Modelação Numérica 2017

Aula 18, 2/Maio

- Estimativa de parâmetros e optimização: Downslope Monte Carlo

http://modnum.ucs.ciencias.ulisboa.pt

Aula passada

Estimativa/optimização de parâmetros

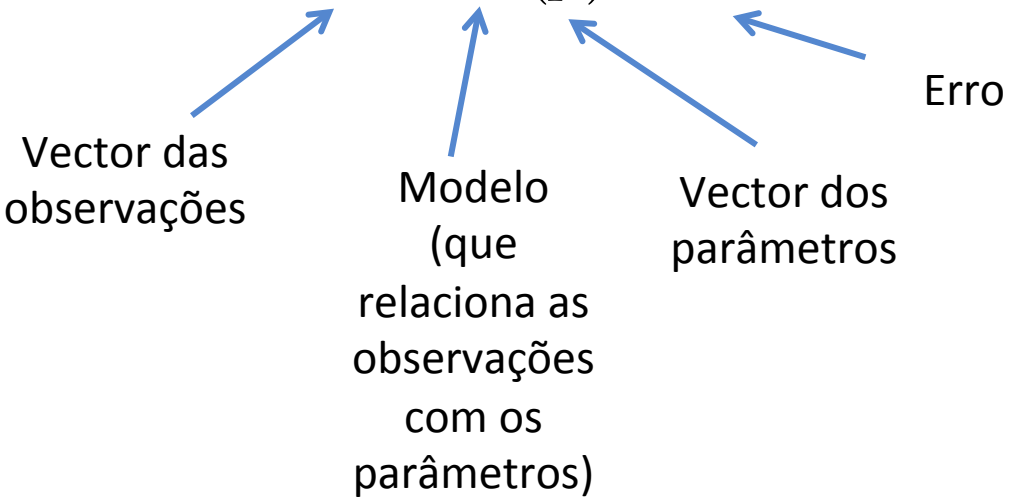
- Muitas vezes queremos estimar parâmetros que não conseguimos medir directamente. No entanto, podemos conseguir relacionar os parâmetros que queremos estimar com medições que conseguimos fazer. Dizemos então que queremos estimar parâmetros de um modelo. [Por exemplo, queremos descobrir a densidade das rochas/minérios a certa profundidade].
- Realizamos então um conjunto de medidas (observações) que sabemos relacionar com os parâmetros a estimar [Por exemplo, vamos medir a aceleração da gravidade à superfície da Terra, para estimar a densidade em profundidade].
- As medições têm erro, e os parâmetros estimados também têm erro.

Aula passada

Estimativa/optimização de parâmetros

- Em linguagem matemática, o problema a resolver é o seguinte:

$$\vec{o} = M(\vec{p}) + \varepsilon$$



- Se o modelo fosse linear, ficaríamos com: $\vec{o} = L\vec{p} + \varepsilon$

sendo *L* a matriz dos coeficientes do modelo.

Aula passada

Problema linear sobredeterminado

- O caso mais simples é o da regressão linear:

$$y = ax + b$$

- Os parâmetros a estimar são: a, b.
- As observações são os valores y.

- Exemplo: Quero estimar a velocidade de um carro (v) e a sua posição de origem (x_0) [parâmetros], e para tal vou medir a sua posição (x) [observações] em determinados tempos (t).

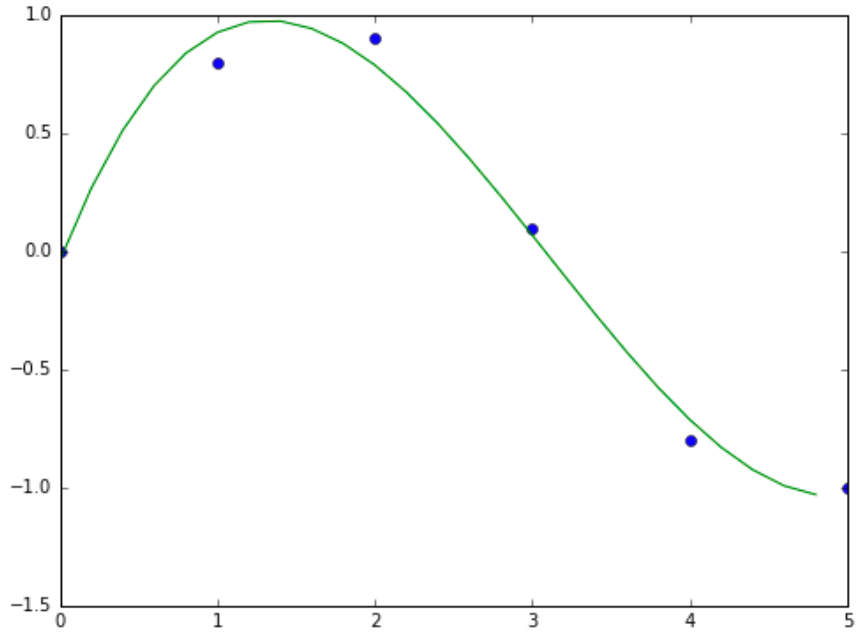
$$x = vt + x_0$$

Aula passada

Regressão linear (com constrangimento)

- A função numy.polyfit calcula os parâmetros (a, b) da regressão linear, ou os de uma regressão polinomial de ordem mais elevada (e.g: $y = ax^2 + bx + c$), minimizando o erro médio quadrático.

```
import matplotlib.pyplot as plt
import numpy as np

#%%
plt.rcParams['figure.figsize'] = 8, 6

x = np.array([0.0, 1.0, 2.0, 3.0,  4.0,  5.0])
y = np.array([0.0, 0.8, 0.9, 0.1, -0.8, -1.0])
p = np.polyfit(x, y, 3)

x2=np.arange(0,5,.2)
yfit = p[3] + p[2]*x2 + p[1]*x2**2 + p[0]*x2**3

plt.plot(x,y, 'o', x2,yfit)
```



Aula passada

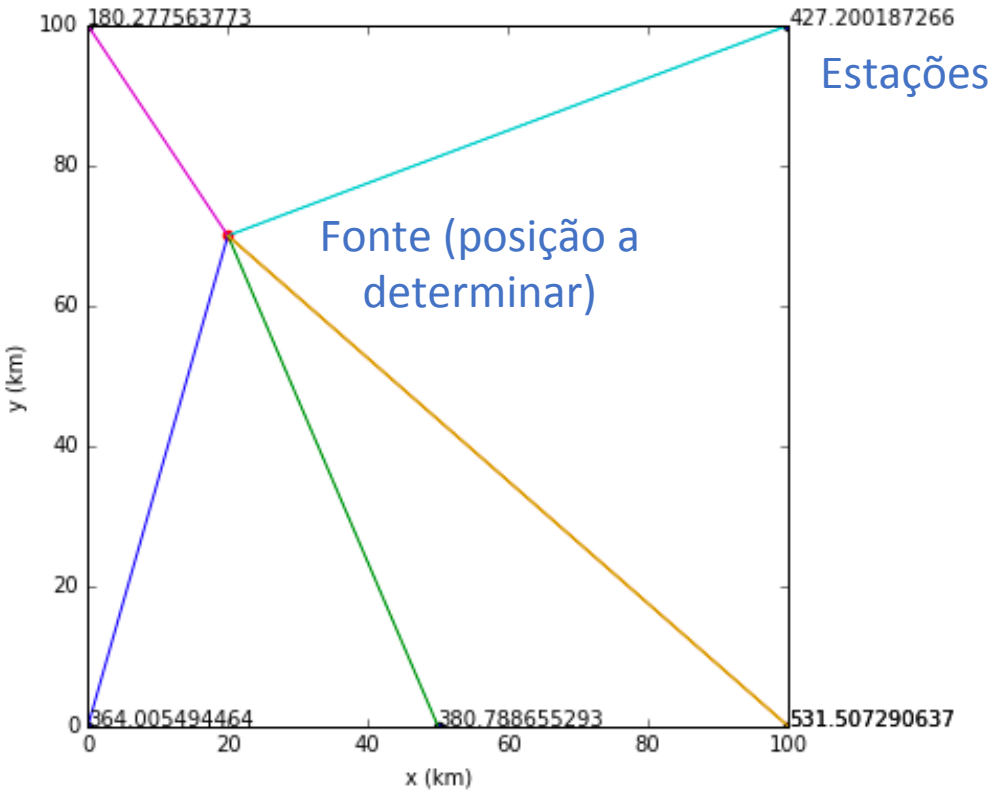
Caso geral: função custo

- Em geral, não é possível obter diretamente (analiticamente) uma solução óptima.
- Nesses casos, é necessário proceder iterativamente.
- Os métodos iterativos requerem:
- Um método para obter potenciais soluções, e de as aceitar se for caso disso.
  - Função custo J, ex:
    - Minimização do erro (norma L1): $J = \varepsilon = |\vec{o} - M(\vec{p})|$
    - Minimização do erro quadrático (norma L2): $J = \varepsilon^2 = |\vec{o} - M(\vec{p})|^2$
  - Aceitar soluções que (pelo menos em média) reduzam J.
- Um critério de paragem:
  - Número de iterações.
  - Valor atingido pela função custo.
  - Falta de progresso em J ou em p

Um problema real:

- Temos uma fonte (sismo, tsunami, emissor GPS, etc) que gera ondas que se propagam e que são registadas em estações.
- Nas estações mais longe da fonte, as chegadas das ondas são registadas mais tarde.

- Queremos localizar a fonte (sismo, tsunami, emissor GPS)
- Conhecendo os tempos de chegada das ondas às estações.
- E sabendo a velocidade de propagação das ondas (caso simples, c=contante)

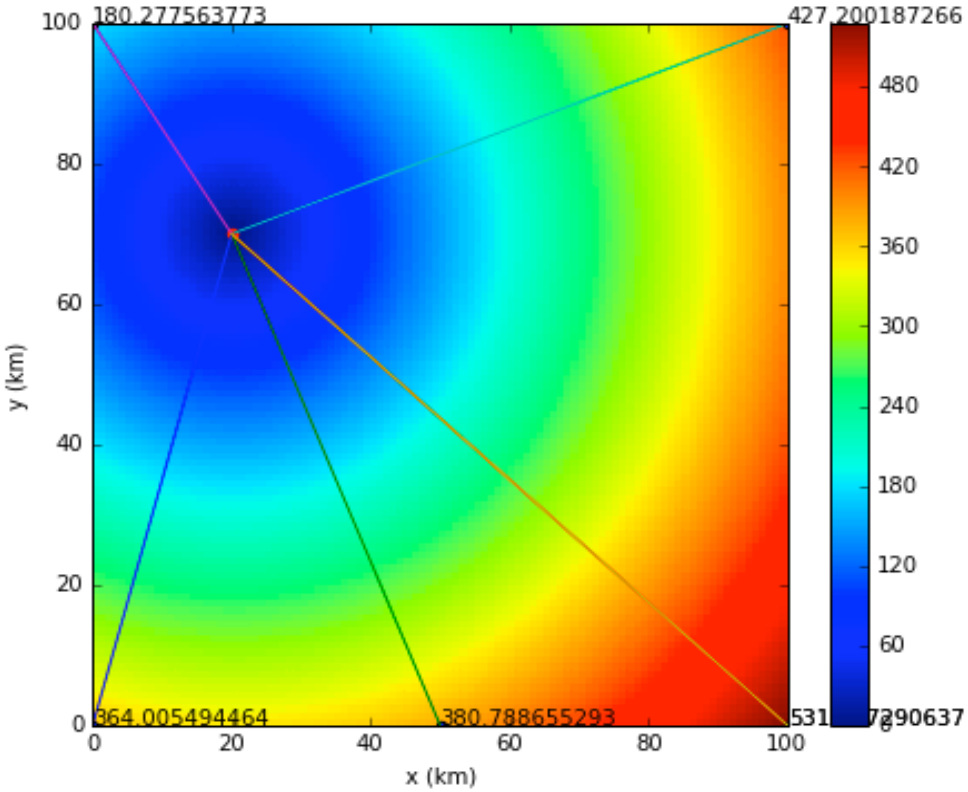


Aula passada

Um problema real:

- Temos uma fonte (sismo, tsunami, emissor GPS, etc) que gera ondas que se propagam e que são registadas em estações.
- Nas estações mais longe da fonte, as chegadas das ondas são registadas mais tarde.

- Queremos localizar a fonte (sismo, tsunami, emissor GPS)
- Conhecendo os tempos de chegada das ondas às estações.
- E sabendo a velocidade de propagação das ondas (caso simples, c=contante)



```python
import matplotlib.pyplot as plt
import numpy as np

#%%
plt.rcParams['figure.figsize'] = 10, 6

################## Inicializar

#%% Setup

# Limites do domínio espacial
xmin=0.
xmax=100e3
ymin=0.
ymax=100e3

# Espaçamento entre pontos
dx=1e3
dy=1e3

# Pontos em x e y
x=np.arange(xmin,xmax+dx,dx)
y=np.arange(ymin,ymax+dy,dy)

# Nr de pontos
nx=len(x)
ny=len(y)

# velocidade de propagação do sinal
c=200
```

```
# Matrizes 2D de x, y, t
xx=np.zeros([nx,ny])
yy=np.zeros([nx,ny])
tt=np.zeros([nx,ny])

for ix in range(nx):
    for iy in range(ny):
        xx[ix,iy]=ix*dx
        yy[ix,iy]=iy*dy

xr=np.array([xmin,xmax])/1e3      # x range
yr=np.array([ymin,ymax])/1e3      # y range

#%% Estações e fonte

# Coordenadas das estações
xE=np.array([0, 50, 100, 100,0, 100])*1e3
yE=np.array([0, 0, 0, 100, 100,0])*1e3

# Coordenadas da fonte
xF=20e3
yF=70e3;

# Propagação do sinal
distE=np.sqrt((xE-xF)**2+(yE-yF)**2)    # Distância Fonte-Estações
tE=distE/c;                             # Tempo de chegada da onda às estações

# Tempo de chegada da onda a cada ponto da grelha
dd=np.sqrt((xx-xF)**2+(yy-yF)**2)       # Distância
tt=dd/c                                 # Tempo
```

```
#%% Plot

plt.rcParams['figure.figsize'] = 7,6
plt.close()

plt.pcolor(xx/1e3,yy/1e3,tt)           # matriz dos tempos de propagação
plt.colorbar()
plt.scatter(xF/1e3,yF/1e3, color='r');      # fonte

# estações
for iE in range(len(xE)):
    plt.plot([xE[iE]/1000, xF/1000], [yE[iE]/1000, yF/1000])
    plt.scatter(xE[iE]/1e3,yE[iE]/1e3)
    plt.text(xE[iE]/1e3,yE[iE]/1e3, str(tE[iE]))

plt.xlabel('x (km)')
plt.ylabel('y (km)')
plt.xlim(xr)
plt.ylim(yr)

plt.savefig('p1-TempoPropagacao.png')
```

Função de custo

- J: função de custo
- nE: nr de estações
- xE, yE: coordenadas das estações
- tE: observações (tempo de chegada das ondas às estações)
- Na iteração m.

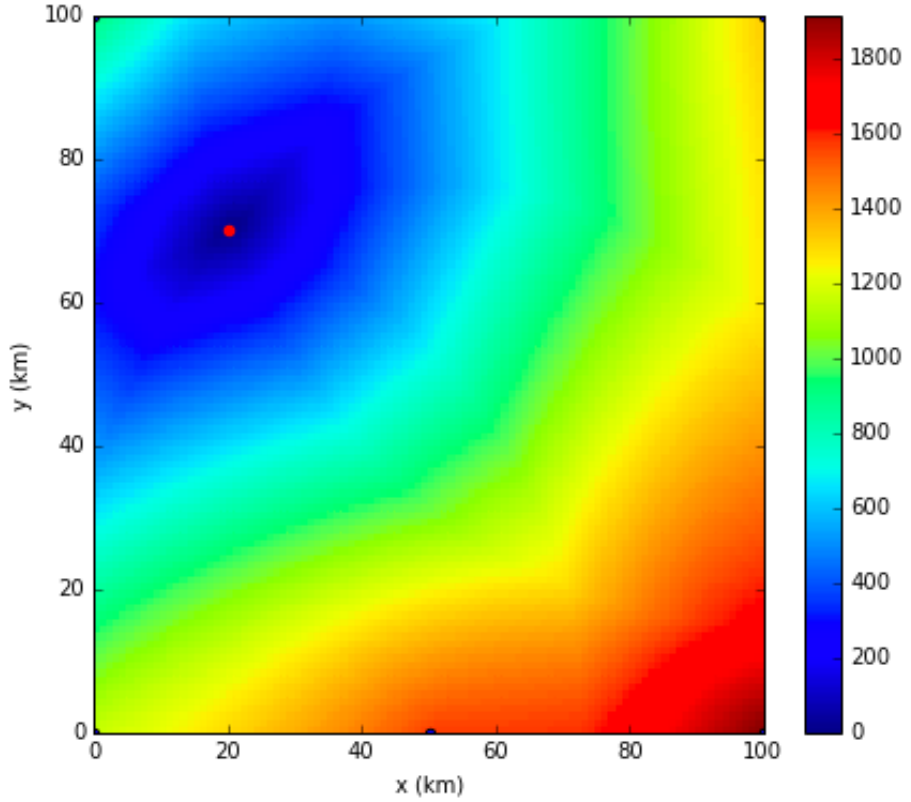
$$J_m = \sum_{k=1}^{nE} |t_{m,k} - t_{E,k}|$$

$$= \sum_{k=1}^{nE} \left| \frac{d_{m,k}}{c} - t_{E,k} \right|$$

$$= \sum_{k=1}^{nE} \left| \frac{\sqrt{(x_m - x_{E,k})^2 + (y_m - y_{E,k})^2}}{c} - t_{E,k} \right|$$

```
c=200

def custo(X,Y,xE,yE,tE,c):
    dist=np.sqrt((X-xE)**2+(Y-yE)**2);
    erro=dist/c-tE;
    custo=np.sum(np.abs(erro))
    return custo
```

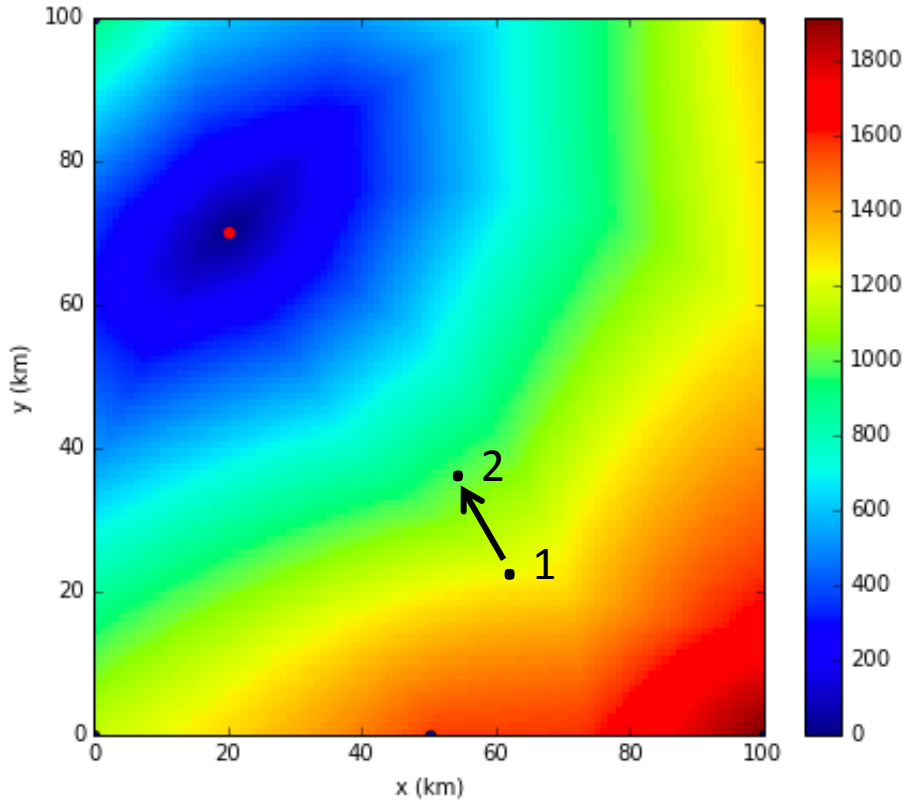
```
#%% Função custo
def custo(X,Y,xE,yE,tE,c):
    dist=np.sqrt((X-xE)**2+(Y-yE)**2);       # distância entre a estação e o ponto X,Y
    erro=dist/c-tE;                          # diferença entre os tempos observados e os tempos cal
    custo=np.sum(np.abs(erro))               # soma das diferenças de tempos para todas as estações
    return custo

cc=np.zeros([nx,ny])        # inicializar a grelha de custos

# calcular a função de custo para todos os pontos
for ix in range(nx):
    for iy in range(ny):
        cc[ix,iy]=custo(xx[ix,iy],yy[ix,iy],xE,yE,tE,c)

plt.rcParams['figure.figsize'] = 7,6
plt.close()

plt.pcolor(xx/1e3,yy/1e3,cc)                 # f
plt.colorbar()
plt.scatter(xF/1e3,yF/1e3, color='r');     # fonte
plt.scatter(xE/1e3,yE/1e3)                 # estaçã

plt.xlabel('x (km)')
plt.ylabel('y (km)')
plt.xlim(xr)
plt.ylim(yr)

plt.savefig('p2-Custo.png')
```



Caso muito simples: só há um mínimo!

Algoritmo

- Começar num local aleatório: first guess.
- Pesquisar a vizinhança desse local aleatoriamente (Monte Carlo).
- Rejeitar saltos que aumentem o custo.
- Continuar até estar não conseguir melhorar.



Dificuldades

- O salto aleatório é feito com:

```
rr=np.random.rand()-0.5      # nr aleatório entre -.5 e .5
XI=X+xstep*rr;               # salto aleatório até xstep/2 de distância
```

- No início convém que xstep seja “grande” para nos aproximarmos rapidamente da solução.
- Quando estamos próximos da solução precisamos que xstep seja cada vez menor, sob pena de ser impossível convergir.

Algoritmo de salto aleatório “Downslope MC”

- Convém ter dois ciclos, um dentro do outro:
- Um ciclo externo:
  - Em que vamos reduzindo a dimensão do salto aleatório até um valor comparável com o erro aceitável nos parâmetros a estimar.
  - Em cada passo do ciclo externo fazemos:

```
xstep=xstep*COOL
ystep=ystep*COOL
```

  - COOL é um número <1 (0.5 no exemplo). A designação corresponde a uma analogia com um processo de arrefecimento que será explorada mais tarde (simulated annealing).
- Um ciclo interno de muitos saltos aleatórios.

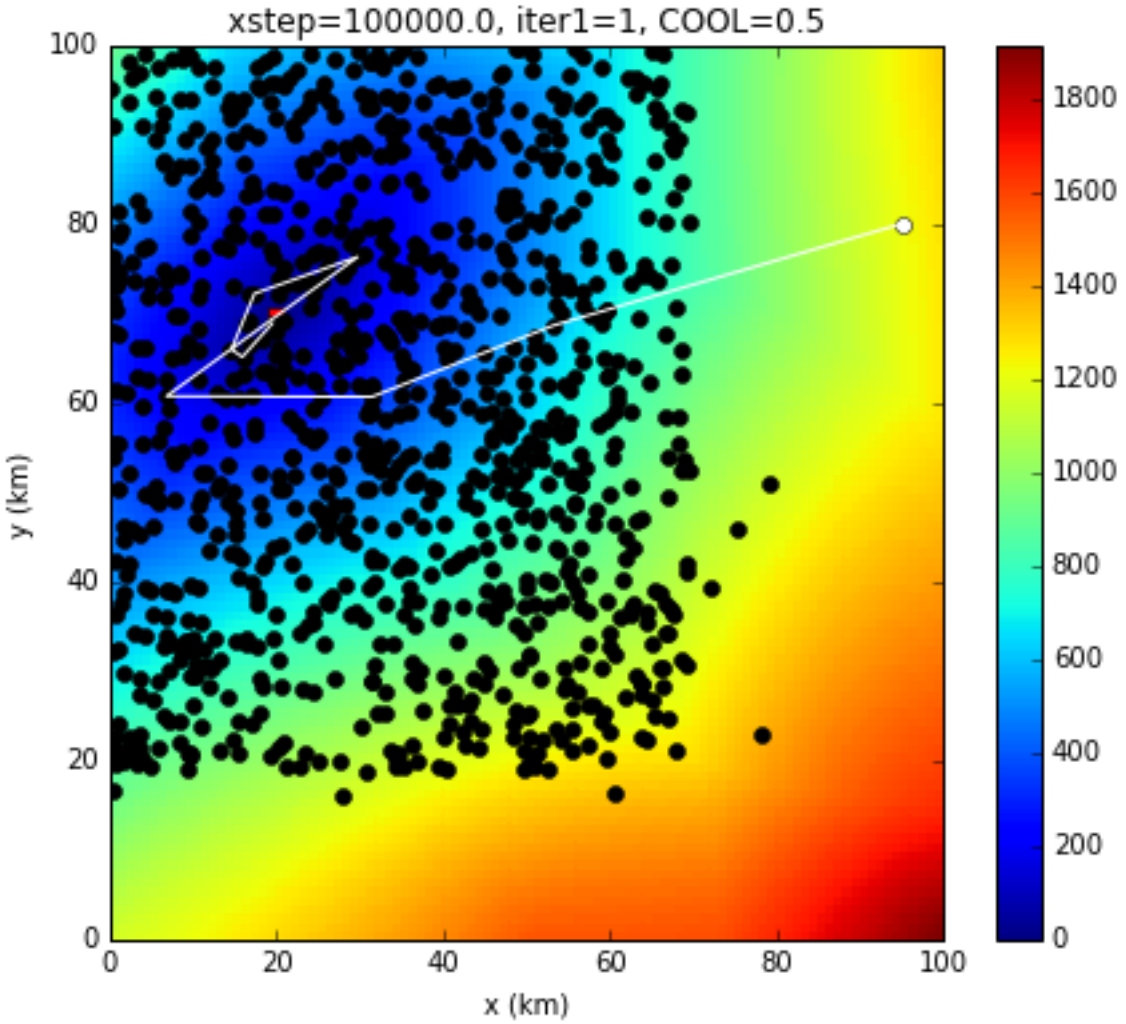
```
#%% ######################## Optimização

maxITER1=100.       # nr máximo de iterações, ciclo externo (1)
maxPERT2=1e3        # nr máximo de perturbações, ciclo interno (2)
maxHIT2=10000.      # nr máximo de soluções boas encontradas dentro do ciclo interior (2)

# vectores para guardar os resultados selecionados (a melhor posição) de cada iteração (ciclo ext
xSEL=np.nan*np.ones([maxITER1])
ySEL=np.nan*np.ones([maxITER1])

# salto
xstep0=100e3        # tamanho do salto inicial (x)
ystep0=100e3        # tamanho do salto inicial (y)
COOL=0.5            # percentagem de redução do tamanho do salto em cada iteração

# critério de paragem
minstep=1.          # tamanho mínimo do salto

#%% iniciar a optimização

# Posição e custo inicial
X=xmin+(xmax-xmin)*np.random.rand();   # Posição em x inicial (aleatória)
Y=ymin+(ymax-ymin)*np.random.rand();   # Posição em y inicial (aleatória)
J=custo(X,Y,xE,yE,tE,c);               # Função custo para a posição inicial

xSEL[0]=X
ySEL[0]=Y

xstep=xstep0
ystep=ystep0

iter1=0               # índice do ciclo exterior (1)
kHIT1=0               # nr total de boas soluções encontradas
```

```
while iter1<maxITER1 and xstep>minstep:          # condições para estarmos dentro do ciclo exter
    iter1+=1        # incrementar o índice do ciclo externo (1)
    iPERT2=0        # colocar a zero o índice do ciclo interno (2)
    nHIT2=0;        # colocar a zero o nr de boas soluções encontradas dentro do ciclo interno (2)
    pertX=[]        # construir um vector para guardar as posições perturbadas, x
    pertY=[]        # construir um vector para guardar as posições perturbadas, y

    while iPERT2<maxPERT2 and nHIT2<maxHIT2:     # condições para estarmos dentro do ciclo 2

        # salto (nova posição) em x
        rr=np.random.rand()-0.5     # nr aleatório entre -.5 e .5
        XI=X+xstep*rr;              # novo x varia em torno de X por um valor aleatório, cujo valo
        while XI<xmin or XI>xmax:  # aceitar apenas valores dentro do domínio, se fora do domínio
            rr=np.random.rand()-0.5
            XI=X+xstep*rr

        # salto (nova posição) em y
        rr=np.random.rand()-0.5        # nr aleatório entre -.5 e .5
        YI=Y+ystep*rr;                 # novo y varia em torno de Y por um valor aleatório, cujo va
        while YI<ymin or YI>ymax:      # aceitar apenas valores dentro do domínio, se fora do domí
            rr=np.random.rand()-0.5
            YI=Y+ystep*rr;

        pertX.append(XI)        # guardar as posições perturbadas, XI
        pertY.append(YI)        # guardar as posições perturbadas, YI

        JI=custo(XI,YI,xE,yE,tE,c);    # calcular a função custo para a nova posição perturbada (a
        dJ=JI-J;                       # diferença entre a função custo de referiencia (J) e a cal
        if dJ<0:                       # se a nova função custo (JI) tiver um valor mais baixo do
            J=JI                    # actualizar a função custo de referência
            X=XI                    # actualizar a posição x de referência
            Y=YI                    # actualizar a posição y de referência
            nHIT2=nHIT2+1         # actualizar o nr de boas soluções no ciclo interior (2))
            kHIT1=kHIT1+1         # actualizar o nr de boas soluções total (1)
            xSEL[kHIT1]=X           # guardar a posição x seleccionada
            ySEL[kHIT1]=Y           # guardar a posição y seleccionada

        iPERT2+=1;          # incrementar o índice do ciclo interno (2)
```

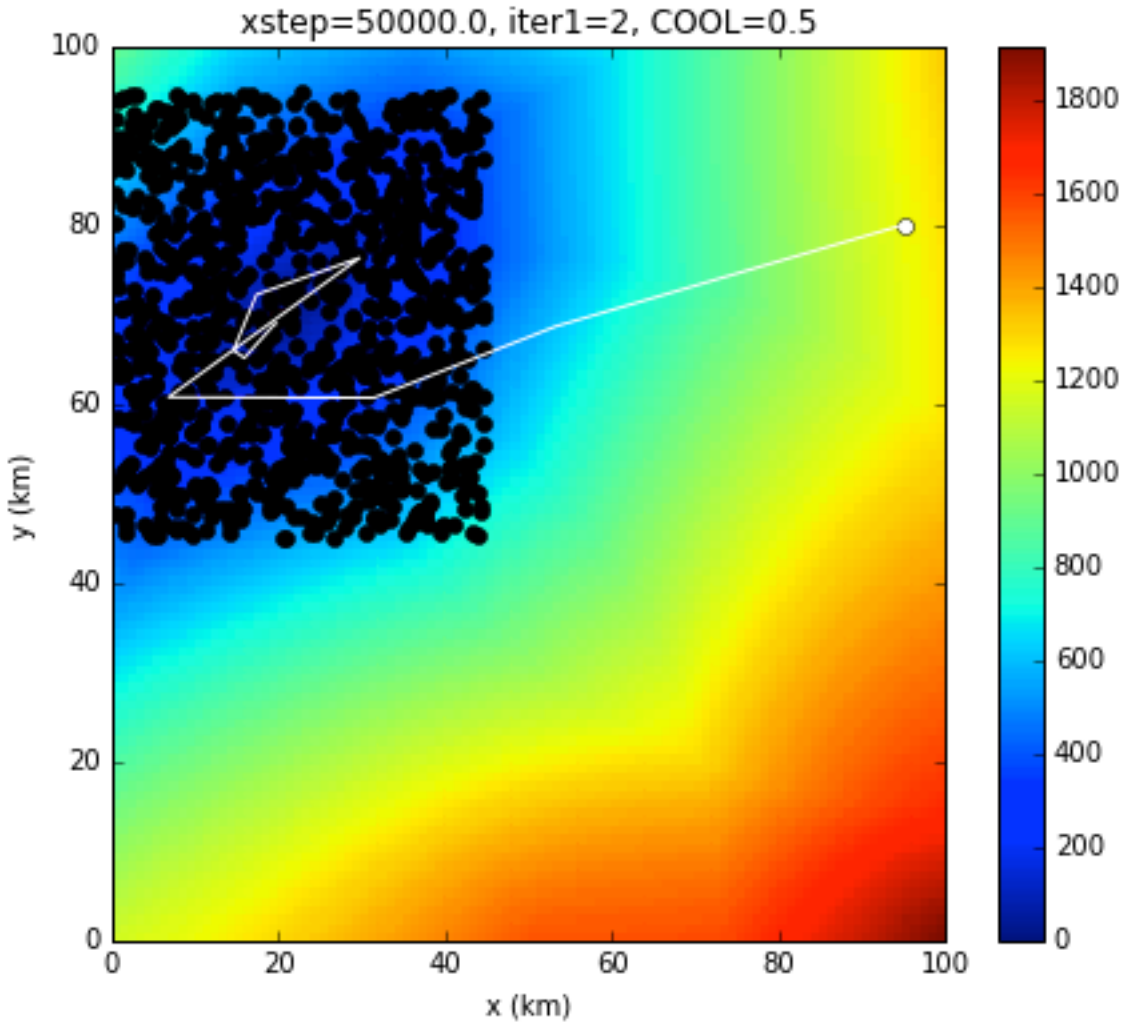
```
    print('iter1='+str(iter1) + ', kHIT1='+str(kHIT1) + ', xstep='+str(xstep) + \
       ', iPERT2='+str(iPERT2) + ', nHIT2='+str(nHIT2)+ ', J='+str(J)+', DJ='+str(dJ)+ \
       ' XI='+str(XI)+', YI='+str(YI) + ' \n')


if iter1 in [1, 2, 5, 10, 100, 1000]:
  plt.rcParams['figure.figsize'] = 7,6
  plt.close()

  plt.pcolor(xx/1e3,yy/1e3,cc)
  plt.plot(xF/1e3, yF/1e3, 's', markerfacecolor='red')
  plt.colorbar()
  plt.plot(np.array(pertX)/1e3, np.array(pertY)/1e3, 'o', markerfacecolor='black')

  plt.plot(xSEL[0]/1e3, ySEL[0]/1e3, 'o', markerfacecolor='white');
  plt.plot(xSEL[:kHIT1]/1e3, ySEL[:kHIT1]/1e3, 'white');
  plt.xlabel('x (km)')
  plt.ylabel('y (km)')
  plt.xlim(xr)
  plt.ylim(yr)
  plt.title('xstep='+str(xstep) +', iter1='+str(iter1)+', COOL='+str(COOL))
  plt.savefig('p4-iter1_'+str(iter1)+'.png')

xstep=xstep*COOL;       # definir a amplitude do salto na próxima iteração (ciclo externo, 1)
ystep=ystep*COOL;       # definir a amplitude do salto na próxima iteração (ciclo externo, 1)
```

```
#%% Plot

plt.rcParams['figure.figsize'] = 7,6
plt.close()

plt.plot(xF/1e3, yF/1e3, 's', markerfacecolor='red');
plt.pcolor(xx/1e3,yy/1e3,cc)
plt.colorbar()
plt.plot(xSEL[0]/1e3, ySEL[0]/1e3, 'o', markerfacecolor='white');
plt.plot(xSEL[:kHIT1]/1e3, ySEL[:kHIT1]/1e3, 'white');
plt.title('Final, xstep0='+str(xstep0) +', niter='+str(iter1)+', COOL='+str(COOL))

plt.xlabel('x (km)')
plt.ylabel('y (km)')
plt.xlim(xr)
plt.ylim(yr)

plt.savefig('p3-Final.png')
```

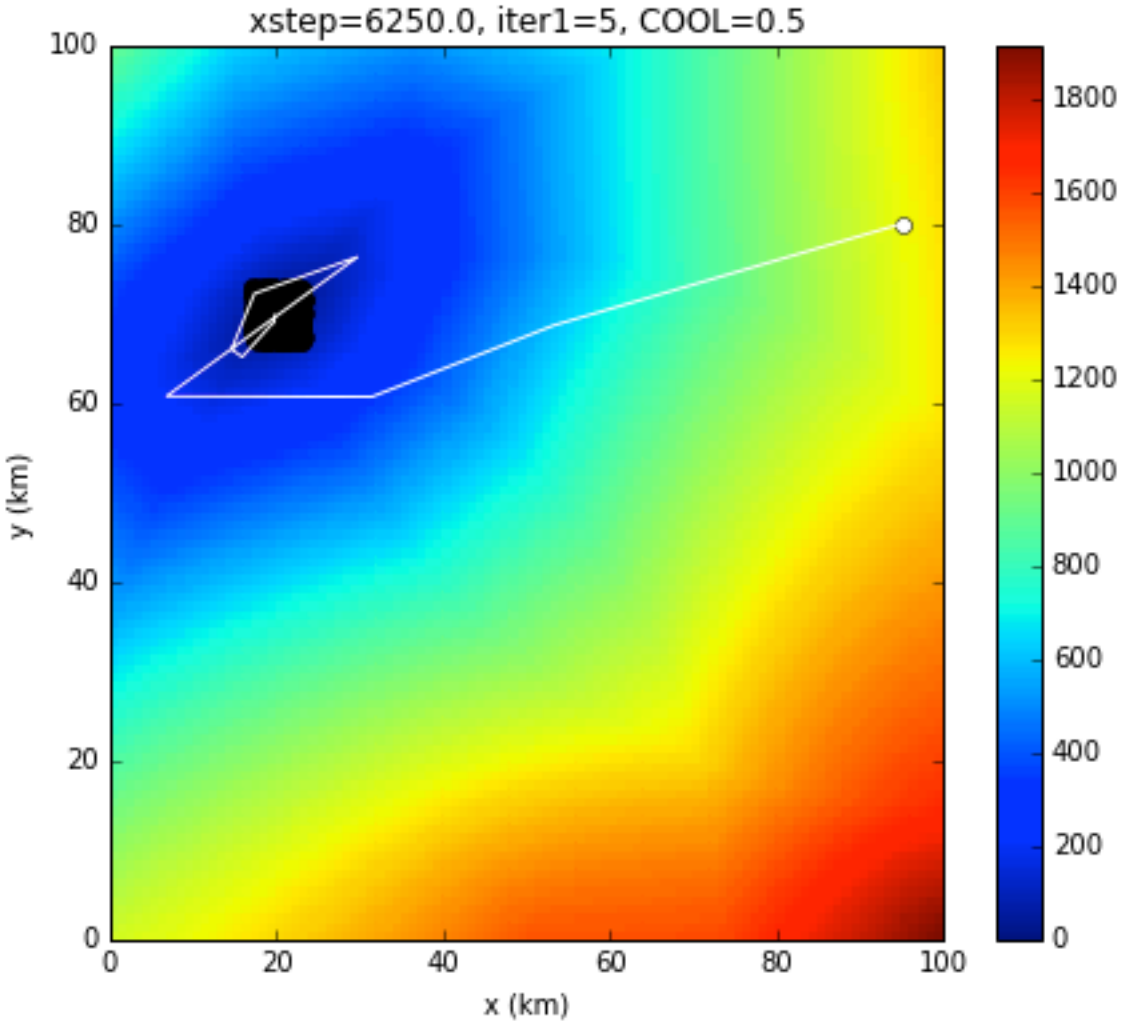
Resultado


xstep=100000.0, iter1=1, COOL=0.5
y (km)
x (km)
0
20
40
60
80
100
0
200
400
600
800
1000
1200
1400
1600
1800

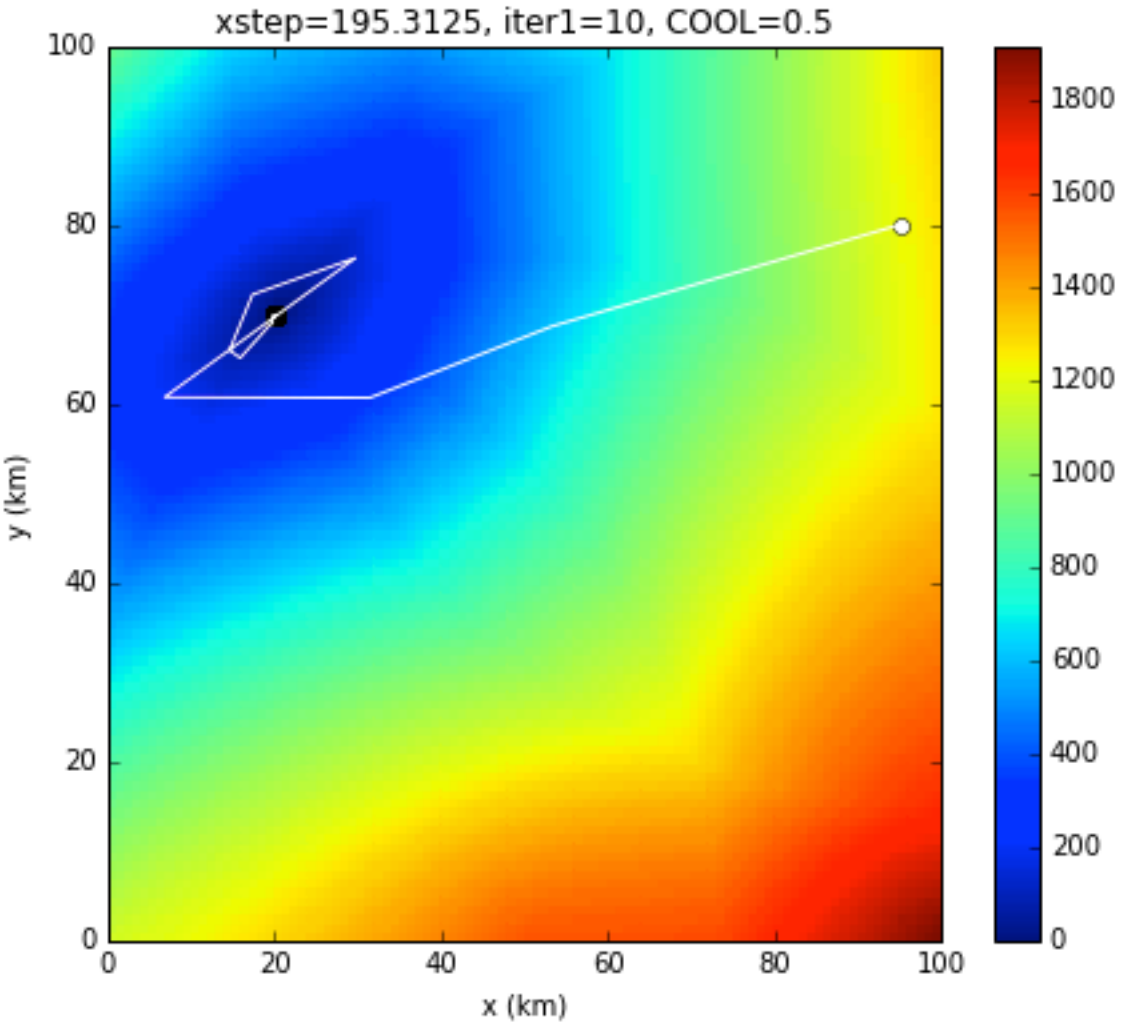
Resultado


xstep=50000.0, iter1=2, COOL=0.5
y (km)
x (km)
0
20
40
60
80
100
0
200
400
600
800
1000
1200
1400
1600
1800

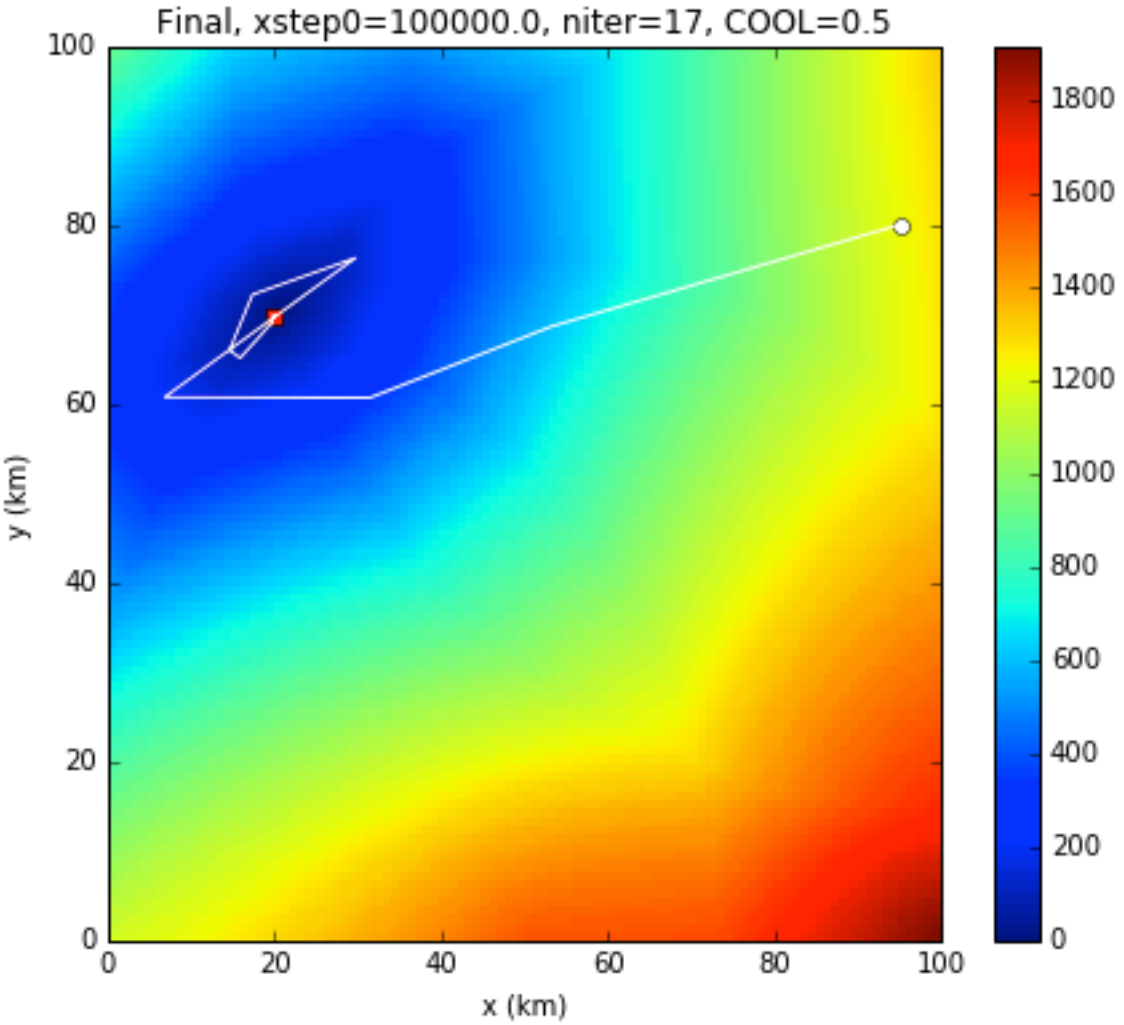
Resultado


xstep=6250.0, iter1=5, COOL=0.5
y (km)
x (km)
0
20
40
60
80
100
0
200
400
600
800
1000
1200
1400
1600
1800

Resultado


xstep=195.3125, iter1=10, COOL=0.5
y (km)
x (km)
0
20
40
60
80
100
0
200
400
600
800
1000
1200
1400
1600
1800

Resultado


Final, xstep0=100000.0, niter=17, COOL=0.5
y (km)
x (km)
0
20
40
60
80
100
0
200
400
600
800
1000
1200
1400
1600
1800