# Relatório do Projeto 1

Daniel Moreira Cestari - 5746193

29 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 *Laplace* e ao final são unidas em duas malhas que são refinadas na região desejada.

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 given 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 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.

```
def partitionate_domain(domain, k, heuristic, filename=''):
  """
####
# Partitionate the domain, generate the borders, and save to file
                                    all partitions
###
# domain:   Dictionary. The dictionary with the domain information,
                                     the same returned
#       by the function generate_curve
# k:      Integer. The number of splits to perform on the domain
# heuristic:  Function. The function that split the domain and
                                    generate the border.
#       See docstring for function heuristic_1 and heuristic_2 for
                                    more details.
# filename:   String. An identifier of the execution, used to name
                                    the files saved
##
# Return the list of the borders for every partition, and saves to
                                    file the borders

# Usage example:
```

```
imp.reload(pjt); vv=pjt.generate_curve(100, 2, 10, 5, {'radius':1},
                                       pjt.circle, ''); k=4; bb = pjt.
                                      heuristic_1(vv, k, 2); mm=bb[1];
                                      [[plt.plot(mm[xk][i][0], mm[xk][i
                                      ][1], '*') for i in range(4)] for
                                       xk in range(k)]; plt.show(); plt
                                      .close('all')


  """
  # 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):
  """
####
# Partitionate the domain and generate the borders following the
```

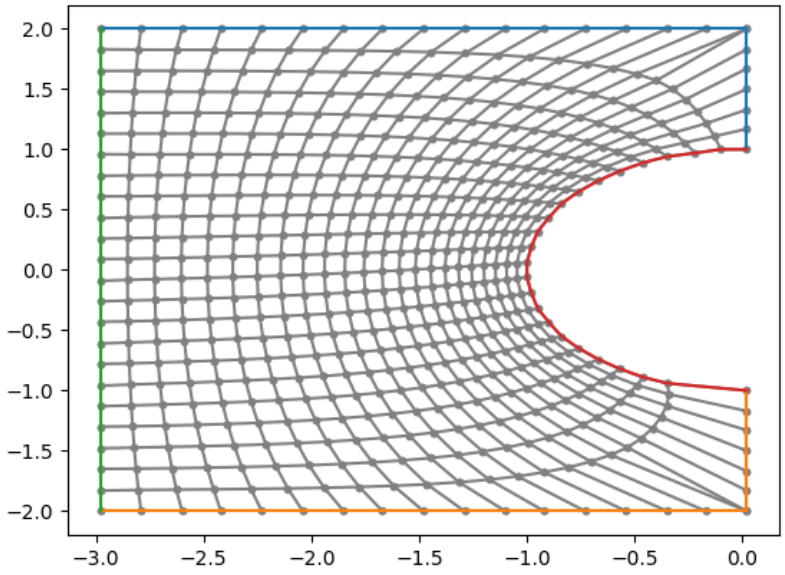


Figura 1: Malha gerada pela heurística 1 na primeira metade do círculo

```
                                    heuristic 2
##
# domain:   Dictionary. The dictionary with the domain information,
                                     the same returned
#       by the function generate_curve
# k:      Integer. The number of splits to perform on the domain
# threshold:  Number. The minimum distance between the first half
                                    of the curve and the
#       left border of the given partition. If the distance between
                                     the curve and the
#       start of the domain is less than the given threshold does
                                    not perform a split
#       before the curve. Otherwise perform a split before the
                                    curve.
##
# Return a tuple were the first element is the positions where the
                                    domain
# were split, and the second is the list of the borders for every
                                    partition
#
# The first heuristic turns the points on the left as the left
                                    border, and on the curve only as
                                    the right border, unless the
                                    curve is on the right, then this
                                    logic is inverse. The other
                                    points become the top and bottom
                                    borders

# Usage example:
```

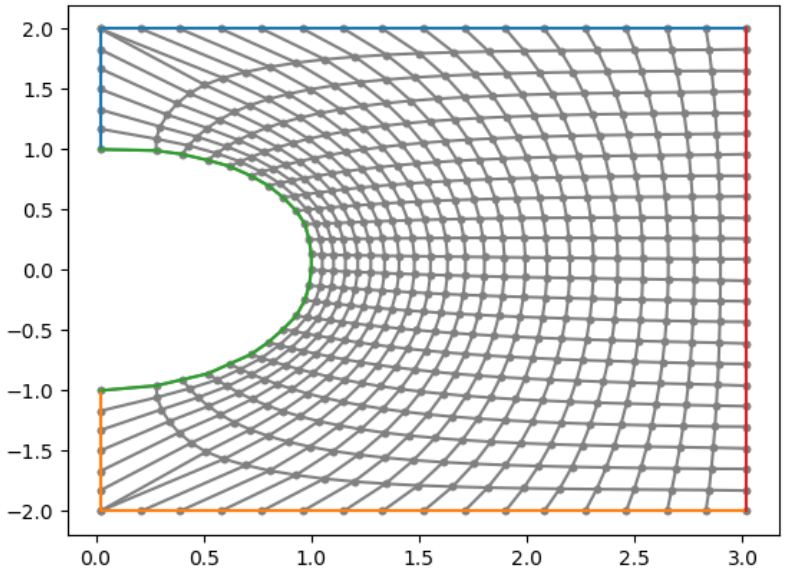


Figura 2: Malha gerada pela heurística 1 na segunda metade do círculo

```
"""

# the resolution of the grid/border is given by the number of
                                    points in the curve
resolution = len(domain['curve'][0])/2

# the partitions are equally spaced in X
# the first two partitions define how much of the domain 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/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]


    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]


    else:
      # the rest of the domain, i.e., the partitions without the
                                curve

      # 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]

#   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.

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

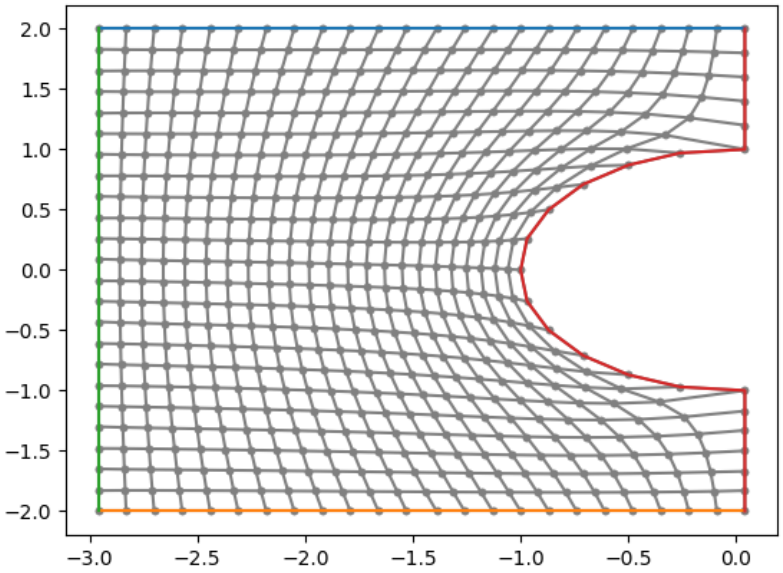


Figura 3: Malha gerada pela heurística 2 na primeira metade do círculo

```
# Partitionate the domain and generate the borders following the
                                    heuristic 2
##
# domain:   Dictionary. The dictionary with the domain information,
                                     the same returned
#       by the function generate_curve
# k:      Integer. The number of splits to perform on the domain
# threshold:  Number. The minimum distance between the first half
                                    of the curve and the
#       left border of the given partition. If the distance between
                                     the curve and the
#       start of the domain is less than the given threshold does
                                    not perform a split
#       before the curve. Otherwise perform a split before the
                                    curve.
##
# Return a tuple were the first element is the positions where the
                                    domain
# were split, and the second is the list of the borders for every
                                    partition
#
# The second heuristic create the borders fitting the points into a
                                     square, so points to the left
                                    become the left borders, to the
                                    right the right borders and so on
                                    .

# Usage example:
```

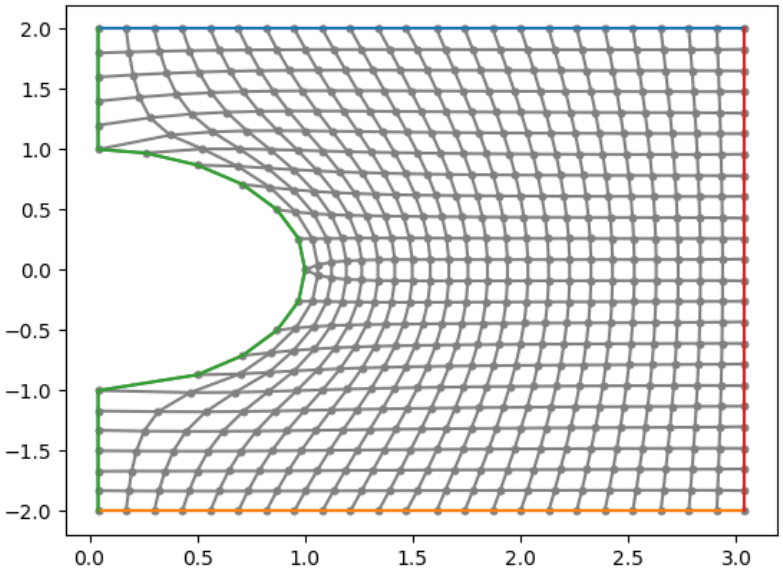


Figura 4: Malha gerada pela heurística 2 na segunda metade do círculo

```
"""
# 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 = 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

  # 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)
      # 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
      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]
      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)),
            ))
          ]


    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)),
            ))
          ]


    else:
      # the rest of the domain, i.e., the partitions without the
                                curve

      # 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]

#   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 *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.

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_rf0=[], etas_rf0=[], points_rf0=[],
          a_xis0=[], b_xis0=[], c_xis0=[], d_xis0=[],
          a_etas0=[], b_etas0=[], c_etas0=[], d_etas0=[],
          xis_rf1=[], etas_rf1=[], points_rf1=[],
          a_xis1=[], b_xis1=[], c_xis1=[], d_xis1=[],
          a_etas1=[], b_etas1=[], c_etas1=[], d_etas1=[],
          plot=False):
  """
####
# Generate the grid for a given configuration
###

# Parameters regarding the curve generation
# 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_curve:   String. The filename from which the curve will
                                    be read from

# Parameters regarding the domain partition and borders creation
# domain:       Dictionary. The dictionary with the domain
                                    information, the same returned
#           by the function generate_curve
```

```
# k:          Integer. The number of splits to perform on the
                                    domain
# heuristic:      Function. The function that split the domain and
                                    generate the border.
#           See docstring for function heuristic_1 and heuristic_2
                                    for more details.
# filename_borders:   String. An identifier of the execution, used
                                    to name the files saved

# Parameters related to the Poisson's equation, i.e., grid
                                    generation
# iter_number:      Integer. Number of iterations to solve the
                                    Poisson's equation

# Refinement parameters of the first half of the grid
# xis_rf0:        List. The list with the positions on xi to refine
                                     the grid
# etas_rf0:       List. The list with the positions on eta to
                                    refine the grid
# points_rf0:     List. The list with the points to refine the grid
# a_xis0:       List. Parameter a in the TTM method to refine the
                                    grid, related to xi
# b_xis0:       List. Parameter b in the TTM method to refine the
                                    grid, related to xi
# c_xis0:       List. Parameter c in the TTM method to refine the
                                    grid, related to xi
# d_xis0:       List. Parameter d in the TTM method to refine the
                                    grid, related to xi
# a_etas0:        List. Parameter a in the TTM method to refine the
                                     grid, related to eta
# b_etas0:        List. Parameter b in the TTM method to refine the
                                     grid, related to eta
# c_etas0:        List. Parameter c in the TTM method to refine the
                                     grid, related to eta
# d_etas0:        List. Parameter d in the TTM method to refine the
                                     grid, related to eta

# Refinement parameters of the second half of the grid
# xis_rf1:        List. The list with the positions on xi to refine
                                     the grid
# etas_rf1:       List. The list with the positions on eta to
                                    refine the grid
# points_rf1:     List. The list with the points to refine the grid
# a_xis1:       List. Parameter a in the TTM method to refine the
                                    grid, related to xi
# b_xis1:       List. Parameter b in the TTM method to refine the
                                    grid, related to xi
# c_xis1:       List. Parameter c in the TTM method to refine the
                                    grid, related to xi
# d_xis1:       List. Parameter d in the TTM method to refine the
                                    grid, related to xi
# a_etas1:        List. Parameter a in the TTM method to refine the
                                     grid, related to eta
# b_etas1:        List. Parameter b in the TTM method to refine the
                                     grid, related to eta
# c_etas1:        List. Parameter c in the TTM method to refine the
                                     grid, related to eta
# d_etas1:        List. Parameter d in the TTM method to refine the
                                     grid, related to eta

# plot:         Boolean. True to use matplotlib to plot the grids
                                    generated, for the
```

```
#           partitions and for the grid as a whole


###
# Returns a tuple with the grids generated, a list with the grid
                                    for every partition,
# and the tuple's second element is the grid as a whole, all grid
                                    merged together
# Also generate the VTK for every partition and for the whole grid,
                                     saves all to file


# Usage example:

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")


# with refinement parameters, both heuristics works with the same
                                    parameters
imp.reload(pjt);  grid = pjt.generate_grid(resolution=50,
                                    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",
                                    xis_rf0=[1], xis_rf1=[0],
                                    etas_rf1=[0.4], a_xis0=[5],
                                    a_xis1=[2.5], c_xis0=[5], c_xis1
                                    =[5], a_etas1=[5], c_etas1=[15])


imp.reload(pjt);  grid = pjt.generate_grid(resolution=50,
                                    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",
                                    xis_rf0=[1], xis_rf1=[0],
                                    etas_rf1=[0.45], a_xis0=[5],
                                    a_xis1=[5], c_xis0=[5], c_xis1=[5
                                    ], a_etas1=[7.5], c_etas1=[20])


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_2, k=3,
                                    filename_borders="circle")


# merging the grids into one
grid = np.array(grid)
final_grid_x = np.vstack(grid[:, 0, :, :]), np.vstack(grid[:, 1, :,
                                     :])
final_grid_y = np.hstack(grid[:, 0, :, :]), np.hstack(grid[:, 1, :,
                                     :])
nx,ny = final_grid_x[0].shape



nx,ny = final_grid_x[0].shape; [plt.plot(final_grid_x[0][i, :],
                                    final_grid_x[1][i,:], ".-", color
                                    ="gray") for i in range(nx)]; [
                                    plt.plot(final_grid_y[0][:,i],
                                    final_grid_y[1][:,i], ".-", color
                                    ="gray") for i in range(nx)]; plt
                                    .show(); plt.close("all")
  """
  import poisson
  imp.reload(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):
    # do not refine now, only refine after the two final grid parts
                                     are finished
    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=[], etas_rf=[], points_rf=[],
            a_xis=[], b_xis=[], c_xis=[], d_xis=[],
            a_etas=[], b_etas=[], c_etas=[], d_etas=[],
            plot=plot) )


  # merging the grids into two parts
  grid = np.array(grid)
# print((grid[0].shape[2]))
  print(('grid.shape', grid.shape))
# np.hstack(grid[:, 0, 0, :])
  threshold = abs(domain['x_min_cv'] - domain['center'][0])
  id_half = 2 if abs(domain['x_min'] - domain['x_min_cv']) >= 2*
                                    threshold else 1
  split_grid = np.array((
          np.array((
            [np.hstack(grid[:id_half, 0, i, :]) for i in range(grid
                                    [0].shape[2])] ,
```

```
            [np.hstack(grid[:id_half, 1, i, :]) for i in range(grid
                                [0].shape[2])]
          )),
          np.array((
            [np.hstack(grid[id_half:, 0, i, :]) for i in range(grid
                                [0].shape[2])] ,
            [np.hstack(grid[id_half:, 1, i, :]) for i in range(grid
                                [0].shape[2])]
          ))
        ))
  print(('split_grid', split_grid.shape))


  final_grid = np.array((
          [np.hstack(grid[:, 0, i, :]) for i in range(grid[0].shape
                                [2])] ,
          [np.hstack(grid[:, 1, i, :]) for i in range(grid[0].shape
                                [2])]
          ))
# 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() ))

  # TODO check if the vtk is ok, it does not seem so
  # I think the only solution is to work with two halves of the
                                grid
  # before the hole, i.e., the curve, and after

  # get the index of the position of the center of the curve, where
                                 the grid should be split
  threshold = abs(domain['x_min_cv'] - domain['center'][0])
  id_half = final_grid.shape[1]
  id_half *= 3 if abs(domain['x_min'] - domain['x_min_cv']) >= 2*
                                threshold else 2



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

  # plotting the two halves before the refinement
  NULL, nx,ny = split_grid[0].shape
  [plt.plot(split_grid[0][0][:,i], split_grid[0][1][:,i], '.-',
                                color='gray') for i in range(ny)]
                                ;
  [plt.plot(split_grid[1][0][:,i], split_grid[1][1][:,i], '.-',
                                color='gray') for i in range(ny)]
                                ;
  NULL, nx,ny = split_grid[1].shape
  [plt.plot(split_grid[0][0][i,:], split_grid[0][1][i,:], '.-',
                                color='gray') for i in range(nx)]
                                ;
  [plt.plot(split_grid[1][0][i,:], split_grid[1][1][i,:], '.-',
                                color='gray') for i in range(nx)]
                                ;
  plt.show(); plt.close('all')


  # apply the refinement but for the first half
```

```
final_grid1 = poisson.grid(filename='',
          save_file='%s_final_refined_1.vtk'%(filename_borders),
          iter_number=iter_number,
          xis_rf=xis_rf0 if len(xis_rf0) != 0 else [],
          etas_rf=etas_rf0 if len(etas_rf0) != 0 else [],
          points_rf=points_rf0 if len(points_rf0) != 0 else [],
          a_xis=a_xis0 if len(a_xis0) != 0 else [],
          b_xis=b_xis0 if len(b_xis0) != 0 else [],
          c_xis=c_xis0 if len(c_xis0) != 0 else [],
          d_xis=d_xis0 if len(d_xis0) != 0 else [],
          a_etas=a_etas0 if len(a_etas0) != 0 else [],
          b_etas=b_etas0 if len(b_etas0) != 0 else [],
          c_etas=c_etas0 if len(c_etas0) != 0 else [],
          d_etas=d_etas0 if len(d_etas0) != 0 else [],
          plot=plot,
          comp_grid=(split_grid[0][0], split_grid[0][1]))


# apply the refinement but for the second half
final_grid2 = poisson.grid(filename='',
          save_file='%s_final_refined_2.vtk'%(filename_borders),
          iter_number=iter_number,
          xis_rf=xis_rf1 if len(xis_rf1) != 0 else [],
          etas_rf=etas_rf1 if len(etas_rf1) != 0 else [],
          points_rf=points_rf1 if len(points_rf1) != 0 else [],
          a_xis=a_xis1 if len(a_xis1) != 0 else [],
          b_xis=b_xis1 if len(b_xis1) != 0 else [],
          c_xis=c_xis1 if len(c_xis1) != 0 else [],
          d_xis=d_xis1 if len(d_xis1) != 0 else [],
          a_etas=a_etas1 if len(a_etas1) != 0 else [],
          b_etas=b_etas1 if len(b_etas1) != 0 else [],
          c_etas=c_etas1 if len(c_etas1) != 0 else [],
          d_etas=d_etas1 if len(d_etas1) != 0 else [],
          plot=plot,
          comp_grid=(split_grid[1][0], split_grid[1][1]))

# plotting the two halves after the refinement
NULL, nx,ny = split_grid[0].shape
[plt.plot(final_grid1[0][:,i], final_grid1[1][:,i], '.-', color='
                                gray') for i in range(ny)];
[plt.plot(final_grid2[0][:,i], final_grid2[1][:,i], '.-', color='
                                gray') for i in range(ny)];
NULL, nx,ny = split_grid[1].shape
[plt.plot(final_grid1[0][i,:], final_grid1[1][i,:], '.-', color='
                                gray') for i in range(nx)];
[plt.plot(final_grid2[0][i,:], final_grid2[1][i,:], '.-', color='
                                gray') for i in range(nx)];
plt.show(); plt.close('all')

"""
# TODO smooth the borders on the interface between halves, only
                                  the y (eta) values
mask = [True] * final_grid1[1].shape[0]
mask = [False if j in domain['curve'][1] else mask[i] for i, j in
                                   enumerate( final_grid1[1][:, -1]
                                   )]
mask = [False if j in domain['curve'][1] else mask[i] for i, j in
                                   enumerate( final_grid2[1][:, 0]
                                  )]
print(('mask', mask))
print()
avg = np.average( np.array(( final_grid1[1][:, -2], final_grid2[1
```

```
                                    ][:, 1] )), axis=0)
  final_grid1[1][:, -1] = avg
  final_grid2[1][:, 0] = avg
  print(('final_grid1[1].shape',final_grid1[1].shape))
  print(('final_grid2[1].shape',final_grid2[1].shape))
  print(('avg.shape', avg.shape))
  print(('np.average(avg).shape', np.average(avg).shape))
  print()
  print(('np.average(avg, axis=0)', np.average(avg, axis=0)))
  print()
  print(('final_grid1[1][:,-2]',final_grid1[1][:,-2]))
  print()
  print(('final_grid2[1][:,1]',final_grid2[1][:,1]))


  NULL, nx,ny = split_grid[0].shape
  [plt.plot(final_grid1[0][:,i], final_grid1[1][:,i], '.-', color='
                                    gray') for i in range(ny)];
  [plt.plot(final_grid2[0][:,i], final_grid2[1][:,i], '.-', color='
                                    gray') for i in range(ny)];
  NULL, nx,ny = split_grid[1].shape
  [plt.plot(final_grid1[0][i,:], final_grid1[1][i,:], '.-', color='
                                    gray') for i in range(nx)];
  [plt.plot(final_grid2[0][i,:], final_grid2[1][i,:], '.-', color='
                                    gray') for i in range(nx)];
  plt.show(); plt.close('all')
  """

  # modify grid2vtk to receive two halves, acutally can be several
                                    parts, and properly create one
                                    vtk

  import grid2vtk as g2vtk
  imp.reload(g2vtk)
# grid2vtk(final_grid[0], final_grid[1], '%s_final.vtk'%
                                    filename_borders)
  g2vtk.grid2vtk([final_grid1[0], final_grid2[0]], [final_grid1[1],
                                     final_grid2[1]], '%s_final.vtk'%
                                    filename_borders)


# return (grid, final_grid_x, final_grid_y)
  return (grid, final_grid1, final_grid2)
```

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.

# 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.

Também utilizarei como curva o perfil do aerofólio definido no arquivo *naca012.txt*.

## 3.1 Heurística 1

Código para gerar uma malha com 50 pontos nos bordos esquerdo e direito. Curva gerada com 50 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 feito ao redor da curva e após a curva no centro do domínio em relação à $y$, ou $\eta$.

```
imp.reload(pjt);
grid = pjt.generate_grid(resolution=50, 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",
xis_rf0=[1], xis_rf1=[0], etas_rf1=[0.4], a_xis0=[5],
a_xis1=[2.5], c_xis0=[5], c_xis1=[5], a_etas1=[5], c_etas1=[15])
```

Abaixo o código agora utilizando 100 pontos.

```
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",
xis_rf0=[1], xis_rf1=[0], etas_rf1=[0.4], a_xis0=[5],
a_xis1=[2.5], c_xis0=[5], c_xis1=[5], a_etas1=[5], c_etas1=[15])
```

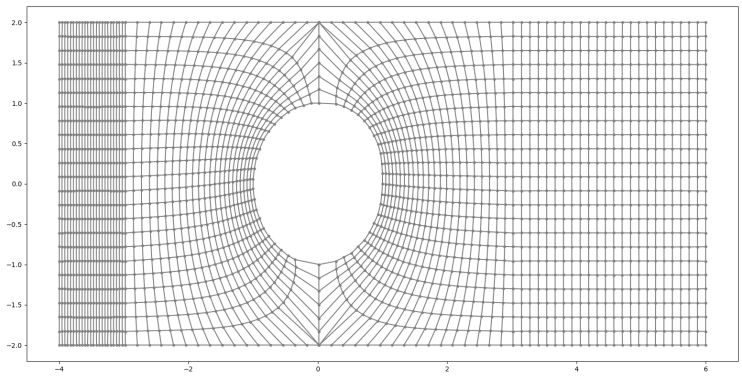


Figura 5: Malha gerada pela heurística 1 com 50 pontos sem refinamento.

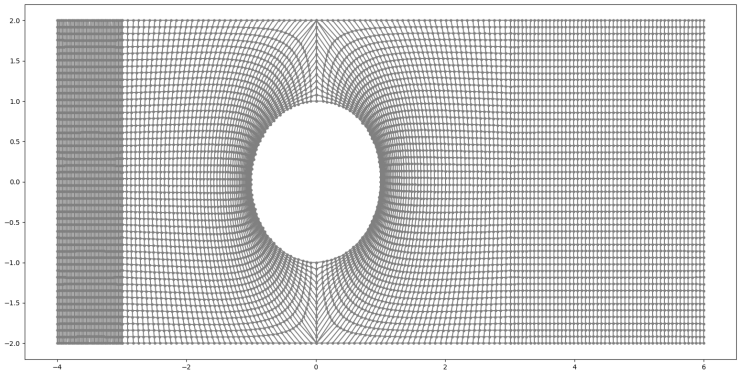


Figura 6: Malha gerada pela heurística 1 com 100 pontos sem refinamento.

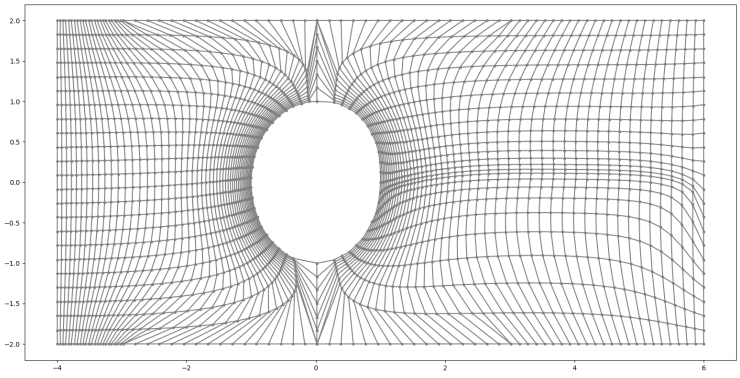


Figura 7: Malha gerada pela heurística 1 com 50 pontos com refinamento.

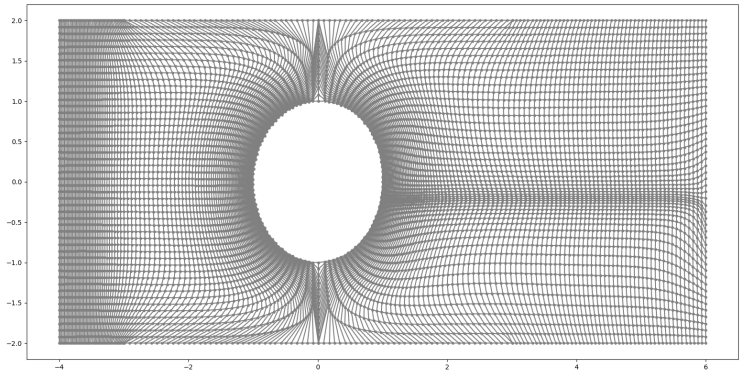


Figura 8: Malha gerada pela heurística 1 com 100 pontos com refinamento.

## 3.2 Heurística 2

Código para gerar uma malha com 25 pontos nos bordos esquerdo e direito. Curva gerada com 25 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 feito ao redor da curva e após a curva no centro do domínio em relação à $y$, ou $\eta$.

```
imp.reload(pjt);
grid = pjt.generate_grid(resolution=25, left_border=3, domain_length=10,
domain_height=4, curve_params={"radius":1}, equation=pjt.circle,
filename_curve="", heuristic=pjt.heuristic_2, k=3, filename_borders="circle_h2_25pts",
xis_rf0=[1], xis_rf1=[0], etas_rf1=[0.4], a_xis0=[5],
a_xis1=[2.5], c_xis0=[5], c_xis1=[5], a_etas1=[5], c_etas1=[15])
```

Abaixo o código agora utilizando 50 pontos.

```
imp.reload(pjt);
grid = pjt.generate_grid(resolution=50, left_border=3, domain_length=10,
domain_height=4, curve_params={"radius":1}, equation=pjt.circle,
filename_curve="", heuristic=pjt.heuristic_2, k=3, filename_borders="circle_h2_50pts",
xis_rf0=[1], xis_rf1=[0], etas_rf1=[0.4], a_xis0=[5],
a_xis1=[2.5], c_xis0=[5], c_xis1=[5], a_etas1=[5], c_etas1=[15])
```

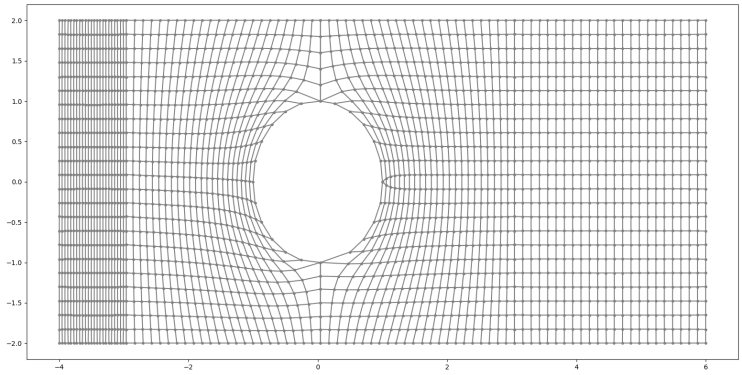


Figura 9: Malha gerada pela heurística 2 com 25 pontos sem refinamento.

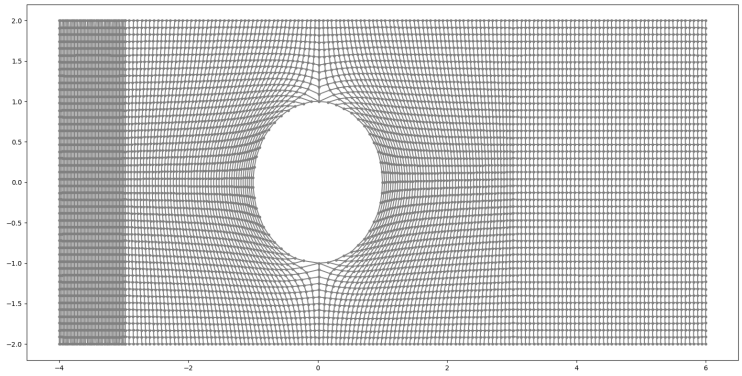


Figura 10: Malha gerada pela heurística 2 com 50 pontos sem refinamento.

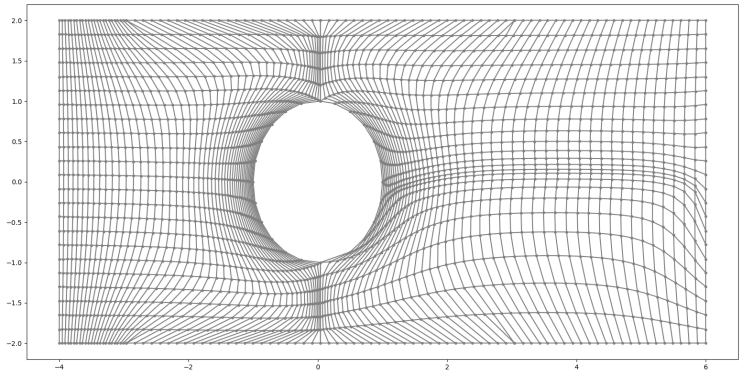

Figura 11: Malha gerada pela heurística 2 com 25 pontos com refinamento.

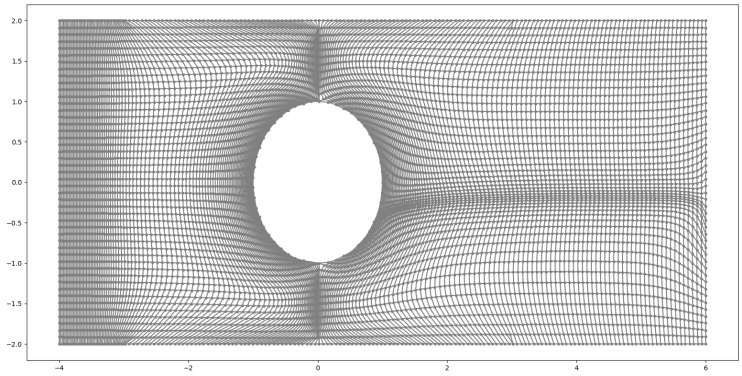

Figura 12: Malha gerada pela heurística 2 com 50 pontos com refinamento.

## 3.3 Aerofólio

Código para gerar uma malha para a curva *naca012*, a resolução da malha é determinada pela quantidade de pontos na curva. O domínio com comprimento 4, altura 6, e comprimento 1 à direita da curva. O domínio é divido em 4 partes, e o refinamento é feito ao redor da curva e após a curva no centro do domínio em relação à $y$, ou $\eta$.

Código utilizando a heurística 1.

```
imp.reload(pjt);
grid = pjt.generate_grid(filename_curve="naca012.txt",
left_border=1, domain_length=4, domain_height=3,
```

```
heuristic=pjt.heuristic_1, k=3, filename_borders="naca012_h1",
xis_rf0=[1], xis_rf1=[0], etas_rf1=[0.419], a_xis0=[5],
a_xis1=[2.5], c_xis0=[5], c_xis1=[5], a_etas1=[5], c_etas1=[15])
```

Código utilizando a heurística 2.

```
imp.reload(pjt);
grid = pjt.generate_grid(filename_curve="naca012.txt",
left_border=1, domain_length=4, domain_height=3,
heuristic=pjt.heuristic_2, k=3, filename_borders="naca_h2",
xis_rf0=[1], xis_rf1=[0], etas_rf1=[0.419], a_xis0=[5],
a_xis1=[2.5], c_xis0=[5], c_xis1=[5], a_etas1=[5], c_etas1=[15])
```

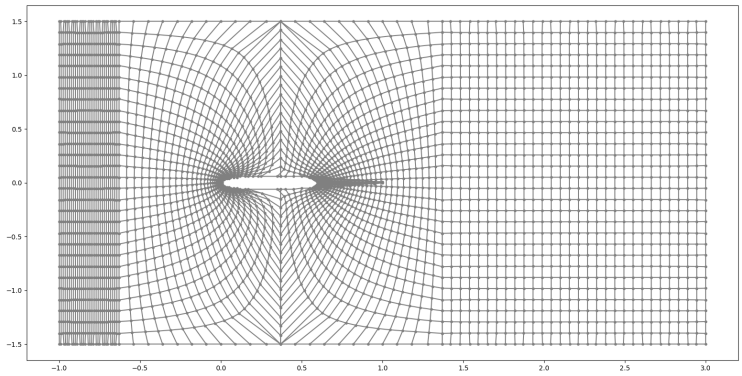


Figura 13: Malha gerada para a curva *naca012* pela heurística 1 sem refinamento.

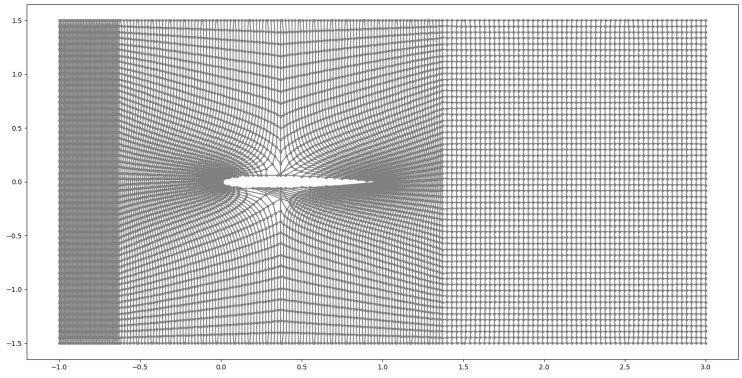


Figura 14: Malha gerada para a curva *naca012* pela heurística 2 sem refinamento.

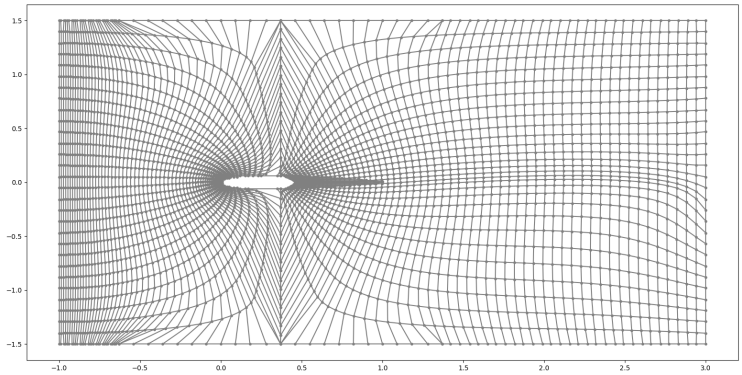


Figura 15: Malha gerada para a curva *naca012* pela heurística 1 com refinamento.

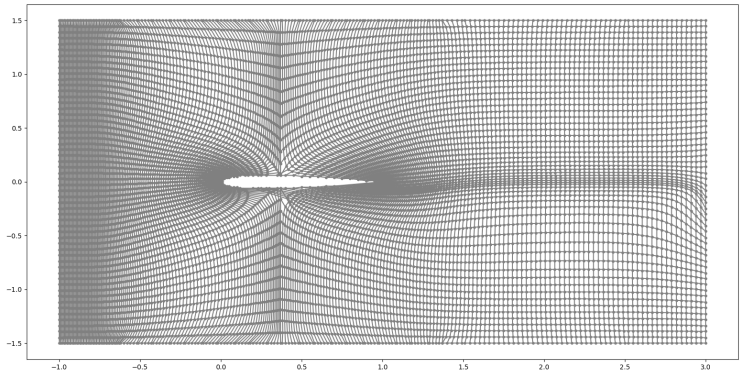


Figura 16: Malha gerada para a curva *naca012* pela heurística 2 com refinamento.