DEVELOPER DOCUMENTATION

INTRA**LATTICE**

CORE MODULES

v0.7.5

PREFACE

VERSION **0.7.5** — ALPHA

INTRA**LATTICE** is a C# plugin for Grasshopper, used to generate solid lattice structures within a design space. It was developed as an extensible, open-source alternative to current commercial solutions. As an ongoing project developed at McGill's Additive Design & Manufacturing Laboratory (ADML), it has been a valuable research tool, serving as a platform for breakthroughs in multi-scale design and optimization.

By providing a modular approach to lattice design, and giving you full access to the source, we hope to collectively explore lattice design at a deeper level, and consequently, engineer better products.

WEBSITE — http://intralattice.com

GITHUB — https://github.com/dnkrtz/intralattice/

USER DOCS — http://intralattice.com/userdocs

THE MIT LICENSE

OPEN SOURCE

SUPPORT

If you have any issues, questions or feedback — please contact support@intralattice.com

LEAD DEVELOPERS

This project was started in the summer of 2014. Although the process was a collaborative effort, the contributions can be broadly categorized as follows:

Aidan Kurtz — Development of CORE modules, website and documentation.

Yunlong Tang — Development of OPTI modules (PhD Research).

Prof. Fiona Zhao — Project supervisor and head of the research lab.

ACKNOWLEDGEMENTS

Many students/researchers have contributed to the project since its inception.

Marc Wang —

Ken Nsiempba —

Huiyuan Yang —

If you submit a pull request to the GitHub repository, and it is merged, your name and contribution will be listed. For more information on how to contribute, refer to the Developer Documentation.

The mesh generation methods used in this project are based on the Exoskeleton algorithms developed by David Stasiuk. And of course a big thanks to David Rutten for his work on Grasshopper.

TABLE OF CONTENTS

Section 1

MODULE I/O

Before we get started, let me just remind you that this is the developer documentation, and it assumes you have a basic understanding of Intralattice and how to use it.

When designing this plugin, we tried to find a balance between flexibility and performance. Grasshopper allows us to pass custom data types from one component to the next, so long as it has a GH_Goo wrapper. However, this also imposes our data structure on users and other developers. In that sense, we opted for passing "raw" native data instead.

To better understand this, suppose the frame module outputs a custom data type, that includes information valuable to the mesh module, that, in turn, requires input data of this particular type. This would not only limit the users ability to easily manipulate the data. It would also limit compatibility, meaning if a user designed a lattice wireframe himself, as a list of curves, he could not solidify it with our mesh components. Of course, this means that the mesh module must do some extra computation, to reformat the data, but this is the approach we chose.

That being said, we've tried to minimize the I/O requirements imposed on other developers. If you are developing a new component for Intralattice, they should obey the following convention:

CELL - There are no specific **input** requirements.
The **Topo** output should be a UnitCell object.

FRAME - The **Topo** input comes from a cell component, as a UnitCell object.
The **Struts** output

MESH - The **Struts** input is expected to come from a frame component. It may contain duplicate struts, meaning we should pass it through the FrameTools.CleanNetwork() method.
The **Mesh** output should be a single closed mesh. Of course, in some cases the meshing process fails. If this happens, the component should return useful error messages.

UTILITY - There are no specific I/O requirements for this module.

Section 2

CELL MODULE

The cell module has two primary functions.

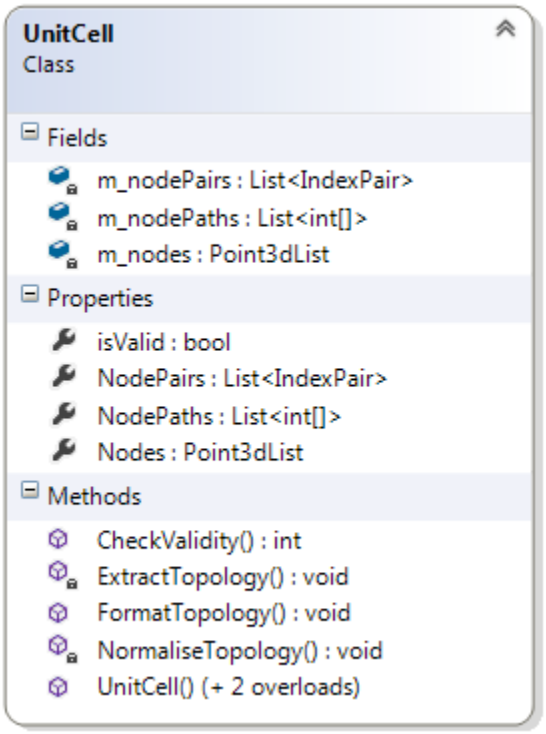
- Convert list of lines to a UnitCell data object.
- Ensure validity of the unit cell topology.

If you are designing a unit cell, the bounding box must be aligned with the World XYZ coordinate system.

A **valid unit cell** has the following property: opposing faces of the bounding box are identical, and each face has at least one node lying on it. That is to say, opposing faces must have mirror nodes. To understand this, you can simply imagine appending the cell in each direction. In order to ensure continuity, opposing faces need to have the same nodes, and a minimum of one node per face.

2.0 DATA STRUCTURE

The **UnitCell** class diagram is shown to the right. This class includes methods and fields for constructing unit cell data. Essentially, we want to convert a list of lines representing the unit cell topology, to a format which contains additional information useful for lattice mapping. To do this, we use an instance constructor.
This constructor executes the following private methods.




```
// Instantiate UnitCell object
var cell = new UnitCell(lines);
```

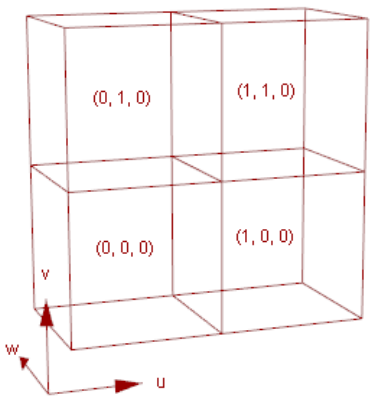
The **ExtractTopology()** method first ensures that there are no line intersections, such that all the nodes are defined. If it finds intersections, it will explode the lines at those locations. Secondly, it converts the list of lines to a list of unique nodes (m_nodes) and a list of unique index pairs, representing the struts (m_nodePairs). In doing so, it ignores duplicates. The **NormaliseTopology**() method then scales the unit cell to unit size (1x1x1 bounding box), and moves the cell to the origin.

By the end of this process, we have a UnitCell object defining a set of unique nodes and node pairs. You may check the validity of the cell

RELATIVE PATHS AND THE UVW TREE

To understand the need for relative paths (m_nodePaths), we must jump ahead and look at how the lattice wireframe is mapped. If we think of the simplest example possible, a box lattice, it's tempting to simply copy and move the unit cell over and over. However, this often results in a large number of duplicate struts, which must be removed. This issue becomes increasingly confusing for conformed and morphed lattices. Instead, our approach is to first map the unit cell nodes to the design space, and store them in such a way that the nodepair relationships can be used to quickly generate the struts.

The first concept to introduce is the **UVW cell grid**. This is implemented as a Grasshopper data tree*, where each unit cell in the lattice has a unique path (u,v,w). This format makes cell adjacency information intrinsic, since we know which cells are neighbouring a specific cell based on data paths alone. The path also provides spatial information.
Expanding on the UVW concept, each cell has multiple nodes, so we let each tree item be a list of nodes. If we include the index of the node, we get the **UVWi node grid,** which serves as our data structure for the lattice nodes. This allows us to very quickly define node relationships (struts) to create a lattice.



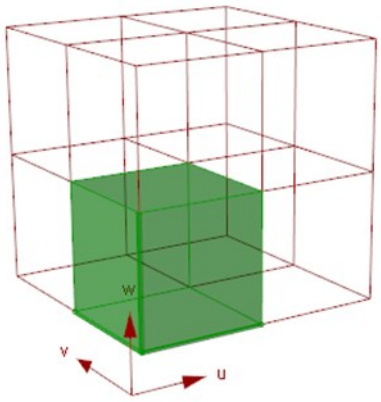
Nodes and struts that lie on cell interfaces are shared between two or more cells, and as such, are prone to redundancy. This is where the **FormatTopology()** method comes in. See next page.

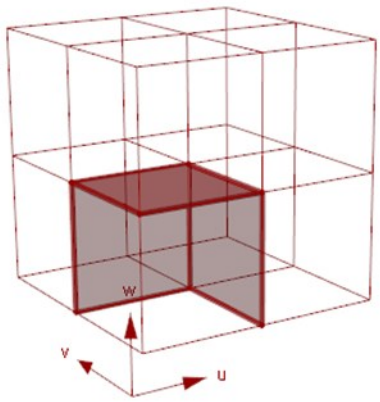
* In Grasshopper, the DataTree class is not a tree in the conventional sense. The data is stored as a SortedList<TKey, TValue>. In other words, it behaves like a sorted dictionary - data is accessed based on it's key (in our case, a UVW path).

RELATIVE PATHS AND THE UVW TREE (cont.)

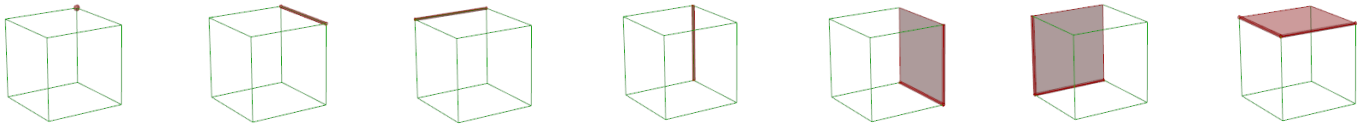
Let's consider how the lattice can be represented efficiently. Within a lattice, cells will always have neighbours, and so we need a way to make sure neighbours can share their interface nodes and struts, without creating duplicates. To do this, we state that nodes and struts lying on the positive faces of the unit cell boundary box, as well as all edges coincident with these faces, belong to neighbouring cells.

To better understand this statement, consider the following illustrations. The green area is the current cell. The red area is the positive boundary of the cell – nodes and struts lying on this area belong to neighbouring cells.





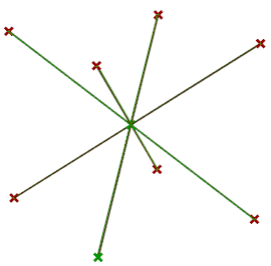
The **FormatTopology()** method defines the relative path of each node and removes struts (nodepairs) that belong to neighbouring cells. To determine the relative node paths, we perform the following node location checks, in an else-if fashion. The relative paths corresponding to each case is shown below the image.

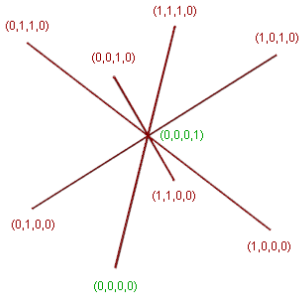


(1,1,1,i)	(1,0,1,i)	(0,1,1,i)	(1,1,0,i)	(1,0,0,i)	(0,1,0,i)	(0,0,1,i)

All other nodes have a relative path of (0,0,0,i), meaning they belong to the current cell. To determine the 'i' index for nodes which do not belong to the current cell, we rely on the assumption that opposing faces have mirror nodes (cell is valid). So before running the FormatTopology() method, **ensure the cell is valid**.

Once formatted, we have a UnitCell object ready to be mapped to the design space.





Section 3

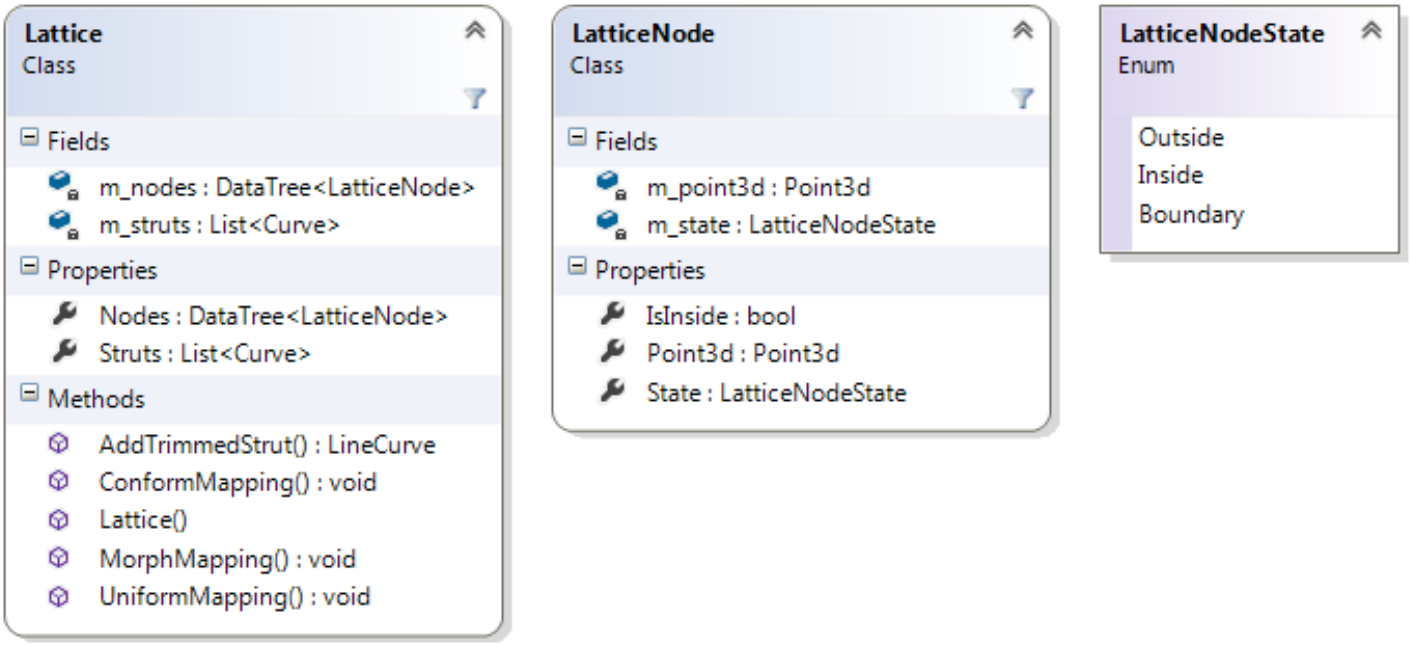
FRAME MODULE

Components compatible with the Intralattice framework need to meet certain I/O requirements. When developing a new component, keep these constraints in mind.

The modular framework is quite straight-forward.

3.0 DATA STRUCTURE

The diagram below shows the classes used to construct and organize lattice wireframe data. The **Lattice** class represents the full lattice wireframe, as a tree of LatticeNode objects, and a list of Curves. It contains various methods for mapping struts to LatticeNode data tree. The **LatticeNode** class represents individual nodes in the lattice, and is nothing more than a Point3d and a state which represents the nodes location with respect to the design space.



A lattice object is instantiated with a default constructor, and built up from scratch.

```
// Instantiate Lattice object
var lattice = new Lattice();
```

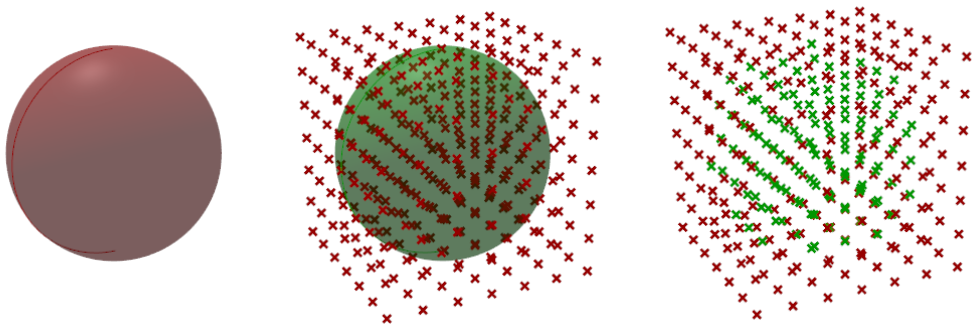
The LatticeNode objects are organized in a DataTree. As mentioned, the DataTree class is a Grasshopper construct, which is not a tree in the conventional sense, but rather, a SortedList<TKey, TValue>. Within this tree, each item represents a cell, containing a list of LatticeNode objects. The (u,v,w) path targets a particular cell. The 'i' index is used to target a specific node within the list of nodes.

DataTree Example	
Key	**Value**
GH_Path(0,0,0)	*List<LatticeNode>*
GH_Path(0,0,1)	*List<LatticeNode>*
GH_Path(0,1,0)	*List<LatticeNode>*
GH_Path(0,1,1)	*List<LatticeNode>*

3.1 UNIFORM LATTICE

NODE MAPPING

We begin by generating cell nodes uniformally within the bounding box of the design space. As we do this, we store them appropriately in the UVWi tree, and also save their state (i.e. is inside/outside the design space). When computing the state, we use a loose tolerance, meaning that the node may be outside the design space by a specified tolerance. Note that we generate the nodes for the particular input UnitCell object — the example below maps the nodes for the simple grid topology.



STRUT MAPPING

To map the struts, the **UniformMapping()** method is invoked, which enforces the cell nodepair relationships on the full UVWi node grid. As it loops through the nodes, there are 3 possible scenarios:

1. Both nodes are outside the design space.
   In this case, we don't do anything—we skip to the next node pair.

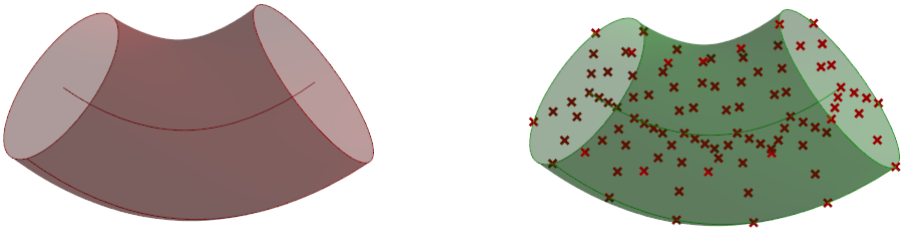
2. One node is inside, the other is outside the design space.
   In this case, we create a trimmed strut. (refer to **AddTrimmedStrut()** method)
   If the trimmed strut length is smaller than the specified minimum acceptable length, the strut isn't created.
   If the trimming is less than the minimum acceptable length, the full strut is created instead.

3. Both nodes are inside the design space.
   In this case, we create a full strut joining the two nodes.

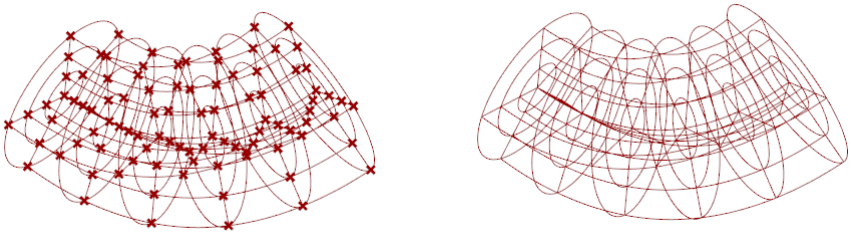
3.2 CONFORM LATTICE

NODE MAPPING

To generate the lattice nodes conformally, we map them to a uvw-map based on the surface(s) uv-map(s). This is nice since the uvw-map is consistent with the UVW cell tree concept. We won't go in detail here; for a better understanding, check out the code on GitHub. In the example below, we create a uvw-map based on two surfaces.



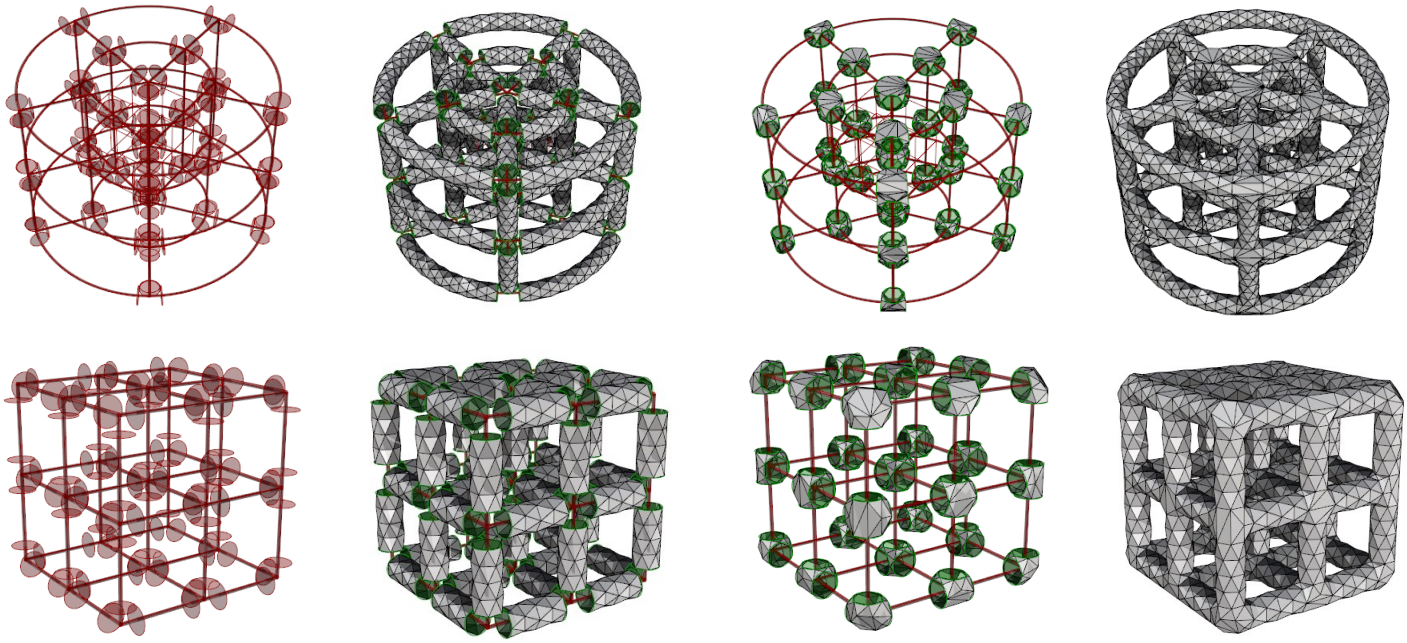
STRUT MAPPING



Section 4

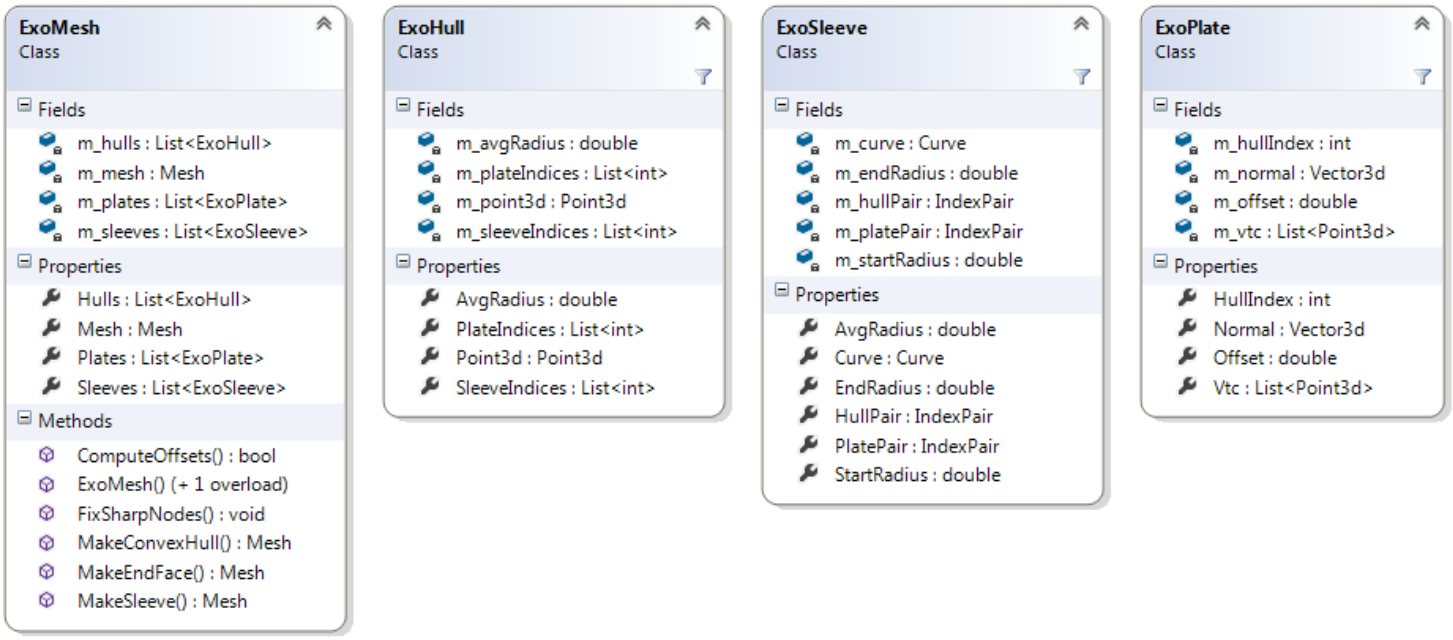
MESH MODULE

Our meshing approach is inspired by “Solidifying wireframes” (Ergun Ackleman et al.) and Exoskeleton by David Stasiuk. The idea is to generate 3D convex hulls at the nodes, and mesh sleeves along the curves.



2.2.1 DATA STRUCTURE

The diagram below summarizes the classes used to construct the mesh.



We instantiante the ExoMesh object with an instance constructor based on a list of curves. The constructor will first pass the list of curves through the CleanNetwork() method, which will remove duplicate, invalid and very short curves. As it does this, it defines a list of unique nodes and node index pairs. Note that the node index pairs will be parallel to list of curves. Next, based on this new cleaned data, the lists of ExoHulls, ExoSleeves and ExoPlates are instantiated.

```
// Instantiate ExoMesh object
var exoMesh = new ExoMesh(struts);
```

Relational fields are also set, to define association between items in the 3 lists. Each sleeve is associated to 2 hulls and 2 plates, hence the m_platePair and m_hullPair IndexPair fields. Each hull is associated to a variable number of hulls and sleeves, hence the m_plateIndices and m_sleeveIndices lists. Each plate is associated to a single hull and a single sleeve. In this case, we only store the hull index. The reason we use indexes as relational parameters instead of object references, is a question of memory. A C# object reference will be 8 bytes on a 64-bit system, whereas an integer representing an index will be 4 bytes. This format is also portable, meaning we can move/copy the data, and the relations remain valid.

2.2.2 SET STRUT RADII

Setting the radii is done on a strut-by-strut basis, where we define the start and end radius of each strut. This approach was chosen over a node-based radius, for optimization reasons. In real-world load cases, some struts may be under very little stress, and we might want to reduce their thickness regardless of the node they are connected to. In any case, it is still very easy to set radii in a node-based fashion, only that the actual radii are defined on the struts.

HOMOGENEOUS

For homogeneous lattice, the process is trivial. We set all start and end radii to the same fixed value.

HETEROGENEOUS GRADIENT

To convert the gradient function input to discrete values, we use the GH_ExpressionParser class provided by Grasshopper, to evaluate the expression string. We loop through all struts, and evaluate the radius at the start and end points. As discussed in the User Docs, the gradient function has a unit domain. Therefore, when evaluating a point in the lattice, we need to transform the (x,y,z) location to act as if the lattice bounding box was of unit size, and cornered at the origin (in the positive octant). Once we transform the point, and evaluate the parsed expression, we interpolate the result. The result will be between 0 and 1, so we interpolate it as a value between minRadius and maxRadius.
The code is shown below.

```
// Instantiate ExoMesh object
var parser = new Grasshopper.Kernel.Expressions.GH_ExpressionParser();
parser.AddVariable("x", (node.Point3d.X - fullBox.Min.X) / boxSizeX);
parser.AddVariable("y", (node.Point3d.Y - fullBox.Min.Y) / boxSizeY);
parser.AddVariable("z", (node.Point3d.Z - fullBox.Min.Z) / boxSizeZ);
sleeve.StartRadius = minRadius + (parser.Evaluate(gradientString)._Double) * (maxRadius - minRadius);
```

HETEROGENEOUS CUSTOM

In this case, we let the user do all the work for us, but the extra work gives them freedom. On our end, we simply extract the user input StartRadii/EndRadii lists into the ExoSleeve objects. This approach is of particular use for structural optimization, where we define the radii of each strut based on internal forces.

2.2.3 COMPUTE OFFSETS

At this point, we need to compute the offset of each plate from it's parent node. This is a crucial step, and we have two requirements of the plate layout at a node:

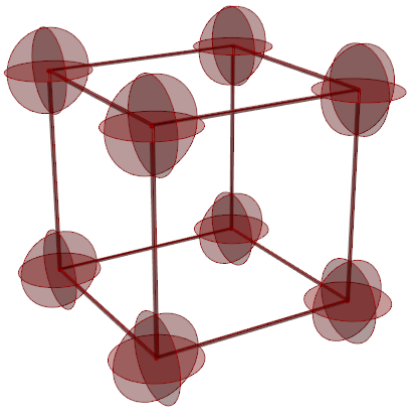
A. The sleeve meshes at these plates must not overlap (i.e. the plate circles must not intersect)

B. The plate layout must be convex, meaning that it ensures no plate vertices get engulfed by the convex hull. If this were to happen, the sleeve mesh would not be able to connect to the hull mesh. A condition that ensures this state, is such that none of the plate planes intersect any of the plate circles.

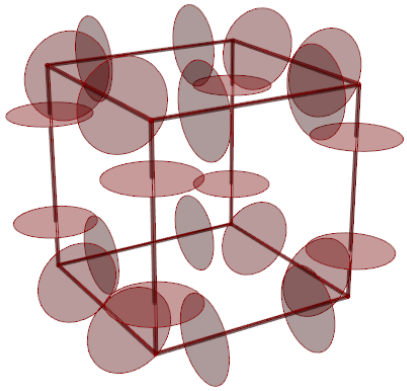
Of course, condition B precludes condition A, but both are expressed for clarity.

ROBUST, INCREMENTAL OFFSET COMPUTATION

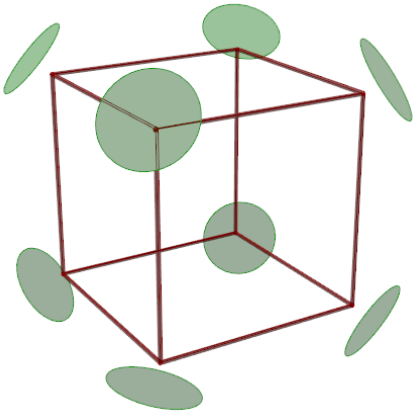
At each node, we are looking for a plate layout that ensures none of the plate circles intersect any of the plate planes. To do this, we use an incremental approach, increasing the offset for all plates simultaneously. At every iteration, we check for plane-circle intersections. If a plate's plane intersects any of the plate circle, we will offset that plate. Otherwise, we leave that plate where it is. We continue incrementing in this fashion until no intersections are found. The illustrations below show a very basic example of the concepts discussed. Naturally, this process also works very well curves.



Initial configuration
(offsets=0)



Convex configuration



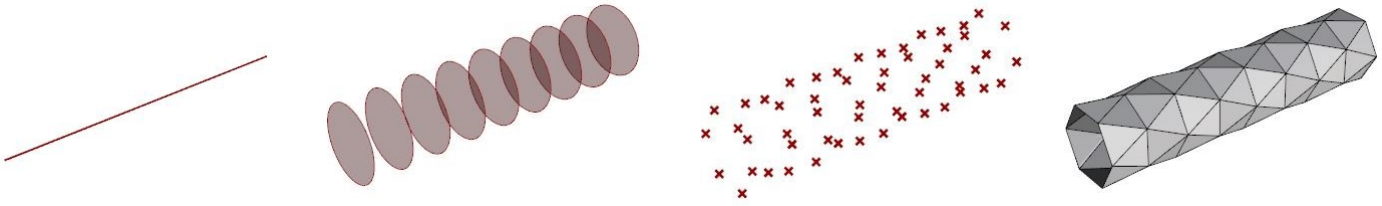
Extra plates for sharp nodes

IMPROVING SHARP NODES (FOR CONVEX HULL)

We say a node is "sharp" if all the struts lie in a 180° peripheral. In these situations, the convex hull will not capture the corner correctly. The solution is to add another plate, this pulls the convex hull outwards (jump forward 2 pages for illustrations of this). To define position and orientation of the extra plate, we first sum all strut normals at the node. The normal of the extra plate is the negative of this vector sum, and the offset distance is a factor of the magnitude.

2.2.4 SLEEVE MESHING

The sleeve meshing occurs in two steps. Each ExoSleeve has an underlying curve, which directs the sleeve process. We call the



Curve **Knuckles** **Vertices** **Mesh**

KNUCKLE DIVISIONS

The number of divisions required is a function of the length and average radius. For our stitching method, we need an even number of divisions (multiple of 2). A calibrated equation is used, that ensures that all triangles are roughly equilateral.

```
// Compute number of divisions for sleeve
double divisions = Math.Max((Math.Round(length * 0.5 / avgRadius) * 2), 2);
```

GENERATE VERTICES

By default, we use 6 vertices per plate. On each plate, we rotate the set of vertices about 1/12th of a rotation. On the ends of the sleeve, the vertices are shared between the sleeves and hulls, so we save them as plate vertices.

GENERATE MESHES

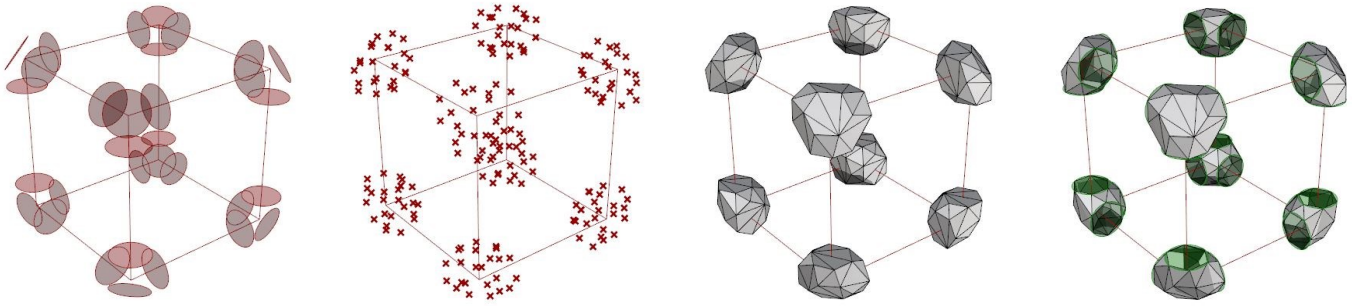
To generate the mesh faces, we stitch the vertices. We use the order of the vertices to do this correctly.

2.2.5 HULL MESHING

The first step is to **gather all points** for the 3D convex hull. Recall that we have created a plate layout that ensures all points will lie on the hull (none will be engulfed), so let's call the points vertices. We use the plate vertices created in previous steps, as well as the extra plate vertices (if sharp node).

At this stage, we have everything we need to compute the **3D convex hulls**. We use the We implement an incremental convex hull algorithm described in "Computation Geometry—Applications and Algorithms" by Mark de Berg et al. We also borrowed some of the seen-face conditions from Exoskeleton by David Stasiuk. In terms of optimization, we don't use a conflict graph to reduce computation, since our convex hulls all generally deal with a low number of points, so benefits would be negligible. *Refer to the code for more information.* (i.e. MakeConvexHull() method in the ExoMesh class)

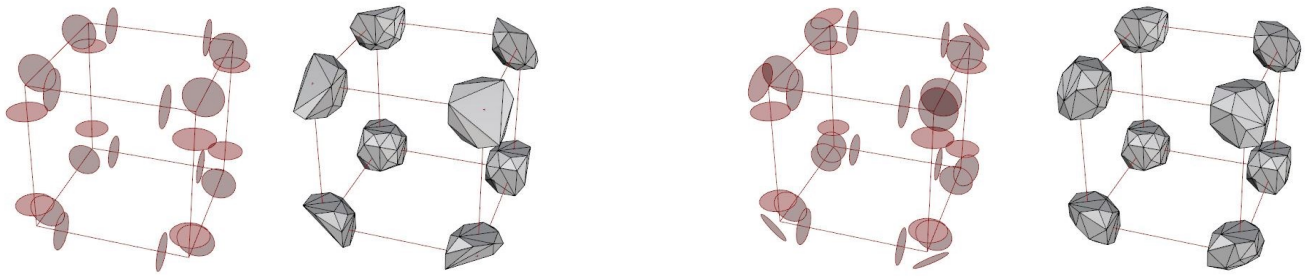
After the convex hulling process, we **clean the hull**, removing the faces that lie on the plates, so that the sleeves can connect directly to the hulls and form a single solid mesh. To do this, we store a list all plate vertices (excluding the extra plate). Any mesh face that has vertices strictly from this list, is removed. Again, the fact that we deal with a relatively low number of vertices makes this process less expensive than you would think.



Plates **Vertices** **Convex hulls** **Cleaned hulls**

VISUALIZING "SHARP" NODES

The reason for extra plates at sharp nodes is shown below. This example shows a moderate case: on the left, there is no extra plate, and the hull is flat cornered. In fact, you can even see the actual node poking out of the hull, so the node isn't even contained in the hull. To address that, we add an extra plate.
The position and orientation computations were discussed previously.



without extra plate **with extra plate**

Section 3

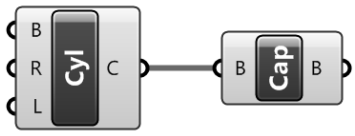
HOW-TO

Components compatible with the Intralattice framework need to meet certain I/O requirements. When developing a new component, keep these constraints in mind.

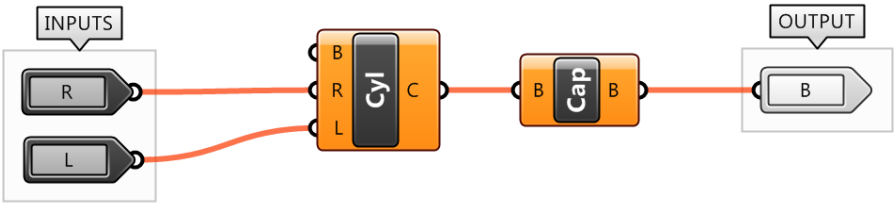
The modular framework is quite straight-forward.

3.0 CREATING USER OBJECTS (.GHUSER)

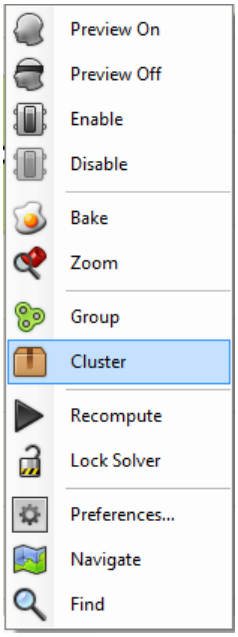
Grasshopper “User Objects” are essentially encapsulated Grasshopper algorithms, or black-box components. This is a great feature that allows you to quickly test algorithms and create prototype components.

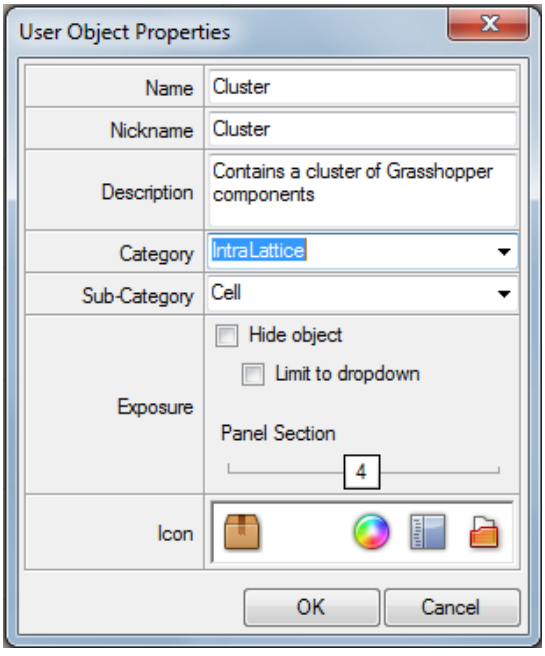


Next, create a ‘**Cluster Input**’ component for each input your component will have, and connect it to it’s corresponding parameter. Same for ‘**Cluster Output**’ components. By double clicking on these components, you can set the Name, Nickname and Description for your new component.



Next, select all components, including inputs and outputs, right-click, and select ‘Cluster’. Your algorithm will be encapsulated into a single ‘Cluster’ component. The final step is to save the newly created cluster as a User Object. To do this, select your cluster, and go to ‘File -> Create User Object’. You will be prompted to enter some properties and select an icon. Your user object .GHUSER file can then be found in ‘File -> Special Folders -> User Objects Folder’.





3.1 CREATING ASSEMBLIES (.GHA)

Grasshopper assemblies are custom components written in C# with the RhinoCommon SDK.

Section 4

GITHUB REPO

Components compatible with the Intralattice framework need to meet certain I/O requirements. When developing a new component, keep these constraints in mind.

The modular framework is quite straight-forward.