

Table of Contents

1. [Acerca de](#) 0
2. [Introducción](#) 1
 1. [¿Qué es la programación visual?](#) 1.1
 2. [¿Qué es Dynamo?](#) 1.2
 3. [Dynamo en acción](#) 1.3
3. [¡Hola, Dynamo!](#) 2
 1. [Instalación e inicio de Dynamo](#) 2.1
 2. [Interfaz del usuario](#) 2.2
 3. [El espacio de trabajo](#) 2.3
 4. [Procedimientos iniciales](#) 2.4
4. [Anatomía de un programa visual](#) 3
 1. [Nodos](#) 3.1
 2. [Cables](#) 3.2
 3. [Biblioteca](#) 3.3
 4. [Administración del programa](#) 3.4
5. [Los bloques de construcción de los programas](#) 4
 1. [Datos](#) 4.1
 2. [Matemáticas](#) 4.2
 3. [Lógica](#) 4.3
 4. [Cadenas](#) 4.4
 5. [Color](#) 4.5
6. [Geometría para el diseño computacional](#) 5
 1. [Descripción general de la geometría](#) 5.1
 2. [Vectores](#) 5.2
 3. [Puntos](#) 5.3
 4. [Curvas](#) 5.4
 5. [Superficies](#) 5.5
 6. [Sólidos](#) 5.6
 7. [Mallas](#) 5.7
 8. [Importación de geometría](#) 5.8
7. [Diseño con listas](#) 6
 1. [¿Qué es una lista?](#) 6.1
 2. [Trabajo con listas](#) 6.2
 3. [Listas de listas](#) 6.3
 4. [Listas de n dimensiones](#) 6.4
8. [Bloques de código y DesignScript](#) 7
 1. [¿Qué es un bloque de código?](#) 7.1
 2. [Sintaxis de DesignScript](#) 7.2
 3. [Abreviatura](#) 7.3
 4. [Funciones](#) 7.4
9. [Dynamo para Revit](#) 8
 1. [La conexión de Revit](#) 8.1
 2. [Selección](#) 8.2
 3. [Edición](#) 8.3
 4. [Creación](#) 8.4
 5. [Personalización](#) 8.5
 6. [Documentación](#) 8.6
10. [Diccionarios en Dynamo](#) 9
 1. [¿Qué es un diccionario?](#) 9.1
 2. [Usos de los nodos](#) 9.2
 3. [Usos de los bloques de código](#) 9.3
 4. [Casos de uso](#) 9.4
11. [Nodos personalizados](#) 10
 1. [Introducción a los nodos personalizados](#) 10.1
 2. [Creación de un nodo personalizado](#) 10.2
 3. [Publicación en la biblioteca](#) 10.3
 4. [Nodos de Python](#) 10.4
 5. [Python y Revit](#) 10.5
 6. [Plantillas de Python en Dynamo 2.0](#) 10.6
12. [Paquetes](#) 11
 1. [Introducción a los paquetes](#) 11.1
 2. [Caso real de paquete: Kit de herramientas de malla](#) 11.2
 3. [Desarrollo de un paquete](#) 11.3
 4. [Publicación de un paquete](#) 11.4
 5. [Importación de Zero-Touch](#) 11.5
13. [Geometría con DesignScript](#) 12
 1. [Conceptos básicos de geometría con DesignScript](#) 12.1
 2. [Primitivas geométricas](#) 12.2
 3. [Matemáticas vectoriales](#) 12.3
 4. [Curvas: puntos interpolados y de control](#) 12.4
 5. [Traslación, rotación y otras transformaciones](#) 12.5
 6. [Superficies: interpoladas, puntos de control, solevación y revolución](#) 12.6
 7. [Parametrización geométrica](#) 12.7
 8. [Intersección y recorte](#) 12.8
 9. [Operaciones booleanas geométricas](#) 12.9
 10. [Generadores de puntos de Python](#) 12.10
14. [Prácticas recomendadas](#) 13
 1. [Estrategias gráficas](#) 13.1

- 2. [Estrategias de creación de secuencias de comandos](#) 13.2
 - 3. [Referencia de secuencias de comandos](#) 13.3
- 15. [Apéndice](#) 14
 - 1. [Recursos](#) 14.1
 - 2. [Índice de nodos](#) 14.2
 - 3. [Paquetes útiles](#) 14.3
 - 4. [Archivos de ejemplo](#) 14.4

Acerca de Dynamo Primer

Para Dynamo v2.0

Descargue [Dynamo v1.3 Primer aquí](#).



Dynamo

Dynamo es una plataforma de programación visual de código abierto para diseñadores.

Bienvenido

Acaba de abrir Dynamo Primer, una guía completa de programación visual en Autodesk Dynamo. Este manual de introducción es un proyecto continuo para compartir los conceptos básicos de programación. Entre los temas, se incluyen el trabajo con geometría computacional, los procedimientos recomendados para el diseño basado en reglas, las aplicaciones de programación interdisciplinarias y muchos más aspectos relacionados con la plataforma Dynamo.

La potencia de Dynamo se puede encontrar en una amplia variedad de actividades relacionadas con el diseño. Dynamo ofrece una lista ampliable de métodos accesibles para que dé sus primeros pasos:

- **Explore** la programación visual por primera vez.
- **Conecte** flujos de trabajo en varios programas.
- **Participe** en una comunidad activa de usuarios, colaboradores y desarrolladores.
- **Desarrolle** una plataforma de código abierto para seguir mejorando.

En medio de esta actividad y magnífica oportunidad para trabajar con Dynamo, necesitamos un documento del mismo calibre, Dynamo Primer.

Este manual de introducción incluye capítulos desarrollados con Mode Lab. Estos capítulos se centran en los conceptos básicos que necesita para empezar a trabajar y desarrollar sus propios programas visuales con Dynamo e influyen decisivamente en cómo llevar Dynamo a un nivel superior. Esto es lo que puede aprender con el manual de introducción:

- **Contexto:** ¿qué es exactamente la "programación visual" y cuáles son los conceptos necesarios para profundizar en Dynamo?
- **Introducción:** ¿cómo puedo obtener Dynamo y crear mi primer programa?
- **Contenido de un programa:** ¿cuáles son las partes funcionales de Dynamo y cómo se utilizan?
- **Bloques de construcción:** ¿qué son los "datos" y qué tipos fundamentales puedo utilizar en mis programas?
- **Geometría para el diseño:** ¿cómo puedo trabajar con elementos geométricos en Dynamo?
- **Listas, listas, listas:** ¿cómo administro y coordino las estructuras de datos?
- **Código en nodos:** ¿cómo puedo empezar a ampliar Dynamo con mi propio código?
- **BIM computacional:** ¿cómo puedo utilizar Dynamo con un modelo de Revit?
- **Nodos personalizados:** ¿cómo puedo crear mis propios nodos?
- **Paquetes:** ¿cómo puedo compartir mis herramientas con la comunidad?

Este es un momento emocionante para aprender a usar Dynamo, trabajar con él y desarrollar para esta solución. Comencemos.

Código abierto

El proyecto Dynamo Primer es de código abierto. Nos hemos dedicado a proporcionar contenido de calidad y agradeceremos cualquier comentario que pueda tener. Si desea informar de un problema sobre cualquier aspecto, publíquelo en la página de problemas de GitHub: <https://github.com/DynamoDS/DynamoPrimer/issues>.

Si desea contribuir con una nueva sección, modificaciones o cualquier otra aportación a este proyecto, consulte el repositorio de GitHub para empezar: <https://github.com/DynamoDS/DynamoPrimer>.

El proyecto Dynamo Primer

Dynamo Primer es un proyecto de código abierto iniciado por Matt Jczyk y el equipo de desarrollo de Dynamo de Autodesk.

Mode Lab recibió el encargo de escribir la primera edición del manual de introducción. Les agradecemos todos sus esfuerzos para establecer este valioso recurso.



A **John Pierson de Parallax Team** se le encargó actualizar el manual de introducción para incluir las revisiones de Dynamo 2.0.



Agradecimientos

Un agradecimiento especial a Ian Keough por iniciar y guiar el proyecto de Dynamo.

Deseamos agradecer a Matt Jezyk, Ian Keough, Zach Kron, Racel Williams y Colin McCrone su colaboración entusiasta y la oportunidad de participar en una amplia gama de proyectos de Dynamo.

Software y recursos

Dynamo La versión estable* actual de Dynamo es la 2.1.0.

<http://dynamobim.com/download/> o <http://dynamobuilds.com>

- Nota: a partir de Revit 2020, Dynamo se incluye en las versiones de Revit, por lo que no es necesaria la instalación manual. Hay más información disponible en esta [entrada del blog](#).

DynamoBIM La mejor fuente de información adicional, contenido de aprendizaje y foros es el sitio web de DynamoBIM.

<http://dynamobim.org>

Dynamo en GitHub Dynamo es un proyecto de desarrollo de código abierto en GitHub. Para colaborar, consulte DynamoDS.

<https://github.com/DynamoDS/Dynamo>

Contacto Infórmenos de cualquier problema relacionado con este documento.

Dynamo@autodesk.com

Licencia

Copyright 2019 Autodesk

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. Puede obtener una copia de la licencia en

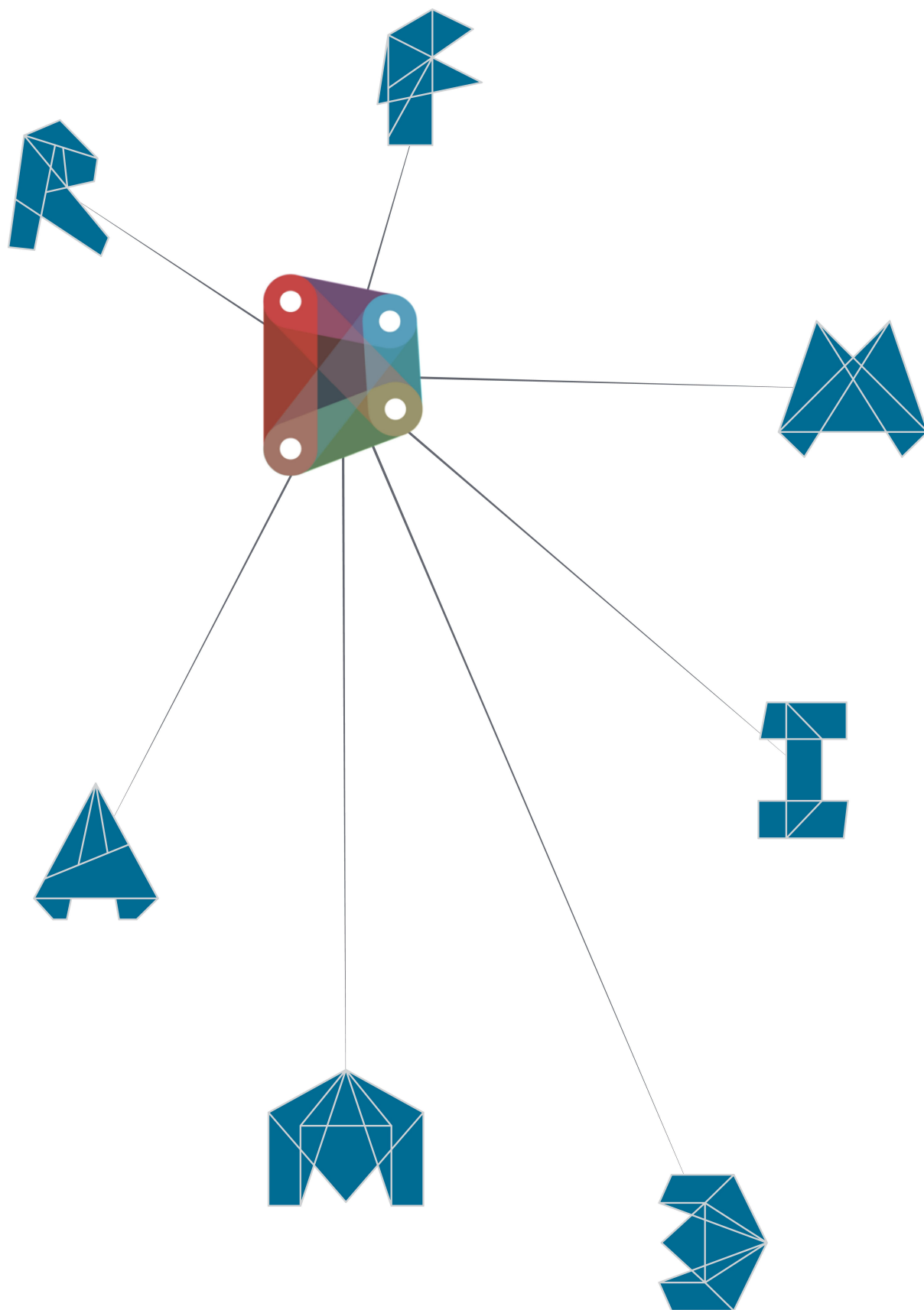
<http://www.apache.org/licenses/LICENSE-2.0>

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.

Introducción

INTRODUCCIÓN

Dynamo ha evolucionado desde sus orígenes como un complemento para BIM (Building Information Modeling) en Revit para convertirse en algo mucho más diverso. Ante todo, se trata de una plataforma que permite a los diseñadores explorar la programación visual, resolver problemas y crear sus propias herramientas. Comencemos nuestro viaje con Dynamo proporcionando algo de contexto: ¿qué es y cómo puedo utilizarlo?



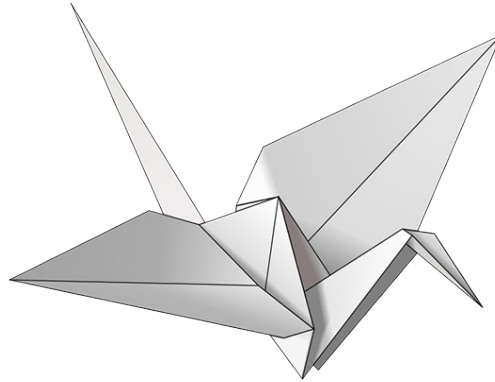
¿Qué es la programación visual?

¿Qué es la programación visual?

El diseño suele implicar establecer relaciones visuales, sistémicas o geométricas entre las partes de un diseño. Por lo general, estas relaciones se desarrollan mediante flujos de trabajo que nos llevan del concepto al resultado mediante el uso de reglas. Quizás sin saberlo, estamos trabajando de forma algorítmica, definiendo un conjunto de acciones paso a paso que siga una lógica básica de entrada, procesamiento y salida. La programación nos permite seguir trabajando de esta forma, pero formalizando nuestros algoritmos.

Algoritmos disponibles

Aunque ofrece algunas oportunidades interesantes, el término **algoritmo** puede dar lugar a algunas ideas erróneas. Los algoritmos pueden generar elementos inesperados, salvajes o increíbles, pero no son mágicos. De hecho, en realidad, son componentes bastante básicos. Usemos un ejemplo tangible como una grulla de papiroflexia. Comenzaremos con un trozo de papel cuadrado (entrada), seguiremos con una serie de pasos de plegado (acciones de procesamiento) y obtendremos una grulla (salida).

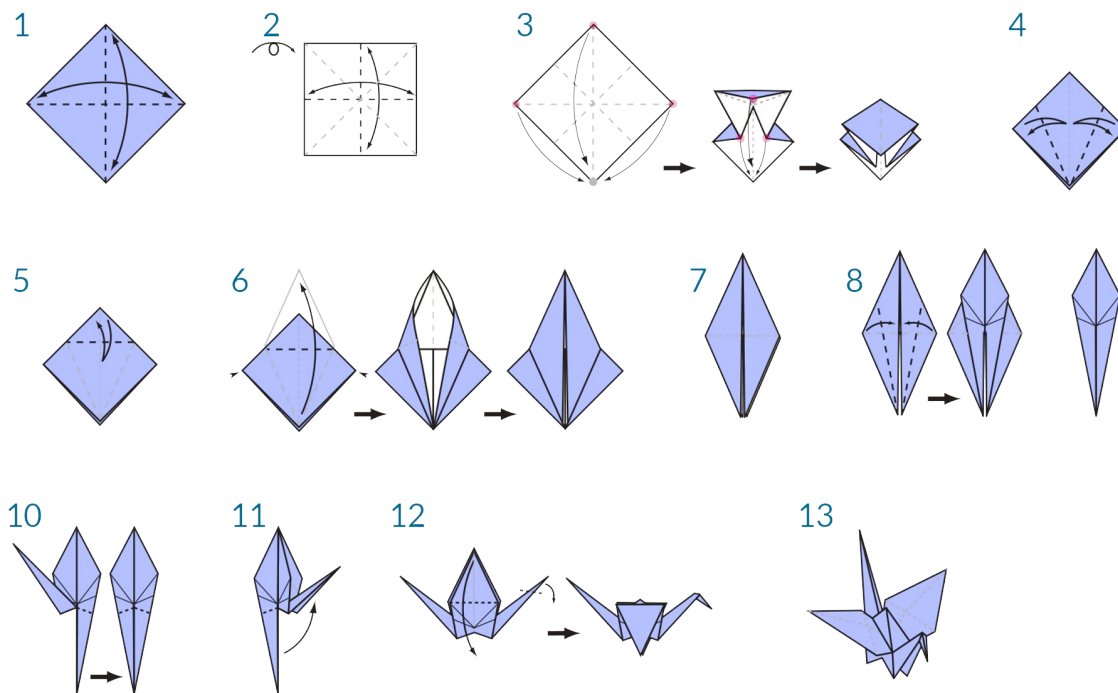


Por lo tanto, ¿dónde está el algoritmo? Es el conjunto abstracto de pasos, que podemos representar de dos maneras: textualmente o gráficamente.

Instrucciones textuales:

1. Comience con un trozo de papel cuadrado, con la parte de color hacia arriba. Dóblelo a la mitad y ábralo. Dóblelo a la mitad por el otro lado.
2. Gire el papel por el lado blanco. Doble el papel por la mitad, pliéguelo bien y ábralo; a continuación, vuelva a doblarlo en la otra dirección.
3. Con los pliegues creados, lleve las tres esquinas superiores del modelo a la esquina inferior. Aplane el modelo.
4. Doble las pestañas triangulares de la parte superior en el centro y despliéguelo.
5. Doble la parte superior del modelo hacia abajo, pliéguelo bien y despliéguelo.
6. Abra la pestaña superior del modelo, trayéndola hacia arriba y presionando los lados del modelo hacia dentro al mismo tiempo. Aplánelo y pliéguelo bien.
7. Gire el modelo y repita los pasos 4 a 6 en el otro lado.
8. Doble las pestañas superiores en el centro.
9. Repita esta operación en el otro lado.
10. Doble ambas "patas" del modelo hacia arriba, pliegue el modelo muy bien y, a continuación, despliéguelo.
11. Doble hacia dentro las "patas" a lo largo de los pliegues que acaba de crear.
12. Doble hacia dentro un lado para crear una cabeza y, a continuación, doble hacia abajo las alas.
13. Ya tiene una grulla.

Instrucciones gráficas:



Definición de programación

El uso de cualquiera de estos conjuntos de instrucciones debería dar como resultado una grulla y, si se ha seguido el proceso, ya ha aplicado un algoritmo. La única diferencia es la forma en que leemos la formalización de ese conjunto de instrucciones y que nos lleva a la **programación**. La programación, un término abreviado de *programación informática*, es el acto de formalizar el procesamiento de una serie de acciones en un programa ejecutable. Si hemos convertido las instrucciones anteriores para crear una grulla en un formato que el equipo puede leer y ejecutar, esto significa que estamos programando.

La clave y el primer obstáculo que encontramos en la programación es que debemos utilizar alguna forma de abstracción para comunicarnos eficazmente con nuestro ordenador. Esto adopta la forma de diversos lenguajes de programación, como JavaScript, Python o C. Si podemos escribir un conjunto repetible de instrucciones, como en el caso de la grulla de papiroflexia, solo necesitamos traducirlo para que lo entienda el ordenador. Estamos intentando que el ordenador cree una grulla o incluso multitud de grullas diferentes con ligeras variaciones entre ellas. Este es el poder de la programación: el ordenador ejecutará repetidamente cualquier tarea o conjunto de tareas que le asignemos, sin retrasos ni errores humanos.

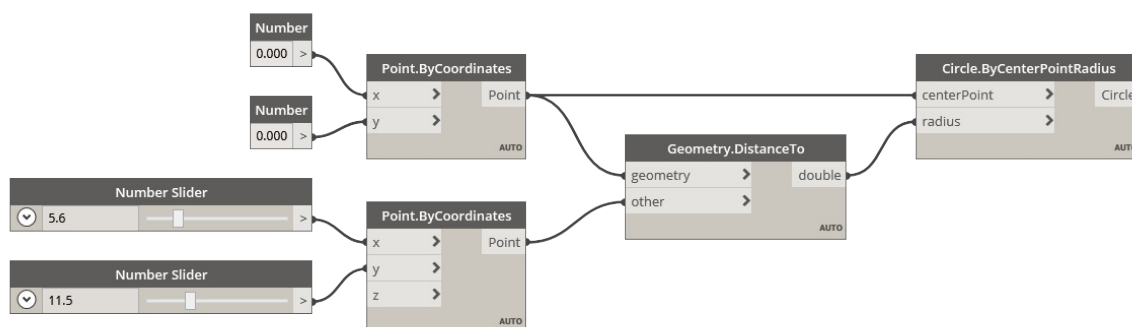
Definición de programación visual

Descargue el archivo de ejemplo que acompaña a este ejercicio (haga clic con el botón derecho y seleccione "Guardar vínculo como..."): [Visual Programming - Circle Through Point.dyn](#). En el Apéndice, se incluye una lista completa de los archivos de ejemplo.

Si se le encomendase la tarea de escribir instrucciones para crear una grulla de papiroflexia, ¿cómo lo haría? ¿Lo haría con gráficos, texto o una combinación de ambos?

Si su respuesta es con gráficos, la **programación visual** es definitivamente para usted. El proceso es básicamente el mismo tanto para la programación como para la programación visual. Utilizan el mismo marco de formalización. Sin embargo, definimos las instrucciones y las relaciones de nuestro programa a través de una interfaz de usuario gráfica (o "visual"). En lugar de escribir texto vinculado a la sintaxis, conectamos nodos preconfigurados. A continuación, se incluye una comparación del mismo algoritmo ("dibujar un círculo a través de un punto") programado tanto con nodos como con código:

Programa visual:



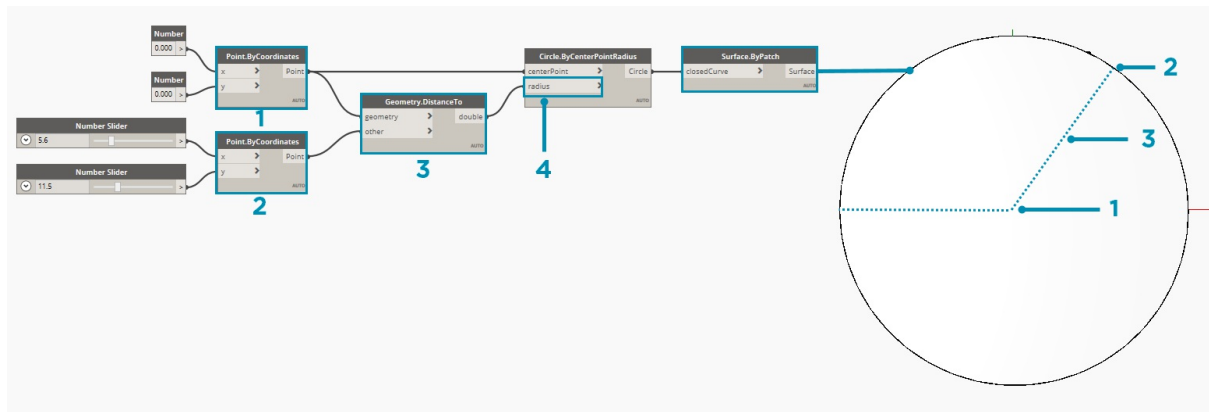
Programa textual:

```

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);

```

Los resultados del algoritmo:



El aspecto visual de la programación permite reducir la barrera en relación con la entrada y, con frecuencia, ofrece mayor claridad a los diseñadores. Dynamo se incluye en el paradigma de la programación visual, pero, como veremos más adelante, podemos seguir utilizando también la programación textual en la aplicación.

¿Qué es Dynamo?

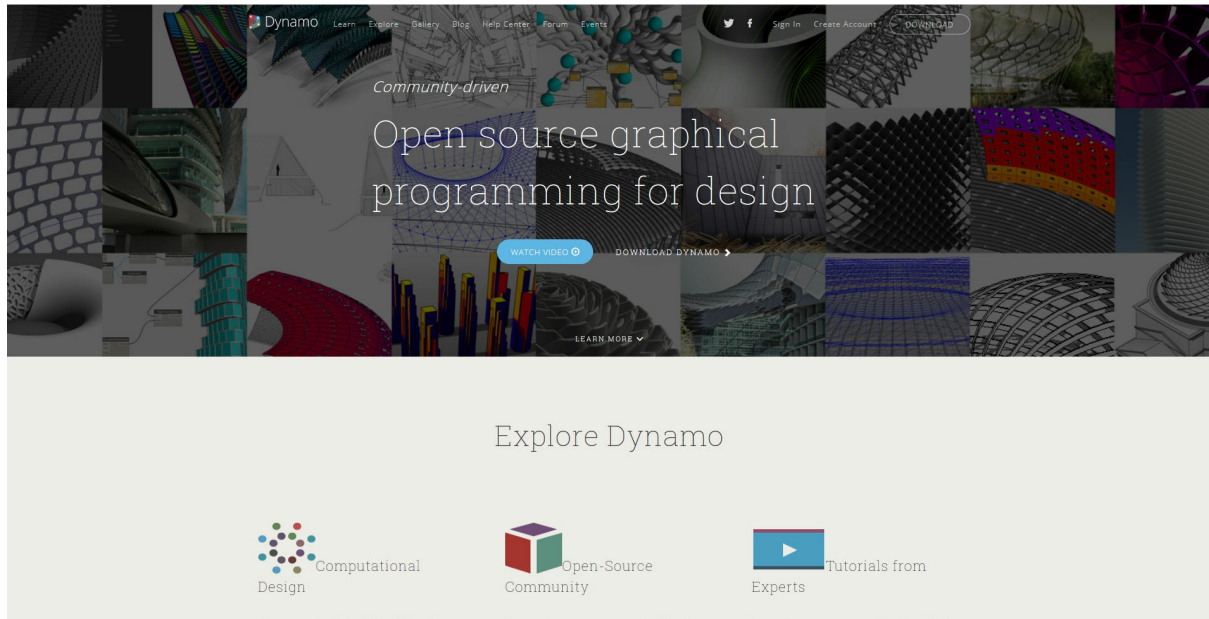
¿Qué es Dynamo?

Dynamo es, casi literalmente, lo que desee que sea. El trabajo con Dynamo puede incluir el uso de la aplicación junto con otro software de Autodesk o no, la implicación en un proceso de programación visual o la participación en una amplia comunidad de usuarios y colaboradores.

La aplicación

Dynamo, la aplicación, es un software que se puede descargar y ejecutar en modo "Sandbox" autónomo o como un módulo de extensión para otro software como Revit o Maya. Se describe como:

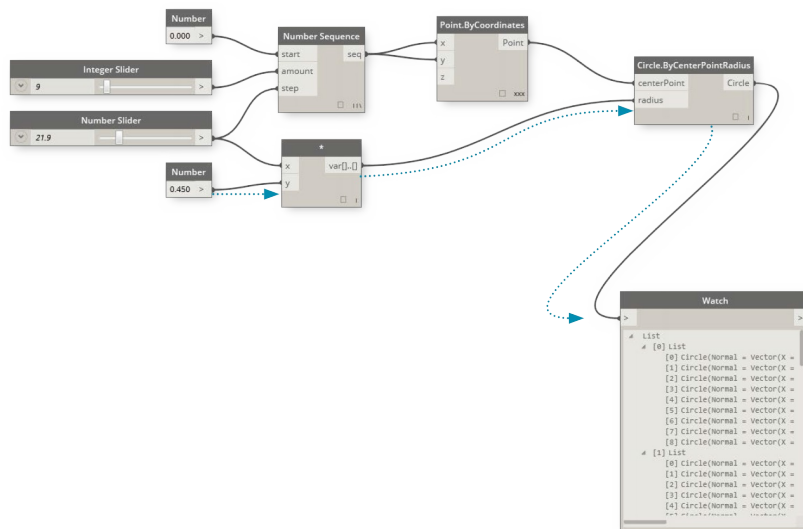
Una herramienta de programación visual cuyo objetivo es ser accesible tanto para los usuarios que son programadores como para los que no lo son. Proporciona a los usuarios la capacidad de definir el comportamiento visualmente con secuencias de comandos, definir lógica personalizada y definir secuencias de comandos mediante diversos idiomas de programación textual.



1. Vea Dynamo en acción con Revit.
2. Descargue el instalador.

El proceso

Una vez instalada la aplicación, Dynamo nos permitirá trabajar en un proceso de programación visual en el que conectaremos elementos para definir las relaciones y las secuencias de acciones que componen algoritmos personalizados. Podemos utilizar nuestros algoritmos para una amplia gama de aplicaciones, desde el procesamiento de datos hasta la generación de geometría, todo en tiempo real y sin necesidad de escribir el mínimo code.



Añada elementos, conéctelos y ya estaremos creando programas visuales.

La comunidad

Dynamo no sería lo que es sin un grupo fiel de fervientes usuarios y colaboradores activos. Participe en la comunidad: siga el blog, añada su trabajo a la galería o hable de Dynamo en el foro.

Dynamo						
Learn		Explore	Gallery	Blog	Help Center	Forum
		Twitter	Facebook	Download	Log In	Search
all categories Latest Top Categories						
Topic	Category	Users	Replies	Views	Activity	
# Dynamo and IronPython - incompatibilities with other applications python The Dynamo team recently became aware of some reports that python in Rhino 6 was not functioning on some machines with Dynamo installed. like this one We've added some information to the Dynamo FAQ wiki entry, and woul... read more	FAQ		8	530	15d	
# How to get help on the Dynamo forums One of the main goals of the Dynamo community is to help others. One way to do that is by answering your questions. But to get the right answer, you'll first have to ask your question in the right way. Although people ar... read more	FAQ		4	9.6k	Dec '17	
# Welcome to the Dynamo Forum Welcome to the new and improved Dynamo Forum! Please make yourself at home and feel free to ask/answer questions about Dynamo, the general purpose graphical algorithm editor for designers, architects, artists, and en... read more	FAQ		21	7.0k	Nov '17	
☒ Place multiple views on multiple sheets by coordinates	Revit		62	4.0k	4m	
Testing			0	3	4m	
Naming Rooms from Area Plans revit	Lists-Logic		4	13	7m	
Zero touch node: how to create a package using visual studio	Developers		0	7	11m	
The Randomized Panel Problem	Share		5	78	17m	
Export schedule columns to Python array dynamo, python, schedule	Developers		17	16	20m	
Model Groups from Linked File	Developers		3	225	1h	
☒ Count Wall Types to Excel			13	27	1h	

La plataforma

Dynamo es una herramienta de programación visual para diseñadores, lo que nos permite crear herramientas que utilicen bibliotecas externas o cualquier producto de Autodesk que tenga una API. Dynamo Sandbox permite desarrollar programas en una aplicación de estilo "Sandbox". Y el ecosistema de Dynamo sigue creciendo.

El código fuente del proyecto es de código abierto, lo que nos permite ampliar su funcionalidad como deseemos. Eche un vistazo al proyecto en GitHub y examine los trabajos en curso de los usuarios que personalizan Dynamo.

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 Clone or download

ramramps	Merge pull request #8714 from ramramps/custom-node-crash-fix	Latest commit 3e87859 2 days ago
.github	Fixing a typo about DynamoRevit repo	9 months ago
doc	Upgrade Samples and fix smoke tests (#8715)	2 days ago
extern	LibG Binaries Update (#8695)	8 days ago
src	Merge branch 'master' of https://github.com/DynamoDS/Dynamo into cust...	2 days ago
test	Deserialize Node View IsSetAsInput property on DynamoMode File Open (#...	2 days ago
tools	Show dictionary docs	29 days ago
.gitattributes	One time renormalization of line endings.	5 years ago
.gitignore	Update .gitignore	8 months ago
.gitmodules	checking ssh key	3 years ago
.travis.yml	Update travis	3 years ago
CONTRIBUTING.md	Fixing typo about DynamoRevit	9 months ago
LICENSE.txt	Clarify dependency licenses and their provenance	2 years ago
README.md	Updated pull request link to new wiki page	9 months ago
appveyor.yml	resolved merge conflicts	3 years ago
dynamo-nuget.config	Set nuget dependency version to highest.	2 years ago
dynamo_sublime	Add a sublime text project.	4 years ago

README.md





Dynamo

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

Get Dynamo

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

Develop

Create a Node Library for Dynamo

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

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

Examine, bifurque y empiece a ampliar Dynamo en función de sus necesidades.

Dynamo en acción

DYNAMO EN ACCIÓN

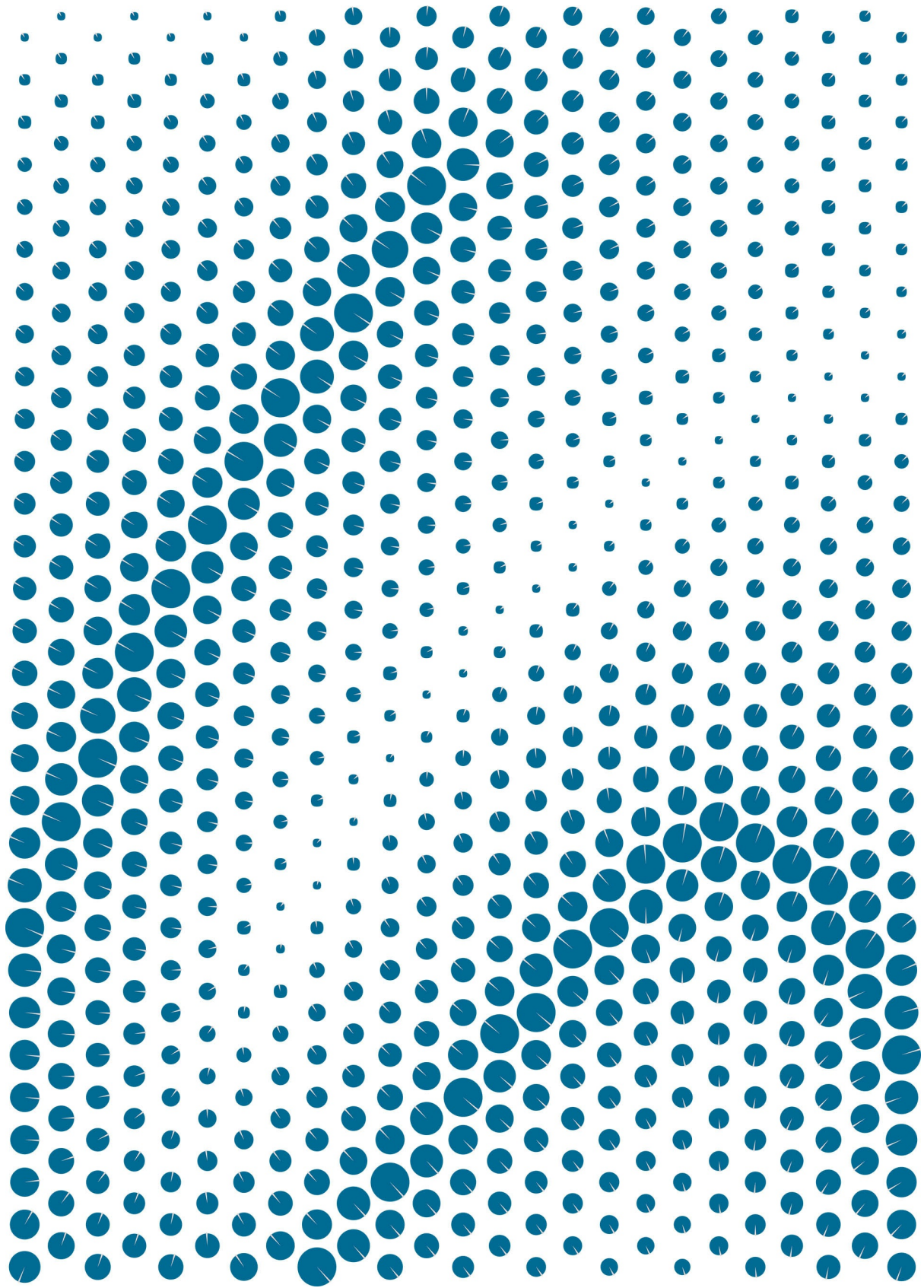
Desde el uso de la programación visual para flujos de trabajo de proyectos hasta el desarrollo de herramientas personalizadas, Dynamo es un aspecto integral de una amplia variedad de aplicaciones interesantes.

[Siga el panel de Dynamo en acción en Pinterest.](#)

¡Hola, Dynamo!

¡HOLA, DYNAMO!

En esencia, Dynamo es una plataforma de programación visual, además de una herramienta de diseño flexible y ampliable. Como puede funcionar como una aplicación independiente o como un complemento para otro software de diseño, podemos utilizarlo para desarrollar una amplia gama de flujos de trabajo creativos. Vamos a instalar Dynamo y empezar a revisar las funciones principales de la interfaz.



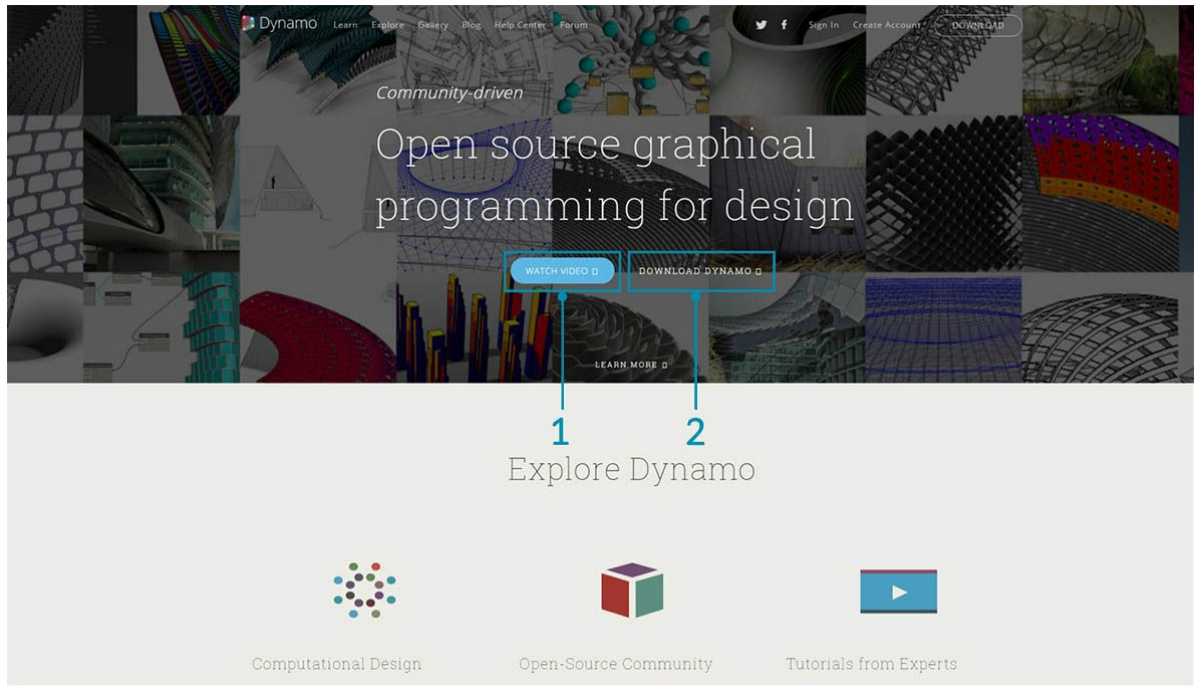
Instalación e inicio de Dynamo

Instalación e inicio de Dynamo

Dynamo es un proyecto de desarrollo activo de código abierto con instaladores descargables tanto para versiones oficiales como para versiones preliminares, es decir, versiones de "compilación diaria". Descargue la versión oficial para empezar o contribuya al futuro de Dynamo a través de las compilaciones diarias o del proyecto de GitHub.

Descarga

Para descargar la versión oficial de Dynamo, visite el [sitio web de Dynamo](#). Inicie la descarga al instante. Para ello, haga clic en la página de inicio o vaya a la página de descarga específica.



1. Vea un vídeo sobre el diseño computacional con Dynamo para arquitectura.
2. O vaya a la página de descarga.

Aquí puede descargar las versiones de desarrollo de "vanguardia" o acceder al proyecto de [Dynamo en GitHub](#).



Dynamo

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

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

1 → [DOWNLOAD](#)

Version 1.3.2

DYNAMO STUDIO

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

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

[BUY OR TRY](#)

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

Pre-Release Daily Builds

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

2 →

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

[VIEW ALL BUILDS](#)

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 → [DISCOVER PACKAGES](#)

Get Involved with Open Source

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

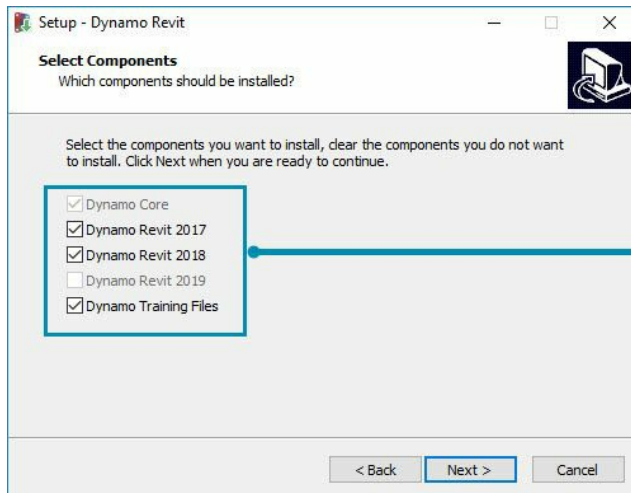
4 → [JOIN THE COMMUNITY](#)

[VIEW SOURCE ON GITHUB](#) [REPORT BUGS](#)

1. Descargue el instalador de la versión oficial.
2. Descargue los instaladores de las compilaciones diarias.
3. Consulte los paquetes personalizados de una comunidad de desarrolladores.
4. Participe en el desarrollo de Dynamo en GitHub.

Instalación

Vaya al directorio del instalador descargado y ejecute el archivo ejecutable. Durante el proceso de instalación, puede personalizar los componentes que se van a instalar.

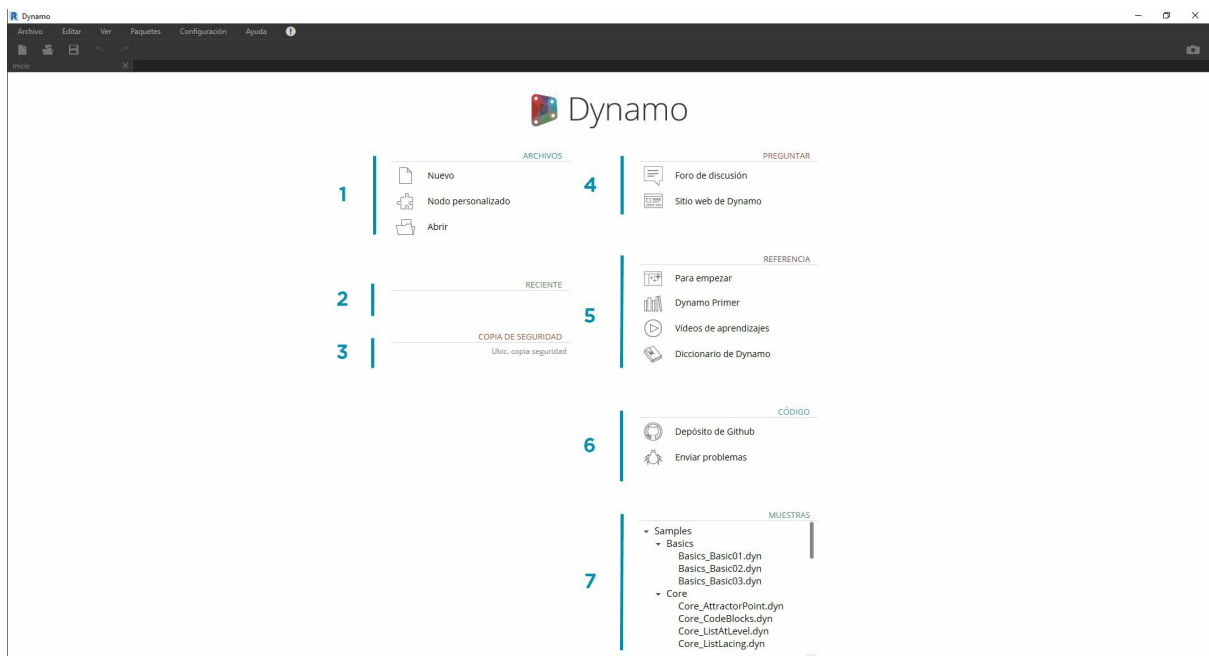


1. Seleccione los componentes que desee instalar.

Aquí debemos decidir si deseamos incluir los componentes que conectan Dynamo a otras aplicaciones instaladas, como Revit. Para obtener más información sobre la plataforma Dynamo, consulte el **capítulo 1.2**.

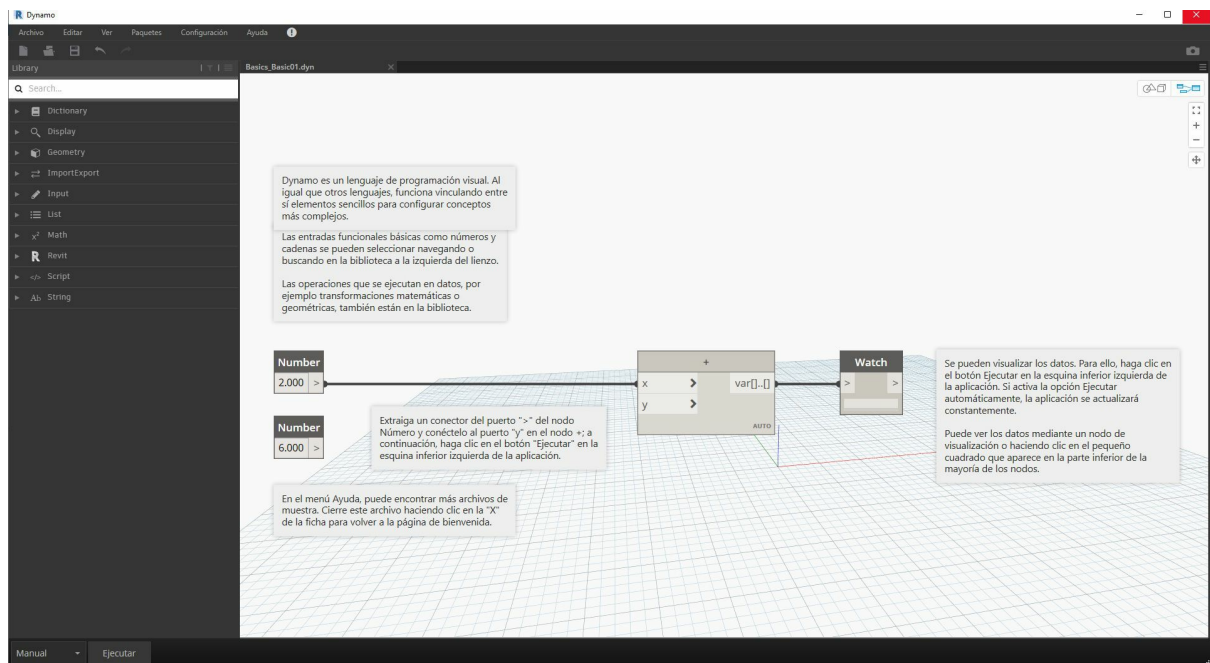
Inicio

Para iniciar Dynamo, vaya a \Archivos de programa\Dynamo\Dynamo Revit\x.y y, a continuación, seleccione DynamoSandbox.exe. Se abre la versión autónoma y se muestra la *página de inicio* de Dynamo. En esta página, se muestran los menús y la barra de herramientas estándar, así como una colección de accesos directos que nos permiten acceder a las funciones de archivo o recursos adicionales.



1. Archivos: inicie un archivo nuevo o abra uno existente.
2. Reciente: desplácese por los archivos recientes.
3. Copia de seguridad: acceda a las copias de seguridad.
4. Preguntar: obtenga acceso directo al foro de usuarios o al sitio web de Dynamo.
5. Referencia: amplíe sus conocimientos con los recursos de aprendizaje adicionales.
6. Código: participe en el proyecto de desarrollo de código abierto.
7. Muestras: consulte los ejemplos que se incluyen en la instalación.

Abra el primer archivo de ejemplo para abrir el primer espacio de trabajo y confirmar que Dynamo funciona correctamente. Haga clic en Muestras > Basics > **Basics_Basic01.dyn**.



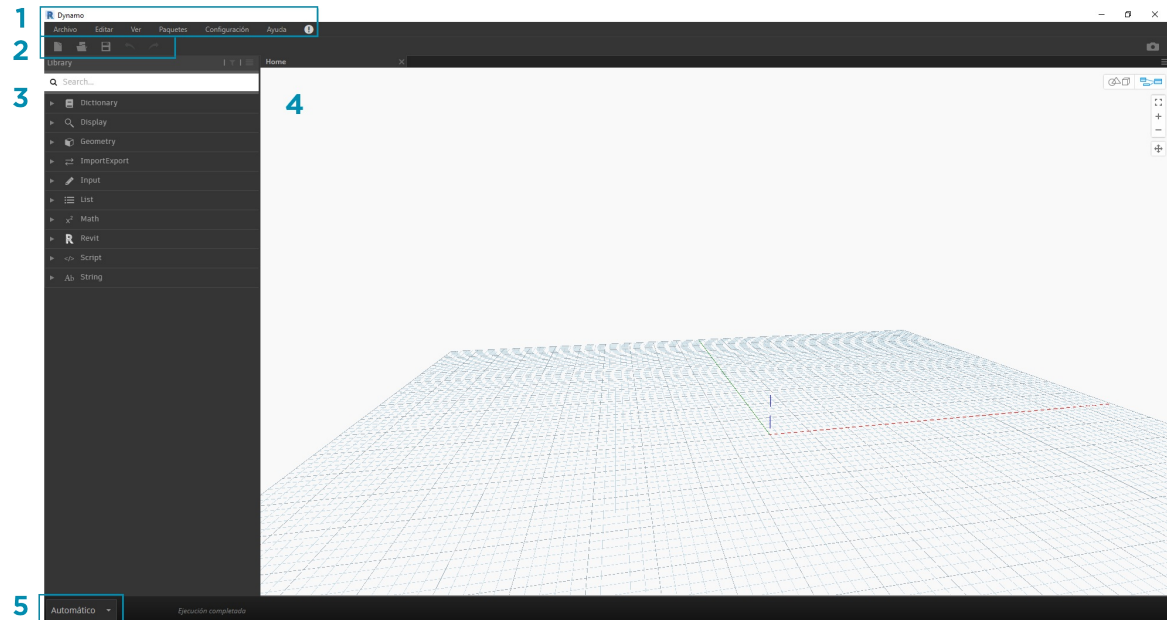
1. Confirme que la barra de ejecución indica "Automático" o haga clic en Ejecutar.
2. Siga las instrucciones y conecte el nodo **Number** al nodo +.
3. Confirme que este nodo Watch muestra un resultado.

Si este archivo se carga correctamente, debería poder ejecutar su primer programa visual con Dynamo.

Interfaz del usuario

La interfaz de usuario de Dynamo

La interfaz de usuario (IU) de Dynamo se organiza en cinco regiones principales, siendo la mayor de ellas el espacio de trabajo en el que se crean los programas visuales.

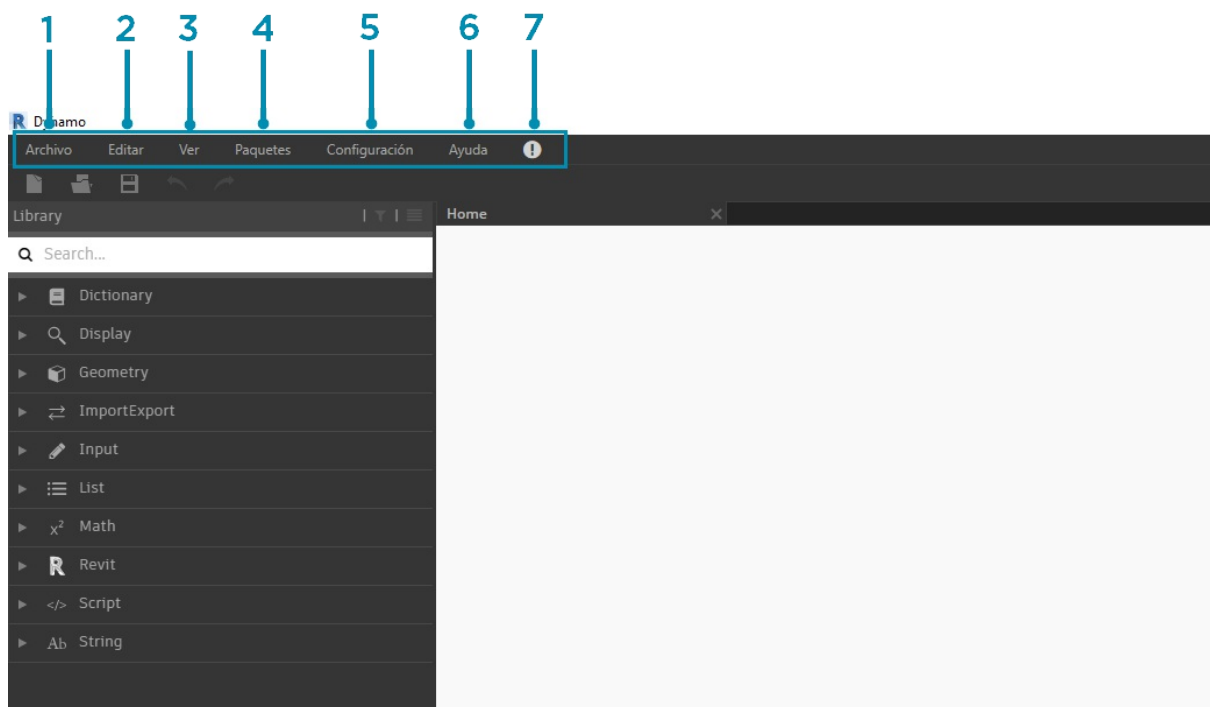


1. Menús
2. Barra de herramientas
3. Biblioteca
4. Espacio de trabajo
5. Barra de ejecución

Examinemos de forma más detallada la interfaz de usuario y veamos las funciones de cada región.

Menús

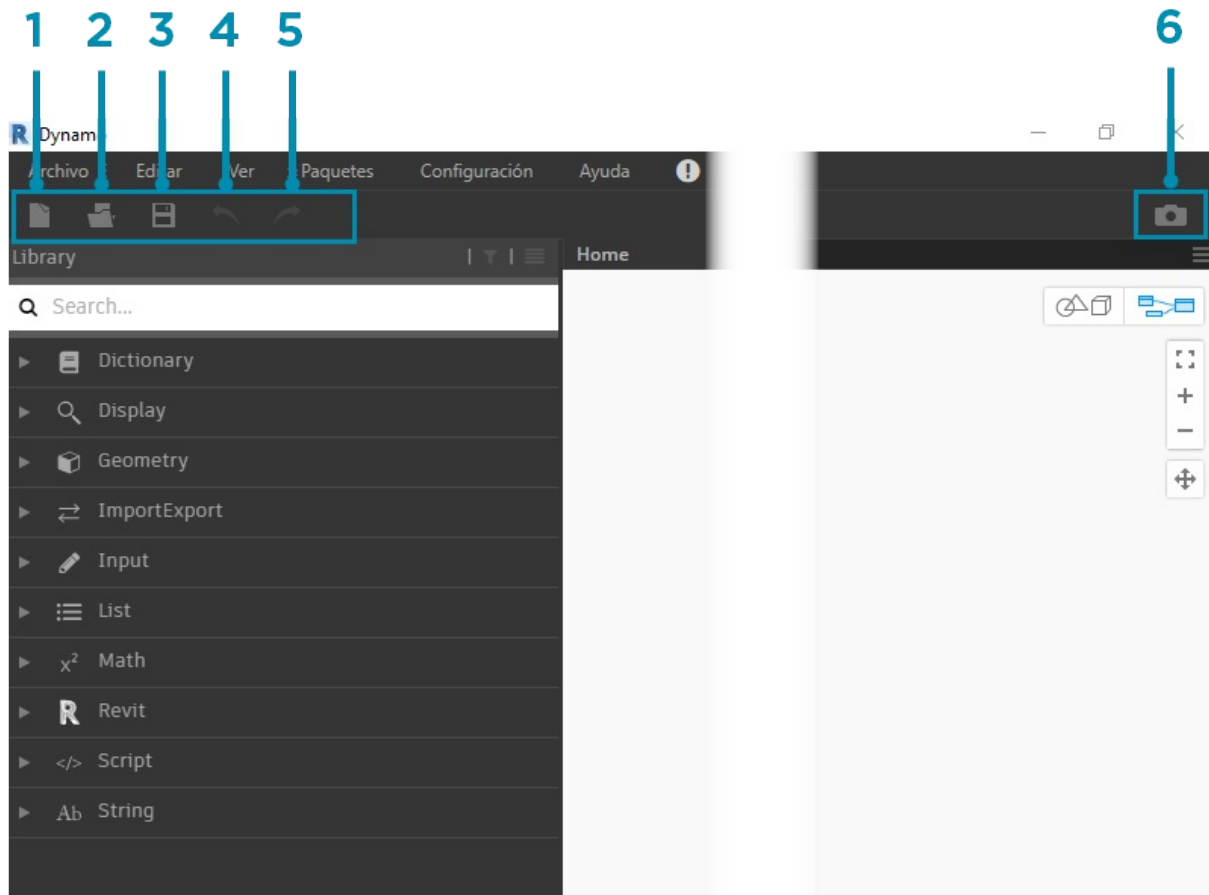
En los menús desplegables, se encuentran algunas de las funciones básicas de la aplicación Dynamo. Al igual que la mayoría del software Windows, las acciones relacionadas con la administración de archivos y las operaciones de selección y contenido se encuentran en los dos primeros menús. El resto de los menús son más específicos de Dynamo.



1. Archivo
2. Editar
3. Vista
4. Paquetes
5. Configuración
6. Ayuda
7. Notificaciones

Barra de herramientas

La barra de herramientas de Dynamo contiene una serie de botones para acceder rápidamente a los archivos, así como a los comandos Deshacer (Ctrl + Z) y Rehacer (Ctrl + Y). En el extremo derecho, hay otro botón que exporta una instantánea del espacio de trabajo, lo que resulta muy útil para la documentación y el uso compartido.



1. Nuevo: cree un nuevo archivo .dyn.
2. Abrir: abra un archivo .dyn (espacio de trabajo) o .dyf (nodo personalizado) existente.
3. Guardar/Guardar como: guarde el archivo .dyn o .dyf activo.
4. Deshacer: deshaga la última acción.
5. Rehacer: rehaga la siguiente acción.
6. Exportar espacio de trabajo como imagen: exporte el espacio de trabajo visible como un archivo PNG.

Biblioteca

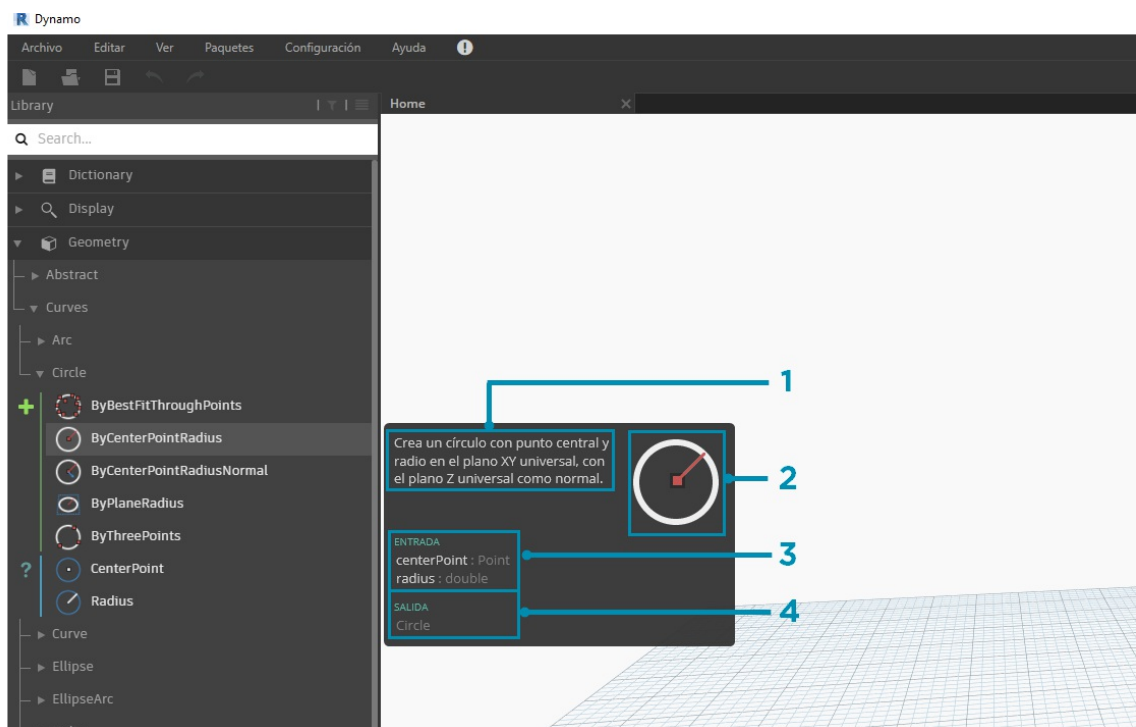
La biblioteca contiene todos los nodos cargados, incluidos los nodos por defecto que se suministran con la instalación, así como los paquetes o los nodos personalizados cargados de forma adicional. Los nodos de la biblioteca se organizan de forma jerárquica en bibliotecas, categorías y, si es necesario, en subcategorías en función de si los nodos **crean** datos, ejecutan una **acción** o **consultan** datos.

Exploración

Por defecto, la **biblioteca** contiene ocho categorías de nodos. **Núcleo** y **Geometría** son excelentes menús para empezar a explorar, ya que contienen la mayor cantidad de nodos. La exploración a través de estas categorías es la manera más rápida de conocer la jerarquía de lo que podemos añadir a nuestro espacio de trabajo y la mejor forma de descubrir nuevos nodos que no ha utilizado antes.

Ahora nos centraremos en la colección por defecto de nodos, pero tenga en cuenta que ampliaremos posteriormente esta biblioteca con nodos personalizados, bibliotecas adicionales y Package Manager.

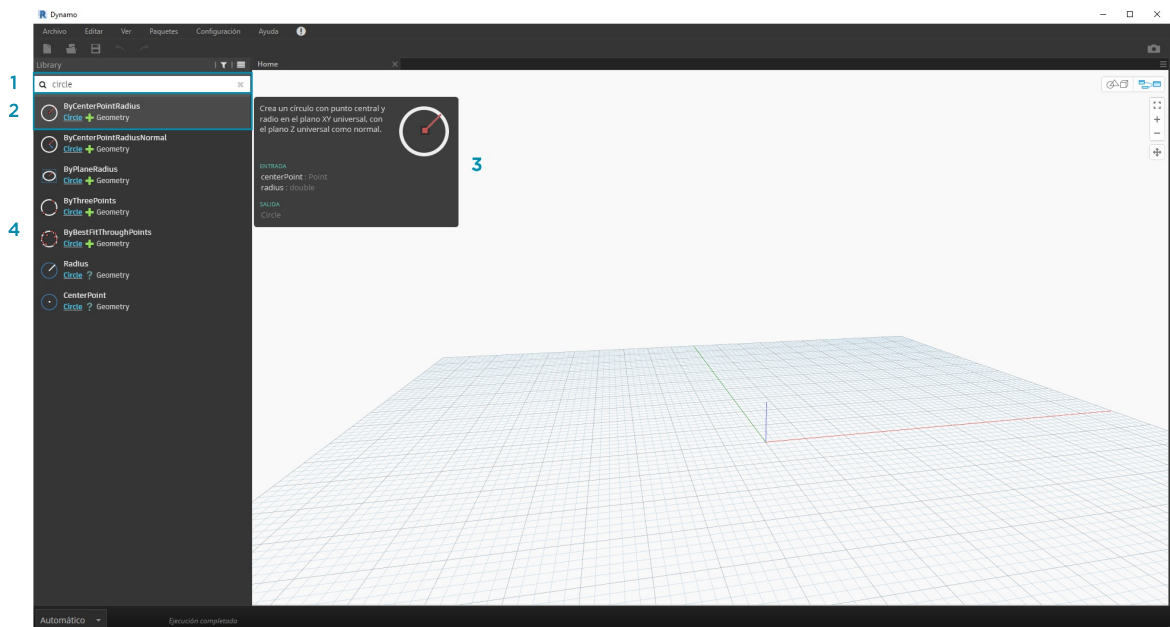
En el mismo menú **Círculo**, coloque el cursor sobre **ByCenterPointRadius**. En la ventana, se muestra información más detallada sobre el nodo, además de su nombre e icono. Esto nos permite conocer rápidamente lo que realiza el nodo, lo que necesitará para las entradas y lo que proporcionará como salida.



1. Descripción: descripción del nodo en lenguaje normal.
2. Icono: versión más grande del icono en el menú Biblioteca.
3. Entrada(s): el nombre, y el tipo y la estructura de datos.
4. Salida(s): el tipo y la estructura de datos.

Búsqueda

Si sabe de forma relativamente específica el nodo que desea añadir al espacio de trabajo, el campo **Buscar** es la mejor herramienta. Cuando no se están modificando parámetros o especificando valores en el espacio de trabajo, el cursor está siempre presente en este campo. Si empieza a escribir, la biblioteca de Dynamo mostrará la coincidencia más precisa seleccionada (con rutas de navegación en relación con las categorías de nodos) y una lista de coincidencias alternativas de la búsqueda. Si pulsa Intro o hace clic en el elemento del navegador truncado, el nodo resaltado se añade al centro del espacio de trabajo.

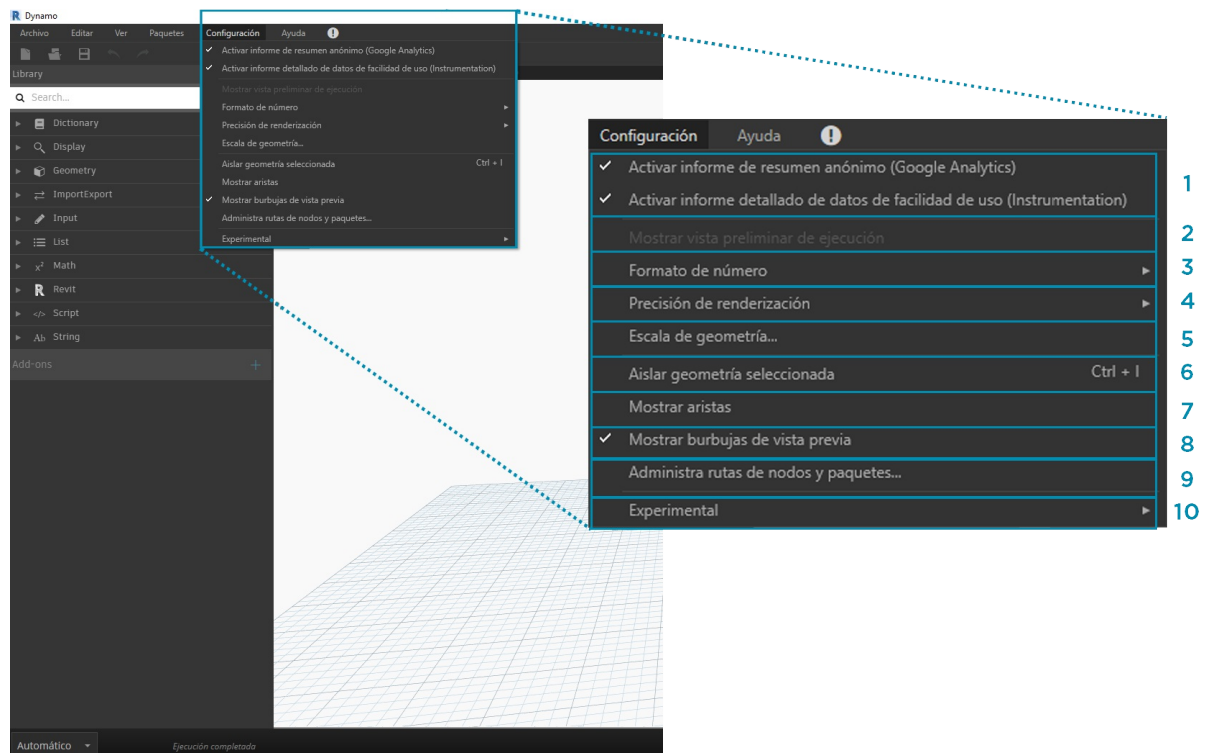


1. Campo de búsqueda
2. Resultado más preciso/seleccionado
3. Coincidencias alternativas

Configuración

En el menú **Configuración**, se encuentran diversas opciones de configuración, desde geométricas hasta de usuario. Aquí puede optar o no por compartir los

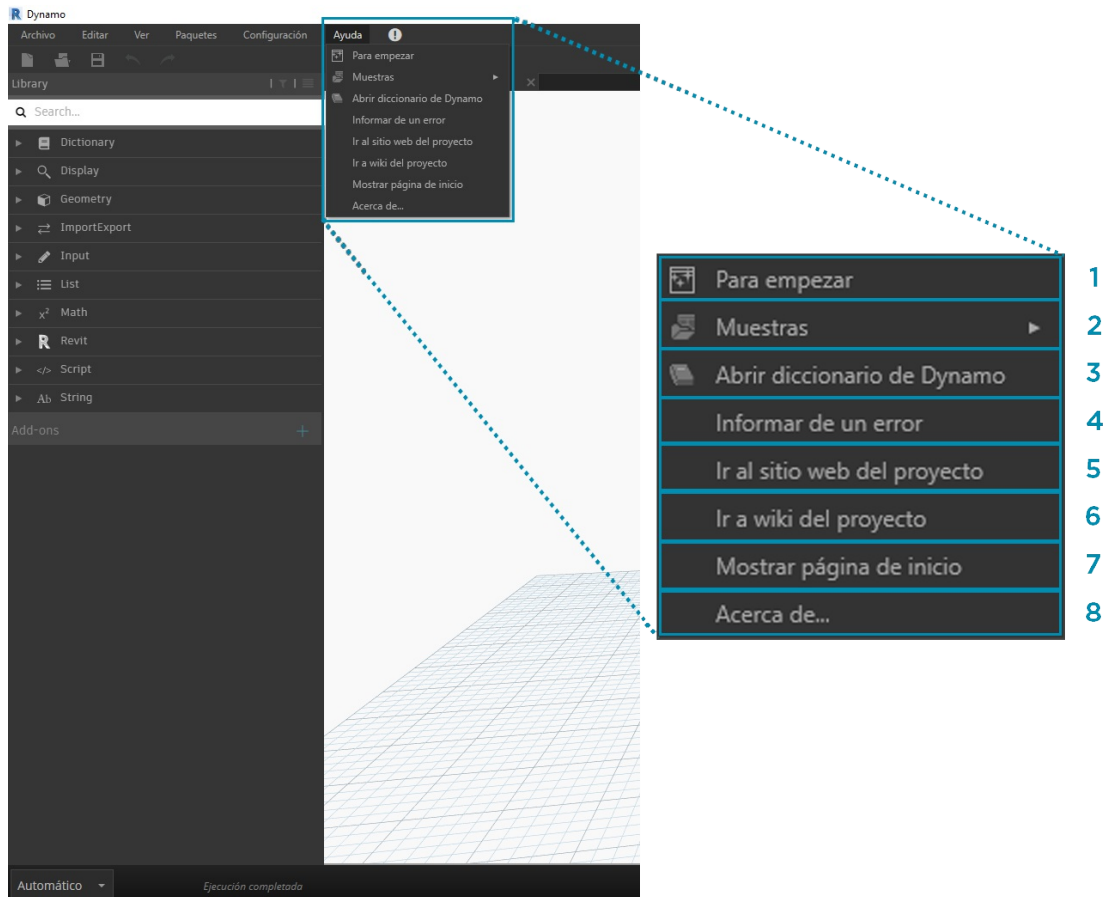
datos de usuario con el fin de mejorar Dynamo, así como definir la precisión del separador decimal y la calidad de renderización de la geometría de la aplicación.



1. Activación de informes: opciones para compartir datos de usuario con el fin de mejorar Dynamo.
2. Mostrar vista preliminar de ejecución: obtenga una vista preliminar del estado de ejecución del gráfico. Los nodos programados para la ejecución se resaltarán en el gráfico.
3. Opciones de formato de número: cambie la configuración de decimales del documento.
4. Precisión de renderización: aumente o reduzca la calidad de renderización del documento.
5. Escala de geometría: seleccione el intervalo de geometría en el que está trabajando.
6. Aislar geometría seleccionada: aísle la geometría en segundo plano en función de la selección de nodos.
7. Mostrar/ocultar aristas de geometría: active o desactive las aristas de geometría 3D.
8. Mostrar/ocultar burbujas de vista previa: active o desactive las burbujas de vista previa de datos debajo de los nodos.
9. Administrar rutas de nodos y paquetes: administre las rutas de archivos para que los nodos y los paquetes aparezcan en la biblioteca.
10. Activación de funciones experimentales: utilice las nuevas funciones de la versión beta de Dynamo.

Ayuda

Si no sabe cómo funciona algo, consulte el menú de **Ayuda**. Aquí encontrará los archivos de ejemplo que se incluyen con la instalación, así como acceso a uno de los sitios web de referencia de Dynamo mediante el navegador de Internet. Si es necesario, compruebe la versión de Dynamo instalada y si está actualizada mediante la opción **Acerca de**.



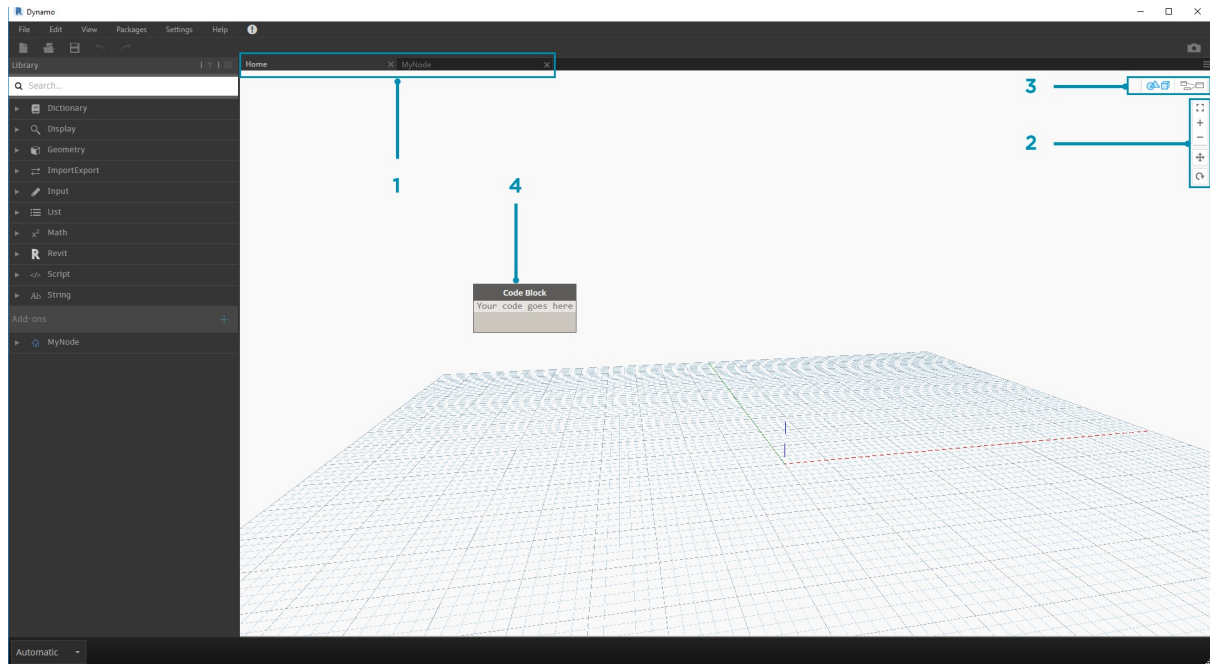
1. Para empezar: una breve introducción al uso de Dynamo.
2. Muestras: archivos de ejemplo de referencia.
3. Abrir diccionario de Dynamo: recursos con documentación sobre todos los nodos.
4. Informar de un error: abra una incidencia en GitHub.
5. Ir al sitio web del proyecto: vea el proyecto de Dynamo en GitHub.
6. Ir a wiki del proyecto: visite la página wiki para obtener información sobre el desarrollo mediante la API de Dynamo con bibliotecas y herramientas.
7. Mostrar página de inicio: regrese a la página de inicio de Dynamo cuando se encuentre en un documento.
8. Acerca de: los datos de la versión de Dynamo.

El espacio de trabajo

El espacio de trabajo

El **espacio de trabajo** de Dynamo es donde se desarrollan los programas visuales, pero también es donde se obtiene una vista preliminar de la geometría resultante. Tanto si trabajamos en un espacio de trabajo de inicio como en un nodo personalizado, podemos desplazarnos con el ratón o con los botones ubicados en la parte superior derecha. Al alternar entre los modos de la parte inferior derecha, se cambia la vista preliminar por la que nos desplazamos.

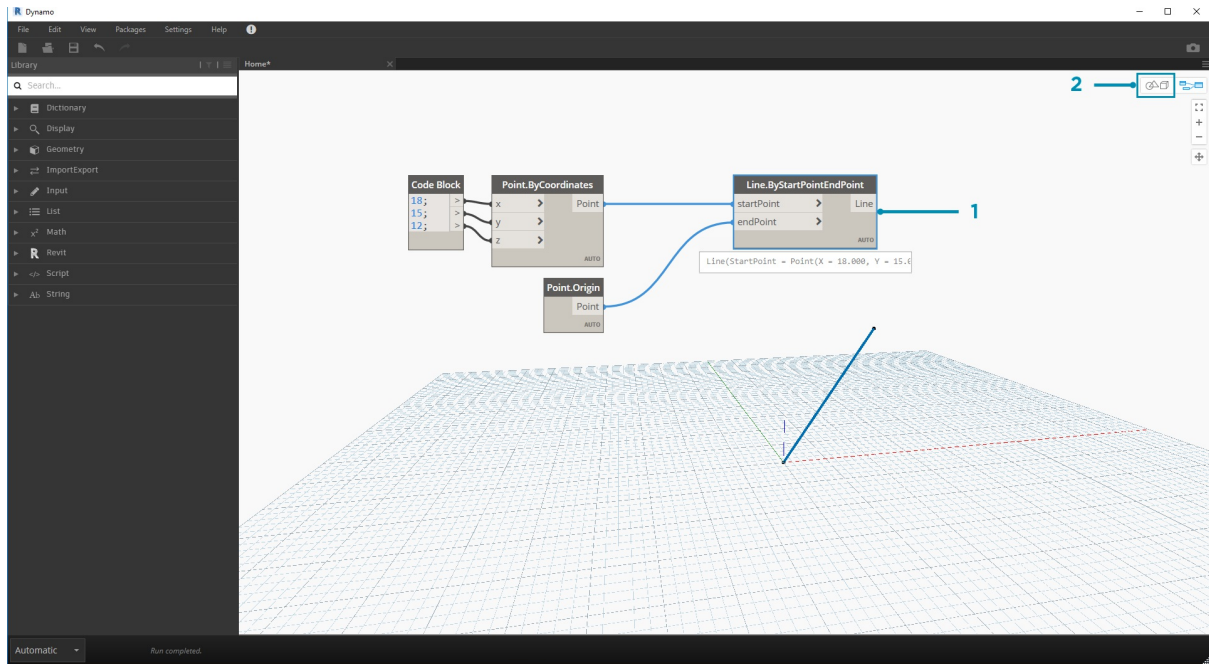
Nota: los nodos y la geometría presentan un orden de objetos, por lo que es posible que tenga objetos renderizados unos encima de otros. Esto puede resultar confuso al añadir varios nodos en una secuencia, ya que pueden renderizarse en la misma posición en el espacio de trabajo.



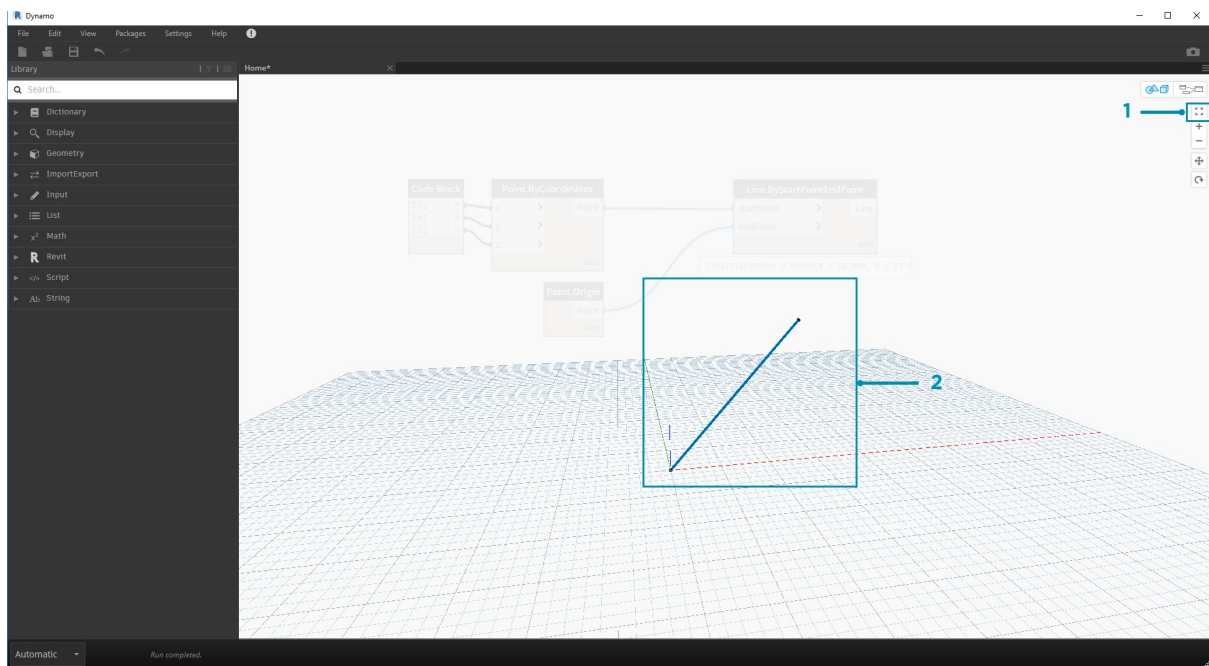
1. Fichas
2. Botones de zoom/encuadre
3. Modo de vista preliminar
4. Haga doble clic en el espacio de trabajo

Fichas

La ficha activa del espacio de trabajo permite desplazarse por el programa y modificarlo. Al abrir un archivo nuevo, se abre por defecto un nuevo espacio de trabajo de **inicio**. También puede abrir un espacio de trabajo de **nodo personalizado** nuevo desde el menú Archivo o haciendo clic con el botón derecho en *Nuevo nodo por selección* cuando se seleccionen nodos (se proporcionará más información sobre esta función más adelante).



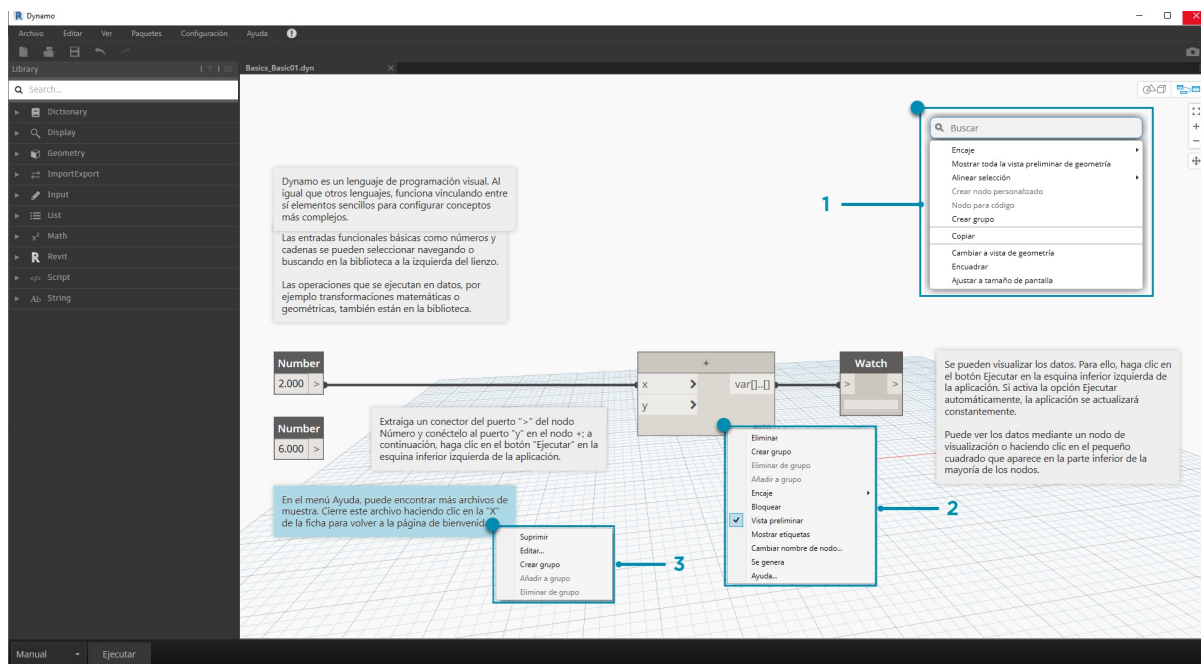
1. Seleccione el nodo correspondiente a la geometría que centrará la vista.
2. Cambie al modo de navegación de vista preliminar 3D.



1. Haga clic en el icono Aplicar zoom a todo situado en la parte superior derecha.
2. La geometría seleccionada se centrará dentro de la vista.

Uso del ratón

En función del modo de vista preliminar activo, los botones del ratón funcionarán de forma diferente. En general, el botón izquierdo del ratón selecciona y especifica entradas, el botón derecho proporciona acceso a las opciones y el botón central permite desplazarse por el espacio de trabajo. Al hacer clic con el botón central, aparecerán opciones basadas en el contexto de la ubicación en la que hacemos clic.



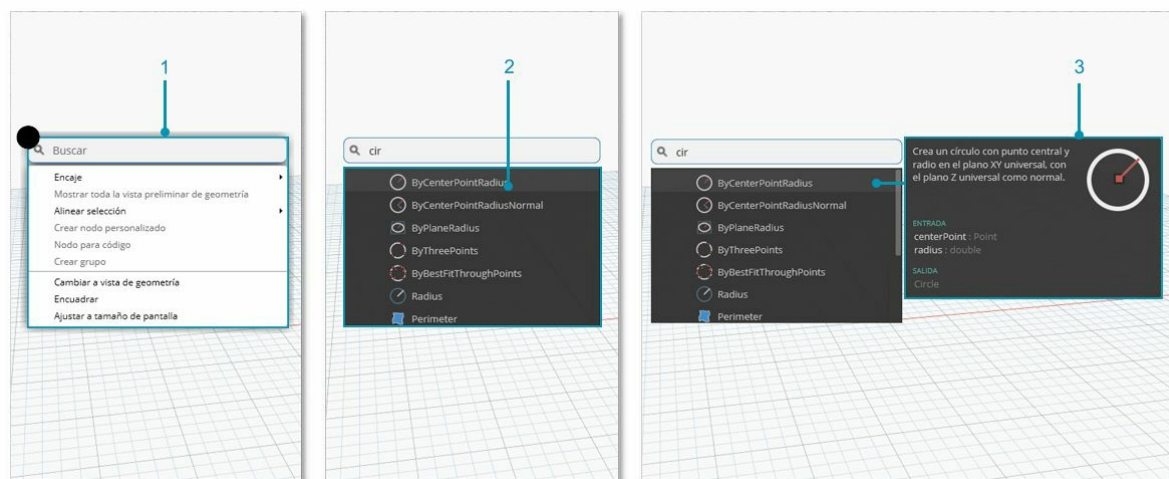
1. Haga clic con el botón derecho en la espacio de trabajo.
2. Haga clic con el botón derecho en un nodo.
3. Haga clic con el botón derecho en una nota.

A continuación, se muestra una tabla de interacciones del ratón por vista preliminar:

Acción del ratón	Vista preliminar de gráfico	Vista preliminar 3D
Hacer clic con el botón izquierdo	Seleccionar	N/A
Hacer clic con el botón derecho	Menú contextual	Opciones de zoom
Hacer clic con el botón central	Encuadre	Encuadre
Desplazamiento	Ampliar/reducir vista	Ampliar/reducir vista
Hacer doble clic	Crear bloque de código	N/A

Búsqueda en el lienzo

Mediante la "búsqueda en el lienzo", puede aumentar considerablemente la velocidad de flujo de trabajo de Dynamo al proporcionar acceso a descripciones de nodos e información de herramientas sin alejarse de su ubicación en el gráfico. Al hacer clic con el botón derecho, puede acceder a todas las funciones útiles de la "búsqueda de biblioteca" desde cualquier ubicación del lienzo en la que esté trabajando.

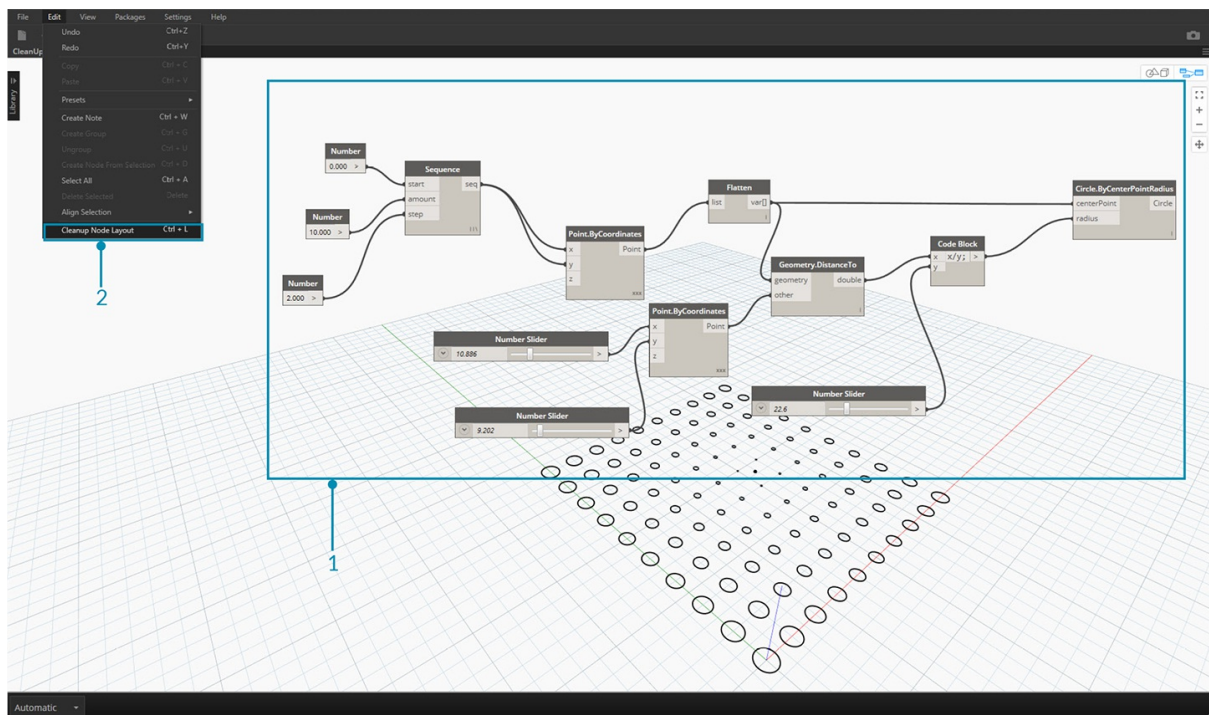


1. Haga clic con el botón derecho en cualquier lugar del lienzo para que aparezca la función de búsqueda. Cuando la barra de búsqueda está vacía, el menú desplegable es un menú de vista preliminar.
2. A medida que escriba en la barra de búsqueda, el menú desplegable se actualizará continuamente para mostrar los resultados de búsqueda más pertinentes.
3. Coloque el cursor sobre los resultados de la búsqueda para obtener las descripciones y la información de herramientas correspondientes.

Limpiar el diseño de nodos

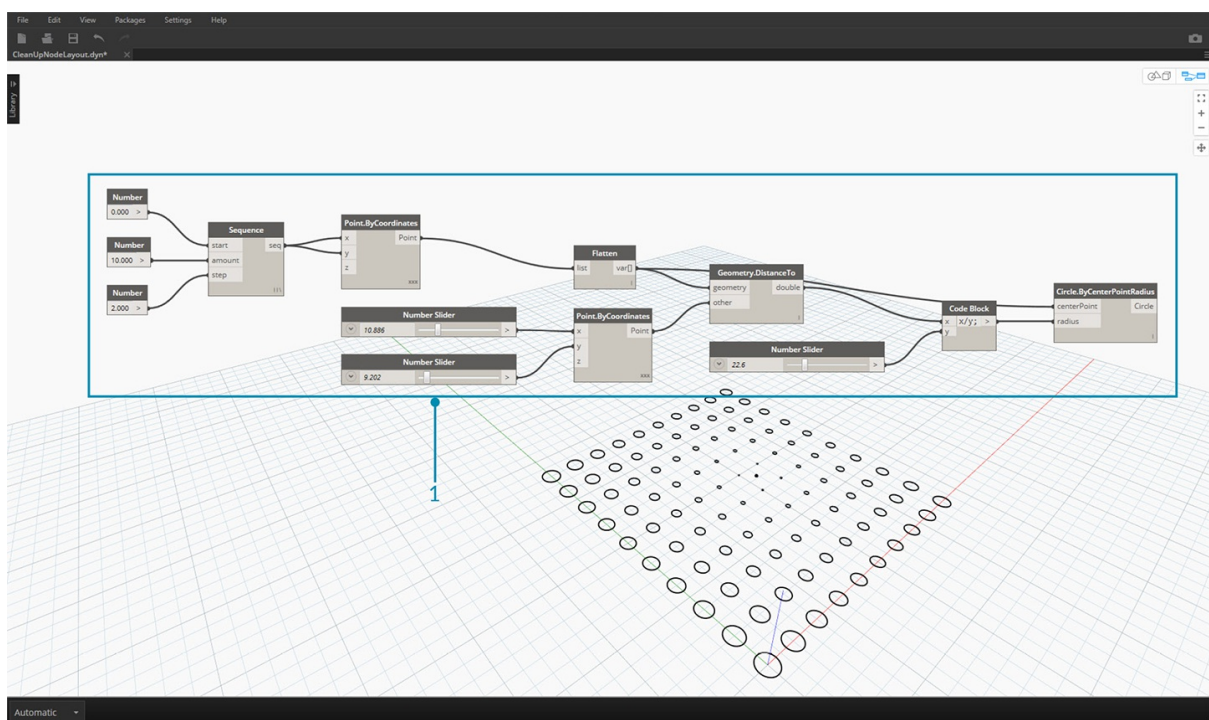
La importancia de la organización del lienzo de Dynamo aumenta a medida que la compilación de archivos gana en complejidad. Aunque está disponible la herramienta **Alinear selección** para trabajar con pequeñas cantidades de nodos seleccionados, Dynamo también incluye la herramienta **Presentación de nodo de limpieza** para ayudar con la limpieza general de archivos.

Antes de la limpieza de nodos



1. Seleccione los nodos que se van a organizar automáticamente o deje todo sin seleccionar para limpiar todos los nodos del archivo.
2. La función Presentación de nodo de limpieza se encuentra en la ficha Editor.

Después de la limpieza de nodos



1. Los nodos se redistribuirán y se alinearán automáticamente, limpiando los nodos escalonados o solapados y alineándolos con los nodos adyacentes.

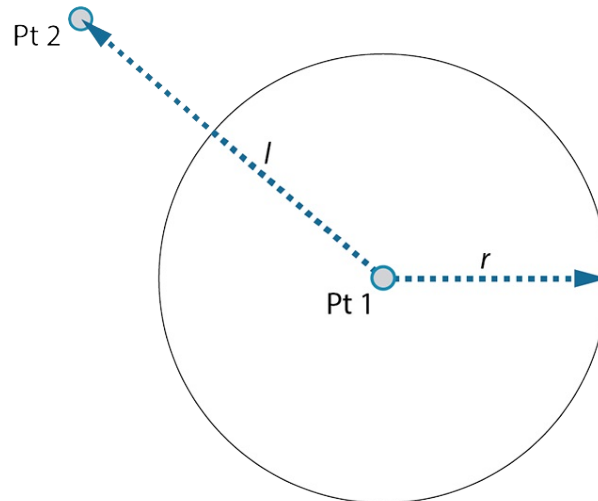
Procedimientos iniciales

PARA EMPEZAR

Ahora que nos hemos familiarizado con el diseño de la interfaz y la navegación por el espacio de trabajo, nuestro siguiente paso es conocer el flujo de trabajo típico para desarrollar un gráfico en Dynamo. Creemos primero un círculo de tamaño dinámico y, a continuación, creemos una matriz de círculos con radios variables.

Definición de objetivos y relaciones

Antes de añadir cualquier elemento al espacio de trabajo de Dynamo, es fundamental conocer en detalle lo que intentamos conseguir y cuáles serán las relaciones más importantes. Recuerde que cada vez que conectamos dos nodos, estamos creando un vínculo explícito entre ellos; podemos cambiar el flujo de datos más tarde, pero, una vez conectados, debemos mantener esa relación. En este ejercicio, vamos a crear un círculo (*objetivo*) donde la entrada de radio se define mediante una distancia hasta un punto cercano (*relación*).

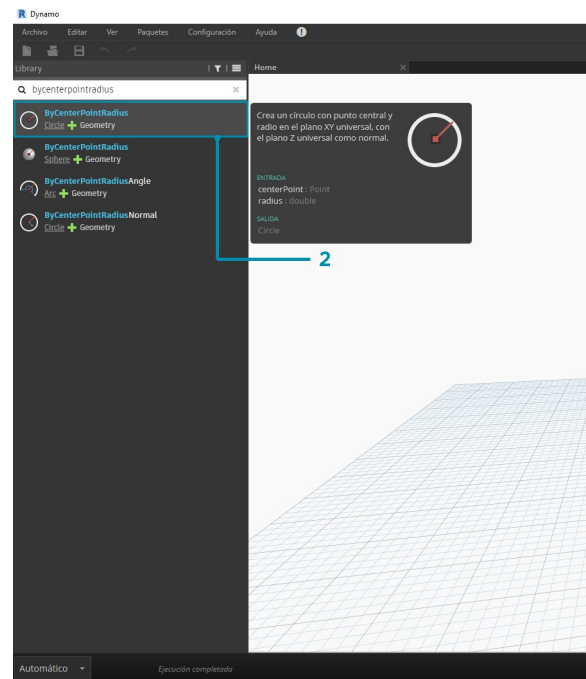
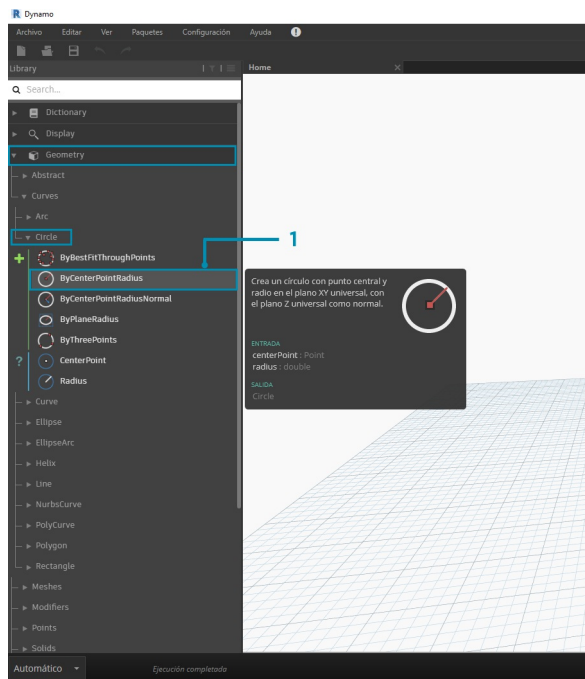


$$\text{Longitud } (l) \propto \text{Radio } (r)$$

Un punto que define una relación basada en la distancia se suele denominar "atractor". Aquí, la distancia a nuestro punto atractor se utilizará para especificar el tamaño del círculo.

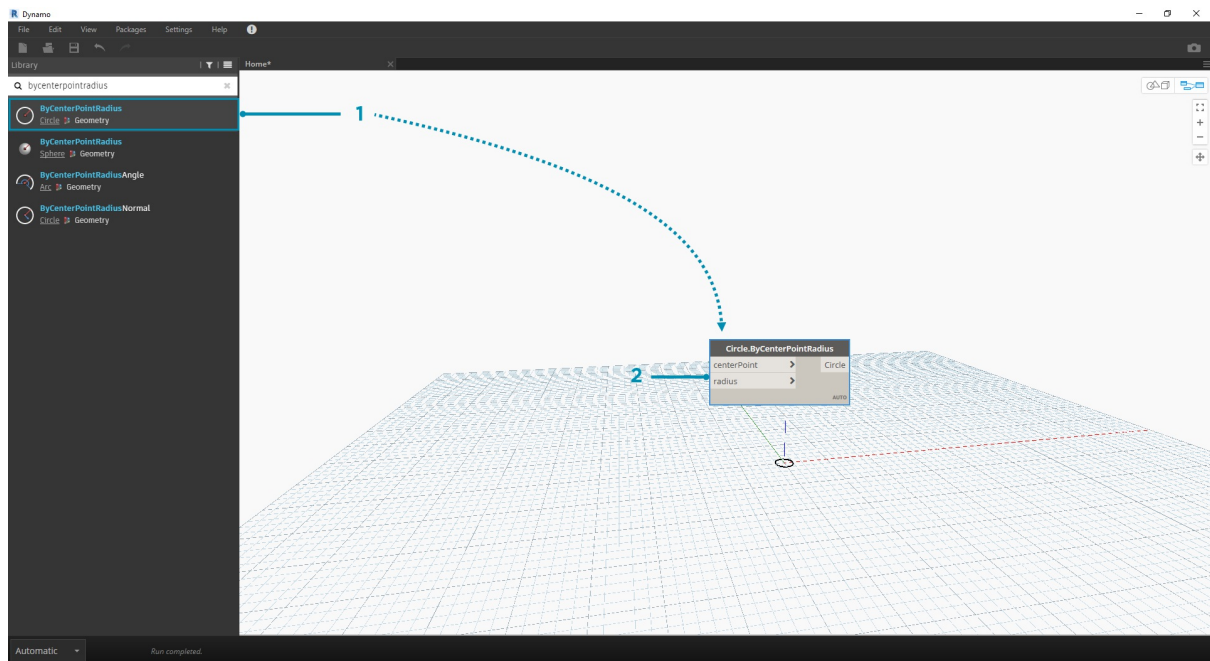
Adición de nodos al espacio de trabajo

Ahora que hemos dibujado un boceto de nuestros objetivos y relaciones, podemos empezar a crear el gráfico. Necesitamos los nodos que representarán la secuencia de acciones que ejecutará Dynamo. Como sabemos que estamos intentando crear un círculo, busquemos primero un nodo que realice esta acción. Mediante el campo Buscar o la exploración de la biblioteca, descubriremos que existen varias formas de crear un círculo.



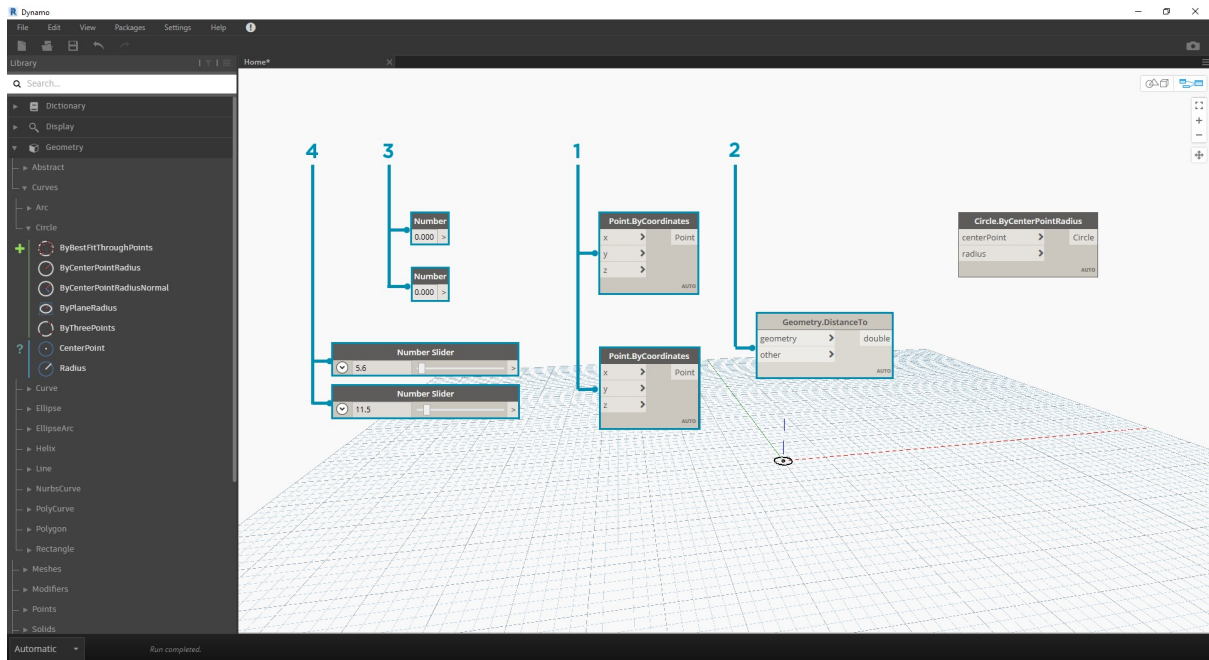
1. Vaya a Geometría > Curvas > Círculo > Circle.ByCenterPointRadius.
2. Buscar > ByCenterPointRadius...

Vamos a añadir el nodo **Circle.CenterByPointRadius** al espacio de trabajo haciendo clic en él en la biblioteca; esta acción debería añadir el nodo en el centro del espacio de trabajo.



1. El nodo Circle.ByPointCenterRadius de la biblioteca.
2. Al hacer clic en el nodo de la biblioteca, este se añade al espacio de trabajo.

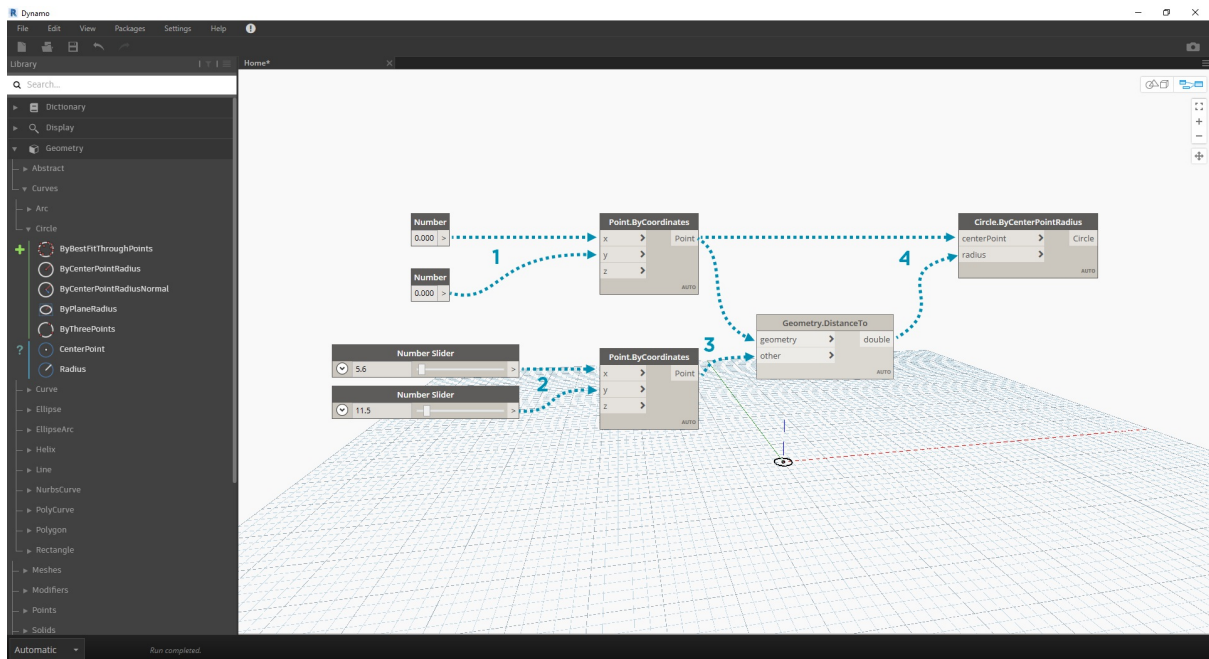
También necesitaremos los nodos **Point.ByCoordinates**, **Number Input** y **Number Slider**.



1. Geometría > Puntos > Punto > **Point.ByCoordinates**
2. Geometría > Geometría > **DistanceTo**
3. Core > Entrada > **Number**
4. Core > Entrada > **Number Slider**

Conexión de nodos con cables

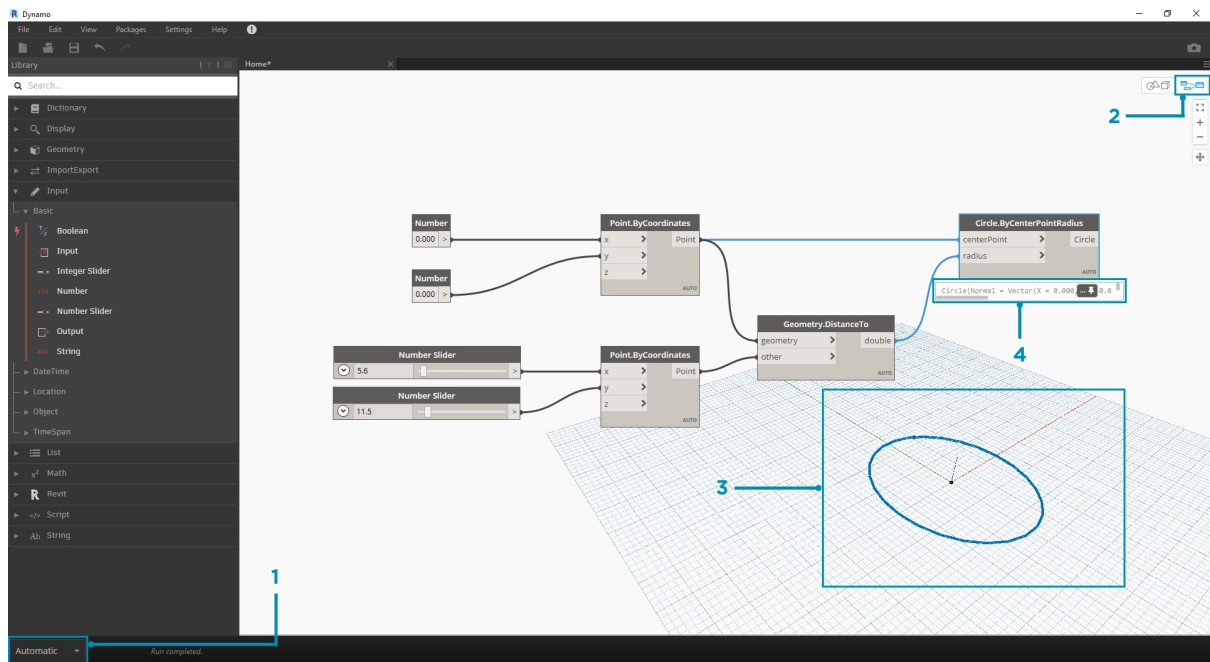
Ahora que tenemos algunos nodos, debemos conectar los puertos de los nodos con cables. Estas conexiones definirán el flujo de datos.



1. De **Number** a **Point.ByCoordinates**
2. De **Number Sliders** a **Point.ByCoordinates**
3. De **Point.ByCoordinates (2)** a **DistanceTo**
4. De **Point.ByCoordinates** y **DistanceTo** a **Circle.ByCenterPointRadius**

Ejecución del programa

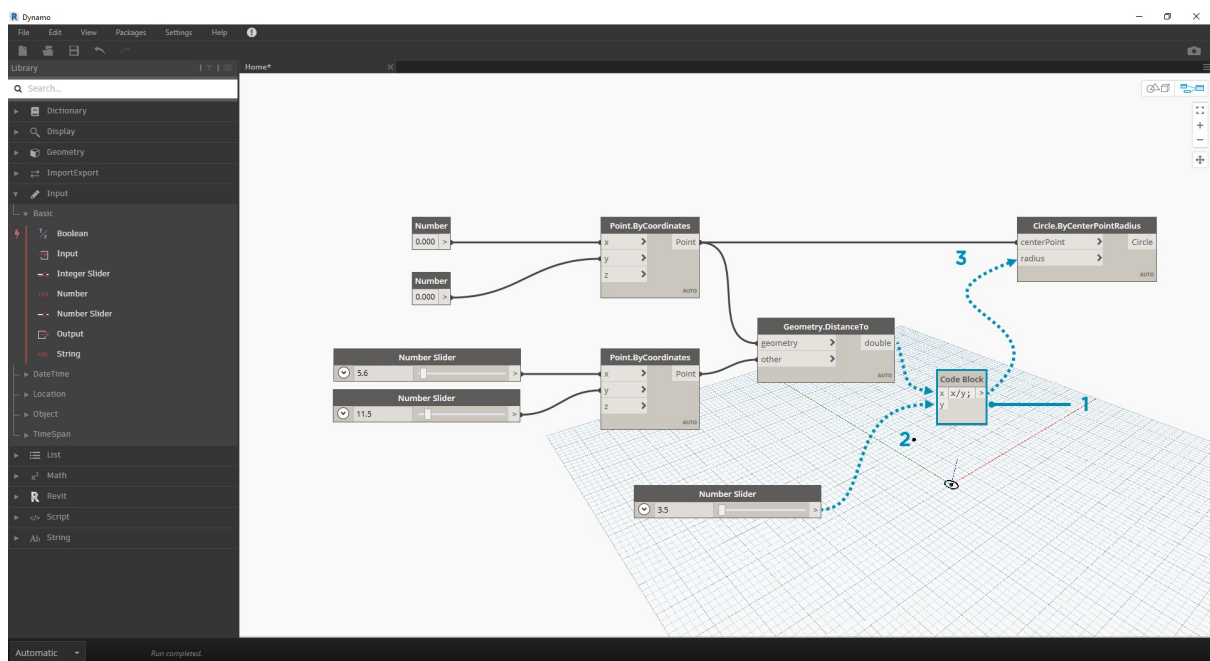
Con el flujo de programa definido, solo tenemos que indicarle a Dynamo que lo ejecute. Una vez ejecutado el programa (automáticamente o al hacer clic en Ejecutar en modo manual), los datos pasarán por los cables y se mostrará la vista preliminar 3D.



1. (Haga clic en Ejecutar): si la barra de ejecución está en modo manual, debemos hacer clic en Ejecutar para ejecutar el gráfico.
2. Vista preliminar de nodos: al colocar el puntero sobre el cuadro situado en la esquina inferior derecha de un nodo, se mostrará un elemento emergente con los resultados.
3. Vista preliminar 3D: si alguno de los nodos crea geometría, esta aparecerá en la vista preliminar 3D.
4. La geometría de salida en el nodo de creación.

Adición de detalles

Si el programa está en funcionamiento, debería aparecer un círculo en la vista preliminar 3D que atraviesa el punto atractor. Esto es genial, pero es posible que desee añadir más detalles o controles. Ajustemos la entrada al nodo del círculo para que podamos calibrar la influencia en el radio. Añada otro nodo **Number Slider** al espacio de trabajo y haga doble clic en un área en blanco del espacio de trabajo para añadir un nodo de **bloque de código**. Edite el campo en el bloque de código. Para ello, especifique X/Y.

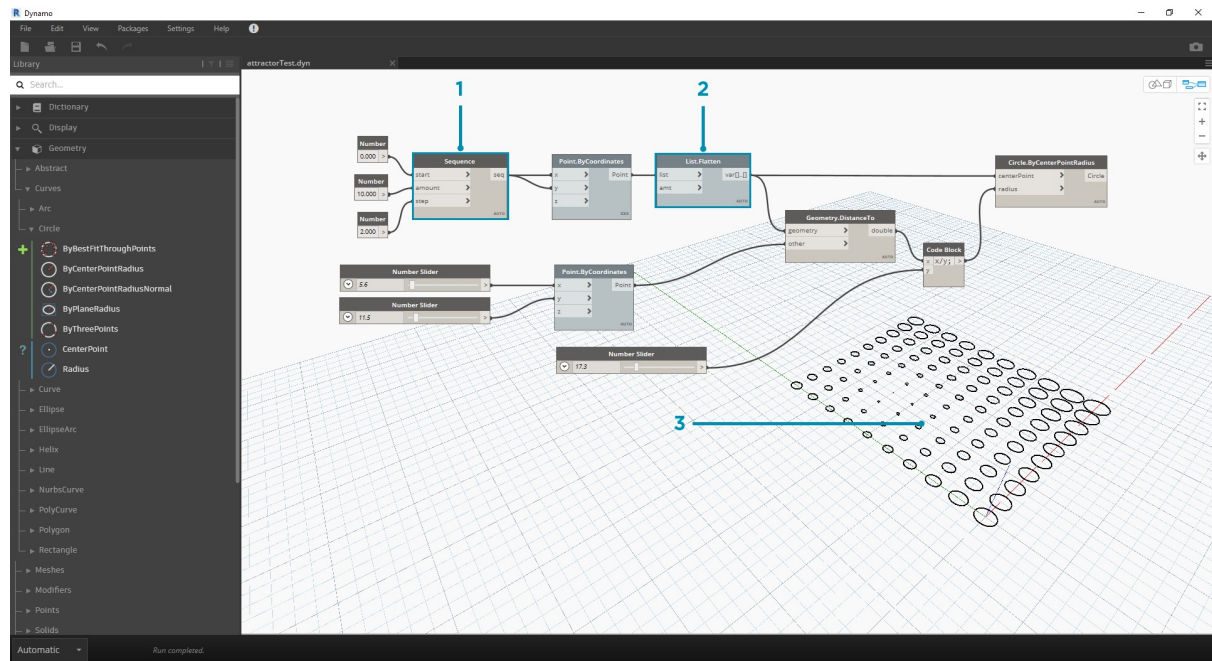


1. Bloque de código
2. De DistanceTo y Number Slider al bloque de código
3. Del bloque de código a Circle.ByCenterPointRadius

Adición de complejidad

Un comienzo sencillo al que se le añada complejidad es una manera eficaz de desarrollar el programa de manera incremental. Una vez que esto funcione para un círculo, apliquemos la potencia del programa a varios círculos. En lugar de un punto central, si utilizamos una rejilla de puntos y acomodamos el cambio en la estructura de datos resultante, el programa creará muchos círculos, cada uno con un valor de radio exclusivo definido por la distancia calibrada

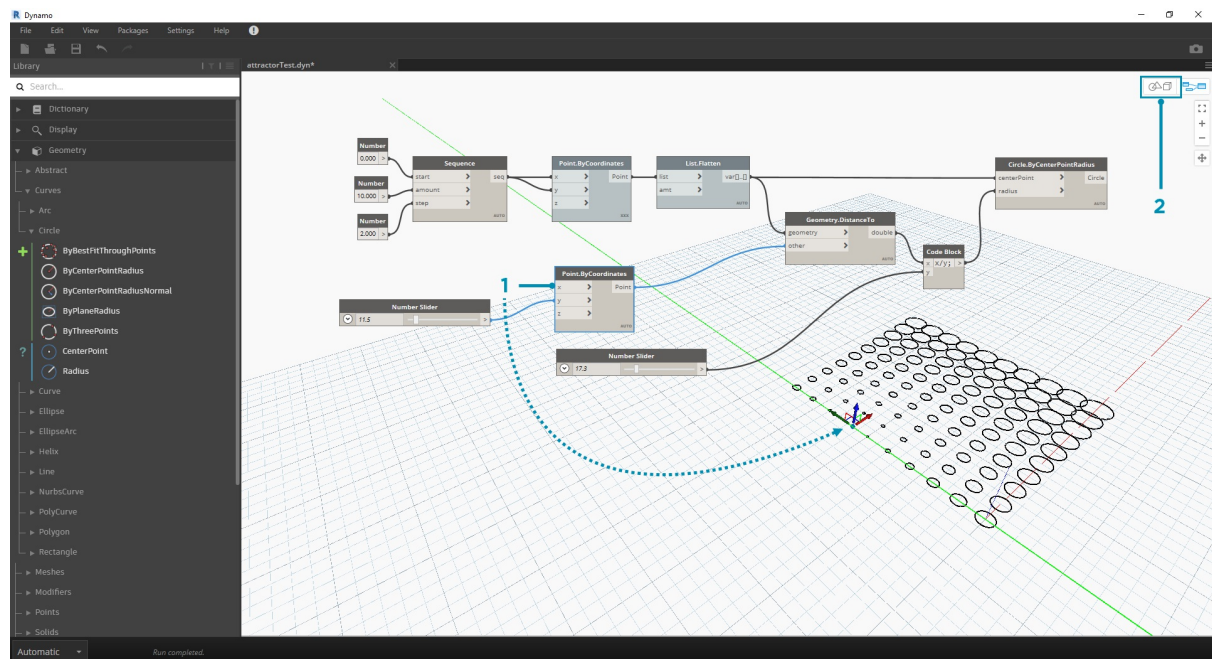
al punto atractor.



1. Añada un nodo **Number Sequence** y sustituya las entradas de **Point.ByCoordinates**: haga clic con el botón derecho en Point.ByCoordinates y seleccione Encaje > Referencia cruzada.
2. Añada un nodo **Flatten** después de Point.ByCoordinates. Para aplanar una lista por completo, mantenga la entrada **amt** en el valor por defecto -1.
3. La vista preliminar 3D se actualizará con una rejilla de círculos.

Ajuste con manipulación directa

En ocasiones, la manipulación numérica no es el enfoque adecuado. Ahora puede empujar y estirar manualmente la geometría de punto al desplazarse por la vista preliminar 3D en segundo plano. También podemos controlar otra geometría construida por un punto. Por ejemplo, **Sphere.ByCenterPointRadius** también es capaz de realizar la manipulación directa. Podemos controlar la ubicación de un punto a partir de una serie de valores X, Y y Z con **Point.ByCoordinates**. Sin embargo, con el enfoque de manipulación directa, puede actualizar los valores de los controles deslizantes moviendo manualmente el punto en el modo de **navegación de vista preliminar 3D**. Esto ofrece un enfoque más intuitivo para controlar un conjunto de valores específicos que identifican la ubicación de un punto.

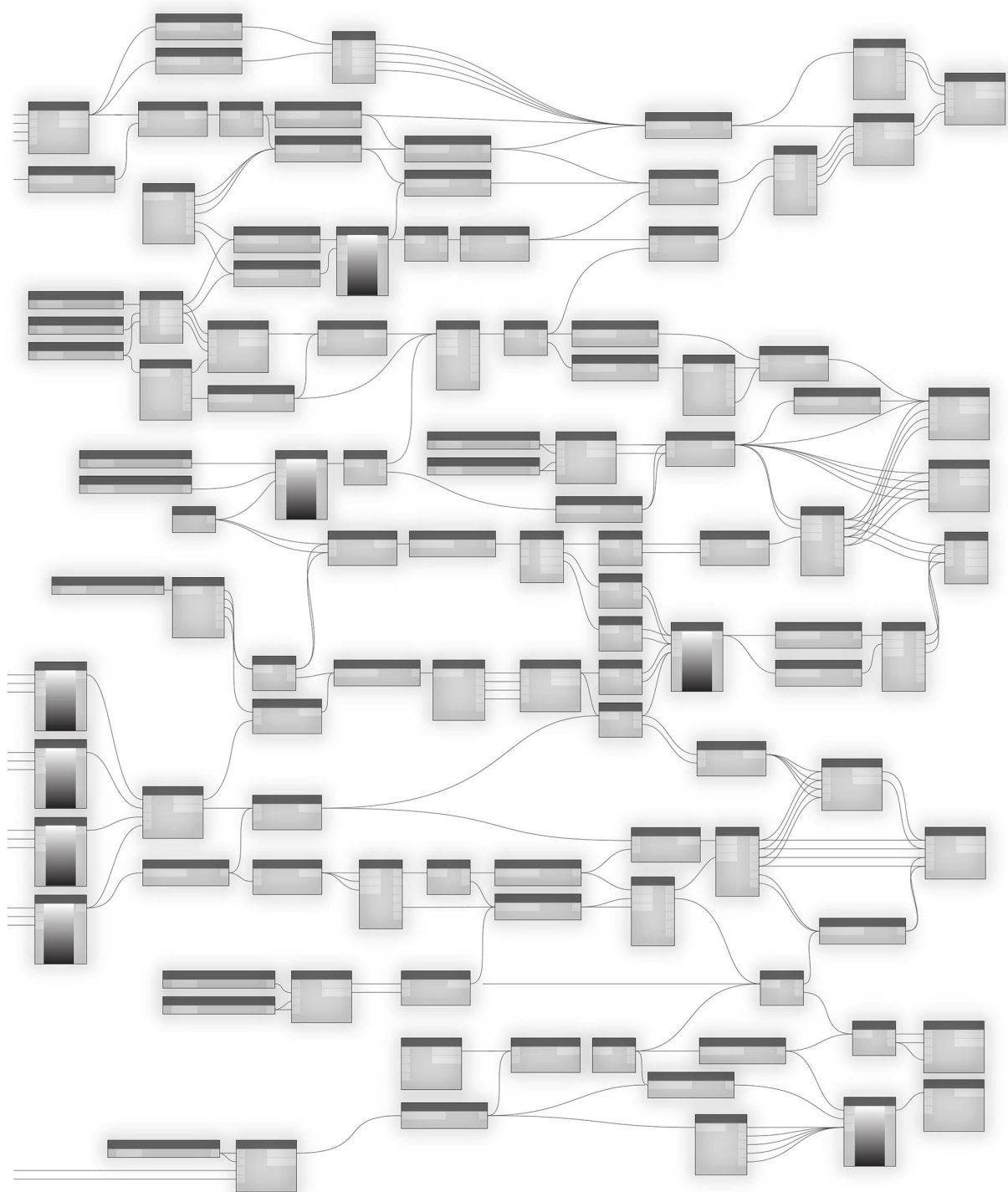


1. Para utilizar la **manipulación directa**, seleccione el panel del punto que se va a mover; aparecerán flechas sobre el punto seleccionado.
2. Cambie al modo de **navegación de vista preliminar 3D**.

Anatomía de un programa visual

ANATOMÍA DE UN PROGRAMA VISUAL

Dynamo permite crear programas visuales en un espacio de trabajo mediante la conexión de nodos con cables para especificar el flujo lógico del programa visual resultante. En este capítulo, se presentan los elementos de los programas visuales, la organización de los nodos disponibles en las bibliotecas de Dynamo, las piezas y los estados de los nodos, y los procedimientos recomendados para los espacios de trabajo.



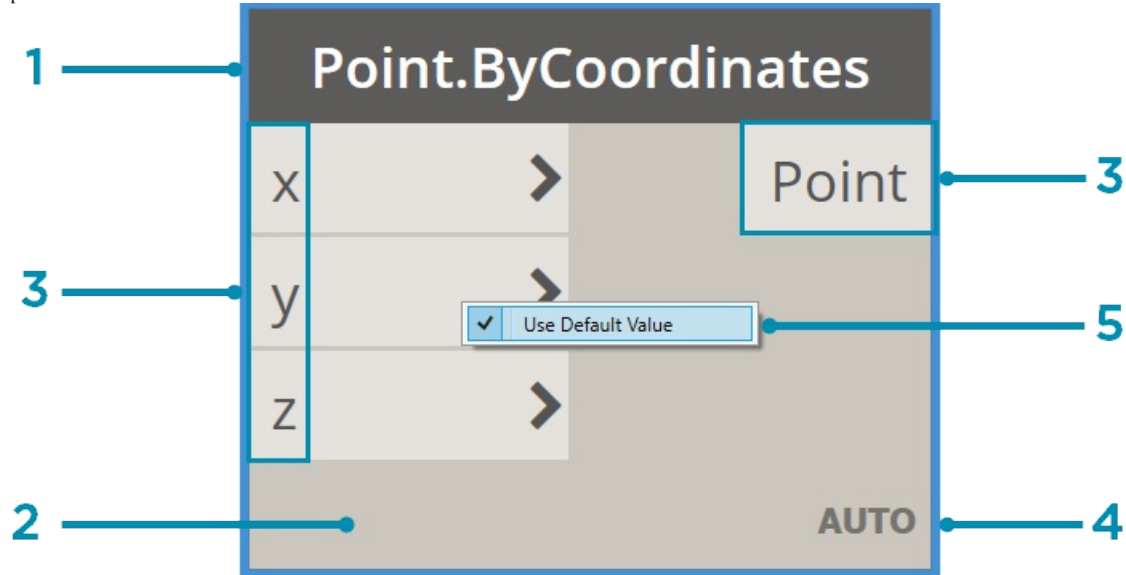
Nodos

Nodos

En Dynamo, los **nodos** son los objetos que se conectan para formar un programa visual. Cada **nodo** realiza una operación: en ocasiones, es algo tan sencillo como almacenar un número, o puede ser una acción más compleja, como crear o consultar geometría.

Anatomía de un nodo

La mayoría de los nodos de Dynamo están compuestos por cinco partes. Aunque existen excepciones, como los nodos de entrada, la anatomía de cada nodo se puede describir como se indica a continuación:

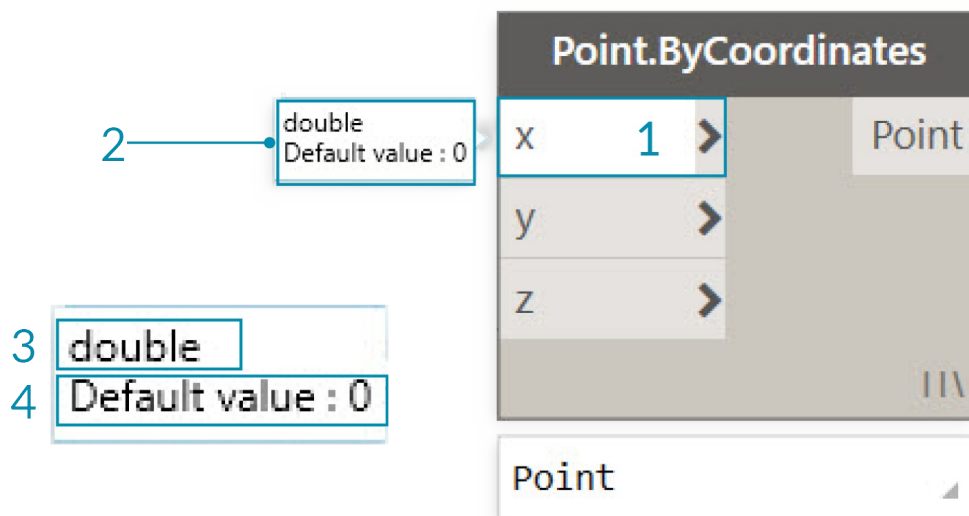


1. Nombre: nombre del nodo con la convención de nomenclatura Categoría.Nombre.
2. Parte principal: el cuerpo principal del nodo. Al hacer clic con el botón derecho, se presentan opciones en el nivel de todo el nodo.
3. Puertos (entrada y salida): los receptores de los cables que proporcionan los datos de entrada al nodo, así como los resultados de la acción del nodo.
4. Icono de encaje: indica la opción de encaje especificada para las entradas de lista coincidentes (se explicará detalladamente más adelante).
5. Valor por defecto (haga clic con el botón derecho en un puerto de entrada): algunos nodos tienen valores por defecto que se pueden utilizar o no.

Puertos

Las entradas y las salidas de los nodos se denominan puertos y actúan como receptores de cables. Los datos entran por la izquierda en el nodo a través de los puertos y salen del nodo por la derecha después de que se haya ejecutado su operación. Los puertos esperan recibir datos de un tipo determinado. Por ejemplo, al conectar un número como 2.75 a los puertos de un nodo Point.ByCoordinates, se creará correctamente un punto. Sin embargo, si se proporciona Red al mismo puerto, se producirá un error.

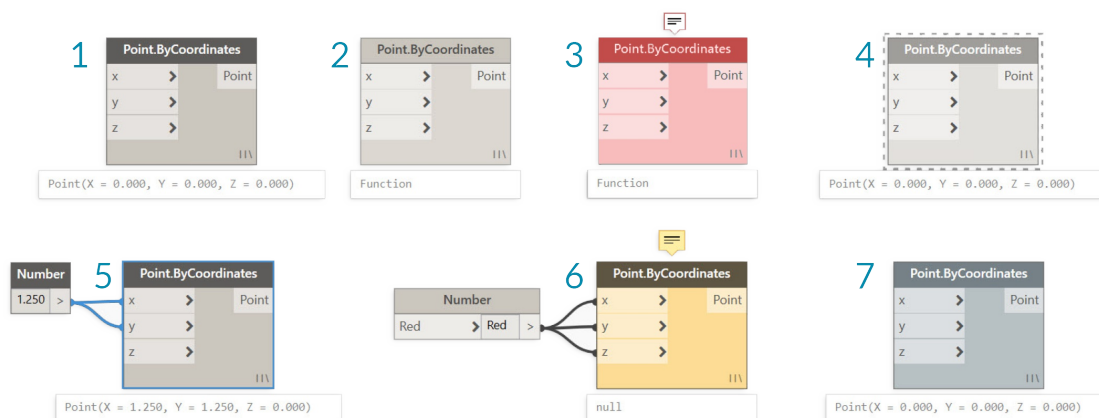
Sugerencia: coloque el cursor sobre un puerto para ver la información de herramientas que contiene el tipo de datos esperado.



1. Etiqueta de puerto
2. Información de herramientas
3. Tipo de datos
4. Valor por defecto

Estados

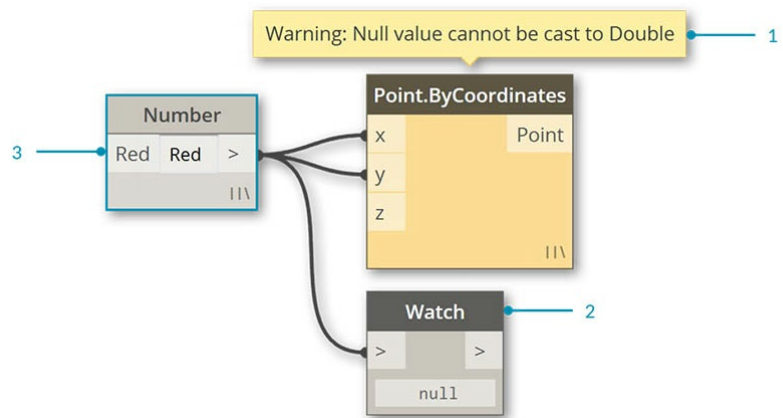
Dynamo proporciona una indicación del estado de ejecución del programa visual mediante la renderización de los nodos con diferentes esquemas de color en función del estado de cada nodo. Además, al colocar el cursor sobre el nombre o los puertos, o hacer clic con el botón derecho en ellos, se ofrece información y opciones adicionales.



1. Activo: los nodos con un fondo de nombre gris oscuro están conectados correctamente y todas sus entradas se han conectado de forma adecuada.
2. Inactivo: los nodos grises están inactivos y deben estar conectados con cables para que formen parte del flujo del programa en el espacio de trabajo activo.
3. Estado de error: el color rojo indica que el nodo presenta un estado de error.
4. Bloquear: un nodo transparente tiene la opción Bloquear activada, por lo que se suspende la ejecución del nodo.
5. Seleccionados: actualmente los nodos seleccionados se resaltan con un borde de color turquesa.
6. Advertencia: los nodos de color amarillo presentan un estado de advertencia, lo que significa que pueden tener tipos de datos incorrectos.
7. Vista preliminar de fondo: el color gris oscuro indica que la vista preliminar de la geometría está desactivada.

Si el programa visual contiene advertencias o errores, Dynamo proporcionará información adicional sobre el problema. Todos los nodos en amarillo también presentarán una información de herramientas sobre el nombre. Coloque el cursor sobre la información de herramientas para expandirla.

Sugerencia: con esta información de herramientas a mano, examine los nodos ascendentes para ver si el tipo de datos o la estructura de datos necesarios presentan errores.



1. Información de herramientas de advertencia: el valor "null" (nulo) o la falta de datos no se pueden considerar como doble, por ejemplo, un número.
2. Utilice el nodo Watch para examinar los datos de entrada.
3. El nodo Number ascendente almacena "Red", no un número.

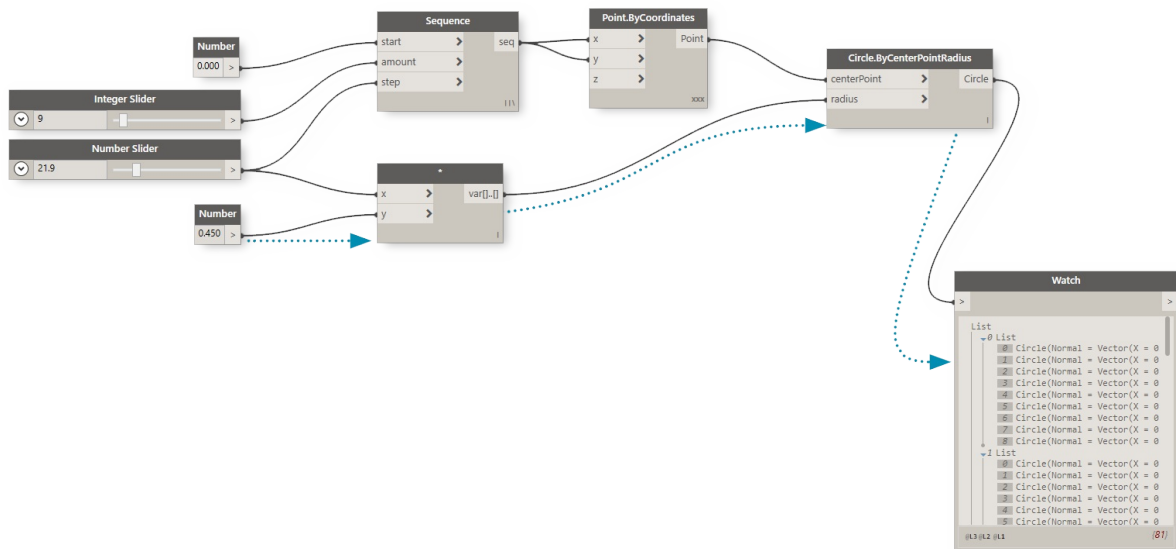
Cables

Cables

Los cables se conectan entre nodos para crear relaciones y establecer el flujo de nuestro programa visual. Podemos considerarlos literalmente como cables eléctricos que transportan pulsos de datos de un objeto al siguiente.

Flujo del programa

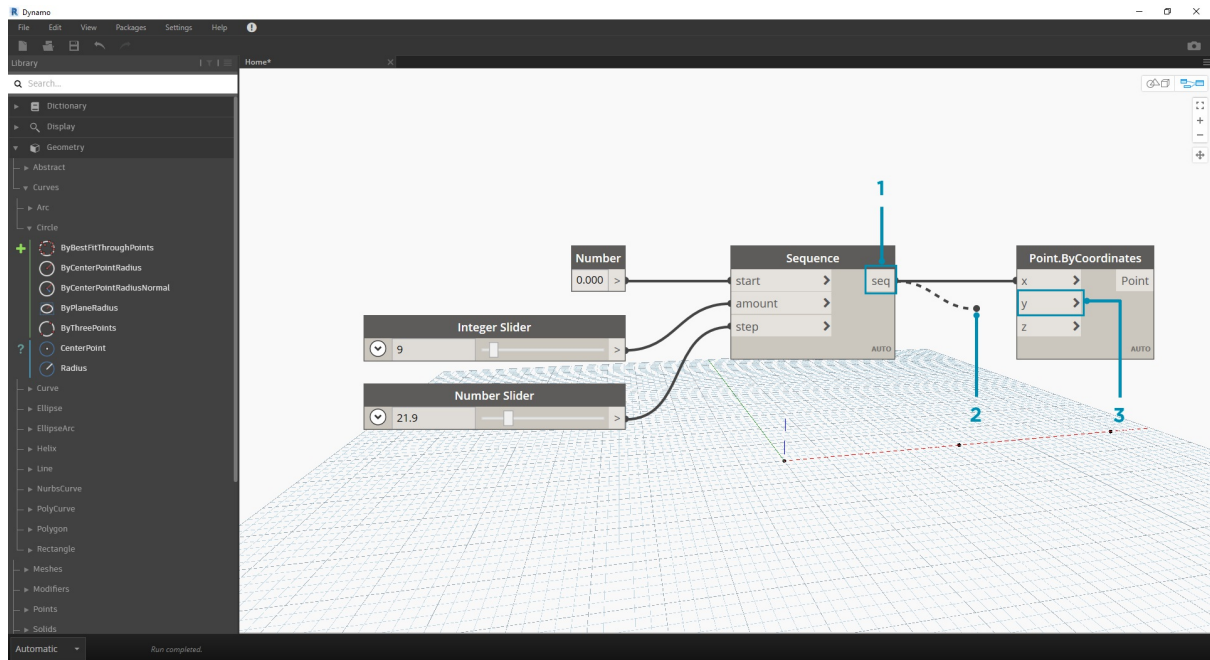
Los cables conectan el puerto de salida de un nodo al puerto de entrada de otro nodo. Esta dirección establece el **flujo de datos** en el programa visual. Aunque podemos organizar los nodos como deseemos en el espacio de trabajo, como los puertos de salida se encuentran a la derecha de los nodos y los puertos de entrada a la izquierda, por lo general, podemos decir que el flujo del programa se desplaza de izquierda a derecha.



Creación de cables

Para crear un cable, se debe hacer clic con el botón izquierdo del ratón en un puerto y, a continuación, hacer clic con el botón izquierdo en el puerto de otro nodo para crear una conexión. Durante el proceso de creación de una conexión, el cable se mostrará discontinuo y se ajustará para convertirse en líneas continuas cuando se establezca correctamente la conexión. Los datos siempre fluirán a través de este cable desde la salida hasta la entrada. No obstante, podemos crear el cable en cualquier dirección en cuanto a la secuencia en la que se hace clic en los puertos conectados.

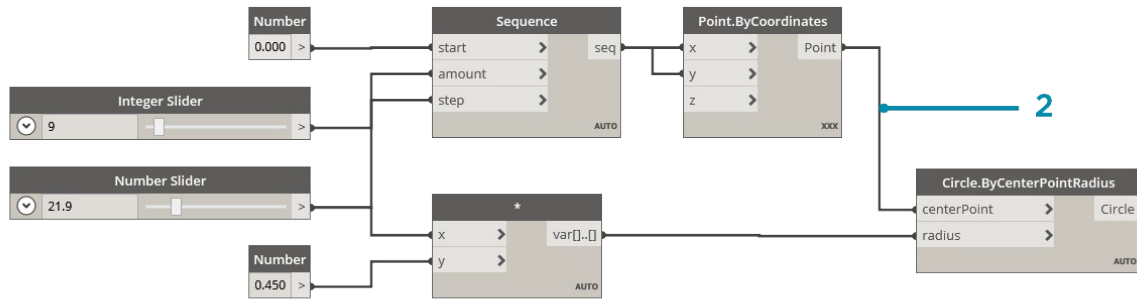
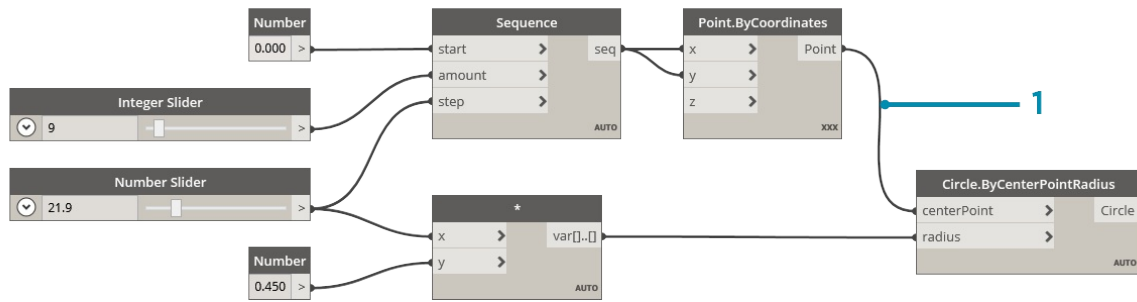
Sugerencia: antes de completar la conexión con el segundo clic, permita que se ajuste el cable a un puerto y coloque el cursor en esa ubicación para ver la información de herramientas del puerto.



1. Haga clic en el puerto de salida **seq** del nodo Number Sequence.
2. Al desplazar el cursor a otro puerto, el cable aparece como una línea discontinua.
3. Haga clic en el puerto de entrada **y** de Point.ByCoordinates para completar la conexión.

Edición de cables

Con frecuencia, desearíamos ajustar el flujo de programa de nuestro programa visual mediante la edición de las conexiones representadas por los cables. Para editar un cable, haga clic con el botón izquierdo en el puerto de entrada del nodo que ya está conectado. Ahora tiene dos opciones:



1. Tipo de conector: curvas
2. Tipo de conector: polilíneas

Biblioteca

Biblioteca de Dynamo

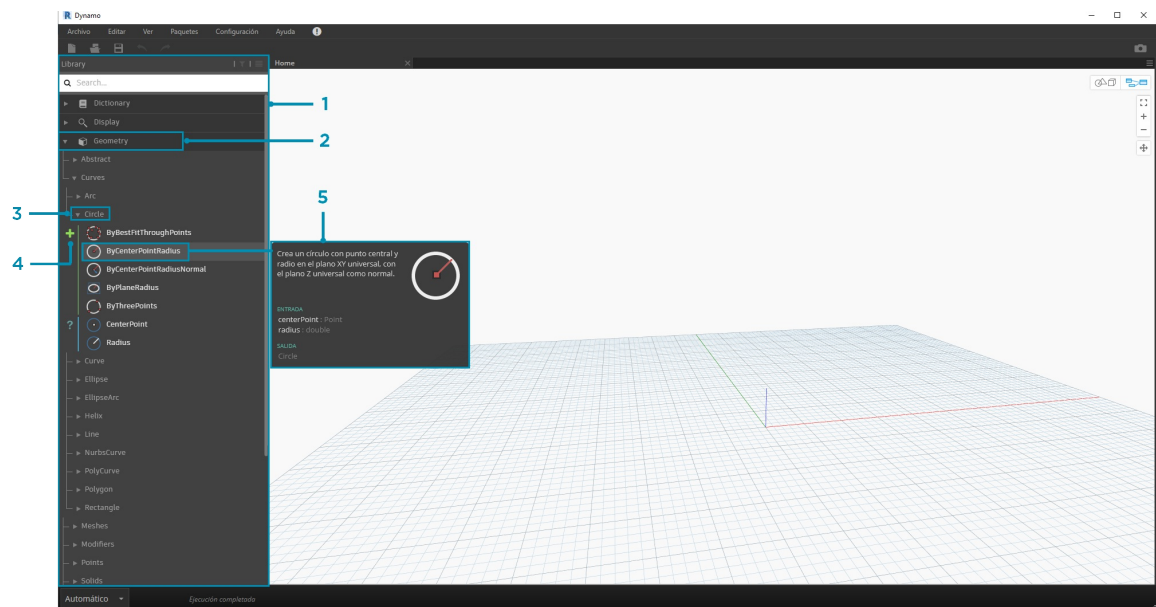
La **biblioteca de Dynamo** contiene los nodos que se añaden al espacio de trabajo para definir programas visuales para su ejecución. En la biblioteca, se pueden buscar nodos o desplazarse a ellos. Los nodos que se incluyen aquí, los nodos básicos instalados, los nodos personalizados que se definen y los nodos de Package Manager que se añaden a Dynamo, se organizan jerárquicamente por categoría. Revisemos esta organización y exploremos los nodos clave que utilizaremos con frecuencia.

Biblioteca de bibliotecas

La **biblioteca** de Dynamo con la que interactuamos en la aplicación es en realidad un conjunto de bibliotecas funcionales, cada una de las cuales contiene nodos agrupados por categoría. Aunque esto puede parecer poco acertado al principio, es un marco flexible para organizar los nodos que se incluyen con la instalación por defecto de Dynamo y mejorará cuando empecemos a ampliar esta funcionalidad base con nodos personalizados y paquetes adicionales.

El esquema de organización

La sección **Biblioteca** de la interfaz de usuario de Dynamo se compone de bibliotecas organizadas jerárquicamente. A medida que profundizamos en la biblioteca, estamos examinando secuencialmente una biblioteca, las categorías de la biblioteca y las subcategorías de la categoría para encontrar el nodo.

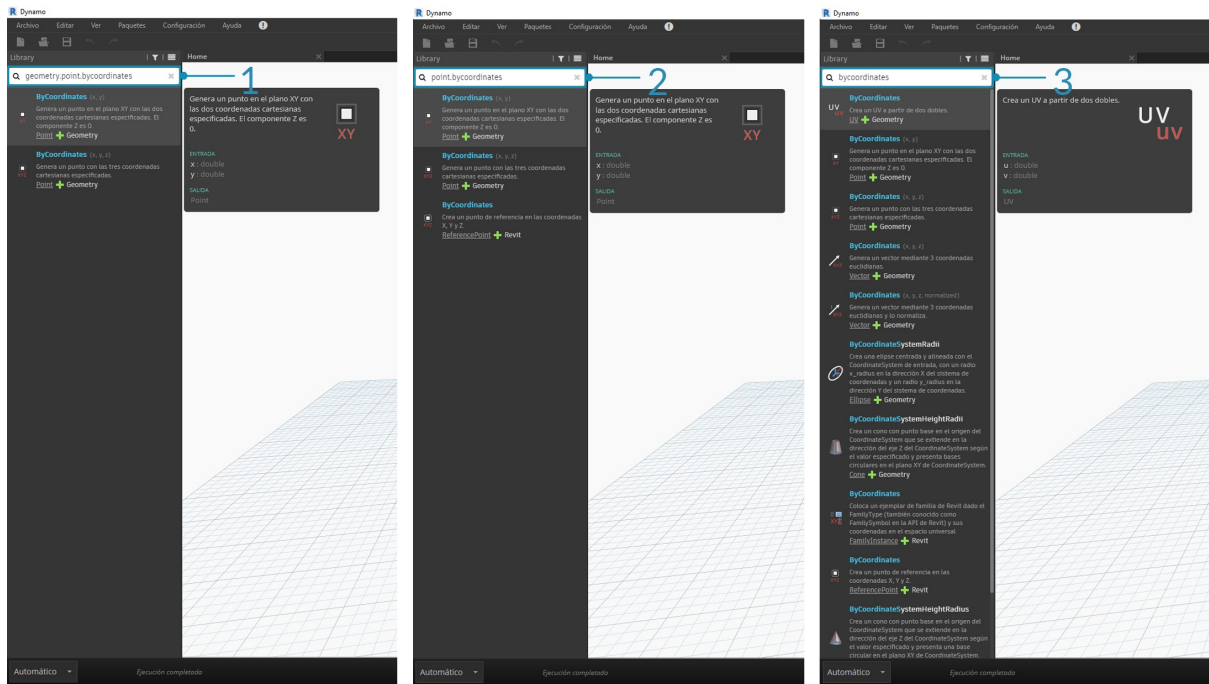


1. La sección Biblioteca: la región de la interfaz de Dynamo.
2. Una biblioteca: una colección de categorías relacionadas como, por ejemplo, **Geometría**.
3. Una categoría: una colección de nodos relacionados como, por ejemplo, todos los relacionados con **círculos**.
4. Subcategoría: desglose de los nodos de la categoría, normalmente, por **Crear**, **Acción** o **Consultar**.
5. Un nodo: los objetos que se añaden al espacio de trabajo para realizar una acción.

Convenciones de nomenclatura

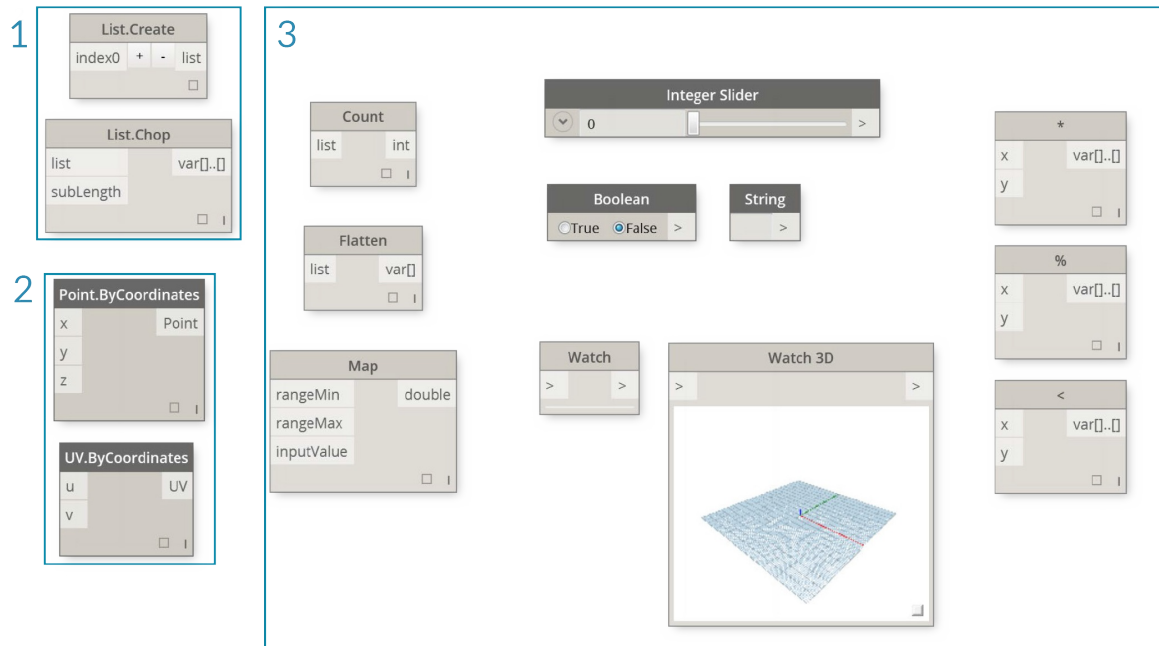
La jerarquía de cada biblioteca aparece reflejada en el nombre de los nodos añadidos al espacio de trabajo, que también podemos utilizar en el campo de búsqueda o con los bloques de código (que utilizan el *lenguaje de textura de Dynamo*). Además de utilizar palabras clave para intentar encontrar nodos, podemos escribir la jerarquía separada por un punto.

Al escribir distintas partes de la ubicación del nodo en la jerarquía de biblioteca con el formato `biblioteca.categoría.nombreNodo`, se obtienen resultados diferentes:



1. biblioteca.categoría.nombreNodo
2. categoría.nombreNodo
3. nombreNodo o palabra clave

Por lo general, el nombre del nodo en el espacio de trabajo se renderizará con el formato `categoría.nombreNodo`, con algunas excepciones destacadas, sobre todo, en las categorías de entrada y vista. Tenga en cuenta la diferencia de categoría en nodos con nombres similares:



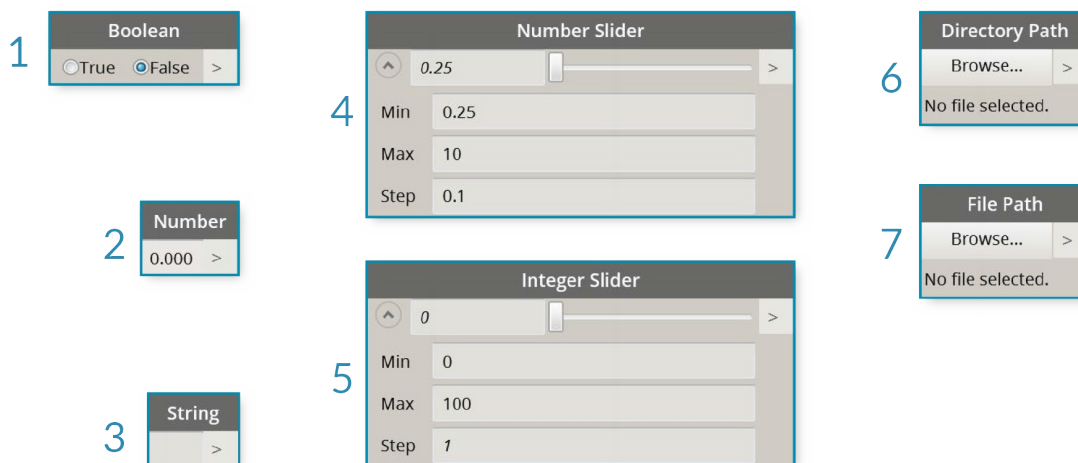
1. Los nodos de la mayoría de las bibliotecas incluirán el formato de categoría.
2. `Point.ByCoordinates` y `UV.ByCoordinates` tienen el mismo nombre, pero proceden de diferentes categorías.
3. Entre las excepciones destacables, se incluyen las funciones integradas, `Core.Input`, `Core.View` y los operadores.

Nodos utilizados frecuentemente

Con cientos de nodos incluidos en la instalación básica de Dynamo, ¿cuáles son fundamentales para desarrollar nuestros programas visuales? Veamos los parámetros de nuestro programa (**entrada**), consultemos los resultados de la acción de un nodo (**visualización**) y definamos entradas o funciones mediante un acceso directo (**bloque de código**).

Entrada

Los nodos de entrada son el principal medio para que el usuario de nuestro programa visual (ya sea usted mismo u otro usuario) interactúe con los parámetros clave. A continuación, se indican los nodos disponibles en la categoría de entrada de la biblioteca principal:

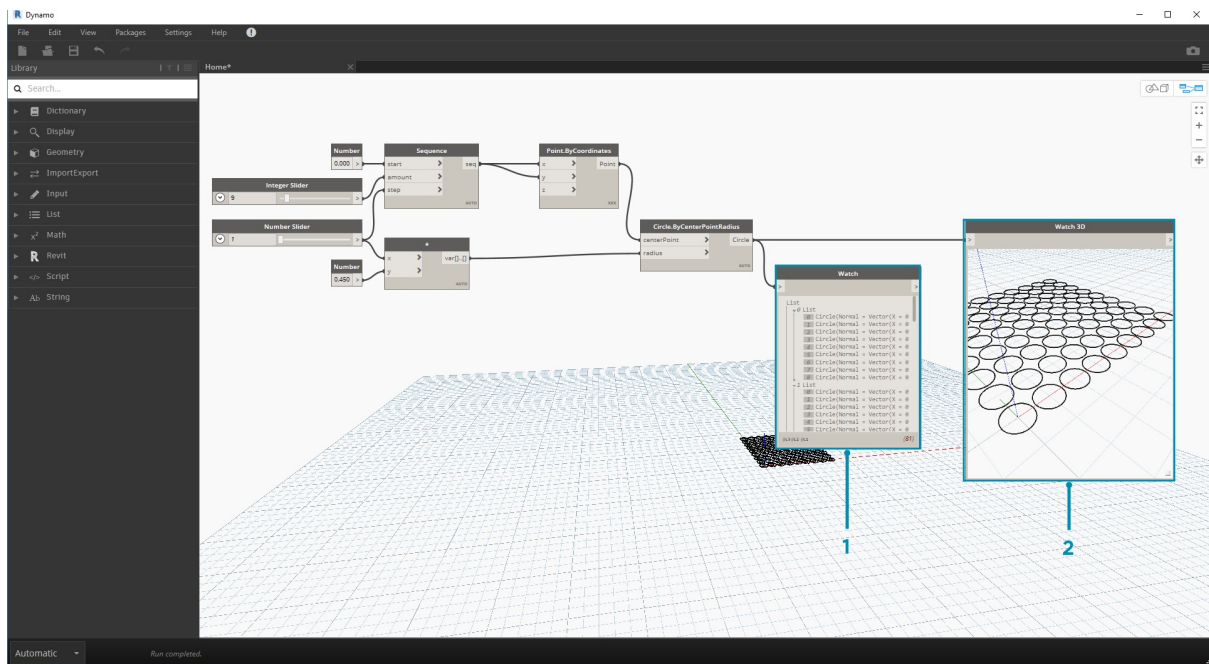


1. Booleano
2. Número
3. Cadena
4. Control deslizante de número
5. Control deslizante de entero
6. Ruta de directorio
7. Ruta de archivo

Visualización

Los nodos de visualización, Watch, son esenciales para administrar los datos que se ejecutan a través del programa visual. Aunque puede ver el resultado de un nodo a través de la vista preliminar de datos del nodo, es posible que desee que se mantenga su visualización en un nodo **Watch** o que desee ver los resultados de la geometría a través de un nodo **Watch3D**. Ambos se encuentran en la categoría de vista de la biblioteca principal.

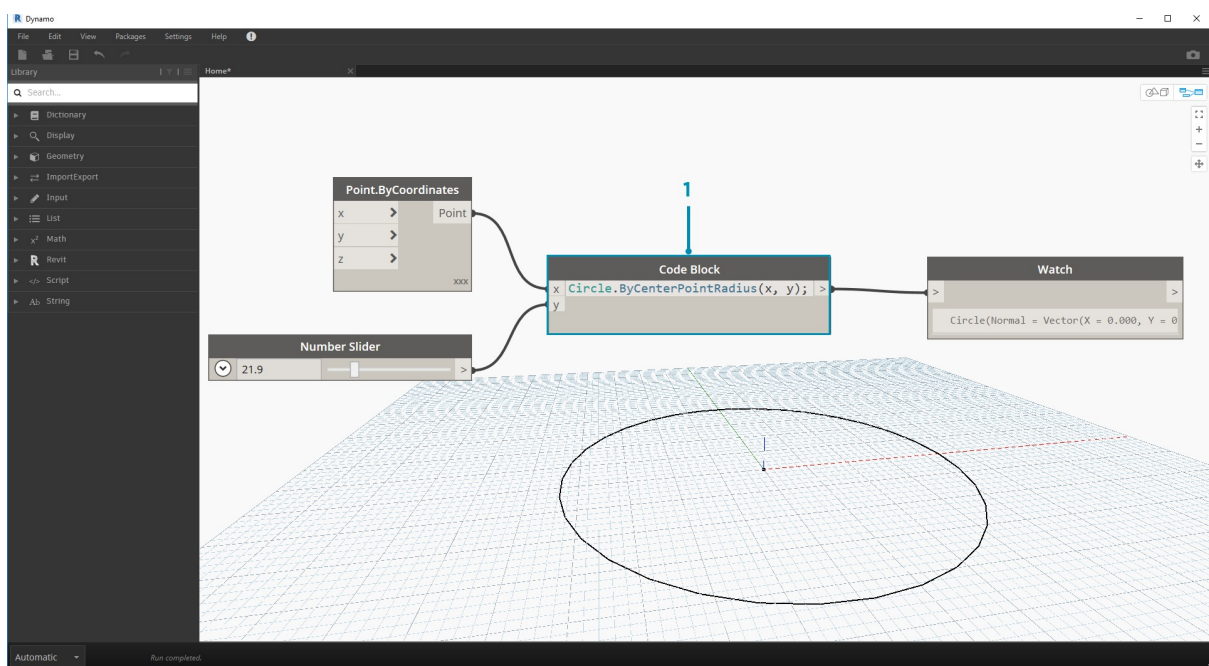
Sugerencia: en ocasiones, la vista preliminar 3D puede ser molesta cuando el programa visual contiene muchos nodos. Considere la posibilidad de desactivar la opción Mostrando vista preliminar 3D en segundo plano en el menú Configuración y utilizar un nodo Watch3D para obtener una vista preliminar de la geometría.



1. Watch: tenga en cuenta que cuando se selecciona un elemento en el nodo Watch, este se etiquetará en las vistas preliminares 3D y Watch3D.
2. Watch3D: arrastre el pinzamiento inferior derecho para cambiar el tamaño y desplácese con el ratón de la misma forma que lo haría en la vista preliminar 3D.

Bloque de código

Los nodos de bloque de código, **Code Block**, se pueden utilizar para definir un bloque de código con líneas separadas por signos de punto y coma. Esto puede ser tan sencillo como X/Y. También podemos utilizar bloques de código como un acceso directo para definir una entrada de número o una llamada a la función de otro nodo. La sintaxis necesaria para ello sigue la convención de nomenclatura del lenguaje textual de Dynamo, DesignScript y se describe en la sección 7.2. Vamos a crear un círculo con este acceso directo:



1. Haga doble clic para crear un nodo de **bloque de código**.
2. Escriba `Circle.ByCenterPointRadius(x, y);`.
3. Si hace clic en el espacio de trabajo para borrar la selección, se deben añadir las entradas x e y automáticamente

4. Cree un nodo **Point.ByCoordinates** y un componente **Number Slider**; a continuación, conecte las entradas al bloque de código.
5. La ejecución del programa visual debe dar como resultado un círculo en la vista preliminar 3D.

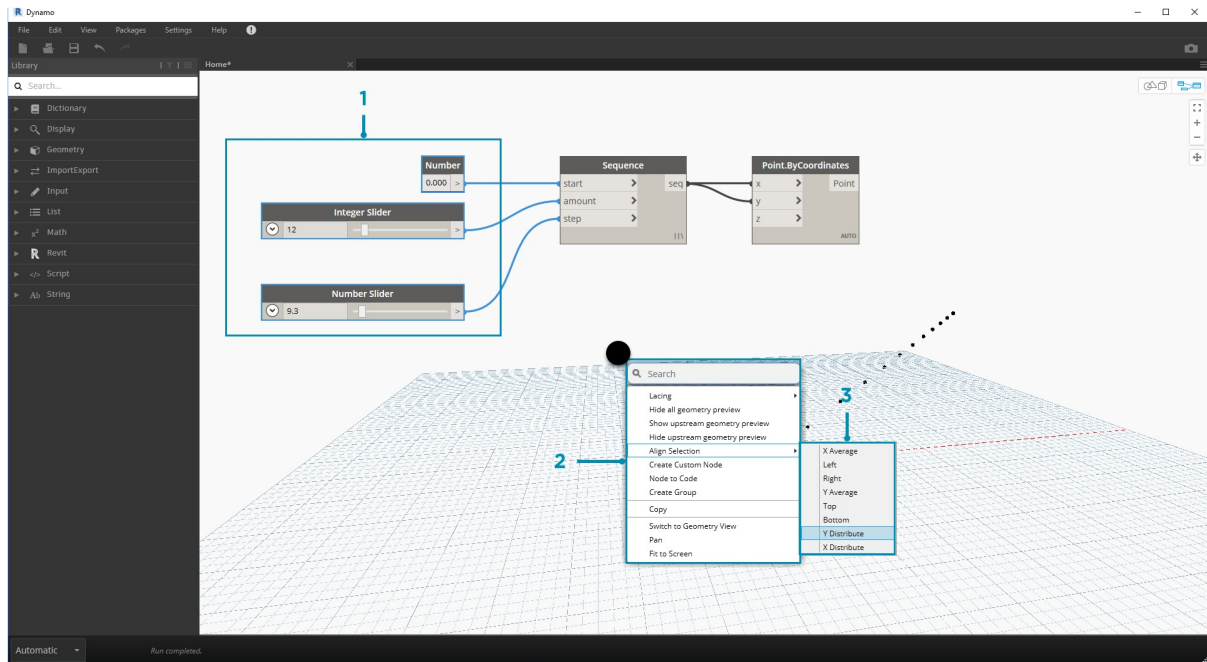
Administración del programa

Administración del programa

Trabajar en un proceso de programación visual puede ser una potente actividad creativa, pero en muy poco tiempo el flujo de programa y las entradas clave de usuario pueden verse eclipsados por la complejidad o la presentación del espacio de trabajo. Veamos algunos procedimientos recomendados para administrar el programa.

Alineación

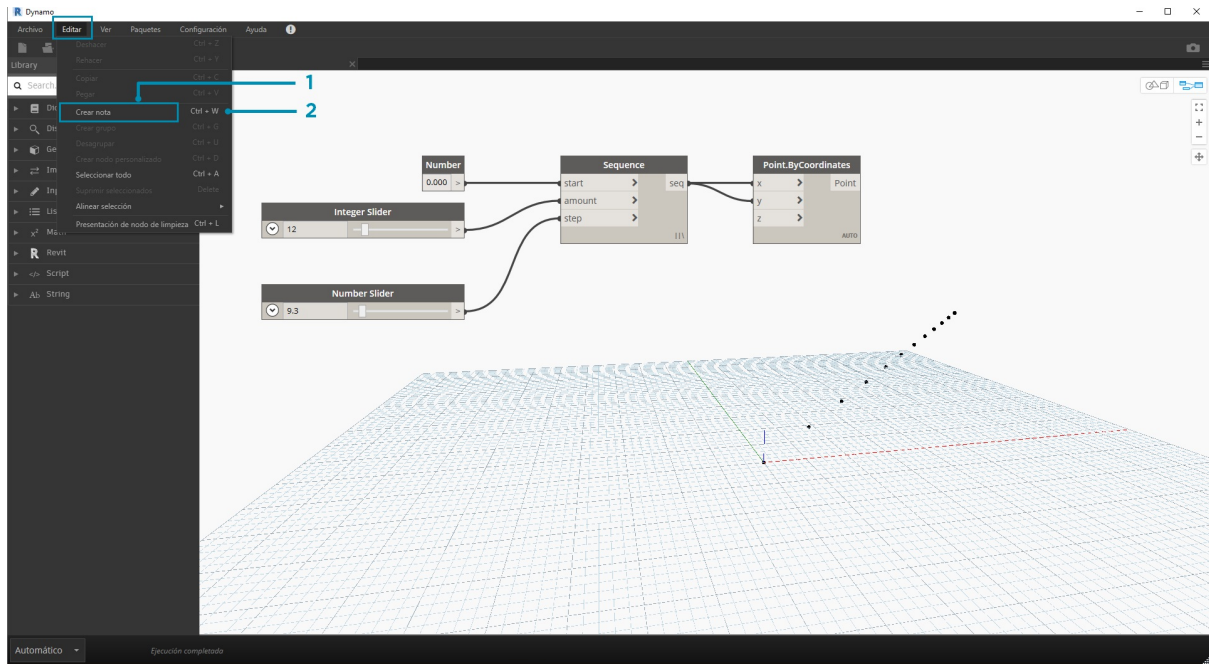
Una vez que hemos añadido varios nodos al espacio de trabajo, es posible que deseemos reorganizar el diseño de los nodos para obtener mayor claridad. Al seleccionar más de un nodo y hacer clic con el botón derecho en el espacio de trabajo, la ventana emergente incluye un menú **Alinear selección** con opciones de justificación y distribución en X e Y.



1. Seleccione varios nodos.
2. Haga clic con el botón derecho en el espacio de trabajo.
3. Utilice las opciones de **Alinear selección**.

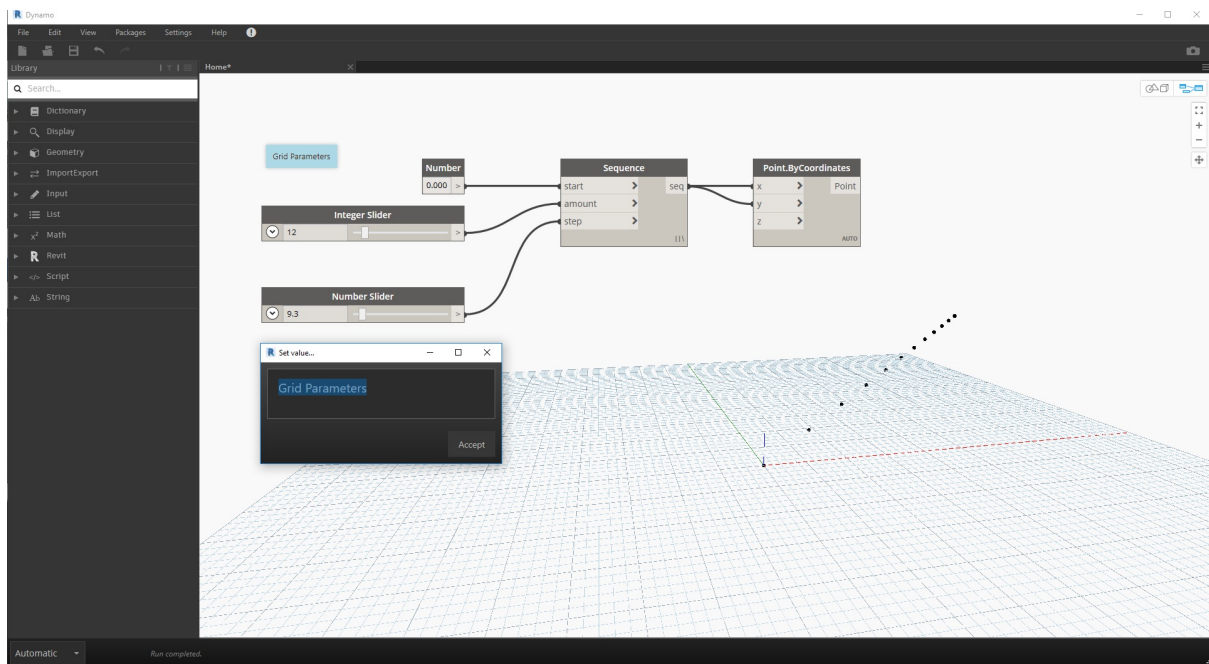
Notas

Con cierta experiencia, es posible que podamos "leer" el programa visual revisando los nombres de nodo y siguiendo el flujo de programa. Para los usuarios de todos los niveles de experiencia, es recomendable también incluir etiquetas y descripciones en lenguaje sencillo. Para ello, Dynamo presenta un nodo **Notes** con un campo de texto editable. Podemos añadir notas al espacio de trabajo de dos formas:



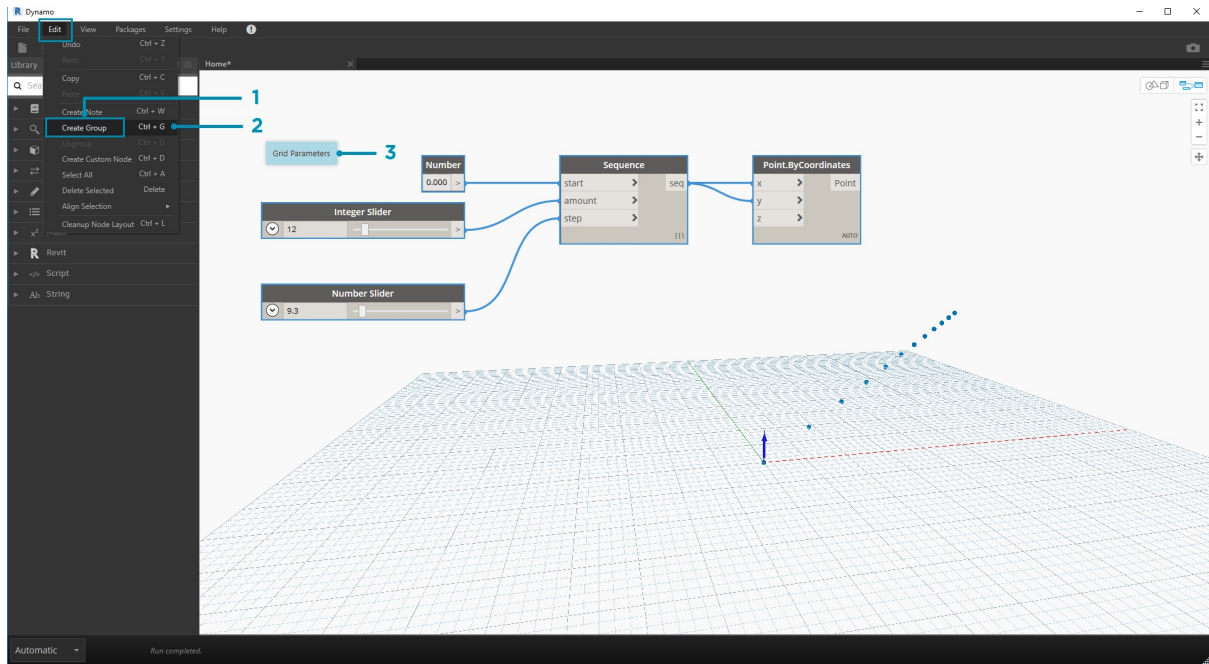
1. Vaya al menú Editar > Crear nota.
2. Utilice el método abreviado de teclado Ctrl + W.

Una vez que se ha añadido la nota al espacio de trabajo, aparece un campo de texto que nos permite editar el texto de la nota. Una vez creada, se puede editar la nota. Para ello, haga doble clic o haga clic con el botón derecho en el nodo Note.



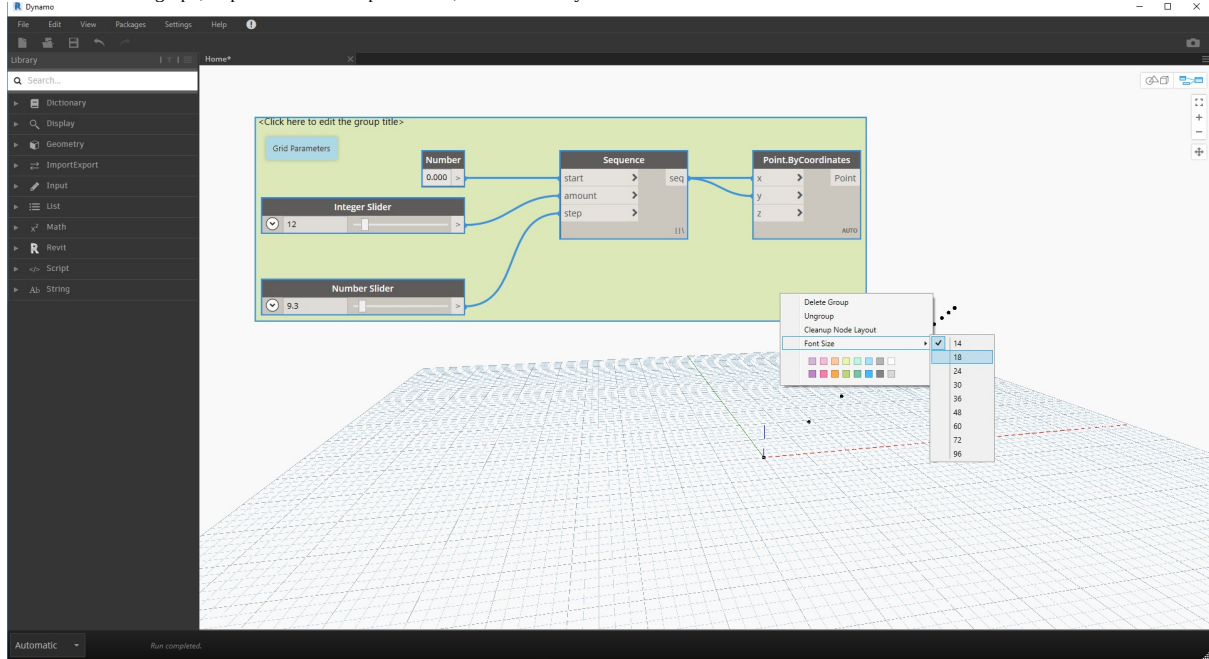
Agrupación

Cuando aumente el tamaño del programa visual, resultará útil identificar los pasos más grandes que se ejecutarán. Podemos resaltar conjuntos más grandes de nodos con un **grupo** para etiquetarlos con un rectángulo de color en el fondo y un título. Hay tres formas de crear un grupo con más de un nodo seleccionado:



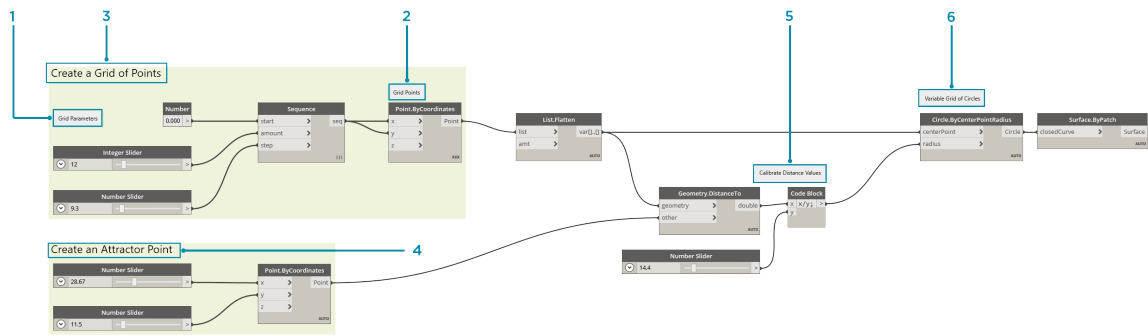
1. Vaya al menú Editar > Crear grupo.
2. Utilice el acceso directo de teclado Ctrl + G.
3. Haga clic con el botón derecho en el espacio de trabajo y seleccione Crear grupo.

Una vez creado un grupo, se pueden editar sus parámetros, como el título y el color.



Sugerencia: el uso de las notas y los grupos es una forma eficaz de anotar el archivo y aumentar la legibilidad.

Este es nuestro programa de la sección 2.4 con notas y grupos añadidos:



1. Nota: "Grid Parameters" (Parámetros de rejilla)
2. Nota: "Grid Points" (Puntos de rejilla)
3. Grupo: "Create a Grid of Points" (Crear una rejilla de puntos)
4. Grupo: "Create an Attractor Point" (Crear un punto de atractor)
5. Nota: "Calibrate Distance Values" (Calibrar valores de distancia)
6. Nota: "Variable Grid of Circles" (Rejilla variable de círculos)

Los bloques de construcción de los programas

LOS BLOQUES DE CONSTRUCCIÓN DE LOS PROGRAMAS

Una vez que estemos listos para profundizar en el desarrollo de programas visuales, necesitaremos una mejor comprensión de los componentes de construcción que utilizaremos. En este capítulo, se presentan conceptos fundamentales sobre los datos: el contenido que recorre los cables de nuestro programa de Dynamo.



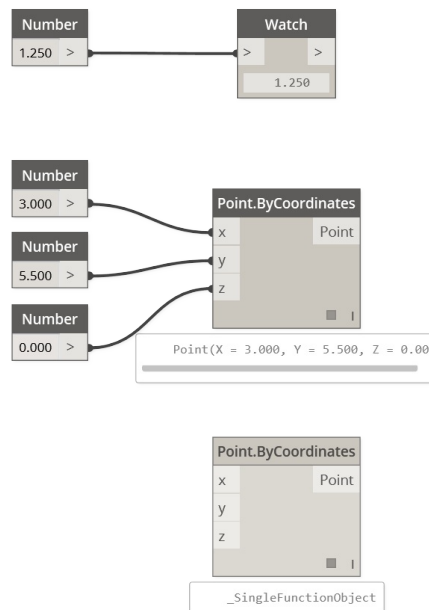
Datos

Datos

Los datos son el contenido de nuestros programas. Viajan a través de cables proporcionando entradas a los nodos, donde se procesan y se transforman en una nueva forma de datos de salida. Revisemos la definición de los datos, cómo se estructuran y cómo utilizarlos en Dynamo.

¿Qué son los datos?

Los datos son un conjunto de valores de variables cualitativas o cuantitativas. La forma de datos más sencilla son números como 0, 3, 14 o 17. Sin embargo, los datos también pueden ser de diferentes tipos: una variable que representa números cambiantes (**altura**); caracteres (**miNombre**); geometría (**Círculo**); o una lista de elementos de datos (1, 2, 3, 5, 8, 13, etc.). Necesitamos datos que poder añadir a los puertos de entrada de los nodos de Dynamo; podemos tener datos sin acciones, pero necesitamos datos para procesar las acciones que representan nuestros nodos. Cuando se añade un nodo al espacio de trabajo, si no se le proporciona ninguna entrada, el resultado será una función y no el resultado de la acción en sí.



1. Datos simples.
2. Datos y una acción (un nodo); se ejecuta correctamente.
3. Una acción (un nodo) sin entradas de datos; devuelve una función genérica.

Cuidado con el tipo "null" (nulo)

El tipo "null" representa la ausencia de datos. Aunque se trata de un concepto abstracto, es probable que esto ocurra mientras trabaja con programación visual. Si una acción no crea un resultado válido, el nodo devolverá un valor "null". Comprobar si existen valores "null" y eliminarlos de la estructura de datos es una parte crucial para crear programas sólidos.

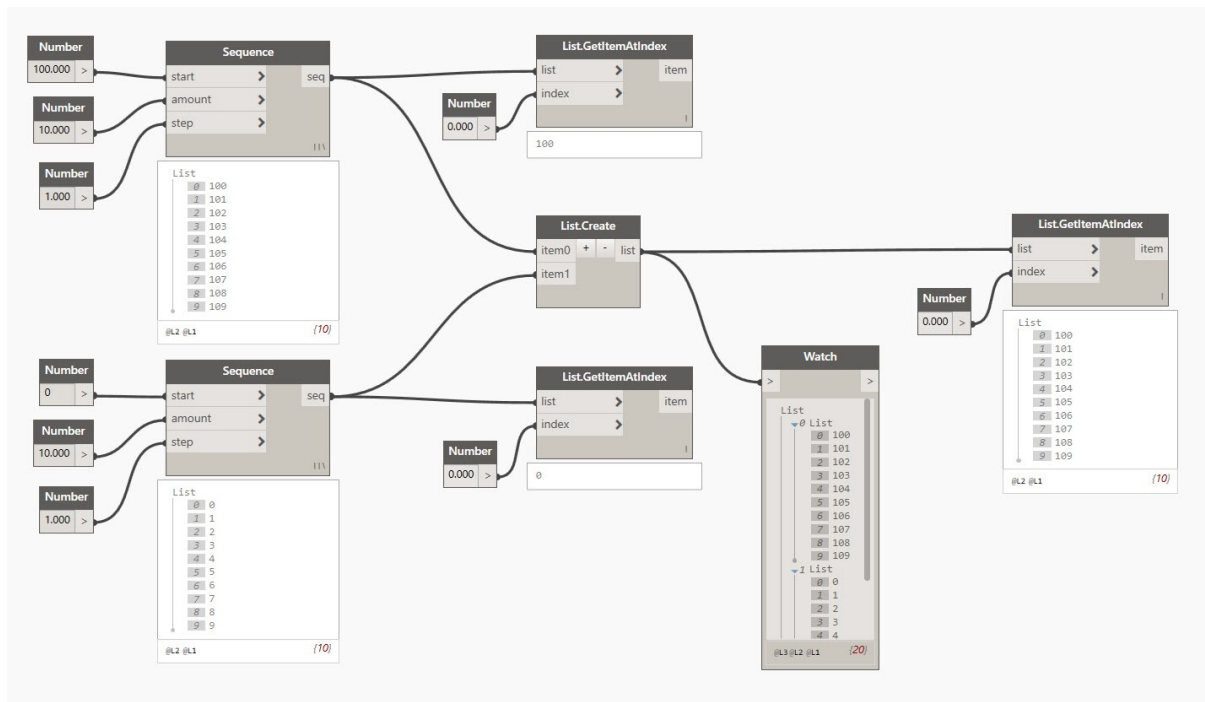
Icono	Nombre/sintaxis	Entradas	Salidas
	Object.IsNull	obj	bool

Estructuras de datos

Cuando somos programadores visuales, podemos generar una gran cantidad de datos rápidamente y requerir un medio de administración de su jerarquía. Esta es la función de las estructuras de datos, los esquemas organizativos en los que almacenamos los datos. Las características específicas de las estructuras de datos y su uso varían de un lenguaje de programación a otro. En Dynamo, se añade jerarquía a los datos a través de las listas. Exploraremos esto en profundidad en capítulos posteriores, pero empecemos con algo sencillo.

Una lista representa una colección de elementos colocados en una estructura de datos:

- Tengo cinco dedos (*elementos*) en la mano (*lista*).
- Hay diez casas (*elementos*) en mi calle (*lista*).



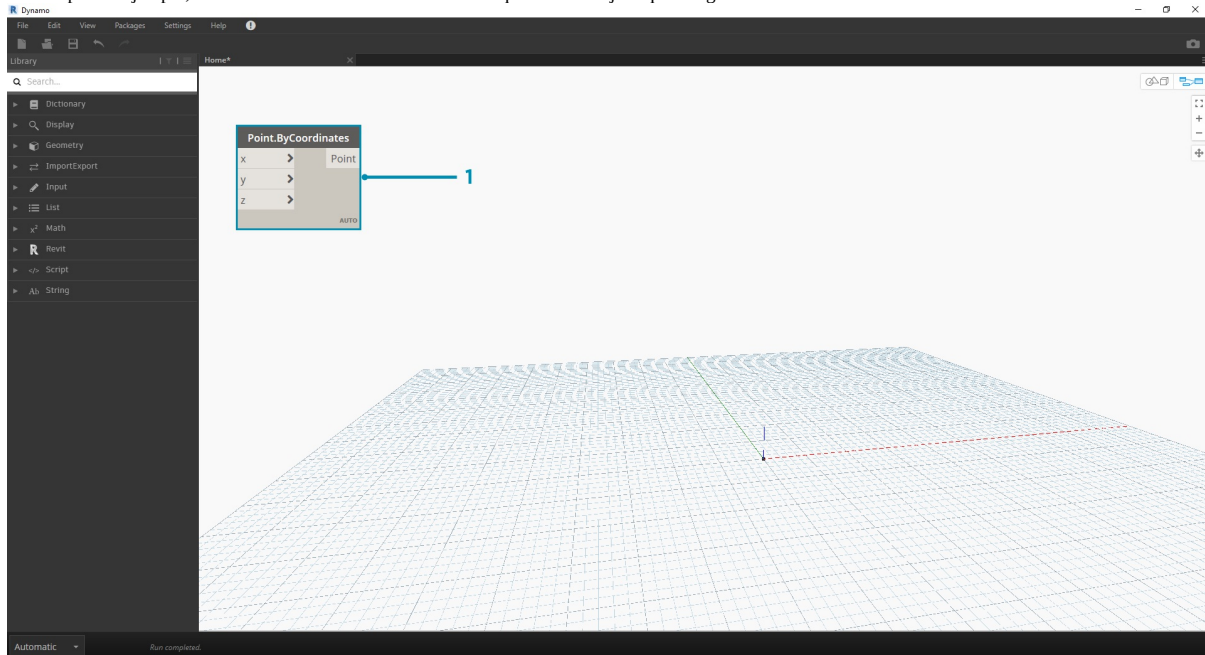
1. Un nodo **Number Sequence** define una lista de números utilizando entradas *start*, *amount* y *step*. Con estos nodos, hemos creado dos listas independientes de diez números, una que abarca de 100 a 109 y otra de 0 a 9.
2. El nodo **List.GetItemAtIndex** selecciona un elemento de una lista en un índice determinado. Al seleccionar 0, se obtiene el primer elemento de la lista (100 en este caso).
3. Al aplicar el mismo proceso a la segunda lista, obtenemos un valor de 0, el primer elemento de la lista.
4. Ahora combinamos las dos listas en una mediante el nodo **List.Create**. Observe que el nodo crea una *lista de listas*. Esto cambia la estructura de los datos.
5. Al volver a utilizar **List.GetItemAtIndex**, con el índice establecido en 0, se obtiene la primera lista de la lista de listas. Esto es lo que significa tratar una lista como un elemento, lo cual es ligeramente diferente de otros lenguajes de secuencias de comandos. En capítulos posteriores, nos adentraremos más en la manipulación de listas y la estructura de datos.

El concepto clave para comprender la jerarquía de datos en Dynamo es que **en relación con la estructura de datos, las listas se consideran elementos**. En otras palabras, Dynamo funciona con un proceso descendente para comprender las estructuras de datos. ¿Qué quiere decir esto? Veámoslo con un ejemplo.

Uso de datos para crear una cadena de cilindros

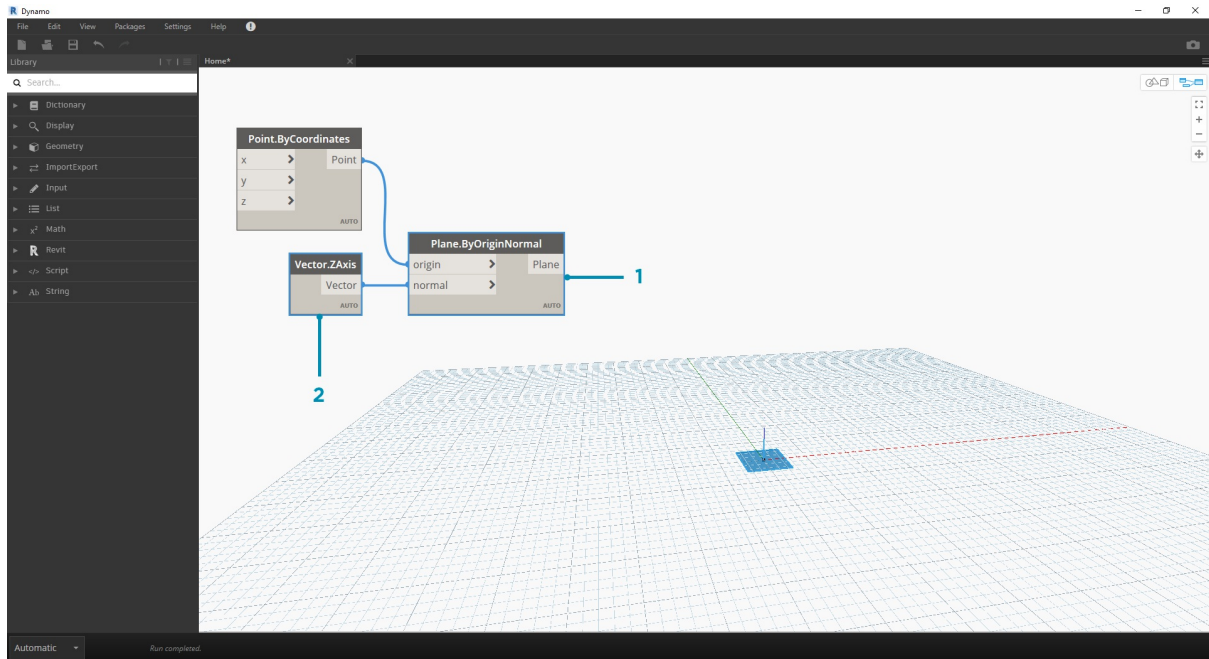
Descargue el archivo de ejemplo que acompaña a este ejercicio (haga clic con el botón derecho y seleccione "Guardar enlace como..."): [Building Blocks of Programs - Data.dyn](#). En el Apéndice se incluye una lista completa de los archivos de ejemplo.

En este primer ejemplo, vamos a ensamblar un cilindro vaciado que recorre la jerarquía de geometría tratada en esta sección.

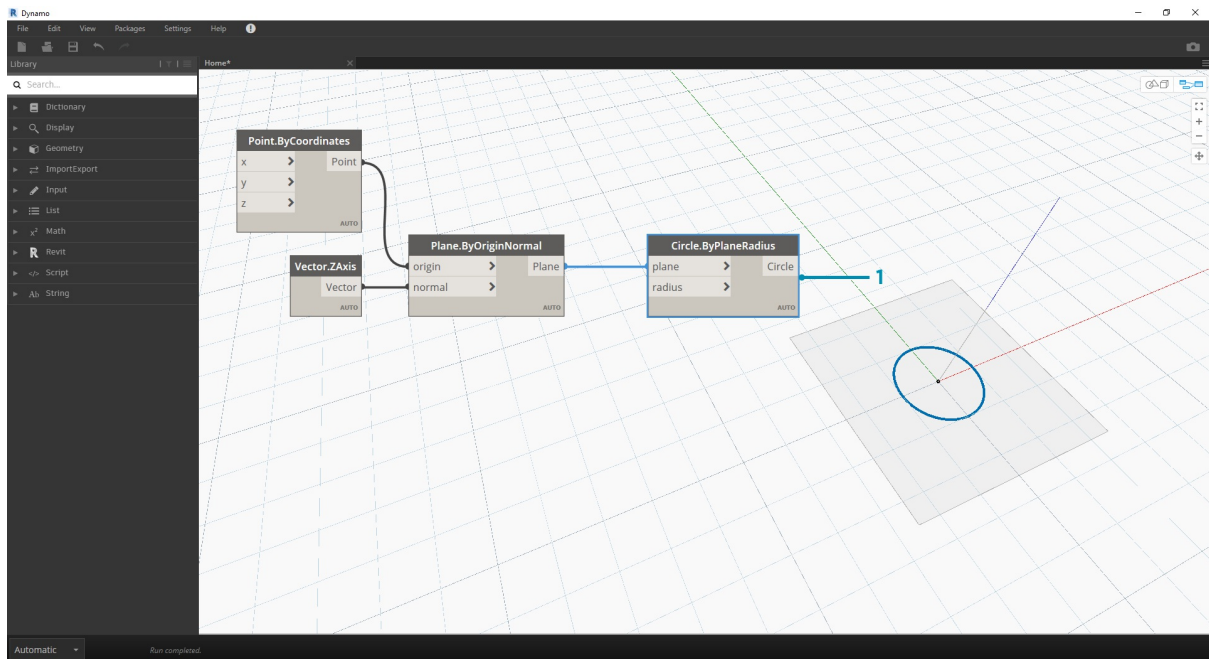


1. **Point.ByCoordinates**: después de añadir el nodo al lienzo, se muestra un punto en el origen de la rejilla de vista preliminar de Dynamo. Los

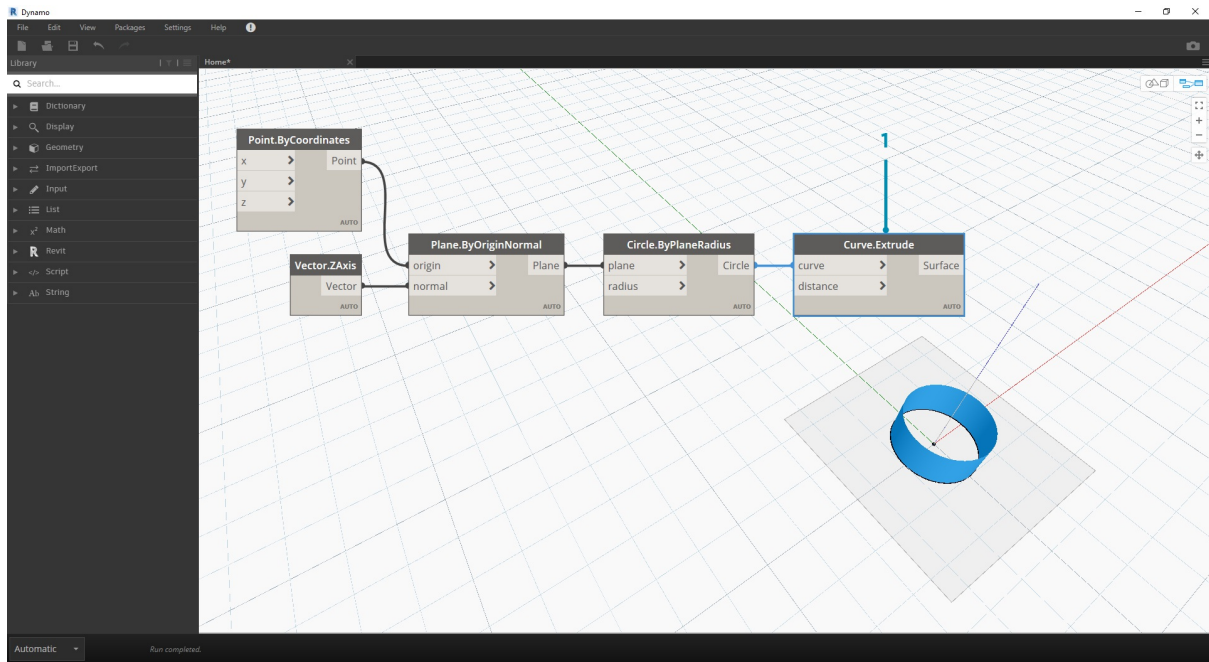
valores por defecto de las entradas x , y y z son $0,0,0$, lo que nos da un punto en esta ubicación.



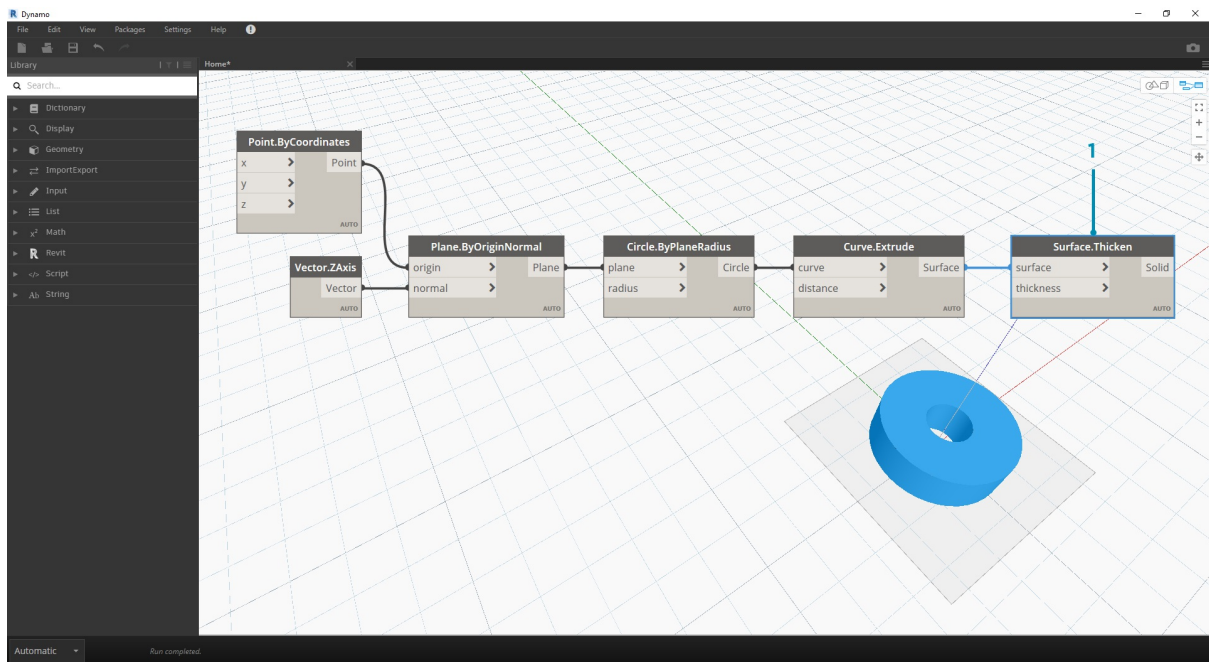
1. **Plane.ByOriginNormal:** el siguiente paso en la jerarquía de geometría es un plano. Existen varias formas de construir un plano; utilizaremos un origen y una normal para la entrada. El origen es el nodo de punto creado en el paso anterior.
2. **Vector.ZAxis:** es un vector unificado en la dirección Z. Observe que no hay entradas, solo un vector con un valor $[0,0,1]$. Se utiliza como la entrada *normal* para el nodo *Plane.ByOriginNormal*. Esto nos proporciona un plano rectangular en la vista preliminar de Dynamo.



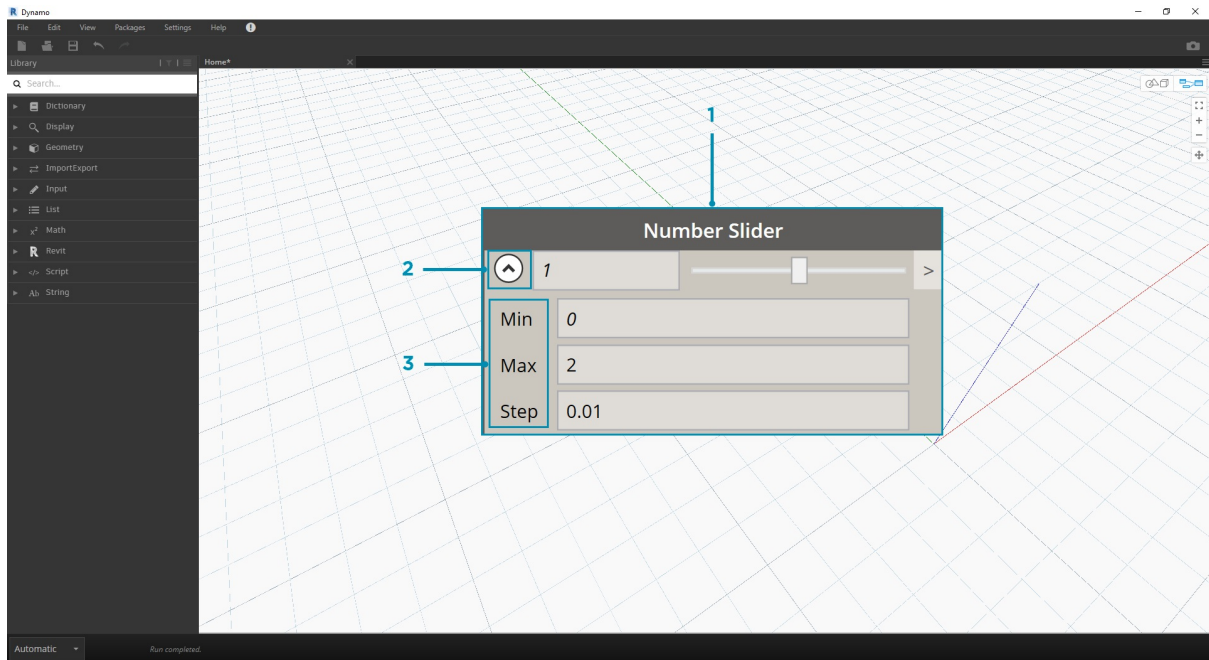
1. **Circle.ByPlaneRadius:** ascendiendo en la jerarquía, ahora creamos una curva a partir del plano del paso anterior. Después de conectar el nodo, se obtiene un círculo en el origen. El radio predeterminado del nodo es el valor de 1 .



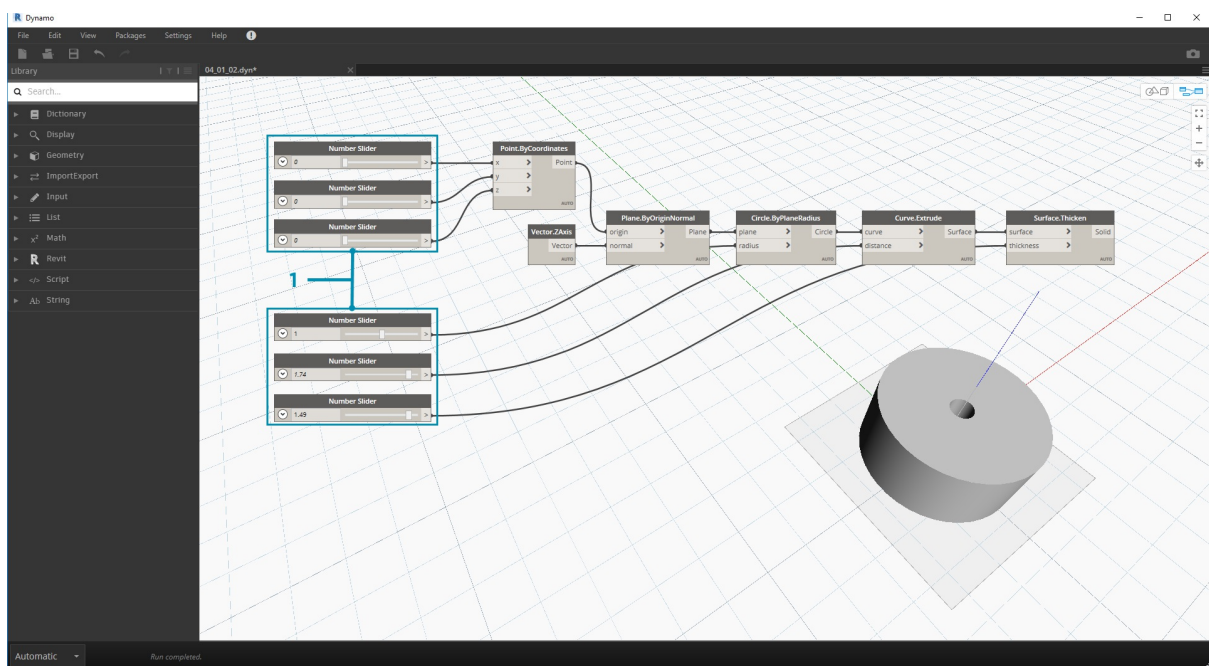
1. **Curve.Extrude:** ahora vamos a hacer que este elemento crezca dándole profundidad y añadiendo la tercera dimensión. Este nodo crea una superficie a partir de una curva extruyéndola. La distancia por defecto en el nodo es 1, y debería aparecer un cilindro en la ventana gráfica.



1. **Surface.Thicken:** este nodo nos proporciona un sólido cerrado desfasando la superficie una distancia dada y cerrando la forma. El valor de grosor por defecto es 1, y vemos un cilindro vaciado en la ventana gráfica en línea con estos valores.

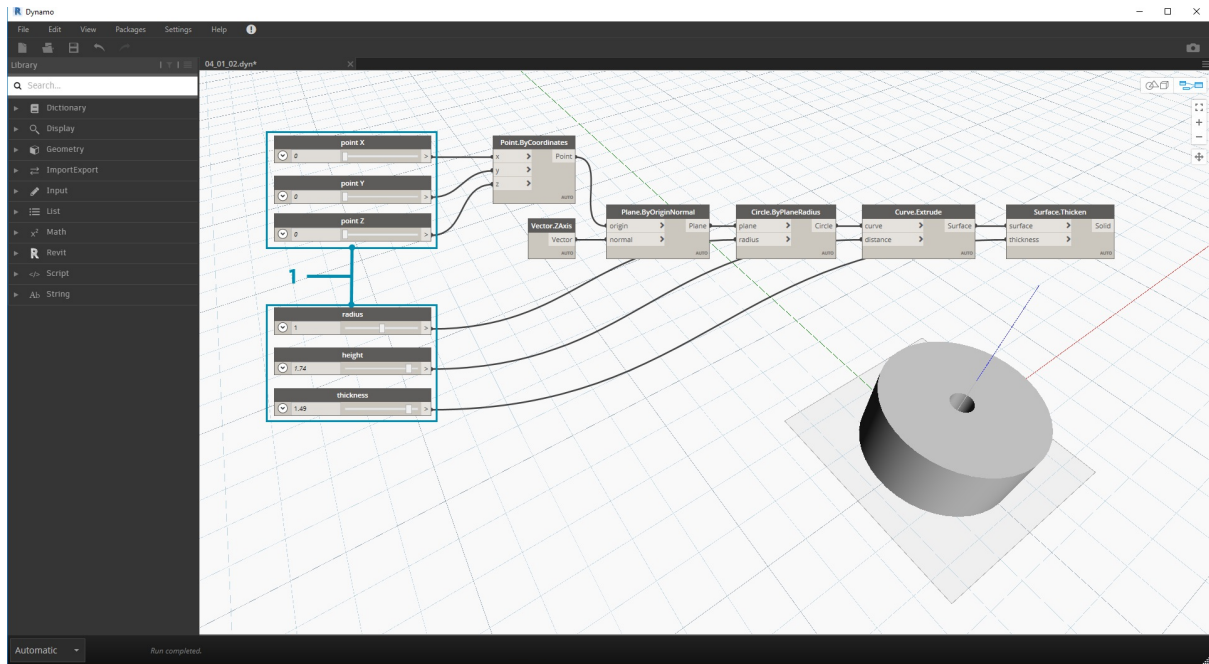


1. **Number Slider:** en lugar de utilizar los valores por defecto para todas estas entradas, vamos a añadir algún control paramétrico al modelo.
2. **Edición del dominio:** después de añadir el control deslizante de número al lienzo, haga clic en el signo de intercalación situado en la parte superior izquierda para ver las opciones de dominio.
3. **Min/Max/Step:** cambie los valores *min*, *max* y *step* a 0, 2 y 0,01 respectivamente. Esto lo hacemos para controlar el tamaño de la geometría global.



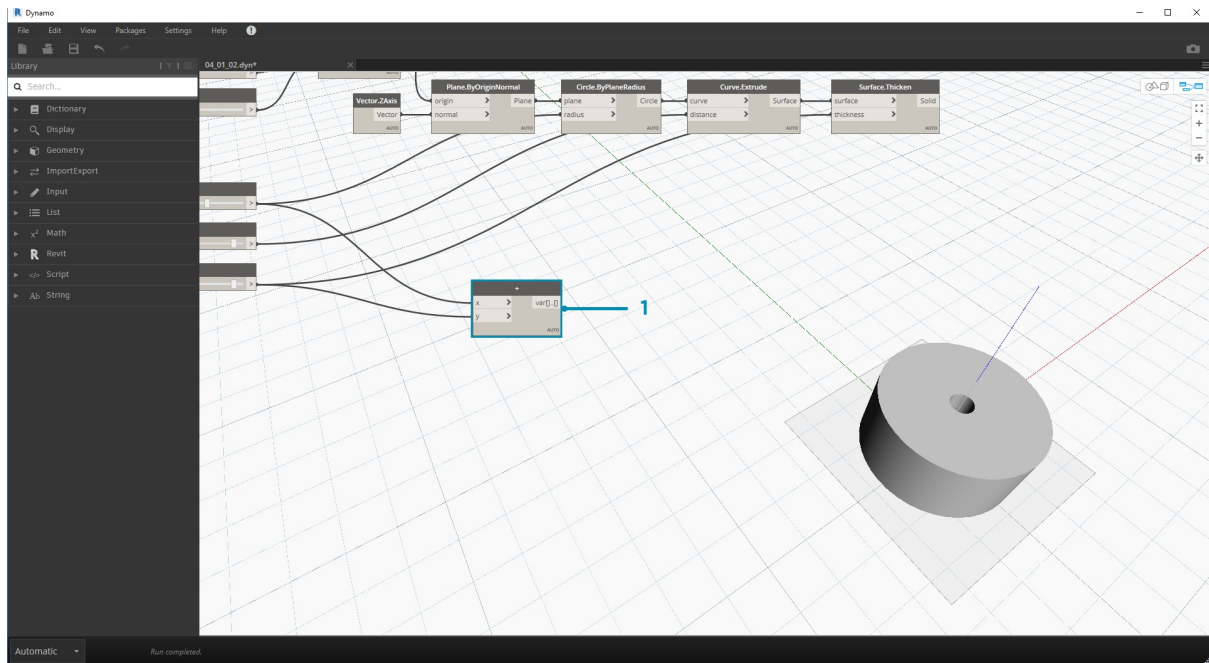
1. **Nodos Number Slider:** en todas las entradas por defecto, copiemos y peguemos este control deslizante de número (seleccione el control deslizante, pulse Ctrl+C y, a continuación, Ctrl+V) varias veces hasta que todas las entradas con valores por defecto tengan un control deslizante. Algunos de los valores del control deslizante tendrán que ser mayores que cero para que la definición funcione (es decir, para que una superficie se engrose, se necesita una profundidad de extrusión).

Hemos creado un cilindro paramétrico vaciado con estos controles deslizantes. Pruebe a modificar algunos de estos parámetros y ver cómo la geometría se actualiza de forma dinámica en la ventana gráfica de Dynamo.

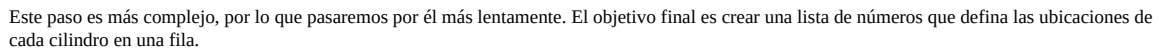


1. Nodos **Number Slider**: al avanzar en este ejemplo, hemos añadido muchos controles deslizantes al lienzo y debemos limpiar la interfaz de la herramienta que acabamos de crear. Haga clic con el botón derecho en un control deslizante, seleccione "Cambiar nombre..." y cambie cada control deslizante al nombre apropiado para su parámetro. Puede tomar como referencia la imagen anterior para definir los nombres.

En este punto, hemos creado un fantástico elemento de cilindro engrosado. Se trata de un objeto actualmente; veamos cómo crear una matriz de cilindros que permanezca enlazada de forma dinámica. Para ello, vamos a crear una lista de cilindros, en lugar de trabajar con un único elemento.

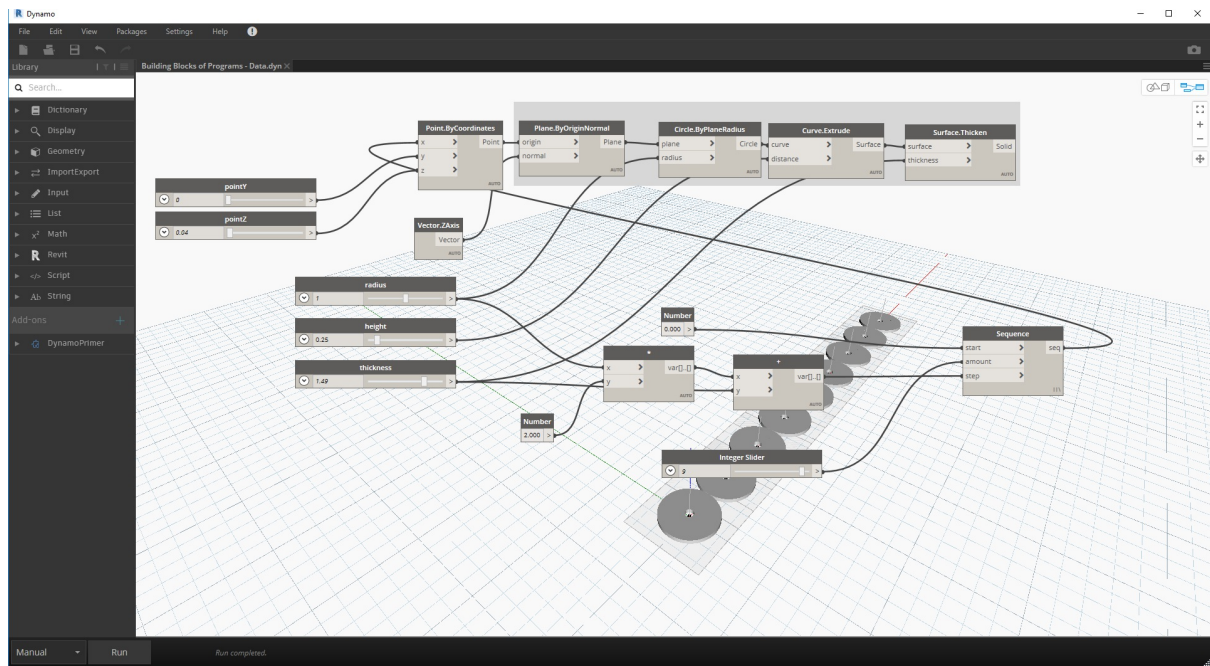


1. **Adición (+)**: nuestro objetivo es añadir una fila de cilindros junto al cilindro que hemos creado. Si se desea añadir un cilindro adyacente al actual, se deben tener en cuenta el radio del cilindro y el grosor del vaciado. Para obtener este número, se añaden los dos valores de los controles deslizantes.



-
- The screenshot shows the Dynamo software interface with a complex graph for creating a spiral staircase. The graph is connected to a 3D model of a spiral staircase on a grid background. The graph includes the following nodes and connections:
- Input Nodes:**
 - pointY:** A number input node with the value 0.
 - pointZ:** A number input node with the value 0.04.
 - radius:** A number input node with the value 7.
 - height:** A number input node with the value 1.74.
 - thickness:** A number input node with the value 1.49.
 - Number (2.000):** A number input node.
 - Integer Slider:** A slider node set to 9.
 - Processing Nodes:**
 - Point.ByCoordinates:** Takes pointY, pointZ, and Vector.ZAxis as inputs to create a point.
 - Plane.ByOriginNormal:** Takes the point from Point.ByCoordinates and Vector.ZAxis as inputs to create a plane.
 - Circle.ByPlaneRadius:** Takes the plane from Plane.ByOriginNormal and the radius input as inputs to create a circle.
 - Curve.Extrude:** Takes the circle from Circle.ByPlaneRadius and the height input as inputs to create a curve.
 - Surface.Thicken:** Takes the curve from Curve.Extrude and the thickness input as inputs to create a surface.
 - Sequence:** A sequence node that generates a series of values for the curve's position.
 - Number (0.000):** A number input node used in the sequence node.
 - Integer Slider:** A slider node used in the sequence node.
 - Output and Visualization:**
 - The final output is a 3D model of a spiral staircase, which is a series of connected surfaces.
 - The staircase is shown on a grid background, with a red dashed line indicating the path of the curve.
 - A blue arrow labeled '1' points to the staircase, indicating the final result.

1. Este paso es bastante sencillo: conecte la secuencia definida en el paso anterior a la entrada *x* de *Point.ByCoordinates* original. De este modo, se sustituye el control deslizante *pointX*, que se puede eliminar. Ahora vemos una matriz de cilindros en la ventana gráfica (asegúrese de que el control deslizante de enteros es mayor que 0).



La cadena de cilindros sigue enlazada dinámicamente a todos los controles deslizantes. Mueva cada uno de los controles deslizantes para ver cómo se actualiza la definición.

Matemáticas

Matemáticas

Si la forma más sencilla de datos es el número, la forma más sencilla de relacionarlos es mediante las matemáticas. Desde operadores sencillos como la división a funciones trigonométricas y fórmulas más complejas, las matemáticas son un método excelente para empezar a explorar relaciones y patrones numéricos.

Operadores aritméticos

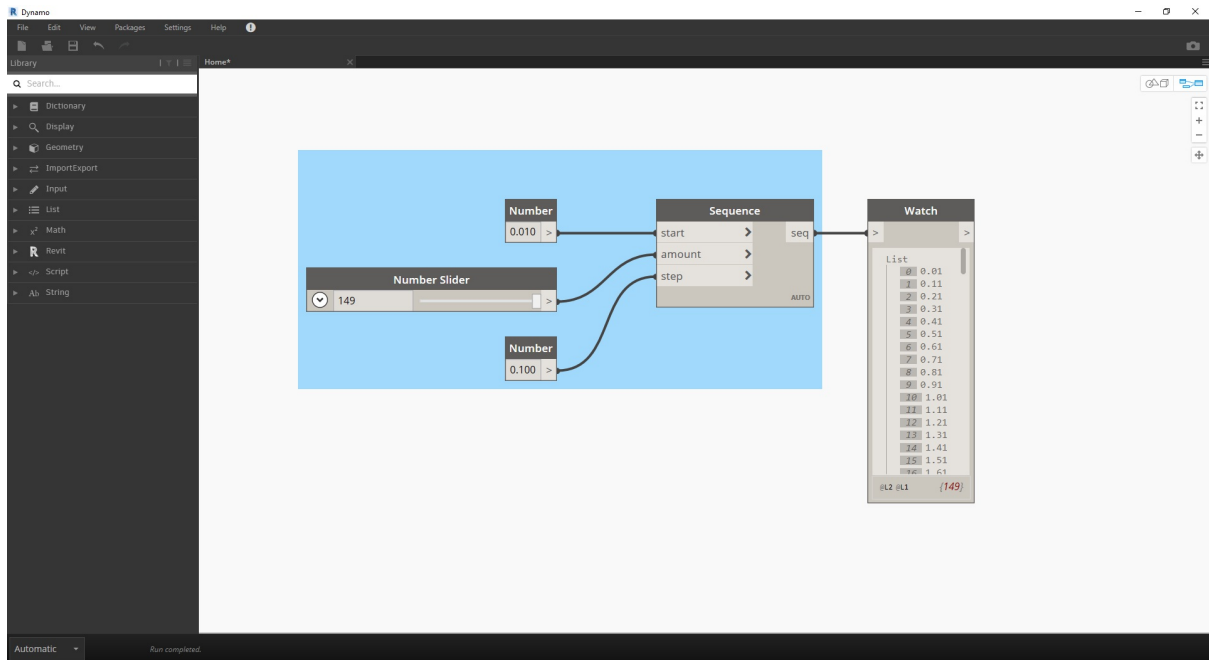
Los operadores son un conjunto de componentes que utilizan funciones algebraicas con dos valores numéricos de entrada que dan como resultado un valor de salida (suma, resta, multiplicación, división, etc.). Estos se pueden encontrar en Operators > Actions.

Icono	Nombre	Sintaxis	Entradas	Salidas
	Sumar	+	var[...] , var[...]	var[...]
	Restar	-	var[...] , var[...]	var[...]
	Multiplicar *		var[...] , var[...]	var[...]
	Dividir	/	var[...] , var[...]	var[...]

Fórmula paramétrica

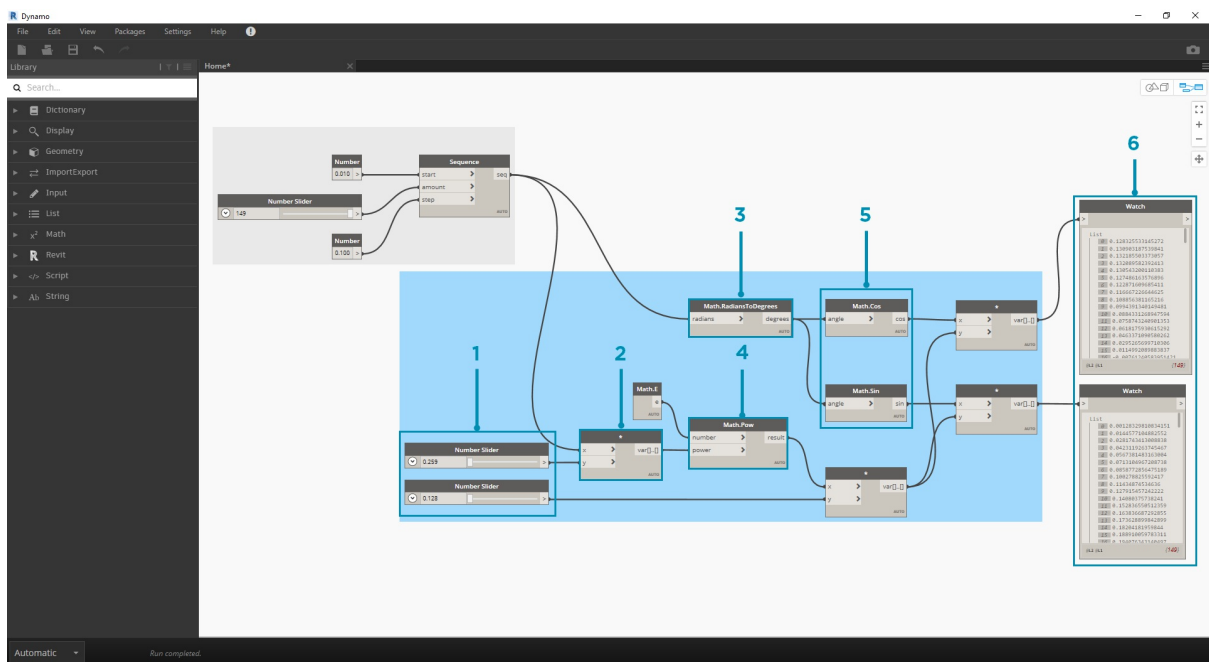
Descargue el archivo de ejemplo que acompaña a este ejercicio (haga clic con el botón derecho y seleccione "Guardar enlace como..."): [Building Blocks of Programs - Math.dyn](#). En el Apéndice se incluye una lista completa de los archivos de ejemplo.

A partir de los operadores, el siguiente paso lógico consiste en combinar operadores y variables para formar una relación más compleja mediante **fórmulas**. Hagamos una fórmula que se pueda controlar mediante parámetros de entrada, como los controles deslizantes.



1. **Number Sequence:** define una secuencia de números basada en tres entradas: *start*, *amount* y *step*. Esta secuencia representa la "t" de la ecuación paramétrica, por lo que debemos utilizar una lista lo suficientemente grande como para definir una espiral.

En el paso anterior hemos creado una lista de números para definir el dominio paramétrico. La espiral dorada se define como la ecuación: $x = r \cos \theta e^{b\theta}$ y $y = r \sin \theta e^{b\theta}$. El grupo de nodos que se muestra a continuación representa esta ecuación en forma de programación visual.



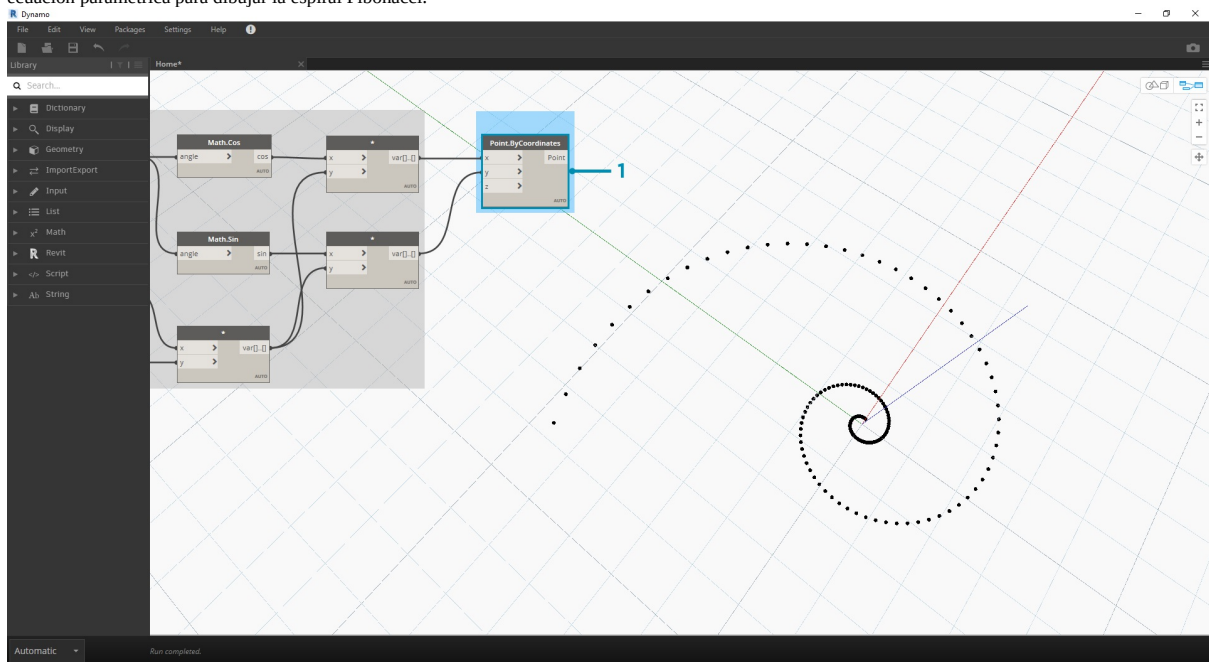
Al desplazarse por el grupo de nodos, intente prestar atención al paralelismo entre el programa visual y la ecuación por escrito.

1. **Number Slider:** añade dos controles deslizantes de número al lienzo. Estos controles deslizantes representarán las variables *a* y *b* de la ecuación paramétrica. Estas representan una constante que es flexible o parámetros que podemos ajustar hacia el resultado deseado.
2. *****: el nodo de multiplicación se representa mediante un asterisco. Lo usaremos con frecuencia para conectar variables de multiplicación.
3. **Math.RadiansToDegrees:** los valores de "t" deben traducirse a grados para ser evaluados en las funciones trigonométricas. Recuerde que Dynamo utiliza por defecto los grados para evaluar estas funciones.
4. **Math.Pow:** como función de "t" y del número "e", crea la secuencia Fibonacci.
5. **Math.Cos** y **Math.Sin:** estas dos funciones trigonométricas distinguirán la coordenada X y la coordenada Y, respectivamente, de cada punto paramétrico.
6. **Watch:** ahora vemos que las salidas son dos listas, que serán las coordenadas x e y de los puntos utilizados para generar la espiral.

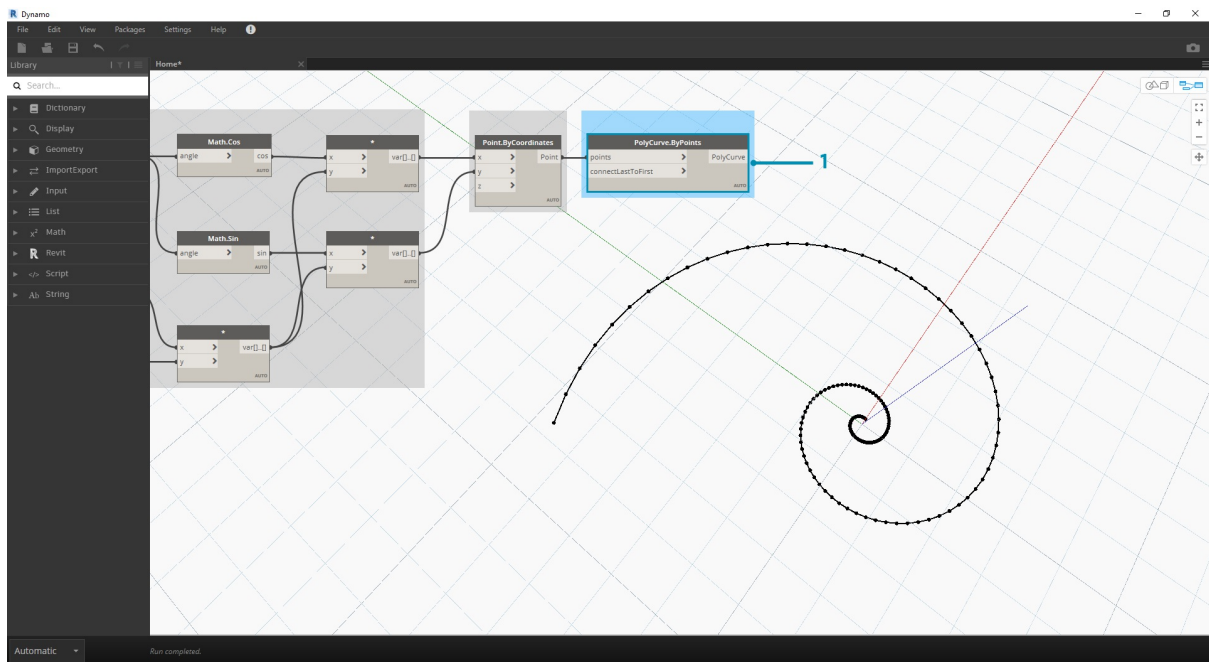
De fórmula a geometría

El cúmulo de nodos del paso anterior funcionará correctamente, pero conlleva mucho trabajo. Para crear un flujo de trabajo más eficiente, consulte **Bloques**

de código (sección 3.3.2.3) para definir una cadena de expresiones de Dynamo en un nodo. En la siguiente serie de pasos, veremos cómo utilizar la ecuación paramétrica para dibujar la espiral Fibonacci.



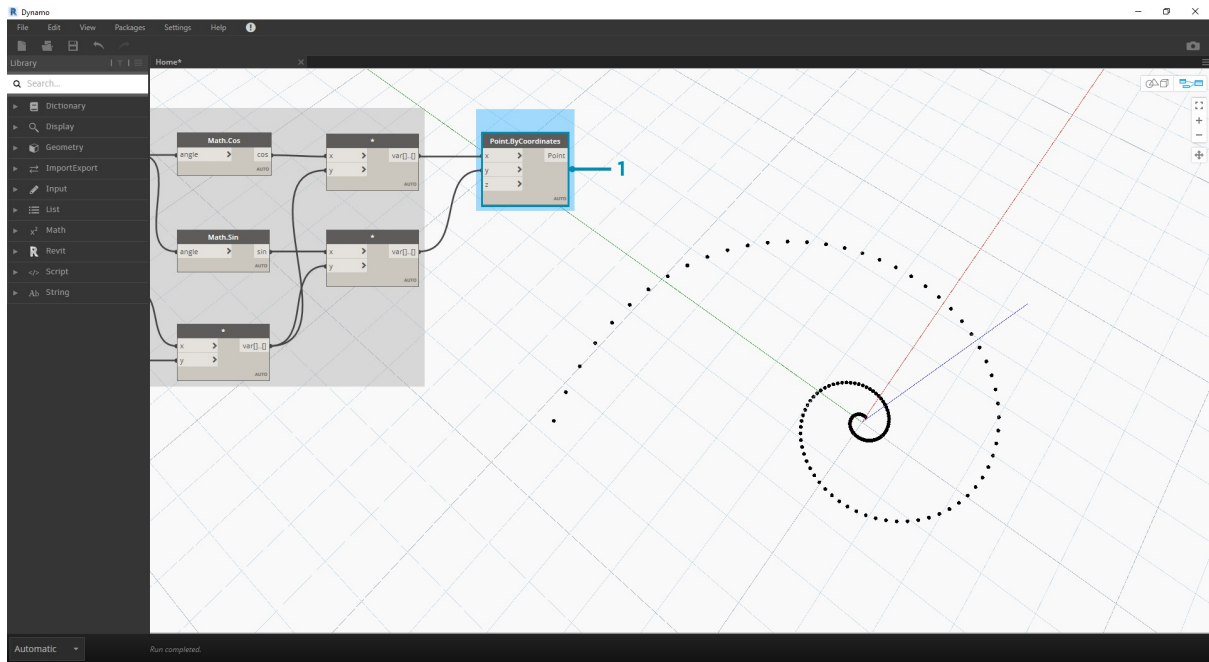
1. **Point.ByCoordinates:** conecte el nodo de multiplicación superior a la entrada "x" y el inferior a la entrada "y". Ahora se muestra una espiral paramétrica de puntos en la pantalla.



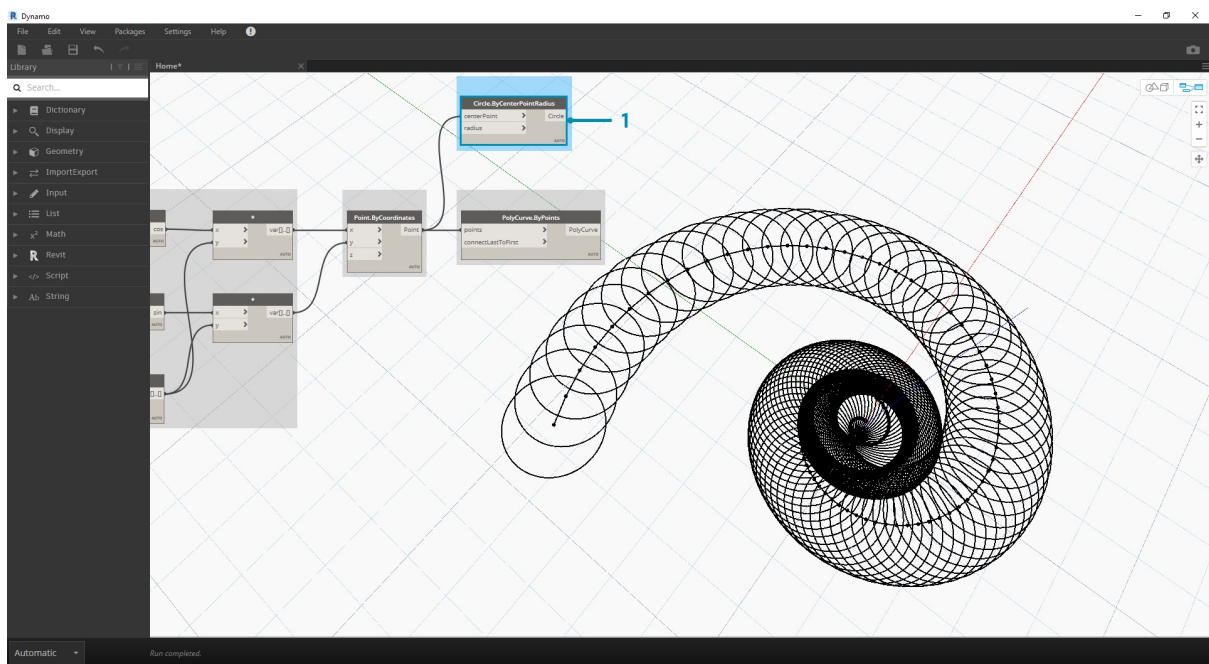
1. **Polycurve.ByPoints:** Conecte Point.ByCoordinates del paso anterior a *points*. Podemos dejar *connectLastToFirst* sin entrada porque no estamos creando una curva cerrada. Se crea una espiral que pasa a través de cada punto definido en el paso anterior.

Ya hemos completado la espiral Fibonacci. Vamos a profundizar en estos conceptos con dos ejercicios separados a partir de aquí; a uno lo llamaremos caracol y al otro girasol. Se trata de abstracciones de sistemas naturales, pero las dos aplicaciones diferentes de la espiral Fibonacci estarán bien representadas.

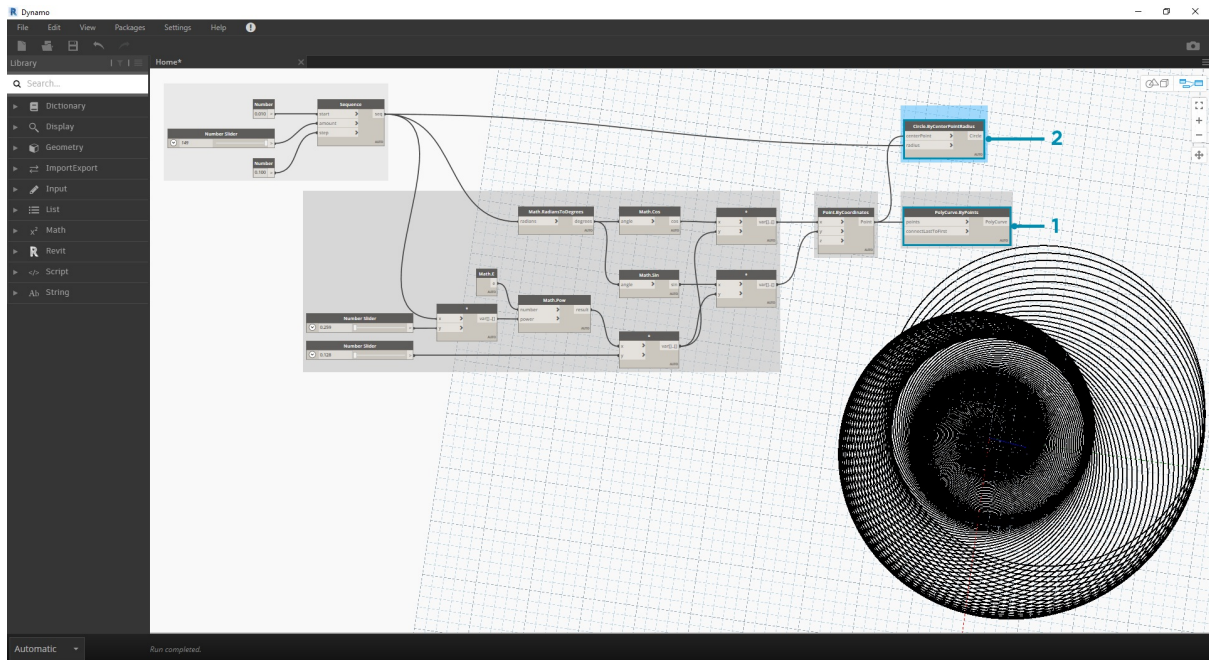
De espiral a caracol



1. Como punto de partida, vamos a realizar el mismo paso del ejercicio anterior: crear de una matriz de puntos de espiral con el nodo **Point.ByCoordinates**.



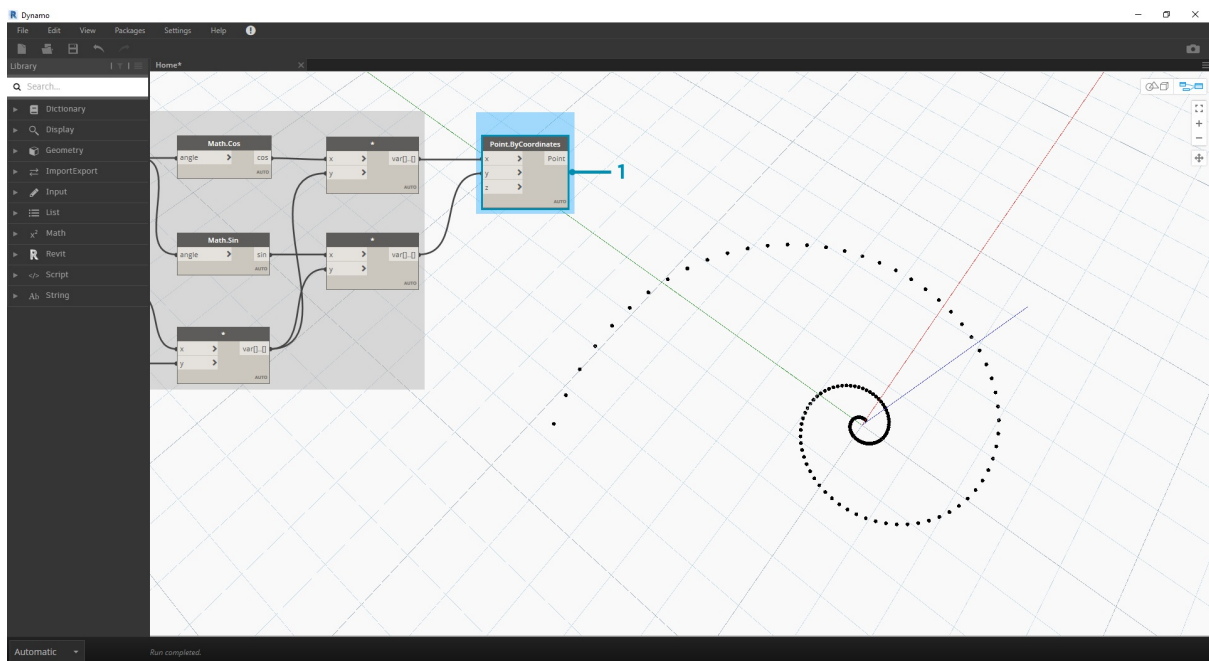
1. **Polycurve.ByPoints**: este nodo también lo utilizamos en el ejercicio anterior y lo usaremos como referencia.
2. **Circle.ByCenterPointRadius**: utilizaremos aquí un nodo de círculo con las mismas entradas que en el paso anterior. El valor por defecto del radio es $1,0$, por lo que vemos una salida de círculos inmediata. Se hace inmediatamente evidente cómo los puntos divergen más respecto al origen.



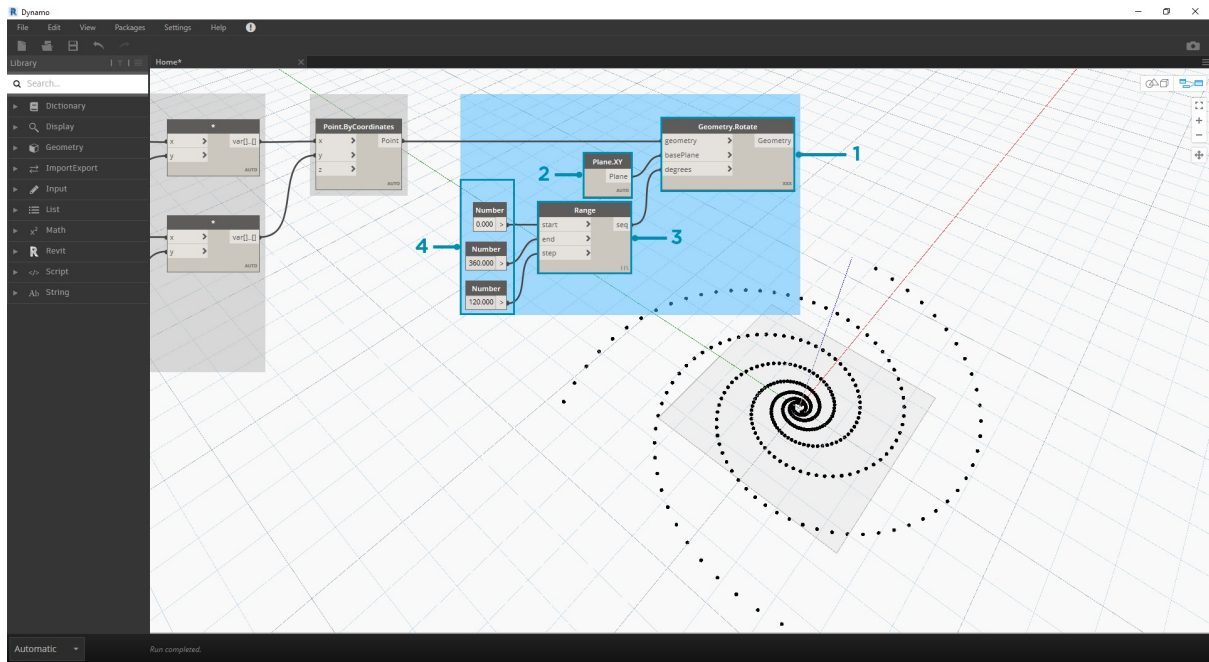
1. **Circle.ByCenterPointRadius:** para crear una matriz de círculos más dinámica, conectamos la secuencia de números original (la secuencia "t") en el valor del radio.
2. **Number Sequence:** es la matriz original de "t". Al conectarlo con el valor del radio, los centros de círculo siguen divergiendo más respecto del origen, pero el radio de los círculos aumenta, lo que crea un gráfico de círculos Fibonacci con mucho estilo. Un premio para quién consiga convertirlo a 3D.

De caracol al patrón de filotaxis

Ahora que hemos hecho una concha de caracol circular, vamos a pasar a las rejillas paramétricas. Vamos a usar una rotación básica en la espiral Fibonacci para crear una rejilla Fibonacci y modelar el resultado según el [crecimiento de las semillas de girasol](#).

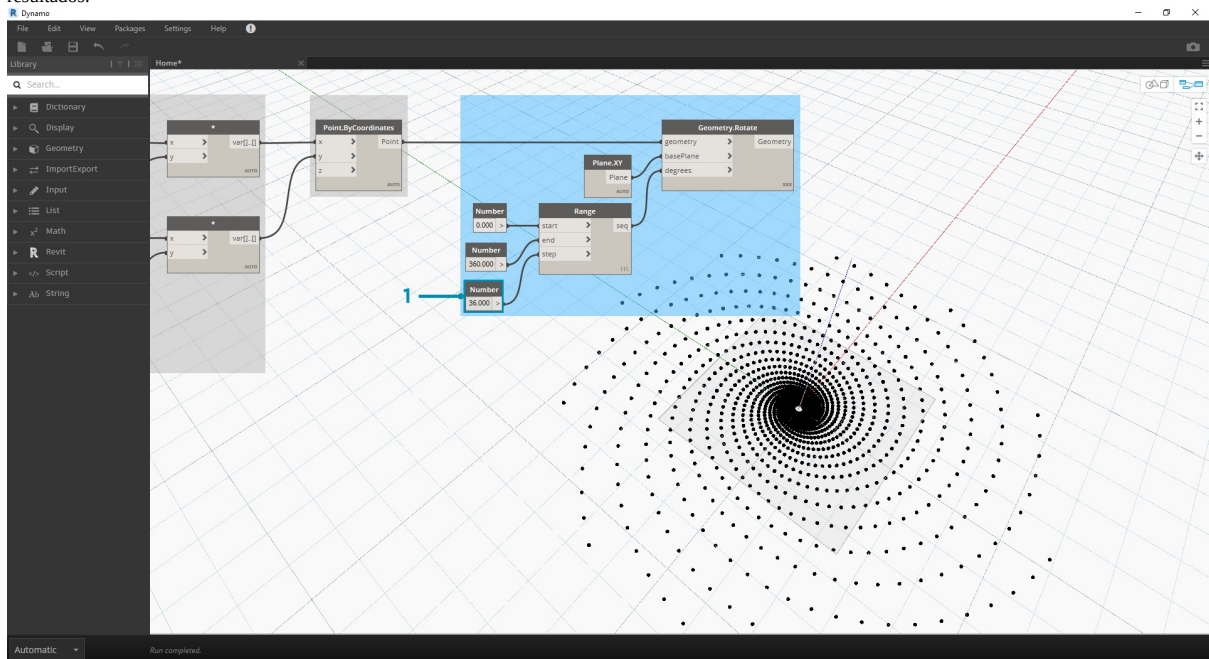


1. Como punto de partida, vamos a realizar otra vez el mismo paso del ejercicio anterior: crear de una matriz de puntos de espiral con el nodo **Point.ByCoordinates**.

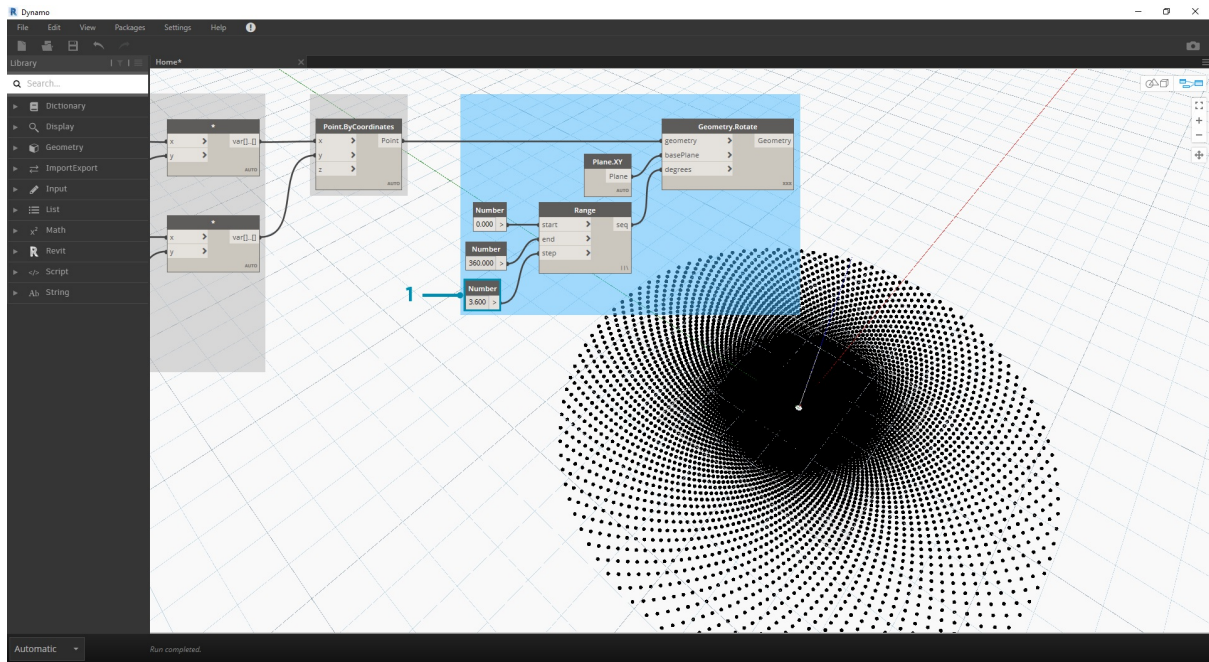


1. **Geometry.Rotate**: existen varias opciones de Geometry.Rotate; asegúrese de que ha elegido el nodo que contiene las entradas *geometry*, *basePlane* y *degrees*. Conecte **Point.ByCoordinates** en la entrada *geometry*.
2. **Plano.XY**: conéctelo a la entrada *basePlane*. Rotaremos alrededor del origen, que es la misma ubicación que la base de la espiral.
3. **Number Range**: para la entrada de grados, vamos a crear varias rotaciones. Esto se puede realizar rápidamente con un componente Number Range. Conéctelo a la entrada *degrees*.
4. **Number**: para definir el rango de números, añada tres nodos numéricos al lienzo en orden vertical. De arriba a abajo, asigne valores de 0,0; 360,0 y 120,0 respectivamente. Estos valores controlan la rotación de la espiral. Observe los resultados de las salidas del nodo **Number Range** después de conectar los tres nodos numéricos al mismo.

Nuestro resultado está empezando a parecerse a un remolino. Ajustemos algunos de los parámetros de **Number Range** y veamos cómo cambian los resultados:



1. Cambie el tamaño de paso del nodo **Number Range** de 120,0 a 36,0. Observe que esto crea más rotaciones y que, por tanto, nos proporciona una rejilla más densa.



1. Cambie el tamaño de paso del nodo **Number Range** de 36,0 a 3,6. Esto nos da una rejilla mucho más densa y la dirección de la espiral no está clara. Damas y caballeros, hemos creado un girasol.

Lógica

Lógica

La **lógica** o, más específicamente, la **lógica condicional** nos permite especificar una acción o un conjunto de acciones basadas en una prueba. Tras evaluar la prueba, dispondremos de un valor booleano que representa **Verdadero** o **Falso**, que podemos utilizar para controlar el flujo del programa.

Booleanos

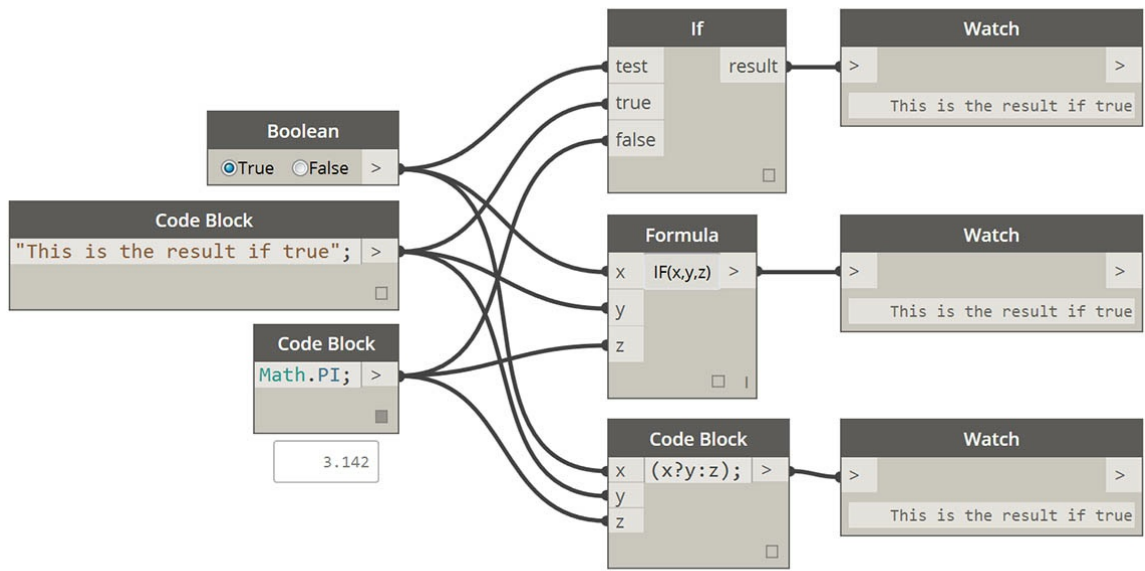
Las variables numéricas pueden almacenar una gran variedad de números diferentes. Las variables booleanas solo pueden almacenar dos valores denominados Verdadero o Falso, Sí o No, 1 o 0. Rara vez utilizamos booleanos para realizar cálculos debido a su rango limitado.

Instrucciones condicionales

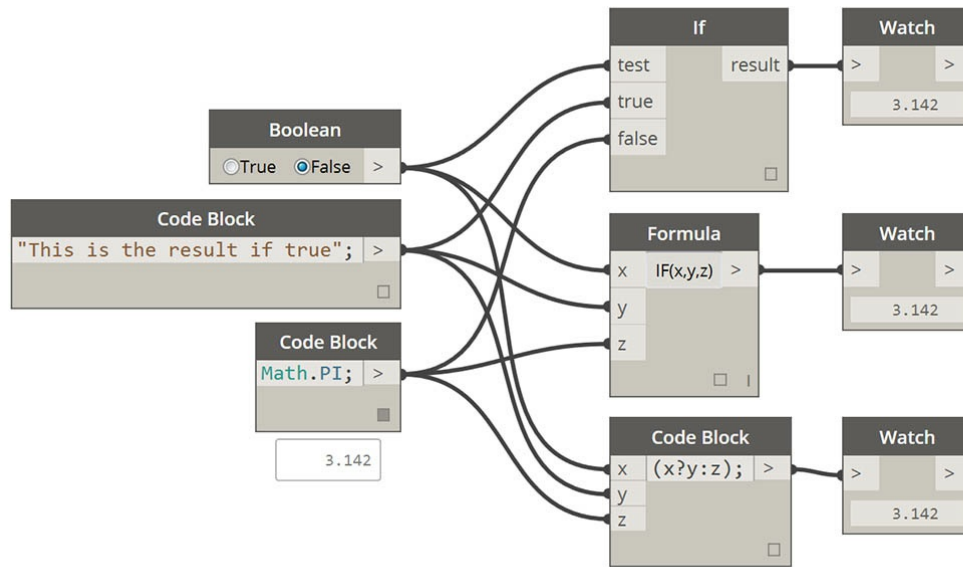
La instrucción "If" ("si") es un concepto clave en la programación: "Si *esto* es cierto, *eso* ocurre; de lo contrario, *otra cosa* ocurre. La acción resultante de la instrucción se rige por un valor booleano. Hay varias formas de definir una instrucción "If" en Dynamo:

Icono	Nombre	Sintaxis	Entradas	Salidas
	If	If	test, true, false	resultado
	Fórmula	IF(x,y,z)	x, y, z	resultado
	Bloque de código (x?y:z)	x, y, z		resultado

Veamos un breve ejemplo de cada uno de estos tres nodos en acción utilizando la instrucción condicional "If":



En esta imagen, el no *Boolean* se ha establecido en *Verdadero (True)*, lo que significa que el resultado es una cadena que dice: "este es el resultado si es verdadero". Los tres nodos que crean la instrucción *If* funcionan de la misma forma aquí.

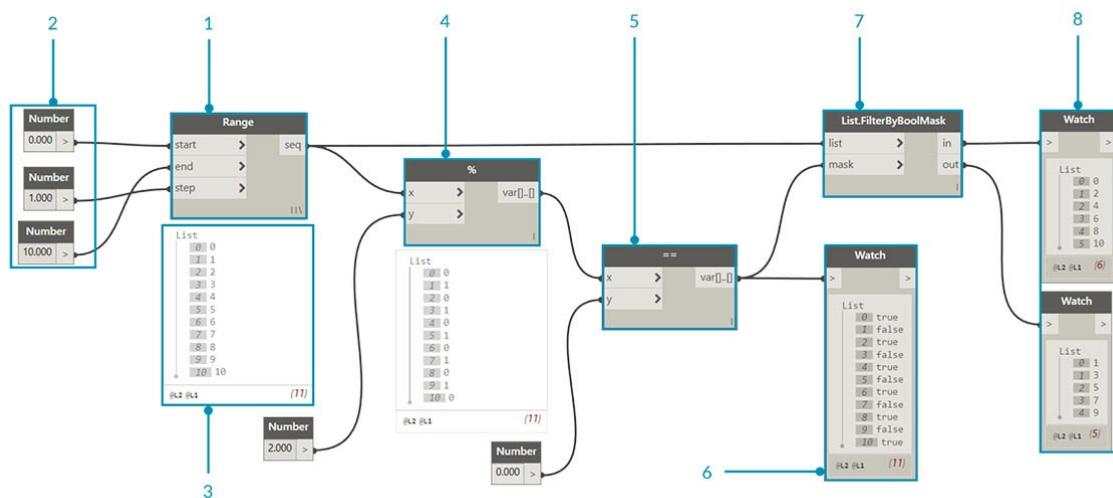


De nuevo, los nodos funcionan de la misma forma. Si el nodo *Boolean* se cambia a *Falso (False)*, el resultado es el número *Pi*, tal y como se define en la instrucción *If* original.

Filtrado de una lista

Descargue el archivo de ejemplo que acompaña a este ejercicio (haga clic con el botón derecho y seleccione "Guardar enlace como..."): [Building Blocks of Programs - Logic.dyn](#). En el Apéndice se incluye una lista completa de los archivos de ejemplo.

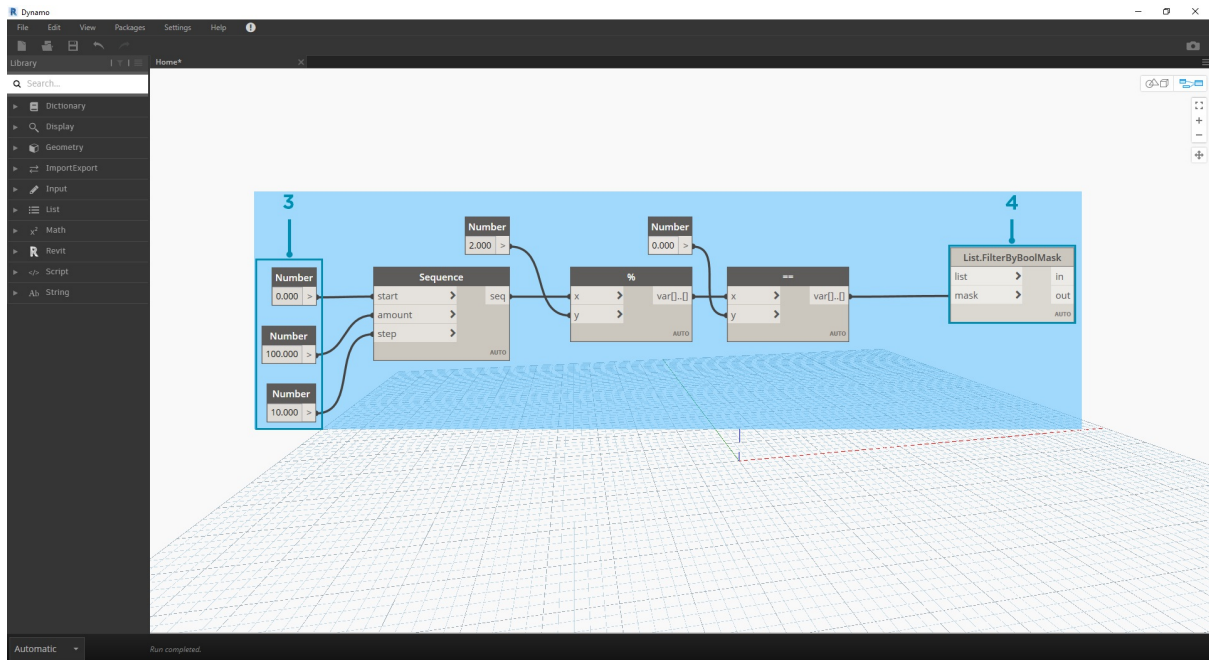
Usamos la lógica para separar una lista de números en una lista de números pares y una lista de números impares.



1. **Number Range**: añade un rango de números al lienzo.
2. **Nodos Number**: añade tres nodos numéricos al lienzo. El valor de cada nodo numérico debe ser: 0,0 para *start*, 10,0 para *end* y 1,0 para *step*.
3. **Salida**: nuestra salida es una lista de 11 números que van del 0 al 10.
4. **Módulo (%)**: *Number Range* se conecta a *x* y 2,0 a *y*. De este modo, se calcula el resto de cada número de la lista dividido por 2. La salida de esta lista nos proporciona una lista de valores alternantes entre 0 y 1.
5. **Prueba de igualdad (==)**: añade una prueba de igualdad al lienzo. Conecte la salida de *módulo* a la entrada *x* y 0,0 en la entrada *y*.
6. **Watch**: la salida de la prueba de igualdad es una lista de valores que alterna entre verdadero y falso. Estos son los valores utilizados para separar los elementos de la lista. 0 (o *true*) representa números pares y 1 (o *false*) representa números impares.
7. **List.FilterByBoolMask**: este nodo filtrará los valores en dos listas diferentes en función del valor booleano de entrada. Conecte el nodo *Number Range* original a la entrada *list* y la salida de la *prueba de igualdad* en la entrada *mask*. La salida *in* representa los valores verdaderos, mientras que la salida *out* representa los valores falsos.
8. **Watch**: como resultado, ahora tenemos una lista de números pares y una lista de números impares. Hemos utilizado operadores lógicos para separar las listas en patrones.

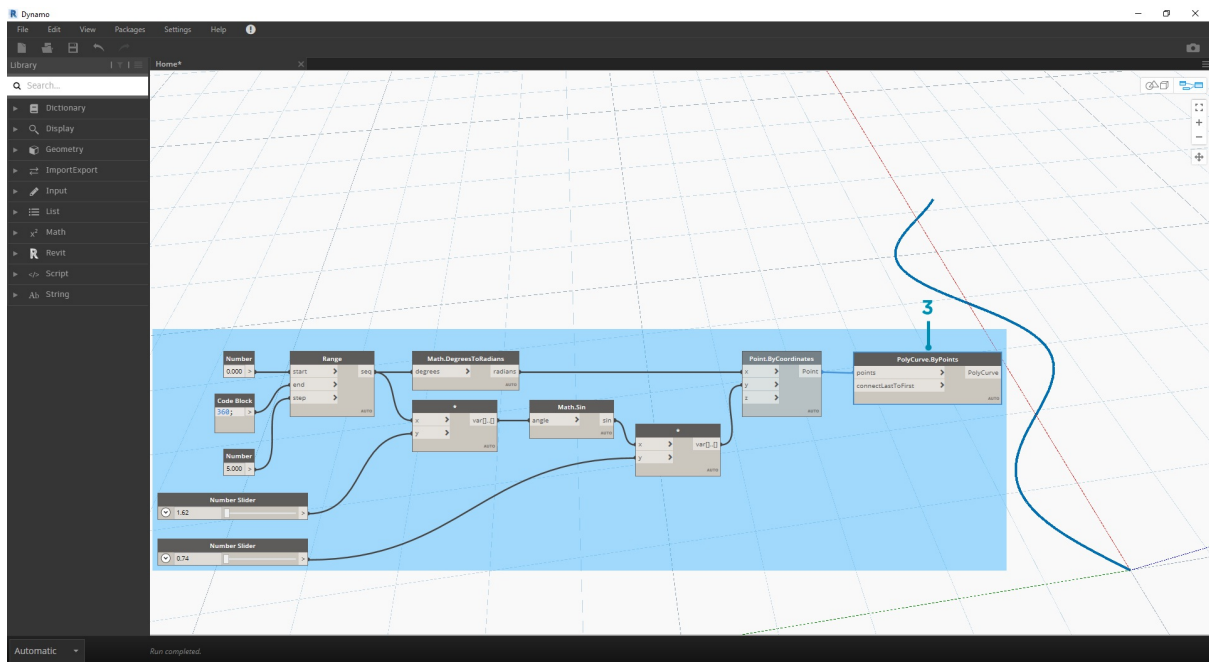
De lógica a geometría

Basándonos en la lógica establecida en el primer ejercicio, vamos a aplicar esta configuración en una operación de modelado.



Comenzaremos por donde dejamos el ejercicio anterior, con los mismos nodos. Las únicas excepciones (además de cambiar el formato son):

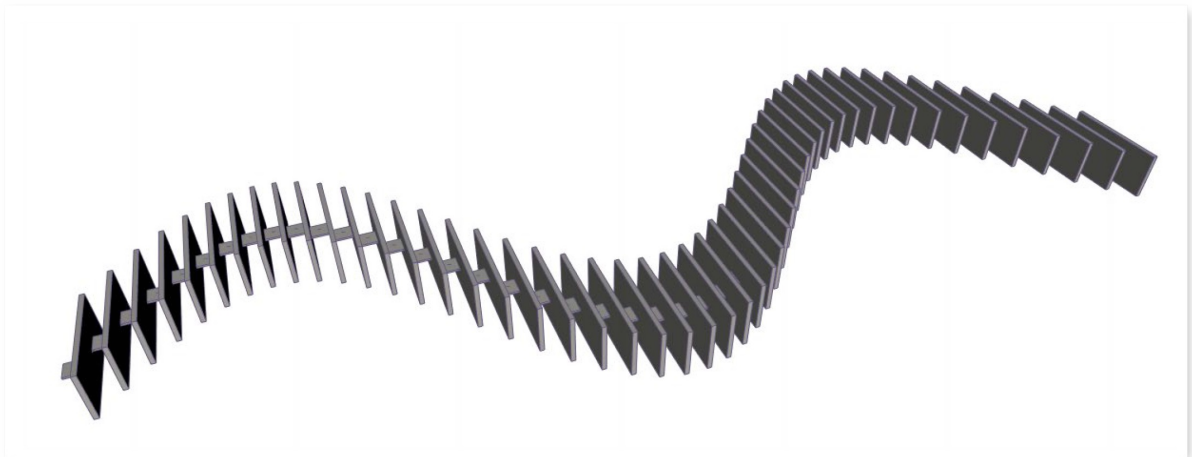
1. Los valores de entrada han cambiado.
2. Se ha desconectado la entrada list en *List.FilterByBoolMask*. Dejaremos estos nodos a un lado por ahora, pero en fases posteriores del ejercicio serán muy útiles.



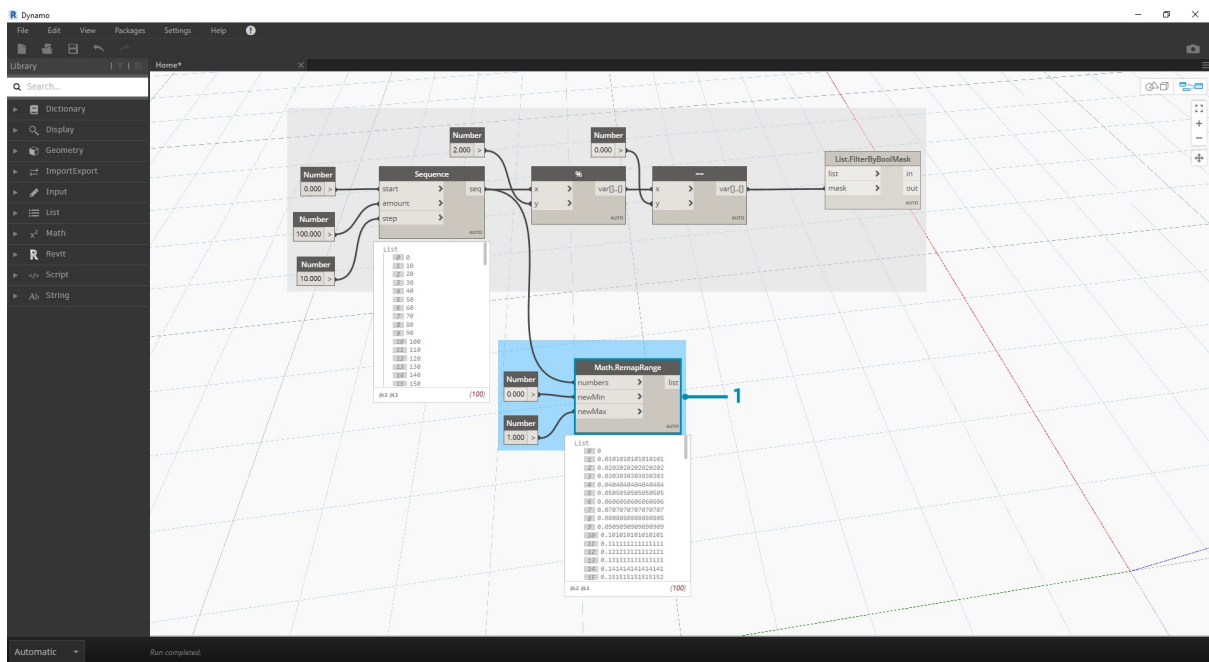
Comencemos por conectar los nodos tal como se muestra en la imagen anterior. Este grupo de nodos representa una ecuación paramétrica para definir una curva de línea. Información que debemos tener en cuenta:

1. El **primer control deslizante** debe tener un mínimo de 1, un máximo de 4 y un paso de 0,01.
2. El **segundo control deslizante** debe tener un mínimo de 0, un máximo de 1 y un paso de 0,01.
3. **PolyCurve.ByPoints**: si se copia el diagrama de nodos anterior, el resultado es una curva de seno en la ventana de vista preliminar de Dynamo.

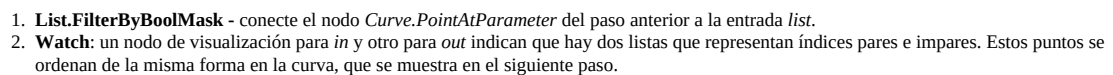
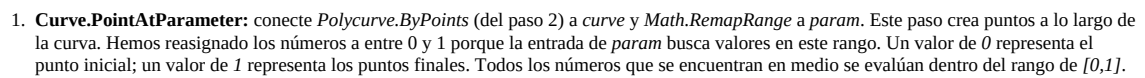
El método que se sigue para las entradas es utilizar nodos numéricos para las propiedades más estáticas y controles deslizantes de número para las más flexibles. Vamos a mantener el rango de números original que definimos al principio de este paso. Sin embargo, la curva de seno que se crea aquí debe tener cierta flexibilidad. Se pueden mover estos controles deslizantes para ver cómo se actualiza la frecuencia y la amplitud de la curva.

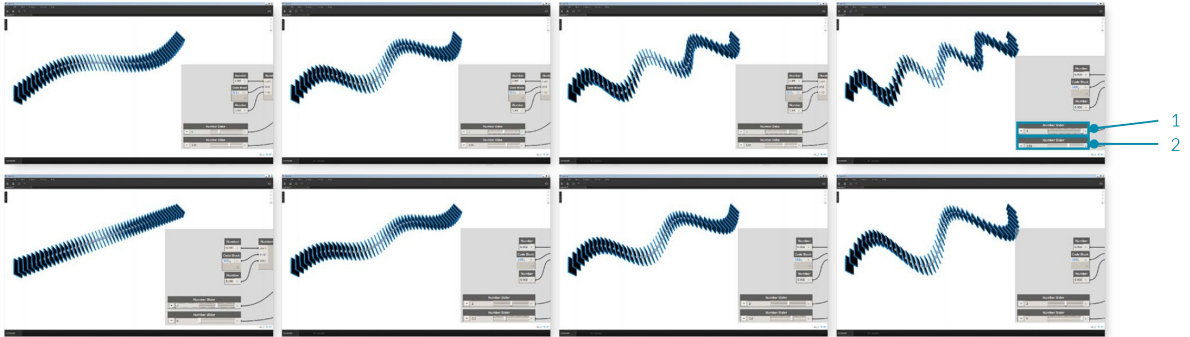


Vamos a saltar un poco en la definición, así que veamos el resultado final para que podamos tener como referencia lo que vamos a conseguir. Los dos primeros pasos se realizan por separado; ahora queremos conectar los dos. Utilizaremos la curva de seno base para controlar la ubicación de los componentes de cremallera y utilizaremos la lógica de verdadero/falso para alternar entre cuadros pequeños y cuadros más grandes.



1. **Math.RemapRange:** con la secuencia de números creada en el paso 01, vamos a crear una nueva serie de números reasignando el rango. Los números originales del paso 01 van del 0 al 100. Estos números oscilan entre 0 y 1 en las entradas *newMin* y *newMax* respectivamente.





1. **Number Slider**: retrocediendo al principio de la definición, podemos mover el control deslizante de número y ver cómo se actualiza la cremallera. La fila superior de imágenes representa un rango de valores para el control deslizante de número superior. Es la frecuencia de la onda.
2. **Number Slider**: la fila inferior de imágenes representa un rango de valores para el control deslizante inferior. Es la amplitud de la onda.

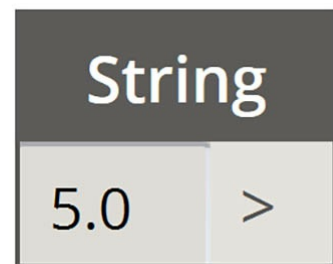
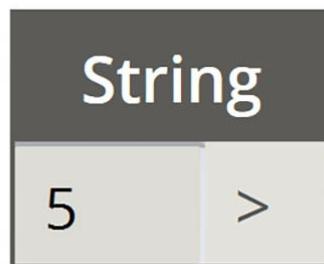
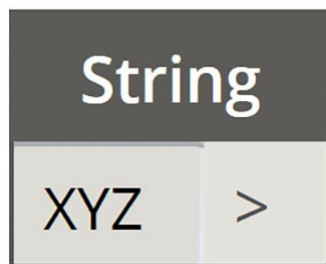
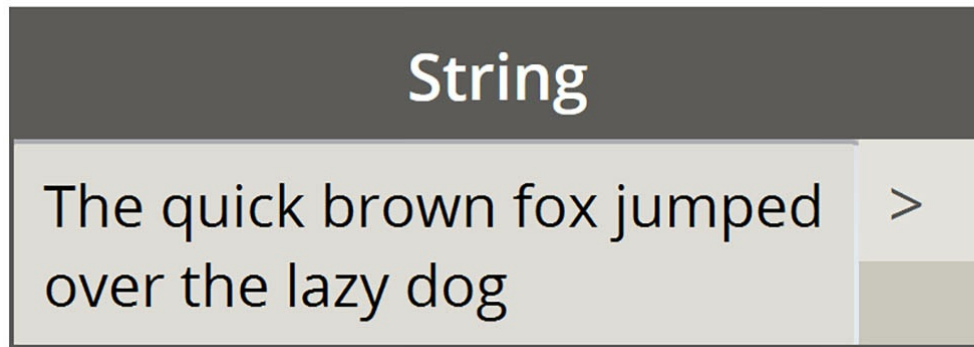
Cadenas

Cadenas

Formalmente, una **cadena** es una secuencia de caracteres que representa una constante literal o algún tipo de variable. Informalmente, una cadena es texto en la jerga de la programación. Hemos trabajado con números, enteros y decimales, para controlar los parámetros y podemos hacer lo mismo con el texto.

Creación de cadenas

Las cadenas se pueden utilizar para una amplia gama de aplicaciones, como la definición de parámetros personalizados, la anotación de conjuntos de documentación y el análisis mediante conjuntos de datos basados en texto. El nodo string se encuentra en la categoría Core > Input.



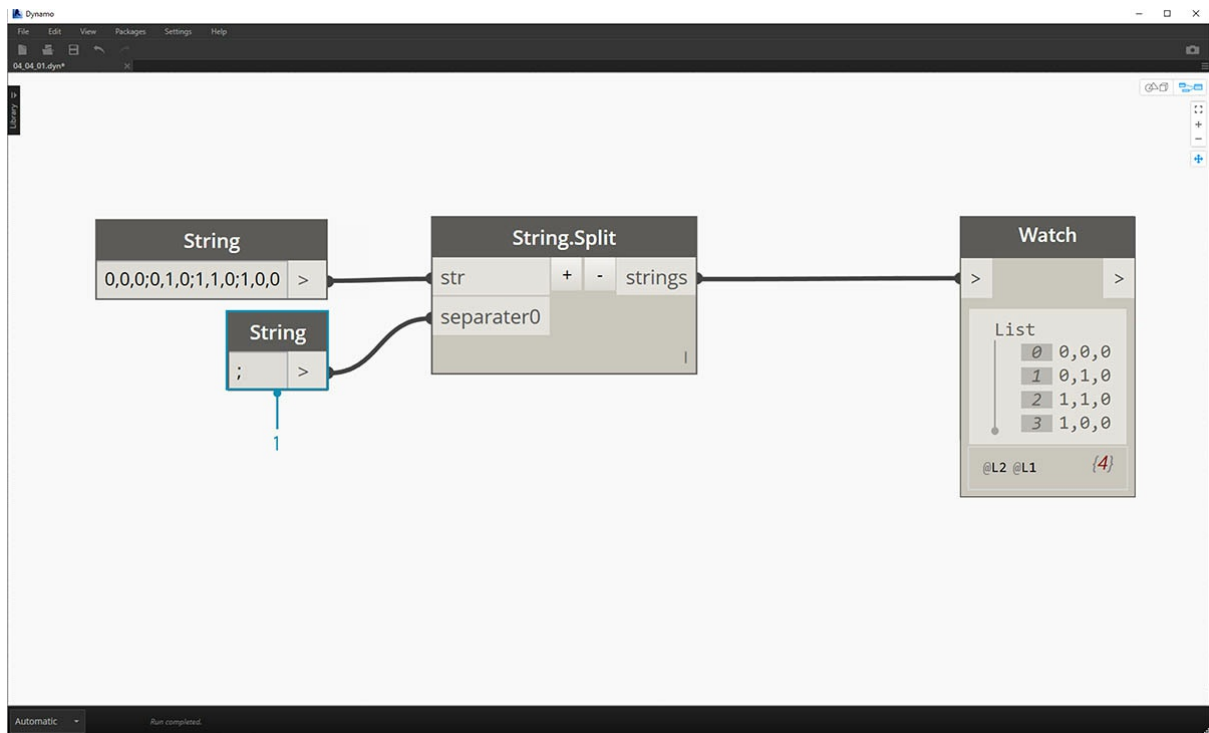
Los nodos de ejemplo anteriores son cadenas. Un número puede representarse como una cadena, al igual que una letra o una matriz completa de texto.

Consulta de cadenas

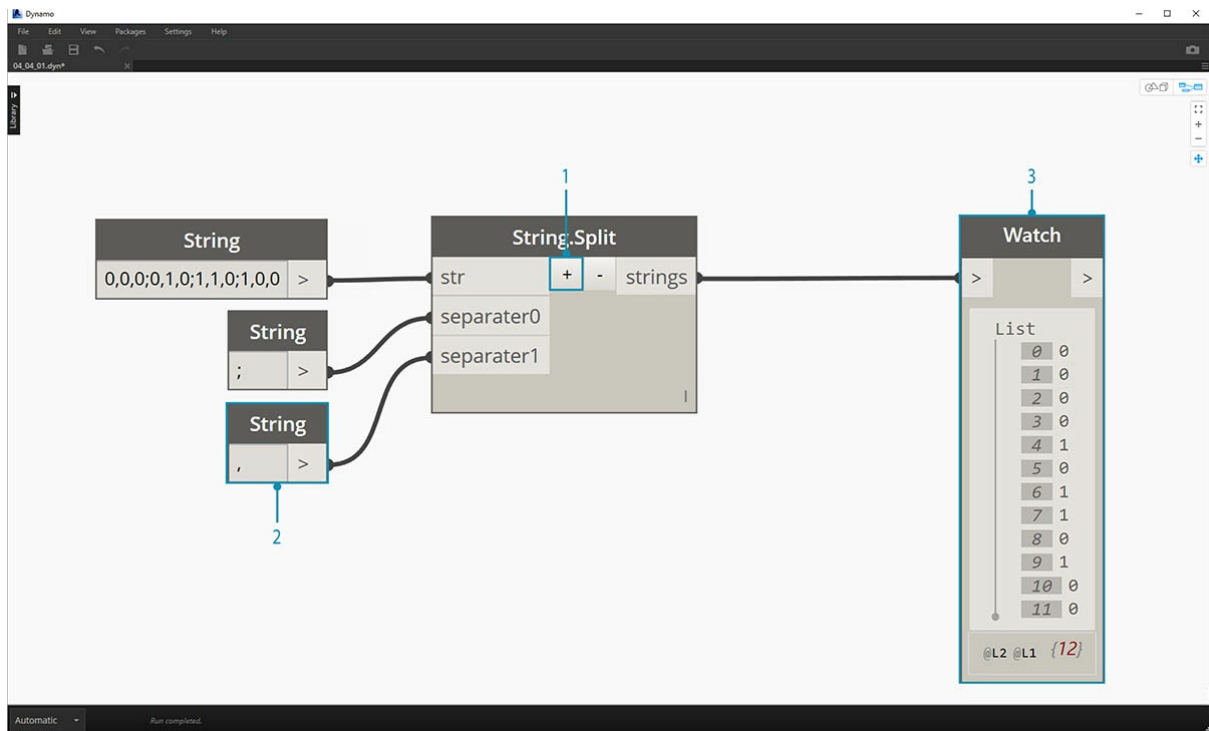
Descargue el archivo de ejemplo que acompaña a este ejercicio (haga clic con el botón derecho y seleccione "Guardar enlace como..."): [Building Blocks of Programs - Strings.dyn](#). En el Apéndice se incluye una lista completa de los archivos de ejemplo.

Puede analizar rápidamente grandes cantidades de datos consultando cadenas. Hablaremos de algunas operaciones básicas que pueden acelerar un flujo de trabajo y ayudar a mejorar la interoperabilidad de software.

La imagen siguiente se basa en una cadena de datos procedentes de una hoja de cálculo externa. La cadena representa los vértices de un rectángulo en el plano XY. Vamos a dividir algunas operaciones de división de cadenas en un ejercicio de miniaturas:

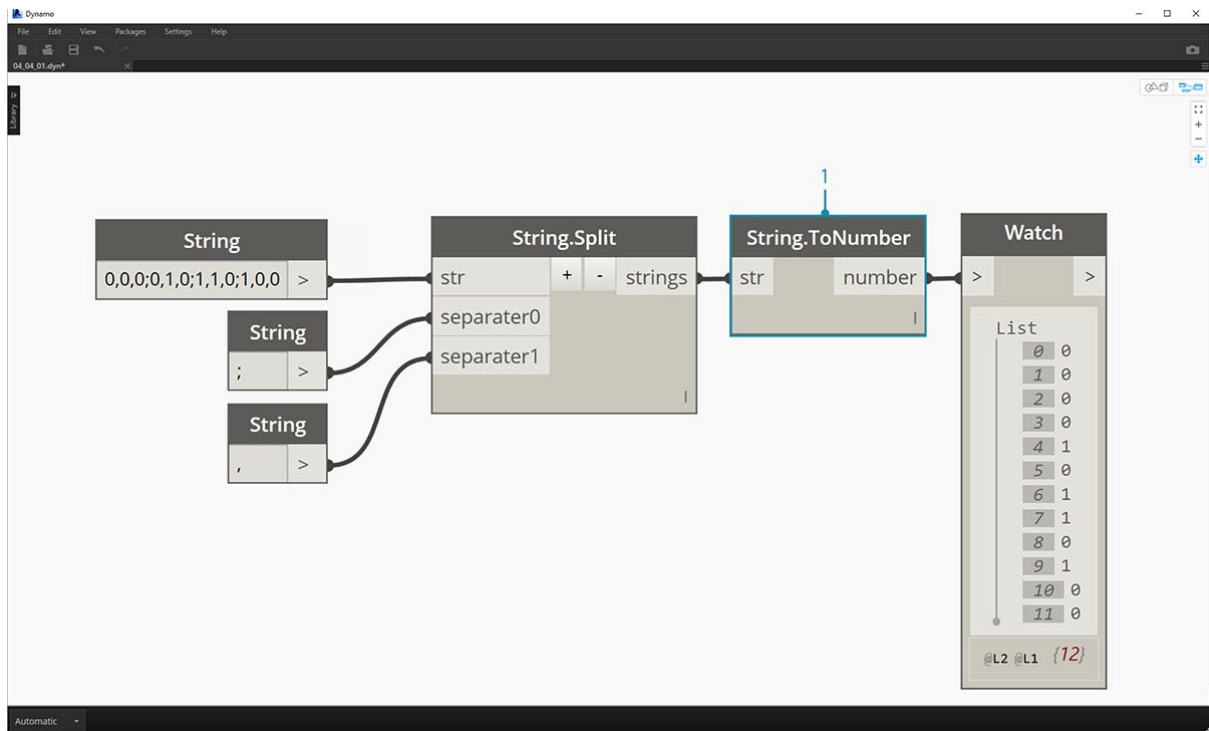


1. El separador ";" divide cada vértice del rectángulo. De este modo, se crea una lista con cuatro elementos para cada vértice.

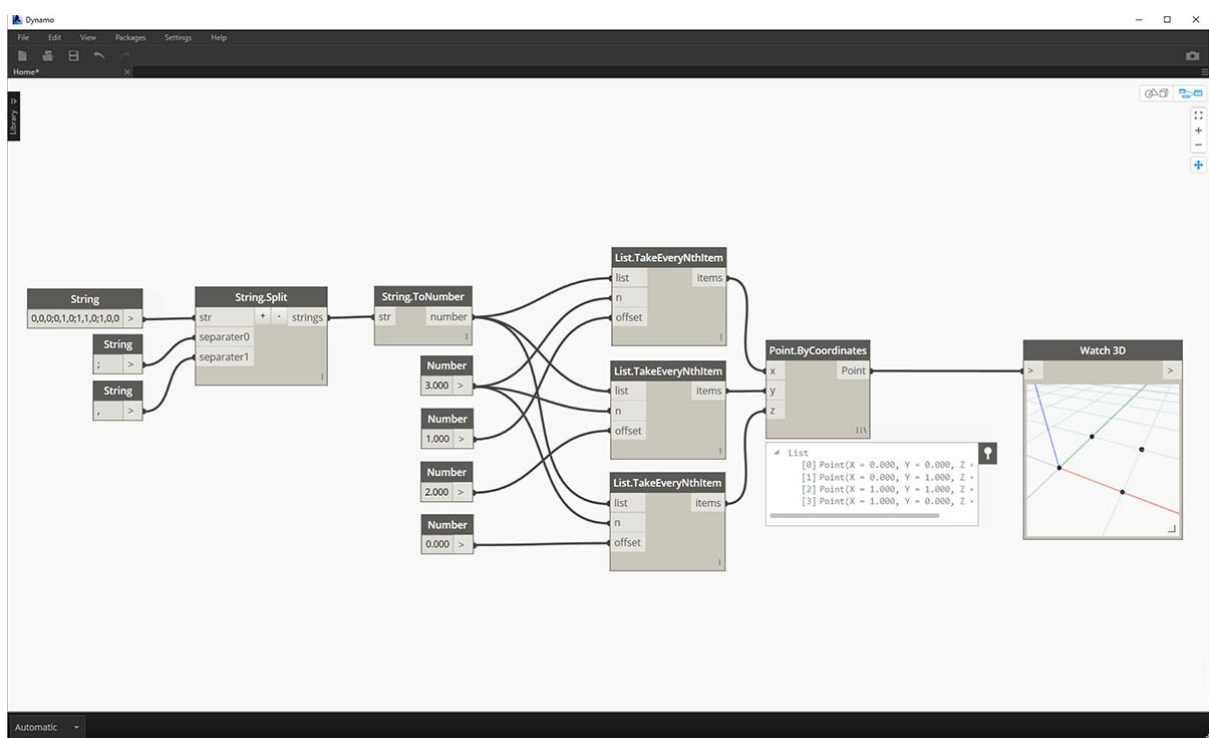


1. Al pulsar "+" en el centro del nodo, se crea un nuevo separador.
2. Añada una cadena "," al lienzo y conéctela a la nueva entrada de separador.
3. Nuestro resultado es ahora una lista de diez artículos. El nodo realiza la división primero en función de *separator0* y, a continuación, en función de *separator1*.

Aunque los elementos de la lista anterior pueden tener el aspecto de números, se siguen considerando cadenas individuales en Dynamo. Para crear puntos, su tipo de datos debe convertirse de una cadena a un número. Esto se realiza con el nodo *String.ToNumber*.



1. Este nodo es sencillo. Conecte los resultados de String.Split a la entrada. La salida no tiene un aspecto diferente, pero el tipo de datos es ahora un *número* en lugar de una *cadena*.

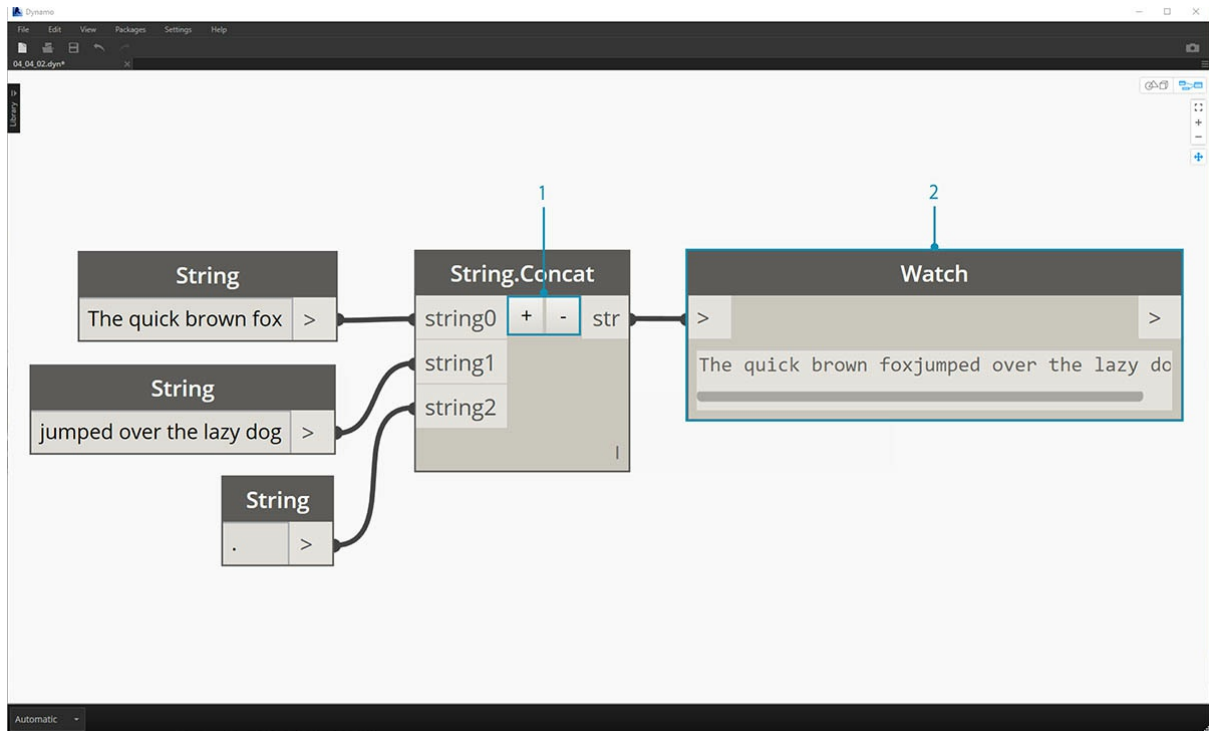


1. Con algunas operaciones básicas adicionales, ahora se ha dibujado un rectángulo en el origen en función de la entrada de cadena original.

Manipulación de cadenas

Dado que una cadena es un objeto de texto genérico, alberga una amplia gama de aplicaciones. Echemos un vistazo a algunas de las acciones principales de la categoría Core > String de Dynamo:

Este es un método para fusionar dos cadenas en orden. Toma cada cadena literal de una lista y crea una cadena combinada.

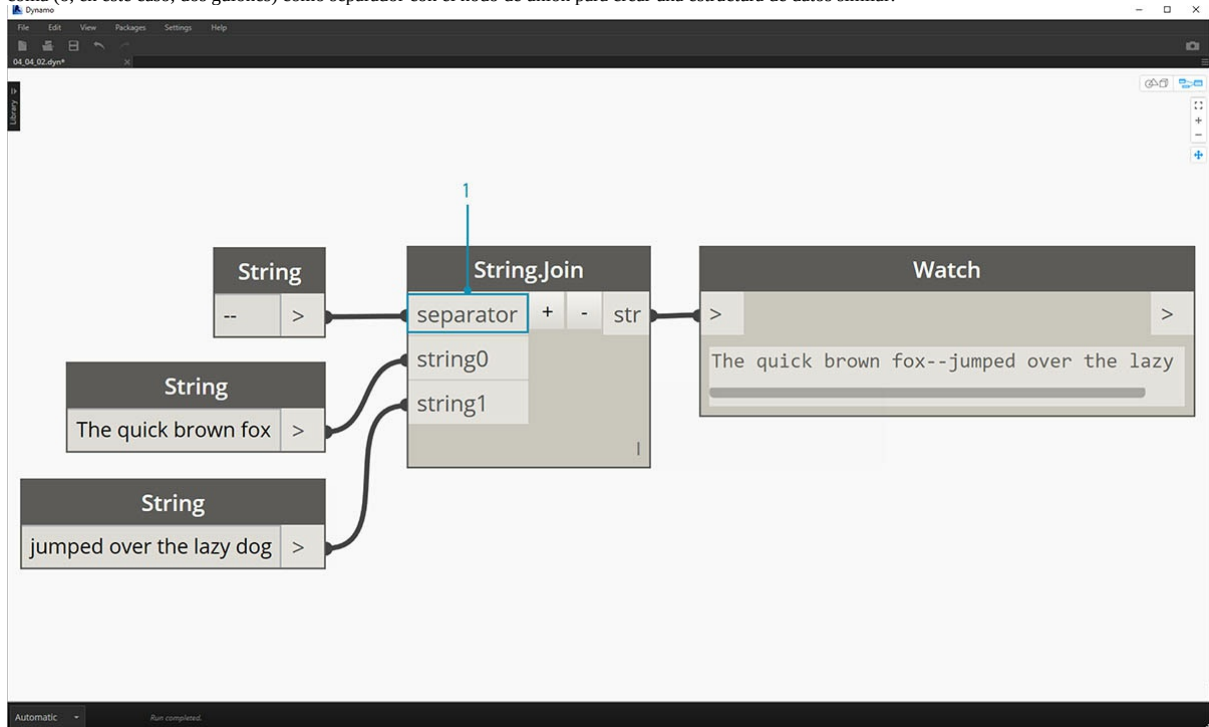


La imagen anterior representa la concatenación de tres cadenas:

1. Añada o sustraiga cadenas a la concatenación haciendo clic en los botones +/- del centro del nodo.
2. La salida es una cadena concatenada, incluidos espacios y signos de puntuación.

El método de unión es muy similar a la concatenación, excepto que tiene una capa añadida de signos de puntuación.

Si ha trabajado en Excel, es posible que haya encontrado un archivo CSV. Se trata de un archivo de valores separados por comas. Se puede utilizar una coma (o, en este caso, dos guiones) como separador con el nodo de unión para crear una estructura de datos similar:



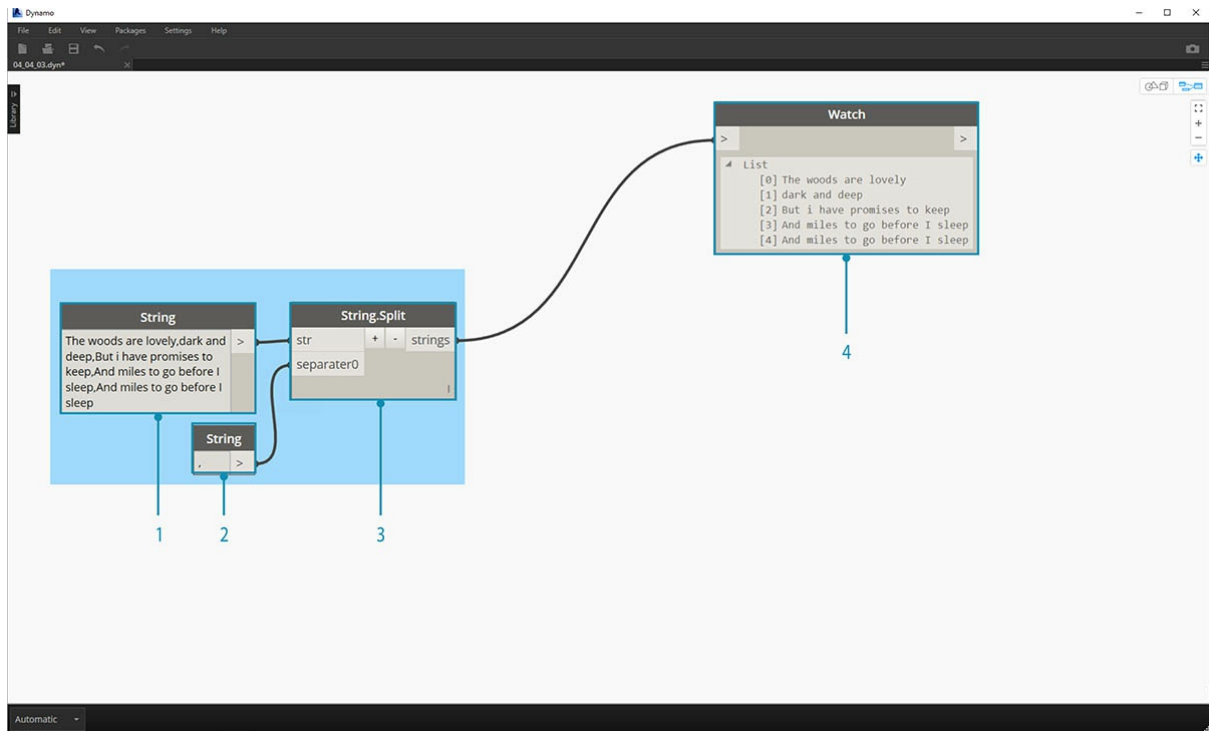
La imagen anterior representa la unión de dos cadenas:

1. La entrada separator permite crear una cadena que divide las cadenas unidas.

Trabajo con cadenas

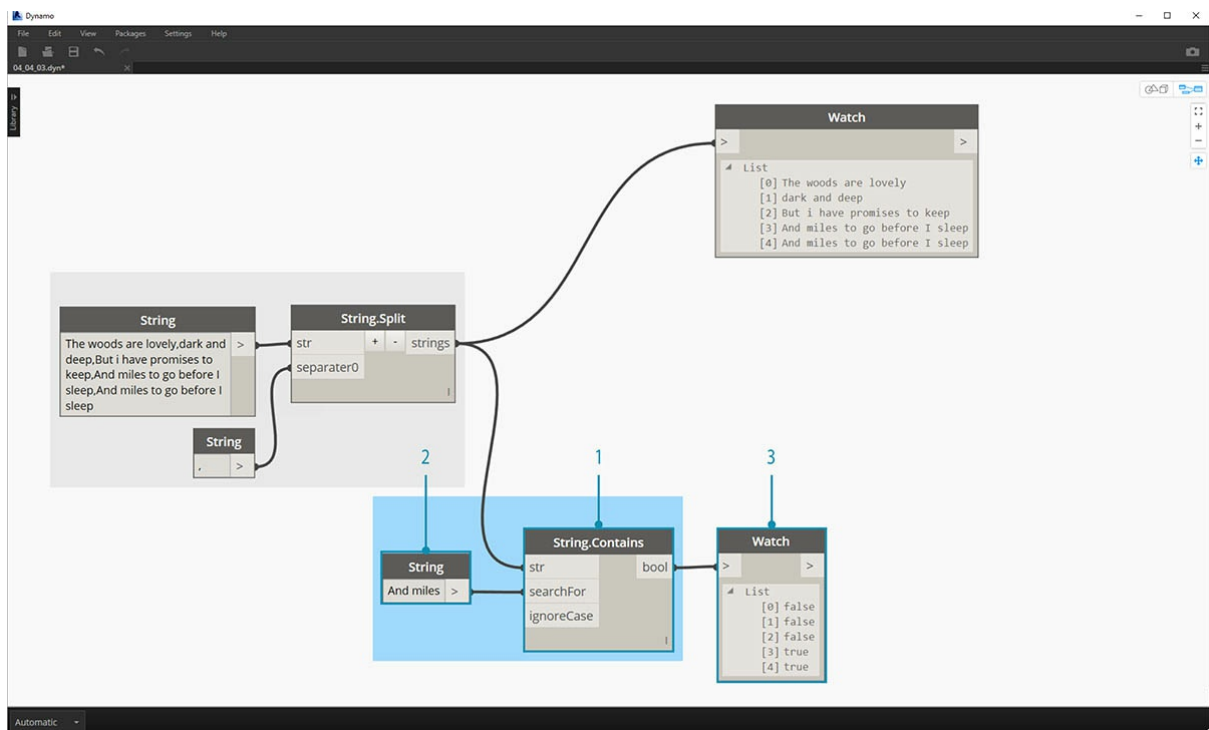
En este ejercicio, vamos a usar métodos de consulta y manipulación de cadenas para deconstruir la última estrofa del poema [Alto en el bosque en una noche de invierno](#) de Robert Frost. No es la aplicación más práctica, pero nos ayudará a comprender las acciones de cadenas conceptuales a medida que las

aplicamos a líneas legibles de ritmo y rima.



Comencemos con una división de cadenas básica de la estrofa. Primero, observamos que el formato de la escritura se basa en comas. Utilizaremos este formato para separar cada línea en elementos individuales.

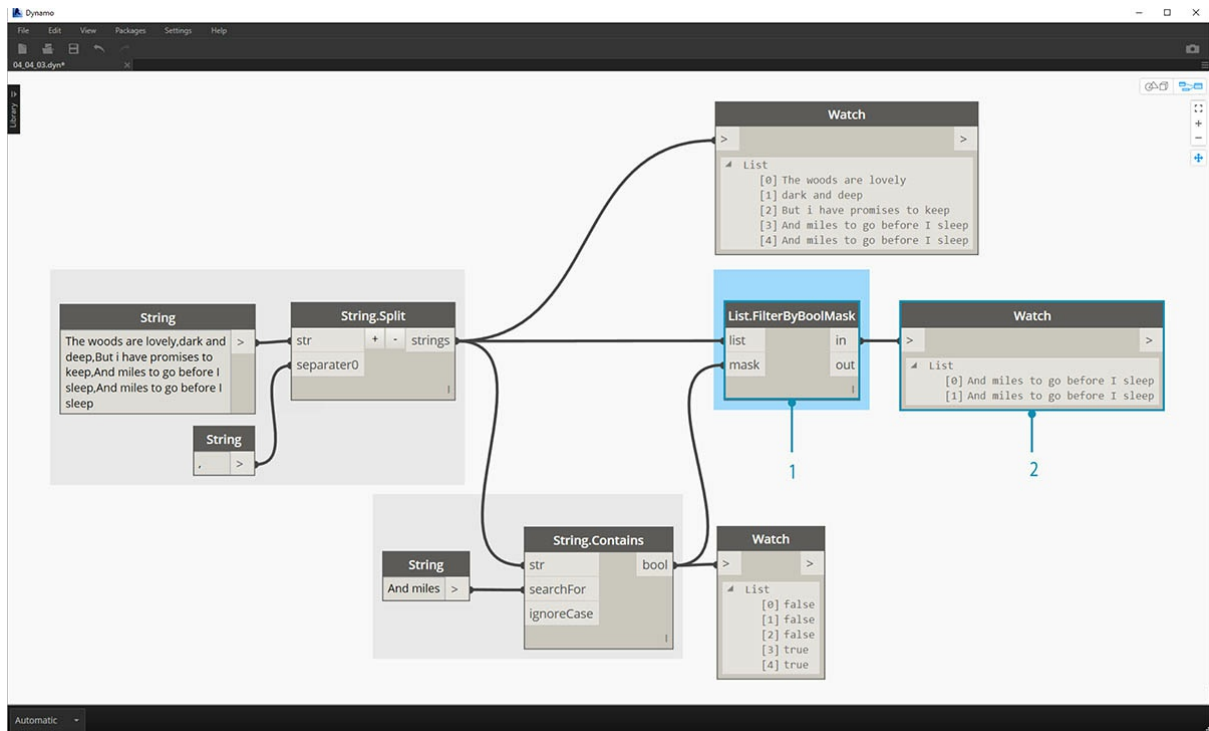
1. La cadena base se pega en un nodo String.
2. Se utiliza otro nodo String para indicar el separador. En este caso, usamos una coma.
3. Se añade un nodo String.Split al lienzo y se conecta a las dos cadenas.
4. La salida muestra ahora que hemos separado las líneas en elementos individuales.



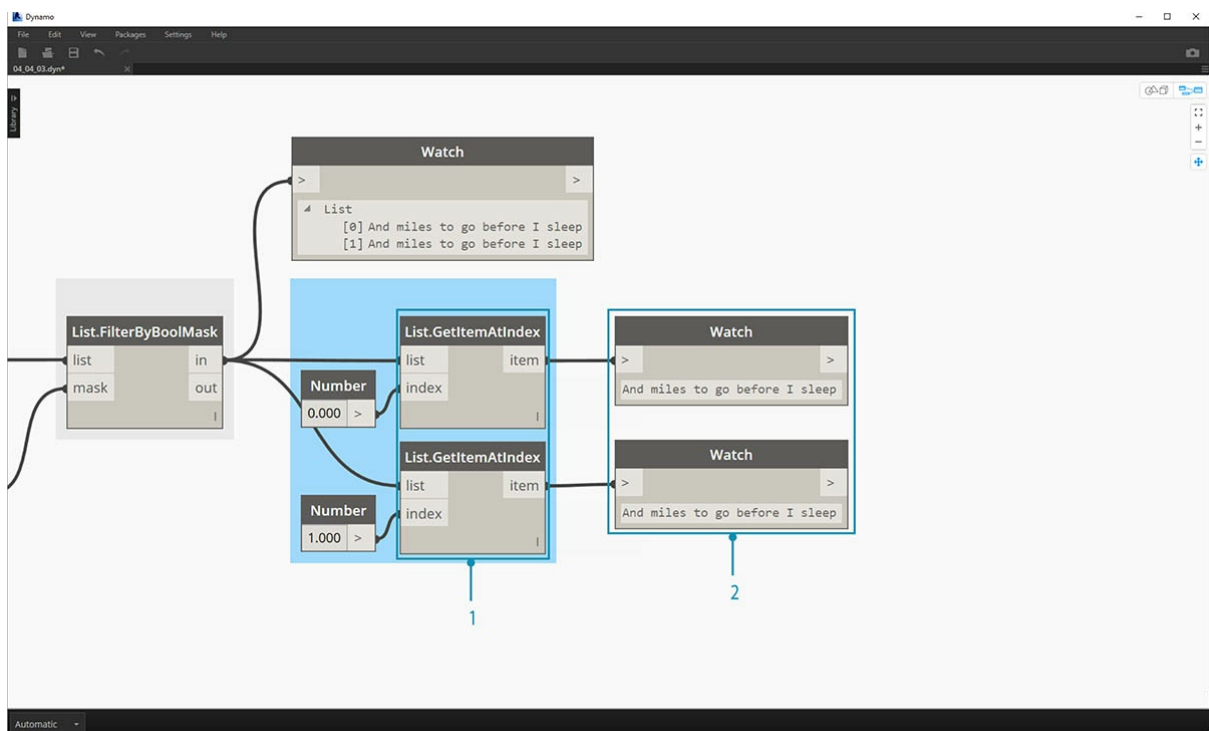
Ahora, vamos a la parte buena del poema: las dos últimas líneas. La estrofa original era un elemento de datos. En el primer paso, separamos estos datos en elementos individuales. Ahora debemos buscar el texto que deseamos encontrar. Y, aunque *podemos* hacerlo seleccionando los dos últimos elementos de la lista, si se tratara de un libro completo, no sería un método eficaz leerlo todo y aislar manualmente los elementos.

1. En lugar de realizar una búsqueda manual, se utiliza un nodo String.Contains para realizar una búsqueda de un conjunto de caracteres. Esto es similar a utilizar el comando "Buscar" en un procesador de texto. En este caso, obtenemos un valor de "verdadero" o "falso" si se encuentra esa subcadena en el elemento.

2. En la entrada "searchFor", definimos la subcadena que buscamos dentro de la estrofa. Vamos a usar un nodo String con el texto "And miles".
3. La salida nos proporciona una lista de resultados de verdadero y falso. Utilizaremos esta lógica booleana para filtrar los elementos en el siguiente paso.

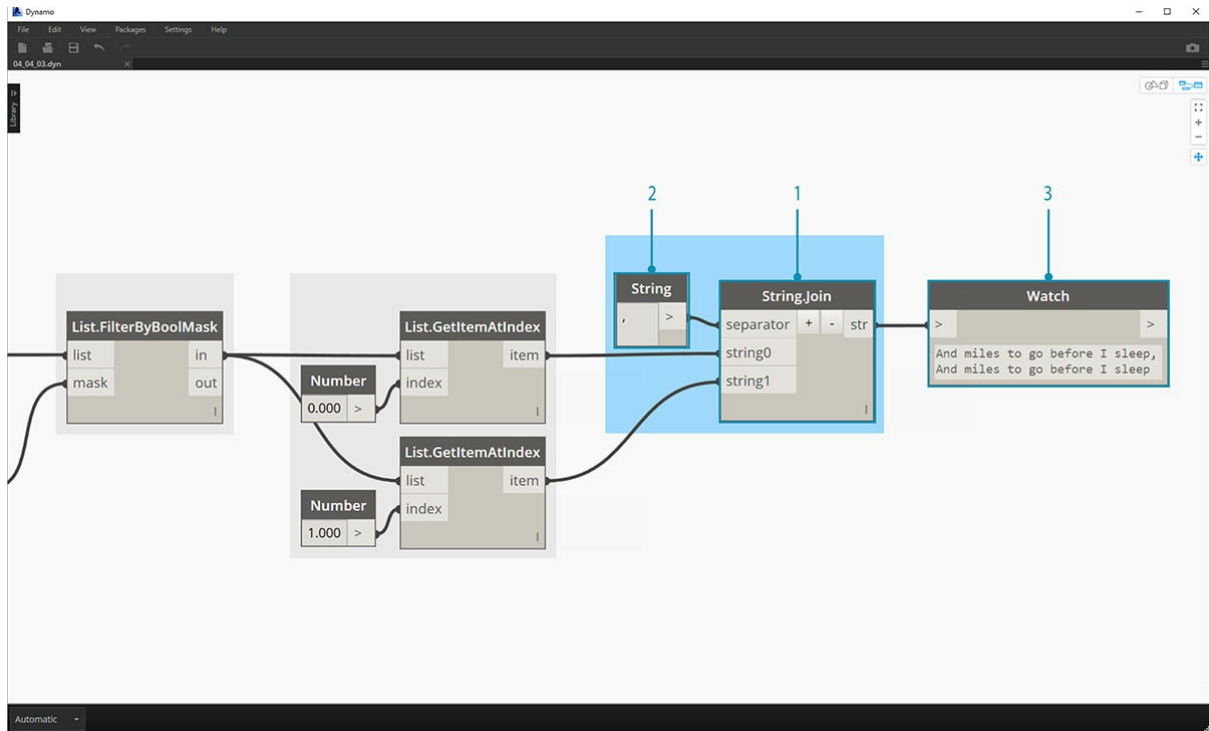


1. List.FilterByBoolMask es el nodo que debemos utilizar para seleccionar los resultados con valor verdadero o falso. La salida "in" devuelve las instrucciones con una entrada "mask" de "verdadero", mientras que la salida "out" devuelve las que tienen un valor de "falso".
2. Nuestra salida de "in" es como se esperaba, ya que devuelve las dos frases finales de la estrofa.



Ahora, vamos a dejar clara la repetición de la estrofa fusionando las dos líneas. Al visualizar el resultado del paso anterior, observamos que hay dos elementos en la lista:

1. Mediante dos nodos List.GetItemAtIndex, se pueden aislar los elementos con los valores 0 y 1 como entrada del índice.
2. La salida de cada nodo nos proporciona, por orden, las dos líneas finales.



Para fusionar estos dos elementos en uno, se utiliza el nodo String.Join:

1. Después de añadir el nodo String.Join, observamos que necesitamos un separador.
2. Para crear el separador, se añade un nodo String al lienzo y se escribe una coma.
3. La última salida ha fusionado los dos últimos elementos en uno.

Puede parecer muy laborioso aislar las dos últimas líneas y es cierto que, a menudo, las operaciones de cadena requieren trabajo inicial. Pero son escalables y se pueden aplicar a grandes conjuntos de datos con relativa facilidad. Si trabaja paramétricamente con hojas de cálculo e interoperabilidad, asegúrese de mantener en mente las operaciones de cadena.

Color

Color

El color es un excelente tipo de datos para crear imágenes convincentes, así como para renderizar la diferencia en la salida del programa visual. Cuando se trabaja con datos abstractos y números variables, a veces es difícil ver qué se está cambiando y en qué grado. Esta es una aplicación excelente para los colores.

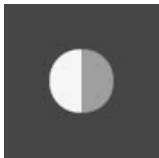
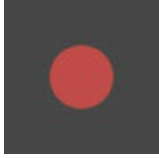
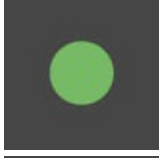
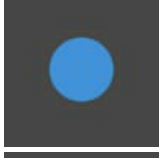
Creación de colores

Los colores de Dynamo se crean con entradas de ARGB. Estas corresponden a los canales alfa, rojo, verde y azul. El canal alfa representa la *transparencia* del color, mientras que los otros tres se utilizan como colores principales para generar todo el espectro de color sincronizado.

Icono	Nombre	Sintaxis	Entradas	Salidas
	Color ARGB	Color.ByARGB	A,R,G,B	color

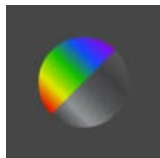
Consulta de los valores de color

Los colores de la tabla siguiente consultan las propiedades utilizadas para definir el color: alfa, rojo, verde y azul. Observe que el nodo Color.Components nos proporciona las cuatro salidas diferentes, lo que hace que este nodo sea preferible para consultar las propiedades de un color.

Icono	Nombre	Sintaxis	Entradas	Salidas
	Alfa	Color.Alpha	color	A
	Rojo	Color.Red	color	R
	Verde	Color.Green	color	G
	Azul	Color.Blue	color	B
	Componentes	Color.Components	color	A,R,G,B

Los colores de la tabla siguiente corresponden al **espacio de color HSB**. La división del color en el matiz, la saturación y el brillo es, probablemente, más intuitiva en la forma en que se interpreta el color: ¿Qué color debería ser? ¿Debería ser más o menos colorido? ¿Y debería ser más o menos claro u oscuro? Este es el desglose del matiz, la saturación y el brillo respectivamente.

Icono	Nombre de consulta	Sintaxis	Entradas	Salidas
	Matiz	Color.Hue	color	Matiz



Saturación

Color.Saturation color

Saturación



Brillo

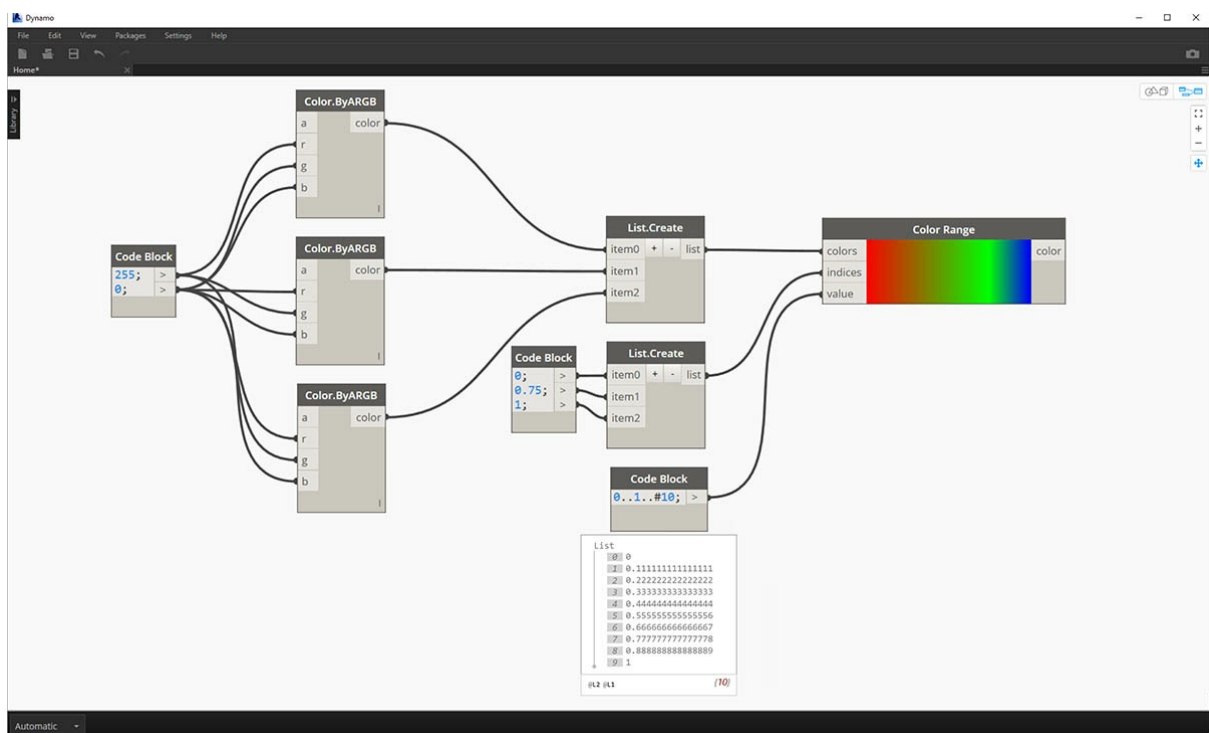
Color.Brightness color

Brillo

Rango de colores

El rango de colores es similar al nodo **Remap Range** de la sección 4.2: reasigna una lista de números a otro dominio. Sin embargo, en lugar de asignarla a un dominio de *número*, la asigna a un *degradado de color* basado en números de entrada que van del 0 al 1.

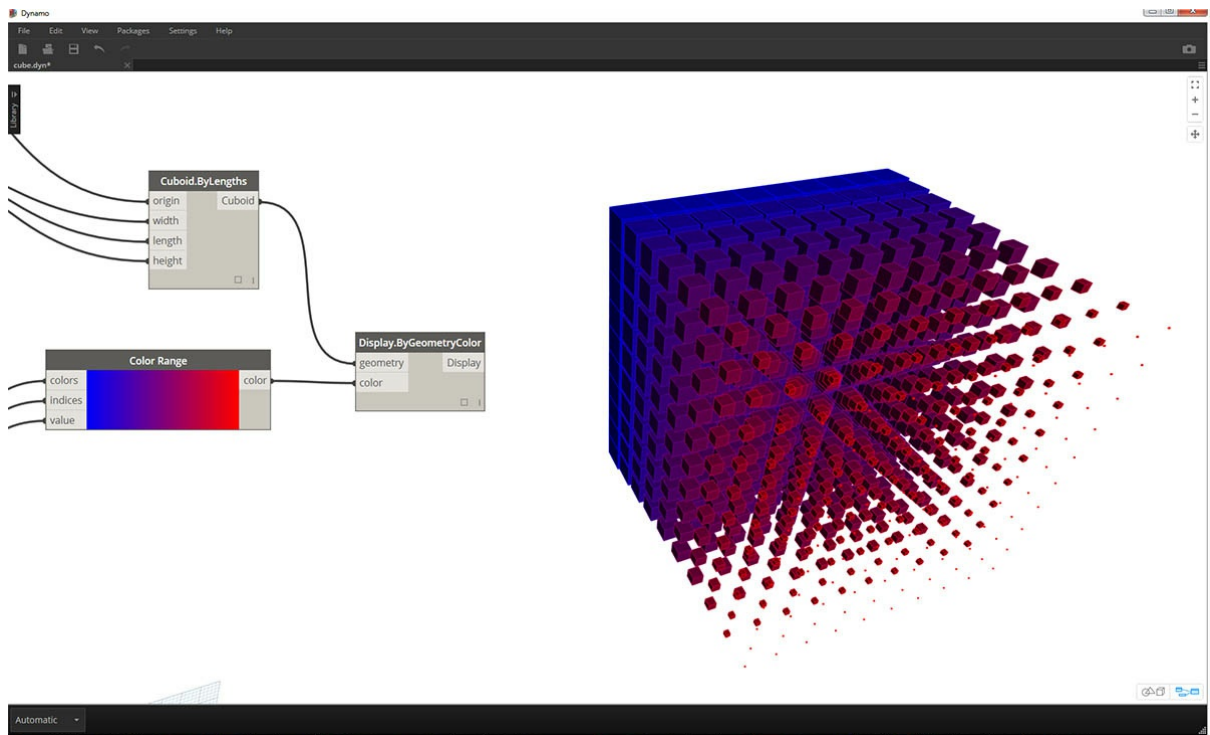
El nodo actual funciona bien, pero puede resultar un poco incómodo para que todo funcione la primera vez. La mejor forma de familiarizarse con el degradado de color es probarlo de forma interactiva. Hagamos un ejercicio rápido para revisar cómo configurar un degradado con colores de salida que corresponden a números.



1. **Definir tres colores:** a través de un nodo Code Block, defina *rojo*, *verde* y *azul* mediante la conexión de las combinaciones adecuadas de 0 y 255.
2. **Crear lista:** fusione los tres colores en una lista.
3. **Definir índices:** cree una lista para definir las posiciones de pinzamiento de cada color (de 0 a 1). Observe el valor de 0,75 para el verde. De este modo, se coloca el color verde a tres cuartos de camino en el degradado horizontal del control deslizante de rango de colores.
4. **Code Block:** valores de entrada (entre 0 y 1) para convertir en colores.

Vista previa de color

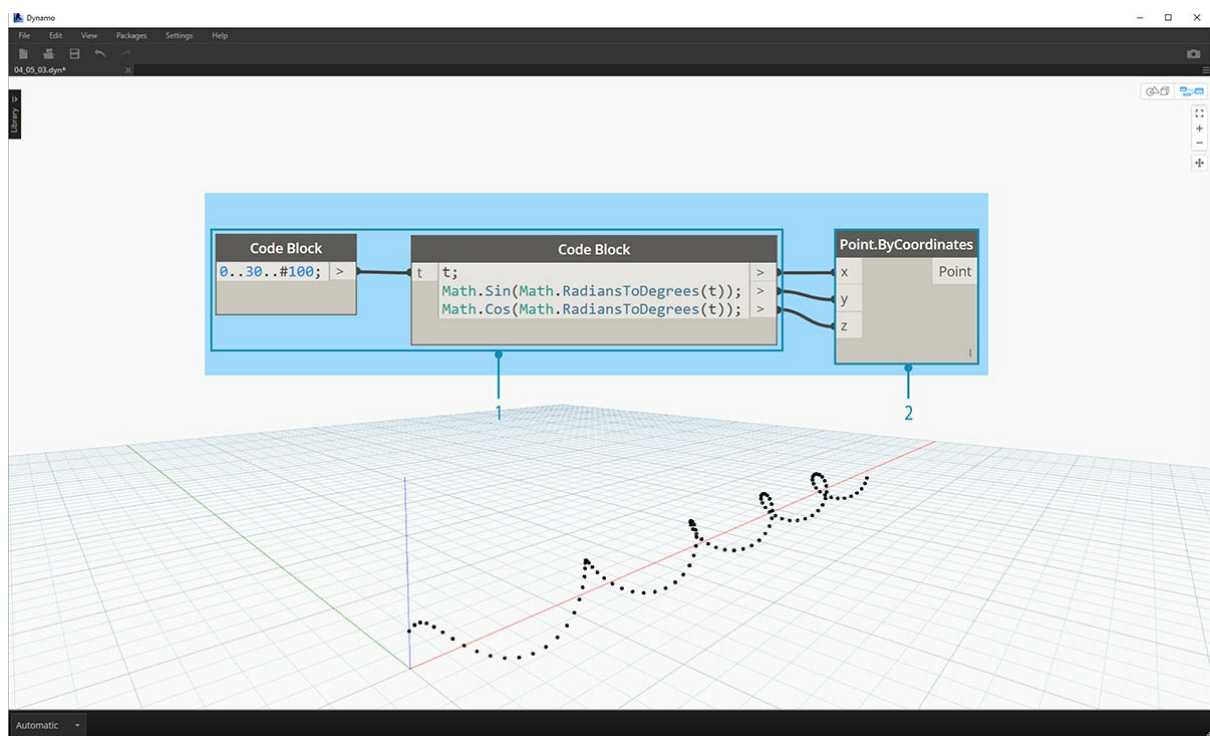
El nodo **Display.ByGeometry** nos permite colorear la geometría en la ventana gráfica de Dynamo. Esto resulta útil para separar diferentes tipos de geometría, demostrar un concepto paramétrico o definir una leyenda de análisis para la simulación. Las entradas son sencillas: geometría y color. Para crear un degradado como el de la imagen anterior, la entrada de color se conecta al nodo **Color Range**.



Ejercicio de color

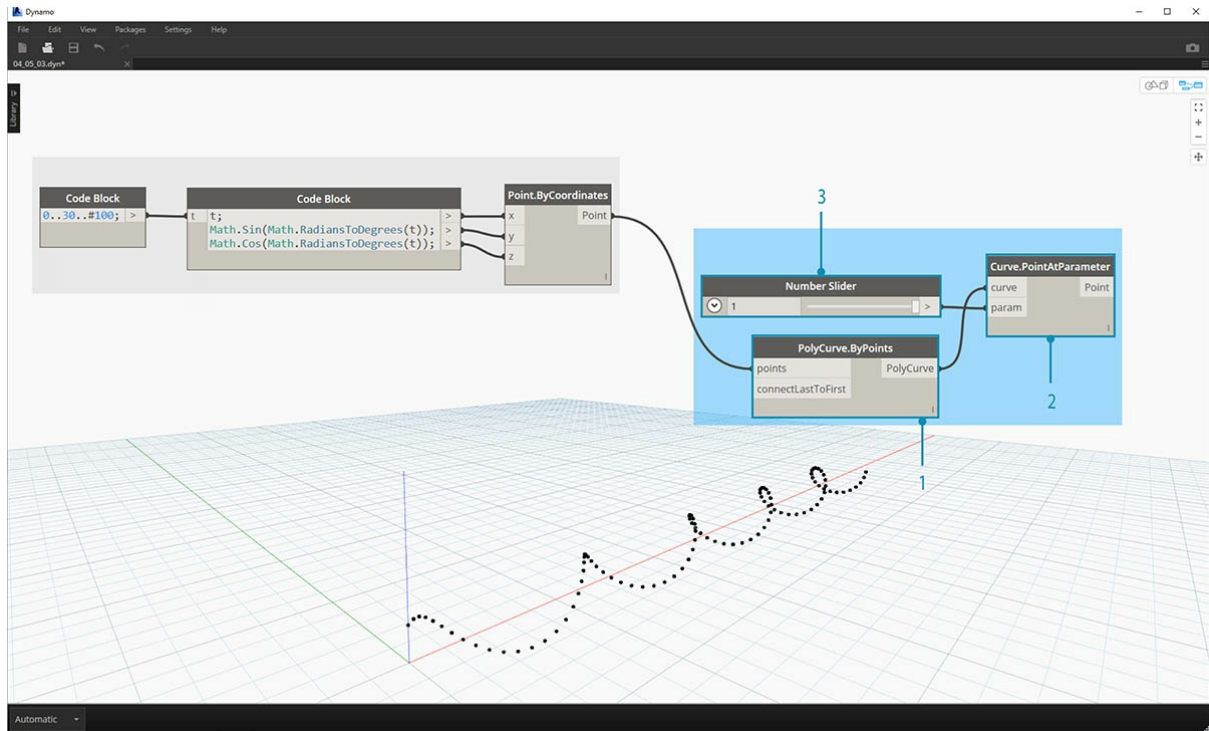
Descargue el archivo de ejemplo que acompaña a este ejercicio (haga clic con el botón derecho y seleccione "Guardar enlace como..."): [Building Blocks of Programs - Color.dyn](#). En el Apéndice se incluye una lista completa de los archivos de ejemplo.

Este ejercicio se centra en el control paramétrico del color en paralelo a la geometría. La geometría es una hélice básica que se define a continuación mediante el **bloque de código** (3.2.3). Esta es una forma rápida y sencilla de crear una función paramétrica y, como nuestro enfoque está en el color (en lugar de en la geometría), utilizamos el bloque de código para crear la hélice de forma eficaz sin sobrecargar el lienzo. Utilizaremos el bloque de código con más frecuencia a medida que el manual de introducción se adentre en materiales más avanzados.



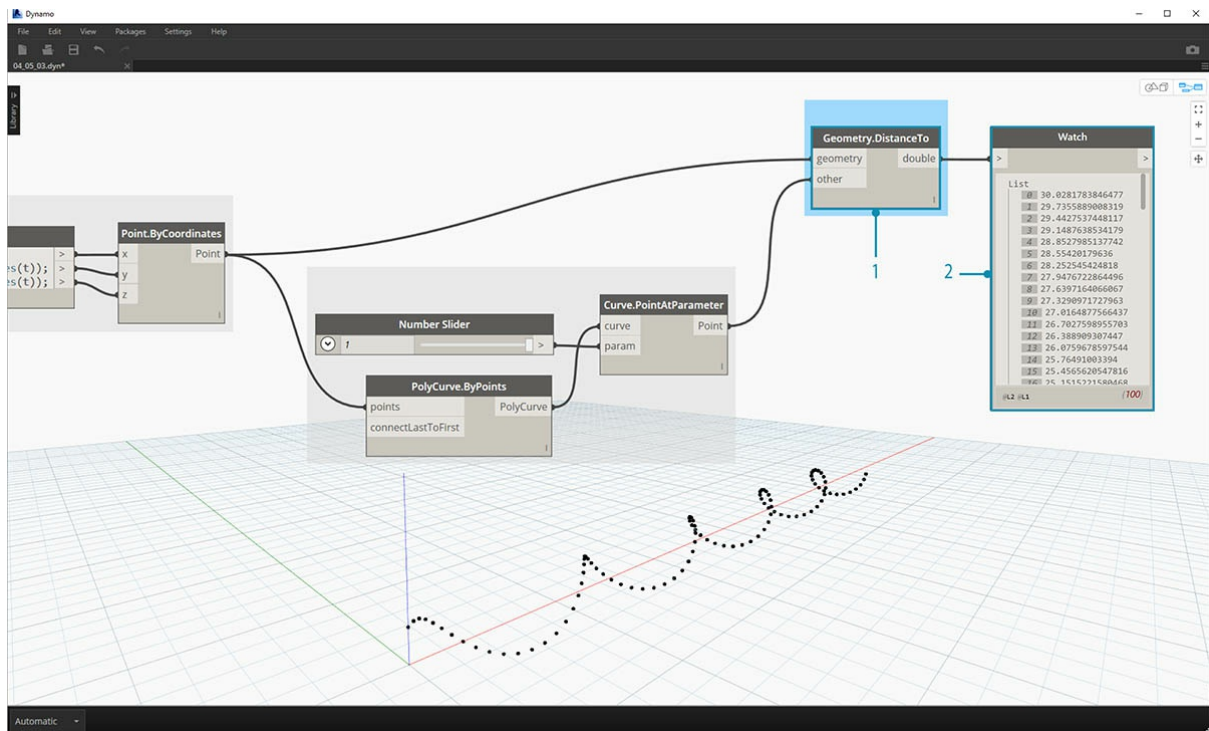
1. **Code Block:** defina los dos bloques de código con las fórmulas mostradas anteriormente. Este es un método paramétrico rápido para crear una espiral.
2. **Point.ByCoordinates:** conecte las tres salidas del bloque de código a las coordenadas del nodo.

Ahora se muestra una matriz de puntos que crean una hélice. El siguiente paso consiste en crear una curva que atraviese los puntos para que podamos visualizar la hélice.



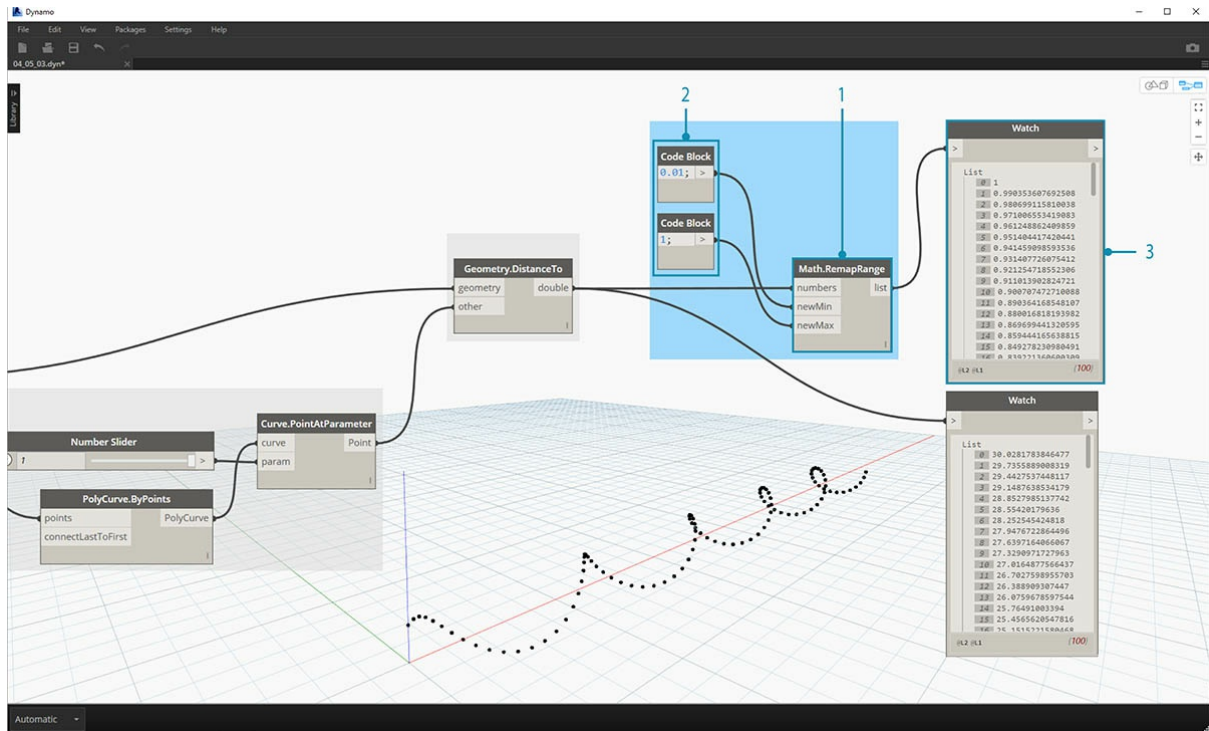
1. **PolyCurve.ByPoints:** conecte la salida *Point.ByCoordinates* en la entrada *points* del nodo. Se obtiene una curva helicoidal.
2. **Curve.PointAtParameter:** conecte la salida de *PolyCurve.ByPoints* a la entrada *curve*. La finalidad de este paso es crear un punto atractor paramétrico que se deslice a lo largo de la curva. Dado que la curva está evaluando un punto en el parámetro, necesitaremos introducir un valor en *param* entre 0 y 1.
3. **Number Slider:** después de añadirlo al lienzo, cambie el valor *mínimo* a 0,0, el valor *máximo* a 1,0 y el valor de *paso* a 0,01. Conecte la salida del control deslizante a la entrada *param* de *Curve.PointAtParameter*. Ahora se muestra un punto a lo largo de la longitud de la hélice representado por un porcentaje del control deslizante (0 en el punto inicial, 1 en el punto final).

Con el punto de referencia creado, ahora comparamos la distancia desde el punto de referencia a los puntos originales que definen la hélice. Este valor de distancia controlará la geometría y el color.



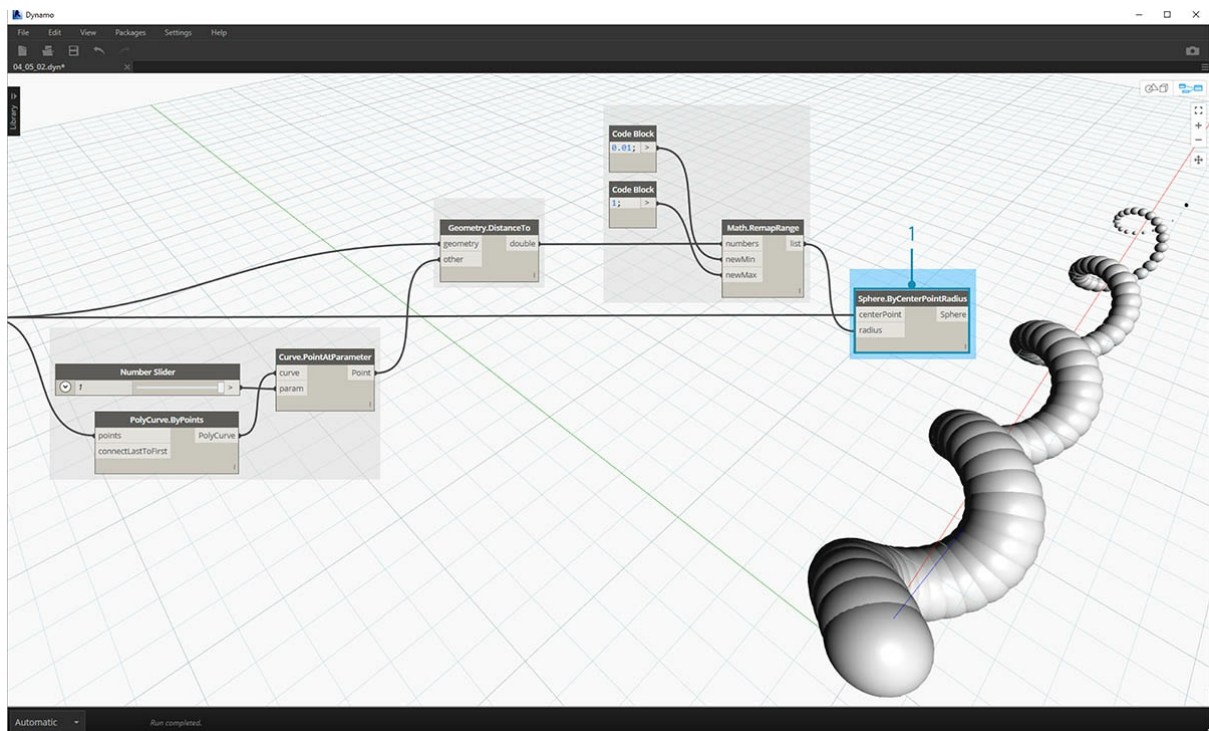
1. **Geometry.DistanceTo:** conecte la salida *Curve.PointAtParameter* a la entrada. Conecte *Point.ByCoordinates* a la entrada **geometry*.
2. **Watch:** la salida resultante muestra una lista de las distancias desde cada punto helicoidal hasta el punto de referencia a lo largo de la curva.

El siguiente paso consiste en controlar los parámetros con la lista de distancias entre los puntos helicoidales y el punto de referencia. Utilizamos estos valores de distancia para definir los radios de una serie de esferas a lo largo de la curva. Para mantener las esferas en un tamaño adecuado, es necesario reasignar los valores de la distancia.

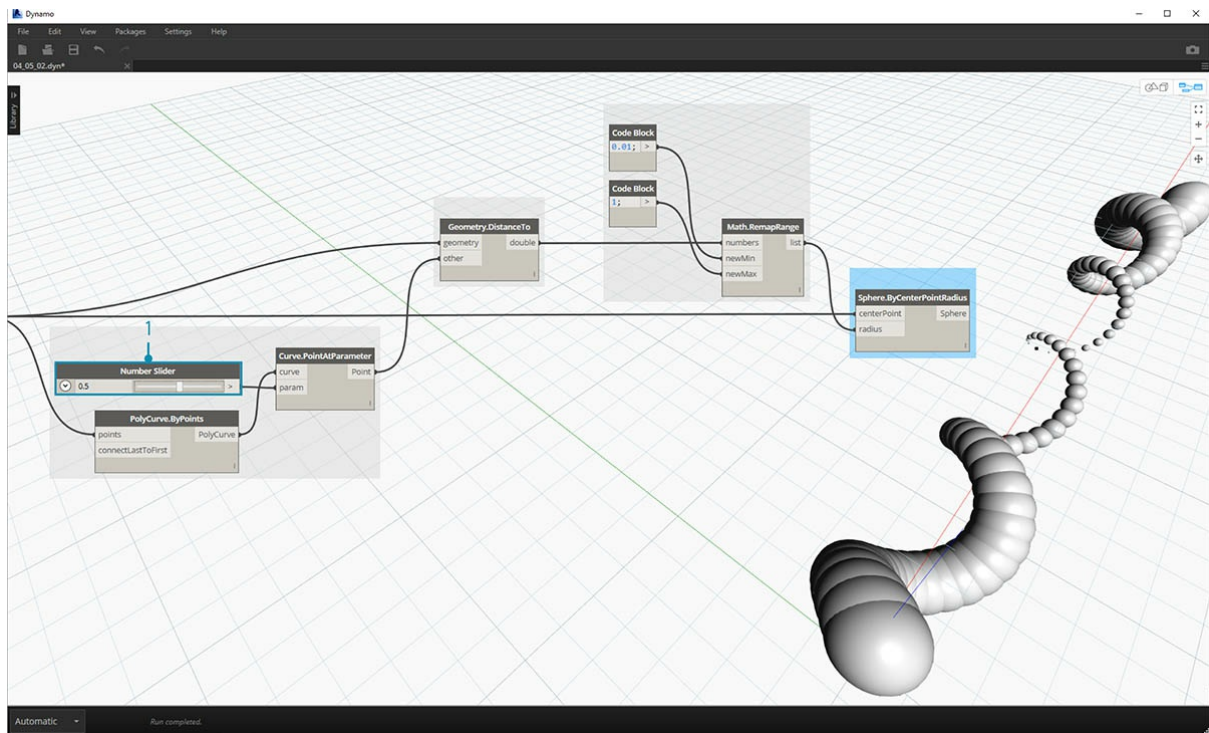


1. **Math.RemapRange:** conecte la salida *Geometry.DistanceTo* a la entrada *numbers*.
2. **Code Block:** conecte un bloque de código con un valor de *0,01* a la entrada *newMin* y un bloque de código con un valor de *1* a la entrada *newMax*.
3. **Watch:** conecte la salida *Math.RemapRange* a un nodo y la salida *Geometry.DistanceTo* a otro. Compare los resultados.

Este paso ha reasignado la lista de distancias para que tenga un rango más pequeño. Podemos editar los valores de *newMin* y *newMax* según consideremos adecuado. Los valores se reasignarán y tendrán la misma *relación de distribución* en todo el dominio.

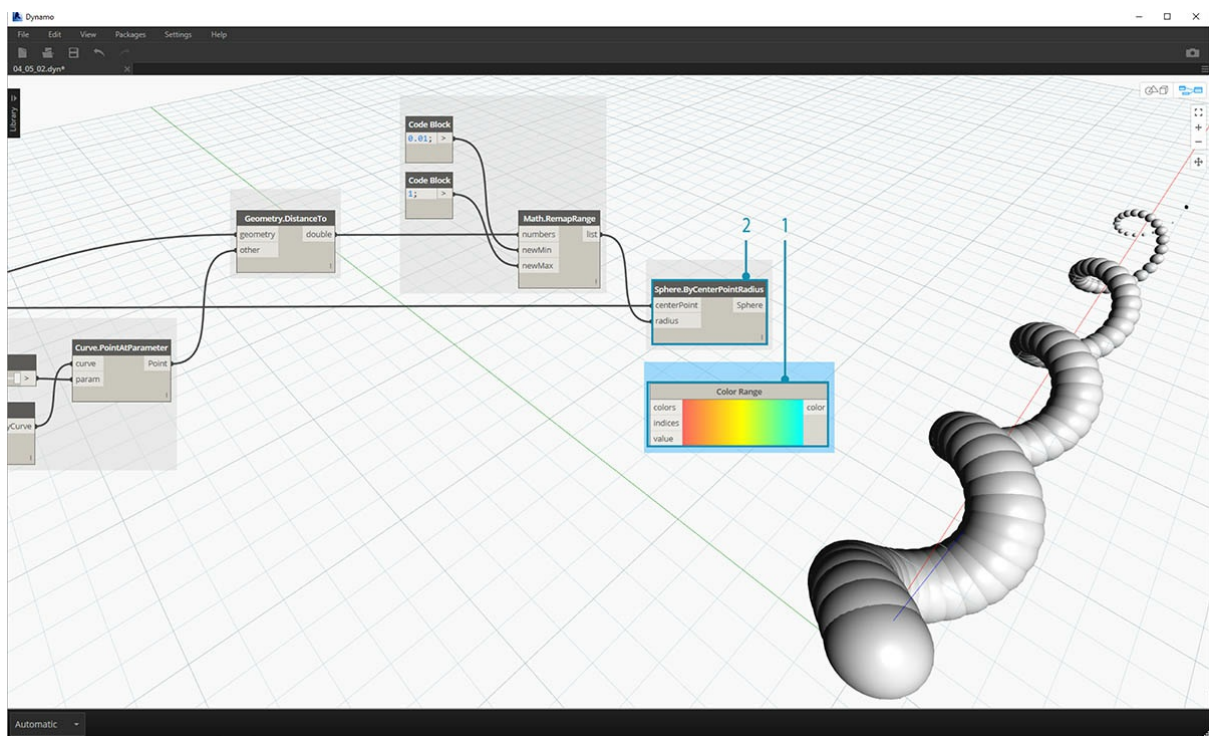


1. **Sphere.ByCenterPointRadius:** conecte la salida *Math.RemapRange* a la entrada *radius* y la salida *Point.ByCoordinates* original a la entrada *centerPoint*.

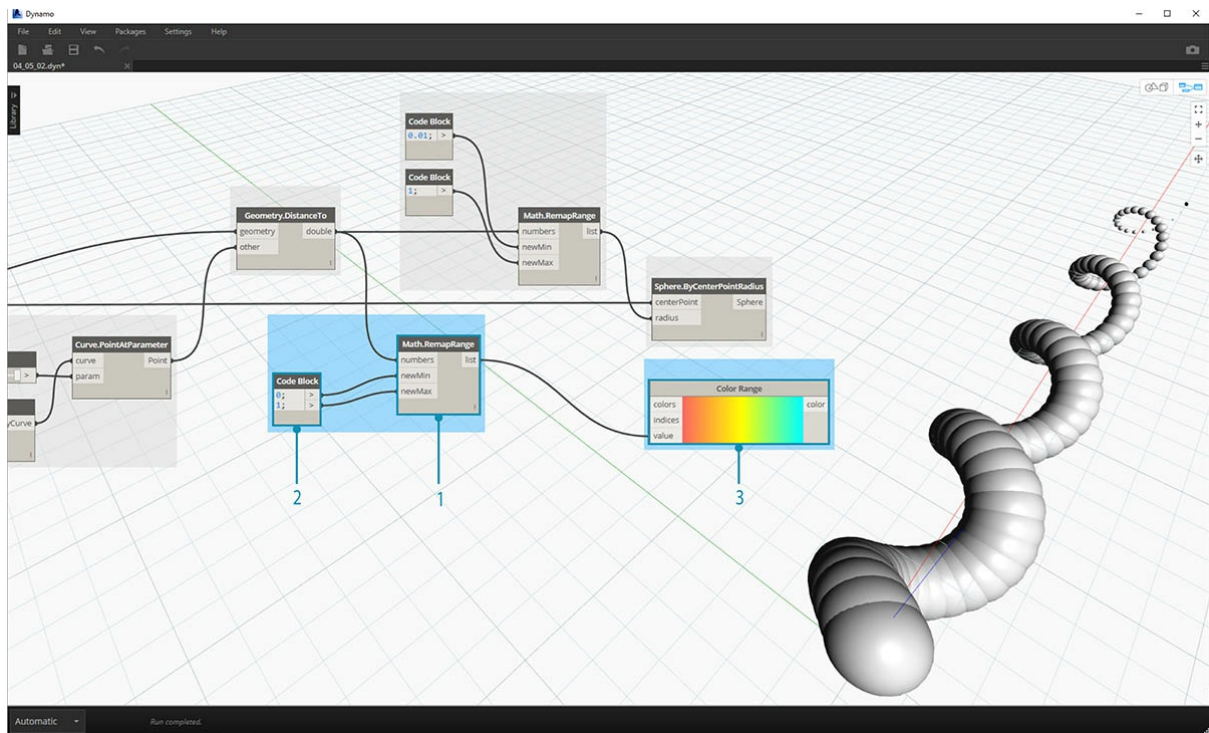


1. **Number Slider:** cambie el valor del control deslizante de número y observe como el tamaño de las esferas se actualiza. Ahora tenemos una guía paramétrica.

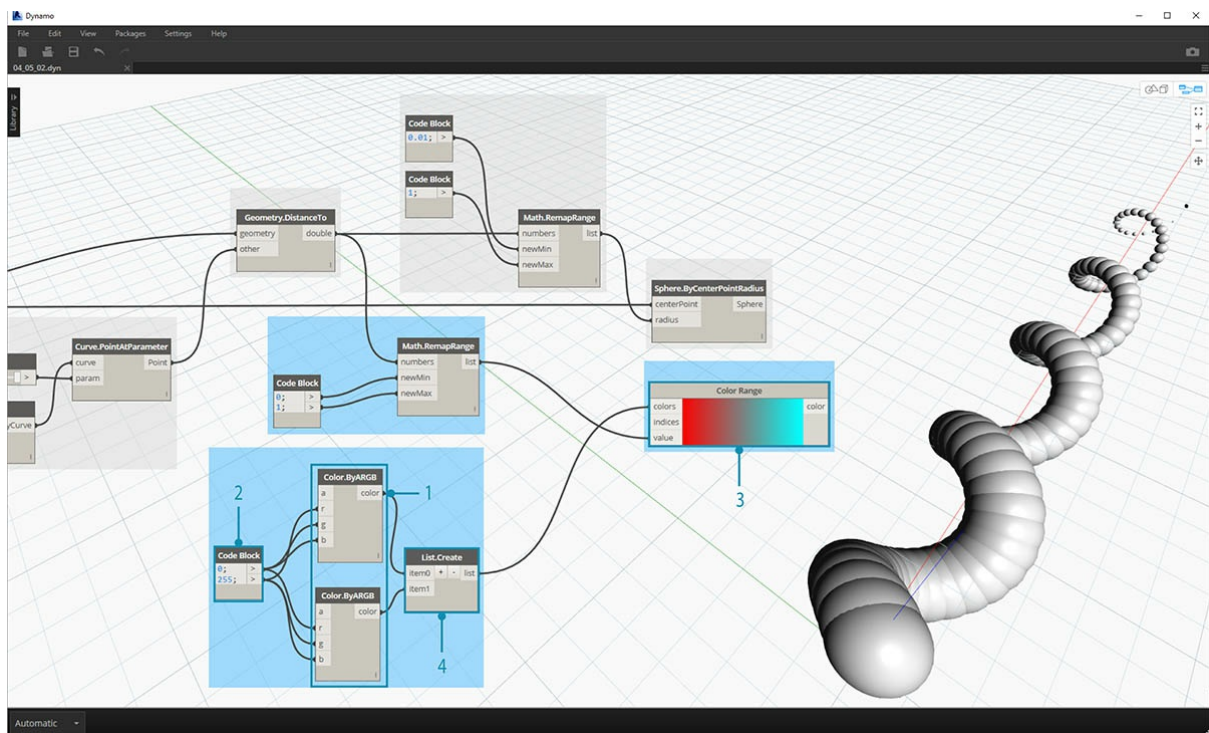
El tamaño de las esferas muestra la matriz paramétrica definida por un punto de referencia a lo largo de la curva. Usaremos el mismo concepto para que el radio de la esfera controle su color.



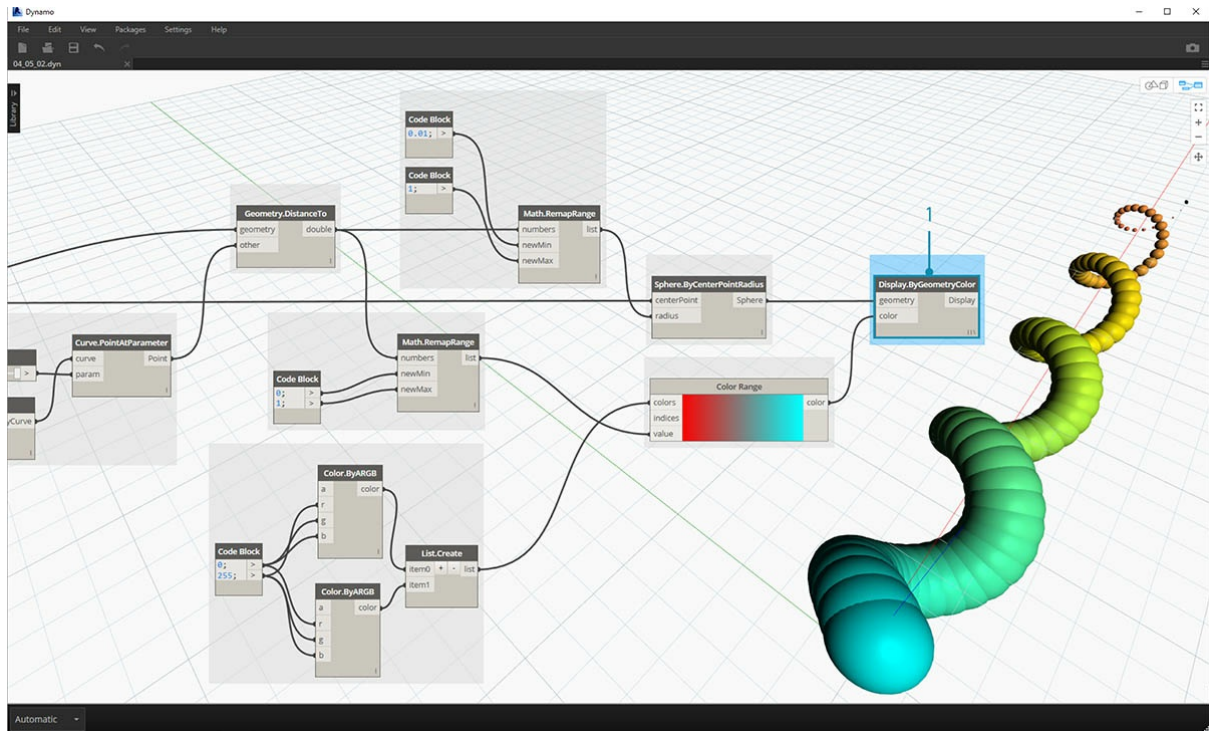
1. **Color Range:** añádalo al lienzo. Al pasar el cursor sobre la entrada *value*, observamos que los números solicitados se encuentran entre 0 y 1. Es necesario volver a asignar los números de la salida *Geometry.DistanceTo* para que sean compatibles con este dominio.
2. **Sphere.ByCenterPointRadius:** por el momento, desactivemos la vista preliminar en este nodo (clic con el botón derecho > Vista preliminar).



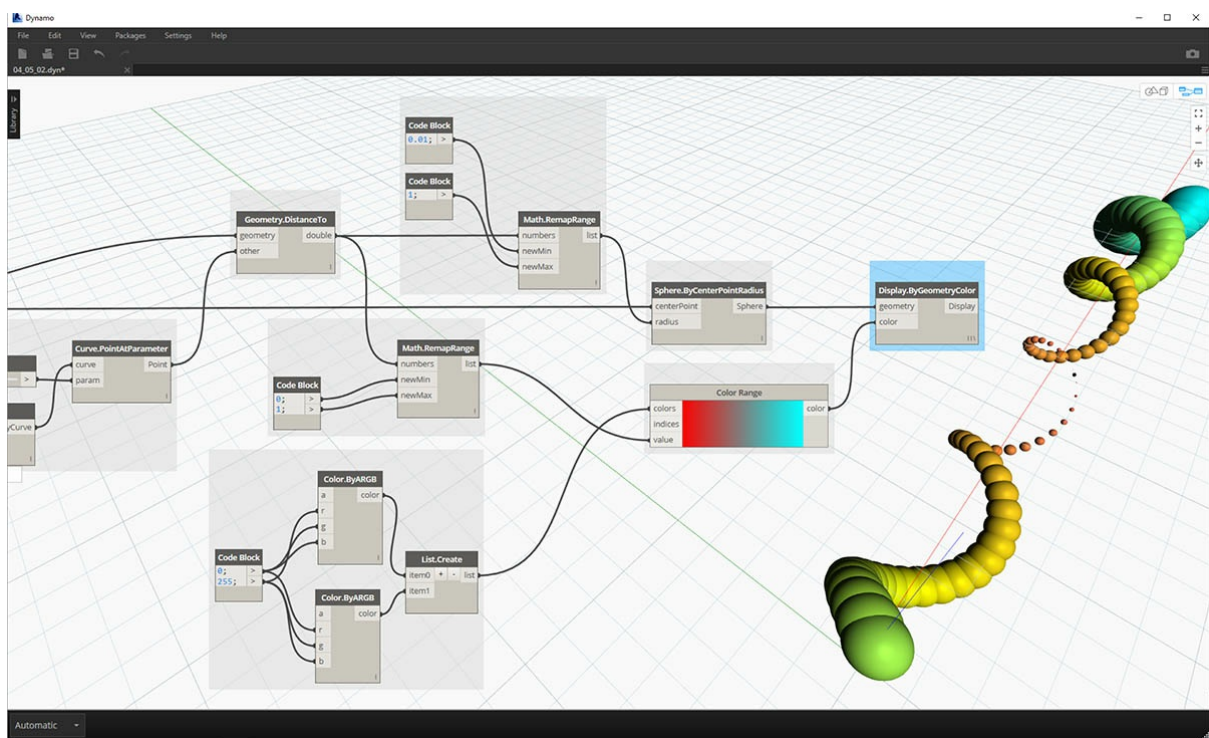
1. **Math.RemapRange:** este proceso debería resultarles familiar. Conecte la salida de *Geometry.DistanceTo* a la entrada *numbers*.
2. **Code Block:** como hicimos en un paso anterior, cree un valor de 0 para la entrada *newMin* y un valor de 1 para la entrada *newMax*. Observe que en este caso podemos definir dos salidas de un bloque de código.
3. **Color Range:** conecte la salida de *Math.RemapRange* a la entrada *value*.



1. **Color.ByARGB:** esto es lo que haremos para crear dos colores. Aunque este proceso puede parecer incómodo, es el mismo que el utilizado con los colores RGB en otro software, solo que usamos programación visual para hacerlo.
2. **Code Block:** cree dos valores de 0 y 255. Conecte las dos salidas a las dos entradas de *Color.ByARGB* de acuerdo con la imagen anterior (o cree sus dos colores favoritos).
3. **Color Range:** la entrada *colors* requiere una lista de colores. Se debe crear esta lista a partir de los dos colores creados en el paso anterior.
4. **List.Create:** combine los dos colores en una lista. Conecte la salida a la entrada *colors* de *Color Range*.



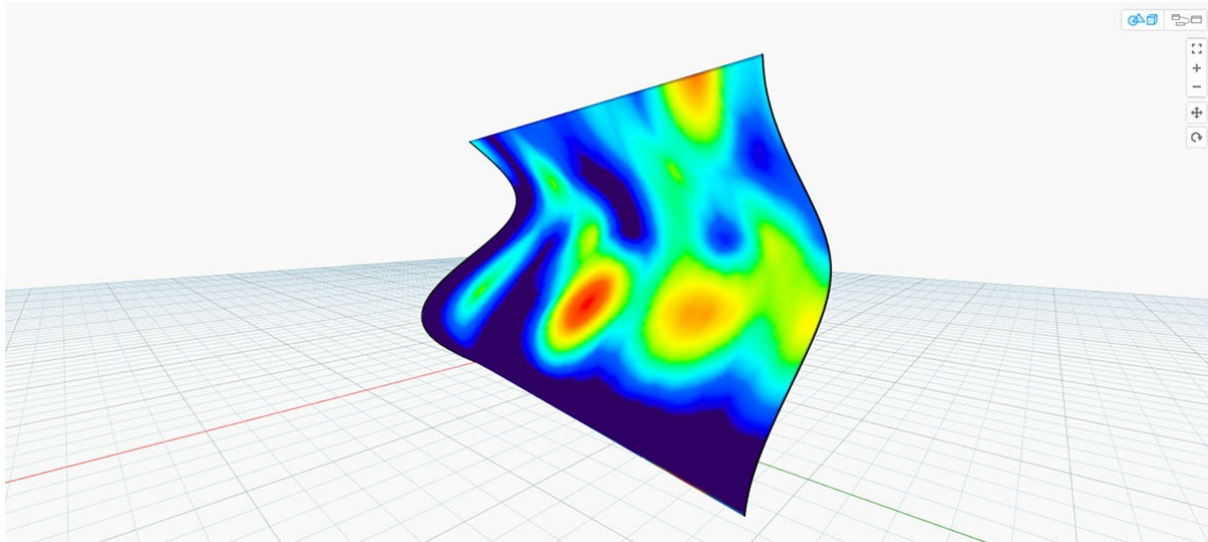
1. **Display.ByGeometryColor:** conecte *Sphere.ByCenterPointRadius* a la entrada *geometry* y *Color Range* a la entrada *color*. Ahora tenemos un degradado suave en el dominio de la curva.



Si se cambia el valor del control deslizante de número que apareció anteriormente, los colores y los tamaños se actualizan. Los colores y el tamaño del radio están directamente relacionados en este caso; ahora tenemos un vínculo visual entre dos parámetros.

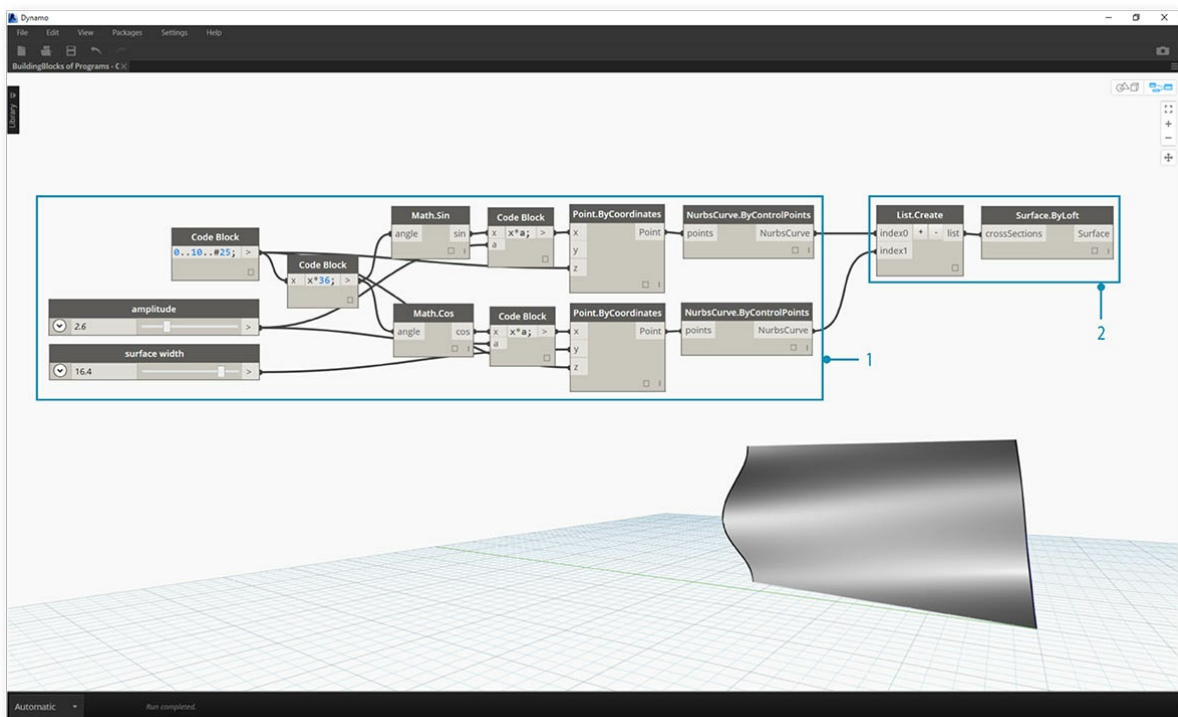
Color en superficies

El nodo **Display.BySurfaceColors** nos permite asignar datos en una superficie mediante los colores. Esta función ofrece posibilidades interesantes para visualizar los datos obtenidos a través de análisis independientes como análisis solares, de energía o de proximidad. La aplicación de color a una superficie en Dynamo es similar a la aplicación de una textura a un material en otros entornos de CAD. Vamos a demostrar cómo utilizar esta herramienta en el siguiente ejercicio breve.



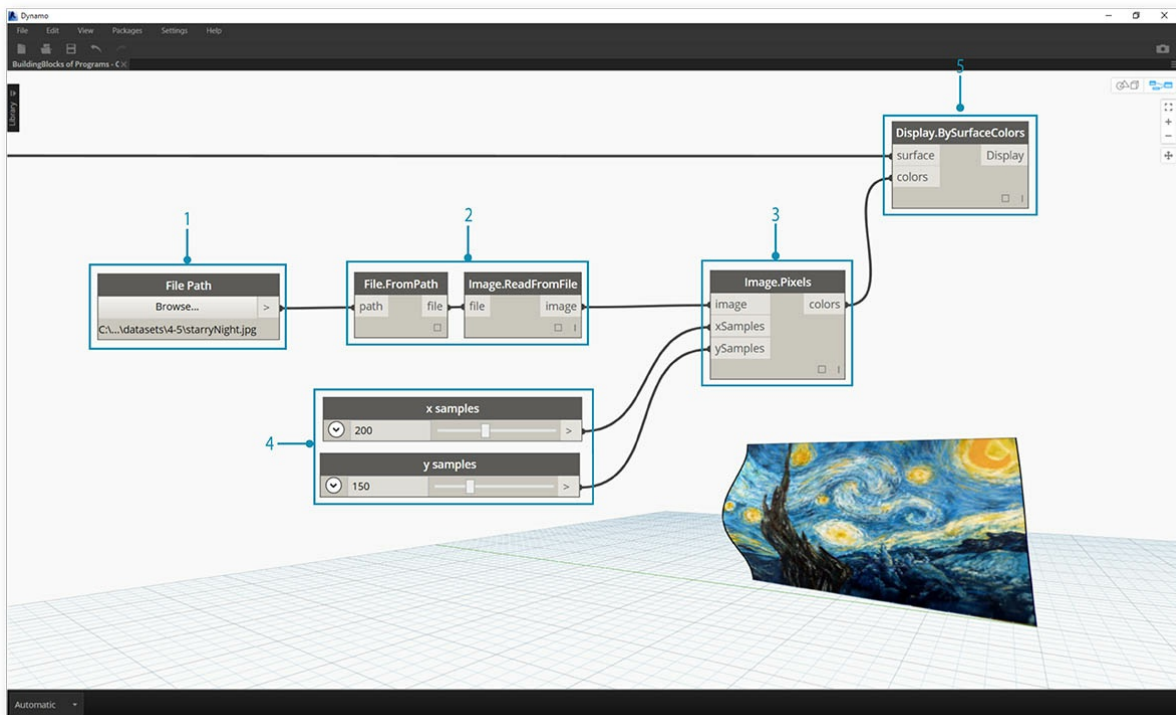
Ejercicio de color en superficies

Descargue el archivo de ejemplo que acompaña a este ejercicio (haga clic con el botón derecho y seleccione "Guardar enlace como..."): [Building Blocks of Programs - ColorOnSurface.zip](#). En el Apéndice se incluye una lista completa de los archivos de ejemplo.

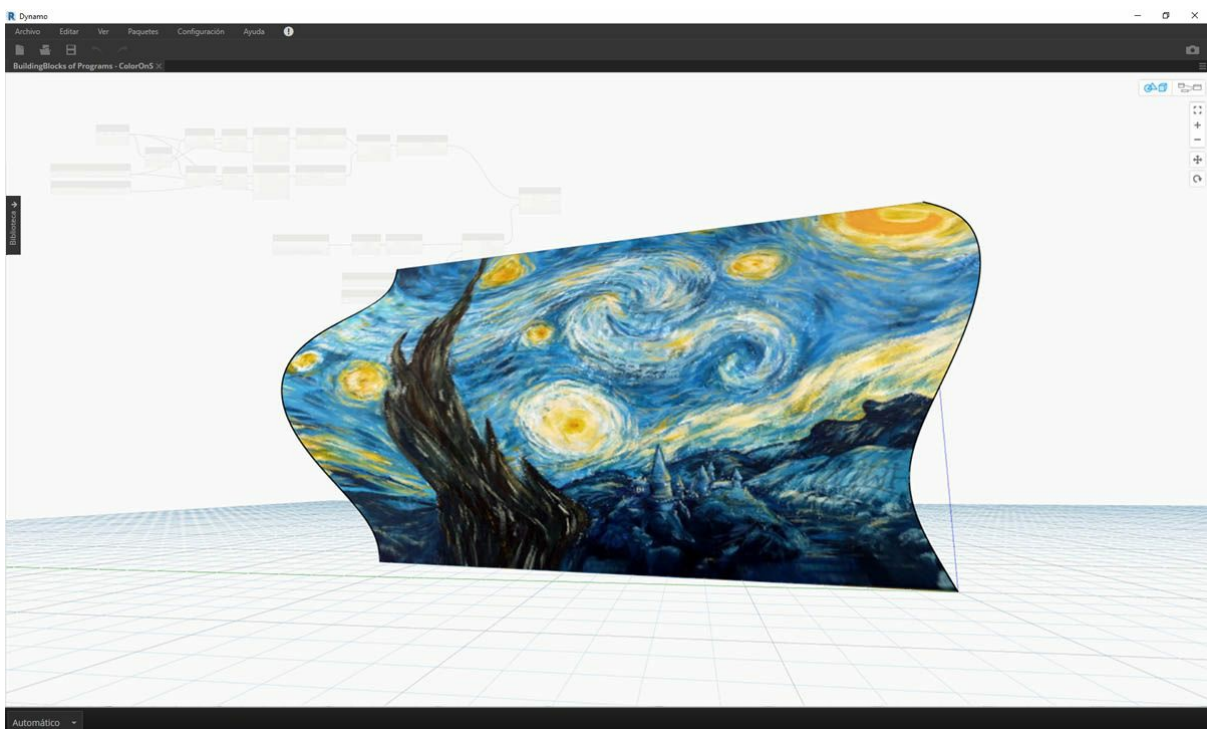


En primer lugar, es necesario crear una superficie o hacer referencia a ella para utilizarla como entrada en el nodo **Display.BySurfaceColors**. En este ejemplo se realiza la elevación entre una curva seno y coseno.

1. Este **grupo** de nodos crea puntos a lo largo del eje Z y los desplaza en función de las funciones de seno y coseno. Las dos listas de puntos se utilizan a continuación para generar curvas NURBS.
2. **Surface.ByLoft**: genere una superficie interpolada entre las curvas NURBS de la lista.



1. **File Path:** seleccione un archivo de imagen para muestrear datos de píxeles.
2. Utilice **File.FromPath** para convertir la ruta de archivo en un archivo y, a continuación, páselo a **Image.ReadFromFile** para obtener una imagen para el muestreo.
3. **Image.Pixels:** introduzca una imagen y proporcione un valor de muestra para utilizarlo en las cotas X e Y de la imagen.
4. **Control deslizante:** proporcione valores de muestra para **Image.Pixels**.
5. **Display.BySurfaceColors:** asigne los valores de matriz de color de la superficie a lo largo de X e Y respectivamente.

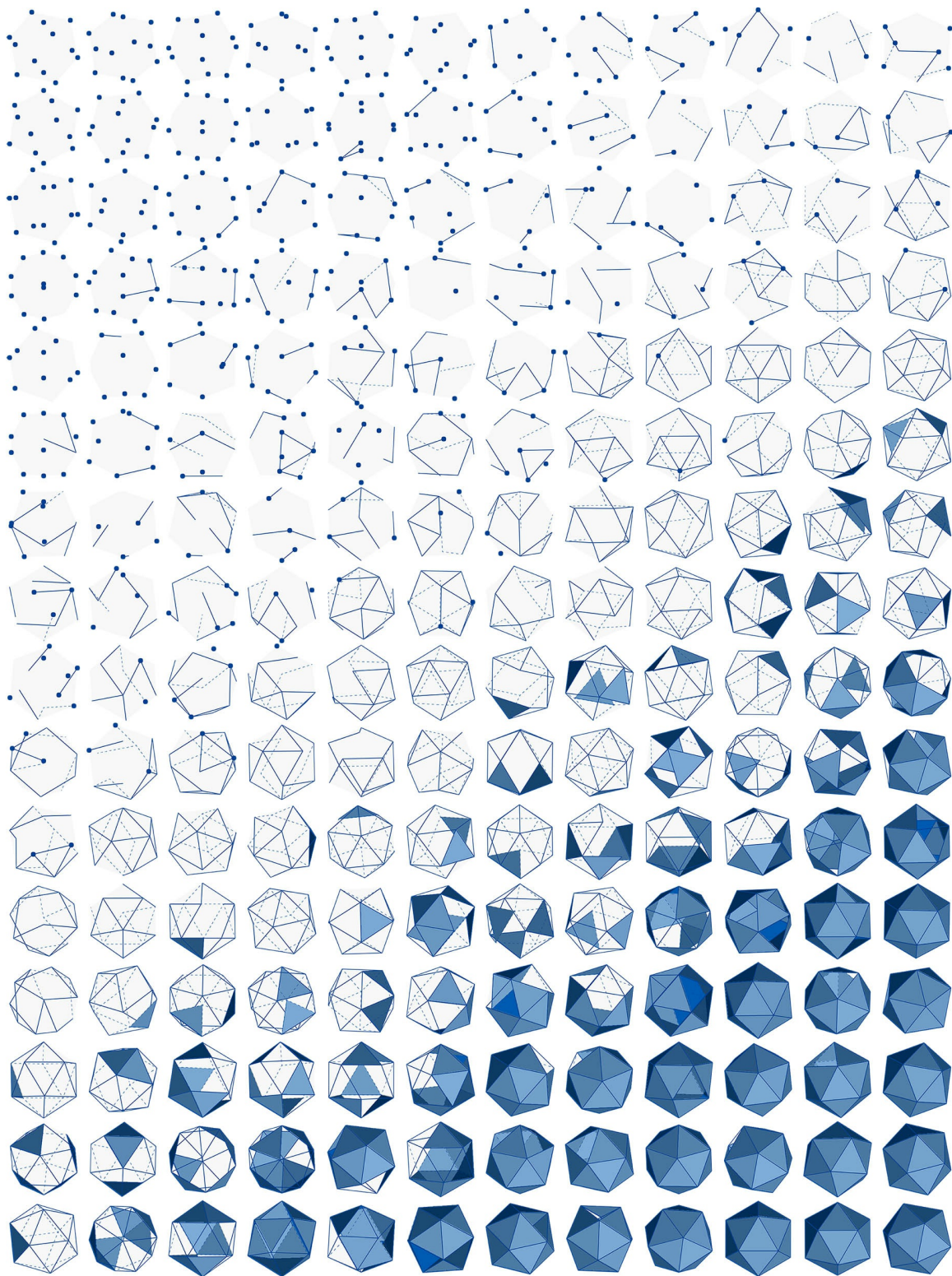


Vista preliminar ampliada de la superficie de salida con resolución de 400 x 300 muestras

Geometría para el diseño computacional

GEOMETRÍA PARA EL DISEÑO COMPUTACIONAL

Dynamo, un entorno de programación visual, le permite diseñar la forma en que se procesan los datos. Los datos son números o texto, pero también lo es la geometría. Tal y como lo entiende el equipo, la geometría (denominada a veces geometría computacional) son los datos que se pueden utilizar para crear modelos atractivos, complejos o basados en el rendimiento. Para ello, debemos conocer los pormenores de los diversos tipos de geometría que podemos utilizar.



Descripción general de la geometría

Descripción general de la geometría

La **geometría** es el idioma del diseño. Cuando un lenguaje o un entorno de programación tienen un núcleo de geometría, podemos descubrir las posibilidades del diseño de modelos precisos y sólidos, la automatización de las rutinas de diseño y la creación de iteraciones de diseño con algoritmos.

Conceptos básicos

La geometría se define tradicionalmente como el estudio de la forma, el tamaño y la posición relativa de las figuras y las propiedades del espacio. Este campo tiene una historia muy rica que se remonta a miles de años. Con la llegada y la popularización de la informática, disponemos de una poderosa herramienta para definir, explorar y generar geometría. Ahora es muy fácil calcular el resultado de interacciones geométricas complejas y lo hacemos de forma casi transparente.



Si tiene curiosidad por comprobar lo diversa y compleja que puede ser la geometría mediante la potencia de su ordenador, realice una búsqueda rápida en la Web del conejito de Stanford (o "Stanford Bunny"), un modelo canónico que se utiliza para probar algoritmos.

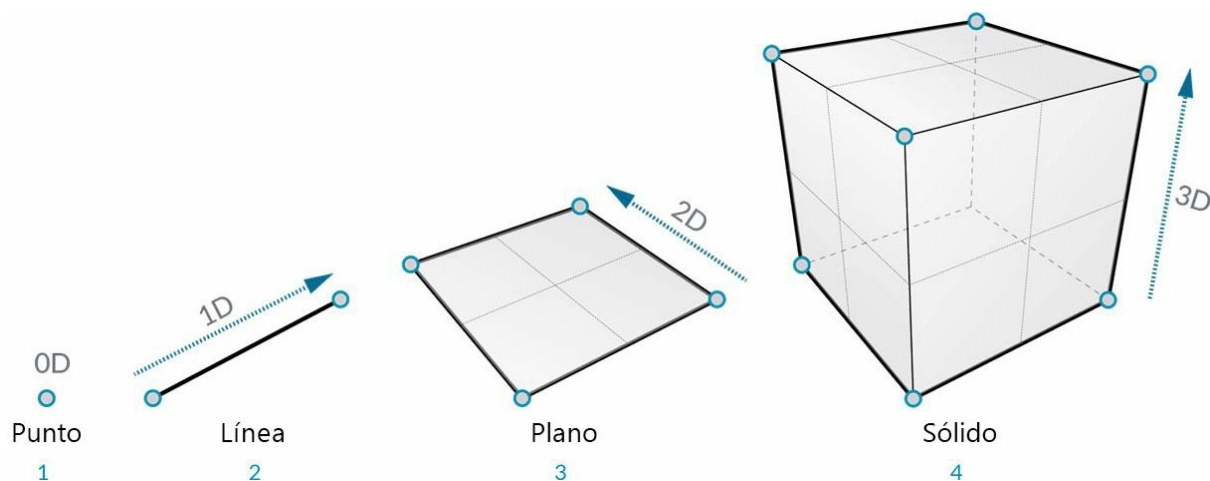
Comprender la geometría en el contexto de los algoritmos, la informática y la complejidad puede parecer una tarea abrumadora. Sin embargo, existen algunos principios clave y relativamente sencillos que podemos establecer como base para empezar a crear aplicaciones más avanzadas:

1. La geometría se compone de **datos**: para el ordenador y Dynamo, un conejito no se diferencia mucho de un número.
2. La geometría se basa en la **abstracción**: sobre todo, los elementos geométricos se describen mediante números, relaciones y fórmulas dentro de un sistema de coordenadas espacial específico.
3. La geometría tiene una **jerarquía**: los puntos se combinan para crear líneas, las líneas se unen para crear superficies, etc.
4. La geometría describe simultáneamente **la parte y el todo**: cuando tenemos una curva, se tiene en cuenta tanto la forma como todos los posibles puntos a lo largo de ella.

En la práctica, estos principios nos indican que debemos tener en cuenta con qué tipo de geometría estamos trabajando (qué tipo de geometría, cómo se creó, etc.) para poder crear, descomponer y recomponer de forma fluida diferentes geometrías a medida que desarrollamos modelos más complejos.

Recorrido por la jerarquía

Dediquemos un momento a observar la relación entre las descripciones abstractas y jerárquicas de la geometría. Como estos dos conceptos están relacionados, pero esto no es siempre evidente al principio, podemos encontrarnos rápidamente con un bloqueo conceptual una vez que empezamos a desarrollar flujos de trabajo o modelos más profundos. Para empezar, vamos a usar la dimensionalidad como un sencillo descriptor del "material" que modelamos. El número de dimensiones necesarias para describir una forma nos muestra cómo organizar la forma en que la geometría se organiza jerárquicamente.



1. Un **punto** (definido por coordenadas) no tiene ninguna dimensión asociada; son solo números que describen cada coordenada.
2. Una **línea** (definida por dos puntos) tiene ahora *una* dimensión; podemos "recorrer" la línea hacia delante (dirección positiva) o hacia atrás

(dirección negativa).

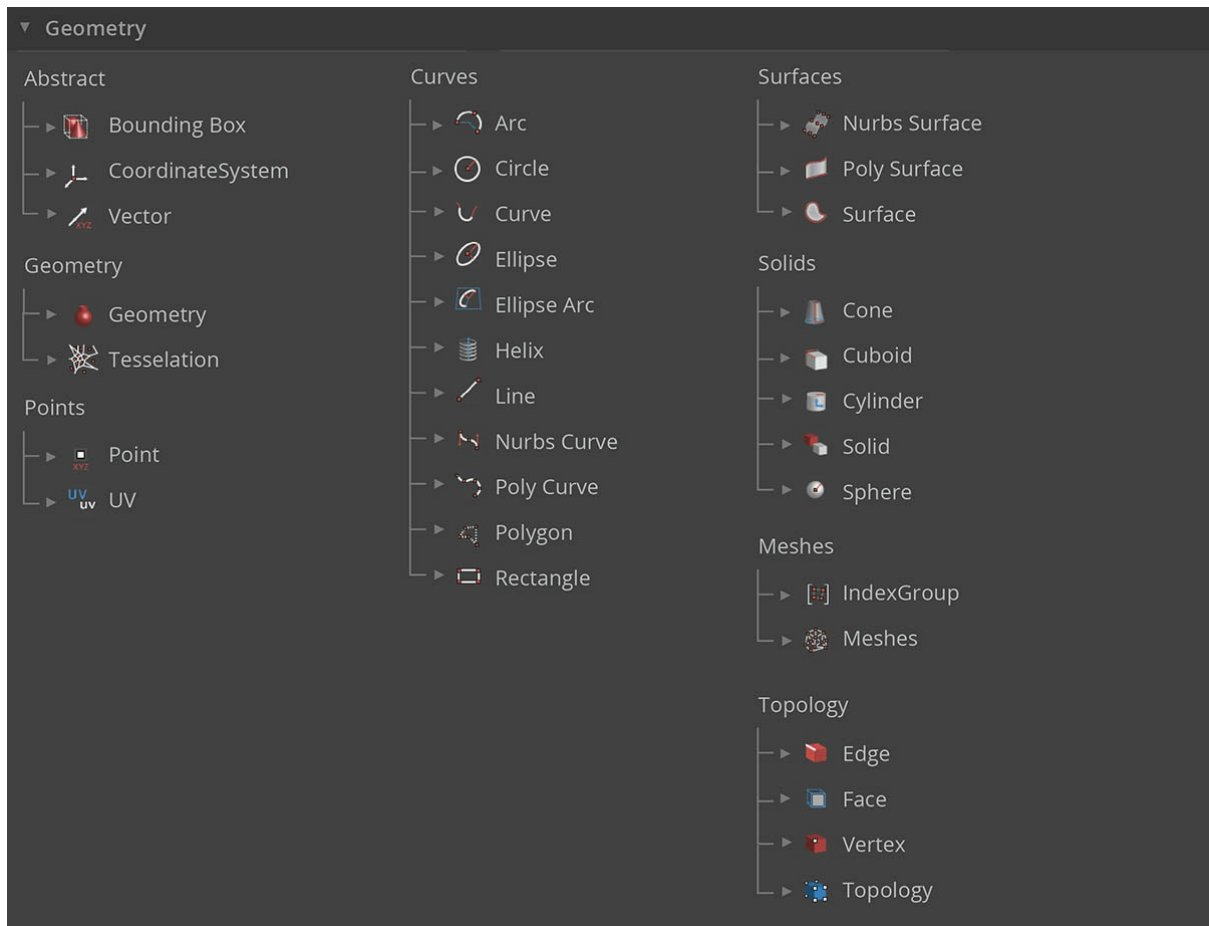
3. Un **plano** (definido por dos líneas) tiene *dos* dimensiones: es posible desplazarse más a la izquierda o más a la derecha.
4. Un **cubo** (definido por dos planos) tiene *tres* dimensiones: se puede definir una posición hacia arriba o hacia abajo.

La dimensionalidad es un método práctico para empezar a organizar en categorías la geometría, pero no es necesariamente el mejor. Después de todo, no modelamos solo puntos, líneas, planos y cuadros, ¿qué ocurre si deseo utilizar un elemento curvado? Además, existe toda una categoría adicional de tipos geométricos que son totalmente abstractos; por ejemplo, definen propiedades como la orientación, el volumen o las relaciones entre las partes. No podemos realmente agarrar un vector, así que, ¿cómo lo definimos en relación con lo que vemos en el espacio? Una categorización más detallada de la jerarquía geométrica debería incluir la diferencia entre los tipos abstractos o "ayudas", que podemos agrupar por lo que ayudan a hacer, y los tipos que ayudan a describir la forma de los elementos del modelo.

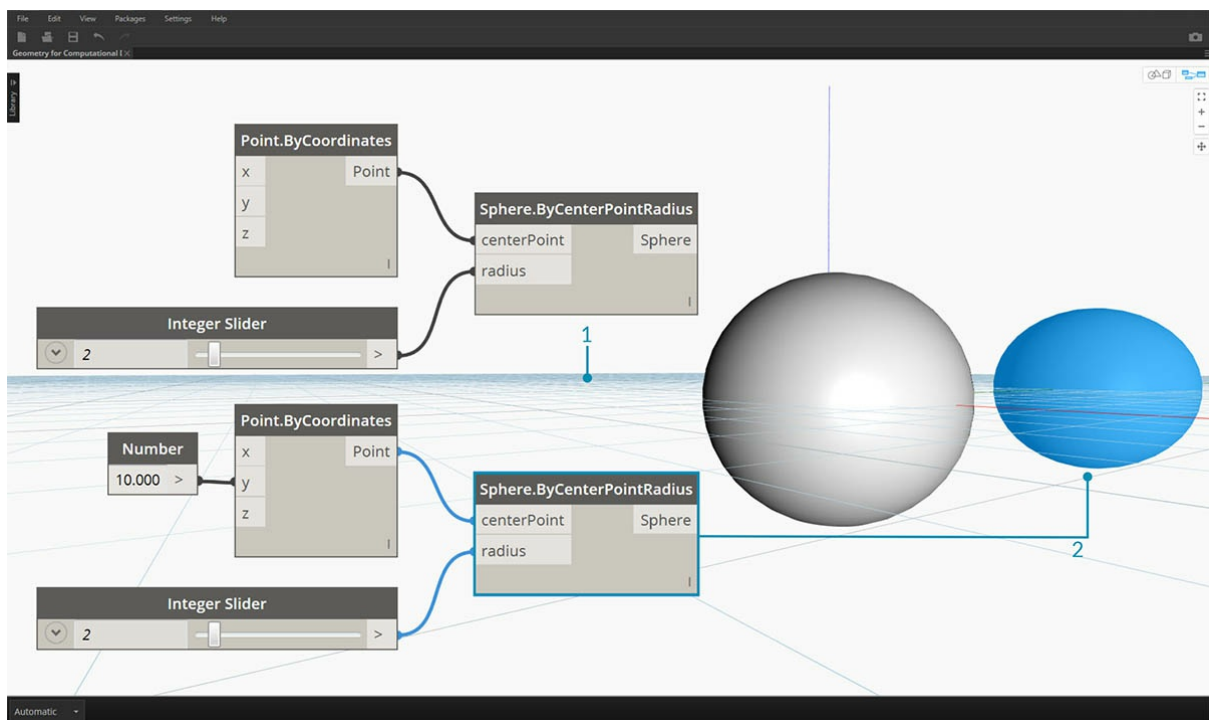
Jerarquía de tipos de datos							
Tipos abstractos			Tipos de geometría				
Define la ubicación + orientación	Define la posición + volumen	Define relaciones	Elementos de modelo				
Vector	Cuadro delimitador	Topología	Punto	Curva	Superficie	Sólido	Malla
• Vector • Plano • Sistema de coordenadas	• Cuadro delimitador	• Vértices • Borde • Cara	• Coordenada XYZ • Coordenada UV	• Línea • Polígono • Arco • Círculo • Elipse • Curva NURBS • PolyCurve	• Superficie NURBS • PolySurface	• Ortoedro • Esfera • Cono • Cilindro	• Malla

Geometría en Dynamo Sandbox

¿Cómo se refleja esto al utilizar Dynamo? Conocer los tipos de geometría y cómo se relacionan nos permitirá desplazarnos por la colección de **nodos de geometría** disponibles en la biblioteca. Los nodos de geometría se organizan alfabéticamente en lugar de jerárquicamente; aquí se muestran de forma similar a su presentación en la interfaz de Dynamo.



Además, la creación de modelos en Dynamo y la conexión de la vista preliminar de lo que aparece en la vista preliminar en segundo plano al flujo de datos del gráfico deberían ser procesos más intuitivos con el tiempo.



1. Observe el supuesto sistema de coordenadas renderizado por la rejilla y los ejes coloreados.
2. Los nodos seleccionados renderizarán la geometría correspondiente (si el nodo crea la geometría) en segundo plano con el color resaltado.

Descargue el archivo de ejemplo que acompaña a esta imagen (haga clic con el botón derecho y seleccione "Guardar vínculo como..."): [Geometry for Computational Design - Geometry Overview.dyn](#). En el Apéndice, se incluye una lista completa de los archivos de ejemplo.

Uso de la geometría para ir al siguiente nivel

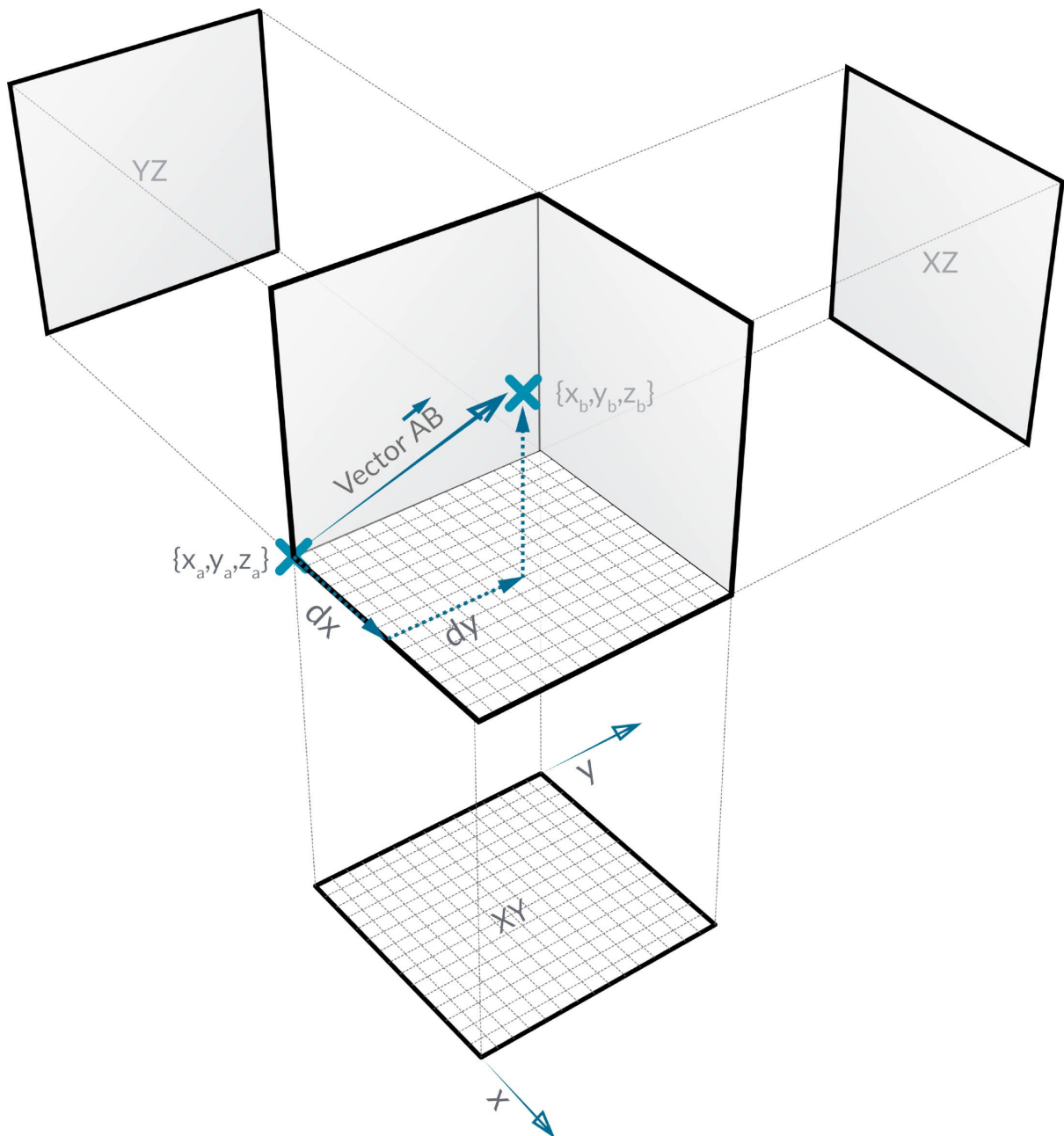
La creación de modelos en Dynamo no se limita a lo que se puede generar con nodos. A continuación, se indican algunas formas clave para llevar el proceso al siguiente nivel con la geometría:

1. Dynamo permite importar archivos: pruebe a utilizar un archivo CSV para nubes de puntos o SAT para incorporar superficies.
2. Al trabajar con Revit, podemos hacer referencia a los elementos de Revit que se utilizarán en Dynamo.
3. Dynamo Package Manager ofrece funciones adicionales para operaciones y tipos de geometría ampliados; consulte el paquete de [Kit de herramientas de malla](#).

Vectores

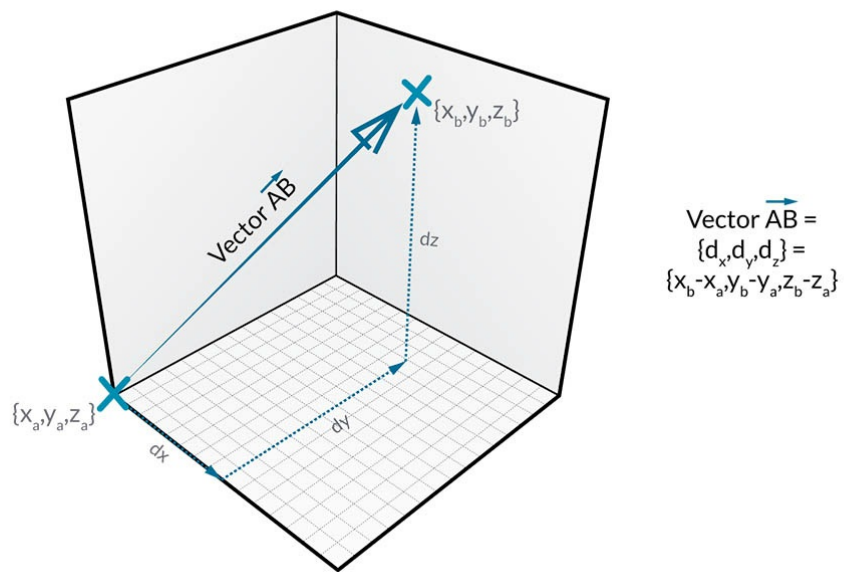
Vectores, planos y sistemas de coordenadas

Los vectores, los planos y los sistemas de coordenadas constituyen el grupo principal de los tipos de geometría abstracta. Nos ayudan a definir la ubicación, la orientación y el contexto espacial de otra geometría que describe formas. Si, por ejemplo, digo que estoy en la ciudad de Nueva York en la calle 42 con Broadway (sistema de coordenadas), de pie en el nivel de la calle (plano), mirando al norte (vector), acabo de usar estas "ayudas" para definir dónde estoy. Lo mismo sucede con la carcasa de un teléfono o un rascacielos; necesitamos este contexto para desarrollar el modelo.



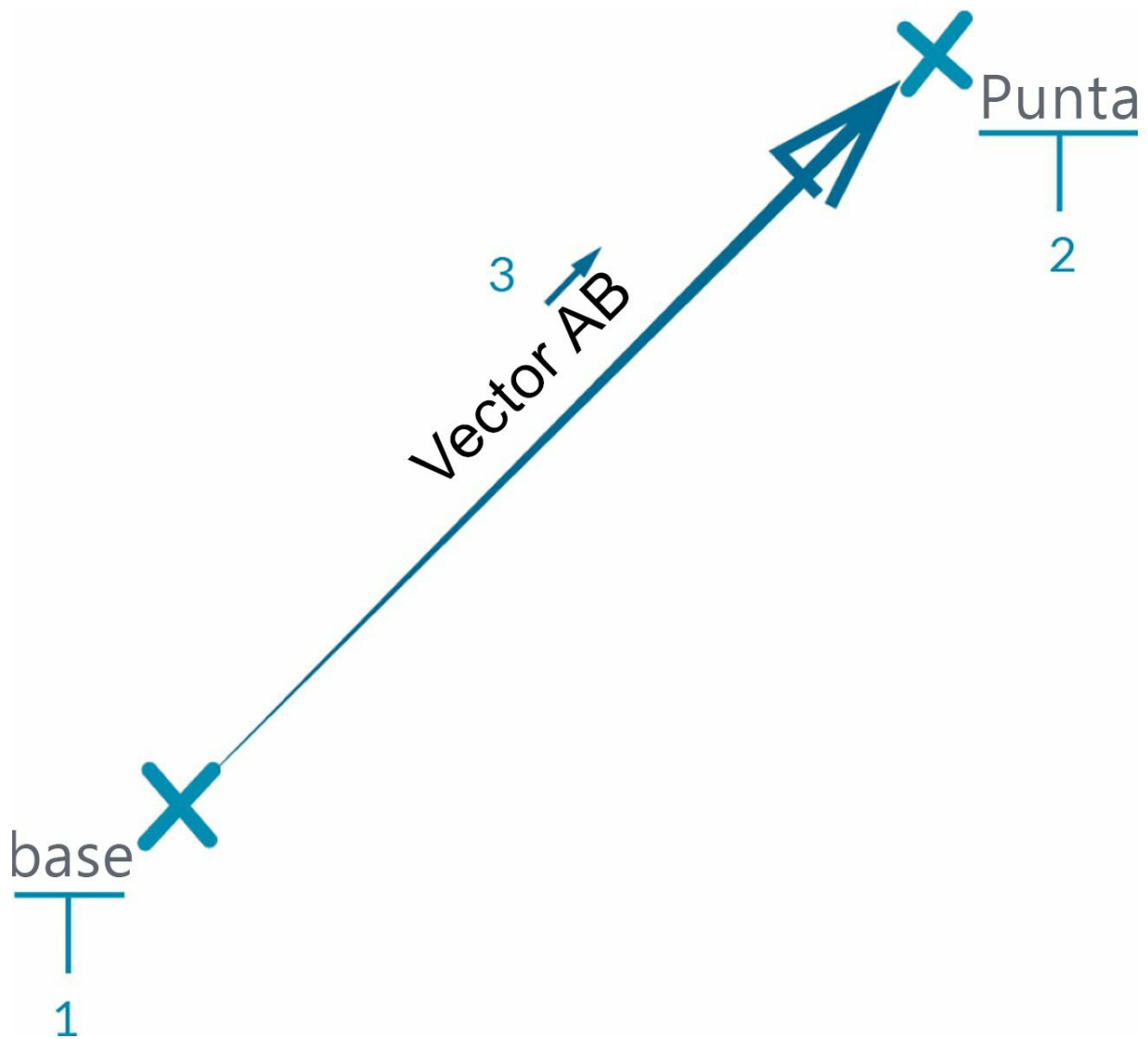
¿Qué es un vector?

Un vector es una cantidad geométrica que describe la dirección y la magnitud. Los vectores son abstractos, es decir, representan una cantidad, no un elemento geométrico. Los vectores se pueden confundir fácilmente con los puntos porque ambos están compuestos de una lista de valores. Aunque existe una diferencia clave: los puntos describen una posición en un sistema de coordenadas específico, mientras que los vectores describen una diferencia relativa en la posición, lo que equivale a la "dirección".



Si la idea de la diferencia relativa es confusa, piense en el Vector AB como "Estoy de pie en el punto A, mirando hacia el punto B". La dirección, de aquí (A) a allí (B), es el vector.

Este es un desglose de los vectores en sus diferentes partes mediante la misma notación AB:

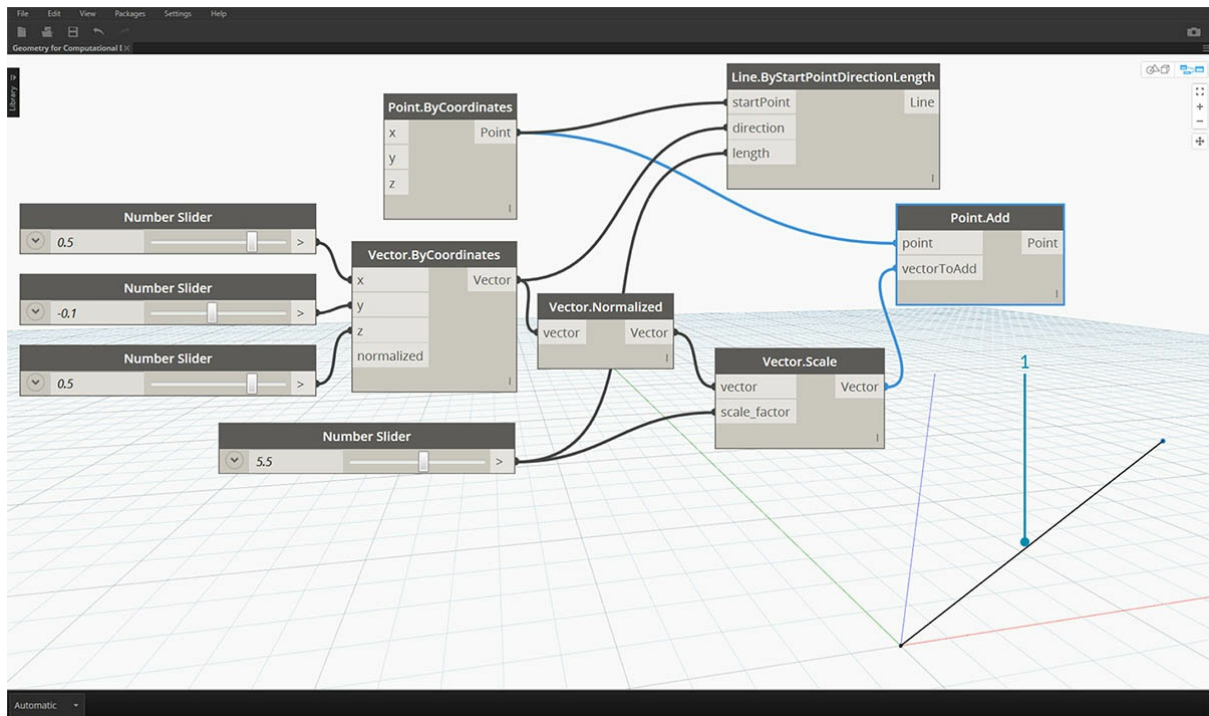


1. El **punto inicial** del vector se denomina **base**.
2. El **punto final** del vector se denomina **punta** o el **sentido**.
3. El vector AB no es igual al vector BA, que señalaría la dirección opuesta.

Si desea ver un momento cómico en relación con los vectores (y su definición abstracta), eche un vistazo al clásico de la comedia "Aterrizas como puedas" y escuche la siguiente línea de diálogo burlona bastante citada:

Bien, bien. ¿Cuál es nuestro vector, Víctor?

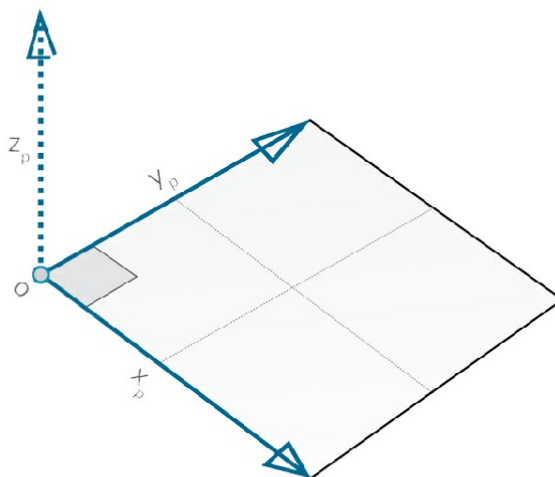
Los vectores son un componente clave de los modelos de Dynamo. Tenga en cuenta que, como se encuentran en la categoría abstracta de "ayudas", cuando creamos un vector, no aparecerá nada en la vista preliminar en segundo plano.



1. Podemos utilizar una línea como sustituto de una vista preliminar del vector. Descargue el archivo de ejemplo que acompaña a esta imagen (haga clic con el botón derecho y seleccione "Guardar vínculo como..."): [Geometry for Computational Design - Vectors.dyn](#). En el Apéndice, se incluye una lista completa de los archivos de ejemplo.

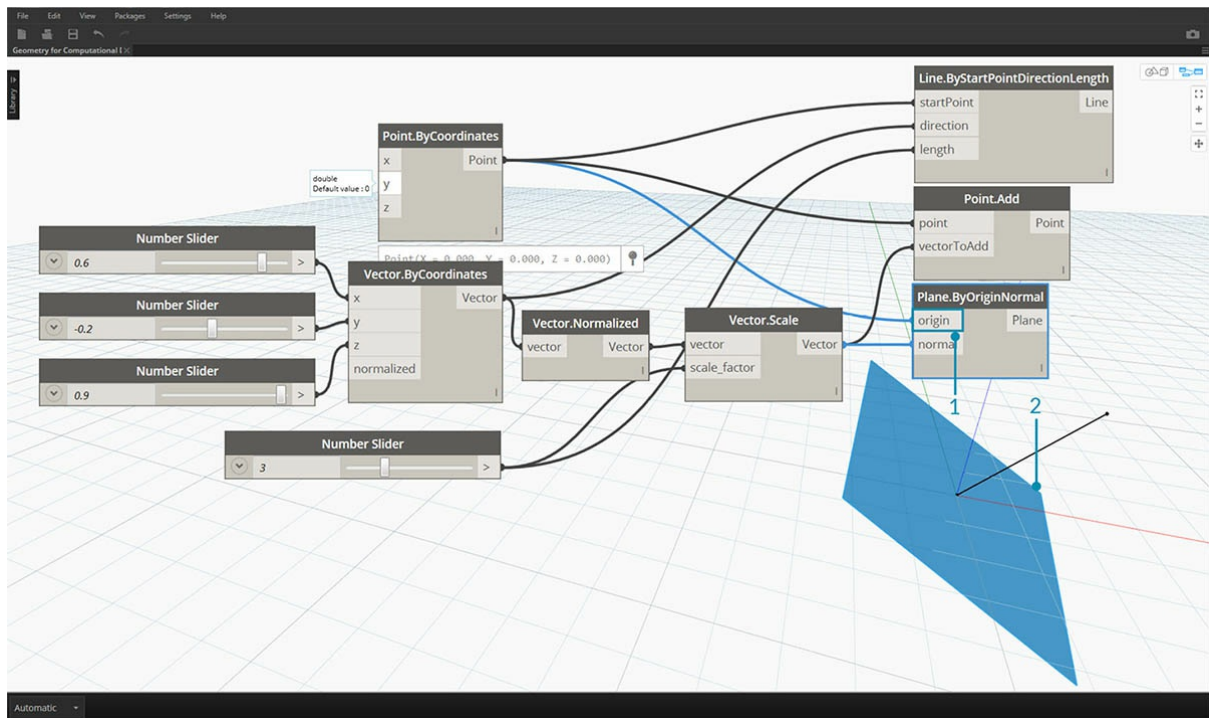
¿Qué es un plano?

Los planos son "ayudas" abstractas bidimensionales. En concreto, los planos son conceptualmente un "lano" que se extiende infinitamente en dos direcciones. Por lo general, se renderizan como un rectángulo más pequeño próximo a su origen.



Es posible que piense: "¡Un momento! ¿Origen? Eso suena a un sistema de coordenadas... como el que utilizo para modelar en el software de CAD".

Y tiene razón. La mayoría del software de modelado aprovecha los planos de construcción o "niveles" para definir un contexto local bidimensional en el que dibujar. Es posible que XY, XZ, YZ, o Norte, Sudeste o Nivel le resulten términos más familiares. Todos son planos que definen un contexto "lano" infinito. Los planos no tienen profundidad, pero nos ayudan a describir la dirección: cada plano tiene un origen, y una dirección X, Y y Z (arriba).

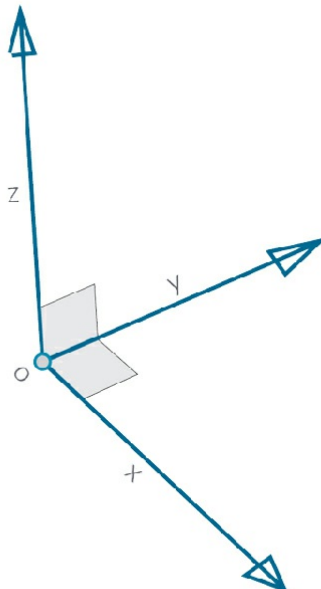


1. Aunque son abstractos, los planos tienen una posición de origen, por lo que podemos localizarlos en el espacio.
2. En Dynamo, los planos se renderizan en la vista preliminar en segundo plano. Descargue el archivo de ejemplo que acompaña a esta imagen (haga clic con el botón derecho y seleccione "Guardar vínculo como..."): [Geometry for Computational Design - Planes.dyn](#). En el Apéndice, se incluye una lista completa de los archivos de ejemplo.

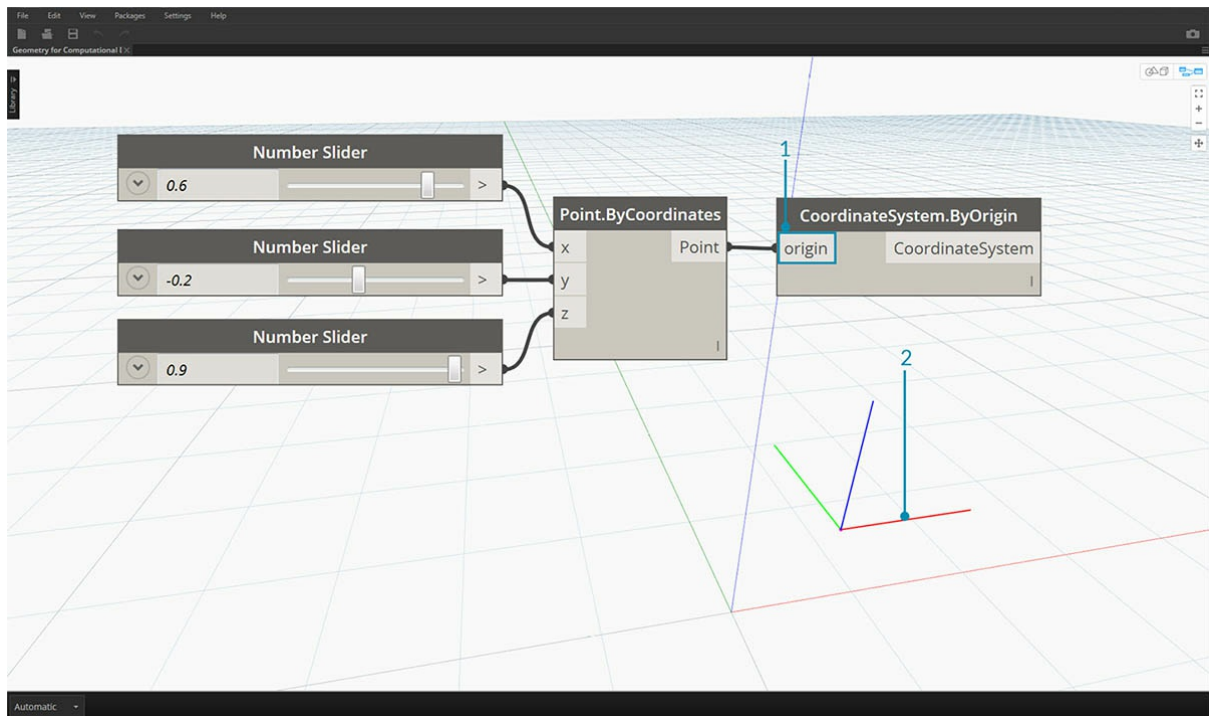
¿Qué es un sistema de coordenadas?

Si estamos familiarizados con los planos, estamos a un paso de comprender los sistemas de coordenadas. Un plano tiene las mismas partes que un sistema de coordenadas, siempre que se trate de un sistema de coordenadas euclidiano o XYZ estándar.

Sin embargo, hay otros sistemas de coordenadas alternativos como, por ejemplo, cilíndricos o esféricos. Como veremos en las secciones posteriores, los sistemas de coordenadas también se pueden aplicar a otros tipos de geometría para definir una posición en esa geometría.



Añadir sistemas de coordenadas alternativos: cilíndricos y esféricos.

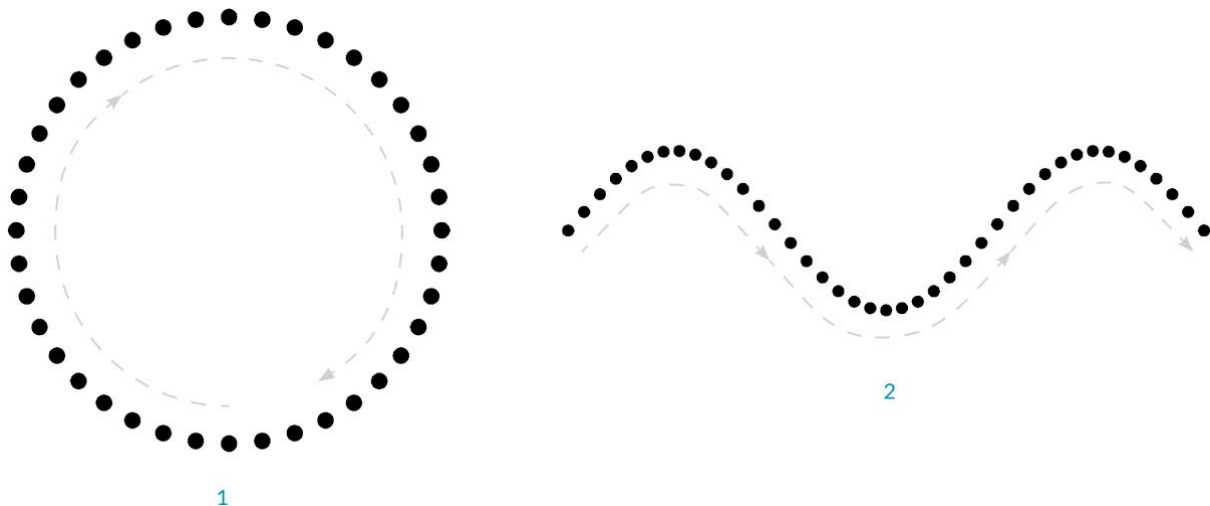


1. Aunque son abstractos, los sistemas de coordenadas también tienen una posición de origen para poder localizarlos en el espacio.
2. En Dynamo, los sistemas de coordenadas se renderizan en la vista preliminar en segundo plano como un punto (origen) y líneas que definen los ejes (X es rojo, Y es verde y Z es azul según la convención). Descargue el archivo de ejemplo que acompaña a esta imagen (haga clic con el botón derecho y seleccione "Guardar vínculo como..."): [Geometry for Computational Design - Coordinate System.dyn](#). En el Apéndice, se incluye una lista completa de los archivos de ejemplo.

Puntos

Puntos

Si la geometría es el idioma de un modelo, los puntos son el alfabeto. Los puntos son la base sobre la que se crea el resto de la geometría; se necesitan al menos dos puntos para crear una curva, tres puntos para crear un polígono o una cara de malla, etc. La definición de la posición, el orden y la relación entre los puntos (pruebe con una función de seno) nos permite definir geometrías de orden superior, como elementos que reconocemos como círculos o curvas.



1. Un círculo con las funciones $x=r*\cos(t)$ y $y=r*\sin(t)$.
2. Una curva de seno mediante las funciones $x=(t)$ y $y=r*\sin(t)$.

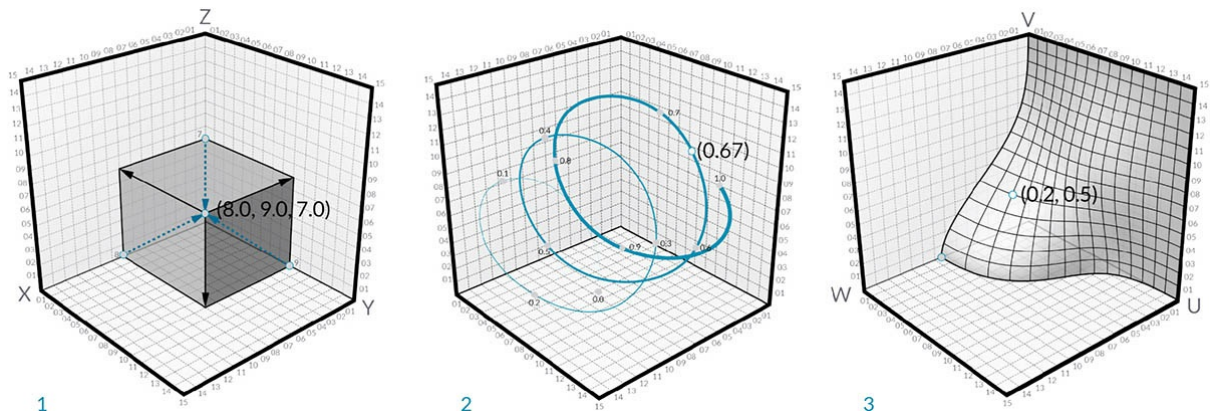
¿Qué es un punto?

Un punto se define por uno o más valores denominados coordenadas. El número de valores de coordenadas necesarios para definir el punto depende del sistema de coordenadas o del contexto en el que se encuentra. El tipo de punto más común en Dynamo se encuentra en el sistema de coordenadas universales tridimensional y tiene tres coordenadas [X,Y,Z].



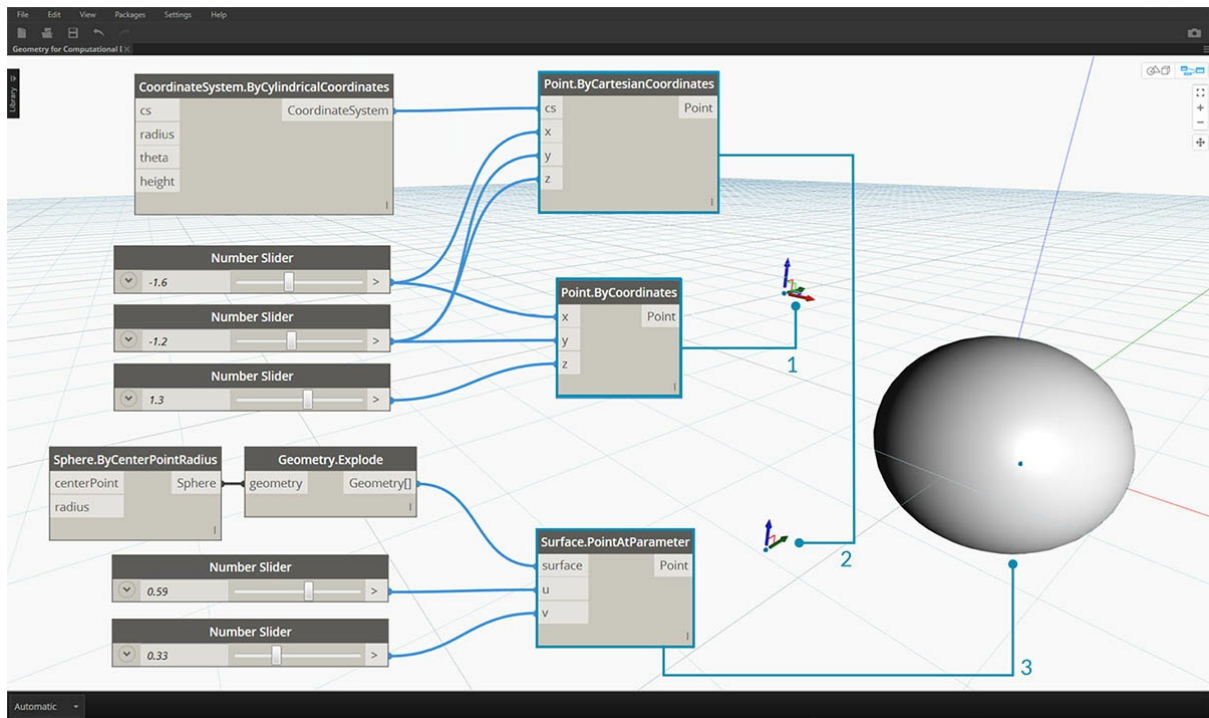
Punto como coordenadas

Los puntos también pueden existir en un sistema de coordenadas bidimensional. La convención tiene diferentes notaciones de letras en función del tipo de espacio con el que se trabaja: es posible que se utilice [X,Y] en un plano o [U,V] en una superficie.



1. Un punto en el sistema de coordenadas euclidianas: [X,Y,Z]
2. Un punto en un sistema de coordenadas de parámetro de curva: [t]
3. Un punto en un sistema de coordenadas de parámetro de superficie: [U,V]

Aunque puede parecer que va en contra del sentido común, los parámetros de las curvas y las superficies son continuos y se extienden más allá de la arista de la geometría especificada. Como las formas que definen el espacio paramétrico se encuentran en un sistema de coordenadas universales tridimensional, siempre se puede convertir una coordenada paramétrica en una coordenada "universal". Por ejemplo, el punto [0,2, 0,5] de la superficie es el mismo que el punto [1,8, 2,0, 4,1] de las coordenadas universales.

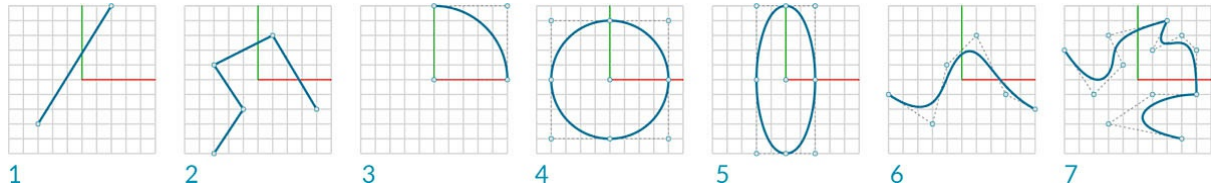


1. Punto en presuntas coordenadas XYZ globales
2. Punto relativo a un sistema de coordenadas especificado (cilíndrico)
3. Punto como coordenada UV en una superficie Descargue el archivo de ejemplo que acompaña a esta imagen (haga clic con el botón derecho y seleccione "Guardar vínculo como..."): [Geometry for Computational Design - Points.dyn](#). En el Apéndice, se incluye una lista completa de los archivos de ejemplo.

Curvas

Curvas

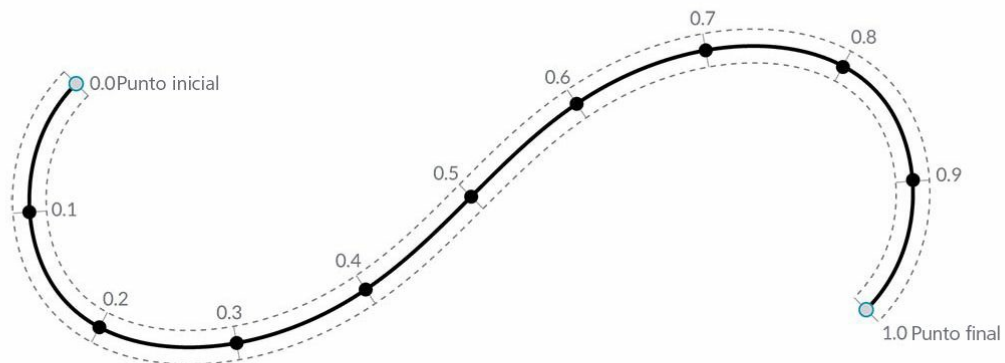
Las curvas son el primer tipo de datos geométricos abordado que presenta un conjunto más familiar de propiedades descriptivas de forma... ¿Qué grado de curvatura o rectitud? ¿Cómo de larga o corta? Recuerde que los puntos siguen siendo nuestros bloques de construcción para definir cualquier elemento, desde una línea a una spline y todos los tipos de curva intermedios.



1. Línea
2. Polilínea
3. Arco
4. Círculo
5. Elipse
6. Curva NURBS
7. PolyCurve

¿Qué es una curva?

El término **curva** suele ser un comodín para todo tipo de formas curvas (incluso aunque adopten un aspecto recto). La curva en su sentido primordial es la categorización principal de todos estos tipos de forma: líneas, círculos, splines, etc. Desde una perspectiva más técnica, una curva describe cada punto posible que se puede encontrar introduciendo "t" en una conjunto de funciones, que pueden ir desde la sencilla función ($x = -1,26 * t$, $y = t$) hasta funciones que impliquen cálculo. Independientemente del tipo de curva con el que trabajemos, este **parámetro** denominado "t" es una propiedad que se puede evaluar. Además, independientemente del aspecto de la forma, todas las curvas tienen también un punto inicial y un punto final, que se alinean de forma coincidente con los valores t mínimo y máximo utilizados para crear la curva. Esto también nos ayuda a comprender su direccionalidad.

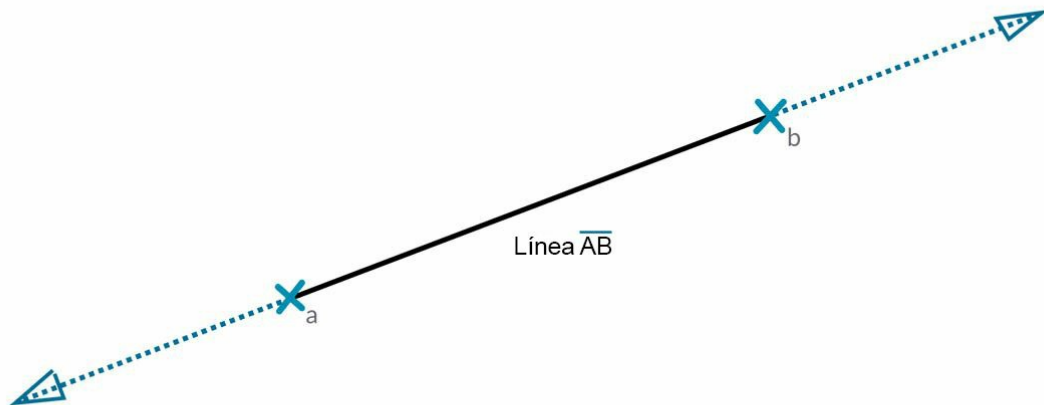


Es importante tener en cuenta que Dynamo presupone que el dominio de los valores "t" de una curva se entiende como de 0.0 a 1.0.

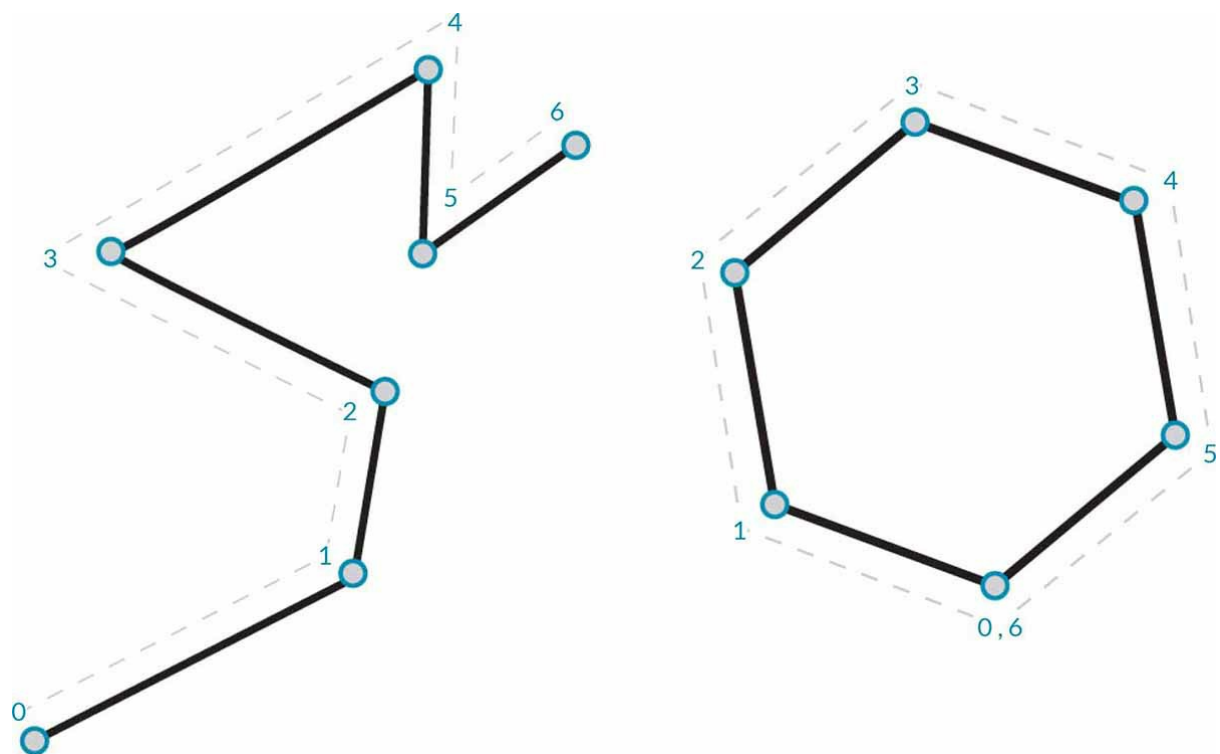
Todas las curvas también poseen una serie de propiedades o características que se pueden utilizar para describirlas o analizarlas. Cuando la distancia entre los puntos inicial y final es cero, la curva es "cerrada". Además, cada curva tiene varios puntos de control; si todos estos puntos se encuentran en el mismo plano, la curva es "plana". Algunas propiedades se aplican a la curva como un todo, mientras que otras solo se aplican a puntos específicos a lo largo de la curva. Por ejemplo, la planaridad es una propiedad global, mientras que un vector tangente en un valor t especificado es una propiedad local.

Líneas

Las **líneas** son la forma más sencilla de las curvas. Puede que no parezcan curvadas, pero en realidad son curvas, solo que no tienen ninguna curvatura. Existen varias formas diferentes de crear líneas; el más intuitivo desde el punto A al punto B. La forma de la línea AB se dibujará entre los puntos, pero matemáticamente se extiende infinitamente en ambas direcciones.

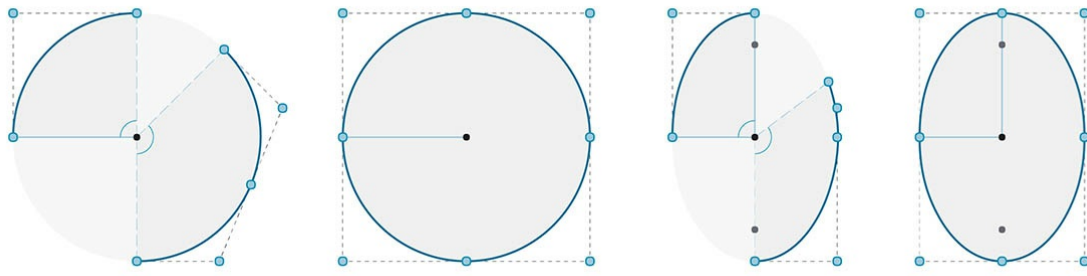


Cuando conectamos dos líneas, tenemos una **polilínea**. Aquí tenemos una representación directa de lo que es un punto de control. Si se edita cualquiera de estas ubicaciones de punto, cambiará la forma de la polilínea. Si la polilínea está cerrada, se trata de un polígono. Si todas las longitudes de arista del polígono son iguales, se describe como normal.



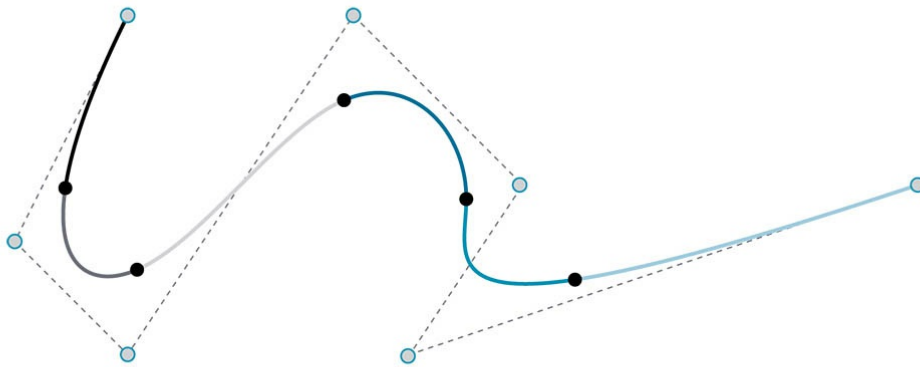
Arcos, círculos, arcos elípticos y elipses

A medida que se añade más complejidad a las funciones paramétricas que definen una forma, podemos ir un poco más allá de la línea y crear un **arco**, un **círculo**, un **arco elíptico** o una **elipse** mediante la descripción de uno o dos radios. Las diferencias entre la versión de arco y el círculo o la elipse se encuentran en si la forma está cerrada o no.



NURBS + PolyCurves

NURBS (Non-uniform Rational Basis Splines, splines de base racionales no uniformes) son representaciones matemáticas que pueden modelar con precisión cualquier forma, desde líneas, círculos, arcos o rectángulos bidimensionales sencillos a la curva orgánica tridimensional de forma libre más compleja. Gracias a su flexibilidad (hay relativamente pocos puntos de control, aunque la interpolación suave se basa en parámetros de grado) y su precisión (vinculada a complejas operaciones matemáticas), los modelos NURBS se pueden utilizar en cualquier proceso, desde la ilustración y la animación hasta la fabricación.

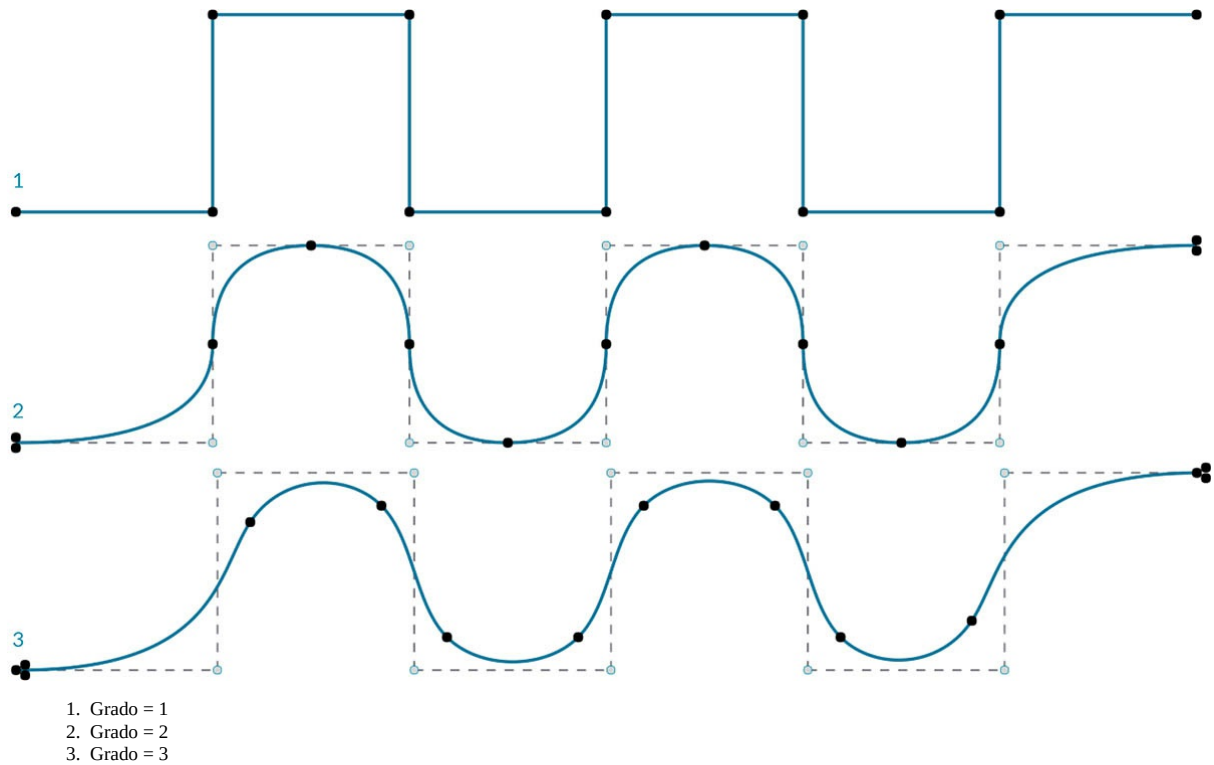


Grado: el grado de la curva determina el rango de influencia que los puntos de control tienen en una curva; cuanto mayor sea el grado, mayor será el rango. El grado es un número entero positivo. Este número suele ser 1, 2, 3 o 5, pero puede ser cualquier número entero positivo. Las líneas y las polilíneas NURBS suelen ser de grado 1 y la mayoría de las curvas de forma libre son de grado 3 o 5.

Puntos de control: los puntos de control son una lista de puntos de al menos $\text{grado}+1$. Una de las formas más fáciles de cambiar la forma de una curva NURBS es desplazar sus puntos de control.

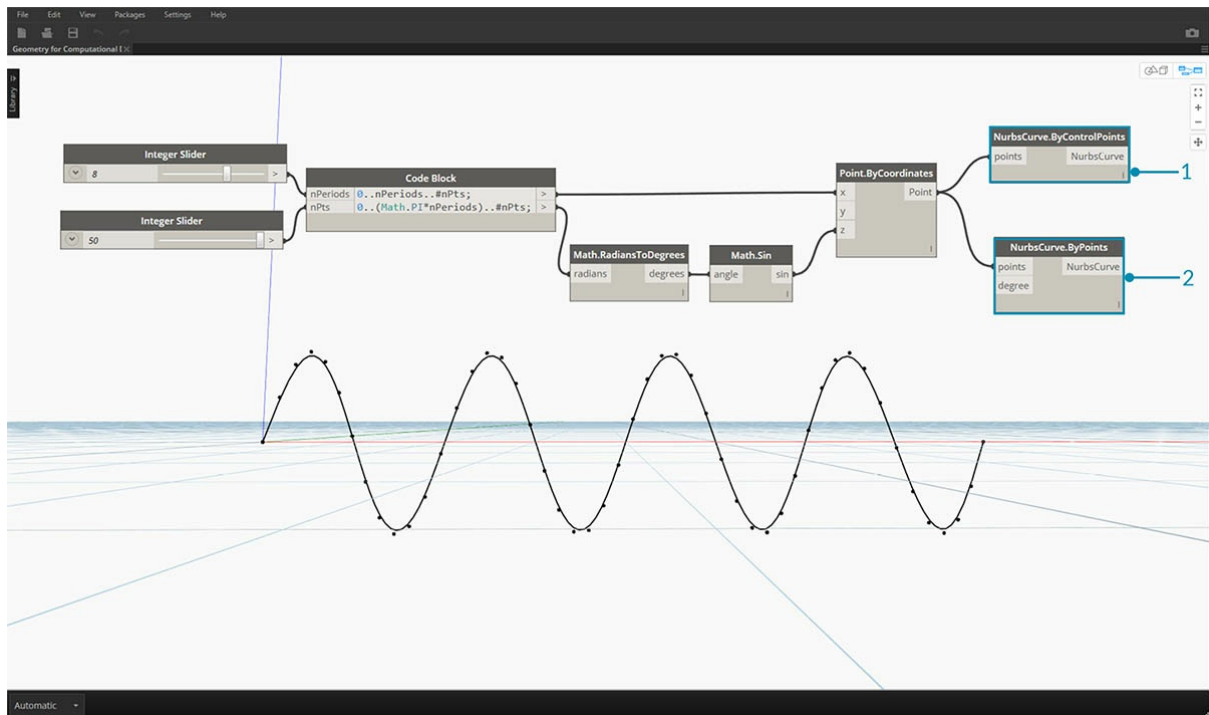
Grosor: los puntos de control tienen un número asociado denominado grosor. Los grosores son normalmente números positivos. Cuando todos los puntos de control de una curva tienen el mismo grosor (normalmente, 1), la curva se denomina no racional; de lo contrario, la curva se denomina racional. La mayoría de las curvas NURBS son no racionales.

Nudos: los nudos son una lista de números ($\text{grado}+N-1$), donde N es el número de puntos de control. Los nudos se utilizan junto con los grosores para controlar la influencia de los puntos de control en la curva resultante. Por ejemplo, los nudos se pueden utilizar para crear puntos de torsión en determinados puntos de la curva.



Tenga en cuenta que cuanto mayor sea el valor de grado, más puntos de control se utilizarán para interpolar la curva resultante.

Vamos a crear una curva de seno en Dynamo mediante dos métodos diferentes para crear curvas NURBS a fin de comparar los resultados.



1. *NurbsCurve.ByControlPoints* utiliza la lista de puntos como puntos de control.
2. *NurbsCurve.ByPoints* dibuja una curva a través de la lista de puntos. Descargue el archivo de ejemplo que acompaña a esta imagen (haga clic con el botón derecho y seleccione "Guardar vínculo como..."): [Geometry for Computational Design - Curves.dyn](#). En el Apéndice, se incluye una lista completa de los archivos de ejemplo.

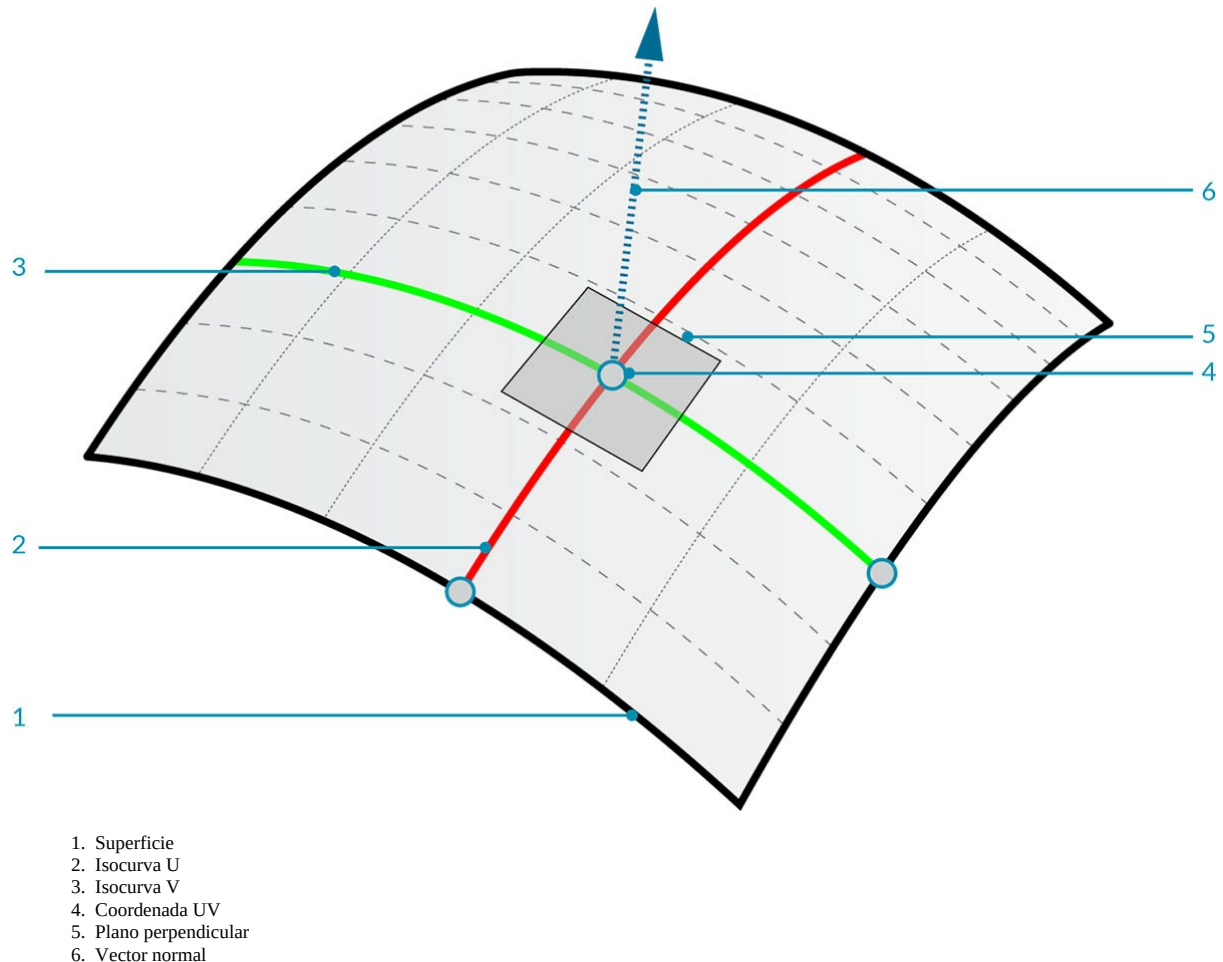
Superficies

Superficies

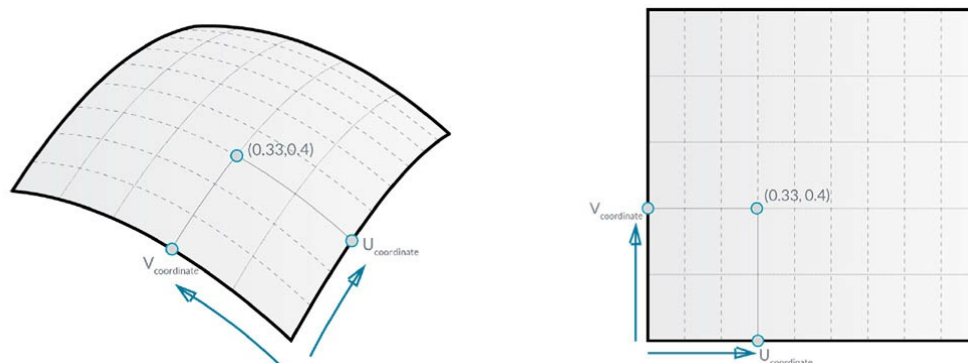
A medida que pasamos de usar curvas a usar superficies en un modelo, ahora podemos empezar a representar objetos presentes en nuestro mundo tridimensional. Aunque las curvas no son siempre planas, es decir, son tridimensionales, el espacio que definen siempre está vinculado a una dimensión. Las superficies nos ofrecen otra dimensión y un conjunto de propiedades adicionales que se pueden utilizar en otras operaciones de modelado.

¿Qué es una superficie?

Una superficie es una forma matemática definida por una función y dos parámetros; en lugar de t para las curvas, utilizamos U y V para describir el espacio de parámetros correspondiente. Esto significa que hay más datos geométricos que dibujar cuando se trabaja con este tipo de geometría. Por ejemplo, las curvas tienen vectores y planos normales (que se pueden rotar o girar a lo largo de la longitud de la curva), mientras que las superficies tienen vectores normales y planos tangentes que serán coherentes en su orientación.



Dominio de superficie: un dominio de superficie se define como el rango de parámetros (U, V) que se evalúan en un punto tridimensional de esa superficie. El dominio de cada dimensión (U o V) se describe normalmente como dos números, de U mín. a U máx. y de V mín. a V máx.

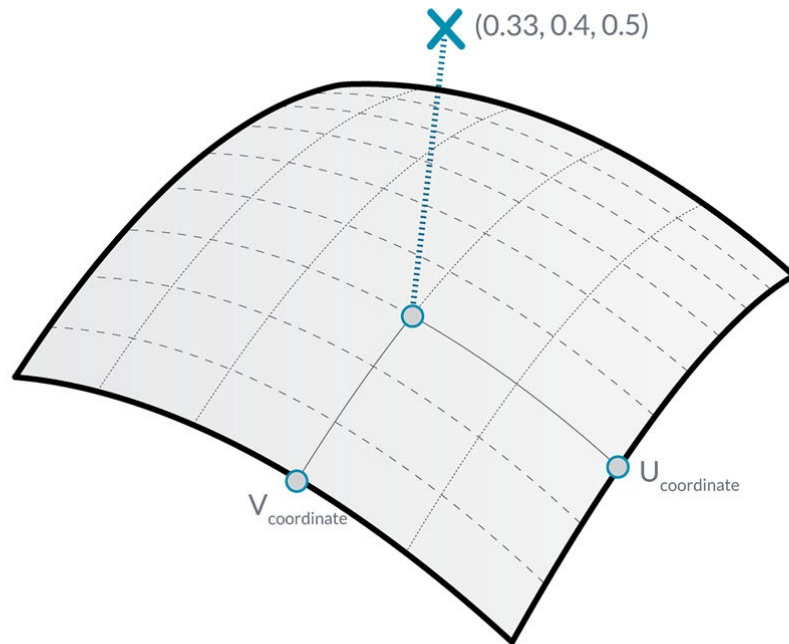


Aunque la forma de la superficie puede no parecer "rectangular" y es posible que tenga un conjunto de isocurvas más ajustadas o sueltas, el "espacio"

definido por su dominio siempre es bidimensional. En Dynamo, se entiende que las superficies tienen un dominio definido por un mínimo de 0.0 y un máximo de 1.0 en las direcciones U y V. Las superficies planas o recortadas pueden tener dominios diferentes.

Isocurva (o curva isoparamétrica): curva definida por un valor U o V constante en la superficie y un dominio de valores para la otra dirección U o V correspondiente.

Coordenada UV: el punto del espacio del parámetro UV definido por U, V y, a veces, W.

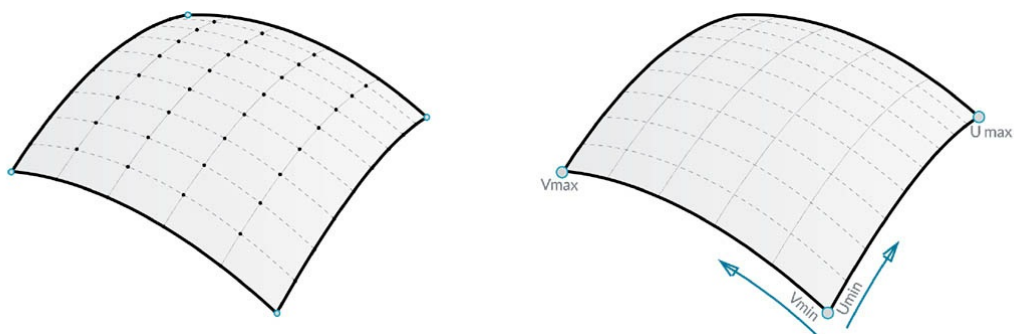


Plano perpendicular: un plano perpendicular a las isocurvas U y V en una coordenada UV especificada.

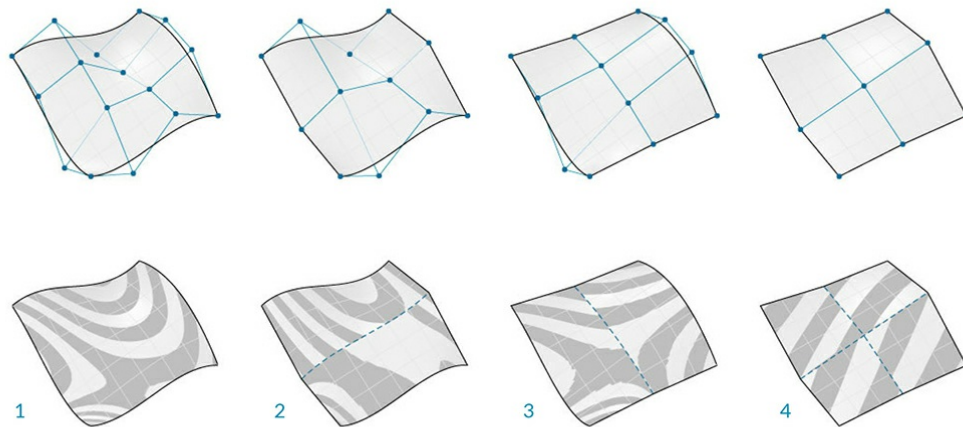
Vector normal: un vector que define la dirección de "arriba" en relación con el plano perpendicular.

Superficies NURBS

Las **superficies NURBS** son muy similares a las curvas NURBS. Las superficies NURBS se pueden considerar como una rejilla de curvas NURBS que van en dos direcciones. La forma de una superficie NURBS se define mediante un número de puntos de control y el grado de esa superficie en las direcciones U y V. Se utilizan los mismos algoritmos para calcular la forma, las normales, las tangentes, las curvaturas y otras propiedades mediante puntos de control, grosores y grados.



En el caso de las superficies NURBS, la geometría indica dos direcciones, ya que las superficies NURBS son, independientemente de la forma que veamos, rejillas rectangulares de puntos de control. Y, aunque estas direcciones son a menudo arbitrarias en relación con el sistema de coordenadas universal, las utilizaremos frecuentemente para analizar los modelos o generar otra geometría en función de la superficie.

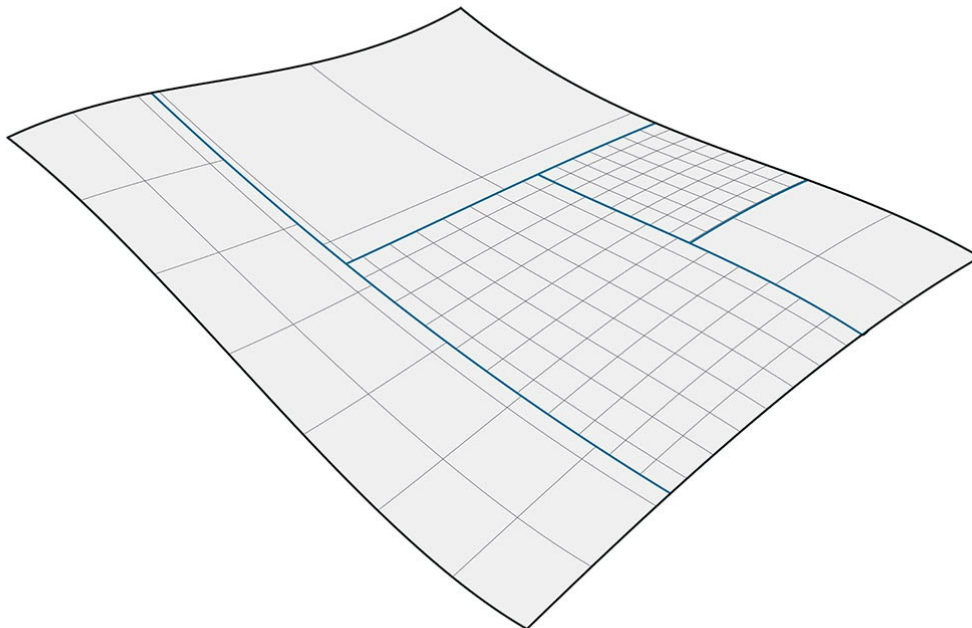


1. Grado (U,V) = (3,3)
2. Grado (U,V) = (3,1)
3. Grado (U,V) = (1,2)
4. Grado (U,V) = (1,1)

PolySurfaces

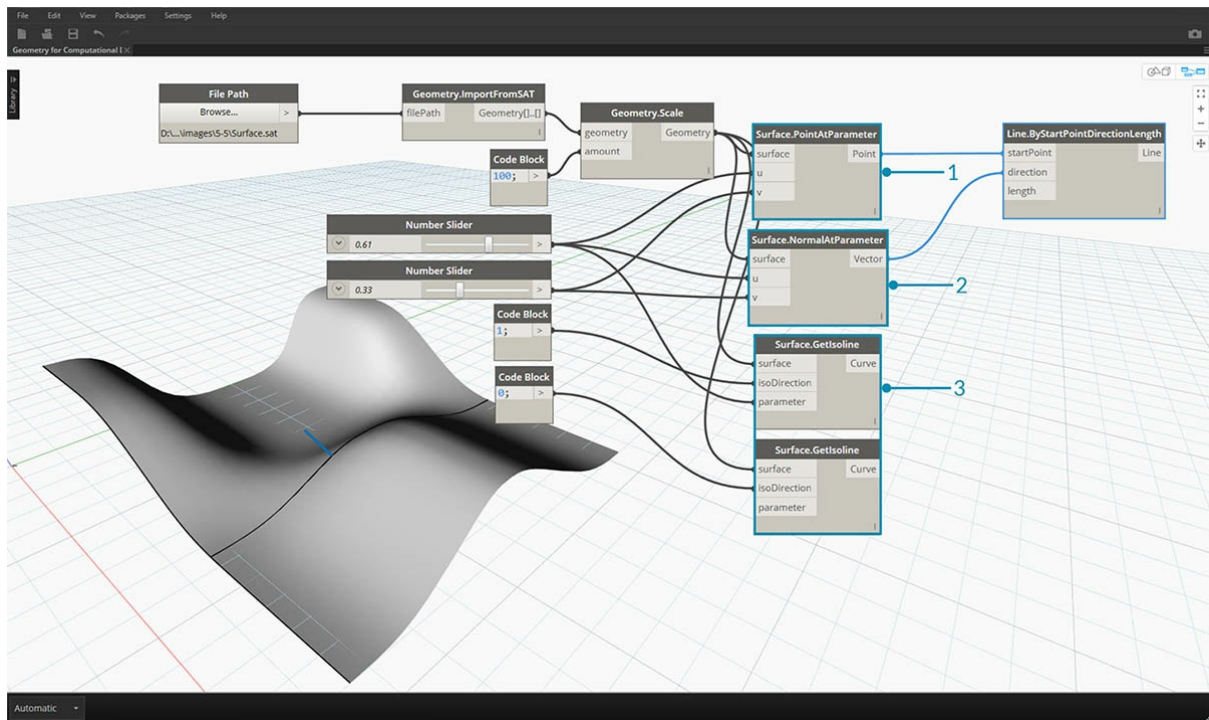
Las **PolySurfaces** se componen de superficies unidas a lo largo de una arista. Las PolySurfaces ofrecen más de una definición de UV bidimensional, por lo que ahora podemos desplazarnos por las formas conectadas a través de su topología.

Aunque la "topología" describe normalmente un concepto sobre el modo en que las piezas están conectadas o relacionadas, la topología en Dynamo es también un tipo de geometría. En concreto, es una categoría principal de superficies, PolySurfaces y sólidos.



En ocasiones denominadas parches, este modo de unión de superficies nos permite crear formas más complejas y definir detalles a lo largo de la unión. De forma oportuna, podemos aplicar una operación de empalme o chaflán a las aristas de una PolySurface.

Vamos a importar y evaluar una superficie en un parámetro de Dynamo para ver qué tipo de información se puede extraer.



1. *Surface.PointAtParameter* devuelve el punto en una coordenada UV especificada.
2. *Surface.NormalAtParameter* devuelve el vector normal en una coordenada UV especificada.
3. *Surface.GetIsoline* devuelve la curva isoparamétrica en una coordenada U o V; observe la entrada de isoDirection. Descargue los archivos de ejemplo que acompañan a esta imagen (haga clic con el botón derecho y seleccione "Guardar enlace como..."). En el Apéndice, se incluye una lista completa de los archivos de ejemplo.
4. [Geometry for Computational Design - Surfaces.dyn](#)
5. [Surface.sat](#)

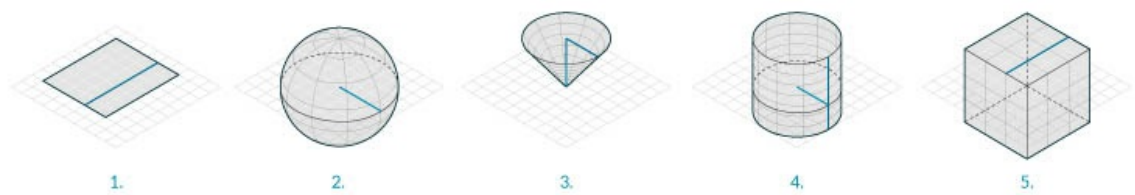
Sólidos

Sólidos

Si deseamos crear modelos más complejos que no se puedan crear a partir de una única superficie o si queremos definir un volumen explícito, debemos aventurarnos en el terreno de los sólidos (y las PolySurfaces). Incluso un cubo sencillo es lo suficientemente complejo como para necesitar seis superficies, una por cara. Los sólidos proporcionan acceso a dos conceptos clave que las superficies no ofrecen: una descripción topológica más refinada (caras, aristas y vértices) y operaciones booleanas.

¿Qué es un sólido?

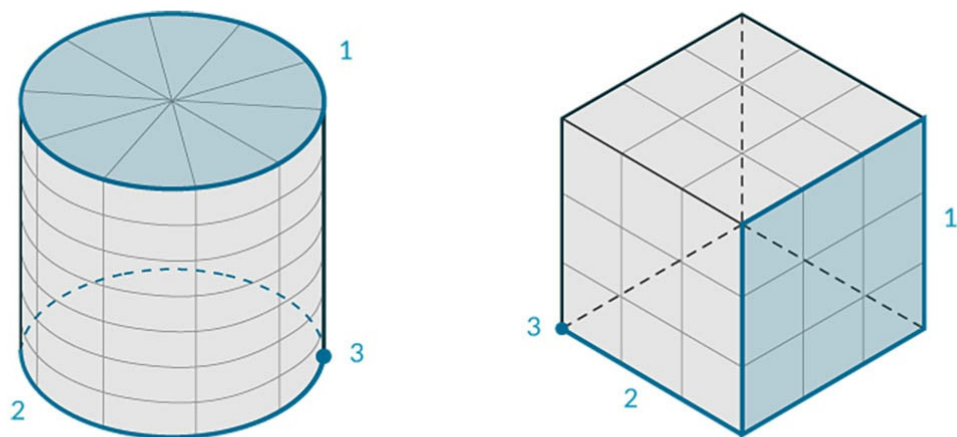
Los sólidos constan de una o varias superficies que contienen volumen a partir de un contorno cerrado que define lo que está "dentro" o "fuera". Independientemente de cuántas superficies haya, estas deben formar un volumen "hermético" para que se considere un sólido. Los sólidos se pueden crear mediante la unión de superficies o PolySurfaces, o mediante operaciones como, por ejemplo, elevación, barrido y revolución. Las primitivas de esfera, cubo, cono y cilindro también son sólidos. Un cubo con al menos una cara eliminada se considera una PolySurface, que tiene algunas propiedades similares, pero no es un sólido.



1. Un plano está compuesto por una única superficie y no es un sólido.
2. Una esfera está formada por una única superficie, pero es un sólido.
3. Un cono está formado por dos superficies unidas para crear un sólido.
4. Un cilindro está formado por tres superficies unidas para crear un sólido.
5. Un cubo está formado por seis superficies unidas para crear un sólido.

Topología

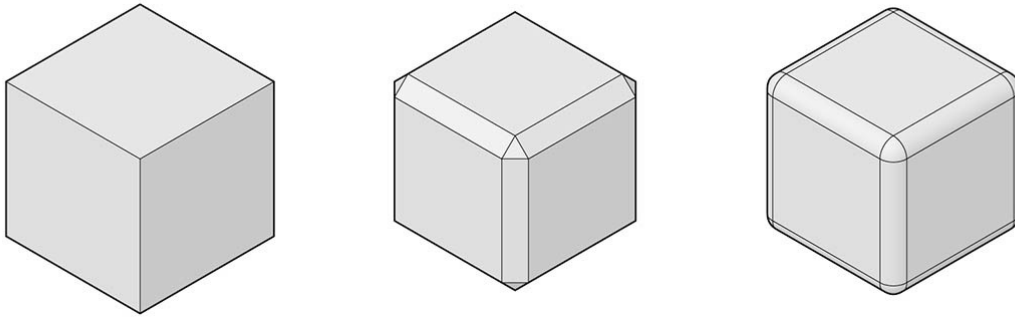
Los sólidos se componen de tres tipos de elementos: vértices, aristas y caras. Las caras son las superficies que conforman el sólido. Las aristas son las curvas que definen la conexión entre las caras adyacentes y los vértices son los puntos inicial y final de esas curvas. Estos elementos se pueden consultar mediante los nodos de topología.



1. Caras
2. Aristas
3. Vértices

Operaciones

Los sólidos se pueden modificar mediante empalmes o achaflanados de sus aristas para eliminar esquinas y ángulos agudos. La operación de chaflán crea una superficie reglada entre dos caras, mientras que un empalme se fusiona entre las caras para mantener la tangencia.



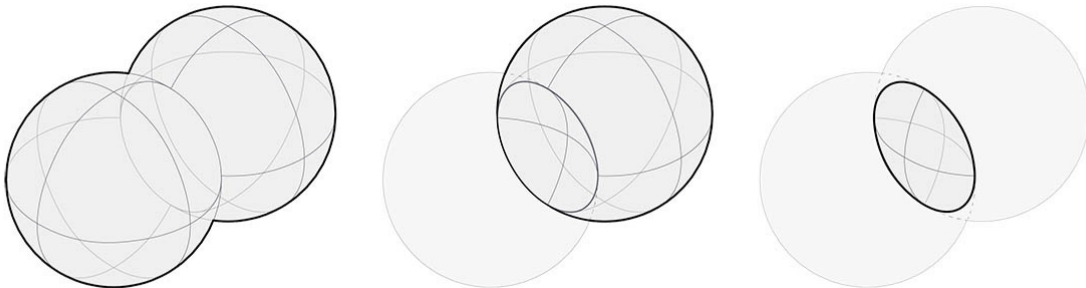
1. Cubo sólido
2. Cubo achaflanado
3. Cubo empalmado

Operaciones booleanas

Las operaciones booleanas de sólidos son métodos para combinar dos o más sólidos. Una única operación booleana implica realmente la realización de cuatro operaciones:

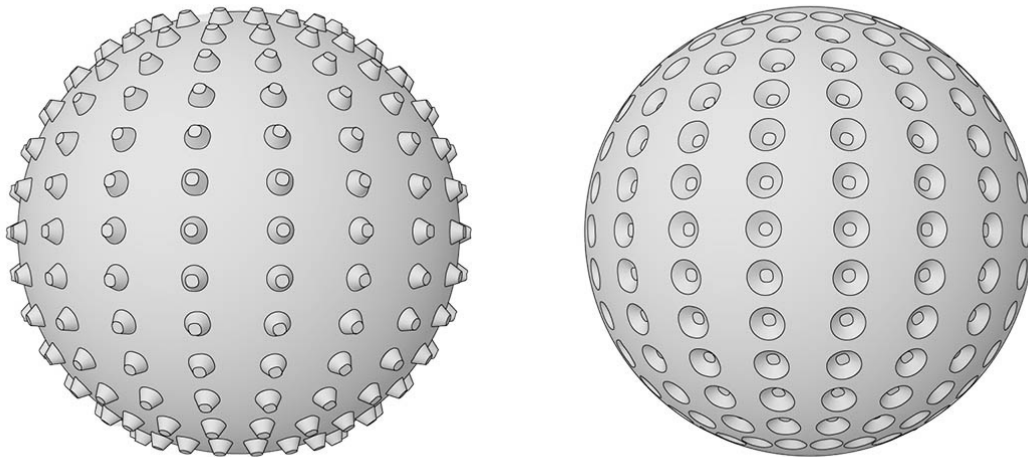
1. **Interseque** dos o más objetos.
2. **Divídalos** en las intersecciones.
3. **Suprime** las partes no deseadas de la geometría.
4. **Una** todo de nuevo.

De este modo, las operaciones booleanas de sólidos permiten ahorrar mucho tiempo. Existen tres operaciones booleanas de sólidos que distinguen las partes de la geometría que se conservan.



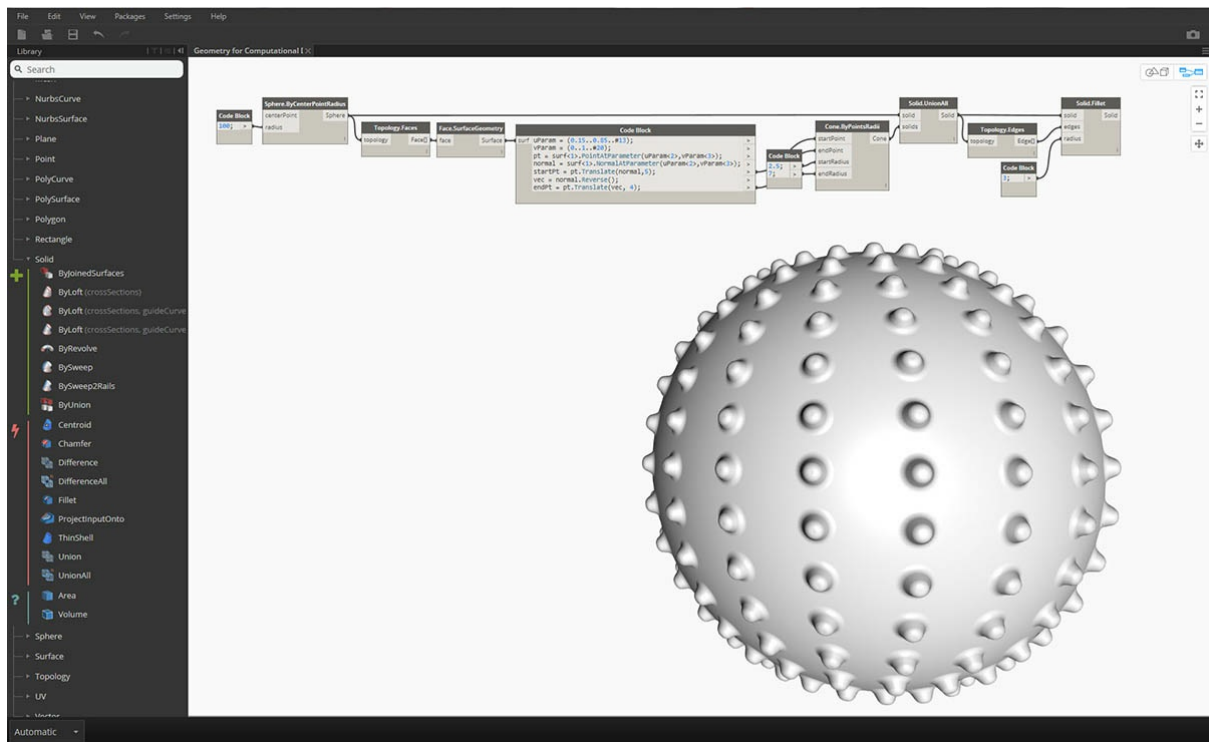
1. **Unión:** elimine las partes que se solapan de los sólidos y únalas en un único sólido.
2. **Diferencia:** reste un sólido a otro. El sólido que se va a restar se conoce como herramienta. Tenga en cuenta que puede cambiar el sólido que será la herramienta para conservar el volumen inverso.
3. **Intersección:** conserve solo el volumen de intersección de los dos sólidos.

Además de estas tres operaciones, Dynamo incluye los nodos **Solid.DifferenceAll** y **Solid.UnionAll** para realizar operaciones de diferencia y unión con varios sólidos.



1. **UnionAll**: operación de unión con esfera y conos orientados hacia fuera.
2. **DifferenceAll**: operación de diferencia con esfera y conos orientados hacia dentro.

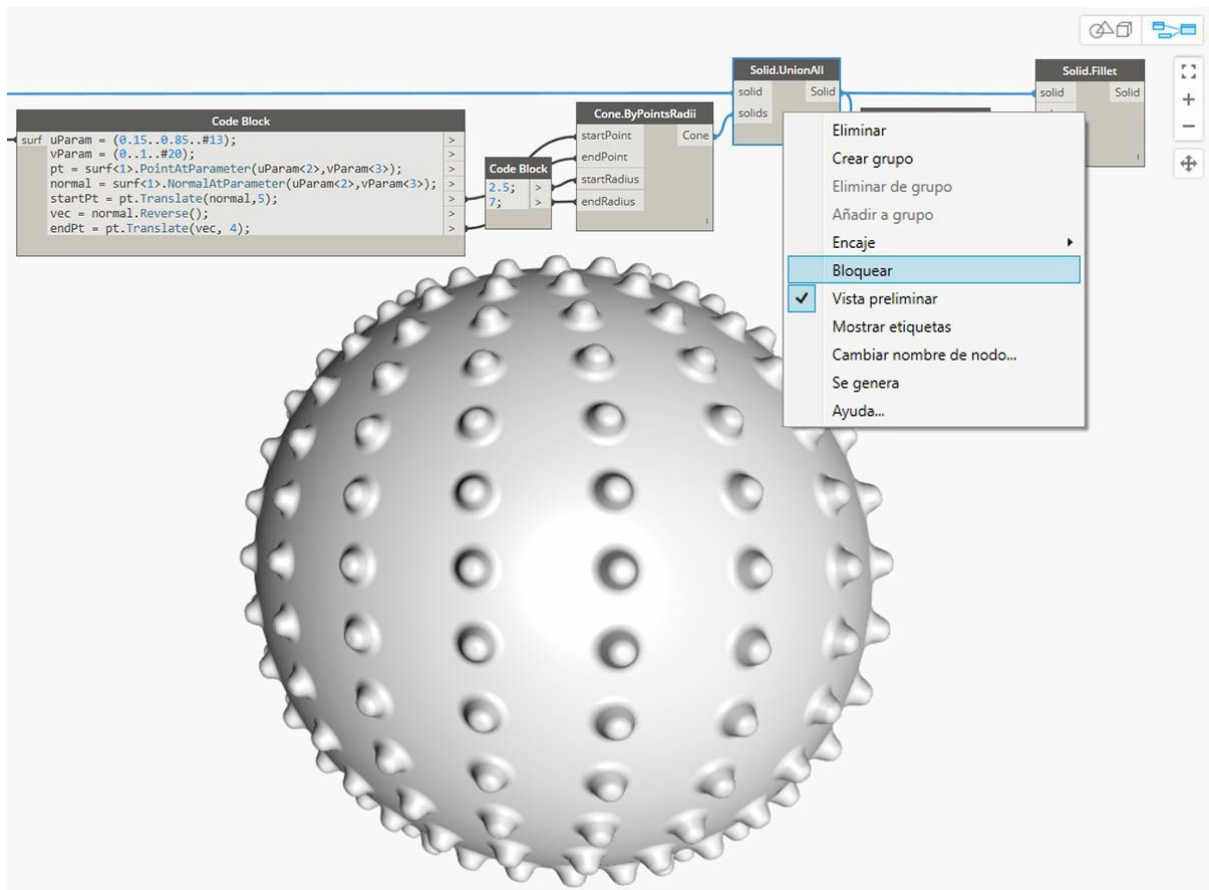
Vamos a utilizar algunas operaciones booleanas para crear una bola llena de puntas.



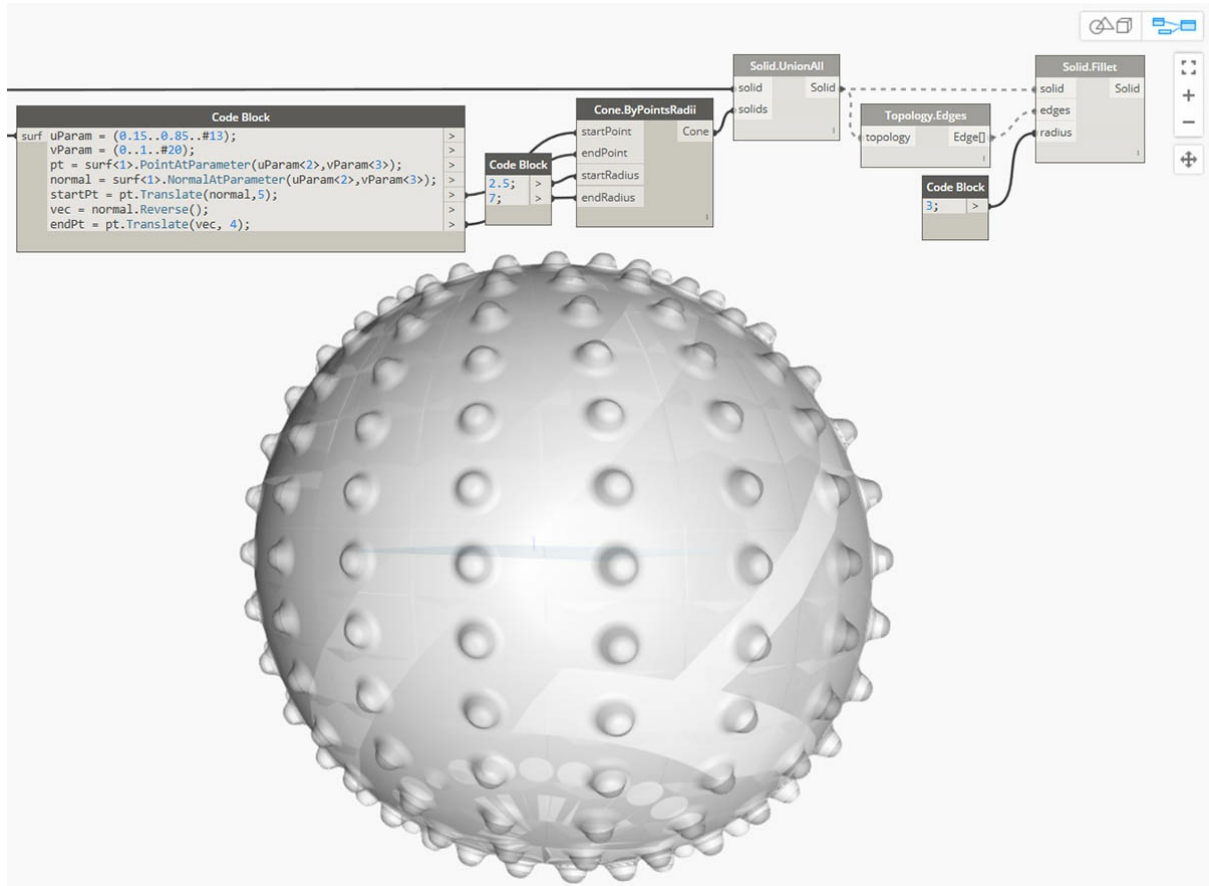
1. **Sphere.ByCenterPointRadius**: cree el sólido base.
2. **Topology.Faces**, **Face.SurfaceGeometry**: consulte las caras del sólido y conviértalas en geometría de superficie; en este caso, la esfera solo tiene una cara.
3. **Cone.ByPointsRadii**: cree conos mediante puntos en la superficie.
4. **Solid.UnionAll**: una los conos y la esfera.
5. **Topology.Edges**: consulte las aristas del nuevo sólido.
6. **Solid.Fillet**: empalme las aristas de la bola llena de puntas. Descargue los archivos de ejemplo que acompañan a esta imagen (haga clic con el botón derecho y seleccione "Guardar enlace como..."). En el Apéndice, se incluye una lista completa de los archivos de ejemplo. [Geometry for Computational Design - Solids.dyn](#)

Bloqueo

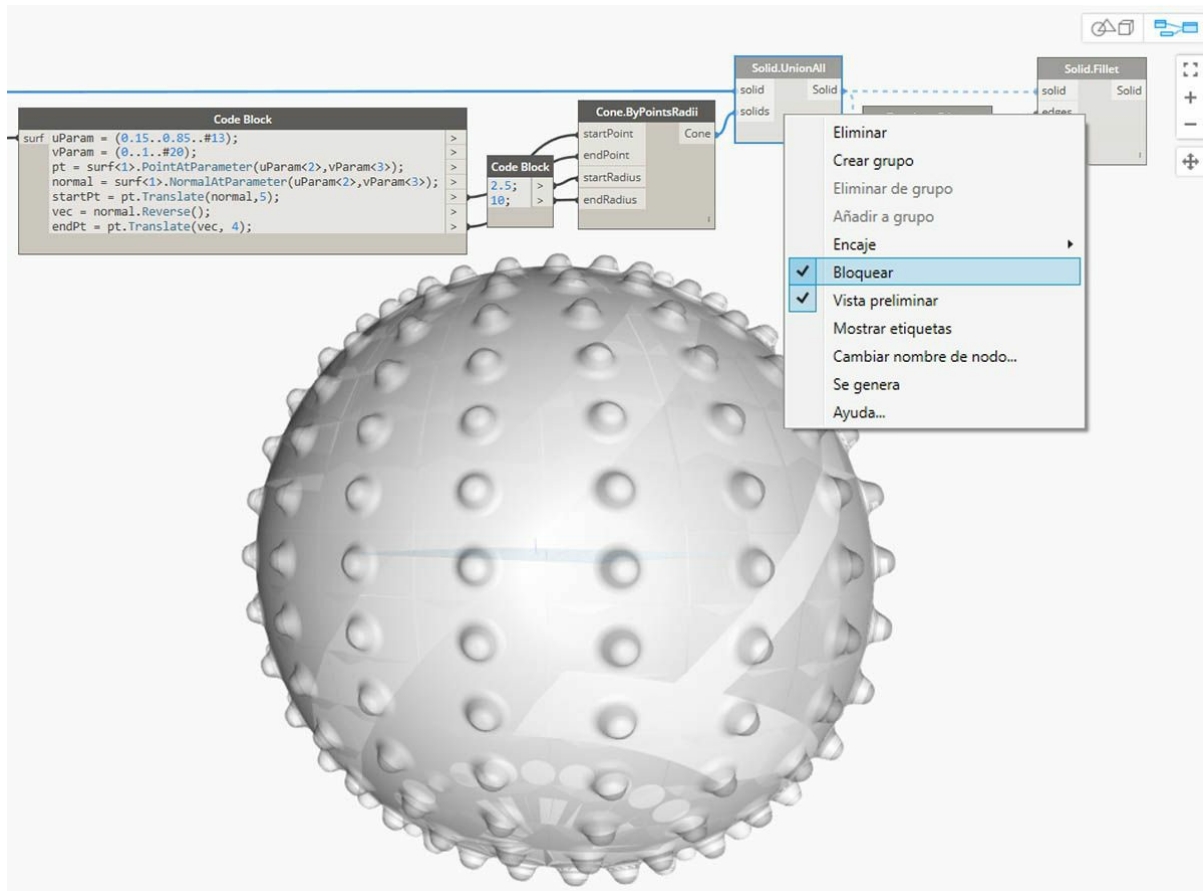
Las operaciones booleanas son complejas y pueden resultar lentas de calcular. Utilice la función Bloquear para suspender la ejecución de nodos seleccionados y nodos descendentes afectados.



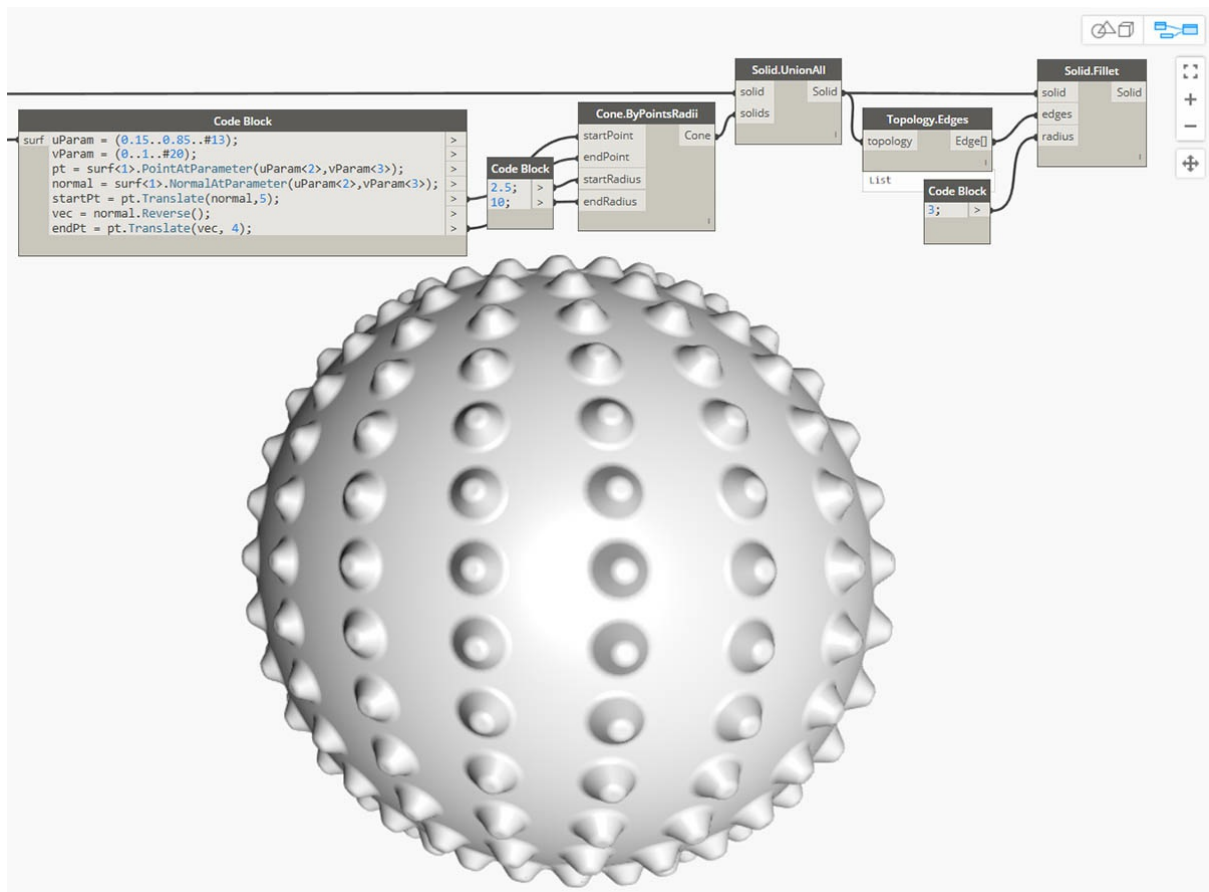
Utilice el menú contextual para bloquear la operación de unión de sólidos.



El nodo seleccionado y todos los nodos descendentes se prevvisualizan en modo atenuado con un color gris claro y los cables afectados se muestran como líneas continuas. La vista preliminar de la geometría afectada también se mostrará atenuada. Ahora puede cambiar los valores ascendentes sin calcular la unión booleana.



Para desbloquear los nodos, haga clic con el botón derecho y desactive la opción Bloquear.



Todos los nodos afectados y las vistas preliminares de la geometría asociada se actualizarán y se restablecerán al modo de vista preliminar estándar.

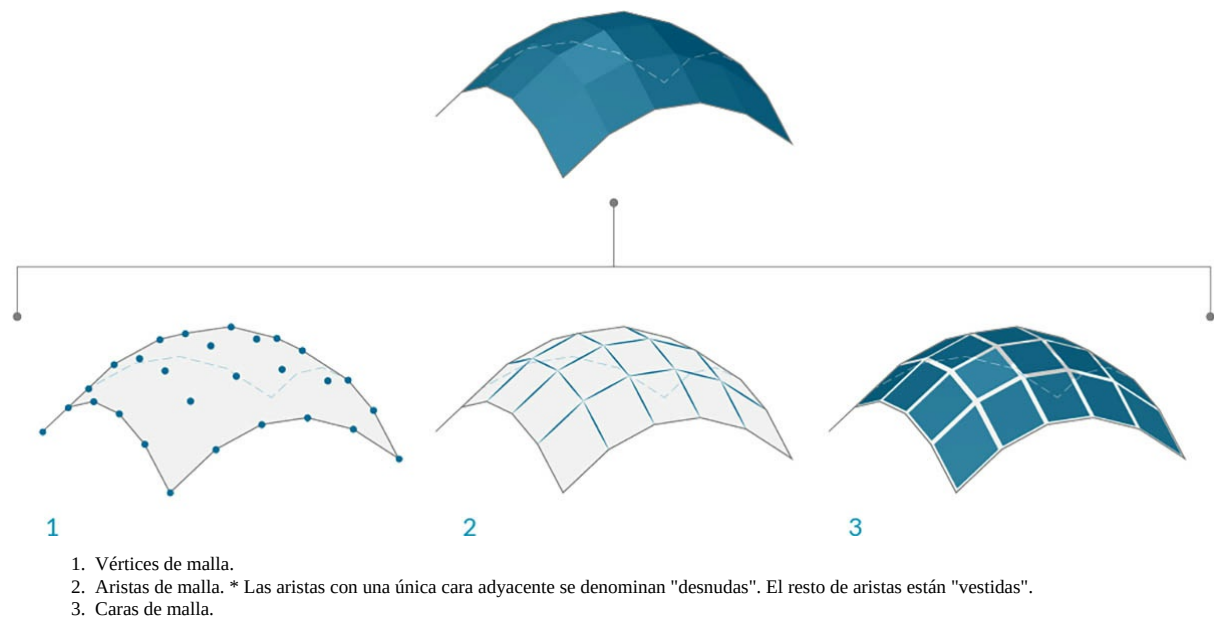
Mallas

Mallas

En el campo del modelado computacional, las mallas son una de las formas más generalizadas de representación de la geometría 3D. La geometría de malla puede ser una alternativa ligera y flexible al uso de NURBS, mientras que las mallas se utilizan en todo, desde la renderización y las visualizaciones hasta la fabricación digital y la impresión 3D.

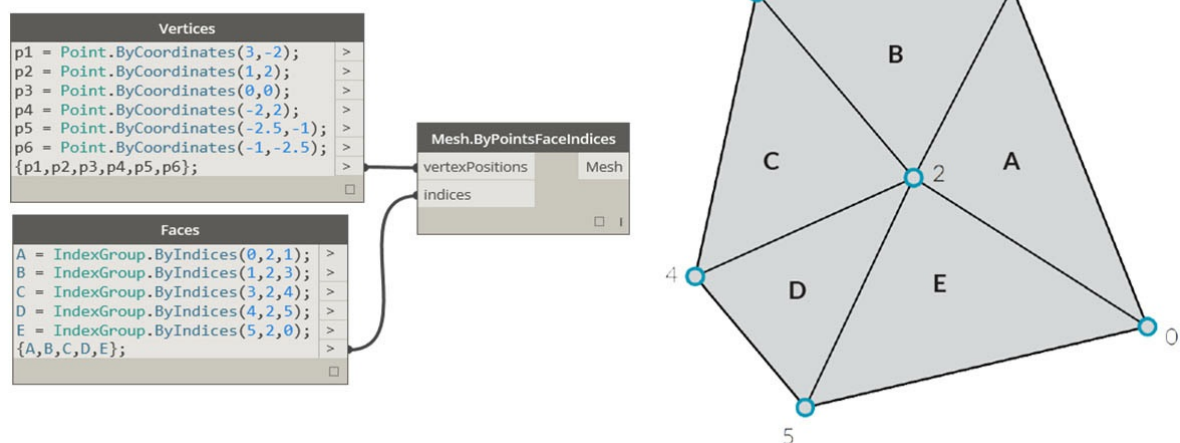
¿Qué es una malla?

Una malla es un conjunto de cuadriláteros y triángulos que representa una superficie o una geometría sólida. Al igual que los sólidos, la estructura de un objeto de malla incluye vértices, aristas y caras. Existen propiedades adicionales que hacen que las mallas sean únicas como, por ejemplo, las normales.



Elementos de malla

Dynamo define mallas mediante una estructura de datos de cara-vértice. En su nivel más básico, esta estructura es simplemente un conjunto de puntos que se agrupan en polígonos. Los puntos de una malla se denominan vértices, mientras que los polígonos similares a superficies se denominan caras. Para crear una malla, se necesita una lista de vértices y un sistema de agrupación de esos vértices en caras denominado grupo de índice.

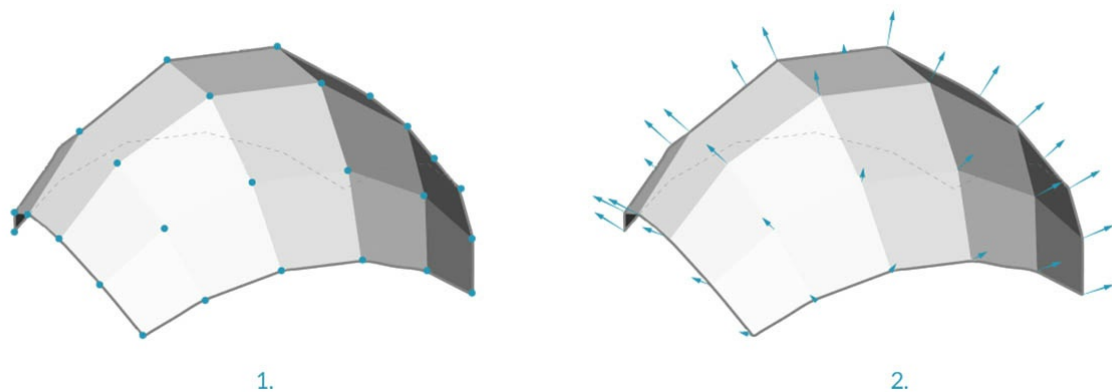


1. Lista de vértices
2. Lista de grupos de índice para definir caras

Vértices + normales de vértice

Los vértices de una malla son simplemente una lista de puntos. El índice de los vértices es muy importante al crear una malla u obtener información sobre la estructura de una malla. Para cada vértice, también existe una normal de vértice (vector) correspondiente que describe la dirección media de las caras

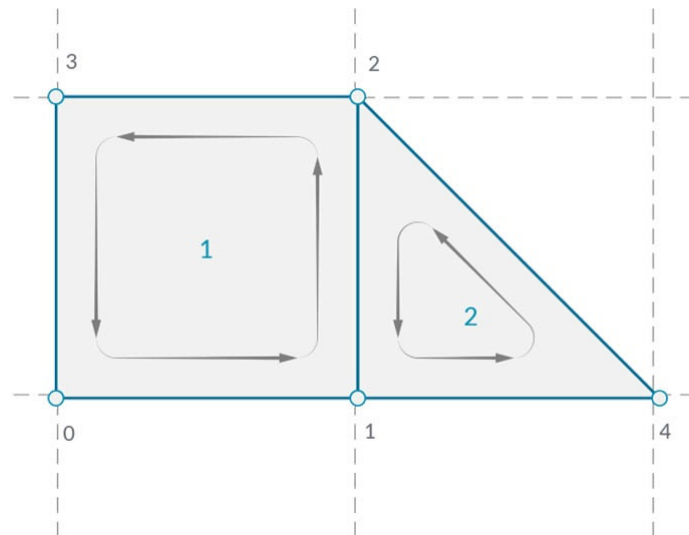
enlazadas y ayuda a comprender la orientación de "entrada" y "salida" de la malla.



1. Vértices
2. Normales de vértice

Caras

Una cara es una lista ordenada de tres o cuatro vértices. Por lo tanto, la representación de la "superficie" de una cara de malla se deduce en función de la posición de los vértices que se están indexando. Ya tenemos la lista de vértices que componen la malla, por lo que, en lugar de proporcionar puntos individuales para definir la malla, solo tenemos que usar el índice de los vértices. Esto también nos permite utilizar el mismo vértice en más de una cara.



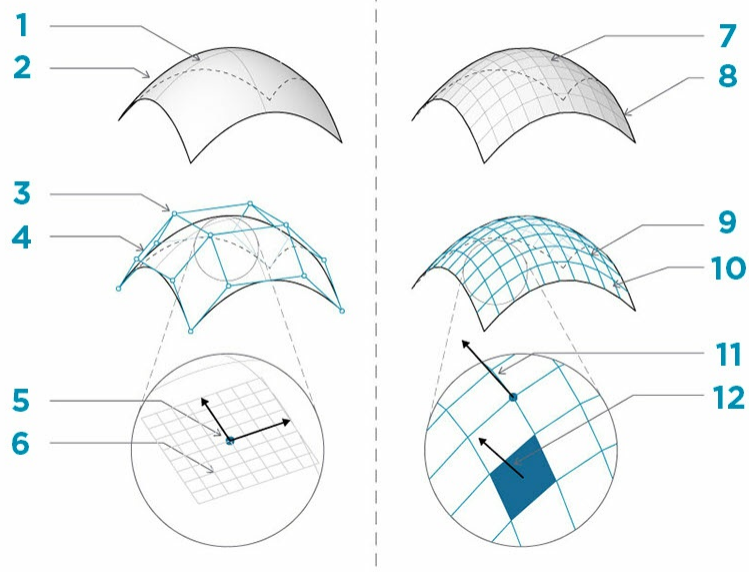
1. Una cara cuadrática compuesta por los índices 0, 1, 2 y 3.
2. Una cara triangular creada con los índices 1, 4 y 2. Tenga en cuenta que se puede cambiar el orden de los grupos de índice; siempre que la secuencia se ordene en el sentido contrario a las agujas del reloj, la cara se definirá correctamente.

Mallas frente a superficies NURBS

¿En qué se diferencia la geometría de malla de la geometría NURBS? ¿Cuándo es recomendable utilizar una en lugar de la otra?

Parametrización

En un capítulo anterior, vimos que las superficies NURBS se definen mediante una serie de curvas NURBS que se dirigen en dos direcciones. Estas direcciones se etiquetan como U y V, y permiten que una superficie NURB se parametrize de acuerdo con un dominio de superficie bidimensional. Las curvas propiamente dichas se almacenan como ecuaciones en el ordenador, lo que permite calcular las superficies resultantes con un grado de precisión arbitrariamente pequeño. Sin embargo, puede ser difícil combinar varias superficies NURBS. La unión de dos superficies NURBS genera una PolySurface, donde las distintas secciones de la geometría tendrán diferentes parámetros UV y definiciones de curva.

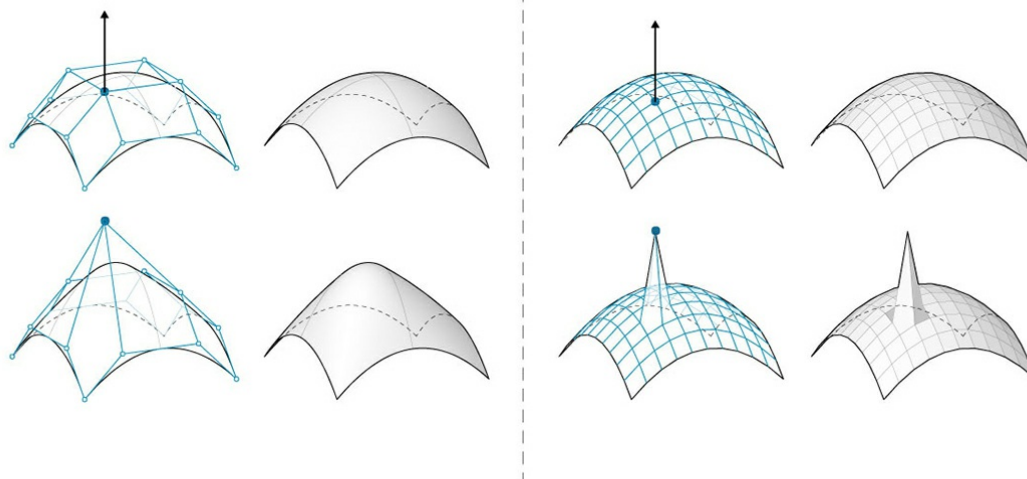


1. Superficie
2. Curva isoparamétrica (línea isoparamétrica)
3. Punto de control de superficie
4. Polígono de control de superficie
5. Punto isométrico
6. Marco de superficie
7. Malla
8. Arista desnuda
9. Red de malla
10. Aristas de malla
11. Normal de vértice
12. Cara de malla/normal de cara de malla

Por otra parte, las mallas están formadas por un número específico de caras y vértices definidos exactamente. La red de vértices no se puede definir normalmente mediante coordenadas UV sencillas, y como las caras son independientes, la cantidad de precisión se incorpora en la malla y solo se puede cambiar mediante el ajuste preciso de la malla y la adición de más caras. La falta de descripciones matemáticas permite que las mallas gestionen con mayor flexibilidad geometrías complejas dentro de una única malla.

Influencia local frente a global

Otra diferencia importante es la extensión en la que un cambio local en la geometría de malla o NURBS afecta a toda la forma. El desplazamiento de un vértice de una malla solo afecta a las caras adyacentes a ese vértice. En las superficies NURBS, la extensión de la influencia es más complicada y depende del grado de la superficie, así como de los grosores y los nudos de los puntos de control. Sin embargo, por lo general, al desplazar un único punto de control en una superficie NURBS, se produce un cambio más suave y extensivo en la geometría.



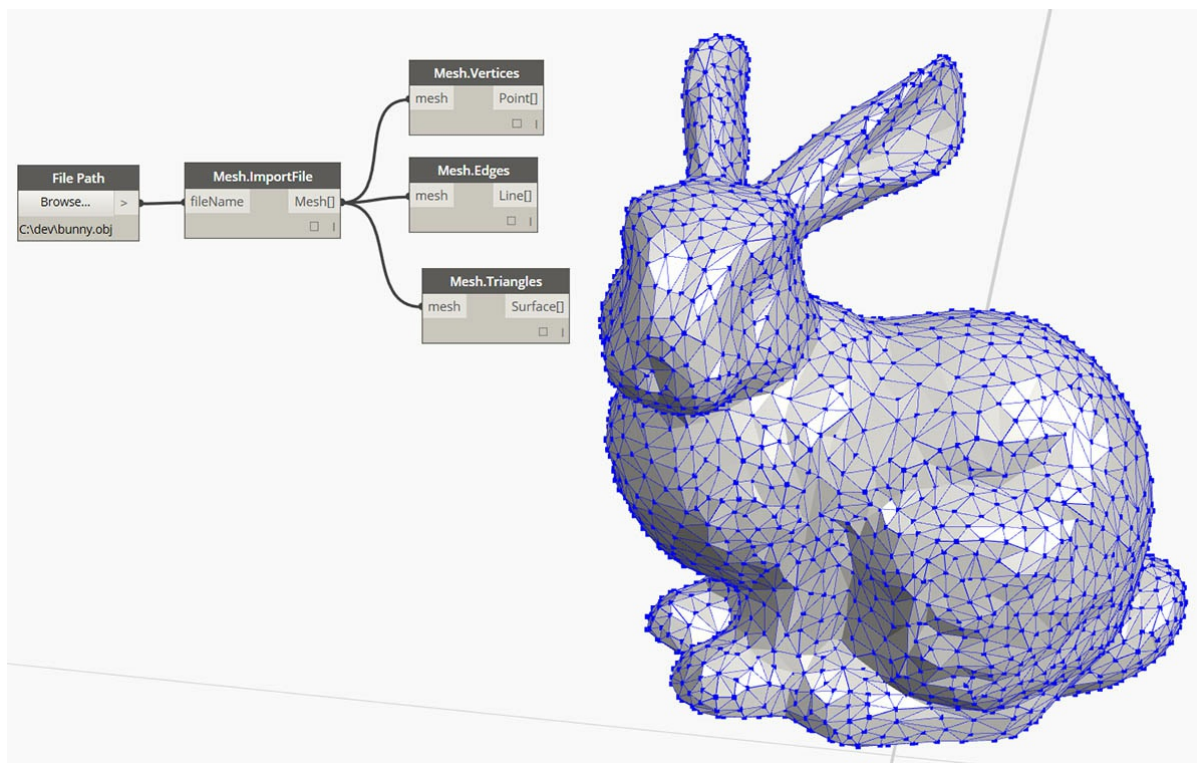
1. Superficie NURBS: el desplazamiento de un punto de control tiene una influencia que se extiende a lo largo de la forma.
2. Geometría de malla: el desplazamiento de un vértice solo influye en los elementos adyacentes.

Una analogía que puede resultar útil es comparar una imagen vectorial (compuesta de líneas y curvas) con una imagen ráster (compuesta de píxeles individuales). Si amplía la vista de una imagen vectorial, las curvas permanecen nítidas y claras, mientras que si se amplía la vista de una imagen ráster, los píxeles individuales aumentan de tamaño. En esta analogía, las superficies NURBS se pueden comparar con una imagen vectorial porque existe una relación matemática fluida, mientras que una malla se comporta de forma similar a una imagen ráster con una resolución establecida.

Kit de herramientas de malla

Las funciones de malla de Dynamo se pueden ampliar mediante la instalación del paquete de [Kit de herramientas de malla](#). El Kit de herramientas de malla de Dynamo proporciona herramientas para importar mallas desde formatos de archivo externos, crear una malla a partir de objetos de geometría de Dynamo y generar manualmente mallas mediante sus vértices e índices. La biblioteca también proporciona herramientas para modificar y reparar mallas, o extraer cortes horizontales para su uso en la fabricación.

Consulte el capítulo 10.2 para ver un ejemplo mediante el Kit de herramientas de malla.



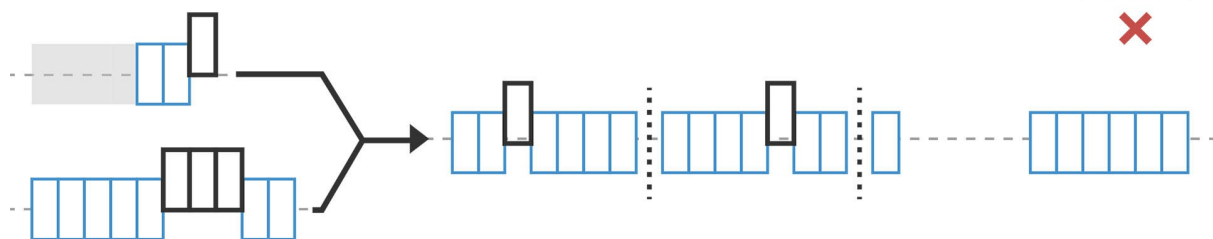
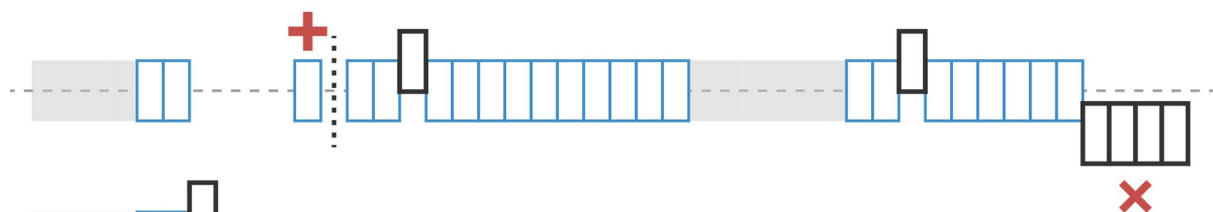
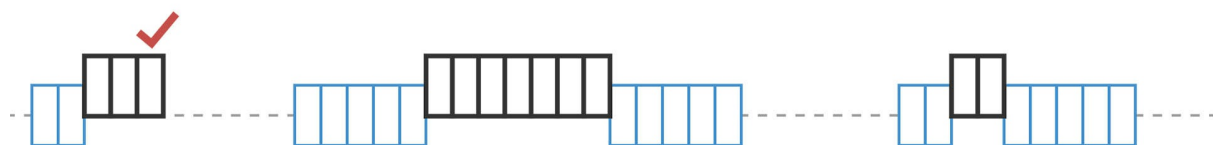
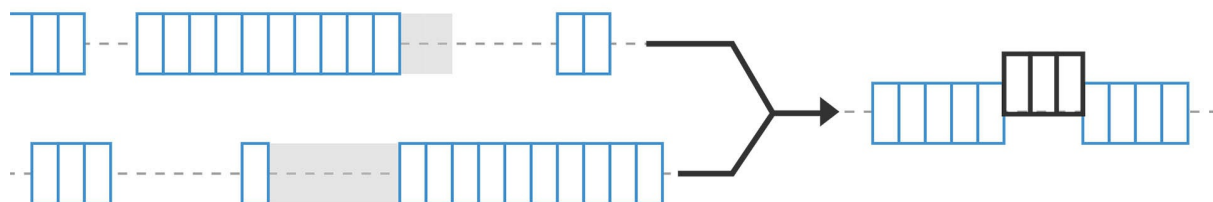
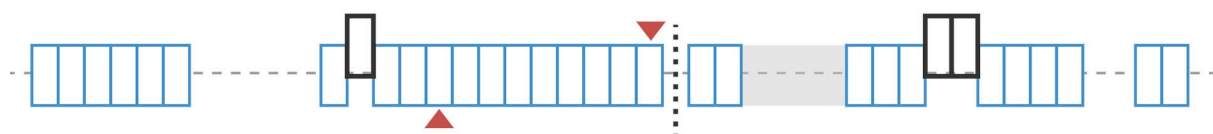
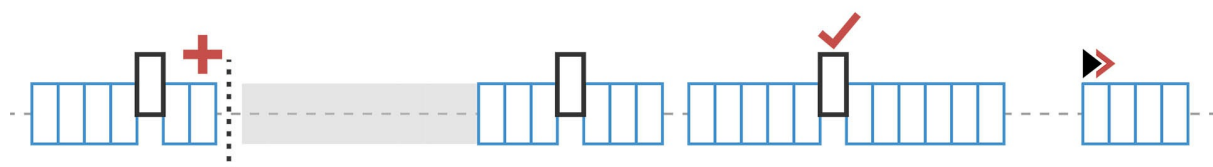
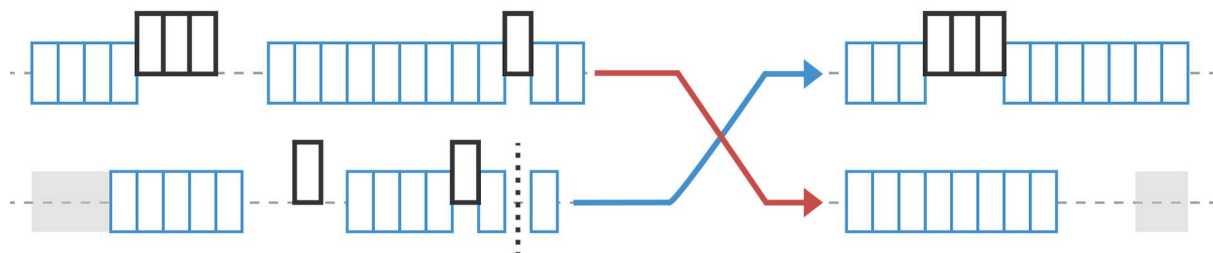
Importación de geometría

Importación de geometría

Diseño con listas

Diseño con listas

Las listas son la forma en que organizamos los datos. En el sistema operativo del equipo, tiene archivos y carpetas. En Dynamo, podemos considerarlos como elementos y listas respectivamente. Al igual que su sistema operativo, existen muchas formas de crear, modificar y consultar datos. En este capítulo, vamos a desglosar el modo en que se gestionan las listas en Dynamo.



¿Qué es una lista?

¿Qué es una lista?

Una lista es un conjunto de elementos o ítems. Pensemos en un racimo de plátanos, por ejemplo. Cada plátano es un elemento de la lista (o racimo). Es más fácil coger un racimo de plátanos que cada plátano individualmente, y lo mismo se aplica a la agrupación de elementos por relaciones paramétricas en una estructura de datos.



Foto de [Augustus Binu](#).

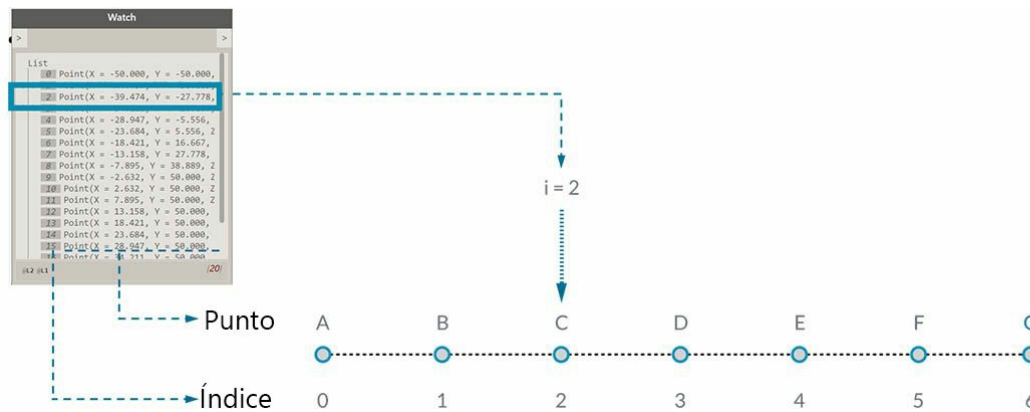
Cuando hacemos la compra, ponemos todos los artículos comprados en una bolsa. Esta bolsa también es una lista. Si vamos a hacer pan de plátano, necesitamos tres racimos de plátanos (vamos a hacer *mucho* pan de plátano). La bolsa representa una lista de racimos de plátanos y cada racimo representa una lista de plátanos. La bolsa es una lista de listas (bidimensional) y el plátano es una lista (unidimensional).

En Dynamo, los datos de las listas se ponen en orden y el primer elemento de cada lista tiene un índice "0". A continuación, vamos a ver cómo se definen las listas en Dynamo y cómo se relacionan varias listas entre sí.

Índices basados en cero

Algo que podría parecer extraño al principio es que el primer índice de una lista es siempre 0; no 1. Por lo tanto, cuando hablamos del primer elemento de una lista, en realidad nos referimos al elemento que corresponde al índice 0.

Por ejemplo, si tuviéramos que contar el número de dedos que tenemos en la mano derecha, es probable que contáramos del 1 al 5. Sin embargo, si tuviéramos que poner los dedos en una lista, Dynamo les asignaría los índices del 0 al 4. Aunque esto puede parecer un poco extraño para programadores principiantes, el índice de base cero es una práctica estándar en la mayoría de los sistemas de cálculo.



Elemento

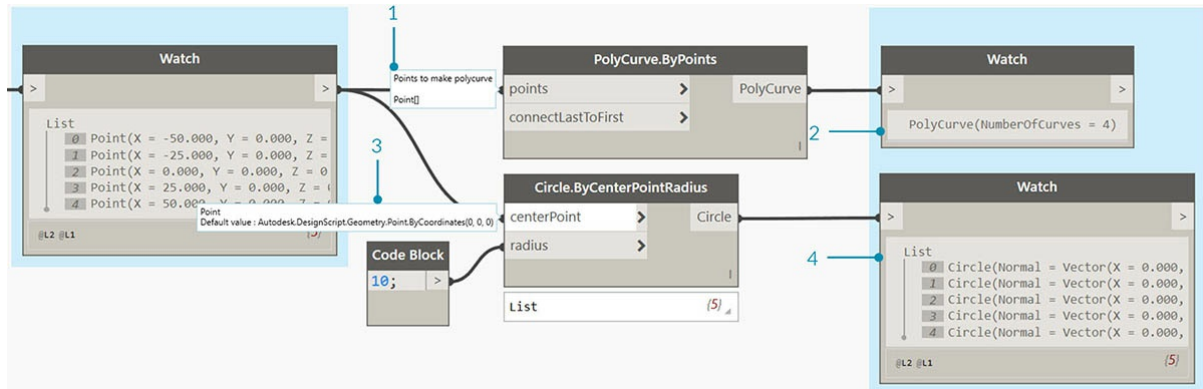
Tenga en cuenta que sigue habiendo cinco elementos en la lista; solo que la lista utiliza un sistema de recuento basado en cero. Y los elementos que se almacenan en la lista no solo pueden ser números. Pueden ser cualquier tipo de datos compatible con Dynamo, como puntos, curvas, superficies, familias, etc.

A menudo, la forma más sencilla de echar un vistazo al tipo de datos almacenados en una lista es conectar un nodo Watch con la salida de otro nodo. Por defecto, el nodo Watch muestra automáticamente todos los índices en el lado izquierdo de la lista y los elementos de datos en el lado derecho.

Estos índices son elementos decisivos cuando se trabaja con listas.

Entradas y salidas

Las entradas y las salidas, que pertenecen a las listas, varían en función del nodo de Dynamo que se utilice. Como ejemplo, vamos a utilizar una lista de cinco puntos y a conectar esta salida a dos nodos de Dynamo diferentes: *PolyCurve.ByPoints* y *Circle.ByCenterPointRadius*:



1. La entrada *points* de *PolyCurve.ByPoints* busca "Point[]". Esto representa una lista de puntos.
2. La salida de *PolyCurve.ByPoints* es una *PolyCurve* única creada a partir de una lista de cinco puntos.
3. La entrada *centerPoint* de *Circle.ByCenterPointRadius* solicita "Point".
4. La salida de *Circle.ByCenterPointRadius* es una lista de cinco círculos cuyos centros corresponden a la lista original de puntos.

Los datos de entrada de *PolyCurve.ByPoints* y *Circle.ByCenterPointRadius* son los mismos; sin embargo, el nodo *PolyCurve* nos proporciona una *PolyCurve*, mientras que el nodo *Círculo* nos proporciona cinco círculos con centros en cada punto. Esto resulta intuitivo: la *PolyCurve* se dibuja como una curva que conecta los 5 puntos, mientras que los círculos crean un círculo distinto en cada punto. Entonces, ¿qué ocurre con los datos?

Al pasar el cursor sobre la entrada *points* de *Polycurve.ByPoints*, vemos que la entrada busca "Point[]". Observe los corchetes que aparecen al final. Este elemento representa una lista de puntos y, para crear una *PolyCurve*, la entrada debe ser una lista para cada *PolyCurve*. Por tanto, este nodo condensará cada lista en una *PolyCurve*.

Por otra parte, la entrada *centerPoint* de *Circle.ByCenterPointRadius* solicita "Point". Este nodo busca un punto, como elemento, para definir el centro del círculo. Por este motivo, se obtienen cinco círculos a partir de los datos de entrada. Reconocer esta diferencia en las entradas de Dynamo ayuda a comprender mejor cómo funcionan los nodos al administrar los datos.

Encaje

La correspondencia de datos es un problema sin una solución definitiva. Se produce cuando un nodo tiene acceso a entradas de distinto tamaño. El cambio del algoritmo de correspondencia de datos puede dar lugar a resultados muy diferentes.

Imagine un nodo que crea segmentos de línea entre puntos (*Line.ByStartPointEndPoint*). Tendrá dos parámetros de entrada y ambos proporcionarán

coordenadas de punto:

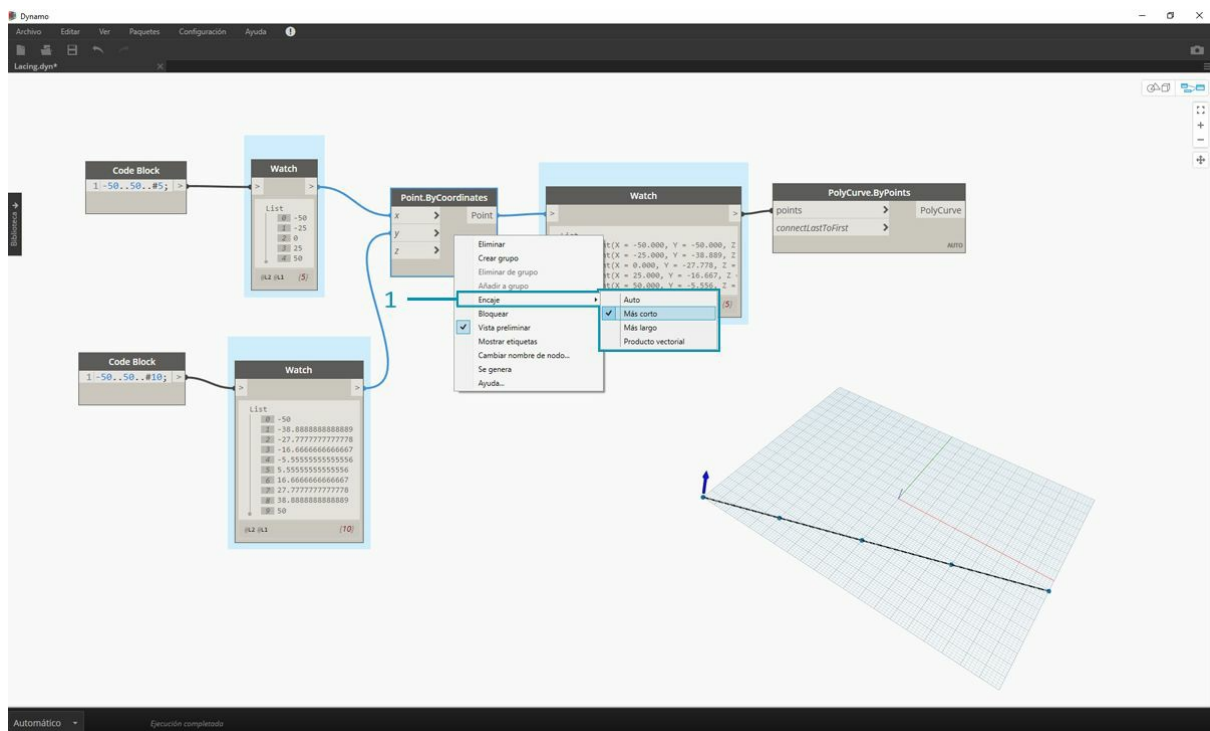


Como puede ver, existen diferentes métodos para dibujar líneas entre estos conjuntos de puntos. Las opciones de encaje se encuentran haciendo clic con el botón derecho en el centro de un nodo y eligiendo el menú "Encaje".

Archivo base

Descargue el archivo de ejemplo que acompaña a este ejercicio (haga clic con el botón derecho y seleccione "Guardar enlace como..."): [Lacing.dyn](#). En el Apéndice se incluye una lista completa de los archivos de ejemplo.

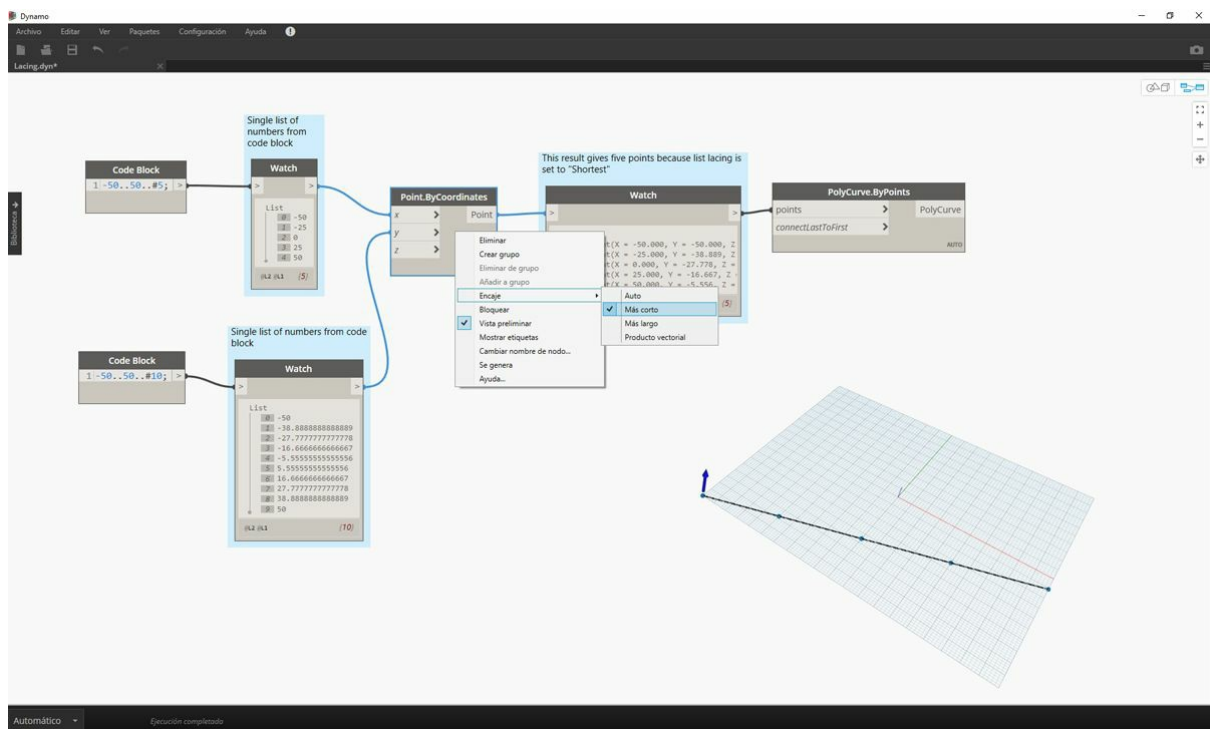
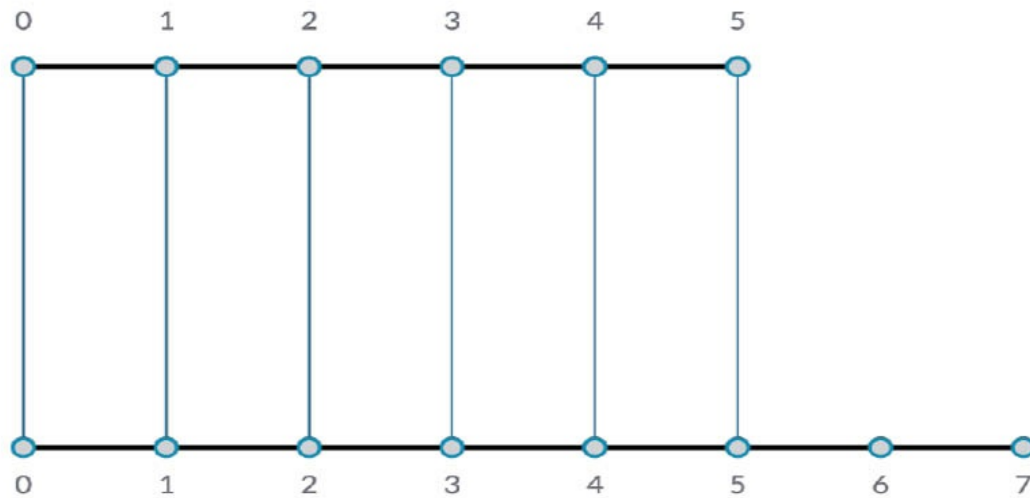
Para realizar una demostración de las operaciones de encaje desarrolladas a continuación, utilizaremos este archivo base para definir la lista más corta, la más larga y el producto cartesiano.



1. Cambiaremos el encaje en `Point.ByCoordinates`, pero no cambiaremos nada más en el gráfico anterior.

Lista más corta

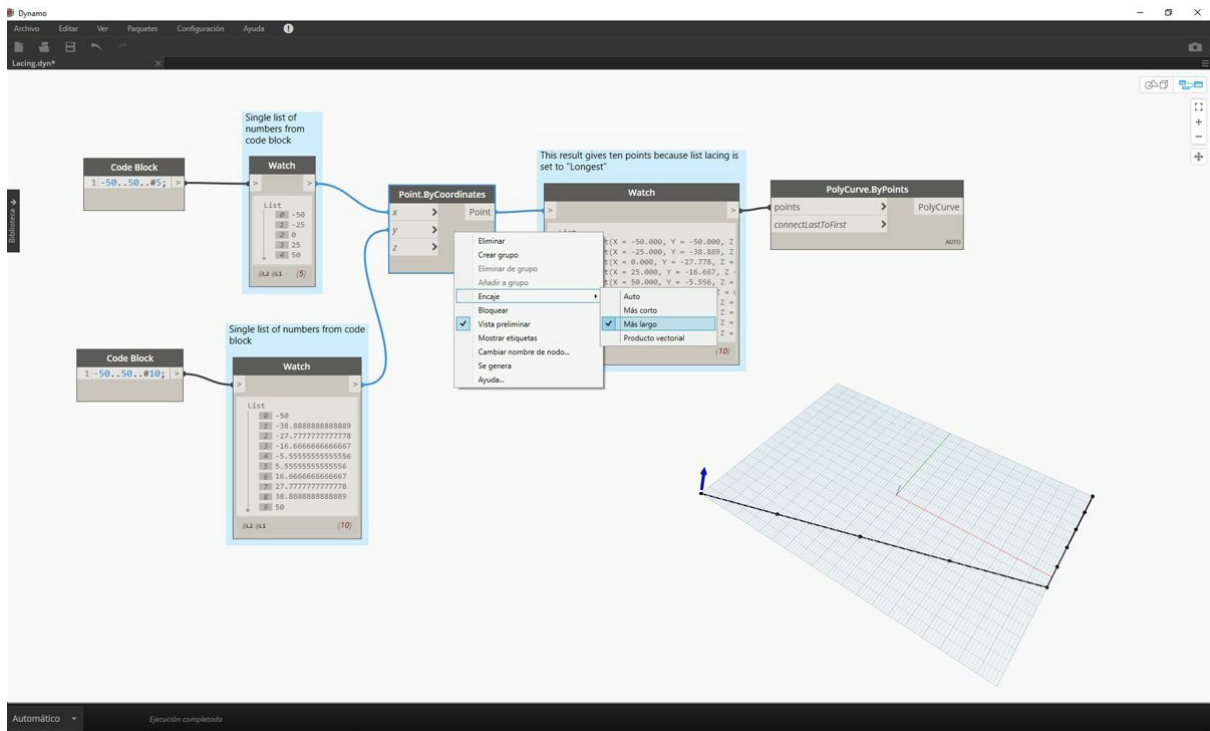
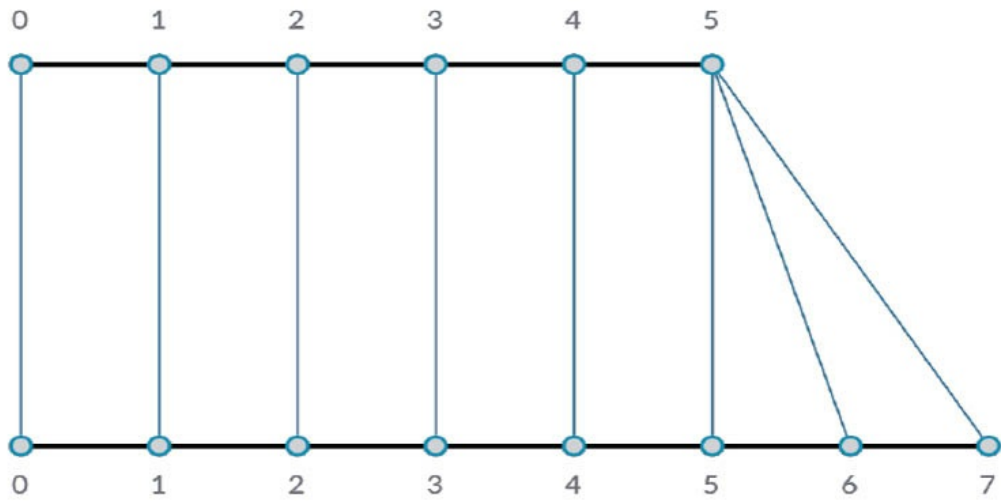
La forma más sencilla es conectar las entradas una a una hasta que uno de los flujos se acabe. Esto se denomina algoritmo "Lista más corta". Este es el comportamiento por defecto de los nodos de Dynamo:



Al cambiar el encaje a la *lista más corta*, obtenemos una línea diagonal básica compuesta de cinco puntos. Cinco puntos es la longitud de la lista menor, de modo que el encaje de la lista más corta se detiene cuando alcanza el final de una lista.

Lista más larga

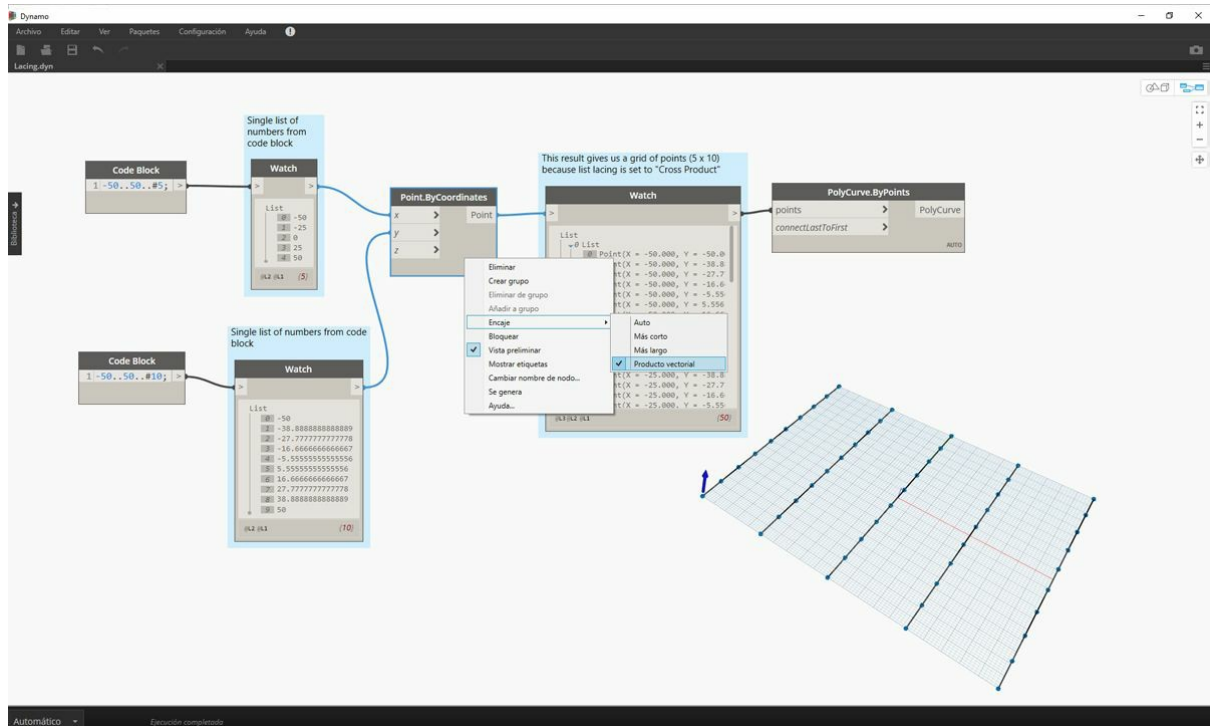
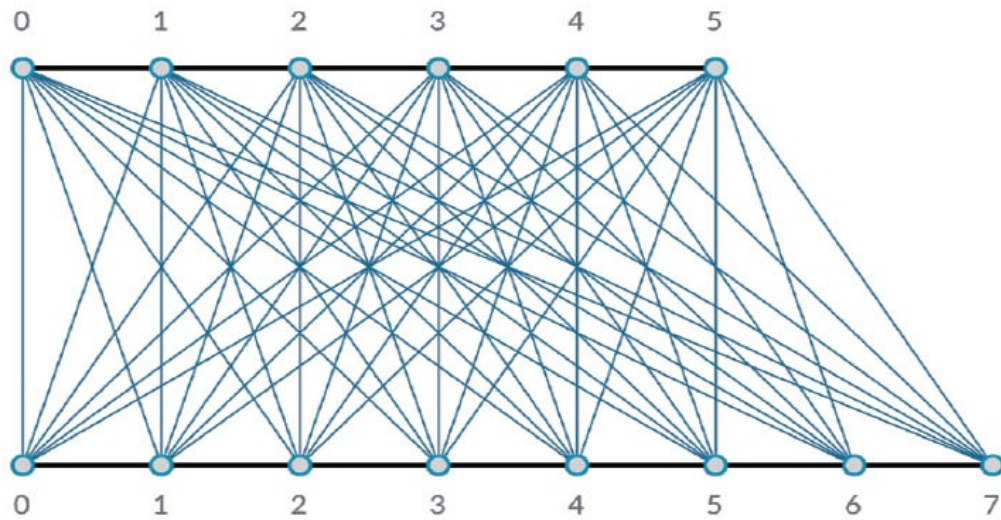
El algoritmo "Lista más larga" sigue conectando entradas, reutilizando elementos, hasta que todos los flujos se acaben:



Al cambiar el encaje a la *lista más larga*, obtenemos una línea diagonal que se extiende verticalmente. Con el mismo método que el diagrama conceptual, el último elemento de la lista de 5 elementos se repetirá para alcanzar la longitud de la lista más larga.

Producto cartesiano

Por último, el método "Producto cartesiano" hace todas las conexiones posibles:



Al cambiar el enlace a *Producto cartesiano*, obtenemos todas las combinaciones entre cada lista, lo que nos proporciona una rejilla de puntos de 5 x 10. Esta es una estructura de datos equivalente al producto cartesiano que se muestra en el diagrama de conceptos anterior, excepto que nuestros datos son ahora una lista de listas. Al conectar una PolyCurve, podemos ver que cada lista está definida por su valor X, lo que nos da una fila de líneas verticales.

Trabajo con listas

Trabajo con listas

Ahora que hemos establecido lo que es una lista, hablemos de las operaciones que podemos realizar en ella. Imagine una lista como un mazo de cartas. El mazo es la lista y cada carta representa un elemento.



Foto de [Christian Gidlöf](#)

¿Qué **consultas** se pueden realizar a partir de la lista? Esto permite acceder a las propiedades existentes.

- ¿Número de cartas del mazo? 52.
- ¿Número de palos? 4.
- ¿Material? Papel.
- ¿Longitud? 3,5" u 89 mm.
- ¿Anchura? 2,5" o 64 mm.

¿Qué **acciones** se pueden realizar en la lista? Estas acciones cambian la lista en función de una operación especificada.

- Podemos barajar el mazo de cartas.
- Podemos ordenar el mazo por valor.
- Podemos ordenar el mazo por palo.
- Podemos dividir el mazo.
- Podemos dividir el mazo repartiendo manos individuales.
- Podemos seleccionar una carta específica del mazo.

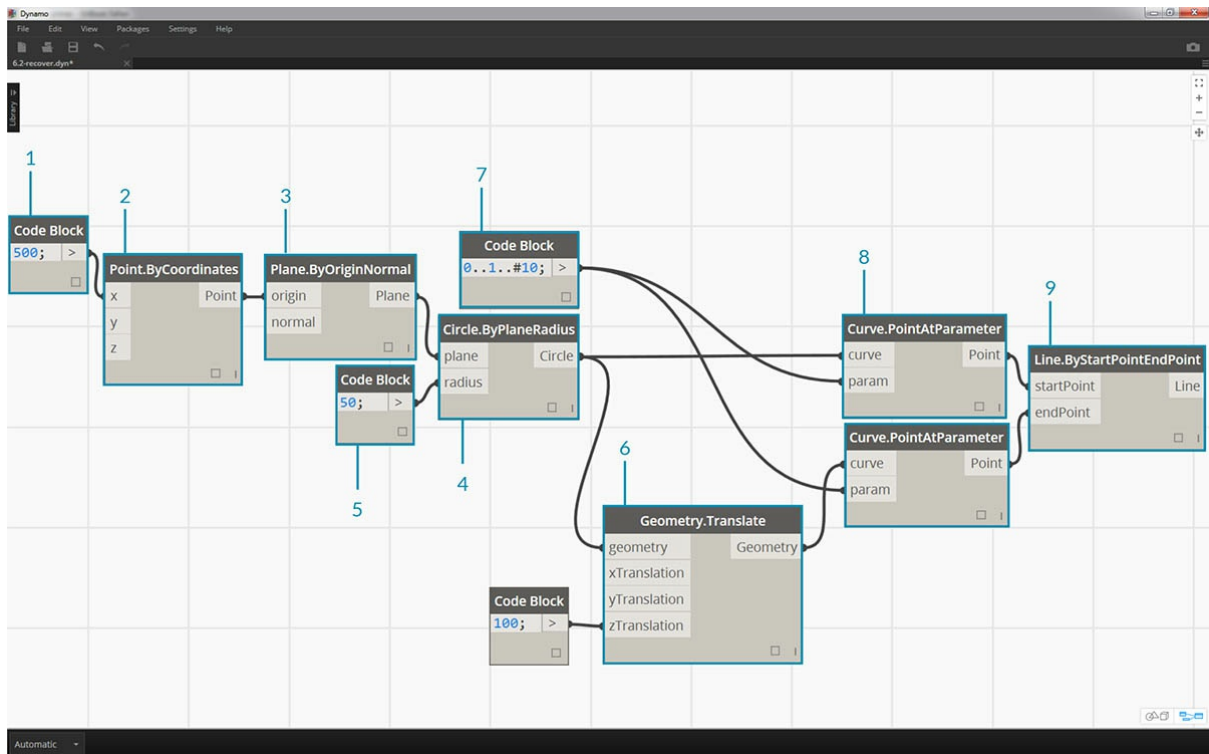
Todas las operaciones mencionadas anteriormente tienen nodos análogos de Dynamo para trabajar con listas de datos genéricos. Las lecciones siguientes muestran algunas de las operaciones fundamentales que podemos realizar en las listas.

Operaciones de lista

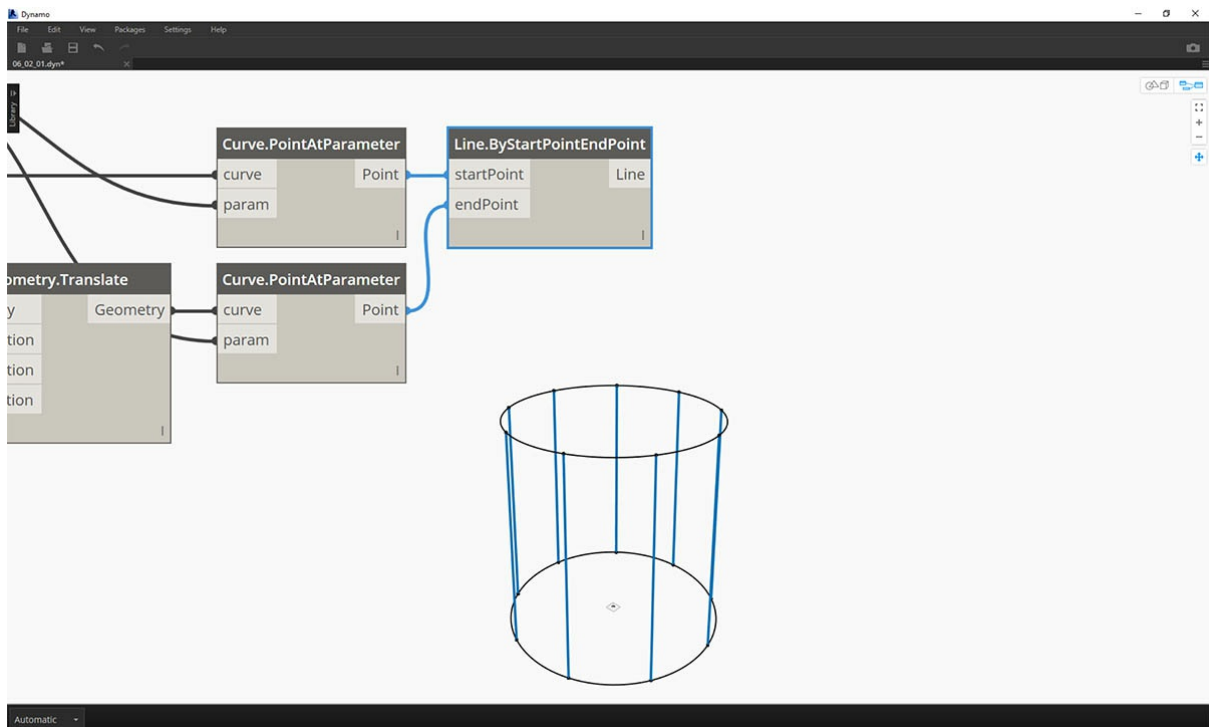
La imagen siguiente es el gráfico base que utilizaremos para ilustrar las operaciones básicas de lista. Exploraremos cómo gestionar los datos de una lista y mostraremos los resultados visuales.

Ejercicio: lista de operaciones

Descargue el archivo de ejemplo que acompaña a este ejercicio (haga clic con el botón derecho y seleccione "Guardar enlace como..."): [List-Operations.dyn](#). En el Apéndice se incluye una lista completa de los archivos de ejemplo.

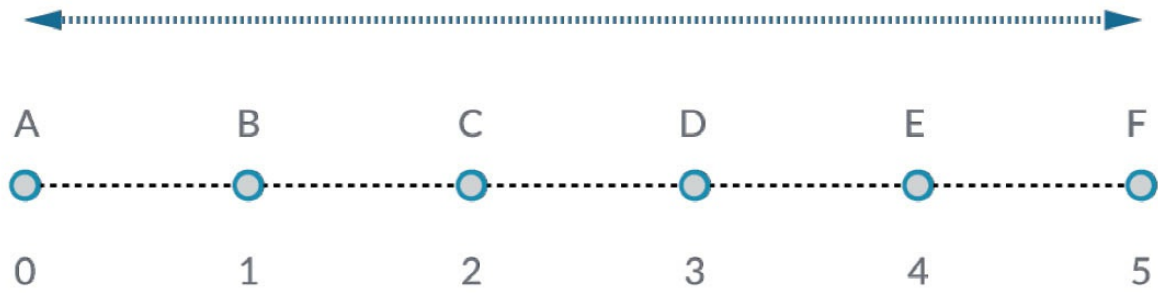


1. Comience con un *bloque de código* con un valor de `500;`;
2. Conéctelo a la entrada *x* de un nodo *Point.ByCoordinates*.
3. Conecte el nodo del paso anterior a la entrada *origin* de un nodo *Plane.ByOriginNormal*.
4. Con un nodo *Circle.ByPlaneRadius*, conecte el nodo del paso anterior en la entrada *plane*.
5. Mediante el *bloque de código*, designe un valor de `50;` para la entrada *radius*. Este es el primer círculo que crearemos.
6. Con un nodo *Geometry.Translate*, desplace el círculo hacia arriba 100 unidades en la dirección *Z*.
7. Con un nodo *Code Block*, defina un rango de diez números entre 0 y 1 con esta línea de código: `0..1..#10;`;
8. Conecte el bloque de código del paso anterior a la entrada *param* de dos nodos *Curve.PointAtParameter*. Conecte *Circle.ByPlaneRadius* a la entrada *curve* del nodo superior y *Geometry.Translate* a la entrada *curve* del nodo situado debajo del mismo.
9. Con un nodo *Line.ByStartPointEndPoint*, conecte los dos nodos *Curve.PointAtParameter*.



1. Un nodo *Watch3D* muestra los resultados de *Line.ByStartPointEndPoint*. Estamos dibujando líneas entre dos círculos para representar operaciones de lista básicas y utilizaremos este gráfico base de Dynamo para pasar por las acciones de lista que se indican a continuación.

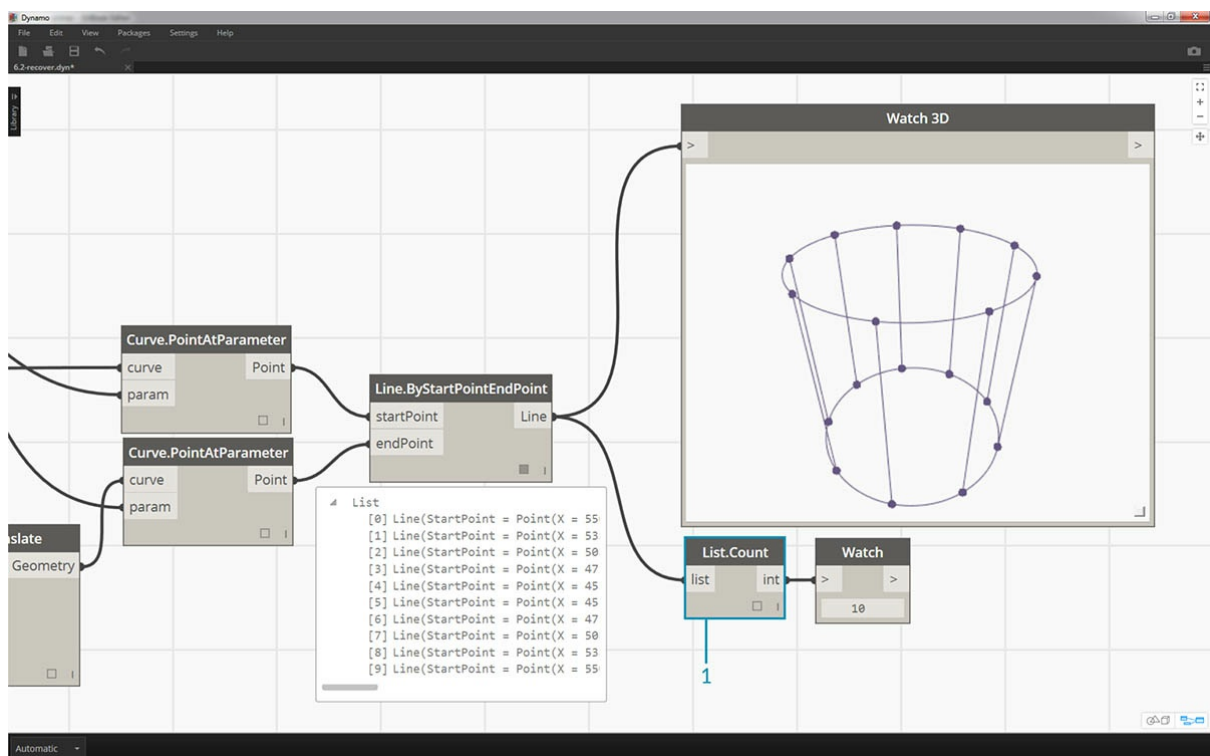
List.Count



El nodo *List.Count* es sencillo: cuenta el número de valores de una lista y devuelve ese número. Este nodo presenta más detalles cuando se trabaja con listas de listas, pero eso lo veremos en las secciones siguientes.

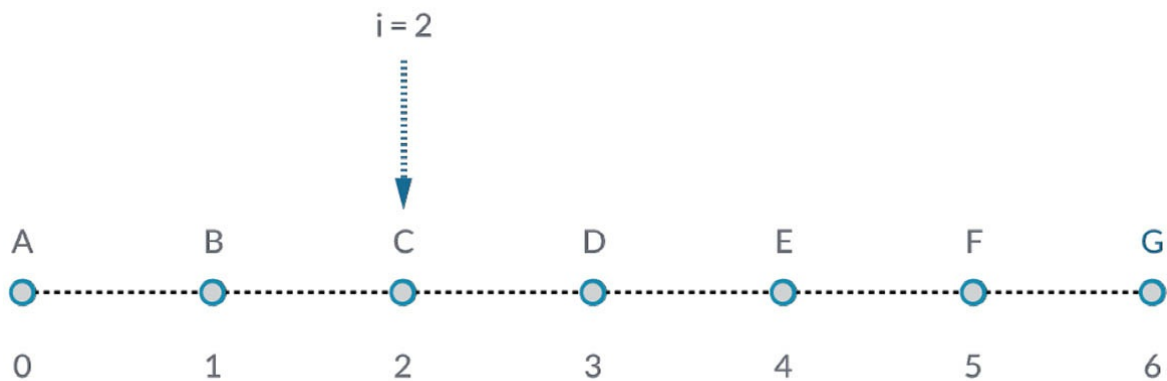
Ejercicio: List.Count

Descargue el archivo de ejemplo que acompaña a este ejercicio (haga clic con el botón derecho y seleccione "Guardar enlace como..."): [List.Count.dyn](#). En el Apéndice se incluye una lista completa de los archivos de ejemplo.



1. El nodo *List.Count* devuelve el número de líneas del nodo *Line.ByStartPointEndPoint*. En este caso, el valor es 10, lo cual coincide con el número de puntos creados a partir del nodo *Code Block* original.

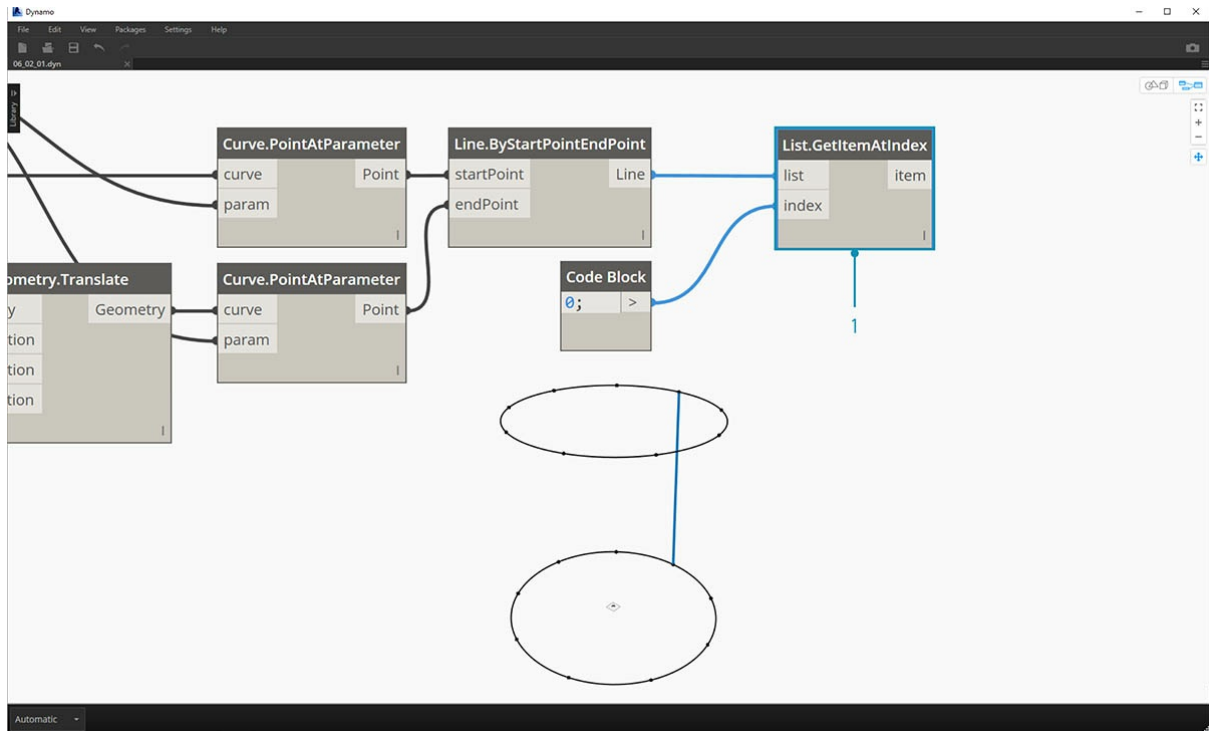
List.GetItemAtIndex



List.GetItemAtIndex es una forma fundamental de consultar un elemento de la lista. En la imagen anterior, se utiliza un índice de "2" para consultar el punto etiquetado como "C".

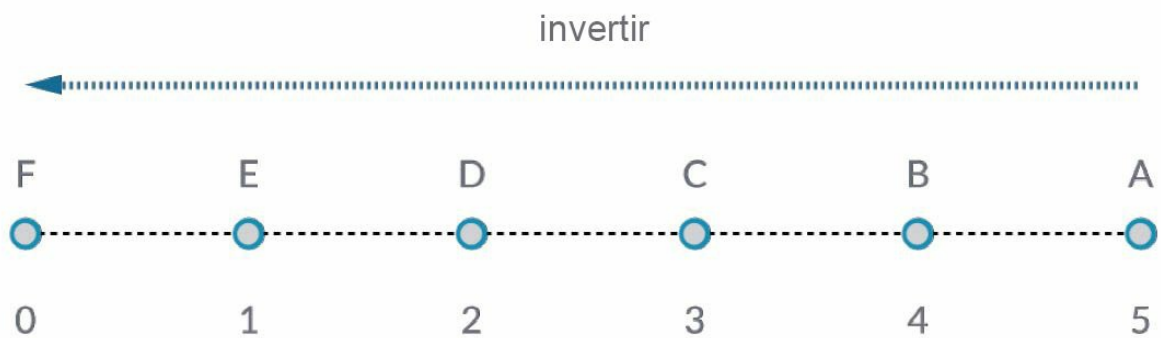
Ejercicio: List.GetItemAtIndex

Descargue el archivo de ejemplo que acompaña a este ejercicio (haga clic con el botón derecho y seleccione "Guardar enlace como..."): [List-GetItemAtIndex.dyn](#). En el Apéndice se incluye una lista completa de los archivos de ejemplo.



1. Con el nodo *List.GetItemAtIndex*, seleccionamos el índice "0" o el primer elemento de la lista de líneas.
2. El nodo *Watch3D* indica que se ha seleccionado una línea. Nota: para obtener la imagen anterior, asegúrese de desactivar la vista preliminar de *Line.ByStartPointEndPoint*.

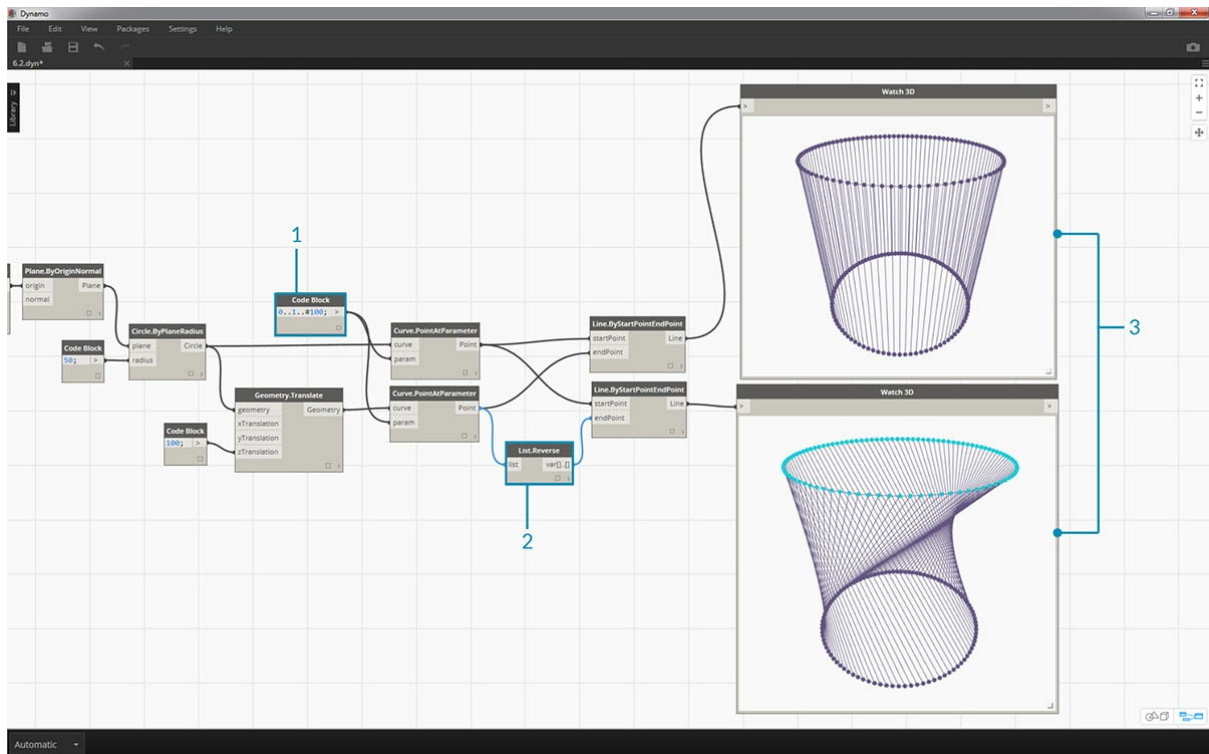
List.Reverse



List.Reverse invierte el orden de todos los elementos de una lista.

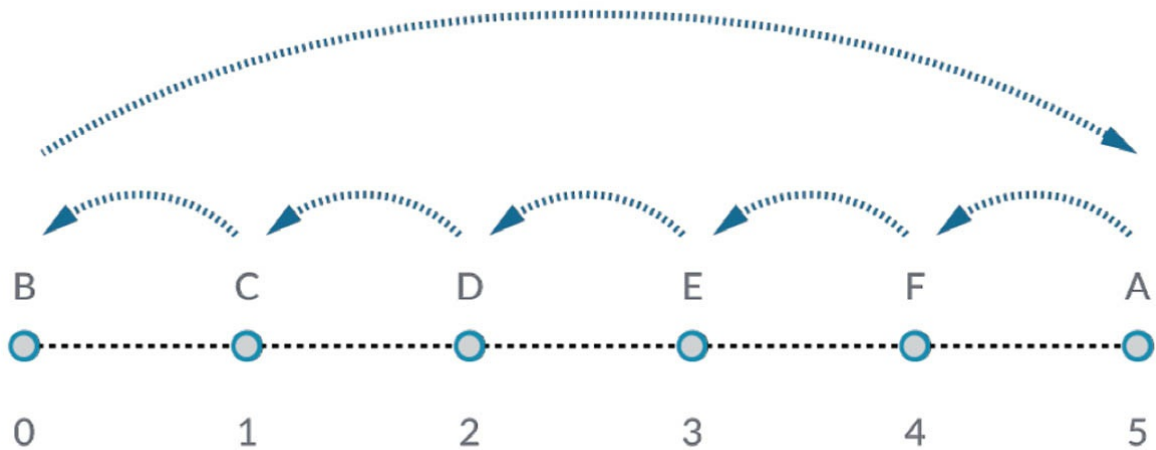
Ejercicio: List.Reverse

Descargue el archivo de ejemplo que acompaña a este ejercicio (haga clic con el botón derecho y seleccione "Guardar enlace como..."): [List-Reverse.dyn](#). En el Apéndice se incluye una lista completa de los archivos de ejemplo.



1. Para visualizar correctamente la lista de líneas invertida, cree más líneas cambiando el bloque de código a `0..1..#100`;
2. Inserte un nodo `List.Reverse` entre los nodos `Curve.PointAtParameter` y `Line.ByStartPointEndPoint` de una de las listas de puntos.
3. Los nodos `Watch3D` muestran dos resultados diferentes. El primero muestra el resultado sin una lista invertida. Las líneas se conectan verticalmente con los puntos adyacentes. Sin embargo, la lista invertida conectará todos los puntos con la otra lista en el orden opuesto.

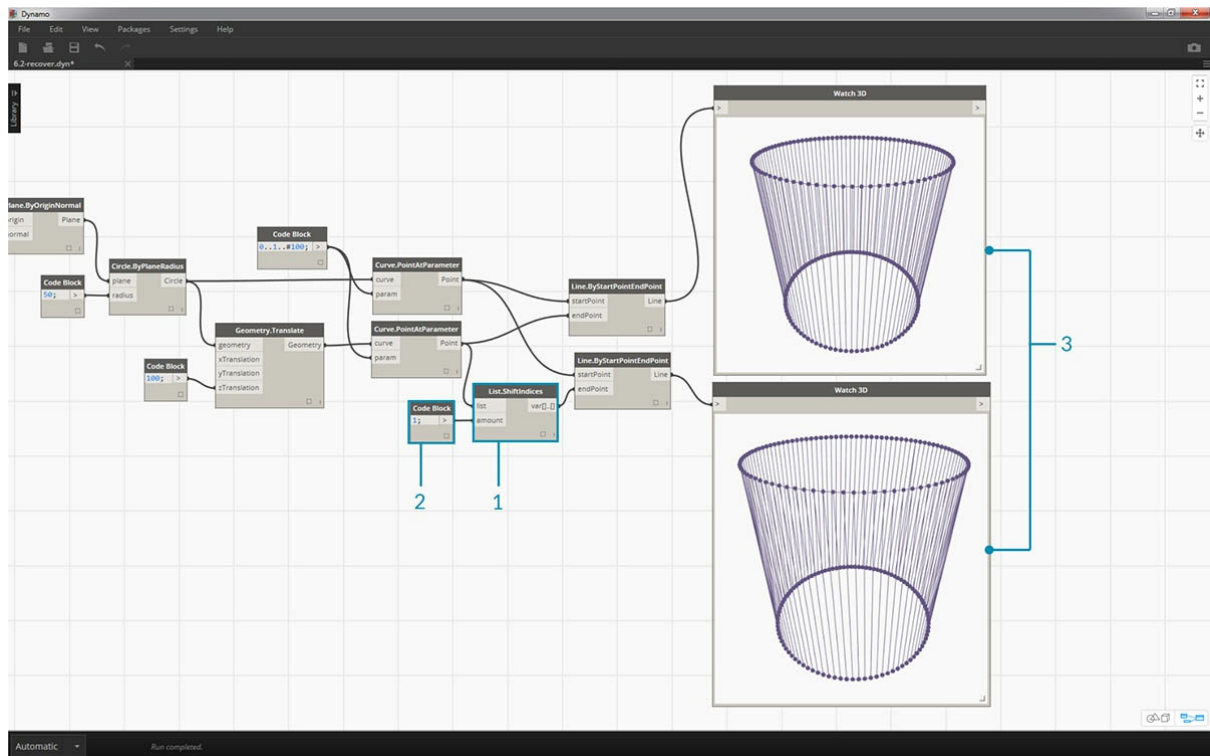
List.ShiftIndices



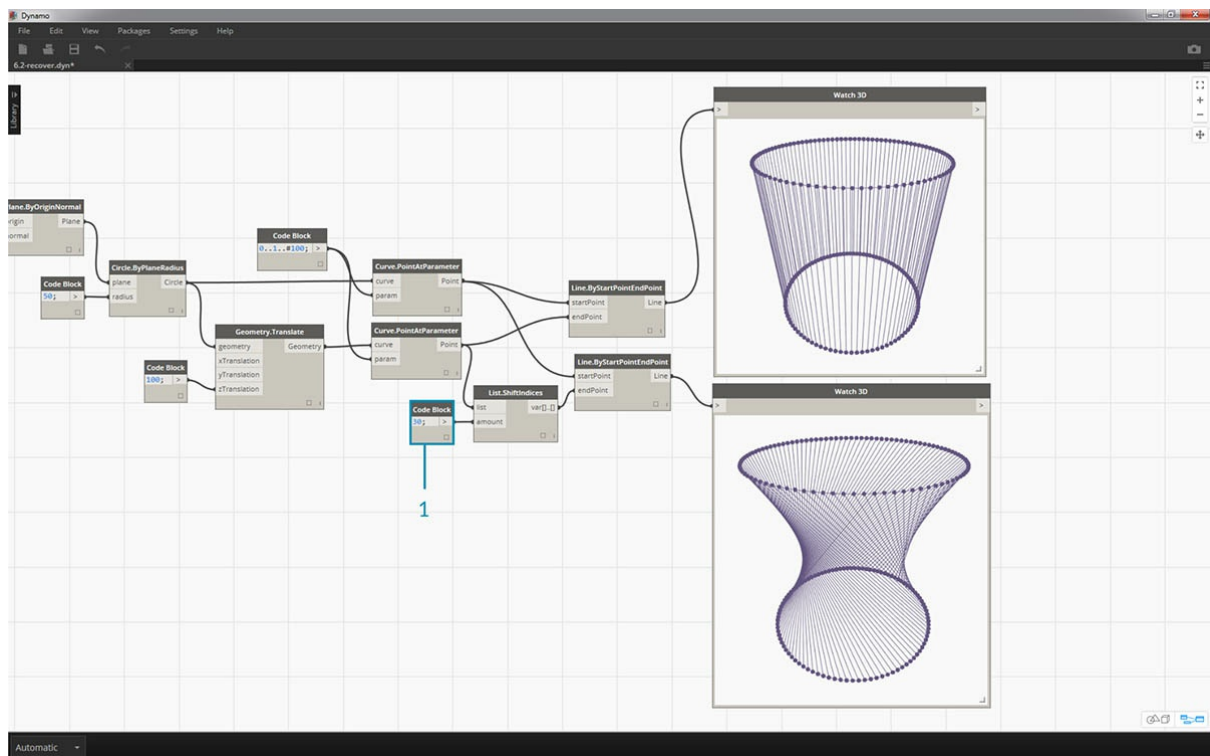
`List.ShiftIndices` es una buena herramienta para crear torsiones o patrones helicoidales, así como cualquier otra manipulación de datos similar. Este nodo desplaza los elementos de una lista un determinado número de índices.

Ejercicio: List.ShiftIndices

Descargue el archivo de ejemplo que acompaña a este ejercicio (haga clic con el botón derecho y seleccione "Guardar enlace como..."): [List-ShiftIndices.dyn](#). En el Apéndice se incluye una lista completa de los archivos de ejemplo.

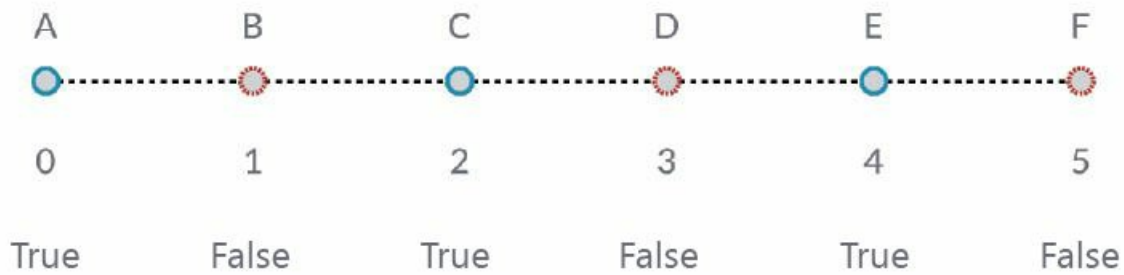


1. En el mismo proceso que la lista inversa, inserte un nodo **List.ShiftIndices** en **Curve.PointAtParameter** y **Line.ByStartPointEndPoint**.
2. Mediante un **bloque de código**, designe un valor de "1" para desplazar la lista un índice.
3. Observe que el cambio es sutil, pero todas las líneas del nodo **Watch3D** inferior se han desplazado un índice al conectarlas al otro conjunto de puntos.



1. Al cambiar el **bloque de código** a un valor más grande, por ejemplo "30", observamos una diferencia significativa en las líneas diagonales. El desplazamiento funciona como el iris de una cámara en este caso, creando una torsión en la forma cilíndrica original.

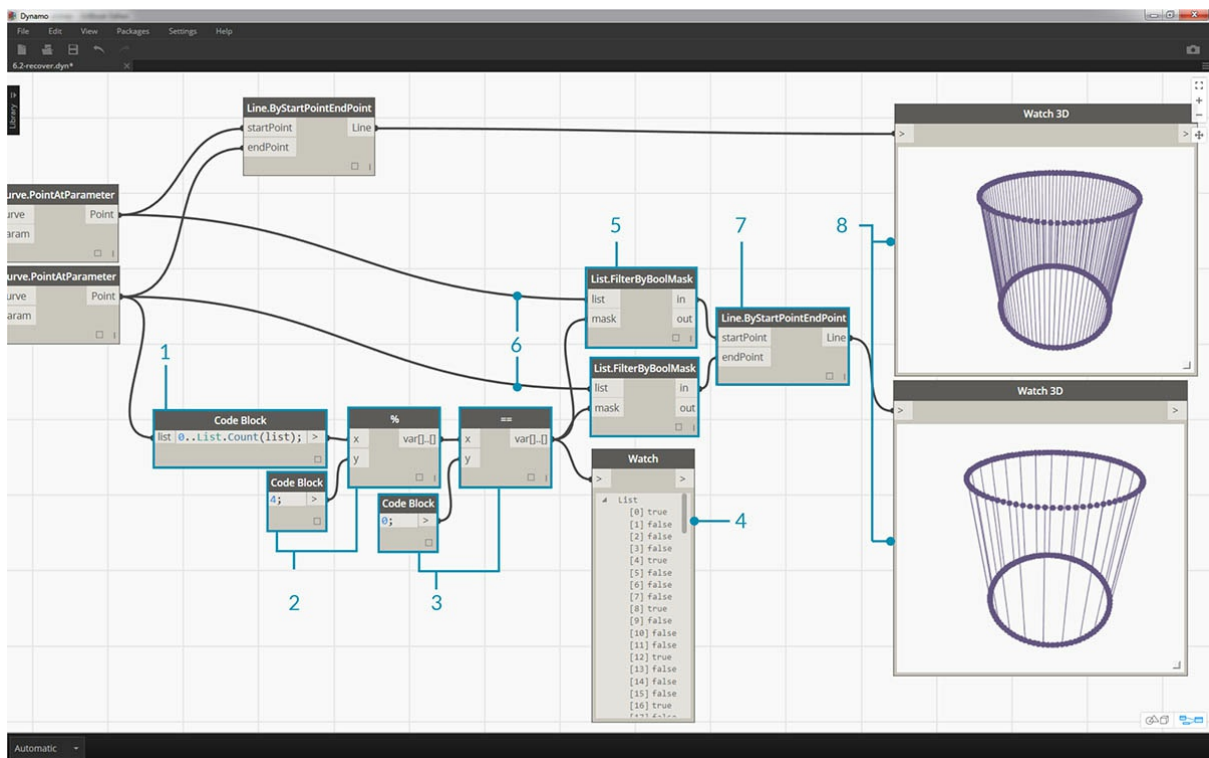
List.FilterByBooleanMask



List.FilterByBooleanMask eliminará determinados elementos en función de una lista de operaciones booleanas o valores de "verdadero" o "falso".

Ejercicio: *List.FilterByBooleanMask*

Descargue el archivo de ejemplo que acompaña a este ejercicio (haga clic con el botón derecho y seleccione "Guardar enlace como..."): [List-FilterByBooleanMask.dyn](#). En el Apéndice se incluye una lista completa de los archivos de ejemplo.



Para crear una lista de valores de "verdadero" o "falso", necesitamos realizar un poco más de trabajo...

1. Mediante un *bloque de código*, defina una expresión con la sintaxis: `0..List.Count(list)-1`. Conecte el nodo *Curve.PointAtParameter* a la entrada *list*. Veremos más detalles sobre esta configuración en el capítulo del bloque de código, pero la línea de código en este caso nos proporciona una lista que representa cada índice del nodo *Curve.PointAtParameter*.
2. Mediante un nodo "%" (módulo), conecte la salida del *bloque de código* a la entrada *x* y un valor de 4 a la entrada *y*. Esto nos dará el resto al dividir la lista de índices entre 4. El módulo es un nodo muy útil para la creación de patrones. Todos los valores serán los restos posibles de 4: 0, 1, 2, 3.
3. En el nodo *módulo*, sabemos que un valor de 0 significa que el índice es divisible por 4 (0, 4, 8, etc.). Mediante un nodo "=", podemos probar la divisibilidad frente a un valor de "0".
4. El nodo *Watch* muestra solo esto: disponemos de un patrón verdadero/falso que dice: *verdadero,falso,falso,falso,falso...*
5. Mediante este patrón verdadero/falso, conéctelo a la entrada *mask* de dos nodos *List.FilterByBooleanMask*.
6. Conecte el nodo *Curve.PointAtParameter* en cada una de las entradas *list* de los nodos *List.FilterByBooleanMask*.
7. Las salidas de *Filter.ByBooleanMask* son "in" y "out". "in" representa los valores que tenían un valor de máscara de "Verdadero (True)" mientras que "out" representa los valores que tenían un valor de "Falso (False)". Al conectar las salidas "in" a las entradas *startPoint* y *endPoint* de un nodo *Line.ByStartPointEndPoint*, hemos creado una lista filtrada de líneas.
8. El nodo *Watch3D* indica que hay menos líneas que puntos. Hemos seleccionado solo el 25% de los nodos filtrando solo los valores verdaderos.

Listas de listas

Listas de listas

Añadamos un nivel más a la jerarquía. Si utilizamos la baraja de cartas del ejemplo original y creamos un cuadro que contiene varias barajas, este representa ahora una lista de barajas y cada baraja representa una lista de cartas. Esta es una lista de listas. En esta sección, usaremos esta analogía: el cuadro rojo mostrado a continuación contiene un rollo de monedas y cada rollo contiene una lista de peniques.



Foto de [Dori](#).

¿Qué **consultas** se pueden realizar a partir de la lista de listas? Esto permite acceder a las propiedades existentes.

- ¿Número de tipos de moneda? 2.
- ¿Valores de tipo moneda? 0,01 y 0,25 USD.
- ¿Material de los cuartos de dólar? 75 % de cobre y 25 % de níquel.
- ¿Material de los peniques? 97,5 % de zinc y 2,5% de cobre.

¿Qué **acciones** se pueden realizar en la lista de listas? Esta acción cambia la lista de listas en función de la operación especificada.

- Seleccione un montón específico de cuartos de dólar o peniques.
- Seleccione un cuarto de dólar o un penique específicos.
- Reorganice los montones de cuartos de dólar y peniques.
- Junte los montones.

De nuevo, Dynamo tiene un nodo análogo para cada una de las operaciones anteriores. Como trabajamos con datos abstractos y no con objetos físicos, necesitamos un conjunto de reglas para controlar el desplazamiento hacia arriba y hacia abajo por la jerarquía de datos.

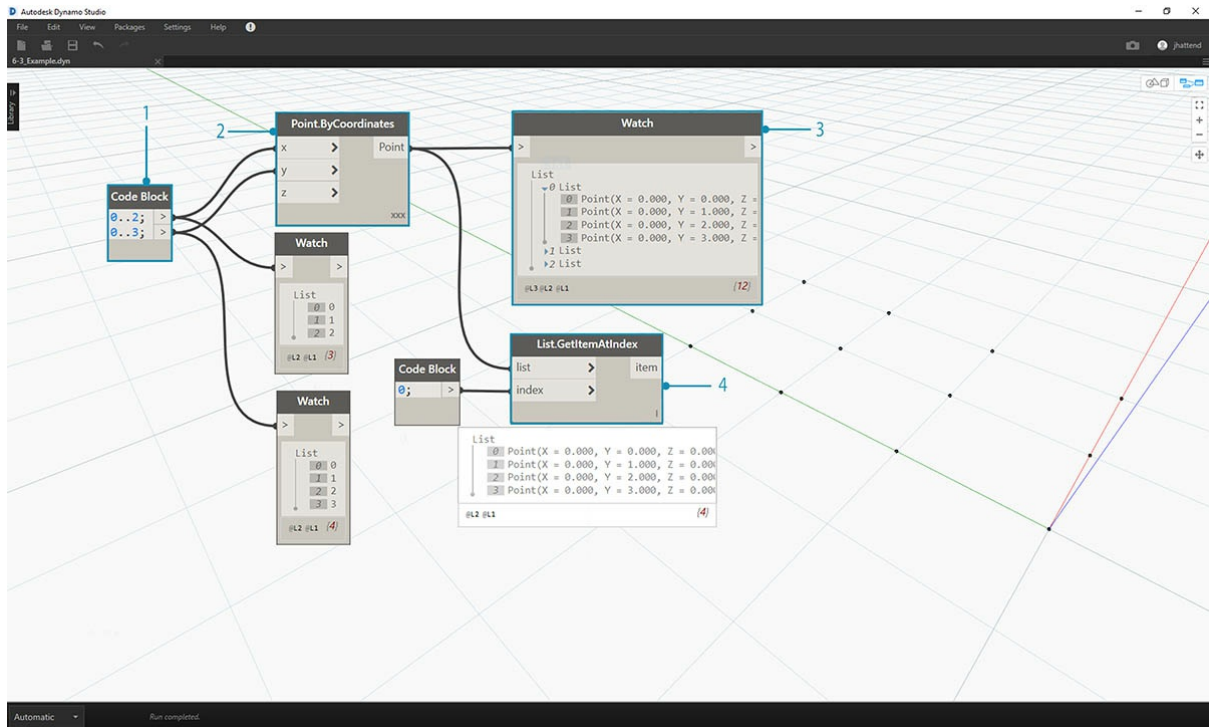
Al trabajar con listas de listas, los datos son complejos y se dividen en capas, pero esto ofrece la oportunidad de realizar operaciones paramétricas impresionantes. Vamos a desglosar los conceptos básicos y abordar algunas operaciones más en las lecciones mostradas a continuación.

Jerarquía descendente

El concepto fundamental que se debe aprender en esta sección: **Dynamo trata las listas como objetos en sí mismos**. Esta jerarquía descendente se desarrolla teniendo en cuenta la programación orientada a objetos. En lugar de seleccionar subelementos con un comando como `List.GetItemAtIndex`, Dynamo seleccionará ese índice de la lista principal en la estructura de datos. Y ese elemento puede ser otra lista. Vamos a desglosar esto con una imagen de ejemplo:

Ejercicio: jerarquía descendente

Descargue el archivo de ejemplo que acompaña a este ejercicio (haga clic con el botón derecho y seleccione "Guardar vínculo como..."): [Top-Down-Hierarchy.dyn](#). En el Apéndice, se incluye una lista completa de los archivos de ejemplo.



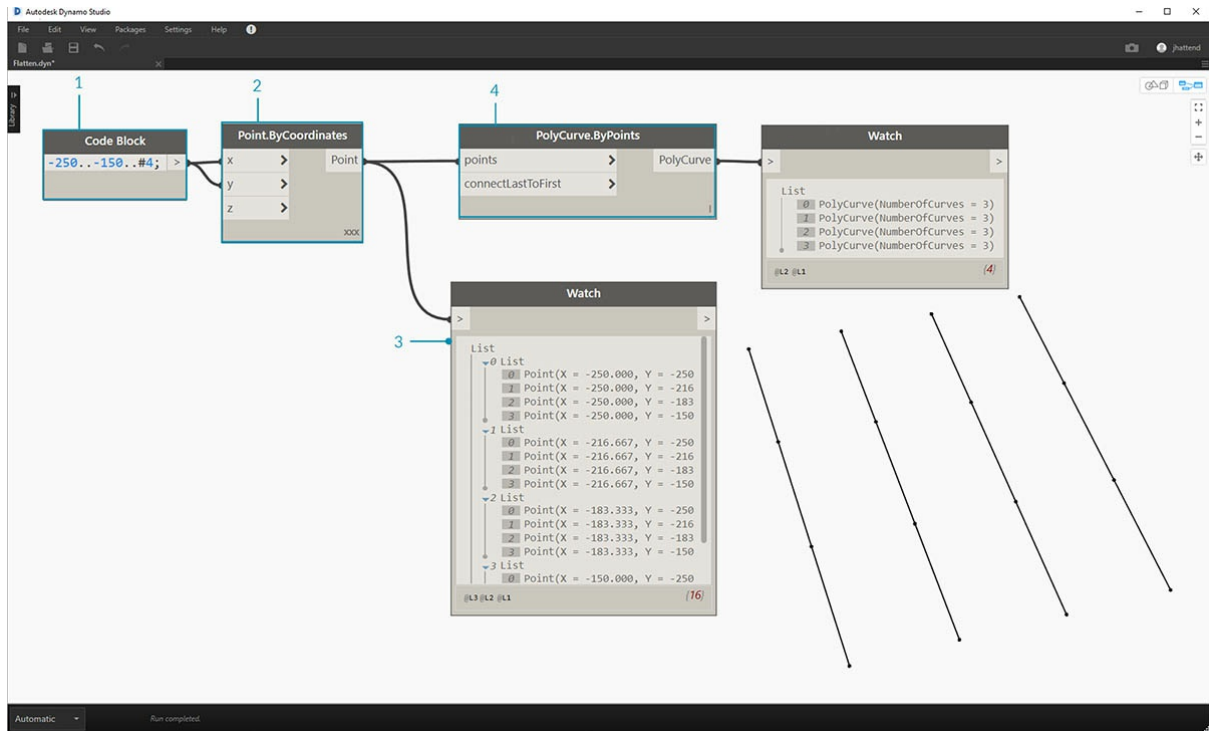
1. Con el *bloque de código*, hemos definido dos intervalos: `0..2;` `0..3;`
2. Estos intervalos están conectados a un nodo *Point.ByCoordinates* con el encaje establecido en "Producto vectorial". De este modo, se crea una rejilla de puntos y también se devuelve una lista de listas como salida.
3. Observe que el nodo *Watch* proporciona tres listas con cuatro elementos en cada lista.
4. Cuando se utiliza *List.GetItemAtIndex* con un índice 0, Dynamo selecciona la primera lista y todo su contenido. Es posible que otros programas seleccionen el primer elemento de cada lista de la estructura de datos, pero Dynamo utiliza una jerarquía descendente al trabajar con los datos.

Flatten y List.Flatten

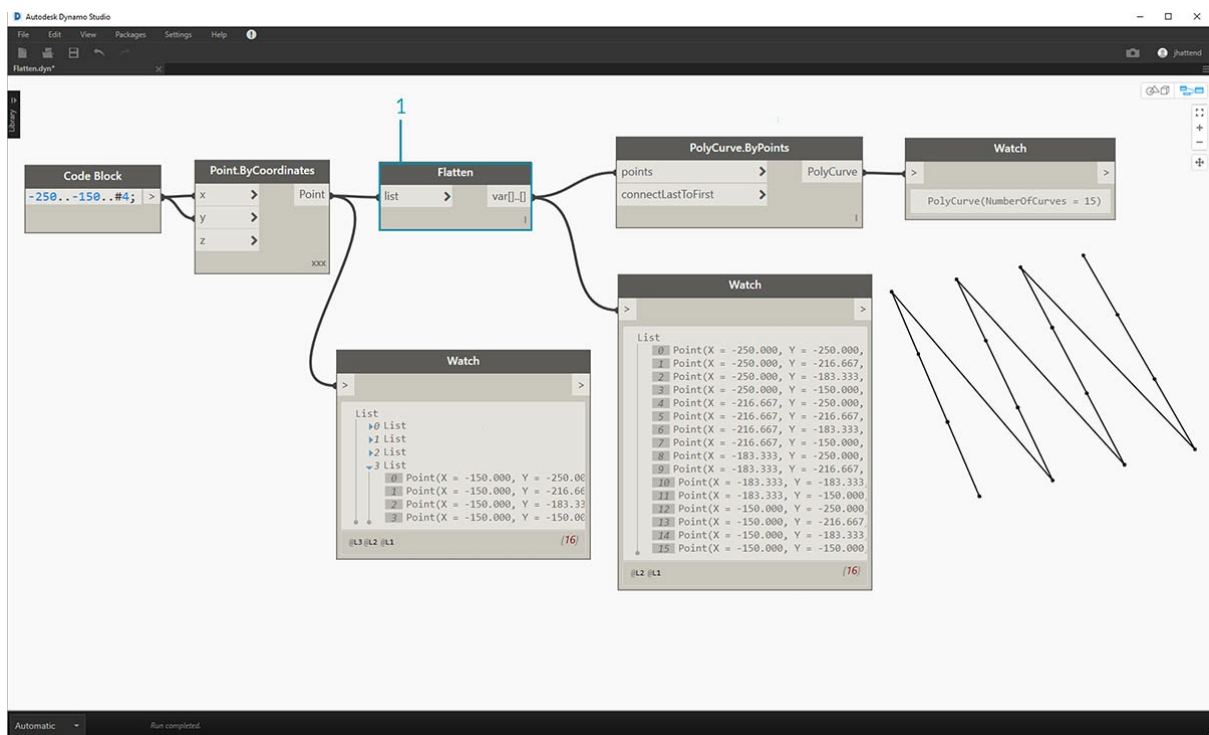
Flatten elimina todos los niveles de datos de una estructura de datos. Esto resulta útil cuando las jerarquías de datos no son necesarias para la operación, pero puede resultar peligroso porque elimina información. En el siguiente ejemplo, se muestra el resultado de aplanar una lista de datos.

Ejercicio: Flatten

Descargue el archivo de ejemplo que acompaña a este ejercicio (haga clic con el botón derecho y seleccione "Guardar vínculo como..."): [Flatten.dyn](#). En el Apéndice, se incluye una lista completa de los archivos de ejemplo.



1. Inserte una línea de código para definir un intervalo en el bloque de código: `-250..-150..#4;`
2. Al conectar el bloque de código a la entrada x e y de un nodo Point.ByCoordinates, se establece el encaje como "Producto vectorial" para obtener una rejilla de puntos.
3. En el nodo Watch, se muestra que tenemos una lista de listas.
4. Un nodo PolyCurve.ByPoints hará referencia a cada lista y creará la PolyCurve respectiva. En la vista preliminar de Dynamo, observe que tenemos cuatro PolyCurves que representan cada fila de la rejilla.



1. Al insertar un elemento Flatten antes del nodo de PolyCurve, hemos creado una única lista para todos los puntos. El nodo de PolyCurve hace referencia a una lista para crear una curva y, como todos los puntos están en una lista, se obtiene una PolyCurve en zigzag que se ejecuta en toda la lista de puntos.

También existen opciones para aplanar niveles aislados de datos. Mediante el nodo List.Flatten, puede definir un número establecido de niveles de datos que aplanar desde la parte superior de la jerarquía. Esta es una herramienta muy útil si tiene problemas con estructuras de datos complejas que no son necesariamente pertinentes para el flujo de trabajo. Otra opción consiste en utilizar el nodo de aplanado como una función en List.Map. A continuación, analizaremos [List.Map](#).

Chop

En el modelado paramétrico, también hay ocasiones en las que se desean añadir más estructuras de datos a una lista existente. Hay muchos nodos disponibles para ello y el elemento "Chop" es la versión más básica. Con "Chop", se puede dividir una lista en sublistas con un número establecido de elementos.

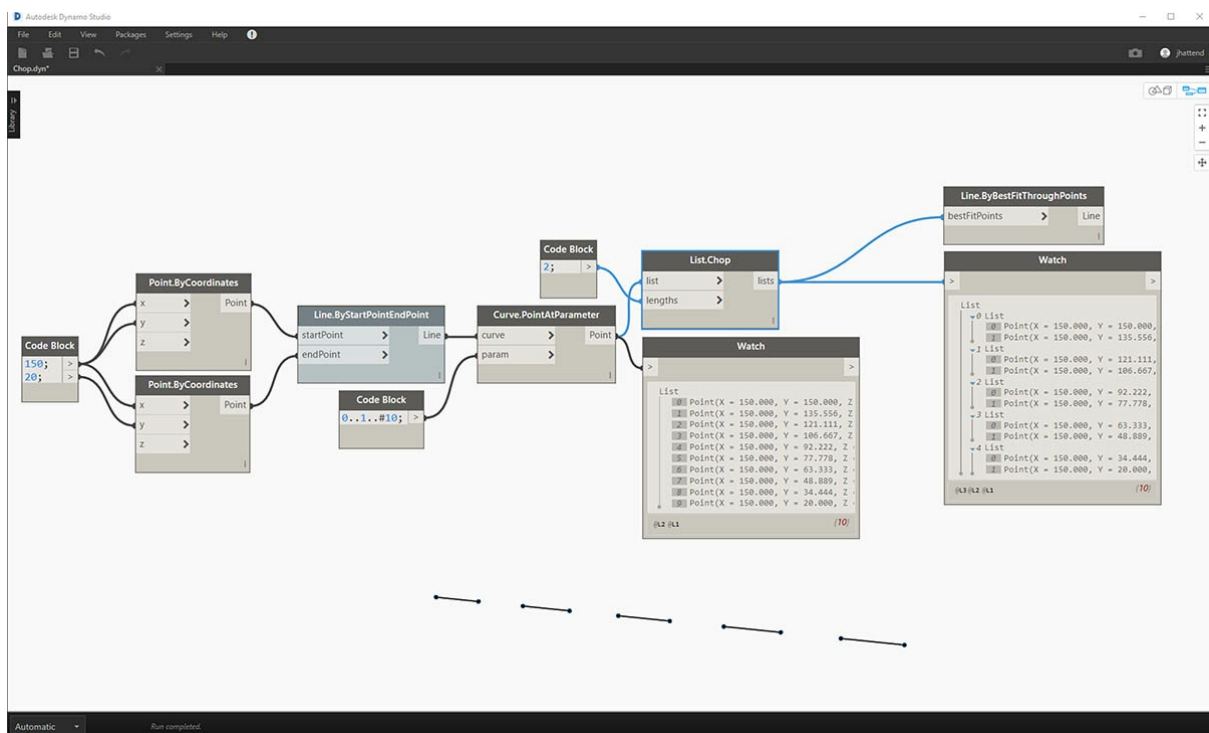
Ejercicio: List.Chop

Descargue el archivo de ejemplo que acompaña a este ejercicio (haga clic con el botón derecho y seleccione "Guardar vínculo como..."): [Chop.dyn](#). En el Apéndice, se incluye una lista completa de los archivos de ejemplo.



El valor 2 de `List.Chop` con `_with a _subLength` crea cuatro listas con dos elementos cada una.

El comando "Chop" divide listas en función de una longitud de lista determinada. De alguna manera, el efecto de "Chop" es contrario al efecto de "Flatten": en lugar de eliminar una estructura de datos, se añaden nuevos niveles a ella. Es una herramienta útil para operaciones geométricas, como el ejemplo siguiente.



List.Map y List.Combine

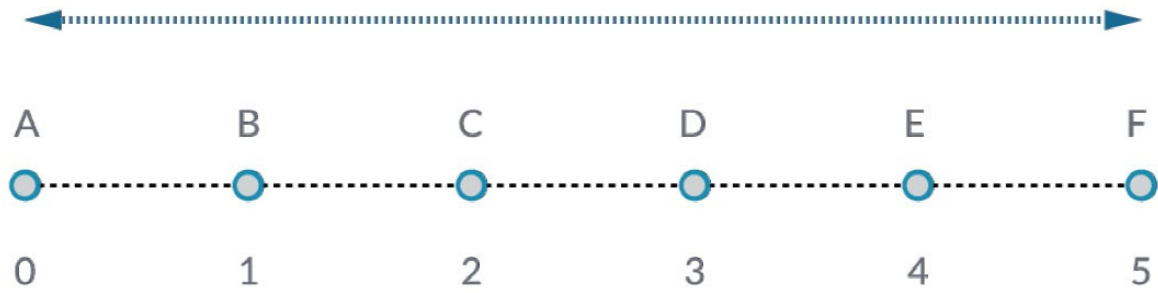
List.Map y List.Combine aplican una función establecida a una lista de entradas, pero un nivel hacia abajo en la jerarquía. Las combinaciones son iguales a las asignaciones, excepto que las combinaciones pueden tener varias entradas correspondientes a la entrada de una determinada función.

Ejercicio: List.Map

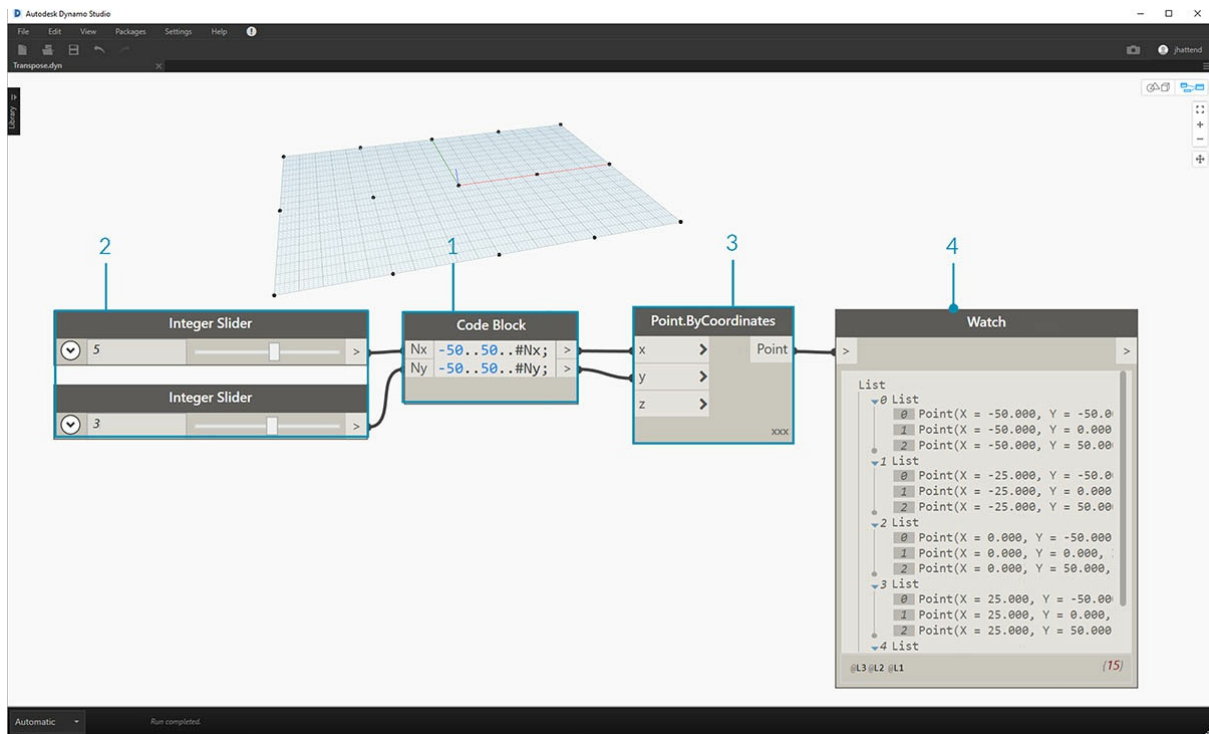
Nota: este ejercicio se ha creado con una versión anterior de Dynamo. Gran parte de la funcionalidad de List.Map se ha resuelto con la adición de la función List@Level. Para obtener más información, consulte [List@Level](#) a continuación.

Descargue el archivo de ejemplo que acompaña a este ejercicio (haga clic con el botón derecho y seleccione "Guardar vínculo como..."): [Map.dyn](#). En el Apéndice, se incluye una lista completa de los archivos de ejemplo.

Como introducción rápida, revisemos el nodo List.Count de una sección anterior.



El nodo *List.Count* realiza un recuento de todos los elementos de una lista. Utilizaremos esta opción para mostrar cómo funciona *List.Map*.

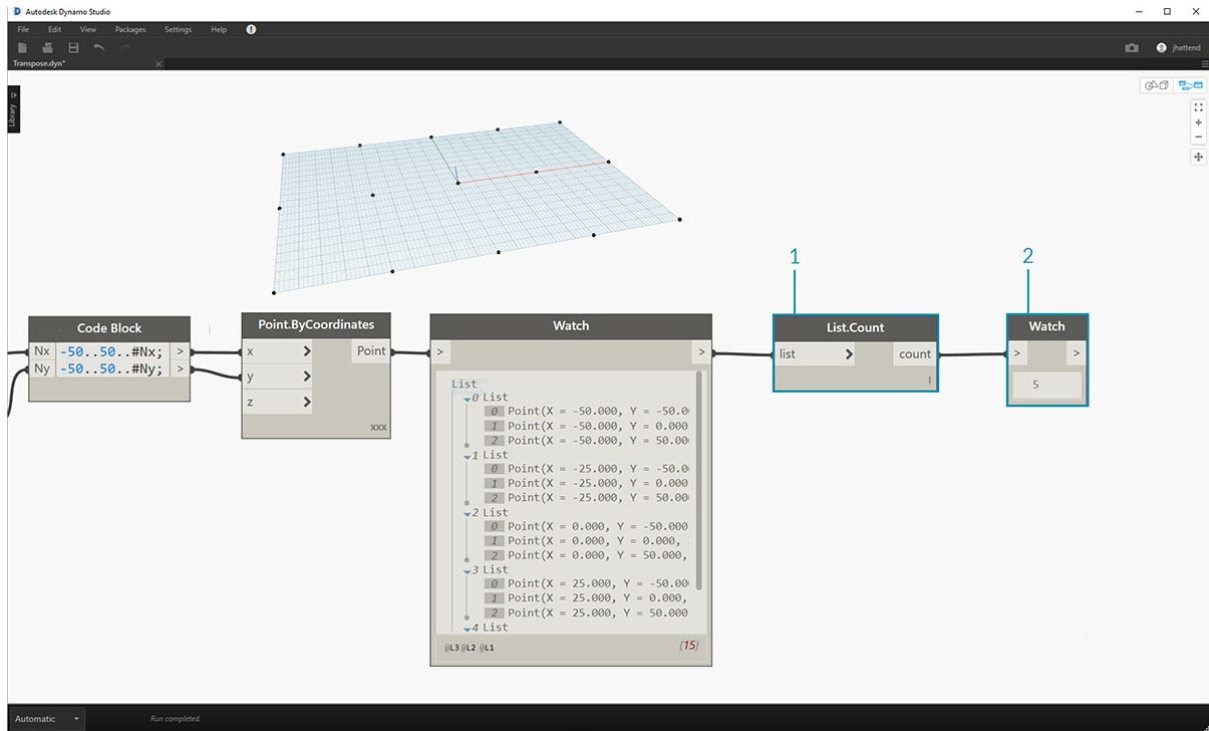


1. Inserte dos líneas de código en el *bloque de código*:

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

Después de escribir este código, el bloque de código creará dos entradas para Nx y Ny.

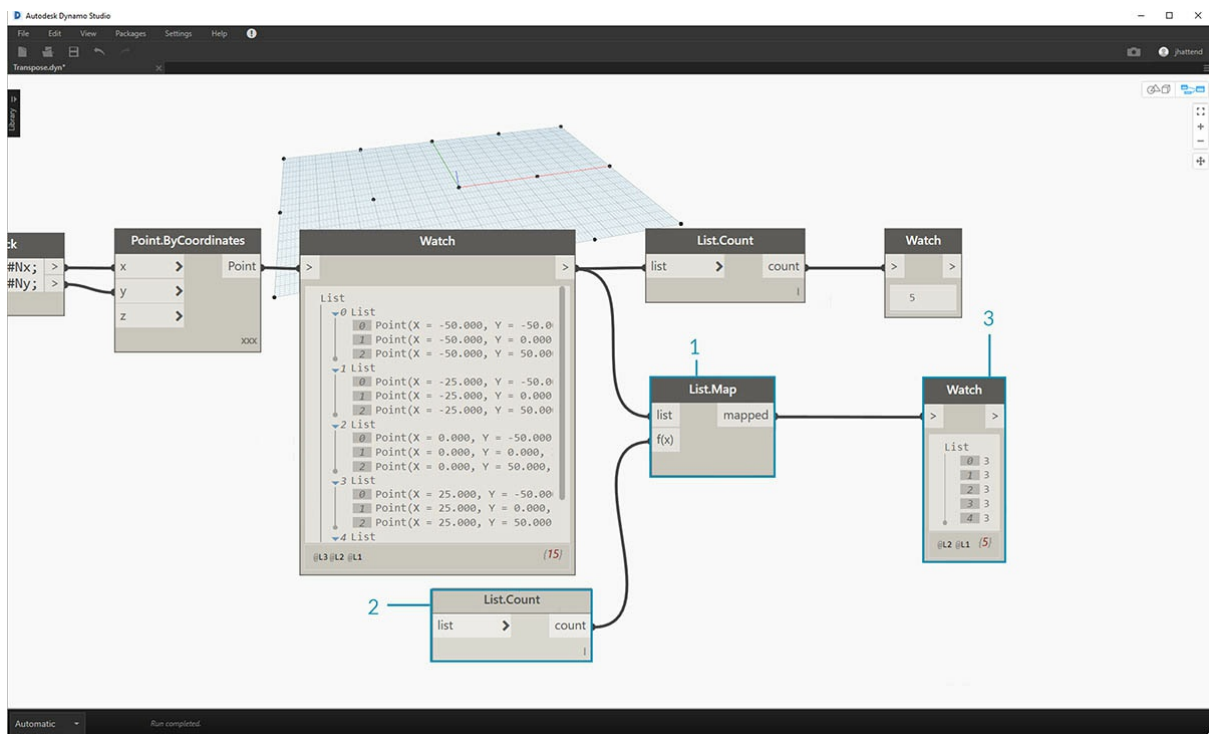
1. Con dos *controles deslizantes de enteros*, defina los valores de Nx y Ny conectándolos al *bloque de código*.
2. Conecte cada línea del bloque de código a las entradas X e Y respectivas de un nodo *Point.ByCoordinates*. Haga clic con el botón derecho en el nodo, seleccione "Encaje" y, a continuación, elija "Producto vectorial". Esta acción crea una rejilla de puntos. Como hemos definido el intervalo de -50 a 50, se extiende la rejilla de Dynamo por defecto.
3. En el nodo *Watch*, se muestran los puntos creados. Observe la estructura de datos. Hemos creado una lista de listas. Cada lista representa una fila de puntos de la rejilla.



1. Enlance un nodo *List.Count* a la salida del nodo de visualización del paso anterior.
2. Conecte un nodo *Watch* a la salida de *List.Count*.

Observe que el nodo *List.Count* ofrece el valor 5. Este es igual a la variable "Nx", tal y como se ha definido en el bloque de código. ¿Por qué ocurre eso?

- En primer lugar, el nodo *Point.ByCoordinates* utiliza la entrada "x" como entrada principal para crear listas. Si Nx es 5 y Ny es 3, se obtiene una lista de cinco listas, cada una con tres elementos.
- Como Dynamo trata las listas como objetos en sí mismos, se aplica un nodo *List.Count* a la lista principal de la jerarquía. El resultado es el valor 5 o el número de listas de la lista principal.



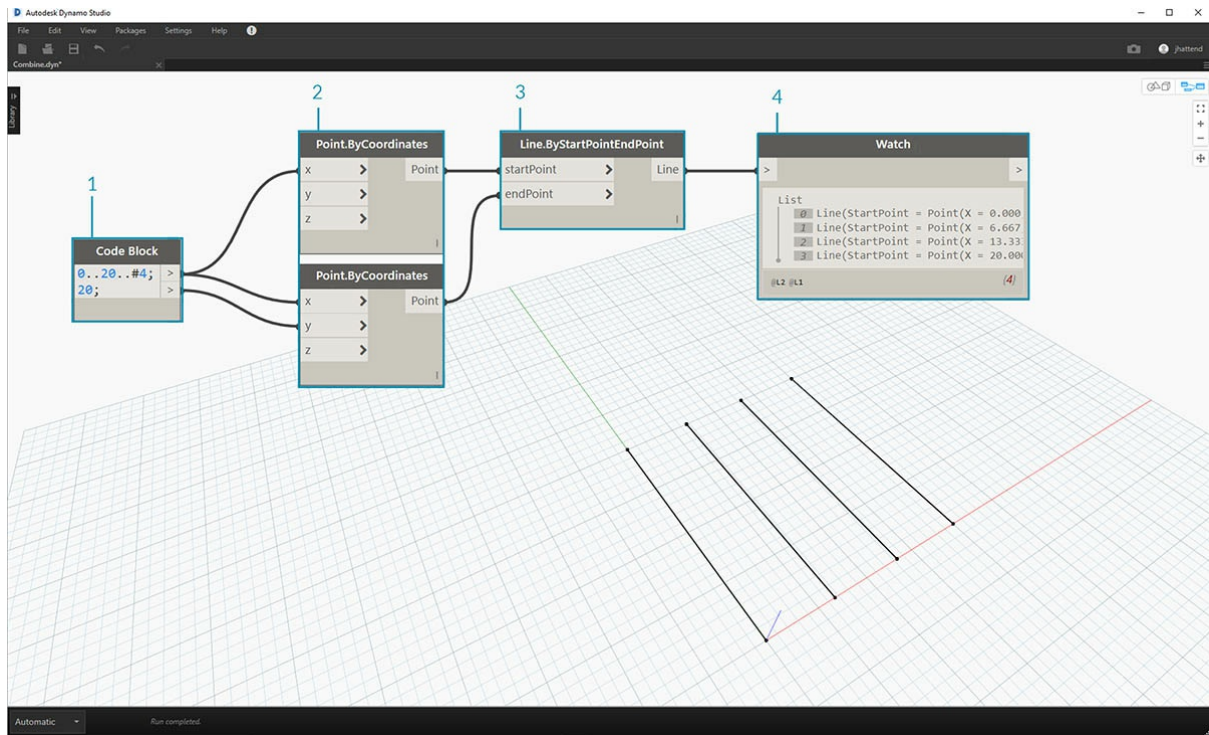
1. Mediante un nodo *List.Map*, nos desplazaremos un nivel hacia abajo en la jerarquía y ejecutaremos una "función" en este nivel.
2. Observe que el nodo *List.Count* no tiene ninguna entrada. Se utiliza como una función, por lo que el nodo *List.Count* se aplicará a cada lista individual un nivel hacia abajo en la jerarquía. La entrada en blanco de *List.Count* corresponde a la entrada de lista de *List.Map*.
3. Los resultados de *List.Count* ahora proporcionan una lista de cinco elementos, cada uno con el valor 3. Esto representa la longitud de cada sublista.

Ejercicio: List.Combine

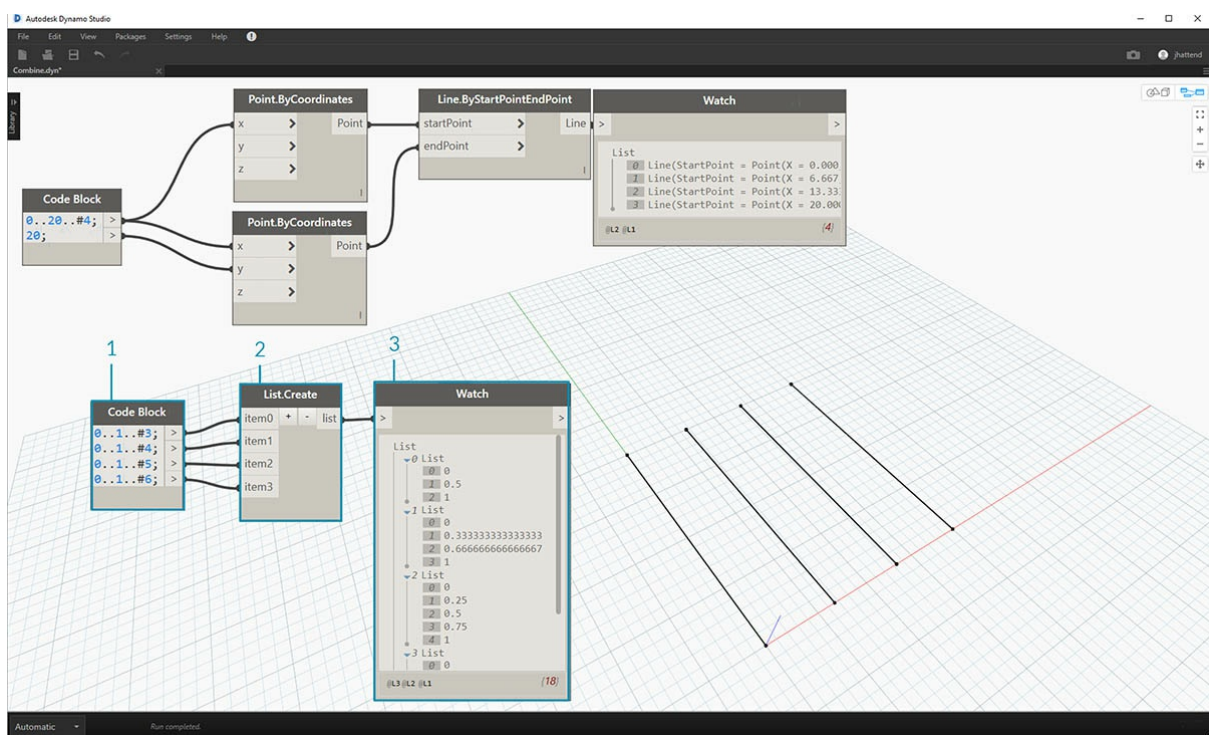
Nota: este ejercicio se ha creado con una versión anterior de Dynamo. Gran parte de la funcionalidad de List.Combine se ha resuelto con la adición de la función List@Level. Para obtener más información, consulte [List@Level](#listlevel) a continuación.

Descargue el archivo de ejemplo que acompaña a este ejercicio (haga clic con el botón derecho y seleccione "Guardar vínculo como..."): [Combine.dyn](#). En el Apéndice, se incluye una lista completa de los archivos de ejemplo.

En este ejercicio, utilizaremos una lógica similar a List.Map, pero con varios elementos. En este caso, deseamos dividir una lista de curvas por un número único de puntos.



1. Mediante el bloque de código, defina un intervalo con la sintaxis: `..20..#4;` y un valor de `20;` debajo de esa línea.
2. Conecte el bloque de código a dos nodos `Point.ByCoordinates`.
3. Cree `Line.ByStartPointEndPoint` a partir de los nodos `Point.ByCoordinates`.
4. En el nodo `Watch`, se muestran cuatro líneas.

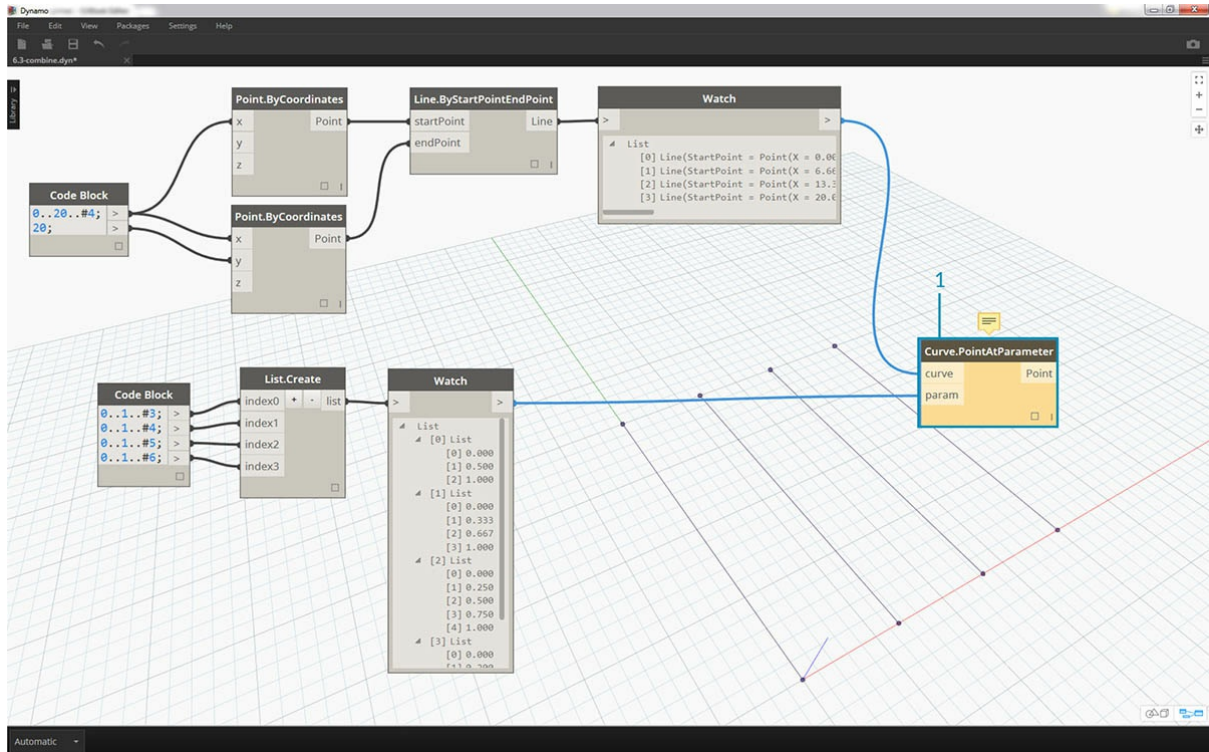


1. Debajo del gráfico de creación de líneas, deseamos utilizar el bloque de código para crear cuatro intervalos distintos a fin de dividir las líneas

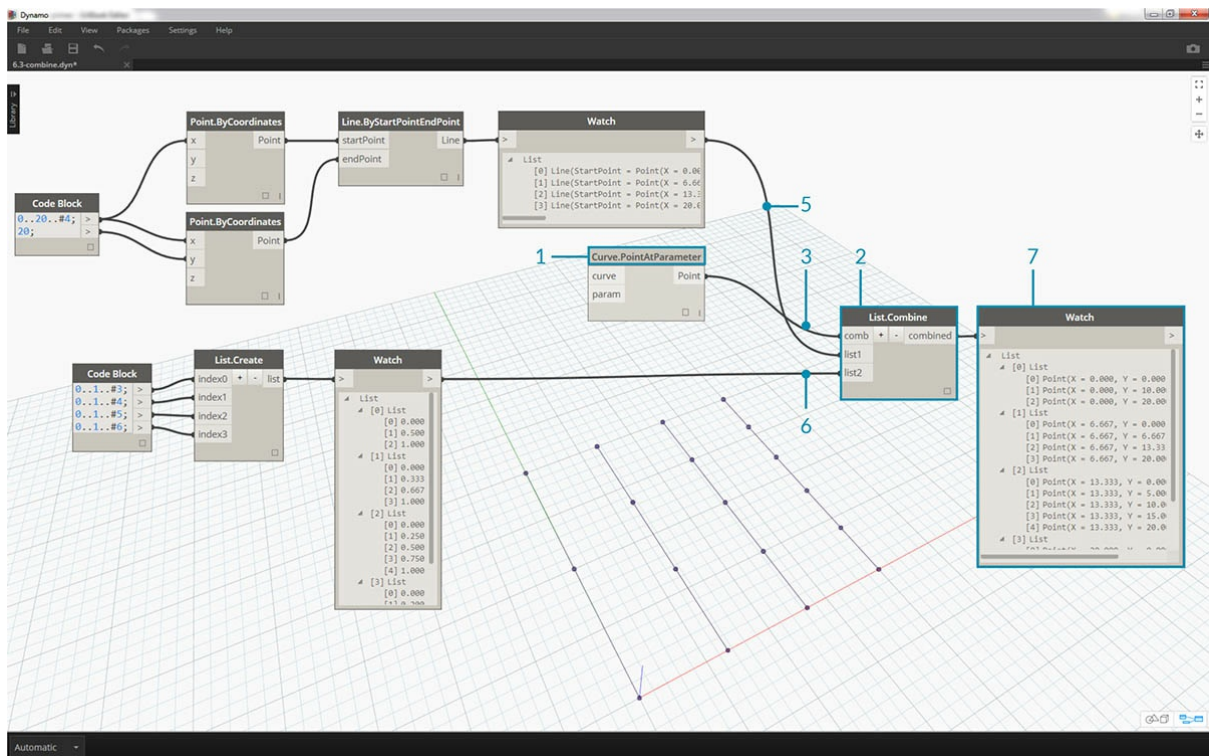
de forma exclusiva. Esto se realiza con las siguientes líneas de código:

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

1. Con un nodo *List.Create*, combinamos las cuatro líneas del *bloque de código* en una lista.
2. En el nodo *Watch*, se muestra una lista de listas.



1. *Curve.PointAtParameter* no funcionará mediante la conexión de las líneas directamente con los valores de *parámetro*. Necesitamos bajar un nivel en la jerarquía. Para ello, utilizaremos *List.Combine*.



Mediante *List.Combine*, podemos dividir correctamente cada línea por los intervalos especificados. Esto se pone un poco complicado, así que lo

desglosaremos en profundidad.

1. Añada primero un nodo *Curve.PointAtParameter* al lienzo. Será la "función" o el "combinador" que aplicamos al nodo *List.Combine*. Abordaremos esto de nuevo en un segundo.
2. Añada un nodo *List.Combine* al lienzo. Pulse "+" o "-" para añadir o quitar entradas. En este caso, usaremos las dos entradas por defecto del nodo.
3. Deseamos conectar el nodo *Curve.PointAtParameter* en la entrada "comb" de *List.Combine*. Y otro nodo más importante: asegúrese de hacer clic con el botón derecho en la entrada "_param" de *_Curve.PointAtParameter* y desactive "Utilizar valor por defecto". Los valores por defecto de las entradas de Dynamo deben eliminarse al ejecutar un nodo como una función. En otras palabras, debemos considerar que los valores por defecto tienen nodos adicionales conectados a ellos. Por este motivo, en este caso, es necesario eliminar los valores por defecto.
4. Sabemos que tenemos dos entradas, las líneas y los parámetros para crear puntos. Pero, ¿cómo los conectamos a las entradas de *List.Combine* y en qué orden?
5. Las entradas vacías de *Curve.PointAtParameter*, de arriba a abajo, deben rellenarse en el combinador en el mismo orden. Por lo tanto, las líneas se conectan a *list1* de *List.Combine*.
6. Del mismo modo, los valores de los parámetros se conectan a la entrada *list2* de *List.Combine*.
7. El nodo *Watch* y la vista preliminar de Dynamo nos muestran cuatro líneas, cada una dividida en función de los intervalos de los bloques de código.

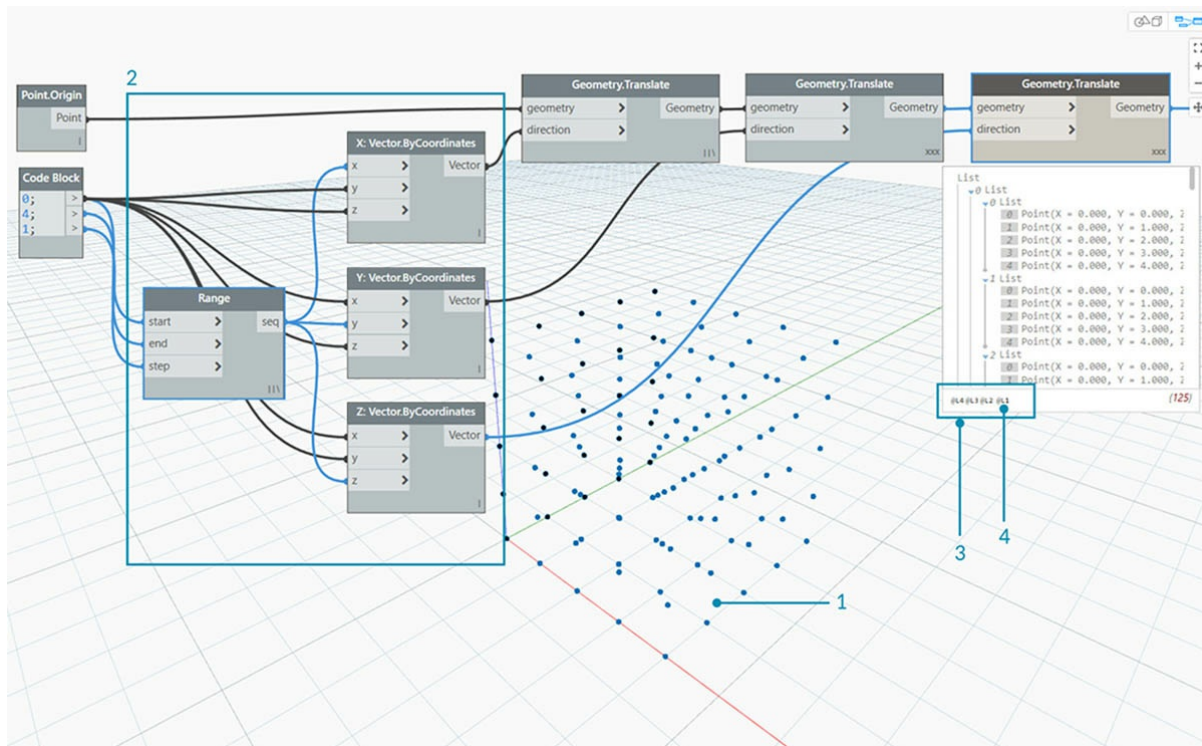
List@Level

Como alternativa preferida a List.Map, la función List@Level permite seleccionar directamente el nivel de lista con el que se desea trabajar directamente en el puerto de entrada del nodo. Esta función se puede aplicar a cualquier entrada de un nodo y le permitirá acceder a los niveles de las listas de forma más rápida y sencilla que con otros métodos. Solo tiene que indicar al nodo el nivel de la lista que desea utilizar como entrada y permitir que el nodo realice el resto.

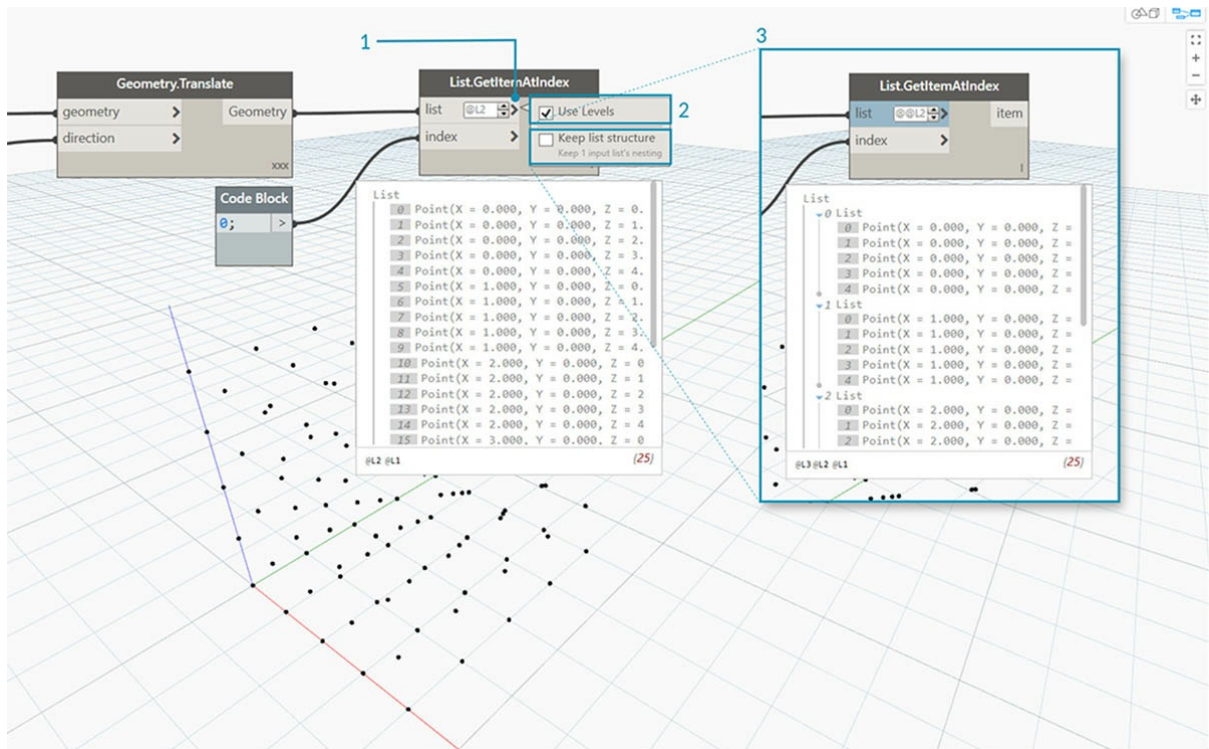
Ejercicio de List@Level

En este ejercicio, utilizaremos la función `List@Level` para aislar un nivel de datos específico.

Descargue el archivo de ejemplo que acompaña a este ejercicio (haga clic con el botón derecho y seleccione "Guardar vínculo como..."): [List@Level](#). En el Apéndice, se incluye una lista completa de los archivos de ejemplo.

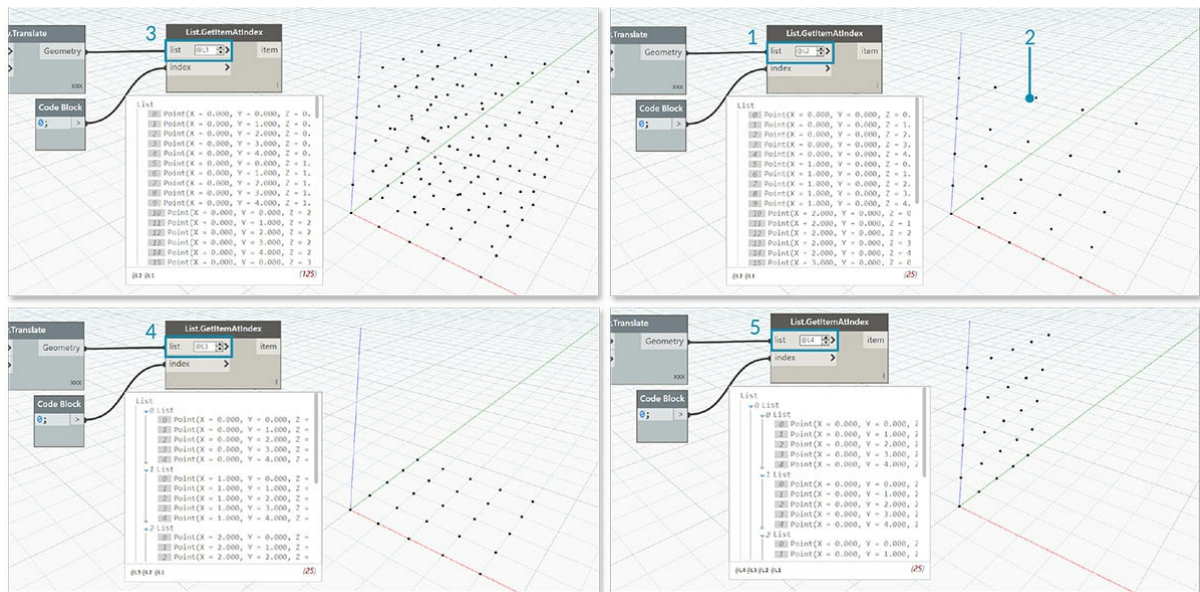


1. Comenzaremos con una rejilla 3D de puntos sencilla.
2. Como la rejilla se ha creado con un intervalo para X, Y y Z, sabemos que los datos están estructurados con tres capas: una lista X, una lista Y y una lista Z.
3. Estas capas existen en **niveles** diferentes. Los niveles se indican en la parte inferior de la burbuja de vista preliminar. Las columnas de niveles de la lista corresponden a los datos de la lista anterior para ayudar a identificar el nivel en el que se va a trabajar.
4. Los niveles de la lista se organizan en orden inverso, por lo que los datos de nivel inferior siempre se encuentran en "L1". Esto ayudará a garantizar que los gráficos funcionen según lo planeado, incluso aunque se realicen cambios en niveles superiores.



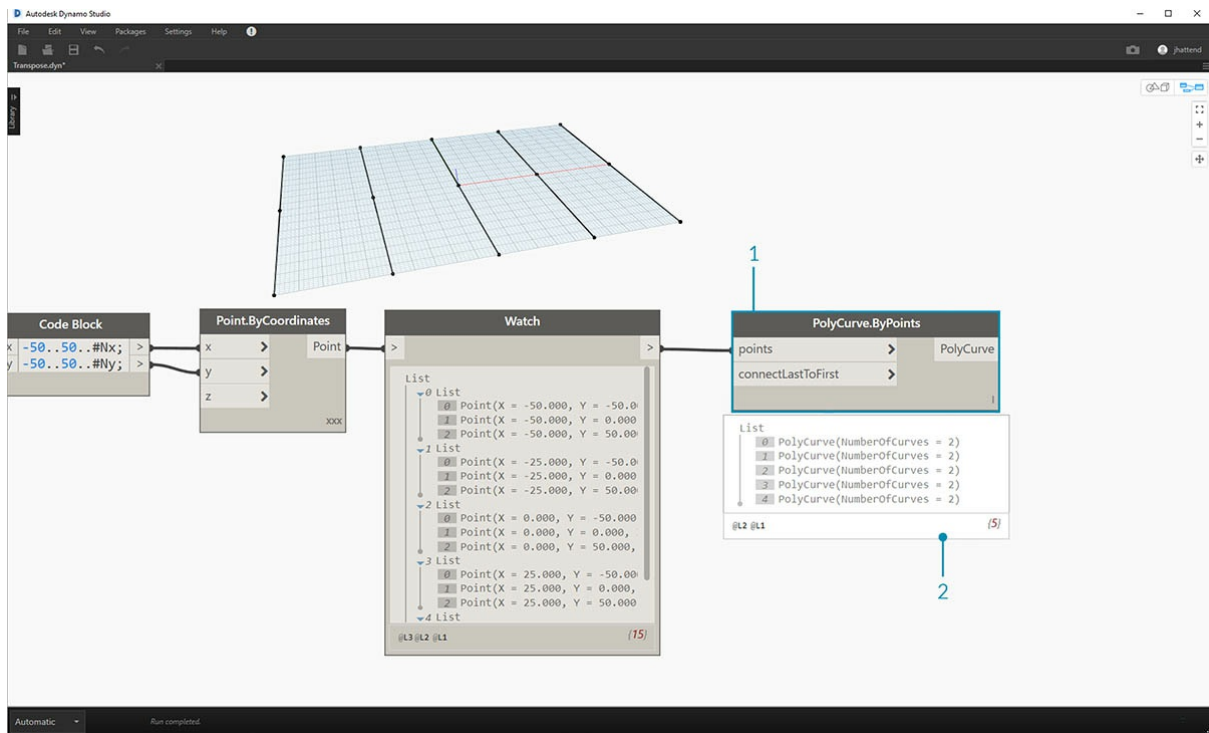
1. Para utilizar la función List@Level, haga clic en ">". En este menú, verá dos casillas de verificación.
2. **Utilizar niveles:** permite utilizar la función List@Level. Después de hacer clic en esta opción, podrá hacer clic por los niveles de lista de entrada que desee que utilice el nodo, además de seleccionarlos. Con este menú, puede probar rápidamente diferentes opciones de nivel haciendo clic arriba o abajo.
3. **Mantener la estructura de listas:** si se ha activado, dispondrá de la opción para mantener la estructura de niveles de esa entrada. En ocasiones, es posible que haya organizado deliberadamente los datos en sublistas. Al seleccionar esta opción, puede mantener la organización de listas intacta sin perder información.

Con nuestra sencilla rejilla 3D, podemos acceder a la estructura de listas y visualizarla mediante la activación o la desactivación de los niveles de la lista. Cada combinación de nivel de lista e índice devolverá un conjunto de puntos diferente de nuestro conjunto 3D original.



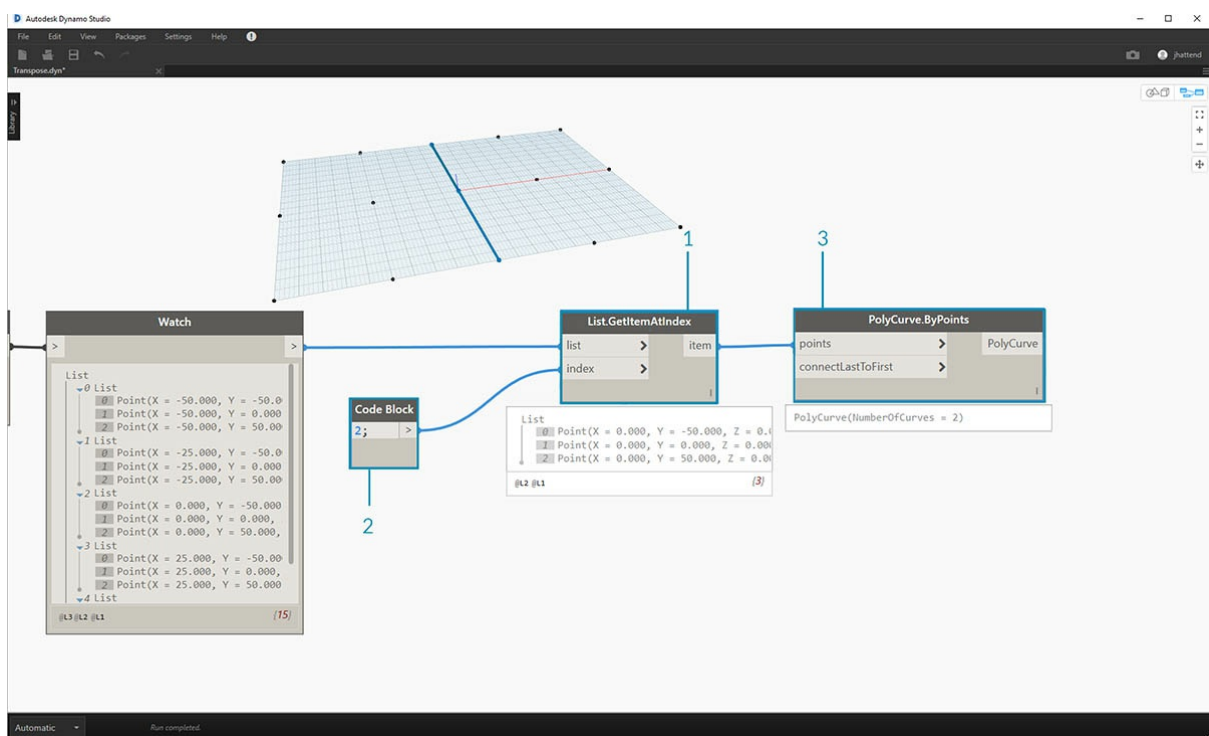
1. "@L2" en DesignScript nos permite seleccionar únicamente la lista en el nivel 2.
2. La lista del nivel 2 con el índice 0 incluye solo el primer conjunto de puntos Y y devuelve solo la rejilla XZ.
3. Si cambiamos el filtro Nivel a "L1", podremos ver todos los elementos del primer nivel de lista. La lista del nivel 1 con el índice 0 incluye todos los puntos 3D en una lista plana.
4. Si intentamos realizar lo mismo para "L3", solo veremos los puntos del tercer nivel de lista. La lista del nivel 3 con el índice 0 incluye solo el primer conjunto de puntos Z y devuelve solo la rejilla XY.
5. Si intentamos realizar lo mismo para "L4", solo veremos los puntos del tercer nivel de lista. La lista del nivel 4 con el índice 0 incluye solo el primer conjunto de puntos X y devuelve solo la rejilla YZ.

Aunque este ejemplo específico también se puede crear con List.Map, List@Level simplifica considerablemente la interacción, lo que facilita el acceso a los datos del nodo. Consulte la siguiente comparación entre los métodos List.Map y List@Level:

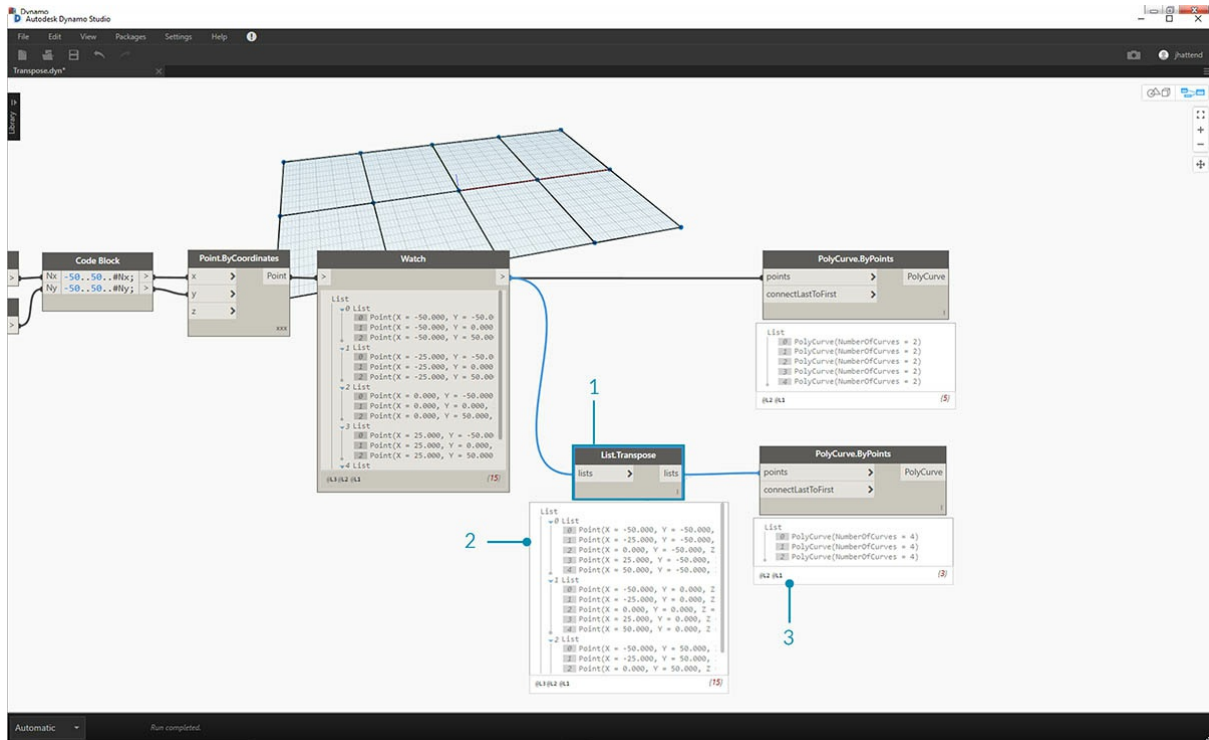


Suprimiremos los nodos *List.Count* del ejercicio anterior y pasaremos a alguna geometría para ver cómo se estructuraron los datos.

1. Conecte *PolyCurve.ByPoints* a la salida del nodo de inspección desde *Point.ByCoordinates*.
2. En la salida, se muestran cinco *PolyCurves*. Podemos ver las curvas en la vista preliminar de Dynamo. El nodo de Dynamo busca una lista de puntos (o, en este caso, una lista de listas de puntos) y crea una única *PolyCurve* a partir de ellas. Básicamente, cada lista se ha convertido en una curva en la estructura de datos.



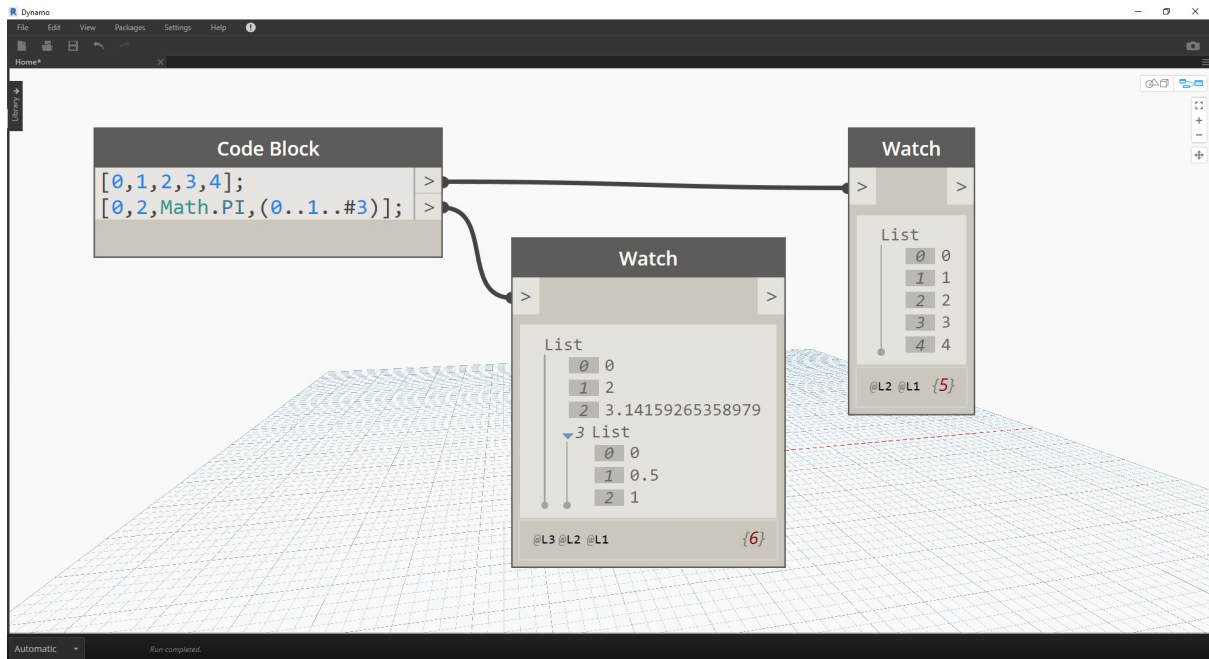
1. Si deseamos aislar una fila de curvas, debemos utilizar el nodo *List.GetItemAtIndex*.
2. Mediante el valor de *bloque de código* 2, consulte el tercer elemento de la lista principal.
3. *PolyCurve.ByPoints* nos proporciona una curva, ya que solo una lista está conectada al nodo.



1. Un nodo *List.Transpose* intercambiará todos los elementos con todas las listas de una lista de listas. Esto suena complicado, pero es la misma lógica que la transposición en Microsoft Excel: intercambiar las columnas y las filas en una estructura de datos.
2. Observe el resultado abstracto: la transposición ha cambiado la estructura de listas de cinco listas con tres elementos cada una a tres listas con cinco elementos cada una.
3. Observe el resultado geométrico: al utilizar *PolyCurve.ByPoints*, se obtienen tres PolyCurves en la dirección perpendicular a las curvas originales.

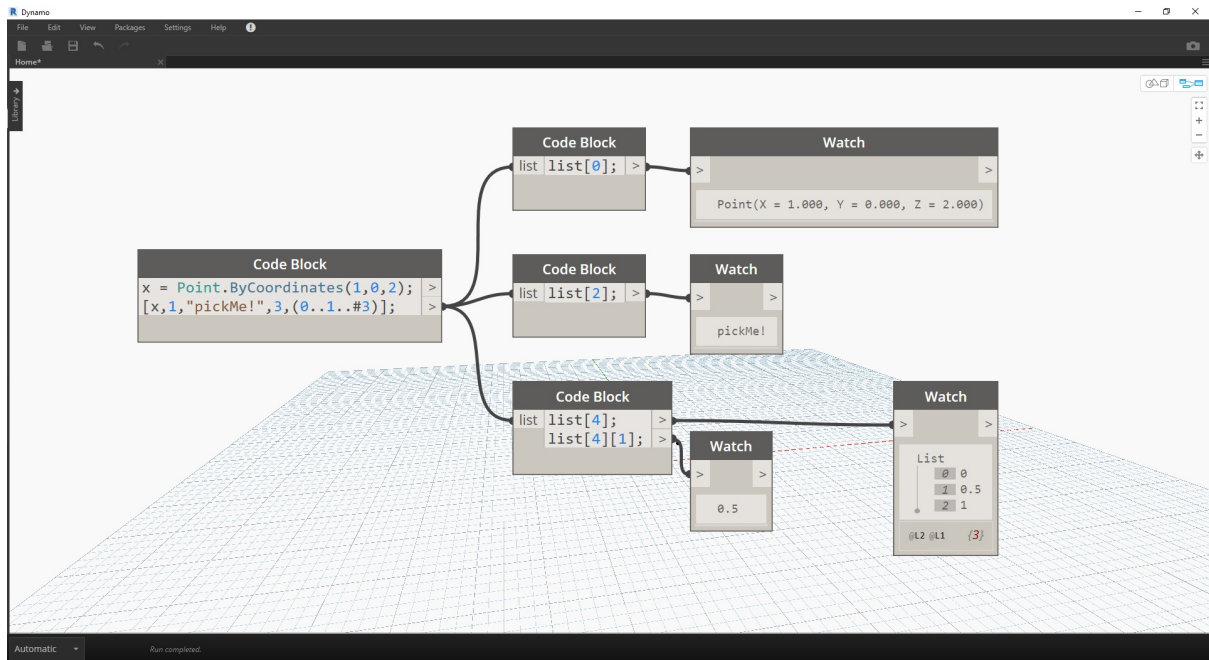
Creación del bloque de código

El modo de bloque de código abreviado utiliza "[]" para definir una lista. Esta es una forma mucho más rápida y fluida de crear una lista que con el nodo *List.Create*. El bloque de código se describe forma más detallada en el capítulo 7. Consulte la imagen mostrada a continuación para observar cómo se puede definir una lista con varias expresiones mediante el bloque de código.



Consulta del bloque de código

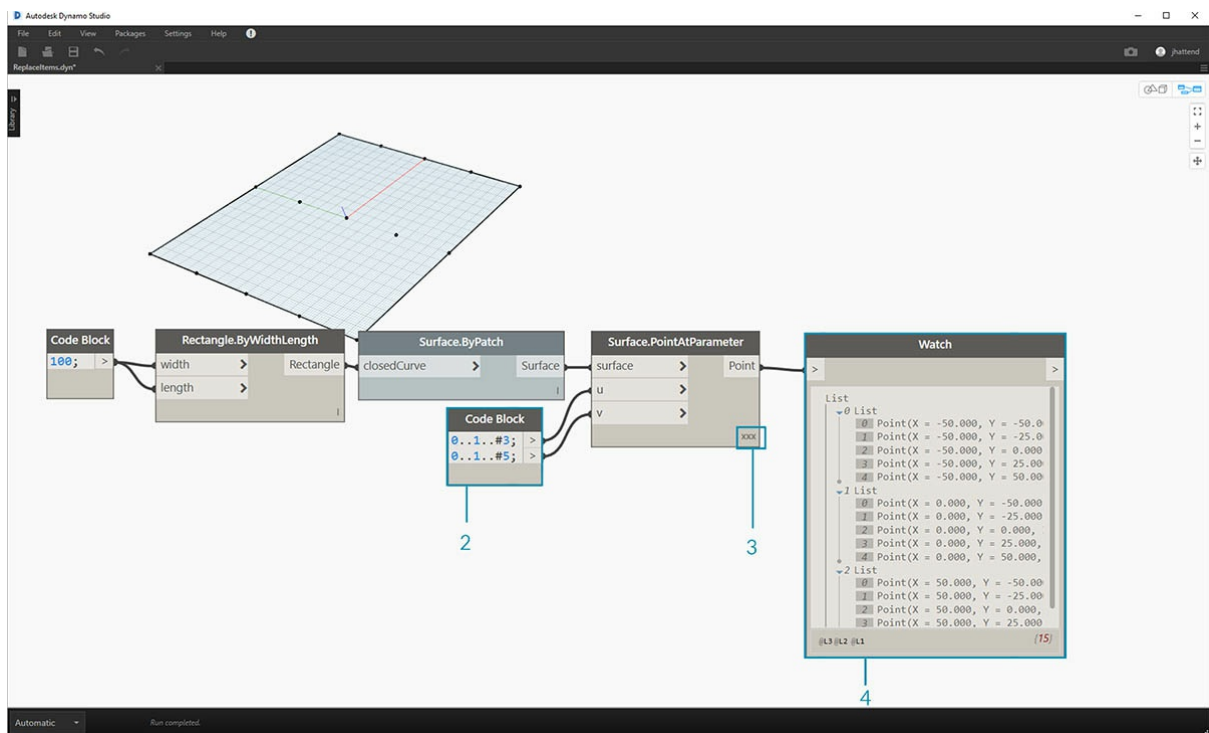
Los límites del bloque de código utilizan "[]" como método rápido y sencillo para seleccionar elementos específicos deseados en una estructura de datos compleja. Los bloques de código se describen forma más detallada en el capítulo 7. Consulte la imagen mostrada a continuación para observar cómo se puede consultar una lista con varios tipos de datos con un bloque de código.



Ejercicio: consulta e inserción de datos

Descargue el archivo de ejemplo que acompaña a este ejercicio (haga clic con el botón derecho y seleccione "Guardar vínculo como..."): [ReplaceItems.dyn](#). En el Apéndice, se incluye una lista completa de los archivos de ejemplo.

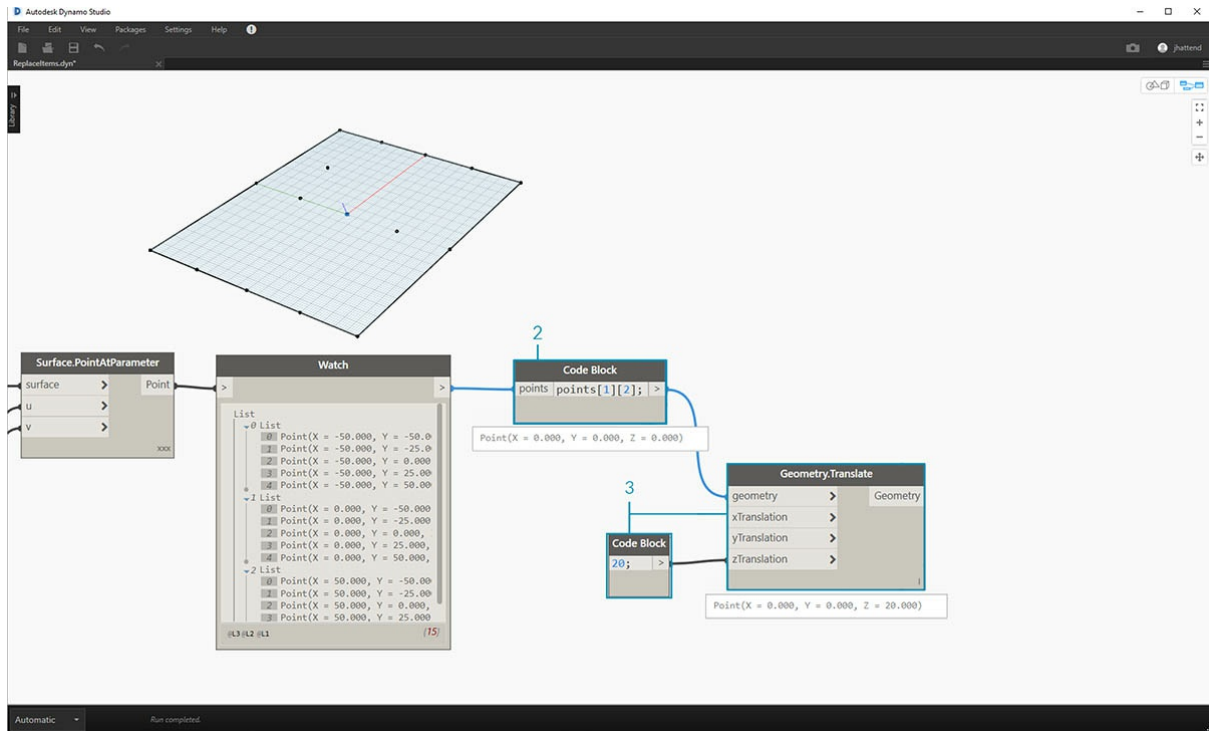
En este ejercicio, se utiliza parte de la lógica establecida en el ejercicio anterior para editar una superficie. Nuestro objetivo es intuitivo, pero la navegación por la estructura de datos es más participativa. Deseamos articular una superficie mediante el desplazamiento de un punto de control.



1. Comience con la cadena de nodos anterior. Vamos a crear una superficie básica que abarque la rejilla de Dynamo por defecto.
2. Mediante el *bloque de código*, inserte estas dos líneas de código y conéctelas a las entradas *u* y *v* de *Surface.PointAtParameter* respectivamente:

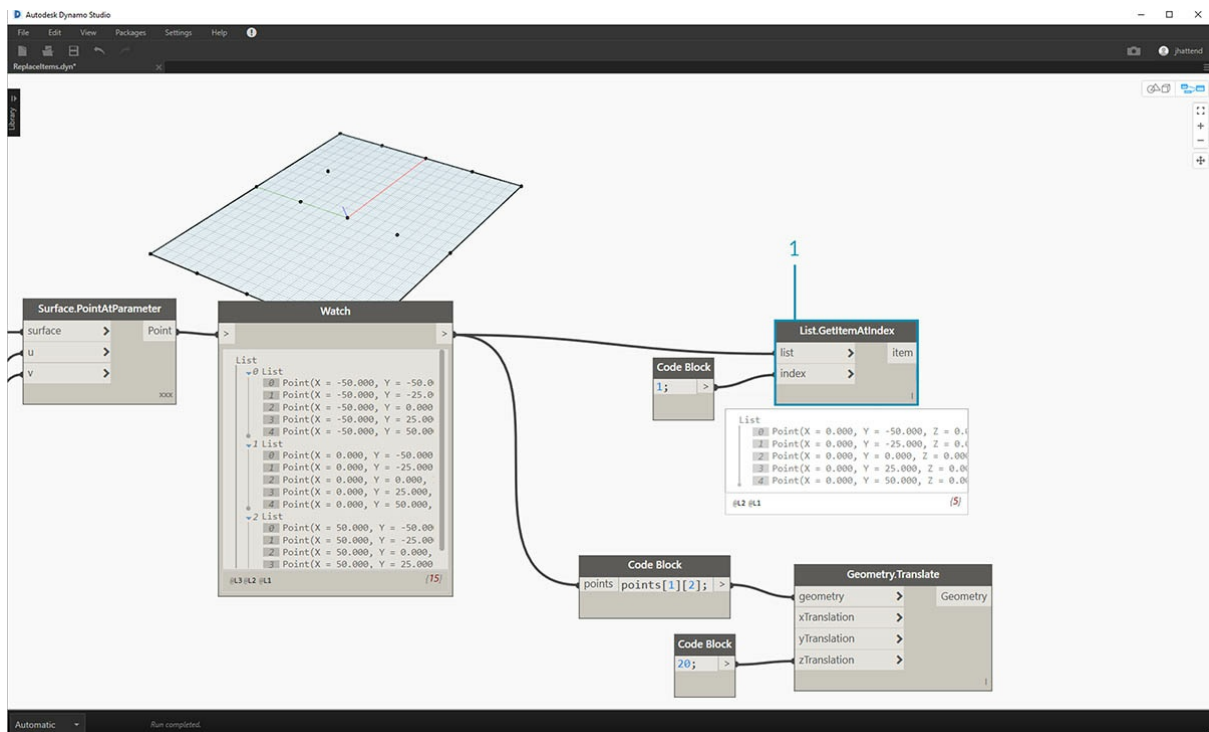
```
-50..50..#3;
-50..50..#5;
```

1. Asegúrese de establecer el encaje de *Surface.PointAtParameter* en "Producto vectorial".
2. En el nodo *Watch*, se muestra que tenemos una lista de tres listas con cinco elementos cada una.

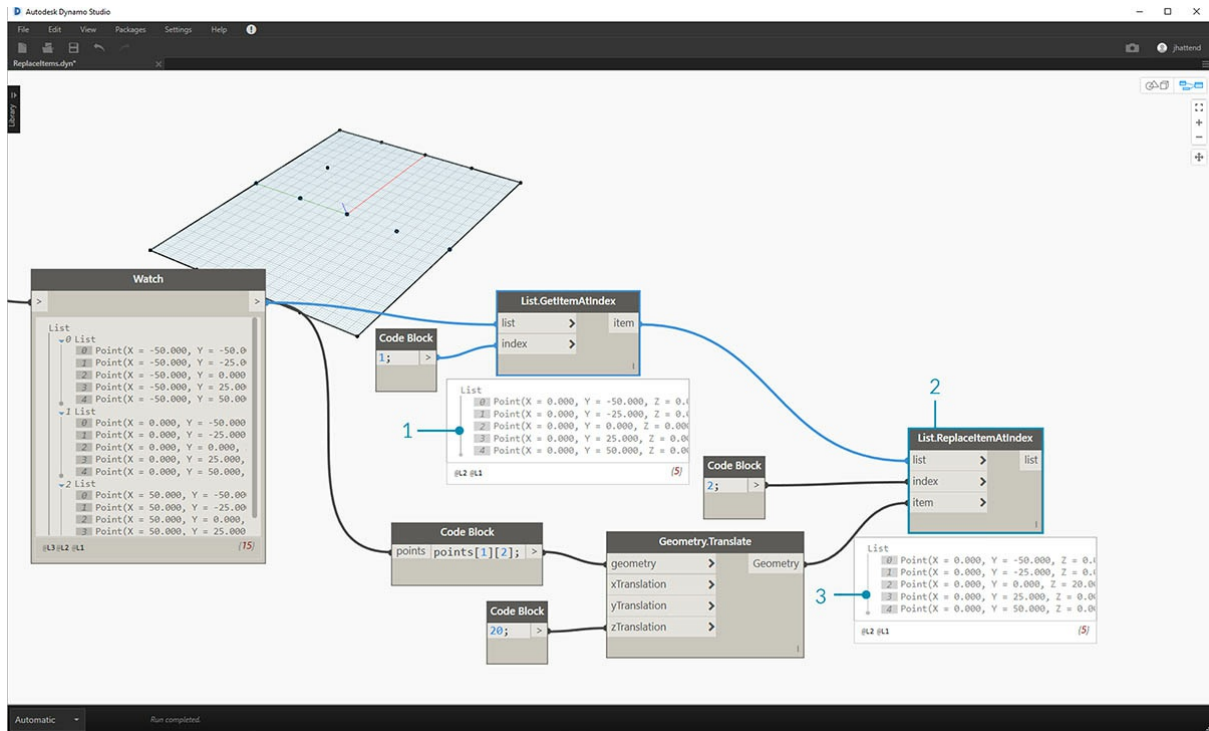


En este paso, deseamos consultar el punto central de la rejilla que hemos creado. Para ello, seleccionaremos el punto medio de la lista central. Tiene sentido, ¿verdad?

1. Para confirmar que este es el punto adecuado, también podemos hacer clic por los elementos del nodo de visualización para confirmar que se trata del objetivo correcto.
2. Mediante el *bloque de código*, escribiremos una línea de código básica para consultar una lista de listas: `points[1][2];`
3. Con *Geometry.Translate*, desplazaremos el punto seleccionado hacia arriba en la dirección Z en 20 unidades.

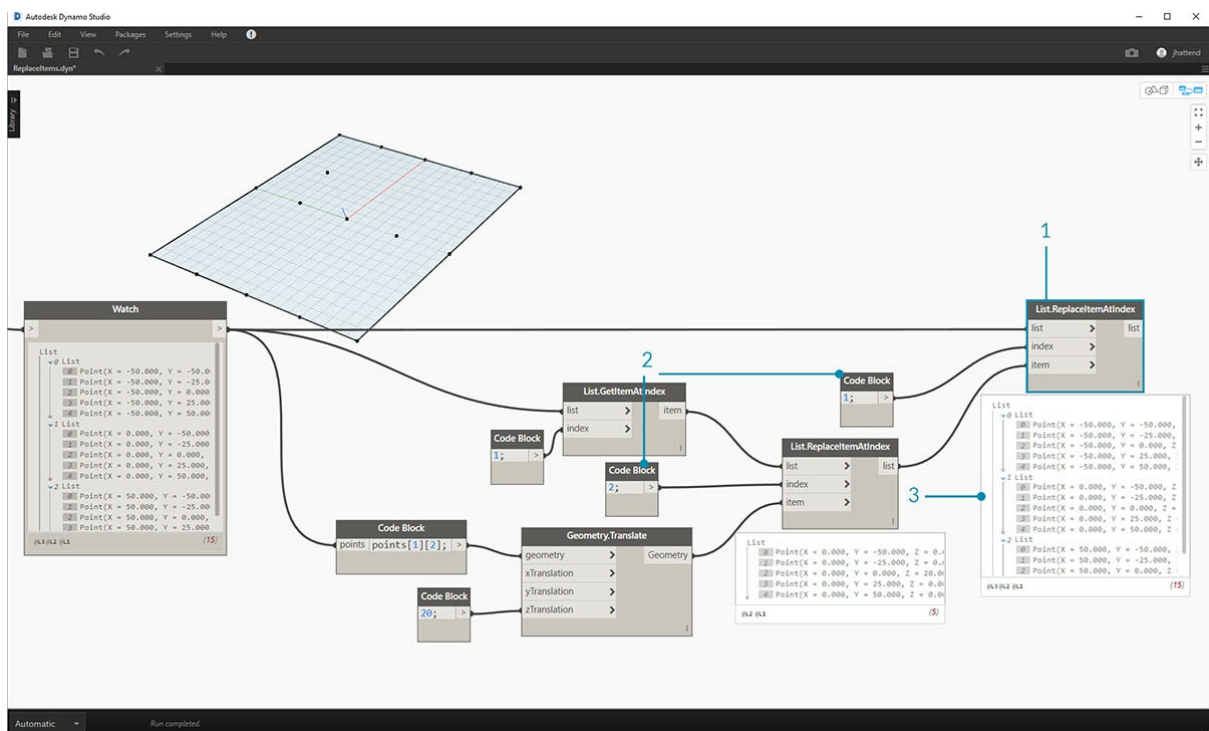


1. Vamos a seleccionar la fila central de puntos con un nodo *List.GetItemAtIndex*. Nota: al igual que en un paso anterior, también podemos consultar la lista con *bloque de código* mediante una línea de `points[1];`



Hasta ahora, hemos consultado correctamente el punto central y lo hemos desplazado hacia arriba. Ahora deseamos insertar de nuevo este punto desplazado en la estructura de datos original.

1. Deseamos reemplazar primero el elemento de la lista que hemos aislado en un paso anterior.
2. Con *List.ReplaceItemAtIndex*, sustituiremos el elemento central mediante el uso de un índice "2" con el elemento de reemplazo conectado al punto desplazado (*Geometry.Translate*).
3. En la salida, se muestra que hemos introducido el punto desplazado en el elemento central de la lista.



Ahora que hemos modificado la lista, debemos volver a insertarla en la estructura de datos original: la lista de listas.

1. Siguiendo la misma lógica, utilice *List.ReplaceItemAtIndex* para reemplazar la lista central por la lista modificada.
2. Observe que los bloques de código que definen el índice de estos dos nodos son 1 y 2, que coinciden con la consulta original del bloque de código (*points[1][2]*).
3. Al seleccionar la lista en el índice 1, veremos la estructura de datos resaltada en la vista preliminar de Dynamo. Hemos fusionado correctamente el punto desplazado con la estructura de datos original.

1. Conecte *Surface.ByLoft* a la salida desde *NurbsCurve.ByPoints*. Ahora tenemos una superficie modificada. Podemos cambiar el valor Z original de la geometría. Realice el traslado y vea cómo se actualiza la geometría.

Listas de n dimensiones

Listas de n dimensiones

Adentrándonos aún más en el laberinto, vamos a agregar más niveles a la jerarquía. La estructura de datos puede ampliarse mucho más allá de una lista de listas bidimensional. Dado que las listas son elementos en sí mismos en Dynamo, podemos crear datos con tantas dimensiones como sea posible.

La analogía con la que trabajaremos aquí son las muñecas rusas. Cada lista se puede considerar como un contenedor que contiene varios elementos. Cada lista tiene sus propias propiedades y también se considera su propio objeto.



Un conjunto muñecas rusas (foto de [Zeta](#)) es una analogía de las listas de n dimensiones. Cada capa representa una lista y cada lista contiene elementos. En el caso de Dynamo, cada contenedor puede tener varios contenedores dentro (que representan los elementos de cada lista).

Las listas de n dimensiones son difíciles de explicar visualmente, pero hemos configurado algunos ejercicios en este capítulo que se centran en trabajar con listas que van más allá de las dos dimensiones.

Asignación y combinaciones

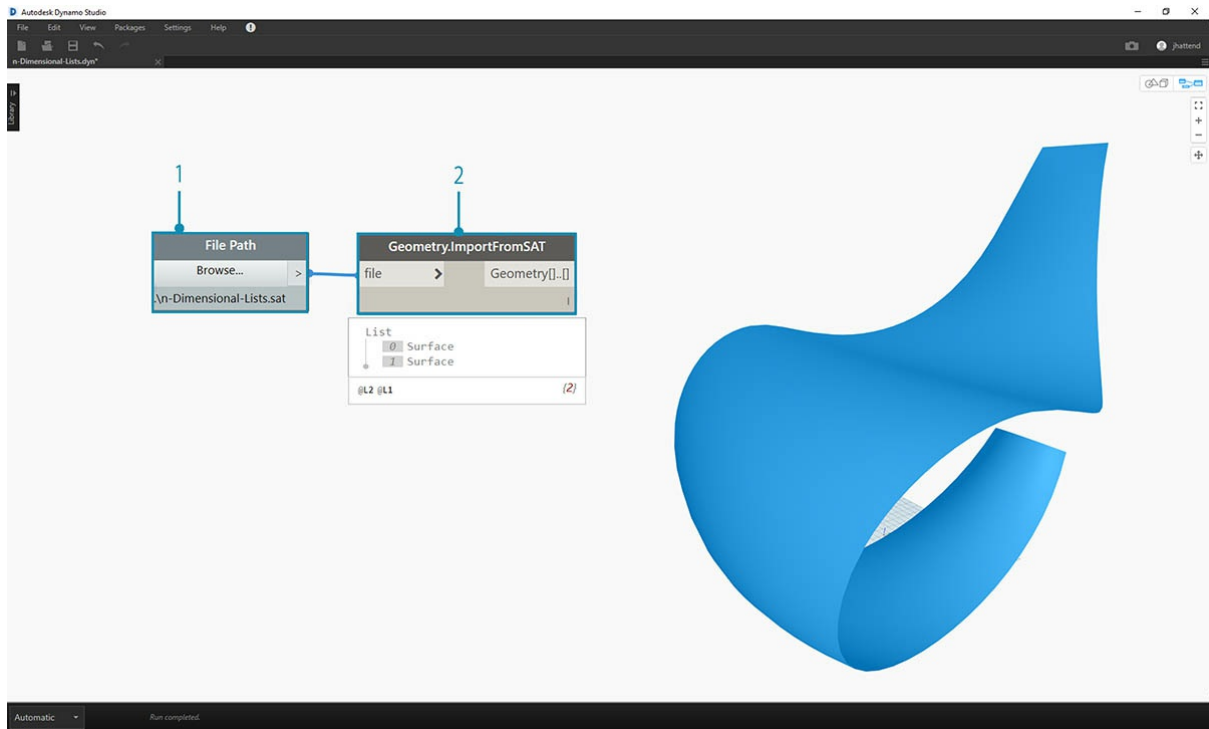
La asignación es, probablemente, la parte más compleja de la administración de datos en Dynamo y es especialmente relevante al trabajar con jerarquías de listas complejas. Con la serie de ejercicios incluidos a continuación, mostraremos cuándo utilizar la asignación y las combinaciones a medida que los datos se convierten en datos multidimensionales.

Las introducciones preliminares a List.Map y List.Combine se encuentran en la sección anterior. En el último ejercicio de los que se incluyen a continuación, usaremos estos nodos en una estructura de datos compleja.

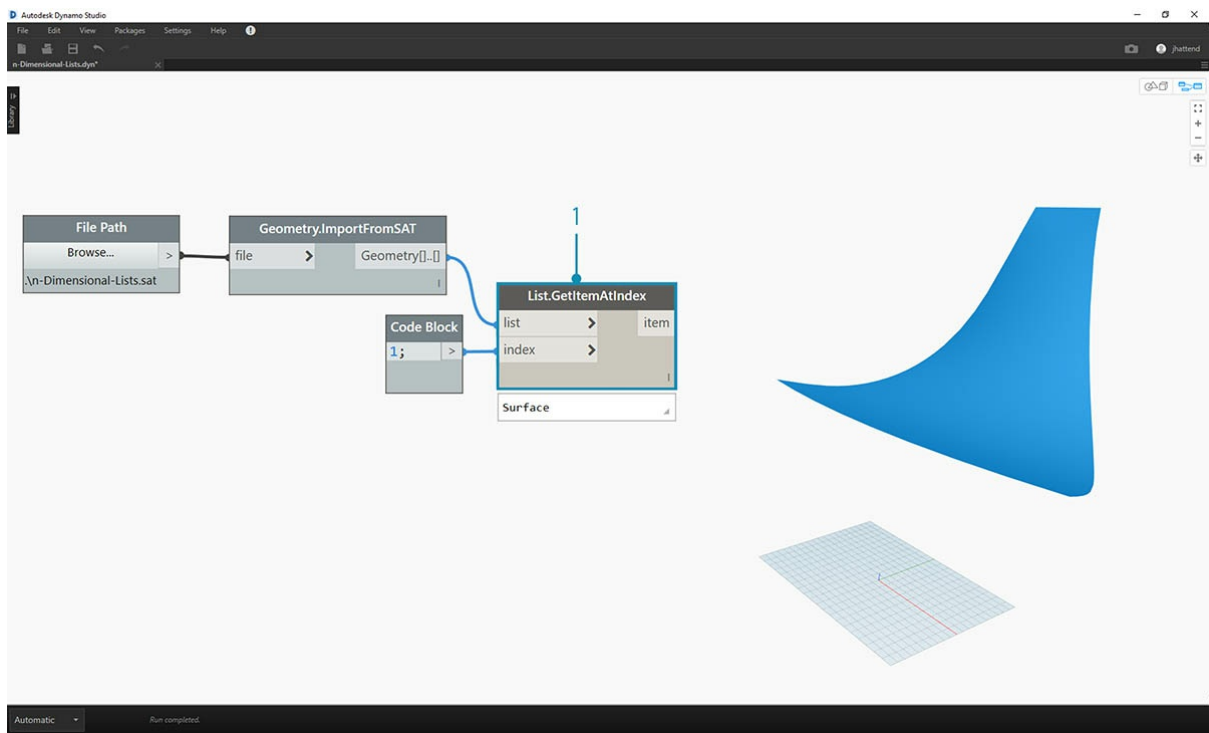
Ejercicio - Listas 2D: nivel básico

Descargue los archivos de ejemplo que acompañan a este ejercicio (haga clic con el botón derecho y seleccione "Guardar enlace como..."). En el Apéndice se incluye una lista completa de los archivos de ejemplo. [1.n-Dimensional-Lists.dyn](#) [2.n-Dimensional-Lists.sat](#)

Este es el primero de una serie de tres ejercicios que se centra en la articulación de la geometría importada. Cada parte de esta serie de ejercicios aumentará en la complejidad de la estructura de datos.

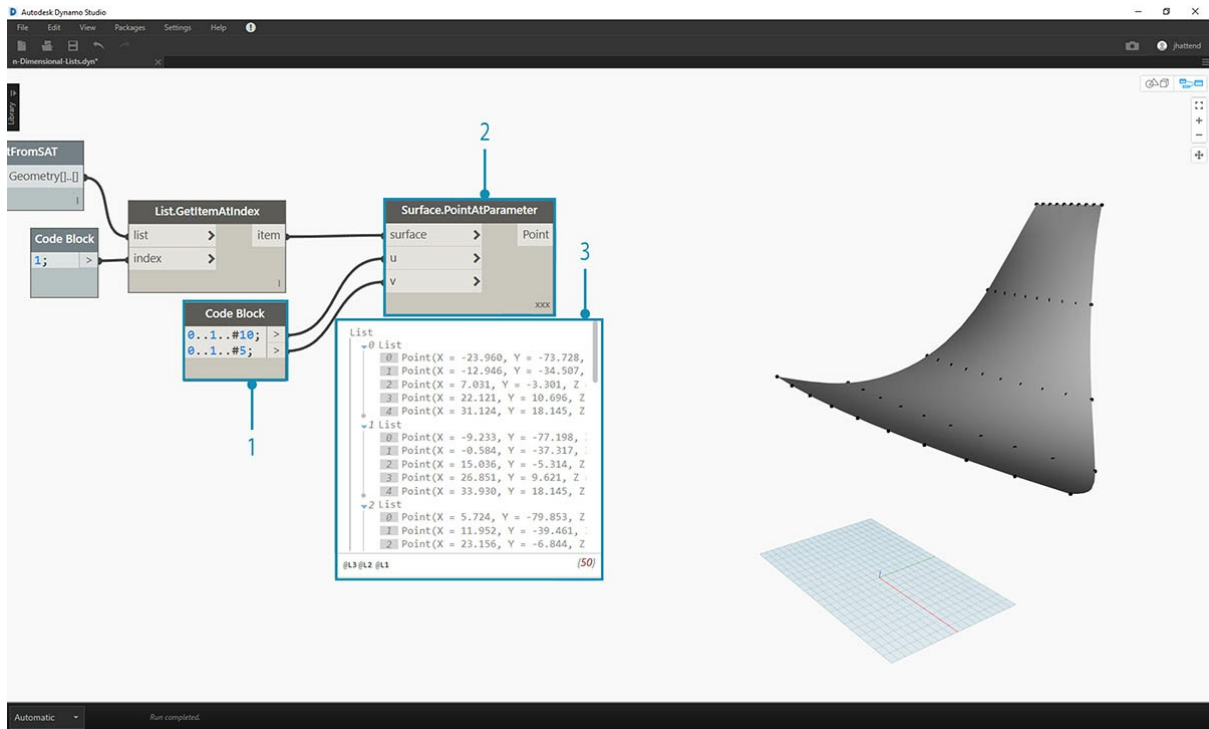


1. Comencemos con el archivo .sat de la carpeta de archivos del ejercicio. Se puede acceder a este archivo mediante el nodo *File Path*.
2. Con *Geometry.ImportFromSAT*, la geometría se importa a nuestra vista preliminar de Dynamo como dos superficies.



En este sencillo ejercicio, vamos a trabajar con una de las superficies.

1. Seleccionaremos el índice de 1 para capturar la superficie superior. Esta acción se realiza con el nodo *List.GetItemAtIndex*.

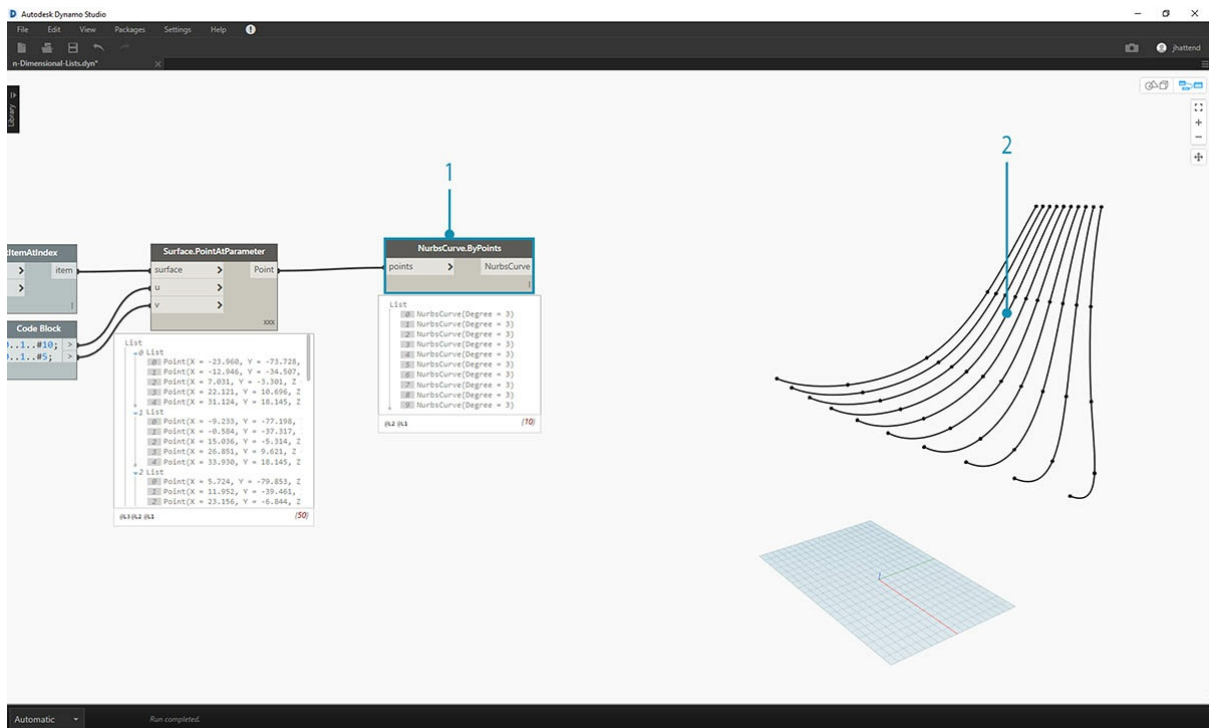


El siguiente paso es dividir la superficie en una rejilla de puntos.

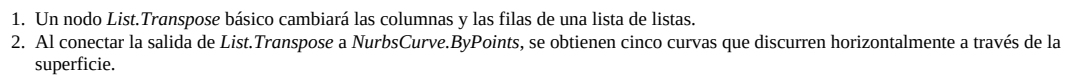
1. Mediante el bloque de código, inserte estas dos líneas de código:

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

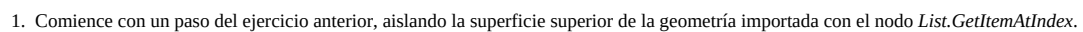
1. Con *Surface.PointAtParameter*, conecte los dos valores del bloque de código a *u* y *v*. Cambie el encaje de este nodo a "Producto vectorial".
2. La salida muestra la estructura de datos, que también está visible en la vista preliminar de Dynamo.

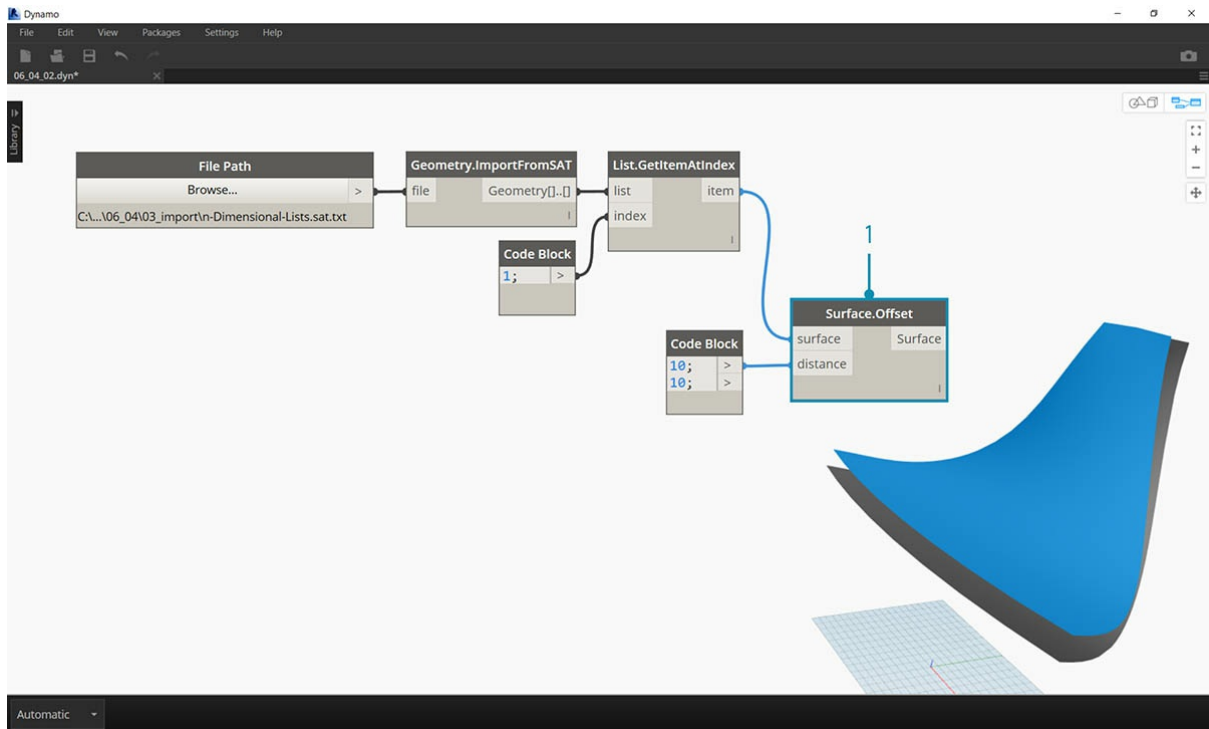


1. Para obtener información sobre cómo se organiza la estructura de datos, conectemos un nodo *NurbsCurve.ByPoints* a la salida de *Surface.PointAtParameter*.
2. Observe que hay diez curvas que discurren verticalmente a lo largo de la superficie.

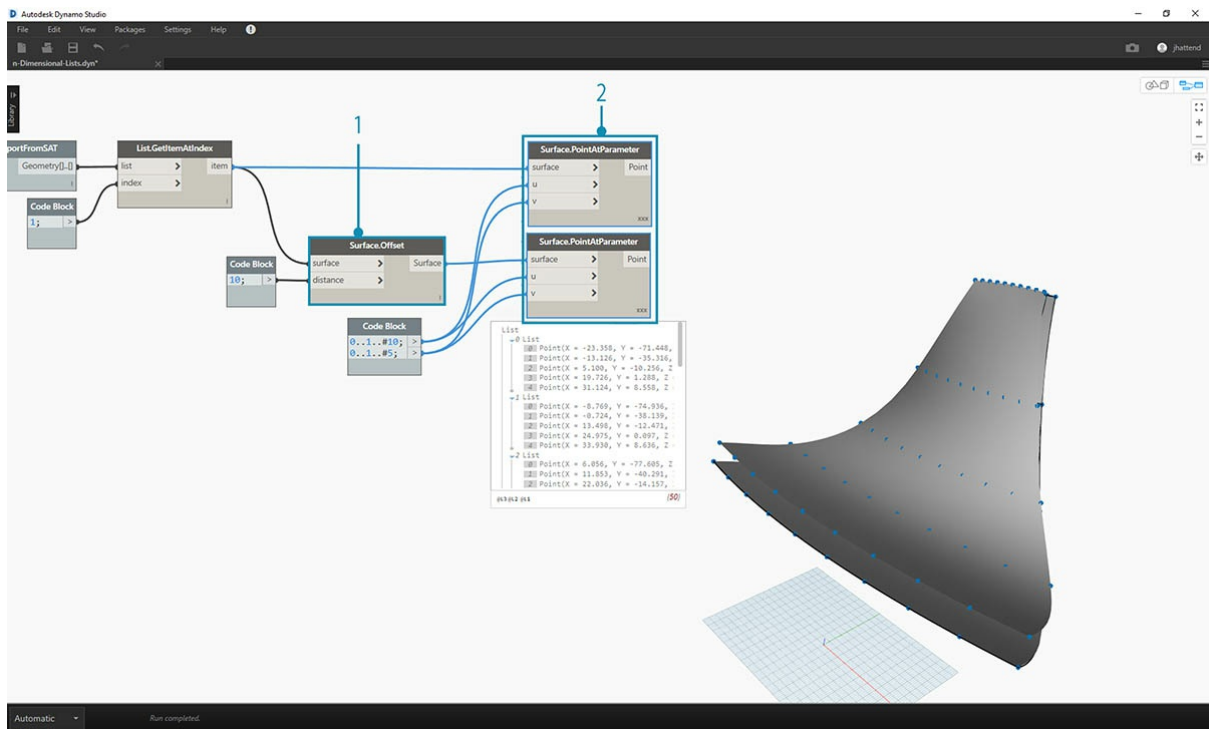


Aumentemos la complejidad. Supongamos que deseamos realizar una operación en las curvas creadas en el ejercicio anterior. Tal vez nos gustaría relacionar estas curvas con otra superficie y solevar entre ellas. Esto requiere más atención en la estructura de datos, pero la lógica subyacente es la misma.





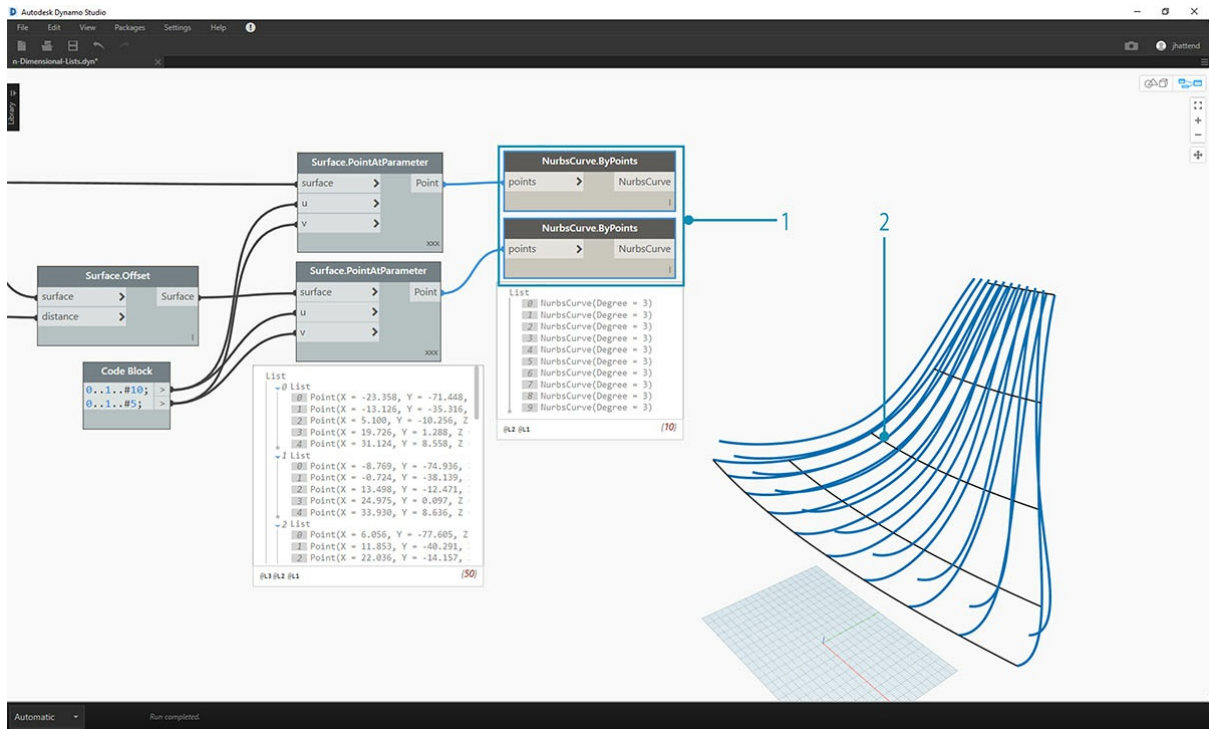
1. Con *Surface.Offset*, desfase la superficie con un valor de 10.



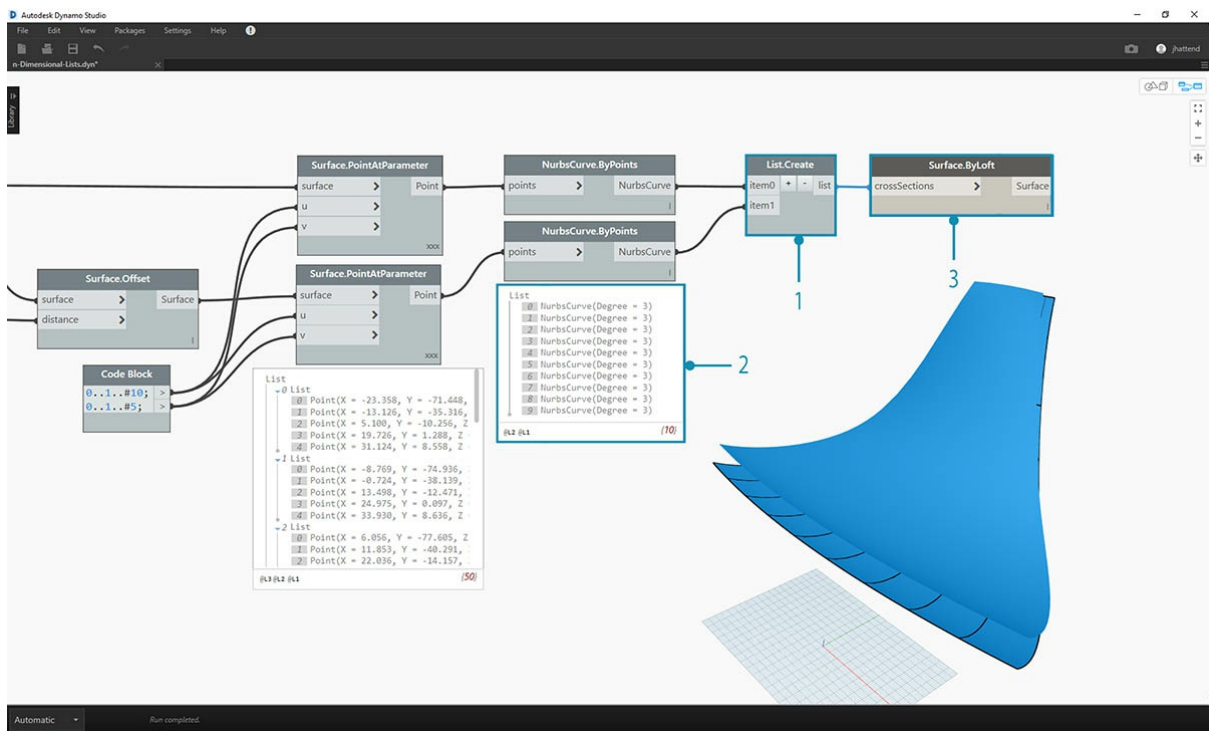
1. Del mismo modo que en el ejercicio anterior, defina un *bloque de código* con estas dos líneas de código:

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

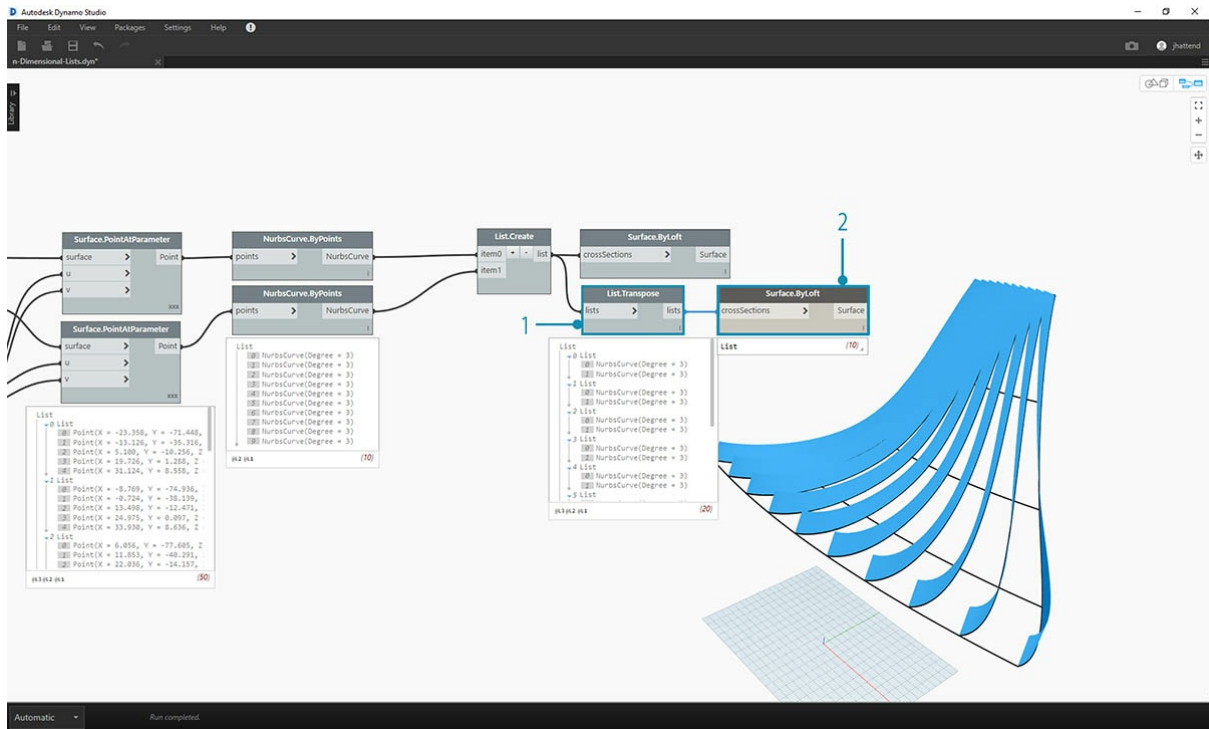
1. Conecte estas salidas a dos nodos *Surface.PointAtParameter*, cada uno con el *encaje* establecido en "Producto vectorial". Uno de estos nodos está conectado a la superficie original, mientras que el otro está conectado a la superficie desfasada.



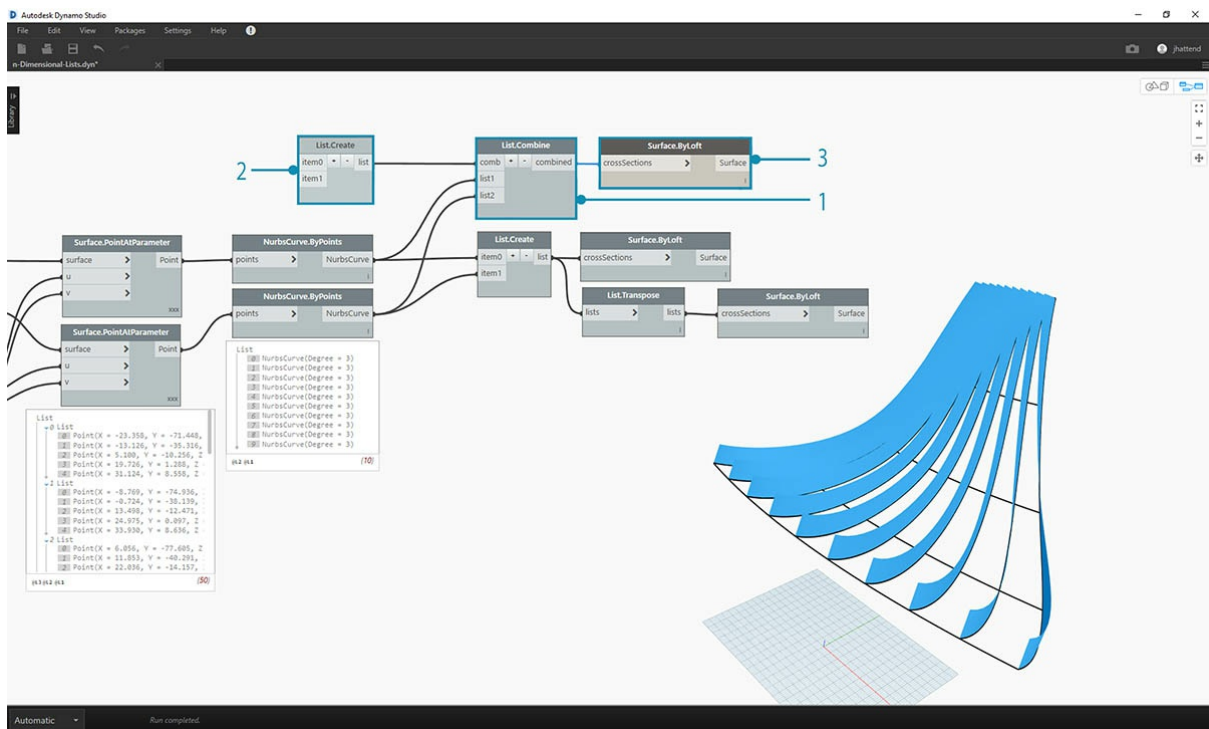
1. Como en el ejercicio anterior, conecte la salida a dos nodos *NurbCurve.ByPoints*.
2. Nuestra vista preliminar de Dynamo muestra dos curvas correspondientes a dos superficies.



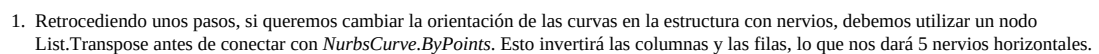
1. Con *List.Create*, podemos combinar los dos conjuntos de curvas en una lista de listas.
2. Observe en la salida que tenemos dos listas con diez elementos cada una que representan cada conjunto de conexiones de curvas NURBS.
3. Al ejecutar *Surface.ByLoft*, podemos visualizar con claridad esta estructura de datos. El nodo solleva todas las curvas de cada sublist.



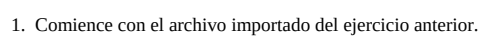
1. Al utilizar *List.Transpose*, recuerde que estamos volteando todas las columnas y filas. Este nodo transferirá dos listas de diez curvas a diez listas de dos curvas. Ahora cada curva NURBS está relacionada con la curva contigua de la otra superficie.
2. Con *Surface.ByLoft*, llegamos a una estructura con nervios.

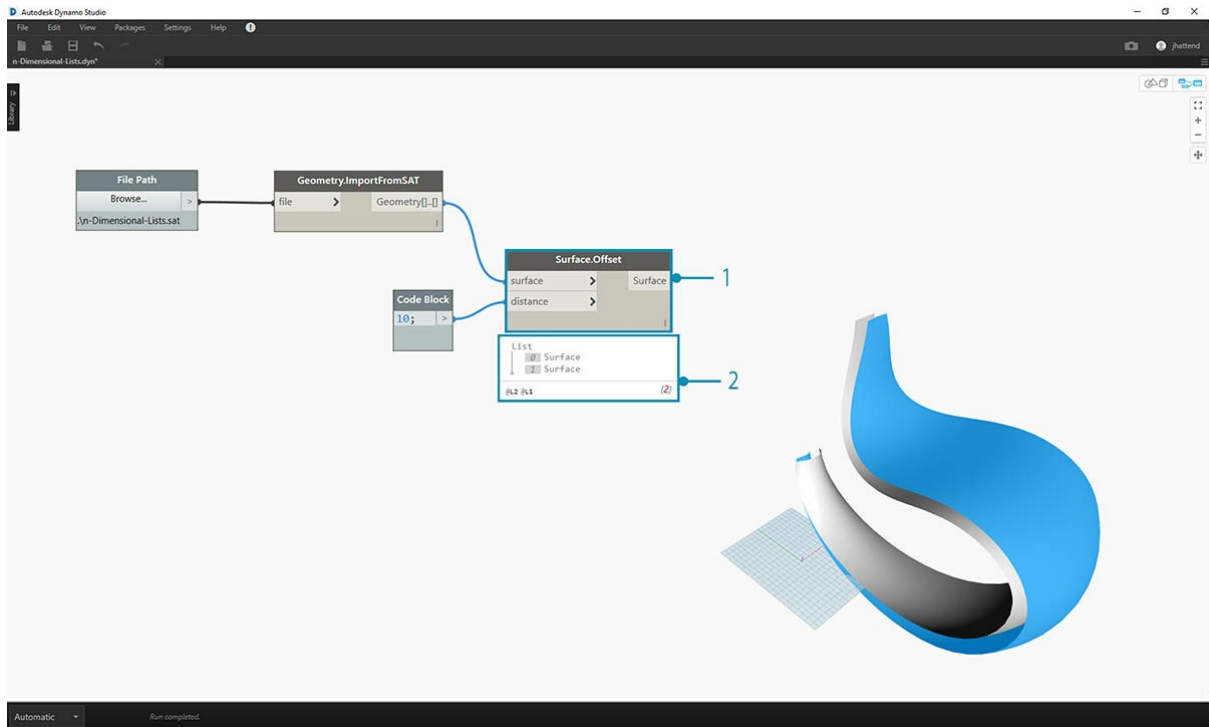


1. Una alternativa a *List.Transpose* es utilizar *List.Combine*. Esta acción pondrá en funcionamiento "combinador" en cada sublista.
2. En este caso, utilizaremos *List.Create* como "combinador", que creará una lista de cada elemento de las sublistas.
3. Con el nodo *Surface.ByLoft*, se obtienen las mismas superficies que en el paso anterior. La transposición es más fácil de utilizar en este caso, pero cuando la estructura de datos se vuelve más compleja, *List.Combine* es más fiable.

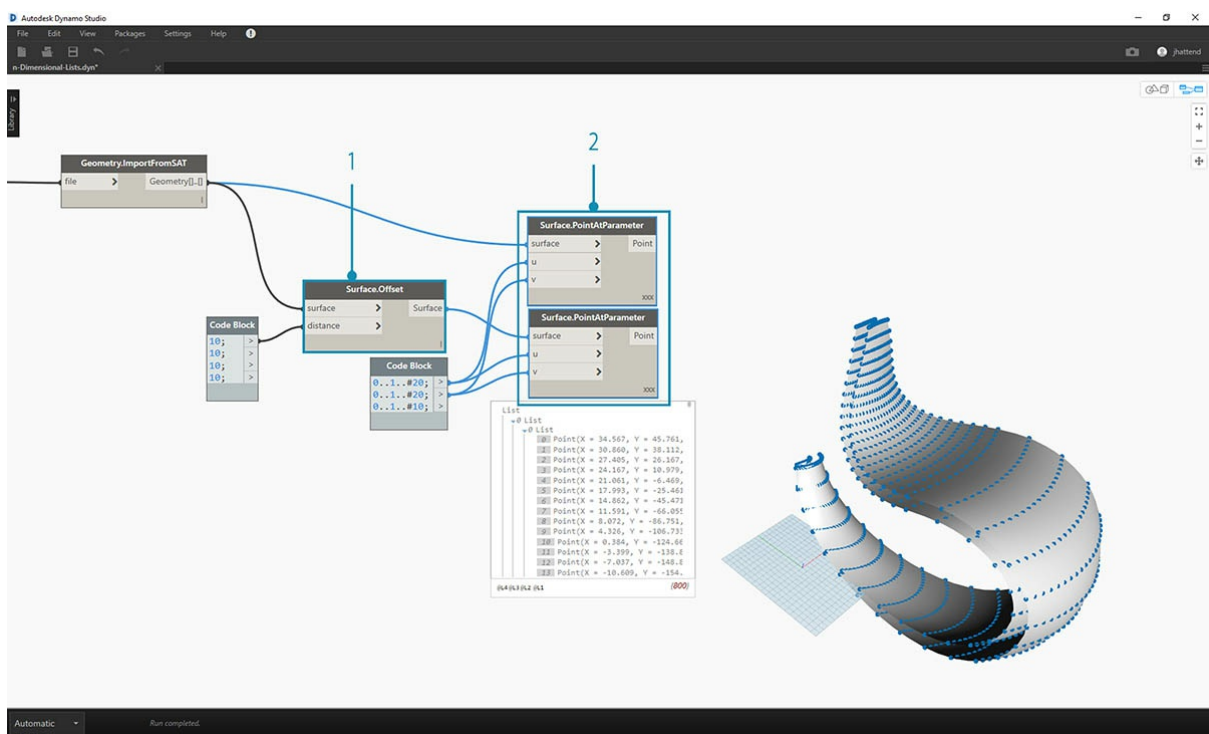


Ahora, vamos a ir un paso más allá. En este ejercicio, trabajaremos con ambas superficies importadas y crearemos una jerarquía de datos compleja. Aun así, nuestro objetivo es completar la misma operación con la misma lógica subyacente.





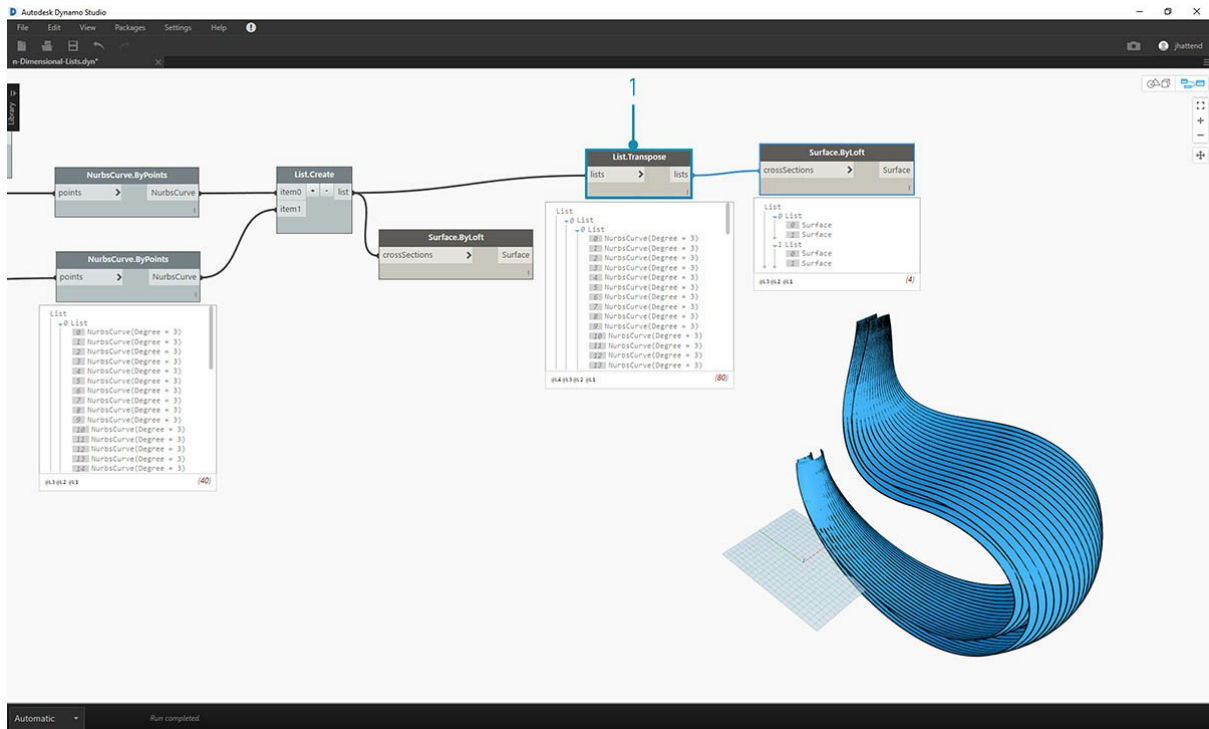
1. Como en el ejercicio anterior, utilice el nodo *Surface.Offset* para desfasar por un valor de 10.
2. Observe en la salida que hemos creado dos superficies con el nodo de desfase.



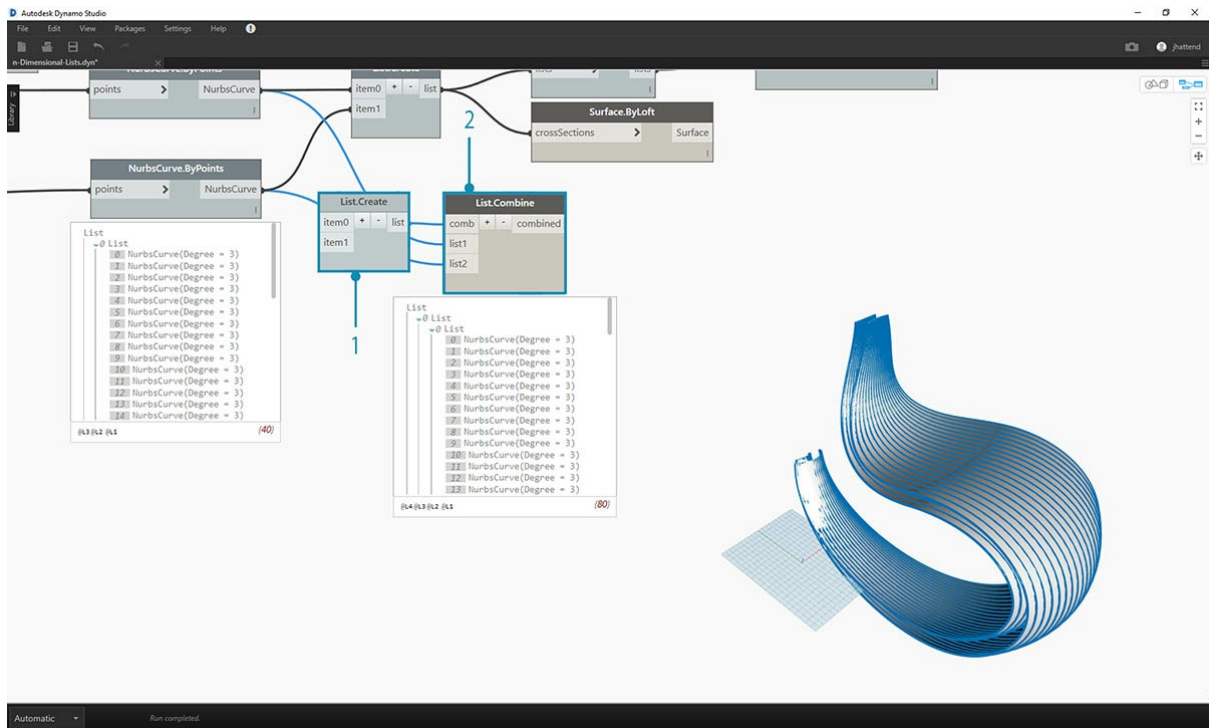
1. Del mismo modo que en el ejercicio anterior, defina un bloque de código con estas dos líneas de código:

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

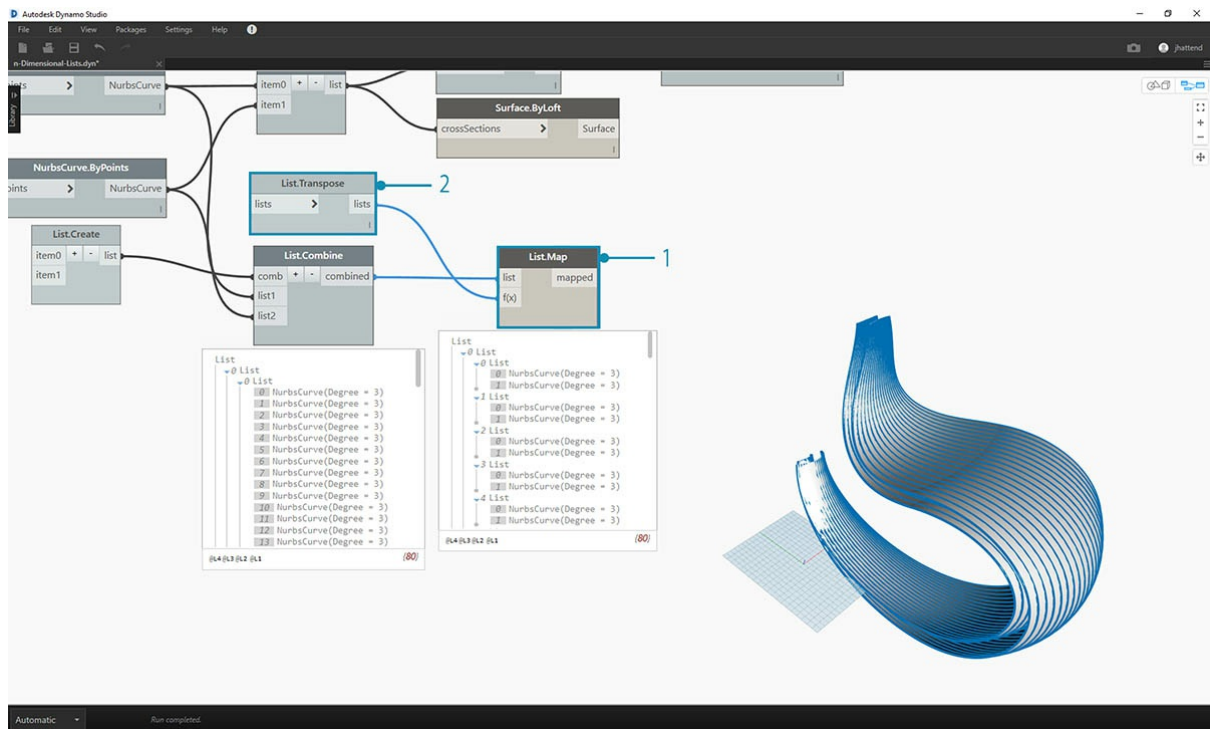
1. Conecte estas salidas a dos nodos *Surface.PointAtParameter*, cada uno con el *encaje* establecido en "Producto vectorial". Uno de estos nodos está conectado a las superficies originales, mientras que el otro está conectado a las superficies desfasadas.



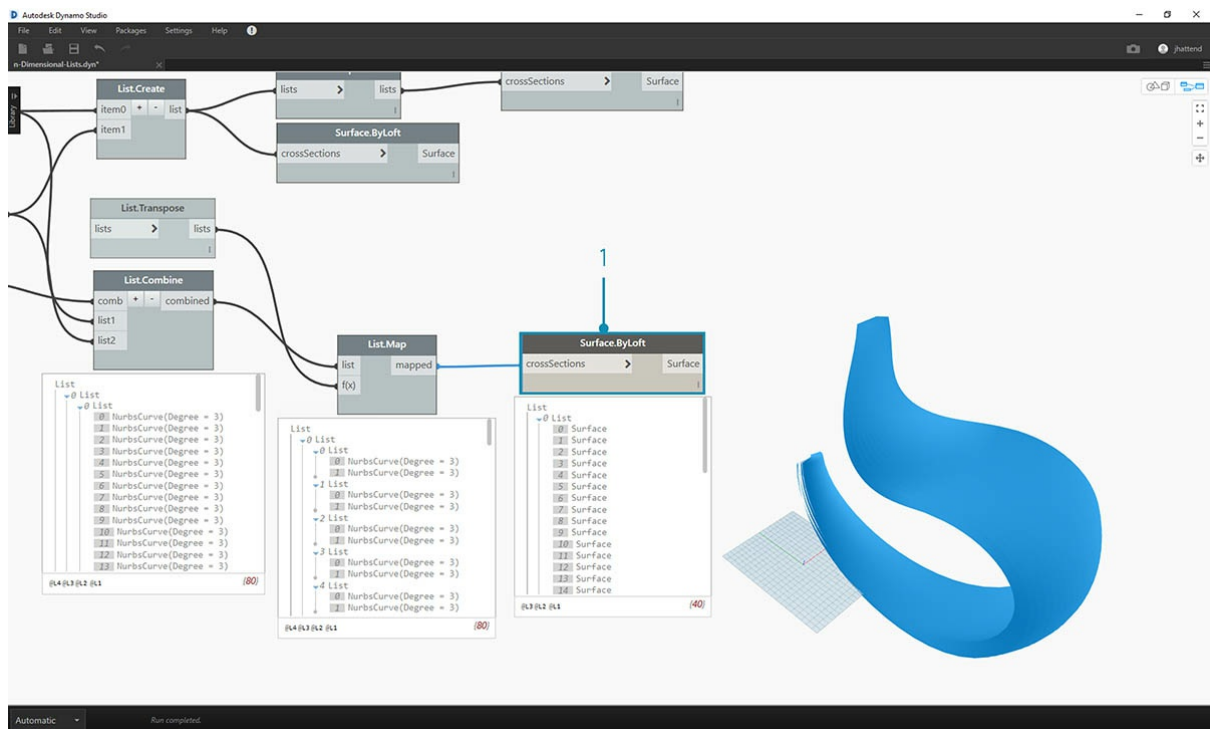
1. En el ejercicio anterior, pudimos utilizar *List.Transpose* para crear una estructura con nervios. Esto no funcionaría aquí. La transposición se debe utilizar en una lista bidimensional y, puesto que tenemos una lista tridimensional, una operación de "voltear columnas y filas" no funcionaría de manera tan sencilla. Recuerde que las listas son objetos, de modo que *List.Transpose* voltearía las listas sin sublistas, pero no voltearía las curvas NURBS ubicadas una lista más abajo en la jerarquía.



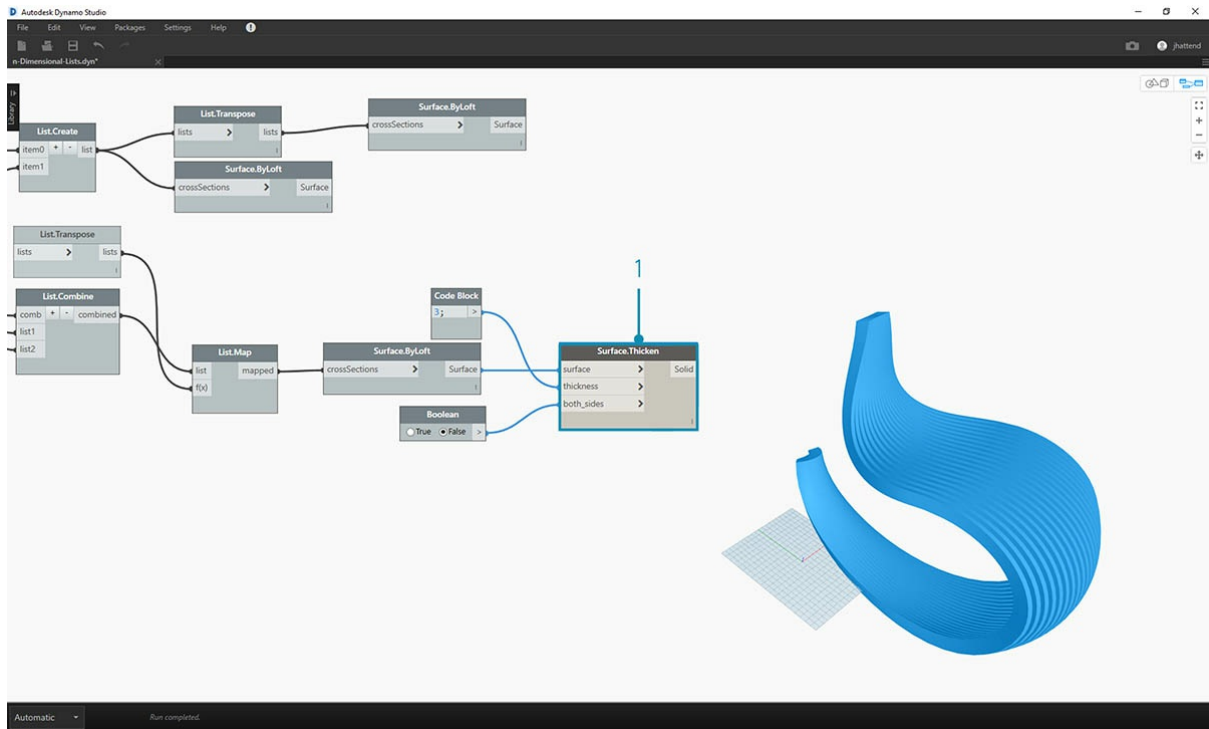
1. *List.Combine* funcionará mejor aquí. Debemos usar los nodos *List.Map* y *List.Combine* cuando trabajamos con estructuras de datos más complejas.
2. Al utilizar *List.Create* como "combinador", creamos una estructura de datos que funcionará mejor para nosotros.



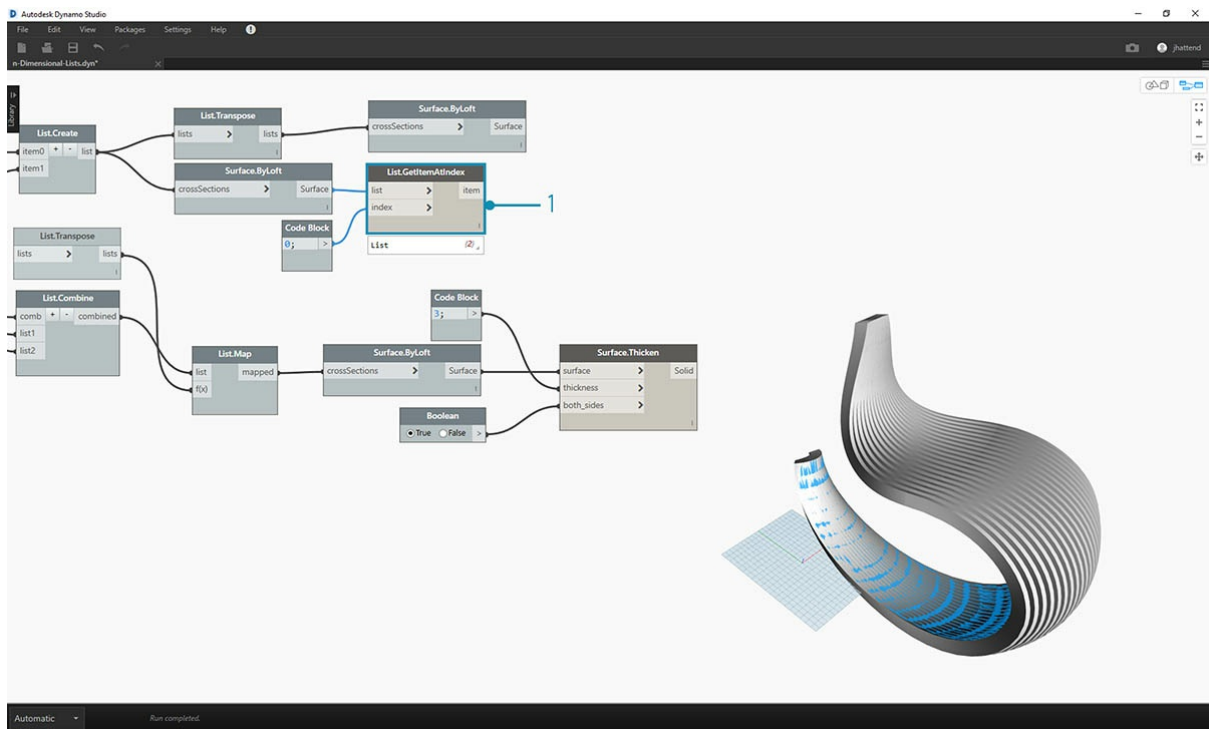
1. La estructura de datos aún debe transponerse un paso hacia abajo en la jerarquía. Para ello, utilizaremos *List.Map*. Funciona como *List.Combine*, salvo que utiliza solo una lista de entrada, en lugar de dos o más.
2. La función que aplicaremos a *List.Map* es *List.Transpose*, que volteará las columnas y las filas de las sublistas de nuestra lista principal.



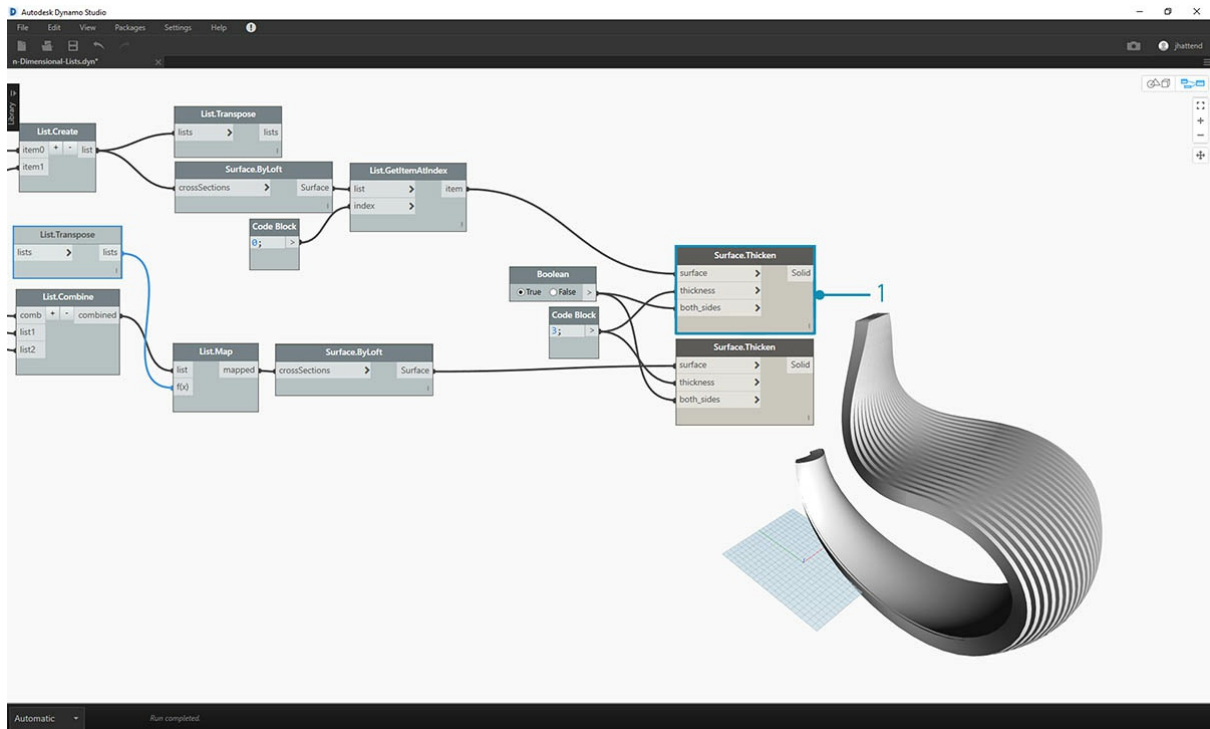
1. Por último, podemos solear las curvas NURBS junto con una jerarquía de datos adecuada, lo que nos proporciona una estructura con nervios.



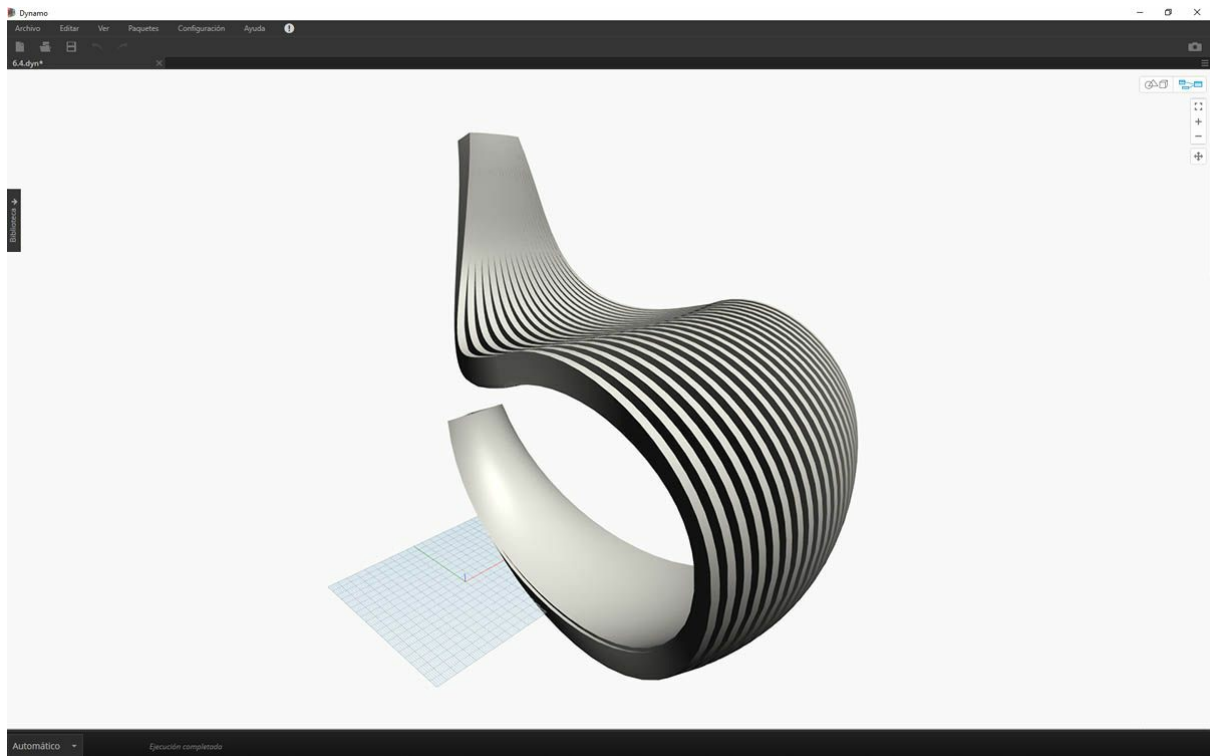
1. Añadiremos profundidad a la geometría con un nodo *Surface.Thicken*.



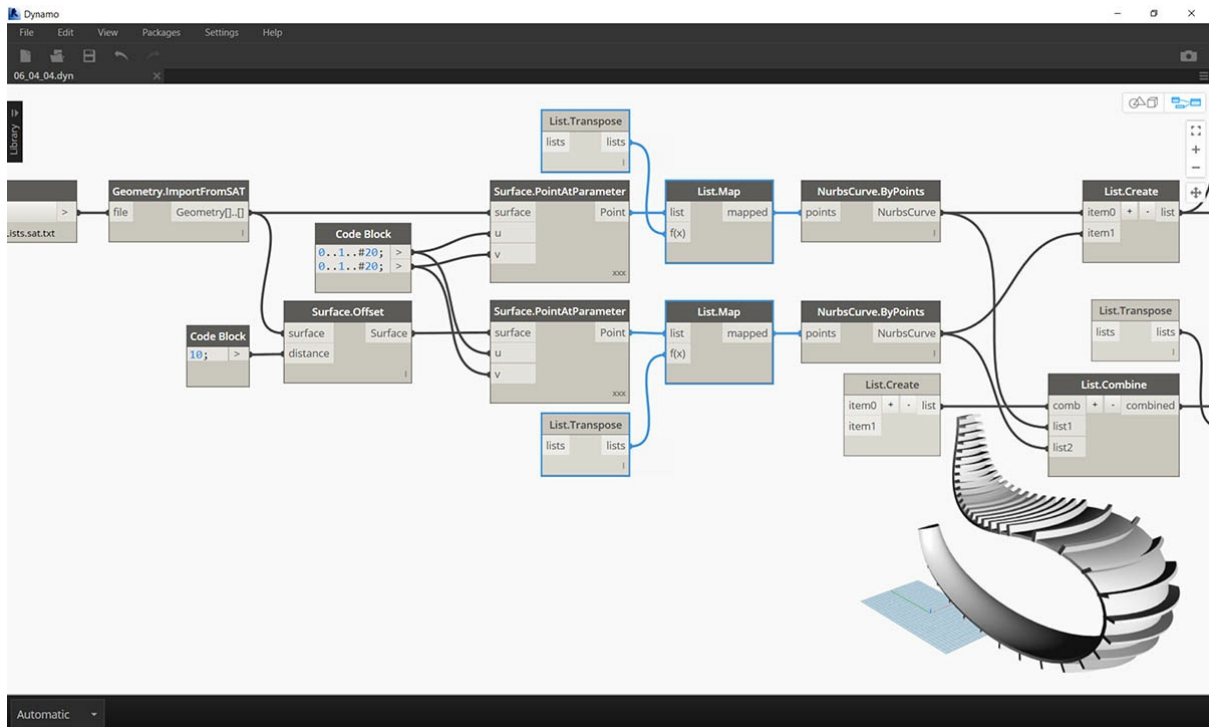
1. Es conveniente añadir una superficie que respalde esta estructura, por lo que utilizaremos *List.GetItemAtIndex* para seleccionar la superficie posterior de entre las superficies sollevadas de los pasos anteriores.



1. Y si engrosamos las superficies seleccionadas, la articulación está completa.

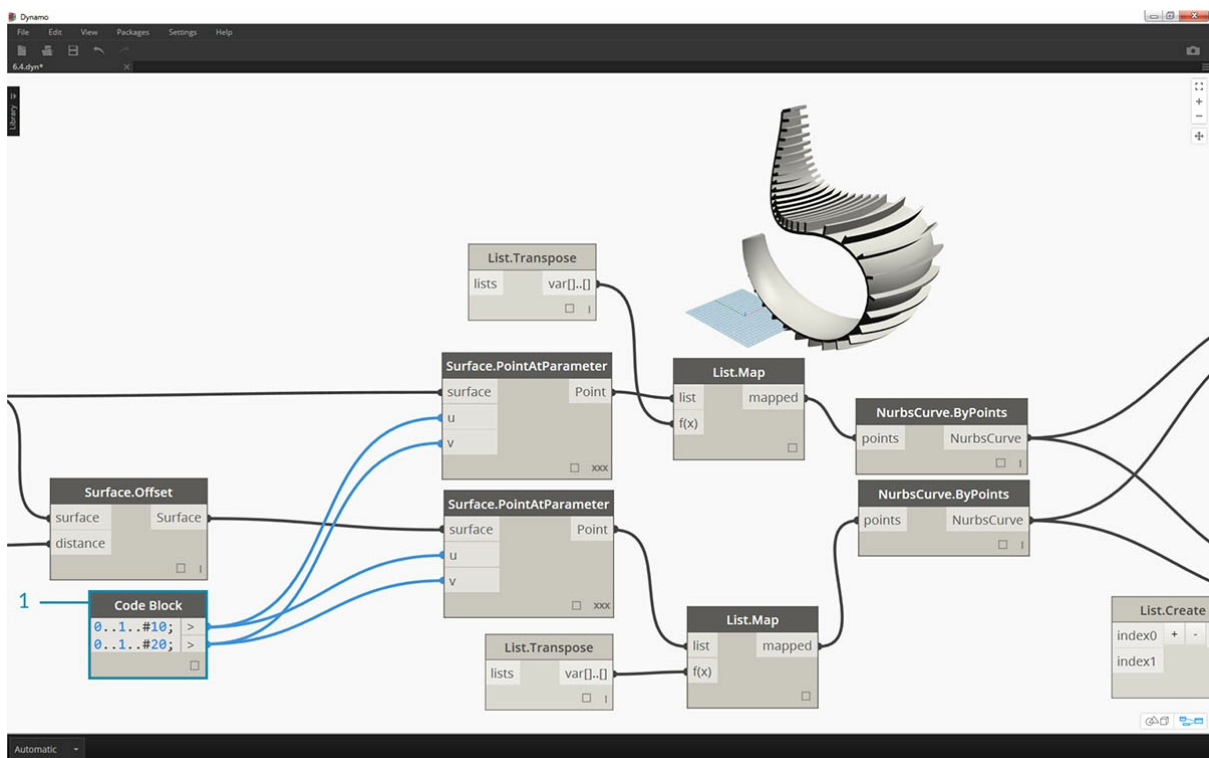


No es la mecedora más cómoda del mercado, pero hay muchos datos en ella.



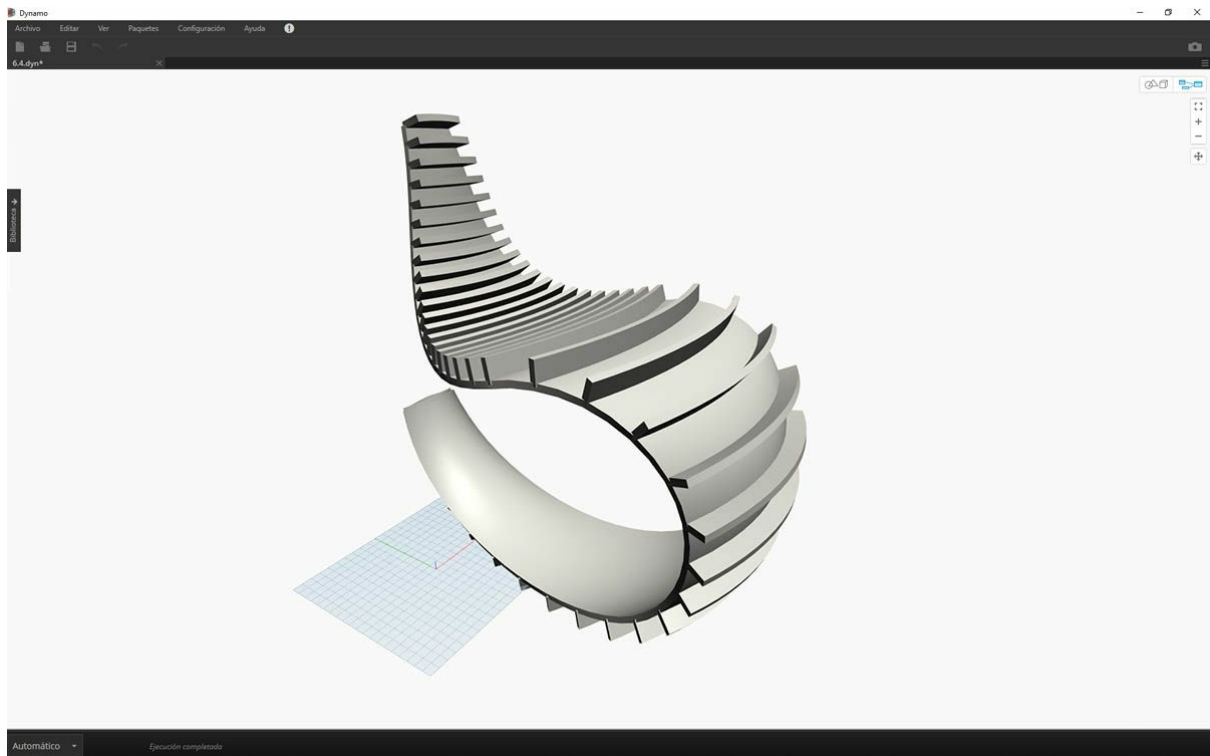
En el último paso, vamos a invertir la dirección de los miembros estriados. Como ya usamos la transposición en el ejercicio anterior, haremos algo similar aquí.

1. Tenemos un nivel más en la jerarquía, por lo que debemos utilizar *List.Map* con la función *List.Transpose* para cambiar la dirección de las curvas NURBS.



1. Es posible que queramos aumentar el número de huellas, para lo cual podemos cambiar el bloque de código a

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

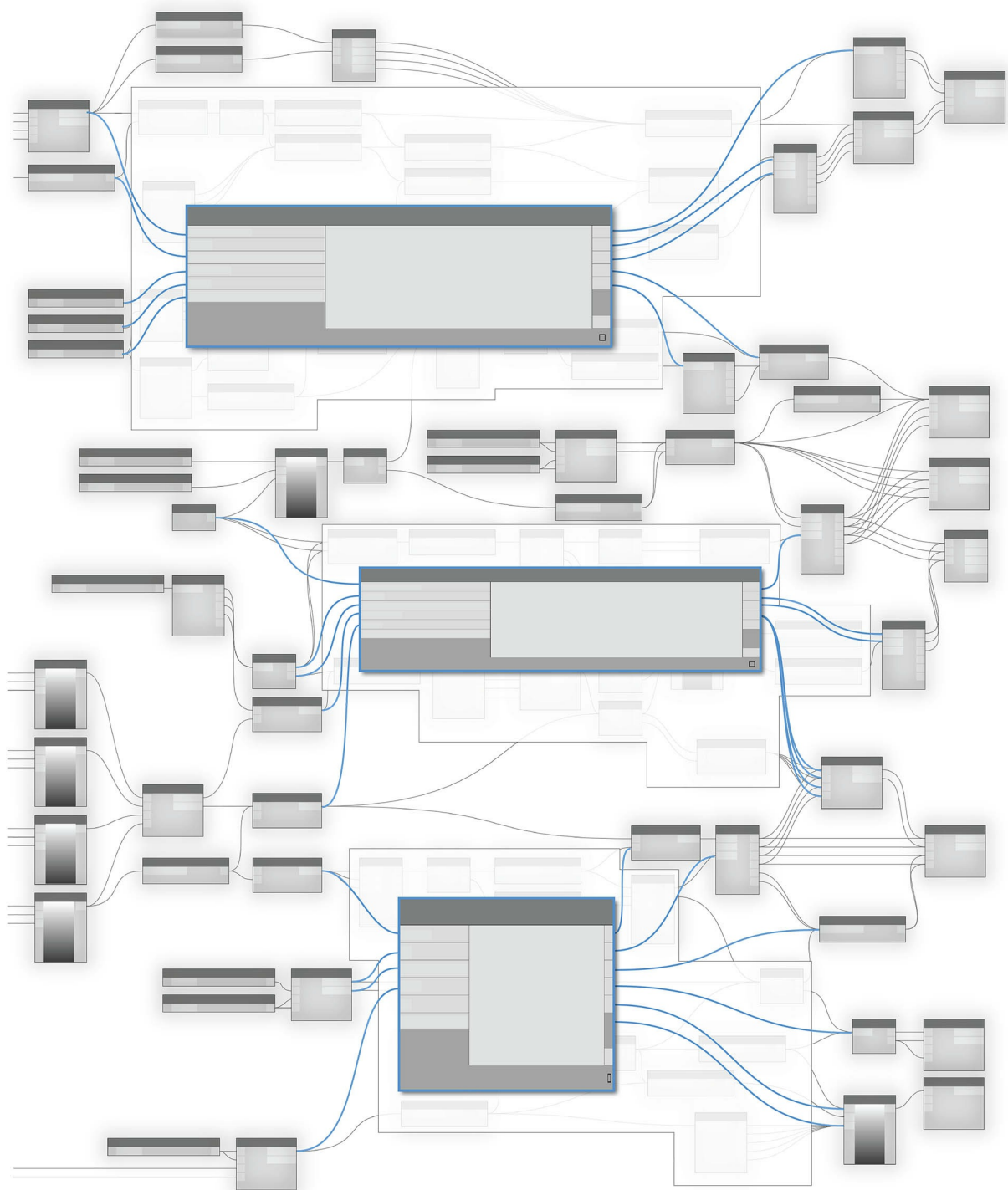


La primera versión de la mecedora era elegante, así que nuestro segundo modelo ofrece una versión todoterreno del asiento reclinable.

Bloques de código y DesignScript

Bloques de código y DesignScript

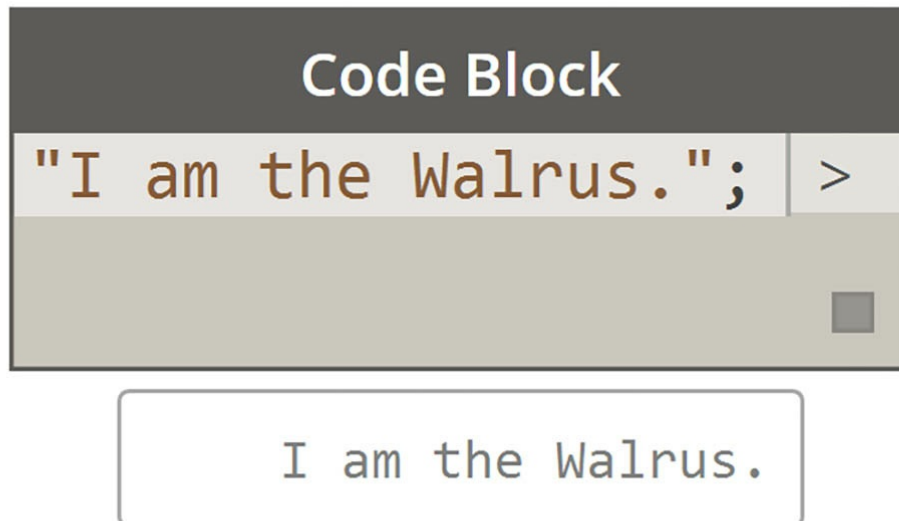
El bloque de código es una función exclusiva de Dynamo que vincula dinámicamente un entorno de programación visual con uno basado en texto. El bloque de código tiene acceso a todos los nodos de Dynamo y puede definir un gráfico completo en un nodo. Lea este capítulo detenidamente, ya que el bloque de código es un elemento fundamental de Dynamo.



¿Qué es un bloque de código?

¿Qué es un bloque de código?

Los bloques de código son una ventana a DesignScript, el lenguaje de programación que es la base de Dynamo. Creado desde cero para admitir flujos de trabajo de diseño exploratorios, DesignScript es un lenguaje legible y conciso que ofrece información inmediata a pequeños fragmentos de código y también se adapta a interacciones grandes y complejas. DesignScript también forma la columna vertebral del motor que impulsa la mayoría de los aspectos subyacentes de Dynamo. Como casi todas las funciones de los nodos y las interacciones de Dynamo tienen una relación de uno a uno con el lenguaje de creación de secuencias de comandos, se ofrecen oportunidades únicas de desplazarse entre secuencias de comandos e interacciones basadas en nodos de forma fluida.

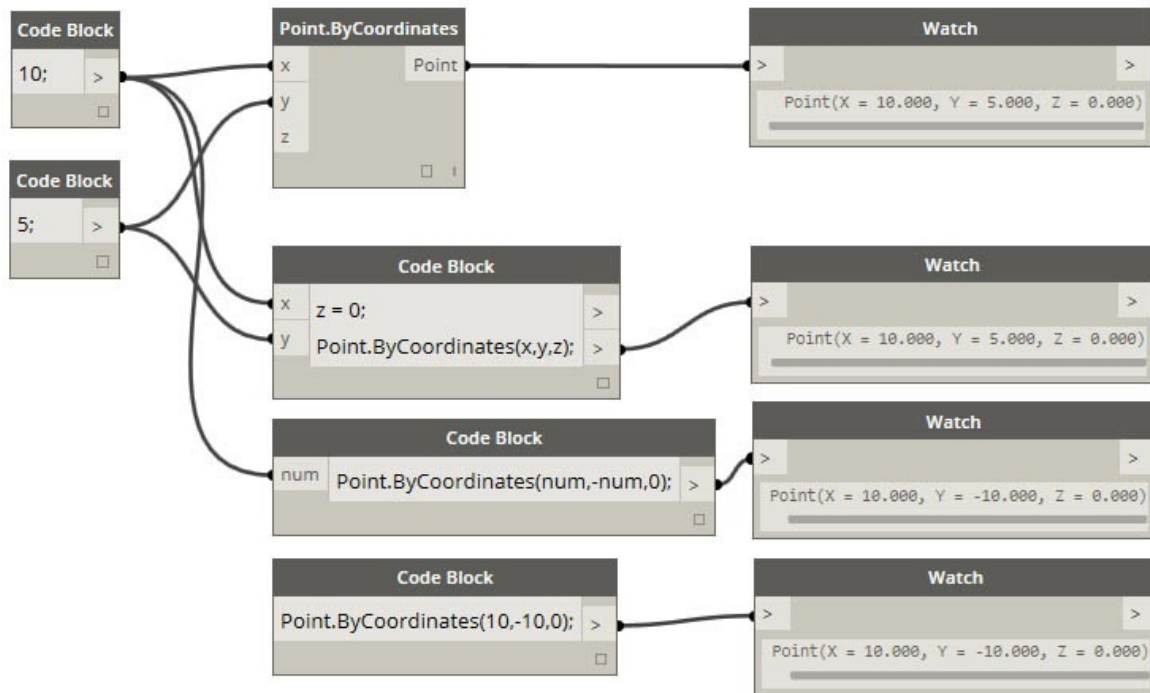


Para los principiantes, los nodos se pueden convertir automáticamente en sintaxis de texto para facilitar el aprendizaje de DesignScript o simplemente para reducir el tamaño de secciones de gráficos de mayor tamaño. Esto se realiza mediante un proceso denominado "de nodo a código", que se describe con más detalle en la [sección Sintaxis de DesignScript](#). Los usuarios con más experiencia pueden utilizar bloques de código para crear combinaciones personalizadas de funciones existentes y relaciones creadas por el usuario mediante una gran cantidad de paradigmas de codificación estándar. En un nivel intermedio entre los principiantes y los usuarios avanzados, hay disponibles un gran número de accesos directos y fragmentos de código que agilizarán los diseños. Si bien el término "bloque de código" puede intimidar un poco a los que no son programadores, es una herramienta eficaz y fácil de usar. Un principiante puede utilizar de forma eficaz el bloque de código con una mínima creación de código y un usuario avanzado puede establecer definiciones de secuencias de comandos que se pueden recuperar en cualquier parte de una definición de Dynamo.

Una breve descripción de los bloques de código

En resumen, los bloques de código son una interfaz de creación de secuencias de comandos de texto dentro de un entorno de creación de secuencias de comandos visuales. Se pueden utilizar como números, cadenas, fórmulas y otros tipos de datos. El bloque de código se ha diseñado para Dynamo, por lo que se pueden definir variables arbitrarias en el bloque de código y esas variables se añaden automáticamente a las entradas del nodo:

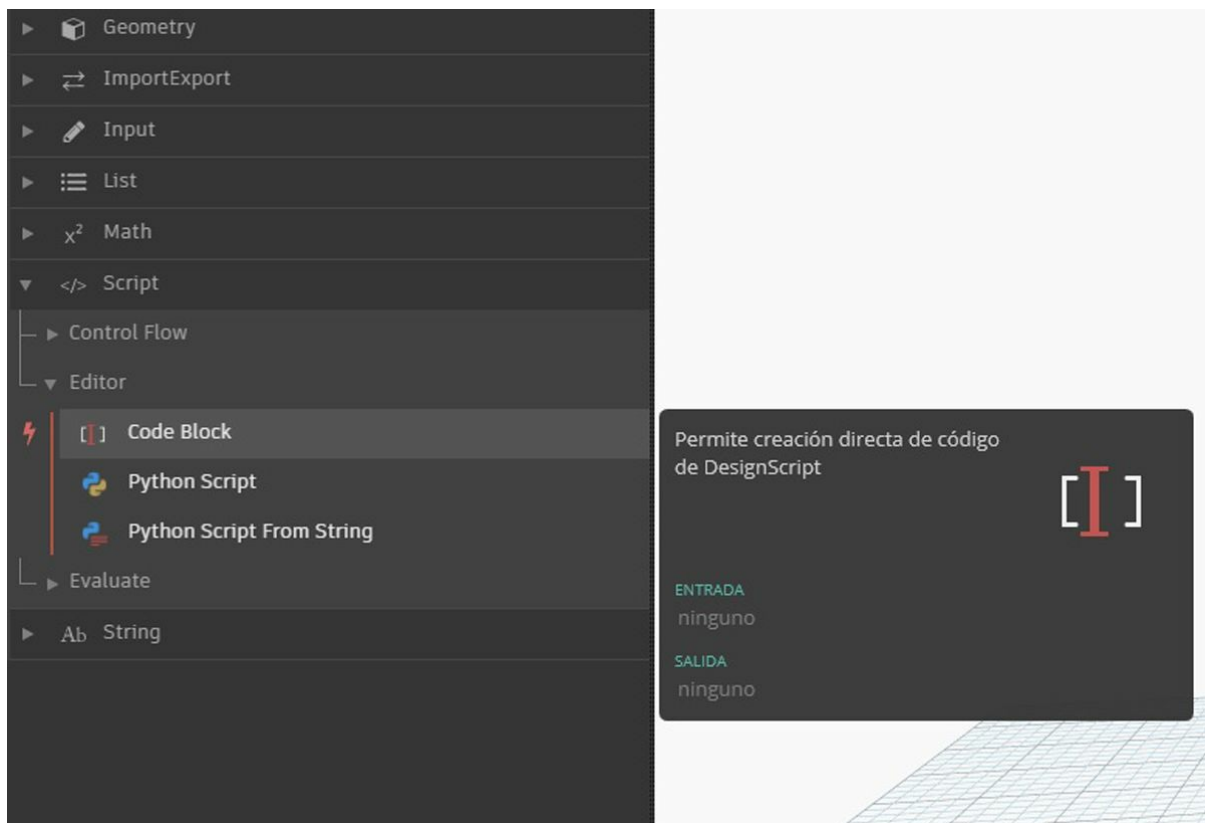
Con los bloques de código, un usuario dispone de flexibilidad para decidir cómo especificar entradas. A continuación, se indican varias formas diferentes de crear un punto básico con coordenadas (10, 5, 0):



A medida que aprende más sobre las funciones disponibles en la biblioteca, es posible que le sea más rápido escribir "Point.ByCoordinates" que buscar en la biblioteca y encontrar el nodo adecuado. Por ejemplo, al escribir "Point.", Dynamo mostrará una lista de posibles funciones que aplicar a un punto. Esto hace que la creación de secuencias de comandos sea más intuitiva y le ayudará a aprender a aplicar funciones en Dynamo.

Creación de nodos de bloque de código

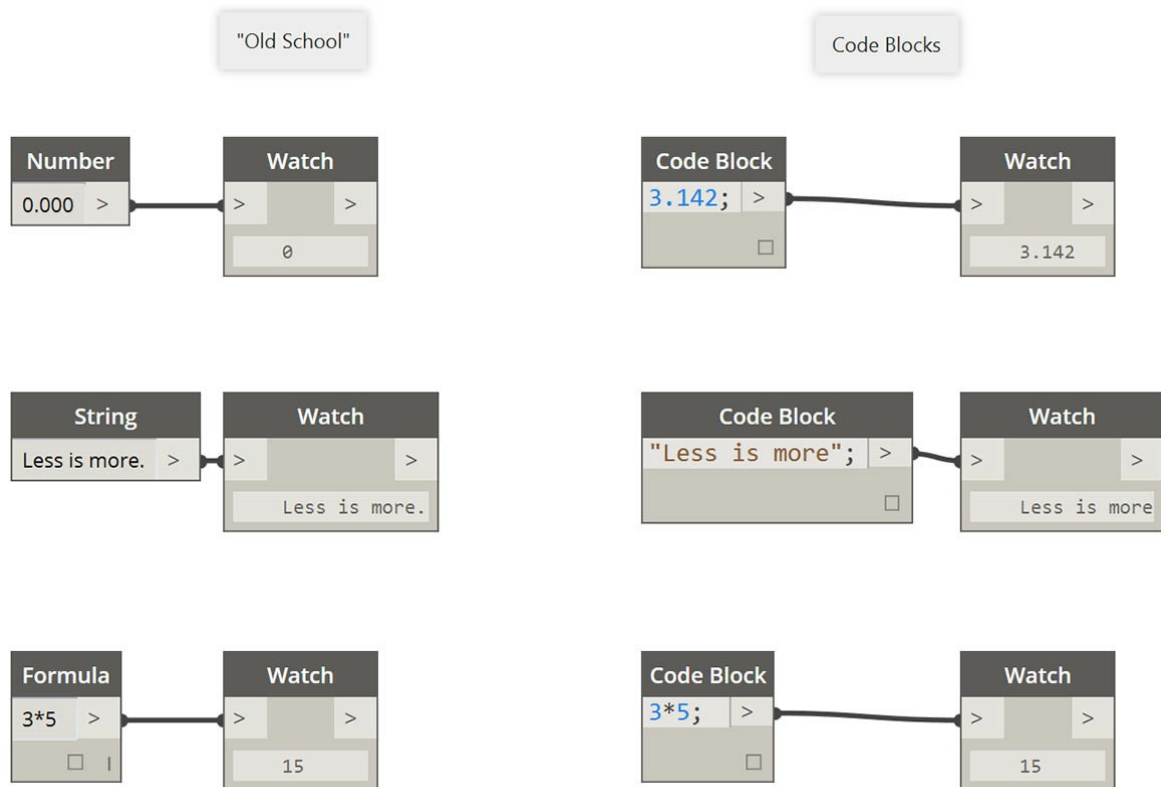
El bloque de código se encuentra en *Core > Entrada > Acciones > Bloque de código*. O, simplemente, de forma más rápida, haga doble clic en el lienzo para que aparezca el bloque de código. Este nodo se utiliza con tanta frecuencia que se le han proporcionado privilegios completos de doble clic.



Números, cadenas y fórmulas

Los bloques de código también son flexibles en relación con los tipos de datos. El usuario puede definir rápidamente números, cadenas y fórmulas, y el bloque de código generará la salida deseada.

En la imagen siguiente, puede comprobar que la forma tradicional de trabajo es algo prolíja: el usuario busca el nodo previsto en la interfaz, lo añade al lienzo y, a continuación, introduce los datos. Con el bloque de código, el usuario puede hacer doble clic en el lienzo para desplegar el nodo y escribir el tipo de datos correcto con la sintaxis básica.



Los nodos *number*, *string* y *formula* son tres ejemplos de nodos de Dynamo que posiblemente estén obsoletos en comparación con el *bloque de código*.

Sintaxis de DesignScript

Sintaxis de DesignScript

Es posible que haya observado que en los nombres de los nodos de Dynamo se da un elemento común: cada nodo utiliza una sintaxis "." sin espacios. Esto se debe a que el texto de la parte inicial de cada nodo representa la sintaxis real para las secuencias de comandos y el "." (o *notación de puntos*) separa un elemento de los métodos a los que es posible invocar. Esto permite traducir fácilmente las secuencias de comandos visuales a secuencias de comandos basadas en texto.



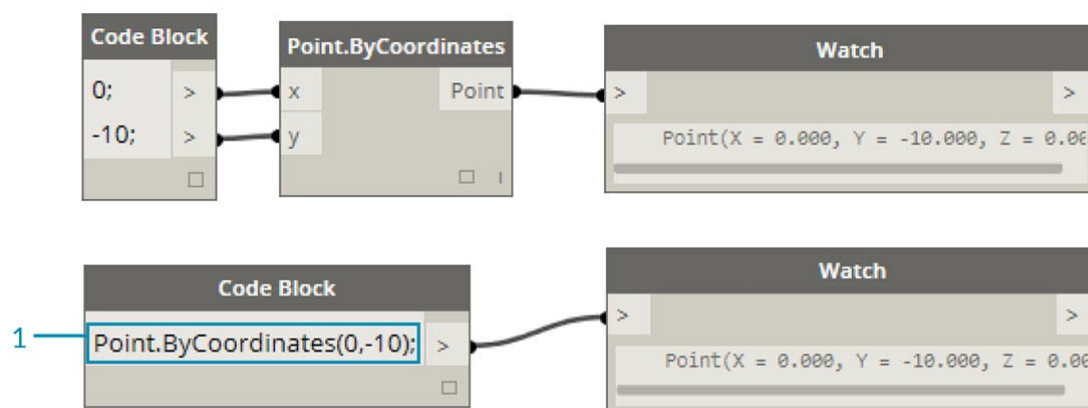
Como analogía general para la notación de puntos, ¿cómo podemos tratar una manzana paramétrica en Dynamo? A continuación se indican algunos métodos que ejecutaremos en la manzana antes de decidir consumirla. (Nota: estos no son métodos reales de Dynamo):

Legible para el ser humano	Notación de puntos	Salida
¿De qué color es la manzana?	Manzana.color	rojo
¿Está madura la manzana?	Manzana.madura	verdadero
¿Cuánto pesa la manzana?	Manzana.peso	170 g
¿De dónde viene la manzana?	Manzana.padre	árbol
¿Qué crea la manzana?	Manzana.hijo	semillas
¿Esta manzana se produjo de forma local?	Manzana.distanciaDeHuerta	96 km

A juzgar por las salidas de la tabla de arriba, esta parece una manzana sabrosa. Creo que voy a *Manzana.comer*.

Notación de puntos en el bloque de código

Con la analogía de manzana en mente, fijémonos en *Point.ByCoordinates* y veamos cómo podemos crear un punto mediante el bloque de código:



La sintaxis del *bloque de código* `Point.ByCoordinates(0,10);` proporciona el mismo resultado que el nodo *Point.ByCoordinates* de Dynamo, excepto que permite crear un punto mediante un nodo. Esto es más eficaz que conectar un nodo independiente a "X" e "Y".

1. Al utilizar *Point.ByCoordinates* en el bloque de código, se especifican las entradas en el mismo orden que en el nodo (X,Y) predefinido.

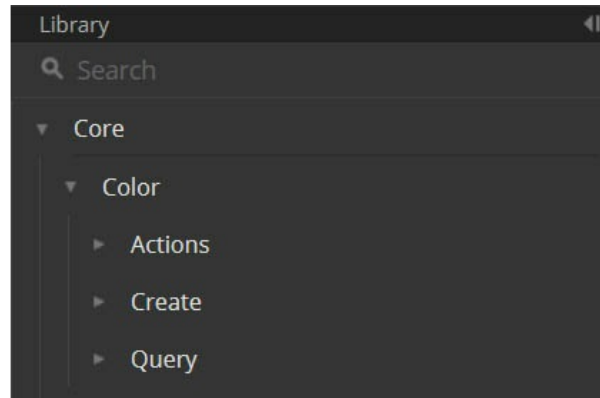
Invocación de nodos

Puede invocar cualquier nodo normal de la biblioteca a través de un bloque de código siempre que el nodo no sea un nodo de "IU": los que tienen una función de interfaz de usuario especial. Por ejemplo, puede invocar a *Circle.ByCenterPointRadius*, pero no tendría mucho sentido invocar a un nodo *Watch 3D*.

Los nodos normales (la mayoría de los incluidos en la biblioteca) suelen ser de tres tipos:

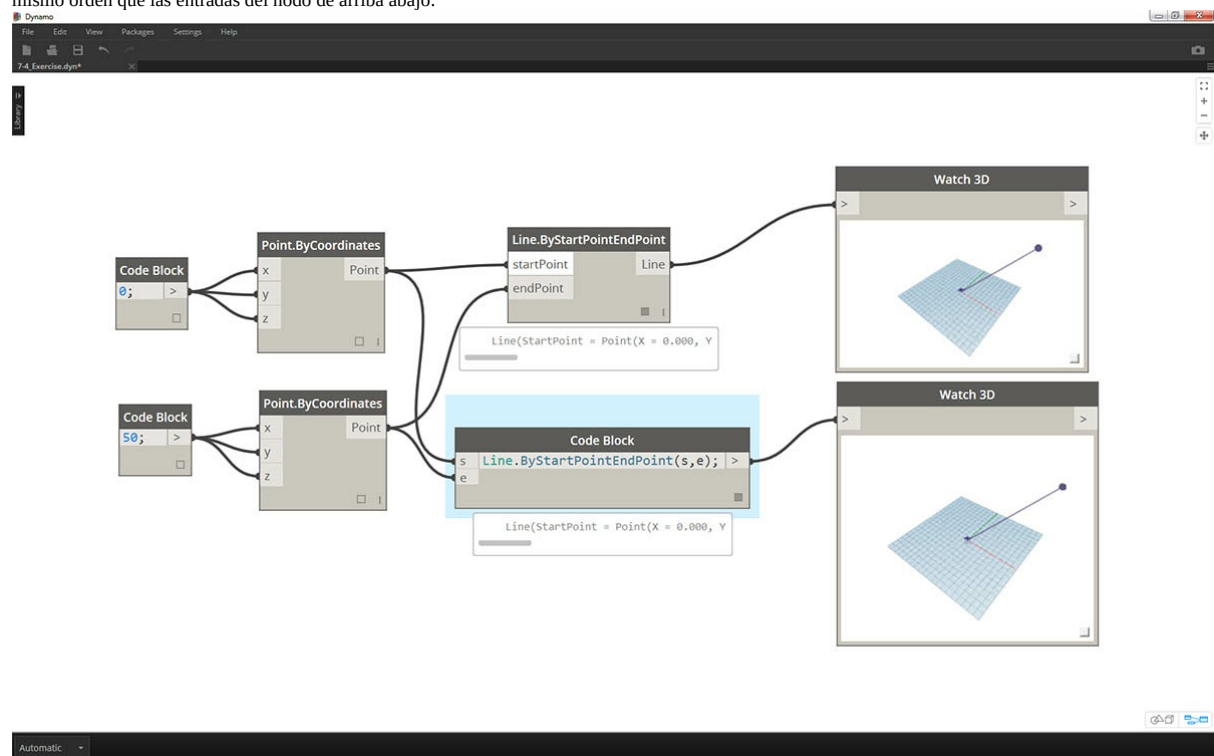
- **Crear:** crea (o construye) algo.
- **Acción:** realiza una acción sobre algún elemento.
- **Consulta:** obtiene una propiedad de algún elemento que ya existe.

Descubrirá que la biblioteca se organiza teniendo en cuenta estas categorías. Los métodos o nodos de estos tres tipos se tratan de forma diferente cuando se invocan desde un bloque de código.



Crear

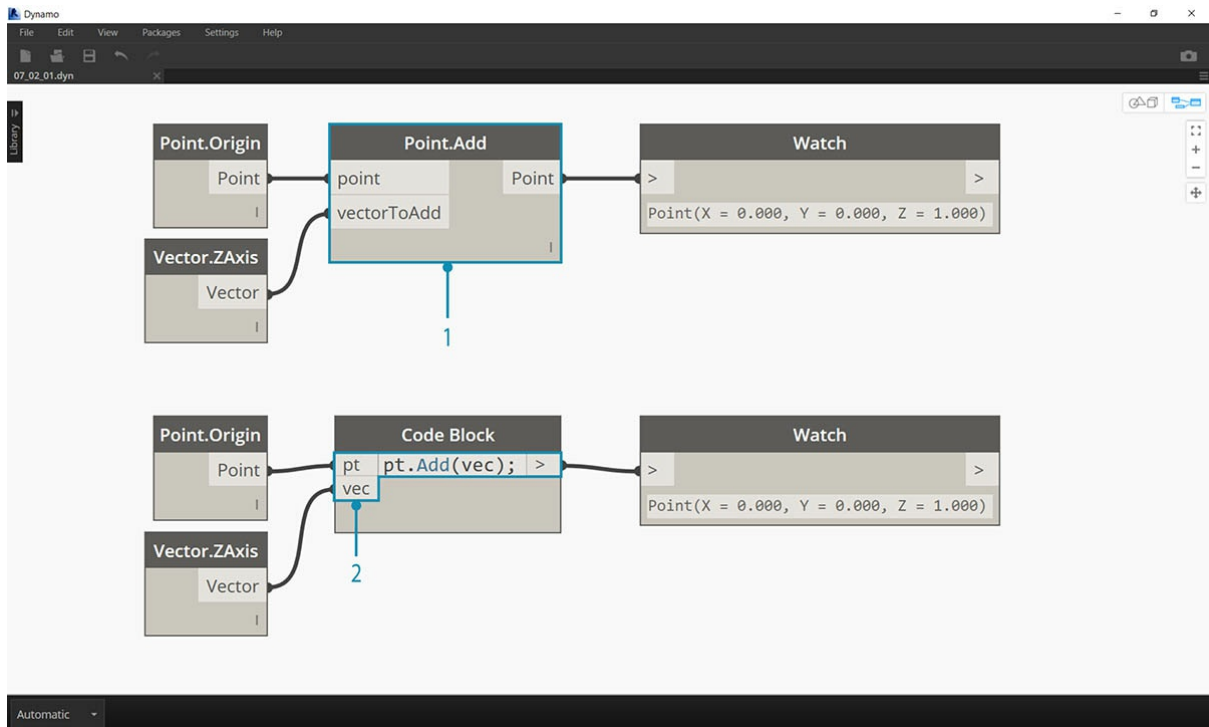
La categoría "Crear" genera geometría desde cero. En el bloque de código, los valores se introducen de izquierda a derecha. Estas entradas están en el mismo orden que las entradas del nodo de arriba abajo:



Si comparamos el nodo *Line.ByStartPointEndPoint* y la sintaxis correspondiente en el bloque de código, vemos que se obtienen los mismos resultados.

Acción

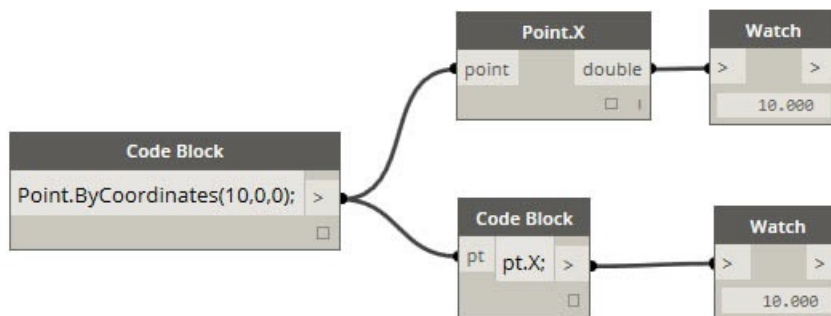
Una acción es algo que se hace a un objeto de un tipo concreto. Dynamo utiliza la *notación de puntos*, común a muchos idiomas de codificación, para aplicar una acción a un elemento. Una vez que tenga el elemento, escriba un punto y, a continuación, el nombre de la acción. La entrada del método de acción se coloca entre paréntesis, del mismo modo que con los métodos de creación, solo que no es necesario especificar la primera entrada que se ve en el nodo correspondiente. En su lugar, especificamos el elemento en el que realizamos la acción:



1. El nodo *Point.Add* es un nodo de acción, por lo que la sintaxis funciona de forma algo distinta.
2. Las entradas son (1) el punto y (2) el vector que añadir. En un *bloque de código*, hemos designado al punto (el elemento) "pt". Para añadir un vector denominado "vec" a "pt", escribimos *pt.Add(vec)* o: elemento, punto, acción. La acción Add (añadir) solo tiene una entrada o todas las entradas del nodo *Point.Add* menos la primera. La primera entrada del nodo *Point.Add* es el punto en sí.

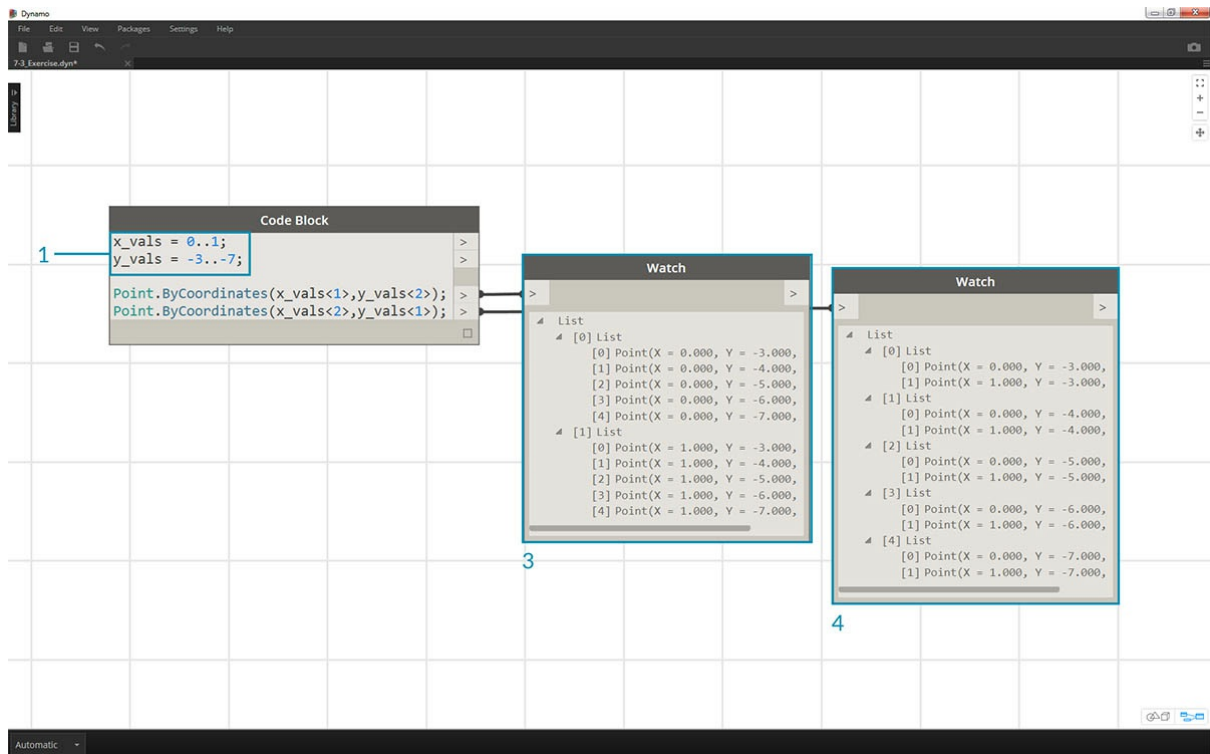
Consulta

Los métodos de consulta obtienen una propiedad de un objeto. Dado que el objeto en sí es la entrada, no es necesario especificar ninguna entrada. No se requieren paréntesis.



Acerca del encaje

El encaje con nodos es algo diferente al encaje con el bloque de código. Con los nodos, el usuario hace clic con el botón derecho en el nodo y selecciona la opción de encaje que se va a utilizar. Con el bloque de código, el usuario tiene mucho más control sobre la estructura de los datos. El método de abreviatura del bloque de código utiliza *guías de replicación* para definir cómo se deben emparejar varias listas unidimensionales. Los números entre corchetes angulares "<>" definen la jerarquía de la lista anidada resultante: <1>, <2>, <3>, etc.

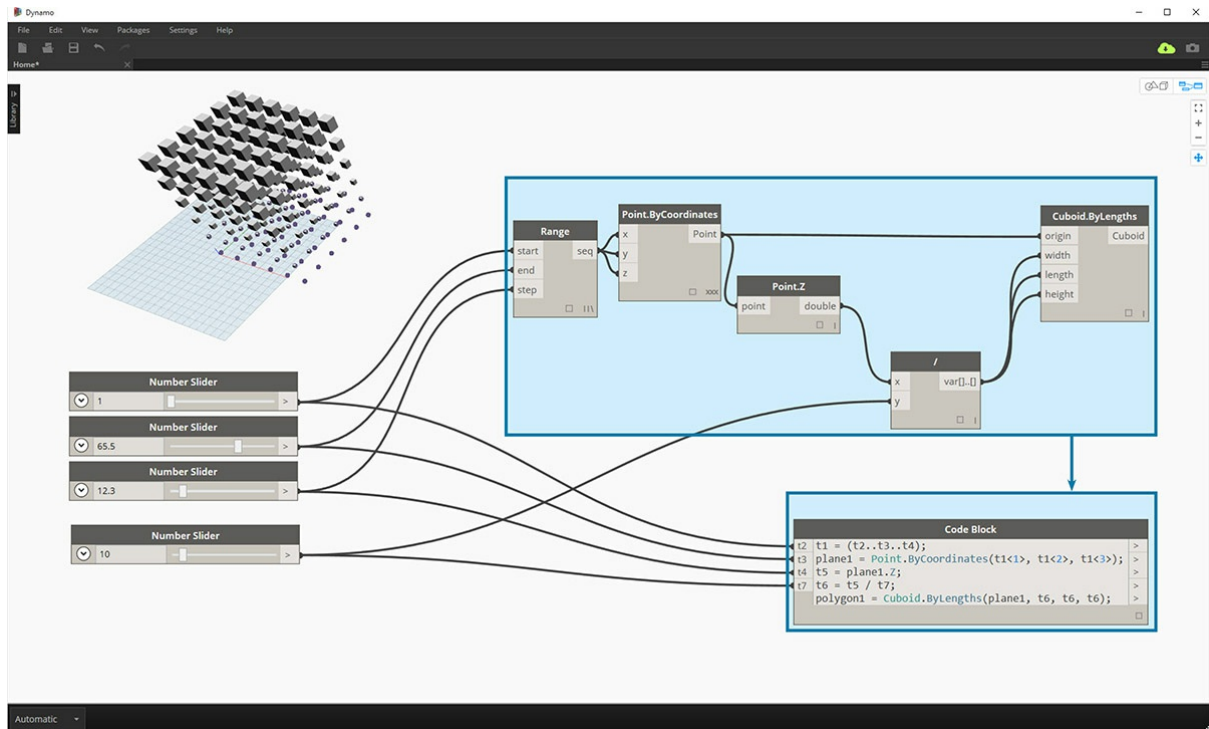


1. En este ejemplo, se utiliza un método abreviado para definir dos rangos (la siguiente sección de este capítulo contiene más información sobre el método abreviado). En síntesis, `0..1`; es equivalente a `{0, 1}` y `-3..-7` es equivalente a `{-3, -4, -5, -6, -7}`. El resultado nos proporciona listas de 2 valores X y 5 valores Y. Si no utilizamos guías de replicación con estas listas no coincidentes, obtenemos una lista de dos puntos, que es la longitud de la lista más corta. Con las guías de replicación, podemos encontrar todas las combinaciones posibles de 2 y 5 coordenadas (o un **Producto cartesiano**).
2. Utilizando la sintaxis `Point.ByCoordinates(x_vals<1>,y_vals<2>)`; se obtienen **dos** listas con **cinco** elementos en cada lista.
3. Utilizando la sintaxis `Point.ByCoordinates(x_vals<2>,y_vals<1>)`; se obtienen **cinco** listas con **dos** elementos en cada lista.

Con esta notación, también podemos especificar cuál será la lista dominante: 2 listas de 5 elementos o 5 listas de 2 elementos. En el ejemplo, al cambiar el orden de las guías de replicación, el resultado es una lista de filas de puntos o una lista de columnas de puntos en una rejilla.

Nodo para código

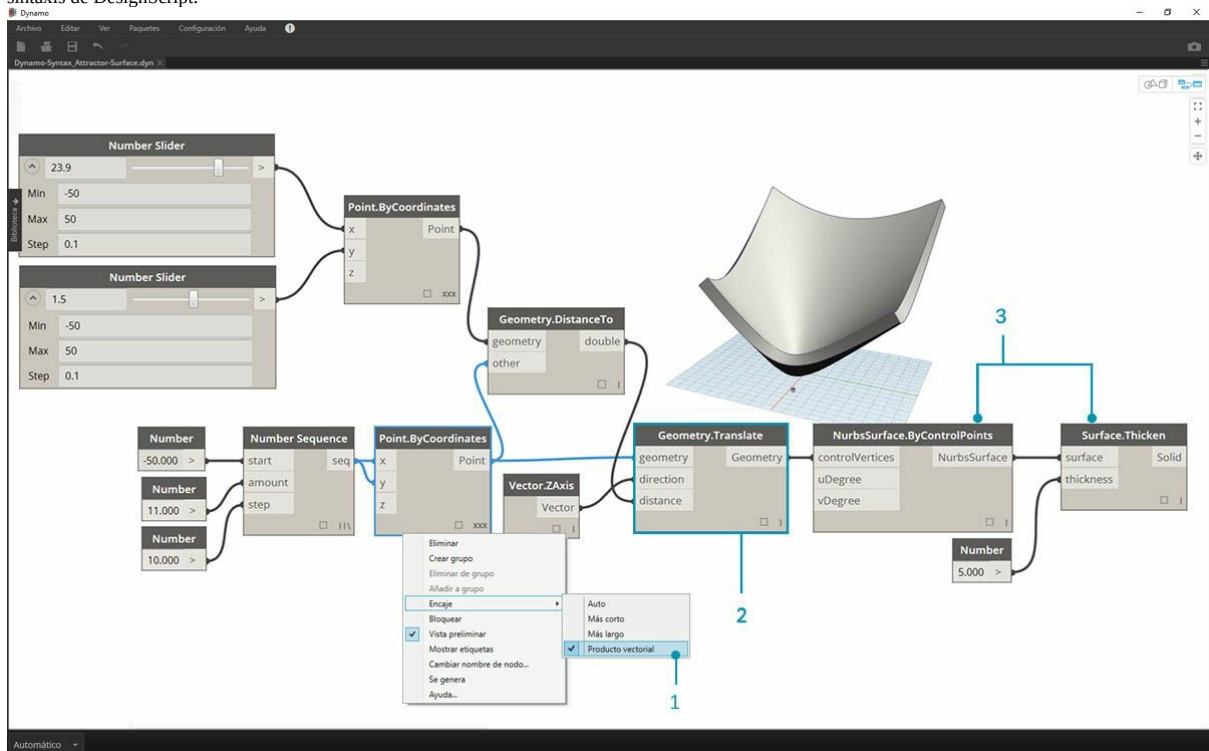
Aunque los métodos de bloque de código anteriores pueden requerir cierto tiempo para acostumbrarse, en Dynamo existe una función denominada "nodo para código" que facilita el proceso. Para utilizar esta función, seleccione una matriz de nodos en el gráfico de Dynamo, haga clic con el botón derecho en el lienzo y seleccione "Nodo para código". Dynamo comprime estos nodos en un bloque de código con todas las entradas y salidas. No solo es una excelente herramienta para aprender a utilizar bloques de código, sino que también permite trabajar con un gráfico Dynamo más eficaz y paramétrico. Concluiremos el siguiente ejercicio utilizando "Nodo para código", así que no se lo pierda.



Ejercicio:

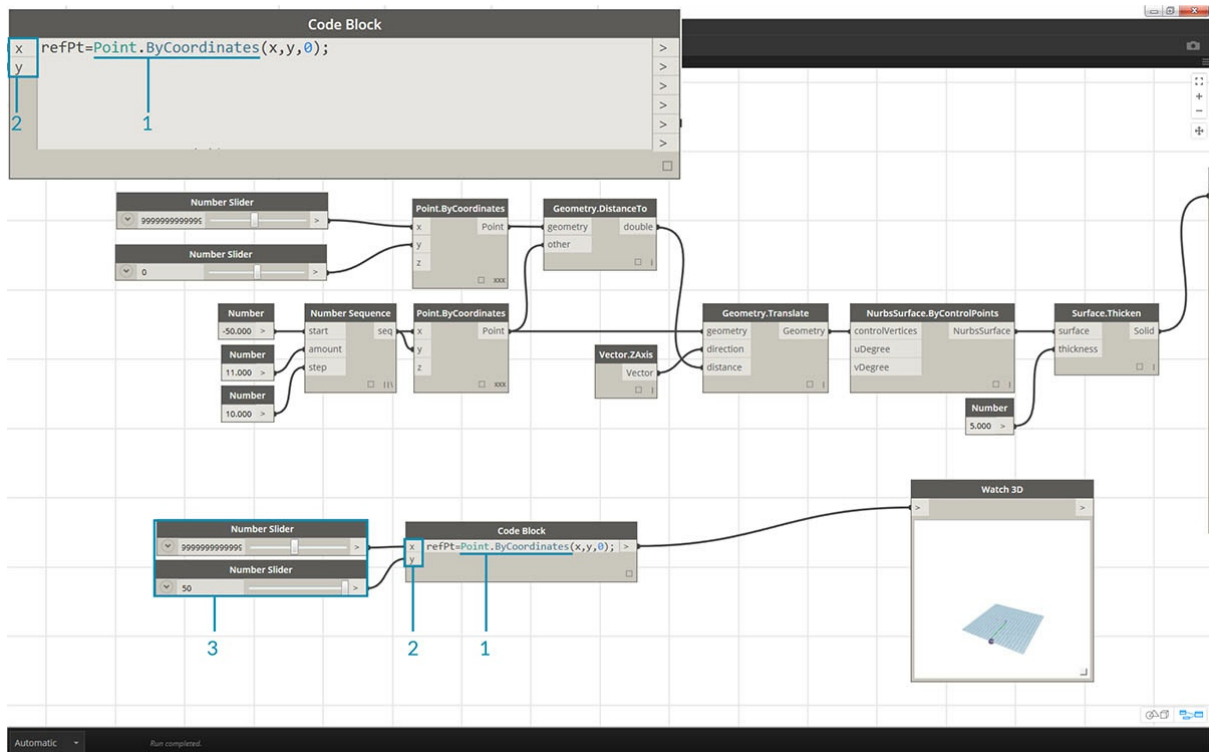
Descargue el archivo de ejemplo que acompaña a este ejercicio (haga clic con el botón derecho y seleccione "Guardar vínculo como..."). En el Apéndice se incluye una lista completa de los archivos de ejemplo. [Dynamo-Syntax Attractor-Surface.dyn](#)

Para mostrar la eficacia del bloque de código, vamos a convertir una definición de campo atractor existente al lenguaje del bloque de código. Al trabajar con una definición existente, se muestra cómo se relaciona el bloque de código con las secuencias de comandos visuales y es útil para aprender a utilizar la sintaxis de DesignScript.

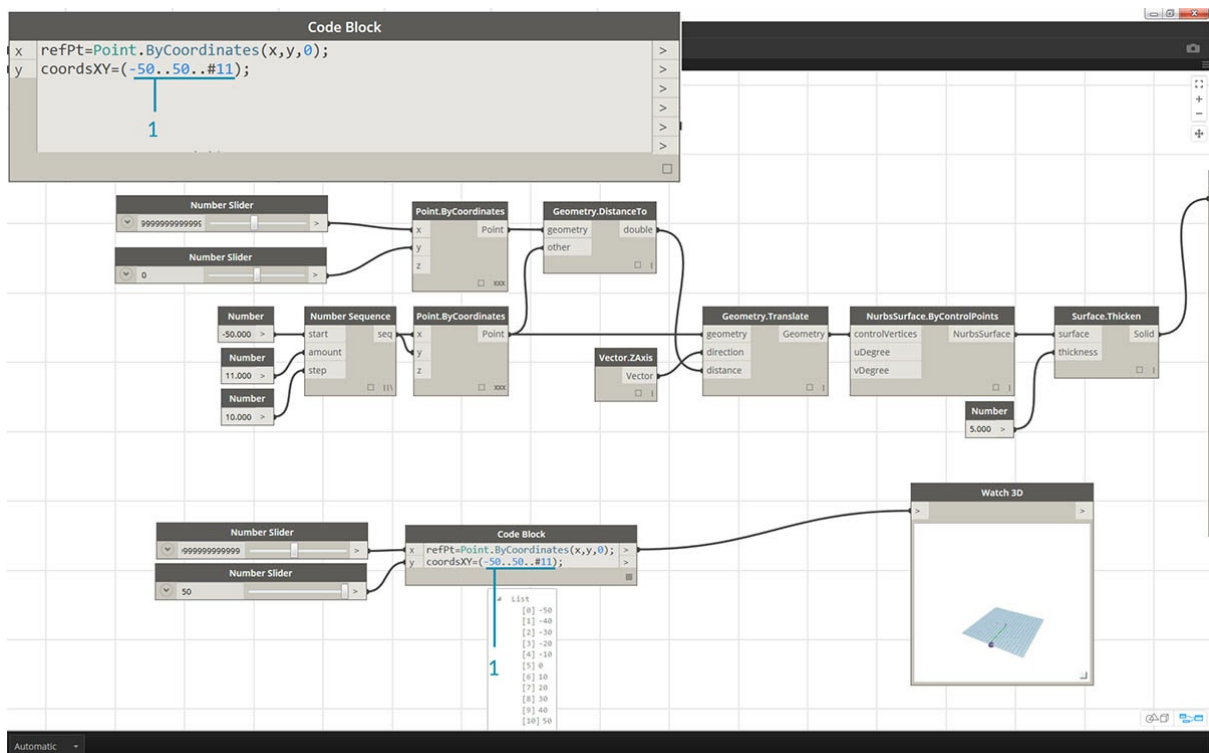


Comience volviendo a crear la definición de la imagen anterior (o abriendo el archivo de muestra).

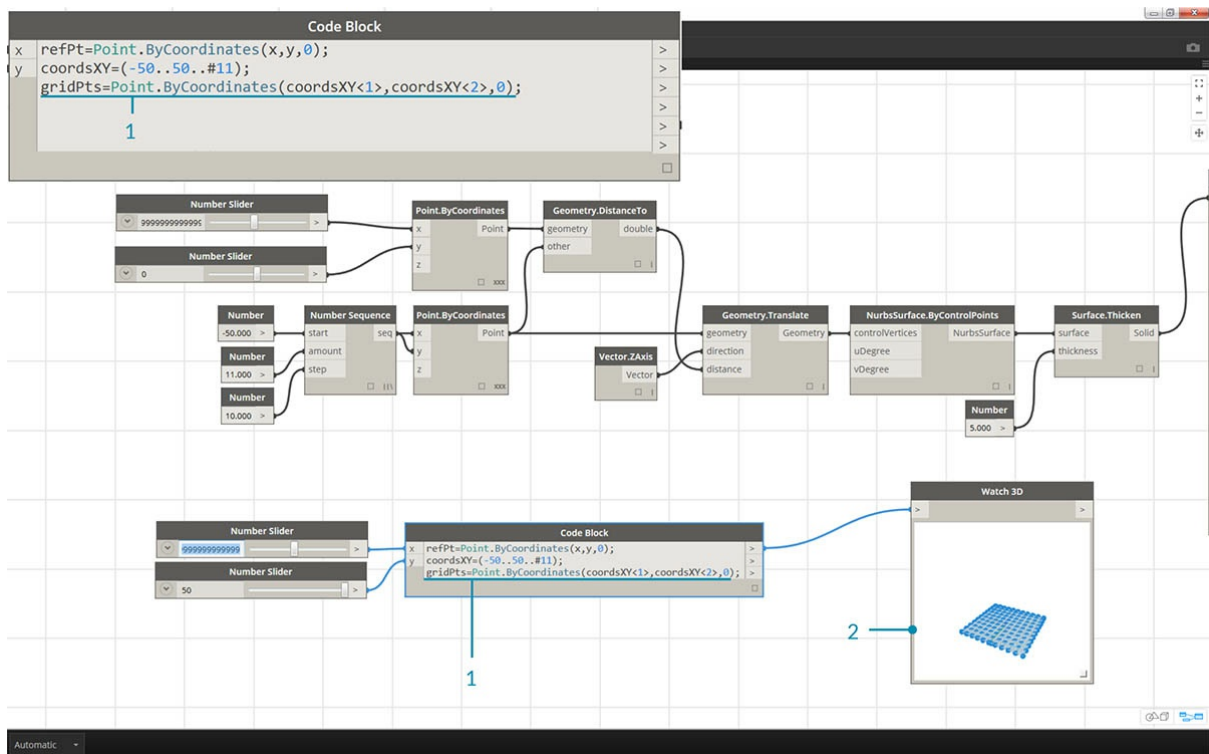
1. Observe que el encaje en *Point.ByCoordinates* se ha establecido en *Producto cartesiano*.
2. Cada punto de una rejilla se desplaza hacia arriba en la dirección Z en función de su distancia con respecto al punto de referencia.
3. Se vuelve a crear una superficie y se engrosa, lo cual genera una curvatura en la geometría relativa a la distancia con respecto al punto de referencia.



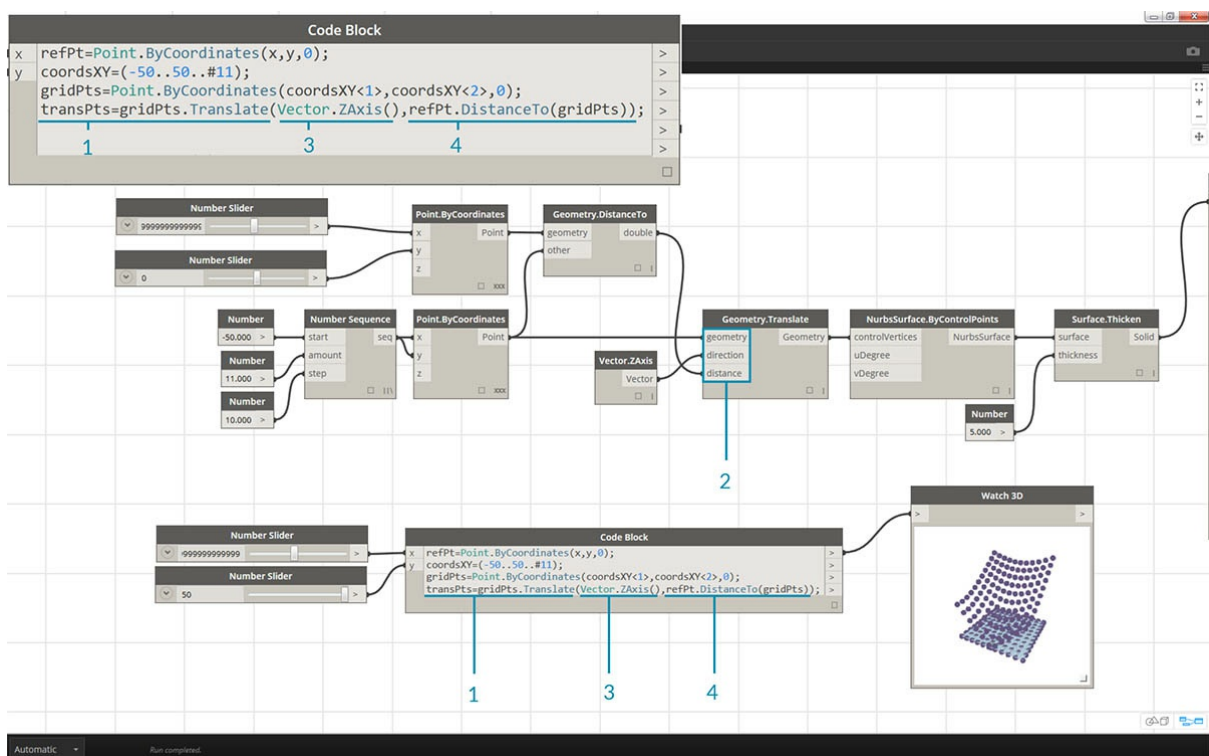
1. Para comenzar por el principio, definimos primero el punto de referencia: `Point.ByCoordinates(x,y,0);`. Utilizamos la misma sintaxis `Point.ByCoordinates` que se especifica en la parte inicial del nodo de punto de referencia.
2. Las variables `x` e `y` se insertan en el bloque de código para que podamos actualizarlas dinámicamente con los controles deslizantes.
3. Añada algunos *controles deslizantes* a las entradas del *bloque de código* que vayan del -50 al 50. De este modo, podemos abarcar la rejilla por defecto de Dynamo.



1. En la segunda línea del *bloque de código*, definimos un método abreviado para reemplazar el nodo de secuencia de números: `coordsXY = (-50..50..#11);`. Trataremos este tema en mayor profundidad en la siguiente sección. Por ahora, observe que este método abreviado es equivalente al nodo *Number Sequence* de la secuencia de comandos visual.

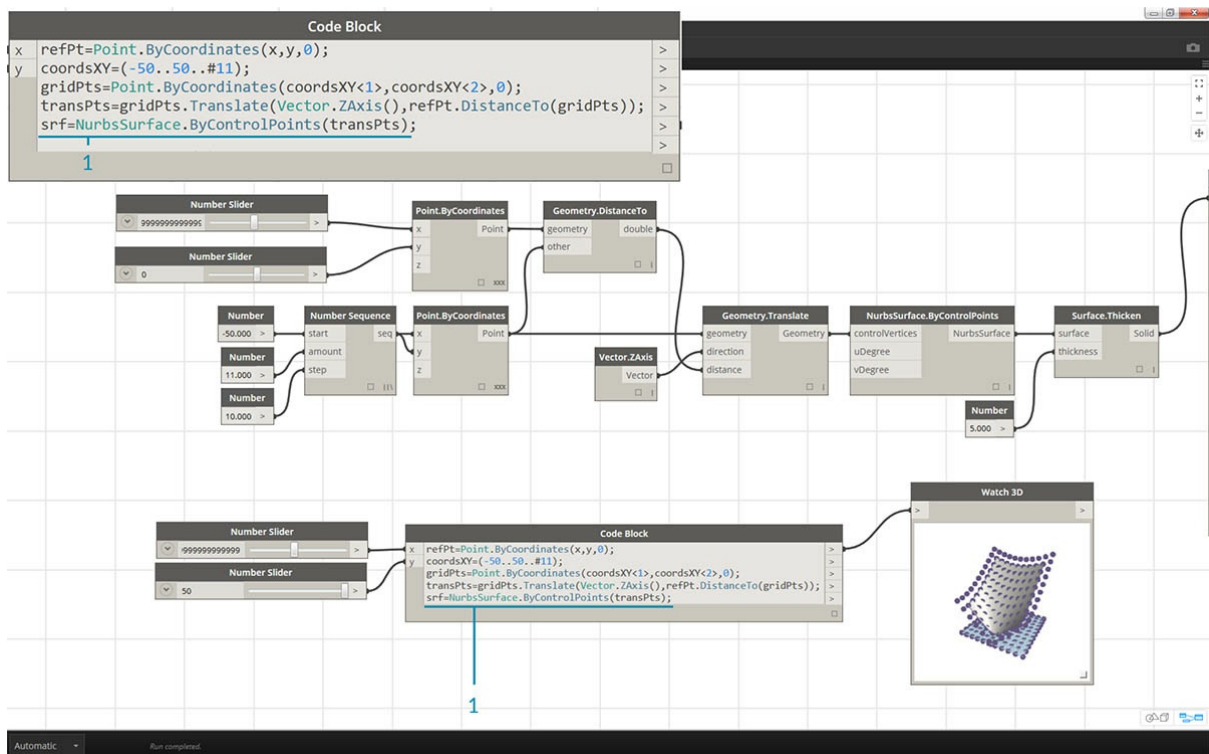


1. Ahora vamos a crear una rejilla de puntos a partir de la secuencia *coordsXY*. Para ello, vamos a utilizar la sintaxis *Point.ByCoordinates*, pero también debemos iniciar un *Producto cartesiano* de la lista de la misma forma que lo hicimos en la secuencia de comandos visual. Para hacerlo, se escribe la línea: `gridPts = Point.ByCoordinates(coordsXY<1>, coordsXY<2>, 0);`. Los corchetes angulares indican la referencia del producto cartesiano.
2. Observe en el nodo *Watch3D* que tenemos una rejilla de puntos en la rejilla de Dynamo.

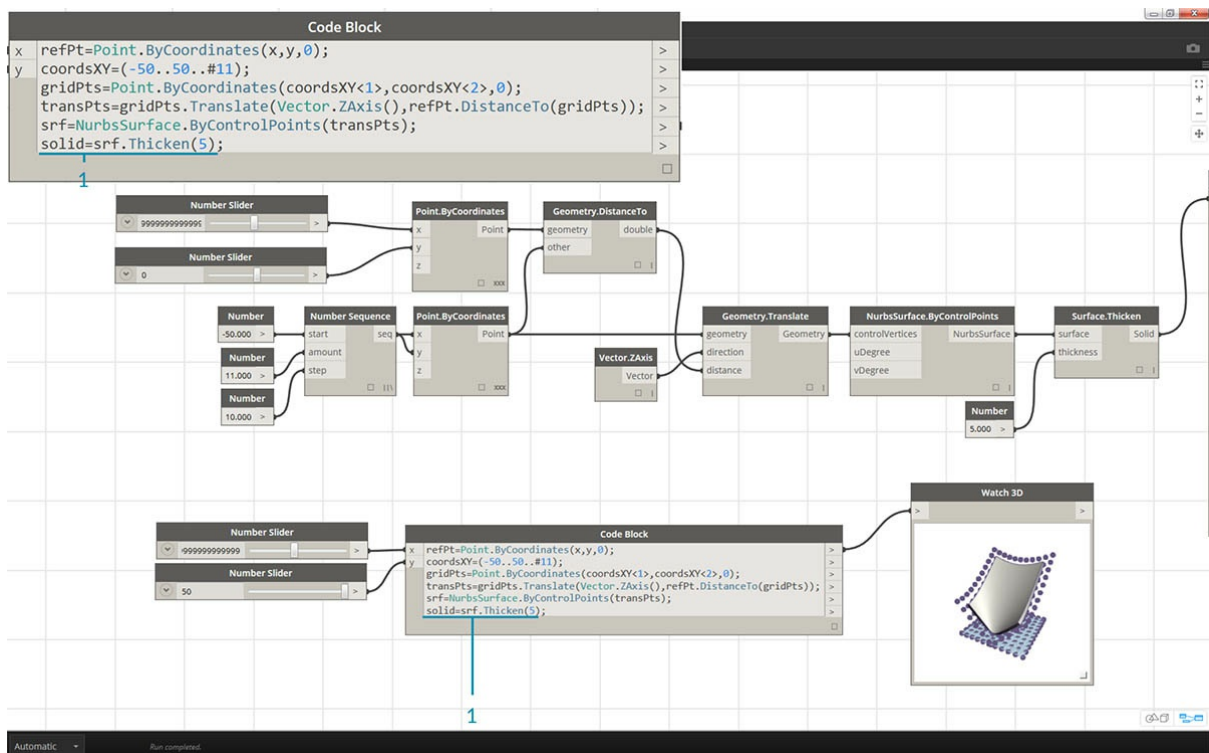


1. Ahora llega la parte compleja: mover la rejilla de puntos hacia arriba según su distancia con respecto al punto de referencia. Primero, llamaremos a este nuevo conjunto de puntos *transPts*. Como la traslación es una acción que se lleva a cabo en un elemento existente, en lugar de utilizar *Geometry.Translate...*, utilizamos *gridPts.Translate*.
2. Al leer el nodo real en el lienzo, vemos que hay tres entradas. La geometría que se va a trasladar ya se ha definido porque llevamos a cabo la acción en ese elemento (con *gridPts.Translate*). Las dos entradas restantes se insertarán en los paréntesis de las funciones *direction* y *distance*.
3. Definir la dirección es sencillo; utilizamos *Vector.ZAxis()* para realizar un movimiento vertical.
4. La distancia entre el punto de referencia y cada punto de rejilla aún debe calcularse. Por lo tanto, lo hacemos como una acción al punto de referencia del mismo modo: `refPt.DistanceTo(gridPts)`.

5. La línea final del código nos proporciona los puntos trasladados: `transPts = gridPts.Translate(Vector.ZAxis(),refPt.DistanceTo(gridPts));`.



1. Ahora tenemos una rejilla de puntos con la estructura de datos adecuada para crear una superficie NURBS. Construimos la superficie mediante `srf = NurbsSurface.ByControlPoints(transPts);`.

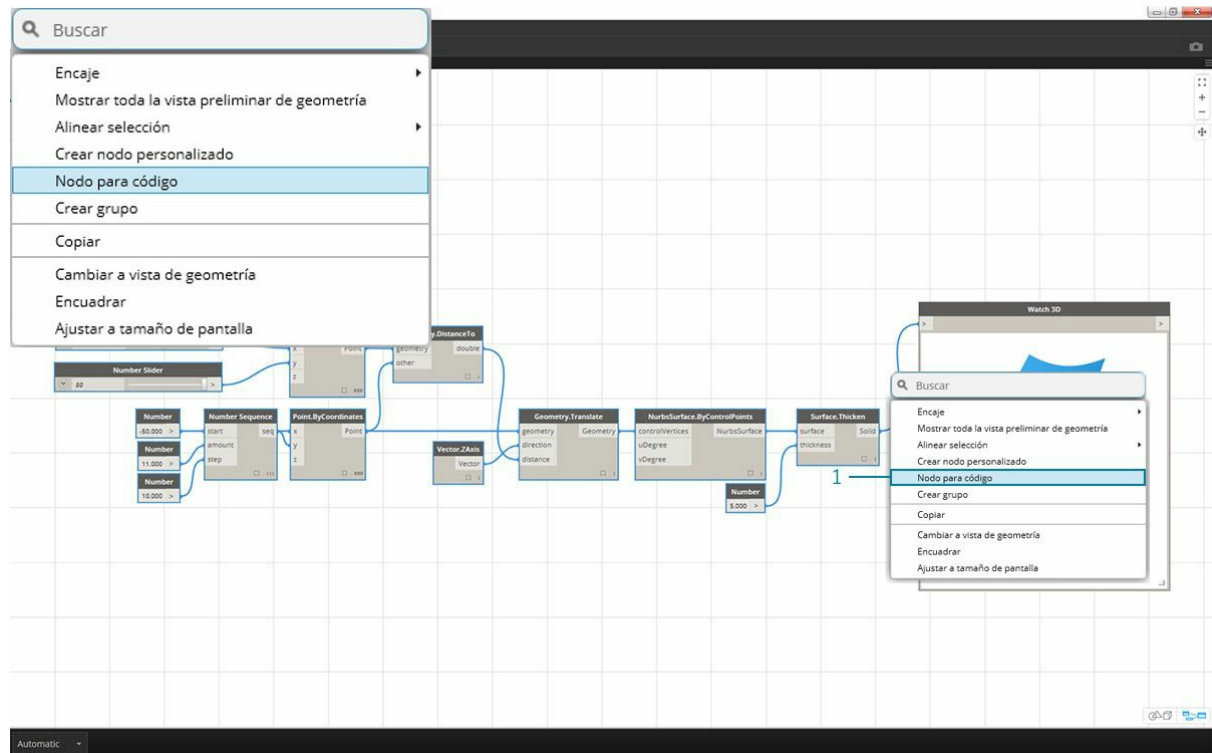


1. Por último, para añadir cierta profundidad a la superficie, construimos un sólido mediante `solid = srf.Thicken(5);`. En este caso, hemos engrosado la superficie 5 unidades en el código, pero siempre podríamos definirla como una variable —denominándola *thickness* (grosor), por ejemplo— y después controlar ese valor con un control deslizante.

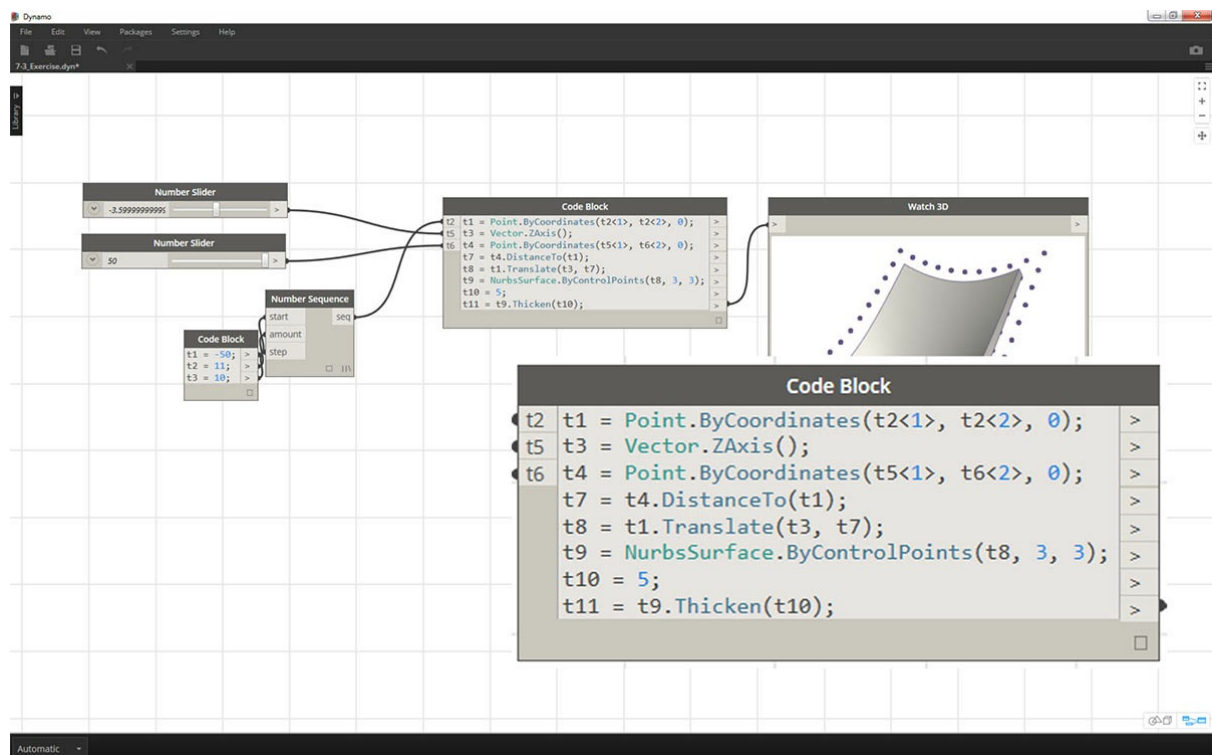
Simplificar el gráfico con "Nodo para código"

La función "nodo para código" automatiza todo el ejercicio que acabamos de completar con un solo clic. No solo es una herramienta eficaz para crear definiciones personalizadas y bloques de código reutilizables, sino que también es una herramienta muy útil para aprender a usar las secuencias de

comandos de Dynamo:



1. Empezar con la secuencia de comandos visual existente del paso 1 del ejercicio. Seleccione todos los nodos, haga clic con el botón derecho en el lienzo y seleccione "Nodo para código". Tan simple como eso.

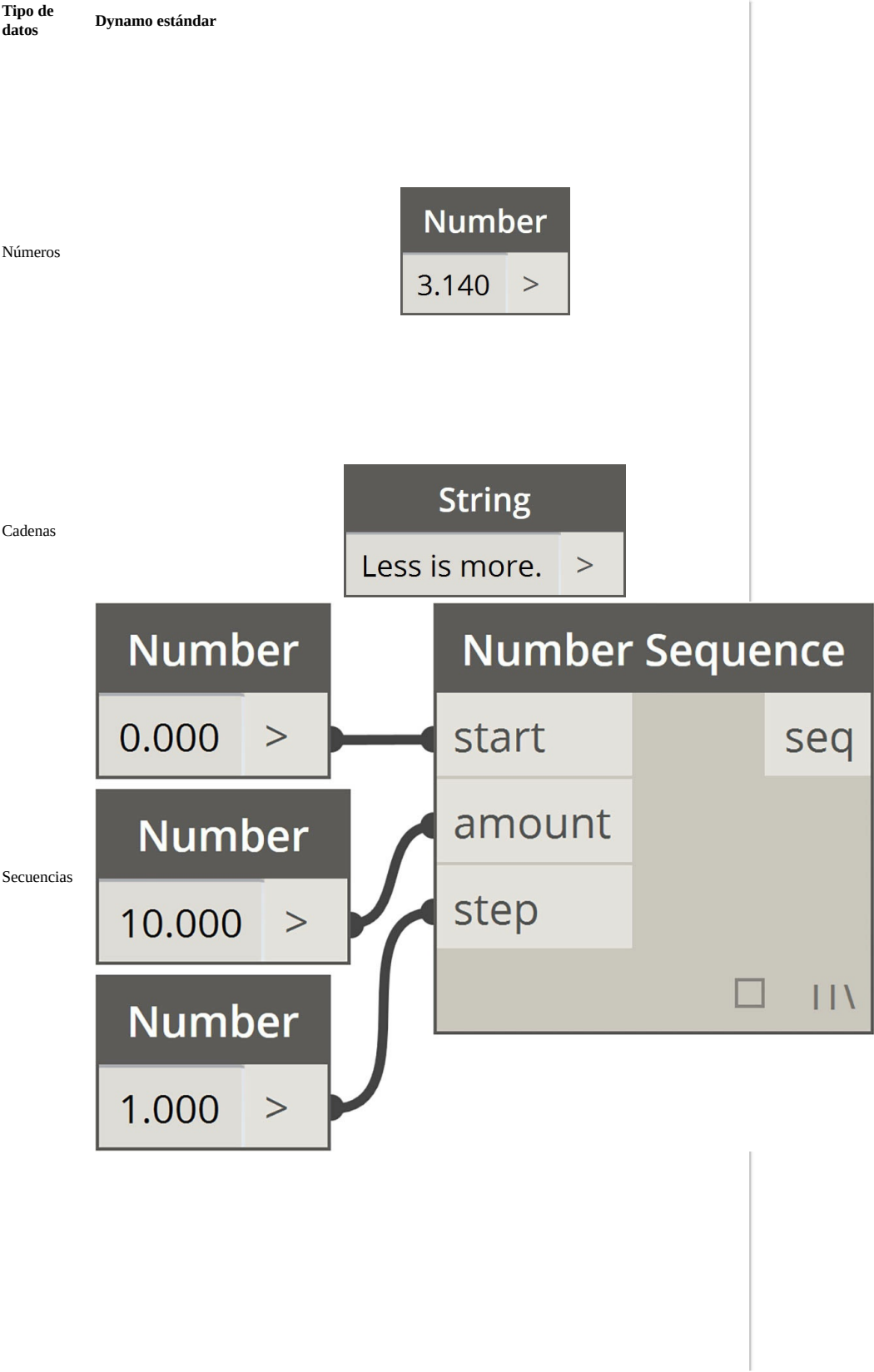


Dynamo crea de forma automática una versión basada en texto del gráfico visual, con el encaje incluido. Pruebe esta función con las secuencias de comandos visuales y aproveche las posibilidades del bloque de código.

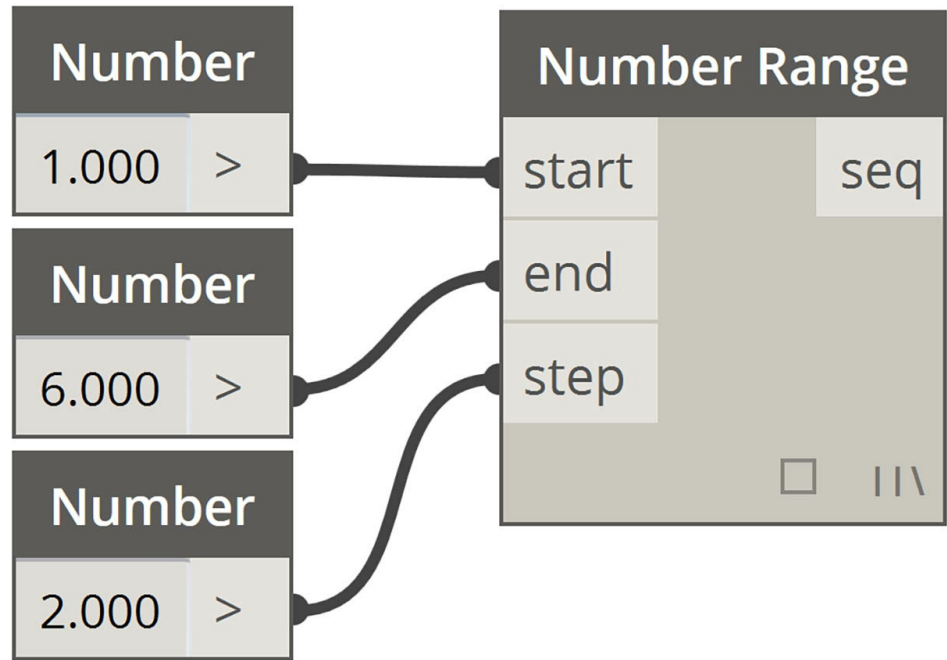
Abreviatura

Abreviatura

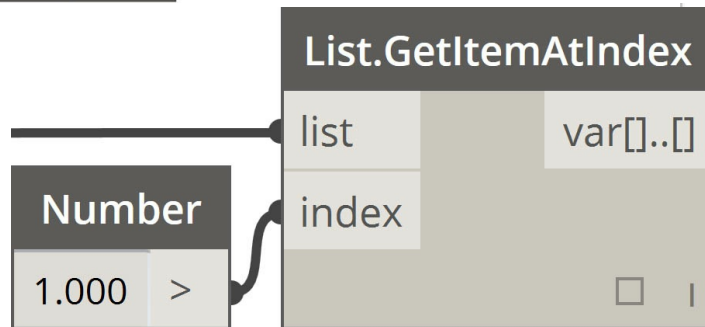
Existen varios métodos básicos de abreviatura en el bloque de código que, sencillamente, facilitan *mucho* la administración de datos. A continuación, vamos a desglosar los conceptos básicos y analizar cómo se puede utilizar la abreviatura para crear y consultar datos.



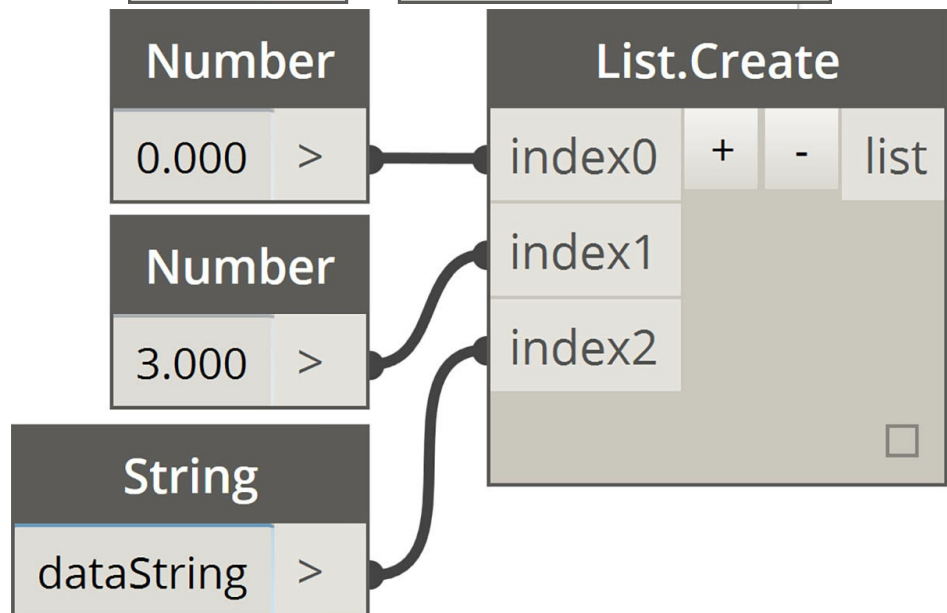
Rangos



Obtener elemento en índice

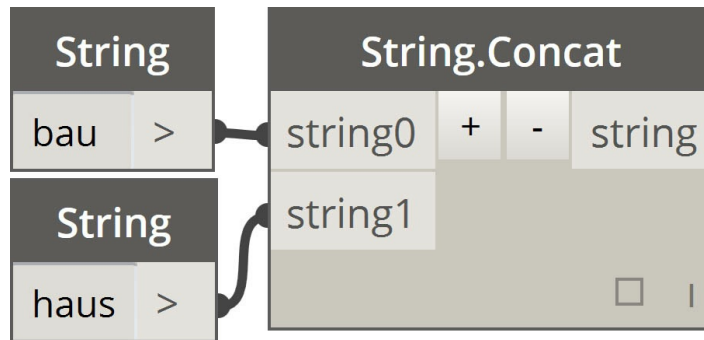


Crear lista

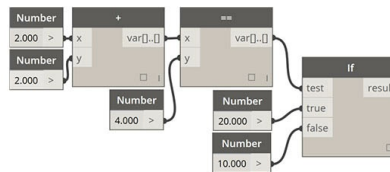


Concatenar cadenas

Concatenar
cadenas



Instrucciones
condicionales

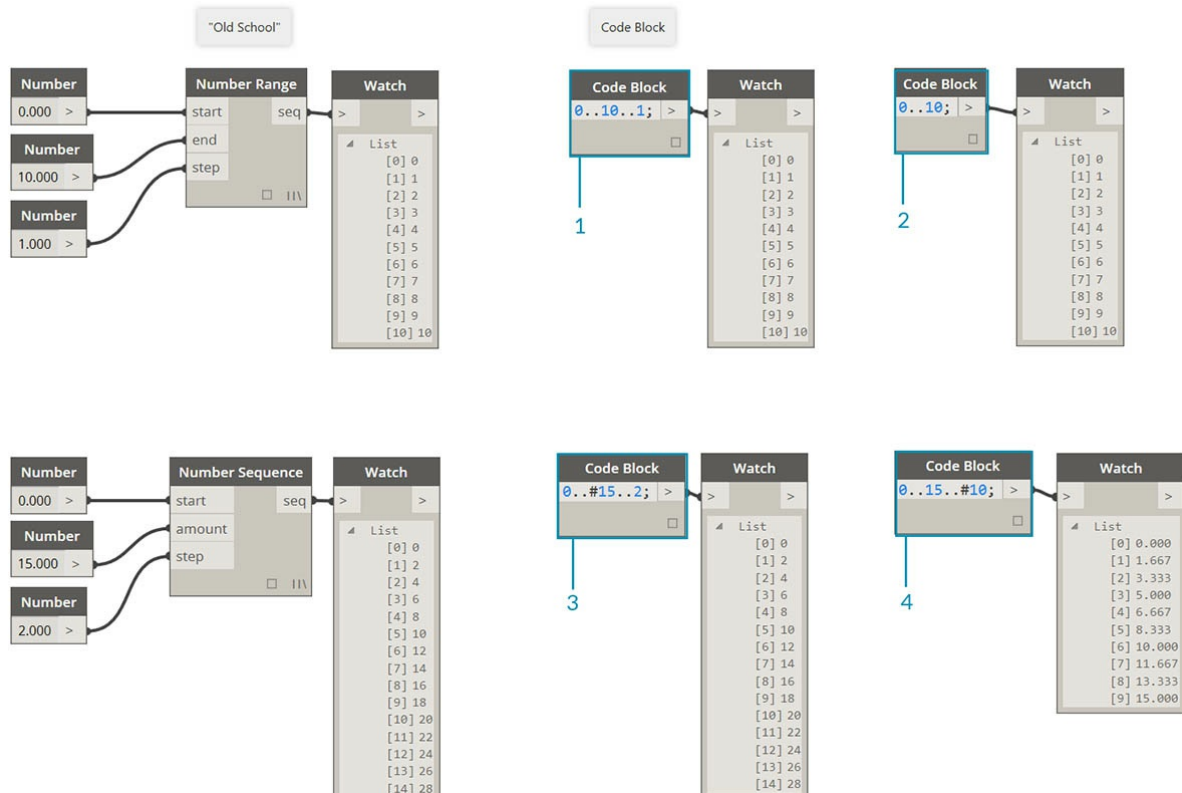


Sintaxis adicional

Nodo(s)	Bloque de código equivalente	Nota
Cualquier operador (+, &&, >=, Not, etc.)	+, &&, >=, !=, etc.	Tenga en cuenta que "Not" se convierte en "!" pero el nodo se denomina "Not" para distinguirlo de "Factorial".
Boolean True	true;	Se deben respetar las minúsculas.
Boolean False	false;	Se deben respetar las minúsculas.

Rangos

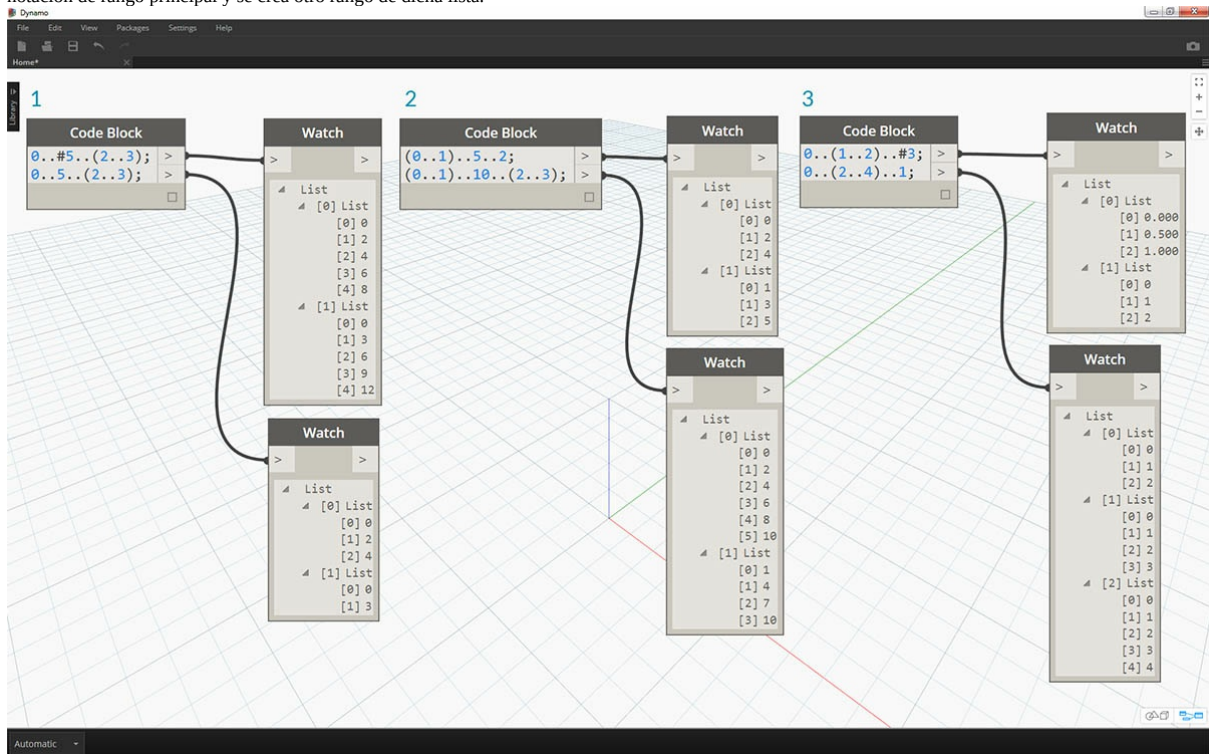
El método para definir rangos y secuencias se puede reducir a un método básico de abreviatura. Utilice la imagen siguiente como guía de la sintaxis "..." para definir una lista de datos numéricos con un bloque de código. Una vez que aprendemos a utilizar esta notación, crear datos numéricos es un proceso realmente eficaz:



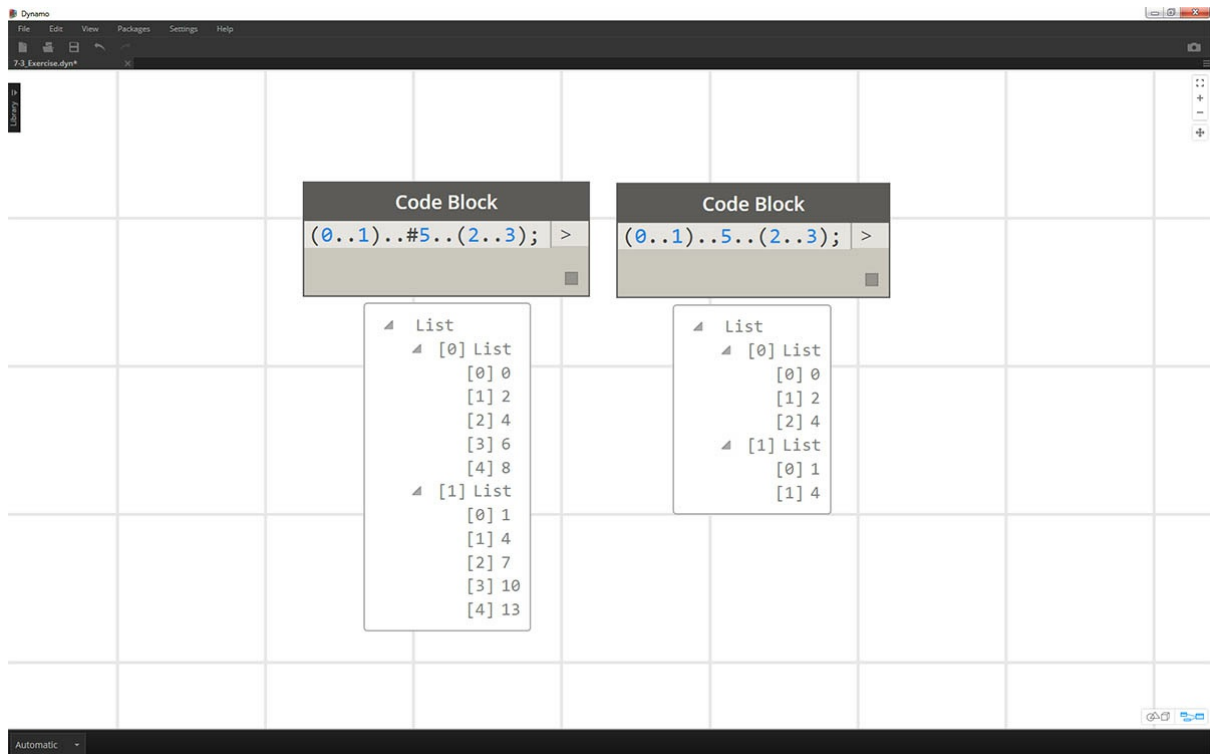
3. El ejemplo de *secuencia numérica* es similar, excepto que se utiliza un símbolo "#" para indicar que queremos que haya 15 valores en la lista, en lugar de una lista que llega hasta 15. En este caso, definimos: *inicio* . . *#dePasos* . . *tamaño-de-paso* : . La sintaxis real de la secuencia es 0 . . #15 . . 2.
4. Ahora vamos a introducir el símbolo "#" del paso anterior en la sección "*tamaño-de-paso*" de la sintaxis. Ahora, tenemos un *rango de números* que abarca de "*inicio*" a "*fin*" y la notación de "*tamaño de paso*" distribuye uniformemente un número de valores entre los dos elementos: *inicio* . . *fin* . . #*dePasos*.

Rangos avanzados

La creación de rangos avanzados nos permite trabajar con listas de listas de una forma sencilla. En los ejemplos siguientes, se aísla una variable de la notación de rango principal y se crea otro rango de dicha lista.



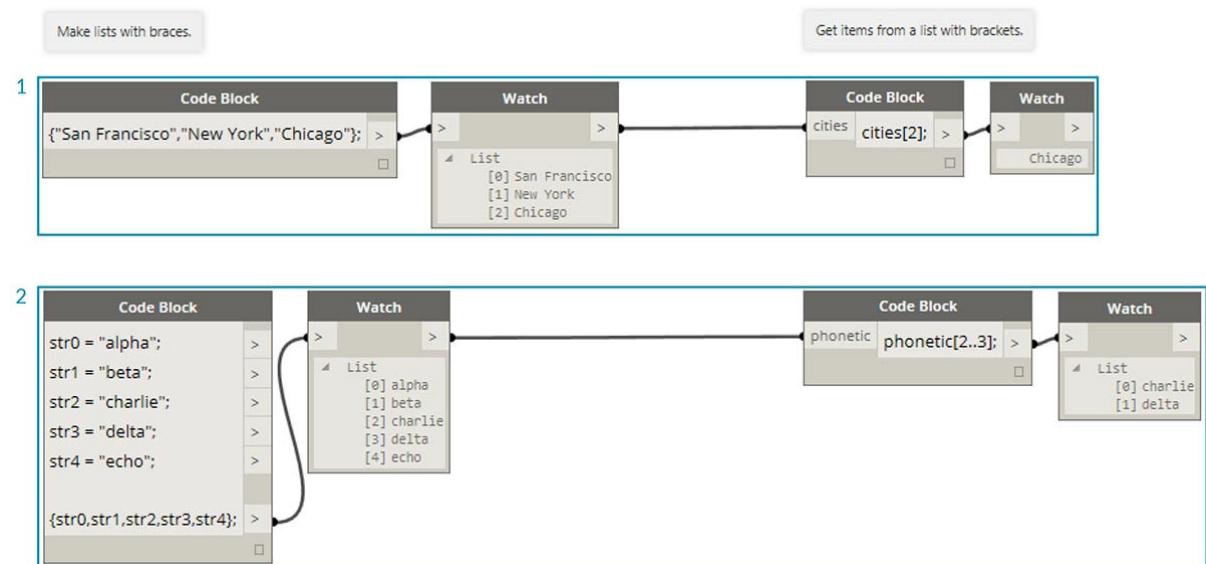
1. Mediante la creación de rangos anidados, compare la notación con el símbolo "#" frente a la notación sin este símbolo. Se aplica la misma lógica que en los rangos básicos, pero se vuelve un poco más compleja.
2. Podemos definir un subrango en cualquier lugar dentro del rango principal, y observe que también podemos tener dos subrangos.
3. Al controlar el valor "*fin*" de un rango, se crean más rangos de longitudes diferentes.



Como ejercicio lógico, compare las dos abreviaturas anteriores e intente analizar el modo en que los *subrangos* y la notación "#" controlan la salida.

Crear listas y obtener elementos de una lista

Además de crear listas con la función de abreviatura, también podemos crear listas sobre la marcha. Estas listas pueden contener una amplia gama de tipos de elementos y también se pueden consultar (recuerde que las listas son objetos en sí mismas). En resumen, con el bloque de código se crean listas con llaves y se consultan los elementos de una lista con corchetes:



1. Cree listas rápidamente con cadenas y consulte las listas mediante el índice de elementos.
2. Cree listas con variables y realice consultas utilizando la notación de abreviatura de rango.

Además, el proceso de gestión con listas anidadas es similar. Tenga en cuenta el orden de la lista y recuerde utilizar varios conjuntos de corchetes:

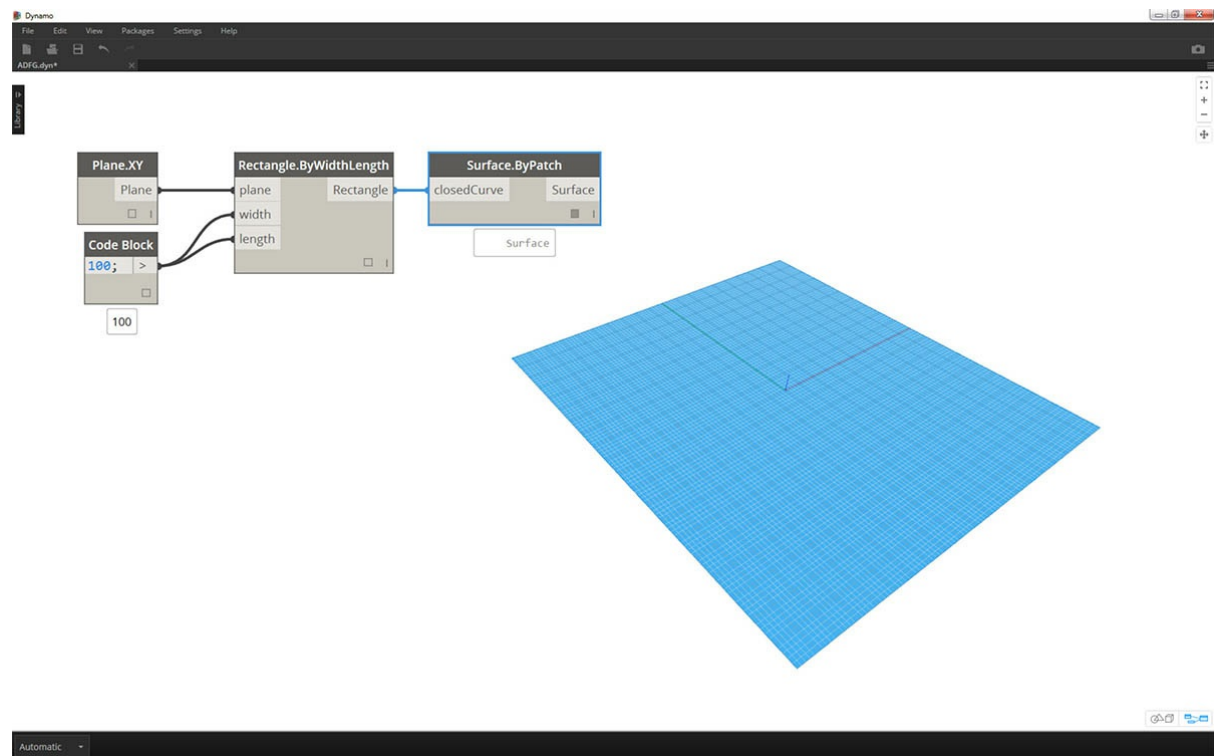


1. Defina una lista de listas.
2. Consulte una lista con notación de un solo corchete.
3. Consulte un elemento con notación de doble corchete.

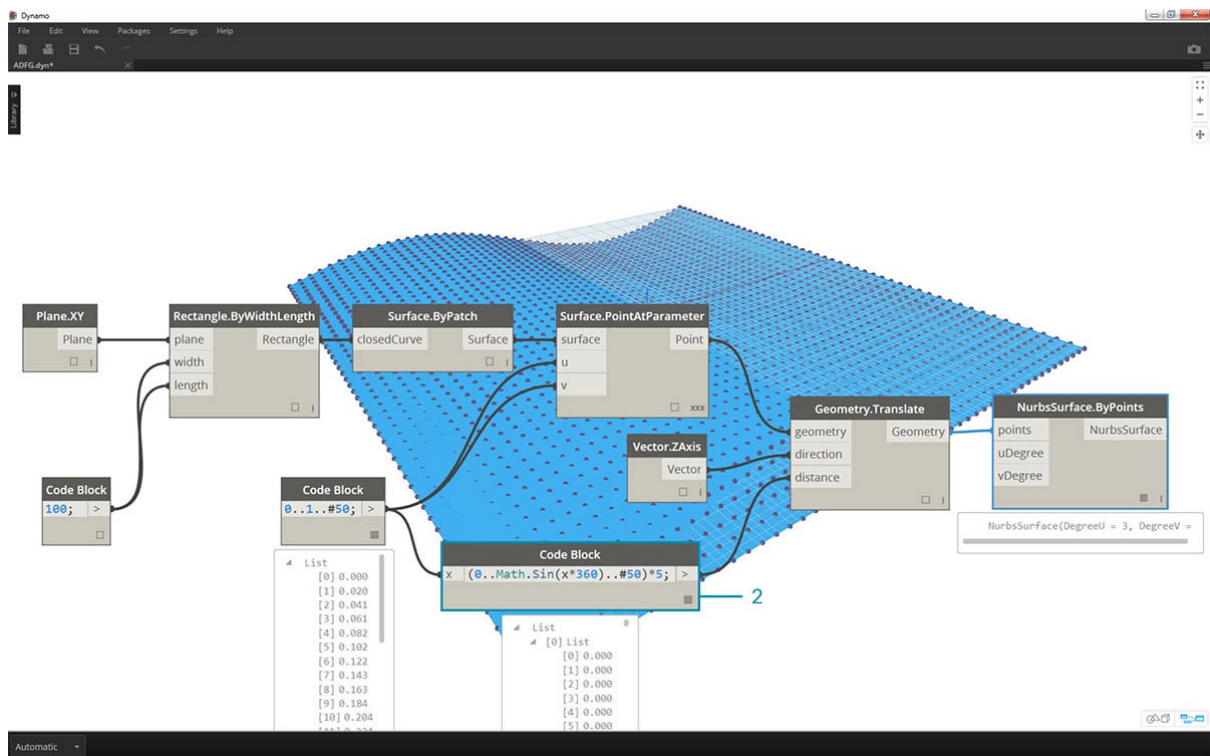
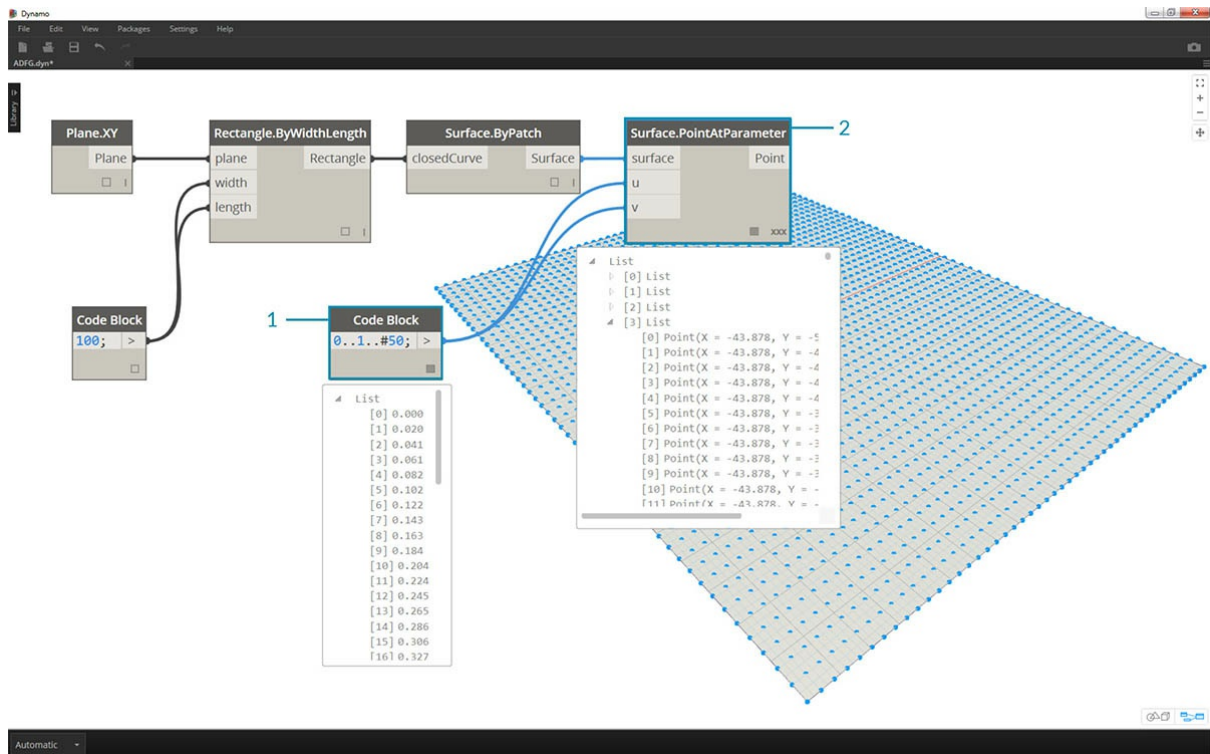
Ejercicio

Descargue el archivo de ejemplo que acompaña a este ejercicio (haga clic con el botón derecho y seleccione "Guardar vínculo como..."). En el Apéndice se incluye una lista completa de los archivos de ejemplo. [Obsolete-Nodes_Sine-Surface.dyn](#)

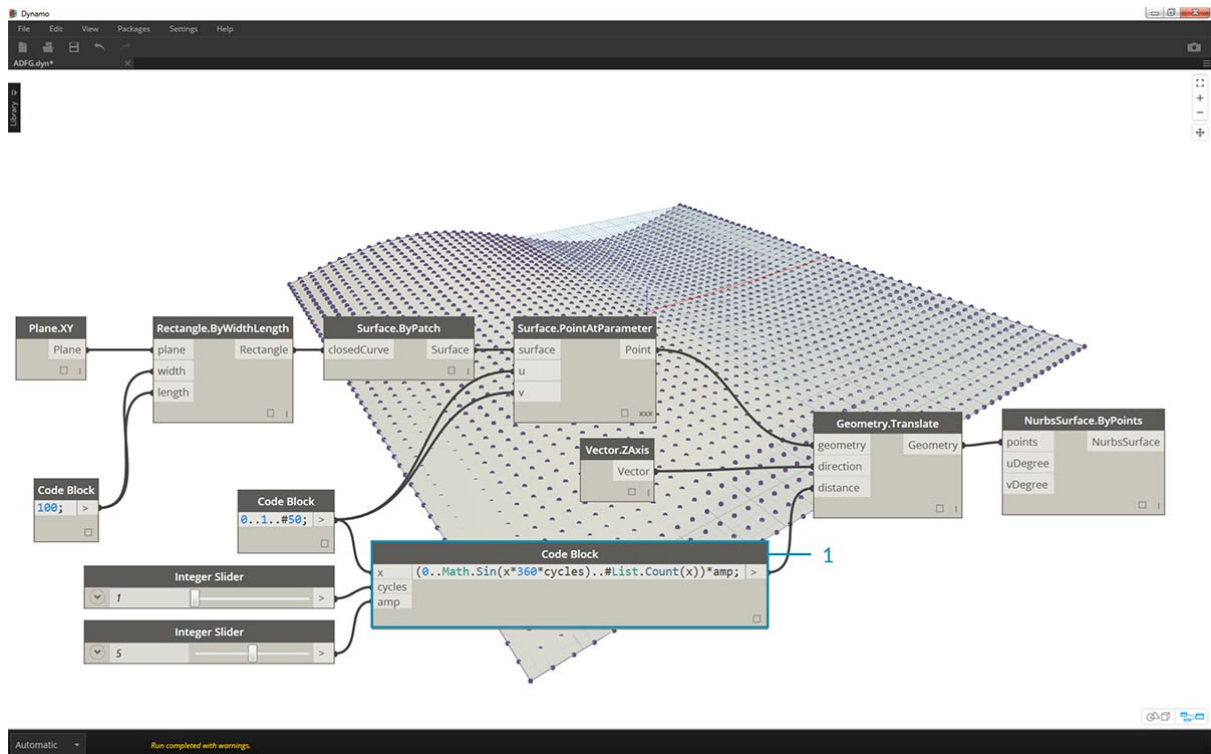
En este ejercicio, vamos a poner en práctica nuestras nuevas habilidades de abreviación para crear una superficie de cáscaras de huevo con mucho estilo y definida por rangos y fórmulas. A lo largo de este ejercicio, observe cómo utilizamos el bloque de código y los nodos existentes en Dynamo en tándem: se utiliza el bloque de código para la elevación de datos pesados mientras que los nodos de Dynamo se disponen visualmente para determinar la legibilidad de la definición.



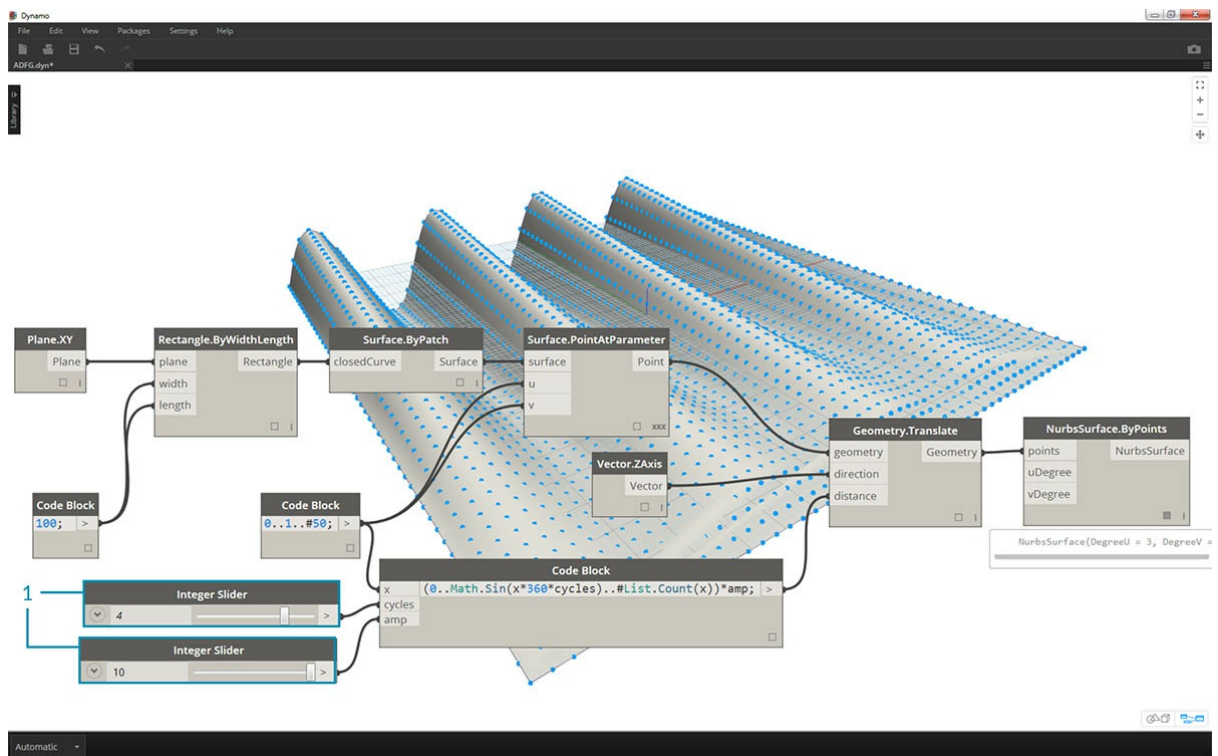
Comience por crear una superficie conectando los nodos anteriores. En lugar de utilizar un nodo numérico para definir la anchura y la longitud, haga doble clic en el lienzo y escriba 100; en un bloque de código.



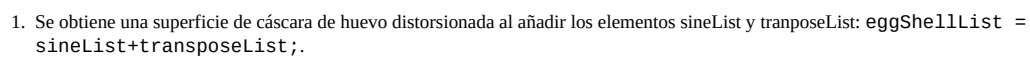
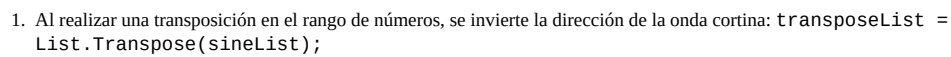
En este paso, utilizamos nuestra primera función para mover la rejilla de puntos hacia arriba en el eje Z. Esta rejilla controlará una superficie generada a partir de la función subyacente.

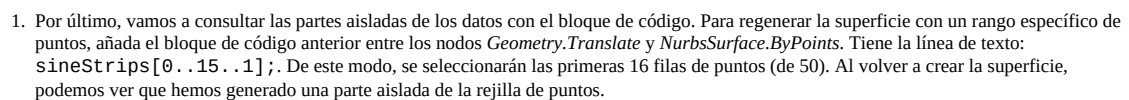
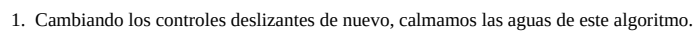


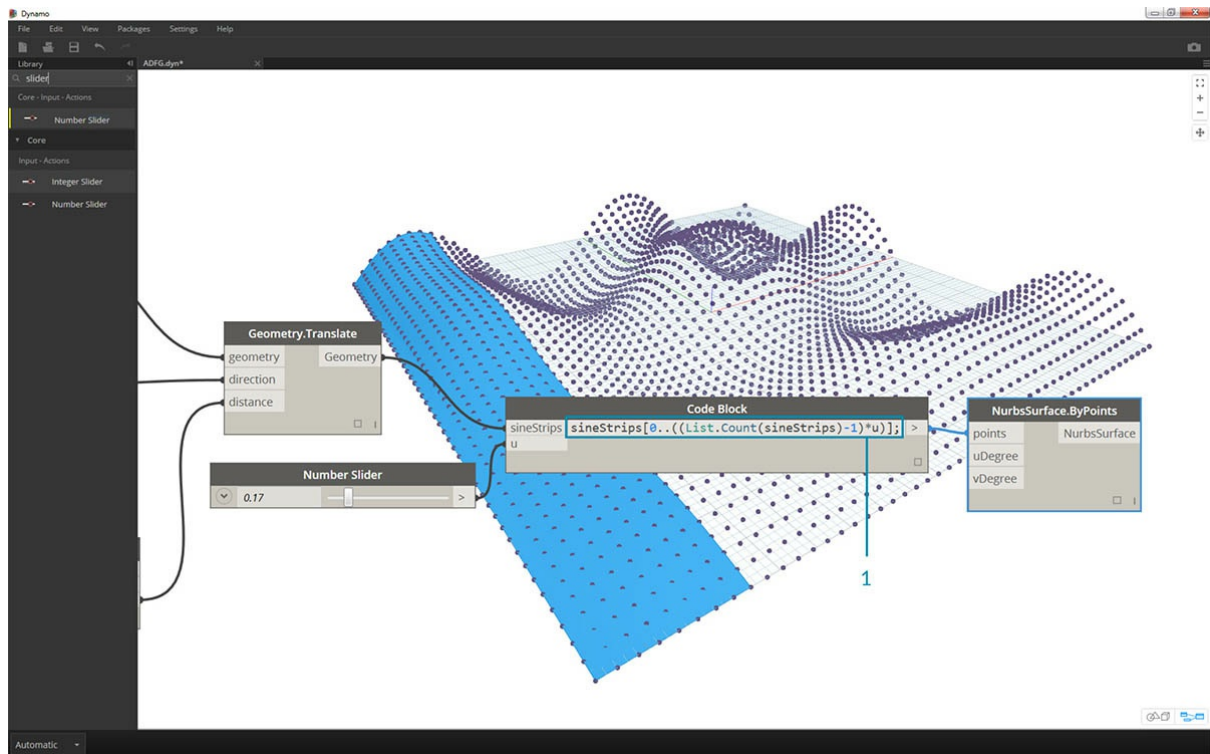
1. Aunque el bloque de código anterior funcionaba bien, no era totalmente paramétrico. Queremos controlar sus parámetros de forma dinámica, por lo que vamos a sustituir la línea del paso anterior por $(0..Math.Sin(x*360*cycles)..#List.Count(x))*amp;$. Esto nos permite definir estos valores en función de las entradas.



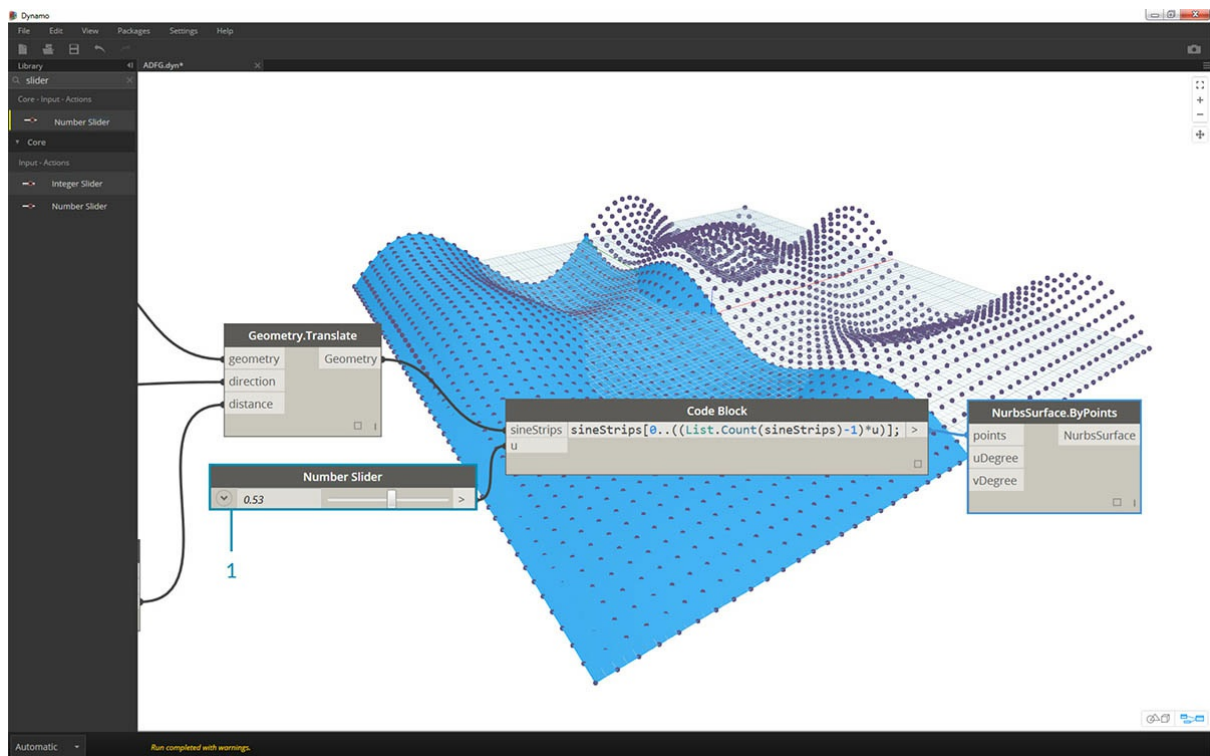
1. Al cambiar los controles deslizantes (de 0 a 10), obtenemos resultados interesantes.







1. En el paso final, para que este bloque de código sea más paramétrico, la consulta se controla mediante un control deslizante que va de 0 a 1. Lo hacemos con esta línea de código: `sineStrips[0..((List.Count(sineStrips)-1)*u)];`. Esto puede resultar confuso, pero la línea de código nos proporciona una forma rápida de modificar la longitud de la lista con un multiplicador entre 0 y 1.



1. Un valor de 0,53 en el control deslizante crea una superficie justo después del punto medio de la rejilla.

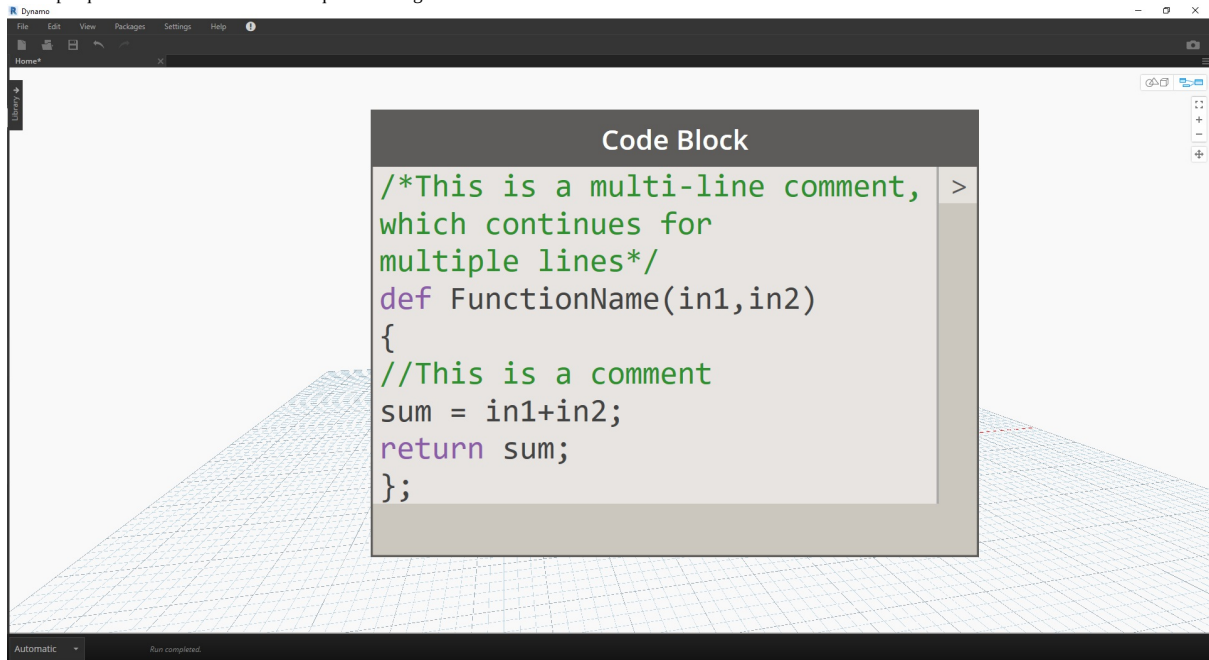
Funciones

Funciones de bloque de código

Las funciones se pueden crear en un bloque de código y recuperar en cualquier parte de una definición de Dynamo. Esto crea otra capa de control en un archivo paramétrico y se puede ver como una versión de texto de un nodo personalizado. En este caso, se puede acceder fácilmente al bloque de código "principal", que se encuentra en cualquier parte del gráfico. No se necesitan cables.

Elemento principal

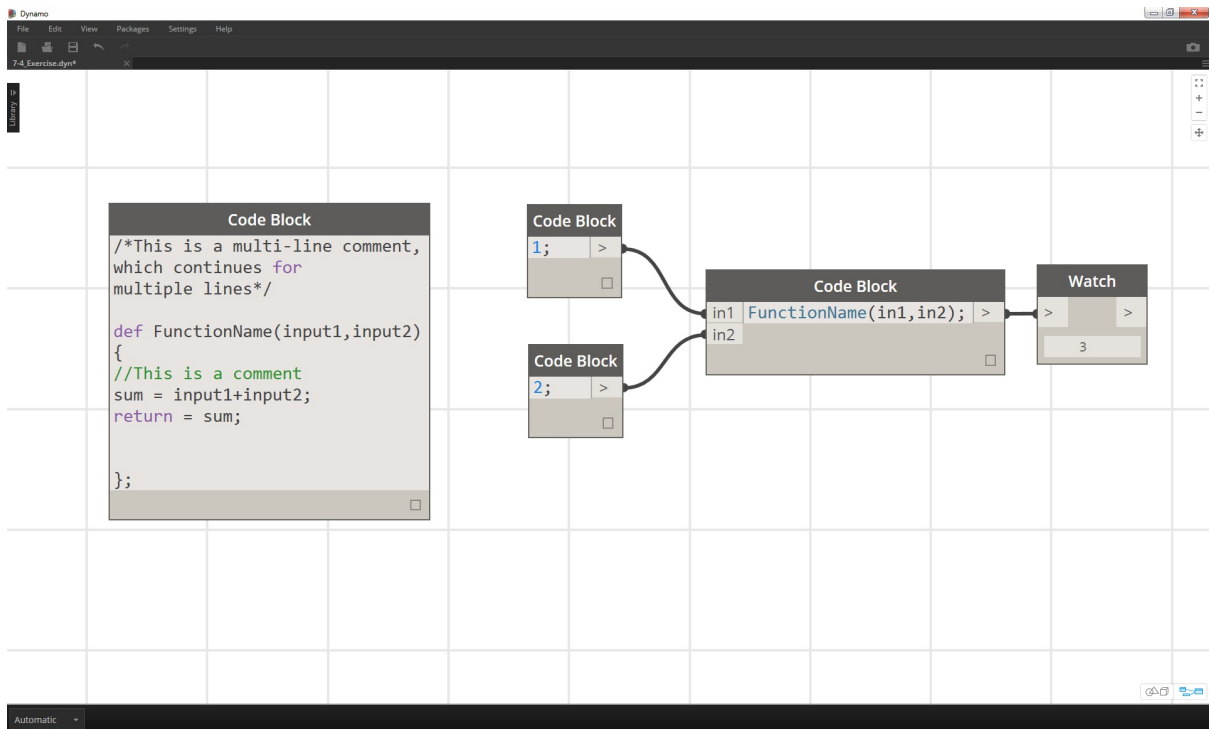
En la primera línea, aparece la palabra clave "def", después el nombre de la función y, a continuación, los nombres de las entradas entre paréntesis. Las llaves definen el cuerpo de la función. Se devuelve un valor con "return =". Los bloques de código que definen una función no tienen puertos de entrada o salida porque se invocan desde otros bloques de código.



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

Elementos secundarios

Invoque la función con otro bloque de código en el mismo archivo asignando el nombre y la misma cantidad de argumentos. Funciona igual que los nodos predefinidos de la biblioteca.

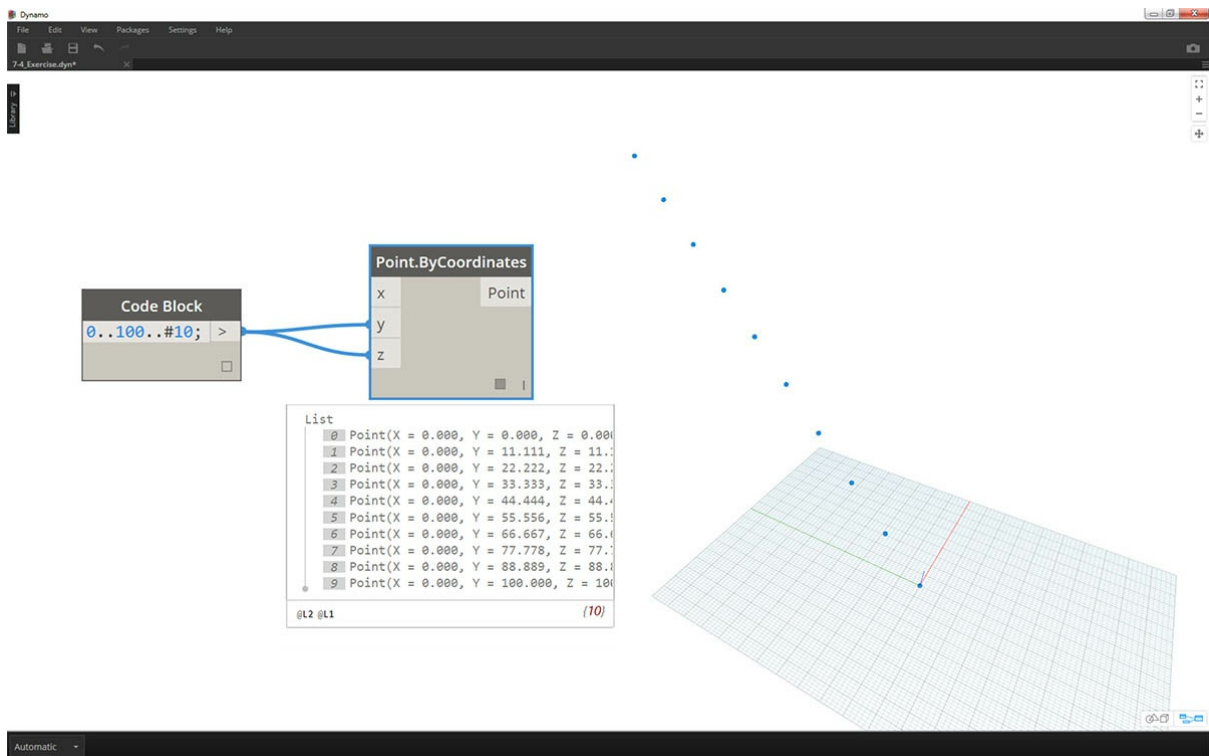


FunctionName(in1,in2);

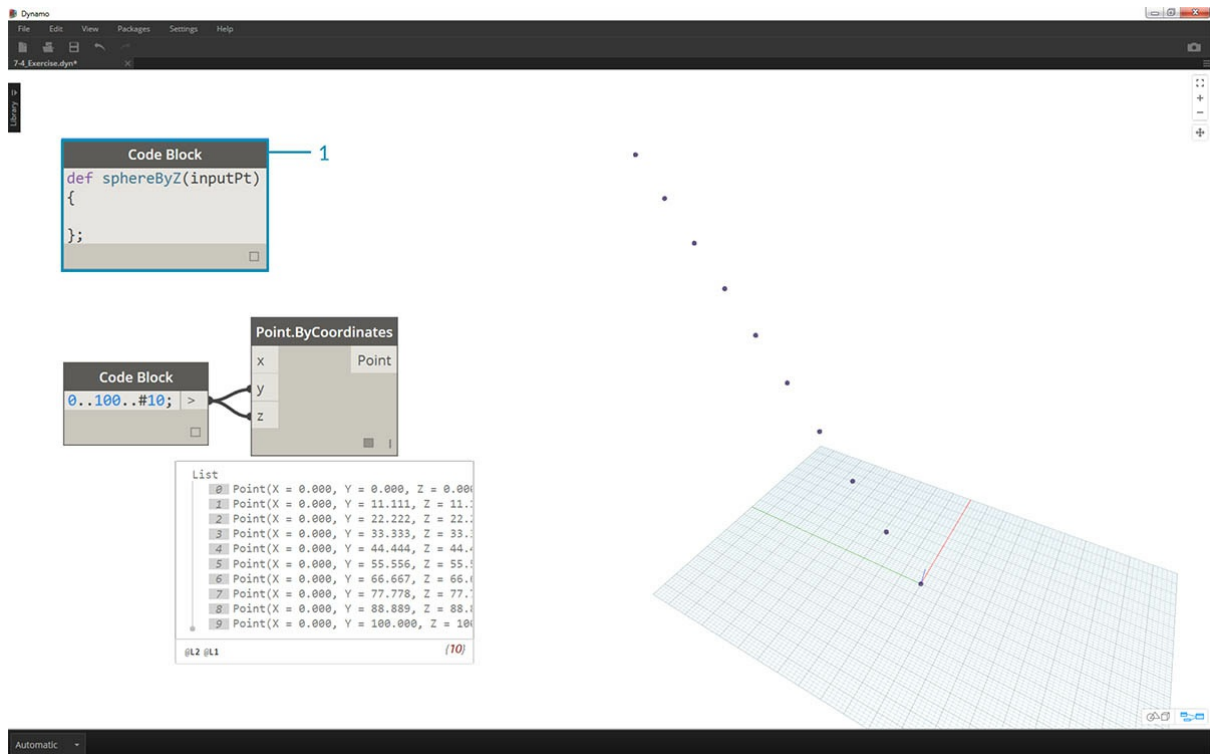
Ejercicio

Descargue el archivo de ejemplo que acompaña a este ejercicio (haga clic con el botón derecho y seleccione "Guardar vínculo como..."). En el Apéndice, se incluye una lista completa de los archivos de ejemplo. [Functions_SphereByZ.dyn](#)

En este ejercicio, crearemos una definición genérica que creará esferas a partir de una lista de entrada de puntos. El radio de estas esferas depende de la propiedad Z de cada punto.



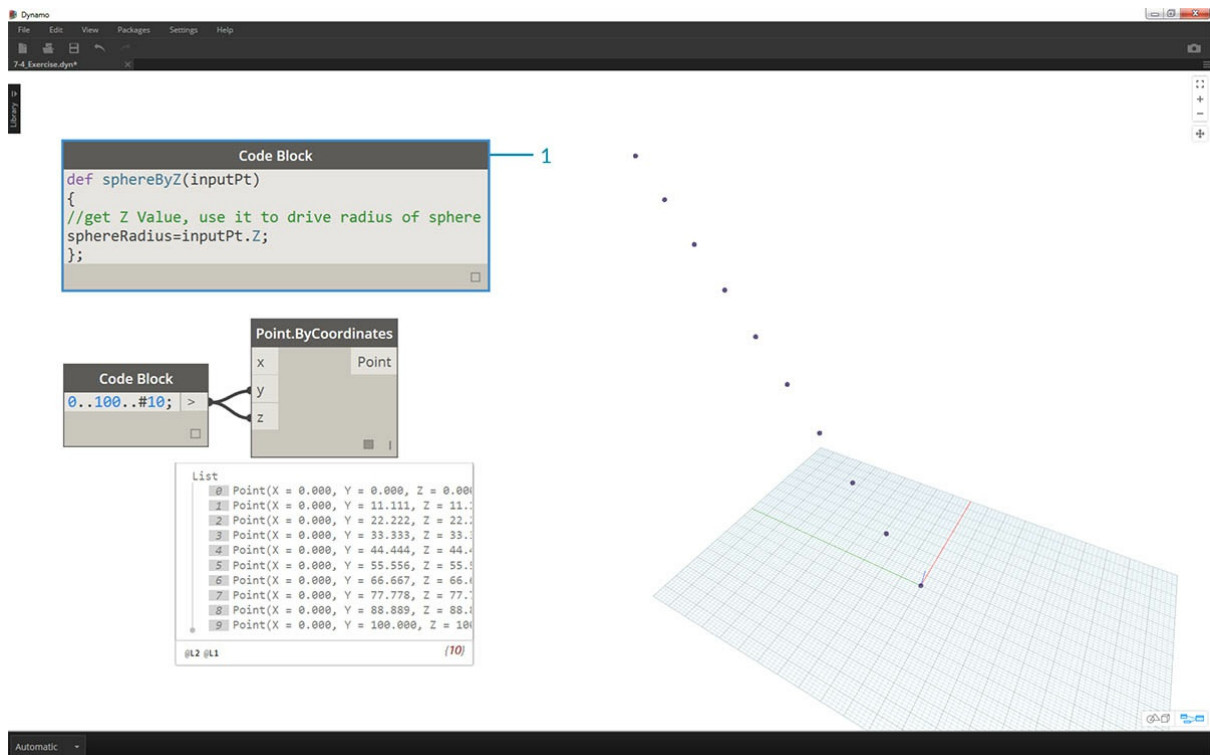
Comencemos con un intervalo de diez valores que abarca de 0 a 100. Conecte estos nodos a un nodo *Point.ByCoordinates* para crear una línea diagonal.



1. Cree un bloque de código e introduzca la definición mediante la línea de código:

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

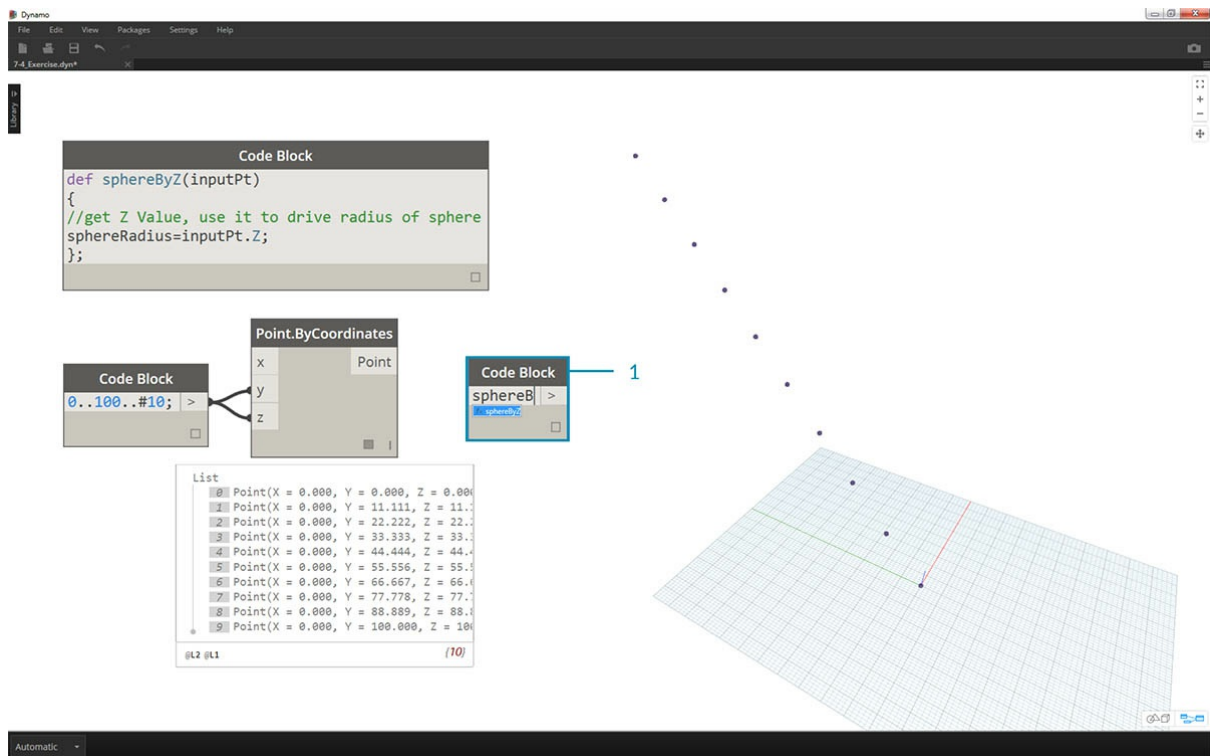
El parámetro *inputPt* es el nombre que se le ha asignado para representar los puntos que controlarán la función. En este momento, la función no realiza ninguna acción, pero la ampliaremos en los pasos siguientes.



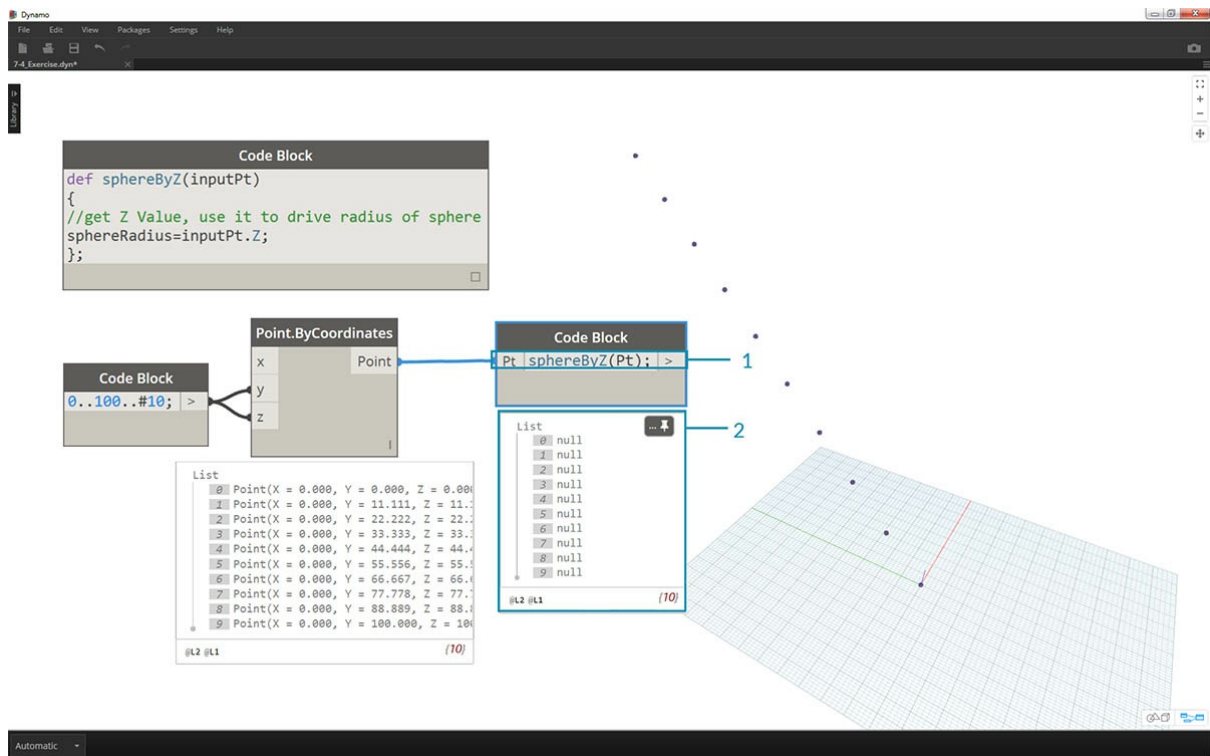
1. Al añadir la función de *bloque de código*, se coloca un comentario y una variable *sphereRadius* que consulta la posición Z de cada punto. Recuerde que *inputPt.Z* no necesita paréntesis como método. Se trata de una *consulta* de las propiedades de un elemento existente, por lo que no se necesita ninguna entrada:

```
def sphereByZ(inputPt, radiusRatio)
{
```

```
//get Z Value, use it to drive radius of sphere
sphereRadius=inputPt.Z;
};
```



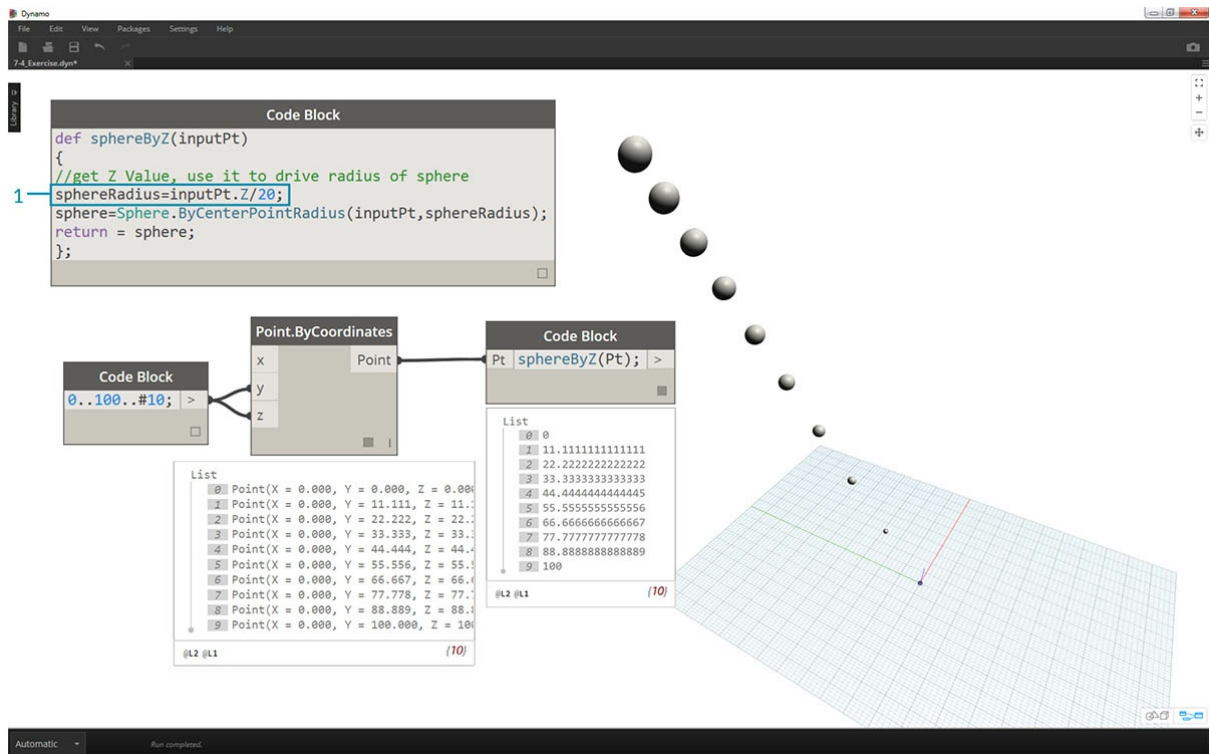
1. Ahora, vamos a recuperar la función que hemos creado en otro *bloque de código*. Si hacemos doble clic en el lienzo para crear un nuevo *bloque de código* y escribimos *sphereB*, observaremos que Dynamo sugiere la función *sphereByZ* que hemos definido. La función se ha añadido a la biblioteca de IntelliSense. ¡Genial!



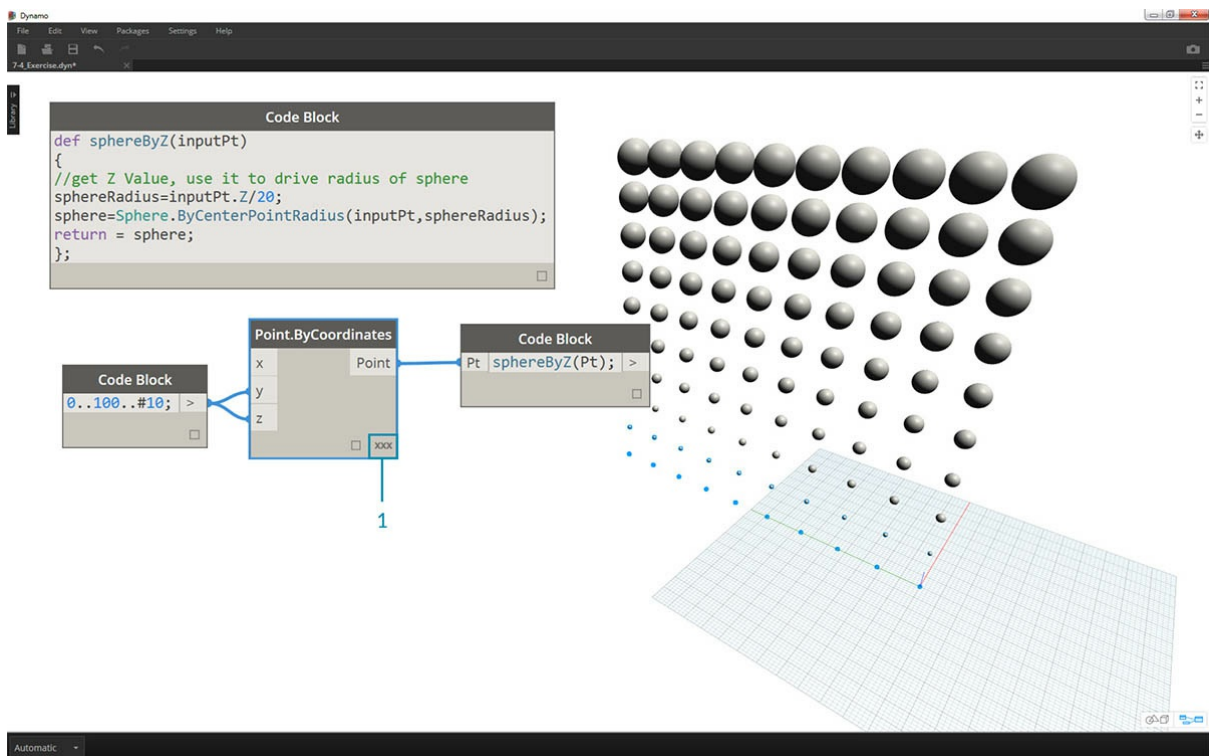
1. Ahora llamamos a la función y creamos una variable denominada *Pt* para conectar los puntos creados en los pasos anteriores:

`sphereByZ(Pt)`

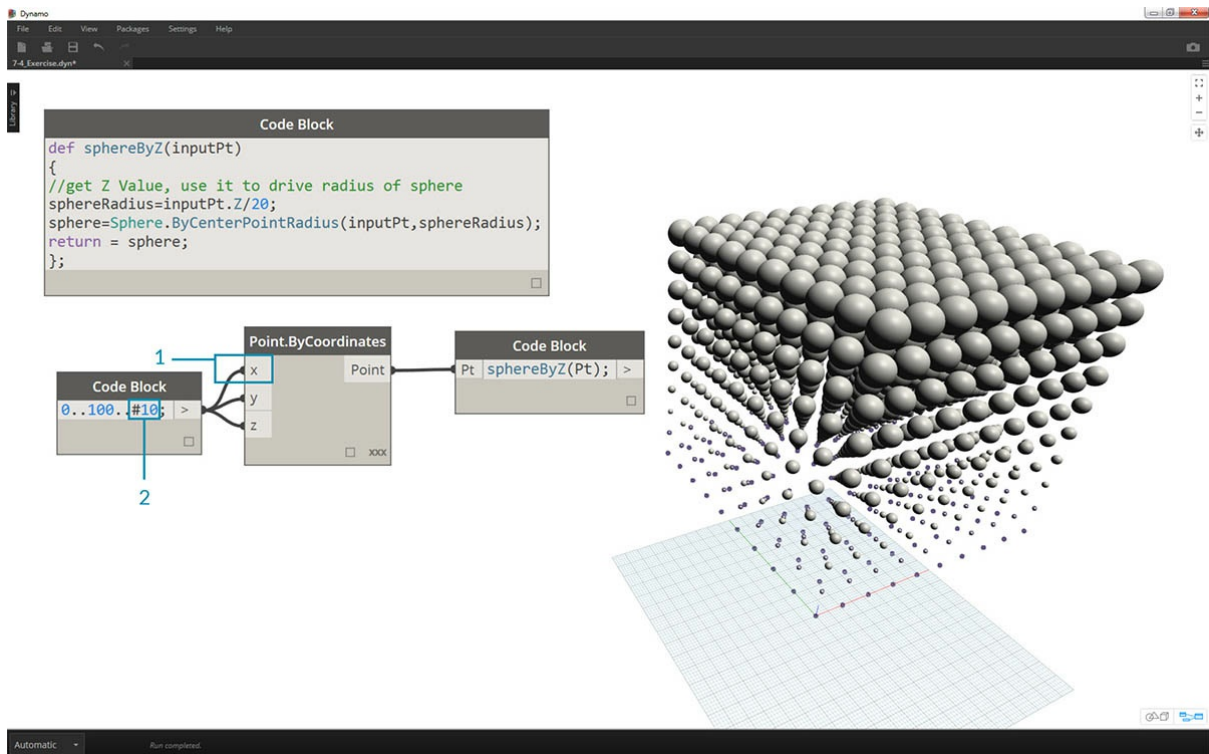
1. En la salida, podemos observar que todos los valores son nulos. ¿Por qué ocurre eso? Al definir la función, calculamos la variable *sphereRadius*, pero no hemos definido qué debe *devolver* la función como una *salida*. Podemos solucionar esto en el siguiente paso.



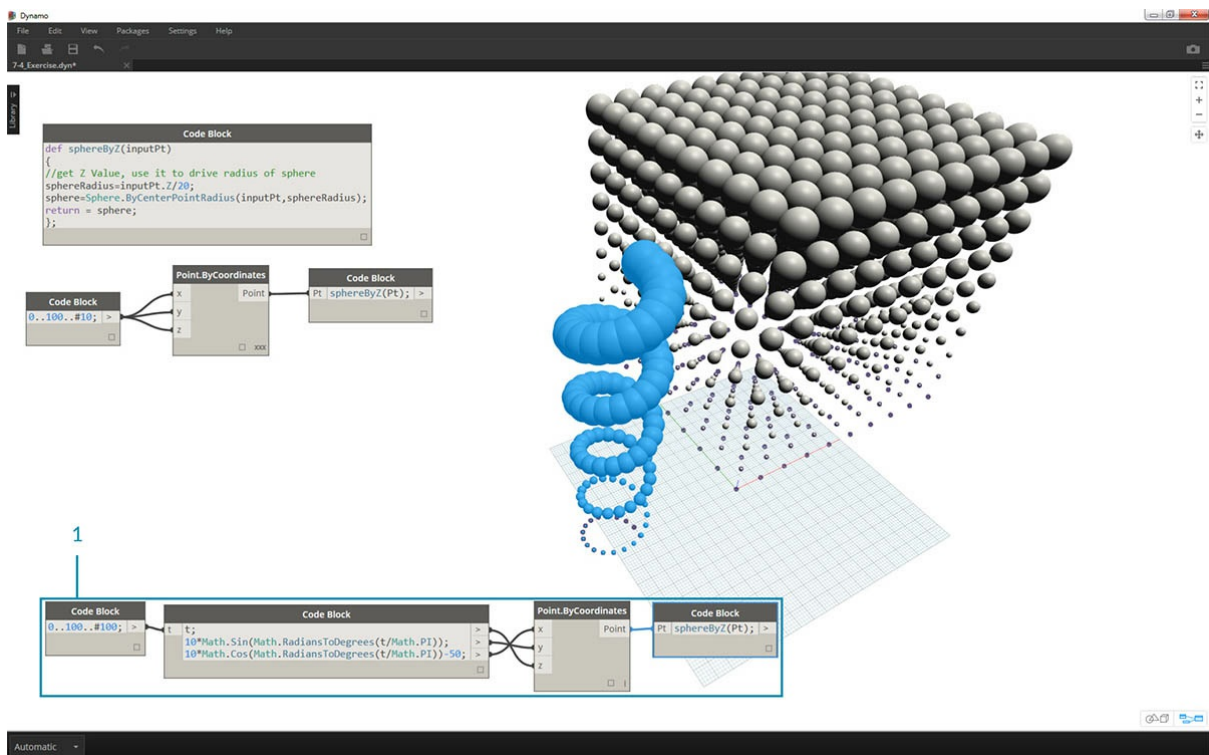
1. Para reducir el tamaño de estas esferas, vamos a actualizar el valor de *sphereRadius* mediante la adición de un divisor: $sphereRadius = inputPt.Z/20$. Ahora podemos ver las diferentes esferas y comenzar a desentrañar la relación entre el radio y el valor Z.



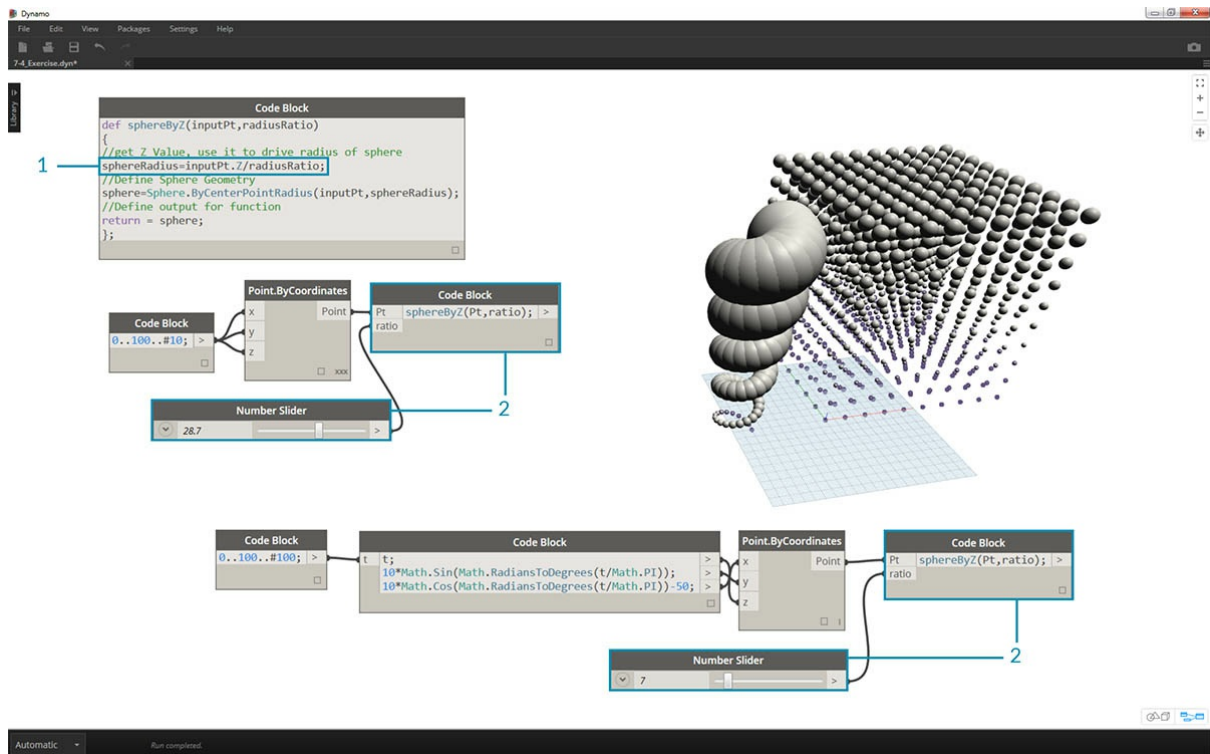
1. En el nodo *Point.ByCoordinates*, al cambiar el encaje de *Más corto* a *Producto cartesiano*, creamos una rejilla de puntos. La función *sphereByZ* sigue estando totalmente activa, por lo que todos los puntos crean esferas con radios basados en valores Z.



1. Y, solo para tantear el terreno, conectamos la lista original de números a la entrada X de *Point.ByCoordinates*. Ahora tenemos un cubo de esferas.
2. Nota: si el cálculo tarda mucho en completarse en el equipo, pruebe a cambiar el valor #10 por un valor como #5.



1. Recuerde que la función *sphereByZ* que hemos creado es una función genérica, por lo que podemos recuperar la hélice de una lección anterior y aplicarle la función.



Como paso final, vamos a controlar la relación de radio con un parámetro definido por el usuario. Para ello, debemos crear una entrada nueva para la función y reemplazar el divisor 20 por un parámetro.

1. Actualice la definición de *sphereByZ* a:

```
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. Actualice los bloques de código secundarios mediante la adición de una variable *ratio* a la entrada: *sphereByZ(Pt, ratio)*; . Conecte un control deslizante a la entrada de bloque de código recién creada y varíe el tamaño de los radios en función de la relación de radio.

Dynamo para Revit

Dynamo para Revit

Dynamo es un entorno flexible diseñado para su uso en una amplia gama de programas y creado originalmente para su uso con Revit. Un programa visual crea opciones eficaces para BIM (Building Information Modeling). Dynamo ofrece un conjunto completo de nodos diseñados específicamente para Revit, así como bibliotecas de terceros de una creciente comunidad de AEC. Este capítulo se centra en los conceptos básicos de uso de Dynamo en Revit.



La conexión de Revit

La conexión de Revit



Dynamo para Revit amplía el modelado de información de construcción mediante el entorno de datos y lógica de un editor gráfico de algoritmos. Su flexibilidad, junto con una base de datos de Revit robusta, ofrece una nueva perspectiva para BIM.

Este capítulo se centra en los flujos de trabajo de Dynamo para BIM. Las secciones se basan principalmente en ejercicios, ya que la mejor forma de familiarizarse con un editor gráfico de algoritmos para BIM es pasar directamente a un proyecto. No obstante, veamos primero los inicios del programa.

#

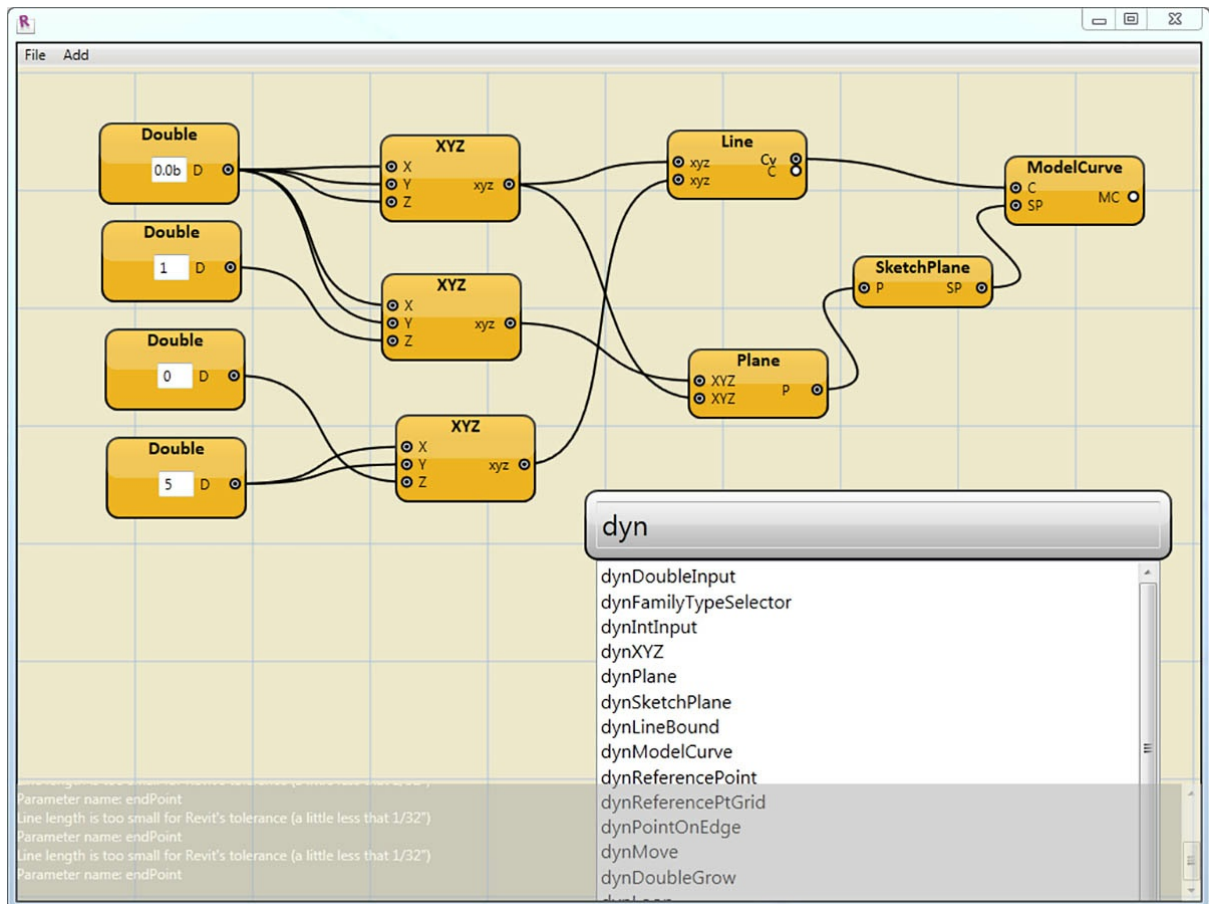
Compatibilidad con versiones de Revit

A medida que Revit y Dynamo continúan evolucionando, es posible que observe que la versión de Revit con la que está trabajando no es compatible con la versión de Dynamo para Revit que ha instalado en su equipo. A continuación, se describen las versiones de Dynamo para Revit que son compatibles con Revit.

Versión de Revit	Primera versión de Dynamo estable	Última versión de Dynamo para Revit compatible
2013	0.6.1	0.6.3
2014	0.6.1	0.8.2
2015	0.7.1	1.2.1
2016	0.7.2	1.3.2
2017	0.9.0	1.3.4 / 2.0.3
2018	1.3.0	1.3.4 / 2.0.3
2019	1.3.3	1.3.4 / 2.0.3
2020	2.1.0: Revit 2020 ahora incluye Dynamo y recibe actualizaciones al igual que Revit.	N/A

#

Historia de Dynamo

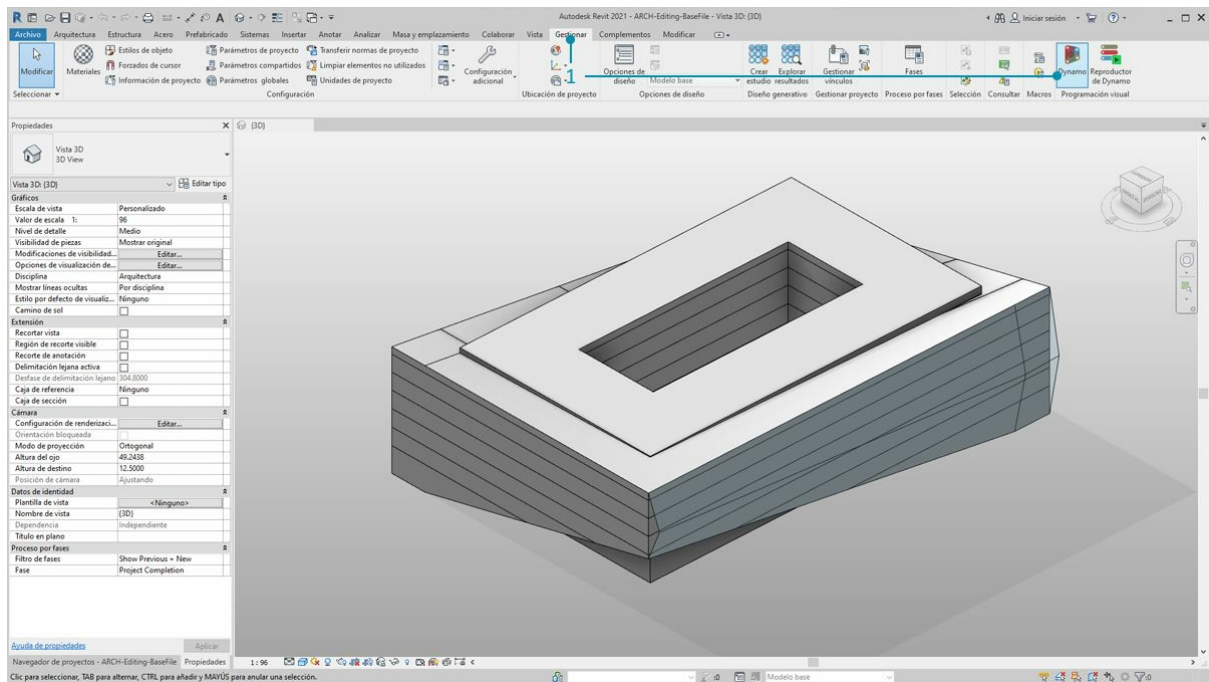


Con un equipo dedicado de desarrolladores y una ferviente comunidad, el proyecto ha avanzado mucho desde sus humildes inicios.

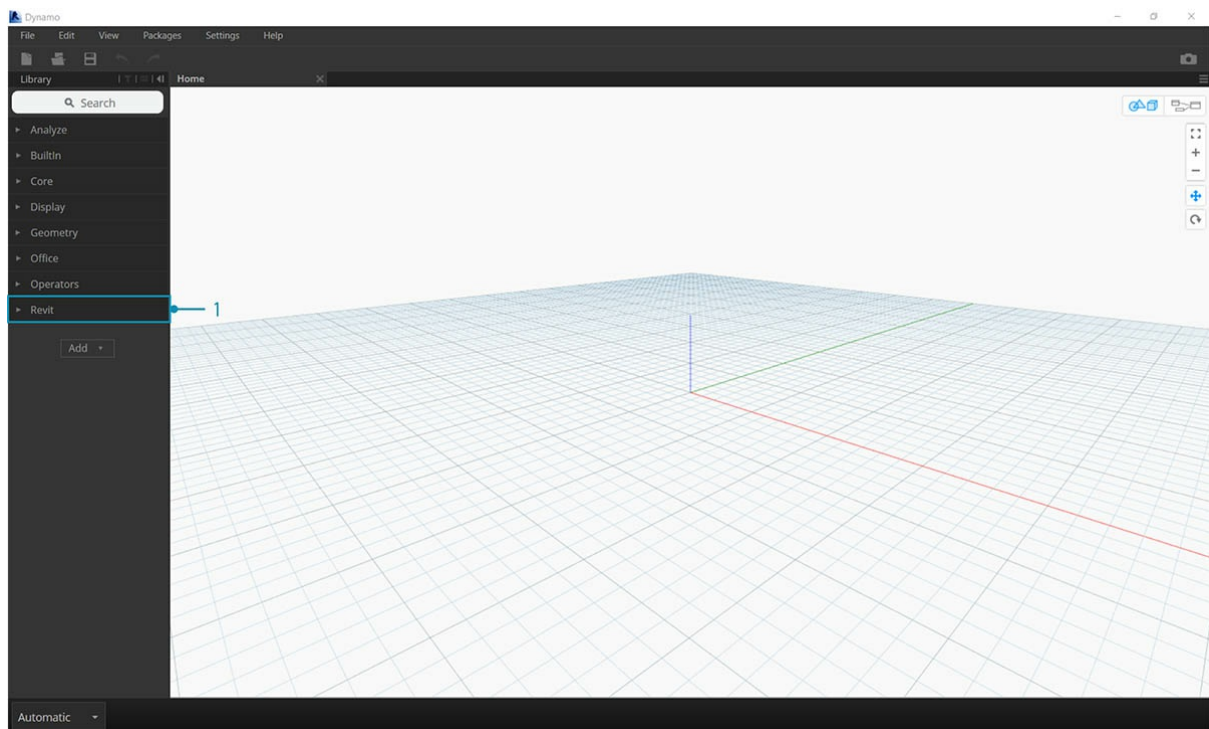
Dynamo se creó originalmente para agilizar los flujos de trabajo de AEC en Revit. Aunque Revit crea una robusta base de datos para cada proyecto, puede resultar difícil para un usuario medio acceder a esta información fuera de las restricciones de la interfaz. Revit incluye una API (interfaz de programación de aplicaciones) completa que permite a desarrolladores de terceros crear herramientas personalizadas. Y los programadores han estado utilizando esta API durante años, pero la creación de secuencias de comandos basadas en texto no está accesible para todos. Dynamo busca democratizar los datos de Revit a través de un editor gráfico de algoritmos de fácil manejo.

Mediante el uso de los nodos principales de Dynamo, junto con los nodos personalizados de Revit, un usuario puede ampliar sustancialmente los flujos de trabajo paramétricos en cuanto a interoperabilidad, documentación, análisis y generación. Con Dynamo, los flujos de trabajo tediosos pueden automatizarse, mientras que pueden impulsarse las exploraciones de diseño.

Ejecución de Dynamo en Revit



1. En un editor de proyectos o familias de Revit, vaya a Complementos y haga clic en *Dynamo*. Tenga en cuenta que Dynamo solo se ejecutará en el archivo en el que se abrió.



1. Al abrir Dynamo en Revit, hay una nueva categoría denominada *Revit*. Se trata de una adición completa a la interfaz de usuario que ofrece nodos que se adaptan específicamente a los flujos de trabajo de Revit.*

* Nota: el gráfico de *Dynamo* solo funcionará al abrirse en *Dynamo* para *Revit* mediante la familia de nodos específica de *Revit*. Si se abre un gráfico de *Dynamo* para *Revit* en *Dynamo Sandbox*, por ejemplo, no aparecerán los nodos de *Revit*.

Bloqueo de nodos

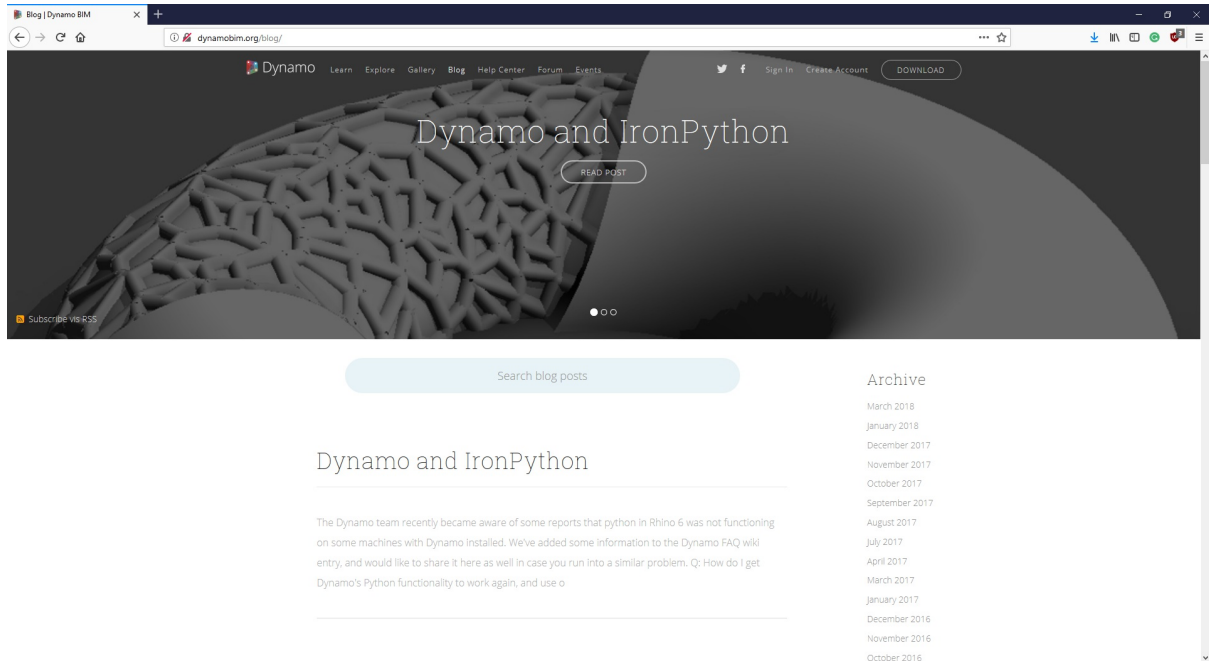
Como Revit es una plataforma que proporciona una sólida gestión de proyectos, las operaciones paramétricas de *Dynamo* pueden ser complejas y lentas de calcular. Si *Dynamo* tarda mucho tiempo en calcular los nodos, puede usar la función "Bloquear" para poner en pausa la ejecución de operaciones de *Revit* mientras desarrolla el gráfico. Para obtener más información sobre el bloqueo de nodos, consulte la sección "Bloqueo" del [capítulo sobre sólidos](#).

Comunidad

Como *Dynamo* se creó originalmente para AEC, su comunidad grande y creciente es un recurso excelente para aprender del sector y ponerse en contacto con expertos de la industria. La comunidad de *Dynamo* la integran arquitectos, ingenieros, programadores y diseñadores a los que les apasiona compartir y

crear.

Dynamo es un proyecto de código abierto que evoluciona constantemente y gran parte del desarrollo está relacionado con Revit. Si es un nuevo usuario, diríjase al foro de debate y comience a [publicar preguntas](#). Si es un programador y desea participar en el desarrollo de Dynamo, consulte la [página de GitHub](#). Además, un recurso excelente para las bibliotecas de terceros es [Dynamo Package Manager](#). Muchos de estos paquetes se crean teniendo en mente AEC y echaremos un vistazo a los paquetes de terceros para la panelización en este capítulo.

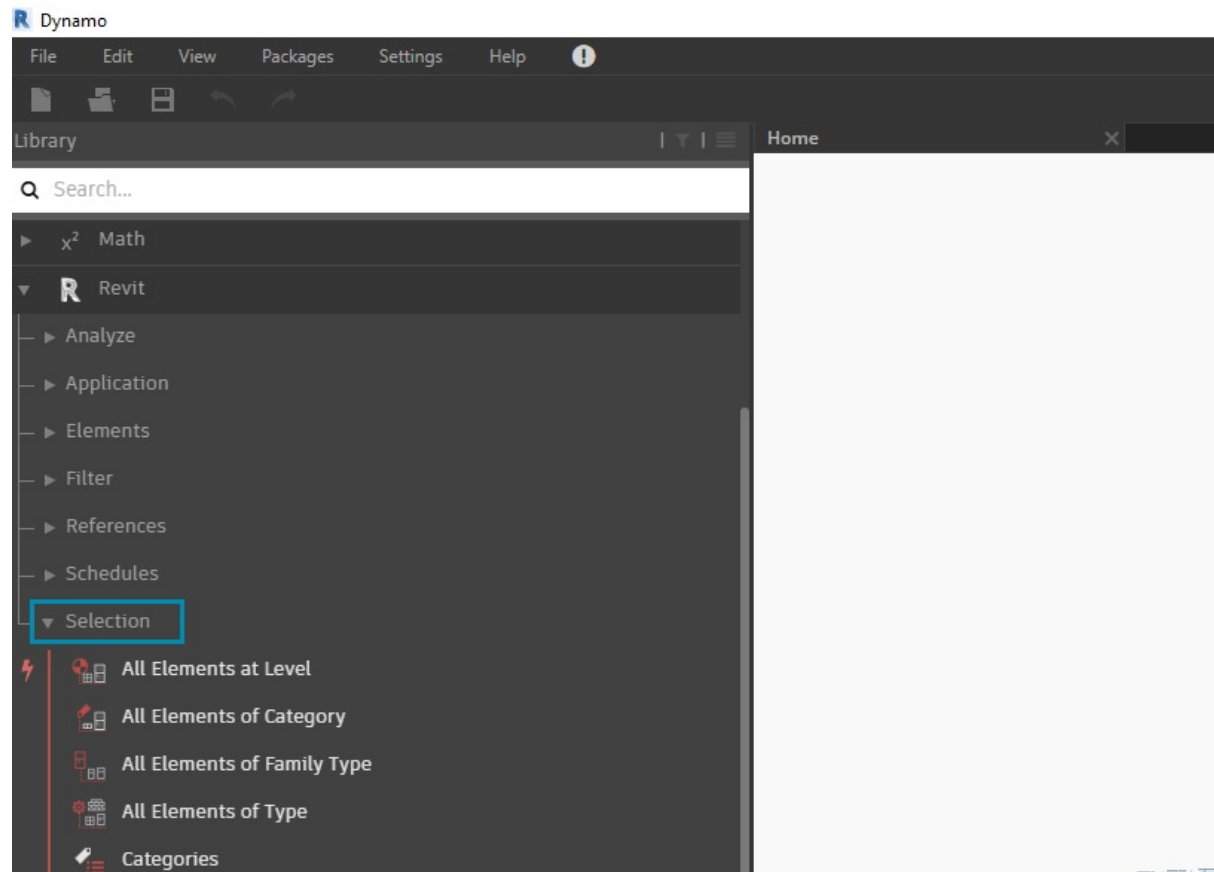


Dynamo también mantiene un [blog](#) activo. Lea las últimas publicaciones para obtener información sobre los avances más recientes.

Selección

Seleccionar

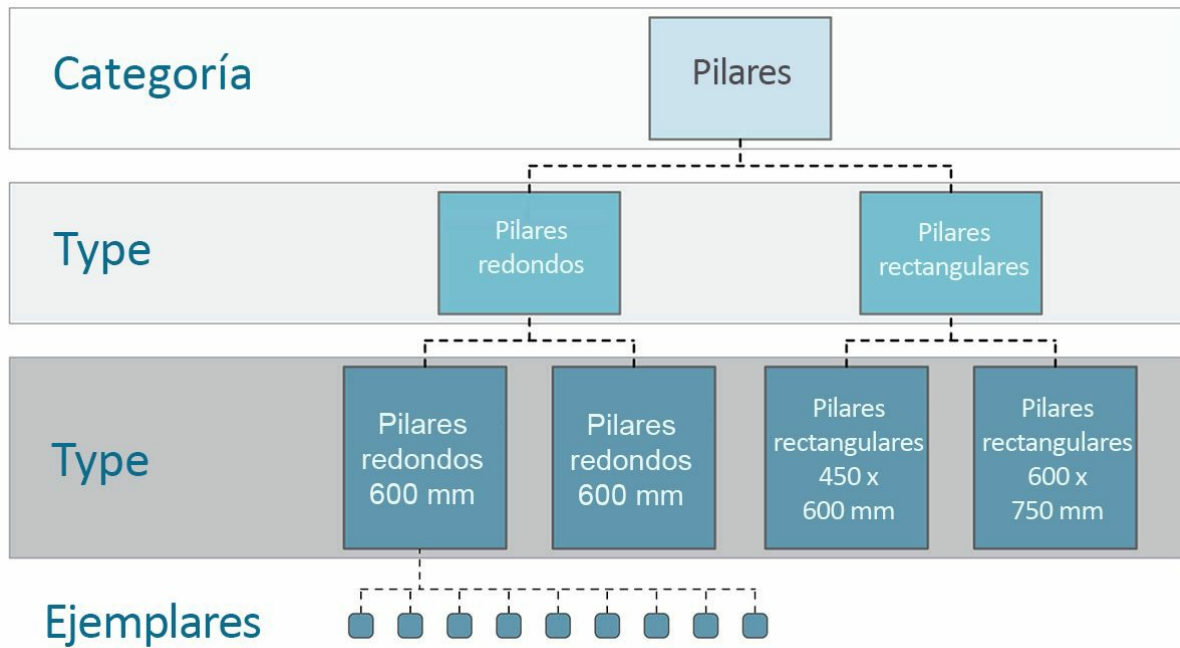
Revit es un entorno con gran cantidad de datos. Esto nos proporciona una gama de posibilidades de selección que se expande mucho más allá de "señalar y hacer clic". Podemos consultar la base de datos de Revit y vincular dinámicamente elementos de Revit a la geometría de Dynamo mientras realizamos operaciones paramétricas.



La biblioteca de Revit de la interfaz de usuario ofrece la categoría "Selection" (selección) que permite elegir varias formas de seleccionar la geometría.

Para seleccionar elementos de Revit correctamente, es importante comprender totalmente la jerarquía de elementos de Revit. ¿Desea seleccionar todos los muros de un proyecto? Seleccione por categoría. ¿Desea seleccionar todas las sillas Eames en su lobby moderno de mediados de siglo? Seleccione por familia. Antes de pasar a un ejercicio, vamos a hacer una revisión rápida de la jerarquía de Revit.

Jerarquía de Revit

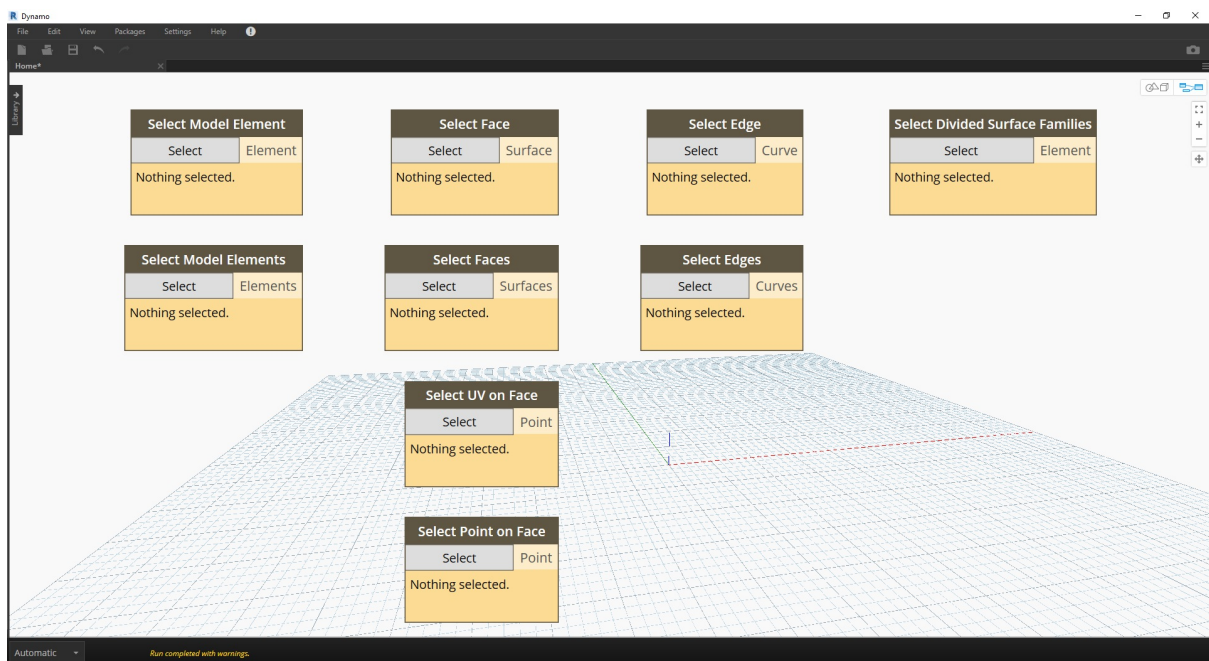


¿Recuerda la taxonomía de la biología? Reino, phylum, clase, orden, familia, género y especie. Los elementos de Revit se clasifican de forma similar. En un nivel básico, la jerarquía de Revit se puede dividir en categorías, familias, tipos* y ejemplares. Un ejemplar es un elemento de modelo individual (con un ID exclusivo), mientras que una categoría define un grupo genérico (como "muros" o "suelos"). Con la base de datos de Revit organizada de este modo, podemos seleccionar un elemento y elegir todos los elementos similares en función de un nivel especificado en la jerarquía.

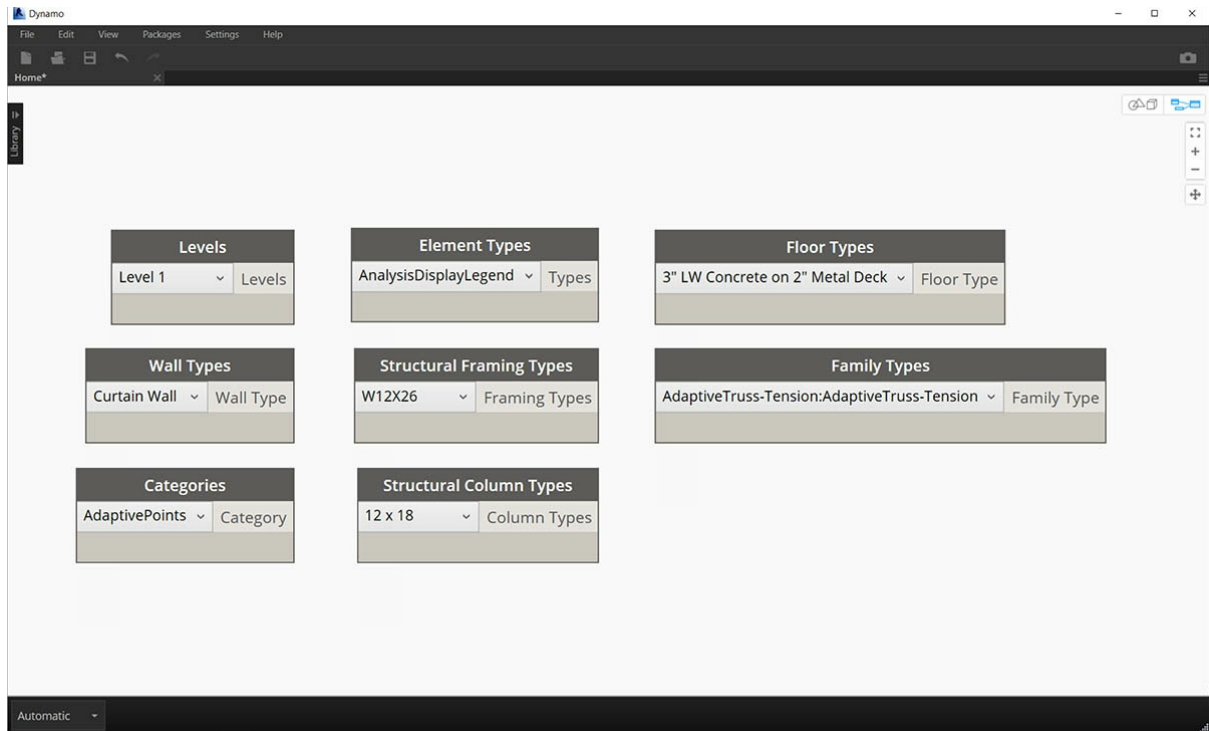
**Nota: los tipos en Revit se definen de forma distinta a los tipos en programación. En Revit, un tipo hace referencia a una ramificación de la jerarquía, en lugar de a un "tipo de datos".*

Navegación por las bases de datos con nodos de Dynamo

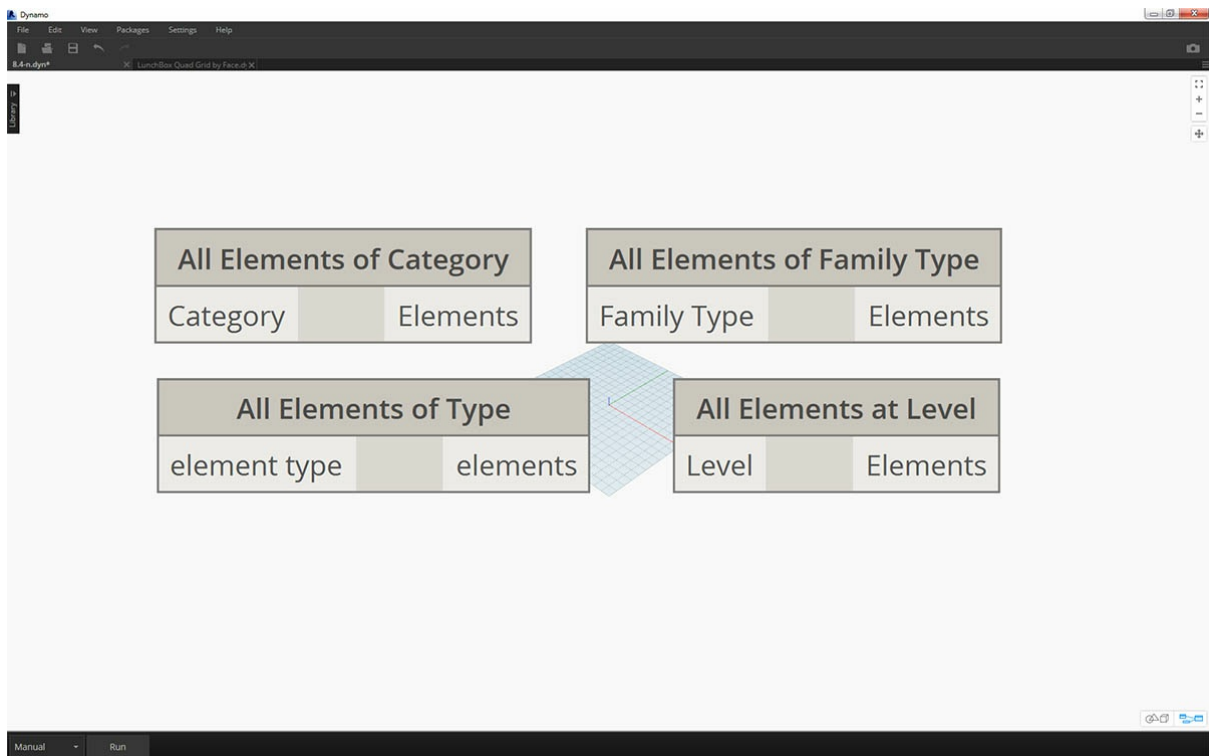
Las tres imágenes siguientes dividen las categorías principales para la selección de elementos de Revit en Dynamo. Son herramientas excelentes para usarlas en combinación y exploraremos algunas de ellas en los ejercicios siguientes.



Señalar y hacer clic es el método más sencillo para seleccionar directamente un elemento de Revit. Puede seleccionar un elemento de modelo completo o partes de su topología (como una cara o un borde). Este elemento permanece vinculado dinámicamente a ese objeto de Revit, por lo que, cuando el archivo de Revit actualiza su ubicación o parámetros, el elemento de Dynamo al que se hace referencia se actualiza en el gráfico.



Los *menús desplegables* crean una lista de todos los elementos a los que se puede acceder en un proyecto de Revit. Puede utilizar esta opción para hacer referencia a elementos de Revit que no son necesariamente visibles en una vista. Esta es una herramienta excelente para consultar elementos existentes o crear nuevos elementos en un editor de proyectos o familias de Revit.



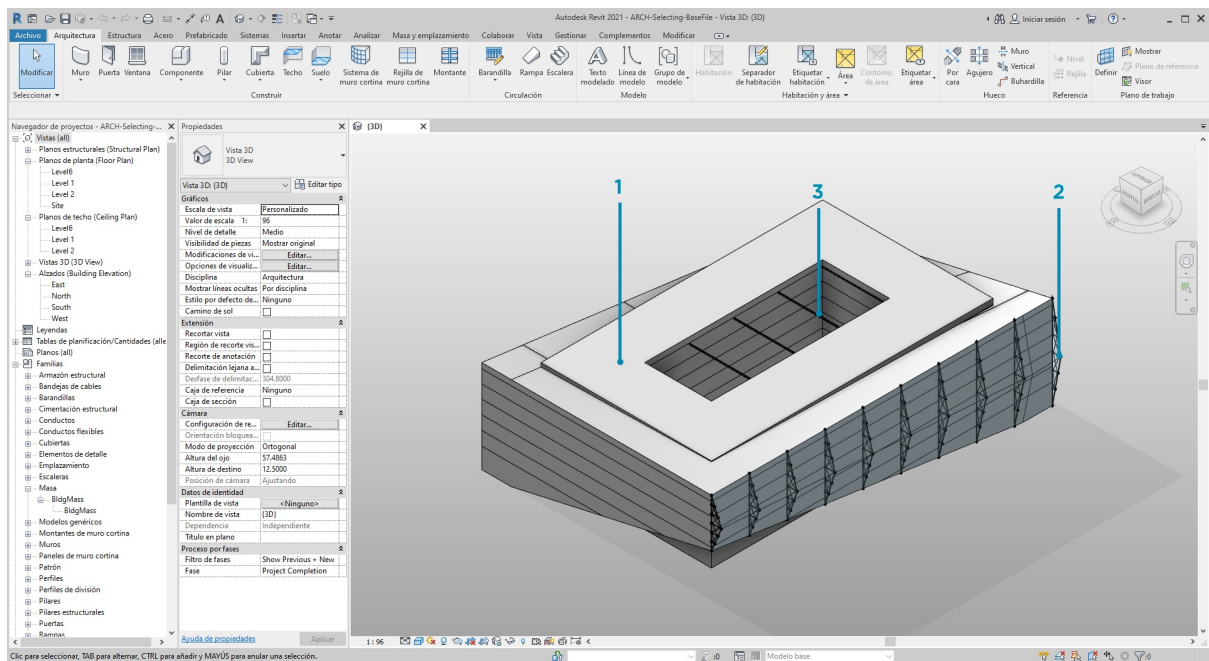
También puede seleccionar elementos de Revit por niveles específicos en la *jerarquía de Revit*. Esta es una opción eficaz para personalizar grandes matrices de datos para preparar la documentación o la creación de ejemplares y personalización generativas.

Teniendo en cuenta las tres imágenes anteriores, vamos a profundizar en un ejercicio que selecciona elementos de un proyecto básico de Revit para preparar las aplicaciones paramétricas que crearemos en las secciones restantes de este capítulo.

Ejercicio:

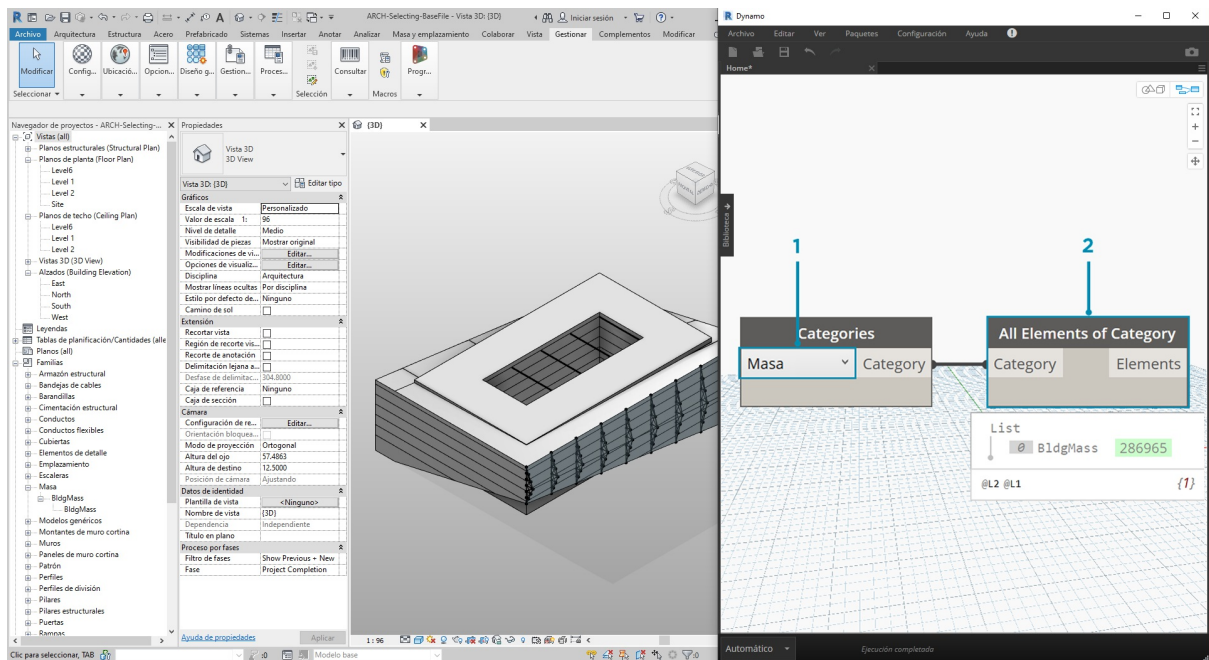
Descargue los archivos de ejemplo que acompañan a este ejercicio (haga clic con el botón derecho y seleccione "Guardar enlace como..."). En el Apéndice se incluye una lista completa de los archivos de ejemplo.

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



En este archivo de Revit de ejemplo, tenemos tres tipos de elementos de un edificio sencillo. Vamos a usar este archivo como ejemplo para seleccionar elementos de Revit en el contexto de la jerarquía de Revit:

1. Masa de construcción
2. Vigas de celosía (componentes adaptativos)
3. Vigas (armazón estructural)

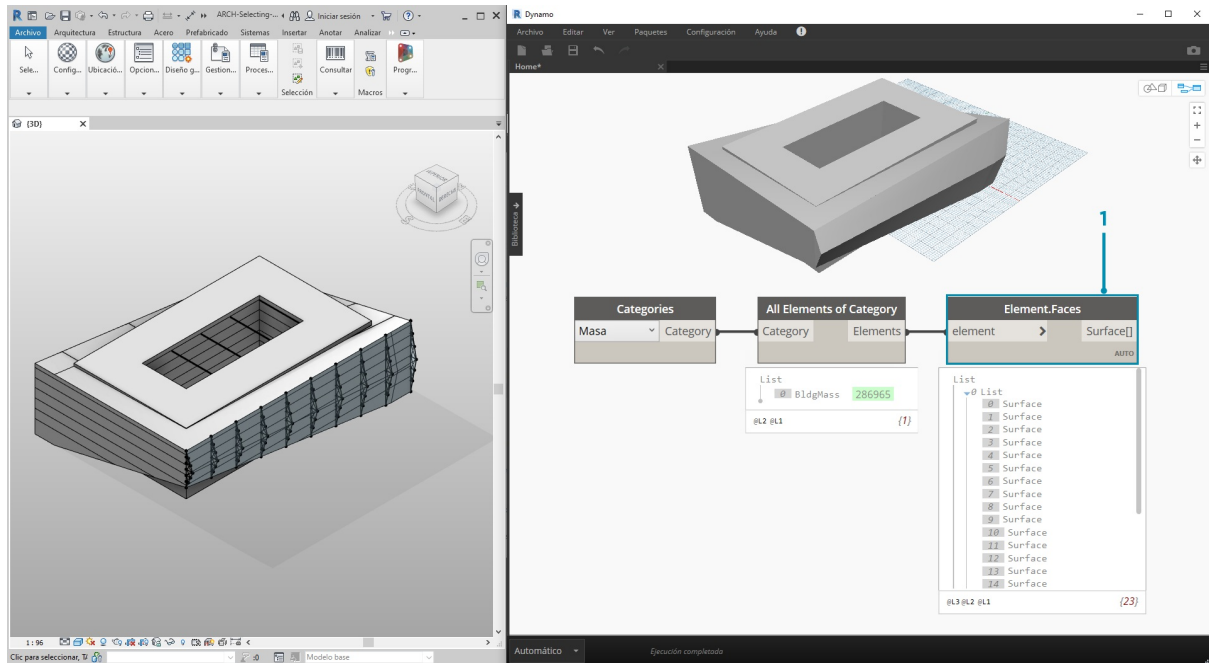


¿Qué conclusiones podemos sacar de los elementos que hay en la vista de proyecto de Revit? Y, ¿cuánto debemos descender en la jerarquía para seleccionar los elementos adecuados? Por supuesto, esta tarea se volverá más compleja al trabajar en un proyecto de gran tamaño. Hay una gran cantidad de opciones disponibles; se pueden seleccionar elementos por categorías, niveles, familias, ejemplares, etc.

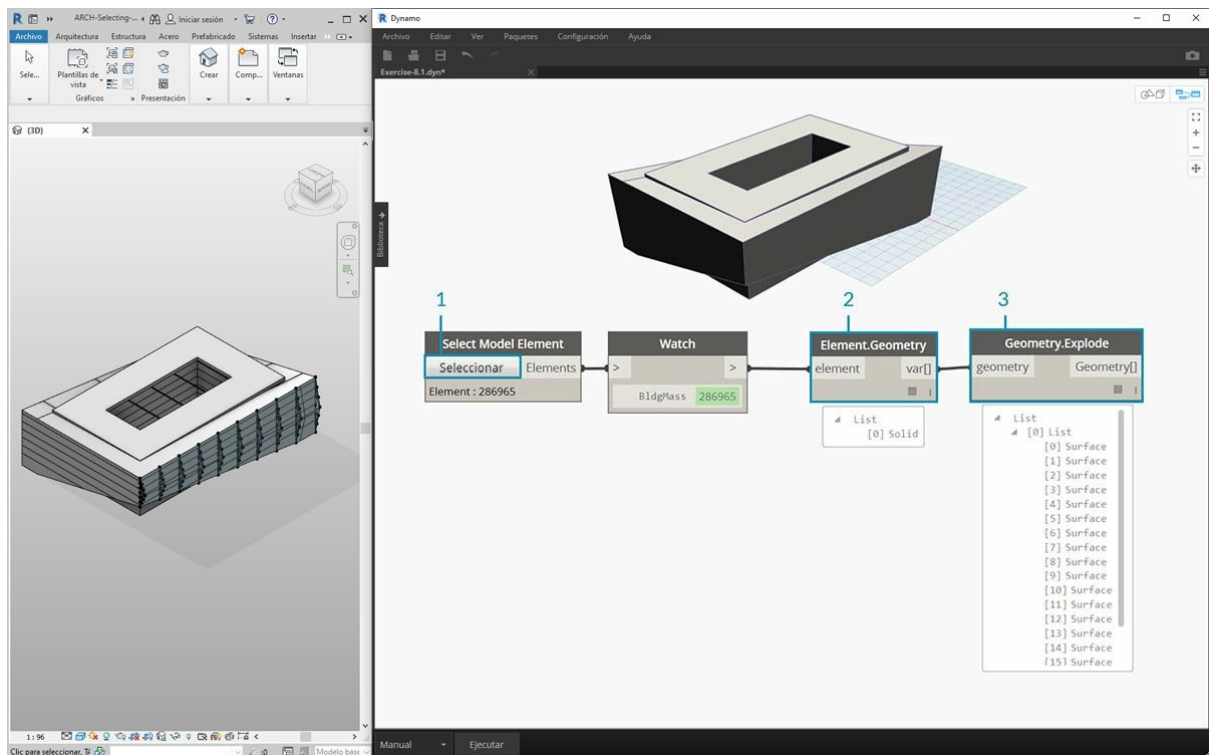
1. Como estamos trabajando con una configuración básica, seleccionaremos la masa de construcción eligiendo "Mass" en el nodo desplegable Categories. Puede encontrarlo en la ficha Revit > Selection.
2. La salida de la categoría Mass es solo la categoría en sí. Debemos seleccionar los elementos. Para ello, se utiliza el nodo "All Elements of Category".

En este punto, observe que no se ve ninguna geometría en Dynamo. Hemos seleccionado un elemento de Revit, pero no lo hemos convertido en geometría de Dynamo. Se trata de una distinción importante. Si tuviéramos que seleccionar un gran número de elementos, no es deseable obtener una vista preliminar de todos ellos en Dynamo porque esto ralentizaría todo el proceso. Dynamo es una herramienta para gestionar un proyecto de Revit sin realizar necesariamente operaciones de geometría, y lo veremos en la siguiente sección de este capítulo.

En este caso trabajamos con geometría sencilla, por lo que vamos a incorporar la geometría a la vista preliminar de Dynamo. El elemento "BldgMass" del nodo Watch anterior tiene un número verde* junto a él. Esto representa el ID del elemento e indica que estamos trabajando con un elemento de Revit, no con una geometría de Dynamo. El siguiente paso es convertir este elemento de Revit en geometría en Dynamo.

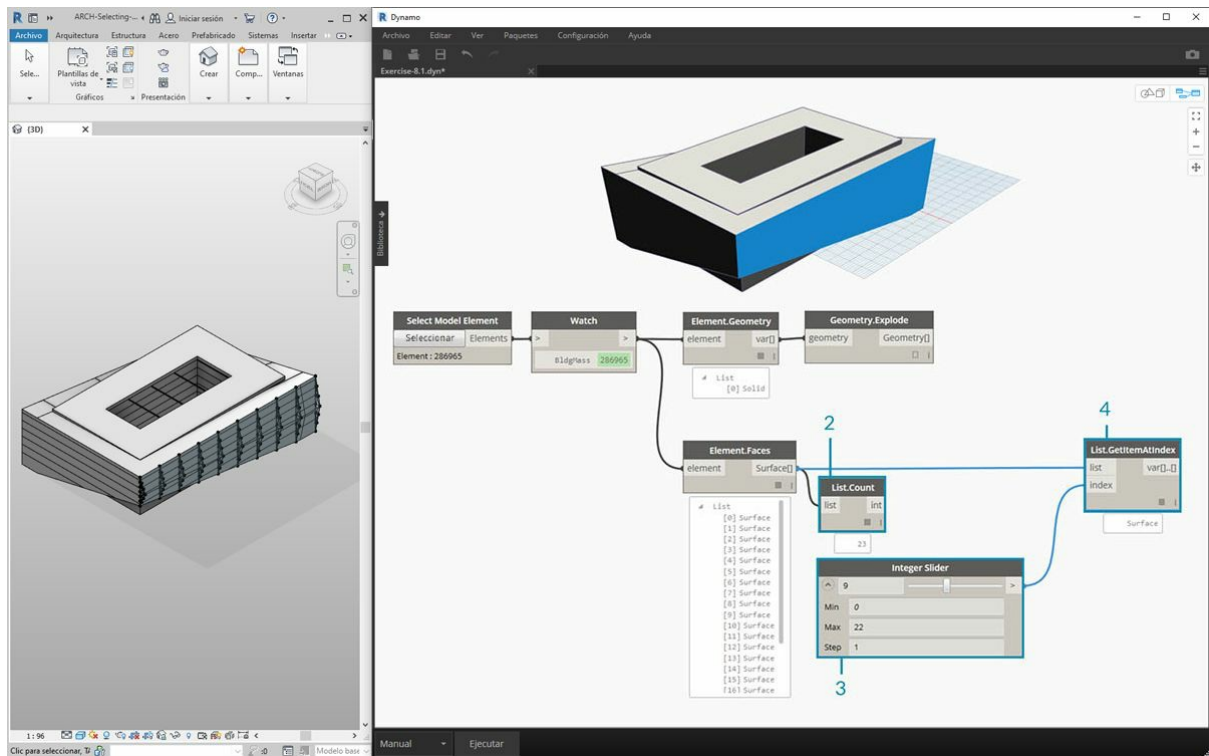


1. Mediante el nodo *Element.Faces*, se obtiene una lista de las superficies que representan cada cara de la masa de Revit. Ahora podemos ver la geometría en la ventana gráfica de Dynamo y comenzar a hacer referencia a la cara para realizar operaciones paramétricas.

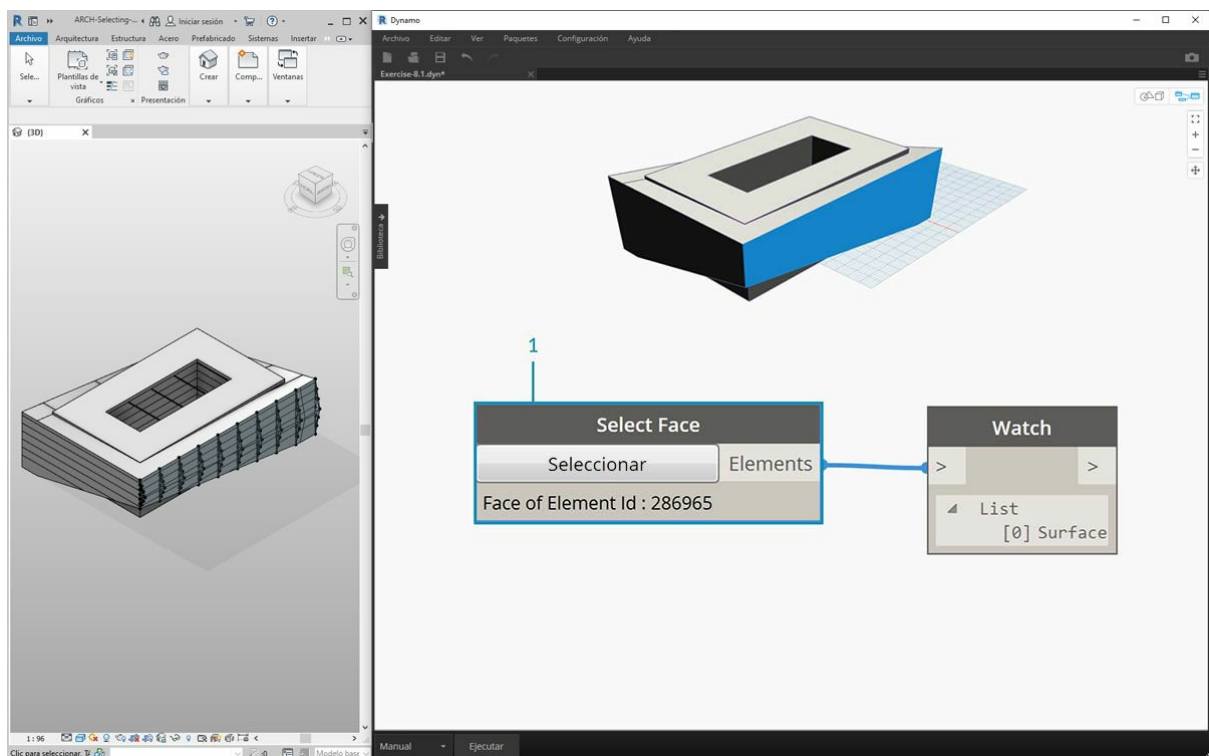


A continuación se incluye un método alternativo. En este caso, no vamos a realizar la selección mediante la jerarquía de Revit ("All Elements of Category"), sino que vamos a seleccionar de forma explícita la geometría en Revit.

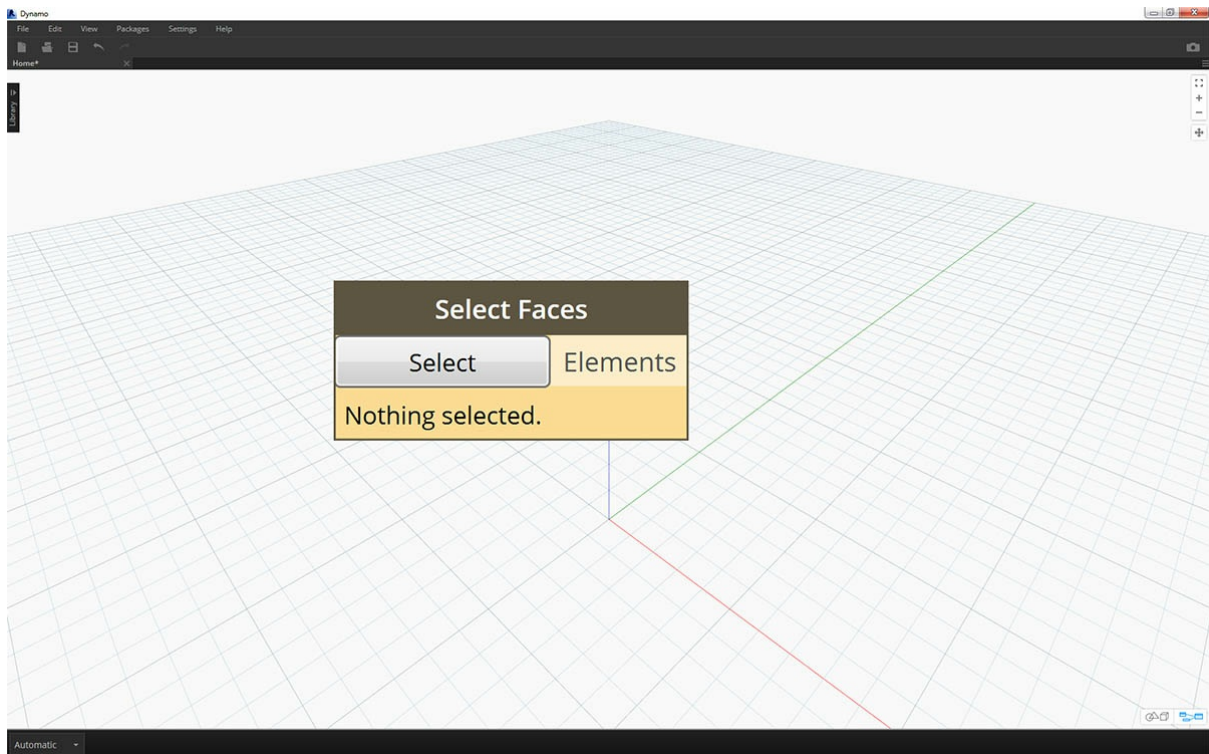
1. Mediante el nodo *Select Model Element*, haga clic en el botón "select" (o "change"). En la ventana gráfica de Revit, seleccione el elemento que desee. En este caso, seleccionamos la masa de construcción.
2. En lugar de *Element.Faces*, se puede seleccionar la masa completa como una geometría sólida mediante *Element.Geometry*. De este modo, se selecciona toda la geometría contenida en la masa.
3. Mediante *Geometry.Explode*, podemos obtener la lista de superficies de nuevo. Estos dos nodos funcionan de la misma forma que *Element.Faces*, pero ofrecen opciones alternativas para profundizar en la geometría de un elemento de Revit.



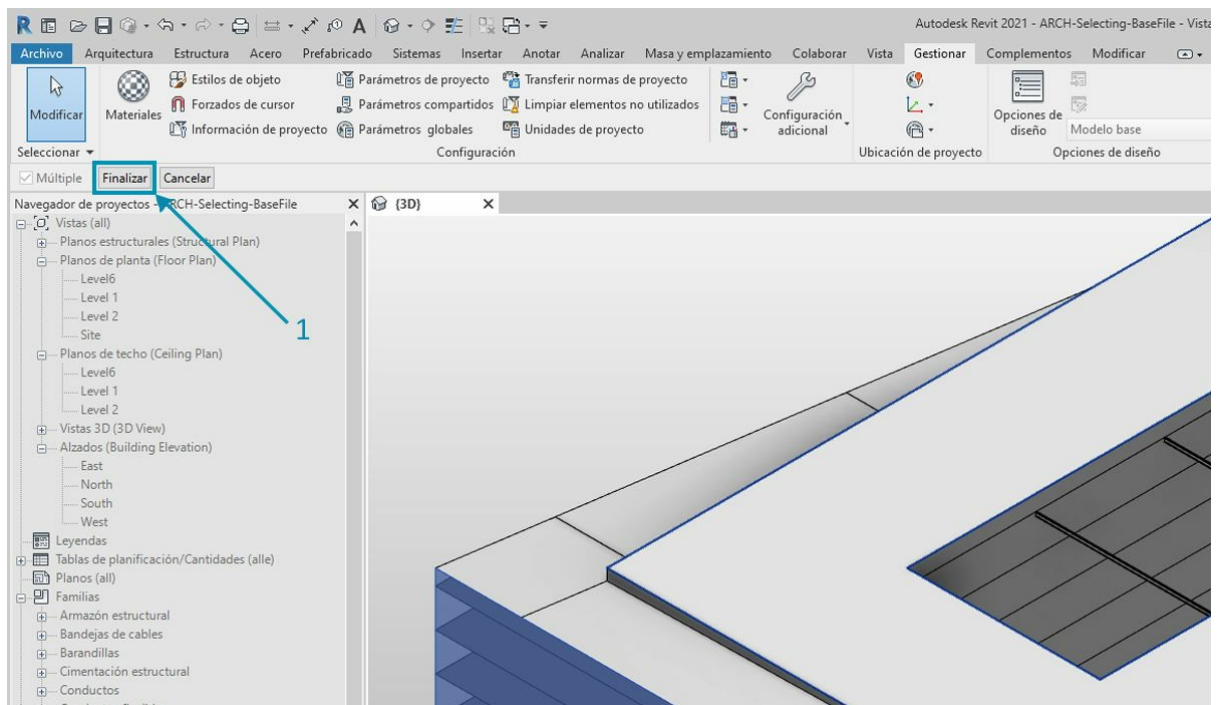
1. Mediante algunas operaciones de lista básicas, podemos consultar una cara de interés.
2. En primer lugar, el nodo *List.Count* indica que estamos trabajando con 23 superficies en la masa.
3. Al hacer referencia a este número, se cambia el valor máximo de un *control deslizante de enteros* a "22".
4. Con *List.GetItemAtIndex*, se introducen las listas y el *control deslizante de enteros* para el índice. Deslizamos la selección y nos detenemos cuando lleguemos al índice 9 y hayamos aislado la fachada principal que contiene las vigas de celosía.



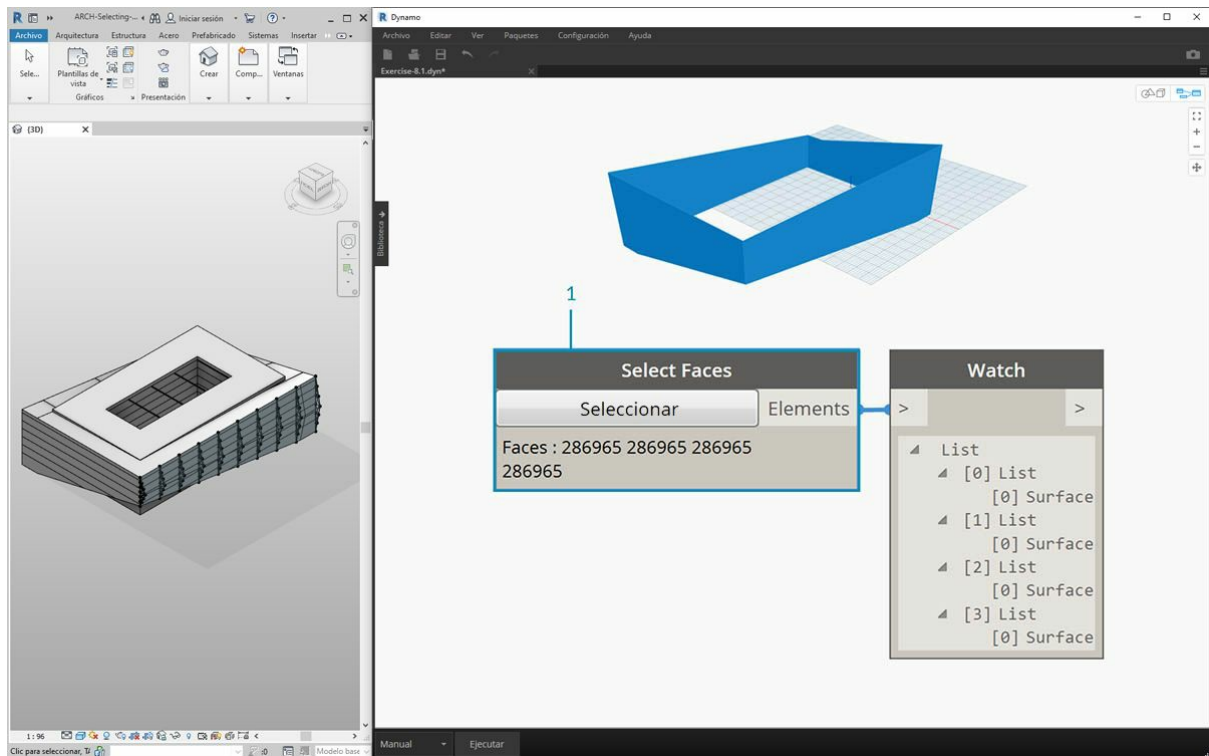
1. El paso anterior era un poco engorroso. Esto se puede hacer mucho más rápido con el nodo "Select Face". Este nos permite aislar una cara que no es un elemento en el proyecto de Revit. Se aplica la misma interacción que con "Select Model Element", excepto que seleccionamos la superficie en lugar del elemento completo.



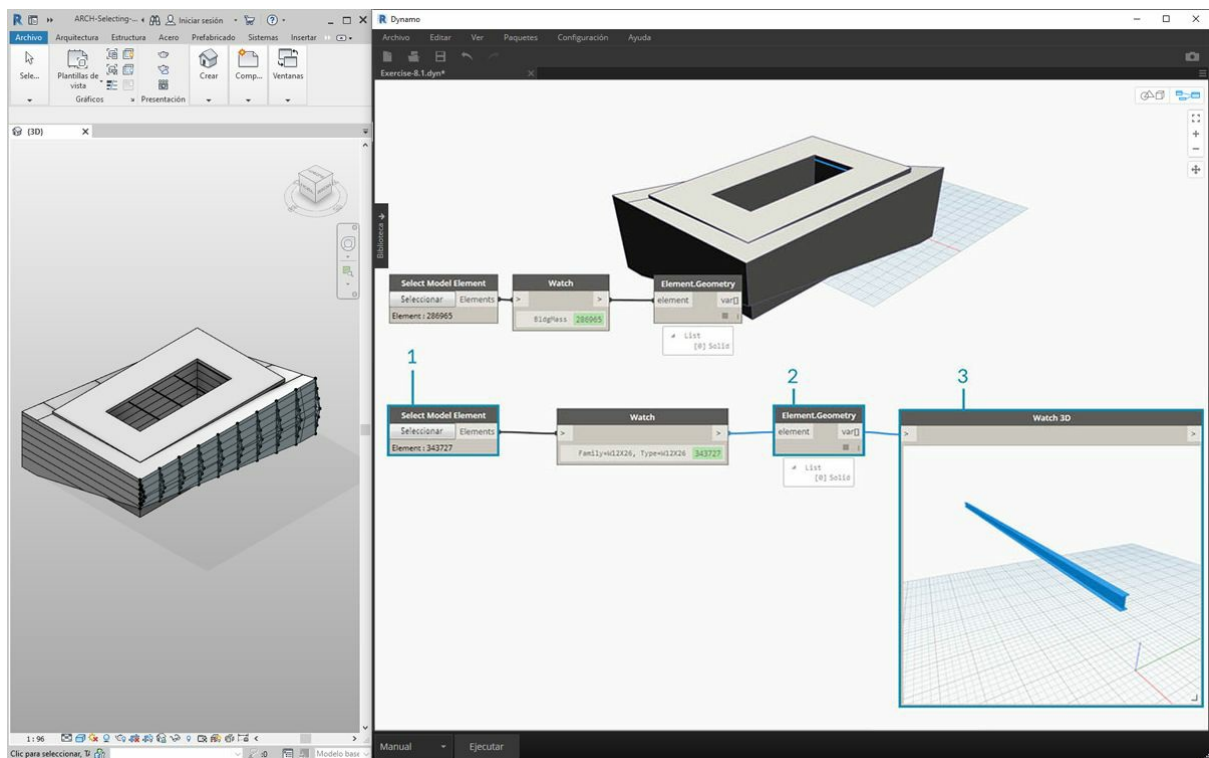
Supongamos que deseamos aislar los muros de las fachadas principales del edificio. Para ello, se puede utilizar el nodo "Select Faces". Haga clic en el botón "Select" y, a continuación, seleccione las cuatro fachadas principales en Revit.



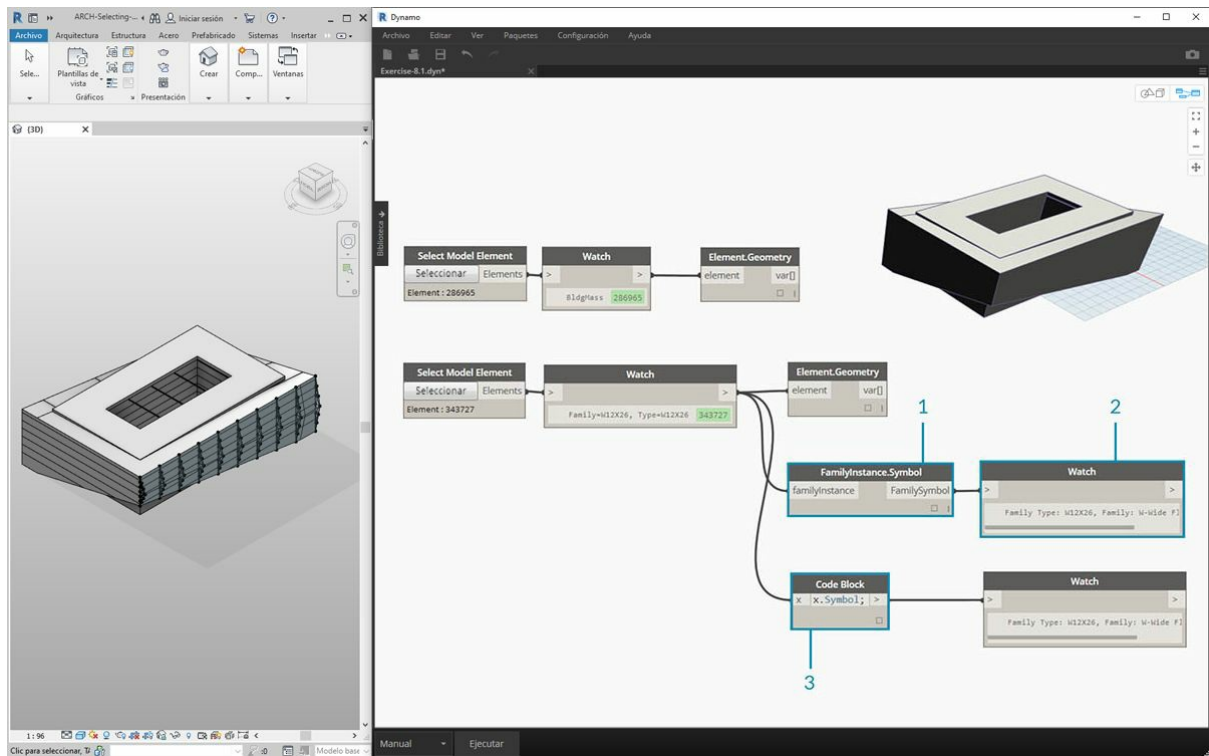
1. Después de seleccionar los cuatro muros, asegúrese de hacer clic en el botón "Finalizar" en Revit.



1. Las caras se importan en Dynamo como superficies.



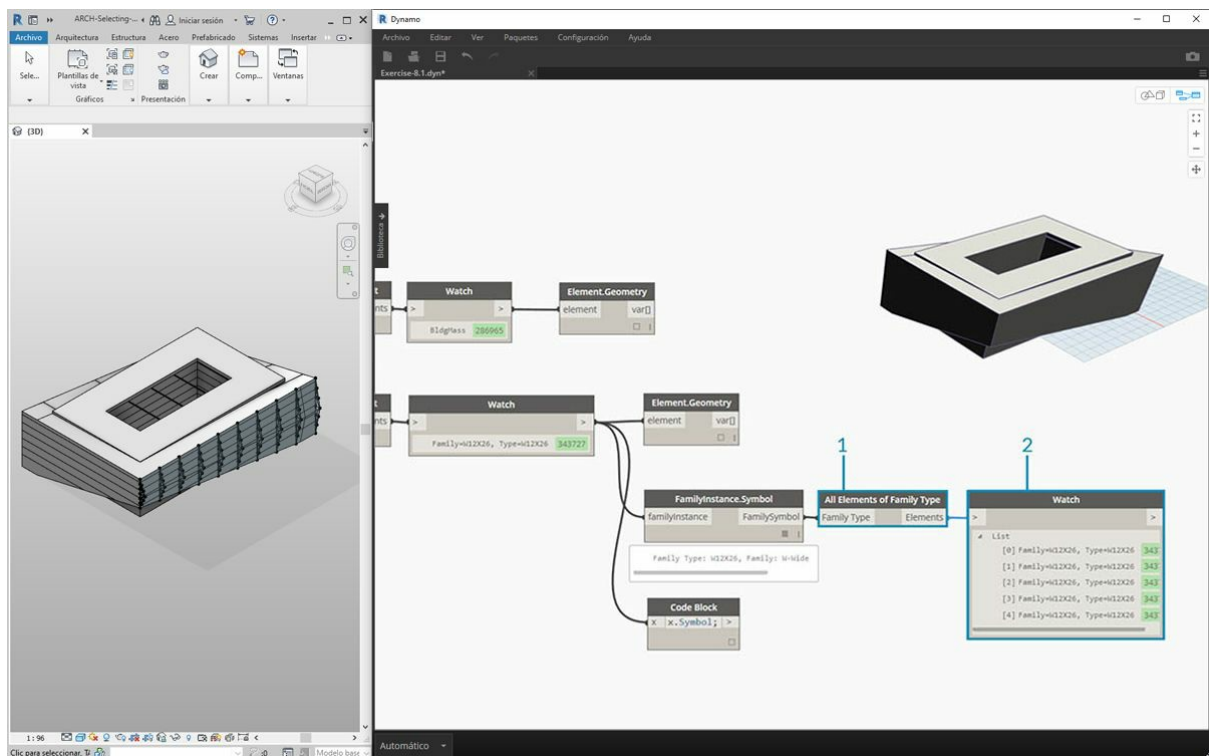
1. Ahora, veamos las vigas situadas sobre el atrio. Con el nodo "Select Model Element", seleccione una de las vigas.
2. Conecte el elemento de viga al nodo *Element.Geometry* y la viga aparecerá en la ventana gráfica de Dynamo.
3. Podemos ampliar la geometría con un nodo *Watch3D* (si no ve la viga en Watch 3D, haga clic con el botón derecho y pulse "Ajustar en ventana").



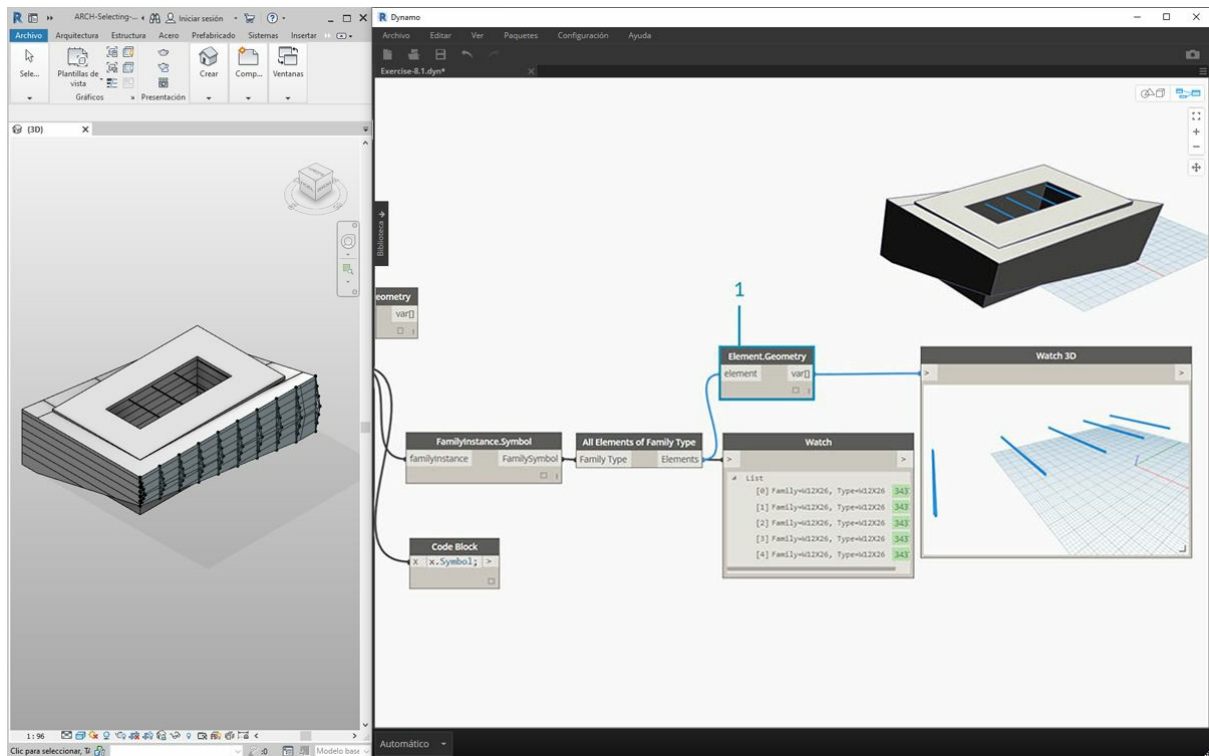
Una duda que puede aparecer con frecuencia en los flujos de trabajo de Revit o Dynamo es la siguiente: ¿cómo selecciono un elemento y obtengo todos los elementos similares? Como el elemento de Revit seleccionado contiene toda su información jerárquica, podemos consultar su tipo de familia y seleccionar todos los elementos de ese tipo.

1. Conecte el elemento de viga a un nodo *FamilyInstance.Symbol**
2. El nodo *Watch* indica que la salida es ahora un símbolo de familia en lugar de un elemento de Revit.
3. *FamilyInstance.Symbol* es una consulta sencilla, por lo que podemos realizarla en el bloque de código de la misma forma con *x.Symbol1*; y obtener los mismos resultados.

*Nota: símbolo de familia es la terminología de la API de Revit para el tipo de familia. Dado que esto puede crear confusión, se actualizará en las próximas versiones.



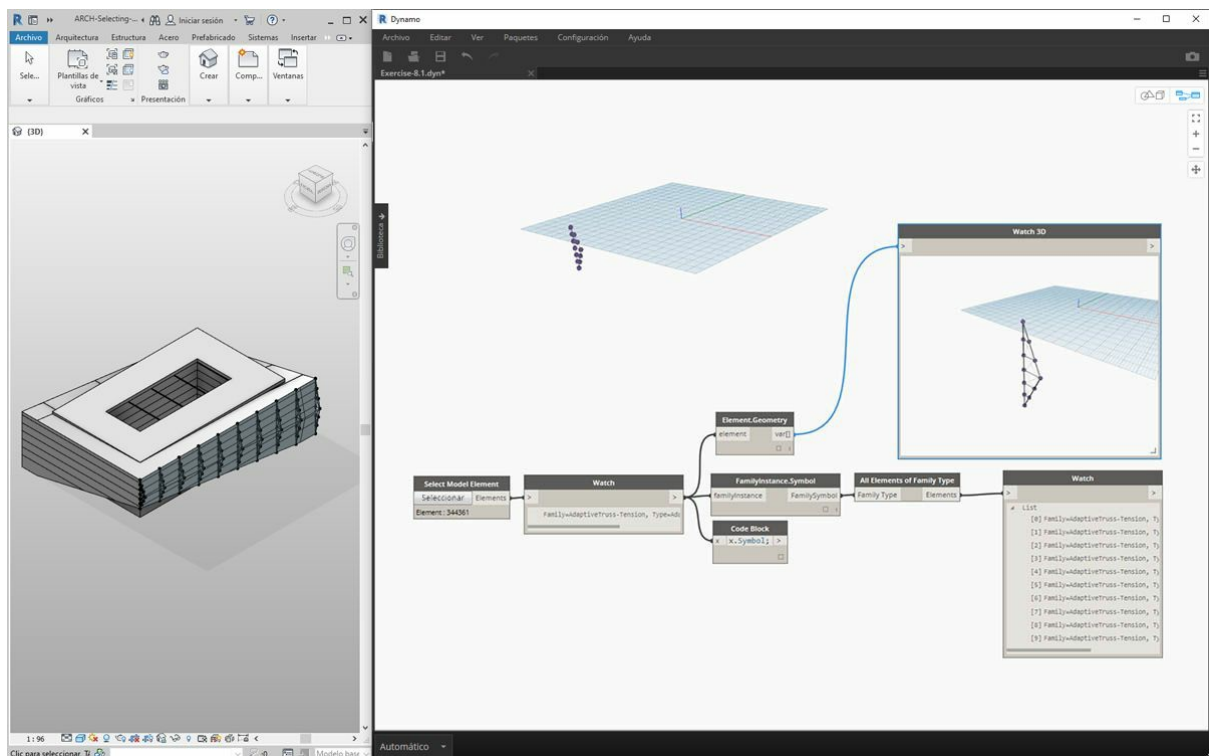
1. Para seleccionar las vigas restantes, utilizaremos el nodo "All Elements of Family Type".
2. El nodo Watch muestra que se han seleccionado cinco elementos de Revit.



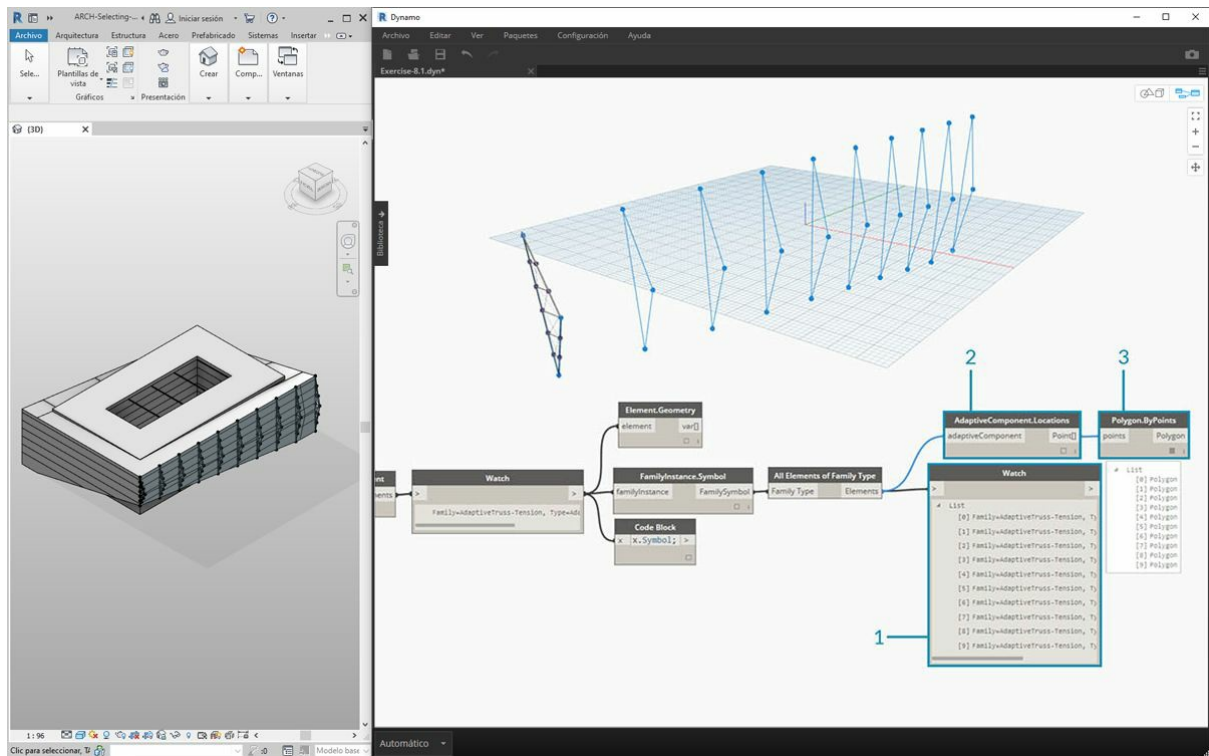
1. También podemos convertir estos cinco elementos en geometría de Dynamo.

¿Y si tuviéramos 500 vigas? Convertir todos estos elementos en geometría de Dynamo sería muy lento. Si Dynamo tarda mucho tiempo en calcular los nodos, puede usar la función "Bloquear" para poner en pausa la ejecución de operaciones de Revit mientras desarrolla el gráfico. Para obtener más información sobre el bloqueo de nodos, consulte la sección "Bloqueo" del [capítulo de sólidos](#).

En cualquier caso, si importamos 500 vigas, ¿necesitamos que todas las superficies realicen la operación paramétrica deseada? ¿O podemos extraer información básica de las vigas y realizar tareas generativas con geometría fundamental? Esta es una pregunta que tendremos en cuenta a medida que avanzamos en este capítulo. Por ejemplo, veamos el sistema de vigas de celosía:



Con el mismo gráfico de nodos, seleccione el elemento de viga de celosía en lugar del elemento de viga. Antes de hacerlo, suprima el nodo Element.Geometry del paso anterior.



1. En el nodo *Watch*, podemos ver que tenemos una lista de componentes adaptativos seleccionados en Revit. Vamos a extraer la información básica, por lo que comenzaremos con los puntos adaptativos.
2. Conecte el nodo *"All Elements of Family Type"* al nodo *"AdaptiveComponent.Location"*. Esto nos proporciona una lista de listas, cada una con tres puntos que representan las ubicaciones de los puntos adaptativos.
3. Al conectar un nodo *"Polygon.ByPoints"*, se obtiene una PolyCurve. Se puede ver en la ventana gráfica de Dynamo. Con este método, hemos visualizado la geometría de un elemento y hemos abstraído la geometría de la matriz de elementos restantes (que podría ser mayor en número que este ejemplo).

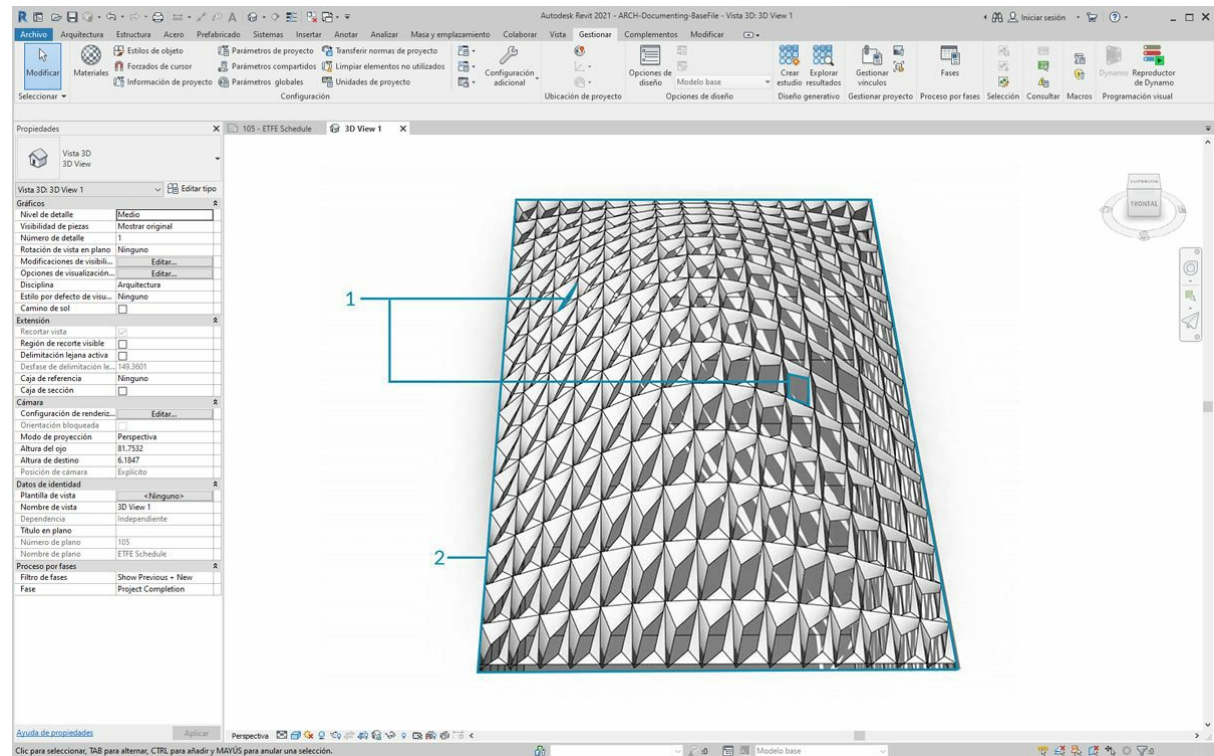
**Sugerencia: si hace clic en el número verde de un elemento de Revit en Dynamo, la ventana gráfica de Revit ampliará dicho elemento.*

Edición

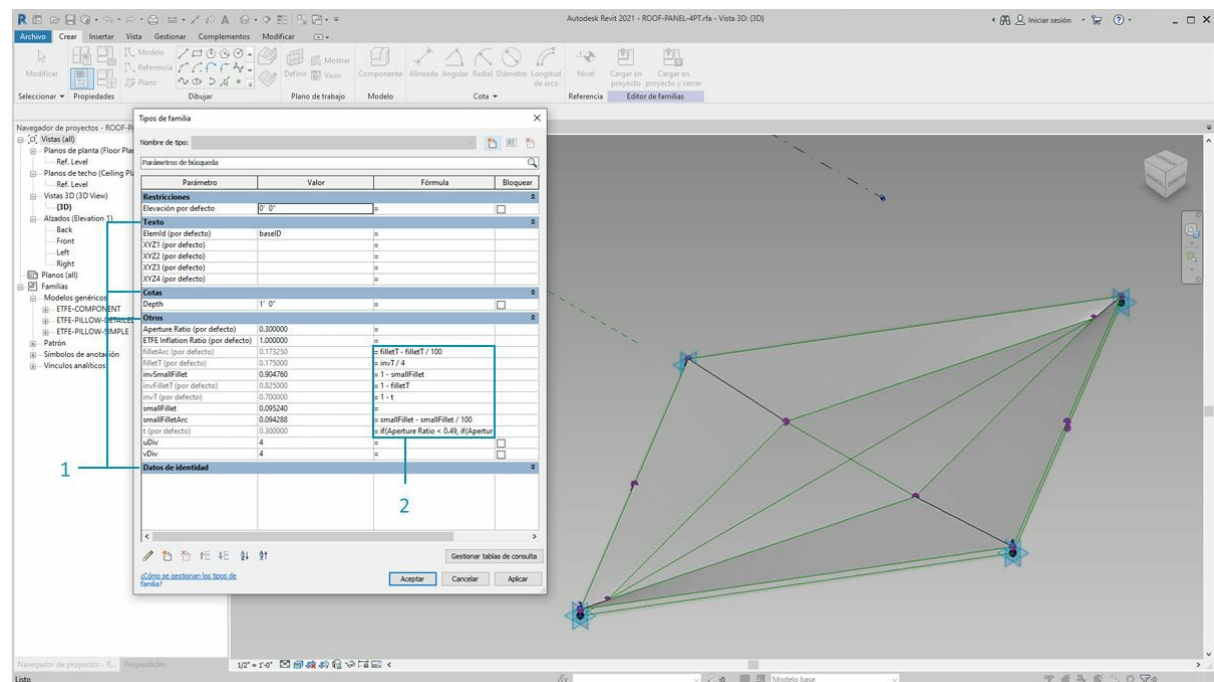
Edición

Una característica útil de Dynamo es la posibilidad de editar parámetros en un nivel paramétrico. Por ejemplo, se puede utilizar un algoritmo generativo o los resultados de una simulación para controlar los parámetros de una matriz de elementos. De este modo, un conjunto de ejemplares de la misma familia puede tener propiedades personalizadas en el proyecto de Revit.

Parámetros de tipo y ejemplar



1. Los parámetros de ejemplar definen la apertura de los paneles en la superficie de la cubierta, desde una proporción de apertura de 0,1 a 0,4.
2. Los parámetros basados en tipo se aplican a todos los elementos de la superficie porque son del mismo tipo de familia. El material de cada panel, por ejemplo, puede controlarse mediante un parámetro basado en tipo.



1. Si ha configurado una familia de Revit con anterioridad, recuerde que debe asignar un tipo de parámetro (cadena, número, cota, etc.).

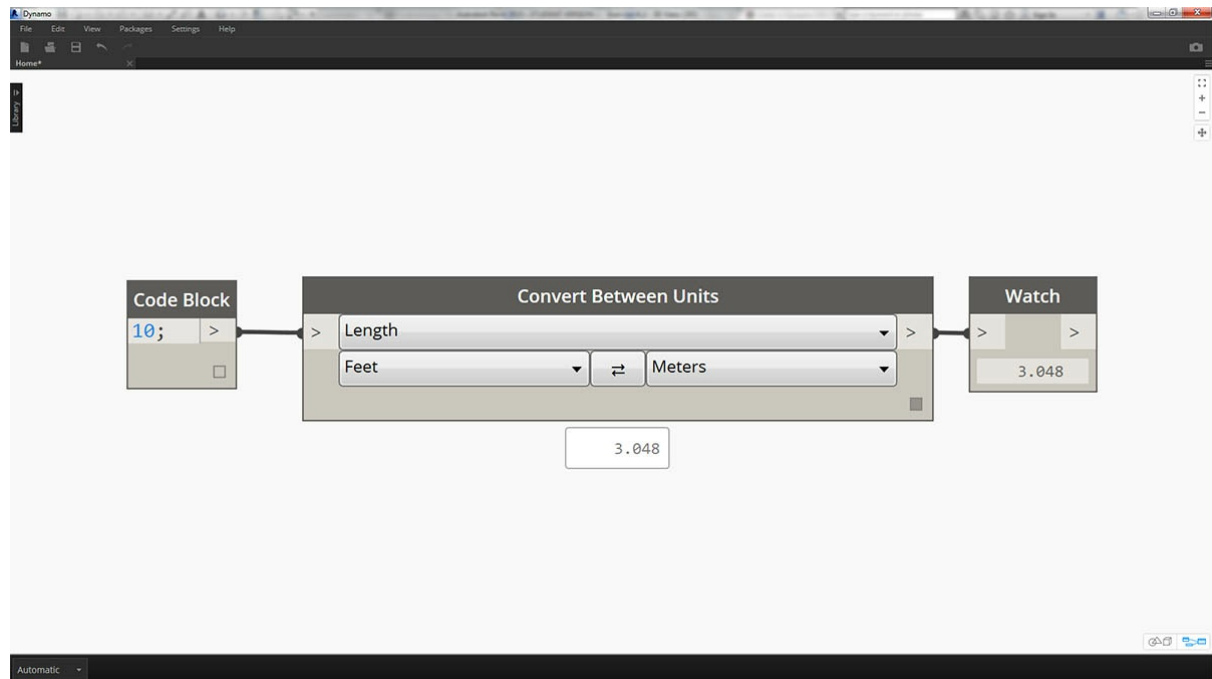
- Asegúrese de utilizar el tipo de datos correcto al asignar parámetros de Dynamo.
2. También puede utilizar Dynamo en combinación con restricciones paramétricas definidas en las propiedades de una familia de Revit.

Como revisión rápida de los parámetros en Revit, recordemos que existen parámetros de tipo y parámetros de ejemplar. Ambos se pueden editar en Dynamo, pero trabajaremos con parámetros de ejemplar en el ejercicio siguiente.

Nota: a medida que descubra la amplia aplicación de la edición de parámetros, puede que desee editar una gran cantidad de elementos en Revit con Dynamo. Esta puede ser una operación *computacionalmente exigente*, lo que significa que puede ser lenta. Si va a editar un gran número de elementos, puede usar la función de nodo "Bloquear" para poner en pausa la ejecución de operaciones de Revit mientras desarrolla el gráfico. Para obtener más información sobre el bloqueo de nodos, consulte la sección "Bloqueo" del [capítulo de sólidos](#).

Unidades

A partir de la versión 0.8, Dynamo prácticamente no trabaja con unidades. Esto permite que Dynamo siga siendo un entorno de programación visual abstracto. Los nodos de Dynamo que interactúan con las cotas de Revit harán referencia a las unidades del proyecto de Revit. Por ejemplo, si configura un parámetro de longitud en Revit desde Dynamo, el número del valor en Dynamo se corresponderá con las unidades por defecto del proyecto de Revit. El siguiente ejercicio utiliza metros.



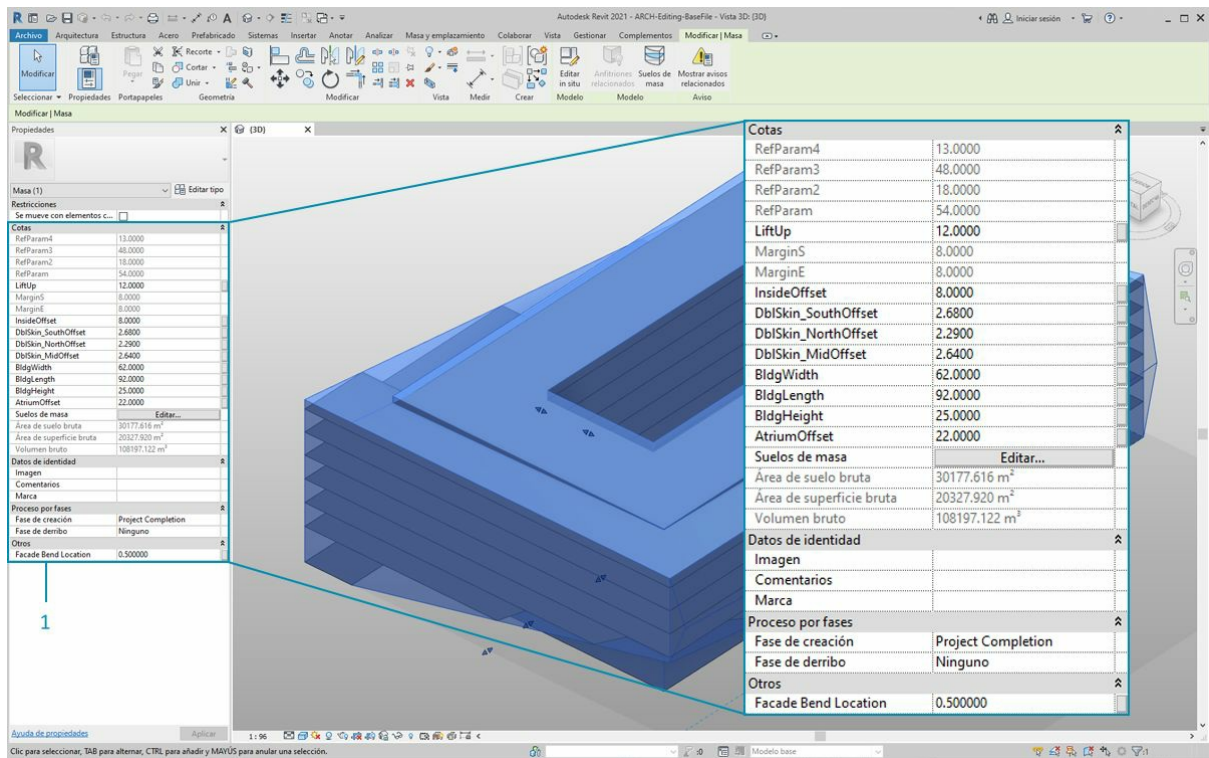
Para obtener una conversión rápida de unidades, utilice el nodo "*Convert Between Units*". Esta es una herramienta práctica para convertir unidades de longitud, área y volumen sobre la marcha.

Ejercicio

Descargue los archivos de ejemplo que acompañan a este ejercicio (haga clic con el botón derecho y seleccione "Guardar enlace como..."). En el Apéndice se incluye una lista completa de los archivos de ejemplo.

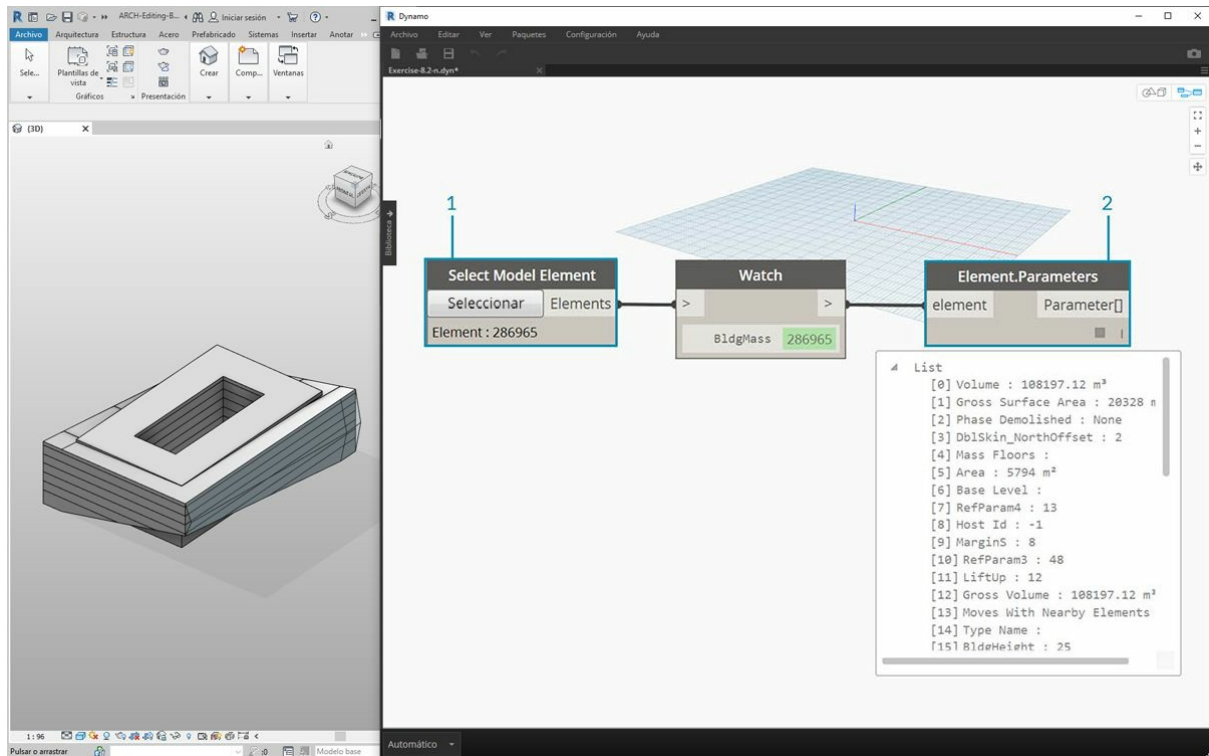
1. [Editing.dyn](#)
2. [ARCH-Editing-BaseFile.rvt](#)

Este ejercicio se centra en la edición de elementos de Revit sin realizar operaciones geométricas en Dynamo. No vamos a importar la geometría de Dynamo aquí, solo vamos a editar los parámetros de un proyecto de Revit. Este ejercicio es básico; para los usuarios más avanzados de Revit, observe que se trata de parámetros de ejemplar de una masa, pero la misma lógica se puede aplicar a una matriz de elementos para personalizarlos a gran escala. Esto se realiza con el nodo "Element.SetParameterByName".

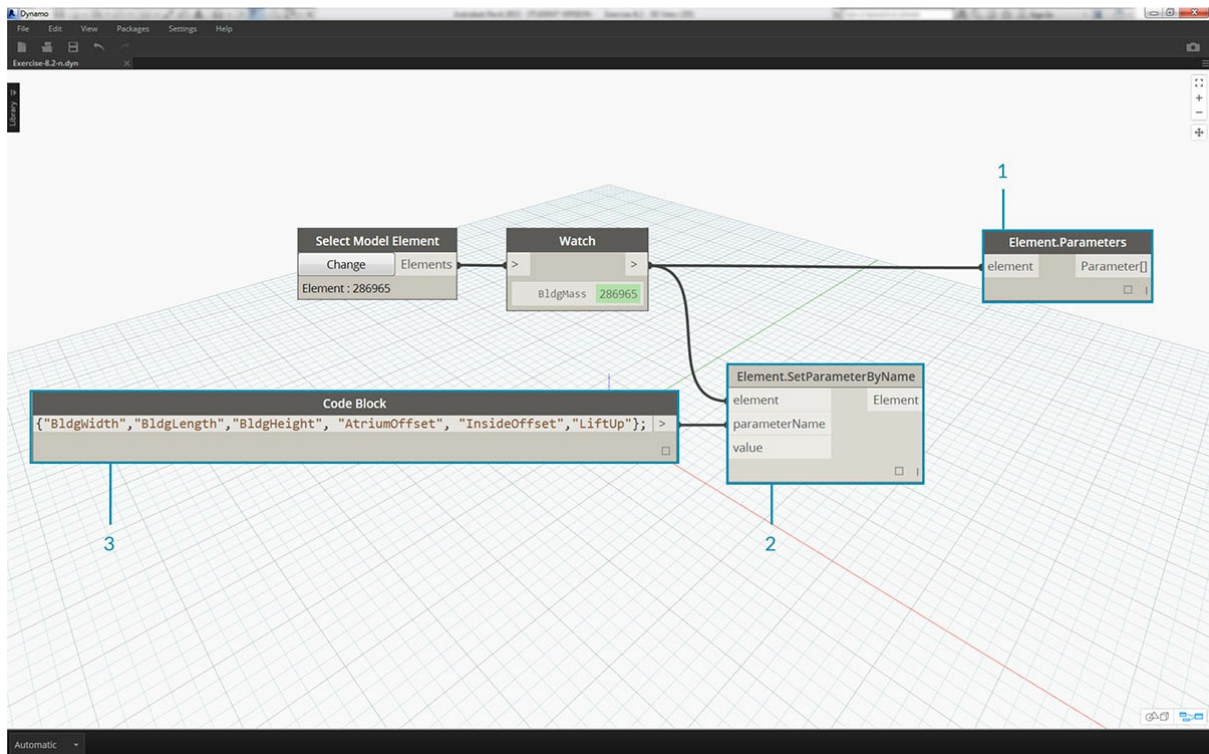


Empiece con el archivo de ejemplo de Revit para esta sección. Hemos eliminado los elementos estructurales y las vigas de celosía adaptativas de la sección anterior. En este ejercicio, nos centraremos en una manipulación paramétrica en Revit y en la manipulación en Dynamo.

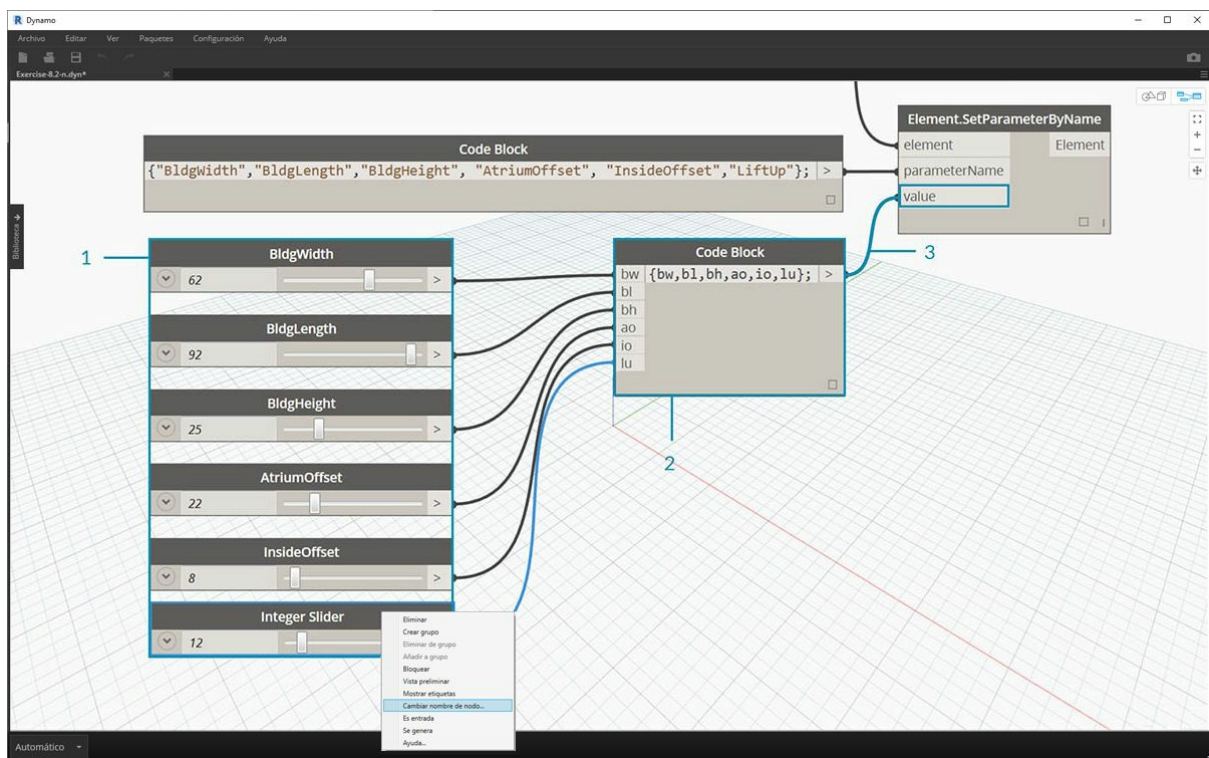
1. Al seleccionar el edificio dentro de Masa en Revit, vemos una matriz de parámetros de ejemplar en el panel Propiedades.



1. Seleccione la masa de construcción con el nodo *Select Model Element*.
2. Se pueden consultar todos los parámetros de esta masa mediante el nodo *"Element.Parameters"*. Esto incluye los parámetros de tipo y ejemplar.



1. Haga referencia al nodo *Element.Parameters* para buscar los parámetros objetivo. También podemos ver el panel Propiedades del paso anterior para elegir los nombres de parámetro que deseamos editar. En este caso, buscamos los parámetros que afectan a los grandes movimientos geométricos en la masa de construcción.
2. Realizaremos cambios en el elemento de Revit mediante el nodo *Element.SetParameterByName*.
3. Con el *bloque de código*, definimos una lista de estos parámetros utilizando comillas a ambos lados de cada elemento para formar una cadena. También se puede utilizar el nodo *List.Create* con una serie de nodos *String* conectados a varias entradas. El bloque de código es simplemente más rápido y sencillo. Asegúrese de que la cadena coincida con el nombre exacto de Revit, incluidas las mayúsculas y minúsculas: {\"BldgWidth\", \"BldgLength\", \"BldgHeight\", \"AtriumOffset\", \"InsideOffset\", \"LiftUp\"};.

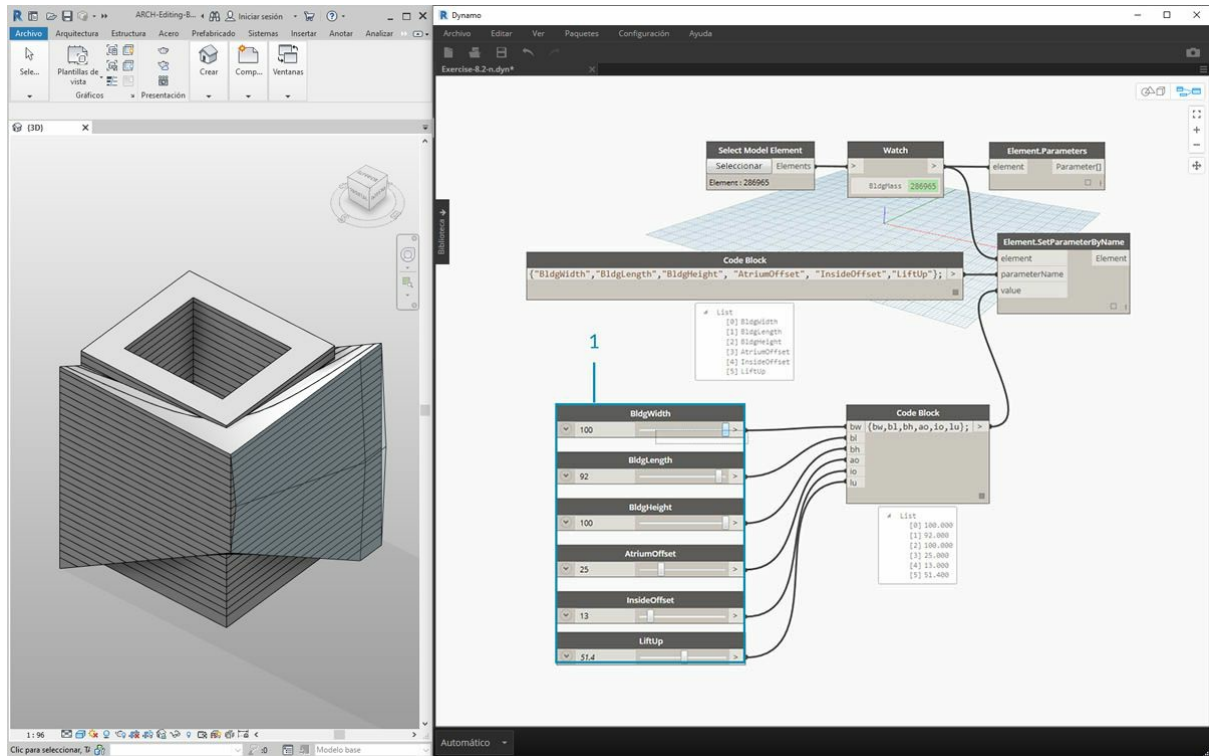


1. También vamos a designar valores para cada parámetro. Añada seis *controles deslizantes de enteros* al lienzo y cambie el nombre de estos al parámetro correspondiente de la lista. Asimismo, defina los valores de cada control deslizante a los que se pueden ver en la imagen anterior. Por orden de arriba a abajo: 62, 92, 25, 22, 8, 12.
2. Defina otro *bloque de código* con una lista de la misma longitud que los nombres de parámetro. En este caso, se asignan nombres a las variables (sin comillas) que crean entradas para el *bloque de código*. Conecte los *controles deslizantes* a cada entrada respectiva:

{bw, b1, bh, ao, io, lu};

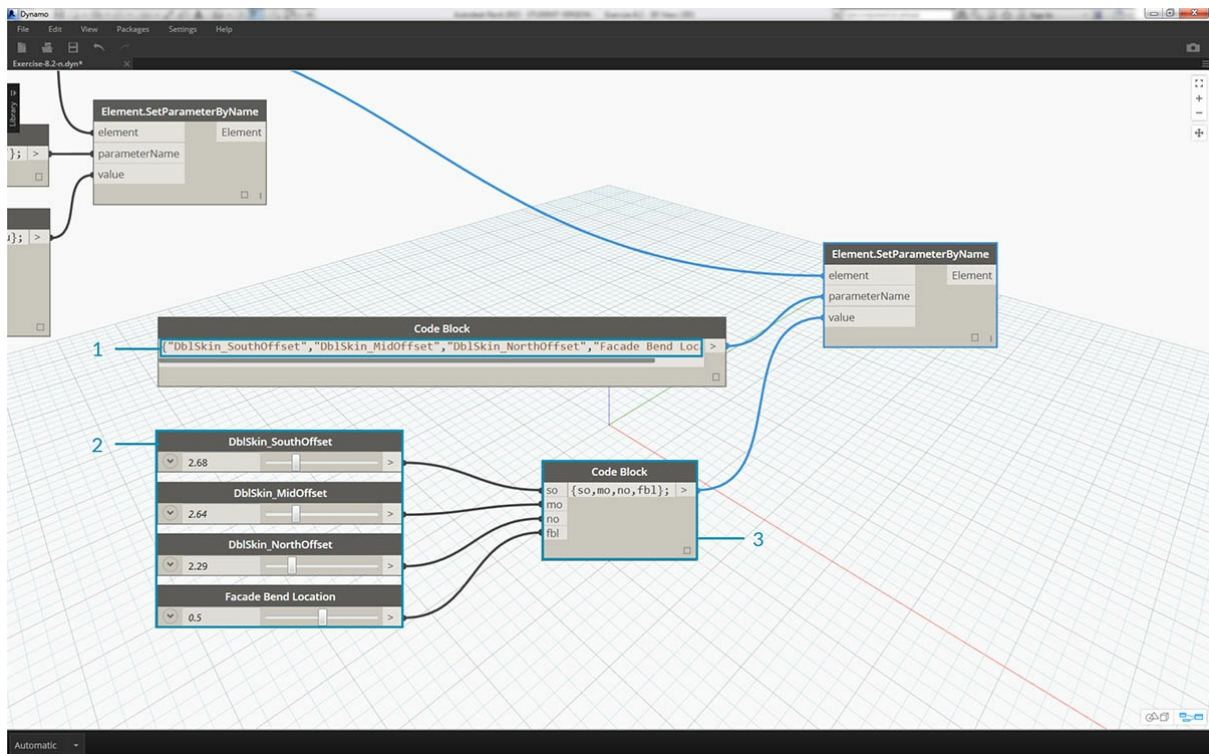
3. Conecte el bloque de código al nodo "Element.SetParameterByName"*. Con opción Ejecutar automáticamente seleccionada, veremos los resultados automáticamente.

*Nota: esta demostración funciona con parámetros de ejemplar, pero no con parámetros de tipo.



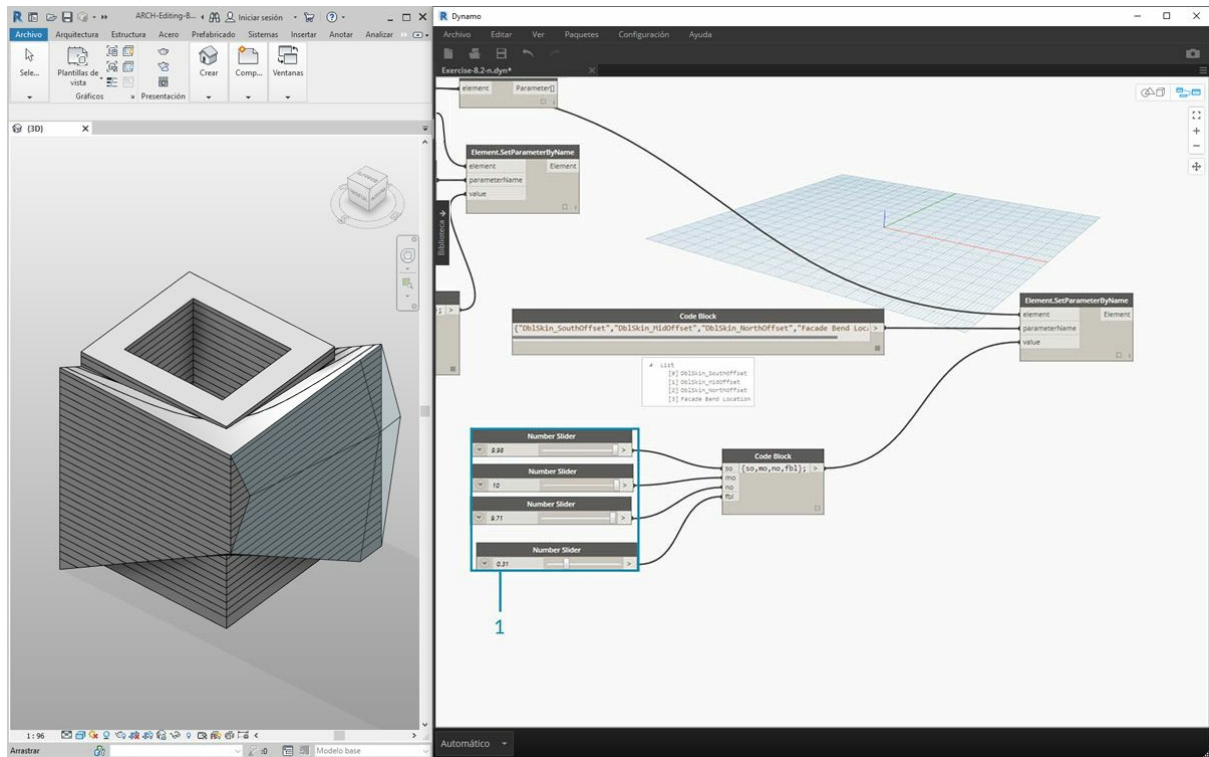
Al igual que en Revit, muchos de estos parámetros dependen unos de otros. Por supuesto, existen combinaciones en las que la geometría se puede partir. Podemos solucionar este problema con las fórmulas definidas en las propiedades de los parámetros, o bien podemos configurar una lógica similar con operaciones matemáticas en Dynamo (este es un reto adicional si desea ampliar el ejercicio).

1. Esta combinación proporciona un nuevo diseño con estilo a la masa de construcción: 100, 92, 100, 25, 13, 51.4.



1. Vamos a copiar el gráfico y centrarnos en la cristalería de la fachada que alojará el sistema de vigas de celosía. Aislamos cuatro parámetros en este caso: {"Db1Skin_SouthOffset", "Db1Skin_MidOffset", "Db1Skin_NorthOffset", "Facade Bend Location"};.

- Además, creamos *controles deslizantes de número* y les cambiamos el nombre por los parámetros adecuados. Los tres primeros controles deslizantes empezando por arriba deben reasignarse a un dominio de [0,10], mientras que el control deslizante final, "*Facade Bend Location*", se debe reasignar a un dominio de [0,1]. Estos elementos, de arriba a abajo, deben empezar con estos valores (aunque son arbitrarios): 2,68; 2,64; 2,29; 0,5.
- Define un nuevo *bloque de código* y conecta los controles deslizantes: {so, mo, no, fb1};.

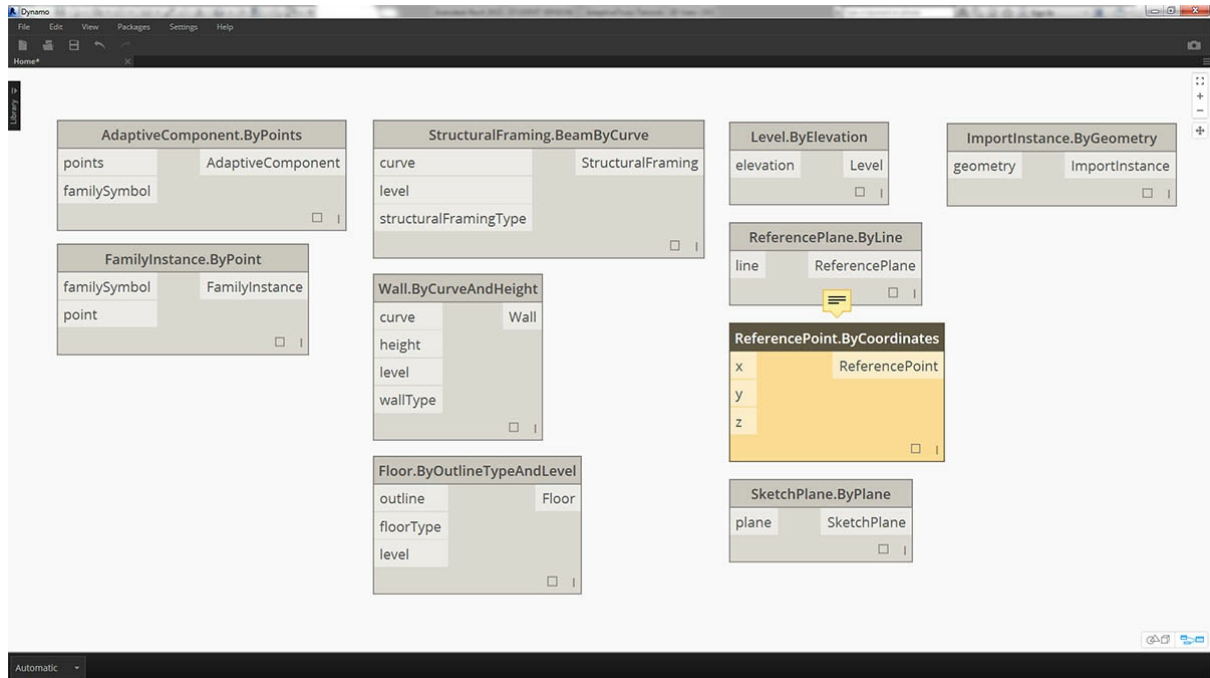


- Al cambiar los *controles deslizantes* en esta parte del gráfico, podemos hacer que la cristalería de la fachada sea mucho más sustancial: 9,98;10,0;9,71;0,31.

Creación

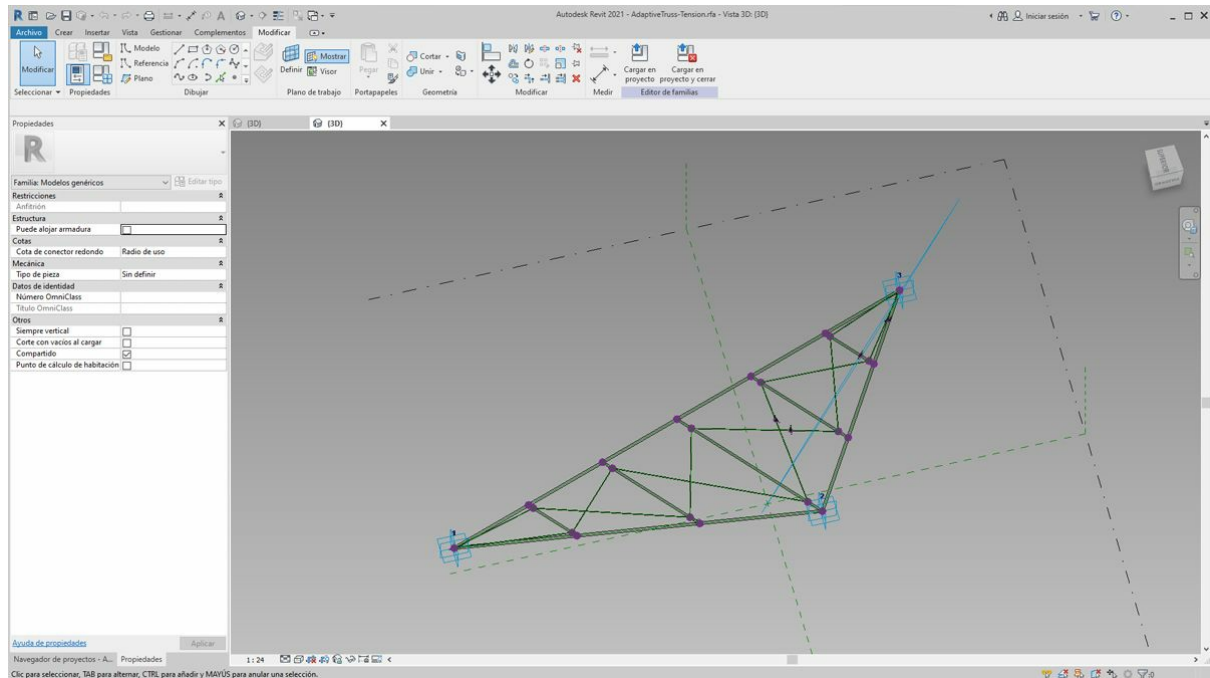
Creación

Puede crear una matriz de elementos de Revit en Dynamo con control paramétrico completo. Los nodos de Revit en Dynamo permiten importar elementos de geometrías genéricas a tipos de categorías específicos (como muros y suelos). En esta sección, nos centraremos en la importación paramétrica de elementos flexibles con componentes adaptativos.



Componentes adaptativos

Un componente adaptativo es una categoría de familia flexible que se presta bien a las aplicaciones generativas. Una vez creada la instancia, puede crear un elemento geométrico complejo controlado por la ubicación fundamental de los puntos adaptativos.



Un ejemplo de un componente adaptativo de tres puntos en el Editor de familias. Esto genera una viga de celosía definida por la posición de cada punto adaptativo. En el ejercicio siguiente, utilizaremos este componente para generar una serie de vigas de celosía en una fachada.

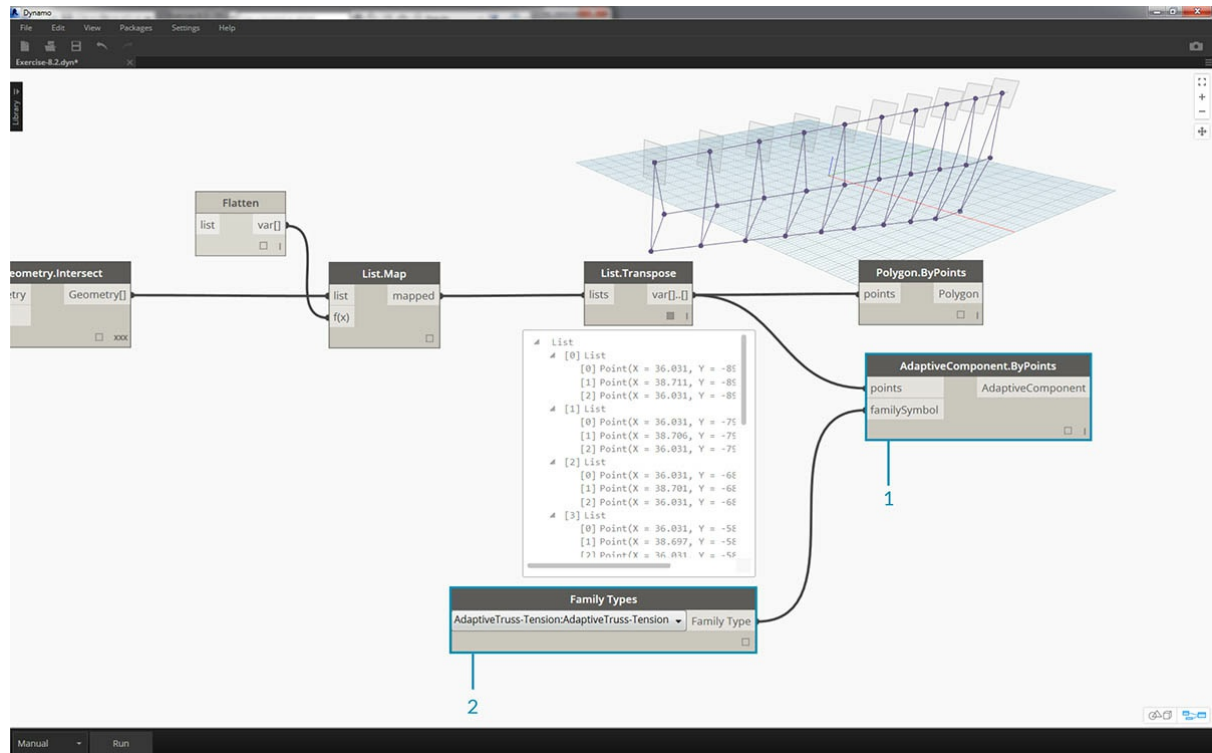
Principios de interoperabilidad

El componente adaptativo es un buen ejemplo de los procedimientos recomendados de interoperabilidad. Podemos crear una matriz de componentes adaptativos mediante la definición de los puntos adaptativos fundamentales. Y, al transferir estos datos a otros programas, tenemos la posibilidad de reducir la geometría a datos simples. La importación y la exportación con un programa como Excel siguen una lógica similar.

Supongamos que un consultor de fachada desea conocer la ubicación de los elementos de la viga de celosía sin necesidad de realizar un análisis mediante geometría totalmente articulada. Para preparar la fabricación, el consultor puede examinar la ubicación de los puntos adaptativos a fin de regenerar la geometría en un programa como Inventor.

El flujo de trabajo que configuraremos en el siguiente ejercicio nos permite acceder a todos estos datos mientras creamos la definición para la creación de elementos de Revit. Mediante este proceso, podemos combinar la conceptualización, la documentación y la fabricación en un flujo de trabajo perfecto. Esto permite generar un proceso más inteligente y eficaz para la interoperabilidad.

Varios elementos y listas

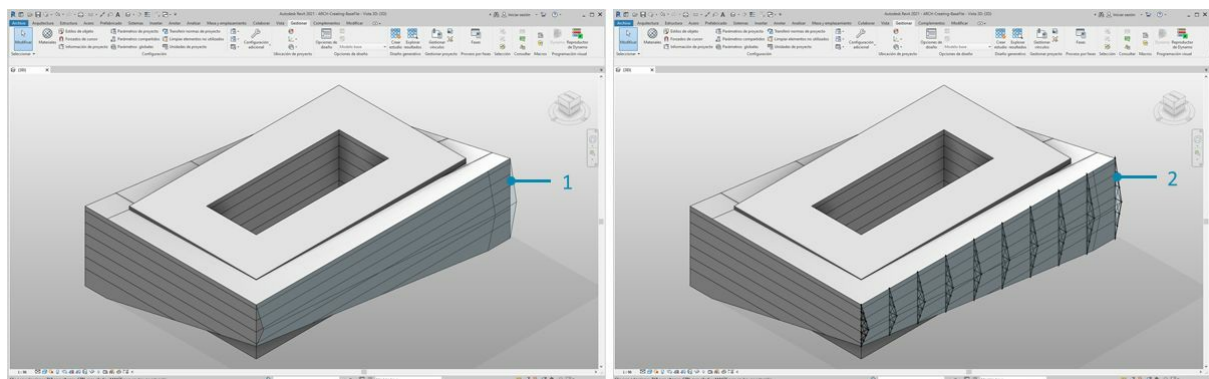


En el siguiente ejercicio, se describe cómo Dynamo consulta los datos para la creación de elementos de Revit. Para generar varios componentes adaptativos, definimos una lista de listas en la que cada lista tiene tres puntos que representan cada punto del componente adaptativo. Tendremos esto en cuenta a medida que gestionamos las estructuras de datos en Dynamo.

Ejercicio

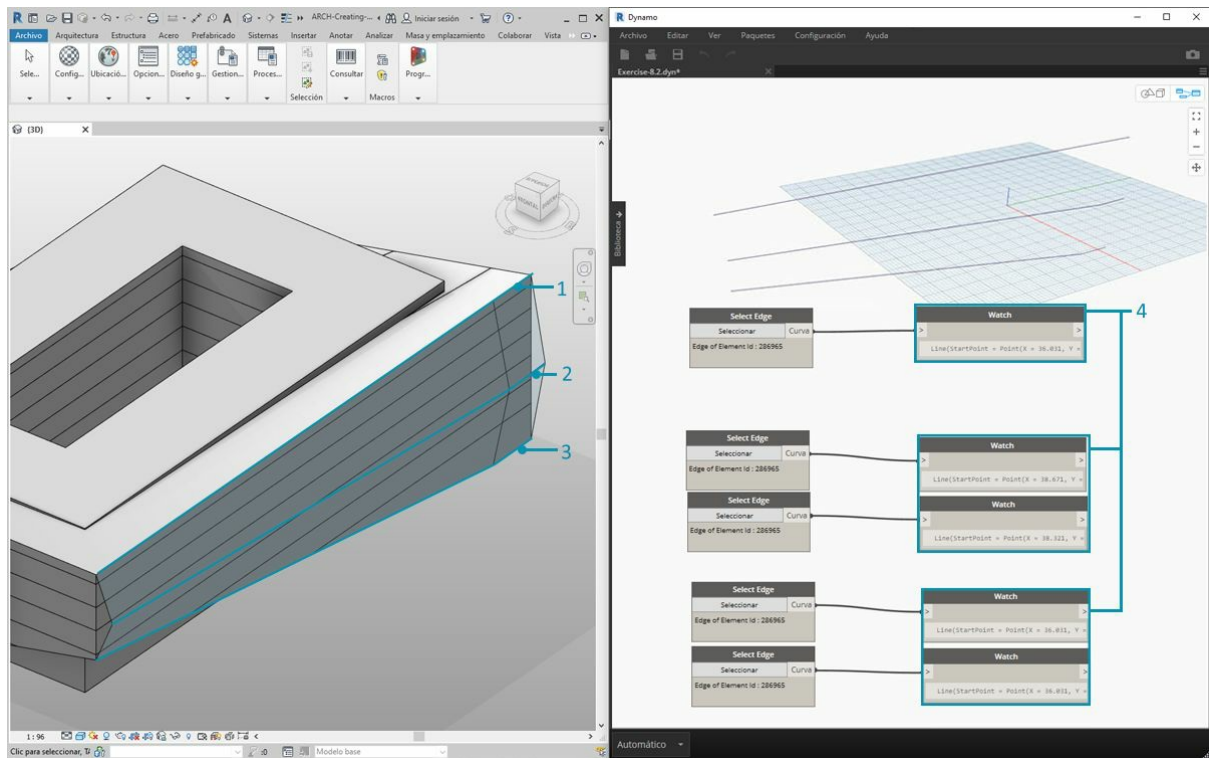
Descargue los archivos de ejemplo que acompañan a este ejercicio (haga clic con el botón derecho y seleccione "Guardar enlace como..."). En el Apéndice, se incluye una lista completa de los archivos de ejemplo.

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



Comenzando con el archivo de ejemplo de esta sección (o continuando con el archivo de Revit de la sesión anterior), veremos la misma masa de Revit.

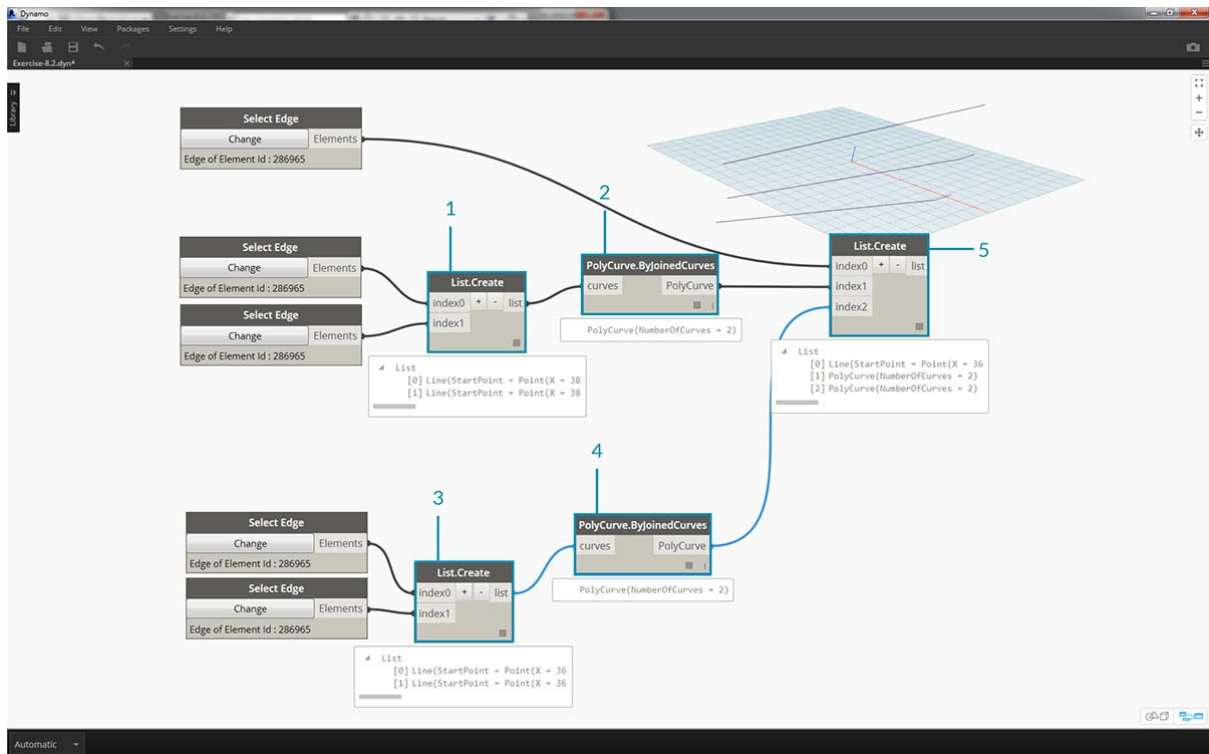
1. Este es el archivo, tal y como se ha abierto.
2. Este es el sistema de vigas de celosía que hemos creado con Dynamo, vinculado de forma inteligente a la masa de Revit.



Hemos utilizado los nodos "Select Model Element" y "Select Face", bajamos un nivel más en la jerarquía y utilizamos "Select Edge". Con el solucionador de Dynamo establecido en "Automático", el gráfico se actualizará continuamente con los cambios realizados en el archivo de Revit. El borde que seleccionamos se asocia dinámicamente a la topología del elemento de Revit. Mientras la topología* no cambie, la conexión permanecerá vinculada entre Revit y Dynamo.

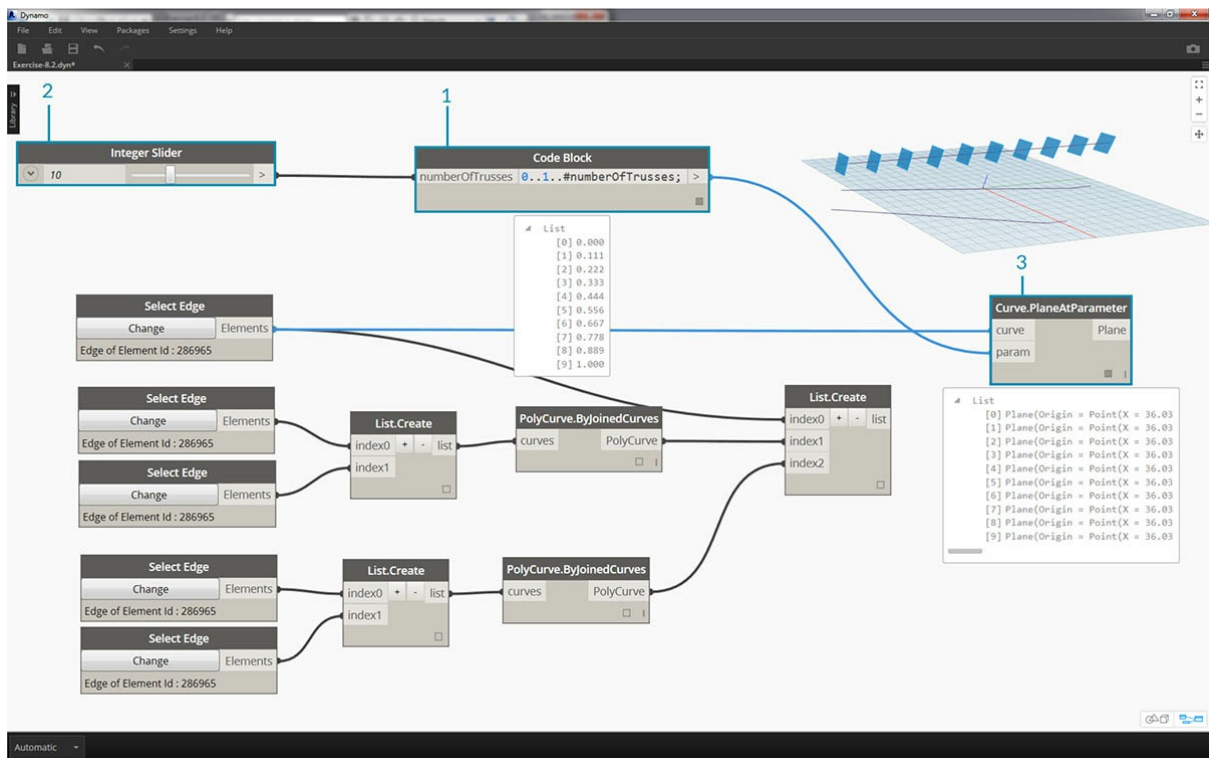
1. Seleccione la curva superior de la fachada de la cristalera. Esto abarca toda la longitud del edificio. Si tiene problemas para seleccionar el borde, recuerde realizar la selección en Revit. Para ello, coloque el cursor sobre el borde y pulse "Tab" hasta que se resalte el borde seleccionado.
2. Mediante dos nodos "Select Edge", seleccione cada borde que represente la inclinación en el centro de la fachada.
3. Haga lo mismo para los bordes inferiores de la fachada en Revit.
4. En los nodos Watch, se muestra que ahora tenemos líneas en Dynamo. Esto se convierte automáticamente en geometría de Dynamo, ya que los bordes en sí no son elementos de Revit. Estas curvas son las referencias que utilizaremos para crear instancias de vigas de celosía adaptativas a lo largo de la fachada.

**Nota: para mantener la coherencia de la topología, consultaremos un modelo sin caras o bordes adicionales añadidos. Aunque los parámetros pueden cambiar su forma, el modo en que se crea sigue siendo coherente.*



Debemos unir primero las curvas y fusionarlas en una lista. De esta forma, podemos "agrupar" las curvas para realizar operaciones geométricas.

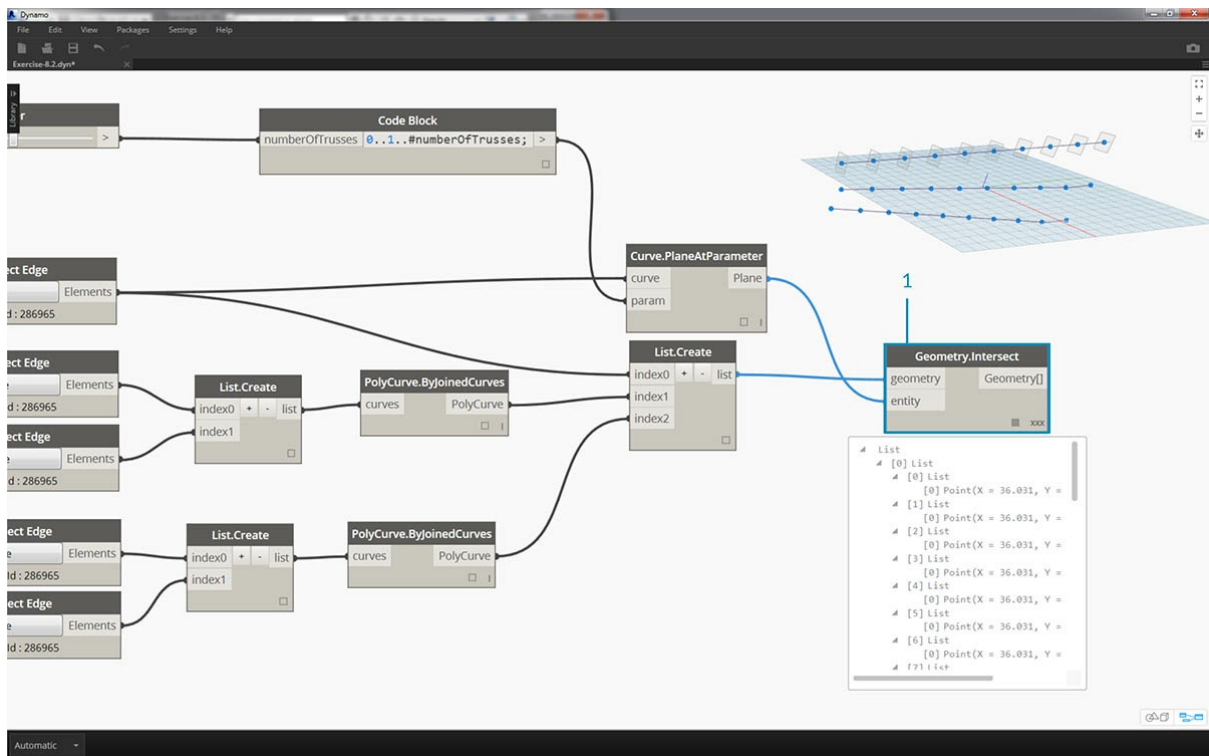
1. Cree una lista para las dos curvas del centro de la fachada.
2. Una las dos curvas en una PolyCurve. Para ello, conecte el componente *List.Create* a un nodo *PolyCurve.ByJoinedCurves*.
3. Cree una lista para las dos curvas de la parte inferior de la fachada.
4. Una las dos curvas en una PolyCurve. Para ello, conecte el componente *List.Create* a un nodo *PolyCurve.ByJoinedCurves*.
5. Por último, una las tres curvas principales (una línea y dos PolyCurves) en una lista.



Deseamos aprovechar la curva superior, que es una línea, y representa toda la extensión de la fachada. Crearemos planos a lo largo de esta línea para que se intersecten con el conjunto de curvas que hemos agrupado en una lista.

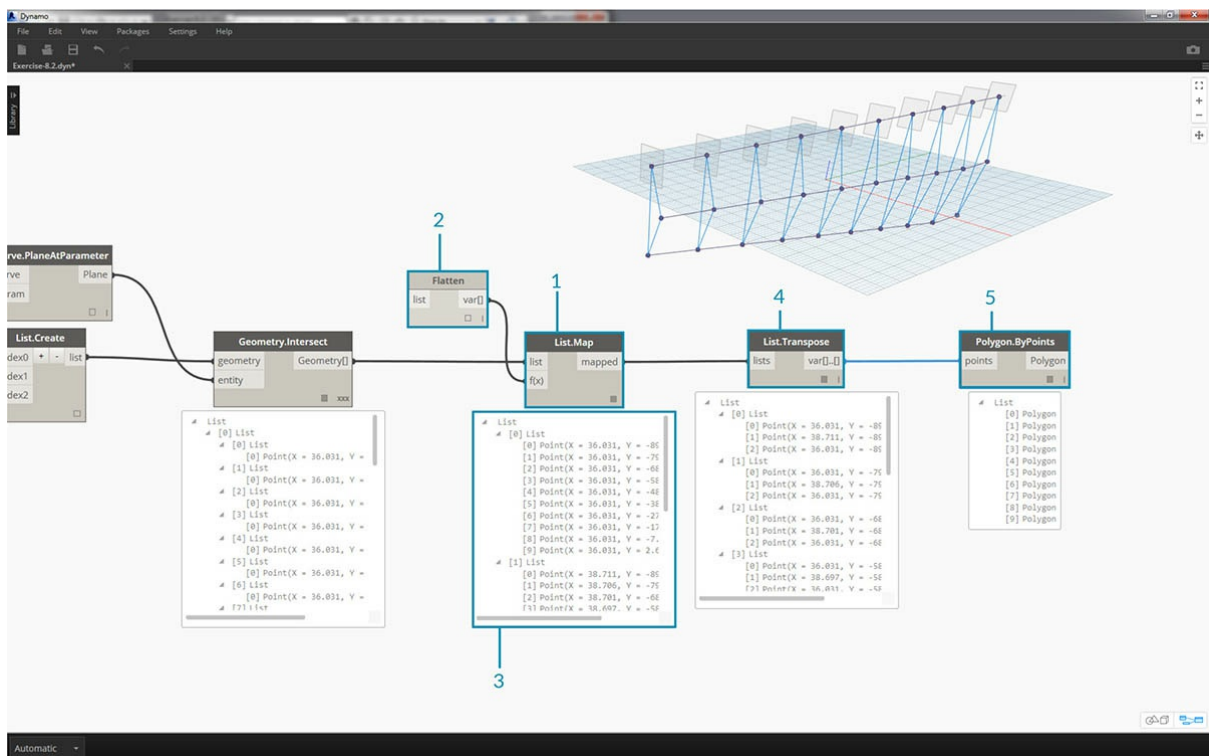
1. Con un *bloque de código*, defina un intervalo mediante la sintaxis: `0..1..#numberOfTrusses;`.
2. Conecte un *control deslizante de entero* a la entrada del bloque de código. Como podrá haber adivinado, esto representará el número de vigas de celosía. Observe que el control deslizante determina el número de elementos del intervalo definido de 0 a 1.

3. Conecte el bloque de código a la entrada *param* de un nodo "Curve.PlaneAtParameter" y conecte el borde superior a la entrada *curve*. Esto nos dará diez planos, distribuidos de forma uniforme a lo largo de la extensión de la fachada.



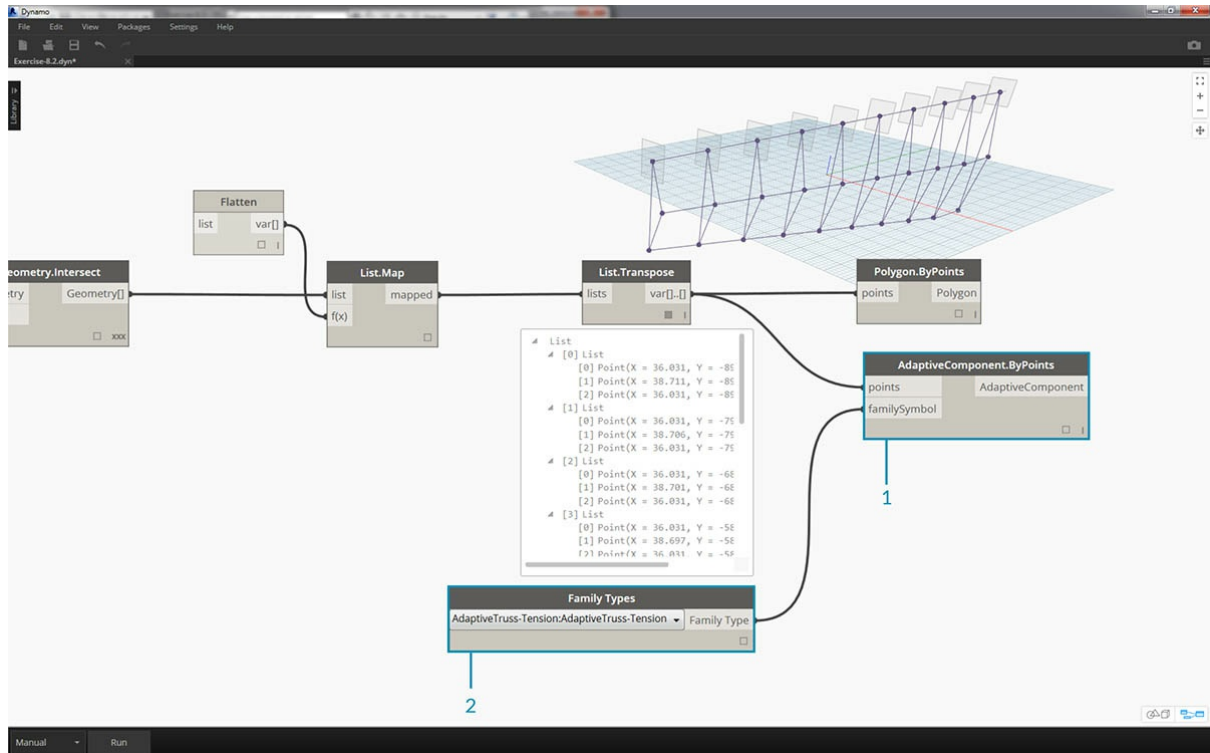
Un plano es una unidad abstracta de geometría que representa un espacio bidimensional infinito. Los planos son ideales para la creación de contornos y la intersección, tal y como los vamos a configurar en este paso.

1. Con el nodo **Geometry.Intersect** (observe el encaje de producto cartesiano), conecte **Curve.PlaneAtParameter** a la entrada *entity* del nodo **Geometry.Intersect**. Conecte el nodo **List.Create** principal a la entrada *geometry*. Ahora aparecen puntos en la ventana gráfica de Dynamo que representan la intersección de cada curva con los planos definidos.



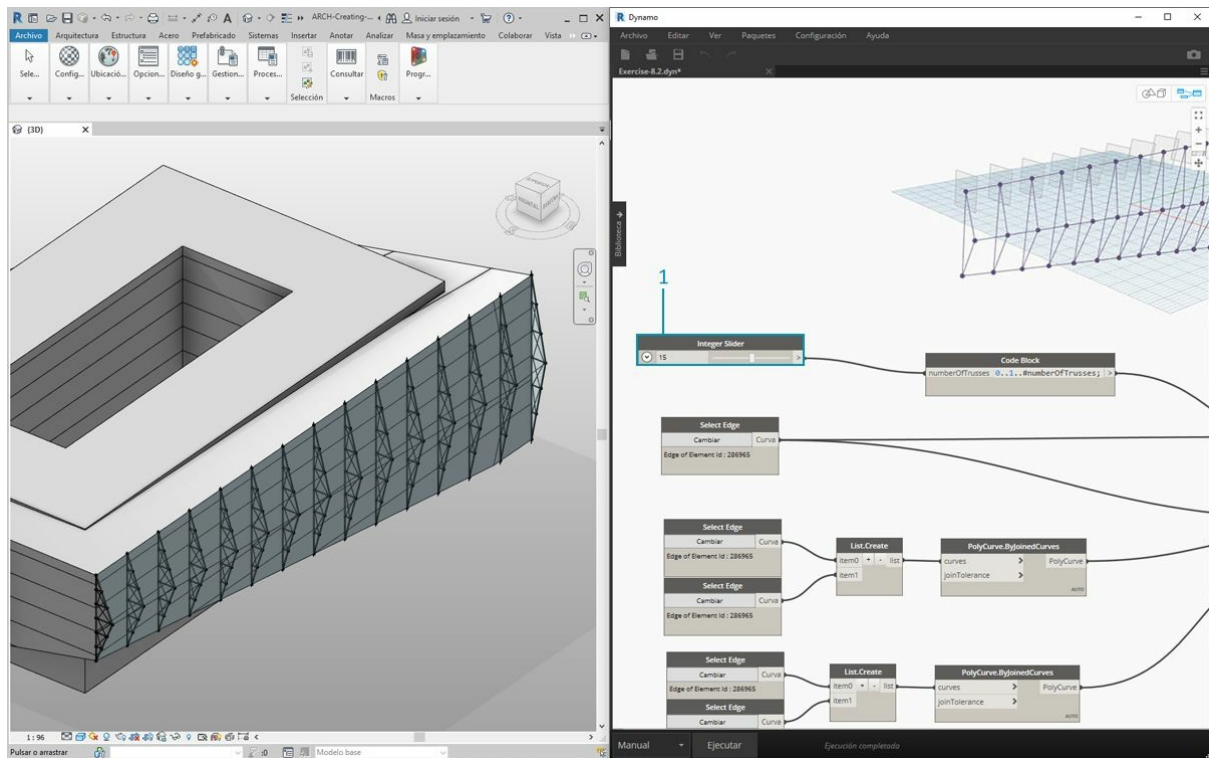
Observe que la salida es una lista de listas de listas. Hay demasiadas listas para nuestros fines. Vamos a realizar un aplanamiento parcial aquí. Debemos bajar un nivel en la lista y aplanar el resultado. Para ello, utilizaremos la operación **List.Map**, tal y como se explica en el capítulo sobre las listas del manual de introducción.

1. Conecte el nodo *Geometry.Intersect* a la entrada de lista de *List.Map*.
2. Conecte un nodo *Flatten* a la entrada f(x) de *List.Map*. Los resultados proporcionan tres listas, cada uno con un número igual a la cantidad de vigas de celosía.
3. Debemos cambiar estos datos. Si desea crear una instancia de la viga de celosía, debemos utilizar el mismo número de puntos adaptativos que se ha definido en la familia. Se trata de un componente adaptativo de tres puntos, por lo que, en lugar de tres listas con 10 elementos cada una (*NumberOfTrusses*), deseamos obtener 10 listas con tres elementos cada una. De este modo, podemos crear 10 componentes adaptativos.
4. Conecte *List.Map* a un nodo *List.Transpose*. Ahora tenemos la salida de datos deseada.
5. Para confirmar que los datos sean correctos, añada un nodo *Polygon.ByPoints* al lienzo y realice una comprobación con la vista preliminar de Dynamo.

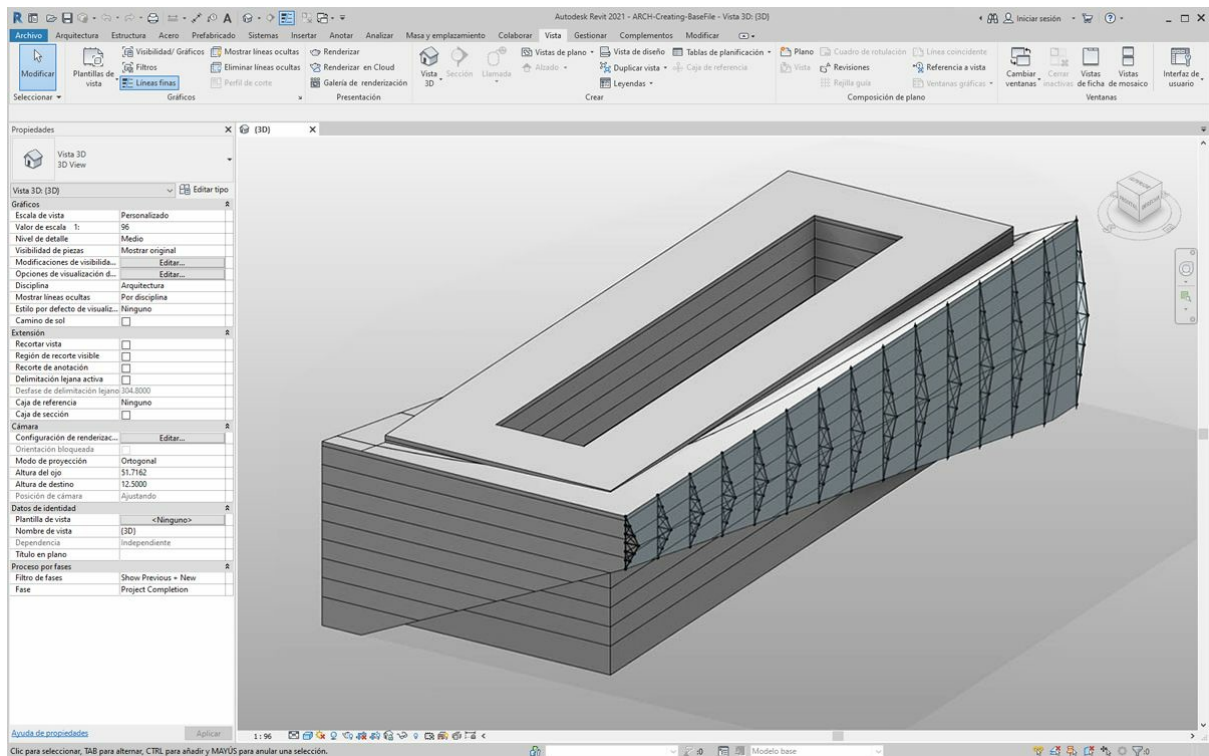


Del mismo modo que hemos creado los polígonos, disponemos en una matriz los componentes adaptativos.

1. Añada un nodo *AdaptiveComponent.ByPoints* al lienzo y conecte el nodo *List.Transpose* a la entrada *points*.
2. Mediante un nodo *Family Types*, seleccione la familia "AdaptiveTruss" y conecte esta a la entrada *familySymbol* del nodo *AdaptiveComponent.ByPoints*.



1. Para ajustar el sistema de vigas de celosía, asignemos el valor 15 a *numberOfTrusses*.



Y, como prueba final, al seleccionar la masa en Revit y editar los parámetros de ejemplar, podemos cambiar la forma del edificio y comprobar cómo cambia también la viga de celosía. Recuerde que este gráfico de Dynamo debe estar abierto para ver esta actualización y que el vínculo se romperá en cuanto se cierre.

Elementos DirectShape

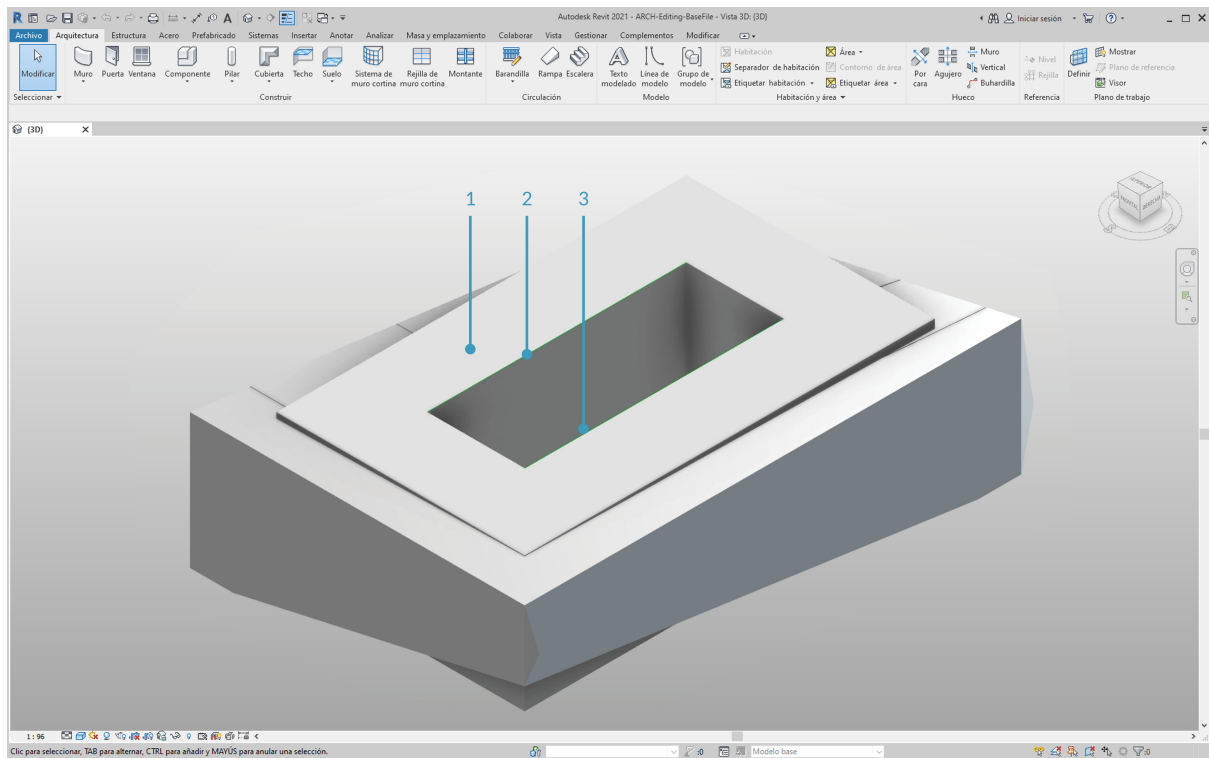
Otro método para importar geometría paramétrica de Dynamo a Revit es DirectShape. En resumen, el elemento DirectShape y las clases relacionadas permiten almacenar formas geométricas creadas externamente en un documento de Revit. La geometría puede incluir mallas o sólidos cerrados. DirectShape se ha diseñado principalmente para la importación de formas de otros formatos de datos, como IFC o STEP, donde no hay suficiente información disponible para crear un elemento de Revit "real". Al igual que en el flujo de trabajo de IFC y STEP, la función DirectShape permite importar sin problemas geometrías creadas en Dynamo a proyectos de Revit como elementos reales.

Repasemos cómo importar geometría de Dynamo como DirectShape en el proyecto de Revit y realicemos ejercicios sobre este tema. Mediante este método, podemos asignar la categoría, el material y el nombre de una geometría importada, mientras se mantiene un vínculo paramétrico con nuestro gráfico de Dynamo.

Ejercicio

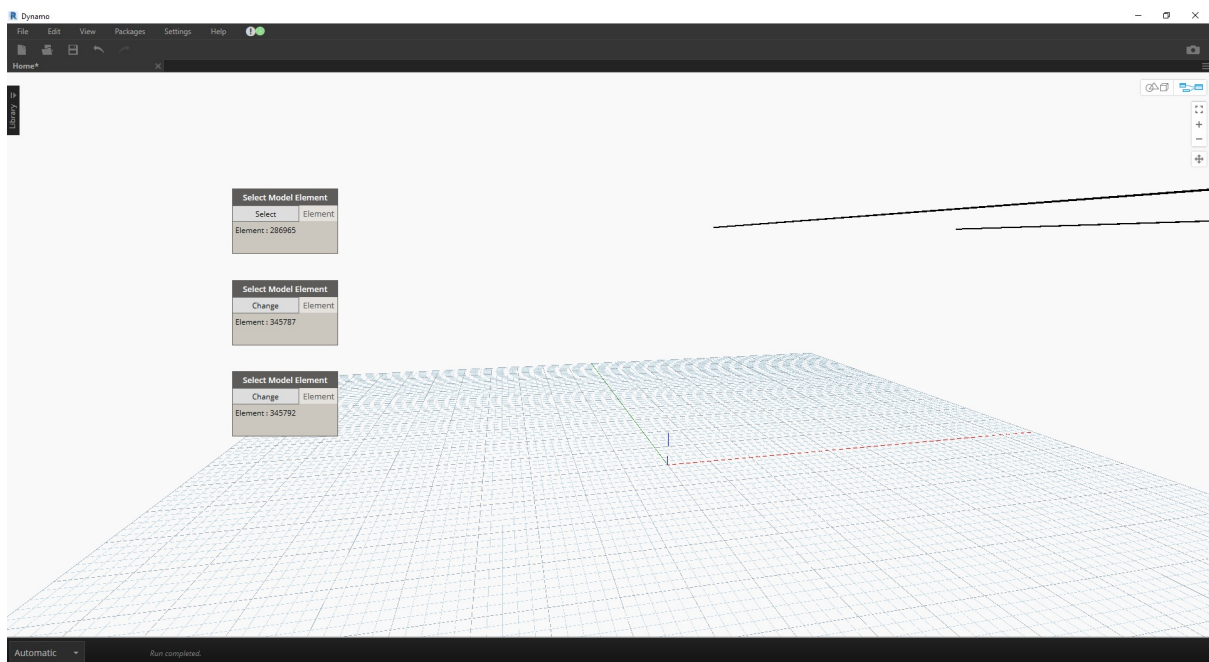
Descargue los archivos de ejemplo que acompañan a este ejercicio (haga clic con el botón derecho y seleccione "Guardar enlace como..."). En el Apéndice, se incluye una lista completa de los archivos de ejemplo.

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

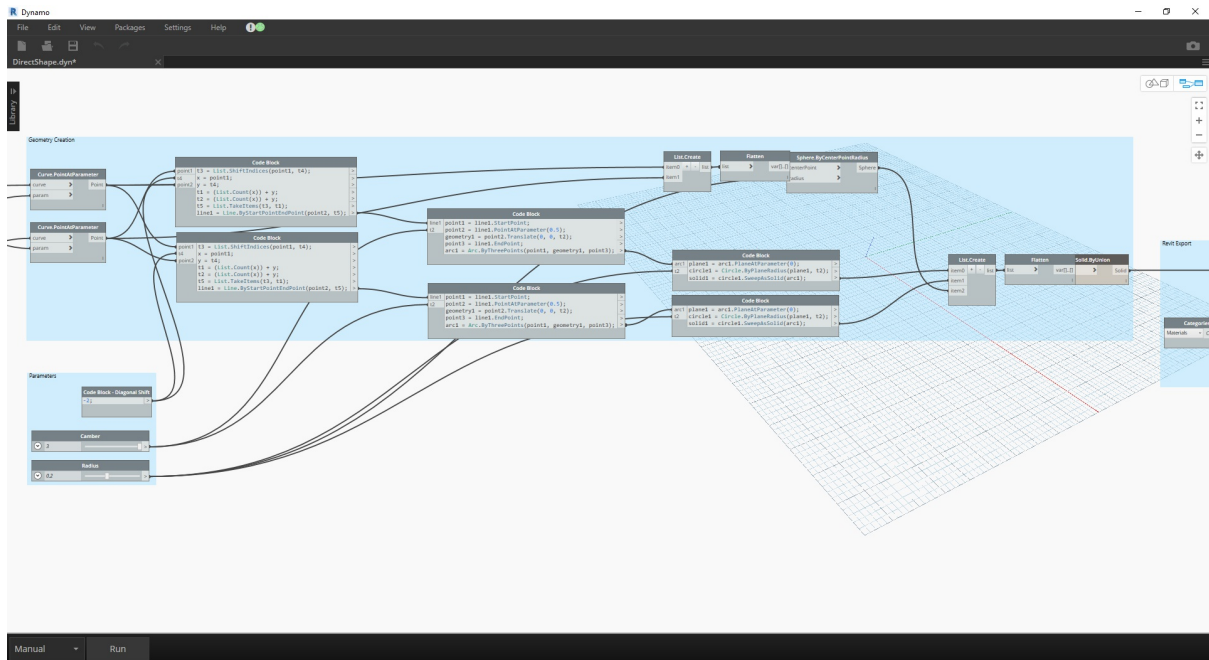


Abra primero el archivo de ejemplo para esta lección: ARCH-DirectShape-BaseFile.rvt.

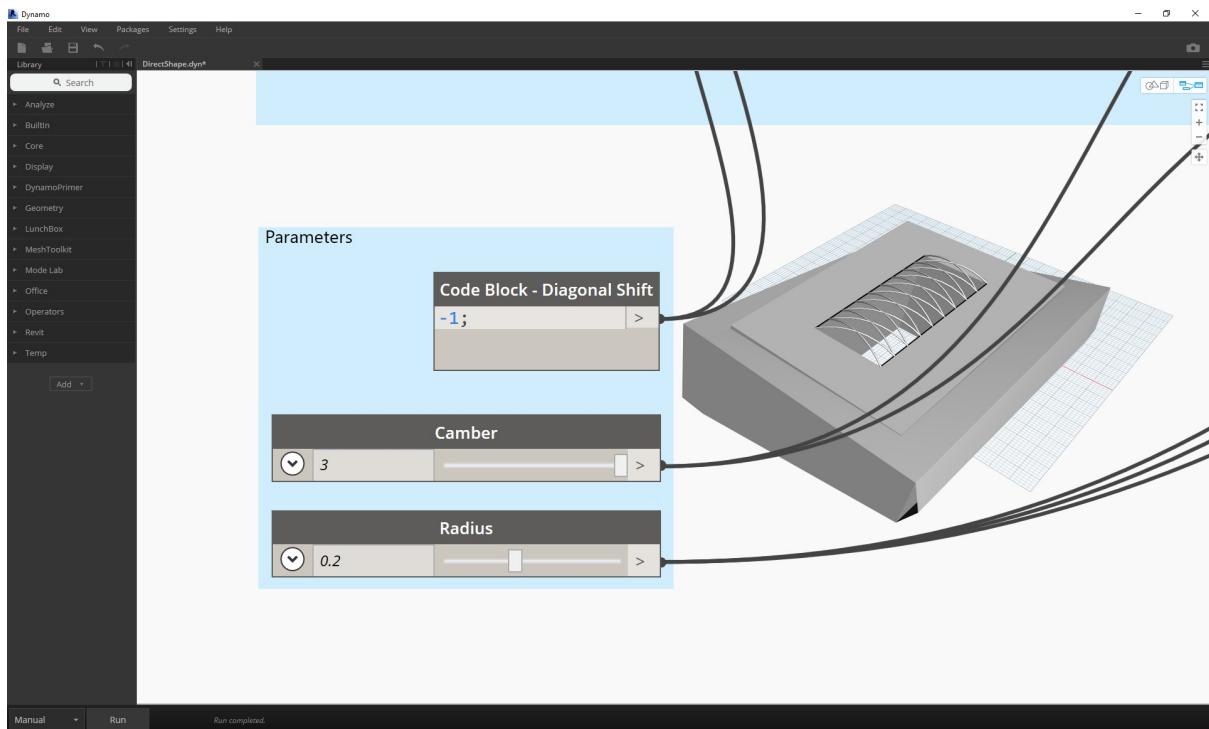
1. En la vista 3D, vemos la masa de construcción de la lección anterior.
2. A lo largo del borde del atrio se encuentra una curva de referencia, la utilizaremos como una curva a la que haremos referencia en Dynamo.
3. A lo largo del borde opuesto del atrio se encuentra otra curva de referencia a la que también haremos referencia en Dynamo.



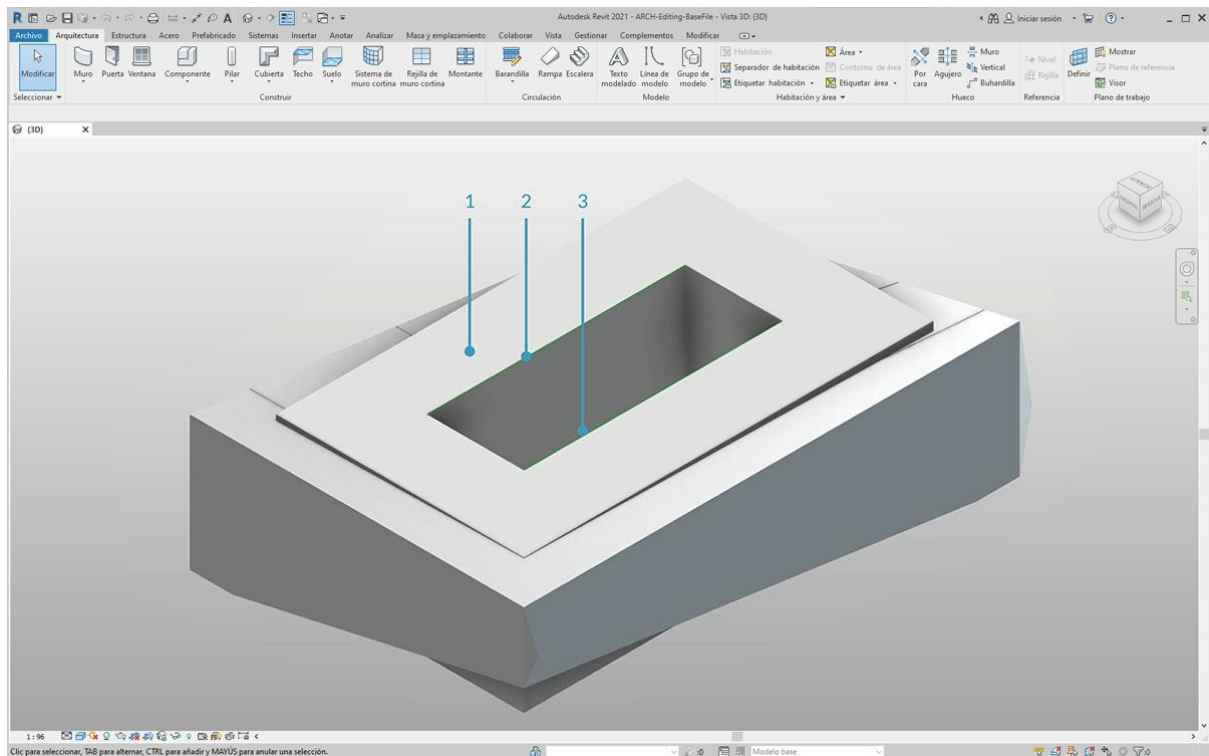
1. Para hacer referencia a la geometría en Dynamo, utilizaremos *Select Model Element* para cada miembro en Revit. Seleccione la masa en Revit e importe la geometría en Dynamo mediante *Element.Faces*: la masa debería estar visible en la vista preliminar de Dynamo.
2. Importe una curva de referencia en Dynamo mediante *Select Model Element* y *CurveElement.Curve*.
3. Importe la otra curva de referencia en Dynamo mediante *Select Model Element* y *CurveElement.Curve*.



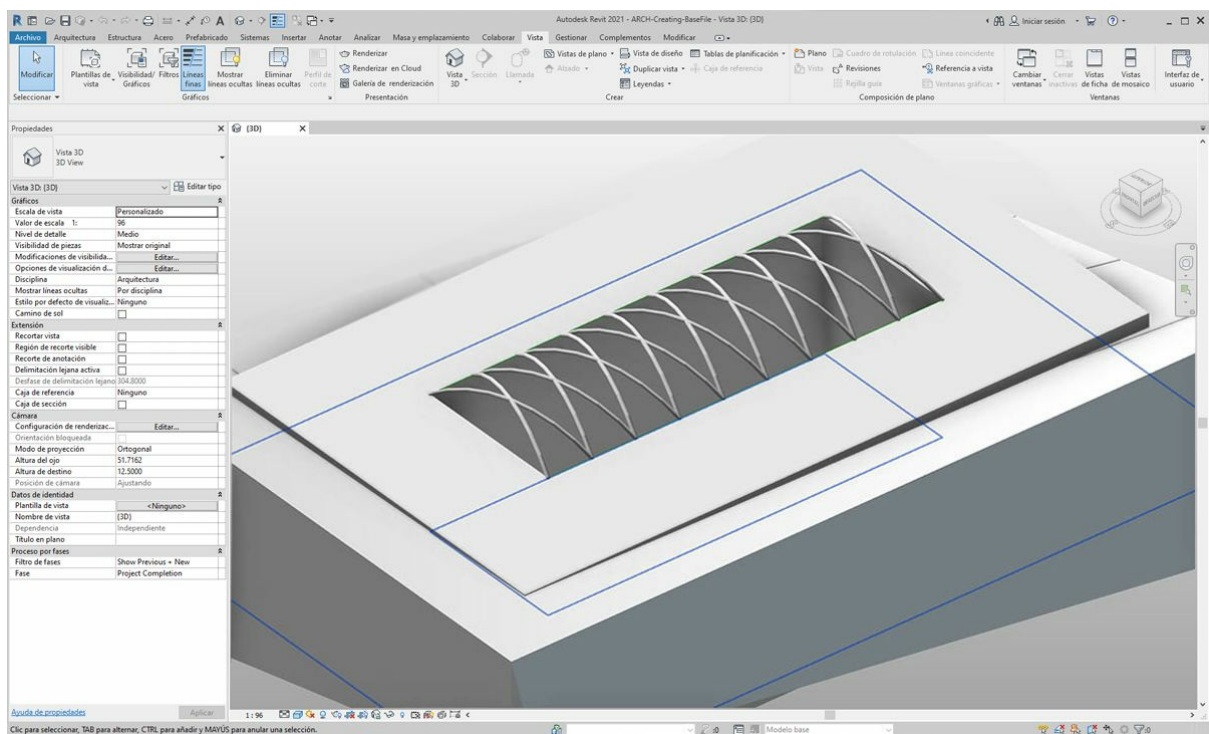
1. Al alejar la vista y encuadrar a la derecha en el gráfico de ejemplo, vemos un gran grupo de nodos, que son operaciones geométricas que generan la estructura de cubierta de enrejado visible en la vista preliminar de Dynamo. Estos nodos se generan mediante la función de *nodo a código*, tal como se describe en la [sección de bloque de código](#) del manual de introducción.
2. La estructura se rige por tres parámetros principales: Desplazamiento diagonal, Contraflecha y Radio.



Al aplicar zoom, se obtiene una vista en primer plano de los parámetros de este gráfico. Podemos ajustarlos para obtener diferentes salidas de geometría.



1. Al soltar el nodo **DirectShape.ByGeometry** en el lienzo, vemos que tiene cuatro entradas: **geometría, categoría, material y nombre**.
2. La geometría será el sólido creado a partir de la parte de creación de geometría del gráfico
3. La entrada de categoría se elige mediante el nodo **Categorías** del menú desplegable. En este caso, utilizaremos "Armazón estructural".
4. La entrada de material se selecciona a través de la matriz de nodos anterior, aunque puede definirse más simplemente como "Por defecto" en este caso.



Después de ejecutar Dynamo, al volver a Revit, tenemos la geometría importada en la cubierta del proyecto. Se trata de un elemento de armazón estructural en lugar de un modelo genérico. El vínculo paramétrico a Dynamo permanece intacto.

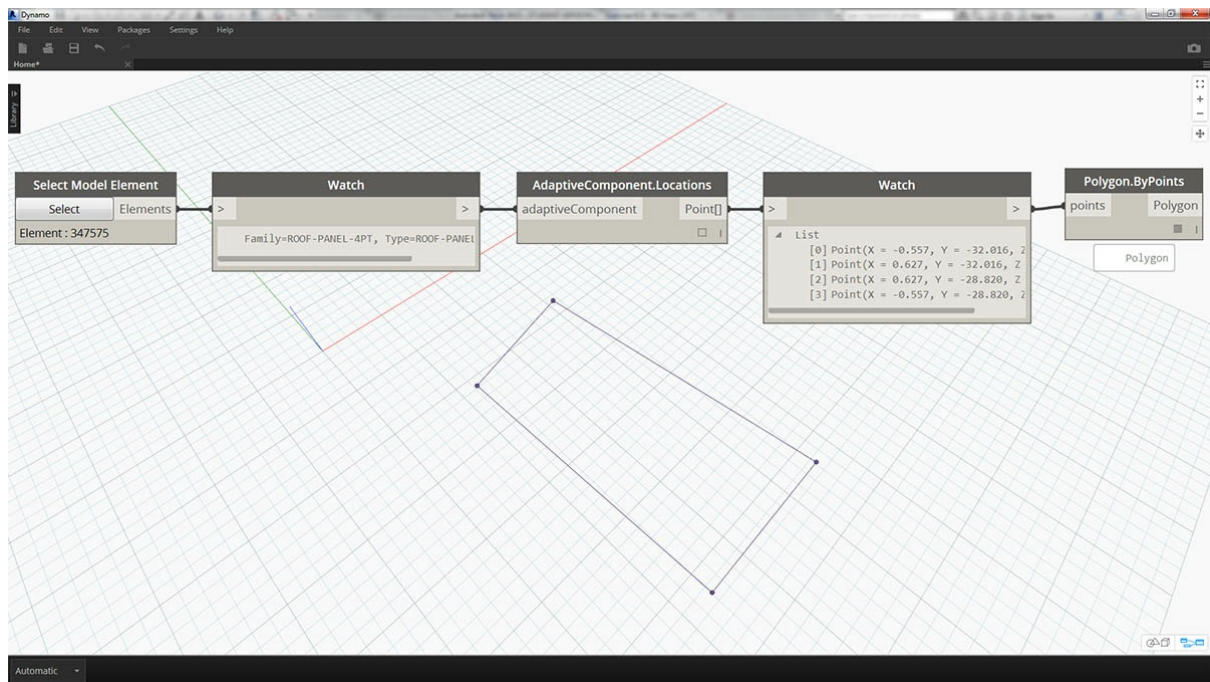
Personalización

Personalización

Aunque ya hemos visto cómo editar una masa de construcción básica, queremos profundizar en el vínculo entre Dynamo y Revit mediante la edición de un gran número de elementos de una sola vez. La personalización a gran escala se vuelve más compleja ya que las estructuras de datos requieren operaciones de lista más avanzadas. Sin embargo, los principios subyacentes de su ejecución son esencialmente los mismos. Vamos a estudiar algunas oportunidades de análisis a partir de un conjunto de componentes adaptativos.

Ubicación de puntos

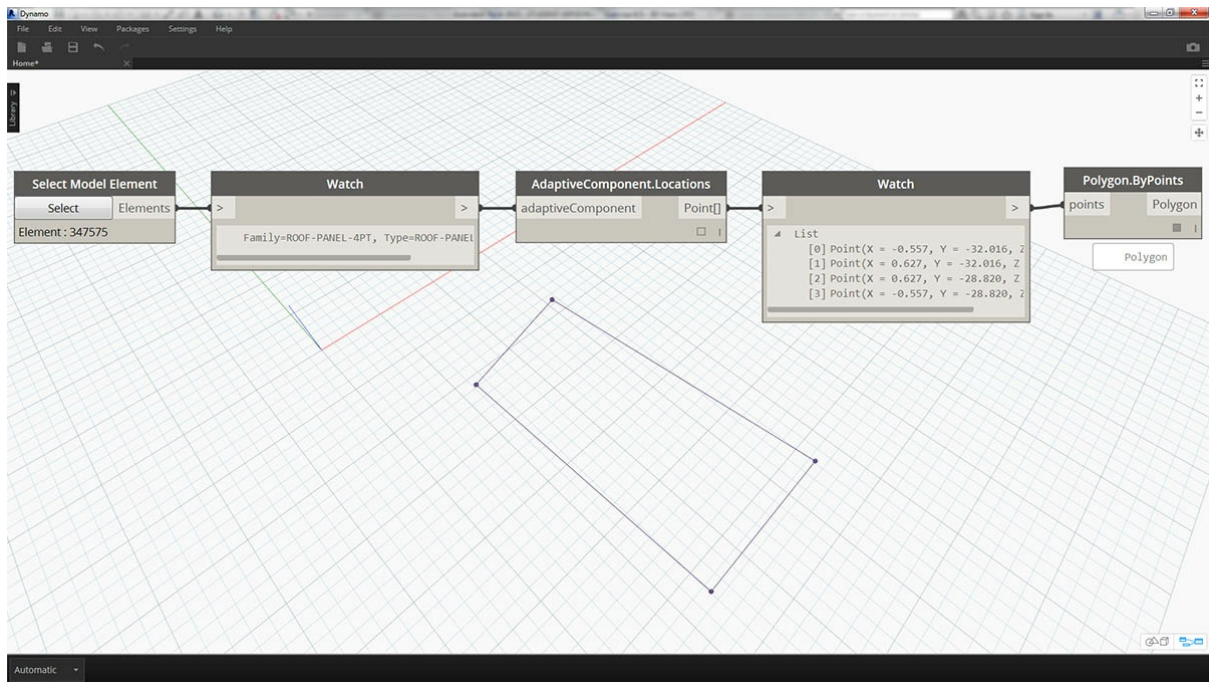
Supongamos que hemos creado un rango de componentes adaptativos y queremos editar parámetros en función de sus ubicaciones de puntos. Los puntos podrían, por ejemplo, controlar un parámetro de grosor relacionado con el área del elemento. O bien, podrían controlar un parámetro de opacidad relacionado con la exposición solar a lo largo del año. Dynamo permite la conexión de análisis a parámetros en unos sencillos pasos; exploraremos una versión básica en el ejercicio siguiente.



Consulte los puntos adaptativos de un componente adaptativo seleccionado mediante el nodo *AdaptiveComponent.Locations*. Esto nos permite trabajar con una versión abstraída de un elemento de Revit para el análisis.

Al extraer la ubicación de los puntos de los componentes adaptativos, se pueden ejecutar distintos análisis para ese elemento. Un componente adaptativo de cuatro puntos le permitirá estudiar la desviación desde el plano en un panel determinado, por ejemplo.

Análisis de orientación solar



Utilice la reasignación para asignar un conjunto de datos a un rango de parámetros. Esta es una herramienta fundamental que se utiliza en un modelo paramétrico. Haremos una demostración en el siguiente ejercicio.

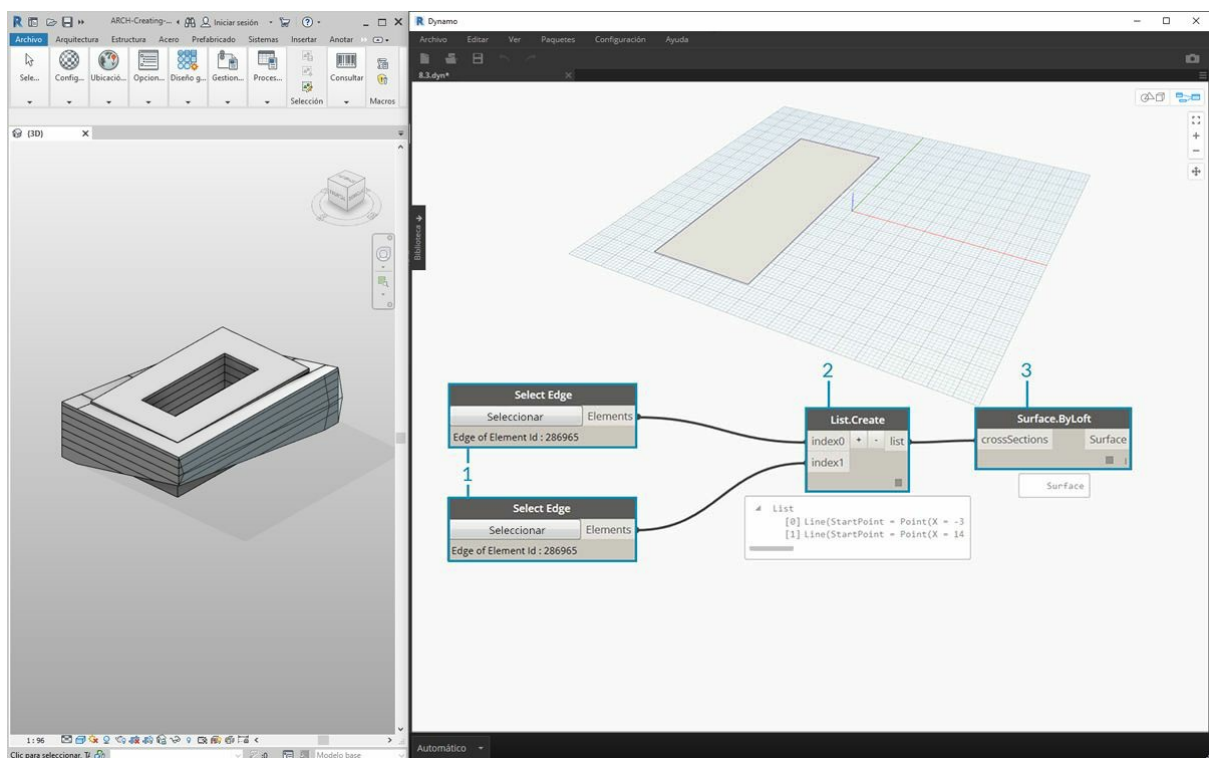
Dynamo permite usar las ubicaciones de los puntos de los componentes adaptativos para crear un plano de ajuste óptimo para cada elemento. También podemos consultar la posición del sol en el archivo de Revit y estudiar la orientación relativa del plano con respecto al sol en comparación con otros componentes adaptativos. Configuremos esta acción en el siguiente ejercicio mediante la creación de una cubierta algorítmica.

Ejercicio:

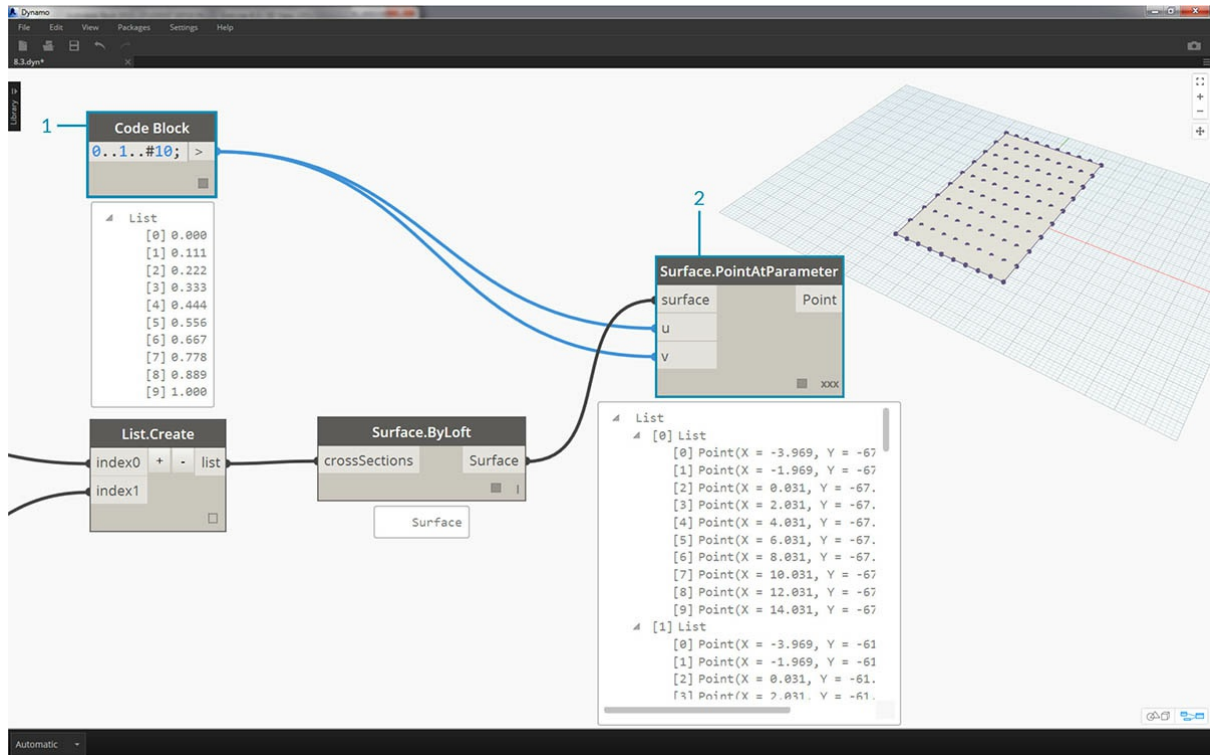
Descargue los archivos de ejemplo que acompañan a este ejercicio (haga clic con el botón derecho y seleccione "Guardar enlace como..."). En el Apéndice se incluye una lista completa de los archivos de ejemplo.

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

En este ejercicio se amplían las técnicas mostradas en la sección anterior. En este caso, vamos a definir una superficie paramétrica a partir de elementos de Revit; para ello, crearemos ejemplares de los componentes adaptativos de cuatro puntos y, a continuación, los editaremos según la orientación del sol.

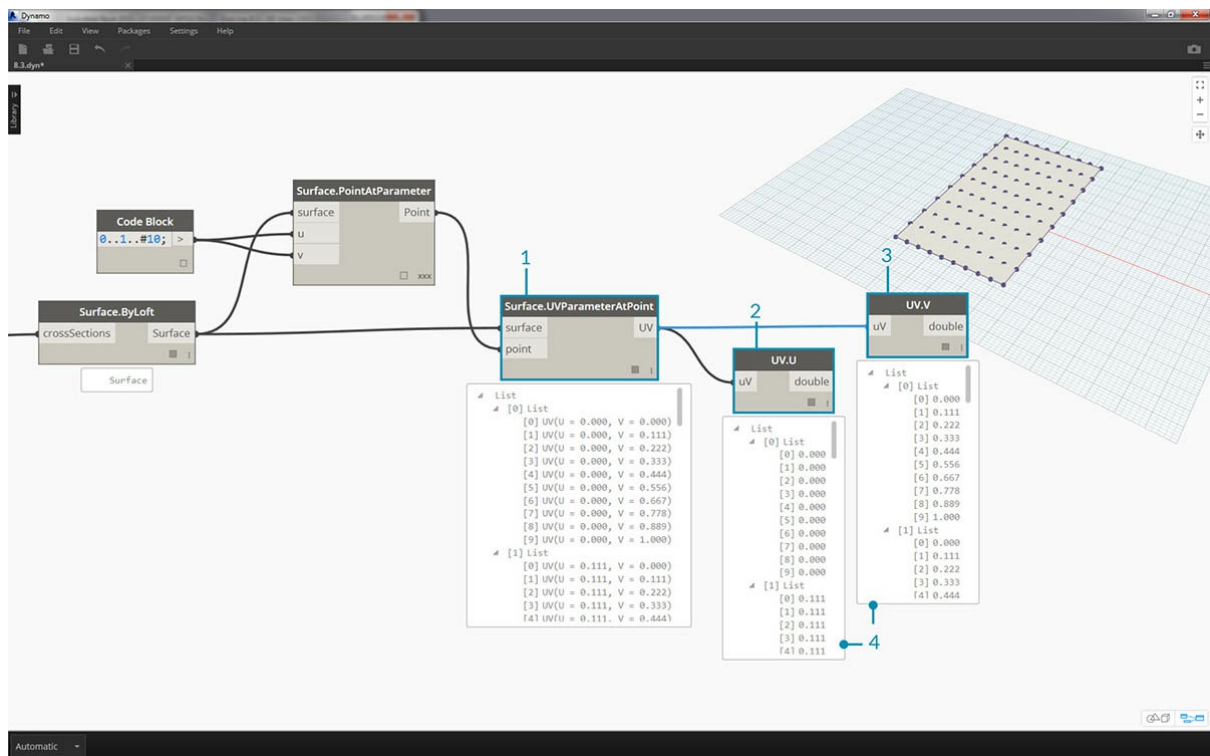


1. Comience por seleccionar dos bordes con el nodo "Select Edge". Los dos bordes son los tramos largos del atrio.
2. Combine los dos bordes en una lista con el nodo List.Create.
3. Cree una superficie entre los dos bordes con un nodo Surface.ByLoft.



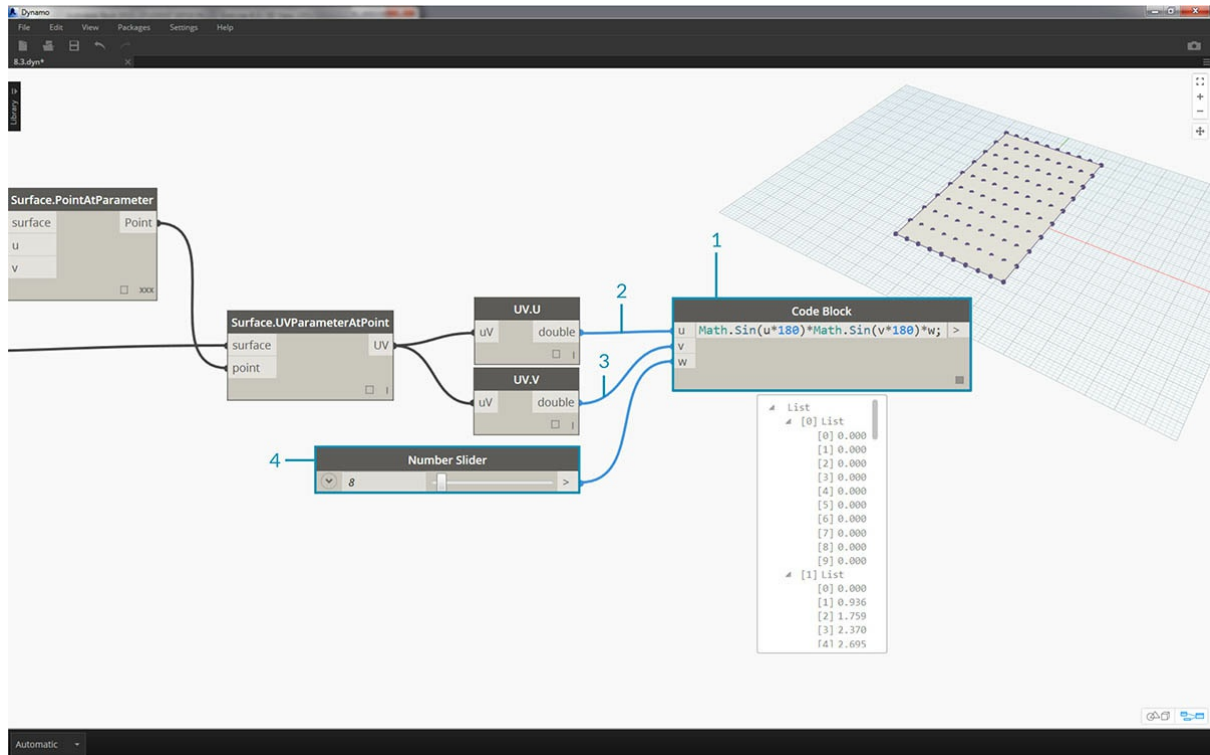
1. Mediante el bloque de código, defina un rango de 0 a 1 con 10 valores espaciados uniformemente: $0 \dots 1 \dots \#10$;
2. Conecte el bloque de código a las entradas u y v de un nodo Surface.PointAtParameter y conecte el nodo Surface.ByLoft a la entrada surface. Haga clic con el botón derecho en el nodo y cambie el encaje a Producto cartesiano. Esto proporcionará una rejilla de puntos en la superficie.

Esta rejilla de puntos sirve como puntos de control para una superficie definida paramétricamente. Queremos extraer las posiciones u y v de cada uno de estos puntos para poder conectarlos a una fórmula paramétrica y mantener la misma estructura de datos. Para ello, podemos consultar las ubicaciones de parámetros de los puntos que acabamos de crear.

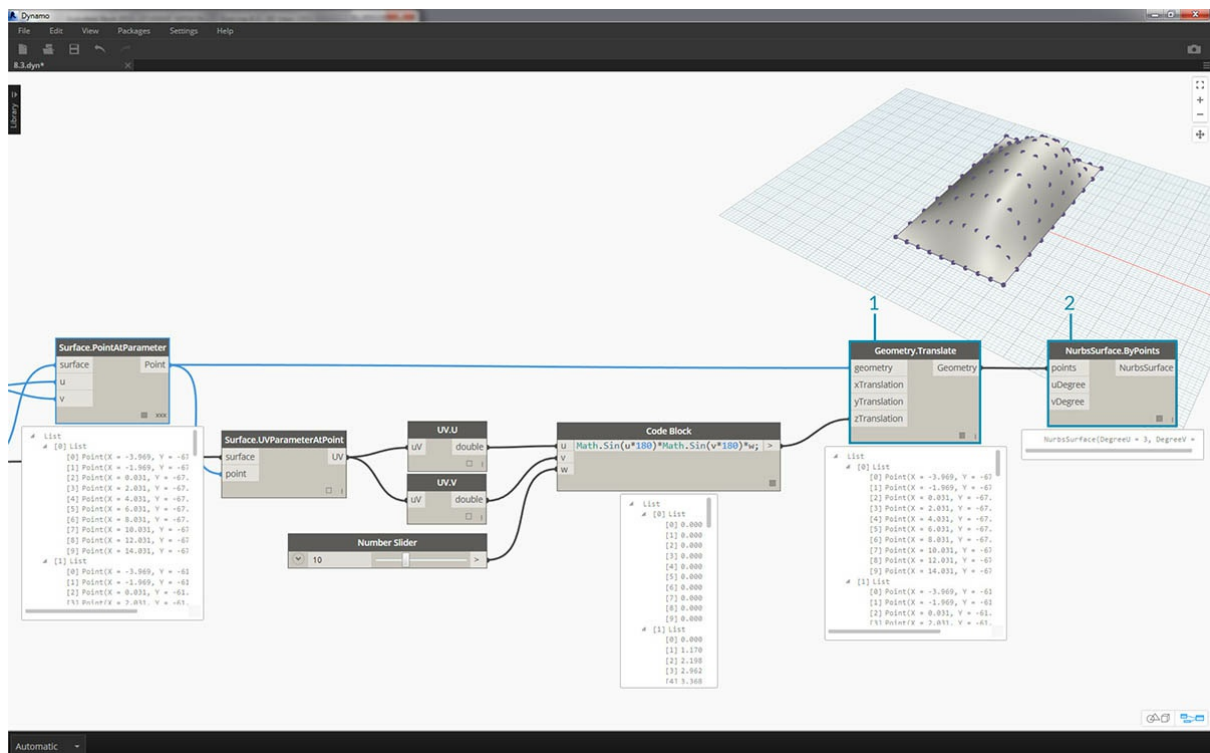


1. Añada un nodo Surface.ParameterAtPoint al lienzo y conecte las entradas como se muestra más arriba.
2. Consulte los valores u de estos parámetros con el nodo UV.U.

3. Consulte los valores v de estos parámetros con el nodo $UV.V$.
4. Las salidas muestran los valores u y v correspondientes a cada punto de la superficie. Ahora tenemos un rango de 0 a 1 para cada valor en la estructura de datos adecuada, por lo que estamos listos para aplicar un algoritmo paramétrico.

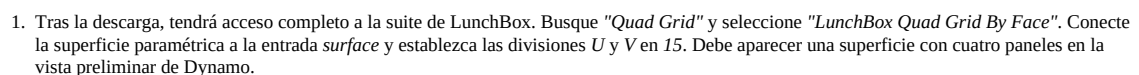
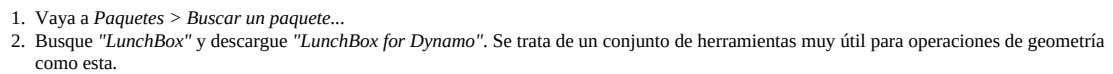


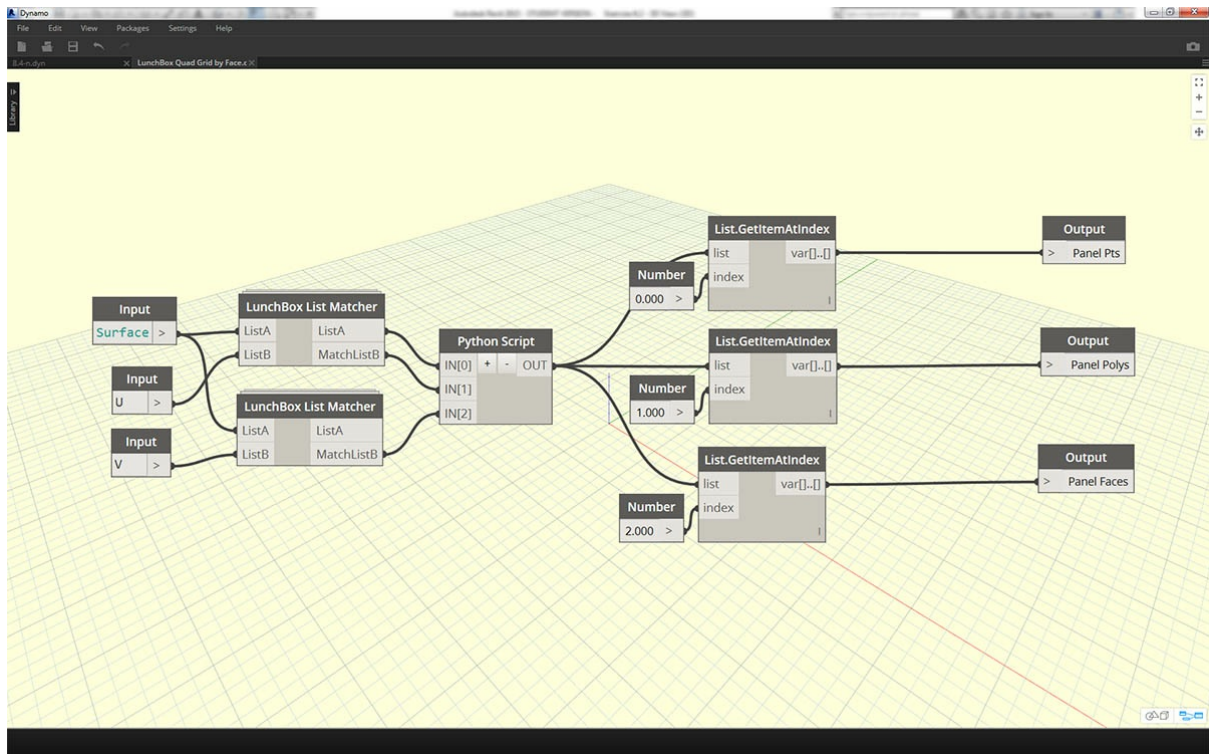
1. Añade un *bloque de código* al lienzo e introduzca el código: $\text{Math.Sin}(u \cdot 180) \cdot \text{Math.Sin}(v \cdot 180) \cdot w$. Esta es una función paramétrica que crea un relieve de seno a partir de una superficie plana.
2. La entrada u se conecta a **UV.U**.
3. La entrada v se conecta a **UV.V**.
4. La entrada w representa la *amplitud* de la forma, por lo que le conectamos un *control deslizante de número*.



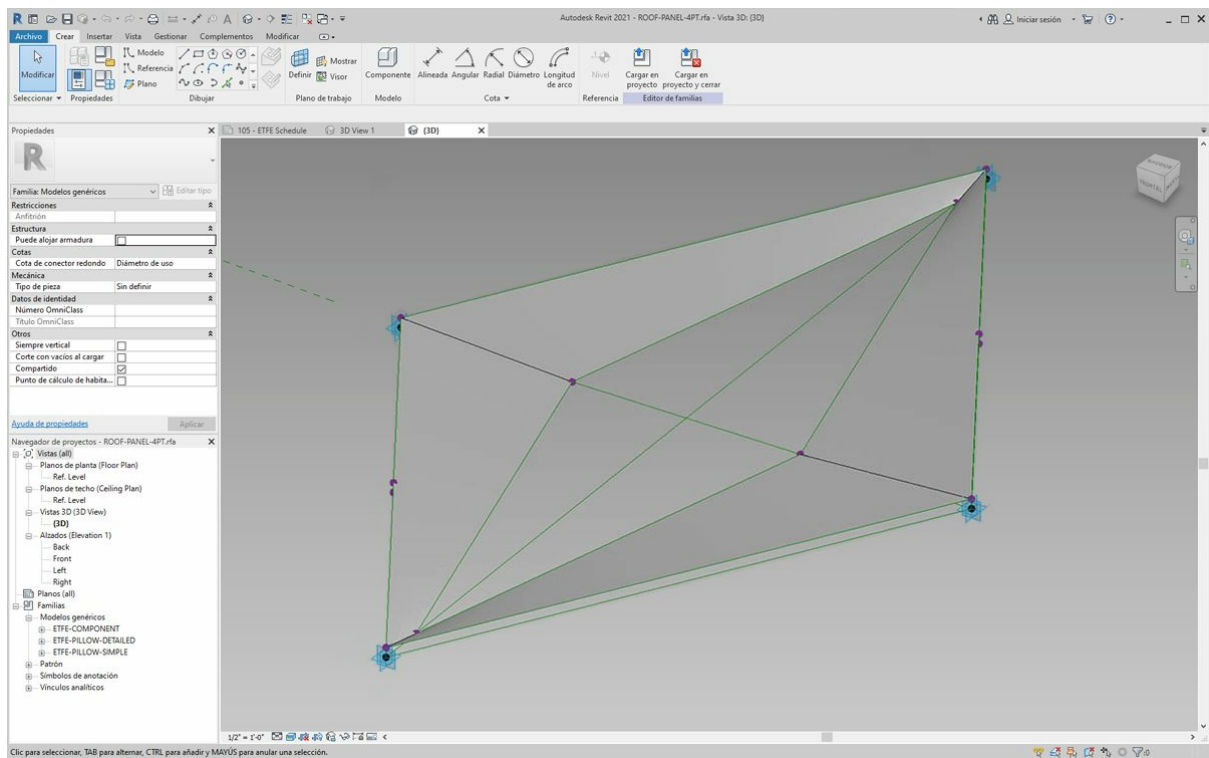
1. Ahora tenemos una lista de valores tal y como los define el algoritmo. Usaremos esta lista de valores para mover los puntos hacia arriba en la dirección $+z$. Con **Geometry.Translate**, conecte el *bloque de código* en **zTranslation** y el nodo **Surface.PointAtParameter** en la entrada **geometry**. Deben aparecer los nuevos puntos en la vista preliminar de Dynamo.
2. Por último, creamos una superficie con el nodo **NurbsSurface.ByPoints** y conectamos el nodo del paso anterior a la entrada **points**. Ya tenemos

Con la superficie paramétrica, vamos a definir un método para dividirla en paneles y crear la matriz de componentes adaptativos de cuatro puntos. Dynamo no dispone de funciones predefinidas para la división en paneles de las superficies, por lo que podemos buscar en la comunidad paquetes de Dynamo útiles.

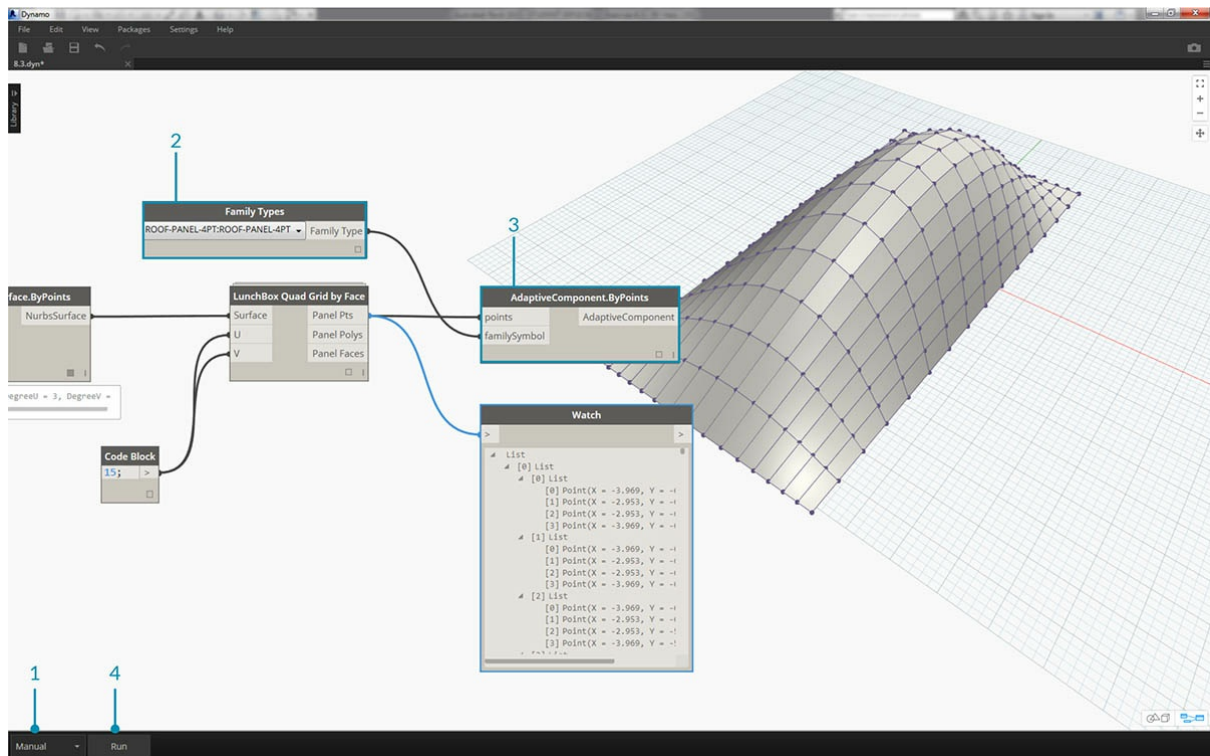




Si tiene curiosidad sobre su configuración, puede hacer doble clic en el nodo *Lunch Box* y ver cómo se ha realizado.

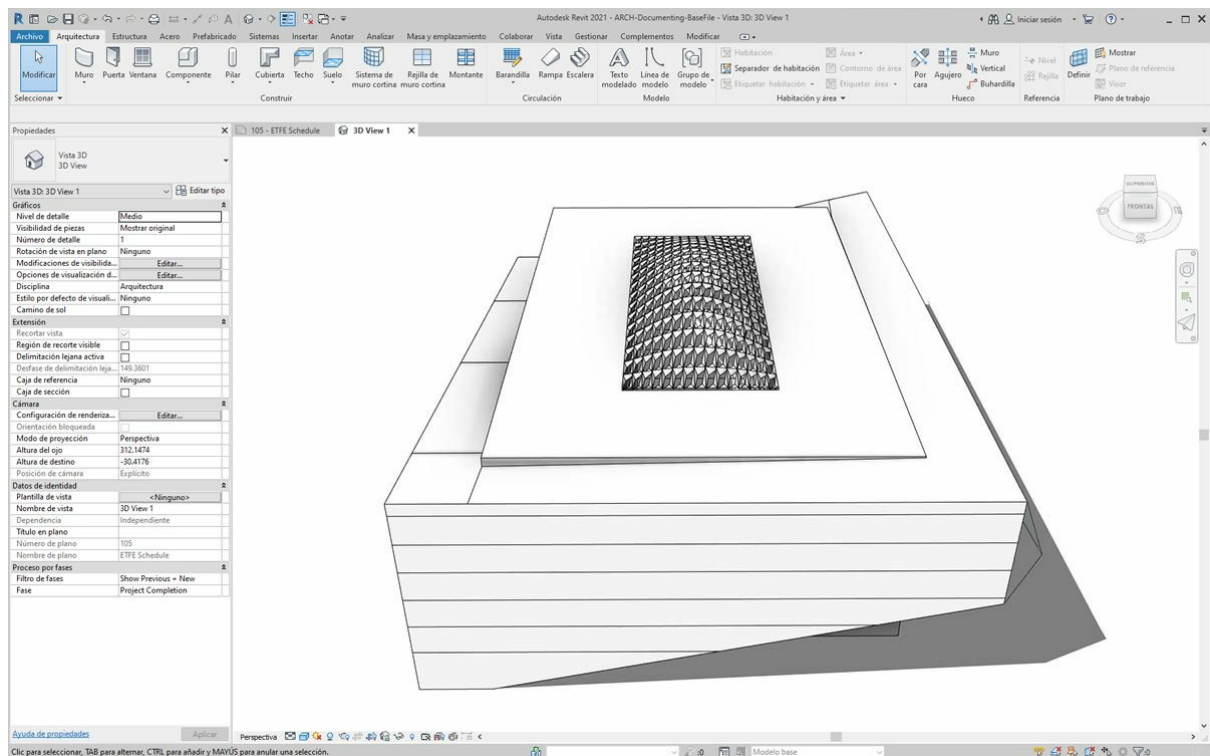


Volvamos a Revit para comprobar rápidamente el componente adaptativo que estamos utilizando aquí. No hace falta seguirlo, pero este es el panel de cubierta del que vamos a crear un ejemplar. Se trata de un componente adaptativo de cuatro puntos que es una representación en bruto de un sistema ETFE. La apertura del vacío central se encuentra en un parámetro denominado "*ApertureRatio*".

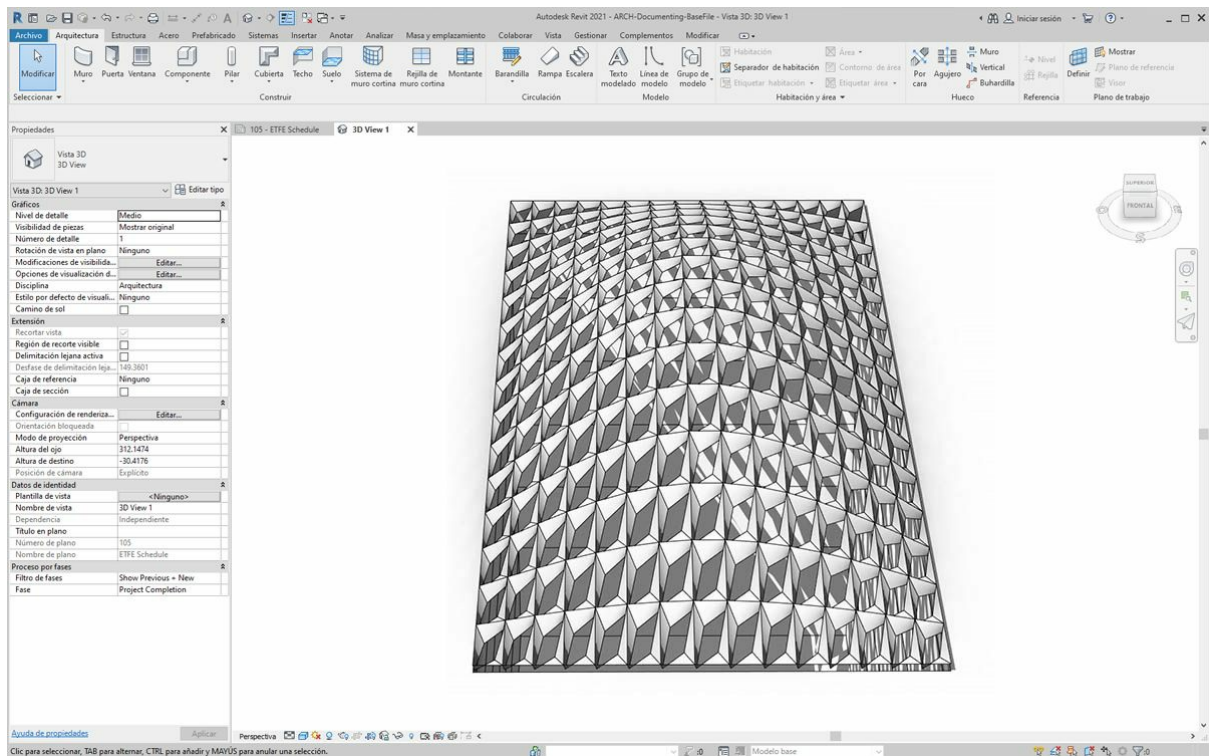


1. Vamos a crear una gran cantidad de ejemplares de geometría en Revit, por lo que debe asegurarse de que el solucionador de Dynamo se encuentre en el modo "Manual".
2. Añada un nodo *Family Types* al lienzo y seleccione "ROOF-PANEL-4PT".
3. Añada un nodo *AdaptiveComponent.ByPoints* al lienzo, conecte *Panel Pts* de la salida de "LunchBox Quad Grid by Face" a la entrada *points*. Conecte el nodo *Family Types* a la entrada *familySymbol*.
4. Pulse *Ejecutar*. Revit tendrá que *pensar* un poco mientras se crea la geometría. Si tarda demasiado, reduzca el valor de 15 del bloque de código a un número inferior. De este modo, se reduce el número de paneles de la cubierta.

Nota: si Dynamo tarda mucho tiempo en calcular los nodos, puede usar la función "Bloquear" para poner en pausa la ejecución de operaciones de Revit mientras desarrolla el gráfico. Para obtener más información sobre el bloqueo de nodos, consulte la sección "Bloqueo" del [capítulo de sólidos](#).

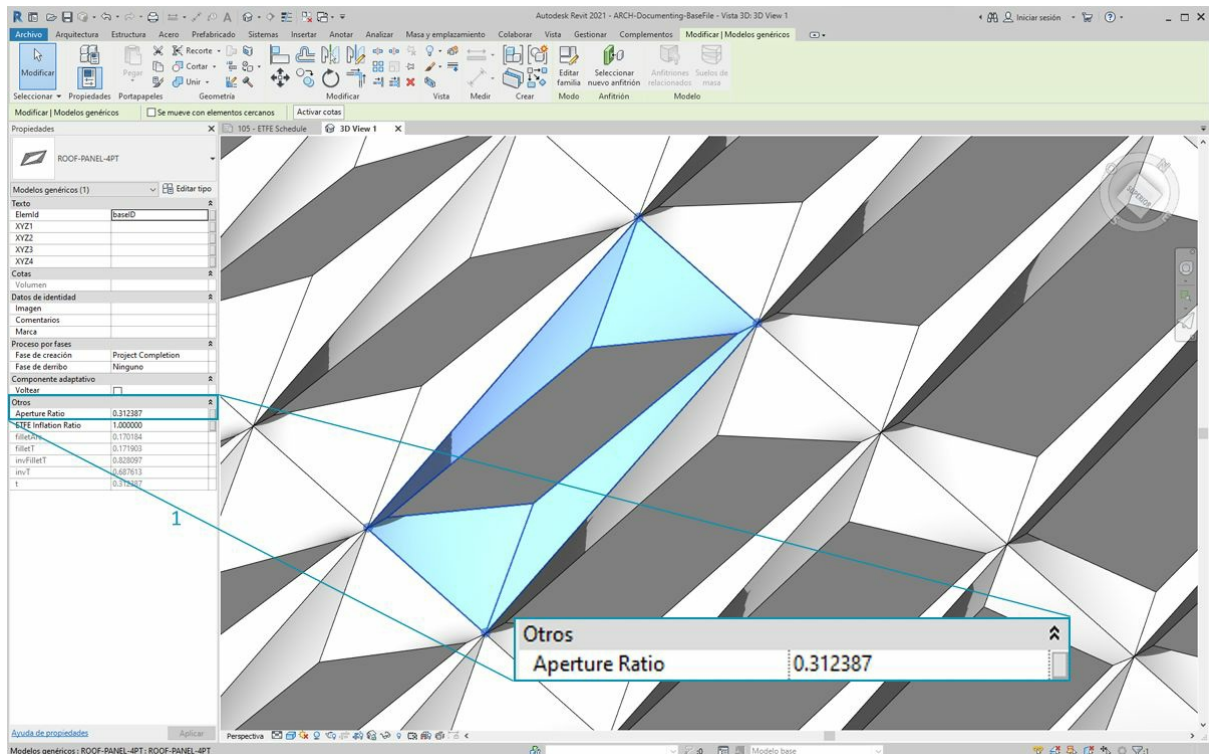


Al volver a Revit, disponemos de la matriz de paneles en la cubierta.

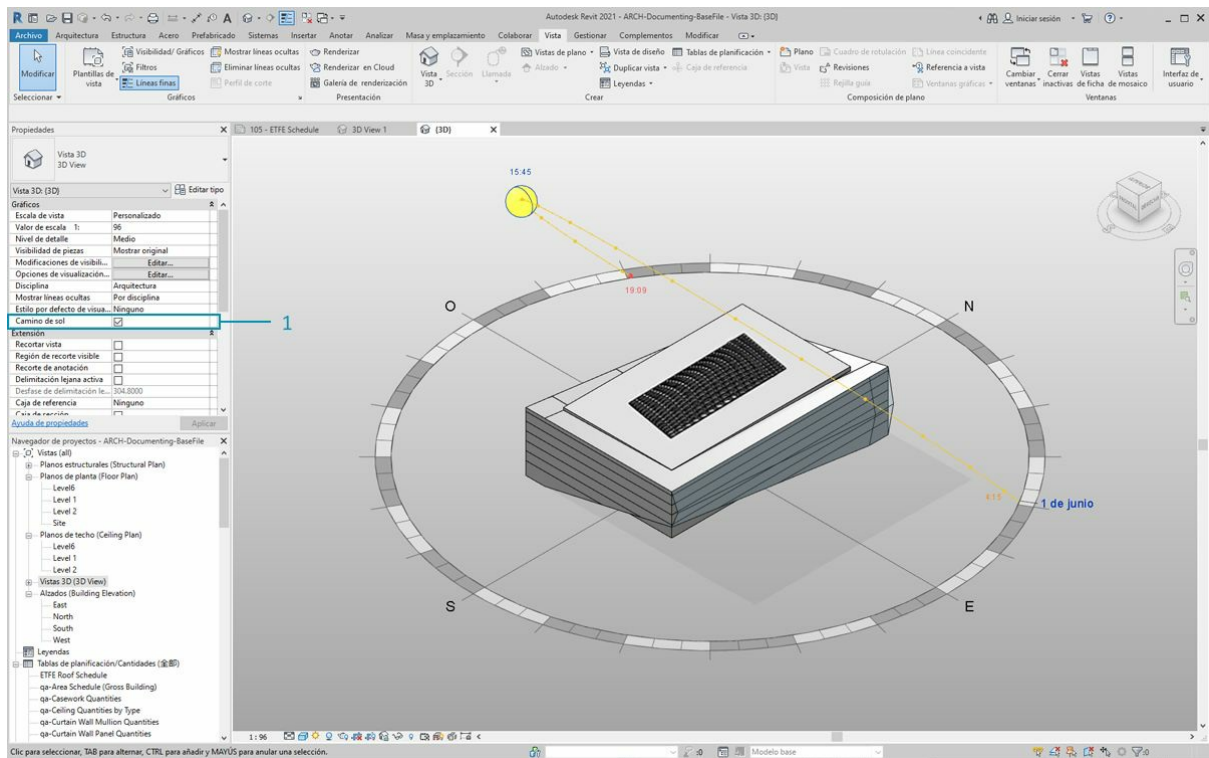


Al ampliar, podemos observar mejor las cualidades de la superficie.

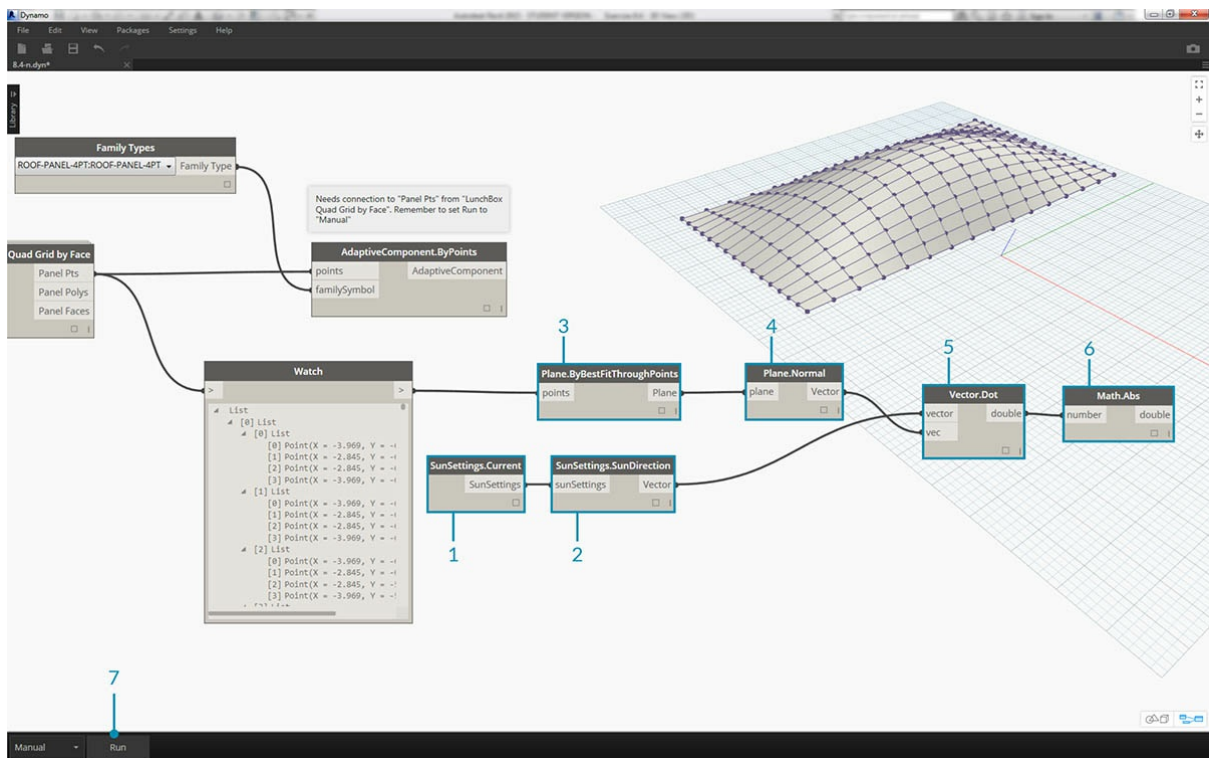
Análisis



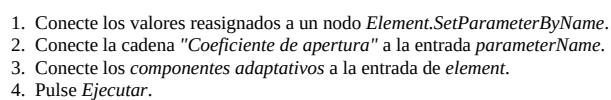
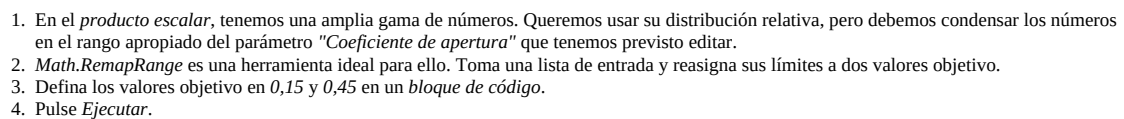
1. Continuando con el ejemplo anterior, vamos a ir un paso más allá y vamos a definir la apertura de cada panel en función de su exposición al sol. Al ampliar en Revit y seleccionar un panel, vemos que hay un parámetro denominado "Coeficiente de apertura" en la barra de propiedades. La familia se configura para que los rangos de apertura estén comprendidos aproximadamente entre 0,05 y 0,45.

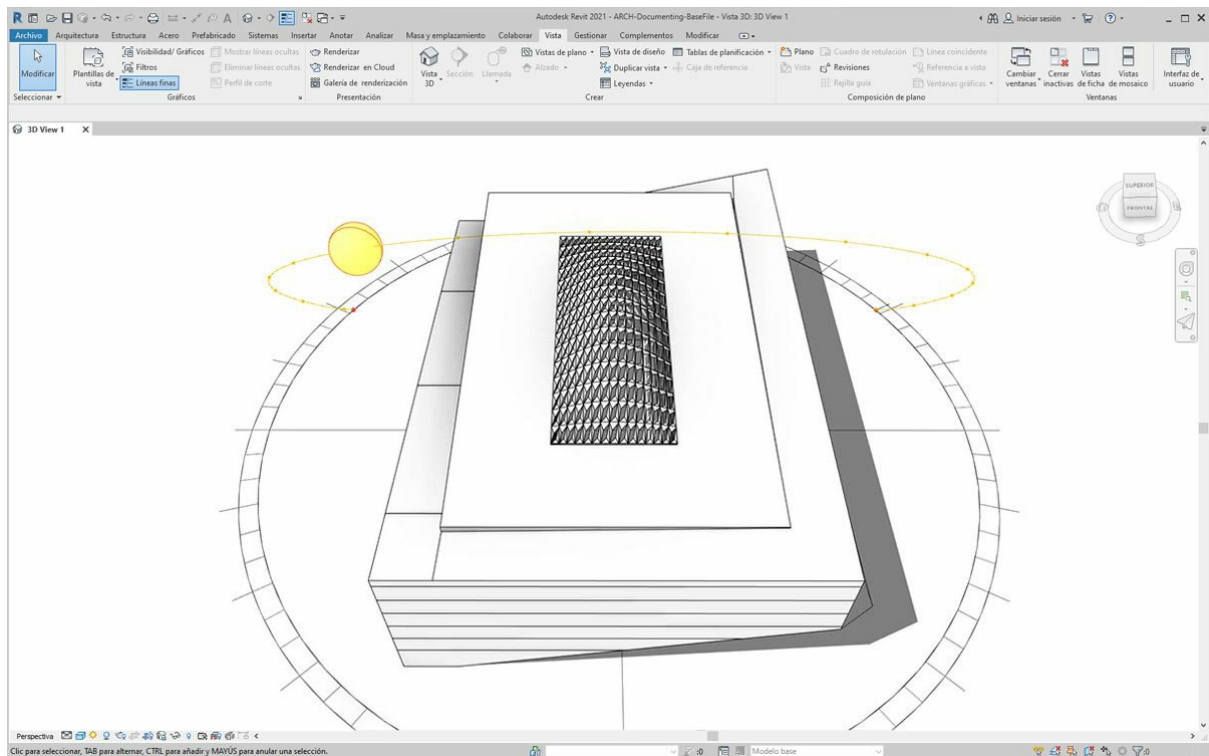


1. Si se activa el camino solar, se puede ver la ubicación actual del sol en Revit.

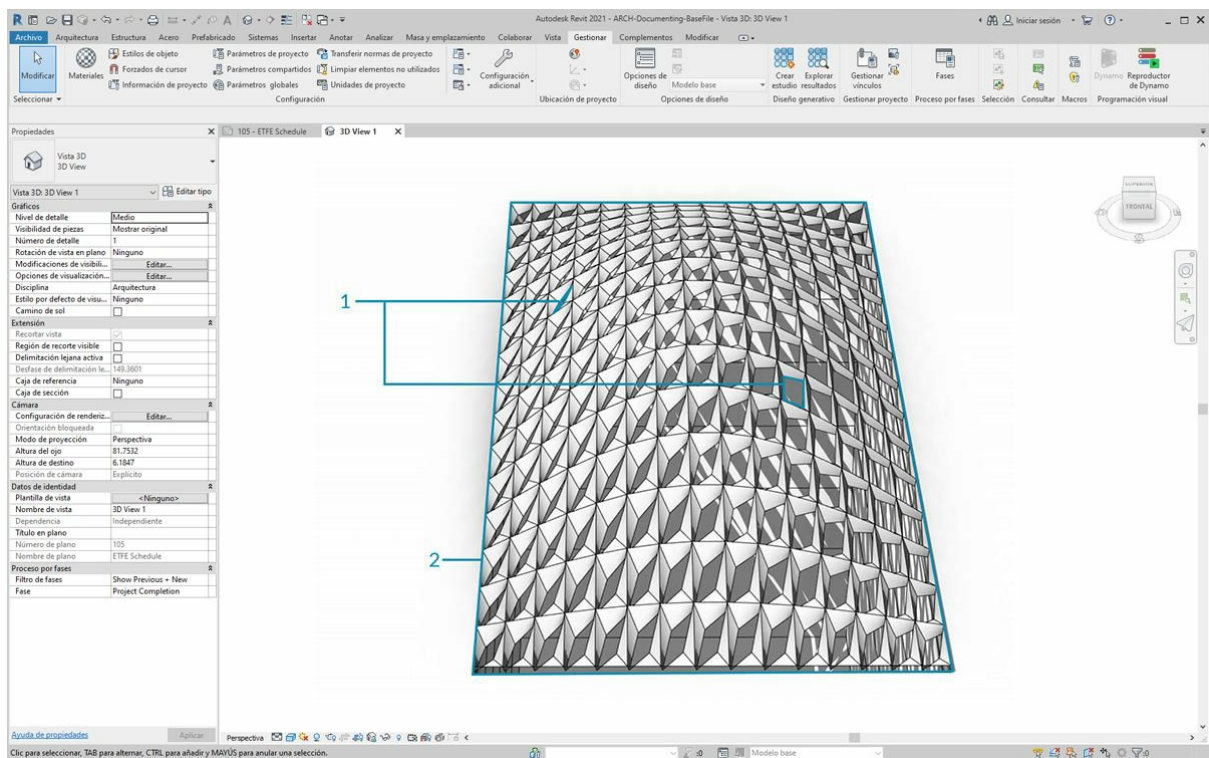


1. Podemos hacer referencia a esta ubicación del sol mediante el nodo *SunSettings.Current*.
2. Conecte la configuración del sol a *SunSettings.SunDirection* para obtener el vector solar.
3. En el elemento *Panel Pts* utilizado para crear los componentes adaptativos, utilice *Plane.ByBestFitThroughPoints* para aproximar un plano para el componente.
4. Consulte la *normal* de este plano.
5. Use el *producto escalar* para calcular la orientación solar. El producto escalar es una fórmula que determina el grado de paralelismo o antiparalelismo de dos vectores. Por tanto, tomamos la normal del plano de cada componente adaptativo y lo comparamos con el vector solar para simular aproximadamente la orientación solar.
6. Tome el *valor absoluto* del resultado. De esta forma, se garantiza que el producto escalar sea preciso si la normal del plano está orientada hacia la dirección inversa.
7. Pulse *Ejecutar*.





De vuelta en Revit, visualizamos a una distancia que nos permita determinar el efecto de la orientación solar en la apertura de los paneles ETFE.



Al ampliar, vemos que los paneles ETFE están más cerrados cuando miran hacia el sol. Nuestro objetivo es reducir el sobrecalentamiento por la exposición solar. Si quisiéramos dejar pasar más luz en función de la exposición solar, solo tendríamos que cambiar el dominio en *Math.RemapRange*.

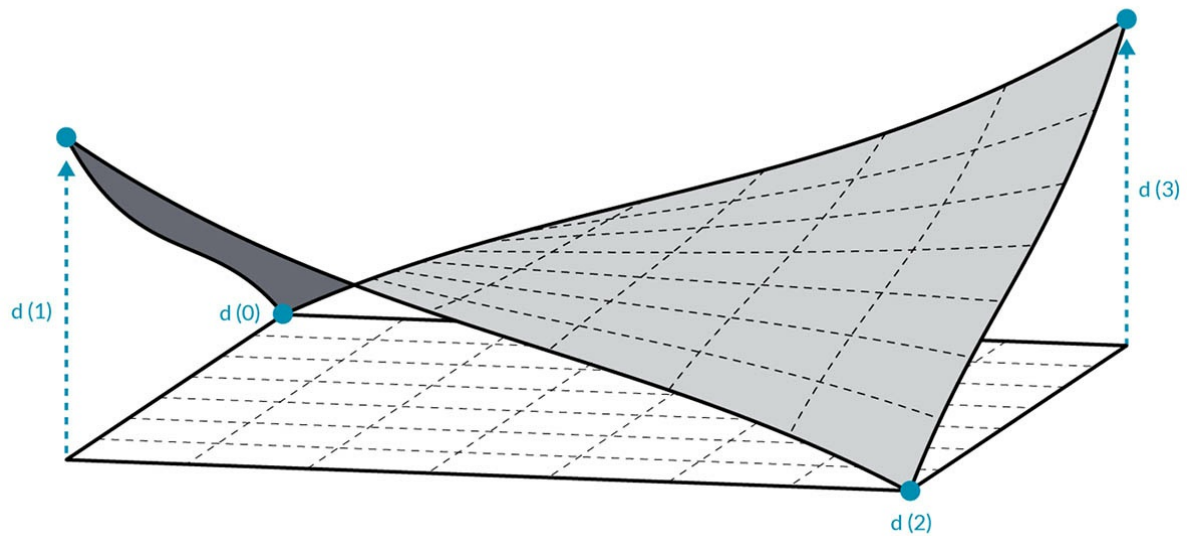
Documentación

Documentación

La edición de parámetros para la documentación sigue la misma dinámica que las lecciones aprendidas en las secciones anteriores. En esta sección, vamos a ver cómo editar parámetros que no afectan a las propiedades geométricas de un elemento, sino que preparan un archivo de Revit para la documentación.

Desviación

En el siguiente ejercicio, utilizaremos una desviación básica del nodo del plano para crear una hoja de Revit para la documentación. Cada panel de nuestra estructura de cubierta definida paramétricamente tiene un valor diferente para la desviación, y queremos invocar el rango de valores mediante el color y la planificación de los puntos adaptativos para entregarlos a un consultor, ingeniero o contratista de fachadas.



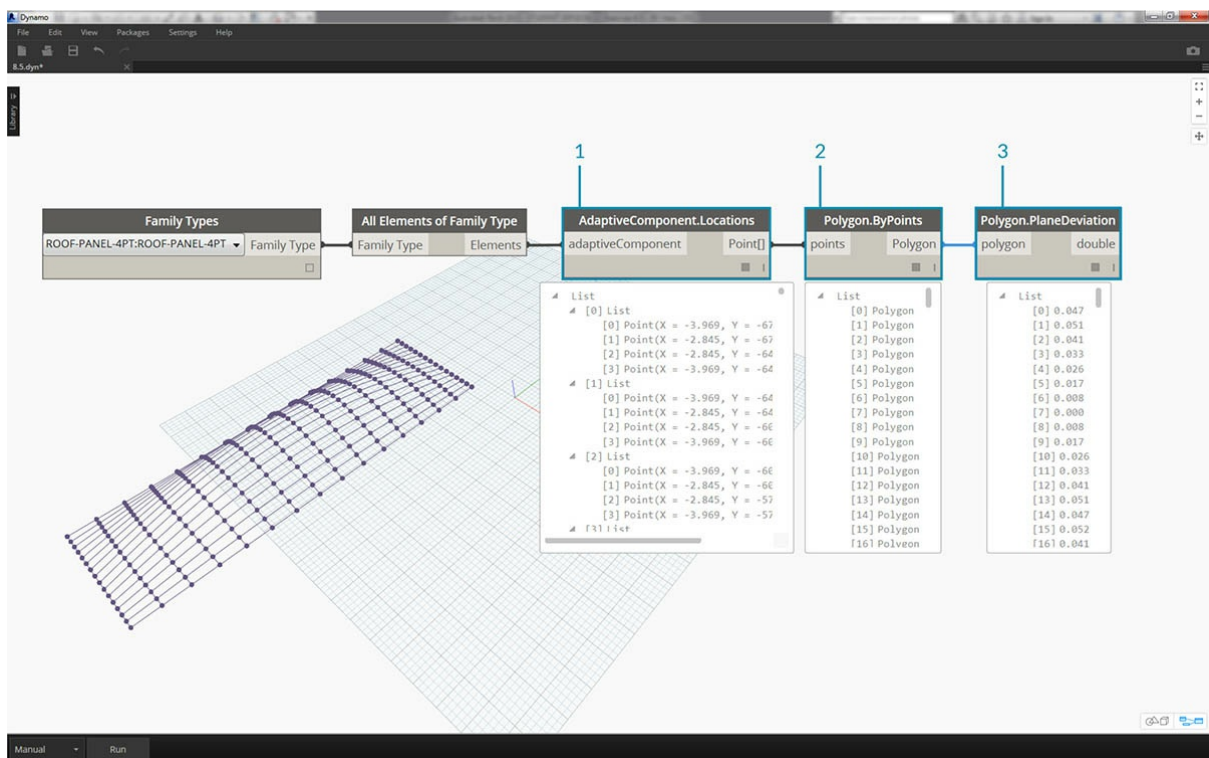
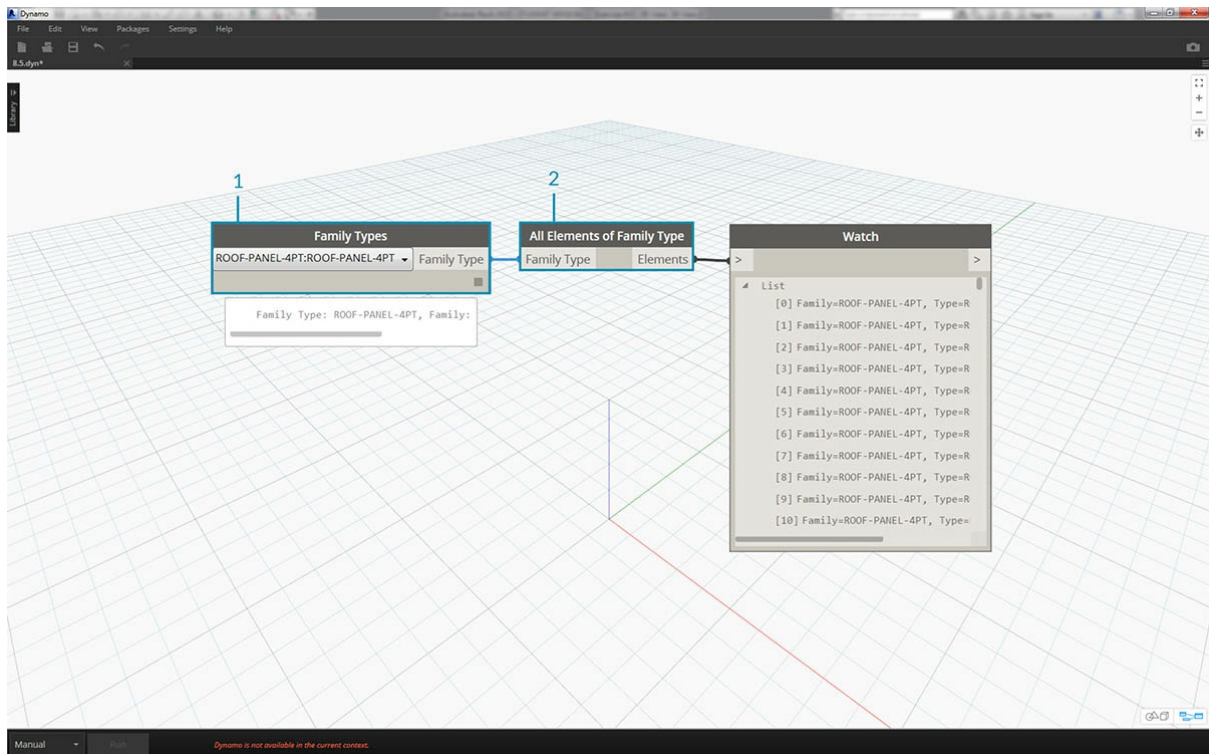
La desviación desde el nodo del plano calcula la distancia en que el conjunto de cuatro puntos varía con respecto al plano de ajuste óptimo entre ellos. Esta es una forma rápida y sencilla de estudiar la viabilidad de la construcción.

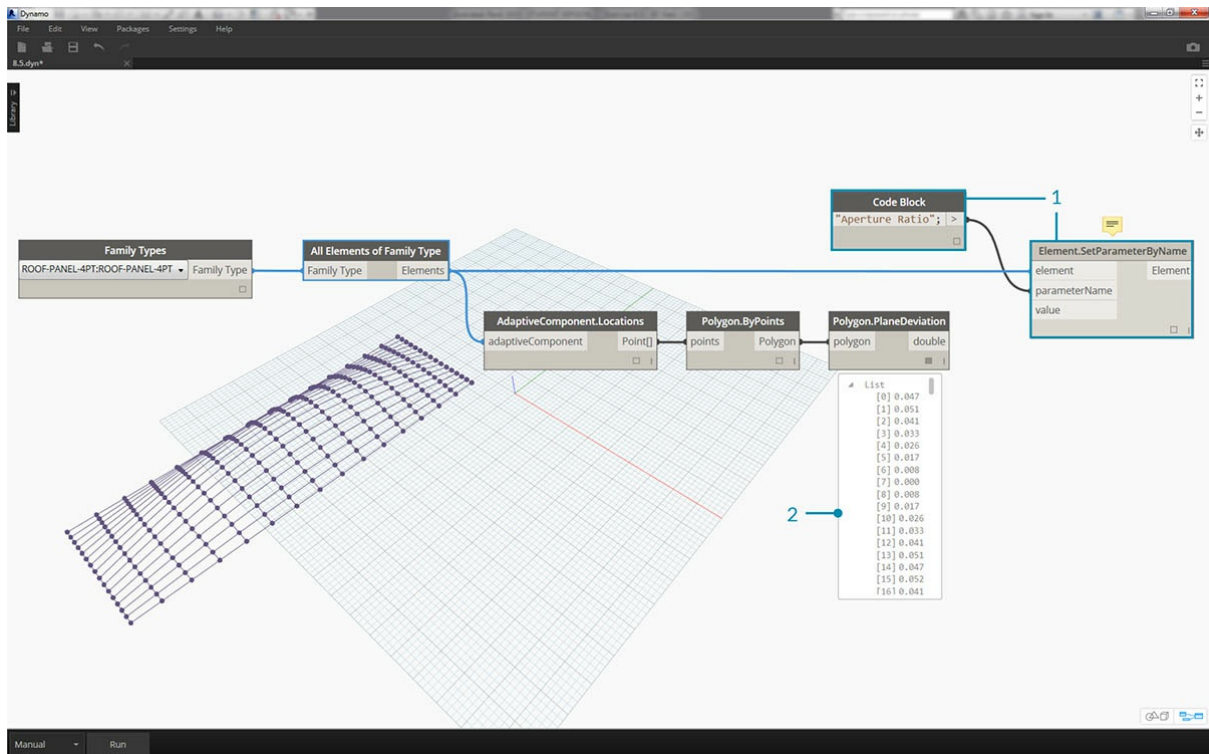
Ejercicio

Descargue los archivos de ejemplo que acompañan a este ejercicio (haga clic con el botón derecho y seleccione "Guardar enlace como..."). En el Apéndice se incluye una lista completa de los archivos de ejemplo.

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

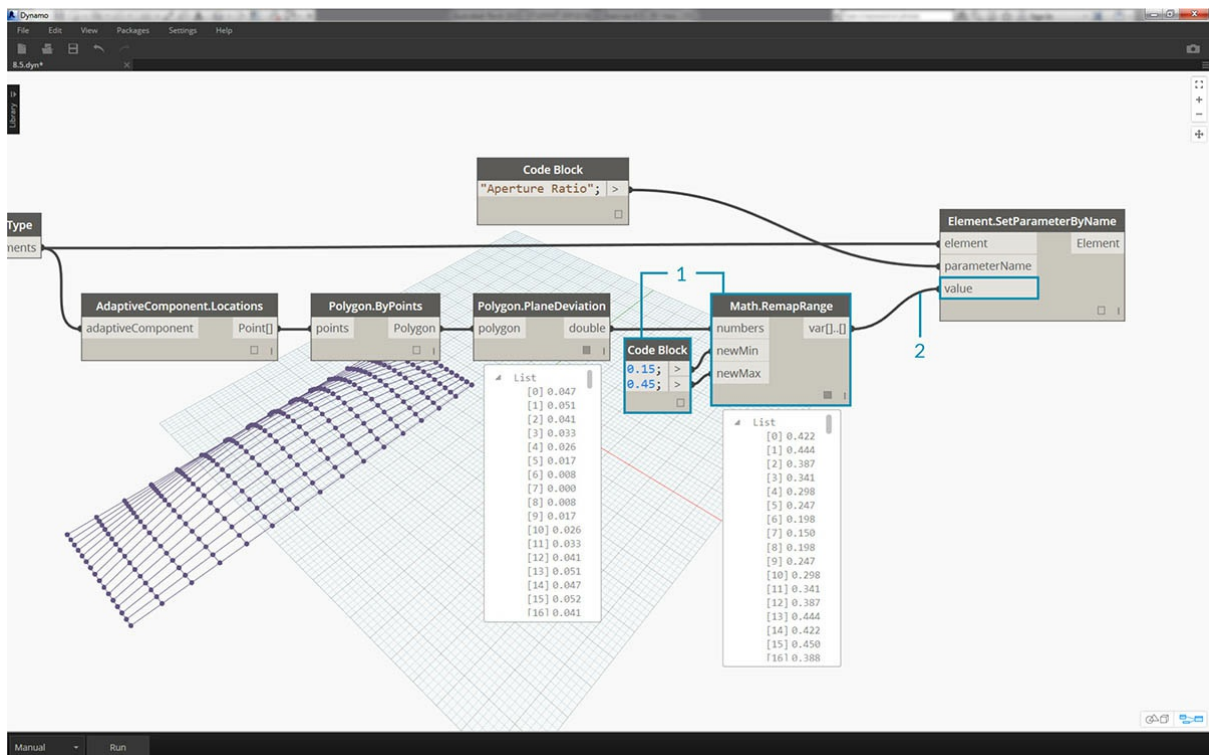
Empiece con el archivo de Revit para esta sección (o continúe desde la sección anterior). Este archivo tiene una matriz de paneles ETFE en la cubierta. Haremos referencia a estos paneles en este ejercicio.



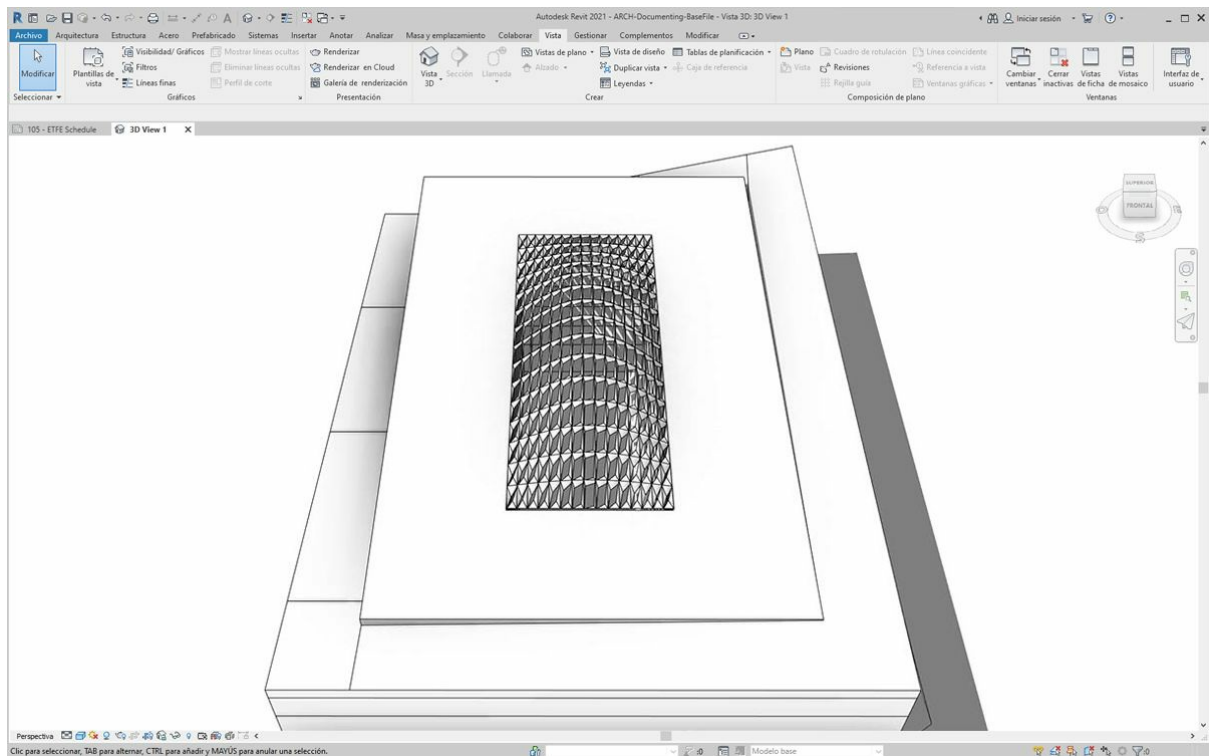


Solo por probar, como en el ejercicio anterior, vamos a configurar el *coeficiente de apertura* de cada panel en función de su desviación plana.

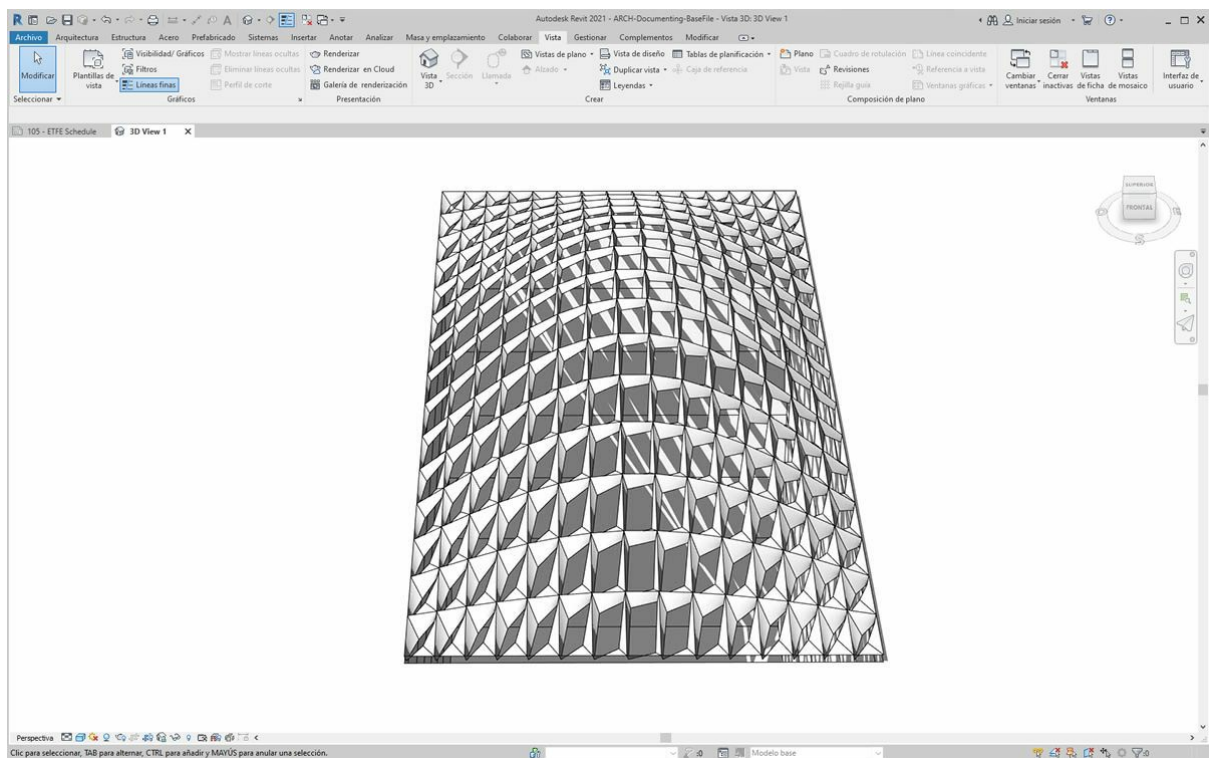
1. Añada un nodo *Element.SetParameterByName* al lienzo y conecta los componentes adaptativos a la entrada *element*. Conecte un *bloque de código* con el texto *"Aperture Ratio"* a la entrada *parameterName*.
2. No se pueden conectar directamente los resultados de la desviación a la entrada *value* porque es necesario volver a asignar los valores al rango de parámetros.



1. Con *Math.RemapRange*, reasigne los valores de desviación a un dominio entre 0,15 y 0,45.
2. Conecte estos resultados a la entrada *value* de *Element.SetParameterByName*.



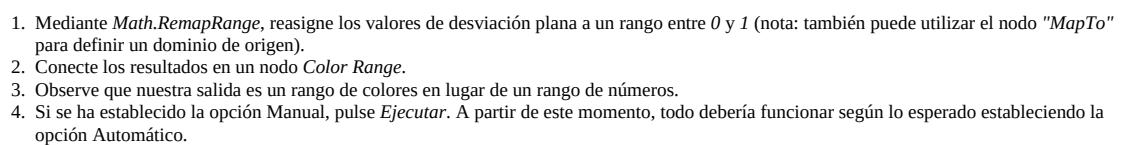
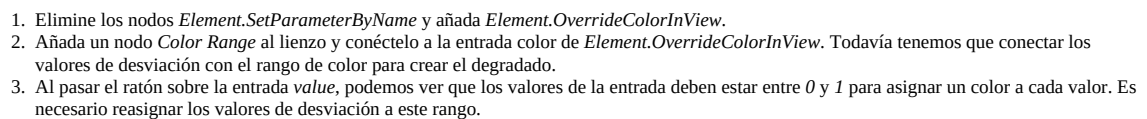
De vuelta en Revit podemos ver *aproximadamente* el cambio en la apertura a través de la superficie.

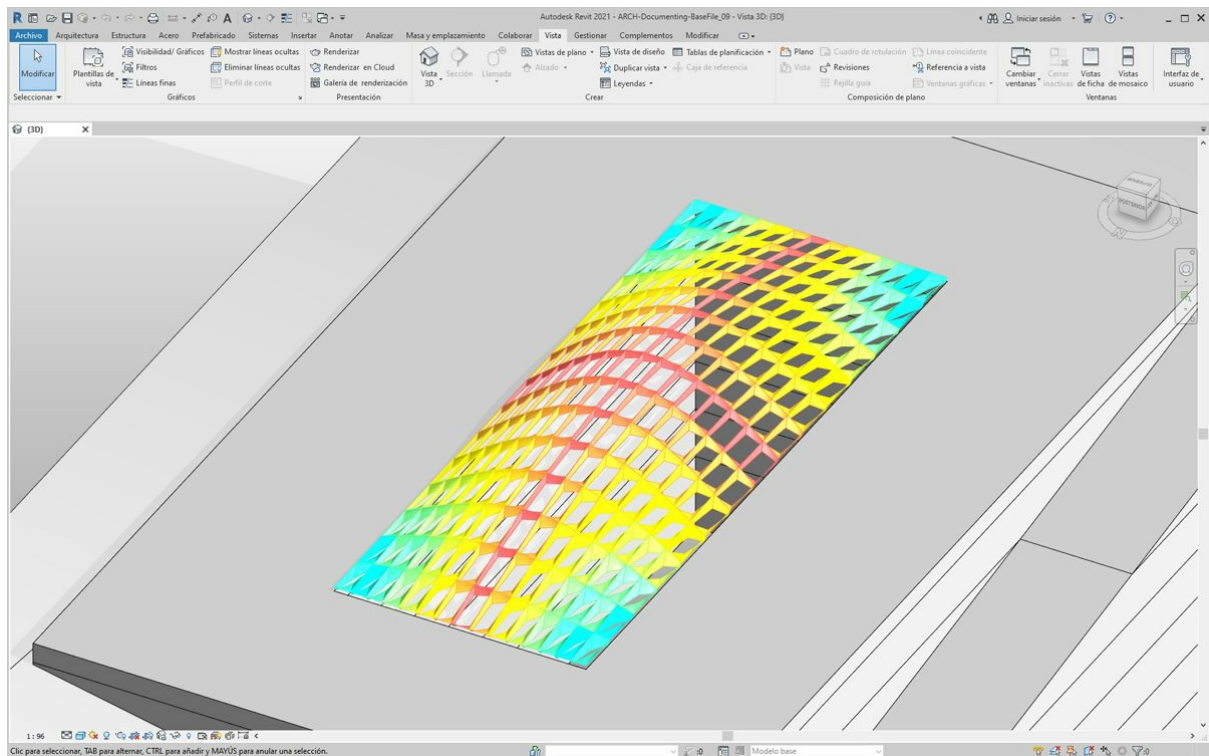


Al ampliar, resulta más evidente que los paneles cerrados están ponderados hacia las esquinas de la superficie. Las esquinas abiertas se orientan hacia la parte superior. Las esquinas representan áreas de mayor desviación, mientras que la curvatura es mínima, lo cual tiene sentido.

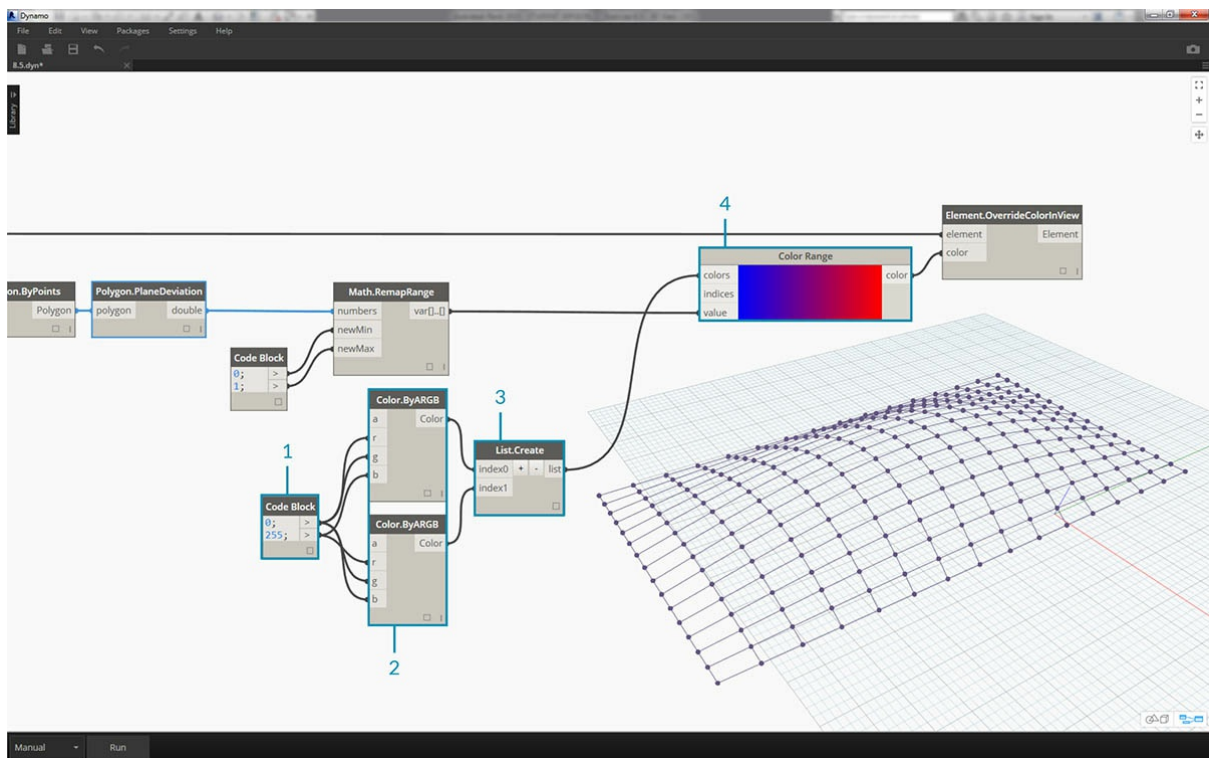
Color y documentación

La configuración de la relación de apertura no demuestra claramente la desviación de los paneles de la cubierta y, además, también se cambia la geometría del elemento real. Supongamos que solo queremos estudiar la desviación desde el punto de vista de la viabilidad de fabricación. Sería útil colorear los paneles según el rango de desviación para la documentación. Esto se puede realizar con la serie de pasos que se indican a continuación y en un proceso muy similar a los pasos anteriores.

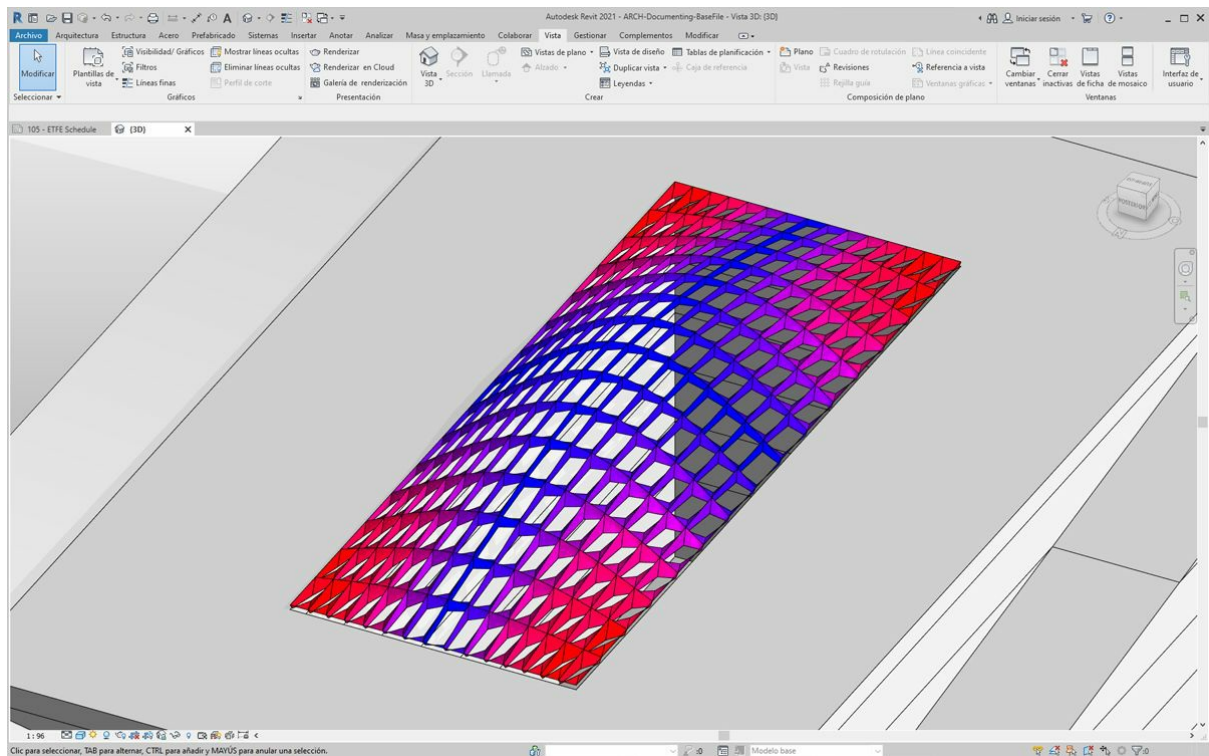




Al regresar a Revit, vemos un degradado mucho más legible que es representativo de la desviación plana basada en nuestro rango de colores. Pero, ¿qué pasa si queremos personalizar los colores? Observe que los valores de desviación mínima se representan en rojo, que parece ser lo opuesto a lo que se esperaría. Queremos que la desviación máxima sea roja y que la desviación mínima esté representada por un color más suave. Volvamos a Dynamo y arreglemos esto.

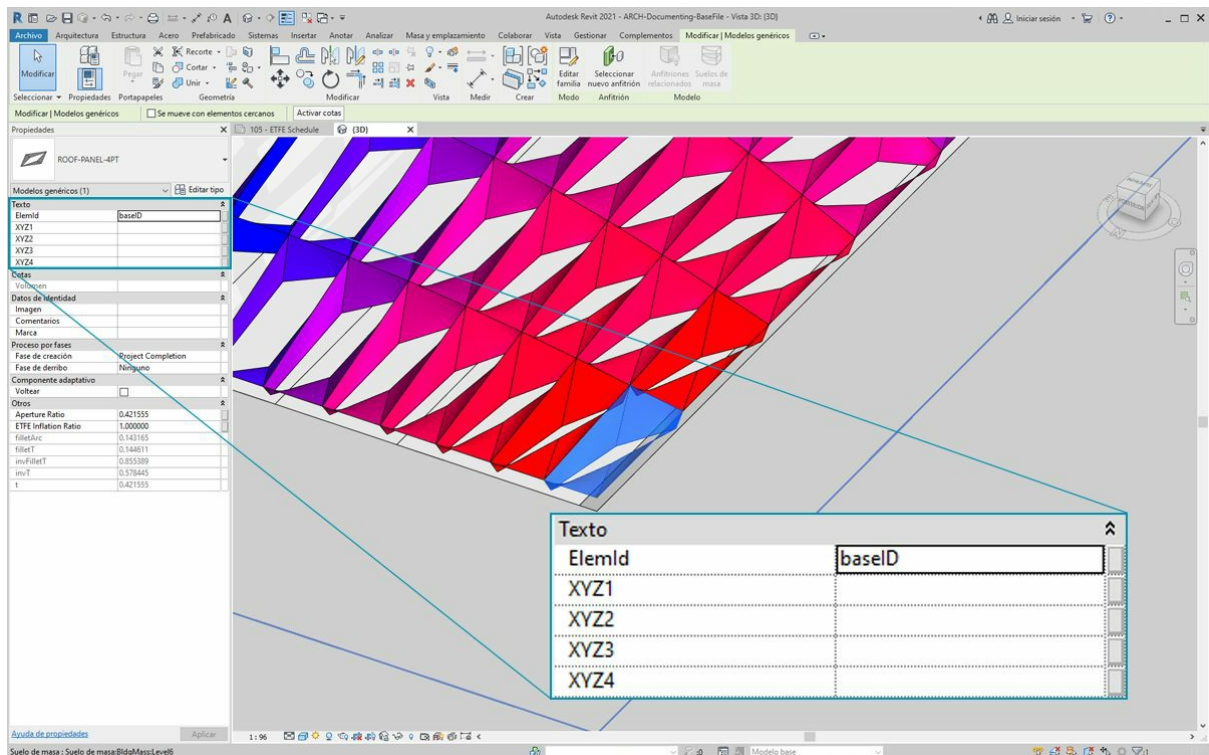


1. Mediante un **bloque de código**, añada dos números en dos líneas diferentes: 0 ; y 255 ;.
2. Cree los colores rojo y azul mediante la conexión de los valores adecuados a dos nodos **Color.ByARGB**.
3. Cree una lista a partir de estos dos colores.
4. Conecte esta lista a la entrada **colors** del nodo **Color.Range** y observe cómo se actualiza el rango de colores personalizado.

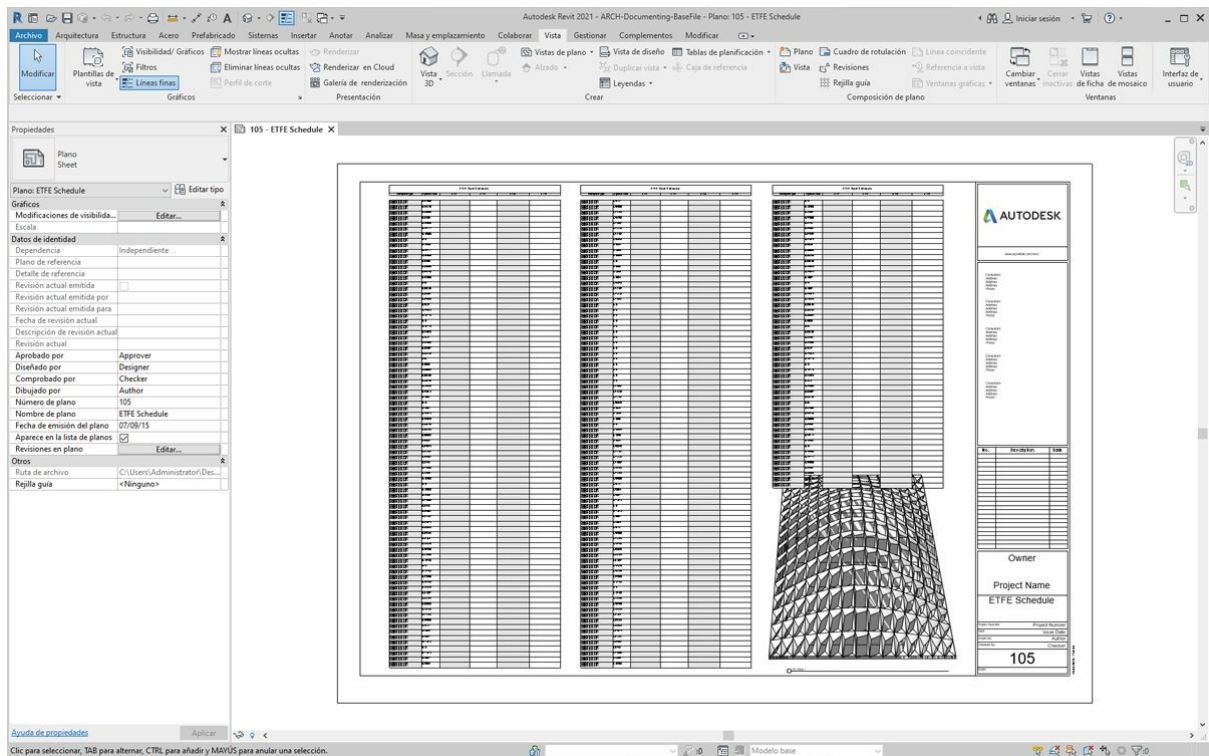


De vuelta en Revit, ahora podemos entender mejor las áreas de máxima desviación en las esquinas. Recuerde que este nodo es para modificar un color en una vista, por lo que puede resultar muy útil si tenemos una hoja concreta en el conjunto de dibujos que se centra en un tipo de análisis concreto.

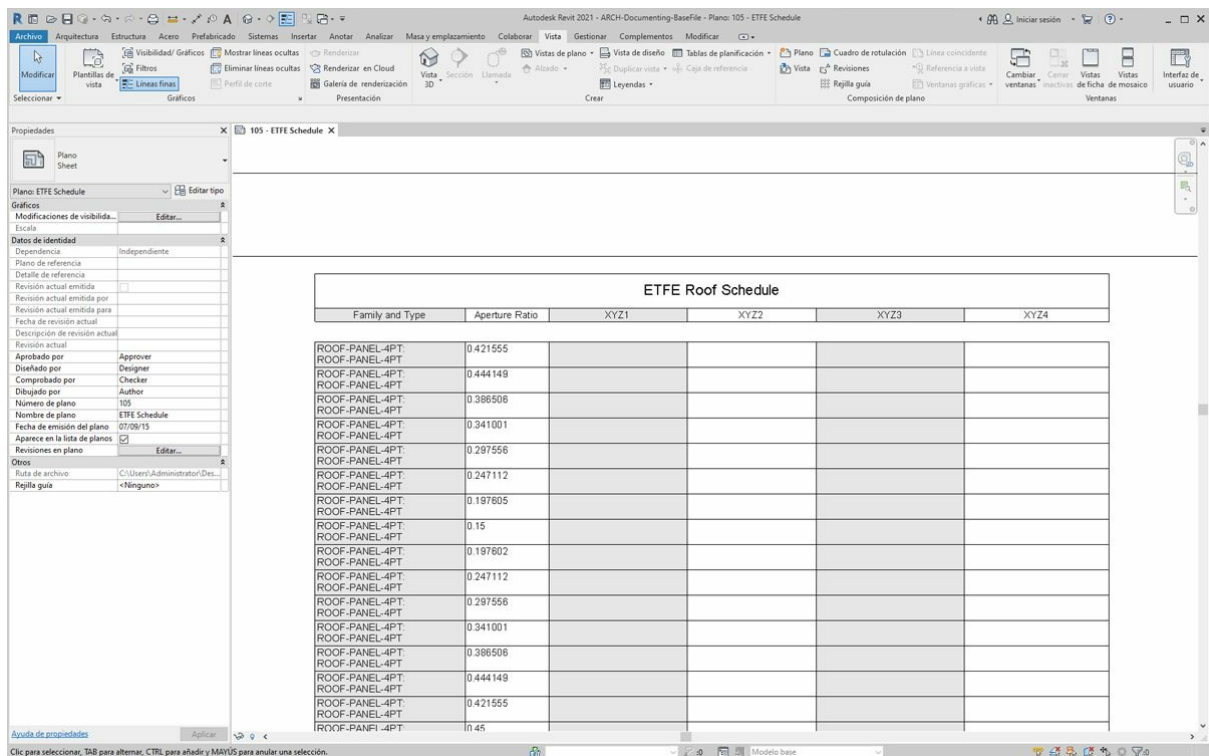
Planificación



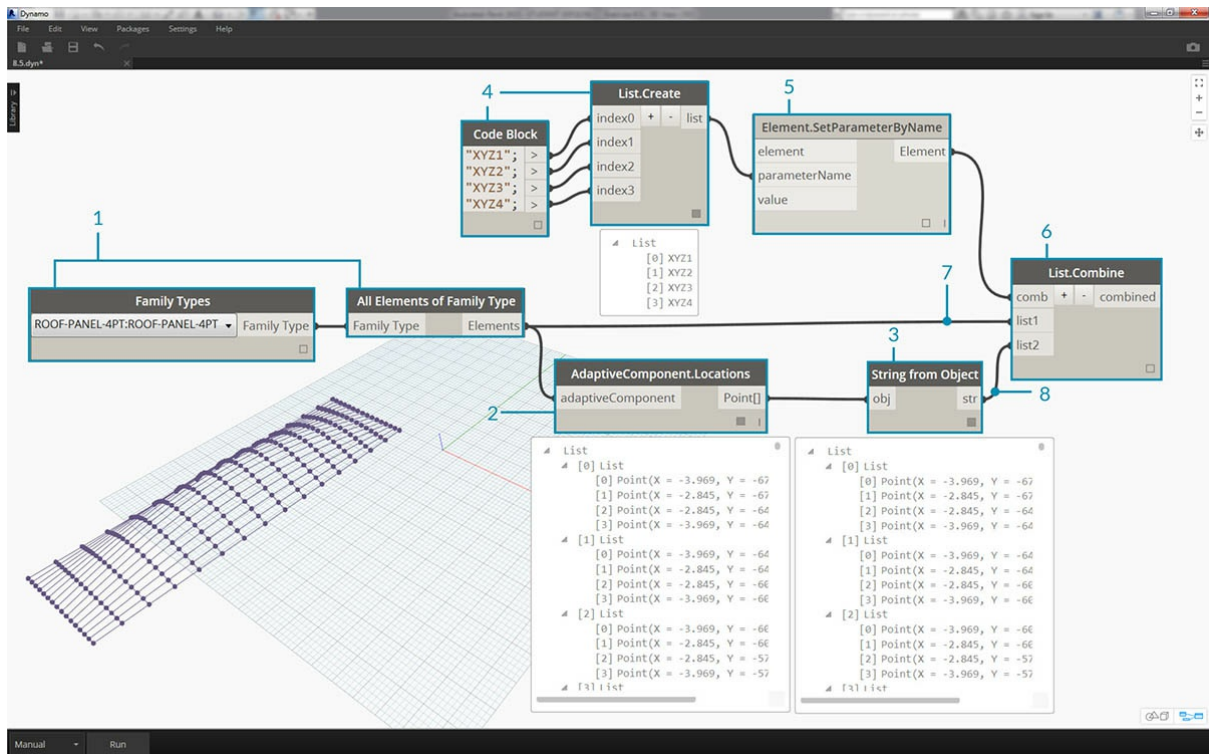
1. Al seleccionar un panel ETFE en Revit, vemos que hay cuatro parámetros de ejemplo: XYZ1, XYZ2, XYZ3 y XYZ4. Todos están en blanco después de crearlos. Se trata de parámetros basados en texto y necesitan valores. Utilizaremos Dynamo para escribir las ubicaciones de puntos adaptativos en cada parámetro. Esto aumenta la interoperabilidad si es necesario enviar la geometría a un ingeniero o consultor de fachadas.



En un plano de ejemplo, tenemos una tabla de planificación grande y vacía. Los parámetros XYZ son parámetros compartidos en el archivo de Revit, lo que nos permite añadirlos a la tabla de planificación.

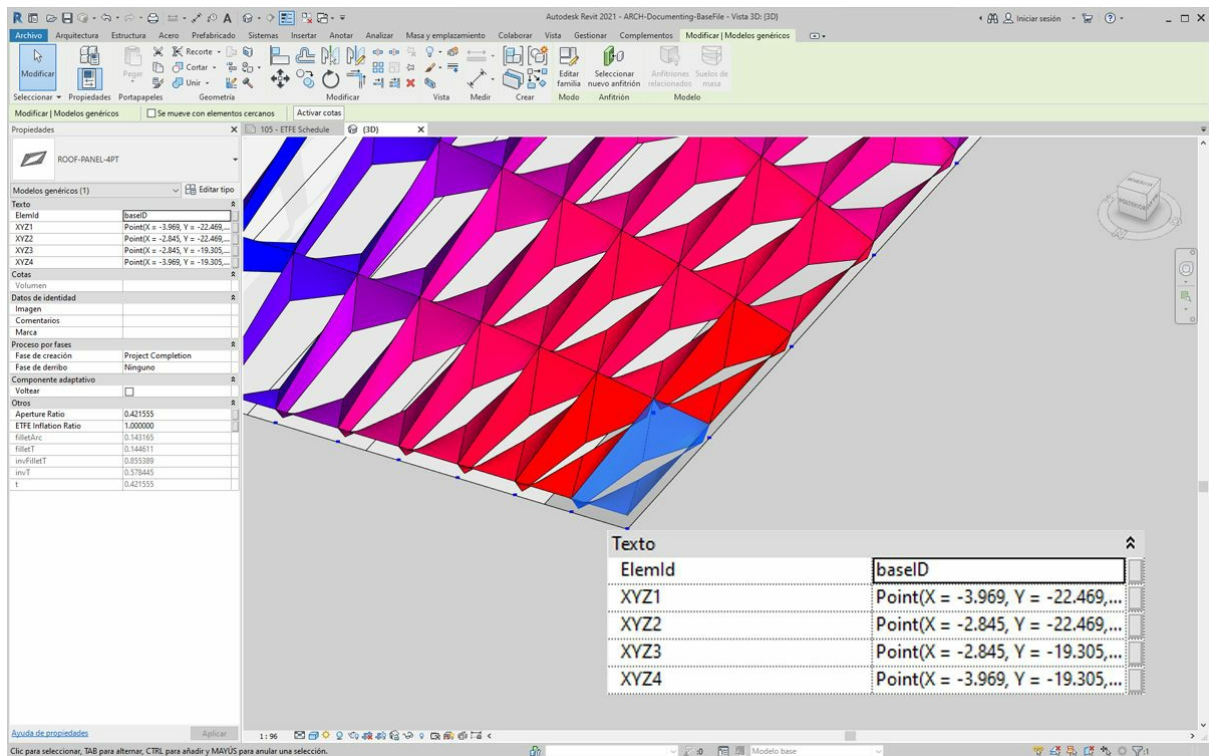


Al ampliar, vemos que los parámetros XYZ aún no se han rellenado. Revit se ocupa de los dos primeros parámetros.

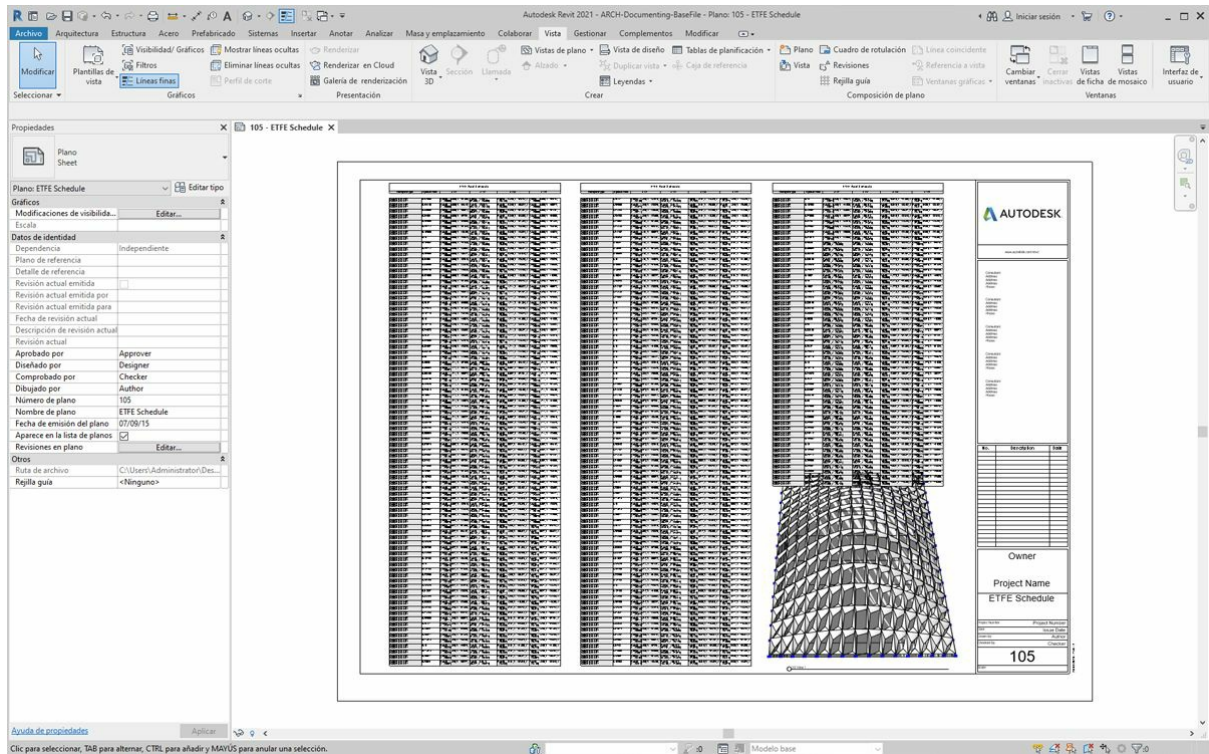


Para escribir estos valores, realizaremos una operación de lista compleja. El gráfico en sí es sencillo, pero los conceptos se hacen más complejos a partir de la asignación de lista, como se explica en el capítulo sobre las listas.

1. Seleccione todos los componentes adaptativos con dos nodos.
2. Extraiga la ubicación de cada punto con *AdaptiveComponent.Locations*.
3. Convierta estos puntos en cadenas. Recuerde que el parámetro se basa en texto, por lo que es necesario introducir el tipo de datos correcto.
4. Cree una lista de las cuatro cadenas que definen los parámetros que se van a cambiar: XYZ1, XYZ2, XYZ3 y XYZ4.
5. Conecte esta lista a la entrada *parameterName* de *Element.SetParameterByName*.
6. Conecte *Element.SetParameterByName* en la entrada *combinator* de *List.Combine*.
7. Conecte los componentes adaptativos a *list1*.
8. Conecte *String from Object* a *list2*.
9. Estamos asignando listas porque vamos a escribir cuatro valores para cada elemento, lo que crea una estructura de datos compleja. El nodo *List.Combine* define una operación un escalón más abajo en la jerarquía de datos. Por este motivo, las entradas *element* y *value* se dejan en blanco. *List.Combine* conecta las sublistas de sus entradas a las entradas vacías de *List.SetParameterByName* en función del orden en el que están conectadas.



Al seleccionar un panel en Revit, ahora vemos que hay valores de cadena para cada parámetro. En términos realistas, crearíamos un formato más sencillo para escribir un punto (X,Y,Z). Esto se puede realizar con operaciones de cadena en Dynamo, pero vamos a omitir este paso para mantenernos dentro del ámbito de este capítulo.



Autodesk Revit 2021 - ARCH-Documenting-BaseFile - Plano: 105 - ETFE Schedule

Architectural Structure Access Prefabricated Systems Insert Annotate Mass and Placement Collaborate Views Manage Components Modify

Modificar Plantillas de vista Filtros Eliminar líneas ocultas Perfil de corte Galería de renderización Presentación Vista 3D Recorte Luminaria Alzado Duplicar vista Caja de referencia Leyendas Crear

Plano Cuadro de rotulación Línea coincidente Revisión Referencia a vista Ventanas gráficas Composición de plano

Plano Sheet

Plano: ETFE Schedule Editar tipo

Gráficos Modificaciones de visibilidad Editar...

Escala

Datos de identidad

Dependencia Independiente

Plano de referencia

Detalle de referencia

Revisión actual emitida

Revisión actual emitida por

Revisión actual emitida para

Fecha de revisión actual

Descripción de revisión actual

Revisión actual

Aprobado por Approver

Diseñado por Designer

Comprobado por Checker

Dibujado por Author

Número de plano 105

Nombre de plano ETFE Schedule

Fecha de emisión del plano 07/09/15

Aparece en la lista de planos

Revisiones en plano Editar...

Notas

Ruta de archivo C:\Users\Administrator\Desktop

Rutina guía <Ninguno>

Aplicar

Click para seleccionar, TAB para alternar, CTRL para añadir y MAYUS para anular una selección.

Modelo base

ETFE Roof Schedule

Family and Type	Aperture Ratio	XYZ1	XYZ2	XYZ3	XYZ4
ROOF-PANEL-4PT	0.421555	Point(X = -3.969, Y = -67.305, Z = 25.000)	Point(X = -2.845, Y = -67.305, Z = 25.000)	Point(X = -2.845, Y = -64.142, Z = 25.188)	Point(X = -3.969, Y = -64.142, Z = 25.000)
ROOF-PANEL-4PT	0.444149	Point(X = -3.969, Y = -64.142, Z = 25.000)	Point(X = -2.845, Y = -64.142, Z = 25.188)	Point(X = -2.845, Y = -60.972, Z = 25.386)	Point(X = -3.969, Y = -60.972, Z = 25.000)
ROOF-PANEL-4PT	0.386506	Point(X = -3.969, Y = -60.972, Z = 25.000)	Point(X = -2.845, Y = -60.972, Z = 25.386)	Point(X = -2.845, Y = -57.791, Z = 25.570)	Point(X = -3.969, Y = -57.791, Z = 25.000)
ROOF-PANEL-4PT	0.341001	Point(X = -3.969, Y = -57.791, Z = 25.000)	Point(X = -2.845, Y = -57.791, Z = 25.570)	Point(X = -2.845, Y = -54.595, Z = 25.719)	Point(X = -3.969, Y = -54.595, Z = 25.000)
ROOF-PANEL-4PT	0.297556	Point(X = -3.969, Y = -54.595, Z = 25.000)	Point(X = -2.845, Y = -54.595, Z = 25.719)	Point(X = -2.845, Y = -51.384, Z = 25.841)	Point(X = -3.969, Y = -51.384, Z = 25.000)
ROOF-PANEL-4PT	0.247112	Point(X = -3.969, Y = -51.384, Z = 25.000)	Point(X = -2.845, Y = -51.384, Z = 25.841)	Point(X = -2.845, Y = -48.159, Z = 25.926)	Point(X = -3.969, Y = -48.159, Z = 25.000)
ROOF-PANEL-4PT	0.197805	Point(X = -3.969, Y = -48.159, Z = 25.000)	Point(X = -2.845, Y = -48.159, Z = 25.926)	Point(X = -2.845, Y = -44.924, Z = 25.969)	Point(X = -3.969, Y = -44.924, Z = 25.000)
ROOF-PANEL-4PT	0.15	Point(X = -3.969, Y = -44.924, Z = 25.000)	Point(X = -2.845, Y = -44.924, Z = 25.969)	Point(X = -2.845, Y = -41.686, Z = 25.989)	Point(X = -3.969, Y = -41.686, Z = 25.000)
ROOF-PANEL-4PT	0.197802	Point(X = -3.969, Y = -41.686, Z = 25.000)	Point(X = -2.845, Y = -41.686, Z = 25.969)	Point(X = -2.845, Y = -38.452, Z = 25.926)	Point(X = -3.969, Y = -38.452, Z = 25.000)
ROOF-PANEL-4PT	0.247112	Point(X = -3.969, Y = -38.452, Z = 25.000)	Point(X = -2.845, Y = -38.452, Z = 25.926)	Point(X = -2.845, Y = -35.227, Z = 25.841)	Point(X = -3.969, Y = -35.227, Z = 25.000)
ROOF-PANEL-4PT	0.297556	Point(X = -3.969, Y = -35.227, Z = 25.000)	Point(X = -2.845, Y = -35.227, Z = 25.841)	Point(X = -2.845, Y = -32.016, Z = 25.719)	Point(X = -3.969, Y = -32.016, Z = 25.000)
ROOF-PANEL-4PT	0.341001	Point(X = -3.969, Y = -32.016, Z = 25.000)	Point(X = -2.845, Y = -32.016, Z = 25.719)	Point(X = -2.845, Y = -28.820, Z = 25.570)	Point(X = -3.969, Y = -28.820, Z = 25.000)
ROOF-PANEL-4PT	0.386506	Point(X = -3.969, Y = -28.820, Z = 25.000)	Point(X = -2.845, Y = -28.820, Z = 25.570)	Point(X = -2.845, Y = -25.639, Z = 25.386)	Point(X = -3.969, Y = -25.639, Z = 25.000)
ROOF-PANEL-4PT	0.444149	Point(X = -3.969, Y = -25.639, Z = 25.000)	Point(X = -2.845, Y = -25.639, Z = 25.386)	Point(X = -2.845, Y = -22.469, Z = 25.188)	Point(X = -3.969, Y = -22.469, Z = 25.000)
ROOF-PANEL-4PT	0.421555	Point(X = -3.969, Y = -22.469, Z = 25.000)	Point(X = -2.845, Y = -22.469, Z = 25.188)	Point(X = -2.845, Y = -19.305, Z = 25.000)	Point(X = -3.969, Y = -19.305, Z = 25.000)

Cada panel ETFE tiene ahora las coordenadas XYZ escritas para cada punto adaptativo, con lo que se representan las esquinas de cada panel para la fabricación.

Diccionarios en Dynamo

Diccionarios en Dynamo

Los diccionarios representan una recopilación de datos relacionados con otro elemento de datos conocido como clave. Los diccionarios muestran la capacidad de buscar, suprimir e insertar datos en una colección.

Básicamente, podemos considerar un diccionario como una manera realmente inteligente de buscar algo.

Aunque la función de diccionario lleva disponible en Dynamo durante algún tiempo, Dynamo 2.0 presenta una nueva forma de administrar este tipo de datos.

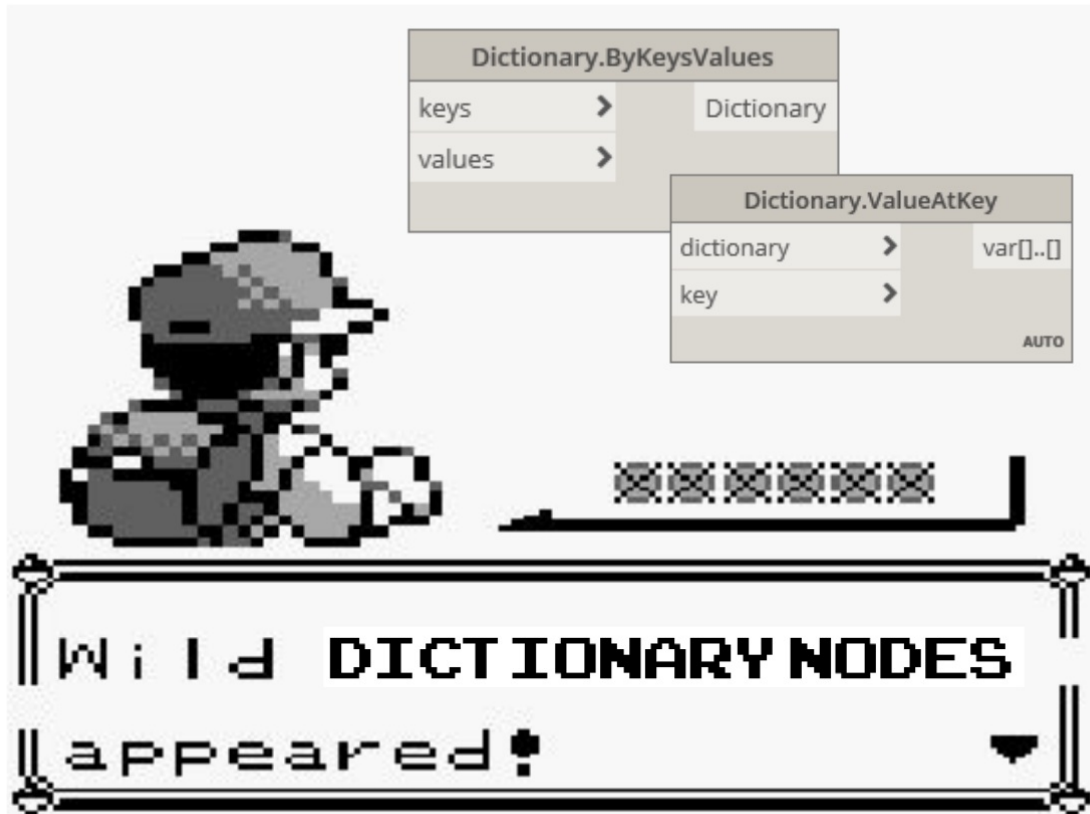


Imagen por cortesía de XVItysecondvit.com

¿Qué es un diccionario?

Diccionarios

Dynamo 2.0 introduce el concepto de separación del tipo de datos del diccionario del tipo de datos de lista. Esta novedad puede conllevar algunos cambios importantes en la forma de crear y trabajar con datos en los flujos de trabajo. Antes de la versión 2.0, los diccionarios y las listas se combinaban como un tipo de datos. En resumen, las listas eran en realidad diccionarios con claves de enteros.

- **¿Qué es un diccionario?**

Un diccionario es un tipo de datos compuesto por una colección de pares de clave-valor en los que cada clave es exclusiva para cada colección. Un diccionario no tiene orden y, básicamente, se pueden "buscar elementos" mediante una clave en lugar de un valor de índice como en una lista. *En Dynamo 2.0, las claves solo pueden ser cadenas.*

- **¿Qué es una lista?**

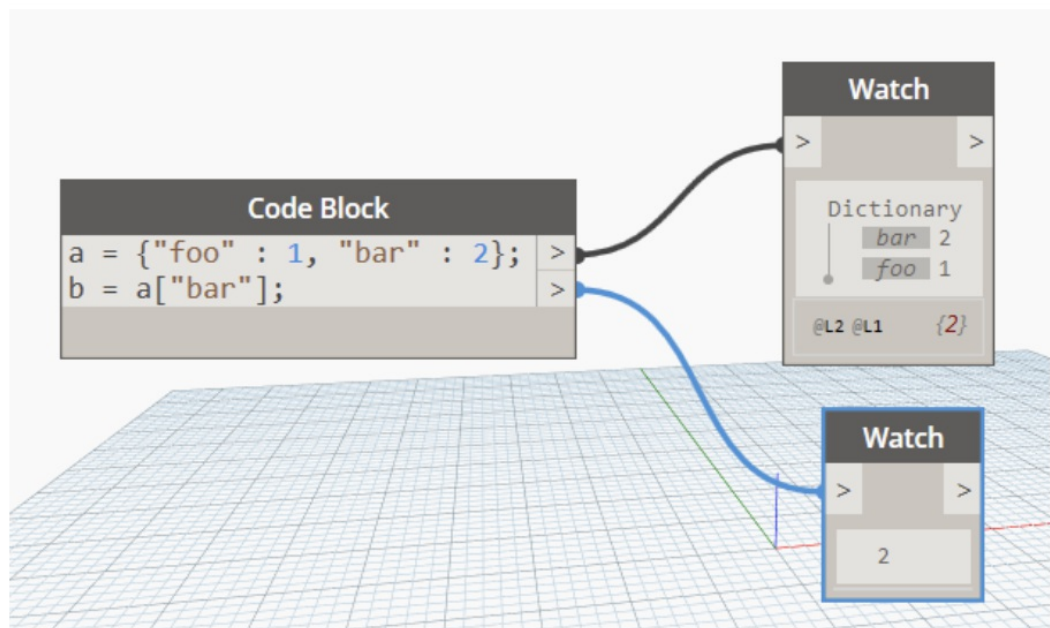
Una lista es un tipo de datos compuesto por un conjunto de valores ordenados. En Dynamo, las listas utilizan enteros como valores de índice.

- **¿Por qué se ha realizado este cambio y en qué me afecta?**

La separación de los diccionarios y las listas convierte a los diccionarios en componentes de primera clase que se pueden utilizar para almacenar y buscar valores de forma rápida y sencilla sin necesidad de recordar un valor de índice o mantener una estructura de listas estricta a lo largo del flujo de trabajo. Durante las pruebas realizadas por los usuarios, hemos detectado una reducción considerable del tamaño del gráfico cuando se utilizaron los diccionarios en lugar de varios nodos `GetItemAtIndex`.

- **¿Cuáles son los cambios?**

- Se han producido cambios en la *sintaxis* que modifican el modo en que se inicializarán y se utilizarán los diccionarios y las listas en los bloques de código.
 - Los diccionarios utilizan la siguiente sintaxis: `{clave:valor}`.
 - Las listas utilizan la siguiente sintaxis: `[valor, valor, valor]`.
- Se han introducido *nuevos nodos* en la biblioteca para ayudarle a crear, modificar y consultar diccionarios.
- Las listas creadas con bloques de código 1.x se migrarán automáticamente al cargar la secuencia de comandos a la nueva sintaxis de lista que utiliza corchetes `[ ]` en lugar de llaves `{ }`



- **¿En qué me afecta? ¿Para qué se utilizan?**

En las ciencias informáticas, los diccionarios, al igual que las listas, son colecciones de objetos. Mientras que las listas se encuentran en un orden específico, los diccionarios son colecciones *sin ordenar*. No dependen de números secuenciales (índices), sino de *claves*.

En la imagen siguiente, se muestra un posible caso de uso de un diccionario. A menudo, los diccionarios se utilizan para relacionar dos segmentos de datos que podrían no tener una correlación directa. En este caso, conectamos la versión en español de una palabra a la versión en inglés para su posterior búsqueda.

```
Code Block
//english numbers
["one","two","three","four","five"]; >
```

List

0	one
1	two
2	three
3	four
4	five

@L2 @L1 {5}

```
Code Block
//spanish numbers
["uno","dos","tres","cuatro","cinco"]; >
```

List

0	uno
1	dos
2	tres
3	cuatro
4	cinco

@L2 @L1 {5}

Build a dictionary to relate the two pieces of data

Dictionary.ByKeysValues	
keys	dictionary
values	AUTO

Get the value given the key

Dictionary.ValueAtKey	
dictionary	value
key	AUTO

Code Block

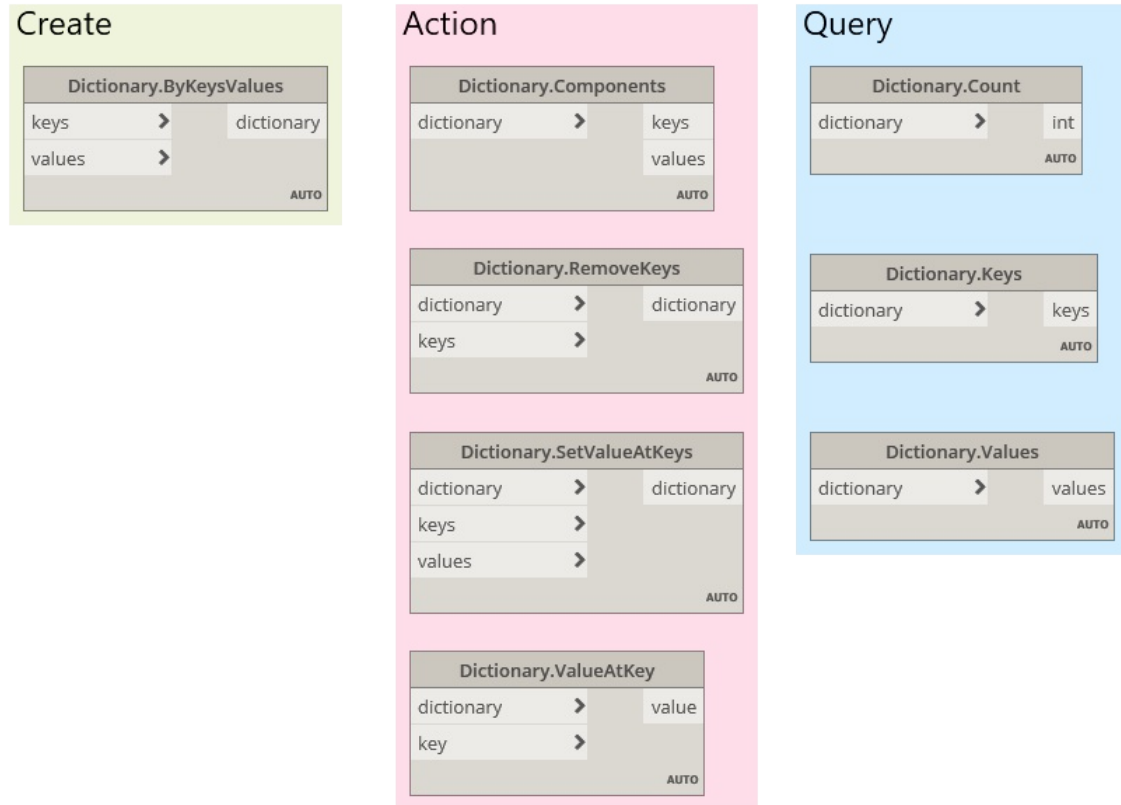
```
"four"; >
```

cuatro

Usos de los nodos

Nodos de diccionario

Dynamo 2.0 presenta una gran variedad de nodos de diccionario para nuestro uso. Esto incluye los nodos de *creación*, *acción* y *consulta*.



- **Dictionary.ByKeysValues** creará un diccionario con los valores y las claves proporcionados. (El número de entradas será el de la entrada de lista más corta).
- **Dictionary.Components** generará los componentes del diccionario de entrada. (Es el proceso inverso al nodo creado).
- **Dictionary.RemoveKeys** generará un nuevo objeto de diccionario con las claves de entrada eliminadas.
- **Dictionary.SetValueAtKeys** creará un nuevo diccionario basado en las claves de entrada y los valores para reemplazar el valor actual en las claves correspondientes.
- **Dictionary.ValueAtKey** devolverá el valor en la clave de entrada.
- **Dictionary.Count** le indicará cuántos pares de clave-valor hay en el diccionario.
- **Dictionary.Keys** devolverá las claves almacenadas actualmente en el diccionario.
- **Dictionary.Values** devolverá los valores almacenados actualmente en el diccionario.

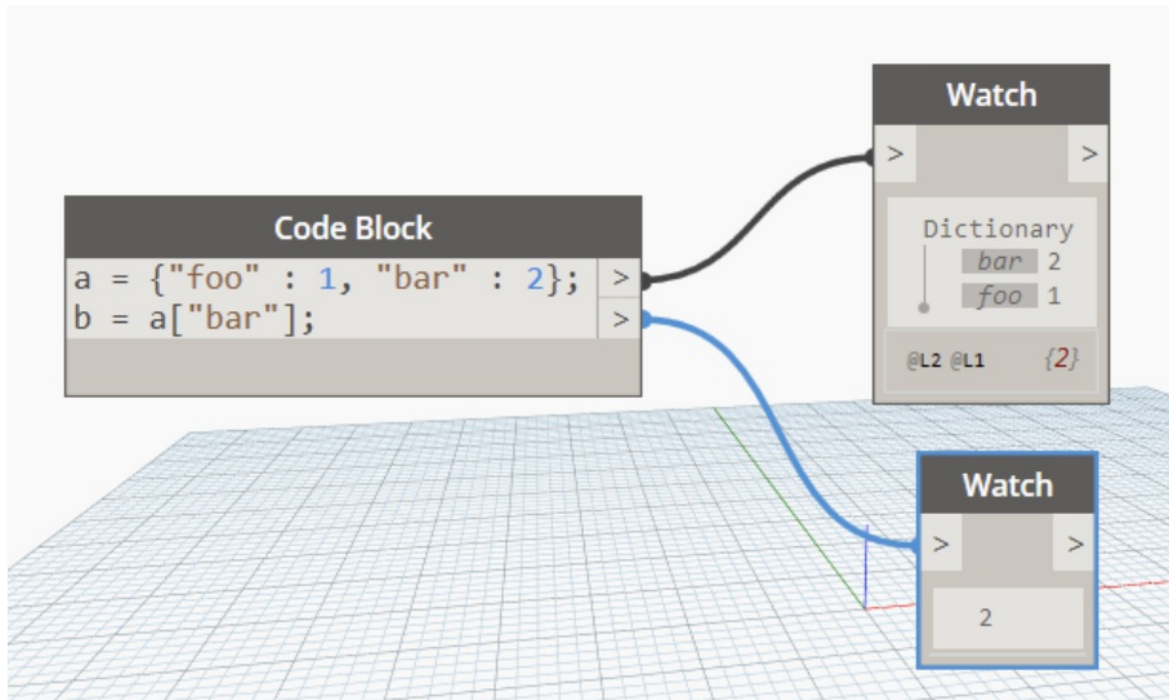
Relacionar de forma general datos con diccionarios es una magnífica alternativa al antiguo método de trabajo con índices y listas.

Usos de los bloques de código

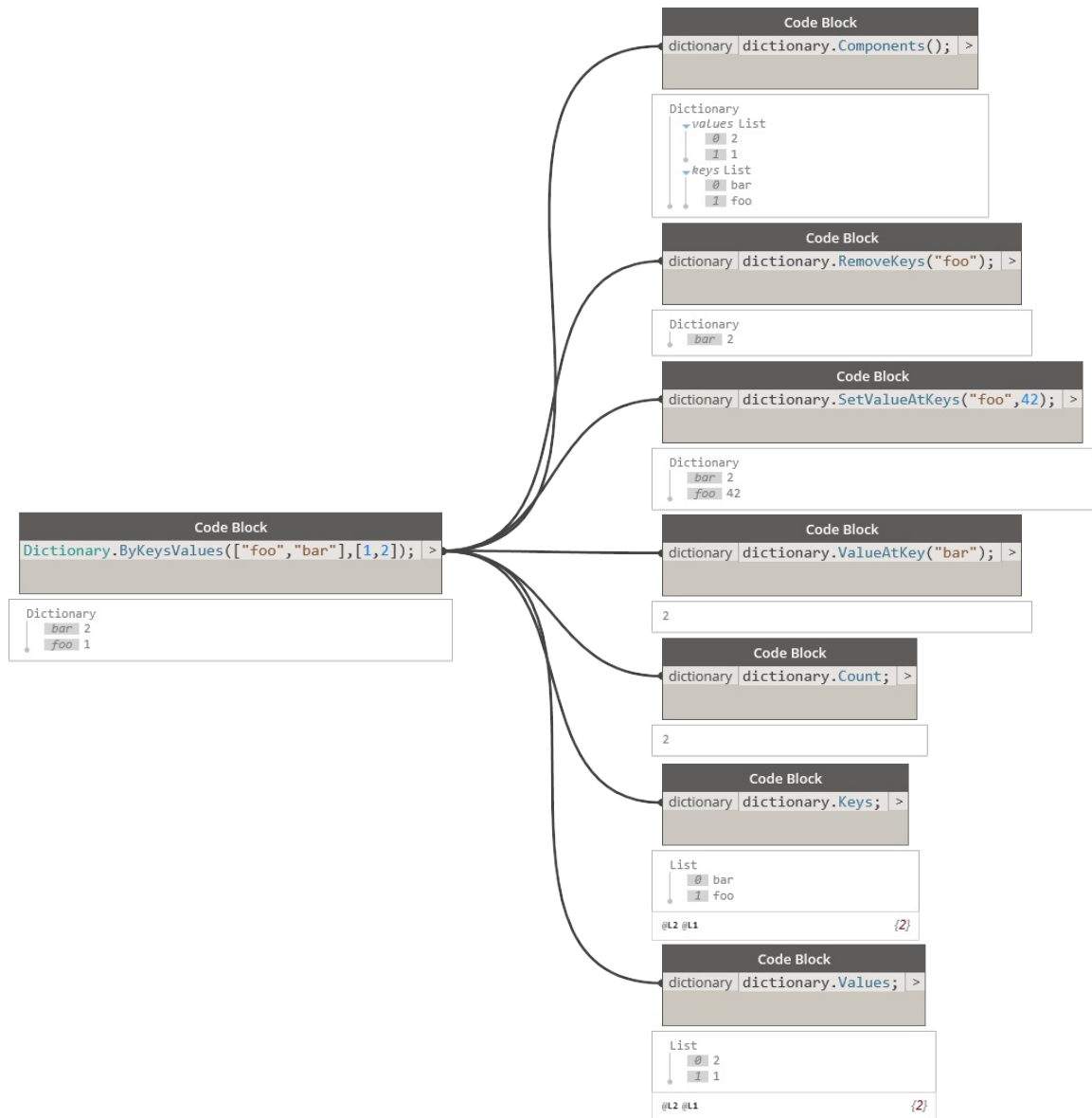
Diccionarios en bloques de código

Dynamo 2.0 no solo presenta los nodos indicados anteriormente para los diccionarios, sino que también hay nuevas funciones en los bloques de código para ello.

Puede utilizar una sintaxis como la siguiente o representaciones basadas en DesignScript de los nodos.



Dado que un diccionario es un tipo de objeto de Dynamo, podemos realizar las siguientes acciones con él.



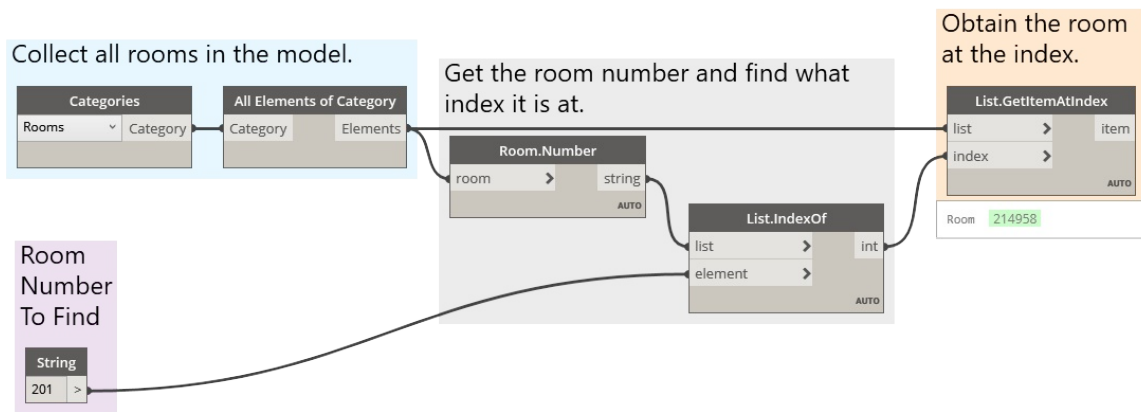
Mantener este tipo de interacciones resulta especialmente útil al relacionar datos de Revit con cadenas. A continuación, veremos algunos casos de uso de Revit.

Casos de uso

Diccionarios: casos de uso de Revit

¿Alguna vez ha deseado buscar algo en Revit mediante un segmento de datos incluido en la solución?

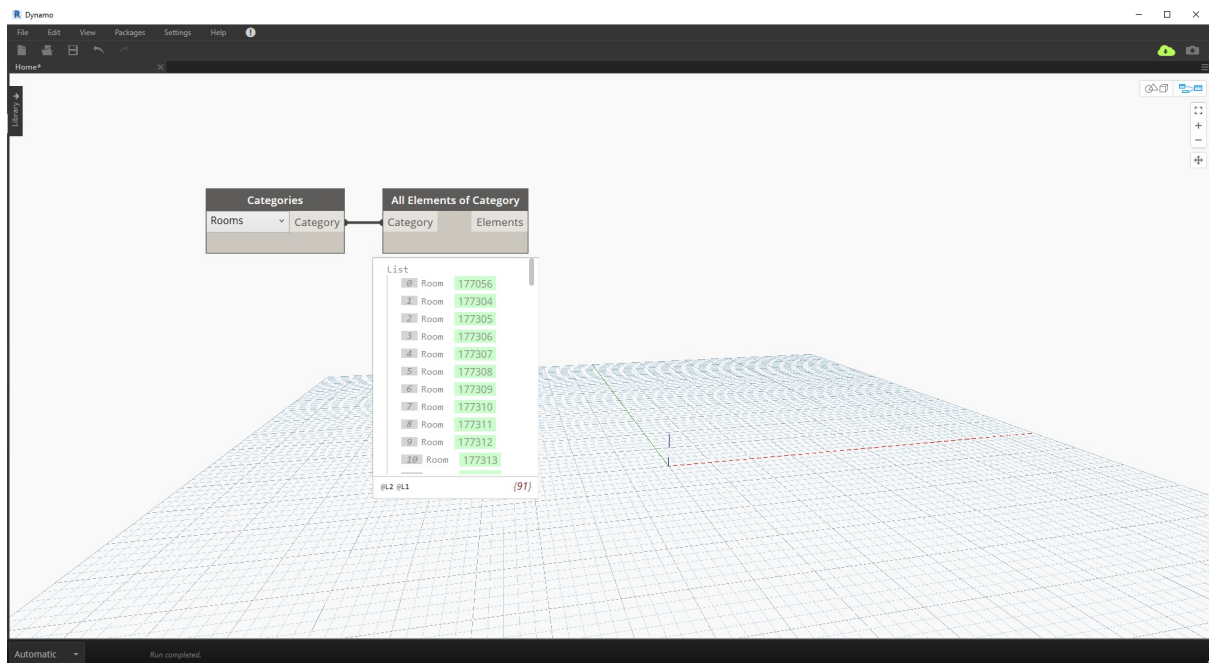
Es probable que, si lo ha hecho, haya realizado algo así:



En la imagen anterior, recopilamos todas las habitaciones del modelo de Revit, obtenemos el índice de la habitación que deseamos (por número de habitación) y, por último, obtenemos la habitación en el índice.

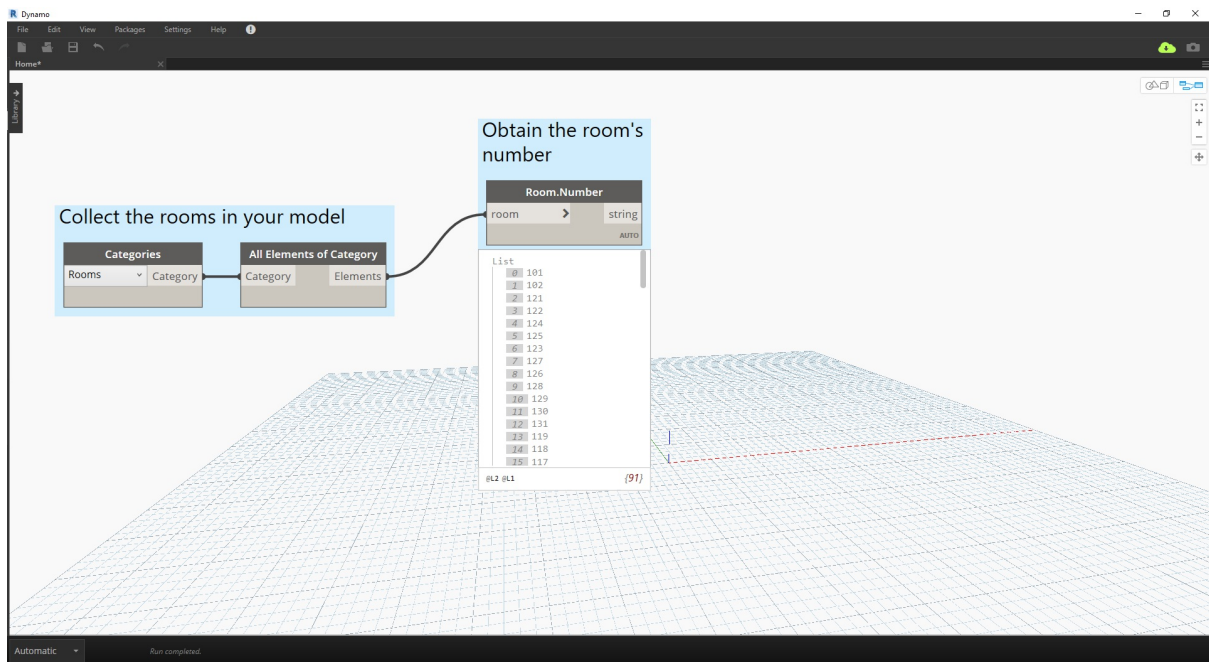
Ahora vamos a recrear esta idea mediante diccionarios.

Descargue el archivo de ejemplo que acompaña a este ejercicio (haga clic con el botón derecho y seleccione "Guardar enlace como..."); [RoomDictionary.dyn](#). En el Apéndice, se incluye una lista completa de los archivos de ejemplo.



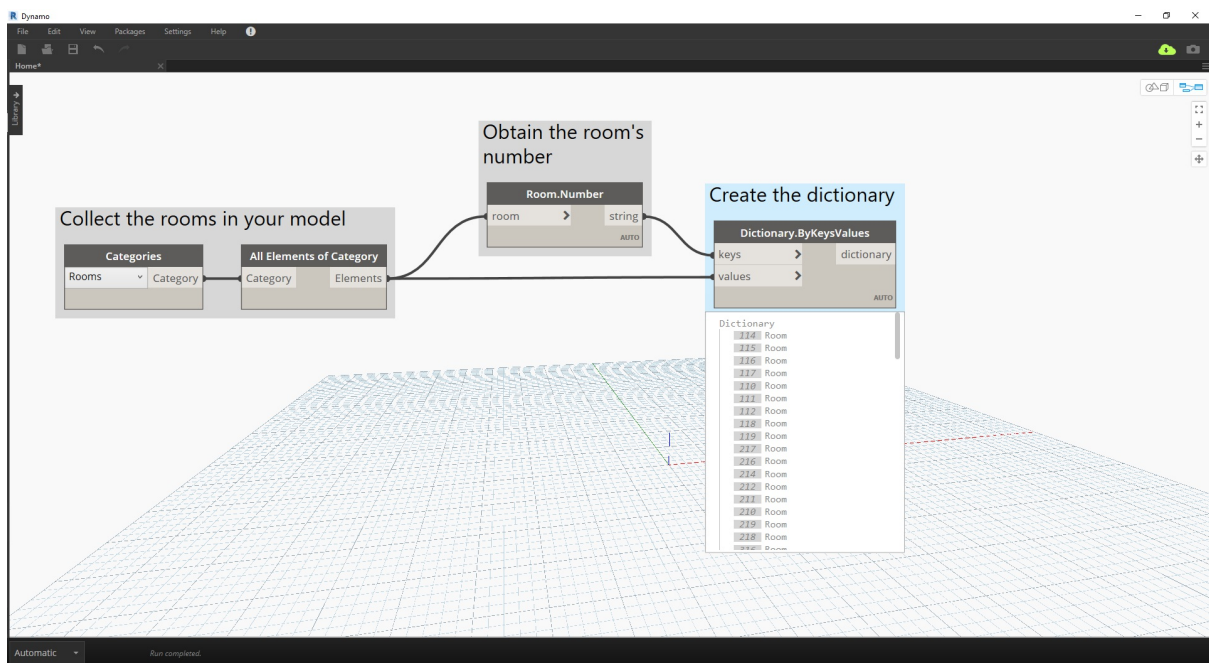
Debemos recopilar primero todas las habitaciones de nuestro modelo de Revit.

- Seleccionamos la categoría de Revit con la que deseamos trabajar (en este caso, trabajaremos con habitaciones).
- Le indicamos a Dynamo que recopile todos esos elementos.



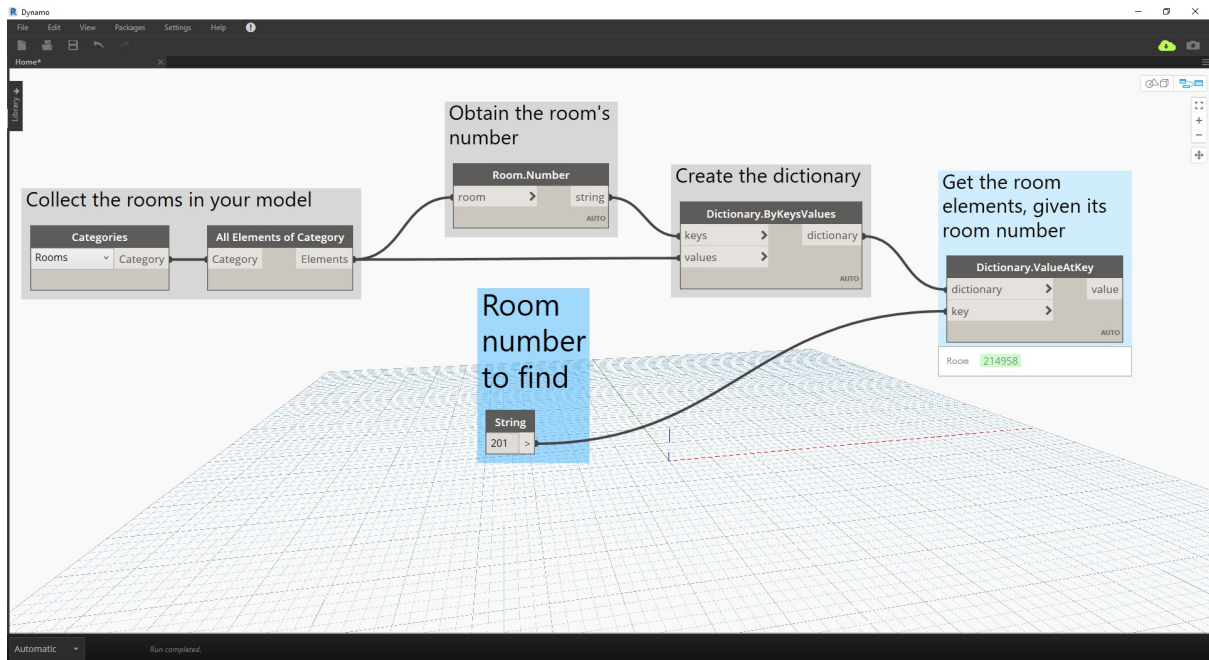
A continuación, debemos decidir las claves que vamos a utilizar para buscar estos datos. (La información sobre las claves se encuentra en la sección [9-1 ¿Qué es un diccionario?](#)).

- Los datos que usaremos son el número de habitación.



Ahora crearemos el diccionario con las claves y los elementos especificados.

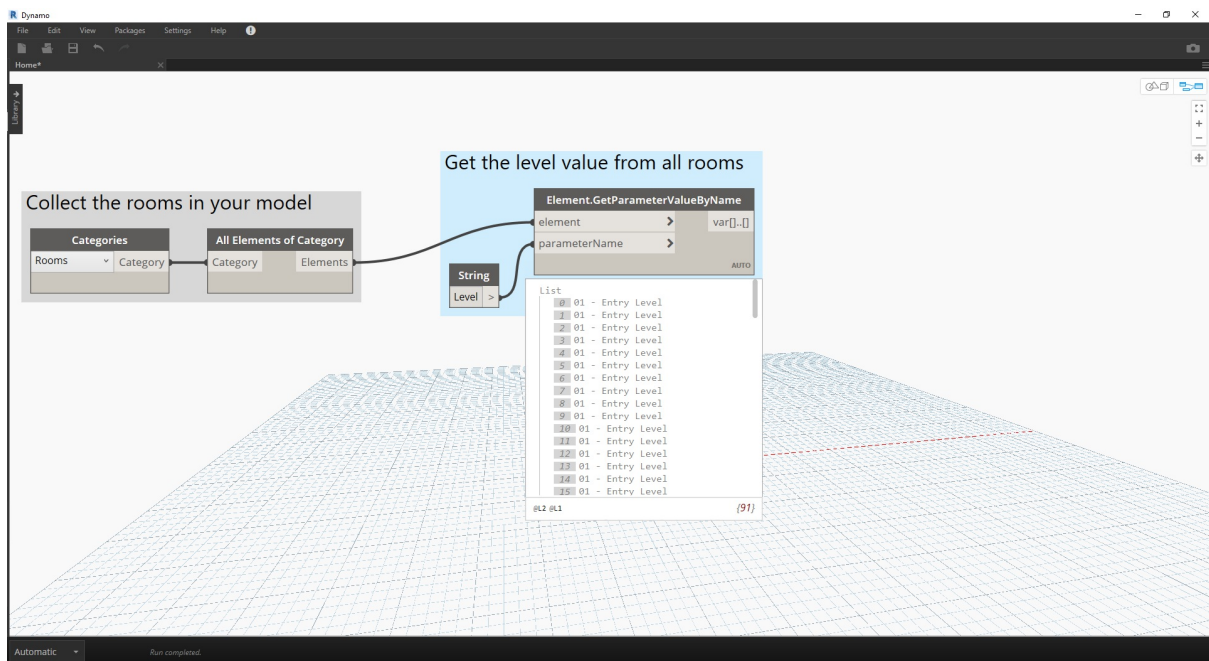
- El nodo `Dictionary.ByKeysValues` creará un diccionario con las entradas correspondientes especificadas.
- Las claves deben ser una cadena, mientras que los valores pueden ser diferentes tipos de objeto.



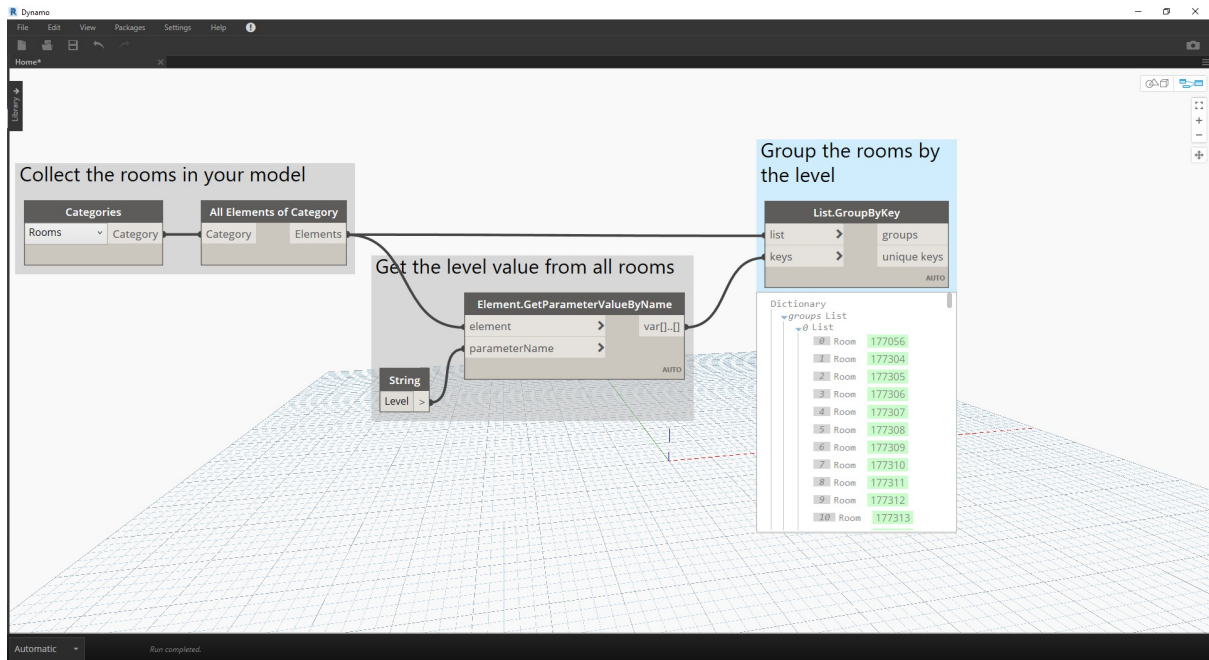
Por último, ahora podemos recuperar una habitación del diccionario con su número de habitación.

- String será la clave que utilizaremos para buscar un objeto en el diccionario.
- Dictionary.ValueAtKey obtendrá ahora el objeto del diccionario.

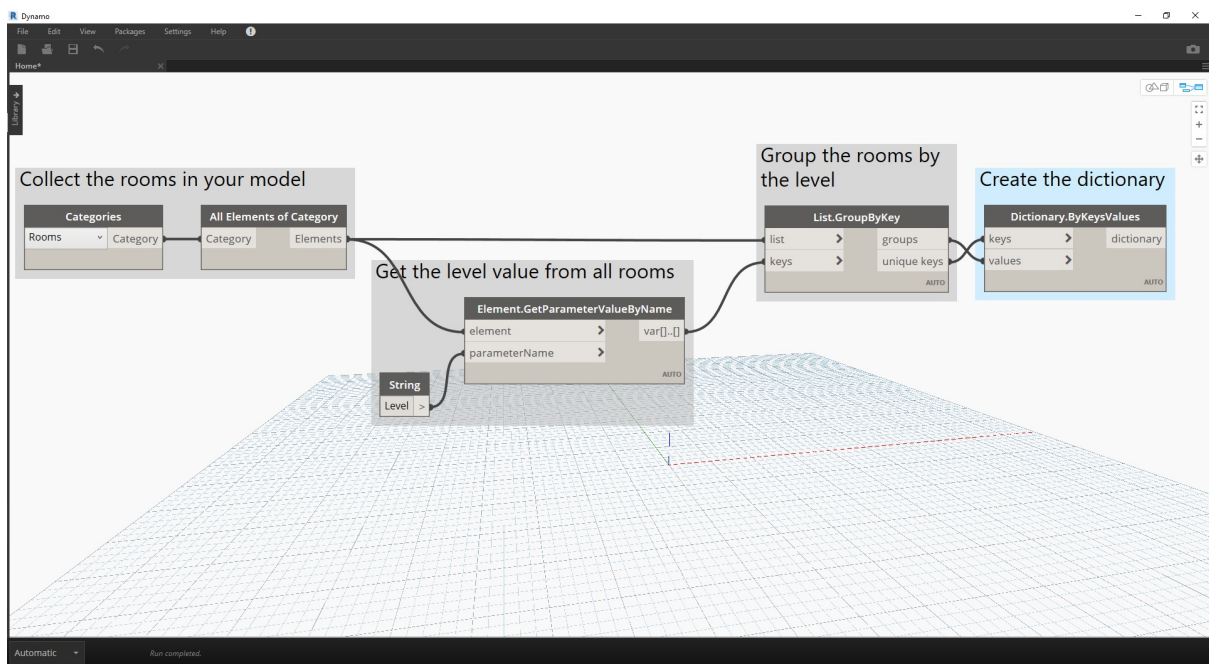
Con esta misma lógica de diccionario, también podemos crear diccionarios con objetos agrupados. Si deseamos buscar todas las habitaciones en un nivel determinado, podemos modificar el gráfico anterior de la siguiente manera.



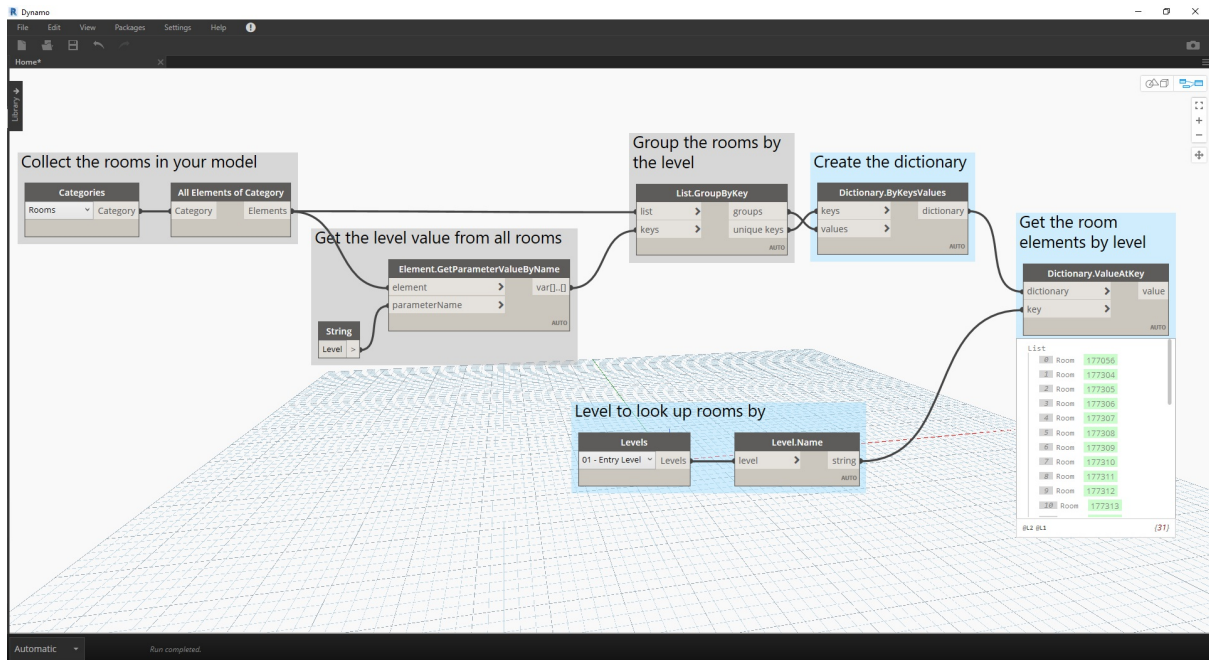
- En lugar de utilizar el número de habitación como clave, ahora podemos utilizar un valor de parámetro (en este caso, utilizaremos nivel).



- Ahora podemos agrupar las habitaciones por el nivel en el que residen.



- Con los elementos agrupados por el nivel, ahora podemos utilizar las claves compartidas (claves exclusivas) como nuestra clave para el diccionario y las listas de habitaciones como elementos.



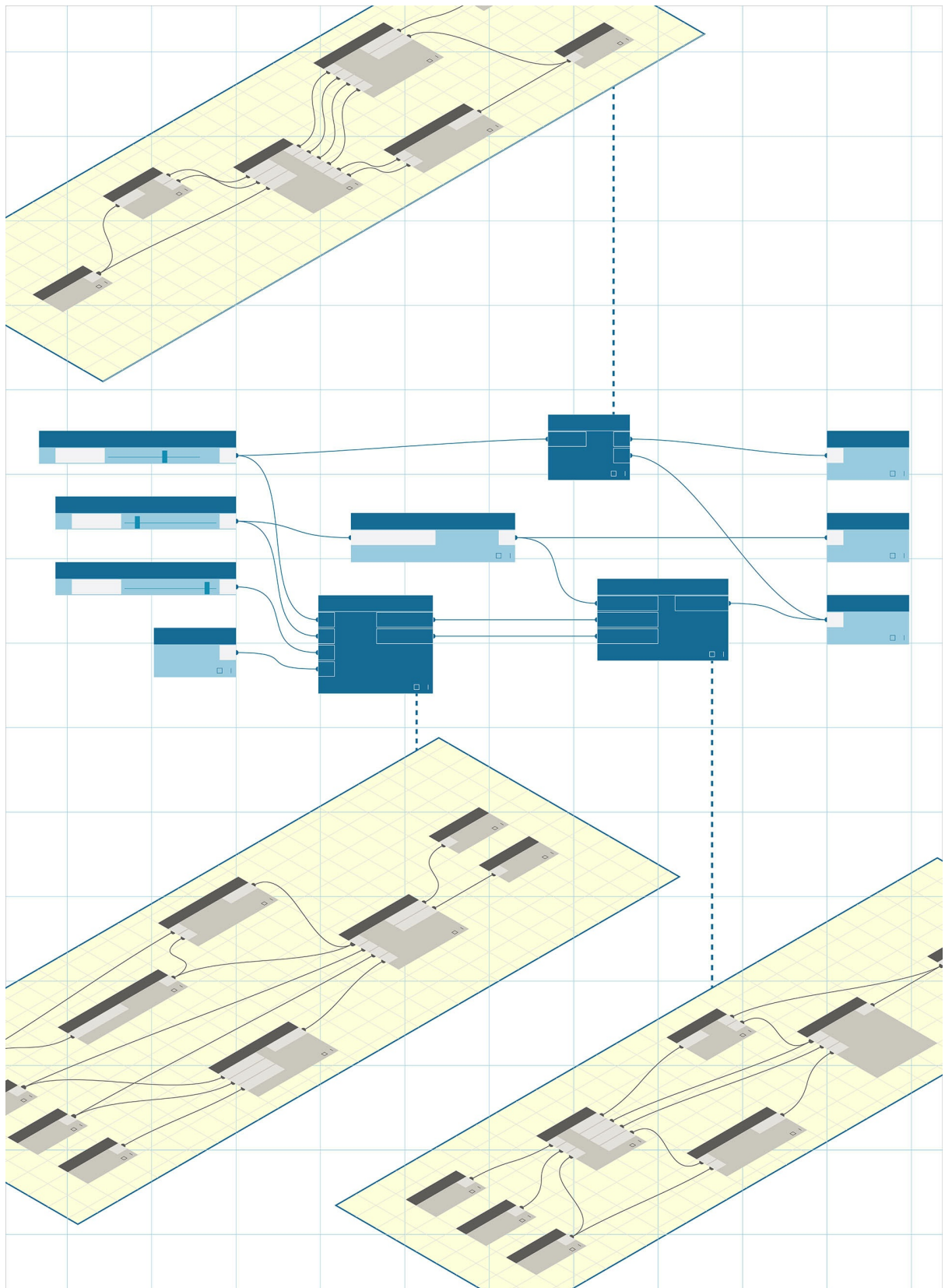
- Por último, mediante los niveles del modelo de Revit, podemos buscar en el diccionario las habitaciones que se encuentran en ese nivel. `Dictionary.ValueAtKey` utilizará el nombre de nivel y devolverá los objetos de habitación presentes en ese nivel.

Las oportunidades de uso del diccionario son realmente infinitas. La capacidad de relacionar los datos de BIM de Revit con el propio elemento plantea diversos casos de uso.

Nodos personalizados

Nodos personalizados

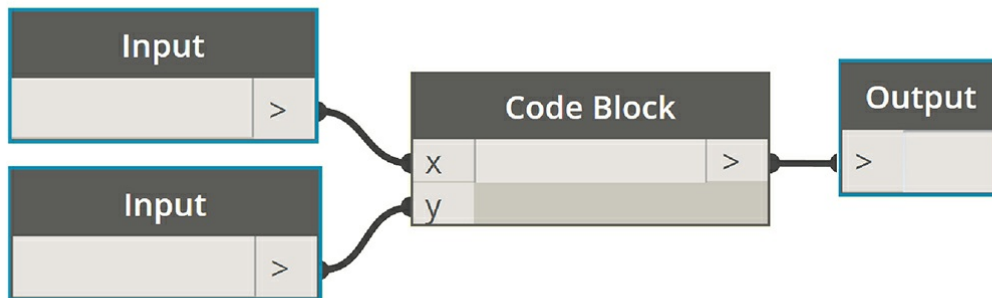
De forma predefinida, Dynamo tiene muchas funciones almacenadas en su biblioteca de nodos. Para las rutinas de uso frecuente o ese gráfico especial que desea compartir con la comunidad, los nodos personalizados son un método excelente para ampliar Dynamo aún más.



Introducción a los nodos personalizados

Nodos personalizados

Dynamo ofrece muchos nodos principales para una amplia gama de tareas de programación visual. En ocasiones, una solución más rápida, elegante o fácil de compartir consiste en crear sus propios nodos. Estos se pueden reutilizar en distintos proyectos, lo que hace que el gráfico sea más claro y limpio, y se pueden transferir a Package Manager y compartir con la comunidad global de Dynamo.



Limpieza del gráfico

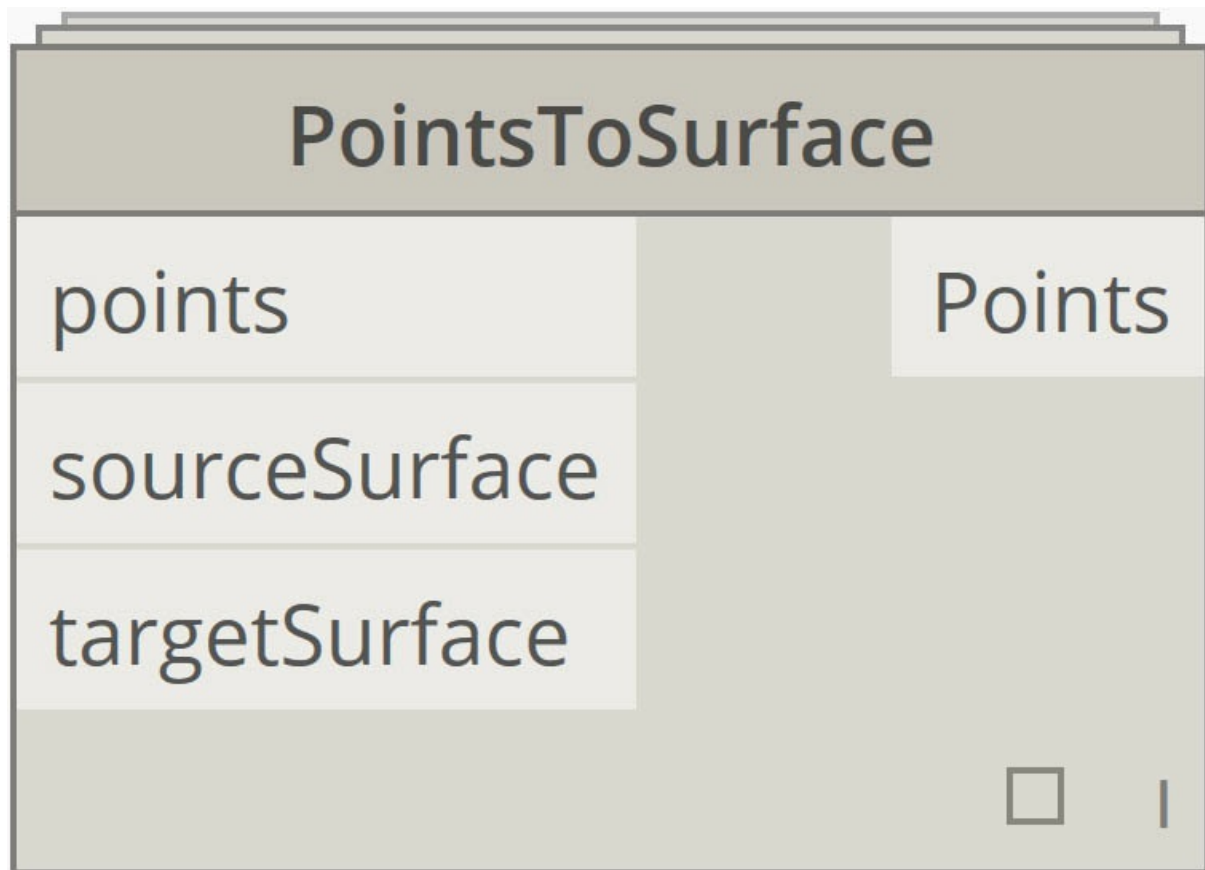
Los nodos personalizados se crean mediante la anidación de otros nodos y nodos personalizados dentro de un "nodo personalizado de Dynamo", que podemos considerar conceptualmente como un contenedor. Cuando se ejecuta este nodo contenedor en el gráfico, se ejecutará todo lo que contiene para permitirle reutilizar y compartir una combinación útil de nodos.

Adaptación a los cambios

Cuando existen varias copias de un nodo personalizado en el gráfico, puede actualizarlas todas mediante la edición del nodo personalizado base. Esto permite actualizar el gráfico sin problemas adaptándolo a los cambios que puedan producirse en el flujo de trabajo o el diseño.

Uso compartido

Podría decirse que la mejor función de los nodos personalizados es su capacidad para compartir el trabajo. Si un "usuario avanzado" crea un gráfico complejo de Dynamo y se lo entrega a un diseñador que no esté familiarizado con esta herramienta, este puede condensar el gráfico a sus aspectos básicos para interactuar con el diseño. El nodo personalizado se puede abrir para editar el gráfico interno, pero manteniendo la simplicidad del "contenedor". Con este proceso, los nodos personalizados permiten a los usuarios de Dynamo diseñar un gráfico limpio e intuitivo.

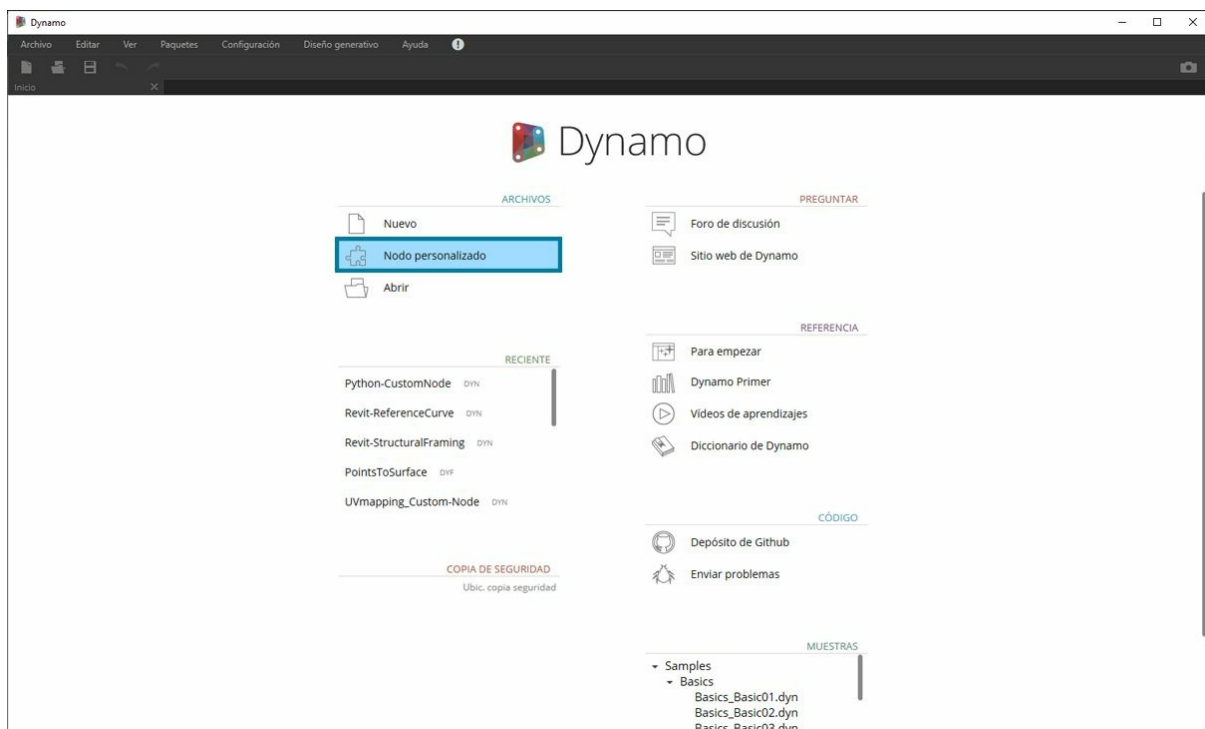


Muchos métodos para crear un nodo

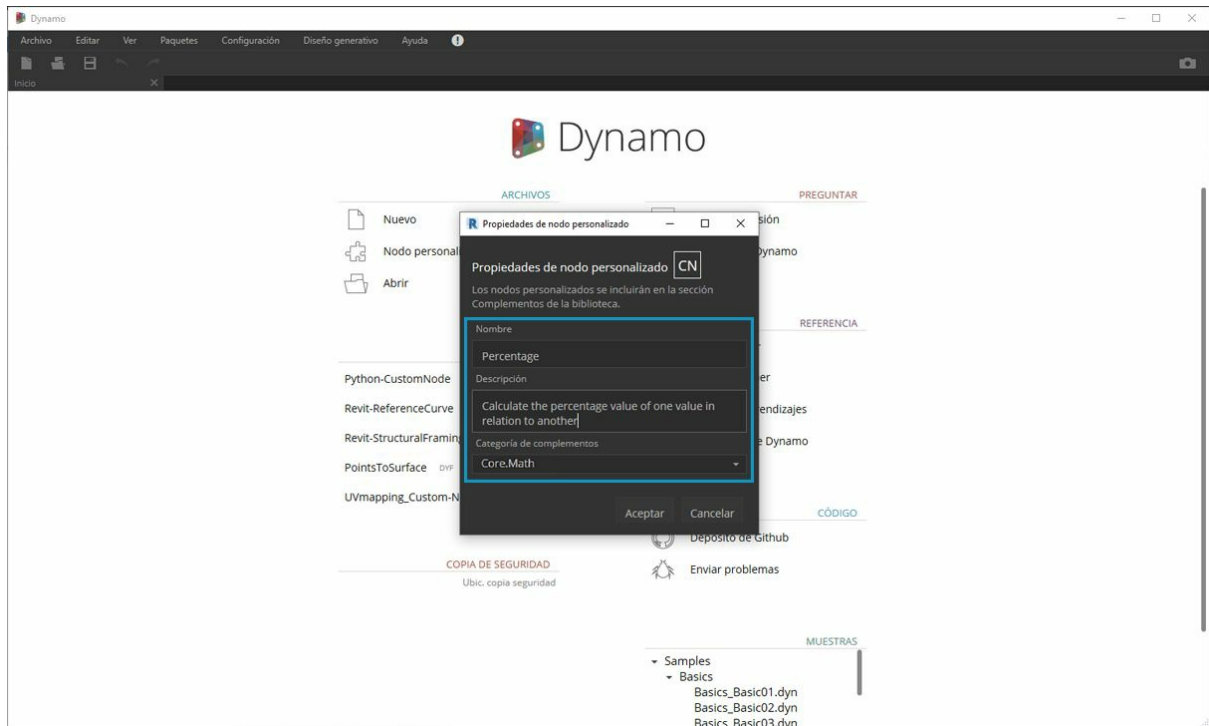
Hay una gran variedad de métodos para crear nodos personalizados en Dynamo. En los ejemplos de este capítulo, crearemos nodos personalizados directamente desde la interfaz de usuario de Dynamo. Si es un programador y está interesado en el formato C# o Zero-Touch, puede consultar [esta página](#) en la página wiki de Dynamo para obtener una revisión más detallada.

Entorno de nodos personalizados

Pasemos al entorno de nodos personalizados y creemos un nodo sencillo para calcular un porcentaje. El entorno de nodos personalizados es diferente al entorno de gráficos de Dynamo, pero la interacción es básicamente la misma. Una vez señalado esto, crearemos nuestro primer nodo personalizado.

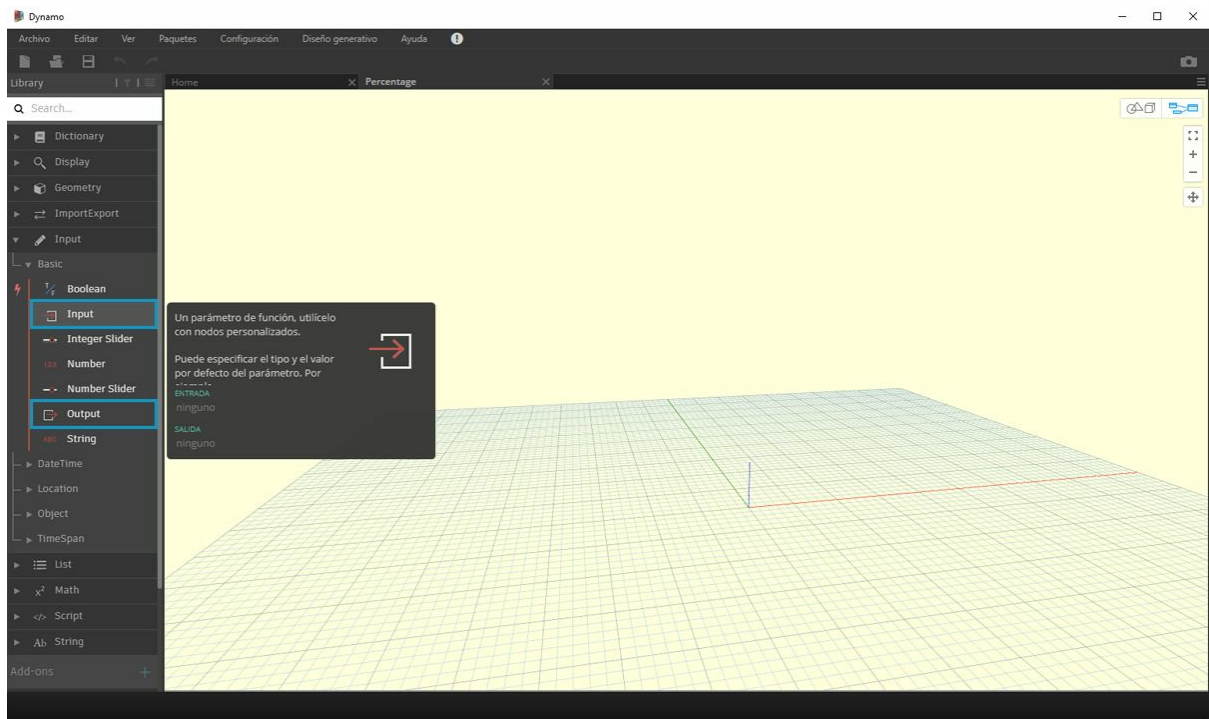


Para crear un nodo personalizado desde cero, inicie Dynamo y seleccione **Nodo personalizado** o pulse **Ctrl + Mayús + N** en el lienzo.

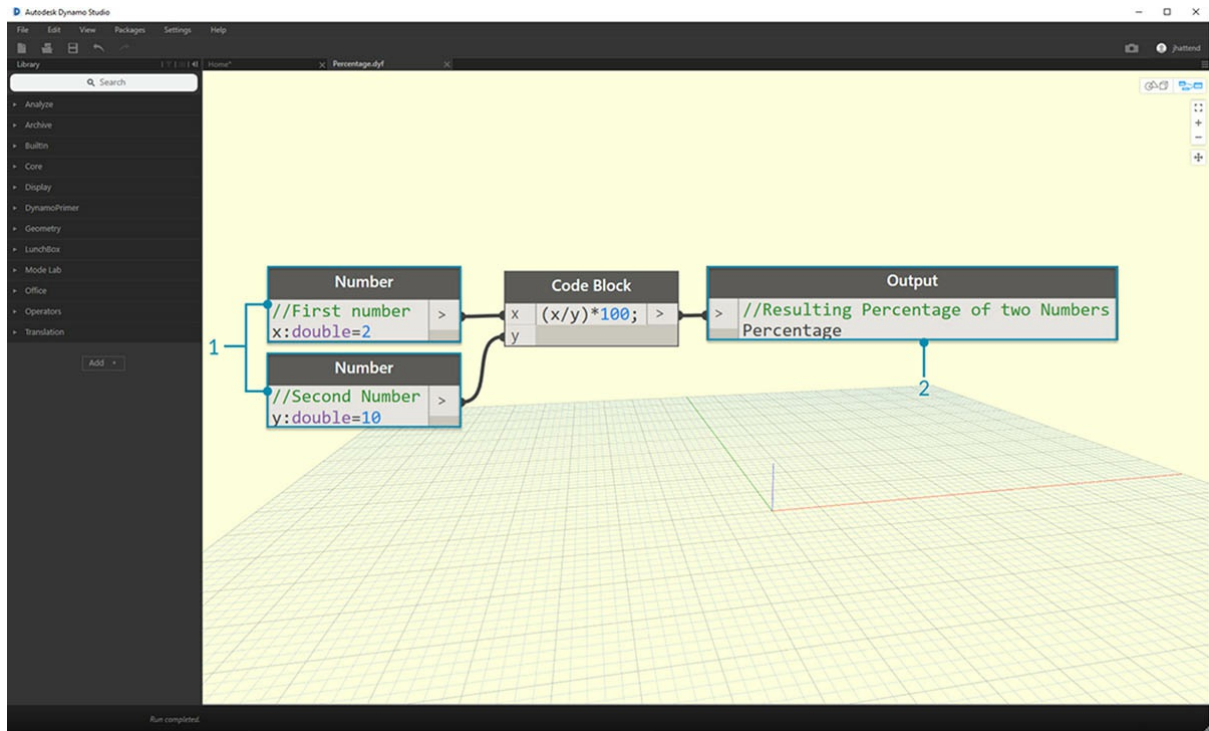


Asigne un nombre, una descripción y una categoría en el cuadro de diálogo **Propiedades de nodo personalizado**.

1. **Nombre:** porcentaje.
2. **Descripción:** calcule el porcentaje de un valor en relación con otro.
3. **Categoría:** Core.Math.

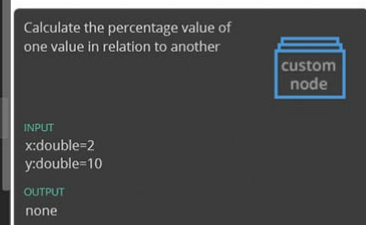
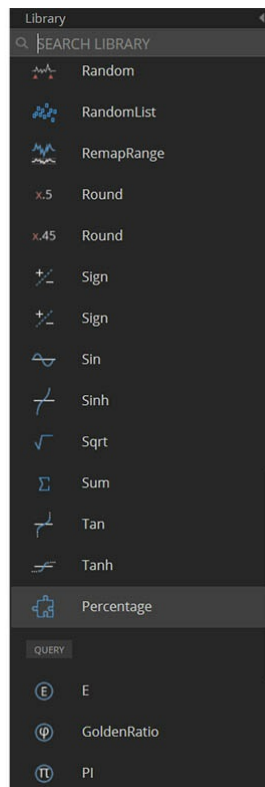
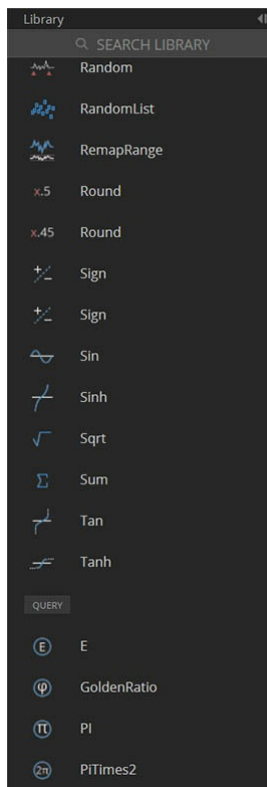


Se abre un lienzo con un fondo de color amarillo, lo que indica que está trabajando en un nodo personalizado. En este lienzo, tiene acceso a todos los nodos principales de Dynamo, así como a los nodos **Input** y **Output**, que etiquetan los datos que entran en el nodo personalizado y salen de él. Se encuentran en **Core > Entrada**.



1. **Entradas:** los nodos de entrada crean puertos de entrada en el nodo personalizado. La sintaxis de un nodo de entrada es *nombre_entrada: tipo_de_datos = valor_por_defecto (opcional)*.
1. **Salidas:** son similares a las entradas; estas crearán puertos en el nodo personalizado y les asignarán un nombre. Considere la posibilidad de añadir un **comentario personalizado** a los puertos de entrada y salida para que se muestren como sugerencia en los tipos de entrada y salida. Esto se explica con más detalle en la sección [Creación de nodos personalizados](#).

Puede guardar este nodo personalizado como un archivo .dyf (en lugar de como un archivo .dyn estándar); este se añadirá automáticamente a la sesión actual y futuras. Encontrará el nodo personalizado en la biblioteca, en la categoría especificada en las propiedades del nodo personalizado.



Izquierda: la categoría Core > Math de la biblioteca por defecto Derecha: Core > Math con el nuevo nodo personalizado

Avancemos

Ahora que hemos creado nuestro primer nodo personalizado, las secciones siguientes profundizarán en las funciones de los nodos personalizados y en la publicación de flujos de trabajo genéricos. En la siguiente sección, estudiaremos el desarrollo de un nodo personalizado que transfiere geometría de una superficie a otra.

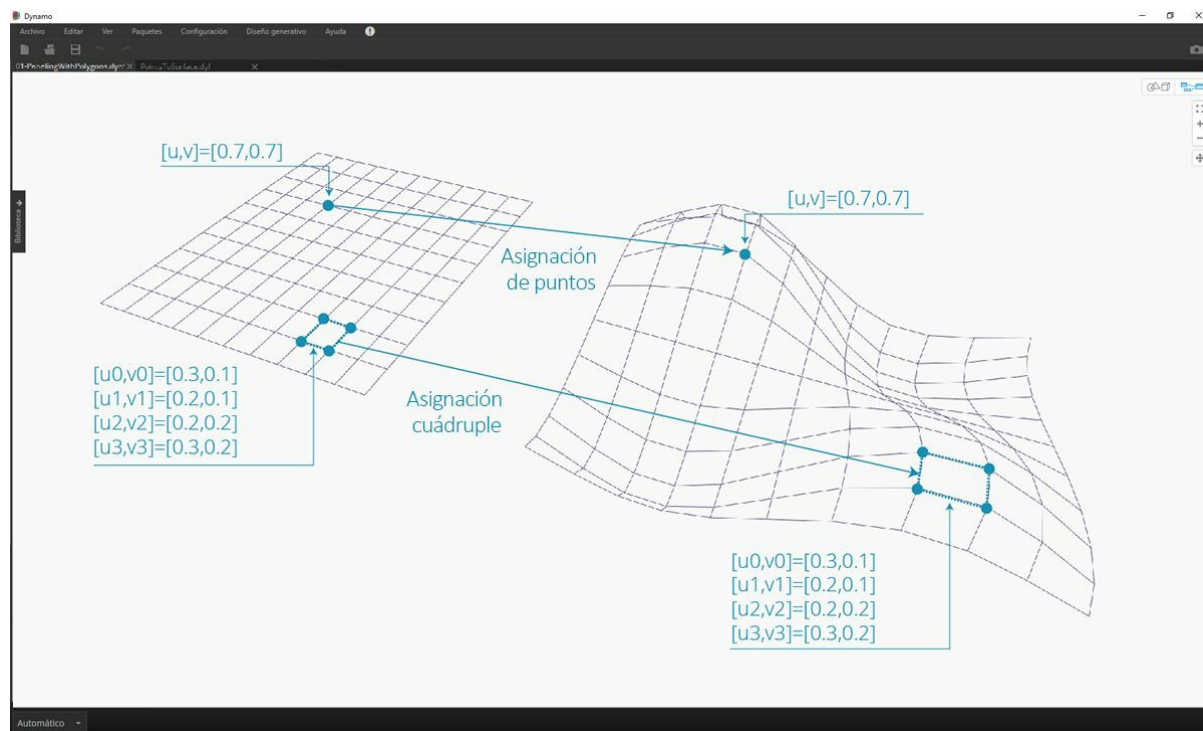
Creación de un nodo personalizado

Creación de un nodo personalizado

Dynamo ofrece varios métodos diferentes para crear nodos personalizados. Puede crear nodos personalizados desde cero, a partir de un gráfico existente o de forma explícita en C#. En esta sección, vamos a cubrir la generación de un nodo personalizado en la interfaz de usuario de Dynamo a partir de un gráfico existente. Este método es ideal para limpiar el espacio de trabajo, así como para empaquetar una secuencia de nodos con el fin de reutilizarlos en otros entornos.

Nodos personalizados para la asignación de UV

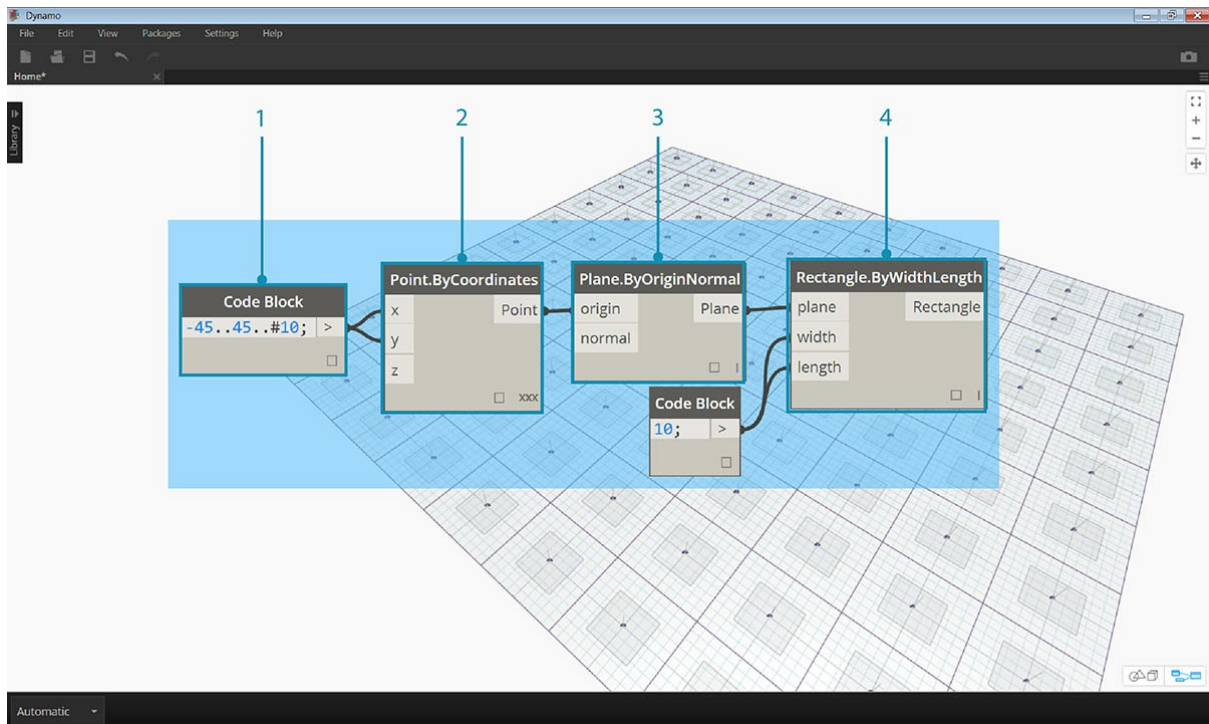
En la imagen siguiente, se asigna un punto de una superficie a otra mediante coordenadas UV. Utilizaremos este concepto para crear una superficie panelizada que haga referencia a curvas en el plano XY. Crearemos paneles de cuadrados para esta panelización pero, con esta misma lógica, podemos crear una amplia variedad de paneles con asignación de UV. Esta es una gran oportunidad para el desarrollo de nodos personalizados, ya que vamos a poder repetir un proceso similar más fácilmente en este gráfico o en otros flujos de trabajo de Dynamo.



Creación de un nodo personalizado a partir de un gráfico existente

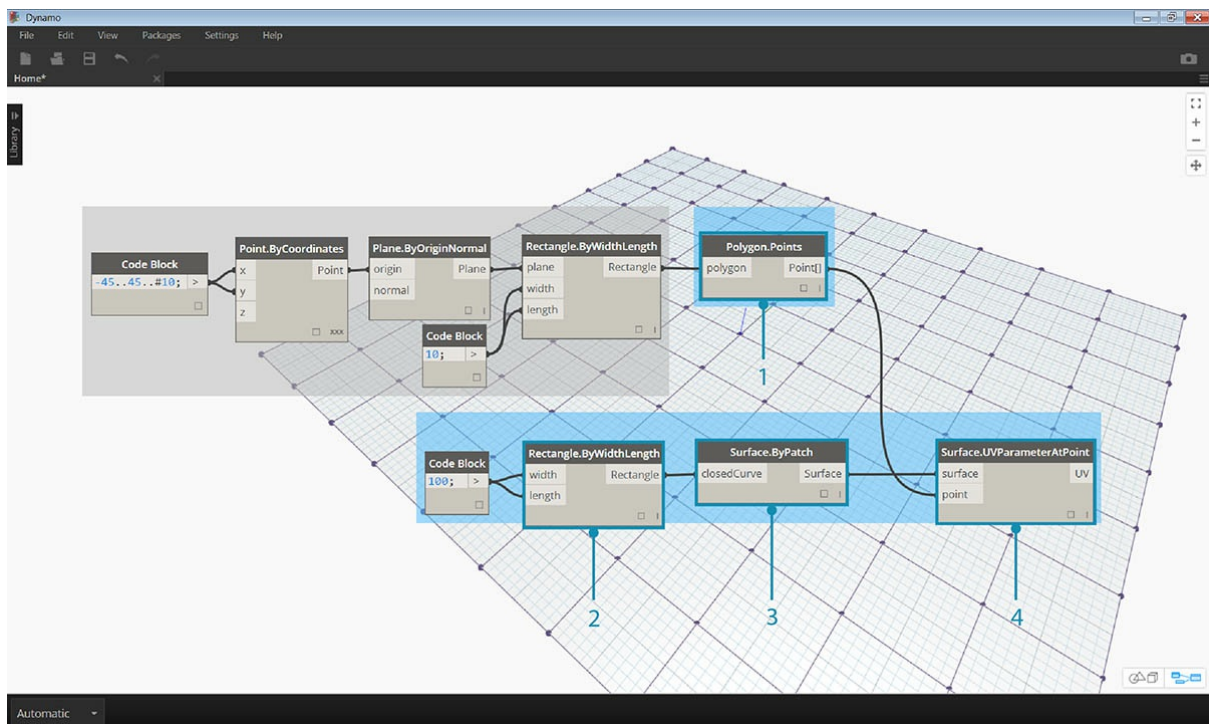
Descargue y descomprima los archivos de ejemplo para este ejercicio (haga clic con el botón derecho y elija "Guardar enlace como..."). En el Apéndice se incluye una lista completa de los archivos de ejemplo. [UV-CustomNode.zip](#)

Comenzaremos con la creación de un gráfico que deseamos anidar en un nodo personalizado. En este ejemplo, crearemos un gráfico que asigna polígonos de una superficie base a una superficie de destino mediante coordenadas UV. Este proceso de asignación de UV es algo que se utiliza con frecuencia, lo que lo convierte en un buen candidato para un nodo personalizado. Para obtener más información sobre las superficies y el espacio UV, consulte la sección 5.5. El gráfico completo es *UVmapping_Custom-Node.dyn*, del archivo .zip que hemos descargado anteriormente.



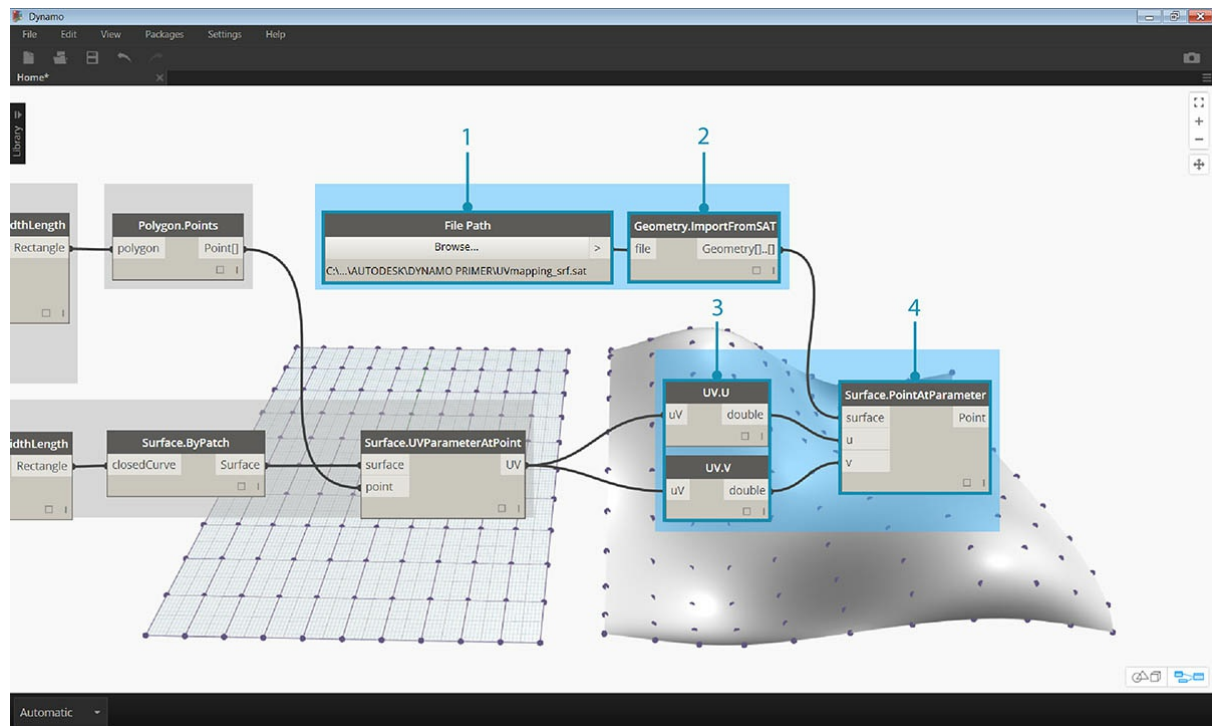
1. **Code Block:** cree un rango de 10 números entre 45 y -45 mediante un bloque de código.
2. **Point.ByCoordinates:** conecte la salida del bloque de código a las entradas "x" e "y" y establezca el encaje como producto vectorial. Ahora debería ver una rejilla de puntos.
3. **Plane.ByOriginNormal:** conecte la salida "Point" a la entrada "origin" para crear un plano en cada uno de los puntos. Se utilizará el vector normal por defecto (0,0,1).
4. **Rectangle.ByWidthLength:** conecte los planos del paso anterior a la entrada "plane" y utilice un bloque de código con un valor de 10 para especificar la anchura y la longitud.

Ahora debería ver una rejilla de rectángulos. Asignemos ahora estos rectángulos a una superficie de destino mediante coordenadas UV.



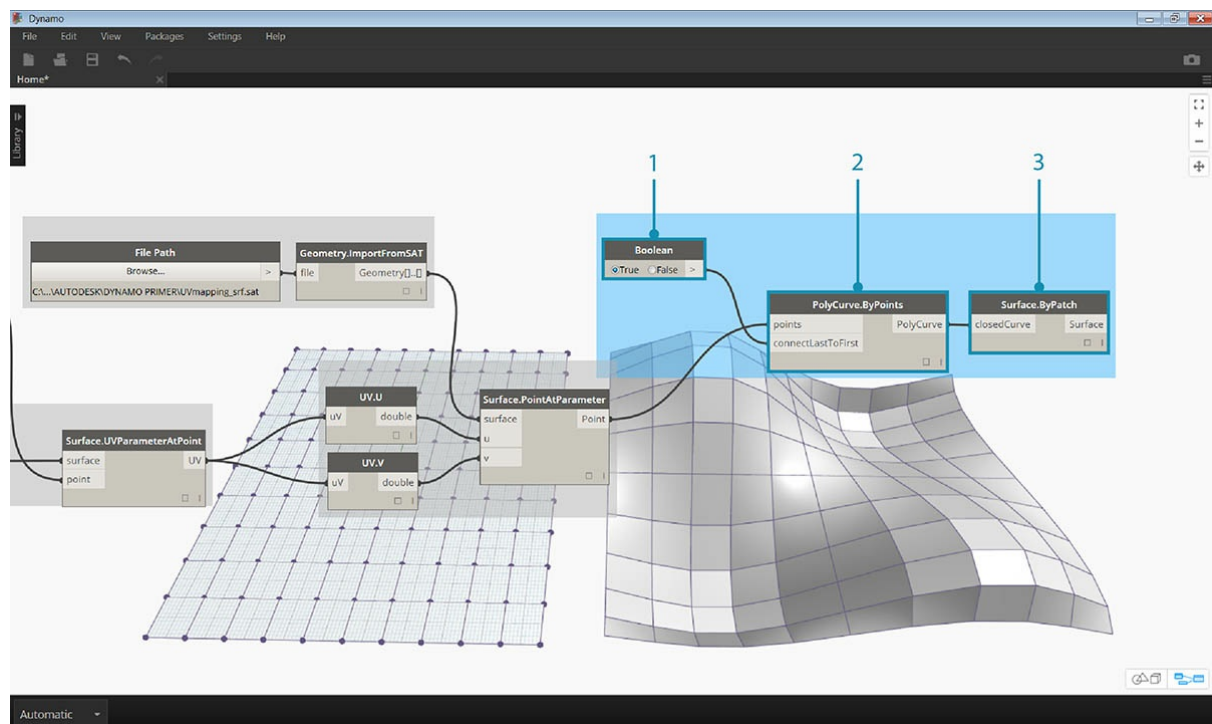
1. **Polygon.Points:** conecte la salida Rectangle del paso anterior a la entrada "polygon" para extraer los puntos de esquina de cada rectángulo. Estos son los puntos que asignaremos a la superficie de destino.
2. **Rectangle.ByWidthLength:** utilice un bloque de código con un valor de 100 para especificar la anchura y la longitud de un rectángulo. Este será el contorno de la superficie base.
3. **Surface.ByPatch:** conecte la salida Rectangle del paso anterior a la entrada "closedCurve" para crear una superficie base.
4. **Surface.UVParameterAtPoint:** conecte la salida "Point" del nodo Polygon.Points y la salida "Surface" del nodo Surface.ByPatch para devolver el parámetro UV de cada punto.

Ahora que tenemos una superficie base y un conjunto de coordenadas UV, podemos importar una superficie de destino y asignar los puntos entre las superficies.



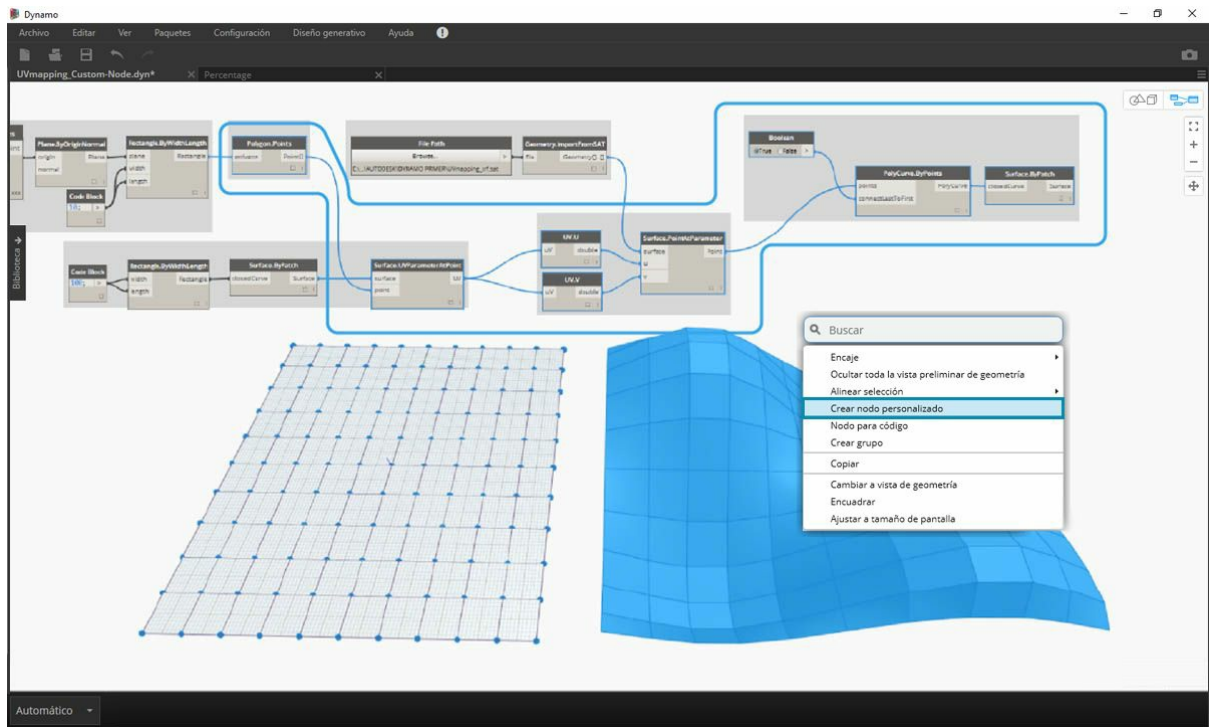
1. **File Path**: seleccione la ruta de archivo de la superficie que desea importar. El tipo de archivo debe ser .SAT. Haga clic en el botón "Examinar..." y desplácese hasta el archivo *UVmapping_srf.sat* del archivo .zip que hemos descargado anteriormente.
2. **Geometry.ImportFromSAT**: conecte la ruta de archivo para importar la superficie. La superficie importada se muestra en la vista preliminar de la geometría.
3. **UV**: conecte la salida del parámetro UV a un nodo **UV.U** y un nodo **UV.V**.
4. **Surface.PointAtParameter**: conecte la superficie importada, así como las coordenadas u y v. Ahora debería ver una rejilla de puntos 3D en la superficie de destino.

El último paso consiste en utilizar los puntos 3D para crear parches de superficie rectangulares.

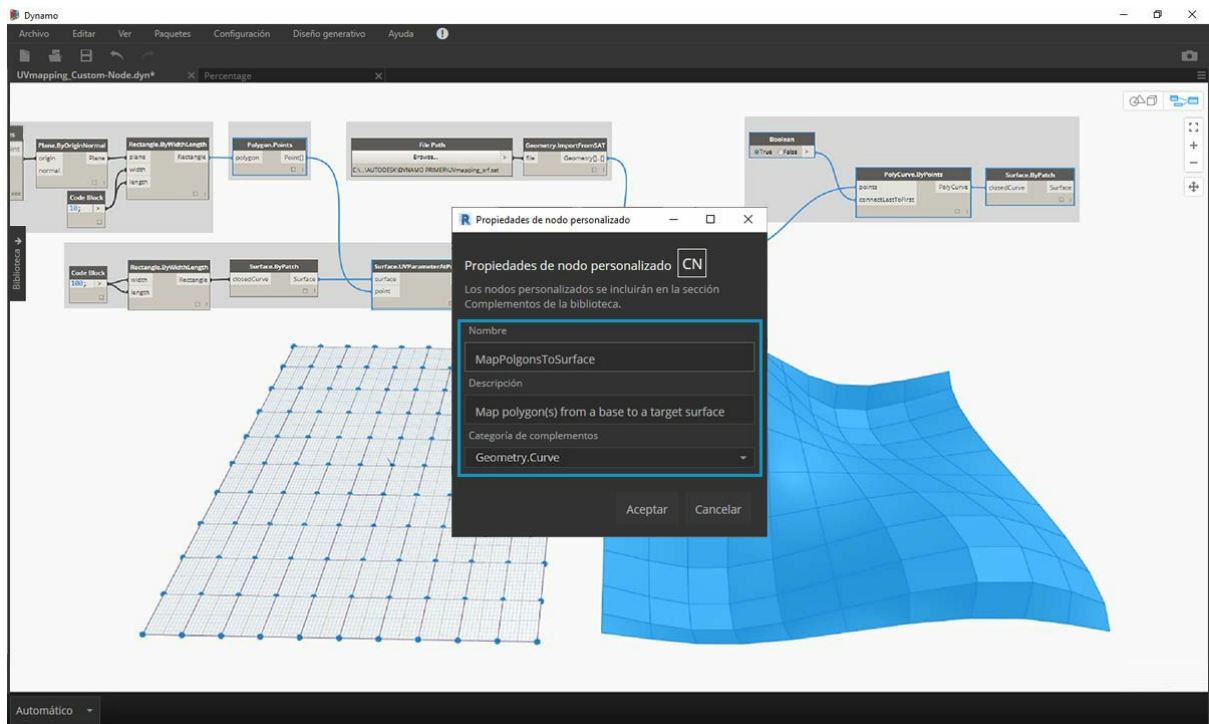


1. **PolyCurve.ByPoints**: conecte los puntos de la superficie para construir una PolyCurve a través de los puntos.
2. **Boolean**: añada un nodo Boolean al espacio de trabajo, conéctelo a la entrada "connectLastToFirst" y, a continuación, cambie el valor a Verdadero (True) para cerrar las PolyCurves. Ahora debería ver los rectángulos asignados a la superficie.
3. **Surface.ByPatch**: conecte las PolyCurves a la entrada "closedCurve" para construir parches de superficie.

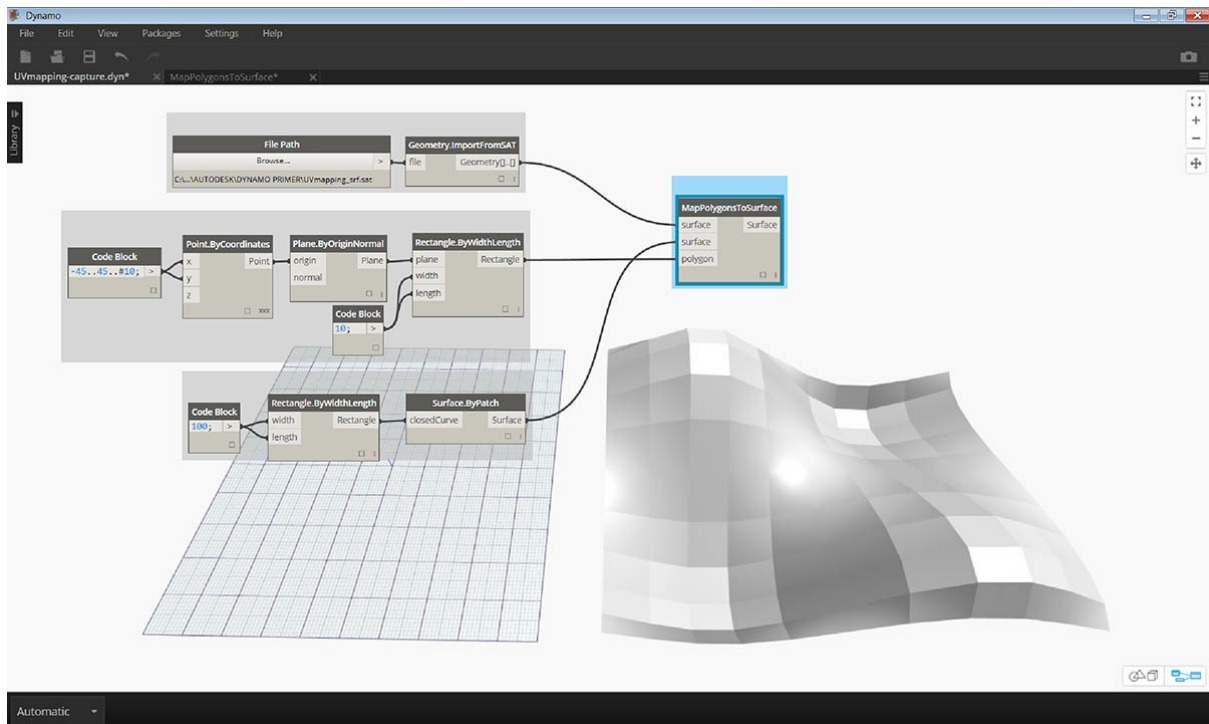
Ahora vamos a seleccionar los nodos que queremos anidar en un nodo personalizado teniendo en cuenta las entradas y salidas que este debe tener. Queremos que nuestro nodo personalizado sea lo más flexible posible, de modo que debería poder asignar cualquier polígono, no solo rectángulos.



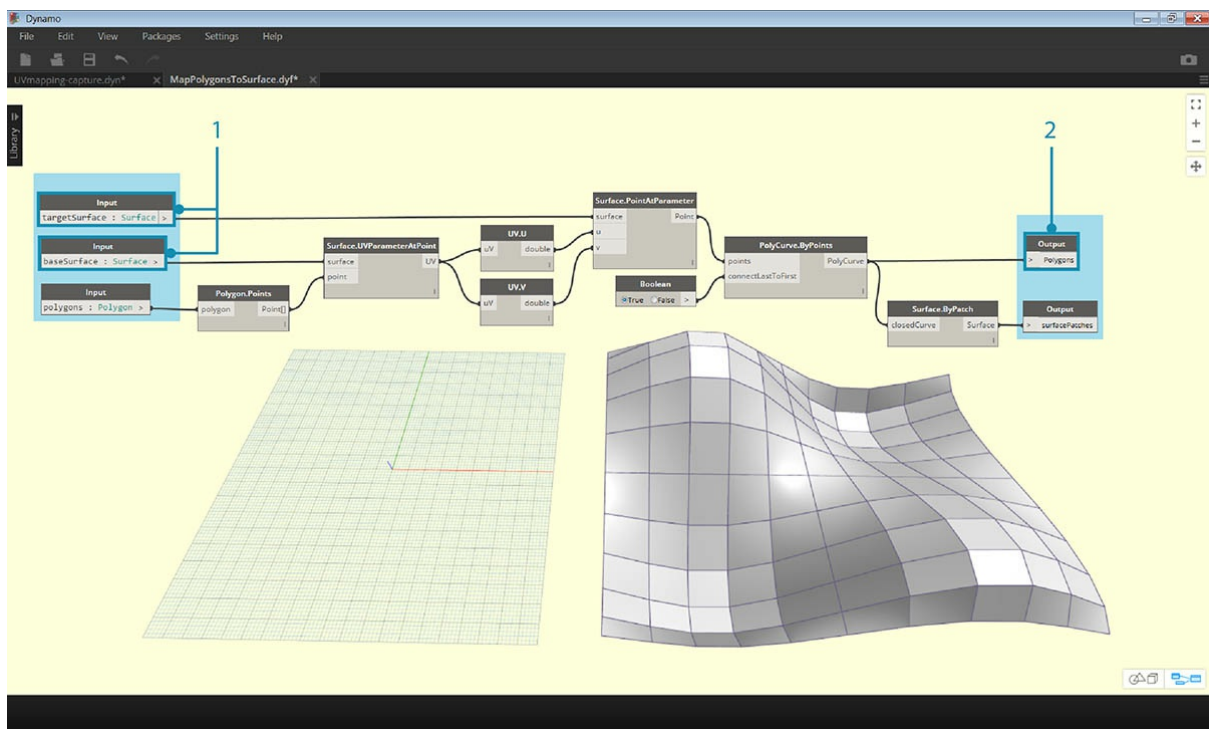
Seleccione los nodos anteriores (comenzando por *Polygon.Points*), haga clic con el botón derecho en el espacio de trabajo y seleccione "nodo a partir de selección".



En el cuadro de diálogo *Propiedades de nodo personalizado*, asigne un nombre, una descripción y una categoría al nodo personalizado.

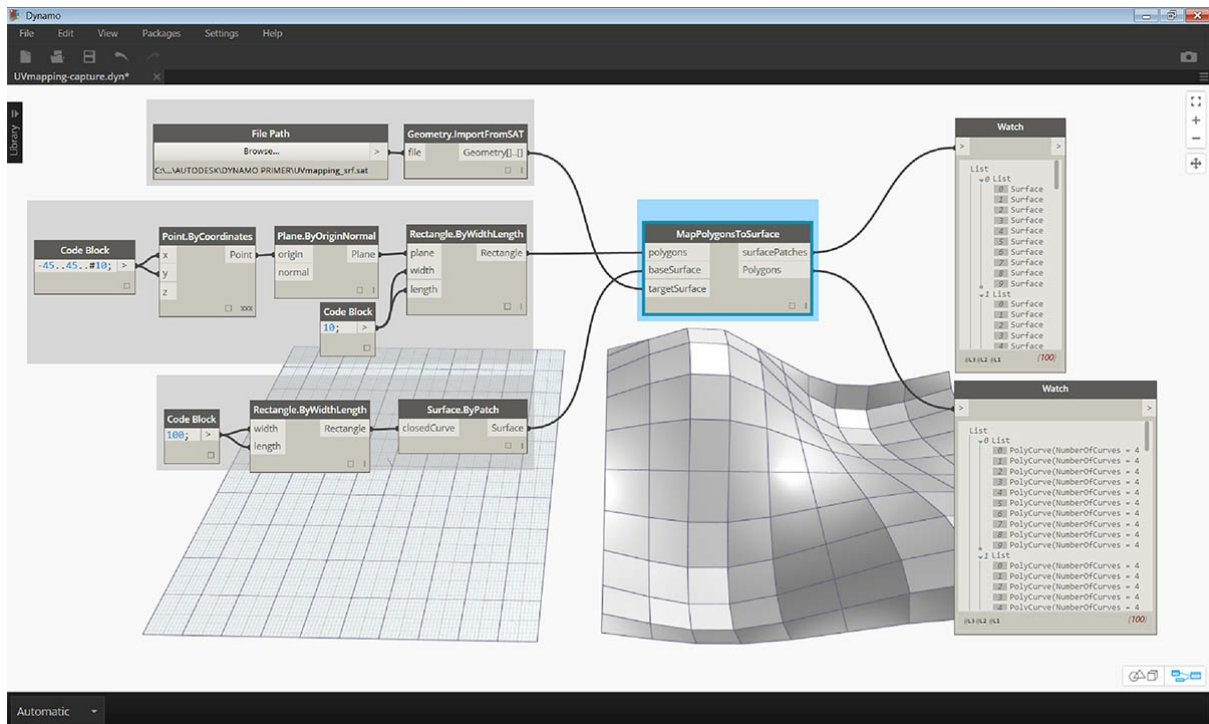


El nodo personalizado ha limpiado considerablemente el espacio de trabajo. Observe que las entradas y salidas se han nombrado en función de los nodos originales. Editaremos el nodo personalizado para que los nombres sean más descriptivos.



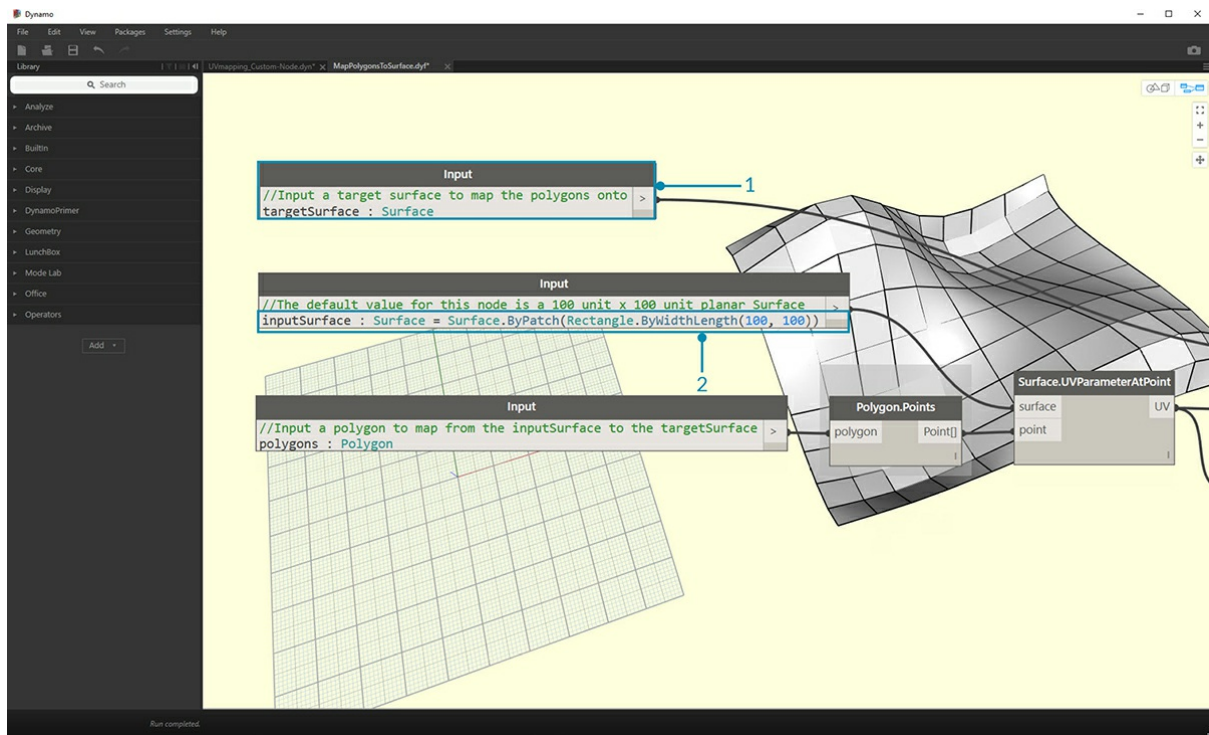
Haga doble clic en el nodo personalizado para editarlo. Se abrirá un espacio de trabajo con un fondo amarillo que representa el interior del nodo.

1. **Entradas:** cambie los nombres de entrada a *baseSurface* y *targetSurface*.
2. **Salidas:** añada una salida adicional para los polígonos asignados. Guarde el nodo personalizado y vuelva al espacio de trabajo de inicio.



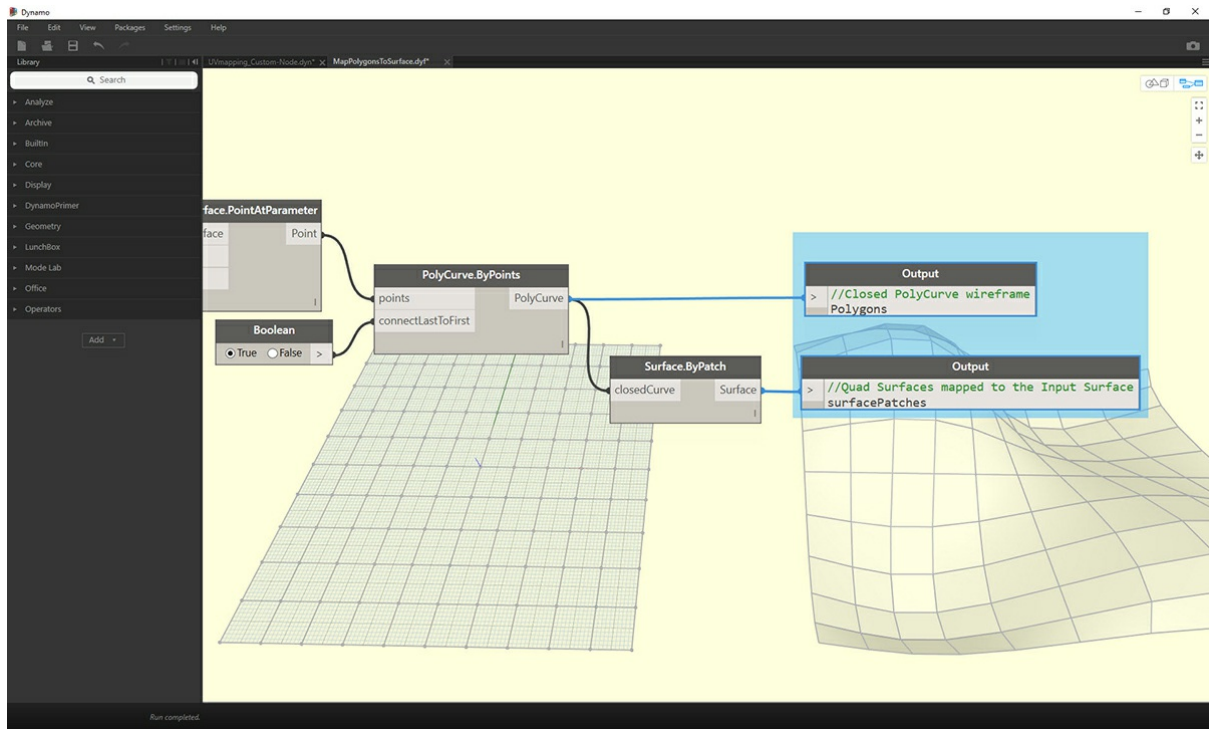
El nodo **MapPolygonsToSurface** refleja los cambios que acabamos de realizar.

También podemos reforzar la solidez del nodo personalizado añadiendo **comentarios personalizados**. Los comentarios pueden servir como ayuda en las entradas y salidas o explicar el funcionamiento del nodo. Los comentarios aparecerán cuando el usuario coloque el cursor sobre una entrada o una salida de un nodo personalizado.

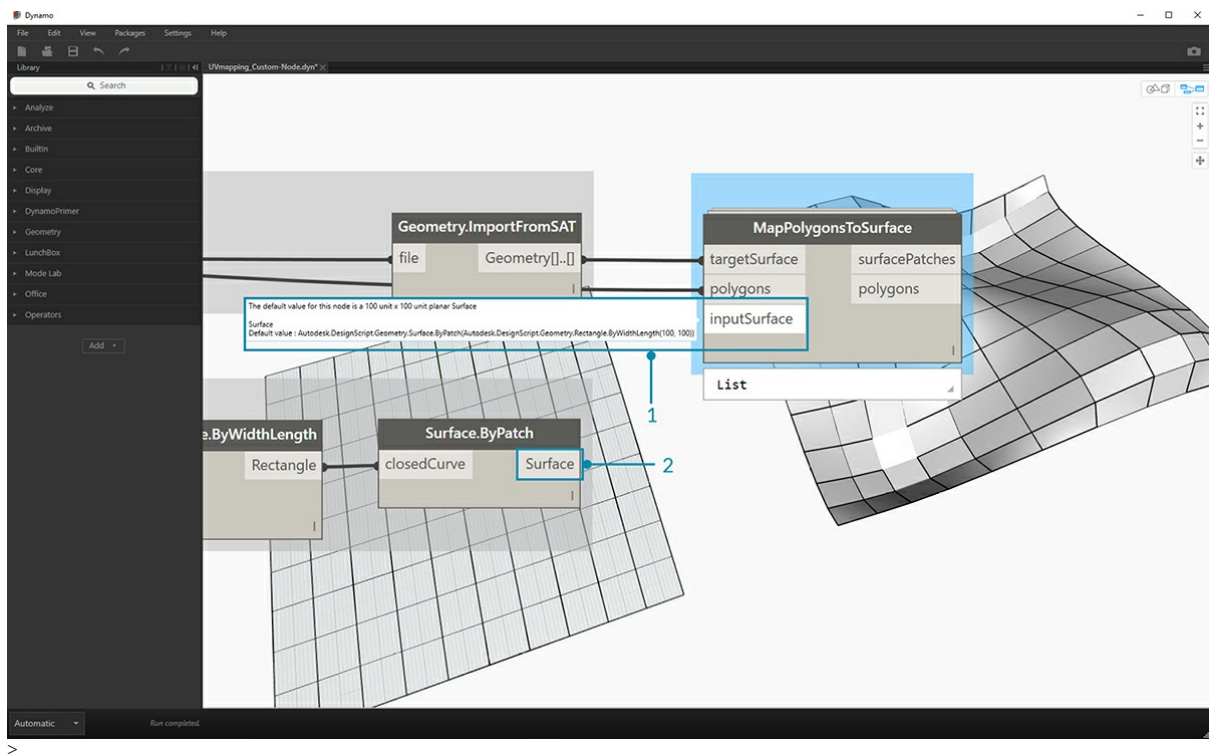


Haga doble clic en el nodo personalizado para editarlo. De este modo, se volverá a abrir el espacio de trabajo con fondo amarillo.

1. Comience por editar el bloque de código Input. Para iniciar un comentario, escriba "/" seguido del texto del comentario. Escriba cualquier información que pueda ayudar a entender mejor el nodo; aquí describiremos *targetSurface*.
2. Vamos a definir también el valor por defecto de *inputSurface* estableciendo el valor a que equivale un tipo de entrada. Aquí, vamos a definir el valor por defecto en el conjunto *Surface.ByPatch* original.



También se pueden aplicar comentarios a las salidas. Comience por editar el texto en el bloque de código Output. Para iniciar un comentario, escriba "//" seguido del texto del comentario. Aquí vamos a aclarar las salidas *Polygons* y *surfacePatches* añadiendo una descripción más detallada.



1. Coloque el cursor sobre las entradas del nodo personalizado para ver los comentarios.
2. Con el valor por defecto establecido en *inputSurface*, también podemos ejecutar la definición sin una entrada de superficie.

Publicación en la biblioteca

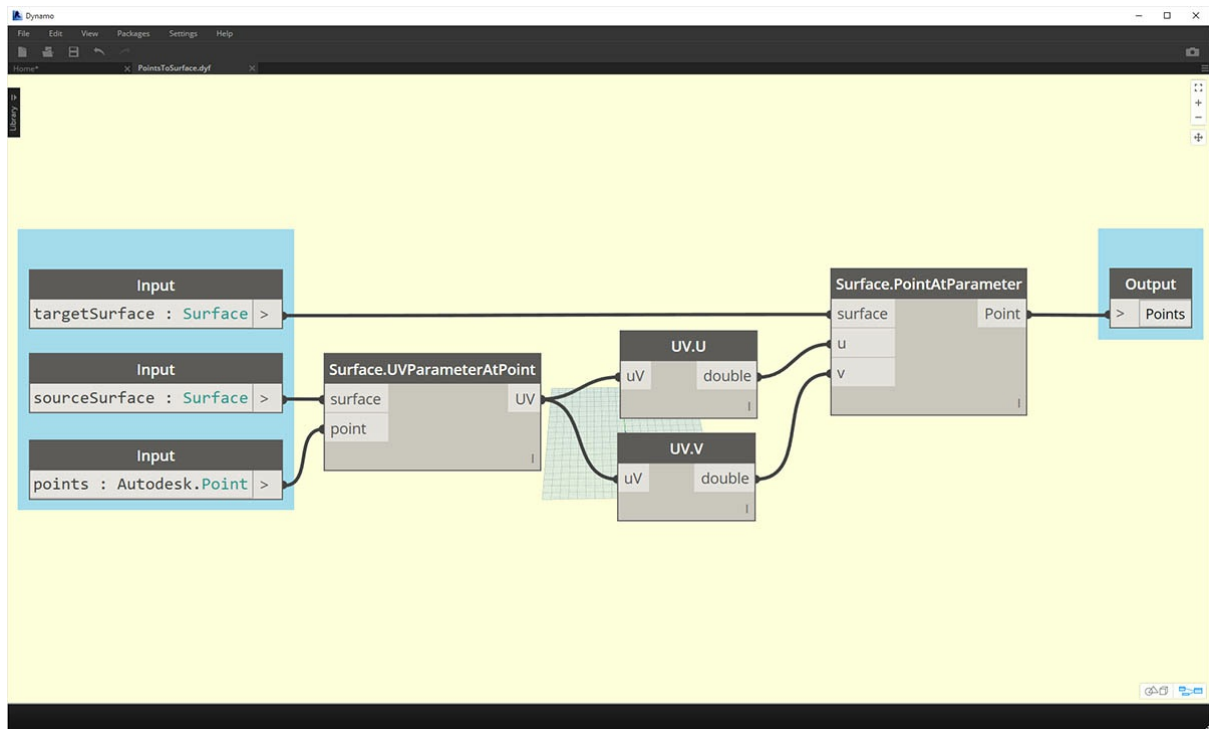
Adición a la biblioteca

Acabamos de crear un nodo personalizado y lo hemos aplicado a un proceso específico en el gráfico de Dynamo. Y nos gusta tanto este nodo que deseamos conservarlo en nuestra biblioteca de Dynamo para hacer referencia a él en otros gráficos. Para ello, publicaremos el nodo localmente. Este es un proceso similar al de la publicación de un paquete, que se explicará con más detalle en el siguiente capítulo.

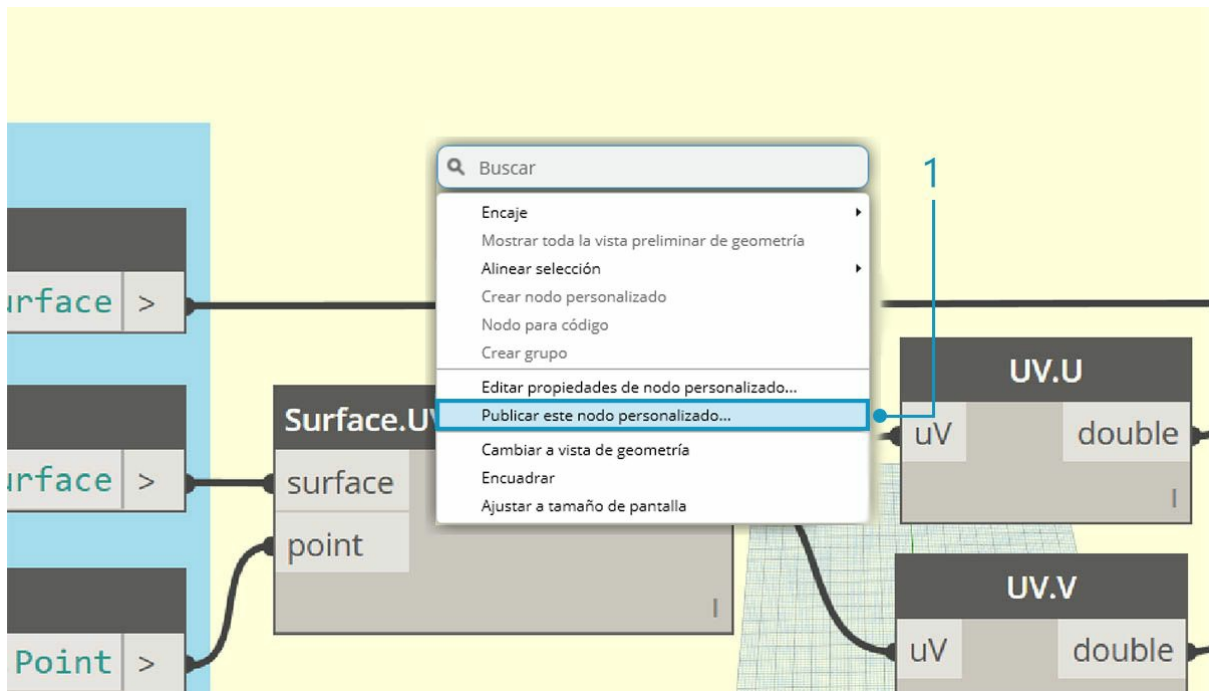
Publicación local de un nodo personalizado

Avancemos con el nodo personalizado que hemos creado en la sección anterior. Al publicar un nodo localmente, se podrá acceder a este en la biblioteca de Dynamo cuando se abra una nueva sesión. Si no se publica un nodo, un gráfico de Dynamo que haga referencia a un nodo personalizado deberá incluir ese nodo personalizado en su carpeta (o este deberá importarse a Dynamo mediante *Archivo > Importar biblioteca*).

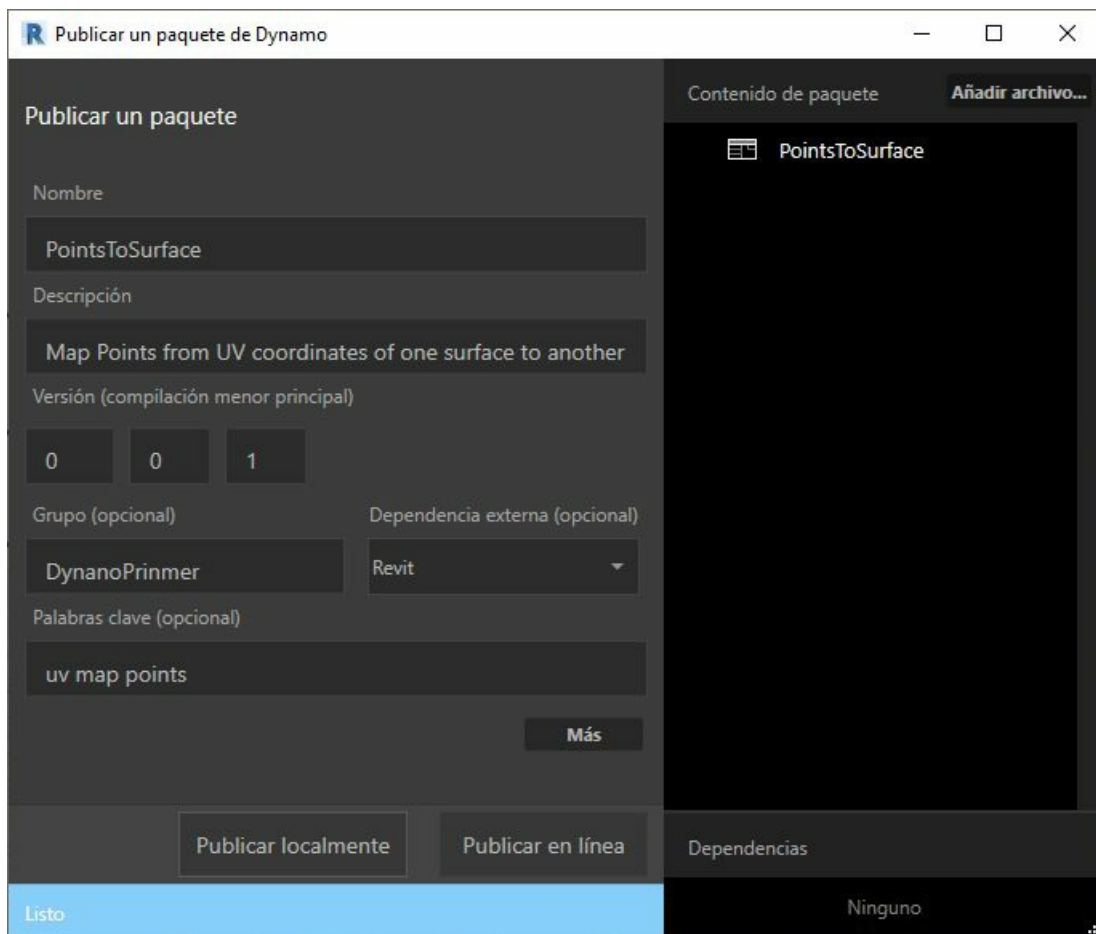
Descargue el archivo de ejemplo que acompaña a este ejercicio (haga clic con el botón derecho y seleccione "Guardar enlace como..."). En el Apéndice, se incluye una lista completa de los archivos de ejemplo. [PointsToSurface.dyf](#)



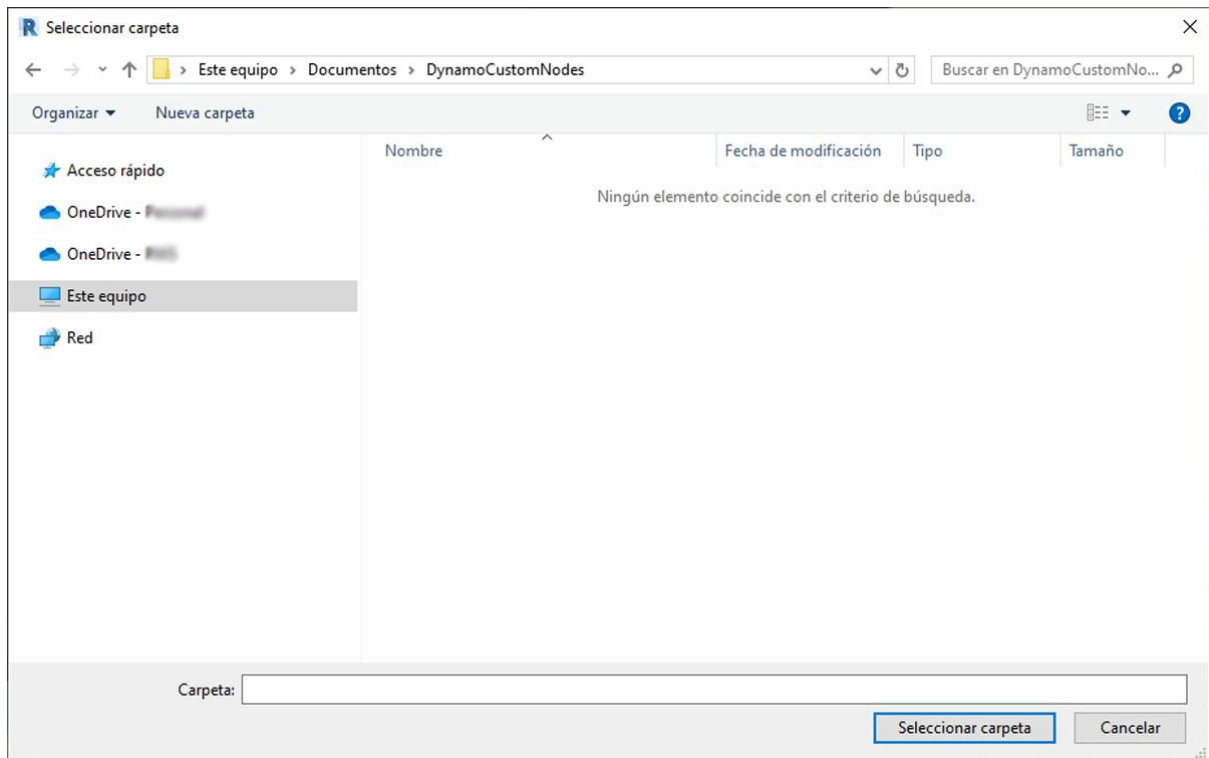
Después de abrir el nodo personalizado PointsToSurface, aparecerá el gráfico sobre el editor de nodos personalizados de Dynamo. También puede abrir un nodo personalizado haciendo doble clic en él en el editor de gráficos de Dynamo.



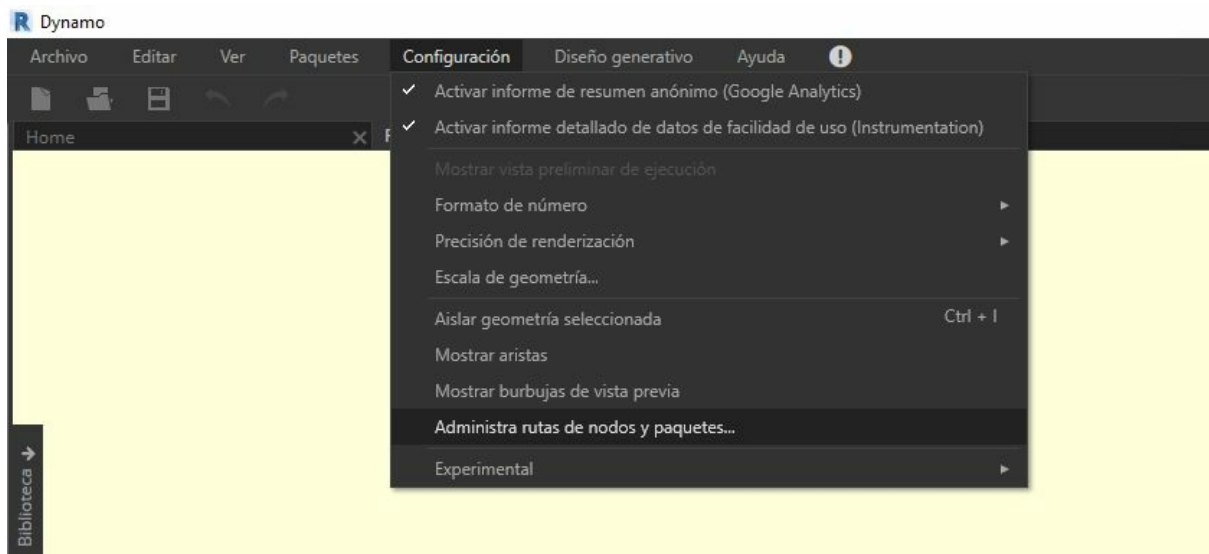
1. Para publicar localmente un nodo personalizado, haga clic con el botón derecho en el lienzo y seleccione *Publicar este nodo personalizado...*



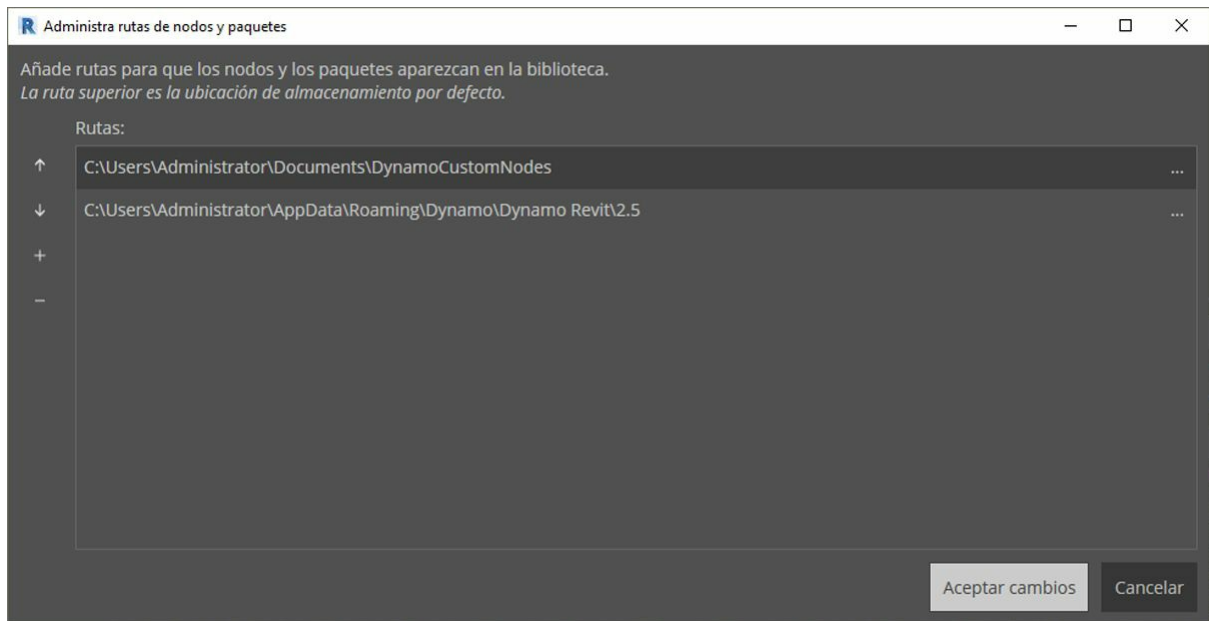
Rellene la información pertinente similar a la imagen anterior y seleccione *Publicar localmente*.. Observe que el campo Grupo define el elemento principal al que se puede acceder desde el menú de Dynamo.



Seleccione una carpeta para alojar todos los nodos personalizados que tiene previsto publicar localmente. Dynamo comprobará esta carpeta cada vez que se cargue, por lo que debe asegurarse de que esta se encuentre en una ubicación permanente. Vaya a esta carpeta y elija *Seleccionar carpeta*. El nodo de Dynamo se publica ahora localmente y permanecerá en la barra de herramientas de Dynamo cada vez que cargue el programa.

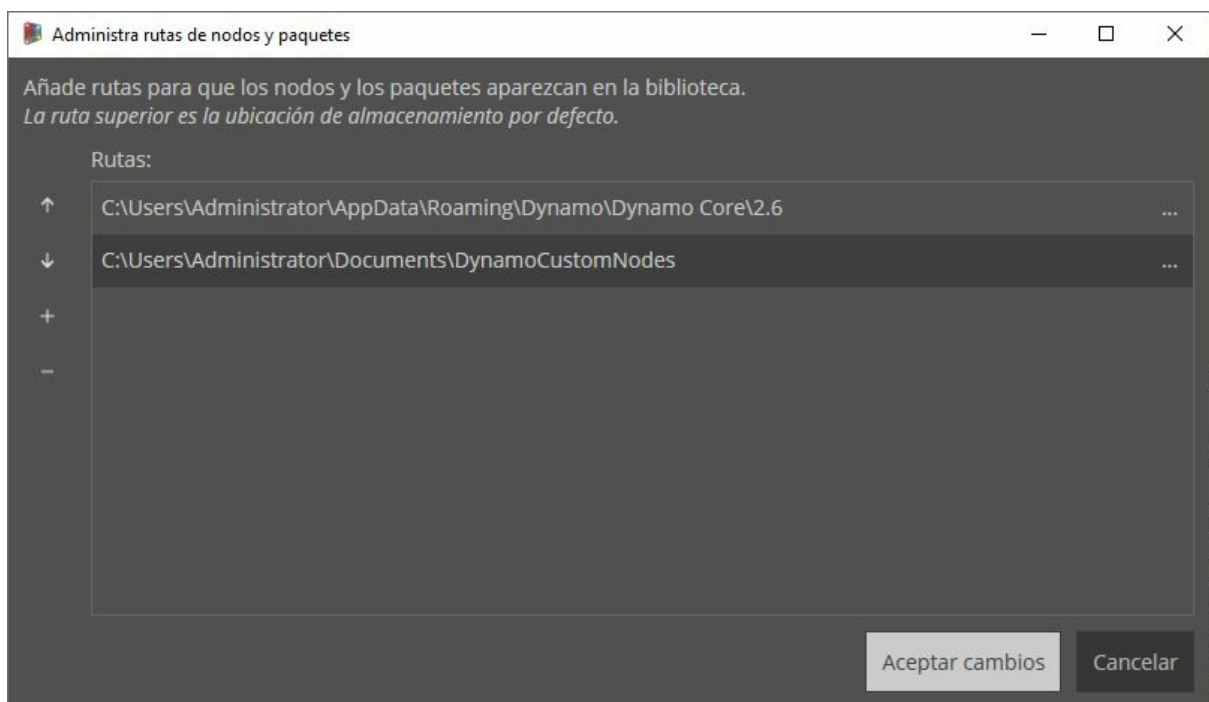


1. Para comprobar la ubicación de la carpeta de nodos personalizados, vaya a *Configuración > Administrar rutas de nodos y paquetes...*

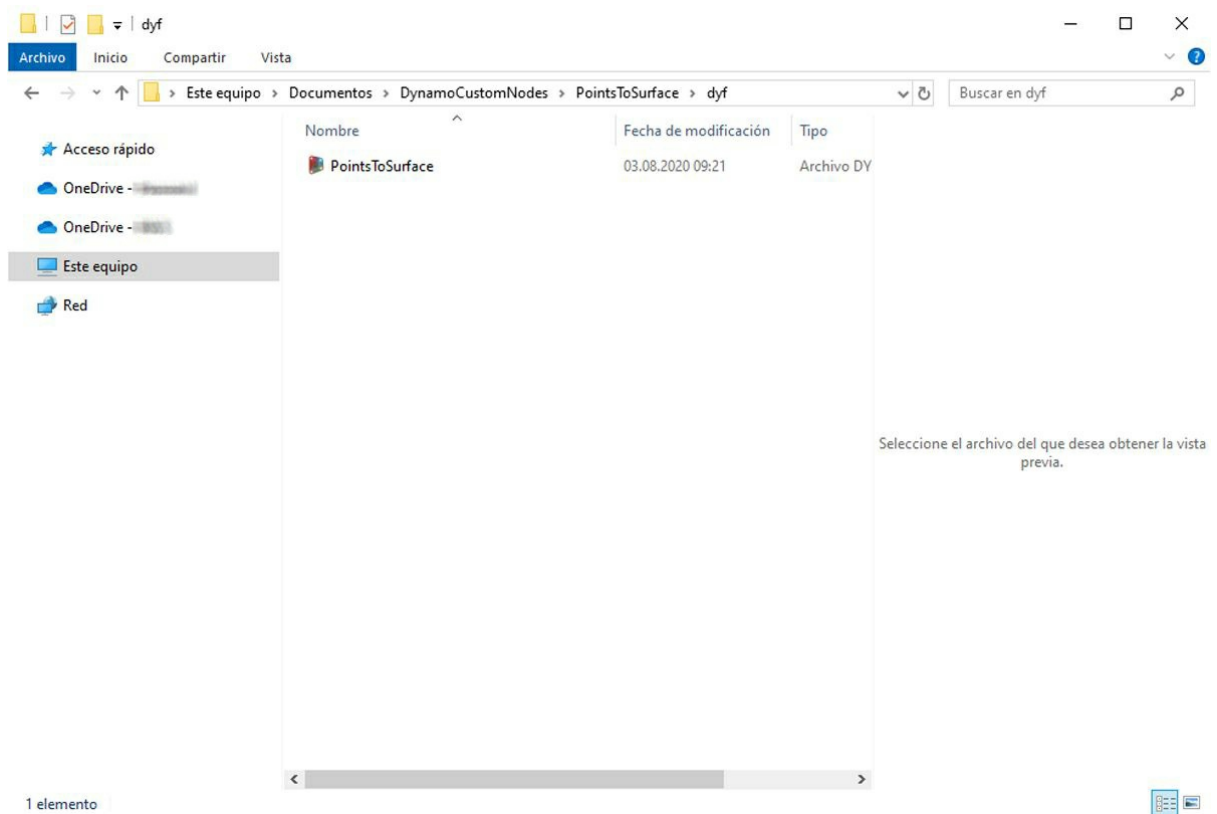


En esta ventana, aparecen dos rutas: *AppData\Roaming\Dynamo...* hace referencia a la ubicación por defecto de los paquetes de Dynamo instalados en línea. *Documents\DynamoCustomNodes...* hace referencia a la ubicación de los nodos personalizados que se han publicado localmente.

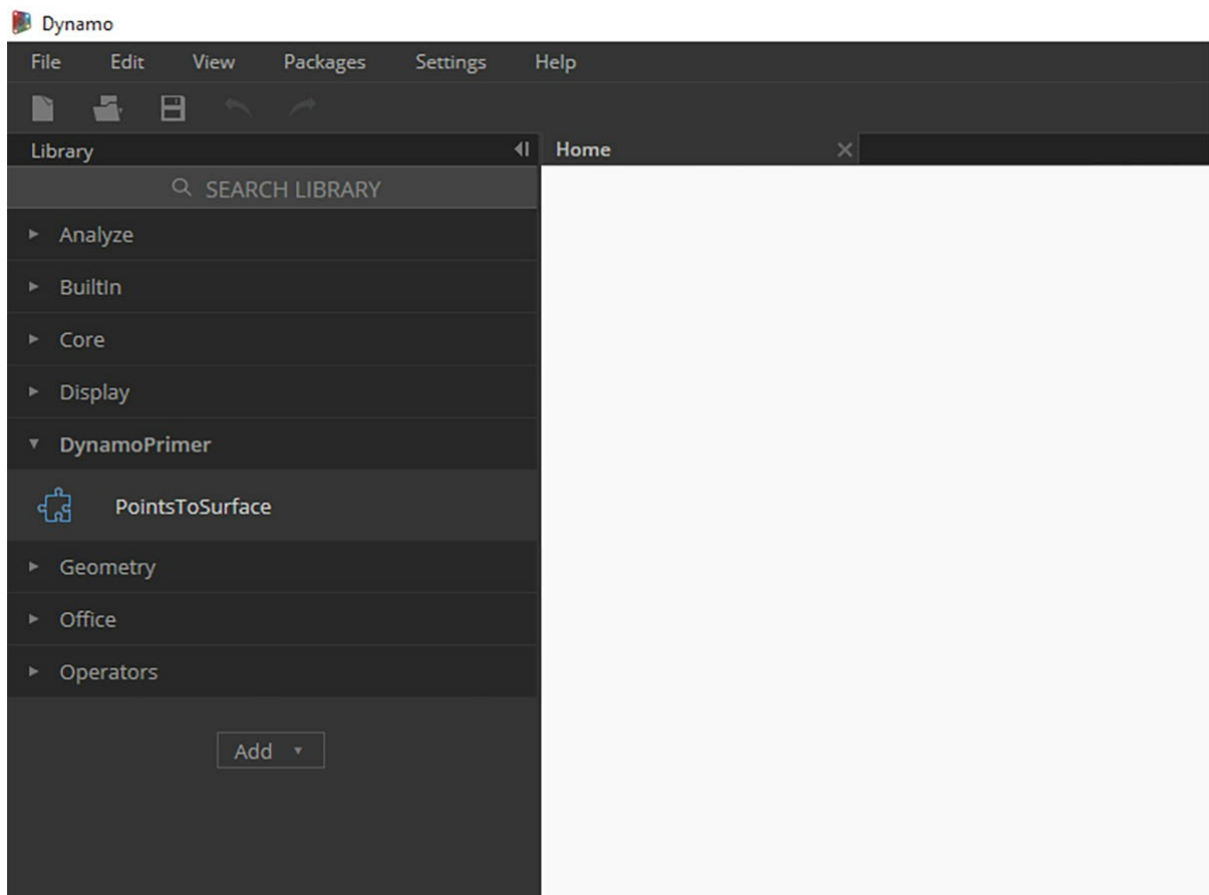
1. Es posible que desee desplazar la ruta de carpeta local hacia abajo en el orden de la lista anterior. (Para ello, seleccione la ruta de carpeta y haga clic en la flecha hacia abajo situada a la izquierda de los nombres de ruta). La carpeta superior es la ruta por defecto para las instalaciones de paquetes. Por lo tanto, al mantener la ruta de instalación por defecto de paquetes de Dynamo como la carpeta por defecto, los paquetes en línea se separarán de los nodos publicados localmente.*



Cambiamos el orden de los nombres de ruta para que la ruta por defecto de Dynamo sea la ubicación de instalación de paquetes.



Al acceder a esta carpeta local, podemos encontrar el nodo personalizado original en la carpeta *.dyf*, que es la extensión de un archivo de nodo personalizado de Dynamo. Podemos editar el archivo en esta carpeta; el nodo se actualizará en la interfaz de usuario. También podemos añadir más nodos a la carpeta principal de *DynamoCustomNode*; Dynamo los añadirá a la biblioteca al reiniciar.



Dynamo ahora se cargará cada vez con *PointsToSurface* en el grupo *DynamoPrimer* de la biblioteca de Dynamo.

Nodos de Python

Python

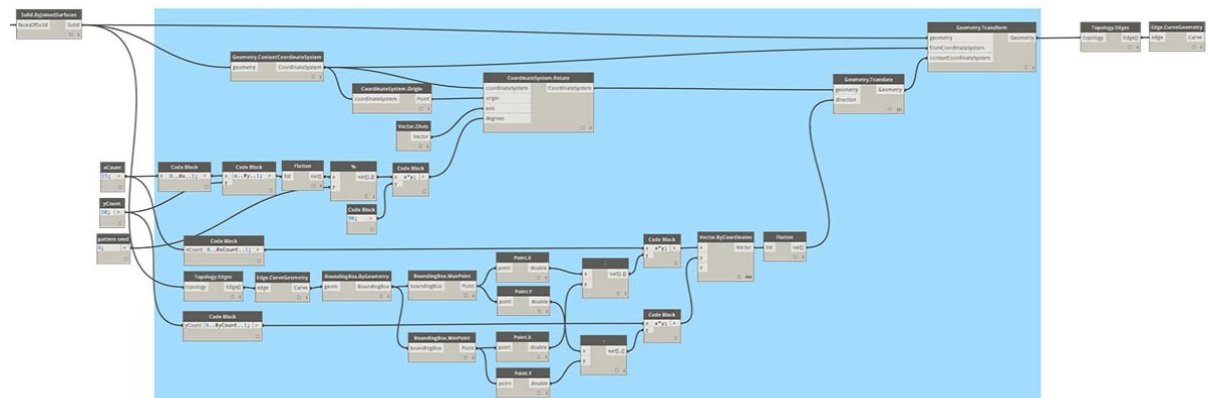


Python es un lenguaje de programación muy utilizado cuya popularidad tiene mucho que ver con su estilo de sintaxis. Es altamente legible, lo que hace que su aprendizaje sea más sencillo que el de muchos otros idiomas. Python admite módulos y paquetes, y se puede incrustar en aplicaciones existentes. Los ejemplos de esta sección presuponen una familiaridad básica con Python. Para obtener información sobre cómo empezar a trabajar con Python, un buen recurso es la página ["Getting Started"](#) (Guía de introducción) de [Python.org](#).

Programación visual frente a textual

¿Por qué habría que utilizar la programación textual en el entorno de programación visual de Dynamo? Como hemos explicado en el capítulo 1.1, la programación visual tiene muchas ventajas. Permite crear programas sin necesidad de aprender sintaxis especial en una interfaz visual intuitiva. Sin embargo, un programa visual se puede sobrecargar y, a veces, su funcionalidad puede ser reducida. Por ejemplo, Python ofrece métodos mucho más eficaces para escribir instrucciones condicionales (si/entonces) y bucles. Python es una potente herramienta que permite ampliar las funciones de Dynamo y reemplazar muchos nodos por unas pocas líneas de código concisas.

Programa visual:



Programa textual:

```
import clr
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *

solid = IN[0]
seed = IN[1]
xCount = IN[2]
yCount = IN[3]

solids = []

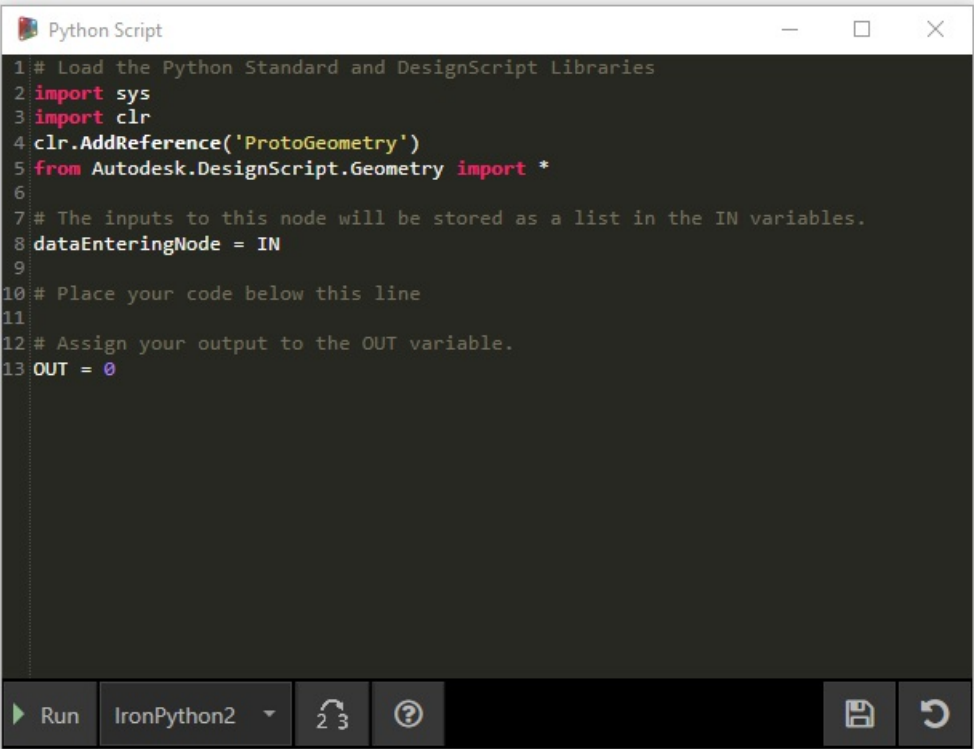
yDist = solid.BoundingBox.MaxPoint.Y-solid.BoundingBox.MinPoint.Y
xDist = solid.BoundingBox.MaxPoint.X-solid.BoundingBox.MinPoint.X

for i in xrange:
    for j in xrange:
        fromCoord = solid.ContextCoordinateSystem
        toCoord = fromCoord.Rotate(solid.ContextCoordinateSystem.Origin, Vector.ByCoordinates(0,0,1),(90*(i+j%val)
        vec = Vector.ByCoordinates((xDist*i),(yDist*j),0)
        toCoord = toCoord.Translate(vec)
        solids.append(solid.Transform(fromCoord,toCoord))

OUT = solids
```

El nodo de Python

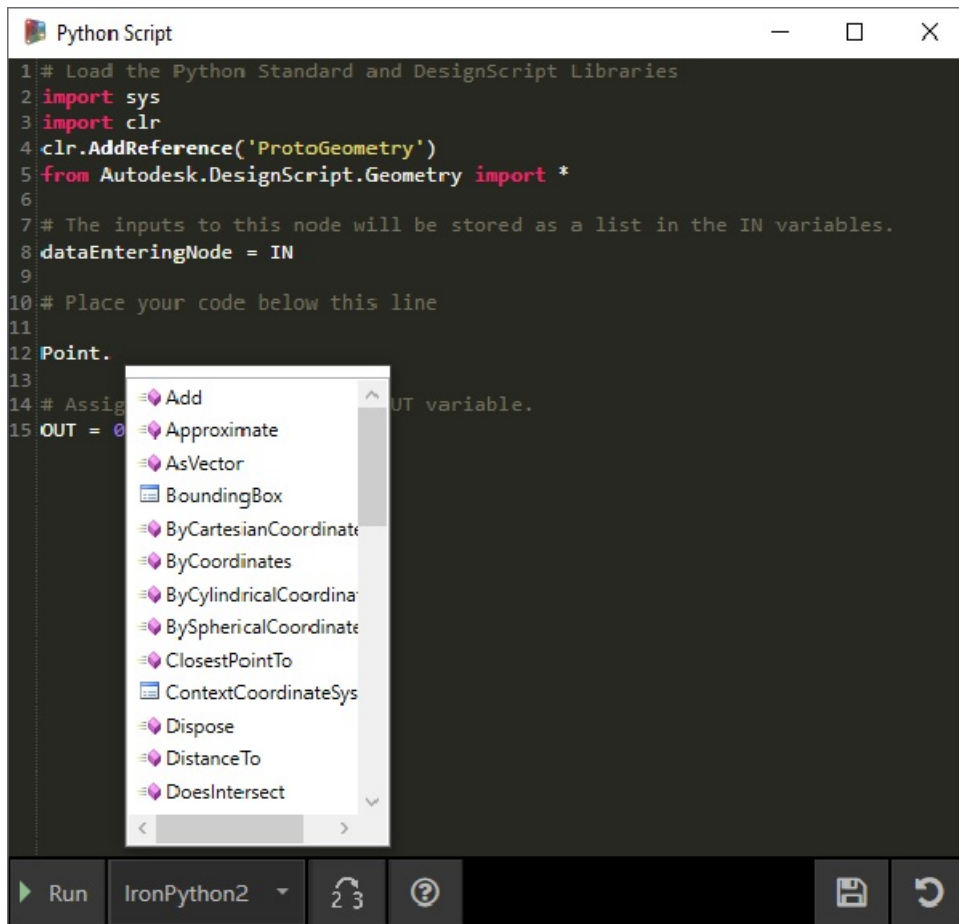
Al igual que los bloques de código, los nodos de Python son una interfaz de secuencias de comandos dentro de un entorno de programación visual. El nodo de Python se encuentra en *Core > Scripting* en la biblioteca. Al hacer doble clic en el nodo, se abre el editor de secuencias de comandos de Python (también puede hacer clic con el botón derecho en el nodo y seleccionar *Editar...*).



```
1 # Load the Python Standard and DesignScript Libraries
2 import sys
3 import clr
4 clr.AddReference('ProtoGeometry')
5 from Autodesk.DesignScript.Geometry import *
6
7 # The inputs to this node will be stored as a list in the IN variables.
8 dataEnteringNode = IN
9
10 # Place your code below this line
11
12 # Assign your output to the OUT variable.
13 OUT = 0
```

Observará que aparece texto modelo en la parte superior como ayuda para hacer referencia a las bibliotecas necesarias. Las entradas se almacenan en la matriz IN. Los valores se devuelven a Dynamo asignándolos a la variable OUT.

La biblioteca Autodesk.DesignScript.Geometry permite utilizar la notación de puntos como ocurre con los bloques de código. Para obtener más información sobre la sintaxis de Dynamo, consulte el capítulo 7.2 y el [manual de DesignScript](#). Al escribir un tipo de geometría como, por ejemplo, "Point.", se muestra una lista de métodos para crear y consultar puntos.

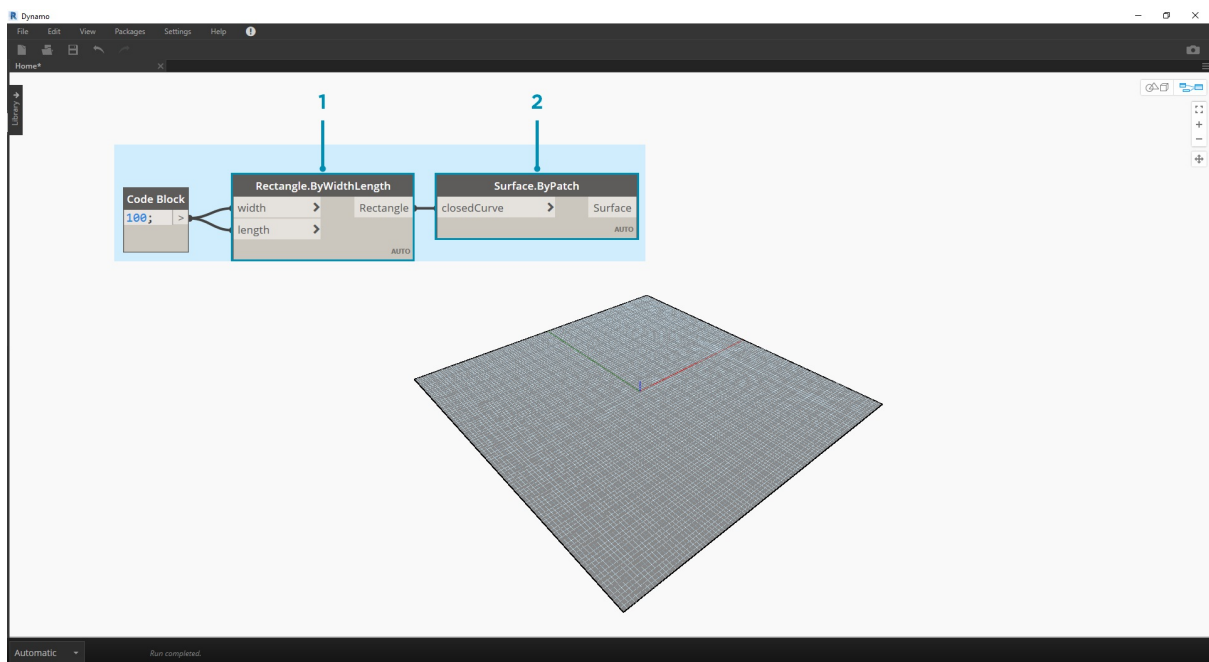


Los métodos incluyen constructores como *ByCoordinates*, acciones como *Add* y consultas como las coordenadas X, Y y Z.

Ejercicio

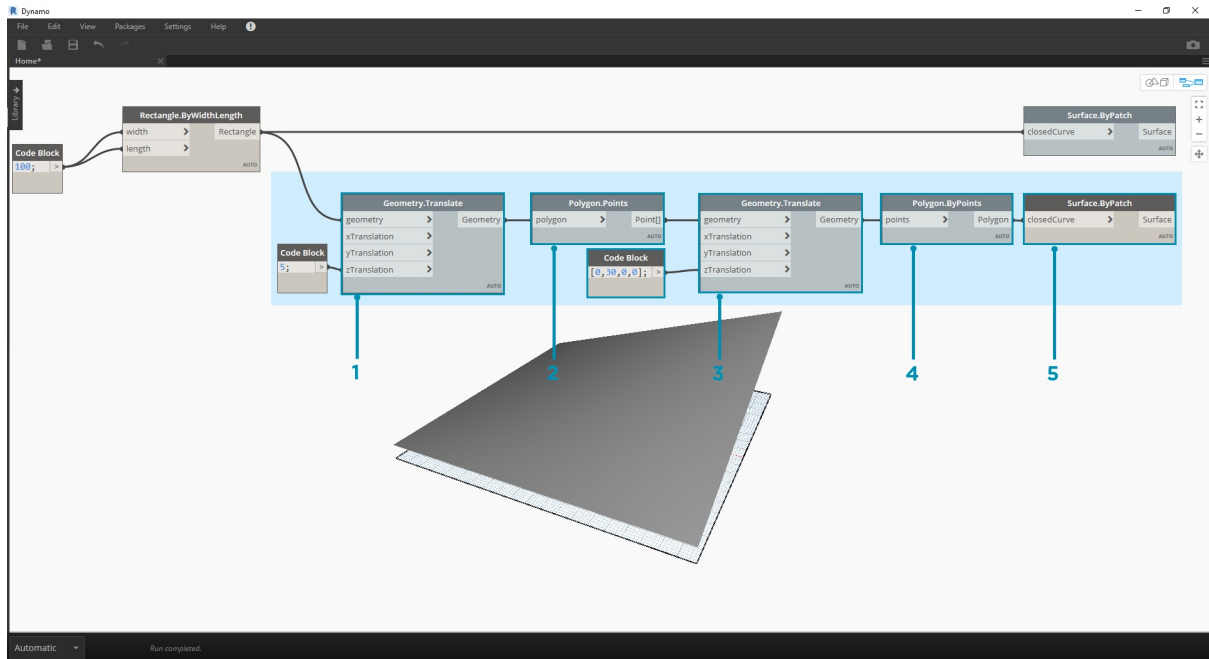
Descargue el archivo de ejemplo que acompaña a este ejercicio (haga clic con el botón derecho y seleccione "Guardar vínculo como..."). En el Apéndice se incluye una lista completa de los archivos de ejemplo. [Python_Custom-Node.dyn](#)

En este ejemplo, vamos a escribir una secuencia de comandos de Python que crea patrones a partir de un módulo sólido y lo vamos a convertir en un nodo personalizado. Primero, vamos a crear nuestro módulo sólido mediante nodos de Dynamo.



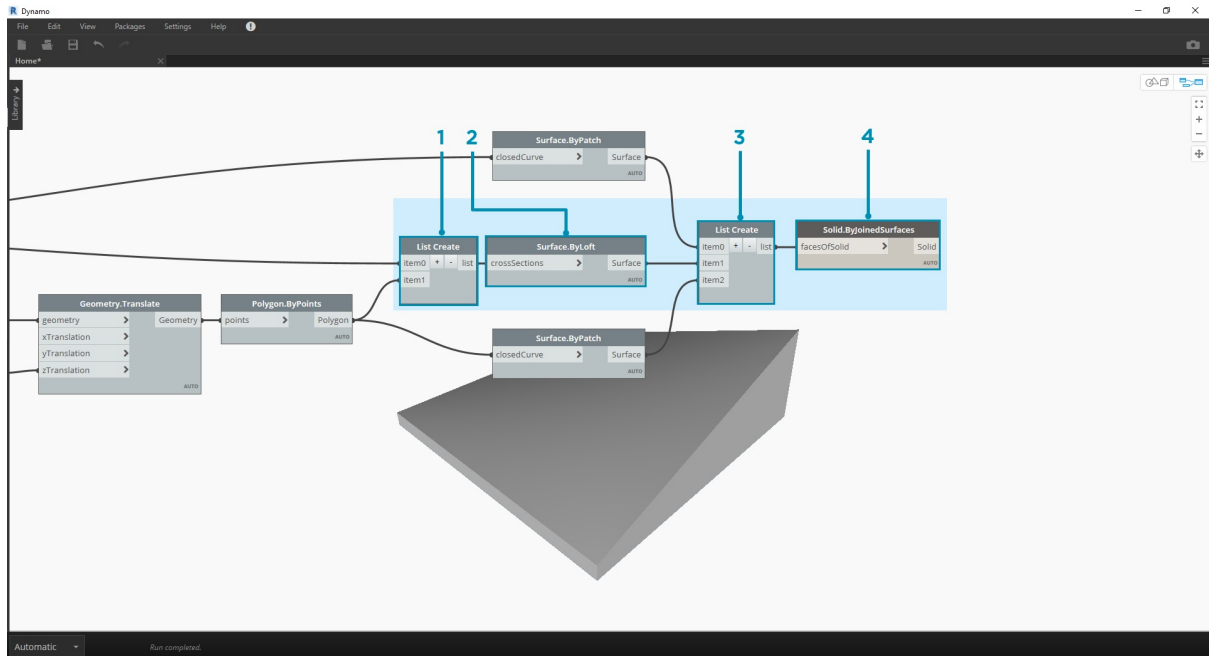
1. **Rectangle.ByWidthLength:** cree un rectángulo que será la base del sólido.

2. **Surface.ByPatch**: conecte el rectángulo con la entrada "closedCurve" para crear la superficie inferior.



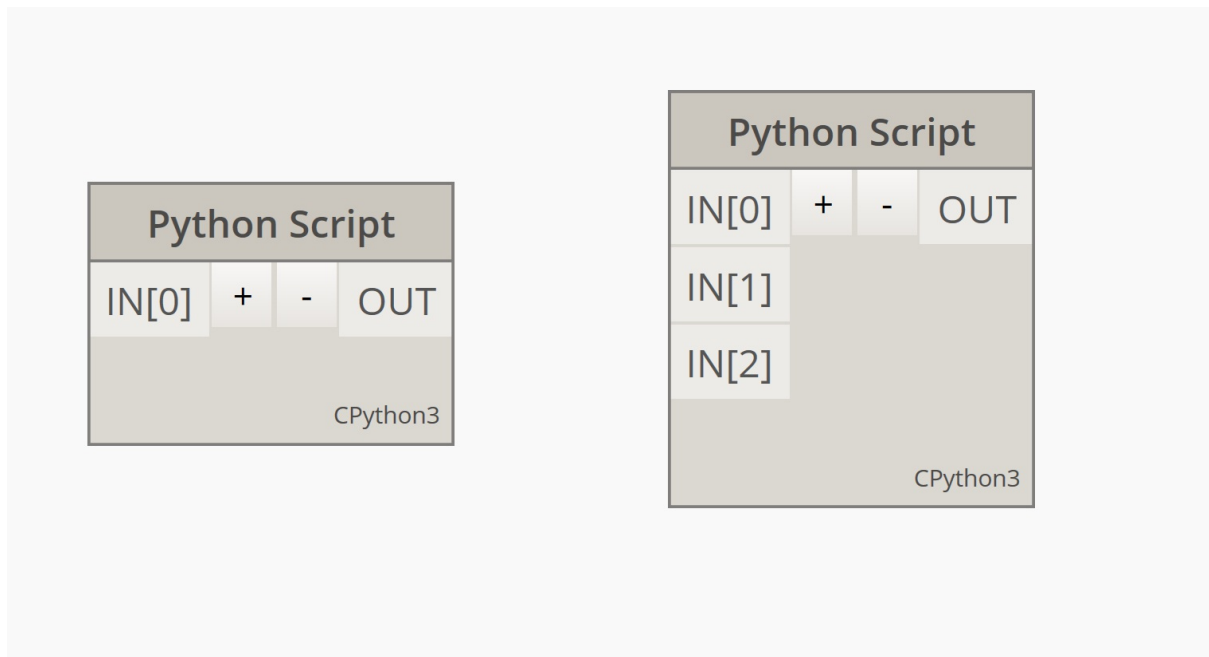
1. **Geometry.Translate**: conecte el rectángulo a la entrada "geometry" para desplazarlo hacia arriba y utilice un bloque de código para especificar el grosor de base del sólido.
2. **Polygon.Points**: consulte el rectángulo trasladado para extraer los puntos de esquina.
3. **Geometry.Translate**: utilice un bloque de código para crear una lista de cuatro valores correspondientes a los cuatro puntos y traslade una esquina del sólido hacia arriba.
4. **Polygon.ByPoints**: utilice los puntos trasladados para reconstruir el polígono superior.
5. **Surface.ByPatch**: conecte el polígono para crear la superficie superior.

Ahora que tenemos las superficies superior e inferior, solevaremos los dos perfiles para crear los lados del sólido.



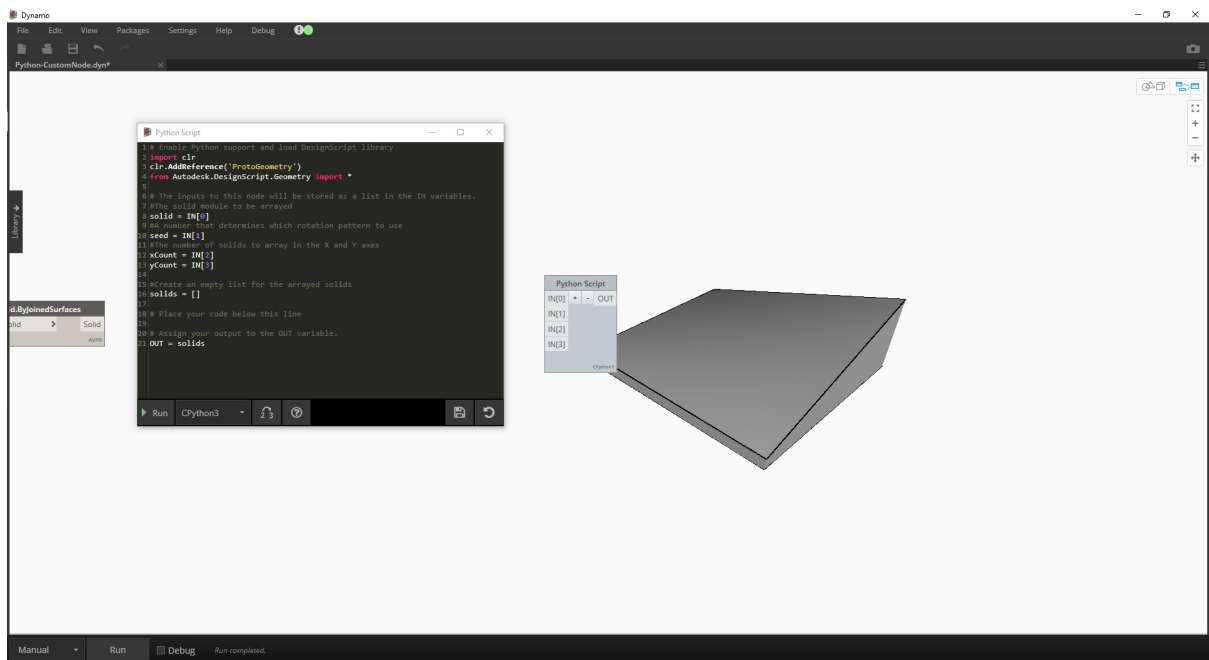
1. **List.Create**: conecte el rectángulo inferior y el polígono superior a las entradas de índice.
2. **Surface.ByLoft**: soleve los dos perfiles para crear los lados del sólido.
3. **List.Create**: conecte las superficies superior, lateral e inferior a las entradas de índice para crear una lista de superficies.
4. **Solid.ByJoinedSurfaces**: una las superficies para crear el módulo sólido.

Ahora que ya tenemos el sólido, soltaremos un nodo de secuencia de comandos de Python en el espacio de trabajo.



Para añadir entradas adicionales al nodo, cierre el editor y haga clic en el icono + del nodo. Las entradas se denominan IN[0], IN[1], etc., para indicar que representan elementos de una lista.

Empezaremos definiendo las entradas y la salida. Haga doble clic en el nodo para abrir el editor de Python.



```
# Enable Python support and load DesignScript library
import clr
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *
```

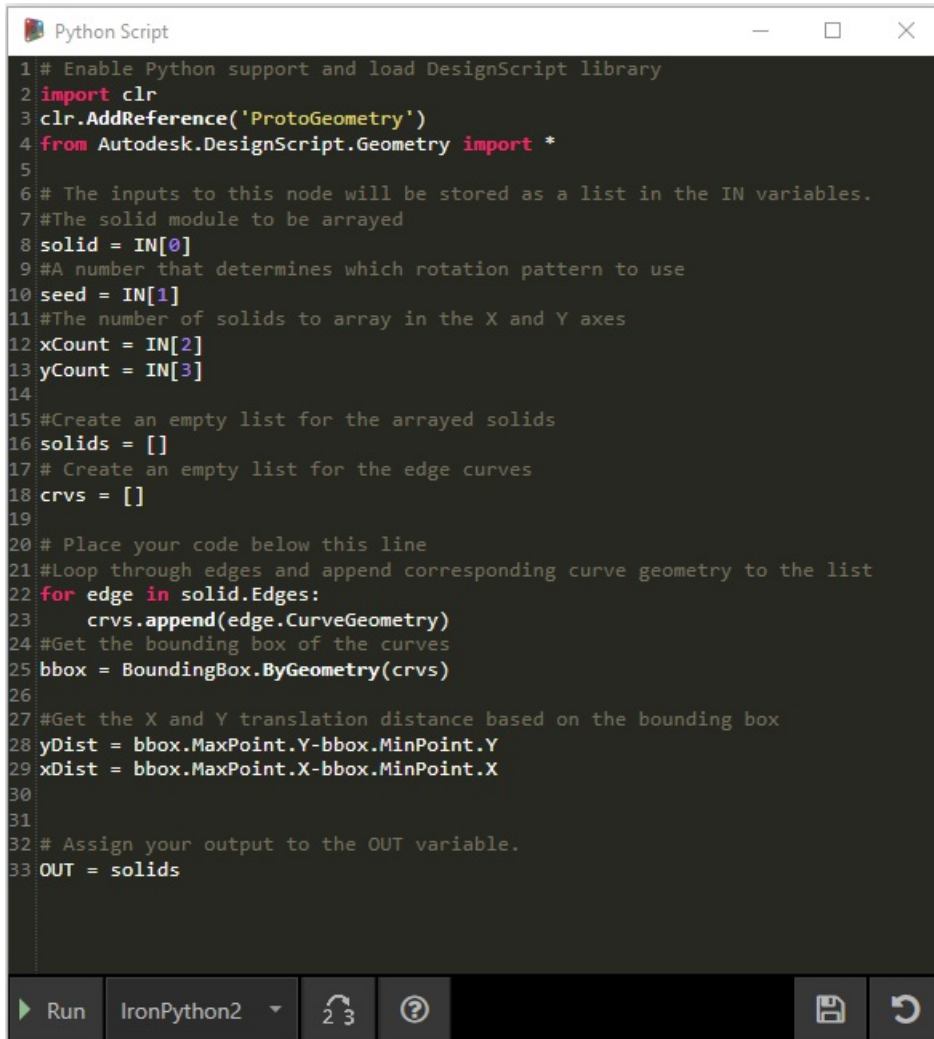
```
# The inputs to this node will be stored as a list in the IN variables.
#The solid module to be arrayed
solid = IN[0]
#A number that determines which rotation pattern to use
seed = IN[1]
#The number of solids to array in the X and Y axes
xCount = IN[2]
yCount = IN[3]
```

```
#Create an empty list for the arrayed solids
solids = []
```

```
# Place your code below this line
```

```
# Assign your output to the OUT variable.
OUT = solids
```

Este código cobrará más sentido a medida que avancemos en el ejercicio. A continuación, debemos pensar en qué información se requiere para disponer la matriz de nuestro módulo sólido. En primer lugar, es necesario conocer las dimensiones del sólido para determinar la distancia de traslación. Debido a un error del cuadro delimitador, tendremos que utilizar la geometría de curva de borde para crear un cuadro delimitador.



```
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
32 # Assign your output to the OUT variable.
33 OUT = solids
```

Echemos un vistazo al nodo de Python en Dynamo. Observe que estamos utilizando la misma sintaxis que vemos en los títulos de los nodos de Dynamo. El código comentado se encuentra a continuación.

```
# 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]

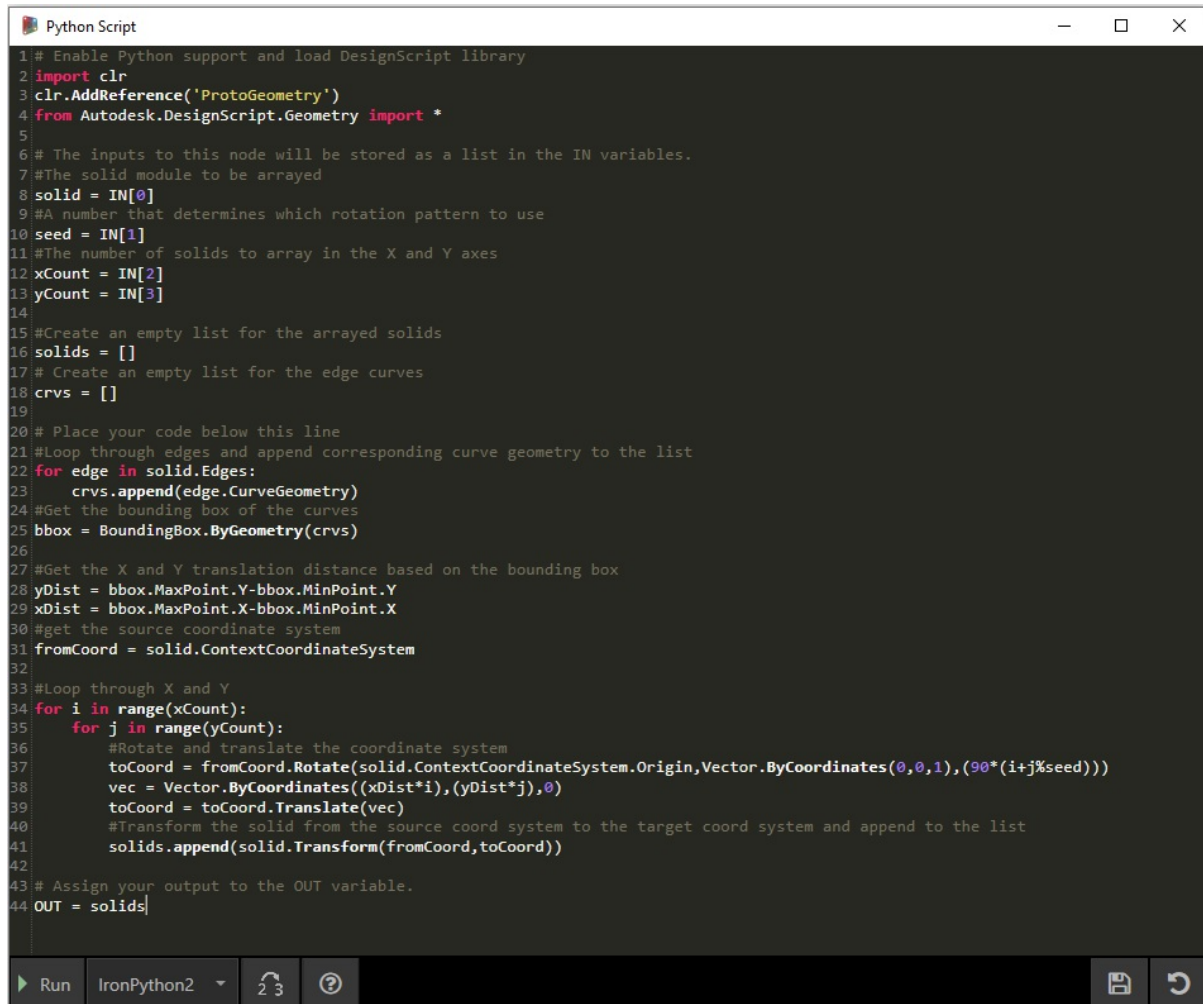
#Create an empty list for the arrayed solids
solids = []
# Create an empty list for the edge curves
crvs = []

# 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
```

Ya que vamos a trasladar y girar los módulos sólidos, vamos a utilizar la operación Geometry.Transform. Al observar el nodo Geometry.Transform, vemos que vamos a necesitar un sistema de coordenadas de origen y un sistema de coordenadas de destino para transformar el sólido. El origen es el sistema de coordenadas de contexto del sólido, mientras que el destino será un sistema de coordenadas diferente para cada módulo de matriz. Esto significa que tendremos que crear un bucle a través de los valores de X e Y para transformar el sistema de coordenadas de forma diferente en cada caso.



```
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
```

Echemos un vistazo al nodo de Python en Dynamo. El código comentado se encuentra a continuación.

```
# Enable Python support and load DesignScript library
import clr
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *

# The inputs to this node will be stored as a list in the IN variables.
#The solid module to be arrayed
solid = IN[0]
#A number that determines which rotation pattern to use
seed = IN[1]
#The number of solids to array in the X and Y axes
xCount = IN[2]
yCount = IN[3]

#Create an empty list for the arrayed solids
solids = []
# Create an empty list for the edge curves
crvs = []

# Place your code below this line
#Loop through edges and append corresponding curve geometry to the list
for edge in solid.Edges:
```

```

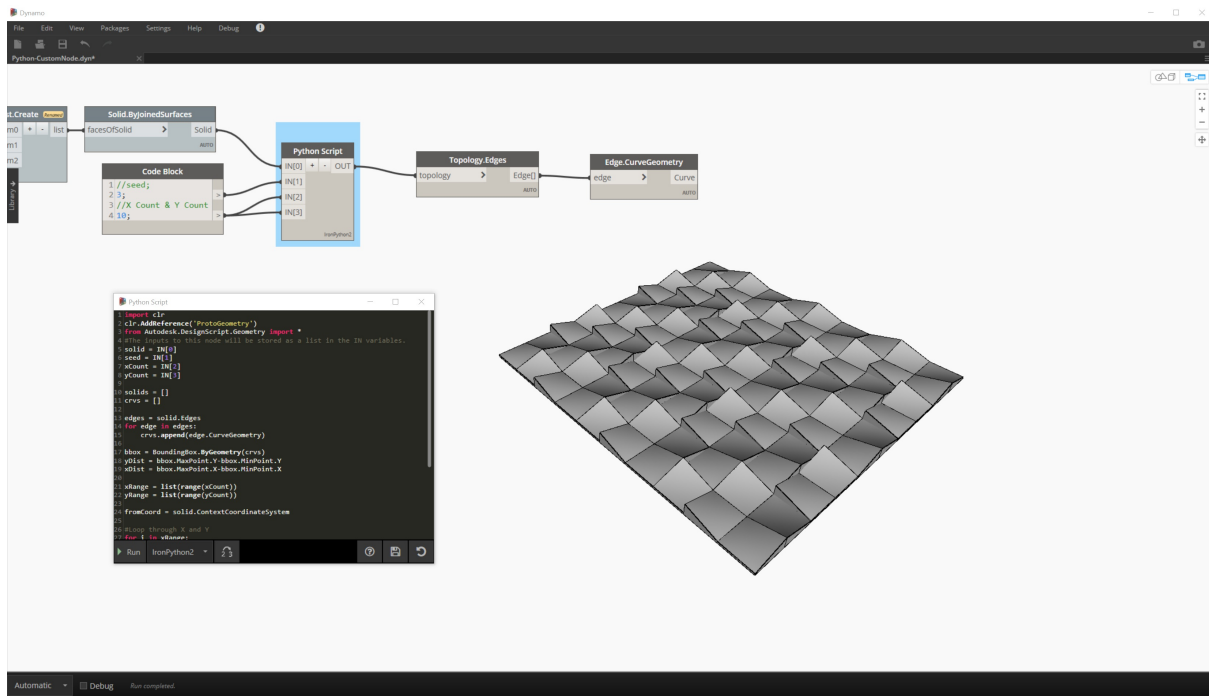
crvs.append(edge.CurveGeometry)
#Get the bounding box of the curves
bbox = BoundingBox.ByGeometry(crvs)

#Get the X and Y translation distance based on the bounding box
yDist = bbox.MaxPoint.Y-bbox.MinPoint.Y
xDist = bbox.MaxPoint.X-bbox.MinPoint.X
#get the source coordinate system
fromCoord = solid.ContextCoordinateSystem

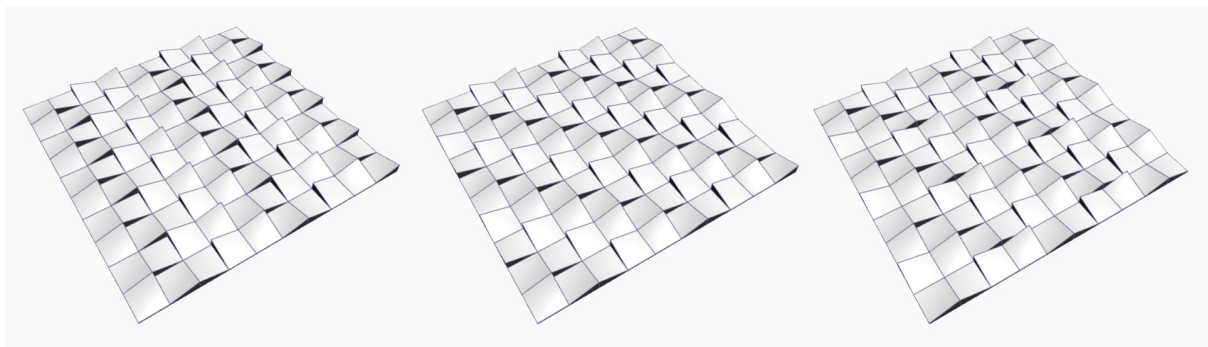
#Loop through X and Y
for i in range(xCount):
for j in range(yCount):
#Rotate and translate the coordinate system
toCoord = fromCoord.Rotate(solid.ContextCoordinateSystem.Origin,Vector.ByCoordinates(0,0,1),(90*(i+j%seec
vec = Vector.ByCoordinates((xDist*i),(yDist*j),0)
toCoord = toCoord.Translate(vec)
#Transform the solid from the source coord system to the target coord system and append to the list
solids.append(solid.Transform(fromCoord,toCoord))

# Assign your output to the OUT variable.
OUT = solids

```



Si hace clic en Ejecutar en el nodo de Python, se ejecutará el código.



Pruebe a cambiar el valor de semilla para crear patrones diferentes. También puede cambiar los parámetros del módulo sólido para obtener distintos efectos. En Dynamo 2.0, puede simplemente cambiar la semilla y hacer clic en Ejecutar sin cerrar la ventana de Python.

Ahora que hemos creado una secuencia de comandos de Python útil, vamos a guardarla como un nodo personalizado. Seleccione el nodo de secuencia de comandos de Python, haga clic con el botón derecho sobre el mismo y seleccione "Nuevo nodo a partir de selección".

Propiedades de nodo personalizado

Propiedades de nodo personalizado **CN**

Los nodos personalizados se incluirán en la sección Complementos de la biblioteca.

Nombre

patternFromSolidModule

Descripción

Create patterns by arraying and rotating a solid module.

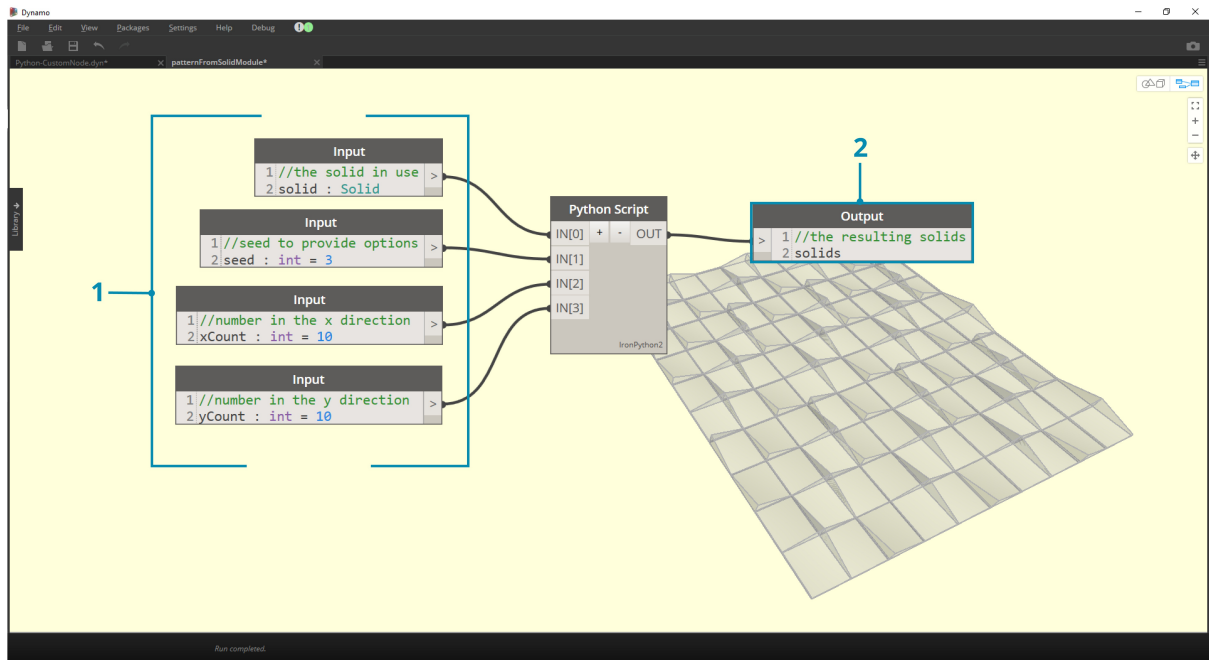
Categoría de complementos

DynamoPrimer.Geometry.Solid

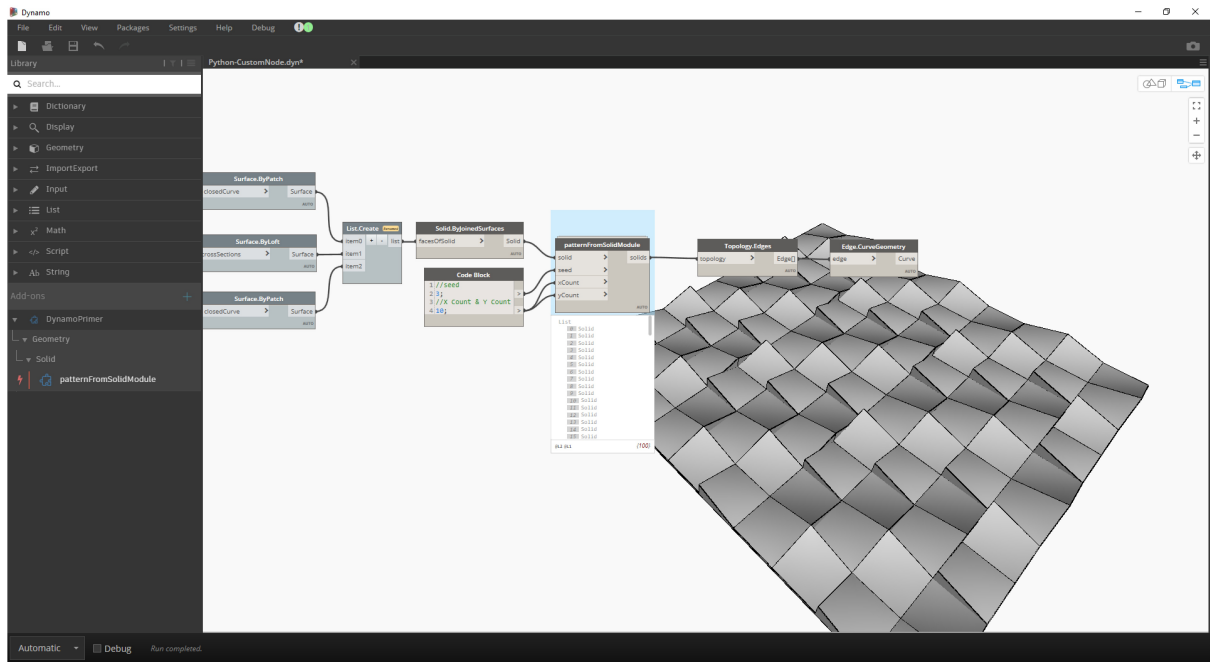
Aceptar Cancelar

Asigne un nombre, una descripción y una categoría.

Se abrirá un nuevo espacio de trabajo en el que se puede editar el nodo personalizado.



1. **Inputs:** cambie los nombres de las entradas para que sean más descriptivos y añada tipos de datos y valores por defecto.
2. **Output:** cambie el nombre de la salida y guarde el nodo como un archivo .dyf.



El nodo personalizado refleja los cambios que acabamos de realizar.

Python y Revit

Python y Revit

Ahora que hemos mostrado cómo utilizar las secuencias de comandos de Python en Dynamo, vamos a echar un vistazo a la conexión de bibliotecas de Revit en el entorno de secuencias de comandos. Recuerde que hemos importado nuestros nodos centrales de Dynamo con las tres primeras líneas del bloque de código que aparece más abajo. Para importar los nodos de Revit, los elementos de Revit y el administrador de documentos de Revit, solo hemos de añadir algunas líneas adicionales:

```
import clr
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *

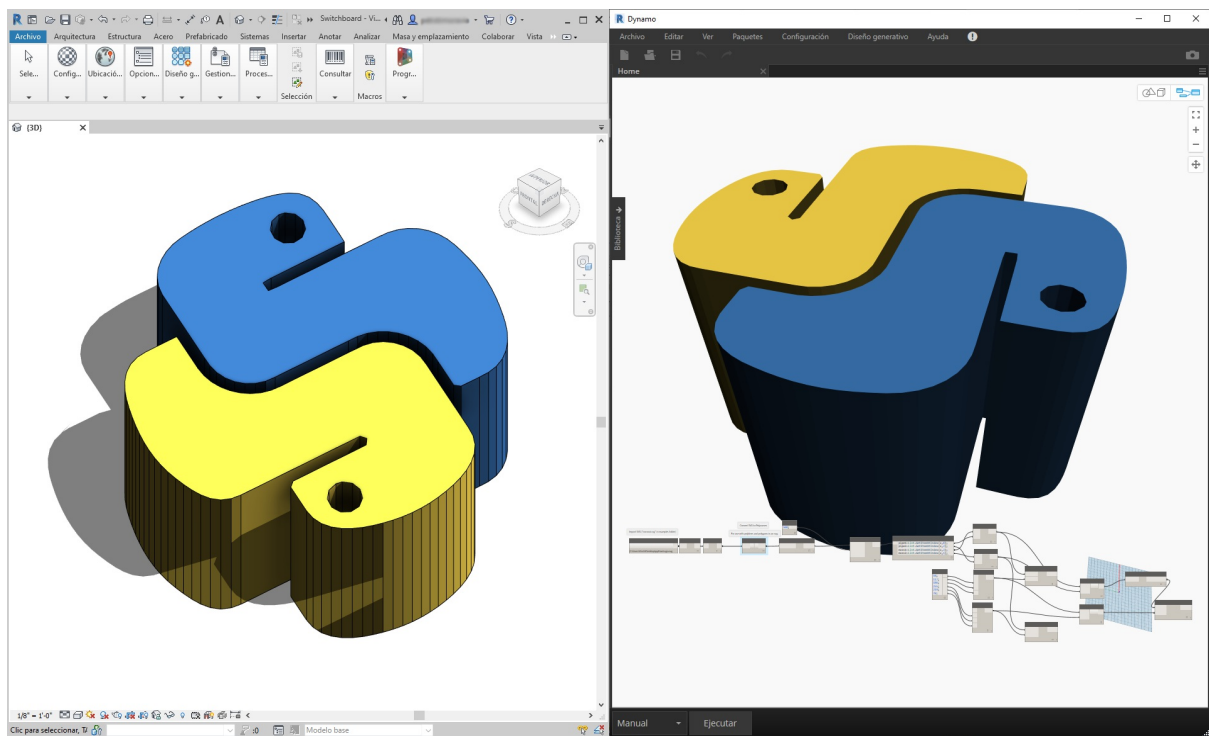
# Import RevitNodes
clr.AddReference("RevitNodes")
import Revit

# Import Revit elements
from Revit.Elements import *

# Import DocumentManager
clr.AddReference("RevitServices")
import RevitServices
from RevitServices.Persistence import DocumentManager

import System
```

Esto nos permite acceder a la API de Revit y disponer de secuencias de comandos personalizadas para cualquier tarea de Revit. La combinación del proceso de programación visual con las secuencias de comandos de la API de Revit, mejora significativamente la colaboración y el desarrollo de herramientas. Por ejemplo, un administrador de BIM y un diseñador de esquemas pueden trabajar juntos en el mismo gráfico. En esta colaboración, pueden mejorar el diseño y la ejecución del modelo.



API específicas de plataforma

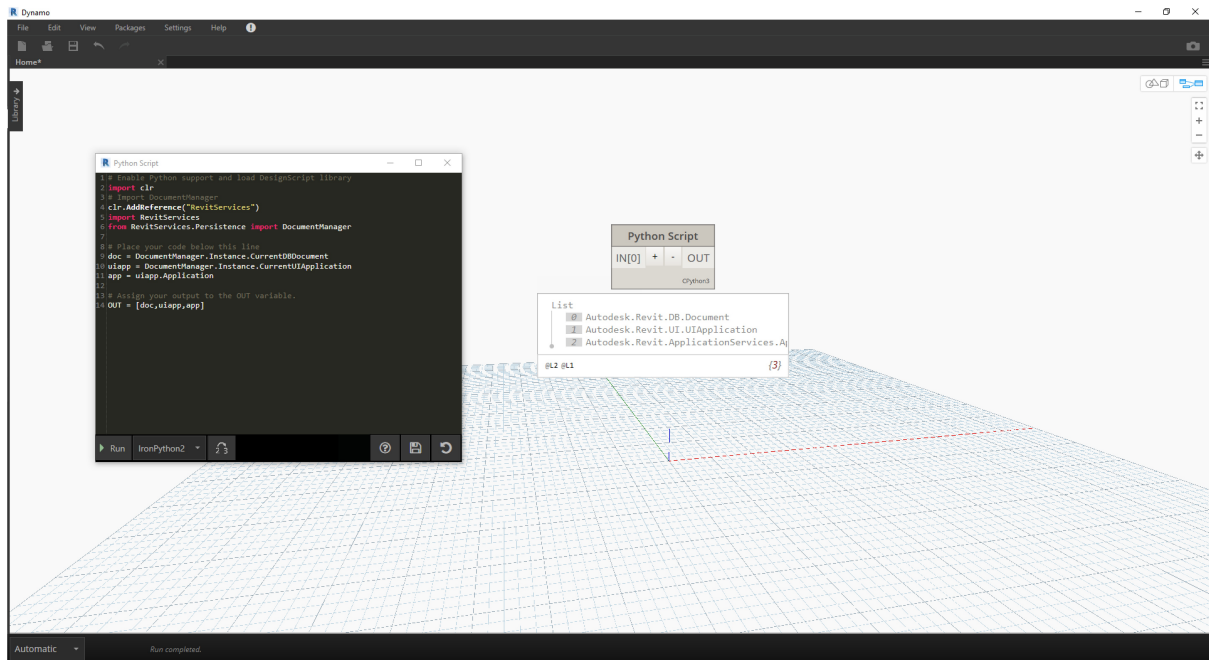
El plan tras el proyecto de Dynamo es ampliar el alcance de la implementación de plataformas. A medida que Dynamo añade más programas a la lista, los usuarios obtendrán acceso a las API específicas de plataforma desde el entorno de secuencias de comandos de Python. Aunque Revit representa el caso real de esta sección, podemos prever más capítulos en el futuro que ofrecerán módulos de aprendizaje completos sobre secuencias de comandos en otras plataformas. Además, hay muchas bibliotecas de [IronPython](#) a las que se puede acceder ahora y que se pueden importar a Dynamo.

Los ejemplos siguientes muestran métodos para implementar operaciones específicas de Revit desde Dynamo mediante el uso de Python. Para obtener una descripción más detallada de la relación de Python con Dynamo y Revit, consulte la [página wiki de Dynamo](#). Otro recurso útil para Python y Revit es el proyecto [Revit Python Shell](#).

Ejercicio 01

Cree un nuevo proyecto de Revit. Descargue el archivo de ejemplo que acompaña a este ejercicio (haga clic con el botón derecho y seleccione "Guardar enlace como..."). En el Apéndice se incluye una lista completa de los archivos de ejemplo. [Revit-Doc.dyn](#)

En estos ejercicios, vamos a explorar secuencias de comandos básicas de Python en Dynamo for Revit. El ejercicio se centrará en el uso de los archivos y elementos de Revit, así como en la comunicación entre Revit y Dynamo.



Este es un método sencillo para recuperar los elementos *doc*, *uiapp* y *app* del archivo de Revit vinculado a la sesión de Dynamo. Es posible que los programadores que hayan trabajado antes en la API de Revit reconozcan los elementos de la lista de inspección. Si estos elementos no le resultan familiares, no hay problema; vamos a usar otros ejemplos en los ejercicios siguientes.

A continuación se indica cómo vamos a importar los servicios de Revit y recuperar los datos del documento en Dynamo:

```
Python Script
1 # Enable Python support and load DesignScript library
2 import clr
3 # Import DocumentManager
4 clr.AddReference("RevitServices")
5 import RevitServices
6 from RevitServices.Persistence import DocumentManager
7
8 # Place your code below this line
9 doc = DocumentManager.Instance.CurrentDBDocument
10 uiapp = DocumentManager.Instance.CurrentUIApplication
11 app = uiapp.Application
12
13 # Assign your output to the OUT variable.
14 OUT = [doc,uiapp,app]
```

Echemos un vistazo al nodo de Python en Dynamo. El código comentado se encuentra a continuación.

```
# Enable Python support and load DesignScript library
import clr
# Import DocumentManager
clr.AddReference("RevitServices")
```

```
import RevitServices
from RevitServices.Persistence import DocumentManager

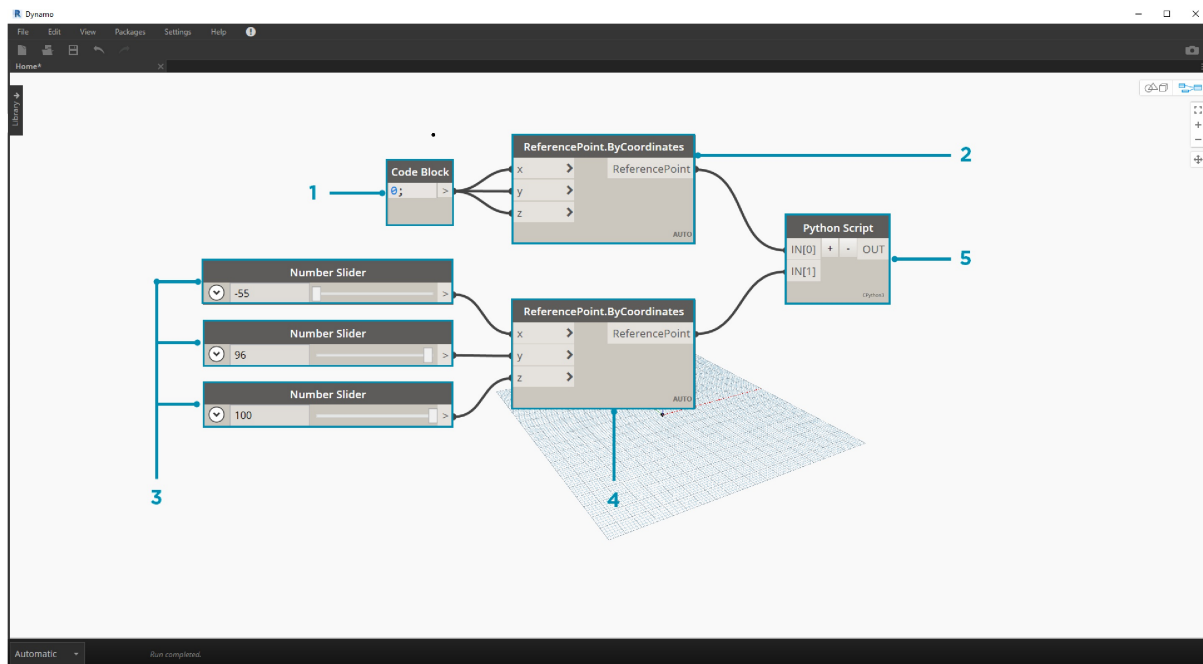
# Place your code below this line
doc = DocumentManager.Instance.CurrentDBDocument
uiapp = DocumentManager.Instance.CurrentUIApplication
app = uiapp.Application

# Assign your output to the OUT variable.
OUT = [doc, uiapp, app]
```

Ejercicio 02

Descargue los archivos de ejemplo que acompañan a este ejercicio (haga clic con el botón derecho y seleccione "Guardar enlace como..."). En el Apéndice se incluye una lista completa de los archivos de ejemplo. [Revit-ReferenceCurve.dyn](#)

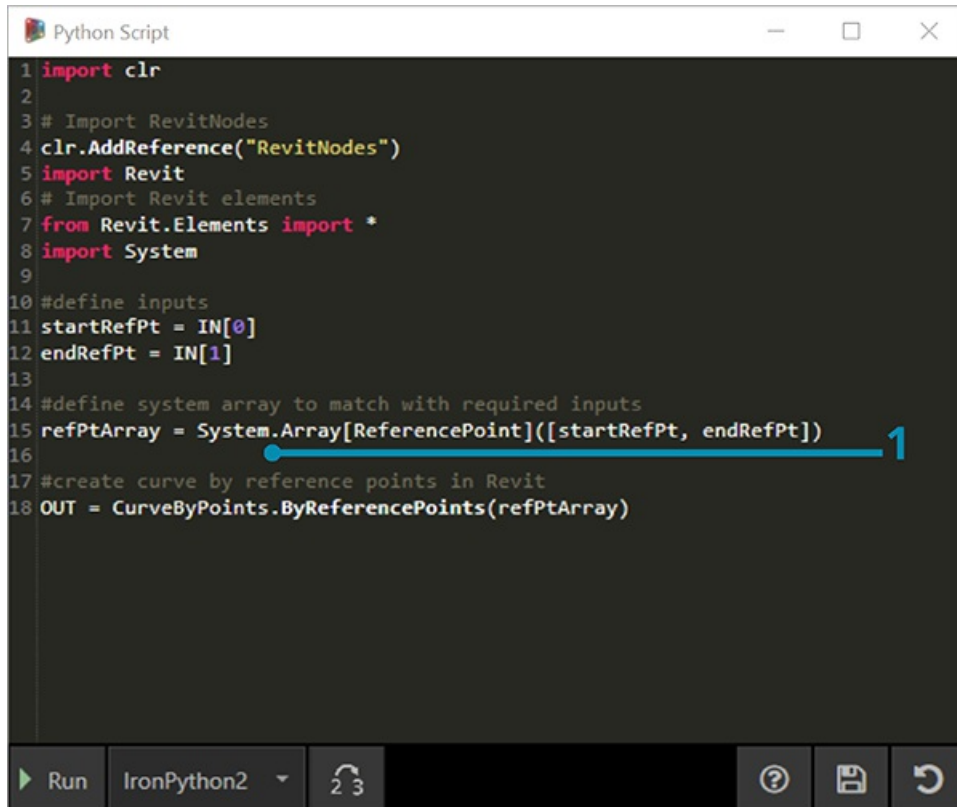
En este ejercicio, crearemos una curva de modelo simple en Revit mediante el nodo de Dynamo Python.



Comience con el conjunto de nodos de la imagen anterior. Primero crearemos dos puntos de referencia en Revit a partir de los nodos de Dynamo.

Empiece creando una nueva familia de masas conceptuales en Revit. Inicie Dynamo y cree el conjunto de nodos de la imagen anterior. Primero crearemos dos puntos de referencia en Revit a partir de los nodos de Dynamo.

1. Cree un bloque de código y asígnele un valor de "0;"
2. Conecte este valor a un nodo ReferencePoint.ByCoordinates para las entradas X, Y y Z.
3. Cree tres controles deslizantes que vayan de -100 a 100 con un tamaño de paso de 1.
4. Conecte cada control deslizante a un nodo ReferencePoint.ByCoordinates.
5. Añada un nodo de Python al espacio de trabajo, haga clic en el botón "+" del nodo para añadir otra entrada y conecte los dos puntos de referencia en cada entrada. Abra el nodo de Python.



Echemos un vistazo al nodo de Python en Dynamo. El código comentado se encuentra a continuación.

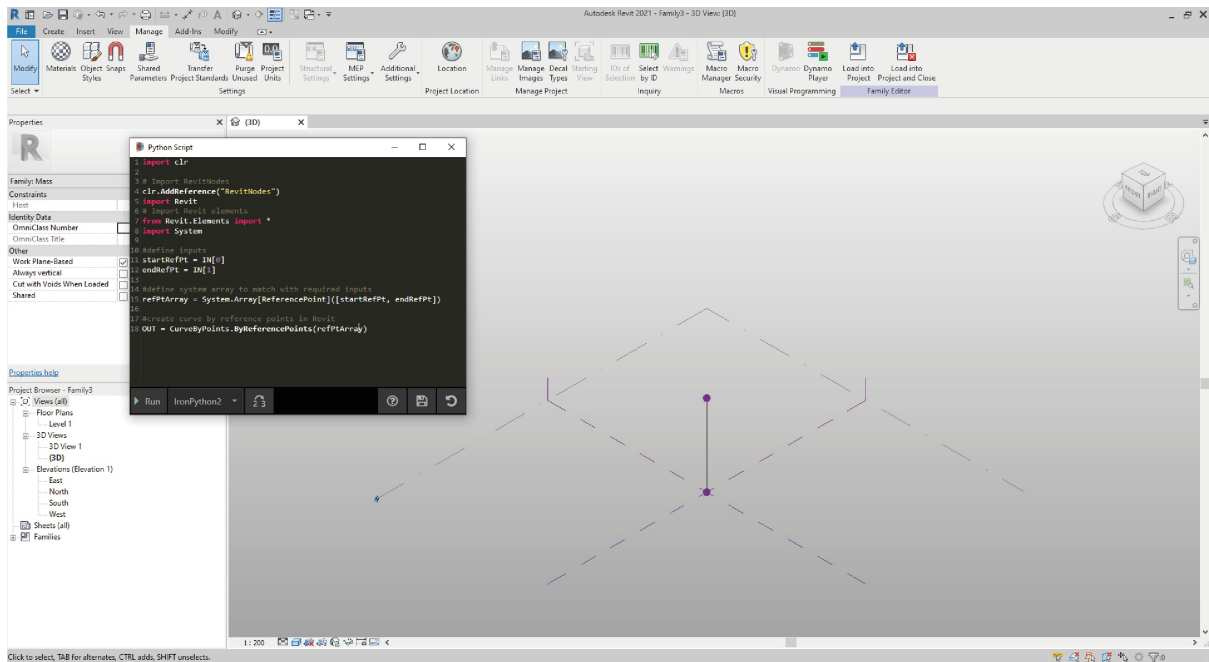
1. **System.Array**: Revit necesita una matriz del sistema como entrada (en lugar de una lista de Python). Esta es solo una línea de código más, pero prestar atención a los tipos de argumentos facilitará la programación de Python en Revit.

```
import clr

# Import RevitNodes
clr.AddReference("RevitNodes")
import Revit
# Import Revit elements
from Revit.Elements import *
import System

#define inputs
startRefPt = IN[0]
endRefPt = IN[1]

#define system array to match with required inputs
refPtArray = System.Array[ReferencePoint]([startRefPt, endRefPt])
#create curve by reference points in Revit
OUT = CurveByPoints.ByReferencePoints(refPtArray)
```

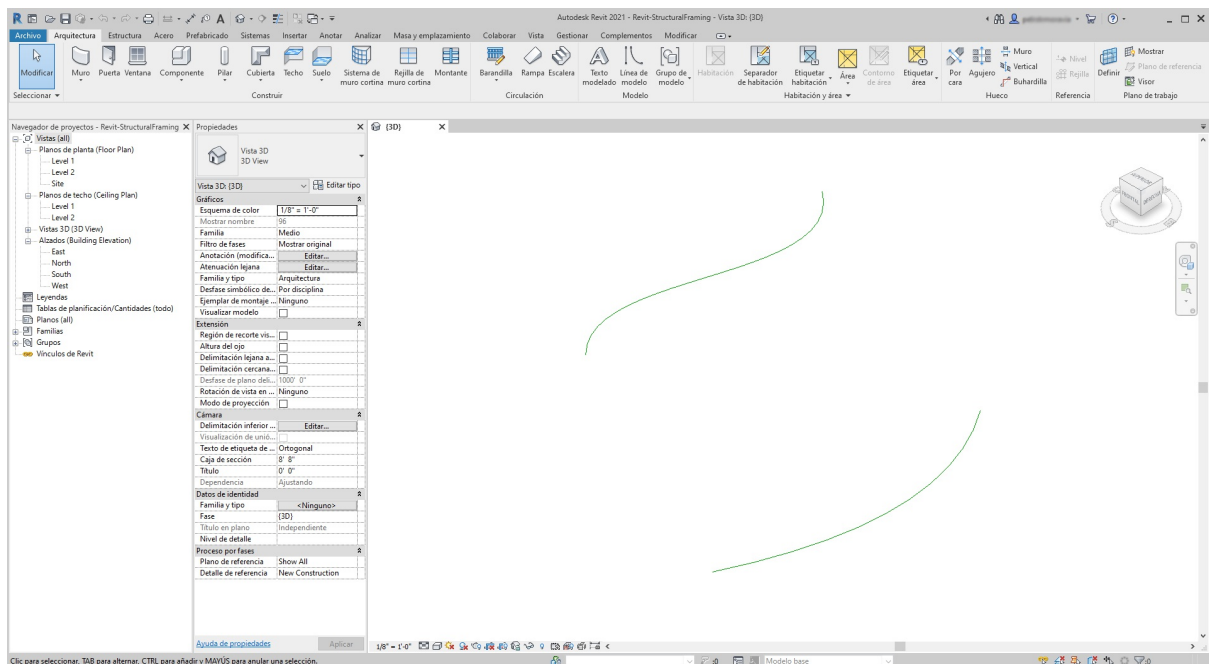


Desde Dynamo, hemos creado dos puntos de referencia con una línea que los conecta mediante Python. Vamos a ir un poco más lejos en el siguiente ejercicio.

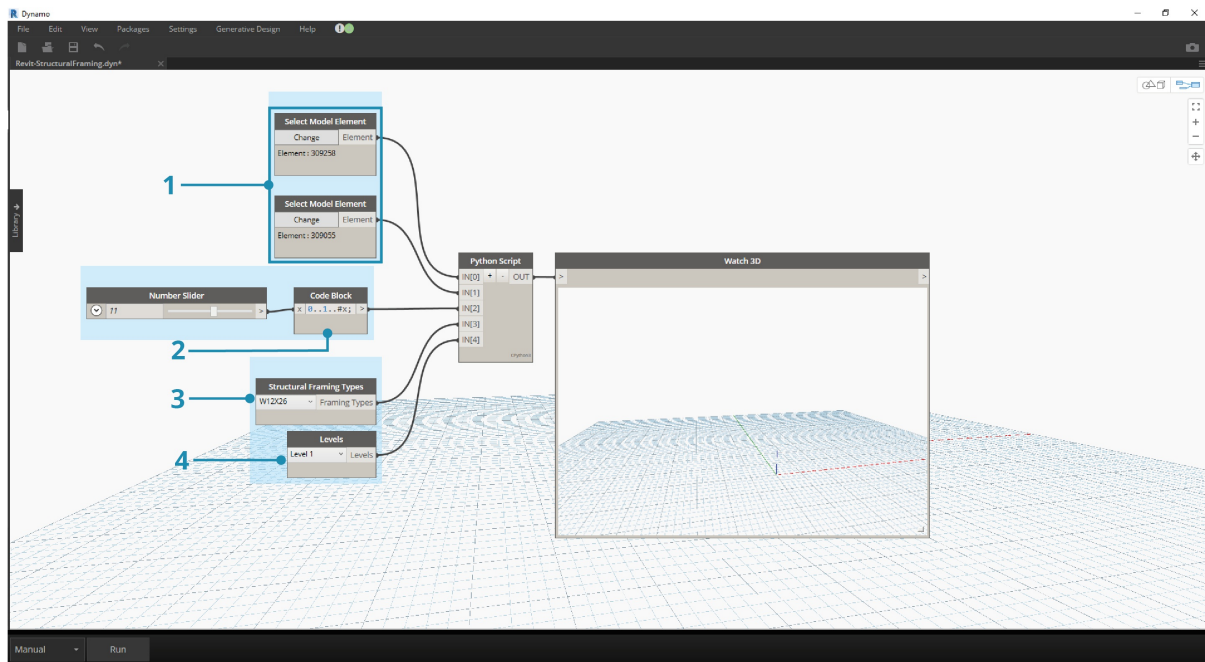
Ejercicio 03

Descargue y descomprima los archivos de ejemplo que acompañan a este ejercicio (haga clic con el botón derecho y seleccione "Guardar enlace como..."). En el Apéndice se incluye una lista completa de los archivos de ejemplo. [Revit-StructuralFraming.zip](#)

Este es un ejercicio sencillo, pero hace hincapié en la conexión de los datos y la geometría de Revit con Dynamo y viceversa. Comencemos abriendo Revit-StructuralFraming.rvt. Una vez abierto, cargue Dynamo y abra el archivo Revit-StructuralFraming.dyn.



Este archivo de Revit es totalmente básico. Dos curvas de referencia: una dibujada en el nivel 1 y la otra dibujada en el nivel 2. Queremos llevar estas curvas a Dynamo y mantener una conexión activa.



En este archivo, tenemos un conjunto de nodos conectados a cinco entradas de un nodo de Python.

1. **Nodos Select Model Element:** pulse el botón de selección para cada uno de ellos y seleccione la curva correspondiente en Revit.
2. **Code Block:** mediante la sintaxis "0..1..#x;", conecte un control deslizante de enteros que abarque de 0 a 20 en la entrada x. De este modo, se designa el número de vigas que se dibujarán entre las dos curvas.
3. **Structural Framing Types:** en el menú desplegable, elija la viga por defecto W12x26.
4. **Levels:** seleccione "Level 1".

```

Python Script
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)
  
```

Este código de Python es un poco más denso, pero los comentarios del código describen lo que sucede en el proceso:

```

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

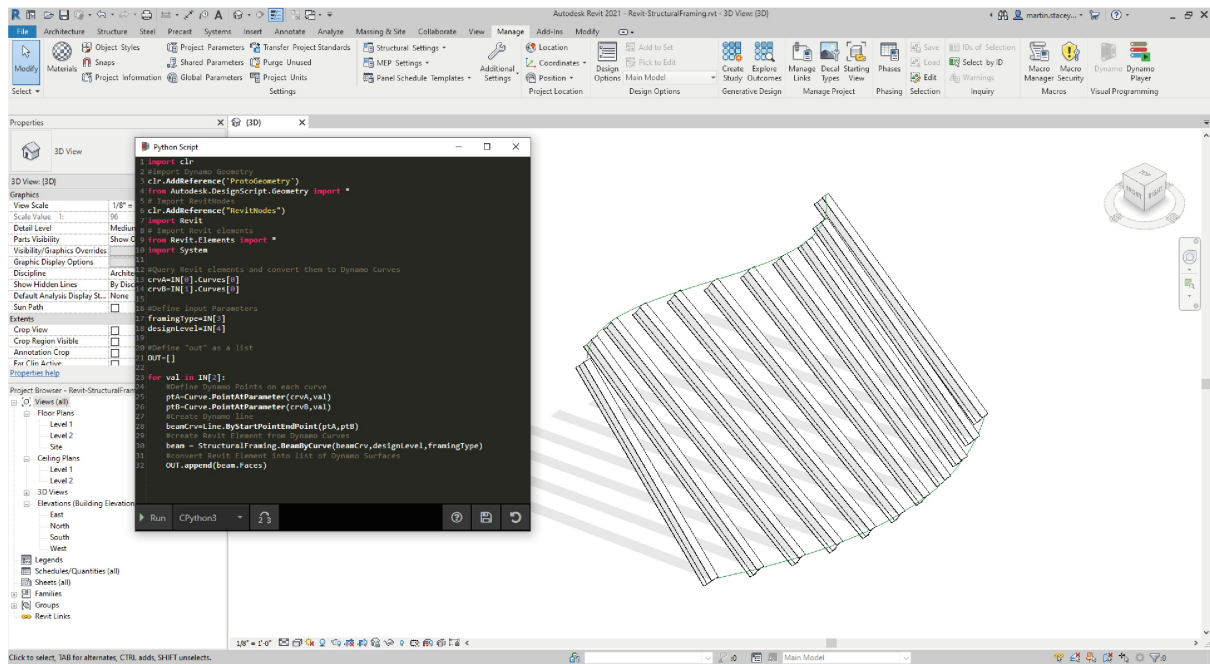
#Query Revit elements and convert them to Dynamo Curves
crvA=IN[0].Curves[0]
crvB=IN[1].Curves[0]

#Define input Parameters
framingType=IN[3]
designLevel=IN[4]

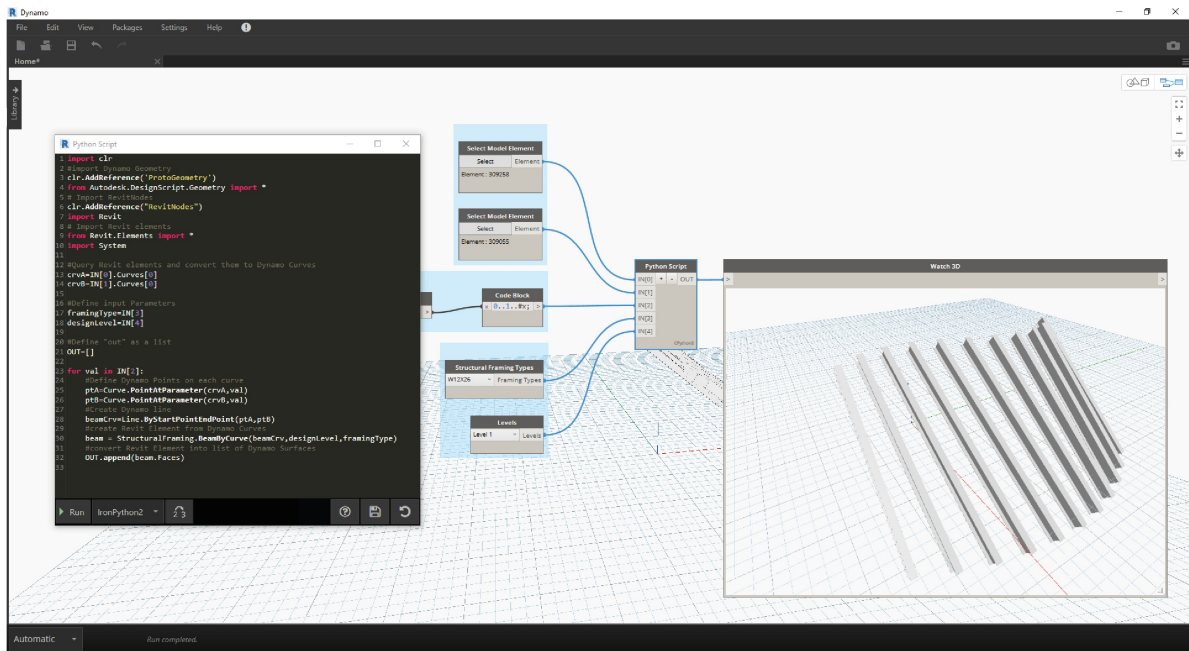
#Define "out" as a list
OUT=[]

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)

```



En Revit, tenemos una matriz de vigas que abarca las dos curvas como elementos estructurales. Nota: este no es un ejemplo realista; los elementos estructurales se utilizan como ejemplo de ejemplares nativos de Revit creados a partir de Dynamo.

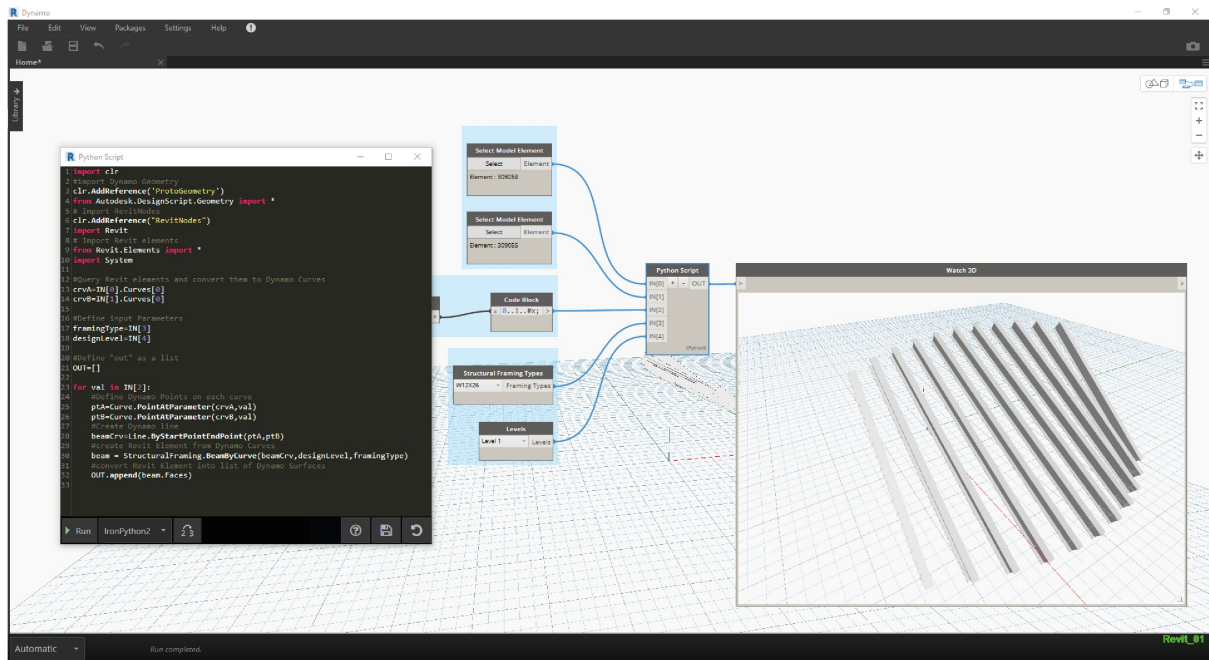


En Dynamo, también podemos ver los resultados. Las vigas del nodo Watch3D hacen referencia a la geometría consultada desde los elementos de Revit.

Observe que existe un proceso continuo de traslación de datos del entorno de Revit al entorno de Dynamo. En resumen, así es como se desarrolla el proceso:

1. Se selecciona el elemento de Revit.
2. Se convierte el elemento de Revit en curva de Dynamo.
3. Se divide la curva de Dynamo en una serie de puntos de Dynamo.
4. Se utilizan los puntos de Dynamo entre dos curvas para crear líneas de Dynamo.
5. Se crean vigas de Revit mediante referencias a líneas de Dynamo.
6. Se generan superficies de Dynamo mediante una consulta sobre la geometría de las vigas de Revit.

Esto puede parecer un poco pesado, pero la secuencia de comandos hace que sea tan sencillo como editar la curva en Revit y volver a ejecutar el solucionador (aunque es posible que deba suprimir las vigas anteriores al hacerlo). *Esto se debe al hecho de que hemos colocado vigas en Python, lo que rompe la asociación que tienen los nodos OOTB.*



Con una actualización de las curvas de referencia de Revit, obtenemos una nueva matriz de vigas.

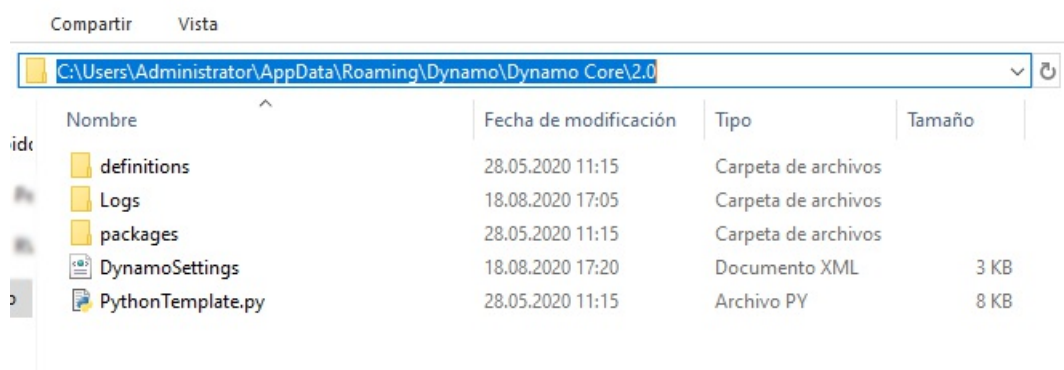
Plantillas de Python en Dynamo 2.0

Plantillas de Python

Con Dynamo 2.0, podemos especificar una plantilla por defecto (extensión .py) para utilizarla al abrir la ventana de Python por primera vez. Esta ha sido una petición deseada durante mucho tiempo, ya que agiliza el uso de Python en Dynamo. La capacidad de utilizar una plantilla nos permite disponer de importaciones por defecto listas para usar cuando deseamos desarrollar una secuencia de comandos de Python personalizada.

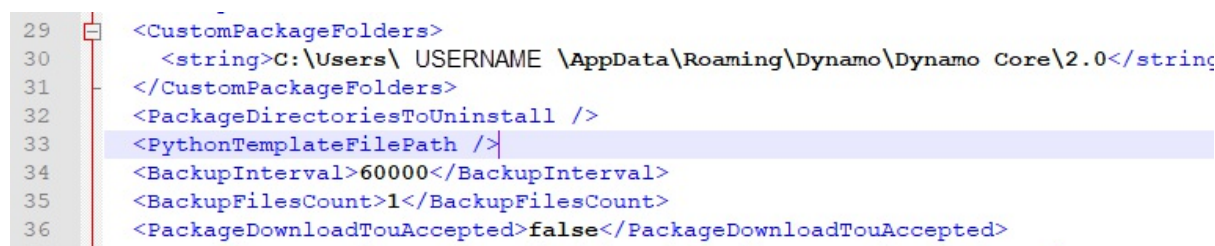
Esta plantilla se encuentra en la ubicación APPDATA de la instalación de Dynamo.

Por lo general, es la siguiente: (%appdata%\Dynamo\Dynamo Core\{versión}\).



Configuración de la plantilla

Para utilizar esta función, debemos añadir la siguiente línea en el archivo `DynamoSettings.xml`. (Éditelo en el Bloc de notas).



Cuando aparece `<PythonTemplateFilePath />`, solo tenemos que sustituir esto por lo siguiente:

```
<PythonTemplateFilePath>
<string>C:\Users\CURRENTUSER\AppData\Roaming\Dynamo\Dynamo Core\2.0\PythonTemplate.py</string>
</PythonTemplateFilePath>
```

Nota: sustituya `CURRENTUSER` por su nombre de usuario.

A continuación, debemos crear una plantilla con la funcionalidad integrada que deseamos utilizar. En este caso, insertaremos las importaciones relacionadas con Revit y algunos de los demás elementos habituales al trabajar con Revit.

Puede iniciar un documento del Bloc de notas en blanco y pegar el siguiente código:

```
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)
```

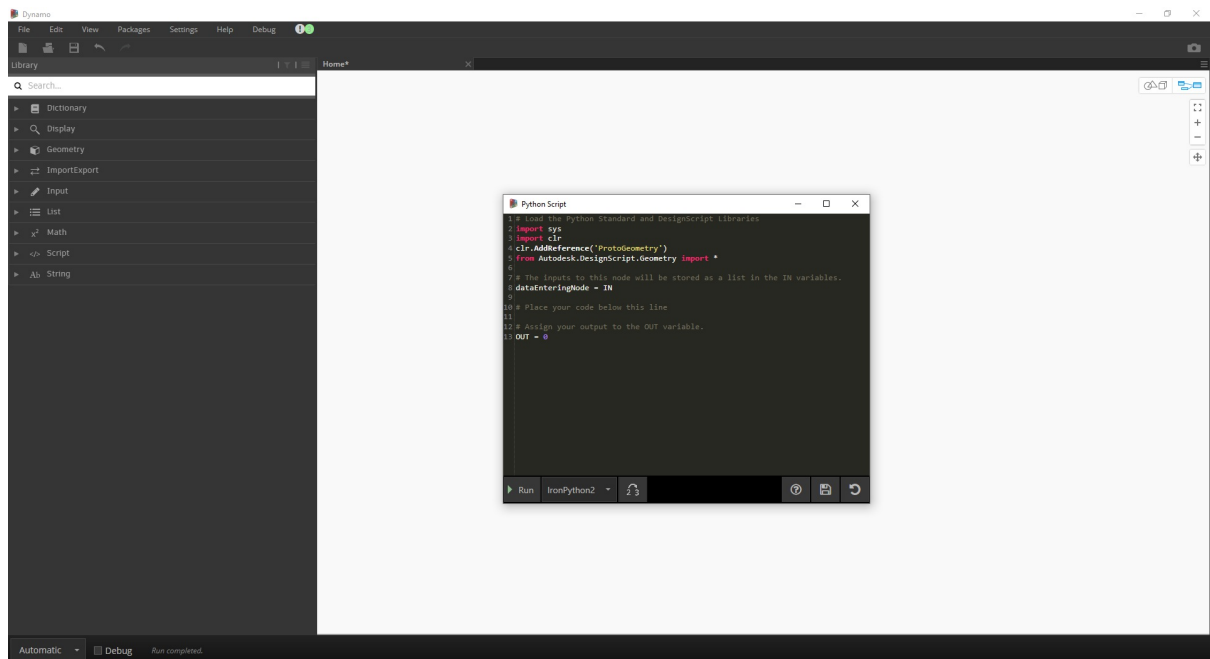
```
TransactionManager.Instance.TransactionTaskDone()
```

```
OUT = element
```

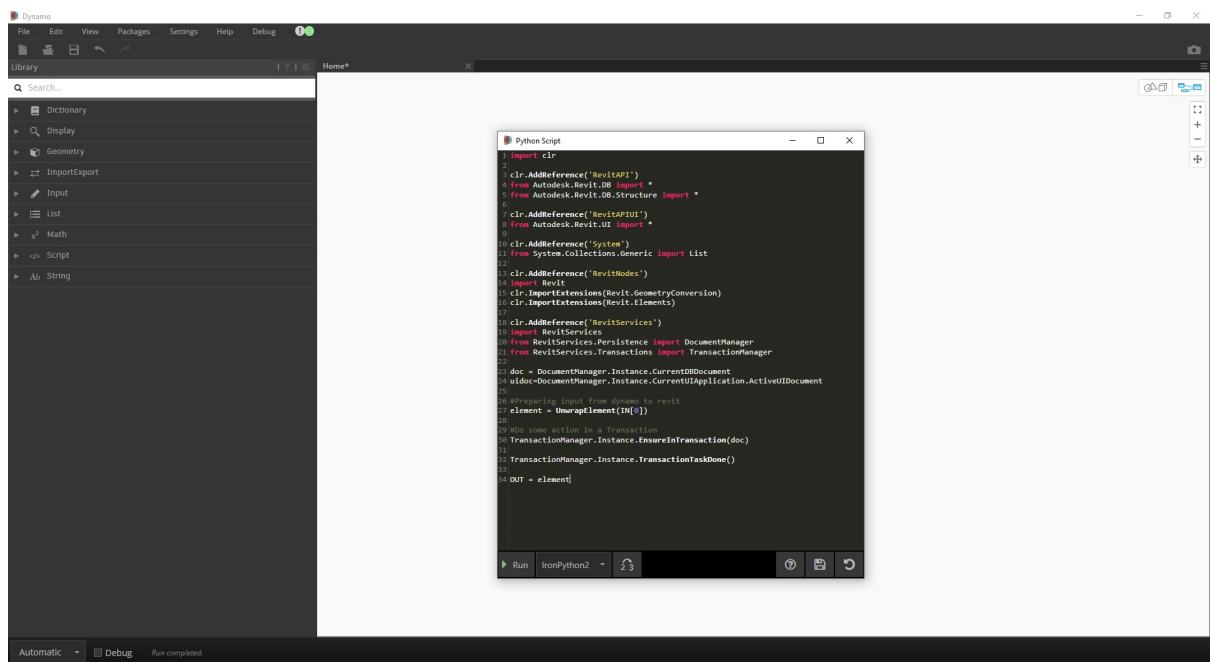
Una vez hecho esto, guarde este archivo como `PythonTemplate.py` en la ubicación `APPDATA`.

Comportamiento posterior de la secuencia de comandos de Python

Una vez que se haya definido la plantilla de Python, Dynamo la buscará cada vez que se coloque un nodo de Python. Si no se encuentra, se mostrará la ventana de Python por defecto.



Si se encuentra la plantilla de Python (como, por ejemplo, nuestra plantilla de Revit), aparecerán todos los elementos por defecto integrados.

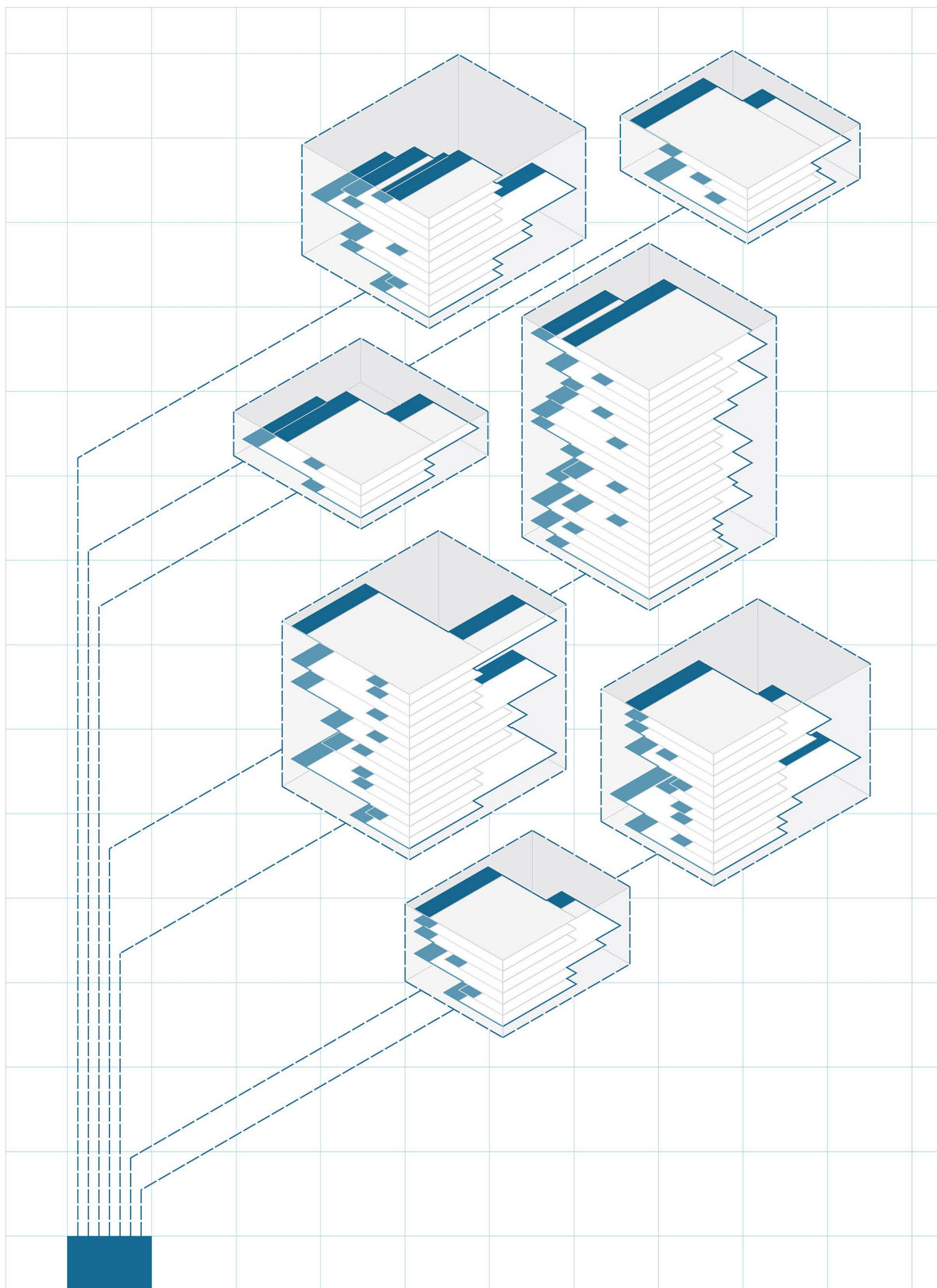


Aquí encontrará información adicional (de Radu Gidei) sobre esta excelente incorporación. <https://github.com/DynamoDS/Dynamo/pull/8122>

Paquetes

Paquetes

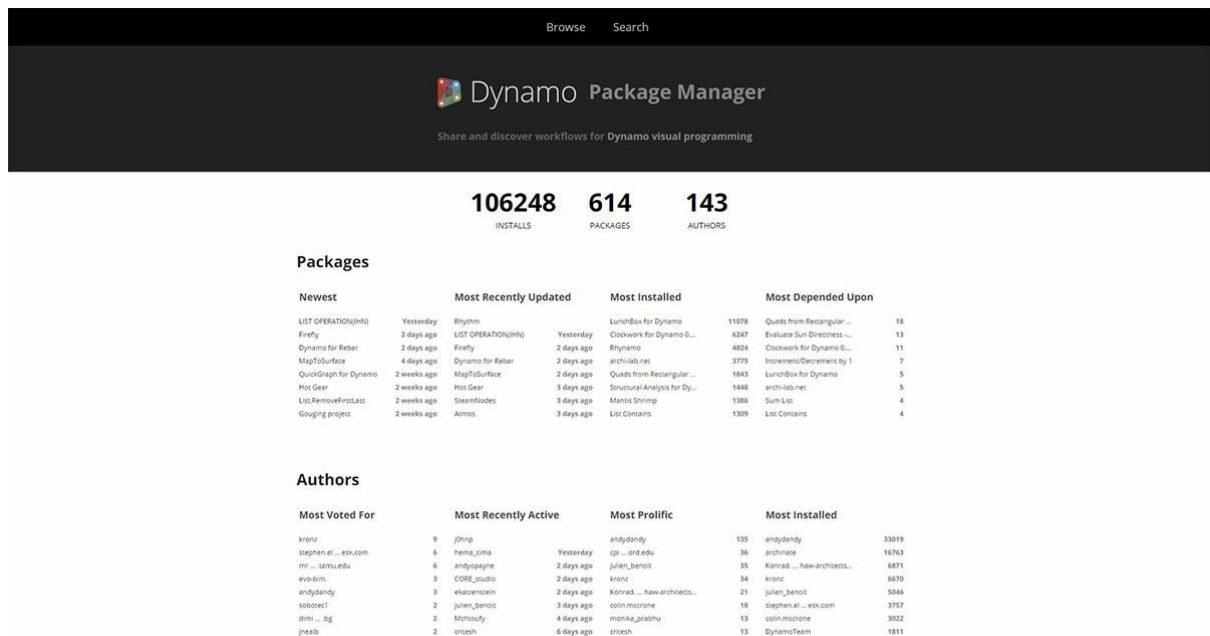
Una vez que haya creado algunos nodos personalizados, el siguiente paso es comenzar a organizarlos y publicarlos mediante paquetes, una forma cómoda de almacenar y compartir sus nodos con la comunidad de Dynamo.



Introducción a los paquetes

Paquetes

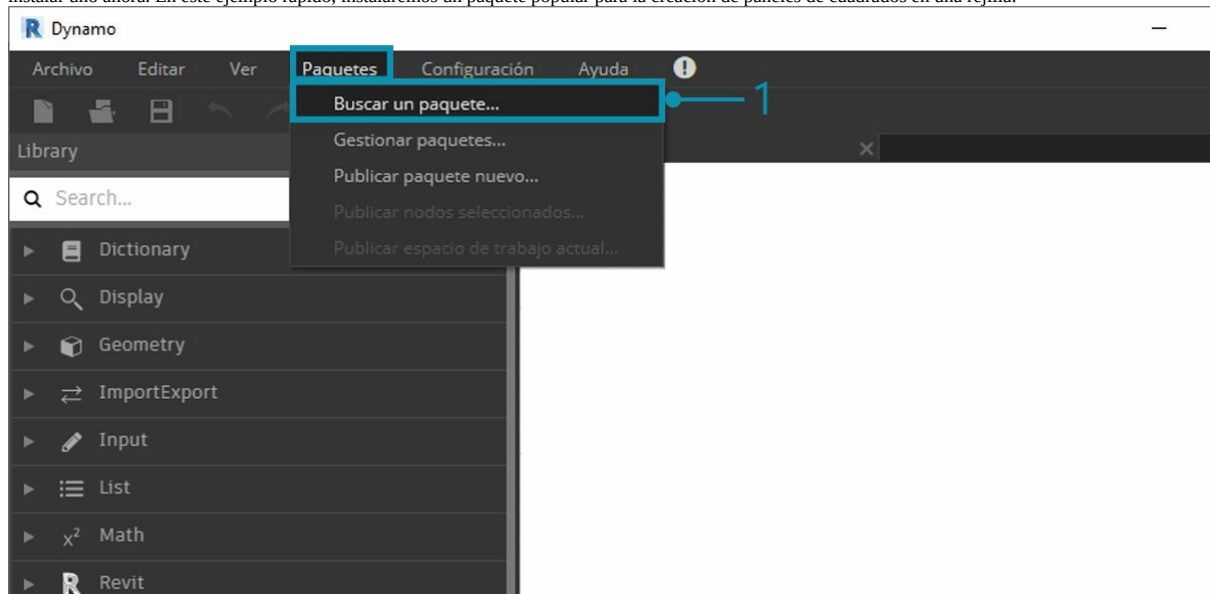
En pocas palabras, un paquete es un conjunto de nodos personalizados. El administrador de paquetes de Dynamo es un portal para que la comunidad descargue cualquier paquete que se haya publicado en línea. Estos conjuntos de herramientas los desarrollan terceros para ampliar la funcionalidad principal de Dynamo. Están disponibles para todos los usuarios y listos para descargar con solo hacer clic.



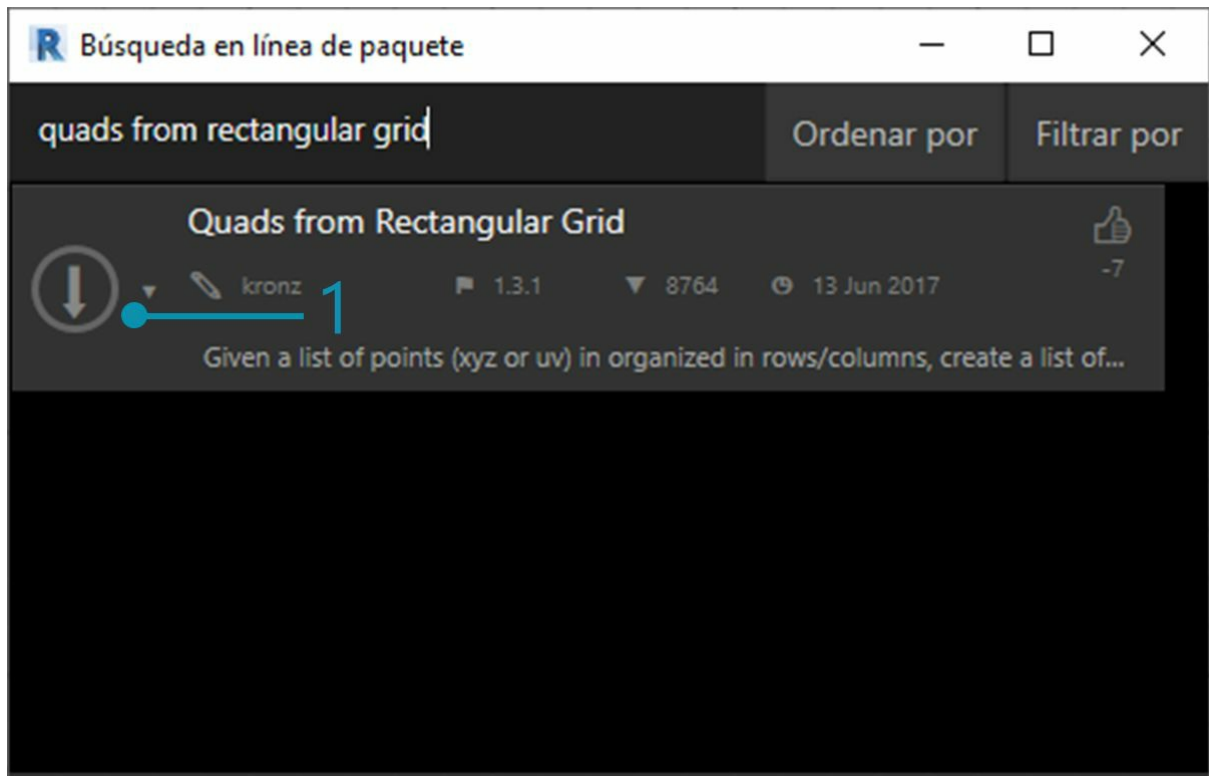
Un proyecto de código abierto como Dynamo crece con este tipo de participación de la comunidad. Con desarrolladores independientes dedicados, Dynamo puede ampliar su alcance a los flujos de trabajo de una amplia gama de sectores. Por este motivo, el equipo de Dynamo ha realizado esfuerzos coordinados para optimizar el desarrollo y la publicación de paquetes (lo cual se trata en mayor detalle en las siguientes secciones).

Instalación de un paquete

La forma más sencilla de instalar un paquete es mediante la barra de herramientas Paquetes de la interfaz de Dynamo. Vamos a ir directos a la acción e instalar uno ahora. En este ejemplo rápido, instalaremos un paquete popular para la creación de paneles de cuadrados en una rejilla.

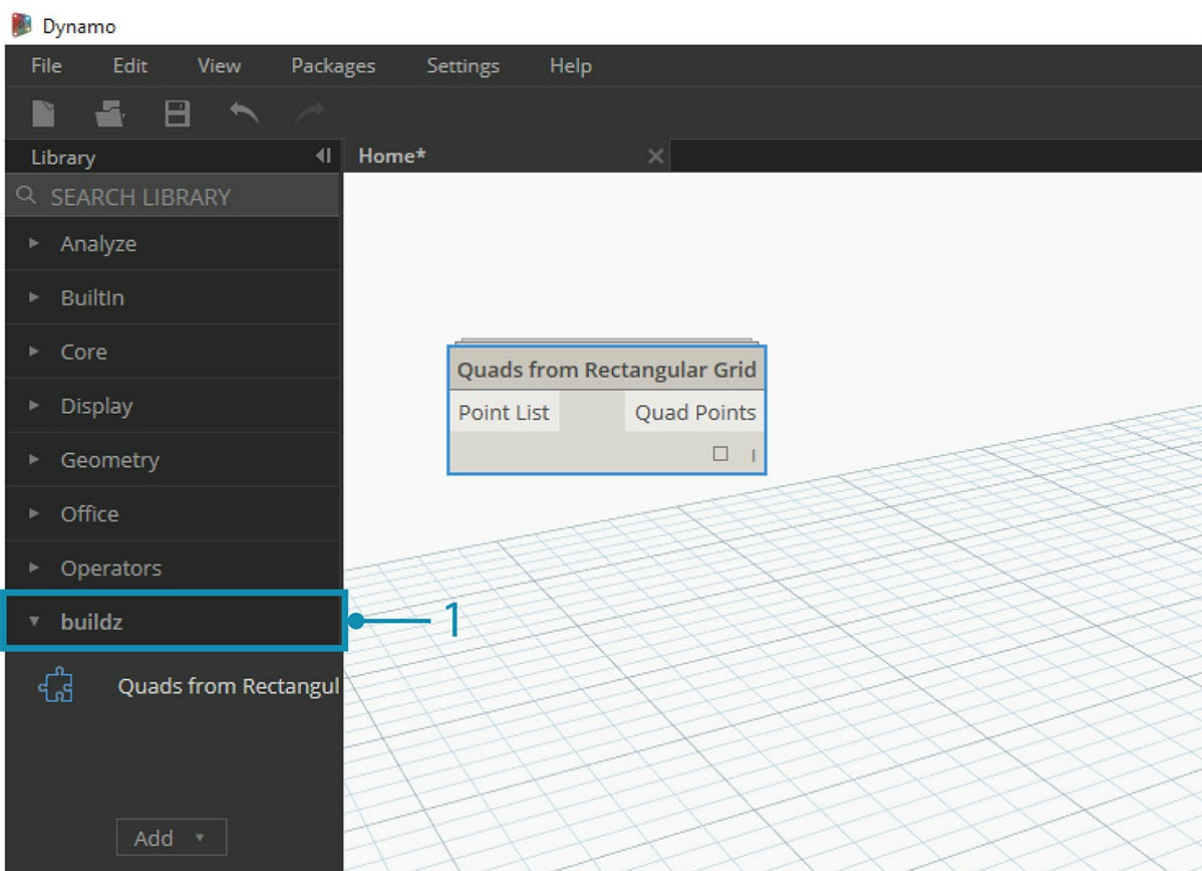


1. En Dynamo, vaya a *Paquetes* > *Buscar un paquete...*

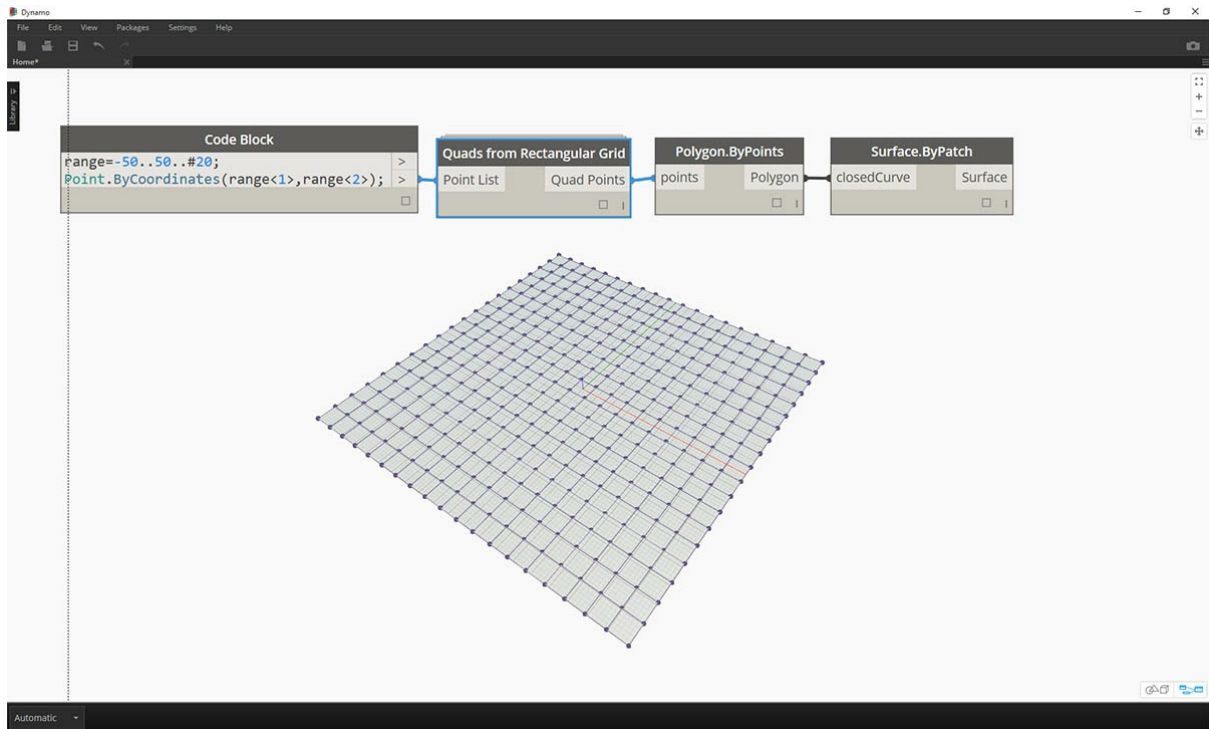


En la barra de búsqueda, buscamos "cuadrados de rejilla rectangular". Tras unos segundos, deberían aparecer todos los paquetes que coincidan con esta consulta de búsqueda. Vamos a seleccionar el primer paquete cuyo nombre coincida.

1. Haga clic en la flecha de descarga situada a la izquierda del nombre del paquete y el paquete se instalará. Listo.



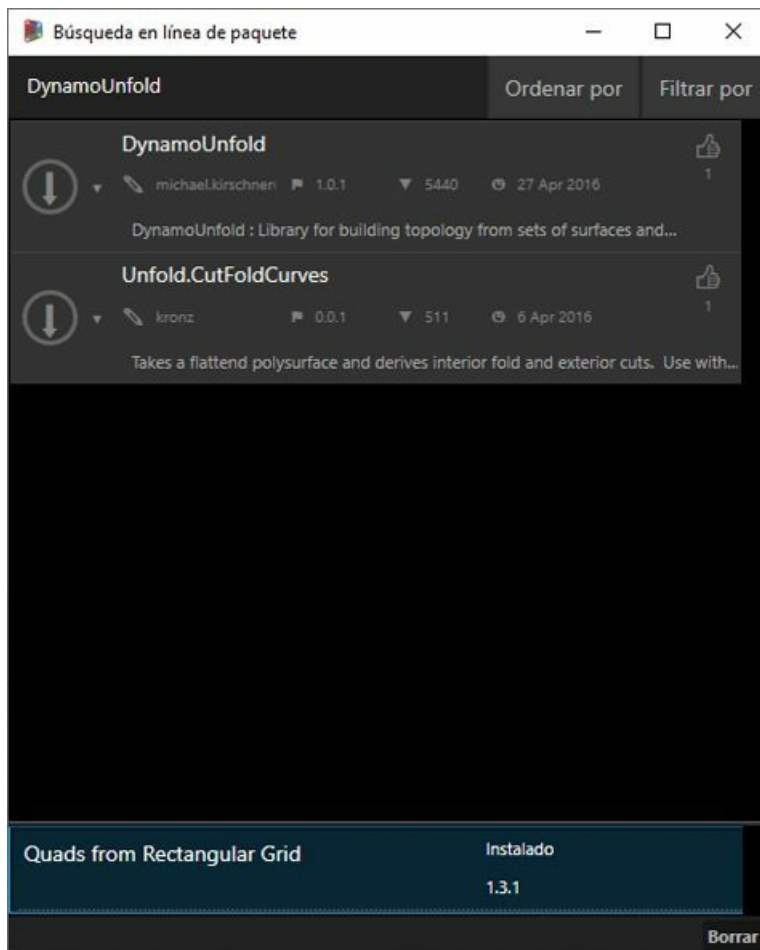
1. Observe que ahora tenemos otro grupo en la biblioteca de Dynamo denominado "buildz". Este nombre hace referencia al [desarrollador](#) del paquete y el nodo personalizado se encuentra en este grupo. Podemos empezar a utilizarlo al instante.



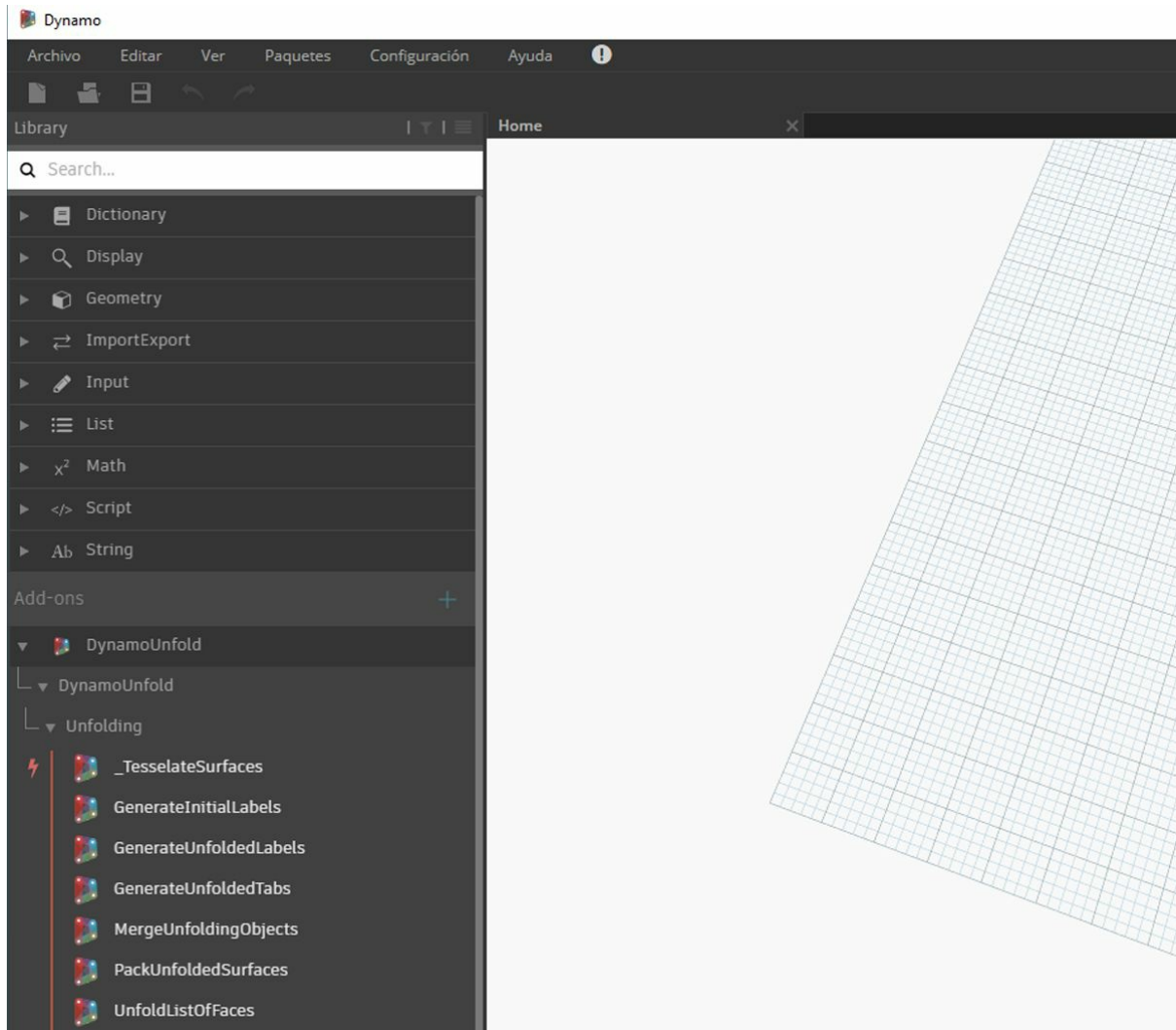
Con una operación de bloque de código rápida para definir una rejilla rectangular, hemos creado una lista de paneles rectangulares.

Carpetas de paquete

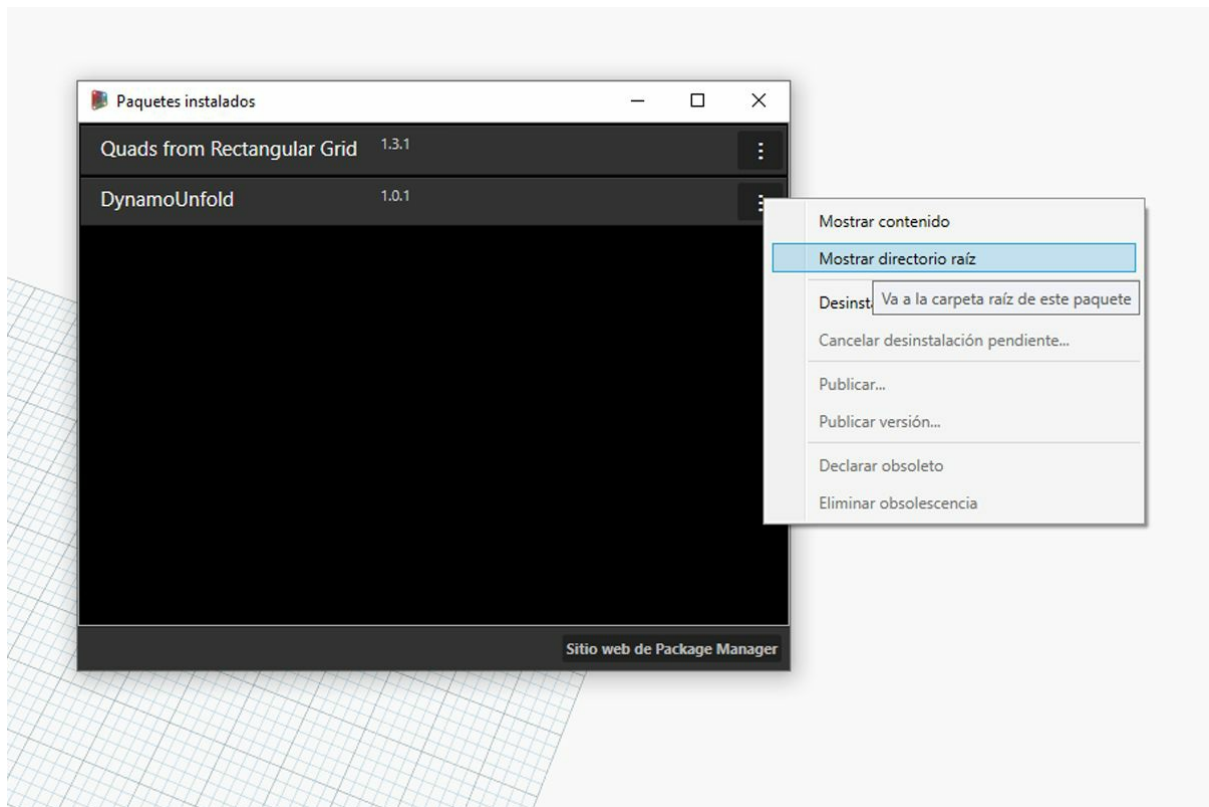
El ejemplo anterior se centra en un paquete con un nodo personalizado, pero se utiliza el mismo proceso para descargar paquetes con varios nodos personalizados y archivos de datos complementarios. Vamos a demostrar esto a continuación con un paquete más completo: Dynamo Unfold.



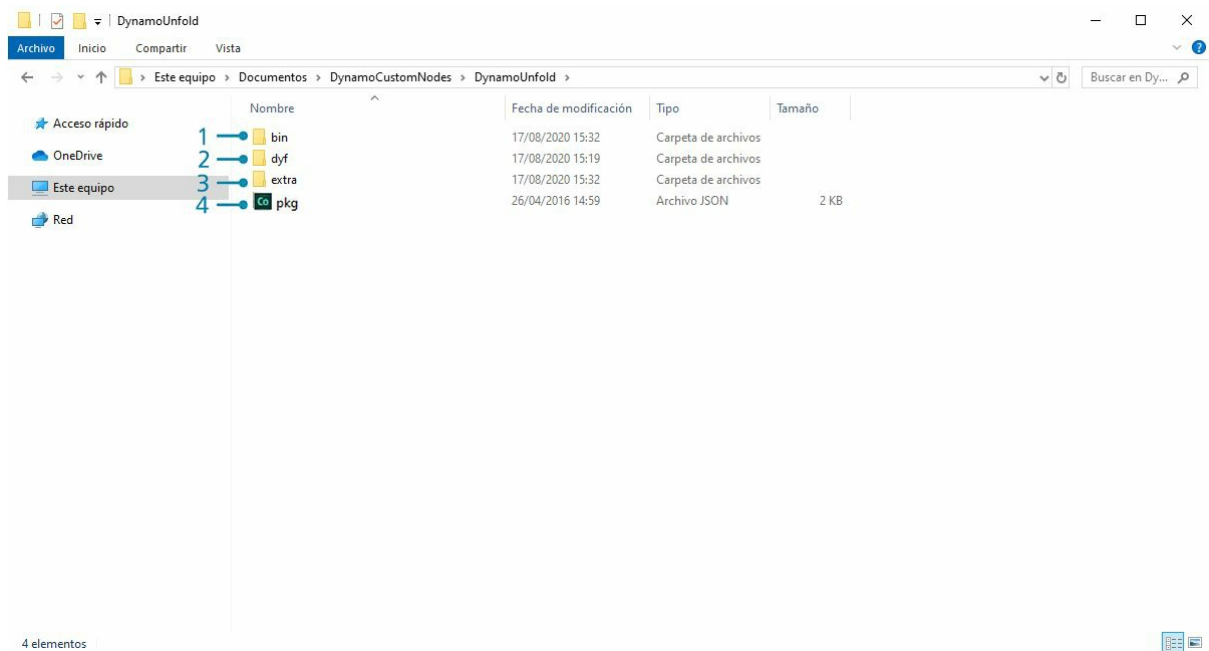
Como en el ejemplo anterior, seleccione *Paquetes > Buscar un paquete...* Esta vez, buscaremos "*DynamoUnfold*", en una sola palabra y respetando las mayúsculas y minúsculas. Cuando aparezcan los paquetes, descárguelo haciendo clic en la flecha situada a la izquierda del nombre del paquete. Dynamo Unfold se instalará en la biblioteca de Dynamo.



En la biblioteca de Dynamo, tenemos un grupo de *DynamoUnfold* con varias categorías y nodos personalizados.

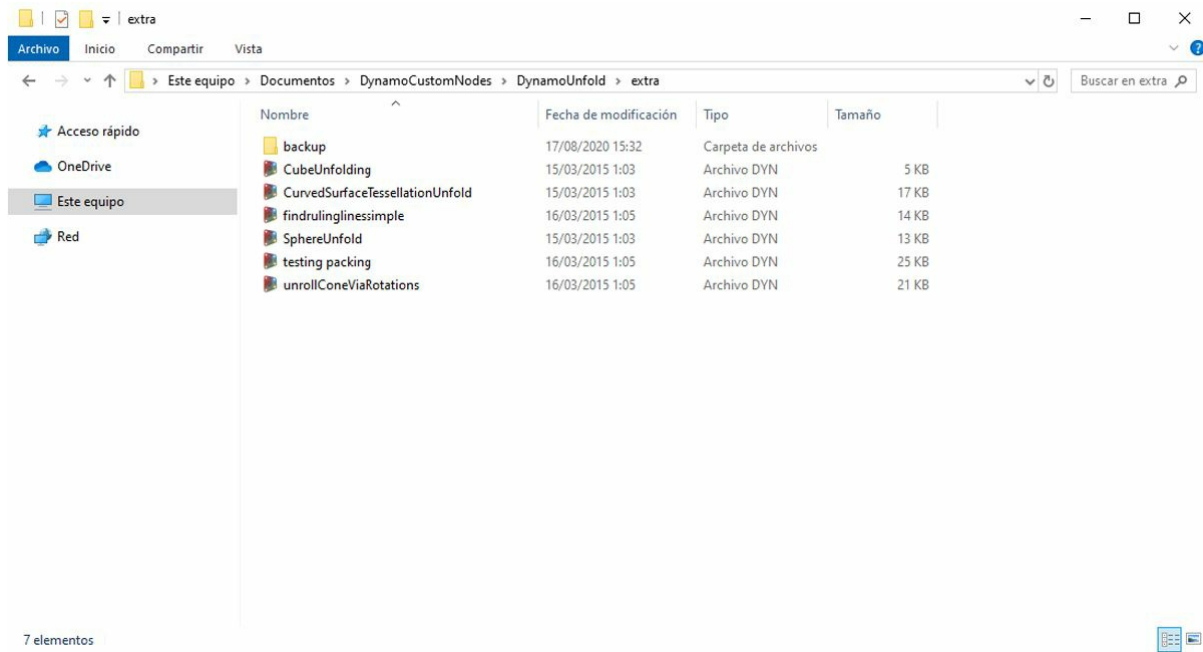


Ahora, veamos la estructura de archivos del paquete. Seleccione "*Paquetes > Gestionar paquetes...*" en Dynamo. Veremos la ventana anterior con las dos bibliotecas que hemos instalado. Haga clic en el botón situado a la derecha de *DynamoUnfold* y seleccione "*Mostrar directorio raíz*".

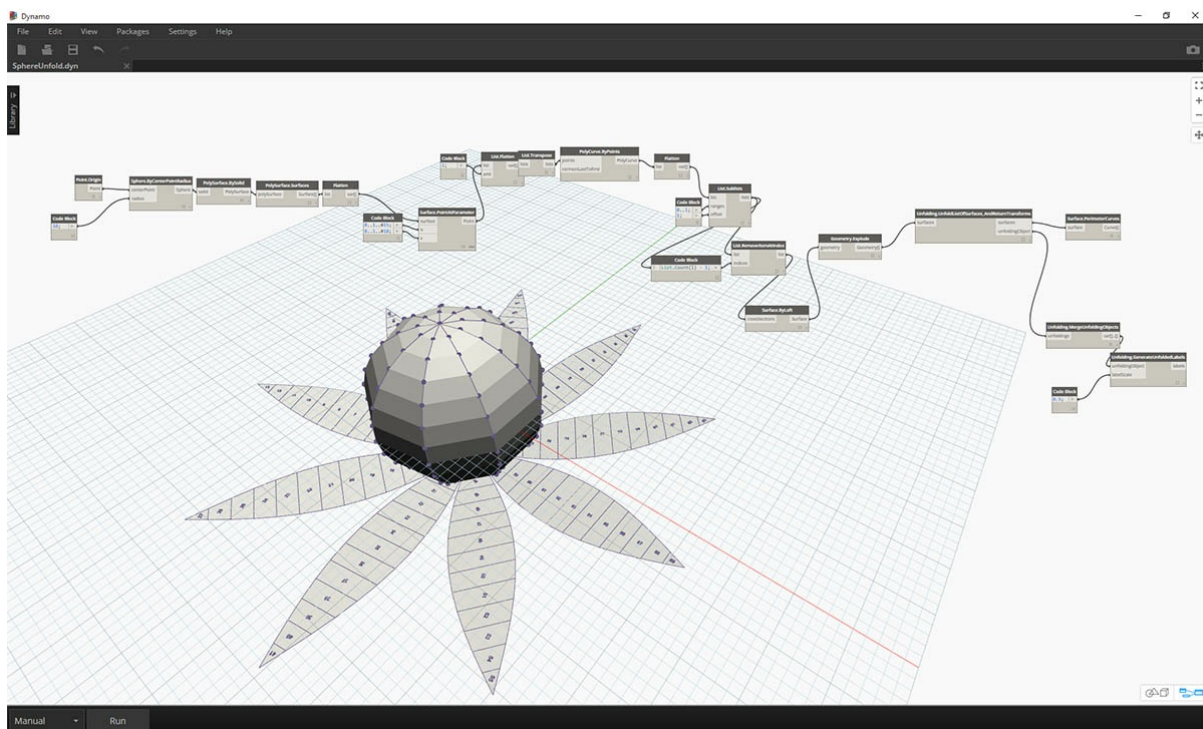


Esto nos llevará al directorio raíz del paquete. Observe que hay tres carpetas y un archivo.

1. La carpeta *bin* contiene archivos .dll. Este paquete de Dynamo se desarrolló mediante el uso de Zero-Touch, por lo que los nodos personalizados se guardan en esta carpeta.
2. La carpeta *dyf* contiene los nodos personalizados. Este paquete no se desarrolló mediante nodos personalizados de Dynamo, por lo que esta carpeta está vacía para este paquete.
3. La carpeta *extra* contiene todos los archivos adicionales, incluidos los archivos de ejemplo.
4. El archivo *pkg* es un archivo de texto básico que define los parámetros del paquete. Podemos pasarlo por alto por ahora.



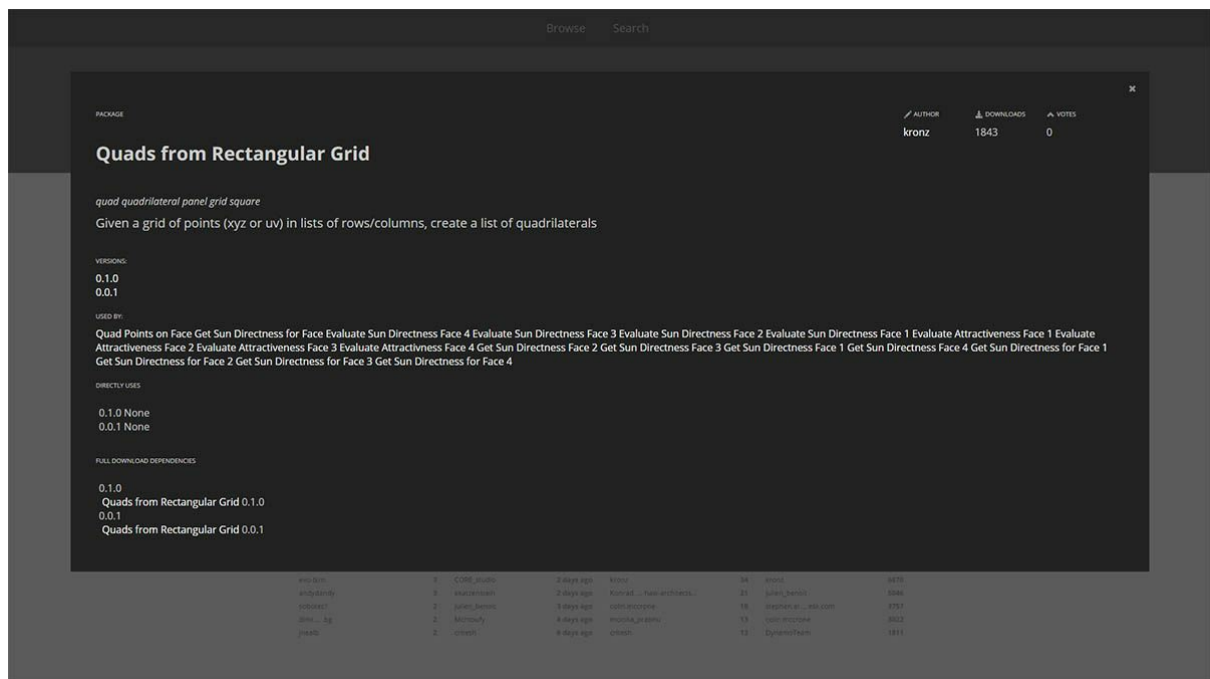
Al abrir la carpeta "extra", vemos un conjunto de archivos de ejemplo que se han descargado con la instalación. No todos los paquetes tienen archivos de ejemplo, pero aquí es donde se pueden encontrar si forman parte de un paquete. Abriremos "SphereUnfold".



Después de abrir el archivo y pulsar "Ejecutar" en el solucionador, tenemos una esfera desplegada. Estos archivos de ejemplo son útiles para aprender a trabajar con un nuevo paquete de Dynamo.

Administrador de paquetes de Dynamo

Otra forma de descubrir los paquetes de Dynamo es explorar el [Administrador de paquetes de Dynamo](#) en línea. Es una buena forma de buscar paquetes, ya que el repositorio ordena los paquetes por número de descargas y popularidad. Asimismo, es una forma sencilla de recopilar información sobre las actualizaciones recientes para los paquetes, ya que algunos paquetes de Dynamo están sujetos a las versiones y dependencias de Dynamo.



Al hacer clic en "Quads from Rectangular Grid" en el administrador de paquetes de Dynamo, aparecen las descripciones, las versiones, el desarrollador y las posibles dependencias.

También puede descargar los archivos de paquete desde el administrador de paquetes de Dynamo, pero este proceso es más directo si se realiza desde Dynamo.

¿Dónde se almacenan los archivos localmente?

Si descarga archivos desde el administrador de paquetes de Dynamo o si desea ver dónde se guardan todos los archivos del paquete, haga clic en *Configuración > Administrar rutas de nodos y paquetes...* Al hacer clic en los puntos suspensivos situados junto al directorio de carpetas, puede copiar la carpeta raíz y profundizar en el paquete en la ventana del explorador. Por defecto, los paquetes se instalan en una ubicación similar a esta ruta de carpeta: *C:/Usuarios/[nombredeusuario]/AppData/Roaming/Dynamo/[Versión de Dynamo]*.

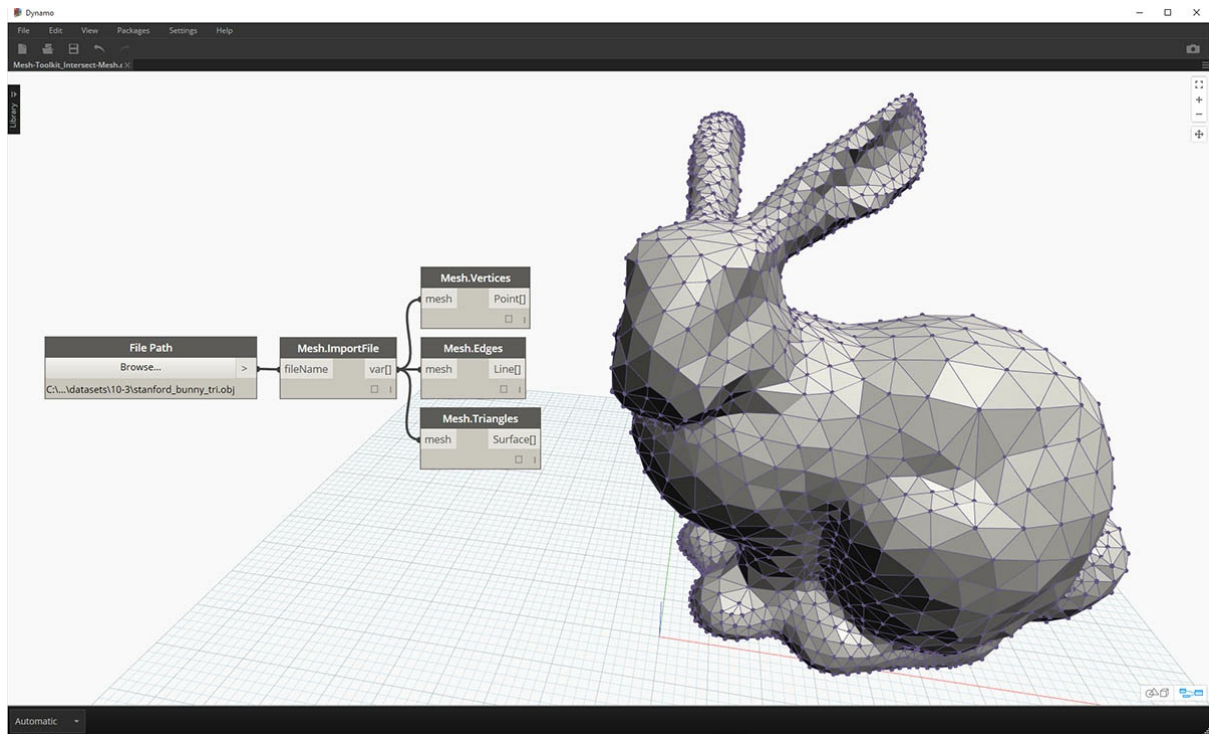
Más detalles sobre los paquetes

La comunidad de Dynamo está en constante crecimiento y evolución. Si explora el administrador de paquetes de Dynamo de vez en cuando, descubrirá algunos avances excelentes. En las secciones siguientes, analizaremos en profundidad los paquetes, desde la perspectiva del usuario final hasta la autoría de un paquete de Dynamo propio.

Caso real de paquete: Kit de herramientas de malla

Caso real de paquete: Kit de herramientas de malla

El Kit de herramientas de malla de Dynamo proporciona herramientas para importar mallas desde formatos de archivo externos, crear una malla a partir de objetos de geometría de Dynamo y generar manualmente mallas mediante sus vértices e índices. La biblioteca también proporciona herramientas para modificar y reparar mallas, o extraer cortes horizontales para su uso en la fabricación.

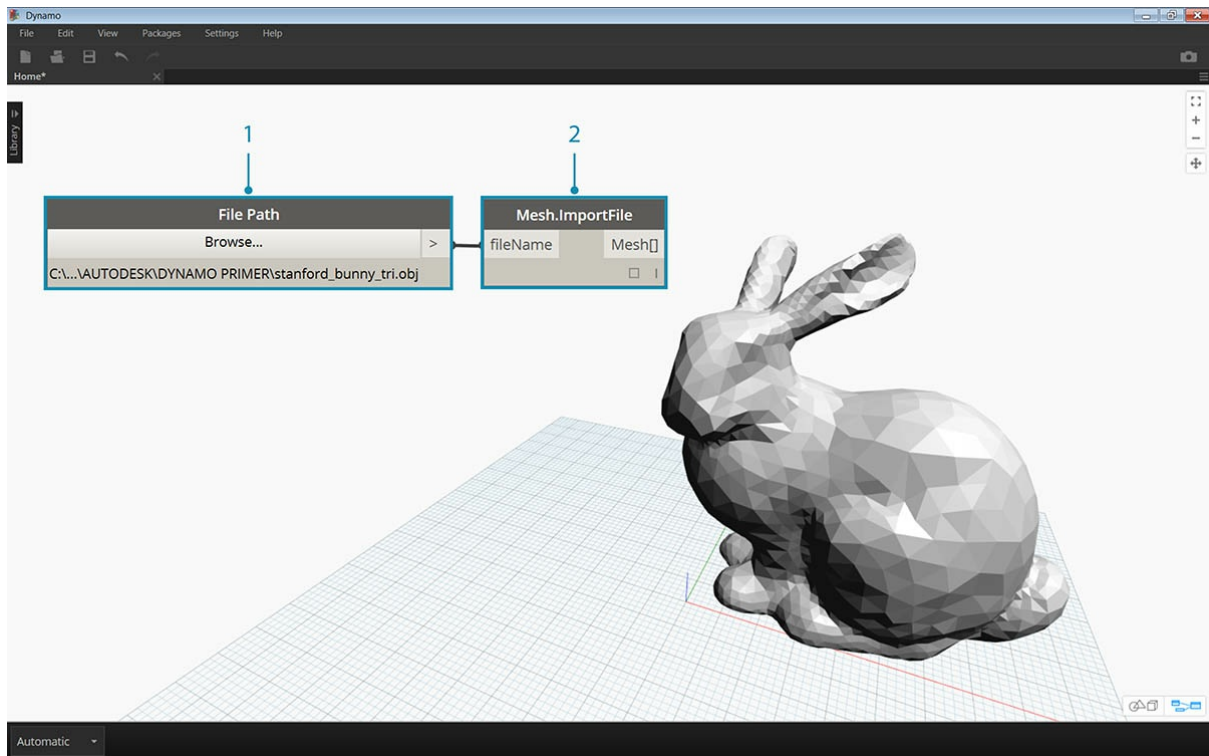


El Kit de herramientas de malla de Dynamo forma parte de la investigación de mallas en curso de Autodesk y, como tal, seguirá creciendo a lo largo de los próximos años. Esperamos que aparezcan con frecuencia nuevos métodos en este kit y no dude en ponerse en contacto con el equipo de Dynamo para ofrecer comentarios, indicar errores y enviar sugerencias sobre las nuevas funciones.

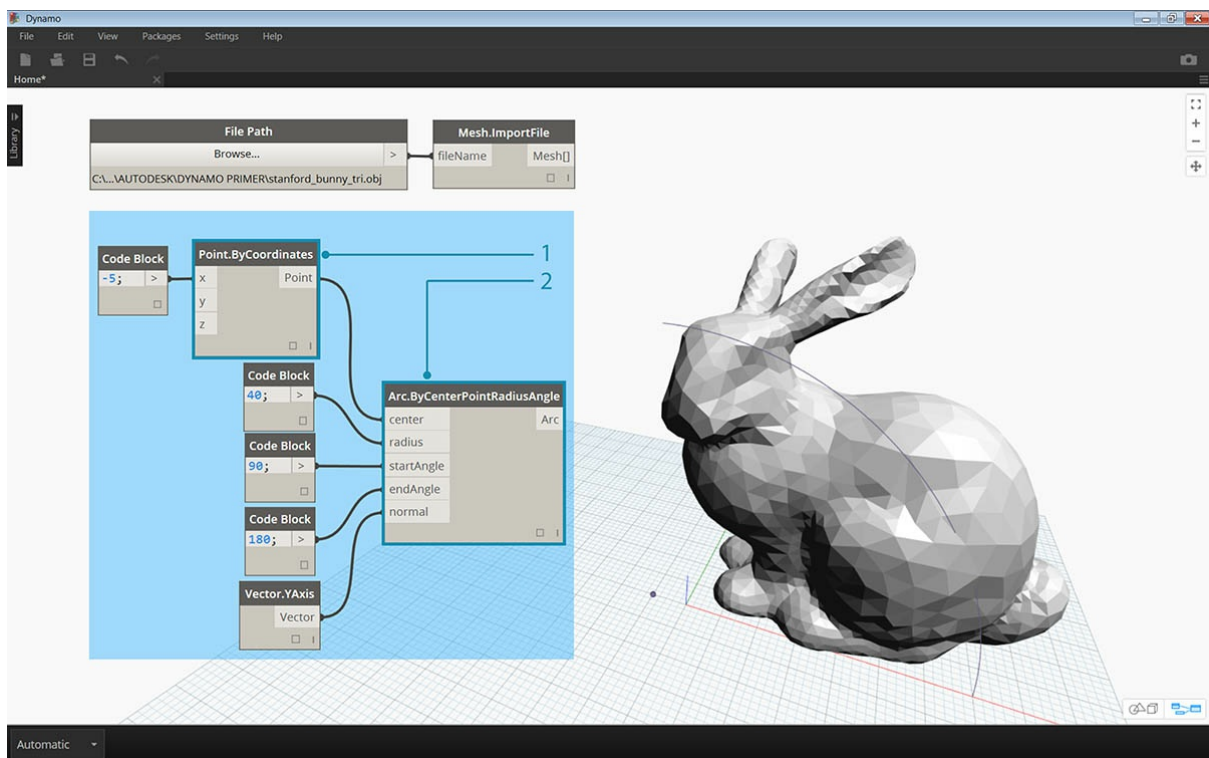
Mallas frente a sólidos

En el siguiente ejercicio, se muestran algunas operaciones básicas de malla que utiliza el Kit de herramientas de malla. En el ejercicio, intersecamos una malla con una serie de planos, lo que puede ser muy costoso desde una perspectiva computacional si se utilizan sólidos. A diferencia de los sólidos, las mallas tienen una "resolución" establecida y no se definen matemáticamente, sino topológicamente, y podemos definir esta resolución en función de la tarea que se está llevando a cabo. Para obtener más información sobre las relaciones entre mallas y sólidos, puede consultar el capítulo [Geometría para el diseño computacional](#) de este manual de introducción. Para examinar de forma más exhaustiva el Kit de herramientas de malla, consulte la [página wiki de Dynamo](#). Pasemos al paquete en el ejercicio siguiente.

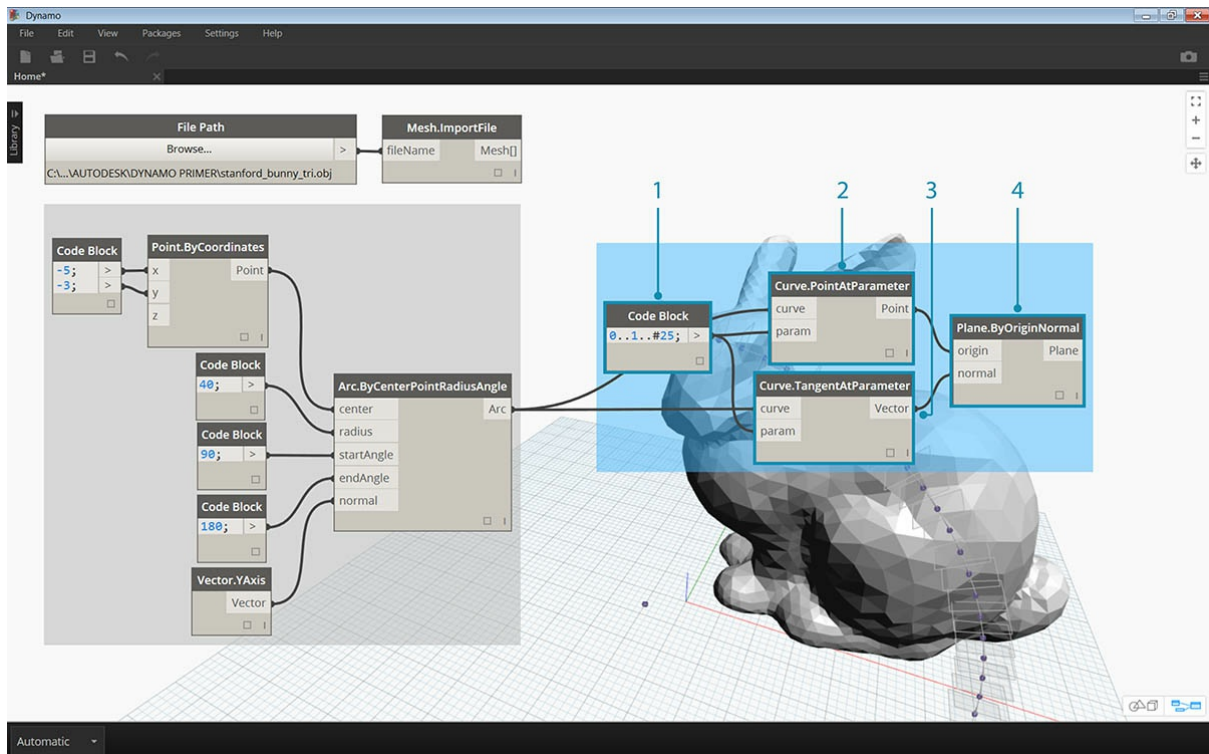
Instalar el Kit de herramientas de malla



1. **Ruta de archivo:** busque el archivo de malla que desee importar (*stanford_bunny_tri.obj*). Los tipos de archivo admitidos son .mix y .obj.
2. **Mesh.ImportFile:** conecte la ruta de archivo para importar la malla.

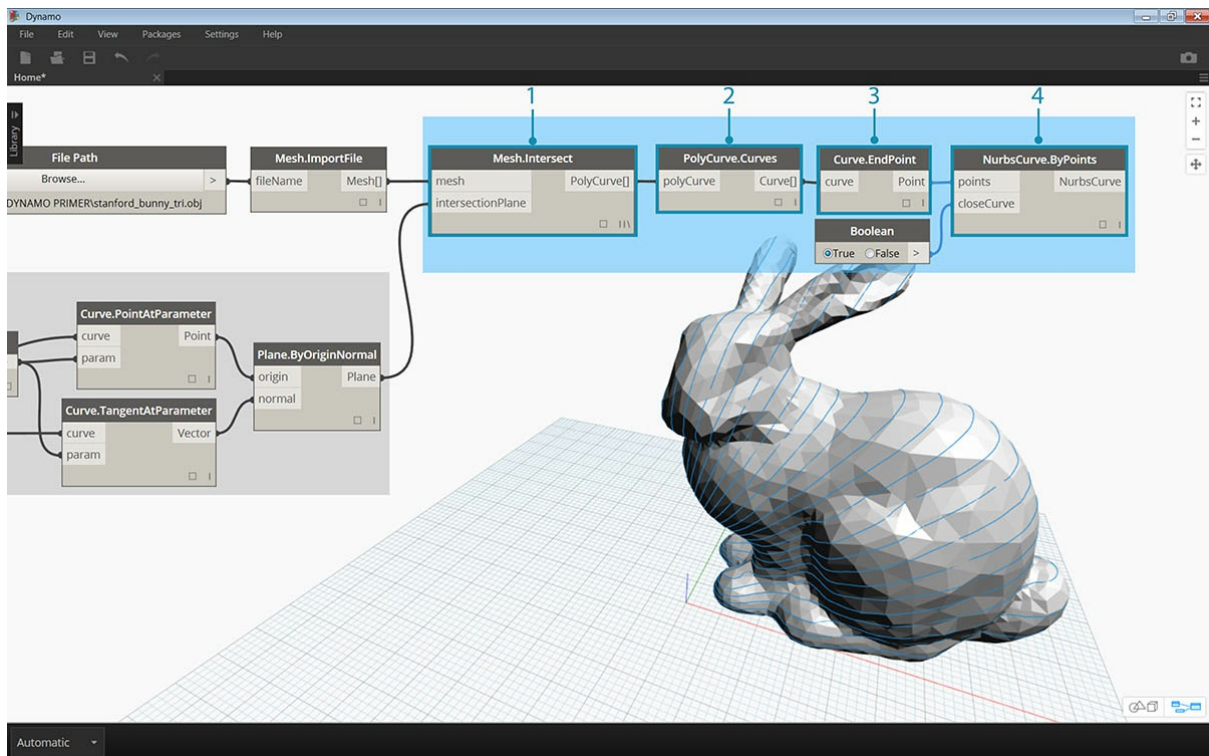


1. **Point.ByCoordinates:** cree un punto; este será el centro de un arco.
2. **Arc.ByCenterPointRadiusAngle:** cree un arco alrededor del punto. Esta curva se utilizará para colocar una serie de planos.

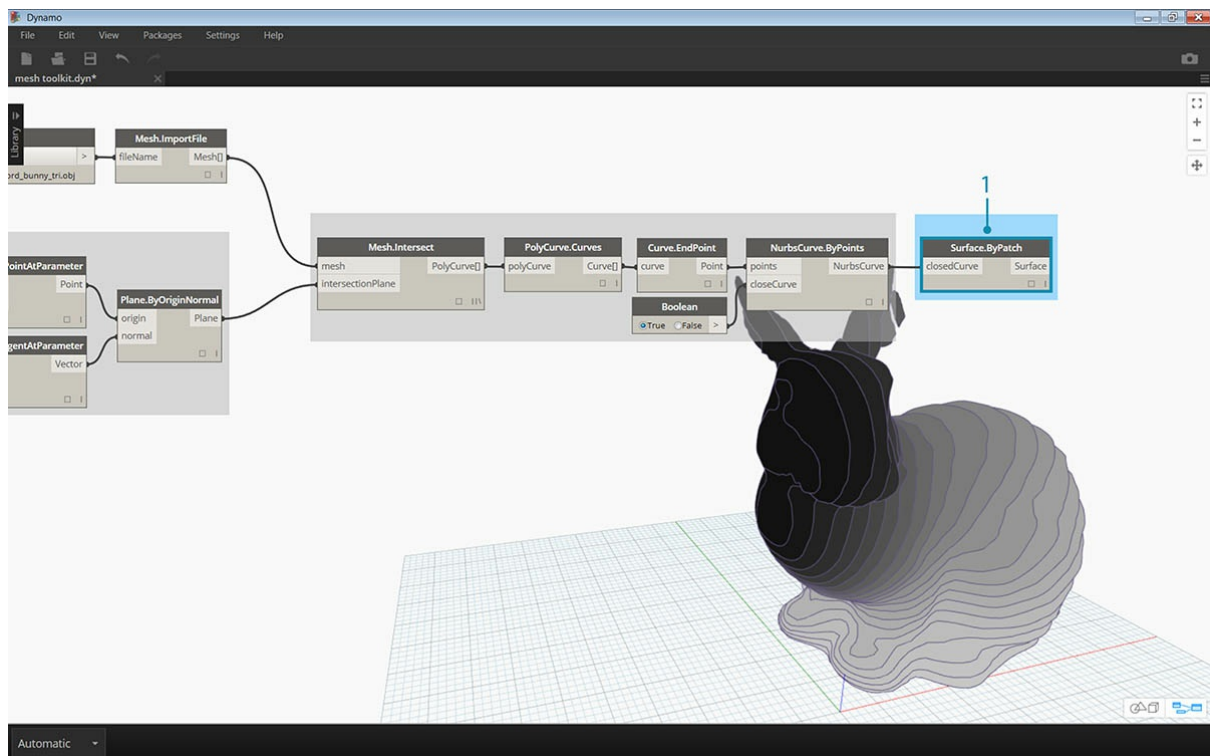


1. Code Block: cree un intervalo de números entre cero y uno.
2. **Curve.PointAtParameter**: conecte el arco a la entrada *curve* y la salida de bloque de código a la entrada *param* para extraer una serie de puntos a lo largo de la curva.
3. **Curve.TangentAtParameter**: conecte las mismas entradas que en el nodo anterior.
4. **Plane.ByOriginNormal**: conecte los puntos a la entrada *origin* y los vectores a la entrada *normal* para crear una serie de planos en cada punto.

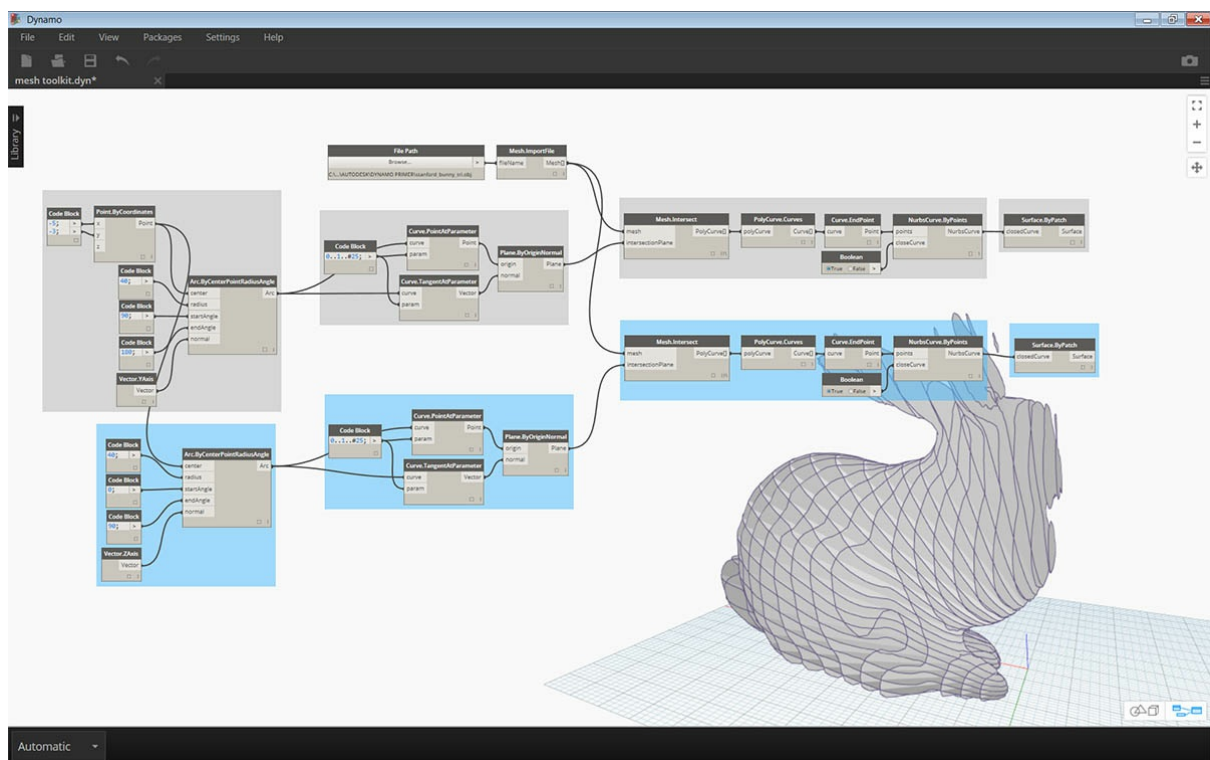
Ahora debería ver una serie de planos orientados a lo largo del arco. A continuación, utilizaremos estos planos para intersecar la malla.



1. **Mesh.Intersect**: interseque los planos con la malla importada, creando una serie de contornos de PolyCurve.
2. **PolyCurve.Curves**: divida las PolyCurves en fragmentos de curva.
3. **Curve.EndPoint**: extraiga los puntos finales de cada curva.
4. **NurbsCurve.ByPoints**: utilice los puntos para crear una NurbsCurve. Utilice un nodo booleano establecido en *True* (verdadero) para cerrar las curvas.



1. **Surface.ByPatch**: cree parches de superficie para cada contorno con el fin de crear "cortes" de la malla.



Añada un segundo conjunto de cortes para obtener un efecto de gofre/cartón de huevos.

Es posible que se haya dado cuenta de que las operaciones de intersección se calculan de forma más rápida con una malla que con un sólido comparable. Los flujos de trabajo como el que se muestra en este ejercicio son excelentes para trabajar con mallas.

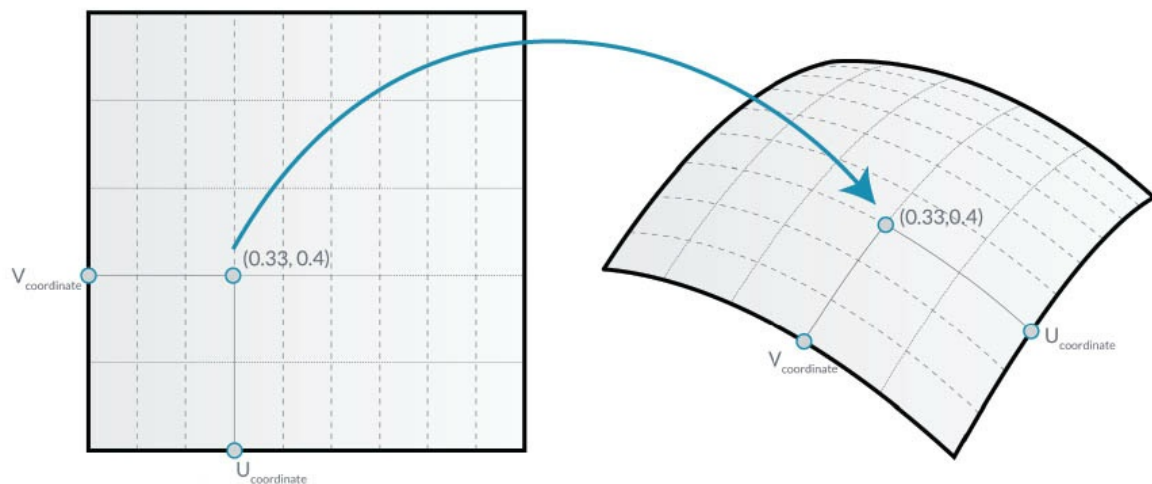
Desarrollo de un paquete

Desarrollo de un paquete

Dynamo ofrece una gran variedad de métodos para crear un paquete para uso personal o para compartir con la comunidad de Dynamo. En el caso real incluido a continuación, veremos cómo se configura un paquete mediante la desestructuración de uno existente. Este caso real se basa en las lecciones del capítulo anterior y proporciona un conjunto de nodos personalizados para la asignación de geometría, mediante coordenadas UV, de una superficie de Dynamo a otra.

MapToSurface

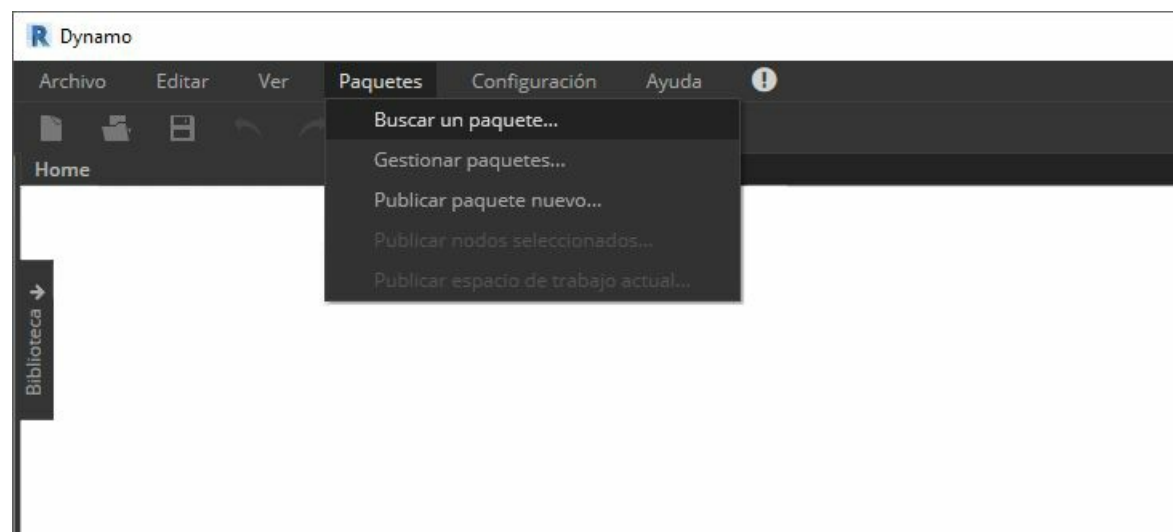
Vamos a trabajar con un paquete de ejemplo que demuestra la asignación de UV de puntos de una superficie a otra. Ya hemos aprendido los conceptos básicos de la herramienta en la sección [Creación de un nodo personalizado](#) de este manual de introducción. Los archivos siguientes muestran cómo podemos utilizar el concepto de asignación de UV y desarrollar un conjunto de herramientas para una biblioteca publicable.



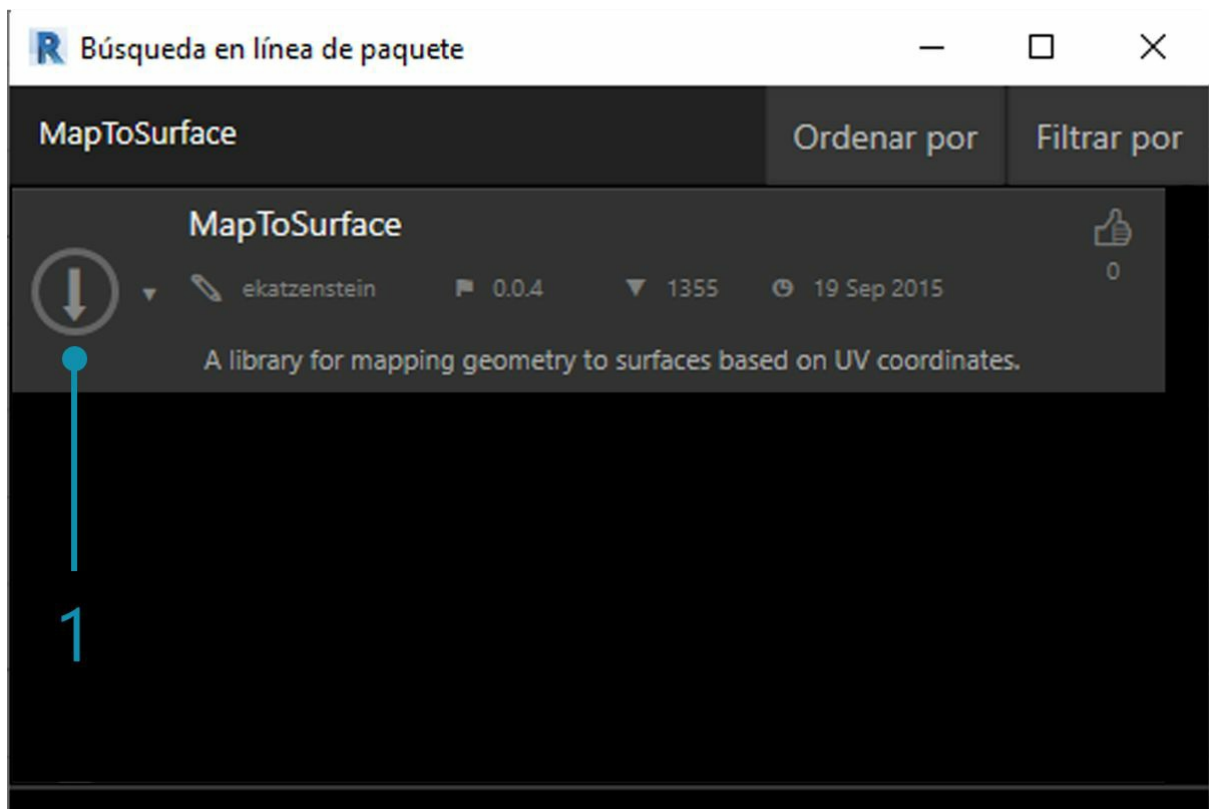
En esta imagen, se asigna un punto de una superficie a otra mediante coordenadas UV. El paquete se basa en este concepto, pero con una geometría más compleja.

Instalación del paquete

En el capítulo anterior, exploramos métodos para panelizar una superficie en Dynamo según las curvas definidas en el plano XY. Este caso real amplía estos conceptos a más dimensiones de la geometría. Vamos a instalar este paquete como se creó para mostrar cómo se ha desarrollado. En la siguiente sección, mostraremos cómo se ha publicado este paquete.

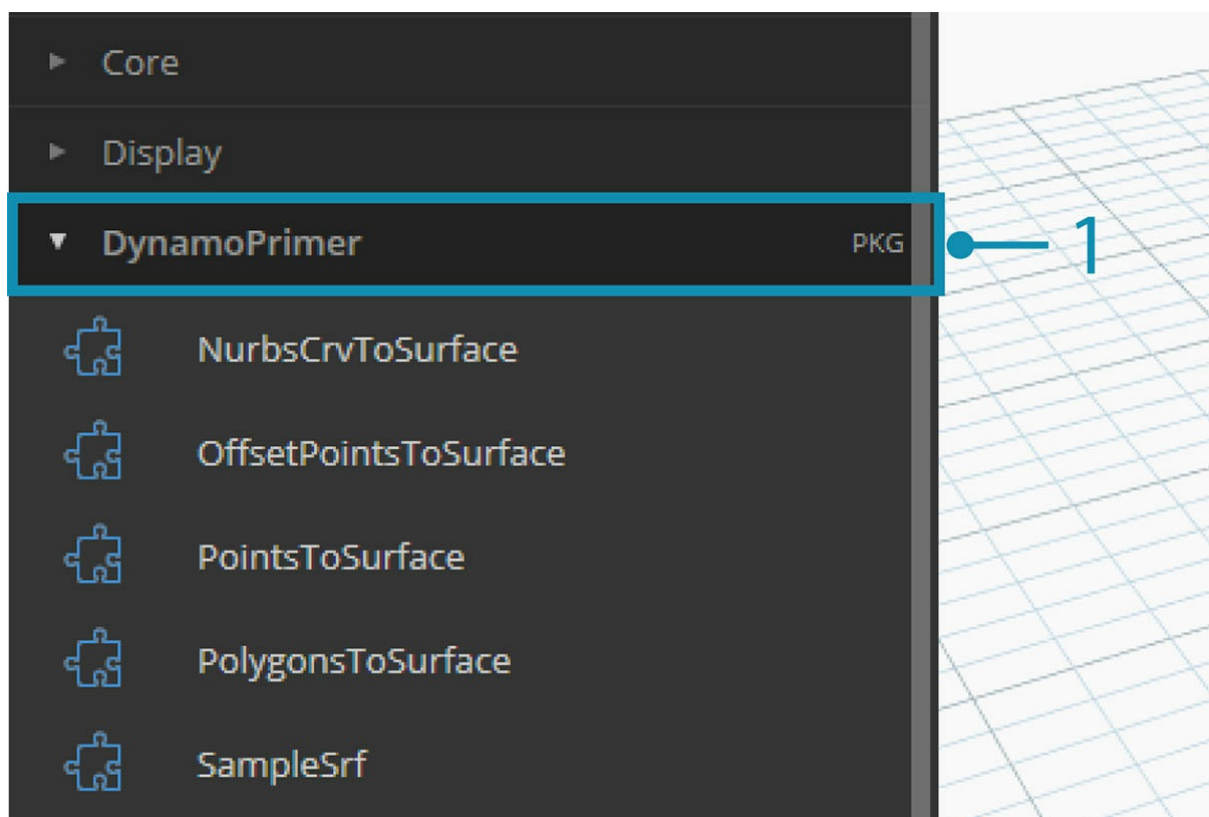


Esta es la parte fácil. En Dynamo, vaya a "Paquetes > Buscar un paquete..."



Busque el paquete "MapToSurface" (en una sola palabra).

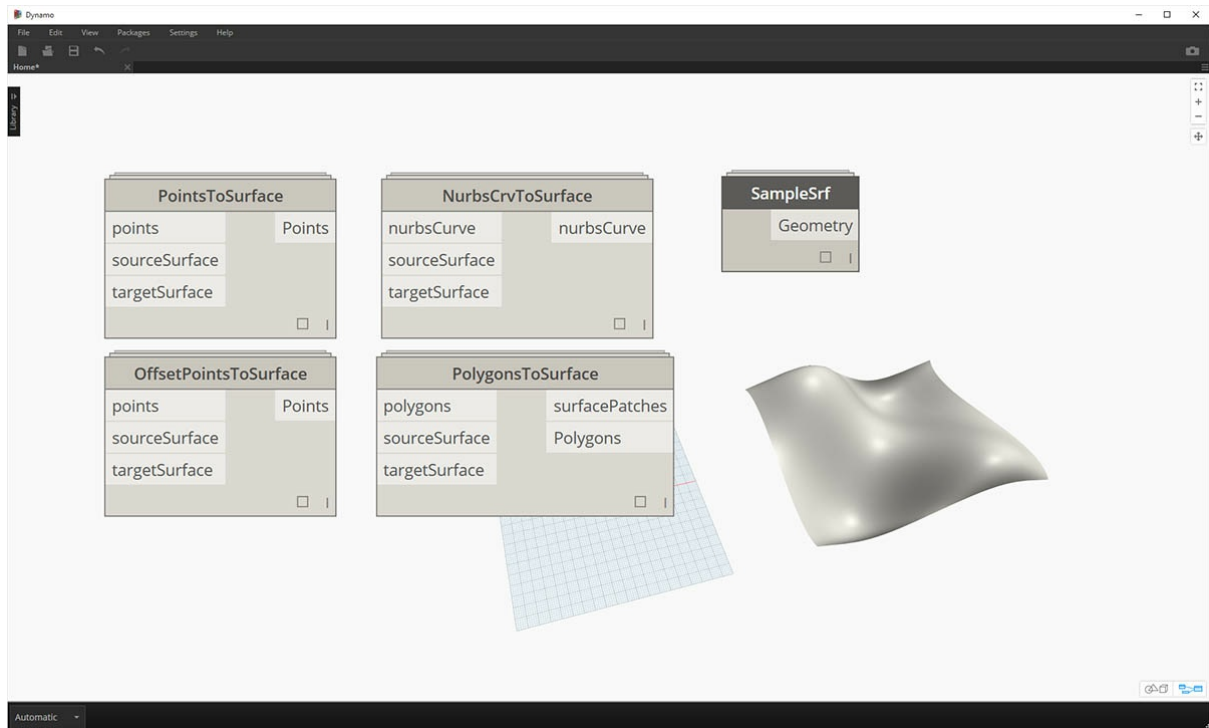
1. Cuando encuentre el paquete, haga clic en la flecha de descarga grande situada a la izquierda del nombre del paquete. De este modo, se instalará el paquete en Dynamo.



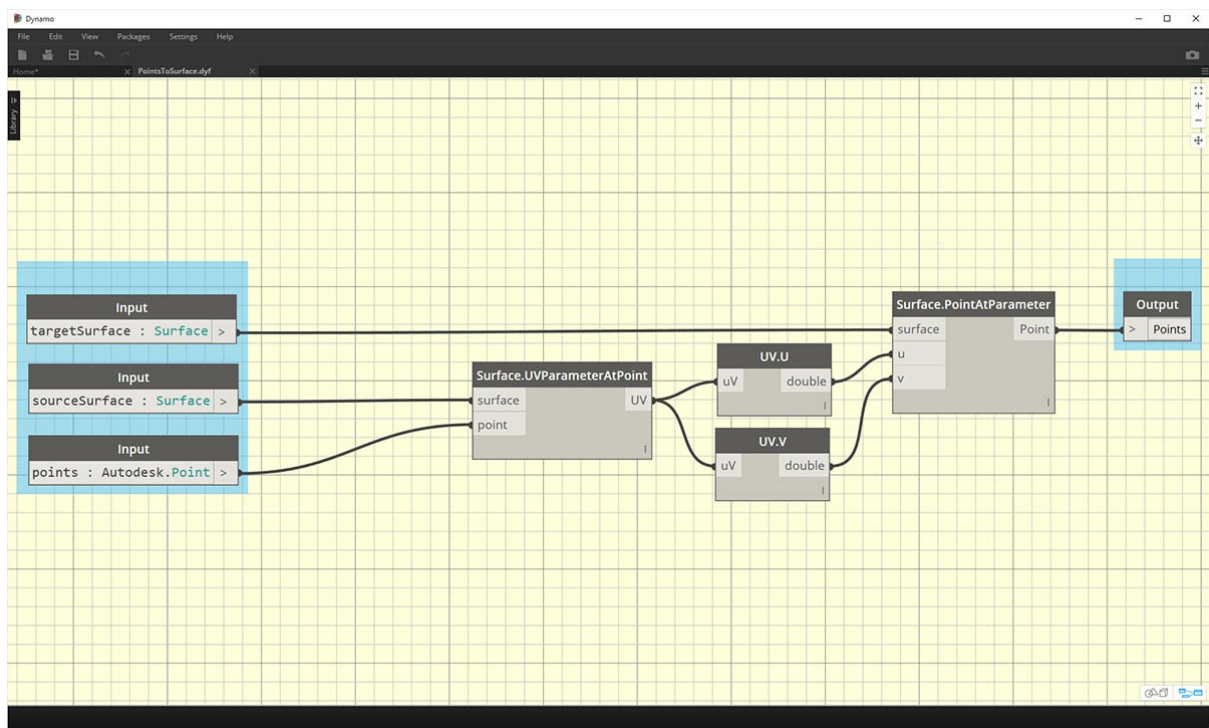
1. Tras la instalación, los nodos personalizados deberían estar disponibles en el grupo "DynamoPrimer" o en la biblioteca de Dynamo. Con el paquete instalado, veamos cómo se configura.

Nodos personalizados

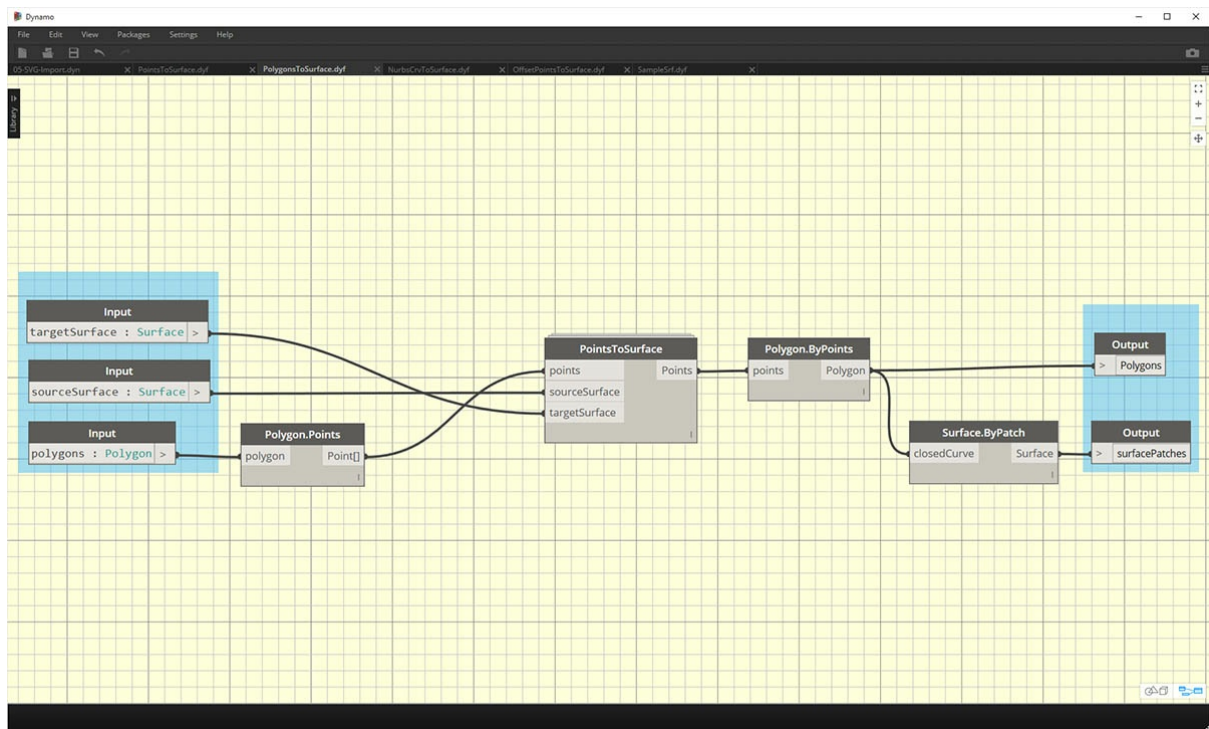
El paquete que estamos creando utiliza cinco nodos personalizados que hemos creado como referencia. Veamos qué hace cada nodo a continuación. Algunos nodos personalizados se basan en otros nodos personalizados y los gráficos tienen un diseño que los demás usuarios pueden comprender con total claridad.



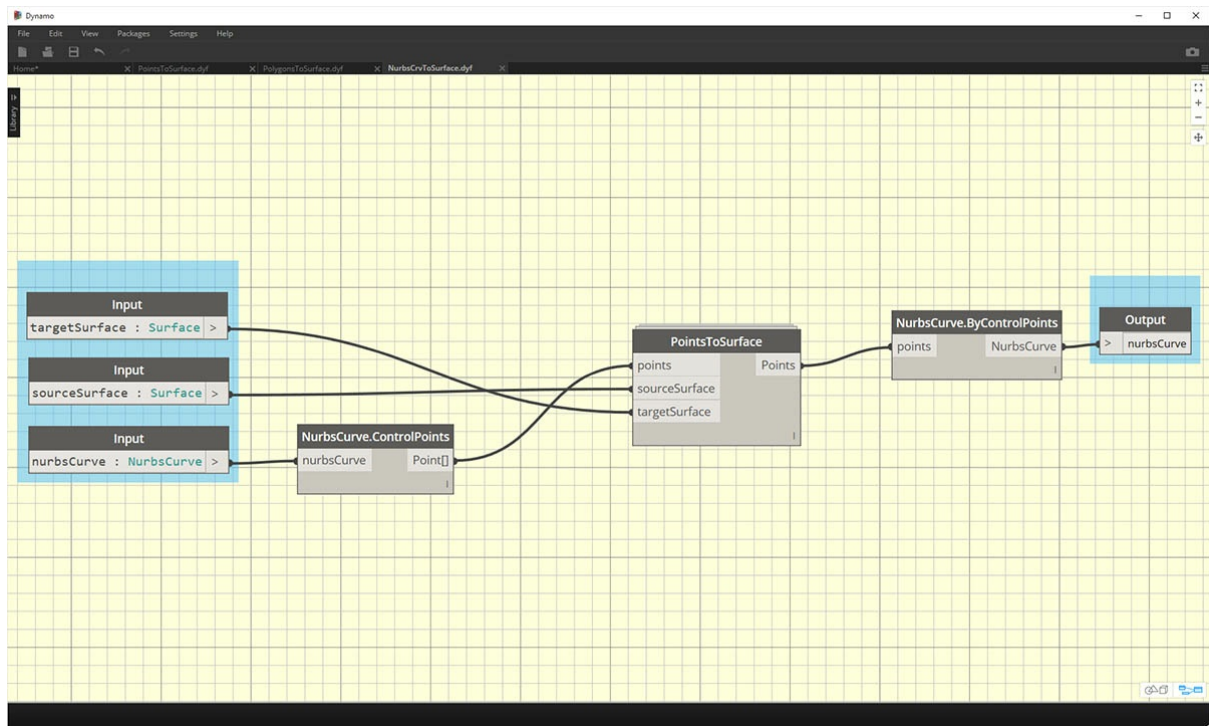
Este es un paquete sencillo con cinco nodos personalizados. En los pasos siguientes, hablaremos brevemente de la configuración de cada nodo personalizado.



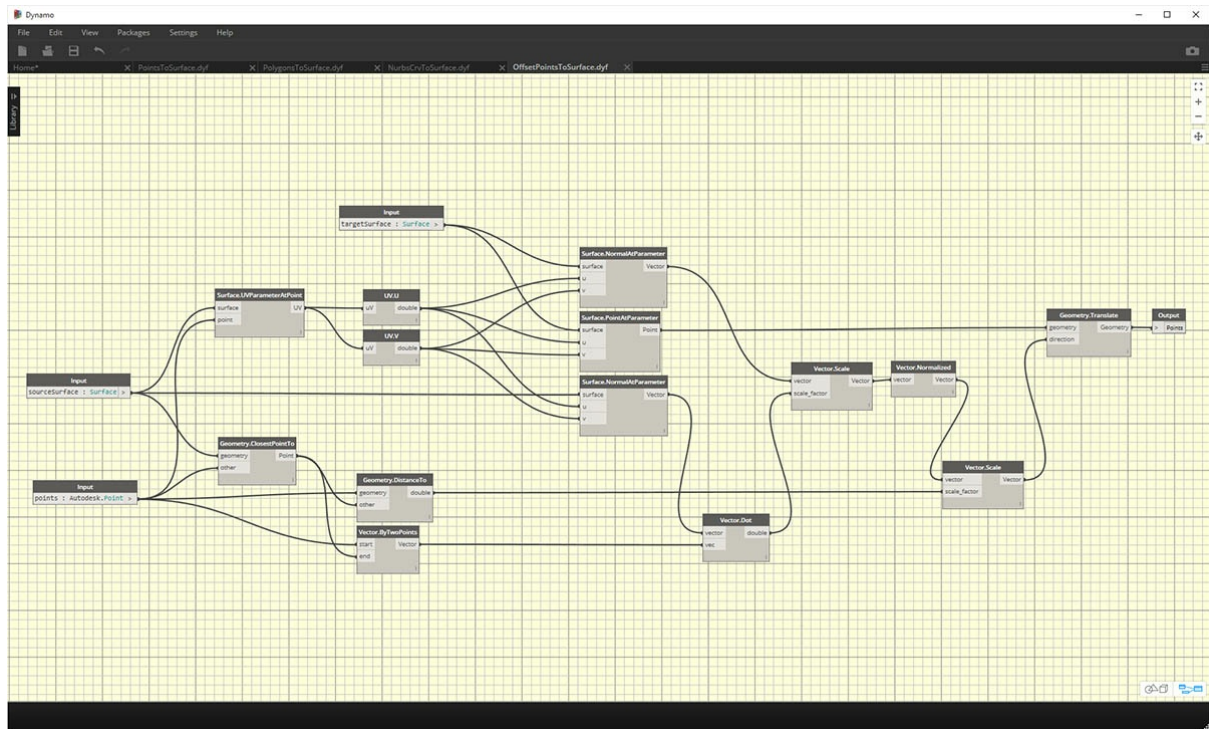
PointsToSurface: se trata de un nodo personalizado básico en el que se basan los demás nodos de asignación. En pocas palabras, el nodo asigna un punto desde una coordenada UV de la superficie de origen a la ubicación de la coordenada UV de la superficie de destino. Como los puntos son la geometría más primitiva, a partir de la cual se genera geometría más compleja, podemos utilizar esta lógica para asignar geometría 2D e incluso geometría 3D de una superficie a otra.



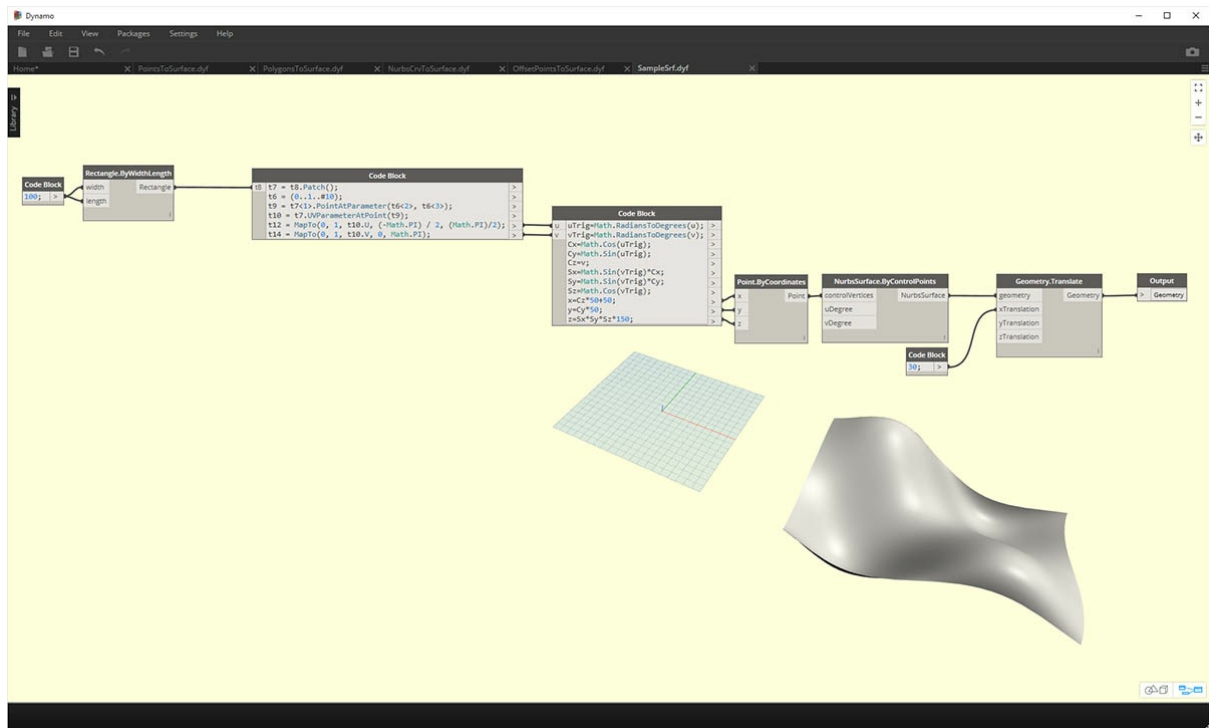
PolygonsToSurface: la lógica de ampliar puntos asignados de geometría 1D a geometría 2D se muestra aquí de forma sencilla con polígonos. Observe que se ha anidado el nodo "PointsToSurface" en este nodo personalizado. De esta forma, podemos asignar los puntos de cada polígono a la superficie y, a continuación, volver a generar el polígono a partir de los puntos asignados. Si se mantiene la estructura de datos correcta (una lista de listas de puntos), se pueden mantener los polígonos separados después de reducirlos a un conjunto de puntos.



NurbsCrvToSurface: aquí se aplica la misma lógica que en el nodo "PolygonsToSurface". En lugar de asignar puntos poligonales, asignamos puntos de control de una curva NURBS.



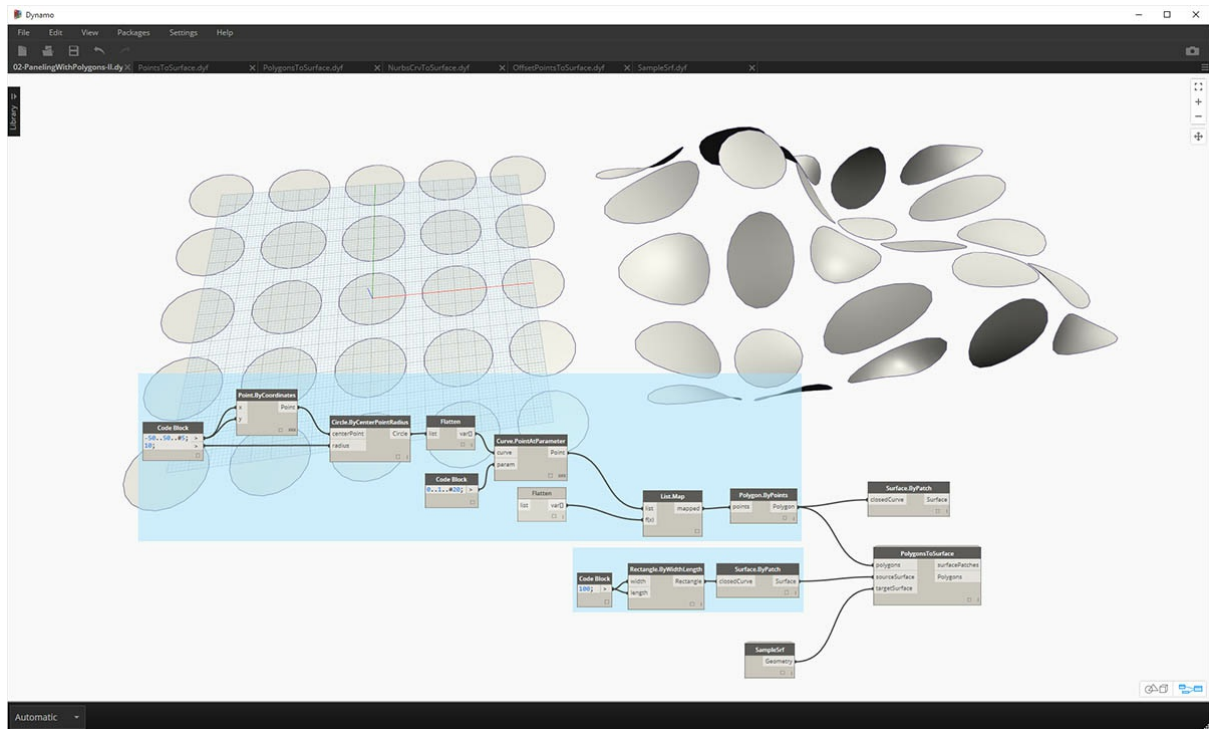
OffsetPointsToSurface: este nodo es un poco más complejo, pero el concepto es simple: como el nodo "PointsToSurface", este nodo asigna puntos de una superficie a otra. Sin embargo, también considera los puntos que no se encuentran en la superficie de origen inicial, obtiene su distancia al parámetro UV más cercano y asigna esta distancia a la normal de la superficie de destino en la coordenada UV correspondiente. Esto se ve con mayor claridad al examinar los archivos de ejemplo.



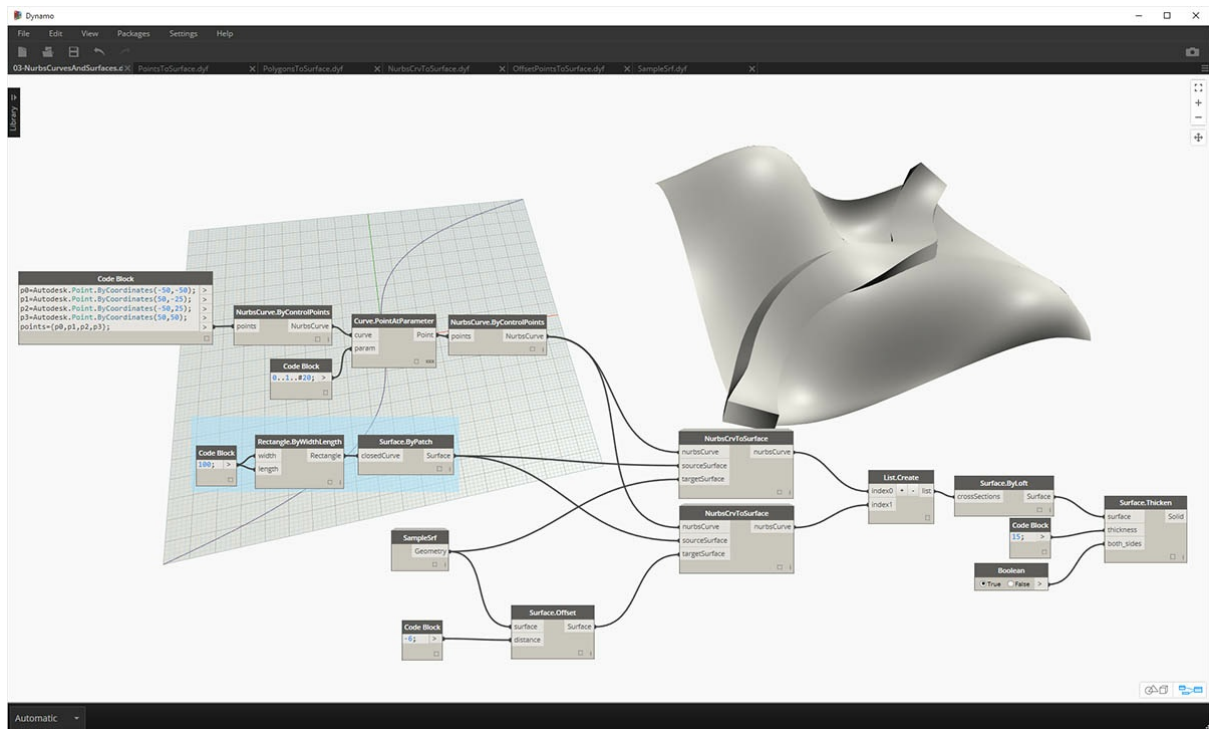
SampleSrf: se trata de un nodo simple que crea una superficie paramétrica para asignar de la rejilla de origen a una superficie ondulada en los archivos de ejemplo.

Archivos de ejemplo

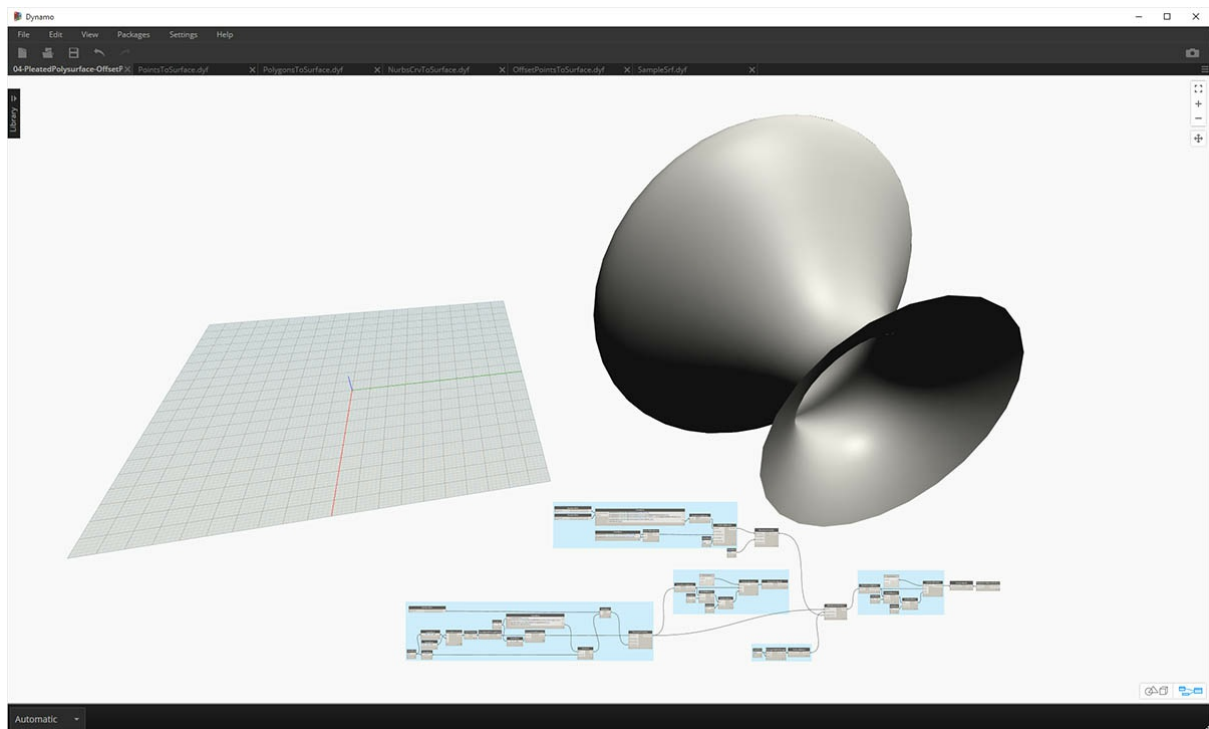
Los archivos de ejemplo se pueden encontrar en la carpeta raíz del paquete (en Dynamo, vaya a esta carpeta seleccionando *Paquetes > Gestionar paquetes...*).



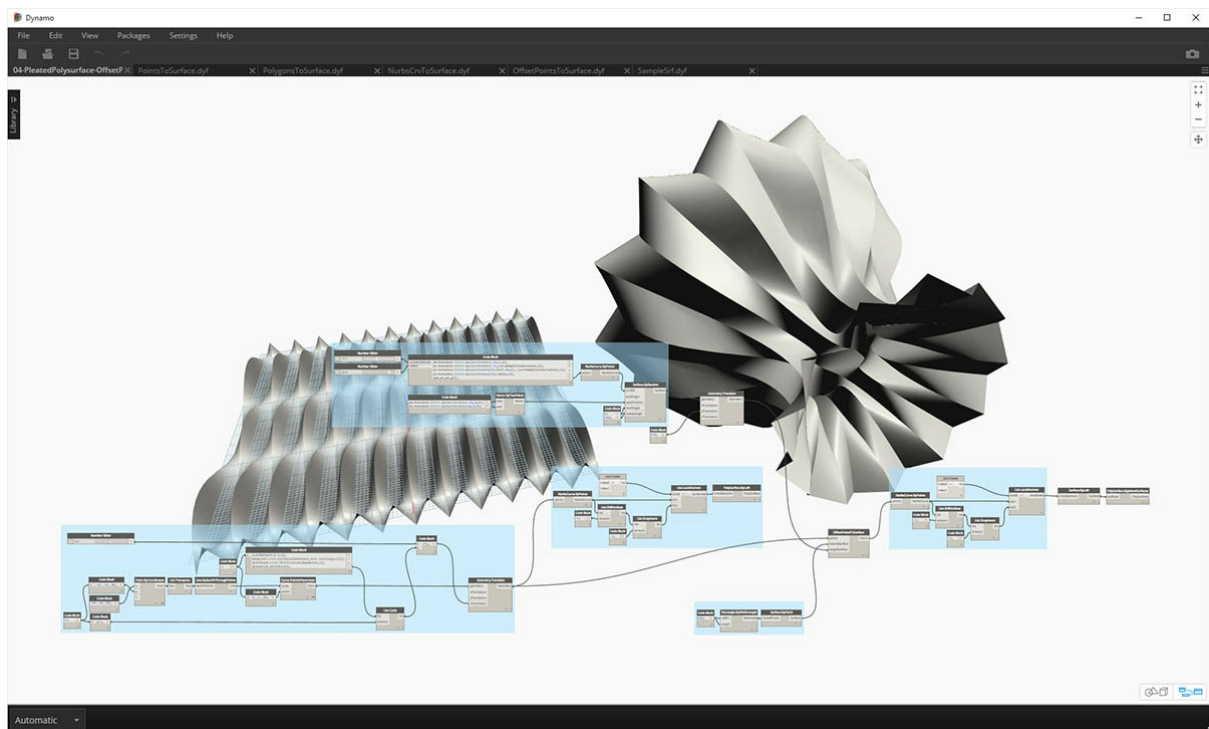
02-PanelingWithPolygons-II: mediante un flujo de trabajo similar, este archivo de ejercicio muestra una configuración para la asignación de círculos (o polígonos que representan círculos) de una superficie a otra. Utiliza el nodo "PolygonsToSurface".



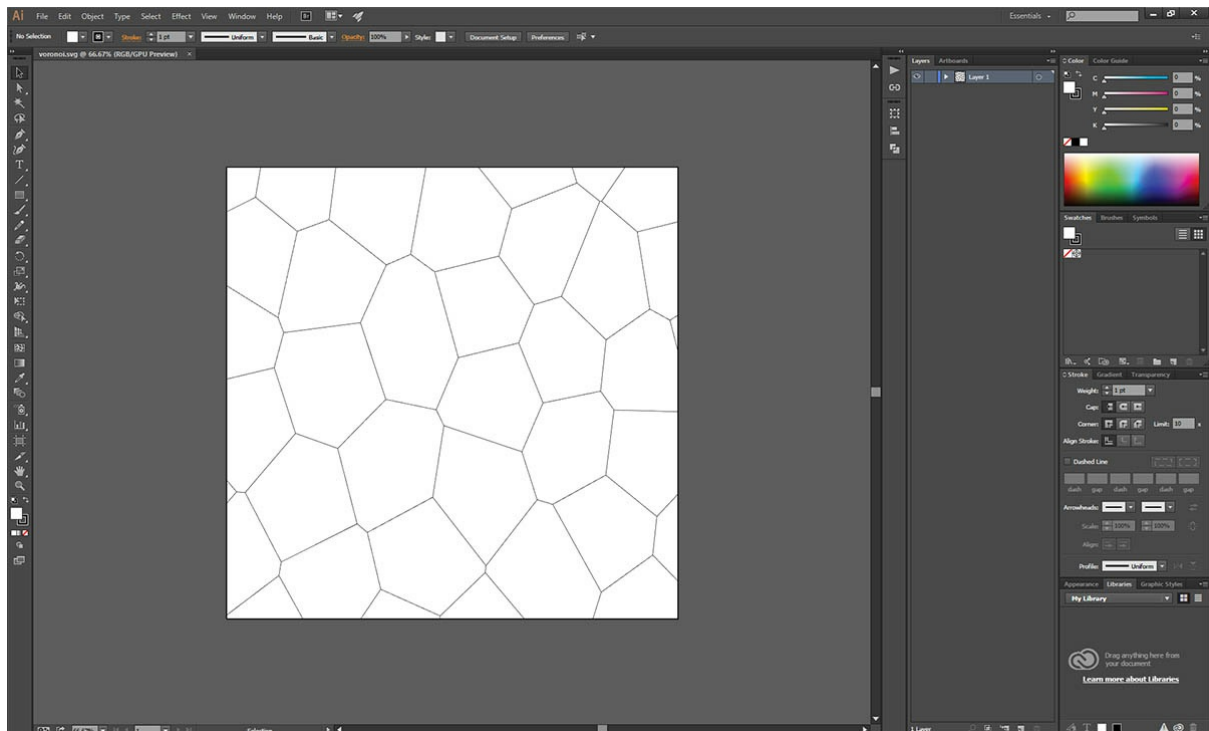
03-NurbsCurvesAndSurface: este archivo de ejemplo añade cierta complejidad al trabajar con el nodo "NurbsCrvToSurface". La superficie de destino se desfasa una distancia dada y la curva NURBS se asigna a la superficie de destino original y a la superficie de desfase. A partir de ahí, las dos curvas asignadas se sollevan para crear una superficie que, a continuación, se engrosa. Este sólido resultante tiene una ondulación que es representativa de las normales de la superficie de destino.



04-PleatedPolysurface-OffsetPoints: este archivo de ejemplo muestra cómo asignar una PolySurface plegada de una superficie de origen a una superficie de destino. Las superficies de origen y de destino son superficies rectangulares que abarcan la rejilla y una superficie de revolución, respectivamente.



04-PleatedPolysurface-OffsetPoints: la PolySurface de origen asignada desde la superficie de origen a la superficie de destino.



05-SVG-Import: dado que los nodos personalizados pueden asignar diferentes tipos de curvas, este último archivo hace referencia a un archivo SVG exportado de Illustrator y asigna las curvas importadas a una superficie de destino.

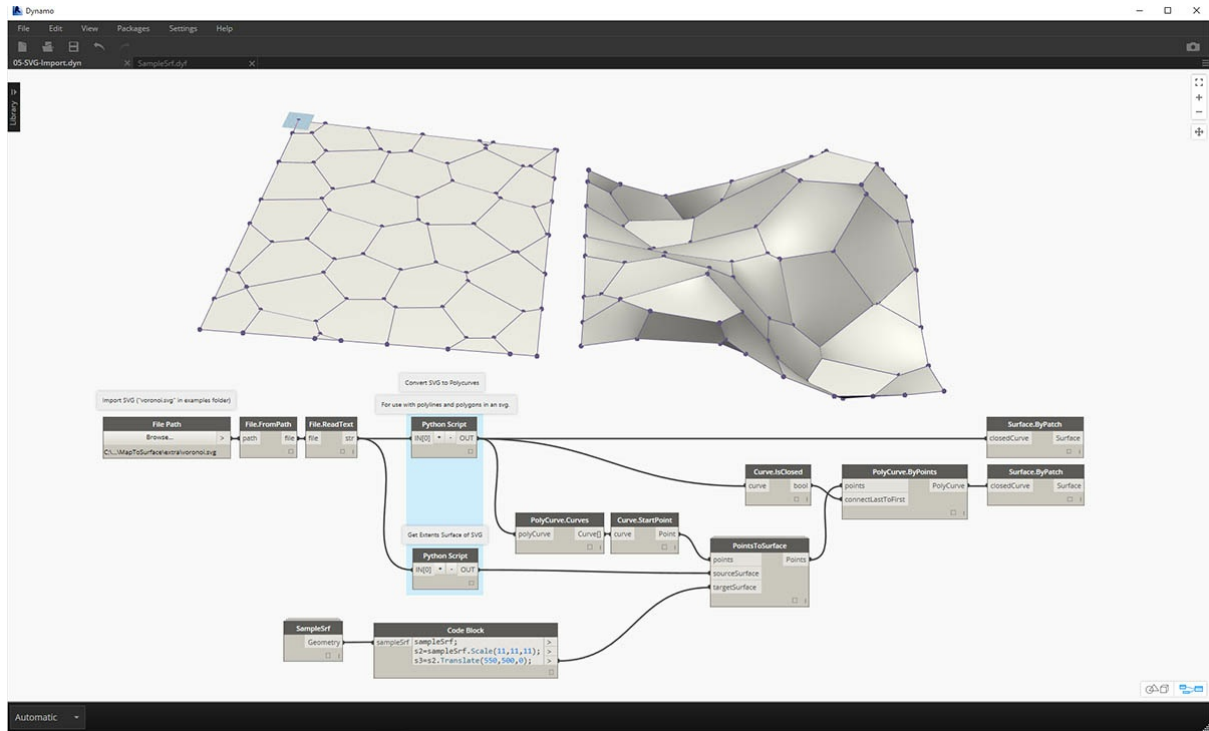
```

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: mediante el análisis de la sintaxis de un archivo .svg, las curvas se trasladan del formato .xml a PolyCurves de Dynamo.

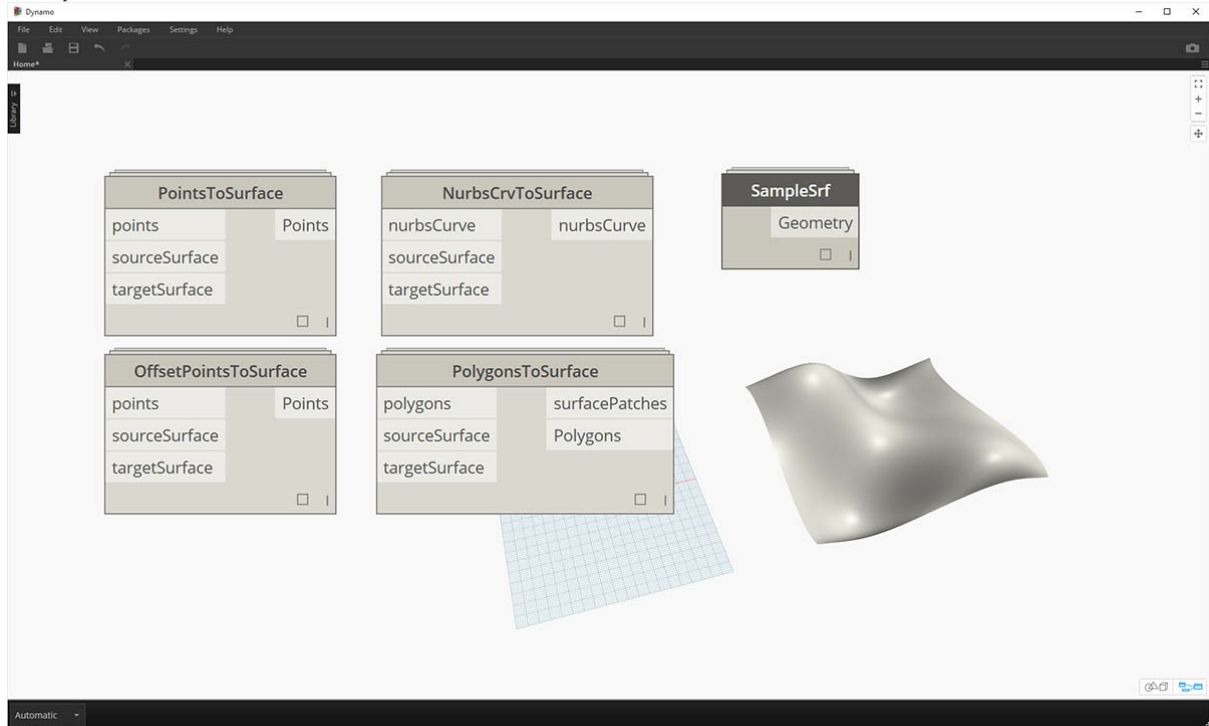


05-SVG-Import: las curvas importadas se asignan a una superficie de destino. Esto nos permite diseñar de forma explícita (señalar y hacer clic) una panelización en Illustrator, importarla en Dynamo y aplicarla a una superficie de destino.

Publicación de un paquete

Publicación de un paquete

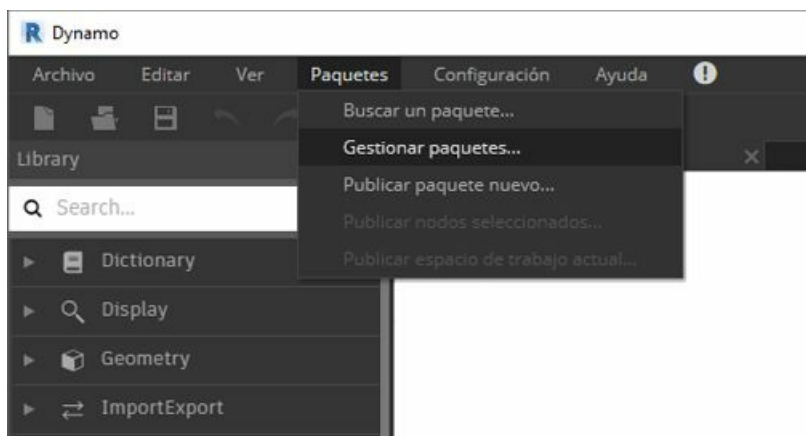
En las secciones anteriores, examinamos cómo se configura el paquete de *MapToSurface* con nodos personalizados y archivos de ejemplo. Pero, ¿cómo publicamos un paquete que se ha desarrollado localmente? En este caso real, se muestra cómo publicar un paquete a partir de un conjunto de archivos en una carpeta local.



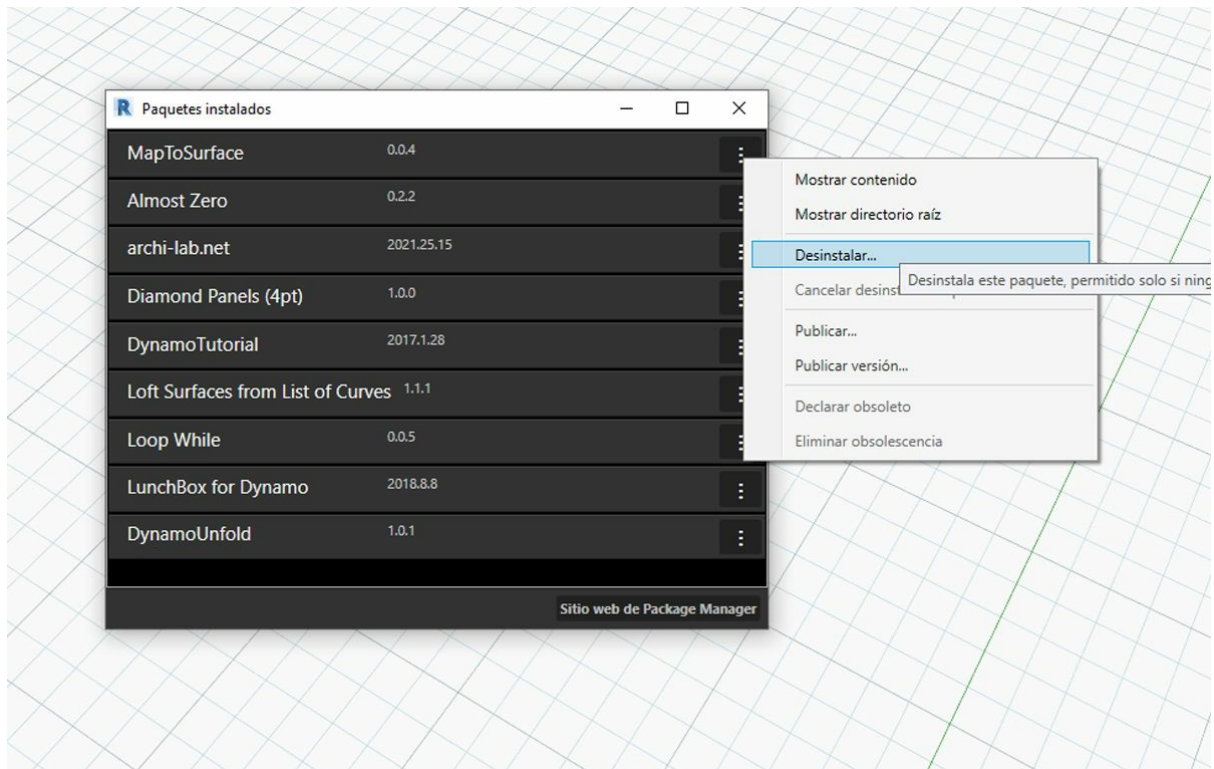
Hay muchas formas de publicar un paquete. A continuación, se indica el proceso recomendado: **publique localmente, desarrolle localmente y, a continuación, publique en línea**. Comenzaremos con una carpeta que contiene todos los archivos del paquete.

Desinstalación de un paquete

Antes de pasar a la publicación del paquete *MapToSurface*, si ha instalado el paquete en una lección anterior, desinstálelo para que no trabaje con paquetes idénticos.



Vaya primero *Paquetes* > *Gestionar paquetes*....

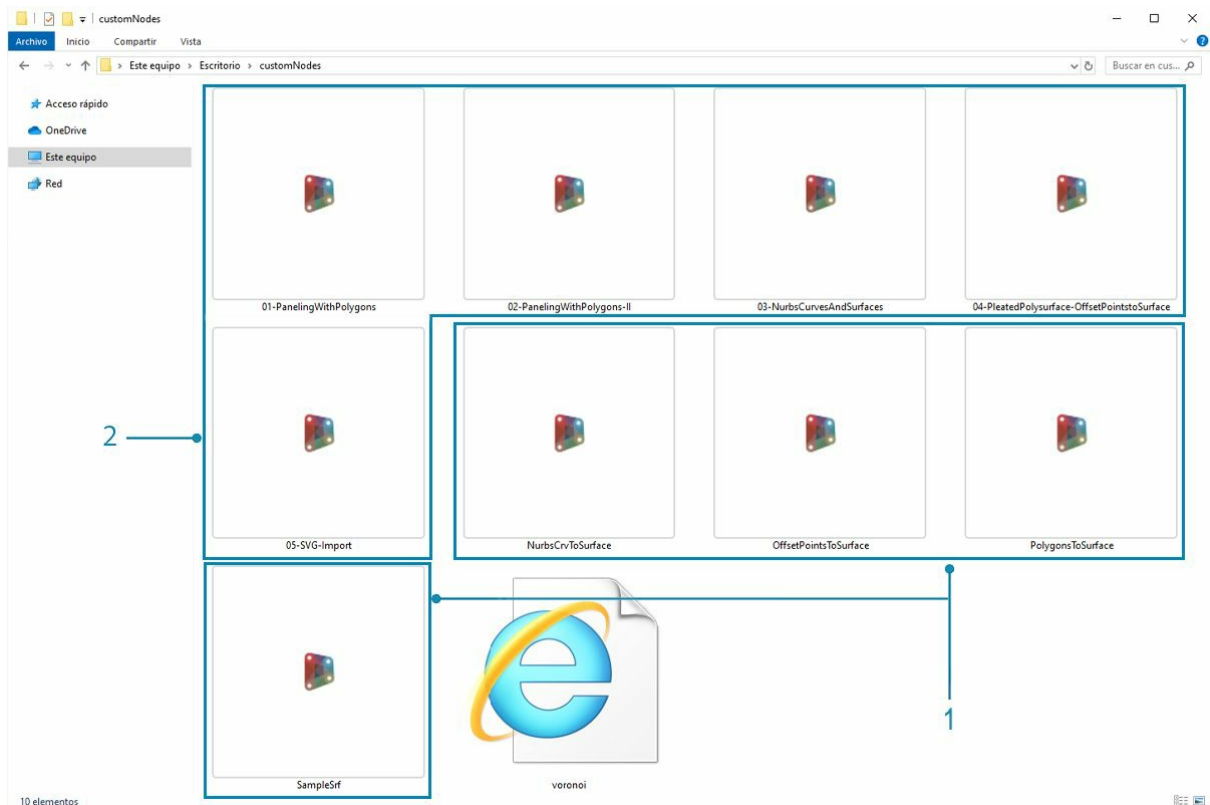


Seleccione el botón correspondiente a *MapToSurface* y, a continuación, seleccione *Desinstalar...*. A continuación, reinicie Dynamo. Al volver a abrirlo, cuando active la ventana *Gestionar paquetes*, *MapToSurface* ya no debería aparecer. Ya estamos listos para empezar desde el principio.

Publicación local de un paquete

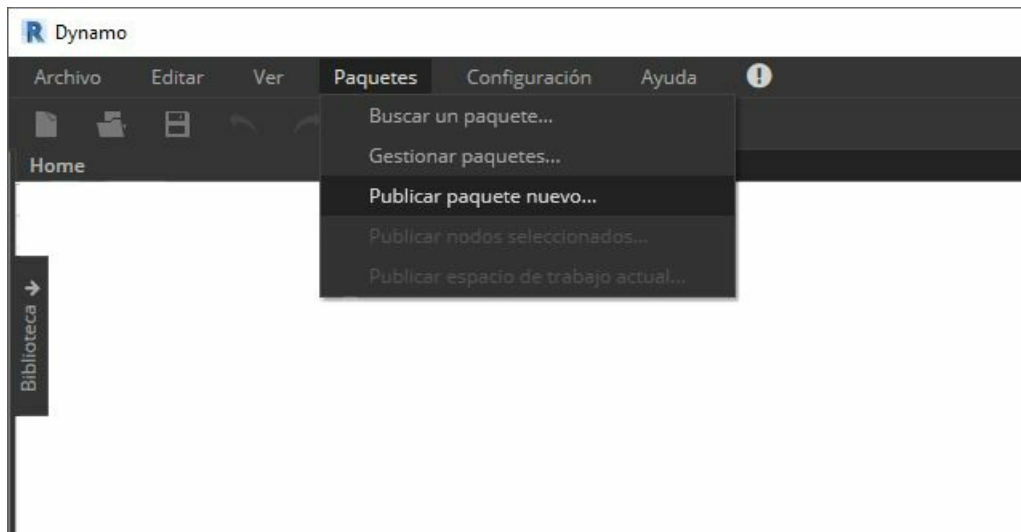
Nota A medida que escriba, la publicación del paquete de Dynamo solo se activará en Dynamo for Revit y Dynamo for Civil 3d. Dynamo Sandbox no incluye funciones de publicación.

Descargue y descomprima los archivos de ejemplo que acompañan a este caso real de paquete (haga clic con el botón derecho y seleccione "Guardar enlace como..."). En el Apéndice, se incluye una lista completa de los archivos de ejemplo. [MapToSurface.zip](#)

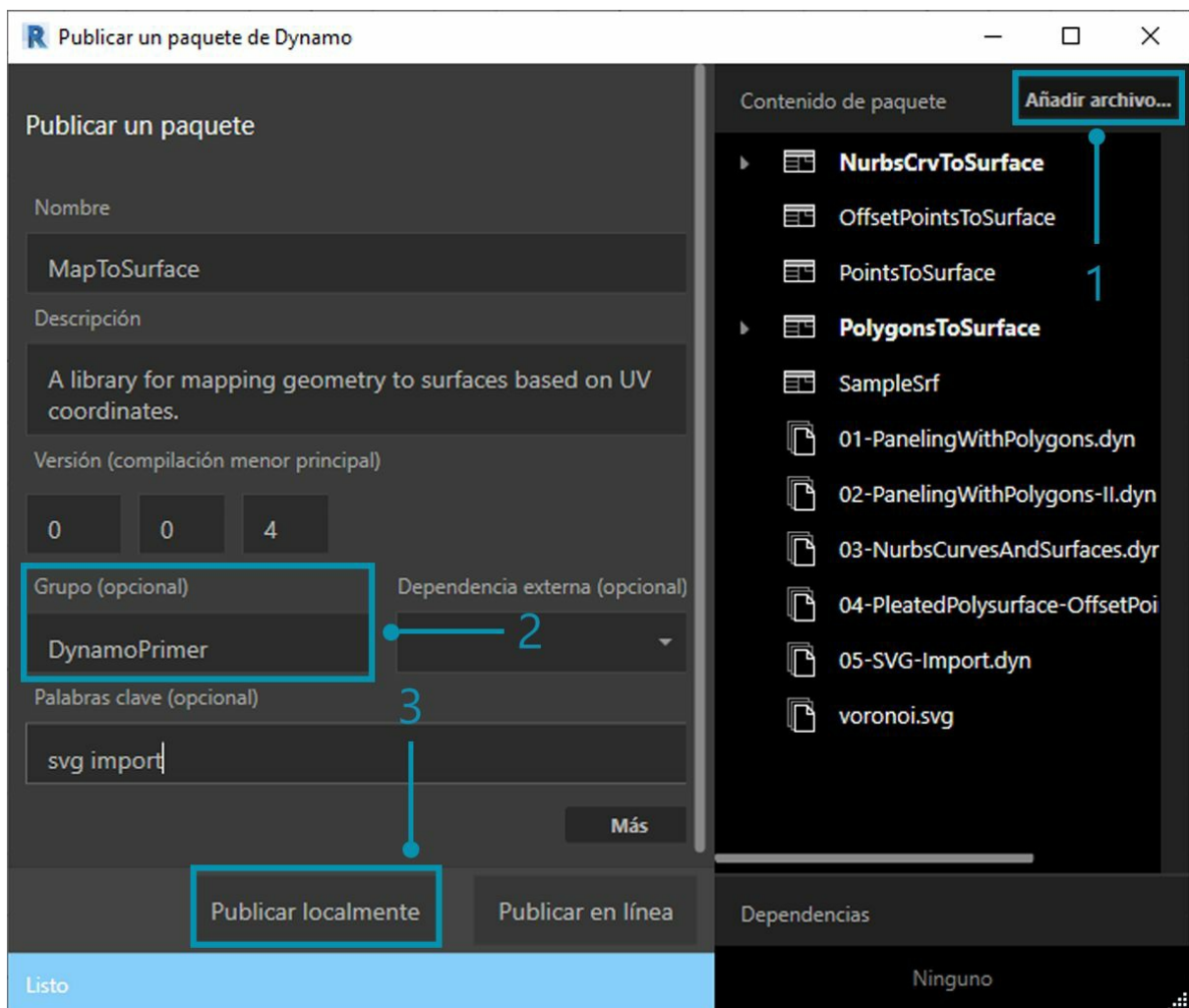


Este es el primer envío del paquete y se han incluido todos los nodos personalizados y los archivos de ejemplo en una única carpeta. Con esta carpeta preparada, ya podemos realizar la carga en Dynamo Package Manager.

1. Esta carpeta contiene cinco nodos personalizados (.dyf).
2. Esta carpeta también contiene cinco archivos de ejemplo (.dyn) y un archivo de vectores importado (.svg). Estos archivos se utilizarán como ejercicios de introducción para mostrar al usuario cómo trabajar con los nodos personalizados.



En Dynamo, vaya primero a *Paquetes > Publicar paquete nuevo...*

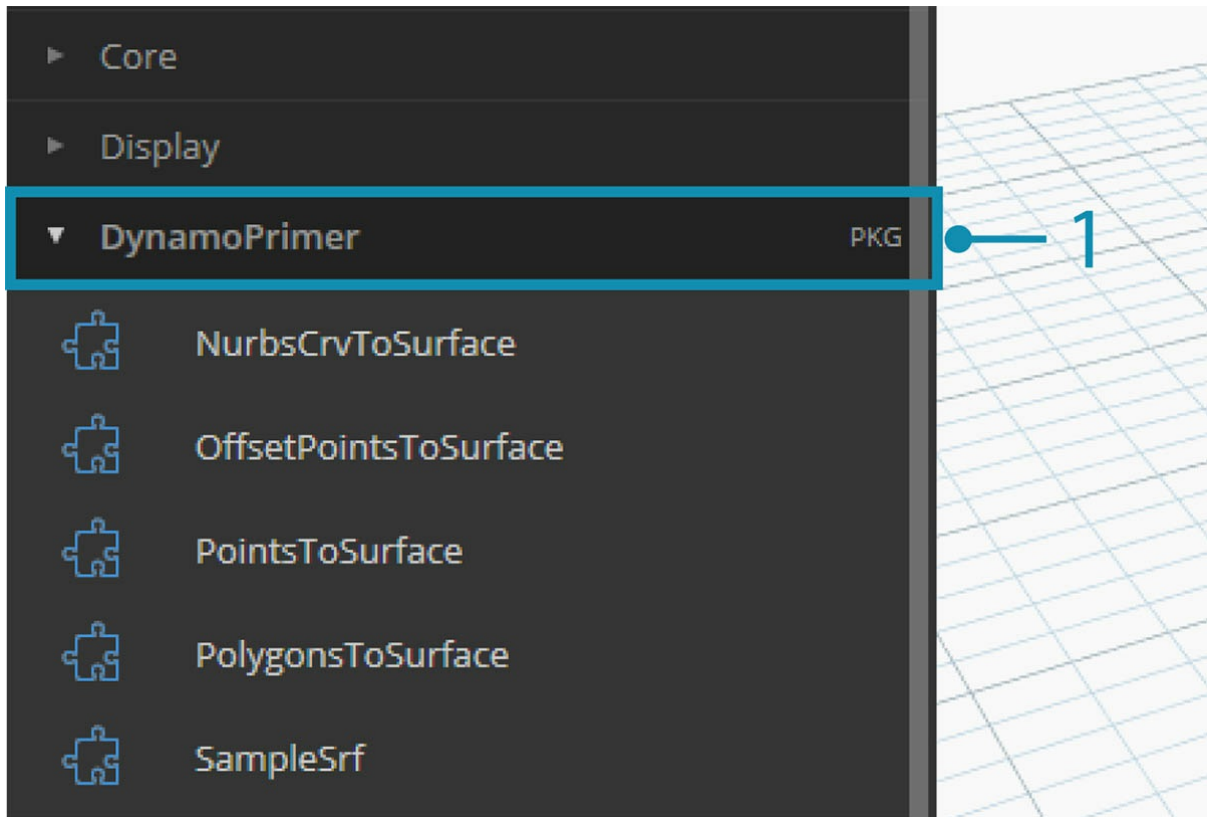


En la ventana *Publicar un paquete de Dynamo*, hemos rellenado los formularios correspondientes a la izquierda.

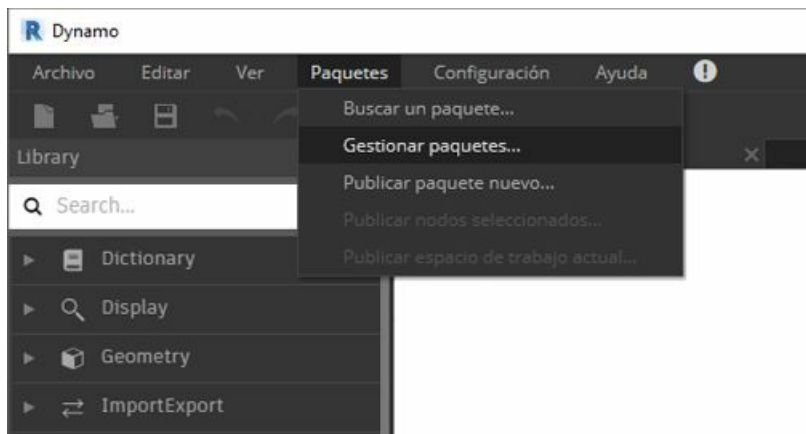
1. Al hacer clic en *Añadir archivo*, también hemos añadido los archivos de la estructura de carpetas en la parte derecha de la pantalla (para añadir

archivos que no sean .dyf, asegúrese de cambiar el tipo de archivo en la ventana del navegador a **Todos los archivos (.)**". Observe que se han añadido sin distinción todos los archivos, los nodos personalizados (.dyf) o los archivos de ejemplo (.dyn). Dynamo clasificará estos elementos cuando publiquemos el paquete.

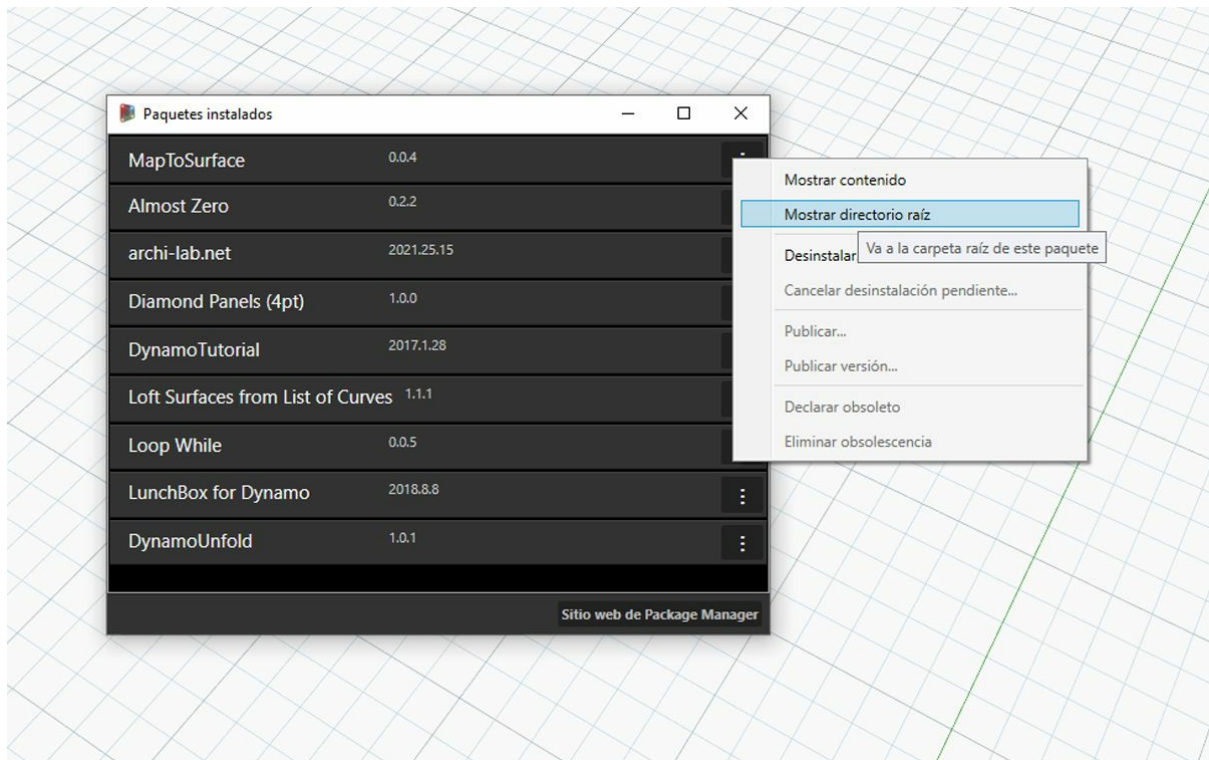
2. El campo "Grupo" define el grupo en el que buscar los nodos personalizados en la interfaz de usuario de Dynamo.
3. Para publicar, haga clic en "Publicar localmente". Si sigue este proceso, asegúrese de hacer clic en *Publicar localmente* y **no** en *Publicar en línea*. No deseamos un conjunto de paquetes duplicados en Package Manager.



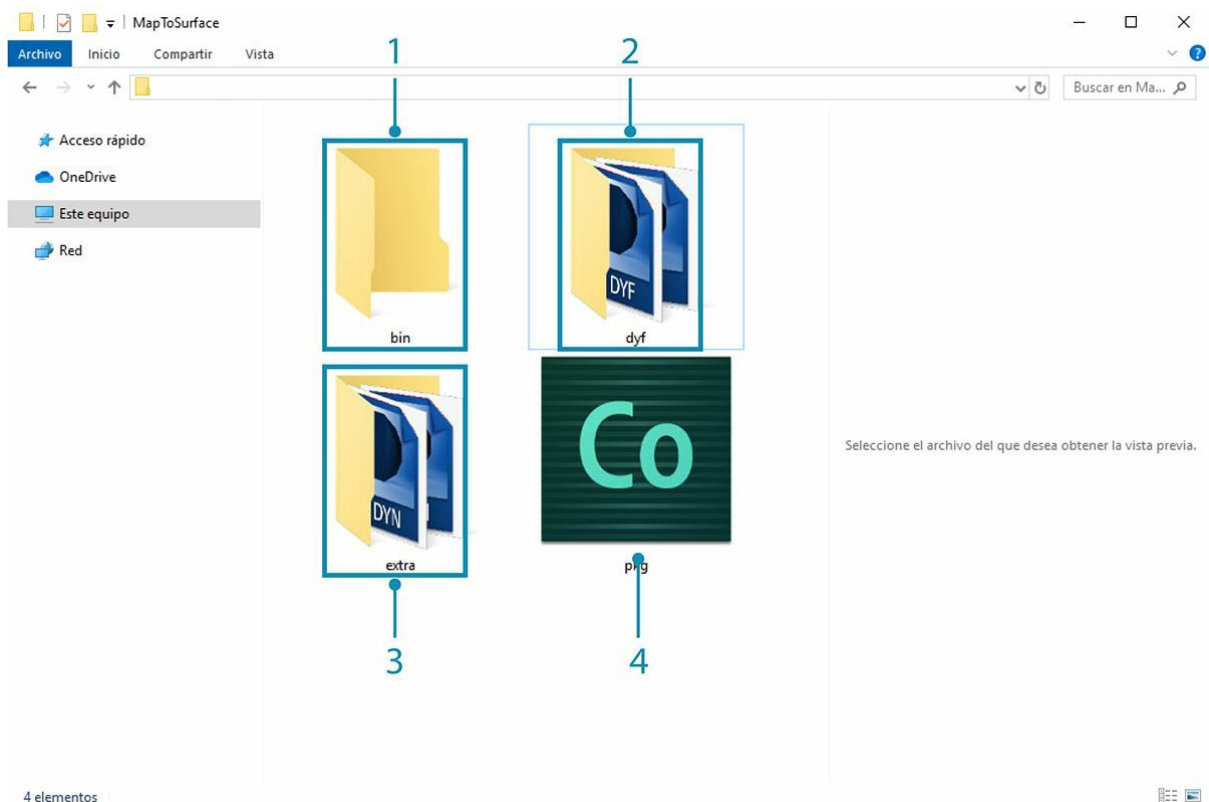
1. Tras la publicación, los nodos personalizados deberían estar disponibles en el grupo "DynamoPrimer" o en la biblioteca de Dynamo.



Ahora veamos el directorio raíz para comprobar cómo Dynamo ha aplicado formato al paquete que acabamos de crear. Para ello, haga clic en *Paquetes* > *Gestionar paquetes...*



En la ventana Gestionar paquetes, haga clic en los tres puntos verticales a la derecha de *MapToSurface* y elija *Mostrar directorio raíz*.

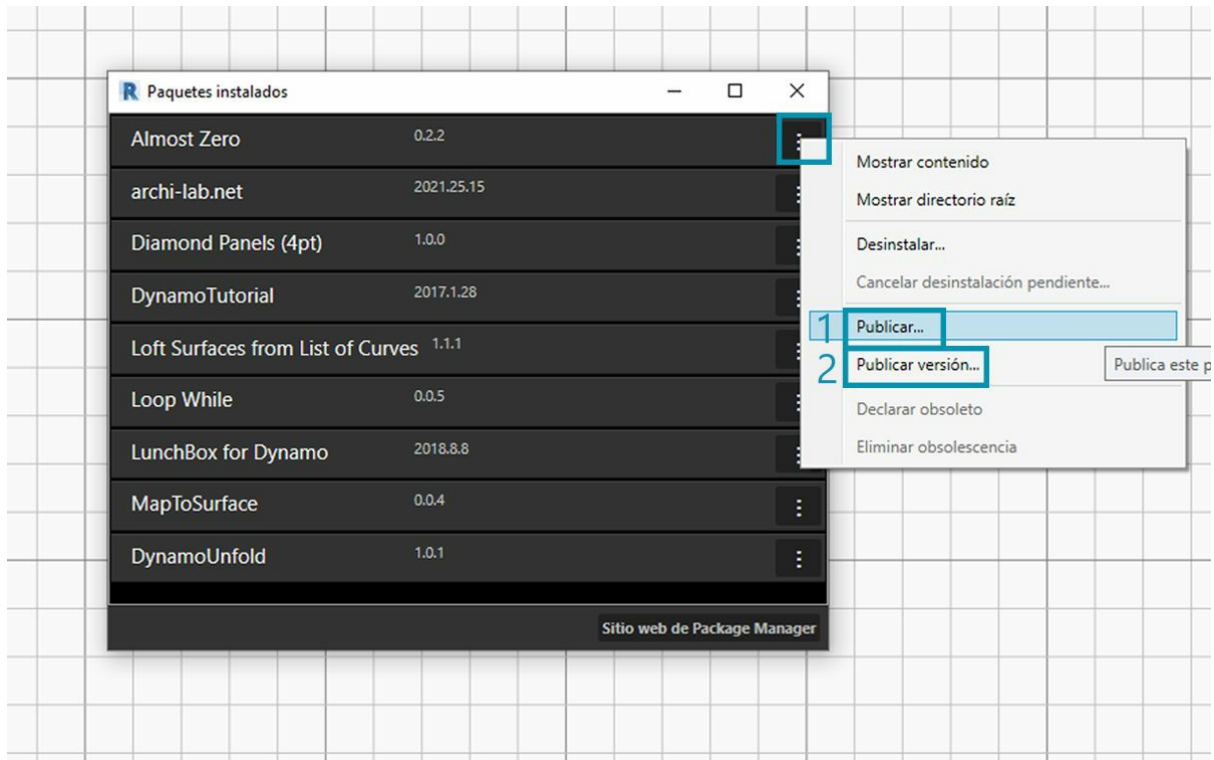


Observe que el directorio raíz se encuentra en la ubicación local del paquete (recuerde que hemos publicado el paquete "localmente"). Dynamo hace referencia a esta carpeta para leer nodos personalizados. Por lo tanto, es importante publicar localmente el directorio en una ubicación de carpeta permanente (es decir, no en el escritorio). A continuación, se desglosan las carpetas del paquete de Dynamo:

1. La carpeta *bin* contiene archivos .dll creados con bibliotecas de C# o Zero-Touch. No hay ninguna para este paquete, por lo que esta carpeta aparece vacía en este ejemplo.
2. La carpeta *dyf* contiene los nodos personalizados. Al abrir esta, se mostrarán todos los nodos personalizados (archivos .dyf) de este paquete.
3. La carpeta "extra" contiene todos los archivos adicionales. Es probable que estos archivos sean archivos de Dynamo (.dyn) o archivos adicionales necesarios (.svg, .xls, .jpeg, .sat, etc.).
4. El archivo *pkg* es un archivo de texto básico que define los parámetros del paquete. Esta función está automatizada en Dynamo, pero se puede

modificar si desea acceder a los detalles.

Publicación de un paquete en línea



Nota: no siga este paso, a menos que publique realmente un paquete suyo.

1. Cuando esté listo para publicar, en la ventana Gestionar paquetes, seleccione el botón ubicado a la derecha de MapToSurface y elija *Publicar...*
2. Si va a actualizar un paquete que ya se ha publicado, seleccione *Publicar versión*; Dynamo actualizará el paquete en línea en función de los nuevos archivos del directorio raíz del paquete. Así de fácil.

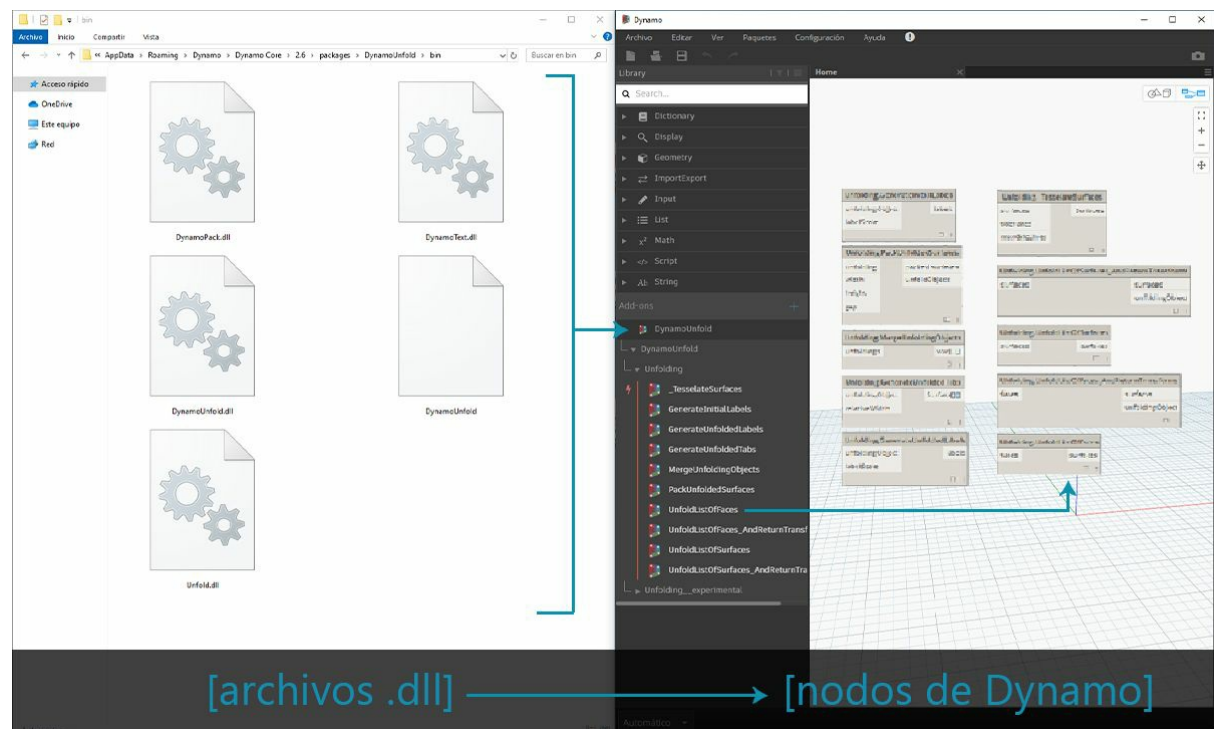
Publicar versión...

Al actualizar los archivos de la carpeta raíz del paquete publicado, puede publicar una nueva versión del paquete. Para ello, seleccione *Publicar versión...* en la ventana *Gestionar paquetes*. Esta es una forma perfecta de realizar las actualizaciones necesarias del contenido y compartirlas con la comunidad. *Publicar versión* solo funcionará si es la persona encargada del mantenimiento del paquete.

Importación de Zero-Touch

¿Qué es Zero-Touch?

La importación Zero-Touch hace referencia a un método sencillo para importar bibliotecas C# con solo apuntar y hacer clic. Dynamo leerá los métodos públicos de un archivo *.dll* y los convertirá en nodos de Dynamo. Puede utilizar Zero-Touch para desarrollar sus propios paquetes y nodos personalizados y para importar bibliotecas externas en el entorno de Dynamo.



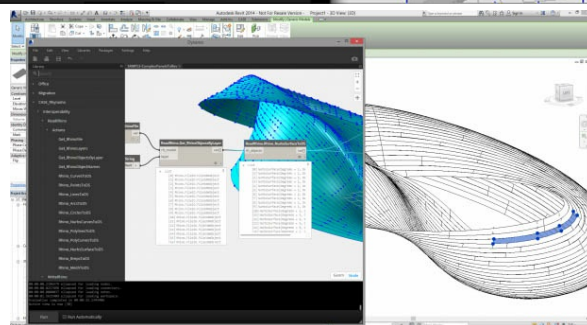
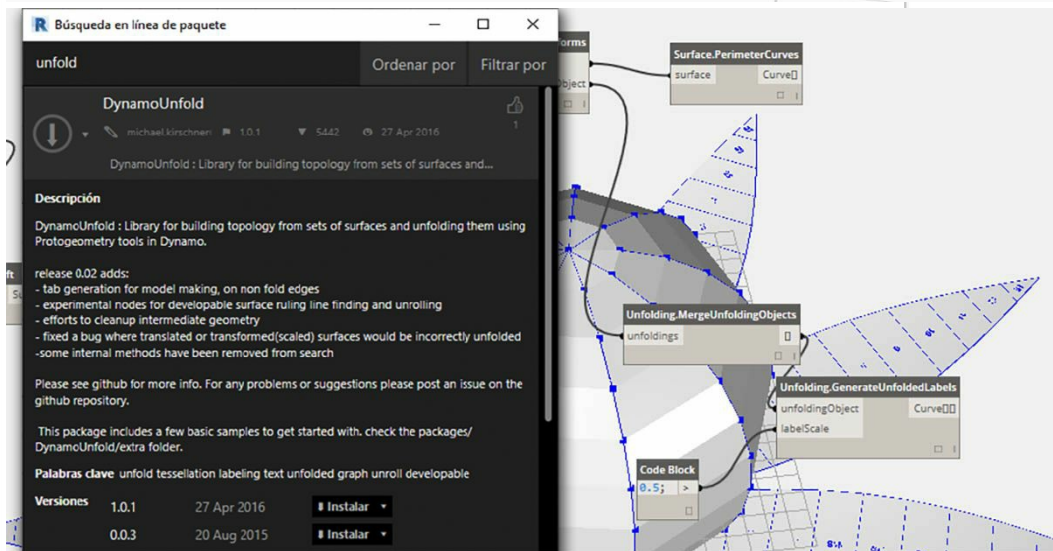
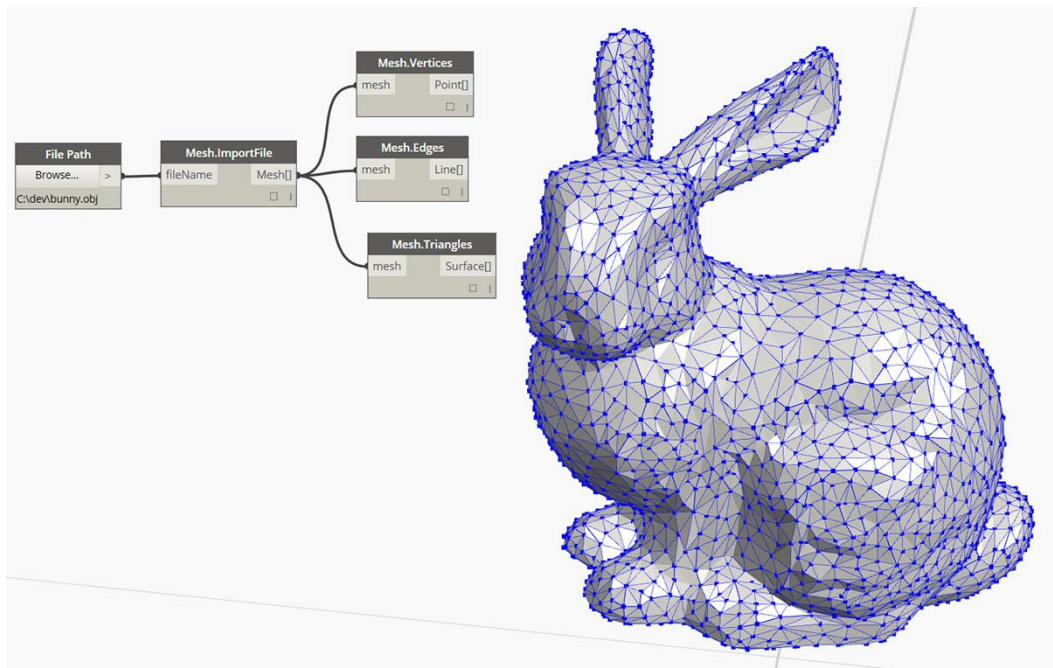
Con Zero-Touch, puede importar una biblioteca que no se haya desarrollado necesariamente para Dynamo y crear un conjunto de nodos nuevos. La función Zero-Touch muestra la mentalidad multiplataforma del proyecto Dynamo.

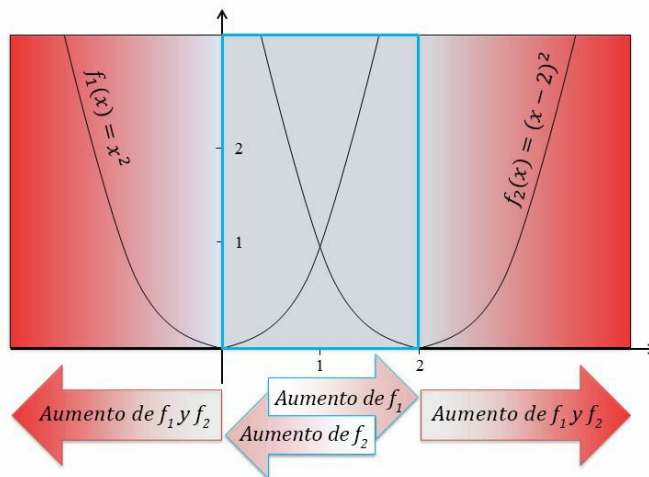
En esta sección se muestra cómo utilizar Zero-Touch para importar una biblioteca de terceros. Para obtener información sobre el desarrollo de su propia biblioteca Zero-Touch, consulte la [página wiki de Dynamo](#).

Paquetes de Zero-Touch

Los paquetes de Zero-Touch son un buen complemento para los nodos personalizados definidos por el usuario. En la tabla siguiente se muestran algunos paquetes que utilizan bibliotecas C#. Para obtener información más detallada sobre los paquetes, visite la [sección Paquetes](#) del Apéndice.

Logotipo/imagen



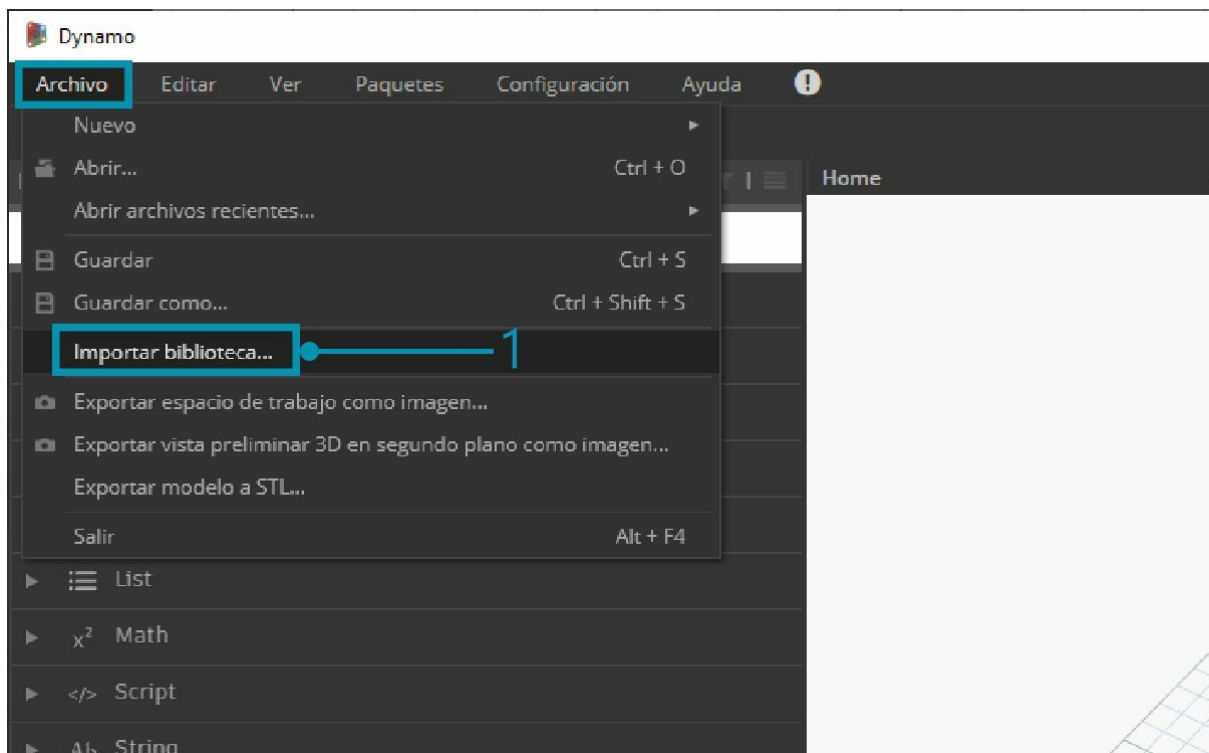


Caso real: importación de AForge

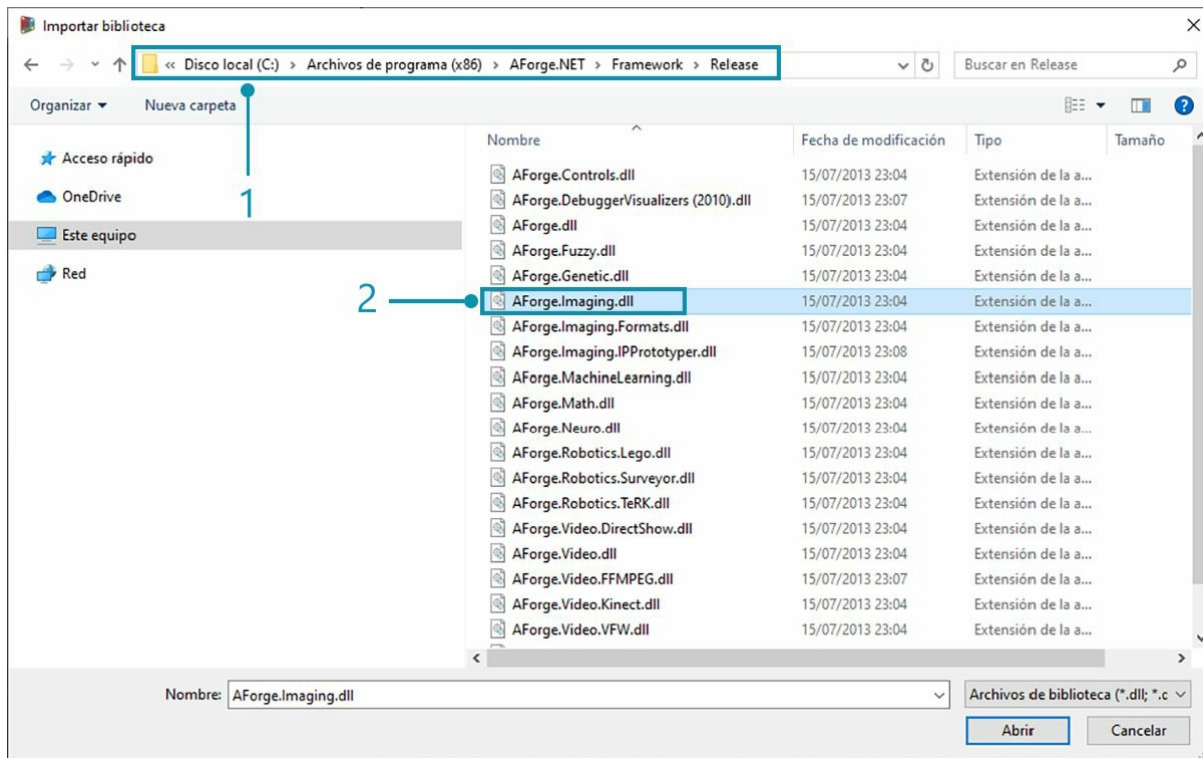
En este caso real, mostraremos cómo importar la biblioteca externa [.dll AForge](#). AForge es una biblioteca sólida que ofrece una gama de funciones que van desde el procesamiento de imágenes hasta la inteligencia artificial. Haremos referencia a la clase de imágenes en AForge para realizar algunos ejercicios de procesamiento de imágenes a continuación.

Descargue y descomprima los archivos de ejemplo que acompañan a este caso real de paquete (haga clic con el botón derecho y seleccione "Guardar enlace como..."). En el Apéndice, se incluye una lista completa de los archivos de ejemplo. [Zero-Touch-Examples.zip](#).

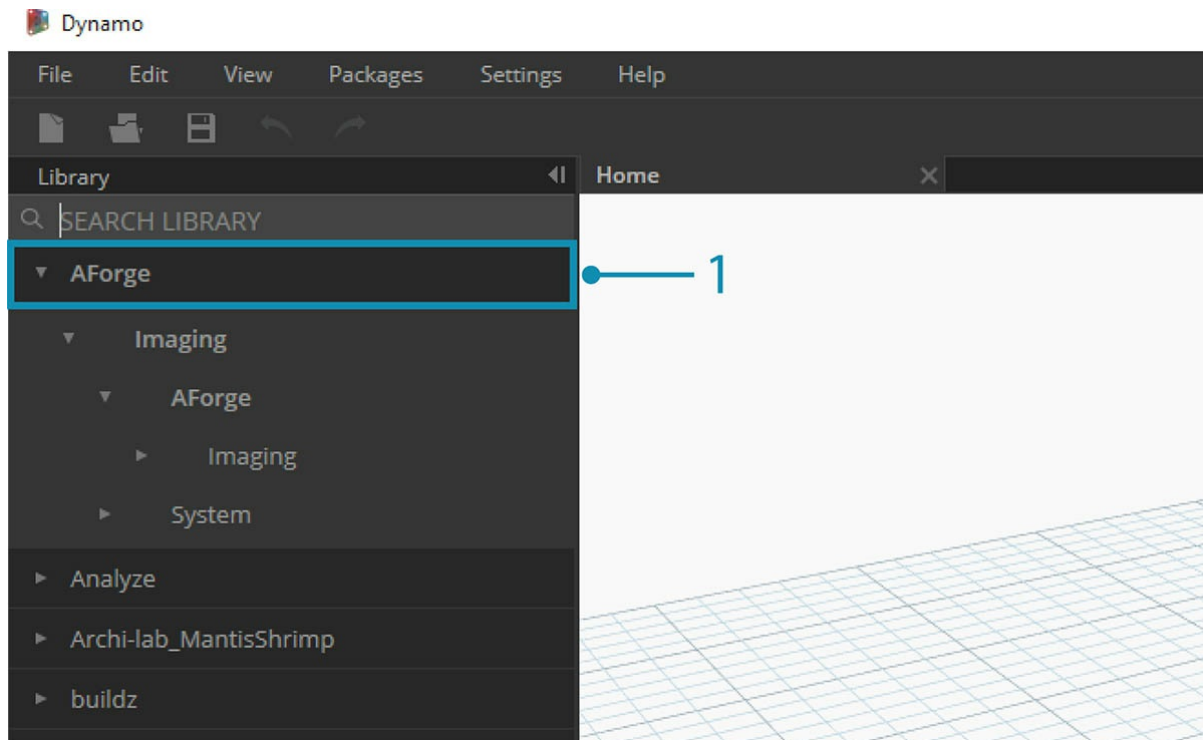
1. Comencemos por descargar AForge. En la [página de descarga de AForge](#), seleccione [Download Installer] (Descargar instalador) e instálelo una vez que haya finalizado la descarga.



1. En Dynamo, cree un archivo nuevo y seleccione *Archivo > Importar biblioteca...*



1. En la ventana emergente, vaya a la carpeta de la versión contenida en la ubicación de instalación de AForge. Probablemente se encuentre en una carpeta similar a esta: `C:\Program Files (x86)\AForge.NET\Framework\Release`.
2. **AForge.Imaging.dll**: solo vamos a utilizar este archivo de la biblioteca AForge para este caso real. Seleccione este archivo `.dll` y pulse "Abrir".



1. Al regresar a Dynamo, debería ver el grupo de nodos "AForge" añadido a la barra de herramientas Biblioteca. Ahora tenemos acceso a la biblioteca de imágenes AForge desde nuestro programa visual.

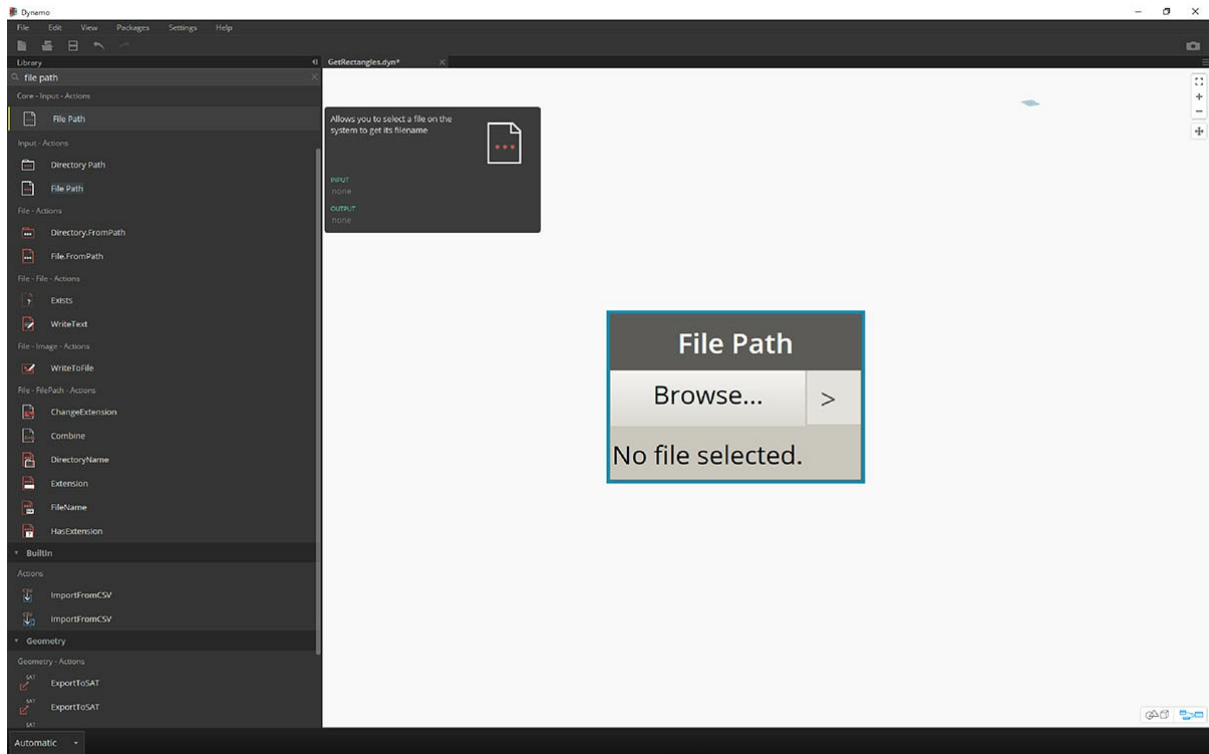
Ejercicio 1: detección de borde

Ahora que la biblioteca se ha importado, empezaremos con un sencillo ejercicio. Vamos a llevar a cabo un procesamiento de imagen básico en una imagen de muestra para comprobar cómo se filtra la imagen AForge. Utilizaremos el nodo "Watch Image" para mostrar los resultados y aplicar filtros en Dynamo similares a los de Photoshop.

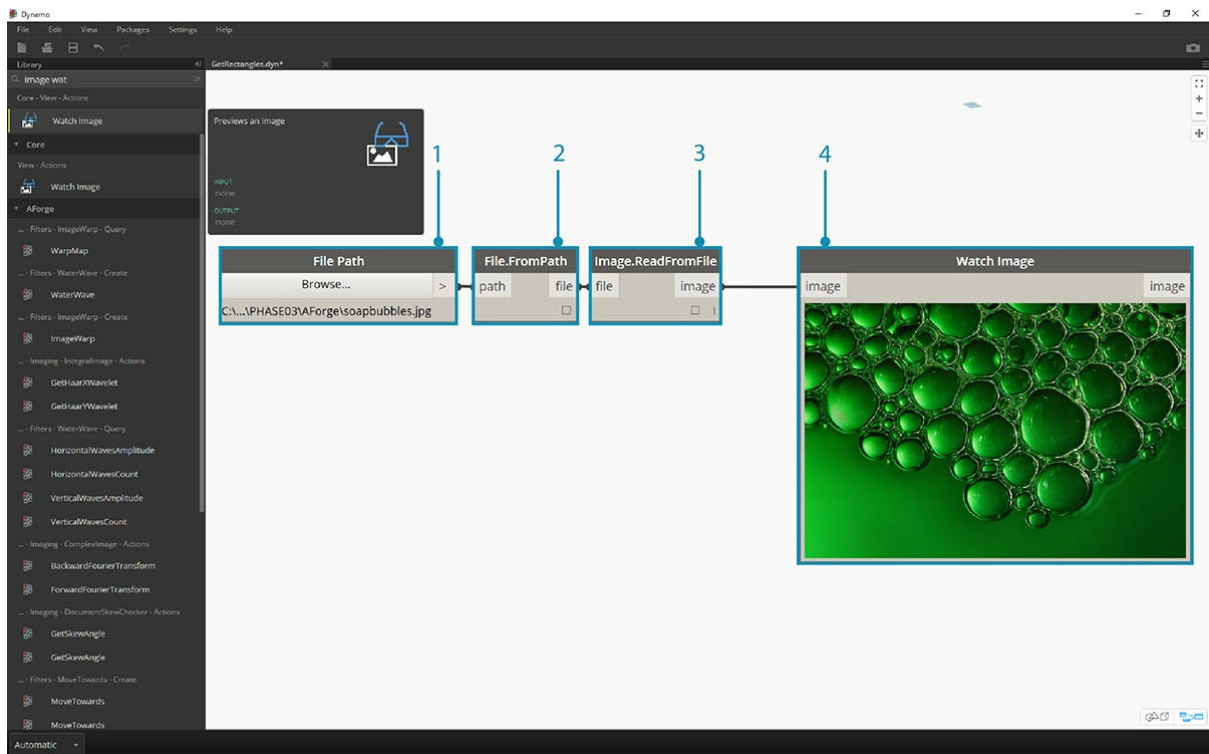
Descargue y descomprima los archivos de ejemplo que acompañan a este caso real de paquete (haga clic con el botón derecho y seleccione "Guardar

enlace como...). En el Apéndice, se incluye una lista completa de los archivos de ejemplo. [ZeroTouchImages.zip](#)

Ahora que la biblioteca se ha importado, empezaremos con un sencillo ejercicio (*01-EdgeDetection.dyn*). Vamos a llevar a cabo un procesamiento de imagen básico en una imagen de muestra para comprobar cómo se filtra la imagen AForge. Utilizaremos el nodo "Watch Image" para mostrar los resultados y aplicar filtros en Dynamo similares a los de Photoshop

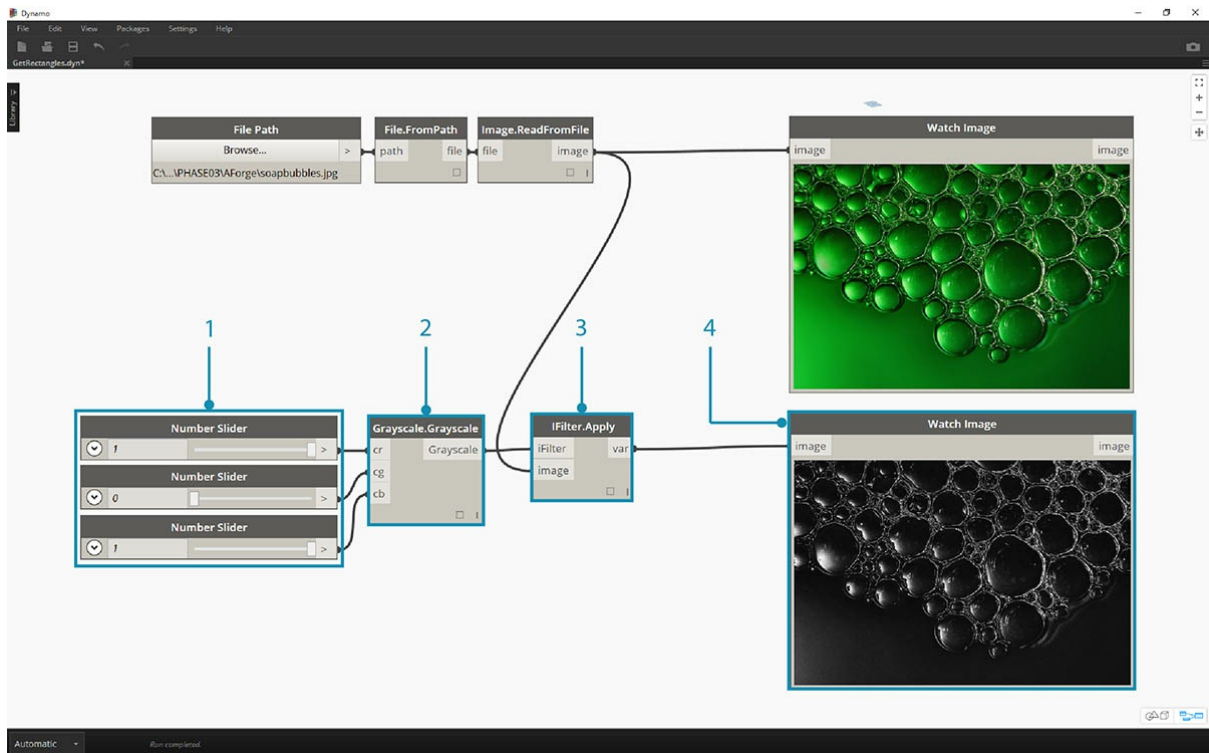


En primer lugar, vamos a importar una imagen con la que trabajar. Añada un nodo *File Path* al lienzo y seleccione "soapbubbles.jpg" en la carpeta del ejercicio que hemos descargado (crédito de la foto: [flickr](#)).



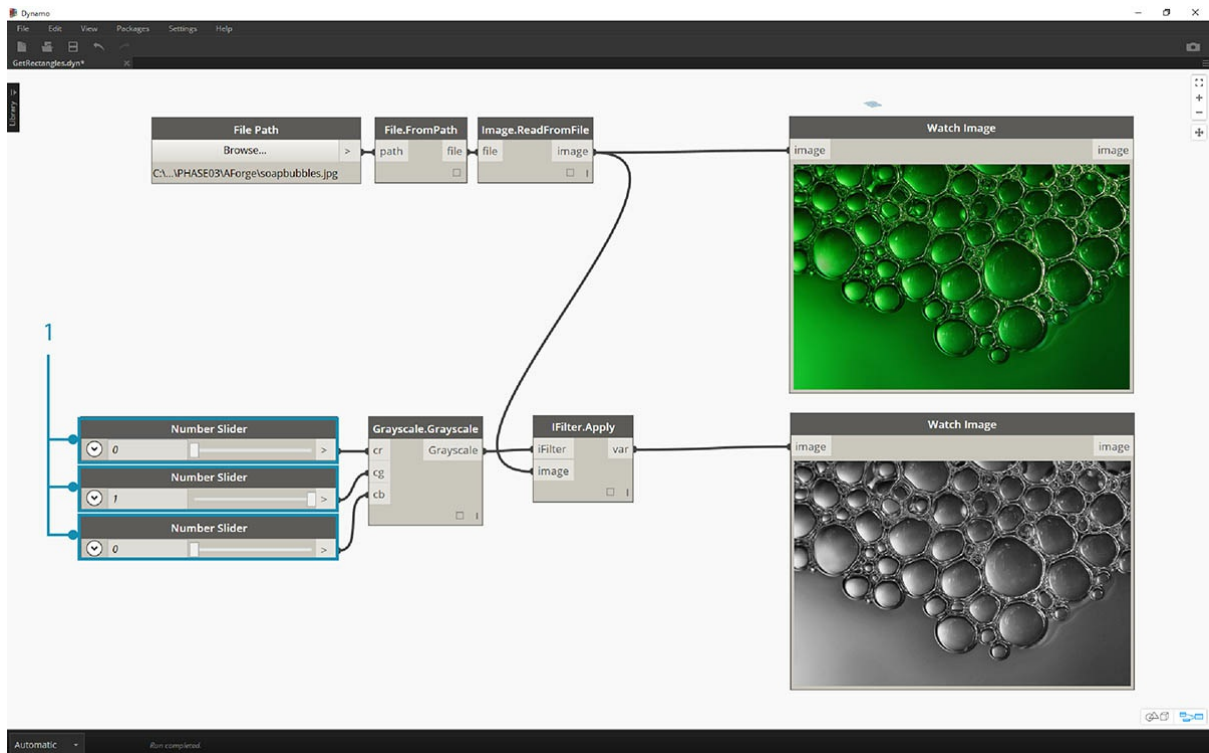
1. El nodo File Path simplemente proporciona una cadena de la ruta a la imagen que hemos seleccionado. Necesitamos convertir esta ruta de archivo en una imagen en el entorno de Dynamo.
2. Conecte el nodo File Path al nodo File.FromPath.
3. Para convertir este archivo en una imagen, utilizaremos el nodo Image.ReadFromFile.
4. Por último, veamos el resultado. Suelte un nodo Watch Image en el lienzo y conéctelo a Image.ReadFromFile. Aún no hemos utilizado

AForge, pero hemos importado correctamente una imagen en Dynamo.



En AForge.Imaging.AForge.Filters (en el menú de navegación), verá que hay una amplia gama de filtros disponibles. Vamos a utilizar uno de estos filtros para reducir la saturación de una imagen mediante los valores de umbral.

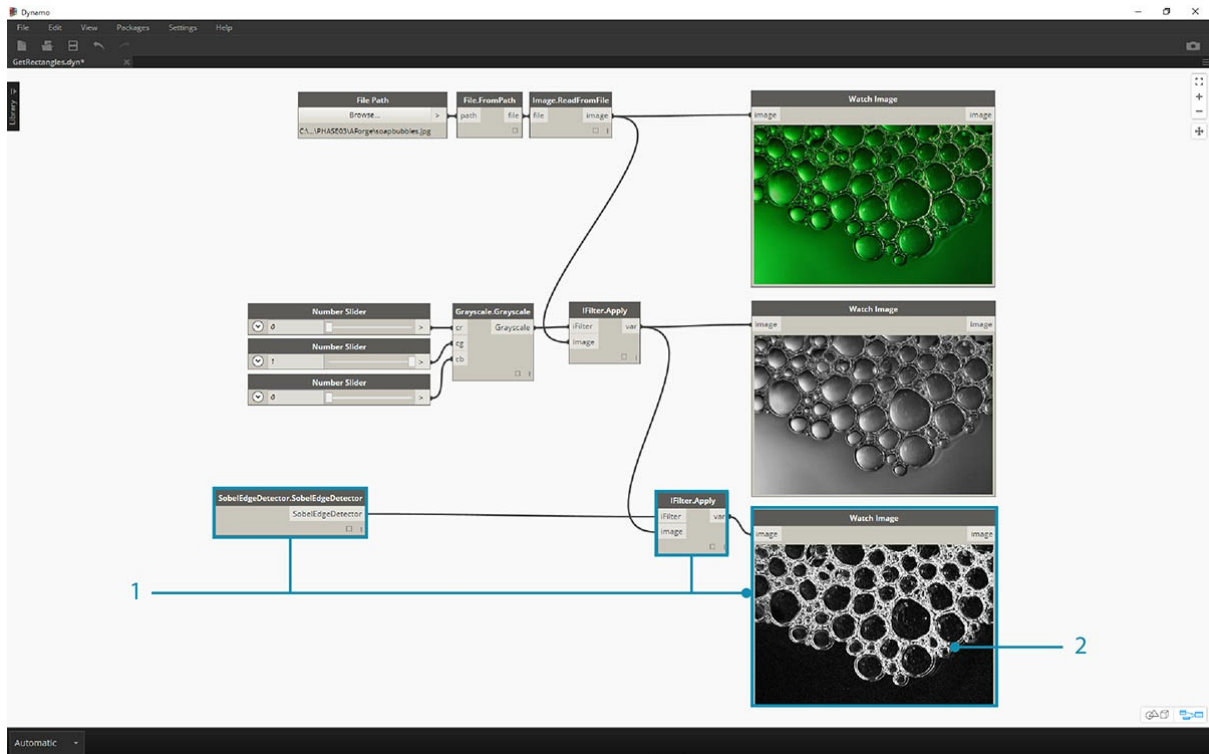
1. Coloque tres controles deslizantes en el lienzo, cambie los rangos a entre 0 a 1 y los valores de paso a 0,01.
2. Añada el nodo **Grayscale.Grayscale** al lienzo. Este es un filtro de AForge que aplica un filtro de escala de grises a una imagen. Conecte los tres controles deslizantes del paso 1 a cr, cg y cb. Cambie los controles deslizantes superior e inferior para que tengan un valor de 1 y el control deslizante central para que tenga un valor de 0.
3. Para aplicar el filtro **Grayscale**, necesitamos una acción que llevar a cabo en nuestra imagen. Para ello, utilizamos **IFilter.Apply**. Conecte la imagen a la entrada de imagen y **Grayscale.Grayscale** a la entrada de **IFilter**.
4. Al conectar un nodo **Watch Image**, se obtiene una imagen con reducción de saturación.



Podemos controlar cómo se reduce la saturación de esta imagen mediante los valores de umbral de rojo, verde y azul. Estos se definen mediante los

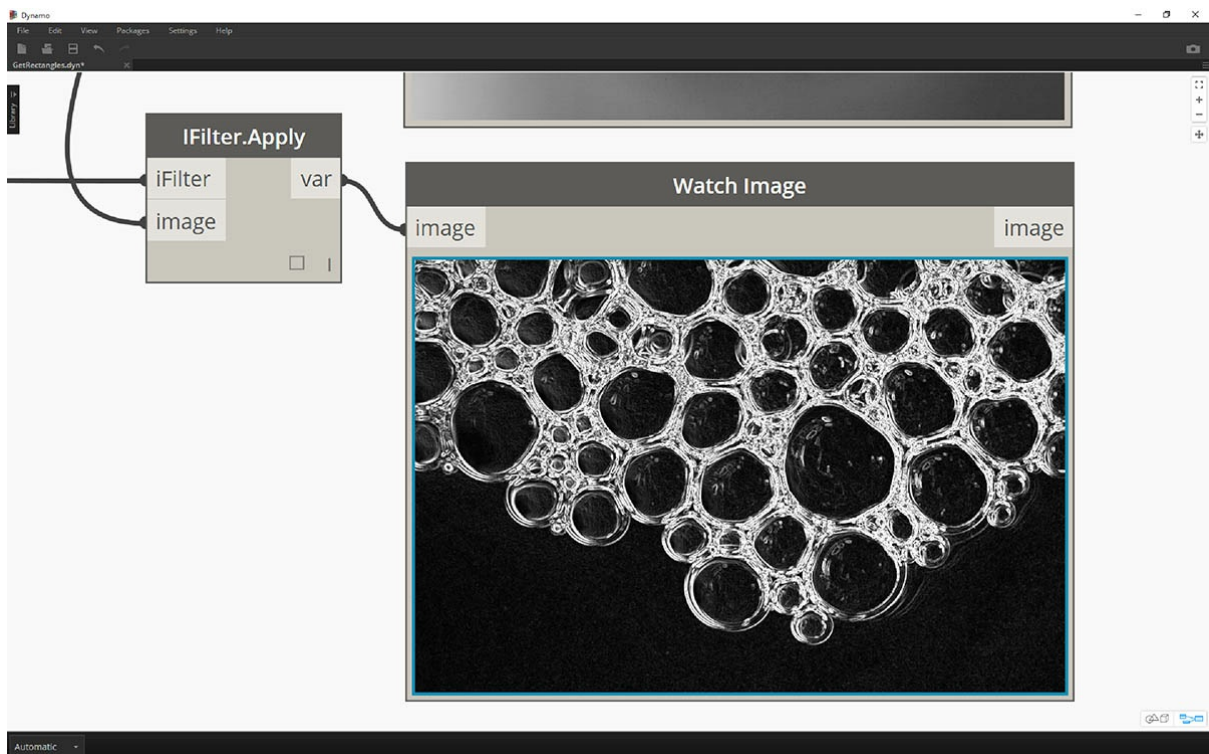
datos introducidos en el nodo Grayscale.Grayscale. Observe que la imagen se ve bastante atenuada; esto se debe a que el valor de verde se ha establecido en 0 en el control deslizante.

1. Cambie los controles deslizantes superior e inferior para que tengan un valor de 0 y el control deslizante central para que tenga un valor de 1. De este modo, obtenemos una imagen con reducción de saturación más legible.



Vamos a utilizar la imagen con reducción de saturación y aplicar otro filtro encima. La imagen con reducción de saturación tiene cierto contraste, por lo que vamos a probar una detección de borde.

1. Añada un nodo **SobelEdgeDetector.SobelEdgeDetector** al lienzo. Conecte este nodo como **IFilter** a un nuevo nodo **IFilter** y conecte la imagen con reducción de saturación a la entrada de imagen del nodo **IFilter**.
2. El detector de borde Sobel ha resaltado los bordes en una nueva imagen.



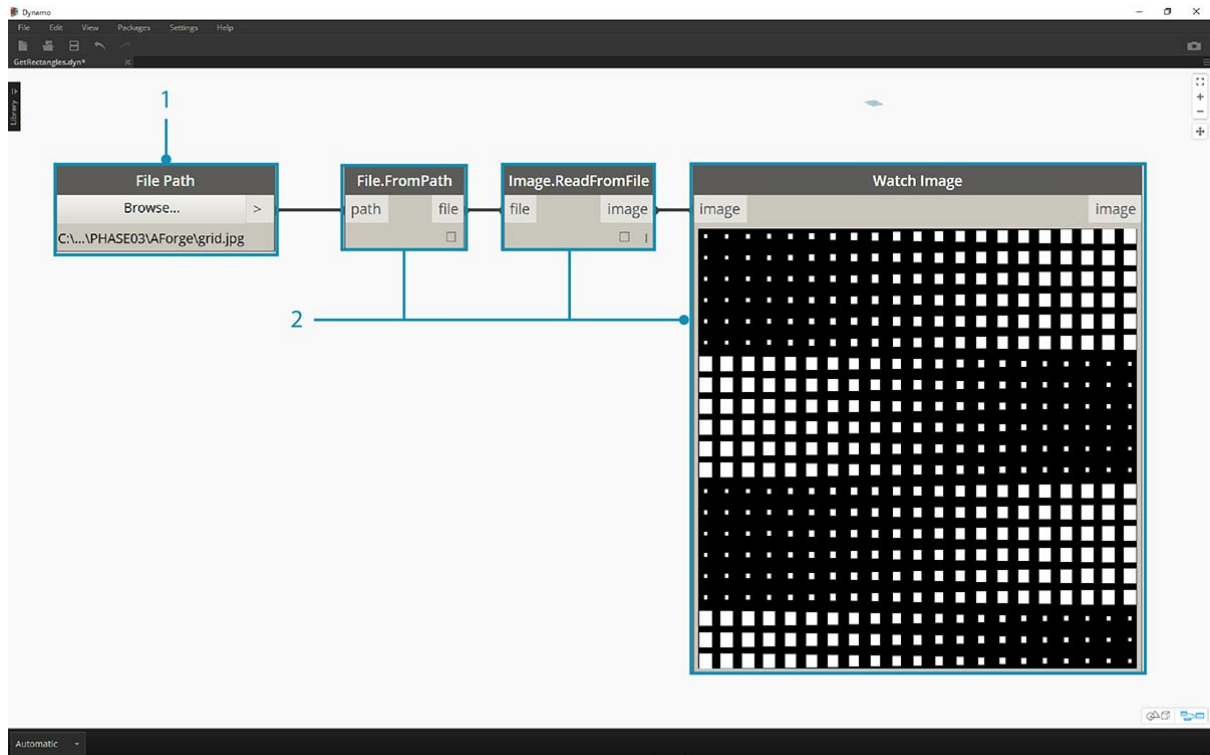
Al ampliar, el detector de bordes ha representado los contornos de las burbujas con píxeles. La biblioteca AForge dispone de herramientas para tomar

resultados como este y crear geometría de Dynamo. Lo exploraremos en el siguiente ejercicio.

Ejercicio 2: creación de rectángulos

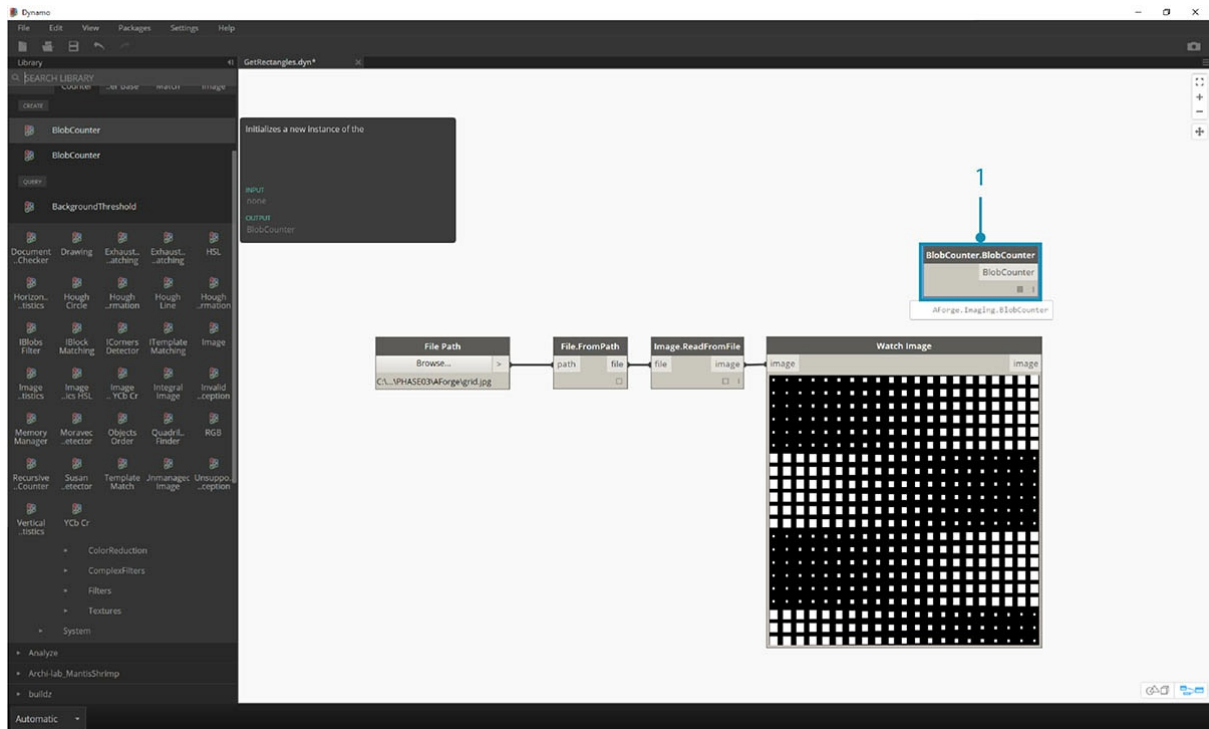
Descargue y descomprima los archivos de ejemplo que acompañan a este caso real de paquete (haga clic con el botón derecho y seleccione "Guardar enlace como..."). En el Apéndice, se incluye una lista completa de los archivos de ejemplo. [ZeroTouchImages.zip](#)

Ahora que hemos presentado un procesamiento de imagen básico, vamos a usar una imagen para aplicar la geometría de Dynamo. En un nivel elemental, en este ejercicio pretendemos realizar un "trazado directo" de una imagen mediante AForge y Dynamo. Vamos a realizar una tarea sencilla como es extraer rectángulos de una imagen de referencia, pero hay herramientas disponibles en AForge para operaciones más complejas. Trabajaremos con *02-RectangleCreation.dyn*, uno de los archivos del ejercicio que hemos descargado.

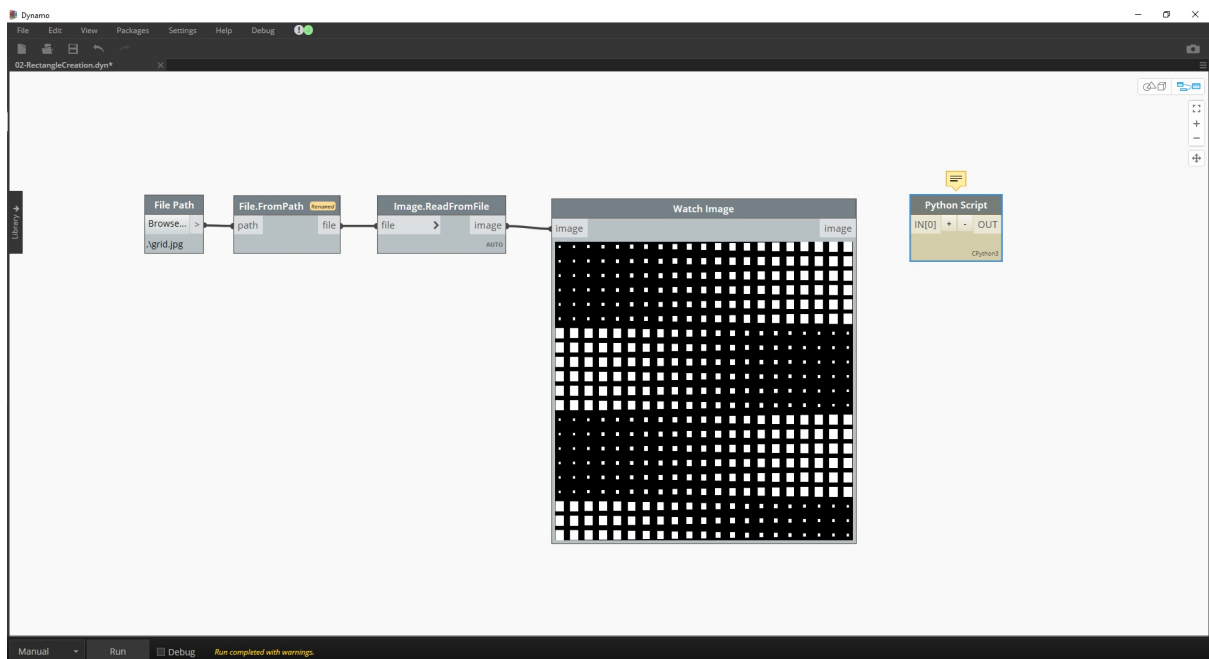


1. Con el nodo File Path, desplácese hasta grid.jpg en la carpeta del ejercicio.
2. Conecte la serie de nodos restantes anterior para que se muestre una rejilla paramétrica de rumbo.

En este paso, vamos a hacer referencia a los cuadrados blancos de la imagen y convertirlos a la geometría real de Dynamo. AForge tiene muchas herramientas eficaces de visión por ordenador y aquí vamos a usar una de las más importantes para la biblioteca llamada [BlobCounter](#).



1. Después de añadir un BlobCounter al lienzo, necesitamos un método para procesar la imagen (similar a la herramienta IFilter del ejercicio anterior). Desafortunadamente, el nodo "Process Image" no aparece de inmediato en la biblioteca de Dynamo. Esto se debe a que la función puede no ser visible en el código fuente AForge. Necesitaremos encontrar una solución para esta situación.

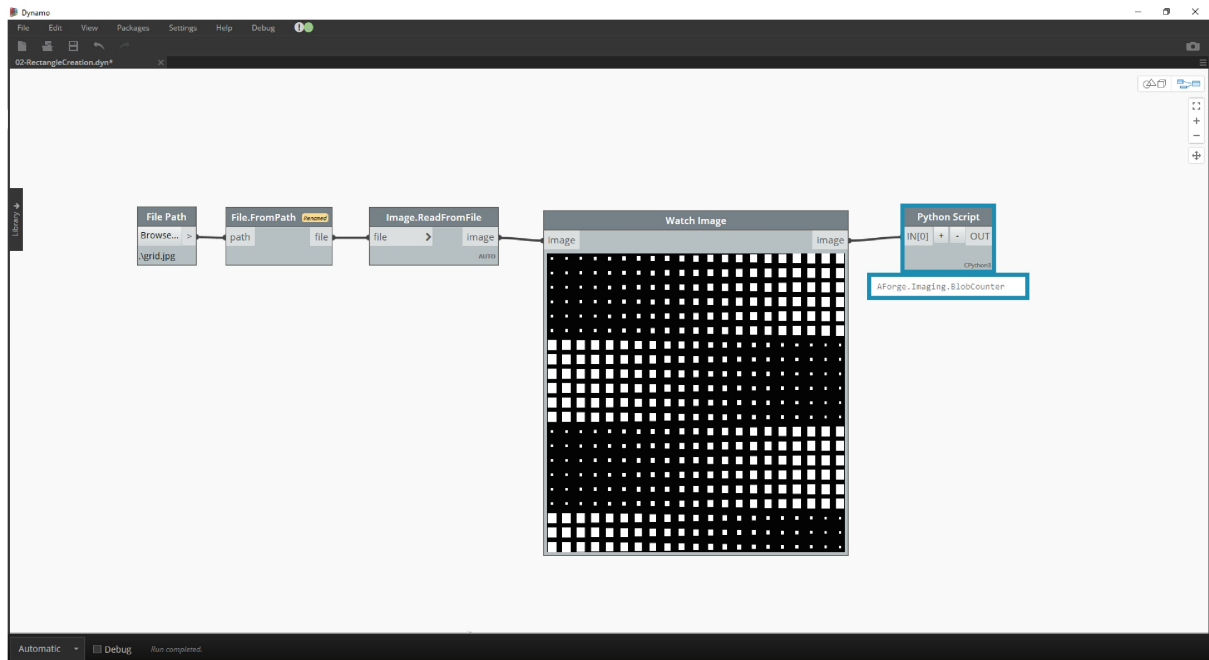


1. Añade un nodo de Python al lienzo.

```
import clr
clr.AddReference('AForge.Imaging')
from AForge.Imaging import *

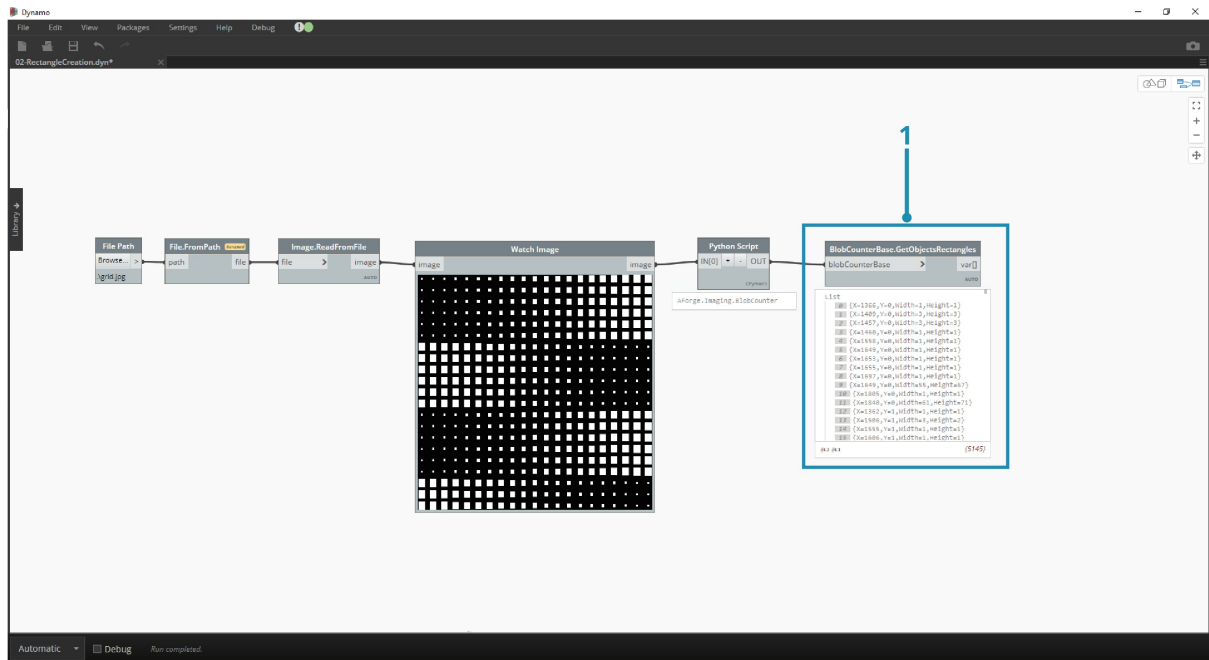
bc= BlobCounter()
bc.ProcessImage(IN[0])
OUT=bc
```

Añade el código anterior al nodo de Python. Este código importa la biblioteca AForge y, a continuación, procesa la imagen importada.

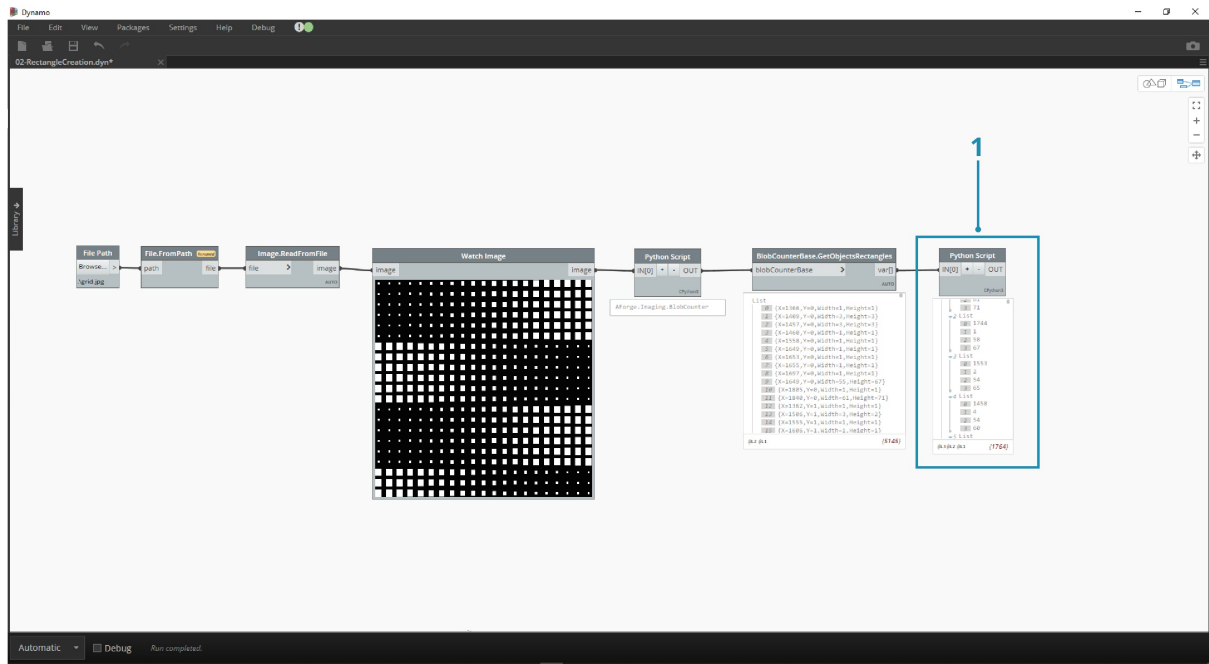


Al conectar la salida de imagen a la entrada del nodo de Python, se obtiene un resultado `AForge.Imaging.BlobCounter` del nodo de Python.

Los siguientes pasos contienen ciertos procedimientos que permiten familiarizarse con la [API de creación de imágenes AForge](#). No es necesario aprender todo esto para trabajar con Dynamo. Esta es más bien una demostración de cómo trabajar con bibliotecas externas dentro de la flexibilidad del entorno de Dynamo.

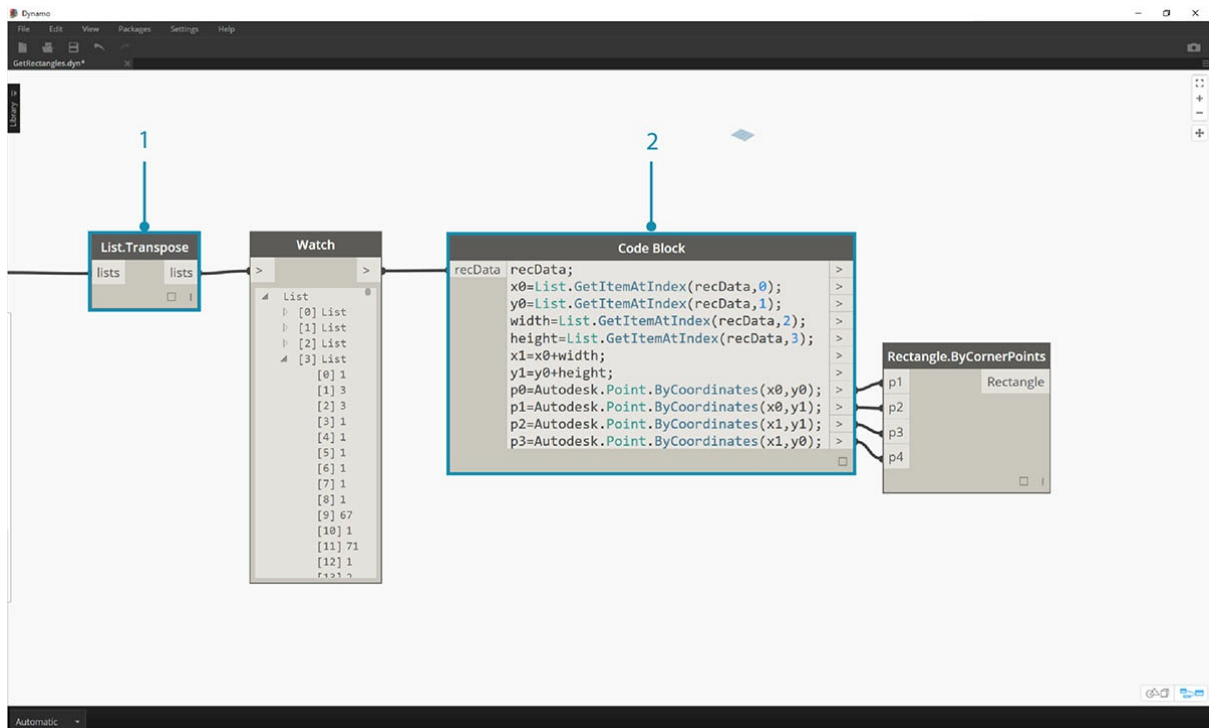


1. Conecte la salida de la secuencia de comandos de Python a BlobCounterBase.GetObjectRectangles. Este elemento lee objetos de una imagen basándose en un valor de umbral y extrae rectángulos cuantificados del espacio de píxeles.



1. Añade otro nodo de Python al lienzo, conéctelo a GetObjectRectangles e introduzca el código siguiente. De este modo, se creará una lista organizada de objetos de Dynamo.

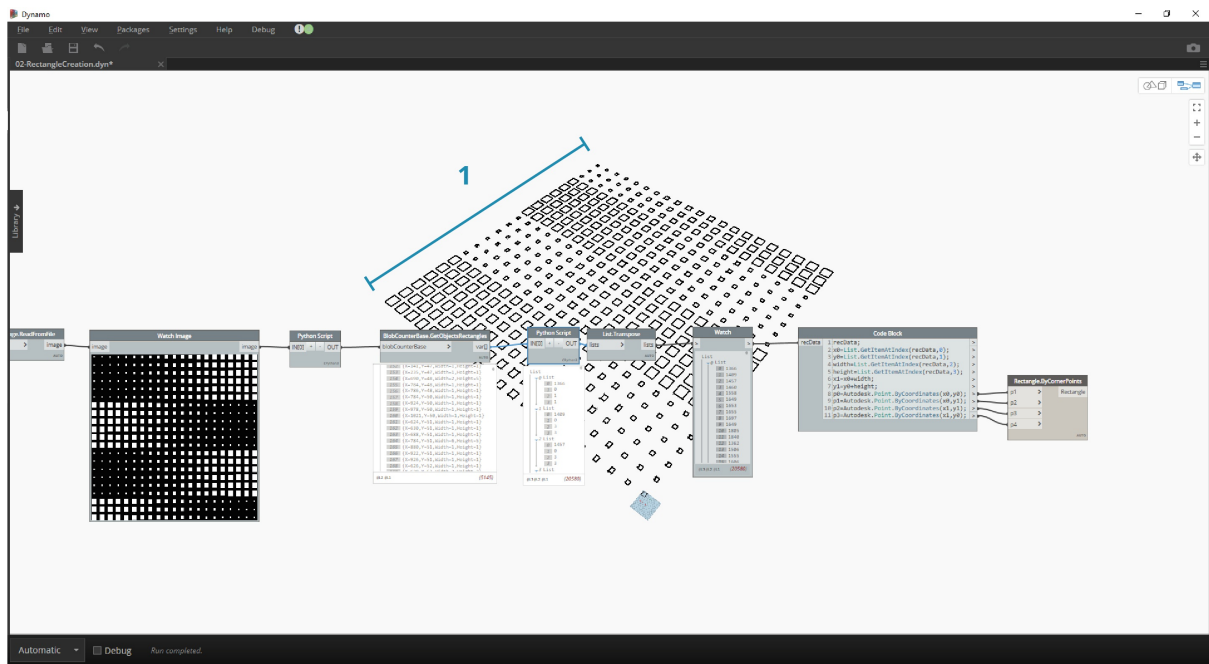
```
OUT = []
for rec in IN[0]:
    subOUT=[]
    subOUT.append(rec.X)
    subOUT.append(rec.Y)
    subOUT.append(rec.Width)
    subOUT.append(rec.Height)
    OUT.append(subOUT)
```



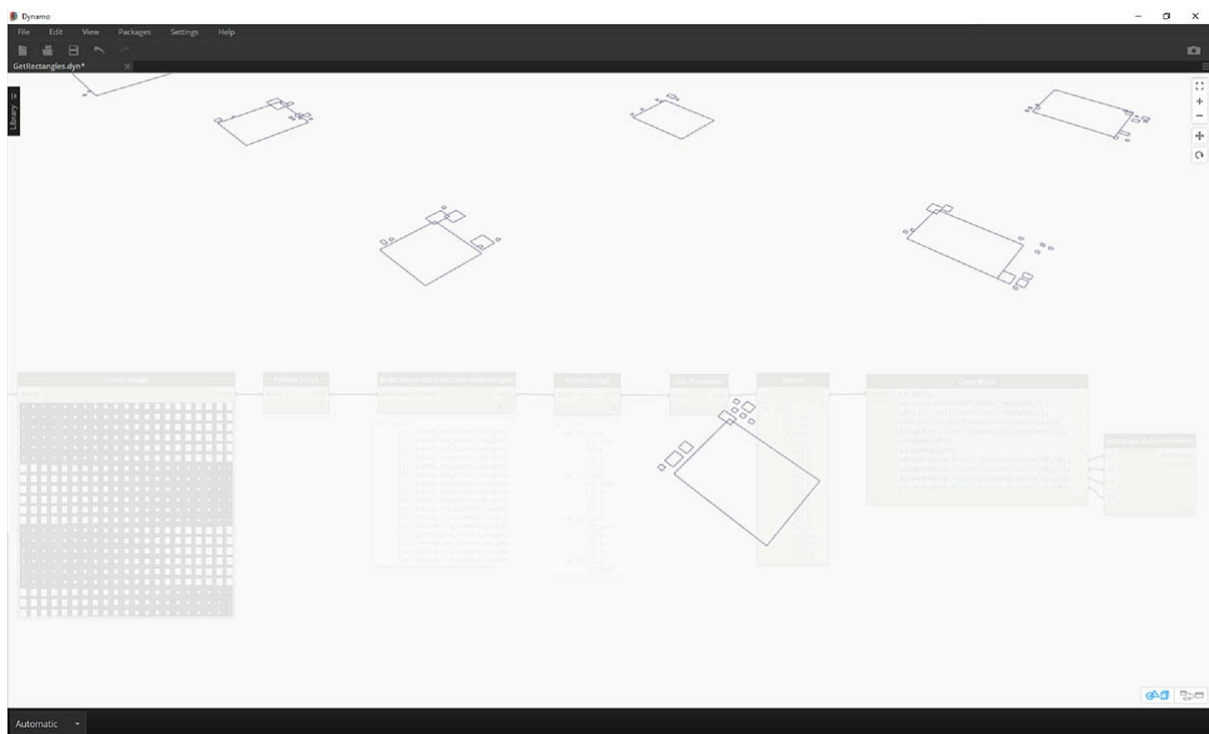
1. Transponga la salida del nodo de Python desde el paso anterior. Se crean cuatro listas, cada una de las cuales representa los valores X, Y, anchura y altura para cada rectángulo.
2. Mediante el nodo Code Block, se organizan los datos en una estructura que acompaña al nodo Rectangle.ByCornerPoints (el código siguiente).

```
recData;
x0=List.GetItemAtIndex(recData,0);
y0=List.GetItemAtIndex(recData,1);
width=List.GetItemAtIndex(recData,2);
```

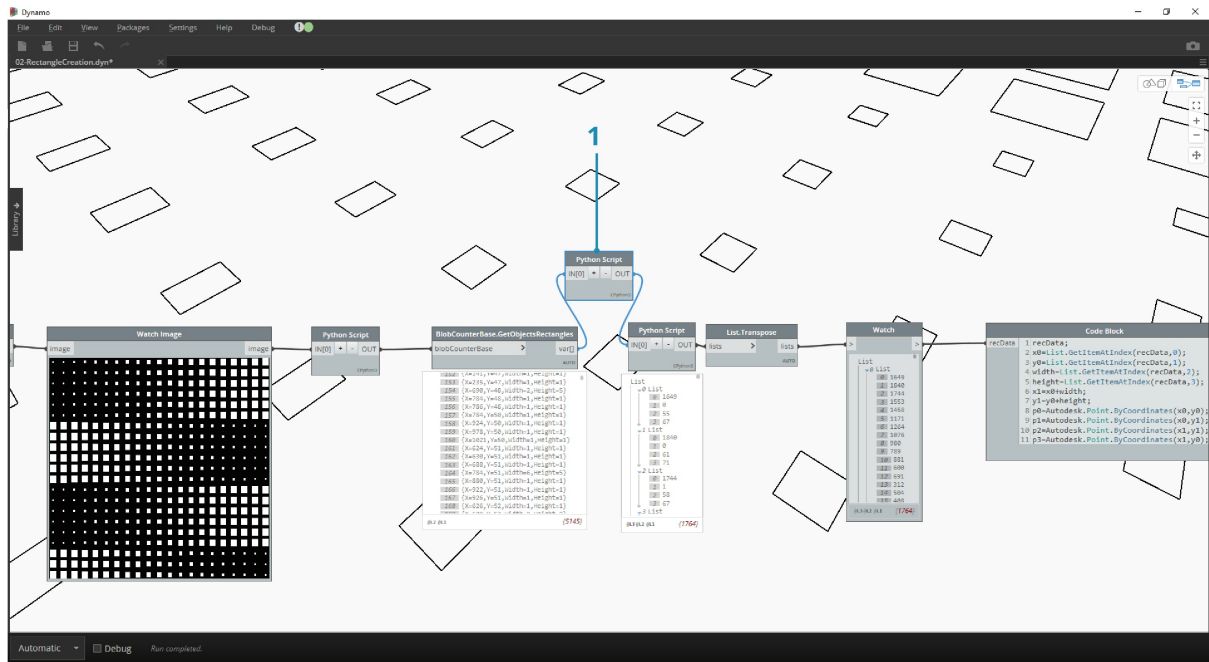
```
height=List.GetItemAtIndex(recData,3);
x1=x0+width;
y1=y0+height;
p0=Autodesk.Point.ByCoordinates(x0,y0);
p1=Autodesk.Point.ByCoordinates(x0,y1);
p2=Autodesk.Point.ByCoordinates(x1,y1);
p3=Autodesk.Point.ByCoordinates(x1,y0);
```

Al reducir la vista, disponemos de una matriz de rectángulos que representan los cuadrados blancos de la imagen. A través de la programación, hemos realizado una acción a grandes rasgos similar a un trazado directo en Illustrator.



Sin embargo, aún necesitamos realizar una limpieza. Al ampliar, podemos ver que hay un puñado de pequeños rectángulos no deseados.

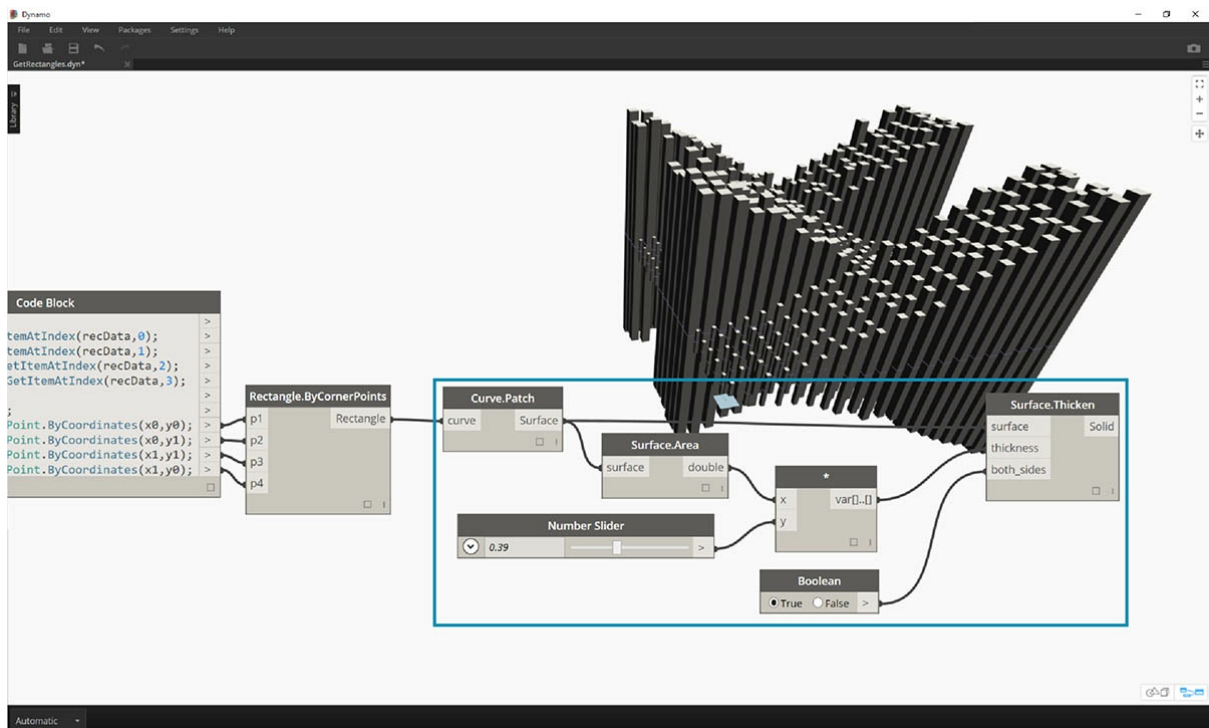


1. Los rectángulos no deseados se eliminan insertando un nodo de Python entre el nodo GetObjectRectangles y otro nodo de Python. El código del nodo es el siguiente y elimina todos los rectángulos que están por debajo de un tamaño determinado.

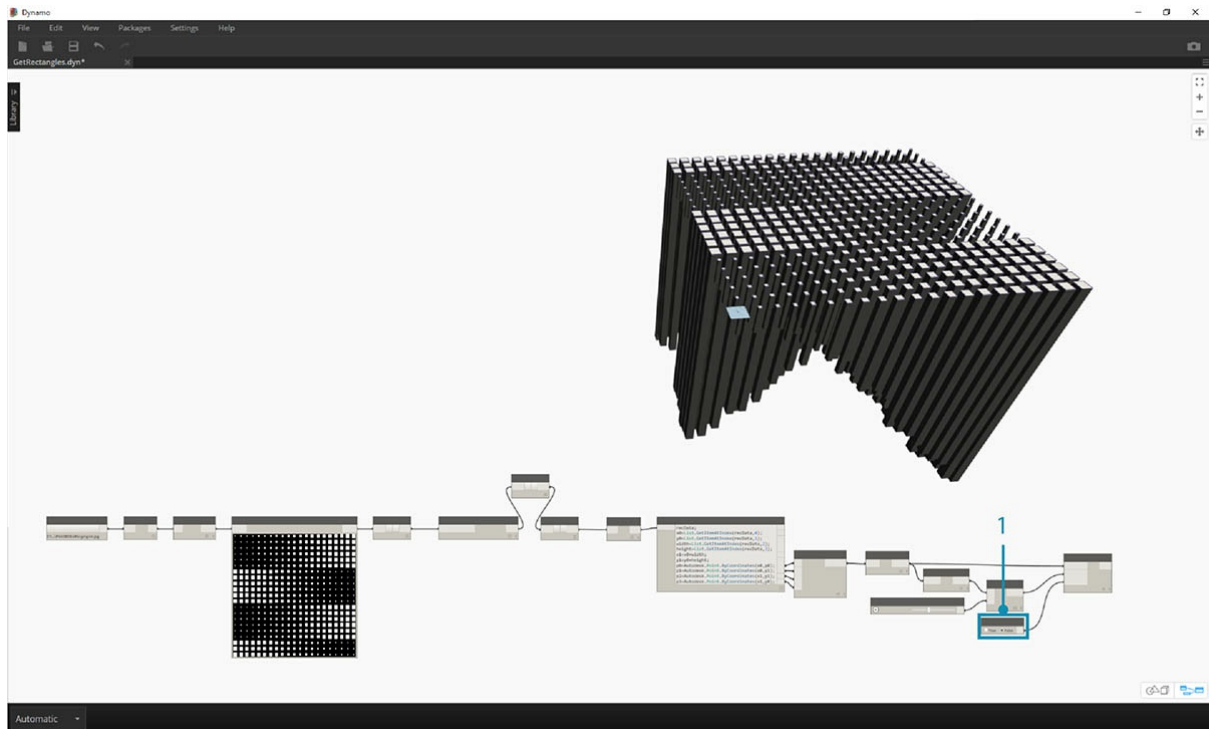
```

rectangles=IN[0]
OUT=[]
for rec in rectangles:
if rec.Width>8 and rec.Height>8:
OUT.append(rec)

```



Con los rectángulos superfluos eliminados, solo por probar, vamos a crear una superficie a partir de estos rectángulos y extruirlas una distancia basada en sus áreas.



1. Por último, cambie el valor de `both_sides` a falso (false) y se obtendrá una extrusión en una dirección. Bañe este objeto en resina y obtendrá una mesa muy peculiar.

Estos son ejemplos básicos, pero los conceptos descritos aquí son transferibles a aplicaciones apasionantes en el mundo real. La visión por ordenador se puede utilizar para multitud de procesos. Por nombrar algunos: lectores de código de barras, coincidencia de perspectivas, [mapeado de proyección](#) y [realidad aumentada](#). Para consultar temas más avanzados de AForge relacionados con este ejercicio, consulte [este artículo](#).

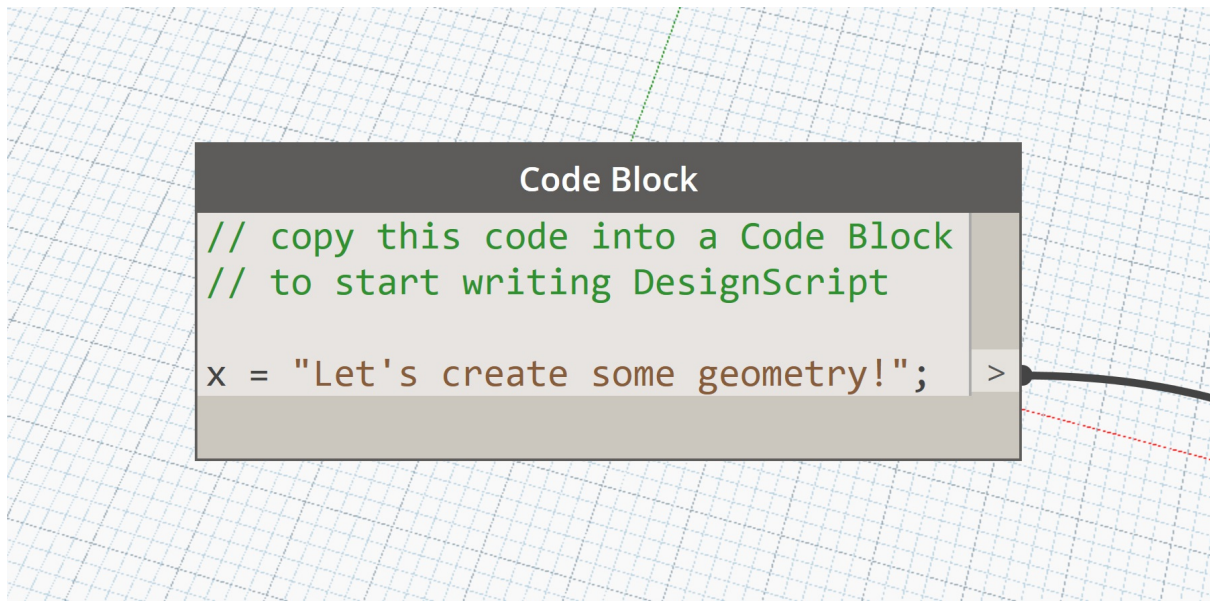
Geometría con DesignScript

Geometría con DesignScript

En esta sección, encontrará una serie de lecciones sobre la creación de geometría con DesignScript. Para continuar, copie el contenido de DesignScript de ejemplo en bloques de código de Dynamo.

```
// copy this code into a Code Block  
// to start writing DesignScript
```

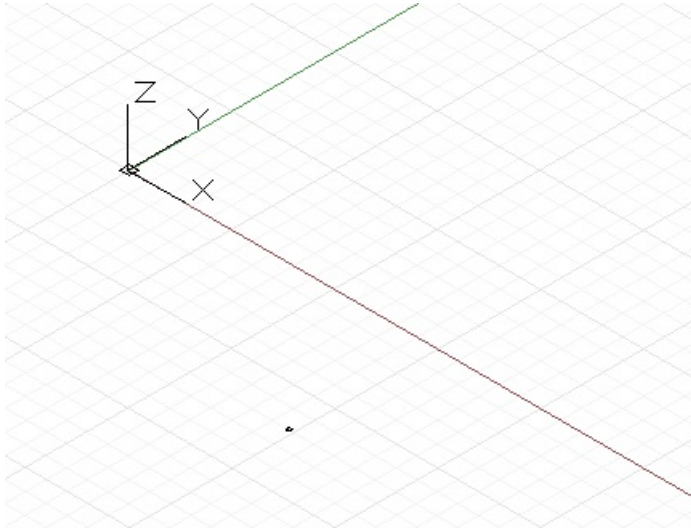
```
x = "Let's create some geometry!";
```



Conceptos básicos de geometría con DesignScript

Conceptos básicos de geometría con DesignScript

El objeto geométrico más sencillo de la biblioteca de geometría estándar de Dynamo es un punto. Toda la geometría se crea mediante funciones especiales denominadas constructores, que devuelven un nuevo ejemplar de ese tipo de geometría específico. En Dynamo, los constructores empiezan por el nombre del tipo de objeto, en este caso, "Point", seguido del método de construcción. Para crear un punto tridimensional especificado con las coordenadas cartesianas X, Y y Z, utilice el constructor *ByCoordinates*:

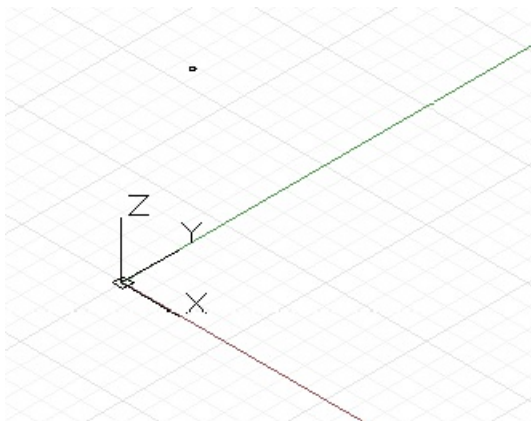


```
// create a point with the following x, y, and z
// coordinates:
x = 10;
y = 2.5;
z = -6;
```

```
p = Point.ByCoordinates(x, y, z);
```

Los constructores de Dynamo se designan normalmente con el prefijo *By* y, al invocar estas funciones, se devuelve un objeto recién creado de ese tipo. Este objeto recién creado se almacena en la variable designada a la izquierda del signo igual.

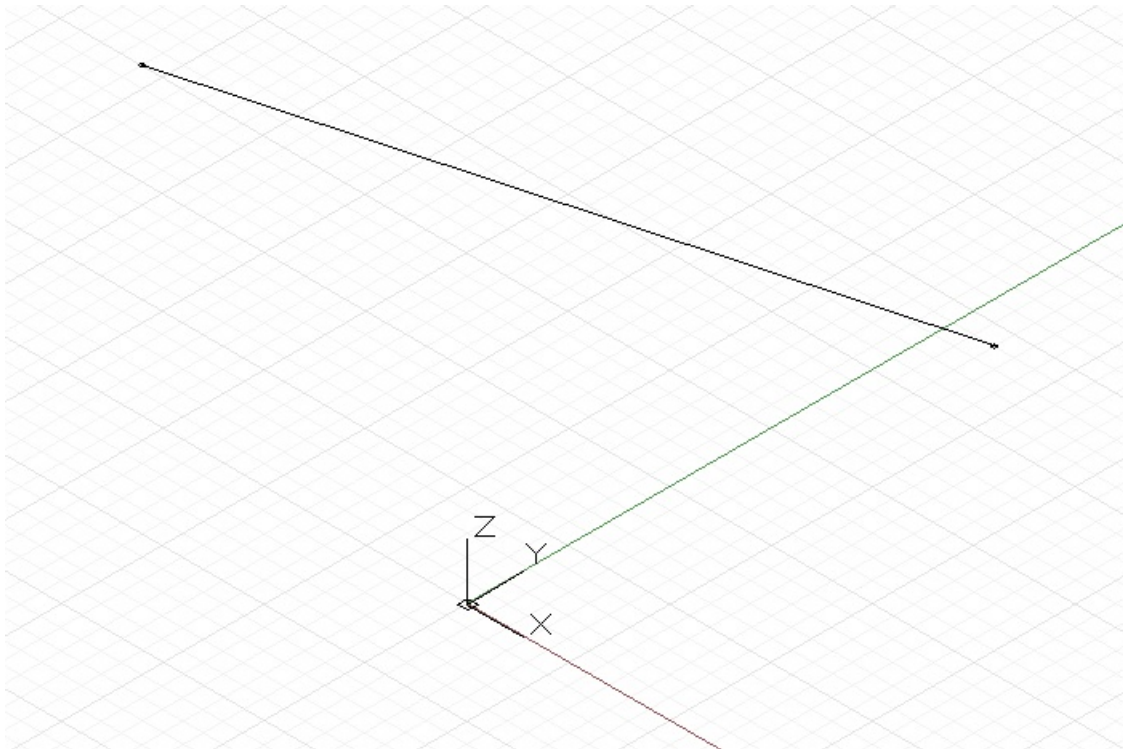
La mayoría de los objetos tienen muchos constructores distintos y se puede utilizar el constructor *BySphericalCoordinates* para crear un punto que resida en una esfera, especificado por el radio de la esfera, y un primer y un segundo ángulo de rotación (especificados en grados):



```
// create a point on a sphere with the following radius,
// theta, and phi rotation angles (specified in degrees)
radius = 5;
theta = 75.5;
phi = 120.3;
cs = CoordinateSystem.Identity();
```

```
p = Point.BySphericalCoordinates(cs, radius, theta,
    phi);
```

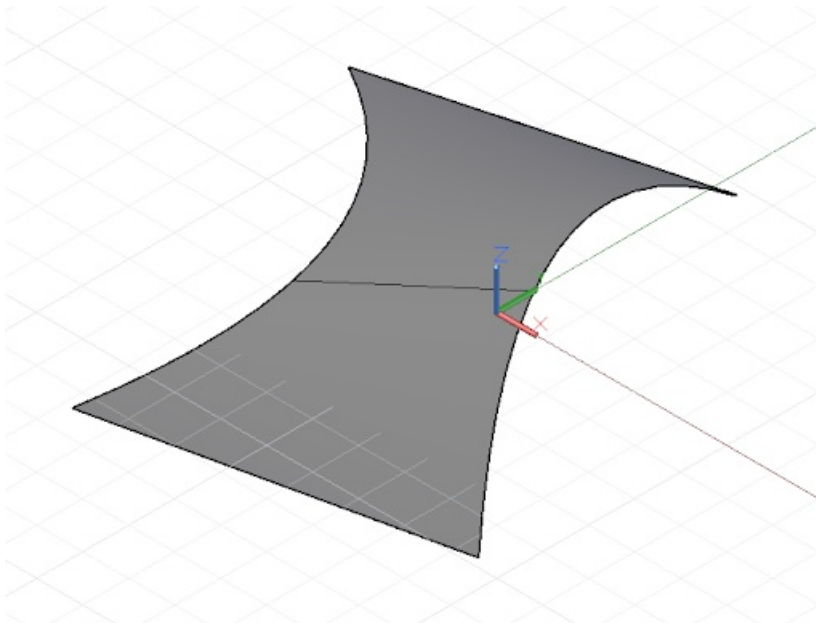
Los puntos se pueden utilizar para crear geometría dimensional superior como, por ejemplo, líneas. Se puede utilizar el constructor *ByStartPointEndPoint* para crear un objeto de línea entre dos puntos:



```
// create two points:
p1 = Point.ByCoordinates(3, 10, 2);
p2 = Point.ByCoordinates(-15, 7, 0.5);

// construct a line between p1 and p2
l = Line.ByStartPointEndPoint(p1, p2);
```

De forma similar, las líneas se pueden utilizar para crear geometría de superficie dimensional superior, por ejemplo, mediante el constructor *Loft*, que utiliza una serie de líneas o curvas e interpola una superficie entre ellas.



```
// create points:
p1 = Point.ByCoordinates(3, 10, 2);
p2 = Point.ByCoordinates(-15, 7, 0.5);

p3 = Point.ByCoordinates(5, -3, 5);
p4 = Point.ByCoordinates(-5, -6, 2);

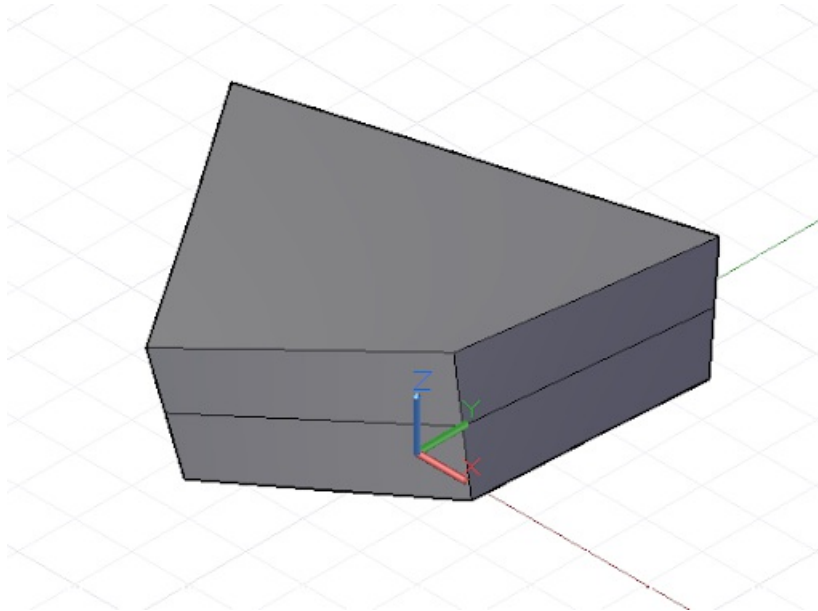
p5 = Point.ByCoordinates(9, -10, -2);
p6 = Point.ByCoordinates(-11, -12, -4);

// create lines:
```

```
l1 = Line.ByStartPointEndPoint(p1, p2);
l2 = Line.ByStartPointEndPoint(p3, p4);
l3 = Line.ByStartPointEndPoint(p5, p6);
```

```
// loft between cross section lines:
surf = Surface.ByLoft([l1, l2, l3]);
```

Las superficies también se pueden utilizar para crear geometría sólida dimensional superior, por ejemplo, mediante el engrosado de la superficie a una distancia especificada. Muchos objetos tienen funciones enlazadas a ellos denominadas métodos, que permiten al programador ejecutar comandos en ese objeto específico. Entre los métodos comunes a todas las partes de geometría, se incluyen *Translate* y *Rotate*, que trasladan (desplazan) y rotan la geometría una cantidad especificada. Las superficies presentan el método *Thicken*, que utiliza una única entrada, un número que especifica el nuevo grosor de la superficie.



```
p1 = Point.ByCoordinates(3, 10, 2);
p2 = Point.ByCoordinates(-15, 7, 0.5);

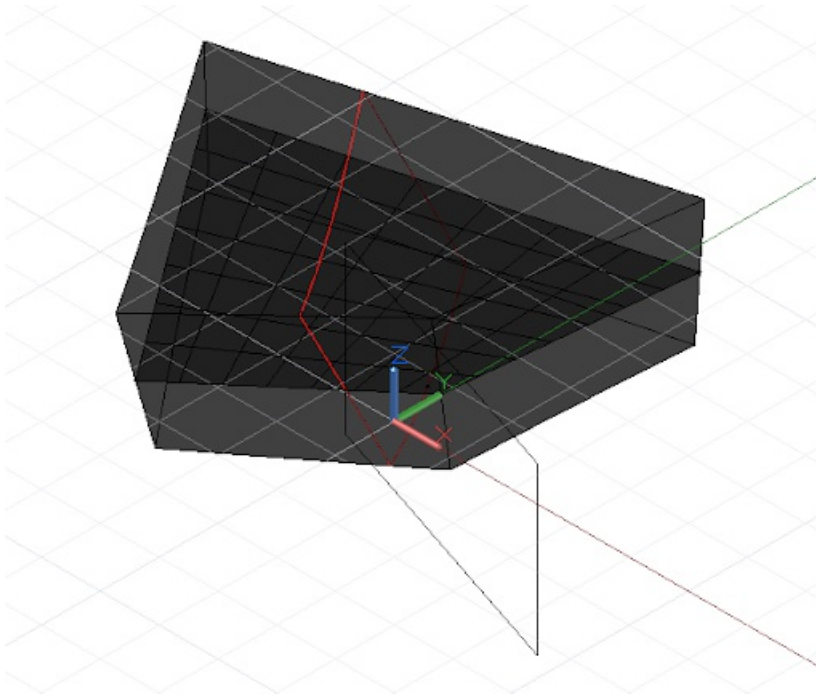
p3 = Point.ByCoordinates(5, -3, 5);
p4 = Point.ByCoordinates(-5, -6, 2);
```

```
l1 = Line.ByStartPointEndPoint(p1, p2);
l2 = Line.ByStartPointEndPoint(p3, p4);
```

```
surf = Surface.ByLoft([l1, l2]);
```

```
// true indicates to thicken both sides of the Surface:
solid = surf.Thicken(4.75, true);
```

Los comandos de *intersección* pueden extraer geometría dimensional inferior de objetos dimensionales superiores. Esta geometría dimensional inferior extraída puede formar la base para una geometría dimensional superior en un proceso cíclico de creación, extracción y recreación. En este ejemplo, se utiliza el sólido generado para crear una superficie y se utiliza la superficie para crear una curva.



```

p1 = Point.ByCoordinates(3, 10, 2);
p2 = Point.ByCoordinates(-15, 7, 0.5);

p3 = Point.ByCoordinates(5, -3, 5);
p4 = Point.ByCoordinates(-5, -6, 2);

l1 = Line.ByStartPointEndPoint(p1, p2);
l2 = Line.ByStartPointEndPoint(p3, p4);

surf = Surface.ByLoft([l1, l2]);

solid = surf.Thicken(4.75, true);

p = Plane.ByOriginNormal(Point.ByCoordinates(2, 0, 0),
    Vector.ByCoordinates(1, 1, 1));

int_surf = solid.Intersect(p);

int_line = int_surf.Intersect(Plane.ByOriginNormal(
    Point.ByCoordinates(0, 0, 0),
    Vector.ByCoordinates(1, 0, 0)));

```

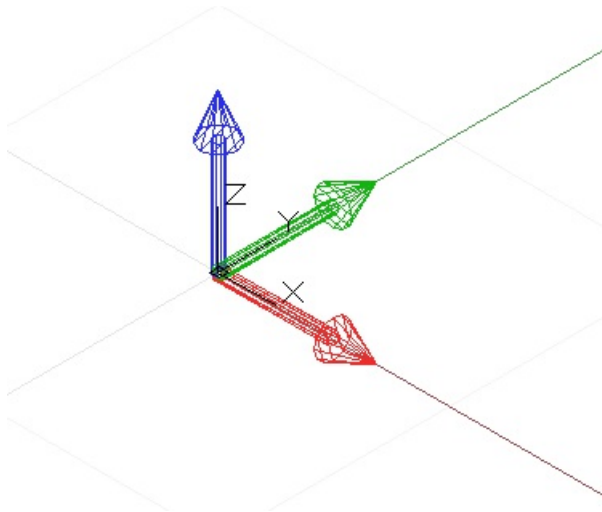
Primitivas geométricas

Primitivas geométricas

Aunque Dynamo puede crear diversas formas geométricas complejas, las primitivas geométricas simples forman la espina dorsal de cualquier diseño computacional: expresadas directamente en la forma diseñada final o utilizadas como andamiaje a partir del que se genera geometría más compleja.

Aunque no es estrictamente una parte de la geometría, `CoordinateSystem` es una herramienta importante para crear geometría. Un objeto `CoordinateSystem` realiza un seguimiento de las transformaciones geométricas y de posición, como la rotación, el corte y la escala.

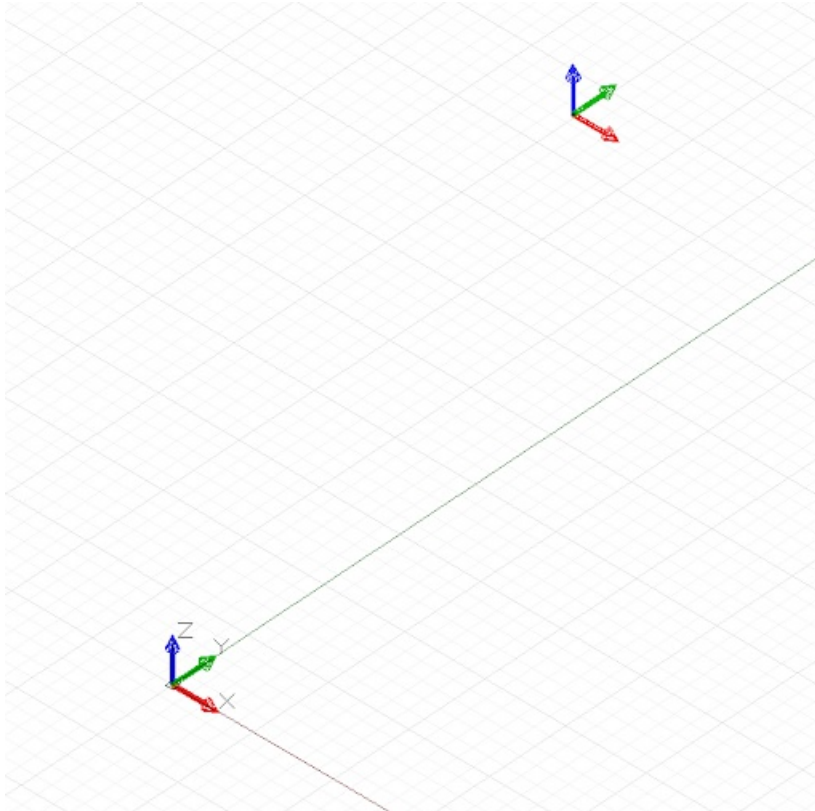
Para crear un `CoordinateSystem` centrado en un punto con $x = 0$, $y = 0$ y $z = 0$ sin transformaciones de rotación, escala o corte, basta con llamar al constructor `Identity`:



```
// create a CoordinateSystem at x = 0, y = 0, z = 0,  
// no rotations, scaling, or sheering transformations
```

```
cs = CoordinateSystem.Identity();
```

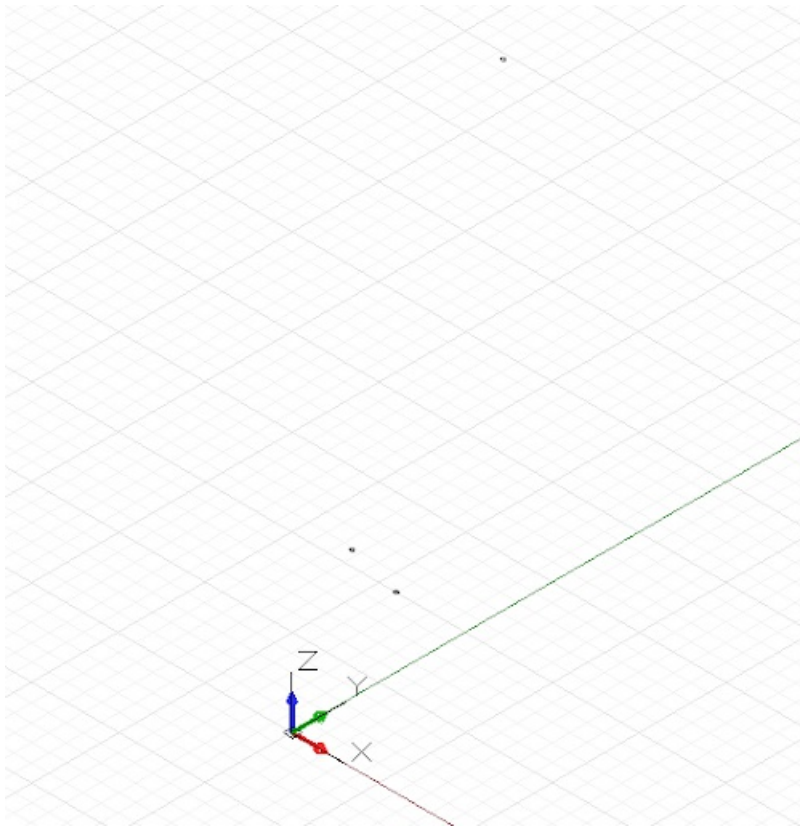
Los objetos `CoordinateSystem` con transformaciones geométricas se encuentran fuera del ámbito de este capítulo, aunque otro constructor permite crear un sistema de coordenadas en un punto específico, `CoordinateSystem.ByOriginVectors`:



```
// create a CoordinateSystem at a specific location,  
// no rotations, scaling, or sheering transformations  
x_pos = 3.6;  
y_pos = 9.4;  
z_pos = 13.0;  
  
origin = Point.ByCoordinates(x_pos, y_pos, z_pos);  
identity = CoordinateSystem.Identity();  
  
cs = CoordinateSystem.ByOriginVectors(origin,  
    identity.XAxis, identity.YAxis, identity.ZAxis);
```

La primitiva geométrica más sencilla es un punto, que representa una ubicación de cero dimensiones en un espacio tridimensional. Como se ha mencionado anteriormente, existen varias formas diferentes de crear un punto en un sistema de coordenadas específico: *Point.ByCoordinates* crea un punto con las coordenadas X, Y y Z especificadas; *Point.ByCartesianCoordinates* crea un punto con las coordenadas X, Y y Z especificadas en un determinado sistema de coordenadas; *Point.ByCylindricalCoordinates* crea un punto en un cilindro con un radio, un ángulo de rotación y una altura, y *Point.BySphericalCoordinates* crea un punto en una esfera con un radio y dos ángulos de rotación.

En este ejemplo, se muestran puntos creados en varios sistemas de coordenadas:



```
// create a point with x, y, and z coordinates
x_pos = 1;
y_pos = 2;
z_pos = 3;

pCoord = Point.ByCoordinates(x_pos, y_pos, z_pos);

// create a point in a specific coordinate system
cs = CoordinateSystem.Identity();
pCoordSystem = Point.ByCartesianCoordinates(cs, x_pos,
    y_pos, z_pos);

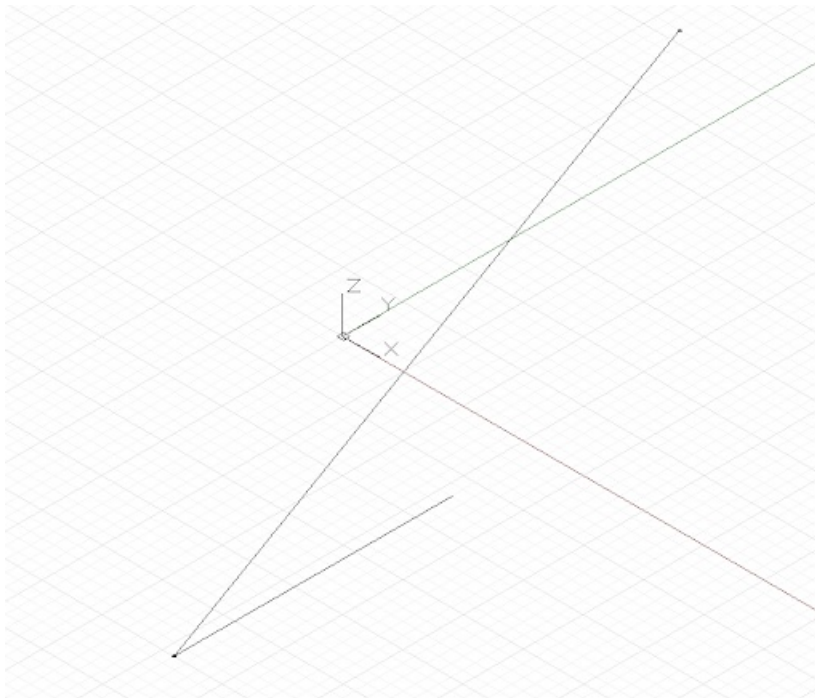
// create a point on a cylinder with the following
// radius and height
radius = 5;
height = 15;
theta = 75.5;

pCyl = Point.ByCylindricalCoordinates(cs, radius, theta,
    height);

// create a point on a sphere with radius and two angles
phi = 120.3;

pSphere = Point.BySphericalCoordinates(cs, radius,
    theta, phi);
```

La siguiente primitiva de Dynamo dimensional superior es un segmento de línea, que representa un número infinito de puntos entre dos puntos finales. Las líneas se pueden crear indicando explícitamente los dos puntos de contorno con el constructor *Line.ByStartPointEndPoint* o especificando un punto inicial, una dirección y una longitud en esa dirección, *Line.ByStartPointDirectionLength*.

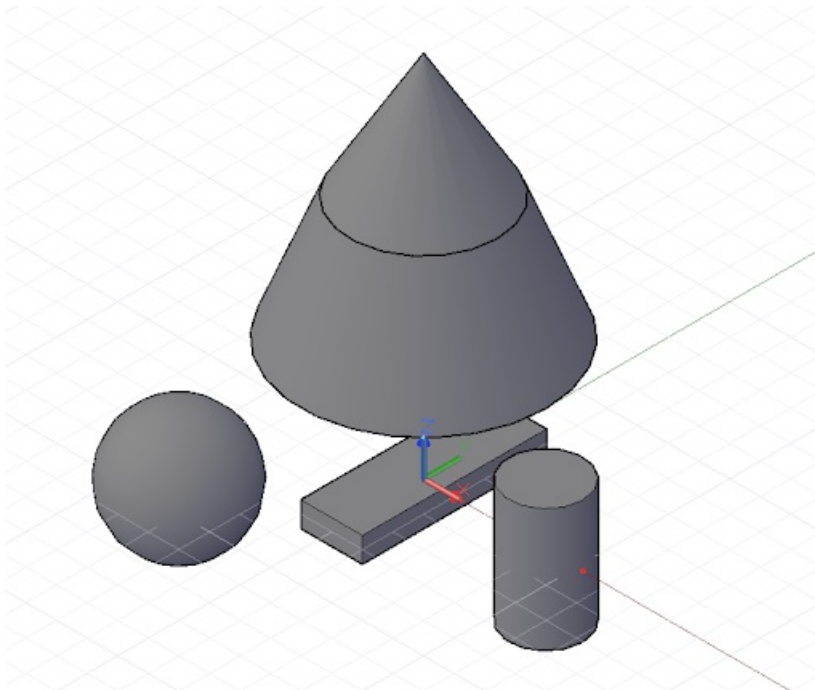


```
p1 = Point.ByCoordinates(-2, -5, -10);
p2 = Point.ByCoordinates(6, 8, 10);

// a line segment between two points
l2pts = Line.ByStartPointEndPoint(p1, p2);

// a line segment at p1 in direction 1, 1, 1 with
// length 10
lDir = Line.ByStartPointDirectionLength(p1,
    Vector.ByCoordinates(1, 1, 1), 10);
```

Dynamo tiene objetos que representan los tipos más básicos de primitivas geométricas en tres dimensiones: cubos, creados con *Cuboid.ByLengths*; conos, creados con *Cone.ByPointsRadius* y *Cone.ByPointsRadii*; cilindros, creados con *Cylinder.ByRadiusHeight*, y esferas, creadas con *Sphere.ByCenterPointRadius*.



```
// create a cuboid with specified lengths
cs = CoordinateSystem.Identity();

cub = Cuboid.ByLengths(cs, 5, 15, 2);
```

```
// create several cones
p1 = Point.ByCoordinates(0, 0, 10);
p2 = Point.ByCoordinates(0, 0, 20);
p3 = Point.ByCoordinates(0, 0, 30);

cone1 = Cone.ByPointsRadii(p1, p2, 10, 6);
cone2 = Cone.ByPointsRadii(p2, p3, 6, 0);

// make a cylinder
cylCS = cs.Translate(10, 0, 0);

cyl = Cylinder.ByRadiusHeight(cylCS, 3, 10);

// make a sphere
centerP = Point.ByCoordinates(-10, -10, 0);

sph = Sphere.ByCenterPointRadius(centerP, 5);
```

Matemáticas vectoriales

Matemáticas vectoriales

Los objetos de los diseños computacionales rara vez se crean explícitamente en su posición y forma finales y, con mayor frecuencia, se convierten, se giran y se colocan de cualquier otro modo a partir de la geometría existente. Las matemáticas vectoriales sirven como una especie de andamiaje geométrico para proporcionar dirección y orientación a la geometría, así como para conceptualizar movimientos a través del espacio 3D sin representación visual.

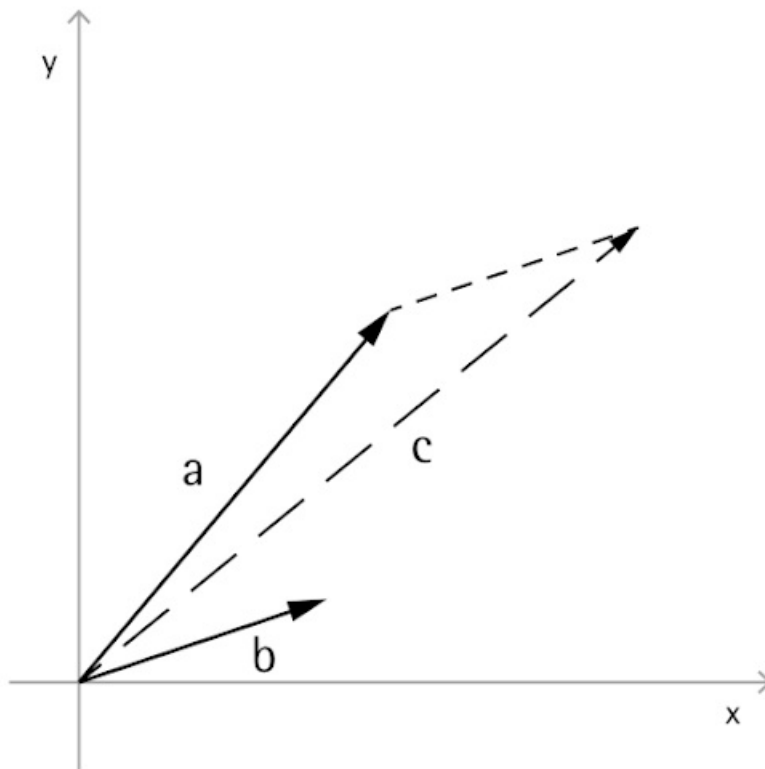
Como elemento más básico, un vector representa una posición en el espacio 3D y a menudo se considera el punto final de una flecha desde la posición (0, 0, 0) hasta esa posición. Los vectores se pueden crear con el constructor *ByCoordinates*, utilizando la posición X, Y y Z del objeto vectorial que se acaba de crear. Observe que los objetos vectoriales no son objetos geométricos y no aparecen en la ventana de Dynamo. Sin embargo, la información sobre un vector recién creado o modificado se puede imprimir en la ventana de la consola:



```
// construct a Vector object  
v = Vector.ByCoordinates(1, 2, 3);  
  
s = v.X + " " + v.Y + " " + v.Z;
```

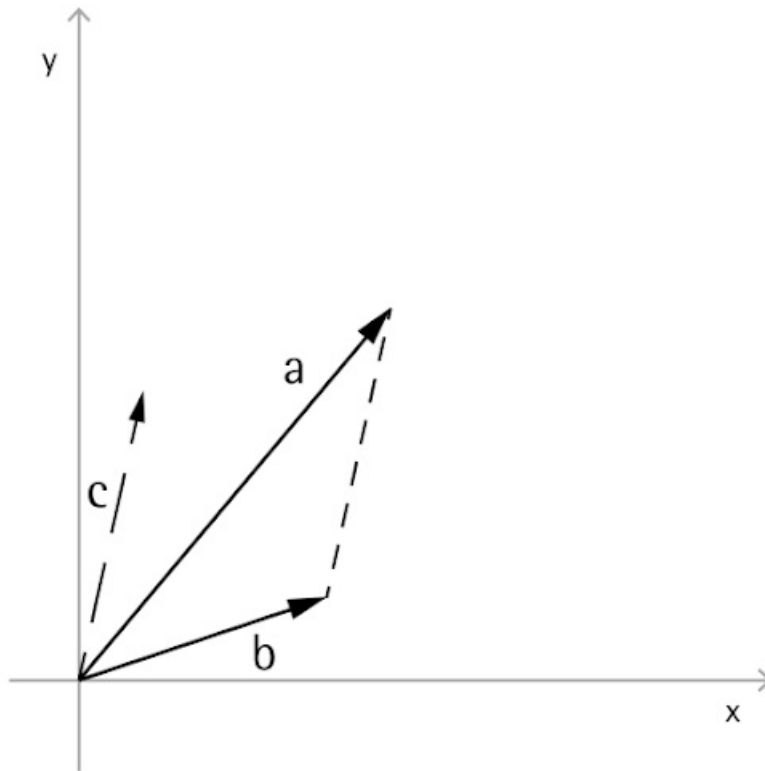
Un conjunto de operaciones matemáticas se define en objetos vectoriales, lo que permite sumar, restar, multiplicar y, en general, desplazar objetos en un espacio 3D, tal y como desplazaría números reales en un espacio unidimensional en una línea numérica.

La adición de vectores se define como la suma de los componentes de dos vectores y se puede entender como el vector resultante si las dos flechas de vector de componente se colocan "de la punta a la cola". La adición de vectores se realiza con el método *Add* y se representa con el diagrama de la izquierda.



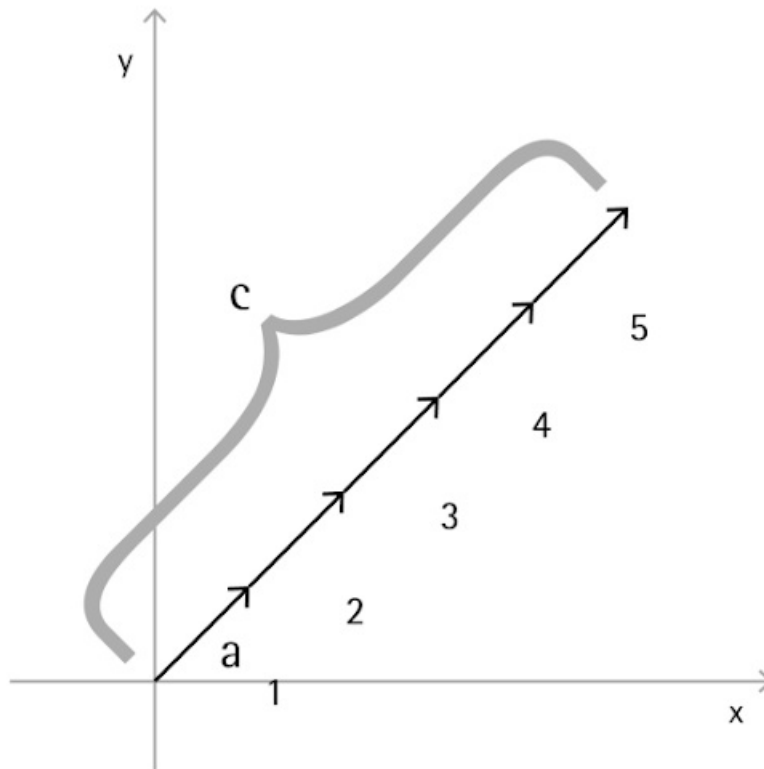
```
a = Vector.ByCoordinates(5, 5, 0);  
b = Vector.ByCoordinates(4, 1, 0);  
  
// c has value x = 9, y = 6, z = 0  
c = a.Add(b);
```

Del mismo modo, se pueden restar dos objetos vectoriales entre sí mediante el método *Subtract*. La resta de vectores se puede entender como la dirección que va desde el primer vector hasta el segundo.



```
a = Vector.ByCoordinates(5, 5, 0);  
b = Vector.ByCoordinates(4, 1, 0);  
  
// c has value x = 1, y = 4, z = 0  
c = a.Subtract(b);
```

La multiplicación de vectores se puede entender como el desplazamiento del punto final de un vector en su propia dirección mediante un factor de escala especificado.



```
a = Vector.ByCoordinates(4, 4, 0);

// c has value x = 20, y = 20, z = 0
c = a.Scale(5);
```

A menudo, al ajustar la escala de un vector, es deseable que su longitud sea exactamente igual a la cantidad de escala ajustada. Esto se puede conseguir fácilmente. Para ello, debe normalizarse primero un vector, en otras palabras, se debe establecer la longitud del vector en un valor exactamente igual a uno.

Code Block

```
a = Vector.ByCoordinates(1, 2, 3);
a_len = a.Length;

// set the a's length equal to 1.0
b = a.Normalized();
c = b.Scale(5);

// len is equal to 5
len = c.Length;
```

Watch

3.742

Watch

5.000

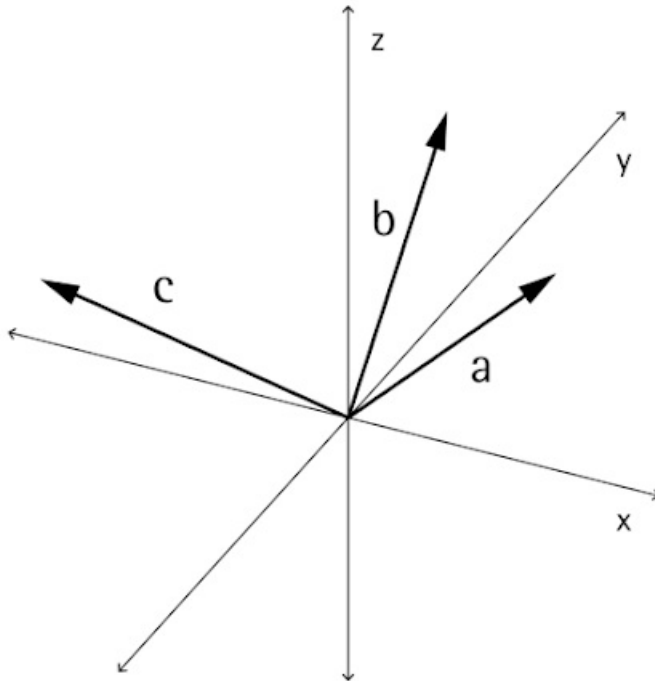
```
a = Vector.ByCoordinates(1, 2, 3);
a_len = a.Length;

// set the a's length equal to 1.0
b = a.Normalized();
c = b.Scale(5);

// len is equal to 5
len = c.Length;
```

c sigue apuntando en la misma dirección que a (1, 2, 3), aunque ahora tiene una longitud exactamente igual a 5.

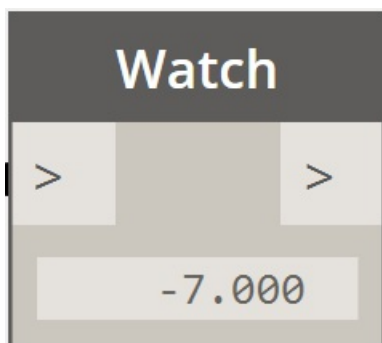
Existen dos métodos adicionales en las matemáticas vectoriales que no tienen equivalentes claros con las matemáticas unidimensionales: el producto vectorial y el producto escalar. El producto vectorial permite generar un vector ortogonal (a 90 grados) a dos vectores existentes. Por ejemplo, el producto vectorial de los ejes X e Y es el eje Z, aunque los dos vectores de entrada no tienen que ser ortogonales entre sí. Un vector de producto vectorial se calcula con el método *Cross*.



```
a = Vector.ByCoordinates(1, 0, 1);
b = Vector.ByCoordinates(0, 1, 1);

// c has value x = -1, y = -1, z = 1
c = a.Cross(b);
```

Una función adicional, aunque algo más avanzada, de matemáticas vectoriales es el producto escalar. El producto escalar entre dos vectores es un número real (no un objeto vectorial) que hace referencia al ángulo entre dos vectores, aunque no exactamente. Una de las propiedades útiles de este tipo de producto es que el producto escalar entre dos vectores será 0 solo si son perpendiculares. El producto escalar se calcula con el método *Dot*.



```
a = Vector.ByCoordinates(1, 2, 1);
b = Vector.ByCoordinates(5, -8, 4);

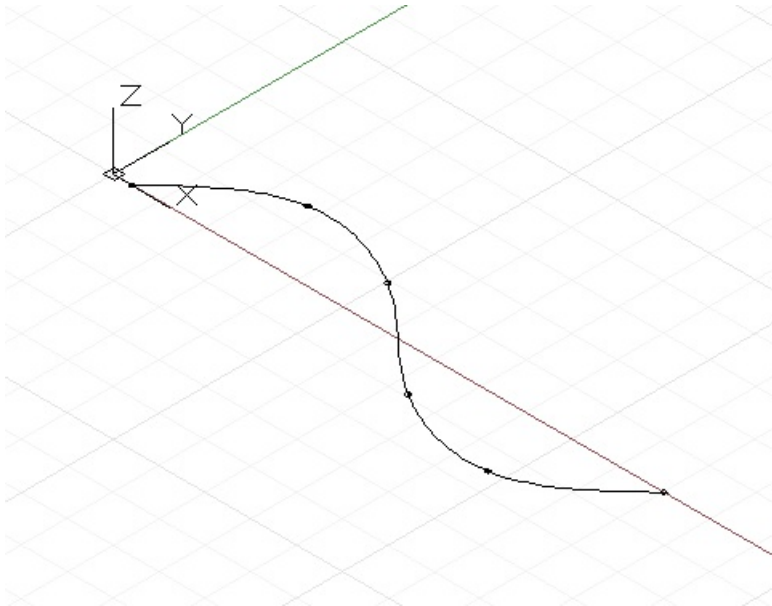
// d has value -7
d = a.Dot(b);
```

Curvas: puntos interpolados y de control

Curvas: puntos interpolados y de control

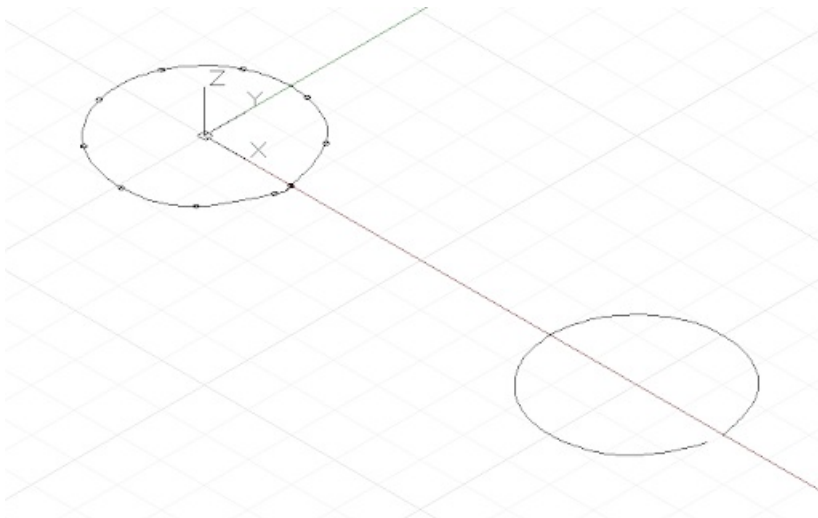
Existen dos formas fundamentales de crear curvas de forma libre en Dynamo: especificar una colección de puntos y conseguir que Dynamo interpole una curva suavizada entre ellos, o un método de nivel más bajo mediante la especificación de los puntos de control subyacentes de una curva de un determinado grado. Las curvas interpoladas son útiles cuando un diseñador conoce exactamente la forma que debe adoptar una línea o si el diseño tiene restricciones específicas en cuanto a la ubicación que la curva puede o no atravesar. Las curvas especificadas mediante puntos de control son, en esencia, una serie de segmentos de línea recta que un algoritmo suaviza para formar una curva final. La especificación de una curva mediante puntos de control puede resultar útil para explorar formas de curva con diferentes grados de suavizado o cuando se requiere una continuidad suave entre segmentos de curva.

Para crear una curva interpolada, simplemente transfiera un conjunto de puntos al método *NurbsCurve.ByPoints*.



```
num_pts = 6;  
s = Math.Sin(0..360..#num_pts) * 4;  
pts = Point.ByCoordinates(1..30..#num_pts, s, 0);  
int_curve = NurbsCurve.ByPoints(pts);
```

La curva generada interseca cada uno de los puntos de entrada, comenzando y terminando en el primer y el último punto del conjunto, respectivamente. Se puede utilizar un parámetro periódico opcional para crear una curva periódica cerrada. Dynamo rellenará automáticamente el segmento que falta, por lo que no es necesario un punto final duplicado (idéntico al punto inicial).

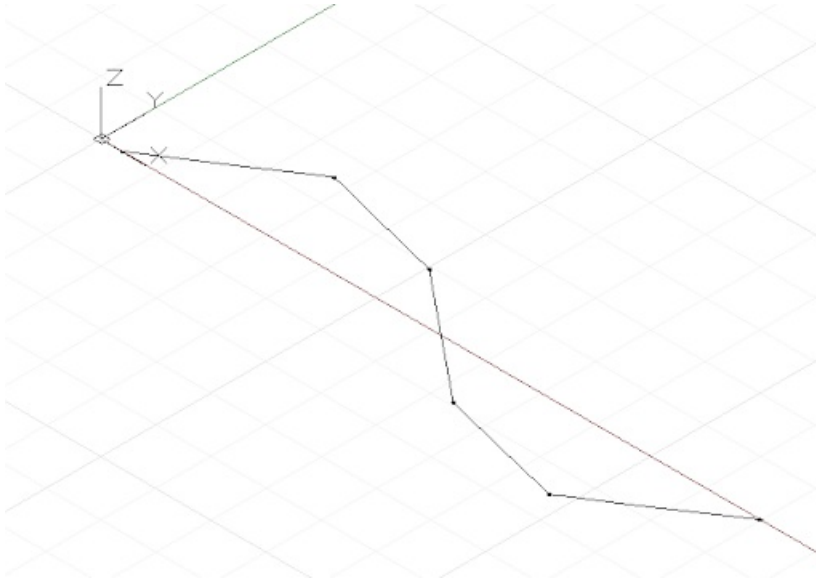


```
pts = Point.ByCoordinates(Math.Cos(0..350..#10),  
    Math.Sin(0..350..#10), 0);
```

```
// create an closed curve
crv = NurbsCurve.ByPoints(pts, true);

// the same curve, if left open:
crv2 = NurbsCurve.ByPoints(pts.Translate(5, 0, 0),
    false);
```

Las NurbsCurves se generan de forma muy similar, con puntos de entrada que representan los puntos finales de un segmento de línea recta y un segundo parámetro que especifica la cantidad y el tipo de suavizado de la curva, que se denomina grado.* Una curva con el grado 1 no presenta suavizado; es una polilínea.

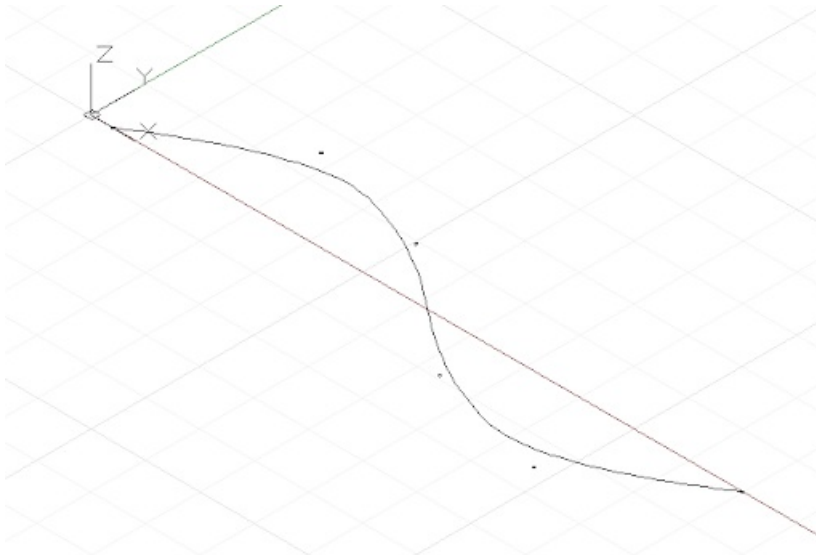


```
num_pts = 6;

pts = Point.ByCoordinates(1..30..#num_pts,
    Math.Sin(0..360..#num_pts) * 4, 0);

// a B-Spline curve with degree 1 is a polyline
ctrl_curve = NurbsCurve.ByControlPoints(pts, 1);
```

Una curva con el grado 2 se suaviza de forma que esta se interseque con el punto medio de los segmentos de polilínea y sea tangente a él:

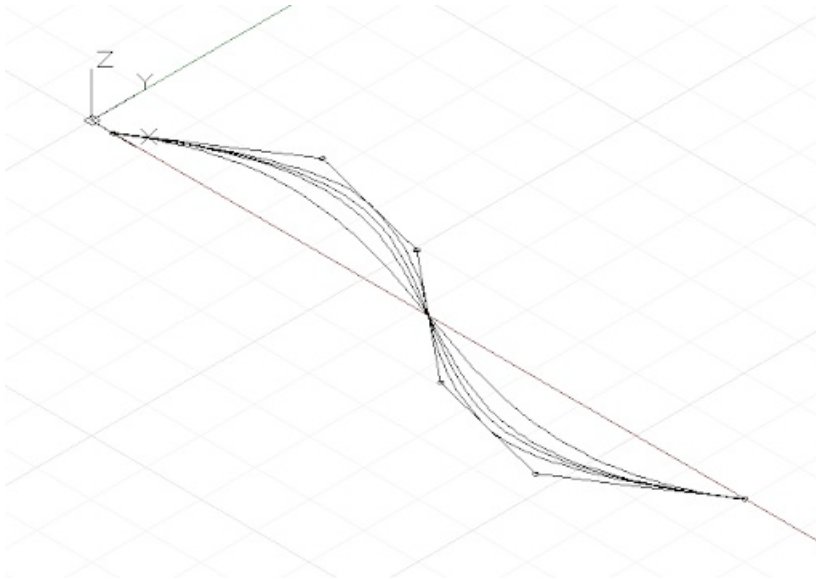


```
num_pts = 6;

pts = Point.ByCoordinates(1..30..#num_pts,
    Math.Sin(0..360..#num_pts) * 4, 0);

// a B-Spline curve with degree 2 is smooth
ctrl_curve = NurbsCurve.ByControlPoints(pts, 2);
```

Dynamo admite curvas NURBS (B-spline racional no uniforme) hasta un grado 20; la siguiente secuencia de comandos ilustra el efecto que tiene el aumento de niveles de suavizado en la forma de una curva:



```
num_pts = 6;

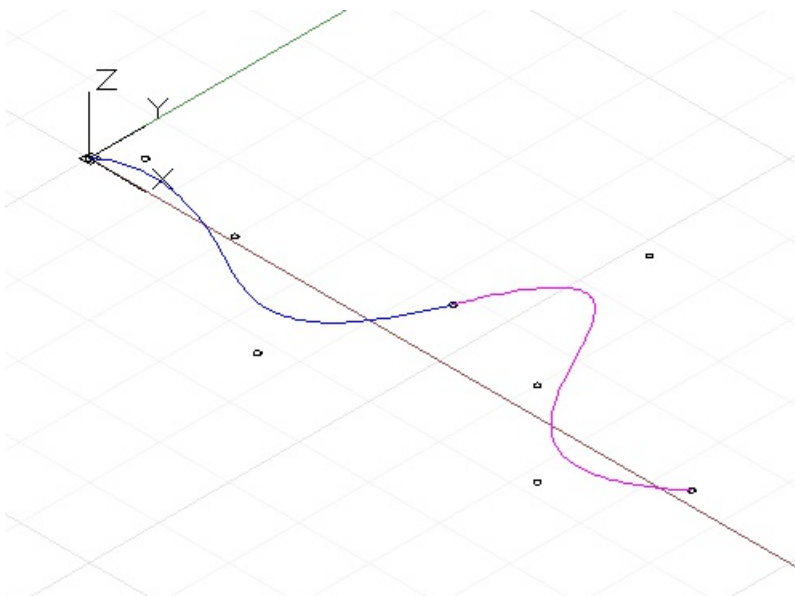
pts = Point.ByCoordinates(1..30..num_pts,
    Math.Sin(0..360..num_pts) * 4, 0);

def create_curve(pts : Point[], degree : int)
{
    return = NurbsCurve.ByControlPoints(pts,
        degree);
}

ctrl_crvs = create_curve(pts, 1..11);
```

Tenga en cuenta que debe tener al menos un punto de control más que el grado de la curva.

Otra ventaja de crear curvas mediante vértices de control es la capacidad de mantener la tangencia entre segmentos de curva individuales. Para ello, se extrae la dirección entre los dos últimos puntos de control y se continúa esta dirección con los dos primeros puntos de control de la curva siguiente. En el siguiente ejemplo, se crean dos curvas NURBS independientes que son, no obstante, tan suaves como una curva:



```
pts_1 = {};

pts_1[0] = Point.ByCoordinates(0, 0, 0);
pts_1[1] = Point.ByCoordinates(1, 1, 0);
pts_1[2] = Point.ByCoordinates(5, 0.2, 0);
pts_1[3] = Point.ByCoordinates(9, -3, 0);
pts_1[4] = Point.ByCoordinates(11, 2, 0);

crv_1 = NurbsCurve.ByControlPoints(pts_1, 3);

pts_2 = {};
```



```
pts_2[0] = pts_1[4];
end_dir = pts_1[4].Subtract(pts_1[3].AsVector());

pts_2[1] = Point.ByCoordinates(pts_2[0].X + end_dir.X,
    pts_2[0].Y + end_dir.Y, pts_2[0].Z + end_dir.Z);

pts_2[2] = Point.ByCoordinates(15, 1, 0);
pts_2[3] = Point.ByCoordinates(18, -2, 0);
pts_2[4] = Point.ByCoordinates(21, 0.5, 0);

crv_2 = NurbsCurve.ByControlPoints(pts_2, 3);
```

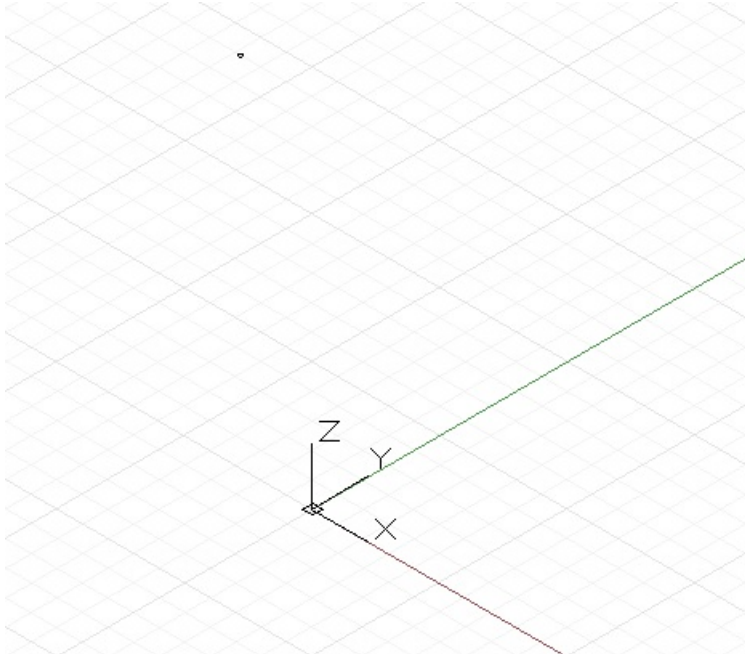
- Esta es una descripción muy simplificada de la geometría de curva NURBS; para obtener una explicación más precisa y detallada, consulte Pottmann, et al. 2007, en las referencias.

Traslación, rotación y otras transformaciones

Traslación, rotación y otras transformaciones

Algunos objetos de geometría se pueden crear indicando de forma explícita las coordenadas X, Y y Z en un espacio tridimensional. Sin embargo, con mayor frecuencia, la geometría se desplaza a su posición final mediante transformaciones geométricas en el propio objeto o en su `CoordinateSystem` subyacente.

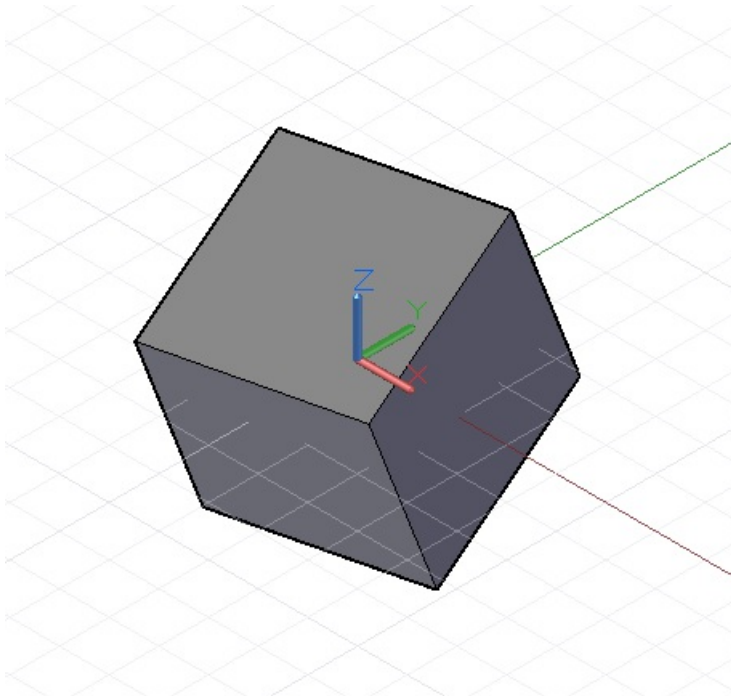
La transformación geométrica más sencilla es una traslación, que desplaza un objeto un número determinado de unidades en las direcciones X, Y y Z.



```
// create a point at x = 1, y = 2, z = 3
p = Point.ByCoordinates(1, 2, 3);

// translate the point 10 units in the x direction,
// -20 in y, and 50 in z
// p2's new position is x = 11, y = -18, z = 53
p2 = p.Translate(10, -20, 50);
```

Aunque todos los objetos de Dynamo se pueden trasladar mediante la adición del método *.Translate* al final del nombre del objeto, las transformaciones más complejas requieren transformar el objeto de un `CoordinateSystem` subyacente a un nuevo `CoordinateSystem`. Por ejemplo, para girar un objeto 45 grados alrededor del eje X, debemos transformar el objeto de su `CoordinateSystem` existente sin rotación a un `CoordinateSystem` que se había girado 45 grados alrededor del eje X con el método *.Transform*:



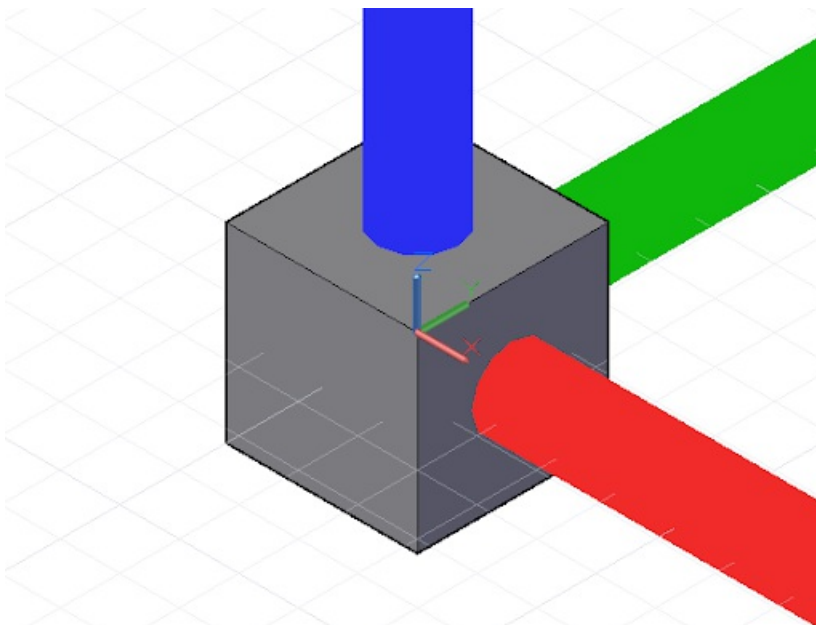
```
cube = Cuboid.ByLengths(CoordinateSystem.Identity(),
    10, 10, 10);

new_cs = CoordinateSystem.Identity();
new_cs2 = new_cs.Rotate(Point.ByCoordinates(0, 0),
    Vector.ByCoordinates(1,0,0.5), 25);

// get the existing coordinate system of the cube
old_cs = CoordinateSystem.Identity();

cube2 = cube.Transform(old_cs, new_cs2);
```

Además de trasladarse y rotarse, los objetos `CoordinateSystem` también se pueden crear cortados o con su escala ajustada. Se puede ajustar la escala de un `CoordinateSystem` con el método `.Scale`:



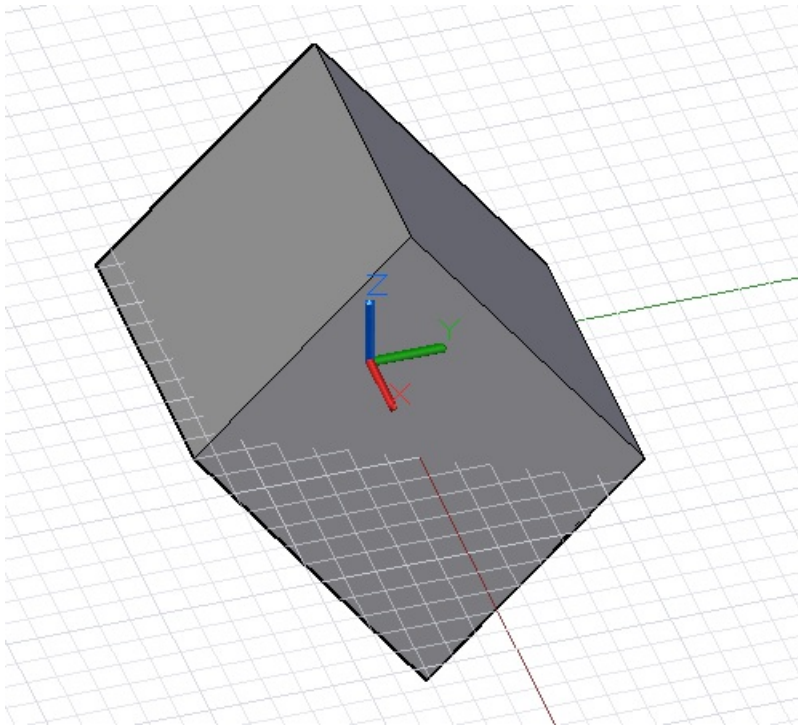
```
cube = Cuboid.ByLengths(CoordinateSystem.Identity(),
    10, 10, 10);

new_cs = CoordinateSystem.Identity();
new_cs2 = new_cs.Scale(20);

old_cs = CoordinateSystem.Identity();
```

```
cube2 = cube.Transform(old_cs, new_cs2);
```

Los objetos CoordinateSystem cortados se crean mediante la introducción de vectores no ortogonales en el constructor CoordinateSystem.



```
new_cs = CoordinateSystem.ByOriginVectors(
    Point.ByCoordinates(0, 0, 0),
    Vector.ByCoordinates(-1, -1, 1),
    Vector.ByCoordinates(-0.4, 0, 0));

old_cs = CoordinateSystem.Identity();

cube = Cuboid.ByLengths(CoordinateSystem.Identity(),
    5, 5, 5);

new_curves = cube.Transform(old_cs, new_cs);
```

La escala y el corte son transformaciones geométricas más complejas que la rotación y la traslación, por lo que no todos los objetos de Dynamo pueden someterse a ellas. En la siguiente tabla, se describen los objetos de Dynamo que pueden tener objetos CoordinateSystem con escala no uniforme y cortados.

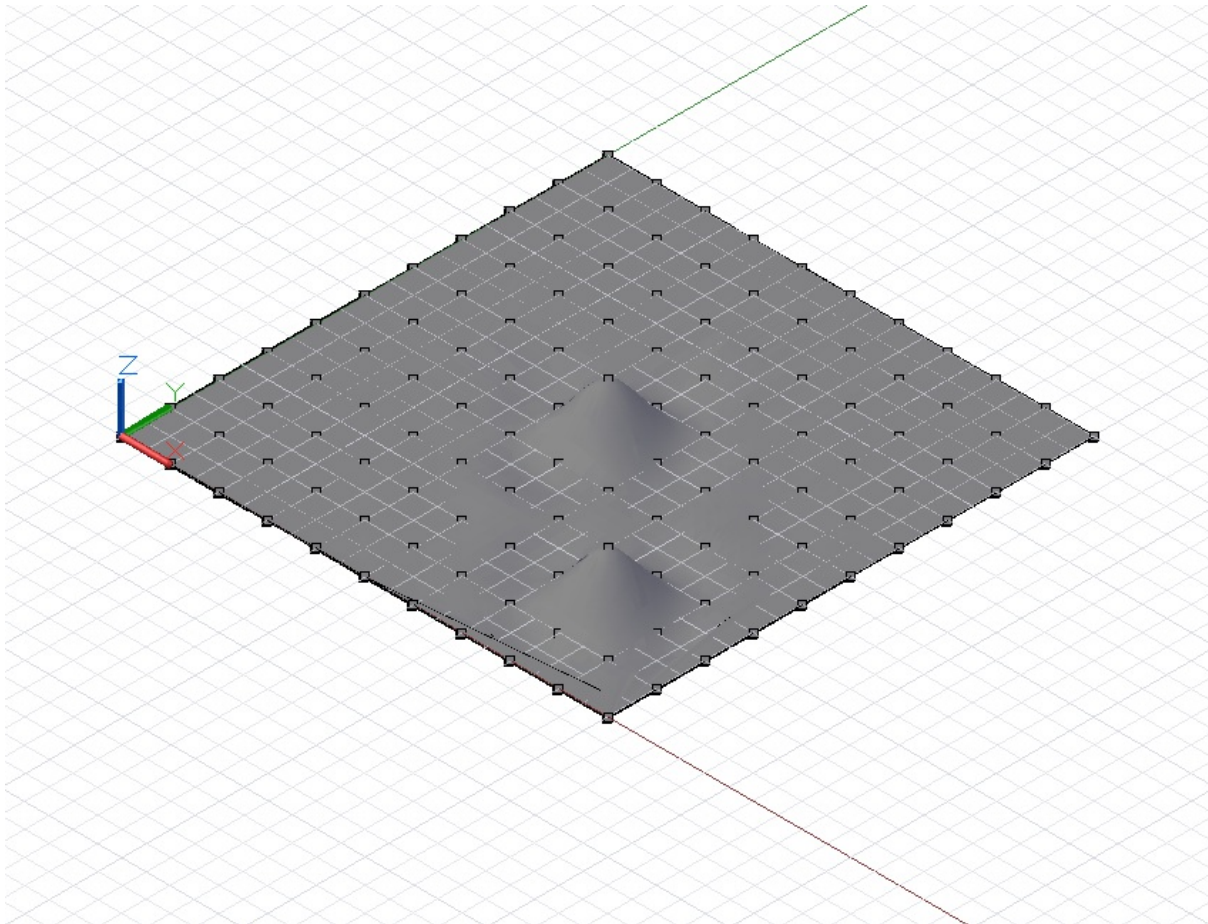
Clase	CoordinateSystem con escala no uniforme	CoordinateSystem cortado
Arco	No	No
NurbsCurve	Sí	Sí
NurbsSurface	No	No
Círculo	No	No
Línea	Sí	Sí
Plano	No	No
Punto	Sí	Sí
Polígono	No	No
Sólido	No	No
Superficie	No	No
Texto	No	No

Superficies: interpoladas, puntos de control, solevación y revolución

Superficies: interpoladas, puntos de control, solevación y revolución

El componente análogo bidimensional a una NurbsCurve es la NurbsSurface y, al igual que la NurbsCurve de forma libre, las NurbsSurfaces se pueden crear con dos métodos básicos: la introducción de un conjunto de puntos base y la interpolación entre ellos por parte de Dynamo, y la especificación explícita de los puntos de control de la superficie. Al igual que las curvas de forma libre, las superficies interpoladas son útiles cuando un diseñador conoce con precisión la forma que debe adoptar una superficie o si un diseño requiere que la superficie pase por los puntos de restricción. Por otra parte, las superficies creadas mediante puntos de control pueden ser más útiles para los diseños exploratorios a través de varios niveles de suavizado.

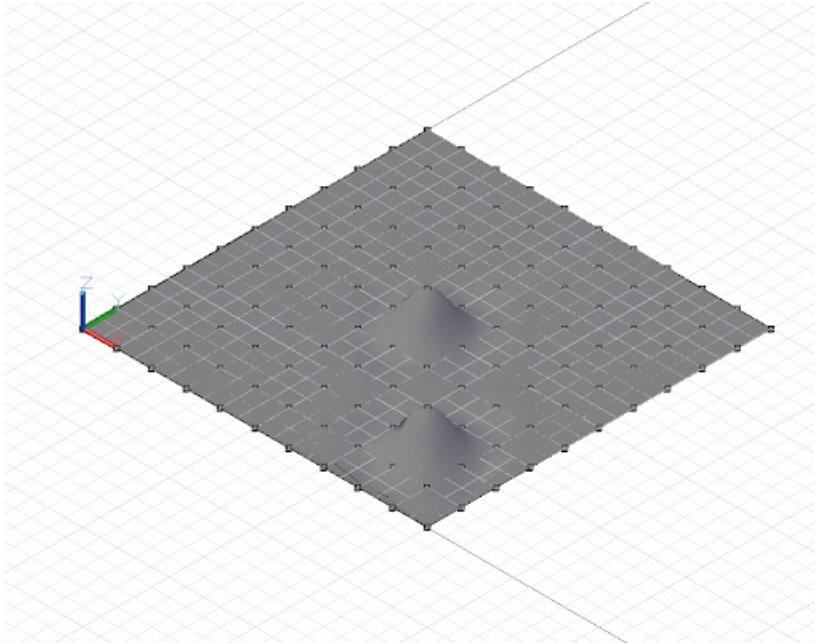
Para crear una superficie interpolada, simplemente genere una colección bidimensional de puntos que se aproxime a la forma de una superficie. La colección debe ser rectangular, es decir, no irregular. El método *NurbsSurface.ByPoints* crea una superficie a partir de estos puntos.



```
// python_points_1 is a set of Points generated with  
// a Python script found in Chapter 12, Section 10
```

```
surf = NurbsSurface.ByPoints(python_points_1);
```

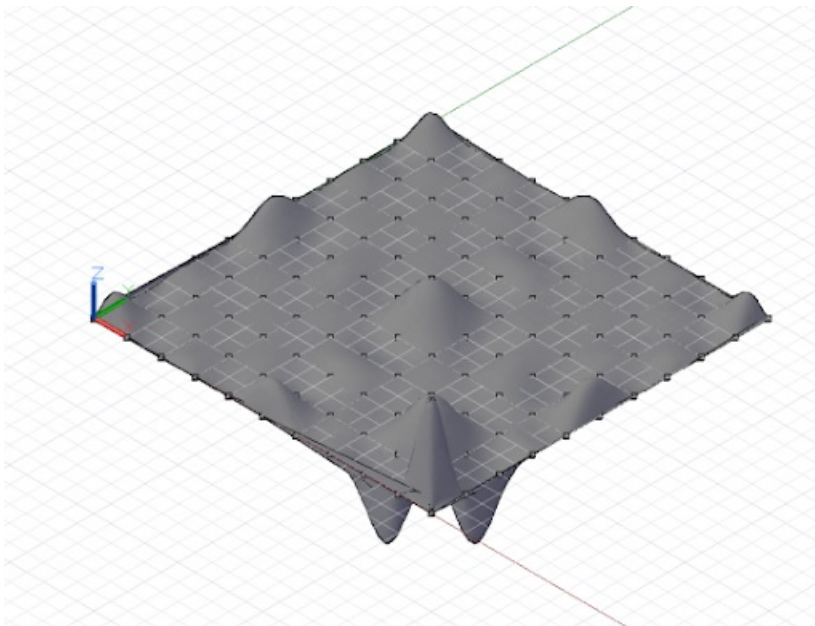
También se pueden crear NurbsSurfaces de forma libre mediante la especificación de los puntos de control subyacentes de una superficie. Al igual que las NurbsCurves, se puede considerar que los puntos de control representan una malla cuadrilateral con segmentos rectos que, en función del grado de la superficie, se suaviza hasta alcanzar la forma de superficie final. Para crear una NurbsSurface mediante puntos de control, incluya dos parámetros adicionales en *NurbsSurface.ByPoints*, que indiquen los grados de las curvas subyacentes en ambas direcciones de la superficie.



```
// python_points_1 is a set of Points generated with
// a Python script found in Chapter 12, Section 10

// create a surface of degree 2 with smooth segments
surf = NurbsSurface.ByPoints(python_points_1, 2, 2);
```

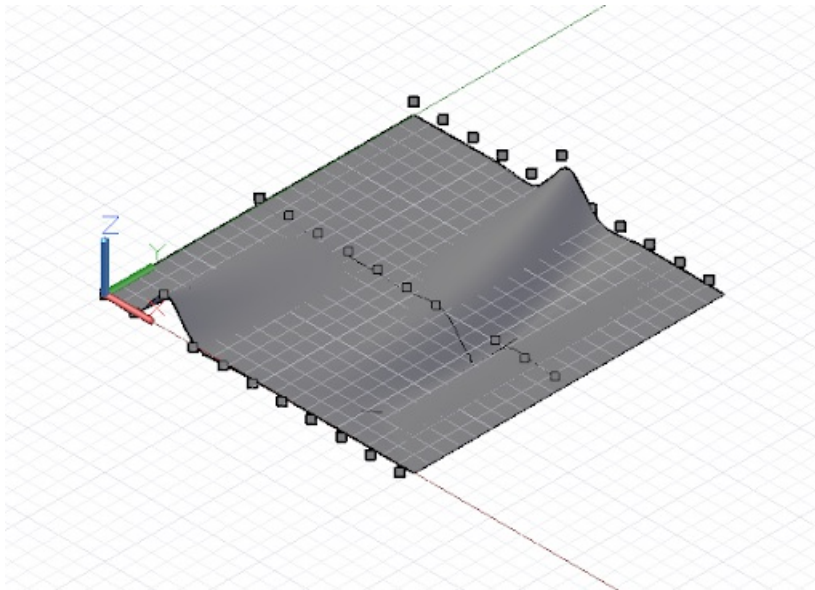
Podemos aumentar el grado de la NurbsSurface para cambiar la geometría de superficie resultante:



```
// python_points_1 is a set of Points generated with
// a Python script found in Chapter 12, Section 10

// create a surface of degree 6
surf = NurbsSurface.ByPoints(python_points_1, 6, 6);
```

Del mismo modo que las superficies se pueden crear mediante la interpolación entre un conjunto de puntos de entrada, estas se pueden crear mediante la interpolación entre un conjunto de curvas base. Esta operación se denomina *solevado*. Las curvas solevadas se crean mediante el constructor *Surface.ByLoft* con una colección de curvas de entrada como único parámetro.



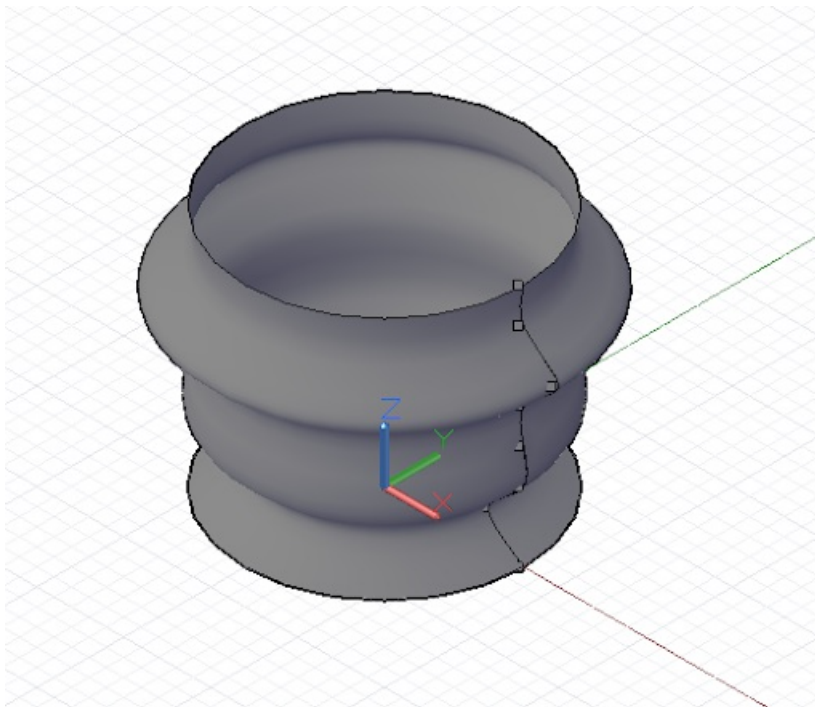
```
// python_points_2, 3, and 4 are generated with
// Python scripts found in Chapter 12, Section 10
```

```
c1 = NurbsCurve.ByPoints(python_points_2);
c2 = NurbsCurve.ByPoints(python_points_3);
c3 = NurbsCurve.ByPoints(python_points_4);
```

```
loft = Surface.ByLoft([c1, c2, c3]);
```

Las superficies de revolución son un tipo adicional de superficie creada mediante el barrido de una curva base alrededor de un eje central. Si las superficies interpoladas son el elemento análogo bidimensional a las curvas interpoladas, las superficies de revolución son el elemento análogo bidimensional a los círculos y los arcos.

Las superficies de revolución se especifican mediante una curva base que representa la "arista" de la superficie; un origen de eje, el punto base de la superficie; una dirección de eje, la dirección central del "núcleo", y un ángulo inicial y final de barrido. Se utilizan como entrada en el constructor *Surface.Revolve*.



```
pts = {};
pts[0] = Point.ByCoordinates(4, 0, 0);
pts[1] = Point.ByCoordinates(3, 0, 1);
pts[2] = Point.ByCoordinates(4, 0, 2);
pts[3] = Point.ByCoordinates(4, 0, 3);
pts[4] = Point.ByCoordinates(4, 0, 4);
pts[5] = Point.ByCoordinates(5, 0, 5);
```

```
pts[6] = Point.ByCoordinates(4, 0, 6);  
pts[7] = Point.ByCoordinates(4, 0, 7);  
  
crv = NurbsCurve.ByPoints(pts);  
  
axis_origin = Point.ByCoordinates(0, 0, 0);  
axis = Vector.ByCoordinates(0, 0, 1);  
  
surf = Surface.ByRevolve(crv, axis_origin, axis, 0,  
    360);
```

Parametrización geométrica

Parametrización geométrica

En los diseños computacionales, las curvas y las superficies se utilizan con frecuencia como el andamiaje subyacente para crear la geometría posterior. Para que esta geometría inicial se pueda utilizar como base para la geometría posterior, la secuencia de comandos debe poder extraer cualidades como, por ejemplo, la posición y la orientación en toda el área del objeto. Tanto las curvas como las superficies admiten esta extracción y este proceso se denomina parametrización.

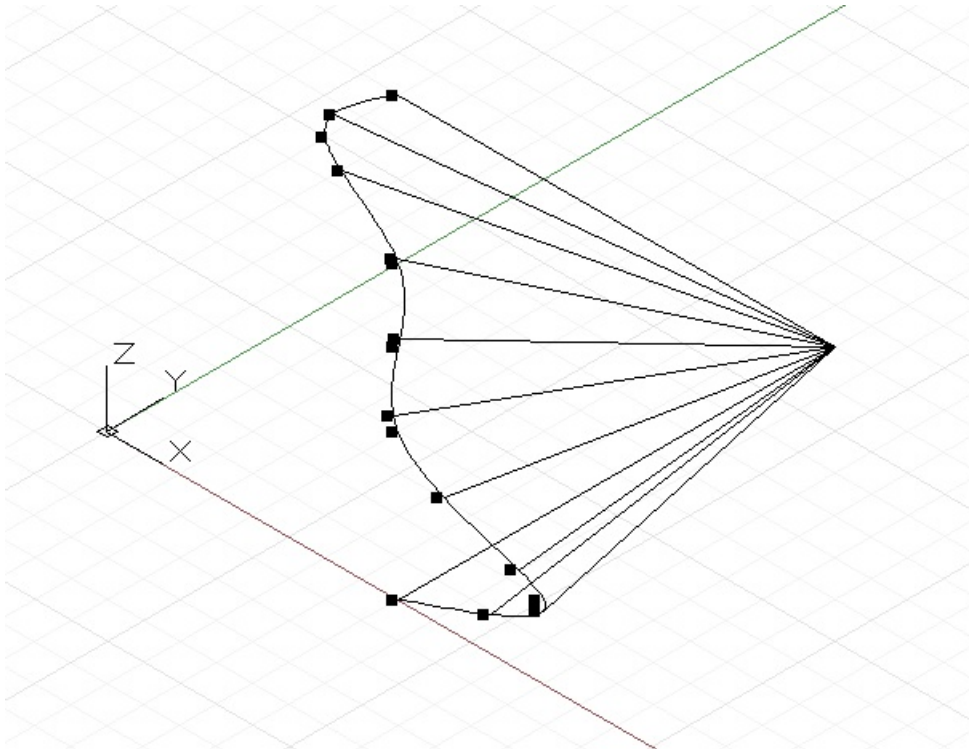
Todos los puntos de una curva se pueden considerar como si tuvieran un parámetro único que oscila entre 0 y 1. Si creáramos una NurbsCurve a partir de varios puntos de control o interpolación, el primer punto tendría el parámetro 0 y el último, el parámetro 1. Es imposible saber por adelantado cuál es el parámetro exacto para cualquier punto intermedio, lo que puede parecer una limitación grave, aunque se mitiga mediante una serie de funciones de utilidad. Las superficies tienen una parametrización similar a la de las curvas, aunque con dos parámetros en lugar de uno, denominados u y v. Si creáramos una superficie con los siguientes puntos:

```
pts = [ [p1, p2, p3],  
        [p4, p5, p6],  
        [p7, p8, p9] ];
```

p1 tendría el parámetro $u = 0$ $v = 0$, mientras que p9 tendría los parámetros $u = 1$ $v = 1$.

La parametrización no resulta especialmente útil a la hora de determinar los puntos utilizados para generar curvas. Su principal finalidad es determinar las ubicaciones si se utilizan los puntos intermedios generados por los constructores NurbsCurve y NurbsSurface.

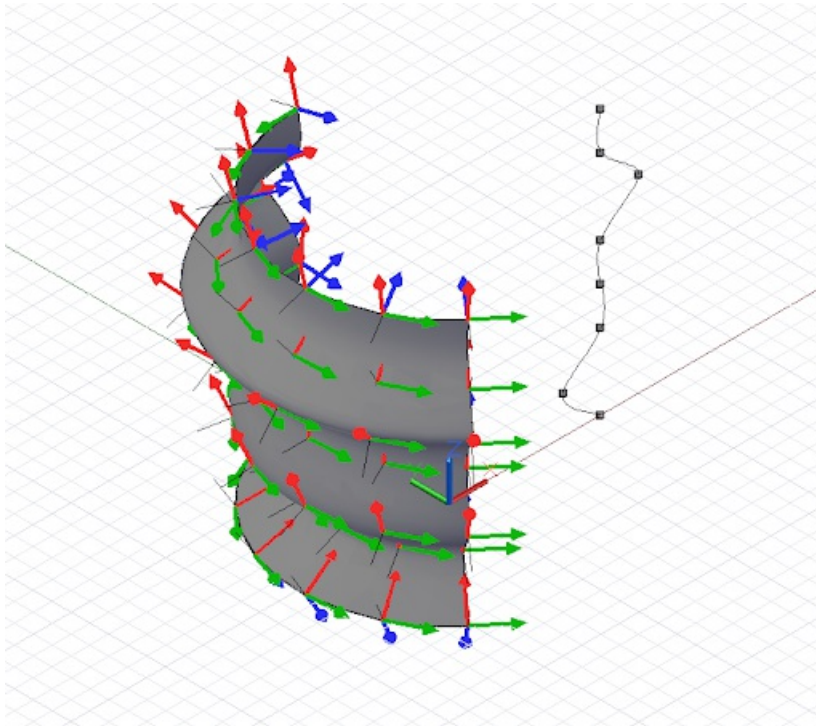
Las curvas presentan el método *PointAtParameter*, que utiliza un único argumento doble entre 0 y 1, y devuelve el objeto de punto en ese parámetro. Por ejemplo, esta secuencia de comandos busca los puntos en los parámetros 0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9 y 1:



```
pts = {};  
pts[0] = Point.ByCoordinates(4, 0, 0);  
pts[1] = Point.ByCoordinates(6, 0, 1);  
pts[2] = Point.ByCoordinates(4, 0, 2);  
pts[3] = Point.ByCoordinates(4, 0, 3);  
pts[4] = Point.ByCoordinates(4, 0, 4);  
pts[5] = Point.ByCoordinates(3, 0, 5);  
pts[6] = Point.ByCoordinates(4, 0, 6);  
  
crv = NurbsCurve.ByPoints(pts);  
  
pts_at_param = crv.PointAtParameter(0..1..#11);  
  
// draw Lines to help visualize the points  
lines = Line.ByStartPointEndPoint(pts_at_param,  
    Point.ByCoordinates(4, 6, 0));
```

De forma similar, las superficies presentan el método *PointAtParameter*, que utiliza dos argumentos, el parámetro u y v del punto generado.

Aunque la extracción de puntos individuales en una curva y una superficie puede ser útil, a menudo, es necesario que las secuencias de comandos conozcan las características geométricas específicas de un parámetro, como la dirección hacia la que está orientada la curva o la superficie. El método *CoordinateSystemAtParameter* no solo busca la posición, sino un *CoordinateSystem* orientado en el parámetro de una curva o una superficie. Por ejemplo, el siguiente archivo de comandos extrae objetos *CoordinateSystem* orientados a lo largo de una superficie de revolución y utiliza la orientación de los objetos *CoordinateSystem* para generar líneas que permanecen en la dirección normal a la superficie:



```
pts = {};
pts[0] = Point.ByCoordinates(4, 0, 0);
pts[1] = Point.ByCoordinates(3, 0, 1);
pts[2] = Point.ByCoordinates(4, 0, 2);
pts[3] = Point.ByCoordinates(4, 0, 3);
pts[4] = Point.ByCoordinates(4, 0, 4);
pts[5] = Point.ByCoordinates(5, 0, 5);
pts[6] = Point.ByCoordinates(4, 0, 6);
pts[7] = Point.ByCoordinates(4, 0, 7);

crv = NurbsCurve.ByPoints(pts);

axis_origin = Point.ByCoordinates(0, 0, 0);
axis = Vector.ByCoordinates(0, 0, 1);

surf = Surface.ByRevolve(crv, axis_origin, axis, 90,
140);

cs_array = surf.CoordinateSystemAtParameter(
(0..1..#7)<1>, (0..1..#7)<2>);

def make_line(cs : CoordinateSystem) {
  lines_start = cs.Origin;
  lines_end = cs.Origin.Translate(cs.ZAxis, -0.75);

  return = Line.ByStartPointEndPoint(lines_start,
  lines_end);
}

lines = make_line(Flatten(cs_array));
```

Como se ha mencionado anteriormente, la parametrización no siempre es uniforme a lo largo de una curva o una superficie, lo que significa que el parámetro 0.5 no siempre se corresponde con el punto medio y 0.25 no siempre se corresponde con el punto un cuarto a lo largo de una curva o una superficie. Para evitar esta limitación, las curvas presentan un conjunto adicional de comandos de parametrización que permiten buscar un punto en longitudes específicas a lo largo de una curva.

Intersección y recorte

Intersección y recorte

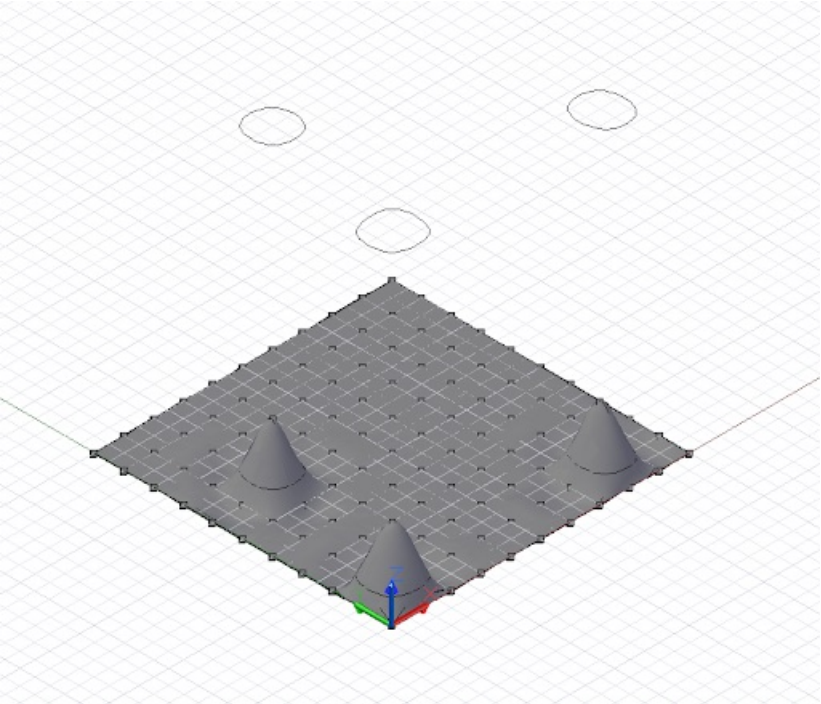
Muchos de los ejemplos mostrados hasta ahora se han centrado en la creación de geometría dimensional superior a partir de objetos dimensionales inferiores. Los métodos de intersección permiten que esta geometría dimensional superior genere objetos dimensionales inferiores, mientras que los comandos de recorte y de selección de recorte permiten que el archivo de comandos modifique las formas geométricas una vez creadas.

El método *Intersect* se define en todas las partes de la geometría de Dynamo, lo que significa que, en teoría, cualquier elemento de geometría se puede intersectar con cualquier otro. Naturalmente, algunas intersecciones no tienen sentido, como las intersecciones que involucran puntos, ya que el objeto resultante siempre será el propio punto de entrada. Las demás combinaciones posibles de intersecciones entre objetos se describen en el siguiente gráfico. En la siguiente tabla, se muestra el resultado de varias operaciones de intersección:

Intersect

Con:	Superficie	Curva	Plano	Sólido
Superficie	Curva	Punto	Punto, curva	Superficie
Curva	Punto	Punto	Punto	Curva
Plano	Curva	Punto	Curva	Curva
Sólido	Superficie	Curva	Curva	Sólido

En el siguiente ejemplo muy sencillo, se muestra la intersección de un plano con una NurbsSurface. La intersección genera una matriz de NurbsCurves, que se puede utilizar como cualquier otra NurbsCurve.



```
// python_points_5 is a set of Points generated with
// a Python script found in Chapter 12, Section 10

surf = NurbsSurface.ByPoints(python_points_5, 3, 3);

WCS = CoordinateSystem.Identity();

pl = Plane.ByOriginNormal(WCS.Origin.Translate(0, 0,
0.5), WCS.ZAxis);

// intersect surface, generating three closed curves
crvs = surf.Intersect(pl);

crvs_moved = crvs.Translate(0, 0, 10);
```

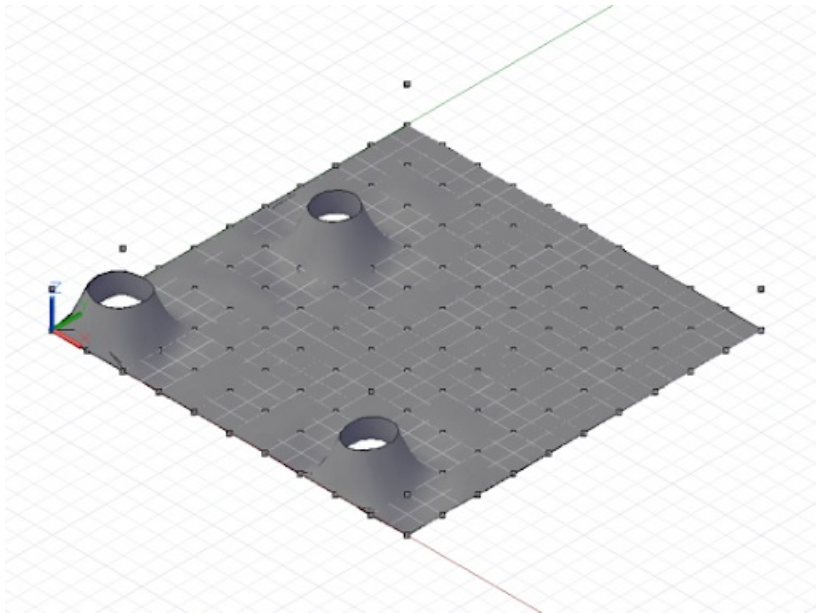
El método *Trim* es muy similar al método *Intersect*, ya que se define para casi cada elemento de geometría. Sin embargo, existen muchas más limitaciones en el método *Trim* que en *Intersect*.

Trim

Uso:	Punto	Curva	Plano	Superficie	Sólido
En: Curva	Sí	No	No	No	No

Polígono	-	No	Sí	No	No
Superficie	-	Sí	Sí	Sí	Sí
Sólido	-	-	Sí	Sí	Sí

Algo que se debe tener en cuenta en los métodos *Trim* es el requisito de un punto de "selección", un punto que determina la geometría que se descartará y las partes que se deben conservar. Dynamo busca y descarta la geometría recortada más cercana al punto de selección.



```
// python_points_5 is a set of Points generated with
// a Python script found in Chapter 12, Section 10

surf = NurbsSurface.ByPoints(python_points_5, 3, 3);

tool_pts = Point.ByCoordinates((-10,.20,.10)<1,
                              (-10,.20,.10)<2>, 1);

tool = NurbsSurface.ByPoints(tool_pts);

pick_point = Point.ByCoordinates(8, 1, 3);

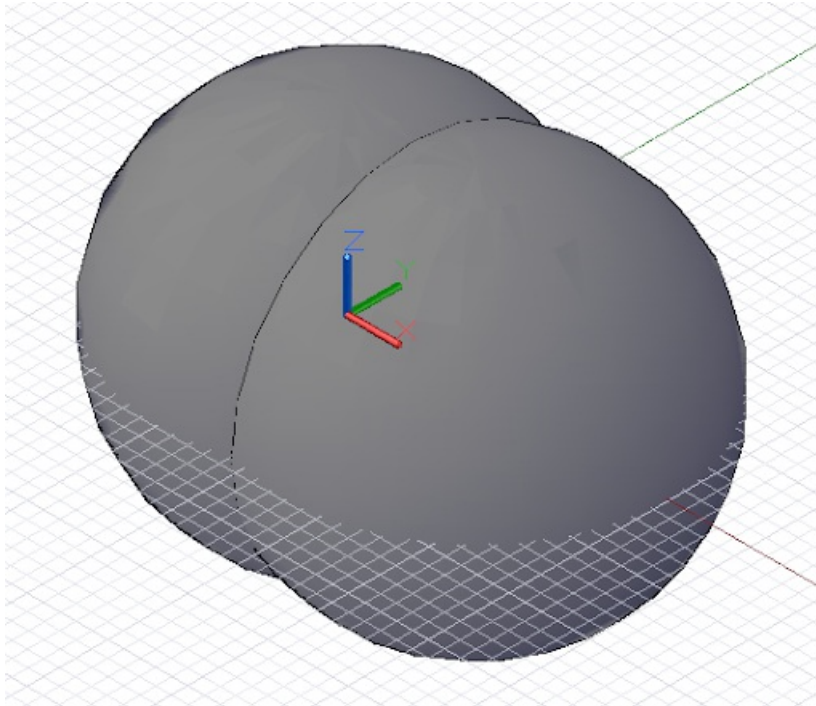
result = surf.Trim(tool, pick_point);
```

Operaciones booleanas geométricas

Operaciones booleanas geométricas

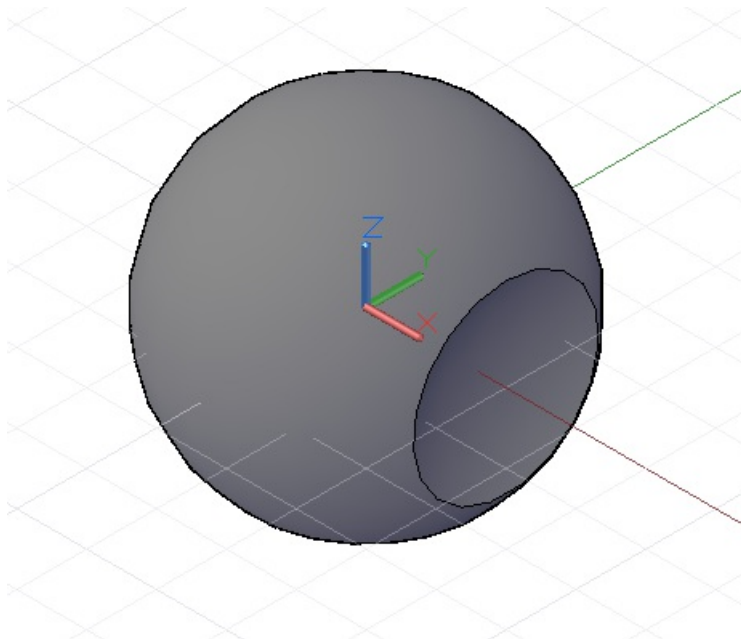
Los métodos *Intersect*, *Trim* y *SelectTrim* se utilizan principalmente en geometrías dimensionales inferiores como puntos, curvas y superficies. Por otra parte, la geometría sólida cuenta con un conjunto adicional de métodos para modificar la forma después de su creación, tanto sustrayendo material de forma similar a *Trim* como combinando elementos para formar un todo mayor.

El método *Union* utiliza dos objetos sólidos para crear un único objeto sólido a partir del espacio cubierto por ambos objetos. El espacio solapado entre los objetos se combina en la forma final. Este ejemplo combina una esfera y un cubo en una forma de esfera-cubo sólida:



```
s1 = Sphere.ByCenterPointRadius(  
    CoordinateSystem.Identity().Origin, 6);  
  
s2 = Sphere.ByCenterPointRadius(  
    CoordinateSystem.Identity().Origin.Translate(4, 0,  
    0), 6);  
  
combined = s1.Union(s2);
```

El método *Difference*, al igual que *Trim*, resta el contenido del sólido de la herramienta de entrada del sólido base. En este ejemplo, se extrae una pequeña hendidura de una esfera:

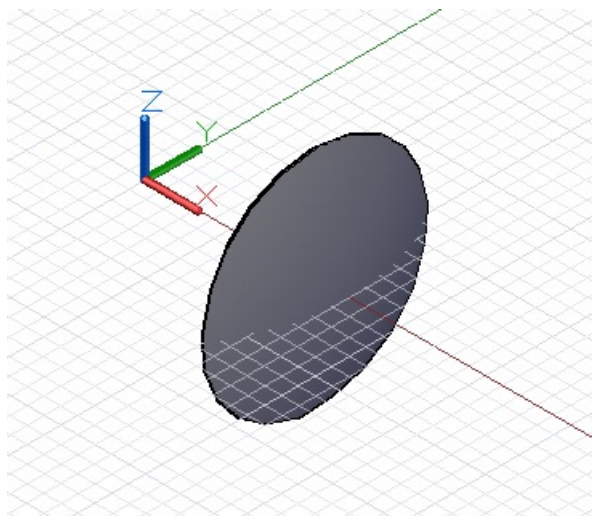


```
s = Sphere.ByCenterPointRadius(
    CoordinateSystem.Identity().Origin, 6);

tool = Sphere.ByCenterPointRadius(
    CoordinateSystem.Identity().Origin.Translate(10, 0,
    0), 6);

result = s.Difference(tool);
```

El método *Intersect* devuelve el sólido solapado entre dos entradas sólidas. En el siguiente ejemplo, *Difference* se ha cambiado a *Intersect* y el sólido resultante es el vacío que falta que se ha eliminado inicialmente:



```
s = Sphere.ByCenterPointRadius(
    CoordinateSystem.Identity().Origin, 6);

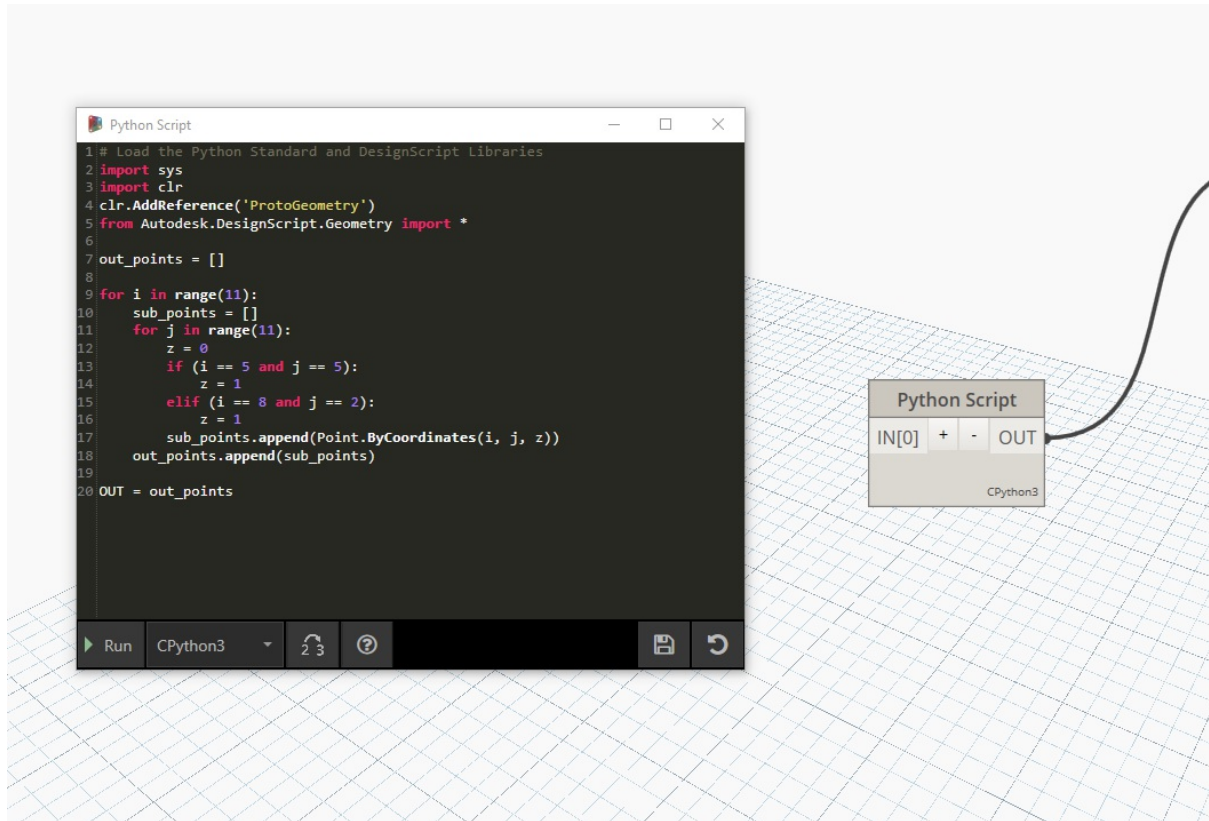
tool = Sphere.ByCenterPointRadius(
    CoordinateSystem.Identity().Origin.Translate(10, 0,
    0), 6);

result = s.Intersect(tool);
```

Generadores de puntos de Python

Generadores de puntos de Python

Las siguientes secuencias de comandos de Python generan matrices de puntos para varios ejemplos. Se deben pegar en un nodo Python Script, como se indica a continuación:



python_points_1

```
out_points = []

for i in range(11):
    sub_points = []
    for j in range(11):
        z = 0
        if (i == 5 and j == 5):
            z = 1
        elif (i == 8 and j == 2):
            z = 1
        sub_points.append(Point.ByCoordinates(i, j, z))
    out_points.append(sub_points)

OUT = out_points
```

python_points_2

```
out_points = []

for i in range(11):
    z = 0
    if (i == 2):
        z = 1
    out_points.append(Point.ByCoordinates(i, 0, z))

OUT = out_points
```

python_points_3

```
out_points = []

for i in range(11):
    z = 0
    if (i == 7):
```



```

        z = -1
    out_points.append(Point.ByCoordinates(i, 5, z))

OUT = out_points

python_points_4
out_points = []

for i in range(11):
    z = 0
    if (i == 5):
        z = 1
    out_points.append(Point.ByCoordinates(i, 10, z))

OUT = out_points

python_points_5
out_points = []

for i in range(11):
    sub_points = []
    for j in range(11):
        z = 0
        if (i == 1 and j == 1):
            z = 2
        elif (i == 8 and j == 1):
            z = 2
        elif (i == 2 and j == 6):
            z = 2
        sub_points.append(Point.ByCoordinates(i, j, z))
    out_points.append(sub_points)

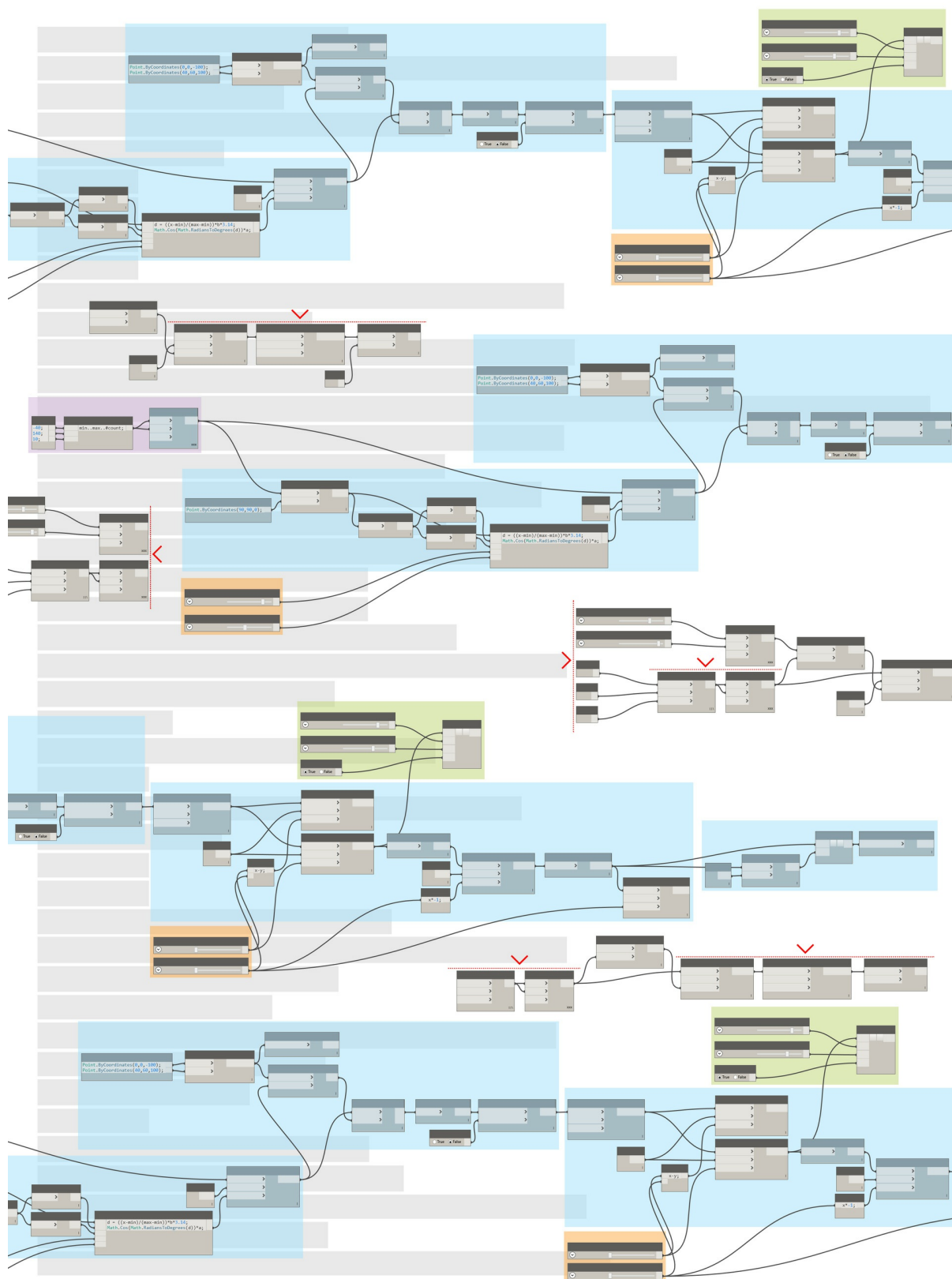
OUT = out_points

```

Prácticas recomendadas

Prácticas recomendadas

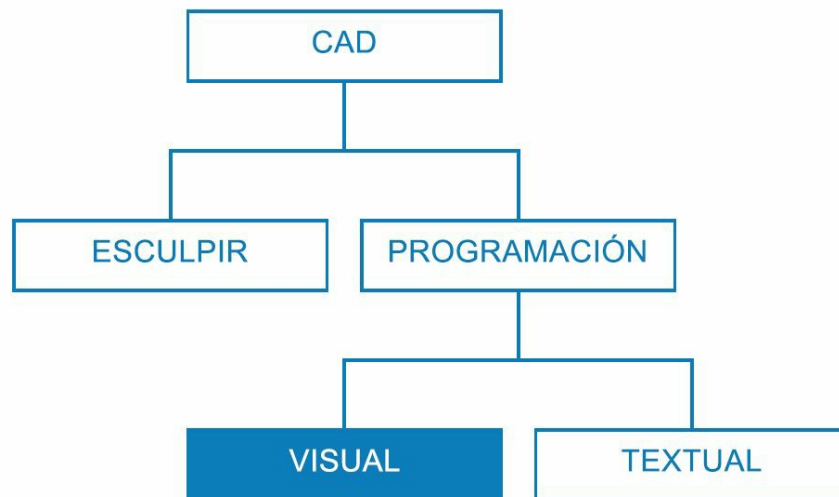
Esta parte del manual de introducción se ha organizado como un diario de "prácticas recomendadas". Ofrece información sobre varias estrategias que hemos aprendido a través de la experiencia y la investigación para ser más favorables a los flujos de trabajo paramétricos de calidad. Como diseñadores y programadores, nuestro indicador de calidad se basa principalmente en la capacidad de mantenimiento, fiabilidad, facilidad de uso y eficacia de nuestras herramientas. Aunque estas prácticas recomendadas presentan ejemplos específicos para secuencias de comandos visuales o basadas en texto, los principios se aplican a todos los entornos de programación y pueden informar de muchos flujos de trabajo computacionales.



Estrategias gráficas

Estrategias gráficas

Antes de este capítulo, en el manual de introducción, se describe cómo implementar las eficaces funciones de creación de secuencias de comandos visuales de Dynamo. Un conocimiento exhaustivo de estas funciones constituye una base sólida y el primer paso para la creación de programas visuales potentes. Cuando utilizamos los programas visuales sobre el terreno, los compartimos con compañeros, solucionamos errores o probamos los límites, nos encontraremos con problemas adicionales que deberemos afrontar. Si otra persona va a utilizar el programa o espera abrirlo dentro de seis meses, este debe tener una claridad gráfica y lógica inmediata. Dynamo dispone de numerosas herramientas para gestionar la complejidad del programa y, en este capítulo, se proporcionan las directrices para utilizarlas.

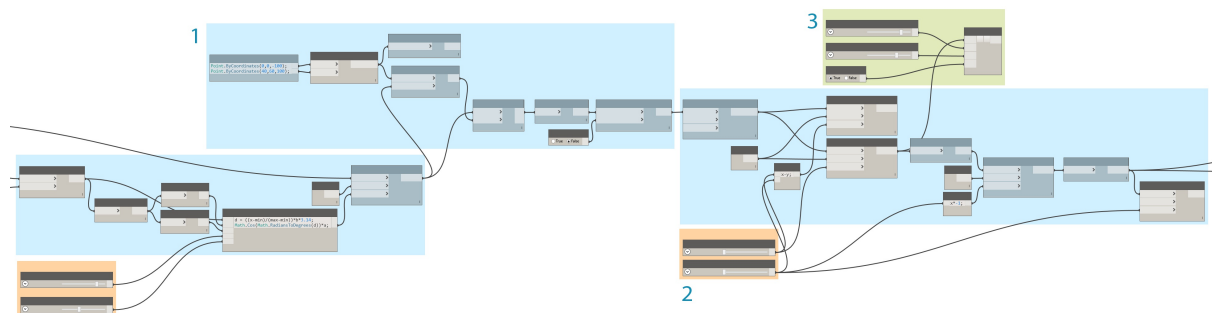


Reducción de la complejidad

A medida que desarrolla el gráfico de Dynamo y prueba sus ideas, su tamaño y su complejidad pueden aumentar rápidamente. Aunque es importante crear un programa que funcione, es igualmente importante hacerlo lo más sencillo posible. No solo el gráfico se ejecutará de forma más rápida y predecible, sino que además usted y los demás usuarios entenderán su lógica más adelante. A continuación, se incluyen varias formas que le ayudarán a aclarar la lógica del gráfico.

Modularidad con grupos

- Los grupos permiten **crear partes con diferentes funciones** a medida que se genera un programa
- Los grupos permiten **desplazar grandes partes del programa** sin perder la modularidad y la alineación
- Puede cambiar el **color del grupo para diferenciar** su finalidad (entradas y funciones)
- Puede utilizar grupos para empezar a **organizar el gráfico a fin de simplificar la creación de nodos personalizados**.



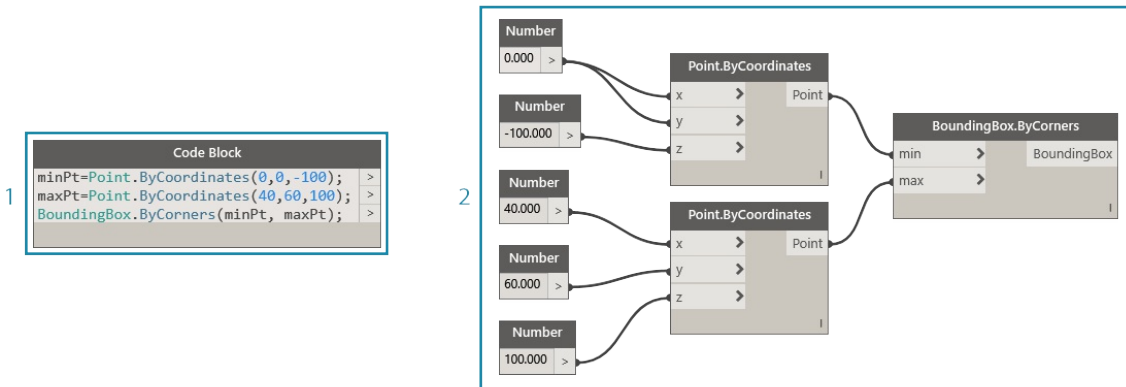
Los colores de este programa identifican la finalidad de cada grupo. Esta estrategia se puede utilizar para crear una jerarquía en cualquier norma o plantilla gráficas que desarrolle.

1. Grupo de funciones (azul)
2. Grupo de entradas (naranja)
3. Grupo de secuencias de comandos (verde) Para obtener información sobre cómo utilizar los grupos, consulte [Administración del programa](#).

Desarrollo eficaz con bloques de código

- En ocasiones, se puede utilizar un bloque de código para **escribir un método de nodo o número de forma más rápida que con la búsqueda** (Point.ByCoordinates, Number, String y Formula).

- Los bloques de código son útiles **cundo se desean definir funciones personalizadas en DesignScript para reducir el número de nodos de un gráfico.**

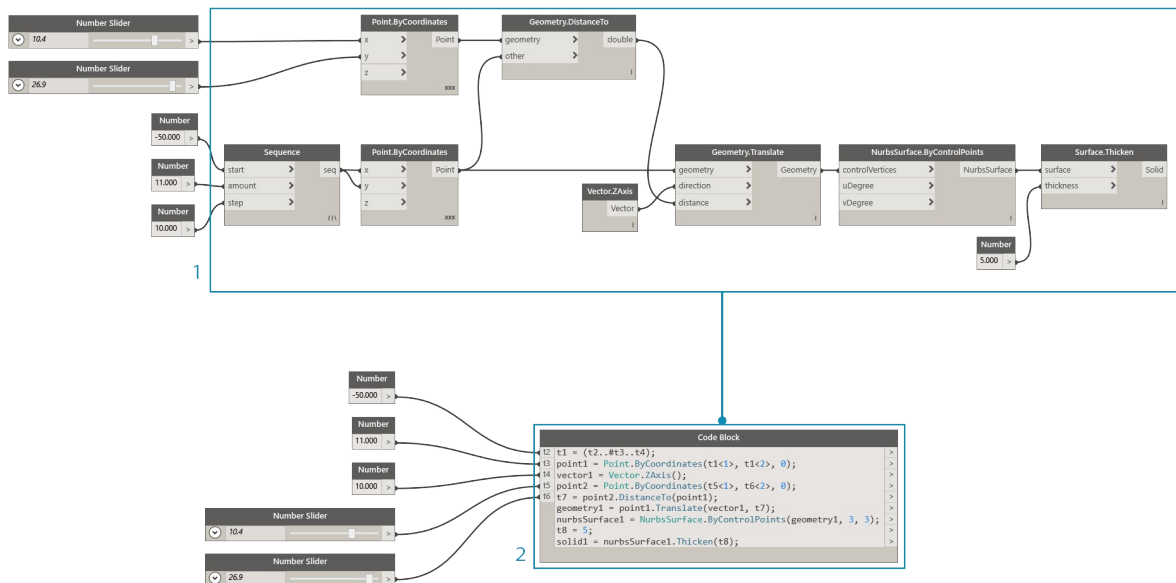


Los ejemplos 1 y 2 realizan la misma función. Se tardó mucho menos en escribir unas pocas líneas de código que en buscar y añadir cada nodo individualmente. El bloque de código también es mucho más conciso.

- DesignScript escrito en bloques de código
- Programa equivalente en nodos Para obtener información sobre cómo utilizar el bloque de código, consulte [¿Qué es un bloque de código?](#).

Condensación con de nodo a código

- Puede **reducir la complejidad de un gráfico mediante el método de nodo a código**, que utilizará una colección de nodos sencillos y escribirá la secuencia DesignScript correspondiente en un único bloque de código.
- El método de nodo a código permite **condensar código sin que el programa pierda claridad**.
- A continuación, se indican las **ventajas** de utilizar el método de nodo a código:
 - Condensa el código fácilmente en un componente que aún puede editarse.
 - Puede simplificar una parte significativa del gráfico.
 - Es útil si no se modifica con frecuencia el "miniprograma".
- Permite incorporar otras características de bloque de código, como las funciones.
- A continuación, se indican las **desventajas** de utilizar el método de nodo a código:
 - La asignación de nombres genéricos reduce su facilidad de lectura.
 - Es más difícil de comprender para otros usuarios.
 - No hay una forma sencilla de volver a la versión de programación visual.

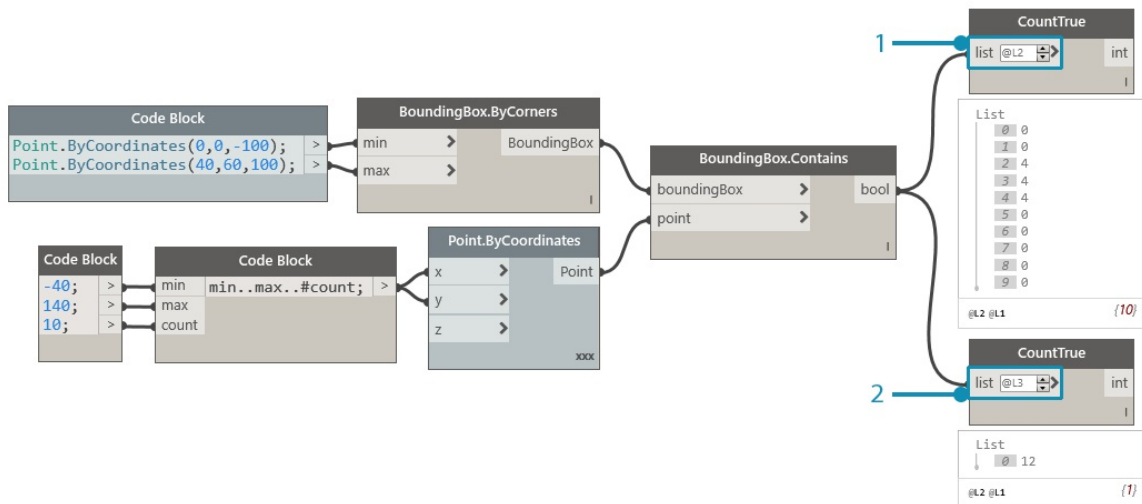


- Programa existente
- Bloque de código creado con el método de nodo a código Para obtener información sobre cómo utilizar el método de nodo a código, consulte [Sintaxis de DesignScript](#).

Acceso a los datos de forma flexible con List@Level

- El uso de List@Level puede ayudarle a **reducir la complejidad del gráfico mediante la sustitución de los nodos List.Map y List.Combine**, que pueden ocupar una cantidad considerable de espacio en el lienzo.

- List@Level proporciona un método **más rápido que List.Map/List.Combine** para **crear una lógica de nodo**, lo que permite acceder a los datos de cualquier nivel de una lista directamente desde el puerto de entrada de un nodo



Podemos comprobar cuántos valores devuelve True BoundingBox.Contains y en qué listas mediante la activación de List@Level para la entrada "list" de CountTrue. List@Level permite al usuario determinar de qué nivel se obtendrán los datos. El uso de List@Level es flexible, eficaz y muy recomendado frente a otros métodos en los que se utilizan List.Map y List.Combine.

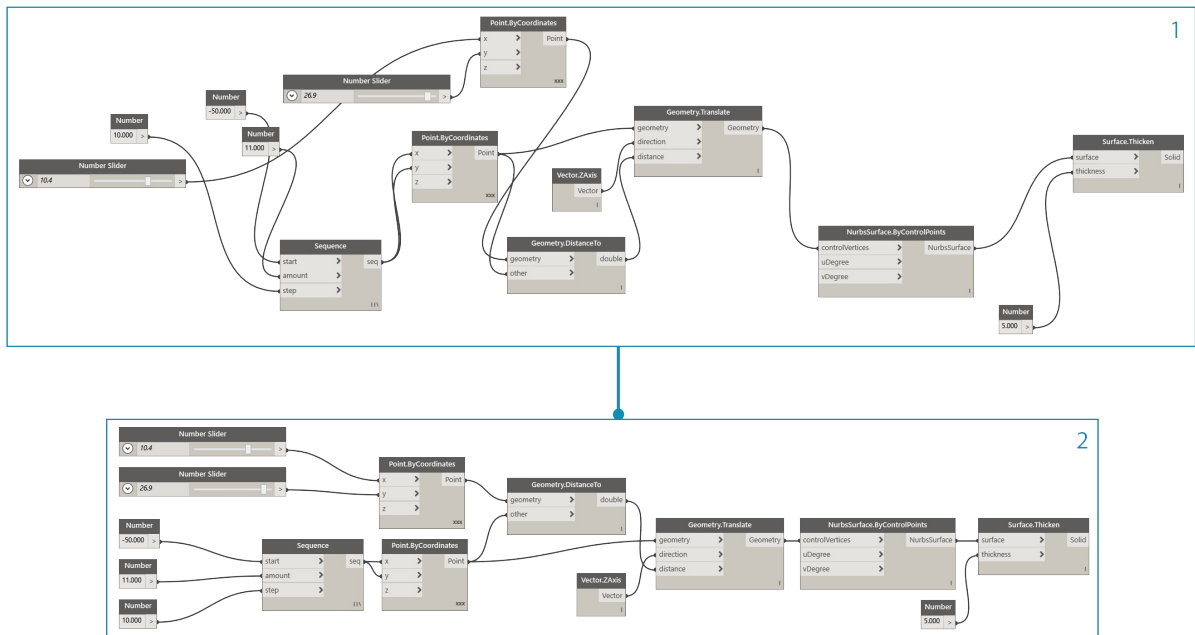
1. Recuento de valores "true" (verdaderos) en el nivel de lista 2
2. Recuento de valores "true" (verdaderos) en el nivel de lista 3 Para obtener información sobre cómo utilizar List@Level, consulte [Listas de listas](#).

Legibilidad

Además de conseguir que el gráfico sea lo más sencillo y eficaz posible, intente lograr una claridad gráfica. A pesar de hacer todo lo posible para que el gráfico sea intuitivo con agrupaciones lógicas, es posible que las relaciones no sean evidentes a simple vista. Una sencilla nota dentro de un grupo o el cambio de nombre de un control deslizante puede ahorrarle a usted u otro usuario confusión innecesaria o tener que desplazarse por el gráfico. A continuación, se describen varias formas que le ayudarán a aplicar coherencia dentro de los gráficos y entre ellos.

Continuidad visual con alineación de nodos

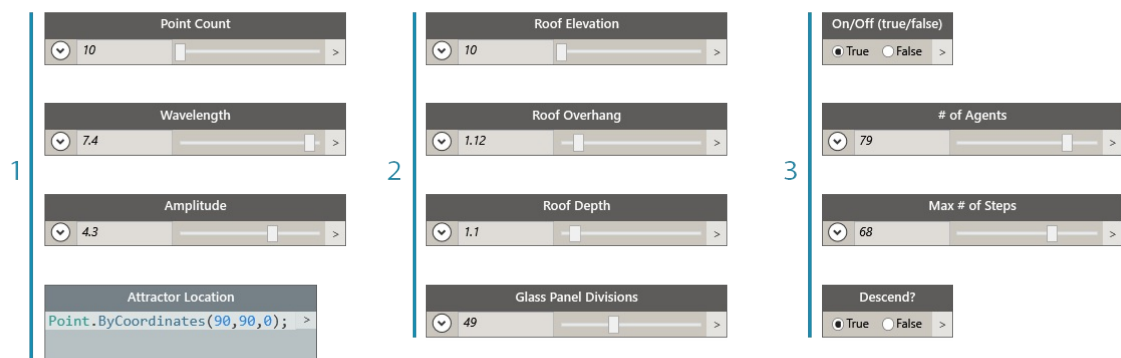
- Para reducir el trabajo después de terminar de generar el gráfico, debe intentar asegurarse de que el diseño del nodo sea legible mediante la **alineación de los nodos con frecuencia y a medida que progresa**.
- Si otros van a trabajar con el gráfico, **asegúrese de que el diseño de cables y nodos fluya con facilidad antes de enviarlo**.
- Para facilitar la alineación, **utilice la función "Presentación de nodo de limpieza" para alinear automáticamente** el gráfico, aunque de forma menos precisa que manualmente.



1. Gráfico desorganizado
2. Gráfico alineado Para obtener información sobre cómo utilizar la alineación de nodos, consulte [Administración del programa](#).

Etiquetado descriptivo mediante cambio de nombre

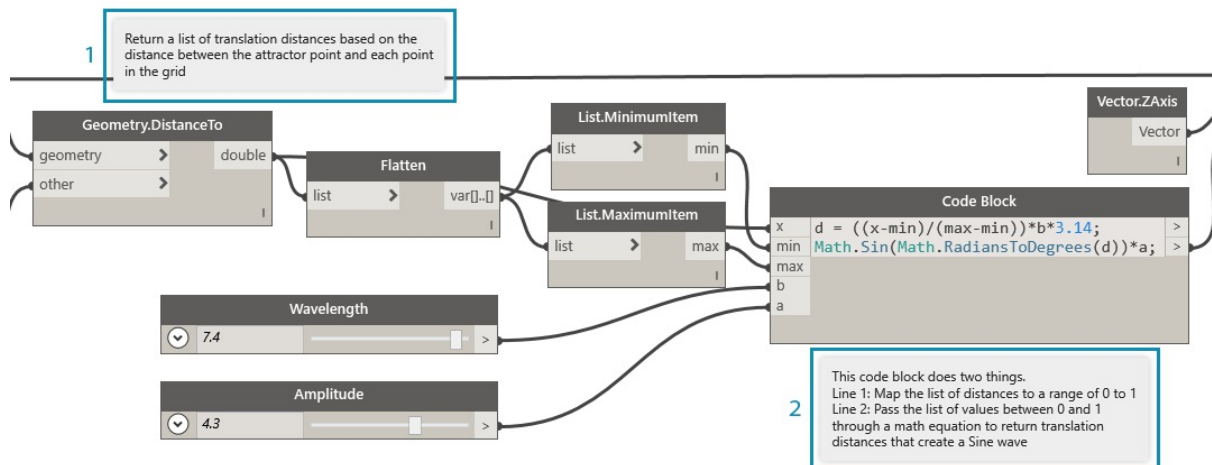
- El cambio de nombre de las entradas puede ayudar a otros a comprender más fácilmente el gráfico, **sobre todo, si los elementos a los que se van a conectar no aparecen en la pantalla.**
- **Tenga cuidado de no cambiar el nombre de los nodos que no sean entradas.** Una alternativa a esto es crear un nodo personalizado a partir de un clúster de nodos y cambiarle el nombre; se comprenderá que contiene otro elemento.



1. Entradas para la manipulación de superficies
2. Entradas para los parámetros arquitectónicos
3. Entradas para la secuencia de comandos de simulación de drenaje Para cambiar el nombre de un nodo, haga clic con el botón derecho en su nombre y seleccione "Cambiar nombre de nodo..."

Explicación con notas

- Debe añadir una nota si alguna parte del **gráfico requiere una explicación en lenguaje sencillo** que los nodos no pueden expresar.
- Debe añadir una nota si un conjunto de **nodos o un grupo son demasiado grandes o complejos y no se pueden entender al instante.**



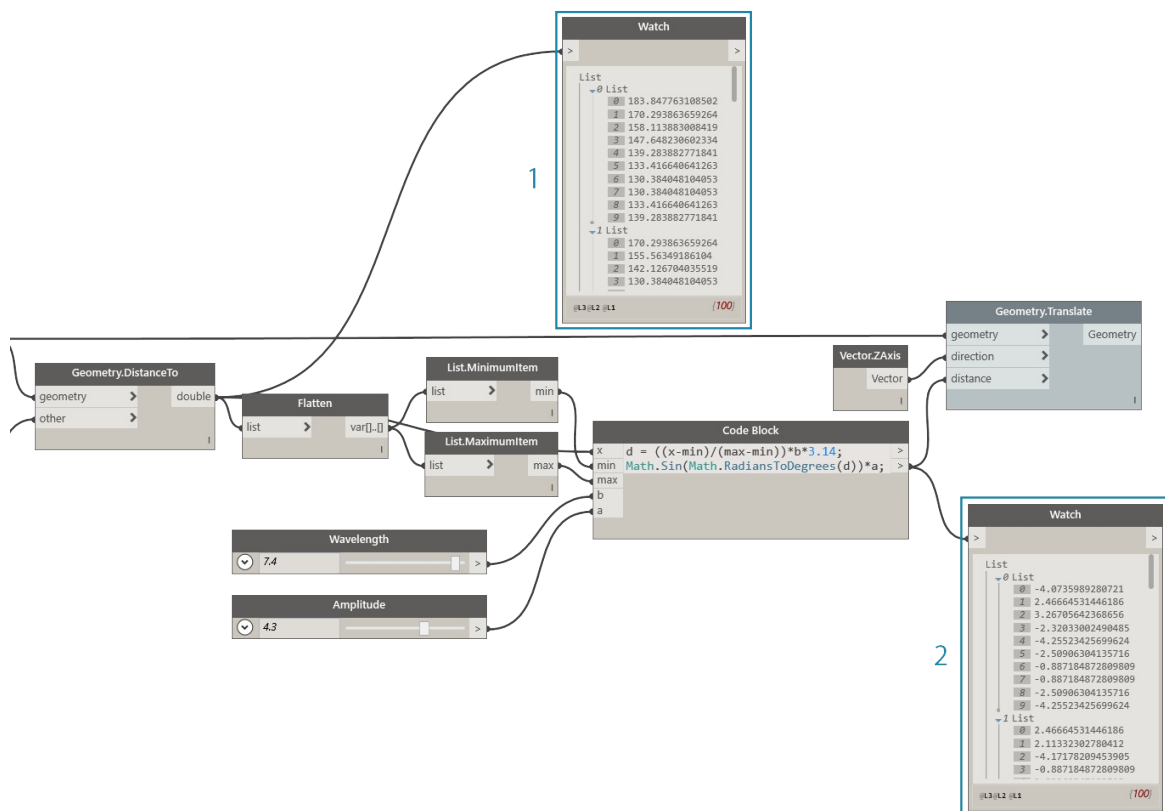
1. Una nota que describe la parte del programa que devuelve distancias de traslado sin procesar.
2. Una nota que describe el código que asigna dichos valores a una onda sinusoidal. Para obtener información sobre cómo añadir una nota, consulte [Administración del programa](#).

Ajuste continuo

Al crear la secuencia de comandos visual, es importante comprobar que se devuelve lo que se esperaba. No todos los errores o los problemas provocarán que el programa falle inmediatamente, sobre todo, los valores nulos o cero que podrían afectar a niveles inferiores. Esta estrategia también se aborda en el contexto de la creación de secuencias de comandos de texto en las [Estrategias de creación de secuencias de comandos](#). El siguiente procedimiento le ayudará a asegurarse de que obtiene lo que esperaba.

Supervisión de datos con burbujas de visualización y vista preliminar

- Utilice las burbujas de visualización y vista preliminar al crear el programa para **comprobar que las salidas clave devuelvan lo que esperaba**.



Los nodos de visualización se utilizan para comparar:

1. Las distancias de traslado sin procesar.
2. Los valores que se transfieren a través de la ecuación de seno. Para obtener información sobre cómo utilizar la visualización, consulte [Biblioteca](#).

Garantizar la reutilización

Es muy probable que otro usuario abra el programa en algún momento, incluso aunque trabaje de forma independiente. Este debería poder comprender rápidamente lo que el programa necesita y genera a partir de sus entradas y salidas. Esto es especialmente importante cuando se desarrolla un nodo personalizado que se compartirá con la comunidad de Dynamo y se utilizará en el programa de otro usuario. Estos procedimientos generan programas y nodos fiables y reutilizables.

Administración de E/S

- Para garantizar la legibilidad y la escalabilidad, debe probar y **minimizar las entradas y las salidas lo máximo posible**.
- Debe intentar **elaborar una estrategia de creación de la lógica mediante la elaboración inicial de un esquema aproximado** de cómo puede funcionar la lógica antes de añadir un único nodo al lienzo. A medida que desarrolle el esquema aproximado, debe realizar un seguimiento de las entradas y las salidas que se incluirán en las secuencias de comandos.

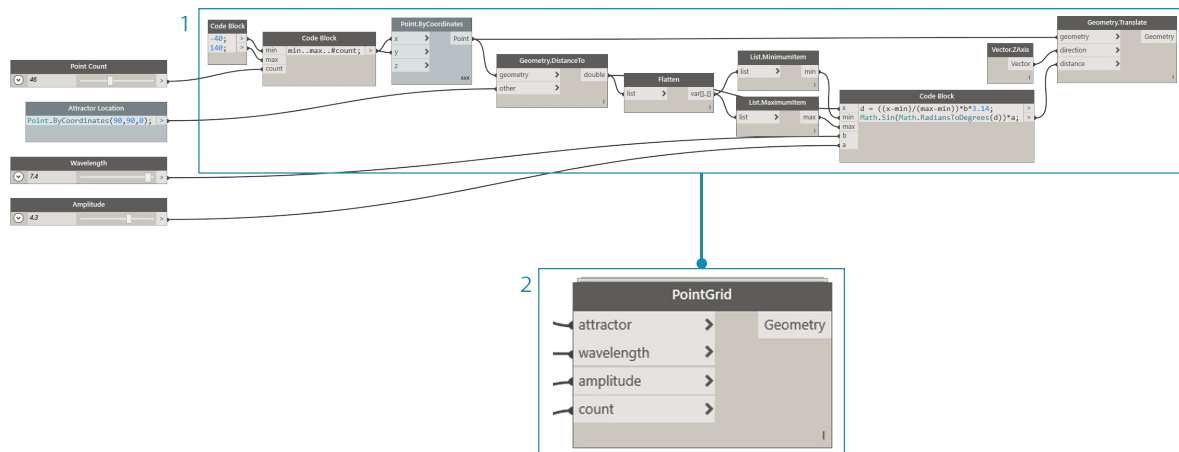
Uso de valores predefinidos para insertar valores de entrada

- Si desea insertar en el gráfico **condiciones u opciones específicas**, debe utilizar valores predefinidos para acceder a ellos rápidamente.
- También puede utilizar valores predefinidos para **reducir la complejidad mediante el almacenamiento en caché de los valores de controles deslizantes** de un gráfico con tiempos de ejecución prolongados.

Para obtener información sobre cómo utilizar los valores predefinidos, consulte [Administración de datos con valores predefinidos](#).

Inclusión de programas con nodos personalizados

- Debe utilizar un nodo personalizado si el **programa se puede agrupar en un único contenedor**.
- Debe utilizar un nodo personalizado **si una parte del gráfico se reutilizará a menudo** en otros programas.
- Debe utilizar un nodo personalizado si desea **compartir una función con la comunidad de Dynamo**.

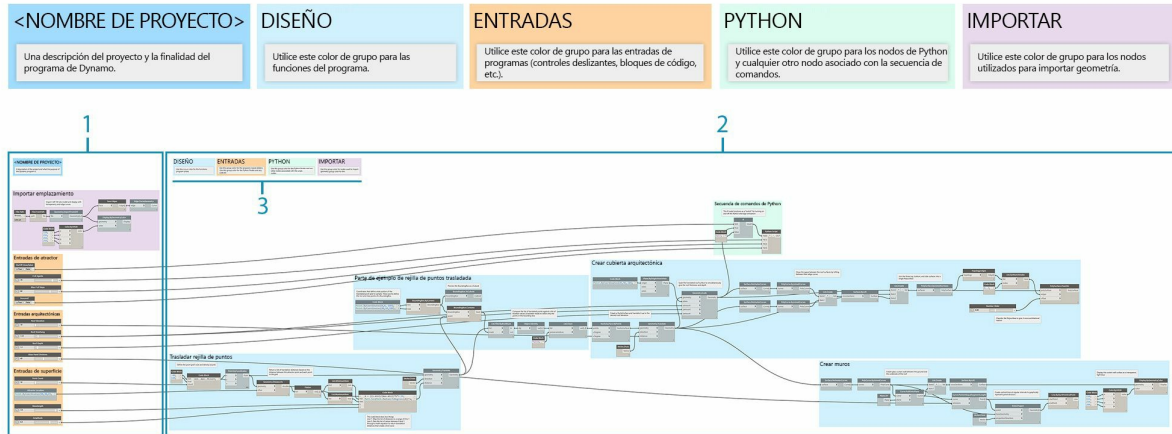


La recopilación del programa de traslado de puntos en un nodo personalizado consigue que un programa potente y exclusivo sea más fácil de comprender. Los puertos de entrada con nombres adecuados ayudarán a otros usuarios a comprender cómo utilizar el nodo. No olvide añadir descripciones y los tipos de datos necesarios para cada entrada.

1. Programa de atractor existente.
2. Nodo personalizado que recopila este programa, PointGrid. Para obtener información sobre cómo utilizar nodos personalizados, consulte [Introducción a los nodos personalizados](#).

Creación de plantillas

- Puede crear plantillas para **establecer normas gráficas en todos los gráficos visuales a fin de garantizar que los colaboradores dispongan de una forma normalizada de comprender el gráfico**.
- Al crear una plantilla, puede normalizar **los colores de grupo y los tamaños de tipo de letra** para organizar en categorías los flujos de trabajo o las acciones de datos.
- Al crear una plantilla, puede incluso normalizar el modo en que desea **utilizar una etiqueta, un color o un estilo para indicar la diferencia entre los flujos de trabajo front-end y back-end del gráfico**.

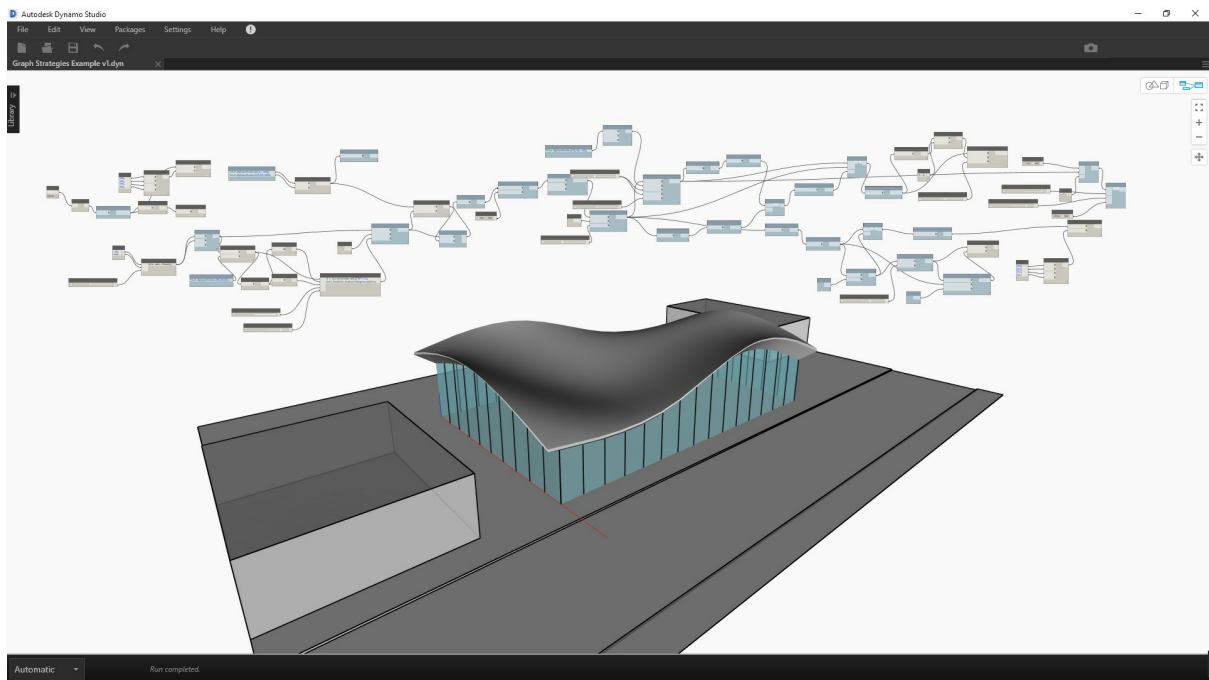


1. La interfaz de usuario, o front-end, del programa incluye un nombre de proyecto, controles deslizantes de entrada y geometría de importación.
2. El back-end del programa.
3. Categorías de colores de grupo (el diseño general, las entradas, las secuencias de comandos de Python y la geometría importada).

Ejercicio: cubierta arquitectónica

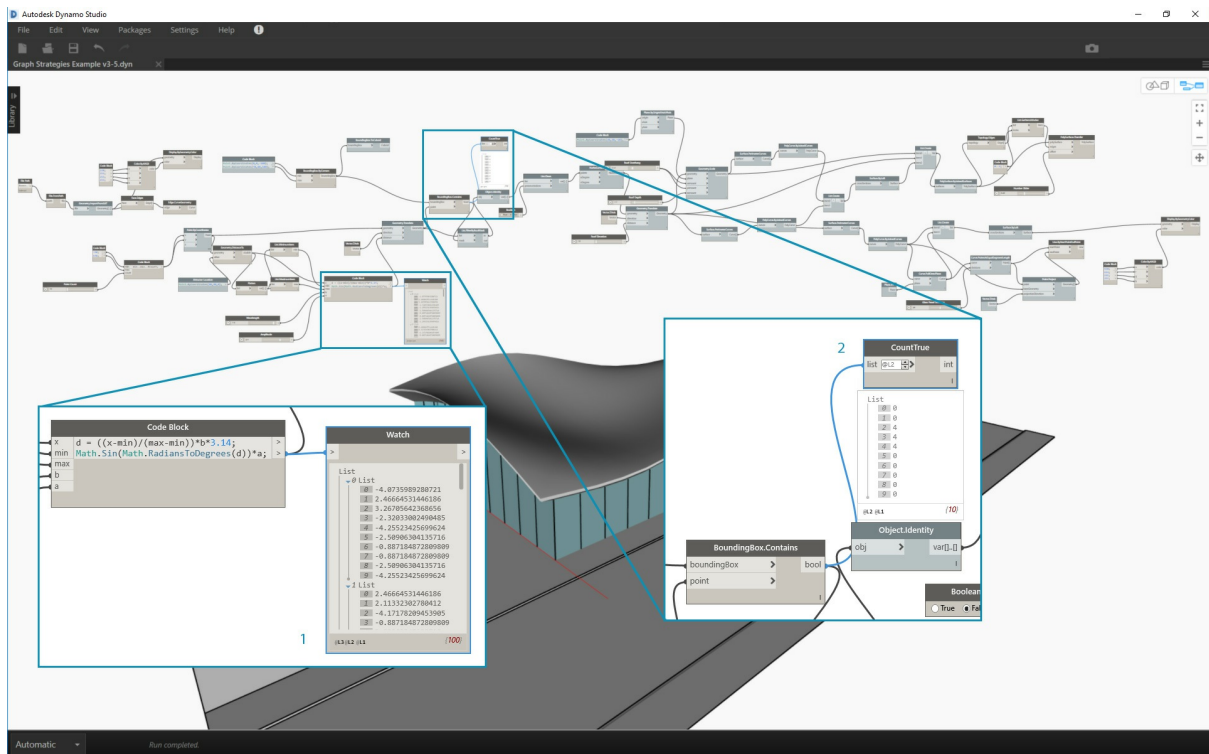
Descargue el archivo de ejemplo que acompaña a este ejercicio (haga clic con el botón derecho y seleccione "Guardar vínculo como..."). En el Apéndice, se incluye una lista completa de los archivos de ejemplo. [RoofDrainageSim.zip](#)

Ahora que hemos establecido algunos procedimientos recomendados, vamos a aplicarlos a un programa que hemos creado rápidamente. Aunque el programa genera correctamente la cubierta, el estado del gráfico es un "mapa mental" del autor. Carece de organización o una descripción de su uso. Recorreremos los procedimientos recomendados para organizar, describir y analizar el programa a fin de que otros usuarios puedan comprender cómo utilizarlo.



El programa funciona, pero el gráfico está desorganizado.

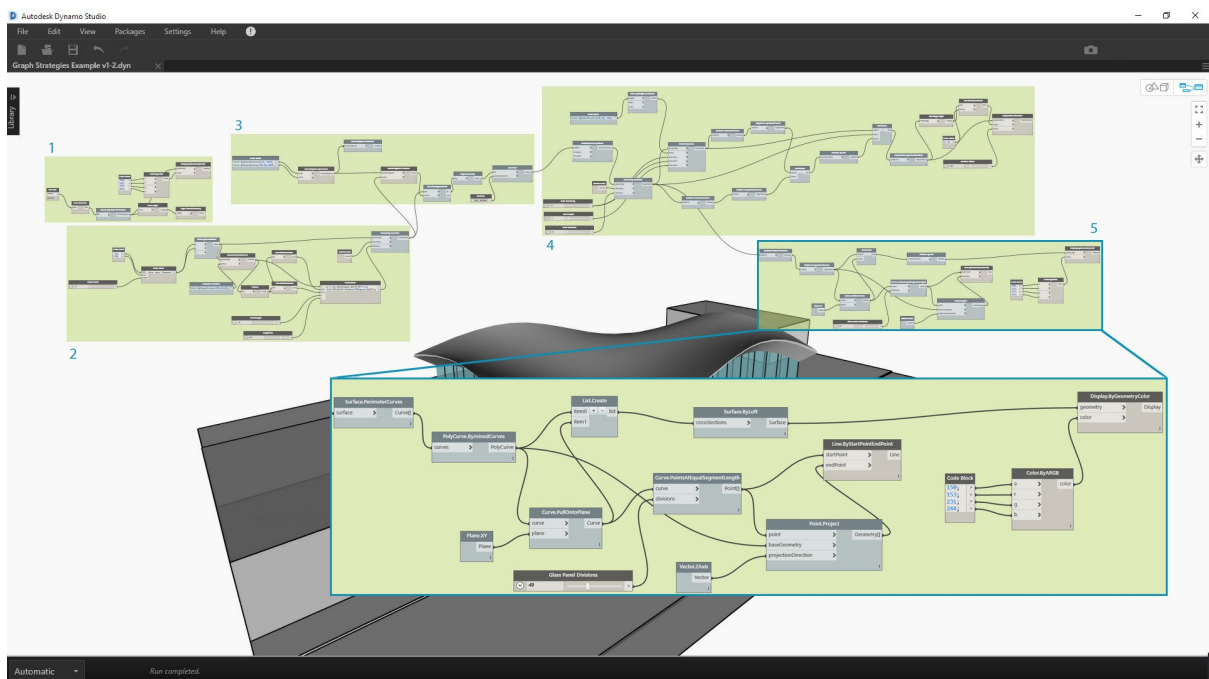
Comencemos por determinar los datos y la geometría devueltos por el programa.



Es crucial comprender cuándo se producen cambios importantes en los datos para establecer divisiones lógicas o modularidad. Pruebe a inspeccionar el resto del programa con nodos de visualización para ver si se pueden determinar los grupos antes de pasar al siguiente paso.

1. Este bloque de código con una ecuación matemática parece una parte crucial del programa. Un nodo de visualización indica que se devuelven listas de distancias de traslado.
2. La finalidad de esta área no es evidente a simple vista. La disposición de los valores "True" (verdadero) en el nivel de lista L2 de BoundingBox.Contains y la presencia de List.FilterByBoolMask sugiere que se están obteniendo muestras de una parte de la rejilla de puntos.

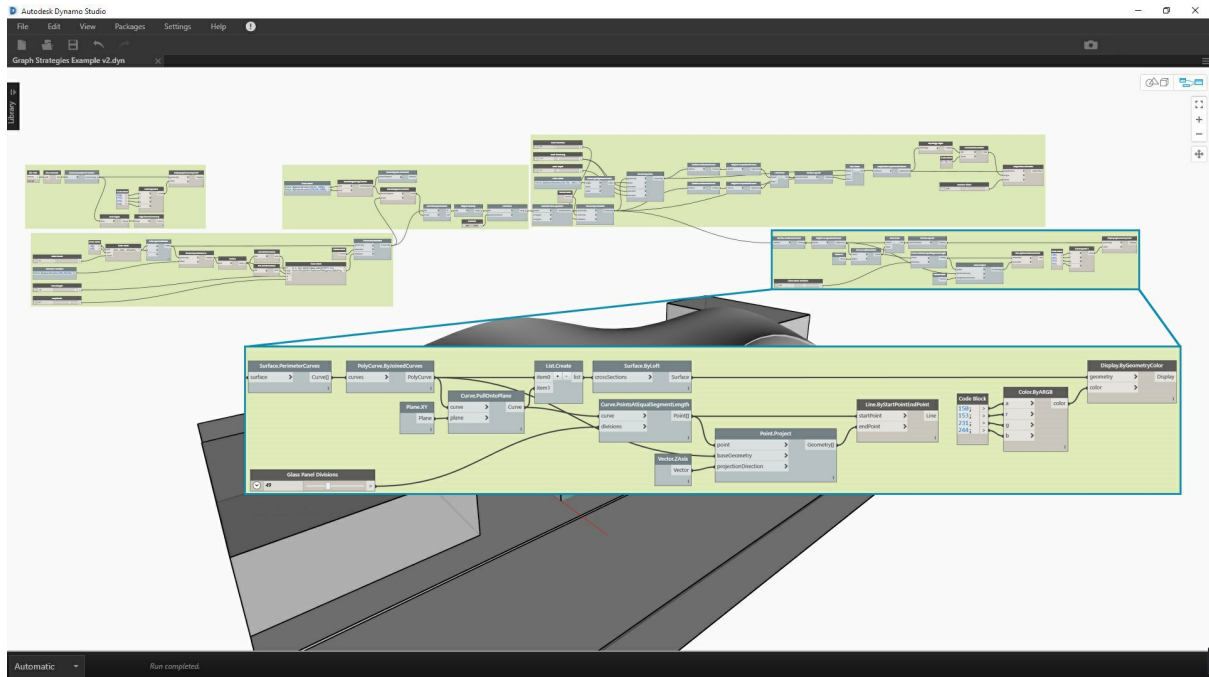
Una vez que entendemos las partes elementales del programa, organicémoslas en grupos.



Los grupos permiten al usuario diferenciar visualmente las partes del programa.

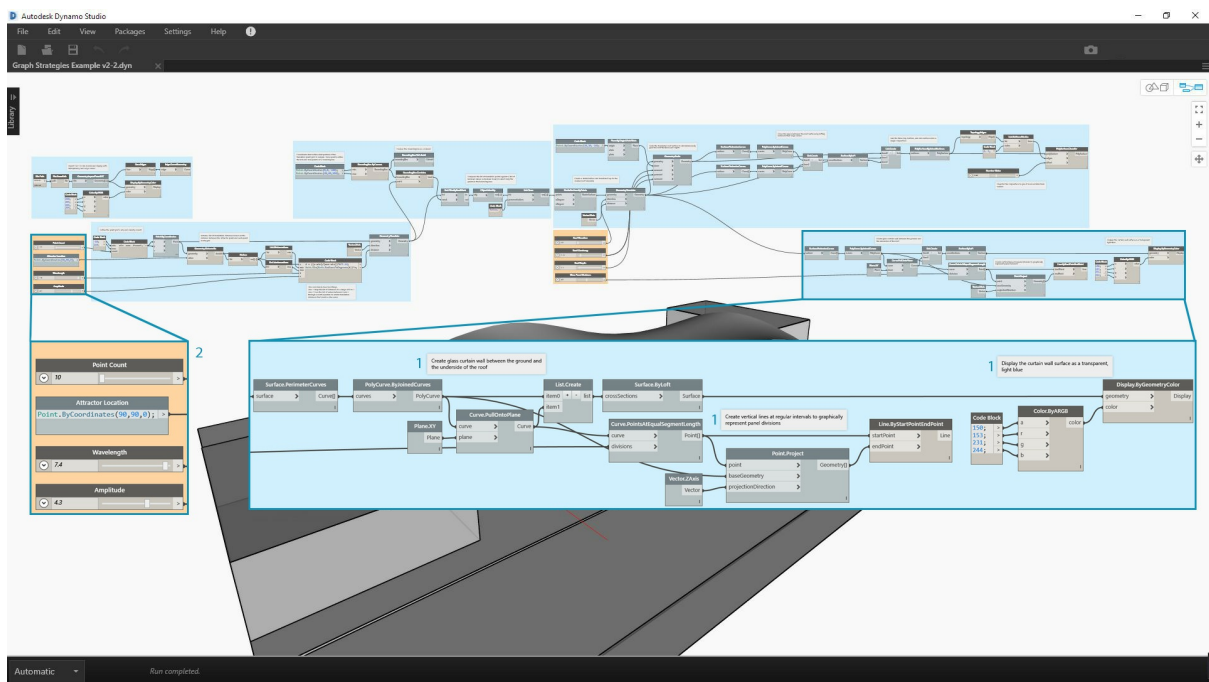
1. Importar el modelo de emplazamiento 3D.
2. Trasladar la rejilla de puntos en función de la ecuación de seno.
3. Obtener muestra de la rejilla de puntos.
4. Crear una superficie de cubierta arquitectónica.
5. Crear un muro cortina de vidrio.

Con los grupos establecidos, alinee los nodos para crear continuidad visual en el gráfico.



La continuidad visual ayuda al usuario a ver el flujo del programa y las relaciones implícitas entre nodos.

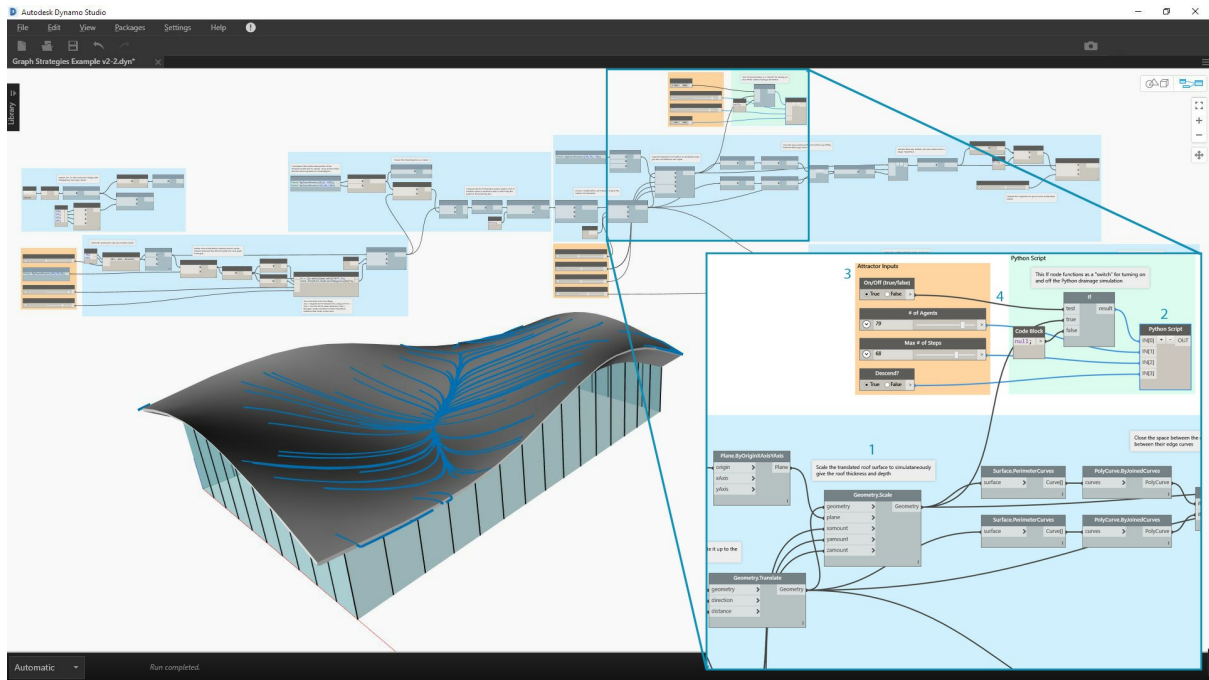
Consiga que el programa sea más accesible mediante la adición de otra capa de mejoras gráficas. Añada notas para describir cómo funciona un área específica del programa, proporcione nombres personalizados a las entradas y asigne colores a los distintos tipos de grupos.



Estas mejoras gráficas permiten al usuario obtener más información sobre lo que realiza el programa. Los diferentes colores de grupo ayudan a distinguir las entradas de las funciones.

1. Notas
2. Entradas con nombres descriptivos

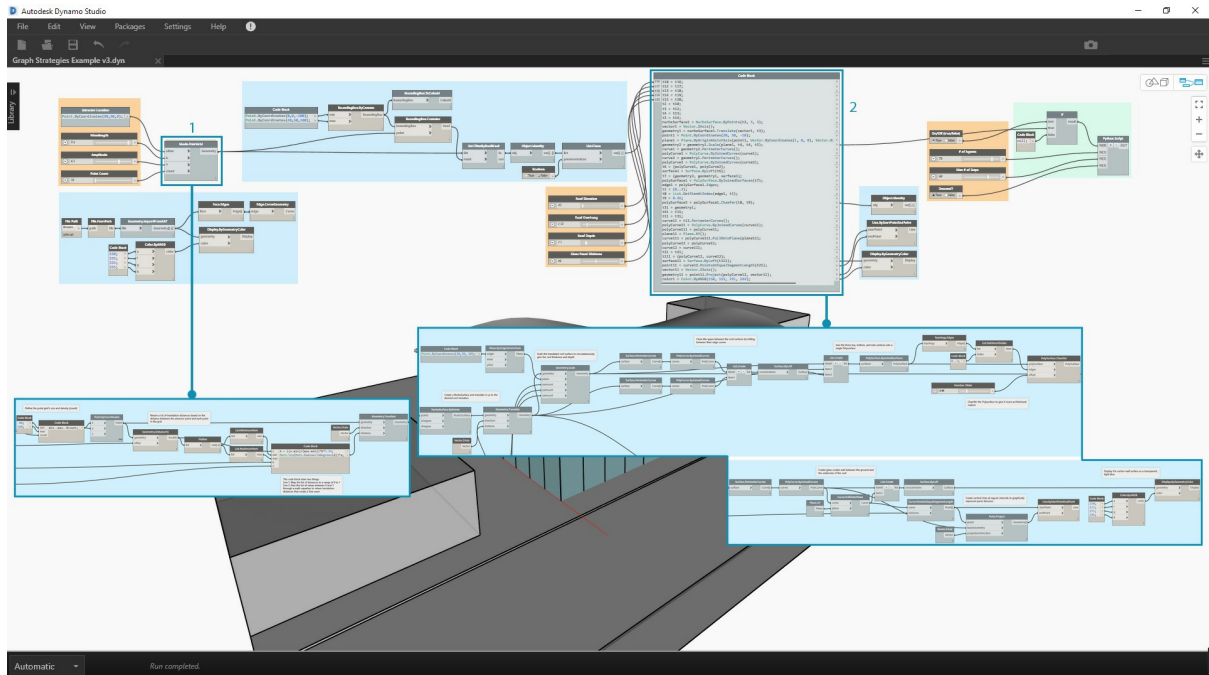
Antes de empezar a condensar el programa, busquemos una ubicación estratégica para introducir el simulador de drenaje de la secuencia de comandos de Python. Conecte la salida de la primera superficie de cubierta a escala con la entrada de secuencias de comandos respectiva.



Hemos optado por integrar secuencias de comandos en este punto del programa para que la simulación de drenaje pueda ejecutarse en la superficie de cubierta original. No se está previsualizando esa superficie específica, pero esto nos evita tener que elegir la superficie superior de la PolySurface achaflanada.

1. Geometría de origen para la entrada de secuencia de comandos
2. Nodo de Python
3. Controles deslizantes de entrada
4. "Conmutador" de activación/desactivación

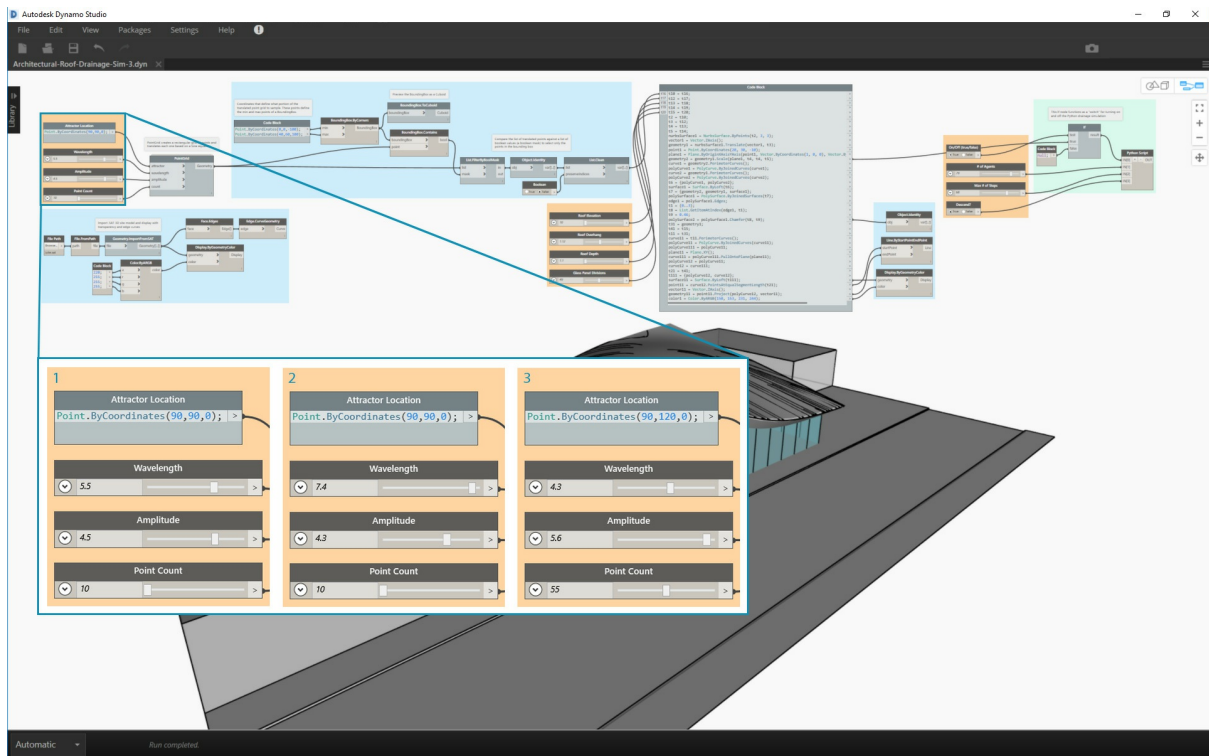
Ahora que todo está en su lugar, vamos a simplificar el gráfico.



La condensación del programa con el método de nodo a código y el nodo personalizado ha reducido considerablemente el tamaño del gráfico. Los grupos que crean la superficie de cubierta y los muros se han convertido en código, ya que son muy específicos de este programa. El grupo de traslado de puntos se encuentra en un nodo personalizado, ya que podría utilizarse en otro programa. En el archivo de ejemplo, cree su propio nodo personalizado a partir del grupo de puntos de traslado.

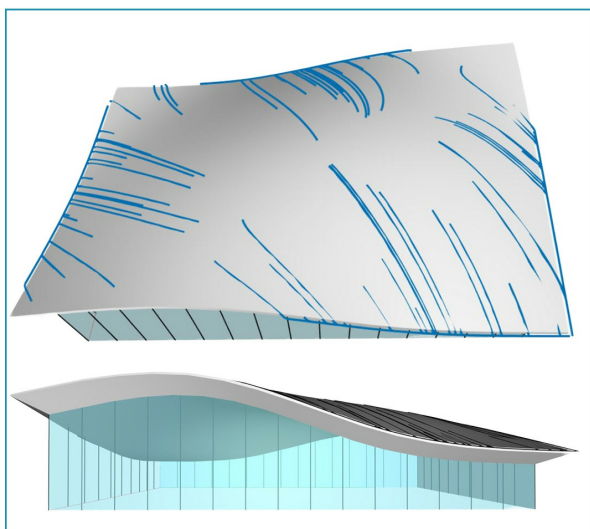
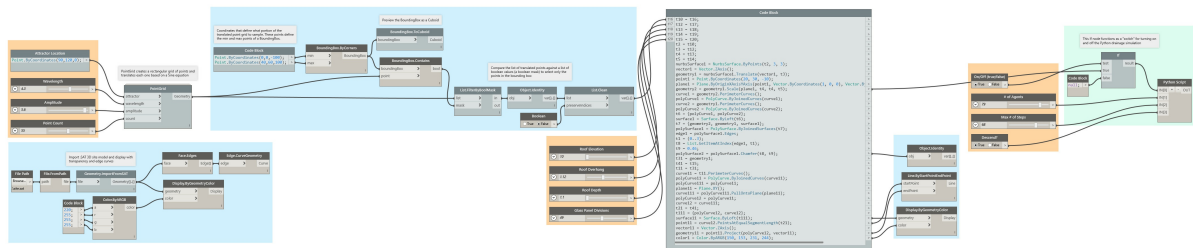
1. Nodo personalizado que contiene el grupo de "rejilla de puntos de traslado".
2. Método de nodo a código para condensar los grupos de "crear superficie de cubierta arquitectónica y muro cortina".

Como paso final, cree valores predefinidos para formas de cubierta de ejemplo.

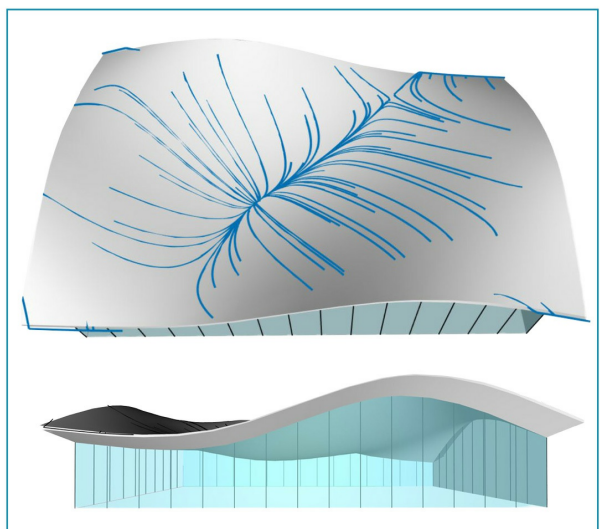


Estas entradas son los controladores principales de la forma de cubierta y ayudarán a los usuarios a ver el potencial del programa.

Nuestro programa con vistas de dos valores predefinidos.



1



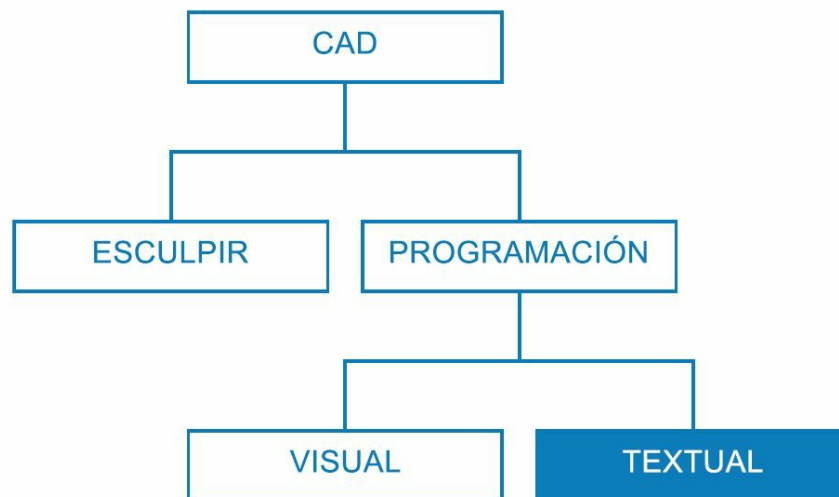
2

Los patrones de drenaje de la cubierta proporcionan al usuario una vista analítica de los valores predefinidos respectivos.

Estrategias de creación de secuencias de comandos

Estrategias de creación de secuencias de comandos

Las secuencias de comandos basadas en texto en el entorno de creación de secuencias de comandos visual permite relaciones visuales y potentes mediante DesignScript, Python y ZeroTouch (C#). El usuario puede visualizar elementos como, por ejemplo, controles deslizantes de entrada, condensar operaciones grandes en DesignScript y acceder a herramientas y bibliotecas eficaces a través de Python o C# en el mismo espacio de trabajo. Si se administran de manera eficaz, la combinación de estas estrategias puede proporcionar una gran cantidad de personalización, claridad y eficacia a todo el programa. A continuación, se ofrece un conjunto de directrices para ayudarle a aumentar la secuencia de comandos visual con la secuencia de comandos de texto.



Cuándo crear secuencias de comandos

La creación de secuencias de comandos de texto puede establecer relaciones de una mayor complejidad que la programación visual, aunque sus funciones también se solapan considerablemente. Esto es así porque los nodos son código compilado previamente de forma eficaz y es probable que podamos escribir un programa de Dynamo completo en DesignScript o Python. Sin embargo, utilizamos la creación de secuencias de comandos visuales porque la interfaz de nodos y cables crea un flujo intuitivo de información gráfica. Saber dónde las funciones de creación de secuencias de comandos de texto superan a la creación de secuencias de comandos visuales le proporcionará importantes pistas sobre cuándo deben utilizarse sin renunciar a la naturaleza intuitiva de los nodos y los cables. A continuación, se proporcionan directrices sobre cuándo crear una secuencia de comandos y qué lenguaje se debe utilizar.

Utilice secuencias de comandos de texto para:

- Creación de bucles
- Repetición
- Acceso a bibliotecas externas

Elija un lenguaje:

	Creación de bucles	Repetición	Condensar nodos	Bibliotecas ext.	Abreviatura
DesignScript	Sí	Sí	Sí	No	Sí
Python	Sí	Sí	Parcialmente	Sí	No
ZeroTouch (C#)	No	No	No	Sí	No

Consulte [Referencia de secuencias de comandos](#) para obtener una lista de los elementos a los que cada biblioteca de Dynamo proporciona acceso.

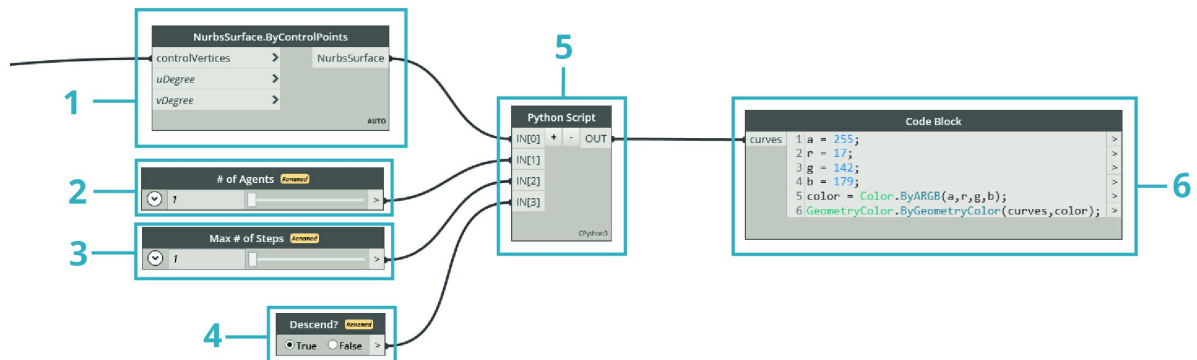
Pensamiento paramétrico

Al crear secuencias de comandos en Dynamo, un entorno inevitablemente paramétrico, es aconsejable estructurar el código en relación con la estructura de nodos y cables en los que se basará. Considere el nodo que contiene la secuencia de comandos de texto como si se tratara de cualquier otro nodo del programa con unas pocas entradas específicas, una función y una salida prevista. Esto proporciona al instante al código del nodo un conjunto de variables desde las que trabajar, la clave para obtener un sistema paramétrico limpio. A continuación, se proporcionan algunas directrices para una integración más eficaz del código en un programa visual.

Identifique las variables externas:

- Intente determinar los parámetros especificados en el problema de diseño para que pueda construir un modelo que se base directamente en esos datos.
- Antes de escribir el código, identifique las variables:
 - Un conjunto mínimo de entradas

- o La salida deseada
- o Constantes



Se han establecido varias variables antes de escribir el código.

1. La superficie en la que vamos a simular la lluvia.
2. El número de gotas de lluvia (agentes) que deseamos.
3. Hasta dónde deseamos que se desplacen las gotas de lluvia.
4. Alterne entre descender la ruta más escarpada o atravesar la superficie.
5. El nodo de Python con el número respectivo de entradas.
6. Un bloque de código para establecer en azul las curvas devueltas.

Diseñe las relaciones internas:

- El uso de parámetros permite editar determinados parámetros o variables para manipular o modificar el resultado final de una ecuación o un sistema.
- Siempre que las entidades de la secuencia de comandos estén relacionadas lógicamente, el objetivo es definirlas como funciones entre sí. De este modo, al modificar una, la otra se puede actualizar de forma proporcional.
- Minimice el número de entradas visualizando solo los parámetros clave:
 - o Si un conjunto de parámetros se puede derivar de más parámetros principales, solo se mostrarán los parámetros principales como entradas de la secuencia de comandos. Esto aumenta la facilidad de uso de la secuencia de comandos al reducir la complejidad de su interfaz.

```

1 import sys
2 import clr
3 clr.AddReference('ProtoGeometry')
4 from Autodesk.DesignScript.Geometry import *
5
6 solid = IN[0]
7 seed = IN[1]
8 xCount = IN[2]
9 yCount = IN[3]
10
11 solids = []
12
13 yDist = solid.BoundingBox.MaxPoint.Y - solid.BoundingBox.MinPoint.Y
14 xDist = solid.BoundingBox.MaxPoint.X - solid.BoundingBox.MinPoint.X
15
16 for i in xrange:
17     for j in xrange:
18         fromCoord = solid.ContextCoordinateSystem
19         toCoord = fromCoord.Rotate(solid.ContextCoordinateSystem.Origin, Vector.ByCoordinates(0, 0, 1), 90*(i+j%val)))
20         vec = Vector.ByCoordinates(xDist*i, yDist*j, 0)
21         toCoord = toCoord.Translate(vec)
22         solids.append(solid.Transform(fromCoord, toCoord))
23
24 OUT = solids

```

El código "modules" del ejemplo de [Nodo de Python](#).

1. Entradas.
2. Variables internas de la secuencia de comandos.
3. Un bucle que utiliza estas entradas y variables para realizar su función. Consejo: céntrese tanto en el proceso como lo hace en la solución.

No se repita (principio de no repetición):

- Cuando hay varias formas de expresar lo mismo en la secuencia de comandos, en algún momento, las representaciones duplicadas quedarán fuera de la sincronización, lo que puede llevar a problemas de mantenimiento, factorización incorrecta y contradicciones internas.
- El principio de no repetición (DRY, Don't Repeat Yourself) se define como "Cada parte del conocimiento debe tener una representación única, sin ambigüedades y autorizada dentro de un sistema":
 - Si se aplica correctamente este principio, todos los elementos relacionados de la secuencia de comandos cambiarán de forma predecible y uniforme, y todos los elementos no relacionados no presentarán consecuencias lógicas entre sí.

```

### 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)

```

Consejo: antes de duplicar entidades en la secuencia de comandos (como la constante en el ejemplo anterior), pregúntese si puede establecer un vínculo con el origen en su lugar.

Estructuración modular

A medida que aumenta la longitud y la complejidad del código, la "gran idea" (o el algoritmo general) se vuelve cada vez más ilegible. También resulta más difícil realizar un seguimiento de los eventos y su ubicación, identificar errores cuando hay problemas, integrar otro código y asignar tareas de desarrollo. Para evitar estos problemas, es aconsejable escribir código en módulos, una estrategia organizativa que divide el código en función en la tarea que ejecuta. A continuación, se incluyen algunos consejos para que las secuencias de comandos sean más manejables a través de la modularización.

Escribir código en módulos:

- Un "módulo" es un grupo de código que realiza una tarea específica, similar a un nodo de Dynamo en el espacio de trabajo.
- Puede tratarse de cualquier elemento que se deba separar visualmente del código adyacente (una función, una clase, un grupo de entradas o las bibliotecas que se están importando).
- El desarrollo de código en módulos aprovecha la calidad visual e intuitiva de los nodos, así como las relaciones complejas que solo pueden obtenerse mediante secuencias de comandos de texto.

```

1 import sys
2 import clr
3 clr.AddReference('ProtoGeometry')
4 from Autodesk.DesignScript.Geometry import *
5
6 solid = IN[0]
7 seed = IN[1]
8 xCount = IN[2]
9 yCount = IN[3]
10
11 solids = []
12
13 yDist = solid.BoundingBox.MaxPoint.Y - solid.BoundingBox.MinPoint.Y
14 xDist = solid.BoundingBox.MaxPoint.X - solid.BoundingBox.MinPoint.X
15
16 for i in xRange:
17     for j in yRange:
18         fromCoord = solid.ContextCoordinateSystem
19         toCoord = fromCoord.Rotate(solid.ContextCoordinateSystem.Origin, Vector.ByCoordinates(0,0,1), 90*(i+j%val)))
20         vec = Vector.ByCoordinates((xDist*i), (yDist*j), 0)
21         toCoord = toCoord.Translate(vec)
22         solids.append(solid.Transform(fromCoord, toCoord))
23
24 OUT = solids

```

Estos bucles llaman a una clase denominada "agente" que desarrollaremos más detalladamente en el ejercicio.

1. Un módulo de código que define el punto inicial de cada agente.
2. Un módulo de código que actualiza el agente.
3. Un módulo de código que dibuja una trayectoria para la ruta del agente.

Detección de la reutilización de código:

- Si detecta que el código realiza la misma tarea (o una muy similar) en más de una ubicación, busque formas de agruparlo en una función a la que se pueda llamar.
- Las funciones de "administrador" controlan el flujo del programa y contienen principalmente llamadas a funciones de "trabajo" que gestionan detalles de bajo nivel, como desplazar datos entre estructuras.



En este ejemplo, se crean esferas con radios y colores basados en el valor Z de los centros.

1. Dos funciones principales de "trabajo": una que crea esferas con radios y colores de visualización en función del valor Z del centro.
2. Una función principal de "administrador" que combina las dos funciones de trabajo. Al llamar a esta, se llamará a las dos funciones incluidas en ella.

Muestre solo lo que se necesite ver:

- Una interfaz de módulo indica los elementos que el módulo proporciona y necesita.
- Una vez definidas las interfaces entre las unidades, el diseño detallado de cada unidad puede continuar por separado.

Capacidad de separación/sustitución:

- Los módulos no se conocen ni se preocupan entre sí.

Formas generales de modularización:

- Agrupación de código:

```

# 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]

```

- Funciones:

```

def get_step_size():
    area = surfIn.Area
    stepSize = math.sqrt(area)/100
    return stepSize

```

```

stepSize = get_step_size()

```

- Clases:

```

class MyClass:
    i = 12345

```

```

def f(self):
    return 'hello world'

```

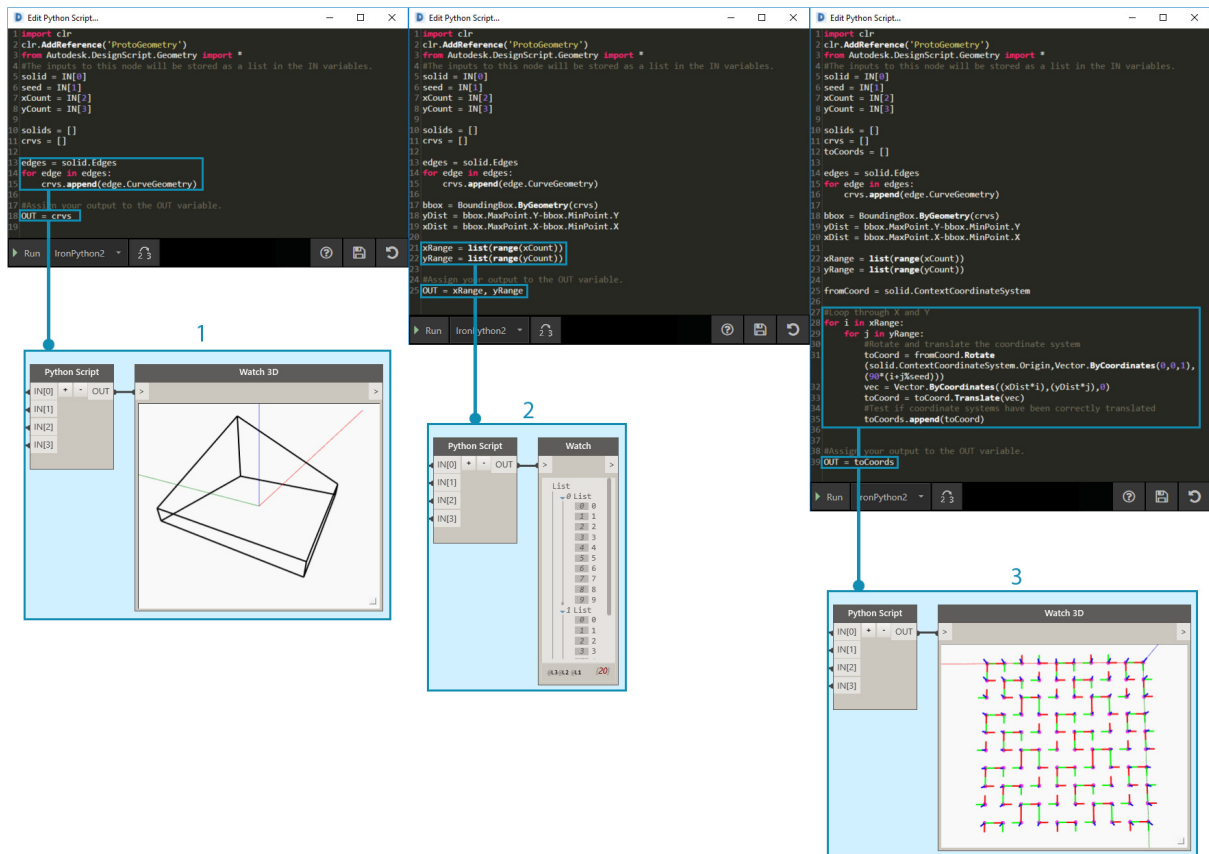
```
numbers = MyClass.i  
greeting = MyClass.f
```

Ajuste continuo

Al desarrollar secuencias de comandos de texto en Dynamo, es recomendable asegurarse constantemente de que lo que se crea en realidad está en consonancia con lo que se espera. Esto garantizará que los eventos imprevistos, como errores de sintaxis, discrepancias lógicas, valores imprecisos y salidas anómalas, entre otros, se detecten y se aborden rápidamente a medida que aparecen en lugar de todos al final. Como las secuencias de comandos se encuentran dentro de los nodos en el lienzo, ya están integradas en el flujo de datos del programa visual. Esto permite que la supervisión continua de la secuencia de comandos sea tan sencilla como asignar los datos que se van a obtener, ejecutar el programa y evaluar los flujos externos a la secuencia de comandos que utilizan un nodo de visualización. A continuación, se proporcionan algunos consejos para inspeccionar de forma continua las secuencias de comandos a medida que las crea.

Realice pruebas durante el proceso:

- Cuando complete un grupo de funciones:
 - Retroceda y revise el código.
 - Sea crítico. ¿Podría un colaborador entender la finalidad del código? ¿Es necesario que haga esto? ¿Esta función se puede realizar de forma más eficaz? ¿Estoy creando dependencias o duplicados innecesarios?
 - Realice pruebas rápidas para asegurarse de que se devuelven datos que "tienen sentido".
- Asigne los datos más recientes con los que está trabajando en la secuencia de comandos como salida para que el nodo siempre presente datos pertinentes cuando se actualice la secuencia de comandos:



Ajuste del código de ejemplo del [nodo de Python](#).

1. Compruebe que todos los bordes del sólido se devuelvan como curvas para crear un cuadro delimitador alrededor.
2. Compruebe que las entradas de recuento se hayan convertido correctamente en intervalos.
3. Compruebe que los sistemas de coordenadas se hayan trasladado y rotado correctamente en este bucle.

Anticipar "casos extremos":

- Durante la creación de secuencias de comandos, establezca los parámetros de entrada en los valores mínimo y máximo del dominio asignado para comprobar si el programa sigue funcionando en condiciones extremas.
- Aunque el programa funcione en los extremos, compruebe si devuelve valores nulos, vacíos o cero no deseados.
- En algunos casos, los errores y las incidencias que revelan algún problema subyacente en la secuencia de comandos solo se detectarán durante estos casos extremos.
 - Conozca las causas del error y decida posteriormente si se debe corregir internamente o si se debe volver a definir un dominio de parámetro para evitar el problema.

Consejo: siempre presuponga que el usuario utilizará todas las combinaciones de cada uno de los valores de entrada que se le hayan presentado. Esto ayudará a eliminar sorpresas no deseadas.

Depuración eficaz

La depuración es el proceso de eliminar "errores" de la secuencia de comandos. Los errores pueden ser incidencias, aspectos ineficaces, imprecisiones o resultados no deseados. La resolución de errores puede ser tan sencilla como corregir un nombre de variable mal escrito hasta hacer frente a problemas generalizados y estructurales en la secuencia de comandos. Lo ideal es que el ajuste de la secuencia de comandos durante el proceso de creación le ayude a detectar estos problemas potenciales en una fase inicial, aunque esto no garantiza que esté libre de errores. A continuación, se revisan varios procedimientos recomendados indicados anteriormente para ayudar a resolver los errores de forma sistemática.

Utilice la burbuja de visualización:

- Compruebe los datos devueltos en diferentes partes del código. Para ello, asíguelos a la variable OUT, de modo similar al proceso de ajuste del programa.

Escriba comentarios descriptivos:

- Un módulo de código será mucho más fácil de depurar si se describe con claridad el resultado previsto.

```
# Loop through X and Y
for i in range(xCount):
for j in range(yCount):
```

```
# Rotate and translate the coordinate system
toCoord = fromCoord.Rotate(solid.ContextCoordinateSystem.Origin, Vector.ByCoordinates(0,0,1), (90*(i+j%sec
```



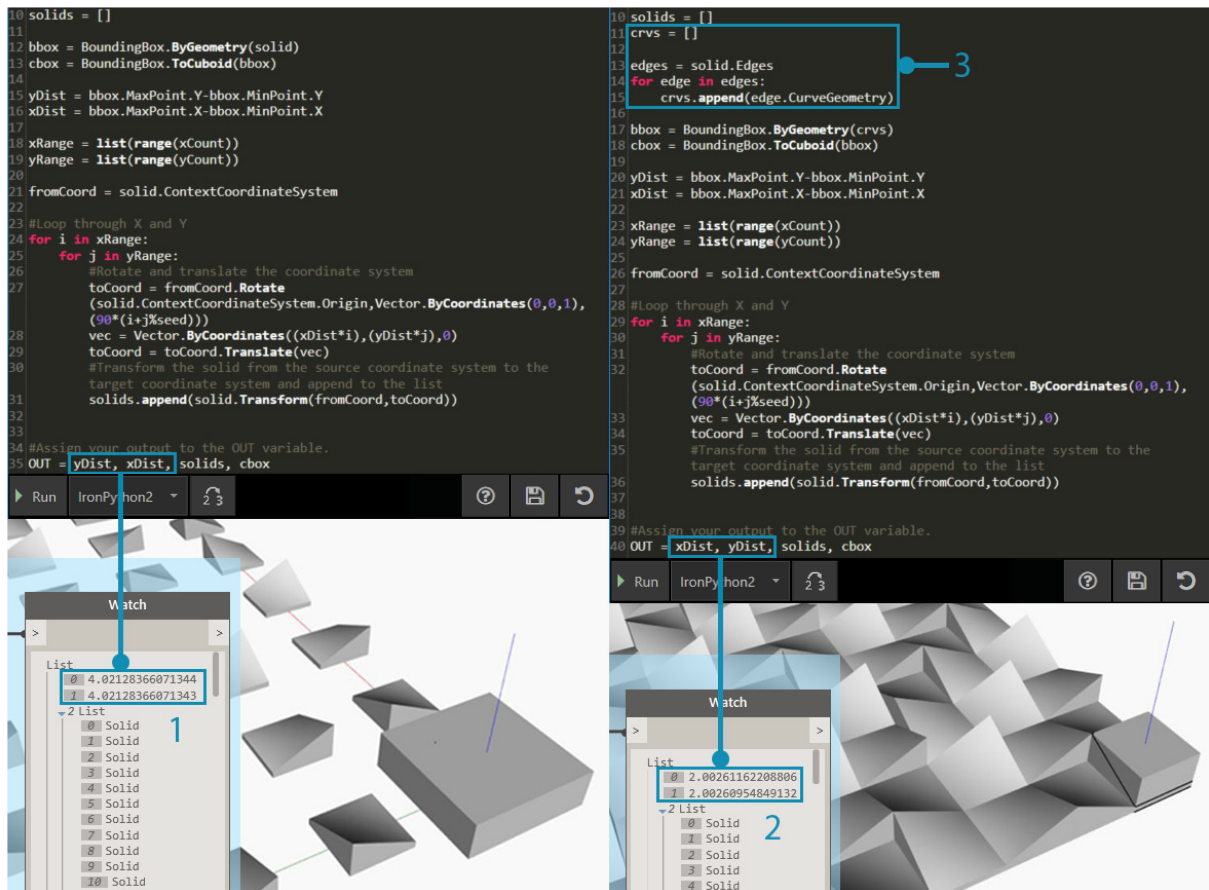
```
vec = Vector.ByCoordinates((xDist*i), (yDist*j), 0)
toCoord = toCoord.Translate(vec)
```

```
# Transform the solid from the source coord system to the target coord system and append to the list
solids.append(solid.Transform(fromCoord, toCoord))
```

Por lo general, esto supone una cantidad excesiva de comentarios y líneas en blanco, pero, cuando se depura, puede ser útil dividir la información en partes manejables.

Aproveche la modularidad del código:

- El origen de un problema se puede aislar en determinados módulos.
- Una vez identificado el módulo defectuoso, se puede corregir de forma mucho más fácil el problema.
- Cuando se debe modificar un programa, el código desarrollado en los módulos será mucho más fácil de cambiar:
 - Puede insertar módulos nuevos o depurados en un programa existente con la seguridad de que el resto del programa no cambiará.



Depuración del archivo de ejemplo del [nodo de Python](#).

1. La geometría de entrada devuelve un cuadro delimitador de mayor tamaño, como podemos ver si se asigna xDist e yDist a OUT.
2. Las curvas de borde de la geometría de entrada devuelven un cuadro delimitador adecuado con las distancias correctas para xDist e yDist.
3. El "módulo" de códigos que hemos insertado para solucionar el problema del valor de xDist e yDist.

Ejercicio: la ruta más escarpada

Descargue el archivo de ejemplo que acompaña a este ejercicio (haga clic con el botón derecho y seleccione "Guardar vínculo como..."). En el Apéndice, se incluye una lista completa de los archivos de ejemplo. [SteepestPath.dyn](#)

Teniendo en cuenta los procedimientos recomendados de creación de secuencias de comandos de texto, vamos a escribir una secuencia de comandos de simulación de lluvia. Aunque hemos podido aplicar los procedimientos recomendados a un programa visual desorganizado en Estrategias gráficas, es mucho más difícil hacerlo con el lenguaje de secuencias de comandos de texto. Las relaciones lógicas establecidas en las secuencias de comandos de texto son menos visibles y pueden ser casi imposibles de desentrañar en un código complicado. La eficacia que proporciona la creación de secuencias de comandos de texto también conlleva una mayor responsabilidad en cuanto a la organización. Recorreremos cada paso y aplicaremos los procedimientos recomendados durante el proceso.

☐

La secuencia de comandos se aplica a una superficie deformada por un atractor.

Lo primero que debemos hacer es importar las bibliotecas de Dynamo necesarias. Esto proporcionará acceso global a las funciones de Dynamo en Python.

☐

Todas las bibliotecas que tenemos intención de utilizar deben importarse aquí.

A continuación, debemos definir las entradas y las salidas de la secuencia de comandos, que se mostrarán como puertos de entrada en el nodo. Estas entradas externas son la base de nuestra secuencia de comandos y la clave para establecer un entorno paramétrico.

☐

Se deben definir entradas que se correspondan con variables en la secuencia de comandos de Python y determinar la salida deseada:

1. La superficie que deseamos descender.
2. El número de agentes que deseamos que realicen el recorrido.
3. El número máximo de pasos que pueden dar los agentes.
4. Una opción para tomar la ruta descendente más corta por la superficie o para atravesarla.
5. El nodo de Python con identificadores de entrada que corresponden a entradas de la secuencia de comandos (IN[0], IN[1]).
6. Las curvas de salida que se pueden mostrar con un color diferente.

Ahora utilicemos el procedimiento de modularidad y creemos el cuerpo de nuestra secuencia de comandos. La simulación de la ruta descendente más corta

por una superficie para varios puntos iniciales es una tarea importante que requerirá varias funciones. En lugar de llamar a las diferentes funciones a lo largo de la secuencia de comandos, podemos modularizar el código recopilándolas en una sola clase, nuestro agente. Se puede llamar a las diferentes funciones de esta clase o "módulo" con distintas variables, o incluso volver a utilizarlas en otra secuencia de comandos.



Necesitaremos definir una clase o un plan de acción para un agente con la intención de descender la superficie optando por el desplazamiento en la dirección más escarpada posible cada vez que este dé un paso:

1. Nombre.
2. Atributos globales que comparten todos los agentes.
3. Atributos de ejemplar exclusivos de cada agente.
4. Una función para dar un paso.
5. Una función para catalogar la posición de cada paso en una lista de trayectorias.

Inicializaremos los agentes mediante la definición de su ubicación inicial. Esta es una buena oportunidad para ajustar nuestra secuencia de comandos y asegurarnos de que funciona la clase de agente.



Tendremos que crear una instancia de todos los agentes cuyo descenso por la superficie deseamos observar y definir sus atributos iniciales:

1. Una nueva lista de trayectorias vacía.
2. La ubicación en la que se iniciará el paseo por la superficie.
3. Hemos asignado la lista de agentes como salida para comprobar lo que devuelve la secuencia de comandos aquí. Se devuelve el número correcto de agentes, pero necesitaremos ajustar de nuevo la secuencia de comandos más adelante para comprobar la geometría que devuelve.

Actualice cada agente en cada paso.



A continuación, deberemos introducir un bucle anidado en el que, para cada agente y paso, actualizaremos y registraremos la posición en la lista de trayectorias. En cada paso, también nos aseguraremos de que el agente no haya alcanzado un punto de la superficie donde no pueda dar otro paso, lo que le permitirá descender. Si se cumple esa condición, finalizaremos el desplazamiento del agente.

Ahora que se han actualizado por completo los agentes, volvamos a la geometría que los representa.



Una vez que todos los agentes hayan alcanzado su límite de descenso o su número máximo de pasos, crearemos una PolyCurve a través de los puntos de la lista de trayectorias y generaremos las trayectorias de PolyCurve.

Nuestra secuencia de comandos para buscar las rutas más escarpadas.



1. Un valor predefinido que simula la lluvia en la superficie subyacente.
2. En lugar de buscar la ruta más escarpada, se pueden activar o desactivar los agentes para que atraviesen la superficie subyacente.



La secuencia de comandos de texto completa de Python.

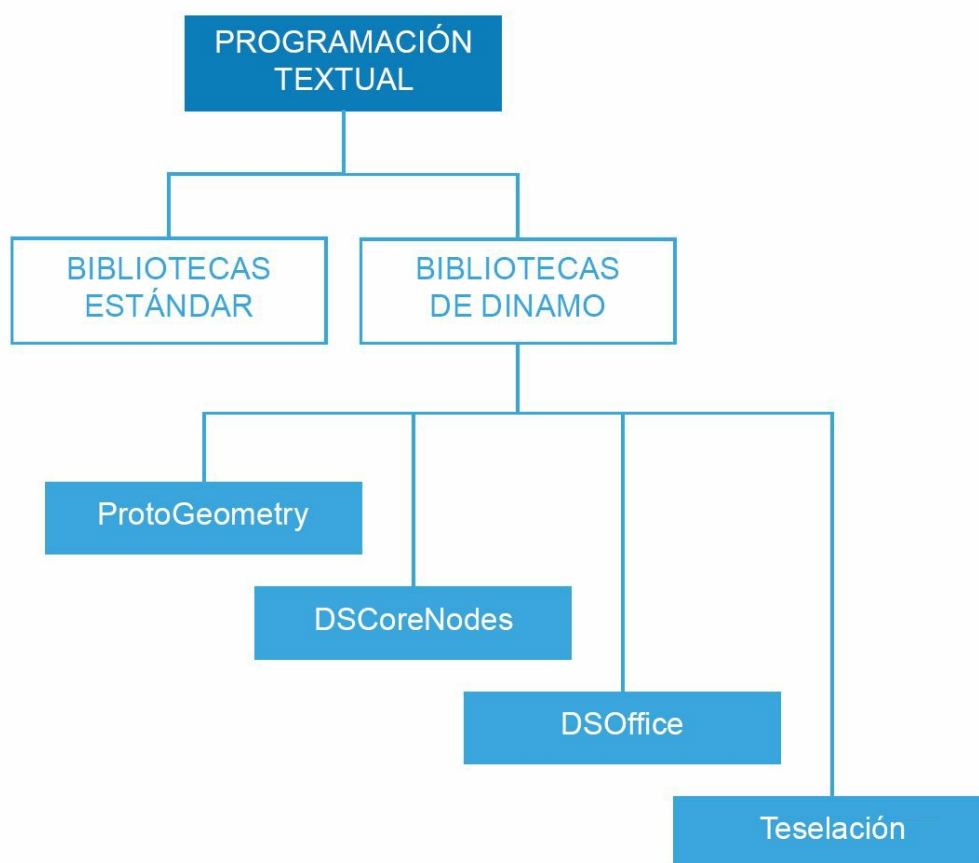
Referencia de secuencias de comandos

Referencia de secuencias de comandos

Esta página de referencia amplía las prácticas recomendadas incluidas en las estrategias de creación de secuencias de comandos con más detalles sobre las bibliotecas de códigos, el etiquetado y la aplicación de estilos. Utilizaremos Python para ilustrar los conceptos que se indican a continuación; no obstante, tenga en cuenta que se aplican los mismos principios en Python y C#(Zerotouch), solo que con una sintaxis diferente.

Qué bibliotecas utilizar

Las bibliotecas estándar son externas a Dynamo y están presentes en los lenguajes de programación Python y C# (Zerotouch). Dynamo también tiene su propio conjunto de bibliotecas que corresponden directamente con su jerarquía de nodos, lo que permite al usuario crear mediante código cualquier elemento que se pueda crear con nodos y líneas. A continuación se muestra una guía sobre los elementos a los que cada biblioteca de Dynamo proporciona acceso y cuándo utilizar una biblioteca estándar.



Bibliotecas estándar y bibliotecas de Dynamo

- Las bibliotecas estándar de Python y C# se pueden utilizar para generar datos avanzados y estructuras de flujo en el entorno de Dynamo.
- Las bibliotecas de Dynamo se corresponden directamente con la jerarquía de nodos para crear geometría y otros objetos de Dynamo.

Bibliotecas de Dynamo

- ProtoGeometry
 - Funciones: Arco, Cuadro delimitador, Círculo, Cono, Sistema de coordenadas, Cubo, Curva, Cilindro, Borde, Elipse, Arco de elipse, Cara, Geometría, Hélice, Grupo de índice, Línea, Malla, Curva Nurbs, Superficie Nurbs, Plano, Punto, Polígono, Rectángulo, Sólido, Esfera, Superficie, Topología, T-Spline, UV, Vector, Vértice.
 - Cómo importar: `import Autodesk.DesignScript.Geometry`
 - **Tenga en cuenta que, al utilizar ProtoGeometry a través de Python o C#, se crean objetos no administrados, que requieren que su memoria se administre manualmente. Consulte la sección **Objetos no administrados** más abajo para obtener más información.**
- DSCoreNodes
 - Funciones: Color, Rango de colores 2D, Fecha y hora, Intervalo de tiempo, IO, Fórmula, Lógica, Lista, Matemáticas, Árbol cuádruple, Cadena, Rosca.
 - Cómo importar: `import DSCore`
- Teselación
 - Funciones: Rejilla convexa, Delaunay, Voronoi.

- Cómo importar: `import Tessellation`
- DSOOffice
 - Función: Excel.
 - Cómo importar: `import DSOoffice`

Atención en el uso de etiquetas

Durante la creación de secuencias de comandos, utilizamos constantemente identificadores para indicar elementos como variables, tipos, funciones y otras entidades. A través de este sistema de notación simbólica, podemos hacer referencia cómodamente a la información mediante etiquetas (normalmente compuestas por una secuencia de caracteres) mientras creamos los algoritmos. Una correcta denominación de los elementos es importante en la escritura de código para que este se pueda leer y entender con facilidad por parte de otras personas y por usted mismo en un momento posterior. A continuación se indican algunos consejos que se deben tener en cuenta al asignar nombres a los elementos de su secuencia de comandos:

Está bien utilizar abreviaturas, pero explique la abreviatura con un comentario:

```
### BAD
csfX = 1.6
csfY= 1.3
csfZ = 1.0

### GOOD
# column scale factor (csf)
csfX = 1.6
csfY= 1.3
csfZ = 1.0
```

Evite el etiquetado redundante:

```
### BAD
import car
seat = car.CarSeat()
tire = car.CarTire()

### GOOD
import car
seat = car.Seat()
tire = car.Tire()
```

Utilice lógica positiva para los nombres de las variables, en lugar de lógica negativa:

```
### 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'
```

Dé preferencia a la "notación inversa":

```
### BAD
agents = ...
active_agents = ...
dead_agents ...

### GOOD
agents = ...
agents_active = ...
agents_dead = ...
```

Es más apropiado en términos estructurales.

Los alias se deben utilizar para acortar cadenas demasiado largas y a menudo repetidas:

```
### 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
```

El uso de alias puede generar rápidamente programas muy confusos y no estándares.

Utilice solo las palabras necesarias:

```
#### BAD
rotateToCoord = rotateFromCoord.Rotate(solid.ContextCoordinateSystem.Origin, Vector.ByCoordinates(0,0,1),5)

#### GOOD
toCoord = fromCoord.Rotate(solid.ContextCoordinateSystem.Origin, Vector.ByCoordinates(0,0,1),5)
```

"Todo debe hacerse tan simple como sea posible, pero no más". - Albert Einstein

Estilo coherente

Por lo general, hay más de una manera de programar prácticamente cualquier cosa, por lo que su "estilo personal" de crear secuencias de comandos es el resultado de las innumerables pequeñas decisiones que usted toma (o no toma) a lo largo del camino. Dicho esto, la legibilidad y las posibilidades de mantenimiento del código son resultado directo de su coherencia interna, así como del cumplimiento de las convenciones estilísticas generales. Como regla general, el código que tiene el mismo aspecto en dos lugares diferentes, también debería funcionar de la misma manera. A continuación se ofrecen algunos consejos para escribir código claro y coherente.

Convenciones para los nombres: (elija una de las convenciones siguientes para cada tipo de entidad del código y úsela siempre)

- Variables, funciones, métodos, paquetes, módulos: `minúsculas_con_guiones_bajos`
- Clases y excepciones: `PalabrasMayúsculas`
- Métodos protegidos y funciones internas: `_guión_bajo_sencillo_inicial_(uno, ...)`
- Métodos privados: `__guión_bajo_doble_inicial_(uno, ...)`
- Constantes: `TODO_MAYÚSCULAS_CON_GUIONES_BAJOS`

Sugerencia: evite las variables de una sola letra (especialmente l, O, I) excepto en bloques muy cortos, cuando el significado es claramente visible desde el contexto inmediato.

Uso de líneas en blanco:

- Rodee las definiciones de clase y función de nivel superior con dos líneas en blanco.
 - Las definiciones de método dentro de una clase se rodean con una sola línea en blanco.
 - Se pueden utilizar líneas en blanco adicionales (con moderación) para separar grupos de funciones relacionadas.

Evite los espacios innecesarios:

- Inmediatamente después de paréntesis, corchetes o llaves:

```
#### BAD
function( apples[ 1 ], { oranges: 2 } )

#### GOOD:
function(apples[1], {oranges: 2})
```

- Inmediatamente antes de coma, punto y coma o dos puntos:

```
#### BAD
if x == 2 : print x , y ; x , y = y , x

#### GOOD
if x == 2: print x, y; x, y = y, x
```

- Inmediatamente antes del paréntesis de apertura que inicia la lista de argumentos de una llamada de función:

```
#### BAD
function (1)

#### GOOD
function(1)
```

- Inmediatamente antes del paréntesis de apertura que inicia una indexación o un fragmento:

```
#### BAD
dict ['key'] = list [index]

#### GOOD
dict['key'] = list[index]
```

- Rodee siempre estos operadores binarios con un solo espacio a cada lado:

```
assignment ( = )
augmented assignment ( += , -= etc.)
comparisons ( == , < , > , != , <> , <= , >= , in , not in , is , is not )
Booleans ( and , or , not )
```

Vigile la longitud de las líneas:

- No las alargue más allá de aproximadamente 79 caracteres.
- La limitación de la anchura necesaria en la ventana del editor permite tener varios archivos abiertos en paralelo y posibilita el correcto

funcionamiento de las herramientas de revisión de código que presentan las dos versiones en columnas adyacentes.

- Las líneas largas se pueden dividir en varias líneas encerrando las expresiones entre paréntesis:

Evite los comentarios obvios y redundantes:

- En algunos casos, se requieren menos comentarios para obtener un código más legible. Especialmente si ello le obliga a utilizar nombres de símbolo significativos en lugar de descripciones.
- La adopción de buenos hábitos de codificación reduce la dependencia de los comentarios:

```
### 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()
```

Sugerencia: los comentarios indican por qué, el código indica cómo.

Tenga en cuenta el código abierto:

- Los proyectos de código abierto se basan en los esfuerzos colaborativos de muchos desarrolladores. Estos proyectos necesitan mantener un alto nivel de legibilidad del código para que el equipo pueda trabajar en conjunto de la forma más eficiente posible. Por lo tanto, es recomendable examinar el código fuente de estos proyectos para observar lo que hacen estos desarrolladores.
- Mejore las convenciones:
 - Pregúntese si cada una de las convenciones funciona o no para las necesidades en cuestión.
 - ¿Se está poniendo en peligro la funcionalidad/eficiencia?

Normas de C# (Zerotouch)

Consulte estas páginas wiki para obtener instrucciones sobre cómo escribir C# para Zerotouch y contribuir con Dynamo:

- Este wiki describe algunas normas de codificación generales para documentar y probar el código:
<https://github.com/DynamoDS/Dynamo/wiki/Coding-Standards>
- Este wiki describe específicamente las normas de nomenclatura de bibliotecas, categorías, nombres de nodos, nombres de puertos y abreviaturas:
<https://github.com/DynamoDS/Dynamo/wiki/Naming-Standards>

Objetos no administrados:

Cuando se utiliza la biblioteca de geometría de Dynamo (*ProtoGeometry*) desde Python o C#, los objetos de geometría que se creen no se gestionarán mediante la máquina virtual, y la memoria de muchos de estos objetos deberá limpiarse manualmente. Para limpiar objetos nativos o no administrados, puede utilizar el método **Dispose** o la palabra clave **using**. Consulte esta entrada wiki para obtener una descripción general: <https://github.com/DynamoDS/Dynamo/wiki/Zero-Touch-Plugin-Development#dispose—using-statement>.

Solo es necesario eliminar los recursos no administrados que no se devuelvan al gráfico y los que no almacenan una referencia. En el resto de esta sección, nos referiremos a estos objetos como *geometría intermedia*. Puede ver un ejemplo de esta clase de objeto en el ejemplo de código siguiente. La función de Zero Touch C# **singleCube** devuelve un único cubo, pero crea 10000 cubos adicionales durante su ejecución. Podemos fingir que esta otra geometría se ha utilizado como geometría de construcción intermedia.

Es muy probable que esta función de Zero Touch bloquee Dynamo. Esto se debe a que hemos creado 10000 sólidos, pero solo hemos almacenado uno de ellos y solo hemos devuelto ese mismo. Deberíamos desechar todos los cubos intermedios, excepto el que devolvemos. No debemos desechar lo que devolvemos, ya que se propaga en el gráfico y lo utilizan otros nodos.

```
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;
}
```

El código fijo tendría un aspecto similar al siguiente:

```
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;  
}
```

En general, únicamente es necesario desechar geometría del tipo **Surfaces**, **Curves** y **Solids**. Sin embargo, para asegurarse, puede eliminar todos los tipos de geometría (**Vectors**, **Points**, **CoordinateSystems**).

Apéndice

Apéndice A: recursos

En esta sección, encontrará recursos adicionales para ampliar sus conocimientos sobre Dynamo. También hemos añadido un índice de nodos importantes, una colección de paquetes útiles y un repositorio de los archivos de ejemplo de este manual de introducción. No dude en añadir lo que desee a esta sección... Recuerde que [Dynamo Primer](#) es un proyecto de código abierto.

Recursos

Recursos

Página wiki de Dynamo

Esta página wiki se ha diseñado para obtener información sobre el desarrollo mediante la API de Dynamo, e incluye bibliotecas y herramientas.

<https://github.com/DynamoDS/Dynamo/wiki>

Blog de Dynamo

Este blog es la colección más actualizada de artículos del equipo de Dynamo, donde se explican las nuevas funciones, los flujos de trabajo y todos los aspectos relacionados con Dynamo.

<http://dynamobim.com/blog/>

Guía de DesignScript

Se crean lenguajes de programación para expresar ideas, normalmente, con lógica y cálculo. Además de estos objetivos, el lenguaje textual de Dynamo (anteriormente, DesignScript) se ha creado para expresar las intenciones del diseño. Por lo general, se reconoce que el diseño computacional es exploratorio y Dynamo intenta apoyar esto: esperamos que el lenguaje le resulte flexible y lo suficientemente rápido como para desarrollar un diseño desde el concepto y las iteraciones de diseño hasta la forma final. Este manual se ha organizado para ofrecer a los usuarios sin conocimientos de programación o geometría arquitectónica información completa sobre diversos temas de estas dos disciplinas relacionadas.

<http://dynamobim.org/wp-content/links/DesignScriptGuide.pdf>

El proyecto Dynamo Primer

Dynamo Primer es un proyecto de código abierto iniciado por Matt Jezyk y el equipo de desarrollo de Dynamo de Autodesk. La primera versión de este manual de introducción fue desarrollada por Mode Lab. Para colaborar, bifurque el repositorio, añada contenido y envíe una solicitud de extracción.

<https://github.com/DynamoDS/DynamoPrimer>

Desarrollo del módulo de extensión Zero Touch para Dynamo

En esta página se describe el proceso de desarrollo de un nodo de Dynamo personalizado en C# mediante la interfaz de "Zero Touch". En la mayoría de los casos, se pueden importar clases y métodos estáticos de C# sin modificaciones. Si la biblioteca solo necesita funciones de llamada y no crear nuevos objetos, esto se puede conseguir muy fácilmente con métodos estáticos. Cuando Dynamo carga el archivo DLL, eliminará el espacio de nombres de las clases y mostrará todos los métodos estáticos como nodos.

<https://github.com/DynamoDS/Dynamo/wiki/Zero-Touch-Plugin-Development>

Python For Beginners

Python es un lenguaje de programación interpretado, interactivo y orientado a objetos. Incorpora módulos, excepciones, escritura dinámica, tipos de datos dinámicos de alto nivel y clases. Python combina una potencia notable con una sintaxis muy clara. Incluye interfaces para muchas llamadas y bibliotecas del sistema, así como para diversos sistemas de ventanas, y es extensible en C o C++. También se puede utilizar como lenguaje de extensión para aplicaciones que necesitan una interfaz programable. Por último, Python es portátil: se ejecuta en muchas variantes de Unix, en Mac, y en Windows 2000 y versiones posteriores. La guía para principiantes de Python ofrece vínculos a otros aprendizajes y recursos de introducción para el aprendizaje de Python.

<https://www.python.org/about/gettingstarted>

AForge

AForge.NET es un marco de trabajo de C# de código abierto diseñado para desarrolladores e investigadores de los campos de la visión y la inteligencia artificiales: procesamiento de imágenes, redes neuronales, algoritmos genéticos, lógica indefinida, aprendizaje automático, robótica, etc.

<http://www.aforgenet.com/framework/>

Wolfram MathWorld

MathWorld es un recurso en línea de matemáticas recopilado por Eric W. Weisstein con ayuda de miles de colaboradores. Desde que su contenido apareció por primera vez en línea en 1995, MathWorld se ha convertido en un nexo de información matemática tanto en las comunidades matemáticas como en las educativas. Sus entradas hacen referencia ampliamente a diarios y libros que abarcan todos los niveles educativos.

<http://mathworld.wolfram.com/>

Recursos de Revit

buildz

Estas publicaciones tratan principalmente sobre la plataforma Revit, con recomendaciones sobre cómo disfrutarla.

<http://buildz.blogspot.com/>

Nathan's Revit API Notebook

Este bloc de notas intenta solucionar algunas "deficiencias de recursos" en el aprendizaje y la aplicación de la API de Revit en el contexto de un flujo de trabajo de diseño.

<http://wiki.theprovingground.org/revit-api>

Revit Python Shell

RevitPythonShell añade un intérprete de IronPython a Autodesk Revit y Vasari. Este proyecto es anterior a Dynamo y es una excelente referencia para el desarrollo mediante Python. Proyecto RPS: <https://github.com/architecture-building-systems/revitpythonshell> Blog del desarrollador: <http://darenatwork.blogspot.com/>

The Building Coder

Un completo catálogo de flujos de trabajo de la API de Revit de uno de los expertos más importantes de BIM.

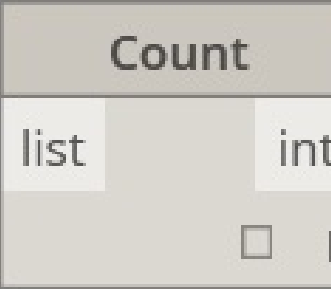
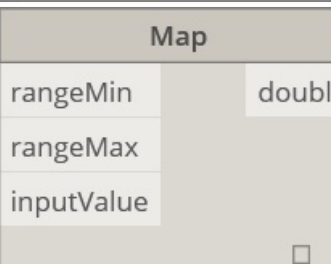
<http://thebuildingcoder.typepad.com/>

Índice de nodos

ÍNDICE DE NODOS

Este índice proporciona información adicional sobre todos los nodos utilizados en este manual de introducción, así como sobre otros componentes que puede resultar útiles. Esto es solo una introducción a algunos de los 500 nodos disponibles en Dynamo.

Funciones integradas

	Count Devuelve el número de elementos de la lista especificada	
	Flatten Devuelve la lista 1D simplificada de la lista de entrada multidimensional.	
	Map Asigna un valor a un rango de entrada.	

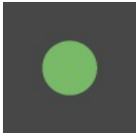
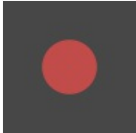
Core

Core.Color

	CREAR Color.ByARGB Genera un color mediante componentes alfa, rojo, verde y azul.	
---	---	--

		Color.ByARGB a r g b Color ☐
	Color Range Obtiene un color de un degradado de color entre un color inicial y un color final.	Color Range colors indices value color ☐
	ACCIONES	
	Color.Brightness Obtiene el valor de brillo de este color.	Color.Brightness c double ☐
	Color.Components Enumera los componentes del color en el orden siguiente: alfa, rojo, verde, azul.	

		Color.Components c a r g b ☐ I
Color.Saturation Obtiene el valor de saturación de este color.		Color.Saturation c double ☐ I
Color.Hue Obtiene el valor de matiz de este color.		Color.Hue c double ☐ I
	CONSULTA	
Color.Alpha Busca el componente alfa del color, de 0 a 255.		Color.Alpha color int ☐ I
Color.Blue Busca el componente azul del color, de 0 a 255.		

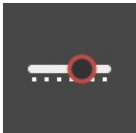
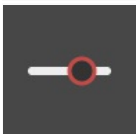
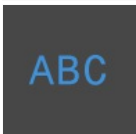
		Color.Blue colorint ☐ I
	Color.Green Busca el componente verde del color, de 0 a 255.	Color.Green colorint ☐ I
	Color.Red Busca el componente rojo del color, de 0 a 255.	Color.Red colorint ☐ I

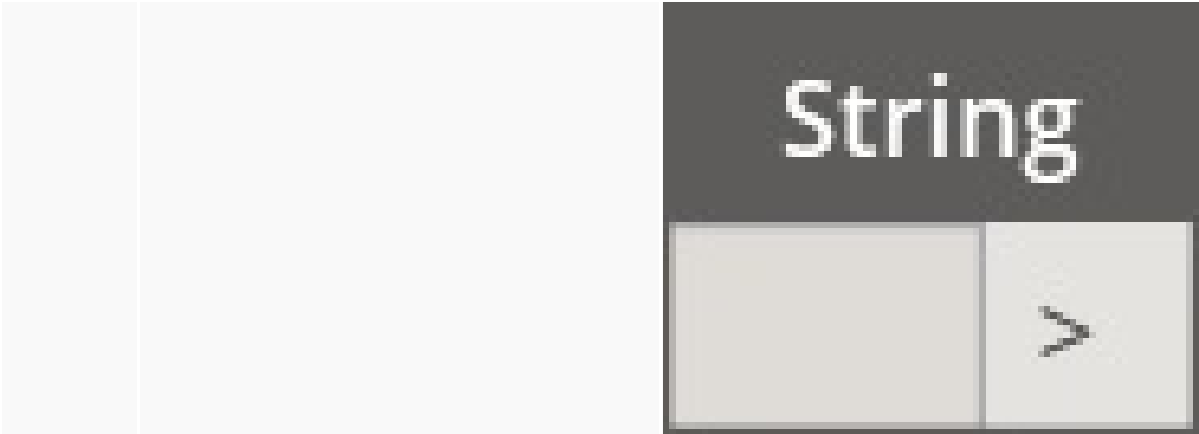
Core.Display

	CREAR Display.ByGeometryColor Muestra geometría mediante un color.	Display.ByGeometryColor geometrycolor ☐ I
		

Core.Input

	ACCIONES Boolean Selección entre un valor "true" (verdadero) y "false" (falso).	Boolean ☐ True ☒ False >
		
	Bloque de código Permite la creación directa de código de DesignScript.	

		Code Block Your code goes here
	Directory Path Permite seleccionar un directorio del sistema para obtener su ruta.	Directory Path Browse... > No file selected.
	File Path Permite seleccionar un archivo del sistema para obtener su nombre de archivo	File Path Browse... > No file selected.
	Integer Slider Un control deslizante que genera valores enteros.	Integer Slider ▼ 0 >
	Number Crea un número.	Number 0.000 >
	Number Slider Un control deslizante que genera valores numéricos.	Number Slider ▼ 0 >
	String Crea una cadena.	



Core.List

	CREAR	
	List.Create Crea una nueva lista a partir de las entradas especificadas.	List.Create index0 + - list ☐
	List.Combine Aplica un combinador a cada elemento en dos secuencias	List.Combine comb + - combined list1 list2 ☐
	Intervalo de números Crea una secuencia de números o letras en el intervalo especificado.	Number Range start seq end step ☐ I\
	Secuencia de números Crea una secuencia de números.	Number Sequence start seq amount step ☐ I\
	ACCIONES	
	List.Chop Divide una lista en un conjunto de listas en el que cada una contiene la cantidad especificada de elementos.	List.Chop list var[..<] subLength ☐ I
	List.Count Obtiene el número de elementos almacenados en la lista especificada.	List.Count list int ☐ I
	List.Flatten Aplana una lista anidada de listas en función de una determinada cantidad.	List.Flatten list var[..<] amt ☐ I
	List.FilterByBoolMask Filtra una secuencia mediante la búsqueda de los índices correspondientes en una lista independiente de valores booleanos.	List.FilterByBoolMask list in mask out ☐ I

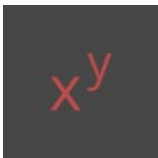
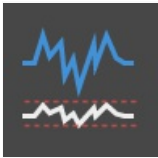
	List.GetItemAtIndex Devuelve un elemento de la lista especificada que se encuentra en el índice indicado.	List.GetItemAtIndex listvar[]..[] index ☐
	List.Map Aplica una función a todos los elementos de una lista y genera una lista nueva a partir de los resultados	List.Map listmapped f(x) ☐
	List.Reverse Crea una lista con los elementos de la lista especificada, pero en el orden inverso.	List.Reverse listvar[]..[] ☐
	List.ReplaceItemAtIndex Sustituye un elemento de la lista especificada que se encuentra en el índice indicado.	List.ReplaceItemAtIndex listvar[]..[] index item ☐
	List.ShiftIndices Desplaza a la derecha los índices de la lista en función de la cantidad especificada.	List.ShiftIndices listvar[]..[] amount ☐
	List.TakeEveryNthItem Recupera elementos de la lista especificada en los índices que son múltiplos del valor especificado y después del desfase indicado.	List.TakeEveryNthItem listvar[]..[] n offset ☐
	List.Transpose Intercambia filas y columnas en una lista de listas. Si hay algunas filas más cortas que otras, los valores nulos se insertan como marcadores de posición en la matriz resultante de forma que siempre sean rectangulares.	List.Transpose listsvar[]..[] ☐

Core.Logic

	ACCIONES	
	If Sentencia condicional. Comprueba el valor booleano de la entrada de prueba. Si la entrada de prueba es "true" (verdadera), el resultado genera la entrada "true"; de lo contrario, el resultado genera la entrada "false" (falsa).	If testresult true false ☐

Core.Math

	ACCIONES	
	Math.Cos Devuelve el coseno de un ángulo.	Math.Cos angledouble ☐
	Math.DegreesToRadians Convierte un ángulo expresado en grados en un ángulo expresado en radianes.	Math.DegreesToRadians degreesdouble ☐
	Math.Pow Eleva un número a la potencia especificada.	

		Math.Pow number power double ☐
	Math.RadiansToDegrees Convierte un ángulo expresado en radianes en un ángulo expresado en grados.	Math.RadiansToDegrees radians double ☐
	Math.RemapRange Ajusta el intervalo de una lista de números conservando la relación de distribución.	Math.RemapRange numbers newMin newMax var[]..[] ☐
	Math.Sin Calcula el seno de un ángulo.	Math.Sin angle double ☐

Core.Object

	ACCIONES	
	Object.IsNull Determina si el objeto especificado es nulo.	Object.IsNull obj bool ☐

Core.Scripting

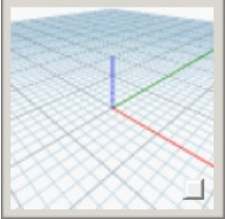
	ACCIONES	
	Formula Evalúa fórmulas matemáticas. Utiliza NCalc para la evaluación. Consulte http://ncalc.codeplex.com .	Formula > ☐

Core.String

	ACCIONES	
	String.Concat Concatena varias cadenas y crea una sola cadena.	String.Concat string0 + - string ☐
	String.Contains Determina si la cadena indicada contiene la subcadena especificada.	

		String.Contains str searchFor ignoreCase bool
	String.Join Concatena varias cadenas en una sola e inserta el separador especificado entre cada una de las cadenas unidas.	String.Join separator + - string string0
	String.Split Divide una cadena en una lista de cadenas. Las cadenas de separación especificadas determinan las divisiones.	String.Split str + - string[] separator0
	String.ToNumber Convierte una cadena en un número entero o doble.	String.ToNumber str var[]..[]

Core.View

	ACCIONES	
	View.Watch Permite visualizar la salida del nodo.	Watch > >
	View.Watch 3D Muestra una vista preliminar dinámica de la geometría.	Watch 3D > > 

Geometría

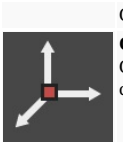
Geometry.Circle

	CREAR	
	Circle.ByCenterPointRadius Crea un círculo con punto central y radio en el plano XY universal, con el plano Z universal como normal.	Circle.ByCenterPointRadius centerPoint radius Circle
	Circle.ByPlaneRadius Crea un círculo centrado en el origen de plano de entrada (raíz), que se encuentra en el plano de entrada, con el radio especificado.	



Circle.ByPlaneRadius	
plane	Circle
radius	
☐	

Geometry.CoordinateSystem



CREAR

CoordinateSystem.ByOrigin

Crea un CoordinateSystem con origen en el punto de entrada con los ejes X e Y establecidos como los ejes X e Y de SCU.

CoordinateSystem.ByOrigin	
origin	CoordinateSystem
☐	

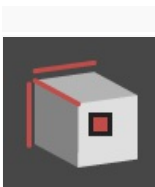


CoordinateSystem.ByCylindricalCoordinates

Crea un CoordinateSystem en los parámetros de coordenadas cilíndricas especificados respecto al sistema de coordenadas especificado.

CoordinateSystem.ByCylindricalCoordinates	
cs	CoordinateSystem
radius	
theta	
height	
☐	

Geometry.Cuboid



CREAR

Cuboid.ByLengths (origin)

Crea un cubo centrado en el punto de entrada con la anchura, la longitud y la altura especificadas.

Cuboid.ByLengths	
origin	Cuboid
width	
length	
height	
☐	

Geometry.Curve



ACCIONES

Curve.Extrude (distance)

Extruye una curva en la dirección del vector normal.

Curve.Extrude	
curve	Surface
distance	
☐	



Curve.PointAtParameter

Obtiene un punto en la curva en el parámetro especificado entre StartParameter() y EndParameter().

Curve.PointAtParameter	
curve	Point
param	
☐	

Geometry.Geometry



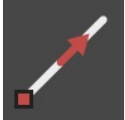
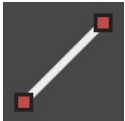
ACCIONES

Geometry.DistanceTo

Obtiene la distancia entre esta geometría y otra.

		Geometry.DistanceTo geometry double other ☐
	Geometry.Explode Separa elementos compuestos o no separados en sus componentes.	Geometry.Explode geometry Geometry[] ☐
	Geometry.ImportFromSAT Lista de geometrías importadas	Geometry.ImportFromSAT file Geometry[]..[] ☐
	Geometry.Rotate (basePlane) Rota un objeto sobre la normal y el origen del plano el número de grados especificado.	Geometry.Rotate geometry Geometry basePlane degrees ☐
	Geometry.Translate Traslada cualquier tipo de geometría según la distancia especificada en la dirección indicada.	Geometry.Translate geometry Geometry direction distance ☐

Geometry.Line

	CREAR	
	Line.ByBestFitThroughPoints Crea una línea que aproxima al máximo un diagrama de dispersión de puntos.	Line.ByBestFitThroughPoints bestFitPoints Line ☐
	Line.ByStartPointDirectionLength Crea una línea recta desde el punto inicial y la extiende la longitud especificada en la dirección del vector.	Line.ByStartPointDirectionLength startPoint Line direction length ☐
	Line.ByStartPointEndPoint Crea una línea recta entre dos puntos de entrada.	Line.ByStartPointEndPoint startPoint Line endPoint ☐
	Line.ByTangency	



Crea una línea tangente a la curva de entrada, que se coloca en el parámetro de punto de la curva de entrada.

Line.ByTangency

curve

parameter

Line

☐ |

CONSULTA



Line.Direction
La dirección de la curva.

Line.Direction

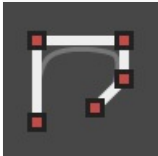
line

Vector

☐ |

Geometry.NurbsCurve

Crear



NurbsCurve.ByControlPoints
Crea una BSplineCurve mediante puntos de control explícitos.

NurbsCurve.ByControlPoints

points

degree

NurbsCurve

☐ |



NurbsCurve.ByPoints
Crea una BSplineCurve mediante la interpolación entre puntos.

NurbsCurve.ByPoints

points

degree

NurbsCurve

☐ |

qcomm

Geometry.NurbsSurface

Crear



NurbsSurface.ByControlPoints
Crea una NurbsSurface mediante el uso de puntos de control explícitos y los grados especificados para U y V.

NurbsSurface.ByControlPoints

controlVertices

uDegree

vDegree

NurbsSurface

☐ |



NurbsSurface.ByPoints
Crea una NurbsSurface con los puntos interpolados y los grados especificados para U y V. La superficie resultante atravesará todos los puntos.

NurbsSurface.ByPoints

points

uDegree

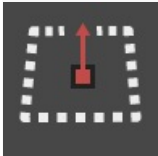
vDegree

NurbsSurface

☐ |

Geometry.Plane

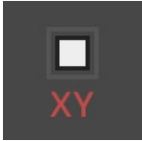
CREAR

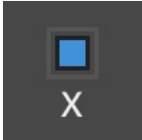
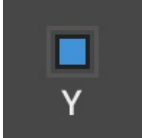
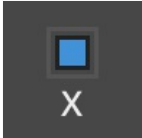


Plane.ByOriginNormal
Crea un plano centrado en el punto raíz con el vector normal de entrada.

		Plane.ByOriginNormal origin Plane normal ☐
	Plane.XY Crea un plano en el plano XY universal.	Plane.XY Plane ☐

Geometry.Point

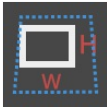
	CREAR	
	Point.ByCartesianCoordinates Genera un punto en el sistema de coordenadas especificado con tres coordenadas cartesianas.	Point.ByCartesianCoordinates CS Point x y z ☐
	Point.ByCoordinates (2D) Genera un punto en el plano XY con las dos coordenadas cartesianas especificadas. El componente Z es 0.	Point.ByCoordinates Point x y ☐
	Point.ByCoordinates (3D) Genera un punto con las tres coordenadas cartesianas especificadas.	Point.ByCoordinates Point x y z ☐
	Point.Origin Obtiene el punto de origen (0,0,0).	Point.Origin Point ☐
	ACCIONES	
	Point.Add Añade un vector a un punto. Equivale a Translate (Vector).	

		Point.Add point vectorToAdd Point
	CONSULTA	
	Point.X Obtiene el componente X de un punto.	Point.X point double
	Point.Y Obtiene el componente Y de un punto.	Point.Y point double
	Point.Z Obtiene el componente Z de un punto.	Point.Z point double

Geometry.Polycurve

	CREAR	
	Polycurve.ByPoints Crea una PolyCurve a partir de una secuencia de líneas que conectan puntos. En una curva cerrada, el último punto debe estar en la misma ubicación que el punto inicial.	PolyCurve.ByPoints points connectLastToFirst PolyCurve

Geometry.Rectangle

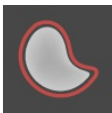
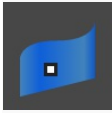
	CREAR	
	Rectangle.ByWidthLength (plane) Crea un rectángulo centrado en la raíz del plano de entrada con la anchura (longitud del eje X del plano) y la longitud de entrada (longitud del eje Y del plano).	Rectangle.ByWidthLength plane width length Rectangle

Geometry.Sphere

	CREAR	
	Sphere.ByCenterPointRadius Crea una esfera sólida centrada en el punto de entrada con el radio especificado.	Sphere.ByCenterPointRadius centerPoint radius Sphere

Geometry.Surface

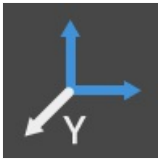
	CREAR	
	Surface.ByLoft Crea una superficie mediante la solevación entre curvas de sección transversal de entrada.	Surface.ByLoft crossSections Surface
	Surface.ByPatch	

	Crea una superficie mediante el relleno del interior de un contorno cerrado definido por curvas de entrada.	Surface.ByPatch closedCurveSurface ☐ I
ACCIONES		
	Surface.Offset Desfasa la superficie en la dirección de la normal de superficie en función de la distancia especificada.	Surface.Offset surfaceSurface distance ☐ I
	Surface.PointAtParameter Devuelve el punto en los parámetros U y V especificados.	Surface.PointAtParameter surfacePoint u v ☐ I
	Surface.Thicken Engrosa la superficie hasta convertirla en un sólido y realiza la extrusión en la dirección de las normales de superficie en ambos lados de la superficie.	Surface.Thicken surfaceSolid thickness ☐ I

Geometry.UV

CREAR		
	UV.ByCoordinates Crea un UV a partir de dos dobles.	UV.ByCoordinates uUV v ☐ I

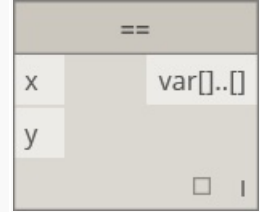
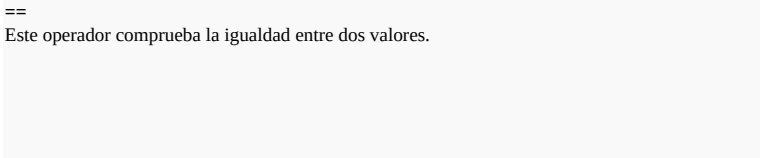
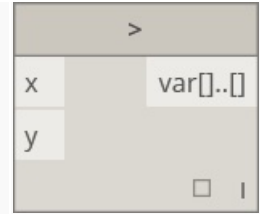
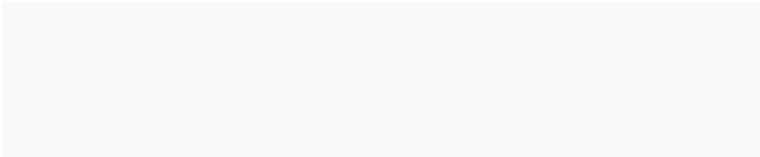
Geometry.Vector

CREAR		
	Vector.ByCoordinates Genera un vector mediante tres coordenadas euclidianas.	Vector.ByCoordinates xVector y z ☐ I
	Vector.XAxis Obtiene el vector del eje X canónico (1,0,0).	Vector.XAxis Vector ☐ I
	Vector.YAxis Obtiene el vector del eje Y canónico (0,1,0).	Vector.YAxis Vector ☐ I
	Vector.ZAxis Obtiene el vector del eje Z canónico (0,0,1).	

		Vector.ZAxis Vector ☐ I
	ACCIONES	
	Vector.Normalized Obtiene la versión normalizada de un vector.	Vector.Normalized vector Vector ☐ I

Operadores

	+ Suma	+ x var[]..[] y ☐ I
	- Resta	- x var[]..[] y ☐ I
	* Multiplicación	* x var[]..[] y ☐ I
	/ División	/ x var[]..[] y ☐ I
	% La división modular busca el resto de la primera entrada después la división por la segunda entrada	% x var[]..[] y ☐ I
	< Menor que	< x var[]..[] y ☐ I
	> Mayor que	



==
Este operador comprueba la igualdad entre dos valores.

Paquetes útiles

Paquetes de Dynamo

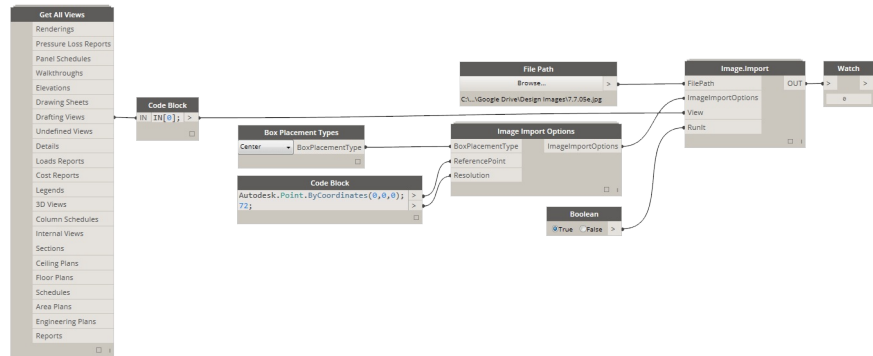
A continuación, se incluye una lista de algunos de los paquetes más populares de la comunidad de Dynamo. Es recomendable que los desarrolladores los añadan a la lista. Recuerde que [Dynamo Primer](#) es una solución de código abierto.

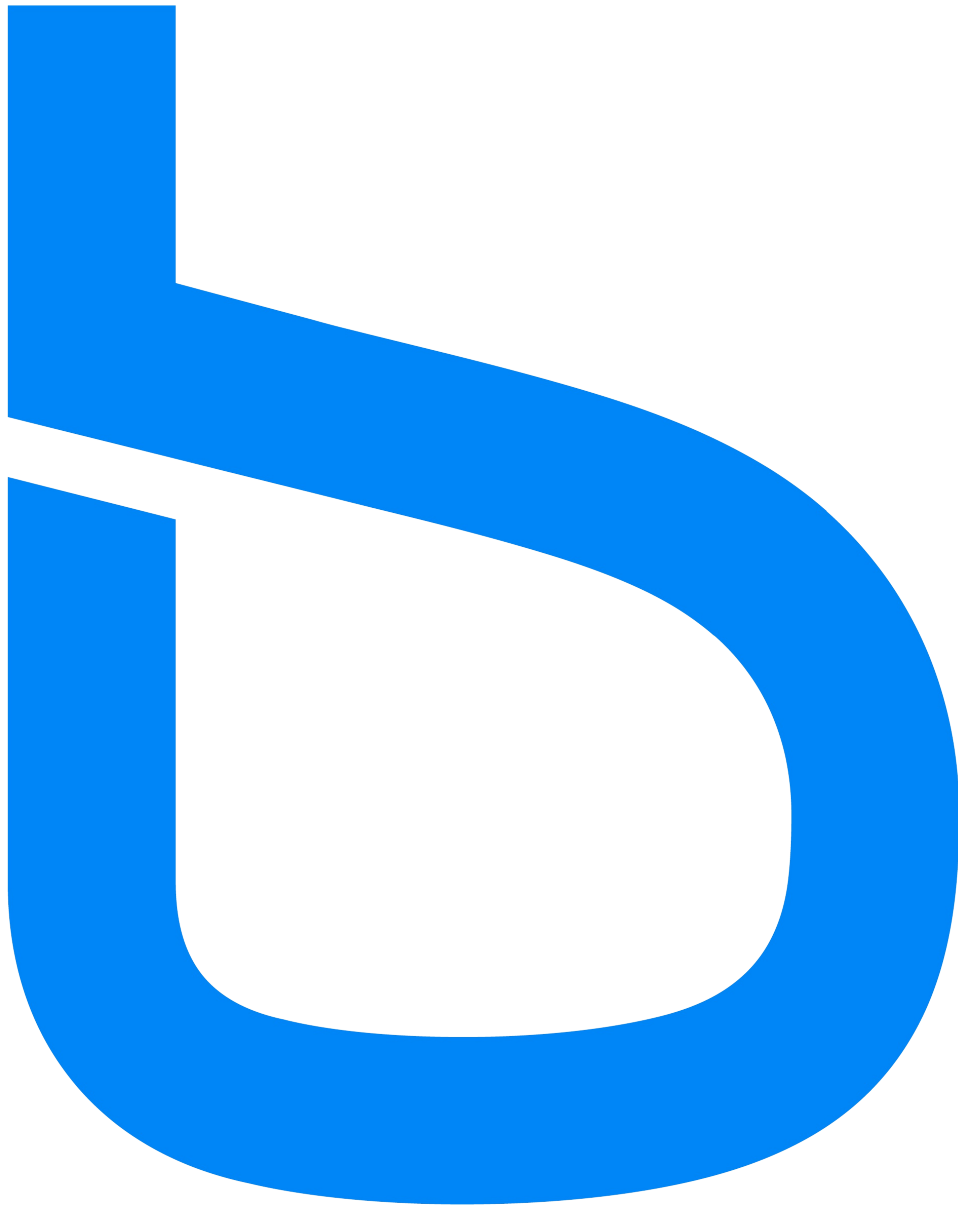
archi+lab

ARCHI-
LAB

[Visite el sitio oficial de archi-lab.](#)

archi-lab es una colección de más de 50 paquetes personalizados que amplían considerablemente la capacidad de Dynamo para interactuar con Revit. Entre los nodos del paquete de archi-lab, se incluyen desde operaciones básicas de lista a nodos de marcos avanzados de visualización de análisis para Revit.





BimorphNodes es un conjunto versátil de potentes nodos de utilidad. El paquete incluye nodos de detección de conflictos e intersección de geometría muy e conversión de curvas ImportInstance (CAD) y recopiladores de elementos vinculados que resuelven las limitaciones de la API de Revit. Para obtener más in gama completa de nodos disponibles, visite el diccionario de BimorphNodes.



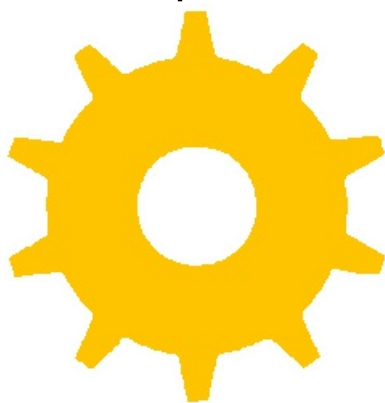
**BUMBLEBEE
PARA
DYNAMO**

[Visite el sitio oficial de Bumblebee.](#)

Bumblebee es un módulo de extensión de interoperabilidad de Excel y Dynamo que mejora considerablemente la capacidad de Dynamo para leer y escribir archivos de Excel.

	A	B	C	D	E	F	G	H	I
1	A-001	INTERSECTION LAMP DETAILS	XS	140	140	140	140	140	140
2	A-002	CONCRETE DETAIL	XS	140	140	140	140	140	140
3	A-003	JOIST LANDING PLANS	XS	140	140	140	140	140	140
4	A-004	JOIST RAFTERS	XS	140	140	140	140	140	140
5	A-005	LANDSCAPE LOTS	XS	140	140	140	140	140	140
6	A-006	LANDSCAPE TYPICAL LOTS	XS	140	140	140	140	140	140
7	A-007	LANDSCAPE TYPICAL PLANS	XS	140	140	140	140	140	140
8	A-008	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
9	A-009	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
10	A-010	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
11	A-011	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
12	A-012	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
13	A-013	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
14	A-014	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
15	A-015	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
16	A-016	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
17	A-017	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
18	A-018	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
19	A-019	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
20	A-020	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
21	A-021	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
22	A-022	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
23	A-023	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
24	A-024	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
25	A-025	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
26	A-026	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
27	A-027	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
28	A-028	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
29	A-029	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
30	A-030	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
31	A-031	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
32	A-032	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
33	A-033	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
34	A-034	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
35	A-035	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
36	A-036	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
37	A-037	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
38	A-038	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
39	A-039	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
40	A-040	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
41	A-041	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
42	A-042	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
43	A-043	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
44	A-044	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
45	A-045	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
46	A-046	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
47	A-047	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
48	A-048	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
49	A-049	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
50	A-050	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
51	A-051	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
52	A-052	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
53	A-053	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
54	A-054	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
55	A-055	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
56	A-056	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
57	A-057	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
58	A-058	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
59	A-059	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
60	A-060	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
61	A-061	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
62	A-062	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
63	A-063	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
64	A-064	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
65	A-065	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
66	A-066	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
67	A-067	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
68	A-068	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
69	A-069	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
70	A-070	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
71	A-071	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
72	A-072	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
73	A-073	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
74	A-074	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
75	A-075	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
76	A-076	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
77	A-077	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
78	A-078	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
79	A-079	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
80	A-080	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
81	A-081	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
82	A-082	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
83	A-083	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
84	A-084	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
85	A-085	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
86	A-086	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
87	A-087	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
88	A-088	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
89	A-089	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
90	A-090	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
91	A-091	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
92	A-092	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
93	A-093	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
94	A-094	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
95	A-095	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
96	A-096	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
97	A-097	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
98	A-098	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
99	A-099	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140
100	A-100	LANDSCAPE TYPICAL PLANS - LOTS	XS	140	140	140	140	140	140

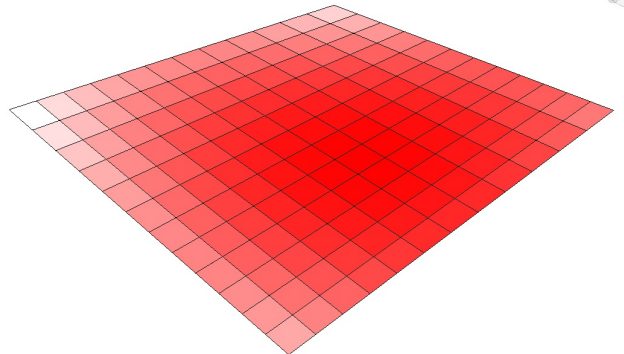
Si los valores del rango C1:C11 == El autor aplica el estilo de relleno. (10/10)



**CLOCKWORK
PARA
DYNAMO**

[Visite Clockwork para Dynamo en GitHub.](#)

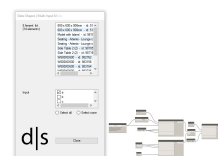
Clockwork es un conjunto de nodos personalizados para el entorno de programación visual de Dynamo. Contiene muchos nodos relacionados con Revit, pero también una gran cantidad de nodos para distintos fines, como la gestión de listas, las operaciones matemáticas, las operaciones de cadenas, las conversiones de unidades, las operaciones geométricas (principalmente, cuadros delimitadores, mallas, planos, puntos, superficies, UV y vectores) y paneles.



d|s

DATA|SHAPES [Visite DataShapes en GitHub.](#)

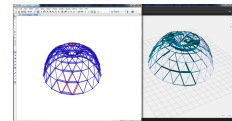
DataShapes es un paquete cuyo objetivo es ampliar la funcionalidad de las secuencias de comandos de Dynamo para el usuario. Se centra principalmente en incorporar una mayor funcionalidad al Reproductor de Dynamo. Para obtener más información, visite <https://data-shapes.net/>. ¿Desea crear impresionantes flujos de trabajo del Reproductor de Dynamo? Utilice este paquete.



DYNAMO [Visite el proyecto](#)
SAP [DynamoSAP en Core](#)
[Studio.](#)



DynamoSAP es una interfaz paramétrica para SAP2000 desarrollada sobre la base de Dynamo. El proyecto permite a los diseñadores y los ingenieros crear y analizar de forma genérica sistemas estructurales en SAP mediante el uso de Dynamo para controlar el modelo de SAP. El proyecto establece varios flujos de trabajo comunes que se describen en los archivos de ejemplo incluidos y ofrece una amplia gama de oportunidades para la automatización de tareas típicas en SAP.

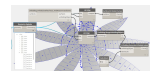


[Visite](#)

DYNAMOUNFOLD [DynamoUnfold en GitHub.](#)



Esta biblioteca amplía la funcionalidad de Dynamo/Revit al permitir a los usuarios desplegar geometría de superficie y PolySurface. La biblioteca permite a los usuarios convertir superficies en topología triangular plana y, a continuación, desplegarlas con las herramientas de proto geometría de Dynamo. Este paquete también incluye algunos nodos experimentales, así como algunos archivos de ejemplo básicos.



DYNASTRATOR [Descargue Dynastrator en Package Manager.](#)

Importe imágenes vectoriales desde Illustrator o la Web mediante .svg. Esto permite importar dibujos creados manualmente en Dynamo para operaciones paramétricas.



**ENERGY
ANALYSIS
FOR
DYNAMO**

[Visite el proyecto Energy Analysis for Dynamo en GitHub.](#)

Energy Analysis for Dynamo permite el modelado energético paramétrico y los flujos de trabajo de análisis energético de todo el edificio en Dynamo 0.8. También permite al usuario configurar el modelo energético desde Autodesk Revit, enviarlo a Green Building Studio para el análisis energético DOE2 y explorar los resultados devueltos por el análisis. El paquete está siendo desarrollado por CORE studio de Thornton Tomasetti.



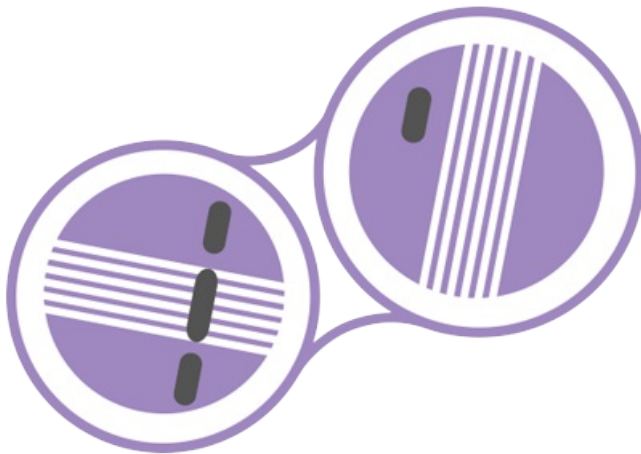
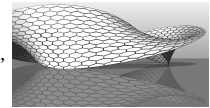


Firefly es un conjunto de nodos que permiten a Dynamo comunicarse con dispositivos de entrada y salida, como el microcontrolador Arduino. Como el flujo produce "en directo", Firefly ofrece muchas oportunidades para crear prototipos interactivos entre el mundo digital y el físico a través de cámaras web, teléfonos móviles, controladores de juegos, sensores, etc.

LUNCHBOX [Consulte LunchBox para](#)
PARA [Dynamo en Proving](#)
DYNAMO [Ground.](#)



LunchBox es un conjunto de nodos reutilizables de geometría y administración de datos. Las herramientas se han probado con Dynamo 0.8.1 y Revit 2016. La herramienta incluye nodos para paneles de superficies, geometría, recopilación de datos de Revit, etc.



MANTIS SHRIMP [Visite el sitio oficial de Mantis Shrimp.](#)

Mantis Shrimp es un proyecto de interoperabilidad que permite importar fácilmente geometría de Griciopper o Rhino a Dynamo.

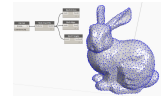




**KIT DE
HERRAMIENTAS
DE MALLA**

[Visite el Kit de
herramientas de
malla en GitHub.](#)

El Kit de herramientas de malla de Dynamo proporciona muchas herramientas útiles para trabajar con la geometría de malla. La funcionalidad de este paquete incluye la posibilidad de importar mallas desde formatos de archivo externos, generar mallas a partir de objetos de geometría de Dynamo existentes y crear manualmente mallas a través de información de vértices y conectividad. Además, este kit incluye herramientas para modificar y reparar la geometría de malla.



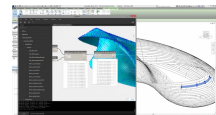
OPTIMO [Visite Optimo en GitHub.](#)

Optimo ofrece a los usuarios de Dynamo la capacidad de optimizar los problemas de diseño autodefinidos mediante diversos algoritmos evolutivos. Los usuarios pueden definir el objetivo del problema o un conjunto de objetivos, así como funciones de adecuación específicas.



RHYNAMO [Visite Rhynamo en Bitbucket.](#)

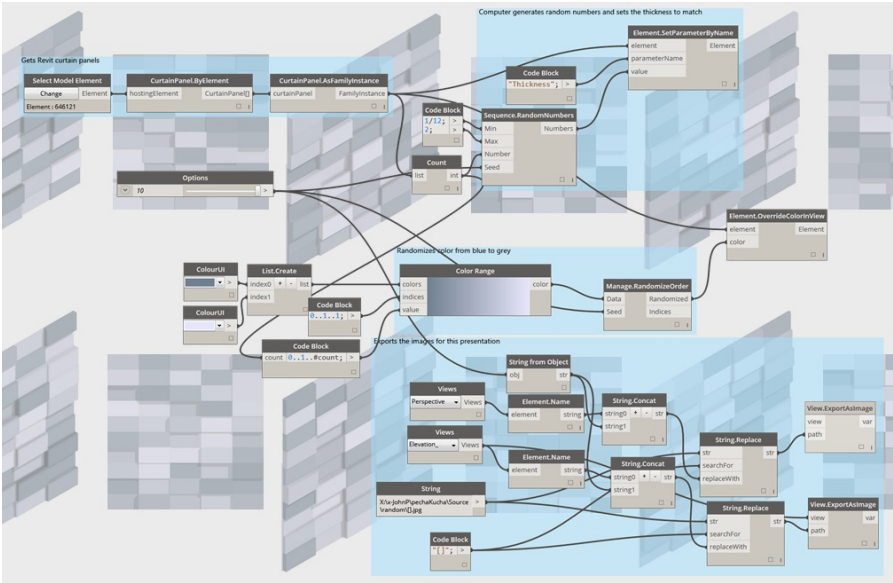
La biblioteca de nodos de Rhynamo permite a los usuarios leer y escribir archivos 3DM de Rhino en Dynamo. Rhynamo convierte la geometría de Rhino en geometría utilizable en Dynamo mediante la biblioteca OpenNURBS de McNeel, lo que ofrece nuevos flujos de trabajo que pueden intercambiar geometría y datos con fluidez entre Rhino y Revit. Este paquete contiene también algunos nodos experimentales que permiten el acceso "en directo" a la línea de comandos de Rhino.



RHYTHM [Visite Rhythm en GitHub.](#)

A primera vista, Rhythm no parece nada especial. No tiene código sofisticado ni ningún elemento de esa naturaleza. Sin embargo, lo que Rhythm representa es el resultado del

pensamiento práctico y la diligencia. Este paquete se ha diseñado para ayudar a los usuarios a usar de forma eficaz Revit con Dynamo mediante Rhythm. Rhythm se compone principalmente de nodos de Dynamo predefinidos que se utilizan de forma inteligente a medida que se aplican al entorno de Revit.



Archivos de ejemplo

Archivos de ejemplo de Dynamo

Estos archivos de ejemplo se adjuntan a Dynamo Primer y se organizan por capítulo y sección.

Haga clic con el botón derecho en los archivos y utilice "Guardar vínculo como...".

Introducción

Sección	Descargar archivo
¿Qué es la programación visual?	Visual Programming - Circle Through Point.dyn

Anatomía de una definición de Dynamo

Sección	Descargar archivo
Valores predefinidos	Presets.dyn

Los bloques de construcción de los programas

Sección	Descargar archivo
Datos	Building Blocks of Programs - Data.dyn
Matemáticas	Building Blocks of Programs - Math.dyn
Lógica	Building Blocks of Programs - Logic.dyn
Cadenas	Building Blocks of Programs - Strings.dyn
Color	Building Blocks of Programs - Color.dyn

Geometría para el diseño computacional

Sección	Descargar archivo
Descripción general de la geometría	Geometry for Computational Design - Geometry Overview.dyn
Vectores	Geometry for Computational Design - Vectors.dyn
	Geometry for Computational Design - Plane.dyn
	Geometry for Computational Design - Coordinate System.dyn
	Geometry for Computational Design - Points.dyn
Puntos	Geometry for Computational Design - Points.dyn
Curvas	Geometry for Computational Design - Curves.dyn
Superficies	Geometry for Computational Design - Surfaces.dyn
	Surface.sat

Diseño con listas

Sección	Descargar archivo
¿Qué es una lista?	Lacing.dyn
Trabajo con listas	List-Count.dyn
	List-FilterByBooleanMask.dyn
	List-GetItemAtIndex.dyn
	List-Operations.dyn
	List-Reverse.dyn
	List-ShiftIndices.dyn
	List-Union.dyn
Listas de listas	Chop.dyn
	Combine.dyn
	Flatten.dyn
	Map.dyn
	ReplaceItems.dyn
	Top-Down-Hierarchy.dyn
	Transpose.dyn
Listas de n dimensiones	n-Dimensional-Lists.dyn
	n-Dimensional-Lists.sat

Bloques de código y DesignScript

Sección	Descargar archivo
Sintaxis de DesignScript	Dynamo-Syntax_Attractor-Surface.dyn
Abreviatura	Obsolete-Nodes_Sine-Surface.dyn
Functions	Functions_SphereByZ.dyn

Dynamo para Revit

Sección	Descargar archivo
Seleccionar	Selecting.dyn ARCH-Selecing-BaseFile.rvt
Edición	Editing.dyn ARCH-Editing-BaseFile.rvt
Creando	Creating.dyn ARCH-Creating-BaseFile.rvt
Personalización	Customizing.dyn ARCH-Customizing-BaseFile.rvt
Documentación	Documenting.dyn ARCH-Documenting-BaseFile.rvt

Diccionarios en Dynamo

Sección	Descargar archivo
Diccionario de habitaciones	RoomDictionary.dyn

Nodos personalizados

Sección	Descargar archivo
Creación de un nodo personalizado	UV-CustomNode.zip
Publicación en la biblioteca	PointsToSurface.dyf
Nodos de Python	Python-CustomNode.dyn
Python y Revit	Revit-Doc.dyn
Python y Revit	Revit-ReferenceCurve.dyn
Python y Revit	Revit-StructuralFraming.zip

Paquetes

Sección	Descargar archivo
Caso real de paquete: Kit de herramientas de malla	MeshToolkit.zip
Publicación de un paquete	MapToSurface.zip
Importación de Zero-Touch	ZeroTouchImages.zip