The basic `larcc` module *

The LARCC team

January 10, 2014

Contents

*This document is part of the *Linear Algebraic Representation with CoChains* (LAR-CC) framework [?]. January 10, 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 $\mathtt{BRC}(A)$ is used for the corresponding data structure.

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

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. @d From list of triples to scipy.sparse @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$_{m}atrix((data,ij)).asformat(shape)@The function$ `brc2Coo` $transforms a$ `BRC` $representati$

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. @d Test example of Brc to Coo transformation @print "¿¿¿ 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]] @ @d Coo to Csr transformation @def coo2Csr(COOm): CSRm = triples2mat(COOm,"csr") return CSRm @

Two CSR sparse matrices `csrFV` and `csrEV` are generated (by *scipy.sparse*) in the following example: @d Test example of Coo to Csr transformation @csrFV = coo2Csr(cooFV) csrEV = coo2Csr(cooEV) print "(FV) =", repr(csrFV) print "(EV) =", repr(csrEV) @ 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. @d Brc to Csr transformation @def csrCreate(BRCmatrix,shape=(0,0)): triples = brc2Coo(BRCmatrix) if shape == (0,0): CSRmatrix = coo2Csr(triples) else: CSRmatrix = scipy.sparse.csr$_matrix(shape)fori,j,vint$ $CSRmatrix[i,j] = vreturnCSRmatrix@TheconversiontoCSRformatofthecharacteristicmatrixfaces\text{-}verti$ $manifolds, converselythanmostmodernsolidmodellingrepresentationschemes, asshownbyremovingfrom$ `FV`

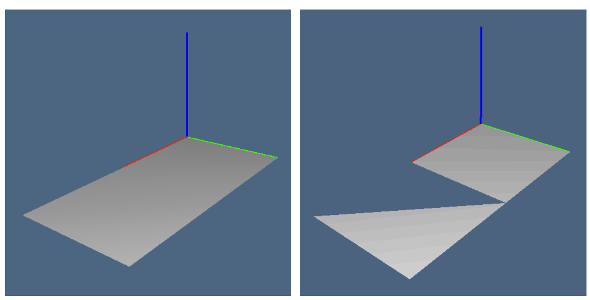

Figure 1: (a) Manifold two-dimensional space; (b) non-manifold space.

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*). @d Query Matrix shape @def csrGetNumberOfRows(CSRmatrix): Int = CSRmatrix.shape[0] return Int

def csrGetNumberOfColumns(CSRmatrix): Int = CSRmatrix.shape[1] return Int @ @d Test examples of Query Matrix shape @print "¿¿¿ csrGetNumberOfRows" print "(csrFV)

=", csrGetNumberOfRows(csrFV) print "(csrEV) =", csrGetNumberOfRows(csrEV) print "¿¿¿ csrGetNumberOfColumns" print "(csrFV) =", csrGetNumberOfColumns(csrFV) print "(csrEV) =", csrGetNumberOfColumns(csrEV) @

@d Sparse to dense matrix transformation @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 @ @d Test examples of Sparse to dense matrix transformation @print "¿¿¿ csr2DenseMatrix" print "=", csr2DenseMatrix(csrFV) print "=", csr2DenseMatrix(csrEV) @

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/ex8.py` the reader will get the two matrices shown in Example **??** @o test/py/larcc/ex8.py @¡ Test example of Brc to Csr transformation @¿ @¡ Test examples of Sparse to dense matrix transformation @¿ @

Example 1 (Dense Characteristic matrices)**.** *ex:denseMat aaaa*

@d Matrix product and transposition @def matrixProduct(CSRm1,CSRm2): CSRm = CSRm1 * CSRm2 return CSRm

def csrTranspose(CSRm): CSRm = CSRm.T return CSRm @ @d Matrix filtering to produce the boundary matrix @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$_matrix((coo.data, (coo.row, coo.col)), shape = inputShape)out = mtx.tocsr()returnout@@dTestexampleofMatrixfilteringtoproducetheboundarymatrix@print" >>> csrBo$

3 Topological operations

@d From cells and facets to boundary operator @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) @ @d Test examples of From cells and facets to boundary operator @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 "$_2 = ", csr2DenseMatrix(coboundary(CV, FV))print"_1 = ", csr2DenseMatrix(coboundary(FV, EV$ ", $csr2DenseMatrix(coboundary(EV, AA(LIST)(range(len(V)))))@@dFromcellsandfacetstoboundarycells$

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

def boundaryCells(cells,facets): csrBoundaryMat = boundary(cells,facets) csrChain = totalChain(cells) csrBoundaryChain = matrixProduct(csrBoundaryMat, csrChain) for k,value in enumerate(csrBoundaryChain.data): if value boundaryCells = [k for k,val in enumerate(csrBoundaryChain.data.tolist()) if val == 1] return boundaryCells @ @d Test examples of From cells and facets to boundary cells @boundaryCells$_2 = boundaryCells(CV, FV)boundaryCells_1$ $boundaryCells([FV[k]forkinboundaryCells_2], EV)$

print "$_2 = ", boundaryCells_2print"_1 = ", boundaryCells_1$

boundary = (V,[FV[k] for k in boundaryCells$_2$]$)VIEW(EXPLODE(1.5, 1.5, 1.5)(MKPOLS(boundary)))$

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

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

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

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

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

signed boundary matrix csrSignedBoundaryMat = csr$_matrix((signPairProd, TRANS(pairs)))returncsrS$

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

@d Oriented boundary cells for simplicial models @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] @ @d Computation of cell adjacencies @def larCellAdjacencies(CSRm): CSRm = matrixProduct(CSRm,csrTranspose(CSRm)) return CSRm @ @d Test examples of Computation of cell adjacencies @print "¿¿¿ larCellAdjacencies" adj$_{2c}ells = larCellAdjacencies(csrFV)print"_{2c}ells = ", csr2DenseMatrix(adj_{2c}ells)adj_{1c}ells =$ $larCellAdjacencies(csrEV)print"_{1c}ells = ", csr2DenseMatrix(adj_{1c}ells)@@dExtractionoffacetsofacellcom$

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

$_, EV = larFacets((V, FV), dim = 2)print" = ", EVVIEW(EXPLODE(1.5, 1.5, 1.5)(MKPOLS((V, EV$

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

$_, EV = larFacets((V, FV), dim = 2)print" = ", EV@$

4 Exporting the library

4.1 MIT licence

@d The MIT Licence @ """ 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. """ @

4.2 Importing of modules or packages

@d Importing of modules or packages @from pyplasm import * import collections import scipy import numpy as np from scipy import zeros,arange,mat,amin,amax from scipy.sparse import vstack,hstack,csr$_{m}atrix, coo_{m}atrix, lil_{m}atrix, triu$

from lar2psm import * @

4.3 Writing the library file

@o lib/py/larcc.py @ -*- coding: utf-8 -*- """ Basic LARCC library """ @¡ The MIT Licence @¿ @¡ Importing of modules or packages @¿ @¡ From list of triples to scipy.sparse @¿ @¡ Brc to Coo transformation @¿ @¡ Coo to Csr transformation @¿ @¡ Brc to Csr transformation @¿ @¡ Query Matrix shape @¿ @¡ Sparse to dense matrix transformation @¿ @¡ Matrix product and transposition @¿ @¡ Matrix filtering to produce the boundary matrix @¿ @¡ Matrix filtering via a generic predicate @¿ @¡ From cells and facets to boundary operator @¿ @¡ From cells and facets to boundary cells @¿ @¡ Signed boundary matrix for simplicial models @¿ @¡ Oriented boundary cells for simplicial models @¿ @¡ Computation of cell adjacencies @¿ @¡ Extraction of facets of a cell complex @¿

if $_{name_{=="}main_{",:@<Testexamples@>@}}$

5 Unit tests

@d Test examples @ @¡ Test example of Brc to Coo transformation @¿ @¡ Test example of Coo to Csr transformation @¿ @¡ Test example of Brc to Csr transformation @¿ @¡ Test examples of Query Matrix shape @¿ @¡ Test examples of Sparse to dense matrix transformation @¿ @¡ Test example of Matrix filtering to produce the boundary matrix @¿ @¡ Test example of Matrix filtering via a generic predicate @¿ @¡ Test examples of From cells and facets to boundary operator @¿ @¡ Test examples of From cells and facets to boundary cells @¿ @¡ Test examples of Computation of cell adjacencies @¿ @¡ Test examples of Extraction of facets of a cell complex @¿ @

A Appendix: Tutorials

A.1 Model generation, skeleton and boundary extraction

@o test/py/larcc/ex1.py @ from larcc import * from largrid import * @¡ input of 2D topology and geometry data @¿ @¡ characteristic matrices @¿ @¡ incidence matrix @¿ @¡ boundary and coboundary operators @¿ @¡ product of cell complexes @¿ @¡ 2-skeleton extraction @¿ @¡ 1-skeleton extraction @¿ @¡ 0-coboundary computation @¿ @¡ 1-coboundary computation @¿ @¡ 2-coboundary computation @¿ @¡ boundary chain visualisation @¿ @

@d input of 2D topology and geometry data @ 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]] @

@d characteristic matrices @ characteristic matrices csrFV = csrCreate(FV) csrEV = csrCreate(EV) print "=", csr2DenseMatrix(csrFV) print "=", csr2DenseMatrix(csrEV) @

@d incidence matrix @ product csrEF = matrixProduct(csrEV, csrTranspose(csrFV)) print "=", csr2DenseMatrix(csrEF) @

@d boundary and coboundary operators @ boundary and coboundary operators facetLengths = [csrCell.getnnz() for csrCell in csrEV] boundary = csrBoundaryFilter(csrEF,facetLengths) coboundary$_1 = csrTranspose(boundary) print"_1 = ", csr2DenseMatrix(coboundary_1)$@

@d product of cell complexes @ product operator mod$_2D = (V2, FV) V1, topol_0 = [[0.], [1.], [2.]], [[0], [1], [2]] topol_1 = [[0, 1], [1, 2]] mod_0D = (V1, topol_0) mod_1D = (V1, topol_1) V3, CV = larModelProduct([mod_2D, mod_1D]) mod_3D = (V3, CV) VIEW(EXPLODE(1.2, 1.2, 1.2)(MKPOLS(mod_3D$
", $len(CV),$""@

@d 2-skeleton extraction @ 2-skeleton of the 3D product complex mod$_2D_1 = (V2, EV) mod_3D_h 2 = larModelProduct([mod_2D, mod_0D]) mod_3D_v 2 = larModelProduct([mod_2D_1, mod_1D]), FV_h = mod_3D_h 2, FV_v = mod_3D_v 2 FV3 = FV_h + FV_v SK2 = (V3, FV3) VIEW(EXPLODE(1.2, 1.2, 1.2)(MKPOLS($
", $len(FV3),$""@

@d 1-skeleton extraction @ 1-skeleton of the 3D product complex mod$_2D_0 = (V2, AA(LIST)(range(len(V2$
$larModelProduct([mod_2D_1, mod_0D]) mod_3D_v 1 = larModelProduct([mod_2D_0, mod_1D]), EV_h =$

$mod_3D_h1,EV_v = mod_3D_v1EV3 = EV_h+EV_vSK1 = (V3, EV3)VIEW(EXPLODE(1.2, 1.2, 1.2)(MKPOLS$
", $len(EV3),$ ""@

@d 0-coboundary computation @ boundary and coboundary operators np.set$_printoptions(threshold = sys.maxint)csrFV3 = csrCreate(FV3)csrEV3 = csrCreate(EV3)csrVE3 = csrTranspose(csrEV3)facetL$
$[csrCell.getnnz()forcsrCellincsrEV3]boundary = csrBoundaryFilter(csrVE3, facetLengths)coboundary_0$
$csrTranspose(boundary)print"_0 = ", csr2DenseMatrix(coboundary_0)$@

@d 1-coboundary computation @csrEF3 = matrixProduct(csrEV3, csrTranspose(csrFV3)) facetLengths = [csrCell.getnnz() for csrCell in csrFV3] boundary = csrBoundaryFilter(csrEF3,facetLengths) coboundary$_1$ = $csrTranspose(boundary)print"_1.T = ", csr2DenseMatrix(coboundary_1.T)$@

@d 2-coboundary computation @csrCV = csrCreate(CV) csrFC3 = matrixProduct(csrFV3, csrTranspose(csrCV)) facetLengths = [csrCell.getnnz() for csrCell in csrCV] boundary = csrBoundaryFilter(csrFC3,facetLengths) coboundary$_2$ = $csrTranspose(boundary)print"_2 =$
", $csr2DenseMatrix(coboundary_2)$@

@d boundary chain visualisation @ boundary chain visualisation boundaryCells$_2$ = $boundaryCells(CV, FV3)boundary = (V3, [FV3[k]forkinboundaryCells_2])VIEW(EXPLODE(1.5, 1.5, 1.5)$

A.2 Boundary of 3D simplicial grid

@o test/py/larcc/ex2.py @ @¡ boundary of 3D simplicial grid @¿ @

@d boundary of 3D simplicial grid @from simplexn import * from larcc import *

V,CV = larSimplexGrid([10,10,3]) VIEW(EXPLODE(1.5,1.5,1.5)(MKPOLS((V,CV)))) SK2 = (V,larSimplexFacets(CV)) VIEW(EXPLODE(1.5,1.5,1.5)(MKPOLS(SK2))) $_,FV = SK2SK1 = (V, larSimplexFacets(FV))_,EV = SK1VIEW(EXPLODE(1.5, 1.5, 1.5)(MKPOLS(SK1)))$

boundaryCells$_2$ = $boundaryCells(CV, FV)boundary = (V, [FV[k]forkinboundaryCells_2])VIEW(EXPL$
", $boundaryCells_2$@

A.3 Oriented boundary of a random simplicial complex

@o test/py/larcc/ex3.py @@¡ Importing external modules @¿ @¡ Generating and viewing a random 3D simplicial complex @¿ @¡ Computing and viewing its non-oriented boundary @¿ @¡ Computing and viewing its oriented boundary @¿ @

@d Importing external modules @from simplexn import * from larcc import * from scipy.spatial import Delaunay import numpy as np @

@d Generating and viewing a random 3D simplicial complex @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),[]])) @

@d Computing and viewing its non-oriented boundary @FV = larSimplexFacets(CV) VIEW(MKPOL([V,AA(AA(lambda k:k+1))(FV),[]])) boundaryCells$_2$ = $boundaryCells(CV, FV)print"_2 =$

", $boundaryCells_2 bndry = (V, [FV[k] for k in boundaryCells_2]) VIEW(EXPLODE(1.5, 1.5, 1.5)(MKPOLS(b$

@d Computing and viewing its oriented boundary @boundaryCells$_2$ = $signedBoundaryCells(V, CV, FV) pr$
", $boundaryCells_2 def swap(mylist) : return [mylist[1]]+[mylist[0]]+mylist[2 :] boundaryFV =$
$[FV[-k] if k < 0 else swap(FV[k]) for k in boundaryCells_2] bndry = (V, boundaryFV) VIEW(EXPLODE(1.5,$

A.4 Oriented boundary of a simplicial grid

@o test/py/larcc/ex4.py @@¡ Generate and view a 3D simplicial grid @¿ @¡ Computing and viewing the 2-skeleton of simplicial grid @¿ @¡ Computing and viewing the oriented boundary of simplicial grid @¿ @

@d Generate and view a 3D simplicial grid @from simplexn import * from larcc import * V,CV = larSimplexGrid([4,4,4]) VIEW(EXPLODE(1.5,1.5,1.5)(MKPOLS((V,CV)))) @

@d Computing and viewing the 2-skeleton of simplicial grid @FV = larSimplexFacets(CV) EV = larSimplexFacets(FV) VIEW(EXPLODE(1.5,1.5,1.5)(MKPOLS((V,FV)))) @

@d Computing and viewing the oriented boundary of simplicial grid @csrSignedBoundaryMat = signedBoundary (V,CV,FV) boundaryCells$_2$ = $signedBoundaryCells(V, CV, FV) def swap(l) : return [l[1], l[0], l[2]] boundaryFV = [FV[-k] if k < 0 else swap(FV[k]) for k in boundaryCells_2] boundary = (V, boundaryFV) VIEW(EXPLODE(1.5, 1.5, 1.5)(MKPOLS(boundary)))$@

A.5 Skeletons and oriented boundary of a simplicial complex

@o test/py/larcc/ex5.py @@¡ Skeletons computation and vilualisation @¿ @¡ Oriented boundary matrix visualization @¿ @¡ Computation of oriented boundary cells @¿ @

@d Skeletons computation and vilualisation @from simplexn import * from larcc import * V,FV = larSimplexGrid([3,3]) VIEW(EXPLODE(1.5,1.5,1.5)(MKPOLS((V,FV)))) EV = larSimplexFacets(FV) VIEW(EXPLODE(1.5,1.5,1.5)(MKPOLS((V,EV)))) VV = larSimplexFacets(EV) VIEW(EXPLODE(1.5,1.5,1.5)(MKPOLS((V,VV)))) @

@d Oriented boundary matrix visualization @np.set$_p rintoptions(threshold =' nan') csrSignedBoundaryM$
$signedBoundary(V, FV, EV) Z = csr2DenseMatrix(csrSignedBoundaryMat) print" =$
", $Z from pylab import * matshow(Z) show()$@

@d Computation of oriented boundary cells @boundaryCells$_1$ = $signedBoundaryCells(V, FV, EV) print"_1$
", $boundaryCells_1 def swap(mylist) : return [mylist[1]]+[mylist[0]]+mylist[2 :] boundaryEV =$
$[EV[-k] if k < 0 else swap(EV[k]) for k in boundaryCells_1] bndry = (V, boundaryEV) VIEW(EXPLODE(1.5,$

A.6 Boundary of random 2D simplicial complex

@o test/py/larcc/ex6.py @from simplexn import * from larcc import * from scipy.spatial import Delaunay @¡ Test for quasi-equilateral triangles @¿ @¡ Generation and selection of random triangles @¿ @¡ Boundary computation and visualisation @¿ @

@d Test for quasi-equilateral triangles @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 @

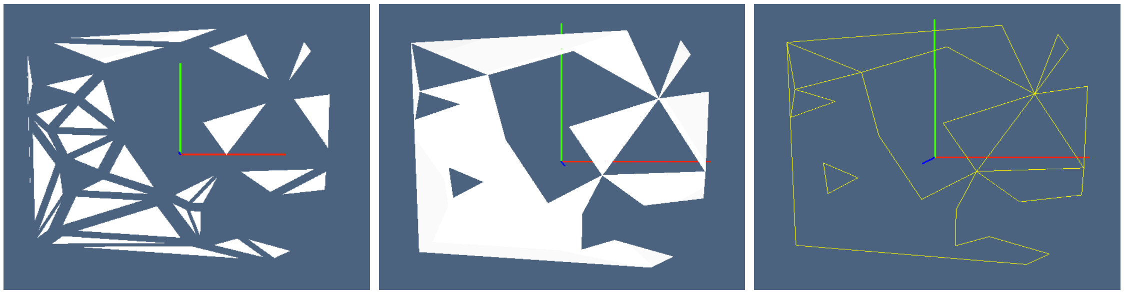

Figure 2: example caption

@d Generation and selection of random triangles @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)) @

@d Boundary computation and visualisation @boundaryCells$_1$ = $signedBoundaryCells(V,FV,EV)print"$ ", $boundaryCells_1 def swap(mylist) : return[mylist[1]]+[mylist[0]]+mylist[2 :]boundaryEV =$ $[EV[-k] if k < 0 else swap(EV[k]) for k in boundaryCells_1]bndry = (V, boundaryEV)VIEW(STRUCT(MKP$ $pols2D))VIEW(COLOR(RED)(STRUCT(MKPOLS(bndry))))$@

@d Compute the topologically ordered chain of boundary vertices @ @

@d Decompose a permutation into cycles @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 $_{name_{=="}}{}_{main_{"}}: print[2,3,4,5,6,7,0,1] print permutationOrbits([2,3,4,5,6,7,0,1]) print[3,9,8,4,10,7,2,11,6,0,1,5] print permutationOrbits([3,9,8,4,10,7,2,$

A.7 Assemblies of simplices and hypercubes

@o test/py/larcc/ex7.py @from simplexn import * from larcc import * from largrid import * @¡ Definition of 1-dimensional LAR models @¿ @¡ Assembly generation of squares and triangles @¿ @¡ Assembly generation of cubes and tetrahedra @¿ @

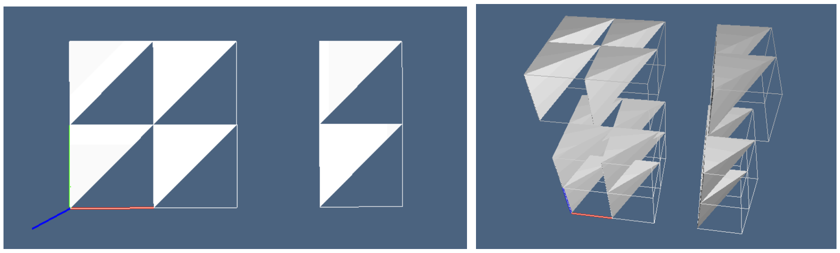

Figure 3: (a) Assemblies of squares and triangles; (b) assembly of cubes and tetrahedra.

@d Definition of 1-dimensional LAR models @geom$_0, topol_0 = [[0.], [1.], [2.], [3.], [4.]], [[0, 1], [1, 2], [3, 4]] geom_{\ldots}$
$[[0.], [1.], [2.]], [[0, 1], [1, 2]] mod_0 = (geom_0, topol_0) mod_1 = (geom_1, topol_1)$@

@d Assembly generation of squares and triangles @squares = larModelProduct([mod$_0, mod_1]) V, FV =$
$squares simplices = pivotSimplices(V, FV, d = 2) VIEW(STRUCT([MKPOL([V, AA(AA(C(SUM)(1)))(si\ldots$

@d Assembly generation of cubes and tetrahedra @cubes = larModelProduct([squares,mod$_0$])$V, CV =$
$cubes simplices = pivotSimplices(V, CV, d = 3) VIEW(STRUCT([MKPOL([V, AA(AA(C(SUM)(1)))(simp\ldots$