MO431A_Tarefa3

April 24, 2021

MO431A - Fundamentos de Álgebra Linear e Otimização para Aprendizado de Máquina
Equipe:

- Maria Fernanda Tejada Begazo - RA 197488
- Jose Italo da Costa Silva - RA 265682
- Gian Franco Joel Condori Luna - RA 234826

Tarefa 03

A tarefa foi desenvolvida na linguagem python. Para isso utilizou-se notebooks jupyter no ambiente Google Colaboratory (Google Colab).

Pré-requisitos (codificação inicial, instalação e importação de pacotes):

[ ]:
```
!pip install Py-BOBYQA
```

[ ]:
```
!pip install nlopt
```

[ ]:
```
from google.colab import drive
drive.mount('/content/drive')
```

[4]:
```
#Imports necessários:
import numpy as np
import cv2
from google.colab.patches import cv2_imshow
from numpy import linalg
import matplotlib.pyplot as plt
import matplotlib as mpl
from matplotlib.colors import LogNorm
from sympy import Derivative, diff, simplify
from sympy import Symbol, exp, Heaviside
import scipy.linalg as sciLa
from scipy.optimize import minimize, rosen, rosen_der
from autograd import elementwise_grad, value_and_grad
from scipy.optimize import minimize_scalar
from scipy.optimize import line_search
import pybobyqa
import nlopt
import time
import pandas as pd
import cv2
import io
```

```
import tensorflow as tf
```

[5]:
```
var_tolerancia = 1e-5
```

Funções básicas:

[6]:
```
# Plot da função em curvas de nivel e em 3D
def functionPlot(title='', data = None, xmin = None):
  x_lim = 4.5
  x0 = 4.0
  y0 = 4.0
  x_min = np.array([[3.0, 2.0], [-2.805118, 3.131312], [-3.779310, -3.283186],␣
 ↪[3.584428, -1.848126]])
  x = np.arange(-x_lim, x_lim+0.2, 0.2)
  y = np.arange(-x_lim, x_lim+0.2, 0.2)
  X, Y = np.meshgrid(x, y)
  Z = f(X, Y)

  fig = plt.figure(figsize=(14,6))
  cmp = plt.cm.jet

  # Calculando a curva de Nivel
  ax = fig.add_subplot(1, 2, 1)
  dz_dx = elementwise_grad(f, argnum=0)(X, Y)
  dz_dy = elementwise_grad(f, argnum=1)(X, Y)
  ax.contour(X, Y, Z, levels=np.logspace(0, 5, 35), norm=LogNorm(), cmap=cmp,␣
 ↪alpha=.4)
  ax.quiver(X, Y, X - dz_dx, Y - dz_dy, alpha=.5)
  ax.plot(x0,y0, 'ko', markersize=8, label='initial x')

  ax.plot(x_min[0,0],x_min[0,1], 'r*', markersize=10, label='x_min')
  for i in range(1, len(x_min)):
    ax.plot(x_min[i,0],x_min[i,1], 'r*', markersize=10)

  if data is not None:
    for i in range(len(data)):
      ax.plot(data[i,0], data[i,1], 'k', marker=7, markersize=8 )
    ax.plot(data[:,0], data[:,1], 'k-')

  if (xmin is not None):
    ax.plot(xmin[0], xmin[1],'m*', markersize=10, label='Find x_min')


  ax.set_xlabel('x')
  ax.set_ylabel('y')
  ax.legend()

  # Plotando a imagem em 3D
  ax = fig.add_subplot(1, 2, 2, projection='3d')
```

```
 ax.plot_surface(X,Y, Z, norm=LogNorm(), rstride=1, cstride=1,␣
↪edgecolor='none', alpha=.4, cmap=cmp)

 # Verificação dos dados em pontos tridimensionais dispersos
 ax.scatter(x0,y0, f(x0,y0), color='k', marker='o', s=50, label='initial x')

 ax.scatter(x_min[0,0],x_min[0,1], f(x_min[0,0],x_min[0,1]), color='r',␣
↪marker='+', s=100, label='x_min')
 for i in range(1, len(x_min)):
   ax.scatter(x_min[i,0],x_min[i,1], f(x_min[i,0],x_min[i,1]), color='r',␣
↪marker='+', s=100)

 if data is not None:
   fz = []
   for i in range(len(data)):
     ax.scatter3D(data[i,0], data[i,1], f(data[i,0],data[i,1]), color='k',␣
↪marker=7, s=80)
     fz.append(f(data[i,0],data[i,1]))
   ax.plot(data[:,0], data[:,1], np.array(fz), color='k')

 if (xmin is not None):
   ax.scatter3D(xmin[0], xmin[1], f(xmin[0],xmin[1]), color='m', marker='*',␣
↪s=80, label='Find x_min')

 ax.set_xlabel('x')
 ax.set_ylabel('y')
 ax.set_zlabel('f(x,y)')
 ax.legend()
 #(elev=ele, azim=azm)
 #elev is the angle between the eye and the xy plane
 #The azimuth is the rotation around the z axis
 ax.view_init(60, -120)
 #dist: distance from the center visible point in data coordinates.
 ax.dist = 9

 plt.suptitle(title, size=20)
 plt.show()
```

[7]:
```
#Função para exibição dos resultados
def printResult(res, t_, x_0, title=''):
  global ps
  print("Ponto Minimo: ", np.round(res.x, 2))
  print("Valor da Função: ", np.round(f(res.x[0], res.x[1]), 2))
  print("Número de chamadas: ", res.nit)
  print("Tempo (ms): ", np.round(t_,4))
  functionPlot(title, np.array(ps), xmin = res.x)
```

```
  ps = [x_0]

def reporter(p):
  """Reporter function to capture intermediate states of optimization."""
  global ps
  ps.append(p)

class Reporter:
  def __init__(self, pto, valorPto, nroIter, t_):
    self.x = pto
    self.jac = valorPto
    self.nit = nroIter
    self.t = t_
```

Introdução

Na otimização matemática, a função de Himmelblau é uma função multimodal, que vamos a usar para testar o desempenho de nossos alogritmos de descida de gradiente.

A função é definida como:

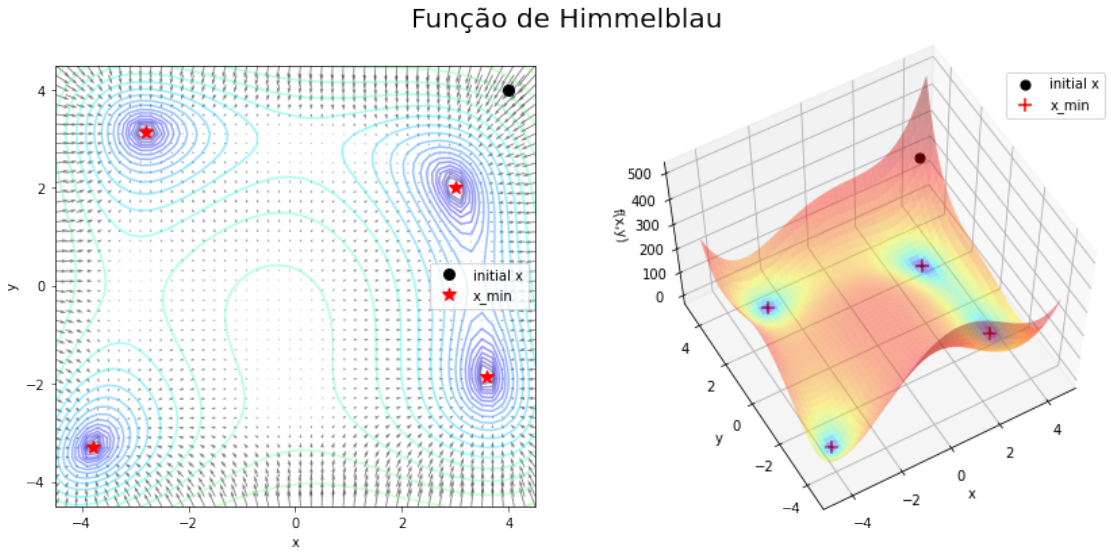
$f(x, y) = (x^2 + y - 11)^2 + (x + y^2 - 7)^2$

Tem quatro mínimos locais:

- $[3.0, 2.0]$
- $[-2.805118, 3.131312]$
- $[-3.779310, -3.283136]$
- $[3.584428, -1.848126]$

[8]:
```
#Função de Himmelblau
def f(x,y):
  return (x**2 + y - 11)**2 + (x+y**2 - 7)**2
```

[9]:
```
#Plotando a Função de Himmelblau e verificando o comportamento dos pontos
functionPlot(title = "Função de Himmelblau")
```



Resolução das tarefas:

1 Conjugado Gradiente

O método do conjugado gradiente resolve o problema de ter valores estreitos, inspirado nos métodos de otimização de funções quadráticas. Onde:

minimize$_x f(x) = \frac{1}{2}x^T Ax + b^T x + c$

Neste, A é simétrico e positivo. Portanto, f tem um mínimo local único.

O método do gradiente conjugado pode otimizar funções quadráticas n-dimensionais em n etapas. Suas direções são mutuamente conjugadas em relação a A:

$d^{(i)T} A\, d^{(j)} = 0$, for all $i \neq j$

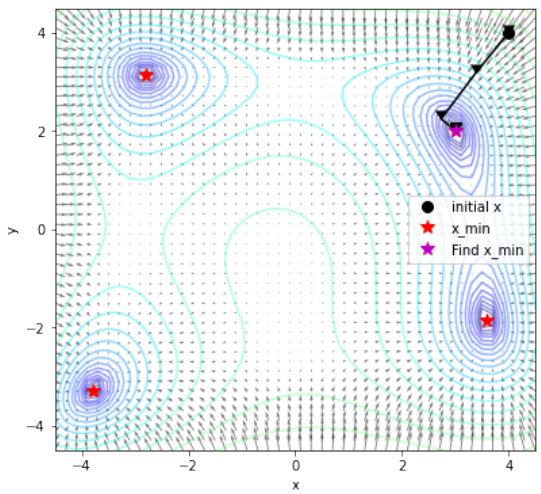
Os vetores mutuamente conjuntos são o vetor base de A, ou seja, são ortogonais entre si.

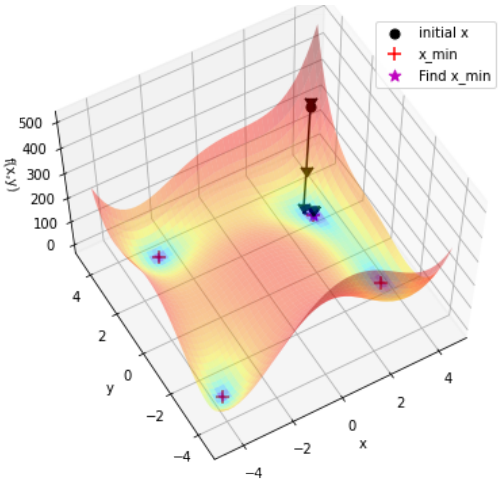
As direções conjugadas sucessivas são calculadas usando as informações da direção e gradiente da etapa anterior ($d^1 = -\nabla f$). No início do método, a direção é um grande passo.

[10]:
```
#Ponto inicial
x0 = [4.0, 4.0]
ps = [x0]
fun = lambda x: (x[0]**2 + x[1] - 11)**2 + (x[0] + x[1]**2 -7)**2
```

[11]:
```
#Implementação do Conjugado Gradiente
t0=time.time()
res = minimize(fun, x0, method='CG', tol=1e-6, callback=reporter)
t1=time.time()
printResult(res, t1-t0, x0)
# Ao executar o conjugado gradiente('CG') acima, obtemos os resultados:
```

```
Ponto Minimo:  [3. 2.]
Valor da Função:  0.0
Número de chamadas:  8
Tempo (ms):  0.0013
```





2 Descida do gradiente com busca em linha

A busca por linhas nos permite selecionar o fator de passo que permite que uma função seja minimizada:

$\text{minimize}_{\alpha} f(x + \alpha\, d)$

A busca por linhas é um problema de otimização invariável. Você pode usar a derivada do alvo da pesquisa de linha, que é simplemente a derivada direcional ao longo de d em $x + \alpha\, d$.

A desvantagem desse método é o custo computacional de otimizar o α com um alto grau de precisão. Portanto, é comum encontrar rapidamente um valor razoável e logo mover em, escolhe x^{k+1} e, em seguida, escolher uma nova direção d^{k+1}.

[12]:
```
#Função que computa o gradiente de f
x = Symbol('x')
y = Symbol('y')
fx = (x**2 + y - 11)**2 + (x + y**2 -7)**2
dx = Derivative(fx, x).doit()
dy = Derivative(fx, y).doit()
print("Derivative of x: ", dx)
print("Derivative of y: ", dy)
```

```
Derivative of x:  4*x*(x**2 + y - 11) + 2*x + 2*y**2 - 14
Derivative of y:  2*x**2 + 4*y*(x + y**2 - 7) + 2*y - 22
```

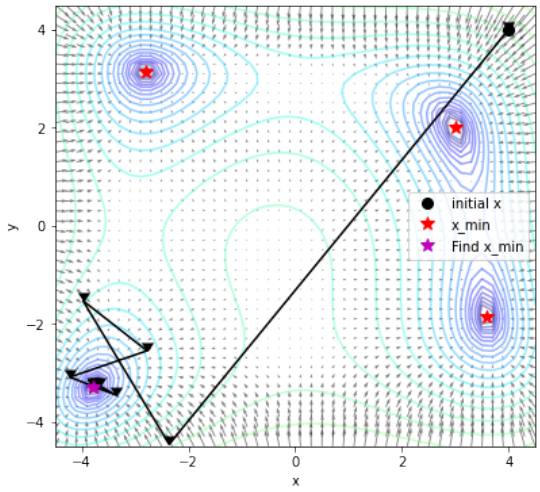
[13]:
```
def obj_func(x):
  return (x[0]**2 + x[1] - 11)**2 + (x[0] + x[1]**2 -7)**2

def obj_grad(x):
  grad_x = 4*x[0]*(x[0]**2 + x[1] - 11) + 2*x[0] + 2*x[1]**2 - 14
  grad_y = 2*x[0]**2 + 4*x[1]*(x[0] + x[1]**2 - 7) + 2*x[1] - 22
  return np.array([grad_x, grad_y])

def error(x_velho, x_novo):
  return np.abs(obj_func(x_novo) - obj_func(x_velho))
```

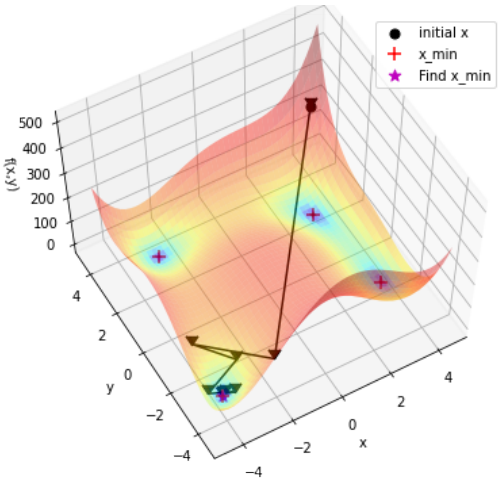
[14]:
```
#Implementação da busca em linha
def buscaLinha(x0, tolerancia):
  global ps
  start_point = np.copy(np.array(x0))
  nro_iter  = 0

  err = 9999
  t0=time.time()
  while (err > tolerancia):
    nro_iter = nro_iter + 1
```

```
    search_gradient = - np.array(obj_grad(start_point)) #Nos vamos caminhar␣
↪para o negativo do gradiente
    res = line_search(obj_func, obj_grad, start_point, search_gradient)
    point = start_point + res[0] * search_gradient #Fazendo passo grande
    err = error(start_point, point)
    start_point = point
    ps.append(start_point)
  t1=time.time()

  r = Reporter(start_point, obj_func(start_point), nro_iter, t1-t0 )
  return r
```

[15]:
```
x0 = np.array([4,4])
res = buscaLinha(x0, var_tolerancia)
printResult(res, res.t, x0)
# Ao executar a busca em linha acima, obtemos os resultados:
```

```
Ponto Minimo:  [-3.78 -3.28]
Valor da Função:  0.0
Número de chamadas:  11
Tempo (ms):  0.0059
```





3 Nelder-Mead

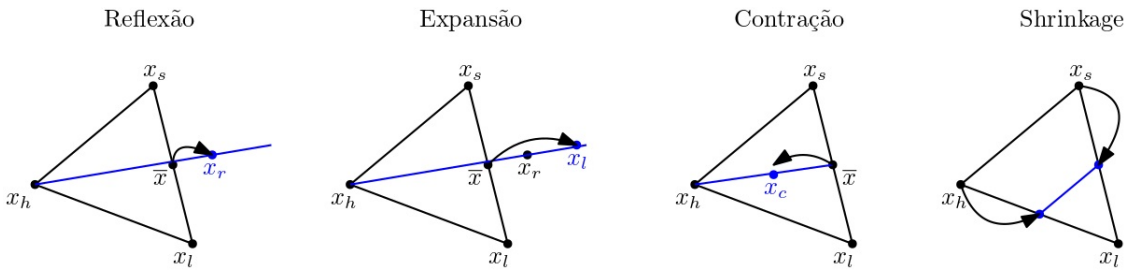
O método Nelder-Mead usa um simplex para percorrer o espaço em busca de um mínimo. Um simplex é a generalização de um tetraedro no espaço n-dimensional. Por exemplo, um simplex em 1D é uma linha e em 2D é um triângulo.

O simplex consiste nos pontos $x^{(1)}, \ldots, x^{(n+1)}$. Em que x_h é o vértice com o valor de função mais alto, x_s é o vértice com o segundo valor de função mais alto e x_l o vértice com o valor de função mais baixo. Dado que $\overline{x}$ é a média de todos os vértices, exceto o ponto mais alto (x_h).

O método Nelder-Mead usa uma série de regras que ditam como o simplex é atualizado com base nas avaliações da função objetivo em seus vértices. Em uma iteração, quatro operações simplex são avaliadas:

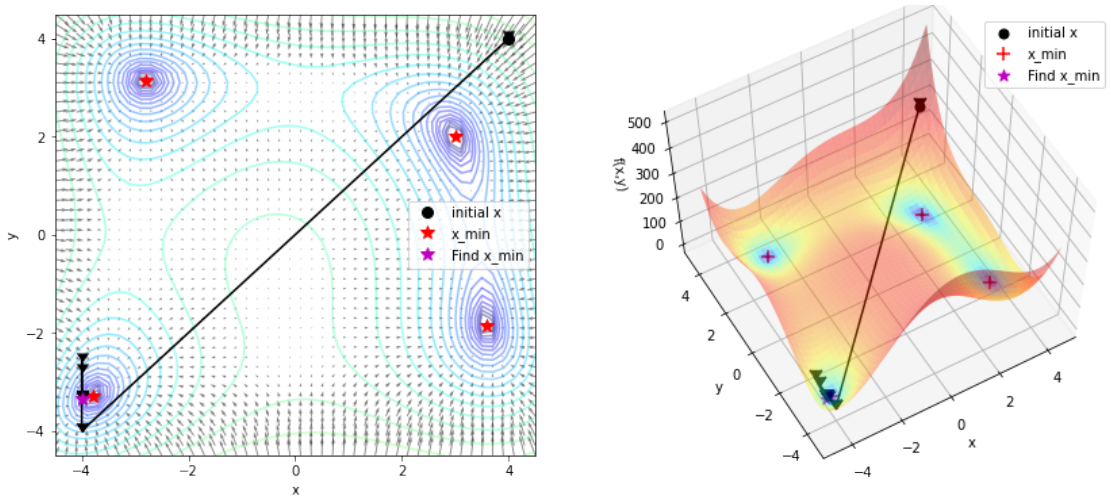
- Reflexão: Reflete o ponto de maior valor no centróide. $x_r = \overline{x} + \alpha(\overline{x} - x_h),\ \alpha > 0$
- Expansão: Isso é feito quando o ponto refletido tem um valor de função objetivo menor que todos os pontos no simplex. $x_e = \overline{x} + \beta(x_r - \overline{x}),\ \beta > \max(1, \alpha)$
- Contração: O simplex se afasta do pior ponto. $x_c = \overline{x} + \gamma(x_h - \overline{x}),\ \gamma \in (0, 1)$
- Shrinkage: Todos os pontos se movem em direção ao melhor ponto, geralmente reduzindo pela metade a distância de separação.

[16]:
```
imagem = cv2.imread("drive/MyDrive/Algebra Lineal/simplex.jpg")
cv2_imshow(imagem)
```



[17]:
```
#Implementação do Nelder-Mead
x0 = [4,4]
triangle0 = [[-4,-4], [-4,1], [-4,-1]]
t0=time.time()
res = minimize(fun, x0, method="Nelder-Mead", options={'initial_simplex':␣
 ↪triangle0}, callback=reporter)
t1=time.time()
printResult(res, t1-t0, x0)
# Ao executar o Nelder-Mead('Nelder-Mead') acima, obtemos os resultados:
```

```
Ponto Minimo:  [-4.   -3.35]
Valor da Função:  2.77
Número de chamadas:  28
Tempo (ms):  0.002
```



4 BFGS

O método BFGS é um método quase-Newton. Portanto, primeiro definiremos os métodos quase-Newton para entender esse método. O método quase-Newton aproxima o inverso de Hessian. As atualizações deste método têm a seguente forma:

$x^{(k+1)} \leftarrow x^{(k)} - \alpha^k \, Q^{(k)} \, g^{(k)}$

onde $\alpha^{(k)}$ é um fator escalar escalonado e $Q^{(k)}$ se aproxima do inverso do Hessiano em $x^{(k)}$.

Os métodos quase Newton geralmente definem Q na matriz de identidade e, em seguida, aplicam atualizações para refletir as informações aprendidas com cada iteração. As equações dos vários métodos são definidas como:

$\gamma^{(k+1)} \equiv g^{(k+1)} - g^{(k)}$
$\delta^{(k+1)} \equiv x^{(k+1)} - x^{(k)}$

Um dos métodos é o Davidon-Fletcher-Powell (DFP), onde usa:

$$Q \leftarrow Q - \frac{Q\,\gamma\,\gamma^T Q}{\gamma^T Q\,\gamma} + \frac{\delta\,\delta^T}{\delta^T\,\gamma}$$

Uma alternativa para o DFP é o método *Broyeden-Fletcher-Goldfarb-Shanno* **(BFGS)**, que usa:

$$Q \leftarrow Q - \frac{\delta\,\gamma^T + Q\,\gamma\,\delta^T}{\delta^T\,\gamma} + \left(1 + \frac{\gamma^T Q\,\gamma}{\delta^T\,\gamma}\right)\frac{\delta\,\delta^T}{\delta^T\,\gamma}$$

O **BFGS** funciona melhor do que o DFP com aproximação da busa em linha, mas ainda usa uma matriz $n \times n$ densa. Portanto, tem uma desvantagem para grandes problemas em que o armazenamento é uma preocupação, pois os BFGS têm memória limitada. **L-BFGS** é uma melhora, já que salva os últimos m valores em vez do inverso completo de Hessian.

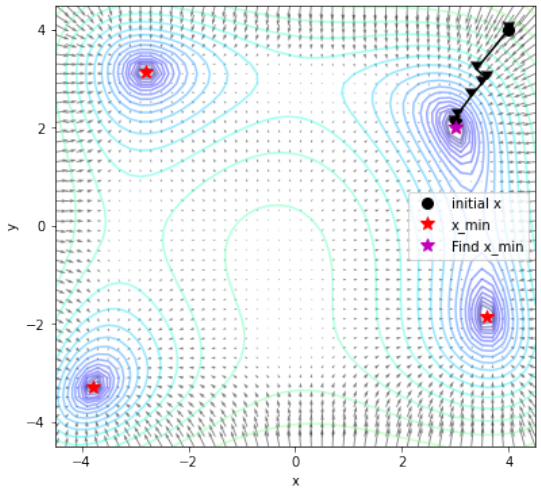
Abaixo, tem-se dois modos de implementação, o primeiro considera a função gradiente e o segundo não considera.

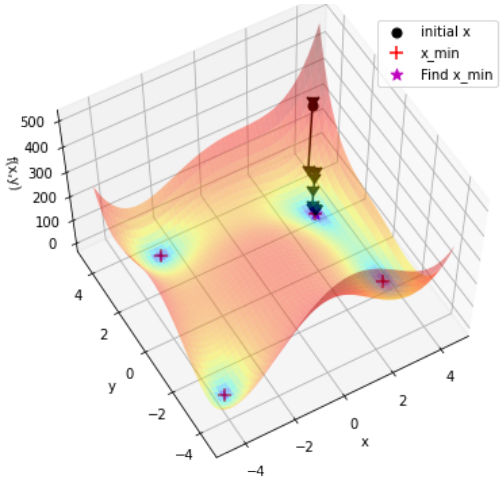
4.1 Passando a função do gradiente

4.1.1 BFGS

[18]:
```
#Implementação do BFGS com função gradiente
x0 = [4,4]
t0=time.time()
res = minimize(obj_func, x0, method="BFGS", jac=obj_grad, callback=reporter)
t1=time.time()
printResult(res, t1-t0, x0)
# Ao executar o BFGS('BFGS') com a função de gradiente('jac=obj_grad') acima,␣
 ↪obtemos os resultados:
```

```
Ponto Minimo:  [3. 2.]
Valor da Função:  0.0
Número de chamadas:  13
Tempo (ms):  0.0016
```



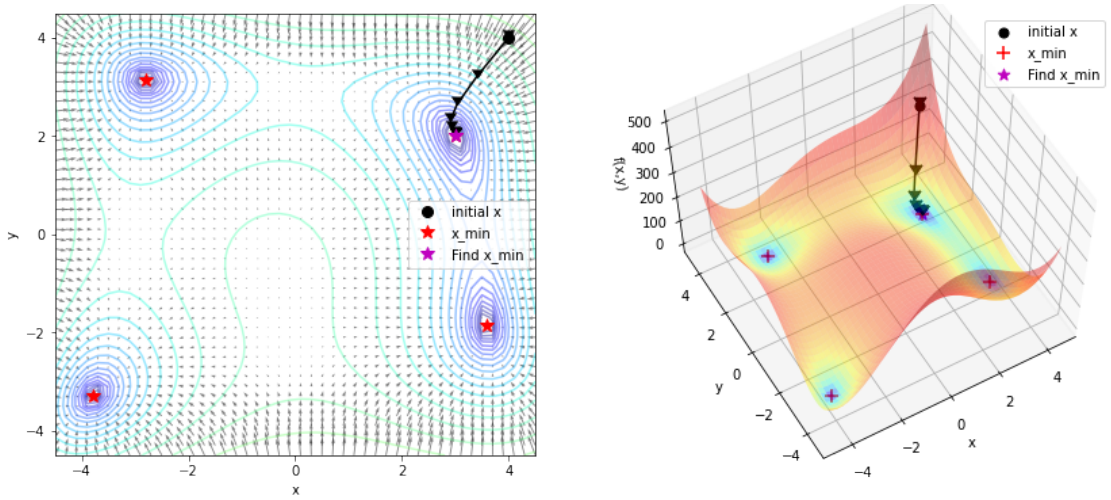


4.1.2 L-BFGS-B

[19]:
```
#Implementação do L-BFGS-B com função gradiente
x0 = [4,4]
t0=time.time()
res = minimize(obj_func, x0, method="L-BFGS-B", jac=obj_grad, callback=reporter)
t1=time.time()
printResult(res, t1-t0, x0)
# Ao executar o LBFGS('L-BFGS-B') com a função de gradiente('jac=obj_grad')␣
 ↪acima, obtemos os resultados:
```

```
Ponto Minimo:  [3. 2.]
Valor da Função:  0.0
```

```
Número de chamadas:  9
Tempo (ms):  0.0051
```

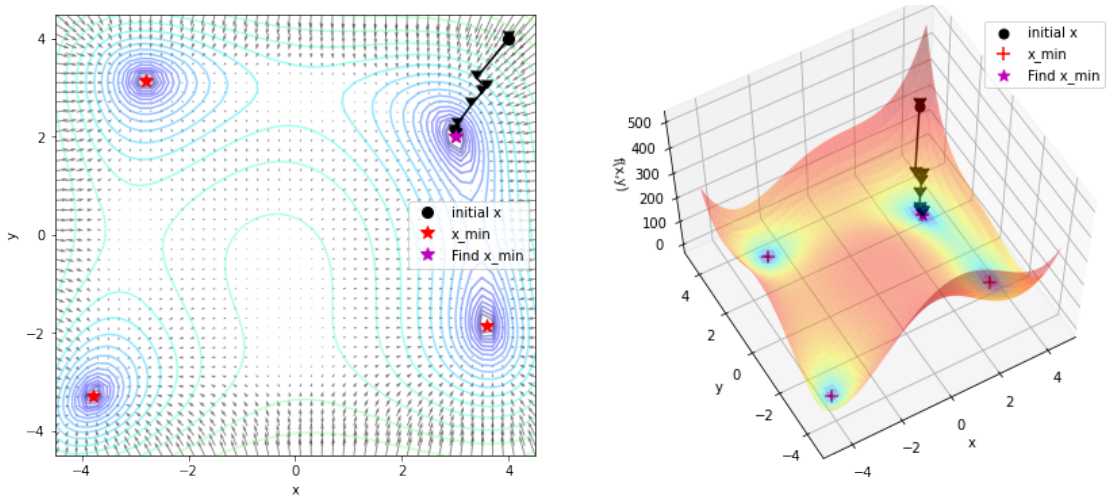


4.2 Sem a função gradiente

4.2.1 BFGS

[20]:
```
#Implementação do BFGS sem  função gradiente
x0 = [4,4]
t0=time.time()
res = minimize(obj_func, x0, method="BFGS", callback=reporter)
t1=time.time()
printResult(res, t1-t0, x0)
# Ao executar o BFGS('BFGS') acima, sem a função de gradiente, obtemos os␣
 ↪resultados:
```

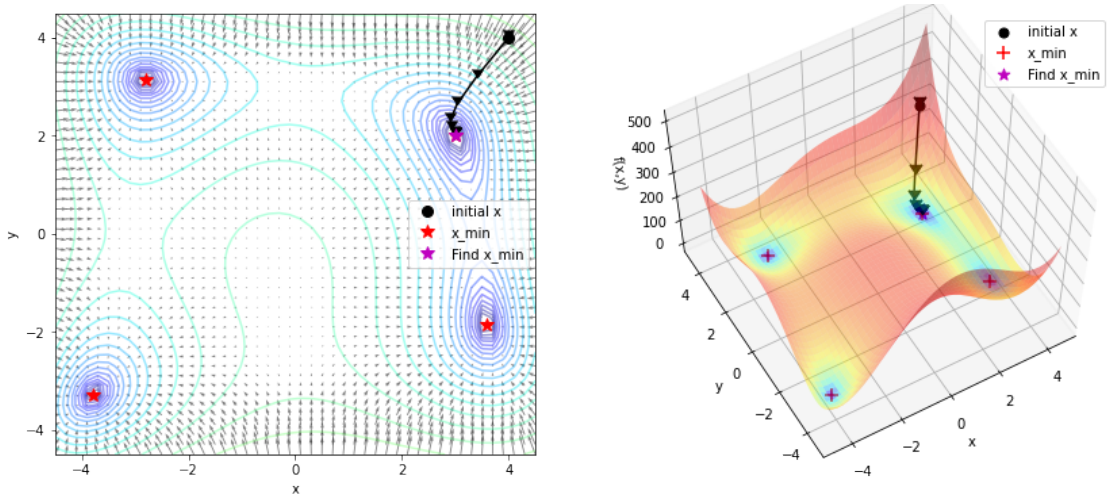
```
Ponto Minimo:  [3. 2.]
Valor da Função:  0.0
Número de chamadas:  13
Tempo (ms):  0.0053
```



4.2.2 L-BFGS-B

[21]:
```
#Implementação do L-BFGS-B sem função gradiente
x0 = [4,4]
t0=time.time()
res = minimize(obj_func, x0, method="L-BFGS-B", callback=reporter)
t1=time.time()
printResult(res, t1-t0, x0)
# Ao executar o LBFGS('L-BFGS-B')acima, sem a função de gradiente, obtemos os␣
 ↪resultados:
```

```
Ponto Minimo:  [3. 2.]
Valor da Função:  0.0
Número de chamadas:  9
Tempo (ms):  0.0015
```



5 NEWOA ou BOBYQA

NEWUOA e BOBYQA são métodos derivados do método de Powell. O método de Powell procura direções que não são ortogonais entre si. O método se ajusta automaticamente para vales longos e estreitos. O algoritmo mantém uma lista de endereços de pesquisa $u^{(1)}, \ldots, u^{(n)}$, que são inicialmente o valor da coordenada de base, $u^{(i)} = e^{(i)}$ for all i.

O algoritmo realizará uma busca em linha para cada direção de busca em sucessão, atualizando o ponto em cada iteração:

$x^{(i+1)} \leftarrow \text{lineSearch}(f, x^{(i)}, u^{(i)})$ for all $i \in \{1, \ldots, n-1\}$

Em seguida, todas as direções de busca são roladas para baixo em um índice, removendo o endereço de pesquisa mais antigo ($u^{(1)}$):

$u^{(i)} \leftarrow u^{(i+1)}$ for all $i \in \{1, \ldots, n-1\}$

A última direção de busca é substituída pela direção de $x^{(1)}$ a $x^{(n+1)}$, que é a direção geral de progresso durante o último ciclo:

$u^{(n)} \leftarrow u^{(n+1)} - x^{(1)}$

Depois, e feita outra búsqueda de linha ao longo da nova direção para obter um novo $x^{(1)}$. Este processo é repetido até a convergência.

NEWUO é um algoritmo iterativo e sua vantagem esta na região de confiança. Em cada iteração, o algoritmo estabelece uma função modelo $Q^{(k)}$ por interpolação quadrática e, em seguida, minimiza $Q^{(k)}$ dentro de uma região de confiança.

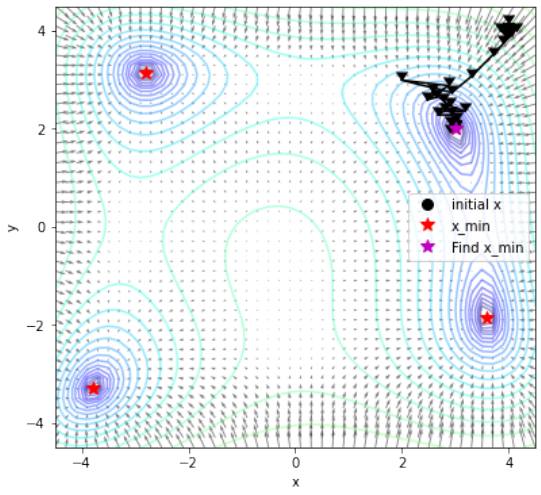
NEWUOA é a técnica de atualização mínima do padrão Frobenius. Suponha que a função objetivo f tenha n variáveis e se deseje determinar a forma única do modelo quadrático o $Q^{(k)}$ interpolando os valores da função f, então é necessário avaliar f em $\frac{(n+1)(n+2)}{2}$ pontos. Portanto, não é prático quando o problema é grande, porque os valores da função são considerados caros na otimização sem derivadas. Assim, NEWUOA modela $Q^{(k)}$ interpola apenas m (um número inteiro entre $n+2$ e $\frac{(n+1)(n+2)}{2}$, normalmente da ordem n) valores de função de f, e os graus de liberdade restantes são tomados minimizando a norma Frobenius de $\delta^2 Q^{(k)} - \delta^2 Q^{(k)}$. Esta técnica imita as atualizações do método da secante para métodos quase-Newton e pode ser considerada como a versão livre de derivativos da atualização PSB (atualização Simétrica Broyden de Powell).

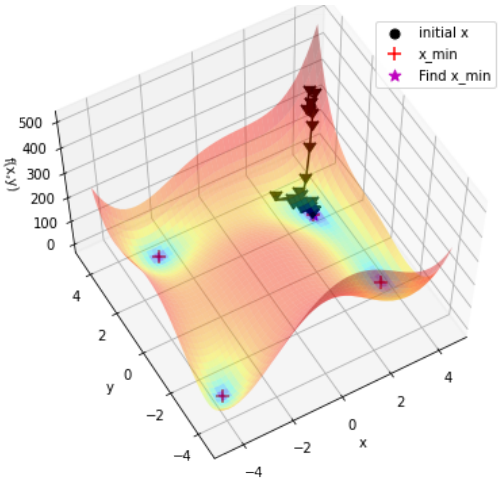
5.1 NEWOA

[22]:
```
iter = 0

def myfunc(x, grad):
  global iter, ps
  fpto = (x[0]**2 + x[1] - 11)**2 + (x[0] + x[1]**2 -7)**2
  if grad.size > 0:
    grad[0] = 4*x[0]*(x[0]**2 + x[1] - 11) + 2*x[0] + 2*x[1]**2 - 14
    grad[1] = 2*x[0]**2 + 4*x[1]*(x[0] + x[1]**2 - 7) + 2*x[1] - 22
  iter = iter + 1
  ps.append(np.copy(x))
  return fpto
```

```
x0 = np.array([4,4])

# Escolher o método e dimensão
opt = nlopt.opt(nlopt.LN_NEWUOA, 2)
# Passo do método da função que faz a função e o gradiente
opt.set_min_objective(myfunc)
opt.set_upper_bounds([4.2, 4.2])
# Definir a tolerancia
opt.set_xtol_rel(var_tolerancia)

# Definir o optimizador
t0=time.time()
res = opt.optimize(x0)
t1=time.time()
r = Reporter([res[0], res[1]], f(res[0], res[1]), iter, t1-t0)

printResult(r, r.t, x0)
# Ao executar o NEWOA acima, obtemos os resultados:
```

```
Ponto Minimo:  [3. 2.]
Valor da Função:  0.0
Número de chamadas:  47
Tempo (ms):  0.0015
```

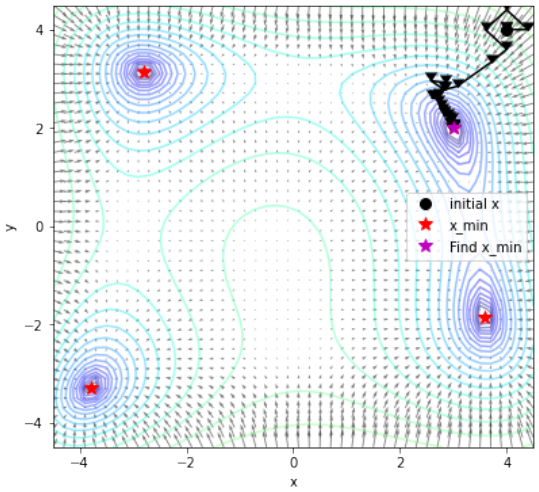


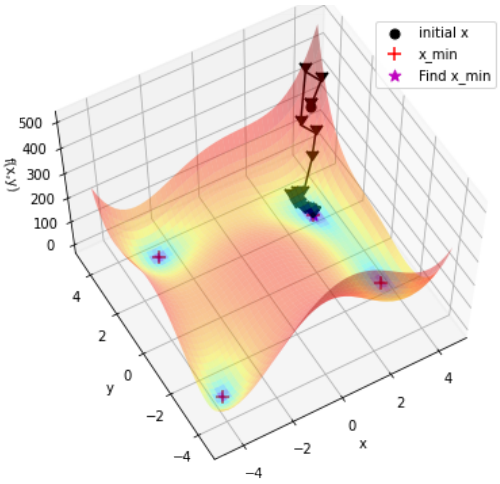


5.2 BOBYQA

[23]:
```
def funBoby(x):
  global ps
  r = (x[0]**2 + x[1] - 11)**2 + (x[0] + x[1]**2 - 7)**2
  ps.append(np.copy(x))
  return r

x0 = [4, 4]

t0=time.time()
res = pybobyqa.solve(funBoby,x0)
t1=time.time()
r = Reporter(res.x, fun(res.x), res.nf, t1-t0)
printResult(r, r.t, x0)

# Ao executar o BOBYQA ('pybobyqa') acima, obtemos os resultados:
```

```
Ponto Minimo:  [3. 2.]
Valor da Função:  0.0
Número de chamadas:  58
Tempo (ms):  0.1492
```





6 Bibliografía

- Kochenderfer, M. J., & Wheeler, T. A. (2019). Algorithms for optimization. Mit Press.
- Powell, M. J. (2004). Least Frobenius norm updating of quadratic models that satisfy interpolation conditions. Mathematical Programming, 100(1), 183-215.
- Powell, M. J. D. (2013). Beyond symmetric Broyden for updating quadratic models in minimization without derivatives. Mathematical Programming, 138(1), 475-500.