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 OPTIMIZE 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

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.

Section 2

ALGORITHMS

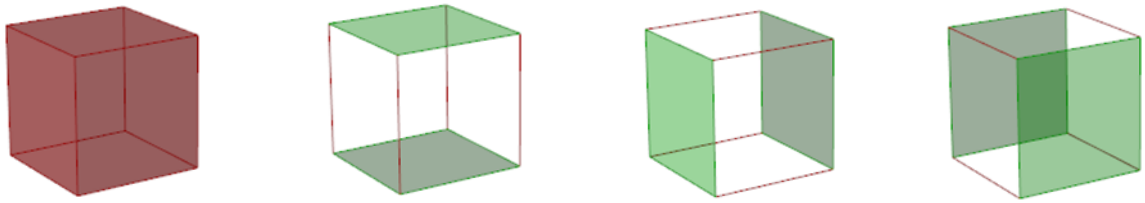
Components compatible with the Intralattice framework need to meet certain I/O requirements. When developing a new component, keep these constraints in mind.

The modular framework is quite straight-forward.

2.0 CELL MODULE

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed pretium, felis quis pretium interdum, augue est ultrices leo, et ornare urna lorem eget mi. Fusce sit amet sollicitudin eros. Integer nec magna eget magna volutpat vestibulum. Vestibulum varius ipsum ut nisl mollis lobortis. Nulla ut accumsan neque. Nullam eget tellus vulputate, porta nunc non, malesuada lorem. Suspendisse ac accumsan felis, ac lacinia lorem. Proin ut justo a enim sodales suscipit. Fusce vel nisl a metus viverra fermentum. Fusce iaculis libero non felis maximus gravida. Mauris at ipsum risus.

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.



2.1 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.

```
// UnitCell Object

Point3dList Nodes
List<IndexPair> NodePairs
List<List<int>> NodeNeighbours
List<int[]> NodePaths
```

2.1.0 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.

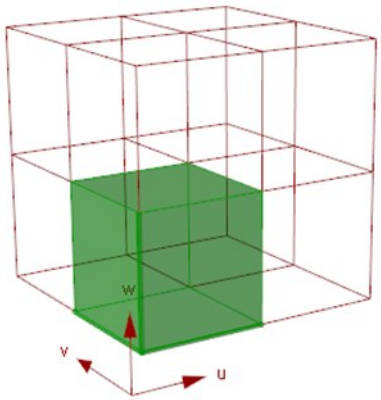
The second step starts building the UnitCell object. The list of lines is converted to a list of unique nodes (i.e. Nodes), and lines between these nodes are stored in two parallel lists, which each have their own purposes:

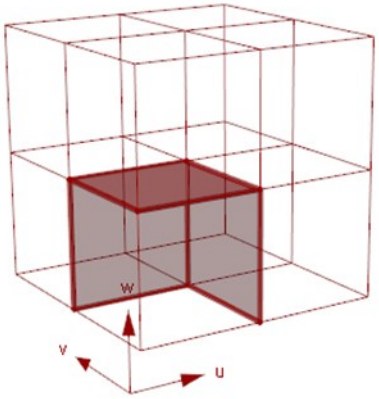
- List of node index pairs (i.e. NodePairs)
- List of node adjacency lists (i.e. NodeNeighbours)

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.

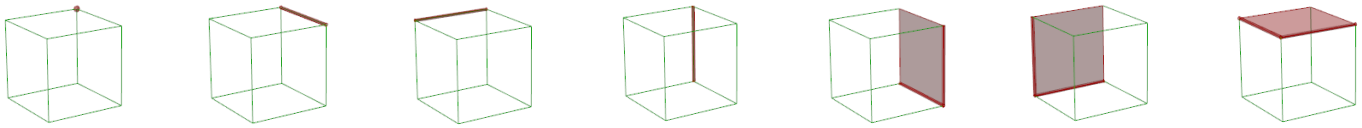
The fourth step involves computing the relative tree paths. To understand what this means, let's consider how the topology can be represented efficiently. Within a lattice, cells will always have neighbours, and so we need a way to make sure neighbours can share their interface nodes and struts, without 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.



(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.

2.1.2 NODE MAPPING

Once the unit cell is translated to a UnitCell object, we are ready to map the grid of nodes to the design space. 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 3 orthogonal directions (i.e. 3D Cartesian).

NODE MAPPING — UNIFORM

Once the unit cell is translated to a UnitCell object, we are ready to map the grid of nodes to the design space. 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 3 orthogonal directions (i.e. 3D Cartesian).

NODE MAPPING — CONFORM

2.1.3 STRUT MAPPING

Once the unit cell is translated to a UnitCell object, we are ready to map the grid of nodes to the design space. 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 3 orthogonal directions (i.e. 3D Cartesian).

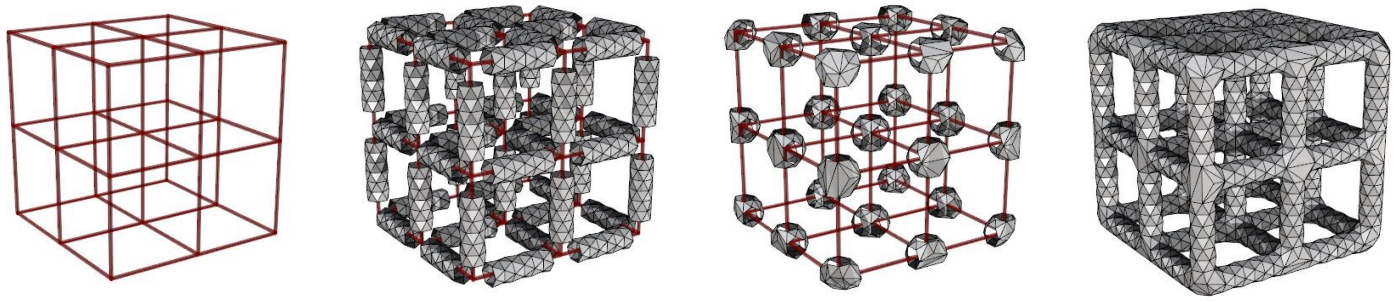
STRUT MAPPING — UNIFORM

Once the unit cell is translated to a UnitCell object, we are ready to map the grid of nodes to the design space. 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 3 orthogonal directions (i.e. 3D Cartesian).

STRUT MAPPING — CONFORM

2.2 MESH MODULE

The meshing module thickens a line network as a mesh.



The process can be summarized as follows:

0 Clean network

- Remove duplicate struts and nodes.
- Remove null, invalid or very small struts.

1 Data structure :

- Construct
- Set nodal radii
- Compute offsets

2 Sleeve meshing :

- Generate vertices
- Stitch vertices into mesh faces

3 Hull meshing :

2.2.0 CLEAN NETWORK

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.

2.2.1 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.

2.2.2 SET NODAL RADII

2.2.3 COMPUTE OFFSETS

2.2.4 SLEEVE MESHING

2.2.5 HULL MESHING

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 (.GHO)

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.

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.