Accelerated intersection of geometric objects [*]

Alberto Paoluzzi

September 29, 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]. September 29, 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 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 .

Splitting the input above and below a threshold

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

```python
""" 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 .

Iterative splitting of box buckets

⟨Iterative splitting of box buckets 3a⟩ ≡

```python
""" 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):
```

3

```
        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 17a.

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)
```

4

```
                        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 17a.

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 17a.

Accelerate intersection of lines

⟨ Accelerate intersection of lines 5 ⟩ ≡

```
    """ Accelerate intersection of lines """
    def lineIntersection(lineArray):

        from collections import defaultdict
        pointStorage = defaultdict(list)
        for line in lineArray:
            p1,p2 = line
            key = '['+ vcode(p1) +','+ vcode(p2) +']'
            pointStorage[key] = []

        boxes = containment2DBoxes(lineArray)
        buckets = boxBuckets(boxes)
        intersectionPoints = set()
        for h,bucket in enumerate(buckets):
```

5

```
                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 .

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 ver-
tices, 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 ⟩ ≡

```python
    """ 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])])

        """
        # identification of close vertices
        #closePairs = scipy.spatial.KDTree(V).query_pairs(10**(-PRECISION+1))
        closePairs = scipy.spatial.cKDTree(V).query_pairs(10**(-PRECISION+2))
        if closePairs != []:
            EV_ = []
            for v1,v2 in EV:
                for v,w in closePairs:
                    if v1 == w: v1 = v
                    if v2 == w: v2 = v
                EV_ += [[v1,v2]]
            EV = EV_

        # Remove zero edges
        EV = list(set([ tuple(sorted([v1,v2])) for v1,v2 in EV if v1!=v2 ]))
        return (V,EV)
        """
        model = (V,EV)
        return larSimplify(model)
    ◇
```

Macro referenced in 17a.

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 8a ⟩
⟨ Hopcroft-Tarjan algorithm 8b ⟩
⟨ Output of biconnected components 9 ⟩
◇

Macro referenced in 17a.

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 8a ⟩ ≡

```
""" 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

Hopcroft-Tarjan algorithm

⟨Hopcroft-Tarjan algorithm 8b⟩ ≡
```
    """ 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

Output of biconnected components

⟨Output of biconnected components 9⟩ ≡
```
    """ 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

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

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 each non-zero (sparse) matrix element $\texttt{angle\_EV}(h, k)$ 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 $\texttt{angle\_EV}$, i.e. the rows of $\texttt{angle\_VE} := \texttt{angle\_EV}^t$, 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 11a ⟩ ≡

```
""" 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 17a.

11

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 11b ⟩ ≡

```
""" 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 .

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 12 ⟩ ≡

```
""" 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 17a.

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 $csrVE(k, h)$ 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.

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 14a ⟩ ≡

```
""" Transformation of an array of lines in a 2D LAR complex """
def larFromLines(lines):
    V,EV = lines2lar(lines)
    V,EVs = biconnectedComponent((V,EV))
    EV = list(set(AA(tuple)(sorted(AA(sorted)(CAT(EVs))))))
    V,EV = larRemoveVertices(V,EV)
    V,FV,EV = facesFromComps((V,EV))
    areas = integr.surfIntegration((V,FV,EV))
    boundaryArea = max(areas)
    interiorFaces = [FV[f] for f,area in enumerate(areas) if area!=boundaryArea and len(areas):
    boundaryFace = FV[areas.index(boundaryArea)]
    return V,interiorFaces+[boundaryFace],EV
```
◇

Macro referenced in .

2.6 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 14b⟩ ≡

```
    """ 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 17a.

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 15 ⟩ ≡
```
    """ Return a simplified LAR model """
    def larSimplify(model):
        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)
        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(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]
            return W,CV,FV
        else: return W,CV
```
◇

Macro referenced in 17a.

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" 16a ≡
```
    """ 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]
```

16

```
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" 16b ≡`

```
""" 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]))
◇
```

3   Exporting the module

`"larlib/larlib/inters.py" 17a ≡`

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

⟨ Coding utilities 28b ⟩
⟨ Generation of random lines 29a ⟩
⟨ Containment boxes 2a ⟩
⟨ Splitting the input above and below a threshold 2b ⟩
⟨ Box metadata computation ? ⟩
⟨ Iterative splitting of box buckets 3a ⟩

4 Examples

Generation of random line segments and their boxes

"test/py/inters/test01.py" 17b ≡

```
""" 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" 18a ≡

```
""" 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" 18b ≡

```
""" 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" 19a ≡

```
""" 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" 19b ≡
```
""" LAR from splitting of othogonal lines """
from larlib import *
⟨ Orthogonal example 19c ⟩
◇
```

⟨ Orthogonal example 19c ⟩ ≡

```
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 19b, 20b, 22a.
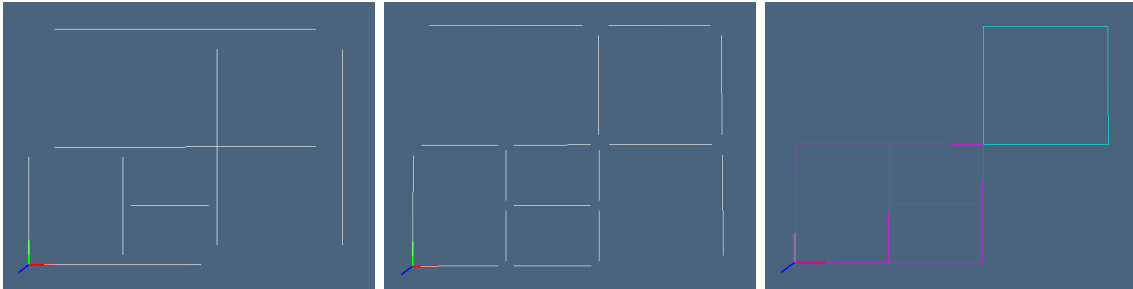
Figure 2: Splitting of orthogonal lines: (a) exploded input; (a) exploded output; (c) biconnected components.

Random coloring of the generated 1-complex LAR

"test/py/inters/test06.py" 20a ≡
```
""" 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))
◇
```
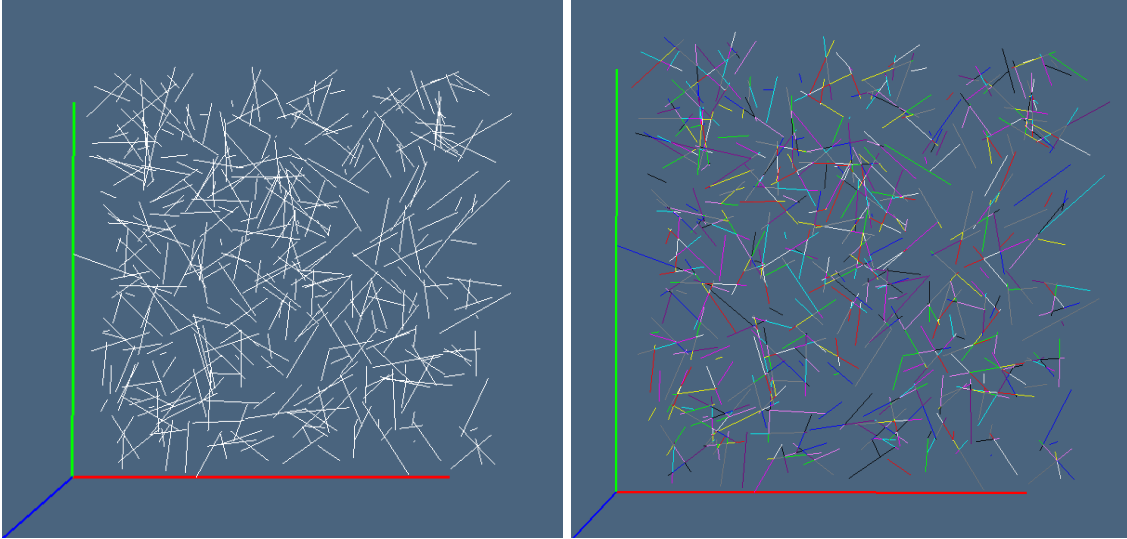

Figure 3: Splitting of intersecting lines: (a) random input; (a) splitted and colored LAR output.

Biconnected components from orthogonal LAR model

"test/py/inters/test07.py" 20b ≡

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

⟨Orthogonal example 19c⟩
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))
```
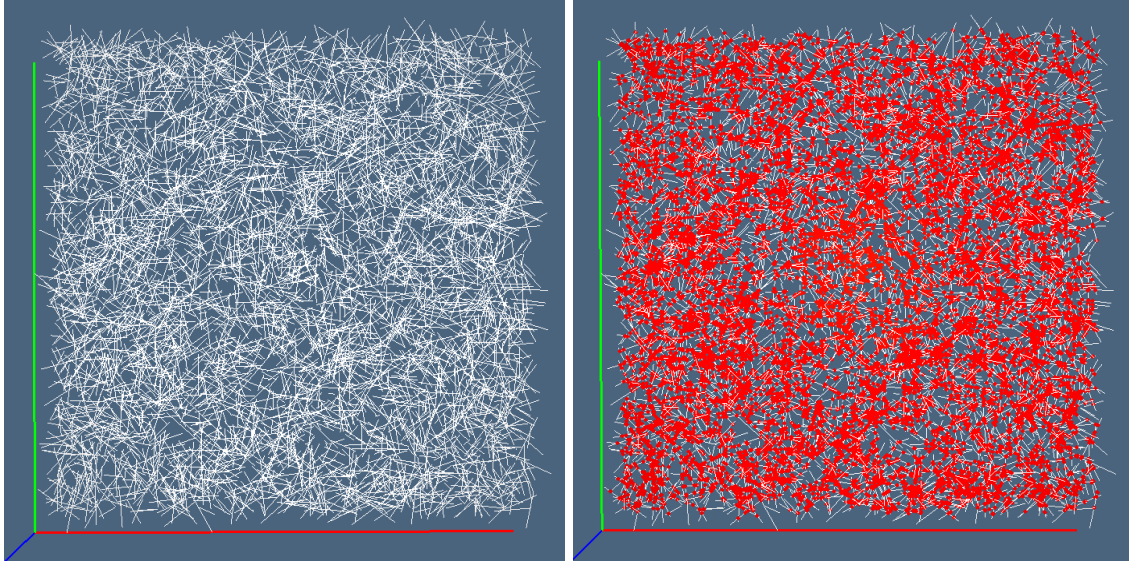
21

Figure 4: The intersection of 5000 random lines in the unit interval, with `scaling` parameter equal to `0.1`

```
VIEW(STRUCT(MKPOLS((V,CAT(EVs)))))

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

2-complex from orthogonal line segments

"test/py/inters/test08.py" 22a ≡
```
""" 2-complex from orthogonal line segments """
from larlib import *
colors = [CYAN, MAGENTA, WHITE, RED, YELLOW, GREEN, ORANGE, BLACK, BLUE, PURPLE]

⟨Orthogonal example 19c⟩
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))
```

22

```
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)))
◊
```
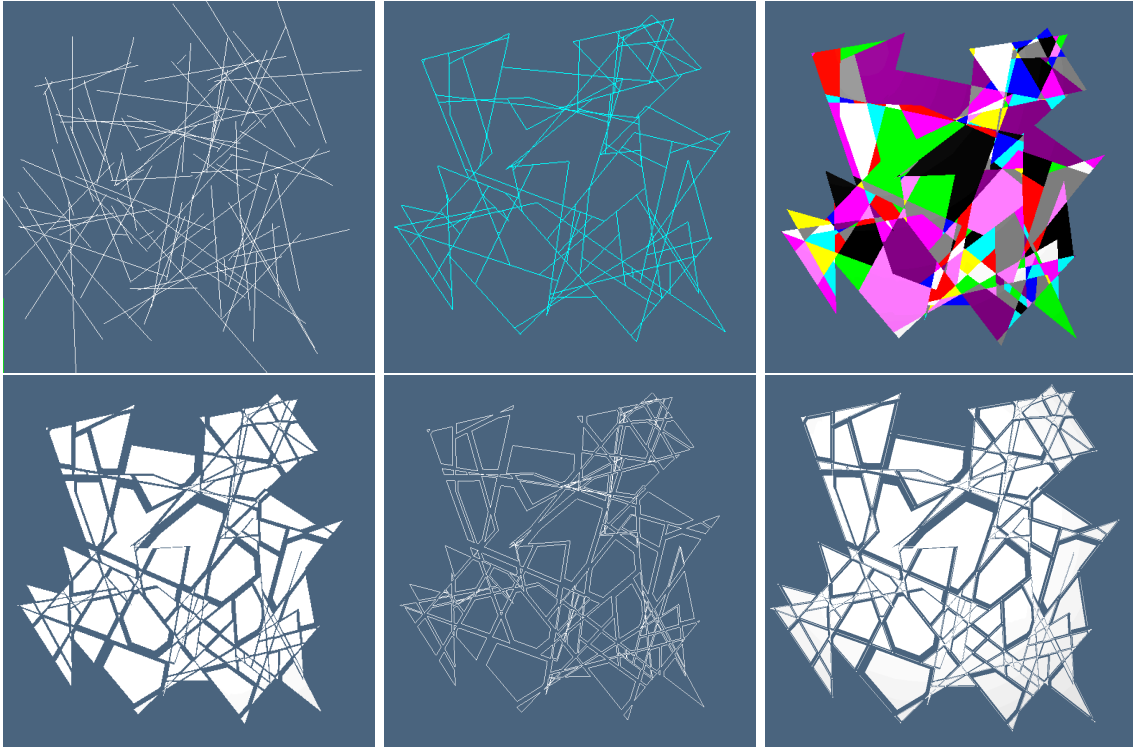

Figure 5: `LAR` complex generation random lines. (a) the input random lines; (b) maximal biconnected graph extracted from the 1D LAR of intersected lines; (c) 2D cells of such *regularized* 2-complex; (d) 2-cells, drawn exploded; (e) boundaries of 2D cells; (f) regularized cellular 2-complex extracted from lines.

Biconnected components from random LAR model

"test/py/inters/test09.py" 22b ≡

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

    lines = randomLines(100,.8)
```

23

```
    V,EV = lines2lar(lines)
    model = V,EV
    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))
    ◇
```

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 25⟩ ≡

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

    def svg2lines(filename,containmentBox=[],rect2lines=True):
```
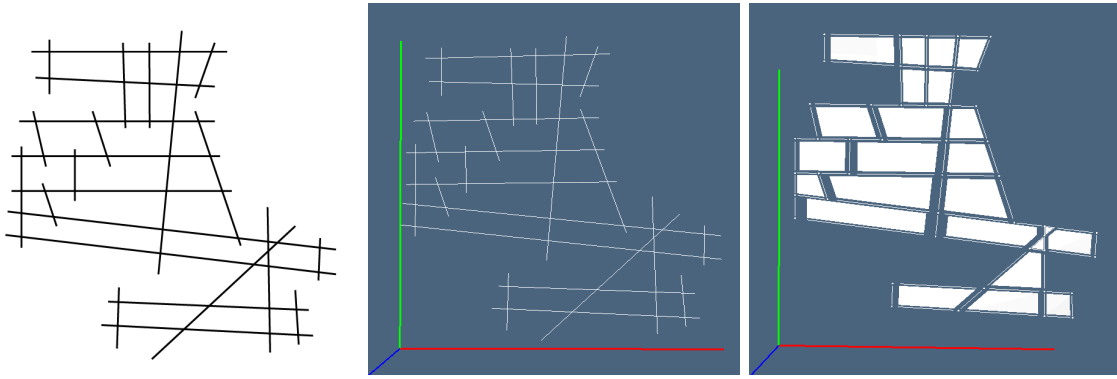
Figure 6: LAR complex generation from SVG file. (a) the input set of lines; (b) imported in pyplasm environment; (c) the extracted *regularized* 2-complex, drawn exploded.

```
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 26a ⟩

25

```
        containmentBox = box

        return lines
    ◇
```

Macro referenced in 17a.

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 26a⟩ ≡
```
    """ 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 25.

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"` 26b ≡

```
""" 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))))
◇
```
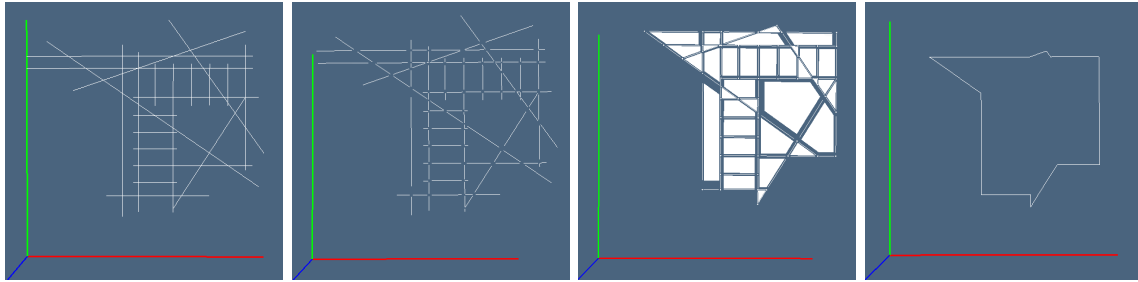

Figure 7: `LAR` complex generation from `SVG` file. (a) the input set of lines parsed from an `SVG` file; (b) the intersection of lines; (c) the extracted *regularized* 2-complex, drawn exploded; (d) the boundary `LAR`.

"test/py/inters/test11.py" 27 ≡
```
""" 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" 28a ≡

```
    """ 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 28b ⟩ ≡

```
    """ Coding utilities """
    ⟨ Generation of a random point 29b ⟩
    ⟨ Generation of a random line segment 29c ⟩
    ⟨ Transformation of a 2D box into a closed polyline 30a ⟩
    ⟨ Computation of the 1D centroid of a list of 2D boxes 30b ⟩
    ⟨ Pyplasm XOR of FAN of ordered points 31 ⟩
    ◇
```

Macro referenced in 17a.

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 29a ⟩ ≡

```
    """ 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 17a.

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 `rpoint()` function, with no input parameters.

⟨ Generation of a random point 29b ⟩ ≡

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

Macro referenced in 28b.

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 29c ⟩ ≡
```
""" Generation of a random line segment """
def redge(scaling):
    v1,v2 = array(rpoint()), array(rpoint())
    c = (v1+v2)/2
    pos = rpoint()
    v1 = (v1-c)*scaling + pos
    v2 = (v2-c)*scaling + pos
    return tuple(eval(vcode(v1))), tuple(eval(vcode(v2)))
```
◇
Macro referenced in 28b.

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 30a ⟩ ≡
```
""" 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 28b.

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 30b ⟩ ≡
```
""" 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 28b.

Pyplasm XOR of FAN of ordered points

⟨ Pyplasm XOR of FAN of ordered points 31 ⟩ ≡

```
""" 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 28b.

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.