Optmization

December 8, 2017

1 Projeto Final - CK0031

1.1 Parte 1 - Otimização

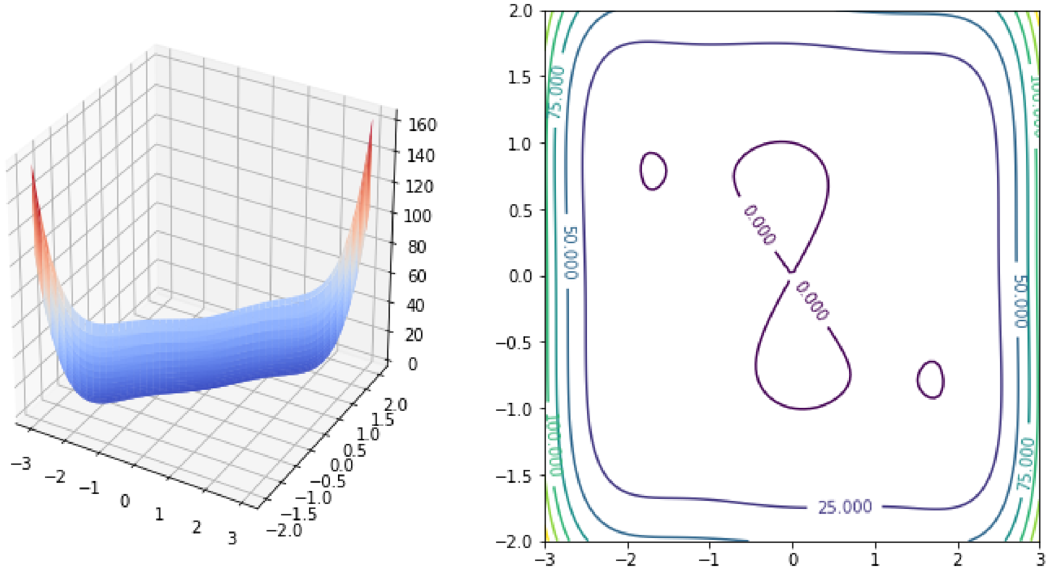
1.1.1 Aluno:

- Marcos Felipe de Menezes Mota - 354080
- Vilma Bezerra Alves-354094 #### Link GitHub do relatório: https://github.com/vilmabezerra/AI-assignment/blob/mf/Optmization.ipynb

Objetivo: Minimizar a função objetiva $f(x) = (4 - 2.1x_1^2 + \frac{x_1^4}{3})x_1^2 + x_1x_2 - 4(1 - x_2^2)x_2^2$

In [1]:
```
%matplotlib inline

import matplotlib
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import cm
from mpl_toolkits.mplot3d import Axes3D


def f (x,y):
    part1 = ((4 -(2.1*(x**2))) + ((1/3.0)*(x**4)))*(x**2)
    part2 = (x*y) - (4*(1 - y**2))*(y**2)
    return part1 + part2


fig = plt.figure(figsize=(12,6))

ax = fig.add_subplot(1, 2, 1, projection='3d')

# p = ax.plot_wireframe(X, Y, Z)

X = np.linspace(-3, 3, 100)
Y = np.linspace(-2, 2, 100)
xv, yv = np.meshgrid(X, Y)
vf = np.vectorize(f)
```

```
Z = vf(x=xv, y=yv)
surf = ax.plot_surface(X, Y, Z, cmap=cm.coolwarm, linewidth=0)

ax = fig.add_subplot(1, 2, 2)
cp = plt.contour(X, Y, Z)
plt.clabel(cp, inline=1, fontsize=10)



plt.show()
```



Analisando o mapa de contorno podemos ver que os valores da função crescem vertigiosamente fora do intervalo $x_1 \in [-2, 2]$ e $x_2 \in [-1.5, 1.5]$ assim facilitando limitando a escolha de um bom x0 ao escopo desse intervalo. Além disso a função tem 4 mínimos locais onde dois deles são por volta do ponto $[0, 0]$

1.1.2 Cálculo do Gradiente e da Hessiana

Para a implementação e verificação dos métodos de otimização precisamos do vetor gradiente e da matrix Hessiana.

Inicialmente vamos simplificar a função objetiva.

$$f(x) = 4x_1^2 - 2.1x_1^4 + \frac{x_1^6}{3} + x_1x_2 - 4x_2^2 + 4x_2^4$$

Depois calculamos o gradiente e a Hessiana

$$\nabla f = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \end{bmatrix} = \begin{bmatrix} 8x_1 - 8.4x_1^3 + 2x_1^5 + x_2 \\ 16x_2^3 + x_1 - 8x_2 \end{bmatrix}$$

Dado que em ambas as funções do gradiente depende apenas da outra variável apenas de forma aditiva simples. As derivadas parciais cruzadas são igual a 1. Tirando a derivada segunda do gradiente, temos:

$$\nabla^2 f = \begin{bmatrix} 10x_1^4 - 25.2x_1^2 + 8 & 1 \\ 1 & 48y^2 - 8 \end{bmatrix}$$

1.1.3 Implementando a otimização com direção gradiente

In [2]:
```
import math
def f_grad(vec):
    x = vec[0]
    y = vec[1]
    def f_grad_x(x, y):
        return 8*x - 8.4*(x**3) + 2*(x**5) + y
    def f_grad_y(x, y):
        return 16*(y**3) + x - 8*y
    f_g = np.array([f_grad_x(x, y), f_grad_y(x, y)])
    return f_g


def optimize(fun, x0, fun_grad, step_size=0.01,max_steps=100, tolerance=0.001):

    xk = x0
    xk1 = math.inf
    res = [(x0, fun(x0[0], x0[1]))]
    for _ in range(max_steps):
        dx = -f_grad(xk)
        xk1 = xk + step_size*dx
        if (math.fabs(xk1[0] - xk[0]) or math.fabs(xk1[1] - xk[1])) < tolerance:
            break

        xk = xk1
        res += [(xk, fun(xk[0], xk[1]))]

    return res
```

In [3]:
```
def pretty_print(opt_array):

    print("|           x           |    f(x)   |")
    print("------------------------------------")
    for i in range(len(opt_array)):
        print("|{0:2f} , {1:3f}  | {2:4f} |".format(opt_array[i][0][0], opt_array[i][0
        if i >= 9:
            print("|          .          |     .     |")
            print("|          .          |     .     |")
            print("|          .          |     .     |")
            print("|{0:2f} , {1:3f}  | {2:4f} |".format(opt_array[-1][0][0], opt_array
            break
```

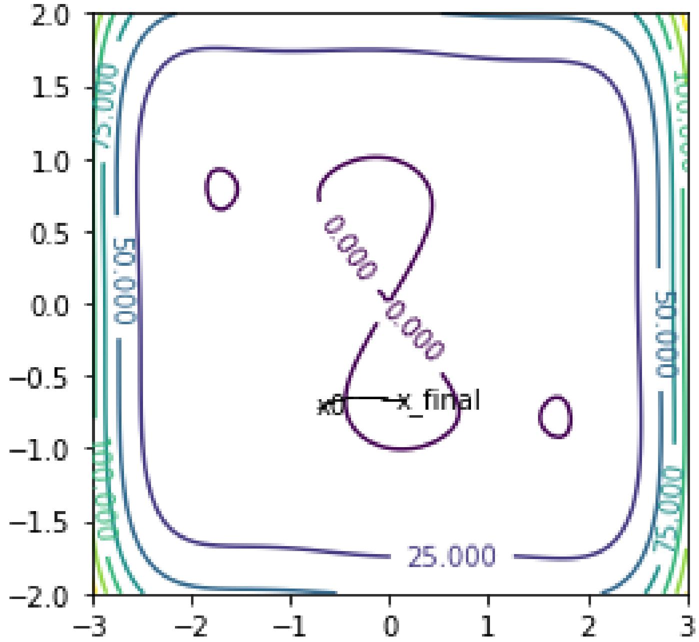
```
print("# steps: ", len(opt_array) )
```

```
In [4]: x0 = np.array([-0.75, -0.75])
        opt_h = optimize(f, x0, f_grad)
        pretty_print(opt_h)
```

```
|          x           |    f(x)   |
------------------------------------
|-0.750000 , -0.750000  | 1.222998 |
|-0.713191 , -0.735000  | 1.065796 |
|-0.675568 , -0.723138  | 0.910469 |
|-0.637376 , -0.713729  | 0.756029 |
|-0.598895 , -0.706281  | 0.602914 |
|-0.560423 , -0.700424  | 0.452363 |
|-0.522265 , -0.695874  | 0.305994 |
|-0.484714 , -0.692406  | 0.165504 |
|-0.448044 , -0.689838  | 0.032449 |
|-0.412496 , -0.688020  | -0.091904 |
|          .           |     .     |
|          .           |     .     |
|          .           |     .     |
|0.077192 , -0.711178  | -1.031004 |
# steps:  56
```

1.1.4 Plot da linha de progressão do algoritmo no mapa de contorno

```
In [5]: fig = plt.figure(figsize=(4,4))
        ax = fig.add_subplot(1, 1, 1)
        cp = plt.contour(X, Y, Z)
        plt.clabel(cp, inline=1, fontsize=10)

        ax.annotate("x0", xy=opt_h[0][0])
        for i in range(len(opt_h)):
            ax.annotate("-", xy=opt_h[i][0])
        ax.annotate("x_final", xy=opt_h[-1][0])
        plt.show()
```




```
In [6]: x0 = np.array([0.75, 0.75])
        opt_h = optimize(f, x0, f_grad)
        pretty_print(opt_h)
```

```
|           x           |    f(x)   |
------------------------------------
|0.750000 , 0.750000  | 1.222998 |
|0.713191 , 0.735000  | 1.065796 |
|0.675568 , 0.723138  | 0.910469 |
|0.637376 , 0.713729  | 0.756029 |
|0.598895 , 0.706281  | 0.602914 |
|0.560423 , 0.700424  | 0.452363 |
|0.522265 , 0.695874  | 0.305994 |
|0.484714 , 0.692406  | 0.165504 |
|0.448044 , 0.689838  | 0.032449 |
|0.412496 , 0.688020  | -0.091904 |
|          .          |     .     |
|          .          |     .     |
|          .          |     .     |
|-0.077192 , 0.711178  | -1.031004 |
# steps:  56
```

```
In [7]: x0 = np.array([-1, -0.5])
        opt_h = optimize(f, x0, f_grad)
        pretty_print(opt_h)
```

```
|           x          |    f(x)   |
-----------------------------------
|-1.000000 , -0.500000  | 1.983333 |
|-0.979000 , -0.510000  | 1.927660 |
|-0.956412 , -0.519786  | 1.865310 |
|-0.932184 , -0.529335  | 1.795564 |
|-0.906281 , -0.538629  | 1.717745 |
|-0.878692 , -0.547654  | 1.631273 |
|-0.849432 , -0.556399  | 1.535729 |
|-0.818552 , -0.564856  | 1.430927 |
|-0.786140 , -0.573023  | 1.316988 |
|-0.752324 , -0.580899  | 1.194400 |
|          .          |     .     |
|          .          |     .     |
|          .          |     .     |
|0.077702 , -0.711219  | -1.031053 |
# steps:  66
```

1.1.5 Os exemplos acima são exemplos onde a variação dos valores de x são adequadas e o método converge para um dos mínimos da função

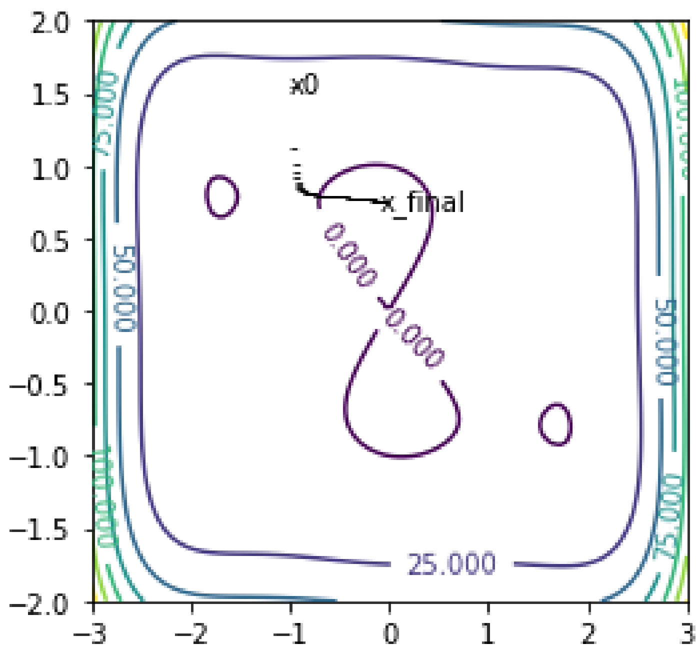
In [8]:
```
x0 = np.array([-1, 1.5])
opt_h = optimize(f, x0, f_grad)
pretty_print(opt_h)
```

```
|           x          |    f(x)   |
-----------------------------------
|-1.000000 , 1.500000  | 11.983333 |
|-0.999000 , 1.090000  | 2.036746 |
|-0.993828 , 0.979985  | 1.097152 |
|-0.987186 , 0.917738  | 0.774788 |
|-0.979449 , 0.877356  | 0.630689 |
|-0.970766 , 0.849283  | 0.554935 |
|-0.961201 , 0.828922  | 0.509226 |
|-0.950782 , 0.813718  | 0.477572 |
|-0.939514 , 0.802116  | 0.452450 |
|-0.927395 , 0.793109  | 0.429989 |
|          .          |     .     |
|          .          |     .     |
|          .          |     .     |
|-0.102274 , 0.714047  | -1.031029 |
# steps:  78
```

1.1.6 Plot da linha de progressão do algoritmo no mapa de contorno

In [9]:
```
fig = plt.figure(figsize=(4,4))
ax = fig.add_subplot(1, 1, 1)
```

```
cp = plt.contour(X, Y, Z)
plt.clabel(cp, inline=1, fontsize=10)

ax.annotate("x0", xy=opt_h[0][0])
for i in range(len(opt_h)):
    ax.annotate("-", xy=opt_h[i][0])
ax.annotate("x_final", xy=opt_h[-1][0])
plt.show()
```



1.1.7 No exemplos abaixo o algoritmo está convergindo muito devagar, logo fica abaixo da tolerância de variação de x

```
In [10]: x0 = np.array([-1, 1.5])
         opt_h = optimize(f, x0, f_grad, max_steps=100, tolerance=0.01)
         pretty_print(opt_h)
```

```
|           x           |    f(x)   |
-------------------------------------
|-1.000000 , 1.500000  | 11.983333 |
# steps:  1
```

```
In [11]: x0 = np.array([0, 0])
         opt_h = optimize(f, x0, f_grad,max_steps=100, tolerance=0.00001)
         pretty_print(opt_h)
```

```
|           x          |    f(x)   |
------------------------------------
|0.000000 , 0.000000  | 0.000000 |
# steps:  1
```

No caso específico de x = [0, 0] não importa aumentar a tolerância pois é um saddle point

1.2 Parte 2 - Probabilidade

1.2.1 Dado o contexto do problema temos as seguintes informações sobre as probabilidades:

- $p(B, E, A, R, C) = p(B)p(E)p(A|B, E)p(C|A)p(R|E)$
  Variável
  Valor
  $p(B = 1)$
  $\beta = 0.001$
  $p(E = 1)$
  $\epsilon = 0.001$
  α_b
  $\epsilon = 0.99$
  α_e
  $\epsilon = 0.01$
  f
  $\epsilon = 0.001$
  $p(A = 0|B = 0, E = 0)$
  $(1 - f)$
  $p(A = 0|B = 1, E = 0)$
  $(1 - f)(1 - \alpha_b)$
  $p(A = 0|B = 0, E = 1)$
  $(1 - f)(1 - \alpha_e)$
  $p(A = 0|B = 1, E = 1)$
  $(1 - f)(1 - \alpha_e)(1 - \alpha_b)$

A) Encontrar $P(B = 1|C = 1)$ - Pela definição de probabilidade condicional temos:

$$P(B = 1|C = 1) = \frac{P(B = 1, C = 1)}{P(C = 1)}$$

- Podemos marginalizar as probabilidade de tal forma que:

$$P(B = 1|C = 1) = \frac{\Sigma_E \Sigma_A \Sigma_R P(B = 1, E, A, R, C = 1)}{\Sigma_E \Sigma_A \Sigma_R \Sigma_B P(B, E, A, R, C = 1)}$$

- Substituindo pela fatoração da distribuição conjunta temos:

$$P(B = 1|C = 1) = \frac{\Sigma_E \Sigma_A \Sigma_R p(B = 1)p(E)p(A|B = 1, E)p(C = 1|A)p(R|E)}{\Sigma_E \Sigma_A \Sigma_R \Sigma_B p(B)p(E)p(A|B, E)p(C = 1|A)p(R|E)}$$

- Como R aparece apenas em $P(R|E)$ e varia de forma livre tanto no numerador como denominador podemos simplificar a probabilidade para:

$$P(B = 1|C = 1) = \frac{\Sigma_E \Sigma_A p(B = 1)p(E)p(A|B = 1, E)p(C = 1|A)}{\Sigma_E \Sigma_A \Sigma_B p(B)p(E)p(A|B, E)p(C = 1|A)}$$

- Expandindo os somatórios, usando o fato que p(C=1|A=0)=0 e algumas simplificações, chegamos a seguinte expressão probabilistica:

$$P(B=1|C=1) = \frac{P(B=1)(P(E=0)P(A=1|B=1,E=0)+I}{P(B=0)P(E=0)P(A=1|B=0,E=0)+P(B=1)P(E=0)P(A=1|B=1,E=0)+P(B=}$$

- Esses valores estão definidos na tabela acima e podemos substituir em termos de $\alpha_e, \alpha_b, f, \beta, \epsilon$. Portanto obtemos a expressão:

$$P(B=1|C=1) = \frac{\beta((1-\epsilon)(1-(1-f)(1-\alpha_b))+\epsilon(1-(1-f)(1-\alpha_b)(1-\alpha_e)}{(1-\beta)(1-\epsilon)f+\beta(1-\epsilon)(1-(1-f)(1-\alpha_b))+(1-\beta)\epsilon(1-(1-f)(1-\alpha_e))+\beta\epsilon(1}$$

- Substituindo os valores de $\alpha_e, \alpha_b, f, \beta, \epsilon$ obtemos o valor da probabilidade:

$$P(B=1|C=1) = \frac{0.0010}{0.0020} = 0.5$$

A) Encontrar $P(B=1|C=1,R=1)$ - Pela definição de probabilidade condicional temos:

$$P(B=1|C=1,R=1) = \frac{P(B=1,C=1,R=1)}{P(C=1,R=1)}$$

- Podemos marginalizar as probabilidade de tal forma que:

$$P(B=1|C=1,R=1) = \frac{\Sigma_E \Sigma_A P(B=1,E,A,R=1,C=1)}{\Sigma_E \Sigma_A \Sigma_B P(B,E,A,R=1,C=1)}$$

- Nesse caso não podemos eliminar um somatório diretamente mas ao expandir os somátorios, as possibilidades que contém $P(C=1|A=1)eP(R=1|E=0)$ são iguais a 0. Dessa forma temos a simplificação a seguir:

$$P(B=1|C=1,R=1) = \frac{P(B=1)P(E=1)P(A=1|B=1,E=1)P(C=}{P(B=1)P(E=1)P(A=1|B=1,E=1)P(C=1|A=1)P(R=1|E=1)+P(B=0)P}$$

- Passando em termos de $\alpha_e, \alpha_b, f, \beta, \epsilon$. Portanto obtemos a expressão:

$$P(B=1|C=1,R=1) = \frac{\beta\epsilon(1-(1-f)(1-\alpha_b)(1-\alpha_e))}{\beta\epsilon(1-(1-f)(1-\alpha_b)(1-\alpha_e))+(1-\beta)\epsilon(1-(1-f)(1-\alpha_e))}$$

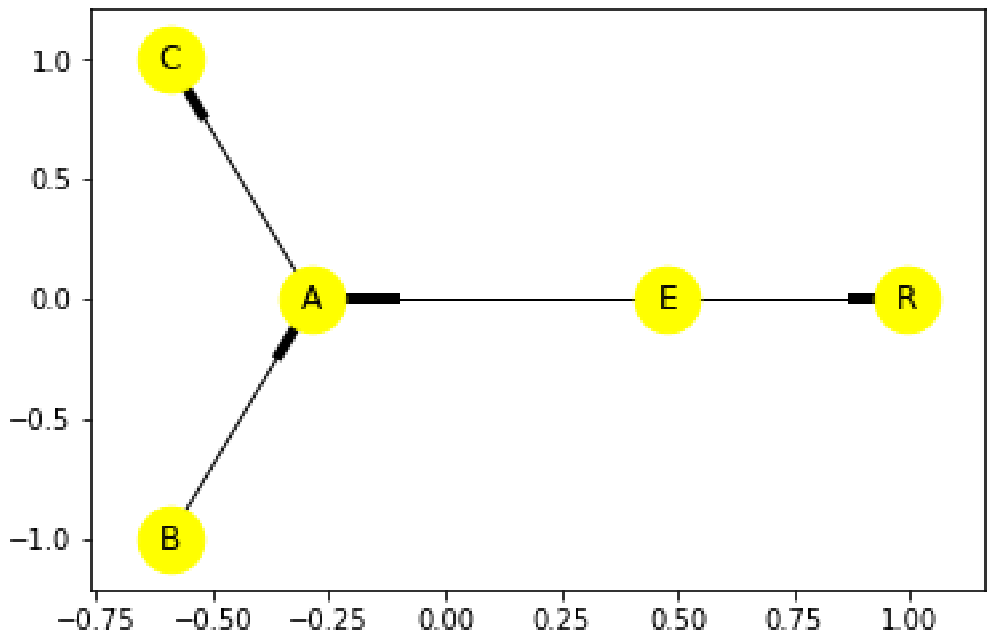
- Substituindo os valores temos:

$$P(B=1|C=1,R=1) = \frac{0.000001}{0.000001+0.00011} = 0.083$$

C) Desenhar a belif network do problema

Dado que temos a fatoração: $p(B,E,A,R,C) = p(B)p(E)p(A|B,E)p(C|A)p(R|E)$ E sabendo que uma belif networks expressa a relações de condicionamento das variáveis temos o seguinte grafo $G(V,E)$ onde $V = \{A,B,C,E,R\}$ e $E = \{(B,A),(E,A),(A,C),(E,R)\}$

In [12]:
```
import networkx as nx

G = nx.DiGraph()
G.add_edges_from([('B','A'), ('E','A'), ('A','C'), ('E', 'R')])
pos = nx.spectral_layout(G)
nx.draw_networkx_nodes(G, pos, cmap=plt.get_cmap('jet'), node_size = 600, node_color=
nx.draw_networkx_labels(G,pos)
nx.draw_networkx_edges(G, pos, arrows=True)
plt.show()
```



D) Indicar o Markov Blanket para cada variável

- $MB(C) = \{A\}$
- $MB(B) = \{A, E\}$
- $MB(A) = \{C, B, E\}$
- $MB(E) = \{A, R, B\}$
- $MB(R) = \{E\}$