Accelerated intersection of geometric objects *

Alberto Paoluzzi

November 19, 2015

Abstract

This module contains the first experiments of a parallel implementation of the intersection of (multidimensional) geometric objects. The first installment is being oriented to the intersection of line segment in the 2D plane. A generalization of the algorithm, based on the classification of the containment boxes of the geometric values, will follow quickly.

Contents

*This document is part of the *Linear Algebraic Representation with CoChains* (LAR-CC) framework [CL13]. November 19, 2015

1 Introduction

An easily parallelizable implementation of the accelerated intersection of geometric objects is given in this module. Our first aim is to implement a specialized version for simplices, that generalizes the nD-trees of points (that are 0-simplices), to $(d-1)$-dimensional simplices in d-space, starting with the intersection of line segments in the plane. Our plan is to follow with an implementation for intersection of general *non convex* sets.

2 Implementation

The first implementation of this module concerns the computation of the intersection points among a set of line segment in the 2D plane. The containment boxes of the input segments are iteratively classified against the 1-dimensional centroid of smaller and smaller buckets of data.

At the end of the classification, where the same geometric object may be inserted in several different buckets, a *brute-force* intersection is applied to each final subset. Finally, the duplicated intersection points are removed, and a 1-dimensional LAR data structure is generated, with 1-cells given by the split line segments.

A complete LAR of the plane partition generated by the arrangment of lines is then computed by: (a) generating the maximal 2-connected components of such 1-dimensional graph; and (b) by traversing in counter-clockwise order the generated subgraphs to report the 2-dimensional cells of the plane partition.

The splitting algorithm may be easily parallelized, since both during their generation and at the end of this one, the various buckets of data can be dispatched to different processors for independent computation, followed by elimination of duplicates. In particular, a standard *map-reduce* software infrastructure may be used for this parallelization purpose.

2.1 Construction of independent buckets

Containment boxes Given as input a list `randomLineArray` of pairs of 2D points, the function `containment2DBoxes` returns, in the same order, the list of *containment boxes* of the input lines. A *containment box* of a geometric object of dimension d is defined as the minimal d-cuboid, equioriented with the reference frame, that contains the object. For a 2D line it is given by the tuple $(x1, y1, x2, y2)$, where $(x1, y1)$ is the point of minimal coordinates, and $(x2, y2)$ is the point of maximal coordinates.

⟨ Containment boxes 2a ⟩ ≡

```
    """ Containment boxes """
    def containment2DBoxes(randomLineArray):
        boxes = [eval(vcode([min(x1,x2),min(y1,y2),max(x1,x2),max(y1,y2)]))
                for ((x1,y1),(x2,y2)) in randomLineArray]
        return boxes
    ◇
```

Macro referenced in 34.

Splitting the input above and below a threshold

⟨ Splitting the input above and below a threshold 2b ⟩ ≡

```
    """ Splitting the input above and below a threshold """
    def splitOnThreshold(boxes,subset,coord):
        theBoxes = [boxes[k] for k in subset]
        threshold = centroid(theBoxes,coord)
        ncoords = len(boxes[0])/2
        a = coord%ncoords
        b = a+ncoords
        below,above = [],[]
        for k in subset:
            if boxes[k][a] <= threshold: below += [k]
        for k in subset:
            if boxes[k][b] >= threshold: above += [k]
        return below,above
    ◇
```

Macro referenced in 34.

Iterative splitting of box buckets

⟨ Iterative splitting of box buckets 3a ⟩ ≡

```
    """ Iterative splitting of box buckets """
    def splitting(bucket,below,above, finalBuckets,splittingStack):
        if (len(below)<4 and len(above)<4) or len(set(bucket).difference(below))<7 \
            or len(set(bucket).difference(above))<7:
            finalBuckets.append(below)
            finalBuckets.append(above)
        else:
            splittingStack.append(below)
            splittingStack.append(above)

    def geomPartitionate(boxes,buckets):
        geomInters = [set() for h in range(len(boxes))]
        for bucket in buckets:
            for k in bucket:
                geomInters[k] = geomInters[k].union(bucket)
```

```
        for h,inters in enumerate(geomInters):
            geomInters[h] = geomInters[h].difference([h])
        return AA(list)(geomInters)

    def boxBuckets(boxes):
        bucket = range(len(boxes))
        splittingStack = [bucket]
        finalBuckets = []
        while splittingStack != []:
            bucket = splittingStack.pop()
            below,above = splitOnThreshold(boxes,bucket,1)
            below1,above1 = splitOnThreshold(boxes,above,2)
            below2,above2 = splitOnThreshold(boxes,below,2)
            splitting(above,below1,above1, finalBuckets,splittingStack)
            splitting(below,below2,above2, finalBuckets,splittingStack)
            finalBuckets = list(set(AA(tuple)(finalBuckets)))
        parts = geomPartitionate(boxes,finalBuckets)
        return AA(sorted)(parts)
        #return finalBuckets
```
◇

Macro referenced in 34.

2.2 Brute force intersection within the buckets

Intersection of two line segments

⟨ Intersection of two line segments 3b ⟩ ≡

```
    """ Intersection of two line segments """
    def segmentIntersect(boxes,lineArray,pointStorage):
        def segmentIntersect0(h):
            p1,p2 = lineArray[h]
            line1 = '['+ vcode(p1) +','+ vcode(p2) +']'
            (x1,y1),(x2,y2) = p1,p2
            B1,B2,B3,B4 = boxes[h]
            def segmentIntersect1(k):
                p3,p4 = lineArray[k]
                line2 = '['+ vcode(p3) +','+ vcode(p4) +']'
                (x3,y3),(x4,y4) = p3,p4
                b1,b2,b3,b4 = boxes[k]
                if not (b3<B1 or B3<b1 or b4<B2 or B4<b2):
                #if True:
                    m23 = mat([p2,p3])
                    m14 = mat([p1,p4])
                    m = m23 - m14
                    v3 = mat([p3])
                    v1 = mat([p1])
```

```
                    v = v3-v1
                    a=m[0,0]; b=m[0,1]; c=m[1,0]; d=m[1,1];
                    det = a*d-b*c
                    if det != 0:
                        m_inv = mat([[d,-b],[-c,a]])*(1./det)
                        alpha, beta = (v*m_inv).tolist()[0]
                        #alpha, beta = (v*m.I).tolist()[0]
                        if -0.0<=alpha<=1 and -0.0<=beta<=1:
                            pointStorage[line1] += [alpha]
                            pointStorage[line2] += [beta]
                            return list(array(p1)+alpha*(array(p2)-array(p1)))
                return None
            return segmentIntersect1
        return segmentIntersect0
    ◇
```

Macro referenced in 34.

Brute force bucket intersection

⟨ Brute force bucket intersection 4 ⟩ ≡

```
    """ Brute force bucket intersection """
    def lineBucketIntersect(boxes,lineArray, h,bucket, pointStorage):
        intersect0 = segmentIntersect(boxes,lineArray,pointStorage)
        intersectionPoints = []
        intersect1 = intersect0(h)
        for line in bucket:
            point = intersect1(line)
            if point != None:
                intersectionPoints.append(eval(vcode(point)))
        return intersectionPoints
    ◇
```

Macro referenced in 34.

Accelerate intersection of lines

⟨ Accelerate intersection of lines 5 ⟩ ≡

```
    """ Accelerate intersection of lines """
    def lineIntersection(lineArray):
        lineArray = [line for line in lineArray if len(line)>1]

        from collections import defaultdict
        pointStorage = defaultdict(list)
        for line in lineArray:
            print "line =",line
            p1,p2 = line
```

```
            key = '['+ vcode(p1) +','+ vcode(p2) +']'
            pointStorage[key] = []

        boxes = containment2DBoxes(lineArray)
        buckets = boxBuckets(boxes)
        intersectionPoints = set()
        for h,bucket in enumerate(buckets):
            pointBucket = lineBucketIntersect(boxes,lineArray, h,bucket, pointStorage)
            intersectionPoints = intersectionPoints.union(AA(tuple)(pointBucket))

        frags = AA(eval)(pointStorage.keys())
        params = AA(COMP([sorted,list,set,tuple,eval,vcode]))(pointStorage.values())

        return intersectionPoints,params,frags  ### GOOD: 1, WRONG: 2 !!!
```
◇

Macro referenced in 34.

2.3 Generation of LAR representation of split segments

The function `lines2lar` is used to generate a 1-dimensional LAR complex from an array of lines, i.e. of pairs of 2D points. For every *line* in `frags` is computed an *ordered* list `outline` of *symbolic* intersection points, including the first and last vertex of the line, and every interior point generated by the list `params[k]`.

Then, for every symbolic representation `key` of a point in `outline`, a dictionary vertex is either created or retrieved, and a corresponding edge is orderly created, using the index of the point. At the same time, the vertices created in this way are accumulated within the `V` array. Finally, each edge in EV is extended to contain a second vertex index using the subsequent edge.

The third stage finalizes the vertex set of the output LAR, by identifying the closest vertices, i.e. those at distance less or equal to the current resolution, set to `10**(-PRECISION)`, by searching via the `scipy.spatialKDTree` the pairs of vertices at less than this distance.

A fourth stage identifies the possibly duplicated edges. Some of these could appear, e.g., when importing a set of adjacent boxes from some drawing program, to generate an array of lines, to be mutually intersected and transformed into a LAR data structure.

Create the LAR of fragmented lines

⟨ Create the LAR of fragmented lines 6 ⟩ ≡

```
    """ Create the LAR of fragmented lines """
    from scipy import spatial

    def lines2lar(lineArray):
        _,params,frags = lineIntersection(lineArray)
        vertDict = dict()
```

```
    index,defaultValue,V,EV = -1,-1,[],[]

    for k,(p1,p2) in enumerate(frags):
        outline = [vcode(p1)]
        if params[k] != []:
            for alpha in params[k]:
                if alpha != 0.0 and alpha != 1.0:
                    p = list(array(p1)+alpha*(array(p2)-array(p1)))
                    outline += [vcode(p)]
        outline += [vcode(p2)]

        edge = []
        for key in outline:
            if vertDict.get(key,defaultValue) == defaultValue:
                index += 1
                vertDict[key] = index
                edge += [index]
                V += [eval(key)]
            else:
                edge += [vertDict[key]]
            EV.extend([[edge[k],edge[k+1]] for k,v in enumerate(edge[:-1])])

    model = (V,EV)
    return larSimplify(model)
```
◇

Macro referenced in 34.

2.4 Biconnected components of a 1-complex

An implementation of the Hopcroft-Tarjan algorithm [HT73] for computation of the biconnected components of a graph is given here.

Biconnected components

⟨ Biconnected components 7a ⟩ ≡

```
    """ Biconnected components """
```
⟨ Adjacency lists of 1-complex vertices 7b ⟩
⟨ Main procedure for biconnected components 7c ⟩
⟨ Hopcroft-Tarjan algorithm 8a ⟩
⟨ Output of biconnected components 8b ⟩
◇

Macro referenced in 34.

Adjacency lists of 1-complex vertices

⟨ Adjacency lists of 1-complex vertices 7b ⟩ ≡

```
""" Adjacency lists of 1-complex vertices """
def vertices2vertices(model):
    V,EV = model
    csrEV = csrCreate(EV)
    csrVE = csrTranspose(csrEV)
    csrVV = matrixProduct(csrVE,csrEV)
    cooVV = csrVV.tocoo()
    data,rows,cols = AA(list)([cooVV.data, cooVV.row, cooVV.col])
    triples = zip(data,rows,cols)
    VV = [[] for k in range(len(V))]
    for datum,row,col in triples:
        if row != col: VV[col] += [row]
    return AA(sorted)(VV)
```

◇

Macro referenced in 7a.

Main procedure for biconnected components

⟨ Main procedure for biconnected components 7c ⟩ ≡

```
""" Main procedure for biconnected components """
def biconnectedComponent(model):
    W,_ = model
    V = range(len(W))
    count = 0
    stack,out = [],[]
    visited = [None for v in V]
    parent = [None for v in V]
    d = [None for v in V]
    low = [None for v in V]
    for u in V: visited[u] = False
    for u in V: parent[u] = []
    VV = vertices2vertices(model)
    for u in V:
        if not visited[u]:
            DFV_visit( VV,out,count,visited,parent,d,low,stack, u )
    return W,[component for component in out if len(component) > 1]
```

◇

Macro referenced in 7a.

Hopcroft-Tarjan algorithm

⟨ Hopcroft-Tarjan algorithm 8a ⟩ ≡

```
""" Hopcroft-Tarjan algorithm """
def DFV_visit( VV,out,count,visited,parent,d,low,stack,u ):
    visited[u] = True
    count += 1
    d[u] = count
    low[u] = d[u]
    for v in VV[u]:
        if not visited[v]:
            stack += [(u,v)]
            parent[v] = u
            DFV_visit( VV,out,count,visited,parent,d,low,stack, v )
            if low[v] >= d[u]:
                out += [outputComp(stack,u,v)]
            low[u] = min( low[u], low[v] )
        else:
            if not (parent[u]==v) and (d[v] < d[u]):
                stack += [(u,v)]
                low[u] = min( low[u], d[v] )
```

◇

Macro referenced in 7a.

Output of biconnected components

⟨ Output of biconnected components 8b ⟩ ≡

```
""" Output of biconnected components """
def outputComp(stack,u,v):
    out = []
    while True:
        e = stack.pop()
        out += [list(e)]
        if e == (u,v): break
    return list(set(AA(tuple)(AA(sorted)(out))))
```

◇

Macro referenced in 7a.

2.5 2D cells from biconnected components

It is very easy, using the LAR representation of topology, to compute the 2-cells of the plane partitions (see Figures 1b and 1c) induced by the biconnected components extracted from a graph (1-complex).

In particular, let us consider the CSR (Compressed Sparse Row) representation of the characteristic matrix M_1, here usually denoted as `EV`, in order to remark that we represent the edges on the rows, and the vertices on the columns of the matrix. As such it is a binary matrix. So, we can readily reconstruct the topology of 2-cells by associating to

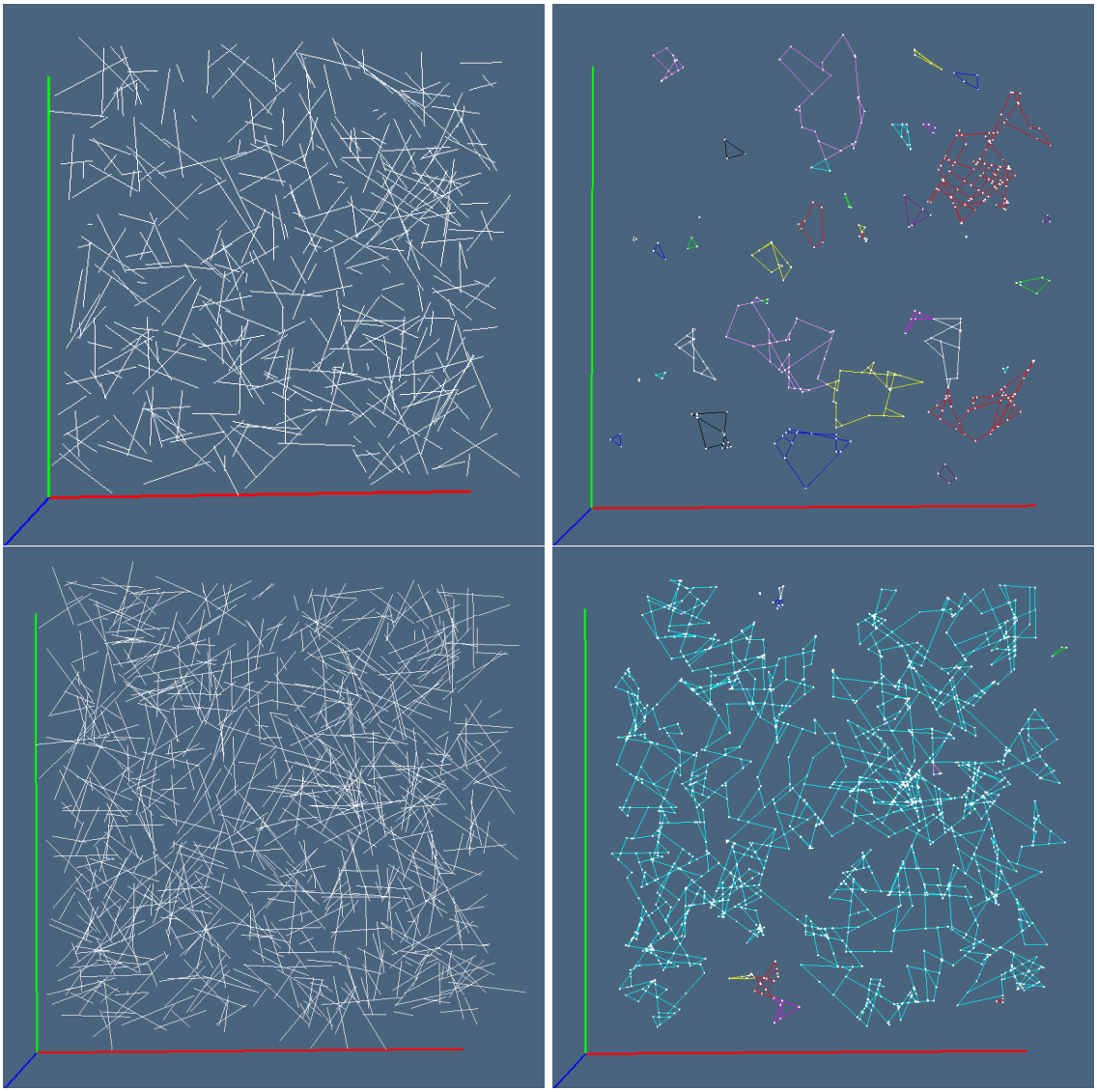

Figure 1: Two random line arrangements, and the biconnected components extracted by their LAR 1-complexes.

each non-zero (sparse) matrix element `angle_EV`(h,k) the angle in radians that the edge e_h forms with the orizontal line, when it incides on the vertex v_k.

Of course, if $e_h = (v_{k_1}, v_{k_2})$, then it will be

$$\texttt{angle\_EV}(h,k_2) = \texttt{angle\_EV}(h,k_1) + \pi = -\texttt{angle\_EV}(h,k_1)$$

Therefore, the columns of `angle_EV`, i.e. the rows of `angle_VE` := `angle_EV`^{t}, after being sorted on their angles α, and associated with the angle differences $\Delta\alpha$, will provide a basis of elementary $1-$*cochains* that evaluate to zero for each closed 1-cochain, i.e. for every cycle supported by the linear space of 1-chains on the given line arrangment.

Slope of edges

Circular ordering of edges around vertices

⟨ Slope of edges 9 ⟩ ≡

```
"""" Circular ordering of edges around vertices """"
def edgeSlopeOrdering(model):
    V,EV = model
    VE,VE_angle = invertRelation(EV),[]
    for v,ve in enumerate(VE):
        ve_angle = []
        if ve != []:
            for edge in ve:
                v0,v1 = EV[edge]
                if v == v0:     x,y = list(array(V[v1]) - array(V[v0]))
                elif v == v1:    x,y = list(array(V[v0]) - array(V[v1]))
                angle = math.atan2(y,x)
                ve_angle += [180*angle/PI]
        pairs = sorted(zip(ve_angle,ve))
        #VE_angle += [TRANS(pairs)[1]]
        VE_angle += [[pair[1] for pair in pairs]]
    return VE_angle
```

◇

Macro referenced in 34.

Ordered incidence relationship vertices to edges As we have seen, the `VE_angle` list of lists reports, for every vertex in `V`, the list of incident edges, *counterclockwise ordered* around the vertex. Therefore the `ordered_csrVE` function, given below, returns the "compressed sparse row" matrix, row-indexed by vertices and column-indexed by edges, and such that in position (v,e) contains the index ℓ of the next edge (after e, say) in the counterclockwise ordering of edges around v.

⟨ Ordered incidence relationship of vertices and edges 11a ⟩ ≡

```
""" Ordered incidence relationship of vertices and edges """
def ordered_csrVE(VE_angle):
    triples = []
    for v,ve in enumerate(VE_angle):
        n = len(ve)
        for k,edge in enumerate(ve):
            triples += [[v, ve[k], ve[ (k+1)%n ]]]
    csrVE = triples2mat(triples,shape="csr")
    return csrVE
```
◇

Macro referenced in 34.

Faces from biconnected components Since edges in the plane partition induced by a line arrangement are $(d-1)$-cells, they are located on the boundary of *two* d-cells (faces) of the partition. Hence, the traversal algorithm of the data structure storing the relevant information may be driven by signing the two extremes (vertices) of each edge as either already visited or not.

⟨ Faces from biconnected components 11b ⟩ ≡

```
""" Faces from biconnected components """

def firstSearch(visited):
    for edge,vertices in enumerate(visited):
        for v,vertex in enumerate(vertices):
            if visited[edge,v] == 0.0:
                visited[edge,v] = 1.0
                return edge,v
    return -1,-1

def facesFromComps(model):
    V,EV = model
    # Remove zero edges
    EV = list(set([ tuple(sorted([v1,v2])) for v1,v2 in EV if v1!=v2 ]))
    FV = []
    VE_angle = edgeSlopeOrdering((V,EV))
    csrEV = ordered_csrVE(VE_angle).T
    visited = zeros((len(EV),2))
    edge,v = firstSearch(visited)
    vertex = EV[edge][v]
    fv = []
    while True:
        if (edge,v) == (-1,-1):
            break #return [face for face in FV if face != None]
        elif (fv == []) or (fv[0] != vertex):
```

```
                fv += [vertex]
                nextEdge = csrEV[edge,vertex]
                v0,v1 = EV[nextEdge]

                try:
                    vertex, = set([v0,v1]).difference([vertex])
                except ValueError:
                    print 'ValueError: too many values to unpack'
                    break

                if v0==vertex: pos=0
                elif v1==vertex: pos=1

                if visited[nextEdge, pos] == 0:
                    visited[nextEdge, pos] = 1
                    edge = nextEdge
            else:
                FV += [fv]
                fv = []
                edge,v = firstSearch(visited)
                vertex = EV[edge][v]
            FV = [face for face in FV if face != None]
        return V,FV,EV
    ◇
```

Macro referenced in 34.

Txample The *ordered* `csrVE` (vertex-edge) matrix generated by the example of file `test/py/inters/test07.py` is shown in dense format in the example script below. Let us notice the each non-zero element $\texttt{csrVE}(k, h)$ stores the index of the previous edge inciding on the vertex v_k *before* the edge e_h. The traversal of the data structure is made accordingly, in order to extract the vertices of all the faces (minimal edge cycles) generated by a line arrangement in the plane.

⟨ Example of VE matrix with nextEdge indices 13 ⟩ ≡

```
csr2DenseMatrix(csrVE)
>>> array([
    [12,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0, 11,  0,  0,  0],
    [ 1,  2,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0],
    [ 0, 14,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  1,  0],
    [ 0,  0,  6,  5,  0,  2,  3,  0,  0,  0,  0,  0,  0,  0,  0,  0],
    [ 0,  0,  0, 10,  0,  0,  0,  0,  0,  3,  9,  0,  0,  0,  0,  0],
    [ 0,  0,  0,  0, 15,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  4],
    [ 0,  0,  0,  0, 12,  4,  0,  0,  0,  0,  0,  0,  5,  0,  0,  0],
    [ 0,  0,  0,  0,  0,  0,  7,  8,  6,  0,  0,  0,  0,  0,  0,  0],
```

```
    [ 0,  0,  0,  0,  0,  0,  0,  7,  0,  0,  0,  0,  0,  0,  0,  0],
    [ 0,  0,  0,  0,  0,  0,  0,  0, 10,  0,  8,  0,  0,  0,  0,  0],
    [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  9,  0,  0,  0,  0,  0,  0],
    [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0, 13,  0, 14, 11,  0],
    [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0, 15,  0, 13]])
```

◇

Macro never referenced.

2.6 Containments between non intersecting cycles

In this section we compute the containment relation between non-intersecting cycles generated on 2-faces by the incident faces. This step is preparatory to the representation of fragmented 2-faces — embedded in 2D — as LAR data structures, to be subsequently restored in the ambient 3D and sticked together to generate the the 2-skeleton of the Boolean complex.

For this purpose, the set of non-intersecting cycles, as lists of edges, are returned in the `EVs` array by the `biconnectedComponent` function. The pair `V,EVs` is therefore passed as input to the `latticeArray` function, that returns a dense matrix with elements in $\{-1, 0, 1\}$, where the element i, j either contains 0 if anyone (and hence all) of vertices of cycle j-th is *external* to cycle i-th, or contains 1 if anyone (and hence all) of vertices of cycle j-th is *internal* to cycle i-th, or finally contains -1 if anyone (and hence all) of vertices of cycle j-th is *on boundary* of cycle i-th. The last condition may hold only for diagonal elements i, j where $i = j$.

The returned `testArray` matrix of dimension $n \times n$, where n is the number of non-intersecting cycles on a fragmented 2D face, is the used by the `cellsFromCycles` function, that return a list of lists of cycles, each one defining the boundary of a single connected but possibly non path-connected 2-cell in the LAR representation of the fragmented 2-complex.

Classification of non intersecting cycles The function `latticeArray` takes as input the pair `V,EVs`, where V is the list of vertices and `EVs` is the list of cycles of a fragmented 2-cell in 2D. Each `EVs` element is given as a list of edges, given os pairs of integer vertex indices. The returned `testArray` matrix — characterizing the incidences between cycles — has dimension $n \times n$.

⟨ Classification of non intersecting cycles 14 ⟩ ≡

```
""" Classification of non intersecting cycles """
def latticeArray(V,EVs):
    n = len(EVs)
    testArray = []
    for k,ev in enumerate(EVs):
        row = []
        classify = pointInPolygonClassification((V,ev))
        for h in range(0,n):
```

```
                i = EVs[h][0][0]
                point = V[i]
                test = classify(point)
                if test=="p_in": row += [1]
                elif test=="p_out": row += [0]
                elif test=="p_on": row += [-1]
                else: print "error: in cycle classification"
            testArray += [row]
        return testArray
```

◇

Macro referenced in 34.

Extraction of path-connected boundaries The function `cellsFromCycles`, given by the script below, takes as input the cycle-incidence matrix `testArray`, and return the list `out` of lists of cycles, providing the boundaries of a decomposition into connected (but possibly non path-connected) 2-cells of the fragmented 2-face whose non-intersecting cycles were computed as output of the `biconnectedComponent` function.

First the `sons` of each cycle are computed, i.e. the indices of cycles containd in it, as well as the `level` of every cycle, i.e. their position within the lattice of the containment relation (that is a partial order). The level of a cycle is computed as the sum of elements in its matrix column. The roots of the lattice, i.e. the more external cycles have level -1; the following levels have values $0, 1, 2, ...$, respectively.

Then the cycles are ordered in the encreasing value of their level (or *rank*), and finally the significant subsets of disjoint cycles are extracted within the `out` list, starticg from the root cycle(s). The important properties exploited for the extraction are the following: (a) the first element of each sublist is the external boundary cycle, whereas the following cycles, if any, are its internal boundaries; (b) the rank difference between each external and internal boundary must be less or equal to one. It will be 0 only whe the sublist contains only one element (the external boundary) wich is being compared with itself.

Finally the sublists are pruned, by eliminating those whose first element has benn previously used within some of the previous ones (of course: was already used as an internal cycle).

⟨ Extraction of path-connected boundaries 15a ⟩ ≡

```
    """ Extraction of path-connected boundaries """
    def cellsFromCycles (testArray):
        n = len(testArray)
        sons = [[h]+[k for k in range(n) if row[k]==1] for h,row in enumerate(testArray)]
        level = [sum(col) for col in TRANS(testArray)]

        def rank(sons): return [level[x] for x in sons]
        preCells = sorted(sons,key=rank)
```

```
        def levelDifference(son,father): return level[son]-level[father]
        root = preCells[0][0]
        out = [[son for son in preCells[0] if (levelDifference(son,root)<=1) ]]
        for k in range(1,n):
            father = preCells[k][0]
            inout = [son for son in preCells[k] if levelDifference(son,father)<=1 ]
            if not (inout[0] in CAT(out)):
                out += [inout]
        return out
    ◇
```

Macro referenced in 34.

Testing containments between non intersecting cycles The test code for verifying the approch to computation of the containment lattice between non intersecting cycles is given below. Three test files, named respectively `lattice`, `lattice1` and `lattice2`, with the different situations shown in Figure **??**, may be imported from the directory `test/svg/inters/`. Other tests may be easily generated by inserting in this directory some `.svg` files generated with a drawing program.

"test/py/inters/test15.py" 15b ≡

```
    """ Testing containments between non intersecting cycles """
    from larlib import *

    filename = "test/svg/inters/facade.svg"
    lines = svg2lines(filename)
    VIEW(STRUCT(AA(POLYLINE)(lines)))

    V,EV = lines2lar(lines)
    V,EVs = biconnectedComponent((V,EV))
    # candidate face
    FVs = AA(COMP([list,set,CAT]))(EVs)

    testArray = latticeArray(V,EVs)

    for k in range(len(testArray)):
       print k,testArray[k]
    print "\ncells = ", cellsFromCycles(testArray),"\n"

    VV = AA(LIST)(range(len(V)))
    submodel = STRUCT(MKPOLS((V,EV)))
    VIEW(larModelNumbering(1,1,1)(V,[VV,EV,FVs],submodel,0.15))
    ◇
```

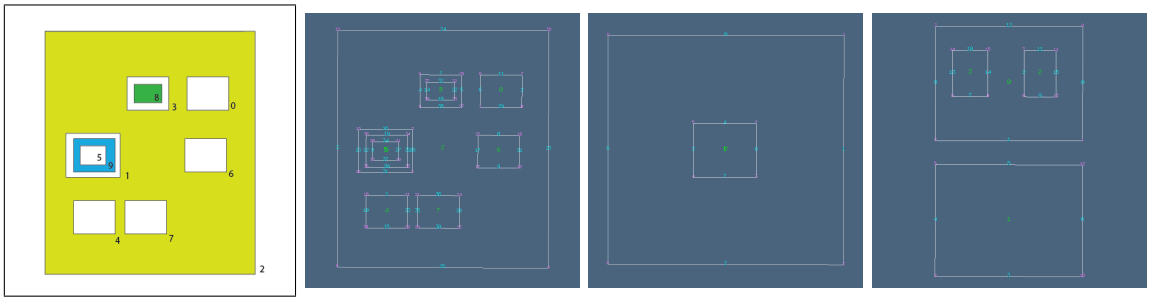

Figure 2: Some examples of nested non intersecting cycles. The corresponding solutions are given in the text.

2.7 Reduction of multiple cycles to a single polyline

The reduction of a lattice of non-intersecting 1-cycles on the boundary of a 2-cell into a single polyline is performed using a scan-line algorithm.

In order to filter the complications induced by edges aligned with the reference axes, first we perform a transformation of vertices from Cartesian to polar coordinates (see Figure 3).

Then a specialized scan-line algorithm is executed, producing a set of *bridge-edges* [YT85] that, added in double instance to the set `EV` of the LAR of the 2-cell, allow for a triangulation of its interior using the algorithm provided by the `poly2tria` module.

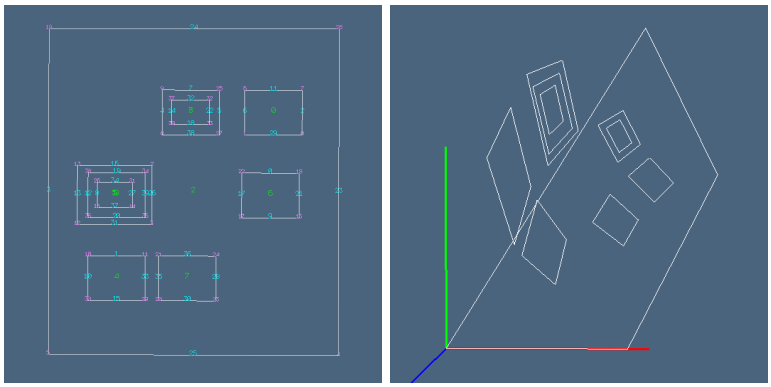

Figure 3: A set of non-intersecting boundary cycles in Cartesian and polar coordinates, using (improperly) an Euclidean metric in the transformed space.

Transforming to polar coordinates The transformation from Cartesian to polar coordinates of the vertices of a two-dimensional LAR model is given in the following script. An image of such transformation is shown in Figure 3. It is only used here to put the vertex coordinates in general position, so simplifying the scan-line algorithm.

⟨ Transforming to polar coordinates 17a ⟩ ≡

```
""" Transforming to polar coordinates """
def cartesian2polar(V):
    Z = [[sqrt(x*x + y*y),math.atan2(y,x)] for x,y in V]
    VIEW(STRUCT(MKPOLS((Z,EV))))
    return Z
```

◇

Macro never referenced.

Scan line algorithm

⟨ Scan line algorithm 17b ⟩ ≡

```
""" Scan line algorithm """
def scan(V,FVs, group,cycleGroup,cycleVerts):
    bridgeEdges = []
    scannedCycles = []
    for k,(point,cycle,v) in enumerate(cycleGroup[:-2]):

        nextCycle = cycleGroup[k+1][1]
        n = len(FVs[group][cycle])
        if nextCycle != cycle:
            if not ((nextCycle in scannedCycles) and (cycle in scannedCycles)):
                print "k =",k
                scannedCycles += [nextCycle]
                m = len(FVs[group][nextCycle])
                v1,v2 = v,cycleGroup[k+1][2]
                minDist = VECTNORM(VECTDIFF([V[v1],V[v2]]))
                for i in FVs[group][cycle]:
                    for j in FVs[group][nextCycle]:
                        dist = VECTNORM(VECTDIFF([V[i],V[j]]))
                        if  dist < minDist:
                            minDist = dist
                            v1,v2 = i,j
                bridgeEdges += [(v1,v2)]
    return bridgeEdges[:-1]
```

◇

Macro referenced in 34.

Scan line algorithm input/output The two-dimensional LAR `model` ≡ `V,EV` was previously created by a `lines2lar(lines)` expression.

⟨ Scan line algorithm input/output 18 ⟩ ≡

```
""" Scan line algorithm input/output """
def connectTheDots(model):
    V,EV = model
```

```
    V,EVs = biconnectedComponent((V,EV))
    FV = AA(COMP([sorted,set,CAT]))(EVs)
    testArray = latticeArray(V,EVs)
    cells = cellsFromCycles(testArray)
    FVs = [[FV[cycle] for cycle in cell] for cell in cells]

    indexedCycles = [zip(FVs[h],range(len(FVs[h])))   for h,cell in enumerate(cells)]
    indexedVerts = [CAT(AA(DISTR)(cell)) for cell in indexedCycles]
    sortedVerts = [sorted([(V[v],c,v) for v,c in cell]) for cell in indexedVerts]

    bridgeEdges = []
    for (group,cycleGroup,cycleVerts) in zip(range(len(cells)),sortedVerts,indexedVerts):
        bridgeEdges += [scan(V,FVs, group,cycleGroup,cycleVerts)]
    return cells,bridgeEdges
```

◇

Macro referenced in 34.

Orientation of component cycles of unconnected boundaries This step of the algorithm needs some preliminary computation, starting from the list `EV` of edges by vertices. Just notice that, at this point of the implementation (as the edges of a set of non intersecting cycles) they all are boundary edges, and hence the `edgeBoundary` variable can be filled with consecutive integers. The `edgeCycles` returned by `boundaryCycles` are very well characterized, according to how discussed in the description of the `Edge cycles associated to a closed chain of edges` paragraph. The corresponding `vertexCycles` are first oriented accordingly to `boundaryCycles`, then rotated (as equivalent permutations) in order to put their element of minimum index in first position (accordingly to the numeration of cycles used in the variable `FVs`). Then, the `CVs` will be set to contain the circularly ordered vertices of the various boundary cycles of each LAR 2-cell, in a manner analogous to FVs, where the vertices are not circularly ordered. Finally, the first (i.e. the *external*) cycle of each cell is counterclockwise oriented, whereas the other cycles (i.e. the *internal* ones), are oriented clockwise.

⟨ Orientation of component cycles of unconnected boundaries 19 ⟩ ≡

```
""" Orientation of component cycles of unconnected boundaries """
def rotatePermutation(inputPermutation,transpositionNumber):
    n = transpositionNumber
    perm = inputPermutation
    permutation = range(n,len(perm))+range(n)
    return [perm[k] for k in permutation]

def canonicalRotation(permutation):
    n = permutation.index(min(permutation))
    return rotatePermutation(permutation,n)
```

```
def setCounterClockwise(h,k,cycle,areas,CVs):
    if areas[cycle] < 0.0:
        chain = copy.copy(CVs[h][k])
        CVs[h][k] = canonicalRotation(REVERSE(chain))

def setClockwise(h,k,cycle,areas,CVs):
    if areas[cycle] > 0.0:
        chain = copy.copy(CVs[h][k])
        CVs[h][k] = canonicalRotation(REVERSE(chain))

def orientBoundaryCycles(model,cells):
    V,EV = model
    edgeBoundary = range(len(EV))
    edgeCycles = boundaryCycles(edgeBoundary,EV)
    vertexCycles = [[ EV[e][1] if e>0 else EV[-e][0] for e in cycle ] for cycle in edgeCycles]
    rotations = [cycle.index(min(cycle)) for cycle in vertexCycles]
    theCycles = sorted([rotatePermutation(perm,n) for perm,n in zip(vertexCycles,rotations)])
    CVs = [[theCycles[cycle] for cycle in cell] for cell in cells]
    areas = surfIntegration((V,theCycles,EV),signed=True)

    for h,cell in enumerate(cells):
        for k,cycle in enumerate(cell):
            if k == 0: setCounterClockwise(h,k,cycle,areas,CVs)
            else: setClockwise(h,k,cycle,areas,CVs)
    return CVs
```
◇

Macro referenced in 34.

From nested boundary cycles to triangulation

⟨ From nested boundary cycles to triangulation 20 ⟩ ≡

```
""" From nested boundary cycles to triangulation """
def boundaryCycles2triangulation( (V,EV) ):
    model = V,EV
    cells,bridgeEdges = connectTheDots(model)
    CVs = orientBoundaryCycles(model,cells)

    polygons = [[[V[u] for u in cycle] for cycle in cell] for cell in CVs]
    triangleSet = []

    for polygon in polygons:
        triangledPolygon = []
        externalCycle = polygon[0]
        polyline = []
        for p in externalCycle:
            polyline.append(Point(p[0],p[1]))
```

```
            cdt = CDT(polyline)

            internalCycles = polygon[1:]
            for cycle in internalCycles:
                hole = []
                for p in cycle:
                    hole.append(Point(p[0],p[1]))
                cdt.add_hole(hole)

            triangles = cdt.triangulate()
            trias = [ [[t.a.x,t.a.y,0],[t.c.x,t.c.y,0],[t.b.x,t.b.y,0]] for t in triangles ]
            triangleSet += [AA(REVERSE)(trias)]
        return triangleSet
    ◇
```

Macro referenced in 34.

2.8 Monocyclic polygons using bridge-edges

This subsection, even if correct, is not currently used, since the used triangulation algorithm, from the `poly2tri` package, cannot handle repeated vertices.

Generation of 1-boundaries as vertex permutation In algebraic topology a k-cycle is a k-chain whose boundary is empty. Also, an unconnected k-cycle is the direct sum of two or more k-cycles. A good formal representation of every simplicial k-cycle, where *each* component k-simplex has $k+1$ $(k-1)$-adjacent k-simplices is a $(k+1)$-array, indexed by k-simplices, i.e. the *Winged Representation* [PBCF93].

In the case of oriented 1-cycles, a good representation is given by considering the (ordering of) 0-faces (vertices) as a permutation of n integers, i.e. as a bijective function $\pi : [0, n] \to [0, n]$ that can be represented as an array `verts` of integers indexed on integers, and the 1-faces (edges) as a dictionary (mapping) `nextVert` $verts \to verts$.

In order to join two component cycles using one of `bridgeEdges`, say (u, v), computed by the function `connectTheDots` previously given, we must save $\pi(u)$ and $\pi(v)$, say, within x and y, respectively

⟨ Generation of 1-boundaries as vertex permutation 21a ⟩ ≡

```
    """ Generation of 1-boundaries as vertex permutation """
    def boundaryCycles2vertexPermutation( model ):
        V,EV = model
        cells,bridgeEdges = connectTheDots(model)
        CVs = orientBoundaryCycles(model,cells)

        verts = CAT(CAT( CVs ))
        n = len(verts)
        W = copy.copy(V)
```

```
    assert len(verts) == sorted(verts)[n-1]-sorted(verts)[0]+1
    nextVert = dict([(v,cycle[(k+1)%(len(cycle))]) for cell in CVs for cycle in cell
                  for k,v in enumerate(cycle)])
    for k,(u,v) in enumerate(CAT(bridgeEdges)):
        x,y = nextVert[u],nextVert[v]
        nextVert[u] = n+2*k+1
        nextVert[v] = n+2*k
        nextVert[n+2*k] = x
        nextVert[n+2*k+1] = y
        W += [W[u]]
        W += [W[v]]
        EW = nextVert.items()
    return W,EW
```

◇

Macro referenced in 34.

Wire-frame LAR to boundary polygons The 2-dimensional LAR model `(W,EW)` returned by the function `boundaryCycles2vertexPermutation` is pretty special, since it is at the same time both a standard LAR model, i.e. a pair (vertices, edges_by_vertices), and a permutation of vertex indices providing implicitly the ordered cycles on the boundary of a 2-complex. In other words, `EW` is both a (possibly unconnected) 1-cycle and an 1-boundary.

In the following script we extract the list of connected boundaries (including bridge-edges) from the `EW` permutation of the first m integers.

⟨ Wire-frame LAR to boundary polygons 21b ⟩ ≡

```
""" lar2boundaryPolygons """
def lar2boundaryPolygons(model):
    W,EW = boundaryCycles2vertexPermutation( model )
    EW = AA(list)(EW)
    polygons = []
    for k,edge in enumerate(EW):
        polygon = []
        if edge[0]>=0:
            first = edge[0]
            done = False
        while (not done) and edge[0] >= 0:
            polygon += [edge[0]]
            edge[0] = -edge[0]
            edge = EW[edge[1]]
            if len(polygon)>1 and polygon[-1] == first:
                EW[first][0] = -float(first)
                break
        if polygon != []:
            if polygon[0]==polygon[-1]: polygon=polygon[:-1]
            polygons += [polygon]
```

```
        return W,polygons
```
◇

Macro referenced in 34.

2.9 Transformation of an array of lines in a 2D LAR complex

Transformation of an array of lines in a 2D LAR complex The whole transformation of an array of lines into a two-dimensional `LAR` complex is executed by the function `larFromLines`. The function returns the model triple `V,FV,EV`. The last element in `FV` is the *ordered* boundary chain. just notice that

⟨ Transformation of an array of lines in a 2D LAR complex 22 ⟩ ≡

```
""" Transformation of an array of lines in a 2D LAR complex """
def larFromLines(lines):
    V,EV = lines2lar(lines)
    #VIEW(EXPLODE(1.2,1.2,1.2)(MKPOLS((V,EV))))
    V,EVs = biconnectedComponent((V,EV))
    if EVs != []:
        EV = list(set(AA(tuple)(AA(sorted)(max(EVs, key=len)))))  ## NB
        V,EV = larRemoveVertices(V,EV)
        V,FV,EV = facesFromComps((V,EV))
        areas = integr.surfIntegration((V,FV,EV))
        orderedFaces = sorted([[area,FV[f]] for f,area in enumerate(areas)])
        interiorFaces = [face for area,face in orderedFaces[:-1]]
        return V,interiorFaces,EV
    else: return None
```
◇

Macro referenced in 34.

2.10 Pruning LAR models from parts out of proper resolution

Pruning of clusters of too close vertices is executed by taking a LAR model as input, executing the following computations, and producing a new simplified LAR model.

Pruning away clusters of close vertices First, reduce the array of vertices `pts` to its *quotient set* with respect to the transitive closure of the relation of "nearness". Two vertices are "near" when their (Euclidean) distance is less than a given `RADIUS`. The subgraphs of the graph of this relation are *contracted* in a single point, set to the centroid of the vertices of the subgraph. The function `W` takes as input the array `pts` of vertex points, and returns: (a) the array `newV` of new vertices; (b) the list of lists `close` of sorted indices of pairs of close vertices, removed from duplicates; (c) the list of `clusters` of `pts` indices; (d) the integer `vmap` array, mapping old vertex indices to new vertex indices.

⟨ Pruning away clusters of close vertices 23 ⟩ ≡

```
""" Pruning away clusters of close vertices """
from scipy.spatial import cKDTree

def pruneVertices(pts,radius=0.001):
    tree = cKDTree(pts)
    a = cKDTree.sparse_distance_matrix(tree,tree,radius)
    print a.keys()
    close = list(set(AA(tuple)(AA(sorted)(a.keys()))))
    import networkx as nx
    G=nx.Graph()
    G.add_nodes_from(range(len(pts)))
    G.add_edges_from(close)
    clusters, k, h = [], 0, 0

    subgraphs = list(nx.connected_component_subgraphs(G))
    V = [None for subgraph in subgraphs]
    vmap = [None for k in xrange(len(pts))]
    for k,subgraph in enumerate(subgraphs):
        group = subgraph.nodes()
        if len(group)>1:
            V[k] = CCOMB([pts[v] for v in group])
            for v in group: vmap[v] = k
            clusters += [group]
        else:
            oldNode = group[0]
            V[k] = pts[oldNode]
            vmap[oldNode] = k
    return V,close,clusters,vmap
```

◇

Macro referenced in 34.

Export a simplified LAR model Next, update the arrays of compressed characteristic matrices of a Linear Algebraic Representation. The standard approach is to read row-wise the arrays of matrices of incidence of cells on vertices; translate every index using the `vmap` array, mapping old vertex indices to new ones; remove repeated indices and substitute them with a single instance; check if the new index list has lenght greater or equal to the number of vertices of the simplex of the proper dimension. Finally, write an output cell if and only if the previous test is true.

⟨ Return a simplified LAR model 24 ⟩ ≡

```
""" Return a simplified LAR model """
def larSimplify(model,radius=0.001):
    if len(model)==2: V,CV = model
    elif len(model)==3: V,CV,FV = model
```

```
        else: print "ERROR: model input"

        W,close,clusters,vmap = pruneVertices(V,radius)
        celldim = DIM(MKPOL([V,[[v+1 for v in CV[0]]],None]))
        newCV = [list(set([vmap[v] for v in cell])) for cell in CV]
        CV = list(set([tuple(sorted(cell)) for cell in newCV if len(cell) >= celldim+1]))
        CV = sorted(CV,key=len) # to get the boundary cell as last one (in most cases)

        if len(model)==3:
            celldim = DIM(MKPOL([V,[[v+1 for v in FV[0]]],None]))
            newFV = [list(set([vmap[v] for v in facet])) for facet in FV]
            FV = [facet for facet in newFV if len(facet) >= celldim]
            FV = list(set(AA(tuple)(AA(sorted)(FV))))
            return W,CV,FV
        else: return W,CV
    ◇
```

Macro referenced in 34.

Test of pruning clusters of close vertices Here a list of random 2D points is generated. Then the set of vertices is pruned by updating it to its quotient set with respect to the transitive closure of a relation of "nearness" within an Euclidean distance of given `RADIUS`. The pruning of vertices is performed by the `pruneVertices` function, with input the array `pts` of points. The dictionary `vmap`

"test/py/inters/test13.py" 25a ≡

```
    """ Test of pruning clusters of close vertices """
    from larlib import *
    from scipy import rand
    from scipy.spatial import cKDTree
    POINTS = 1000
    RADIUS = 0.01

    pts = [rand(2).tolist() for k in range(POINTS)]
    VIEW(STRUCT(AA(MK)(pts)))
    V,close,clusters,vmap = pruneVertices(pts,RADIUS)
    circles = [T([1,2])(pts[h])(CIRCUMFERENCE(RADIUS)(18)) for h,k in close]
    convexes = [JOIN(AA(MK)([pts[v] for v in cluster])) for cluster in clusters]
    W = COLOR(CYAN)(STRUCT(AA(MK)(V)))
    VIEW(STRUCT(AA(MK)(pts)+AA(COLOR(YELLOW))(circles)))
    VIEW(STRUCT(AA(COLOR(RED))(convexes)+AA(MK)(pts)+AA(COLOR(YELLOW))(circles)+[W]))
    ◇
```

Test for exporting a simplified LAR model

"test/py/inters/test14.py" 25b ≡

```
""" Test for exporting a simplified LAR model """
from larlib import *
filename = "test/svg/inters/closepoints.svg"
lines = svg2lines(filename)
VIEW(STRUCT(AA(POLYLINE)(lines)))

V,FV,EV = larFromLines(lines)
VIEW(EXPLODE(1.2,1.2,1)(MKPOLS((V,FV[:-1]+EV)) + AA(MK)(V)))
VV = AA(LIST)(range(len(V)))
submodel = STRUCT(MKPOLS((V,EV)))
VIEW(larModelNumbering(1,1,1)(V,[VV,EV,FV[:-1]],submodel,0.5))

V,EV = lines2lar(lines)
VIEW(EXPLODE(1.2,1.2,1)(MKPOLS((V,EV))))
pts = V
RADIUS = 0.05
V,close,clusters,vmap = pruneVertices(pts,RADIUS)
circles = [T([1,2])(pts[h])(CIRCUMFERENCE(RADIUS)(18)) for h,k in close]
convexes = [JOIN(AA(MK)([pts[v] for v in cluster])) for cluster in clusters]
W = COLOR(CYAN)(STRUCT(AA(MK)(V)))
VIEW(STRUCT(AA(COLOR(RED))(convexes)+AA(MK)(pts)+AA(COLOR(YELLOW))(circles)+[W]))
◇
```

2.11 Point in polygon classification

⟨ Point in polygon testing 25c ⟩ ≡

⟨ Half-line crossing test 28a ⟩
⟨ Tile codes computation 26a ⟩
⟨ Point-in-polygon classification algorithm 26b ⟩
◇

Macro referenced in 34.

Tile codes computation

⟨ Tile codes computation 26a ⟩ ≡

```
""" Tile codes computation """
def setTile(box):
    tiles = [[9,1,5],[8,0,4],[10,2,6]]
    b1,b2,b3,b4 = box
    def tileCode(point):
        x,y = point
        code = 0
        if y>b1: code=code|1
        if y<b2: code=code|2
        if x>b3: code=code|4
```

```
            if x<b4: code=code|8
            return code
        return tileCode
```
◇

Macro referenced in 25c.

Point in polygon testing

⟨ Point-in-polygon classification algorithm 26b ⟩ ≡

```
    """ Point in polygon classification """
    def pointInPolygonClassification(pol):

        V,EV = pol
        # edge orientation
        FV = [sorted(set(CAT(EV)))]
        orientedCycles = boundaryPolylines(Struct([(V,FV,EV)]))
        EV = []
        for cycle in orientedCycles:
            EV += zip(cycle[:-1],cycle[1:])

        def pointInPolygonClassification0(p):
            x,y = p
            xmin,xmax,ymin,ymax = x,x,y,y
            tilecode = setTile([ymax,ymin,xmax,xmin])
            count,status = 0,0

            for k,edge in enumerate(EV):
                p1,p2 = edge[0],edge[1]
                (x1,y1),(x2,y2) = p1,p2
                c1,c2 = tilecode(p1),tilecode(p2)
                c_edge, c_un, c_int = c1^c2, c1|c2, c1&c2

                if c_edge == 0 and c_un == 0: return "p_on"
                elif c_edge == 12 and c_un == c_edge: return "p_on"
                elif c_edge == 3:
                    if c_int == 0: return "p_on"
                    elif c_int == 4: count += 1
                elif c_edge == 15:
                    x_int = ((y-y2)*(x1-x2)/(y1-y2))+x2
                    if x_int > x: count += 1
                    elif x_int == x: return "p_on"
                elif c_edge == 13 and ((c1==4) or (c2==4)):
                        crossingTest(1,2,status,count)
                elif c_edge == 14 and (c1==4) or (c2==4):
                        crossingTest(2,1,status,count)
                elif c_edge == 7: count += 1
```

```
                elif c_edge == 11: count = count
                elif c_edge == 1:
                    if c_int == 0: return "p_on"
                    elif c_int == 4: crossingTest(1,2,status,count)
                elif c_edge == 2:
                    if c_int == 0: return "p_on"
                    elif c_int == 4: crossingTest(2,1,status,count)
                elif c_edge == 4 and c_un == c_edge: return "p_on"
                elif c_edge == 8 and c_un == c_edge: return "p_on"
                elif c_edge == 5:
                    if (c1==0) or (c2==0): return "p_on"
                    else: crossingTest(1,2,status,count)
                elif c_edge == 6:
                    if (c1==0) or (c2==0): return "p_on"
                    else: crossingTest(2,1,status,count)
                elif c_edge == 9 and ((c1==0) or (c2==0)): return "p_on"
                elif c_edge == 10 and ((c1==0) or (c2==0)): return "p_on"
            if ((round(count)%2)==1): return "p_in"
            else: return "p_out"
        return pointInPolygonClassification0
```

◇

Macro referenced in 25c.

Half-line crossing test

⟨ Half-line crossing test 28a ⟩ ≡

```
    """ Half-line crossing test """
    def crossingTest(new,old,count,status):
        if status == 0:
            status = new
            count += 0.5
        else:
            if status == old: count += 0.5
            else: count -= 0.5
            status = 0
```

◇

Macro referenced in 25c.

2.12 Drawing multiply-connected boundary polylines

A multiply-connected boundary polyline gives the boundary edges of a 2-cell. In LAR, 2-cells must be connected, but not necessarily simply-connected (or path-connected, or homotopic to a point). The "solid" drawing of such a generally non-convex 2-cells is not easy. In particular, they must be decomposed into a coherently-oriented simplicial 2-complex, i.e. into a set of equioriented triangles. The library function used at this purpose

(), contained in the `poly2tria` module, only accepts a single boundary polyline. Hence the purpose of this section is to transform a list of cycles, corresponding orderly to the exterior and the interior boundaries of a 2-cell, into a single polyline, using the so-called *bridge-edges* [YT85] mechanism. The transformation from a set of non-intersecting boundary cycles to a single connected polyline is given in Section 2.7.

Generating the LAR of a set of non-intersecting cycle We use the `test/svg/lattice.svg` file to test the exporting of different boundary chains. The example `test15.py` aims to prepare the computational environment for writing down the LAR of the 2-complex generated by any set of non-intersecting 1-cles in 2D.

Let us remenber that any d-cell in the domain of the LAR scheme must be connected, but not necessarily contractible to a point, i.e. may contain internal holes. The goal af this test and of the previous one, is hence to generalize the creation of 2-complexes (sets of polygons) starting from several non-intersecting boundary cycles, instead than starting from just one closed polyline.

The motivation arises from situations created by the Boolean algorithm, as well as from the imput of a 2-complex from general wire-frame drawings.

"test/py/inters/test16.py" 28b ≡

```
""" Generating the LAR of a set of non-intersecting cycles """
from larlib import *

sys.path.insert(0, 'test/py/inters/')
from test15 import *

cells = cellsFromCycles(testArray)
CV = AA(COMP([list,set,CAT]))(EVs)
EVdict = dict(zip(EV,range(len(EV))))
FE = [[EVdict[edge] for edge in cycle] for cycle in EVs]
edges = [CAT([FE[cycle] for cycle in cell]) for cell in cells]
FVs = [[CV[cycle] for cycle in cell] for cell in cells]
FV = AA(CAT)(FVs)

def allBinarySubsetsOfLenght(n):
   out = [list(('{0:0'+str(n)+'b}').format(k)) for k in range(1,2**n)]
   return AA(AA(int))(out)

n = len(cells)
chains = allBinarySubsetsOfLenght(n)

cycles = STRUCT(MKPOLS((V,EV)))
for chain in chains:
    chainBoundary = COLOR(RED)(STRUCT(MKPOLS((V,[EV[e]
                        for e in chain2BoundaryChain(FV,EV)(chain)]))))
```

```
    VIEW(STRUCT([cycles, chainBoundary]))
```
◇

LAR of the 2-complex generated by non-intersecting cycles Therefore, the LAR of the 2-complex generated from the input get from `test15.py` is `(V,FV,EV)` with the components given by the the following:

```
V = [[0.7808,0.6751],[0.6044,0.6751],[0.319,0.5804],[0.319,0.3994],[0.8936,0.0],
[0.0,0.0],[0.6044,0.8123],[0.7808,0.8123],[0.3495,0.6751],[0.3495,0.8123],
[0.1218,0.3036],[0.2983,0.3036],[0.0886,0.3994],[0.0886,0.5804],[0.2581,0.4505],
[0.1495,0.4505],[0.7717,0.4213],[0.5952,0.4213],[0.7717,0.5585],[0.0,1.0],
[0.3403,0.1664],[0.3403,0.3036],[0.5952,0.5585],[0.5168,0.1664],[0.5168,0.3036],
[0.5259,0.8123],[0.1495,0.5293],[0.5259,0.6751],[0.8936,1.0],[0.2983,0.1664],
[0.1218,0.1664],[0.2581,0.5293],[0.4965,0.7815],[0.4965,0.7059],[0.2983,
0.5585],[0.2983,0.4213],[0.1218,0.4213],[0.3789,0.7815],[0.1218,0.5585],
[0.3789,0.7059]]

FV = [[0,1,2,3,4,5,6,7,8,9,10,11,12,13,16,17,18,19,20,21,22,23,24,25,27,
28,29,30],[32,33,37,39],[34,35,36,38,26,15,14,31]]

EV = [(0,1),(0,7),(1,6),(2,3),(2,13),(3,12),(4,5),(4,28),(5,19),(6,7),(8,9),
(8,27),(9,25),(10,11),(10,30),(11,29),(12,13),(14,15),(14,31),(15,26),
(16,17),(16,18),(17,22),(18,22),(19,28),(20,21),(20,23),(21,24),(23,24),
(25,27),(26,31),(29,30),(32,33),(32,37),(33,39),(34,35),(34,38),(35,36),
(36,38),(37,39)]
```

Of course, this LAR representation gives full control of the complex topology, including k-(co)chains and (co)boundary operators. For example, in Figure 4 we show a solid image of the three 2-cells in `FV`, and, drawn in red the boundary 1-cells in the complex, corresponding to the 2-chains of coordinates $[1,0,0]$, $[0,1,0]$, $[0,0,1]$, and $[1,1,0]$, respectively.

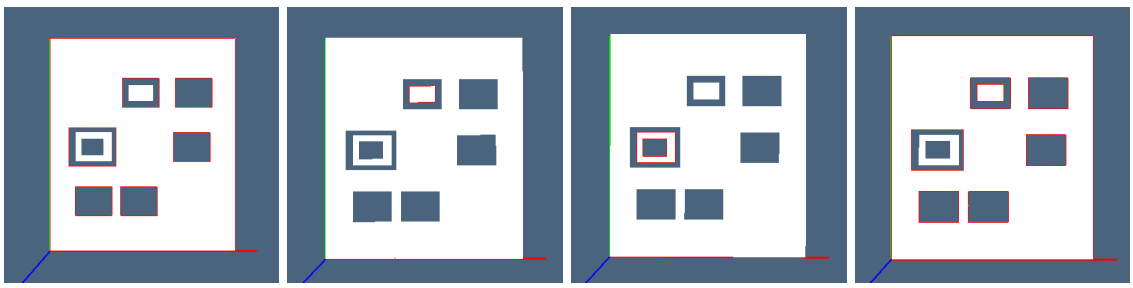

Figure 4: Some examples of boundaries (in red) of 2-chains from nested non intersecting cycles.

2.13 Boundary polylines

Generating the LAR of a set of non-intersecting cycles The example provided by `test/py/inters/test17.py` is completed here by showing the solid drawing of the generated LAR data structure, and superimposing to it the boundary 1-chains generated by several 2-chains.

"test/py/inters/test17.py" 30 ≡

```
""" Generating the LAR of a set of non-intersecting cycles """
from larlib import *

sys.path.insert(0, 'test/py/inters/')
from test16 import *

lar = (V,FV,EV)

bcycles = boundaryCycles(range(len(EV)),EV)
polylines = [[V[EV[e][1]] if e>0 else V[EV[-e][0]] for e in cycle ] for cycle in bcycles]
polygons = [polyline + [polyline[0]] for polyline in polylines]

complex = SOLIDIFY(STRUCT(AA(POLYLINE)(polygons)))
for chain in chains:
    chainBoundary = COLOR(RED)(STRUCT(MKPOLS((V,[EV[e]
                        for e in chain2BoundaryChain(FV,EV)(chain)]))))
    VIEW(STRUCT([complex, chainBoundary]))
◇
```

"test/py/inters/test18.py" 31a ≡

```
""" Orienting a set of non-intersecting cycles """
from larlib import *

sys.path.insert(0, 'test/py/inters/')
from test17 import *

cells,bridgeEdges = connectTheDots((V,EV))
CVs = orientBoundaryCycles((V,EV),cells)

print "\nCVs =",CVs
◇
```

"test/py/inters/test19.py" 31b ≡

```
""" Generating the LAR of a set of non-intersecting cycles """
from larlib import *

sys.path.insert(0, 'test/py/inters/')
from test17 import *
```

```
    W,EW = boundaryCycles2vertexPermutation( (V,EV) )
    VIEW(EXPLODE(1.2,1.2,1.2)(MKPOLS((W,EW))))
    ◇
```

"test/py/inters/test20.py" 31c ≡

```
    """ Generating the Triangulation of a set of non-intersecting cycles """
    from larlib import *

    sys.path.insert(0, 'test/py/inters/')
    from test17 import *

    triangleSet = boundaryCycles2triangulation( (V,EV) )

    VIEW(STRUCT(AA(JOIN)(AA(AA(MK))(CAT(triangleSet)))))
    VIEW(SKEL_1(STRUCT(AA(JOIN)(AA(AA(MK))(CAT(triangleSet))))))

    """
    model = V,EV
    W,FW = lar2boundaryPolygons(model)
    polygons = [[W[u] for u in poly] for poly in FW]
    VIEW(STRUCT(AA(POLYLINE)(polygons)))

    triangleSet,triangledFace = [],[]
    for polygon in polygons:
        triangledPolygon = []
        polyline = []
        for p in polygon:
            polyline.append(Point(p[0],p[1]))
        cdt = CDT(polyline)

        triangles = cdt.triangulate()
        trias = [ [[t.a.x,t.a.y,0],[t.c.x,t.c.y,0],[t.b.x,t.b.y,0]] for t in triangles ]
        triangleSet += [AA(REVERSE)(trias)]
    """
    ◇
```

From structures to boundary polylines

⟨ From structures to boundary polylines 32a ⟩ ≡

```
    """ From structures to boundary polylines """
    def boundaryPolylines(struct):
        V,boundaryEdges = structBoundaryModel(struct)
        polylines = boundaryModel2polylines((V,boundaryEdges))
        return polylines
    ◇
```

Macro referenced in 34.

From LAR oriented boundary model to polylines

⟨ From LAR boundary model to polylines 32b ⟩ ≡

```
""" From LAR oriented boundary model to polylines """
def boundaryModel2polylines(model):
    V,EV = model
    polylines = []
    succDict = dict(EV)
    visited = [False for k in range(len(V))]
    nonVisited = [k for k in succDict.keys() if not visited[k]]
    while nonVisited != []:
        first = nonVisited[0]; v = first; polyline = []
        while visited[v] == False:
            visited[v] = True;
            polyline += V[v],
            v = succDict[v]
        polyline += [V[first]]
        polylines += [polyline]
        nonVisited = [k for k in succDict.keys() if not visited[k]]
    return polylines
```

◇

Macro referenced in 34.

From Struct object to LAR boundary model

⟨ From Struct object to LAR boundary model 33a ⟩ ≡

```
""" From Struct object to LAR boundary model """
def structFilter(obj):
    if isinstance(obj,list):
        if (len(obj) > 1):
            return [structFilter(obj[0])] + structFilter(obj[1:])
        return [structFilter(obj[0])]
    if isinstance(obj,Struct):
        if obj.category in ["external_wall", "internal_wall", "corridor_wall"]:
            return
        return Struct(structFilter(obj.body),obj.name,obj.category)
    return obj

def structBoundaryModel(struct):
    filteredStruct = structFilter(struct)
    #import pdb; pdb.set_trace()
    V,FV,EV = struct2lar(filteredStruct)
    edgeBoundary = boundaryCells(FV,EV)
    cycles = boundaryCycles(edgeBoundary,EV)
    edges = [signedEdge for cycle in cycles for signedEdge in cycle]
    orientedBoundary = [ AA(SIGN)(edges), AA(ABS)(edges)]
```

```
        cells = [EV[e] if sign==1 else REVERSE(EV[e]) for (sign,e) in zip(*orientedBoundary)]
        if cells[0][0]==cells[1][0]: # bug badly patched! ... TODO better
            temp0 = cells[0][0]
            temp1 = cells[0][1]
            cells[0] = [temp1, temp0]
        return V,cells
```
◇

Macro referenced in 34.

Edge cycles associated to a closed chain of edges The output `cycles` are returned as lists of oriented edges, given in consecutive sequence in each list. Each cycle is given in the opposite ordering of its first edge. The first edge of each cycle is the one of minimum index in the cycle. The list of output cycles is returned ordered for increasing (or better, in decreasing order, since negative) order of its first element. The first output cycle starts from index 0, that cannot oriented directly, since -0 is not allowed for integer indices. It must be considered as negative, i.e. as oriented as the opposite of its canonical orientation.

⟨ Edge cycles associated to a closed chain of edges 33b ⟩ ≡

```
    """ Edge cycles associated to a closed chain of edges """
    def boundaryCycles(edgeBoundary,EV):
        verts2edges = defaultdict(list)
        for e in edgeBoundary:
            verts2edges[EV[e][0]] += [e]
            verts2edges[EV[e][1]] += [e]
        cycles = []

        cbe = copy.copy(edgeBoundary)
        while cbe != []:
            e = cbe[0]
            v = EV[e][0]
            cycle = []
            while True:
                cycle += [(e,v)]
                e = list(set(verts2edges[v]).difference([e]))[0]
                cbe.remove(e)
                v = list(set(EV[e]).difference([v]))[0]
                if (e,v)==cycle[0]:
                    break
            n = len(cycle)
            cycles += [[e if EV[e]==(cycle[(k-1)%n][1],cycle[k%n][1]) else -e
                for k,(e,v) in enumerate(cycle)]]
        return cycles
```
◇

Macro referenced in 34.

3 Exporting the module

"larlib/larlib/inters.py" 34 ≡

```
""" Module for pipelined intersection of geometric objects """
from larlib import *
from scipy import mat
DEBUG = True
```

⟨ Coding utilities 46a ⟩
⟨ Generation of random lines 46b ⟩
⟨ Containment boxes 2a ⟩
⟨ Splitting the input above and below a threshold 2b ⟩
⟨ Box metadata computation ? ⟩
⟨ Iterative splitting of box buckets 3a ⟩
⟨ Intersection of two line segments 3b ⟩
⟨ Brute force bucket intersection 4 ⟩
⟨ Accelerate intersection of lines 5 ⟩
⟨ Edge cycles associated to a closed chain of edges 33b ⟩
⟨ From Struct object to LAR boundary model 33a ⟩
⟨ From structures to boundary polylines 32a ⟩
⟨ From LAR boundary model to polylines 32b ⟩
⟨ Point in polygon testing 25c ⟩
⟨ Classification of non intersecting cycles 14 ⟩
⟨ Extraction of path-connected boundaries 15a ⟩
⟨ Scan line algorithm 17b ⟩
⟨ Scan line algorithm input/output 18 ⟩
⟨ Orientation of component cycles of unconnected boundaries 19 ⟩
⟨ From nested boundary cycles to triangulation 20 ⟩
⟨ Generation of 1-boundaries as vertex permutation 21a ⟩
⟨ Wire-frame LAR to boundary polygons 21b ⟩
⟨ Create the LAR of fragmented lines 6 ⟩
⟨ Biconnected components 7a ⟩
⟨ Slope of edges 9 ⟩
⟨ Ordered incidence relationship of vertices and edges 11a ⟩
⟨ Faces from biconnected components 11b ⟩
⟨ SVG input parsing and transformation 42 ⟩
⟨ Transformation of an array of lines in a 2D LAR complex 22 ⟩
⟨ Pruning away clusters of close vertices 23 ⟩
⟨ Return a simplified LAR model 24 ⟩
◇

4 Examples

Generation of random line segments and their boxes

"test/py/inters/test01.py" 35a ≡

```
""" Generation of random line segments and their boxes """
from larlib import *

randomLineArray = randomLines(200,0.3)
VIEW(STRUCT(AA(POLYLINE)(randomLineArray)))

boxes = containment2DBoxes(randomLineArray)
rects= AA(box2rect)(boxes)
cyan = COLOR(CYAN)(STRUCT(AA(POLYLINE)(randomLineArray)))
yellow = COLOR(YELLOW)(STRUCT(AA(POLYLINE)(rects)))
VIEW(STRUCT([cyan,yellow]))
◇
```

Split segment array in four independent buckets

"test/py/inters/test02.py" 35b ≡

```
""" Split segment array in four independent buckets """
from larlib import *

randomLineArray = randomLines(200,0.3)
VIEW(STRUCT(AA(POLYLINE)(randomLineArray)))
boxes = containment2DBoxes(randomLineArray)
bucket = range(len(boxes))
below,above = splitOnThreshold(boxes,bucket,1)
below1,above1 = splitOnThreshold(boxes,above,2)
below2,above2 = splitOnThreshold(boxes,below,2)

cyan = COLOR(CYAN)(STRUCT(AA(POLYLINE)(randomLineArray[k] for k in below1)))
yellow = COLOR(YELLOW)(STRUCT(AA(POLYLINE)(randomLineArray[k] for k in above1)))
red = COLOR(RED)(STRUCT(AA(POLYLINE)(randomLineArray[k] for k in below2)))
green = COLOR(GREEN)(STRUCT(AA(POLYLINE)(randomLineArray[k] for k in above2)))

VIEW(STRUCT([cyan,yellow,red,green]))
◇
```

Generation and random coloring of independent line buckets

"test/py/inters/test03.py" 36a ≡

```
""" Generation and random coloring of independent line buckets """
from larlib import *

lines = randomLines(200,0.3)
VIEW(STRUCT(AA(POLYLINE)(lines)))

boxes = containment2DBoxes(lines)
buckets = boxBuckets(boxes)
```

```
colors = [CYAN, MAGENTA, WHITE, RED, YELLOW, GRAY, GREEN, ORANGE, BLACK, BLUE, PURPLE, BROWN]
sets = [COLOR(colors[k%12])(STRUCT(AA(POLYLINE)([lines[h]
            for h in bucket]))) for k,bucket in enumerate(buckets) if bucket!=[]]

VIEW(STRUCT(sets))
◇
```

Construction of LAR = (V,EV) of random line arrangement

"test/py/inters/test04.py" 36b ≡

```
""" LAR of random line arrangement """
from larlib import *

lines = randomLines(300,0.2)
VIEW(STRUCT(AA(POLYLINE)(lines)))

intersectionPoints,params,frags = lineIntersection(lines)

marker = CIRCLE(.005)([4,1])
markers = STRUCT(CONS(AA(T([1,2]))(intersectionPoints))(marker))
VIEW(STRUCT(AA(POLYLINE)(lines)+[COLOR(RED)(markers)]))

V,EV = lines2lar(lines)
marker = CIRCLE(.01)([4,1])
markers = STRUCT(CONS(AA(T([1,2]))(V))(marker))
#markers = STRUCT(CONS(AA(T([1,2]))(intersectionPoints))(marker))
polylines = STRUCT(MKPOLS((V,EV)))
VIEW(STRUCT([polylines]+[COLOR(MAGENTA)(markers)]))
◇
```

Splitting of othogonal lines

"test/py/inters/test05.py" 37a ≡

```
""" LAR from splitting of othogonal lines """
from larlib import *
⟨ Orthogonal example 37b ⟩
◇
```

⟨ Orthogonal example 37b ⟩ ≡

```
lines = [[[0,0],[6,0]], [[0,4],[10,4]], [[0,0],[0,4]], [[3,0],[3,4]],
[[6,0],[6, 8]], [[3,2],[6,2]], [[10,0],[10,8]], [[0,8],[10,8]]]

VIEW(EXPLODE(1.2,1.2,1)(AA(POLYLINE)(lines)))
```

```
V,EV = lines2lar(lines)
VIEW(EXPLODE(1.2,1.2,1)(MKPOLS((V,EV))))
◇
```

Macro referenced in 37a, 38b, 39.

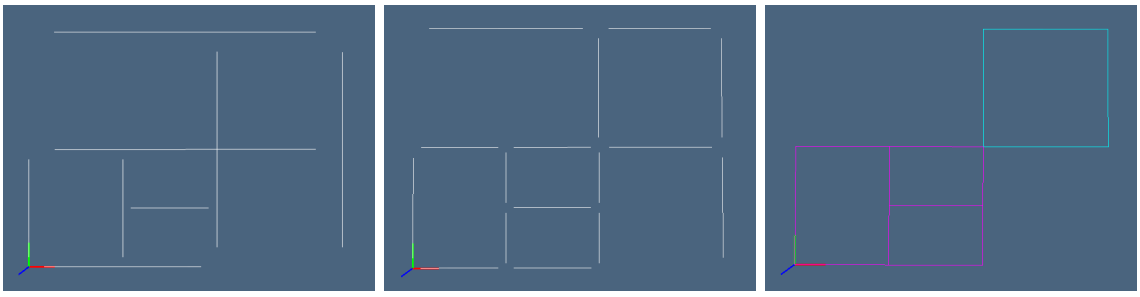

Figure 5: Splitting of orthogonal lines: (a) exploded input; (a) exploded output; (c) biconnected components.

Random coloring of the generated 1-complex LAR

"test/py/inters/test06.py" 38a ≡

```
""" Random coloring of the generated 1-complex """
from larlib import *

lines = randomLines(800,0.2)
VIEW(STRUCT(AA(POLYLINE)(lines)))

V,EV = lines2lar(lines)
colors = [CYAN, MAGENTA, WHITE, RED, YELLOW, GRAY, GREEN, ORANGE, BLACK, BLUE, PURPLE, BROWN]
sets = [COLOR(colors[k%12])(POLYLINE([V[e[0]],V[e[1]]])) for k,e in enumerate(EV)]

VIEW(STRUCT(sets))
◇
```

Biconnected components from orthogonal LAR model

"test/py/inters/test07.py" 38b ≡

```
""" Biconnected components from orthogonal LAR model """
from larlib import *
colors = [CYAN, MAGENTA, WHITE, RED, YELLOW, GREEN, ORANGE, BLACK, BLUE, PURPLE]

⟨ Orthogonal example 37b ⟩
model = V,EV
```

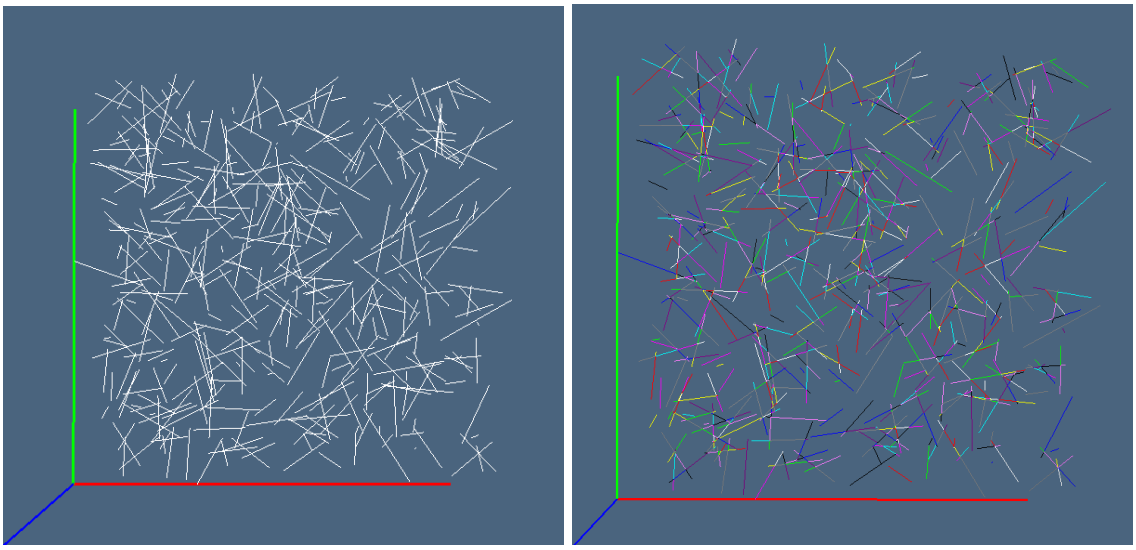

Figure 6: Splitting of intersecting lines: (a) random input; (a) splitted and colored LAR output.

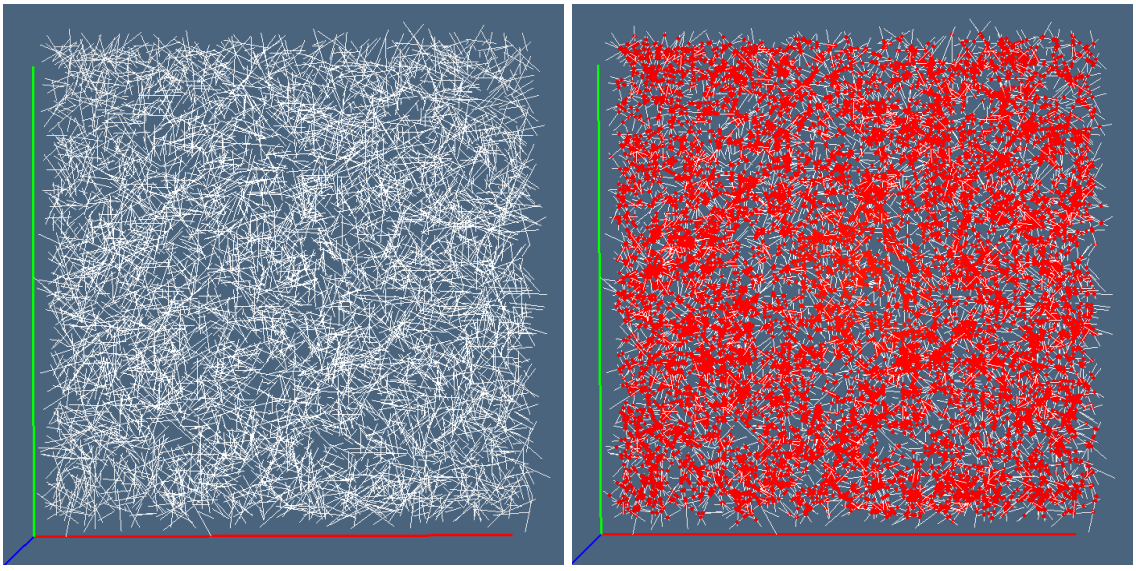

Figure 7: The intersection of 5000 random lines in the unit interval, with `scaling` parameter equal to `0.1`

```
V,EVs = biconnectedComponent(model)
HPCs = [STRUCT(MKPOLS((V,EV))) for EV in EVs]

sets = [COLOR(colors[k%10])(hpc) for k,hpc in enumerate(HPCs)]
VIEW(STRUCT(sets))
VIEW(STRUCT(MKPOLS((V,CAT(EVs)))))

#V,EV = larRemoveVertices(V,CAT(EVs))
◇
```

2-complex from orthogonal line segments

"test/py/inters/test08.py" 39 ≡

```
""" 2-complex from orthogonal line segments """
from larlib import *
colors = [CYAN, MAGENTA, WHITE, RED, YELLOW, GREEN, ORANGE, BLACK, BLUE, PURPLE]

⟨ Orthogonal example 37b ⟩
model = V,EV
V,EVs = biconnectedComponent(model)
HPCs = [STRUCT(MKPOLS((V,EV))) for EV in EVs]

sets = [COLOR(colors[k%10])(hpc) for k,hpc in enumerate(HPCs)]
VIEW(STRUCT(sets))

EV = sorted(CAT(EVs))
VIEW(STRUCT(MKPOLS((V,EV))))

V,FV,EV = facesFromComps((V,EV))

areas = surfIntegration((V,FV,EV))
boundaryArea = max(areas)
FV = [FV[f] for f,area in enumerate(areas) if area!=boundaryArea]
VIEW(EXPLODE(1.2,1.2,1)(MKPOLS((V,FV+EV)) + AA(MK)(V)))
◇
```

Biconnected components from random LAR model

"test/py/inters/test09.py" 41 ≡

```
""" Biconnected components from orthogonal LAR model """
from larlib import *
colors = [CYAN, MAGENTA, YELLOW, RED, GREEN, ORANGE, PURPLE, WHITE, BLACK, BLUE]

lines = randomLines(100,.8)
V,EV = lines2lar(lines)
model = V,EV
```

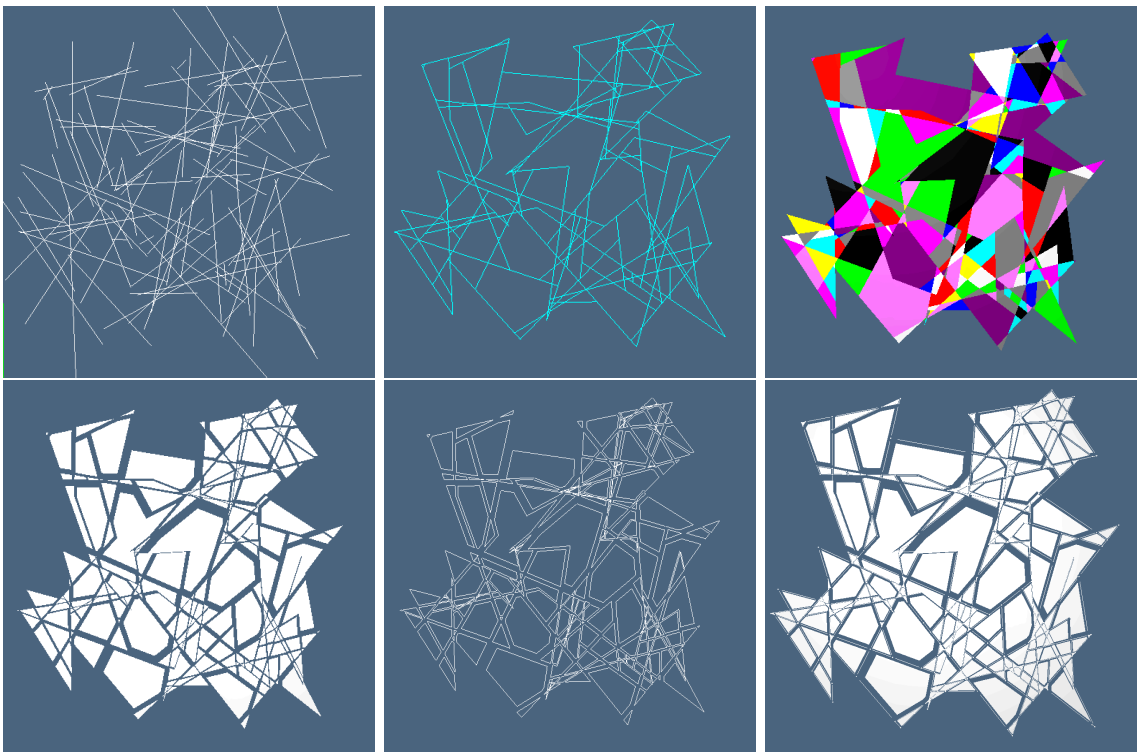

Figure 8: `LAR` complex generation random lines. (a) the input random lines; (b) maximal biconnected graph extracted from the 1D LAR of intersected lines; (c) 2D cells of such *regularized* 2-complex; (d) 2-cells, drawn exploded; (e) boundaries of 2D cells; (f) regularized cellular 2-complex extracted from lines.

```
VIEW(STRUCT(AA(POLYLINE)(lines)))

V,EVs = biconnectedComponent(model)
HPCs = [STRUCT(MKPOLS((V,EV))) for EV in EVs]
sets = [COLOR(colors[k%10])(hpc) for k,hpc in enumerate(HPCs)]
VIEW(STRUCT(sets))

EV = CAT(EVs)
V,EV = larRemoveVertices(V,EV)
V,FV,EV = facesFromComps((V,EV))
areas = surfIntegration((V,FV,EV))
boundaryArea = max(areas)
FV = [FV[f] for f,area in enumerate(areas) if area!=boundaryArea]

polylines = [[V[v] for v in face+[face[0]]] for face in FV]
VIEW(EXPLODE(1.2,1.2,1)(MKPOLS((V,EV)) + AA(MK)(V) + AA(FAN)(polylines) ))

colors = [CYAN, MAGENTA, WHITE, RED, YELLOW, GRAY, GREEN, ORANGE, BLACK, BLUE, PURPLE, BROWN]
sets = [COLOR(colors[k%12])(FAN(pol)) for k,pol in enumerate(polylines)]
VIEW(STRUCT(sets))


VIEW(EXPLODE(1.2,1.2,1)((AA(FAN)(polylines))))
VIEW(EXPLODE(1.2,1.2,1)((AA(POLYLINE)(polylines))))

VV = AA(LIST)(range(len(V)))
submodel = STRUCT(MKPOLS((V,EV)))
VIEW(larModelNumbering(1,1,1)(V,[VV,EV],submodel,0.1))
◇
```

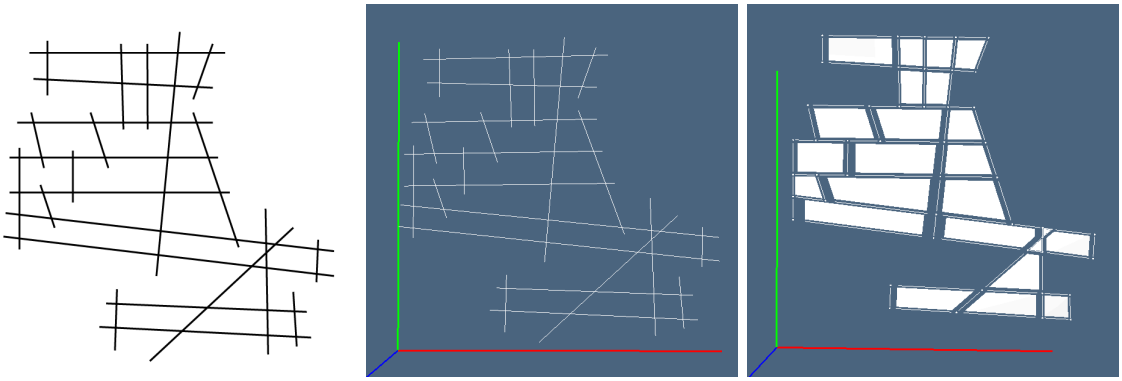

Figure 9: `LAR` complex generation from `SVG` file. (a) the input set of lines; (b) imported in `pyplasm` environment; (c) the extracted *regularized* 2-complex, drawn exploded.

SVG input parsing and transformation We postulate here that the input file `test/py/inters/test.svg` should contain only `<line>` primitives, so we skip any other content. Such primitives are parsed by matching against regular expressions, and their `x1,y1,x2,y2` attributes are extracted and stored into the `lines` variable. An isomorphic window-viewport transformation is then performed, to transform the data within the standard unit 2D square $[0,1]^2$. The input vertices are finally set to a fixed resolution, using the `vcode` function.

⟨ SVG input parsing and transformation 42 ⟩ ≡

```
""" SVG input parsing and transformation """
from larlib import *
import re # regular expression

def svg2lines(filename,containmentBox=[],rect2lines=True):
    stringLines = [line.strip() for line in open(filename)]

    # SVG <line> primitives
    lines = [string.strip() for string in stringLines if re.match("<line ",string)!=None]
    outLines = ""
    for line in lines:
        searchObj = re.search( r'(<line )(.+)(" x1=")(.+)(" y1=")(.+)(" x2=")(.+)(" y2=")(.+)(
        if searchObj:
            outLines += "[["+searchObj.group(4)+","+searchObj.group(6)+"], ["+searchObj.group(8
    if lines != []:
        lines = list(eval(outLines))

    # SVG <rect> primitives
    rects = [string.strip() for string in stringLines if re.match("<rect ",string)!=None]
    outRects,searchObj = "",False
    for rect in rects:
        searchObj = re.search( r'(<rect x=")(.+?)(" y=")(.+?)(" )(.*?)( width=")(.+?)(" height=
        if searchObj:
            outRects += "[["+searchObj.group(2)+","+searchObj.group(4)+"], ["+searchObj.group(8

    if rects != []:
        rects = list(eval(outRects))
        if rect2lines:
            lines += CAT([[[x,y],[x+w,y]],[[x+w,y],[x+w,y+h]],[[x+w,y+h],[x,y+h]],[[x,y+h],[x
        else:
            lines += [[[x,y],[x+w,y+h]] for [x,y],[w,h] in rects]
    for line in lines: print line

    ⟨ SVG input normalization transformation 43 ⟩
    containmentBox = box

    return lines
```

◇

Macro referenced in 34.

SVG input normalization transformation The normalization transformation maps the input `lines` to the $[0,1]^2$ viewport, i.e. to the standard unit square.

⟨ SVG input normalization transformation 43 ⟩ ≡

```
""" SVG input normalization transformation """
# window-viewport transformation
xs,ys = TRANS(CAT(lines))
box = [min(xs), min(ys), max(xs), max(ys)]

# viewport aspect-ratio checking, setting a computed-viewport 'b'
b = [None for k in range(4)]
if (box[2]-box[0])/(box[3]-box[1]) > 1:
    b[0]=0; b[2]=1; bm=(box[3]-box[1])/(box[2]-box[0]); b[1]=.5-bm/2; b[3]=.5+bm/2
else:
    b[1]=0; b[3]=1; bm=(box[2]-box[0])/(box[3]-box[1]); b[0]=.5-bm/2; b[2]=.5+bm/2

# isomorphic 'box -> b' transform to standard unit square
lines = [[[
((x1-box[0])*(b[2]-b[0]))/(box[2]-box[0]) ,
((y1-box[1])*(b[3]-b[1]))/(box[1]-box[3]) + 1], [
((x2-box[0])*(b[2]-b[0]))/(box[2]-box[0]),
((y2-box[1])*(b[3]-b[1]))/(box[1]-box[3]) + 1]]
      for [[x1,y1],[x2,y2]] in lines]

# line vertices set to fixed resolution
lines = eval("".join(['['+ vcode(p1) +','+ vcode(p2) +'], ' for p1,p2 in lines]))
```

◇

Macro referenced in 42.

2-complex extraction from svg file The input `lines` arrangments produces a 1-dimensional complex stored into the `LAR` model `V,EV`. Then the *dangling edges* are removed from `EV_`, and the whole data set is renumbered, in order to remove the unused vertices, using the `larRemoveVertices` function. Finally the 2-cells are computed and stored in `FV`, and the positive areas of every 2cells are computed, so allowing for identify and removal of the exterior face, corresponding to the boundary of the complex. The polygonal boundary of the complex is finally drawn.

"test/py/inters/test10.py" 44a ≡

```
""" Biconnected components from orthogonal LAR model """
from larlib import *

filename = "test/py/inters/plan.svg"
```

```
#filename = "test/py/inters/building.svg"
#filename = "test/py/inters/complex.svg"
lines = svg2lines(filename)
VIEW(STRUCT(AA(POLYLINE)(lines)))

V,FV,EV = larFromLines(lines)
VIEW(EXPLODE(1.2,1.2,1)(MKPOLS((V,FV[:-1]+EV)) + AA(MK)(V)))

VV = AA(LIST)(range(len(V)))
submodel = STRUCT(MKPOLS((V,EV)))
VIEW(larModelNumbering(1,1,1)(V,[VV,EV,FV[:-1]],submodel,0.05))


verts,faces,edges = polyline2lar([[ V[v] for v in FV[-1] ]])
VIEW(STRUCT(MKPOLS((verts,edges))))
◇
```

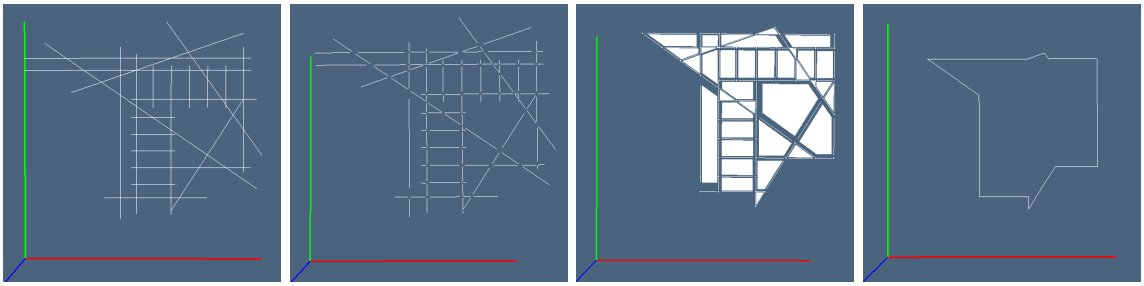

Figure 10: LAR complex generation from SVG file. (a) the input set of lines parsed from an SVG file; (b) the intersection of lines; (c) the extracted *regularized* 2-complex, drawn exploded; (d) the boundary LAR.

"test/py/inters/test11.py" 44b ≡

```
""" Fast Polygon Triangulation based on Seidel's Algorithm """
# data generated by test10.py on file polygon.svg
from larlib import *

V,FV,EV = ([[0.222, 0.889],
  [0.722, 1.0],
  [0.519, 0.763],
  [1.0, 0.659],
  [0.859, 0.233],
  [0.382, 0.119],
  [0.519, 0.348],
  [0.296, 0.53],
```

```
  [0.0, 0.059]],
 [[0, 1, 2, 3, 4, 5, 6, 7, 8]],
 [[2, 3], [6, 7], [0, 8], [3, 4], [1, 2], [7, 8], [4, 5], [5, 6], [0, 1]])

VV = AA(LIST)(range(len(V)))
submodel = STRUCT(MKPOLS((V,EV)))
VIEW(larModelNumbering(1,1,1)(V,[VV,EV],submodel,0.5))


xord = TRANS(sorted(zip(V,range(len(V))))[1]
trapezoids = zip(xord[:-1],xord[1:])
vert2forw_trap = dict()
vert2back_trap = dict()

for k,(a,b) in enumerate(trapezoids[1:-1]):
    print k,(a,b)
    vert2back_trap[a]=k
    vert2forw_trap[a]=k+1
    vert2back_trap[b]=k+1
    vert2forw_trap[b]=k+2
vert2forw_trap[trapezoids[0][0]] = 0
vert2back_trap[trapezoids[-1][1]] = len(trapezoids)-1
◇
```

"test/py/inters/test12.py" 45 ≡

```
""" Biconnected components from orthogonal LAR model """
from larlib import *

V = [[0.395, 0.296], [0.593, 0.0], [0.79, 0.773], [0.671, 0.889], [0.79, 0.0], [0.593, 0.296],
FV = [[0, 5, 4, 1], [1, 9, 0], [8, 7, 0, 9], [7, 8, 3, 2, 4, 5, 6]]
EV = [[0, 1], [8, 9], [6, 7], [4, 5], [1, 4], [3, 8], [5, 6], [2, 3], [1, 9], [0, 9], [0, 5],
polylines = [[V[v] for v in face+[face[0]]] for face in FV]
VIEW(EXPLODE(1.1,1.1,1)(MKPOLS((V,EV)) + AA(MK)(V) + AA(FAN)(polylines) ))

VV = AA(LIST)(range(len(V)))
submodel = STRUCT(MKPOLS((V,EV)))
VIEW(larModelNumbering(1,1,1)(V,[VV,EV,FV],submodel,.6))

VIEW(EXPLODE(1.1,1.1,1)(AA(POLYLINE)(polylines)))
◇
```

A Code utilities

Coding utilities Some utility fuctions used by the module are collected in this appendix. Their macro names can be seen in the below script.

⟨ Coding utilities 46a ⟩ ≡

```
""" Coding utilities """
⟨ Generation of a random point 47a ⟩
⟨ Generation of a random line segment 47b ⟩
⟨ Transformation of a 2D box into a closed polyline 47c ⟩
⟨ Computation of the 1D centroid of a list of 2D boxes 47d ⟩
⟨ Pyplasm XOR of FAN of ordered points 48 ⟩
◇
```

Macro referenced in 34.

Generation of random lines The function `randomLines` returns the array `randomLineArray` with a given number of lines generated within the unit 2D interval. The `scaling` parameter is used to scale every such line, generated by two randow points, that could be possibly located to far from each other, even at the distance of the diagonal of the unit square.

The arrays `xs` and `ys`, that contain the x and y coordinates of line points, are used to compute the minimal translation `v` needed to transport the entire set of data within the positive quadrant of the 2D plane.

⟨ Generation of random lines 46b ⟩ ≡

```
""" Generation of random lines """
def randomLines(numberOfLines=200,scaling=0.3):
    randomLineArray = [redge(scaling) for k in range(numberOfLines)]
    [xs,ys] = TRANS(CAT(randomLineArray))[:2]
    xmin, ymin = min(xs), min(ys)
    v = array([-xmin,-ymin])
    randomLineArray = [[list(v1[:2]+v), list(v2[:2]+v)] for v1,v2 in randomLineArray]
    return randomLineArray
◇
```

Macro referenced in 34.

Generation of a random point A single random point, codified in floating point format, and with a fixed (quite small) number of digits, is returned by the `rpoint2d()` function, with no input parameters.

⟨ Generation of a random point 47a ⟩ ≡

```
""" Generation of a random point """
def rpoint2d():
    return eval( vcode([ random.random(), random.random() ]) )
◇
```

Macro referenced in 46a.

Generation of a random line segment A single random segment, scaled about its centroid by the `scaling` parameter, is returned by the `redge()` function, as a tuple ot two random points in the unit square.

⟨ Generation of a random line segment 47b ⟩ ≡

```
""" Generation of a random line segment """
def redge(scaling):
    v1,v2 = array(rpoint2d()), array(rpoint2d())
    c = (v1+v2)/2
    pos = rpoint2d()
    v1 = (v1-c)*scaling + pos
    v2 = (v2-c)*scaling + pos
    return tuple(eval(vcode(v1))), tuple(eval(vcode(v2)))
```

◇

Macro referenced in 46a.

Transformation of a 2D box into a closed polyline The transformation of a 2D box into a closed rectangular polyline, given as an ordered sequwncw of 2D points, is produced by the function `box2rect`

⟨ Transformation of a 2D box into a closed polyline 47c ⟩ ≡

```
""" Transformation of a 2D box into a closed polyline """
def box2rect(box):
    x1,y1,x2,y2 = box
    verts = [[x1,y1],[x2,y1],[x2,y2],[x1,y2],[x1,y1]]
    return verts
```

◇

Macro referenced in 46a.

Computation of the 1D centroid of a list of 2D boxes The 1D `centroid` of a list of 2D boxes is computed by the function given below. The direction of computation (either x or y) is chosen depending on the value of the `xy` parameter.

⟨ Computation of the 1D centroid of a list of 2D boxes 47d ⟩ ≡

```
""" Computation of the 1D centroid of a list of 2D boxes """
def centroid(boxes,coord):
    delta,n = 0,len(boxes)
    ncoords = len(boxes[0])/2
    a = coord%ncoords
    b = a+ncoords
    for box in boxes:
        delta += (box[a] + box[b])/2
    return delta/n
```

◇

Macro referenced in 46a.

Pyplasm XOR of FAN of ordered points

⟨ Pyplasm XOR of FAN of ordered points 48 ⟩ ≡

```
"""" XOR of FAN of ordered points """"
def FAN(points):
    pairs = zip(points[1:-2],points[2:-1])
    triangles = [MKPOL([[points[0],p1,p2],[[1,2,3]],None]) for p1,p2 in pairs]
    return XOR(triangles)

if __name__=="__main__":
    pol = [[0.476,0.332],[0.461,0.359],[0.491,0.375],[0.512,0.375],[0.514,0.375],
    [0.527,0.375],[0.543,0.34],[0.551,0.321],[0.605,0.314],[0.602,0.307],[0.589,
    0.279],[0.565,0.244],[0.559,0.235],[0.553,0.227],[0.527,0.239],[0.476,0.332]]

    VIEW(EXPLODE(1.2,1.2,1)(FAN(pol)))
```

◇

Macro referenced in 46a.

References

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

[HT73] John Hopcroft and Robert Tarjan, *Algorithm 447: Efficient algorithms for graph manipulation*, Commun. ACM **16** (1973), no. 6, 372–378.

[PBCF93] A. Paoluzzi, F. Bernardini, C. Cattani, and V. Ferrucci, *Dimension-independent modeling with simplicial complexes*, ACM Trans. Graph. **12** (1993), no. 1, 56–102.

[YT85] F. Yamaguchi and T. Tokieda, *Bridge edge and triangulation approach in solid modeling*, Frontiers in Computer Graphics (Berlin) (T.L. Kunii, ed.), Springer Verlag, 1985.