Lista de Exercícios 2 **Luis Vinicius Costa Silva**

Otimização Clássica
Prof. Romes Antonio Borges
Data de Entrega: 27/05/2019

Questão 1

Os seguintes métodos de otimização foram implementados:

- Ordem Zero:
  - Busca Aleatória
  - Seção Áurea
  - Fibonacci
  - Polítopo
  - Powell
- Ordem Um:
  - Máxima Descida
  - Gradiente Conjugado
- Ordem Dois:
  - Newton
- Quasi-Newton:
  - BFGS-DFP
- Outros:
  - Interpolação Polinomial

Métodos de Ordem Zero tem como entrada apenas um intervalo de busca, um chute inicial (opcional em alguns casos) e a própria função a ser otimizada. Estes algoritmos utilizam apenas o valor de um determinado X na função a fim de computar o ponto ótimo.

Métodos de Primeira Ordem acrescentam a primeira derivada (ou gradiente no caso de otimização multivariável) a entrada do algoritmo, enquanto que Métodos de Segunda Ordem utilizam-se da segunda derivada (ou hessiana no caso de otimização multivariável).

A derivada de primeira ordem sinaliza o decréscimo ou incremento da função em um determinado ponto, ou seja a derivada primeira ordem serve como uma linha tangente a um ponto em sua superfície de erro. Nos métodos de segunda ordem, a informação da derivada segunda denota o incremento/decremento da primeira derivada, indicando as características da curvatura da função (e.g: côncava/convexa).

Métodos Quasi-Newton são similares aos Métodos de Segunda Ordem, com a diferença de que ao invés de usarem a hessiana da função na regra de atualização, esta é substituída por um termo aproximado, que converge (em situações ideais) para a hessiana exata, geralmente estes métodos são classificados como Métodos de Ordem Zero ou à parte.

O método de Interpolação Polinomial não tem como propósito minimizar uma função diretamente, ao invés disso, este método computa um polinômio que tem como pontos pré-determinados uma entrada do usuário. Tal algoritmo se faz útil em problemas de minimização através da simplificação da função original (ou um intervalo em particular da mesma) em um polinômio, e a busca subsequente do mínimo da mesma através da aplicação de um método de otimização no polinômio computado anteriormente. Neste trabalho, o algoritmo de interpolação polinomial implementado consiste na montagem da matriz de Vandermonde e a aplicação da eliminação da eliminação Gaussiana a fim de obter os coeficientes α do polinômio contidos na dita matriz.

Questão 2

Dada as funções abaixo, foram computados seus respectivos mínimos através do Método de Fibonacci e Método da Seção Áurea:

(a) $F(x_1) = 3x_1^2 - 5x_1$ $\quad 0.0 \leq x_1 \leq 3.0$

(b) $F(x_1) = (x_1 - 3)^2$ $\quad 0.0 \leq x_1 \leq 10$

(c) $F(x_1) = \frac{(\sin(0.1+2x_1))}{(1+x_1)}$ $\quad 0.5 \leq x_1 \leq 3.5$

Sabe-se que os mínimos são respectivamente:

(a) $x_1 = \frac{5}{6}$

(b) $x_1 = 3$

(c) $x_1 \approx 2.22939$

As duas primeiras iterações do método da Seção Áurea são realizadas da seguinte forma:

Iteração 1 do método da Seção Áurea na Função (a):

$$\begin{aligned} \alpha_1 = a_1 + \rho(b_1 - a_1) \approx 0 + 0.3820(3-0) = 1.597 \\ \beta_1 = a_1 + (1-\rho)(b_1 - a_1) \approx 0 + 0.6180(3-0) = 2.584 \\ f(\alpha_1) \approx 0.0 \\ f(\beta_1) \approx 7.1111 \\ f(\beta_1) \geq f(\alpha_1) \\ x^* \in [a, \beta_1] = [0, 2.584] \end{aligned}$$

Iteração 2 do método da Seção Áurea na Função (a):

$$\begin{aligned}
\alpha_2 = a_2 + \rho(b_2 - a_2) \approx 0 + 0.3820(7.1111 - 0) = 0.987 \\
\beta_2 = a_2 + (1-\rho)(b_2 - a_2) \approx 0 + 0.6180(7.1111 - 0) = 1.597 \\
f(\alpha_2) \approx -0.3337 \\
f(\beta_2) \approx 7.1111 \\
f(\beta_2) \geq f(\alpha_2) \\
x^* \in [a, \beta_2] = [0, 1.597]
\end{aligned}$$

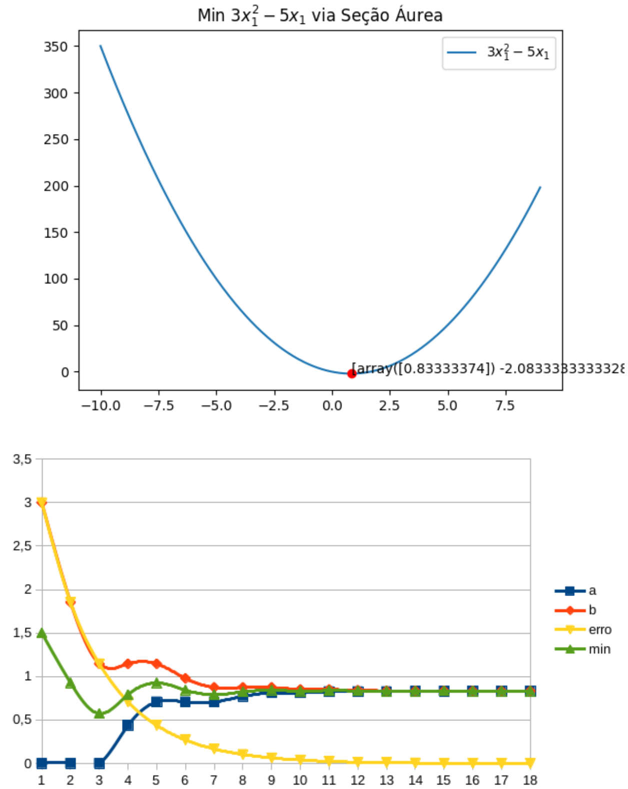
Iteração 1 do método Fibonacci na Função (a):

$$\begin{aligned}
n = 18 \\
F_18/F_17 \approx 0.382 \\
\alpha_1 = a_1 + \rho_1(b_1 - a_1) \approx 1.597 \\
\beta_1 = a_1 + (1-\rho_1)(b_1 - a_1) \approx 2.584 \\
f(\alpha_1) \approx 0.0 \\
f(\beta_1) \approx 7.111 \\
f(\beta_1) \geq f(\alpha_1) \\
x* \in [a, \beta_1] = [0.2.584]
\end{aligned}$$

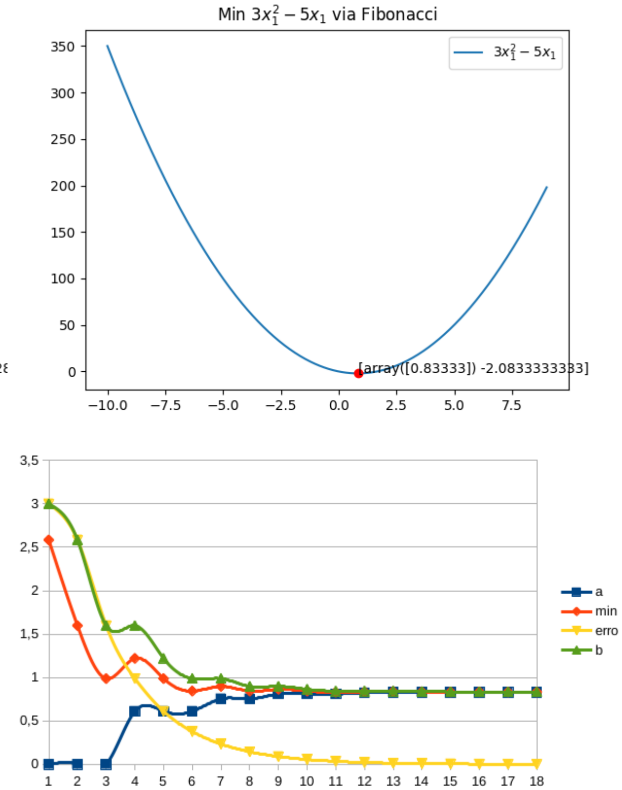
Iteração 2 do método Fibonacci na Função (a):

$$\begin{aligned}
\alpha_2 = a_2 + \rho_2(b_2 - a_2) \approx 0.987 \\
\beta_2 = a_2 + (1-\rho_2)(b_2 - a_2) \approx 1.597 \\
f(\alpha_2) \approx -0.3337 \\
f(\beta_2) \approx 7.1111 \\
f(\beta_2) \geq f(\alpha_1) \\
x* \in [a, \beta_2] = [0.1.597]
\end{aligned}$$

As figuras abaixo representam o gráfico das funções e o mínimo computado por cada algoritmo, assim como detalhes da convergência dos mesmos:

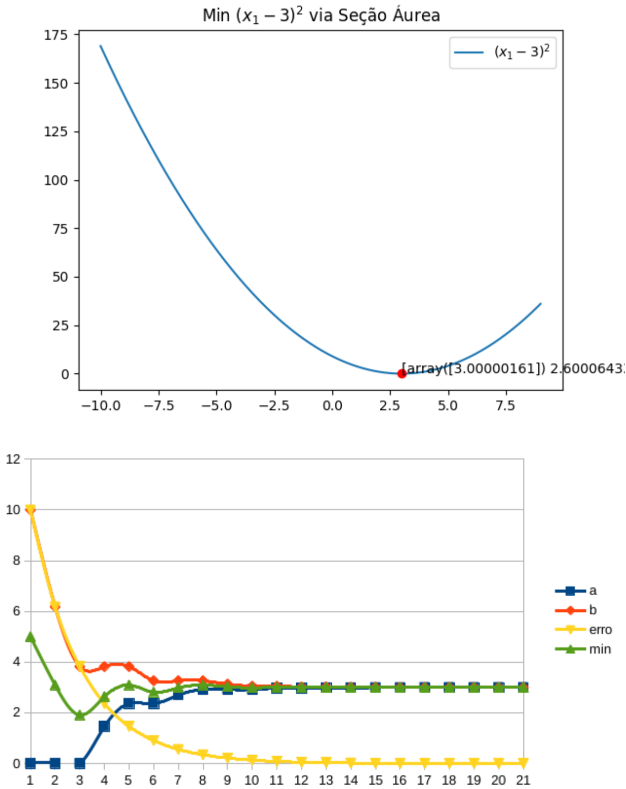


(a) Seção Áurea

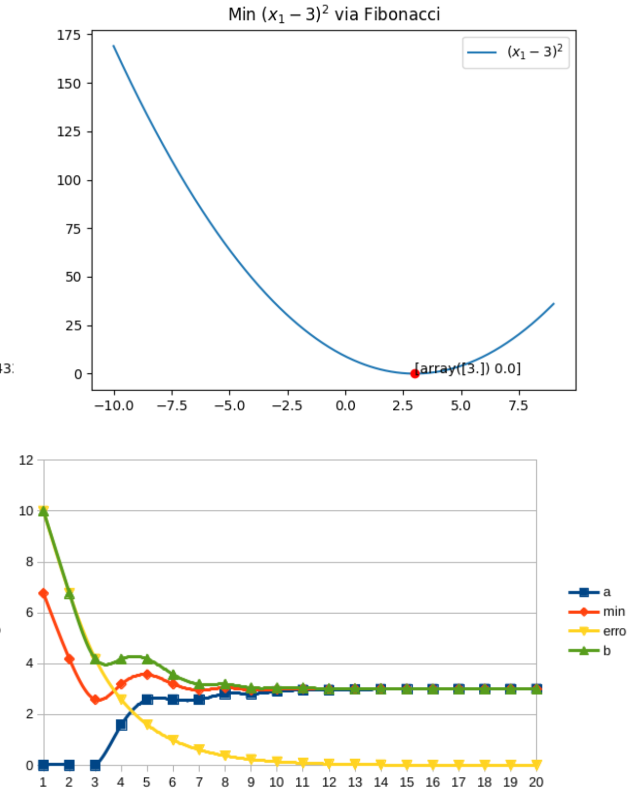


(b) Fibonacci

Figura 1: Questão 2 – Letra A - Mínimos Computados e convergência dos algoritmos

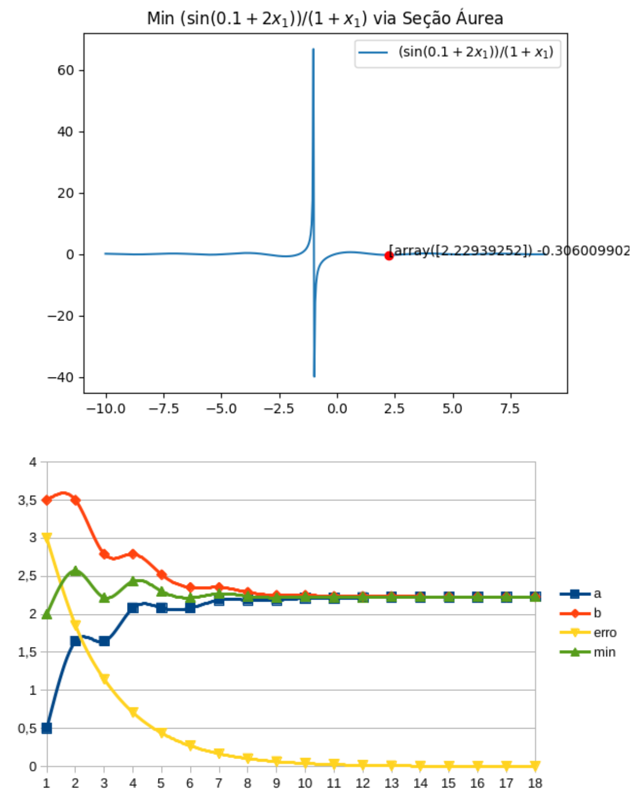


(a) Seção Áurea

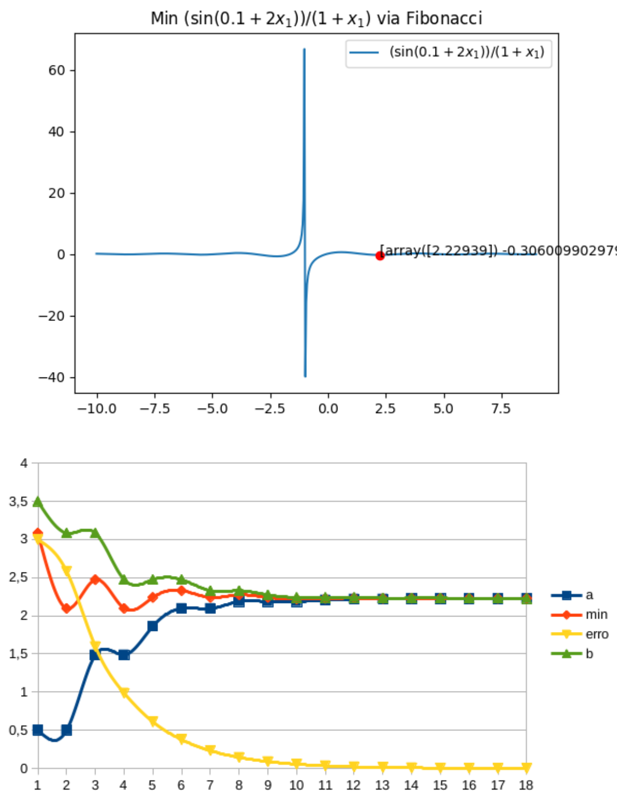


(b) Fibonacci

Figura 2: Questão 2 – Letra B - Mínimos Computados e convergência dos algoritmos

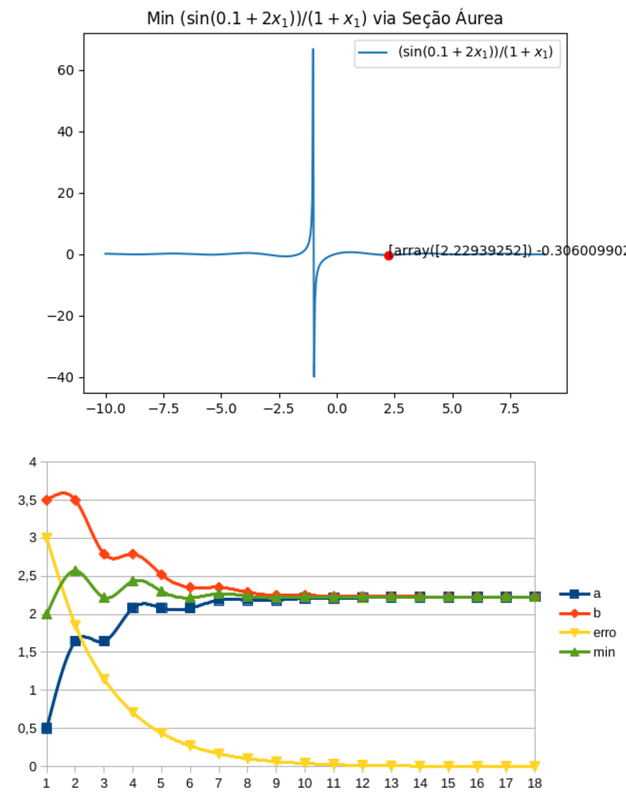


(a) Seção Áurea

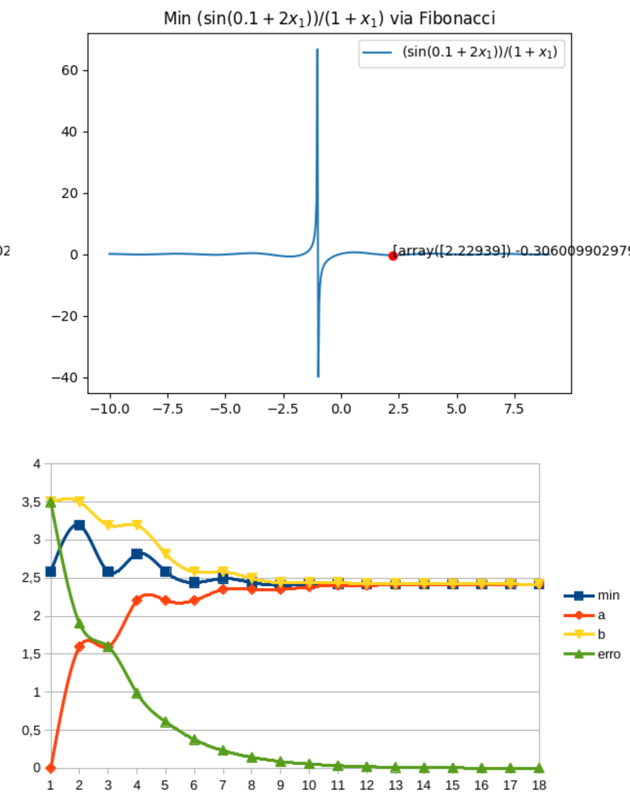


(b) Fibonacci

Figura 3: Questão 2 – Letra C - Mínimos Computados e convergência dos algoritmos



(a) Seção Áurea



(b) Fibonacci

Figura 4: Questão 2 – Letra C – Versão Interpolada
Mínimos Computados e convergência dos algoritmos

Os gráficos de convergência foram gerados a partir dos arquivos CSV anexos a este relatório. Os algoritmos foram executados com as funções listadas acima sob tolerância de 10^{-3}. Nota-se que ambos os algoritmos obtiveram o $x*$ na mesma quantidade de iterações em todos os casos (com exceção da letra B com 1 iteração de diferença), algo esperado visto que ambos possuem uma razão de convergência similar. Além disso, nota-se que os intervalos delimitados por a e b foram reduzidos de maneira similar nas diversas execuções, visto que a sequência de tais termos sobre as iterações possuem r-convergência linear com coeficiente $\tau \approx 0.618$. É possível concluir tal fato através das análises experimentais do algoritmo, e de forma contundente, pelo fato de que a cada iteração, o tamanho do intervalo contendo o mínimo de $f(x)$ é reducido por τ, de tal forma que $b_k - a_k = \tau^k(b_0 - a_0)$. Visto que $x* \in [a_k, b_k] \forall k \in \mathbb{N}$, temos que:

$$\begin{aligned}(b_k - x*) &\leq (b_k - a_k) \leq \tau^k(b_0 - a_0) \\ (x - a_k) &\leq (b_k - a_k) \leq \tau^k(b_0 - a_0)\end{aligned} \tag{1}$$

Foram gerados três polinômios interpoladores com N pontos equidistantes, onde N foi a diferença entre os intervalos a e b de interesse de cada função respectivamente. Logo, foram gerados os polinômios com os pontos abaixo:

Letra A	
x	f(x)
0	0
1	-2
2	2
3	12

Letra B	
x	f(x)
0	9
1	4
2	1
3	0
4	1
5	4
6	9
7	16
8	25
9	36

Letra C	
x	f(x)
0	0,09983341664682815
1	0,43160468332443686
2	-0,2727590370214701
3	-0,04554062606802397

Tabela 1: Pontos gerados para criação do polinômio interpolador

Os pontos gerados para os itens (a) e (b) resultaram na função original do exercício, logo os $x's*$ encontrandos foram os mesmos, esta "coincidência"resulta do fato de que a interpolação de pontos equidistantes de um polinômio gera o próprio polinômio (ver algoritmo de Gregory-Newton). Visto que o item (c) não era um polinômio, a função interpolada não foi exatamente igual a função original, o polinômio obtido $0.327952853053811x^3 - 1.50192605267319x^2 + 1.50574446629699x + 0.0998334166468282$ teve como mínimo no intervalo $[0.5, 3.5] \in \mathbb{R}$ um $x* = 2.420$ e $x* = 2.421$, obtidos respectivamente com o algoritmo da Seção Áurea e Fibonacci, uma aproximação aceitável do $x*$ encontrado para a solução exata, sendo este igual a 2.229. Alternativamente foi obtida a derivada do polinômio interpolador, e igualando-o a zero, obtendo-se as raízes da parábola como $x_1 = 0.63216286445301$ e $x_2 = 2.4209712241569$, sendo x_2 o mínimo do polinômio interpolador novamente. Embora ambos os algoritmos gastaram a mesma quantidade de iterações para minimizar a função original e a interpolada, é sabido que a interpolação de um intervalo da função e a posterior computação do mínimo sob a função interpolada tende a ser menos custoso computacionalmente em alguns casos (onde a avaliação da função original é computacionalmente cara) ao custo de uma considerável perda de precisão na computação do $x*$ da função real (visto que a interpolação pode não representar fielmente a função original).

Questão 3

Dada as funções abaixo, foram computados seus respectivos mínimos através dos métodos numéricos listados na Questão 1, para cada execução foi analisada a convergência do método.

(a) $F(x_1) = "x_1^2 - 3x_1x_2 + 4x_2^2 + x_1 - x_2"$

(b) $F(x_1) = 6x_1^2 + 2x_2^2 - 6x_1x_2 - x_1 - 2x_2$

(c) $F(x_1) = x_1 - x_2 + 2x_1^2 + 2x_1x_2 + x_2^2$

É sabido que os mínimos de tais funções são respectivamente $\left(-\frac{5}{7}, -\frac{1}{7}\right)$, $\left(\frac{4}{3}, \frac{5}{2}\right)$ e $\left(-1, \frac{3}{2}\right)$.

Abaixo pode-se observar os gráficos dos mínimos computados de cada função e a convergência de cada algoritmo:

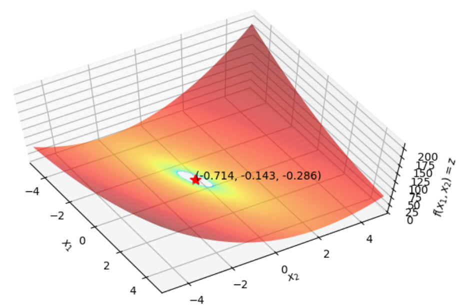


Figura 5: $\min f(X) = x_1^2 - 3x_1x_2 + 4x_2^2 + x_1 - x_2$

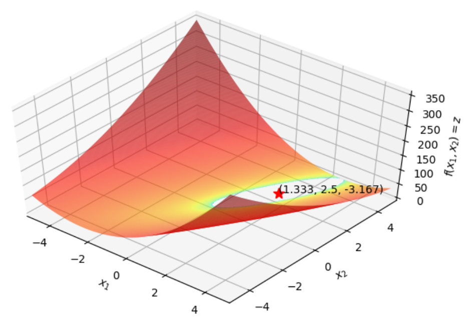


Figura 6: $\min f(X) = 6x_1^2 + 2x_2^2 - 6x_1x_2 - x_1 - 2x_2$

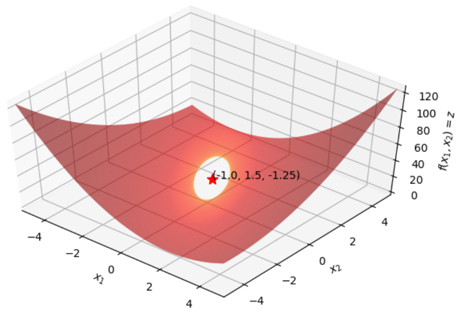


Figura 7: $\min f(X) = x_1^2 - 3x_1x_2 + 4x_2^2 + x_1 - x_2$

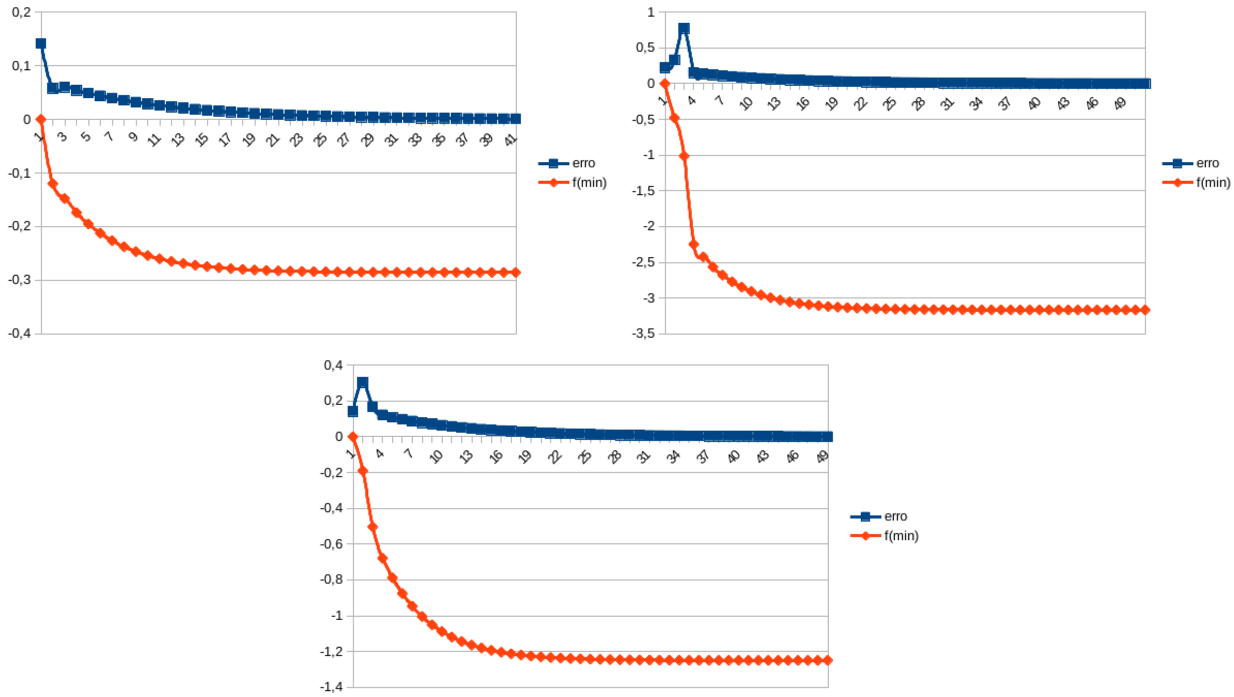


Figura 8: Método BFGS sendo executado com as funções (a), (b) e (c)

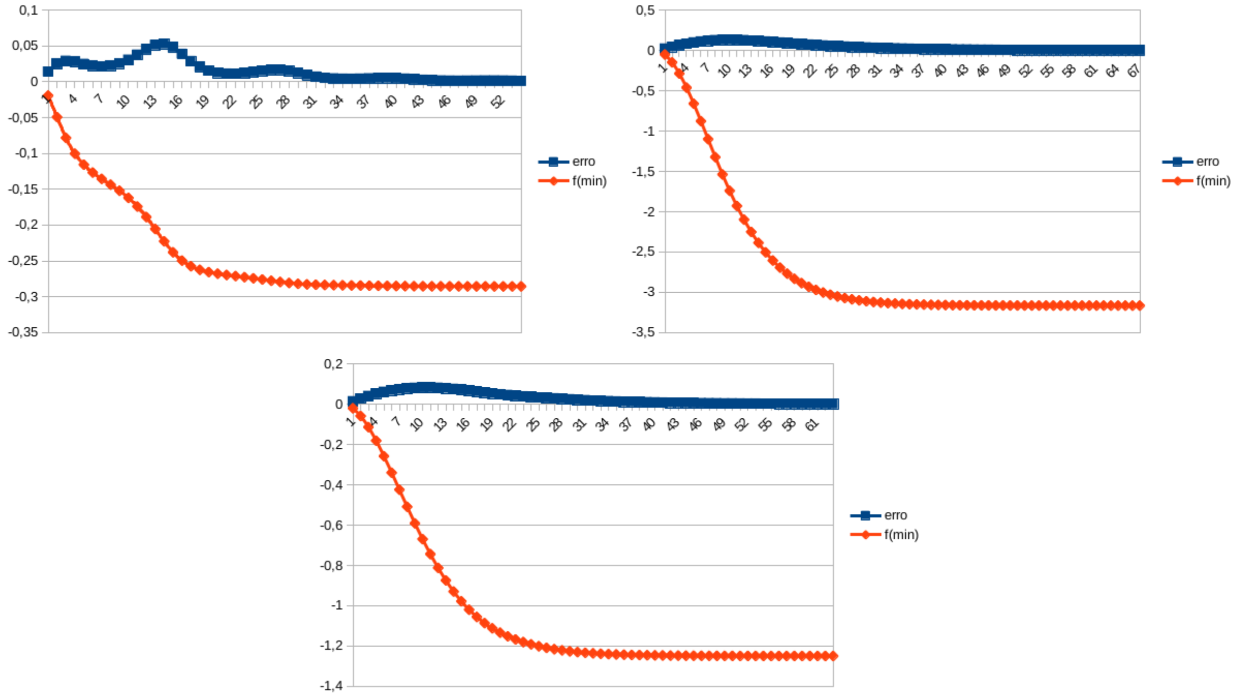


Figura 9: Método do Gradiente Conjugado sendo executado com as funções (a), (b) e (c)

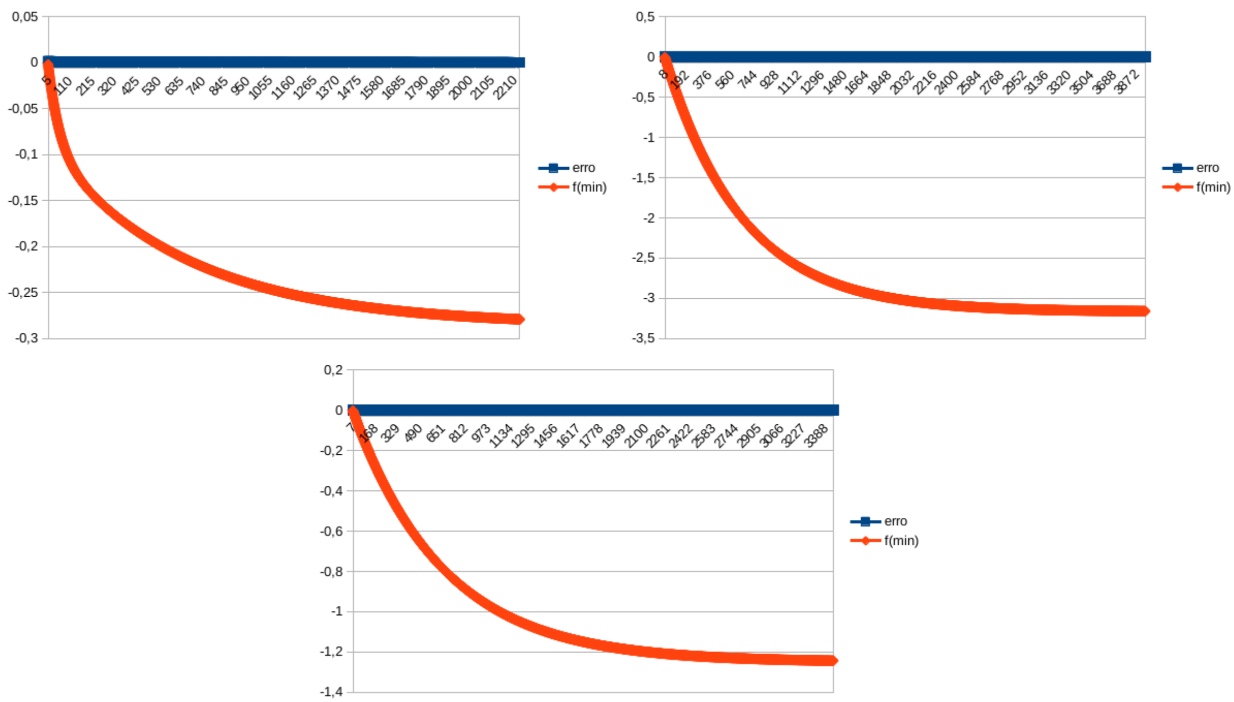


Figura 10: Método da Máxima Descida sendo executado com as funções (a), (b) e (c)

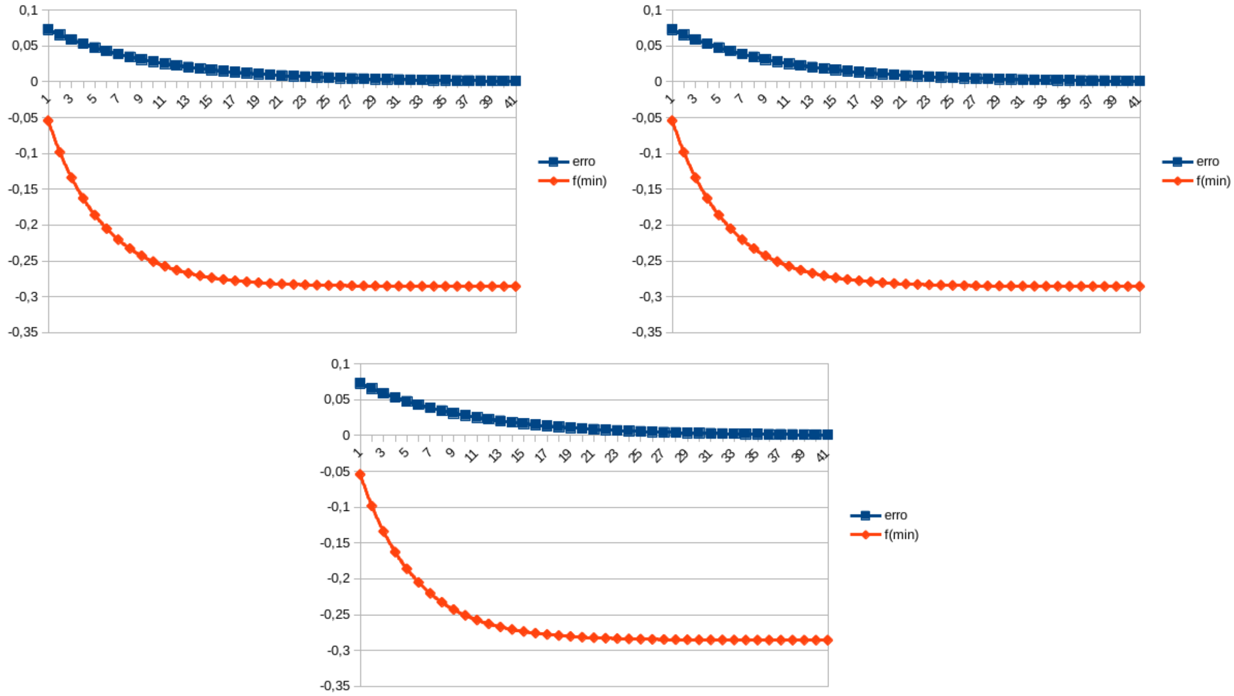


Figura 11: Método de Newton sendo executado com as funções (a), (b) e (c)

Todos os métodos foram capazes de convergir para um $x*$ com 3 casas decimais de precisão sob tolerância $= 10^{-5}$. Os dados utilizados para plotagem podem ser encontrados em arquivos CSV anexos a este relatório. Assim como esperava-se, métodos de segunda ordem convergem quase que imediatamente para o $x*$, visto que estes utilizam a primeira e segunda derivada da função, visto que estas foram computadas *a priori*, o algoritmo implementado foi extremamente veloz, visto que não houve a necessidade de computar tais termos numericamente (é possível consultar as entradas usadas em cada método nos arquivos *.sh anexos a este relatório). Métodos Quasi-Newton como o BFGS e DFP obtiveram performance semelhante, visto que apesar de começarem com a matriz identidade como a Hessiana, esta rapidamente se torna a hessiana exata. No pior caso do BFGS, a hessiana exata foi atingida na iteração 13, enquanto que o pior caso do DFP atinge a hessiana exata na iteração 37, fazendo métodos Quasi-Newton ótimos substitutos para métodos de Segunda Ordem em aplicações de tempo real.
Métodos de Ordem Um possuem um decaimento do erro da ordem de e^{-x}, o autor deste relatório desconhece a razão disto, mas conjectura que tal fato possa estar relacionado com o fato que algoritmos desta classe utilizam-se da informação da primeira derivada como uma linha tangente a um ponto em sua superfície de erro, e o fato de que uma série de Taylor pode ser vista como uma forma de "enumerar"todas as derivadas de f em $x = b$ (isto é, se a série de Taylor de uma função é conhecida é possível extrair qualquer derivada da mesma no ponto b), visto que uma série de Taylor de f em $x = b$ é dada por:

$$\sum_{n=0}^{\infty} \frac{f^{(n)}(b)}{n!}(x-b)^n \tag{2}$$

Métodos de Ordem um possuem performance intermediária, computando o $x*$ em menos iterações que métodos de ordem zero, embora sejam facilmente superados por métodos de segunda ordem/Quasi-Newton neste quesito.
Métodos de Ordem Zero são fáceis de implementar, mas possuem uma taxa de convergência consideravelmente lenta quando comparados a demais métodos, consequentemente estes métodos avaliam a função objetivo um número demasiado de vezes, tornando-os computacionalmente dispendiosos (e em alguns casos inviáveis). As exigências para a garantia de convergência destes métodos é a continuidade e unimodalidade da função objetivo, uma clara vantagem sobre os demais métodos visto que a diferenciabilidade da função é irrelevante para tais métodos. De forma excepcional, o algoritmo do Polítopo apresenta convergência semelhante aos métodos de Ordem Um (especialmente em modo adaptativo)[1].

Códigos

Método da Busca Aleatória

```python
from numpy import *

def minBuscaAleatoria (f,a,b,tol,max_it,isAleatorio):
 tol = tol**(-1.0)
 x_otimo = 0.0
 espacoBusca = linspace(a,b,tol)
 if isAleatorio:#nao garante minimo
  for i in range(max_it):
   indice = random.choice(espacoBusca.shape[0], len(a), replace=False)
   elemento_aleatorio = espacoBusca[indice]
   x_otimo = elemento_aleatorio if x_otimo==None or \
             f([elemento_aleatorio])<f([x_otimo]) else x_otimo
 else:#garante um pouco mais
  espacoBusca = [({'x':x,'f(x)':f([x])}) for x in linspace(a,b,tol)]
  x_otimo = sorted(espacoBusca, key = lambda i: i['f(x)'])[0]['x']
 return x_otimo


def main():
 f = lambda x: (12*x[0]**2-16*x[0]+8)
 a = [0.0]
 b = [10.0]
 tol = 0.0001
 max_it = 1000
 res = minBuscaAleatoria(f,a,b,tol,max_it,False)
 print({'x':res,'f(x)':f(res)})


if __name__ == "__main__":
 main()
```

Método da Seção Áurea

```
from numpy import *

def reduz_intervalo(func, xa=array([0.0]), xb=array([1.0]), \
                    max_cresc=110.0, maxiter=1000):
 razao_aurea = (1.0+sqrt(5.0))/2.0
 tol = 1e-21
 fa = func(*(xa,))
 fb = func(*(xb,))
 if (fa < fb):
  xa, xb = xb, xa
  fa, fb = fb, fa
 xc = xb + razao_aurea * (xb - xa)
 fc = func(*((xc,)))
 avaliacoesFO = 3
 iter = 0
 while (fc < fb):
  tmp1 = (xb - xa) * (fb - fc)
  tmp2 = (xb - xc) * (fb - fa)
  val = tmp2 - tmp1
  if abs(val) < tol:
   denom = 2.0 * tol
  else:
   denom = 2.0 * val
  w = xb - ((xb - xc) * tmp2 - (xb - xa) * tmp1) / denom
  wlim = xb + max_cresc * (xc - xb)
  iter += 1
  if (w - xc) * (xb - w) > 0.0:
   fw = func(*((w,)))
   avaliacoesFO += 1
   if (fw < fc):
    xa = xb
    xb = w
    fa = fb
    fb = fw
    return xa, xb, xc, fa, fb, fc, avaliacoesFO
   elif (fw > fb):
    xc = w
    fc = fw
    return xa, xb, xc, fa, fb, fc, avaliacoesFO
   w = xc + razao_aurea * (xc - xb)
   fw = func(*((w,)))
   avaliacoesFO += 1
  elif (w - wlim)*(wlim - xc) >= 0.0:
```

```
    w = wlim
    fw = func(*((w,)))
    avaliacoesFO += 1
   elif (w - wlim)*(xc - w) > 0.0:
    fw = func(*((w,)))
    avaliacoesFO += 1
    if (fw < fc):
     xb = xc
     xc = w
     w = xc + razao_aurea * (xc - xb)
     fb = fc
     fc = fw
     fw = func(*((w,)))
     avaliacoesFO += 1
   else:
    w = xc + razao_aurea * (xc - xb)
    fw = func(*((w,)))
    avaliacoesFO += 1
   xa = xb
   xb = xc
   xc = w
   fa = fb
   fb = fc
   fc = fw
  return xa, xb, xc, fa, fb, fc

def minimiza_secao_aurea (func, xa, xb, xc, xtol=0.001, maxiter=5000):
 fa,fb,fc = func(xa),func(xb),func(xc)
 tol = xtol
 avaliacoesFO = 0
 conjSecaoAurea = ((1.0+sqrt(5.0))/2.0)-1.0
 complConjSecaoAurea = 1.0 - conjSecaoAurea
 x3 = xc
 x0 = xa
 if (abs(xc - xb) > abs(xb - xa)):
  x1 = xb
  x2 = xb + complConjSecaoAurea * (xc - xb)
 else:
  x2 = xb
  x1 = xb - complConjSecaoAurea * (xb - xa)
 f1 = func(*((x1,)))
 f2 = func(*((x2,)))
 avaliacoesFO += 2
 nit = 0
 for i in range(maxiter):
```

```
  if abs(x3 - x0) <= tol * (abs(x1) + abs(x2)):
   break
  if (f2 < f1):
   x0 = x1
   x1 = x2
   x2 = conjSecaoAurea * x1 + complConjSecaoAurea * x3
   f1 = f2
   f2 = func(*((x2,)))
  else:
   x3 = x2
   x2 = x1
   x1 = conjSecaoAurea * x2 + complConjSecaoAurea * x0
   f2 = f1
   f1 = func(*((x1,)))
  avaliacoesFO += 1
  nit += 1
  xmin = x1 if f1 < f2 else x2
  ymin = f1 if f1 < f2 else f2
 return xmin

def main():
 f = lambda x: 12*x[0]**2-16*x[0]+8
 xa, xb, xc, fa, fb, fc = reduz_intervalo(f)
 res = minimiza_secao_aurea(f,xa, xb, xc)
 print({"x":res,"f(x)":f(res)})

if __name__ == "__main__":
 main()
```

Método de Fibonacci

```
from numpy import *

fibonacci = lambda n: n if n<=1 else fibonacci(n-2) + fibonacci(n-1)

def minimiza_fibonacci(f, a, b, tol=0.01):
 h = abs(tol)
 fa = f(a)
 fb = f(b)
 num_fibonacci = 100

 n = (b-a)/tol
 n = n-1 if n*tol==b-a else n

 i = 1
 while (i<num_fibonacci):
  if (fibonacci(i-1) + fibonacci(i))>=n:
   break
  if i>num_fibonacci:
   return
  i += 1
 x1 = a + fibonacci(i - 1) * tol
 x2 = a + fibonacci(i) * tol
 fx1 = f(x1)
 if x2 <= b:
  fx2 = f(x2)
 else:
  fx2 = sys.maxint
 while i > 0:
  if (fx1 > fx2): #min max
   a = x1
   fa = fx1
   x1 = x2
   fx1 = fx2
   i = i-1
   if i==0:
    break
   x2 = a + fibonacci(i) * tol
   if x2 <= b:
    fx2 = f(x2)
   else:
    fx2 = sys.maxint
  else:
   b = x2
```

```
      fb = fx2
      x2 = x1
      fx2 = fx1
      i = i-1
      if i==0:
       break
      x1 = a + fibonacci(i - 1) * tol
      fx1 = f(x1)
  return x2.ravel()

def main():
 f = lambda x: (12*x[0]**2-16*x[0]+8)
 a = array([0.0])
 b = array([12.0])
 tol = 0.0001
 res = minimiza_fibonacci(f, a, b, tol)
 print({"x":res,"f(x)":f(res)})

if __name__ == "__main__":
 main()
```

Método de Powell

```
from numpy import *

def minimiza_powell (f,a,b,tol):
 x1 = a
 x2 = a+b
 x3 = b

 f1 = f(x1)
 f2 = f(x2)
 f3 = f(x3)
 tol = 0.01

 while x3-x1>tol:
  x4 = 0.5 * (((x2**2 - x3**2) * f1 + (x3**2-x1**2) * f2 + \
            (x1**2 - x2**2) * f3) / (( x2-x3) * f1 + (x3-x1)*f2 + \
            (x1-x2)*f3))
  f4 = f(x4)
  if x4>x2:
   if f4<f2:
    x1 = x2
    x2 = x4
    f1 = f2
    f2 = f4
    x3 = x4
    f3 = f4
   if f4<f2:
    x3 = x2
    x2 = x4
    f3 = f2
    f2 = f4
   x1 = x4
   f1 = f4
 return x4

def main():
 f = lambda x: 12*x[0]**2-16*x[0]+8
 a = array([-0.5])
 b = array([0.5])
 tol = 0.001
 res = minimiza_powell(f,a,b,tol)
 print({'x':res,'f(x)':f(res)})

if __name__ == "__main__":
```

```
main()
```

Método de Nelder-Mead/Polítopo

```
from copy import *
from numpy import *

def nelder_mead(f,  x_inicial, passo=0.1, limiar_semMelhora=10e-6, \
                max_sem_melhora=10, max_iter=0, alfa=1., gama=2., \
                rho=-0.5, sigma=0.5):
 dim = len( x_inicial)
 melhor_anterior = f( x_inicial)
 sem_melhora = 0
 res = [[ x_inicial, melhor_anterior]]

 for i in range(dim):
  x = copy(x_inicial)
  x[i] = x[i] + passo
  score = f(x)
  res.append([x, score])

 it = 0
 while 1:
  res.sort(key=lambda x: x[1])
  melhor_ponto = res[0][1]

  if max_iter and it >= max_iter:
   return res[0]
  it += 1

  if melhor_ponto < melhor_anterior - limiar_semMelhora:
   sem_melhora = 0
   melhor_anterior = melhor_ponto
  else:
   sem_melhora += 1
  if sem_melhora >= max_sem_melhora:
   return res[0]

  #reposiciona centroide
  x0 = [0.] * dim
  for tup in res[:-1]:
   for i, c in enumerate(tup[0]):
    x0[i] += c / (len(res)-1)

   #reflexao
  xr = x0 + alfa*(x0 - res[-1][0])
  rscore = f(xr)
```

```
        if res[0][1] <= rscore < res[-2][1]:
            del res[-1]
            res.append([xr, rscore])
            continue

            #expansao
        if rscore < res[0][1]:
            xe = x0 + gama*(x0 - res[-1][0])
            escore = f(xe)
            if escore < rscore:
                del res[-1]
                res.append([xe, escore])
                continue
            else:
                del res[-1]
                res.append([xr, rscore])
                continue

            #contracao
        xc = x0 + rho*(x0 - res[-1][0])
        cscore = f(xc)
        if cscore < res[-1][1]:
            del res[-1]
            res.append([xc, cscore])
            continue

            reducao
        x1 = res[0][0]
        nres = []
        for tup in res:
            redx = x1 + sigma*(tup[0] - x1)
            score = f(redx)
            nres.append([redx, score])
        res = nres
    return res

if __name__ == "__main__":
    f = lambda x: (12*x[0]**2-16*x[0]+8)
    x_inicial = array([0.])
    res = nelder_mead(f, x_inicial)[0]
    print({'x':res,'f(x)':f(res)})
```

Método da Máxima Descida

```
from numpy import *

def minimiza_descida_maxima(f, grad, x_inicial, h, max_it, tol):
  x = x_inicial
  x_anterior = x
  for i in xrange(max_it):
    direcao_descida = -grad(x)
    alfa = h
    x = x + alfa * direcao_descida
    if linalg.norm(x - x_anterior) < tol:
      break
    x_anterior = x
  return x

def main():
 f = lambda x: 12*x[0]**2-16*x[0]+8
 df = lambda x: 24*x[0]-16
 x_0 = array([0.0])
 h = 1E-5
 tol = 1E-10
 max_it = 100000000
 res = minimiza_descida_maxima(f,df,x_0,h,max_it,tol)
 print({'x':res,'f(x)':f(res)})

if __name__ == "__main__":
 main()
```

Método do Gradiente Conjugado

```python
from numpy import *

def minimiza_gradiente_conjugado(f, df, x0, max_it, tol):
  xk = x0
  fk = f(xk)
  gk = df(xk)
  pk = -gk
  for i in range(max_it):
    alfa = 0.01
    xk1 = xk + alfa * pk
    gk1 = df(xk1)
    beta_k1 = dot(gk1, gk1) / dot(gk, gk)
    pk1 = -gk1 + beta_k1 * pk
    if linalg.norm(xk1 - xk) < tol:
      xk = xk1
      break
    xk = xk1
    gk = gk1
    pk = pk1
  return xk

def main():
 f  = lambda x: (12*x[0]**2-16*x[0]+8)
 df = lambda x: (24*x[0]-16)
 x_inicial = array([0.0])
 tol = 1e-4
 max_it = 1000
 x = minimiza_gradiente_conjugado(f, df, x_inicial,max_it=max_it, tol=tol)
 print({'x':x,'f(x)':f(x)})

if __name__ == '__main__':
 main()
```

Método de Newton

```python
from numpy import *

def minimiza_newton(f, g, H, x0, max_it, tol):
  x = x0
  x_anterior = x
  for i in xrange(max_it):
    direcao_descida = -linalg.solve(H(x), g(x))
    alfa = 0.1
    x = x + alfa * direcao_descida
    if linalg.norm(x - x_anterior) < tol:
      break
    x_anterior = x
  return x.ravel()


def main():
 f   = lambda x: 12*x[0]**2-16*x[0]+8
 df  = lambda x: array([24*x-16])
 d2f = lambda x: array([[24]])
 x_0 = 0.0
 h = 0.01
 max_it = 100
 tol = 0.0001
 res = minimiza_newton(f,df,d2f,x_0,max_it,tol)
 print({'x':res,'f(x)':f(res)})

if __name__ == "__main__":
 main()
```

Método da Métrica Variável/BFGS

```
from numpy import *

def minimiza_bfgs(f, df, x_inicial, max_it, tol):
  xk = x_inicial
  I = eye(xk.size)
  Hk = I

  for i in range(max_it):
    direcao_descida_1 = df(xk)
    direcao_descida_2 = -Hk.dot(direcao_descida_1)

    alfa = 0.1

    xk1 = xk + alfa * direcao_descida_2
    direcao_descida_1_1 = df(xk1)

    sk = xk1 - xk
    yk = direcao_descida_1_1 - direcao_descida_1

    rho_k = 1.0 / yk.dot(sk)

    Hk1 = (I - rho_k * outer(sk, yk)).dot(Hk). \
    dot(I - rho_k * outer(yk, sk)) + rho_k * outer(sk, sk)

    if linalg.norm(xk1 - xk) < tol:
      xk = xk1
      break

    Hk = Hk1
    xk = xk1

  return xk.ravel()


def main():
 f  = lambda x: 12*x[0]**2-16*x[0]+8
 df = lambda x: 24*x[0]-16
 x_inicial = array([0.0])
 max_it = 10000
 tol    = 0.00001
 res = minimiza_bfgs(f, df, x_inicial,max_it,tol)
 print({"x":res,"f(x)":f(res)})
```

```
if __name__ == "__main__":
    main()
```

Método da Interpolação de Polinomial

```python
from sympy import Matrix, Symbol, zeros

def combina (n, k):
    if k == 0:
        return [[]]
    combs = [[i] for i in range(n)]
    for i in range(k - 1):
        atual = []
        for p in combs:
            for m in range(p[-1], n):
                atual.append(p + [m])
        combs = atual
    return combs

def gera_simbolo (simbolo_str):
    n = 0
    while True:
        yield Symbol("%s_%d" % (simbolo_str, n))
        n += 1

def vandermonde (grau, dim=1, simbolos='a b c d'):
    simbolos = simbolos.split()
    n = len(simbolos)
    if n < dim:
        novos_simbolos = []
        for i in range(dim - n):
            j, resto = divmod(i, n)
            novos_simbolos.append(simbolos[resto] + str(j))
        simbolos.extend(novos_simbolos)
    termos = []
    for i in range(grau + 1):
        termos.extend(combina(dim, i))
    posto = len(termos)
    V = zeros(posto)
    geradores = [gera_simbolo(simbolos[i]) for i in range(dim)]
    todos_simbolos = []
    for i in range(posto):
        linha_simbolos = [next(g) for g in geradores]
        todos_simbolos.append(linha_simbolos)
        for j, termo in enumerate(termos):
            elemento_v = 1
            for k in termo:
                elemento_v *= linha_simbolos[k]
```

```
            V[i*posto + j] = elemento_v
    return V, todos_simbolos, termos

def gera_polinomio (pontos, grau, simbolos):
    qtde_pts = len(pontos)
    dim = len(pontos[0]) - 1
    V, simb_temp, termos = vandermonde(grau, dim)
    subs_dict = {}
    for j in range(dim):
        for i in range(qtde_pts):
            subs_dict[simb_temp[i][j]] = pontos[i][j]
    V_pts = V.subs(subs_dict)
    V_inv = V_pts.inv()
    coeficientes = V_inv.multiply(Matrix([pontos[i][-1] for i in range(qtde
    f = 0
    for j, termo in enumerate(termos):
        t = 1
        for k in termo:
            t *= simbolos[k]
        f += coeficientes[j]*t
    return f

def main():
 x = Symbol('x')
 pontos = [(0, 3), (1, 2), (2, 3)]
 print("f(x) = " + str(gera_polinomio(pontos, len(pontos)-1, [x])))

if __name__ == "__main__":
 main()
```

Referências

[1] Gao, F., & Han, L. (2012). Implementing the Nelder-Mead simplex algorithm with adaptive parameters. Computational Optimization and Applications, 51(1), 259-277.