



**Universidade de Brasília**

## **Lista 1: Otimização em RNA.**

Rebeca Chuffi Saccochi

DEPARTAMENTO DE ESTATÍSTICA

19 de setembro de 2025

Prof. Guilherme Rodrigues

Redes Neurais Profundas

Tópicos especiais em Estatística 1

## Observações Iniciais

Tenho pouca experiência com programação (meu primeiro contato foi no início desse ano) e nunca tinha utilizado *R*, então para esse trabalho optei por utilizar *Python* fazendo as alterações necessárias no *Quarto*.

## Bibliotecas

```
# Bibliotecas Python
import numpy as np
import matplotlib.pyplot as plt
import sympy as sp
import math

from scipy.optimize import minimize
```

## Função de estudo

Considere a função

$$f(x_1, x_2) = x_1^4 + x_2^4 + x_1^2 x_2 + x_1 x_2^2 - 20x_1^2 - 15x_2^2 \quad (1)$$

para responder os itens a seguir.

## A. Curvas de Nível e Pontos Críticos

Considere a função dada pela equação 1. Temos que  $f : \mathbb{R}^2 \rightarrow \mathbb{R}$  é uma função polinomial infinitamente diferenciável, ou seja, de classe  $C^\infty$ . As **curvas de nível** de uma função são as linhas que unem os pontos que tem a mesma “altura”. Nesse caso, para  $c \in \mathbb{R}$ , seriam os pontos tais que

$$f(x_1, x_2) = x_1^4 + x_2^4 + x_1^2 x_2 + x_1 x_2^2 - 20x_1^2 - 15x_2^2 = c \quad (2)$$

Vamos usar o **Matplotlib** com a função **contourf** para plotar o gráfico das curvas de nível:

1. Inicialmente testei o código abaixo variando os valores de  $x_i$  entre  $-100$  e  $100$  e considerando 2000 igualmente distribuídos nesse intervalo.

2. Foram consideradas 50 curvas de nível (possibilidades para  $c$ ).

```
#definindo a função f no python
def f(x1,x2):
    return x1**4 + x2**4 + x1**2 * x2 + x1 * x2**2 - 20*x1**2 - 15 * x2**2

# Folha de estilo
plt.style.use("ggplot")

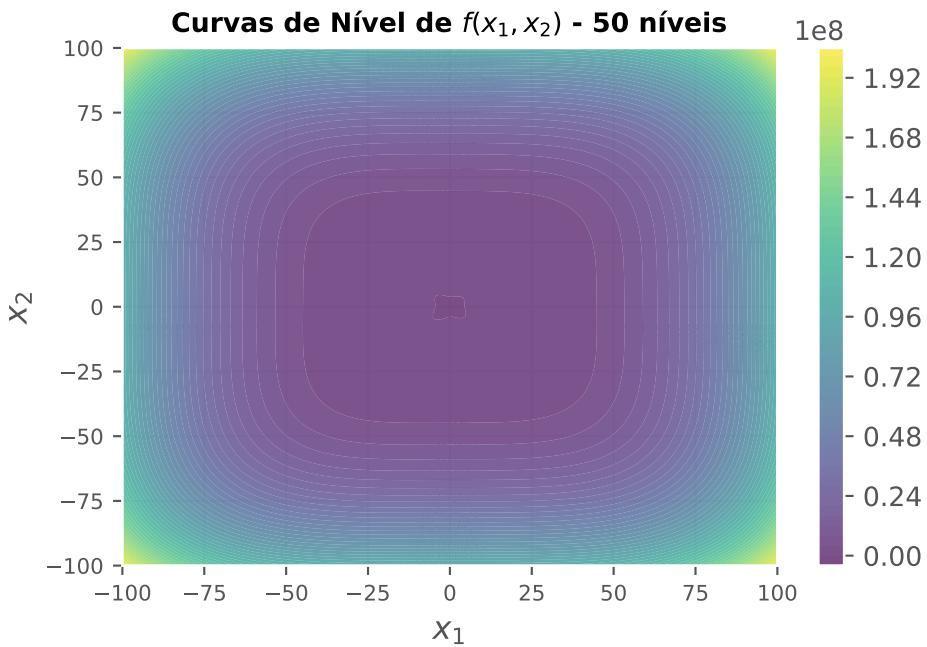
plt.rc("axes", facecolor="#fafafa", grid=True)
plt.rc("grid", color="#f0f0f0")

x1 = np.linspace(-100, 100, 2000)
x2 = np.linspace(-100, 100, 2000)
x1, x2 = np.meshgrid(x1, x2)

Z = f(x1, x2)

plt.contourf(x1, x2, Z, 50, cmap='viridis', alpha=0.7)
plt.colorbar()
plt.title('Curvas de Nível de $f(x_1, x_2)$ - 50 níveis',
          fontsize=10, fontweight='bold')
plt.xlabel('$x_1$', fontsize=12)
plt.ylabel('$x_2$', fontsize=12)
plt.xticks(fontsize=8)
plt.yticks(fontsize=8)

plt.show()
```



Note que o gráfico ficou um pouco distorcido e não conseguimos visualizar com clareza o que acontece no centro dele. Vamos modificar os parâmetros para a seguinte configuração:

1. Valores de  $x_i$  entre  $-5$  e  $5$  e considerando  $100$  pontos igualmente distribuídos nesse intervalo.
2.  $30$  curvas de nível (possibilidades para  $c$ ).

```
# Folha de estilo
plt.style.use("ggplot")
plt.rc("axes", facecolor="#fafafa", grid=True)
plt.rc("grid", color="#f0f0f0")

x1 = np.linspace(-5, 5, 100)
x2 = np.linspace(-5, 5, 100)
x1, x2 = np.meshgrid(x1, x2)

Z = f(x1, x2)

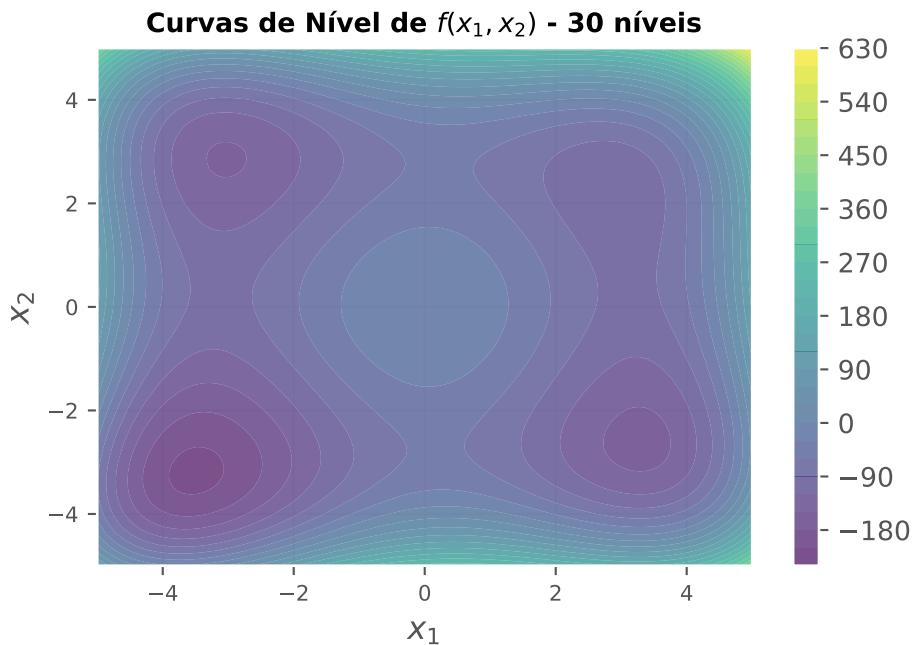
plt.contourf(x1, x2, Z, 30, cmap='viridis', alpha=0.7)
plt.colorbar()
plt.title('Curvas de Nível de $f(x_1, x_2)$ - 30 níveis',
          fontsize=10, fontweight='bold')
plt.xlabel('$x_1$', fontsize=12)
```

```

plt.ylabel('$x_2$', fontsize=12)
plt.xticks(fontsize=8)
plt.yticks(fontsize=8)

plt.show()

```



Note que, agora, conseguimos capturar elementos que o primeiro gráfico não conseguiu (por serem elementos muito pequenos considerando a escala escolhida anteriormente). A imagem da função varia (no intervalo escolhido) entre valores próximos a  $-180$  até valores próximos de  $630$ . Ao analisar a estrutura da função, podemos facilmente ver que  $f(0,0) = 0$ , que é um ponto de nível 0 no gráfico. (apenas uma checagem para verificar se temos algum erro imediato de programação).

Sabemos que curvas de nível mais próximas indicam uma parte mais íngreme da função, que é o que ocorre a medida que o gráfico se aproxima das extremidades da área demarcada. Sobre os candidatos *pontos críticos*, note que temos três áreas mais escuras próximas aos pontos  $(-3.5, 3)$ ,  $(-3.5, -3)$  e  $(3.5, -3.8)$  que podem indicar pontos críticos mínimos (não sabemos se locais ou globais). A região próxima ao ponto  $(-3.5, -3)$  parece ser mais escura, indicando a possibilidade de um mínimo global (ou menor que os outros dois pontos citados). Pelo primeiro gráfico, podemos notar que mesmo considerando um domínio maior, o gráfico não indica valores mínimos fora da caixa que estamos considerando no segundo gráfico (aqui, claro, considerando uma análise totalmente visual que pode estar sendo enganosa num primeiro momento).

## B. Gradiente de $f$

Nesta parte do problema, estamos interessados em encontrar o **gradiente** da função para entender se nossos palpites da parte A estavam no caminho certo. Como temos duas variáveis e uma função com resultado real, o gradiente pode ser representado por:

$$\nabla_x f(x_1, x_2) = \left( \frac{\partial f}{\partial x_1}(x_1, x_2), \frac{\partial f}{\partial x_2}(x_1, x_2) \right) \quad (3)$$

Para o cálculo de derivadas, vamos usar o **sympy**. Para essa função específica poderíamos efetuar os cálculos manualmente, porém vamos deixar um código preparado para que outros problemas possam ser analisados de forma mais sistemática:

```
x1, x2 = sp.symbols('x1 x2')
f = x1**4 + x2**4 + x1**2 * x2 + x1 * x2**2 - 20*x1**2 - 15*x2**2

df_dx1 = sp.diff(f, x1)
df_dx2 = sp.diff(f, x2)

print("Derivada parcial de f com relação a x1:")
print(df_dx1)

print("\nDerivada parcial de f com relação a x2:")
print(df_dx2)
```

Derivada parcial de f com relação a x1:  
 $4x_1^{*3} + 2x_1x_2 - 40x_1 + x_2^{*2}$

Derivada parcial de f com relação a x2:  
 $x_1^{*2} + 2x_1x_2 + 4x_2^{*3} - 30x_2$

Logo,

$$\nabla_x f(x_1, x_2) = (4x_1^3 + x_2^2 + 2x_1x_2 - 40x_1, 4x_2^3 + x_1^2 + 2x_1x_2 - 30x_2) \quad (4)$$

## C. Implementando Método do Gradiente Descendente

Como visto em aula, a ideia do método do **Gradiente Descendente** é partir de um vetor inicial de parâmetros (que nesse caso é representado por  $\Phi = (x_1, x_2)$ ) e andar na *direção oposta* ao gradiente (pois essa será a direção mais provável de um ponto de

mínimo da função). Considerando que  $\Phi_0 = (x_1^0, x_2^0)$  é o parâmetro inicial escolhido para a inicialização. Então, temos a seguinte recorrência:

$$\Phi_{t+1} = \Phi_t - \alpha \nabla_x f(\Phi_t) \quad (5)$$

Como já calculamos o gradiente da função  $f$  em 4, vamos definir uma função que tenha como entrada a taxa de aprendizado  $\alpha$ , o número de passos  $n$  e o ponto de partida  $p = \Phi_0 = (x_1^0, x_2^0)$ :

```
def grad_df(a: float, p: tuple[float, float], n: int):
    # Inicializa o vetor de pontos p com a entrada fornecida
    pontos = [p]

    def gradf(x1, x2):
        return [4*x1**3 + 2*x1*x2 - 40*x1 + x2**2,
               x1**2 + 2*x1*x2 + 4*x2**3 - 30*x2]
    grad = gradf(pontos[0][0], pontos[0][1])

    for t in range(n):
        grad = gradf(pontos[t][0], pontos[t][1])
        pd = (pontos[t][0] - a * grad[0], pontos[t][1] - a * grad[1])
        pontos.append(pd)
    print(f'Ao inicializar no ponto {p}, com {n} passos e taxa')
    print(f'de aprendizagem {a} chegamos a')
    print(f'{pontos[-1]}')
    print(f'pelo método do Gradiente Descendente')
    return pontos[-1]
```

## D. Testando Função Gradiente

Considere o ponto inicial  $p = (0, 5)$  com taxa de aprendizado de 0.01 e 100 passos executados:

```
p = (0,5)
n = 100
a = 0.01

grad_df(a,p,n)
```

Ao inicializar no ponto (0, 5), com 100 passos e taxa de aprendizagem 0.01 chegamos a

(-3.0401412706844373, 2.865981423744074)  
pelo método do Gradiente Descendente

(-3.0401412706844373, 2.865981423744074)

## E. Variando Taxa de Aprendizagem

Agora vamos **variar a taxas de aprendizagem** mantendo o número de passos e o ponto inicial iguais aos do item anterior. Testando os valores menores, temos:

```
taxas = [0.01, 0.001, 0.0001]
p = (0, 5)
n = 100

for a in taxas:
    def f(x1, x2):
        return x1**4+x2**4+x1**2*x2+x1*x2**2-20*x1**2-15*x2**2
    resultado = grad_df(a, p, n)
    fun = f(resultado[0], resultado[1])
    print(f'O valor de f nesse ponto é de {fun}')
    print()
```

Ao inicializar no ponto (0, 5), com 100 passos e taxa de aprendizagem 0.01 chegamos a  
(-3.0401412706844373, 2.865981423744074)  
pelo método do Gradiente Descendente  
O valor de f nesse ponto é de -153.6490976846798

Ao inicializar no ponto (0, 5), com 100 passos e taxa de aprendizagem 0.001 chegamos a  
(-2.9401850139854755, 2.863626932709569)  
pelo método do Gradiente Descendente  
O valor de f nesse ponto é de -153.27803824887297

Ao inicializar no ponto (0, 5), com 100 passos e taxa de aprendizagem 0.0001 chegamos a  
(-0.1983707534904601, 3.489303251581688)  
pelo método do Gradiente Descendente  
O valor de f nesse ponto é de -37.45553147126776

Considerando as taxas 0.01 e 0.001 chegamos a pontos com valores muito próximos:  $-153.649$  e  $-153.278$  respectivamente. Esses valores estão associados ao ponto  $(-3, 2.8)$ , que seria o ponto de mínimo encontrado pelo nosso algoritmo. Se olharmos novamente para o gráfico de curvas de nível da parte A., concluímos que nas condições citadas o algoritmo pára próximo à região superior esquerda, um dos pontos que notamos ser mais escuro no gráfico.

Já a taxa de 0.0001 parece ser muito pequena para a quantidade de passos (100) que estamos considerando, pois o valor da função no ponto no qual o algoritmo pára é de aproximadamente  $-37.455$ , número muito maior que os dois anteriores. Esse valor está associado ao ponto  $(-0.198, 3.489)$ , valor um pouco à direita da região mais escura do gráfico (superior esquerda). A impressão que temos é que “estava chegando lá”, mas não tivemos passos suficientes para atingir o objetivo (considerando o pequeno tamanho de passo).

Agora quando consideramos as taxas 1 e 0.1, não foi possível finalizar o algoritmo pois o Python indicou valor muito alto para a magnitude gradiente depois de algumas etapas, ou seja: o valor da taxa não é suficiente para impedir que o valor do gradiente exploda em termos de módulo.

Para tentar monitorar essa questão, considere a seguinte função (que monitora a magnitude do gradiente em cada etapa):

```
def gradf(x1, x2):
    grad_x1 = 4 * x1**3 + 2 * x1 * x2 - 40 * x1 + x2**2
    grad_x2 = x1**2 + 2 * x1 * x2 + 4 * x2**3 - 30 * x2
    return [grad_x1, grad_x2]

def grad_dfm(a, p, n):
    t = 0
    p = [p]

    while t < n:

        grad = gradf(p[t][0], p[t][1])
        grad_magnitude = (grad[0]**2 + grad[1]**2)**0.5
        print(f"Tamanho do gradiente na iteração {t}: {grad_magnitude}")
        p.append([p[t][0] - a * grad[0], p[t][1] - a * grad[1]])
        t += 1

    return p[-1]
```

Vamos testar apenas para  $n = 4$ , pois depois disso obtivemos a seguinte mensagem de erro:

```
OverflowError: int too large to convert to float
```

```
a = 1
p = (0, 5)
n = 4

grad_dfm(a, p, n)
```

```
Tamanho do gradiente na iteração 0: 350.891721190455
Tamanho do gradiente na iteração 1: 164226292.02316007
Tamanho do gradiente na iteração 2: 1.771679586077537e+25
Tamanho do gradiente na iteração 3: 2.224413563985978e+76
```

```
[-249326014834506929207357610621490765735384600378600,
 22244135639859779482203935983307886306561487197708452939075445824682438682730]
```

Note que na quarta iteração o gradiente já é da ordem de  $e^{76}$ , o que torna o processo impraticável. Assim como para taxa de 1, a taxa de 0.1 não é pequena o suficiente:

```
a = 0.1
p = (0, 5)
n = 5

grad_dfm(a, p, n)
```

```
Tamanho do gradiente na iteração 0: 350.891721190455
Tamanho do gradiente na iteração 1: 106949.27919491791
Tamanho do gradiente na iteração 2: 4851387622580.395
Tamanho do gradiente na iteração 3: 4.567282658546349e+35
Tamanho do gradiente na iteração 4: 3.810953600009575e+104
```

```
[-2.0369409624887195e+68, -3.810953600009575e+103]
```

Na quinta iteração o gradiente já é da ordem de  $e^{104}$ , o que torna o processo impraticável.

Com isso, concluímos que para o ponto inicial  $(0, 5)$  e considerando 100 passos a taxa de aprendizagem de 0.01 é a mais adequada. A taxa de 0.001 não perde por muito, logo, se formos considerar mais passos, essa pode ser uma outra opção.

## F. Modificando o Ponto Inicial

Agora vamos gerar 10 pontos aleatórios escolhidos a partir de um quadrado  $-5 < x_1, x_2 < 5$  (uniformemente) e refazer o gráfico do item A.

```
np.random.seed(42)

x1 = np.random.uniform(-5, 5, 10)
x2 = np.random.uniform(-5, 5, 10)

pontos_aleatorios = []

for i in range(len(x1)):
    pontos_aleatorios.append((float(x1[i]), float(x2[i])))

print(pontos_aleatorios)
```

```
[(-1.254598811526375, -4.7941550570419755), (4.507143064099161, 4.699098521619943), (2.38766088932172384), (-3.439813595575635, -3.181750327928994), (-3.4400547966379733, -3.165954901465662), (-4.419163878318005, -1.9575775704046228), (3.6617614577493516, 0.240.6805498135788426), (2.0807257779604544, -2.0877085980195806)]
```

Vamos refazer o gráfico do item a) adicionando uma linha representando o caminho percorrido por cada uma das 10 otimizações (com os 10 pontos distintos e considerando a taxa de aprendizado de 0.01). Para isso, vamos ter que modificar um pouco a função com o objetivo de armazenar os pontos de cada etapa do Gradiente Descendente:

```
#modificando a função criada anteriormente
#para exibir a lista de pontos gerada e não
#apenas o último ponto

def grad_df_list(a: float, p: tuple[float, float], n: int):
    # Inicializa o vetor de pontos p com a entrada fornecida
    pontos = [p]

    def gradf(x1, x2):
        return [4*x1**3 + 2*x1*x2 - 40*x1 + x2**2,
               x1**2 + 2*x1*x2 + 4*x2**3 - 30*x2]
    grad = gradf(pontos[0][0], pontos[0][1])

    for t in range(n):
        grad = gradf(pontos[t][0], pontos[t][1])
```

```

pd = (pontos[t][0] - a * grad[0], pontos[t][1] - a * grad[1])
pontos.append(pd)

return pontos

# Plotando o gráfico usado em a

plt.style.use("ggplot")
plt.rc("axes", facecolor="#fafafa", grid=True) # Cor do fundo
plt.rc("grid", color="#f0f0f0") # Cor do quadriculado

x1 = np.linspace(-5, 5, 100)
x2 = np.linspace(-5, 5, 100)
X1, X2 = np.meshgrid(x1, x2)

Z = f(X1, X2)

plt.contourf(X1, X2, Z, 30, cmap='viridis', alpha=0.7)
plt.colorbar()
plt.title('Diferentes inicializações - Gradiente Descendente de f',
          fontsize=8, fontweight='bold')
plt.xlabel('$x_1$', fontsize=12)
plt.ylabel('$x_2$', fontsize=12)
plt.xticks(fontsize=8)
plt.yticks(fontsize=8)

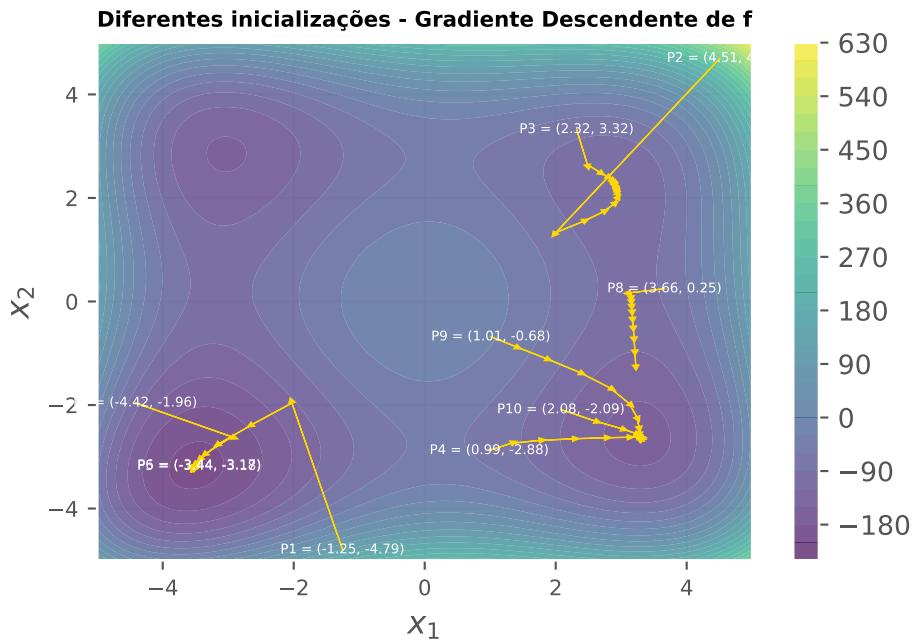
for p in pontos_aleatorios:
    caminho = grad_df_list(0.01, p, 10)

    caminho_x = [point[0] for point in caminho]
    caminho_y = [point[1] for point in caminho]

    for i in range(len(caminho)-1):
        plt.arrow(caminho_x[i], caminho_y[i],
                  caminho_x[i+1] - caminho_x[i], caminho_y[i+1] - caminho_y[i],
                  head_width=0.1, head_length=0.1, fc='#FFD700',
                  ec='#FFD700', linewidth=0.5)
    plt.text(caminho_x[0], caminho_y[0],
             f'P{pontos_aleatorios.index(p) + 1} = ({round(caminho_x[0], 2)}, {round(caminho_y[0], 2)})',
             fontsize=5, ha='center', va='center', color='white')

```

```
plt.show()
```



Analizando a figura acima, note que os pontos  $P_2$  e  $P_3$ , que foram inicializados na parte superior direita chegaram no mínimo (que parece um mínimo local) do canto superior direito depois de 100 passos. Já  $P_4$ ,  $P_8$ ,  $P_9$ ,  $P_{10}$ , depois de 100 passos, caíram no mínimo (possivelmente local) do canto inferior direito. Agora  $P_5$ ,  $P_1$ ,  $P_3$ , que são pontos localizados no canto inferior direito caíram no mínimo (aparentemente global - ou menor que os outros por ser mais escuro) no canto inferior esquerdo.

Note também que  $P_1$ ,  $P_2$ ,  $P_3$  têm tamanhos do primeiro passo grandes, o que indica uma magnitude maior do gradiente nessa etapa. Felizmente a taxa de aprendizado foi suficientemente pequena para que o algoritmo convergisse para um ponto mínimo.

Portanto, concluímos que dos 10 pontos selecionados aleatoriamente, todos caíram num mínimo já observado em no gráfico anterior (todos convergiram), mas apenas 3 deles para um mínimo (aparentemente) global, ou seja, 30 das vezes.

## G. Método do Gradiente com Momentum

Tendo em vista algumas atualizações que podemos realizar no método original do Gradiente Descendente, um deles é adicionar *momentum* no cálculo do mesmo. Considere como referência a *Seção 8.3.2* do livro **Deep Learning** e use a taxa de aprendizado  $\epsilon = 0.01$ , parâmetro de momento  $\alpha = 0.9$  e velocidade inicial  $v = 0$ . Vamos inic平izar

com o ponto (0.5) e 100 passos. O método do *Gradiente com Momento* adiciona um fator de “memória” que leva em consideração contribuição de cada gradiente.

Considerando a notação apresentada na referência citada acima e o caso específico da nossa função  $f$ , temos que  $\Phi_0$  é o ponto de inicialização e  $\Phi_t$  o ponto anterior à movimentação para  $\Phi_{t+1}$ :

$$v \leftarrow \alpha v - \epsilon \nabla_x f(\Phi_t) \quad (6)$$

$$\Phi_{t+1} \leftarrow \Phi_t + v \quad (7)$$

Ao adaptar a função criada anteriormente, temos:

```
def grad_f_mome(a: float, p: tuple[float, float], n: int, e: float):
    pontos = [p]

    def gradf(x1, x2):
        return [4*x1**3 + 2*x1*x2 - 40*x1 + x2**2,
               x1**2 + 2*x1*x2 + 4*x2**3 - 30*x2]
    grad = gradf(pontos[0][0], pontos[0][1])
    v = [0, 0]

    for t in range(n):
        grad = gradf(pontos[t][0], pontos[t][1])
        v = [a * v[i] - e * grad[i] for i in range(len(v))]
        #acima escolhemos a notação em len(v) para ser escalável em
        #mais dimensões
        pd = (pontos[t][0] + v[0], pontos[t][1] + v[1])
        pontos.append(pd)

    print(f'Ao inicializar no ponto {p}, com {n} passos e parâmetros ')
    print(f'a = {a}, e = {e}')
    print(f'pelo método do Gradiente Descendente com Momento, obtivemos:')
    print(f'{pontos[-1]}')
    return pontos[-1]
```

Agora vamos testar a função para os parâmetros sugeridos:

```
def f(x1, x2):
    return x1**4+x2**4+x1**2*x2+x1*x2**2-20*x1**2-15*x2**2
resultado = grad_f_mome(0.9, (0,5), 100, 0.01)
fun = f(resultado[0], resultado[1])
```

```
print(f'O valor de f nesse ponto é de {fun}')
print()
```

Ao inicializar no ponto (0, 5), com 100 passos e parâmetros  
 $a = 0.9$ ,  $e = 0.01$   
pelo método do Gradiente Descendente com Momento, obtivemos:  
(3.275777117878277, -2.617305302766108)  
O valor de f nesse ponto é de -160.9394732174183

Note que, considerando nossa primeira função do método do Gradiente Descendente o resultado foi: (-3.0401412706844373, 2.865981423744074), com valor de função igual à -153.6490976846798. Já usando o método de Gradiente Descendente com Momento, chegamos no ponto (3.275777117878277, -2.617305302766108) e função igual à -160.9394732174183 com a mesma quantidade de passos. Se olharmos para o gráfico de curvas de nível, vamos notar que o método com o momento não pára no ponto crítico superior esquerdo, como aconteceu antes. Nesse caso, ele vai para um outro ponto crítico (outro mínimo local) na área inferior direita. Além disso, se fizermos uma modificação na taxa  $\alpha = 0.001$  conseguimos chegar próximo ao mínimo (possivelmente) global:

```
def f(x1, x2):
    return x1**4+x2**4+x1**2*x2+x1*x2**2-20*x1**2-15*x2**2
resultado = grad_f_mome(0.9, (0,5), 100, 0.001)
fun = f(resultado[0], resultado[1])
print(f'O valor de f nesse ponto é de {fun}')
print()
```

Ao inicializar no ponto (0, 5), com 100 passos e parâmetros  
 $a = 0.9$ ,  $e = 0.001$   
pelo método do Gradiente Descendente com Momento, obtivemos:  
(-3.5243911571405624, -3.202209041020609)  
O valor de f nesse ponto é de -218.71715905030408

## H. Método do Gradiente RMSProp

Agora vamos utilizar o método **RMSProp** da *Seção 8.5.2* do livro **Deep Learning** e usar a taxa de aprendizado global  $\epsilon = 0.001$ , taxa de decaimento  $\rho = 0.9$  e constante  $\delta = 10^{-6}$ . O algoritmo RMSProp ajusta a taxa de aprendizado com base nos gradientes passados normalizando através da raiz quadrada de uma média exponencial dos gradientes anteriores (quadrados).

Considerando a notação apresentada na referência citada acima e o caso específico da nossa função  $f$ , temos que  $\Phi_0$  é o ponto de inicialização e  $\Phi_t$  o ponto anterior à movimentação para  $\Phi_{t+1}$ . Além disso, usaremos  $\odot$  para denotar a operação que multiplica elemento a elemento dentro de uma matriz. Ou seja,  $g \odot g$  seria o gradiente “vezes” o gradiente (elemento a elemento) o que resultaria no quadrado de cada entrada. Chamaremos de  $r$  o quadrado acumulado do gradiente:

$$r \leftarrow \rho r + (1 - \rho) \nabla_x f(\Phi_t) \odot \nabla_x f(\Phi_t) = \rho r + (1 - \rho) g \odot g \quad (8)$$

$$\Delta\Phi_t \leftarrow -\frac{\epsilon}{\delta + \sqrt{r}} \odot g \quad (9)$$

$$\Phi_{t+1} \leftarrow \Phi_t + \Delta\Phi_t \quad (10)$$

Ao adaptar a função criada anteriormente, temos:

```
def grad_f_rmsp(p: tuple[float, float], n: int, e: float, rh: float, de: float):
    # Inicializa o vetor de pontos p com a entrada fornecida
    pontos = [p]

    def gradf(x1, x2):
        return [4*x1**3 + 2*x1*x2 - 40*x1 + x2**2,
               x1**2 + 2*x1*x2 + 4*x2**3 - 30*x2]

    r = [0, 0]

    for t in range(n):
        grad = gradf(pontos[t][0], pontos[t][1])
        grad_2 = (grad[0]**2, grad[1]**2)

        r = [r[i]*rh + (1-rh)*grad_2[i] for i in range(len(r))]
        #acima escolhemos a notação em len(r) para ser escalável em
        #mais dimensões

        d = [-(e / (math.sqrt(r[i] + de))) * grad[i] for i in range(len(r))]
        pd = (pontos[t][0] + d[0], pontos[t][1] + d[1])
        pontos.append(pd)

    print(f'Ao inicializar no ponto {p}, com {n} passos e parâmetros ')
    print(f'rh = {rh}, e = {e}')
    print(f'pelo método do Gradiente RMSProp, obtivemos:')
    print(f'{pontos[-1]}')
    return pontos[-1]
```

Agora vamos aplicar a função acima nos parâmetros indicados:

```
def f(x1, x2):
    return x1**4+x2**4+x1**2*x2+x1*x2**2-20*x1**2-15*x2**2
resultado = grad_f_rmsp((0,5), 100, 0.001, 0.9, math.pow(10, -6))
fun = f(resultado[0], resultado[1])
print(f'O valor de f nesse ponto é de {fun}')
print()
```

Ao inicializar no ponto (0, 5), com 100 passos e parâmetros  
rh = 0.9, e = 0.001  
pelo método do Gradiente RMSProp, obtivemos:  
(-0.10979229843368189, 4.891533086159593)  
O valor de f nesse ponto é de 210.7904925118437

Passei algumas horas tentando entender porque a performance do algoritmo RMSProp está pior que o anterior. Revisei o código e acredito que está tudo correto, então comecei a fazer outros testes para tentar entender o problema. Aparentemente o número de interações é muito pequeno para o tamanho da taxa escolhida de 0.01 e isso acontece porque, começando com  $r$  muito pequeno, o passo fica muito grande inicialmente, logo não temos tempo (interações) suficientes para que o ponto converja para um ponto de mínimo. Como  $\rho$  nesse caso também é muito pequeno, ele não consegue acelerar esse processo. A solução desse problema (para o mesmo número de passos) será feita na próxima questão.

Considere os seguintes exemplos (mudando os parâmetros):

```
#2000 interações
def f(x1, x2):
    return x1**4+x2**4+x1**2*x2+x1*x2**2-20*x1**2-15*x2**2
resultado = grad_f_rmsp((0,5), 2000, 0.001, 0.9, math.pow(10, -6))
fun = f(resultado[0], resultado[1])
print(f'O valor de f nesse ponto é de {fun}')
print()
```

Ao inicializar no ponto (0, 5), com 2000 passos e parâmetros  
rh = 0.9, e = 0.001  
pelo método do Gradiente RMSProp, obtivemos:  
(-2.0143629991659813, 3.021320406339329)  
O valor de f nesse ponto é de -124.41534244930556

```
#1000 interações
def f(x1, x2):
    return x1**4+x2**4+x1**2*x2+x1*x2**2-20*x1**2-15*x2**2
resultado = grad_f_rmsp((0,5), 1000, 0.01, 0.9, math.pow(10, -6))
fun = f(resultado[0], resultado[1])
print(f'0 valor de f nesse ponto é de {fun}')
print()
```

Ao inicializar no ponto (0, 5), com 1000 passos e parâmetros  
 $\eta = 0.9$ , e = 0.01  
pelo método do Gradiente RMSProp, obtivemos:  
(-3.0351291217117033, 2.8609681572725414)  
O valor de f nesse ponto é de -153.64734400284024

```
#1000 interações
def f(x1, x2):
    return x1**4+x2**4+x1**2*x2+x1*x2**2-20*x1**2-15*x2**2
resultado = grad_f_rmsp((0,5), 100, 0.1, 0.9, math.pow(10, -6))
fun = f(resultado[0], resultado[1])
print(f'0 valor de f nesse ponto é de {fun}')
print()
```

Ao inicializar no ponto (0, 5), com 100 passos e parâmetros  
 $\eta = 0.9$ , e = 0.1  
pelo método do Gradiente RMSProp, obtivemos:  
(-3.034475077300627, 2.8208551852942096)  
O valor de f nesse ponto é de -153.5851972977133

Concluímos que para usar 100 passos e atingir um valor próximo de um mínimo (ainda usando esse algoritmo) é preciso aumentar o  $\epsilon$ , como foi feito no último exemplo. Para usar  $\epsilon = 0.001$  tivemos que aumentar o número de interações para 2000 para chegarmos num resultado aceitável (como no algoritmo que usa momento)

## I. Método Adam

Finalmente chegamos ao **Adam**, método amplamente utilizado. Como referência temos a Seção 8.5.3 do livro **Deep Learning** e usando a taxa de aprendizado global  $\epsilon = 0.001$  e taxas de decaimento  $\rho_1 = 0.9$  e  $\rho_2 = 0.999$ . O algoritmo Adam é exatamente a combinação do RMSProp e do gradiente considerando o Momento. De maneira geral, ele trabalha com uma taxa adaptativa que faz com que o passo seja grande no início

(para que o algoritmo possa explorar o espaço) e pequeno no fim, afunilando nos pontos de importância. Além disso, por ter momento, ele considera a “memória” passada dos gradientes das iterações passadas.

Considerando a notação apresentada na referência citada acima e considerando o caso específico da nossa função  $f$ , temos que:

$$r \leftarrow \rho_2 r + (1 - \rho_2) \nabla_x f(\Phi_t) \odot \nabla_x f(\Phi_t) = \rho_2 r + (1 - \rho_2) g \odot g \quad (11)$$

$$s \leftarrow \rho_1 s + (1 - \rho_1) \nabla_x f(\Phi_t) \odot \nabla_x f(\Phi_t) = \rho_1 r + (1 - \rho_1) g \quad (12)$$

$$\hat{s} \leftarrow \frac{s}{1 - \rho_1^t} \quad (13)$$

$$\hat{r} \leftarrow \frac{r}{1 - \rho_2^t} \quad (14)$$

$$\Delta\Phi_t = -\epsilon \frac{\hat{s}}{\sqrt{\hat{r} + \delta}} \quad (15)$$

$$\Phi_{t+1} \leftarrow \Phi_t + \Delta\Phi_t \quad (16)$$

Ao adaptar a função criada anteriormente, temos:

```
def ADAM_f(p: tuple[float, float], n: int, e: float, rh1: float, rh2: float, de: float):
    # Inicializa o vetor de pontos p com a entrada fornecida
    pontos = [p]

    def gradf(x1, x2):
        return [4*x1**3 + 2*x1*x2 - 40*x1 + x2**2,
               x1**2 + 2*x1*x2 + 4*x2**3 - 30*x2]

    r = [0, 0]
    s = [0, 0]

    for t in range(n):
        grad = gradf(pontos[t][0], pontos[t][1])
        grad_2 = (grad[0]**2, grad[1]**2)

        r = [r[i]*rh2 + (1-rh2)*grad_2[i] for i in range(len(r))]
        #acima escolhemos a notação em len(r) para ser escalável em
```

```

#mais dimensões
s = [s[i]*rh1 + (1-rh1)*grad[i] for i in range(len(s))]

s_hat = [s[i] / (1-rh1**(t+1)) for i in range(len(s))]
r_hat = [r[i] / (1-rh2**(t+1)) for i in range(len(r))]

d = [-(e * s_hat[i]) / (math.sqrt(r_hat[i]) + de) for i in range(len(s_hat))]
pd = (pontos[t][0] + d[0], pontos[t][1] + d[1])
pontos.append(pd)

print(f'Ao inicializar no ponto {p}, com {n} passos e parâmetros ')
print(f'rh1 = {rh1}, e = {e} e rh2 = {rh2}')
print(f'pelo método ADAM, obtivemos:')
print(f'{pontos[-1]}')
return pontos[-1]

```

Note que, na função acima, modificamos  $\hat{s}$  e  $\hat{r}$  para potência de  $t+1$  no lugar de potência de  $t$  para evitar divisão por zero.

```

#100 interações
def f(x1, x2):
    return x1**4+x2**4+x1**2*x2+x1*x2**2-20*x1**2-15*x2**2
resultado = ADAM_f((0,5), 100, 0.001, 0.9, 0.999, math.pow(10, -7))
fun = f(resultado[0], resultado[1])
print(f'O valor de f nesse ponto é de {fun}')
print()

```

```

Ao inicializar no ponto (0, 5), com 100 passos e parâmetros
rh1 = 0.9, e = 0.001 e rh2 = 0.999
pelo método ADAM, obtivemos:
(-0.1013468156442363, 4.901360793367915)
O valor de f nesse ponto é de 214.1810222591261

```

Bom, aparentemente encontramos o mesmo problema anterior para o ADAM (poucas interações). Vamos modificar a função para adicionar um debug visual:

```

def ADAM_f2(p: tuple[float, float], n: int, e: float, rh1: float, rh2: float, de: float):
    pontos = [p]

    def gradf(x1, x2):
        return [4*x1**3 + 2*x1*x2 - 40*x1 + x2**2,
               x1**2 + 2*x1*x2 + 4*x2**3 - 30*x2]

```

```

def f(x1, x2):
    return x1**4 + x2**4 + x1**2*x2 + x1*x2**2 - 20*x1**2 - 15*x2**2

s = [0, 0]
r = [0, 0]

print("Iteração | x1           | x2           | f(x)       | Gradiente")
print("-" * 50)

for t in range(n):
    grad = gradf(pontos[t][0], pontos[t][1])
    grad_2 = [grad[0]**2, grad[1]**2]

    s = [s[i]*rh1 + (1-rh1)*grad[i] for i in range(len(s))]
    r = [r[i]*rh2 + (1-rh2)*grad_2[i] for i in range(len(r))]

    s_hat = [s[i] / (1-rh1**2*(t+1)) if t > 0 else s[i] for i in range(len(s))]
    r_hat = [r[i] / (1-rh2**2*(t+1)) if t > 0 else r[i] for i in range(len(r))]

    d = [-e * s_hat[i] / (math.sqrt(max(r_hat[i], 1e-10)) + de) for i in range(len(d))

    pd = (pontos[t][0] + d[0], pontos[t][1] + d[1])
    pontos.append(pd)

    # Debug a cada 20 iterações
    if t % 10 == 0:
        current_f = f(pontos[t][0], pontos[t][1])
        grad_norm = math.sqrt(grad[0]**2 + grad[1]**2)
        print(f"{t:8} | {pontos[t][0]:8.4f} | {pontos[t][1]:8.4f} | {current_f:8.2f}")

    final_f = f(pontos[-1][0], pontos[-1][1])
    print(f"\nResultado final: {pontos[-1]}")
    print(f"Valor da função: {final_f}")

return pontos[-1]

```

```
resultado = ADAM_f2((0,5), 100, 0.001, 0.9, 0.999, math.pow(10, -7))
```

Iteração   x1             x2             f(x)         Gradiente					
0   0.0000   5.0000   250.00   350.8917					
10   -0.0122   4.9878   245.46   347.5228					

20		-0.0222		4.9779		241.76		344.7710
30		-0.0322		4.9679		238.09		342.0385
40		-0.0423		4.9580		234.46		339.3271
50		-0.0524		4.9481		230.87		336.6377
60		-0.0625		4.9382		227.31		333.9706
70		-0.0727		4.9284		223.78		331.3260
80		-0.0830		4.9186		220.30		328.7038
90		-0.0932		4.9089		216.84		326.1038

Resultado final: (-0.10351119046083711, 4.899203282142462)  
 Valor da função: 213.42618796046366

Note que o gradiente está se aproximando de zero muito lentamente e com isso o valor da função está diminuindo lentamente também. Vamos considerar um número de passos maior:

```
def f(x1, x2):
    return x1**4+x2**4+x1**2*x2+x1*x2**2-20*x1**2-15*x2**2
resultado = ADAM_f((0,5), 1000, 0.001, 0.9, 0.999, 1e-8)
fun = f(resultado[0], resultado[1])
print(f'O valor de f nesse ponto é de {fun}')
print()
```

Ao inicializar no ponto (0, 5), com 1000 passos e parâmetros  
 $rh_1 = 0.9$ ,  $e = 0.001$  e  $rh_2 = 0.999$   
 pelo método ADAM, obtivemos:  
 (-1.1443028927601104, 4.167447792218679)  
 O valor de f nesse ponto é de 2.2291269668587574

Comparei a minha função criada com o algoritmo pré-definido e deram o mesmo resultado. Ou seja, talvez seja o número de passos e/ou tamanho do passo que não está ideal nesse contexto. Vamos analisar melhor essas questões na parte seguinte.

## J. Comparação de Métodos

Vamos comparar os seguintes métodos:

1. M1: Gradiente Descendente
2. M2: Gradiente Descendente com momento
3. M3: RMSProp

#### 4. M4: ADAM

Usaremos algo semelhante à definição de `grad_df_list` definido anteriormente, para que consigamos capturar os pontos de cada iteração:

#### M2: Gradiente Descendente com momento (lista)

```
def grad_f_mome_list(a: float, p: tuple[float, float], n: int, e: float):
    pontos = [p]

    def gradf(x1, x2):
        return [4*x1**3 + 2*x1*x2 - 40*x1 + x2**2,
               x1**2 + 2*x1*x2 + 4*x2**3 - 30*x2]

    v = [0, 0]

    for t in range(n):
        grad = gradf(pontos[t][0], pontos[t][1])
        v = [a * v[i] - e * grad[i] for i in range(len(v))]
        pd = (pontos[t][0] + v[0], pontos[t][1] + v[1])
        pontos.append(pd)

    return pontos
```

#### M3: RMSProp (lista)

```
def grad_f_rmsp_list(p: tuple[float, float], n: int, e: float, rh: float, de: float):
    pontos = [p]

    def gradf(x1, x2):
        return [4*x1**3 + 2*x1*x2 - 40*x1 + x2**2,
               x1**2 + 2*x1*x2 + 4*x2**3 - 30*x2]

    r = [0, 0]

    for t in range(n):
        grad = gradf(pontos[t][0], pontos[t][1])
        grad_2 = (grad[0]**2, grad[1]**2)

        r = [r[i]*rh + (1-rh)*grad_2[i] for i in range(len(r))]
        d = [-(e / (math.sqrt(r[i] + de))) * grad[i] for i in range(len(r))]
        pd = (pontos[t][0] + d[0], pontos[t][1] + d[1])
        pontos.append(pd)
```

```
    return pontos
```

#### M4: Adam (lista)

```
def ADAM_f_list(p: tuple[float, float], n: int, e: float, rh1: float, rh2: float, de: float):
    pontos = [p]

    def gradf(x1, x2):
        return [4*x1**3 + 2*x1*x2 - 40*x1 + x2**2,
               x1**2 + 2*x1*x2 + 4*x2**3 - 30*x2]

    r = [0, 0]
    s = [0, 0]

    for t in range(n):
        grad = gradf(pontos[t][0], pontos[t][1])
        grad_2 = (grad[0]**2, grad[1]**2)

        r = [r[i]*rh2 + (1-rh2)*grad_2[i] for i in range(len(r))]
        s = [s[i]*rh1 + (1-rh1)*grad[i] for i in range(len(s))]

        s_hat = [s[i] / (1-rh1**2*(t+1)) for i in range(len(s))]
        r_hat = [r[i] / (1-rh2**2*(t+1)) for i in range(len(r))]

        d = [-(e * s_hat[i]) / (math.sqrt(r_hat[i]) + de) for i in range(len(s_hat))]
        pd = (pontos[t][0] + d[0], pontos[t][1] + d[1])
        pontos.append(pd)

    return pontos
```

Agora vamos gerar alguns gráfico comparativos:

```
ponto_inicial = (0,5)
n_passos = 100

trajetoria_gd = grad_df_list(0.001, ponto_inicial, n_passos)
trajetoria_momentum = grad_f_mome_list(0.9, ponto_inicial, n_passos, 0.001)
trajetoria_rmsprop = grad_f_rmfp_list(ponto_inicial, n_passos, 0.001, 0.9, math.pow(10, -8))
trajetoria_adam = ADAM_f_list(ponto_inicial, n_passos, 0.001, 0.9, 0.999, math.pow(10, -8))
```

```

plt.style.use("ggplot")
plt.rc("axes", facecolor="#fafafa", grid=True)
plt.rc("grid", color="#f0f0f0")

plt.figure(figsize=(12, 10))
plt.contourf(X1, X2, Z, 30, cmap='viridis', alpha=0.7)
plt.colorbar()

gd_x = [p[0] for p in trajetoria_gd]
gd_y = [p[1] for p in trajetoria_gd]

momentum_x = [p[0] for p in trajetoria_momentum]
momentum_y = [p[1] for p in trajetoria_momentum]

rmsprop_x = [p[0] for p in trajetoria_rmsprop]
rmsprop_y = [p[1] for p in trajetoria_rmsprop]

adam_x = [p[0] for p in trajetoria_adam]
adam_y = [p[1] for p in trajetoria_adam]

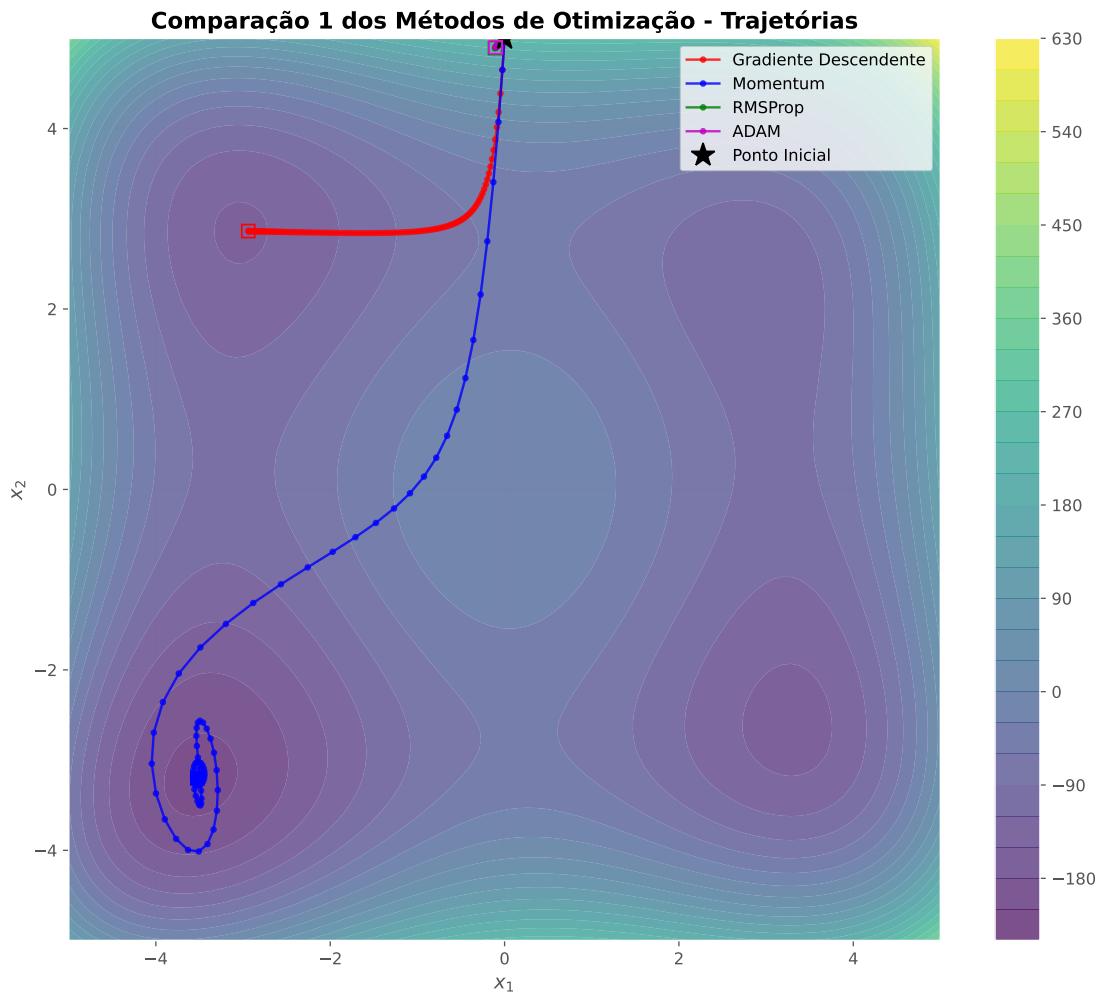
plt.plot(gd_x, gd_y, 'ro-', markersize=3, linewidth=1.5, label='Gradiente Descendente', alpha=0.8)
plt.plot(momentum_x, momentum_y, 'bo-', markersize=3, linewidth=1.5, label='Momentum', alpha=0.8)
plt.plot(rmsprop_x, rmsprop_y, 'go-', markersize=3, linewidth=1.5, label='RMSProp', alpha=0.8)
plt.plot(adam_x, adam_y, 'mo-', markersize=3, linewidth=1.5, label='ADAM', alpha=0.8)

plt.plot(ponto_inicial[0], ponto_inicial[1], 'k*', markersize=15, label='Ponto Inicial')

plt.plot(gd_x[-1], gd_y[-1], 'rs', markersize=8, markeredgecolor='red', markerfacecolor='white')
plt.plot(momentum_x[-1], momentum_y[-1], 'bs', markersize=8, markeredgecolor='blue', markerfacecolor='white')
plt.plot(rmsprop_x[-1], rmsprop_y[-1], 'gs', markersize=8, markeredgecolor='green', markerfacecolor='white')
plt.plot(adam_x[-1], adam_y[-1], 'ms', markersize=8, markeredgecolor='magenta', markerfacecolor='white')

plt.xlabel('$x_1$', fontsize=12)
plt.ylabel('$x_2$', fontsize=12)
plt.title('Comparação 1 dos Métodos de Otimização - Trajetórias', fontsize=14, fontweight='bold')
plt.legend()
plt.grid(True, alpha=0.3)
plt.show()

```



### COMPARACAO 1

```
ponto_inicial = (0,5)
n_passos = 100

grad_df_list(0.001, ponto_inicial, n_passos)
grad_f_mome_list(0.9, ponto_inicial, n_passos, 0.001)
grad_f_rmssp_list(ponto_inicial, n_passos, 0.001, 0.9, math.pow(10, -7))
ADAM_f_list(ponto_inicial, n_passos, 0.001, 0.9, 0.999, math.pow(10, -8))

ponto_inicial = (0,5)
n_passos = 2000
```

```

trajetoria_gd = grad_df_list(0.001, ponto_inicial, n_passos)
trajetoria_momentum = grad_f_mome_list(0.9, ponto_inicial, n_passos, 0.001)
trajetoria_rmsprop = grad_f_rmfp_list(ponto_inicial, n_passos, 0.001, 0.9, math.pow(10, -8))
trajetoria_adam = ADAM_f_list(ponto_inicial, n_passos, 0.001, 0.9, 0.999, math.pow(10, -8))

plt.style.use("ggplot")
plt.rc("axes", facecolor="#fafafa", grid=True)
plt.rc("grid", color="#f0f0f0")

plt.figure(figsize=(12, 10))
plt.contourf(X1, X2, Z, 30, cmap='viridis', alpha=0.7)
plt.colorbar()

gd_x = [p[0] for p in trajetoria_gd]
gd_y = [p[1] for p in trajetoria_gd]

momentum_x = [p[0] for p in trajetoria_momentum]
momentum_y = [p[1] for p in trajetoria_momentum]

rmsprop_x = [p[0] for p in trajetoria_rmsprop]
rmsprop_y = [p[1] for p in trajetoria_rmsprop]

adam_x = [p[0] for p in trajetoria_adam]
adam_y = [p[1] for p in trajetoria_adam]

plt.plot(gd_x, gd_y, 'ro-', markersize=3, linewidth=1.5, label='Gradiente Descendente', alpha=0.8)
plt.plot(momentum_x, momentum_y, 'bo-', markersize=3, linewidth=1.5, label='Momentum', alpha=0.8)
plt.plot(rmsprop_x, rmsprop_y, 'go-', markersize=3, linewidth=1.5, label='RMSProp', alpha=0.8)
plt.plot(adam_x, adam_y, 'mo-', markersize=3, linewidth=1.5, label='ADAM', alpha=0.8)

plt.plot(ponto_inicial[0], ponto_inicial[1], 'k*', markersize=15, label='Ponto Inicial')

plt.plot(gd_x[-1], gd_y[-1], 'rs', markersize=8, markeredgecolor='red', markerfacecolor='white')
plt.plot(momentum_x[-1], momentum_y[-1], 'bs', markersize=8, markeredgecolor='blue', markerfacecolor='white')
plt.plot(rmsprop_x[-1], rmsprop_y[-1], 'gs', markersize=8, markeredgecolor='green', markerfacecolor='white')
plt.plot(adam_x[-1], adam_y[-1], 'ms', markersize=8, markeredgecolor='magenta', markerfacecolor='white')

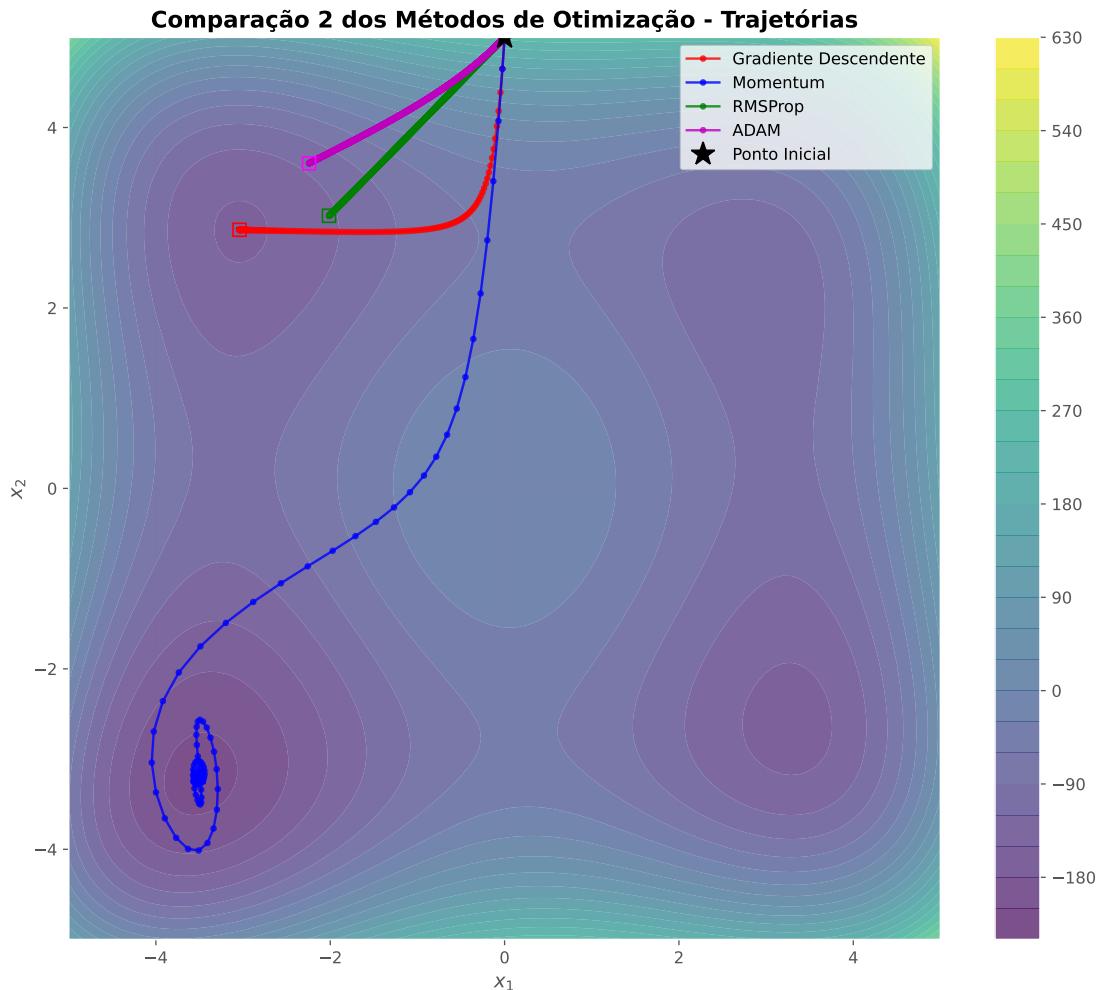
plt.xlabel('$x_1$', fontsize=12)

```

```

plt.ylabel('$x_2$', fontsize=12)
plt.title('Comparação 2 dos Métodos de Otimização - Trajetórias', fontsize=14, fontweight='bold')
plt.legend()
plt.grid(True, alpha=0.3)
plt.show()

```



## COMPARACAO 2

```

ponto_inicial = (0,5)
n_passos = 2000

grad_df_list(0.001, ponto_inicial, n_passos)
grad_f_mome_list(0.9, ponto_inicial, n_passos, 0.001)
grad_f_rmstp_list(ponto_inicial, n_passos, 0.001, 0.9, math.pow(10, -7))

```

```

ADAM_f_list(ponto_inicial, n_passos, 0.001, 0.9, 0.999, math.pow(10, -8))

ponto_inicial = (0,5)
n_passos = 2000

trajetoria_gd = grad_df_list(0.01, ponto_inicial, n_passos)
trajetoria_momentum = grad_f_mome_list(0.9, ponto_inicial, n_passos, 0.01)
trajetoria_rmsprop = grad_f_rmsp_list(ponto_inicial, n_passos, 0.01, 0.9, math.pow(10, -8))
trajetoria_adam = ADAM_f_list(ponto_inicial, n_passos, 0.01, 0.9, 0.999, math.pow(10, -8))

plt.style.use("ggplot")
plt.rc("axes", facecolor="#fafafa", grid=True)
plt.rc("grid", color="#f0f0f0")

plt.figure(figsize=(12, 10))
plt.contourf(X1, X2, Z, 30, cmap='viridis', alpha=0.7)
plt.colorbar()

gd_x = [p[0] for p in trajetoria_gd]
gd_y = [p[1] for p in trajetoria_gd]

momentum_x = [p[0] for p in trajetoria_momentum]
momentum_y = [p[1] for p in trajetoria_momentum]

rmsprop_x = [p[0] for p in trajetoria_rmsprop]
rmsprop_y = [p[1] for p in trajetoria_rmsprop]

adam_x = [p[0] for p in trajetoria_adam]
adam_y = [p[1] for p in trajetoria_adam]

plt.plot(gd_x, gd_y, 'ro-', markersize=3, linewidth=1.5, label='Gradiente Descendente', alpha=0.8)
plt.plot(momentum_x, momentum_y, 'bo-', markersize=3, linewidth=1.5, label='Momentum', alpha=0.8)
plt.plot(rmsprop_x, rmsprop_y, 'go-', markersize=3, linewidth=1.5, label='RMSProp', alpha=0.8)
plt.plot(adam_x, adam_y, 'mo-', markersize=3, linewidth=1.5, label='ADAM', alpha=0.8)

plt.plot(ponto_inicial[0], ponto_inicial[1], 'k*', markersize=15, label='Ponto Inicial')

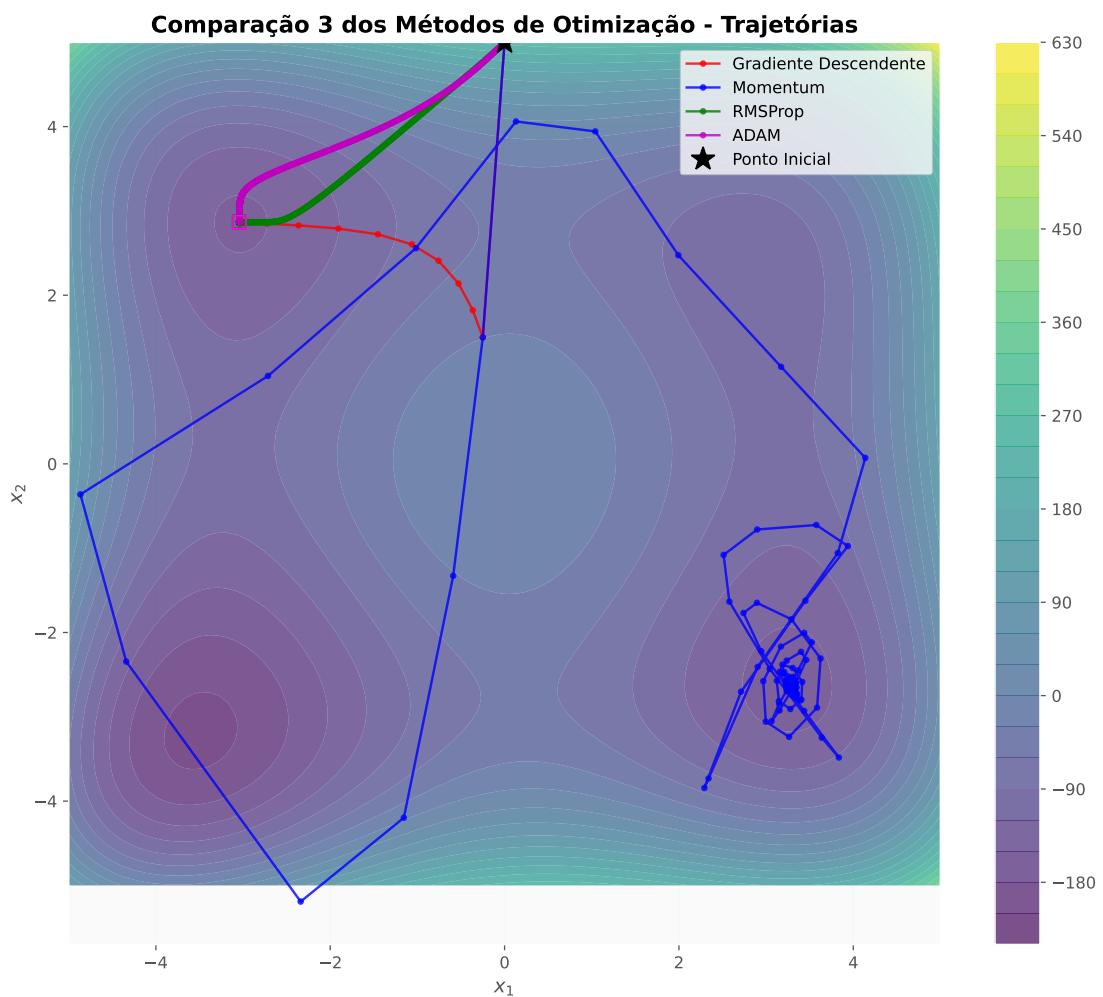
```

```

plt.plot(gd_x[-1], gd_y[-1], 'rs', markersize=8, markeredgewidth=2, markeredgecolor='red', markerfacecolor='red')
plt.plot(momentum_x[-1], momentum_y[-1], 'bs', markersize=8, markeredgewidth=2, markeredgecolor='blue', markerfacecolor='blue')
plt.plot(rmsprop_x[-1], rmsprop_y[-1], 'gs', markersize=8, markeredgewidth=2, markeredgecolor='green', markerfacecolor='green')
plt.plot(adam_x[-1], adam_y[-1], 'ms', markersize=8, markeredgewidth=2, markeredgecolor='magenta', markerfacecolor='magenta')

plt.xlabel('$x_1$', fontsize=12)
plt.ylabel('$x_2$', fontsize=12)
plt.title('Comparação 3 dos Métodos de Otimização - Trajetórias', fontsize=14, fontweight='bold')
plt.legend()
plt.grid(True, alpha=0.3)
plt.show()

```



COMPARACAO 3

```

ponto_inicial = (0,5)
n_passos = 2000

grad_df_list(0.01, ponto_inicial, n_passos)
grad_f_mome_list(0.9, ponto_inicial, n_passos, 0.01)
grad_f_rmsp_list(ponto_inicial, n_passos, 0.01, 0.9, math.pow(10, -7))
ADAM_f_list(ponto_inicial, n_passos, 0.01, 0.9, 0.999, math.pow(10, -8))

ponto_inicial = (0,5)
n_passos = 100

trajetoria_gd = grad_df_list(0.0001, ponto_inicial, 2000)
trajetoria_momentum = grad_f_mome_list(0.9, ponto_inicial, n_passos, 0.001)
trajetoria_rmsp = grad_f_rmsp_list(ponto_inicial, n_passos, 0.05, 0.9, math.pow(10, -7))
trajetoria_adam = ADAM_f_list(ponto_inicial, 85, 0.91, 0.9, 0.999, math.pow(10, -8))

plt.style.use("ggplot")
plt.rc("axes", facecolor="#fafafa", grid=True)
plt.rc("grid", color="#f0f0f0")

plt.figure(figsize=(12, 10))
plt.contourf(X1, X2, Z, 30, cmap='viridis', alpha=0.7)
plt.colorbar()

gd_x = [p[0] for p in trajetoria_gd]
gd_y = [p[1] for p in trajetoria_gd]

momentum_x = [p[0] for p in trajetoria_momentum]
momentum_y = [p[1] for p in trajetoria_momentum]

rmsprop_x = [p[0] for p in trajetoria_rmsp]
rmsprop_y = [p[1] for p in trajetoria_rmsp]

adam_x = [p[0] for p in trajetoria_adam]
adam_y = [p[1] for p in trajetoria_adam]

plt.plot(gd_x, gd_y, 'ro-', markersize=3, linewidth=1.5, label='Gradiente Descendente', alpha=0.7)
plt.plot(momentum_x, momentum_y, 'bo-', markersize=3, linewidth=1.5, label='Momentum', alpha=0.7)
plt.plot(rmsprop_x, rmsprop_y, 'go-', markersize=3, linewidth=1.5, label='RMSprop', alpha=0.7)
plt.plot(adam_x, adam_y, 'yo-', markersize=3, linewidth=1.5, label='Adam', alpha=0.7)

plt.xlabel('x')
plt.ylabel('y')
plt.title('Comparação de Algoritmos de Optimização')
plt.legend()
plt.show()

```

```

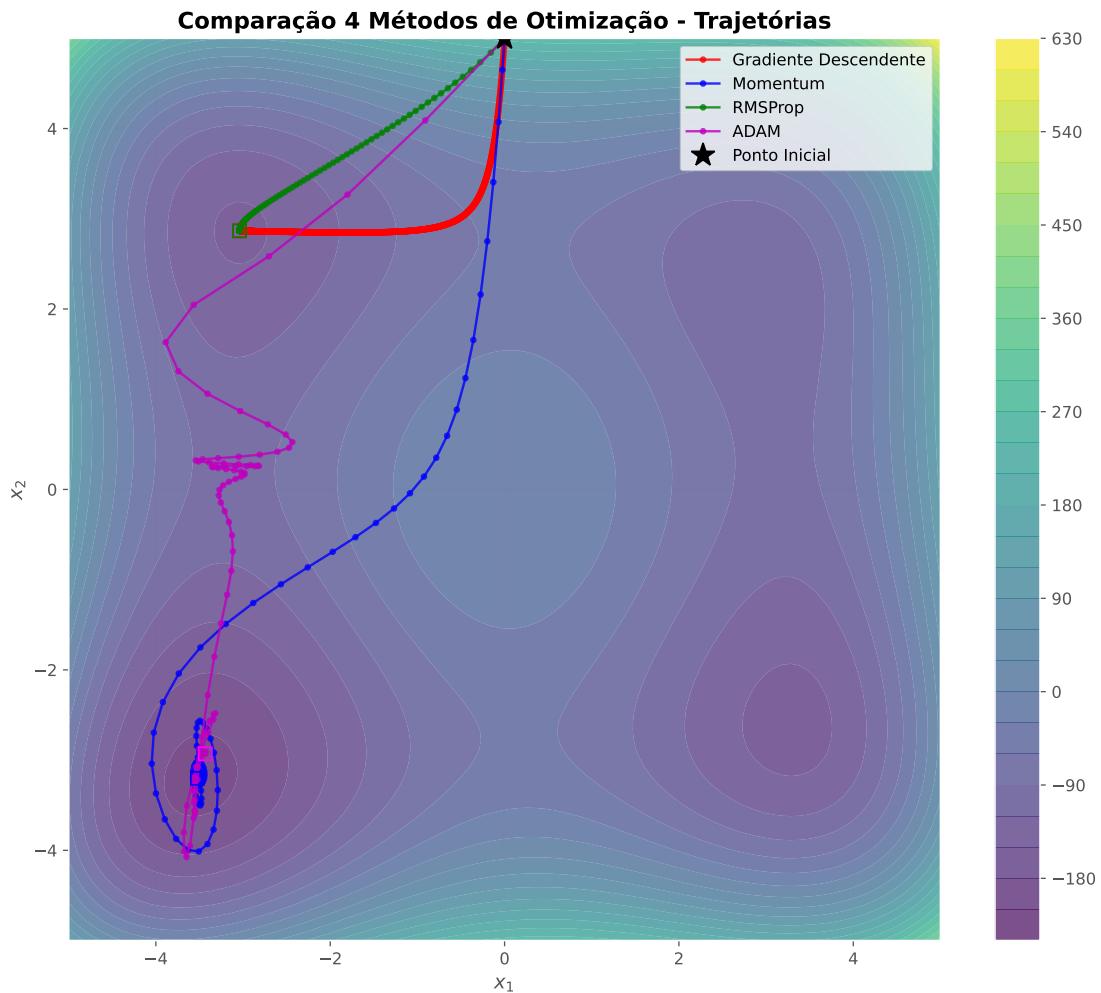
plt.plot(rmsprop_x, rmsprop_y, 'go-', markersize=3, linewidth=1.5, label='RMSProp', alpha=0.8)
plt.plot(adam_x, adam_y, 'mo-', markersize=3, linewidth=1.5, label='ADAM', alpha=0.8)

plt.plot(ponto_inicial[0], ponto_inicial[1], 'k*', markersize=15, label='Ponto Inicial')

plt.plot(gd_x[-1], gd_y[-1], 'rs', markersize=8, markeredgecolor='red', markerfacecolor='red')
plt.plot(momentum_x[-1], momentum_y[-1], 'bs', markersize=8, markeredgecolor='blue', markerfacecolor='blue')
plt.plot(rmsprop_x[-1], rmsprop_y[-1], 'gs', markersize=8, markeredgecolor='green', markerfacecolor='green')
plt.plot(adam_x[-1], adam_y[-1], 'ms', markersize=8, markeredgecolor='magenta', markerfacecolor='magenta')

plt.xlabel('$x_1$', fontsize=12)
plt.ylabel('$x_2$', fontsize=12)
plt.title('Comparação 4 Métodos de Otimização - Trajetórias', fontsize=14, fontweight='bold')
plt.legend()
plt.grid(True, alpha=0.3)
plt.show()

```



#### COMPARACAO 4

```
ponto_inicial = (0,5)
n_passos = 100

grad_df_list(0.0001, ponto_inicial, 2000)
grad_f_mome_list(0.9, ponto_inicial, n_passos, 0.001)
grad_f_rmsp_list(ponto_inicial, n_passos, 0.05, 0.9, math.pow(10, -7))
ADAM_f_list(ponto_inicial, 85, 0.91, 0.9, 0.999, math.pow(10, -8))
```

Na **COMPARACAO 1**, considerando apenas 100 passos com taxa de aprendizado 0.001 observamos que Adam e RMSProp não conseguiram se movimentar e o Gradiente Descendente chegou a um mínimo local.

Na **COMPARACAO 2**, aumentamos o número de passos para 2000 com taxa de aprendizado 0.001 e conseguimos com que Adam e RMSProp se aproximassesem de um mínimo (apesar de local).

Já na **COMPARACAO 3**, ao tentar aumentar a taxa para 0.01, ainda considerando 2000 passos, fizemos com que o Momentum desviasse do mínimo global (no qual ele estava finalizando anteriormente) e fosse parar num mínimo local. Após fazer várias e várias horas tentando o fazer o Adam chegar ao mínimo global partindo do ponto  $(0, 5)$  não achei uma combinação de parâmetros que me levasse ao objetivo. Depois comecei a pensar que as características do “terreno” da função estavam atrapalhando e que ele estava caindo num mínimo local em quase todos as taxas de aprendizagem. Eu tinha tentado valores maiores de taxa, mas não tão grandes. É possível notar que você tem que “pular” o mínimo local (com um passo relativamente grande) para evitar essa região superior esquerda. Tentei primeiro taxa igual a 1, e para minha surpresa, o mínimo global foi atingido considerando 100 passos (com o passo maior ele conseguiu explorar melhor o espaço inicialmente e a medida que foi chegando perto do mínimo, o tamanho do passo inicial não era relevante, ou seja, os pontos não ficariam interpolando). Tentei melhorar ao máximo essa marca, e cheguei nos seguintes parâmetros (**COMPARACAO 4**):

### **Adam**

1. ponto inicial =  $(0, 5)$
2.  $n = 85$
3. taxa de aprendizagem = 0.91
4.  $\rho_1 = 0.9$
5.  $\rho_2 = 0.999$
6.  $\delta = 10^{-8}$

Além disso, no caso abaixo gradiente simples com momento apresentou um resultado satisfatório chegando no mínimo global com uma taxa de aprendizado menor e considerando 100 passados

### **Gradiente com Momentum**

1. ponto inicial =  $(0, 5)$
2.  $n\_passos = 100$
3. taxa de aprendizagem = 0.001

Concluímos que, além do método utilizado, os parâmetros devem ser pensados com atenção considerando a característica de cada função. Nesse caso particular tivemos que “saltar” um mínimo local para chegar no global usando Adam e isso só foi possível após uma análise minuciosa acompanhada da curiosidade pela situação em si.