Relatório do Projeto 2

Daniel Moreira Cestari - 5746193

1 de dezembro de 2017

1 Introdução

O objetivo do Projeto 2 é o desenvolvimento de uma triangulação *Delaunay* 2D.

A definição de uma triangulação *Delaunay* é uma triangulação de pontos tal que nenhum ponto do conjunto inicialmente fornecido está dentro do circuncirculo de qualquer triângulo da triangulação. Essa propriedade de nenhum ponto externo aos triângulos estar dentro do circuncirculo, garante que o mínimo ângulo dos triângulos será maximizado. É provado que dado um conjunto de pontos no espaço Euclidiano, a triangulação *Delaunay* desses pontos é única, isso permite que a implementação deste trabalho possa ser testada utilizando outra biblioteca que já a implementa.

A entrada do algoritmo é um conjunto de pontos, e como saída o programa retorna na representação de uma *Corner table* a triangulação *Delaunay* dos pontos passados. O algoritmo para a triangulação empregado foi o algoritmo de inserção de pontos incrementalmente com *flip* de arestas.

A estrutura de dados *Corner table* foi implementada em atividade passadas da disciplina, mas até então possuía apenas operações de conjunta básica sobre os triângulos representados pela estrutura. Essas operações eram: fecho, estrela, anel, e *link*. Para este trabalho foi necessário a inclusão de mais operações, como adição, remoção de triângulos, adição de vértices, busca de triângulos que compartilhem aresta, encontrar o triângulo que um dado ponto se encontra, coordenadas baricêntricas de um dado ponto, teste de orientação de pontos, teste do incirculo e *flip* de arestas.

2 Implementação

Nesta seção é apresentada a implementação do código.

Abaixo é apresentado o esquema geral da implementação:

- Determinação de um triângulo contendo os dados fornecidos (triângulo externo);
- Inserção incremental dos pontos fornecidos;
  - Correção dos triângulos pelo teste do incírculo e *flip* de arestas;
- Remoção de todos os triângulos com vértices do triângulo externo;
- Limpeza da estrutura de dados de objetos eliminados.

Abaixo é exibida a função que calcula a triangulação:

```
def delaunay_triangulation(pts, plot=False, legalize_plot=False,
                                    legalize=True, remove_outer=False
                                    ):
  """
########################################
# Perform the Delaunay triangulation a set of points
########################################
# points: List. Each element is a n-dimensional point
########################################
# Return a structure with the delaunay triangulation.

# Usage example:

# simple example
import corner_table as cnt
import project2 as pjt
import imp

pts = [(0,1), (3,1), (5,0), (2,2), (4,2), (1,0)]

pts2 = [(0,1), (3,1), (5,0), (2,2), (4,2), (1,0), (3, 1.9)]

outer_tr = [(5.0, 9.0777472107017552), (8.2455342898166109, -5.
                                    2039371148119731), (-5.
                                    7455342898166091, -0.
                                    873810095889783 42)]

imp.reload(pjt); dd = pjt.delaunay_triangulation(pts2)


# example insert point into an edge
import corner_table as cnt
import project2 as pjt
import imp

pts = [(0,0), (1,0), (2,0), (0,1), (1,1), (2,1), (1,2), (1,3),]
pts = [(1,3), (1,1), (1,2), (0,0), (1,0), (2,0), (0,1), (2,1),]
pts = [(1,3), (1,1), (1,2)]

outer_tr = [(5.0, 9.0777472107017552), (8.2455342898166109, -5.
                                    2039371148119731), (-5.
                                    7455342898166091, -0.
                                    873810095889783 42)]

pts2 = vstack((outer_tr, pts))

imp.reload(pjt); dd = pjt.delaunay_triangulation(pts, legalize_plot
                                    =True)

grd_truth = Delaunay(points=pts2, furthest_site=False, incremental=
                                    True)
imp.reload(pjt); my_delaunay = pjt.delaunay_triangulation(pts)

plt.subplot(1,2,1)
plt.triplot(pts2[:,0], pts2[:,1], grd_truth.simplices.copy())
plt.suptitle('Ground truth vs. My triangulation')
my_delaunay.plot(show=True, subplot={'nrows':1, 'ncols':2, 'num':2}
                                    )

plt.close('all')
```

```
# example with random points
import imp
import project2 as pjt
import corner_table as cnt
from scipy.spatial import Delaunay
import matplotlib.pyplot as plt
from numpy.random import uniform
from numpy import array, matrix, vstack, ndarray

imp.reload(cnt);
imp.reload(pjt)

# generate the random points with the outer triangle englobing them
low, high, size = 0, 50, 50
rd_pts = ndarray(buffer=uniform(low=low, high=high, size=2*size),
                                    dtype=float, shape=(size, 2))
outer_pts = pjt.outer_triangle(rd_pts)
rd_pts = vstack((outer_pts, rd_pts))


grd_truth = Delaunay(points=rd_pts, furthest_site=False,
                                    incremental=True)
imp.reload(pjt); my_delaunay = pjt.delaunay_triangulation([tuple(i)
                                     for i in rd_pts[3:]])
my_delaunay._clean_table()

plt.subplot(1,2,1)
plt.triplot(rd_pts[:,0], rd_pts[:,1], grd_truth.simplices.copy())
edges = my_delaunay.plot(show=True, subplot={'nrows':1, 'ncols':2,
                                    'num':2})

plt.close('all')

grd_table = cnt.CornerTable()
a=[grd_table.add_triangle([tuple(i) for i in rd_pts[t]]) for t in
                                    grd_truth.simplices]
my_delaunay.test_delaunay()
grd_table.test_delaunay()

  """
  # initialize the corner table
  cn_table = cnt.CornerTable()
  # compute the outer triangle
  outer_tr = [tuple(i) for i in outer_triangle(pts)]
  # add the outer triangle to the corner table
  cn_table.add_triangle(outer_tr)
  # get a random permutation of the points
  pts_sample = sample(range(len(pts)), size=len(pts), replace=False
                                    , p=None)
# pts_sample = range(len(pts))
  cn_table.plot() if plot else 0
  # iterate over all points
  for p_i in pts_sample:
    cn_table.plot(show=False, subplot={'nrows':1, 'ncols':2, 'num':
                                    1}) if plot else 0
    # p holds the physical position of the i-th point
    p = tuple(pts[p_i])
    # get the triangle containing the point p
    tr_p = cn_table.find_triangle(p)
    v0, v1, v2 = tr_p['vertices']
```

```
# get the triangles sharing edges
tr_share_ed = cn_table.triangles_share_edge(eds=((v0,v1), (v1,
                                v2), (v2,v0)))

# triangles to be added, in the case the point does not lie on
                                some edge
add_faces = [
        [tr_p['physical'][0], p, tr_p['physical'][2]],
        [tr_p['physical'][0], tr_p['physical'][1], p],
        [p, tr_p['physical'][1], tr_p['physical'][2]]
      ]

# check if the point lies on an edge, just see if there is a
                                zero within the baricentric
                                coords
rem_faces = [ tr_p['face'] ]
if tr_p['bari'][0] * tr_p['bari'][1] * tr_p['bari'][2] == 0:
  # determine the triangles to be added, if 3 or 4, and
                                determine
  # which triangles should be removed, if 1 or 2

  # remove the triangle with zero area
  add_faces.pop( 2 if tr_p['bari'][0] == 0 else 0 if tr_p['bari
                                '][1] == 0 else 1 )

  index_bari_zero = 1 if tr_p['bari'][0] == 0 else 2 if tr_p['
                                bari'][1] == 0 else 0

  # result in the opposing vertex of the vertex with
                                baricentric coordinate zero
  opposing_vertex = set(tr_share_ed['physical'][
                                index_bari_zero ][1])
  [opposing_vertex.discard(tuple(v)) for v in tr_p['physical']]
  opposing_vertex = tuple(opposing_vertex.pop())

  # add the 2 new triangles to be added
  [add_faces.append([v, p, opposing_vertex])
      for v in set(tr_share_ed['physical'][ index_bari_zero ][1
                                ]).intersection([tuple(i) for i
                                in tr_p['physical']])]

  # define the faces to remove based on the zero of the
                                baricentric coordinate
  # if the first coordinate if zero, then remove the second
                                triangle on the list tr_share_ed
  # if the second coordinate if zero, then remove the third
                                triangle on the list tr_share_ed
  # if the third coordinate if zero, then remove the first
                                triangle on the list tr_share_ed
  rem_faces.append( tr_share_ed['faces'][ index_bari_zero ][1])


# remove the triangles
[cn_table.remove_triangle(f) for f in rem_faces]

# add the triangles
added_faces = [cn_table.add_triangle(f) for f in add_faces]

# legalize edges
# legalize using the inserted point and the 3/4 triangles added
[cn_table.legalize(point=p, face=f['face'], plot=legalize_plot)
                                for f in added_faces] if
```

```
                                    legalize else 0
    cn_table.plot(show=True, subplot={'nrows':1, 'ncols':2, 'num':2
                                    }) if plot else 0

  # remove outer triangle
  # since the outer triangle is the first one, its vertices are 0,1
                                    ,2
  if remove_outer:
    [cn_table.remove_triangle(t) for t in cn_table.star(vt=[0,1,2])
                                    ['faces']]

  # clean the corner table
  cn_table._clean_table()

  # with the removal of the outer triangle it might lose the convex
                                     hull
  # so it is require to walk over all border vertices drawing edges
                                     between them

  # get the faces that has 1 vertex without opposite vertex, -1
  # these are the faces of the border
  # then I think walking to the right and computing the orientation
                                     of the points of
  # 2 adjacent faces, I can a criterion to know if I should add a
                                    new triangle


  return cn_table
```

A seguir, cada etapa será descrita com mais detalhes.

Na *docstring* de cada método é apresentada uma descrição geral da função e de cada parâmetro.

2.1 Determinação do triângulo externo

Nesta etapa os pontos são centrado, é tomada a distância do ponto mais distante da origem como o raio de um círculo que engloba todos os pontos. Esse raio é definido como a coordenada y do primeiro vértice do triângulo, a coordenada x é definida como 0. Para encontrar os outros dois pontos é feita uma rotação de $\frac{-2\pi}{3} rad$ do ponto inicial duas vezes. Por fim, aos pontos encontrados são levados para a localização inicial dos dados.

A figura 2.1 ilustra esse processo.

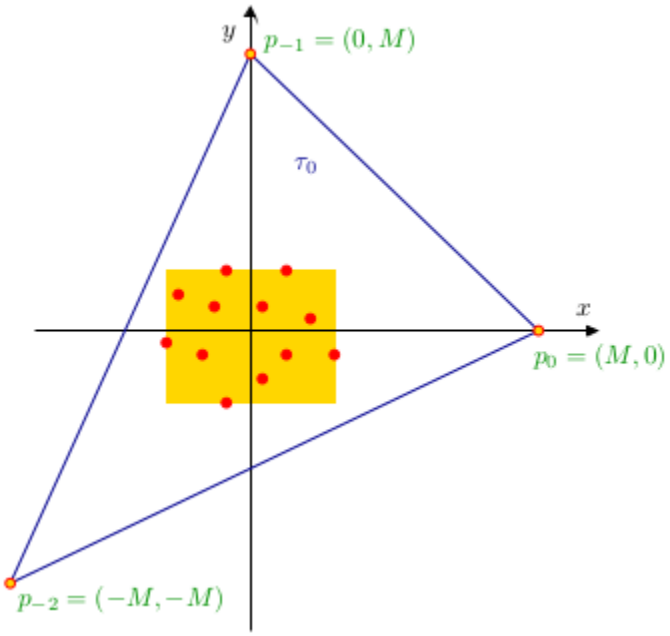


Figura 1: Exemplo de triângulo externo.

A seguir é mostrado a função que calcula os vértices do triângulo externo.

```
def outer_triangle(pts, p0=False):
  """
##########################################
# Determine the points of a outer triangle containing all the
                                    points
##########################################
# pts:  List. Each element is a n-dimensional point
##########################################
# Return a list of points

# TODO generalize for more dimensions
  """
  data = array(pts)
  if p0:
    p0 = data[data[:,1].argmax(), :]
    y_min = data[:,1].min() -1


  # get the center of the data, the mean over each coordinate
  center = data.mean(axis=0)
  data = data - center
  radius = max([sqrt(p.dot(p)) for p in data])

  # to rotate counterclockwise
  theta = -2*pi/3
  rot_mat = array([[cos(theta), -sin(theta)], [sin(theta), cos(
                                    theta)]])

  p0 = [center[0], 3*radius]
  p1 = rot_mat.dot(p0)
  p2 = rot_mat.dot(p1)
```

```
  return (array([p0, p1, p2]) + center)
```

2.2 Encontrar triângulo que contém determinado ponto

Esta função retorna o índice do triângulo que contém o ponto passado. Basicamente é feita uma busca aleatória dos triângulos presentes na estrutura de dados. Através das coordenadas baricêntricas é possível determinar se o ponto pertence ao triângulo em questão.

Posteriormente o Professor passou um algoritmo para a determinação de tal triângulo, mas este algoritmo ainda não foi implementado.

```
  def find_triangle(self, point):
    """
###
# Given a point return the triangle containing the given point
###
# point:  Tuple. The physical coordinate of the point
###
# return the number of the face

# Usage example:

import corner_table as cnt


# triangles from the slides defining the corner table data
                                    structure
tr1 = [(0,1), (1,0), (2,2,),]
tr2 = [(2,2), (1,0), (3,1,),]
tr3 = [(2,2), (3,1), (4,2,),]
tr4 = [(3,1), (5,0), (4,2,),]

imp.reload(cnt)
c_table = cnt.CornerTable()
c_table.add_triangle(tr1)
c_table.add_triangle(tr2)
c_table.add_triangle(tr3)
c_table.add_triangle(tr4)

# should return the first triangle
c_table.find_triangle((1,1))

# should return the second triangle
c_table.find_triangle((2,1))

# should return the third triangle
c_table.find_triangle((3,1.5))

# should return the forth triangle
c_table.find_triangle((4,1))


    """

    # TODO improve the way Castelo said in class

    tr_ret = -1
    # performs the search over all triangles in an random order
```

```
    # TODO it is possible to improve this searching looking for the
                                    big triangles first
    #   maybe change the probability of the sampling by the area of
                                    the triangle
    tr_ids = sample(range(len(self._fc_hash)), size=len(self.
                                _fc_hash), replace=False, p=None)
    tr_tested = []
    while len(tr_ids) > 0:
      tr_i = tr_ids[0]
      tr_ids = tr_ids[1:]
      # if the current triangle was already tested continue
      if tr_i in tr_tested or len(self._fc_hash[tr_i]) == 0:
        continue
      tr_tested.append(tr_i)
      # get the vertices of the given triangle
      c0 = self._fc_hash[tr_i][0]
      # get the physical coordinate of the 3 vertices given their "
                                    virtual" indices
      v0, v1, v2 = self._cn_table[[c0, self._cn_table[c0, 3], self.
                                _cn_table[c0, 4]], 1]
      P_v0 = tuple(self._coord_hash['vir'][v0])
      P_v1 = tuple(self._coord_hash['vir'][v1])
      P_v2 = tuple(self._coord_hash['vir'][v2])
      # perform the incircle test
      # if false continue
      # ATTENTION garantee the order of the vertices are correct
      if not self.inCircle([P_v0, P_v1, P_v2, point]):
        continue
      # get the triangles sharing edges
      # maybe I should get all triangles with that share vertices,
                                    the clousure
      tr_cls = self.closure(fc=[tr_i])
      # for each selected triangle, there are 4 of them, find the
                                    baricentric coordinates
      # of the query point
      for tr in tr_cls['faces']:
        c0 = self._fc_hash[tr][0]
        v0, v1, v2 = self._cn_table[[c0, self._cn_table[c0, 3],
                                    self._cn_table[c0, 4]], 1]
        P_v0, P_v1, P_v2 = array(self._coord_hash['vir'])[ [v0,v1,
                                    v2] ]
        bari_c = self.bari_coord([P_v0, P_v1, P_v2], point)
        # if all baricentric coordinates are positive it mean the
                                    point is inside this triangle
        if bari_c[0] * bari_c[1] * bari_c[2] >= 0:

          return {'face':tr, 'vertices':[v0,v1,v2], 'bari':bari_c,
                                    'physical':[P_v0,P_v1,P_v2]}
```

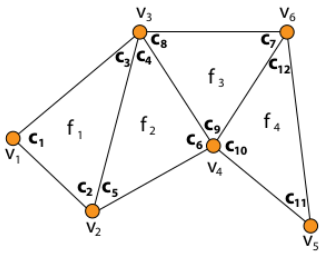
2.3 Encontrar triângulos que compartilhem aresta

Esta função permite fazer a busca pelos índices dos *corners* ou pelas arestas. As arestas são definidas pelos índices dos vértices, sem importar a ordem dos vértices. É retornado o índice dos dois triângulos que compartilhem a aresta passada.

Para essa busca, os vértices são convertidos nos seus respectivos *corners*, então é feita uma simples consulta a estrutura de dados para pegar a primeira face, e a segunda faces é determinada pelo *corner* à direita do primeiro.

A figura 2.3 ilustra um exemplo. É passada a aresta $(V2, V3)$, são deter-

minados os *corners* relacionados à esses vértices, no caso C_2, C_3, C_4, C_5. Como os *corners* são armazenados na estrutura orientados, é preciso apenas do primeiro *corner* relacionado à $V2$, e com o *corner* à direita desse já consegue-se determinar o triângulo adjacente.



corner	c.v	c.t	c.n	c.p	c.o	c.l	c.r
c_1	v_1	f_1	c_2	c_3	c_6	$\emptyset$	$\emptyset$
c_2	v_2	f_1	c_3	c_1	$\emptyset$	$\emptyset$	c_6
c_3	v_3	f_1	c_1	c_2	$\emptyset$	c_6	$\emptyset$
c_4	v_3	f_2	c_5	c_6	$\emptyset$	c_7	c_1
c_5	v_2	f_2	c_6	c_4	c_7	c_1	$\emptyset$
c_6	v_4	f_2	c_4	c_5	c_1	$\emptyset$	c_7
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Figura 2: Exemplo de *Corner table*.

A seguir é exibida o código da função que encontra os triângulos que compartilham aresta.

```
  def triangles_share_edge(self, eds=[], cns=[]):
    """
###
# Get the triangles that share edges
###
# eds:  List. The list of edges
# cns:  List. The list of corners
###
# Return a list of triangles
# It iterate over the vertices of each triangle in the order they
                                    were created
    """
    trs = {'faces':[], 'physical':[], 'virtual':[]}
    corners = cns

    # convert the edges to corners
    if len(eds) > 0:
      corners = []
      for e in eds:
        c0 = set(self._cn_table[self._vt_hash[e[1]], 4]).
                                    intersection( self._vt_hash[e[0]]
                                     )
        # check if the edge exists
        if len(c0) > 0:
          c0 = c0.pop()
          c1 = self._cn_table[c0, 3]
          corners.append( (c0,c1) )

    for c in corners:
      # get the face of c0
      f0 = self._cn_table[c[0], 2]
      # get the face of the right corner of c0, that is equvalent
                                    to the opposite
      # of the previous corner of c0
      f1 = self._cn_table[ self._cn_table[c[0], 7] , 2]
      trs['faces'].append((f0,f1))
```

```
        trs['physical'].append([
                    [tuple(self._coord_hash['vir'][v])
                        for v in self._cn_table[self._fc_hash[f], 1]
                      ]
                    for f in [f0, f1]])
        trs['virtual'].append([ tuple( self._cn_table[self._fc_hash[f
                                        ], 1] )
                    for f in [f0, f1]])
    return trs
```

2.4 Remover triângulo

Função bem simples, que por questões de eficiência apenas marca as linhas dos *corners* do triângulo como -1 sem remover os elementos da *Corner table*. Como, também por questões de eficiência na busca, são mantidas estruturas *hash* auxiliares para relacionar um dado vértice a seus *corners* e uma dada face a seus *corners*, essas estruturas precisam ser atualizadas. Por fim é reatribuído os *corners* à direita e esquerda e *corners* opostos, dos *corners* que faziam referência aos *corners* removidos, basicamente as faces que compartilhavam arestas.

A seguir é exibida o código da função que remove triângulos.

```
  def remove_triangle(self, face):
    """
###
# Remove a triangle from the Corner Table
###
# face: Number. The index of the face
###
# Modify the current corner table

# Usage example:

import corner_table as cnt


# triangles from the slides defining the corner table data
                                        structure
tr1 = [(0,1), (1,0), (2,2,),]
tr2 = [(2,2), (1,0), (3,1,),]
tr3 = [(2,2), (3,1), (4,2,),]
tr4 = [(3,1), (5,0), (4,2,),]

imp.reload(cnt)
c_table = cnt.CornerTable()
c_table.add_triangle(tr1)
c_table.add_triangle(tr2)
c_table.add_triangle(tr3)
c_table.add_triangle(tr4)

c_table.remove_triangle(2)
    """
    # get the corners of the given triangle
    c0, c1, c2 = self._fc_hash[face]
    # get the vertices of these corners
    v0, v1, v2 = self._cn_table[[c0, c1, c2], 1]
```

```
    # to remove this face just need to erase the entries regarding
                                    these corners
    # from the hashs (fc_hash and vt_hash) and from the corner
                                    table
    # meaning to set -1 to the first column of each line, c0, c1,
                                    c2
    self._vt_hash[v0].remove(c0)
    self._vt_hash[v1].remove(c1)
    self._vt_hash[v2].remove(c2)

    self._fc_hash[face] = []


    # fix the surrounding faces
    surrounding_faces = list(set([t[1] for t in self.
                                    triangles_share_edge(cns=((c0,c1)
                                    , (c1,c2), (c2,c0)))['faces']]))
    surrounding_corners = array(self._fc_hash)[surrounding_faces]
    self._cn_table[[c0,c1,c2], 0] = -1
    self.fix_opposite_left_right(self._cn_table, self._vt_hash, ids
                                    =hstack(surrounding_corners))
```

2.5 Adicionar triângulo

Operação simples de adicionar um triângulo especificado por 3 vértices. Primeiro verifica se a ordem dos vértices passados segue a orientação da estrutura, se não inverte a ordem, depois são adicionados os vértices nas estruturas auxiliares, por fim 3 *corners* representando o triângulo são adicionados à *Corner table*. Um último passo é recalcular os *corners* à direita, esquerda e opostos dos *corners* inseridos e dos restantes.

A seguir é exibida o código da função que adiciona triângulos.

```
  def add_triangle(self, vertices):
    """
###
# Add a triangle to the Corner Table
###
# vertices:   List. The list of vertices, the physical coordinate
                                    of each vertice
###
# Return the index of the added triangle and its vertices indices
# Modify the current corner table

# Usage example:

import corner_table as cnt


# triangles from the slides defining the corner table data
                                    structure
tr1 = [(0,1), (1,0), (2,2,),]
tr2 = [(2,2), (1,0), (3,1,),]
tr3 = [(2,2), (3,1), (4,2,),]
tr4 = [(3,1), (5,0), (4,2,),]

imp.reload(cnt)
c_table = cnt.CornerTable()
c_table.add_triangle(tr1)
c_table.add_triangle(tr2)
```

```
c_table.add_triangle(tr3)
c_table.add_triangle(tr4)

    """
    # garantee to have only 3 points
    vts = vertices[:3]
    # check the orientation and reverse the order if it is not
                                counterclockwise
    if not self.orientation(vts):
      vts.reverse()

    # add the z coordinate if it is missing
#   if len(vts[0]) < 3:
#     vts[:, 3] = [1,1,1]

    # first add the vertices to vt_hash

    # get the face index add a new element to fc_hash
    fc_id = len(self._fc_hash)
    v0 = self._add_vertice(vts[0])
    v1 = self._add_vertice(vts[1])
    v2 = self._add_vertice(vts[2])
    c0 = len(self._cn_table)
    c1, c2 = c0+1, c0+2
    self._fc_hash.append( [c0, c0+1, c0+2] )

    # FIRST check if the vertices to be added aren't already in the
                                 structure
    self._vt_hash[v0].append( c0 )
    self._vt_hash[v1].append( c1 )
    self._vt_hash[v2].append( c2 )

    cn_triangle = [
            [c0, v0, fc_id, c1, c2,  -1, -1, -1],
            [c1, v1, fc_id, c2, c0,  -1, -1, -1],
            [c2, v2, fc_id, c0, c1,  -1, -1, -1],
          ]

    # add to the structure and calls fix to garantee consistency
    self._cn_table = vstack([ self._cn_table, cn_triangle ]) if len
                                (self._cn_table) != 0 else array(
                                cn_triangle)


    # INSERT THE TRIANGLE TO THE CORNER TABLE
    # add the vertices, and the face, then calls
                                fix_opposite_left_right
    # passing a subset of the corner table
    # only the corners of the star of the added triangle
    #
    # Maybe the star has more elements then the needed, but it is a
                                 easy start since it is
    # already implemented, but in the future return only the
                                triangle that share edges
    # with the added one

    # just get the faces of the opposite corners to the one being
                                added
    # then get the two next corners and you have one adjacent face
    c_ids = [self._vt_hash[v0], self._vt_hash[v1], self._vt_hash[v2
                                ]]
    fix_ids = set(hstack([
```

```
                hstack(c_ids),
                list(set(hstack([self._fc_hash[fc] for ci in c_ids
                                 for fc in self._cn_table[ci, 2]])
                                 )),
            ]))
    self.fix_opposite_left_right(self._cn_table, self._vt_hash,
                                 fix_ids)

#   return {'face': len(self._fc_hash), 'vertices': (v0,v1,v2)}
    return {'face': fc_id, 'vertices': (v0,v1,v2)}
```

2.6 Legalização de triângulos

Está é uma função importante que garante que quando um ponto é inserido na triangulação já existente, os triângulos modificados continuem sendo *Delaunay*.

É passado para a função um ponto, o ponto que foi inserido, e a face desse ponto. É verificado então se o vértice oposto ao ponto inserido, com relação à aresta oposta, respeita o teste do incírculo, caso o vértice oposto esteja dentro do circuncirculo do triângulo adicionado, é realizado o *flip* da aresta. Esse *flip* resulta em dois outros triângulos que novamente precisam ser verificados em relação ao mesmo ponto e suas arestas opostas.

A figura 2.6 ilustra um exemplo de *flip* de aresta realizado durante a legalização.

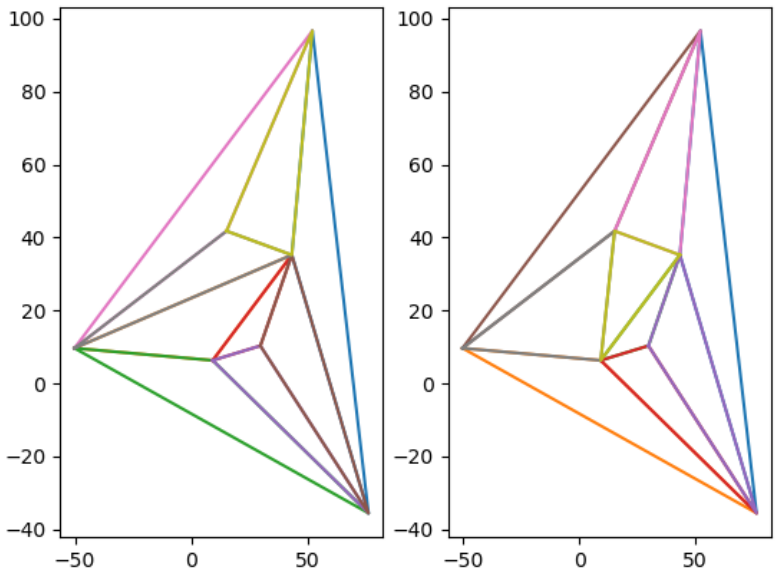


Figura 3: Exemplo de *flip* de aresta.

A seguir é exibida o código da função que legaliza arestas.

```
  def legalize(self, point, face, plot=False):
    """
###
# Verify if the edge need to be legalized and do
###
# point:  Tuple. The physical coordinates of the added point
# face:   Integer. The number of the added triangle
###
# Modify the corner table

# Usage example:


import corner_table as cnt


# triangles from the slides defining the corner table data
                                    structure
tr1 = [(0,1), (1,0), (2,2,),]
tr2 = [(2,2), (1,0), (3,1,),]
tr3 = [(2,2), (3,1), (4,2,),]
tr4 = [(3,1), (5,0), (4,2,),]
tr5 = [(1,0), (5,0), (3,1,),]

imp.reload(cnt)
c_table = cnt.CornerTable()
c_table.add_triangle(tr1)
c_table.add_triangle(tr2)
c_table.add_triangle(tr3)
c_table.add_triangle(tr4)
c_table.add_triangle(tr5)

#c_table.plot(show=False, subplot={'nrows':1, 'ncols':2, 'num':1})
c_table.legalize(point=(0,1), face=0, plot=True)
#c_table.plot(show=True, subplot={'nrows':1, 'ncols':2, 'num':2})
    """
    # get the index of the added vertex
    vi = self._coord_hash['phys'][point]

    # get the vertices of the given face
    corners = self._fc_hash[face]
    v0, v1, v2 = self._cn_table[corners, 1]


    # maybe dont need this
    # get the corner of point
    if not(vi == v0 or vi == v1 or vi == v2):
      return True
    ci = corners[ [v0,v1,v2].index(vi) ]

    # reassign v0,v1,v2 to garantee that the order is correct, i.e
                                    ., they are oriented
    v0 = self._cn_table[ci, 1]
    v1 = self._cn_table[self._cn_table[ci, 3], 1]
    v2 = self._cn_table[self._cn_table[ci, 4], 1]

    # get the opposite vertex of ci
    ci_opp = self._cn_table[ci, 5]
    # if there is no opposite vertex the bondaury was reached
    if ci_opp == -1:
      return True
    # get the vertex index for the oppositve corner of ci
```

```
    opposite_vertex = self._cn_table[ci_opp, 1]

    P_v0, P_v1, P_v2 = array(self._coord_hash['vir'])[[v0,v1,v2]]
    P_v_opp = self._coord_hash['vir'][opposite_vertex]
#   return True
    # check if the 4 points are in a legal arrengement
    oriented_pts = array([P_v2, P_v0, P_v1, P_v_opp ])
    oriented_pts = [P_v2, P_v0, P_v1, P_v_opp ]


    if self.inCircle(oriented_pts):
      # perform the edge flip
      faces = (self._cn_table[ci, 2], self._cn_table[ci_opp, 2])
      # before perform the slip get the MAYBE DONT NEED
      self.plot(show=False, subplot={'nrows':1, 'ncols':2, 'num':1}
                                    ) if plot else 0
      flipped_fcs = self.flip_triangles(faces[0], faces[1])

      self.plot(show=True, subplot={'nrows':1, 'ncols':2, 'num':2})
                                     if plot else 0

      # call legalize for the 2 other edges
      self.legalize(point, flipped_fcs[0]['face'])
      self.legalize(point, flipped_fcs[1]['face'])
```

A operação de *flip* é bem simples, dados dois triângulos que compartilhem uma aresta, esta aresta que eles compartilham é trocada pela aresta dos vértices opostos. Na implementação isso é feito removendo-se os dois triângulos e adicionando dois outros.

A seguir é exibida o código da função que faz *flip* de arestas.

```
  def flip_triangles(self, face0, face1):
    """
###
# Perform the flip of two adjacent triangles
###
# face0:  Number. The index of the face
# face1:  Number. The index of the face
###
# Modify the current corner table
# This method removes the edges between the two triangles and
# add a new one between the opposing corners, performing the edge
                                    flip

# Usage example:

import corner_table as cnt


# triangles from the slides defining the corner table data
                                    structure
tr1 = [(0,1), (1,0), (2,2,),]
tr2 = [(2,2), (1,0), (3,1,),]
tr3 = [(2,2), (3,1), (4,2,),]
tr4 = [(3,1), (5,0), (4,2,),]
tr5 = [(1,0), (5,0), (3,1,),]

imp.reload(cnt)
c_table = cnt.CornerTable()
c_table.add_triangle(tr1)
c_table.add_triangle(tr2)
c_table.add_triangle(tr3)
```

```
c_table.add_triangle(tr4)
c_table.add_triangle(tr5)

c_table.plot(show=False, subplot={'nrows':1, 'ncols':2, 'num':1})
c_table.flip_triangles(1, 2)
c_table.plot(show=True, subplot={'nrows':1, 'ncols':2, 'num':2})
    """

    # TODO check if it is possible to split the triangles

    # TODO having problems when remove a triangle and a vertex is
                                left hanging,
    # this happens for triangles on the boundary


    # the easiest way to implement this is to remove both faces and
                                add the new ones

    # first get the vertices
    v0, v1, v2 = self._cn_table[self._fc_hash[face0], 1]
    v3, v4, v5 = self._cn_table[self._fc_hash[face1], 1]
    # get the vertices repeted between face0 and face1
    v_rep_01 = list(set((v0,v1,v2)).intersection((v3,v4,v5)))

    # get the opposing vertices for faces 0 and 1
    v_opp_0 = set((v3,v4,v5))
    v_opp_1 = set((v0,v1,v2))
    [v_opp_0.discard(i) for i in (v0,v1,v2)]
    [v_opp_1.discard(i) for i in (v3,v4,v5)]
    v_opp_0 = v_opp_0.pop()
    v_opp_1 = v_opp_1.pop()


    # build the new triangles
    # I have to specify the physical position of the vertices, not
                                the edges
    # as I was trying before
    tr0 = [self._coord_hash['vir'][v_opp_1], self._coord_hash['vir'
                                ][v_opp_0], self._coord_hash['vir
                                '][v_rep_01[0]]]
    tr1 = [self._coord_hash['vir'][v_opp_0], self._coord_hash['vir'
                                ][v_opp_1], self._coord_hash['vir
                                '][v_rep_01[1]]]


    # remove face0 and face1 and add tr0 and tr1
    self.remove_triangle(face0)
    self.remove_triangle(face1)
    face0 = self.add_triangle(tr0)
    face1 = self.add_triangle(tr1)

    return (face0, face1)
```

2.7 Limpeza da estrutura

Esta função é chamada após a triangulação ser concluída para remover os *corners* excluídos da estrutura. Basicamente recalcula os índices dos *corners*, vértices e faces, e agora sim removendo as entradas excluídas.

A seguir é exibida o código da função que limpa a estrutura.

```
  def _clean_table(self, corners=0, face=0):
    """
###
# Clean the remove corners, triangles and vertices
###
#
###
# Modify the corner table inplace
    """
    # before exclude the row from the corner table
    # save the opposite corner
    # and get the corners of the faces sharing edges
    # make a list with these corners and call
                                    fix_opposite_left_right

    # THIS WAS ALREADY DONE, otherwise the table would be
                                    inconsistent

    # create a hash that store the amount that should be subtracted
                                     from
    # the corner on the i-th position
    rem_corners = self._cn_table[:,0] == -1
    subtract_c = rem_corners.cumsum()
    corners_values = [i - subtract_c[i] for i in range(len(self.
                                    _cn_table))]
    # set the last value as -1, so every time a corner is -1 (
                                    meaning the corner was not
                                    defined)
    # it returns -1
    corners_values.append(-1)
    valid_corners = rem_corners == False

    # build the same hash as for the corners but to subtract the
                                    face's index
    rem_faces = array([len(f) for f in self._fc_hash]) == 0
    subtract_f = rem_faces.cumsum()
    new_faces = []

    # new vertex hash
    new_vertices = [[] for i in range(len(self._vt_hash))]

    # now iterate over the corner table subtracting from the
                                    corners
    for ci in self._cn_table:
      if ci[0] == -1:
        continue
      ci[0], ci[3] = corners_values[ ci[0] ], corners_values[ ci[3]
                                     ]
      ci[4], ci[5] = corners_values[ ci[4] ], corners_values[ ci[5]
                                     ]
      ci[6], ci[7] = corners_values[ ci[6] ], corners_values[ ci[7]
                                     ]
      ci[2] = ci[2] - subtract_f[ci[2]]

      # update the _fc_hash values
      if ci[2] >= len(new_faces):
        new_faces.append([])
      new_faces[ci[2]].append(ci[0])

      # update the _vt_hash values
      new_vertices[ci[1]].append(ci[0])
```

```
    # keep only the valid corners
    self._cn_table = self._cn_table[valid_corners]


    self._vt_hash = new_vertices

    # clean the faces indices
    self._fc_hash = new_faces
```

2.8 Outras funções

Os métodos descritos anteriormente são os principais para a triangulação, no entanto, a fim de validar a implementação outras funções foram implementadas.

- ***plot*:** Esta função faz o *plot* da triangulação obtida;
- ***test_ delaunay*:** Faz a contagem do número de faces, de faces orientadas e de faces *delaunay*;
- ***bari_ coord*:** Calcula as coordenada baricêntricas de um ponto;
- ***inCircle*:** Realiza o teste do incirculo;
- ***orientation*:** Verifica se a orientação de 3 pontos está consistente;
- ***fix_ opposite_ left_ right*:** Calcula os *corners* opostos, à direita e esquerda, mantendo a estrutura consistente.

3 Resultados

Nesta seção são apresentados os resultados da implementação.

Alterando os parâmetros *plot*, *legalize* e *legalize_ plot* da função que realiza a triangulação, as figuras 3 e 3 foram obtidas. Esses parâmetros permitem fazer *plots* da triangulação passo a passo.

A figura 3 Mostra a inserção de pontos, no triângulo externo, sem o processo de legalização das arestas. Com isso é possível ver como a simples adição de pontos pode resultar em triângulos alongados, de qualidade ruim.

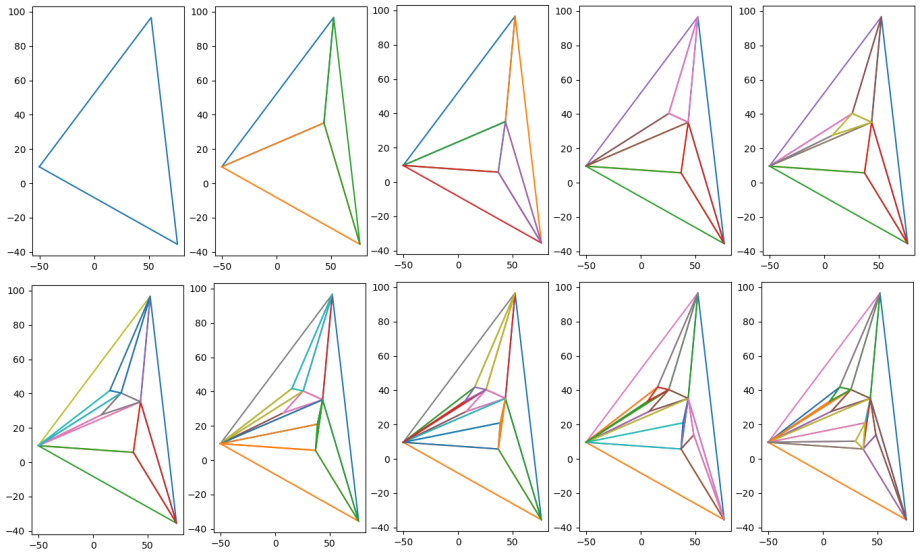


Figura 4: Passo a passo da inserção de pontos **sem** legalização de arestas.

Já na figura 3 é exibida a inserção dos mesmos pontos, embora que em ordem diferente já que inserção dos pontos é aleatória, mas agora com a legalização das arestas. Isso garante que a cada inserção a triangulação continue sendo *Delaunay*.

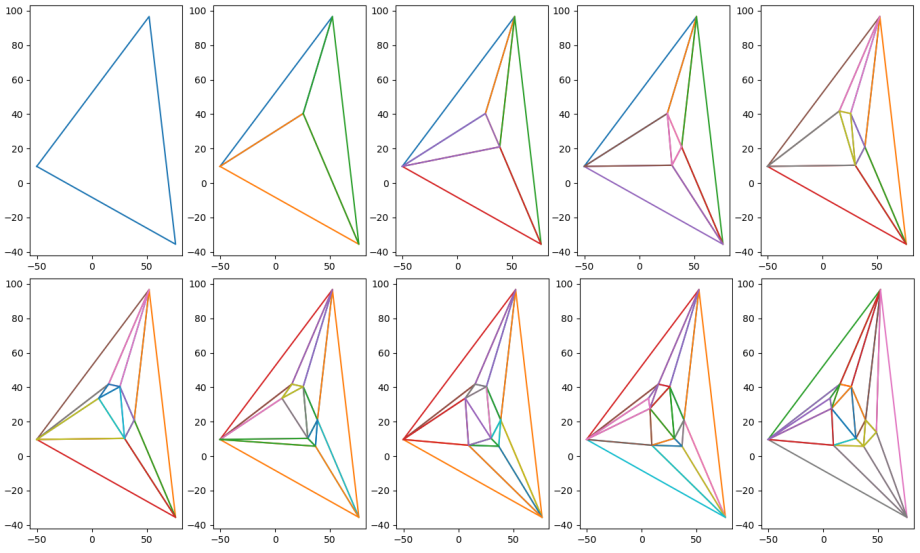


Figura 5: Passo a passo da inserção de pontos **com** legalização de arestas.

A seguir é exibido um exemplo de código para execução de uma triangulação *Delaunay* de 50 pontos gerados aleatoriamente segundo uma distribuição

uniforme de 0 a 50. A figura 3 mostra o resultado final da triangulação, à direita é exibida a triangulação obtida pela bibliteca *SciPy*, em azul, e à direita a triangulação obtida pela implementação do projeto. Neste exemplo, o triângulo externo ainda não foi removido. A figura 3 mostra a triangulação com a remoção do triângulo externo.

```
# example with random points
import imp
import project2 as pjt
import corner_table as cnt
from scipy.spatial import Delaunay
import matplotlib.pyplot as plt
from numpy.random import uniform
from numpy import array, matrix, vstack, ndarray


# generate the random points with the outer triangle englobing them
low, high, size = 0, 50, 10
rd_pts = ndarray(buffer=uniform(low=low, high=high, size=2*size),
dtype=float, shape=(size, 2))
outer_pts = pjt.outer_triangle(rd_pts)
rd_pts = vstack((outer_pts, rd_pts))


grd_truth = Delaunay(points=rd_pts, furthest_site=False, incremental=True)
my_delaunay = pjt.delaunay_triangulation([tuple(i) for i in rd_pts[3:]])

plt.subplot(1,2,1)
plt.triplot(rd_pts[:,0], rd_pts[:,1], grd_truth.simplices.copy())
edges = my_delaunay.plot(show=True, subplot={'nrows':1, 'ncols':2, 'num':2})

plt.close('all')

grd_table = cnt.CornerTable()
a = [grd_table.add_triangle([tuple(i) for i in rd_pts[t]])
for t in grd_truth.simplices]
my_delaunay.test_delaunay()
grd_table.test_delaunay()
```

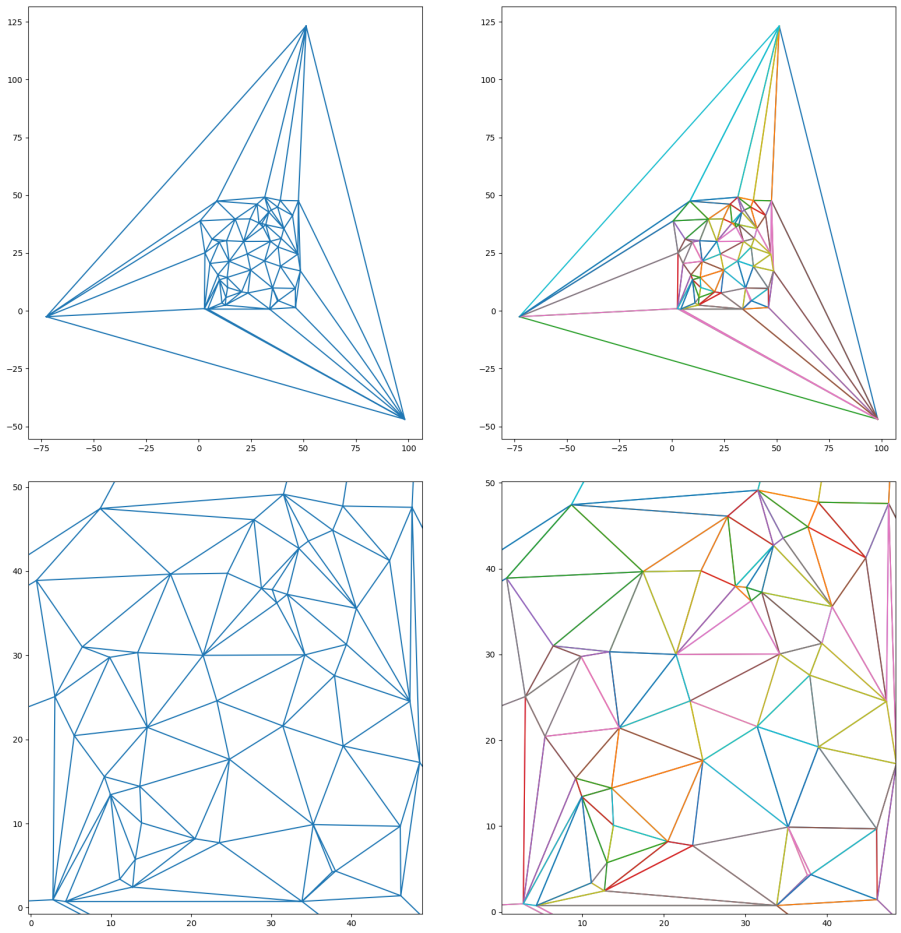


Figura 6: Exemplo de triangulação com 50 pontos aleatórios, ainda com o triângulo externo. Triangulação à esquerda gerada pela biblioteca *SciPy*, à direita gerada pela implementação do projeto. Na parte de baixo é exibida a versão ampliada na triangulação.Triangulação à esquerda gerada pela biblioteca *SciPy*, à direita gerada pela implementação do projeto.

Um detalhe importante no exemplo da figura 3 é que nas referência do algoritmo de inserção de pontos incremental, não há referência sobre a inserção do fecho convexo ao final da triangulação. E o exemplo da figura deixa claro que em alguns casos, quando os vértices, e por consequência os triângulos, do triângulo externo são removidos, a triangulação resultante pode não ser convexa. Apesar disso a biblioteca *SciPy* corretamente resulta numa triangulação que é convexa. E pelo exemplo anterior, figura 3, fica claro que se ambas as triangulações deixarem o triângulo externo, o resultado é o mesmo, então a diferença está no passo após a remoção do triângulo externo.

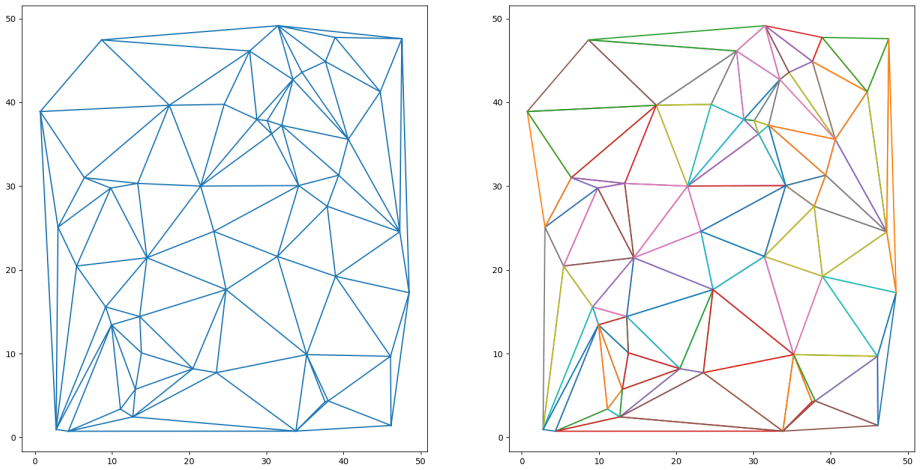


Figura 7: Exemplo de triangulação final com 50 pontos aleatórios. Detalhe para a diferença entre as triangulações, a da direita convexa e a da esquerda não convexa. Triangulação à esquerda gerada pela biblioteca *SciPy*, à direita gerada pela implementação do projeto.