Accelerated intersection of geometric objects *

Alberto Paoluzzi

February 4, 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

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

*This document is part of the *Linear Algebraic Representation with CoChains* (LAR-CC) framework [CL13]. February 4, 2015

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 smallest and smallest 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 subsets. 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 `containmentBoxes` 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 containmentBoxes(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 8b.

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,xy='x'):
    theBoxes = [boxes[k] for k in subset]
    threshold = centroid(theBoxes,xy)
    if xy=='x': a=0;b=2;
    elif xy=='y': a=1;b=3;
    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 8b.

Iterative splitting of box buckets

⟨ Iterative splitting of box buckets 3a ⟩ ≡

```
""" Iterative splitting of box buckets """
def boxBuckets(boxes):
    bucket = range(len(boxes))
    splittingStack = [bucket]
    finalBuckets = []
    while splittingStack != []:
        bucket = splittingStack.pop()
        below,above = splitOnThreshold(boxes,bucket,'x')
        below1,above1 = splitOnThreshold(boxes,above,'y')
        below2,above2 = splitOnThreshold(boxes,below,'y')

        if (len(below1)<4 and len(above1)<4) or len(set(bucket).difference(below1))<7 \
            or len(set(bucket).difference(above1))<7:
            finalBuckets.append(below1)
            finalBuckets.append(above1)
        else:
            splittingStack.append(below1)
            splittingStack.append(above1)

        if (len(below2)<4 and len(above2)<4) or len(set(bucket).difference(below2))<7 \
            or len(set(bucket).difference(above2))<7:
            finalBuckets.append(below2)
            finalBuckets.append(above2)
        else:
            splittingStack.append(below2)
```

```
                splittingStack.append(above2)
        return list(set(AA(tuple)(finalBuckets)))
```
◇

Macro referenced in 8b.

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(pointStorage):
    def segmentIntersect0(segment1):
        p1,p2 = segment1
        line1 = '['+ vcode(p1) +','+ vcode(p2) +']'
        (x1,y1),(x2,y2) = p1,p2
        #B1,B2,B3,B4 = eval(vcode([min(x1,x2),min(y1,y2),max(x1,x2),max(y1,y2)]))
        def segmentIntersect1(segment2):
            p3,p4 = segment2
            line2 = '['+ vcode(p3) +','+ vcode(p4) +']'
            (x3,y3),(x4,y4) = p3,p4
            #b1,b2,b3,b4 = eval(vcode([min(x3,x4),min(y3,y4),max(x3,x4),max(y3,y4)]))
            #if ((B1<=b1<=B3) or (B1<=b3<=B3)) and ((B2<=b2<=B4) or (B2<=b4<=B4)):
            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<=alpha<=1 and 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 8b.

Brute force bucket intersection

⟨ Brute force bucket intersection 4 ⟩ ≡

```
""" Brute force bucket intersection """
def lineBucketIntersect(lines,pointStorage):
    intersect0 = segmentIntersect(pointStorage)
    intersectionPoints = []
    n = len(lines)
    for k,line in enumerate(lines):
        intersect1 = intersect0(line)
        for h in range(k+1,n):
            line1 = lines[h]
            point = intersect1(line1)
            if point != None:
                intersectionPoints.append(eval(vcode(point)))
    return intersectionPoints
```

◇

Macro referenced in 8b.

Accelerate intersection of lines

⟨ Accelerate intersection of lines 5a ⟩ ≡

```
""" 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 = containmentBoxes(lineArray)
    buckets = boxBuckets(boxes)
    intersectionPoints = set()
    for bucket in buckets:
        lines = [lineArray[k] for k in bucket]
        pointBucket = lineBucketIntersect(lines,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 8b.

2.3 Generation of LAR representation of split segments

Create the LAR of fragmented lines

⟨ Create the LAR of fragmented lines 5b ⟩ ≡

```
""" Create the LAR of fragmented lines """
def lines2lar(lineArray):
    intersectionPoints,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])])
    return V,EV
```

◇

Macro referenced in 8b.

2.4 Biconnected components of a 1-complex

Biconnected components

⟨ Biconnected components 6a ⟩ ≡

```
""" Biconnected components """
```

⟨ Adjacency lists of 1-complex vertices 6b ⟩
⟨ Main procedure for biconnected components 7a ⟩
⟨ Hopcroft-Tarjan algorithm 7b ⟩
⟨ Output of biconnected components 8a ⟩

◇

Macro referenced in 8b.

Adjacency lists of 1-complex vertices

⟨ Adjacency lists of 1-complex vertices 6b ⟩ ≡

```
""" 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 6a.

Main procedure for biconnected components

⟨ Main procedure for biconnected components 7a ⟩ ≡

```
""" 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 [component for component in out if len(component) > 1]
```

◇

Macro referenced in 6a.

Hopcroft-Tarjan algorithm

⟨ Hopcroft-Tarjan algorithm 7b ⟩ ≡

```
""" 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 6a.

Output of biconnected components

⟨ Output of biconnected components 8a ⟩ ≡

```
""" 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 6a.

3 Exporting the module

"lib/py/inters.py" 8b ≡

```
""" Module for pipelined intersection of geometric objects """
from pyplasm import *
""" import modules from larcc/lib """
import sys
sys.path.insert(0, 'lib/py/')
from larcc import *
```

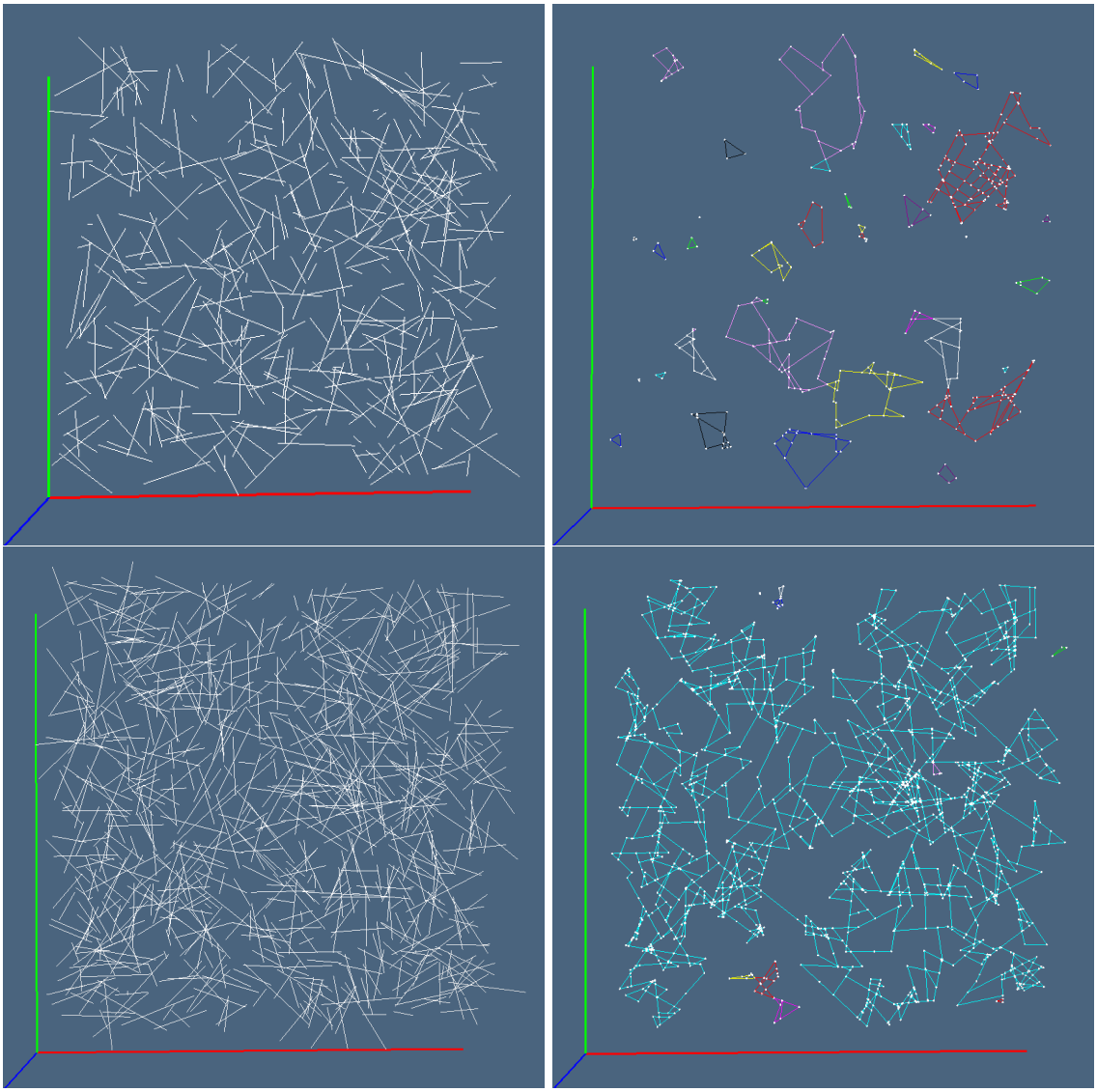

Figure 1: Two random line arrangements, and the biconnected components extracted by their LAR 1-complexes.

```
DEBUG = True
```

⟨ Coding utilities 14c ⟩
⟨ Generation of random lines 15a ⟩
⟨ 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 5a ⟩
⟨ Create the LAR of fragmented lines 5b ⟩
⟨ Biconnected components 6a ⟩
◇

4 Examples

Generation of random line segments and their boxes

"test/py/inters/test01.py" 8c ≡

```
""" Generation of random line segments and their boxes """
import sys
sys.path.insert(0, 'lib/py/')
from inters import *

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

boxes = containmentBoxes(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" 10a ≡

```
""" Split segment array in four independent buckets """
import sys
sys.path.insert(0, 'lib/py/')
from inters import *

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

```
bucket = range(len(boxes))
below,above = splitOnThreshold(boxes,bucket,'x')
below1,above1 = splitOnThreshold(boxes,above,'y')
below2,above2 = splitOnThreshold(boxes,below,'y')

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" 10b ≡

```
""" Generation and random coloring of independent line buckets """
import sys
sys.path.insert(0, 'lib/py/')
from inters import *

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

boxes = containmentBoxes(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)]

VIEW(STRUCT(sets))
◇
```

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

"test/py/inters/test04.py" 11a ≡

```
""" LAR of random line arrangement """
import sys
sys.path.insert(0, 'lib/py/')
from inters import *

lines = randomLines(400,0.2)
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)
#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" 11b ≡

```
""" LAR from splitting of othogonal lines """
import sys
sys.path.insert(0, 'lib/py/')
from inters import *
⟨ Orthogonal example 12a ⟩
◇
```

⟨ Orthogonal example 12a ⟩ ≡

```
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 11b, 14a.

Random coloring of the generated 1-complex LAR

"test/py/inters/test06.py" 12b ≡

```
""" Random coloring of the generated 1-complex """
import sys
sys.path.insert(0, 'lib/py/')
from inters import *

lines = randomLines(400,0.2)
VIEW(STRUCT(AA(POLYLINE)(lines)))
```

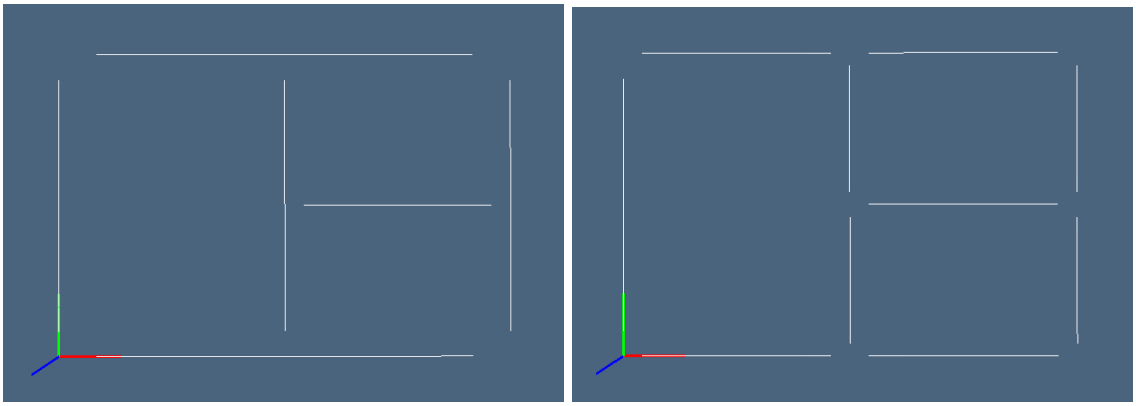

Figure 2: Splitting of othogonal lines: (a) exploded input; (a) exploded output.

```
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" 14a ≡

```
""" Biconnected components from orthogonal LAR model """
import sys
sys.path.insert(0, 'lib/py/')
from inters import *
colors = [CYAN, MAGENTA, WHITE, RED, YELLOW, GREEN, ORANGE, BLACK, BLUE, PURPLE]

⟨ Orthogonal example 12a ⟩
model = V,EV
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))
◇
```

Biconnected components from random LAR model

"test/py/inters/test08.py" 14b ≡

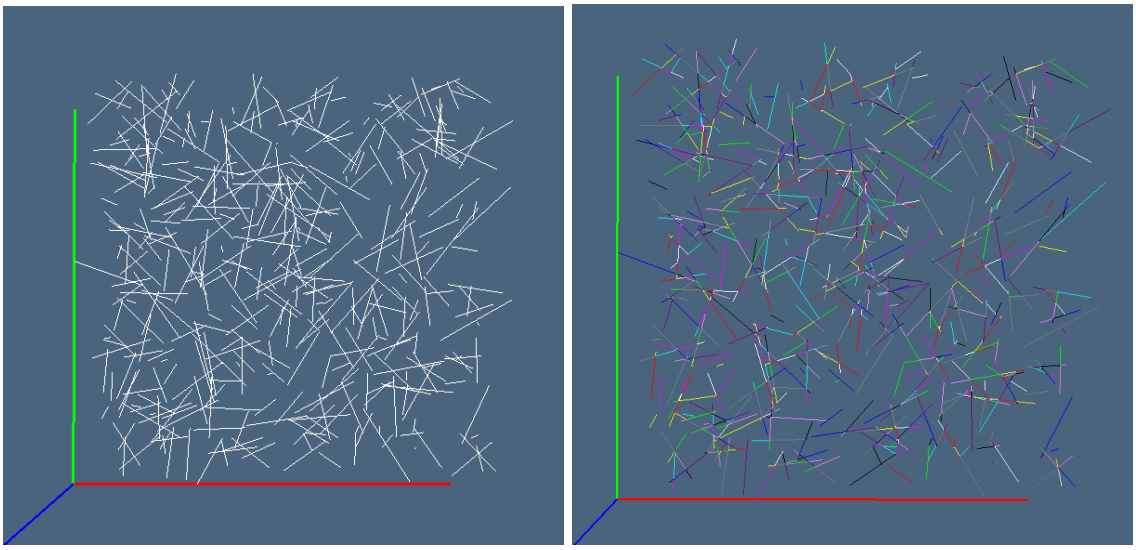

Figure 3: Splitting of intersecting lines: (a) random input; (a) splitted and colored LAR output.

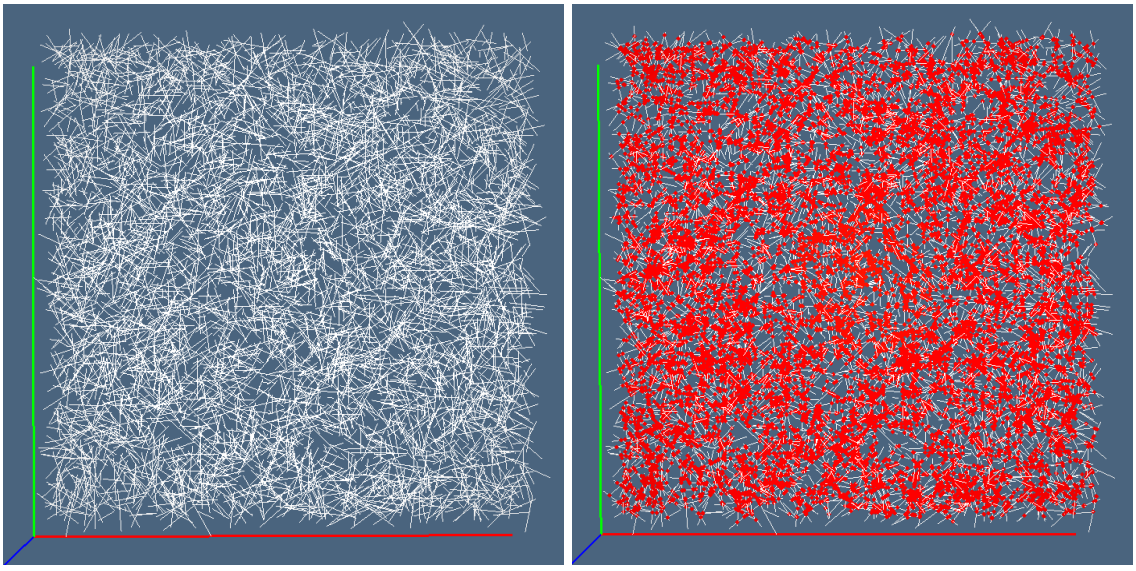

Figure 4: The intersection of 5000 random lines in the unit interval, with `scaling` parameter equal to `0.1`

```
""" Biconnected components from orthogonal LAR model """
import sys
sys.path.insert(0, 'lib/py/')
from inters import *
colors = [CYAN, MAGENTA, YELLOW, RED, GREEN, ORANGE, PURPLE, WHITE, BLACK, BLUE]

lines = randomLines(800,0.2)
V,EV = lines2lar(lines)
model = V,EV
EVs = biconnectedComponent(model)
HPCs = [STRUCT(MKPOLS((V,ev))) for ev in EVs if len(ev)>1]
verts = list(set(CAT(CAT([ev for ev in EVs if len(ev)>1]))))

sets = [COLOR(colors[k%10])(hpc) for k,hpc in enumerate(HPCs)]
VIEW(STRUCT(sets+AA(MK)([V[v] for v in verts])))
VIEW(STRUCT(AA(POLYLINE)(lines)))
◇
```

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 14c ⟩ ≡

```
""" Coding utilities """
⟨ Generation of a random point 15b ⟩
⟨ Generation of a random line segment 15c ⟩
⟨ Transformation of a 2D box into a closed polyline 16a ⟩
⟨ Computation of the 1D centroid of a list of 2D boxes 16b ⟩
◇
```

Macro referenced in 8b.

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 15a ⟩ ≡

```
""" Generation of random lines """
def randomLines(numberOfLines=200,scaling=0.3):
    randomLineArray = [redge(scaling) for k in range(numberOfLines)]
```

```
    [xs,ys] = TRANS(CAT(randomLineArray))
    xmin, ymin = min(xs), min(ys)
    v = array([-xmin,-ymin])
    randomLineArray = [[list(v1+v), list(v2+v)] for v1,v2 in randomLineArray]
    return randomLineArray
```
◇

Macro referenced in 8b.

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 15b ⟩ ≡
```
""" Generation of a random point """
def rpoint():
    return eval( vcode([ random.random(), random.random() ]) )
```
◇

Macro referenced in 14c.

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 15c ⟩ ≡
```
""" 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 14c.

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 16a ⟩ ≡
```
""" 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 14c.

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 16b ⟩ ≡

```
""" Computation of the 1D centroid of a list of 2D boxes """
def centroid(boxes,xy='x'):
    delta,n = 0,len(boxes)
    if xy=='x': a=0; b=2
    elif xy=='y': a=1; b=3
    for box in boxes:
        delta += (box[a] + box[b])/2
    return delta/n
```

◇

Macro referenced in 14c.

References

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