LARCC — LINEAR ALGEBRAIC REPRESENTATION FOR COMPUTING WITH COCHAINS

LARCC Team

January 9, 2014

Contents

Chapter 1

Literate programming IDE for LAR-CC

This document introduces the developer of geometric libreries and applications to the integrated development environment (IDE) set up for documentation and multilanguage development using LAR-CC, the Linear Algebraic Representation for geometry, manufacturing and physics with *Chains* and *CoChains*. This IDE is strongly based on the literate programming tool *Nuweb*, aiming at embedding the code in the documentation, and not vice-versa. The main goal of this framework is to facilitate how to express the *why* of software design decisions, and not only the tricky details of low level coding. I would recommend writing programs as if they were research papers and treat the code as you would write mathematical expressions in a research paper. Using multiple programming languages is allowed and even encouraged in `larcc`. When possible, the same functions coded in different languages should stay close within the same document subsection. The `larcc` IDE integrates a few programming, documentation and version control tools, including LaTeX, *Nuweb*, *Pandoc*, *Git*, and Leo editor.

1.1 Up and running

1.1.1 Prerequisites

LaTeX The `larcc` IDE requires the users to embed the compute code within LaTeX files written for documenting their work. Therefore the first requirement is a working LaTeX environment. "As TeX Live is the basis of MacTeX, and is the TeX system for Unix, if you work cross-platform and want an identical system on all of your machines, then TeX Live is the way to go" [**?**].

Python As of today, most of `larcc` development was done in Python. Hence a working Python environment is required, including three packages: `scipy`, `pyopengl`, and `pyplasm`. On a Mac, Python is installed by default, whereas `scipy` and `pyopengl` may be installed in the terminal by doing

```
$ sudo easy-install scipy
$ sudo easy-install pyopengl
```

Finally, to install `pyplasm` look at the README file in its downloaded directory:

```
$ git clone https://github.com/plasm-language/pyplasm
```

Nuweb In 1984, Knuth introduced the idea of literate programming. The idea was that a programmer writes just one document, the web file—with suffix `.w`, that combines the documentation with the code. *Nuweb* works with any programming language and LaTeX, and is probably the simplest incarnation of the Knut's original work. The web site of the tool is sourceforge.net/projects/nuweb/. A revised version of source files can be found on code.google.com/p/nuweb. This package can build using the standard tools:

```
$ cd <path-to>/nuweb/
$ ./configure
$ make
$ sudo make install
```

For some documentation read the wiki page. Test your installation by just compiling to *pdf* the `nuweb.w` document itself, whose chapter one contains the user documentation:

```
$ nuweb nuweb.w
```

Of course, in order to extend `larcc` and/or to make an efficient use of it, you are supposed to read carefully the first chapter of the *nuweb.pdf* document.

1.1.2 Download

You may or may not put your IDE under the protection of a version control system. The `larcc` project comes from *Github* equipped with an integrated *git* system, that you are free either to use or not to use. Of course, my advice is of making the best use of it. Hence write in the terminal:

```
$ git clone https://github.com/cvdlab/larcc
```

Thats all. Now move to the `larcc` directory and give a look at its content, a bunch of directories, subdirectories and files, of course. Write

```
$ tree -L 2
```

and you will see something like

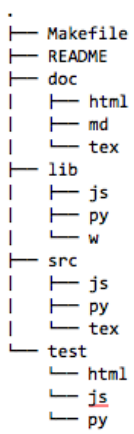

Figure 1.1: A sample of the `larcc` framework

1.1.3 Working-file

For the impatient, open a terminal, and change directory to `larcc`, the root of the LARCC project, if not already there, whenever you hold it on your system. Then create your working-file `src/tex/<name>.tex` as a copy of `src/tex/template.tex`. Open the working-file with an editor, possibly with one aware of the LaTeX syntax, and make few mandatory changes on the template text:

Title change the `title` command, substituting the "Title" string with the actual sentence to be used as document title;

Author do the same for the `author` command, substituting the "TheAuthor" string with the actual document author;

Bibliography substitute the "template" string with the actual working-file name (without the file extension).

Finally you may starting the real work by writing the documentation/code within your `<name>.tex` file, using the simple mark-up rules of Nuweb.

1.1.4 Using the IDE

In short, in order to use the development environment, you must (a) open your terminal and move to the `larcc` directory, (b) write a `tex` file, including documentation and suitably tagged computer code, (c) save it in the `src` subdirectory, and (d) execute a `make` command, asking either for generation of the `pdf` or the `html` documentation, or execution of unit testing, or simply for the compilation of the source code.

Using Leo

Leo is a multi-platform and open-source outliner and *non-linear* text editor allowing for fundamentally different ways of using and organizing data, programs and scripts. Leo has been under active development for 15 years and has an active group of developers and users. The Leo environment allow to structure either simple or very complex projects as an acyclic graph, where nodes (said clones), and hence the subgraphs rooted in them, may have more than one father. Accordingly, every update to a clone node immediately extends to every its instances within the tree-like walk-through of the whole outline.

Using make

When using the IDE, the user must open the `Makefile` with any text editor, and modify the current values of two user-definable variables, according to his will:

```
NAME = <name>
LANGUAGE = <language>
```

where `<name>` is name of the new working document, and `<language>` may be only `py` (for Pyton). Soon such values will be extended to include `<js>` for Javascript and `<lhs>` for Literate Haskell. The make *targets* currently available in the Makefile are the following:

all is the default option. Its execution produces a *pdf* document in the `doc/pdf` subdirectory, and a pair of *tex*/*bbl* documents in the `doc/tex` subdirectory, all using name `<name>`;

html similar to the default option, but produces a directory named `<name>` wth a bunch of *html* pages, located in the `doc/html` subdirectory;

test to execute the tests contained in the directory `test/<language>/<name>`

exec to execute *nuweb* on the working document, i.e. on `<name>.tex`[1]; this execution generates both the LaTeX documentation file and the source output files (for example the unit tests) written in the coding `<language>`;

clear to remove all the working files from the root directory. Used by other commands. To be invoked by the user just in case that something did not work out.

[1] Actually, `<name>.tex` is internally copied to a scratch file `<name>.w`, in order to allow the user to work comfortably with an editor knowledgeable of the LaTeX syntax.

1.2 Structure of larcc

The larcc project is hinged around four subdirectories (see Figure 1.1) and a Makefile. The meaning and function of the four subdirectories are listed below.

src (for *Source*) is the directory src that contains all the source documents, and in particular the tex files including the code of the algorithms and the tests developed in the project. It is divided in subdirectories related to the type of the source file itself. For example a html directory will contain the user-defined css source files, and the lhs directory the *literate Haskell* sorce files, to be processed directly by the Haskell compiler *GHC*. Such directories will also contain other programming resources needed to build the libreries or the applications developed in the LAR-CC project.

test test-driven development (TDD) is a software development process that relies on the repetition of a very short development cycle: write a "unit test", get it to pass, run all tests, clean up the code (see Figure 1.2). The subdirectory test is the repository of test suites, collection of test cases, and of unit test files, possibly grouped depending on the source language, to be launched either individually, while writing each single software function or application, or collectively before committing or pushing novel developments or subsystems.

lib is the repository of compiled and/or executable programs. In particular, it is the place where to store and retrieve all the libreries or modules or applications developed by compilation of any document within the scr subdirectory of the larcc system, excluding the documentation.

doc conversely contains all the documentation generated by the system, once again subdivided depending on the language and tools used for its reading or examination.

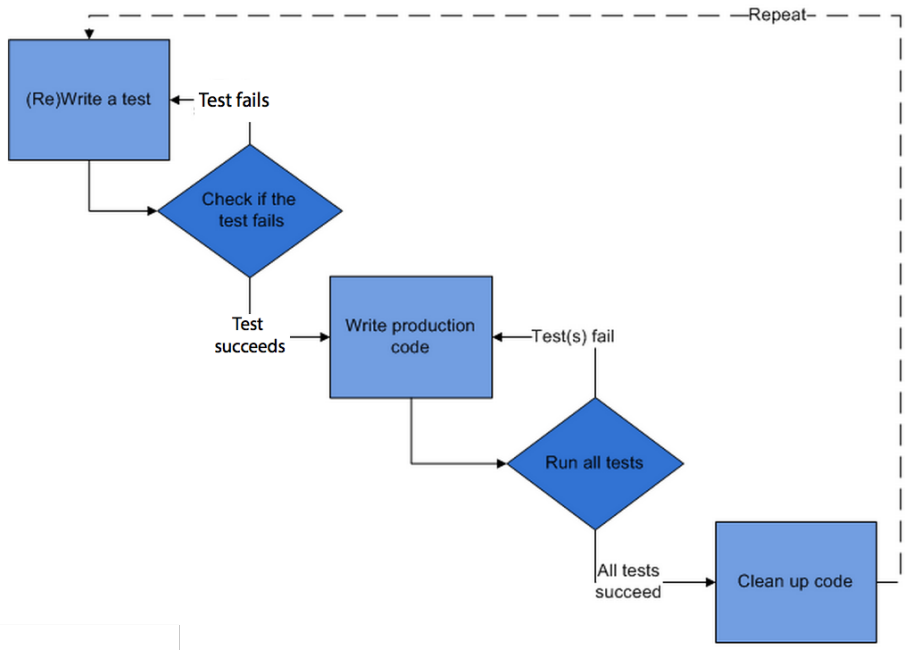

Figure 1.2: test-driven development (TDD) cycle (from Wikipedia)

Bibliography

Chapter 2

Module Lar2psm

This software module contains all the functions needed to interface the LAR data structure and/or the geometric objects defined by it with the Plasm environment. In particular, it will include the interfaces towards the visualization primitives provided by the language.

2.1 Introduction

The standard definition of vectors and matrices in plasm is the list of vector coordinates and the list of matrix rows, respectively.

2.2 Implementation

Since the present `lar2psm` module is an interface between the `larcc` library and the PLaSM language, and its various incarnations, it should allow to import the language itself (in Python, the `pyplasm` module).

nuweb2a ⟨ Import the pyplasm module 2a ⟩ ≡

```
from pyplasm import *
```

.

An useful utility will allow for the creation of a subdirectory from a `dirpath` *string*.

nuweb2b ⟨ Create directory from path 2b ⟩ ≡

```
import os
def createDir(dirpath):
    if not os.path.exists(dirpath):
        os.makedirs(dirpath)
```

nuweb2c2cnuweb6a, 6a.

It may be useful to define the repository(ies) for the unit tests associated to the module:

nuweb2c `"test/py/lar2psm-tests.py"` 2c ≡

```
⟨ Create directory from path nuweb2b2b ⟩
createDir('test/py/lar2psm/')
```

2.2.1 Convex combination

Next we define the `CCOMB` function that accepts as input a `vectors` list (i.e., a matrix) and returns *the* point their convex combination.

nuweb2d ⟨ Compute the convex combination of a list of vectors 2d ⟩ ≡

```
import scipy as sp
from pyplasm import *
def CCOMB(vectors):
    return (sp.array(VECTSUM(vectors)) / float(len(vectors))).tolist()
```

nuweb5d5d.

Unit tests First we test `CCOMB` with some special data, then with some random vectors.

nuweb3a `"test/py/lar2psm/test-ccomb.py"` 3a ≡

```
⟨ Import the module (nuweb3b3b lar2psm ) nuweb5a5a ⟩
from lar2psm import *
⟨ CCOMB unit tests nuweb77 ⟩
```

2.2.2 LAR model of a cell complex

A very important concept introduced by the LAR package is the definition of the *model* of a cell complex, as a pair made by a list of vertices, given as lists of coordinates, and a topological relation.

Definition 1 (LAR model)**.** *A* LAR model *is a pair, e.g. a Python tuple* (V, FV)*, where:*

1. *`V` is the list of vertices, given as lists of coordinates;*
2. *`FV` is a* cell-vertex *relation, in this case the face-vertex relation, given as a list of cells, where each cell is given as a list of vertex indices.*

Examples Some very simple examples of 0D, 1D, and 2D models follows. They are displayed in Figure 2.1.

nuweb3c ⟨ 2D model examples 3c ⟩ ≡

```
V = [[0.,0.],[1.,0.],[0.,1.],[1.,1.],[0.5,0.5]]
VV = [[0],[1],[2],[3],[4]]
EV = [[0,1],[0,2],[0,4],[1,3],[1,4],[2,3],[2,4],[3,4]]
FV = [[0,1,4],[1,3,4],[2,3,4],[0,2,4]]

model0d, model1d, model2d = (V,VV), (V,EV), (V,FV)
```

nuweb6d6d.

2.2.3 Function MKPOLS

The function MKPOLS returns a list of HPC objects, i.e. the geometric type of the PLaSM language. This list is generated to be displayed, possibly exploded, by the pyplasm viewer.

Each cell f in the model (i.e. each vertex list in the FV array of the previous example) is mapped into a polyhedral cell by the pyplasm operator MKPOL. The vertex indices are mapped from base 0 (the Python and C standard) to base 1 (the Plasm, Matlab, and FORTRAN standard).

nuweb4a ⟨ MaKe a list of HPC objects from a LAR model 4a ⟩ ≡

```
def MKPOLS (model):
    V, FV = model
    pols = [MKPOL([[V[v] for v in f],[range(1,len(f)+1)], None]) for f in FV]
    return pols
```

nuweb5d5d.

Unit tests Some simple 3D, 2D, 1D and 0D models are generated and visualised exploded by the file

nuweb4b "test/py/lar2psm/test-models.py" 4b ≡

⟨ Import the module (nuweb4c4c lar2psm) nuweb5a5a ⟩
⟨ View model examples nuweb6d6d ⟩

2.2.4 "Explosion" of the scene

A function EXPLODE used to "explode" an HPC scene defined as a *list* of HPC values, given three real scaling parameters, sx,sy,sz, that are used to transform the position of the centroid of each HPC cell. HPC stands for *HierarchicaL Polyhedral Complex*, the type of plasm geometric values. Of course the assertion

$$sx, sy, sz \geq 1.0$$

must be true, otherways the function would induce some compenetration of the cells of the scene.

nuweb4d ⟨ Explode the scene using sx,sy,sz scaling parameters 4d ⟩ ≡

```
def EXPLODE (sx,sy,sz):
    def explode0 (scene):
        centers = [CCOMB(S1(UKPOL(obj))) for obj in scene]
        scalings = len(centers) * [S([1,2,3])([sx,sy,sz])]
        scaledCenters = [UK(APPLY(pair)) for pair in
                         zip(scalings, [MK(p) for p in centers])]
        translVectors = [ VECTDIFF((p,q)) for (p,q) in zip(scaledCenters, centers) ]
        translations = [ T([1,2,3])(v) for v in translVectors ]
        return STRUCT([ t(obj) for (t,obj) in zip(translations,scene) ])
    return explode0
```

nuweb5d5d.

The `EXPLODE` function is second order: it first application (to the scaling parameters) returns a partial function to be applied to the `scene`, given as a *list* of HPC (Hierarchical Polyhedral Complex) objects. `EXPLODE` is dimension-independent, since it can be applied to points, edges, faces, 3D cells, and even to geometric values of mixed dimensionality (see Figure 2.1).

It works by computing the centroid of each object, and by applying to each of them a translation equal to the difference betwwen the scaled and the initial positions of its centroid. `EXPLODE` returns a single HPC object (the assembly of input objects, properly translated)

2.3 Source Output: `lar2psm` module

2.3.1 Importing a generic module

First we define a parametric macro to allow the importing of `larcc` modules from the project repository `lib/py/`. When the user needs to import some project's module, she may call this macro as done in Section 2.3.2.

nuweb5a ⟨ Import the module 5a ⟩ ≡

```
import sys
sys.path.insert(0, 'lib/py/')
import @1
```

nuweb3a3anuweb4b, 4bnuweb5b, 5bnuweb5dd.

Importing a module A function used to import a generic `lacccc` module within the current environment is also useful.

nuweb5b ⟨ Function to import a generic module 5b ⟩ ≡

```
def importModule(moduleName):
    ⟨ Import the module (nuweb5c5c moduleName ) nuweb5a5a ⟩
```

nuweb5d5d.

2.3.2 Lar2psm exporting

Here we assemble top-down the `lar2psm` module, by orderly listing the functional parts it is composed of. Of course, this one is the module version corresponding to the current state of the system, i.e. to a very initial state. Other functions will be added when needed.

nuweb5d "lib/py/lar2psm.py" 5d ≡

```
"""Module with functions needed to interface LAR with pyplasm"""
⟨ Import the module (nuweb5e5e simplexn ) nuweb5a5a ⟩
⟨ Function to import a generic module nuweb5b5b ⟩
⟨ Compute the convex combination of a list of vectors nuweb2d2d ⟩
⟨ MaKe a list of HPC objects from a LAR model nuweb4a4a ⟩
⟨ Explode the scene using sx,sy,sz scaling parameters nuweb4d4d ⟩
```

2.4 Unit tests

2.4.1 Creation of repository of unit tests

A possible unit test strategy is to create a directory for unit tests associated to each source file in `nuweb`. Therefore we create here a directory in `test/py/` with the same name of the present document. Of course other

nuweb6a ⟨ create directory and echo of creation 6a ⟩ ≡

```
⟨ Create directory from path nuweb2b2b ⟩
createDir('@1')
print "'@1' repository created"
```

.

nuweb6b "test/py/lar2psm/test01.py" 6b ≡

```
⟨ create directory and echo of creation: (nuweb6c6c test/py/lar2psm/ ) ? ⟩
```

2.4.2 Viewing some simplicial complexes

Let we start producing some images, displayed in Figure 2.1, os a small simplicial complex and of its skeletons. Notice that the `+` character operates the join of lists (of HPC values).

nuweb6d ⟨ View model examples 6d ⟩ ≡

```
from lar2psm import *
⟨ 2D model examples nuweb3c3c ⟩
explode = EXPLODE(1.5,1.5,1.5)
VIEW(explode(MKPOLS(model0d)))
VIEW(explode(MKPOLS(model1d)))
VIEW(explode(MKPOLS(model2d)))
VIEW(explode(MKPOLS(model2d) + MKPOLS(model1d) + MKPOLS(model0d)))
```

nuweb4b4b.

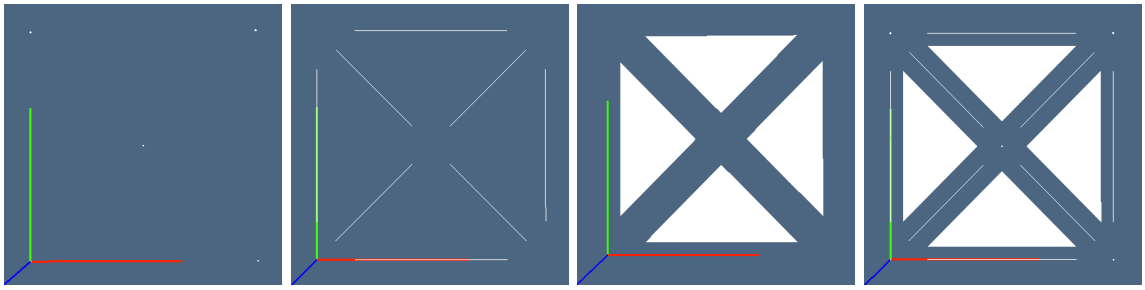

Figure 2.1: Images of the skeletons of a small simplicial complex.

2.4.3 Testing convex combination of vectors

nuweb7 ⟨ CCOMB unit tests 7 ⟩ ≡

```
assert( CCOMB([]) == [] )
assert( CCOMB([[0,1]]) == [0.0, 1.0] )
assert( CCOMB([[0,1],[1,0]]) == [0.5, 0.5] )
assert( CCOMB([[1,0,0],[0,1,0],[0,0,1]]) == [1./3,1./3,1./3])

import random
vects = [[random.random() for i in range(3)] for k in range(4)]
assert( CCOMB([VECTSUM(vects)]) == \
        (sp.array(CCOMB(vects)) * len(vects)).tolist() )
```

nuweb3a3a.

Bibliography

Chapter 3

The `smplxn` module

This module defines a minimal set of functions to generate a dimension-independent grid of simplices. The name of the library was firstly used by our CAD Lab at University of Rome "La Sapienza" in years 1987/88 when we started working with dimension-independent simplicial complexes [**?**]. This one in turn imports some functions from the `scipy` package and the geometric library `pyplasm` [].

3.1 Introduction

The $Simple^n_X$ library, named `simplexn` within the Python version of the LARCC framework, provides combinatorial algorithms for some basic functions of geometric modelling with simplicial complexes. In particular, provides the efficient creation of simplicial complexes generated by simplicial complexes of lower dimension, the production of simplicial grids of any dimension, and the extraction of facets (i.e. of $(d-1)$-faces) of complexes of d-simplices.

3.2 Some simplicial algorithms

The main aim of the simplicial functions given in this library is to provide optimal combinatorial algorithms, whose time complexity is linear in the size of the output. Such a goal is achieved by calculating each cell in the output via closed combinatorial formulas, that do not require any searching nor data structure traversal to produce their results.

3.2.1 Linear extrusion of a complex

Here we discuss an implementation of the linear extrusion of simplicial complexes according to the method discussed in [**?**] and [**?**]. In synthesis, for each d-simplex in the input complex, we generate combinatorially a $(d+1)$-simplicial *tube*, i.e. a chain of $d+1$ simplexes of

dimension $d+1$. It can be shown that if the input simplices are a simplicial complex, then the output simplices are a complex too.

In other words, if the input is a complex, where all d-cells either intersect along a common face or are pairwise disjoints, then the output is also a simplicial complex of dimension $d+1$. This method is computationally optimal, since it does not require any search or traversal of data structures. The algorithm [**?**] just writes the output making a constant number $O(1)$ of operation for each one of its n output d-cells, so that the time complexity is $\Omega(n)$, where $n = d\,m$, being m the number and d the dimension (and the storage size) of the input cells, represented as lists of indices of vertices.

Computation Let us concentrate on the generation of the simplex chain γ^{d+1} of dimension $d+1$ produced by combinatorial extrusion of a single simplex

$$\sigma^d = \langle v_0, v_1, \ldots, v_d, \rangle.$$

Then we have, with $|\gamma^{d+1}| = \sigma^d \times I$, and $I = [0,1]$:

$$\gamma^{d+1} = \sum_{k=0}^{d} (-1)^{kd} \langle v_k, \ldots v_d, v_0^*, \ldots v_k^* \rangle$$

with $v_k \in \sigma^d \times \{0\}$ and $v_k^* \in \sigma^d \times \{1\}$, and where the term $(-1)^{kd}$ is used to generate a chain of coherently-oriented extruded simplices.

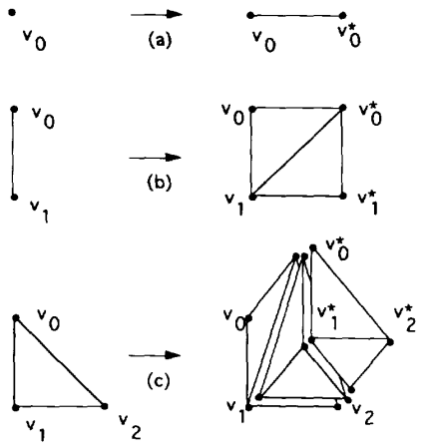

Figure 3.1: Extrusion of (a) a point; (b) a straight line segment; (c) a triangle.

In our implementation the combinatorial algorithm above is twofold generalised:

1. by applying it to all d-simplices of a LAR model of dimension d;
2. by using instead of the single interval $I = [0,1]$, the possibly unconnected set of 1D intervals generated by the list of integer numbers stored in the `pattern` variable

Implementation In the macro below, `larExtrude` is the function to generate the output model vertices in a multiple extrusion of a LAR model.

First we notice that the `model` variable contains a pair (`V`, `FV`), where `V` is the array of input vertices, and `FV` is the array of d-cells (given as lists of vertex indices) providing the input representation of a LAR cellular complex.

The `pattern` variable is a list of integers, whose absolute values provide the sizes of the ordered set of 1D (in local coords) subintervals specified by the `pattern` itself. Such subintervals are assembled in global coordinates, and each one of them is considered either solid or void depending on the sign of the corresponding integer, which may be either positive (solid subinterval) or negative (void subinterval).

Therefore, a value `pattern = [1,1,-1,1]` must be interpreted as the 1D simplicial complex

$$[0,1] \cup [1,2] \cup [3,4]$$

with five vertices `W = [[0.0], [1.0], [2.0], [3.0], [4.0]]` and three 1-cells `[[0,1], [1,2], [3,4]]`.

`V` is the list of input d-vertices (each given as a list of d coordinates); `coords` is a list of absolute translation parameters to be applied to `V` in order to generate the output vertices generated by the combinatorial extrusion algorithm.

The `cellGroups` variable is used to select the groups of $(d+1)$-simplices corresponding to solid intervals in the input `pattern`, and `CAT` provides to flatten their set, by removing a level of square brackets.

nuweb4a ⟨ Simplicial model extrusion in accord with a 1D pattern 4a ⟩ ≡

```
def larExtrude(model,pattern):
    V, FV = model
    d, m = len(FV[0]), len(pattern)
    coords = list(cumsum([0]+(AA(ABS)(pattern))))
    offset, outcells, rangelimit = len(V), [], d*m
    for cell in FV:
        ⟨ Append a chain of extruded cells to outcells nuweb4b4b ⟩
    outcells = AA(CAT)(TRANS(outcells))
    cellGroups = [group for k,group in enumerate(outcells) if pattern[k]>0 ]
    outVertices = [v+[z] for z in coords for v in V]
    outModel = outVertices, CAT(cellGroups)
    return outModel
```

nuweb9b9b.

Extrusion of single cells For each cell in `FV` a chain of vertices is created, then they are separated into groups of $d+1$ consecutive elements, by shifting one position at a time.

nuweb4b ⟨ Append a chain of extruded cells to outcells 4b ⟩ ≡
⟨ Create the indices of vertices in the cell "tube" nuweb4c4c ⟩
⟨ Take groups of d+1 elements, by shifting one position nuweb4d4d ⟩

nuweb4a4a.

Assembling vertex indices in a tube with their shifted images Here the "long" chain of vertices is created.

nuweb4c ⟨ Create the indices of vertices in the cell "tube" 4c ⟩ ≡
```
tube = [v + k*offset for k in range(m+1) for v in cell]
```

nuweb4b4b.

Selecting and reshaping extruded cells in a tube Here the chain of vertices is spitted into subchains, and such subchains are reshaped into three-dimensional arrays of indices.

nuweb4d ⟨ Take groups of d+1 elements, by shifting one position 4d ⟩ ≡
```
cellTube = [tube[k:k+d+1] for k in range(rangelimit)]
outcells += [reshape(cellTube, newshape=(m,d,d+1)).tolist()]
```

nuweb4b4b.

Definition 2 (Big-Omega order)**.** *We say that a function $f(n)$ is* Big-Omega *order of a function $f(n)$, and write $f(n) \in \Omega(g(n))$ when a constant c exists, such that:*

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c > 0, \qquad \textit{where } 0 < c \leq \infty.$$

Theorem 1 (Optimality)**.** *The combinatorial algorithm for extrusion of simplicial complexes has time complexity $\Omega(n)$.*

Proof. Of course, if we denote as $g(n) = nd$ the time needed to write the input of the extrusion algorithm, proportional to the constant length d of cells, and as $f(m) = m(d+1)$ the time needed to write the output, where $m = n(d+1)$, we have

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{m(d+1)}{nd} = \lim_{n\to\infty} \frac{[n(d+1)](d+1)}{nd} = \frac{(d+1)^2}{d} = c > 0$$

□

Examples of simplicial complex extrusions

Example 1 It is interesting to notice that the 2D model extruded in example 1 below and shown in Figure 3.2 is locally non-manifold, and that several instance of the pattern in the z direction are obtained by just inserting a void subinterval (negative size) in the `pattern` value.

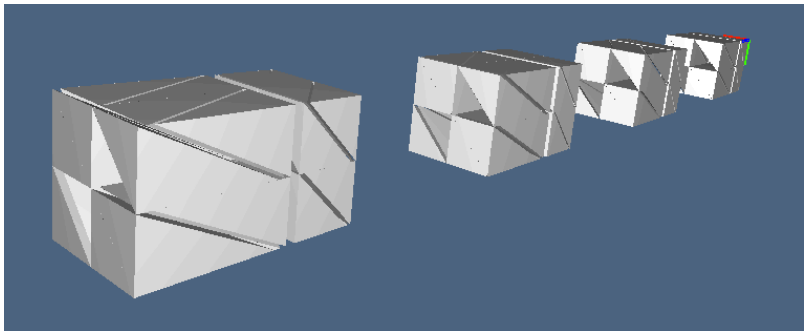

Figure 3.2: A simplicial complex providing a quite complex 3D assembly of tetrahedra.

Examples 2 and 3 The examples show that the implemented `larExtrude` algorithm is fully multidimensional. It may be worth noting the initial definition of the empty `model`, as a pair having the empty list as vertex set and the list `[[0]]` as the cell list. Such initial value is used to define a predefinite constant `VOID`.

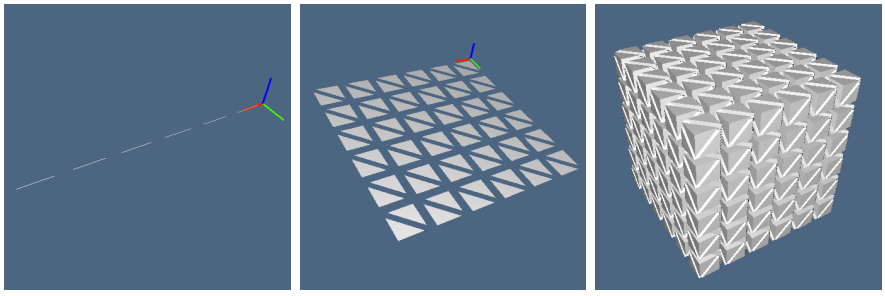

Figure 3.3: 1-, 2-, and 3-dimensional simplicial complex generated by repeated extrusion with the same pattern.

nuweb5 ⟨ Examples of simplicial complex extrusions 5 ⟩ ≡

```
# example 1
V = [[0,0],[1,0],[2,0],[0,1],[1,1],[2,1],[0,2],[1,2],[2,2]]
FV = [[0,1,3],[1,2,4],[2,4,5],[3,4,6],[4,6,7],[5,7,8]]
model = larExtrude((V,FV),4*[1,2,-3])
VIEW(EXPLODE(1,1,1.2)(MKPOLS(model)))

# example 2
model = larExtrude( VOID, 6*[1] )
VIEW(EXPLODE(1.5,1.5,1.5)(MKPOLS(model)))
model = larExtrude( model, 6*[1] )
VIEW(EXPLODE(1.5,1.5,1.5)(MKPOLS(model)))
model = larExtrude( model, 6*[1] )
```

```
VIEW(EXPLODE(1.5,1.5,1.5)(MKPOLS(model)))

# example 3
model = larExtrude( VOID, 10*[1,-1] )
VIEW(EXPLODE(1.5,1.5,1.5)(MKPOLS(model)))
model = larExtrude( model, 10*[1] )
VIEW(EXPLODE(1.5,1.5,1.5)(MKPOLS(model)))
```

nuweb9b9b.

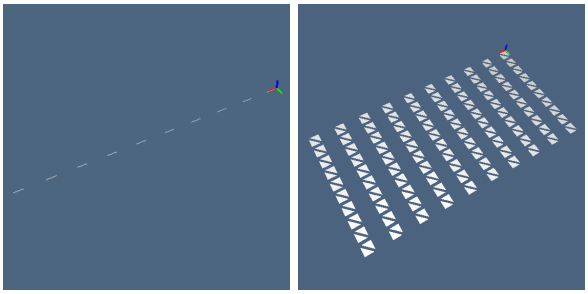

Figure 3.4: 1- and 2-dimensional simplicial complexes generated by different patterns.

3.2.2 Generation of multidimensional simplicial grids

The generation of simplicial grids of any dimension and shape using the `larSimplexGrid` is amazingly simple. The input parameter `shape` is either a tuple or a list of integers used to specify the *shape* of the created array, i.e. both the number of its dimensions (given by `len(shape)`) and the `size` of each dimension k (given by the `shape[k]` element). The implementation starts from the LAR model of the void simplicial complex (denoted as `VOID`, a predefined constant) and updates the `model` variable extruding it iteratively according to the specs given by `shape`. Just notice that the returned grid `model` has vertices with integer coordinates, that can be subsequently scaled and/or translated and/or mapped in any other way, according to the user needs.

nuweb7 ⟨ Generation of simplicial grids 7 ⟩ ≡

```
def larSimplexGrid(shape):
    model = VOID
    for item in shape:
        model = larExtrude(model,item*[1])
    return model
```

nuweb9b9b.

Examples of simplicial grids The two examples of simplicial grids generated by the macro below with `shape` equal to `[3,3]` and `[2,3,4]`, respectively, are displayed in Figure 3.5.

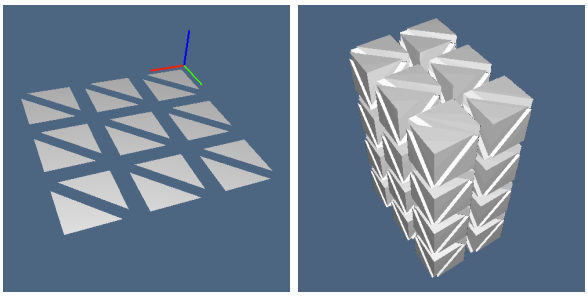

Figure 3.5: 2- and 3-dimensional simplicial grids.

nuweb8a ⟨ Examples of simplicial grids 8a ⟩ ≡

```
grid_2d = larSimplexGrid([3,3])
VIEW(EXPLODE(1.5,1.5,1.5)(MKPOLS(grid_2d)))

grid_3d = larSimplexGrid([2,3,4])
VIEW(EXPLODE(1.5,1.5,1.5)(MKPOLS(grid_3d)))
```

nuweb9b9b.

3.2.3 Facet extraction from simplices

A k-face of a d-simplex is defined as the convex hull of any subset of k vertices. A $(d-1)$-face of a d-simplex

$$\sigma^d = \langle v_0, v_1, \ldots, v_d \rangle$$

is also called a *facet*. Each of the $d+1$ facets of σ^d, obtained by removing a vertex from σ^d, is a $(d-1)$-simplex. A simplex may be oriented in two different ways according to the permutation class of its vertices. The simplex *orientation* is so changed by either multiplying the simplex by -1, or by executing an odd number of exchanges of its vertices.

The chain of oriented boundary facets of σ^d, usually denoted as $\partial\sigma^d$, is generated combinatorially as follows:

$$\partial\,\sigma^d = \sum_{k=0}^{d} (-1)^d \langle v_0, \ldots, v_{k-1}, v_{k+1}, \ldots, v_d \rangle$$

Implementation The `larSimplexFacets` function, for estraction of non-oriented $(d-1)$-facets of d-dimensional simplices, returns a list of d-tuples of integers, i.e. the input LAR

representation of the topology of a cellular complex. The final *list comprehension* is used to remove the duplicated facets, by taking only the last element of any subsequence with possibly duplicated elements.

nuweb8b ⟨ Facets extraction from a set of simplices 8b ⟩ ≡

```
def larSimplexFacets(simplices):
    out = []
    d = len(simplices[0])
    for simplex in simplices:
        out += [simplex[0:k]+simplex[k+1:d] for k in range(d)]
    out = sorted(out)
    return [facet for k,facet in enumerate(out[:-1]) if out[k] != out[k+1]] \
      + [out[-1]]
```

nuweb9b9b.

Examples of facet extraction The simple generation of the LAR model of a simplicial decomposition of a 3D cube as a `larSimplexGrid` with `shape = [1,1,1]` and of its 2D and 1D skeletons is shown here.

nuweb9a ⟨ Examples of facet extraction from 3D simplicial cube 9a ⟩ ≡

```
V,CV = larSimplexGrid([1,1,1])
VIEW(EXPLODE(1.5,1.5,1.5)(MKPOLS((V,CV))))
SK2 = (V,larSimplexFacets(CV))
VIEW(EXPLODE(1.5,1.5,1.5)(MKPOLS(SK2)))
SK1 = (V,larSimplexFacets(SK2[1]))
VIEW(EXPLODE(1.5,1.5,1.5)(MKPOLS(SK1)))
```

nuweb9b9b.

3.2.4 Exporting the $Simple_x^n$ library

The current version of the `simplexn` library is exported here. Next versions will take care of the OpenCL acceleration and data partitioning with very-large size simplicial grids and their sets of faces.

nuweb9b "lib/py/simplexn.py" 9b ≡

```
# -*- coding: utf-8 -*-
"""Module for facet extraction, extrusion and simplicial grids"""
from lar2psm import *
from scipy import *

VOID = V0,CV0 = [[]],[[0]]    # the empty simplicial model
```

⟨ Cumulative sum nuweb1212 ⟩
⟨ Simplicial model extrusion in accord with a 1D pattern nuweb4a4a ⟩
⟨ Generation of simplicial grids nuweb77 ⟩
⟨ Facets extraction from a set of simplices nuweb8b8b ⟩

```
if __name__ == "__main__":
```

⟨ Examples of simplicial complex extrusions nuweb55 ⟩
⟨ Examples of simplicial grids nuweb8a8a ⟩
⟨ Examples of facet extraction from 3D simplicial cube nuweb9a9a ⟩

3.3 Signed (co)boundary matrices of a simplicial complex

3.4 Test examples

3.4.1 Structured grid

2D example

Generate a simplicial decomposition Then we generate and show a 2D decomposition of the unit square $[0,1]^2 \subset \mathbb{E}^2$ into a 3×3 grid of simplices (triangles, in this case), using the `larSimplexGrid` function, that returns a pair `(V,FV)`, made by the array `V` of vertices, and by the array `FV` of "faces by vertex" indices, that constitute a *reduced* simplicial LAR of the $[0,1]^2$ domain. The computed `FV` array is then dispayed "exploded", being ex, ey, ez the explosion parameters in the x, y, z coordinate directions, respectively. Notice that the `MKPOLS` pyplasm primitive requires a pair `(V,FV)`, that we call a "model", as input — i.e. a pair made by the array `V` of vertices, and by a zero-based array of array of indices of vertices. Elsewhere in this document we identified such a data structure as CSR(M_d), for some dimension d. Suc notation stands for the Compressed Sparse Row representation of a binary characteristic matrix.

nuweb10a ⟨ Generate a simplicial decomposition ot the $[0,1]^2$ domain 10a ⟩ ≡

```
V,FV = larSimplexGrid([3,3])
VIEW(EXPLODE(1.5,1.5,1.5)(MKPOLS((V,FV))))
```

nuweb10c10c.

Extract the $(d-1)$-faces Since the complex is simplicial, we can directly extract its facets (in this case the 1-faces, i.e. its edges) by invoking the `larSimplexFacets` function on the argument `FV`, so returning the array `EV` of "edges by vertex" indices.

nuweb10b ⟨ Extract the edges of the 2D decomposition 10b ⟩ ≡

```
EV = larSimplexFacets(FV)
ex,ey,ez = 1.5,1.5,1.5
VIEW(EXPLODE(ex,ey,ez)(MKPOLS((V,EV))))
```

nuweb10c10c.

Export the executable file We are finally able to generate and output a complete test file, including the visualization expressions. This file can be executed by the `test` target of the `make` command.

nuweb10c `"test/py/test01.py"` 10c ≡

⟨ Inport the $Simple^n_X$ library ? ⟩
⟨ Generate a simplicial decomposition ot the $[0,1]^2$ domain nuweb10a10a ⟩
⟨ Extract the edges of the 2D decomposition nuweb10b10b ⟩

3D example

In this case we produce a $2 \times 2 \times 2$ grid of tetrahedra. The dimension (3D) of the model to be generated is inferred by the presence of 3 parameters in the parameter list of the `larSimplexGrid` function.

nuweb11a ⟨ Generate a simplicial decomposition ot the $[0,1]^3$ domain 11a ⟩ ≡

```
V,CV = larSimplexGrid([2,2,2])
VIEW(EXPLODE(1.5,1.5,1.5)(MKPOLS((V,CV))))
```

nuweb11c11c.

and repeat two times the facet extraction:

nuweb11b ⟨ Extract the faces and edges of the 3D decomposition 11b ⟩ ≡

```
FV = larSimplexFacets(CV)
VIEW(EXPLODE(1.5,1.5,1.5)(MKPOLS((V,FV))))
EV = larSimplexFacets(FV)
VIEW(EXPLODE(1.5,1.5,1.5)(MKPOLS((V,EV))))
```

nuweb11c11c.

and finally export a new test file:

nuweb11c `"test/py/test02.py"` 11c ≡

⟨ Inport the $Simple_X^n$ library ? ⟩
⟨ Generate a simplicial decomposition ot the $[0,1]^3$ domain nuweb11a11a ⟩
⟨ Extract the faces and edges of the 3D decomposition nuweb11b11b ⟩

3.4.2 Unstructured grid

2D example

3D example

3.5 Utilities

nuweb12 ⟨ Cumulative sum 12 ⟩ ≡

```
def cumsum(iterable):
    # cumulative addition: list(cumsum(range(4))) => [0, 1, 3, 6]
    iterable = iter(iterable)
    s = iterable.next()
    yield s
    for c in iterable:
        s = s + c
        yield s
```

nuweb9b9b.

Bibliography

Chapter 4

Hypercuboidal grids and topological products in LARCC

Here we develop an efficient implementation of multidimensional grid generation of cuboidal and simplicial cell complexes, and a fast implementation of the more general Cartesian product of cellular complexes. Both kind of operators, depending on the dimension of their input, may generate either full-dimensional (i.e. solid) output complexes or cellular complexes of dimension d embedded in Euclidean space of dimension n, with $d \leq n$.

4.1 Introduction

This report aims to discuss the design and the implementation of the `largrid` module of the LAR-CC library, including also the Cartesian product of general cellular complexes. In particular, we show that both n-dimensional grids of (hyper)-cuboidal cells and their d-dimensional skeletons ($0 \leq d \leq n$), embedded in $\mathbb{E}^n$, may be properly and efficiently generated by assembling the cells produced by a number n of either 0- or 1-dimensional cell complexes, that in such lowest dimensions coincide with simplicial complexes.

In Section 4.2 we give the simple implementation of generation of lower-dimensional (say, either 0- or 1-dimensional) regular cellular complexes with integer coordinates. In Section 4.3 a functional decomposition of the generation of either full-dimensional cuboidal complexes in $\mathbb{E}^n$ and of their d-skeletons ($0 \leq d \leq n$) is given, showing in particular that every skeleton can be efficiently generated as a partition in cell subsets produced by the Cartesian product of a proper disposition of 0-1 complexes, according to the binary representation of a subset of the integer interval $[0, 2^n]$. In Section 4.4 we provide a very simple and general implementation of the topological product of *two* cellular complexes of any topology. When applied to embedded linear cellular complexes (i.e. when the coordinates of 0-cells of arguments are fixed and given) the algorithm produces a Cartesian product of its two arguments. In Section 4.5 the exporting of the module to different languages is

provided. The Section 4.6 contains the unit tests associated to the various algorithms, that are exported by the used literate environment in the proper test subdirectory—depending on the implementation language. In Section 4.7 the indexing structure of the macro sources and variables is exposed by the sake of the reader. The Appendix 4.8 contains some programming utilities possibly needed by the developers.

4.2 0D- and 1D-complexes

We are going to use 0- and 1-dimensional cell complexes as the basic material for several operations, including generation of simplicial and cellular grids and topological and Cartesian product of cell complexes.

4.2.1 Generation of cells

Uniform 0D complex The `grid0` second-order function generates a 0-dimensional uniform complex embedding $n + 1$ equally-spaced (at unit intervals) 0-cells within the 1D interval. It returns the cells of this 0-complex.

nuweb3a ⟨ Generation of uniform 0D cellular complex 3a ⟩ ≡

```
def grid0(n):
    cells = AA(LIST)(range(n+1))
    return cells
```

nuweb14b14b.

Uniform 1D complex A similar `grid1` function returns a uniform 1D cellular complex with n 1D `cells`.

nuweb3b ⟨ Generation of uniform 1D cellular complex 3b ⟩ ≡

```
def grid1(n):
    ints = range(n+1)
    cells = TRANS([ints[:-1],ints[1:]])
    return cells
```

nuweb14b14b.

Uniform 0D or 1D complex A `larGrid` function is finally given to generate the LAR representation of the cells of either a 0- or a 1-dimensional complex, depending on the value of the `d` parameter, to take values in the set $\{0, 1\}$, and providing the *order* of the output complex.

nuweb3c ⟨ Generation of cellular complex of 0/1 dimension d 3c ⟩ ≡

```
def larGrid(n):
    def larGrid1(d):
        if d==0: return grid0(n)
        elif d==1: return grid1(n)
    return larGrid1
```

nuweb14b14b.

4.2.2 Generation of embedding vertices

Generation of grid vertices The second-order `larSplit` function is used to subdivide the real interval $[0, dom]$ into n equal parts. It returns the list of $n + 1$ `vertices` 1D of this decomposition, each represented as a singleton list.

nuweb4a ⟨ Generation of vertices of decompositions of 1D intervals 4a ⟩ ≡

```
def larSplit(dom):
    def larSplit1(n):
        assert n > 0 and type(n) == int
        item = float(dom)/n
        ints = range(n+1)
        items = [item]*(n+1)
        vertices = [[int*item] for (int,item) in zip(ints,items)]
        return vertices
    return larSplit1
```

nuweb14b14bnuweb15d, 15d.

4.3 Cuboidal grids

More interesting is the generation of *hyper-cubical grids* of intrinsic dimension d embedded in n-dimensional space, via the Cartesian product of d 1-complexes and $(n-d)$ 0-complexes. When $d = n$ the resulting grid is said *solid*; when $d = 0$ the output grid is 0-dimensional, and corresponds to a grid-arrangement of a discrete set of points in $\mathbb{E}^n$.

4.3.1 Full-dimensional grids

Vertex generation

First the grid vertices are produced by the `larVertProd` function, via Cartesian product of vertices of the n 1-dimensional arguments (vertex lists in `vertLists`), orderly corresponding to x_0, x_1, ..., x_{n-1} in the output points $(x_0, x_1, \ldots, x_{n-1})$.

nuweb4b ⟨ Generation of grid vertices 4b ⟩ ≡

```
def larVertProd(vertLists):
    return AA(CAT)(CART(vertLists))
```

nuweb14b14b.

Mapping of indices to storage

Multi-index to address transformation The second-order utility `index2addr` function transforms a `shape` list for a multidimensional array into a function that, when applied to a multindex array, i.e. to a list of integers within the `shape`'s bounds, returns the integer address of the array component within the linear storage of the multidimensional array.

The transformation formula for a d-dimensional array with `shape` $(n_0, n_1, ..., n_{d-1})$ is a linear combination of the 0-based[1] multi-index $(i_0, i_1, ..., i_{d-1})$ with `weights` equal to $(w_0, w_1, ..., w_{d-2}, 1)$:

$$addr = i_0 \times w_0 + i_1 \times w_1 + \cdots + i_{d-1} \times w_{d-1}$$

where

$$w_k = n_{k+1} \times n_{k+2} \times \cdots \times n_{d-1}, \qquad 0 \leq k \leq d-2.$$

Therefore, we get `index2addr([4,3,6])([2,2,0])` $= 48 = 2 \times (3 \times 6) + 2 \times (6 \times 1) + 0$, where `[2,2,0]` represent the numbers of (pages, rows, columns) indexing an element in the three-dimensional array of shape `[4,3,6]`.

nuweb5 ⟨ Transformation from multindex to address in a linear array storage 5 ⟩ ≡

```
def index2addr (shape):
    n = len(shape)
    shape = shape[1:]+[1]
    weights = [PROD(shape[k:]) for k in range(n)]
    def index2addr0 (multindex):
        return INNERPROD([multindex, weights])
    return index2addr0
```

nuweb14b14b.

`index2addr` examples In the following example, `[3,6]` is the `shape` of a two-dimensional array with 3 rows and 6 columns, stored in row-major order (i.e. by rows). The expression `index2addr([3,6])([2,0])` returns $12 = 2 \times (6 \times 1) + 0$, since the array element characterised by the multi-index value `[2,0]` is addressed at position 12 (starting from 0) in the linear storage of the array. Analogously, the function `index2addr([3,6])`, when applied to all the index values addressing the array of shape `[3,6]`, produces the integers between 0 and $17 = 3 \times 6 - 1$. In the last example, the function `index2addr([4,3,6])`

[1] 0-based array, like in C, java and python, as opposed to 1-based, like in fortran or matlab.

is applied to all the 0-based triples indexing a three-dimensional array of the given shape. Of course, the mapping works correctly even when the array shape is one-dimensional, as shown by the last example below.

nuweb6a ⟨ Test example 6a ⟩ ≡

```
>>> index2addr([3,6])([2,0])
12
>>> [index2addr([3,6])(index) for index in CART([ range(3), range(6) ])]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17]
>>> [index2addr([4,3,6])(index) for index in CART( AA(range)([4,3,6]) )]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39,
40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58,
59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71]
>>> index2addr([4])([2])
2
```

.

Multidimensional cell generation

In this section we discuss the implementation of the generation of cells as lists of indices to grid vertices. First, we study the case that the output complex is generated by the Cartesian product of *any* number of either 0- or 1-dimensional cell complexes. Then, we discuss an efficient extraction of d-dimensional skeleton of a (solid) n-dimensional grid, for $0 \leq d \leq n$.

Example In order to better understand the generation of cuboidal grids from products of 0- or 1-dimensional complexes, below we show a simple example of 2D grids embedded in $\mathbb{E}^3$. In particular, `v1 = [[0.],[1.],[2.],[3.]]` and `v0 = [[0.],[1.],[2.]]` are two arrays of 1D vertices, `c1 = [[0,1],[1,2],[2,3]]` and `c0 = [[0],[1],[2]]` are the LAR representation of a 1-complex and a 0-complex, respectively. The solid 2-complex named `grid2D` given below is shown in Figure 4.1a.

```
grid2D = larVertProd([v1,v1]),larCellProd([c1,c1])
VIEW(EXPLODE(1.2,1.2,1.2)(MKPOLS(grid2D)))
```

Notice that `grid2D`, generated by product of two 1-complexes, is *solid* in $\mathbb{E}^2$, whereas `grid3D` shown in Figure 4.1b, generated by product of two 1-complexes and one 0-complex, is two-dimensional and embedded in $\mathbb{E}^3$.

nuweb6b ⟨ Example of cuboidal grid of dimensions $(2, 3)$ 6b ⟩ ≡

```
v1, c1 = [[0.],[1.],[2.],[3.]],[[0,1],[1,2],[2,3]]
v0, c0 = [[0.],[1.],[2.]], [[0],[1],[2]]
```

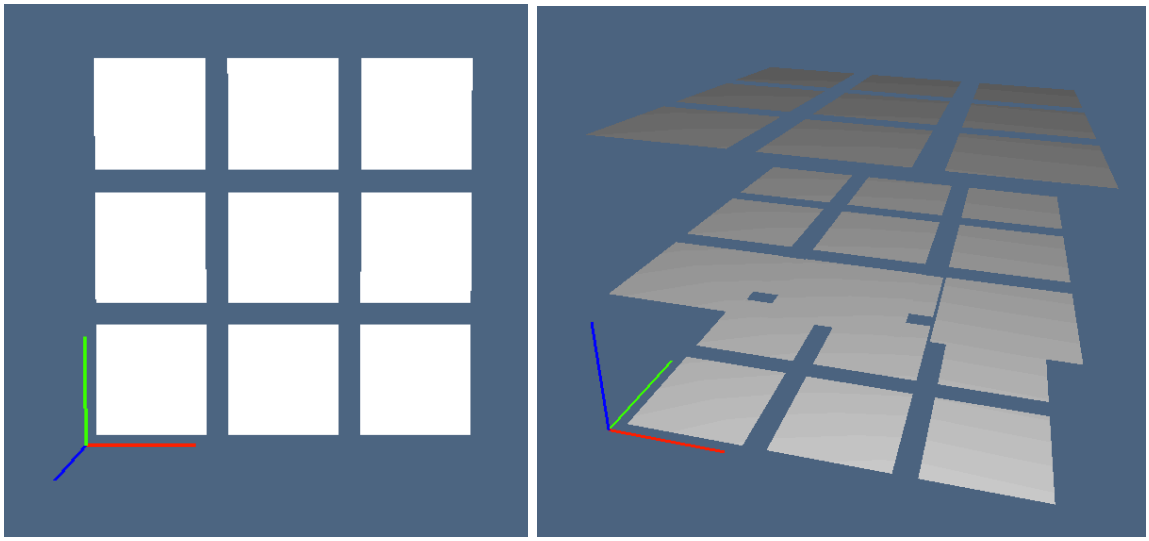

Figure 4.1: Exploded views of models `grid2D` and `grid3D`.

```
vertGrid = larVertProd([v1, v1, v0])
cellGrid = larCellProd([c1, c1, c0])
grid3D = vertGrid,cellGrid
VIEW(EXPLODE(1.2,1.2,1.2)(MKPOLS(grid3D)))
```

.

Cartesian product of 0/1-complexes Here, the input is given by the array `cellLists` of lists of cells of the argument complexes. Hence, the `shapes` variable contains the (list of) numbers $m_0, m_1, ...$ of cells in each argument complex, and the `indices` variable (generated by Cartesian product) collects the whole set $M_0 \times M_1 \times \cdots$ of 0-based multi-indices corresponding to the cells of the output complex, with $M_k = \{0, 1, ..., m_k - 1\}$.

The `jointCells` variable is used to contain the list of outputs of Cartesian products of `cells` corresponding to every `index` in `indices`.

nuweb7 ⟨ Generation of grid cells 7 ⟩ ≡

```
def larCellProd(cellLists):
    shapes = [len(item) for item in cellLists]
    indices = CART([range(shape) for shape in shapes])
    jointCells = [CART([cells[k] for k,cells in zip(index,cellLists)])
                  for index in indices]
    convert = index2addr([ shape+1 if (len(cellLists[k][0]) > 1) else shape
                           for k,shape in enumerate(shapes) ])
    return [AA(convert)(cell) for cell in jointCells]
```

nuweb14b14b.

With reference to the evaluation of the expression `larCellProd([c1,c1])`, where `c1` is the LAR representation of a 1-complex with 3 cells, defined by 4 vertices (0-cells), we have the trace given below. Of course, the function invocation returns the list of cells of the topological product of the input complexes, each one expressed as a list of vertices of the Cartesian product of the corresponding component vertices. The partially evaluated function `index2addr0`, stored in the `convert` variable, is used to execute the mapping, for each output `cell` in `jointCells`, from vertex multi-indices to their linear storage address. The mindful reader should notice that the number of generated cells is always equal to the product of terms in `shape`, in turn equal to the number of elements in `indices` and in `jointCells`. In this case we have $|\texttt{larCellProd([c1,c1])}| = 3 \times 3 = 9$.

nuweb8 ⟨ Tracing the evaluation of expression "`larCellProd([c1,c1])`" 8 ⟩ ≡

```
c1 = [[0,1], [1,2], [2,3]]
cellLists = [[[0,1], [1,2], [2,3]], [[0,1], [1,2], [2,3]]]
shapes = [3,3]
indices = [[0,0], [0,1], [0,2], [1,0], [1,1], [1,2], [2,0], [2,1], [2,2]]
jointCells = [
 [[0,0], [0,1], [1,0], [1,1]],
 [[0,1], [0,2], [1,1], [1,2]],
 [[0,2], [0,3], [1,2], [1,3]],
 [[1,0], [1,1], [2,0], [2,1]],
 [[1,1], [1,2], [2,1], [2,2]],
 [[1,2], [1,3], [2,2], [2,3]],
 [[2,0], [2,1], [3,0], [3,1]],
 [[2,1], [2,2], [3,1], [3,2]],
 [[2,2], [2,3], [3,2], [3,3]]]
convert = <function index2address0>
return [
 [0,1,4,5],
 [1,2,5,6],
 [2,3,6,7],
 [4,5,8,9],
 [5,6,9,10],
 [6,7,10,11],
 [8,9,12,13],
 [9,10,13,14],
 [10,11,14,15]]
```

.

4.3.2 Lower-dimensional grid skeletons

In order to compute the d-skeletons of a n-dimensional cuboidal "grid" complex, with $0 \leq d \leq n$, let us start by remarking a similarity with the generation of the boolean representation of numbers between 0 and $2^n - 1$, generated as a list of strings by the `binaryRange` function, given in Section 4.3.2.

The binary representations of such numbers are in fact filtered according to the number of their ones in Section 4.3.2, and used to generate the distinct components of different order skeletons of the assembled grid complexes in Section 4.3.2.

Generation of skeleton components

The `binaryRange` function, applied to an integer n, returns the string representation of all binary numerals between 0 and $2^n - 1$. All the strings have the same length n. The bits in each strings will be used to select between either a 0- or a 1-dimensional complex as generator (via a Cartesian product of complexes) of a component of an embedded grid skeleton of proper intrinsic dimension.

nuweb9a ⟨ Enumeration of binary ranges of given order 9a ⟩ ≡

```
def binaryRange(n):
    return [('{0:0'+str(n)+'b}').format(k) for k in range(2**n)]
```

nuweb14b14b.

Examples of generation of bit strings Below we show the outputs returned by application of the `binaryRange` function to the first 4 integers.

nuweb9b ⟨ Binary range examples 9b ⟩ ≡

```
>>> print binaryRange(4),
['0000', '0001', '0010', '0011', '0100', '0101', '0110', '0111',
 '1000', '1001', '1010', '1011', '1100', '1101', '1110', '1111']
>>> print binaryRange(3),
['000', '001', '010', '011', '100', '101', '110', '111']
>>> print binaryRange(2),
['00', '01', '10', '11']
>>> print binaryRange(1),
['0', '1']
```

.

Filtering grid skeleton components

The function `filterByOrder` is used to partition the previous binary strings into $n+1$ subsets, such that the bits into each string sum to the same number, ranging from 0 to n included, respectively.

nuweb9c ⟨ Filtering binary ranges by order 9c ⟩ ≡

```
def filterByOrder(n):
    terms = [AA(int)(list(term)) for term in binaryRange(n)]
    return [[term for term in terms if sum(term) == k] for k in range(n+1)]
```

nuweb14b14b.

Examples of bit lists filtering Some examples of application of the `filterByOrder` function to the first few integers are shown below. Of course, the number of elements in each class (i.e. in each returned list) is $\binom{n}{d}$, and the total number of elements for each fixed n is $\sum_{d=0}^{n}\binom{n}{d} = 2^n$.

nuweb10a ⟨ Skeleton component examples 10a ⟩ ≡

```
>>> filterByOrder(4)
[[[0,0,0,0]],
 [[0,0,0,1], [0,0,1,0], [0,1,0,0], [1,0,0,0]],
 [[0,0,1,1], [0,1,0,1], [0,1,1,0], [1,0,0,1], [1,0,1,0], [1,1,0,0]],
 [[0,1,1,1], [1,0,1,1], [1,1,0,1], [1,1,1,0]],
 [[1,1,1,1]]]
>>> filterByOrder(3)
[[[0,0,0]],
 [[0,0,1], [0,1,0], [1,0,0]],
 [[0,1,1], [1,0,1], [1,1,0]],
 [[1,1,1]]]
>>> filterByOrder(2)
[[[0,0]], [[0,1], [1,0]], [[1,1]]]
>>> filterByOrder(1)
[[[0]], [[1]]]
```

.

Assembling grid skeleton components

We are now finally able to generate the various subsets of cells of a d-dimensional cuboidal grid skeleton, produced respectively by the expression `larCellProd(cellLists)` for every permutation of 0- and 1-complexes, according to the partition classes of permtation of n bits previously produced. To understand why this assembling step of cells is necessary, the reader should look at Figure 4.2, where three subsets of 2-cells of the 2-skeleton, respectively generated by the bit dispositions `[[0,1,1], [1,0,1], [1,1,0]]`, are separately

displayed. Notice also that, whereas the dimension n of the embedding space is implicittly provided by the `length` of the `shape` parameter, the intrinsic dimension d of the skeleton to be produced must be given explicitly.

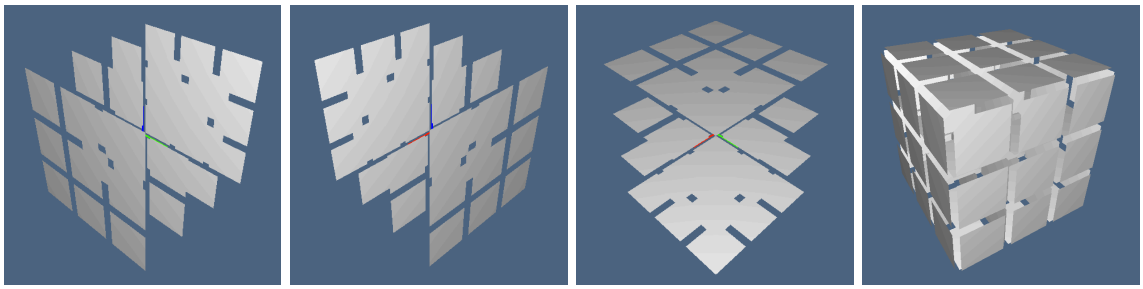

Figure 4.2: (a,b,c) Exploded views of subsets (orthogonal to coordinate axes) of 2-cells of a 2-skeleton grid; (d) their assembled set.

nuweb10b ⟨ Assembling grid skeletons 10b ⟩ ≡

```
def larGridSkeleton(shape):
    n = len(shape)
    def larGridSkeleton0(d):
        components = filterByOrder(n)[d]
        componentCellLists = [AA(APPLY)(zip( AA(larGrid)(shape),(component) ))
                              for component in components]
        return CAT([ larCellProd(cellLists)  for cellLists in componentCellLists ])
    return larGridSkeleton0
```

nuweb14b14b.

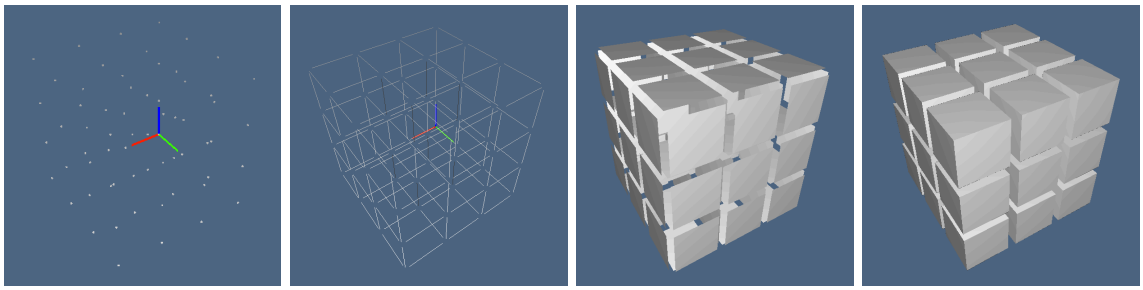

Figure 4.3: Exploded views of 0-, 1-, 2-, and 3-dimensional skeletons.

4.3.3 Highest-level grid interface

The highest-level user interface for (hyper)-cuboidal grid generation is given by the function `larCuboids` applied to the `shape` parameter. For the sake of storage efficiency, the generated vertex coordinates are integer and 0-based in the lowest corner. The model may be properly scaled and/or translated *a posteriori* when needed.

Generation of (hyper)-cuboidal grids The generated complex is always full-dimension, i.e. *solid*, and possibly includes the cells of all dimensions, depending on the Boolean value of the `full` parameter. The grid's intrinsic dimension, as well as the dimension of its embedding space, are specified by the length of the `shape` parameter. See the examples in Figure 4.4, but remember that the PLaSM visualiser always embed in 3D the displayed model.

nuweb12a ⟨ Multidimensional grid generation 12a ⟩ ≡

```
def larCuboids(shape, full=False):
   def vertexDomain(n):
      return [[k] for k in range(n)]
   vertLists = [vertexDomain(k+1) for k in shape]
   vertGrid = larVertProd(vertLists)
   gridMap = larGridSkeleton(shape)
   if not full:
      cells = gridMap(len(shape))
   else:
      skeletonIds = range(len(shape)+1)
      cells = CAT([ gridMap(id) for id in skeletonIds ])
   return vertGrid, cells
```

nuweb14b14b.

Multidimensional visualisation examples Visualisation examples of grid of dimension 1,2, and 3 are given below and are displayed in Figure 4.4. The same input pattern may be used for higher-dimensional grids (say, of dimension 4 and beyond), but to be visualised they should be carefully and properly projected in 3D.

nuweb12b ⟨ Multidimensional visualisation examples 12b ⟩ ≡

```
VIEW(EXPLODE(1.5,1.5,1.5)(MKPOLS(larCuboids([3],True))))
VIEW(EXPLODE(1.5,1.5,1.5)(MKPOLS(larCuboids([3,2],True))))
VIEW(EXPLODE(1.5,1.5,1.5)(MKPOLS(larCuboids([3,2,1],True))))
```

nuweb14b14b.

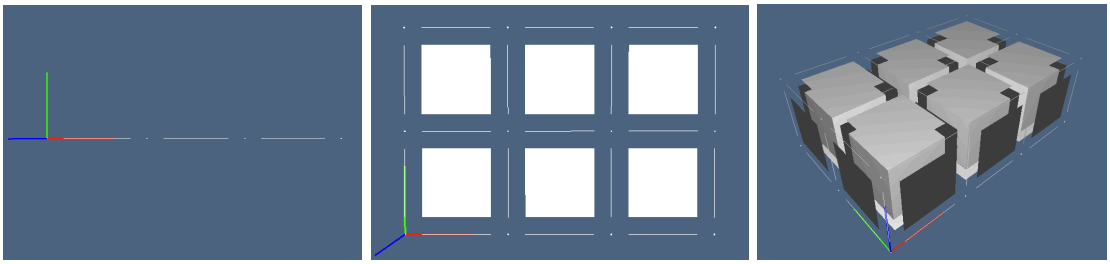

Figure 4.4: Exploded views of 1D, 2D, and 3D cellular complexes (including cells of dimension 0,1,2, and 3).

4.4 Cartesian product of cellular complexes

LAR model of cellular complexes The external representation of a LAR model (necessarily geometrical, i.e. embedded in some $\mathbb{E}^n$, in order to be possible to draw it) is a pair (*geometry*,*topology*), where *geometry* is the list of coordinates of vertices, i.e. a two-dimensional array of numbers, where vertices are given by row, and *topology* is a list of cells of fixed dimension d. When $d = n$ the model is *solid*; otherwise the model is some emberdded d-skeleton ($0 \leq d < n$).

Binary product of cellular complexes The `larModelProduct` function takes as input a pair of LAR models and returns the model of their Cartesian product. Since this is a pair (*geometry*, *topology*), its second element returns the topological product of the input topologies.

nuweb13a ⟨ Cartesian product of two lar models 13a ⟩ ≡

```
def larModelProduct(twoModels):
    (V, cells1), (W, cells2) = twoModels
    ⟨ Cartesian product of vertices nuweb13b13b ⟩
    ⟨ Topological product of cells nuweb13c13c ⟩
    model = [list(v) for v in vertices.keys()], cells
    return model
```

nuweb14b14b.

Cartesian product of argument vertices The following macro is used to generate a dictionary mapping between integer ids of new vertices and the sets V and W of vertices of the input complexes.

nuweb13b ⟨ Cartesian product of vertices 13b ⟩ ≡

```
vertices = collections.OrderedDict(); k = 0
for v in V:
    for w in W:
        id = tuple(v+w)
        if not vertices.has_key(id):
            vertices[id] = k
            k += 1
```

nuweb13a13a.

Topological product of argument vertices Another macro generates the cells of the topological product, represented as lists of new vertices.

nuweb13c ⟨ Topological product of cells 13c ⟩ ≡

```
cells = [ [vertices[tuple(V[v] + W[w])] for v in c1 for w in c2]
         for c1 in cells1 for c2 in cells2]
```

nuweb13a13a.

nuweb14a ⟨ Test examples of Cartesian product 14a ⟩ ≡

```
if __name__ == "__main__":
    geom_0,topol_0 = [[0.],[1.],[2.],[3.],[4.]],[[0,1],[1,2],[2,3],[3,4]]
    geom_1,topol_1 = [[0.],[1.],[2.]], [[0,1],[1,2]]
    mod_0 = (geom_0,topol_0)
    mod_1 = (geom_1,topol_1)
    squares = larModelProduct([mod_0,mod_1])
    VIEW(EXPLODE(1.2,1.2,1.2)(MKPOLS(squares)))
    cubes = larModelProduct([squares,mod_0])
    VIEW(EXPLODE(1.2,1.2,1.2)(MKPOLS(cubes)))
```

nuweb14b14b.

4.5 Largrid exporting

In this section we assemble top-down the `largrid` module, by orderly listing the macros it is composed of. As might be expected, the present one is the module version corresponding to the current state of the system, i.e. to a very initial state. Other functions will be added when needed, and the module translation in different languages (C/C++, Javascript, Haskell, OpenCL kernels) will be (hopefully soon) appended.

nuweb14b "lib/py/largrid.py" 14b ≡

```
"""Module with functions for grid generation and Cartesian product"""
import collections
```

⟨ Importing `simplexn` and `numpy` libraries nuweb1616 ⟩

⟨ Generation of vertices of decompositions of 1D intervals nuweb4a4a ⟩

⟨ Generation of uniform 0D cellular complex nuweb3a3a ⟩
⟨ Generation of uniform 1D cellular complex nuweb3b3b ⟩
⟨ Generation of cellular complex of 0/1 dimension d nuweb3c3c ⟩
⟨ Generation of grid vertices nuweb4b4b ⟩
⟨ Transformation from multindex to address in a linear array storage nuweb55 ⟩
⟨ Generation of grid cells nuweb77 ⟩
⟨ Enumeration of binary ranges of given order nuweb9a9a ⟩
⟨ Filtering binary ranges by order nuweb9c9c ⟩
⟨ Assembling grid skeletons nuweb10b10b ⟩
⟨ Multidimensional grid generation nuweb12a12a ⟩
⟨ Cartesian product of two lar models nuweb13a13a ⟩

```
if __name__=="__main__":
```

⟨ Multidimensional visualisation examples nuweb12b12b ⟩
⟨ Test examples of Cartesian product nuweb14a14a ⟩

4.6 Unit tests

4.6.1 Creation of repository of unit tests

A possible unit test strategy is to create a directory for unit tests associated to each source file in `nuweb`. Therefore we create here a directory in `test/py/` with the same name of the present document. Of course other

nuweb15a ⟨ Create directory and echo of creation 15a ⟩ ≡

⟨ Create directory from path nuweb17a17a ⟩

```
createDir('@1')
print "'@1' repository created"
```

.

nuweb15b `"test/py/largrid/test01.py"` 15b ≡

⟨ Create directory and echo of creation: (nuweb15c15c `test/py/largrid/`) ? ⟩

nuweb15b15bnuweb15dd.

Vertices of 1D decompositions Some test examples of the `larSplit` function are given in the following. First the unit interval $[0, 1]$ is splitter into 10 sub intervals, then the $[0, 2\pi]$ interval is split into 12 parts, used to generate a polyonal approximatetion of the unit circle S_1, centred in the origin and with unit radius.

nuweb15d `"test/py/largrid/test01.py"` 15d ≡

```
from pyplasm import *
```

⟨ Generation of vertices of decompositions of 1D intervals nuweb4a4a ⟩
```
assert larSplit(1)(3) == [[0.0], [0.3333333333333333], [0.6666666666666666], [1.0]]
assert larSplit(1)(1) == [[0.0], [1.0]]
assert larSplit(2*PI)(12) == [[0.0], [0.5235987755982988], [1.0471975511965976],
[1.5707963267948966], [2.0943951023931953], [2.617993877991494],
[3.141592653589793], [3.665191429188092], [4.1887902047863905],
[4.71238898038469], [5.235987755982988], [5.759586531581287],
[6.283185307179586]]
```

nuweb15b15bnuweb15dd.

nuweb15e `"test/py/largrid/test02.py"` 15e ≡
```
from largrid import *

mod_1 = larSplit(1)(4), larGrid(4)(1)
squares = larModelProduct([mod_1,mod_1])
VIEW(EXPLODE(1.2,1.2,1.2)(MKPOLS(squares)))
cubes = larModelProduct([squares,mod_1])
VIEW(EXPLODE(1.2,1.2,1.2)(MKPOLS(cubes)))
```

4.7 Indices

The list of macros follow.

⟨ Test example nuweb6a6a ⟩ .
⟨ Topological product of cells nuweb13c13c ⟩ nuweb13a13a.
⟨ Tracing the evaluation of expression "`larCellProd([c1,c1])`" nuweb88 ⟩ .
⟨ Transformation from multindex to address in a linear array storage nuweb55 ⟩ nuweb14b14b.

4.8 Appendix

4.8.1 Utilities

nuweb16 ⟨ Importing `simplexn` and `numpy` libraries 16 ⟩ ≡

```
from simplexn import *
import numpy as np
```

nuweb14b14b.

An useful utility will allow for the creation of a subdirectory from a `dirpath` *string*.

nuweb17a ⟨ Create directory from path 17a ⟩ ≡

```
import os
def createDir(dirpath):
    if not os.path.exists(dirpath):
        os.makedirs(dirpath)
```

nuweb15a15anuweb17b, 17b.

It may be useful to define the repository(ies) for the unit tests associated to the module:

nuweb17b `"test/py/largrid-tests.py"` 17b ≡

```
⟨ Create directory from path nuweb17a17a ⟩
createDir('test/py/largrid/')
```

Bibliography

Chapter 5

The basic `larcc` module

5.1 Basic representations

A few basic representation of topology are used in LARCC. They include some common sparse matrix representations: CSR (Compressed Sparse Row), CSC (Compressed Sparse Column), COO (Coordinate Representation), and BRC (Binary Row Compressed).

5.1.1 BRC (Binary Row Compressed)

We denote as BRC (Binary Row Compressed) the standard input representation of our LARCC framework. A BRC representation is an array of arrays of integers, with no requirement of equal length for the component arrays. The BRC format is used to represent a (normally sparse) binary matrix. Each component array corresponds to a matrix row, and contains the indices of columns that store a 1 value. No storage is used for 0 values.

BRC format example Let $A = (a_{i,j} \in \{0,1\})$ be a binary matrix. The notation $\texttt{BRC}(A)$ is used for the corresponding data structure.

$$A = \begin{pmatrix} 0,1,0,0,0,0,0,1,0,0 \\ 0,0,1,0,0,0,0,0,0,0 \\ 1,0,0,1,0,0,0,0,0,1 \\ 1,0,0,0,0,0,1,0,0,0 \\ 0,0,0,0,0,1,1,1,0,0 \\ 0,0,1,0,1,0,0,0,1,0 \\ 0,0,0,0,0,0,0,0,0,0 \\ 0,1,0,0,0,0,0,1,0,1 \\ 0,0,0,1,0,0,0,0,1,0 \\ 0,1,1,0,1,0,0,0,0,0 \end{pmatrix} \quad \mapsto \quad \texttt{BRC}(A) = \begin{array}{l} \texttt{[[1,7],} \\ \texttt{[2],} \\ \texttt{[0,3,9],} \\ \texttt{[0,6],} \\ \texttt{[5,6,7],} \\ \texttt{[2,4,8],} \\ \texttt{[],} \\ \texttt{[1,7,9],} \\ \texttt{[3,8],} \\ \texttt{[1,2,4]]} \end{array}$$

5.1.2 Format conversions

First we give the function `format` to make the transformation from the sparse matrix as a list of triples *(row,column,value)* for each non-zero element, to the `scipy.sparse` format corresponding to the `shape` parameter, set by default to `"csr"`, that stands for *Compressed Sparse Row*, the normal matrix format of the LARCC framework.

nuweb3a ⟨ From list of triples to scipy.sparse 3a ⟩ ≡

```
def format(triples,shape="csr"):
    n = len(triples)
    data = arange(n)
    ij = arange(2*n).reshape(2,n)
    for k,item in enumerate(triples):
        ij[0][k],ij[1][k],data[k] = item
    return scipy.sparse.coo_matrix((data, ij)).asformat(shape)
```

nuweb16a16a.

nuweb3b ⟨ Brc to Coo transformation 3b ⟩ ≡

```
def cooCreateFromBrc(ListOfListOfInt):
    COOm = [[k,col,1] for k,row in enumerate(ListOfListOfInt)
            for col in row ]
    return COOm
```

nuweb16a16a.

nuweb3c ⟨ Test example of Brc to Coo transformation 3c ⟩ ≡

```
print "\n>>> cooCreateFromBrc"
V = [[0, 0], [1, 0], [2, 0], [0, 1], [1, 1], [2, 1]]
FV = [[0, 1, 3], [1, 2, 4], [1, 3, 4], [2, 4, 5]]
EV = [[0,1],[0,3],[1,2],[1,3],[1,4],[2,4],[2,5],[3,4],[4,5]]
cooFV = cooCreateFromBrc(FV)
cooEV = cooCreateFromBrc(EV)
print "\ncooCreateFromBrc(FV) =\n", cooFV
print "\ncooCreateFromBrc(EV) =\n", cooEV
```

nuweb16b16b.

nuweb3d ⟨ Coo to Csr transformation 3d ⟩ ≡

```
def csrCreateFromCoo(COOm):
    CSRm = format(COOm,"csr")
    return CSRm
```

nuweb16a16a.

nuweb4a ⟨ Test example of Coo to Csr transformation 4a ⟩ ≡

```
print "\n>>> csrCreateFromCoo"
csrFV = csrCreateFromCoo(cooFV)
csrEV = csrCreateFromCoo(cooEV)
print "\ncsr(FV) =\n", repr(csrFV)
print "\ncsr(EV) =\n", repr(csrEV)
```

nuweb16b16b.

nuweb4b ⟨ Brc to Csr transformation 4b ⟩ ≡

```
def csrCreate(BRCm,shape=(0,0)):
    if shape == (0,0):
        out = csrCreateFromCoo(cooCreateFromBrc(BRCm))
        return out
    else:
        CSRm = scipy.sparse.csr_matrix(shape)
        for i,j,v in cooCreateFromBrc(BRCm):
            CSRm[i,j] = v
        return CSRm
```

nuweb16a16a.

nuweb4c ⟨ Test example of Brc to Csr transformation 4c ⟩ ≡

```
print "\n>>> csrCreateFromCoo"
V = [[0, 0], [1, 0], [2, 0], [0, 1], [1, 1], [2, 1]]
FV = [[0, 1, 3], [1, 2, 4], [1, 3, 4], [2, 4, 5]]
csrFV = csrCreate(FV)
print "\ncsrCreate(FV) =\n", csrFV
```

nuweb16b16b.

5.2 Matrix operations

nuweb4d ⟨ Query Matrix shape 4d ⟩ ≡

```
def csrGetNumberOfRows(CSRm):
    Int = CSRm.shape[0]
    return Int

def csrGetNumberOfColumns(CSRm):
    Int = CSRm.shape[1]
    return Int
```

nuweb16a16a.

nuweb5a ⟨ Test examples of Query Matrix shape 5a ⟩ ≡

```
print "\n>>> csrGetNumberOfRows"
print "\ncsrGetNumberOfRows(csrFV) =", csrGetNumberOfRows(csrFV)
print "\ncsrGetNumberOfRows(csrEV) =", csrGetNumberOfRows(csrEV)
print "\n>>> csrGetNumberOfColumns"
print "\ncsrGetNumberOfColumns(csrFV) =", csrGetNumberOfColumns(csrFV)
print "\ncsrGetNumberOfColumns(csrEV) =", csrGetNumberOfColumns(csrEV)
```

nuweb16b16b.

nuweb5b ⟨ Sparse to dense matrix transformation 5b ⟩ ≡

```
def csrToMatrixRepresentation(CSRm):
    nrows = csrGetNumberOfRows(CSRm)
    ncolumns = csrGetNumberOfColumns(CSRm)
    ScipyMat = zeros((nrows,ncolumns),int)
    C = CSRm.tocoo()
    for triple in zip(C.row,C.col,C.data):
        ScipyMat[triple[0],triple[1]] = triple[2]
    return ScipyMat
```

nuweb16a16a.

nuweb5c ⟨ Test examples of Sparse to dense matrix transformation 5c ⟩ ≡

```
print "\n>>> csrToMatrixRepresentation"
print "\nFV =\n", csrToMatrixRepresentation(csrFV)
print "\nEV =\n", csrToMatrixRepresentation(csrEV)
```

nuweb16b16b.

nuweb5d ⟨ Matrix product and transposition 5d ⟩ ≡

```
def matrixProduct(CSRm1,CSRm2):
    CSRm = CSRm1 * CSRm2
    return CSRm

def csrTranspose(CSRm):
    CSRm = CSRm.T
    return CSRm
```

nuweb16a16a.

nuweb6a ⟨ Matrix filtering to produce the boundary matrix 6a ⟩ ≡

```
def csrBoundaryFilter(CSRm, facetLengths):
    maxs = [max(CSRm[k].data) for k in range(CSRm.shape[0])]
    inputShape = CSRm.shape
    coo = CSRm.tocoo()
    for k in range(len(coo.data)):
        if coo.data[k]==maxs[coo.row[k]]: coo.data[k] = 1
        else: coo.data[k] = 0
    mtx = coo_matrix((coo.data, (coo.row, coo.col)), shape=inputShape)
    out = mtx.tocsr()
    return out
```

nuweb16a16a.

nuweb6b ⟨ Test example of Matrix filtering to produce the boundary matrix 6b ⟩ ≡

```
print "\n>>> csrBoundaryFilter"
csrEF = matrixProduct(csrFV, csrTranspose(csrEV)).T
facetLengths = [csrCell.getnnz() for csrCell in csrEV]
CSRm = csrBoundaryFilter(csrEF, facetLengths).T
print "\ncsrMaxFilter(csrFE) =\n", csrToMatrixRepresentation(CSRm)
```

nuweb16b16b.

nuweb6c ⟨ Matrix filtering via a generic predicate 6c ⟩ ≡

```
def csrPredFilter(CSRm, pred):
   # can be done in parallel (by rows)
   coo = CSRm.tocoo()
   triples = [[row,col,val] for row,col,val
            in zip(coo.row,coo.col,coo.data) if pred(val)]
   i, j, data = TRANS(triples)
   CSRm = scipy.sparse.coo_matrix((data,(i,j)),CSRm.shape).tocsr()
   return CSRm
```

nuweb16a16a.

nuweb6d ⟨ Test example of Matrix filtering via a generic predicate 6d ⟩ ≡

```
print "\n>>> csrPredFilter"
CSRm = csrPredFilter(matrixProduct(csrFV, csrTranspose(csrEV)).T, GE(2)).T
print "\nccsrPredFilter(csrFE) =\n", csrToMatrixRepresentation(CSRm)
```

nuweb16b16b.

5.3 Topological operations

nuweb7a ⟨ From cells and facets to boundary operator 7a ⟩ ≡

```
def boundary(cells,facets):
    csrCV = csrCreate(cells)
    csrFV = csrCreate(facets)
    csrFC = matrixProduct(csrFV, csrTranspose(csrCV))
    facetLengths = [csrCell.getnnz() for csrCell in csrCV]
    return csrBoundaryFilter(csrFC,facetLengths)

def coboundary(cells,facets):
    Boundary = boundary(cells,facets)
    return csrTranspose(Boundary)
```

nuweb16a16a.

nuweb7b ⟨ Test examples of From cells and facets to boundary operator 7b ⟩ ≡

```
V = [[0.0, 0.0, 0.0], [1.0, 0.0, 0.0], [0.0, 1.0, 0.0], [1.0, 1.0, 0.0],
[0.0, 0.0, 1.0], [1.0, 0.0, 1.0], [0.0, 1.0, 1.0], [1.0, 1.0, 1.0]]

CV =[[0, 1, 2, 4], [1, 2, 4, 5], [2, 4, 5, 6], [1, 2, 3, 5], [2, 3, 5, 6],
[3, 5, 6, 7]]

FV =[[0, 1, 2], [0, 1, 4], [0, 2, 4], [1, 2, 3], [1, 2, 4], [1, 2, 5],
[1, 3, 5], [1, 4, 5], [2, 3, 5], [2, 3, 6], [2, 4, 5], [2, 4, 6], [2, 5, 6],
[3, 5, 6], [3, 5, 7], [3, 6, 7], [4, 5, 6], [5, 6, 7]]

EV =[[0, 1], [0, 2], [0, 4], [1, 2], [1, 3], [1, 4], [1, 5], [2, 3], [2, 4],
[2, 5], [2, 6], [3, 5], [3, 6], [3, 7], [4, 5], [4, 6], [5, 6], [5, 7],
[6, 7]]

print "\ncoboundary_2 =\n", csrToMatrixRepresentation(coboundary(CV,FV))
print "\ncoboundary_1 =\n", csrToMatrixRepresentation(coboundary(FV,EV))
print "\ncoboundary_0 =\n", csrToMatrixRepresentation(coboundary(EV,AA(LIST)(range(len(V))))
```

nuweb16b16b.

nuweb8a ⟨ From cells and facets to boundary cells 8a ⟩ ≡

```
def zeroChain(cells):
   pass

def totalChain(cells):
   return csrCreate([[0] for cell in cells])

def boundaryCells(cells,facets):
   csrBoundaryMat = boundary(cells,facets)
   csrChain = totalChain(cells)
   csrBoundaryChain = matrixProduct(csrBoundaryMat, csrChain)
   for k,value in enumerate(csrBoundaryChain.data):
      if value % 2 == 0: csrBoundaryChain.data[k] = 0
   boundaryCells = [k for k,val in enumerate(csrBoundaryChain.data.tolist()) if val == 1]
   return boundaryCells
```

nuweb16a16a.

nuweb8b ⟨ Test examples of From cells and facets to boundary cells 8b ⟩ ≡

```
boundaryCells_2 = boundaryCells(CV,FV)
boundaryCells_1 = boundaryCells([FV[k] for k in boundaryCells_2],EV)

print "\nboundaryCells_2 =\n", boundaryCells_2
print "\nboundaryCells_1 =\n", boundaryCells_1

boundary = (V,[FV[k] for k in boundaryCells_2])
VIEW(EXPLODE(1.5,1.5,1.5)(MKPOLS(boundary)))
```

nuweb16b16b.

nuweb9a ⟨ Signed boundary matrix for simplicial models 9a ⟩ ≡

```
def signedBoundary (V,CV,FV):
   # compute the set of pairs of indices to [boundary face,incident coface]
   coo = boundary(CV,FV).tocoo()
   pairs = [[coo.row[k],coo.col[k]] for k,val in enumerate(coo.data) if val != 0]

   # compute the [face, coface] pair as vertex lists
   vertLists = [[FV[pair[0]], CV[pair[1]]]for pair in pairs]

   # compute two n-cells to compare for sign
   cellPairs = [ [list(set(coface).difference(face))+face,coface]
               for face,coface in vertLists]

   # compute the local indices of missing boundary cofaces
   missingVertIndices = [ coface.index(list(set(coface).difference(face))[0])
                     for face,coface in vertLists]

   # compute the point matrices to compare for sign
   pointArrays = [ [[V[k]+[1.0] for k in facetCell], [V[k]+[1.0] for k in cofaceCell]]
               for facetCell,cofaceCell in cellPairs]

   # signed incidence coefficients
   cofaceMats = TRANS(pointArrays)[1]
   cofaceSigns = AA(SIGN)(AA(np.linalg.det)(cofaceMats))
   faceSigns = AA(C(POWER)(-1))(missingVertIndices)
   signPairProd = AA(PROD)(TRANS([cofaceSigns,faceSigns]))

   # signed boundary matrix
   csrSignedBoundaryMat = csr_matrix( (signPairProd,TRANS(pairs)) )
   return csrSignedBoundaryMat
```

nuweb16a16a.

nuweb9b ⟨ Oriented boundary cells for simplicial models 9b ⟩ ≡

```
def signedBoundaryCells(verts,cells,facets):
   csrBoundaryMat = signedBoundary(verts,cells,facets)
   csrTotalChain = totalChain(cells)
   csrBoundaryChain = matrixProduct(csrBoundaryMat, csrTotalChain)
   coo = csrBoundaryChain.tocoo()
   boundaryCells = list(coo.row * coo.data)
   return AA(int)(boundaryCells)
```

nuweb9b9bnuweb11, 11.
nuweb16a16a.

Orienting polytopal cells

input : "cell" indices of a convex and solid polytopes and "V" vertices;

output : biggest "simplex" indices spanning the polytope.

`m` : number of cell vertices

`d` : dimension (number of coordinates) of cell vertices

`d+1` : number of simplex vertices

`vcell` : cell vertices

`vsimplex` : simplex vertices

`Id` : identity matrix

`basis` : orthonormal spanning set of vectors e_k

`vector` : position vector of a simplex vertex in translated coordinates

`unUsedIndices` : cell indices not moved to simplex

nuweb11 ⟨ Oriented boundary cells for simplicial models 11 ⟩ ≡

```
def pivotSimplices(V,CV,d=3):
   simplices = []
   for cell in CV:
      vcell = np.array([V[v] for v in cell])
      m, simplex = len(cell), []
      # translate the cell: for each k, vcell[k] -= vcell[0], and simplex[0] := cell[0]
      for k in range(m-1,-1,-1): vcell[k] -= vcell[0]
      # simplex = [0], basis = [], tensor = Id(d+1)
      simplex += [cell[0]]
      basis = []
      tensor = np.array(IDNT(d))
      # look for most far cell vertex
      dists = [SUM([SQR(x) for x in v])**0.5 for v in vcell]
      maxDistIndex = max(enumerate(dists),key=lambda x: x[1])[0]
      vector = np.array([vcell[maxDistIndex]])
      # normalize vector
      den=(vector**2).sum(axis=-1) **0.5
      basis = [vector/den]
      simplex += [cell[maxDistIndex]]
      unUsedIndices = [h for h in cell if h not in simplex]

      # for k in {2,d+1}:
      for k in range(2,d+1):
         # update the orthonormal tensor
         e = basis[-1]
         tensor = tensor - np.dot(e.T, e)
         # compute the index h of a best vector
         # look for most far cell vertex
         dists = [SUM([SQR(x) for x in np.dot(tensor,v)])**0.5
         if h in unUsedIndices else 0.0
         for (h,v) in zip(cell,vcell)]
         # insert the best vector index h in output simplex
         maxDistIndex = max(enumerate(dists),key=lambda x: x[1])[0]
         vector = np.array([vcell[maxDistIndex]])
         # normalize vector
         den=(vector**2).sum(axis=-1) **0.5
         basis += [vector/den]
         simplex += [cell[maxDistIndex]]
         unUsedIndices = [h for h in cell if h not in simplex]
      simplices += [simplex]
   return simplices

def simplexOrientations(V,simplices):
   vcells = [[V[v]+[1.0] for v in simplex] for simplex in simplices]
   return [SIGN(np.linalg.det(vcell)) for vcell in vcells]
```

nuweb9b9bnuweb11, 11.
nuweb16a16a.

nuweb12a ⟨ Computation of cell adjacencies 12a ⟩ ≡

```
def larCellAdjacencies(CSRm):
    CSRm = matrixProduct(CSRm,csrTranspose(CSRm))
    return CSRm
```

nuweb16a16a.

nuweb12b ⟨ Test examples of Computation of cell adjacencies 12b ⟩ ≡

```
print "\n>>> larCellAdjacencies"
adj_2_cells = larCellAdjacencies(csrFV)
print "\nadj_2_cells =\n", csrToMatrixRepresentation(adj_2_cells)
adj_1_cells = larCellAdjacencies(csrEV)
print "\nadj_1_cells =\n", csrToMatrixRepresentation(adj_1_cells)
```

nuweb16b16b.

nuweb13 ⟨ Extraction of facets of a cell complex 13 ⟩ ≡

```
def setup(model,dim):
    V, cells = model
    csr = csrCreate(cells)
    csrAdjSquareMat = larCellAdjacencies(csr)
    csrAdjSquareMat = csrPredFilter(csrAdjSquareMat, GE(dim)) # ? HOWTODO ?
    return V,cells,csr,csrAdjSquareMat

def larFacets(model,dim=3):
    """
        Estraction of (d-1)-cellFacets from "model" := (V,d-cells)
        Return (V, (d-1)-cellFacets)
      """
    V,cells,csr,csrAdjSquareMat = setup(model,dim)
    cellFacets = []
    # for each input cell i
    for i in range(len(cells)):
        adjCells = csrAdjSquareMat[i].tocoo()
        cell1 = csr[i].tocoo().col
        pairs = zip(adjCells.col,adjCells.data)
        for j,v in pairs:
            if (i<j):
                cell2 = csr[j].tocoo().col
                cell = list(set(cell1).intersection(cell2))
                cellFacets.append(sorted(cell))
    # sort and remove duplicates
    cellFacets = sorted(AA(list)(set(AA(tuple)(cellFacets))))
    return V,cellFacets
```

nuweb16a16a.

nuweb14 ⟨ Test examples of Extraction of facets of a cell complex 14 ⟩ ≡

```
V = [[0.,0.],[3.,0.],[0.,3.],[3.,3.],[1.,2.],[2.,2.],[1.,1.],[2.,1.]]
FV = [[0,1,6,7],[0,2,4,6],[4,5,6,7],[1,3,5,7],[2,3,4,5],[0,1,2,3]]

_,EV = larFacets((V,FV),dim=2)
print "\nEV =",EV
VIEW(EXPLODE(1.5,1.5,1.5)(MKPOLS((V,EV))))

FV = [[0,1,3],[1,2,4],[2,4,5],[3,4,6],[4,6,7],[5,7,8], # full
   [1,3,4],[4,5,7], # empty
   [0,1,2],[6,7,8],[0,3,6],[2,5,8]] # exterior

_,EV = larFacets((V,FV),dim=2)
print "\nEV =",EV
```

nuweb16b16b.

5.4 Exporting the library

5.4.1 MIT licence

nuweb15a ⟨ The MIT Licence 15a ⟩ ≡

```
"""
The MIT License
===============

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
'Software'), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED 'AS IS', WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
"""
```

nuweb16a16a.

5.4.2 Importing of modules or packages

nuweb15b ⟨ Importing of modules or packages 15b ⟩ ≡

```
from pyplasm import *
import collections
import scipy
import numpy as np
from scipy import zeros,arange,mat,amin,amax
from scipy.sparse import vstack,hstack,csr_matrix,coo_matrix,lil_matrix,triu

from lar2psm import *
```

nuweb16a16a.

5.4.3 Writing the library file

nuweb16a `"lib/py/larcc.py"` 16a ≡

```
# -*- coding: utf-8 -*-
""" Basic LARCC library """
```
⟨ The MIT Licence nuweb15a15a ⟩
⟨ Importing of modules or packages nuweb15b15b ⟩
⟨ From list of triples to scipy.sparse nuweb3a3a ⟩
⟨ Brc to Coo transformation nuweb3b3b ⟩
⟨ Coo to Csr transformation nuweb3d3d ⟩
⟨ Brc to Csr transformation nuweb4b4b ⟩
⟨ Query Matrix shape nuweb4d4d ⟩
⟨ Sparse to dense matrix transformation nuweb5b5b ⟩
⟨ Matrix product and transposition nuweb5d5d ⟩
⟨ Matrix filtering to produce the boundary matrix nuweb6a6a ⟩
⟨ Matrix filtering via a generic predicate nuweb6c6c ⟩
⟨ From cells and facets to boundary operator nuweb7a7a ⟩
⟨ From cells and facets to boundary cells nuweb8a8a ⟩
⟨ Signed boundary matrix for simplicial models nuweb9a9a ⟩
⟨ Oriented boundary cells for simplicial models nuweb9b9b, ... ⟩
⟨ Computation of cell adjacencies nuweb12a12a ⟩
⟨ Extraction of facets of a cell complex nuweb1313 ⟩

```
if __name__ == "__main__":
```
⟨ Test examples nuweb16b16b ⟩

5.5 Unit tests

nuweb16b ⟨ Test examples 16b ⟩ ≡

⟨ Test example of Brc to Coo transformation nuweb3c3c ⟩
⟨ Test example of Coo to Csr transformation nuweb4a4a ⟩
⟨ Test example of Brc to Csr transformation nuweb4c4c ⟩
⟨ Test examples of Query Matrix shape nuweb5a5a ⟩
⟨ Test examples of Sparse to dense matrix transformation nuweb5c5c ⟩
⟨ Test example of Matrix filtering to produce the boundary matrix nuweb6b6b ⟩
⟨ Test example of Matrix filtering via a generic predicate nuweb6d6d ⟩
⟨ Test examples of From cells and facets to boundary operator nuweb7b7b ⟩
⟨ Test examples of From cells and facets to boundary cells nuweb8b8b ⟩
⟨ Test examples of Computation of cell adjacencies nuweb12b12b ⟩
⟨ Test examples of Extraction of facets of a cell complex nuweb1414 ⟩

nuweb16a16a.

5.6 Appendix: Tutorials

5.6.1 Model generation, skeleton and boundary extraction

nuweb17a "test/py/larcc/ex1.py" 17a ≡

```
from larcc import *
from largrid import *
⟨ input of 2D topology and geometry data nuweb17b17b ⟩
⟨ characteristic matrices nuweb17c17c ⟩
⟨ incidence matrix nuweb17d17d ⟩
⟨ boundary and coboundary operators nuweb18a18a ⟩
⟨ product of cell complexes nuweb18b18b ⟩
⟨ 2-skeleton extraction nuweb18c18c ⟩
⟨ 1-skeleton extraction nuweb19a19a ⟩
⟨ 0-coboundary computation nuweb19b19b ⟩
⟨ 1-coboundary computation nuweb19c19c ⟩
⟨ 2-coboundary computation nuweb20a20a ⟩
⟨ boundary chain visualisation nuweb20b20b ⟩
```

nuweb17b ⟨ input of 2D topology and geometry data 17b ⟩ ≡

```
# input of topology and geometry
V2 = [[4,10],[8,10],[14,10],[8,7],[14,7],[4,4],[8,4],[14,4]]
EV = [[0,1],[1,2],[3,4],[5,6],[6,7],[0,5],[1,3],[2,4],[3,6],[4,7]]
FV = [[0,1,3,5,6],[1,2,3,4],[3,4,6,7]]
```

nuweb17a17a.

nuweb17c ⟨ characteristic matrices 17c ⟩ ≡

```
# characteristic matrices
csrFV = csrCreate(FV)
csrEV = csrCreate(EV)
print "\nFV =\n", csrToMatrixRepresentation(csrFV)
print "\nEV =\n", csrToMatrixRepresentation(csrEV)
```

nuweb17a17a.

nuweb17d ⟨ incidence matrix 17d ⟩ ≡

```
# product
csrEF = matrixProduct(csrEV, csrTranspose(csrFV))
print "\nEF =\n", csrToMatrixRepresentation(csrEF)
```

nuweb17a17a.

nuweb18a ⟨ boundary and coboundary operators 18a ⟩ ≡

```
# boundary and coboundary operators
facetLengths = [csrCell.getnnz() for csrCell in csrEV]
boundary = csrBoundaryFilter(csrEF,facetLengths)
coboundary_1 = csrTranspose(boundary)
print "\ncoboundary_1 =\n", csrToMatrixRepresentation(coboundary_1)
```

nuweb17a17a.

nuweb18b ⟨ product of cell complexes 18b ⟩ ≡

```
# product operator
mod_2D = (V2,FV)
V1,topol_0 = [[0.],[1.],[2.]], [[0],[1],[2]]
topol_1 = [[0,1],[1,2]]
mod_0D = (V1,topol_0)
mod_1D = (V1,topol_1)
V3,CV = larModelProduct([mod_2D,mod_1D])
mod_3D = (V3,CV)
VIEW(EXPLODE(1.2,1.2,1.2)(MKPOLS(mod_3D)))
print "\nk_3 =", len(CV), "\n"
```

nuweb17a17a.

nuweb18c ⟨ 2-skeleton extraction 18c ⟩ ≡

```
# 2-skeleton of the 3D product complex
mod_2D_1 = (V2,EV)
mod_3D_h2 = larModelProduct([mod_2D,mod_0D])
mod_3D_v2 = larModelProduct([mod_2D_1,mod_1D])
_,FV_h = mod_3D_h2
_,FV_v = mod_3D_v2
FV3 = FV_h + FV_v
SK2 = (V3,FV3)
VIEW(EXPLODE(1.2,1.2,1.2)(MKPOLS(SK2)))
print "\nk_2 =", len(FV3), "\n"
```

nuweb17a17a.

nuweb19a ⟨ 1-skeleton extraction 19a ⟩ ≡

```
# 1-skeleton of the 3D product complex
mod_2D_0 = (V2,AA(LIST)(range(len(V2))))
mod_3D_h1 = larModelProduct([mod_2D_1,mod_0D])
mod_3D_v1 = larModelProduct([mod_2D_0,mod_1D])
_,EV_h = mod_3D_h1
_,EV_v = mod_3D_v1
EV3 = EV_h + EV_v
SK1 = (V3,EV3)
VIEW(EXPLODE(1.2,1.2,1.2)(MKPOLS(SK1)))
print "\nk_1 =", len(EV3), "\n"
```

nuweb17a17a.

nuweb19b ⟨ 0-coboundary computation 19b ⟩ ≡

```
# boundary and coboundary operators
np.set_printoptions(threshold=sys.maxint)
csrFV3 = csrCreate(FV3)
csrEV3 = csrCreate(EV3)
csrVE3 = csrTranspose(csrEV3)
facetLengths = [csrCell.getnnz() for csrCell in csrEV3]
boundary = csrBoundaryFilter(csrVE3,facetLengths)
coboundary_0 = csrTranspose(boundary)
print "\ncoboundary_0 =\n", csrToMatrixRepresentation(coboundary_0)
```

nuweb17a17a.

nuweb19c ⟨ 1-coboundary computation 19c ⟩ ≡

```
csrEF3 = matrixProduct(csrEV3, csrTranspose(csrFV3))
facetLengths = [csrCell.getnnz() for csrCell in csrFV3]
boundary = csrBoundaryFilter(csrEF3,facetLengths)
coboundary_1 = csrTranspose(boundary)
print "\ncoboundary_1.T =\n", csrToMatrixRepresentation(coboundary_1.T)
```

nuweb17a17a.

nuweb20a ⟨ 2-coboundary computation 20a ⟩ ≡

```
csrCV = csrCreate(CV)
csrFC3 = matrixProduct(csrFV3, csrTranspose(csrCV))
facetLengths = [csrCell.getnnz() for csrCell in csrCV]
boundary = csrBoundaryFilter(csrFC3,facetLengths)
coboundary_2 = csrTranspose(boundary)
print "\ncoboundary_2 =\n", csrToMatrixRepresentation(coboundary_2)
```

nuweb17a17a.

nuweb20b ⟨ boundary chain visualisation 20b ⟩ ≡

```
# boundary chain visualisation
boundaryCells_2 = boundaryCells(CV,FV3)
boundary = (V3,[FV3[k] for k in boundaryCells_2])
VIEW(EXPLODE(1.5,1.5,1.5)(MKPOLS(boundary)))
```

nuweb17a17a.

5.6.2 Boundary of 3D simplicial grid

nuweb20c "test/py/larcc/ex2.py" 20c ≡

⟨ boundary of 3D simplicial grid nuweb20d20d ⟩

nuweb20d ⟨ boundary of 3D simplicial grid 20d ⟩ ≡

```
from simplexn import *
from larcc import *

V,CV = larSimplexGrid([10,10,3])
VIEW(EXPLODE(1.5,1.5,1.5)(MKPOLS((V,CV))))
SK2 = (V,larSimplexFacets(CV))
VIEW(EXPLODE(1.5,1.5,1.5)(MKPOLS(SK2)))
_,FV = SK2
SK1 = (V,larSimplexFacets(FV))
_,EV = SK1
VIEW(EXPLODE(1.5,1.5,1.5)(MKPOLS(SK1)))

boundaryCells_2 = boundaryCells(CV,FV)
boundary = (V,[FV[k] for k in boundaryCells_2])
VIEW(EXPLODE(1.5,1.5,1.5)(MKPOLS(boundary)))
print "\nboundaryCells_2 =\n", boundaryCells_2
```

nuweb20c20c.

5.6.3 Oriented boundary of a random simplicial complex

nuweb21a `"test/py/larcc/ex3.py"` 21a ≡
⟨ Importing external modules nuweb21b21b ⟩
⟨ Generating and viewing a random 3D simplicial complex nuweb21c21c ⟩
⟨ Computing and viewing its non-oriented boundary nuweb21d21d ⟩
⟨ Computing and viewing its oriented boundary nuweb22a22a ⟩

nuweb21b ⟨ Importing external modules 21b ⟩ ≡

```
from simplexn import *
from larcc import *
from scipy.spatial import Delaunay
import numpy as np
```

nuweb21a21a.

nuweb21c ⟨ Generating and viewing a random 3D simplicial complex 21c ⟩ ≡

```
verts = np.random.rand(10000, 3) # 1000 points in 3-d
verts = [AA(lambda x: 2*x)(VECTDIFF([vert,[0.5,0.5,0.5]])) for vert in verts]
verts = [vert for vert in verts if VECTNORM(vert) < 1.0]
tetra = Delaunay(verts)
cells = [cell for cell in tetra.vertices.tolist()
         if  ((verts[cell[0]][2]<0) and (verts[cell[1]][2]<0)
               and (verts[cell[2]][2]<0) and (verts[cell[3]][2]<0) ) ]
V, CV = verts, cells
VIEW(MKPOL([V,AA(AA(lambda k:k+1))(CV),[]]))
```

nuweb21a21a.

nuweb21d ⟨ Computing and viewing its non-oriented boundary 21d ⟩ ≡

```
FV = larSimplexFacets(CV)
VIEW(MKPOL([V,AA(AA(lambda k:k+1))(FV),[]]))
boundaryCells_2 = boundaryCells(CV,FV)
print "\nboundaryCells_2 =\n", boundaryCells_2
bndry = (V,[FV[k] for k in boundaryCells_2])
VIEW(EXPLODE(1.5,1.5,1.5)(MKPOLS(bndry)))
```

nuweb21a21a.

nuweb22a ⟨ Computing and viewing its oriented boundary 22a ⟩ ≡

```
boundaryCells_2 = signedBoundaryCells(V,CV,FV)
print "\nboundaryCells_2 =\n", boundaryCells_2
def swap(mylist): return [mylist[1]]+[mylist[0]]+mylist[2:]
boundaryFV = [FV[-k] if k<0 else swap(FV[k]) for k in boundaryCells_2]
bndry = (V,boundaryFV)
VIEW(EXPLODE(1.5,1.5,1.5)(MKPOLS(bndry)))
```

nuweb21a21a.

5.6.4 Oriented boundary of a simplicial grid

nuweb22b "test/py/larcc/ex4.py" 22b ≡

⟨ Generate and view a 3D simplicial grid nuweb22c22c ⟩
⟨ Computing and viewing the 2-skeleton of simplicial grid nuweb22d22d ⟩
⟨ Computing and viewing the oriented boundary of simplicial grid nuweb22e22e ⟩

nuweb22c ⟨ Generate and view a 3D simplicial grid 22c ⟩ ≡

```
from simplexn import *
from larcc import *
V,CV = larSimplexGrid([4,4,4])
VIEW(EXPLODE(1.5,1.5,1.5)(MKPOLS((V,CV))))
```

nuweb22b22b.

nuweb22d ⟨ Computing and viewing the 2-skeleton of simplicial grid 22d ⟩ ≡

```
FV = larSimplexFacets(CV)
EV = larSimplexFacets(FV)
VIEW(EXPLODE(1.5,1.5,1.5)(MKPOLS((V,FV))))
```

nuweb22b22b.

nuweb22e ⟨ Computing and viewing the oriented boundary of simplicial grid 22e ⟩ ≡

```
csrSignedBoundaryMat = signedBoundary (V,CV,FV)
boundaryCells_2 = signedBoundaryCells(V,CV,FV)
def swap(l): return [l[1],l[0],l[2]]
boundaryFV = [FV[-k] if k<0 else swap(FV[k]) for k in boundaryCells_2]
boundary = (V,boundaryFV)
VIEW(EXPLODE(1.5,1.5,1.5)(MKPOLS(boundary)))
```

nuweb22b22b.

5.6.5 Skeletons and oriented boundary of a simplicial complex

nuweb23a "test/py/larcc/ex5.py" 23a ≡
⟨ Skeletons computation and vilualisation nuweb23b23b ⟩
⟨ Oriented boundary matrix visualization nuweb23c23c ⟩
⟨ Computation of oriented boundary cells nuweb23d23d ⟩

nuweb23b ⟨ Skeletons computation and vilualisation 23b ⟩ ≡

```
from simplexn import *
from larcc import *
V,FV = larSimplexGrid([3,3])
VIEW(EXPLODE(1.5,1.5,1.5)(MKPOLS((V,FV))))
EV = larSimplexFacets(FV)
VIEW(EXPLODE(1.5,1.5,1.5)(MKPOLS((V,EV))))
VV = larSimplexFacets(EV)
VIEW(EXPLODE(1.5,1.5,1.5)(MKPOLS((V,VV))))
```

nuweb23a23a.

nuweb23c ⟨ Oriented boundary matrix visualization 23c ⟩ ≡

```
np.set_printoptions(threshold='nan')
csrSignedBoundaryMat = signedBoundary (V,FV,EV)
Z = csrToMatrixRepresentation(csrSignedBoundaryMat)
print "\ncsrSignedBoundaryMat =\n", Z
from pylab import *
matshow(Z)
show()
```

nuweb23a23a.

nuweb23d ⟨ Computation of oriented boundary cells 23d ⟩ ≡

```
boundaryCells_1 = signedBoundaryCells(V,FV,EV)
print "\nboundaryCells_1 =\n", boundaryCells_1
def swap(mylist): return [mylist[1]]+[mylist[0]]+mylist[2:]
boundaryEV = [EV[-k] if k<0 else swap(EV[k]) for k in boundaryCells_1]
bndry = (V,boundaryEV)
VIEW(EXPLODE(1.5,1.5,1.5)(MKPOLS(bndry)))
```

nuweb23a23a.

5.6.6 Boundary of random 2D simplicial complex

nuweb24a `"test/py/larcc/ex6.py"` 24a ≡

```
from simplexn import *
from larcc import *
from scipy.spatial import Delaunay
⟨ Test for quasi-equilateral triangles nuweb24b24b ⟩
⟨ Generation and selection of random triangles nuweb25a25a ⟩
⟨ Boundary computation and visualisation nuweb25b25b ⟩
```

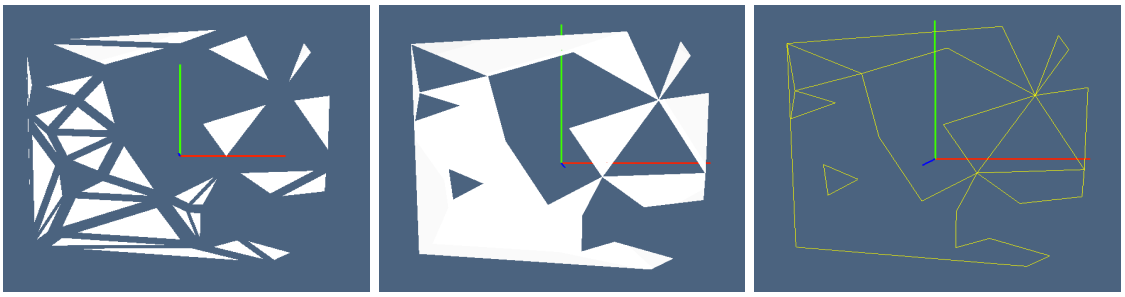

Figure 5.1: example caption

nuweb24b ⟨ Test for quasi-equilateral triangles 24b ⟩ ≡

```
def quasiEquilateral(tria):
    a = VECTNORM(VECTDIFF(tria[0:2]))
    b = VECTNORM(VECTDIFF(tria[1:3]))
    c = VECTNORM(VECTDIFF([tria[0],tria[2]]))
    m = max(a,b,c)
    if m/a < 1.7 and m/b < 1.7 and m/c < 1.7: return True
    else: return False
```

nuweb24a24a.

nuweb25a ⟨ Generation and selection of random triangles 25a ⟩ ≡

```
verts = np.random.rand(20,2)
verts = (verts - [0.5,0.5]) * 2
triangles = Delaunay(verts)
cells = [ cell for cell in triangles.vertices.tolist()
          if (not quasiEquilateral([verts[k] for k in cell])) ]
V, FV = AA(list)(verts), cells
EV = larSimplexFacets(FV)
pols2D = MKPOLS((V,FV))
VIEW(EXPLODE(1.5,1.5,1.5)(pols2D))
```

nuweb24a24a.

nuweb25b ⟨ Boundary computation and visualisation 25b ⟩ ≡

```
boundaryCells_1 = signedBoundaryCells(V,FV,EV)
print "\nboundaryCells_1 =\n", boundaryCells_1
def swap(mylist): return [mylist[1]]+[mylist[0]]+mylist[2:]
boundaryEV = [EV[-k] if k<0 else swap(EV[k]) for k in boundaryCells_1]
bndry = (V,boundaryEV)
VIEW(STRUCT(MKPOLS(bndry) + pols2D))
VIEW(COLOR(RED)(STRUCT(MKPOLS(bndry))))
```

nuweb24a24a.

nuweb25c ⟨ Compute the topologically ordered chain of boundary vertices 25c ⟩ ≡

.

nuweb26a ⟨ Decompose a permutation into cycles 26a ⟩ ≡

```
def permutationOrbits(List):
   d = dict((i,int(x)) for i,x in enumerate(List))
   out = []
   while d:
      x = list(d)[0]
      orbit = []
      while x in d:
         orbit += [x],
         x = d.pop(x)
      out += [CAT(orbit)+orbit[0]]
   return out

if __name__ == "__main__":
   print [2, 3, 4, 5, 6, 7, 0, 1]
   print permutationOrbits([2, 3, 4, 5, 6, 7, 0, 1])
   print [3,9,8,4,10,7,2,11,6,0,1,5]
   print permutationOrbits([3,9,8,4,10,7,2,11,6,0,1,5])
```

.

5.6.7 Assemblies of simplices and hypercubes

nuweb26b "test/py/larcc/ex7.py" 26b ≡

```
from simplexn import *
from larcc import *
from largrid import *
```

⟨ Definition of 1-dimensional LAR models nuweb27a27a ⟩
⟨ Assembly generation of squares and triangles nuweb27b27b ⟩
⟨ Assembly generation of cubes and tetrahedra nuweb27c27c ⟩

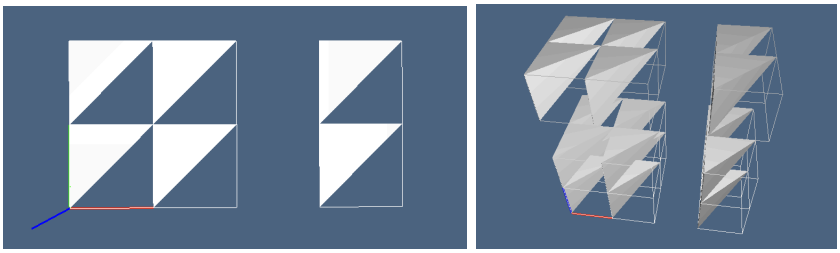

Figure 5.2: (a) Assemblies of squares and triangles; (b) assembly of cubes and tetrahedra.

nuweb27a ⟨ Definition of 1-dimensional LAR models 27a ⟩ ≡

```
geom_0,topol_0 = [[0.],[1.],[2.],[3.],[4.]],[[0,1],[1,2],[3,4]]
geom_1,topol_1 = [[0.],[1.],[2.]], [[0,1],[1,2]]
mod_0 = (geom_0,topol_0)
mod_1 = (geom_1,topol_1)
```

nuweb26b26b.

nuweb27b ⟨ Assembly generation of squares and triangles 27b ⟩ ≡

```
squares = larModelProduct([mod_0,mod_1])
V,FV = squares
simplices = pivotSimplices(V,FV,d=2)
VIEW(STRUCT([ MKPOL([V,AA(AA(C(SUM)(1)))(simplices),[]]),
              SKEL_1(STRUCT(MKPOLS((V,FV)))) ]))
```

nuweb26b26b.

nuweb27c ⟨ Assembly generation of cubes and tetrahedra 27c ⟩ ≡

```
cubes = larModelProduct([squares,mod_0])
V,CV = cubes
simplices = pivotSimplices(V,CV,d=3)
VIEW(STRUCT([ MKPOL([V,AA(AA(C(SUM)(1)))(simplices),[]]),
           SKEL_1(STRUCT(MKPOLS((V,CV)))) ]))
```

nuweb26b26b.

Bibliography