



Ciências
ULisboa

Modelação Numérica

Aula 8

Algoritmo genético

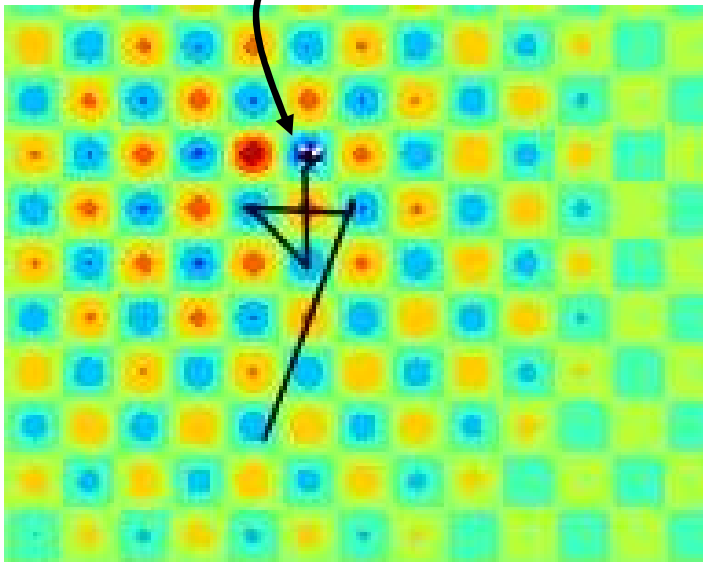
Comentários

O método anterior funciona sempre desde que só exista um mínimo da função de custo. Se existirem vários mínimos locais, pode convergir para um deles e falhar o mínimo absoluto, dependendo do ponto de partida (first guess), e da evolução do passo (xstep).

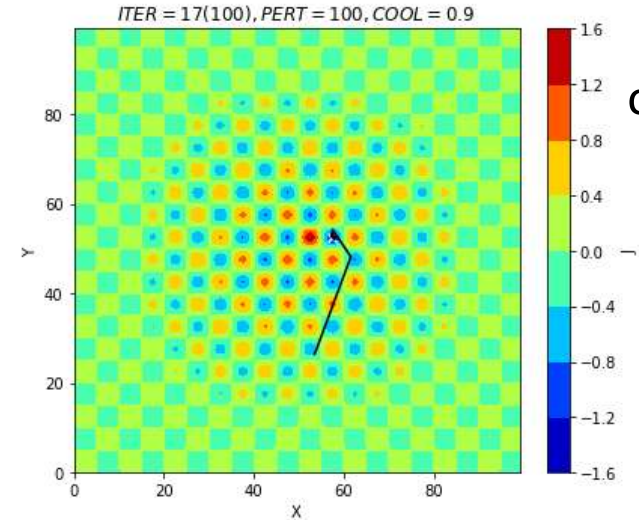
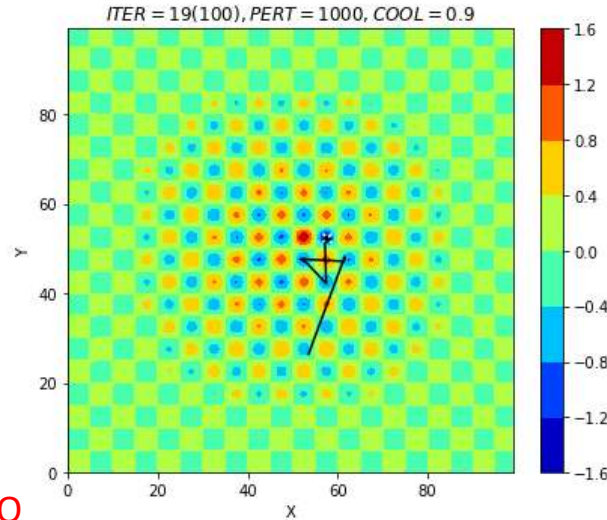
Função de custo com muitos mínimos locais (mesmo first guess)

Também é sensível a outros parâmetros.

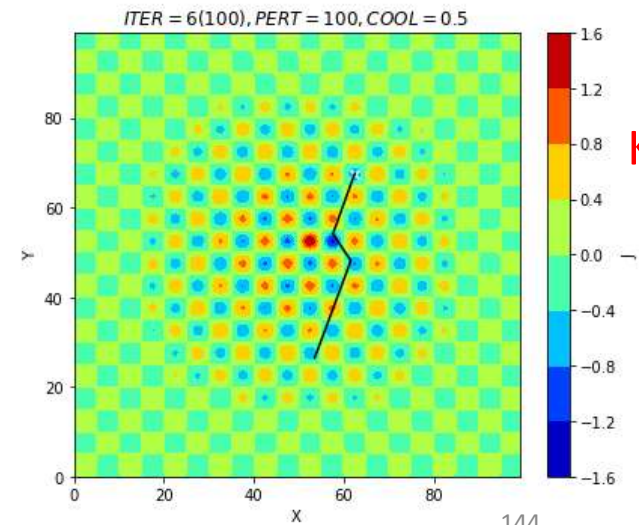
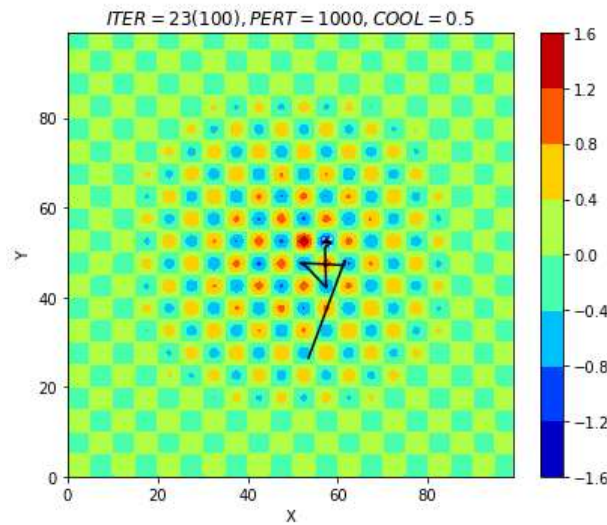
MÍNIMO ABSOLUTO



ok



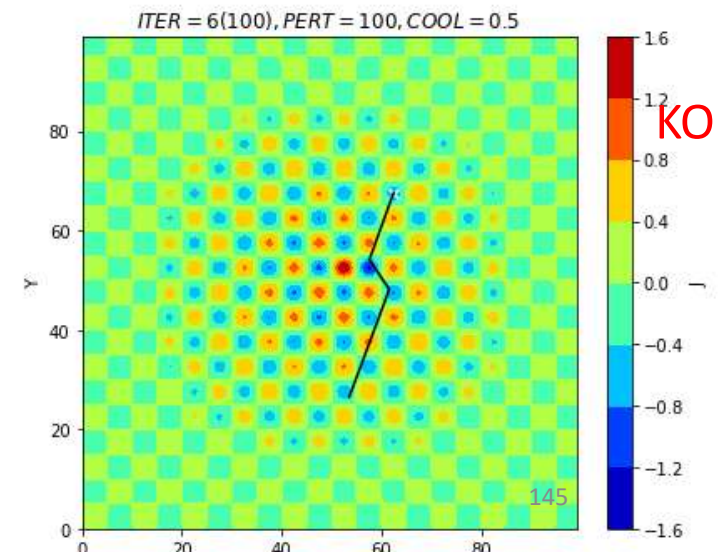
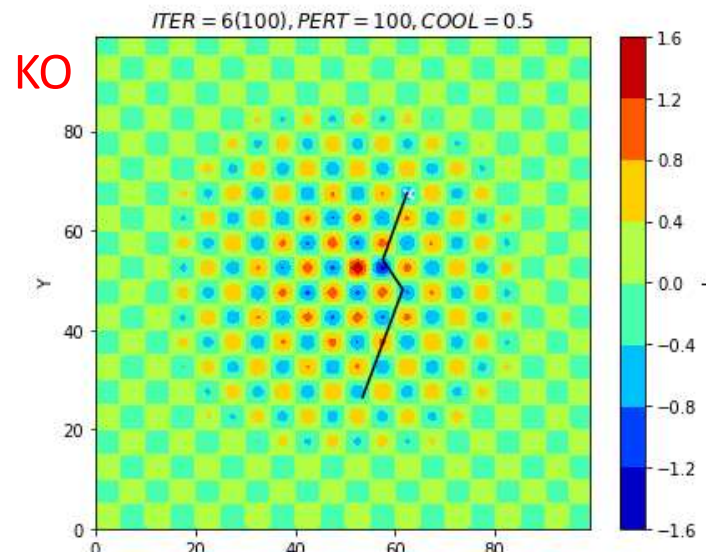
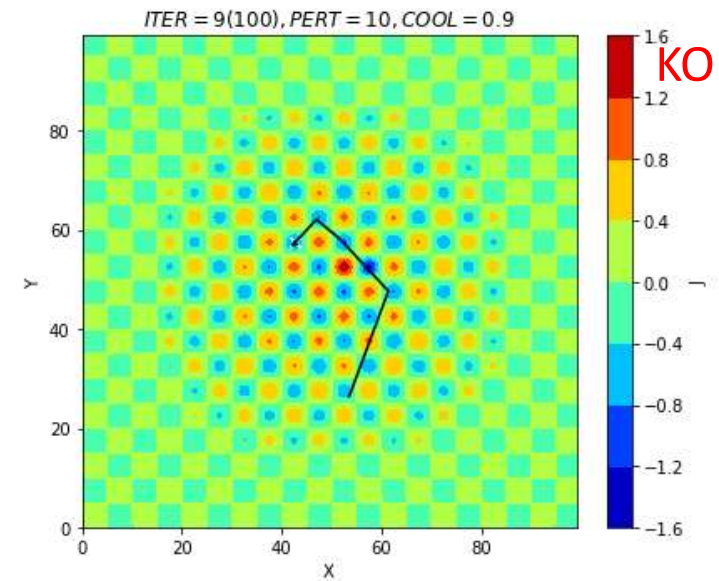
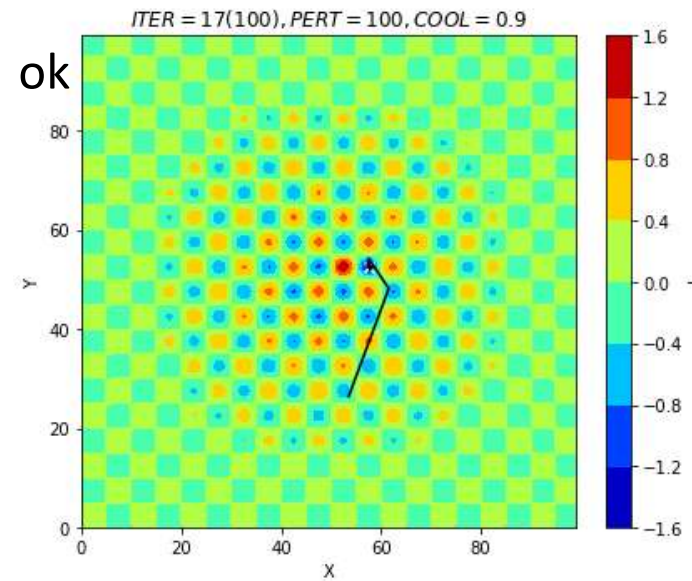
ok



KO

O ciclo interno
(maxPERT) é mais
importante que
externo...

Porquê?



Cálculo dos coeficientes de um polinómio $y = ax^3 + bx^2 + cx + d$

```
...
def inipol(iseed=10, nE=1000) :
    a=5;b=2;c=3;d=2 #SOLUÇÃO
    xmin=0;xmax=10 #domínio xE
    np.random.seed(iseed)
    xE=xmin+(xmax-xmin)*np.random.sample(nE)
    yO=a*xE**3+b*xE**2+c*xE+d
    return xE,yO
def cost(V) :
    a,b,c,d=V
    custo=np.sum(((a*xE**3+b*xE**2+c*xE+d)-yO)**2)
    return custo
xE,yO=inipol() #observações
V=np.zeros(4);vnames=np.array(['a','b','c','d'])
vmin=np.zeros(V.shape);vmax=10*np.ones(V.shape) #domínio (a,b,c,d)
Jmin=-10.;minvstep=(vmax-vmin)/10000
kappa=np.float64(0.1);T=10.
maxITER=1000
maxPERT=100000
COOL=0.9
n,V,iTER,path=annealD(vmin,vmax,Jmin,minvstep,maxITER,maxPERT,COOL,kappa,T,outITER,iseed=2)
```

Solução direta:

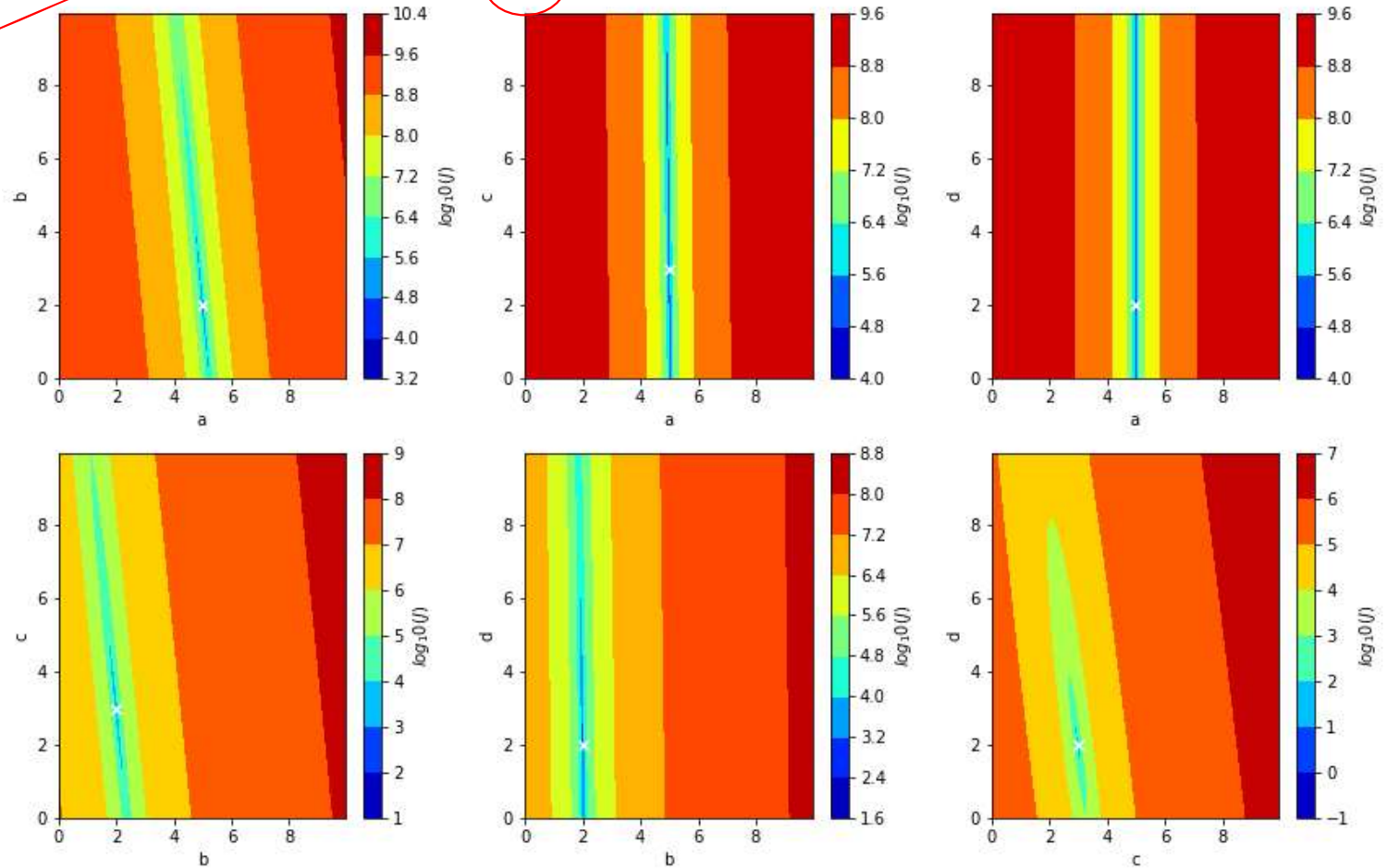
$a, b, c, d = \text{np.polyfit}(xE, yO, 3)$

Otimização em 4 dimensões

5,2,3,2

Para explorar 4 dimensões são precisos muitos movimentos!

ITER = 44(1000), PERT = 100000, min Δx = 0.00100, a = 5.000, b = 2.002, c = 2.990, d = 2.010



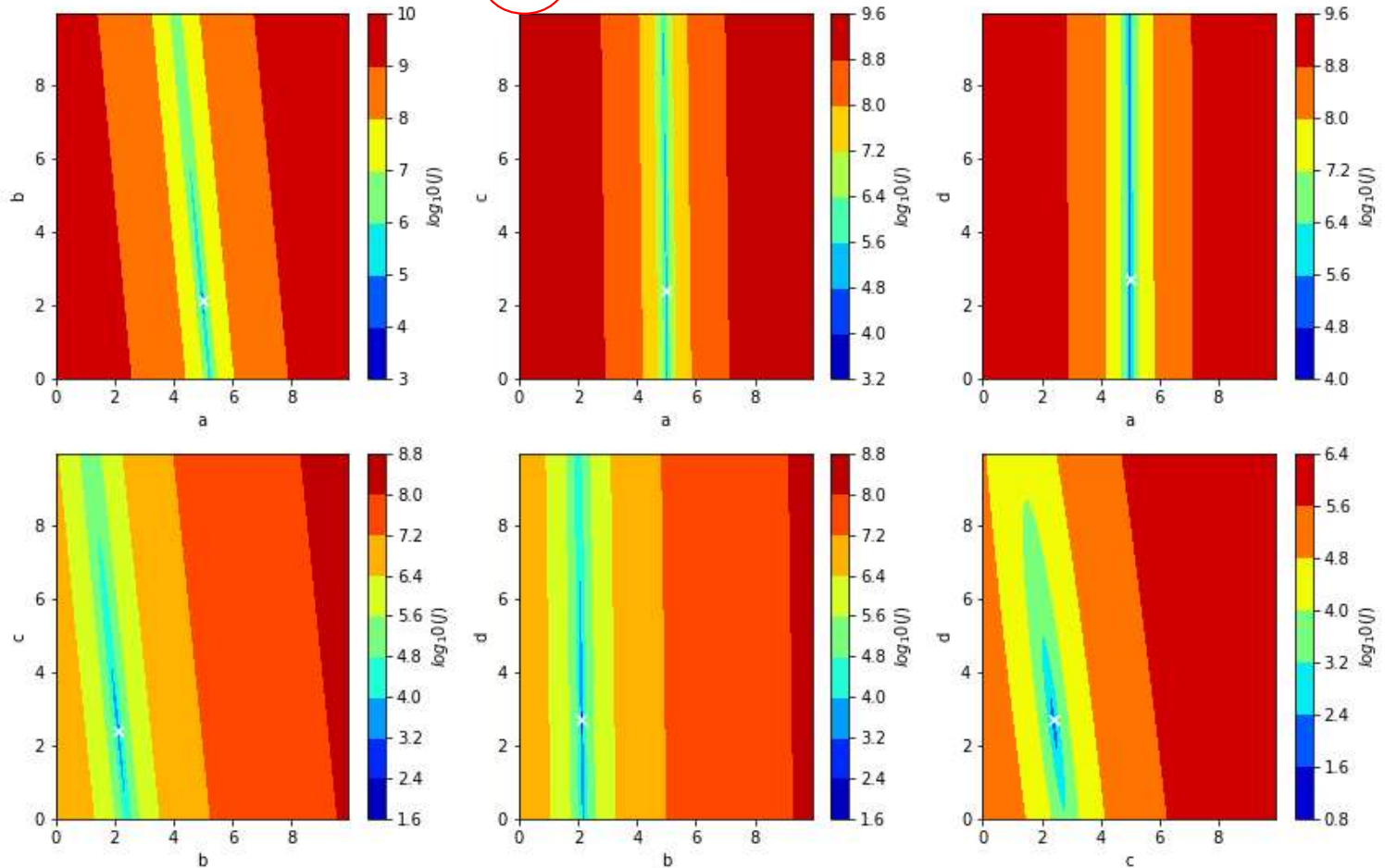
Otimização em 4 dimensões: 6 planos ortogonais
Combinações(4,2)

5,2,3,2

Parece estar no mínimo mas está LONGE!

Notar que se trata de um espaço a 4 dimensões e os planos passam pela “solução” obtida que não é a verdadeira...

ITER = 44(1000), PERT = 10000, min Δx = 0.00100, a = 4.992, b = 2.128, c = 2.395, d = 2.699



Um outro algoritmo de otimização: Algoritmo genético

Neste algoritmo utilizam-se conceitos da genética para resolver um problema de otimização.

Cada iteração do algoritmo simula uma **nova geração** numa **população** de organismos em que existe **diversidade** genética.

A diversidade é produzida no início, de forma aleatória, e **reforçada** em cada geração com **mutação**.

A transição entre gerações inclui:

- (1) Seleção do indivíduo com melhor desempenho (menor função de custo)
- (2) Cruzamento aleatório para produzir os novos (N-2) membros
- (3) Introdução de um **mutante global**
- (4) Mutações locais

Questões

Como realizar **mutações**?

- Escolher um novo elemento aleatoriamente (= ao início)
- Perturbar um membro da geração anterior por uma “pequena” perturbação

Como selecionar os **progenitores**?

- Aleatoriamente (impedir autocruzamentos)

Como **acasalar**?

- Média aritmética dos parâmetros

Parâmetros ajustáveis

Domínio da solução (mínimo e máximo de cada variável a ajustar)

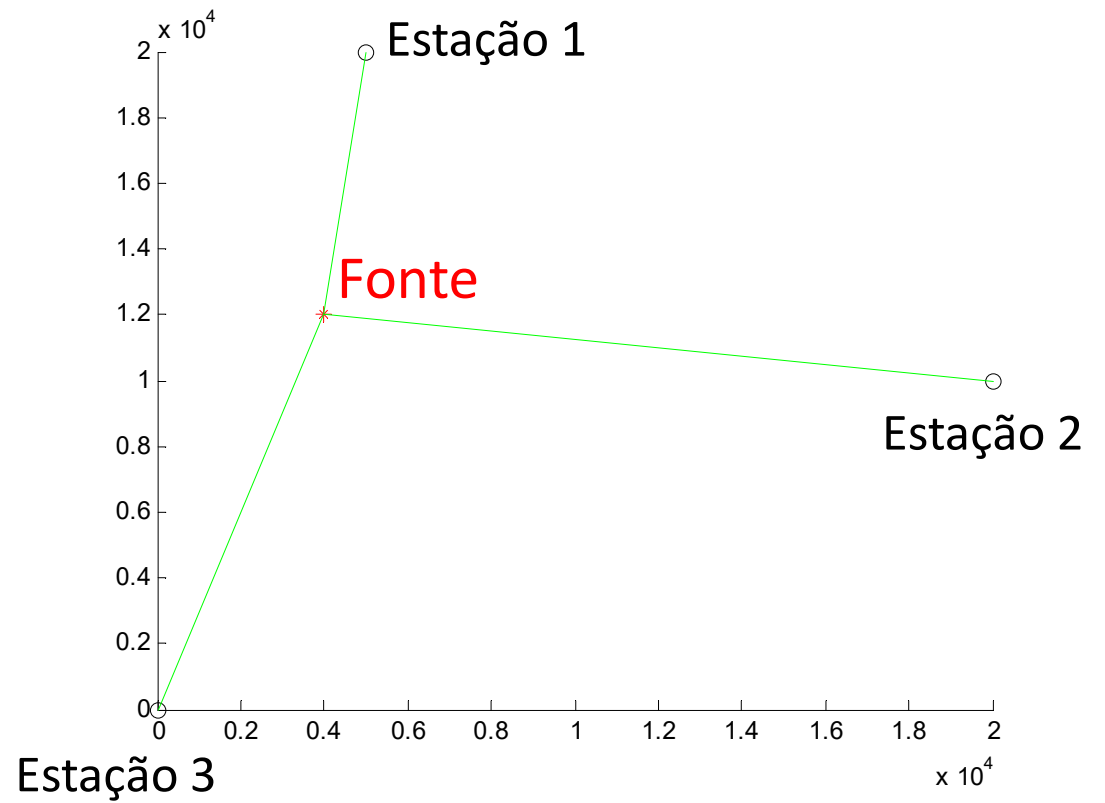
Dimensão da população ($\sim 10 \times$ Número de variáveis a ajustar)

Critérios de **paragem**:

- ✓ Número de gerações (Número máximo de iterações)
- ✓ Target para a função de custo J_{Min} (se for conhecido)

Problema anterior

Localizar uma fonte sísmica em 2 dimensões (x,y) com velocidade constante.



Algoritmo genético

```
def optim(vmin, vmax, Jmin, maxGEN, maxPOP, iseed=None) :
    ndim=len(vmin)
    mutRANGE=1/100
    np.random.seed(iseed)
    P=np.zeros((ndim,maxPOP))
    for iP in range(maxPOP):
        P[:,iP]=vmin+(vmax-vmin)*np.random.random(ndim) #população inicial
    JP=cost(P)
    IS=np.argsort(JP) #ordena (melhor primeiro)
    J=JP[IS[0]];V=P[:,IS[0]] # melhor membro
    bestJ=J;bestV=V
    iGEN=0;kGEN=0;kMute=0
    Evolution=np.zeros((100,ndim+2)) #Evolução dos parâmetros
    Evolution[kGEN,0]=0 #Geração
    Evolution[kGEN,1]=0 #Custo
    Evolution[kGEN,2:]=V[:]
```

Algoritmo genético (2)

```
def optim(vmin, vmax, Jmin, maxGEN, maxPOP, iseed=None) :  
    ...  
    while iGEN < maxGEN and J > Jmin:  
        P[:,0]=P[:,IS[0]] #save best  
        for iP in range (1,maxPOP-1):  
            rr1=int(np.random.random() * (maxPOP-1)) #progenitores  
            rr2=int(np.random.random() * (maxPOP-1))  
            while rr1==rr2:  
                rr2=int(np.random.random() * (maxPOP-1))  
            P[:,iP]=0.5*P[:,IS[rr1]]+0.5*P[:,IS[rr2]] #acasalamento  
            P[:,maxPOP-1]=vmin+(vmax-vmin)*np.random.sample(ndim); #mutante  
  
        JP=cost(P)  
        IS=np.argsort(JP) #ordena melhores primeiro
```

Algoritmo genético (3)

```
def optim(vmin, vmax, Jmin, maxGEN, maxPOP, iseed=None):  
    ...  
    #Mutações locais  
    for iP in range(maxPOP):  
        mute=1+mutRANGE*(np.random.random(ndim)-0.5)  
        Pmute=P[:,iP]*mute  
        Jmute=costONE(Pmute)  
        Jori=costONE(P[:,0])  
        if Jmute<Jori:  
            P[:,iP]=Pmute  
            kMute=kMute+1  
            JP[iP]=Jmute  
            kMute=kMute+1  
    IS=np.argsort(JP) #ordena melhores primeiro
```

Algoritmo genético (4)

```
def optim(vmin, vmax, Jmin, maxGEN, maxPOP, iseed=None) :
    ...
    J=JP[IS[0]];V=P[:,IS[0]] #seleciona melhor
    iGEN=iGEN+1
    if J<bestJ: #Evolução (menor custo)
        bestJ=J
        bestV=V
        kGEN=kGEN+1
        print(' iGEN=%3i, kGEN=%3i, kMute=%3i, J=%f, V=%f, V=%f' %\
              (iGEN, kGEN, kMute, J, V[0], V[1]))
        Evolution[kGEN,0]=iGEN
        Evolution[kGEN,1]=J
        Evolution[kGEN,2:]=V[:]

    return bestV, bestJ, kGEN, Evolution
```

Função de custo

```
def cost(V):  
    [nd, num]=V.shape  
    custo=np.zeros(num)  
    for k in range(num):  
        X=V[0,k];Y=V[1,k] #parameters to optimize  
        dist=np.sqrt((X-xE)**2+(Y-yE)**2)  
        erro=dist/cs-tE  
        custo[k]=np.sum(abs(erro))  
    return custo  
  
def costONE(V):  
    X=V[0];Y=V[1] #parameters to optimize  
    dist=np.sqrt((X-xE)**2+(Y-yE)**2)  
    erro=dist/cs-tE  
    custo=np.sum(abs(erro))  
    return custo
```


Inicialização

```
def dados():  
    cs=8000 #velocidade da onda sismica  
    xS=4000;yS=7000 #solução:posição da fonte  
    xE=np.array([0,10000,5000])  
    yE=np.array([0,10000,10000])  
    distE=np.sqrt((xS-xE)**2+(yS-yE)**2)  
    tE=distE/cs  
    return xE,yE,tE,cs,xS,yS
```

```

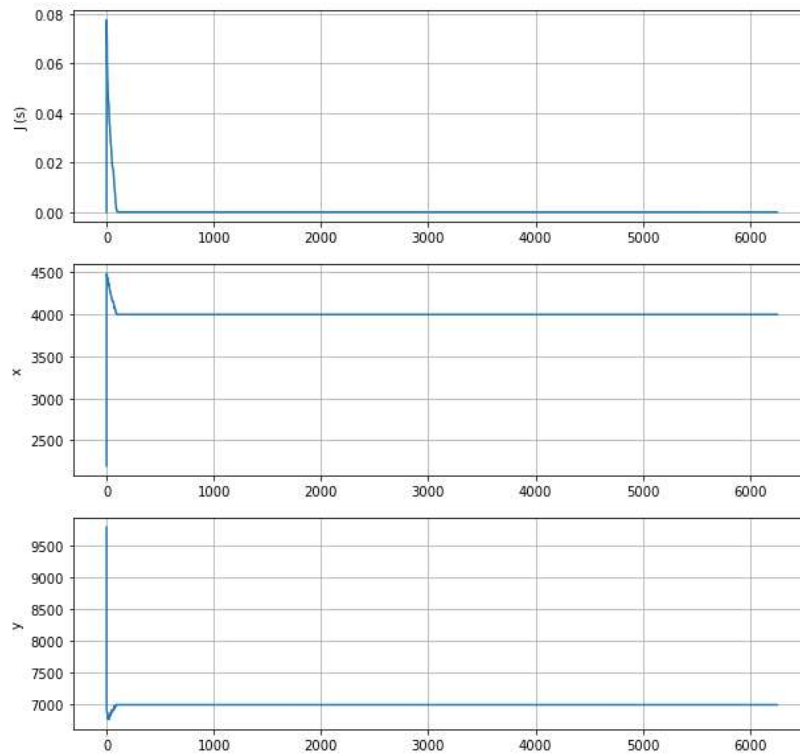
maxITER=10000;maxPERT=10;Jmin=1e-20; #critérios de paragem
xmin=0;xmax=10000;ymin=0;ymax=10000 #domínio
vmin=np.array([xmin,ymin])
vmax=np.array([xmax,ymax])
xE,yE,tE,cs,xS,yS=dados()
plt.figure(figsize=(10,10))
V,J,kGEN,Evolution=optim(vmin,vmax,Jmin,maxITER,maxPERT,iseed=None)
for kP in range(1,4):
    plt.subplot(3,1,kP)
    plt.plot(Evolution[:kGEN+1,0],Evolution[:kGEN+1,kP]\
             ,label=r'S=%2.1f,Mut=%3i,J=%5.3f s' % (select,kGEN,J))
    plt.grid()

plt.suptitle('MaxGEN=%4i, Pop=%3i' % (maxITER,maxPERT))

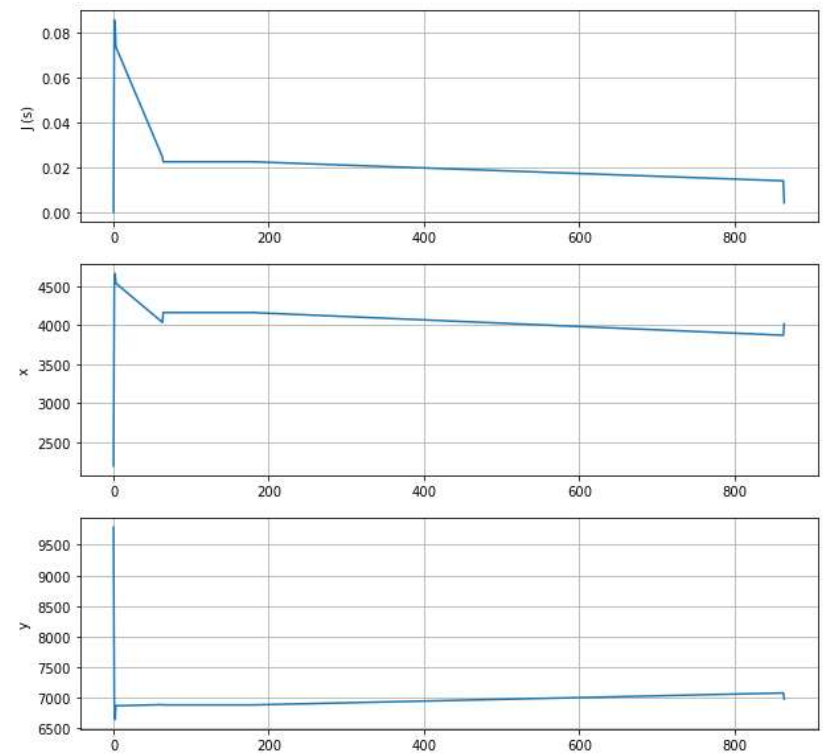
```

Evolução

MaxGEN=10000, Pop= 10, Time=7.67s, mutLocal=True



MaxGEN=10000, Pop= 10, Time=2.58s, mutLocal=False



Algoritmo genético (n-dimensões)

```
def optim(vmin, vmax, Jmin, maxGEN, maxPOP, iseed=None, mutLocal=True):
    ndim=len(vmin)
    mutRANGE=10 #mutação local aditiva
    np.random.seed(iseed)
    P=np.zeros((ndim,maxPOP))
    for iP in range(maxPOP):
        P[:,iP]=vmin+(vmax-vmin)*np.random.random(ndim) #população
    JP=cost(P)
    IS=np.argsort(JP) #ordena (melhor primeiro)
    J=JP[IS[0]];V=P[:,IS[0]] # melhor membro
    bestJ=J;bestV=V
    iGEN=0;kGEN=0;kMute=0
    Evolution=np.zeros((2000,ndim+2))
    Evolution[kGEN,0]=0
    Evolution[kGEN,1]=0
    Evolution[kGEN,2:]=V[:]
```

Algoritmo genético (2)

```
def optim(vmin,vmax,Jmin,maxGEN,maxPOP,iseed=None,mutLocal=True):  
...  
    while iGEN<maxGEN and J>Jmin:  
        P[:,0]=P[:,IS[0]] #save best  
        for iP in range (1,maxPOP-1):  
            rr1=int(np.random.random()*(maxPOP-1)) # progenitores  
            rr2=int(np.random.random()*(maxPOP-1))  
            while rr1==rr2: #rejeita clones  
                rr2=int(np.random.random()*(maxPOP-1))  
            P[:,iP]=0.5*P[:,rr1]+0.5*P[:,rr2] #acasalamento  
            P[:,maxPOP-1]=vmin+(vmax-vmin)*np.random.sample(ndim); #mutante  
  
        JP=cost(P)  
        IS=np.argsort(JP) #ordena melhores primeiro
```

Algoritmo genético (3)

```
def optim(vmin, vmax, Jmin, maxGEN, maxPOP, iseed=None, mutLocal=True):  
...  
    if mutLocal:  
        for iP in range(maxPOP):  
            mute=mutRANGE*(np.random.random(ndim)-0.5)  
            Pmute=P[:,iP]+mute  
            Jmute=costONE(Pmute)  
            Jori=costONE(P[:,0])  
            if Jmute<Jori: #aceita mutações positivas  
                P[:,iP]=Pmute  
                kMute=kMute+1  
                JP[iP]=Jmute  
            kMute=kMute+1  
    IS=np.argsort(JP) #ordena melhores primeiro
```

Algoritmo genético (4)

```
def optim(vmin,vmax,Jmin,maxGEN,maxPOP,iseed=None,mutLocal=True):
...
    J=JP[IS[0]];V=P[:,IS[0]] #seleciona melhor
    iGEN=iGEN+1
    if J<bestJ: #aceita melhoramentos
        bestJ=J
        bestV=V
        kGEN=kGEN+1
        print('iGEN=%3i,kGEN=%3i,kMute=%3i,J=%f,V=%f,V=%f,V=%f' \
              % (iGEN,kGEN,kMute,J,V[0],V[1],V[2]))
        Evolution[kGEN,0]=iGEN #geração em que melhorou
        Evolution[kGEN,1]=J    #nova função de custo
        Evolution[kGEN,2:]=V[:] #novos parâmetros
    return bestV,bestJ,kGEN,Evolution
```

Algoritmo genético (3)

```
def optim(vmin,vmax,Jmin,maxGEN,maxPOP,iseed=None,mutLocal=True):
...
    J=JP[IS[0]];V=P[:,IS[0]] #seleciona melhor
    iGEN=iGEN+1
    if J<bestJ: #aceita melhoramentos
        bestJ=J
        bestV=V
        kGEN=kGEN+1
        print('iGEN=%3i,kGEN=%3i,kMute=%3i,J=%f,V=%f,V=%f,V=%f' \
              % (iGEN,kGEN,kMute,J,V[0],V[1],V[2]))
        Evolution[kGEN,0]=iGEN #geração em que melhorou
        Evolution[kGEN,1]=J    #nova função de custo
        Evolution[kGEN,2:]=V[:] #novos parâmetros
    return bestV,bestJ,kGEN,Evolution
```


Dados 3d (com ruído)

```
def dados (noise=0, iseed=None) :
    cs=8000 #velocidade da onda sismica
    xS=4000;yS=7000;zS=-3000 #solução:posição da fonte
    xE=np.array([0,10000,5000]) #posição das estações
    yE=np.array([0,10000,1000]) #posição das estações
    zE=np.array([-5000,-5000,-5000]) #posição das estações
    distE=np.sqrt((xS-xE)**2+(yS-yE)**2+(zS-zE)**2) #distâncias
    if noise>0:
        np.random.seed(iseed)
        tE=distE/cs*(1+noise*(np.random.random(distE.shape)-0.5))
    else:
        tE=distE/cs #observações
    return xE,yE,zE,tE,cs,xS,yS,zS
```

Função de custo

```
def cost(V): #vetor custo de uma população
    [nd,num]=V.shape
    custo=np.zeros((num))
    for k in range(num):
        X=V[0,k];Y=V[1,k];Z=V[2,k] #parameters to optimize
        dist=np.sqrt((X-xE)**2+(Y-yE)**2+(Z-zE)**2)
        erro=dist/cs-tE
        custo[k]=np.sum(abs(erro))
    return custo

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

Main

```
maxITER=10000;maxPERT=10;Jmin=1e-4; #critérios de paragem
unitJ='s'
xmin=0;xmax=10000;ymin=0;ymax=10000;zmin=-5000;zmax=0 #domínio
vmin=np.array([xmin,ymin,zmin])
vmax=np.array([xmax,ymax,zmax])
ndim=len(vmin)
xE,yE,zE,tE,cs,xS,yS,zS=dados(noise=0) #geração das observações
```

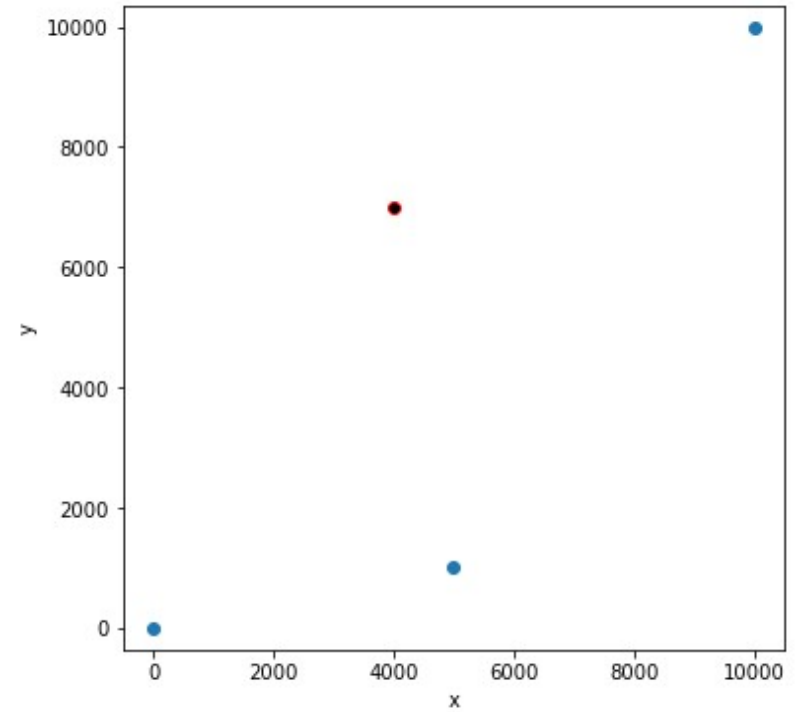
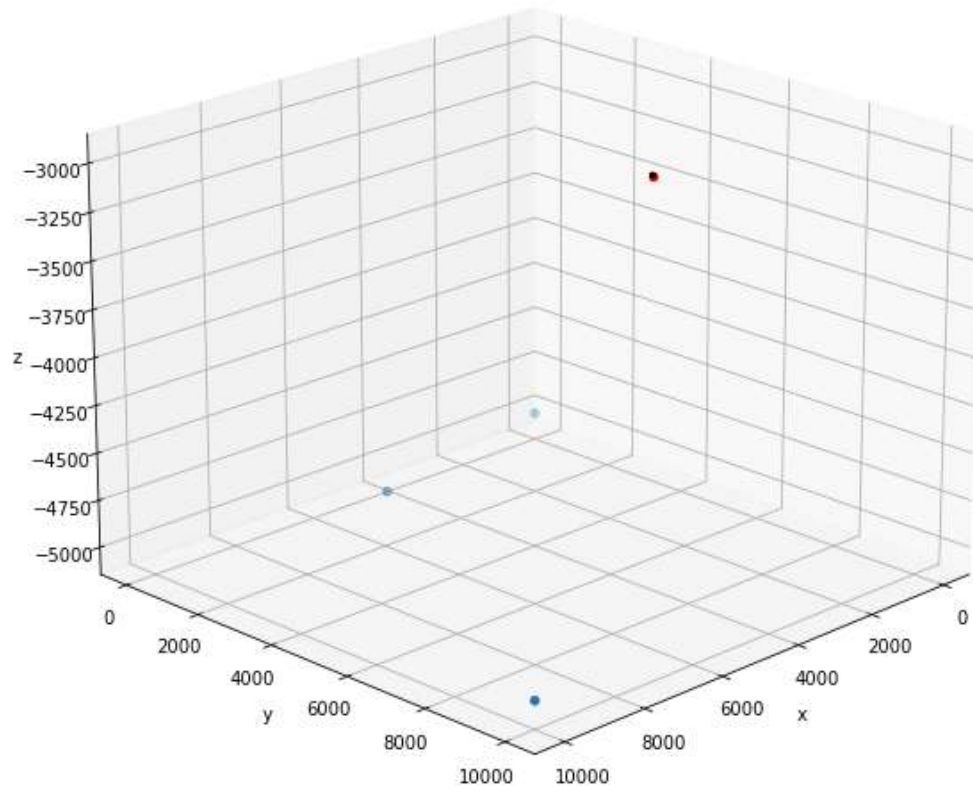
Main (2)

```
runs=10
plt.figure(2,figsize=(10,12))
mutLocal=True
hisJ=np.zeros(runs)
hisV=np.zeros((runs,ndim))
for run in range(runs):
    print(run)
    V,J,kGEN,Evolution=optim(vmin,vmax,Jmin,maxITER,maxPERT,iseed=None,muLocal=mutLocal)
    hisJ[run]=J
    hisV[run,:]=V
    timespent=time.process_time()-timestart
    for kP in range(1,len(vmin)+2):
        plt.subplot(4,1,kP)
        plt.plot(Evolution[:kGEN+1,0],Evolution[:kGEN+1,kP])
        plt.grid()
```

Main (3)

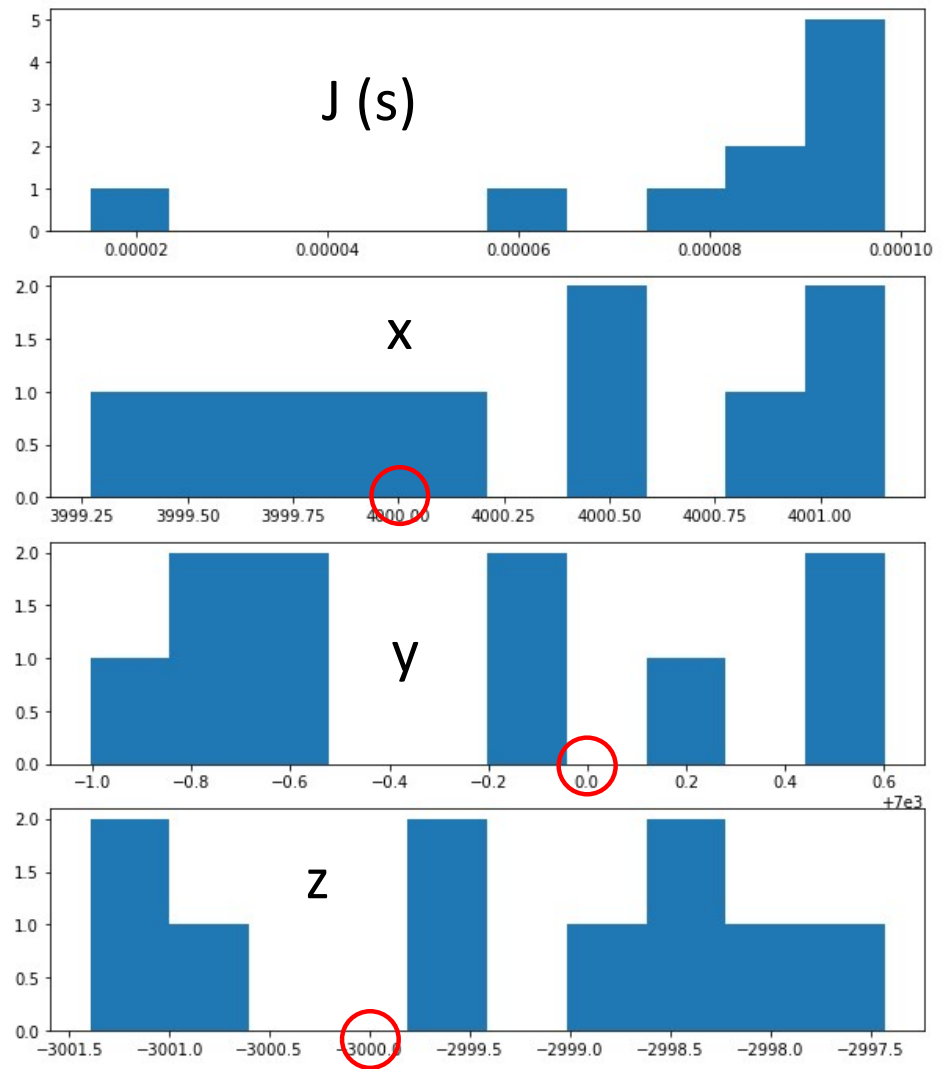
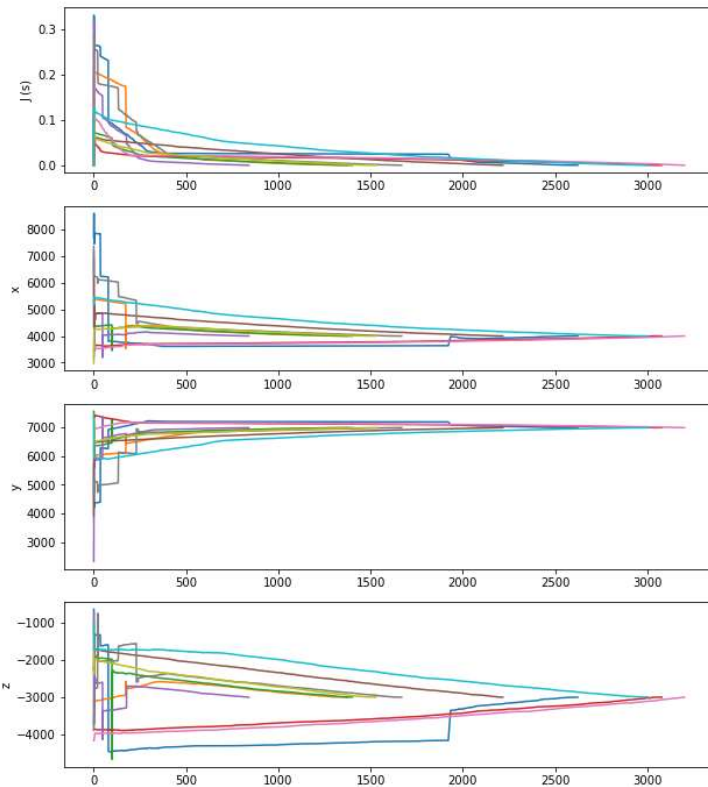
```
plt.suptitle(r'$MaxGEN=%4i, Pop=%3i, J_{min}=%2.1e%s, Time=%3.2fs, mutLocal=%s$'\
            % (maxITER,maxPERT,Jmin,unitJ,timespent,muLocal))
plt.subplot(4,1,1);plt.ylabel('J (s)')
plt.subplot(4,1,2);plt.ylabel('x')
plt.subplot(4,1,3);plt.ylabel('y')
plt.subplot(4,1,4);plt.ylabel('z')
fig=plt.figure(figsize=(10,10)) #figura 3D
ax=fig.add_subplot(111, projection='3d')
ax.scatter(xs=xE,ys=yE,zs=zE) #estações
ax.scatter(xs=xS,ys=yS,zs=zS,color='red') #fonte
ax.scatter(xs=hisV[:,0],ys=hisV[:,1],zs=hisV[:,2],marker='.',color='black') #solução
ax.view_init(elev=20., azim=45)
ax.set_xlabel('x');ax.set_ylabel('y');ax.set_zlabel('z')
plt.figure(figsize=(6,6)) #figura 2D
plt.scatter(xE,yE)
plt.scatter(xS,yS,color='red')
plt.scatter(hisV[:,0],hisV[:,1],color='black',marker='.')
plt.xlim(vmin[0],vmax[0]);plt.ylim(vmin[1],vmax[1])
plt.xlabel('x');plt.ylabel('y');plt.axis('equal')
```

Caso 1 $J_{\min} = 10^{-4} s$, noise=0

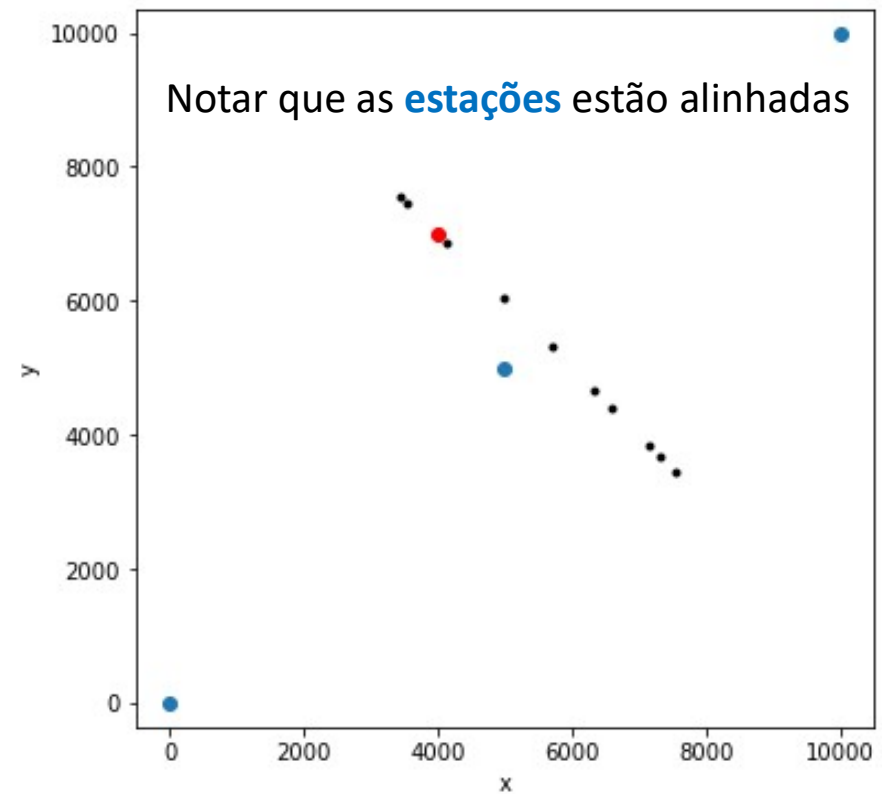
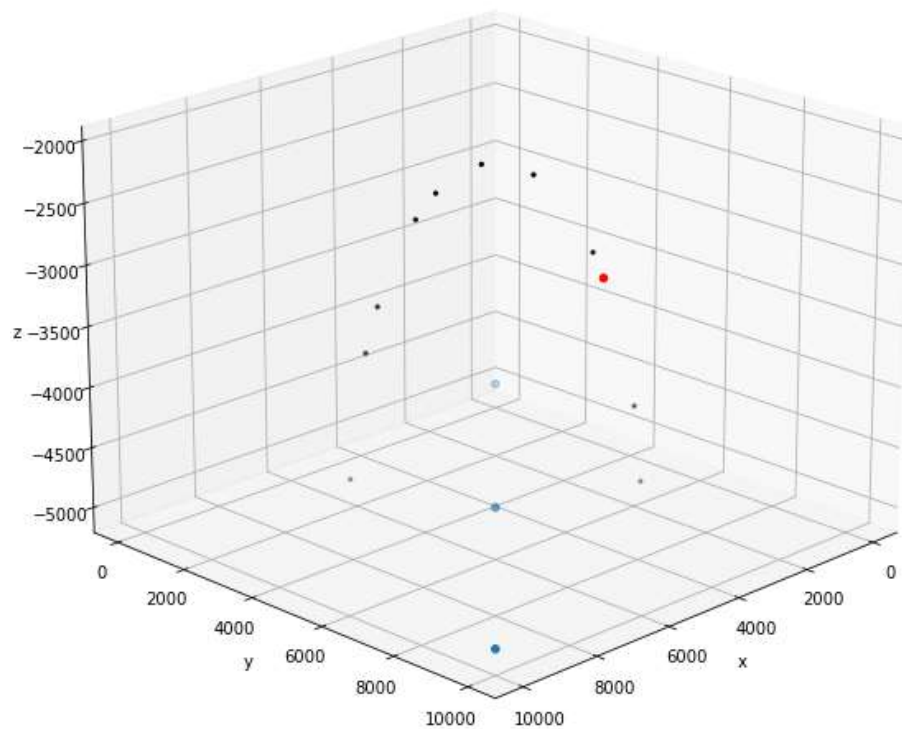


Caso 1 $J_{min}=10^{-4}s$, noise=0

MaxGEN = 10000, Pop = 10, $J_{min} = 1.0e - 04s$, Time = 20.64s, mutLocal = True



Caso 1 $J_{\min} = 10^{-4} \text{ s}$, noise=0: solução ambígua



Caso 2 $J_{min}=10^{-4}s$, noise=0

MaxGEN = 10000, Pop = 10, $J_{min} = 1.0e - 04s$, Time = 3.00s, mutLocal = True

As várias soluções têm erro semelhante

