Ciências ULisboa

Modelação Numérica
Aula 7

Otimização

O problema da otimização

Muitas experiências passam pela realização de um conjunto de medidas (**observações**) que são utilizadas para **aferir parâmetros** de um modelo, com um certo nível de erro. Em geral, não existe uma solução unívoca que permita determinar esses parâmetros, por exemplo porque existe **sobre** ou **sub-determinação**, e é preciso estabelecer uma metodologia "**ótima**" para encontrar o "**melhor**" conjunto de parâmetros que se ajusta aos dados.

Seja $\vec{o}$ o (vetor) conjunto de observações, e $\vec{p}$ o vetor dos parâmetros, tipicamente com dimensão diferente, podemos escrever (com erro ε):

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

Em que a função M representa o modelo. Se o modelo fosse linear seria

$$\vec{o} = L\vec{p} + \varepsilon$$

sendo L uma matriz (dos coeficientes do modelo).

Problema linear sobredeterminado

O caso mais simples é o da regressão linear:

$$y = ax + b$$

onde a, b são os parâmetros a calcular, dados $y_k, x_k\ (k = 0 \ldots N-1), N \geq 2$. Se $N = 2$, o problema tem uma única solução com erro nulo, se $N > 2$ não existe solução exata e os parâmetros a, b são calculados com a condição de **minimização do erro médio quadrático**. **Tratando-se de um modelo linear, podemos formular o problema na forma de um produto matricial**

$$\vec{o} = L\vec{p} + \varepsilon$$

i.e.

$$\begin{bmatrix} y_1 \\ \ldots \\ y_N \end{bmatrix} = \begin{bmatrix} x_1 & 1 \\ \ldots & \ldots \\ x_N & 1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} + \begin{bmatrix} \varepsilon_1 \\ \ldots \\ \varepsilon_N \end{bmatrix}$$

e trata-se de um **problema sobredeterminado**: há mais equações (N) que incógnitas (2).

Regressão linear (exemplo com constrangimento)

A função 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. Por vezes, precisamos de impor condições aos parâmetros, por exemplo, podemos querer fazer o ajuste:

$$y = ax$$

i.e. impor $b = 0$. Nesse caso precisamos de resolver o problema explicitamente. Este problema tem solução analítica simples. Para um dado valor de a, o erro médio quadrático é:

$$\overline{\varepsilon^2} = \frac{1}{N}\sum_{k=1}^{N}(y_k - ax_k)^2$$

e depende de a. A condição de mínimo é dada por:

$$\boldsymbol{y = ax}$$

$$\frac{\partial \overline{\varepsilon^2}}{\partial a} = 0 \Longrightarrow \frac{\partial}{\partial a}\left[\frac{1}{N}\sum_{k=1}^{N} y_k^2 + a^2 x_k^2 - 2a x_k y_k\right] = 0$$

$$\sum_{k=1}^{N} 2a x_k^2 - 2 x_k y_k = 0 \Longrightarrow a = \frac{\sum_k x_k y_k}{\sum_k x_k^2}$$

Caso geral: conceito de função custo

Em geral não é possível obter **diretamente** uma solução ótima, sendo necessário proceder iterativamente.

O método iterativo requer:

- ✓ Um método para obter potenciais soluções, e de as aceitar se for caso disso

  Função de custo,e.g.: $J = |\vec{o} - M(\vec{p})|$

  Aceitar soluções que (*pelo menos em média*) reduzam J

- ✓ Um critério de paragem

  Número de iterações, valor atingido pela função de custo, falta de progresso em J ou em $\vec{p}$

Problema

Localizar uma fonte (sísmica, tsunami,som) conhecendo os tempos de chegada em várias estações e a velocidade de propagação.

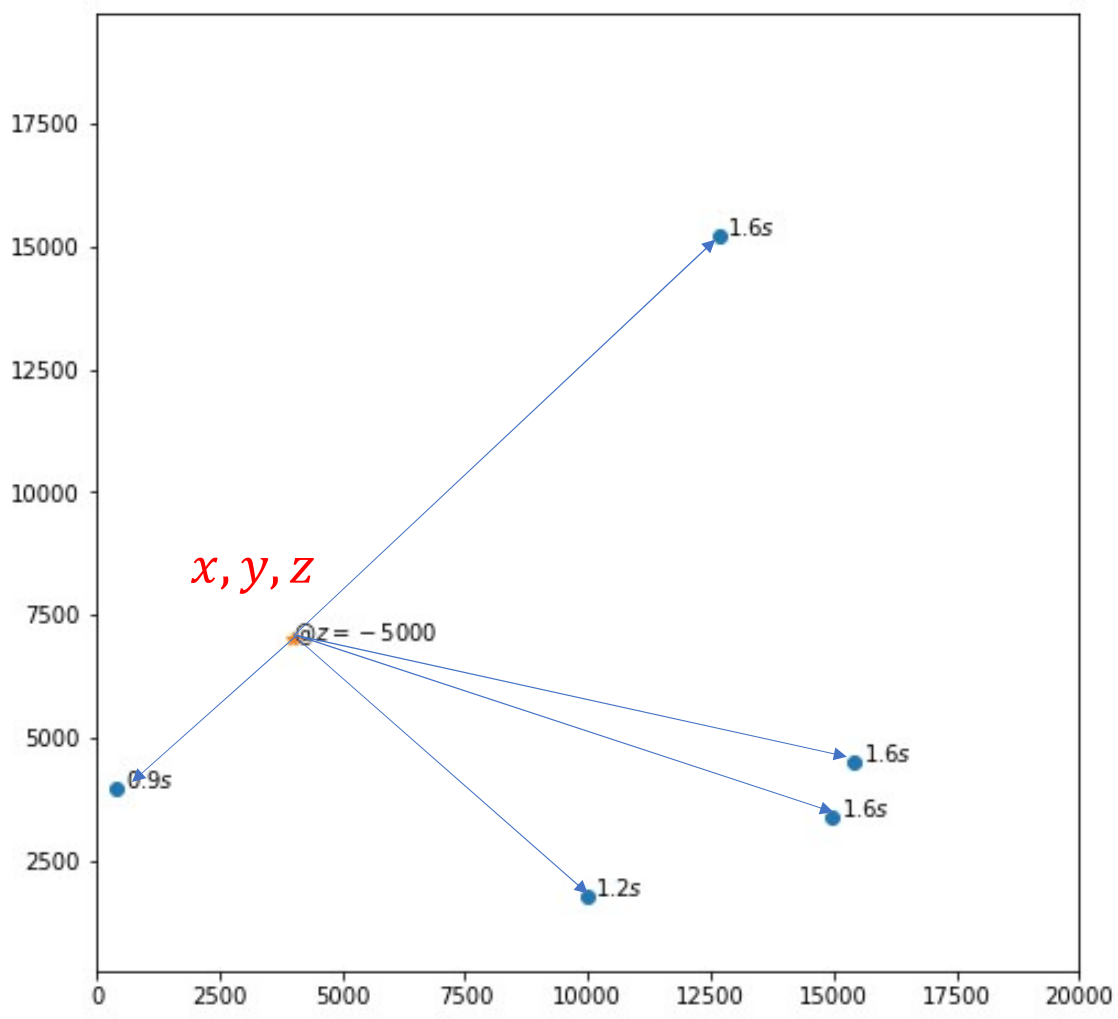
Caso simples com $c = const$

Fonte subterrânea (z=-5000m)

Estações em z=0

Dados: Localização das estações, tempos de chegada

Erro depende de: erro de dados, método, distribuição das estações em relação à fonte.



Função de custo

nE: nº de estações

xE,yE,zE: localização das estações

tE: observações (tempos de chegada)

Na iteração m

$$J_m = \sum_{k=0}^{nE-1} \left| t_{m,k} - t_{E,k} \right|$$

$$= \sum_{k=0}^{nE-1} \left| \frac{\sqrt{\left(x_m - x_{E,k}\right)^2 + \left(y_m - y_{E,k}\right)^2 + \left(z_m - z_{E,k}\right)^2}}{c} - t_{E,k} \right|$$

Estimativa
Na iteração m

Observação

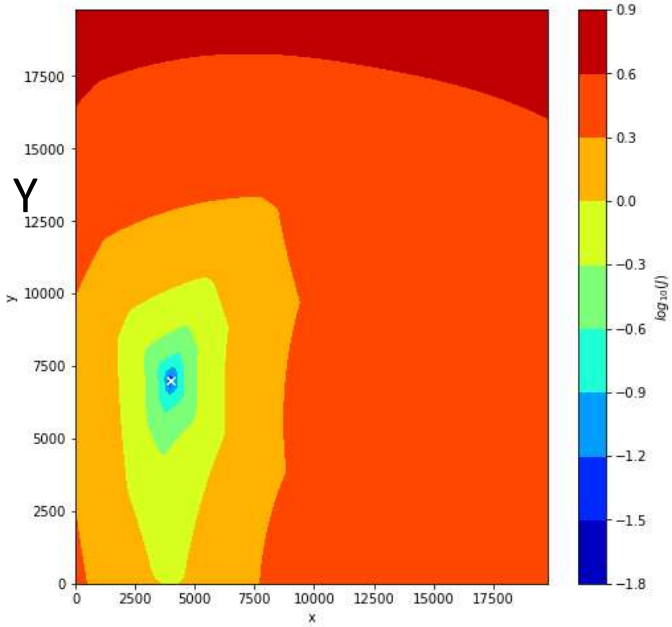
x

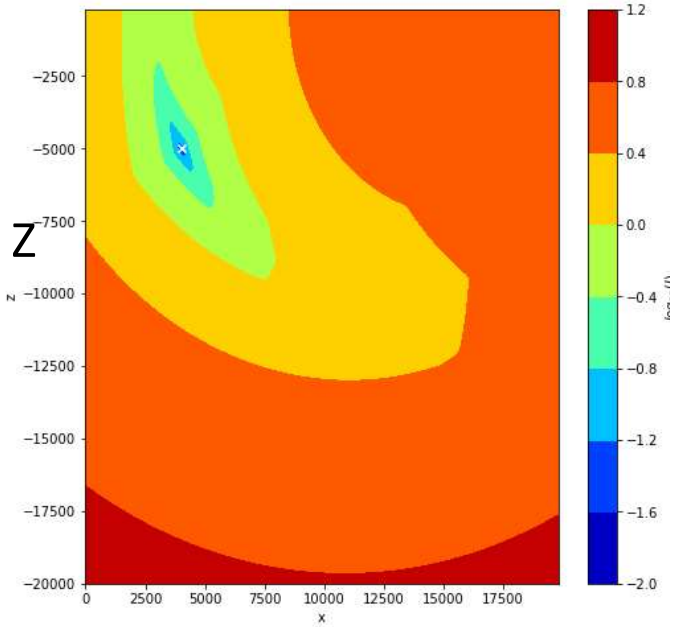
Geometria da função de custo

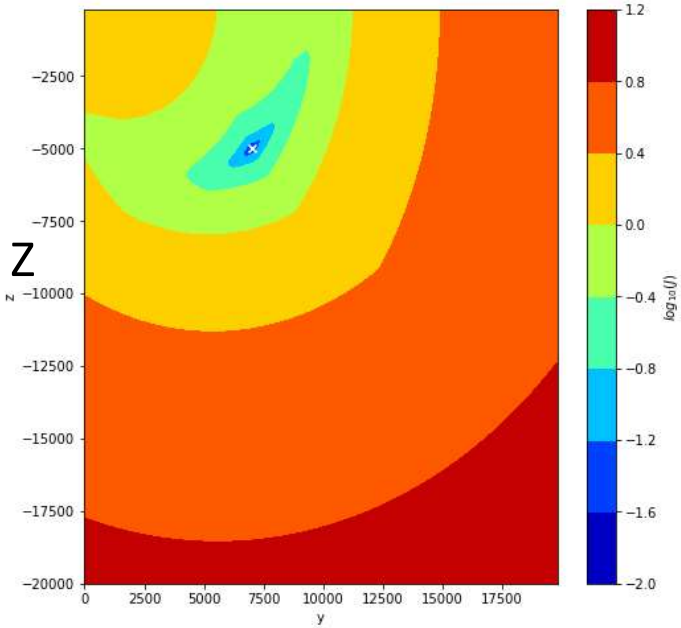
$$J = \sum_{k=0}^{nE-1} \left| \frac{\sqrt{\left(x_m - x_{E,k}\right)^2 + \left(y_m - y_{E,k}\right)^2 + \left(z_m - z_{E,k}\right)^2}}{c} - t_{E,k} \right|$$

Caso extremamente simples: só existe 1 mínimo, limites do domínio são conhecidos.

$\log_{10}(J)$





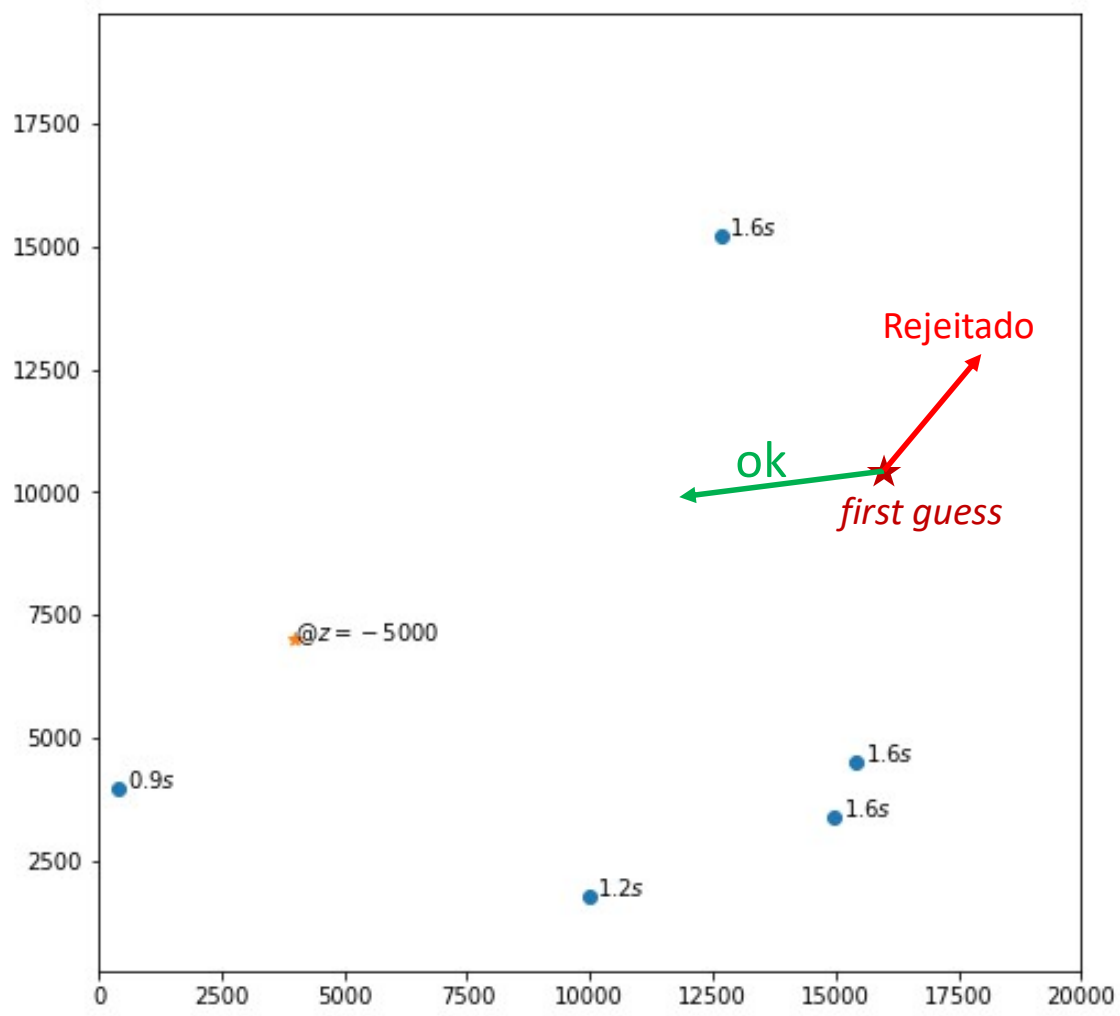


Algoritmo

Começar num local aleatório: *first guess*.

Pesquisar a vizinhança desse local aleatóriamente (Monte Carlo). **Rejeitar** saltos que aumentem o custo.

Continuar até não conseguir melhorar.



Dificuldades

O salto aleatório é feito com:

```
rr=np.random.rand()-0.5 #Número aleatório ∈ [−0.5, +0.5]
x=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: a probabilidade de cair mais perto da solução tende para zero.

Algoritmo de salto aleatório “downslope”

Convém, portanto ter dois ciclos embebidos:

Um ciclo externo em que vamos reduzindo a dimensão do salto aleatório até um valor comparável com o erro aceitável nos parâmetros a estimar (X,Y).

Em cada passo do ciclo externo fazemos

```
xstep=xstep*COOL;
```

`COOL` é um número <1 (0.9 no exemplo). A designação corresponde a uma analogia com um processo de arrefecimento **(Simulated annealing).**

Um ciclo interno de muitos saltos aleatórios.

Localização de mínimo de função de custo em ndim dimensões

```
def annealD(vmin,vmax,Jmin,minvstep,maxITER,maxPERT,COOL,kappa,T,outITER):
    path=[]
    sh=np.shape(vmin)
    ndim=sh[0] #dimensões da função de custo
    vstep=2*(vmax-vmin) #dobro da extensão do domínio
    minxstep=minvstep[0] #salto mínimo
    xstep=vstep[0] #salto inicial em x
    rrr=np.random.sample(sh) #ndim numeros aleatórios
    V=vmin+(vmax-vmin)*rrr #first guess
    print(V) #first guess
    VI=np.zeros(sh) #destino do salto
    J=cost(V) #custo da first guess
    iTER=0;nHIT=1;kPERT=0; #inicialização de contadores
    print(maxITER,Jmin,minxstep)
```

```
    while (iTER<maxITER and J>Jmin and xstep>minxstep): #ciclo externo (step fixo)
        nHIT=0;iP=0;
        while iP<maxPERT: #ciclo interno
            kPERT=kPERT+1;
            for idim in range(ndim): #percorrer dimensões
                rr=np.random.rand()-0.5
                VI[idim]=V[idim]+vstep[idim]*rr #salto aleatório
                while VI[idim]<vmin[idim] or VI[idim]>vmax[idim]: #rejeitar domínio
                    rr=np.random.rand()-0.5;
                    VI[idim]=V[idim]+vstep[idim]*rr;
            JI=cost(VI) #custo depois do salto
            if JI<J: #aceitar se tem menor custo
                J=np.copy(JI) #atualiza custo
                V=np.copy(VI) #atualiza solução
                path.append(V) #guarda trajetória (multidimensional)
                nHIT=nHIT+1
            iP=iP+1;
        iTER=iTER+1;
        T=T*COOL #"arrefecimento"
        vstep=np.maximum(minvstep,vstep*np.exp(-kappa/T)) #novo salto (Boltzmann)
return ndim,V,iTER,path
```

A função annealD é genérica

Em cada caso será necessário:

Escrever a função **cost** (função de custo), contendo os parâmetros a otimizar.

Escolher as **observações** (usadas na função cost)

Definir o **domínio** da solução (min e maximo de cada parâmetro)

Definir as **constantes** do algoritmo (Número de iterações máximo, erro desejado, taxa de arerefecimento, etc.)

Executar

Analisar os resultados.

Localização da fonte

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

def iniSeismic(nE=4,iseed=10): #solução e observações (sem ruído)
    cs=8000 #velocidade da onda sismica
    xS=4000;yS=7000;zS=-5000 #solução:posição da fonte
    xmin=0;xmax=20000;ymin=0;ymax=20000;zmin=-20000;zmax=0000 #domínio
    vmin=np.array([xmin,ymin,zmin])
    vmax=np.array([xmax,ymax,zmax])
    np.random.seed(iseed) #observações em pontos aleatórios (controlados)
    xE=xmin+(xmax-xmin)*np.random.sample(nE)
    yE=ymin+(ymax-ymin)*np.random.sample(nE)
    zE=np.zeros(xE.shape)
    distE=np.sqrt((xS-xE)**2+(yS-yE)**2+(zS-zE)**2)
    tE=distE/cs #observações nas estações
return xE,yE,zE,tE,cs,vmin,vmax
```

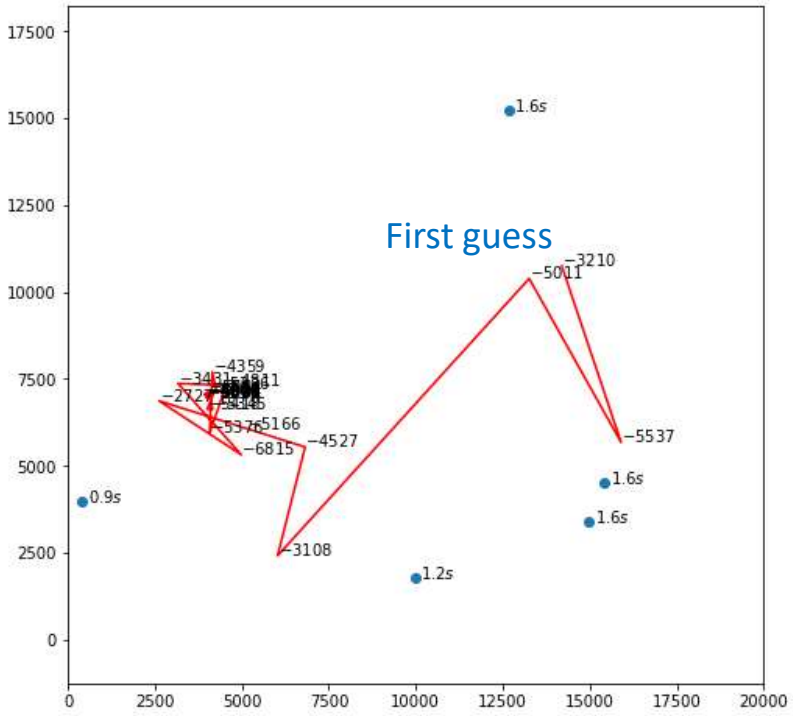
Função de custo

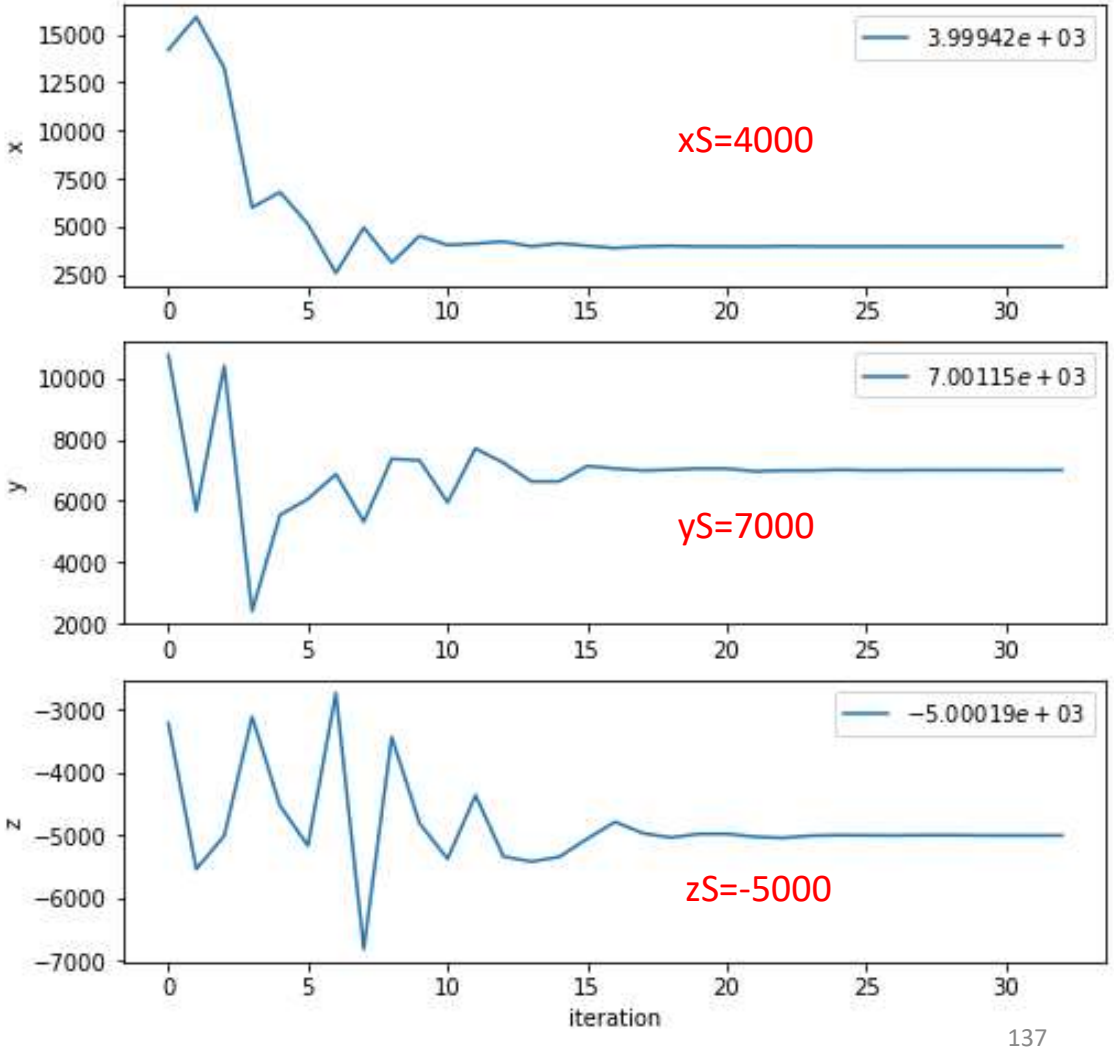
```
def cost(V):
    global xE,yE,zE,tE #observations
    global cs #constants

    X=V[0];Y=V[1];Z=V[2] #parameters to optimize (3 dimensões)
    dist=np.sqrt((X-xE)**2+(Y-yE)**2+(Z-zE)**2)
    erro=dist/cs-tE
    custo=np.sum(abs(erro))
    return custo
```

Main

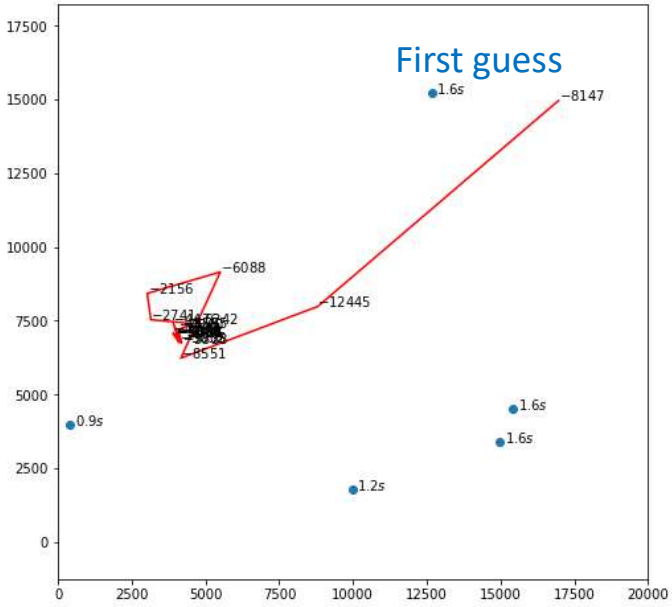
```
xE,yE,zE,tE,cs,vmin,vmax=iniSeismic(nE=5)
Jmin=-10. #como J é sempre positivo este critério está desligado
minvstep=(vmax-vmin)/1000 #salto mínimo (multidimensional)
maxITER=1000 #Ciclo externo
maxPERT=1000 #Ciclo interno
COOL=0.9 #fator de arrefecimento
kappa=np.float64(0.1) #parâmetro de Boltzmann
T=10. #"Temperatura" de Boltzmann

n,V,iTER,path=annealD(vmin,vmax,Jmin,minvstep,maxITER,\
                      maxPERT,COOL,kappa,T,outITER)
```

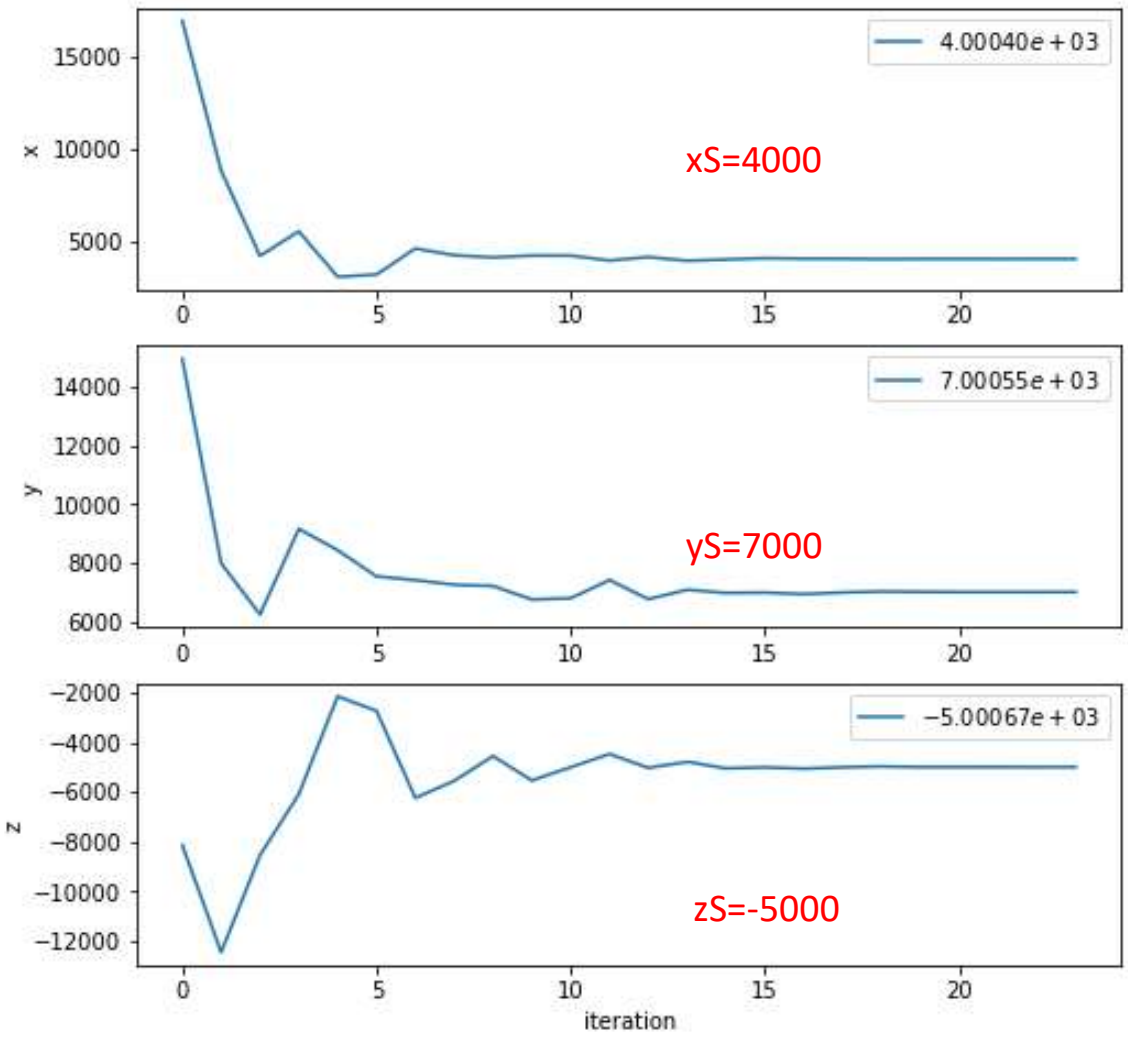
Evolução da solução





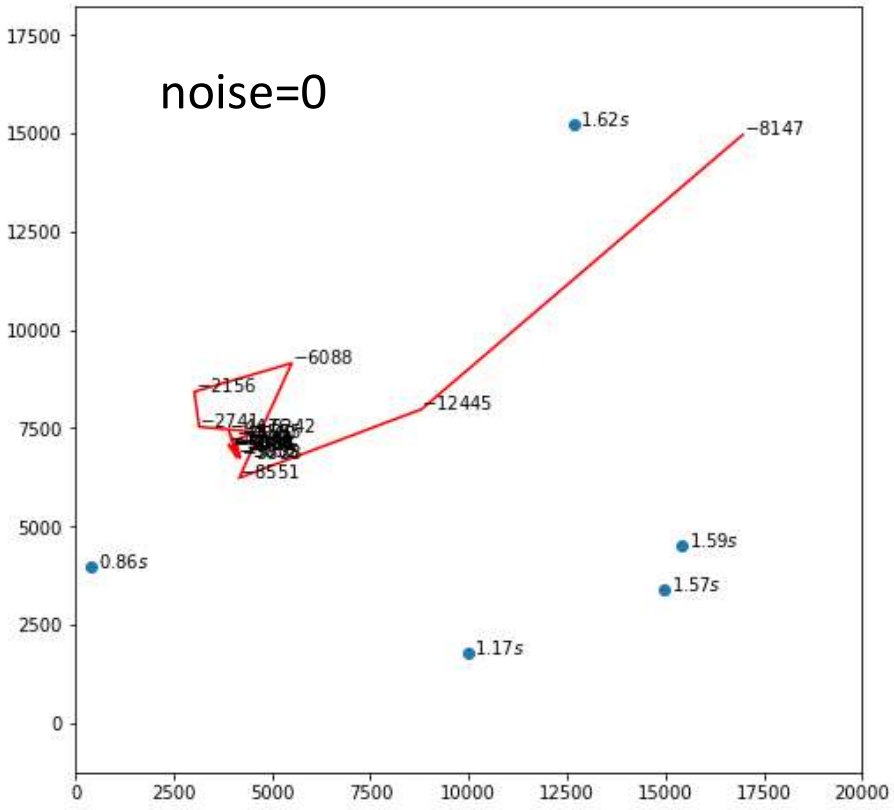
Evolução da solução (outra realização)

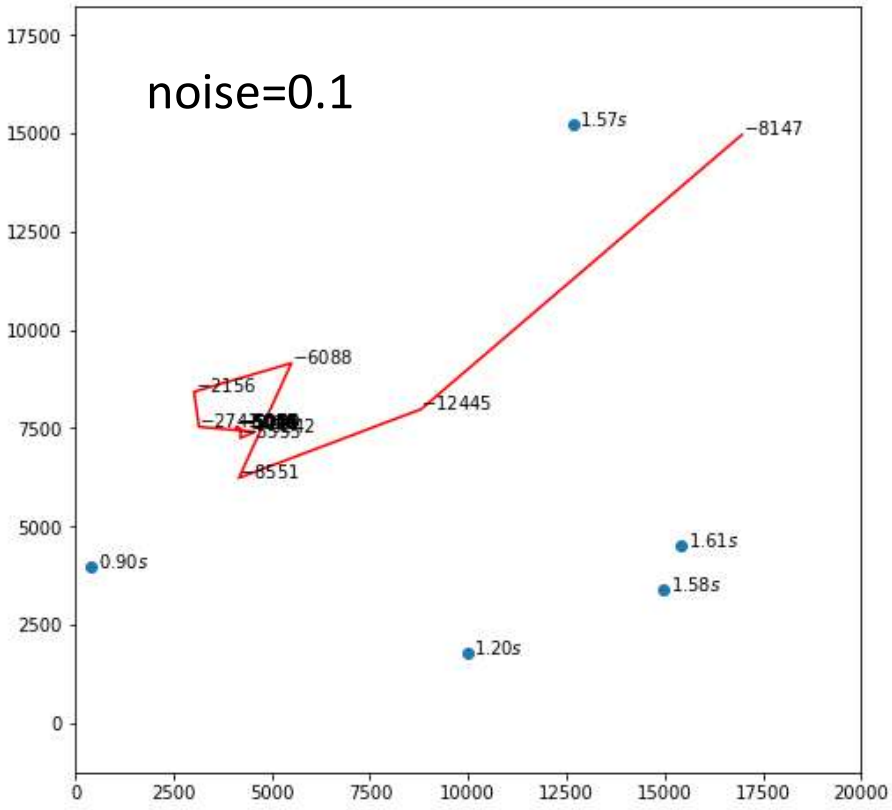




Sensibilidade ao ruído nos dados

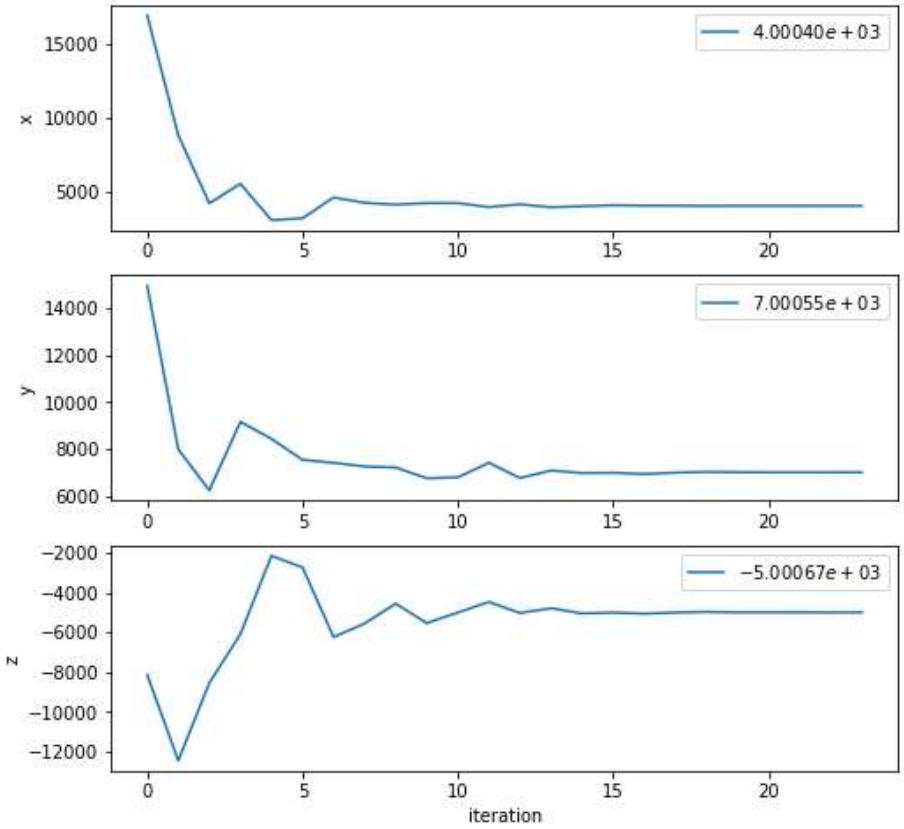
```
def iniSeismic(is3D=False,nE=4,iseed=10,noise=0):
    …# noise tem as unidades de tE (s)
     distE=np.sqrt((xS-xE)**2+(yS-yE)**2+(zS-zE)**2)
     tE=distE/cs+noise*(np.random.sample(distE.shape)-0.5)
     …
return xE,yE,zE,tE,cs,vmin,vmax
…
xE,yE,zE,tE,cs,vmin,vmax=iniSeismic(is3D=True,nE=5,noise=0.1)
```


noise=0
17500
15000
12500
10000
7500
5000
2500
0
0
2500
5000
7500
10000
12500
15000
17500
20000
1.62s
−8147
−6088
−2156
−12445
−8551
1.59s
0.86s
1.57s
1.17s


noise=0.1
17500
15000
12500
10000
7500
5000
2500
0
0
2500
5000
7500
10000
12500
15000
17500
20000
1.57s
−8147
−6088
−2156
−12445
−8551
1.61s
0.90s
1.58s
1.20s

Sensibilidade ao ruido nos dados (mesmo iseed: mesmo first guess e path)

noise=0



noise=0.1 s (~<10%)