Relatório do Projeto 1

Daniel Moreira Cestari - 5746193

28 de setembro de 2017

1 Introdução

O objetivo do Projeto 1 é o desenvolvimento de uma malha passados alguns parâmetros aplicando os conceitos visto até o momento.

A especificação do projeto pede para gerar uma malha dados, uma distância D entre um círculo de raio R e a borda esquerda do domínio retangular, o domínio tem comprimento L e deve ser dividido em k partições. Sendo que, uma dessas partições necessariamente precisa dividir o círculo em dois. O círculo deve estar centrado no domínio em termos da coordenada vertical. Também é pedido para refinar a malha ao redor do círculo e no centro do domínio à direita do círculo.

A maneira de como o círculo é gerado, como as partições e o refinamento são feitos não estão definidos, sendo livre a escolha.

2 Implementação

Nesta seção é apresentada a implementação do código, e o porque de determinadas escolhas.

Basicamente o código está dividido em 3 partes:

- **Geração da curva e domínio**: Nesta etapa é gerada a curva e os limites do domínio baseado nos parâmetros passados. Também é possível ler uma curva de um arquivo ou passar uma função que gere outra curva, tornando a implementação mais genérica.

- **Particionamento do domínio e determinação dos bordos**: Basicamente o particionamento do domínio tem apenas duas restrições, dividir a curva no meio e realizar um número $k + 1$ de partições (k é o parâmetro passado que determina quantos pontos de quebra o eixo x tem). Foram implementadas duas maneiras de determinação dos bordos, chamadas de heurística 1 e 2.

- **Geração da malha**: Esta é a única etapa que utiliza código já implementado dos exercícios práticos. Após as várias partições serem geradas, cada uma têm sua malha gerada resolvendo a equação de *Poisson* e ao final são unidas em uma única malha.

A seguir, cada etapa será descrita com mais detalhes.

2.1 Geração da curva e domínio

A geração da curva, no caso um círculo, é realizada por uma função *circle* que recebe 3 parâmetros. Abaixo é exibido o trecho do código com a função.

```
def circle(resolution, center, radius):
  """
####
# Generate a circle following counter-clockwise orientation from
                                  the rightmost point
###
# resolution: Integer. The number of points in the circle
# center:    Tuple. The center of the circle (x,y)
# radius     Number. The radius of the circle
###
# Return a tuple with two arrays (x,y)
  """
  t = np.linspace(start=0, stop=2*np.pi, num=resolution)
  return (radius*np.cos(t) + center[0], radius*np.sin(t) + center[1
                                  ])
```

A convenção adotada na geração da curva é começar do ponto mais a direita em x e seguir o sentido anti-horário. Uma alternativa à geração do círculo, é a leitura de uma curva em arquivo, como o arquivo *naca012.txt* já utilizado em um exercício prático, ou passar uma função que gere outra curva.

```
def generate_curve(resolution=100, left_border=1, domain_length=5,
                                  domain_height=4,
          curve_params={'radius':1}, equation=circle, filename=''):
  """
####
# Generate the profile of a give curve
###
# resolution:   Integer. The number of points in the curve
# left_border:    Number. The rightmost x point
# domain_length:  Number. The total length of the domain
# domain_height:  Number. The total height of the domain
# curve_params:   Dictionary. Other parameters used for generating
                                  the curve
# equation:     Function. The function that apply the curve
                                  function
# filename:     String. The filename from which the curve will be
                                  read from
###
# Returns a tuple with x_min, x_max, y_min, y_max, curve_points

# Usage example:
imp.reload(pjt)
vv = pjt.generate_curve(100, 2, 10, 6, {'radius':1}, pjt.circle,
                                  '')
plt.plot(vv['curve'][0], vv['curve'][1], )
plt.plot([vv['center'][0]], [vv['center'][1]], '*')
plt.xlim((vv['x_min'], vv['x_max']))
plt.ylim((vv['y_min'], vv['y_max']))
plt.show()
plt.close('all')

  """

  # for simplicity I'm going to let the curve in the origin (0,0)
                                  and adjust the domain based on
                                  that
```

```
  # TODO check for the right measures, radius smaller than the
                                height, etc....


  if filename:
    f = open(filename, 'rt')
    curve = [i.split(' ') for i in f.read().splitlines()]
    curve = [(float(i[0]), float(i[-1])) for i in curve]
    curve = np.array(([i[0] for i in curve], [i[1] for i in curve])
                                )
  else:
    curve = equation(resolution, (0,0), **curve_params)
  center = ( np.average(curve[0]), np.average(curve[1]))
  cv_x_min, cv_x_max = min(curve[0]), max(curve[0])
  cv_y_min, cv_y_max = min(curve[1]), max(curve[1])


  return {  'x_min':cv_x_min - left_border, 'x_max':(cv_x_min -
                                left_border) + domain_length,
        'y_min':-domain_height/2 , 'y_max':domain_height/2,
        'x_min_cv':min(curve[0]), 'x_max_cv':max(curve[0]),
        'y_min_cv':min(curve[1]), 'y_max_cv':max(curve[1]),
        'center':center, 'curve':curve
      }
```

Na *docstring* é apresentada uma descrição geral da função e de cada parâmetro. A função *generate_ curve* retorna um dicionário com os pontos que definem o domínio e a curva.

2.2 Particionamento do domínio e determinação dos bordos

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

```
def partitionate_domain(domain, k, heuristic, filename=''):
  """
####
#
###
#
###
#
  """
  # depending on the space between the leftmost point of the curve
                                to the left border
  # generate the first partition on the middle of the curve,
                                without a "blank" partition
  # before the curve


  # ATTENTION the partition should have the same points on the
                                interface, the connection,
                                between them

  # the threshold is a minimum distance in the front of the curve
```

```
  # it is hard coded to be the radius
  # but, since I dont have the radius it will be computed by the
                                 difference of the center
  # and the x_min_cv
  threshold = abs(domain['x_min_cv'] - domain['center'][0])
  borders = heuristic(domain, k, threshold=threshold)[1]

  # save to file the borders
  if filename:
    for i, part in enumerate(borders):
      f = open('%s_part_%d.txt'%(filename, i), 'wt')
      for bd in part:
        f.write('%d\n'%(len(bd[0])))
        print(np.array(bd).shape)
        [f.write('%.2f %.2f\n'%(bd[0][j], bd[1][j])) for j in range
                                 (len(bd[0]))]
      f.close()


  return borders
```

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 2.2 e 2.2 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.

```
def heuristic_1(domain, k, threshold):
  """
####
#
##
#
#
##
#

  """

  # the resolution of the grid/border is given by the number of
                                 points in the
  # curve
  #resolution:    Integer. The resolution, in number of points, of
                                 each partition. Should be
  #          half the length of the curve
  resolution = len(domain['curve'][0])/2

  # the partitions are equally spaced in X
  # the first two partitions define how much is left

  # if it is not possible to have all partitions equally spaced,
  # at least make the partition after the curve with the same
                                 principle
  # and then for the rest adjust its size to fit all k partitions
```

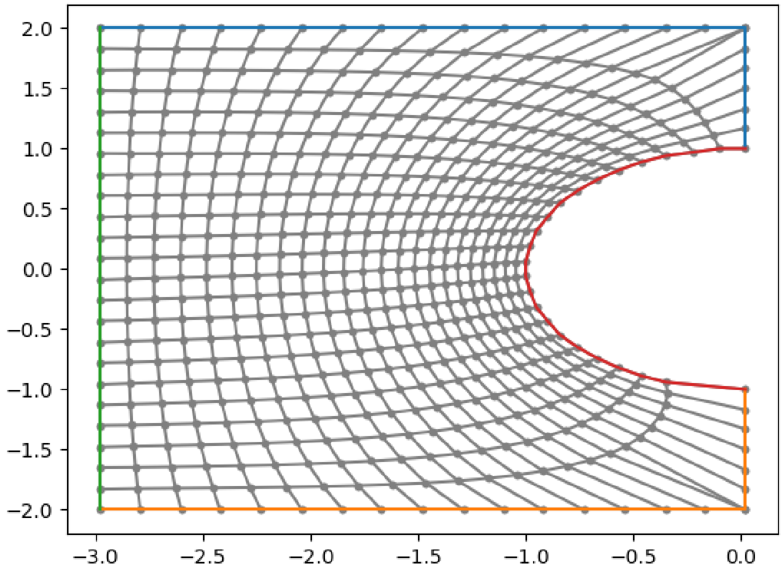

Figura 1: Malha gerada pela heurística 1 na primeira metade do círculo

```
# ATTENTION make the right border of the previous partition equal
                                the left border
# of the next partition

# start by the division of the curve into two, then continue the
                                partition

borders = [[]] * (k+1)
borders = []
# add the leftmost point for the iteration to work later
x_divs = [domain['x_min']]

# the 3 first divisions are different, because the first one I
                                have to check the distance
# between the first division to the curve if it respects the
                                threshold
# then the second is fixed in the center of the curve
# and the third I choose to be symmetric to the second one
                                respect to the curve

# if there is 2*threshold between the leftmost point in the curve
                                and the left border
# then the first partition occurs on domain['x_min'] + threshold
if abs(domain['x_min'] - domain['x_min_cv']) >= 2* threshold :
  x_divs.append( domain['x_min'] + threshold  )
# divide the curve in two
x_divs.append( domain['center'][0] )
```

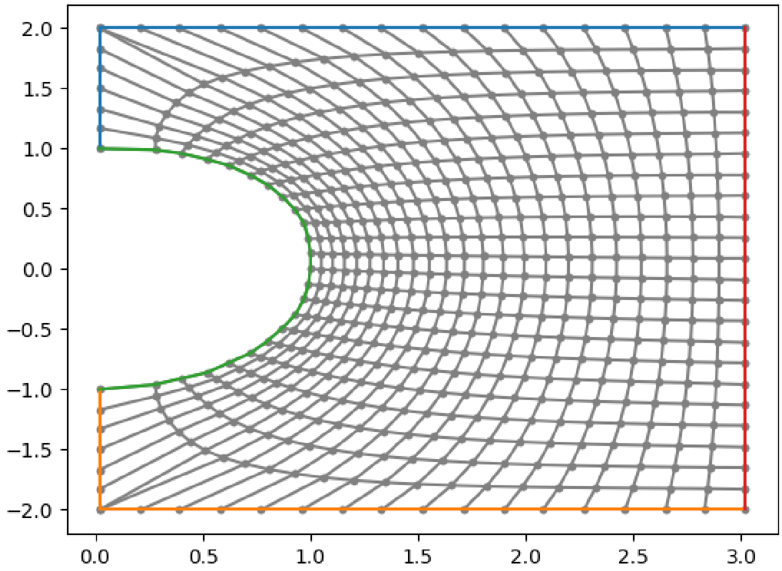

Figura 2: Malha gerada pela heurística 1 na segunda metade do círculo

```
# the next division is symmetrical to the previous one
x_divs.append( x_divs[-1] + abs(x_divs[-1] - x_divs[-2]) )

# the remaining of the domain is equally divided
x_divs.append( np.linspace(start=x_divs[-1], stop=domain['x_max']
                                , num=k - len(x_divs) +3)[1:] )
x_divs = np.hstack(x_divs)

# the convention of the borders on the curve, first half, is
# the left border of the partition is the normal one and the
                                right
# curve itself
# the top is the top and the vertical part until it reaches the
                                curve
# and the bottom is the bottom and the vertical until it reaches
                                the curve

# the second half of the curve is the same, just changing the
                                left for the right
# and the rest of the partition as the square division, top is
                                top, bottom is bottom,
# left is left and right is right

# TODO have not done it yet
# ATTENTION to the corners, remember to start some in the "second
                                " point

# to avoid roundoff errors of the index, reassign resolution
resolution = int(resolution/2)*2
```

```
  # the grid generation has to follow the orientation left to right
                                 for the top and bottom
  # borders,
  # and bottom to top for the left and right borders
  # this is because of the code used to generate the grid, if this
                                orientation is not
  # followed the generated grid becames twisted

  # the order of the borders is top, bottom, left, right
  for i, xi in enumerate(x_divs[:-1]):
    # CHECK if we are dealing with the curve partition
    # Think I need to deal with each side separatedly

    direct_ids = list(range(resolution))
    reverse_ids = list(range(resolution))
    reverse_ids.reverse()

    if (x_divs[i+1] == domain['center'][0]):
      # the first half of the curve

      # compute the number of points in the horizontal and vertical
                                 paths
      n_pts_horizontal = int((abs(x_divs[i+1] - xi)) / (abs(x_divs[
                                i+1] - xi) + abs(domain['y_max']
                                - domain['y_max_cv'])) *
                                resolution)
      n_pts_vertical = resolution - n_pts_horizontal

      top = [ np.hstack((np.linspace(xi, x_divs[i+1],
                                n_pts_horizontal), [x_divs[i+1]]*
                                n_pts_vertical)),
          np.hstack(([domain['y_max']]*n_pts_horizontal, np.
                                linspace(domain['y_max'], domain[
                                'y_max_cv'], n_pts_vertical))) ]
#     top[0], top[1] = top[0][reverse_ids], top[1][reverse_ids]

      bottom = (np.hstack((np.linspace(xi, x_divs[i+1],
                                n_pts_horizontal), [x_divs[i+1]]*
                                n_pts_vertical)),
            np.hstack(([domain['y_min']]*n_pts_horizontal, np.
                                linspace(domain['y_min'], domain[
                                'y_min_cv'], n_pts_vertical))) )

      # since the curve starts at the rightmost point, I can start
                                here with the
      # the point located at 1/4 of the curve length and go up
                                until 3/4
      # JUST EXCHANGED LEFT FOR RIGHT
      right = [ np.hstack((domain['center'][0], domain['curve'][0][
                                int(resolution/2)+1:int(
                                resolution*3/2)-1], domain['
                                center'][0])),
              np.hstack((domain['y_max_cv'], domain['curve'][1][int
                                (resolution/2)+1:int(resolution*3
                                /2)-1], domain['y_min_cv']))]
      right[0], right[1] = right[0][reverse_ids], right[1][
                                reverse_ids]
      left =  [np.array([xi]*resolution), np.linspace(domain['y_max
                                '], domain['y_min'], resolution)]
      left[0], left[1] = left[0][reverse_ids], left[1][reverse_ids]
```

```
        print(('len(left[0])', len(left[0])))
        print(('len(right[0])', len(right[0])))

#       print((int(resolution/2),int(resolution*3/2)))
#       print(domain['curve'][0][int(resolution/2):int(resolution*3/2)])

    elif (xi == domain['center'][0]):
        # the second half of the curve

        # compute the number of points in the horizontal and vertical paths
        n_pts_horizontal = int((abs(x_divs[i+1] - xi)) / (abs(x_divs[i+1] - xi) + abs(domain['y_max'] - domain['y_max_cv'])) * resolution)
        n_pts_vertical = resolution - n_pts_horizontal

        top = [ np.hstack(([xi]*n_pts_vertical, np.linspace(xi, x_divs[i+1], n_pts_horizontal))),
            np.hstack((np.linspace(domain['y_max_cv'], domain['y_max'], n_pts_vertical), [domain['y_max']]*n_pts_horizontal)) ]
#       top[0], top[1] = top[0][reverse_ids], top[1][reverse_ids]


        bottom = (np.hstack(([xi]*n_pts_vertical, np.linspace(xi, x_divs[i+1], n_pts_horizontal))),
              np.hstack((np.linspace(domain['y_min_cv'], domain['y_min'], n_pts_vertical), [domain['y_min']]*n_pts_horizontal)) )

        left = [  np.hstack((domain['center'][0], np.array(domain['curve'][0])[range(-int(resolution/2)+1,int(resolution/2)-1)], domain['center'][0])),
              np.hstack((domain['y_min_cv'], np.array(domain['curve'][1])[np.hstack((range(-int(resolution/2)+1, 0), range(1, int(resolution/2))))], domain['y_max_cv'])) ]
#       left[0], left[1] = left[0][reverse_ids], left[1][reverse_ids]
        right = [np.array([x_divs[i+1]]*resolution), np.linspace(domain['y_max'], domain['y_min'], resolution)]
        right[0], right[1] = right[0][reverse_ids], right[1][reverse_ids]

#       print((int(resolution/2),-int(resolution*3/2)))
#       print(domain['curve'][0][int(resolution/2):-int(resolution*3/2)])
#       print((-int(resolution*3/2),-int(resolution/2)))
#       print(domain['curve'][0][-int(resolution*3/2):-int(resolution/2)])

    else:
        # suppose not
        top = [np.linspace(xi, x_divs[i+1], resolution), np.array([domain['y_max']]*resolution)]
#       top[0], top[1] = top[0][reverse_ids], top[1][reverse_ids]
        bottom = (np.linspace(xi, x_divs[i+1], resolution), [domain['y_min']]*resolution)
```

```
      left = [np.array([xi]*resolution), np.linspace(domain['y_max'
                              ], domain['y_min'], resolution)]
      left[0], left[1] = left[0][reverse_ids], left[1][reverse_ids]
      right = [np.array([x_divs[i+1]]*resolution), np.linspace(
                              domain['y_max'], domain['y_min'],
                               resolution)]
      right[0], right[1] = right[0][reverse_ids], right[1][
                              reverse_ids]
#    top, bottom, left, right = [], [], [], []

#    print(top)
#    print(np.array(top).shape)
    print()

#    borders[i].append( [top, bottom, left, right] )
    borders.append( [top, bottom, left, right] )

#    print(('border.shape', np.array(borders).shape))



  return (x_divs, borders)
```

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. As figuras 2.2 e 2.2 mostram malhas geradas utilizando a heurística 2, as cores têm o mesmo significado que o relatado na heurística 1.

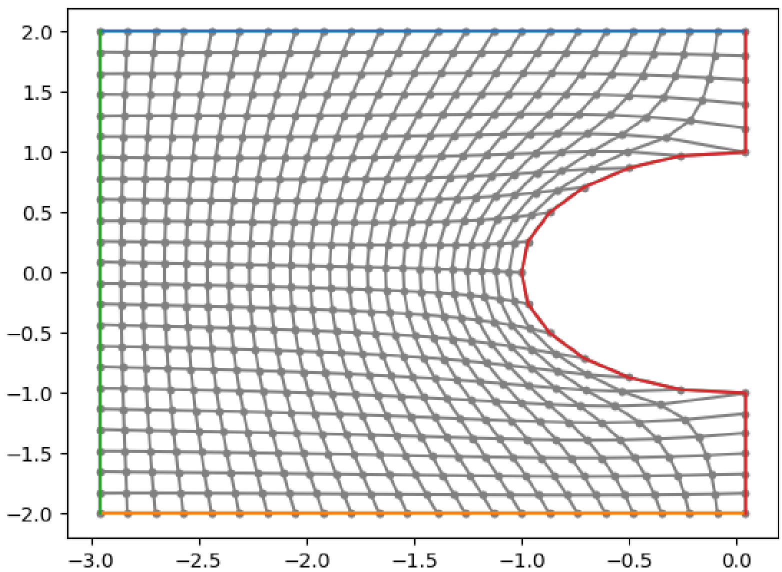

Figura 3: Malha gerada pela heurística 2 na primeira metade do círculo

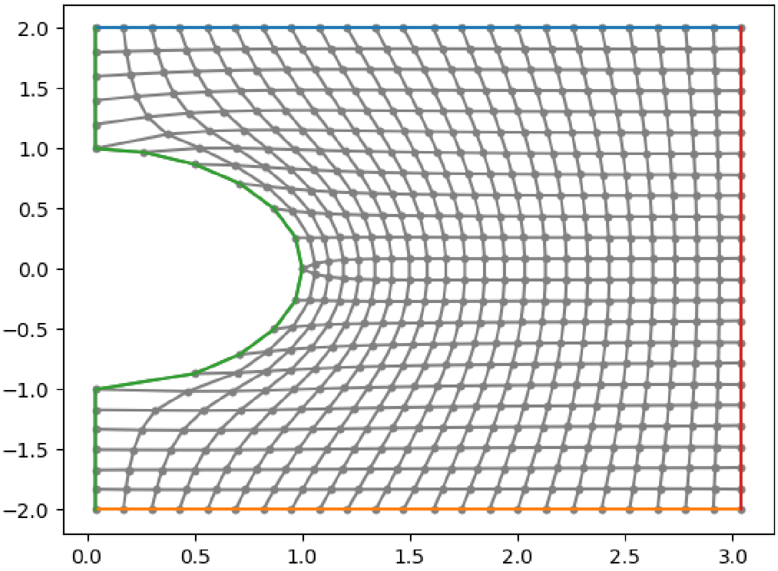

Figura 4: Malha gerada pela heurística 2 na segunda metade do círculo

```
def heuristic_2(domain, k, threshold):
  """
####
#
##
#
##
#

  """
  # TODO think in a way to reduce this case into the previous one
                                just adjusting something

  # the partitions are equally spaced in area
  # the first partition define how much area each should have

  # if it is not possible to have all partitions with the same area
                                ,
  # at least make the partition after the curve with the same
                                principle
  # and then for the rest adjust its size to fit all k partitions


  # ATTENTION make the right border of the previous partition equal
                                 the left border
  # of the next partition

  # start by the division of the curve into two, then continue the
                                partition
```

```
# COPY FROM heuristic_1 and modify just the way the borders are
                                computed



# the resolution of the grid/border is given by the number of
                                points in the
# curve
#resolution:    Integer. The resolution, in number of points, of
                                each partition. Should be
#         half the length of the curve
resolution = len(domain['curve'][0])/2

# the partitions are equally spaced in X
# the first two partitions define how much is left

# if it is not possible to have all partitions equally spaced,
# at least make the partition after the curve with the same
                                principle
# and then for the rest adjust its size to fit all k partitions

# ATTENTION make the right border of the previous partition equal
                                 the left border
# of the next partition

# start by the division of the curve into two, then continue the
                                partition

borders = [[]] * (k+1)
borders = []
# add the leftmost point for the iteration to work later
x_divs = [domain['x_min']]

# the 3 first divisions are different, because the first one I
                                have to check the distance
# between the first division to the curve if it respects the
                                threshold
# then the second is fixed in the center of the curve
# and the third I choose to be symmetric to the second one
                                respect to the curve

# if there is 2*threshold between the leftmost point in the curve
                                 and the left border
# then the first partition occurs on domain['x_min'] + threshold
if abs(domain['x_min'] - domain['x_min_cv']) >= 2* threshold :
  x_divs.append( domain['x_min'] + threshold  )
# divide the curve in two
x_divs.append( domain['center'][0] )

# the next division is symmetrical to the previous one
x_divs.append( x_divs[-1] + abs(x_divs[-1] - x_divs[-2]) )

# the remaining of the domain is equally divided
x_divs.append( np.linspace(start=x_divs[-1], stop=domain['x_max']
                                , num=k - len(x_divs) +3)[1:] )
x_divs = np.hstack(x_divs)

# the convention of the borders on the curve, first half, is
# the left border of the partition is the normal one and the
                                right
```

```
# curve itself
# the top is the top and the vertical part until it reaches the
                                curve
# and the bottom is the bottom and the vertical until it reaches
                                the curve

# the second half of the curve is the same, just changing the
                                left for the right
# and the rest of the partition as the square division, top is
                                top, bottom is bottom,
# left is left and right is right

# TODO have not done it yet
# ATTENTION to the corners, remember to start some in the "second
                                " point

# to avoid roundoff errors of the index, reassign resolution
resolution = int(resolution//4)*8
print(('resolution', resolution))

# the grid generation has to follow the orientation left to right
                                 for the top and bottom
# borders,
# and bottom to top for the left and right borders
# this is because of the code used to generate the grid, if this
                                orientation is not
# followed the generated grid becames twisted

# the order of the borders is top, bottom, left, right
for i, xi in enumerate(x_divs[:-1]):
  # CHECK if we are dealing with the curve partition
  # Think I need to deal with each side separatedly

  direct_ids = list(range(resolution))
  reverse_ids = list(range(resolution))
  reverse_ids.reverse()

  if (x_divs[i+1] == domain['center'][0]):
    # the first half of the curve

    # compute the number of points in the horizontal and vertical
                                 paths
    n_pts_horizontal = int((abs(x_divs[i+1] - xi)) / (abs(x_divs[
                                i+1] - xi) + abs(domain['y_max']
                                - domain['y_max_cv'])) *
                                resolution)
    n_pts_vertical = resolution - n_pts_horizontal

    top = [np.linspace(xi, x_divs[i+1], resolution), np.array([
                                domain['y_max']]*resolution)]
    bottom = (np.linspace(xi, x_divs[i+1], resolution), [domain['
                                y_min']]*resolution)
    left = [  np.array([xi]*int(resolution/4)*4), np.linspace(
                                domain['y_min'], domain['y_max'],
                                 int(resolution/4)*4)]
#     left[0], left[1] = left[0][reverse_ids], left[1][reverse_ids]
    print(('len(right)', len(domain['curve'][0][int(resolution/2)
                                +1:int(resolution*3/2)-1])))
    print((('int(resolution/4)', int(resolution/4))))
    print(('a)', int(resolution/2)))
    print(('b)', int(resolution*3/2)))
    print(('C', np.array([x_divs[i+1]]*int(resolution/4)).shape))
```

```
    print(('D', np.array(domain['curve'][0])[ range(int(
                                    resolution*3/4)-1, int(resolution
                                    /4)-1,-1)].shape))
    right = [ np.hstack((
                np.array([x_divs[i+1]]*int(resolution/4+1)),
#               np.array(domain['curve'][0][int(resolution/2):int(
                                    resolution*3/2)]),
                np.array(domain['curve'][0])[ range(int(resolution*3/
                                    4)-1, int(resolution/4),-1)],
                np.array([x_divs[i+1]]*int(resolution/4)),
              )),
              np.hstack((
                np.linspace(domain['y_min'], domain['y_min_cv'], int(
                                    resolution/4)+1)[:],
#               np.array(domain['curve'][1][int(resolution/2):int(
                                    resolution*3/2)]),
                np.array(domain['curve'][1])[ range(int(resolution*3/
                                    4)-1, int(resolution/4),-1)],
                np.linspace(domain['y_max_cv'], domain['y_max'], int(
                                    resolution/4)),
              ))
          ]
#       right[0], right[1] = right[0][reverse_ids], right[1][
                                    reverse_ids]


#       top = [ np.hstack((np.linspace(xi, x_divs[i+1],
                                    n_pts_horizontal), [x_divs[i+1]]*
                                    n_pts_vertical)),
#           np.hstack(([domain['y_max']]*n_pts_horizontal, np.
                                    linspace(domain['y_max'], domain
                                    ['y_max_cv'], n_pts_vertical))) ]
#       top[0], top[1] = top[0][reverse_ids], top[1][reverse_ids]

#       bottom = (np.hstack((np.linspace(xi, x_divs[i+1],
                                    n_pts_horizontal), [x_divs[i+1]]*
                                    n_pts_vertical)),
#             np.hstack(([domain['y_min']]*n_pts_horizontal, np.
                                    linspace(domain['y_min'], domain
                                    ['y_min_cv'], n_pts_vertical))) )

        # since the curve starts at the rightmost point, I can start
                                    here with the
        # the point located at 1/4 of the curve length and go up
                                    until 3/4
        # JUST EXCHANGED LEFT FOR RIGHT
#       right = [ np.array(domain['curve'][0][int(resolution/2):int(
                                    resolution*3/2)]),
#               np.array(domain['curve'][1][int(resolution/2):int(
                                    resolution*3/2)])]
#       right[0], right[1] = right[0][reverse_ids], right[1][
                                    reverse_ids]
#       left =  [np.array([xi]*resolution), np.linspace(domain['y_max
                                    '], domain['y_min'], resolution)]
#       left[0], left[1] = left[0][reverse_ids], left[1][reverse_ids]

#       print((int(resolution/2),int(resolution*3/2)))
#       print(domain['curve'][0][int(resolution/2):int(resolution*3/2
                                    )])

    elif (xi == domain['center'][0]):
        # the second half of the curve
```

```
    # compute the number of points in the horizontal and vertical
                                    paths
    n_pts_horizontal = int((abs(x_divs[i+1] - xi)) / (abs(x_divs[
                                   i+1] - xi) + abs(domain['y_max']
                                   - domain['y_max_cv'])) *
                                   resolution)
    n_pts_vertical = resolution - n_pts_horizontal


    top = [np.linspace(xi, x_divs[i+1], resolution), np.array([
                                   domain['y_max']]*resolution)]
    bottom = (np.linspace(xi, x_divs[i+1], resolution), [domain['
                                   y_min']]*resolution)
    right = [np.array([x_divs[i+1]]*resolution), np.linspace(
                                   domain['y_min'], domain['y_max'],
                                    resolution)]
#   left[0], left[1] = left[0][reverse_ids], left[1][reverse_ids]
    left = [  np.hstack((
            np.array([xi]*int(resolution/4+1)),
#           np.array(domain['curve'][0][int(resolution/2):int(
                                   resolution*3/2)]),
            np.array(domain['curve'][0])[ np.hstack((range(int(
                                   resolution*3/4)+2, resolution +1)
                                   , range(0, int(resolution/4))))],
            np.array([xi]*int(resolution/4)),
          )),
          np.hstack((
            np.linspace(domain['y_min'], domain['y_min_cv'], int(
                                   resolution/4)+1 )[:],
#           np.array(domain['curve'][1][int(resolution/2):int(
                                   resolution*3/2)]),
            np.array(domain['curve'][1])[ np.hstack((range(int(
                                   resolution*3/4)+2, resolution +1)
                                   , range(0, int(resolution/4))))],
            np.linspace(domain['y_max_cv'], domain['y_max'], int(
                                   resolution/4)),
          ))
        ]


#   top = [ np.hstack(([xi]*n_pts_vertical, np.linspace(xi,
                                   x_divs[i+1], n_pts_horizontal))),
#       np.hstack((np.linspace(domain['y_max_cv'], domain['y_max
                                   '], n_pts_vertical), [domain['
                                   y_max']]*n_pts_horizontal)) ]
#   top[0], top[1] = top[0][reverse_ids], top[1][reverse_ids]


#   bottom = (np.hstack(([xi]*n_pts_vertical, np.linspace(xi,
                                   x_divs[i+1], n_pts_horizontal))),
#         np.hstack((np.linspace(domain['y_min_cv'], domain['
                                   y_min'], n_pts_vertical), [domain
                                   ['y_min']]*n_pts_horizontal)) )

#   left = [  np.array(domain['curve'][0])[range(-int(resolution/
                                   2),int(resolution/2))],
#         np.array(domain['curve'][1])[range(-int(resolution/2),
                                   int(resolution/2))] ]
#   left[0], left[1] = left[0][reverse_ids], left[1][reverse_ids]
#   right = [np.array([x_divs[i+1]]*resolution), np.linspace(
                                   domain['y_max'], domain['y_min'],
```

```
                                resolution)]
#     right[0], right[1] = right[0][reverse_ids], right[1][
                                reverse_ids]

#     print((int(resolution/2),-int(resolution*3/2)))
#     print(domain['curve'][0][int(resolution/2):-int(resolution*3/
                                2)])
#     print((-int(resolution*3/2),-int(resolution/2)))
#     print(domain['curve'][0][-int(resolution*3/2):-int(resolution
                                /2)])

    else:
      # suppose not
      top = [np.linspace(xi, x_divs[i+1], resolution), np.array([
                                domain['y_max']]*resolution)]
      bottom = (np.linspace(xi, x_divs[i+1], resolution), [domain['
                                y_min']]*resolution)
      left = [np.array([xi]*resolution), np.linspace(domain['y_max'
                                ], domain['y_min'], resolution)]
      left[0], left[1] = left[0][reverse_ids], left[1][reverse_ids]
      right = [np.array([x_divs[i+1]]*resolution), np.linspace(
                                domain['y_max'], domain['y_min'],
                                 resolution)]
      right[0], right[1] = right[0][reverse_ids], right[1][
                                reverse_ids]
#   top, bottom, left, right = [], [], [], []

#   print(top)
#   print(np.array(top).shape)
    print()

#   borders[i].append( [top, bottom, left, right] )
    borders.append( [top, bottom, left, right] )

#   print(('border.shape', np.array(borders).shape))



  return (x_divs, borders)
```

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 *Poisson* em cada partição individualmente.

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*.

```
def generate_grid(resolution=100, left_border=1, domain_length=10,
                                domain_height=4,
          curve_params={'radius':1}, equation=circle,
                                filename_curve='',
          heuristic=heuristic_1, k=4, filename_borders='circle',
          iter_number=100, xis_rf=[], etas_rf=[], points_rf=[],
          a_xis=[], b_xis=[], c_xis=[], d_xis=[],
          a_etas=[], b_etas=[], c_etas=[], d_etas=[], plot=False):
  import poisson
```

```
  # load or create the curve
  domain = generate_curve(resolution, left_border, domain_length,
                                  domain_height,
          curve_params, equation, filename_curve)

  # partitionate the domain and create the borders
  borders = partitionate_domain(domain, k, heuristic,
                                  filename_borders)

  # generate the grid from the partitions

  grid = []
  for f in range(k+1):
    grid.append( poisson.grid(filename='%s_part_%d.txt'%(
                                  filename_borders, f),
            save_file='%s_part_%d.vtk'%(filename_borders, f),
                                  iter_number=iter_number,
            xis_rf=xis_rf[f] if len(xis_rf) != 0 else [],
            etas_rf=etas_rf[f] if len(etas_rf) != 0 else [],
            points_rf=points_rf[f] if len(points_rf) != 0 else [],
            a_xis=a_xis[f] if len(a_xis) != 0 else [],
            b_xis=b_xis[f] if len(b_xis) != 0 else [],
            c_xis=c_xis[f] if len(c_xis) != 0 else [],
            d_xis=d_xis[f] if len(d_xis) != 0 else [],
            a_etas=a_etas[f] if len(a_etas) != 0 else [],
            b_etas=b_etas[f] if len(b_etas) != 0 else [],
            c_etas=c_etas[f] if len(c_etas) != 0 else [],
            d_etas=d_etas[f] if len(d_etas) != 0 else [],
            plot=plot) )


  # merging the grids into one
  grid = np.array(grid)
  final_grid = np.array(( np.vstack(grid[:, 0, :, :]), np.vstack(
                                  grid[:, 1, :, :]) ))
# final_grid_y = np.array(( np.hstack(grid[:, 0, :, :]).transpose()
                                  , np.hstack(grid[:, 1, :, :]).
                                  transpose() ))

  print(final_grid.shape)
# print(final_grid_y.shape)

  # ATTENTION there is an error in here
  # create the vtk for the whole grid
  from grid2vtk import grid2vtk
```

3 Resultados

as duas heurísticas com e sem refinamento depois o aerofólio nas duas heurísticas, com e sem refinamento também