Efficient grid generation in LAR-CC [*]

Alberto Paoluzzi

December 31, 2013

Abstract

Here we develop an efficient implementation of multidimensional grid generation of cuboidal 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 $d \leq n$.

Contents

[*]This document is part of the *Linear Algebraic Representation with CoChains* (LAR-CC) framework [?]. December 31, 2013

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 **??** we give the simple implementation of generation of lower-dimensional (say, either 0- or 1-dimensional) regular cellular complexes with integer coordinates. In Section **??** 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 set $\{0, 2^n\}$. In Section **??** 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 4.1 the exporting of the module to different languages is provided. The Section **??** 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 **??** the indexing structure of the macro sources and variables is exposed by the sake of the reader. In the Appendix, the Section **??** contains some programming utilities possibly needed by the developers.

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.

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.

@d Generation of uniform 0D cellular complex @def grid0(n): cells = AA(LIST)(range(n+1)) return cells @

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

@d Generation of uniform 1D cellular complex @def grid1(n): ints = range(n+1) cells = TRANS([ints[:-1],ints[1:]]) return cells @

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 `order` parameter, to take values in the set $\{0, 1\}$. @d Generation of cellular complex of 0/1 dimension d @def larGrid(n): def larGrid1(d): if d==0: return grid0(n) elif d==1: return grid1(n) return larGrid1 @

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.

@d Generation of vertices of decompositions of 1D intervals @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 = AA(LIST)(AA(PROD)(TRANS([ints,items]))) return vertices return larSplit1 @

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 $\mathbb{E}^n$.

3.1 Full-dimensional grids

3.1.1 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 $(x_0, x_1, \ldots, x_{n-1})$. @d Generation of grid vertices @def larVertProd(vertLists): return AA(CAT)(CART(vertLists)) @

3.1.2 Mapping of indices to storage

Multi-index to address transformation The second-order utility `index2address` 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 `index2address([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]`.

@d Transformation from multindex to address in a linear array storage @def index2address (shape): n = len(shape) shape = shape[1:]+[1] weights = [PROD(shape[k:]) for k in range(n)] def index2address0 (multindex): return INNERPROD([multindex, weights]) return index2address0 @

`index2address` 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 `index2address([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 `index2address([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 `index2address([4,3,6])` 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.

@d Test example @¿¿¿ index2address([3,6])([2,0]) 12 ¿¿¿ AA(index2address([3,6]))(CART([range(0,3), range(0,6)])) [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17] ¿¿¿ AA(index2address([4,3,6]))(CART([range(0,4), range(0,3), range(0,6)])) [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] ¿¿¿ index2address([4])([2]) 2 @

3.1.3 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 $0 \leq d \leq n$.

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

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 4a.

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

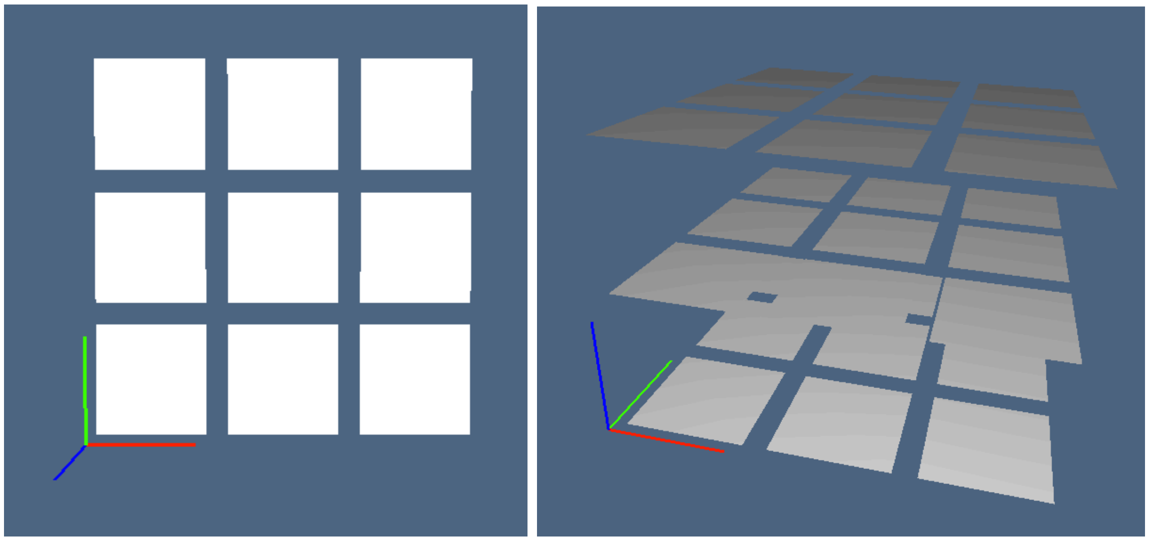

Figure 1: Exploded views of models `grid2D` and `grid3D`.

Notice that `grid2D`, generated by product of two 1-complexes, is *solid* in $\mathbb{E}^2$, whereas `grid3D` shown in Figure 4b, generated by product of two 1-complexes and one 0-complex, is two-dimensional and embedded in $\mathbb{E}^3$.

@D Example of cuboidal grid of dimensions (2, 3) @v1, c1 = [[0.],[1.],[2.],[3.]],[[0,1],[1,2],[2,3]] v0, c0 = [[0.],[1.],[2.]], [[0],[1],[2]] 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 `multindex` in `indices`.

@D Generation of grid cells @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(multindex,cellLists)]) for multindex in indices] convert = index2address([shape+1 if (len(cellLists[k][0]) ¿ 1) else shape for k,shape in enumerate(shapes)]) return [AA(convert)(cell) for cell in jointCells] @

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 `index2address0`, 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 $|$`larCellProd([c1,c1])`$| = 3 \times 3 = 9$.

@D Tracing the evaluation of expression "`larCellProd([c1,c1])`" @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]] @

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 `binaryPowerRange` function, given in Section 3.2.1.

The binary representationn of such numbers are in fact filtered according to the number of their ones in Section 3.2.2, and used to generate the distinct components of the skeletons of different orders of the assembled grid complexes in Section 3.2.3.

3.2.1 Generation of skeleton components

The `binaryPowerRange` 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.

@D Enumeration of binary ranges of given order @def binaryPowerRange (n): return [('0:0'+str(n)+'b').format(k) for k in range(2**n)] @

Examples of generation of bit strings Below we show the outputs returned by application of the `binaryPowerRange` function to the first 4 integers. @D Binary range examples @¿¿¿ print binaryPowerRange (4), ['0000', '0001', '0010', '0011', '0100', '0101', '0110', '0111', '1000', '1001', '1010', '1011', '1100', '1101', '1110', '1111'] ¿¿¿ print binaryPowerRange (3), ['000', '001', '010', '011', '100', '101', '110', '111'] ¿¿¿ print binaryPowerRange (2), ['00', '01', '10', '11'] ¿¿¿ print binaryPowerRange (1), ['0', '1'] @

3.2.2 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.

@D Filtering binary ranges by order @def filterByOrder(n): terms = [AA(int)(list(term)) for term in binaryPowerRange(n)] return [[term for term in terms if sum(term) == k] for k in range(n+1)] @

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 $\sum_{d=0}^{n}\binom{n}{d} = 2^n$.

@D Skeleton component examples @¿¿¿ 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]]] @

3.2.3 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 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.

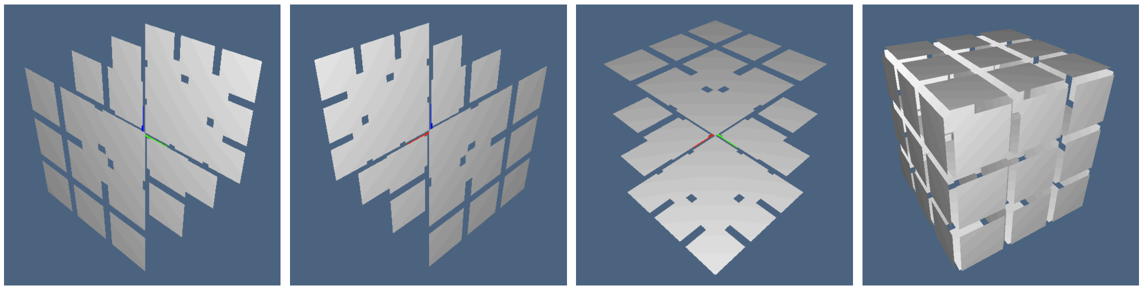
Figure 2: (a,b,c) Exploded views of subsets (orthogonal to coordinate axes) of 2-cells of a 2-skeleton grid; (d) their assembled set.

@D Assembling grid skeletons @def larGridSkeleton(shape): n = len(shape) def larGridSkeleton0(d): components = filterByOrder(n)[d] componentCellLists = [AA(APPLY)(TRANS([AA(larGrid)(shape),(component)])) for component in components] return CAT([larCellProd(cellLists) for cellLists in componentCellLists]) return larGridSkeleton0 @

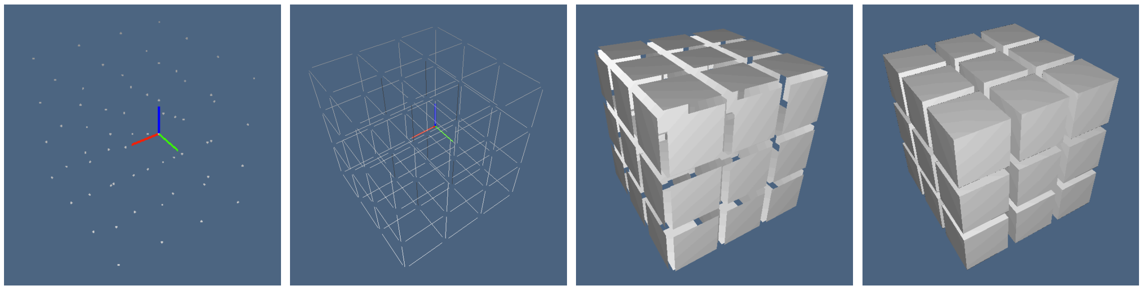
Figure 3: Exploded views of 0-, 1-, 2-, and 3-dimensional skeletons.

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, but remember that the PLaSM visualiser always embed in 3D the displayed model.

@D Multidimensional grid generation @def larCuboids(shape, full=False): def vertex-Domain(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 @

Multidimensional visualisation examples Visualisation examples of grid of dimension 1,2, and 3 are given below and are displayed in Figure 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.

@d Multidimensional visualisation examples @VIEW(EXPLODE(1.5,1.5,1.5)(MKPOLS(larCuboids([3],Tru
VIEW(EXPLODE(1.5,1.5,1.5)(MKPOLS(larCuboids([3,2],True)))) VIEW(EXPLODE(1.5,1.5,1.5)(MKPOLS(
@

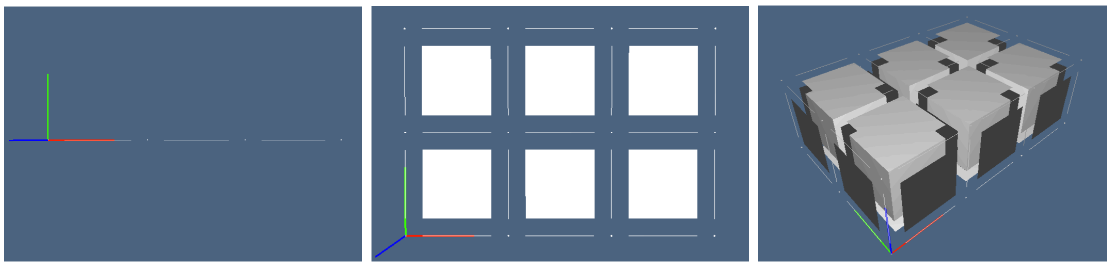

Figure 4: Exploded views of 1D, 2D, and 3D cellular complexes (including cells of dimension 0,1,2, and 3).

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 emberddded d-skeleton $(0 \leq d < n)$.

Binary product of cellular complexes The texttttlarModelProduct 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.

@d Cartesian product of two lar models @def larModelProduct(twoModels): model1, model2 = twoModels V, cells1 = model1 W, cells2 = model2 @¡ Generation of product vertices @¿ @¡ Generation of product cells @¿ model = AA(list)(vertices.keys()), sorted(cells) return model @

@d Generation of product vertices @vertices = collections.OrderedDict(); k = 0 for v in V: for w in W: id = tuple(v+w) if not vertices.has$_k ey(id) : vertices[id] = kk+ = 1$@

@d Generation of product cells @cells = [sorted([verts[tuple(V[v]+W[w])] for v in c1 for w in c2]) for c1 in cells1 for c2 in cells2] @

@d Test examples @if $_name_{==''} {}_{main_{''}:geom_0,topol_0=[[0.],[1.],[2.],[3.],[4.]],[[0,1],[1,2],[2,3],[3,4]]geom_1,topol_1=[[0.],[1.],[2.]],[[0,1],[1,2]]mod_0=(g}$

squares = larModelProduct([mod$_0, mod_1])VIEW(EXPLODE(1.2, 1.2, 1.2)(MKPOLS(squares)))$

cubes = INSL(larModelProduct)([mod$_0, mod_1, mod_0]) == cubes = larModelProduct([squares, mod_0])VI$

@d Test examples @mod$_1 = grid1(1)(4)squares = larModelProduct([mod_1, mod_1])VIEW(EXPLODE(1$
$larModelProduct([squares, mod_1])VIEW(EXPLODE(1.2, 1.2, 1.2)(MKPOLS(cubes)))cubes =$
$boundCellsAdded([[0, 1], [0, 1], [0, 1]])(cubes)$@

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. @O lib/py/largrid.py @"""Module with functions for grid generation and Cartesian product""" @¡ Importing `smplxn` and `numpy` libraries @¿ @¡ Generation of vertices of decompositions of 1D intervals @¿ @¡ Generation of uniform 0D cellular complex @¿ @¡ Generation of uniform 1D cellular complex @¿ @¡ Generation of cellular complex of 0/1 dimension *d* @¿ @¡ Generation of grid vertices @¿ @¡ Transformation from multindex to address in a linear array storage @¿ @¡ Generation of grid cells @¿ @¡ Enumeration of binary ranges of given order @¿ @¡ Filtering binary ranges by order @¿ @¡ Assembling grid skeletons @¿ @¡ Multidimensional grid generation @¿ @¡ Cartesian product of two lar models @¿ if $_name_{==''} {}_{main_{''}:@<Multidimensionalvisualisationexamples@>@}$

6 Unit tests

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

@d create directory and echo of creation @@¡ Create directory from path @¿ @createDir('@1') print "'@1' repository created" @

@o test/py/largrid/test01.py @@¡ create directory and echo of creation: @(test/py/largrid/@) @¿ @

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.

@O test/py/largrid/test01.py @from pyplasm import * @¡Generation of vertices of decompositions of 1D intervals@¿ 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]] @

7 Indices

The list of macros follow. @m

A Utilities

@d Importing `smplxn` and `numpy` libraries @from smplxn import * import numpy as np @

An useful utility will allow for the creation of a subdirectory from a `dirpath` *string.* @d Create directory from path @import os def createDir(dirpath): if not os.path.exists(dirpath): os.makedirs(dirpath) @— createDir @

It may be useful to define the repository(ies) for the unit tests associated to the module: @o test/py/largrid-tests.py @@¡ Create directory from path @¿ createDir('test/py/largrid/') @