Domain mapping with LAR *

Alberto Paoluzzi

October 8, 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

*This document is part of the *Linear Algebraic Representation with CoChains* (LAR-CC) framework [CL13]. October 8, 2014

1 Introduction

The `mapper` module, introduced here, aims to provide the tools needed to apply both dimension-independent affine transformations and general simplicial maps to geometric objects and assemblies developed within the LAR scheme.

For this purpose, a simplicial decomposition of the $[0,1]^d$ hypercube ($d \geq 1$) with any possible `shape` is firstly given, followed by its scaled version with any according `size` $\in \mathbb{E}^d$, being its position vector the mapped image of the point $\mathbf{1} \in \mathbb{E}^d$. A general mapping mechanism is specified, to map any domain decomposition (either simplicial or not) with a given set of coordinate functions, providing a piecewise-linear approximation of any curved embedding of a d-dimensional domain in any $\mathbb{E}^n$ space, with $n \geq d$. A suitable function is also given to identify corresponding vertices when mapping a domain decomposition of the fundamental polygon (or polyhedron) of a closed manifold.

The geometric tools given in this chapter employ a normalised homogeneous representation of vertices of the represented shapes, where the added coordinate is the *last* of the ordered list of vertex coordinates. The homogeneous representation of vertices is used *implicitly*, by inserting the extra coordinate only when needed by the operation at hand, mainly for computing the product of the object's vertices times the matrix of an affine tensor.

A set of primitive surface and solid shapes is also provided, via the mapping mechanism of a simplicial decomposition of a d-dimensional chart. A simplified version of the PLaSM specification of dimension-independent elementary affine transformation is given as well.

The second part of this module is dedicated to the development of a complete framework for the implementation of hierarchical assemblies of shapes and scene graphs, by using the simplest possible set of computing tools. In this case no hierarchical graphs or multigraph are employed, i.e. no specialised data structures are produced. The ordered list model of hierarchical structures, inherited from PHIGS and PLaSM, is employed in this context. A recursive traversal is used to transform all the component parts of a hierarchical assembly into the reference frame of the first object of the assembly, i.e. in world coordinates.

2 Piecewise-linear mapping of topological spaces

A very simple but foundational software subsystem is developed in this section, by giving a general mechanism to produce curved maps of topological spaces, via the simplicial decomposition of a chart, i.e. of a planar embedding of the fundamental polygon of a d-dimensional manifold, and the definition of coordinate functions to be applied to its vertices (0-cells of the decomposition) to generate an embedding of the manifold.

2.1 Domain decomposition

A simplicial map is a map between simplicial complexes with the property that the images of the vertices of a simplex always span a simplex. Simplicial maps are thus determined by their effects on vertices, and provide a piecewise-linear approximation of their underlying polyhedra.

Since double simmeries are always present in the curved primitives generated in the module, an alternative cellular decomposition with cuboidal cells is provided. The default choice is "cuboid".

Standard and scaled decomposition of unit domain The `larDomain` of given `shape` is decomposed by `larSimplexGrid1` as an hypercube of dimension $d \equiv$ `len(shape)`, where the `shape` tuple provides the number or row, columns, pages, etc. of the decomposition.

$\langle$ Generate a simplicial decomposition ot the $[0, 1]^d$ domain 2 $\rangle \equiv$

```
""" cellular decomposition of the unit d-cube """
def larDomain(shape, cell='cuboid'):
   if cell=='simplex': V,CV = larSimplexGrid1(shape)
   elif cell=='cuboid': V,CV = larCuboids(shape)
   V = larScale( [1./d for d in shape])(V)
   return [V,CV]
◇
```

Macro referenced in 24b.

A scaled simplicial decomposition is provided by the second-order `larIntervals` function, with `len(shape)` and `len(size)` parameters, where the d-dimensionale vector `len(size)` is assumed as the scaling vector to be applied to the point $\mathbf{1} \in \mathbb{E}^d$.

$\langle$ Scaled simplicial decomposition ot the $[0, 1]^d$ domain 3a $\rangle \equiv$

```
def larIntervals(shape, cell='cuboid'):
   def larIntervals0(size):
      V,CV = larDomain(shape,cell)
      V = larScale( size)(V)
      return [V,CV]
   return larIntervals0
◇
```

Macro referenced in 24b.

2.2 Mapping domain vertices

The second-order texttttlarMap function is the LAR implementation of the PLaSM primitive `MAP`. It is applied to the array `coordFuncs` of coordinate functions and to the simplicially decomposed `domain`, returning an embedded and/or curved `domain` instance.

⟨ Primitive mapping function 3b ⟩ ≡

```
def larMap(coordFuncs):
   if isinstance(coordFuncs, list): coordFuncs = CONS(coordFuncs)
   def larMap0(domain,dim=2):
      V,CV = domain
      V = AA(coordFuncs)(V)  # plasm CONStruction
      return [V,CV]
      # checkModel([V,CV])  TODO
   return larMap0
◇
```

Macro referenced in 24b.

2.3 Identify close or coincident points

The function `checkModel`, applied to a `model` parameter, i.e. to a (vertices, cells) pair, returns the model after identification of vertices with coincident or very close position vectors. The `checkModel` function works as follows: first a dictionary `vertDict` is created, with key a suitably approximated position converted into a string by the `vcode` converter (given in the Appendix), and with value the list of vertex indices with the same (approximated) position. Then, an `invertedindex` array is created, associating each original vertex index with the new index produced by enumerating the (distinct) keys of the dictionary. Finally, a new list `CV` of cells is created, by substituting the new vertex indices for the old ones.

⟨ Create a dictionary with key the point location 4a ⟩ ≡

```
from collections import defaultdict
def checkModel(model,dim=2):
   V,CV = model; n = len(V)
   vertDict = defaultdict(list)
   for k,v in enumerate(V): vertDict[vcode(v)].append(k)
```

```
    points,verts = TRANS(vertDict.items())
    invertedindex = [None]*n
    V = []
    for k,value in enumerate(verts):
        V.append(eval(points[k]))
        for i in value:
            invertedindex[i]=k
    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))>=dim+1]
    return [V, CV]
```
◇

Macro referenced in 24b.

2.4 Embedding or projecting LAR models

In order to 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 function `larEmbed` with parameter k, that inserts its d-dimensional geometric argument in the $x_{d+1}, \ldots, x_{d+k} = 0$ subspace of $\mathbb{E}^{d+k}$. A projection transformation, that removes the last k coordinate of vertices, without changing the object topology, is performed by the function `larEmbed` with *negative* integer parameter.

⟨Embedding and projecting a geometric model 4b⟩ ≡
```
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
```
◇

Macro referenced in 24b.

3 Primitive objects

A large number of primitive surfaces or solids is defined in this section, using the `larMap` mechanism and the coordinate functions of a suitable chart.

3.1 1D primitives

Circle

⟨ Circle centered in the origin 5a ⟩ ≡

```
def larCircle(radius=1.,angle=2*PI,dim=1):
   def larCircle0(shape=36):
      domain = larIntervals([shape])([angle])
      V,CV = domain
      x = lambda p : radius*COS(p[0])
      y = lambda p : radius*SIN(p[0])
      return larMap([x,y])(domain,dim)
   return larCircle0
```

◇

Macro referenced in 24b.

Helix curve

⟨ Helix curve about the z axis 5b ⟩ ≡

```
def larHelix(radius=1.,pitch=1.,nturns=2,dim=1):
   def larHelix0(shape=36*nturns):
      angle = nturns*2*PI
      domain = larIntervals([shape])([angle])
      V,CV = domain
      x = lambda p : radius*COS(p[0])
      y = lambda p : radius*SIN(p[0])
      z = lambda p : (pitch/(2*PI)) * p[0]
      return larMap([x,y,z])(domain,dim)
   return larHelix0
```

◇

Macro referenced in 24b.

3.2 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 surface

⟨ Disk centered in the origin 6a ⟩ ≡

```
def larDisk(radius=1.,angle=2*PI):
   def larDisk0(shape=[36,1]):
      domain = larIntervals(shape)([angle,radius])
      V,CV = domain
      x = lambda p : p[1]*COS(p[0])
```

```
      y = lambda p : p[1]*SIN(p[0])
      return larMap([x,y])(domain)
   return larDisk0
◇
```

Macro referenced in 24b.

Helicoid surface

⟨ Helicoid about the z axis 6b ⟩ ≡

```
def larHelicoid(R=1.,r=0.5,pitch=1.,nturns=2,dim=1):
   def larHelicoid0(shape=[36*nturns,2]):
      angle = nturns*2*PI
      domain = larIntervals(shape,'simplex')([angle,R-r])
      V,CV = domain
      V = larTranslate([0,r,0])(V)
      domain = V,CV
      x = lambda p : p[1]*COS(p[0])
      y = lambda p : p[1]*SIN(p[0])
      z = lambda p : (pitch/(2*PI)) * p[0]
      return larMap([x,y,z])(domain,dim)
   return larHelicoid0
◇
```

Macro referenced in 24b.

Ring surface

⟨ Ring centered in the origin 6c ⟩ ≡

```
def larRing(r1,r2,angle=2*PI):
   def larRing0(shape=[36,1]):
      V,CV = larIntervals(shape)([angle,r2-r1])
      V = larTranslate([0,r1])(V)
      domain = V,CV
      x = lambda p : p[1] * COS(p[0])
      y = lambda p : p[1] * SIN(p[0])
      return larMap([x,y])(domain)
   return larRing0
◇
```

Macro referenced in 24b.

Cylinder surface

⟨ Cylinder surface with z axis 7a ⟩ ≡

```
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:])
   return V,out
"""
def larCylinder(radius,height,angle=2*PI):
   def larCylinder0(shape=[36,1]):
      domain = larIntervals(shape)([angle,1])
      V,CV = domain
      x = lambda p : radius*COS(p[0])
      y = lambda p : radius*SIN(p[0])
      z = lambda p : height*p[1]
      mapping = [x,y,z]
      model = larMap(mapping)(domain)
      # model = makeOriented(model)
      return model
   return larCylinder0
◇
```

Macro referenced in 24b.

Spherical surface of given radius

⟨ Spherical surface of given radius 7b ⟩ ≡

```
def larSphere(radius=1,angle1=PI,angle2=2*PI):
   def larSphere0(shape=[18,36]):
      V,CV = larIntervals(shape,'simplex')([angle1,angle2])
      V = larTranslate([-angle1/2,-angle2/2])(V)
      domain = V,CV
      x = lambda p : radius*COS(p[0])*COS(p[1])
      y = lambda p : radius*COS(p[0])*SIN(p[1])
      z = lambda p : radius*SIN(p[0])
      return larMap([x,y,z])(domain)
   return larSphere0
◇
```

Macro referenced in 24b.

Toroidal surface

⟨ Toroidal surface of given radiuses 8a ⟩ ≡

```
def larToroidal(r,R,angle1=2*PI,angle2=2*PI):
   def larToroidal0(shape=[24,36]):
      domain = larIntervals(shape,'simplex')([angle1,angle2])
      V,CV = domain
      x = lambda p : (R + r*COS(p[0])) * COS(p[1])
      y = lambda p : (R + r*COS(p[0])) * SIN(p[1])
      z = lambda p : -r * SIN(p[0])
      return larMap([x,y,z])(domain)
   return larToroidal0
```

◇

Macro referenced in 24b.

Crown surface

⟨ Half-toroidal surface of given radiuses 8b ⟩ ≡

```
def larCrown(r,R,angle=2*PI):
   def larCrown0(shape=[24,36]):
      V,CV = larIntervals(shape,'simplex')([PI,angle])
      V = larTranslate([-PI/2,0])(V)
      domain = V,CV
      x = lambda p : (R + r*COS(p[0])) * COS(p[1])
      y = lambda p : (R + r*COS(p[0])) * SIN(p[1])
      z = lambda p : -r * SIN(p[0])
      return larMap([x,y,z])(domain)
   return larCrown0
```

◇

Macro referenced in 24b.

3.3 3D primitives

Solid Box

⟨ Solid box of given extreme vectors 8c ⟩ ≡

```
def larBox(minVect,maxVect):
   size = DIFF([maxVect,minVect])
   print "size =",size
   box = larApply(s(*size))(larCuboids([1,1,1]))
   print "box =",box
   return larApply(t(*minVect))(box)
```

◇

Macro referenced in 24b.

Solid helicoid

⟨ Solid helicoid about the z axis 9a ⟩ ≡

```
def larSolidHelicoid(thickness=.1,R=1.,r=0.5,pitch=1.,nturns=2.,steps=36):
   def larSolidHelicoid0(shape=[steps*int(nturns),1,1]):
      angle = nturns*2*PI
      domain = larIntervals(shape)([angle,R-r,thickness])
      V,CV = domain
      V = larTranslate([0,r,0])(V)
      domain = V,CV
      x = lambda p : p[1]*COS(p[0])
      y = lambda p : p[1]*SIN(p[0])
      z = lambda p : (pitch/(2*PI))*p[0] + p[2]
      return larMap([x,y,z])(domain)
   return larSolidHelicoid0
◇
```

Macro referenced in 24b.

Solid Ball

⟨ Solid Sphere of given radius 9b ⟩ ≡

```
def larBall(radius=1,angle1=PI,angle2=2*PI):
   def larBall0(shape=[18,36]):
      V,CV = checkModel(larSphere(radius,angle1,angle2)(shape))
      return V,[range(len(V))]
   return larBall0
◇
```

Macro referenced in 24b.

Solid cylinder

⟨ Solid cylinder of given radius and height 9c ⟩ ≡

```
def larRod(radius,height,angle=2*PI):
   def larRod0(shape=[36,1]):
      V,CV = checkModel(larCylinder(radius,height,angle)(shape))
      return V,[range(len(V))]
   return larRod0
◇
```

Macro referenced in 24b.

Hollow cylinder

⟨ Hollow cylinder of given radiuses and height 10a ⟩ ≡

```
def larHollowCyl(r,R,height,angle=2*PI):
   def larHollowCyl0(shape=[36,1,1]):
      V,CV = larIntervals(shape)([angle,R-r,height])
      V = larTranslate([0,r,0])(V)
      domain = V,CV
      x = lambda p : p[1] * COS(p[0])
      y = lambda p : p[1] * SIN(p[0])
      z = lambda p : p[2] * height
      return larMap([x,y,z])(domain)
   return larHollowCyl0
◇
```

Macro referenced in 24b.

Hollow sphere

⟨ Hollow sphere of given radiuses 10b ⟩ ≡

```
def larHollowSphere(r,R,angle1=PI,angle2=2*PI):
   def larHollowSphere0(shape=[36,1,1]):
      V,CV = larIntervals(shape)([angle1,angle2,R-r])
      V = larTranslate([-angle1/2,-angle2/2,r])(V)
      domain = V,CV
      x = lambda p : p[2]*COS(p[0])*COS(p[1])
      y = lambda p : p[2]*COS(p[0])*SIN(p[1])
      z = lambda p : p[2]*SIN(p[0])
      return larMap([x,y,z])(domain)
   return larHollowSphere0
◇
```

Macro referenced in 24b.

Solid torus

⟨ Solid torus of given radiuses 10c ⟩ ≡

```
def larTorus(r,R,angle1=2*PI,angle2=2*PI):
   def larTorus0(shape=[24,36,1]):
      domain = larIntervals(shape)([angle1,angle2,r])
      V,CV = domain
      x = lambda p : (R + p[2]*COS(p[0])) * COS(p[1])
      y = lambda p : (R + p[2]*COS(p[0])) * SIN(p[1])
      z = lambda p : -p[2] * SIN(p[0])
      return larMap([x,y,z])(domain)
   return larTorus0
◇
```

Macro referenced in 24b.

Solid pizza

⟨ Solid pizza of given radiuses 11a ⟩ ≡

```
def larPizza(r,R,angle=2*PI):
   assert angle <= PI
   def larPizza0(shape=[24,36]):
      V,CV = checkModel(larCrown(r,R,angle)(shape))
      V += [[0,0,-r],[0,0,r]]
      return V,[range(len(V))]
   return larPizza0
```

◇

Macro referenced in 24b.

4 Affine transformations

4.1 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.

4.2 Affine mapping

⟨ Apply an affine transformation to a LAR model 11b ⟩ ≡

```
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 = scipy.dot(array([v+[1.0] for v in model.verts]), affineMatrix.T).tolist()
         V = [v[:-1] for v in V]
         CV = copy(model.cells)
         return Model((V,CV))
```

```
        elif isinstance(model,tuple) or isinstance(model,list):
            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
```
◇

Macro referenced in 24b.

4.3 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

⟨ Translation matrices 12a ⟩ ≡

```
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)
```
◇

Macro referenced in 24b.

Scaling

⟨ Scaling matrices 12b ⟩ ≡

```
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)
```
◇

Macro referenced in 24b.

Rotation

⟨ Rotation matrices 12c ⟩ ≡

```
def r(*args):
    args = list(args)
    n = len(args)
    ⟨plane rotation (in 2D) 13a⟩
```

```
⟨ space rotation (in 3D) 13b ⟩
return mat.view(Mat)
```
◇

Macro referenced in 24b.

⟨ plane rotation (in 2D) 13a ⟩ ≡

```
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;
```
◇

Macro referenced in 12c.

⟨ space rotation (in 3D) 13b ⟩ ≡

```
if n == 3: # rotation in 3D
   mat = scipy.identity(4)
   angle = VECTNORM(args); axis = UNITVECT(args)
   cos = COS(angle); sin = SIN(angle)
   ⟨ elementary rotations (in 3D) 13c ⟩
   ⟨ general rotations (in 3D) 13d ⟩
```
◇

Macro referenced in 12c.

⟨ elementary rotations (in 3D) 13c ⟩ ≡

```
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;
```
◇

Macro referenced in 13b.

⟨ general rotations (in 3D) 13d ⟩ ≡

```
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
    ◇
```

Macro referenced in 13b.

5 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.

5.1 Traversal of hierarchical structures

Of course, the main algorithm with hierarchical structures is the *traversal* of the structure network, whose aim is to transform every encountered object from local to global coordinates, where the global coordinates are those of the network root (the only node with indegree zero).

A structure network can be modelled using a directed acyclic multigraph, i.e. a triple (N, A, f) made by a set N of nodes, a set A of arcs, and a function $f : A \to N^2$ from arcs to ordered pairs of nodes. Conversely that in standard oriented graphs, in this kind of structure more than one oriented arc is allowed between the same pair on nodes.

A simple modification of a DFS (Depth First Search) visit of a graph can be used to traverse the structure network This algorithm is given in Figure 1 from [Pao03].

5.1.1 Traversal of nested lists

The representation chosen for structure networks with LAR is the serialised one, consisting in ordered sequences (lists) of either (a) LAR models, or (b) affine transformations, or (c) references to other structures, either directly nested within some given structure, or called by reference (name) from within the list.

The aim of a structure network traversal is, of course, to transform every component structure, usually defined in a local coordinate system, into the reference frame of the

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 an acyclic multigraph.

structure as a whole, normally corresponding with the reference system of the structure's root, called the *world coordinate* system.

The pattern of calls and returned values In order to better understand the behaviour of the traversal algorithm, where every transformation is applied to all the following models, — but only if included in the same structure (i.e. list) — it may be very useful to start with an *algorithm emulation*. In particular, the recursive script below discriminates between three different cases (number, string, or sequence), whereas the actual traversal must do with (a) Models, (b) Matrices, and (c) Structures, respectively.

⟨Emulation of scene multigraph traversal 16a⟩ ≡

```
from pyplasm import *
def __traverse(CTM, stack, o):
   for i in range(len(o)):
      if ISNUM(o[i]): print o[i], REVERSE(CTM)
      elif ISSTRING(o[i]):
         CTM.append(o[i])
      elif ISSEQ(o[i]):
         stack.append(o[i])            # push the stack
         __traverse(CTM, stack, o[i])
         CTM = CTM[:-len(stack)]       # pop the stack

def algorithm(data):
   CTM,stack = ["I"],[]
   __traverse(CTM, stack, data)
◇
```

Macro never referenced.

Some use example of the above algorithm are provided below. The printout produced at run time is shown from the `emulation of traversal algorithm` macro.

⟨Examples of multigraph traversal 16b⟩ ≡

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

data = [1,"A", [2, 3, "B", 4, "C", 5, 6,"D"], "E", 7, 8, 9]
```

```
print algorithm(data)
>>> 1 ['I']
   2 ['A', 'I']
   3 ['A', 'I']
   4 ['B', 'A', 'I']
   5 ['C', 'B', 'A', 'I']
   6 ['C', 'B', 'A', 'I']
   7 ['E', 'A', 'I']
   8 ['E', 'A', 'I']
   9 ['E', 'A', 'I']
◇
```

Macro never referenced.

⟨ Emulation of traversal algorithm 17 ⟩ ≡

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

Macro never referenced.

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`, within the class definition `Struct` in the module `lar2psm`. Otherwise, an exception is generated (TODO).

⟨Check for dimension of a structure element (Verts or V) 18a⟩ ≡

```
⟨Flatten a list 18b⟩
def checkStruct(lst):
   """ Return the common dimension of structure elements.
   """
   vertsDims = [computeDim(obj) for obj in flatten(lst)]
   if EQ(vertsDims):
      return vertsDims[0]
   else:
      print "\n vertsDims =", vertsDims
      print "*** LAR ERROR: Struct dimension mismatch."

def computeDim(obj):
   """ Check for dimension of a structure element (Verts or V).
   """
   if (isinstance(obj,lar2psm.Model)):
      return obj.n
   elif (isinstance(obj,tuple) or isinstance(obj,list)) and len(obj)==2:
      V = obj[0]
      if (isinstance(V,list) and isinstance(V[0],list) and
            (isinstance(V[0][0],float) or isinstance(V[0][0],int))):
         dim = len(V[0])
         return dim
   elif (isinstance(obj,lar2psm.Mat)):
      dim = obj.shape[0]-1
      return dim
   else: return 0
◇
```

Macro referenced in 24b.

Flatten a list using Python generators The `flatten` is a generator that yields the non-list values of its input in order. In the example, the generator is converted back to a list before printing. Modified from *Rosetta code* project. It is used here to flatten a structure in order to check for common dimensionality of elements.

⟨Flatten a list 18b⟩ ≡

```
""" Flatten a list using Python generators """
def flatten(lst):
   for x in lst:
      if (isinstance(x,tuple) or isinstance(x,list)) and len(x)==2:
         yield x
      elif (isinstance(x,tuple) or isinstance(x,list)):
         for x in flatten(x):
            yield x
      elif isinstance(x, Struct):
```

```
            for x in flatten(x.body):
                yield x
        else:
            yield x

# lst = [[1], 2, [[3,4], 5], [[[]]], [[[6]]], 7, 8, []]
# print list(flatten(lst))
# [1, 2, 3, 4, 5, 6, 7, 8]
◇
```

Macro referenced in 18a.

Initialization and call of the algorithm The function `evalStruct` is used to evaluate a structure network, i.e. to return a `scene` list of objects of type `Model`, all referenced in the world coordinate system. The input variable `struct` must contain an object of class `Struct`, i.e. a reference to an unevaluated structure network. The variable `dim` contains the embedding dimension of the structure, i.e. the number of doordinates of its vertices (normally either 2 or 3), the CTM (Current Transformation Matrix) is initialised to the (homogeneous) identity matrix, and the `scene` is returned by calling the `traverse` algorithm.

⟨ Traversal of a scene multigraph 19a ⟩ ≡

```
""" Traversal of a scene multigraph """
⟨ Structure traversal algorithm 19b ⟩
def evalStruct(struct):
    dim = checkStruct(struct.body)
    print "\n dim =",dim
    CTM, stack = scipy.identity(dim+1), []
    print "\n CTM, stack =",CTM, stack
    scene = traversal(CTM, stack, struct)
    return scene
◇
```

Macro referenced in 24b.

Structure traversal algorithm The `traversal` algorithm decides between three different cases, depending on the type of the currently inspected object. If the object is a `Model` instance, then applies to it the `CTM` matrix; else if the object is a `Mat` instance, then the `CTM` matrix is updated by (right) product with it; else if the object is a `Struct` instance, then the `CTM` is pushed on the stack, initially empty, then the `traversal` is called (recursion), and finally, at (each) return from recursion, the `CTM` is recovered by popping the stack.

⟨ Structure traversal algorithm 19b ⟩ ≡

```
def traversal(CTM, stack, obj, scene=[]):
   print "\n CTM, obj =",obj
   for i in range(len(obj)):
      if isinstance(obj[i],Model):
         scene += [larApply(CTM)(obj[i])]
      elif (isinstance(obj[i],tuple) or isinstance(obj[i],list)) and len(obj[i])==2:
         scene += [larApply(CTM)(obj[i])]
      elif isinstance(obj[i],Mat):
         CTM = scipy.dot(CTM, obj[i])
      elif isinstance(obj[i],Struct):
         stack.append(CTM)
         traversal(CTM, stack, obj[i], scene)
         CTM = stack.pop()
   return scene
◇
```

Macro referenced in 19a.

5.2 Example

Some 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.

"test/py/mapper/test04.py" 20a ≡

```
""" Example of non-nested structure with translation and rotations """
⟨Initial import of modules 28b⟩
from mapper import *
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 coordinate frames, is used in the following example.

"test/py/mapper/test05.py" 20b ≡

```
""" Example of non-nested structure with translation and rotations """
⟨Initial import of modules 28b⟩
from mapper import *
square = larCuboids([1,1])
square = Model(square)
table = larApply( t(-.5,-.5) )(square)
chair = larApply( s(.35,.35) )(table)
chair = larApply( t(.75, 0) )(chair)
struct = Struct([table] + 4*[chair, r(PI/2)])
scene = evalStruct(struct)
VIEW(SKEL_1(STRUCT(CAT(AA(MKPOLS)(scene)))))
◇
```

Call of nested structures by reference Finally, a similar 2D example is given, by nesting one (or more) structures via separate definition and call by reference from the interior. Of course, a cyclic set of calls must be avoided, since it would result in a *non acyclic* multigraph of the structure network.

"test/py/mapper/test06.py" 21a ≡

```
""" Example of nested structures with translation and rotations """
⟨Initial import of modules 28b⟩
from mapper import *
square = larCuboids([1,1])
square = Model(square)
table = larApply( t(-.5,-.5) )(square)
chair = Struct([ t(.75, 0), s(.35,.35), table ])
struct = Struct( [t(2,1)] + [table] + 4*[r(PI/2), chair])
scene = evalStruct(struct)
VIEW(SKEL_1(STRUCT(CAT(AA(MKPOLS)(scene)))))
◇
```

"test/py/mapper/test08.py" 21b ≡

```
""" LAR model input and handling """
⟨Input of LAR architectural plan 21c⟩
dwelling = larApply(t(3,0))(Model((V,FV)))
print "\n dwelling =",dwelling
VIEW(EXPLODE(1.2,1.2,1)(MKPOLS(dwelling)))
plan = Struct([dwelling,s(-1,1),dwelling])
VIEW(EXPLODE(1.2,1.2,1)(CAT(AA(MKPOLS)(evalStruct(plan)))))
◇
```

⟨Input of LAR architectural plan 21c⟩ ≡

```
⟨Initial import of modules 28b⟩
from mapper import *
V = [[3,-3],
[9,-3],[0,0],[3,0],[9,0],[15,0],
```

```
[3,3],[6,3],[9,3],[15,3],[21,3],
[0,9],[6,9],[15,9],[18,9],[0,13],
[6,13],[9,13],[15,13],[18,10],[21,10],
[18,13],[6,16],[9,16],[9,17],[15,17],
[18,17],[-3,24],[6,24],[15,24],[-3,13]]
FV = [
[22,23,24,25,29,28], [15,16,22,28,27,30], [18,21,26,25],
[13,14,19,21,18], [16,17,23,22], [11,12,16,15],
[9,10,20,19,14,13], [2,3,6,7,12,11], [0,1,4,8,7,6,3],
[4,5,9,13,18,17,16,12,7,8],[17,18,25,24,23]]
◇
```

Macro referenced in 21b.

Transformation of Struct object to LAR model pair The following test application first generates a grid 3×3 of LAR cubes, extracts its boundary cells as `BV`, then produces a `struct` object with 30 translated instances of it, and finally transforms the `struct` object into a LAR pair `W,FW`. Let us notice that due to the assembly process, some 2-cells in `FW` are doubled.

"test/py/mapper/test09.py" 22a ≡

```
""" Transformation of Struct object to LAR model pair """
import sys
""" import modules from larcc/lib """
sys.path.insert(0, 'lib/py/')
from larcc import *
from mapper import evalStruct

⟨Transform Struct object to LAR model pair 22b⟩
◇
```

The actual generation of the structure and its transformation to a LAR model pair is actually performed in the following macro.

⟨Transform Struct object to LAR model pair 22b⟩ ≡

```
""" Generation of Struct object and transform to LAR model pair """
cubes = larCuboids([3,3,3],True)
V = cubes[0]
FV = cubes[1][-2]
CV = cubes[1][-1]
bcells = boundaryCells(CV,FV)
BV = [FV[f] for f in bcells]
VIEW(EXPLODE(1.2,1.2,1.2)(MKPOLS((V,BV))))

block = Model((V,BV))
struct = Struct(30*[block, t(3,0,0)])
```

```
W,FW = struct2lar(struct)

VIEW(EXPLODE(1.2,1.2,1.2)(MKPOLS((W,FW))))
◇
```

Macro referenced in 22a, 23a.

Remove double instances of cells

"test/py/mapper/test10.py" 23a ≡

```
""" Remove double instances of cells (and the unused vertices) """
import sys
""" import modules from larcc/lib """
sys.path.insert(0, 'lib/py/')
from larcc import *
from mapper import evalStruct

⟨Transform Struct object to LAR model pair 22b⟩
⟨Remove the double instances of cells 23b⟩
VIEW(EXPLODE(1.2,1.2,1.2)(MKPOLS((W,FW))))

⟨Remove the unused vertices 23c⟩
◇
```

The actual removal of double cells (useful in several applications, and in particular in the extraction of boundary models from 3D medical images) is performed by first generating a dictionary of cells, using as key the tuple given by the cells themselves, and then removing those discovered having a double instance. The algorithm is extremely simple, and its implementation, given below, is straightforward.

⟨Remove the double instances of cells 23b⟩ ≡

```
""" Remove the double instances of cells """
cellDict = defaultdict(list)
for k,cell in enumerate(FW):
   cellDict[tuple(cell)] += [k]
FW = [list(key) for key in cellDict.keys() if len(cellDict[key])==1]
◇
```

Macro referenced in 23a.

⟨Remove the unused vertices 23c⟩ ≡

```
""" Remove the unused vertices """
print "len(W) =",len(W)
V,FV = larRemoveVertices(W,FW)
print "len(V) =",len(V)
◇
```

Macro referenced in 23a.

⟨Remove the unused vertices from a LAR model pair 24a⟩ ≡

```
""" Remove the unused vertices """
def larRemoveVertices(V,FV):
   vertDict = dict()
   index,defaultValue,FW,W = -1,-1,[],[]

   for k,incell in enumerate(FV):
      outcell = []
      for v in incell:
         key = vcode(V[v])
         if vertDict.get(key,defaultValue) == defaultValue:
            index += 1
            vertDict[key] = index
            outcell += [index]
            W += [eval(key)]
         else:
            outcell += [vertDict[key]]
      FW += [outcell]
   return W,FW
```

◇

Macro referenced in 24b.

6 Computational framework

6.1 Exporting the library

"lib/py/mapper.py" 24b ≡

```
""" Mapping functions and primitive objects """
```

⟨Initial import of modules 28b⟩
⟨Affine transformations of d-points 28c⟩
⟨Generate a simplicial decomposition ot the $[0,1]^d$ domain 2⟩
⟨Scaled simplicial decomposition ot the $[0,1]^d$ domain 3a⟩
⟨Create a dictionary with key the point location 4a⟩
⟨Primitive mapping function 3b⟩
⟨Basic tests of mapper module 26a⟩
⟨Circle centered in the origin 5a⟩
⟨Helix curve about the z axis 5b⟩
⟨Disk centered in the origin 6a⟩
⟨Helicoid about the z axis 6b⟩
⟨Ring centered in the origin 6c⟩
⟨Spherical surface of given radius 7b⟩
⟨Cylinder surface with z axis 7a⟩
⟨Toroidal surface of given radiuses 8a⟩
⟨Half-toroidal surface of given radiuses 8b⟩
⟨Solid box of given extreme vectors 8c⟩

⟨ Solid Sphere of given radius 9b ⟩
⟨ Solid helicoid about the z axis 9a ⟩
⟨ Solid cylinder of given radius and height 9c ⟩
⟨ Solid torus of given radiuses 10c ⟩
⟨ Solid pizza of given radiuses 11a ⟩
⟨ Hollow cylinder of given radiuses and height 10a ⟩
⟨ Hollow sphere of given radiuses 10b ⟩
⟨ Translation matrices 12a ⟩
⟨ Scaling matrices 12b ⟩
⟨ Rotation matrices 12c ⟩
⟨ Embedding and projecting a geometric model 4b ⟩
⟨ Apply an affine transformation to a LAR model 11b ⟩
⟨ Check for dimension of a structure element (Verts or V) 18a ⟩
⟨ Traversal of a scene multigraph 19a ⟩
⟨ Symbolic utility to represent points as strings 29 ⟩
⟨ Remove the unused vertices from a LAR model pair 24a ⟩
⋄

6.2 Examples

3D rotation about a general axis The approach used by `lar-cc` to specify a general 3D rotation is shown in the following example, by passing the rotation function `r` the components `a,b,c` of the unit vector `axis` scaled by the rotation `angle`.

"test/py/mapper/test02.py" 25a ≡

```
""" General 3D rotation of a toroidal surface """
⟨Initial import of modules 28b⟩
from mapper import *
model = checkModel(larToroidal([0.2,1])())
angle = PI/2; axis = UNITVECT([1,1,0])
a,b,c = SCALARVECTPROD([ angle, axis ])
model = larApply(r(a,b,c))(model)
VIEW(STRUCT(MKPOLS(model)))
⋄
```

3D elementary rotation of a 2D circle A simpler specification is needed when the 3D rotation is about a coordinate axis. In this case the rotation angle can be directly given as the unique non-zero parameter of the the rotation function `r`. The rotation axis (in this case the x one) is specified by the non-zero (angle) position.

"test/py/mapper/test03.py" 25b ≡

```
""" Elementary 3D rotation of a 2D circle """
⟨Initial import of modules 28b⟩
from mapper import *
model = checkModel(larCircle(1)())
```

```
model = larEmbed(1)(model)
model = larApply(r(PI/2,0,0))(model)
VIEW(STRUCT(MKPOLS(model)))
```
◇

6.3 Tests about domain

Mapping domains The generations of mapping domains of different dimension (1D, 2D, 3D) is shown below.

⟨Basic tests of mapper module 26a⟩ ≡
```
if __name__=="__main__":
   V,EV = larDomain([5])
   VIEW(EXPLODE(1.5,1.5,1.5)(MKPOLS((V,EV))))
   V,EV = larIntervals([24])([2*PI])
   VIEW(EXPLODE(1.5,1.5,1.5)(MKPOLS((V,EV))))

   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))))
```
◇

Macro referenced in 24b.

Testing some primitive object generators The various model generators given in Section 3 are tested here, including LAR 2D circle, disk, and ring, as well as the 3D cylinder, sphere, and toroidal surfaces, and the solid objects ball, rod, crown, pizza, and torus.

"test/py/mapper/test01.py" 26b ≡
```
""" Circumference of unit radius """
⟨Initial import of modules 28b⟩
from mapper import *
model = larCircle(1)()
VIEW(EXPLODE(1.2,1.2,1.2)(MKPOLS(model)))
model = larHelix(1,0.5,4)()
VIEW(EXPLODE(1.2,1.2,1.2)(MKPOLS(model)))
model = larDisk(1)([36,4])
VIEW(EXPLODE(1.2,1.2,1.2)(MKPOLS(model)))
model = larHelicoid(1,0.5,0.1,10)()
```

```
VIEW(EXPLODE(1.2,1.2,1.2)(MKPOLS(model)))
model = larRing(.9, 1.)([36,2])
VIEW(EXPLODE(1.2,1.2,1.2)(MKPOLS(model)))
model = larCylinder(.5,2.)([32,1])
VIEW(STRUCT(MKPOLS(model)))
model = larSphere(1,PI/6,PI/4)([6,12])
VIEW(STRUCT(MKPOLS(model)))
model = larBall(1)()
VIEW(EXPLODE(1.2,1.2,1.2)(MKPOLS(model)))
model = larSolidHelicoid(0.2,1,0.5,0.5,10)()
VIEW(STRUCT(MKPOLS(model)))
model = larRod(.25,2.)([32,1])
VIEW(STRUCT(MKPOLS(model)))
model = larToroidal(0.5,2)()
VIEW(STRUCT(MKPOLS(model)))
model = larCrown(0.125,1)([8,48])
VIEW(STRUCT(MKPOLS(model)))
model = larPizza(0.05,1,PI/3)([8,48])
VIEW(STRUCT(MKPOLS(model)))
model = larTorus(0.5,1)()
VIEW(STRUCT(MKPOLS(model)))
model = larBox([-1,-1,-1],[1,1,1])
VIEW(STRUCT(MKPOLS(model)))
model = larHollowCyl(0.8,1,1,angle=PI/4)([12,2,2])
VIEW(STRUCT(MKPOLS(model)))
model = larHollowSphere(0.8,1,PI/6,PI/4)([6,12,2])
VIEW(STRUCT(MKPOLS(model)))
◇
```

6.4 Volumetric utilities

Limits of a LAR Model

⟨Model limits 27⟩ ≡

```
def larLimits (model):
   if isinstance(model,tuple):
      V,CV = model
      verts = scipy.asarray(V)
   else: verts = model.verts
   return scipy.amin(verts,axis=0).tolist(), scipy.amax(verts,axis=0).tolist()

assert larLimits(larSphere()()) == ([-1.0, -1.0, -1.0], [1.0, 1.0, 1.0])
◇
```

Macro never referenced.

Alignment

⟨ Alignment primitive 28a ⟩ ≡

```
def larAlign (args):
   def larAlign0 (args,pols):
      pol1, pol2 = pols
      box1, box2 = (larLimits(pol1), larLimits(pol2))
      print "box1, box2 =",(box1, box2)

   return larAlign0
◇
```

Macro never referenced.

A Utility functions

def FLATTEN(pol) temp = Plasm.shrink(pol,True) hpcList = [] for I in range(len(temp.childs)): g,vmat, hmat = temp.childs[I].g,temp.childs[I].vmat, temp.childs[I].hmat g.embed(vmat.dim) g.transform(vmat, hmat) hpcList += [Hpc(g)] return hpcList

VIEW(STRUCT(FLATTEN(pol)))

⟨ Initial import of modules 28b ⟩ ≡

```
from pyplasm import *
from scipy import *
import os,sys

""" import modules from larcc/lib """
sys.path.insert(0, 'lib/py/')
import lar2psm
from lar2psm import *
from simplexn import *
from larcc import *
from largrid import *
◇
```

Macro referenced in 20ab, 21ac, 24b, 25ab, 26b.

Affine transformations of points Some primitive maps of points to points are given in the following, including translation, rotation and scaling of array of points via direct transformation of their coordinates. Second-order functions are used in order to employ their curried version to transform geometric assemblies.

⟨ Affine transformations of d-points 28c ⟩ ≡

```
def larTranslate (tvect):
   def larTranslate0 (points):
      return [VECTSUM([p,tvect]) for p in points]
   return larTranslate0

def larRotate (angle):     # 2-dimensional !! TODO: n-dim
   def larRotate0 (points):
      a = angle
      return [[x*COS(a)-y*SIN(a), x*SIN(a)+y*COS(a)] for x,y in points]
   return larRotate0

def larScale (svect):
   def larScale0 (points):
      print "\n points =",points
      print "\n svect =",svect
      return [AA(PROD)(TRANS([p,svect])) for p in points]
   return larScale0
◇
```

Macro referenced in 24b.

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.

⟨Symbolic utility to represent points as strings 29⟩ ≡

```
""" TODO: use package Decimal (http://docs.python.org/2/library/decimal.html) """
PRECISION = 4

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

def fixedPrec(value):
   out = round(value*10**PRECISION)/10**PRECISION
   if out == -0.0: out = 0.0
   return str(out)

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))
◇
```

Macro referenced in 24b.

References

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

[Pao03] A. Paoluzzi, *Geometric programming for computer aided design*, John Wiley & Sons, Chichester, UK, 2003.