Hypercuboidal grids and topological products in LARCC *

LARCC team

May 31, 2014

Abstract

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 $d \leq n$.

Contents

*This document is part of the *Linear Algebraic Representation with CoChains* (LARCC) framework [CL13]. May 31, 2014

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 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 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 5 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 6 the exporting of the module to different languages is provided. The Section 7 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 8 the indexing structure of the macro sources and variables is exposed by the sake of the reader. The Appendix A 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.

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

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

◇

Macro referenced in 20b.

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

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

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

◇

Macro referenced in 20b.

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.

⟨ 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
```

◇

Macro referenced in 20b.

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.

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

```
def larSplit(dom):
    def larSplit1(n):
        # assert n > 0 and isinstance(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
```

◇

Macro referenced in 20b, 21d.

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 $(x_0, x_1, \ldots, x_{n-1})$.

⟨ Generation of grid vertices 4b ⟩ ≡

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

◇

Macro referenced in 20b.

3.1.2 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

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

$(w_0, w_1, ..., w_{d-2}, 1)$:

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

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]`.

⟨ 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
```

◇

Macro referenced in 20b.

`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])` 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.

⟨ 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]
>>> convert = index2addr([4,3,6])
<function index2address0>
>>> [convert(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
◇
```

Macro never referenced.

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 $0 \leq d \leq n$.

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 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 1b, generated by product of two 1-complexes and one 0-complex, is two-dimensional and embedded in $\mathbb{E}^3$.

⟨ 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]]
vertGrid = larVertProd([v1, v1, v0])
cellGrid = larCellProd([c1, c1, c0])
```

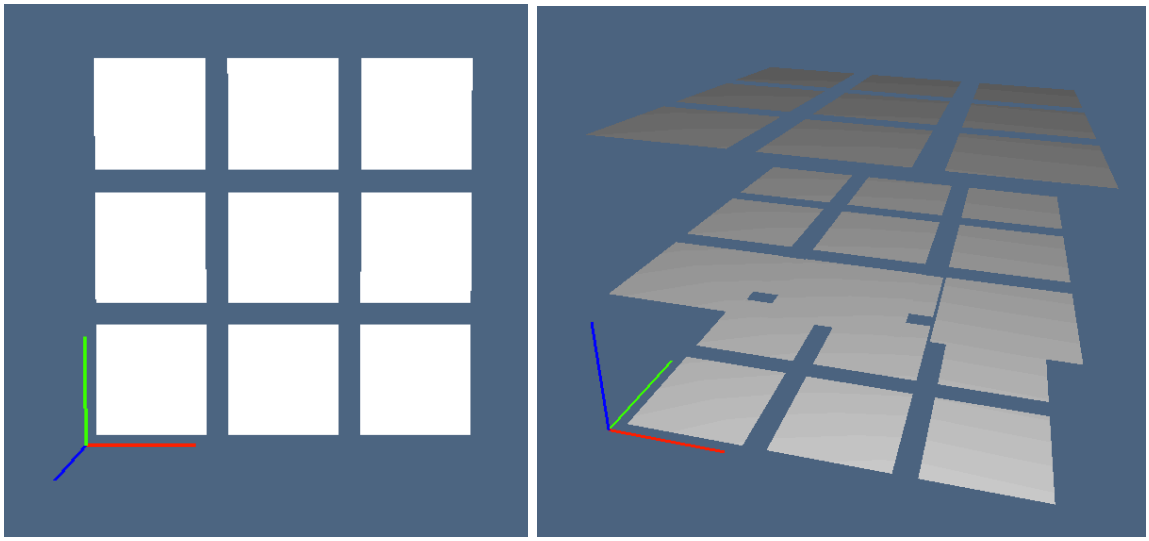

Figure 1: Exploded views of models `grid2D` and `grid3D`.

```
grid3D = vertGrid,cellGrid
VIEW(EXPLODE(1.2,1.2,1.2)(MKPOLS(grid3D)))
```
◇

Macro never referenced.

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`.

⟨ 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]
```
◇

Macro referenced in 20b.

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 $|\texttt{larCellProd([c1,c1])}| = 3 \times 3 = 9$.

⟨ 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]]
```

◇

Macro never referenced.

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 3.2.1.

The binary representations 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 different order skeletons of the assembled grid complexes in Section 3.2.3.

3.2.1 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.

⟨ 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)]
```

◇

Macro referenced in 20b.

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

⟨ 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']
```

◇

Macro never referenced.

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.

⟨ 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)]
```
◇

Macro referenced in 20b.

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

⟨ 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]]]
```
◇

Macro never referenced.

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.

⟨ Assembling grid skeletons 10b ⟩ ≡

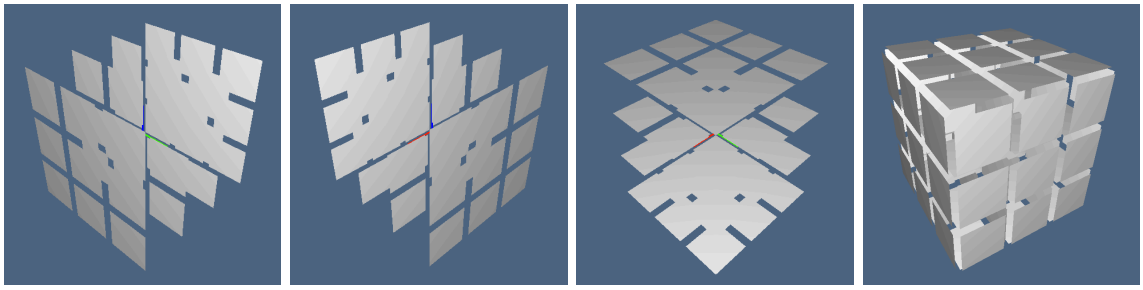

Figure 2: (a,b,c) Exploded views of subsets (orthogonal to coordinate axes) of 2-cells of a 2-skeleton grid; (d) their assembled set.

```
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
◇
```

Macro referenced in 20b.

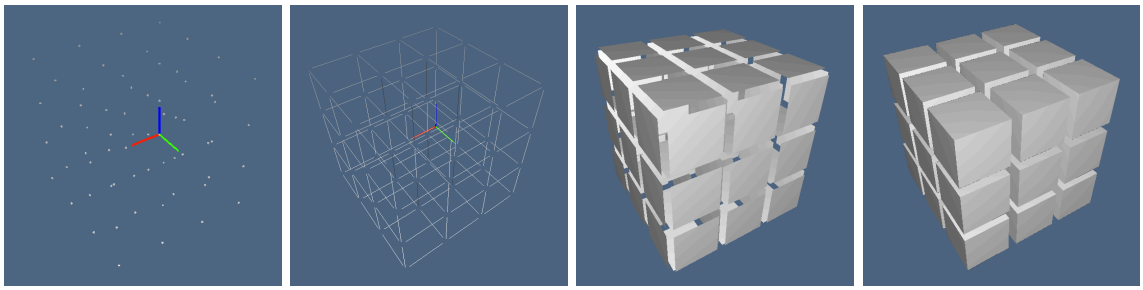

Figure 3: Exploded views of 0-, 1-, 2-, and 3-dimensional skeletons.

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.

⟨ Multidimensional grid generation 12a ⟩ ≡

```
def larImageVerts(shape):
   def vertexDomain(n):
      return [[k] for k in range(n)]
   vertLists = [vertexDomain(k+1) for k in shape]
   vertGrid = larVertProd(vertLists)
   return vertGrid

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

◇

Macro referenced in 20b.

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.

⟨ Multidimensional visualisation examples 12b ⟩ ≡

```
def mergeSkeletons(larSkeletons): return larSkeletons[0],CAT(larSkeletons[1])
VIEW(EXPLODE(1.5,1.5,1.5)(MKPOLS(mergeSkeletons(larCuboids([3],True)))))
VIEW(EXPLODE(1.5,1.5,1.5)(MKPOLS(mergeSkeletons(larCuboids([3,2],True)))))
VIEW(EXPLODE(1.5,1.5,1.5)(MKPOLS(mergeSkeletons(larCuboids([3,2,1],True)))))
```

◇

Macro referenced in 20b.

3.4 Chain of boundary operators

As we know, a *chain complex* is a sequence of (linear) chain spaces C_k $(d \geq k \geq 0)$ and a sequence of boundary operators $\partial_k : C_k \to C_{k-1}$ $(d \geq k \geq 1)$ between adjacent spaces (see

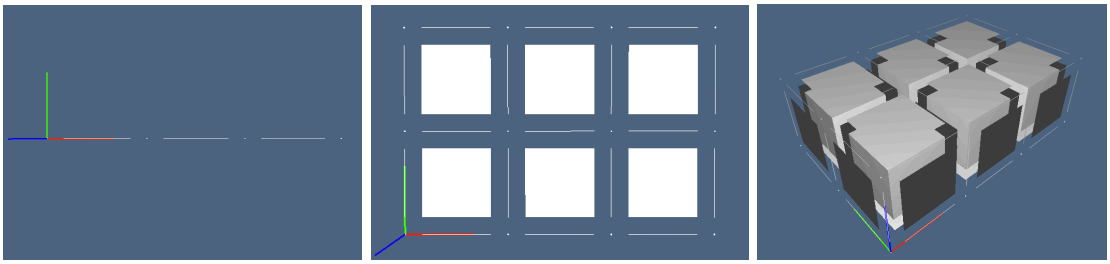

Figure 4: Exploded views of 1D, 2D, and 3D cellular complexes (including cells of dimension 0,1,2, and 3).

Figure 5). In this section, we aim to generate the sequence of boundary matrices `CSR`$([\partial_k])$ $(1 \leq k \leq d)$.

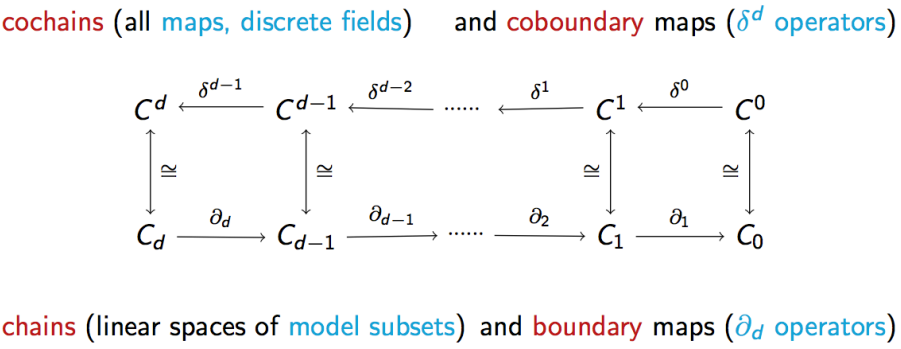

Figure 5: Chain and cochain complexes.

Cuboidal skeletons A list of `BRC` characteristic matrices of cellular k-complexes $(0 \leq k \leq d)$ with dimension d, where $d =$ `len(shape)`, is returned by the function `gridSkeletons` in the macro below, where the input is given by the *shape* of the grid, i.e. by the list of cell items in each coordinate direction. Some simple test examples of skeletons of cuboidal complexes are also printed when the `largrid` module run as the `main`. Just notice that the number of returned d-cells is equal to `PROD(shape)`.

⟨ Multidimensional grid skeletons 13 ⟩ ≡

```
def gridSkeletons(shape):
    gridMap = larGridSkeleton(shape)
    skeletonIds = range(len(shape)+1)
    skeletons = [ gridMap(id) for id in skeletonIds ]
    return skeletons
```

```
if __name__=="__main__":
   print "\ngridSkeletons([3]) =\n", gridSkeletons([3])
   print "\ngridSkeletons([3,2]) =\n", gridSkeletons([3,2])
   print "\ngridSkeletons([3,2,1]) =\n", gridSkeletons([3,2,1])
◇
```

Macro referenced in 20b.

Boundary complex of a cuboidal grid The list of boundary matrices `CSR`($[\partial_k]$) ($1 \leq k \leq d$) is returned by the function `gridBoundaryMatrices`.

⟨ Generation of grid boundary complex 14a ⟩ ≡

```
def gridBoundaryMatrices(shape):
   skeletons = gridSkeletons(shape)
   boundaryMatrices = [boundary(skeletons[k+1],faces)
                  for k,faces in enumerate(skeletons[:-1])]
   return boundaryMatrices

if __name__=="__main__":
   for k in range(1):
      print "\ngridBoundaryMatrices([3]) =\n", \
            csr2DenseMatrix(gridBoundaryMatrices([3])[k])
   for k in range(2):
      print "\ngridBoundaryMatrices([3,2]) =\n", \
            csr2DenseMatrix(gridBoundaryMatrices([3,2])[k])
   for k in range(3):
      print "\ngridBoundaryMatrices([3,2,1]) =\n", \
            csr2DenseMatrix(gridBoundaryMatrices([3,2,1])[k])
◇
```

Macro referenced in 20b.

4 Face stack of cellular complexes

4.1 Simplicial complexes

The stack of faces of a simplicial d-complex is easy to compute making use of the combinatorial properties of the simplex boundary. If the input is the compressed sparse row representation `CSR`(M_d) of the binary characteristic matrix M_d of the highest rank cells (d-simplices), we repeatedly apply the `larSimplexFacets` function.

Simplicial face stack computation The whole stack of `LAR` cell-vertex arrays is computed below for the multidimensional case, and returned ordered from 0-cells to d-cells.

⟨ Simplicial face stack computation 14b ⟩ ≡

```
""" Simplicial face stack computation """
def larSimplicialStack(simplices):
   dim = len(simplices[0])-1
   faceStack = [simplices]
   for k in range(dim):
      faces = larSimplexFacets(faceStack[-1])
      faceStack.append(faces)
   return REVERSE(faceStack)
◇
```

Macro referenced in 20b.

Oriented boundary: Example 2D The file `test/py/largrid/test04.py` gives an example of computation of the oriented 1D boundary of a 2D simplicial grid. In the variable `bases` we store the stack of bases of k-chains, for $0 \leq k \leq 2$. The variable `boundaryCells` contains the indices of boundary cells, signed according to their absolute orientation. To get a coherent orientation of the model boundary, the boundary 2-cells with negative indices must undergo to reversing their orientation.

"test/py/largrid/test04.py" 15a ≡

```
""" Computation of the boundary of a simplicial grid """
import sys; sys.path.insert(0, 'lib/py/')
from larcc import *
from largrid import *

V,simplices = larSimplexGrid1((2,2))
bases = larSimplicialStack(simplices)
boundaryCells = signedBoundaryCells(V,bases[-1],bases[-2])

def swap(mylist): return [mylist[1]]+[mylist[0]]+mylist[2:]
orientedBoundary = [bases[-2][-k] if k<0 else swap(bases[-2][k])
                    for k in boundaryCells]
VIEW(EXPLODE(1.2,1.2,1.2)(MKPOLS((V,orientedBoundary))))
submodel = STRUCT(MKPOLS((V,orientedBoundary)))
VIEW(larModelNumbering(V,bases,submodel))
◇
```

Oriented boundary: Example 3D A very similar example is given below for a 3D simplicial grid, in order to show how to use the components of the LAR cell stack, computed as `larSimplicialStack(CV)`, and stored in the `bases` variable. T

"test/py/largrid/test03.py" 15b ≡

```
""" Computation of the boundary of a simplicial grid """
import sys; sys.path.insert(0, 'lib/py/')
from larcc import *
```

```
from largrid import *

V,CV = larSimplexGrid1((2,2,2))
bases = larSimplicialStack(CV)
VV,EV,FV,CV = bases
boundaryCells = signedBoundaryCells(V,CV,FV)

def swap(mylist): return [mylist[1]]+[mylist[0]]+mylist[2:]
orientedBoundary = [FV[-k] if k<0 else swap(FV[k]) for k in boundaryCells]
VIEW(EXPLODE(1.2,1.2,1.2)(MKPOLS((V,orientedBoundary))))
submodel = EXPLODE(1.05,1.05,1.05)(MKPOLS((V,orientedBoundary)))
VIEW(larModelNumbering(V,bases,submodel))
◇
```

4.2 Cuboidal complexes

In order to compute the faces stack of a cuboidal d-complex, we are going to preeliminary extract the subset of boundary vertices, identified by the condition of having less than 8 incident 3-cells, and in general—for a cuboidal d-complex—a number less than 2^d of incident d-cells.

Identification of boundary vertices

⟨ Identification of boundary vertices of a cuboidal complex 16a ⟩ ≡

```
""" Boundary vertices of a cuboidal complex """
def cuboidalComplexBoundaryVertices(model):
   V,CV = model
   d = len(V[0])
   csrVC = csrCreate(CV).T
   csrVC.todense()
   exterior = [v for v in range(csrVC.shape[0]) if sum(csrVC[v].data)<int(2**d) ]
   return exterior
◇
```

Macro referenced in 20b.

Example 2D

"test/py/largrid/test05.py" 16b ≡

```
""" Extraction of boundary vertices of a cuboidal complex """
import sys; sys.path.insert(0, 'lib/py/')
from largrid import *

shape = (50,50)
model = larCuboids(shape)
V,cells = model
```

```
exterior = cuboidalComplexBoundaryVertices(model)
VIEW(STRUCT(MKPOLS((V,AA(LIST)(exterior)))))
VIEW(STRUCT(MKPOLS(larFacets((V,cells),dim=2))))
V,facets = larFacets((V,cells+[exterior]),dim=2)
EV = improperFacetsCovering(facets,cells,2)
VIEW(EXPLODE(1.2,1.2,1)(MKPOLS((V,EV))))
◇
```

Example 3D

"test/py/largrid/test06.py" 17a ≡

```
""" Extraction of boundary vertices of a cuboidal complex """
import sys; sys.path.insert(0, 'lib/py/')
from largrid import *

shape = (10,10,10)
model = larCuboids(shape)
V,cells = model
exterior = cuboidalComplexBoundaryVertices(model)
VIEW(STRUCT(MKPOLS((V,AA(LIST)(exterior)))))
V,facets = larFacets((V,cells+[exterior]))
VIEW(EXPLODE(1.2,1.2,1.2)(MKPOLS(larFacets((V,cells+[exterior])))))
FV = improperFacetsCovering(facets,cells)
VIEW(EXPLODE(1.2,1.2,1.2)(MKPOLS((V,FV))))
◇
```

4.2.1 Improper facets decomposition

When computing the $(d-1)$-facets adjacent to the *exterior* face, by using the `larFacets` function, some output elements may be said *improper*, since they contain a number of vertices higher than 2^{d-1}, as shown by the previous examples `test/py/largrid/test05.py` and `test/py/largrid/test06.py`. Hence the necessity of properly decomposing such improper subsets of exterior vertices into a suitable collection of regular $(d-1)$-facets (each with 2^{d-1} vertices).

Cofaces of improper facets The input is the set of facets extracted by the `larFacets` function, that include some improper facet, i.e. some vertex subset σ_{d-1}, with $|\sigma_{d-1}| > 2^{d-1}$. The output of the `improperFacetsCovering` function is the full set of regular facets, including the covering with two or more facets of 2^{d-1} cardinality, of each of the improper facets. The first step is the computation of the σ_d cofaces of improper facets. The default case is 3D. To use the `improperFacetsCovering` function for lower (or higher) dimensionality, the dimension `dim` of the input LAR `cells` must be explicitly given.

⟨ Identification of cofaces of improper facets 17b ⟩ ≡

```
""" Improper facets decomposition """
def improperFacetsCovering(facets,cells,dim=3):
   improperFacets = [facet for facet in facets if len(facet)>int(2**(dim-1))]
   cofaces = AA(set)(cells)
   facets = AA(set)(facets)
   fathers = [coface for facet in improperFacets for coface in cofaces
            if set(facet).intersection(coface)==set(facet)]
   brothers = [sorted( [ facet for facet in facets
               if set(facet).intersection(coface)==set(facet) ],
               key=lambda x: len(x) )
                  for coface in fathers]
   out = []
   if dim==2:
      for father,sons in zip(fathers,brothers):
         out += [ list( sons[-1].difference(sons[0]) ),
               list( sons[-1].difference(sons[1]) ) ]
   if dim==3:
      for father,sons in zip(fathers,brothers):
         if len(sons[-1])==7:
            out += [ list( sons[-1].difference(sons[0]) ),
               list( sons[-1].difference(sons[1]) ),
               list( sons[-1].difference(sons[2]) ) ]
         if len(sons[-1])==6:
            a = list( sons[-1].difference(sons[0]) )
            if len(a)==4: out += [a]
            a = list( sons[-1].difference(sons[1]) )
            if len(a)==4: out += [a]
            a = list( sons[-1].difference(sons[2]) )
            if len(a)==4: out += [a]
            a = list( sons[-1].difference(sons[3]) )
            if len(a)==4: out += [a]

   facets = [facet for facet in facets if len(facet)==int(2**(dim-1))]
   return AA(list)(facets) + out
```
◇

Macro referenced in 20b.

4.2.2 Random polytopal complexes

4.3 Polytopal complexes

TODO

5 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.

⟨ Cartesian product of two lar models 19a ⟩ ≡

```
def larModelProduct(twoModels):
    (V, cells1), (W, cells2) = twoModels
    ⟨ Cartesian product of vertices 19b ⟩
    ⟨ Topological product of cells 19c ⟩
    model = [list(v) for v in vertices.keys()], cells
    return model
```

◇

Macro referenced in 20b.

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.

⟨ Cartesian product of vertices 19b ⟩ ≡

```
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
```

◇

Macro referenced in 19a.

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

⟨ Topological product of cells 19c ⟩ ≡

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

◇

Macro referenced in 19a.

⟨ Test examples of Cartesian product 20a ⟩ ≡

```
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)))
```

◇

Macro referenced in 20b.

6 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.

"lib/py/largrid.py" 20b ≡

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

⟨ Importing `simplexn` and `numpy` libraries 23a ⟩
⟨ Import the module (20c `larcc`) 23d ⟩
⟨ Generation of vertices of decompositions of 1D intervals 4a ⟩
⟨ Generation of uniform 0D cellular complex 3a ⟩
⟨ Generation of uniform 1D cellular complex 3b ⟩
⟨ Generation of cellular complex of 0/1 dimension d 3c ⟩
⟨ Generation of grid vertices 4b ⟩
⟨ Transformation from multindex to address in a linear array storage 5 ⟩
⟨ Generation of grid cells 7 ⟩
⟨ Enumeration of binary ranges of given order 9a ⟩
⟨ Filtering binary ranges by order 9c ⟩
⟨ Assembling grid skeletons 10b ⟩
⟨ Multidimensional grid generation 12a ⟩
⟨ Multidimensional grid skeletons 13 ⟩
⟨ Generation of grid boundary complex 14a ⟩
⟨ Cartesian product of two lar models 19a ⟩
⟨ Simplicial face stack computation 14b ⟩
⟨ Identification of boundary vertices of a cuboidal complex 16a ⟩
⟨ Identification of cofaces of improper facets 17b ⟩

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

⟨ Multidimensional visualisation examples 12b ⟩

⟨ Test examples of Cartesian product 20a ⟩
◇

7 Unit tests

7.1 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

⟨ Create directory and echo of creation 21a ⟩ ≡
```
⟨ Create directory from path 23b ⟩
createDir('@1')
print "'@1' repository created"
◇
```
Macro never referenced.

"test/py/largrid/test01.py" 21b ≡
```
⟨ Create directory and echo of creation: (21c test/py/largrid/ ) ? ⟩
◇
```
File defined by 21bd.

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.

"test/py/largrid/test01.py" 21d ≡
```
from pyplasm import *

⟨ Generation of vertices of decompositions of 1D intervals 4a ⟩
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]]
◇
```
File defined by 21bd.

"test/py/largrid/test02.py" 21e ≡

```
import sys; sys.path.insert(0, 'lib/py/')
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)))
◇
```

8 Indices

The list of macros follow.

A Appendix

A.1 Utilities

⟨ Importing `simplexn` and `numpy` libraries 23a ⟩ ≡

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

◇

Macro referenced in 20b.

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

⟨ Create directory from path 23b ⟩ ≡

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

◇

Macro referenced in 21a, 23c.

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

`"test/py/largrid-tests.py"` 23c ≡

```
⟨ Create directory from path 23b ⟩
createDir('test/py/largrid/')
```

◇

A.2 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 **??**.

⟨ Import the module 23d ⟩ ≡

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

◇

Macro referenced in 20b, 24a.

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

⟨ Function to import a generic module 24a ⟩ ≡

```
def importModule(moduleName):
    ⟨ Import the module (24b moduleName ) 23d ⟩
```

◇

Macro never referenced.

References

[CL13] CVD-Lab, *Linear algebraic representation*, Tech. Report 13-00, Roma Tre University, October 2013.