Domain mapping with LAR *

Alberto Paoluzzi

April 14, 2014

Abstract

In this module a first implementation (no optimisations) is done of several `LAR` operators, reproducing the behaviour of the plasm `STRUCT` and `MAP` primitives, but with better handling of the topology, including the stitching of decomposed (simplicial domains) about their possible sewing. A definition of specialised classes `Model`, `Mat` and `Verts` is also contained in this module, together with the design and the implementation of the *traversal* algorithms for networks of structures.

Contents

1 Domain decomposition

Standard and scaled decomposition of unit domain @d Generate a simplicial decomposition ot the $[0,1]^d$ domain @def larDomain(shape): V,CV = larSimplexGrid(shape) V = scalePoints(V, [1./d for d in shape]) return V,CV

def larIntervals(shape): def larIntervals0(size): V,CV = larDomain(shape) V = scalePoints(V, [scaleFactor for scaleFactor in size]) return V,CV return larIntervals0 @

2 Embedding via coordinate functions

2.1 Mapping domain vertices

@D Primitive mapping function @def larMap(coordFuncs): def larMap0(domain): V,CV = domain V = TRANS(CONS(coordFuncs)(V)) plasm CONStruction return V,CV return larMap0 @

*This document is part of the *Linear Algebraic Representation with CoChains* (LAR-CC) framework [**?**]. April 14, 2014

2.2 Identify close or coincident points

@D Create a dictionary with key the point location @def checkModel(model): V,CV = model; n = len(V) vertDict = defaultdict(list) for k,v in enumerate(V): vertDict[vcode(v)].append(k) verts = (vertDict.values()) invertedindex = [None]*n for k,value in enumerate(verts): for i in value: invertedindex[i]=value[0] CV = [[invertedindex[v] for v in cell] for cell in CV] filter out degenerate cells CV = [list(set(cell)) for cell in CV if len(set(cell))==len(cell)] return V, CV @

3 Embedding/Projecting models

In order apply 3D transformations to a two-dimensional LAR model, we must embed it in 3D space, by adding one more coordinate to its vertices.

Embedding or projecting a geometric model This task is performed by the following function `larEmbed` with parameter k, that inserts its d-dimensional geometric argument in the $x_{d+1}, \ldots, x_{d+k} = 0$ subspace. A reverse transformation of projection, removing the last k coordinate of vertices without changing the object topology, is performed by the function `larEmbed` with *negative* integer parameter.

@D Embedding and projecting a geometric model @def larEmbed(k): def larEmbed0(model): V,CV = model if k¿0: V = [v+[0.]*k for v in V] elif k¡0: V = [v[:-k] for v in V] return V,CV return larEmbed0 @

4 Primitive objets

4.1 1D primitives

Circle @D Circle centered in the origin @def larCircle(radius=1.): def larCircle0(shape=36): domain = larIntervals([shape])([2*PI]) V,CV = domain x = lambda V : [radius*COS(p[0]) for p in V] y = lambda V : [radius*SIN(p[0]) for p in V] return larMap([x,y])(domain) return larCircle0 @

4.2 2D primitives

Some useful 2D primitive objects either in $\mathbb{E}^2$ or embedded in $\mathbb{E}^3$ are defined here, including 2D disks and rings, as well as cylindrical, spherical and toroidal surfaces.

Disk @D Disk centered in the origin @def larDisk(radius=1.): def larDisk0(shape=[36,1]): domain = larIntervals(shape)([2*PI,radius]) V,CV = domain x = lambda V : [p[1]*COS(p[0]) for p in V] y = lambda V : [p[1]*SIN(p[0]) for p in V] return larMap([x,y])(domain) return larDisk0 @

Ring @D Ring centered in the origin @def larRing(params): r1,r2 = params def larRing0(shape=[36,1]): V,CV = larIntervals(shape)([2*PI,r2-r1]) V = translatePoints(V,[0,r1]) domain = V,CV x = lambda V : [p[1] * COS(p[0]) for p in V] y = lambda V : [p[1] * SIN(p[0]) for p in V] return larMap([x,y])(domain) return larRing0 @

Cylinder surface @D Cylinder surface with z axis @from scipy.linalg import det """ def makeOriented(model): V,CV = model out = [] for cell in CV: mat = scipy.array([V[v]+[1] for v in cell]+[[0,0,0,1]]) if det(mat) ¡ 0.0: out.append(cell) else: out.append([cell[1]]+[cell[0]]+cell[2:]) print "det(mat) =",det(mat) return V,out """ def larCylinder(params): radius,height= params def larCylinder0(shape=[36,1]): domain = larIntervals(shape)([2*PI,1]) V,CV = domain x = lambda V : [radius*COS(p[0]) for p in V] y = lambda V : [radius*SIN(p[0]) for p in V] z = lambda V : [height*p[1] for p in V] mapping = [x,y,z] model = larMap(mapping)(domain) model = makeOriented(model) return model return larCylinder0 @

Spherical surface of given radius @D Spherical surface of given radius @def larSphere(radius=1): def larSphere0(shape=[18,36]): V,CV = larIntervals(shape)([PI,2*PI]) V = translatePoints(V,[-PI/2,-PI]) domain = V,CV x = lambda V : [radius*COS(p[0])*COS(p[1]) for p in V] y = lambda V : [radius*COS(p[0])*SIN(p[1]) for p in V] z = lambda V : [radius*SIN(p[0]) for p in V] return larMap([x,y,z])(domain) return larSphere0 @

Toroidal surface @D Toroidal surface of given radiuses @def larToroidal(params): r,R = params def larToroidal0(shape=[24,36]): domain = larIntervals(shape)([2*PI,2*PI]) V,CV = domain x = lambda V : [(R + r*COS(p[0])) * COS(p[1]) for p in V] y = lambda V : [(R + r*COS(p[0])) * SIN(p[1]) for p in V] z = lambda V : [-r * SIN(p[0]) for p in V] return larMap([x,y,z])(domain) return larToroidal0 @

Crown surface @D Half-toroidal surface of given radiuses @def larCrown(params): r,R = params def larCrown0(shape=[24,36]): V,CV = larIntervals(shape)([PI,2*PI]) V = translatePoints(V,[-PI/2,0]) domain = V,CV x = lambda V : [(R + r*COS(p[0])) * COS(p[1]) for p in V] y = lambda V : [(R + r*COS(p[0])) * SIN(p[1]) for p in V] z = lambda V : [-r * SIN(p[0]) for p in V] return larMap([x,y,z])(domain) return larCrown0 @

4.3 3D primitives

Ball @D Solid Sphere of given radius @def larBall(radius=1): def larBall0(shape=[18,36]): V,CV = checkModel(larSphere(radius)(shape)) return V,[range(len(V))] return larBall0 @

Solid cylinder @D Solid cylinder of given radius and height @def larRod(params): radius,height= params def larRod0(shape=[36,1]): V,CV = checkModel(larCylinder(params)(shape)) return V,[range(len(V))] return larRod0 @

Solid torus @D Solid torus of given radiuses @def larTorus(params): r,R = params def larTorus0(shape=[24,36,1]): domain = larIntervals(shape)([2*PI,2*PI,r]) V,CV = domain x = lambda V : [(R + p[2]*COS(p[0])) * COS(p[1]) for p in V] y = lambda V : [(R + p[2]*COS(p[0])) * SIN(p[1]) for p in V] z = lambda V : [-p[2] * SIN(p[0]) for p in V] return larMap([x,y,z])(domain) return larTorus0 @

Solid pizza @D Solid pizza of given radiuses @def larPizza(params): r,R= params def larPizza0(shape=[24,36]): V,CV = checkModel(larCrown(params)(shape)) return V,[range(len(V))] return larPizza0 @

5 Affine transformations

5.1 Design decision

First we state the general rules that will be satisfied by the matrices used in this module, mainly devoted to apply affine transformations to vertices of models in structure environments:

1. assume the scipy `ndarray` as the type of vertices, stored in row-major order;
2. use the last coordinate as the homogeneous coordinate of vertices, but do not store it explicitly;
3. store explicitly the homogeneous coordinate of transformation matrices.
4. use labels '`verts`' and '`mat`' to distinguish between vertices and transformation matrices.
5. transformation matrices are dimension-independent, and their dimension is computed as the length of the parameter vector passed to the generating function.

5.2 map

@D Apply an affine transformation to a LAR model @def larApply(affineMatrix): def larApply0(model): if isinstance(model,Model): V = scipy.dot([v.tolist()+[1.0] for v in model.verts], affineMatrix.T).tolist() V = [v[:-1] for v in V] CV = copy(model.cells) d = copy(model.d) return Model((V,CV),d) elif isinstance(model,tuple): V,CV = model V = scipy.dot([v+[1.0] for v in V], affineMatrix.T).tolist() return [v[:-1] for v in V],CV return larApply0 @

5.3 Elementary matrices

Elementary matrices for affine transformation of vectors in any dimensional vector space are defined here. They include translation, scaling, rotation and shearing.

Translation @D Translation matrices @def t(*args): d = len(args) mat = scipy.identity(d+1) for k in range(d): mat[k,d] = args[k] return mat.view(Mat) @

Scaling @D Scaling matrices @def s(*args): d = len(args) mat = scipy.identity(d+1) for k in range(d): mat[k,k] = args[k] return mat.view(Mat) @

Rotation @D Rotation matrices @def r(*args): args = list(args) n = len(args) @¡ plane rotation (in 2D) @¿ @¡ space rotation (in 3D) @¿ return mat.view(Mat) @ @D plane rotation (in 2D) @if n == 1: rotation in 2D angle = args[0]; cos = COS(angle); sin = SIN(angle) mat = scipy.identity(3) mat[0,0] = cos; mat[0,1] = -sin; mat[1,0] = sin; mat[1,1] = cos; @ @D space rotation (in 3D) @if n == 3: rotation in 2D mat = scipy.identity(4) angle = VECTNORM(args); axis = UNITVECT(args) cos = COS(angle); sin = SIN(angle) @¡ elementary rotations (in 3D) @¿ @¡ general rotations (in 3D) @¿ @ @D elementary rotations (in 3D) @if axis[1]==axis[2]==0.0: rotation about x mat[1,1] = cos; mat[1,2] = -sin; mat[2,1] = sin; mat[2,2] = cos; elif axis[0]==axis[2]==0.0: rotation about y mat[0,0] = cos; mat[0,2] = sin; mat[2,0] = -sin; mat[2,2] = cos; elif axis[0]==axis[1]==0.0: rotation about z mat[0,0] = cos; mat[0,1] = -sin; mat[1,0] = sin; mat[1,1] = cos; @ @D general rotations (in 3D) @else: general 3D rotation (Rodrigues' rotation formula) I = scipy.identity(3) ; u = axis Ux = scipy.array([[0, -u[2], u[1]], [u[2], 0, -u[0]], [-u[1], u[0], 0]]) UU = scipy.array([[u[0]*u[0], u[0]*u[1], u[0]*u[2]], [u[1]*u[0], u[1]*u[1], u[1]*u[2]], [u[2]*u[0], u[2]*u[1], u[2]*u[2]]]) mat[:3,:3] = cos*I + sin*Ux + (1.0-cos)*UU @

6 Hierarchical complexes

Hierarchical models of complex assemblies are generated by an aggregation of subassemblies, each one defined in a local coordinate system, and relocated by affine transformations of coordinates. This operation may be repeated hierarchically, with some subassemblies defined by aggregation of simpler parts, and so on, until one obtains a set of elementary components, which cannot be further decomposed.

Two main advantages can be found in a hierarchical modeling approach. Each elementary part and each assembly, at every hierarchical level, are defined independently from each other, using a local coordinate frame, suitably chosen to make its definition easier. Furthermore, only one copy of each component is stored in the memory, and may be instanced in different locations and orientations how many times it is needed.

Script 8.3.1 (Traversal of a multigraph)

```
algorithm TRAVERSAL ((N, A, f) : multigraph) {
    CTM := identity matrix;
    TraverseNode (root)
}

proc TRAVERSENODE (n : node) {
    foreach a ∈ A outgoing from n do TraverseArc (a);
    ProcessNode (n)
}

proc TRAVERSEARC (a = (n, m) : arc) {
    Stack.push (CTM);
    CTM := CTM * a.mat;
    TraverseNode (m);
    CTM := Stack.pop()
}

proc PROCESSNODE (n : node) {
    foreach object ∈ n do Process( CTM * object )
}
```

Figure 1: Traversal algorithm of a multigraph.

6.1 Traversal

6.1.1 Traversal of a multigraph

6.1.2 Traversal of nested lists

show the pattern of calls and returned values @D Traversal of a multigraph @def traverse(CTM, stack, o, flag): i = 0 while i ¡ len(o): print "lunghezza lista", len(o) if ISNUM(o[i]): print o[i], REVERSE(CTM) if ISSTRING(o[i]): print "E' una lettera " + o[i] CTM.append(o[i]) print "trasformazione attuale", CTM if flag: stack.append(o[i]) if ISLIST(o[i]): stack = [] print "E' una sottolista", o[i] traverse(CTM, stack, o[i], True) CTM = CTM[:-len(stack)] stack = [] flag = False i = i + 1

def algorithm(data): CTM,stack = ["I"],[] traverse(CTM, stack, data, False) @

Traversal of a scene multigraph The previous traversal algorithm is here customised for scene multigraph, where the objects are LAR models, i.e. pairs of vertices of type '`Verts` and cells, and where the transformations are matrix transformations of type '`Mat`'.

Check models for common dimension The input list of a call to `larStruct` primitive is preliminary checked for uniform dimensionality of the enclosed LAR models and transformations. The common dimension `dim` of models and matrices is returned by the function `checkStruct`. Otherwise, an exception is generated (TODO).

@D check LAR models in input struct for common dimension @""" Check models for common dimension """

data = [Model(larDomain((3,2)),2), r(PI), Struct([Model(larDomain((1,4)),2), t(1,1), Model(larDomain((4,2)),2)]), Model(larDomain((5,4)),2)]

a = Struct(data) @

Initialization @D Traversal of a scene multigraph @""" Traversal of a scene multigraph """ @¡ Structure traversal algorithm @¿

def evalStruct(struct): @¡ check LAR models in input Struct for common dimension @¿ dim = struct.n CTM,stack = scipy.identity(dim+1),[] print "CTM,stack =",(CTM,stack) traverse(CTM, stack, data, False) @

Structure traversal algorithm @D Structure traversal algorithm @def traverse(CTM, stack, o, flag): i = 0 while i ¡ len(o): print "lunghezza lista", len(o) if ISNUM(o[i]): print o[i], REVERSE(CTM) if ISSTRING(o[i]): print "E' una lettera " + o[i] CTM.append(o[i]) print "trasformazione attuale", CTM if flag: stack.append(o[i]) if ISLIST(o[i]): stack = [] print "E' una sottolista", o[i] traverse(CTM, stack, o[i], True) CTM = CTM[:-len(stack)] stack = [] flag = False i = i + 1 @

@D Examples of multigraph traversal @data = [1,"A", 2, 3, "B", [4, "C", 5], [6,"D", "E", 7, 8], 9] print algorithm(data) data = [1,"A", [2, 3, "B", 4, "C", 5, 6,"D"], "E", 7, 8,

9] print algorithm(data) data = [2, 3, "B", 4, "C", 5, 6,"D"] print algorithm(data) dat = [2, 3, "B", 4, "C", 5, 6,"D"] data = [1,"A", dat, "E", 7, 8, 9] print algorithm(data) @

show the pattern of calls and returned values @D Let us explore the pattern even better @data = [(1,6, (7,1, (8,"12"))), (2,9,10), (3,), 4, (5, (11, ("13",)))]
print list(traverse(data)) @

6.2 Example

Some simple examples of structures as combinations of LAR models and affine transformations are given in this section.

Global coordinates We start with a simple 2D example of a non-nested list of translated 2D object instances and rotation about the origin.

@O test/py/mapper/test04.py @""" Example of non-nested structure with translation and rotations """ square = larCuboids([1,1]) table = larApply(t(-.5,-.5))(square) chair = larApply(s(.35,.35))(table) chair1 = larApply(t(.75, 0))(chair) chair2 = larApply(r(PI/2))(chair1) chair3 = larApply(r(PI/2))(chair2) chair4 = larApply(r(PI/2))(chair3) VIEW(SKEL$_1$($STRUCT(MKPOLS(table)+MKPOLS(chair1)+MKPOLS(chair2)+MKPOLS(chair3)+MKPOLS(chair4))$))@

Local coordinates A different composition of transformations, from local to global doordinate frames, is used in yje following example.

@O test/py/mapper/test05.py @""" Example of non-nested structure with translation and rotations """ square = larCuboids([1,1]) square = Model(square,2) table = larApply(t(-.5,-.5))(square) chair = larApply(s(.35,.35))(table) struct = Struct(4*[chair, r(PI/2)]) scene = evalStruct(struct) VIEW(SKEL$_1$($STRUCT(MKPOLS(scene)$))@

6.3 aaaaaa

7 Exporting the library

@O lib/py/mapper.py @""" Mapping functions and primitive objects """ @¡ Initial import of modules @¿ @¡ Generate a simplicial decomposition ot the $[0,1]^d$ domain @¿ @¡ Create a dictionary with key the point location @¿ @¡ Primitive mapping function @¿ @¡ Basic tests of mapper module @¿ @¡ Circle centered in the origin @¿ @¡ Disk centered in the origin @¿ @¡ Ring centered in the origin @¿ @¡ Spherical surface of given radius @¿ @¡ Cylinder surface with z axis @¿ @¡ Toroidal surface of given radiuses @¿ @¡ Half-toroidal surface of given radiuses @¿ @¡ Solid Sphere of given radius @¿ @¡ Solid cylinder of given radius and height @¿ @¡ Solid torus of given radiuses @¿ @¡ Solid pizza of given radiuses @¿ @¡ Translation matrices @¿ @¡ Scaling matrices @¿ @¡ Rotation matrices @¿ @¡ Embedding

and projecting a geometric model @¿ @¡ Apply an affine transformation to a LAR model @¿ @¡ Traversal of a scene multigraph @¿ @

8 Examples

3D rotation about a general axis @O test/py/mapper/test02.py @""" General 3D rotation of a toroidal surface """ @¡ Initial import of modules @¿ from mapper import * model = checkModel(larToroidal([0.2,1])()) a,b,c = SCALARVECTPROD([PI/2,UNITVECT([1,1,0])]) model = larApply(r(a,b,c))(model) VIEW(STRUCT(MKPOLS(model))) @

3D rotation of a 2D circle @O test/py/mapper/test03.py @""" Elementary 3D rotation of a 2D circle """ @¡ Initial import of modules @¿ from mapper import * model = checkModel(larCircle(1)()) model = larEmbed(1)(model) model = larApply(r(PI/2,0,0))(model) VIEW(STRUCT(MKPOLS(model))) @

9 Tests

@D Basic tests of mapper module @if $_{n}ame_{==}{}^{"}{}_{m}ain_{"}{}_{:}V,EV{=}larDomain([5])VIEW(EXPLODE(1.5,1.5,1.5)(MKPOLS((V,EV))))V,EV{=}l$

V,FV = larDomain([5,3]) VIEW(EXPLODE(1.5,1.5,1.5)(MKPOLS((V,FV)))) V,FV = larIntervals([36,3])([2*PI,1.]) VIEW(EXPLODE(1.5,1.5,1.5)(MKPOLS((V,FV))))

V,CV = larDomain([5,3,1]) VIEW(EXPLODE(1.5,1.5,1.5)(MKPOLS((V,CV)))) V,CV = larIntervals([36,2,3])([2*PI,1.,1.]) VIEW(EXPLODE(1.5,1.5,1.5)(MKPOLS((V,CV))))
@

Circumference of unit radius @O test/py/mapper/test01.py @""" Circumference of unit radius """ @¡ Initial import of modules @¿ from mapper import * model = checkModel(larCircle(1)()) VIEW(EXPLODE(1.2,1.2,1.2)(MKPOLS(model))) """ model = checkModel(larDisk(1)([36,4])) VIEW(EXPLODE(1.2,1.2,1.2)(MKPOLS(model))) model = checkModel(larRing([.9, 1.])([36,2])) VIEW(EXPLODE(1.2,1.2,1.2)(MKPOLS(model))) model = checkModel(larCylinder([.5,2.])([32,1])) VIEW(STRUCT(MKPOLS(model))) model = checkModel(larSphere(1)()) VIEW(STRUCT(MKPOLS(model))) model = larBall(1)() VIEW(STRUCT(MKPOLS(model))) model = larRod([.25,2.])([32,1]) VIEW(STRUCT(MKPOLS(model))) model = checkModel(larToroidal([0.5,1])()) VIEW(STRUCT(MKPOLS(model))) model = checkModel(larCrown([0.125,1])([8,48])) VIEW(STRUCT(MKPOLS(model))) model = larPizza([0.05,1])([8,48]) VIEW(STRUCT(MKPOLS(model))) model = checkModel(larTorus([0.5,1])()) VIEW(STRUCT(MKPOLS(model))) """ @

A Utility functions

@D Initial import of modules @from pyplasm import * from scipy import * import os,sys """ import modules from larcc/lib """ sys.path.insert(0, 'lib/py/') @¡ Import the module @(lar2psm@) @¿ @¡ Import the module @(simplexn@) @¿ @¡ Import the module @(larcc@) @¿ @¡ Import the module @(largrid@) @¿ @¡ Import the module @(boolean2@) @¿ @

@D Import the module @import @1 from @1 import * @

A.1 Numeric utilities

A small set of utilityy functions is used to transform a point representation as array of coordinates into a string of fixed format to be used as point key into python dictionaries.

@D Symbolic utility to represent points as strings @""" TODO: use package Decimal (http://docs.python.org/2/library/decimal.html) """ ROUND$_Z ERO = 1E{-}07 def round_o r_z ero(x, prec =$ 7) : """ $Decision proceduretoapproximateasmallnumbertozero.Returneithertheinputnumberorzero."""def m$ $returneval(('xx = myround(x)if abs(xx) < ROUND_Z ERO : return 0.0else : returnxx$

def prepKey (args): return "["+", ".join(args)+"]"

def fixedPrec(value): if abs(value - int(value))¡ROUND$_Z ERO : value = int(value)out =$ $('if out ==' -0.' : out =' 0.'returnout$

def vcode (vect): """ To generate a string representation of a number array. Used to generate the vertex keys in PointSet dictionary, and other similar operations. """ return prepKey(AA(fixedPrec)(vect)) @