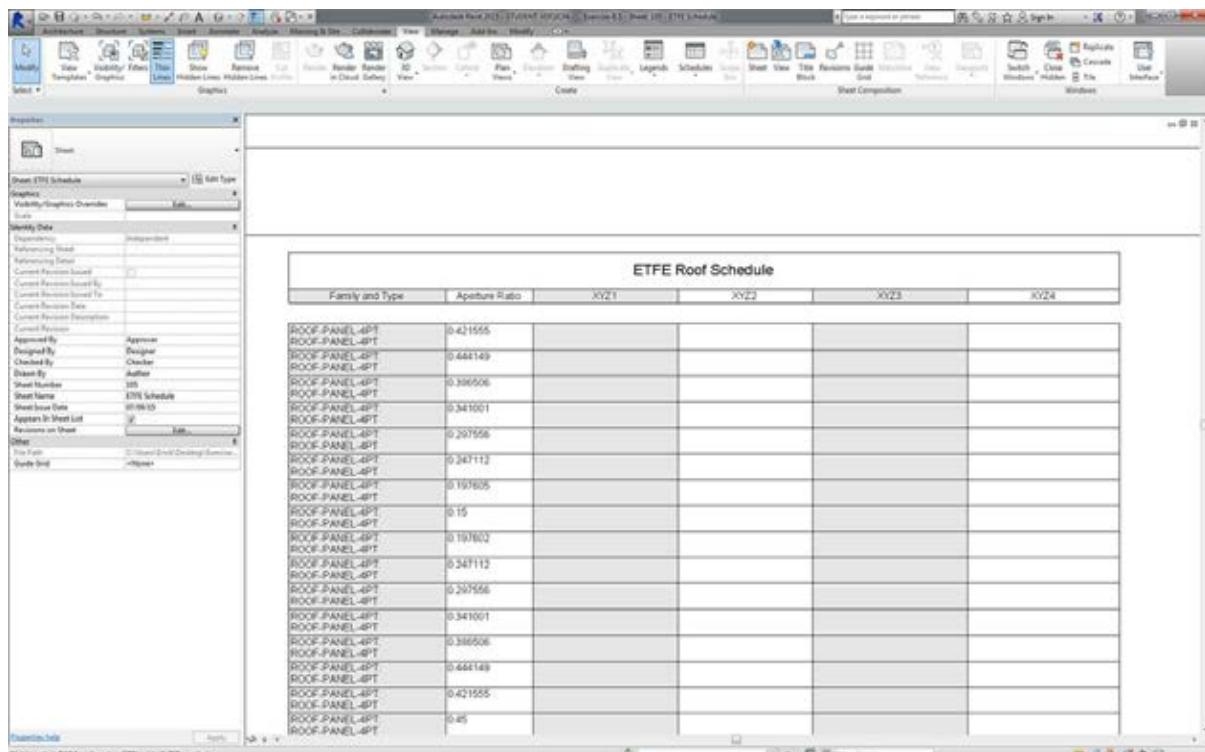


1. Revit で ETFE パネルを選択すると、XYZ1、XYZ2、XYZ3、\*\*XYZ4 という 4 つのインスタンス パラメータが表示されます。正しく作成されている場合、これらのパラメータはすべて空になっています。これらは文字ベースのパラメータであり、値を必要とします。Dynamo を使用して、各パラメータにアダプティビティ点の位置を入力します。この機能は、ジオメトリを

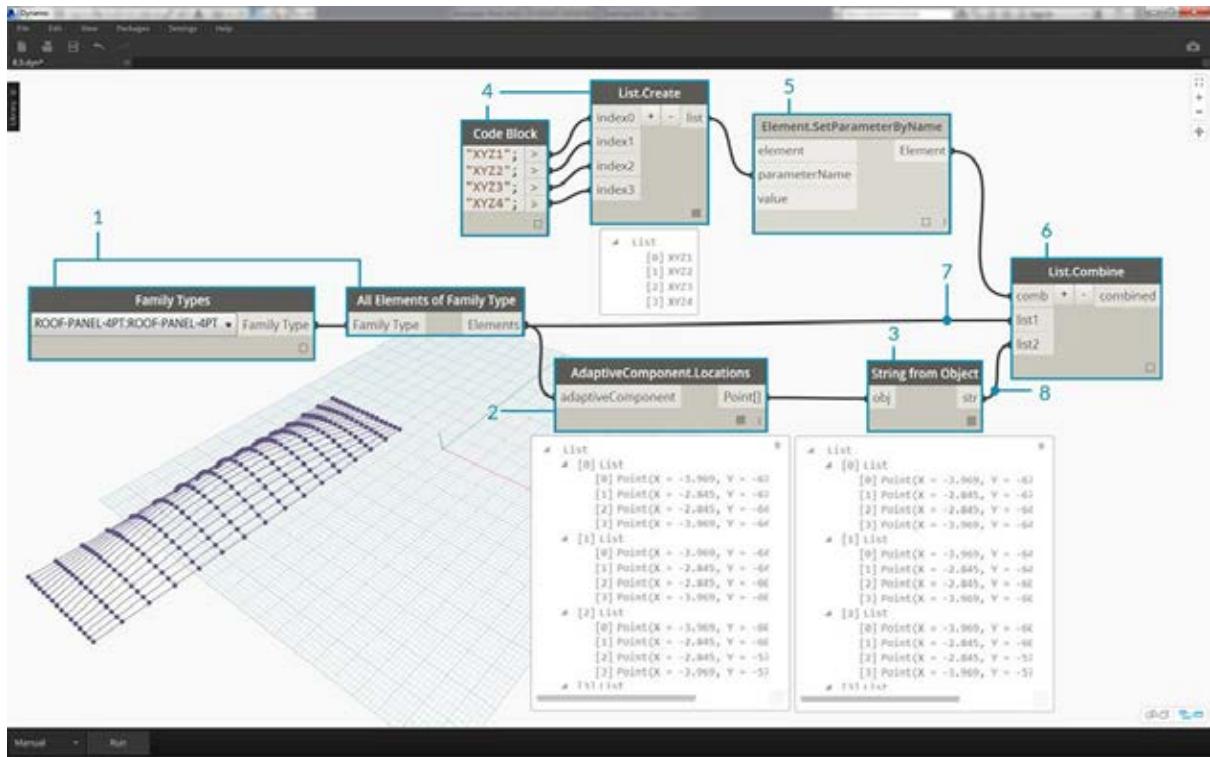
ファサード設計の監修者に送信する必要がある場合に、相互運用性の確保に役立ちます。



サンプルのシートには大規模な空の集計表が含まれています。XYZ パラメータは Revit ファイルでも使用される共有パラメータであり、このファイルによってパラメータを集計表に追加することができます。

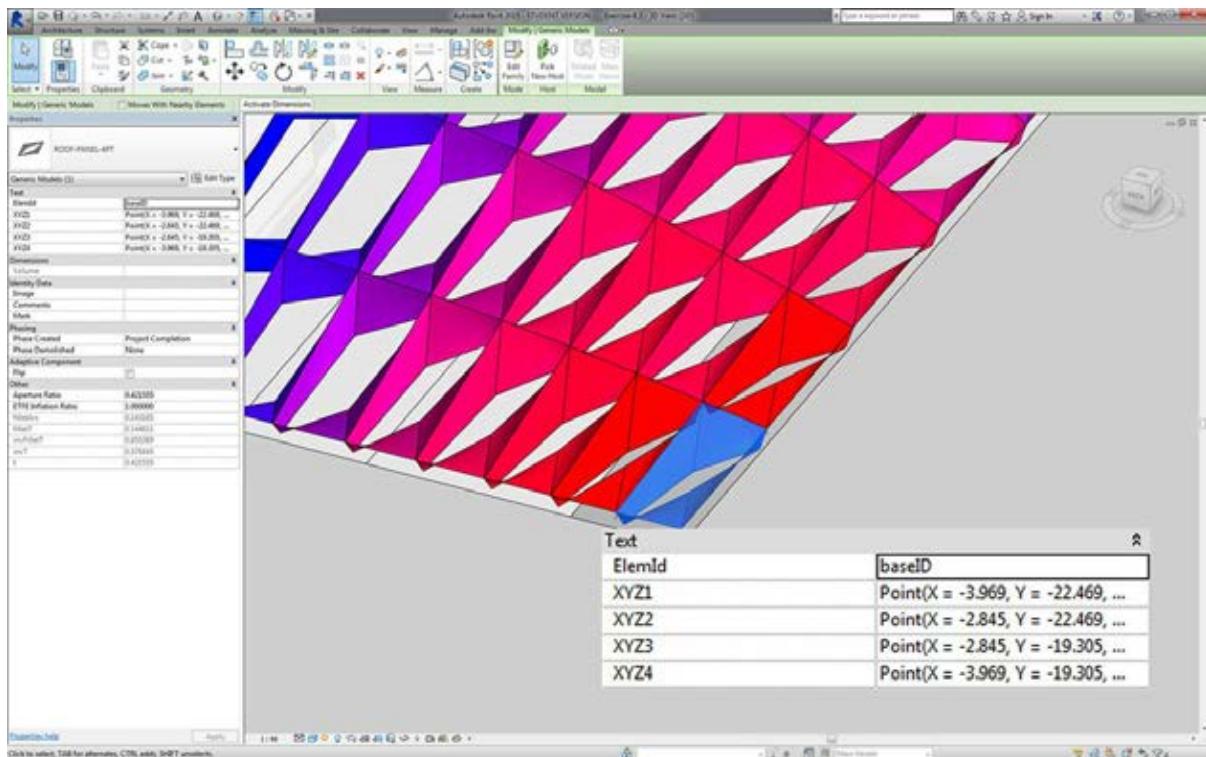


拡大表示すると、XYZ パラメータはまだ入力されていません。左側 2 つのパラメータは Revit によって処理されています。

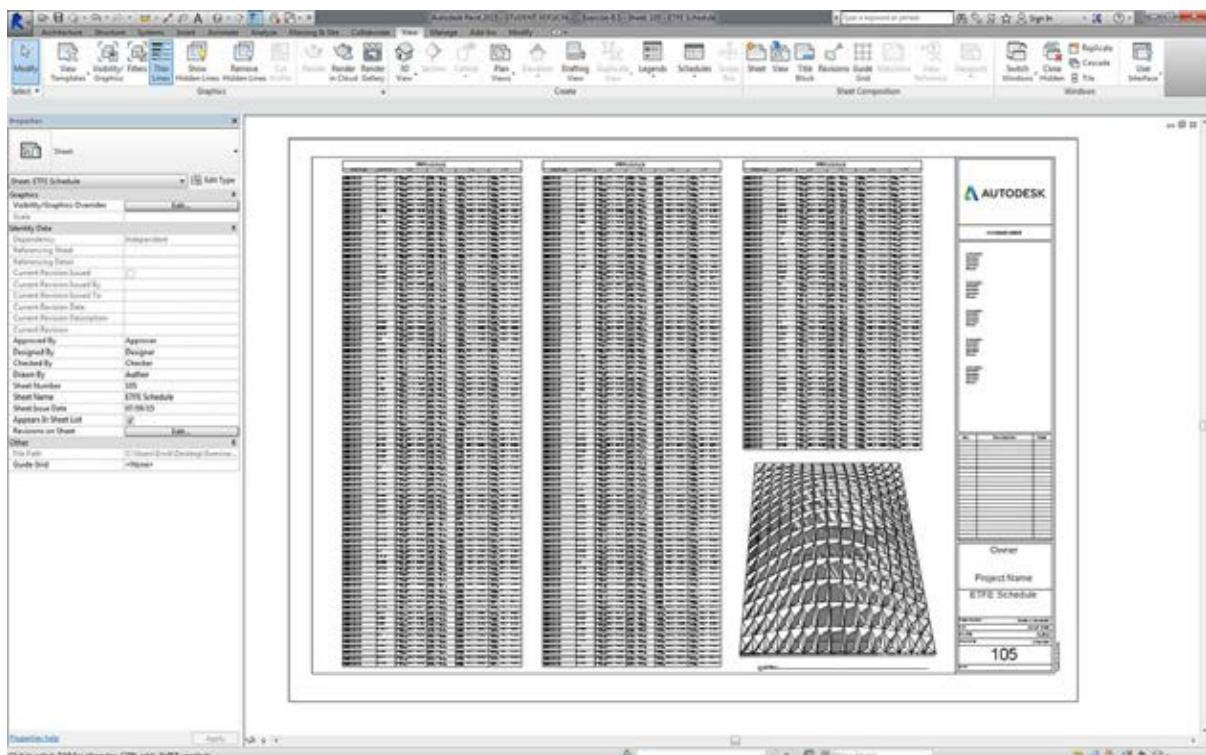


これらのパラメータに値を入力するために、これから複雑なリスト操作を行います。グラフそれ自体は単純ですが、考え方にはリストの章で紹介したリストのマッピングを大いに活用しています。

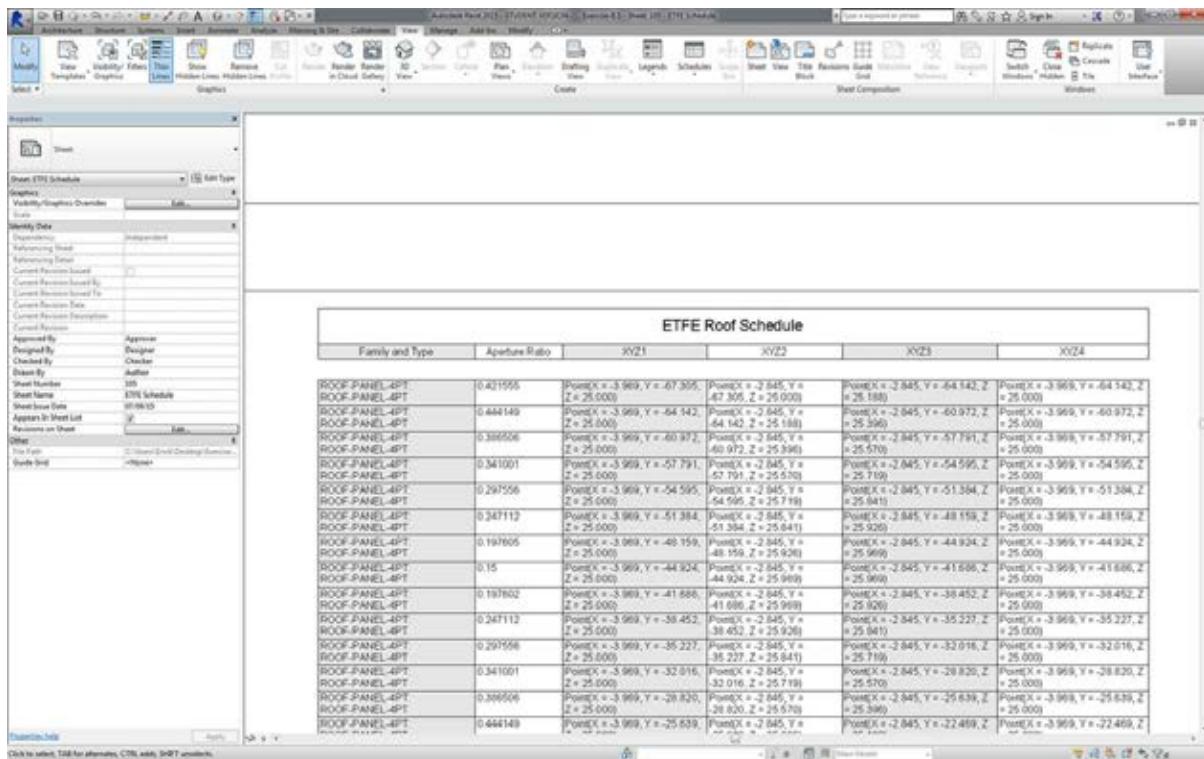
1. 2つのノードを使用してアダプティブコンポーネントをすべて選択します。
2. *AdaptiveComponent.Locations* ノードを使用して、各点の位置を抽出します。
3. これらの点群を文字列に変換します。なお、パラメータはテキストベースですから、正しいデータタイプを入力する必要があることに注意してください。
4. 変更するパラメータを定義する4つの文字列 XYZ1, XYZ2, XYZ3, \*\*XYZ4 から、1つのリストを作成します。
5. このリストを *Element.SetParameterByName* ノードの *parameterName* 入力に接続します。
6. *Element.SetParameterByName* ノードを *List.Combine* ノードの *combinator* 入力に接続します。
7. アダプティブコンポーネントを *list1* 入力に接続します。
8. *String from Object* ノードを *list2* 入力に接続します。
9. ここでリストマッピングを行います。各要素につき4つのパラメータに値を入力することで、複雑なデータ構造を作成するためです。*List.Combine* ノードはデータ階層内の1段階下の層で操作を定義します。element 入力と value の入力が空のままになっているのはこのためです。*List.Combine* ノードは、入力のサブリストを、接続された順番に基づいて *List.SetParameterByName* ノードの空の入力に接続します。



Revit でパネルを選択すると、各パラメータに文字列値が入力された状態で表示されます。実際のプログラミングでは、(X,Y,Z) のようにより単純な形式で 1 つの点を作成するものです。これは Dynamo の文字列操作で可能ですが、この章で取り扱う範囲から逸脱しないようにするために、その方法はここでは紹介しません。



パラメータへの入力が完了しているサンプル集計表のビューです。



各 ETFE パネルを構成するすべてのアダプティブ点について XYZ 座標が記入されています。これらが製造用の各パネルの四隅を表します。

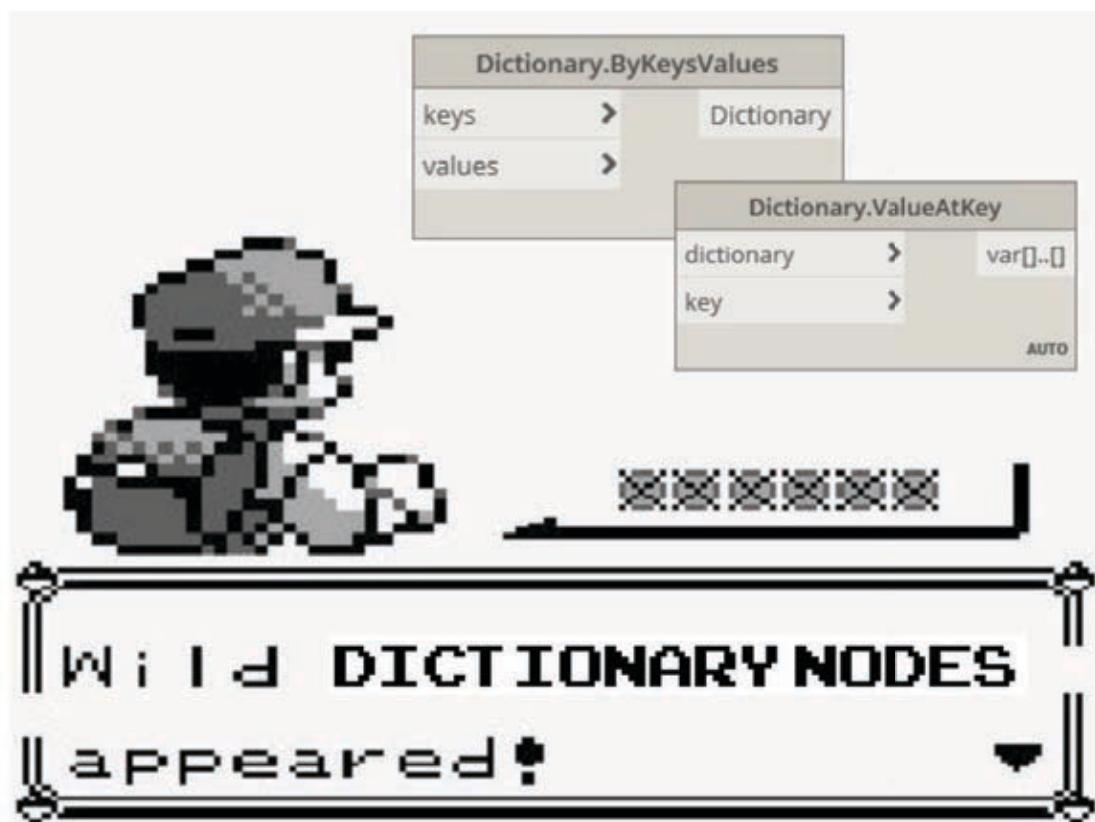
## Dynamo のディクショナリ

### Dynamo のディクショナリ

ディクショナリは、キーと呼ばれる別のデータと関連付けられているデータの集まりです。ディクショナリでは、データを検索、削除、および挿入する機能が公開されています。

基本的には、ディクショナリを非常にスマートな検索方法と考えることができます。

ディクショナリの機能はこれまで *Dynamo* 内で利用可能でしたが、*Dynamo 2.0* では、このデータ タイプを管理する新しい方法を導入しました。



画像は [sixtysecondrevit.com](http://sixtysecondrevit.com) 提供

## ディクショナリ

## ディクショナリ

Dynamo 2.0 では、ディクショナリのデータ タイプをリストのデータ タイプと分離する概念が導入されました。この変更により、ワークフローにおけるデータの作成や操作の方法が大幅に変更される可能性があります。2.0 よりも前のバージョンでは、ディクショナリとリストは 1 つのデータ タイプとして統合されていました。つまり、リストが実際には整数キーを持つディクショナリだったのです。

- ディクショナリとは

ディクショナリは、キーと値のペアの集合で構成されたデータ タイプで、各キーは各集合に固有です。ディクショナリは順序付けされておらず、基本的には、リストにあるようなインデックス値の代わりにキーを使用して「調べる」ことができます。Dynamo 2.0 では、キーに文字列のみを使用できます。

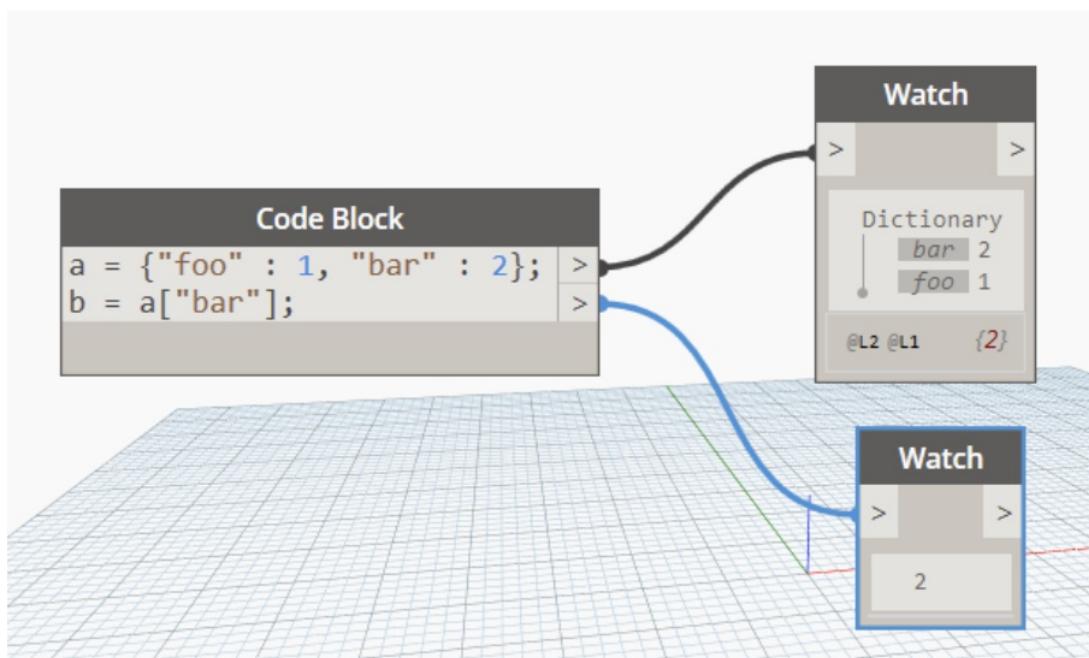
- リストとは

リストは、順序付けされた値の集合で構成されたデータ タイプです。Dynamo では、リストはインデックス値として整数を使用します。

- この変更を行った理由、および注意すべき理由

ディクショナリは、リストと分離されたことによって第一級オブジェクトとなりました。そのため、インデックス値を覚えたり、ワークフロー全体で厳密なリスト構造を維持することなく、値の格納や検索をすばやく容易に行うことができます。ユーザのテストにおいて、複数の GetItemAtIndex ノードの代わりにディクショナリを使用した場合に、グラフのサイズが大幅に低減することがわかりました。

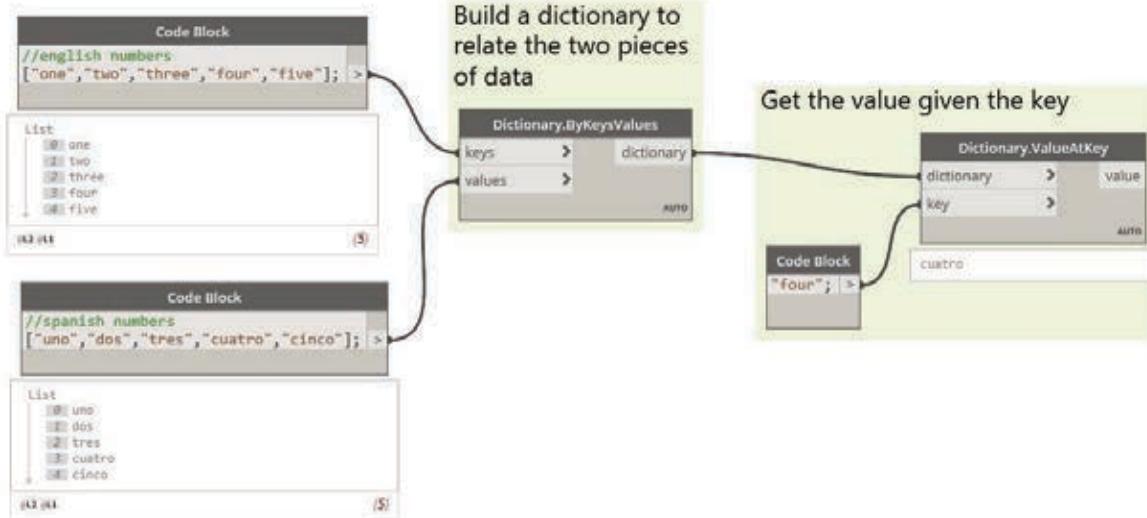
- 変更内容
- 構文の変更により、コード ブロック内のディクショナリとリストの初期化および操作の方法が変更されました。
- ディクショナリは { キー : 値 } の構文を使用します。
- リストは [ 値, 値, 値 ] の構文を使用します。
- ディクショナリを作成、編集、クエリーするための新しいノードが、ライブラリに追加されました。
- 1.x のコード ブロックで作成されたリストは、スクリプトのロード時に、角括弧 [ ] を波括弧 { } の代わりに使用する新しいリストの構文に自動的に移行されます。



- 注意すべき理由、使用する目的

コンピュータ サイエンスにおいて、ディクショナリはリストのように、オブジェクトの集合です。リストは特定の順序で並んでいますが、ディクショナリは順序なしの集合です。一連番号(インデックス)に依存せず、キーを使用します。

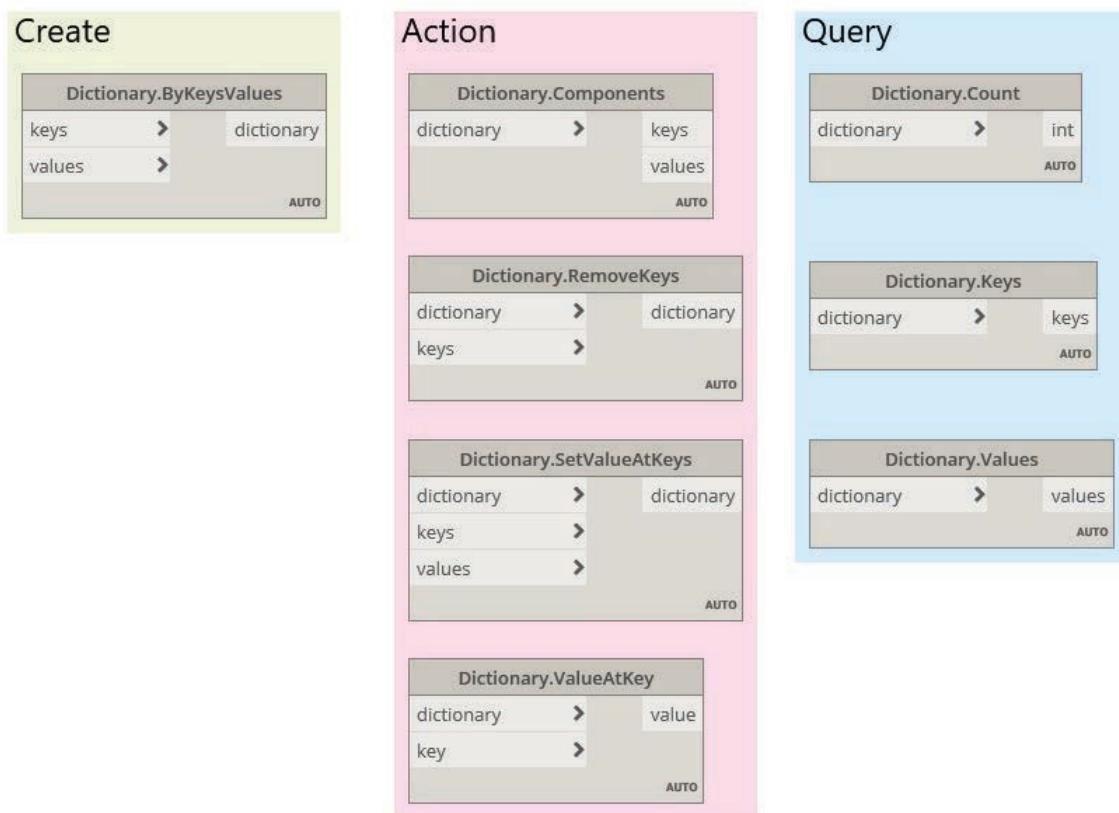
次の画像では、可能性のあるディクショナリの使用例を示しています。多くの場合、ディクショナリを使用して、直接的な関係を持たない2つのデータを関連付けます。ここでは、スペイン語バージョンの単語を英語バージョンに接続して、後で検索できるようにしています。



## [Dictionary]カテゴリのノード

### [Dictionary]カテゴリのノード

Dynamo 2.0 では[Dictionary]カテゴリのさまざまなノードが公開されており、使用することができます。作成、アクション、クエリーのノードがあります。



- Dictionary.ByKeysValues は、指定した値とキーでディクショナリを作成します(項目数は、最短のリストの入力によります)。
- Dictionary.Components は、入力ディクショナリのコンポーネントを作成します(これはノードの作成と逆の操作です)。
- Dictionary.RemoveKeys は、入力キーが削除された新しいディクショナリのオブジェクトを作成します。
- Dictionary.SetValueAtKeys は、入力キーと値に基づいて新しいディクショナリを作成し、対応するキーに対する現在の値を置き換えます。
- Dictionary.ValueAtKey は、入力キーに対する値を返します。
- Dictionary.Count は、ディクショナリに含まれるキーと値のペア数を示します。
- Dictionary.Keys は、現在ディクショナリに格納されているキーを返します。
- Dictionary.Values は、現在ディクショナリに格納されている値を返します。

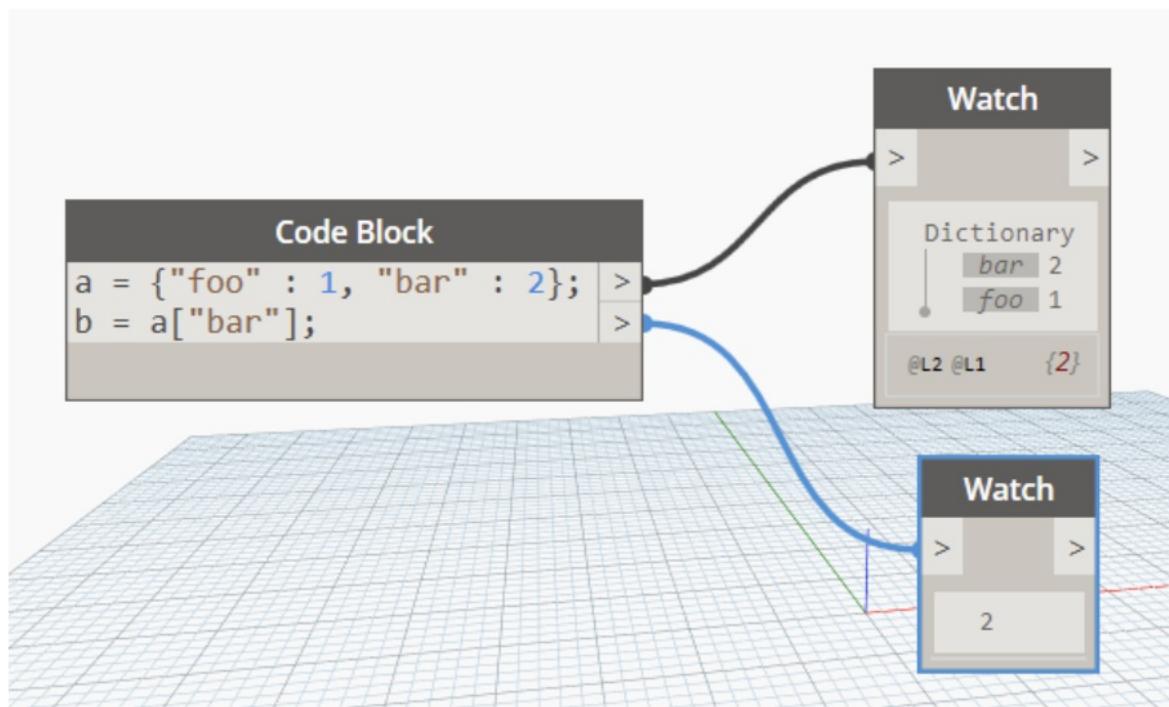
ディクショナリに関するあらゆるデータは、インデックスとリストを操作する従来の方法に代わる優れた手段です。

## コード ブロックにおけるディクショナリ

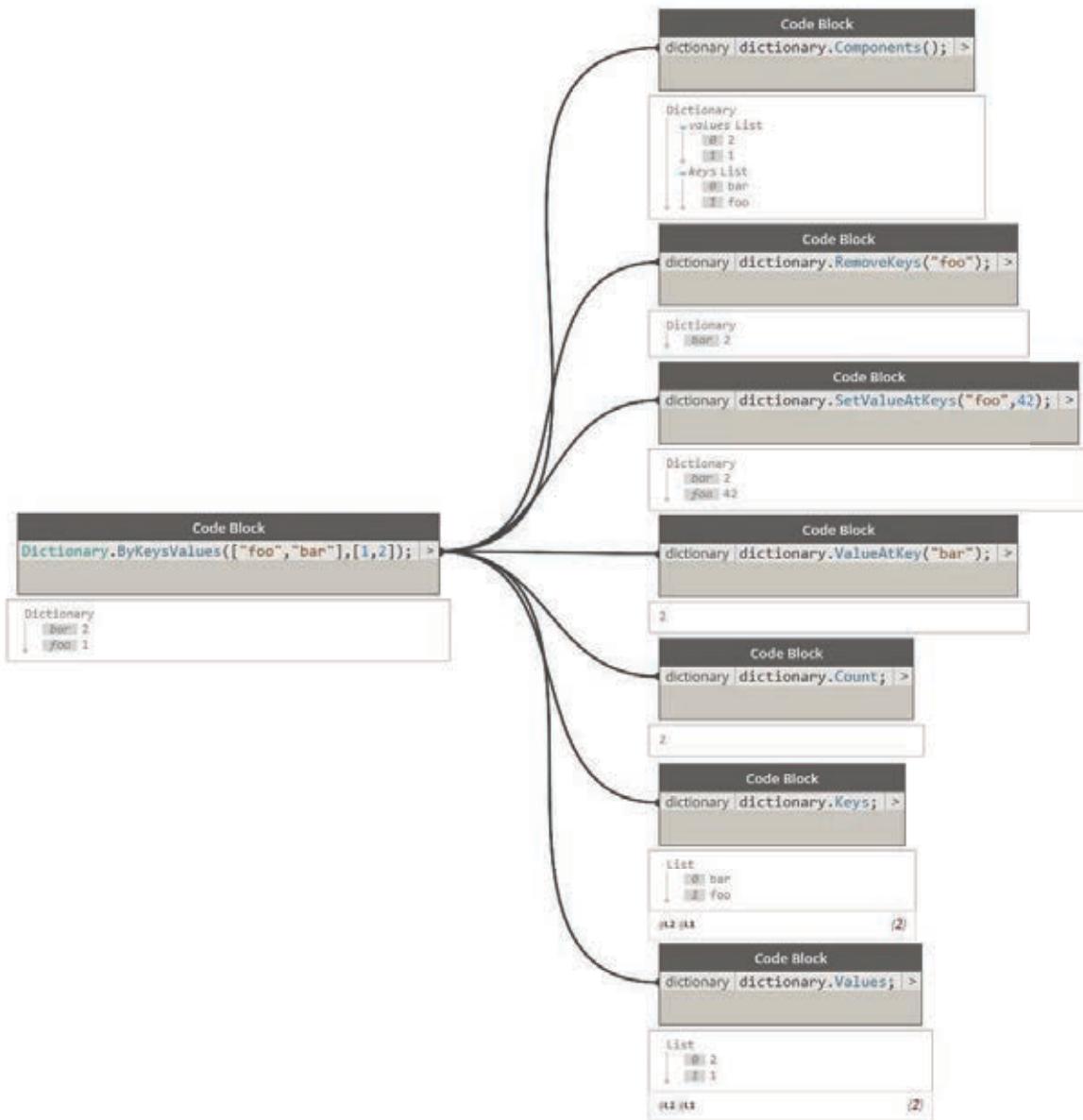
## コード ブロックにおけるディクショナリ

Dynamo 2.0 では、これまでに説明したディクショナリのノードだけでなく、コード ブロックにおけるディクショナリの新機能も導入されました。

ノードについて次のような、または DesignScript ベースの表現のような構文を使用できます。



ディクショナリは Dynamo のオブジェクト タイプであるため、次のアクションを実行できます。



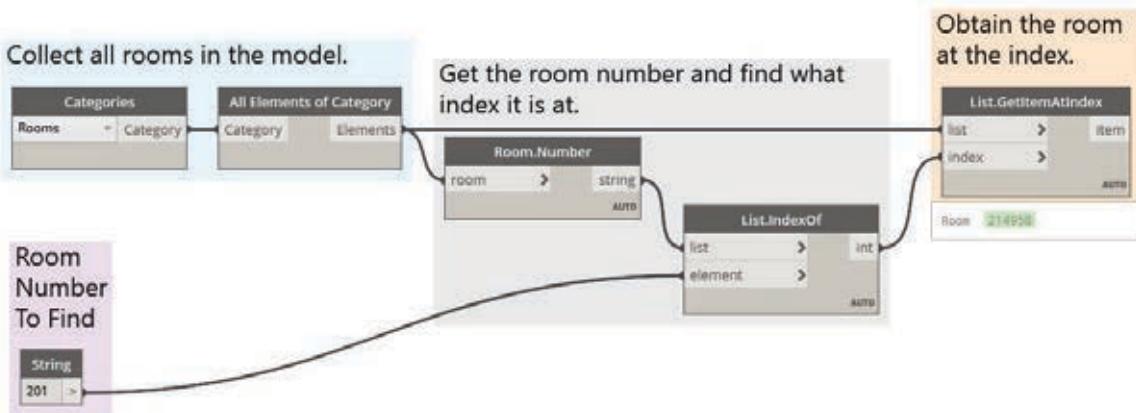
このような操作は、Revit データを文字列に関連付ける場合に特に便利です。次に、Revit での使用例を見てみましょう。

## ディクショナリ - Revit での使用例

### ディクショナリ - Revit での使用例

Revit 内で、そこに含まれるデータの一部を使って検索しようとしたことがありますか?

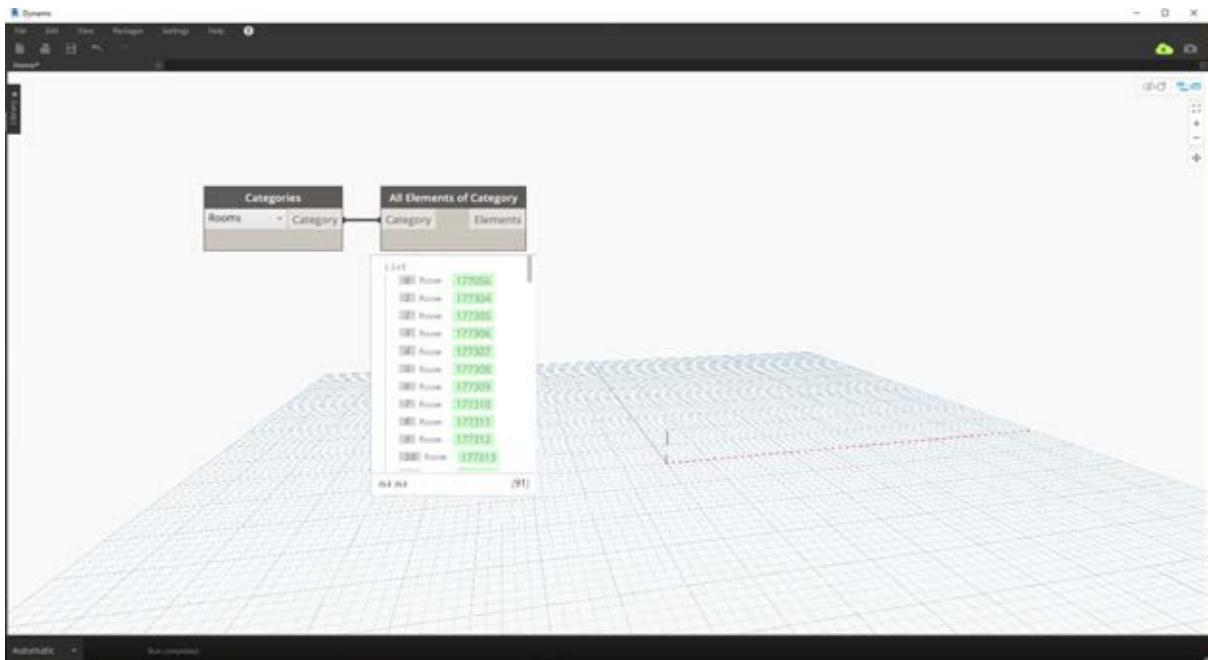
その際、次のような操作を行ったかもしれません。



上図では、Revit モデル内のすべての部屋を収集し、必要な部屋のインデックスを(部屋番号で)取得し、最終的にそのインデックスにある部屋がわかります。

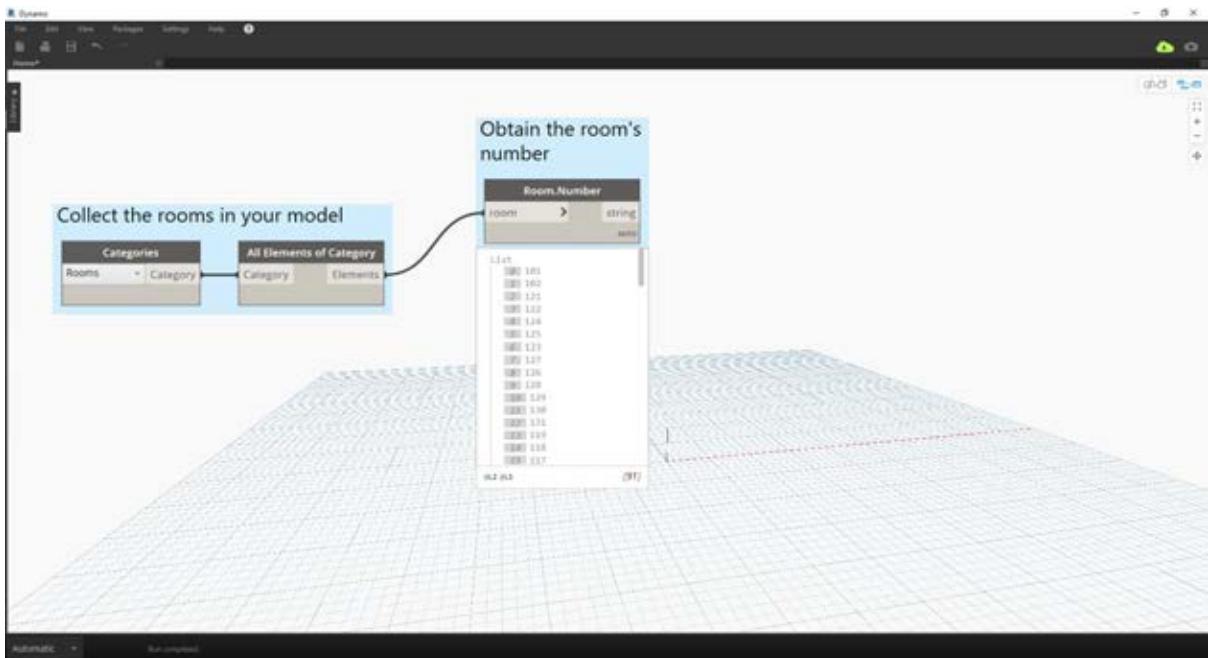
次に、ディクショナリを使用して、これを再作成してみましょう。

この演習に付属しているサンプル ファイルをダウンロードしてください(右クリックして[名前を付けてリンク先を保存]を選択):  
[RoomDictionary.dyn](#)。すべてのサンプルファイルの一覧については、付録を参照してください。



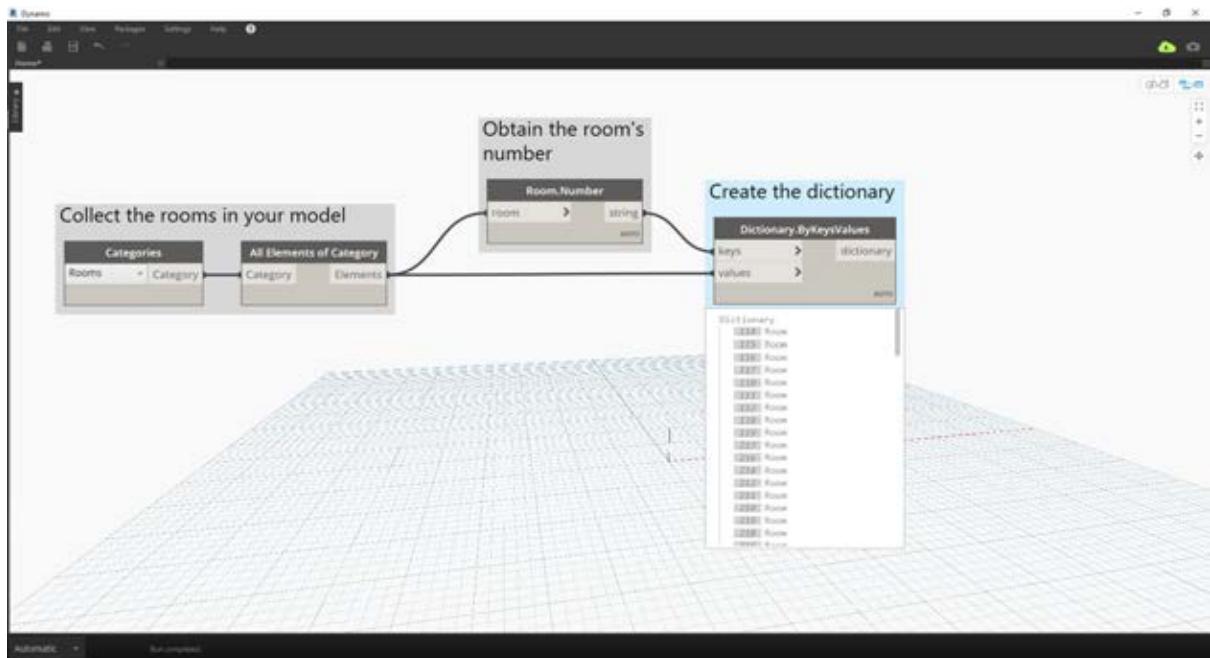
まず、Revit モデル内のすべての部屋を収集する必要があります。

- 操作する Revit カテゴリを選択します(ここでは部屋を操作します)。
- Dynamo に対して、これらの要素すべての収集を指示します。



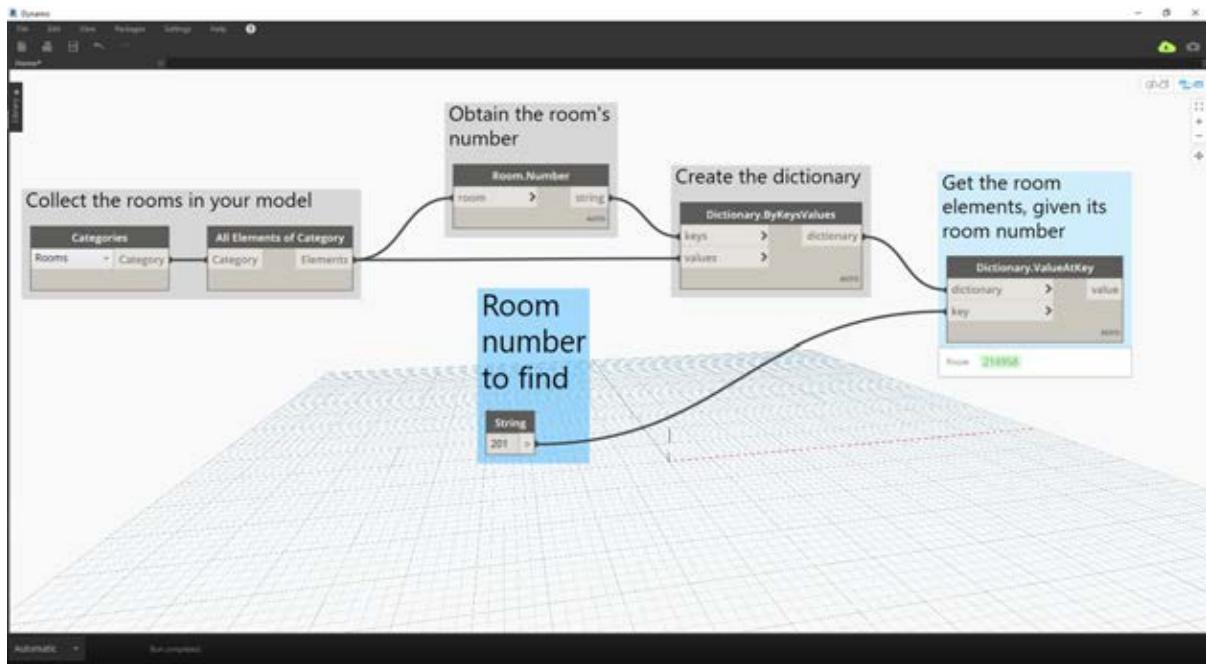
次に、このデータを検索するために使うキーを決定する必要があります(キーに関する情報は、セクション「[9-1 ディクショナリとは](#)」を参照してください)。

- 使用するデータは部屋番号です。



ここでは、指定されたキーと要素でディクショナリを作成します。

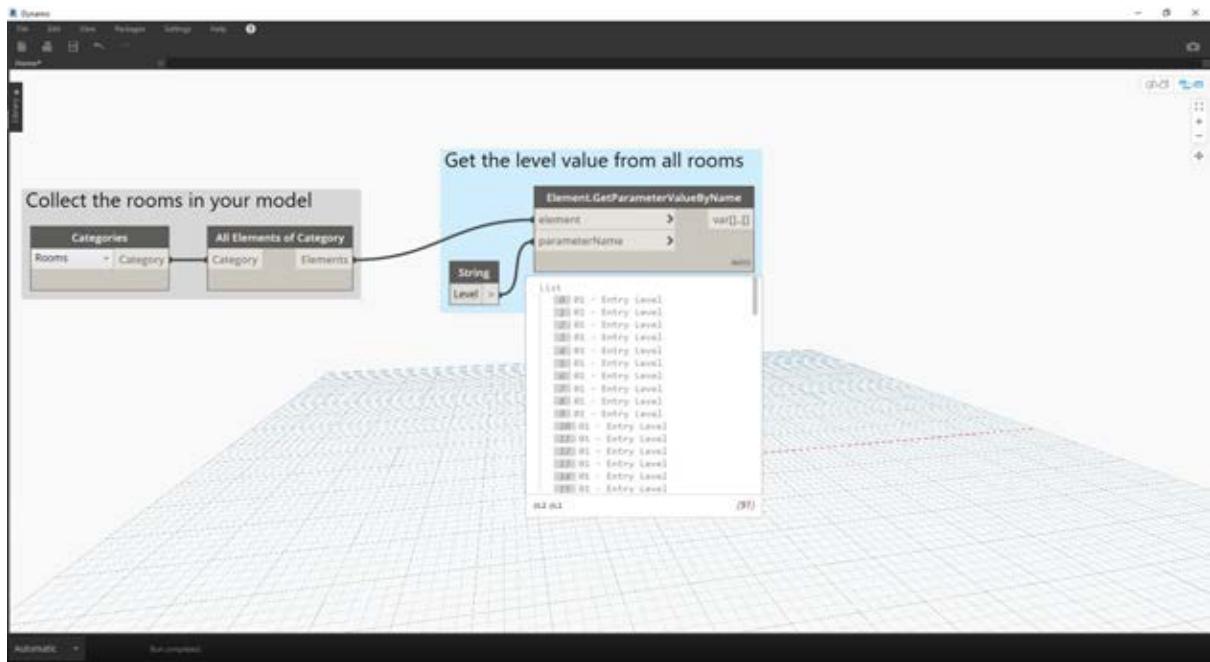
- Dictionary.ByKeysValues ノードは、適切に入力されるとディクショナリを作成します。
- キーは文字列であることが必要ですが、値はさまざまなオブジェクトタイプにすることができます。



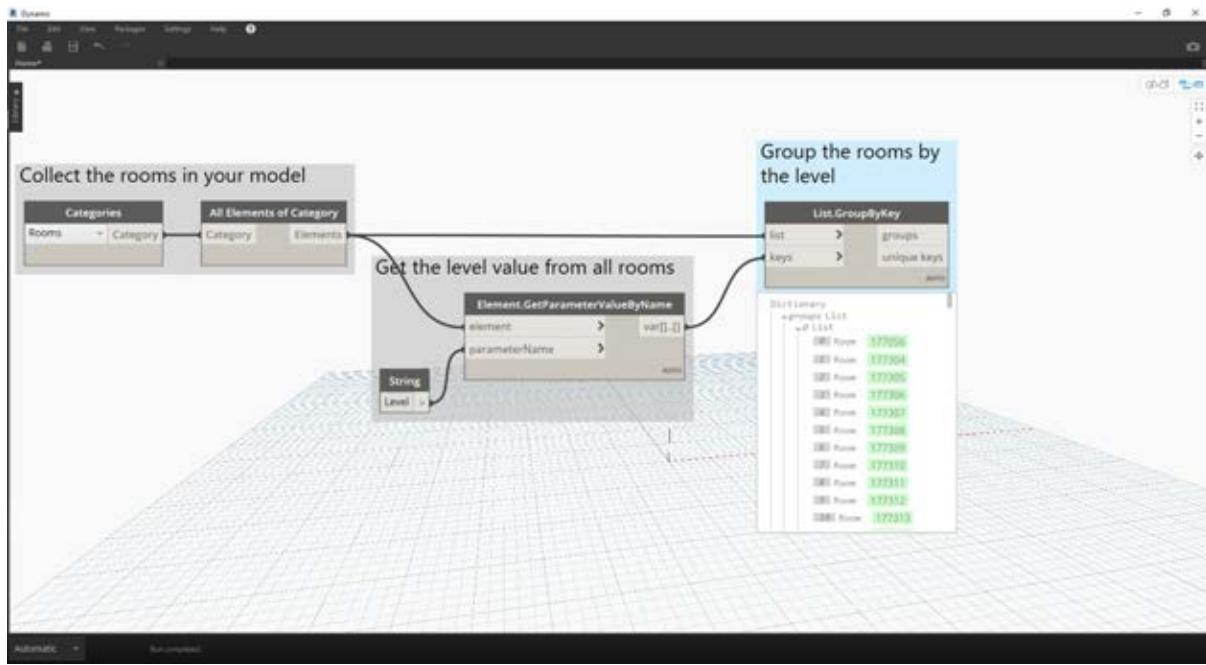
最終的に、部屋番号を使って、ディクショナリから部屋を取得することができます。

- 文字列は、ディクショナリからオブジェクトを検索するために使用しているキーです。
- Dictionary.ValueAtKey ノードで、ディクショナリからオブジェクトを取得します。

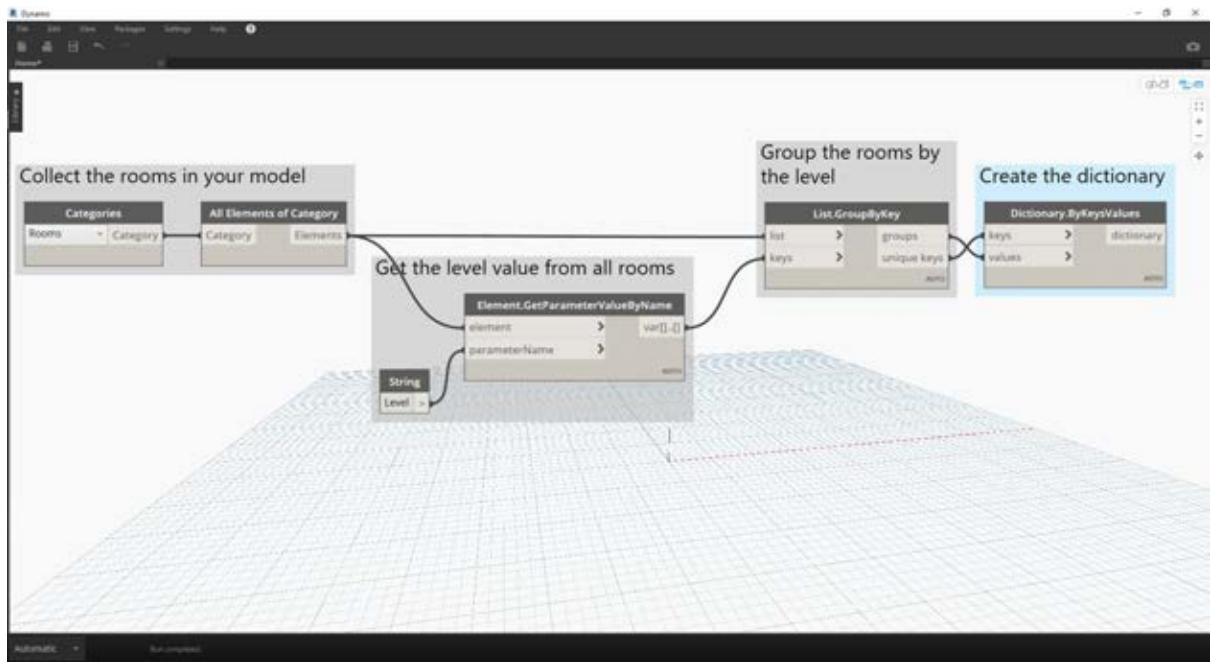
ディクショナリによるこの同じ方法を使用して、グループ化されたオブジェクトでディクショナリを作成することもできます。指定されたレベルですべての部屋を検索する場合、上のグラフを次のように修正できます。



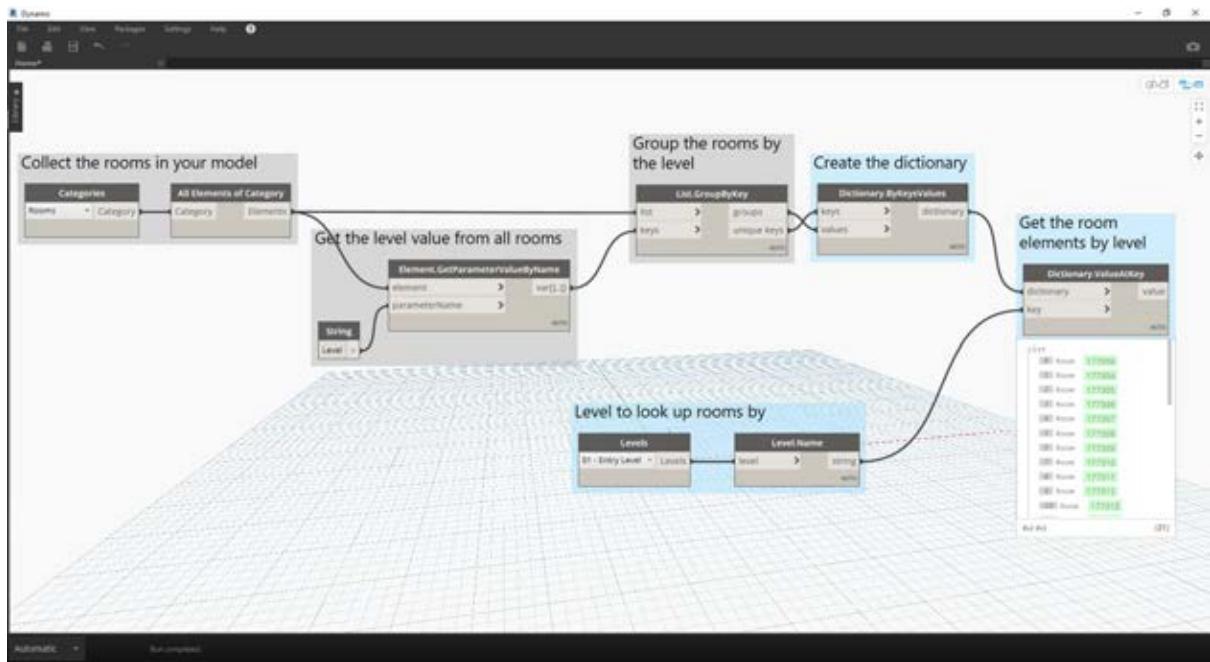
- キーとして部屋番号を使用するのではなく、パラメータの値(ここではレベル)を使用できます。



- これで、部屋をそのレベルごとにグループ化することができます。



- レベルごとにグループ化された要素を使用することで、共有のキー(固有のキー)をディクショナリのキーとして使用し、部屋のリストを要素として使用できるようになります。



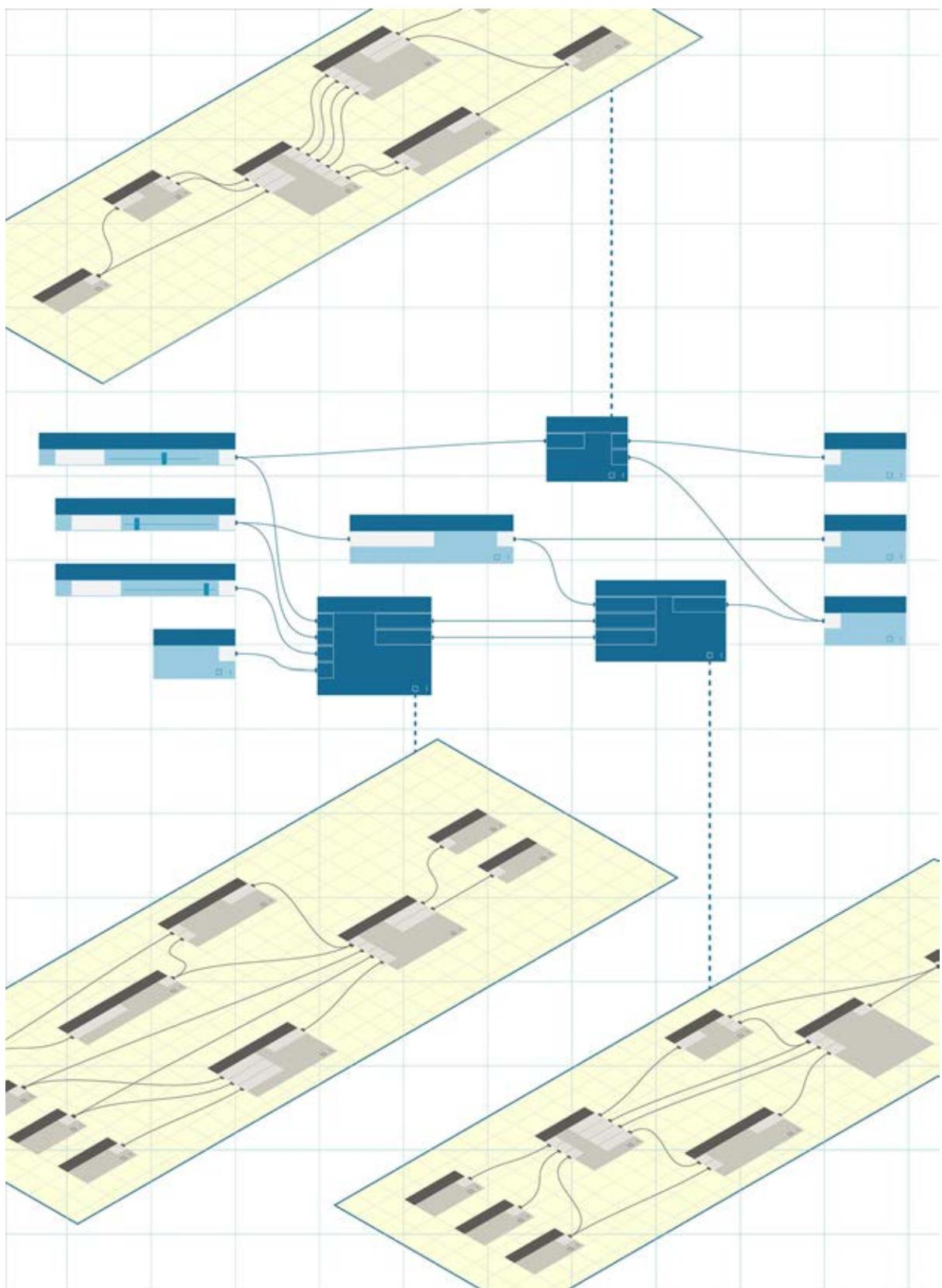
- 最終的に、Revit モデル内のレベルを使用して、ディクショナリ内で、そのレベルに配置されている部屋を検索できます。Dictionary.ValueAtKey は、レベル名を取得して、そのレベルの部屋オブジェクトを返します。

[Dictionary]カテゴリのノードを使用する機会は、実際には無限にあるのです。Revit 内の BIM データを要素自体に関連付けることができる所以、さまざまな使用例が考えられます。

## カスタム ノード

### カスタム ノード

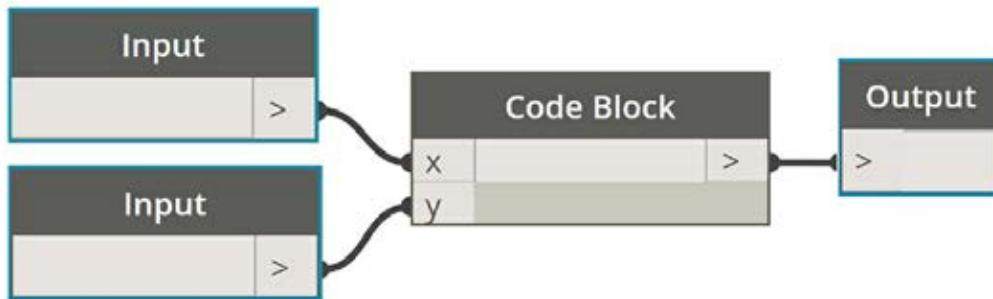
Dynamo のノードのライブラリには、すぐに使用できる多くの機能が保存されています。頻繁に使用するルーチンや、コミュニティと共有する特殊なグラフがある場合、カスタム ノードは Dynamo をさらに拡張するための便利な方法です。



# カスタム ノード

## カスタム ノード

Dynamo には、ビジュアル プログラミングのさまざまなタスクで使用できる多くのコア ノードが用意されています。ただし、独自のノードを作成した方が、迅速かつ効率的な方法で、簡単に問題を解決できる場合があります。これらの独自のノードは複数の異なるプロジェクトで再利用できるため、グラフが見やすくなります。これらのノードをパッケージ マネージャにプッシュして、世界中の Dynamo コミュニティと共有することもできます。



## グラフをクリーン アップする

カスタム ノードは、他のノードとカスタム ノードを「Dynamo のカスタム ノード」内にネストすることによって作成されます。Dynamo のカスタム ノードは、コンテナとして考えることができます。このコンテナ ノードをグラフ内で実行すると、コンテナ ノード内のすべてのノードが実行されるため、ノードを自由に組み合わせて再利用や共有を行うことができます。

## 変更に対応する

グラフ内でカスタム ノードの複数のコピーを使用している場合、基準となるカスタム ノードを編集することにより、一度にすべてのコピーを更新することができます。これにより、ワークフローや設計にどのような変更が生じた場合でも、グラフをシームレスに更新して変更に対応することができます。

## ワーク シェアリング

カスタム ノードの最も便利な機能は、ワーク シェアリング機能です。たとえば、Dynamo の上級ユーザが複雑な Dynamo グラフを作成し、そのグラフを Dynamo を初めて使用する設計者に渡したとします。この場合、グラフを渡されたユーザはそのグラフを簡素化し、設計についてのやり取りに必要な最小限の要素だけに絞り込むことができます。カスタム ノードを開いて内部のグラフを編集することができますが、「コンテナ」はシンプルな状態に保たれます。このプロセスにより、カスタム ノードを使用する Dynamo ユーザは、直感的に見やすいグラフを設計することができます。

# PointsToSurface

points

Points

sourceSurface

targetSurface

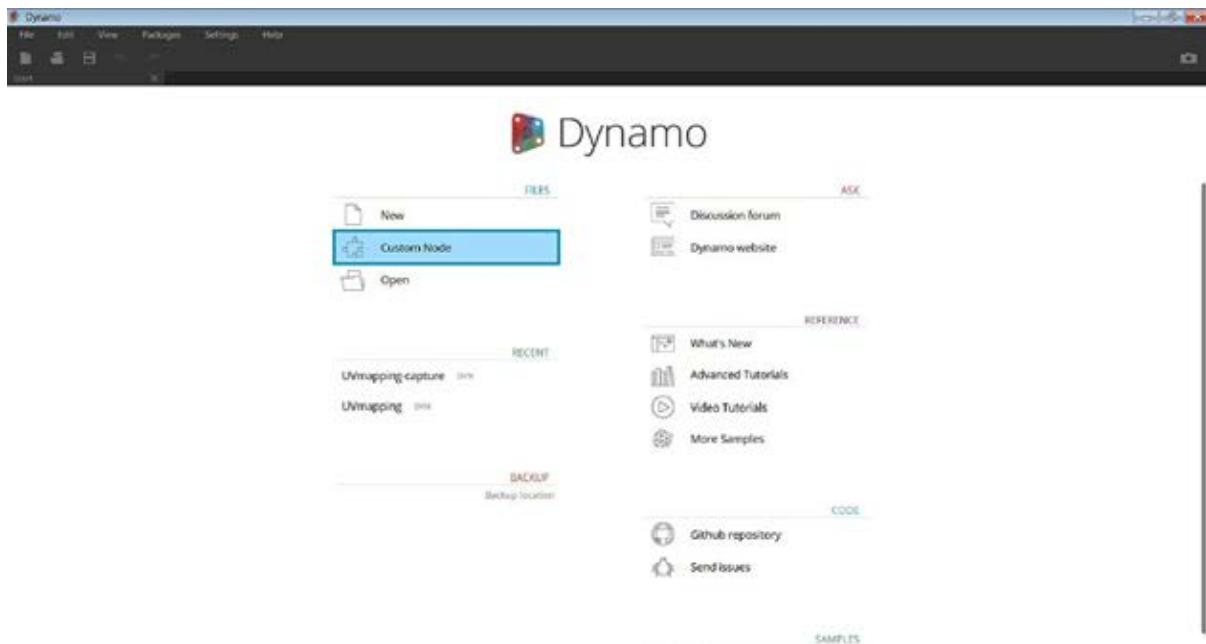


## ノードを作成するさまざまな方法

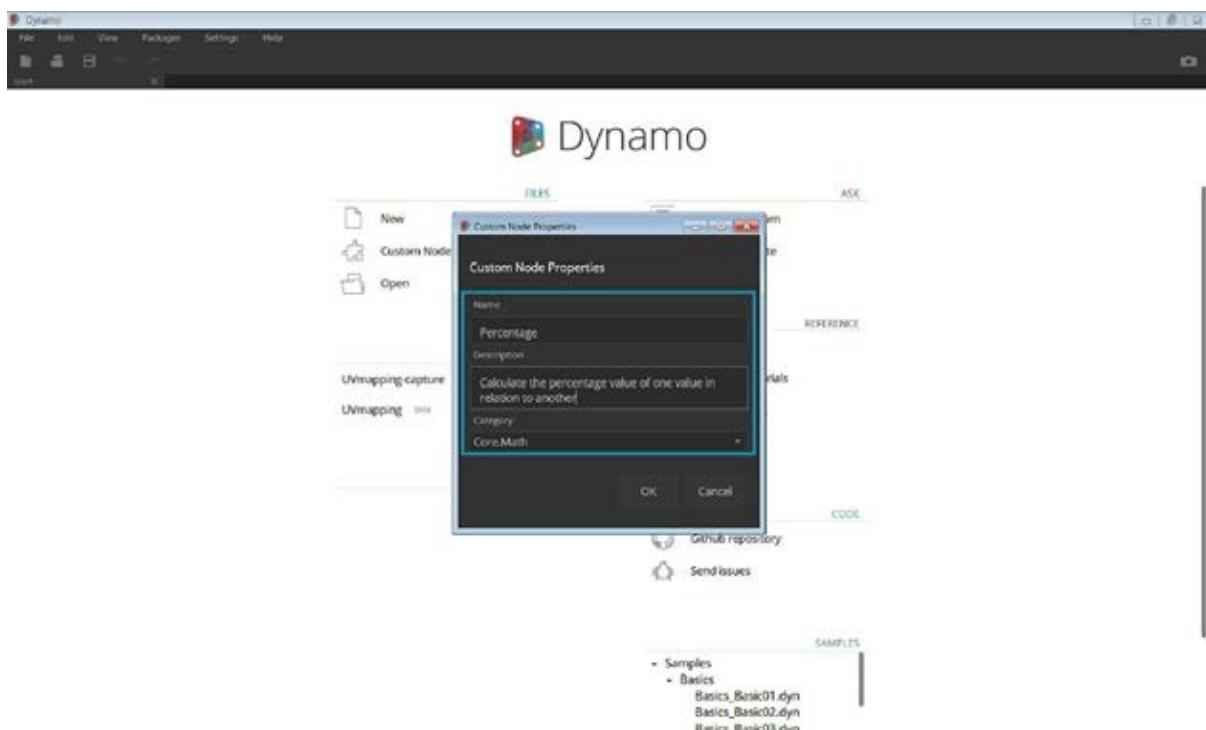
Dynamo では、さまざまな方法でカスタム ノードを作成することができます。この章の例では、Dynamo UI からカスタム ノードを直接作成します。C# や Zero-Touch の構文の詳細については、Dynamo Wiki の[このページ](#)を参照してください。

### カスタム ノード環境

カスタム ノード環境を実際に使用して、パーセンテージを計算する単純なノードを作成してみましょう。カスタム ノード環境は、Dynamo のグラフ環境とは異なりますが、インターフェイスは基本的に同じです。では、最初のカスタム ノードを作成してみましょう。

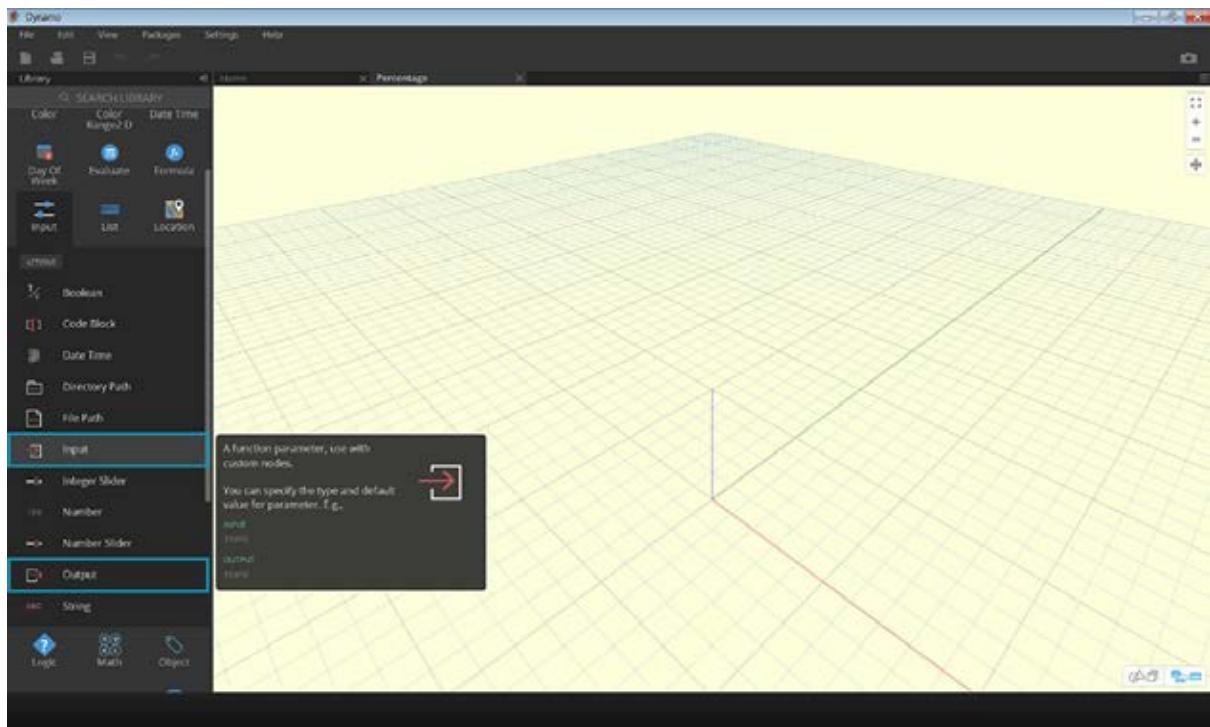


カスタム ノードを一から作成するには、Dynamo を起動して[カスタム ノード]を選択するか、キャンバスで[Ctrl]+[Shift]+[N]を押します。

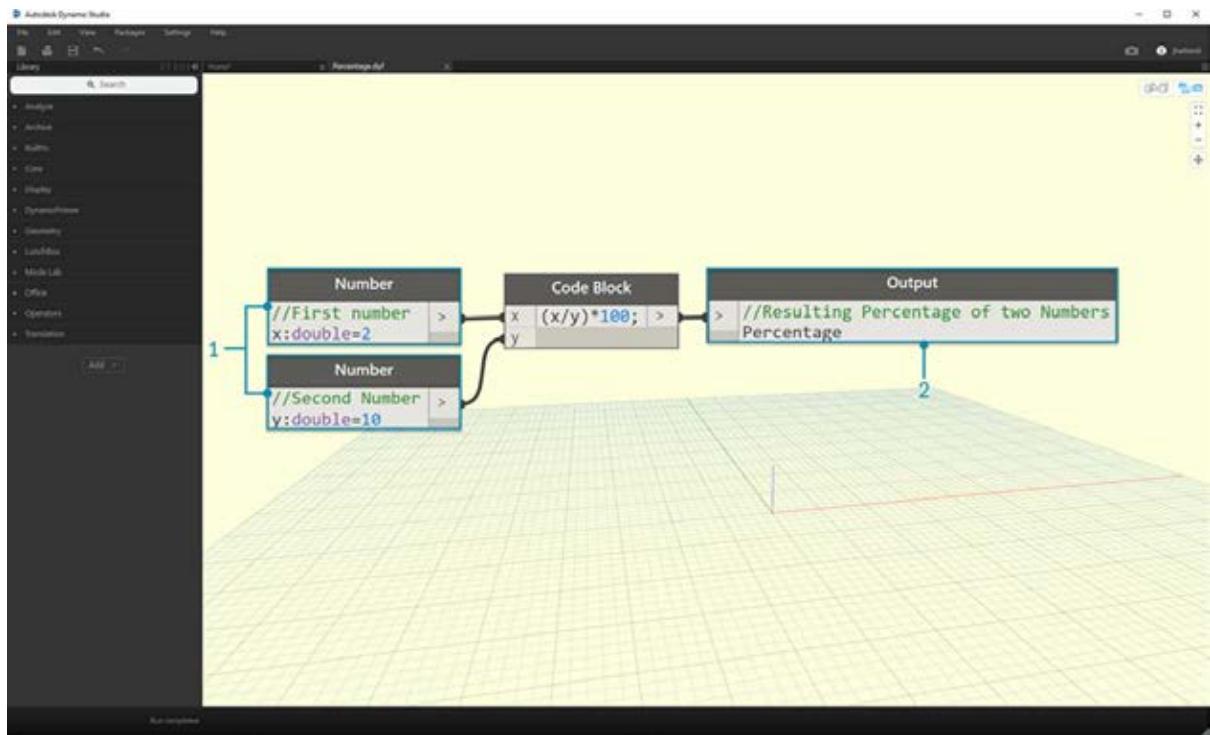


[カスタム ノード プロパティ]ダイアログボックスで、カスタム ノードの名前、説明、カテゴリを入力します。

1. 名前: Percentage
2. 説明: Calculate the percentage of one value in relation to another (一方の値に対するもう一方の値のパーセンテージを計算)
3. カテゴリ: Core.Math



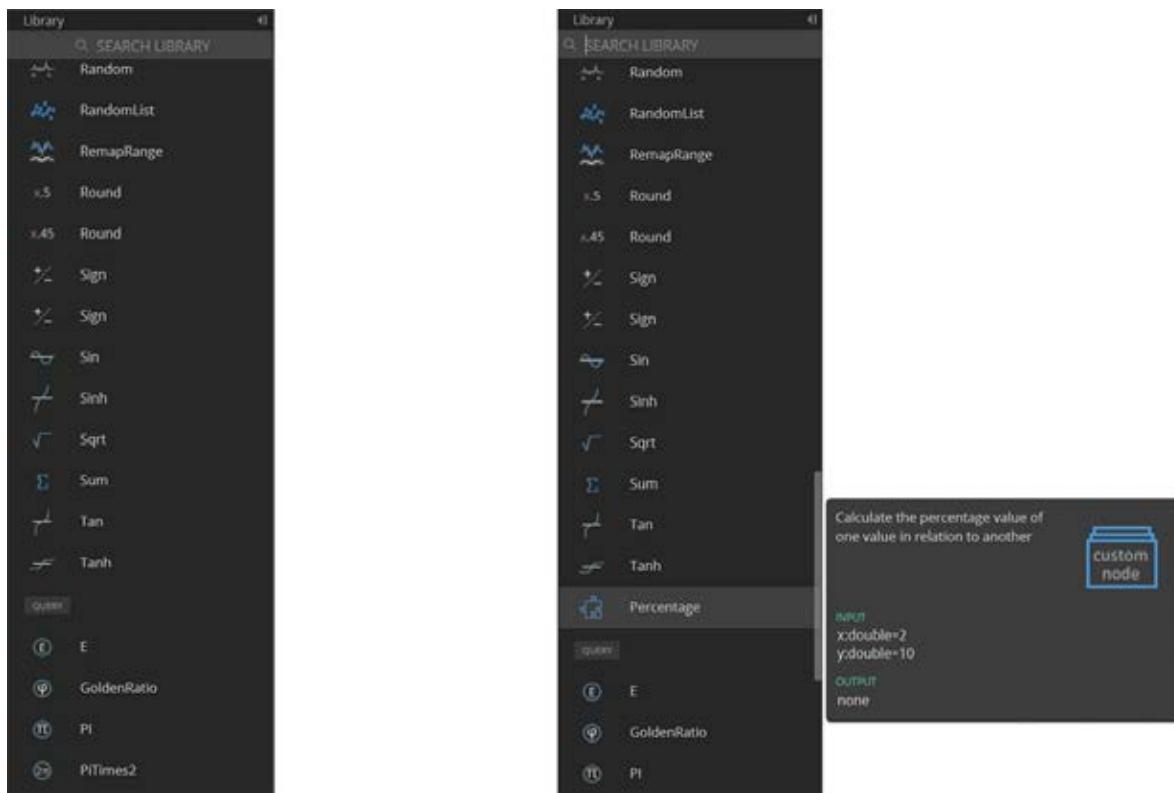
キャンバスの背景色が黄色で表示されます。これは、カスタム ノードの内部を表しています。このキャンバス内では、**Input** ノードや **Output** ノードなど、Dynamo の核となるすべてのノードを使用することができます。この 2 つのノードにより、カスタム ノードでやり取りされるデータにラベルが付けられます。これらのノードは、[Core] > [Input] で使用することができます。



1. **Input** ノードは、カスタム ノード上に入力ポートを作成します。Input ノードの構文は、`input_name : datatype = default_value(optional)` です。
1. **Output** ノードは、Input ノードと同様に、カスタム ノード上に出力ポートを作成して名前を付けます。カスタム コメントを入力ポートと出力ポートに追加して、入力タイプと出力タイプがわかるようにすることをお勧めします。詳細については、「[カスタム ノードを作成する](#)」セクションを参照してください。

作成したカスタム ノードは、.dyf ファイルとして保存することができます(標準ノードの場合は .dyn ファイルとして保存されます)。保存したファイルは、現在のセッションと将来のセッションに自動的に追加されます。カスタム ノードは、そのカスタム ノードのプロパティで指

定されているカテゴリ内のライブラリに登録されます。



左側は、既定のライブラリの[Core] > [Math]カテゴリで、右側は、新しいカスタム ノードが追加された[Core] > [Math]カテゴリです。

## 今後の予定

ここまで手順では、最初のカスタム ノードを作成しました。これ以降の各セクションでは、カスタム ノードの機能と、一般的なワークフローをパブリッシュする方法について詳しく説明します。次のセクションでは、ジオメトリを特定のサーフェスから別のサーフェスに転送するカスタム ノードを作成する方法について説明します。

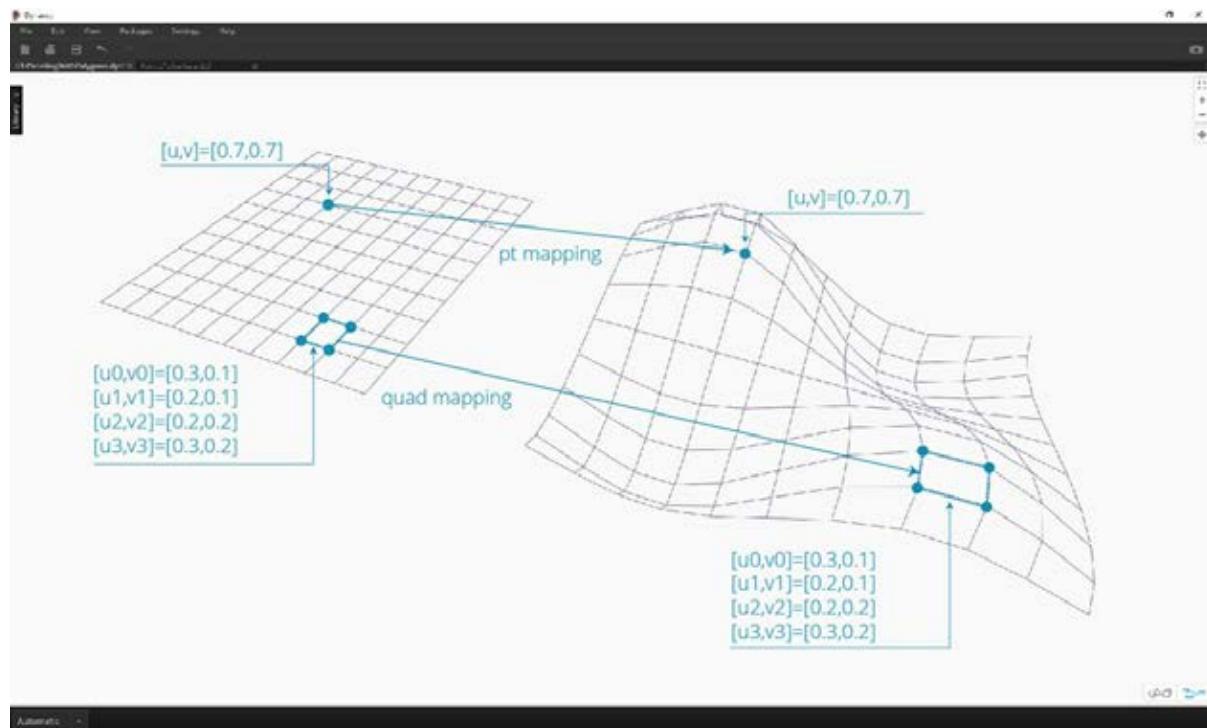
# カスタム ノードを作成する

## カスタム ノードを作成する

Dynamo では、いくつかの方法でカスタム ノードを作成することができます。最初からカスタム ノードを作成することも、既存のグラフから作成することも、C# を使用して明示的に作成することもできます。このセクションでは、既存のグラフを使用して Dynamo UI 内にカスタム ノードを作成する方法について説明します。ワークスペースを整理し、一連のノードをパッケージ化して別の場所で再利用する場合は、この方法が最適です。

### UV マッピング用のカスタム ノード

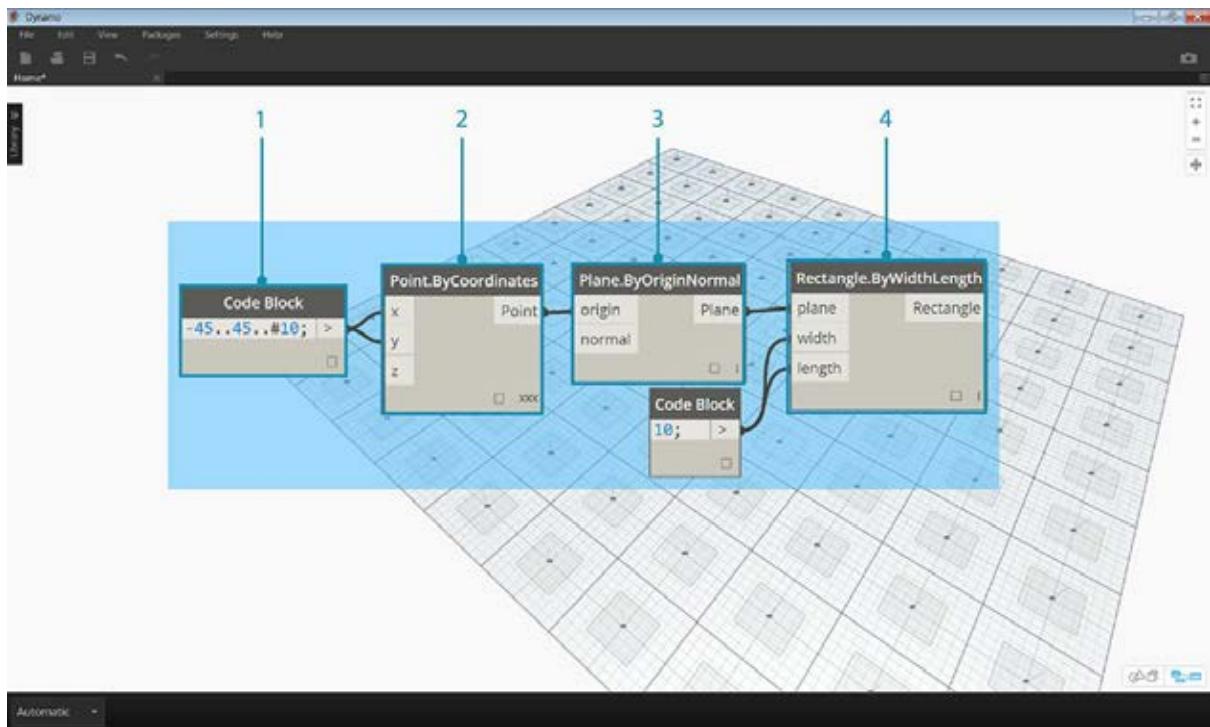
次の図は、UV 座標を使用して、1 つのサーフェスから別のサーフェスに点をマッピングする場合の例を示しています。この概念を適用して、XY 平面上の曲線を参照する、複数の小さなパネルから構成されるサーフェスを作成してみましょう。ここでは、パネル化用の四角形のパネルを作成しますが、同じ概念を適用して、UV マッピングを使用する多様なパネルを作成することもできます。この演習を行うと、このグラフや Dynamo の別のワークフローで同様のプロセスを簡単に繰り返すことができるようになるため、カスタム ノード開発のよい練習になります。



### 既存のグラフからカスタム ノードを作成する

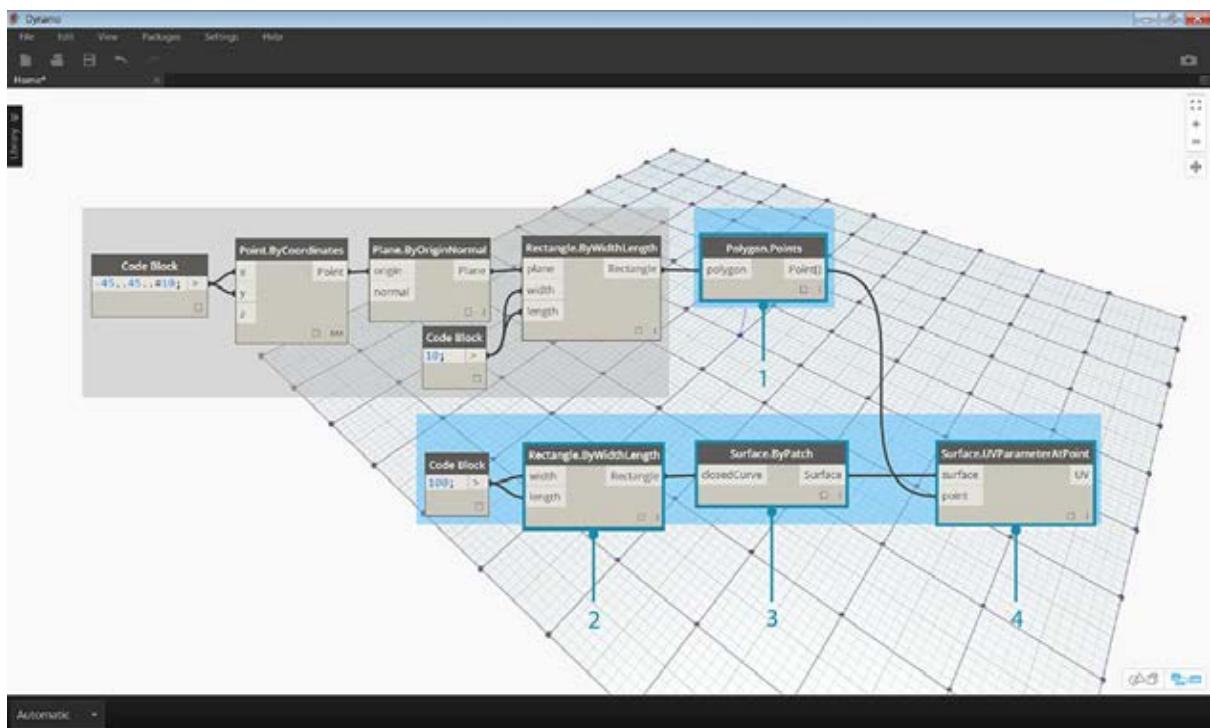
この演習用のサンプル ファイルをダウンロードして解凍してください(右クリックして[名前を付けてリンク先を保存]を選択)。すべてのサンプルファイルの一覧については、付録を参照してください。[UV-CustomNode.zip](#)

最初に、カスタム ノード内にネストするグラフを作成します。この例では、UV 座標を使用して、基準となるサーフェスから目的のサーフェスにポリゴンをマッピングするグラフを作成します。この UV マッピング プロセスは頻繁に使用するプロセスであるため、カスタム ノードの演習素材として適しています。サーフェスと UV 空間の詳細については、セクション 5.5 を参照してください。グラフ全体は、上でダウンロードした .zip ファイルの *UVmapping\_Custom-Node.dyn* で確認できます。



1. **Code Block** ノードを使用して、-45 ~ 45 の範囲内で 10 個の数値を作成します。
2. **Point.ByCoordinates** ノードの x 入力と y 入力に **Code Block** ノードの出力を接続し、[レーシング]を[外積]に設定します。これで、点のグリッドが作成されます。
3. **Plane.ByOriginNormal** ノードの origin 入力に **Point.ByCoordinates** ノードの Point 出力を接続して、各点に平面を作成します。この操作では、既定の法線ベクトル(0,0,1)が使用されます。
4. **Rectangle.ByWidthLength** ノードの plane 入力に前の手順で作成した平面を接続し、別の **Code Block** ノードで値 10 を指定して幅と長さを設定します。

これで、長方形のグリッドが作成されます。UV 座標を使用して、これらの長方形を目的のサーフェスにマッピングします。

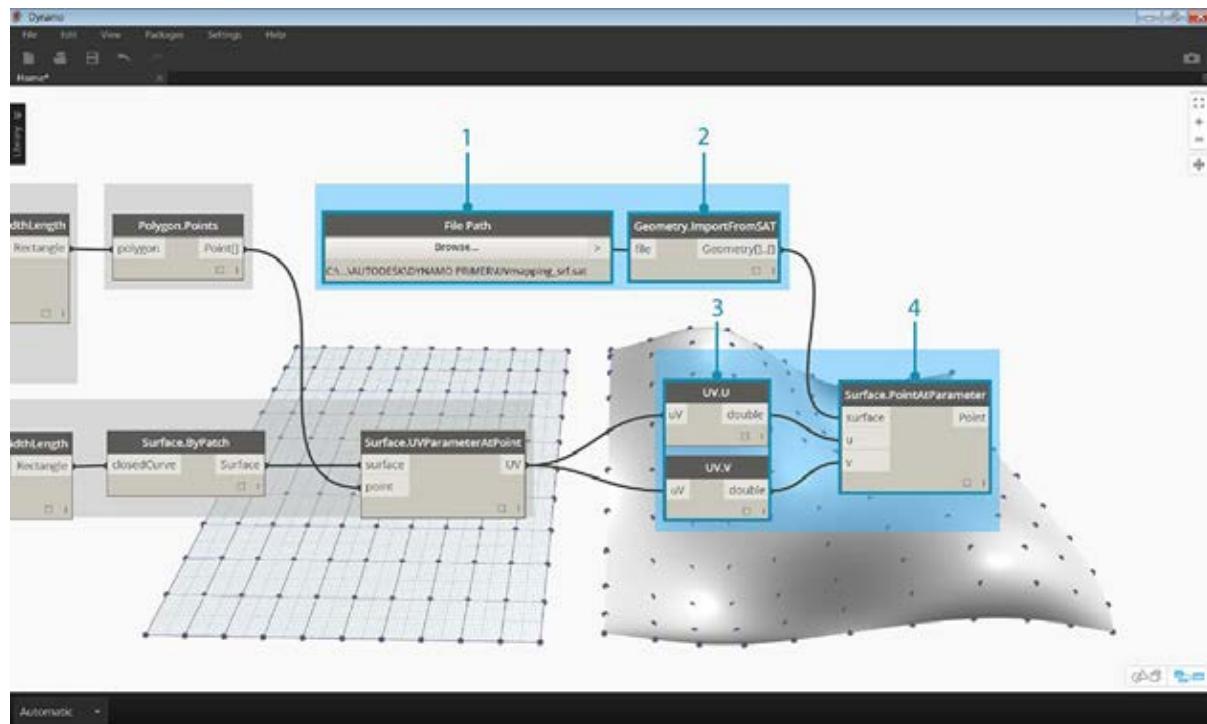


1. **Polygon.Points** ノードの polygon 入力に前の手順の Rectangle 出力を接続し、各長方形の頂点を抽出します。これらの点を、目的のサーフェスにマッピングします。
2. 値 1001 が指定された **Code Block** ノードを **Rectangle.ByWidthLength** ノードに接続して、長方形の幅と長さを指

定します。これが、基準サーフェスの境界線になります。

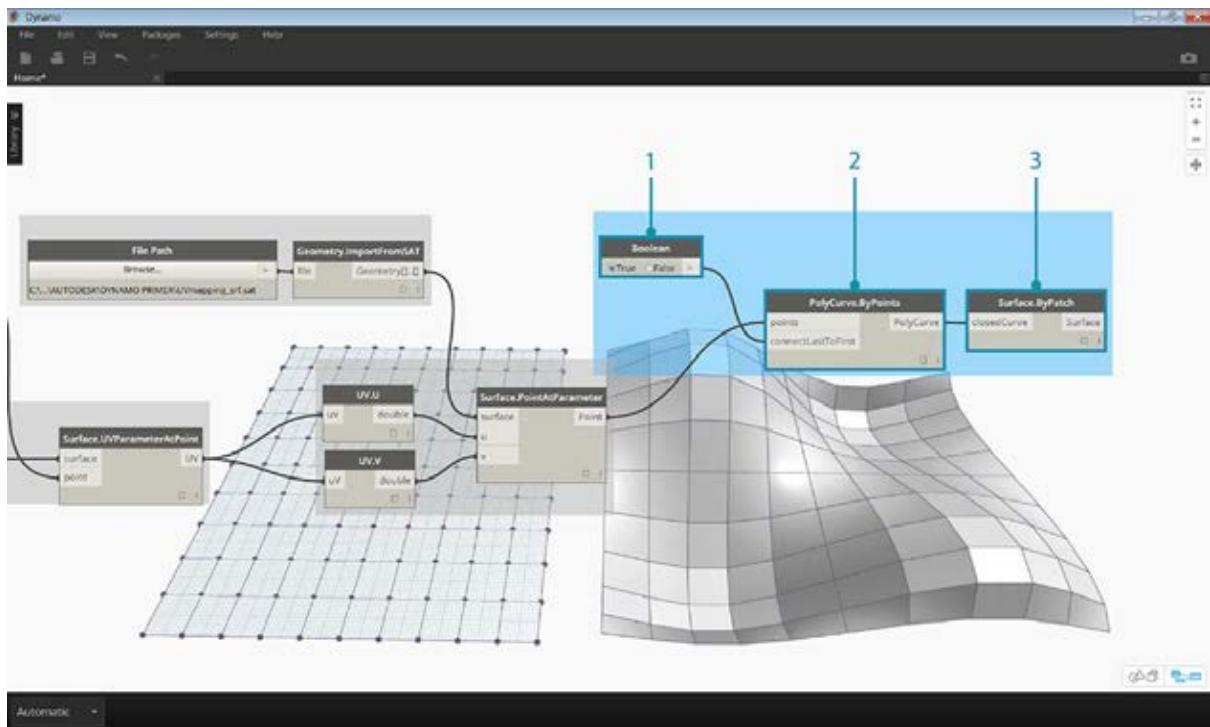
3. **Surface.ByPatch** ノードの *closedCurve* 入力に前の手順の **Rectangle** 出力を接続し、基準となるサーフェスを作成します。
4. **Surface.UVParameterAtPoint** ノードに **Polygon.Points** ノードの *Point* 出力と **Surface.ByPatch** ノードの *Surface* 出力を接続すると、各点における UV パラメータが返されます。

これで、基準となるサーフェスと UV 座標のセットが作成されました。次に、目的のサーフェスを読み込み、2 つのサーフェス間で点をマッピングします。



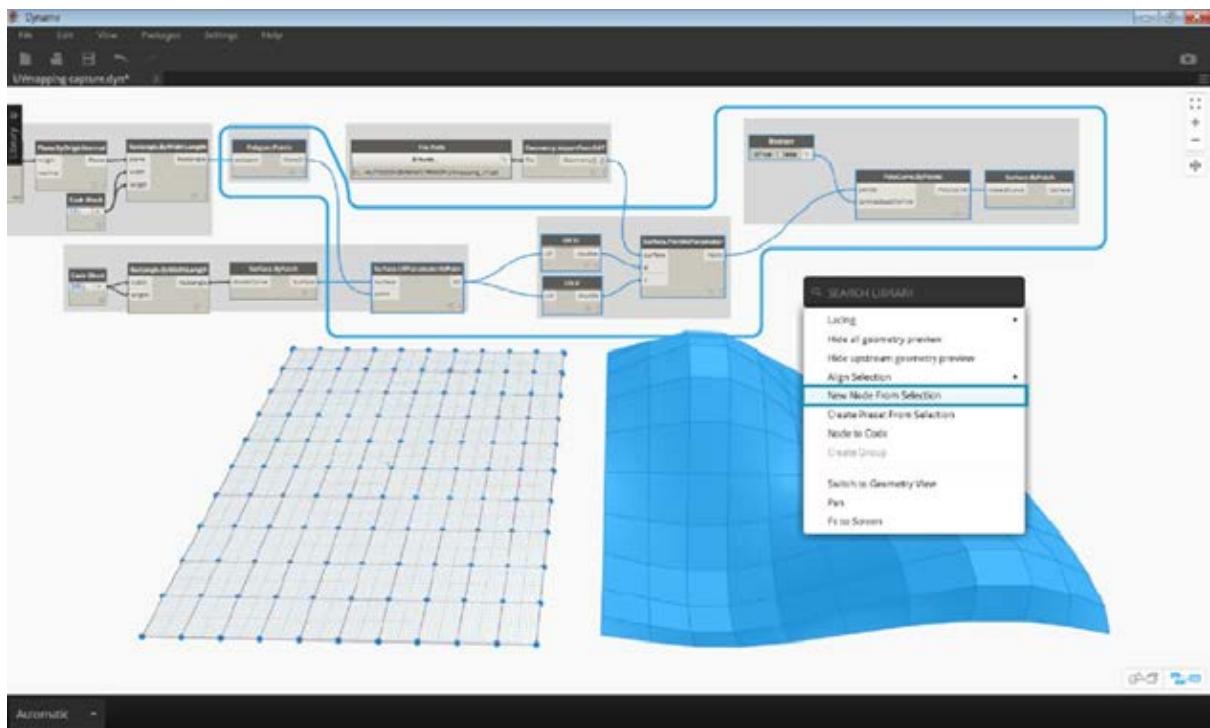
1. **File Path** ノードを使用して、読み込むサーフェスのファイル パスを選択します。ファイル タイプは .sat にしてください。  
[参照...] ボタンをクリックして、上でダウンロードした .zip ファイルの *UVmapping\_srf.sat* [ナビゲート]します。
2. **Geometry.ImportFromSAT** ノードにファイル パスを接続して、サーフェスを読み込みます。読み込んだサーフェスがジオメトリのプレビューに表示されます。
3. UV パラメータ出力を **UV.U** ノードと **UV.V** ノードに接続します。
4. **Surface.PointAtParameter** ノードに、読み込んだサーフェス、U 座標、V 座標を接続します。これで、目的のサーフェス上に 3D の点のグリッドが表示されます。

最後に、3D の点を使用して長方形のサーフェス パッチを作成します。

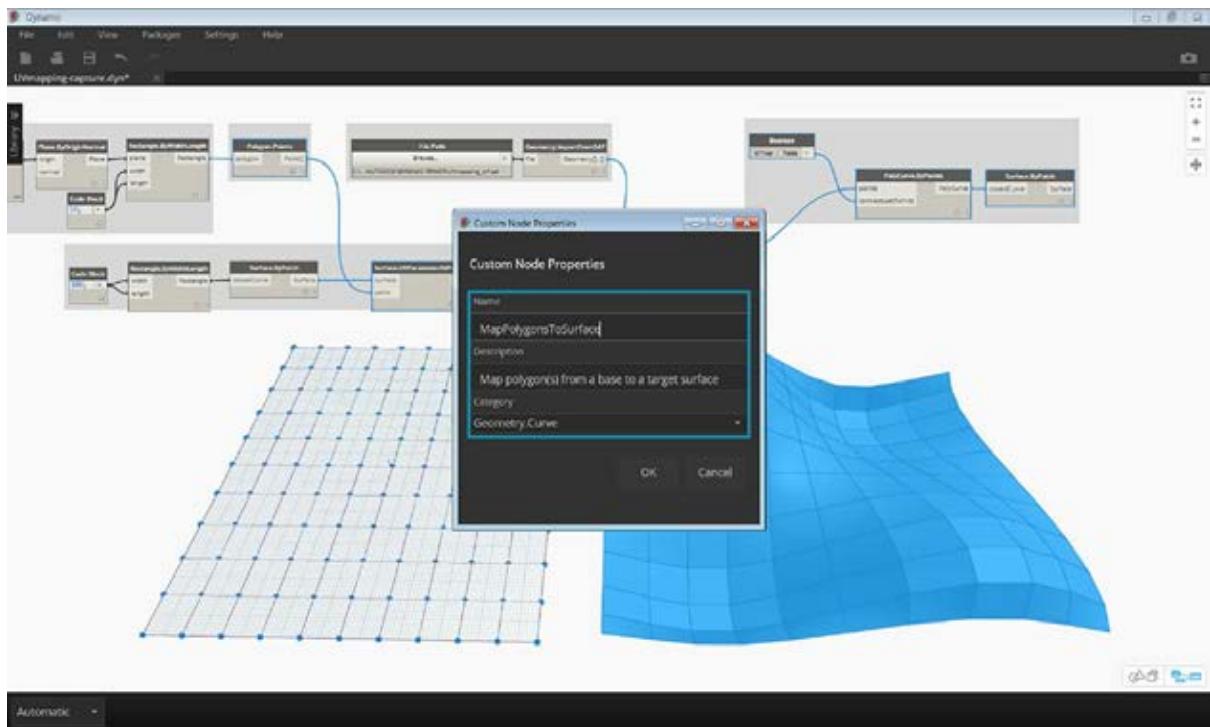


1. **PolyCurve.ByPoints** ノードにサーフェス上の点群を接続し、その点群からポリカーブを作成します。
2. **Boolean** ノードをワークスペースに追加して **PolyCurve.ByPoints** ノードの *connectLastToFirst* 入力に接続し、  
Boolean ノードの値を True に切り替えてポリカーブを閉じます。これで、サーフェスに長方形がマッピングされて表示さ  
れます。
3. **Surface.ByPatch** ノードの *closedCurve* 入力にポリカーブを接続し、サーフェス パッチを作成します。

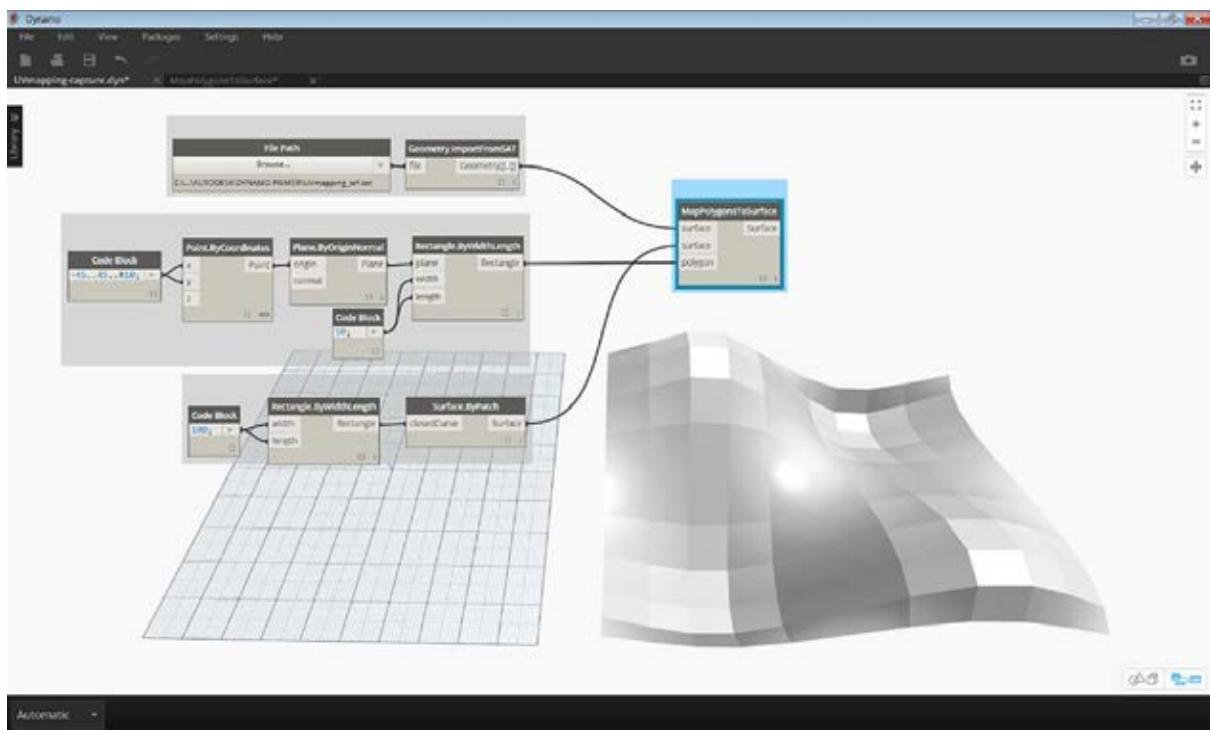
次に、ノードの入力と出力を考慮しながら、カスタム ノード内にネストするノードを選択します。長方形以外の任意のポリゴンをマッピ  
ングできるように、カスタム ノードの柔軟性を可能な限り高めてみましょう。



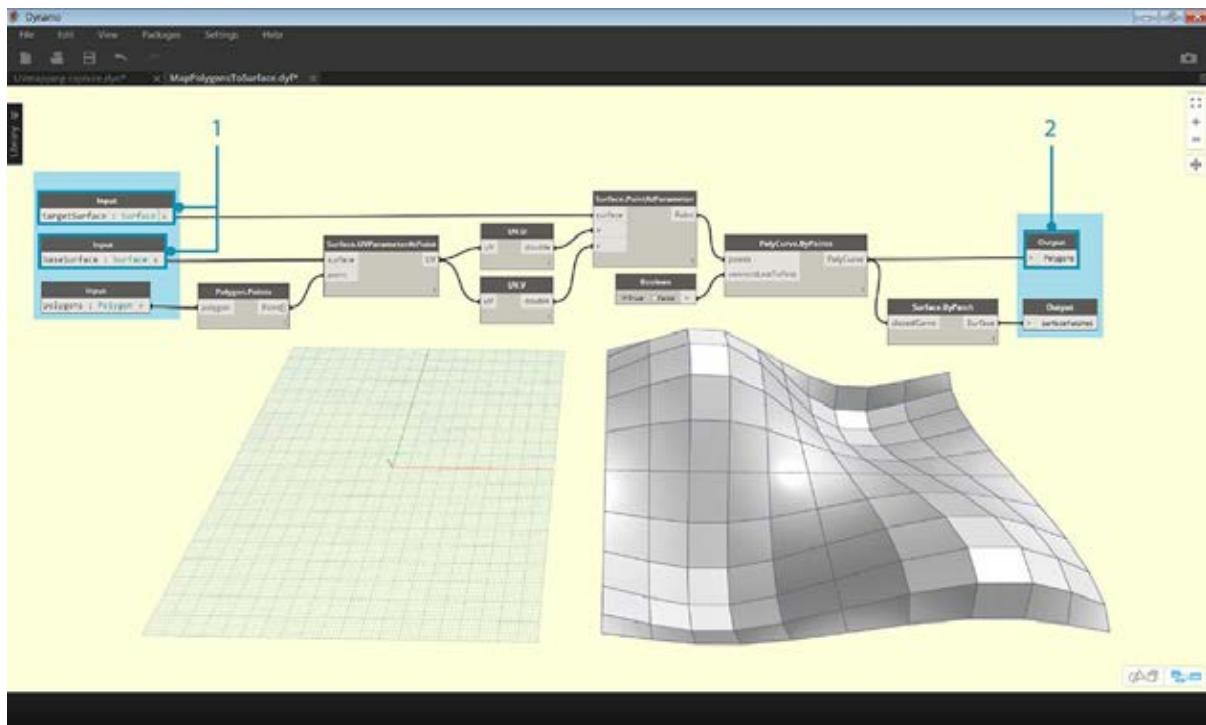
上の図で青い線で囲まれている一連のノード(Polygon.Points ノードから始まる一連のノード)を選択し、ワークスペースを右ク  
リックして[選択からノードを新規作成]を選択します。



[カスタム ノード プロパティ]ダイアログボックスで、カスタム ノードに名前、説明、カテゴリを割り当てます。

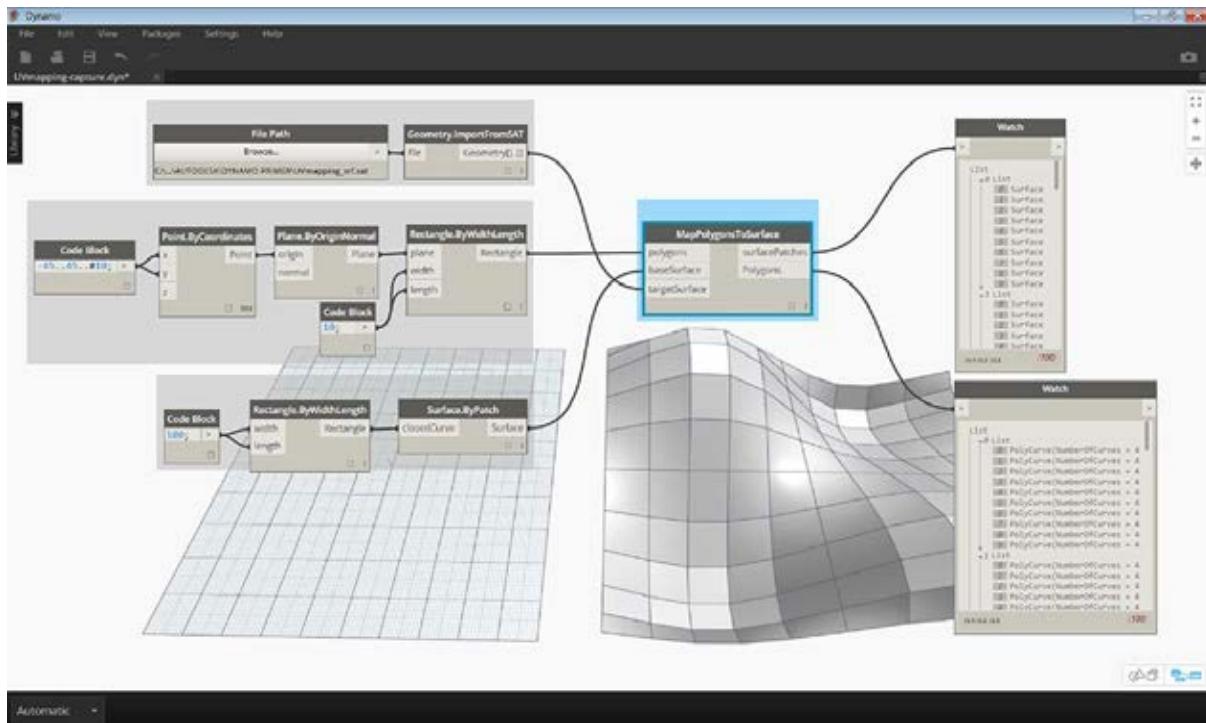


カスタム ノードにより、ワークスペースが見やすくなりました。入力と出力には、元のノードに基づいて名前が付いています。カスタム ノードを編集して、これらの名前をもっとわかりやすい名前に変更しましょう。



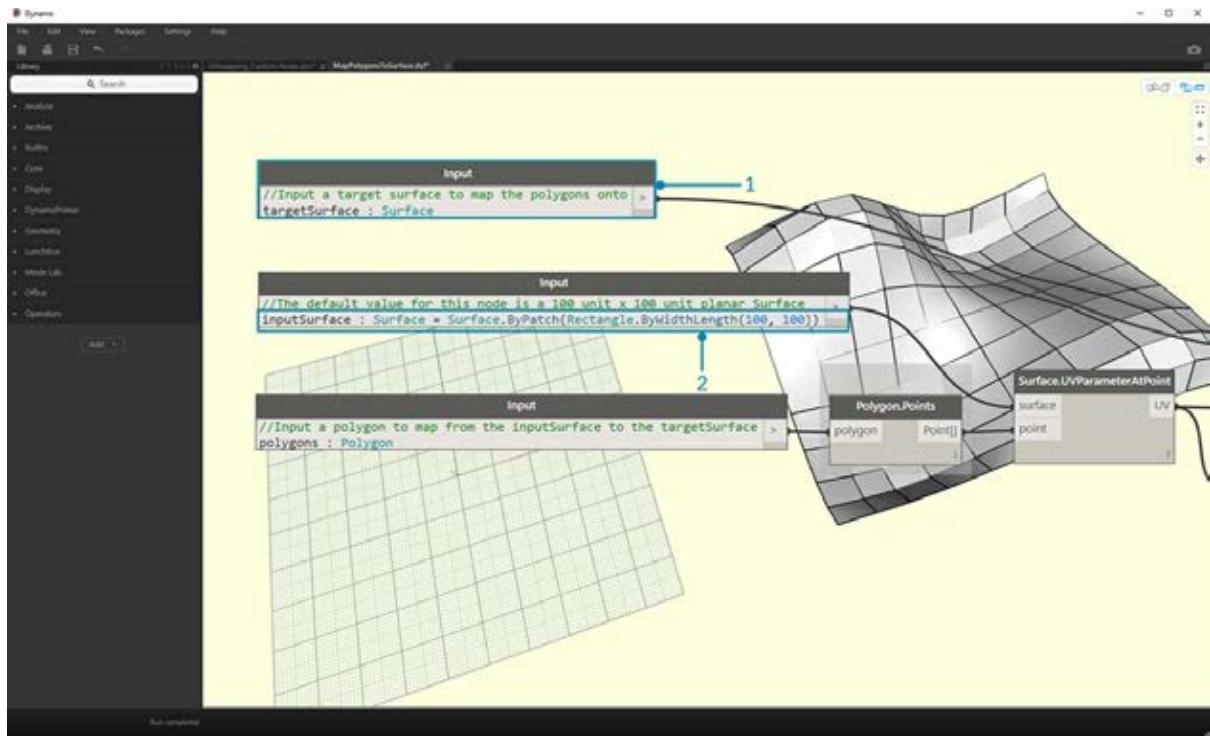
編集するカスタムノードをダブルクリックします。ワークスペースの背景色が黄色で表示されます。これは、カスタムノードの内部を表しています。

1. 各 **Input** ノードの入力名を *baseSurface* と *targetSurface* に変更します。
2. マッピングするポリゴン用に **Output** ノードを追加します。カスタムノードを保存し、ホームワークスペースに戻ります。



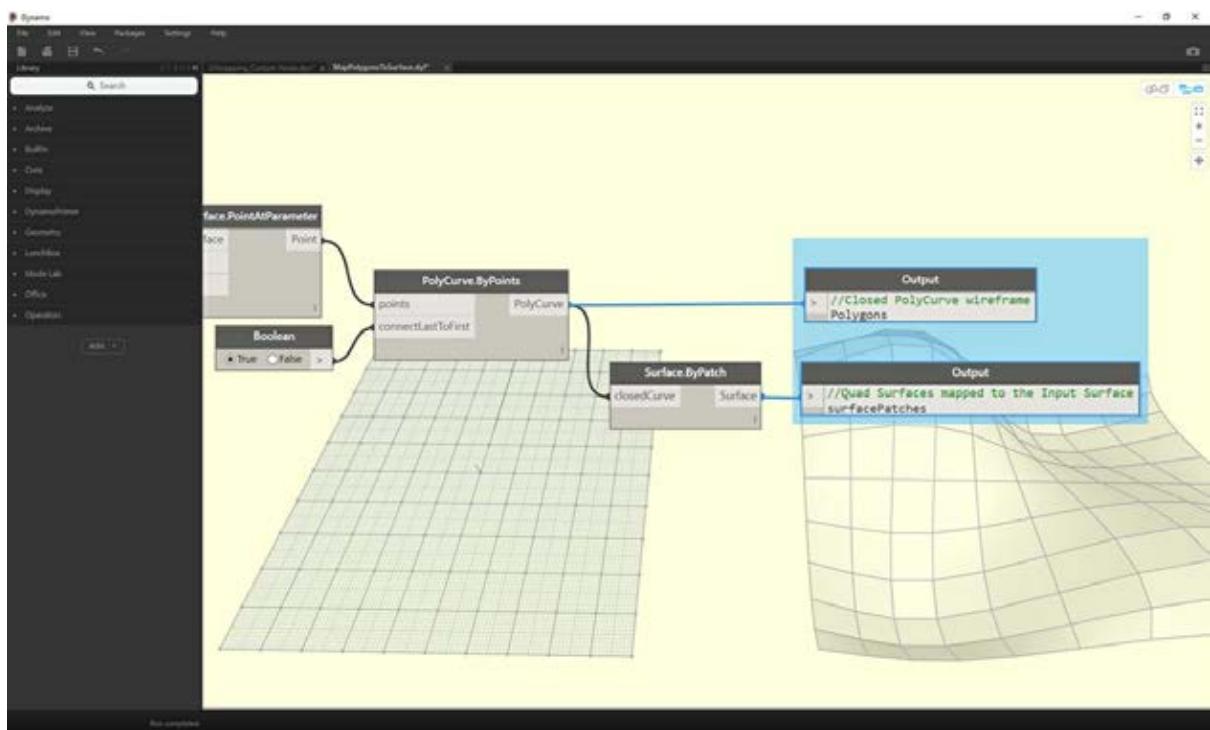
**MapPolygonsToSurface** ノードに変更内容が反映されます。

カスタムコメントを追加して、カスタムノードの内容をさらにわかりやすくすることもできます。コメントを入力すると、入力タイプと出力タイプの内容だけでなく、ノードの機能を説明することができます。カスタムノードの入力や出力にカーソルを置くと、コメントが表示されます。

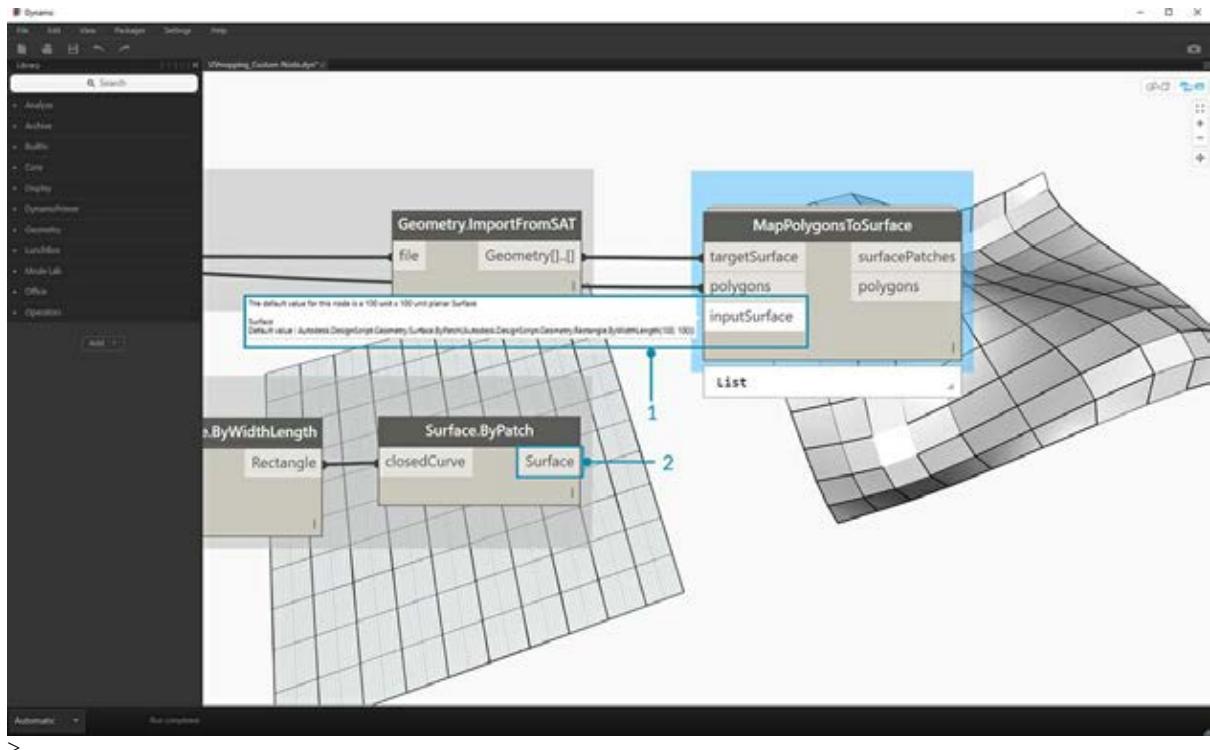


編集するカスタム ノードをダブルクリックします。背景が黄色のワークスペースがもう一度表示されます。

1. Input コード ブロックの編集を開始します。コメントを入力する場合は、最初に「//」を入力してから、コメント テキストを入力します。ノードの内容を説明するためのコメントを入力してください。ここでは、*targetSurface* ノードの説明を入力します。
2. 特定の値に一致する入力タイプを設定して、*inputSurface* ノードの既定値を設定します。ここでは、既定値を元の Surface.ByPatch の値に設定します。



コメントは、出力に対して適用することもできます。Output コード ブロックの編集を開始します。コメントを入力する場合は、最初に「//」を入力してから、コメント テキストを入力します。ここでは、*Polygons* 出力と *surfacePatches* 出力の詳細な説明を追加します。



1. カスタムノード入力にカーソルを置いてコメントを表示します。
2. *inputSurface* ノードの既定値が設定されているため、Surface 入力を使用することなく定義を実行することができます。

# ライブラリへの追加

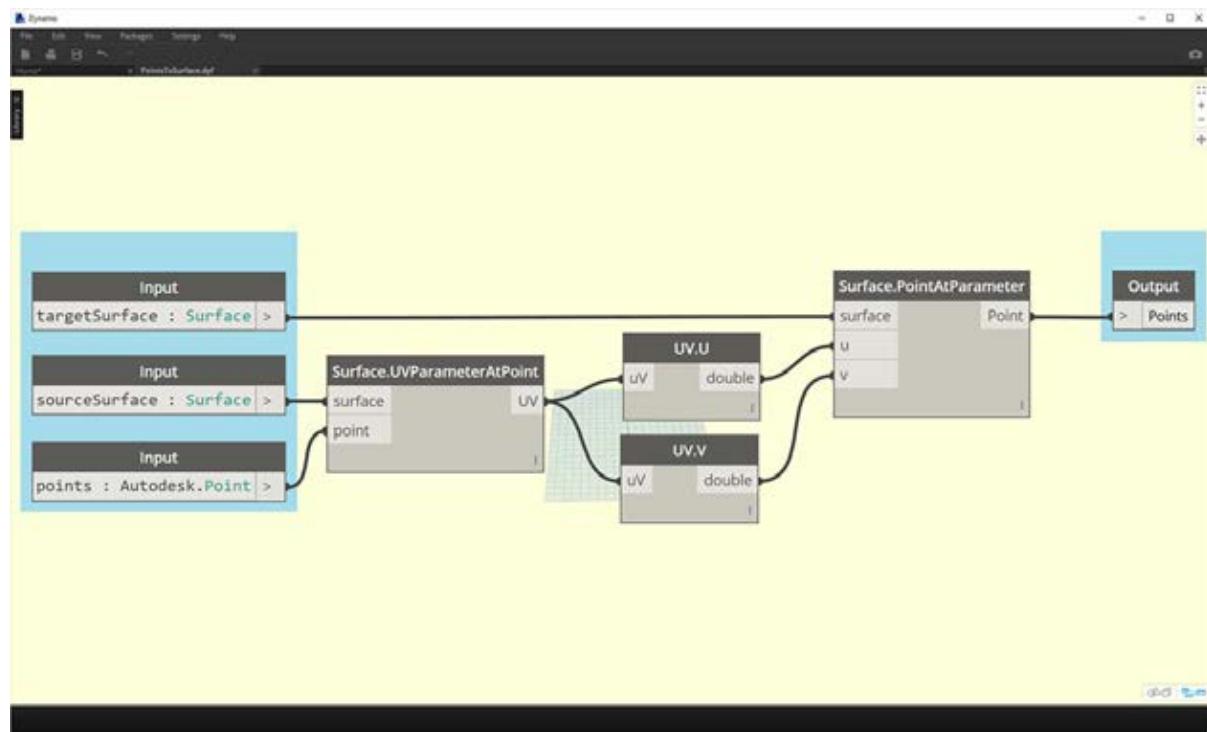
## ライブラリへの追加

ここまで手順で、カスタム ノードを作成して Dynamo グラフ内の特定のプロセスに適用しました。このセクションでは、このノードを他の Dynamo グラフでも参照できるように、このノードをライブラリに保存します。これを実行するには、目的のノードをローカルにパブリッシュします。これは、パッケージをパブリッシュする場合と同様の手順です。パッケージのパブリッシュについては、次の章で詳しく説明します。

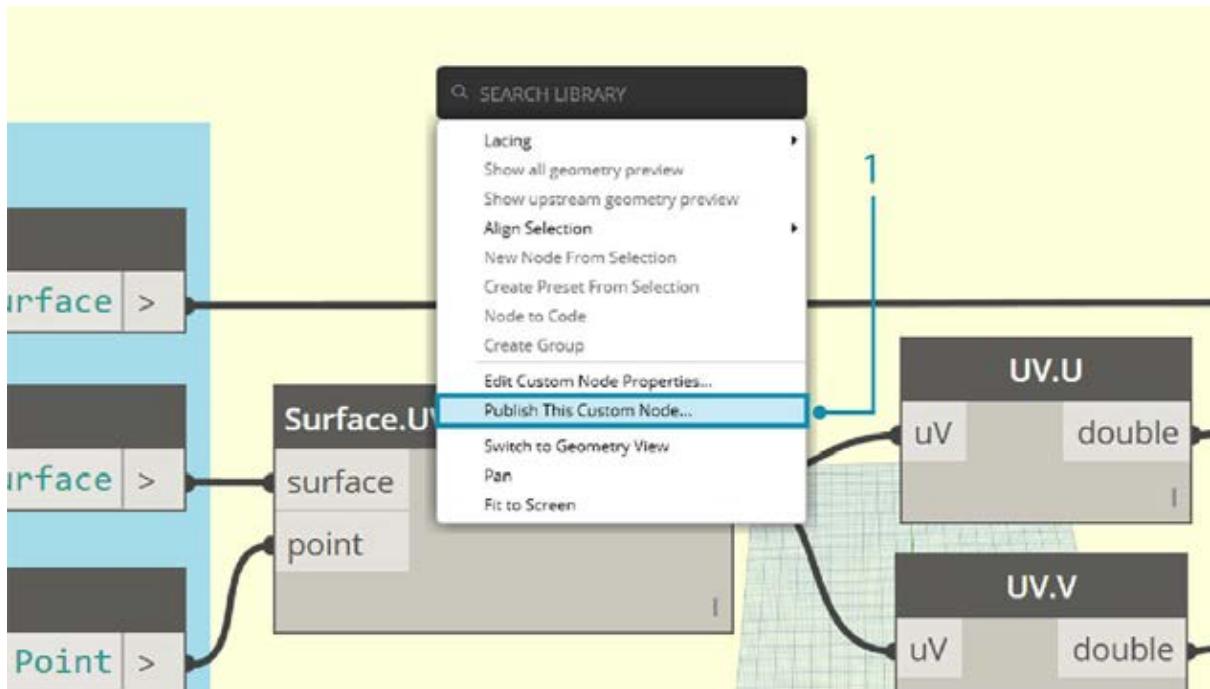
### カスタム ノードをローカルにパブリッシュする

前のセクションで作成したカスタム ノードを使用して、次の手順に進みましょう。ノードをローカルにパブリッシュすると、新しいセッションを開いたときに、Dynamo ライブラリからそのノードにアクセスできるようになります。ノードをパブリッシュせずに Dynamo グラフからカスタム ノードを参照する場合、グラフのフォルダ内に、そのカスタム ノードを含めておく必要があります。または、[File] > [Import Library]を使用してカスタム ノードを Dynamo に読み込む必要があります。

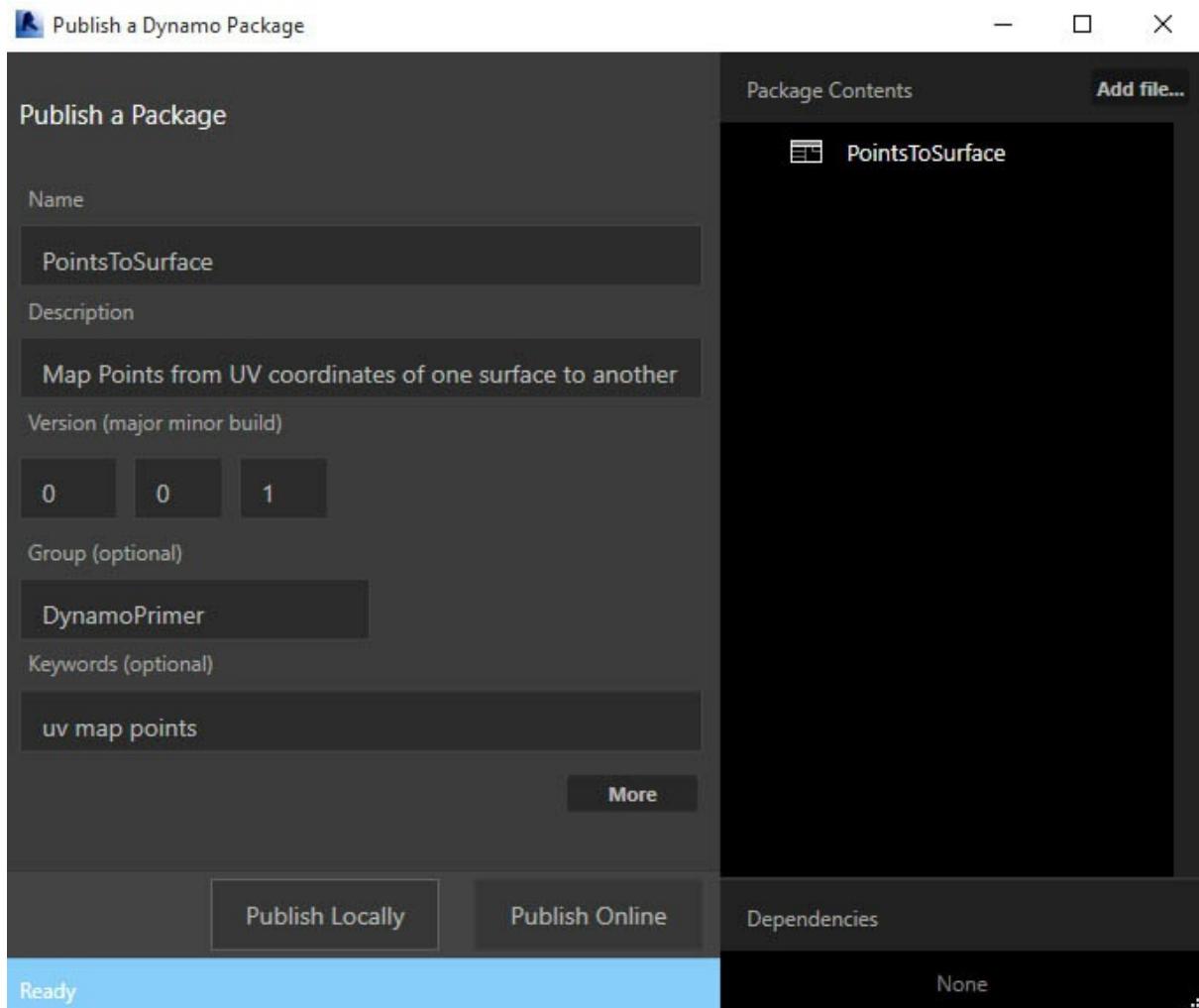
この演習に付属しているサンプル ファイルをダウンロードしてください(右クリックして[名前を付けてリンク先を保存]を選択)。すべてのサンプル ファイルの一覧については、付録を参照してください。[PointsToSurface.dyn](#)



PointsToSurface カスタム ノードを開くと、Dynamo カスタム ノードエディタ内に上図のグラフが表示されます。または、Dynamo グラフエディタ内でカスタム ノードをダブルクリックしてカスタム ノードを開くこともできます。

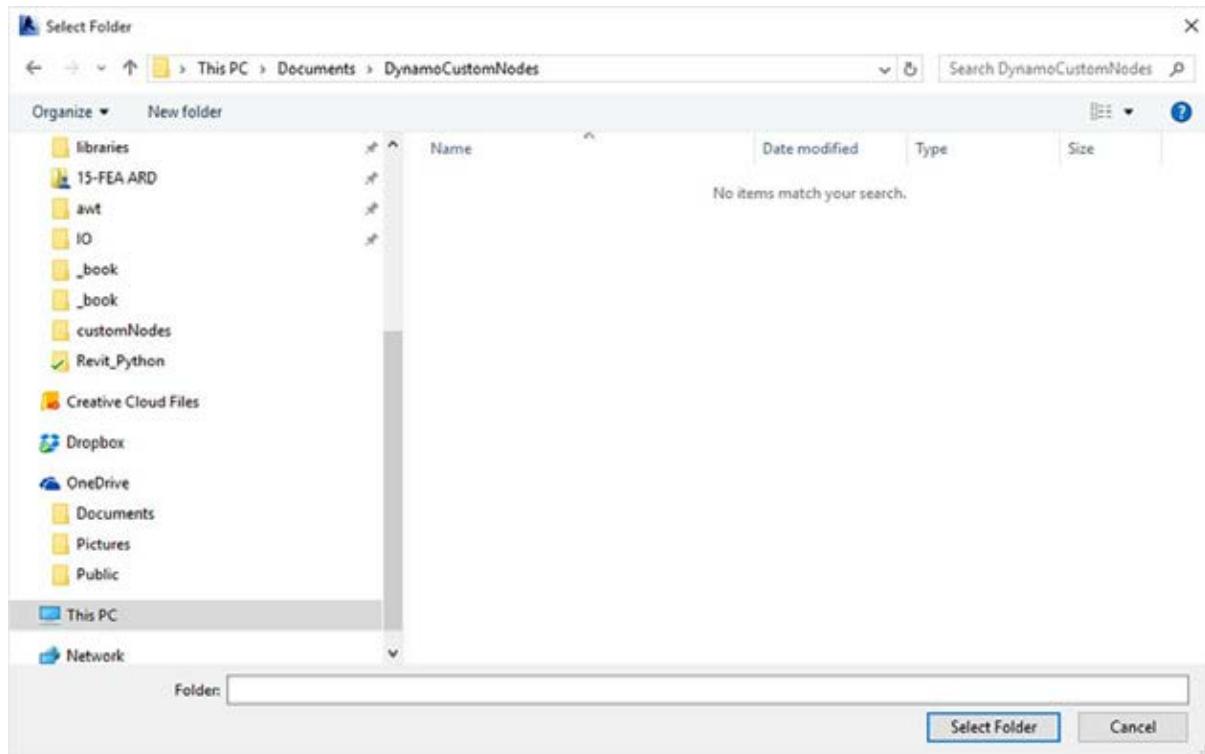


1. カスタム ノードをローカルにパブリッシュするには、キャンバス上で右クリックして、[このカスタム ノードをパブリッシュ...]を選択します。

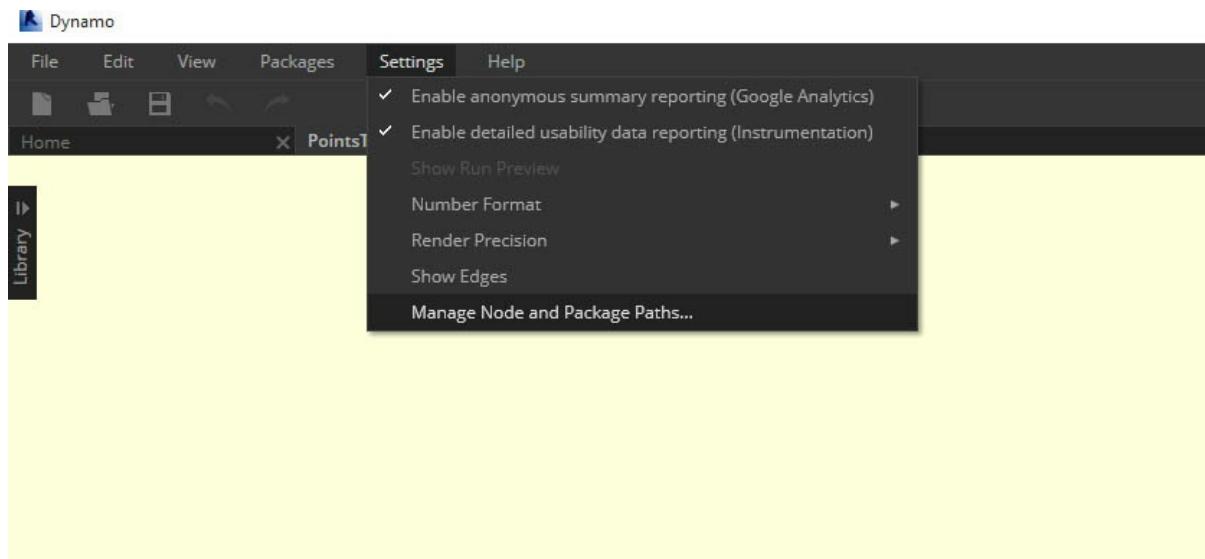


上図のように、関連情報を入力して[ローカルにパブリッシュ] [グループ] フィールドで、Dynamo メニューからアクセスできる主要な

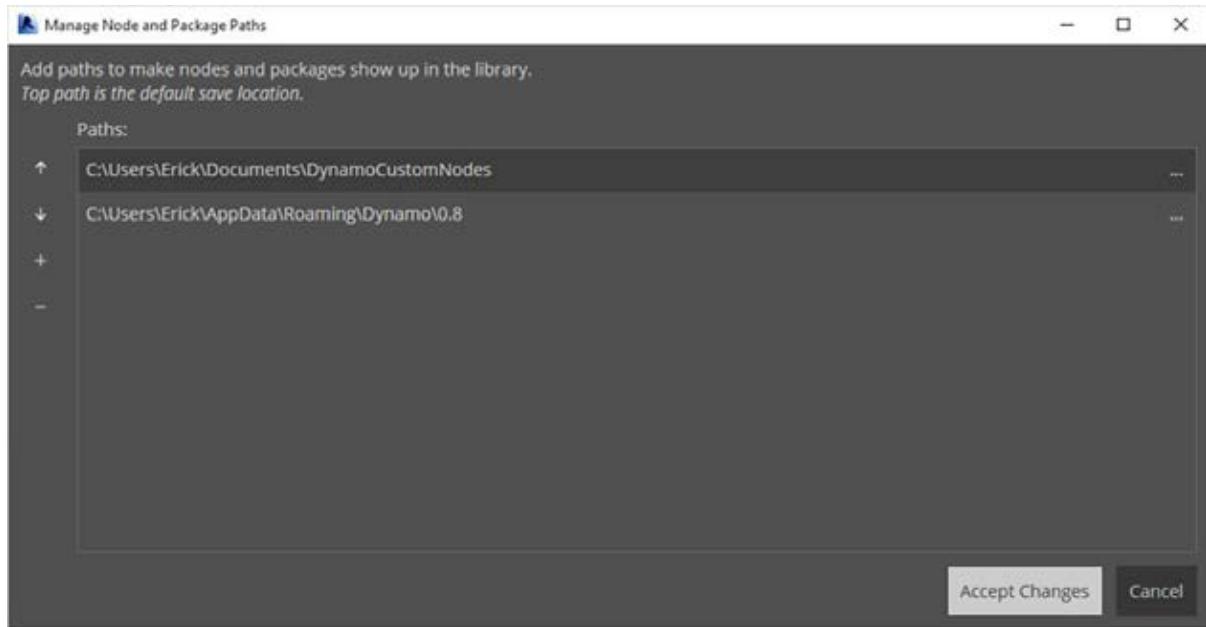
要素を定義します。



ローカルにパブリッシュするすべてのカスタム ノードを格納するフォルダを 1 つ選択します。Dynamo を読み込むたびにこのフォルダが確認されるため、このフォルダの場所は変更しないでください。このフォルダにナビゲートし、[フォルダを選択]を選択します。これで、Dynamo ノードがローカルにパブリッシュされ、プログラムを読み込むたびに、このフォルダが Dynamo ツールバーに表示されるようになります。

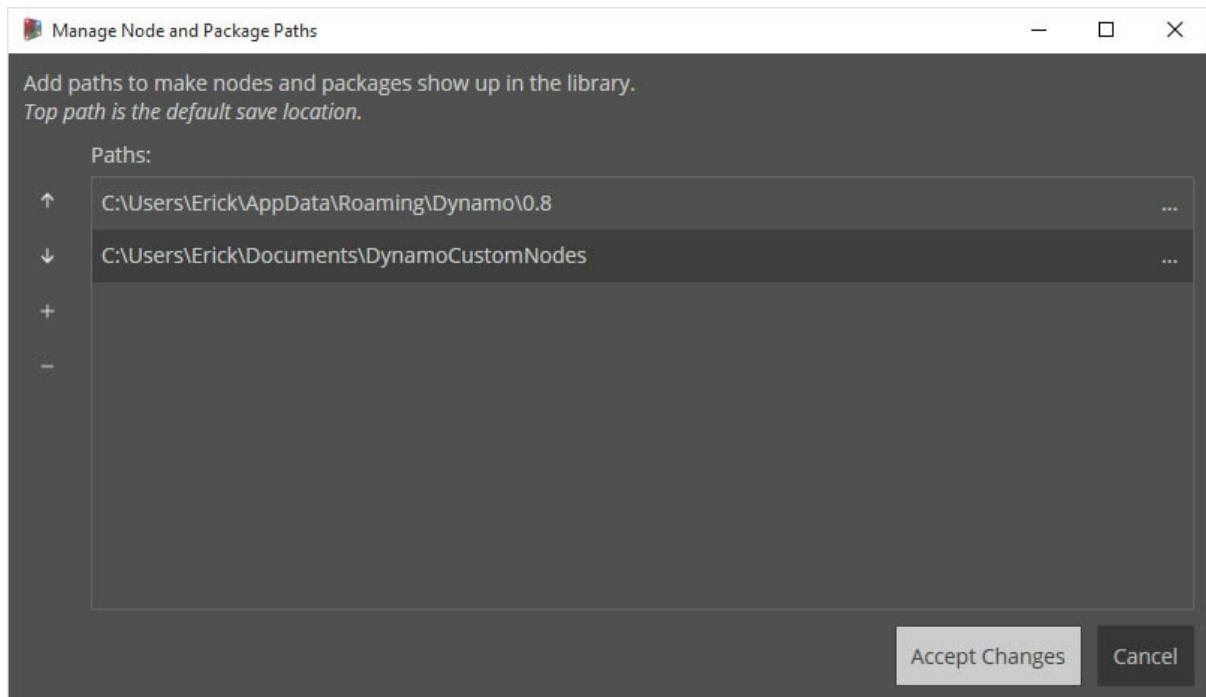


1. カスタム ノードのフォルダの場所を確認するには、[設定] > [ノードとパッケージのパスを管理...]に移動します。

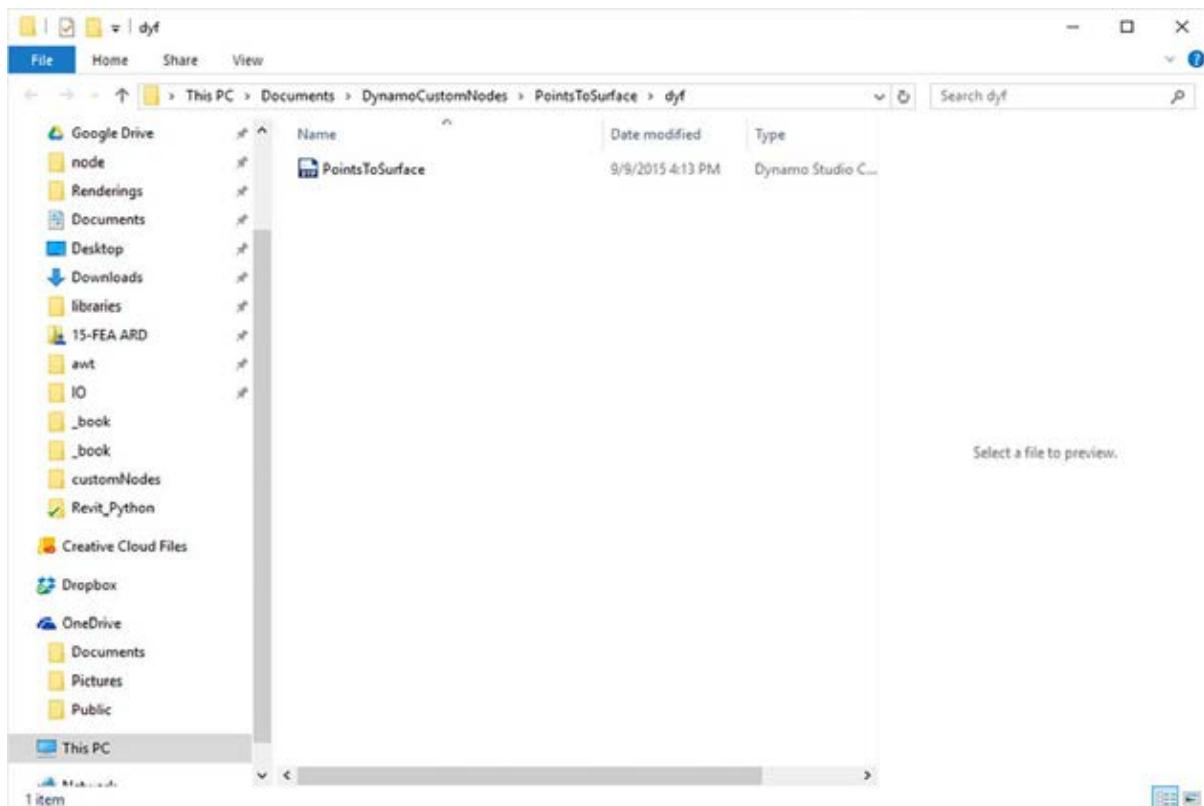


このウィンドウには、次の 2 つのパスが表示されます。AppData\Roaming\Dynamo... は、オンラインでインストールされた Dynamo パッケージの既定の場所を参照します。Documents\DynamoCustomNodes... は、ローカルにパブリッシュされたカスタム ノードの場所を参照します。\*

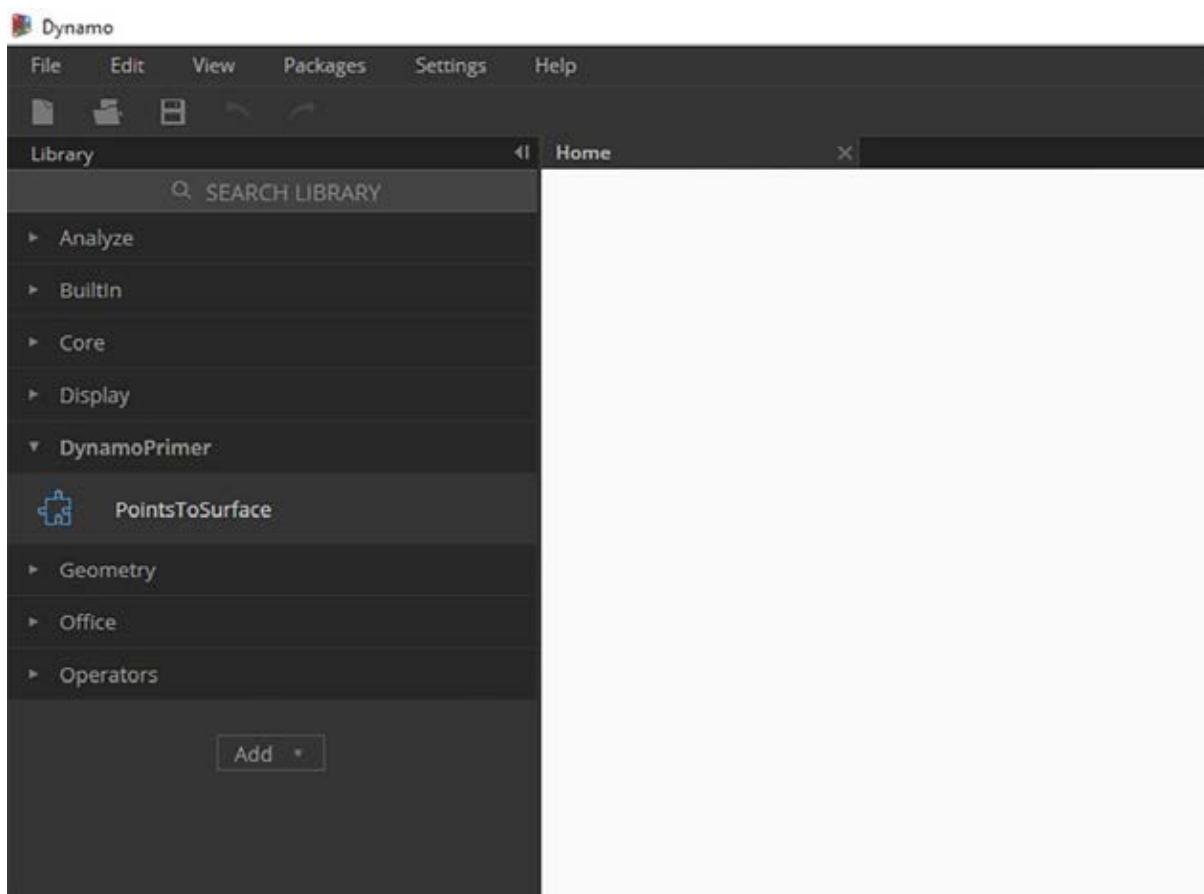
1. 上図のリスト内で、ローカル フォルダのパスを下に移動するには、フォルダ パスを選択して、パス名の左に表示されている下向き矢印をクリックします。一番上に表示されているフォルダが、パッケージがインストールされる既定のパスになります。そのため、Dynamo パッケージの既定のインストール パスを既定のフォルダのままにすると、オンライン パッケージをローカルにパブリッシュしたノードと区別することができます。\*



Dynamo の既定のパスをパッケージのインストール場所に設定するため、パス名の順序を変更しました。



このローカル フォルダにナビゲートすると、Dynamo のカスタム ノードファイルの拡張である元のカスタム ノードが`.dyn` フォルダに表示されます。このフォルダ内のファイルを編集すると、UI 上でノードが更新されます。また、メインの `DynamoCustomNode` フォルダにノードを追加すると、Dynamo の再起動時に、それらのノードがライブラリに追加されます。



これで、Dynamo を読み込むたびに、*PointsToSurface* ノードが Dynamo ライブラリの[DynamoPrimer]グループに表示されるようになります。

# Python

## Python

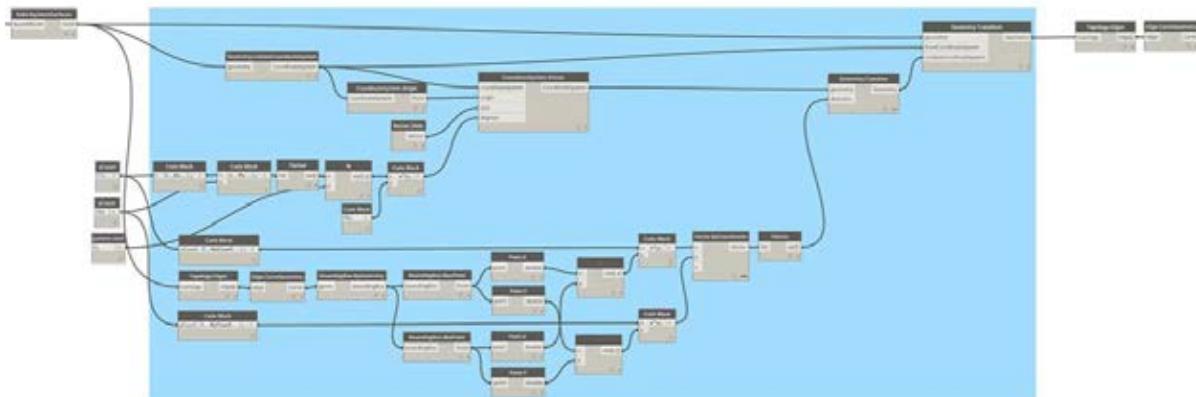


Python は、シンプルな構文が特徴の、幅広く使用されているプログラミング言語です。構文が非常に読みやすいため、他の多くの言語より簡単に習得できます。Python はモジュールとパッケージをサポートしており、既存のアプリケーションに組み込むことができます。このセクションでは、Python についての基本的な知識があることを前提として説明を進めていきます。Python を実行する方法については、[Python.org](#) の『Getting Started』ページを参照してください。

### ビジュアル プログラミングとテキスト プログラミングとの比較

Dynamo のビジュアル プログラミング環境で、テキスト プログラミングを使用するのはなぜでしょうか。第 1.1 章で説明したとおり、ビジュアル プログラミングには多くの利点があります。直感的なビジュアル インターフェースにより、特別な構文を学習することなくプログラムを作成することができます。ただし、ビジュアル プログラムは、処理が煩雑になったり、機能が不足することがあります。Python には、「if/then」の条件ステートメントやループを簡単に記述するための方法が用意されています。Python は、Dynamo の機能を拡張し、多数のノードを数行の簡潔なコード行で置き換えることができる強力なツールです。

#### ビジュアル プログラム:



#### テキスト プログラム:

```
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

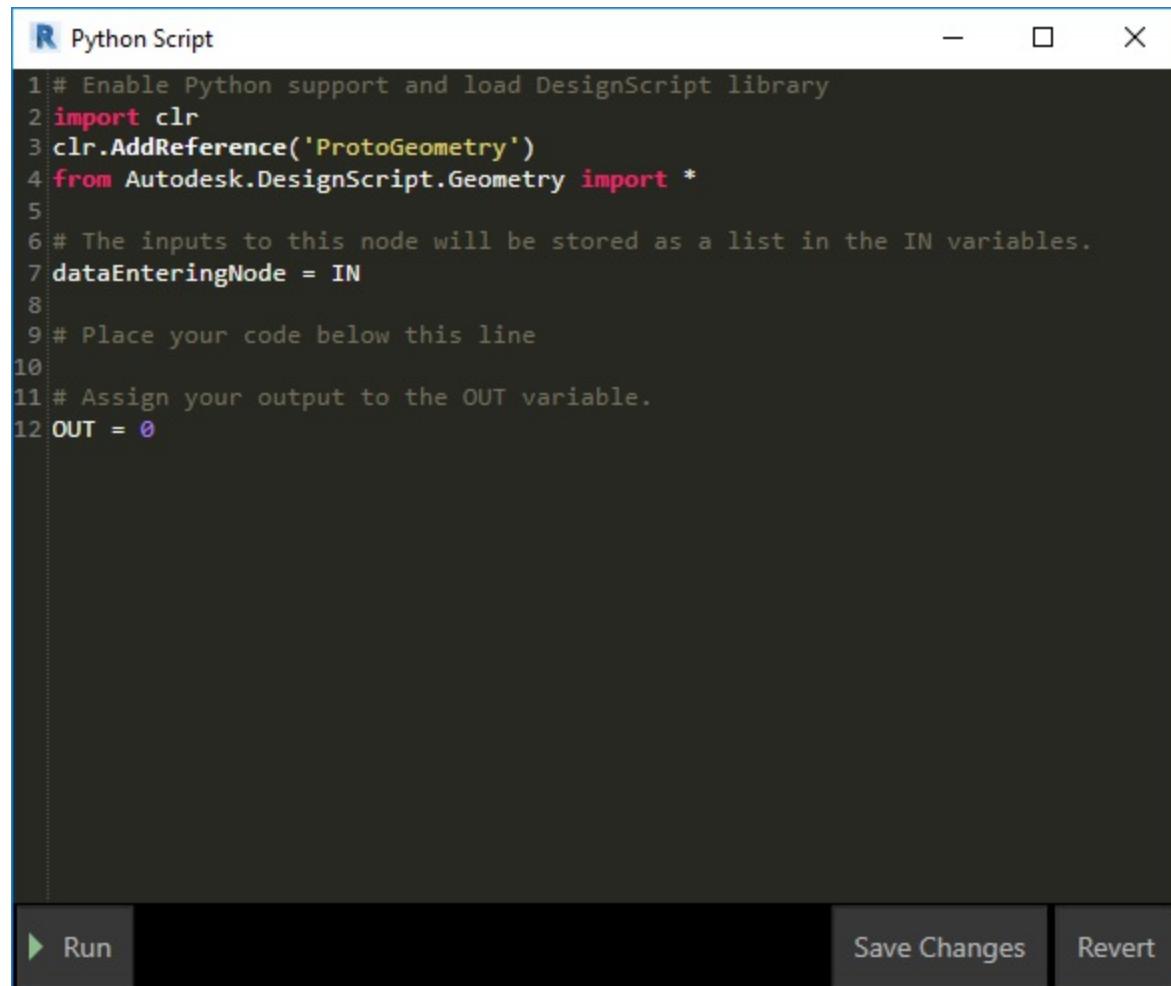
for i in xRange:
    for j in yRange:
        fromCoord = solid.ContextCoordinateSystem
        toCoord = fromCoord.Rotate(solid.ContextCoordinateSystem.Origin, Vector.ByCoordinates
        vec = Vector.ByCoordinates((xDist*i),(yDist*j),0)
        toCoord = toCoord.Translate(vec)
        solids.append(solid.Transform(fromCoord,toCoord))

OUT = solids

```

### Python Script ノード

Code Block ノードと同様に、Python Script ノードはビジュアル プログラミング環境内のスクリプト インタフェースです。Python Script ノードは、ライブラリ内の[Core] > [Scripting]で使用することができます。このノードをダブルクリックすると、Python のスクリプト エディタが開きます。ノードを右クリックして[編集...]を選択することもできます。



エディタ上部の定型文は、必要なライブラリを参照する際に役立ちます。Python Script ノードの入力値は、IN 配列に格納されます。値は、OUT 変数に割り当てられて Dynamo に返されます。

Autodesk.DesignScript.Geometry ライブラリにより、Code Block ノードと同様のドット表記を使用することができます。Dynamo の構文の詳細については、第 7.2 章と『[DesignScript Guide](#)』を参照してください。「Point.」などのジオメトリ タイプを入力すると、点の作成や点のクエリーを実行するためのメソッドのリストが表示されます。

The screenshot shows the Python Script editor window in Dynamo. The code is as follows:

```
1 # Enable Python support and load DesignScript library
2 import clr
3 clr.AddReference('ProtoGeometry')
4 from Autodesk.DesignScript.Geometry import *
5
6 # The inputs to this node will be stored as a list in the IN variables.
7 dataEnteringNode = IN
8
9 # Place your code below this line
10 Point.
11
12 # Assign OUT = Point.
13 # Assign OUT = 0
14 OUT = 0
```

A code completion dropdown is open at the end of the line 'Point.'. It lists several methods and properties of the Point class, such as Add, Approximate, AsVector, BoundingBox, ByCartesianCoordinate, ByCoordinates, ByCylindricalCoordinate, BySphericalCoordinate, ClosestPointTo, ContextCoordinateSys, Dispose, DistanceTo, and DoesIntersect.

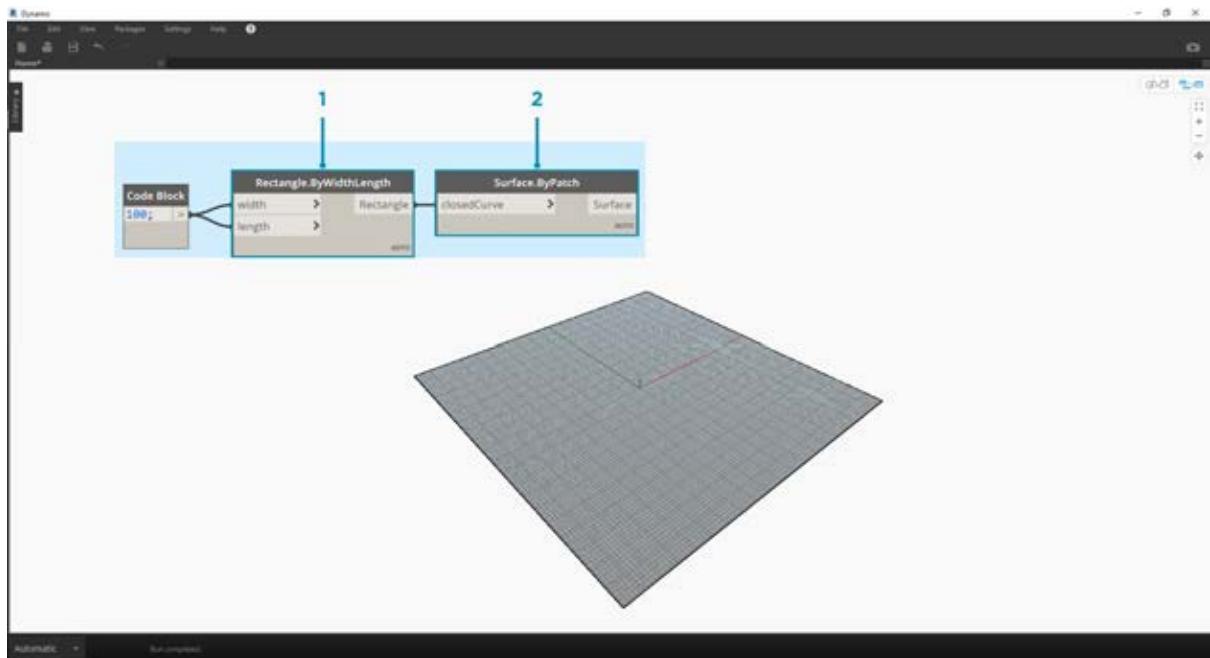
At the bottom of the editor are buttons for Run, Save Changes, and Revert.

これらのメソッドには、*ByCoordinates*などのコンストラクタ、*Add*などのアクション、X、Y、Z 座標などのクエリーがあります。

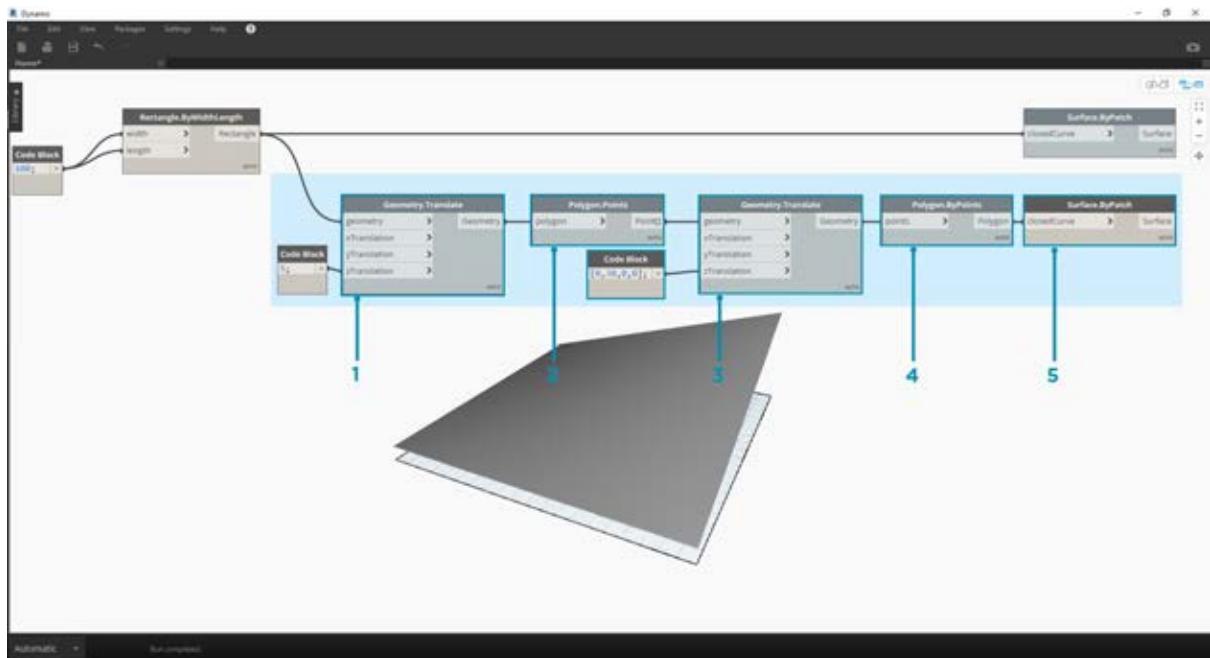
## 演習

この演習に付属しているサンプル ファイルをダウンロードしてください(右クリックして[名前を付けてリンク先を保存]を選択)。すべてのサンプルファイルの一覧については、付録を参照してください。[Python\\_Custom-Node.dyn](#)

この例では、Python Script ノードを記述してソリッド モジュールからパターンを作成し、カスタム ノードに変換します。最初に、Dynamo ノードを使用してソリッド モジュールを作成します。

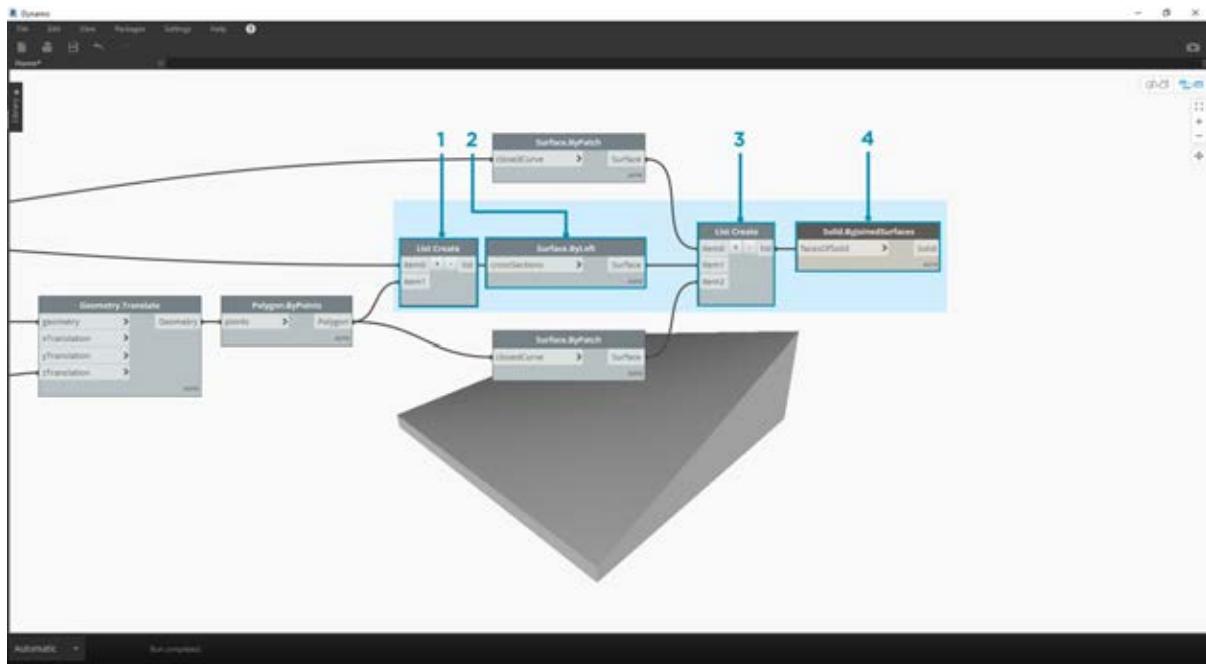


1. **Rectangle.ByWidthLength** ノードを使用して、ソリッドのベースとなる長方形を作成します。
2. **Surface.ByPatch** ノードの *closedCurve* 入力に Rectangle 出力を接続し、下部サーフェスを作成します。



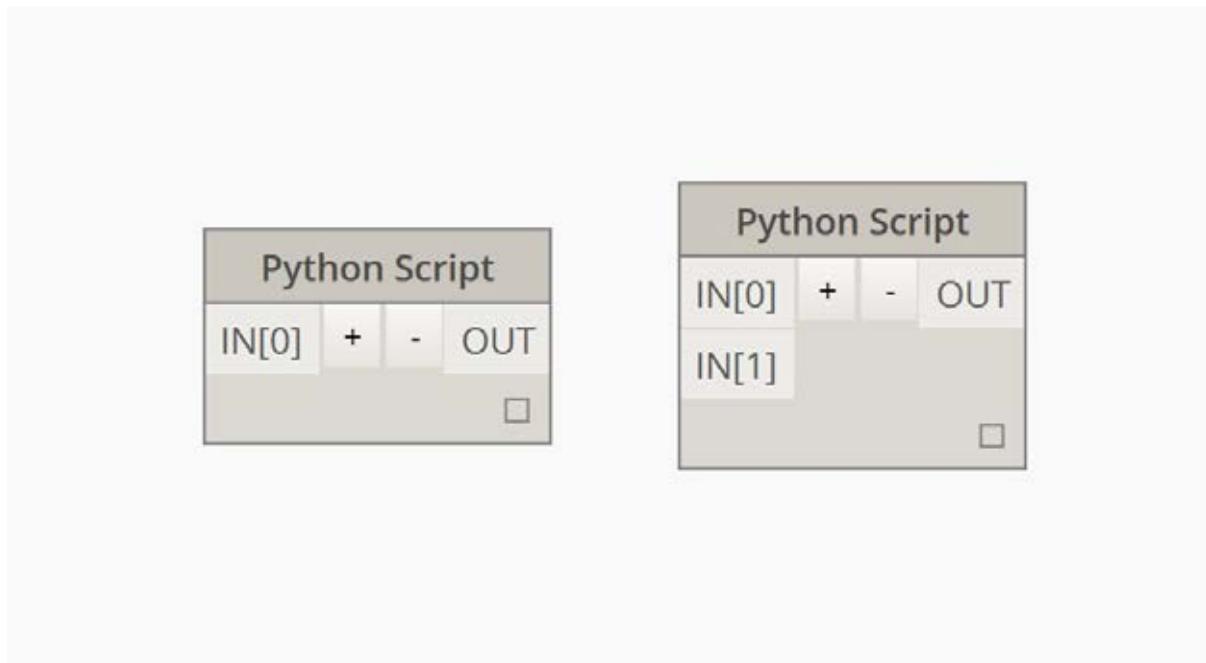
1. **Geometry.Translate** ノードの *geometry* 入力に **Rectangle** 出力を接続し、長方形を上に移動します。次に、Code Block ノードを使用してソリッドの厚さを指定します。
2. **Polygon.Points** ノードを使用して、変換された長方形に対してクエリーを実行し、頂点を抽出します。
3. **Geometry.Translate** ノードを使用して、4 つの点に対する 4 つの値のリストを作成します。この操作により、ソリッドの 1 つの頂点が上に移動します。
4. 変換後の点を **Polygon.ByPoints** ノードで使用して、上部ポリゴンを再作成します。
5. **Surface.ByPatch** ノードを使用してポリゴンを結合し、上部サーフェスを作成します。

これで、上部サーフェスと下部サーフェスが作成されました。次に、2 つのプロファイルの間をロフトしてソリッドの側面を作成しましょう。



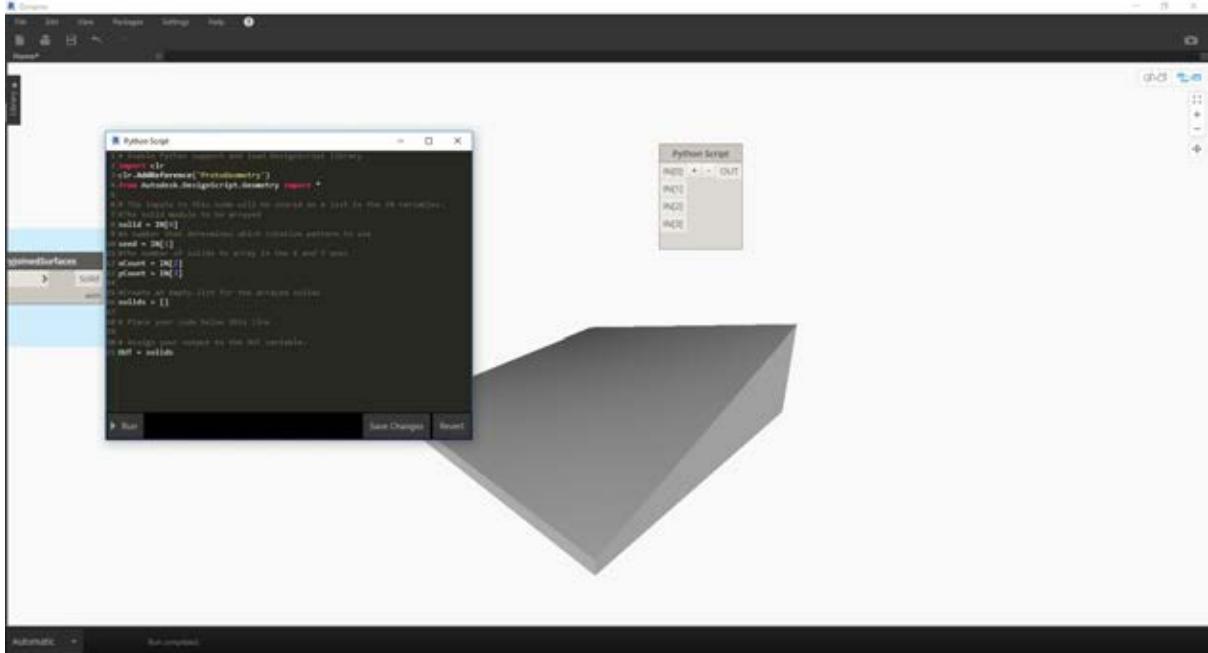
1. List.Create ノードの index 入力に、底面の長方形と上面のポリゴンを接続します。
2. Surface.ByLoft ノードを使用して 2 つのプロファイルをロフトし、ソリッドの側面を作成します。
3. List.Create ノードの index 入力に上部サーフェス、側面サーフェス、下部サーフェスを接続して、サーフェスのリストを作成します。
4. Solid.ByJoinedSurfaces ノードを使用してサーフェスを結合し、ソリッド モジュールを作成します。

これで、ソリッドが作成されました。次に、ワークスペースに Python Script ノードをドロップします。



エディタを閉じてノード上の[+]アイコンをクリックし、ノードに入力を追加します。入力には IN[0]、IN[1]などという名前が付いています。これらはリスト内の項目を表しています。

最初に、入力と出力を定義しましょう。ノードをダブルクリックして、Python エディタを開きます。



```
# 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
```

このコードの意味については、演習を進めながら説明していきます。ここで、ソリッドモジュールを配列化するためには、どのような情報が必要になるかを考慮する必要があります。最初に、移動距離を決定するために、ソリッドの寸法を知る必要があります。境界ボックスにはバグがあるため、境界ボックスを作成するにはエッジ曲線のジオメトリを使用する必要があります。

```
1 # Enable Python support and load DesignScript library
2 import clr
3 clr.AddReference('ProtoGeometry')
4 from Autodesk.DesignScript.Geometry import *
5
6 # The inputs to this node will be stored as a list in the IN variables.
7 #The solid module to be arrayed
8 solid = IN[0]
9 #A number that determines which rotation pattern to use
10 seed = IN[1]
11 #The number of solids to array in the X and Y axes
12 xCount = IN[2]
13 yCount = IN[3]
14
15 #Create an empty list for the arrayed solids
16 solids = []
17 # Create an empty list for the edge curves
18 crvs = []
19
20 # Place your code below this line
21 #Loop through edges and append corresponding curve geometry to the list
22 for edge in solid.Edges:
23     crvs.append(edge.CurveGeometry)
24 #Get the bounding box of the curves
25 bbox = BoundingBox.ByGeometry(crvs)
26
27 #Get the X and Y translation distance based on the bounding box
28 yDist = bbox.MaxPoint.Y-bbox.MinPoint.Y
29 xDist = bbox.MaxPoint.X-bbox.MinPoint.X
30
31 # Assign your output to the OUT variable.
32 OUT = solids
```

Run      Save Changes      Revert

Dynamo の Python Script ノードを確認すると、Dynamo のノードのタイトルと同じ構文が使用されていることがわかります。コメント付きのコードを次に示します。

```
# 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

```

ここでは、ソリッドのモジュールの移動と回転を行うため、Geometry.Transform の操作を使用しましょう。Geometry.Transform ノードを確認すると、ソリッドを変換するにはソース座標系とターゲット座標系が必要になることがわかります。この場合、ソース座標系はソリッドのコンテキストの座標系で、ターゲット座標系は配列化された各モジュールの別の座標系になります。そのため、x 値と y 値をループして、座標系を毎回異なる距離と方向で変換する必要があります。

```

1 # Enable Python support and load DesignScript library
2 import clr
3 clr.AddReference('ProtoGeometry')
4 from Autodesk.DesignScript.Geometry import *
5
6 # The inputs to this node will be stored as a list in the IN variables.
7 #The solid module to be arrayed
8 solid = IN[0]
9 #A number that determines which rotation pattern to use
10 seed = IN[1]
11 #The number of solids to array in the X and Y axes
12 xCount = IN[2]
13 yCount = IN[3]
14
15 #Create an empty list for the arrayed solids
16 solids = []
17 # Create an empty list for the edge curves
18 crvs = []
19
20 # Place your code below this line
21 #Loop through edges and append corresponding curve geometry to the list
22 for edge in solid.Edges:
23     crvs.append(edge.CurveGeometry)
24 #Get the bounding box of the curves
25 bbox = BoundingBox.ByGeometry(crvs)
26
27 #Get the X and Y translation distance based on the bounding box
28 yDist = bbox.MaxPoint.Y-bbox.MinPoint.Y
29 xDist = bbox.MaxPoint.X-bbox.MinPoint.X
30 #Get the source coordinate system
31 fromCoord = solid.ContextCoordinateSystem
32
33 #Loop through X and Y
34 for i in range(xCount):
35     for j in range(yCount):
36         #Rotate and translate the coordinate system
37         toCoord = fromCoord.Rotate(solid.ContextCoordinateSystem.Origin,Vector.ByCoordinates(0,0,1),(90*(i+j%seed)))
38         vec = Vector.ByCoordinates((xDist*i),(yDist*j),0)
39         toCoord = toCoord.Translate(vec)
40         #Transform the solid from the source coord system to the target coord system and append to the list
41         solids.append(solid.Transform(fromCoord,toCoord))
42
43 # Assign your output to the OUT variable.
44 OUT = solids

```



ここで、Dynamo の Python Script ノードを確認します。コメント付きのコードを次に示します。

```

# 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 = []

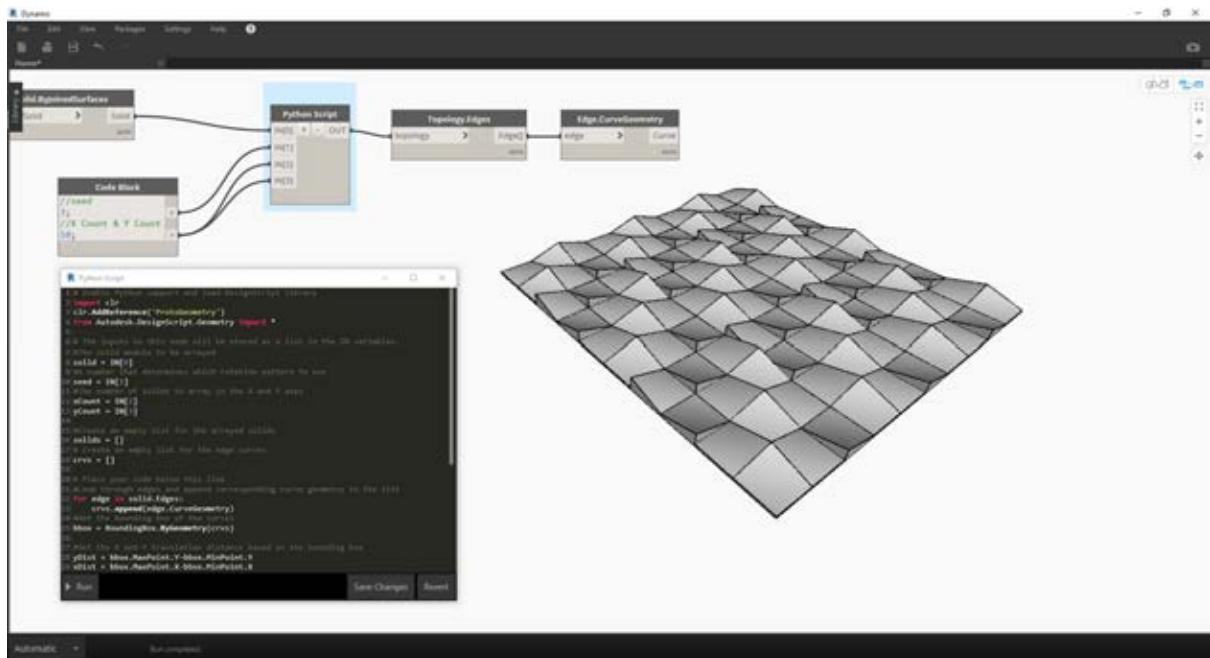
# Place your code below this line
#Loop through edges and append corresponding curve geometry to the list
for edge in solid.Edges:
    crvs.append(edge.CurveGeometry)
#Get the bounding box of the curves
bbox = BoundingBox.ByGeometry(crvs)

#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
        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 app
        solids.append(solid.Transform(fromCoord,toCoord))

# Assign your output to the OUT variable.
OUT = solids

```

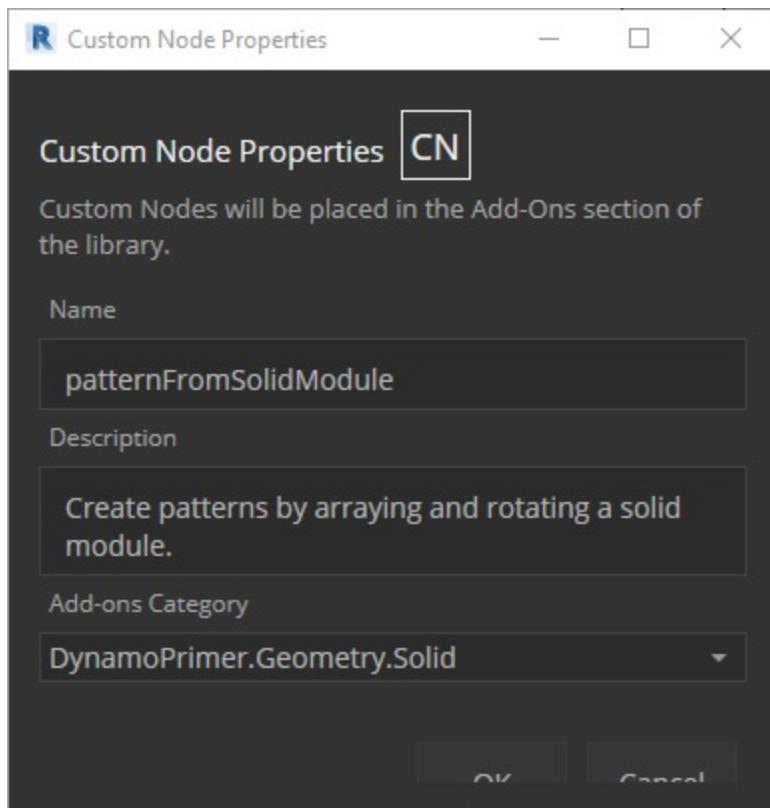


Python Script ノード上で[実行]をクリックすると、コードを実行することができます。



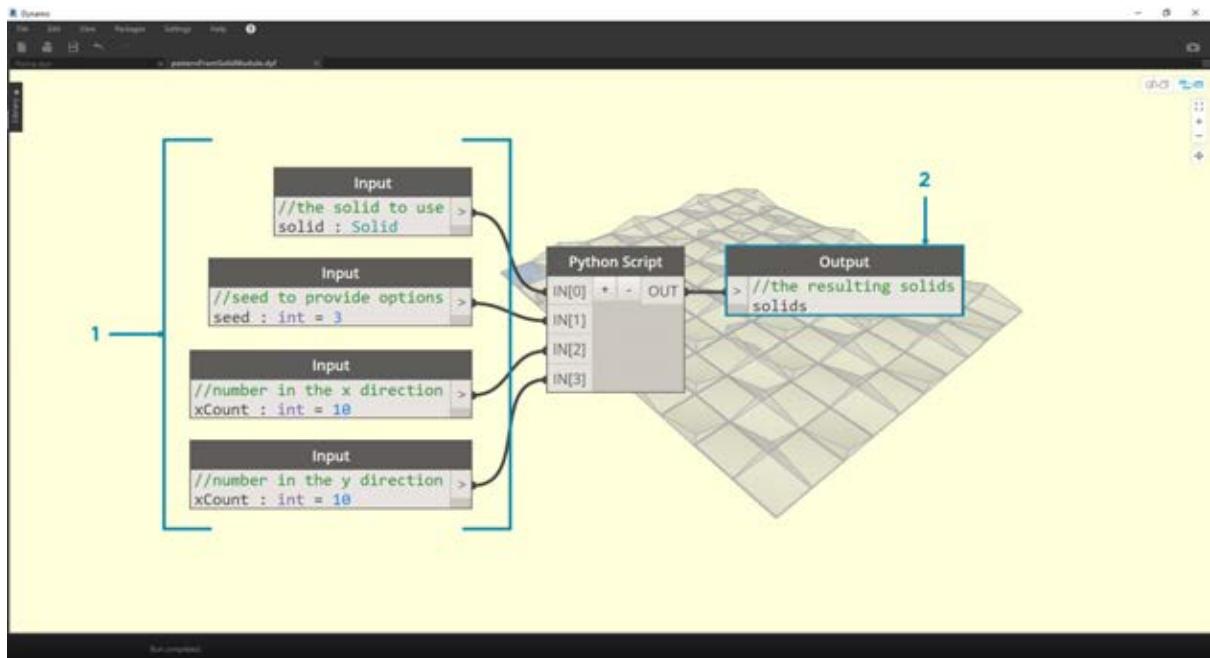
シード値を変更すると、異なるパターンが作成されます。ソリッド モジュールのパラメータを変更して、異なるエフェクトを作成することもできます。Dynamo 2.0 では、Python ウィンドウを閉じることなく、シードを変更して[実行]をクリックすることができます。

これで、便利な Python Script ノードが作成されました。このノードをカスタム ノードとして保存しましょう。Python Script ノードを選択して右クリックし、[選択からノードを新規作成]を選択します。

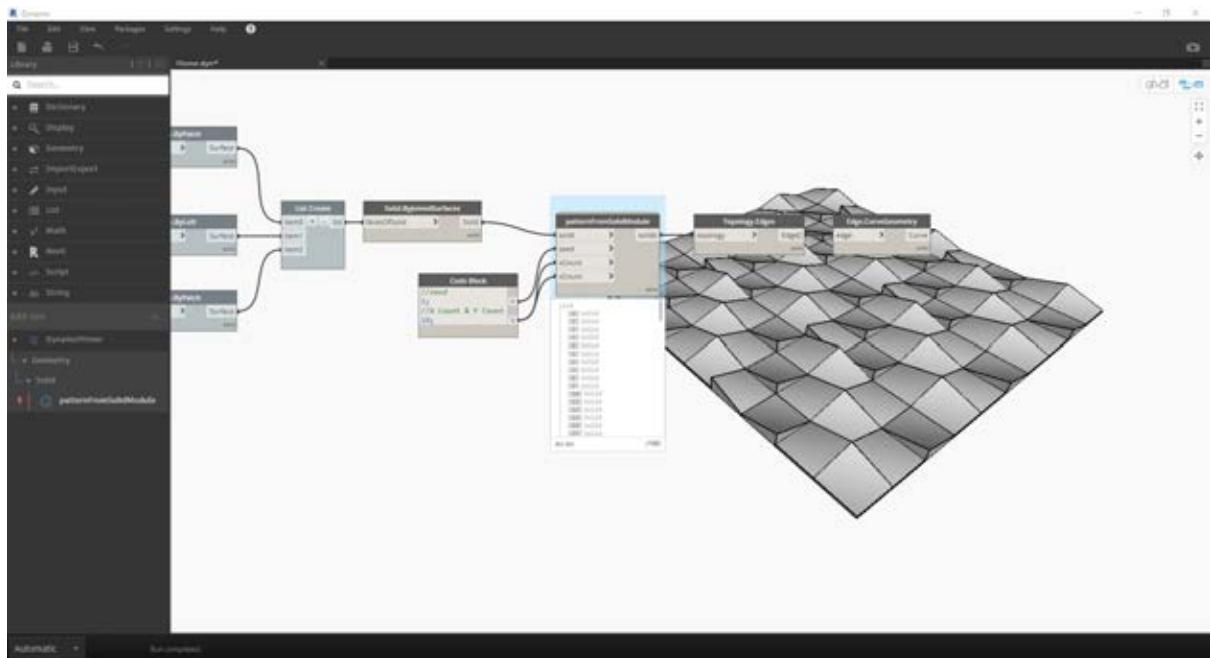


名前、説明、カテゴリを設定します。

この操作により、カスタム ノードを編集するための新しいワークスペースが開きます。



1. 各 Input ノードの入力名をわかりやすい名前に変更し、データ タイプと既定値を追加します。
2. Output ノードの出力名を変更し、このノードを .dyf ファイルとして保存します。



カスタム ノードに変更内容が反映されます。

# Python と Revit

## Python と Revit

前のセクションでは、Dynamo で Python スクリプトを使用する方法について説明しました。このセクションでは、スクリプティング環境に Revit ライブラリを接続する方法を見てみましょう。ここまで手順で、次のコード ブロックの最初の 3 行を使用して、Dynamo の Core ノードが既に読み込まれています。数行のコードを追加するだけで、Revit の各種のノード、要素、ドキュメント マネージャを読み込むことができます。

```
import clr
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *

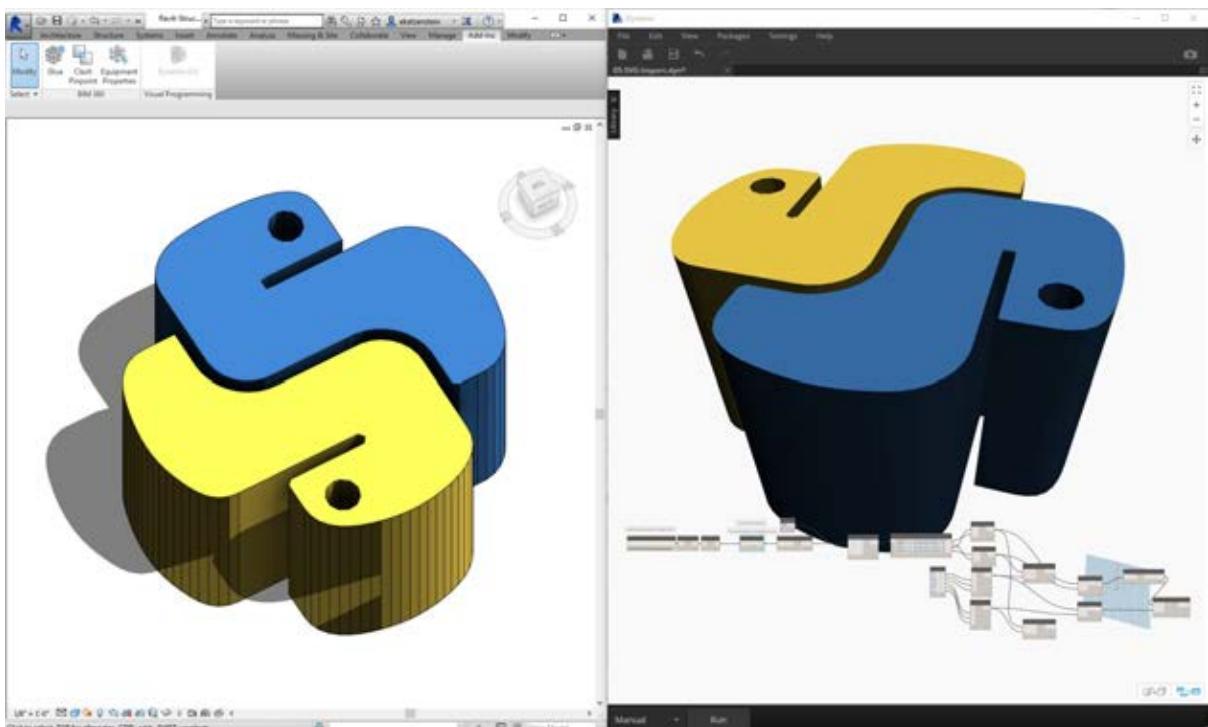
# Import RevitNodes
clr.AddReference("RevitNodes")
import Revit

# Import Revit elements
from Revit.Elements import *

# Import DocumentManager
clr.AddReference("RevitServices")
import RevitServices
from RevitServices.Persistence import DocumentManager

import System
```

これにより、Revit API にアクセスし、任意の Revit タスクでカスタム スクリプトを使用できるようになります。ビジュアル プログラミングのプロセスと Revit API スクリプトを組み合わせることにより、コラボレーションやツールの開発が容易になります。たとえば、BIM マネージャと回路設計者が、同じグラフを使用して同時に作業することができます。こうしたコラボレーションにより、モデルの設計と施工を改善することができます。



## プラットフォーム固有の API

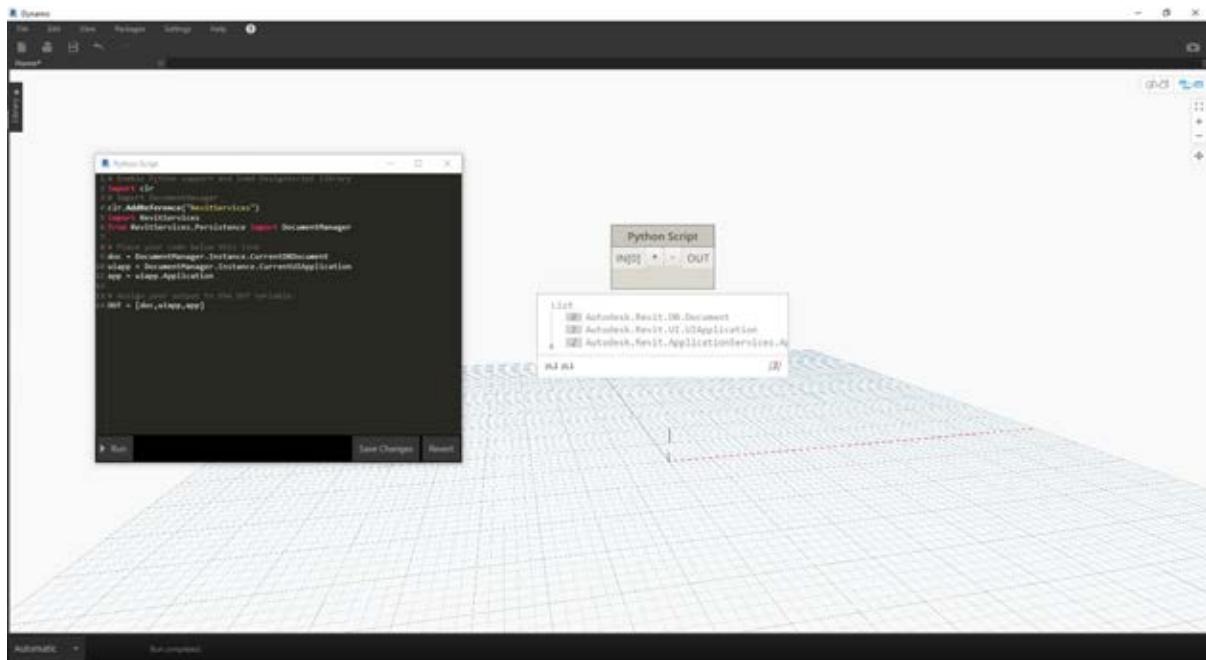
Dynamo プロジェクトの背景には、プラットフォーム実装のスコープを拡大するという計画があります。そのため、Dynamo には新しいプログラムが追加されていく予定になっています。ユーザは、Python スクリプティング環境からプラットフォーム固有の API にアクセスできるようになります。このセクションでは Revit を扱いますが、今後は章の数を増やして、別のプラットフォーム上でのスクリプティングに関する説明を追加する予定になっています。また、さまざまな [IronPython](#) ライブラリにアクセスして Dynamo に読み込むことができるようになりました。

次の例では、Dynamo で Python を使用して、Revit 固有の操作を実行する方法について説明します。Python と Dynamo および Revit との関係の詳細については、[Dynamo の Wiki ページ](#)を参照してください。Python と Revit のもう 1 つの便利なリソースは、[Revit Python Shell](#) プロジェクトです。

## 演習 01

新しい Revit プロジェクトを作成します。この演習に付属しているサンプル ファイルをダウンロードしてください(右クリックして[名前を付けてリンク先を保存]を選択)。すべてのサンプルファイルの一覧については、付録を参照してください。[Revit-Doc.dyn](#)

これ以降の一連の演習では、Dynamo for Revit における基本的な Python スクリプトについて説明します。この演習では、Revit のファイルと要素を使用します。また、Revit と Dynamo 間の通信についても説明します。



ここでは、Dynamo セッションにリンクされた Revit ファイルの *doc*、*uiapp*、*app* を取得するための一般的な方法について説明します。Revit API を使用したことのあるプログラマならば、上図の Watch リストのような項目を見たことがあるでしょう。これらの項目を見たことがなくても、特に問題はありません。これ以降の演習で、別の例を使用して説明します。

RevitServices を読み込み、Dynamo のドキュメントデータを取得するには、次のようなスクリプトを記述します。

A screenshot of the Dynamo software interface. It shows a Python Script node with the same code as the previous screenshot. At the bottom of the interface, there is a "Run" button, which is highlighted with a red rectangle. The other buttons "Save Changes" and "Revert" are also visible at the bottom.

ここで、Dynamo の Python Script ノードを確認します。コメント付きのコードを次に示します。

```
# Enable Python support and load DesignScript library
import clr
# Import DocumentManager
clr.AddReference("RevitServices")
import RevitServices
from RevitServices.Persistence import DocumentManager

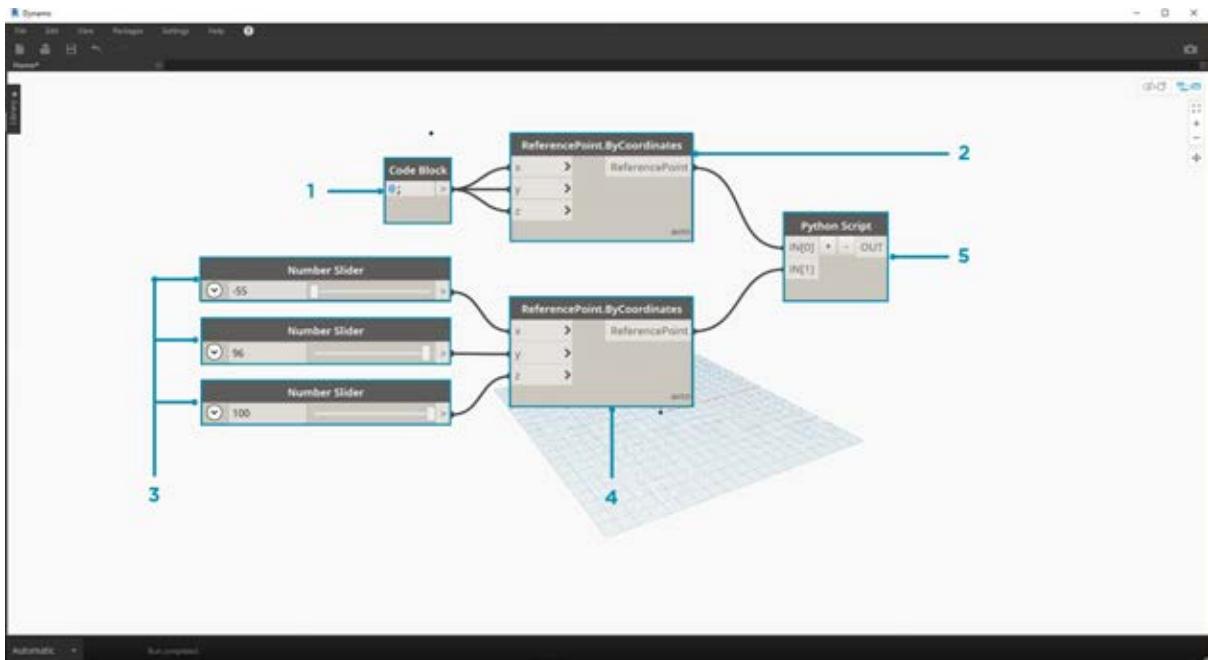
# Place your code below this line
doc = DocumentManager.Instance.CurrentDBDocument
uiapp = DocumentManager.Instance.CurrentUIApplication
app = uiapp.Application

# Assign your output to the OUT variable.
OUT = [doc,uiapp,app]
```

## 演習 02

この演習に付属しているサンプル ファイルをダウンロードしてください(右クリックして[名前を付けてリンク先を保存]を選択)。すべてのサンプルファイルの一覧については、付録を参照してください。[Revit-ReferenceCurve.dyn](#)

この演習では、Dynamo の Python Script ノードを使用して、Revit 内に単純なモデル曲線を作成します。



最初に、上図に示す一連のノードを作成します。次に、Dynamo のノードを使用して、Revit 内に 2 つの参照点を作成します。

最初に、Revit 内に新しいコンセプト マス ファミリを作成します。Dynamo を起動し、上図に示す一連のノードを作成します。次に、Dynamo のノードを使用して、Revit 内に 2 つの参照点を作成します。

1. Code Block ノードを作成し、値を「0」に設定します。
2. この値を、ReferencePoint.ByCoordinates ノードの X、Y、Z 入力に接続します。
3. -100 ~ 100 の範囲内で、ステップ値が 1 の Number Slider ノードを 3 つ作成します。
4. 各 Number Slider ノードを ReferencePoint.ByCoordinates ノードに接続します。
5. Python Script ノードをワークスペースに追加し、このノードの[+]ボタンをクリックして入力をもう 1 つ追加して、各入力に参照点を接続します。Python Script ノードを開きます。

The screenshot shows the Python Script node in the Dynamo interface. The code is as follows:

```
1 # Enable Python support and load DesignScript library
2 import clr
3 # Import RevitNodes
4 clr.AddReference("RevitNodes")
5 import Revit
6 # Import Revit elements
7 from Revit.Elements import *
8 import System
9
10 # The inputs to this node will be stored as a list in the IN variables.
11 startRefPt = IN[0]
12 endRefPt = IN[1]
13
14 # Place your code below this line
15 #define system array to match with required inputs
16 refPtArray = System.Array[ReferencePoint]([startRefPt, endRefPt])
17
18 # Assign your output to the OUT variable.
19 OUT = CurveByPoints.ByReferencePoints(refPtArray)
```

The line `refPtArray = System.Array[ReferencePoint]([startRefPt, endRefPt])` is highlighted with a blue selection bar and has a small number '1' to its right.

At the bottom, there are three buttons: Run, Save Changes, and Revert.

ここで、Dynamo の Python Script ノードを確認します。コメント付きのコードを次に示します。

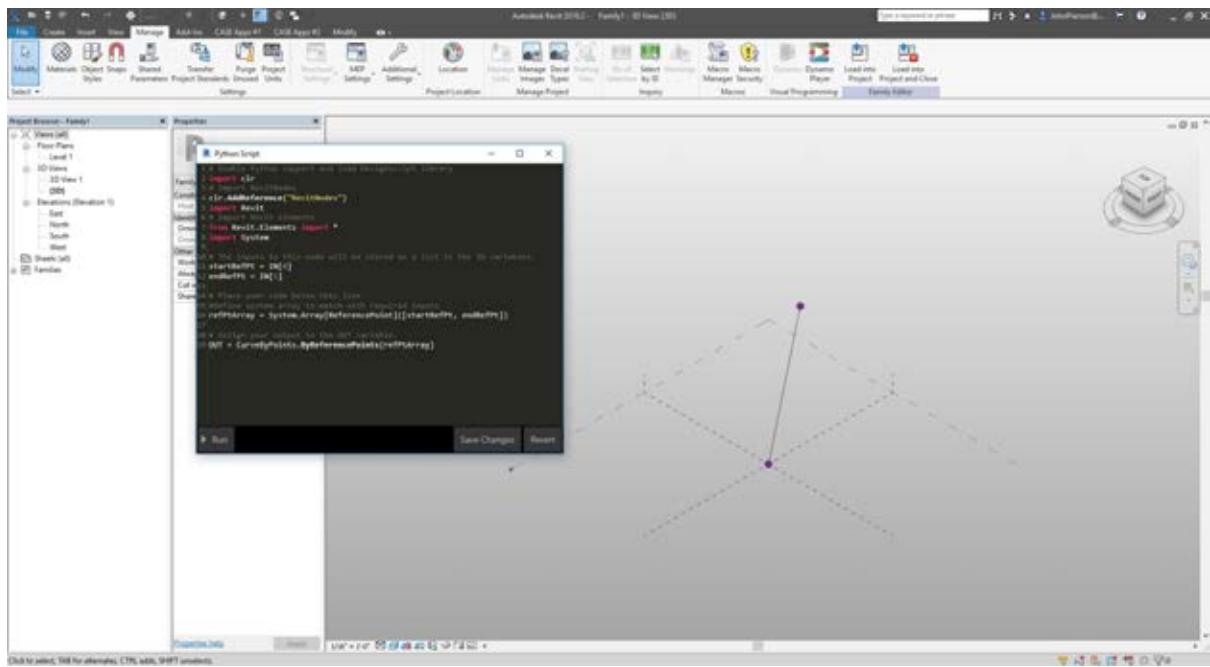
1. Revit の入力には、Python リストではなく `System.Array` が必要です。これは 1 行のコードに過ぎませんが、引数のタイプに注意すると、Revit での Python プログラミングが容易になります。

```
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)
```

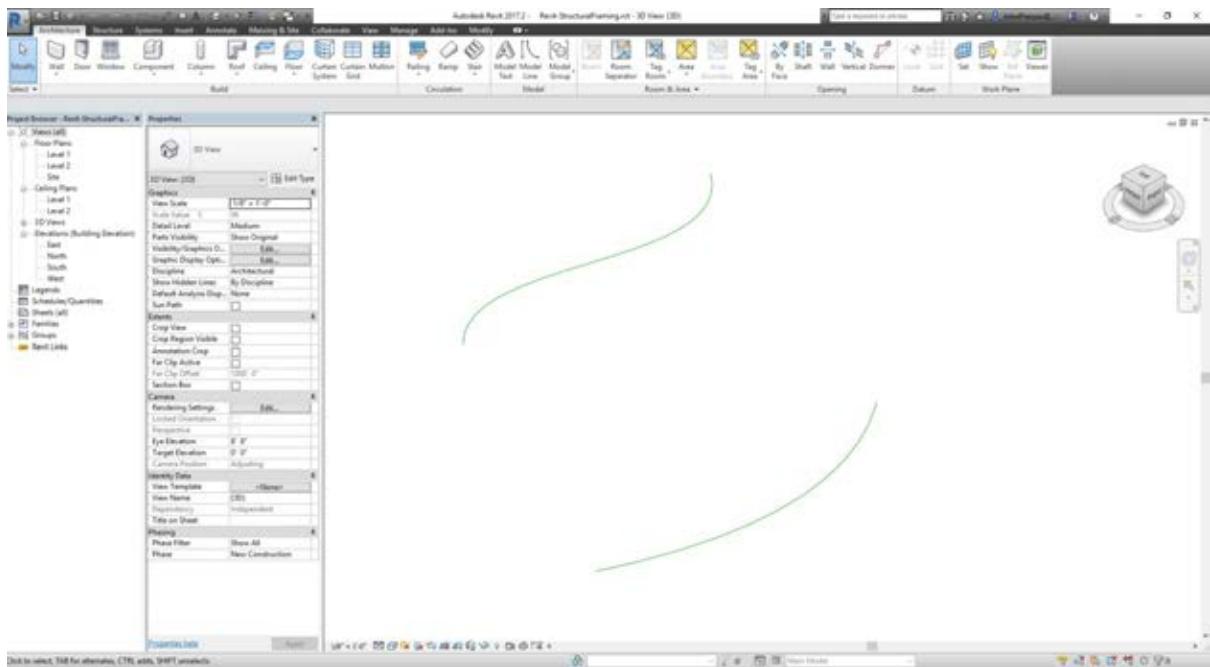


ここまで手順で、Dynamo で Python を使用して、線分で接続された 2 つの参照点を作成しました。次の演習で、さらに操作を進めてみましょう。

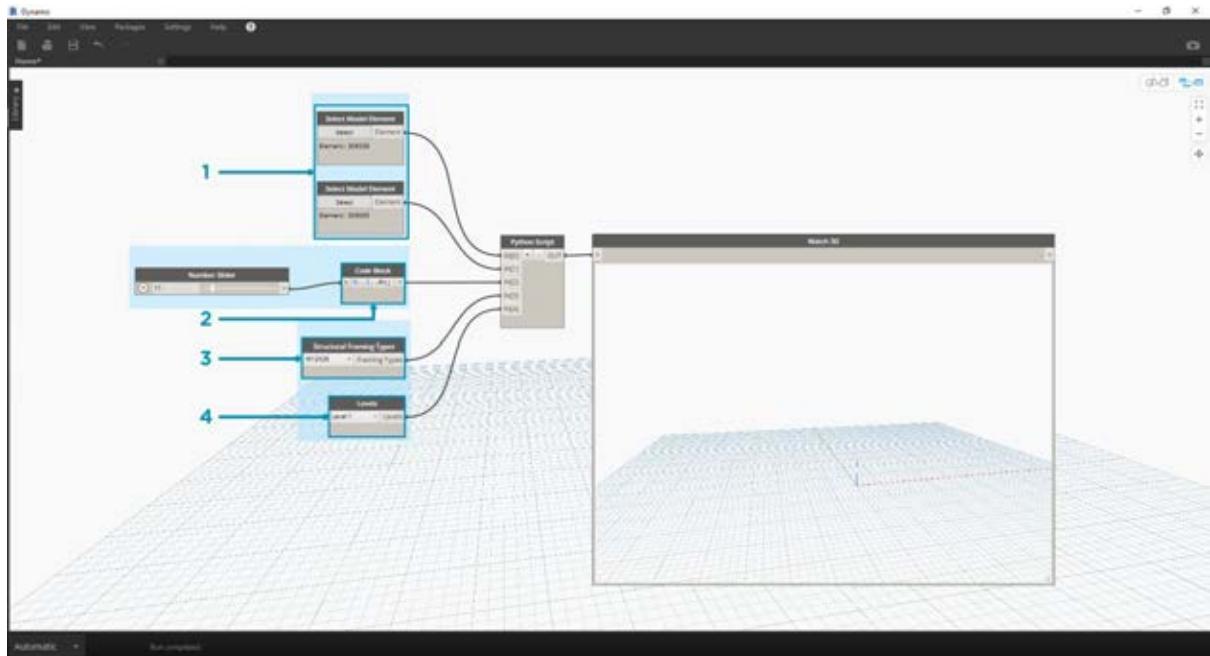
### 演習 03

この演習に付属しているサンプル ファイルをダウンロードして解凍してください(右クリックして[名前を付けてリンク先を保存]を選択)。すべてのサンプルファイルの一覧については、付録を参照してください。[Revit-StructuralFraming.zip](#)

この演習では、簡単な操作を実行することにより、Revit と Dynamo 間で相互にデータやジオメトリを接続する場合の要点について説明します。最初に Revit-StructuralFraming.rvt を開き、次に Dynamo を読み込んで、Revit-StructuralFraming.dyn を開きます。



この Revit ファイルは、基本的なファイルです。レベル 1 とレベル 2 にそれぞれ 1 本ずつ、2 本の異なる参照曲線が描画されています。これらの曲線を Dynamo に読み込み、ライブリンクを作成します。



このファイルでは、Python Script ノードの 5 つの入力に一連のノードが接続されています。

1. 各 **Select Model Element** ノードの[選択]ボタンをクリックし、Revit 内の対応する曲線を選択します。
2. **Code Block** ノードで「`0..1..#x;`」という構文を使用して、0 ~ 20 までの範囲を持つ Integer Slider ノードを `x` 入力に接続します。この操作により、2 本の曲線の間に作成する梁の数を指定します。
3. **Structural Framing Types** ノードのドロップダウン メニューで、既定の W12x26 梁を選択します。
4. **Levels** ノードで、「Level 1」を選択します。

The screenshot shows a Python script editor window titled "Python Script". The code is written in Python and performs the following steps:

- Imports necessary modules: `clr`, `Dynamo Geometry`, `ProtoGeometry`, `Autodesk.DesignScript.Geometry`, `RevitNodes`, `Revit`, and `Revit.Elements`.
- Defines input parameters: `framingType` (from IN[3]), `designLevel` (from IN[4]), and `OUT` (an empty list).
- Loops through values in IN[2] (which contains two Revit curves).
- For each value, it defines points on each curve using `Curve.PointAtParameter`.
- Creates a Dynamo line between the corresponding points.
- Creates a Revit element (beam) from the Dynamo line using `StructuralFraming.BeamByCurve`.
- Converts the Revit element into a list of Dynamo surfaces.
- Appends the surfaces to the `OUT` list.

At the bottom of the window, there are buttons for "Run", "Save Changes", and "Revert".

```
1 import clr
2 #import Dynamo Geometry
3 clr.AddReference('ProtoGeometry')
4 from Autodesk.DesignScript.Geometry import *
5 # Import RevitNodes
6 clr.AddReference("RevitNodes")
7 import Revit
8 # Import Revit elements
9 from Revit.Elements import *
10 import System
11
12 #Query Revit elements and convert them to Dynamo Curves
13 crvA=IN[0].Curves[0]
14 crvB=IN[1].Curves[0]
15
16 #Define input Parameters
17 framingType=IN[3]
18 designLevel=IN[4]
19
20 #Define "out" as a list
21 OUT=[]
22
23 for val in IN[2]:
24     #Define Dynamo Points on each curve
25     ptA=Curve.PointAtParameter(crvA,val)
26     ptB=Curve.PointAtParameter(crvB,val)
27     #Create Dynamo line
28     beamCrv=Line.ByStartPointEndPoint(ptA,ptB)
29     #create Revit Element from Dynamo Curves
30     beam = StructuralFraming.BeamByCurve(beamCrv,designLevel,framingType)
31     #convert Revit Element into list of Dynamo Surfaces
32     OUT.append(beam.Faces)
33
```

この Python コードは、これまでのコードよりも行数が多くなっていますが、コード行の前後のコメントを参照すると、プロセス内の処理内容を確認することができます。

```
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

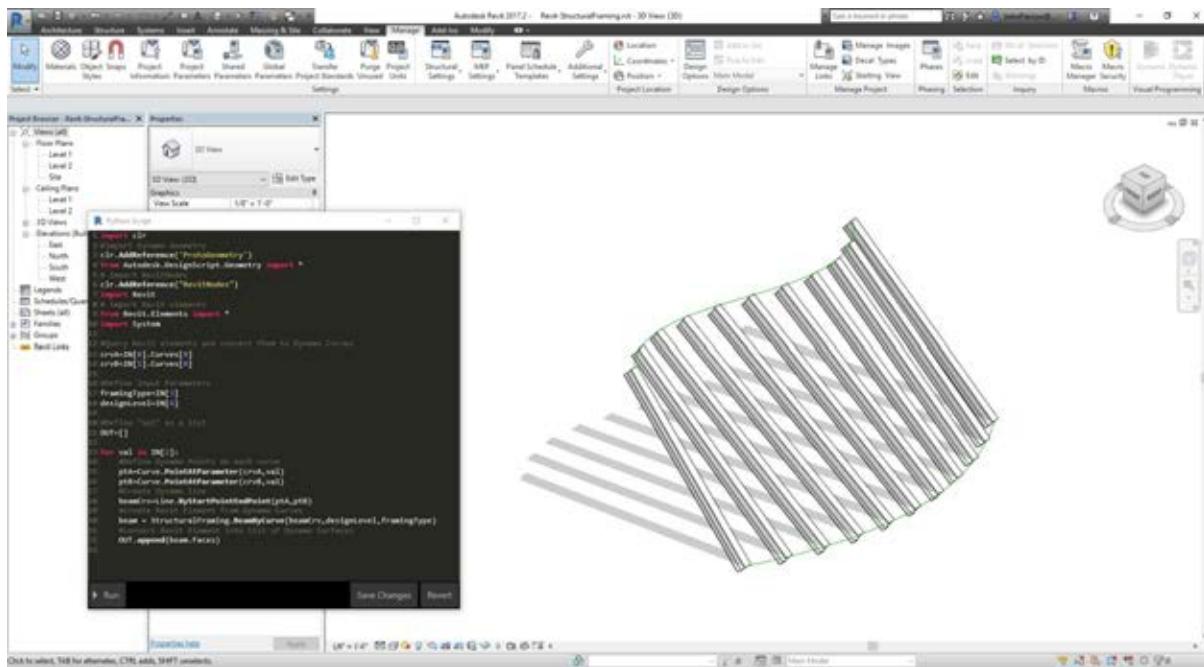
#Query Revit elements and convert them to Dynamo Curves
crvA=IN[0].Curves[0]
crvB=IN[1].Curves[0]

#Define input Parameters
framingType=IN[3]
designLevel=IN[4]

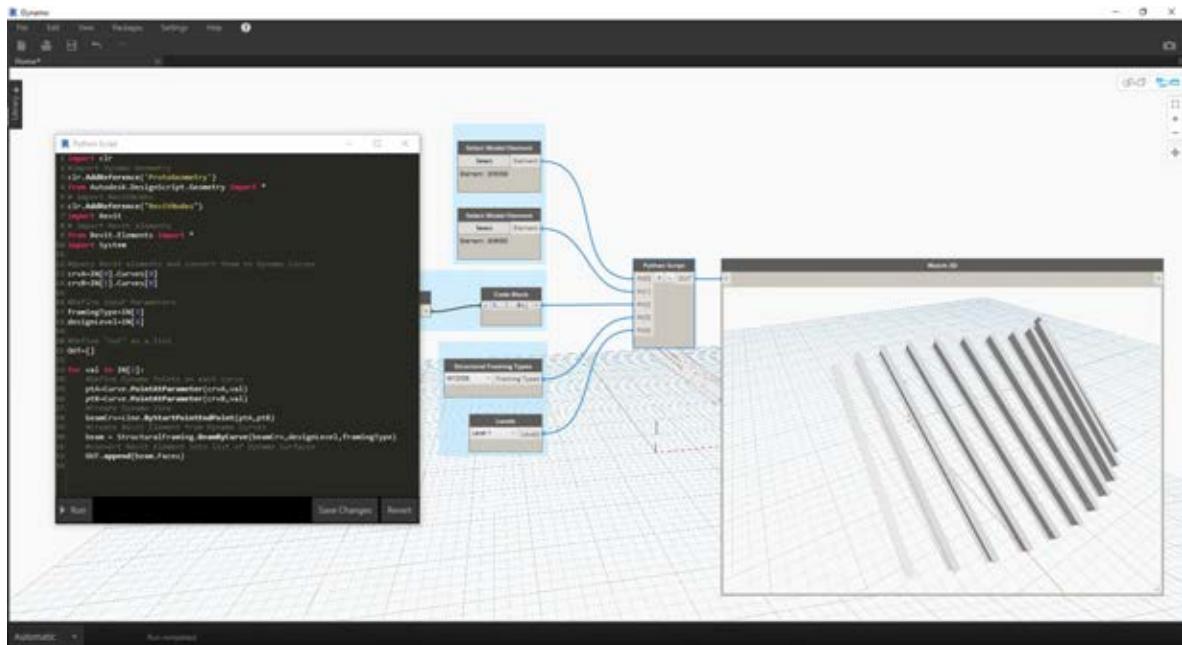
#Define "out" as a list
OUT=[]

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)
```



Revit で、2 つの曲線にわたる梁の配列が構造要素として作成されました。この構造要素は、Dynamo でネイティブの Revit インスタンスを作成する場合の例として使用しているもので、実際にはあり得ない構造要素であることに注意してください。

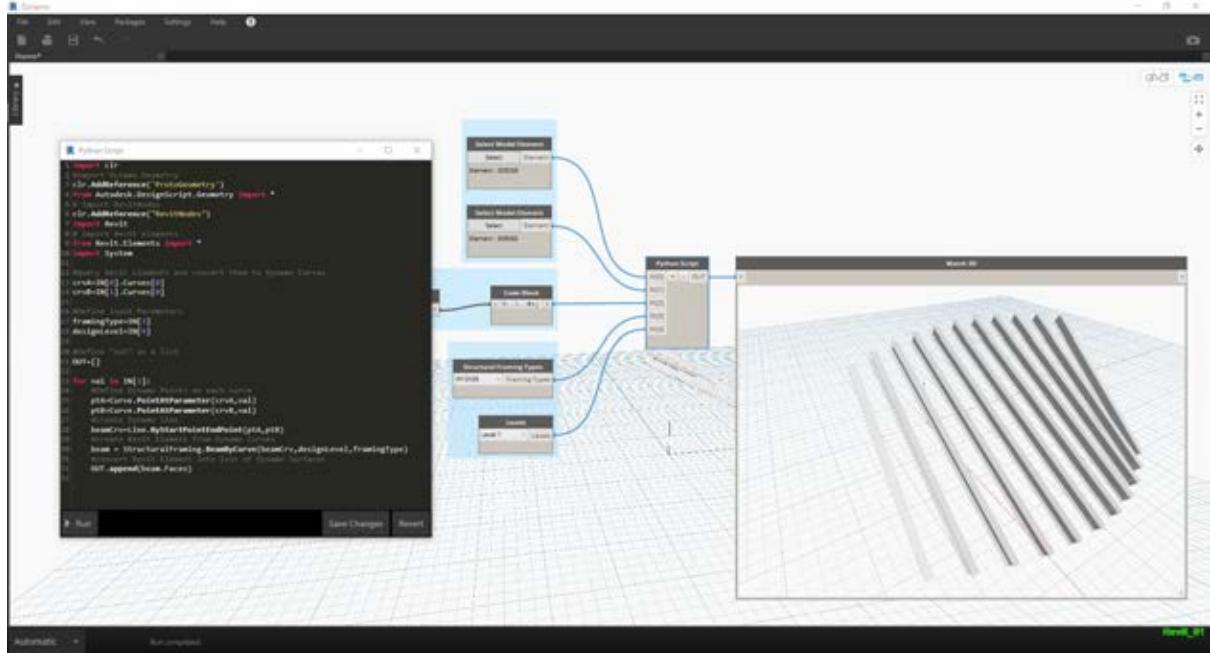


Dynamo でも、結果を確認することができます。Watch 3D ノードの梁は、Revit 要素からクエリーされたジオメトリを参照します。

このセクションでは、Revit 環境から Dynamo 環境にデータを変換する一連のプロセスを作成しました。このプロセスをまとめると、次のようにになります。

1. Revit の要素を選択する
2. Revit の要素を Dynamo の曲線に変換する
3. Dynamo の曲線を一連の Dynamo の点に分割する
4. Dynamo の 2 つの曲線間の点を使用して Dynamo の線分を作成する
5. Dynamo の線分を参照することにより、Revit の梁を作成する
6. Revit の梁のジオメトリに対してクエリーを実行することにより、Dynamo のサーフェスを出力する

これは複雑なプロセスに見えるかもしれません、スクリプトを使用すると、Revit で曲線を編集してソルバを再実行するのと変わらないほど単純な処理になります(ただし、そのためには、元の梁を削除する必要があります)。これは、Python で梁を配置することによって、初期設定のままのノードにある関連付けが解除されるためです。



Revit で参照曲線を更新すると、梁の新しい配列が作成されます。

# Python テンプレート

## Python テンプレート

Dynamo 2.0 では、初めて Python ウィンドウを開く際に、既定で使用するテンプレート (.py 拡張子)を指定することができます。この機能があると、Dynamo 内で Python を効率よく使用できるため以前からご要望をいただきました。テンプレートを使用できる機能があるため、カスタム Python スクリプトを開発する際に既定のインポートをすぐに利用できます。

このテンプレートは、Dynamo をインストールした APPDATA にあります。

これは通常、(%appdata%/Dynamo/Core/{バージョン}/)です。

Share View				
C:\Users\ USERNAME \AppData\Roaming\Dynamo\Dynamo Core\2.0				
	Name	Date modified	Type	Size
ss	definitions	4/17/2018 7:48 AM	File folder	
d	Logs	5/1/2018 9:32 AM	File folder	
d	packages	4/17/2018 9:23 AM	File folder	
r	DynamoSettings.xml	5/1/2018 9:32 AM	XML Document	2 KB
r	PythonTemplate.py	5/8/2018 9:15 AM	Python File	1 KB

### テンプレートを設定する

この機能を使用するには、DynamoSettings.xml ファイルで次の行を追加する必要があります(メモ帳で編集します)。

```
29 <CustomPackageFolders>
30   <string>C:\Users\ USERNAME \AppData\Roaming\Dynamo\Dynamo Core\2.0</string>
31 </CustomPackageFolders>
32 <PackageDirectoriesToUninstall />
33 <PythonTemplateFilePath />
34 <BackupInterval>60000</BackupInterval>
35 <BackupFilesCount>1</BackupFilesCount>
36 <PackageDownloadTouAccepted>false</PackageDownloadTouAccepted>
```

<PythonTemplateFilePath />を見つけて、これを次のように置き換えるだけです。

```
<PythonTemplateFilePath>
C:\Users\CURRENTUSER\AppData\Roaming\Dynamo\Dynamo Core\2.0\PythonTemplate.py
</PythonTemplateFilePath>
```

注: CURRENTUSER を自分のユーザ名に置き換えます

次に、使用する機能を組み込んだテンプレートを作成する必要があります。ここでは、Revit に関連するインポートおよび Revit で作業する際の他の一般的な項目の一部を組み込みます。

空のメモ帳を起動して、次のコードを貼り付けることができます。

```
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)

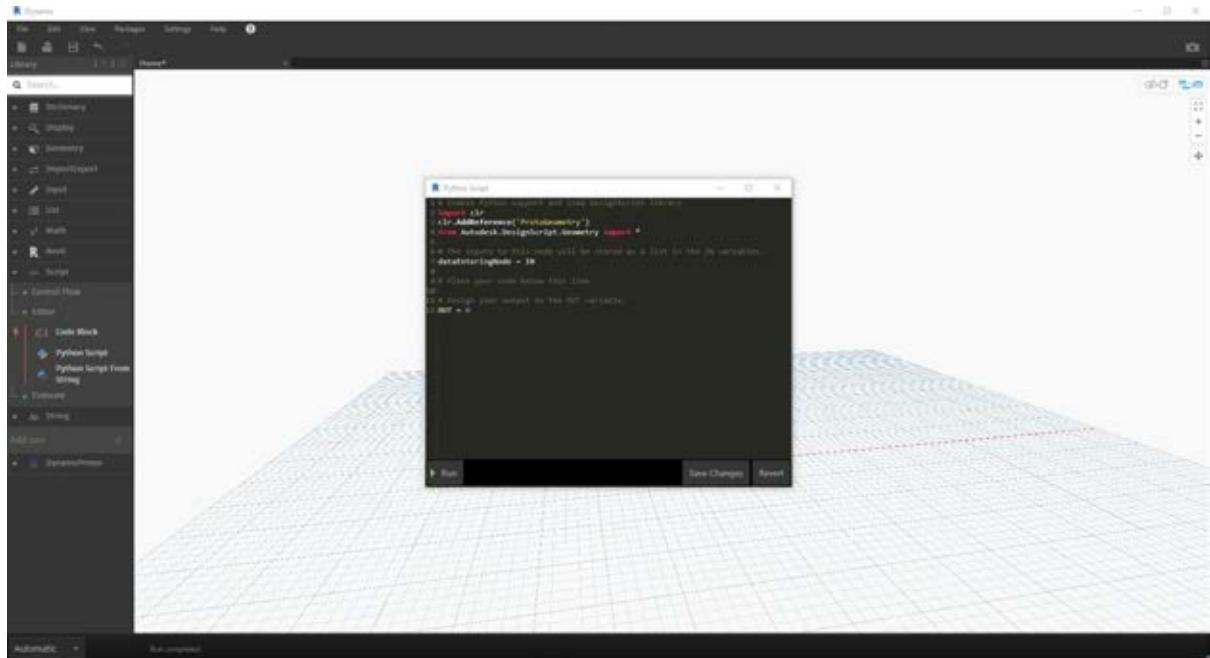
TransactionManager.Instance.TransactionTaskDone()

OUT = element
```

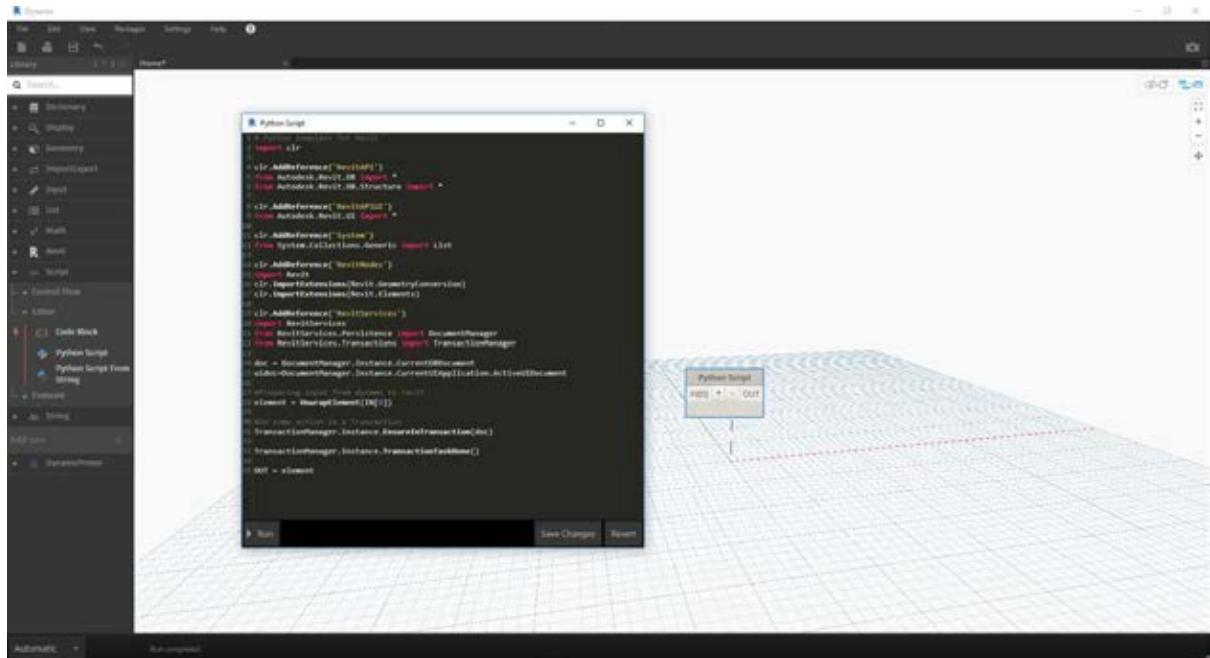
貼り付けが完了したら、このファイルを PythonTemplate.py という名前で APPDATA フォルダ内に保存します。

### Python スクリプトのその後の動作

Python テンプレートが定義されると、Python Script ノードが配置されるたびに、Dynamo はこのテンプレートを検索します。見つからない場合、既定の Python ウィンドウのように表示されます。



Python テンプレート(たとえばここで作成した Revit でのテンプレート)が見つかった場合、組み込んだ既定の項目がすべて表示されます。

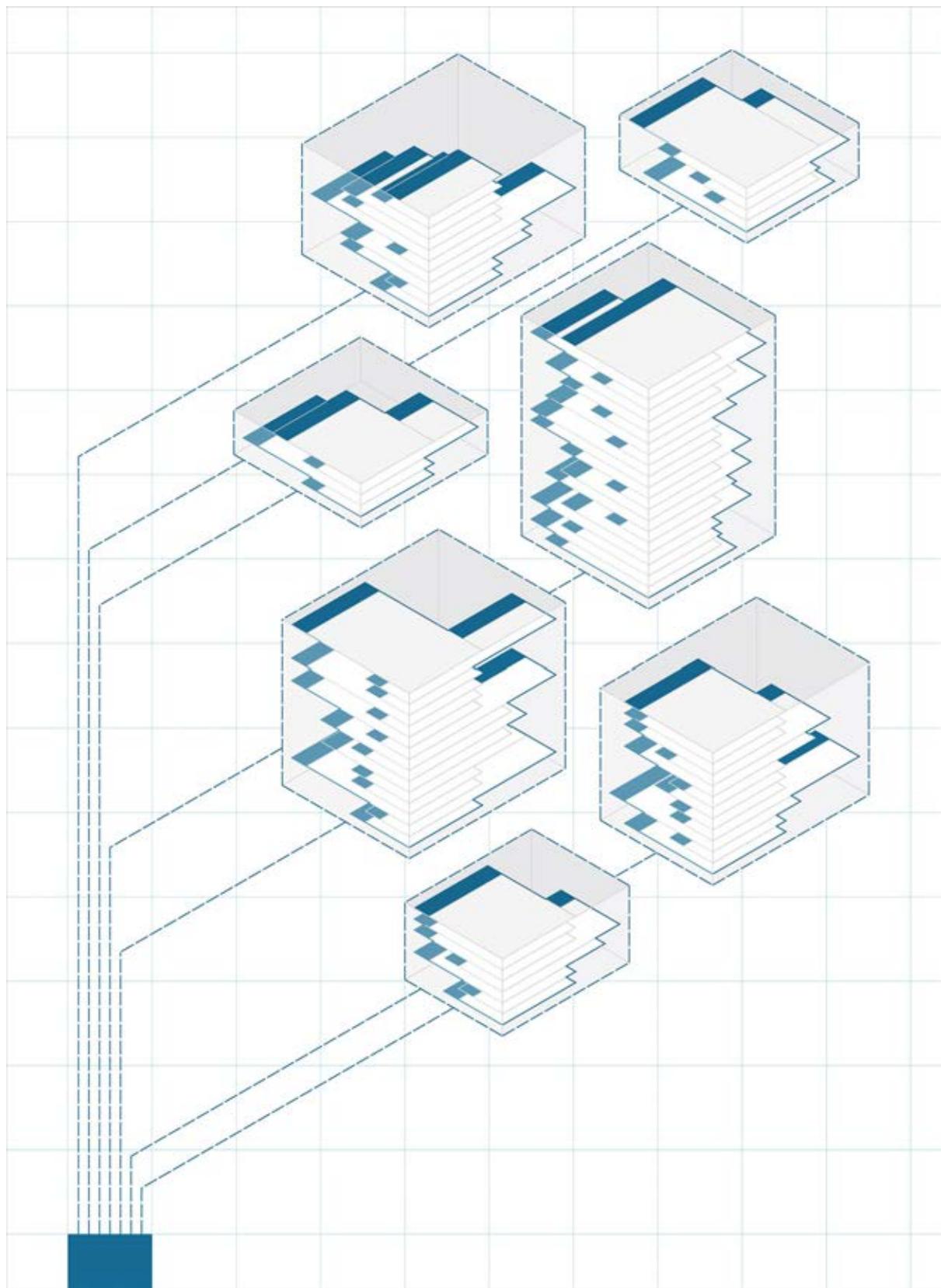


この優れた追加機能(Radu Gidei 氏による)に関する詳細は、<https://github.com/DynamoDS/Dynamo/pull/8122> を参照してください。

## パッケージ

### パッケージ

前の章では、いくつかのカスタム ノードを作成しました。ここでは、パッケージを使用してカスタム ノードを整理し、パブリッシュしてみましょう。パッケージは、独自のノードを保存し、Dynamo コミュニティと共有できる便利な方法です。



# パッケージ

## パッケージ

簡単に説明すると、パッケージとはカスタム ノードの集合のことです。Dynamo Package Manager は、オンラインでパブリッシュされたパッケージをダウンロードするためのコミュニティ ポータルです。これらのツールセットは、Dynamo の基本機能を拡張するためにサードパーティによって開発されたもので、すべてのユーザがアクセスでき、ボタンをクリックするだけでダウンロードすることができます。

The screenshot shows the homepage of the Dynamo Package Manager. At the top, it displays '106248' packages, '614' authors, and '143' meta-tags. Below this, there are two main sections: 'Packages' and 'Authors'. The 'Packages' section lists the most recent packages, including 'LOD OPERATOR(dyn)', 'Dynamite', 'Dynamite for Revit', 'Mechanica', 'Quench for Dynamo', 'Revit User', 'Unidimensional', and 'Sweat equity'. The 'Authors' section lists the most voted-for authors, such as 'dynamite', 'archi-lab', 'mantis shrimp', 'jason', 'mantis shrimp', 'mantis shrimp', 'mantis shrimp', 'mantis shrimp', and 'jason'. It also shows the most recently active authors and the most prolific authors.

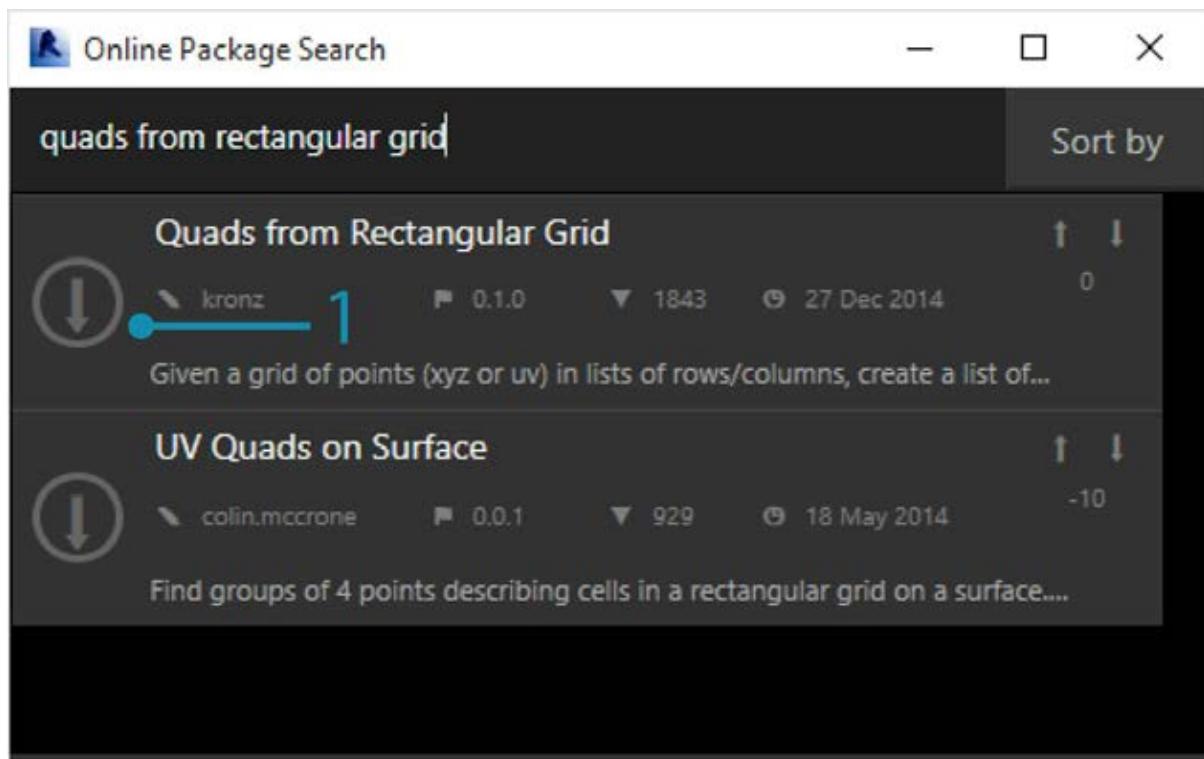
Dynamoのようなオープンソース プロジェクトは、このようなコミュニティによって成長しています。サードパーティの専任開発者の取り組みにより、Dynamo はさまざまな業界のワークフローで採用されています。そのため、Dynamo チームは、パッケージの開発とパブリッシュの合理化を連携して進めています。これについては、以降のセクションで詳しく説明します。

### パッケージをインストールする

パッケージを最も簡単にインストールする方法は、Dynamo インタフェースの[パッケージ]ツールバーを使用する方法です。では、実際にインストールしてみましょう。ここでは、次に示す簡単な例を使用して、グリッド上に四角いパネルを作成するためのパッケージをインストールします。

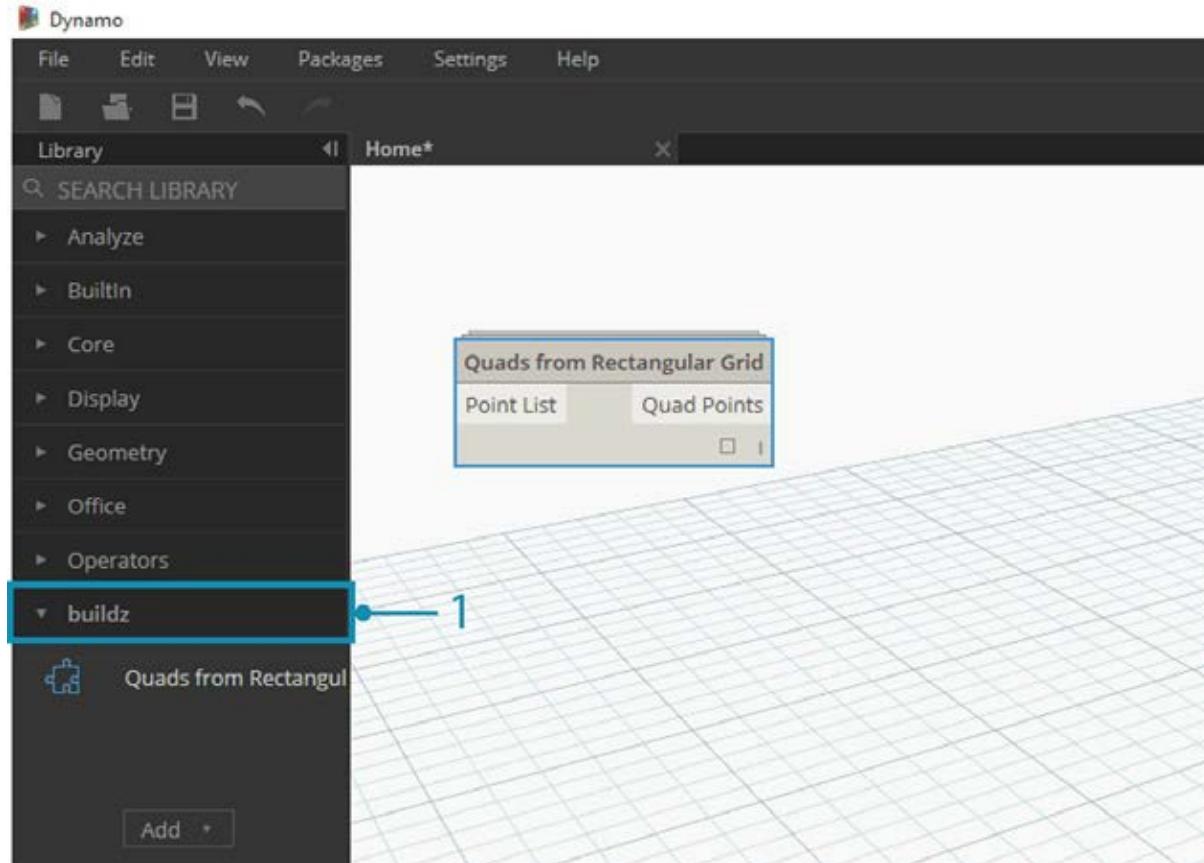
The screenshot shows the Dynamo interface with the 'Packages' tab selected in the toolbar. A tooltip labeled '1' points to the search bar, which contains the text 'Search for a Package...'. Below the search bar, there are several options: 'Manage Packages...', 'Publish New Package...', 'Publish Selected Nodes...', and 'Publish Current Workspace...'. On the left, there is a library sidebar with categories like 'Analyze', 'Archi-lab\_MantisShrimp', 'DynamoPrimer', 'buildz', 'Builtin', 'CAAD\_RWTH', 'Core', and 'Display'. The main workspace area shows a grid with a single node placed on it, and a status bar at the bottom right shows 'PointsToSurface.dyn'.

1. Dynamo で、[パッケージ] > [パッケージの検索]に移動します。

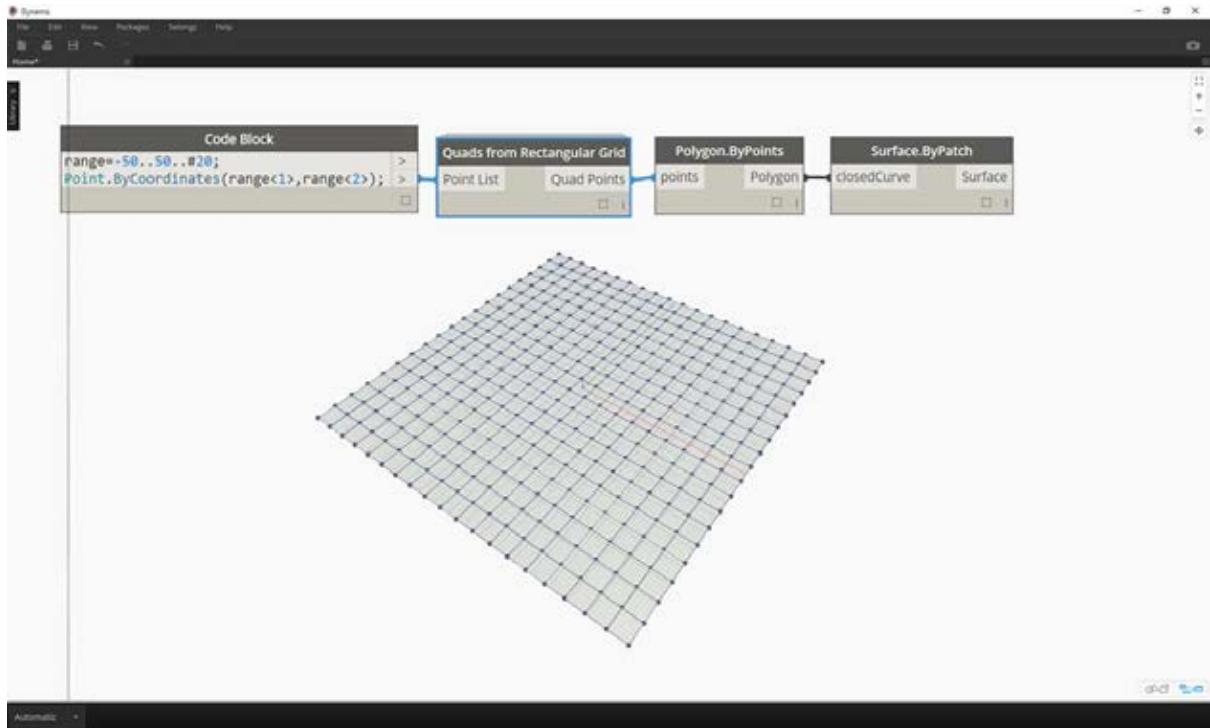


[検索]バーで、「quads from rectangular grid」を検索してみましょう。しばらくすると、この検索クエリーに一致するパッケージがすべて表示されます。一致する名前を持つ最初のパッケージを選択します。

1. パッケージ名の左に表示されているダウンロード矢印アイコンをクリックします。



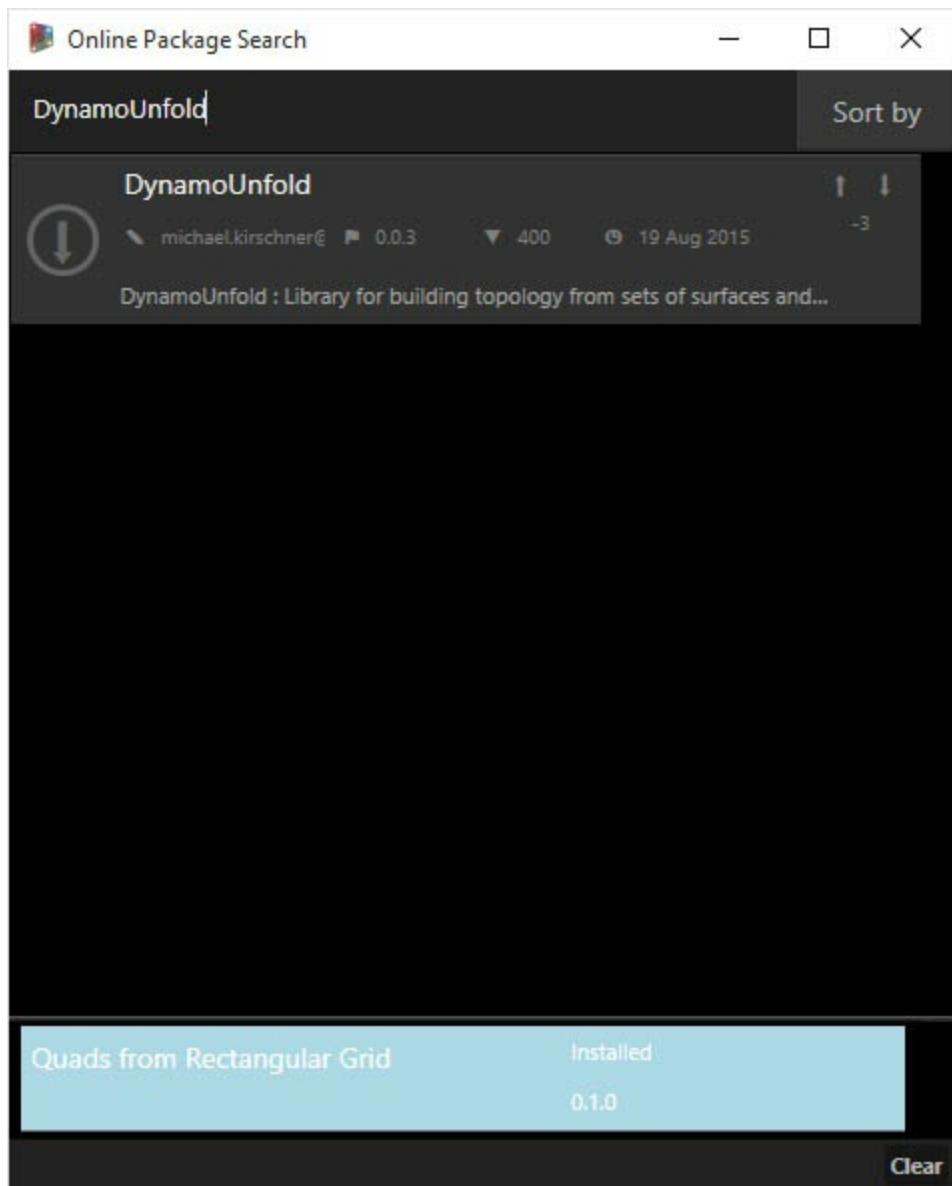
1. Dynamo ライブラリに「buildz」という名前の新しいグループが表示されます。この名前は、パッケージの[開発者](#)を参照して付けられます。また、カスタム ノードはこのグループ内に配置されます。このグループは、すぐに使用することができます。



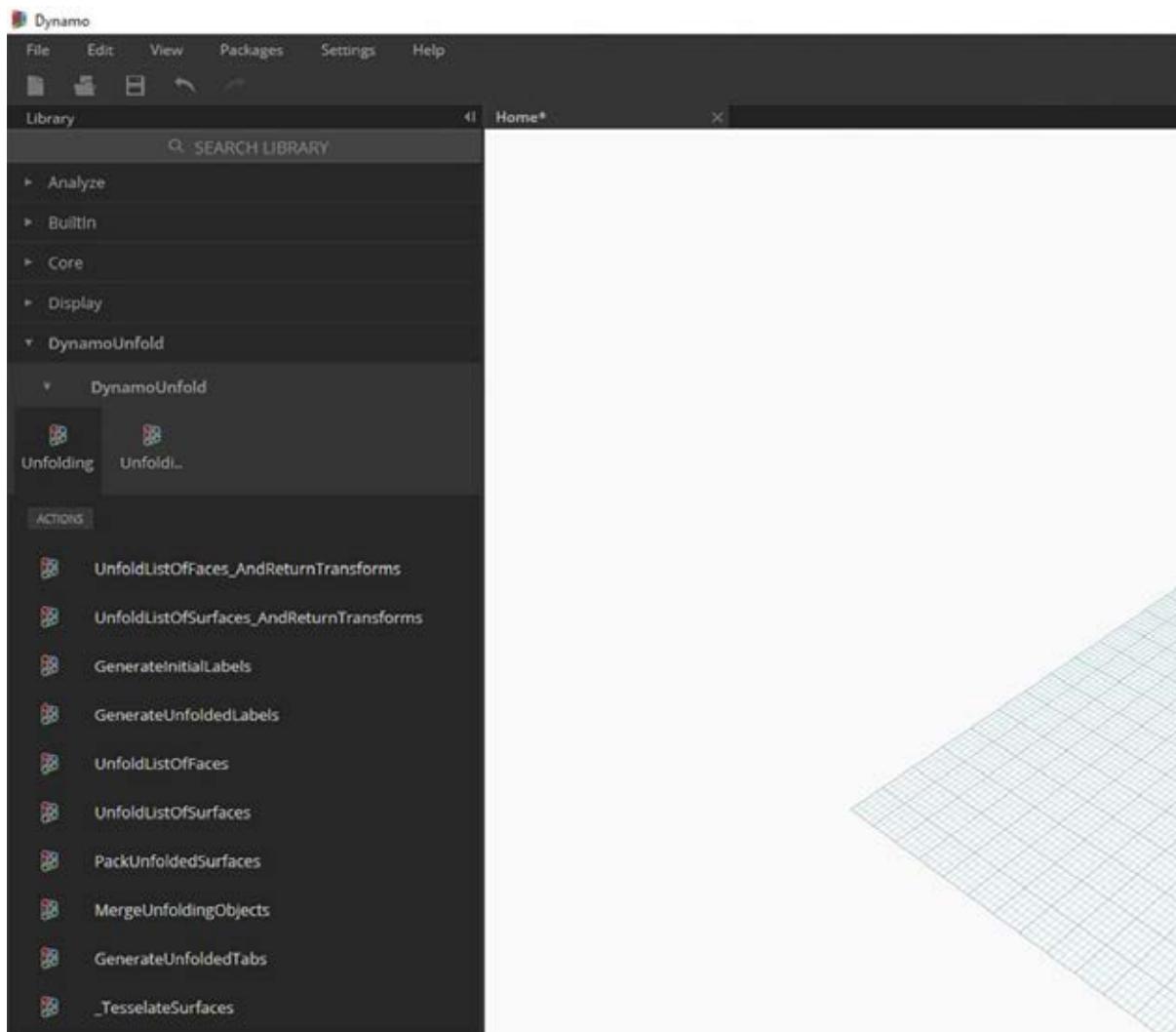
矩形グリッドを定義するためのコード ブロック操作を実行することにより、矩形パネルのリストをすばやく作成できます。

#### パッケージ フォルダ

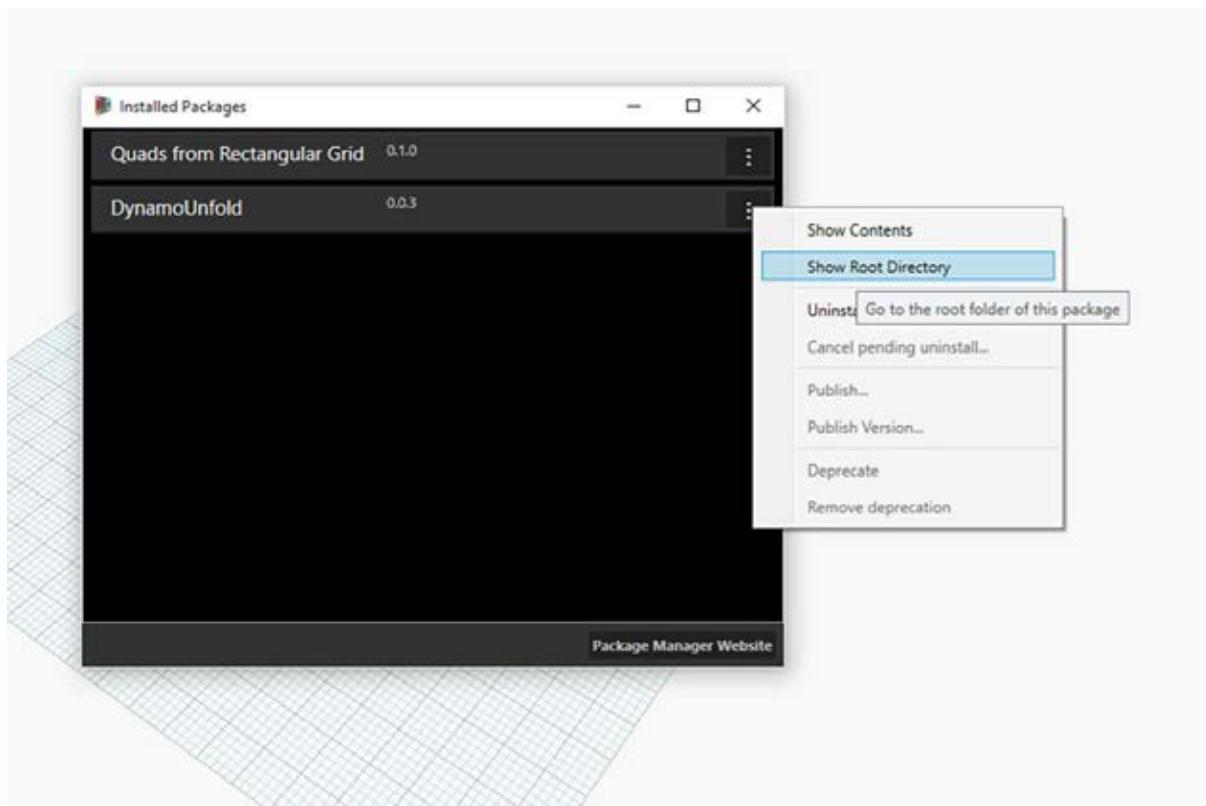
上記の例では、1 つのカスタム ノードが含まれているパッケージを使用しましたが、複数のカスタム ノードやサポート データ ファイルが含まれているパッケージをダウンロードする場合も、同じプロセスを実行します。ここでは、より包括的な Dynamo Unfold パッケージを使用して手順を説明します。



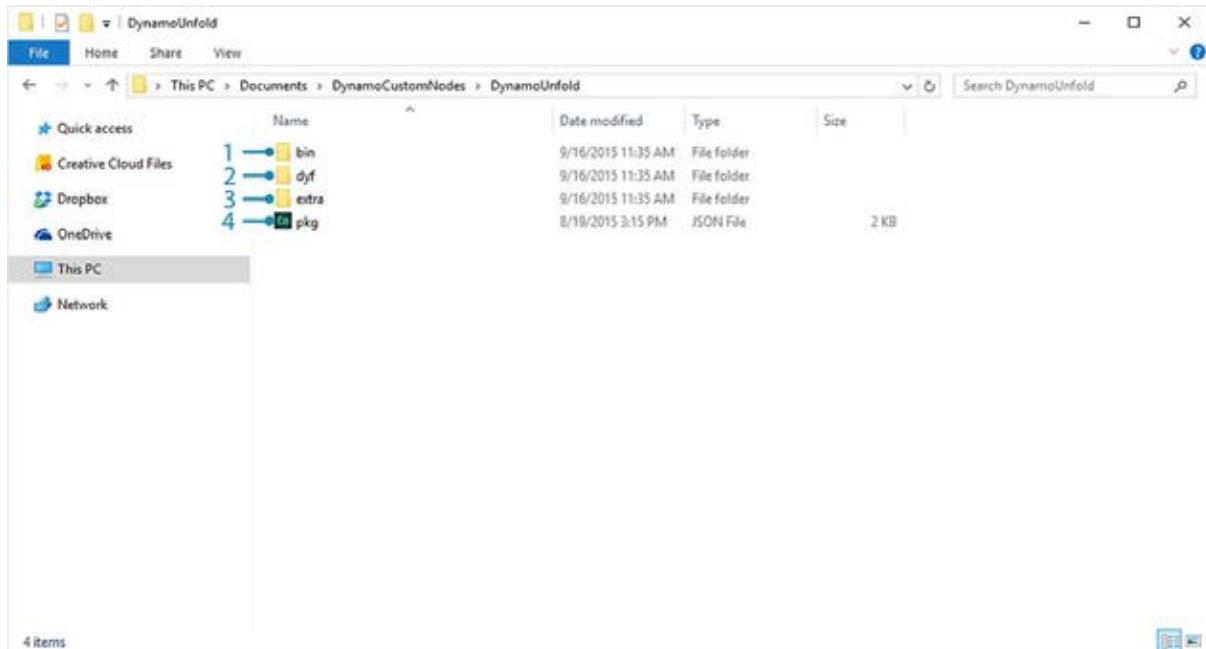
上記の例と同様に、[パッケージ] > [パッケージの検索]を選択します。ここでは、「*DynamoUnfold*」という1つの単語を、大文字と小文字を区別して検索します。パッケージが表示されたら、パッケージ名の左に表示されている矢印をクリックしてパッケージをダウンロードします。Dynamo Unfold が Dynamo ライブラリにインストールされます。



Dynamo ライブラリに *DynamoUnfold* グループが表示されます。このグループに、複数のカテゴリとカスタム ノードが含まれているのがわかります。



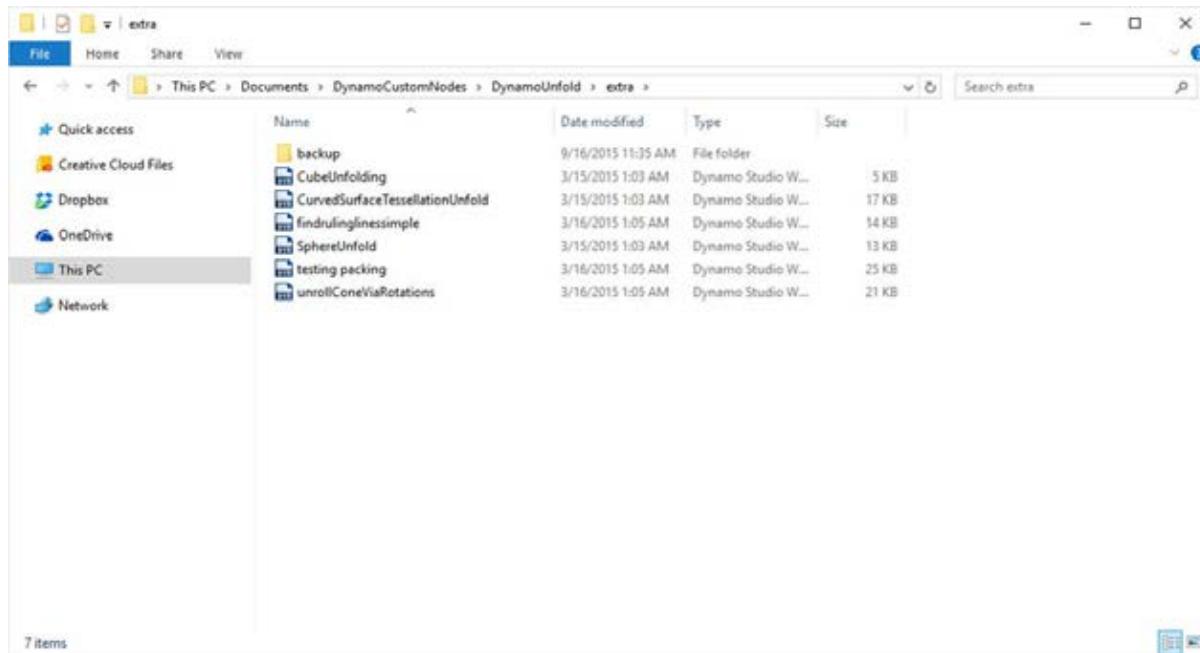
ここで、パッケージのファイル構造を確認しましょう。Dynamo で[パッケージ] > [パッケージを管理]を選択します。上記のウィンドウに、インストールされている 2 つのライブラリが表示されます。DynamoUnfold の右に表示されているボタンをクリックし、[ルート フォルダを表示]を選択します。



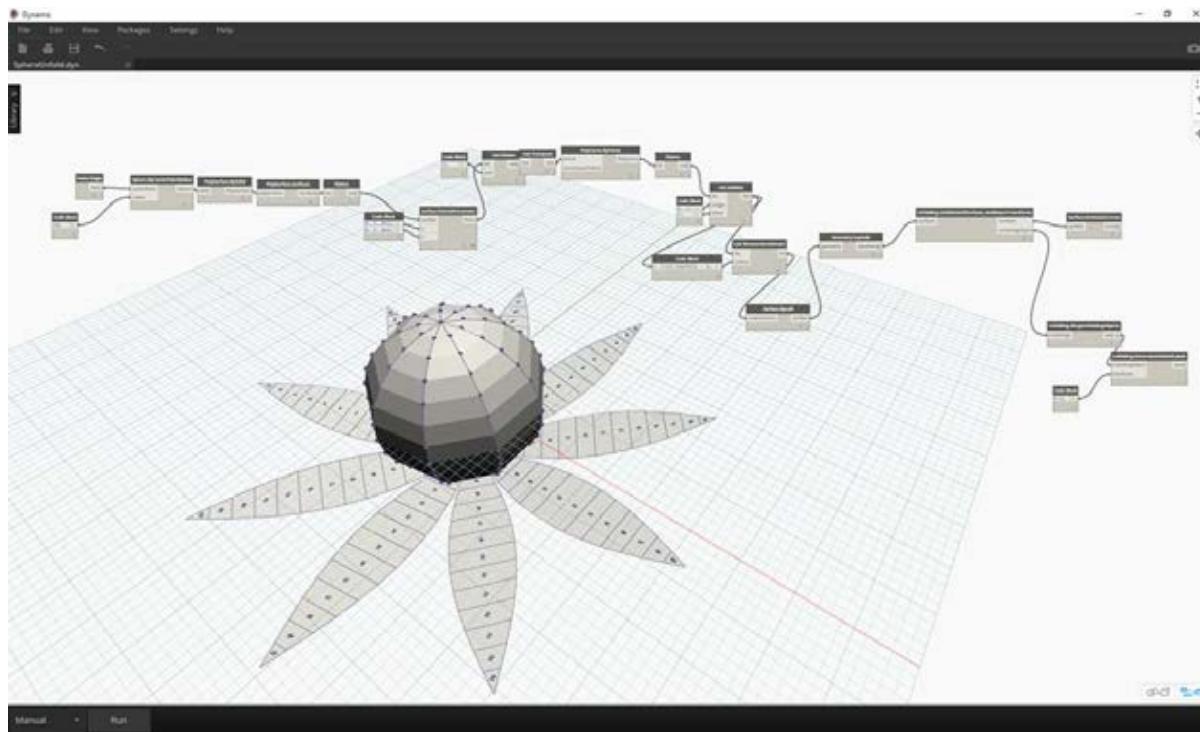
この操作により、パッケージのルート フォルダが表示されます。このルート フォルダには、3 つのフォルダと 1 つのファイルが格納されています。

1. *bin* フォルダには .dll ファイルが格納されます。この Dynamo パッケージは Zero-Touch を使用して開発されているため、カスタム ノードはこのフォルダに格納されます。
2. *dyf* フォルダにはカスタム ノードが格納されます。このパッケージは Dynamo カスタム ノードを使用して開発されたものではないため、このフォルダには格納されません。
3. *extra* フォルダには、サンプル ファイルを含め、すべての追加ファイルが格納されます。
4. *pkg* ファイルは、パッケージの設定を定義する基本のテキスト ファイルです。ここでは、このファイルは無視してかまいませ

h。



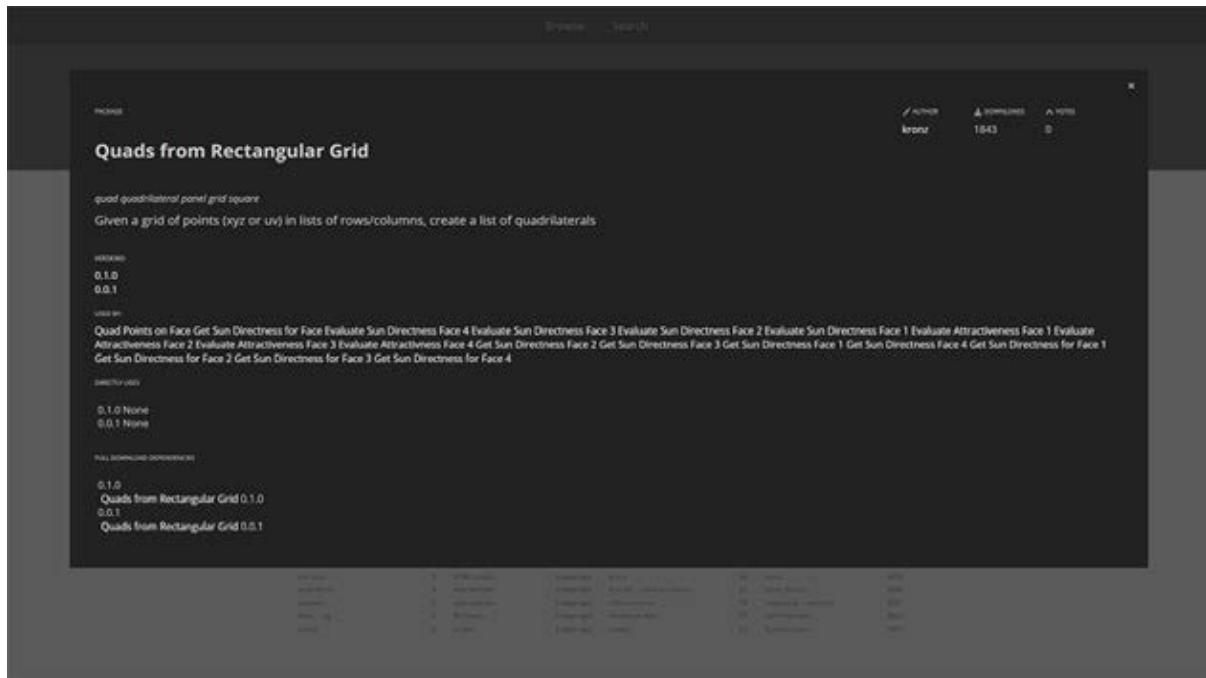
*extra* フォルダを開くと、インストール時にダウンロードされた多数のサンプル ファイルが格納されていることがわかります。すべての パッケージにサンプル ファイルが付属しているわけではありませんが、付属のサンプル ファイルはこのフォルダに格納されています。ここで、*SphereUnfold* ファイルを開いてみましょう。



ファイルを開いてからソルバで[実行]をクリックすると、展開された球形が表示されます。これらのサンプル ファイルは、新しい Dynamo パッケージの使用方法を理解するのに役立ちます。

## Dynamo Package Manager

Dynamo パッケージの仕組みを理解する別の方法として、[Dynamo Package Manager](#) をオンラインで参照する方法もあります。これは、パッケージを参照するのに便利な方法です。リポジトリにより、ダウンロード回数と人気度に応じて、パッケージが並べ替えられます。また、パッケージの最新の更新プログラムに関する情報を簡単に収集することもできます。一部の Dynamo パッケージは、Dynamo ビルドのバージョン管理と依存関係の影響を受けます。



Dynamo Package Manager で[*Quads from Rectangular Grid*]をクリックすると、説明、バージョン、開発者、依存関係を確認することができます。

また、Dynamo Package Manager からパッケージ ファイルをダウンロードすることもできますが、Dynamo から直接ダウンロードした方が簡単です。

### ローカルでのファイルの保存場所

Dynamo Package Manager からファイルをダウンロードした場合や、すべてのパッケージ ファイルの保存場所を参照する場合は、[設定] > [ノードとパッケージのパスを管理]をクリックします。フォルダ ディレクトリの横にある省略記号をクリックすると、ルート フォルダをコピーして、エクスプローラ ウィンドウでパッケージを詳細に調べることができます。既定では、パッケージは C:\Users\[ユーザ名]\AppData\Roaming\Dynamic\[Dynamo バージョン] というフォルダ パスにインストールされます。

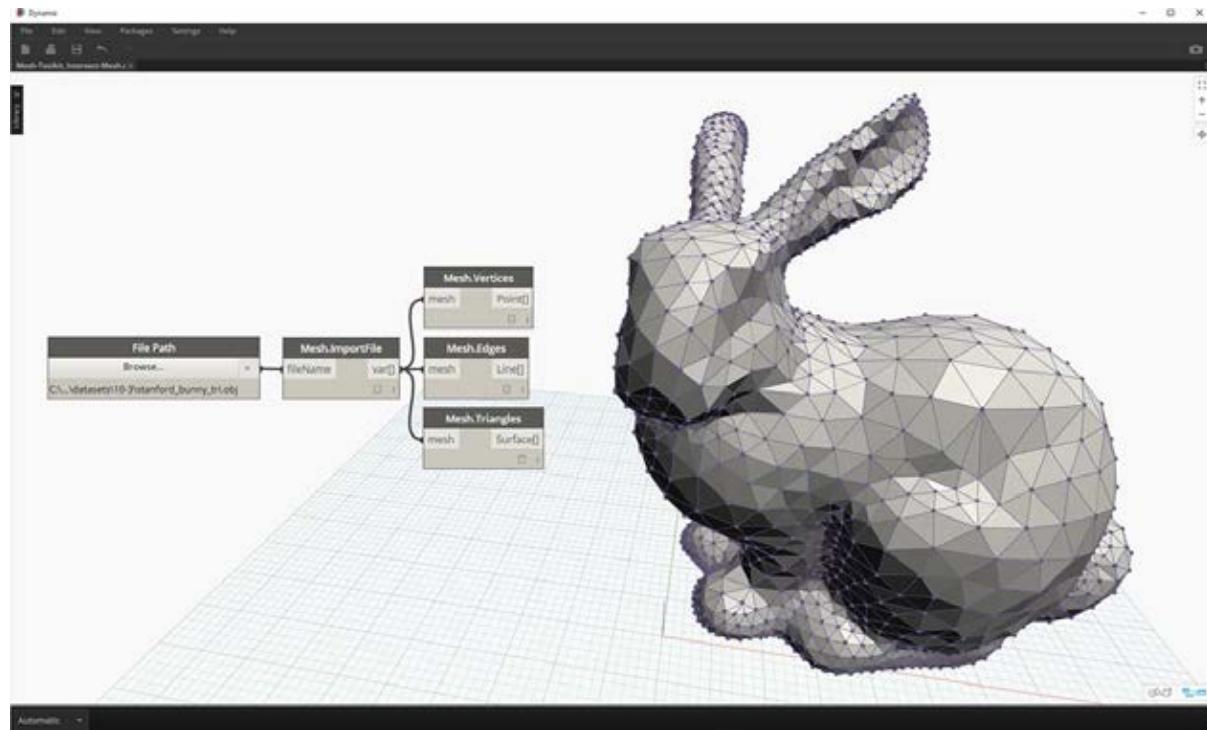
### パッケージの拡張

Dynamo コミュニティは、常に成長と進化を続けています。Dynamo Package Manager を定期的に確認して、便利な新機能を発見してください。これ以降のセクションでは、エンドユーザーの視点から見た独自の Dynamo パッケージの作成など、パッケージについてより詳しく確認していきます。

# パッケージのケーススタディ - Mesh Toolkit

## パッケージのケーススタディ - Mesh Toolkit

Dynamo Mesh Toolkit は、外部ファイル形式からメッシュを読み込む機能、Dynamo のジオメトリ オブジェクトからメッシュを作成する機能、頂点とインデックスからメッシュを手動で作成する機能を提供するライブラリです。このライブラリには、メッシュの変更や修復を行うためのツールや、製造処理で使用する水平方向のスライスを抽出するためのツールも用意されています。

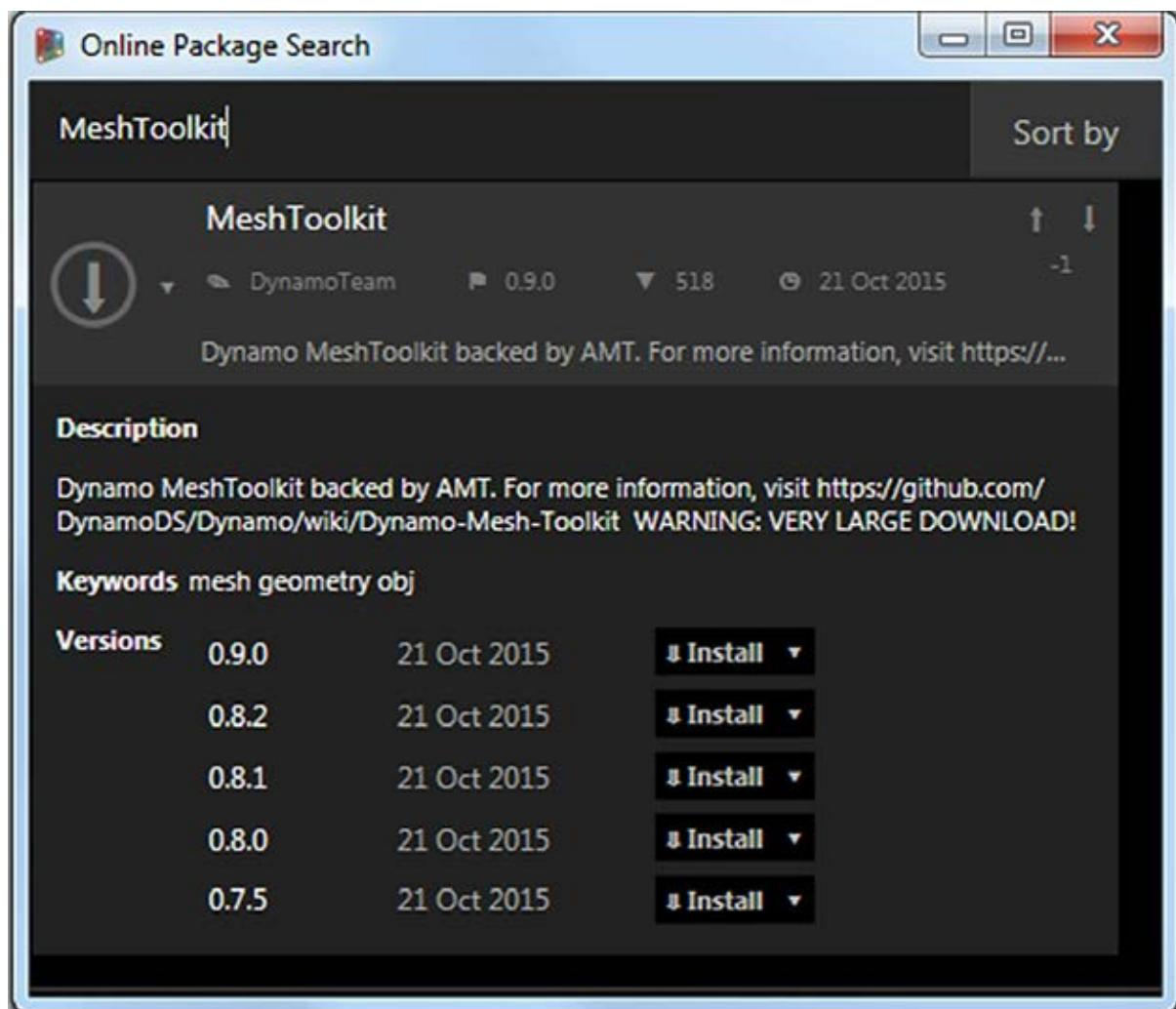


Dynamo Mesh Toolkit は、オートデスクによるメッシュ研究の一環として開発されているため、今後も成長を続けていきます。このツールキットには、頻繁に新しいメソッドが追加される予定になっています。コメント、バグの報告、新機能の提案については、お気軽に Dynamo チームまでご連絡ください。

### メッシュとソリッドとの比較

次の演習では、Mesh Toolkit を使用して、いくつかの基本的なメッシュ操作を説明します。この演習では、メッシュを一連の平面と交差させます。この操作でメッシュではなくソリッドを使用すると、計算量が多くなります。ソリッドとは異なり、メッシュには「解像度」の集合があります。現在の作業に応じて、この解像度を定義することができます。メッシュは、数学的ではなく位相幾何学的に定義されています。メッシュとソリッドとの関係について詳しくは、この手引の「[計算設計用のジオメトリ](#)」の章を参照してください。Mesh Toolkit の詳細な説明については、[Dynamo Wiki ページ](#)を参照してください。次の演習で、パッケージの内容を確認してみましょう。

### Mesh Toolkit のインストール

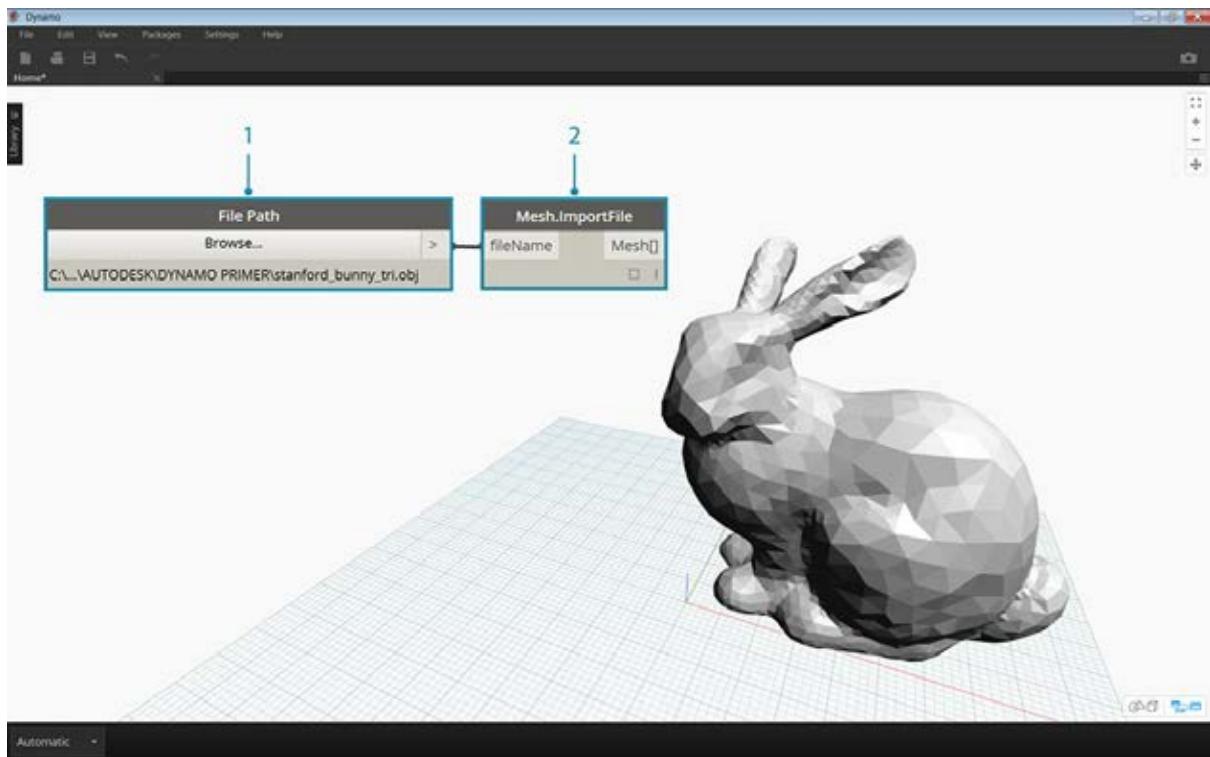


Dynamo の上部メニューバーで[パッケージ] > [パッケージの検索...]に移動します。[検索]フィールドに「MeshToolkit」と入力します。スペースを入れずに 1 つの単語として入力し、大文字と小文字も正確に入力してください。Dynamo のバージョンに対応するパッケージを選択して、ダウンロード矢印をクリックします。このように、操作は非常に簡単です。

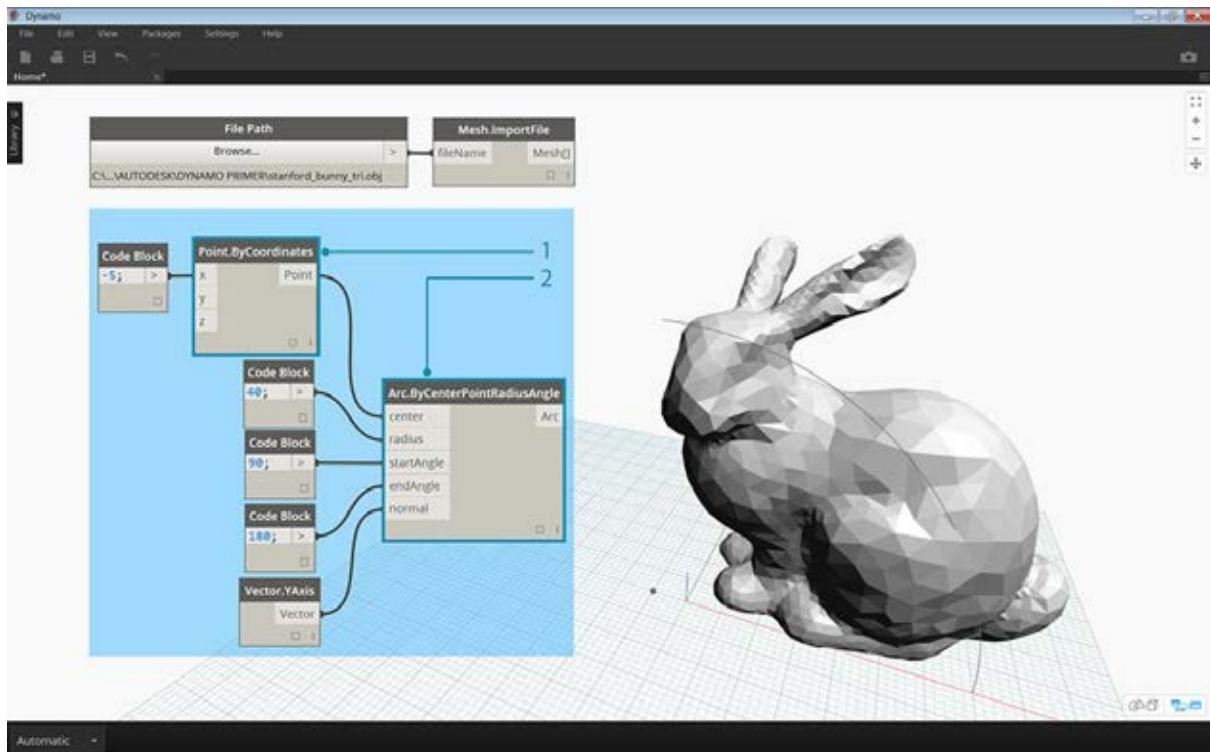
## 演習

この演習用のサンプルファイルをダウンロードして解凍してください(右クリックして[名前を付けてリンク先を保存])。すべてのサンプルファイルの一覧については、付録を参照してください。[MeshToolkit.zip](#)

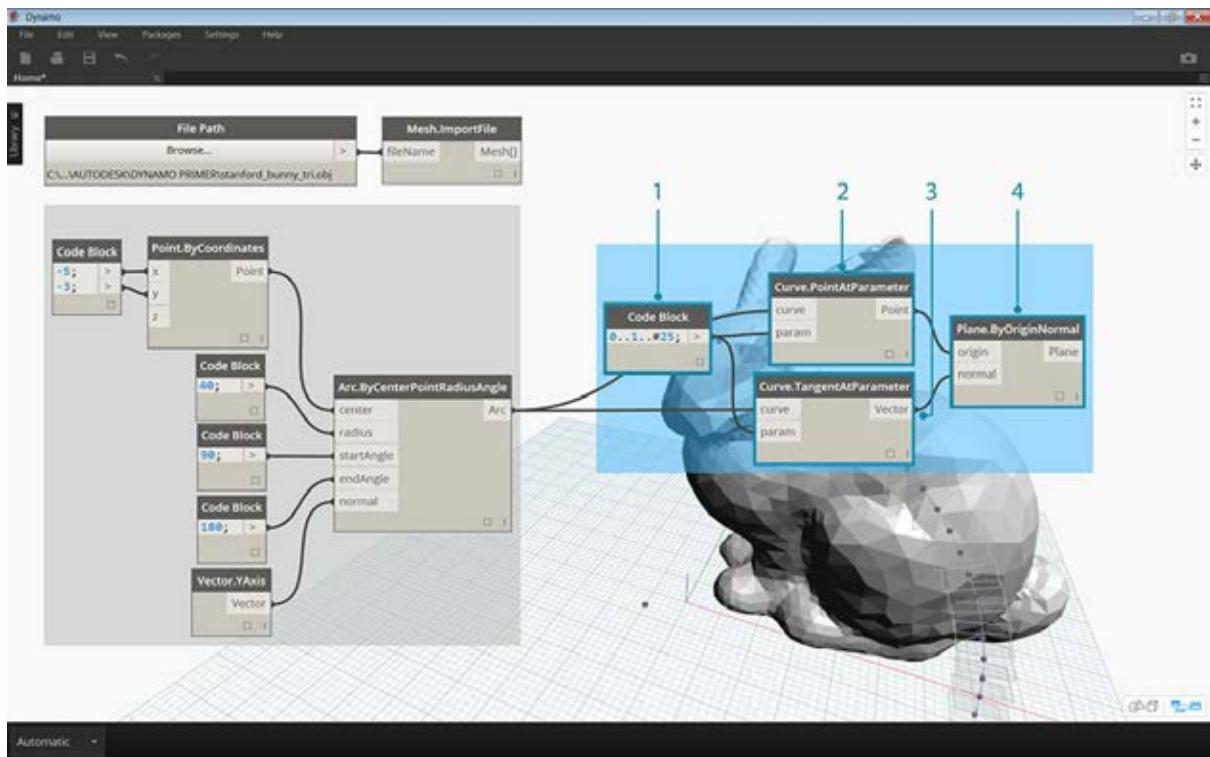
最初に、Dynamo で *Mesh-Toolkit\_Intersect-Mesh.dyn* を開きます。この例では、Mesh Toolkit の Intersection ノードについて説明します。メッシュを読み込み、そのメッシュを一連の入力面と交差させてスライスを作成します。レーザー カッター、ウォータージェット カッター、CNC ミルなどの製造用モデルを作成する場合は、最初にこの操作を実行することになります。



1. File Path ノードで、読み込むメッシュファイル(stanford\_bunny\_tri.obj)の場所を指定します。サポートされるファイルタイプは、.mix と .obj です。
2. Mesh.ImportFile ノードに File Path ノードを接続して、メッシュを読み込みます。

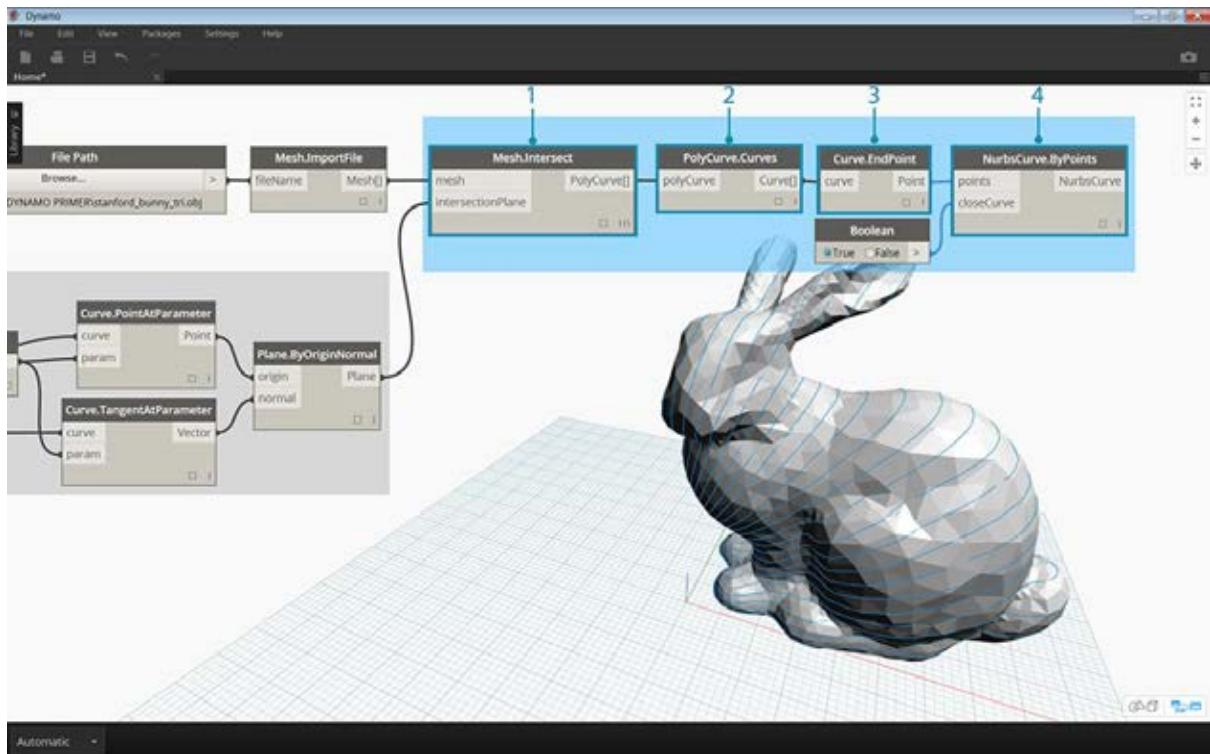


1. Point.ByCoordinates ノードで、点を作成します。この点が、円弧の中心になります。
2. Arc.ByCenterPointRadiusAngle ノードで、上記の点を中心とする円弧を作成します。この曲線を使用して、一連の面が配置されます。

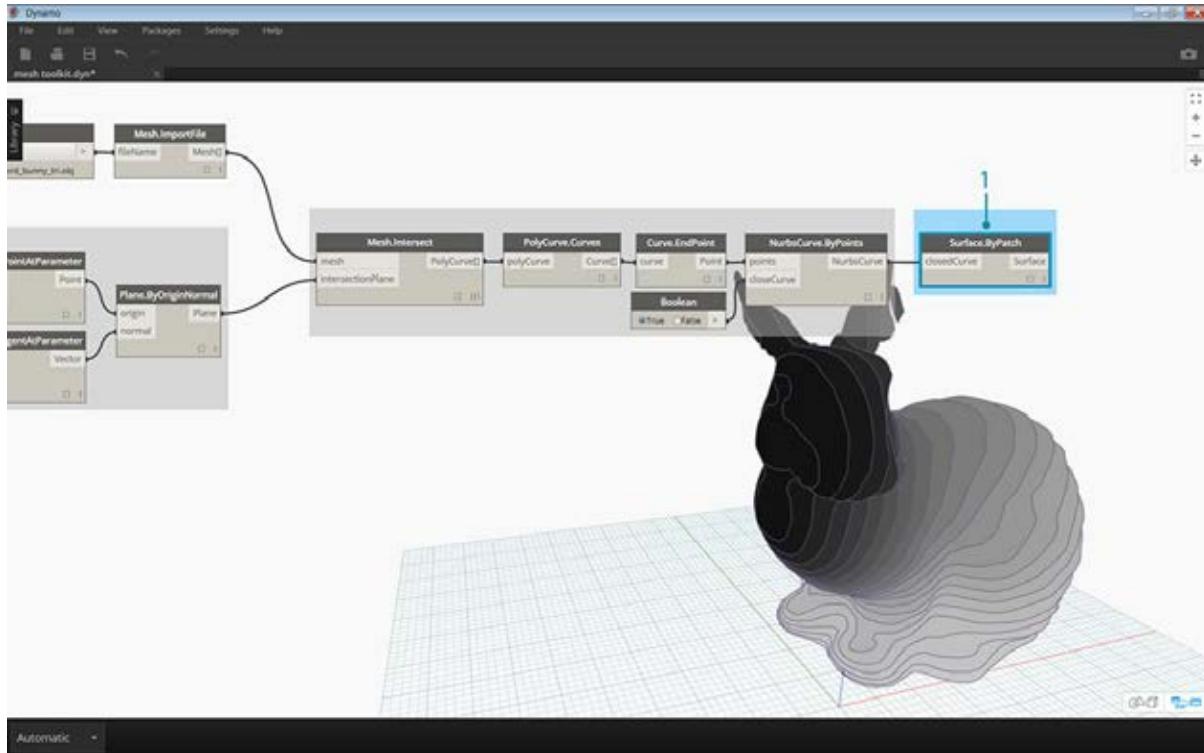


1. Code Block ノードで、0 から 1 までの数値の範囲を作成します。
2. Curve.PointAtParameter ノードの curve 入力に Arc.ByCenterPointRadiusAngle ノードの Arc 出力を接続し、param 入力に Code Block ノードの出力を接続します。これにより、曲線に沿って一連の点が抽出されます。
3. Curve.TangentAtParameter ノードの curve 入力に、Arc.ByCenterPointRadiusAngle ノードの Arc 出力を接続します。
4. Plane.ByOriginNormal ノードの origin 入力に Curve.PointAtParameter ノードの Point 出力を接続し、normal 入力に Curve.TangentAtParameter ノードの Vector 出力を接続します。これにより、各点に一連の平面が作成されます。

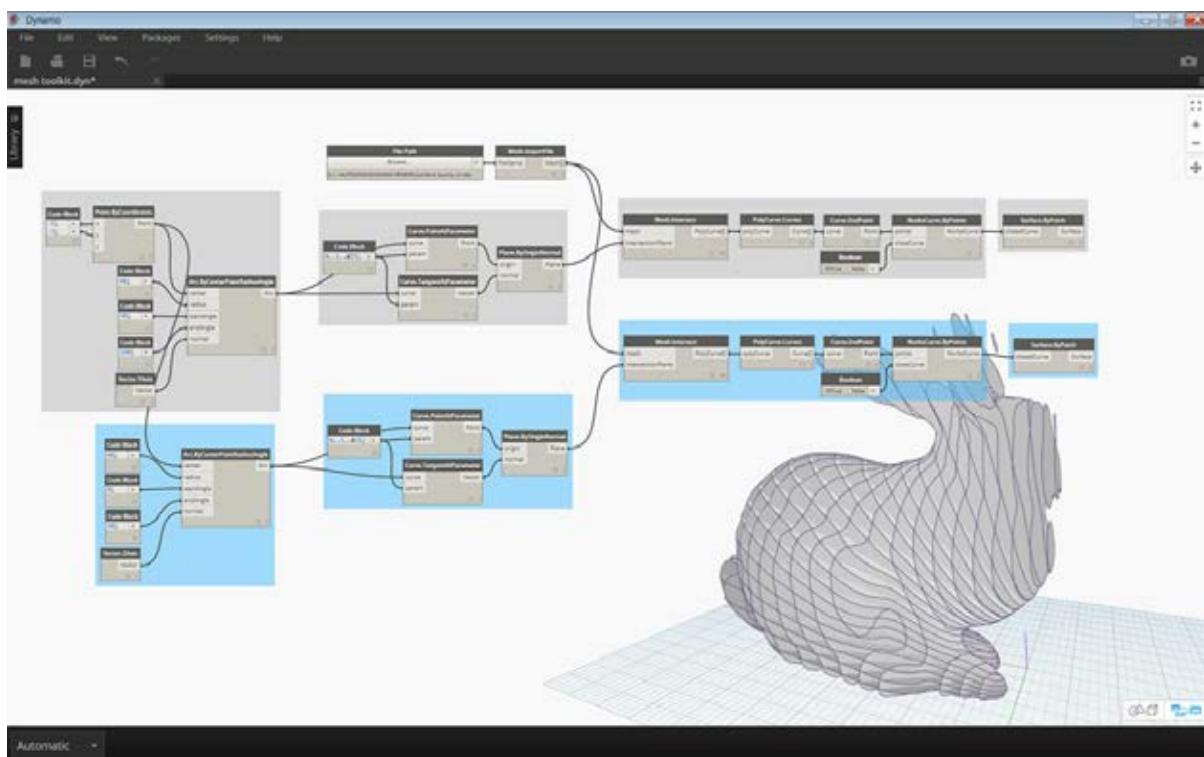
これで、円弧に沿って一連の平面が表示されます。次に、これらの平面を使用してメッシュを交差させます。



1. **Mesh.Intersect** ノードで、読み込んだメッシュと平面を交差させ、一連のポリカーブの輪郭線を作成します。
2. **PolyCurve.Curves** ノードで、ポリカーブを曲線のフラグメントに分割します。
3. **Curve.EndPoint** ノードで、各曲線の終了点を抽出します。
4. **NurbsCurve.ByPoints** ノードで、点群を使用して NURBS 曲線を作成します。次に、Boolean ノードを *True* に設定して曲線を閉じます。



1. **Surface.ByPatch** ノードで、各輪郭線に対してサーフェスを作成して、メッシュの「スライス」を作成します。



2 つ目のスライスの集合を追加すると、2 種類のスライスが格子のように表示されます。

メッシュを使用すると、交差の演算がソリッドよりも高速になることがわかります。この演習で紹介したようなワークフローには、メッシュ

が適しています。

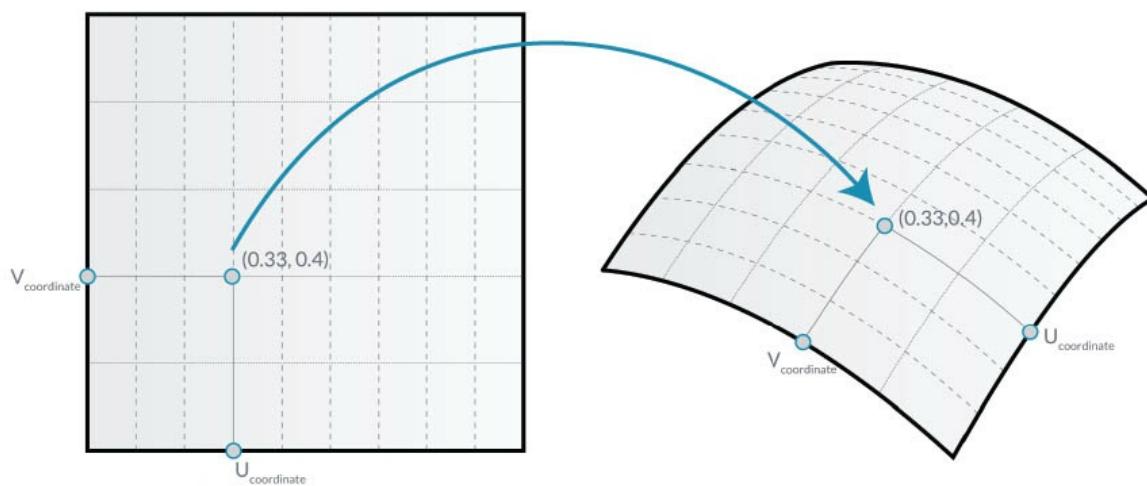
# パッケージを開発する

## パッケージを開発する

Dynamo では、さまざまな方法でパッケージを作成することができます。作成したパッケージは、個人的に使用することも、Dynamo コミュニティで共有することもできます。ここで紹介するケース スタディでは、既存のパッケージの中身を確認しながら、パッケージの設定方法について説明します。このケース スタディは、前の章の演習に基づいて構成されており、UV 座標を使用して Dynamo の特定のサーフェスから別のサーフェスへジオメトリをマッピングする際に使用した一連のカスタム ノードを提供します。

### MapToSurface

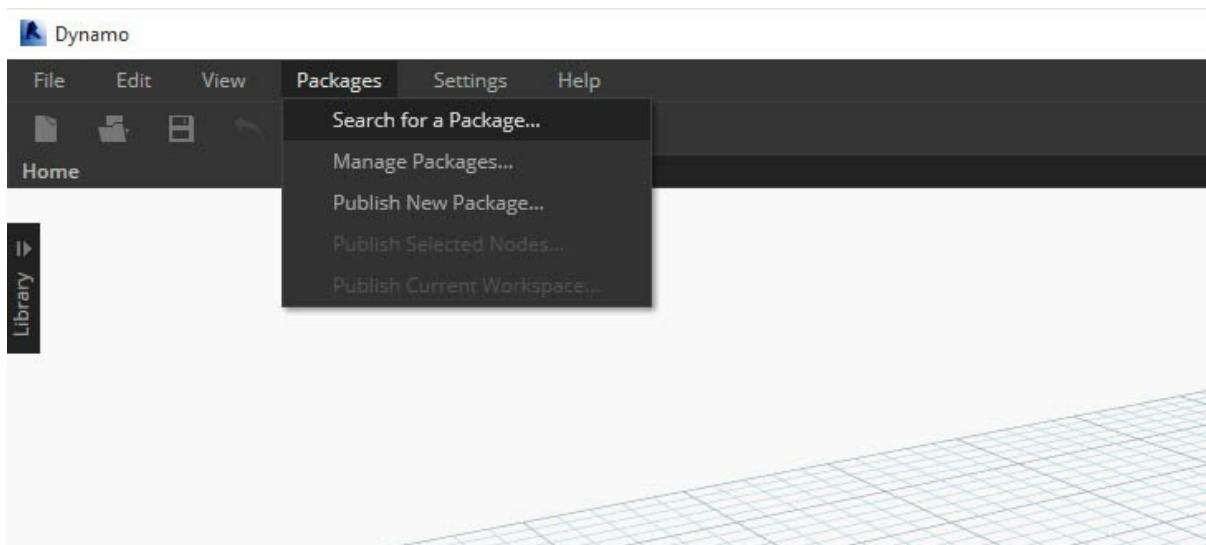
ここでは、点群を特定のサーフェスから別のサーフェスに UV マッピングする演習で使用したサンプル パッケージを使用していきます。ツールの基本部分は、この手引の「[カスタム ノードを作成する](#)」セクションで既に作成されています。ここでは、UV マッピングの概念を理解する方法と、パブリッシュ可能なライブラリ用の一連のツールを開発する方法について確認します。



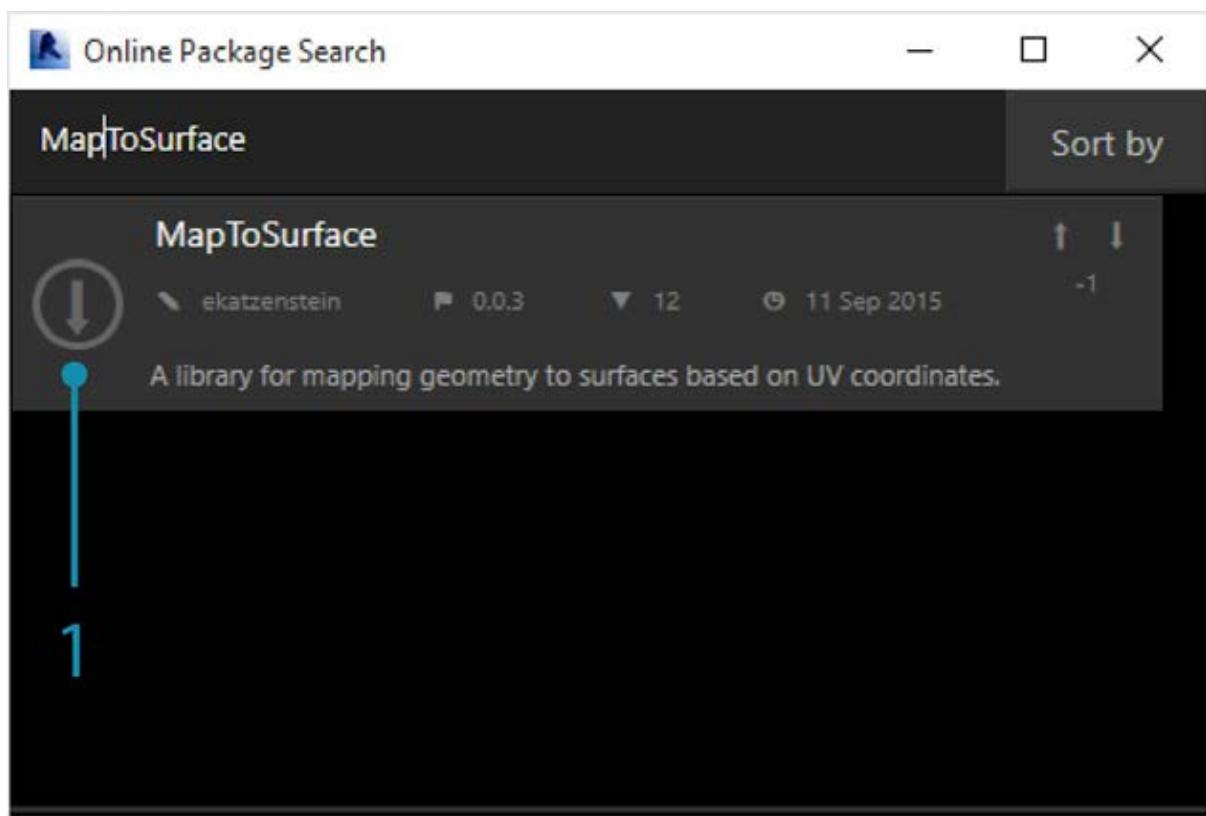
このイメージは、UV 座標を使用して 1 つの点を特定のサーフェスから別のサーフェスにマッピングする場合の例を示しています。パッケージ構成はこの考え方に基づいていますが、パッケージには、より複雑なジオメトリが含まれます。

## パッケージをインストールする

前の章では、Dynamo 内の XY 平面上に定義された複数の曲線に基づいてサーフェスをパネル化する方法について確認しました。このケース スタディでは、この考え方を広げて、より高次元のジオメトリを処理します。ここでは、この構築済みパッケージをインストールし、このパッケージがどのように開発されたかを確認していきます。次のセクションでは、このパッケージのパブリッシュ方法を確認します。

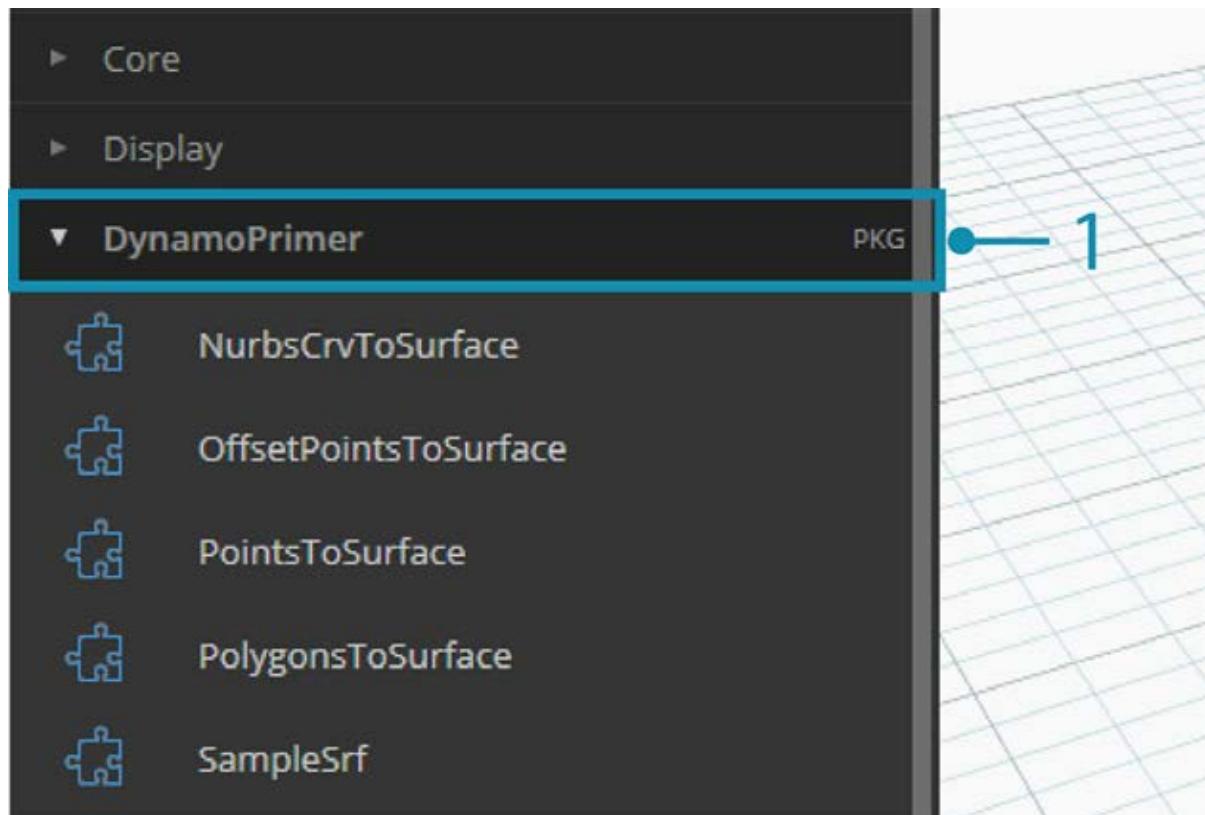


この場合の操作は簡単です。Dynamo で、[パッケージ] > [パッケージの検索] にナビゲートします。



「MapToSurface」というパッケージを検索します。

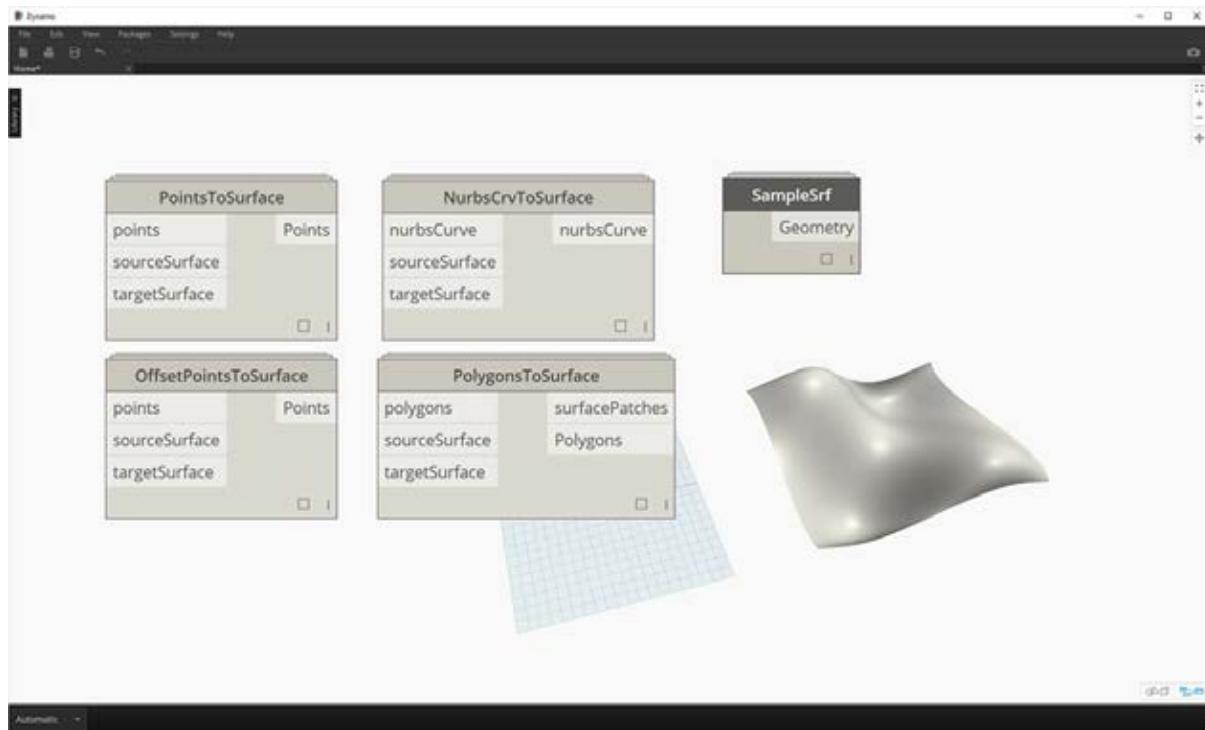
1. パッケージが見つかったら、パッケージ名の左に表示されている大きなダウンロード矢印アイコンをクリックします。この操作により、パッケージが Dynamo にインストールされます。



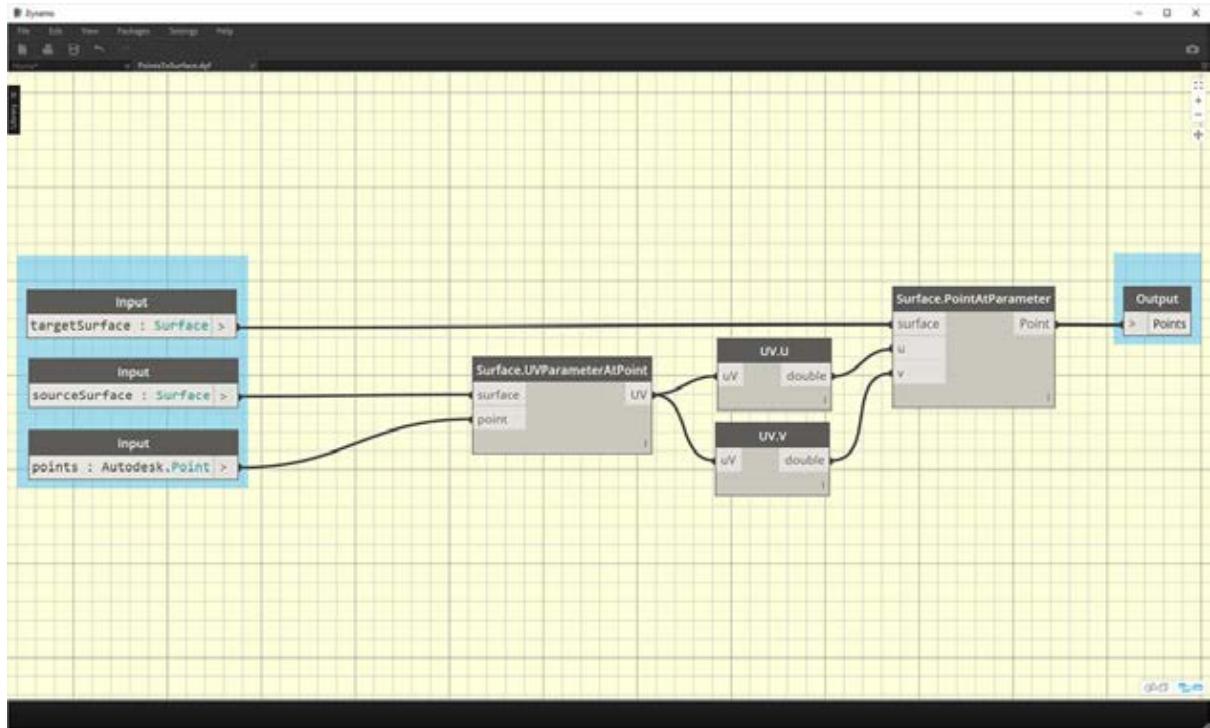
1. パッケージをインストールすると、DynamoPrimer グループまたは Dynamo ライブラリで目的のカスタム ノードを使用できるようになります。これでパッケージのインストールが完了しました。次に、パッケージの設定方法を確認しましょう。

### カスタム ノード

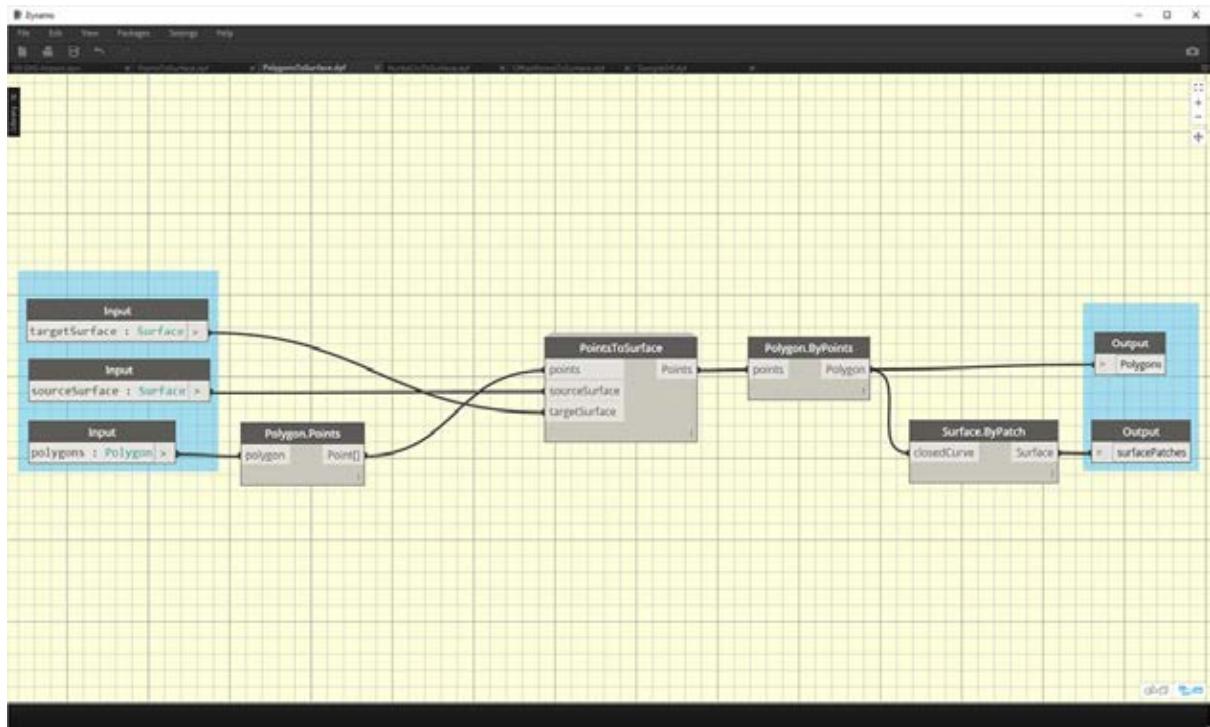
ここで作成するパッケージでは、参照用として既に作成されている 5 つのカスタム ノードを使用します。ここで、各ノードの機能を確認しましょう。一部のカスタム ノードは、他のカスタム ノードを使用して作成されています。また、他のユーザが簡単に理解できるように、グラフにはレイアウトが用意されています。



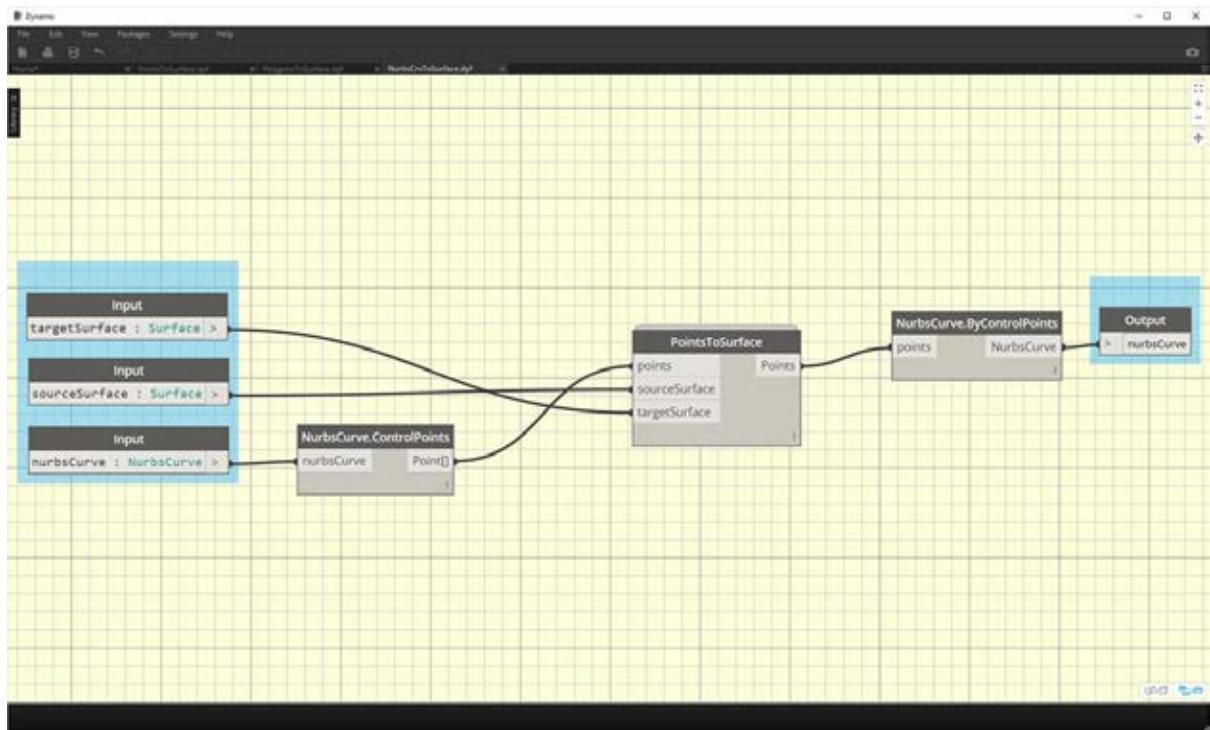
上の図は、5つのカスタムノードによって構成される単純なパッケージを示しています。次の手順で、各カスタムノードの設定について簡単に説明します。



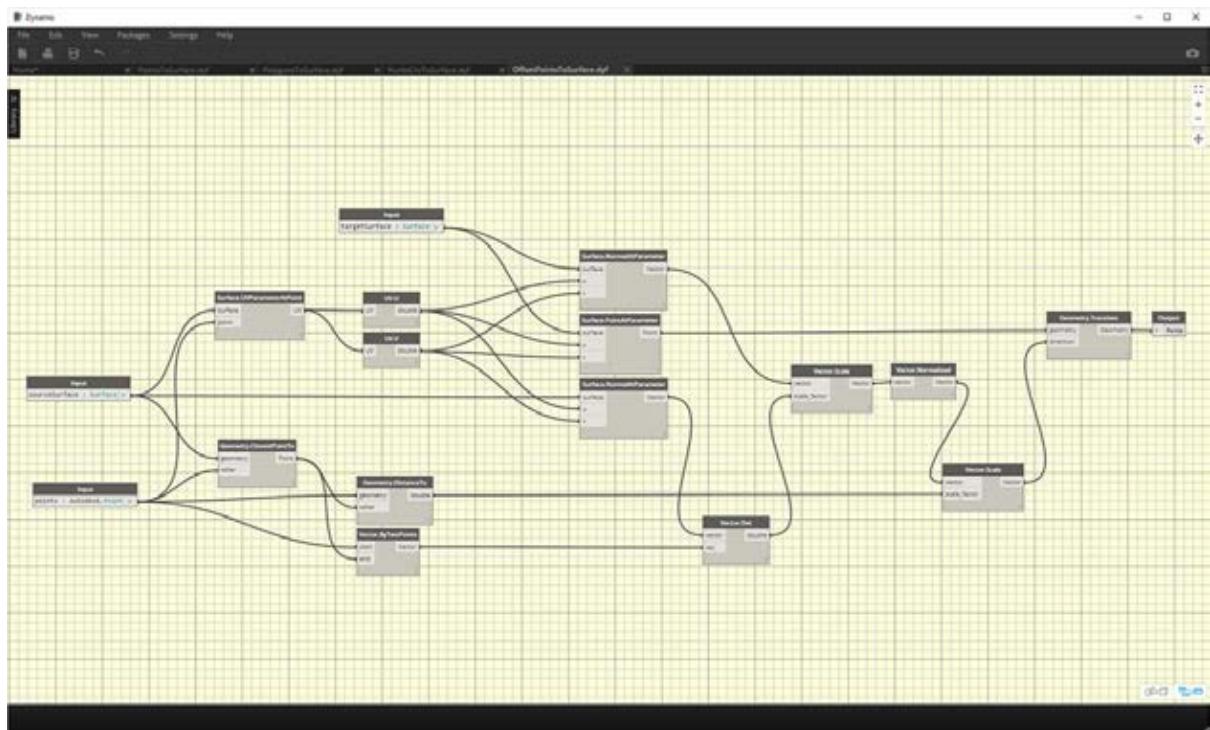
**PointsToSurface** ノードは基本的なカスタムノードで、他のすべてのマッピングノードのベースになるノードです。このノードは、ソース サーフェスの UV 座標の点を、ターゲット サーフェスの UV 座標にマッピングします。点は、複雑なジオメトリを構築するための最も基本的なジオメトリであるため、このロジックを使用して、2D ジオメトリだけでなく 3D ジオメトリについても、特定のサーフェスから別のサーフェスにマッピングすることができます。



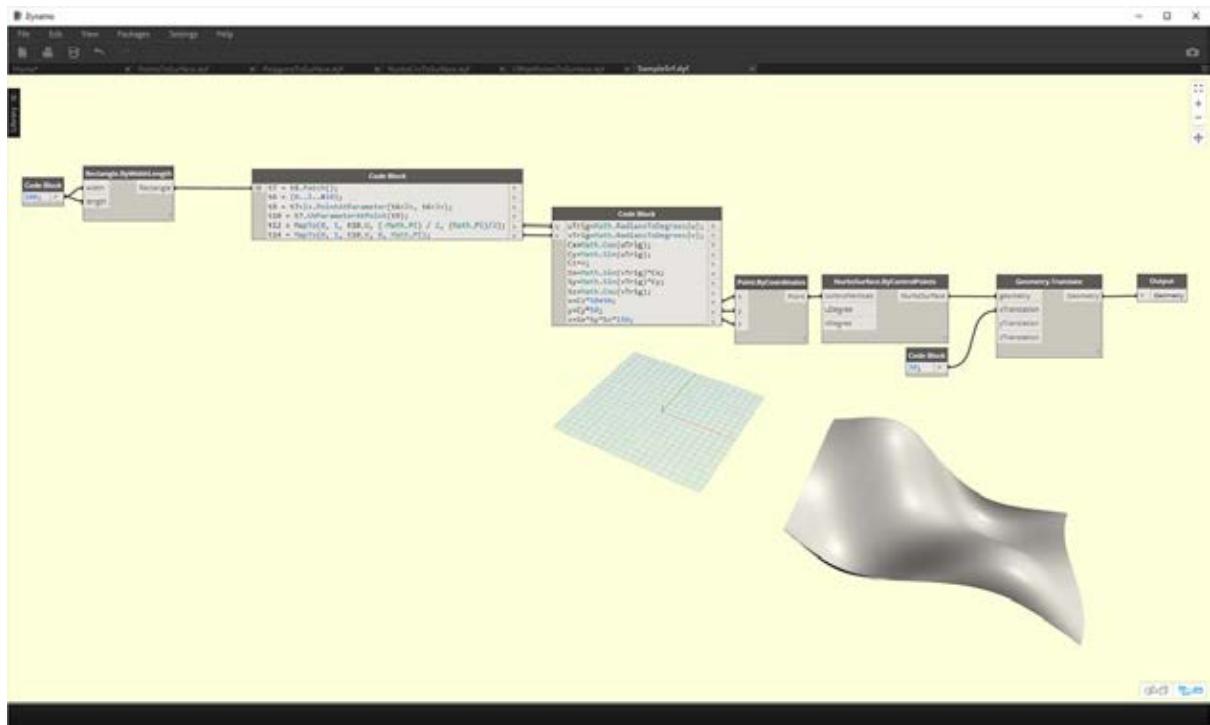
**PolygonsToSurface** ノードを使用すると、1D ジオメトリのマッピングされた点群を 2D ジオメトリに拡張するロジックを、ポリゴンによって簡単に確認することができます。図のように、*PointsToSurface* ノードがこのカスタムノード内にネストされていることがわかります。この方法で各ポリゴンの点群をサーフェスにマッピングし、その点群からポリゴンを再生成することができます。適切なデータ構造(点群のリストのリスト)を維持することにより、ポリゴンを一連の点群に変更した場合でも、それらのポリゴンを個別に保持することができます。



**NurbsCryptoSurface** ノードでは、**PolygonsToSurface** ノードと同じロジックが適用されます。ただし、ここでは、ポリゴンの点群をマッピングするのではなく、NURB 曲線の制御点をマッピングします。



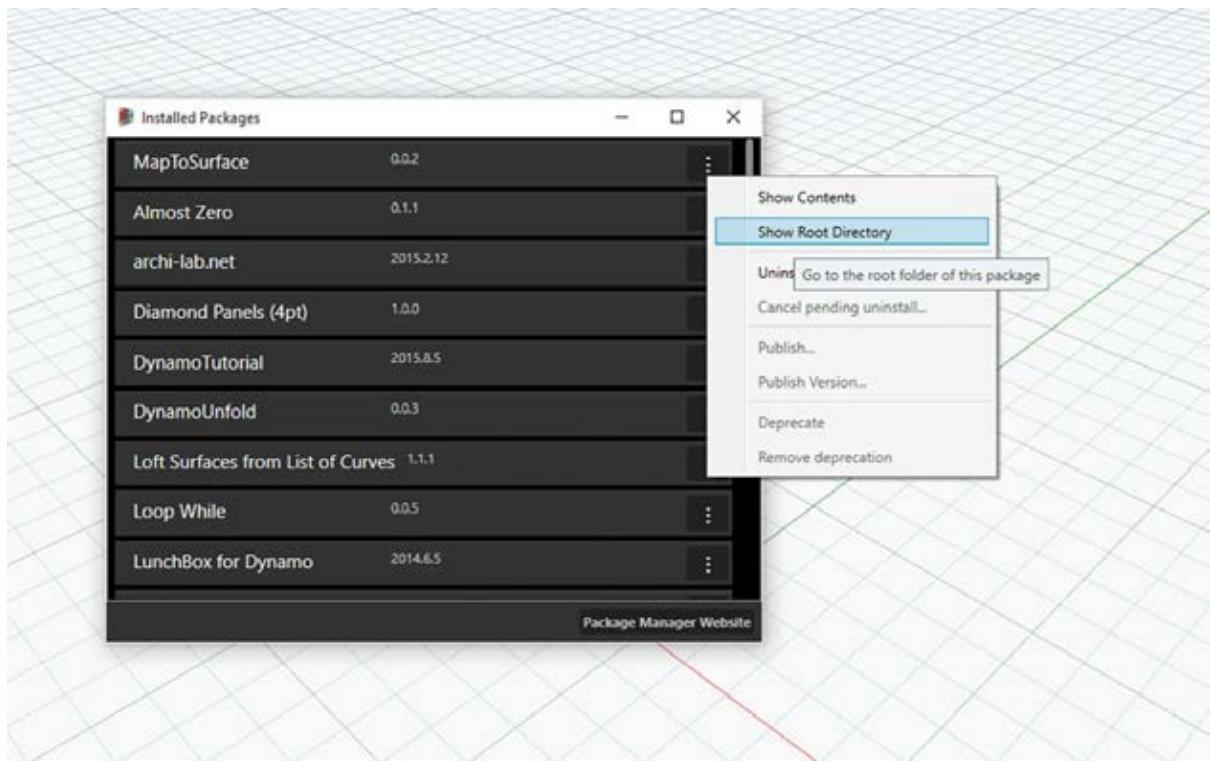
**OffsetPointsToSurface** ノードの構成はやや複雑ですが、その概念は単純です。*PointsToSurface* ノードと同じように、このノードは特定のサーフェスから別のサーフェスに点群をマッピングします。ただし、*OffsetPointsToSurface* ノードは、元のソースサーフェス上には存在しない点群を識別し、その点から最も近い UV パラメータまでの距離を取得して、対応する UV 座標上のターゲットサーフェスの法線にマッピングします。これは、サンプル ファイルを見るとよくわかります。



**SampleSrf** ノードは、サンプル ファイル内のソース グリッドから波形のサーフェスにマッピングするためのパラメータ制御のサーフェスを作成する単純なノードです。

## サンプル ファイル

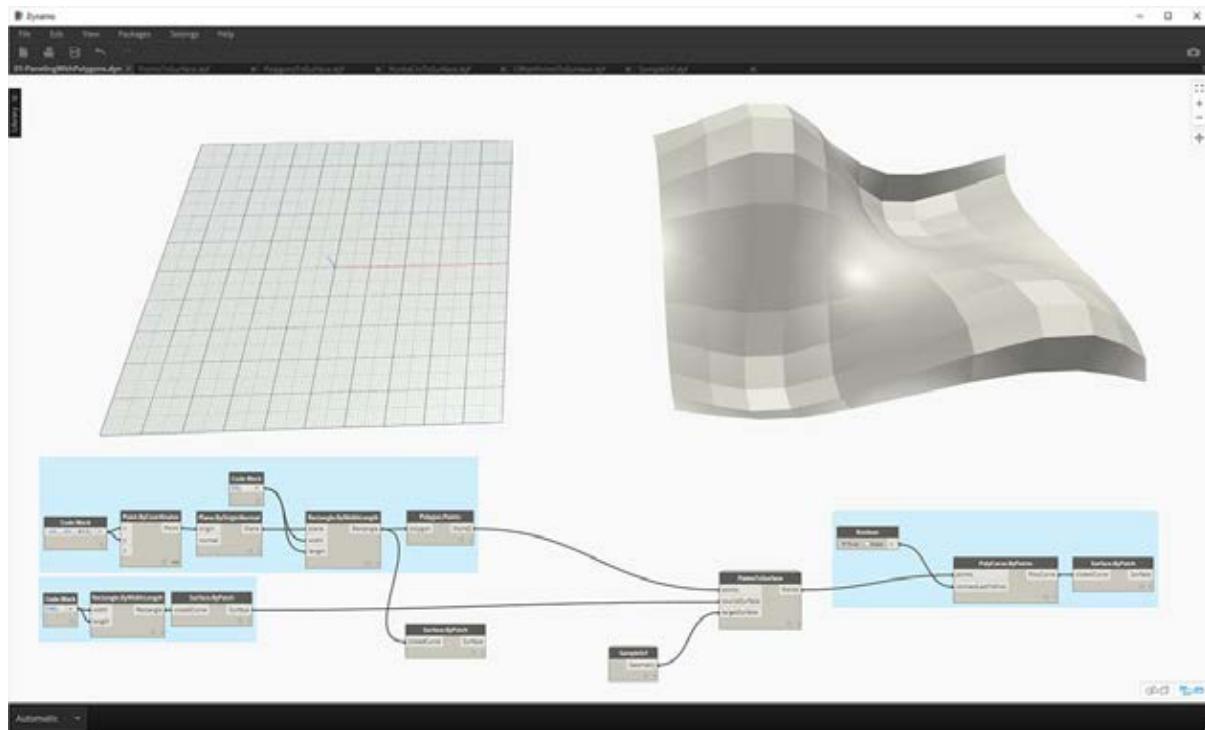
サンプル ファイルは、パッケージのルート フォルダに格納されています(Dynamo で、[パッケージ] > [パッケージを管理...]に移動)。



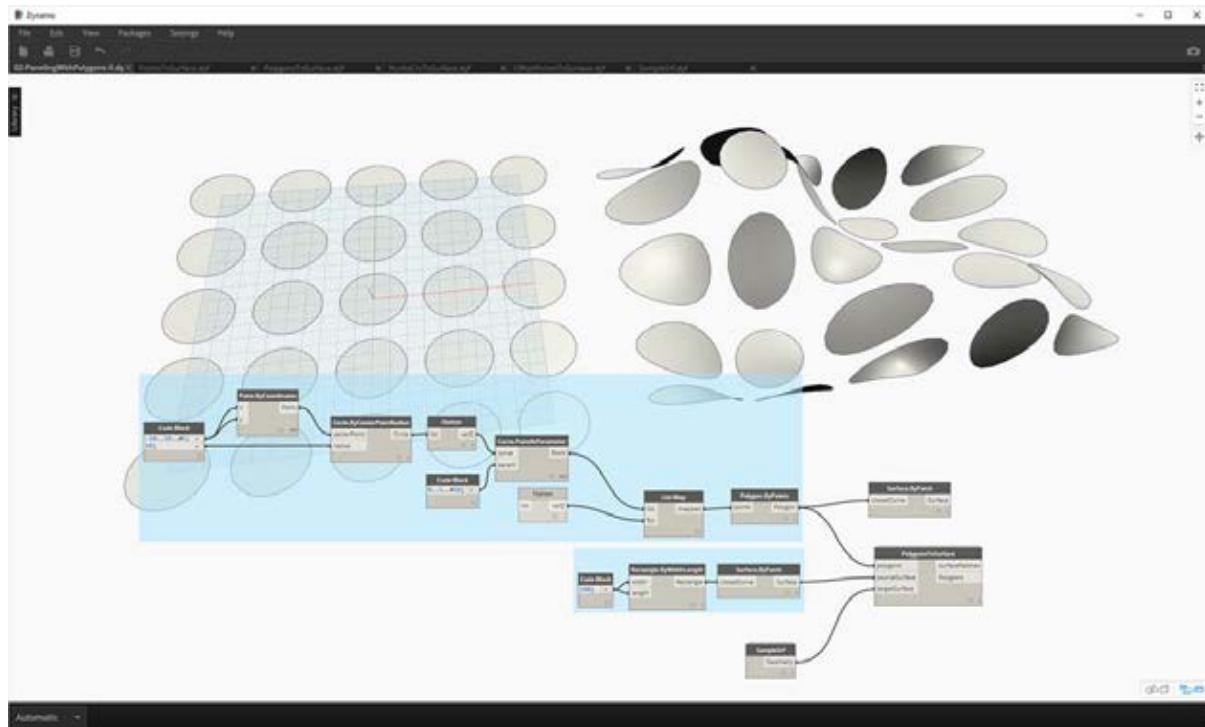
パッケージ管理 ウィンドウで、[MapToSurface] の右側にある垂直に並んだ 3 つの点をクリックし、[ルート フォルダを表示] をクリックします。

ルート フォルダを開いた状態で、「extra」フォルダにナビゲートします。このフォルダには、パッケージ内のすべてのファイル(カスタム ノードを除く)が格納されています。Dynamo パッケージ用のサンプル ファイル(存在する場合)も、このフォルダに格納されています。これ以

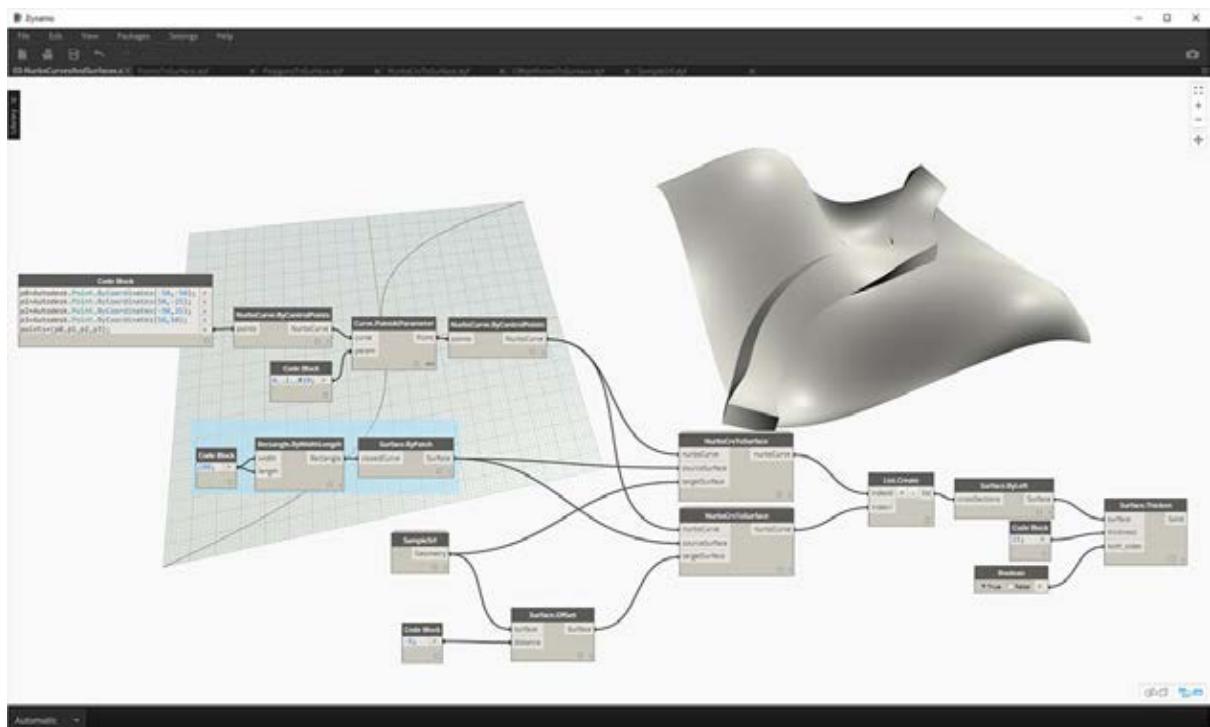
後のスクリーンショットは、各サンプル ファイルの概念を示しています。



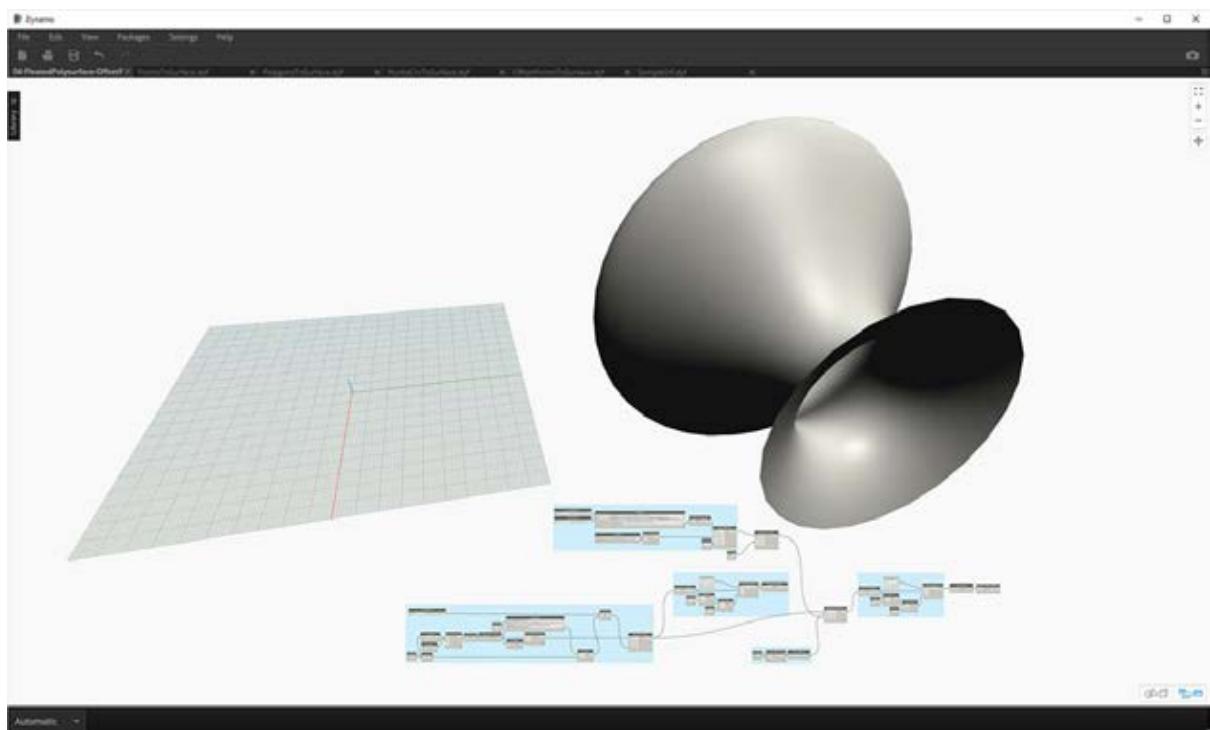
**01-PanelingWithPolygons:** このサンプル ファイルでは、*PointsToSurface* ノードを使用して長方形のグリッドに基づくサーフェスをパネル化する方法を確認することができます。同様のワークフローについては、[前の章](#)で確認しました。



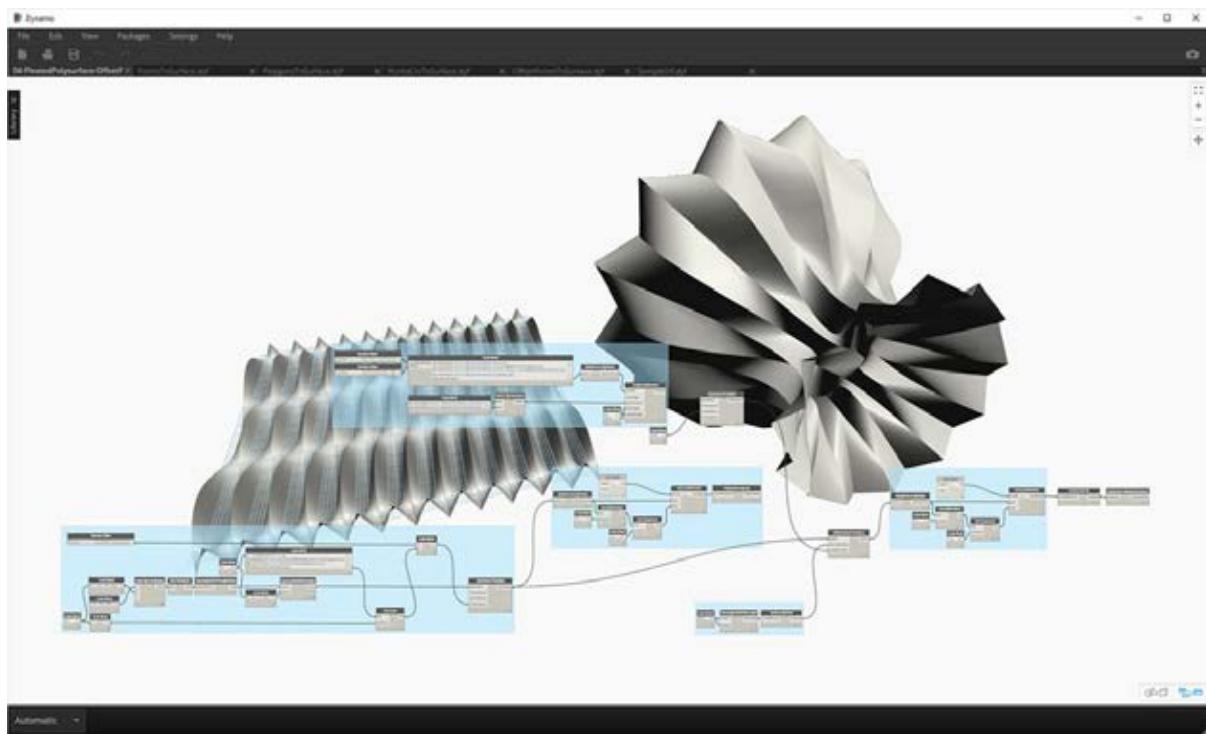
**02-PanelingWithPolygons-II:** このサンプル ファイルでは、同様のワークフローを使用して、特定のサーフェスから別のサーフェスに円弧をマッピングする場合のセットアップ例を確認することができます。このファイルでは *PolygonsToSurface* ノードを使用します。



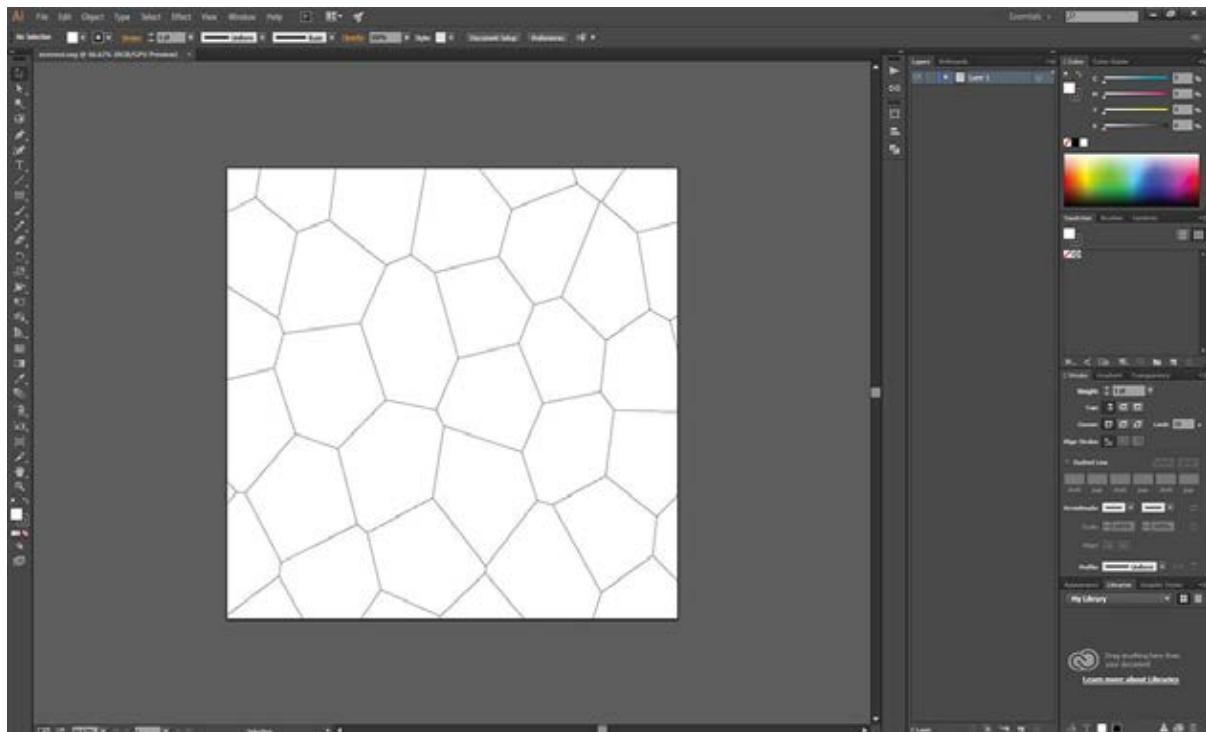
**03-NurbsCrvsAndSurface:** このサンプル ファイルは NurbsCrvToSurface ノードと連携するため、多少複雑な構成になっています。ターゲット サーフェスは指定した距離でオフセットされ、NURB 曲線が元のターゲット サーフェスとオフセット後のサーフェスにマッピングされます。その後、マッピングされた 2 本の曲線がロフトされて 1 つのサーフェスが生成され、そのサーフェスに厚みが加えられます。その結果として出力されるソリッドは、ターゲット サーフェスの法線を表す波形の形状になります。



**04-PleatedPolysurface-OffsetPoints:** このサンプル ファイルでは、ひだがついたポリサーフェスをソース サーフェスからターゲット サーフェスにマッピングする方法について確認することができます。ソース サーフェスはグリッド上に広がる長方形のサーフェスで、ターゲット サーフェスは回転体のサーフェスです。



**04-PleatedPolysurface-OffsetPoints:** このサンプル ファイルでは、ソース サーフェスのソース ポリサーフェスをターゲット サーフェスにマッピングする方法について確認することができます。



**05-SVG-Import:** カスタム ノードを使用すると、さまざまなタイプの曲線をマッピングすることができます。このサンプル ファイルでは、Illustrator から書き出した SVG ファイルを参照し、読み込んだ曲線をターゲット サーフェスにマッピングすることができます。

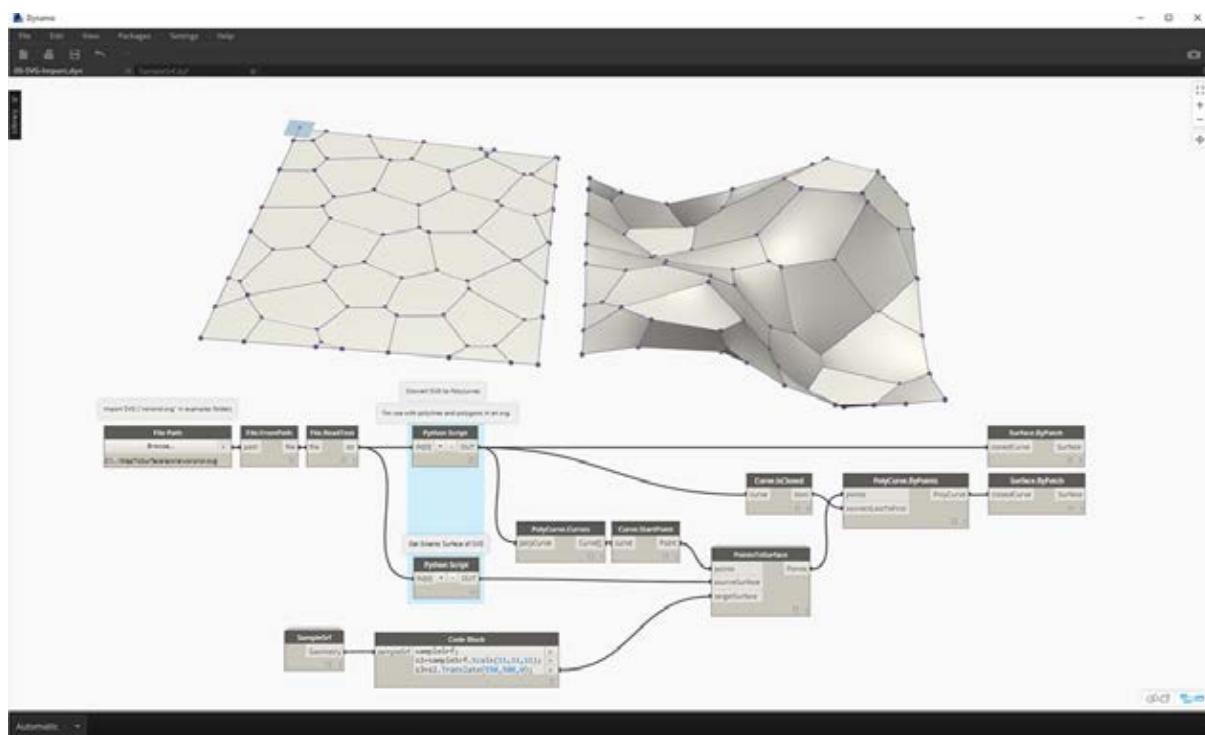
```

Edit Python Script...
1 import clr
2 import re
3 clr.AddReference("System.Xml")
4 import System.Xml
5
6 xmlDoc = System.Xml.XmlDocument()
7 xmlDoc.Load(IN[0])
8
9 OUT=[]
10
11 #for shorthanded quadratic and cubic bezier curves
12 svgDict={}
13 svgDict["et"]=[]
14 svgDict["ed"]=[]
15
16 def segsFromPath(data):
17     subOUT=[]
18     splitPath=re.findall('[A-Za-z][^A-Za-z]*',"".join(line.strip() for line in data))
19     return splitPath
20
21 def viewBox():
22     items = xmlDoc.GetElementsByTagName("svg")
23     for item in items:
24         box=item.getAttribute("viewBox")
25     nums=box.split(' ')
26     floats=[]
27     for num in nums:
28         floats.append(float(num))
29     OUT.append(["viewBox",floats])
30
31
32 def ptsFromPoly(data):
33     subOUT=[]
34     pts=data.split(" ")
35     for points in pts:
36         ptList=points.split(",")
37         if len(ptList)>1:
38             numX=float(ptList[0])
39             numY=float(ptList[1])
40             geoPt=[numX,numY]
41             subOUT.append(geoPt)

```

Accept Changes      Cancel

**05-SVG-Import:** .svg ファイルの構文を解析することにより、曲線が .xml 形式から Dynamo のポリカーブに変換されます。

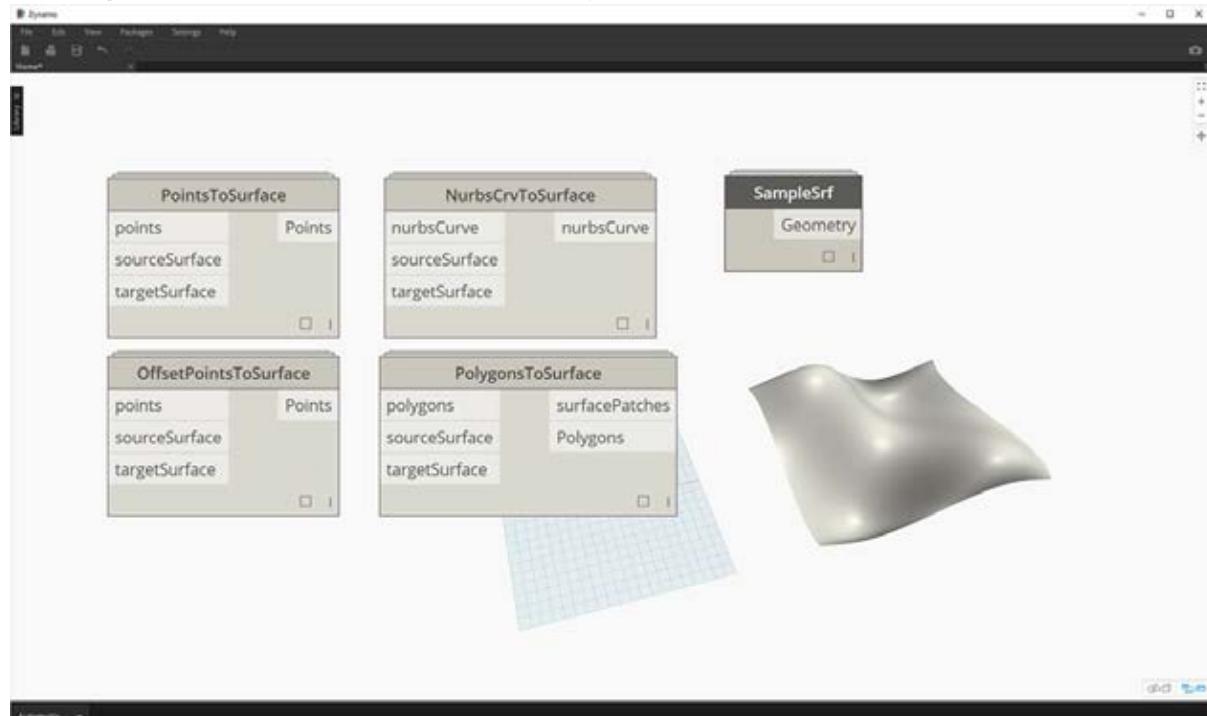


**05-SVG-Import:** 読み込んだ曲線がターゲット サーフェスにマッピングされます。これにより、Illustrator でパネルを明示的に(ポイント アンド クリック操作で)設計し、そのパネルを Dynamo で読み込んでターゲット サーフェスに適用することができます。

# パッケージをパブリッシュする

## パッケージをパブリッシュする

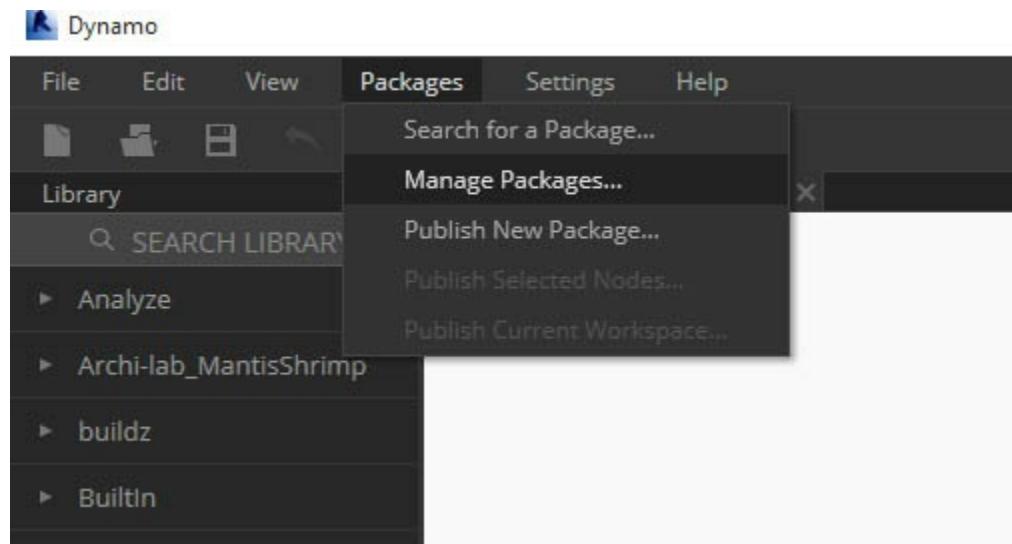
前のセクションでは、カスタム ノードとサンプル ファイルを使用して *MapToSurface* パッケージを設定する方法について確認しました。では、ローカルで作成したパッケージはどのようにパブリッシュすればよいでしょうか。このケース スタディでは、ローカル フォルダ内のファイル セットを使用してパッケージをパブリッシュする方法について確認します。



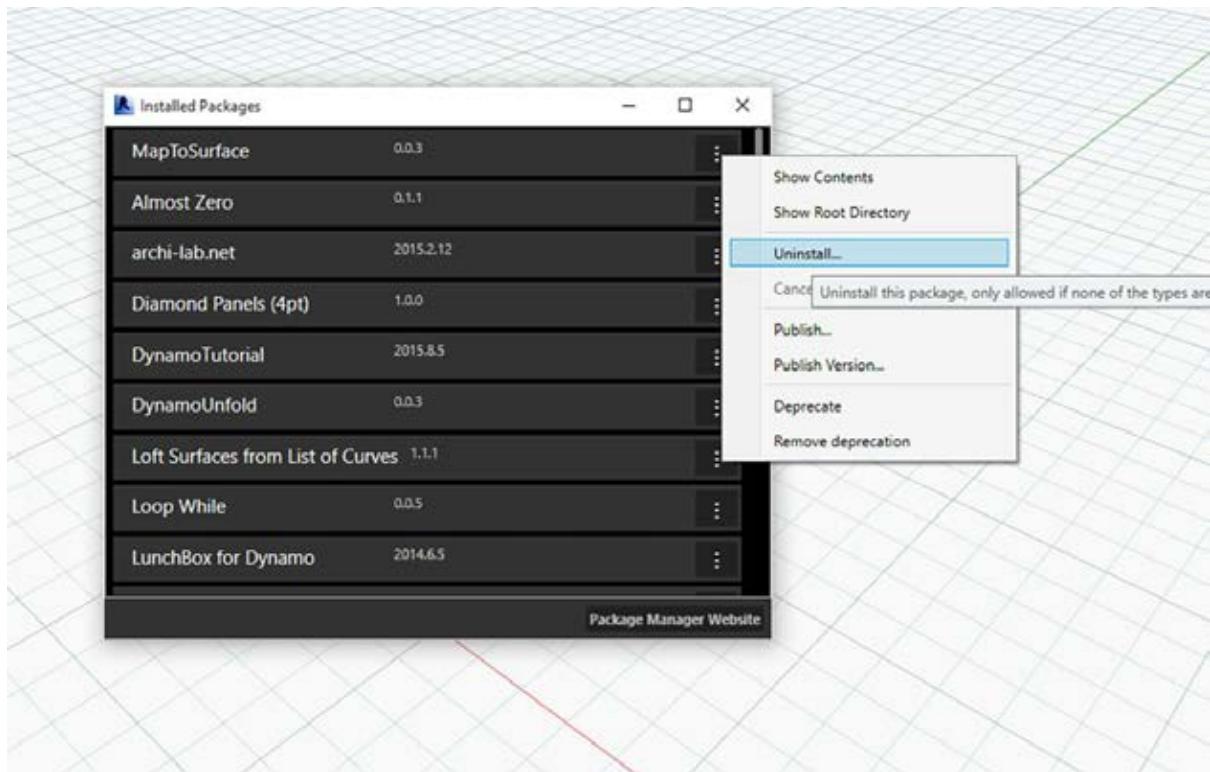
パッケージは、さまざまな方法でパブリッシュすることができます。ここでは、パッケージをローカルにパブリッシュして作成し、オンラインでパブリッシュする方法を確認していきます。最初に、パッケージ内のすべてのファイルを格納するフォルダを作成します。

## パッケージをアンインストールする

前の演習で *MapToSurface* パッケージをインストールした場合は、同じパッケージを使用しないようにするために、このパッケージをアンインストールしてください。



最初に、[パッケージ] > [パッケージを管理...]に移動します。

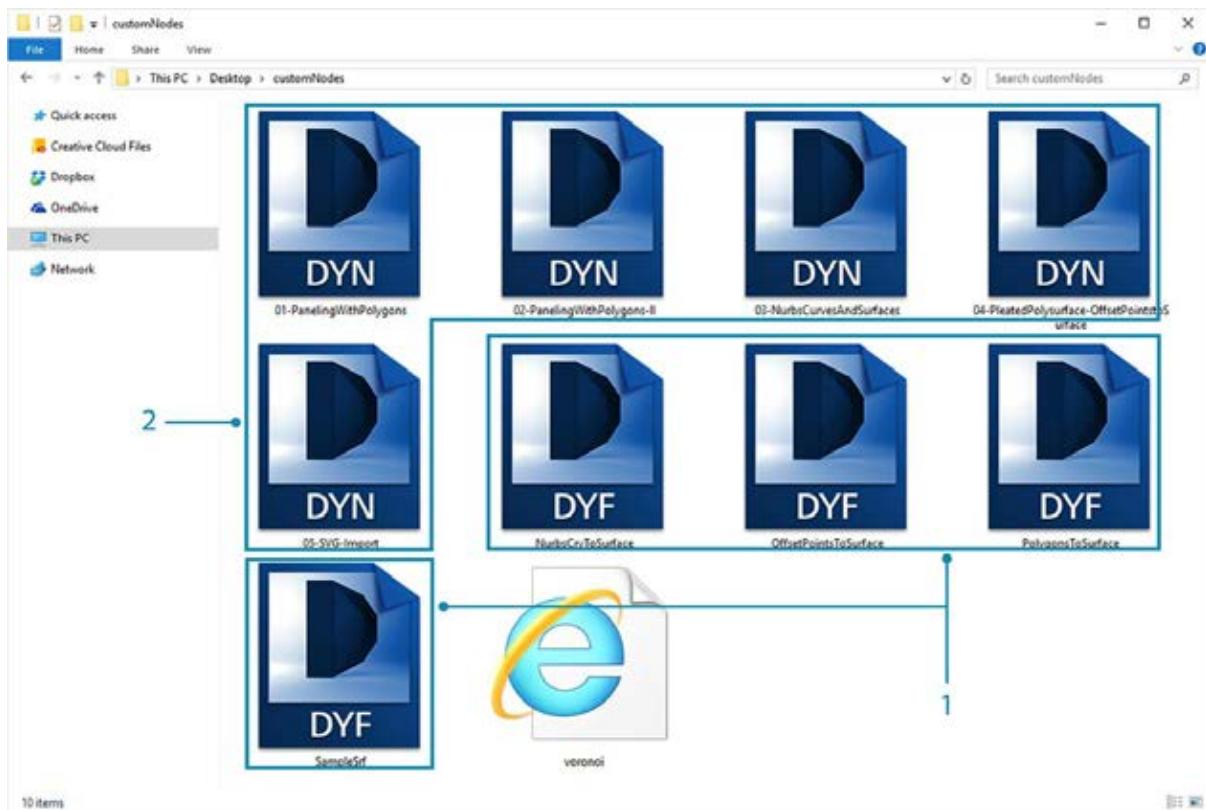


[*MapToSurface*]に対応するボタンを選択し、[アンインストール...]を選択します。次に、Dynamo を再起動します。[パッケージを管理]ウィンドウをもう一度開いて、*MapToSurface* が表示されていないことを確認してください。これで、作業を開始する準備ができました。

### パッケージをローカルにパブリッシュする

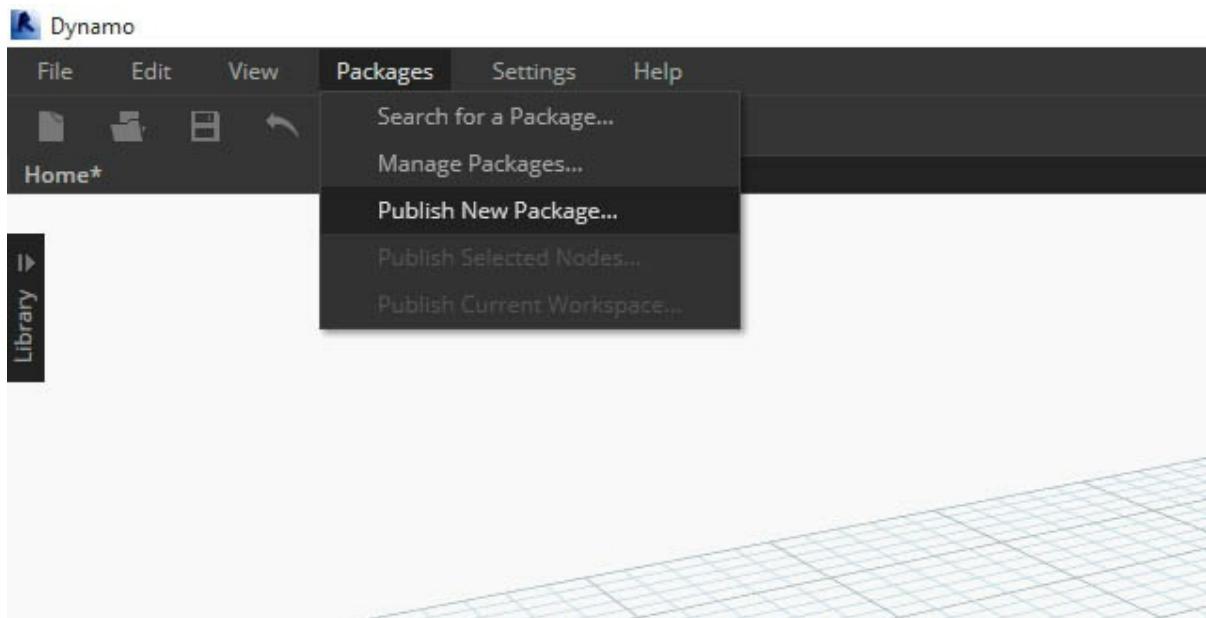
注: この手引の作成時点では、*Dynamo Studio* または *Dynamo for Revit* でのみ *Dynamo* パッケージをパブリッシュすることができます。*Dynamo Sandbox* には、パブリッシュ機能は用意されていません。

このパッケージのケーススタディに付属しているサンプルファイル(Zero-Touch-Examples.zip)をダウンロードして解凍してください(右クリックして[名前を付けてリンク先を保存...]を選択)。すべてのサンプルファイルの一覧については、付録を参照してください。[MapToSurface.zip](#)

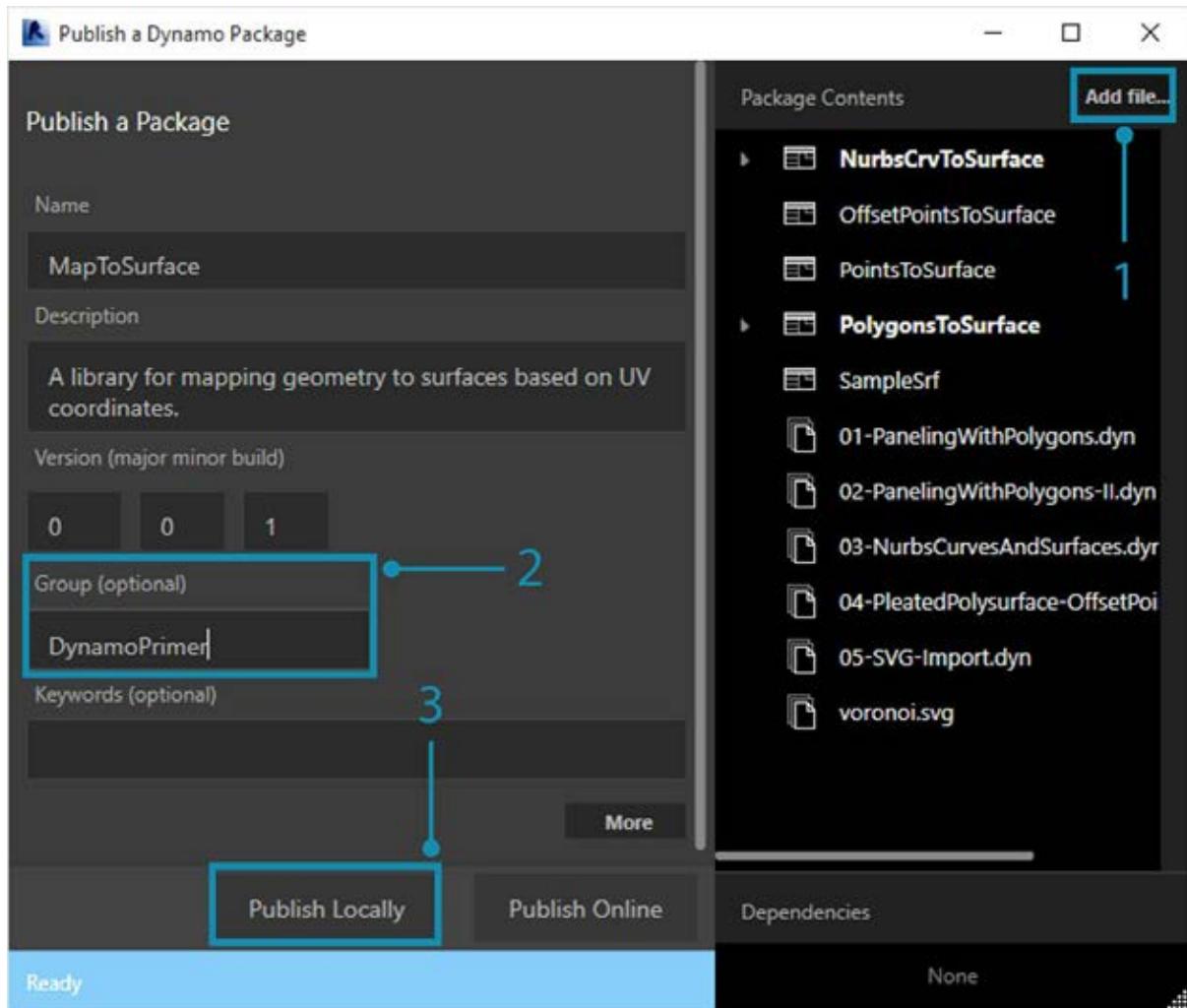


この演習で、パッケージを初めて送信することになります。サンプル ファイルとカスタム ノードは、すべて 1 つのフォルダ内に格納されています。このフォルダが作成されれば、Dynamo Package Manager にパッケージをアップロードすることができます。

1. このフォルダには、5 つのカスタム ノード(.dyf)が格納されています。
2. このフォルダには、5 つのサンプル ファイル(.dyn)と、1 つの読み込み済みベクトル ファイル(.svg)も格納されています。これらのファイルは、カスタム ノードの使用方法を理解するための演習用のファイルです。

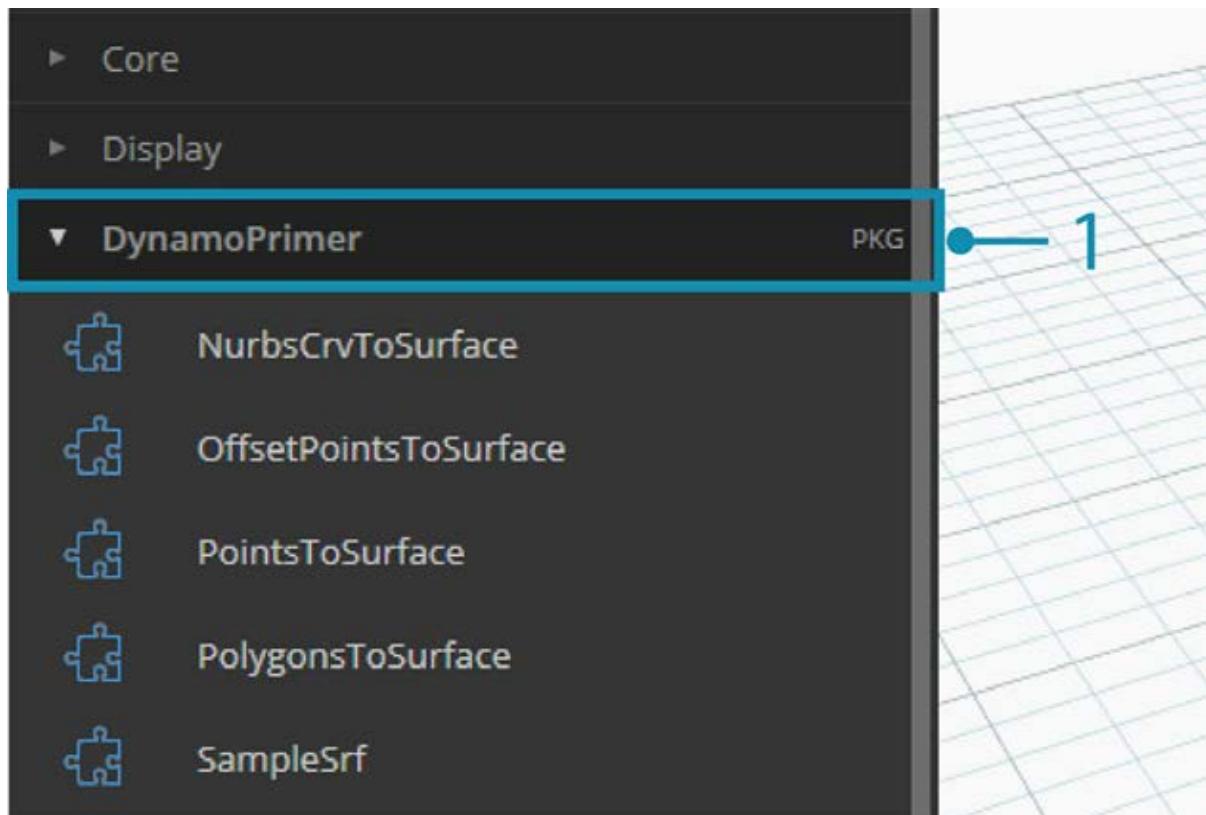


Dynamo で、[パッケージ] > [新しいパッケージをパブリッシュ...]をクリックします。

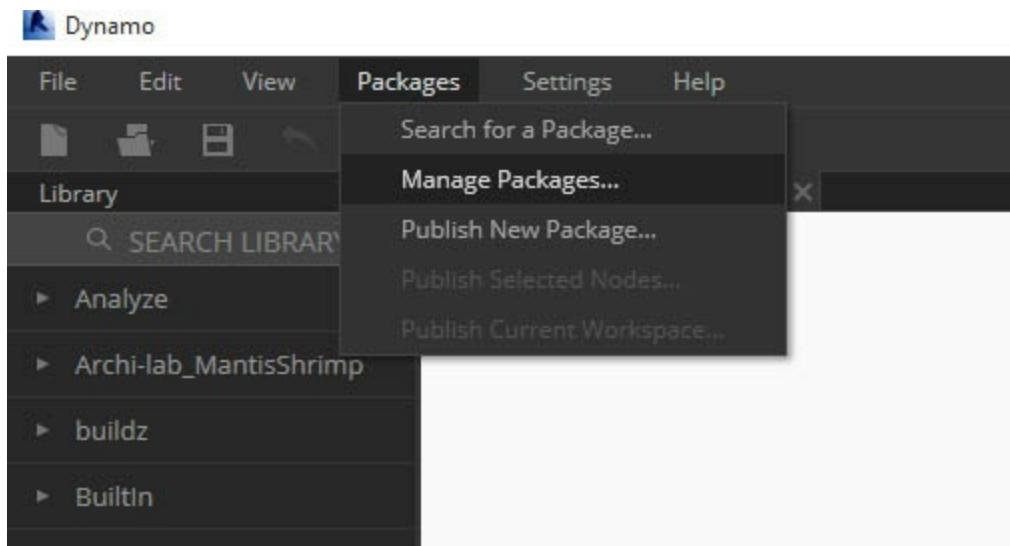


[Dynamo パッケージをパブリッシュ] ウィンドウの左側には、関連フォームが既に入力されています。

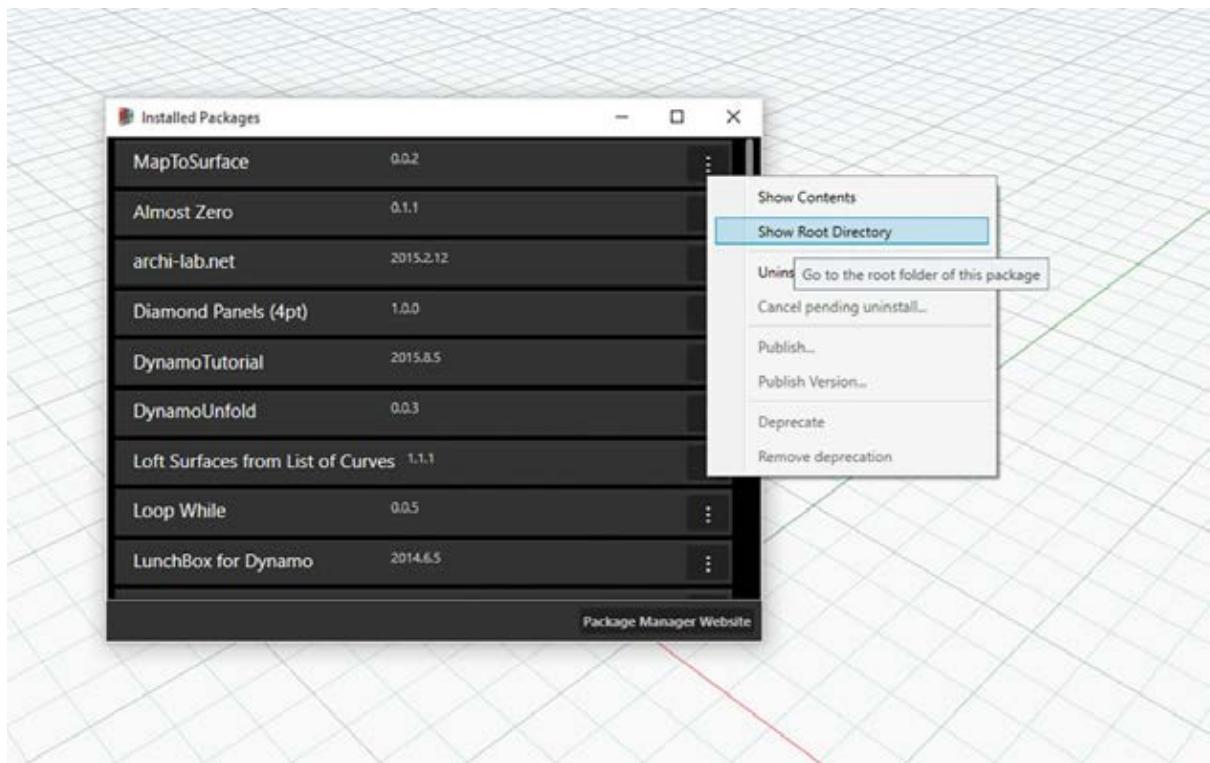
1. [ファイルを追加] をクリックして、画面の右側にあるフォルダ構造から既にファイルが追加されています(.dyf ファイル以外のファイルを追加するには、ブラウザ ウィンドウのファイル タイプを[すべてのファイル(.)]"。カスタム ノード(.dyf)やサンプル ファイル(.dyn)など、すべてのファイルが追加されていることを確認してください。パッケージをパブリッシュすると、Dynamo によってこれらのファイルが分類されます。
2. [グループ] フィールドを使用して、Dymano UI でカスタム ノードを検索するためのグループを定義します。
3. [ローカルにパブリッシュ] をクリックして、パッケージをパブリッシュします。次に、[オンラインでパブリッシュ]ではなく[ローカルにパブリッシュ]をクリックします。これは、多数の複数パッケージを Package Manager にパブリッシュしないようにするためにです。



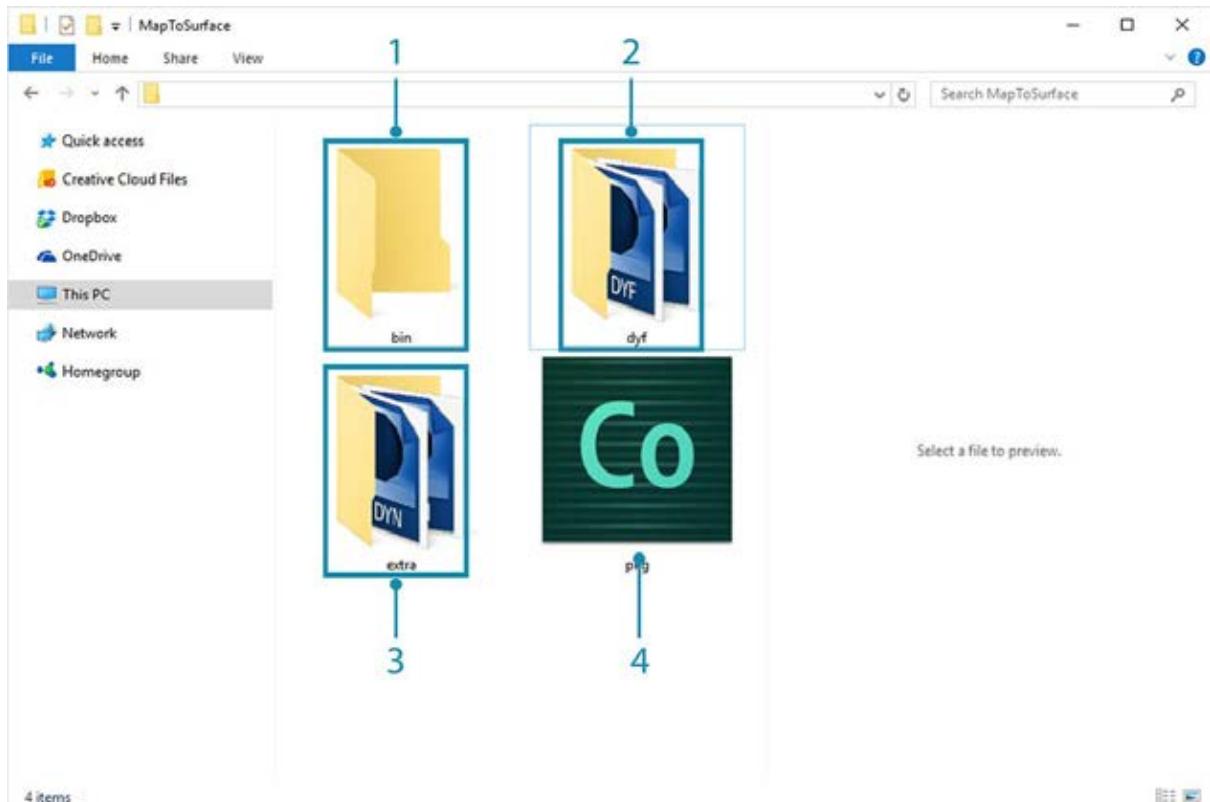
1. パブリッシュが完了すると、DynamoPrimer グループまたは Dynamo ライブラリで目的のカスタム ノードを使用できるようになります。



次に、ルート フォルダを開き、作成したパッケージが Dynamo でどのようにフォーマットされているかを確認します。これを行うには、[パッケージ] > [パッケージを管理...] をクリックします。



パッケージ管理ウィンドウで、「MapToSurface」の右側にある垂直に並んだ 3 つの点をクリックし、[ルート フォルダを表示]をクリックします。

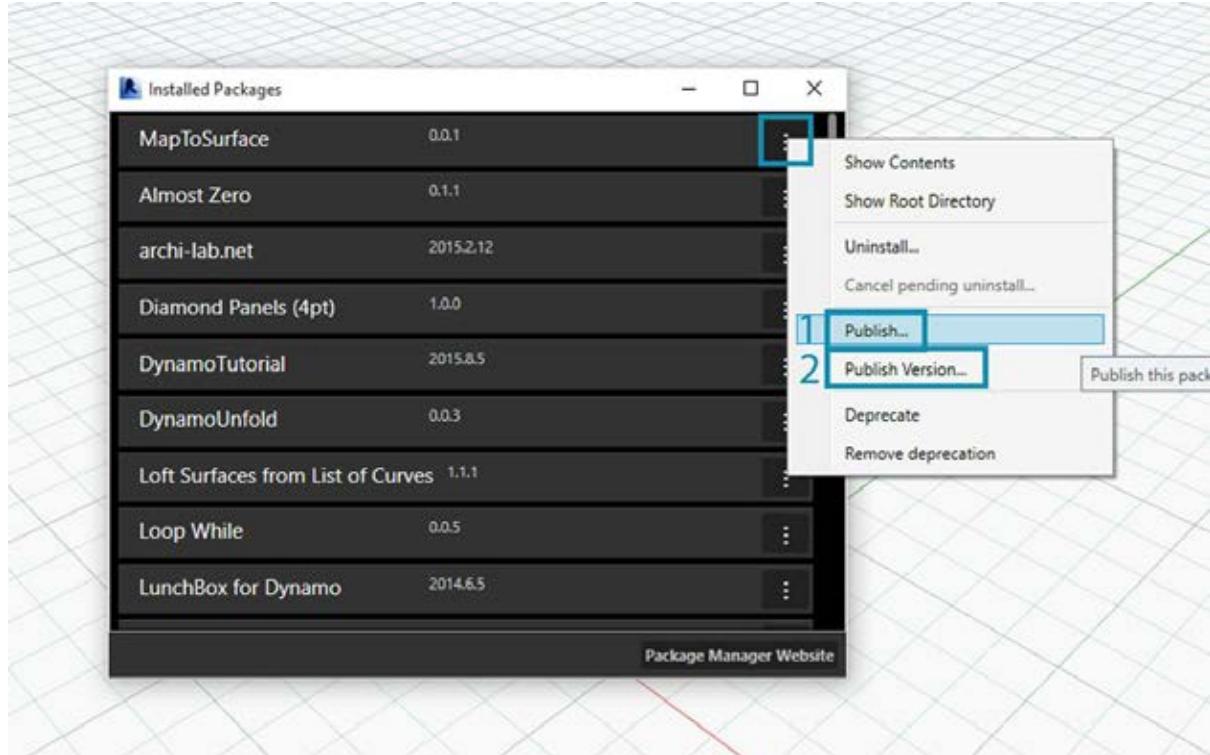


ルート フォルダは、パッケージのローカルの場所にあります(ここまで手順で、パッケージはローカルにパブリッシュされています)。Dynamo は、このフォルダを参照してカスタム ノードを読み込みます。そのため、このフォルダを保存する場合は、デスクトップではなく、ローカルの永続的な場所に保存する必要があります。Dynamo パッケージ フォルダの内容は次のとおりです。

1. *bin* フォルダには、C# ライブリまたは Zero-Touch ライブリを使用して作成された .dll ファイルが格納されます。このパッケージにはこうしたファイルがないため、このフォルダは空になっています。

2. dyf フォルダには、カスタム ノードが格納されます。このフォルダを開くと、このパッケージのすべてのカスタム ノード(.dyf ファイル)が表示されます。
3. extra フォルダには、すべての追加ファイルが格納されます。通常、これらのファイルは、Dynamo ファイル(.dyn)または必須の追加ファイル(.svg、.xls、.jpeg、.sat など)です。
4. pkg ファイルは、パッケージの設定を定義する基本のテキスト ファイルです。このファイルは Dynamo によって自動的に作成されますが、必要な場合は編集することができます。

## パッケージをオンラインでパブリッシュする



注: 独自のパッケージを実際にパブリッシュしない場合は、この手順を実行しないでください。

1. パブリッシュの準備ができたら、[パッケージを管理] ウィンドウで MapToSurface の右に表示されているボタンを選択し、[パブリッシュ...] を選択します。
2. 既にパブリッシュされているパッケージを更新する場合、[パブリッシュ バージョン] を選択すると、パッケージのルート フォルダ内の新しいファイルに基づいて、パッケージがオンラインで更新されます。

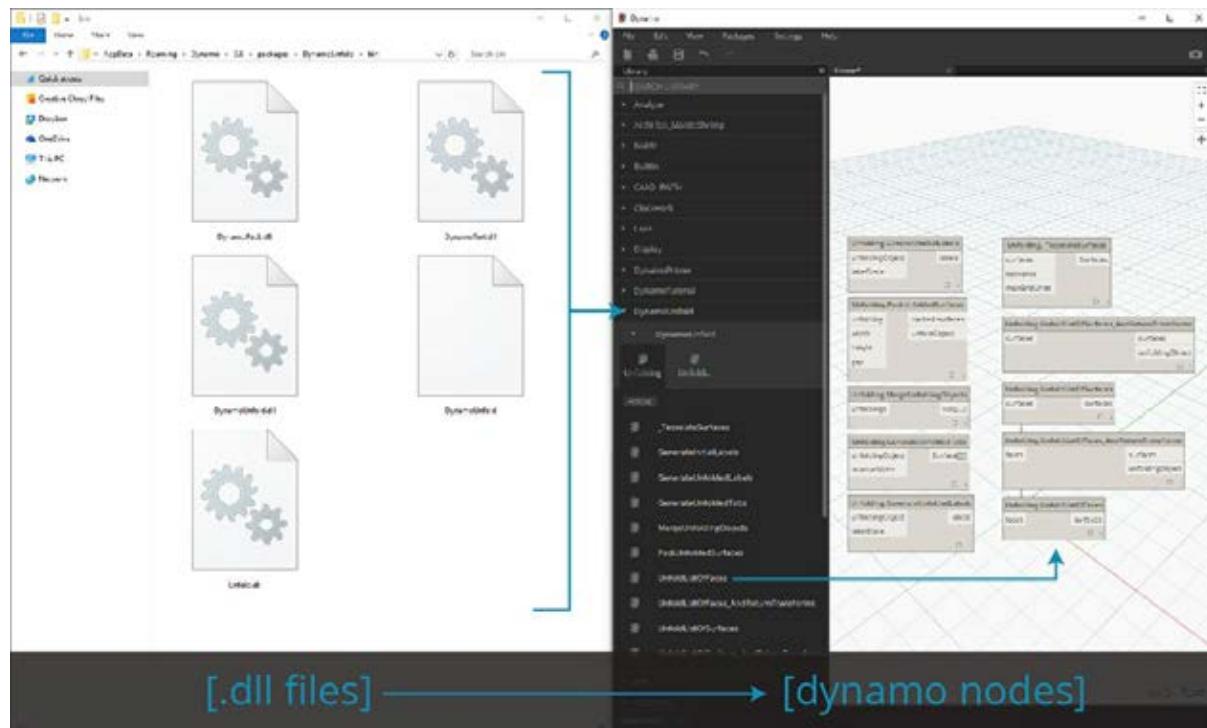
### [パブリッシュ バージョン...] オプション

パブリッシュしたパッケージのルート フォルダ内にあるファイルを更新した場合、[パッケージを管理] ウィンドウで [パブリッシュ バージョン...] を選択すると、新しいバージョンのパッケージをパブリッシュすることができます。この方法により、シームレスにコンテンツを更新してコミュニティ間で共有することができます。[パブリッシュ バージョン] オプションは、ユーザがパッケージを保守している場合にのみ機能します。

# Zero-Touch の概要

## Zero-Touch の概要

Zero-Touch Importing とは、C# ライブリを読み込むための単純なポイントアンドクリック操作のことです。Dynamo は、.dll ファイルの public メソッドを読み取って Dynamo ノードに変換します。Zero-Touch を使用して、独自のカスタム ノードとカスタム パッケージを開発し、外部のライブラリを Dynamo 環境に読み込むことができます。



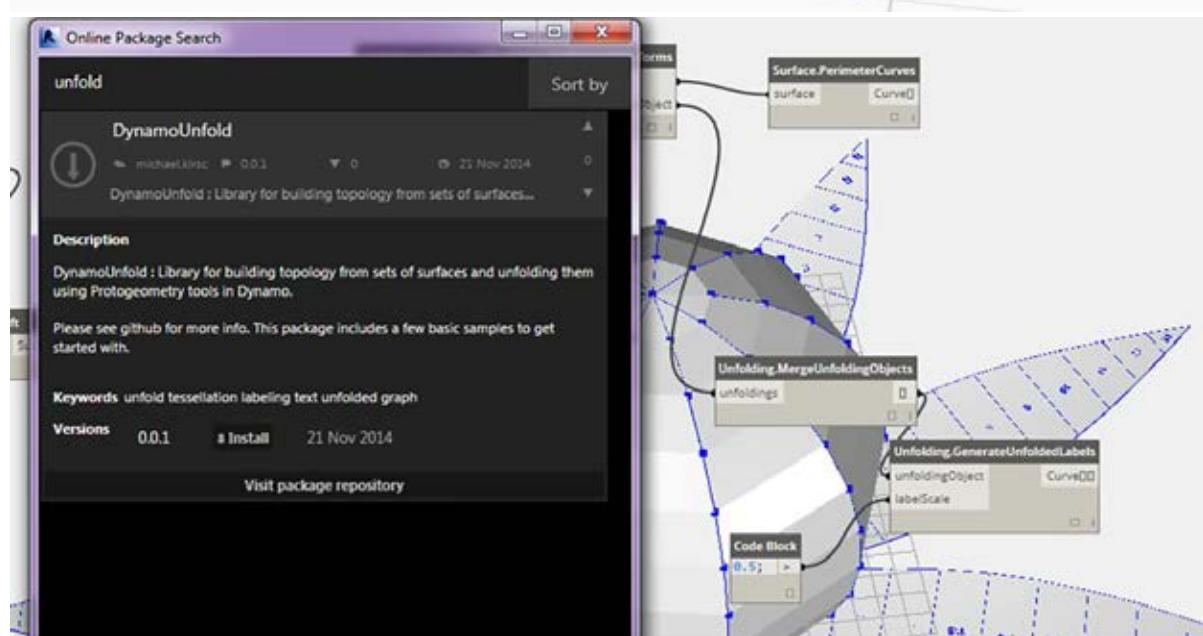
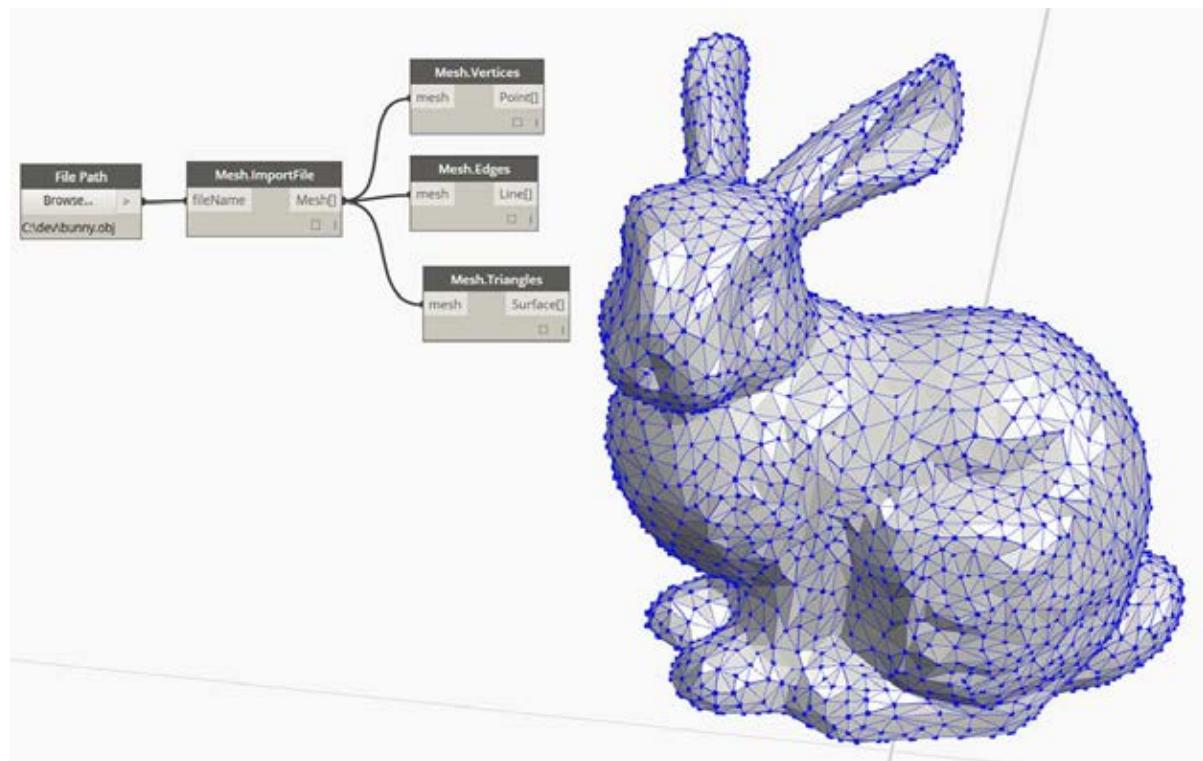
Zero-Touch により、Dynamo 用に開発されたものではないライブラリを読み込み、一連の新しいノードを作成することができます。現在の Zero-Touch 機能は、Dynamo プロジェクトのクロスプラットフォーム志向性を体现しています。

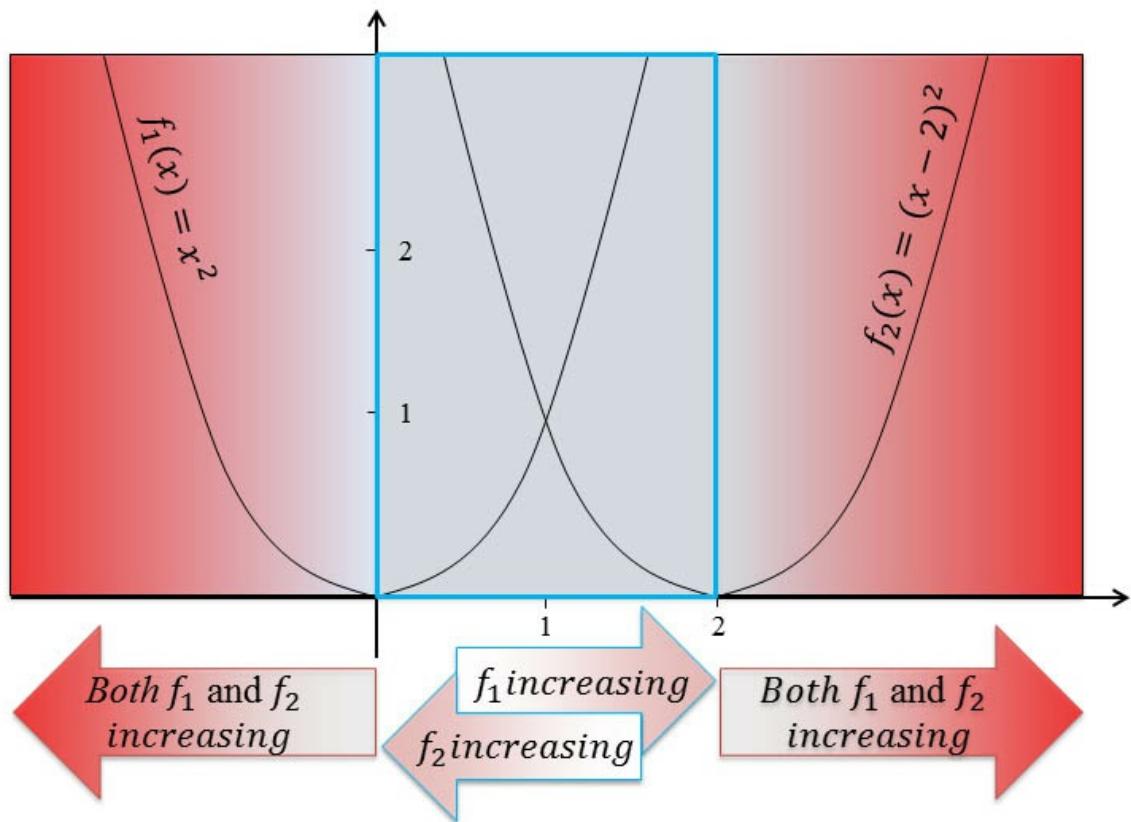
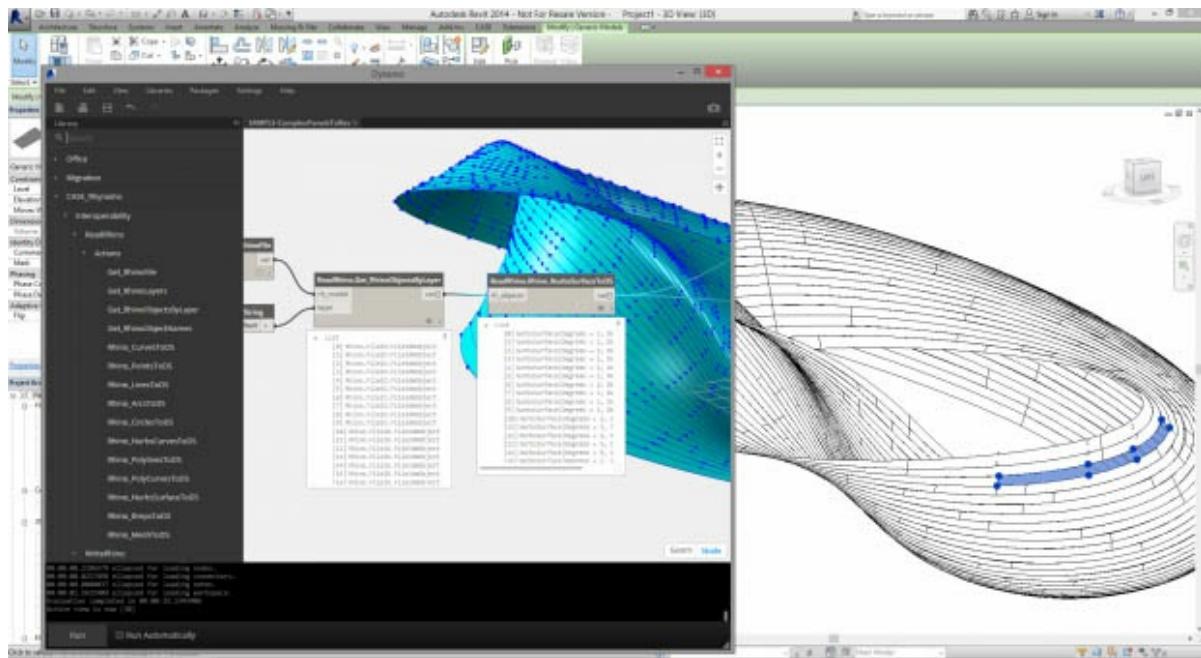
このセクションでは、Zero-Touch を使用してサードパーティのライブラリを読み込む方法について説明します。独自の Zero-Touch ライブリを開発する方法については、[Dynamo の Wiki ページ](#)を参照してください。

## Zero-Touch パッケージ

Zero-touch パッケージは、ユーザが定義するカスタム ノードを補完するパッケージです。次の図は、C# ライブリを使用するいくつかのパッケージを示しています。パッケージの詳細については、付録の「[パッケージ](#)」セクションを参照してください。

ロゴ/イメージ





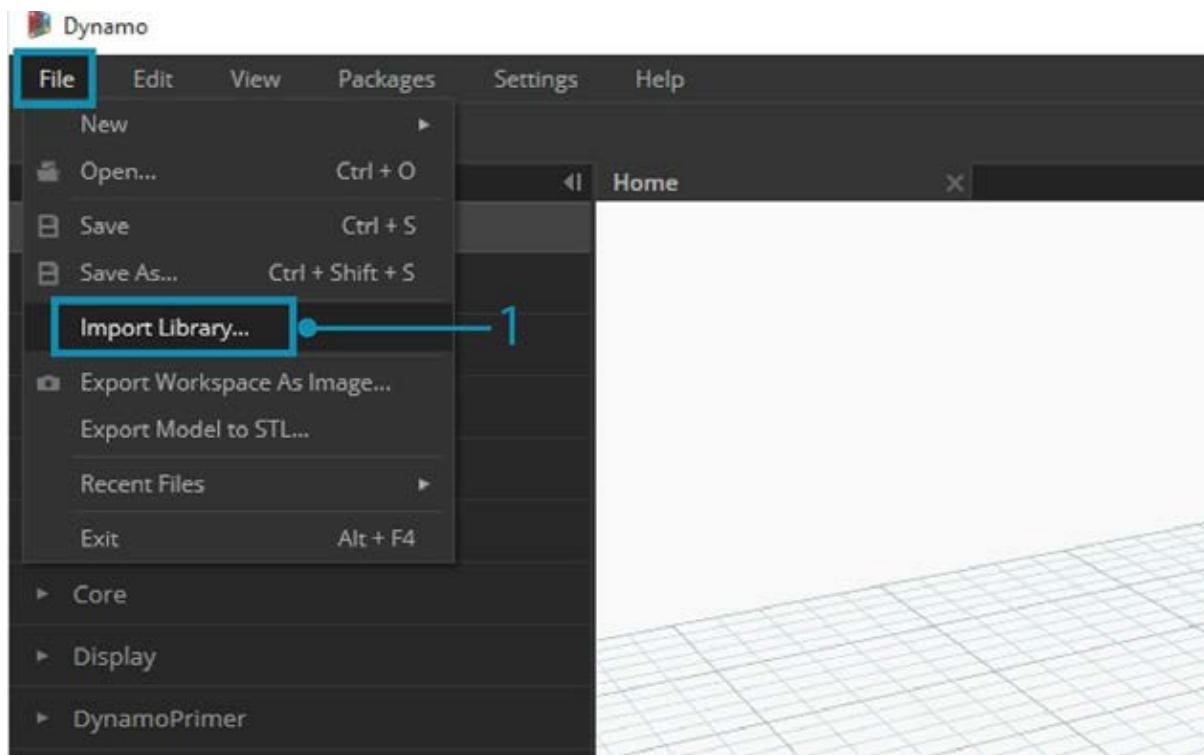
## ケーススタディ - AForge の読み込み

このケーススタディでは、[AForge](#) の外部 .dll ライブラリをインポートする方法について説明します。AForge は、イメージ処理機能から人工知能まで、さまざまな機能を提供する堅固なライブラリです。ここでは、AForge のイメージクラスを使用して、いくつかのイメージ処理を行う方法について説明します。

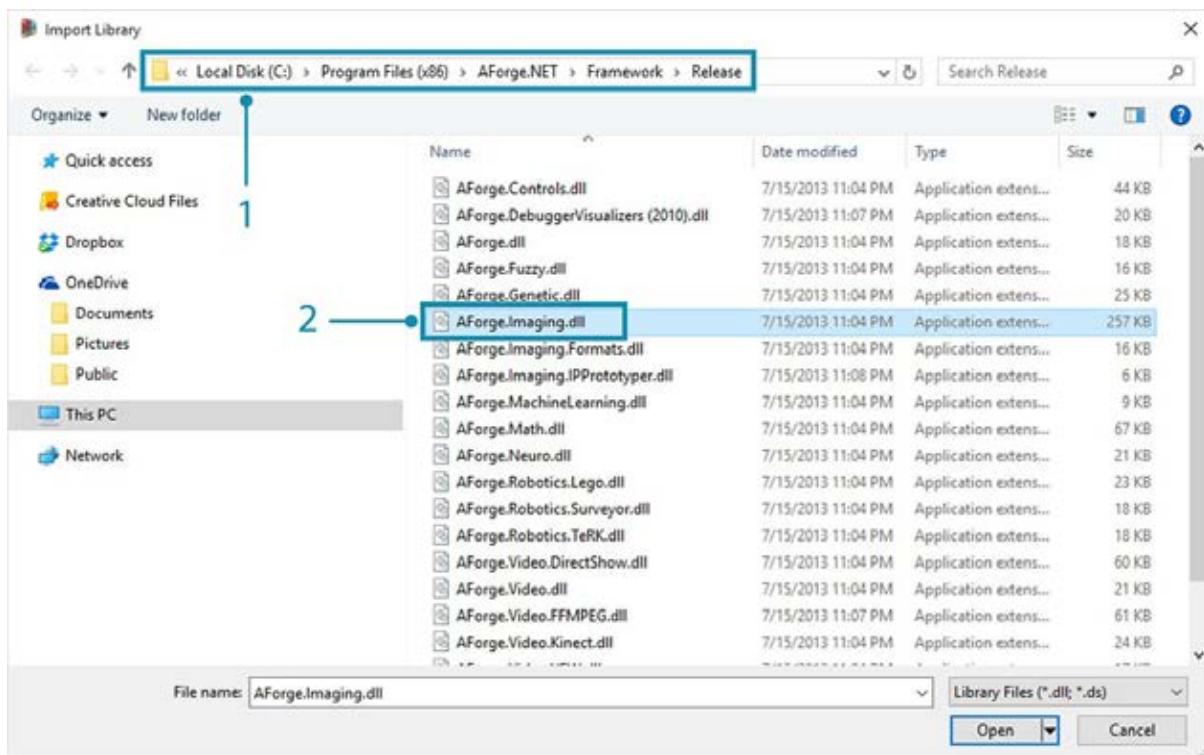
このパッケージのケーススタディに付属しているサンプルファイルをダウンロードして解凍してください(右クリックして[名前を付けてリンク先を保存...]を選択): Zero-Touch-Examples.zip。すべてのサンプルファイルの一覧については、付録を参照してください。[Zero-Touch-Examples.zip](#)

- 最初に、AForge をダウンロードします。[AForge のダウンロードページ](#)で[Download Installer]を選択し、ダウンロード

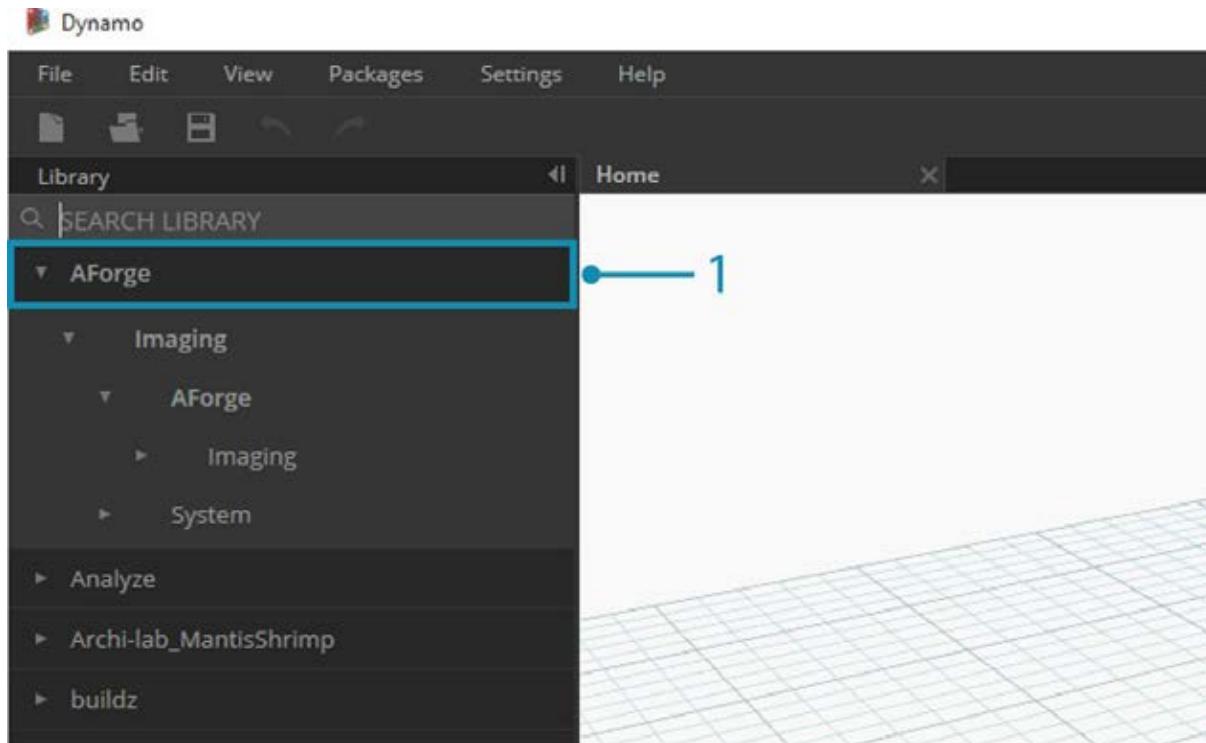
が完了したらインストールを実行します。



1. Dynamo で、新しいファイルを作成して[ファイル] > [ライブラリを読み込む...]を選択します。



1. ポップアップ ウィンドウで、AForge のインストール環境のリリース フォルダにナビゲートします。通常は、C:\Program Files (x86)\AForge.NET\Framework\Release などのフォルダになります。
2. このケース スタディでは、AForge.Imaging.dll だけを使用します。この .dll ファイルを選択して[開く]をクリックします。



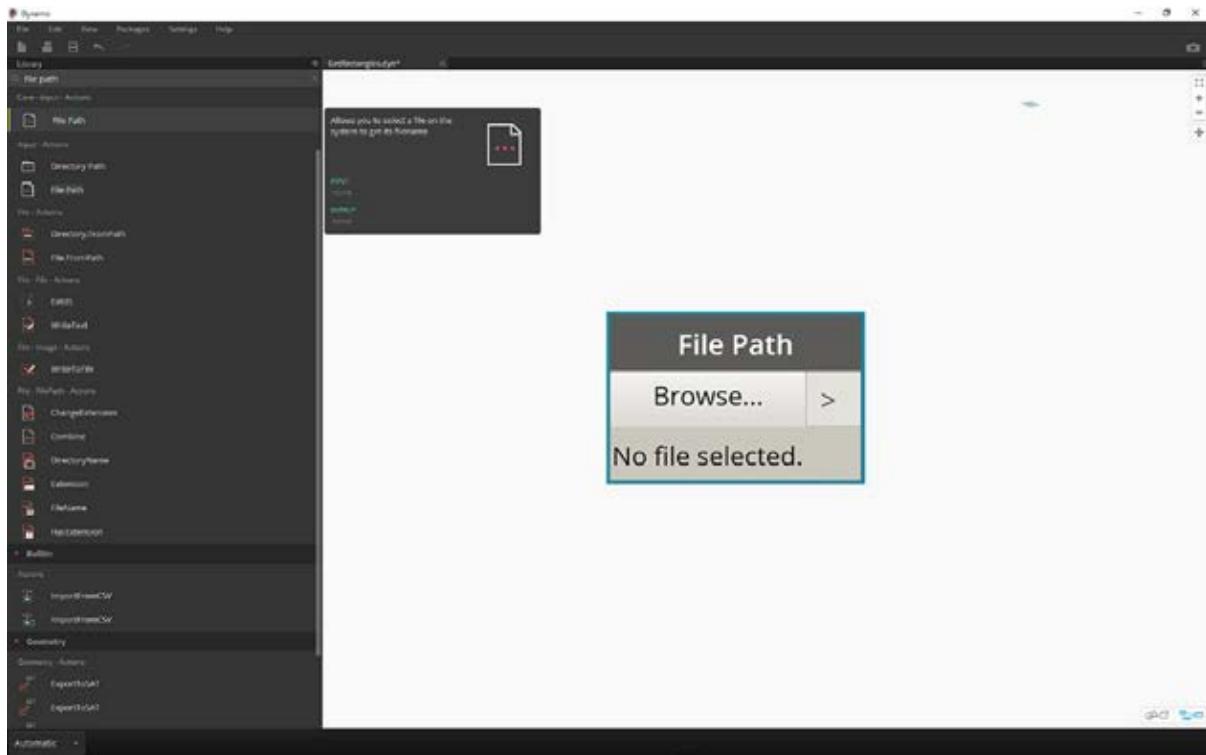
1. Dynamo に戻ると、ノードの[AForge]グループがライブラリのツールバーに新しく表示されます。これで、ビジュアル プログラミングから AForge のイメージ ライブラリにアクセスできるようになりました。

### 演習 1 - 輪郭線の検出

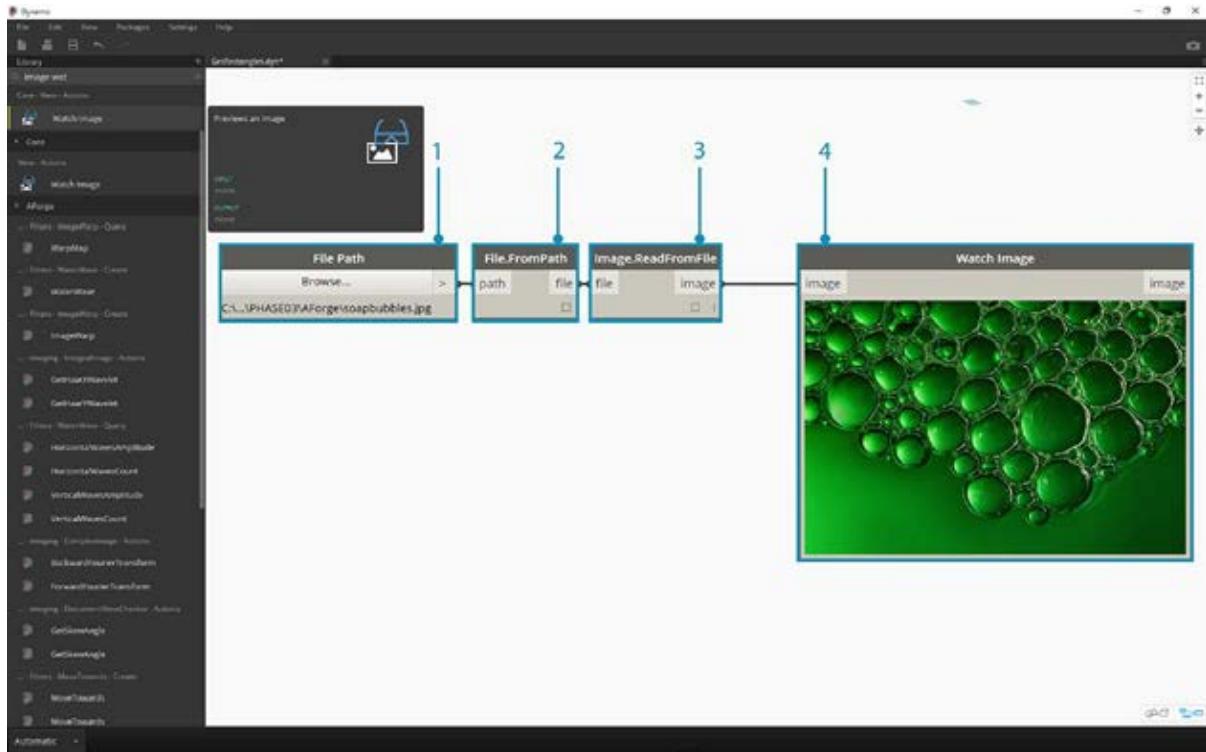
ライブラリを読み込んだら、最初に簡単な演習を行います。ここでは、サンプルのイメージに対して基本的なイメージ処理を実行することにより、AForge イメージのフィルタ機能について説明します。Watch Image ノードを使用して処理の実行結果を表示し、Dynamo のフィルタを適用します。Dynamo のフィルタは、Photoshop のフィルタに似ています。

このパッケージのケース スタディに付属しているサンプル ファイルをダウンロードして解凍してください(右クリックして[名前を付けてリンク先を保存...]を選択): Zero-Touch-Examples.zip。すべてのサンプル ファイルの一覧については、付録を参照してください。[ZeroTouchImages.zip](#)

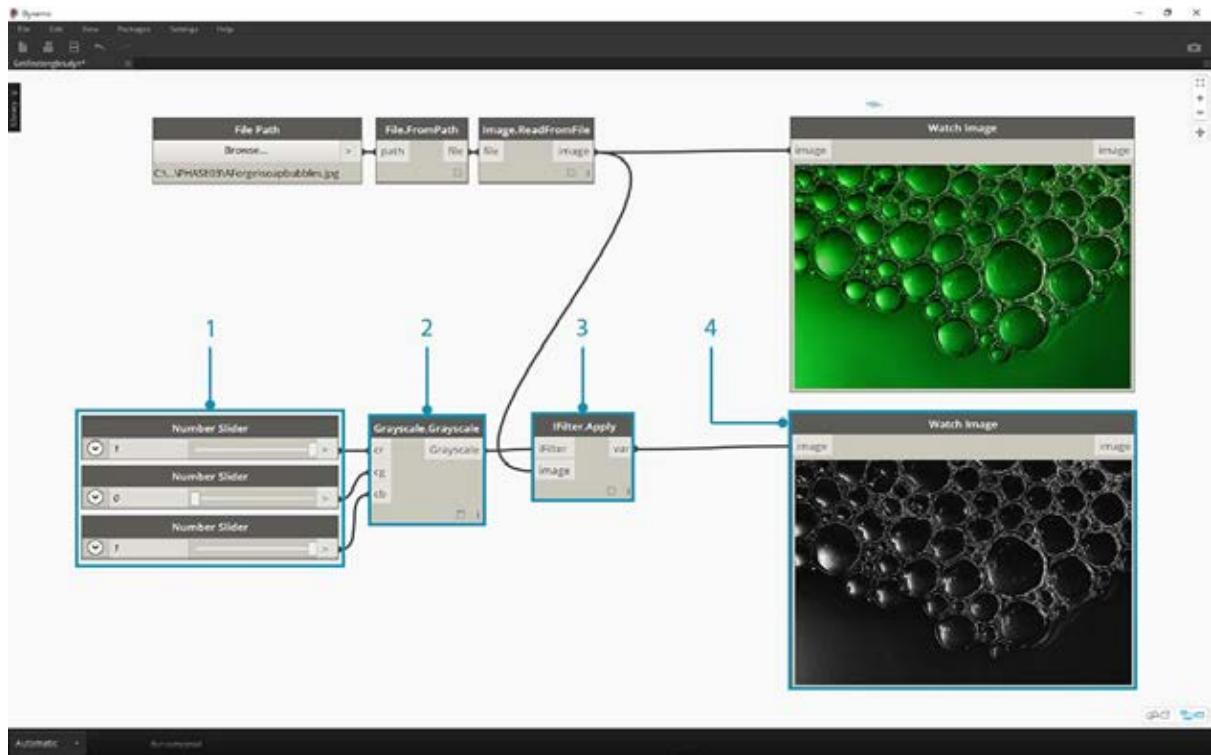
ライブラリを読み込んだら、最初に簡単な演習を行います(01-EdgeDetection.dyn)。ここでは、サンプルのイメージに対して基本的なイメージ処理を実行することにより、AForge イメージのフィルタ機能について説明します。Watch Image ノードを使用して処理の実行結果を表示し、Dynamo のフィルタを適用します。Dynamo のフィルタは、Photoshop のフィルタに似ています。



最初に、使用するイメージを読み込みます。File Path ノードをキャンバスに追加し、ダウンロードした演習フォルダで soapbubbles.jpg を選択します(次の画面で使用されている画像の出典元: [flickr](#))。

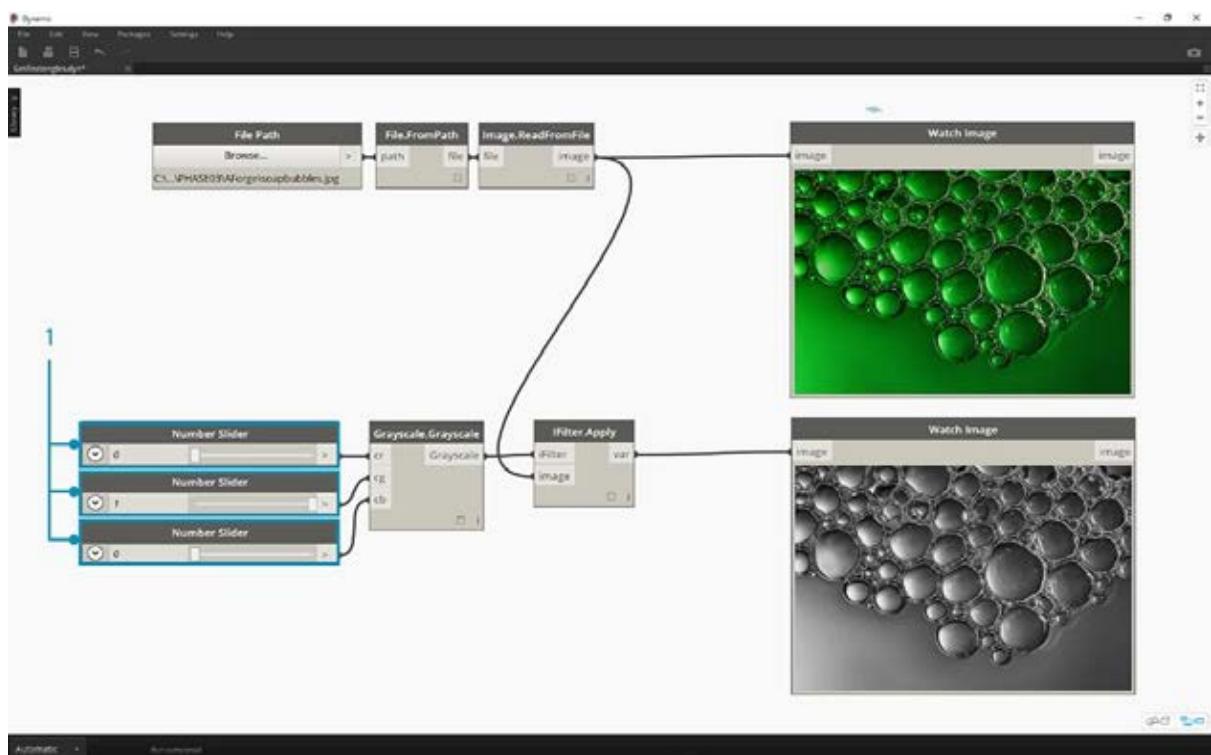


1. File Path ノードにより、ユーザが選択したイメージのパスの文字列が提供されます。この File Path ノードを、Dynamo 環境内のイメージに変換する必要があります。
2. File Path ノードを File.FromPath ノードに接続します。
3. Image.ReadFromFile ノードを使用して、file 出力をイメージに変換します。
4. 最後に、結果を確認します。Watch Image ノードをキャンバスにドロップして Image.ReadFromFile ノードに接続します。ここでは AForge をまだ使用していませんが、イメージを Dynamo 環境に正しく読み込むことができました。



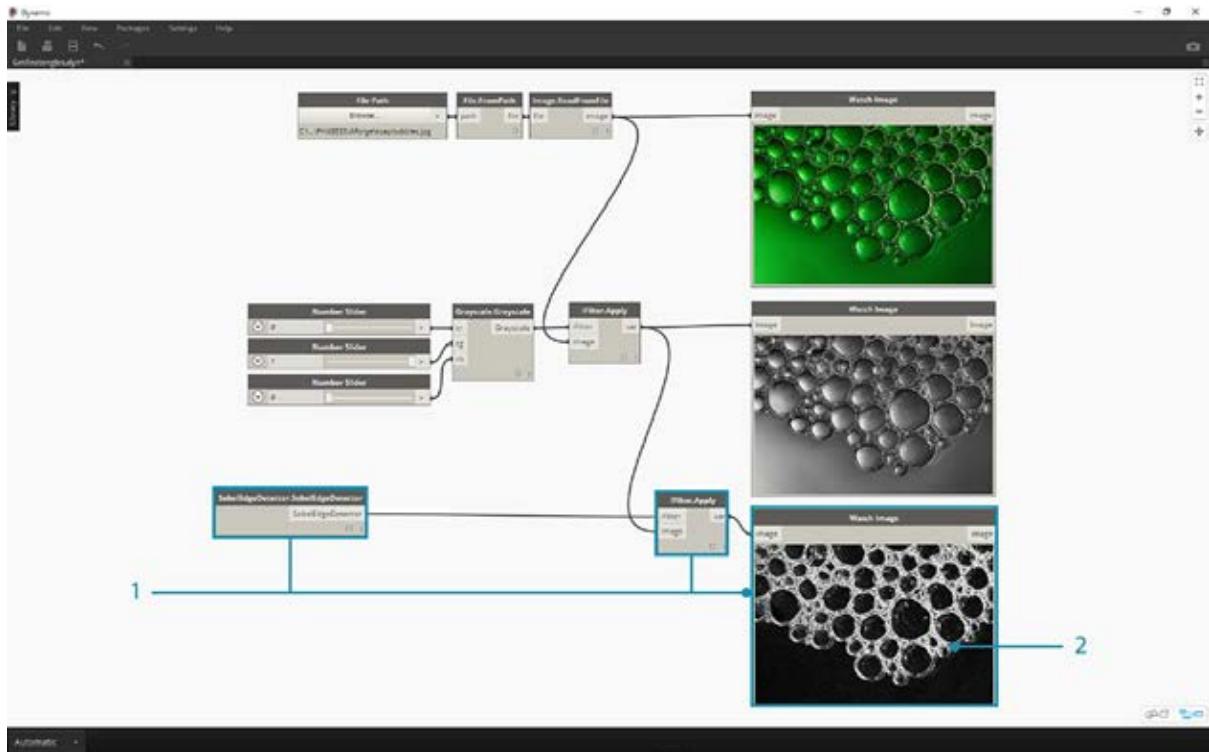
ナビゲーションメニューの AForge.Imaging.AForge.Filters の下に、さまざまな使用可能なフィルタが表示されます。ここでは、1つのフィルタを使用し、しきい値に基づいてイメージの彩度を下げてみましょう。

1. 3つのスライダをキャンバスにドロップし、各スライダの範囲を 0 から 1 に変更して、ステップ値を 0.01 に変更します。
2. キャンバスに Grayscale.Grayscale ノードを追加します。これは、グレースケール フィルタをイメージに適用する AForge フィルタです。手順 1 の 3つのスライダを、Grayscale.Grayscale ノードの入力(cr, cg, cb)にそれぞれ接続します。1番目と3番目のスライダの値を 1 に設定し、2番目のスライダの値を 0 に設定します。
3. グレースケール フィルタを適用するには、イメージに対してアクションを実行する必要があります。そのためには、IFilter.Apply ノードを使用します。Image.ReadFromFile ノードの image 出力を IFilter.Apply ノードの image 入力に接続し、Grayscale.Grayscale ノードを IFilter.Apply ノードの iFilter 入力に接続します。
4. 最後に、IFilter.Apply ノードを Watch Image ノードに接続します。これで、イメージの彩度が下がります。



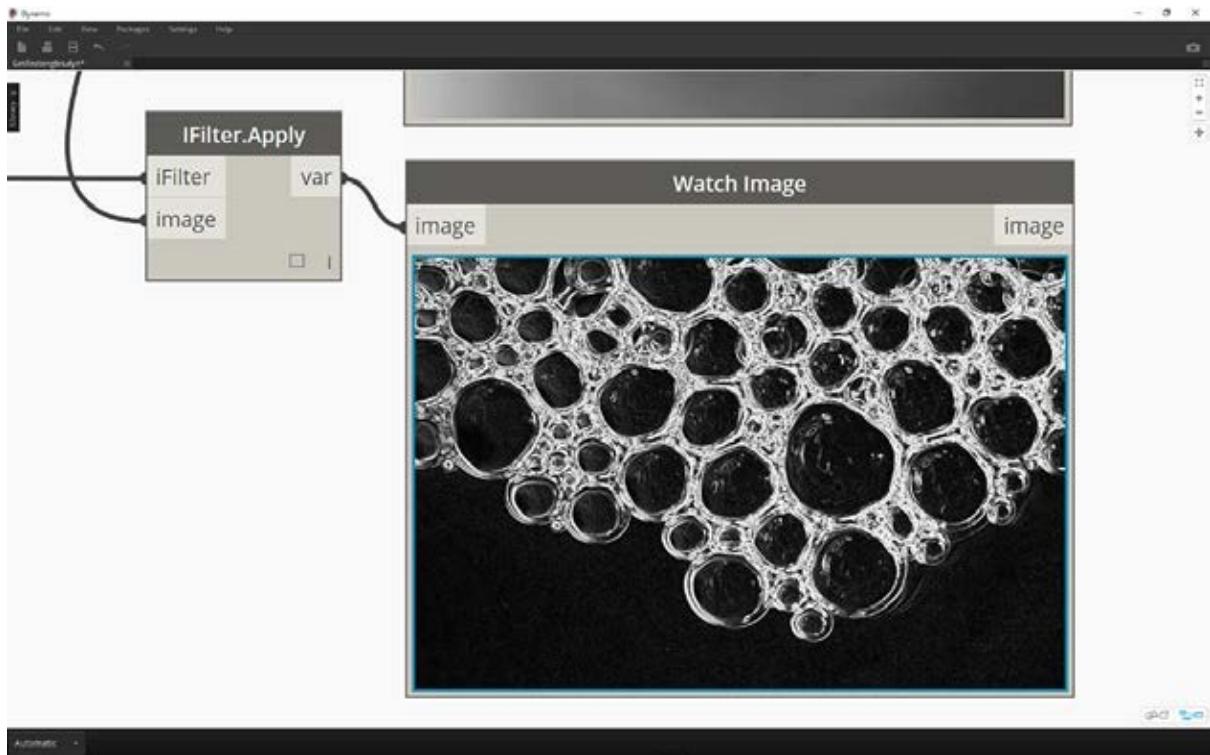
赤、緑、青のしきい値に基づいて、イメージの彩度を下げる方法をコントロールすることができます。これらのしきい値は、Grayscale.Grayscale ノードに対する入力によって定義されます。上図のイメージは暗くなっていますが、これは、スライダで緑のしきい値が 0 に設定されているためです。

1. 次に、1 番目と 3 番目のスライダの値を 0 に設定し、2 番目のスライダの値を 1 に設定します。この設定により、彩度を下げたイメージが明確に表示されます。



次に、彩度を下げたイメージに対して別のフィルタを適用します。彩度が低いイメージにはいくらかのコントラストがあります。ここでは、輪郭線の検出をテストしてみましょう。

1. キャンバスに SobelEdgeDetector.SobelEdgeDetector ノードを追加します。このノードを新しい IFilter.Apply ノードの iFilter 入力に接続し、彩度を下げたイメージを IFilter.Apply ノードの image 入力に接続します。
2. Sobel Edge Detector により、新しいイメージ内で輪郭線がハイライト表示されます。

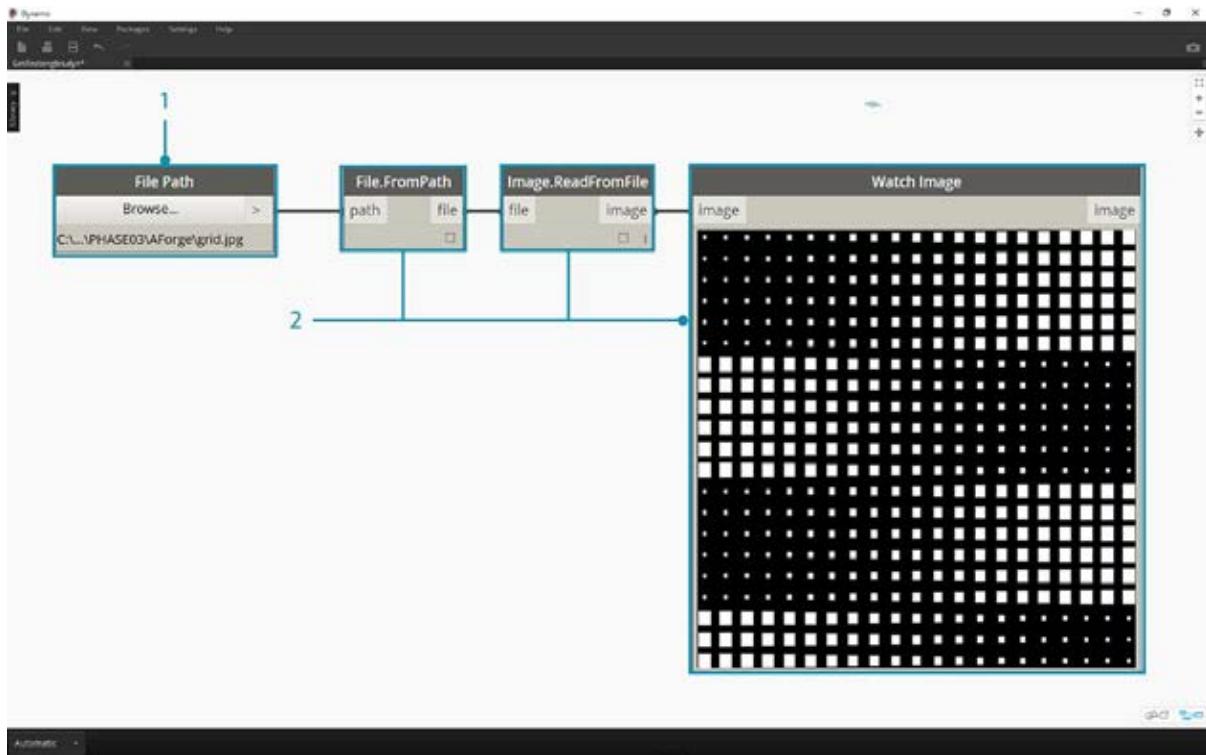


拡大表示すると、Edge Detector により、イメージ内の泡の輪郭がピクセル単位で描画されていることがわかります。AForge ライブリには、このような処理を行うためのツールや、Dynamo のジオメトリを作成するためのツールが用意されています。Dynamo のジオメトリを作成する方法については、次の演習で説明します。

## 演習 2 - 長方形の作成

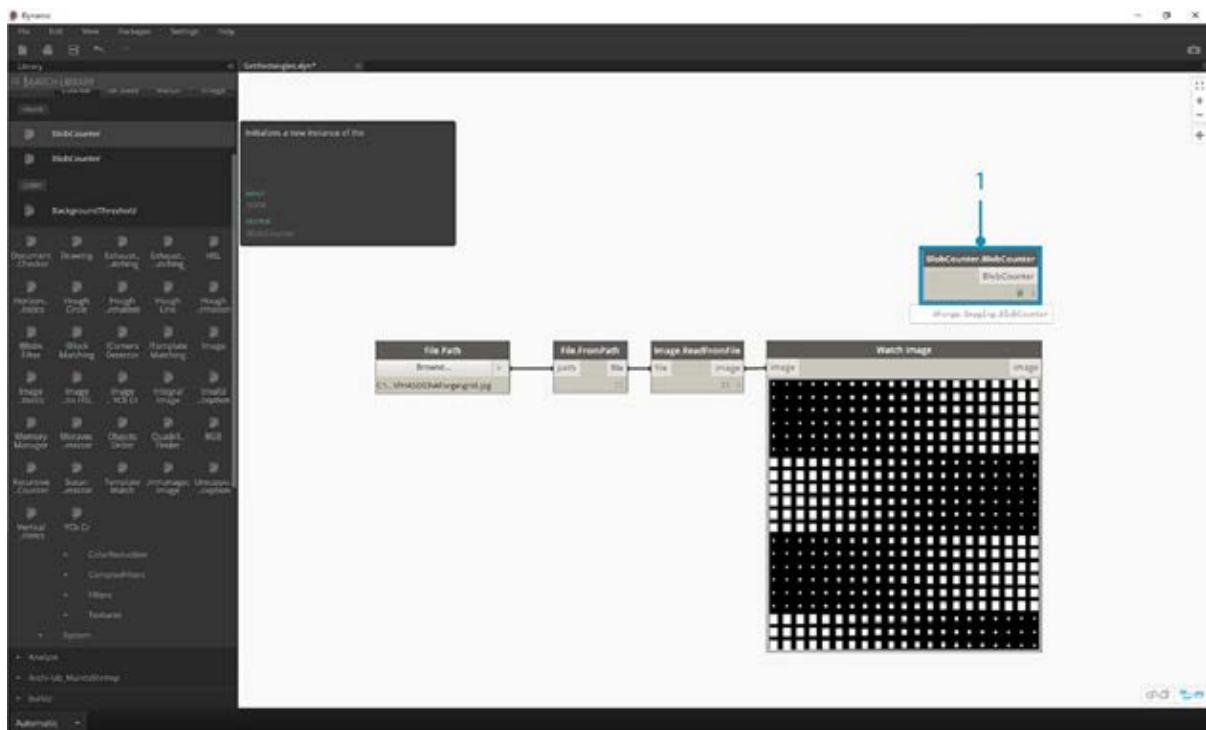
このパッケージのケーススタディに付属しているサンプルファイルをダウンロードして解凍してください(右クリックして[名前を付けてリンク先を保存...]を選択): Zero-Touch-Examples.zip。すべてのサンプルファイルの一覧については、付録を参照してください。[ZeroTouchImages.zip](#)

前の演習では、基本的なイメージ処理について確認しました。この演習では、イメージを使用して Dynamo のジオメトリを操作してみましょう。簡単な操作として、AForge と Dynamo を使用してイメージの「ライブトレース」を実行します。説明を簡単にするために、ここでは参照イメージから長方形を抽出しますが、AForge には、より複雑な操作を実行するための各種ツールが用意されています。この演習では、ダウンロードした演習ファイルの 02-RectangleCreation.dyn を使用します。

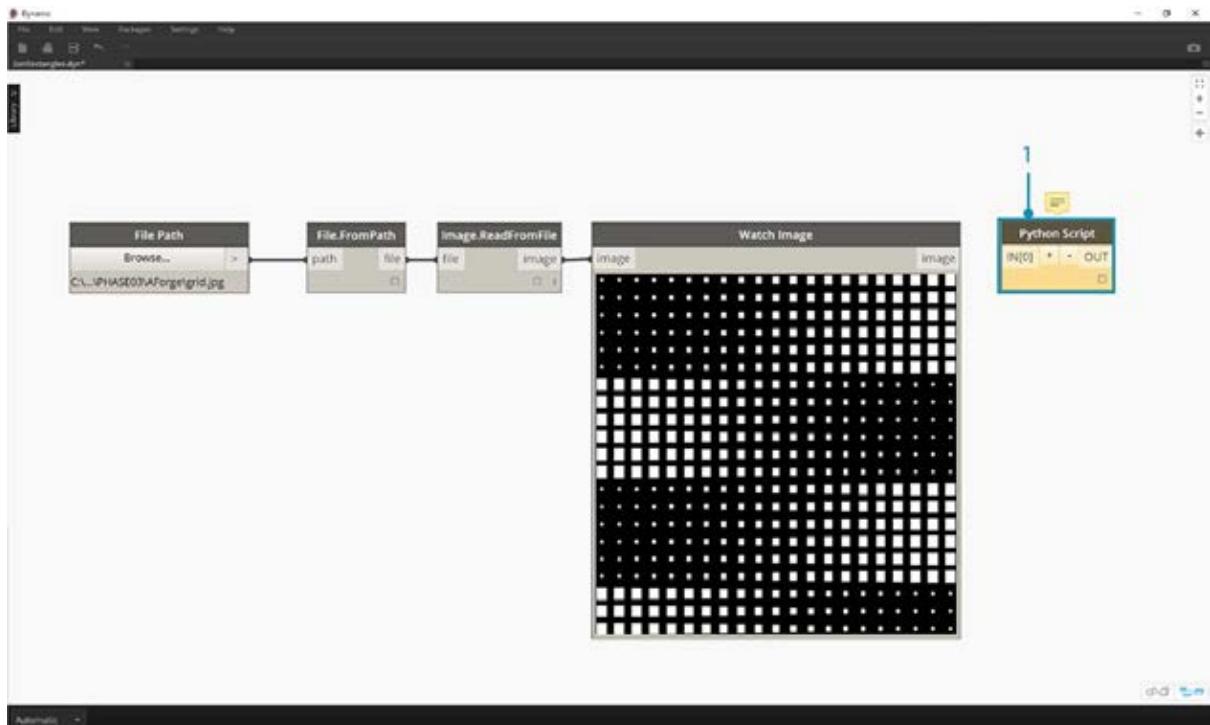


1. File Path ノードを使用して、演習フォルダの grid.jpg にナビゲートします。
2. 残りの一連のノードを接続し、パラメトリック グリッドを表示します。

次の手順では、イメージ内の白い正方形を表示して、実際の Dynamo ジオメトリに変換します。AForge には、多くの便利な Computer Vision ツールが用意されていますが、ここでは、その中でも特に重要なツールを [BlobCounter](#) というライブラリに対して使用します。



1. BlobCounter をキャンバスに追加したら、前の演習で使用した IFilter ツールと同じような方法でイメージを処理する必要があります。しかし、Process Image ノードは Dynamo ライブラリ内に直接表示されません。これは、このノード(閉数)を AForge のソースコード内で表示することができないためです。この問題を解決するには、何らかの回避策が必要です。

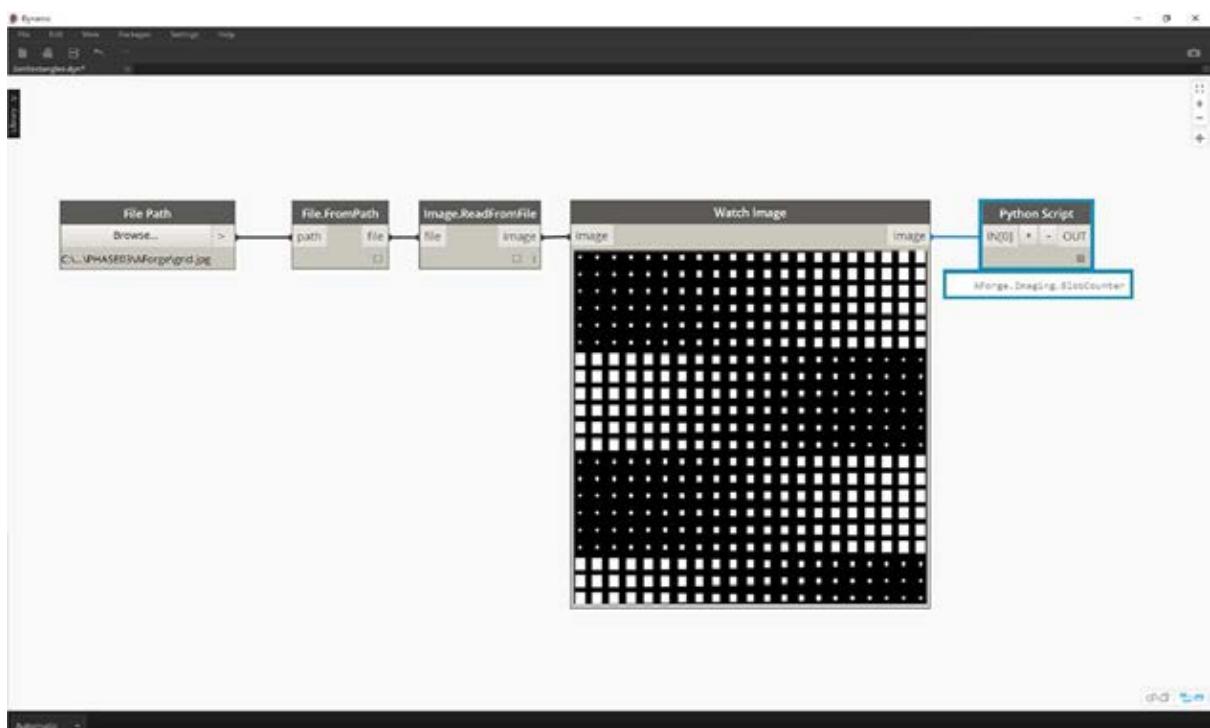


1. キャンバスに Python Script ノードを追加します。

```
import clr
clr.AddReference('AForge.Imaging')
from AForge.Imaging import *

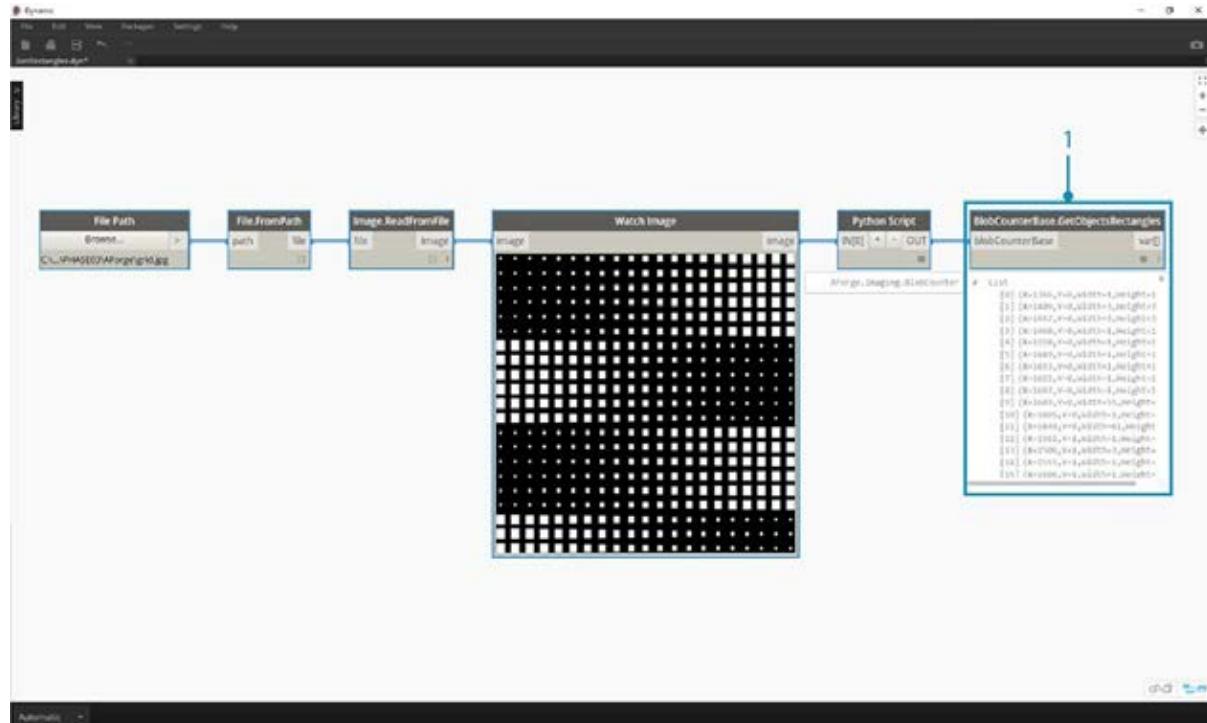
bc= BlobCounter()
bc.ProcessImage(IN[0])
OUT=bc
```

上記のコードを Python Script ノードに追加します。このコードによって AForge ライブラリが読み込まれ、読み込まれたイメージが処理されます。

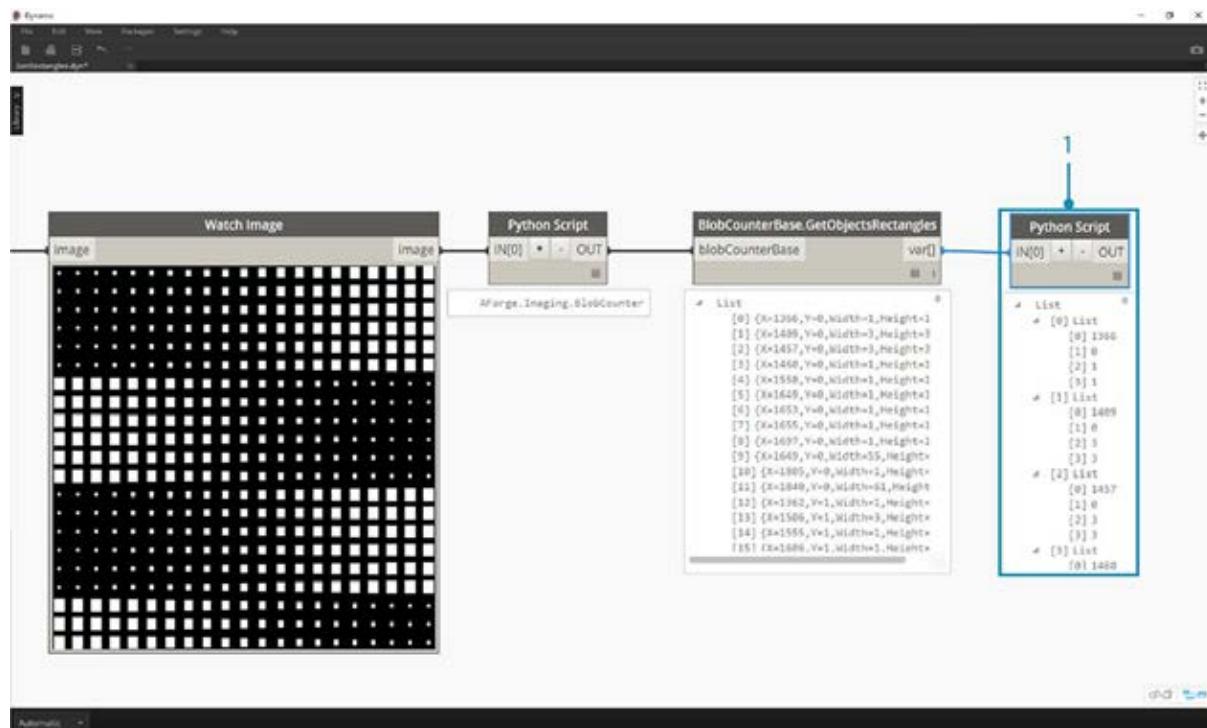


Watch Image ノードの image 出力を Python Script ノードの入力に接続すると、Python Script ノードによって AForge.Imaging.BlobCounter の結果が取得されます。

次の手順では、[AForge Imaging API](#) を使用する場合のヒントとなる操作をいくつか紹介します。ただし、Dynamo で作業を行う場合に、すべての操作を理解する必要はありません。これらの操作を参照して、外部のライブラリを Dynamo 環境内で柔軟に使用できることを理解するのが目的です。



1. Python Script ノードの出力を BlobCounterBase.GetObjectRectangles に接続します。この操作により、しきい値に基づいてイメージ内のオブジェクトが読み取られ、定量化された長方形がピクセル スペースから抽出されます。

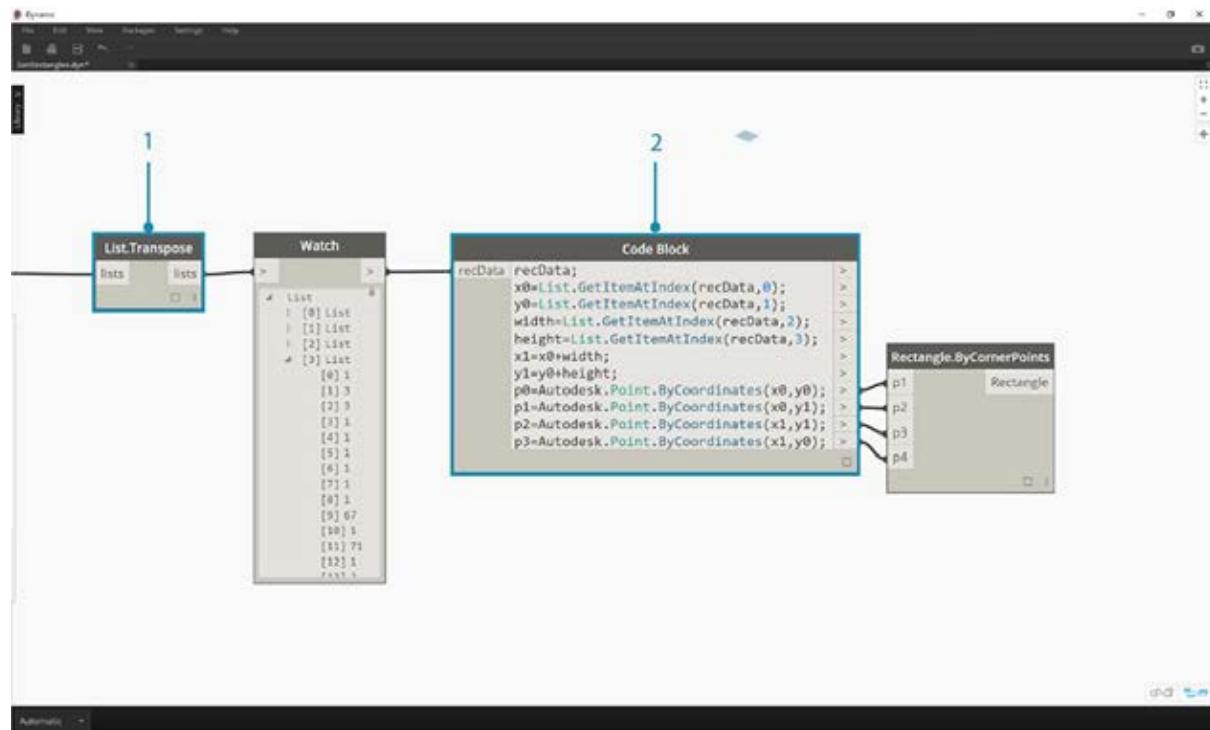


1. 別の Python Script ノードをキャンバスに追加して GetObjectRectangles に接続し、次のコードを入力します。この操作により、Dynamo オブジェクトの整理されたリストが作成されます。

```

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)

```

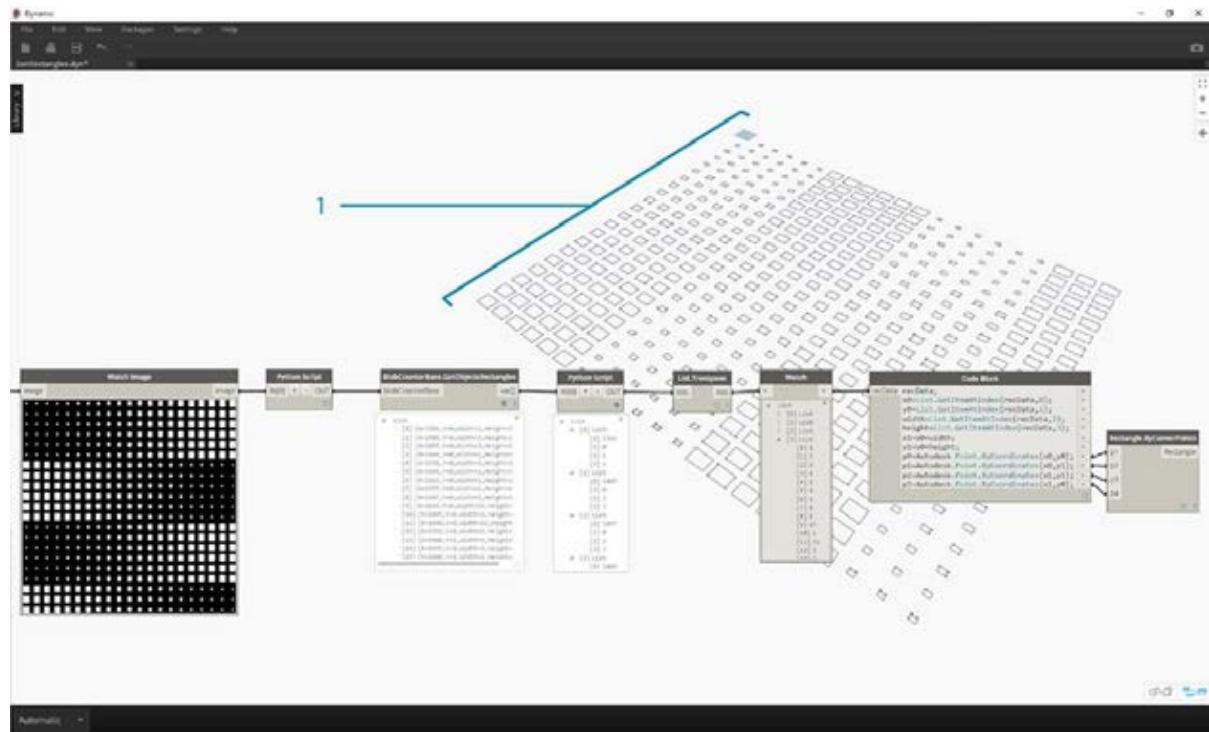


- 前の手順で追加した Python Script ノードの出力を置き換えます。この操作により、各長方形の X 座標、Y 座標、幅、高さを表す 4 つのリストが作成されます。
- Code Block ノードを使用して、Rectangle.ByCornerPoints ノードに対応する構造にデータを編成します(次のコードを参照)。

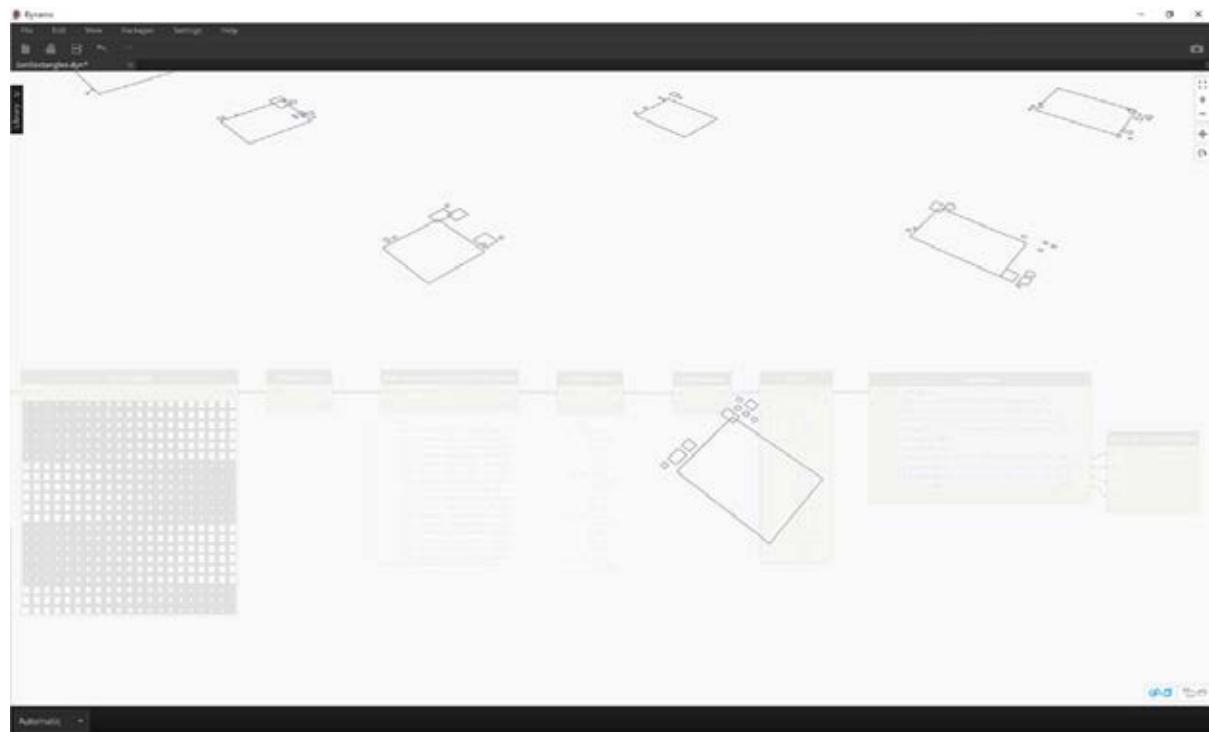
```

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);

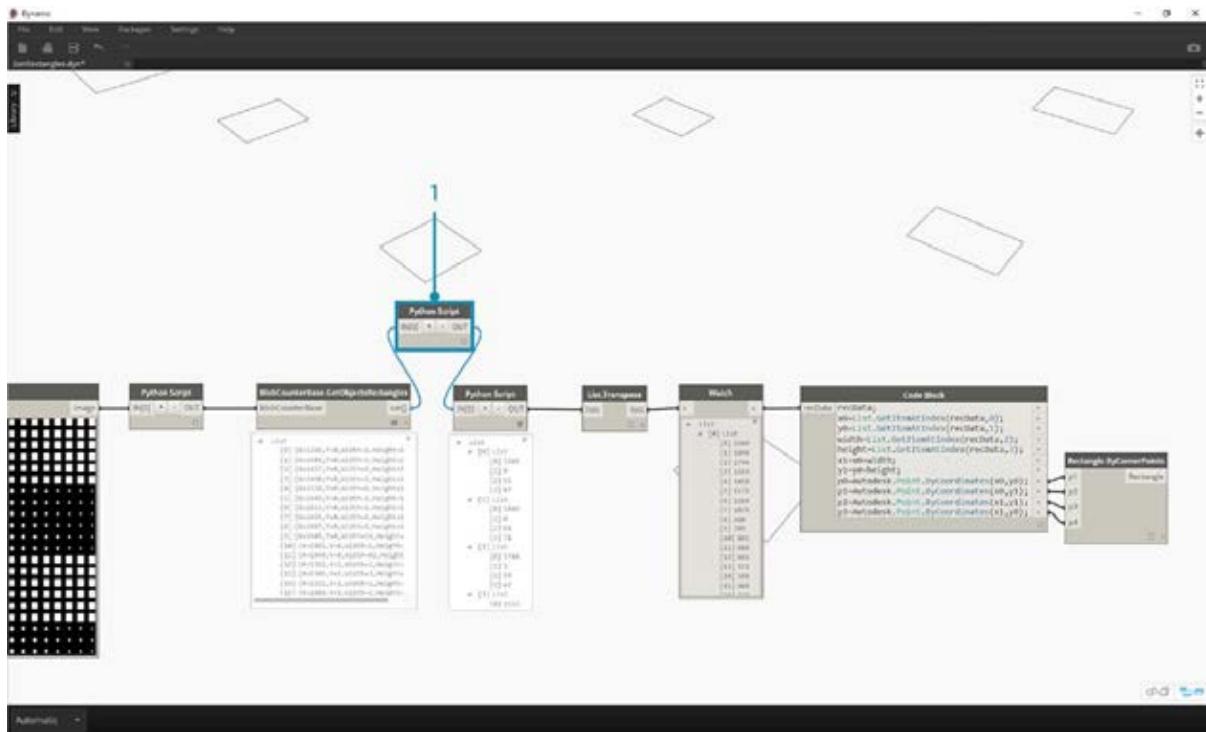
```



縮小表示すると、イメージ内の白い正方形を表す長方形の配列が表示されます。この演習では、プログラミングコードを入力して、Illustrator のライブトレースに類似する機能を作成しました。

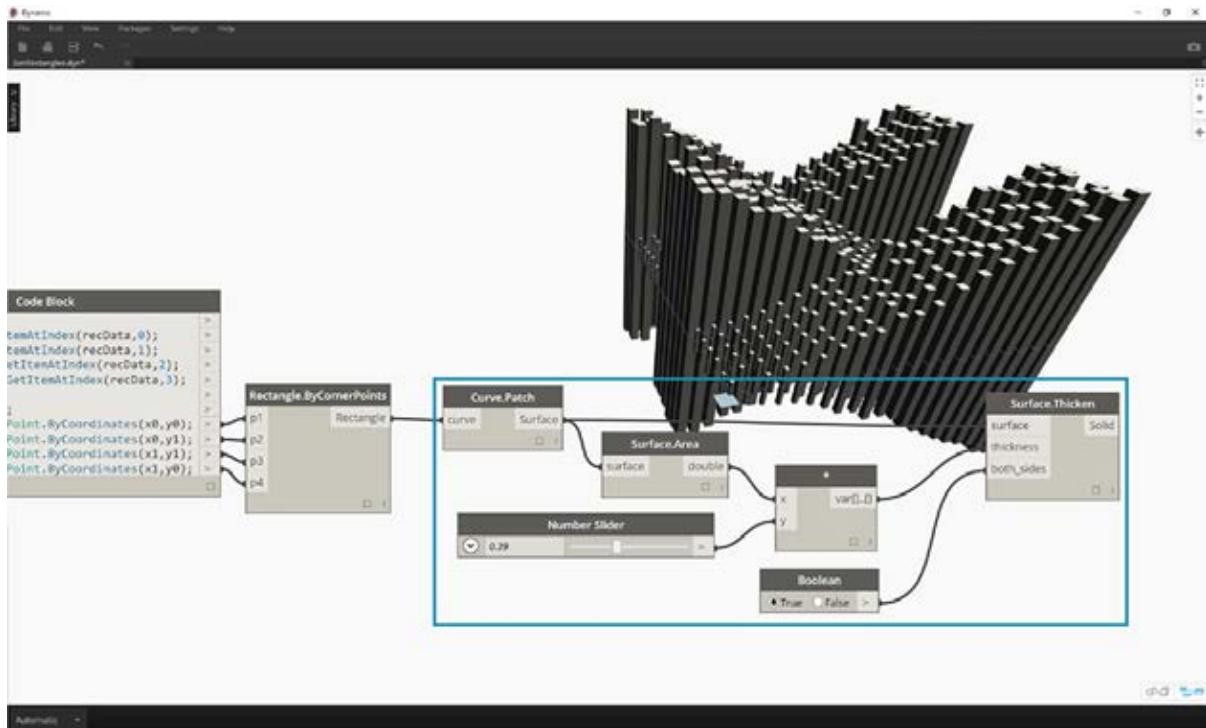


ただし、まだクリーンアップが必要です。拡大表示すると、必要のない小さな長方形がたくさん残っていることがわかります。

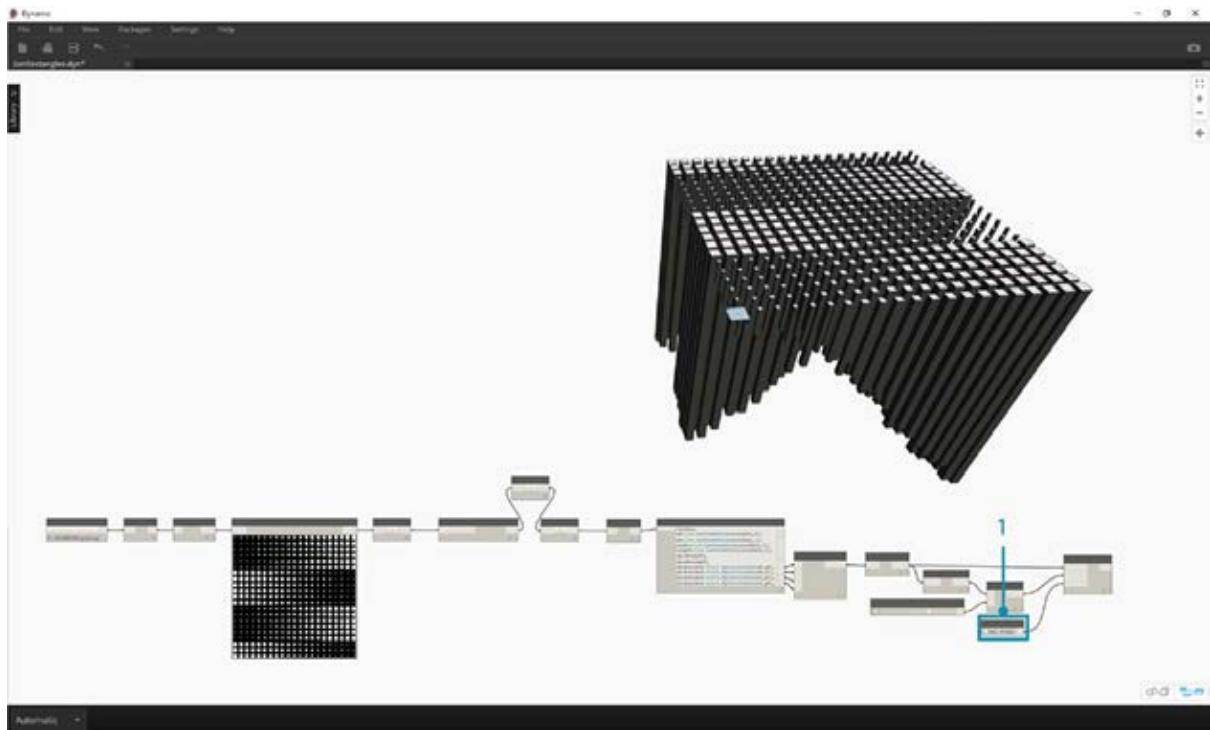


- GetObjectRectangles ノードと Python Script ノードの間に新しい Python Script ノードを挿入して、不要な長方形を削除します。このノードのコードは次のとおりです。このコードで指定したサイズよりも小さな長方形がすべて削除されます。

```
rectangles=IN[0]
OUT=[]
for rec in rectangles:
if rec.Width>8 and rec.Height>8:
OUT.append(rec)
```



不要な長方形を削除したら、それらの長方形からサーフェスを作成し、その長方形の面積に基づく距離を使用して長方形を押し出してみましょう。



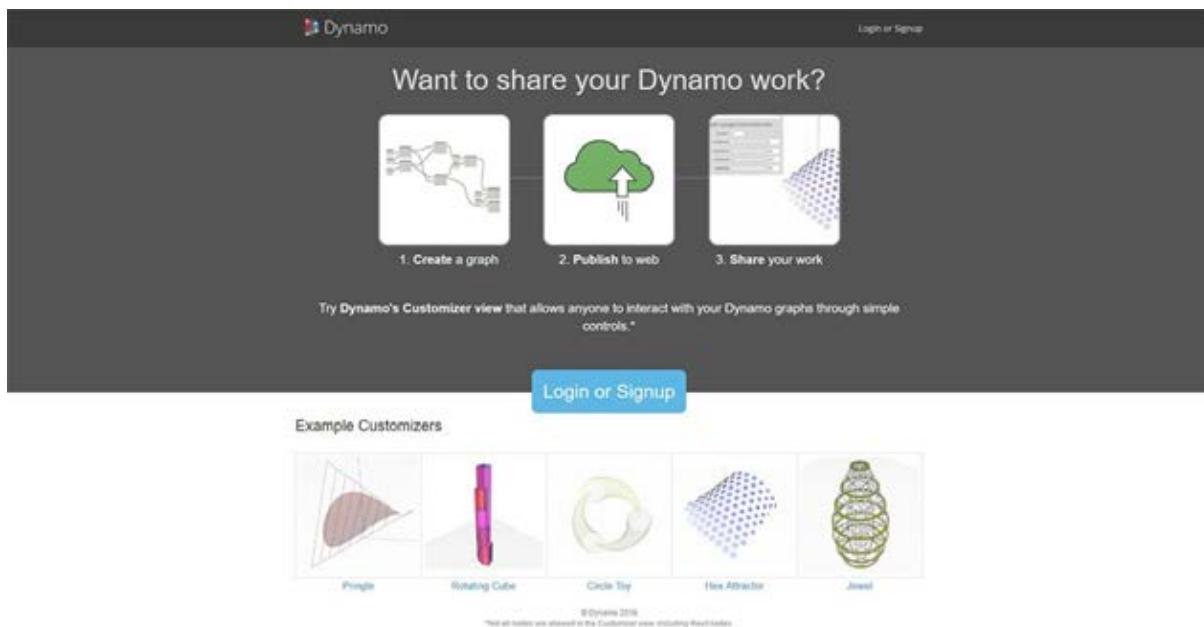
1. both\_sides 入力パラメータを false に変更すると、長方形が一方向に押し出されます。この操作により、非常に面白いテーブルが作成されます。

ここで紹介した例はどれも基本的なものばかりですが、その概念は実際のアプリケーションでも応用することができます。Computer Vision は、さまざまなプロセスで使用することができます。バーコードリーダー、パースペクティブ マッチング、[プロジェクトンマッピング](#)、[オーグメンテッドリアリティ](#)などはその一例です。この演習に関する AForge の詳細なトピックについては、[この記事](#)を参照してください。

## Dynamo の Web エクスペリエンス

### Dynamo の Web エクスペリエンス

Dynamo Studio の「Web に送信」機能により、Dynamo を Web 上で使用できるようになりました。「Web に送信」機能により、他のユーザがシンプルなインターフェースである[カスタマイザ]ビューで各種スクリプトを使用することができます。[カスタマイザ]ビューは、各種スライダ、数値、布尔値など、有効な入力によって構成されています。これにより、Dynamo やビジュアル プログラミングに慣れていないユーザも、スクリプトを使用することができます。



Web 上での Dynamo

# Web に送信(Dynamo Studio を使用)

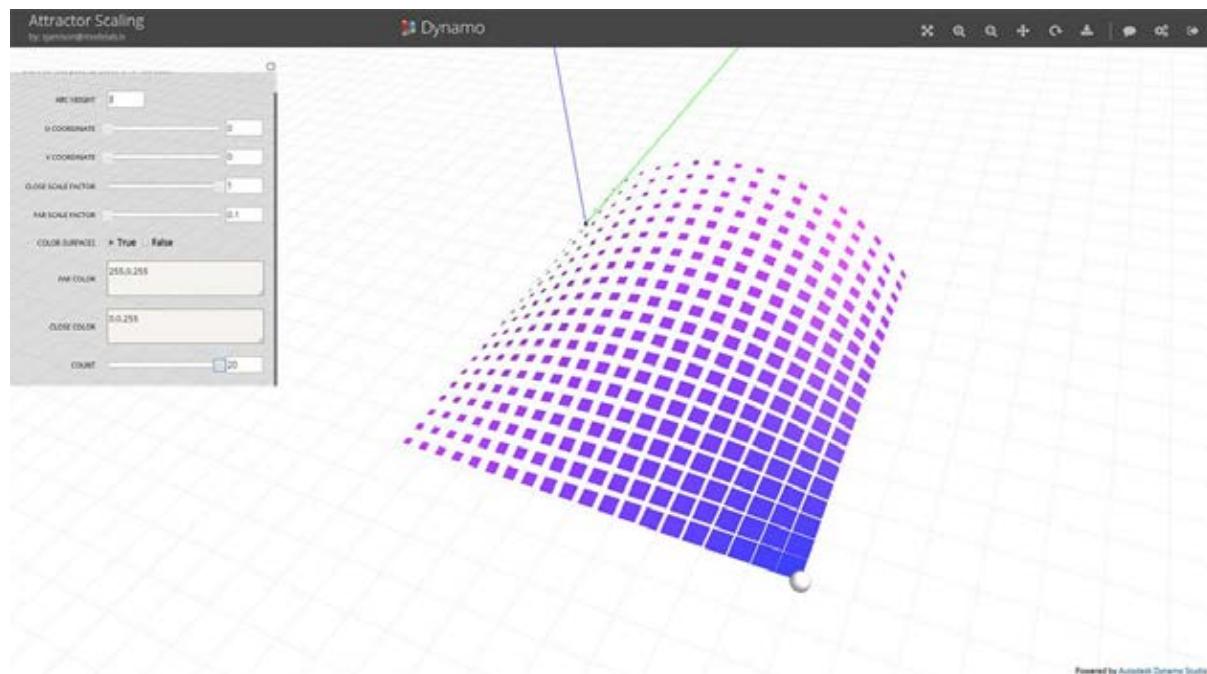
## Web に送信(Dynamo Studio を使用)

Dynamo Studio では、すばやく簡単にファイルを Web にパブリッシュすることができます。ただし、入力の選択とラベル付けには時間をかけることをお勧めします。また、わかりやすいファイル名を付けてください。Dynamo Studio をダウンロードする場合は、[Autodesk の Web サイト](#)で詳細情報を参照してください。

演習:「Web に送信」機能の準備を行う

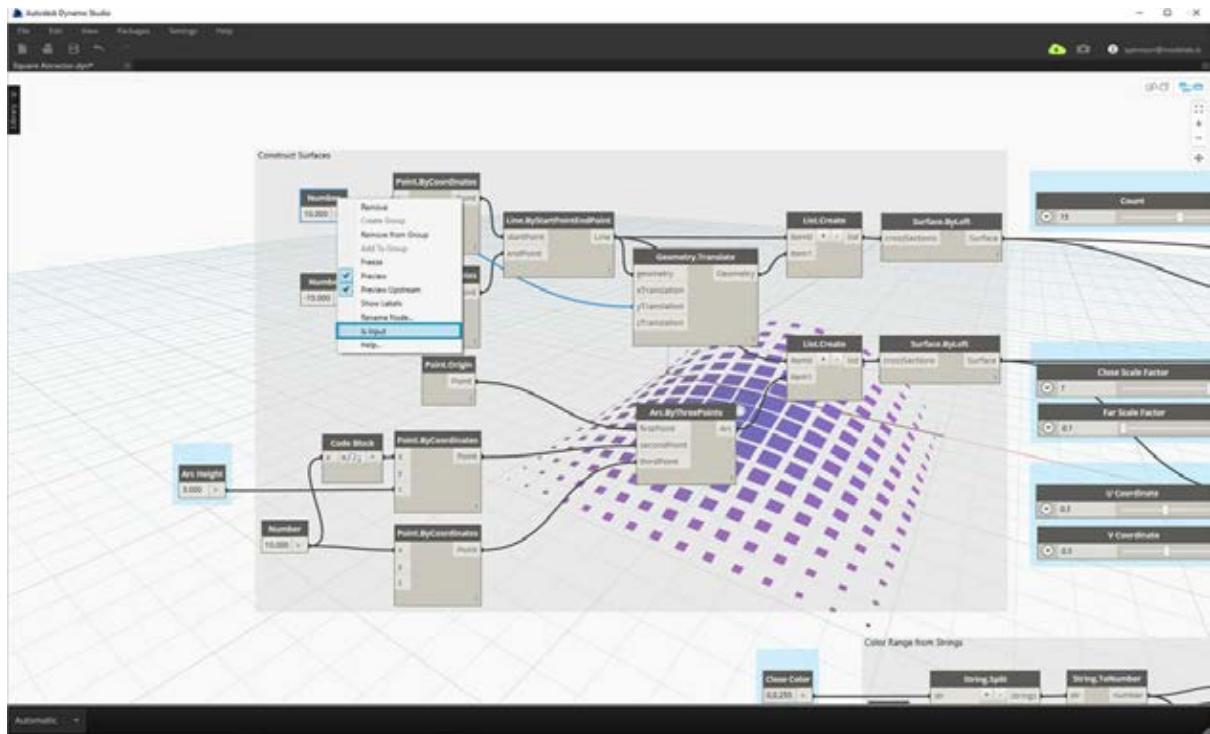
この演習用のサンプル ファイルをダウンロードしてください(右クリックして[名前を付けてリンク先を保存])。すべてのサンプル ファイルの一覧については、付録を参照してください。[アトラクタ スケールのサンプルをダウンロード](#)

この演習では、Dynamo のグラフを Web にパブリッシュします。このファイルにより、長方形のグリッドが作成されます。このグリッドは、アトラクタに基づいてスケールを調整され、基準点からターゲットのサーフェスにマップされます。サーフェスのパッチは各長方形から作成され、アトラクタからの距離に基づいて色が付けられます。



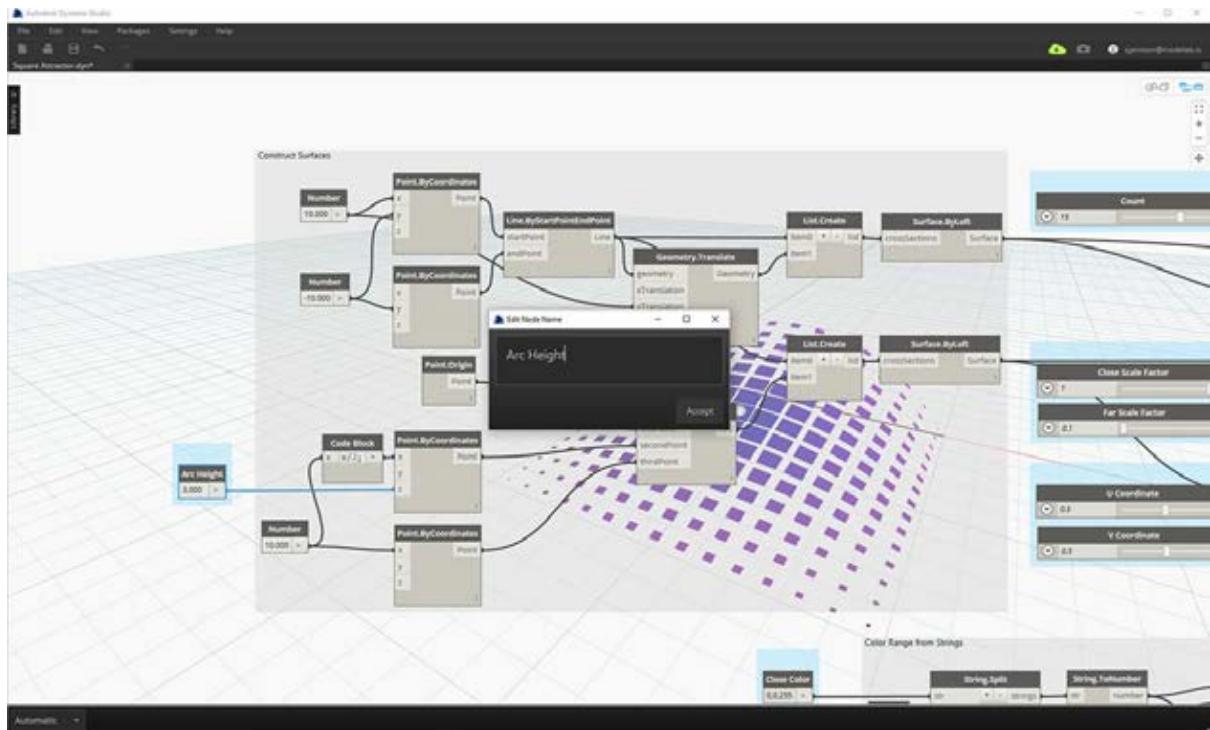
この演習では、上記のカスタマイザを作成します。このサンプルは、[Web](#) で確認できます。

スクリプトのパブリッシュを準備するには、ユーザにアクセスできる入力を最初に決定します。使用可能な入力は、スライダ、数値、文字列、布尔値です。コード ブロックとファイル パスを入力として使用することはできません。[カスタマイザ]ビューに表示したくない入力がある場合は、その入力のコンテキスト メニューで[入力]オプションを無効にします。スライダの入力では、適切な最小値と最大値が設定されていることを確認してください。



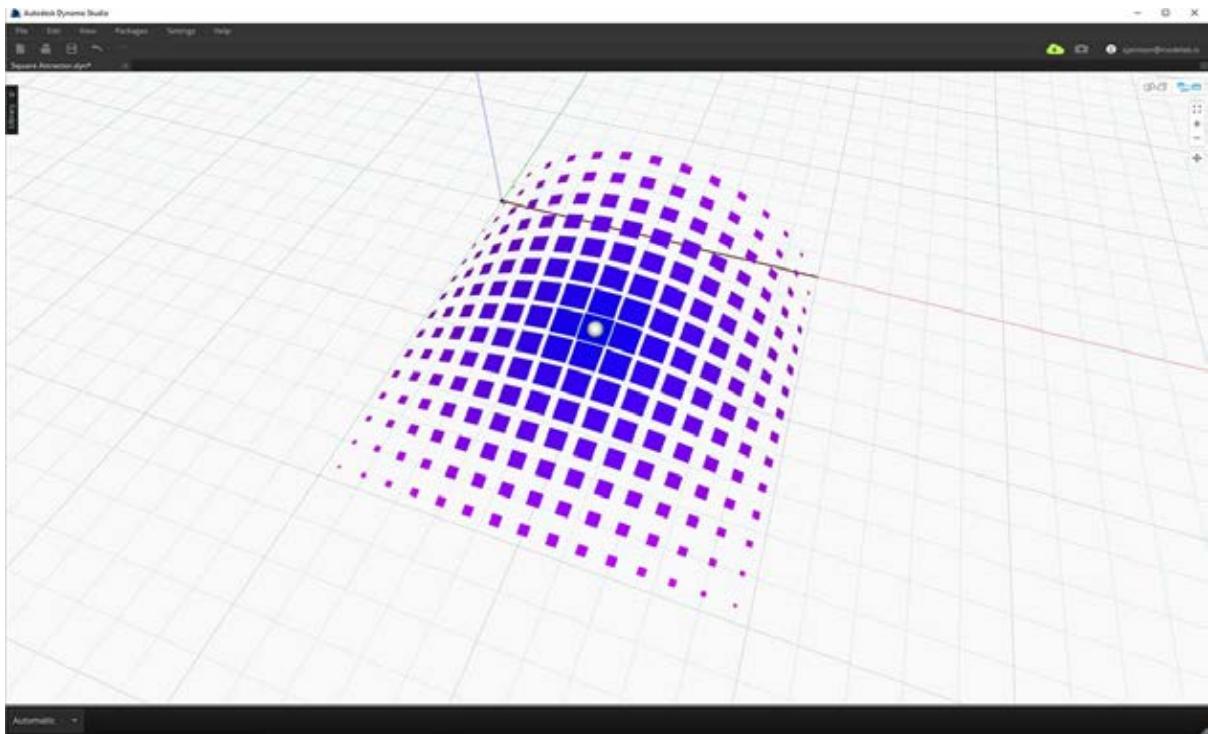
[カスタマイザ]ビューに表示たくない入力がある場合は、その入力のコンテキストメニューで[入力]オプションの選択を解除します。

次に、各入力にわかりやすいラベルを付けます。



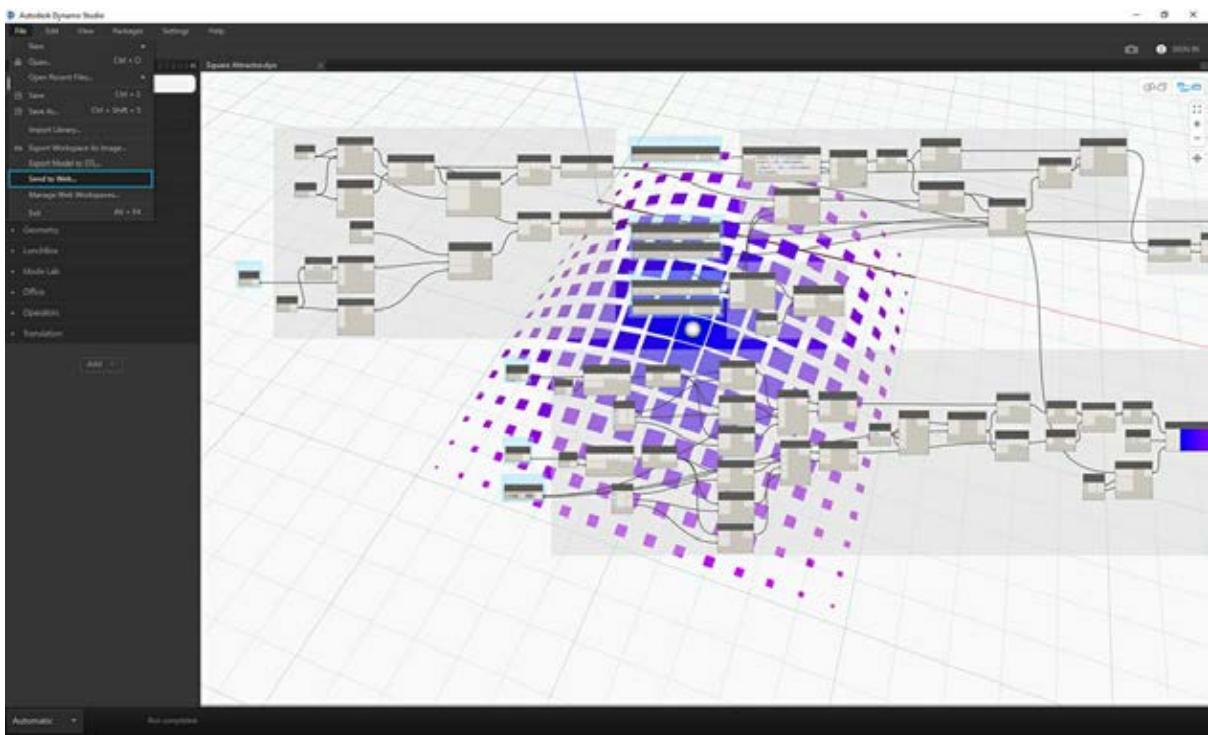
入力にラベルを付けるには、編集するノード名をダブルクリックします。

スクリプトの内容を簡単に把握できるプレビュー ジオメトリを含めます。この例では、球体がアトラクタの位置を示し、アトラクタまでの距離に基づいてサーフェスに色が付けられています。これにより、アトラクタの影響を簡単に視覚化して理解することができます。

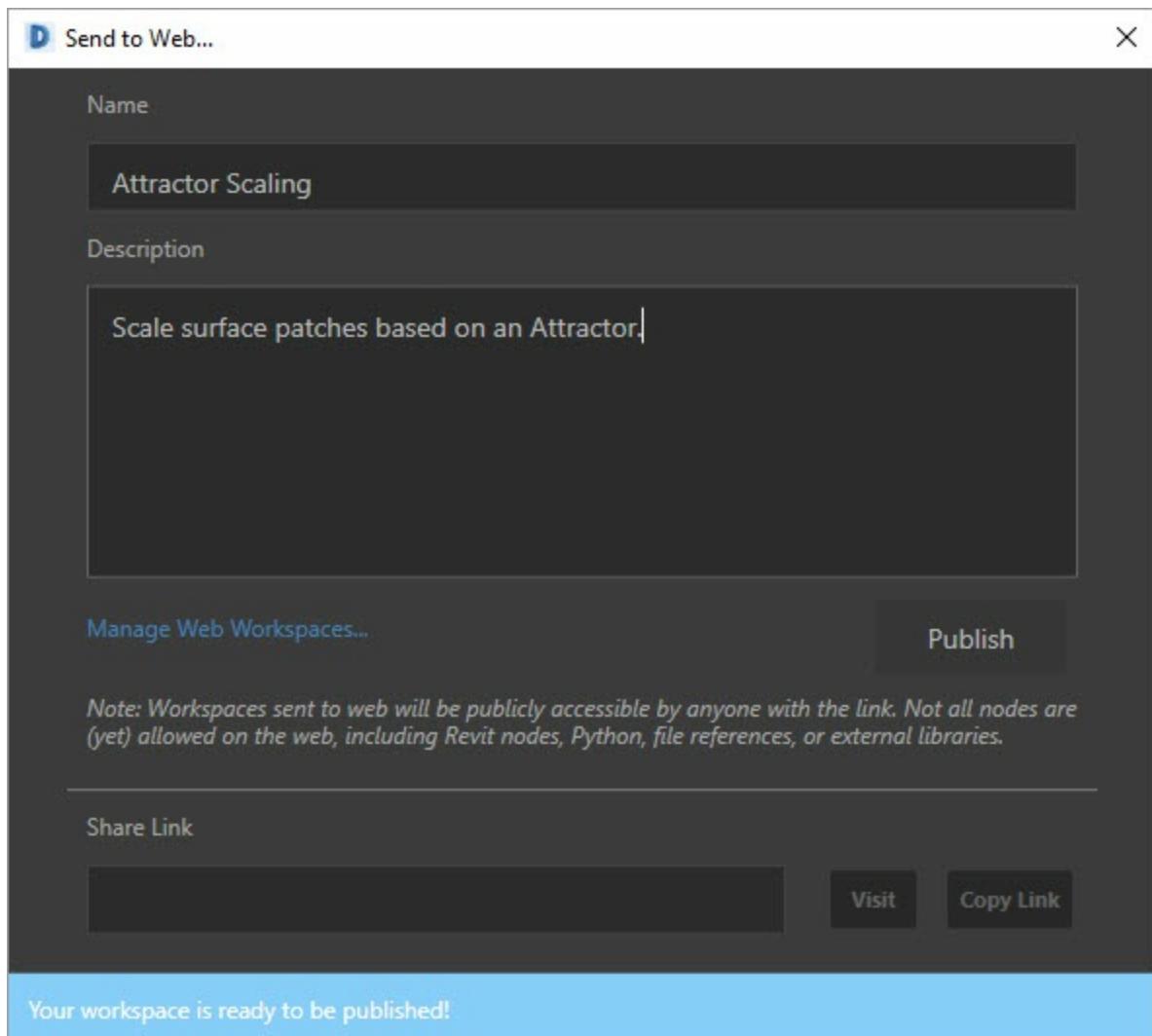


## ファイルをパブリッシュする

ファイルをパブリッシュする準備ができたら、[ファイル]メニューで「Web に送信」を選択します。



ファイルの説明を入力し、ファイルを初めて使用する場合の手順を必要に応じて入力します。ファイルをパブリッシュすると、Autodesk アカウントを持っているすべてのユーザーにそのファイルのリンクを送信できるようになります。このファイルは、現在の入力値とプレビューを持ったままパブリッシュされます。



このサンプルは、[Web](#) で確認することができます。

#### パブリッシュしたファイルを管理する

パブリッシュしたスクリプトを管理するには、<https://dynamo.autodesk.com> にアクセスして自分のアカウントにサインインします。次に、右上のドロップダウンで[管理]を選択します。このページで、パブリッシュされたワークスペースの編集、共有、削除を行うことができます。Dynamo Studio の[ファイル]メニューの[Web ワークスペースの管理]オプションから、このページにアクセスすることもできます。

Name	Description	Last Modified ↑
Attractor Scaling	Scale surface patches based on an attractor.	1 minute ago

© Dynamo 2015

1 2 3

1. ワークスペースの編集
2. ワークスペースの削除
3. リンクの共有

# [カスタマイザ]ビュー

## [カスタマイザ]ビュー

Dynamo の[カスタマイザ]ビューでは、他のユーザが、スライダ、数値、ブール値などの入力から構成されるシンプルなインターフェースを介して、Web 上に置かれた Dynamo のスクリプトにアクセスすることができます。

[カスタマイザ]ビューを使用して複雑なグラフをシンプルなインターフェースに組み込むことにより、Dynamo やビジュアル プログラミングや 3D モデリングに慣れていないユーザも、スクリプトを使用することができます。Autodesk Account を持っているすべてのユーザは、共有リンクから[カスタマイザ]ビューの Dynamo スクリプトにアクセスすることができます。スクリプトへのアクセスは Dynamo のライセンスを持っていない場合でも可能です。

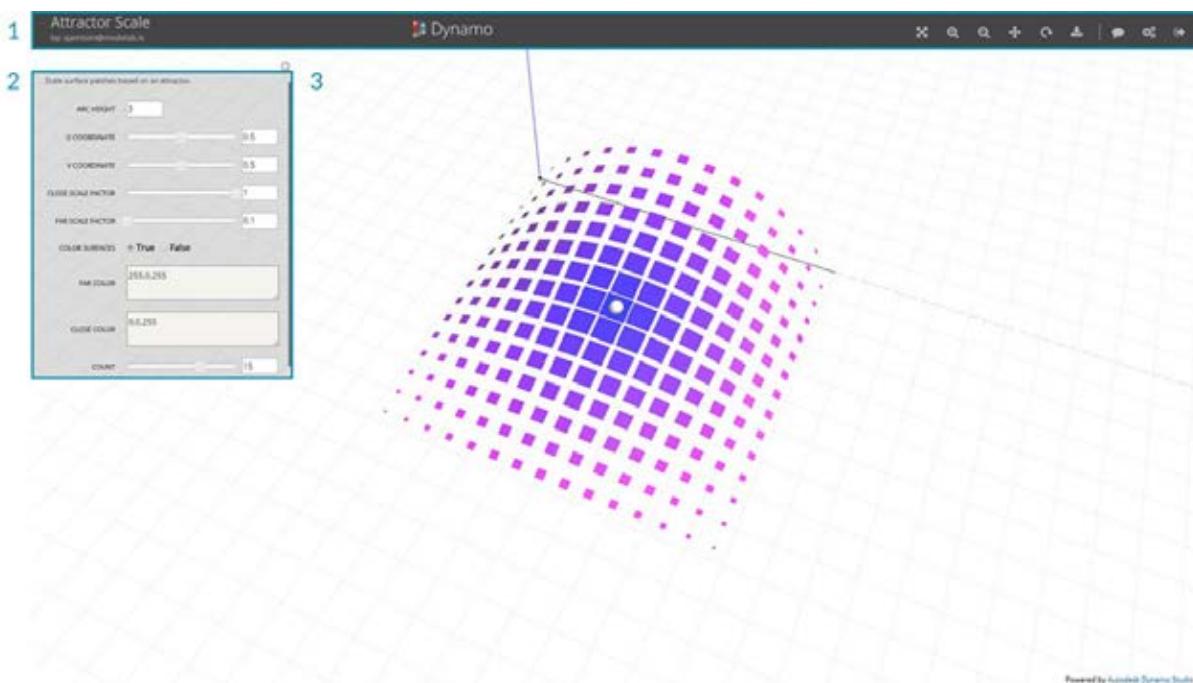
[カスタマイザ]ビューでは、ジオメトリを STL メッシュとして書き出してすばやくプロトタイプを作成することも、Dynamo ファイルとして書き出すこともできます。



サンプルの[カスタマイザ]ビューは、<https://dynamo.autodesk.com/> で参照することができます。

## [カスタマイザ]ビューのユーザ インターフェース

[カスタマイザ]ビューは、メニュー バー、ファイル入力とユーザ入力が表示されるフライアウト メニュー、Dynamo のワークペースに類似した 3D ビューから構成されています。



1. メニュー バー
2. 各種コントロール
3. 3D プレビュー

## [カスタマイザ]ビューのメニュー

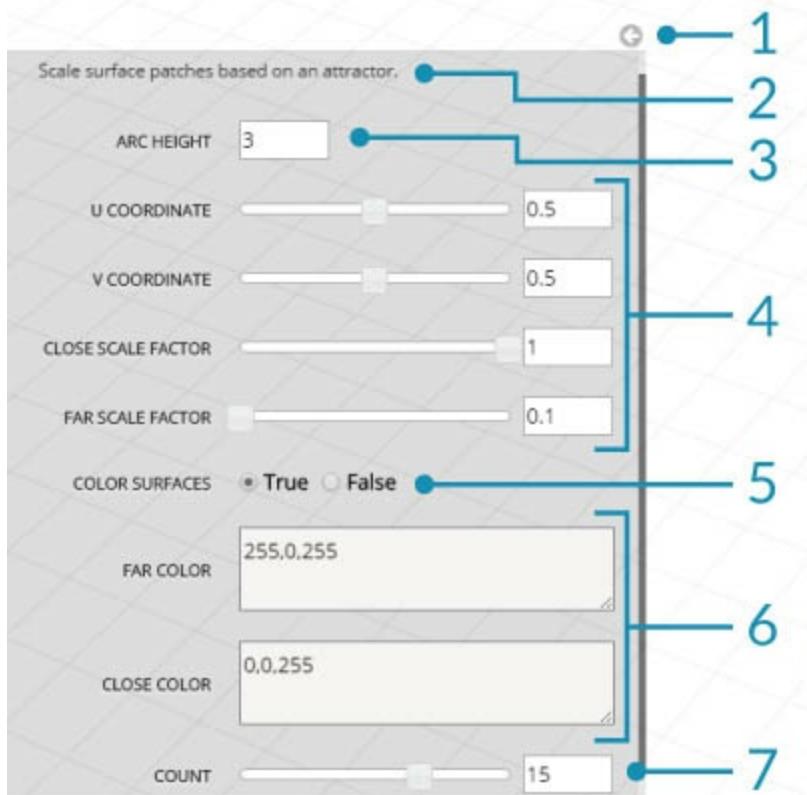
[カスタマイザ]メニュー バーには、ファイル情報、ナビゲーション用の各種コントロール、ダウンロード オプションが表示されます。



1. タイトル: ファイル名が表示されます。
2. 作成者: ファイルの作成者が表示されます。
3. オブジェクト範囲ズーム: ジオメトリの範囲にズームします。
4. 拡大/縮小:- 拡大表示/縮小表示を行います。
5. 画面移動/オービット:- オービットと画面移動を切り替えます。
6. ダウンロード:- ファイルを STL または DYN として保存します。
7. フィードバック: コメント、提案、問題を送信します。
8. 管理/バージョン情報: Dynamo に関する情報が表示されます。
9. ログアウト: アカウントからログアウトして[カスタマイザ]ビューを終了します。

## 各種コントロール

コントロール メニューには、数値、スライダ、文字列、ブール値、ファイルの短い説明など、Dynamo スクリプトに対する入力情報が表示されます。各コントロールには、元の Dynamo ファイルで検出されたさまざまな入力情報が表示されます。表示されるコントロールは、スクリプトに応じて異なります。矢印アイコンをクリックすると、このメニューを折りたたむことができます。

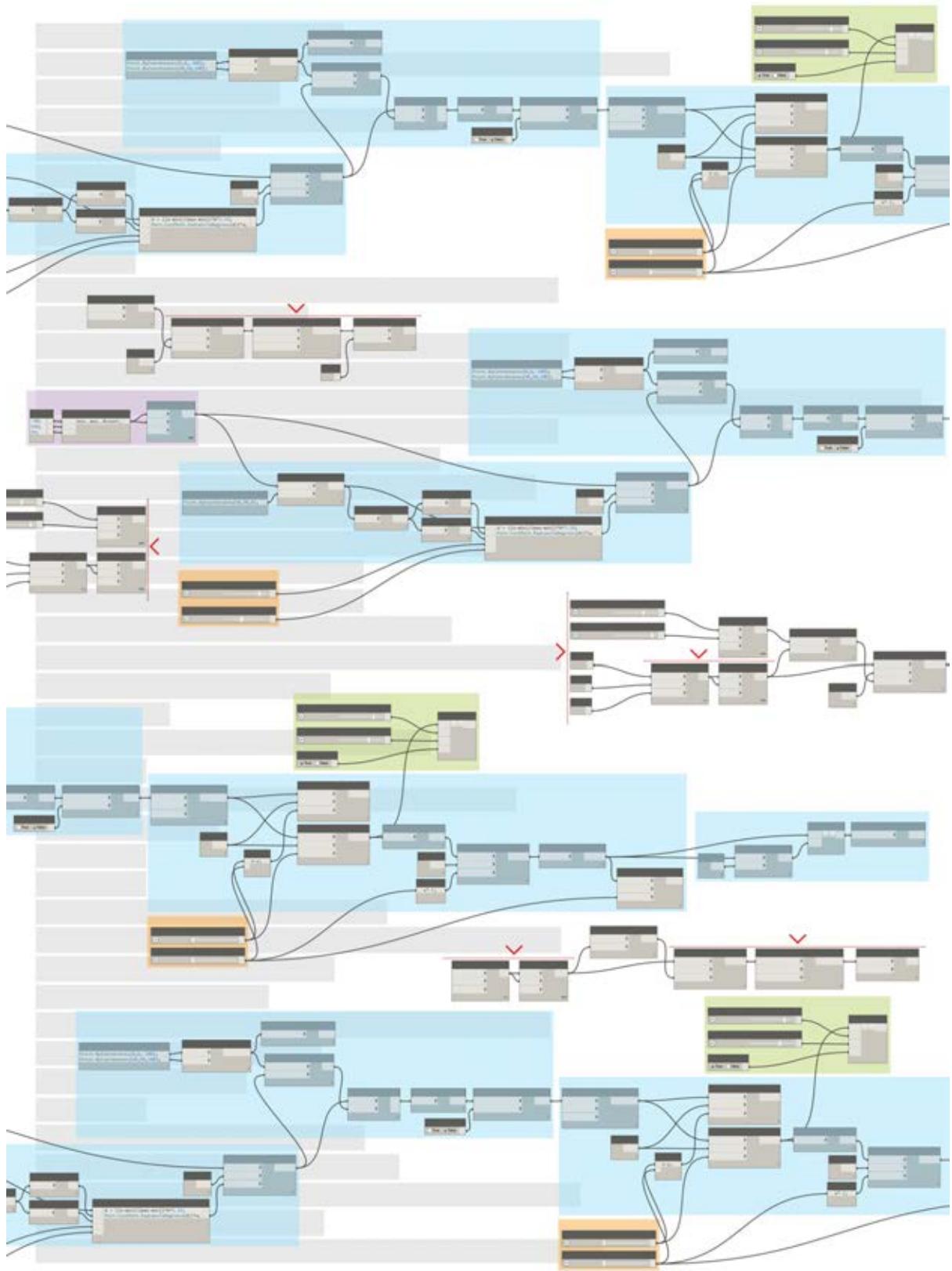


1. コントロールの折りたたみ/展開
2. ファイルの説明
3. 入力された数値
4. 数値スライダ
5. ブール値
6. 文字列
7. 整数スライダ

## ベスト プラクティス

### ベスト プラクティス

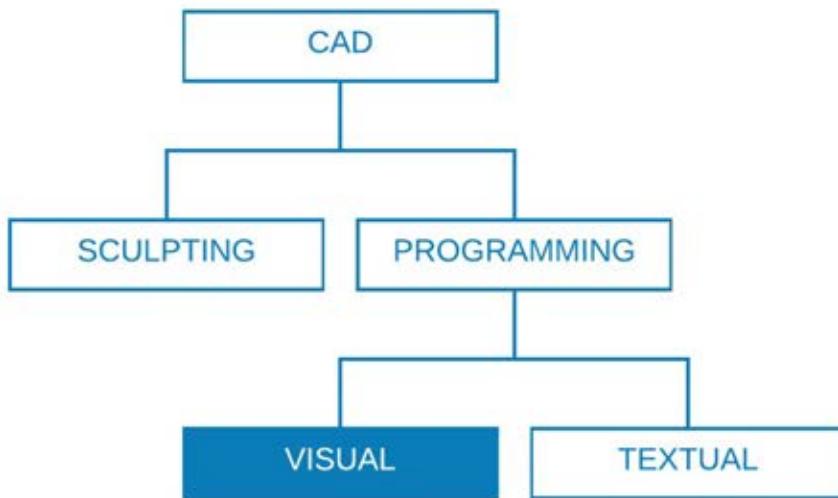
ここでは、さまざまなベスト プラクティスを紹介します。これらのベスト プラクティスは、これまでの経験と調査によって得たいいくつかの方策が、品質の高いパラメトリック ワークフローを構築するために最も効果的であることを明らかにしています。設計者とプログラマの場合、オートデスクの各種ツールの保守性、信頼性、利便性、効率性を主な指標として、品質を評価することになります。ここで紹介するベスト プラクティスでは、ビジュアルベースのスクリプト作成とテキストベースのスクリプト作成における具体的な例を挙げていますが、スクリプト作成の原則は、すべてのプログラミング環境で共通しています。これらの原則を適用して、さまざまな計算ワークフローを示すことができます。



# 見やすいプログラムを作成するためのガイドライン

## 見やすいプログラムを作成するためのガイドライン

この章に入る前に、Dynamo の強力なビジュアル スクリプト機能の実装方法について説明しました。これらの機能を正しく理解することが、堅固なビジュアル プログラムを開発するための基礎となり、第一歩となります。実際の現場でビジュアル プログラムを使用する場合、チーム メンバーとビジュアル プログラムを共有する場合、エラーのトラブルシューティングを行う場合、制限値のテストを行う場合は、新たな問題が発生します。自分が作成したプログラムを他のメンバーが使用する場合や、今から 6 カ月後にプログラムを起動するような場合は、外観的な面でも論理的な面でも、すぐに理解できるようなプログラムを作成する必要があります。Dynamo には、複雑なプログラムを管理するためのさまざまなツールが用意されています。この章では、こうしたツールを使用する場合のガイドラインについて説明します。

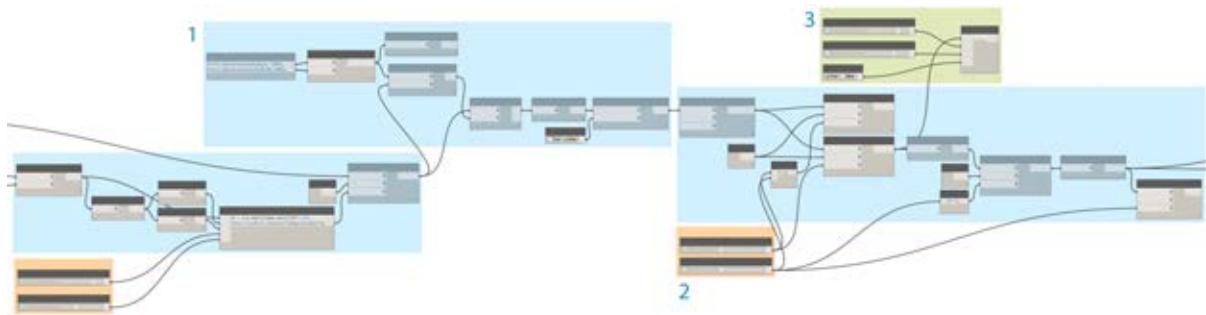


### 複雑さを軽減する

Dynamo でプログラムを作成したり、アイデアをテストする場合、そのサイズと複雑さが急激に増大することがあります。正しく機能するプログラムを作成することはもちろん重要なことです。可能な限り簡潔なプログラムを作成することも同じくらいに重要なことです。簡潔なプログラムを作成することにより、プログラムの処理速度と精度が上がるだけでなく、後で他のメンバーとともにコードを解析する場合も、ロジックを簡単に追うことができるようになります。ここでは、プログラムのロジックを分かりやすく記述するための方法をいくつか紹介します。

#### グループを使用してプログラムをモジュール化する

- プログラムの作成時にグループを使用して、機能別に異なるパートを作成することができます。
- グループを使用すると、各モジュールとその配置を維持しながら、プログラム内の大きなパートを移動することができます。
- グループの色を変えて、各グループの用途(入力や関数など)を区別することができます。
- グループを使用してプログラムを整理し、カスタム ノードの作成を簡素化することができます。

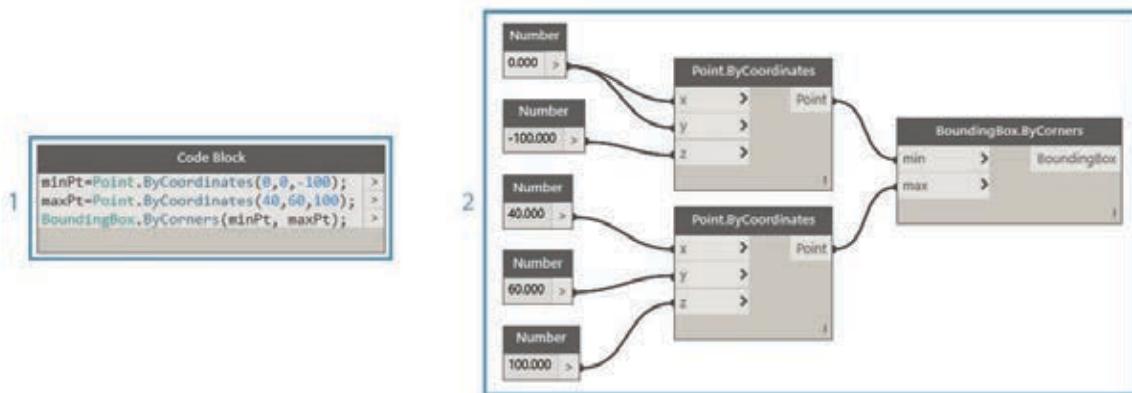


このプログラムでは、各グループの用途を色で示しています。この方法により、開発するプログラムの標準やテンプレート内で階層を作成することができます。

1. 関数グループ(青)
2. 入力グループ(オレンジ色)
3. スクリプトグループ(緑) グループの使用方法については、「[プログラムを管理する](#)」を参照してください。

#### コード ブロックを使用して効率的に開発する

- 必要に応じてコード ブロックを使用すると、検索するよりも速く、数値メソッドやノード メソッドを入力することができます (Point.ByCoordinates、Number、String、Formula)。
- DesignScript でカスタム関数を定義してプログラム内のノードの数を減らす場合は、コード ブロックを使用すると便利です。



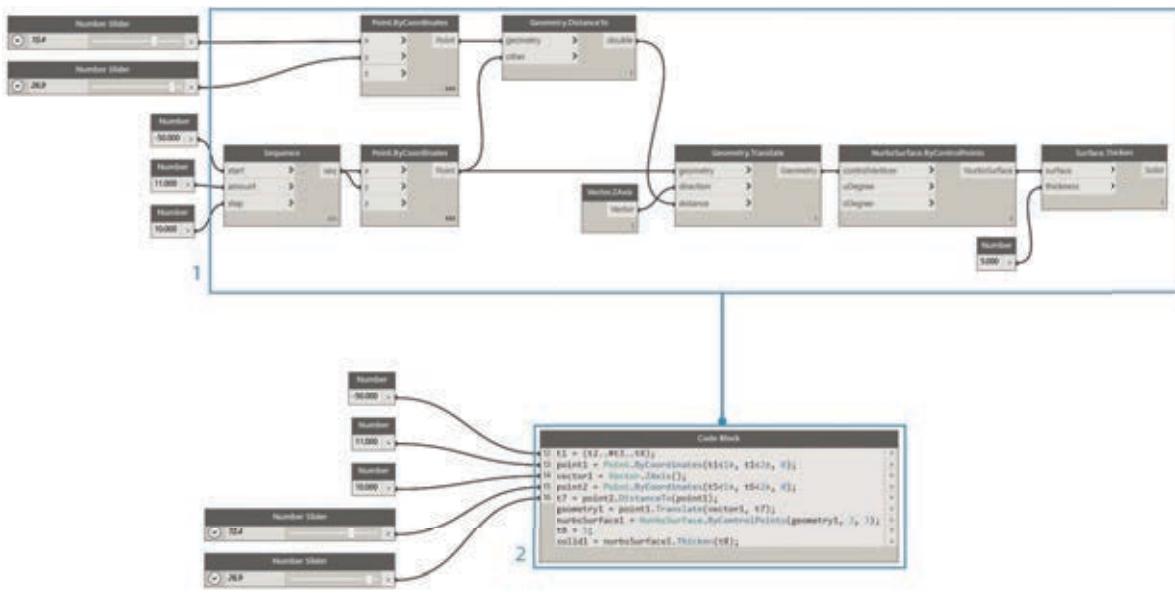
1と2は、どちらも同じ関数を実行します。各ノードを個別に検索して追加するよりも、コードを何行か入力する方が、ずっと速く作業を進めることができます。また、コード ブロックのサイズもずっと小さくなります。

1. コード ブロックを使用して記述した設計スクリプト
2. ノードを使用して同じスクリプトを記述した場合 コード ブロックの使用方法については、「[コード ブロックとは](#)」を参照してください。

#### ノードをコードに変換する

- [ノードをコード化]機能を使用して、画面上の煩雑さを軽減することができます。この機能では、複数の単純なノードを1つにまとめるにより、それらのノードに対応する DesignScript が1つのコード ブロック内で記述されます。
- [ノードをコード化]機能を使用すると、プログラム ロジックのわかりやすさを損なうことなく、複数のノードを1つのコード ブロックにまとめることができます。
- [ノードをコード化]機能を使用するメリットには、次のようなものがあります。
- コードを編集可能な1つのコンポーネントとして簡単に集約できる
- 画面上の大部分を簡素化できる
- 小さなプログラムを頻繁に編集する必要がない場合に便利である
- 関数など、他のコード ブロック機能を組み込む場合に便利である
- [ノードをコード化]機能を使用するデメリットには、次のようなものがあります。
- 一般的な命名規則が使用されるため、コードの内容が多少わかりにくくなる

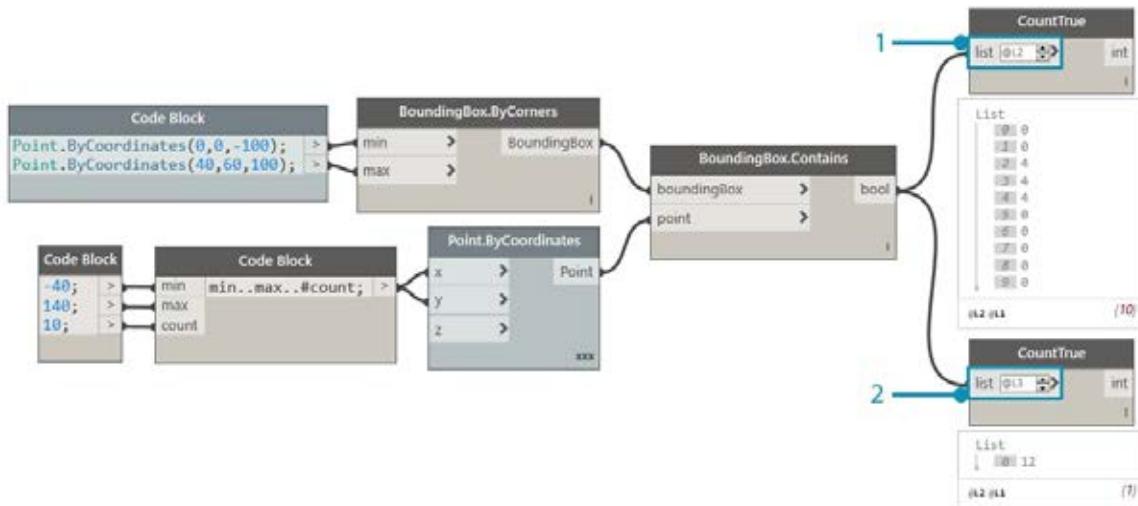
- 他のユーザにとって、コードの理解が難しくなる
- 元のビジュアル プログラムに簡単に戻すことができない



1. 既存のプログラム
2. [ノードをコード化]機能によって作成されたコード ブロック [ノードをコード化]機能の使用方法については、「[DesignScript 構文](#)」を参照してください。

List@Level 機能を使用してデータに柔軟にアクセスする

- List.Map ノードと List.Combine ノードはキャンバス上で大きなスペースを占有する場合がありますが、List@Level 機能を使用してこれらのノードを置き換えることにより、画面上の煩雑さを軽減することができます。
- List@Level 機能を使用すると、ノードの入力ポートからリスト内の任意のレベルに直接アクセスできるため、List.Map ノードや List.Combine ノードを使用するよりも速くノードロジックを記述することができます。



CountTrue ノードの list 入力で List@Level 機能を有効にすると、BoundingBox.Contains ノードから返される True 値の数と、その True 値を返すリストを確認することができます。List@Level 機能により、入力データの取得元となるレベルを特定することができます。List.Map ノードと List.Combine ノードを使用する場合、List@Level 機能は、他の方法よりも柔軟で効率的な方法です。これらのノードで作業を行う場合は、List@Level 機能を使用することを強くお勧めします。

1. リストレベル 2 で True 値の数をカウント
2. リストレベル 3 で True 値の数をカウント List@Level 機能の使用方法については、「[リストのリスト](#)」を参照してください。

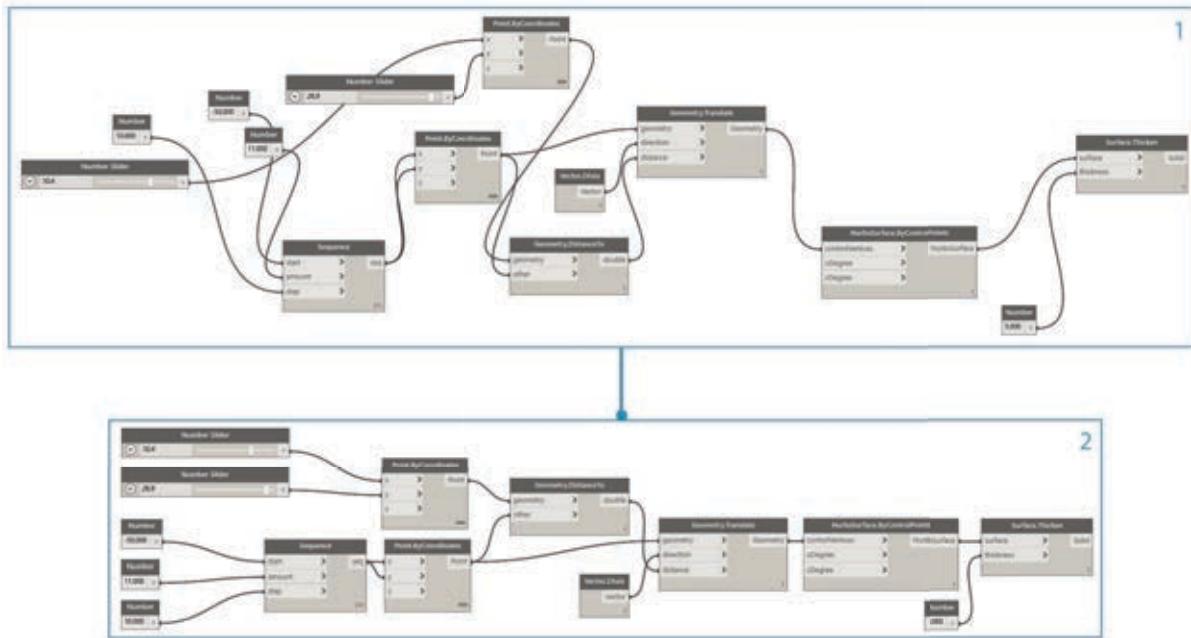
見やすいプログラムにする

可能な限り簡潔で効率的なプログラムを作成するには、見やすいプログラムを作成するということに重点を置く必要があります。論理

的にグループ化された直感的なプログラムを作成した場合であっても、データ間の関係性が分かりにくくなることがあります。グループ内に簡単な注記を記載したり、スライダー名を分かりやすい名前に変更すると、自分だけでなく他のメンバーも、名前などで混乱したり、画面上を無駄に移動することがなくなります。ここでは、プログラム全体で一貫した外観を保つための方法をいくつか紹介します。

#### ノードの配置を調整して視覚的な一貫性を保つ

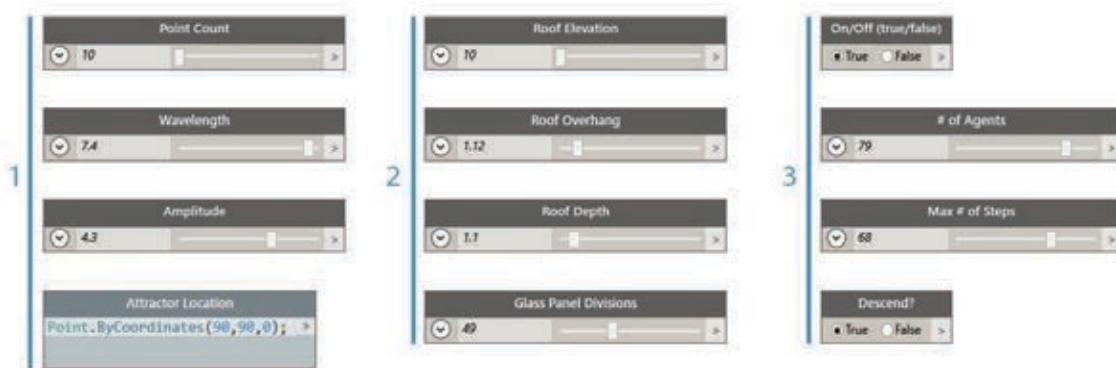
- プログラム作成後の作業量を減らすには、プログラムの作成時に頻繁にノードの配置を調整して、ノードのレイアウトを見やすく整える必要があります。
- 自分が作成したプログラムで他のメンバーが作業を行う場合は、ワイヤが自然な方向に流れるように各ノードを配置する必要があります。
- ノードを配置するには、[ノードのレイアウトをクリーンアップ]機能を使用してノードの配置を自動的に調整します。ただし、手動で調整する場合と比べて、多少精度が下がることに注意してください。



1. ノードが整理されていない画面
2. ノードが整理されている画面 ノードの配置を調整する方法については、「[プログラムを管理する](#)」を参照してください。

名前を変更してわかりやすいラベルを付ける

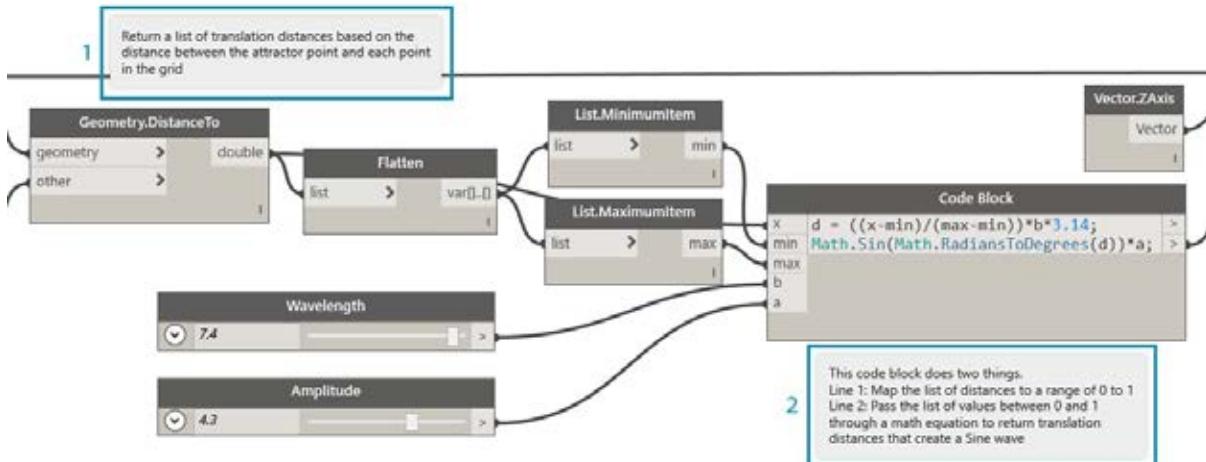
- 入力の名前を変更することにより、自分が作成したプログラムを他のメンバーが使用する場合でも、内容を簡単に理解できるようになります。特に、入力の接続先ノードが画面からはみ出してしまうような場合は、入力名からそのノードの内容を推測できるため、分かりやすい名前を付けておくと便利です。
- 入力以外のノード名を変更する場合は、注意が必要です。その場合は、別の方法として、ノードクラスタからカスタムノードを作成し、そのカスタムノードの名前を変更します。こうすることにより、そのノードの内容は、他のノードとは異なっているということを示すことができます。



1. サーフェスを操作するための入力
2. 建築設計パラメータ用の入力
3. 排水シミュレーションスクリプト用の入力 ノード名を変更するには、変更するノード名を右クリックして[ノードの名前を変更...]を選択します。

説明としてノートを追加する

- ノードでは表現できないような分かりやすい説明をプログラムに挿入する必要がある場合は、ノートと呼ばれる注記を追加します。
- ノードの集合やグループのサイズが大きすぎて(または複雑すぎて)簡単には理解できない場合は、ノートを追加することをお勧めします。



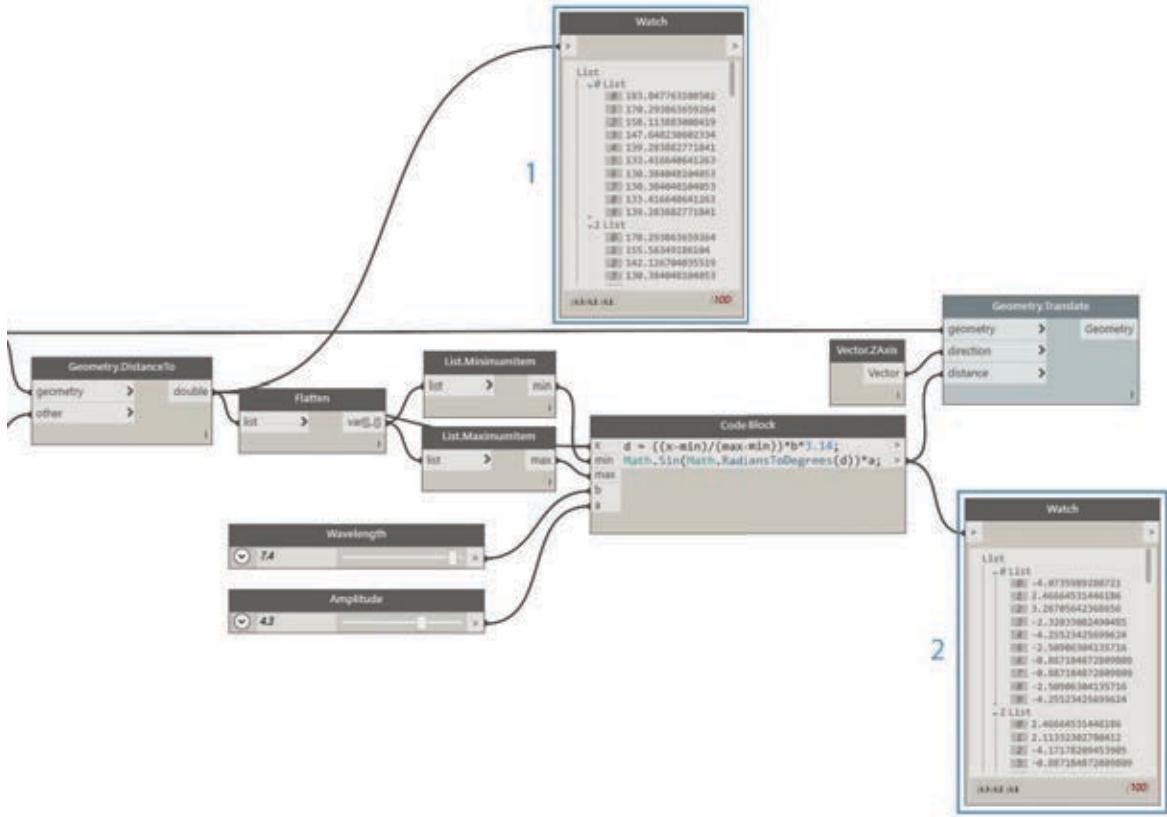
1. 行の移動距離を返すプログラムの特定の箇所を説明するためのノート
2. 上記の移動距離の値を正弦波にマッピングするコードを説明するためのノートノートの追加方法については、「[「プログラムを管理する」](#)」を参照してください。

### スクリプトを継続的にモニタリングする

ビジュアルスクリプトを作成する場合、正しい値がスクリプトから返されるかどうかを確認することが重要になります。すべてのエラーが、プログラムの即時停止につながるエラーというわけではありません。特に、Null値やゼロの値に関するエラーは、プログラムの下流部分に影響する場合があるため、注意が必要です。この方法は、「[スクリプト作成のガイドライン](#)」の章でも、テキストスクリプトに関連する形で記載されています。次の演習では、正しい結果を得る方法について確認していきます。

ウォッチ バルーンとプレビュー バルーンを使用してデータをモニターする

- プログラムの作成時にウォッチ バルーンまたはプレビュー バルーンを使用すると、重要な出力データが正しく返されるかどうかを確認することができます。



この例では、Watch ノードを使用して、次のデータを比較しています。

1. 行の移動距離
2. 正弦式から渡された値 Watch ノードの使用方法については、「[ライブリ](#)」を参照してください。

### 再利用しやすいプログラムを作成する

自分が他のメンバーとは別の作業を行っている場合であっても、自分が作成したプログラムを後で他のメンバーが使用するというのは非常によくあることです。その場合に備えて、入力データと出力データから、プログラムの内容と結果を短時間で把握できるようにしておく必要があります。これは特に、カスタム ノードを開発して Dynamo コミュニティで共有し、他のメンバーが作成したプログラム内でそのカスタム ノードを使用する場合に重要になります。こうすることにより、簡単に再利用できる堅固なプログラムとノードを作成することができます。

#### 入出力を管理する

- プログラムの読みやすさと拡張性を確保するため、入力と出力は可能な限り少なくすることをお勧めします。
- キャンバス上でノードを追加する前に、ロジックの概要を組み立ててから、そのロジックをどのように作成していくかを検討するようにしてください。概要を組み立てる際に、スクリプト内で使用する入力と出力を検討する必要があります。

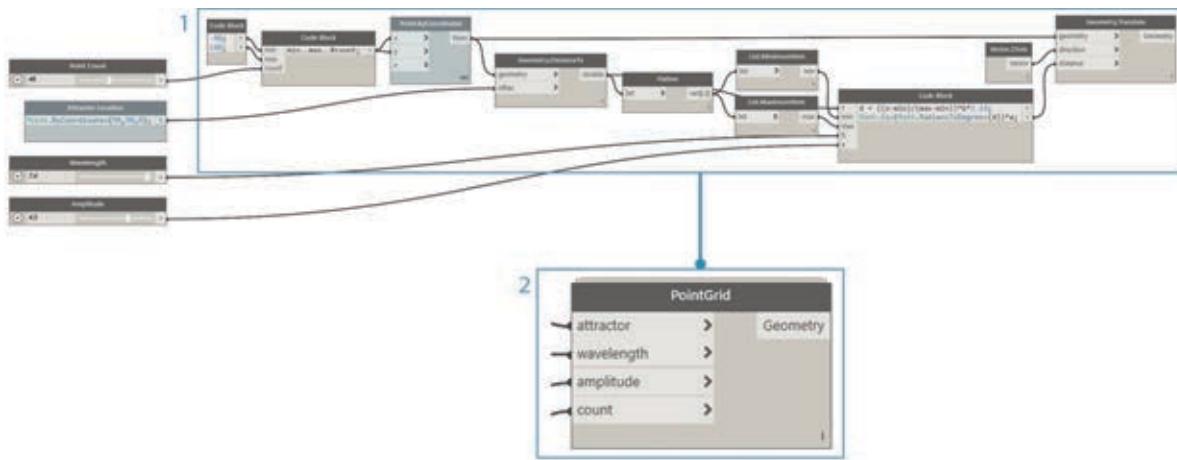
#### プリセットを使用して入力値を組み込む

- 特定のオプションや条件をプログラム内に組み込む場合は、プリセットを使用すると、迅速に作業を進めることができます。
- また、長期間実行されるプログラム内でプリセットを使用して特定のスライダ値をキヤッショすると、プログラムの複雑さを軽減することができます。

プリセットの使用方法については、「[プリセットを使用してデータを管理する](#)」を参照してください。

#### プログラムを構成する各ノードをカスタム ノード内に収集する

- プログラムを構成する各ノードを 1 つのコンテナ内に収集できる場合は、カスタム ノードを使用することをお勧めします。
- 他のプログラム内で一部のコードを頻繁に再利用する場合は、カスタム ノードを使用することをお勧めします。
- **Dynamo** コミュニティ内で特定の機能を共有する場合は、カスタム ノードを使用することをお勧めします。

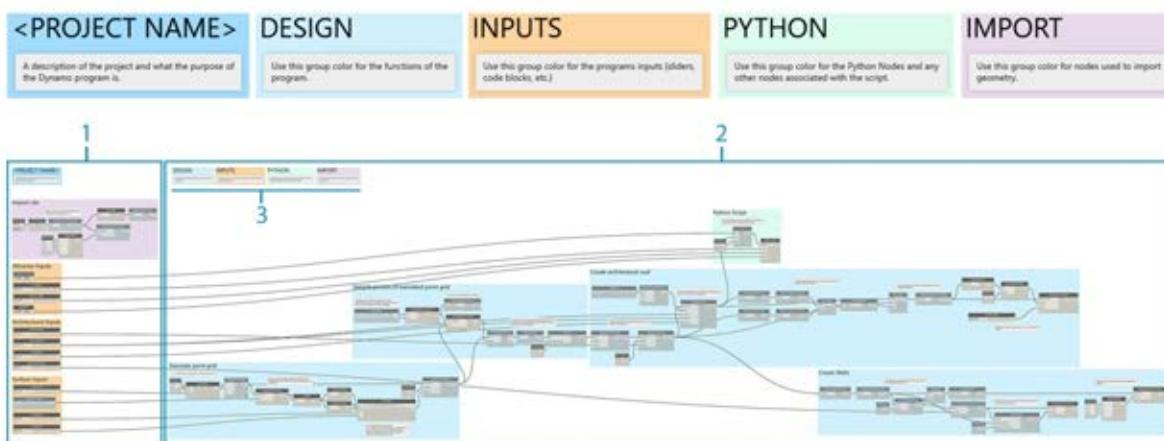


点の移動プログラムを構成する各ノードをカスタムノード内に収集すると、移植可能で堅固な一意のプログラムを作成することができます。また、プログラムの内容も簡単に理解できるようになります。入力ポートに分かりやすい名前を付けると、他のメンバーがそのプログラムを使用する際に、ノードの使用方法を簡単に理解できるようになります。すべての入力について、説明と必要なデータタイプを追加してください。

- 既存のアトラクタープログラム
- 上記のプログラムを構成する各ノードを収集するためのカスタムノード「PointGrid」カスタムノードの使用方法については、「[カスタムノードの概要](#)」を参照してください。

#### テンプレートを作成する

- テンプレートを作成してビジュアルプログラム全体の外観的な標準を定義することにより、すべてのチームメンバーが標準化された方法でプログラムを理解できるようになります。
- テンプレートを作成する際に、グループの色とフォントサイズを標準化して、ワークフローやデータ操作のタイプを分類することができます。
- テンプレートを作成する際に、プログラム内のワークフローのフロントエンドとバックエンドとの差異について、ラベルを付けたり、色を付けたり、スタイルを設定することもできます。



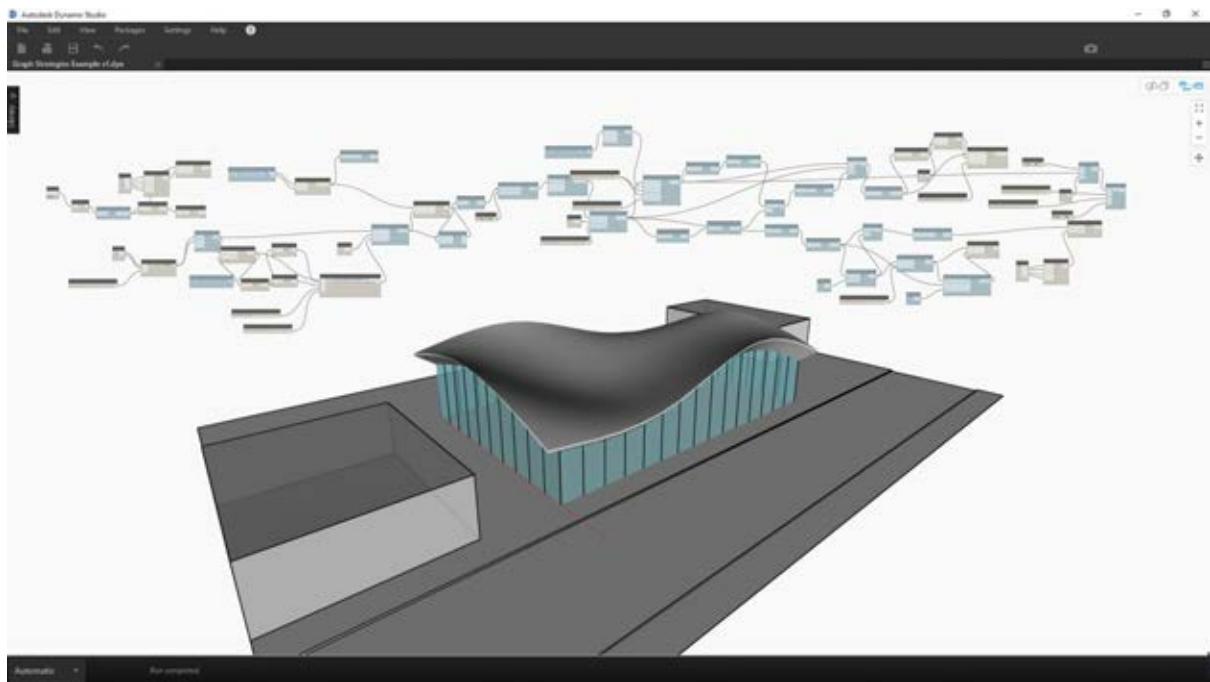
- プログラムのUI(フロントエンド)。プロジェクト名、入力スライダ、読み込みジオメトリが表示されます。
- プログラムのバックエンド。
- グループ(一般的な設計、入力、Pythonスクリプト、読み込まれたジオメトリ)を区別するための色分け。

#### 演習 - 建築設計で使用する屋根を作成する

この演習に付属しているサンプルファイルをダウンロードしてください(右クリックして[名前を付けてリンク先を保存]を選択)。すべてのサンプルファイルの一覧については、付録を参照してください。[RoofDrainageSim.zip](#)

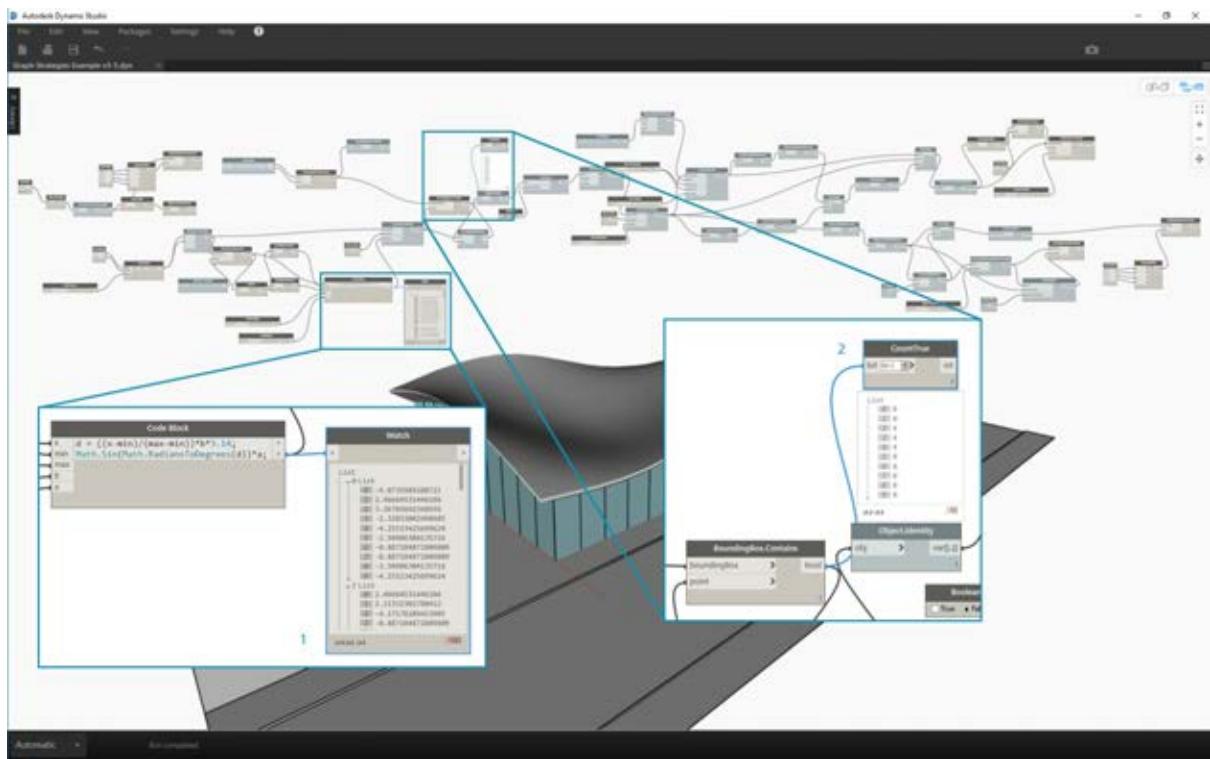
この演習では、上で説明したベストプラクティスを構造化されていないプログラムに適用してみましょう。このプログラムでは、屋根を正しく作成することができますが、計画的に作成されたものではないため、構造化されていない状態になっています。各ノードの配置は整理されておらず、ノードの用途を示す説明も入力されていません。このプログラムの作成者以外のユーザーもこのプログラムの内容を理解できるように、ノード配置の整理、説明の追加、プログラムの解析について、ベストプラクティスを適用する手順を順に見てい

きましょう。



このプログラムは正しく機能しますが、ノードが乱雑に配置されているため、見にくくなっています。

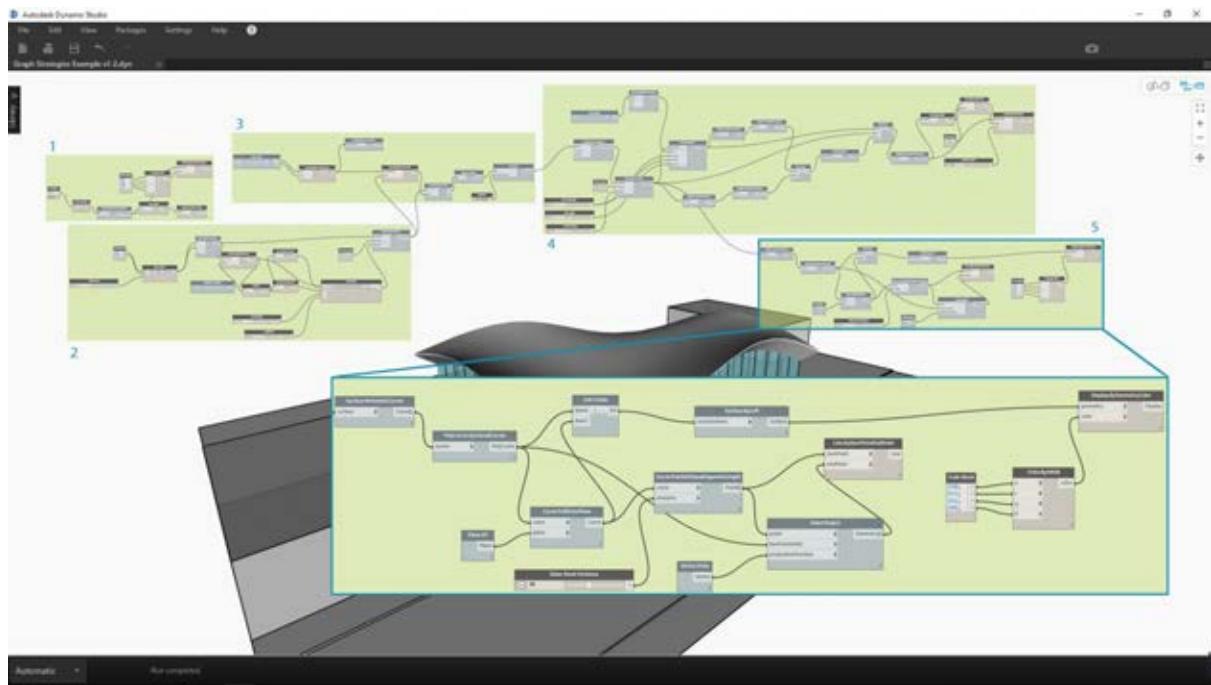
最初に、プログラムから返されるデータとジオメトリを確認しましょう。



論理的な単位であるモジュールを作成する場合、データが大きく変更されるタイミングを把握することが非常に重要になります。Watch ノードを使用してプログラムを検査し、ノードのグループ化を検討してから、次のステップに進むようにしてください。

1. このコード ブロックには数式が記述されているため、プログラムの重要な部分を構成していると考えられます。Watch ノードを使用すると、プログラムから返される移動距離のリストを表示することができます。
2. これらのノードの用途については、一見しただけではよくわかりません。BoundingBox.Contains ノードのリスト レベル L2 の True 値の配置と、List.FilterByBoolMask ノードが使用されていることから考えると、点構成グリッドの一部をサンプリングしていることが推測できます。

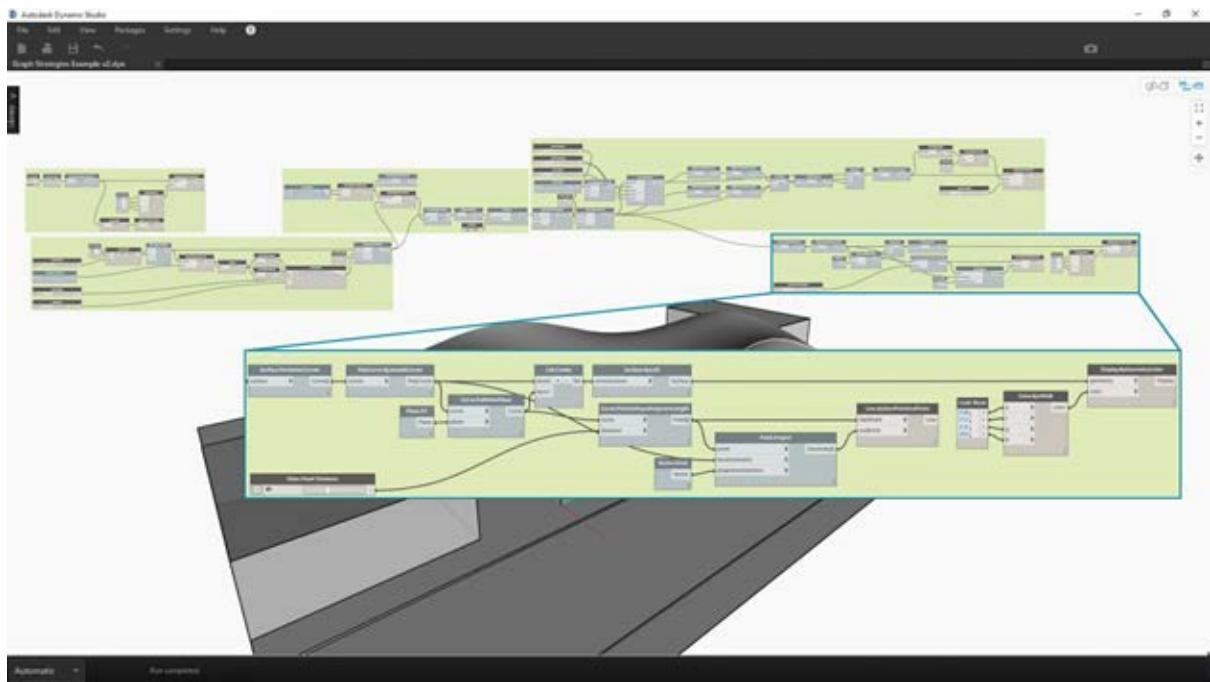
プログラム内の各要素の内容が理解できたら、それらの要素をグループ化します。



ノードをグループ化することにより、プログラムを構成するパートを視覚的に区別できるようになります。

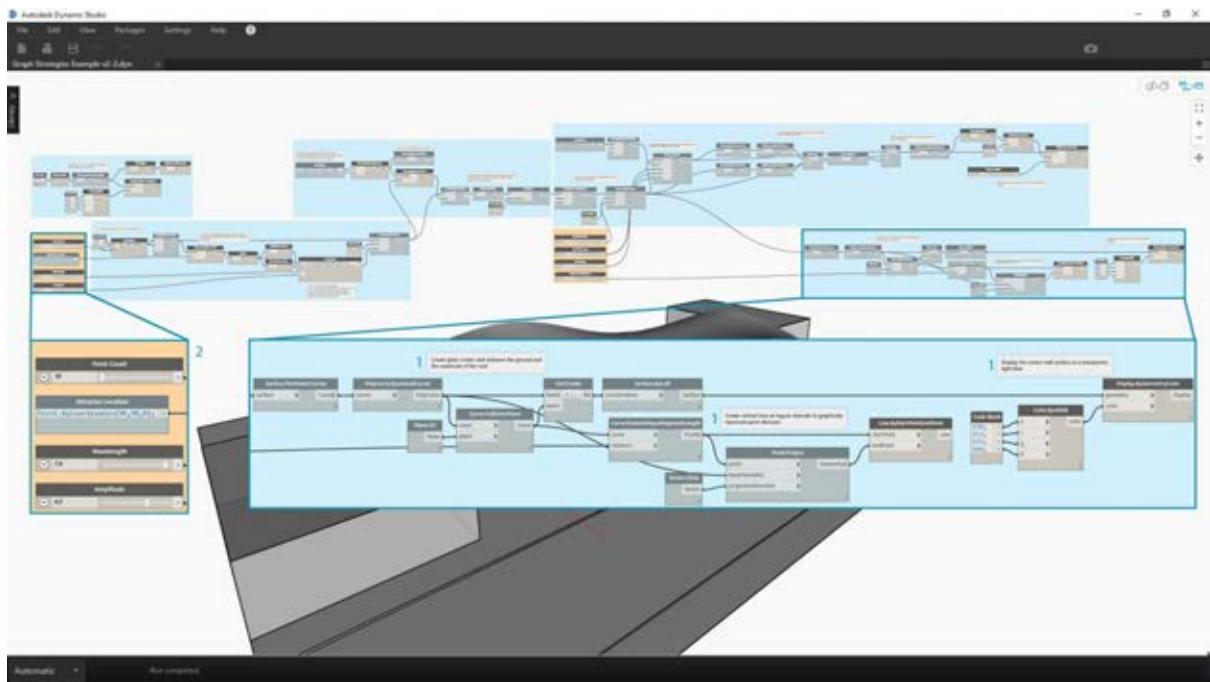
1. 3D サイト モデルを読み込むグループ
2. 正弦式に基づいて点構成グリッドを移動するグループ
3. 点構成グリッドの一部をサンプリングするグループ
4. 建築設計で使用する屋根のサーフェスを作成するグループ
5. ガラス カーテン ウォールを作成するグループ

グループ化が完了したら、ノードの配置を調整して、プログラム全体の流れを見やすく整理します。



プログラムの流れを見やすく整理することにより、プログラムの構造と各ノード間の暗黙的な関係を把握できるようになります。

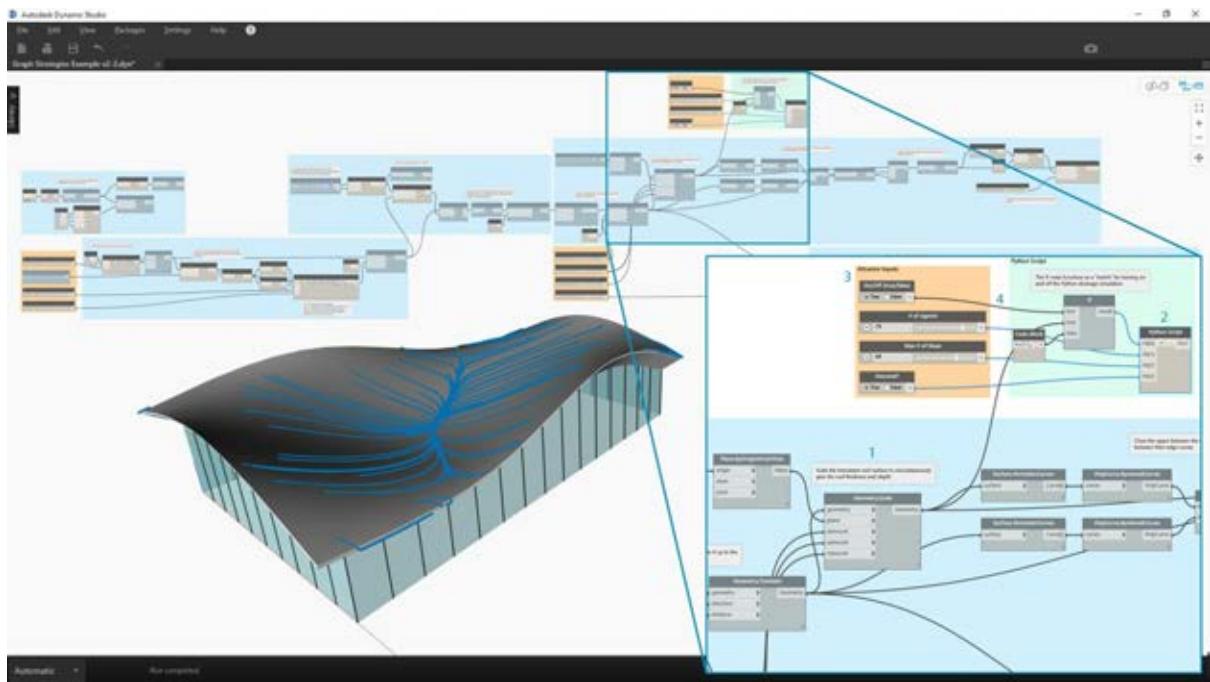
プログラムをさらに見やすくするため、外観的な情報を追加してみましょう。ここでは、プログラムの特定の領域がどのように機能するかを示すためのノートを追加し、入力のカスタム名を設定し、グループのタイプを区別するための色を割り当てます。



これらの外観的な情報を追加することにより、プログラムの内容がさらに分かりやすくなります。グループを色分けすると、関数から渡される入力を簡単に区別することができます。

1. ノート
2. 分かりやすい名前が設定された入力

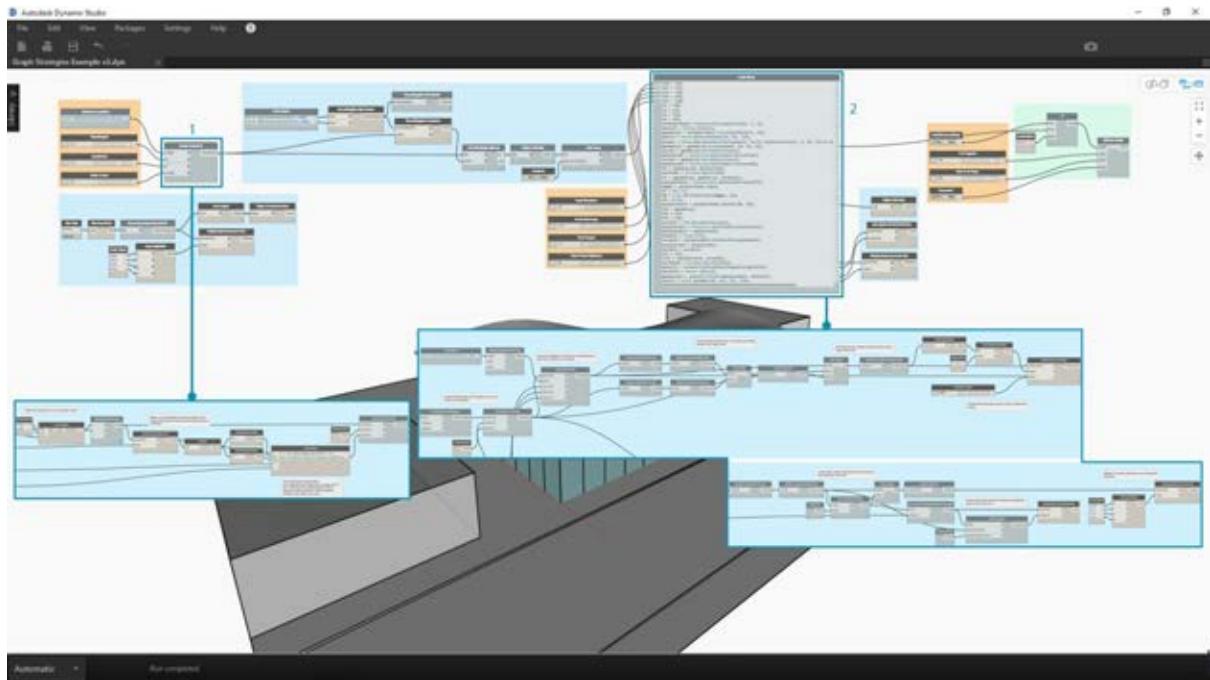
プログラムの集約を開始する前に、プログラムのどの位置に Python スクリプトの排水シミュレータを組み込むかについて確認しましょう。スケーリングされた最初の屋根のサーフェスの出力を、それぞれのスクリプト入力に接続します。



ここでは、上図の位置にスクリプトを組み込みます。これにより、元の屋根のサーフェス上で排水シミュレーションを実行できるようになります。このサーフェスはプレビュー表示されていませんが、このサーフェスにより、面取りされたポリサーフェスの上部サーフェスを選択する必要がなくなります。

1. スクリプト入力のソース ジオメトリ
2. Python ノード
3. 入力スライダ
4. オン/オフ スイッチ

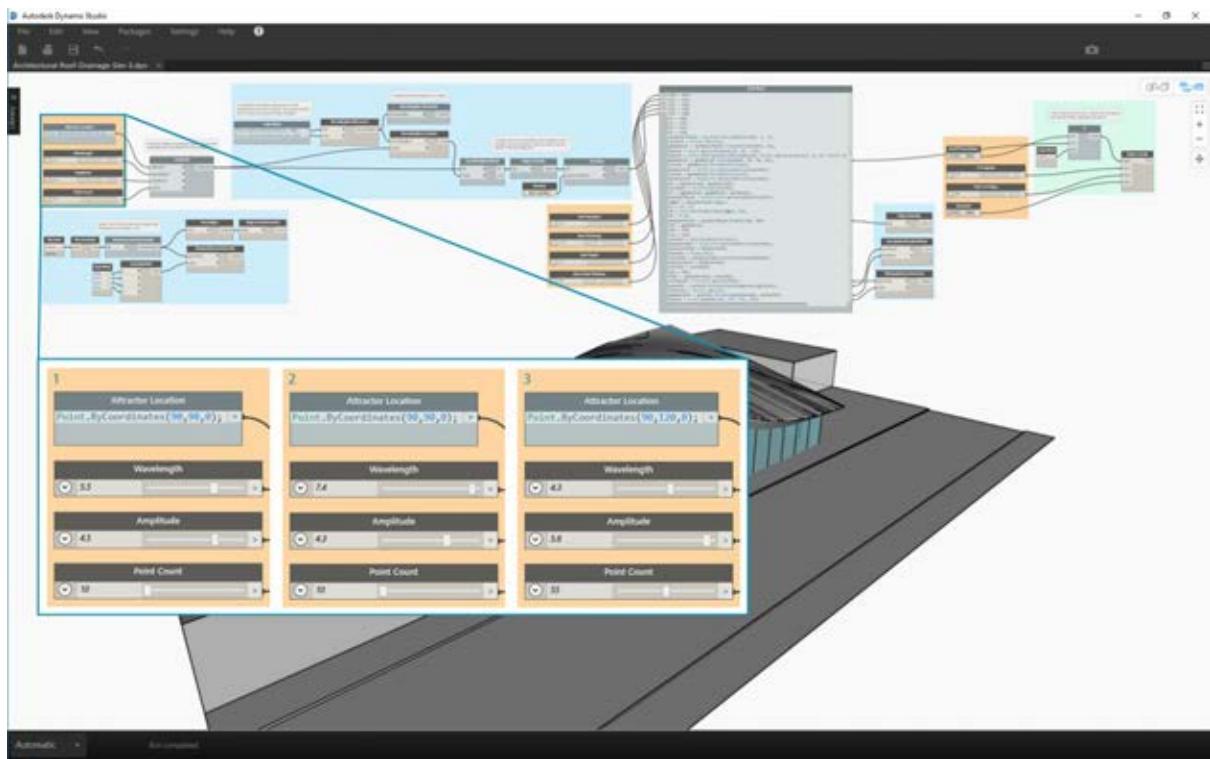
これですべての準備が整ったので、プログラムを集約してみましょう。



[ノードをコード化]機能とカスタム ノードを使用してプログラムを集約すると、プログラムのサイズが大幅に削減されます。屋根のサーフェスと壁を作成するグループは、このプログラムにとって非常に重要な機能を実行するグループであるため、コードに変換されています。点を移動するグループは、他のプログラムでも使用できるように、カスタム ノード内に集約されています。このサンプル ファイルで、点を移動するグループを使用して、独自のカスタム ノードを作成してみてください。

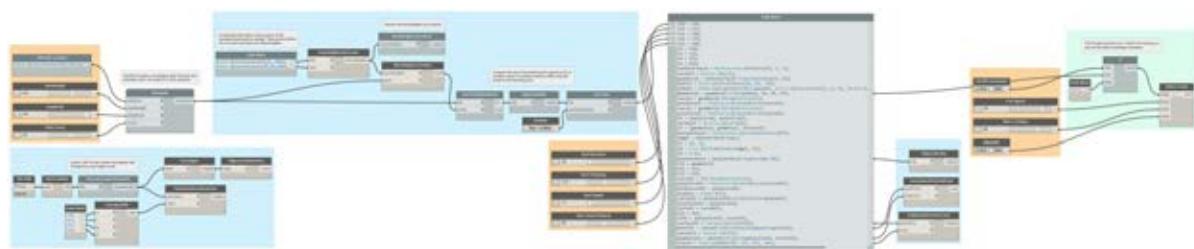
1. 点構成グリッドを移動するグループを格納するためのカスタム ノード
2. 建築設計で使用する屋根のサーフェスとカーテン ウォールを作成するグループを集約するための[ノードをコード化]機能

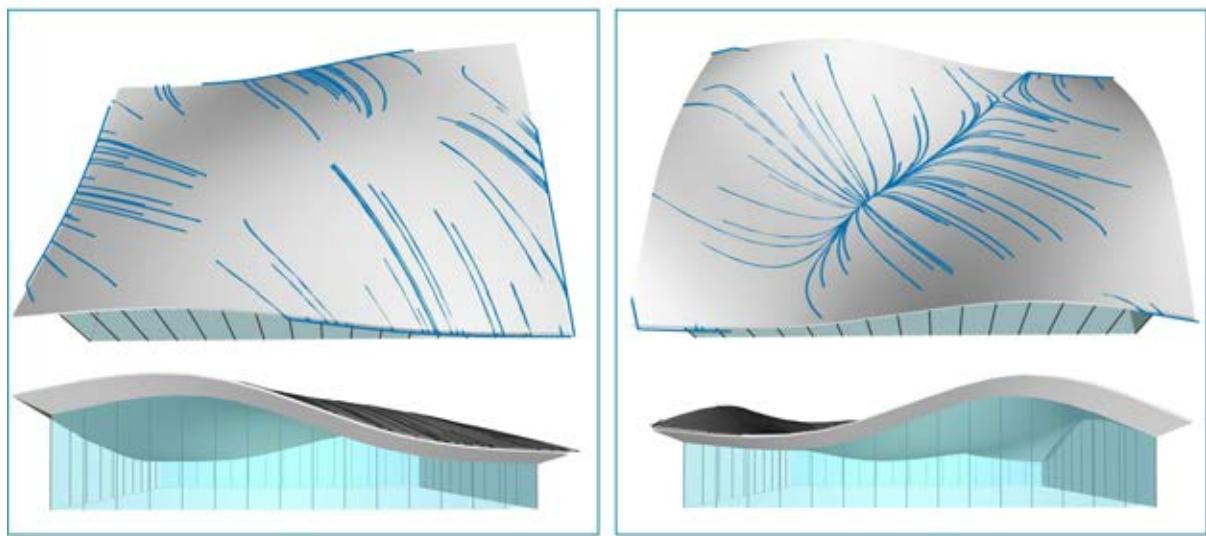
最後に、サンプルの屋根形状用のプリセットを作成します。



上図の各入力が、屋根形状の主要な駆動要素になります。これらの入力を確認することにより、プログラムの概要を把握することができます。

2つのプリセットは、次のようなビューとして表示されます。



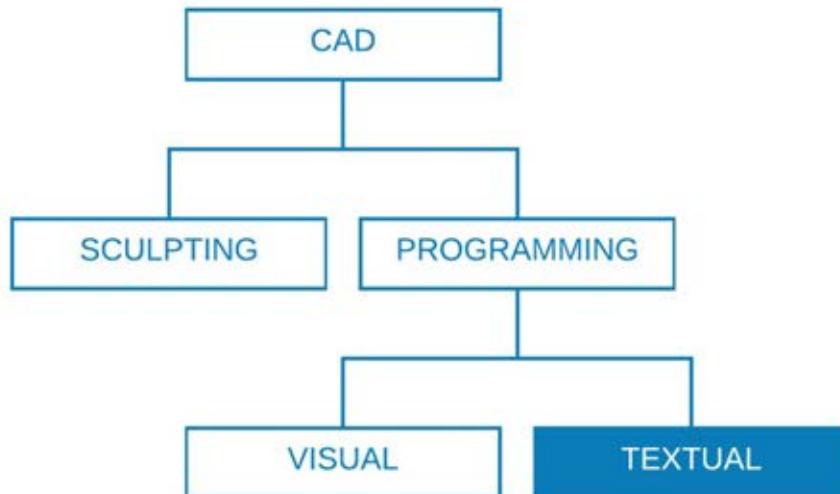


屋根の排水パターンを指定することにより、それぞれのプリセットをさまざまな角度から解析することができます。

# スクリプト作成のガイドライン

## スクリプト作成のガイドライン

ビジュアル スクリプト環境内で、DesignScript、Python、ZeroTouch (C#)を使用してテキストベースのスクリプトを作成すると、機能性の高い視覚的な関係を構築することができます。同じワークスペース内で Python や C# を使用して、入力スライダなどの要素を画面上に表示したり、大規模な操作を DesignScript 内に集約したり、便利なツールやライブラリにアクセスすることができます。こうした方法を組み合わせて効率的に管理することにより、プログラム全体のカスタマイズ性、明確性、効率性が大幅に向上します。ここでは、テキストスクリプトを使用してビジュアル スクリプトを拡張するための一連のガイドラインを紹介します。



### どのような場合にスクリプトを作成するのかを理解する

テキストスクリプトを使用すると、ビジュアルスクリプトよりも複雑な関係を記述することができます。ただし、テキストスクリプトとビジュアルスクリプトの機能は多くの点で重複しています。事前にパッケージ化された便利なコードがビジュアルスクリプトのノードであるため、ビジュアルスクリプトで多くの機能が重複するのは避けられないことです。ビジュアルスクリプトを使用すると、Dynamo プログラム全体を DesignScript または Python で作成することができます。ただし、ノードとワイヤのインターフェースを使用すると、グラフィカルな情報を直感的なフローで作成できるため、Dynamo ではビジュアルスクリプトを使用しています。テキストスクリプトの機能がビジュアルスクリプトの機能よりも優れている点について理解しておくと、ノードやワイヤの直感的な機能の代わりにテキストスクリプトの機能を使用した方が効率的な場合を特定できるようになります。どのような場合にどの言語を使用してスクリプトを作成するかについては、次に示すガイドラインを参照してください。

#### テキストスクリプトを使用するケース:

- ループ処理を記述する場合
- 反復処理を記述する場合
- 外部ライブラリへのアクセス処理を記述する場合

#### 言語の選択

| |ループ|反復|ノードの集約|外部 ライブラリへのアクセス|省略表記| | -- | - - | - - | - - | - - | |DesignScript|可|可|可|可|不可|可|  
|Python|可|可|一部可|可|不可| |ZeroTouch (C#)|不可|不可|不可|可|不可|

各 Dynamo ライブラリでアクセスできるコンテンツのリストについては、「[スクリプトの参照情報](#)」を参照してください。

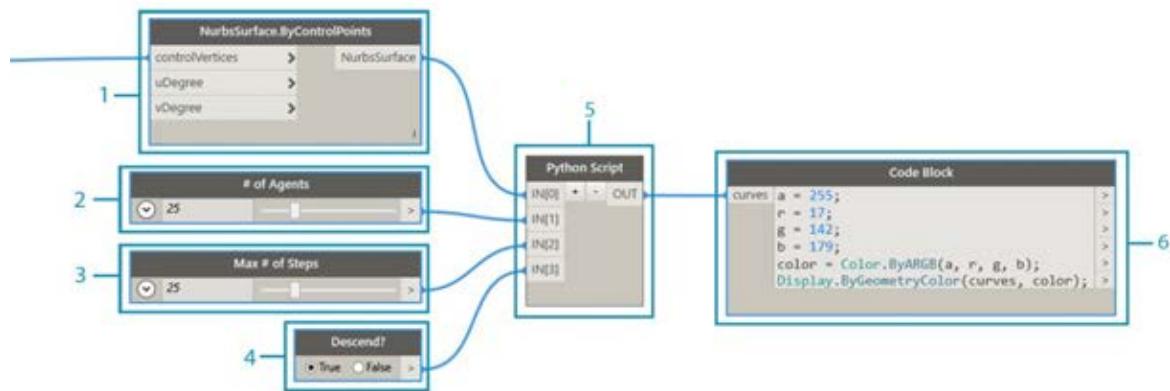
#### パラメトリックに考える

Dynamo でスクリプトを作成する場合は必ずパラメトリック環境で作業を行うことになるため、使用するノードとワイヤのフレームワークに関連する形で、構造化されたコードを記述することをお勧めします。例として、テキストスクリプトが記述されているノードを考えてみます。このノードは、複数の入力、1つの関数、予想される1つの出力が指定されている、プログラム内の他のノードと同じように

構成されているとします。この場合、ノード内のコードにいくつかの作業用変数が即時に設定されます。これらの変数が、クリーンなパラメトリック システムの重要な要素になります。コードをビジュアル プログラムに統合するための適切な方法については、次のガイドラインを参照してください。

#### 外部変数を特定する

- 設計内で使用するパラメータを特定します。これにより、対象のデータから直接作成されるモデルを構築できるようになります。
- コードを記述する前に、次の変数を特定します。
- 入力の最小セット
- 目的の出力
- 定数



一部の変数については、コードを記述する前に既に定義されています。

1. 降雨のシミュレーションを実行するサーフェス。
2. 必要な雨粒の数(エージェント)。
3. 雨粒の移動距離。
4. 最も急なパスの下降とサーフェス横断との切り替え。
5. 各入力の番号が指定された Python Script ノード。
6. 返された曲線が青で表示されるコード ブロック。

#### 内部的な関係を設計する

- パラメトリズム機能で特定のパラメータや変数を編集することにより、計算式やシステムの最終結果を操作または変更できます。
- スクリプト内のエンティティが論理的に関連付けられている場合は、常にそれらのエンティティを相互の関数として定義します。この場合、一方の関数を変更すると、その変更内容に応じてもう一方の関数が更新されます。
- 重要なパラメータだけを表示し、入力の数を最小限に抑えます。
- 追加の親パラメータから一連のパラメータを継承できる場合は、それらの親パラメータだけをスクリプトの入力として表示してください。こうすることにより、インターフェースの複雑さが軽減されるため、スクリプトが使いやすくなります。

```

1 import clr
2 clr.AddReference('ProtoGeometry')
3 from Autodesk.DesignScript.Geometry import *
4
5 solid = IN[0]
6 seed = IN[1] ← 1
7 xCount = IN[2]
8 yCount = IN[3]
9
10 solids = []
11 yDist = solid.BoundingBox.MaxPoint.Y - solid.BoundingBox.MinPoint.Y ← 2
12 xDist = solid.BoundingBox.MaxPoint.X - solid.BoundingBox.MinPoint.X
13
14 for i in xRange:
15     for j in yRange:
16         fromCoord = solid.ContextCoordinateSystem
17         toCoord = fromCoord.Rotate(solid.ContextCoordinateSystem.Origin, Vector.ByCoordinates(0, 0, 1), (90 * (i + j * val))) ← 3
18         vec = Vector.ByCoordinates((xDist * i), (yDist * j), 0)
19         toCoord = toCoord.Translate(vec)
20         solids.append(solid.Transform(fromCoord, toCoord))
21
22 OUT = solids

```

上記のコードは、[Python Script ノード](#)の「modules」というサンプル コードです。

1. 入力。
2. スクリプト内で使用される変数。
3. 上記の入力と変数を使用して関数を実行するループ。ヒント: スクリプトの結果だけでなく、途中のプロセスも非常に重要です。どちらにも同じように重点を置いてください。

#### Don't repeat yourself (DRY 原則)

- スクリプト内で同じ処理を複数の方法で記述すると、いずれかのタイミングで、重複するコード記述間における整合性が取れなくなることがあります。その結果、保守作業に時間がかかったり、コードの解析効率が落ちたり、システム内部の不整合が発生する可能性があります。
- DRY 原則は、「すべての情報は、システム内において、ただ 1 つの明確な信頼できる表現を持っている」ということを表す原則です。
- この原則を正しく適用することにより、スクリプト内の関連要素が予期したとおりに統一して変更されます。関連しない要素については、相互に論理的な影響が及ぶことはありません。

```

### 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)

```

ヒント: 上記の「BAD」の例では定数を記述していますが、スクリプト内でエンティティを複製する前に、ソースにリンクさせることができないかどうかを検討することをお勧めします。

#### モジュール単位で構成する

コードが長くなってより複雑になるにつれて、スクリプトの目的や全体的なアルゴリズムが分かりにくくなります。また、特定の処理の内容や、特定の処理が記述されている箇所を簡単に追跡できなくなり、エラーが発生した場合も、その原因となるバグを見つけることが難しくなります。さらに、他のコードへの統合や開発作業の割り当てについても、困難になります。こうした困難を避けるため、モジュール単位でコードを記述することをお勧めします。この方法では、コードで実行する処理に基づいて、コードを組織的に分割して記述します。ここでは、モジュール化の手法を使用して、スクリプトを簡単に管理するためのヒントをいくつか紹介します。

#### モジュール単位でコードを記述する

- 「モジュール」とは、特定の処理を実行するコードの集まりで、ワークスペース内の Dynamo ノードに似ています。

- モジュールは、隣接するコードから視覚的に離れた位置に記述します(関数、クラス、入力のグループ、読み込むライブラリなどがモジュールになります)。
- モジュール単位でコードを記述すると、視覚的で直感的なノードの利点を活用できるだけなく、テキストスクリプトでしか記述できない複雑な関係を使用することもできます。

```

69
70 # INITIALIZE AGENTS
71 agents = []
72 for i in range(numAgents):
73     u = float(random.randrange(1000))/1000
74     v = float(random.randrange(1000))/1000
75     agent = Agent(u,v)
76     agents.append(agent)
77
78 # UPDATE AGENTS
79 for i in range(maxSteps):
80     for eachAgent in agents:
81         eachAgent.update()
82
83 # DRAW TRAILS
84 trails = []
85 for eachAgent in agents:
86     trailPts = eachAgent.trailPts
87     if (len(trailPts) > 1):
88         trail = PolyCurve.ByPoints(trailPts)
89         trails.append(trail)
90
91 # OUTPUT TRAILS
92 OUT = trails

```

上記のループ処理では、「agent」というクラスを呼び出しています。このクラスは、次の演習で作成します。

- 各エージェントの開始点を定義するコード モジュール。
- エージェントを更新するコード モジュール。
- エージェントのパスの基準線を描画するコード モジュール。

複数の箇所に記述されている同じコードを関数としてまとめる

- 同じ処理(または非常に類似した処理)を実行するコードが複数の箇所に記述されている場合は、呼び出し可能な関数としてそれらのコードをまとめます。
- 「Manager」関数は、プログラム フローをコントロールします。この関数には主に、構造間でのデータ移動などの低レベル詳細の動作を処理する「Worker」関数に対する呼び出しが格納されます。



上記の例では、中心点の Z 値に基づき、半径と色を指定して球体を作成しています。

- 中心線の Z 値に基づき、半径と表示色を指定して球体を作成する 2 つの「Worker」親関数。
- 2 つの Worker 関数を組み合わせる「Manager」親関数。この関数を呼び出すと、この関数に含まれている 2 つの Worker 関数が呼び出されることになります。

必要な要素だけを表示する

- モジュールのインターフェースには、そのモジュールで指定されている必要な要素が表示されます。
- ユニット間のインターフェースを定義すると、各ユニットの詳細設計を個別に進めることができます。

モジュールの分離や置き換え

- モジュール間には相互の関係はないため、必要に応じて分離や置き換えを行うことができます。

モジュール化の一般的な形式

- コードをグループ化する場合は、次のように記述します。

```

# 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]

• 関数は、次のように記述します。

def get_step_size():
area = surfIn.Area
stepSize = math.sqrt(area)/100
return stepSize

stepSize = get_step_size()

• クラスは、次のように記述します。

class MyClass:
i = 12345

def f(self):
return 'hello world'

numbers = MyClass.i
greeting = MyClass.f

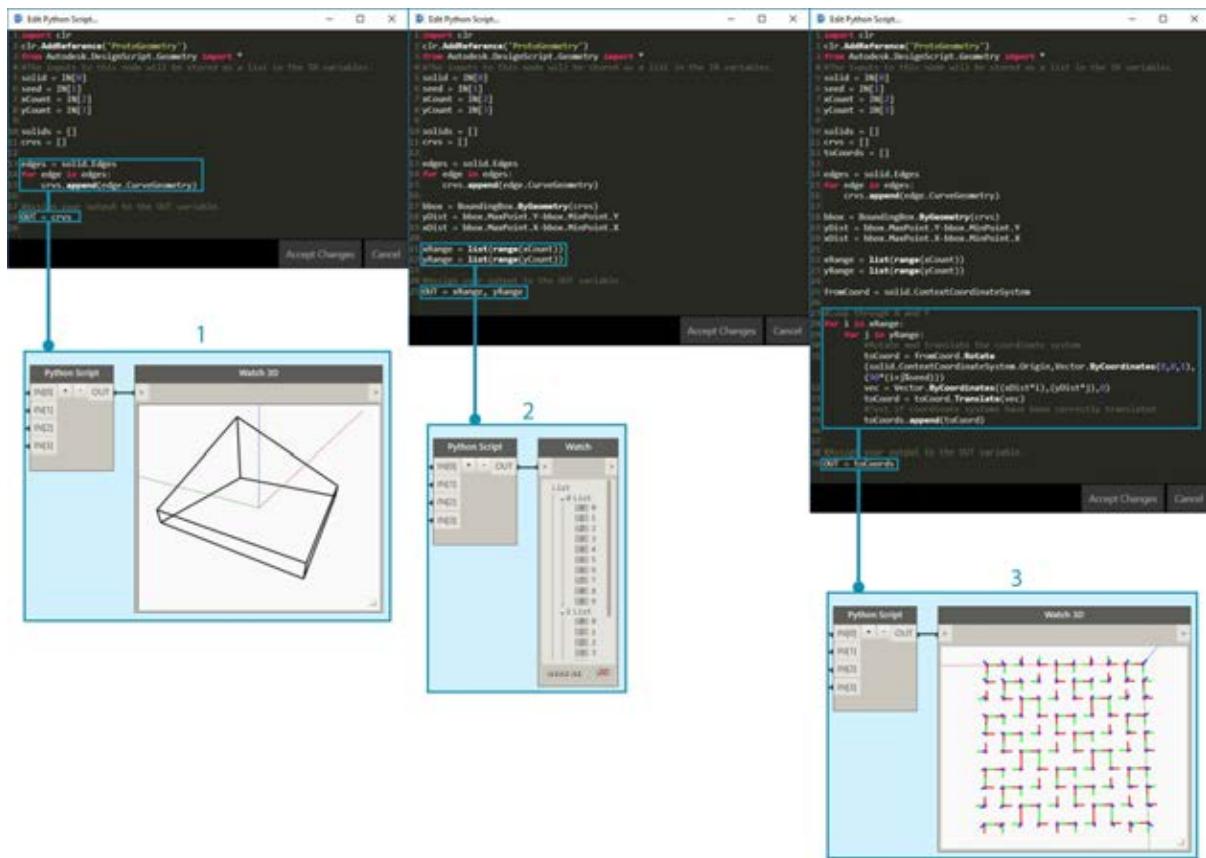
```

### スクリプトを継続的にモニタリングする

Dynamo でテキストスクリプトを開発する場合は、目的とする結果と、実際に作成される結果を頻繁に確認することをお勧めします。こうすることにより、構文エラー、論理的な矛盾点、不正確な値、正しくない出力など、予期しない状況をすばやく検出し、そのたびに修正することができます。最後にまとめて修正するよりも、効率的に作業を進めることができます。テキストスクリプトは、キャンバス上のノード内でアクティブになっているため、ビジュアルプログラムのデータフローに既に統合された状態になっています。そのため、出力データを割り当ててプログラムを実行し、Watch ノードを使用してスクリプトからのフローを評価するだけで、スクリプトを継続的にモニタリングすることができます。ここでは、スクリプトを作成しながら、そのスクリプトを継続的に検査する場合のヒントをいくつか紹介します。

### スクリプトを開発しながらテストを行う

- 機能を実行するためのコードの記述が完了するたびに、次の点を確認します。
- コードの内容を客観的に確認します。
- コードの内容を批判的に確認します。たとえば、共同作業を行っている他のメンバーがこのコードの内容を理解できるかどうか、この処理は本当に必要なかどうか、この機能をさらに効率的に記述することができないかどうか、不要な重複コードや依存コードが記述されていないかどうか、などを確認します。
- 正しいデータが返されるかどうかを簡単にテストします。
- スクリプト内で使用する最新のデータを出力として割り当てます。これにより、スクリプトを更新するたびに、関連するデータがノードによって常に出力されるようになります。



[Python ノート](#)のサンプル コードで、次の動作を確認します。

- ソリッドのすべてのエッジが、境界ボックスを作成するための曲線として返されるかどうかを確認します。
- Count 入力が正しく Range に変換されるかどうかを確認します。
- 座標系が適切に変換され、ループ内で回転するかどうかを確認します。

極端な条件を想定してテストを行う

- スクリプトの作成中に、割り当てられているドメインの最小値と最大値に入力パラメータを設定すると、極端な条件下でもプログラムが正常に機能するかどうかを確認することができます。
- 最小値や最大値を指定して正常にプログラムが機能する場合であっても、予期しない Null 値、空の値、ゼロが返されることがないかどうかを確認する必要があります。
- こうした極端な条件を設定しないと、スクリプト内に存在する問題を示すバグやエラーが見つからないことがあります。
- エラーの原因を特定し、エラーを内部的に修正する必要があるのかどうか、またはパラメータ ドメインを調整してエラーを回避する必要があるのかどうかを判断します。

ヒント: スクリプトを開発する際には、「ユーザは、使用可能なすべての入力値のあらゆる組み合わせを使用する」ということを常に想定して作業を行う必要があります。これにより、予期しない問題が発生するのを防ぐことができます。

## 効率的なデバッグを行う

デバッグとは、スクリプトからバグをなくすためのプロセスのことです。エラー、非効率な処理、不正確な値、予期しない結果などをまとめて「バグ」と呼びます。バグの修正は、変数名のスペルミスを修正するような単純なものもあれば、スクリプトの構成に関する問題のように、広範囲にわたるものもあります。スクリプトを作成しながら、早い段階で潜在的な問題を検出して修正するのが理想的な方法ですが、この方法でも、バグのまったくないスクリプトを作成できるという保証はありません。ここでは、バグを系統的な方法で修正するためのベスト プラクティスをいくつか紹介します。これらのベスト プラクティスは、優先度の高い順に記載されています。

### ウォッチ バルーンを使用する

- 返されるデータを OUT 変数に割り当てて、コード内の複数の場所でそのデータを確認します。これは、スクリプトを作成しながらバグを修正する方法に似ています。

### 分かりやすいコメントを書く

- コードをモジュール化して、目的とする結果をコメントとして明確に記述すると、デバッグが簡単になります。

```
# Loop through X and Y
for i in range(xCount):
    for j in range(yCount):

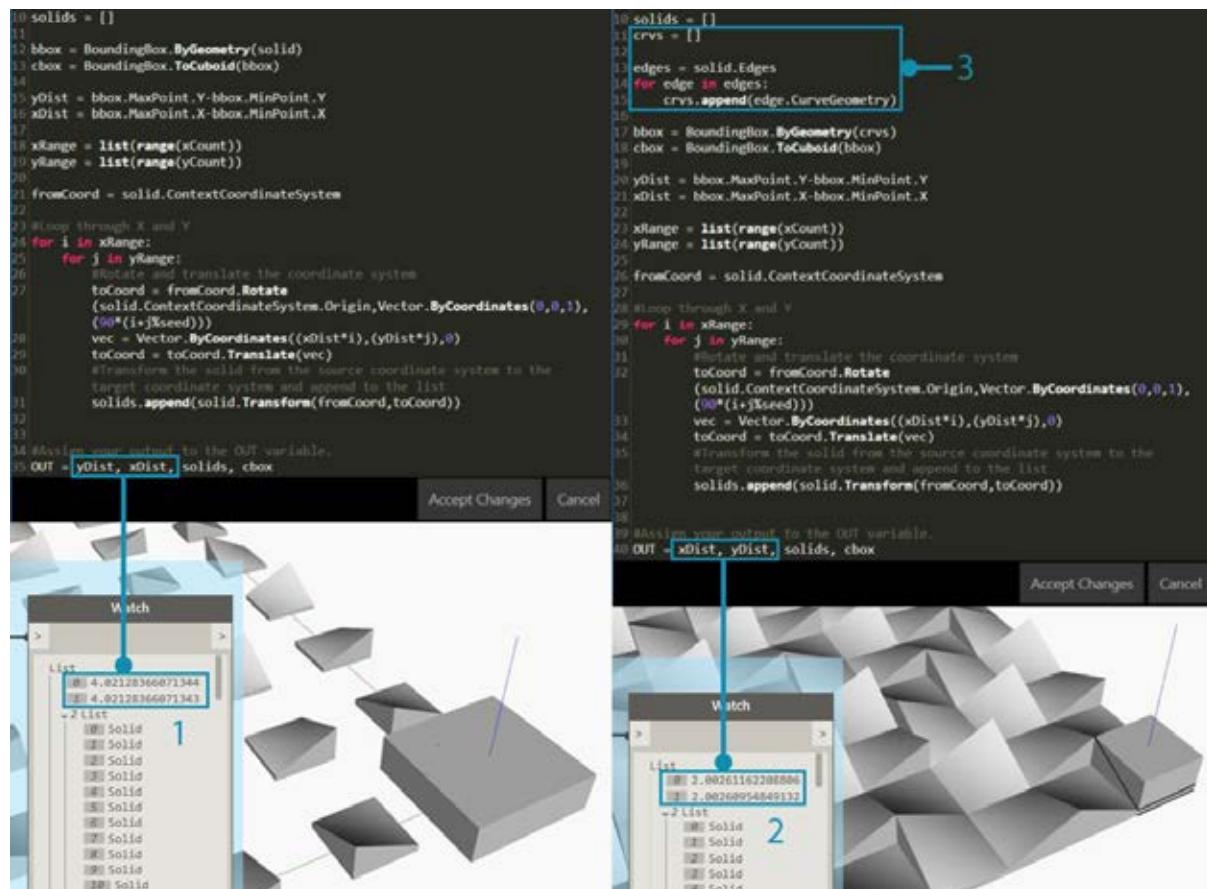
        # Rotate and translate the coordinate system
        toCoord = fromCoord.Rotate(solid.ContextCoordinateSystem.Origin, Vector.ByCoordinates
        vec = Vector.ByCoordinates((xDist*i), (yDist*j), 0)
        toCoord = toCoord.Translate(vec)

        # Transform the solid from the source coord system to the target coord system and ap
solids.append(solid.Transform(fromCoord,toCoord))
```

通常、コメント行と空白行は余分な情報ですが、デバッグを行う場合は、コードを細分化して内容を追跡できるため、便利です。

モジュール化されたコードの利点を活用する

- 分離された特定のモジュール内に問題の原因が存在している場合があります。
- 問題のあるモジュールを特定できれば、問題の修正が非常に簡単になります。
- プログラムを修正する必要がある場合、コードがモジュール化されていれば、非常に簡単に修正を行うことができます。
- プログラムの他の部分を変更することなく、新しいモジュールやデバッグ済みのモジュールを既存のプログラムに挿入することができます。



[Python ノード](#)のサンプル ファイルをデバッグすると、次のようにになります。

- 自分よりも大きな境界ボックスを返す入力ジオメトリ(これを確認するには、xDist と yDist の値を OUT 変数に割り当てます)。
- xDist と yDist の正しい距離が設定された適切な境界ボックスを返す入力ジオメトリのエッジ曲線。
- xDist と yDist の値に関する問題に対応するための「module」コード(これは、これまでの説明で挿入したコードです)。

## 演習 - 最も急なパスを作成する

この演習に付属しているサンプル ファイルをダウンロードしてください(右クリックして[名前を付けてリンク先を保存]を選択)。すべてのサンプル ファイルの一覧については、付録を参照してください。[SteepestPath.dyn](#)

上で説明したテキスト スクリプトのベスト プラクティスを意識しながら、降雨シミュレーション スクリプトを作成してみましょう。「見やすいプログラムを作成するためのガイドライン」の演習では、ベスト プラクティスに従い、構造化されていないビジュアル プログラムを見やすく整理することができましたが、テキスト スクリプトの場合、それは非常に難しくなります。テキスト スクリプトで記述された論理的な関係は、視覚的に把握するのが難しいため、複雑なコードになると、関係を解読するのがほとんど不可能になる場合もあります。そのため、テキスト スクリプトの高度な機能を使用する場合は、組織的に記述する必要性が高くなります。この演習では、ステップごとに操作を確認しながら、ベスト プラクティスについて説明していきます。



アトラクタが変形しているサーフェスに適用されるスクリプト

最初に、必要な Dynamo ライブラリを読み込みます。これにより、Python 内で Dynamo の機能にグローバルにアクセスできるようになります。



この段階で、必要なライブラリをすべて読み込む必要があります。

次に、スクリプトの入力と出力を定義する必要があります。定義した入力は、ノード上で入力ポートとして表示されます。これらの外部入力が、このスクリプトの基礎となり、パラメトリック環境を構築するための重要なデータになります。



Python スクリプト内の変数に対応する入力を定義する必要があります。これらの入力により、目的とする出力が設定されます。

1. ベースとなるサーフェス。
2. 使用するエージェントの数。
3. エージェントが進むことができるステップの最大数。
4. サーフェスからの最短距離のパスを取得するためのオプション(または、サーフェスを横断するためのオプション)。
5. スクリプト IN[0] とスクリプト IN[1] の入力に対応する入力 ID が設定された Python ノード。
6. 異なる色で表示可能な出力曲線。

では、モジュール化の手法を使用して、スクリプトの本文を作成してみましょう。複数の始点に対して、サーフェスからの最短パスのシミュレーションを実行するのは、複数の関数が必要になる複雑な作業です。そのため、スクリプト内の異なる場所で異なる関数を呼び出すのではなく、それらの関数をエージェントの 1 つのクラスとしてまとめることにより、コードをモジュール化します。このクラス(モジュール)の各関数は、それぞれ異なる変数を使用して呼び出すことができます。また、別のスクリプト内で再利用することもできます。



サーフェス上を下降するエージェントに対して、その動作を指示するクラスを定義する必要があります。このクラスを使用して、エージェントがステップを進むたびに、最も急な方向へ移動することを選択します。

1. 名前。
2. すべてのエージェントが共有するグローバル属性。
3. 各エージェント固有のインスタンス属性。
4. ステップを進むための関数。
5. 各ステップの位置を基準線リストにカタログ化するための関数。

ここで、エージェントの開始位置を指定して、エージェントを初期化しましょう。それに合わせて、スクリプトの調整を行い、エージェントクラスが正しく機能するかどうかを確認します。



サーフェスを下降するすべてのエージェントをインスタンス化し、それらのエージェントの初期属性を定義する必要があります。

1. 新しい空の基準線リスト。
2. サーフェス上での下降の開始を指定するためのコード。
3. エージェントリストを出力として割り当て、スクリプトから返される情報を確認するための Watch ノード。エージェントの正確な数が返されていますが、後でスクリプトをもう一度編集して、スクリプトから返されるジオメトリを確認します。

ステップごとにそれぞれのエージェントを更新します。



次に、エージェントとステップごとに、ネストされたループ処理を記述し、エージェントとステップの位置を更新してそれぞれの基準線リストに記録します。また、各ステップで、エージェントがサーフェス上の最終的な位置(それ以上は下降できない位置)にまだ到達していないことを確認します。エージェントがサーフェス上の最終的な位置に到達したら、エージェントの移動を停止します。

これで、各エージェントが完全に更新されました。次に、これらのエージェントを表すジオメトリを返してみましょう。



すべてのエージェントが、それ以上は下降できない位置にまで達したことを確認した後で(または、ステップの最大数に達したことを確認した後で)、各エージェントの基準線リスト内の点を通過するポリカーブを作成し、そのポリカーブの基準線を出力します。

最も急なパスを探すためのスクリプト。



1. ベースとなるサーフェス上の降雨をシミュレーションするためのプリセット。
2. 最も急なパスを探す代わりに、エージェントを切り替えて、ベースとなるサーフェスを横断することもできます。



最終的な Python のテキスト スクリプトはこのようになります。

# スクリプト リファレンス

## スクリプト リファレンス

このリファレンス ページでは、「スクリプト作成のガイドライン」のページで紹介したベスト プラクティスを、コード ライブリ、ラベル付け、スタイル設定によってさらに拡張する方法について説明します。ここでは、Python を使用して次に示す概念を説明しますが、Python だけでなく C#(Zerotouch)についても、同じ概念が異なる構文で適用されます。

### Dynamo ライブリと標準ライブリのどちらを使用するか

標準ライブリは Dynamo の外部に存在し、プログラミング言語の Python や C# (Zerotouch)の内部で使用されます。Dynamo には、ノード階層に直接対応する専用のライブリ セットも用意されています。これらの専用ライブリにより、ノードとワイヤを使用して作成したコード内で、あらゆる処理を記述することができます。ここからは、各 Dynamo ライブリの内容と、どのような場合に標準ライブリを使用するかについて説明します。



#### 標準ライブリと Dynamo ライブリ

- Dynamo 環境内で Python と C# の標準ライブリを使用すると、高度なデータ構造とフロー構造を定義することができます。
- Dynamo ライブリは、ジオメトリなどの Dynamo オブジェクトを作成するためのノード階層に直接対応しています。

#### Dynamo ライブリ

- ProtoGeometry
- 機能: 円弧、境界ボックス、円、円錐、座標系、立方体、曲線、円柱、エッジ、橙円、橙円弧、面、ジオメトリ、らせん、インデックス グループ、線分、メッシュ、NURBS 曲線、NURBS サーフェス、平面、点、ポリゴン、長方形、ソリッド、球体、サーフェス、トポロジ、T スプライン、UV、ベクトル、頂点。
- 読み込み方法: `import Autodesk.DesignScript.Geometry`
- Python または C# で ProtoGeometry を使用すると、非管理オブジェクト(メモリを手動で管理する必要があるオブジェクト)が作成されることに注意してください。詳細については、「非管理オブジェクト」のセクションを参照してください。
- DSCoreNodes
- 機能: 色、色範囲(2D)、日時、期間、I/O、式、ロジック、リスト、数式、クアッドツリー、文字列、スレッド。
- 読み込み方法: `import DSCore`
- モザイク模様
- 機能: 凸型ハル、ドローネー、ボロノイ図。
- 読み込み方法: `import Tessellation`
- DSOffice
- 機能: Excel。
- 読み込み方法: `import DSOffice`

#### 分かりやすい名前を付ける

スクリプトを作成する際には、頻繁に識別子を使用して、変数、タイプ、関数などのエンティティを区別する必要があります。アルゴリズムの作成時に、文字列から構成されるラベルを付けることにより、アルゴリズムの内容が分かりやすくなります。コードを記述する際に分かりやすい名前を付けると、他のメンバーがそのコードの内容を簡単に理解できるようになるだけでなく、後から自分でコードを読む場合にも役立ちます。スクリプト内でエンティティに名前を付ける場合のヒントをいくつか紹介します。

略語を使用する場合は、その略語の意味をコメントで記述する:

```
### BAD
CSfX = 1.6
CSfY= 1.3
CSfZ = 1.0
```

```
### GOOD
# column scale factor (csf)
csfX = 1.6
csfY= 1.3
csfZ = 1.0
```

冗長な名前は付けない:

```
### BAD
import car
seat = car.CarSeat()
tire = car.CarTire()
```

```
### GOOD
import car
seat = car.Seat()
tire = car.Tire()
```

If 文で変数名を記述する場合は、If not 文を使用しない:

```
### 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'
```

単語を連結した変数を使用する場合は、「主語\_形容詞」という順にする:

```
### BAD
agents = ...
active_agents = ...
dead_agents ...
```

```
### GOOD
agents = ...
agents_active = ...
agents_dead = ...
```

このように形式を統一すると、意味がさらに分かりやすくなります。

長い名前を繰り返し使用する場合は、エイリアスを使用する:

```
### 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
```

ただし、頻繁にエイリアスを使用すると、非常に混乱した分かりにくいプログラムになる場合があります。

必要な単語だけを使用する:

```

### BAD
rotateToCoord = rotateFromCoord.Rotate(solid.ContextCoordinateSystem.Origin,Vector.E

### GOOD
toCoord = fromCoord.Rotate(solid.ContextCoordinateSystem.Origin,Vector.ByCoordinates

「ものごとはできるかぎりシンプルにすべきだ。しかし、シンプルすぎてもいけない」— アルベルト・aignシュタイン

```

## スタイルを統一する

プログラムを作成する方法は 1 つだけではないため、スクリプトを記述する場合は「個人的なスタイル」を基準として、さまざまことを判断しながらスクリプトを記述していくことになります。そのため、コード内のスタイルが統一されているかどうか、一般的な形式のスタイルに準拠しているかどうかが、コードの読みやすさと保守作業の容易さに直接影響することになります。一般的に、同じコードが 2 箇所で記述されている場合、それらのコードは同じように機能すると考えられます。ここで、スタイルが統一された分かりやすいコードを記述するためのヒントをいくつか紹介します。

**命名規則:** (次に示すいずれかの命名規則に従って、コード内の各種タイプのエンティティに名前を付けてください)

- 変数、関数、メソッド、パッケージ、モジュール:  
`lower_case_with_underscores` (小文字の単語を下線で連結する)
- クラス、例外:  
`CapWords` (先頭が大文字の単語をそのまま連結する)
- 保護されているメソッド、内部関数:  
`_single_leading_underscore (self, ...)` (先頭に下線を指定し、小文字の単語を下線で連結する)
- プライベートメソッド:  
`__double_leading_underscore(self, ...)` (先頭に下線を 2 つ指定し、小文字の単語を下線で連結する)
- 定数:  
`ALL_CAPS_WITH_UNDERSCORES` (すべて大文字の単語を下線で連結する)

**ヒント:** 非常に短いコード ブロック内で、明確な文脈で意味がはっきりと理解できる場合を除き、1 文字だけの変数は使用しないでください(特に、I、O、I の使用は避けてください)。

## 空白行を使用する:

- 最上位関数とクラス定義の前後に、2 行の空白行を挿入してください。
- クラス内に記述されているメソッド定義の場合は、その前後に空白行を 1 行挿入します。
- 空白行を挿入して、関連する関数グループを区別してもかまいません(ただし、あまり多くの空白行を挿入しないようにしてください)。

## 余分な空白を挿入しない:

- 丸括弧、角括弧、中括弧の内側に、空白を挿入しないでください。

```
### BAD
function( apples[ 1 ], { oranges: 2 } )
```

```
### GOOD:
function(apples[1], {oranges: 2})
```

- カンマ、セミコロン、コロンの直前に、空白を挿入しないでください。

```
### BAD
if x == 2 : print x , y ; x , y = y , x
```

```
### GOOD
if x == 2: print x, y; x, y = y, x
```

- 関数呼び出しの引数リストが記述されている括弧の直前に、空白を挿入しないでください。

```
### BAD
function (1)
```

```

### GOOD
function(1)

• インデックスまたはスライシングが記述された括弧の直前に、空白を挿入しないでください。

### BAD
dict ['key'] = list [index]

### GOOD
dict['key'] = list[index]

• 次に示すバイナリ演算子については、必ず前後に空白を 1 つずつ挿入してください。

assignment ( = )
augmented assignment ( += , -= etc.)
comparisons ( == , < , > , != , <> , <= , >= , in , not in , is , is not )
Booleans ( and , or , not )

```

行の長さに注意する:

- 1 行の長さは 79 文字以内にしてください。
- 使用するエディタ ウィンドウの幅を制限することにより、複数のファイルを並べて表示できるようになるため、2 つのバージョンを並べて表示するコード レビュー ツールを使用して作業を行う場合に便利です。
- 複数の式が記述されている長い行の場合は、式を丸括弧で囲んで改行すると、1 行を複数の行に分割することができます。

余分なコメントや冗長なコメントを記述しない:

- コメントを少なくした方が、コードが読みやすくなる場合があります。特に、コメントの代わりに分かりやすい名前を使用すると効果的です。
- 適切な方法でコードを記述することにより、必要以上にコメントを入力することがなくなります。

```

### 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()

```

ヒント: コードの内容が簡単に理解できるようなコメントを入力し、処理内容が明確に理解できるようなコードを記述することが重要です。

オープンソースコードを確認する:

- オープンソースプロジェクトというプロジェクトがありますが、これは、多くの開発者が協力して立ち上げたプロジェクトです。これらのプロジェクトでは、チーム内で可能な限り効率的に作業を進めることができるように、読みやすいコードを記述することが求められます。そのため、これらのプロジェクトで使用されているソースコードを参照すると、開発者がどのようなコードを記述しているのかを確認することができます。
- 次の項目を確認することにより、命名規則を改善することができます。
- ニーズに適した命名規則になっているかどうか。
- 命名規則が原因で、機能や効率性に影響していないかどうか。

## C# (Zerotouch)の標準

次に示す各 Wiki ページで、Dynamo で C# (Zerotouch)を記述する場合のガイド情報を参照することができます。

- この Wiki ページには、コードのドキュメント化とテストを行う場合の一般的なコーディング標準が記載されています:  
<https://github.com/DynamoDS/Dynamo/wiki/Coding-Standards>
- この Wiki ページには、ライブラリ、カテゴリ、ノード、ポート、略称に関する命名規則の標準が記載されています:  
<https://github.com/DynamoDS/Dynamo/wiki/Naming-Standards>

#### 非管理オブジェクト:

Python または C# で Dynamo のジオメトリ ライブラリ(*ProtoGeometry*)を使用してジオメトリ オブジェクトを作成した場合、それらのオブジェクトは仮想マシンによって管理されないため、それらの多くのオブジェクトについて、メモリを手動でクリーンアップする必要があります。ネイティブ オブジェクトと非管理オブジェクトは、いずれも、**Dispose** メソッドまたは **using** キーワードを使用してクリーンアップすることができます。概要については、次の Wiki エントリを参照してください: <https://github.com/DynamoDS/Dynamo/wiki/Zero-Touch-Plugin-Development#dispose--using-statement>

破棄する必要があるのは、プログラム内に返されることがない非管理オブジェクト、または参照情報が格納されない非管理オブジェクトだけです。ここからは、こうしたオブジェクトのことを「中間ジオメトリ」と呼ぶことにします。次のサンプル コードでは、例として、このようなオブジェクトのクラスが記述されています。このコードに記述されている Zero Touch の C# 関数である **singleCube** は、立方体を 1 つだけ返しますが、この関数の実行中に 10000 個の立方体が作成されます。この処理は、他のジオメトリが一部の中間構築ジオメトリとして使用されたものと想定することができます。

この **Zero Touch** 関数を実行すると、かなりの確率で **Dynamo** がクラッシュします。このコードでは 10000 個の立方体が作成されますが、格納されてプログラムに返されるのは、そのうちの 1 つだけです。代わりに、プログラムに返す 1 つだけを除いて、残りの中間立方体をすべて破棄する必要があります。プログラムに返す立方体は、プログラム内に伝播されて他のノードで使用されるため、ここでは破棄しません。

```
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;
}
```

修正後のコードは次のようになります。

```
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;
}
```

通常、破棄する必要があるジオメトリは、Surfaces、Curves、Solidsだけです。ただし、念のため、すべてのタイプのジオメトリ (Vectors、Points、CoordinateSystems) を破棄することをお勧めします。

## 付録

### 付録

このセクションでは、Dynamo の理解をさらに深めるための追加のリソースを紹介します。さらに、重要なノードの索引、便利なパッケージのコレクション、この手引で使用しているサンプル ファイルのリポジトリも記載します。他にも必要なリソースがあれば、このセクションに追加してください。[Dynamo Primer](#) は、オープン ソースのドキュメントです。



# リソース

## リソース

### Dynamo Wiki

この Wiki では、Dynamo API を使用して開発を行う方法や、各種のライブラリやツールに関する情報を参照することができます。

<https://github.com/DynamoDS/Dynamo/wiki>

### Dynamo Blog

このブログでは、Dynamo の新機能や新しいワークフローなど、Dynamo 開発チームによるさまざまな最新情報を参照することができます。

<http://dynamobim.com/blog/> (英語)

### DesignScript Guide

プログラミング言語の主要な目的は、ロジックや計算を使用して各種の処理を表現することです。この目的に加えて、Dynamo のテキスト言語(旧 DesignScript)は、設計意図を表現する目的で開発されました。一般的に、計算による設計作業は実験的な試みだと考えられていますが、Dynamo はその取り組みをサポートするためのツールです。Dynamo は、設計上の繰り返し処理を通じて、コンセプト デザインから最終的な詳細設計まで対応できる柔軟で扱いやすい言語となることを目指しています。このマニュアルには、プログラミングの経験も建築設計のジオメトリに関する知識もないユーザ向けに構成されています。このマニュアルには、プログラミングと建築設計のジオメトリという 2 つの専門分野に関するさまざまなトピックが記載されています。

[http://dynamobim.org/wp-content/uploads/forum-assets/colin-mccroneautodesk-com/07/10/Dynamo\\_language\\_guide\\_version\\_1.pdf](http://dynamobim.org/wp-content/uploads/forum-assets/colin-mccroneautodesk-com/07/10/Dynamo_language_guide_version_1.pdf) (英語)

### The Dynamo Primer Project

Dynamo Primer は、オートデスクの Matt Jezyk 氏と Dynamo 開発チームによって開始されたオープン ソース プロジェクトです。Dynamo Primer の初版は、Mode Lab によって開発されました。リポジトリをフォークし、コンテンツを追加し、プルリクエストを送信して、開発プロジェクトに参加してください。

<https://github.com/DynamoDS/DynamoPrimer> (英語)

### Zero Touch Plugin Development for Dynamo

このページでは、Zero-Touch インタフェースを使用して C# で Dynamo のカスタム ノードを開発するプロセスについて説明しています。ほとんどの場合、C# の静的メソッドとクラスは、修正せずにそのまま読み込むことができます。ライブラリで関数だけを呼び出す必要があり、新しいオブジェクトを構築する必要がない場合は、静的メソッドを使用する非常に便利です。Dynamo で DLL をロードすると、クラスの名前空間が除去され、すべての静的メソッドがノードとして表示されます。

<https://github.com/DynamoDS/Dynamo/wiki/Zero-Touch-Plugin-Development> (英語)

### Python for Beginners

Python は、インタラクティブ形式の、インタラクティブなオブジェクト指向のプログラミング言語です。この言語には、モジュール、式、動的型付け、高レベルな動的数据型、クラスが組み込まれています。Python は、強力な性能と非常にわかりやすい構文を兼ね備えています。Python には、システムコール、ライブラリ、さまざまなウィンドウ システムに対応するインターフェースが用意されています。C や C++ でさらに機能を拡張することもできます。また、プログラミング可能なインターフェースを必要とするアプリケーションの拡張言語として使用することもできます。Python は、高い可搬性を誇る言語です。多くの UNIX 系 OS、Mac OS、Windows 2000 以降の OS で動作します。次の Python 入門ガイドには、Python の学習に役立つ入門用チュートリアルやリソースのリンクが記載されています。

<https://www.python.org/about/gettingstarted> (英語)

### AForge

AForge.NET は、画像処理、ニューラル ネットワーク、遺伝的アルゴリズム、ファジー論理、機械学習、ロボット工学など、コンピュータビジョンと人工知能の分野における開発者と研究者のために設計されたオープンソースの C# フレームワークです。

<http://www.aforgenet.com/framework/> (英語)

### Wolfram MathWorld

MathWorld は、Eric W. Weisstein 氏監修のもと、多数の寄稿者の協力によって作成されたオンライン数学リソースです。1995 年に初めてオンラインで公開されて以来、MathWorld は数学界と教育界の両方で、数学に関するさまざまな情報を提供してきました。その情報は、あらゆる教育レベルにわたる雑誌や書籍で幅広く参照されています。

<http://mathworld.wolfram.com/> (英語)

## Revit のリソース

### buildz

このブログでは、Revit プラットフォームを中心として、お勧めの活用法が紹介されています。

<http://buildz.blogspot.com/> (英語)

### Nathan's Revit API Notebook

このノートブックは、設計ワークフローにおける Revit API の学習と適用について、他のリソースでは紹介されていない情報を補完するためのリソースです。

<http://wiki.the provingground.org/revit-api>

### Revit Python Shell

RevitPythonShell を使用すると、IronPython インタプリタを Autodesk Revit と Autodesk Vasari に追加することができます。このプロジェクトは Dynamo よりも前に開始されており、Python 開発に関する参考資料として活用することができます。Revit Python Shell プロジェクト(英語): <https://github.com/architecture-building-systems/revitpythonshell> 開発者のブログ(英語): <http://darenatwork.blogspot.com/>

### The Building Coder

業界を牽引する BIM の専門家によって作成された、Revit API ワークフローの信頼できるカタログです。

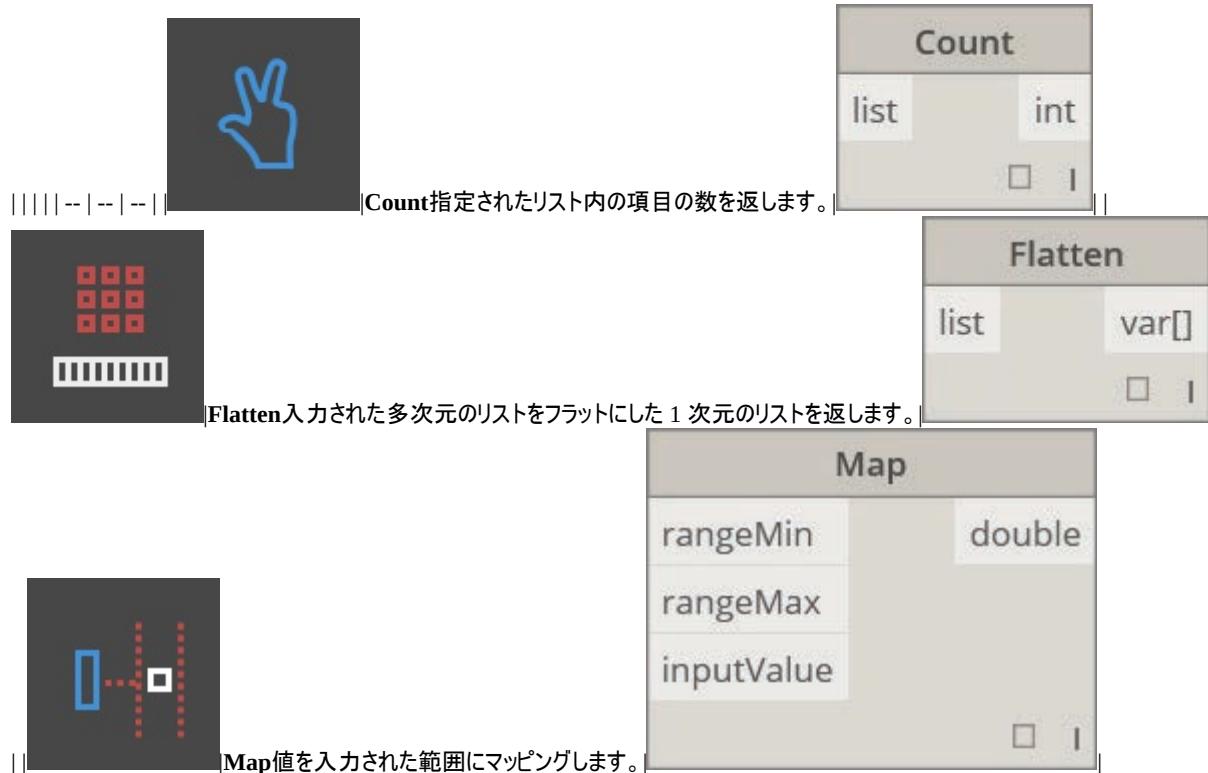
<http://thebuildingcoder.typepad.com/> (英語)

## ノードの索引

## ノードの索引

この索引では、この手引で言及しているすべてのノードと他の便利なコンポーネントについて、補足情報を提供します。ここで紹介するのは、Dynamo で使用できる 500 個のノードのうち一部にすぎません。

## 組み込み関数



## Core

### Core.Color



**Color.ByARGB**

a	Color
r	
g	
b	

Color Range 開始色と終了色間の色のグラデーションから色を取得します。

**Color Range**

colors	color
indices	
value	

|ACTIONS|

|Color.Brightness| Color.Brightness

c	double
---	--------

|Color.Components| Color.Components

c	a
	r
	g
	b

の各成分を、アルファ、赤、緑、青の順のリストとして返します。

|Color.Saturation| Color.Saturation

c	double
---	--------

|Color.Hue| Color.Hue 色の色相の値を取得します。

|Color.Alpha| Color.Alpha 色のアルファ成分の値(0 ~ 1)を取得します。

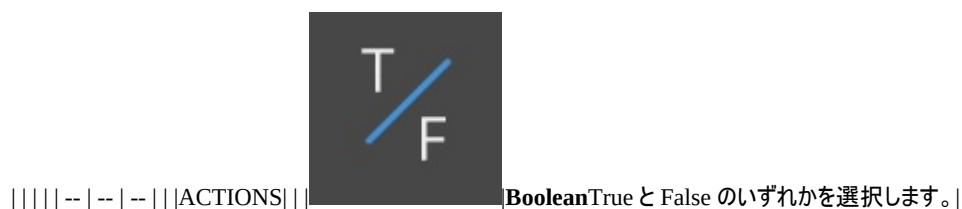
|QUERY|

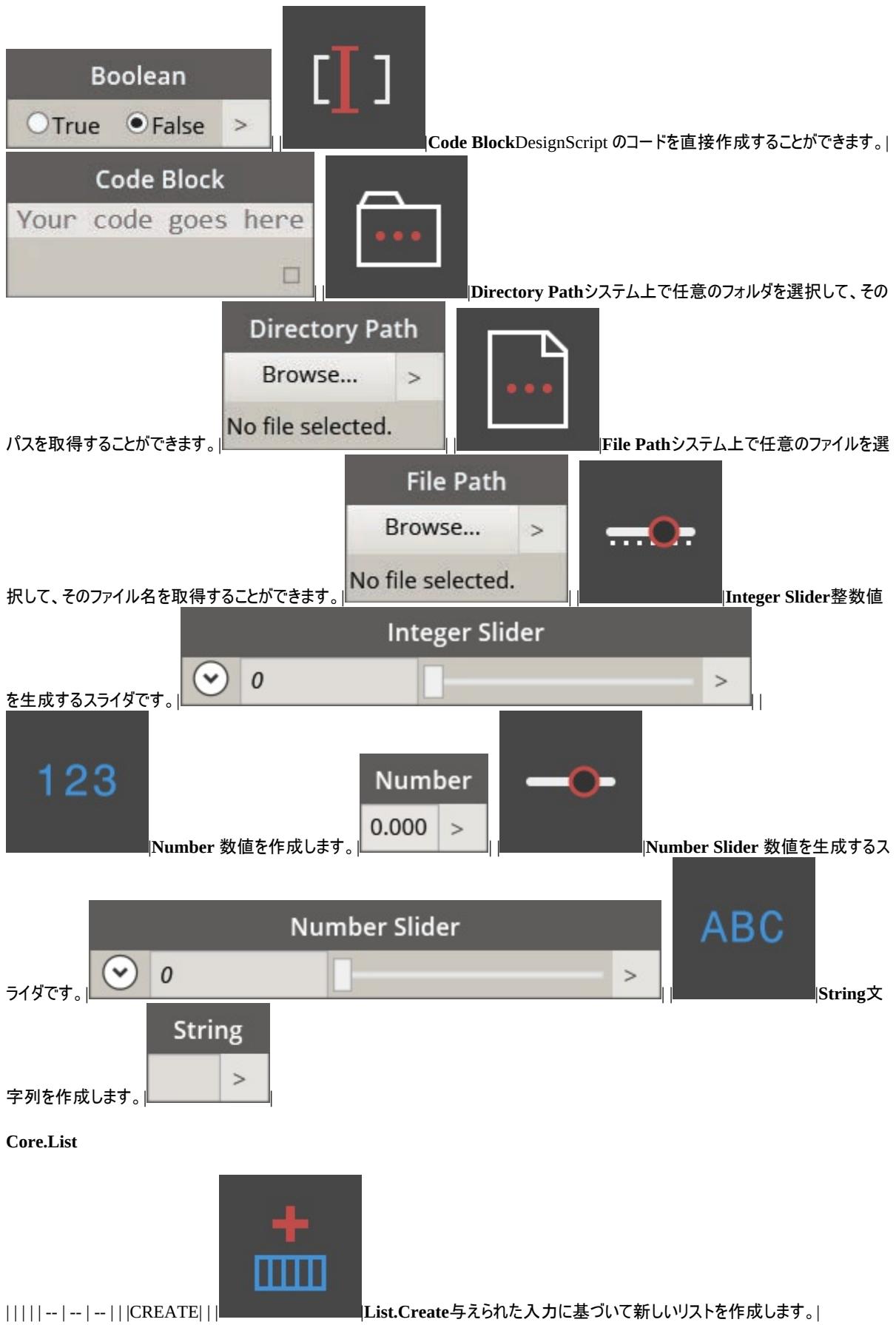


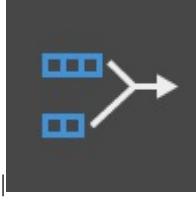
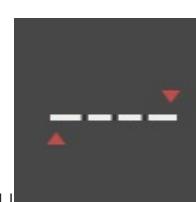
### Core.Display



### Core.Input

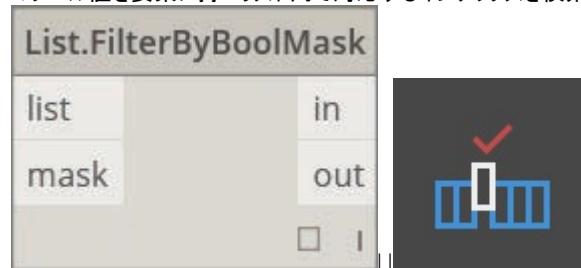




<b>List.Create</b>		List.Combine 2つのシーケンスの各要素にコンビネータを適用します。
<b>List.Combine</b>		
<b>Number Range</b>		Number Range 指定された範囲内で数値のシーケンスを作成します。
<b>Number Sequence</b>		Number Sequence 数値のシーケンスを作成します。
<b>List.Chop</b>		List.Chop リストを、それぞれ指定された個数の項目から成るリストの集合に分割します。
<b>List.Count</b>		List.Count 指定されたりストに格納されている項目の数を返します。
<b>List.Flatten</b>		List.Flatten ネストされたリストのリスト



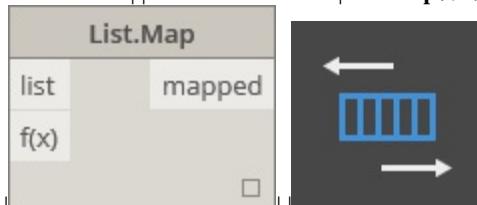
トを、指定された量だけ扁平にします。| List.FilterByBoolMask 別個のブール値を要素を持つリスト内で対応するインデックスを検索して、シーケンスをフィルタします。|



| List.GetItemAtIndex リストの、指定されたインデックスにある項目



を取得します。| List.Map リスト内のすべての要素に関数を適用し、



| List.Reverse 指定されたリスト内の項

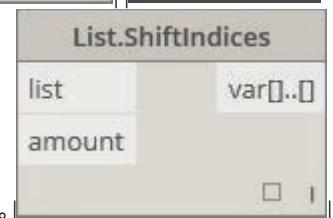
その結果から新しいリストを生成します。| List.Reverse



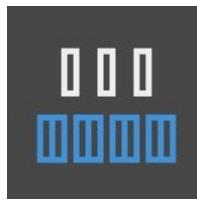
目を逆順で含む新しいリストを作成します。| List.ReplaceItemAtIndex



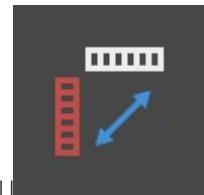
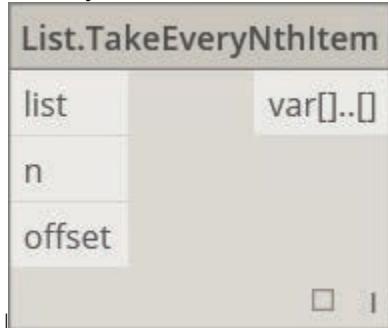
の、指定されたインデックスにある項目を置き換えます。|



| List.ShiftIndices リスト内のインデックスを、指定された量だけ右に移動します。



||| | -- | -- | -- || | ACTIONS || | List.TakeEveryNthItem 指定されたオフセットの後、指定された値の倍数であるインデックスの項目を、指定



されたリストから取得します。List.Transpose 任意のリストのリストの行と列を入れ替えます。他の行よりも短い行がある場合は、作成される配列が常に長方形になるように、プレースホルダーとして

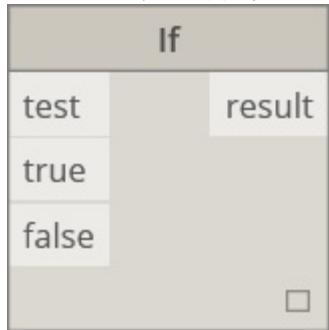


NULL 値が挿入されます。||| | -- | -- | -- || | ACTIONS || |

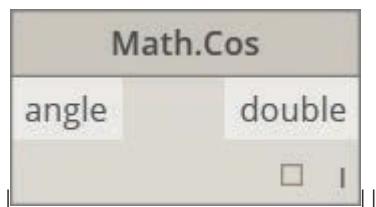
### Core.Logic



||||| -- | -- | -- || | ACTIONS || | If 条件ステートメントです。テスト入力のブール値をチェックします。テスト入力が true である場合は、結果として true の入力を出力します。false である場合は、結果として false の入力を出力します。||



### Core.Math



||||| -- | -- | -- || | ACTIONS || | Math.Cos 角度の余弦を求めます。||



Math.DegreesToRadians 度単位の角度をラジアン単位の角度に変換します。|

Math.DegreesToRadians	
degrees	double
□	



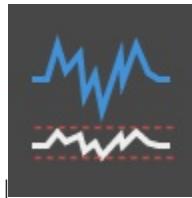
Math.Pow 指定された指数に対して値を累乗します。|

Math.Pow	
number	double
power	double
□	



Math.RadiansToDegrees ラジアン単位の角度を度単位の角度

Math.RadiansToDegrees	
radians	double
□	



Math.RemapRange 分布比率を保持しながら

に変換します。|

Math.RemapRange	
numbers	var[]..[]
newMin	
newMax	
□	



Math.Sin 角度の正弦を

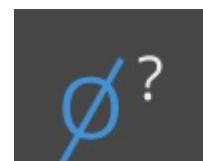
ら数値のリストの範囲を調整します。|

Math.Sin

angle	double
□	

求めます。|

Core.Object



Object.IsNotNull 指定されたオブジェクトが NULL であるかどうかを判断しま

|||||--|--|--|||ACTIONS|||

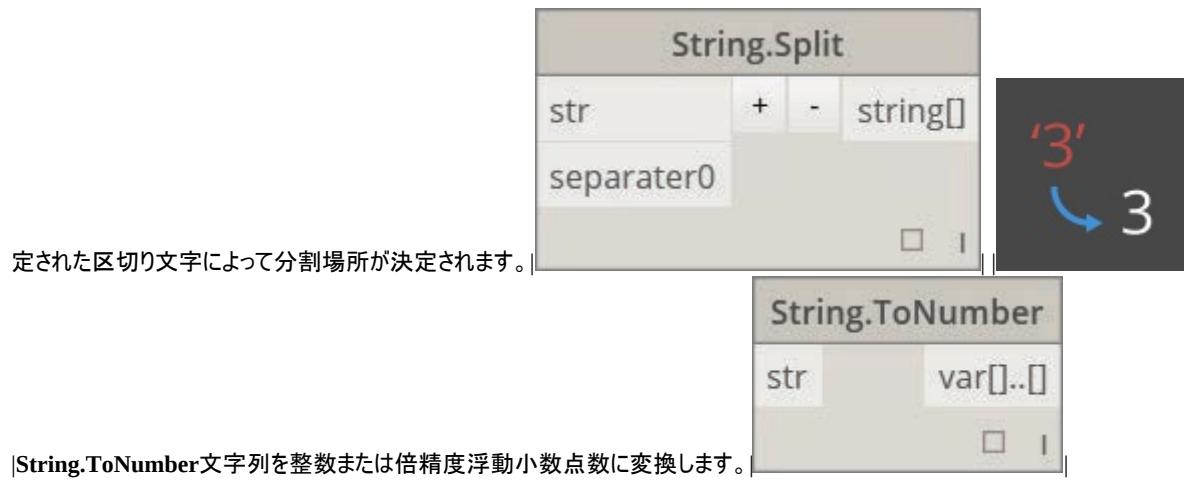


#### Core.Scripting



#### Core.String





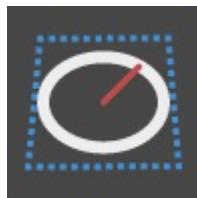
## Core.View



## Geometry

### Geometry.Circle



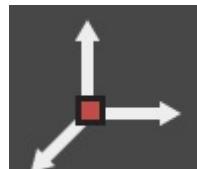


Circle.ByPlaneRadius 入力された平面の基準点(ルート)に中心を持ち、指定された半径を持つ円を平面



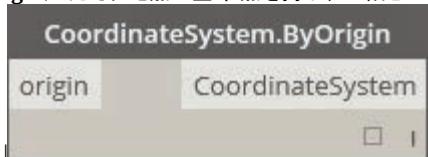
上に作成します。|

#### Geometry.CoordinateSystem

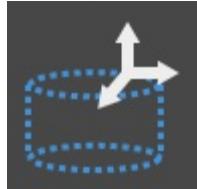


||||| -- | -- | -- || CREATE ||

CoordinateSystem.ByOrigin 入力された点に基準点を持ち、X 軸と Y 軸



を WCS(ワールド座標系)の X 軸および Y 軸に設定した座標系を作成します。|

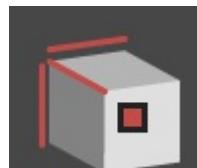


CoordinateSystem.ByCylindricalCoordinates 指定された座標系に対して、指定された円柱座標パラ



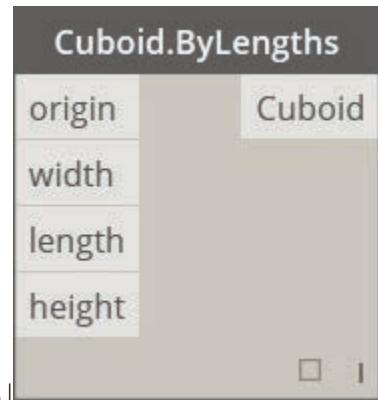
メータに基づいて座標系を作成します。|

#### Geometry.Cuboid

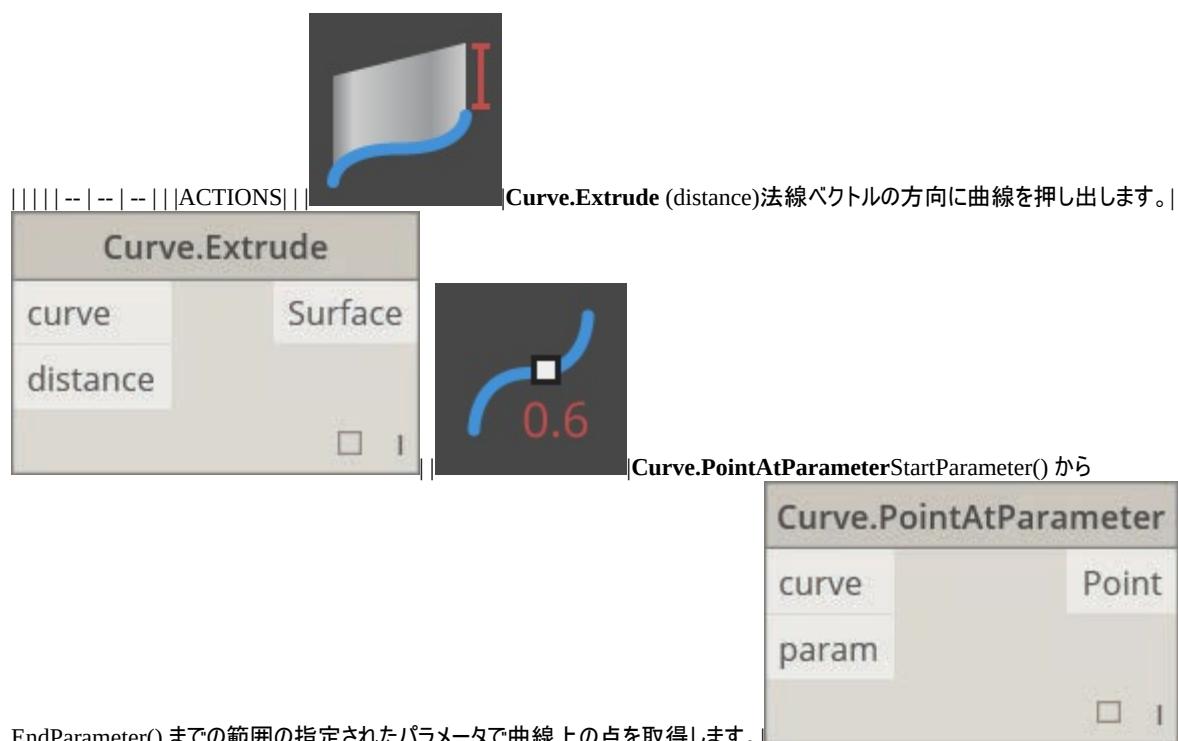


||||| -- | -- | -- || CREATE ||

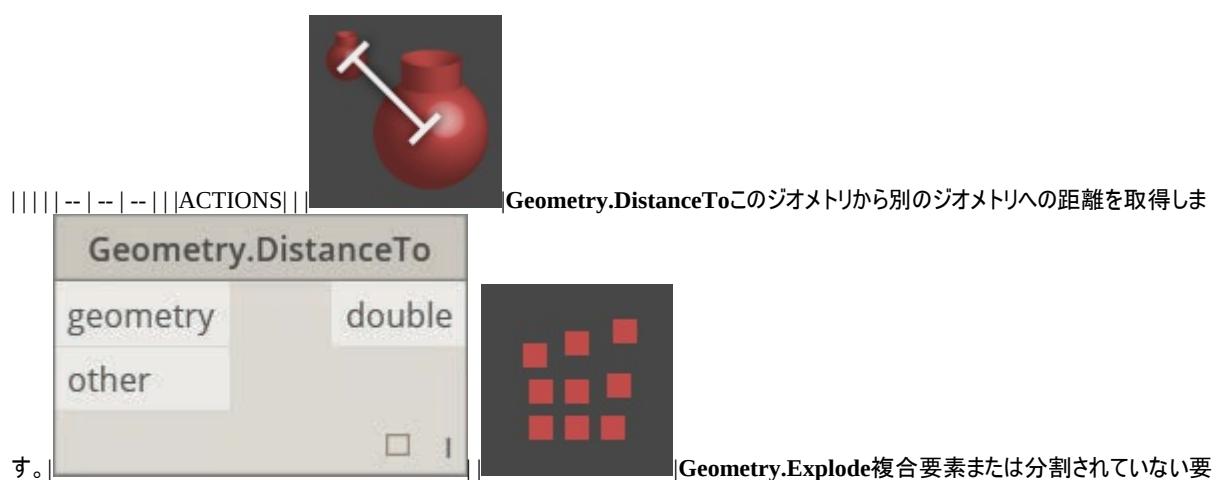
Cuboid.ByLengths (origin) 中心を入力された点に設定し、指定された幅、

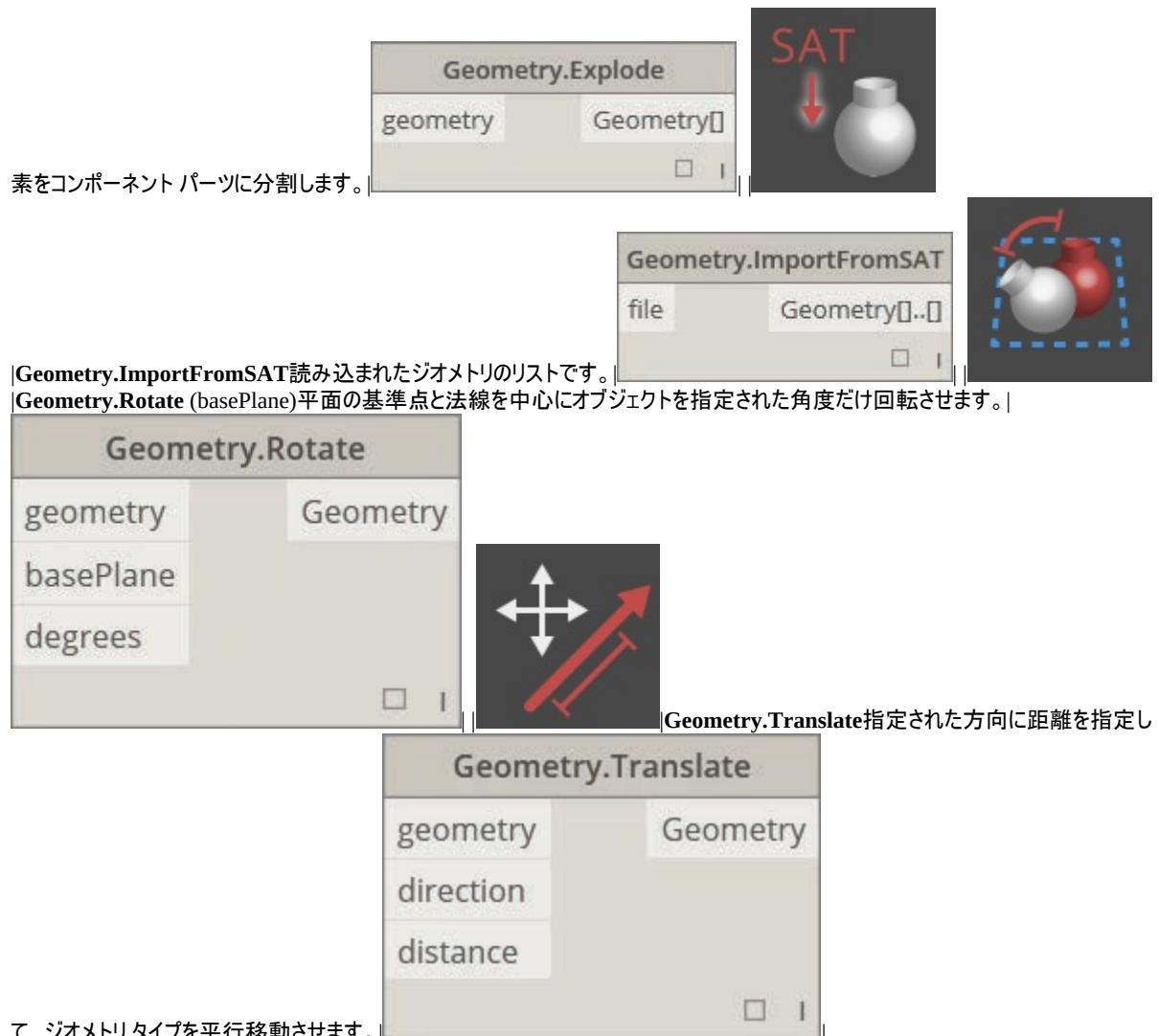


#### Geometry.Curve

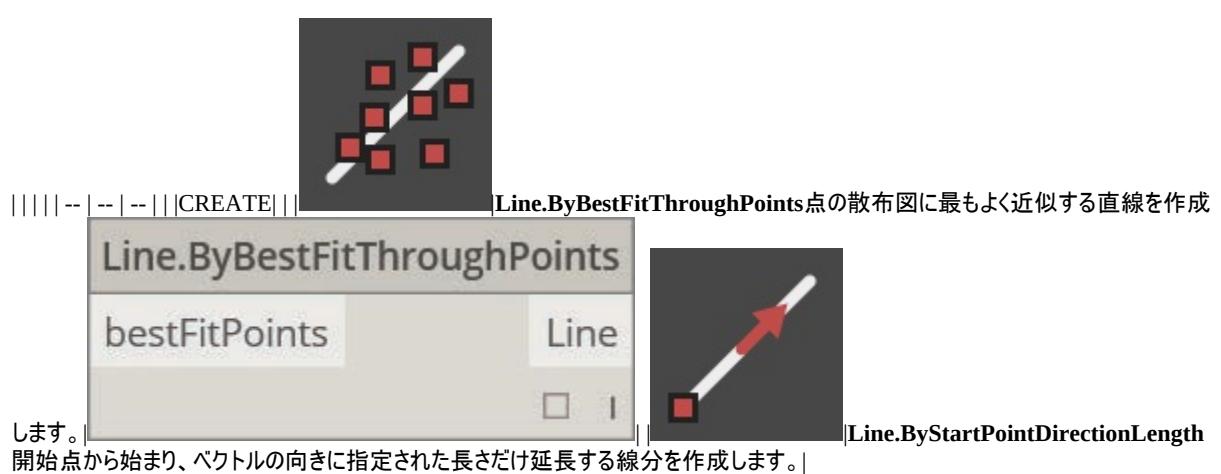


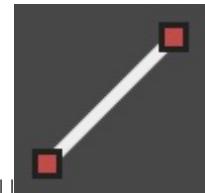
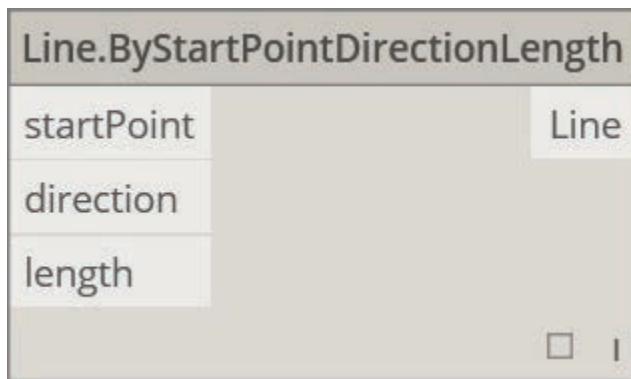
#### Geometry.Geometry



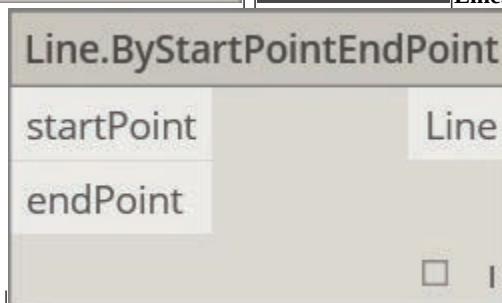


#### Geometry.Line



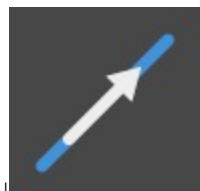
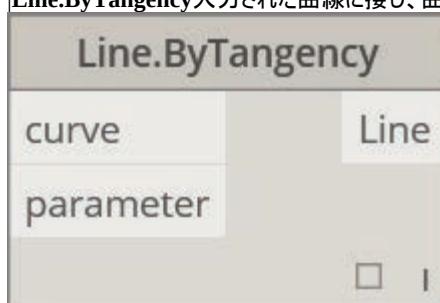


|Line.ByStartPointEndPoint 入力された 2 点を端点とする線分を作成します。|

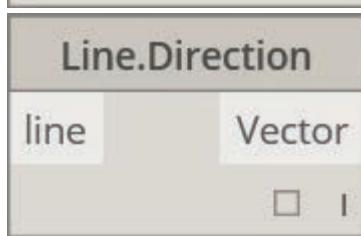


れた 2 点を端点とする線分を作成します。|

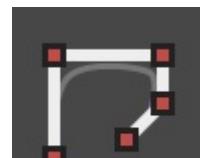
|Line.ByTangency 入力された曲線に接し、曲線のパラメータで指定された点に位置する直線を作成します。|



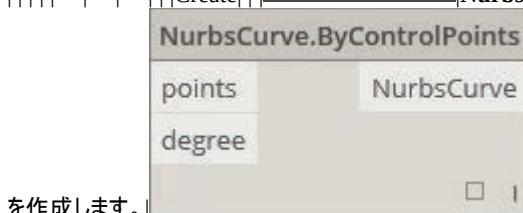
||| QUERY ||| |Line.Direction 曲線の方向を返します。|



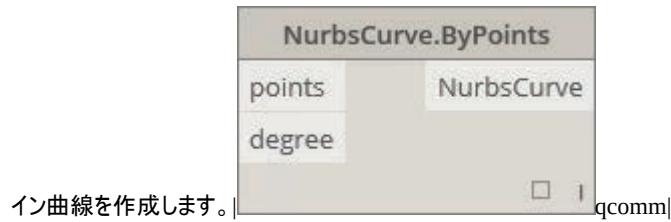
Geometry.NurbsCurve



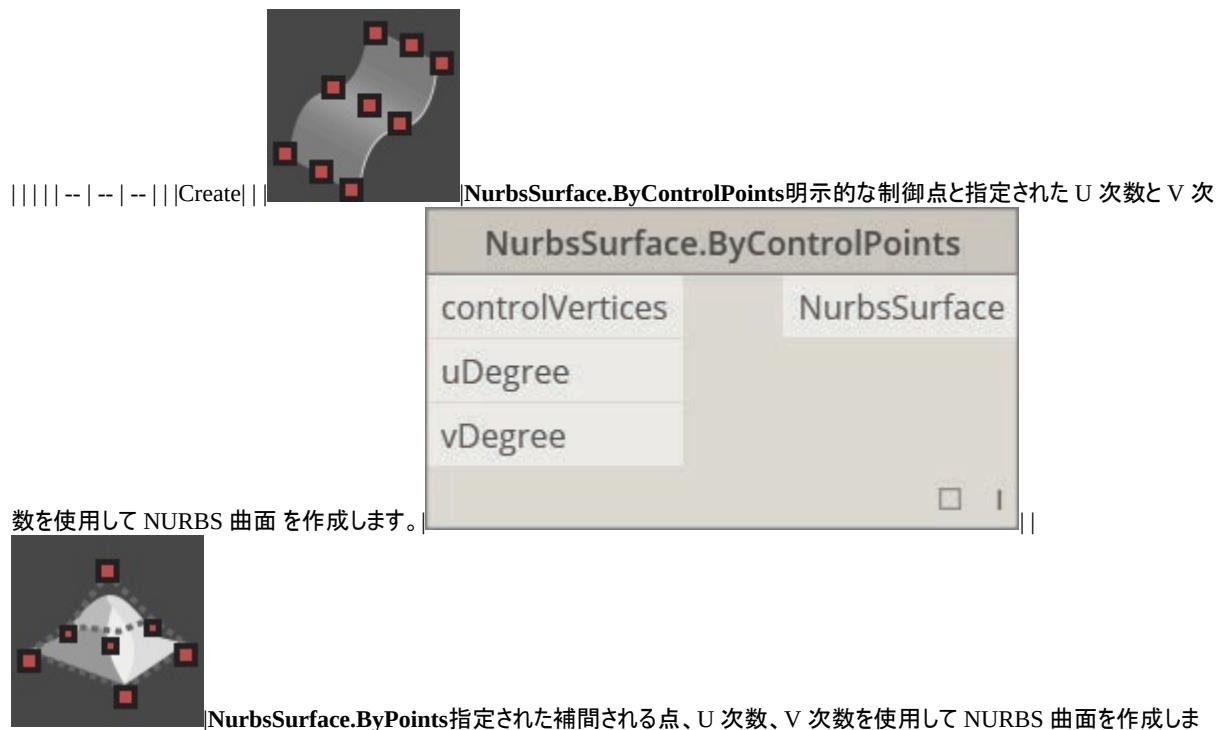
||||--|--|--| |Create| | |NurbsCurve.ByControlPoints 明示的な制御点を使用して B スプライン曲線



を作成します。| |NurbsCurve.ByPoints 点間を補間して B スプラ



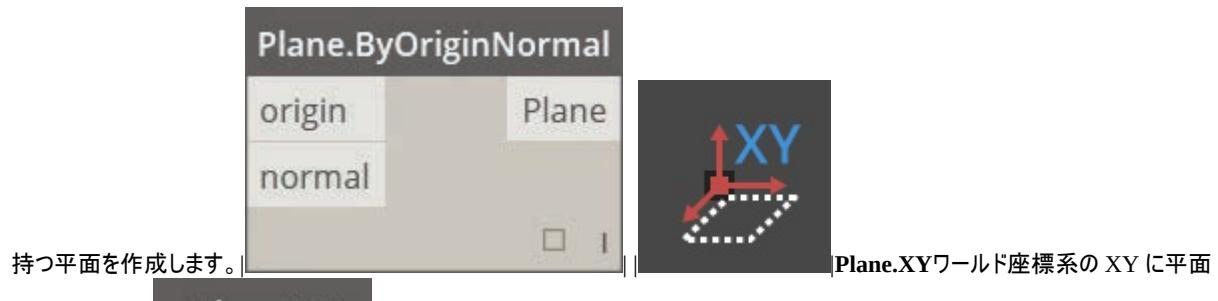
### Geometry.NurbsSurface



す。作成されるサーフェスはすべての指定された点を通過します。

### Geometry.Plane



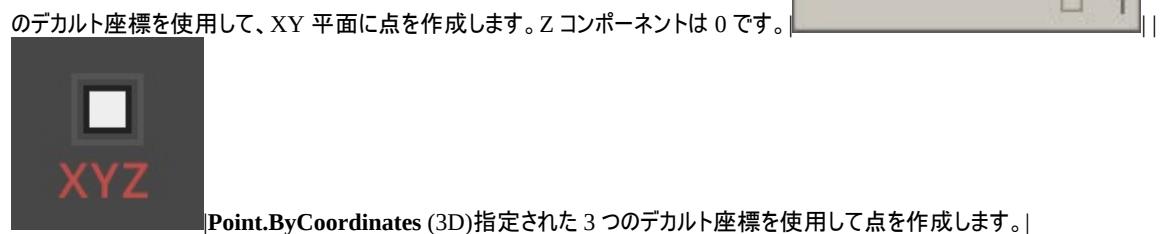


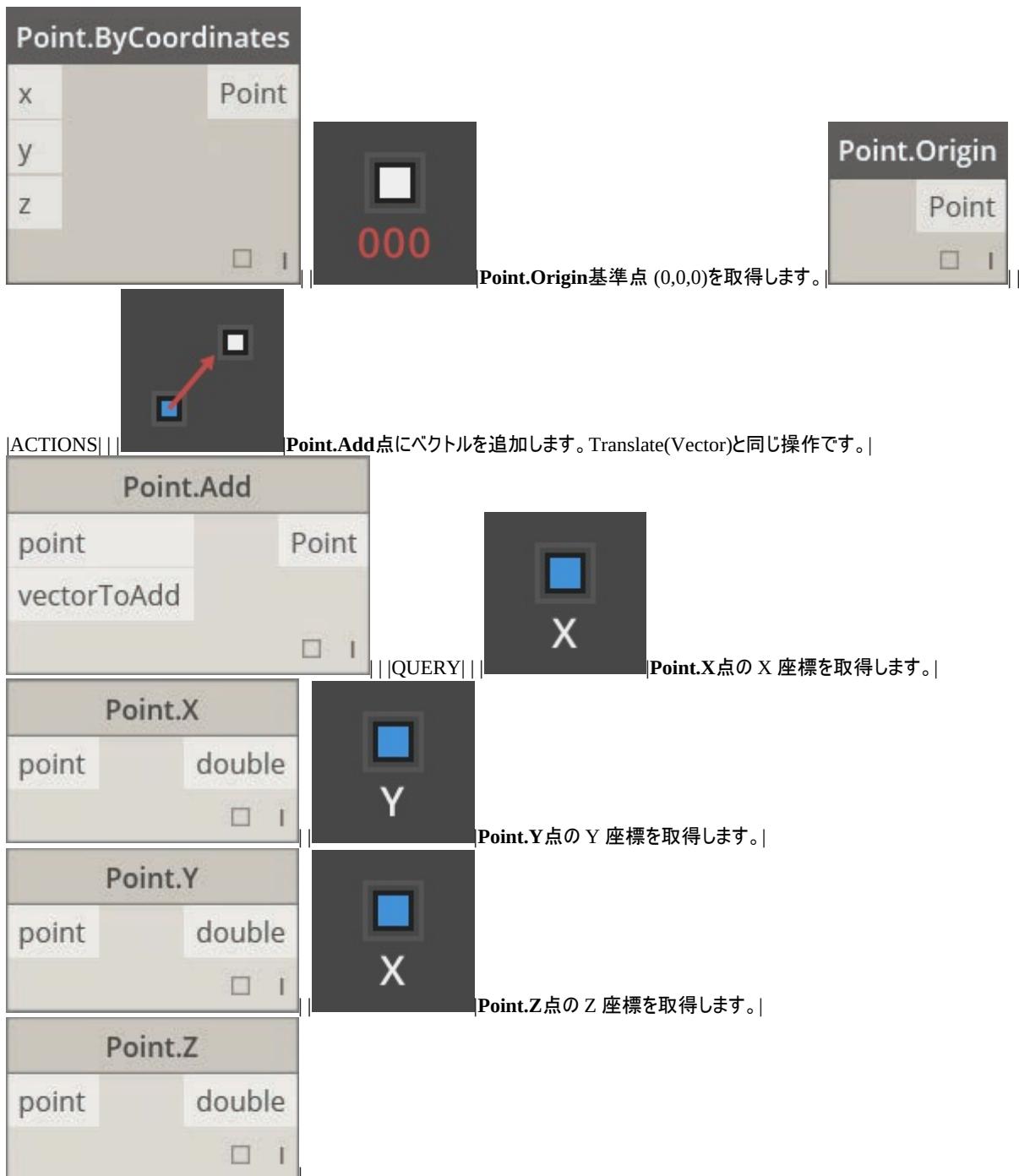
を作成します。|

#### Geometry.Point



点を作成します。| Point.ByCoordinates (2D)



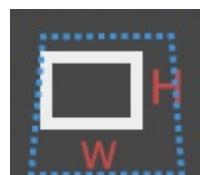


### Geometry.Polycurve





**Geometry.Rectangle**



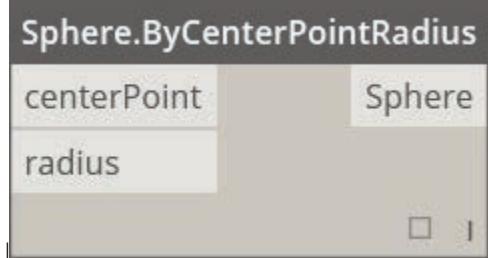
||||| -- | -- | -- ||| CREATE ||| **Rectangle.ByWidthLength** (Plane) 入力された幅(平面の X 軸の長さ)と高さ(平面の Y 軸の長さ)を使用して、平面のルートを中心とする長方形を作成します。|



**Geometry.Sphere**



||||| -- | -- | -- ||| CREATE ||| **Sphere.ByCenterPointRadius** 入力された点を中心とし、指定された半径を

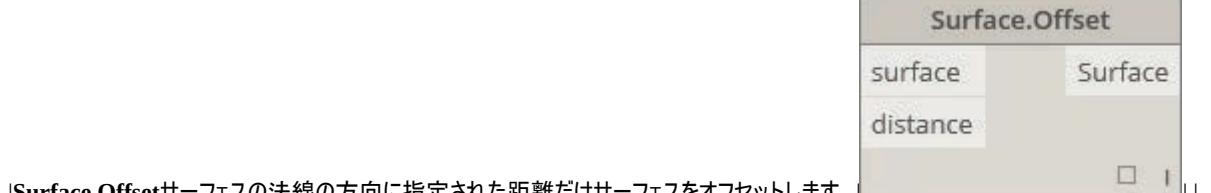
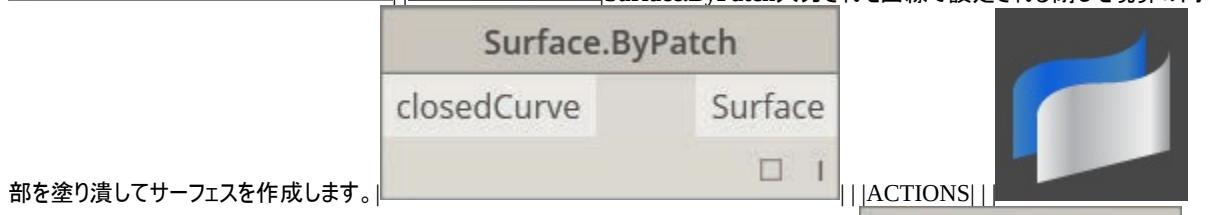


持つソリッド球体を作成します。|

**Geometry.Surface**



||||| -- | -- | -- ||| CREATE ||| **Surface.ByLoft** 入力された断面曲線間をロフトしてサーフェスを作成します。|



#### Geometry.UV





Geometry.Vector

The grid contains the following nodes:

- UV.ByCoordinates**: An input node with inputs **u** and **v**, and output **UV**. It includes a preview showing a red arrow labeled **XYZ**.
- Vector.ByCoordinates**: An input node with inputs **x**, **y**, and **z**, and output **Vector**. It includes a preview showing a blue coordinate system with axes **X**, **Y**, and **Z**.
- Vector.XAxis**: An input node with output **Vector**. It includes a preview showing a blue coordinate system with axes **X**, **Y**, and **Z**.
- Vector.YAxis**: An input node with output **Vector**. It includes a preview showing a blue coordinate system with axes **X**, **Y**, and **Z**.
- Vector.ZAxis**: An input node with output **Vector**. It includes a preview showing a blue coordinate system with axes **X**, **Y**, and **Z**.
- Vector.Normalized**: An input node with input **vector** and output **Vector**. It includes a preview showing a red arrow labeled **1**.

Annotations for the nodes:

- Vector.ByCoordinates**: "Vector.ByCoordinates 3つのヨークリッド座標でベクトルを形成します。| CREATE | | | | | -- | -- | -- | |"
- Vector.XAxis**: "Vector.XAxis 基底 X 軸ベクトル(1,0,0)を取得します。| Vector | | | | | -- | -- | -- | |"
- Vector.YAxis**: "Vector.YAxis 基底 Y 軸ベクトル(0,1,0)を取得します。| Vector | | | | | -- | -- | -- | |"
- Vector.ZAxis**: "Vector.ZAxis 基底 Z 軸ベクトル(0,0,1)を取得します。| Vector | | | | | -- | -- | -- | |"
- Vector.Normalized**: "Vector.Normalized 正規化されたベクトルを取得します。| Vector | | | | | -- | -- | -- | | ACTIONS | |"

## 演算子

The grid contains the following nodes:

- +**: A plus sign operator.
- \***: A multiplication operator.
- : A minus sign operator.
- Vector.Multiplication**: An input node with inputs **x** and **y**, and output **var[]..[]**. It includes a preview showing a blue coordinate system with axes **X**, **Y**, and **Z**.

Annotations for the operators:

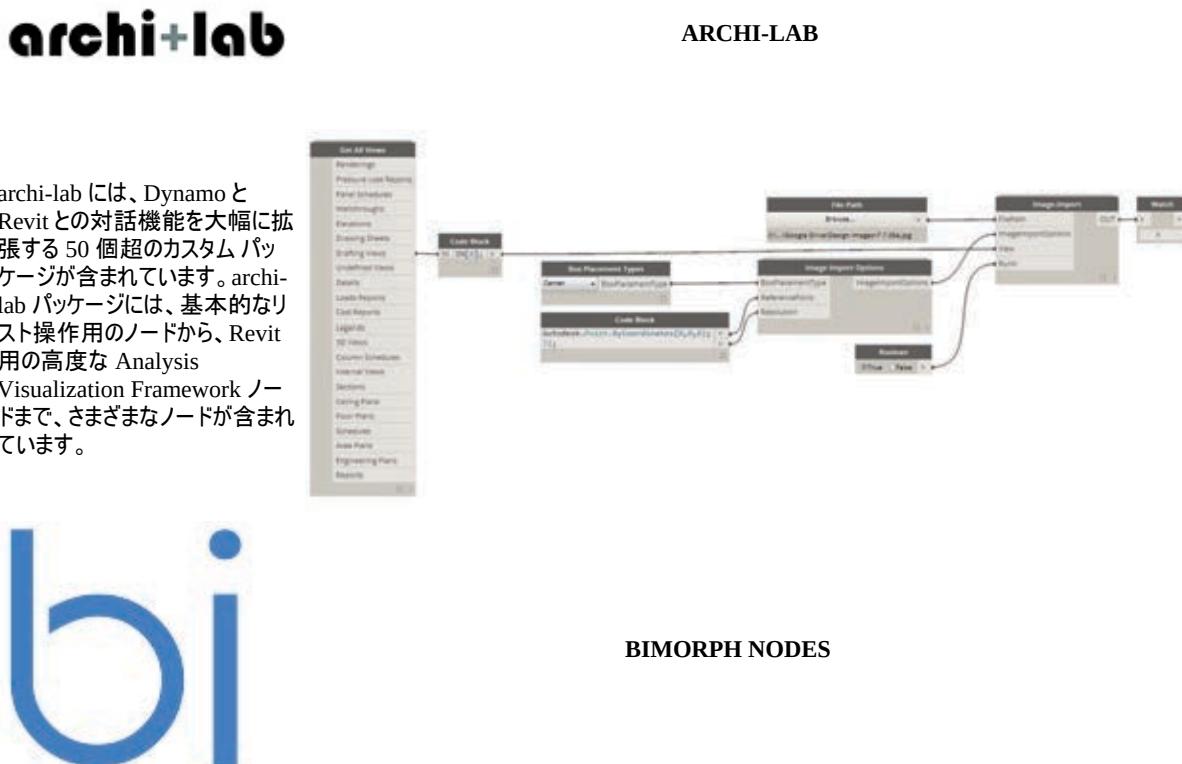
- +**: "+加算 | | | | | -- | -- | -- | |"
- \***: "Vector.Multiplication **x** **y** **var[]..[]** | | | | | -- | -- | -- | |"
- : "-減算 | | | | | -- | -- | -- | |"



## Dynamo パッケージ

## Dynamo パッケージ

このセクションでは、Dynamo コミュニティで人気のあるパッケージをいくつか紹介します。これ以外にも便利なパッケージがあれば、リストに追加してください。[Dynamo Primer](#) は、オープンソース型のドキュメントです。開発者の皆様の積極的な参加をお待ちしています。

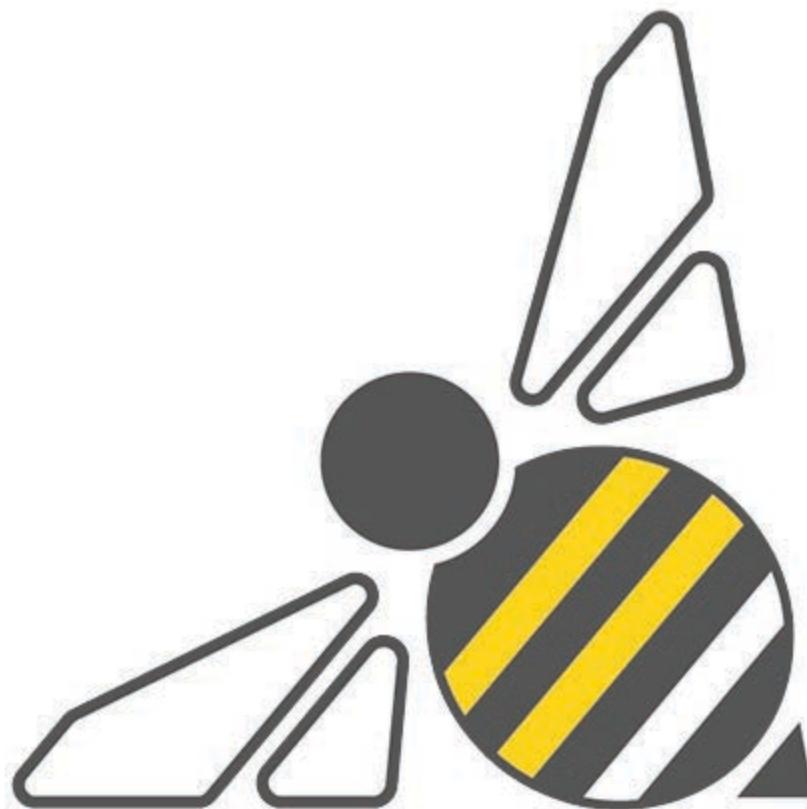
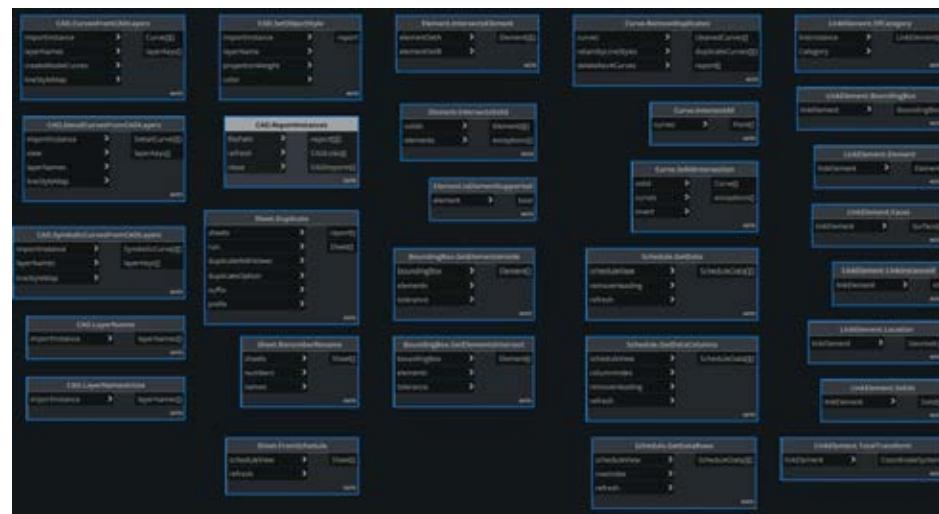


BimorphNodes は強力なユーティリティノードを集めたもので、汎用性があります。パッケージの特徴は、きわめて効率的な干渉検出とジオメトリの交差ノード、ImportInstance (CAD) の

曲線変換ノード、そして、Revit API での制限を解決するリンクされた要素のコレクタにあります。利用可能なすべての範囲のノードについては、BimorphNodes Dictionary を参照してください

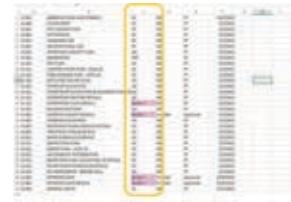


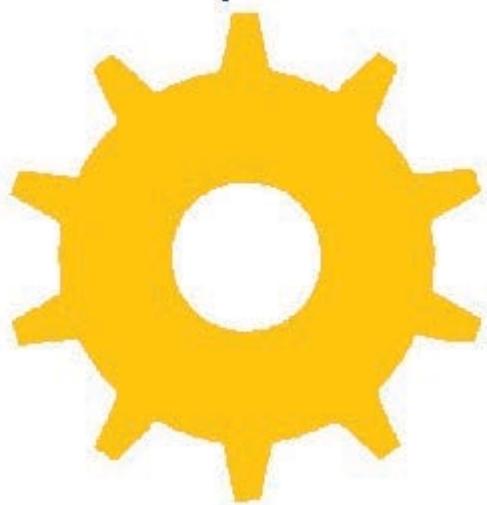
なすべての範囲のノードについては、BimorphNodes Dictionary を参照してください。



**BUMBLEBEE FOR  
DYNAMO**

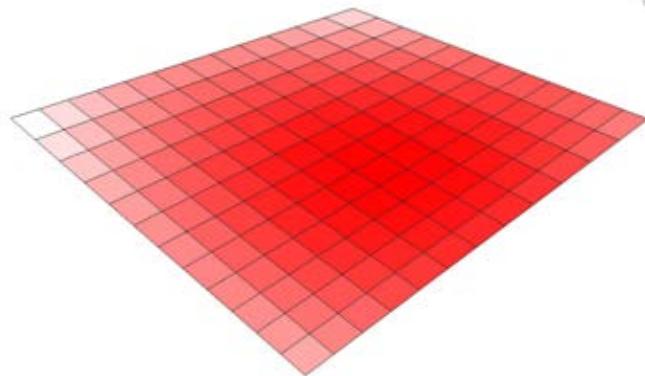
Bumblebee は、Dynamo で Excel ファイルを読み書きする際の機能を大幅に向上させる、Excel と Dynamo の相互運用性プラグインです。





## CLOCKWORK FOR DYNAMO

Clockwork は、Dynamo ビジュアル プログラミング環境用のカスタム ノードが集まつたものです。Clockwork には、Revit に関連する多数のノードに加え、リスト管理、算術演算、文字列処理、単位変換、ジオメトリの操作(主に、境界ボックス、メッシュ、平面、点、サーフェス、UV、ベクトル)、パネル作成など、さまざまな用途のノードが豊富に用意されています。





DataShapes は、Dynamo スクリプトのユーザ機能を拡張するためのパッケージです。このパッケージにより、Dynamo プレーヤーの機能が大幅に拡張されます。詳細については、<https://data-shapes.net/> を参照してください。このパッケージを使用すれば、Dynamo プレーヤーで高度なワークフローを作成することができます。



DYNAM  
SAP



DynamoSAP は、Dynamo 上に構築された、SAP2000 用のパラメトリック インタフェースです。このプロジェクトにより、設計者とエンジニアは Dynamo で SAP モデルを駆動し、SAP 上の構造システムを生成的なアプローチで作成して解析することができます。このプロジェクトでは、付属のサンプル ファイルで説明されているように、一般的なワークフローが事前に用意されています。これらのワークフローにより、SAP における多くの一般的なタスクを自動化することができます。



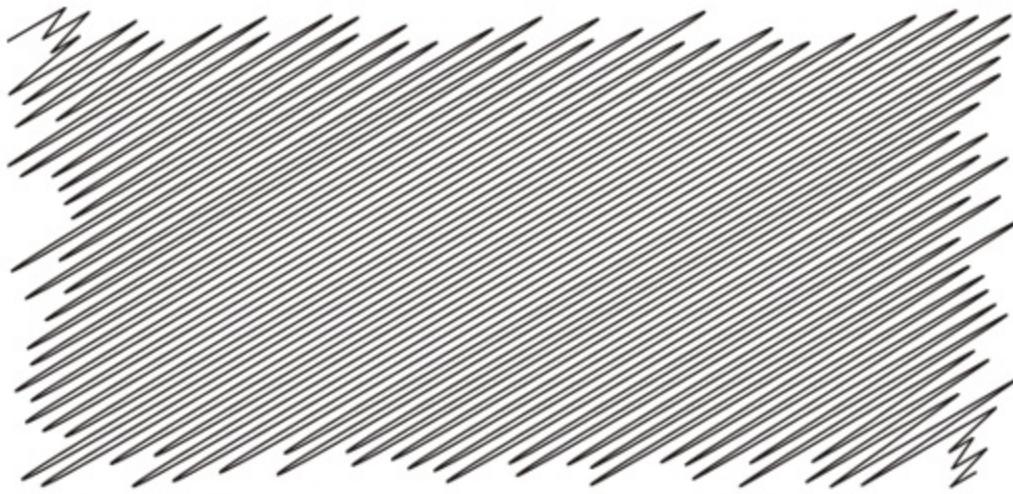
DYNAM  
UNFOL]



このライブラリを使用すると、ユーザ自身がサーフェスとポリサーフェスのジオメトリを展開できるようになるため、Dynamo と Revit の機能が拡張されます。ユーザはこのライブラリを使用して、最初にサーフェスを平らな Tessellate トポロジに変換し、次に Dynamo の ProtoGeometry ツールを使用して Tessellate トポロジを展開することができます。このパッケージには、いくつかのサンプル ノードと、基本的なサンプル ファイルも含まれています。



DYNASTI



**DYNASTRATOR**

Illustrator または Web からベクター アートを .svg 形式で読み込むことにより、手作業で作成したイラストを Dynamo に取り込み、パラメトリック操作を行うことができます。



ENERG  
ANALYS  
FOR  
DYNAM

Energy Analysis for Dynamo を使用すると、Dynamo 0.8 でパラメトリック エネルギー モデリングと建物全体のエネルギー解析ワークフローを実行することができます。また、Autodesk Revit でエネルギー モデルを設定し、そのモデルを Green Building Studio に送信して DOE2 エネルギー解析を行ったり、解析結果を詳しく分析することもできます。このパッケージは、Thornton Tomasetti の CORE studio で開発されています。





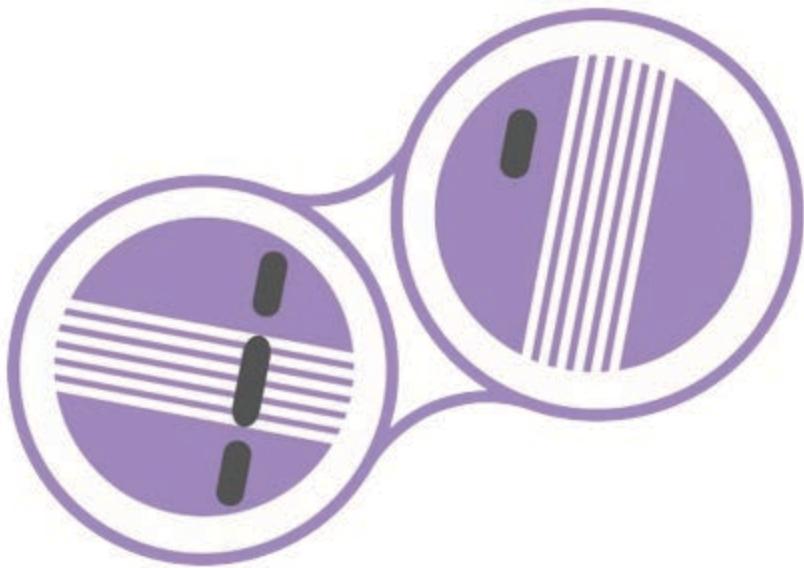
ノードの集まりである Firefly を使用すると、Dynamo で Arduino マイコンなどの入出力デバイスと通信を行うことができます。データがリアルタイムに処理されるため、Web カメラ、携帯電話、ゲーム コントローラ、センサなどを使用して、デジタルの世界と現実の世界を連携させるためのさまざまなプロトタイプを作成することができます。



LunchBox は、再利用可能なジオメトリノードとデータ管理ノードの集合です。このツールは、Dynamo 0.8.1 と Revit 2016 で動作することがテストによって確認されています。このツールには、サーフェスのパネル化、ジオメトリ、Revit データ集合などで使用できるノードが含まれています。



## MANTIS SHRIMP



Mantis Shrimp は、Grasshopper ジオメトリや Rhino ジオメトリを Dynamo に簡単に読み込むことができる、相互運用性プロジェクトです。





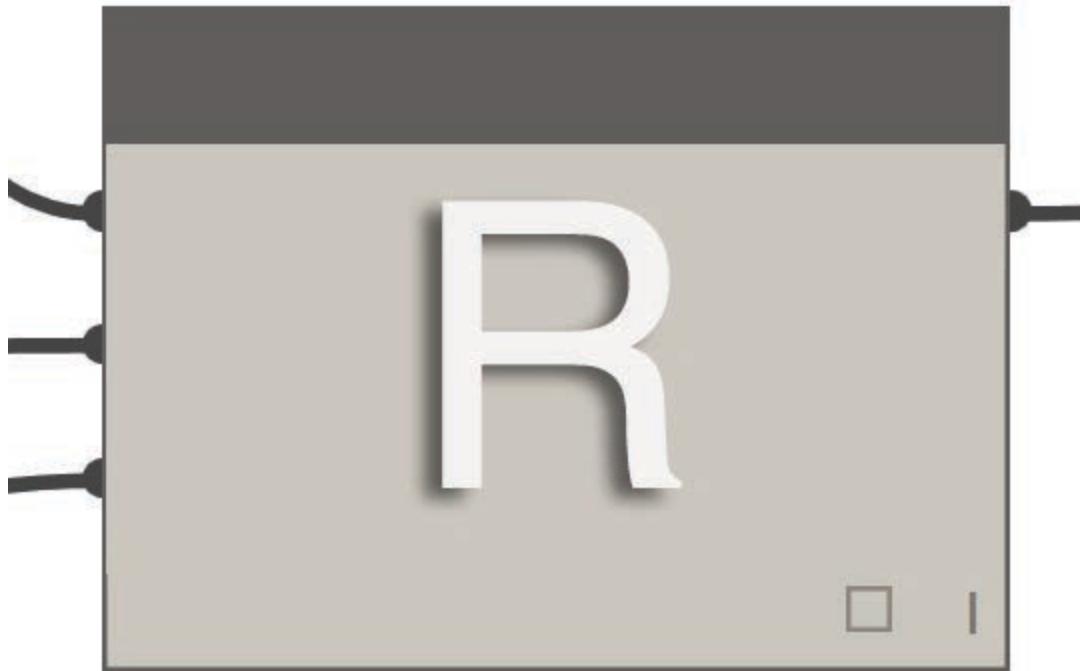
Dynamo Mesh Toolkitには、メッシュ ジオメトリを操作するための便利な各種ツールが用意されています。このパッケージには、外部のファイル形式からメッシュを読み込む機能、既存の Dynamo ジオメトリオブジェクトからメッシュを生成する機能、頂点と接続に関する情報からメッシュを手動で作成する機能が含まれています。また、このツールキットには、メッシュ ジオメトリの修正や修復を行うためのツールが含まれています。



OPTIMC

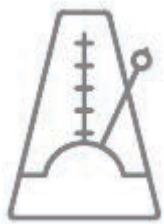


Dynamo ユーザは、Optimo で各種の高度なアルゴリズムを使用して、自分で定義した設計上の問題を最適化することができます。また、問題の目的と適応度関数を定義することができます。



RHYNAL

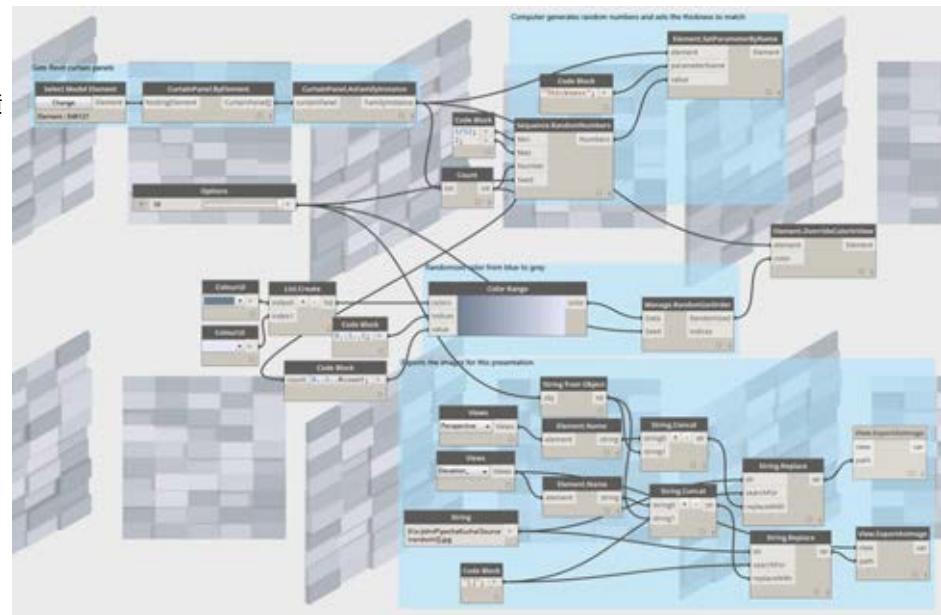
Rhynamo ノードライブラリを使用すると、Rhino 3DM ファイルを Dynamo 内で読み書きすることができます。Rhynamo は、McNeel の OpenNURBS ライブラリを使用して、Rhino のジオメトリを Dynamo で使用できるジオメトリに変換します。これにより、Rhino と Revit 間でジオメトリやデータを柔軟に交換するための新しいワークフローが作成されます。このパッケージには、Rhino コマンドラインへの「ライブ」アクセスが可能なサンプルノードも含まれています。



RHYTHM

一見すると、Rhythm に

は、高度なコードなどの特別な機能は見当たりません。しかし、実践的な思考や絶え間ない取り組みの積み重ねで開発されたのが Rhythm です。このパッケージは、ユーザが Revit と Dynamo でスムーズに作業できるように作成されています。Rhythm を構成する主要なノードは、Revit 環境ですぐに使用できる Dynamo ノードです。



# Dynamo のサンプル ファイル

## Dynamo のサンプル ファイル

ここでは、Dynamo Primer に付属しているサンプル ファイルを、章とセクションで分類して記載します。

サンプル ファイルをダウンロードするには、対象のファイルを右クリックして[名前を付けてリンク先を保存]を選択してください。

はじめに

セクション ダウンロード ファイル

ビジュアル プログラミングの概要 [Visual Programming - Circle Through Point.dyn](#)

ビジュアル プログラムの構造

セクション ダウンロード ファイル

プリセット [Presets.dyn](#)

プログラムの構成要素

セクション ダウンロード ファイル

データ [Building Blocks of Programs - Data.dyn](#)

数学的方法 [Building Blocks of Programs - Math.dyn](#)

ロジック [Building Blocks of Programs - Logic.dyn](#)

文字列 [Building Blocks of Programs - Strings.dyn](#)

色 [Building Blocks of Programs - Color.dyn](#)

計算設計用のジオメトリ

セクション

ダウンロード ファイル

ジオメトリの概要

[Geometry for Computational Design - Geometry Overview.dyn](#)

ベクトル

[Geometry for Computational Design - Vectors.dyn](#)

[Geometry for Computational Design - Plane.dyn](#)

[Geometry for Computational Design - Coordinate System.dyn](#)

点

[Geometry for Computational Design - Points.dyn](#)

曲線

[Geometry for Computational Design - Curves.dyn](#)

サーフェス

[Geometry for Computational Design - Surfaces.dyn](#)

[Surface.sat](#)

リストを使用した設計

セクション ダウンロード ファイル

リストの概要 [Lacing.dyn](#)

リストの操作 [List-Count.dyn](#)

[List-FilterByBooleanMask.dyn](#)

[List-GetItemAtIndex.dyn](#)

[List-Operations.dyn](#)

[List-Reverse.dyn](#)

[List-ShiftIndices.dyn](#)

リストのリスト	<a href="#">Chop.dyn</a>
<a href="#">Combine.dyn</a>	
<a href="#">Flatten.dyn</a>	
<a href="#">Map.dyn</a>	
<a href="#">ReplaceItems.dyn</a>	
<a href="#">Top-Down-Hierarchy.dyn</a>	
<a href="#">Transpose.dyn</a>	
N 次元のリスト	<a href="#">n-Dimensional-Lists.dyn</a>
<a href="#">n-Dimensional-Lists.sat</a>	

#### コード ブロックと DesignScript

セクション	ダウンロード ファイル
DesignScript 構文	<a href="#">Dynamo-Syntax_Attractor-Surface.dyn</a>
省略表記	<a href="#">Obsolete-Nodes_Sine-Surface.dyn</a>
関数	<a href="#">Functions_SphereByZ.dyn</a>

#### Revit で Dynamo を使用する

セクション	ダウンロード ファイル
選択	<a href="#">Selecting.dyn</a>
<a href="#">ARCH-Selecting-BaseFile.rvt</a>	
編集	<a href="#">Editing.dyn</a>
<a href="#">ARCH-Editing-BaseFile.rvt</a>	
作成	<a href="#">Creating.dyn</a>
<a href="#">ARCH-Creating-BaseFile.rvt</a>	
カスタマイズ	<a href="#">Customizing.dyn</a>
<a href="#">ARCH-Customizing-BaseFile.rvt</a>	
設計図書の作成	<a href="#">Documenting.dyn</a>
<a href="#">ARCH-Documenting-BaseFile.rvt</a>	

#### Dynamo のディクショナリ

セクション	ダウンロード ファイル
部屋のディクショナリ	<a href="#">RoomDictionary.dyn</a>

#### カスタム ノード

セクション	ダウンロード ファイル
カスタム ノードの作成	<a href="#">UV-CustomNode.zip</a>
ライブラリへのパブリッシュ	<a href="#">PointsToSurface.dynf</a>
Python Script ノード	<a href="#">Python-CustomNode.dyn</a>
Python と Revit	<a href="#">Revit-Doc.dyn</a>
Python と Revit	<a href="#">Revit-ReferenceCurve.dyn</a>
Python と Revit	<a href="#">Revit-StructuralFraming.zip</a>

#### パッケージ

セクション	ダウンロード ファイル
パッケージのケース スタディ - Mesh Toolkit	<a href="#">MeshToolkit.zip</a>

パッケージのパブリッシュ  
Zero-Touch Importing

[MapToSurface.zip](#)  
[ZeroTouchImages.zip](#)