Relatório do Projeto 2

Daniel Moreira Cestari - 5746193

30 de novembro 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 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)
```

Próximas funções

2.2 Particionamento do domínio e determinação dos bordos

A função *partitionate_ domain* chama as funções que realizam o particionamento e determinação dos bordos, *heuristic_ 1* e *heuristic_ 2*. Para a reutilização do código que resolve a equação de *Laplace* cada partição é salva em arquivo, e essa função que salva cada partição em arquivo. Abaixo é mostrada a função.

A diferença entre as duas heurísticas está na divisão feita sobre as partições que contém o círculo.

A primeira heurística (função *heuristic_ 1*), na partição do círculo, define o bordo de cima como a parte de cima do domínio mais a reta vertical até chegar ao círculo, e o mesmo princípio para o bordo de baixo, seguindo o sentido da esquerda para a direita. Os bordos da esquerda e direita são definidos pela reta vertical e pela curva, dependendo se a partição está a direita ou a esquerda da curva, e seguindo o sentido de baixo para cima. As figuras **??** e **??** mostram as malhas geradas pela heurística 1 nas partições do círculo. As cores definem os bordos, azul bordo de cima, laranja bordo de baixo, verde bordo da esquerda, e vermelho bordo da direita.

Após a apresentação do trabalho foi incorporado o refinamento dos bordos na heurística 1, e foram removidas as singularidades, quadriláteros com um dos lados de comprimento zero.

A heurística 2 difere da 1, na sua definição dos bordos sobre a partição da curva. Neste caso, a divisão é feita seguindo o lado, se lado em questão está a esquerda então é o bordo esquerdo, se está a direita é o bordo direito, se está acima é o de cima e se está abaixo o de baixo. Esta heurística não apresenta refinamento nos bordos. As figuras **??** e **??** mostram malhas geradas utilizando a heurística 2, as cores têm o mesmo significado que o relatado na heurística 1.

2.3 Geração da malha

A função que gera a malha chama as funções que criam o domínio e o particionam, e depois resolve a equação de *Laplace* em cada partição individualmente. Após a malha de cada partição ser gerada, elas são unidas em duas malhas e

então o refinamento é realizado resolvendo a equação de *Poisson* nas duas malhas finais. Mesmo que um refinamento utilizando as funções de controle não seja realizado, como a equação de *Laplace* é utilizada novamente com as malhas unidas, é feita uma suavização nos pontos de união das malhas.

Essa é a principal função que chama todas as outras necessárias, e por isso é cheia de parâmetros. Todos descritos na *docstring*, basicamente juntou todos os parâmetros das funções anteriores mais os parâmetros relativos ao refinamento da malha.

Abaixo é apresentada a função *generate_grid*.

O arquivo *VTK* final é o resultado da união das duas malhas finais refinadas. Foi preciso modificar o código original que gerava o *VTK* para receber uma lista de malhas e quando salvar em disco, produzir apenas um arquivo. Foi tomado o cuidado para vértices posicionados na mesma posição não aparecerem repetidos, ou seja, as malhas são realmente unidas.

3 Resultados

Nesta seção serão apresentados os resultados obtidos com a implementação descrita anteriormente. Primeiro serão mostradas as malhas para a heurística 1 variando o número de pontos na malha, em seguida será feito o mesmo para a heurística 2. A configuração do domínio e da curva será a mesma para ambas as heurísticas.

Após a apresentação do trabalho foram apontados algumas correções, agora incorporadas, mas devido às modificações o exemplo utilizando o aerofólio ficou ruim e foi removido deste relatório.

Todos os arquivos VTK gerados foram anexados.

3.1 Heurística 1

Código para gerar uma malha com 52 pontos nos bordos esquerdo e direito. Curva gerada com 52 pontos, centrada na origem, domínio com comprimento 10, altura 8, e comprimento 3 à direita da curva. O domínio é divido em 4 partes, e o refinamento é deixado apenas nos bordos e a equação de *Laplace* propaga esse refinamento para o interior do domínio.

```
import imp
import numpy as np
from matplotlib import pyplot as plt
import project1 as pjt

imp.reload(pjt);
grid = pjt.generate_grid(resolution=52, left_border=3, domain_length=10,
domain_height=4, curve_params={"radius":1}, equation=pjt.circle,
filename_curve="", heuristic=pjt.heuristic_1, k=3, filename_borders="circle_h1_50pts", plo
```

Abaixo o código agora utilizando 100 pontos.

```
import imp
import numpy as np
from matplotlib import pyplot as plt
```

```
import project1 as pjt

imp.reload(pjt);
grid = pjt.generate_grid(resolution=100, left_border=3, domain_length=10,
domain_height=4, curve_params={"radius":1}, equation=pjt.circle,
filename_curve="", heuristic=pjt.heuristic_1, k=3, filename_borders="circle_h1_100pts", pl
```

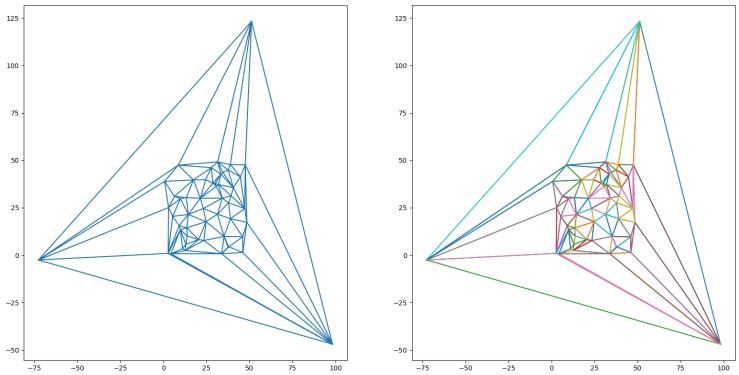

Figura 1:

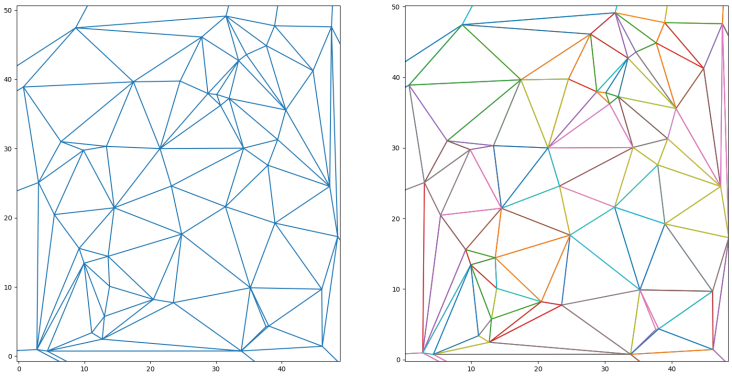

Figura 2:

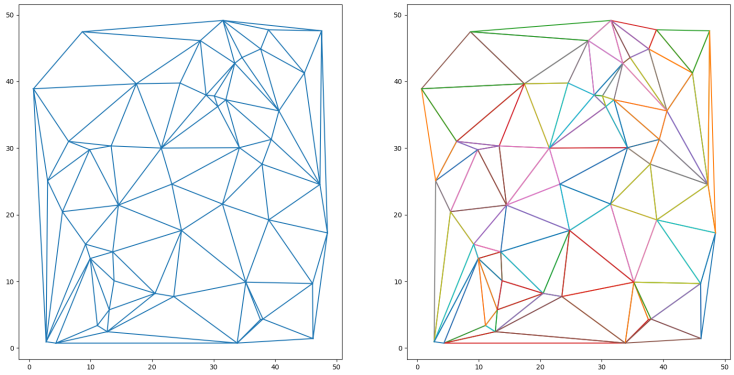

Figura 3: