The basic `larcc` module [*]

The LARCC team

May 30, 2014

Contents

[*]This document is part of the *Linear Algebraic Representation with CoChains* (LAR-CC) framework [CL13]. May 30, 2014

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).

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.

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

1.2 Format conversions

First we give the function `triples2mat` to make the transformation from the sparse matrix, given as a list of triples *row,column,value* (non-zero elements), 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.

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

```
def triples2mat(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)
◇
```

Macro referenced in 23a.

The function `brc2Coo` transforms a `BRC` representation in a list of triples (*row*, *column*, 1) ordered by row.

⟨ Brc to Coo transformation 3b ⟩ ≡

```
def brc2Coo(ListOfListOfInt):
    COOm = [[k,col,1] for k,row in enumerate(ListOfListOfInt)
            for col in row ]
    return COOm
◇
```

Macro referenced in 23a.

Two coordinate compressed sparse matrices `cooFV` and `cooEV` are created below, starting from the `BRC` representation `FV` and `EV` of the incidence of vertices on faces and edges, respectively, for a very simple plane triangulation.

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

```
print "\n>>> brc2Coo"
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 = brc2Coo(FV)
cooEV = brc2Coo(EV)
assert cooFV == [[0,0,1],[0,1,1],[0,3,1],[1,1,1],[1,2,1],[1,4,1],[2,1,1],
[2,3,1], [2,4,1],[3,2,1],[3,4,1],[3,5,1]]
assert cooEV == [[0,0,1],[0,1,1],[1,0,1],[1,3,1],[2,1,1],[2,2,1],[3,1,1],
[3,3,1],[4,1,1],[4,4,1],[5,2,1],[5,4,1],[6,2,1],[6,5,1],[7,3,1],[7,4,1],
[8,4,1],[8,5,1]]
◇
```

Macro referenced in 23b.

⟨ Coo to Csr transformation 4a ⟩ ≡

```
def coo2Csr(COOm):
    CSRm = triples2mat(COOm,"csr")
    return CSRm
◇
```

Macro referenced in 23a.

Two CSR sparse matrices `csrFV` and `csrEV` are generated (by *scipy.sparse*) in the following example:

⟨ Test example of Coo to Csr transformation 4b ⟩ ≡

```
csrFV = coo2Csr(cooFV)
csrEV = coo2Csr(cooEV)
print "\ncsr(FV) =\n", repr(csrFV)
print "\ncsr(EV) =\n", repr(csrEV)
◇
```

Macro referenced in 23b.

The *scipy* printout of the last two lines above is the following:

```
csr(FV) = <4x6 sparse matrix of type '<type 'numpy.int64'>'
   with 12 stored elements in Compressed Sparse Row format>
csr(EV) = <9x6 sparse matrix of type '<type 'numpy.int64'>'
   with 18 stored elements in Compressed Sparse Row format>
```

The transformation from BRC to CSR format is implemented slightly differently, according to the fact that the matrix dimension is either unknown (`shape=(0,0)`) or known.

⟨ Brc to Csr transformation 4c ⟩ ≡

```
def csrCreate(BRCmatrix,shape=(0,0)):
    triples = brc2Coo(BRCmatrix)
    if shape == (0,0):
        CSRmatrix = coo2Csr(triples)
    else:
        CSRmatrix = scipy.sparse.csr_matrix(shape)
        for i,j,v in triples: CSRmatrix[i,j] = v
    return CSRmatrix
◇
```

Macro referenced in 23a.

The conversion to CSR format of the characteristic matrix *faces-vertices* `FV` is given below for our simple example made by four triangle of a manifold 2D space, graphically shown in Figure 1a. The LAR representation with CSR matrices does not make difference between manifolds and non-manifolds, conversely than most modern solid modelling representation schemes, as shown by removing from `FV` the third triangle, giving the model in Figure 1b.

⟨ Test example of Brc to Csr transformation 5a ⟩ ≡

```
print "\n>>> brc2Csr"
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]]
csrFV = csrCreate(FV)
csrEV = csrCreate(EV)
print "\ncsrCreate(FV) =\n", csrFV
VIEW(STRUCT(MKPOLS((V,FV))))
VIEW(STRUCT(MKPOLS((V,EV))))
◇
```

Macro referenced in 6d, 23b.

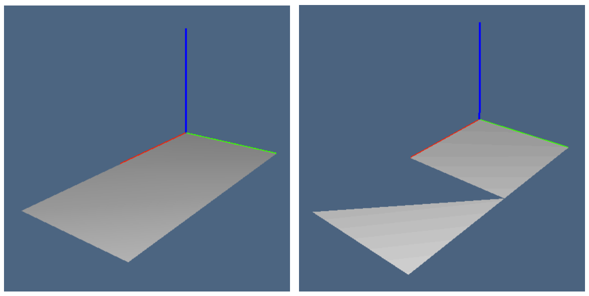

Figure 1: (a) Manifold two-dimensional space; (b) non-manifold space.

2 Matrix operations

As we know, the LAR representation of topology is based on CSR representation of sparse binary (and integer) matrices. Two Utility functions allow to query the number of rows and columns of a CSR matrix, independently from the low-level implementation (that in the following is provided by *scipy.sparse*).

⟨ Query Matrix shape 5b ⟩ ≡

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

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

Macro referenced in 23a.

⟨ Test examples of Query Matrix shape 6a ⟩ ≡

```
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)
◇
```

Macro referenced in 23b.

⟨ Sparse to dense matrix transformation 6b ⟩ ≡

```
def csr2DenseMatrix(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
◇
```

Macro referenced in 23a.

⟨ Test examples of Sparse to dense matrix transformation 6c ⟩ ≡

```
print "\n>>> csr2DenseMatrix"
print "\nFV =\n", csr2DenseMatrix(csrFV)
print "\nEV =\n", csr2DenseMatrix(csrEV)
◇
```

Macro referenced in 6d, 23b.

Characteristic matrices Let us compute and show in dense form the characteristic matrices of 2- and 1-cells of the simple manifold just defined. By running the file `test/py/larcc/test08.py` the reader will get the two matrices shown in Example 2

"`test/py/larcc/test08.py`" 6d ≡

```
import sys
sys.path.insert(0, 'lib/py/')
from larcc import *
⟨ Test example of Brc to Csr transformation 5a ⟩
⟨ Test examples of Sparse to dense matrix transformation 6c ⟩
◇
```

Example 1 (Dense Characteristic matrices)**.** *Let us notice that the two matrices below have the some numbers of columns (indexed by vertices of the cell decomposition). This very fact allows to multiply one matrix for the other transposed, and hence to compute the* *matrix form of linear operators between the spaces of cells of various dimensions.*

$FV =$
```
[[1 1 0 1 0 0]
 [0 1 1 0 1 0]
 [0 1 0 1 1 0]
 [0 0 1 0 1 1]]
```

$EV =$
```
[[1 1 0 0 0 0]
 [1 0 0 1 0 0]
 [0 1 1 0 0 0]
 [0 1 0 1 0 0]
 [0 1 0 0 1 0]
 [0 0 1 0 1 0]
 [0 0 1 0 0 1]
 [0 0 0 1 1 0]
 [0 0 0 0 1 1]]
```

Matrix product and transposition The following macro provides the IDE interface for the two main matrix operations required by LARCC, the binary product of compatible matrices and the unary transposition of matrices.

⟨ Matrix product and transposition 7 ⟩ ≡
```
def matrixProduct(CSRm1,CSRm2):
    CSRm = CSRm1 * CSRm2
    return CSRm

def csrTranspose(CSRm):
    CSRm = CSRm.T
    return CSRm
◇
```
Macro referenced in 23a.

Example 2 (Operators from edges to faces and vice-versa)**.** *As a general rule for operators between two spaces of chains of different dimensions supported by the* same *cellular complex, we use names made by two characters, whose first letter correspond to the target space, and whose second letter to the domain space. Hence* `FE` *must be read as the operator from edges to faces. Of course, since this use correspond to see the first letter as the space generated by rows, and the second letter as the space generated by columns. Notice that the element* (i, j) *of such matrices stores the number of vertices shared between the (row-)cell* i *and the* *(column-)cell j.*

$$FE = FV\ EV^{\top} = \begin{matrix}\texttt{[[2 2 1 2 1 0 0 1 0]} \\ \texttt{[1 0 2 1 2 2 1 1 1]} \\ \texttt{[1 1 1 2 2 1 0 2 1]} \\ \texttt{[0 0 1 0 1 2 2 1 2]]}\end{matrix} \qquad EF = EV\ FV^{\top} = \begin{matrix}\texttt{[[2 1 1 0]} \\ \texttt{[2 0 1 0]} \\ \texttt{[1 2 1 1]} \\ \texttt{[2 1 2 0]} \\ \texttt{[1 2 2 1]} \\ \texttt{[0 2 1 2]} \\ \texttt{[0 1 0 2]} \\ \texttt{[1 1 2 1]} \\ \texttt{[0 1 1 2]]}\end{matrix}$$

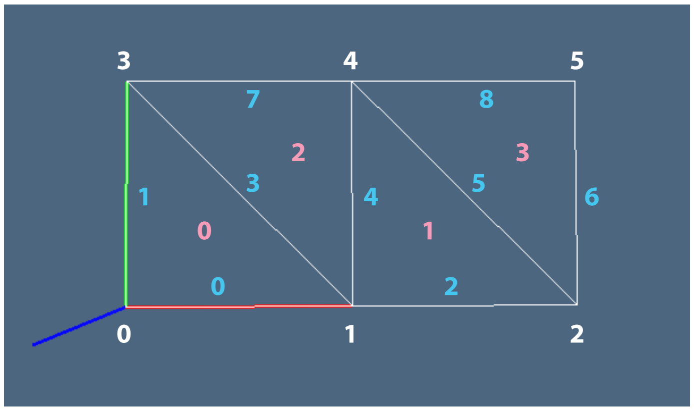

Figure 2: example caption

⟨Matrix filtering to produce the boundary matrix 8⟩ ≡

```
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
```
◇

Macro referenced in 23a.

⟨ Test example of Matrix filtering to produce the boundary matrix 9a ⟩ ≡

```
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", csr2DenseMatrix(CSRm)
◇
```

Macro referenced in 23b.

⟨ Matrix filtering via a generic predicate 9b ⟩ ≡

```
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
◇
```

Macro referenced in 23a.

⟨ Test example of Matrix filtering via a generic predicate 9c ⟩ ≡

```
print "\n>>> csrPredFilter"
CSRm = csrPredFilter(matrixProduct(csrFV, csrTranspose(csrEV)).T, GE(2)).T
print "\nccsrPredFilter(csrFE) =\n", csr2DenseMatrix(CSRm)
◇
```

Macro referenced in 23b.

3 Topological operations

In this section we provide the matrix representation of operators to compute the more important and useful topological operations on cellular complexes, and/or the indexed relations they return. We start the section by giving a graphical tool used to test the developed software, concerning the graphical writing of the full set of indices of the cells of every dimension in a 3D cuboidal complex.

Visualization of cell indices As already outlined, the `modelIndexing` function return the *hpc* value assembling both the 1-skeletons of the cells of every dimensions, and the graphical output of their indices, located on the centroid of each cell, and displayed using colors and sizes depending on the *rank* of the cell.

⟨ Visualization of cell indices 9d ⟩ ≡

```
""" Visualization of cell indices """
```

```
from sysml import *
def modelIndexing(shape):
   V, bases = larCuboids(shape,True)
   # bases = [[cell for cell in cellComplex if len(cell)==2**k] for k in range(4)]
   color = [YELLOW,CYAN,GREEN,WHITE]
   nums = AA(range)(AA(len)(bases))
   hpcs = []
   for k in range(4):
      hpcs += [SKEL_1(STRUCT(MKPOLS((V,bases[k]))))]
      hpcs += [cellNumbering((V,bases[k]),hpcs[2*k])(nums[k],color[k],0.3+0.2*k)]
   return STRUCT(hpcs)
◇
```

Macro defined by 9d, 10.
Macro referenced in 23a.

⟨ Visualization of cell indices 10 ⟩ ≡

```
""" Numbered visualization of a LAR model """
def larModelNumbering(V,bases,submodel):
   color = [YELLOW,CYAN,GREEN,WHITE]
   nums = AA(range)(AA(len)(bases))
   hpcs = [submodel]
   for k in range(len(bases)):
      hpcs += [cellNumbering((V,bases[k]),submodel)(nums[k],color[k],0.3+0.2*k)]
   return STRUCT(hpcs)
◇
```

Macro defined by 9d, 10.
Macro referenced in 23a.

3.1 Incidence and adjacency operators

Let us start by computing the more interesting subset of the binary relationships between the 4 decompositive and/or boundary entities of 3D cellular models. Therefore, in this case we denote with `C`, `F`, `E`, and `V`, the 3-cells and their faces, edges and vertices, respectively. The input is the full-fledged LAR representation provided by

$$\texttt{CV} := \texttt{CSR}(M_3) \quad (1)$$
$$\texttt{FV} := \texttt{CSR}(M_2) \quad (2)$$
$$\texttt{EV} := \texttt{CSR}(M_1) \quad (3)$$
$$\texttt{VV} := \texttt{CSR}(M_0) \quad (4)$$

Of course, $\texttt{CSR}(M_0)$ coincides with the identity matrix of dimension $|V|$ and can by excluded by further considerations. Some binary incidence and adjacency relations we are going to compute are:

$$\texttt{CF} := \texttt{CV} \times \texttt{FV}^t = \texttt{CSR}(M_3) \times \texttt{CSR}(M_2)^t \tag{5}$$
$$\texttt{CE} := \texttt{CV} \times \texttt{EV}^t = \texttt{CSR}(M_3) \times \texttt{CSR}(M_1)^t \tag{6}$$
$$\texttt{FE} := \texttt{FV} \times \texttt{EV}^t = \texttt{CSR}(M_2) \times \texttt{CSR}(M_1)^t \tag{7}$$

The other possible operators follow from a similer computational pattern.

The programming pattern for incidence computation A high-level function `larIncidence` useful to compute the LAR representation of the incidence matrix (operator) and the incidence relations is given in the script below.

⟨Some incidence operators 11a⟩ ≡
```
""" Some incidence operators """
def larIncidence(cells,facets):
   csrCellFacet = csrCellFaceIncidence(cells,facets)
   cooCellFacet = csrCellFacet.tocoo()
   larCellFacet = [[] for cell in range(len(cells))]
   for i,j,val in zip(cooCellFacet.row,cooCellFacet.col,cooCellFacet.data):
      if val == 1: larCellFacet[i] += [j]
   return larCellFacet
```
⟨Cell-Face incidence operator 11b⟩
⟨Cell-Edge incidence operator 12a⟩
⟨Face-Edge incidence operator 12b⟩
◇

Macro referenced in 23a.

Cell-Face incidence The `csrCellFaceIncidence` and `larCellFace` functions are given below, and exported to the `larcc` module.

⟨Cell-Face incidence operator 11b⟩ ≡
```
""" Cell-Face incidence operator """
def csrCellFaceIncidence(CV,FV):
   return boundary(FV,CV)

def larCellFace(CV,FV):
   return larIncidence(CV,FV)
```
◇

Macro referenced in 11a.

Cell-Edge incidence Analogously, the `csrCellEdgeIncidence` and `larCellFace` functions are given in the following script.

⟨ Cell-Edge incidence operator 12a ⟩ ≡
```
""" Cell-Edge incidence operator """
def csrCellEdgeIncidence(CV,EV):
    return boundary(EV,CV)

def larCellEdge(CV,EV):
   return larIncidence(CV,EV)
◇
```
Macro referenced in 11a.

Face-Edge incidence Finally, the `csrCellEdgeIncidence` and `larCellFace` functions are provided below.

⟨ Face-Edge incidence operator 12b ⟩ ≡
```
""" Face-Edge incidence operator """
def csrFaceEdgeIncidence(FV,EV):
   return boundary(EV,FV)

def larFaceEdge(FV,EV):
   return larIncidence(FV,EV)
◇
```
Macro referenced in 11a.

Example The example below concerns a 3D cuboidal grid, by computing a full LAR stack of bases `CV, FV, EV, VV`, showing its fully numbered 3D model, and finally by computing some more useful binary relationships (`CF, CE, FE`), needed for example to compute the signed matrices of boundary operators.

"test/py/larcc/test10.py" 12c ≡
```
""" A mesh model and various incidence operators """
import sys
sys.path.insert(0, 'lib/py/')
from larcc import *
from largrid import *

shape = [2,2,2]
V,(VV,EV,FV,CV) = larCuboids(shape,True)
"""
CV = [cell for cell in cellComplex if len(cell)==8]
FV = [cell for cell in cellComplex if len(cell)==4]
EV = [cell for cell in cellComplex if len(cell)==2]
VV = [cell for cell in cellComplex if len(cell)==1]
"""
VIEW(modelIndexing(shape))
```

```
CF = larCellFace(CV,FV)
CE = larCellFace(CV,EV)
FE = larCellFace(FV,EV)
◇
```

3.2 Boundary and coboundary operators

⟨From cells and facets to boundary operator 13a⟩ ≡

```
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)
◇
```

Macro referenced in 23a.

⟨Test examples of From cells and facets to boundary operator 13b⟩ ≡

```
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", csr2DenseMatrix(coboundary(CV,FV))
print "\ncoboundary_1 =\n", csr2DenseMatrix(coboundary(FV,EV))
print "\ncoboundary_0 =\n", csr2DenseMatrix(coboundary(EV,AA(LIST)(range(len(V)))))
◇
```

Macro referenced in 23b.

⟨ From cells and facets to boundary cells 14a ⟩ ≡
```
def zeroChain(cells):
   pass

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

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
◇
```
Macro referenced in 23a.

⟨ Test examples of From cells and facets to boundary cells 14b ⟩ ≡
```
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

boundaryModel = (V,[FV[k] for k in boundaryCells_2])
VIEW(EXPLODE(1.5,1.5,1.5)(MKPOLS(boundaryModel)))
◇
```
Macro referenced in 23b.

Signed boundary matrix for simplicial complexes The computation of the *signed* boundary matrix starts with enumerating the non-zero elements of the mod two (unoriented) boundary matrix. In particular, the `pairs` variable contains all the pairs of incident $((d-1)$-cell, d-cell), corresponding to all the 1 elements in the binary boundary matrix. Of course, their number equates the product of the number of d-cells, times the number of $(d-1)$-facets on the boundary of each d-cell. For the case of a 3-simplicial complex `CV`, we have $4|$`CV`$|$ `pairs` elements. The actual goal of the function `signedBoundary`, in the macro below, is to compute a sign for each of them.

The `pairs` values must be interpreted as (i, j) values in the incidence matrix `FC` (*facets-cells*), and hence as pairs of indices f and c into the characteristic matrices `FV` $=$ `CSR`(M_{d-1}) and `CV` $=$ `CSR`(M_d), respectively.

For each incidence pair `f,c`, the list `vertLists` contains the two lists of vertices associated to `f` and to `c`, called respectively the `face` and the `coface`. For each `face, coface` pair (i.e. for each unit element in the unordered boundary matrix), the `missingVertIndices` list will contain the index of the `coface` vertex not contained in the incident `face`. Finally the ± 1 (signed) incidence coefficients are computed and stored in the `faceSigns`, and then located in their actual positions within the `csrSignedBoundaryMat`. The sign of the incidence coefficient associated to the pair (facet,cell), also called (face,coface) in the implementation below, is computed as the sign of $(-1)^k$, where k is the position index of the removed vertex in the facet $\langle v_0, \ldots, v_{k-1}, v_{k+1}, \ldots, v_d \rangle$. of the $\langle v_0, \ldots, v_d \rangle$ cell.

$\langle$ Signed boundary matrix for simplicial models 15 $\rangle \equiv$

```
def signedBoundary (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[f], CV[c]] for f,c in pairs]

   # compute the local (interior to the coface) indices of missing vertices
   def missingVert(face,coface): return list(set(coface).difference(face))[0]
   missingVertIndices = [c.index(missingVert(f,c)) for f,c in vertLists]

   # signed incidence coefficients
   faceSigns = AA(C(POWER)(-1))(missingVertIndices)

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

Macro referenced in 23a.

Computation of signed boundary cells Two simplices are said *coherently oriented* when their common facets have opposite orientations. If the boundary cells give a decomposition of the boundary of an orientable solid, that partitionates the embedding space in two subsets corresponding to the *interior* and the *exterior* of the solid, then the boundary cells can be coherently oriented. This task is performed by the function `signedBoundaryCells` below.

The matrix of the signed boundary operator, with elements in $\{-1, 0, 1\}$, is computed in compressed sparse row (CSR) format, and stored in `csrSignedBoundaryMat`. In order to be able to return a list of `signedBoundaryCells` having a coherent orientation, we need to compute the coface of each boundary facet, i.e. the single d-cell having the facet on its boundary, and provide a coherent orientation to such chain of d-cells. The goal is obtained computing the sign of the determinant of the coface matrices, i.e. of square matrices having as rows the vertices of a coface, in normalised homogeneous coordinates.

The chain of boundary facets `boundaryCells`, obtained by multiplying the signed matrix of the boundary operator by the coordinate representation of the total d-chain, is coherently oriented by multiplication times the determinants of the `cofaceMats`.

The `cofaceMats` list is filled with the matrices having per row the position vectors of vertices of a coface, in normalized homogeneous coordinates. The list of signed face indices `orientedBoundaryCells` is returned by the function.

⟨ Oriented boundary cells for simplicial models 16 ⟩ ≡

```
def signedBoundaryCells(verts,cells,facets):
   csrSignedBoundaryMat = signedBoundary(cells,facets)

   csrTotalChain = totalChain(cells)
   csrBoundaryChain = matrixProduct(csrSignedBoundaryMat, csrTotalChain)
   cooCells = csrBoundaryChain.tocoo()

   boundaryCells = []
   for k,v in enumerate(cooCells.data):
      if abs(v) == 1:
         boundaryCells += [int(cooCells.row[k] * cooCells.data[k])]

   boundaryCocells = []
   for k,v in enumerate(boundaryCells):
      boundaryCocells += list(csrSignedBoundaryMat[abs(v)].tocoo().col)

   boundaryCofaceMats = [[verts[v]+[1] for v in cells[c]] for c in boundaryCocells]
   boundaryCofaceSigns = AA(SIGN)(AA(np.linalg.det)(boundaryCofaceMats))

   def swap(mylist): return [mylist[1]]+[mylist[0]]+mylist[2:]
   orientedBoundaryCells = list(array(boundaryCells)*array(boundaryCofaceSigns))

   return orientedBoundaryCells
◇
```

Macro defined by 16, 18.
Macro referenced in 23a.

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

⟨ Oriented boundary cells for simplicial models 18 ⟩ ≡

```
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]
◇
```

Macro defined by 16, 18.
Macro referenced in 23a.

⟨ Computation of cell adjacencies 19a ⟩ ≡

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

Macro referenced in 23a.

⟨ Test examples of Computation of cell adjacencies 19b ⟩ ≡

```
print "\n>>> larCellAdjacencies"
adj_2_cells = larCellAdjacencies(csrFV)
print "\nadj_2_cells =\n", csr2DenseMatrix(adj_2_cells)
adj_1_cells = larCellAdjacencies(csrEV)
print "\nadj_1_cells =\n", csr2DenseMatrix(adj_1_cells)
◇
```

Macro referenced in 23b.

⟨ Extraction of facets of a cell complex 20 ⟩ ≡

```
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,emptyCellNumber=0):
    """
        Estraction of (d-1)-cellFacets from "model" := (V,d-cells)
        Return (V, (d-1)-cellFacets)
      """
    V,cells,csr,csrAdjSquareMat = setup(model,dim)
    solidCellNumber = len(cells) - emptyCellNumber
    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) and (i<solidCellNumber):
                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
◇
```

Macro referenced in 23a.

⟨ Test examples of Extraction of facets of a cell complex 21 ⟩ ≡

```
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
◇
```

Macro referenced in 23b.

4 Exporting the library

4.1 MIT licence

⟨The MIT Licence 22a⟩ ≡

```
"""
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.
"""
◇
```

Macro referenced in 23a.

4.2 Importing of modules or packages

⟨Importing of modules or packages 22b⟩ ≡

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

from lar2psm import *
◇
```

Macro referenced in 23a.

4.3 Writing the library file

"lib/py/larcc.py" 23a ≡

```
# -*- coding: utf-8 -*-
""" Basic LARCC library """
⟨The MIT Licence 22a⟩
⟨Importing of modules or packages 22b⟩
⟨From list of triples to scipy.sparse 3a⟩
⟨Brc to Coo transformation 3b⟩
⟨Coo to Csr transformation 4a⟩
⟨Brc to Csr transformation 4c⟩
⟨Query Matrix shape 5b⟩
⟨Sparse to dense matrix transformation 6b⟩
⟨Matrix product and transposition 7⟩
⟨Matrix filtering to produce the boundary matrix 8⟩
⟨Matrix filtering via a generic predicate 9b⟩
⟨From cells and facets to boundary operator 13a⟩
⟨From cells and facets to boundary cells 14a⟩
⟨Signed boundary matrix for simplicial models 15⟩
⟨Oriented boundary cells for simplicial models 16, ... ⟩
⟨Computation of cell adjacencies 19a⟩
⟨Extraction of facets of a cell complex 20⟩
⟨Some incidence operators 11a⟩
⟨Visualization of cell indices 9d, ... ⟩
⟨Numbered visualization of a LAR model ?⟩

if __name__ == "__main__":
    ⟨Test examples 23b⟩
◇
```

5 Unit tests

⟨Test examples 23b⟩ ≡

```
⟨Test example of Brc to Coo transformation 3c⟩
⟨Test example of Coo to Csr transformation 4b⟩
⟨Test example of Brc to Csr transformation 5a⟩
⟨Test examples of Query Matrix shape 6a⟩
⟨Test examples of Sparse to dense matrix transformation 6c⟩
⟨Test example of Matrix filtering to produce the boundary matrix 9a⟩
⟨Test example of Matrix filtering via a generic predicate 9c⟩
⟨Test examples of From cells and facets to boundary operator 13b⟩
⟨Test examples of From cells and facets to boundary cells 14b⟩
⟨Test examples of Computation of cell adjacencies 19b⟩
⟨Test examples of Extraction of facets of a cell complex 21⟩
◇
```

Macro referenced in 23a.

Comparing oriented and unoriented boundary

"test/py/larcc/test09.py" 24a ≡

```
""" comparing oriented boundary and unoriented boundary extraction on a simple example """
import sys
sys.path.insert(0, 'lib/py/')
from largrid import *
from larcc import *

V,CV = larSimplexGrid1([1,1,1])
FV = larSimplexFacets(CV)

orientedBoundary = signedBoundaryCells(V,CV,FV)
def swap(mylist): return [mylist[1]]+[mylist[0]]+mylist[2:]
orientedBoundaryFV = [FV[-k] if k<0 else swap(FV[k]) for k in orientedBoundary]
VIEW(EXPLODE(1.5,1.5,1.5)(MKPOLS((V,orientedBoundaryFV))))

BF = boundaryCells(CV,FV)
boundaryCellsFV = [FV[k] for k in BF]
VIEW(EXPLODE(1.5,1.5,1.5)(MKPOLS((V,boundaryCellsFV))))
◇
```

A Appendix: Tutorials

A.1 Model generation, skeleton and boundary extraction

"test/py/larcc/test01.py" 24b ≡

```
import sys
sys.path.insert(0, 'lib/py/')
from larcc import *
from largrid import *
⟨input of 2D topology and geometry data 25a⟩
⟨characteristic matrices 25b⟩
⟨incidence matrix 25c⟩
⟨boundary and coboundary operators 25d⟩
⟨product of cell complexes 26a⟩
⟨2-skeleton extraction 26b⟩
⟨1-skeleton extraction 26c⟩
⟨0-coboundary computation 27a⟩
⟨1-coboundary computation 27b⟩
⟨2-coboundary computation 27c⟩
⟨boundary chain visualisation 27d⟩
◇
```

⟨ input of 2D topology and geometry data 25a ⟩ ≡

```
# input of geometry and topology
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]]
◇
```

Macro referenced in 24b.

⟨ characteristic matrices 25b ⟩ ≡

```
# characteristic matrices
csrFV = csrCreate(FV)
csrEV = csrCreate(EV)
print "\nFV =\n", csr2DenseMatrix(csrFV)
print "\nEV =\n", csr2DenseMatrix(csrEV)
◇
```

Macro referenced in 24b.

⟨ incidence matrix 25c ⟩ ≡

```
# product
csrEF = matrixProduct(csrEV, csrTranspose(csrFV))
print "\nEF =\n", csr2DenseMatrix(csrEF)
◇
```

Macro referenced in 24b.

⟨ boundary and coboundary operators 25d ⟩ ≡

```
# boundary and coboundary operators
facetLengths = [csrCell.getnnz() for csrCell in csrEV]
boundary = csrBoundaryFilter(csrEF,facetLengths)
coboundary_1 = csrTranspose(boundary)
print "\ncoboundary_1 =\n", csr2DenseMatrix(coboundary_1)
◇
```

Macro referenced in 24b.

⟨ product of cell complexes 26a ⟩ ≡

```
# 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"
◇
```

Macro referenced in 24b.

⟨ 2-skeleton extraction 26b ⟩ ≡

```
# 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"
◇
```

Macro referenced in 24b.

⟨ 1-skeleton extraction 26c ⟩ ≡

```
# 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"
◇
```

Macro referenced in 24b.

⟨ 0-coboundary computation 27a ⟩ ≡

```
# 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", csr2DenseMatrix(coboundary_0)
◇
```

Macro referenced in 24b.

⟨ 1-coboundary computation 27b ⟩ ≡

```
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", csr2DenseMatrix(coboundary_1.T)
◇
```

Macro referenced in 24b.

⟨ 2-coboundary computation 27c ⟩ ≡

```
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", csr2DenseMatrix(coboundary_2)
◇
```

Macro referenced in 24b.

⟨ boundary chain visualisation 27d ⟩ ≡

```
# 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)))
◇
```

Macro referenced in 24b.

A.2 Boundary of 3D simplicial grid

"test/py/larcc/test02.py" 28a ≡

```
import sys
sys.path.insert(0, 'lib/py/')
⟨ boundary of 3D simplicial grid 28b ⟩
◇
```

⟨ boundary of 3D simplicial grid 28b ⟩ ≡

```
from simplexn import *
from larcc import *

V,CV = larSimplexGrid1([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

boundaryCells_2 = signedBoundaryCells(V,CV,FV)
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]

VIEW(EXPLODE(1.5,1.5,1.5)(MKPOLS((V,boundaryFV))))
print "\nboundaryCells_2 =\n", boundaryFV
◇
```

Macro referenced in 28a.

A.3 Oriented boundary of a random simplicial complex

"test/py/larcc/test03.py" 28c ≡

```
⟨ Importing external modules 29a ⟩
⟨ Generating and viewing a random 3D simplicial complex 29b ⟩
⟨ Computing and viewing its non-oriented boundary 29c ⟩
⟨ Computing and viewing its oriented boundary 29d ⟩
◇
```

⟨ Importing external modules 29a ⟩ ≡

```
import sys
sys.path.insert(0, 'lib/py/')
from simplexn import *
from larcc import *
from scipy import *
from scipy.spatial import Delaunay
import numpy as np
◇
```

Macro referenced in 28c.

⟨ Generating and viewing a random 3D simplicial complex 29b ⟩ ≡

```
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),[]]))
◇
```

Macro referenced in 28c.

⟨ Computing and viewing its non-oriented boundary 29c ⟩ ≡

```
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)))
◇
```

Macro referenced in 28c.

⟨ Computing and viewing its oriented boundary 29d ⟩ ≡

```
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]
boundaryModel = (V,boundaryFV)
VIEW(EXPLODE(1.5,1.5,1.5)(MKPOLS(boundaryModel)))
◇
```

Macro referenced in 28c.

A.4 Oriented boundary of a simplicial grid

"test/py/larcc/test04.py" 30a ≡

```
⟨ Generate and view a 3D simplicial grid 30b ⟩
⟨ Computing and viewing the 2-skeleton of simplicial grid 30c ⟩
⟨ Computing and viewing the oriented boundary of simplicial grid 30d ⟩
◇
```

⟨ Generate and view a 3D simplicial grid 30b ⟩ ≡

```
import sys
sys.path.insert(0, 'lib/py/')
from simplexn import *
from larcc import *
V,CV = larSimplexGrid1([4,4,4])
VIEW(EXPLODE(1.5,1.5,1.5)(MKPOLS((V,CV))))
◇
```

Macro referenced in 30a.

⟨ Computing and viewing the 2-skeleton of simplicial grid 30c ⟩ ≡

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

Macro referenced in 30a.

⟨ Computing and viewing the oriented boundary of simplicial grid 30d ⟩ ≡

```
csrSignedBoundaryMat = signedBoundary (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)))
◇
```

Macro referenced in 30a.

A.5 Skeletons and oriented boundary of a simplicial complex

"test/py/larcc/test05.py" 30e ≡

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

⟨ Skeletons computation and vilualisation 31a ⟩
⟨ Oriented boundary matrix visualization 31b ⟩
⟨ Computation of oriented boundary cells 31c ⟩
◇
```

⟨ Skeletons computation and vilualisation 31a ⟩ ≡

```
from simplexn import *
from larcc import *
V,FV = larSimplexGrid1([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))))
◇
```

Macro referenced in 30e.

⟨ Oriented boundary matrix visualization 31b ⟩ ≡

```
np.set_printoptions(threshold='nan')
csrSignedBoundaryMat = signedBoundary (FV,EV)
Z = csr2DenseMatrix(csrSignedBoundaryMat)
print "\ncsrSignedBoundaryMat =\n", Z
from pylab import *
matshow(Z)
show()
◇
```

Macro referenced in 30e.

⟨ Computation of oriented boundary cells 31c ⟩ ≡

```
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)))
◇
```

Macro referenced in 30e.

A.6 Boundary of random 2D simplicial complex

"test/py/larcc/test06.py" 31d ≡

```
import sys
sys.path.insert(0, 'lib/py/')
from simplexn import *
from larcc import *
from scipy.spatial import Delaunay
⟨ Test for quasi-equilateral triangles 32a ⟩
⟨ Generation and selection of random triangles 32b ⟩
⟨ Boundary computation and visualisation 33a ⟩
◇
```

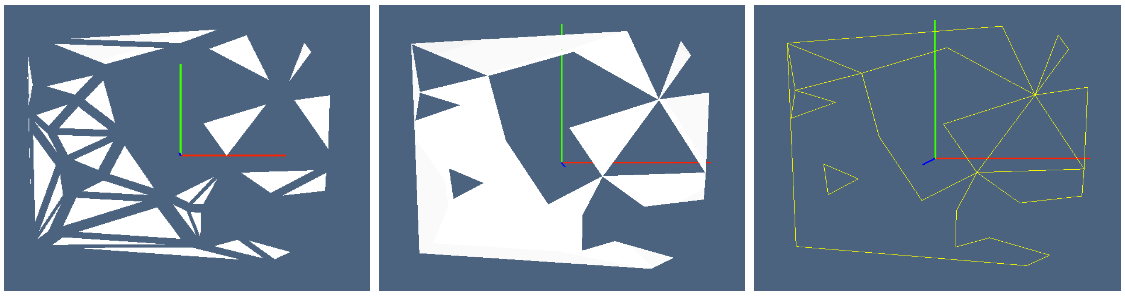

Figure 3: example caption

⟨ Test for quasi-equilateral triangles 32a ⟩ ≡

```
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
◇
```

Macro referenced in 31d.

⟨ Generation and selection of random triangles 32b ⟩ ≡

```
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))
◇
```

Macro referenced in 31d.

⟨Boundary computation and visualisation 33a⟩ ≡

```
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))))
◇
```

Macro referenced in 31d.

⟨Compute the topologically ordered chain of boundary vertices 33b⟩ ≡

```

◇
```

Macro never referenced.

⟨Decompose a permutation into cycles 33c⟩ ≡

```
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])
◇
```

Macro never referenced.

A.7 Assemblies of simplices and hypercubes

"test/py/larcc/test07.py" 34a ≡

```
import sys
sys.path.insert(0, 'lib/py/')
from simplexn import *
from larcc import *
from largrid import *
⟨ Definition of 1-dimensional LAR models 34b ⟩
⟨ Assembly generation of squares and triangles 34c ⟩
⟨ Assembly generation of cubes and tetrahedra 35 ⟩
◇
```

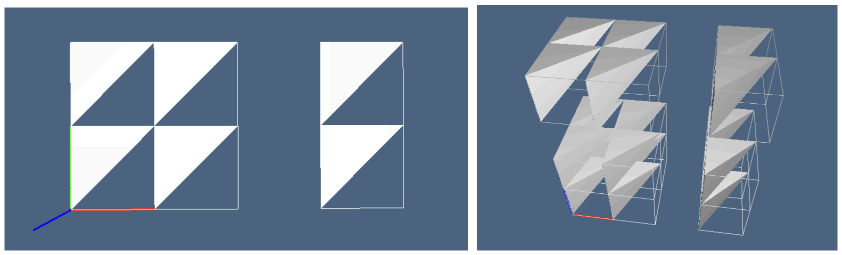
Figure 4: (a) Assemblies of squares and triangles; (b) assembly of cubes and tetrahedra.

⟨ Definition of 1-dimensional LAR models 34b ⟩ ≡

```
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)
◇
```

Macro referenced in 34a.

⟨ Assembly generation of squares and triangles 34c ⟩ ≡

```
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)))) ]))
◇
```

Macro referenced in 34a.

⟨ Assembly generation of cubes and tetrahedra 35 ⟩ ≡

```
from largrid import *
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)))) ]))
◇
```

Macro referenced in 34a.

References

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