Mini projeto 2 - Calculo

Ruan César Oliveira da Silva, Djairo Dantas, Marcus Vinicius Maia

05/11/2025

1 Introdução

1.1 Os objetivos desse relatório são:

- Fazer o estudo das funções:

$$f(x,y) = 3x^2 + 3xy + 2y^2 + x - y \tag{1}$$

$$g(x,y) = \sqrt{x^2 + y^2 + 3 + x^2 e^{-y^2}} + (x-2)^2 \tag{2}$$

$$h(x,y) = 4e^{-x^2-y^2} + 3e^{-x^2-y^2+4x+6y-13} - \frac{x^2}{9} - \frac{y^2}{15} + 2 \tag{3}$$

- Elaborar um código que permita estudar as funções apresentadas, no que tange cálculo de pontos críticos e vetores gradientes.

2 Desenvolvimento

2.1 Tarefa 1 (40%)

a) Utilizando o GeoGebra para representar o gráfico de f(x), obtêm-se:

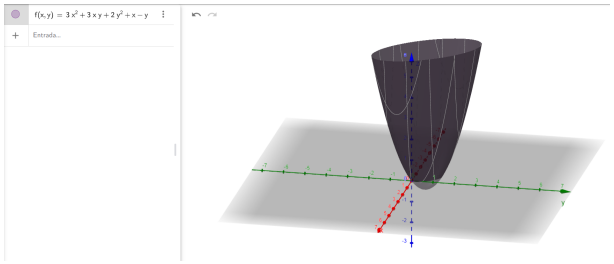

Figure 1: Gráfico da função f(x).

b) Realizando o cálculo do vetor gradiente:

$$\nabla f(x,y) = \left(\frac{\partial f}{\partial x},\ \frac{\partial f}{\partial y}\right) = (6x + 3y + 1,\ 3x + 4y - 1) \tag{4}$$

c) Dado o problema, calcula-se o mínimo da função sabendo que o módulo do gradiente é igual a 0. Devido as limitações computacionais, faz-se o módulo do gradiente tender a 0, definindo a variável tol como limite de parada, iterando os valores de (x,y) até atingir a condição limite. Com isso em mente, o seguinte código foi desenvolvido:

> Observação
>
> A legenda contendo o resultado obtido, número de iterações e a descrição do comportamento são exibidas ao executar o código presente no repositório do GitHub.

```python
def grad_f(x, y):
    #   f   = (6x + 3y + 1, 3x + 4y - 1)
    return np.array([6*x + 3*y + 1, 3*x + 4*y - 1], dtype=float)

# Par metros
alpha = 0.1
tol = 1e-5
max_iters = 100000

# Inicializa  o
x, y = 0.0, 0.0
g = grad_f(x, y)
it = 0

# Gradient descent
while np.linalg.norm(g, ord=2) > tol and it < max_iters:
    x, y = np.array([x, y]) - alpha * g
    g = grad_f(x, y)
    it += 1

print(f"Itera  es: {it}")
print(f"Ponto de m nimo aproximado: (x, y) = ({x:.10f}, {y:.10f})")
print(f"||  f  || = {np.linalg.norm(g):.3e}")
print(f"f(x,y) = {f(x,y):.10f}")
```

Listing 1: Gradiente descendente para $f(x, y)$

d) Mantendo o procedimento para: $\alpha = 0{,}15$, $\alpha = 0{,}2$, $\alpha = 0{,}3$ e $\alpha = 0{,}5$.

```python
    def gradient_descent(alpha, tol=1e-5, max_iters=100000):
        x, y = 0.0, 0.0
        g = grad_f(x, y)
        it = 0
        while np.linalg.norm(g) > tol and it < max_iters:
            x, y = np.array([x, y]) - alpha * g
            g = grad_f(x, y)
            it += 1
        return x, y, it, np.linalg.norm(g), f(x, y)

    # Valores de alpha
    alphas = [0.15, 0.2, 0.3, 0.5]

    for alpha in alphas:
        x, y, it, norm_g, val = gradient_descent(alpha)
        print(f"   = {alpha}")
        print(f" -> Ponto m nimo aproximado: (x, y) = ({x:.6f}, {y:.6f})"
    ↪ )
        print(f" -> f(x,y) = {val:.6f}")
        print(f" -> ||  f  || = {norm_g:.2e}, Itera  es: {it}\n")

```

Listing 2: Função de gradiente descendente com passo variável

Observação

Ao executar o código do item d), nota-se um erro de overflow, pois o valor de α é grande demais.

2.2 Tarefa 2 (20%)

1. Seguindo o mesmo procedimento para a função g(x):

```python
def g(x, y):
    return np.sqrt(x*x + y*y + 3) + x*x*np.exp(-y*y) + (x - 2)**2

def grad_g(x, y):
    r = np.sqrt(x*x + y*y + 3)
    gx = x/r + 2*x*np.exp(-y*y) + 2*(x - 2)
    gy = y/r - 2*y*(x**2)*np.exp(-y*y)
    return np.array([gx, gy], dtype=float)

def gradient_descent(alpha, x0=0.0, y0=0.0, tol=1e-6, max_iters=200000):
    x, y = float(x0), float(y0)
    for it in range(1, max_iters+1):
        gk = grad_g(x, y)
        ng = float(np.linalg.norm(gk))
        if not np.isfinite(ng):
            return None, None, it, np.inf, np.inf  # divergiu
    ↪ numericamente
        if ng < tol:
            return x, y, it, ng, g(x, y)
        x -= alpha * gk[0]
        y -= alpha * gk[1]
    return x, y, it, ng, g(x, y)

#  Encontrar os dois m nimos (ex.:    = 0.1)
alpha = 0.1
sol1 = gradient_descent(alpha, x0=0.0, y0=+3.0)  # um lado do vale
sol2 = gradient_descent(alpha, x0=0.0, y0=-3.0)  # lado sim trico

for tag, sol in [("M nimo A", sol1), ("M nimo B", sol2)]:
    x, y, it, ng, val = sol
    print(f"{tag}: (x,y)=({x:.6f}, {y:.6f}),  f={val:.6f},  || f ||={ng
    ↪ :.2e}, iters={it}")
```

Listing 3: Gradiente descendente aplicado à função $g(x, y)$

2.3 Tarefa 3 (20%)

1. Seguindo o procedimento:

```python
        # ---- fun   o e gradiente ----
        def h(x, y):
            r2 = x*x + y*y
            e1 = np.exp(-r2)                                    # pico em
    ↪ (0,0)
            e2 = np.exp(-((x-2)**2 + (y-3)**2))                 # pico em
    ↪ (2,3)
            return 4*e1 + 3*e2 - x*x/9 - y*y/15 + 2

        def grad_h(x, y):
            r2 = x*x + y*y
            e1 = np.exp(-r2)
            e2 = np.exp(-((x-2)**2 + (y-3)**2))
            #   h / x   = -8x e^{-r2} - 6(x-2) e^{-((x-2)^2+(y-3)^2)} -
    ↪ (2/9)x
            #   h / y   = -8y e^{-r2} - 6(y-3) e^{-((x-2)^2+(y-3)^2)} -
    ↪ (2/15)y
            hx = -8*x*e1 - 6*(x-2)*e2 - (2/9)*x
            hy = -8*y*e1 - 6*(y-3)*e2 - (2/15)*y
            return np.array([hx, hy], dtype=float)

        # gradient ascent com passo fixo
        def gradient_ascent(alpha, x0=0.0, y0=0.0, tol=1e-6, max_iters
    ↪ =200000):
```

```
            x, y = float(x0), float(y0)
            for it in range(1, max_iters+1):
                gk = grad_h(x, y)
                ng = float(np.linalg.norm(gk))
                if not np.isfinite(ng):
                    return None, None, it, np.inf, np.inf, "divergiu-num"
                if ng < tol:
                    return x, y, it, ng, h(x, y), "convergiu"
                # ascent: sobe na dire  o do gradiente
                x += alpha * gk[0]
                y += alpha * gk[1]
            return x, y, it, ng, h(x, y), "limite-iters"

        # (1) Encontre os m ximos (use alphas moderados, ex.: 0.1)
        alpha = 0.1
        candidatos = {
            "prox_origem": (0.0, 0.0),
            "prox_pico_2_3": (2.0, 3.0),
            "longe_1": (3.0, 4.0),
            "longe_2": (-2.0, -3.0),
        }

        print("M ximos locais (ascent,   =0.1):")
        vistos = []
        def ja_visto(px, py, L=1e-3):
            for (ux, uy) in vistos:
                if np.hypot(px-ux, py-uy) < L:
                    return True
            return False

        for nome, (x0, y0) in candidatos.items():
            x, y, it, ng, val, status = gradient_ascent(alpha, x0, y0)
            if x is None:
                print(f"{nome}: divergiu numericamente")
                continue
            if not ja_visto(x, y):
                vistos.append((x, y))
            print(f"{nome:>13}: (x,y)=({x:.6f}, {y:.6f}),  h={val:.6f},
↪ ||  h  ||={ng:.2e}, iters={it}, {status}")

        # (2) Comportamento vs passo    para cada bacia de atra  o
        alphas = [0.05, 0.1, 0.2, 0.3, 0.5]
        starts = [(0,0), (2,3)]
        print("\nEstabilidade por    (converge/diverge):")
        for a in alphas:
            linha = []
            for (xi, yi) in starts:
                x, y, it, ng, val, status = gradient_ascent(a, xi, yi)
                if x is None or not np.isfinite(val):
                    linha.append("diverge")
                else:
                    linha.append("converge")
            print(f"  ={a:>4}: {linha}")

```

Listing 4: Algoritmo de *Gradient Ascent* aplicado à função $h(x, y)$

2.4 Tarefa Desafio (10%)

1. Backtracking (Condição de Armijo) Escolhe o valor de α por tentativas decrescentes:

$$\alpha,\ \beta\alpha,\ \beta^2\alpha, \ldots$$

até que seja satisfeita a condição de Armijo:

$$f(x - \alpha \nabla f) \leq f(x) - c\,\alpha \|\nabla f\|^2 \tag{5}$$

Esse método é **robusto** e garante **decréscimo monotônico** da função objetivo.

2. **Condições de Wolfe e Strong Wolfe** Extensão da regra de Armijo, adicionando uma **condição de curvatura** que controla o tamanho do passo. É amplamente usada em métodos mais avançados como:
   - BFGS (*Broyden–Fletcher–Goldfarb–Shanno*);
   - *Conjugate Gradient*.

3. **Busca Exata (*Line Search*)** Para funções quadráticas da forma:

$$f(x) = \tfrac{1}{2} x^\top H x + b^\top x + c, \tag{6}$$

   o passo ótimo ao longo da direção $-g$ é dado por:

$$\alpha = \frac{g^\top g}{g^\top H g}, \tag{7}$$

   onde $g = \nabla f(x)$. Esse cálculo fornece o passo ideal em uma única iteração, garantindo convergência exata em funções quadráticas convexas.

4. **Método de Barzilai–Borwein** Ajusta o passo α_k usando informações das duas últimas iterações:

$$\alpha_k = \frac{s_{k-1}^\top s_{k-1}}{s_{k-1}^\top y_{k-1}}, \tag{8}$$

   em que:

$$s_{k-1} = x_k - x_{k-1}, \qquad y_{k-1} = \nabla f_k - \nabla f_{k-1}.$$

   Esse método costuma apresentar **convergência mais rápida** na prática em relação ao passo fixo.

5. **Passo Decrescente** Define um passo que diminui ao longo das iterações:

$$\alpha_k = \frac{1}{k}. \tag{9}$$

   É simples de implementar, mas tende a apresentar **convergência lenta**.

6. **Aplicação à Função Teste** Para a função:

$$f(x, y) = 3x^2 + 3xy + 2y^2 + x - y,$$

   temos a Hessiana constante:

$$H = \begin{bmatrix} 6 & 3 \\ 3 & 4 \end{bmatrix}.$$

   Por se tratar de uma função quadrática, o método de **Busca Exata** é o mais adequado, pois fornece o passo ótimo a cada iteração, resultando em convergência em poucas etapas.

7. A seguir, o código responsável por fazer o cálculo utilizando as funções em questão:

```python
# f, grad e dados do problema
H = np.array([[6., 3.],
              [3., 4.]])
b = np.array([1., -1.])          # grad f(x)=H x + b
x0 = np.array([0., 0.], float)
tol = 1e-5
max_iters = 100000

def f(x):
    x = np.asarray(x)
    return 3*x[0]**2 + 3*x[0]*x[1] + 2*x[1]**2 + x[0] - x[1]

def grad(x):
    return H @ x + b

# 1) Passo fixo
def gd_fixed(alpha=0.1):
    x = x0.copy()
    g = grad(x); it = 0
    while np.linalg.norm(g) > tol and it < max_iters:
        x -= alpha * g
        g = grad(x); it += 1
    return x, it, np.linalg.norm(g), f(x)

# 2) Backtracking (Armijo)
def gd_backtracking(alpha0=1.0, beta=0.5, c=1e-4):
    x = x0.copy(); it = 0
    while it < max_iters:
        g = grad(x)
        if np.linalg.norm(g) <= tol: break
        alpha = alpha0
        fx = f(x)
        # Armijo: f(x - a g) <= f(x) - c a ||g||^2
        while f(x - alpha*g) > fx - c*alpha*(g@g):
            alpha *= beta
        x -= alpha * g
        it += 1
    return x, it, np.linalg.norm(grad(x)), f(x)

# 3) Busca exata (quadr tica)
def gd_exact_linesearch():
    x = x0.copy(); it = 0
    g = grad(x)
    while np.linalg.norm(g) > tol and it < max_iters:
        denom = g @ (H @ g)
        alpha = (g @ g) / denom
        x -= alpha * g
        g = grad(x); it += 1
    return x, it, np.linalg.norm(g), f(x)

# Solu  o anal tica (para confer ncia): H x* = -b
x_star = -np.linalg.solve(H, b)

# Rodando os tr s m todos
xf, itf, ngf, vf = gd_fixed(alpha=0.1)
xb, itb, ngb, vb = gd_backtracking(alpha0=1.0)
xe, ite, nge, ve = gd_exact_linesearch()

print("Solu  o anal tica         :", x_star, " f* =", f(x_star))
print("\nGD passo fixo   =0.1      : x =", xf, "  it =", itf, "  ||
    ↪  f  || =", ngf, "  f =", vf)
print("GD backtracking (Armijo)  : x =", xb, "  it =", itb, "  ||  f  ||
    ↪  =", ngb, "  f =", vb)
```

```
print("GD busca exata (quadr t.) : x =", xe, "  it =", ite, "  || f
    ↪ || =", nge, "  f =", ve)

```

Listing 5: Comparação de três variantes do método do *Gradient Descent*

3 Conclusão

Neste projeto foram estudadas e implementadas diferentes abordagens do método do *Gradient Descent* para determinação de pontos de mínimo de funções de duas variáveis. Foram analisadas estratégias de passo fixo, passo variável (Backtracking/Armijo) e busca exata, além da aplicação de técnicas de *Gradient Ascent* para identificação de máximos locais.

Os resultados obtidos mostraram que:

- O **passo fixo** é simples de implementar, mas sua eficiência depende fortemente da escolha adequada de α; passos grandes podem causar divergência e passos pequenos tornam o processo lento.
- O método de **Backtracking (Armijo)** ajusta o passo de forma adaptativa, garantindo decréscimo monotônico e maior estabilidade, mesmo com escolhas iniciais desfavoráveis.
- A **busca exata** apresentou o melhor desempenho para funções quadráticas, encontrando o ponto ótimo em poucas iterações e com alta precisão, confirmando o comportamento teórico previsto.
- O **passo variável** em geral promove um equilíbrio entre rapidez e robustez, tornando o método mais eficiente em comparação com o passo constante.

De forma geral, o estudo reforça a importância da escolha adequada do tamanho de passo em algoritmos de otimização baseados em gradiente. Em particular, observou-se que, para funções convexas e suavemente diferenciáveis, a utilização de métodos com passo adaptativo ou busca exata resulta em convergência mais rápida e estável, enquanto o passo fixo pode demandar ajustes manuais ou heurísticos.

Como perspectiva futura, recomenda-se investigar técnicas de segunda ordem, como o método de Newton e suas variantes (BFGS, quasi-Newton), bem como a aplicação dos métodos de passo variável em problemas não convexos ou de alta dimensão.