Hierarchical structures with LAR *

Alberto Paoluzzi

November 5, 2015

Contents

*This document is part of the *Linear Algebraic Representation with CoChains* (LAR-CC) framework [CL13]. November 5, 2015

1 Affine transformations

1.1 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.

1.2 Affine mapping

$\langle$ Apply an affine transformation to a LAR model 1 $\rangle \equiv$

```
""" Apply an affine transformation to a LAR model  """
from scipy import array

def larApply(affineMatrix):
    def larApply0(model):
        if isinstance(model,Model):
            V = scipy.dot(array([v+[1.0] for v in model.verts]), affineMatrix.T).tolist()
            V = [v[:-1] for v in V]
            CV = copy.copy(model.cells)
            return Model((V,CV))
        elif isinstance(model,tuple) or isinstance(model,list):
            if len(model)==2: V,CV = model
            elif len(model)==3: V,CV,FV = model
            V = scipy.dot([list(v)+[1.0] for v in V], affineMatrix.T).tolist()
            if len(model)==2: return [v[:-1] for v in V],CV
            elif len(model)==3: return [v[:-1] for v in V],CV,FV
    return larApply0
◇
```

Macro referenced in 17b.

1.3 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 2a ⟩ ≡

```
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 17b.

Scaling

⟨ Scaling matrices 2b ⟩ ≡

```
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 17b.

Rotation

⟨ Rotation matrices 3a ⟩ ≡

```
def r(*args):
    args = list(args)
    n = len(args)
    ⟨ plane rotation (in 2D) 3b ⟩
    ⟨ space rotation (in 3D) 3c ⟩
    return mat.view(Mat)
```

◇

Macro referenced in 17b.

⟨ plane rotation (in 2D) 3b ⟩ ≡

```
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 3a.

⟨ space rotation (in 3D) 3c ⟩ ≡

```
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) 3d ⟩
    ⟨ general rotations (in 3D) 3e ⟩
◇
```

Macro referenced in 3a.

⟨ elementary rotations (in 3D) 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;
◇
```

Macro referenced in 3c.

⟨ general rotations (in 3D) 3e ⟩ ≡

```
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 3c.

2 Structure types handling

In order to implement a structure as a list of models and transformations, we need to be able to distinguish between two different types of scipy arrays. The first type is the one of arrays of vertices, the second one is the matrix array used to represent the fine transformations.

2.1 `Mat` and `Verts` classes

⟨types Mat and Verts 4a⟩ ≡

```
""" class definitions for LAR """
import scipy
class Mat(scipy.ndarray): pass
class Verts(scipy.ndarray): pass
◇
```

Macro referenced in 17b.

2.2 `Model` class

⟨Model class 4b⟩ ≡

```
class Model:
    """ A pair (geometry, topology) of the LAR package """
    def __init__(self,(verts,cells)):
        self.n = len(verts[0])
        # self.verts = scipy.array(verts).view(Verts)
        self.verts = verts
        self.cells = cells
    def __getitem__(self,i):
        return list((self.verts,self.cells))[i]
◇
```

Macro referenced in 17b.

2.3 `Struct` iterable class

⟨Struct class 5⟩ ≡

```
""" Struct iterable class """
class Struct:
    """ The assembly type of the LAR package """
    def __init__(self,data=None,name=None,category=None):
        if data==None or data==[]:
            self.body = []
        else:
            self.body = [item for item in data if item != None]
            self.box = box(self)
            self.dim = len(self.box[0])
```

```
        if name != None:
            self.name = str(name)
        else:
            self.name = str(id(self))
        if category != None:
            self.category = str(category)
        else:
            self.category = "feature"
    def __name__(self):
        return self.name
    def __category__(self):
        return self.category
    def __iter__(self):
        return iter(self.body)
    def __len__(self):
        return len(list(self.body))
    def __getitem__(self,i):
        return list(self.body)[i]
    def __setitem__(self,i,value):
        self.body[i] = value
    def __print__(self):
        return "<Struct name: %s>" % self.__name__()
    def __repr__(self):
        return "<Struct name: %s>" % self.__name__()
        #return "'Struct(%s,%s)'" % (str(self.body),str(str(self.__name__())))
    def set_name(self,name):
        self.name = str(name)
    def clone(self,i=0):
        from copy import deepcopy
        newObj = deepcopy(self)
        if i != 0: newObj.name = self.name + "_" + str(i)
        return newObj
    def set_category(self,category):
        self.category = str(category)
    def boundary(self):
        data = struct2lar(self)
        if len(data) == 3:
            V,FV,EV = data
            #import pdb; pdb.set_trace()
            return V,FV,EV
        else:
            return "<Struct name: %s> boundary non computable" % self.__name__()
    def draw(self,color=WHITE,scaling=1,metric=ID):
        vmin,vmax = self.box
        delta = VECTDIFF([vmax,vmin])
        point = CCOMB(self.box)
```

```
        scalingFactor = scaling*delta[0]/20.
        text = TEXTWITHATTRIBUTES (TEXTALIGNMENT='centre', TEXTANGLE=0,
                    TEXTWIDTH=0.1*scalingFactor,
                    TEXTHEIGHT=0.2*scalingFactor,
                    TEXTSPACING=0.025*scalingFactor)
        point = metric([point])[0]
        return T([1,2,3])(point)(COLOR(color)(text(self.name)))
◇
```

Macro referenced in 17b.

2.4 Struct containment box

⟨ Computation of the containment box of a Lar Struct or Model 6 ⟩ ≡

```
""" Computation of the containment box of a Lar Struct or Model """
import copy
def box(model):
    if isinstance(model,Mat): return []
    elif isinstance(model,Struct):
        dummyModel = copy.deepcopy(model)
        dummyModel.body = [term if (not isinstance(term,Struct)) else [term.box,[[0,1]]]  for t
        listOfModels = evalStruct( dummyModel )
        #dim = checkStruct(listOfModels)
        theMin,theMax = box(listOfModels[0])
        for theModel in listOfModels[1:]:
            modelMin, modelMax = box(theModel)
            theMin = [val if val<theMin[k] else theMin[k] for k,val in enumerate(modelMin)]
            theMax = [val if val>theMax[k] else theMax[k] for k,val in enumerate(modelMax)]
        return [theMin,theMax]
    elif isinstance(model,Model):
        V = model.verts
    elif (isinstance(model,tuple) or isinstance(model,list)) and (len(model)==2 or len(model)==
        V = model[0]
    coords = TRANS(V)
    theMin = [min(coord) for coord in coords]
    theMax = [max(coord) for coord in coords]
    return [theMin,theMax]
◇
```

Macro referenced in 17b.

3 Structure to LAR conversion

3.1 Remove duplicate faces

⟨ Remove duplicate faces 7a ⟩ ≡

```
""" Remove duplicate faces  """
from collections import defaultdict
def removeDups (CW):
    CW = list(set(AA(tuple)(CW)))
    CWs = list(set(AA(tuple)  (AA(sorted)(CW))  ))
    no_duplicates = defaultdict(list)
    for f in CWs: no_duplicates[f] = []
    for f in CW:
        no_duplicates[tuple(sorted(f))] += [f]
    CW = [f[0] for f in no_duplicates.values()]
    return CW
◇
```

Macro referenced in 17b.

3.2 Structure to pair (Vertices,Cells) conversion

⟨Structure to pair (Vertices,Cells) conversion 7b⟩ ≡

```
""" Structure to pair (Vertices,Cells) conversion """

def struct2lar(structure,metric=ID):
    listOfModels = evalStruct(structure)
    vertDict = dict()
    index,defaultValue,CW,W,FW = -1,-1,[],[],[]

    for model in listOfModels:
        if isinstance(model,Model):
            V,FV = model.verts,model.cells
        elif (isinstance(model,tuple) or isinstance(model,list)):
            if len(model)==2: V,FV = model
            elif len(model)==3: V,FV,EV = model
        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]]
            CW += [outcell]
        if len(model)==3:
            for k,incell in enumerate(EV):
                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]

    if ((isinstance(model,tuple) or isinstance(model,list)) and len(model)==2) or (
        (isinstance(model,Model) and model.n==2)):
        if len(CW[0])==2:
            CW = list(set(AA(tuple)(AA(sorted)(CW))))
        else: CW = removeDups(CW)
        return metric(W),CW
    if ((isinstance(model,tuple) or isinstance(model,list)) and len(model)==3) or (
        (isinstance(model,Model) and model.n==3)):
        FW = list(set(AA(tuple)(AA(sorted)(FW))))
        CW = removeDups(CW)
        return metric(W),CW,FW
◇
```

Macro referenced in 17b.

3.3 Model simplification

Remove double instances of cells

"test/py/larstruct/test10.py" 8 ≡

```
""" Remove double instances of cells (and the unused vertices) """
from larlib import *

⟨Transform Struct object to LAR model pair 20c⟩
⟨Remove the double instances of cells 9a⟩
VIEW(EXPLODE(1.2,1.2,1.2)(MKPOLS((W,FW))))

⟨Remove the unused vertices 9b⟩
◇
```

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 9a⟩ ≡

```
""" 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 8.

⟨ Remove the unused vertices 9b ⟩ ≡

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

Macro referenced in 8.

⟨ Remove the unused vertices from a LAR model pair 9c ⟩ ≡

```
""" 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 17b.

3.4 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 10⟩ ≡

```
def larEmbed(k):
    def larEmbed0(model):
        if len(model)==2: V,CV = model
        elif len(model)==3: V,CV,FV = model
        if k>0:
            V = [v+[0.]*k for v in V]
        elif k<0:
            V = [v[:-k] for v in V]
        if len(model)==2: return V,CV
        elif len(model)==3: return V,CV,FV
    return larEmbed0

def larEmbed(k):
    def larEmbed0(model):
        if k>0:
            model[0] = [v+[0.]*k for v in model[0]]
        elif k<0:
            model[0] = [v[:-k] for v in model[0]]
        return model
    return larEmbed0
```

◇

Macro referenced in 17b.

4 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.

4.1 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.

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.

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].

4.1.1 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 usual 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 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 12⟩ ≡

```
""" Emulation of scene multigraph traversal """
from larlib 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 13a⟩ ≡

```
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 13b ⟩ ≡

```
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) 14a⟩ ≡

```
⟨Flatten a list 14b⟩
def checkStruct(lst):
    """ Return the common dimension of structure elements.

        TODO: aggiungere test sulla dimensione minima delle celle (legata a quella di immersion
    """
    obj = lst[0]
    if (isinstance(obj,tuple) or isinstance(obj,list)):
        dim = len(obj[0][0])
    elif isinstance(obj,Model):
        dim = obj.n
    elif isinstance(obj,Mat):
        dim = obj.shape[0]-1
    elif isinstance(obj,Struct):
        dim = len(obj.box[0])
    return dim
◇
```

Macro referenced in 17b.

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 14b⟩ ≡

```
""" 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]

#  import itertools
#  chain = itertools.chain.from_iterable([[1,2],[3],[5,89],[],[6]])
#  print(list(chain))
#  [1, 2, 3, 5, 89, 6]    ###  TODO: Bug coi dati sopra?
◇
```

Macro referenced in 14a.

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 coordinates 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 15 ⟩ ≡

```
""" Traversal of a scene multigraph """
⟨Structure traversal algorithm 16a⟩
def evalStruct(struct):
    dim = checkStruct(struct.body)
    CTM, stack = scipy.identity(dim+1), []
    scene = traversal(CTM, stack, struct, [])
    return scene
◇
```

Macro referenced in 17b.

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 16a ⟩ ≡

```
def traversal(CTM, stack, obj, scene=[]):
    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 or len(obj[i])==3):
            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 15.

Structure embedding algorithm

⟨Embed a struct object 16b⟩ ≡

```
""" embed a struct object """
⟨Structure embedding algorithm 17a⟩

def embedStruct(n):
    def embedStruct0(struct,suffix="New"):
        if n==0:
            return struct, len(struct.box[0])
        cloned = Struct()
        cloned.box = hstack((struct.box, [n*[0],n*[0]])).tolist()
        cloned.name = str(id(cloned))  #struct.name+suffix
        cloned.category = struct.category
        cloned.dim = struct.dim + n
        cloned = embedTraversal(cloned,struct,n,suffix)
        return cloned
    return embedStruct0
◇
```

Macro referenced in 17b.

Structure embedding algorithm

⟨Structure embedding algorithm 17a⟩ ≡

```
""" Structure embedding algorithm """
def embedTraversal(cloned, obj,n,suffix):
    for i in range(len(obj)):
        if isinstance(obj[i],Model):
```

```
            cloned.body += [obj[i]]
        elif (isinstance(obj[i],tuple) or isinstance(obj[i],list)) and (
                len(obj[i])==2):
            V,EV = obj[i]
            V = [v+n*[0.0] for v in V]
            cloned.body  += [(V,EV)]
        elif (isinstance(obj[i],tuple) or isinstance(obj[i],list)) and (
                len(obj[i])==3):
            V,FV,EV = obj[i]
            V = [v+n*[0.0] for v in V]
            cloned.body  += [(V,FV,EV)]
        elif isinstance(obj[i],Mat):
            mat = obj[i]
            d,d = mat.shape

            newMat = scipy.identity(d+n*1)
            for h in range(d-1):
                for k in range(d-1):
                    newMat[h,k] = mat[h,k]
                newMat[h,d-1+n*1] = mat[h,d-1]
            cloned.body  +=  [newMat.view(Mat)]

        elif isinstance(obj[i],Struct):
            newObj = Struct()
            newObj.box = hstack((obj[i].box, [n*[0],n*[0]]))
            newObj.name = obj[i].name+suffix
            newObj.category = obj[i].category
            cloned.body  += [embedTraversal(newObj, obj[i], n, suffix)]
    return cloned
◇
```

Macro referenced in 16b.

5 Larstruct exporting

Here we assemble top-down the `lar2psm` module, by orderly listing the functional parts it is composed of. Of course, this one is the module version corresponding to the current state of the system, i.e. to a very initial state. Other functions will be added when needed.

"larlib/larlib/larstruct.py" 17b ≡

```
"""Module with functions needed to interface LAR with pyplasm via Struct"""
from larlib import *

⟨ Symbolic utility to represent points as strings 21d ⟩
⟨ Translation matrices 2a ⟩
⟨ Scaling matrices 2b ⟩
```

⟨ Rotation matrices 3a ⟩
⟨ Embedding and projecting a geometric model 10 ⟩
⟨ Apply an affine transformation to a LAR model 1 ⟩
⟨ Check for dimension of a structure element (Verts or V) 14a ⟩
⟨ Remove duplicate faces 7a ⟩
⟨ Remove the unused vertices from a LAR model pair 9c ⟩
⟨ Traversal of a scene multigraph 15 ⟩
⟨ types Mat and Verts 4a ⟩
⟨ Model class 4b ⟩
⟨ Struct class 5 ⟩
⟨ Structure to pair (Vertices,Cells) conversion 7b ⟩
⟨ Embedding and projecting a geometric model 10 ⟩
⟨ Embed a struct object 16b ⟩
⟨ Computation of the containment box of a Lar Struct or Model 6 ⟩
◇

6 Examples

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/larstruct/test04.py" 18 ≡

```
""" Example of non-nested structure with translation and rotations """
from larlib 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/larstruct/test05.py" 19a ≡

```
""" Example of non-nested structure with translation and rotations """
from larlib 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/larstruct/test06.py" 19b ≡

```
""" Example of nested structures with translation and rotations """
from larlib 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])
struct = Struct(10*[struct,t(0,2.5)])
struct = Struct(10*[struct,t(3,0)])
scene = evalStruct(struct)
VIEW(SKEL_1(STRUCT(CAT(AA(MKPOLS)(scene)))))
```
◇

"test/py/larstruct/test08.py" 19c ≡

```
""" LAR model input and handling """
from larlib import *

⟨Input of LAR architectural plan 20a⟩
dwelling = larApply(t(3,0))(Model((V,FV)))
print "\n dwelling =",dwelling
VIEW(EXPLODE(1.2,1.2,1)(MKPOLS((dwelling.verts,dwelling.cells))))
VIEW(EXPLODE(1.2,1.2,1)(MKPOLS((dwelling.verts,EV))))
plan = Struct([dwelling,s(-1,1),dwelling])
VIEW(EXPLODE(1.2,1.2,1)(CAT(AA(MKPOLS)(evalStruct(plan)))))
```
◇

⟨Input of LAR architectural plan 20a⟩ ≡

```
""" Input of LAR architectural plan """
from larlib 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]]

polylines = lar2polylines((V,FV))
lines = CAT([zip(polyline[:-1],polyline[1:]) for polyline in polylines])
verts = dict(zip(AA(vcode)(V),range(len(V))))
edges = [tuple(sorted([verts[vcode(v1)], verts[vcode(v2)]])) for v1,v2 in lines]
EV = list(set(edges))
◇
```

Macro referenced in 19c.

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/larstruct/test09.py" 20b ≡

```
""" Transformation of Struct object to LAR model pair """
from larlib import *

⟨Transform Struct object to LAR model pair 20c⟩
◇
```

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 20c⟩ ≡

```
""" Generation of Struct object and transform to LAR model pair """
cubes = larCuboids([10,10,10],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(10*[block, t(10,0,0)])
struct = Struct(10*[struct, t(0,10,0)])
struct = Struct(3*[struct, t(0,0,10)])
W,FW = struct2lar(struct)

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

Macro referenced in 8, 20b.

.1 Importing a generic module

First we define a parametric macro to allow the importing of `larcc` modules from the project repository `lib/py/`. When the user needs to import some project's module, she may call this macro as done in Section **??**.

⟨Import the module 21a⟩ ≡

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

Macro referenced in 21b.

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

⟨Function to import a generic module 21b⟩ ≡

```
def importModule(moduleName):
    ⟨Import the module (21c moduleName ) 21a⟩
◇
```

Macro never referenced.

.2 Numeric utilities

A small set of utility functions is used to transform a *point* representation, given 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 21d⟩ ≡

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

```
#PRECISION = 4.

def verySmall(number): return abs(number) < 10**-(PRECISION)

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

def fixedPrec(PRECISION):
    def fixedPrec0(value):
        out = round(value*10**(PRECISION))/10**(PRECISION)
        if out == -0.0: out = 0.0
        return str(out)
    return fixedPrec0

def vcode (vect,PRECISION=4):
    """
    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(PRECISION))(vect))
◇
```

Macro referenced in 17b.

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.