IPRJ/UERJ - INSTITUTO POLITÉCNICO DO ESTADO DO RIO DE JANEIRO

LUCAS RODRIGUES ESTORCK PINTO

202310349511

RELATÓRIO: Simplex de Nelder-Mead

NOVA FRIBURGO

2025

RELATÓRIO: Simplex de Nelder-Mead

Trabalho apresentado ao curso de Engenharia da Computação, na disciplina Métodos Numéricos de Otimização ,com o intuito de implementar e visualizar o método de Nelder-Mead.

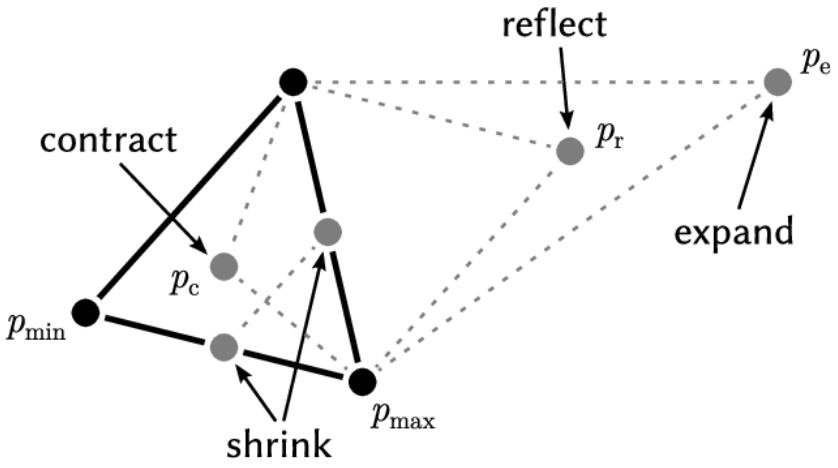
Professor responsável: Gustavo Barbosa Libotte

NOVA FRIBURGO
2025

0. SUMÁRIO

1. INTRODUÇÃO

O trabalho apresentado, visa implementar o método de Nelder-Mead, proposto por John Nelder e Roger Mead em 1965, no qual utiliza o conceito de simplexos para encontrar mínimos ou máximos de funções (neste trabalho foi implementada a minimização). Foi usada a linguagem de programação *Python* para implementar o método, linguagem na qual vem se destacando no meio acadêmico e que apresenta grandes facilitadores para implementação de métodos numéricos , através de bibliotecas como *Sympy* e *Matplotlib.*



representação gráfica de uma iteração no método de Nelder-Mead. Fonte: Wikipedia

2. O MÉTODO

Abaixo, tentarei explicar de maneira simples como funciona o algoritmo De Nelder-Mead, a fim de simplificar o método com uma linguagem mais acessível:

Começa com um "triângulo" (ou simplex): Em duas dimensões, pense num triângulo com 3 pontos (em 3D seria um tetraedro com 4 pontos, e assim por diante). Cada ponto é uma possível solução, e você calcula o valor da função em cada um deles. O ponto com o maior valor (o "pior") é o que está mais longe do mínimo.

Tenta melhorar o triângulo: O método olha para os pontos e faz movimentos para tentar achar um lugar melhor:

- **Reflexão**: Pega o pior ponto e "joga" ele para o outro lado do triângulo, como se fosse um espelho, para ver se o novo lugar é melhor.
- **Expansão**: Se o novo ponto for muito bom, ele tenta ir ainda mais longe na mesma direção, para ver se melhora mais.
- **Contração**: Se o novo ponto foi pior, ele dá um passo para trás, tentando um lugar mais perto do centro do triângulo.
- **Redução**: Se nada disso funcionar, o triângulo "encolhe" em direção ao melhor ponto, como se estivesse desistindo de explorar longe.

Repete até encontrar o melhor ponto: Ele continua fazendo esses movimentos — refletindo, expandindo, contraindo ou reduzindo — até que os pontos do triângulo fiquem muito próximos uns dos outros, ou seja, até que ele ache um ponto bem baixo (o mínimo local). Também pode parar se atingir um número máximo de tentativas, visando reduzir o custo computacional ou então loops em que o método não converge.

3. *IMPLEMENTAÇÃO*

Como de praste em algoritmos em python, primeiramente é realizada a implementação das bibliotecas a serem utilizadas:

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

As bibliotecas *Numpy* e *Matplotlib* são conhecidas, porém, o que pode gerar certa estranheza é *Axes3D* , que consiste basicamente em um módulo do *Matplotlib*, construído para exibição de gráficos 3D.

Após isso, foram definidas as funções objetivo, nas quais são:

$$\mathbb{R}^2 \rightarrow f(x_1, x_2) = (x_1 - 2)^4 + (x_1 - 2x_2)^2 \qquad (I)$$

Onde:

x é um vetor com componentes $x[0] = x_1$ e $x[1] = x_2$

Observe que a função possui mínimo em $(x_1, x_2) = (2, 1)$ o que resulta em $f(x_1, x_2) = 0$

$$\mathbb{R}^3 \rightarrow f(x_1, x_2, x_3) = x_1^2 + 2x_2^2 + 3x_3^2 \qquad (II)$$

Onde:

x é um vetor com componentes $x[0] = x_1$, $x[1] = x_2$ e $x[2] = x_3$

Observe que a função se trata de uma forma quadrática, então o mínimo está em $f(0,0,0) = 0$.

```
def funcao_r2_exemplo(x):
    """Função exemplo R²: (x1 - 2)^4 + (x1 - 2x2)^2"""
    return (x[0] - 2)**4 + (x[0] - 2 * x[1])**2
```

```
def funcao_r3_exemplo(x):
    """Função exemplo R³: x1² + 2x2² + 3x3²"""
    return x[0]**2 + 2 * x[1]**2 + 3 * x[2]**2
```

Após a definição das equações, foi criada uma função específica para a inicialização do *simplex*:

```
def inicializar_simplex(p0, passo):
    n = len(p0)
    simplex = [p0.copy()]
    for i in range(n):
        pi = p0.copy()
        pi[i] += passo
        simplex.append(pi)
    return np.array(simplex)
```

Acima, temos dois parâmetros:

- *p0:* ponto inicial do vetor de dimensão n ;
- *passo*: incremento para gerar os vértices adicionais.

Funcionamento:

- Determina n como a dimensão do espaço (tamanho de *p0*);
- Inicializa o simplex com o ponto *p0*;
- Para cada dimensão i, cria um novo vértice p_i adicionando *passo* à i-ésima coordenada de *p0*, mantendo as demais coordenadas inalteradas;
- Converte a lista de vértices em um array *Numpy* para facilitar operações matriciais.

Determinada a inicialização, podemos implementar o método de *Nelder-Mead:*

```
def otimizar_nelder_mead(f, p0, passo, tolerancia=1e-6, max_iter=1000,
verbose=True):
    n = len(p0)
    simplex = inicializar_simplex(p0, passo)
    historico = []

    for iteracao in range(max_iter):
        valores = np.array([f(x) for x in simplex])
        ordem = np.argsort(valores)
        simplex = simplex[ordem]
        valores = valores[ordem]
        historico.append(simplex.copy())

        if valores[-1] - valores[0] < tolerancia:
            if verbose:
                print(f"Convergiu após {iteracao} iterações")
            return simplex[0], valores[0], iteracao, historico

        centroide = np.mean(simplex[:-1], axis=0)
        refletido = centroide + 1.0 * (centroide - simplex[-1])
        f_refletido = f(refletido)

        if f_refletido < valores[0]:
            expandido = centroide + 2.0 * (refletido - centroide)
            f_expandido = f(expandido)
            simplex[-1] = expandido if f_expandido < f_refletido else
refletido
        elif f_refletido < valores[-2]:
            simplex[-1] = refletido
        else:
            contraido = centroide + 0.5 * (simplex[-1] - centroide)
            f_contraido = f(contraido)
            if f_contraido < valores[-1]:
                simplex[-1] = contraido
            else:
                for i in range(1, n+1):
```

```
                    simplex[i] = simplex[0] + 0.5 * (simplex[i] -
simplex[0])

    if verbose:
        print("Número máximo de iterações atingido")
    return simplex[0], valores[0], max_iter, historico
```

Acima, a função recebe como entrada a função objetivo f, um ponto inicial , um escalar *passo* para construir o simplex inicial, uma tolerância de convergência (*tolerancia* = 10^{-6}), um número máximo de iterações (*max_iter* = *1000*) e um booleano *verbose* que é usado para controlar mensagens de progresso, para fins de debug.

Primeiramente, a função determina a dimensão n do espaço com base no tamanho de p_0 , cria um simplex inicial com $n + 1$ vértices e inicializa uma lista para armazenar a evolução do simplex. Em seguida, entra em um loop que avalia f em todos os vértices do simplex , ordenando os vértices de menor valor ao maior valor, em resumo, ordenando os melhores vértices até os piores vértices. Após isso, as funções fazem as avaliações necessárias para continuar ou não o loop.

Se não houver convergência, é calculado o centróide dos n melhores vértices e tenta melhorar o simplex por meio de operações geométricas: primeiro, realiza uma reflexão do pior vértice através do centróide com coeficiente $\alpha = 1.0$, gerando um ponto refletido x_r. Se $f(x_r)$ for menor que o melhor valor atual, tenta uma expansão com coeficiente $\gamma = 2.0$, gerando um ponto expandido x_e, e escolhe entre x_e e x_r com base em qual resulta em menor valor de f. Se $f(x_r)$ for melhor que o segundo pior valor, mas não o melhor, aceita x_r. Caso contrário, realiza uma contração com coeficiente $\rho = 0.5$, gerando um ponto contraído x_c. Se $f(x_c)$ for melhor que o valor do pior vértice, usa x_c, senão aplica uma redução, encolhendo todos os vértices (exceto o melhor) em direção ao melhor com coeficiente $\sigma = 0.5$.

```
def executar_multiplas_otimizacoes(f, dim, num_execucoes, min_valor,
max_valor, passo, tolerancia, max_iter):
    melhor_valor = float('inf')
    melhor_ponto = None
    for i in range(num_execucoes):
        p0 = np.random.uniform(min_valor, max_valor, dim)
        ponto, valor, iters, _ = otimizar_nelder_mead(f, p0, passo,
tolerancia, max_iter, verbose=False)
        print(f"Execução {i+1}: Valor = {valor:.2e}, Ponto = {ponto},
Iterações = {iters}")
        if valor < melhor_valor:
            melhor_valor = valor
            melhor_ponto = ponto
    return melhor_ponto, melhor_valor
```

A função acima, como explicitado, executa várias vezes o algoritmo de Nelder-Mead, com o intuito de mitigar os riscos do método encontrar um mínimo local , e não um mínimo global.

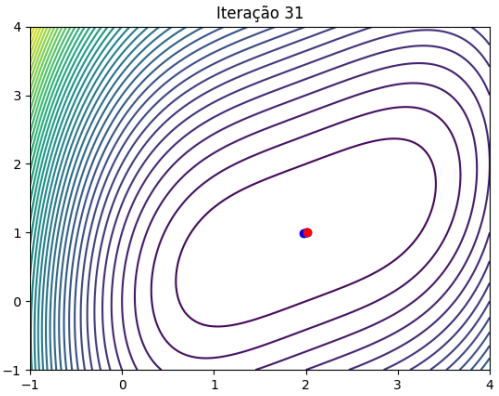
As demais funções do código, dispensam explicação por conta de sua simplicidade, se tratando somente da exibição dos gráficos e dos resultados finais.

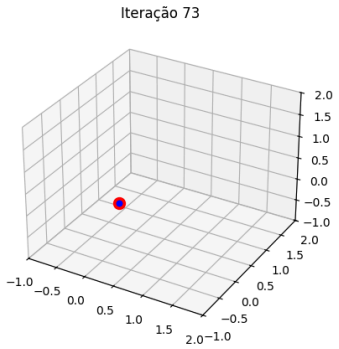
4. RESULTADOS

Após a explicação detalhada das principais funções do algoritmo, nada mais prudente do que demonstrar os resultados do algoritmo usando como base as funções (I) e (II).

```
--- Otimização em R² ---
Execução 1: Valor = 1.29e-07, Ponto = [2.0037735  1.00170742],
Iterações = 30
Execução 2: Valor = 1.03e-06, Ponto = [1.98391142 0.99244602],
Iterações = 34
Execução 3: Valor = 2.01e-07, Ponto = [1.99997485 1.00021131],
Iterações = 26
Execução 4: Valor = 2.11e-07, Ponto = [2.00931873 1.00488469],
Iterações = 34
erações = 30
Melhor ponto R²: [2.01244126 1.00610063], valor: 8.16e-08
```

```
--- Otimização em R³ ---
Convergiu após 73 iterações
Melhor ponto R³: [-0.00044627 -0.00022544 -0.00059583], valor:
1.37e-06, iterações: 73
```





As imagens acima remetem ao output do algoritmo realizado para esse trabalho, note que os resultados estão muito próximos dos “mínimos reais” , o que mostra que o algoritmo além de chegar na tolerância especificada, ele está tendendo ao mínimo. Observe também que a visualização gráfica por este documento fica muito limitada a imagem, porém , na execução do programa, será possível observar a evolução do simplex em tempo real até a convergência.

5. *CONCLUSÃO*

Por fim, pode-se concluir que a implementação do programa foi bem sucedida, apesar da dificuldade de implementação, por conta da complexidade do algoritmo, tudo correu como esperado. Para fins de curiosidade, a princípio considerei modelar o algoritmo em JULIA, porém por conta do prazo e também curva de aprendizado, foi considerado seguir com a implementação em python, por via de consulta, deixarei aqui o link de meu repositório para futura consulta da implementação do algoritmo em JULIA.

Repositório

6. *FONTES BIBLIOGRÁFICAS E CONSULTAS*

[1] Nelder-Mead method - wikipedia - https://en.wikipedia.org/wiki/Nelder%E2%80%93Mead_method - Acesso em 29/04/2025

[2] How to use Nelder-Mead optimization in python - machine learning mastery - https://machinelearningmastery.com/how-to-use-nelder-mead-optimization-in-python/ - Acesso em 29/04/2025

[3] Nelder-Mead optimization example with python - YouTube - https://www.youtube.com/watch?v=MDE9lkDpA-s - Acesso em 30/04/2025

[4] Nelder Mead Method plus Python code explanation - YouTube - https://www.youtube.com/watch?v=WeONP46FD3g - Acesso em 30/04/2025

[5] nelder-mead - GitHub - https://github.com/fchollet/nelder-mead/blob/master/nelder_mead.py - Acesso em 30/04/2025

[6] Implementing Nelder Mead Using Python - Medium - https://medium.com/@budisantosapurwokartiko/implementing-nelder-mead-uisng-python-94ec60d17e37 - Acesso em 01/05/2025

[7] Sympy Docs - https://docs.sympy.org/latest/index.html - Acesso em 01/05/2025

[8] Matplotlib - https://matplotlib.org/ - Acesso em 01/05/2025

[9] Curso Em Video Python Mundo 3 - YouTube - https://www.youtube.com/watch?v=0LB3FSfjvao&list=PLHz_AreHm4dksnH2jVTIVNviIMBVYyFnH - Acesso em 01/05/2025

[10] Material de aula - Aula 5

[11] Repositório institucional da UTFPR - https://repositorio.utfpr.edu.br/jspui/bitstream/1/25760/1/otimizacaosimplexneldermead.pdf - Acessado em 01/05/2025