INTRALATTICE CORE  
Version 0.7.5 - Alpha

**Developer Documentation**

This documentation serves as a guide for people who want to contribute to the Intralattice project, or are simply curious about the algorithms behind it. This document assumes you know what Intralattice is, and its basic workflow. That is to say, it assumes you are familiar with the User documentation.

**Table of Contents**

[1 FRAMEWORK REQUIREMENTS 2](#_Toc425346700)

[2 ALGORITHM OVERVIEW 3](#_Toc425346701)

[2.1 CELL MODULE 3](#_Toc425346702)

[2.2 FRAME MODULE 4](#_Toc425346703)

[2.2.1 Data Structure 5](#_Toc425346704)

[2.2.2 Node Mapping 7](#_Toc425346705)

[2.2.3 Strut Mapping 8](#_Toc425346706)

[2.3 MESH ALGORITHM 10](#_Toc425346707)

[3 CREATING USER OBJECTS 11](#_Toc425346708)

[4 CREATING C# ASSEMBLIES 12](#_Toc425346709)

[5 GITHUB REPOSITORY 13](#_Toc425346710)

# FRAMEWORK REQUIREMENTS

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

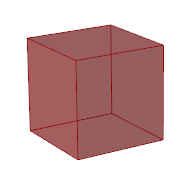
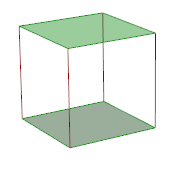
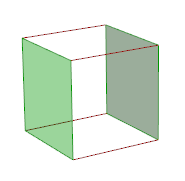
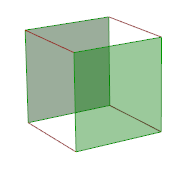
|  |  |  |
| --- | --- | --- |
| Module | Input | Output |
| CELL *Generates the unit cell topology* | No requirements |  |
| FRAME *Generates the lattice wireframe* |  |  |
| MESH *Generates the solid lattice* |  |  |

# ALGORITHM OVERVIEW

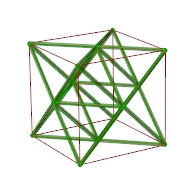
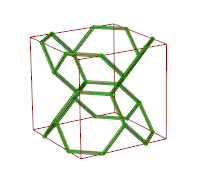
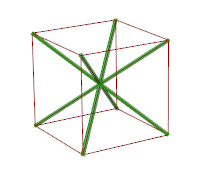
*In this section, we highlight some of the main concepts and algorithms behind the INTRALATTICE Core modules.*

## CELL MODULE

A valid unit cell has the following property: opposing faces of the bounding box are identical. 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.

Some examples of valid unit cells are shown below.

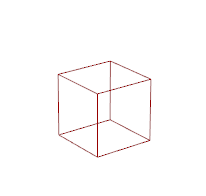


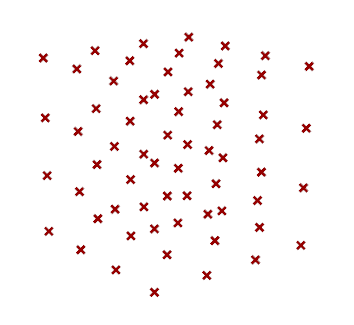
Randomization…

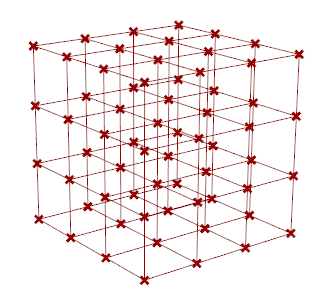
Symmetry…

## FRAME MODULE

*The frame algorithms all work in three steps. First, the cell topology input is formatted to a specific data structure. Second, the node grid is generated. Finally, the topology is mapped to this grid.*







### Data Structure

The first concept to introduce is the **UVW cell grid**. This is implemented in a data tree structure, 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. Expanding on the UVW concept, each cell can potentially have many nodes, and so we extend the data tree with an extra set of children, in the form of the **UVWI node grid**, which serves as our data structure for the lattice nodes. In other words, each node has a unique data path (u,v,w,i). This allows us to very quickly map relationships to this node grid, creating a lattice. To avoid creating duplicate nodes (and struts), we use the **UnitCell** Object Class to represent the unit cell topology.

|  |
| --- |
| // UnitCell C# Object Overview  Point3dList Nodes // List of unique nodes  List<IndexPair> NodePairs // List of lines as node index pairs  List<List<int>> NodeNeighbours // List of node adjacency lists (parallel to Nodes list)  List<int[]> NodePaths // List of relative paths in tree (parallel to Nodes list) |

Let’s have a look at these fields progressively, building up the data as we do in the algorithm. The process of breaking down the topology into a malleable format involves 4 steps, summarized below.

|  |
| --- |
| // Pre-process the ‘topology’ input parameter (a list of lines)  var cell = new UnitCell();  CellTools.FixIntersections(ref topology); // 1  CellTools.ExtractTopology(ref topology, ref cell); // 2  CellTools.NormaliseTopology(ref cell); // 3  CellTools.FormatTopology(ref cell); // 4 |

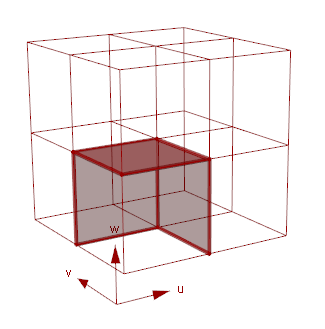
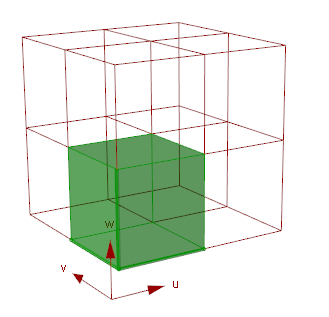
**The first step** is to fix any intersection issues, by splitting all lines which intersect each other; the intersection points are in fact nodes, which need to be defined.   
**The second step** starts building the UnitCell object. The list of lines is converted to a list of unique nodes (i.e. Nodes), and lines between these nodes are stored in two parallel lists, which each have their own purposes:

* List of node index pairs (i.e. NodePairs)
* List of node adjacency lists (i.e. NodeNeighbours)

**The third step** simply normalizes the unit cell. The cell is scaled to unit size (i.e. 1x1x1 bounding box), and moved to the origin (i.e. minimum corner at origin). To do this, the cell.Nodes values are modified.

**The fourth step** involves computing the relative tree paths. To understand what this means, let’s consider how the topology 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 the concept of **relative paths**, consider the following illustrations. The green area is the current cell. The red area is the positive boundary of the cell – nodes lying in this area belong to neighbouring cells, so we must define their relative path in the tree.



To determine the relative path, we perform the following node location checks, in an else-if fashion. The relative paths corresponding to each case is shown below the image. The ‘i’ index will be determined based on the assumption that opposing faces are identical (cell is valid).

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |
| (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) |

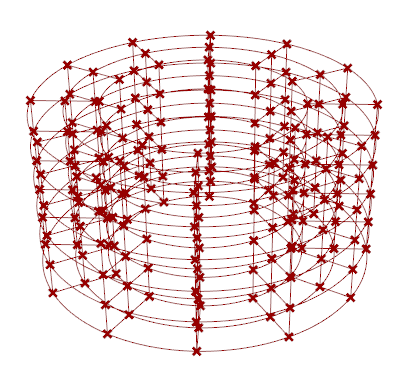
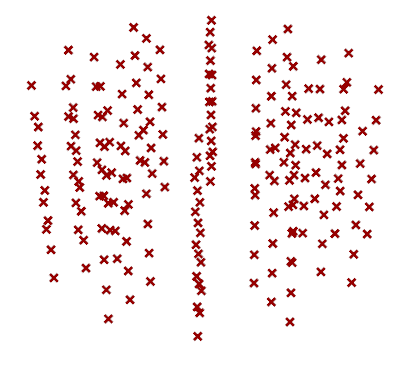
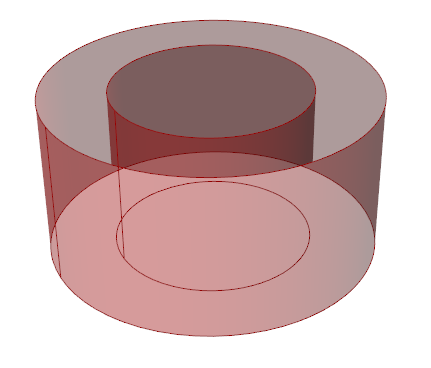
All other nodes have a relative path of (0,0,0,i), meaning they belong to the current cell. To determine the ‘i’ index for nodes which do not belong to the current cell, we rely on the assumption that opposing faces have identical nodes (cell is valid). Therefore, the Cell module must ensure the validity of the cell before passing it to the frame component.

In any case, by the end of this process, we have a valid UnitCell object, ready to be mapped to the design space.

### Node Mapping

Once the unit cell is translated to a UnitCell object, we can map the node grid. There are different ways of doing this. Let’s first consider the simplest example possible: a box lattice, which is a uniform tessellation of unit cells in the 3 orthogonal directions (i.e. 3D Cartesian).

#### Conformal mapping



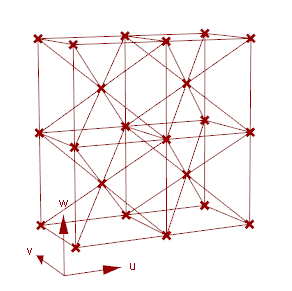
#### Uniform mapping

Conform : - Surface-Surface, Surface-Axis, Surface-Point  
- No morphing, space morphing, bezier morphing

Uniform: - Trimmed by

Random: - Fully randomized

### Strut Mapping

Before we go on, some definitions:

- **Template cell** refers to the input cell (a list of lines). It is not part of the final lattice.  
- **Mapped cell** refers to each cell in the actual lattice. This cell may be deformed, or partially trimmed.

The frame components expect a template cell as input. The only requirement imposed on this cell is that it has mirror faces, meaning opposing faces must be identical.

Each cell is based on a single corner point

From here, the unit cell topology is mapped to the grid using simple node adjacency relationships. It’s important to note that

represent the lattice nodes in a **UVWI grid**. Essentially, the path takes the form (u, v, w, i), where the ‘i’ index represents all the nodes belonging to the uvw-cell.

Although the algorithms are based on cubic unit cells, conformal mapping will generally deform these cells quite a bit. In any case, to make this easy to visualize, let’s consider an example where the design space is a cube. In this case, conformal mapping won’t deform the cells.

## MESH ALGORITHM

# CREATING USER OBJECTS

Grasshopper “User Objects” are essentially encapsulated Grasshopper clusters, or black-box components. You can use this to test new concepts and design prototype components.

# CREATING C# ASSEMBLIES

# GITHUB REPOSITORY

The Github repository is used to version control the repository. If you are new to Git and Github, I highly recommend downloading the new [Github for Windows](https://windows.github.com/), which provides an awesome graphical user interface.

RULES FOR COLLABORATORS

1) **Never push directly to the master branch or release branch.** Always work in a feature/fix branch, and submit a pull request, with a short title, and a comment that details exactly what has been added/changed. Do not merge your pull request, we will have **one person in charge of managing** the repository, and only they should merge pull requests. This person will also ensure cohesion between new release versions and the documentation. For now, I will be that person, since I'm actively working on the docs and website.

2) **When you modify someone else's code, feel free to credit yourself**. If you add a fix or modify someone's code, add your name to the header comment. For example,  in my files, I always have a header with something like "Written by Aidan Kurtz"; that's where you'd something like "Modified by John Doe". Anyone who knows how Github works can also view the history of the file and see who wrote what, but adding your name to the header makes the attribution independent of the version control.

3) **Comment your code well.** If you submit a pull request and the manager doesn't understand your code, they won't merge it. Since this is open-source, we don't just want code that works, we want code that can be built-upon. Don't put off commenting, even you will forget how your code works.

4) **Use C# naming conventions.** Parameter names should be camelCase, and everything else should be PascalCase. [More on that](https://msdn.microsoft.com/en-us/library/vstudio/ms229002(v=vs.100).aspx)