Departamento de Engenharia Elétrica
Universidade Federal do Paraná

Modelo Matemático com Otimização Não Linear e Números Complexos

André Corso Pozzan

Atividade **5** – Iniciação Científica **GICS**

Orientador: Prof. Ph.D. Eduardo Gonçalves de Lima

UFPR – Centro Politécnico – Curitiba, PR
3 de maio de 2025

Conteúdo

1 Atividade 5

Nesta atividade, repetiremos o procedimento da Atividade 4, contudo a partir da resolução proposta na Atividade 3 (em vez de partir da Atividade 2). O objetivo é aplicar otimização não linear para identificar os coeficientes do polinômio de memória (MP) que possui números complexos.

1.1 Etapa A: Construção de uma Função

A função a ser desenvolvida deve atender aos seguintes requisitos:

- **Argumentos (incógnitas) da função:** devem ser valores reais que representem o módulo de cada componente do vetor de erro resultante do MP, ou (alternativa) os valores reais das partes real e imaginária de cada coeficiente.
- **Processamentos internos dentro da função:**
  - Carregar os sinais de treinamento (entrada e saída) utilizados na Atividade 3;
  - Processar o sinal de entrada através do polinômio de memória, permitindo que os coeficientes (passados como argumentos) sejam complexos se construídos internamente;
  - Calcular a saída estimada pelo MP e obter o vetor de erro entre a saída desejada e a saída estimada;
  - Retornar o *módulo* de cada componente do vetor de erro (ou o vetor de erro complexo, caso a otimização suporte tal forma de entrada).
- **Retorno da função:** vetor de números reais correspondentes ao módulo do erro em cada amostra.

1.2 Etapa B: Otimização Não Linear

Para encontrar os valores de todos os coeficientes do MP, utilizaremos otimização não linear. No MATLAB, empregaremos o comando `lsqnonlin` (Optimization Toolbox). Em Python, recomenda-se utilizar `scipy.optimize.least_squares` ou equivalente.

1.2.1 Formas de Passagem de Argumentos

1. **Passagem de valores reais para partes real e imaginária (garantido funcionamento):** para C coeficientes complexos, passar 2C valores reais como argumentos. Dentro da função, montar cada coeficiente como $a_j + i\,b_j$.
2. **Passagem direta de números complexos (não garantido):** passar C argumentos complexos; depende do suporte do otimizador para variáveis complexas.

1.2.2 Orientações de Implementação

- Utilize a função construída na Etapa A como entrada para `lsqnonlin` (ou `least_squares`);
- Teste diferentes estimativas iniciais para os coeficientes:
  - Todos os coeficientes iniciando em zero;
  - Todos os coeficientes iniciando em um;
  - Valores aleatórios no intervalo adequado.
- Ao realizar operações de transposição em vetores/matrizes contendo números complexos, utilize o operador conjugar-transposto (MATLAB: `.'` ou `transpose`; Python: `.T` se aplicável);
- Configure opções de exibição para acompanhar a evolução do resíduo (MSE) por iteração (MATLAB: `optimoptions('lsqnonlin','Display','iter')`; Python: `verbose=2`).

2 Resolução

Inicialmente, ao incorporar a equação da Atividade 3 no código da Atividade 4, tentei passar diretamente os valores complexos como entrada para a função. No entanto, essa abordagem não produziu resultados satisfatórios: o método frequentemente não convergia, e os valores estimados estavam significativamente fora do esperado.

Dessa forma, ao adotar a estratégia de utilizar `2C` coeficientes reais no vetor de entrada para a função de minimização, os resultados obtidos passaram a atender às expectativas, apresentando maior estabilidade e precisão.

$$\tilde{y}(n) = \sum_{p=1}^{P} \sum_{m=0}^{M} \tilde{b}_{2p-1,m} \, |\tilde{x}(n-m)|^{2p-2} \, \tilde{x}(n-m) \tag{1}$$

2.1 Código em python

O código da atividade 4 foi modificado agora para trabalhar com os números complexos da equação da atividade 3 e utilizando os dados contidos em `in_out_SBRT2_direto.mat` para a análise.

2.1.1 Adaptações para os números complexos

Algumas alterações também foram feitas para adequar o código, como esta de iniciar os números estimados com complexos:

```
initial_complex = np.zeros(P*(M+1)) + 1j * np.zeros(P*(M+1))
```

"Quebrar"o vetor de entradas em complexos para metade sendo a parte real e a outra metade a parte imaginária:

```
initial_real = np.concatenate((initial_complex.real, initial_complex.imag))
```

A função `unpackComplexCoefficients` faz o papel de montar novamente os valores complexos que estavam descritos em um vetor de números reais para que a função de resíduo consiga calcular corretamente a estimativa e retornar para a otimização:

```
def unpackComplexCoefficients(x_real, P, M):
    # Metade do vetor são os componentes reais e a outra metade são os imaginários
    real_parts = x_real[:len(x_real)//2]
    imag_parts = x_real[len(x_real)//2:]

    complex_coef = real_parts + 1j * imag_parts
    return complex_coef.reshape((P, M+1))
```

Função que contém os parâmetros da equação (1) para realizar os cálculos de estimativa e também ser chamada na função de resíduos:

```
    def estimatedValueWithComplex(x_data, coef_matrix, P, M):
    y_est = []
```

```
    for n in range(len(x_data)):
        estimated_value = 0.0 + 0.0j
        for p in range(1, P + 1):
            for m in range(M + 1):
                dataIndex = max(0, n - m)
                power = 2*p - 2

                term = (np.abs(x_data[dataIndex])**power) * x_data[dataIndex]
                estimated_value += term * coef_matrix[p-1, m]
        y_est.append(estimated_value)

    return np.array(y_est)
```

Estas foram as alterações mais expressivas, também foram utilizados os passos desenvolvidos na atividade anterior como a visualização do maior/menor erro e o gráfico 3D.

2.1.2 Gráfico 3D, conjunto P,M

Para o gráfico 3D das combinações de P e M para encontrar o melhor erro, algumas alterações foram necessárias por conta de que o processamento se tornou mais pesado. Desse modo, a função de otimização do Python `least_squares` só utiliza um núcleo do processador para cada conjunto P,M. Assim, utilizando a biblioteca `numba` o código distribui várias combinações P,M para todos os núcleos do processador simultaneamente. Como resultado, o tempo de retorno é melhorado significativamente.

Também foi feita uma função para estimar melhor os valores iniciais e reduzir o tempo de convergência do método:

```
def generateInitialComplex(P, M, mean, std):
    shape = P * (M + 1)
    real_part = np.random.normal(loc=mean.real, scale=std, size=shape)
    imag_part = np.random.normal(loc=mean.imag, scale=std, size=shape)
    return real_part + 1j * imag_part
```

2.1.3 Código completo

```
from scipy.io import loadmat
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import least_squares

loaded_data = loadmat('in_out_SBRT2_direto.mat')
in_training = loaded_data['in_extraction']
out_training = loaded_data['out_extraction']

in_validation = loaded_data['in_validation']
out_validation = loaded_data['out_validation']


#Definições dos parâmetros do modelo
# P - Ordem do polinômio
# M - Profundidade de memória
P, M = 3, 2

# Coeficientes iniciais complexos (ex: aleatórios)
```

```python
initial_complex = np.zeros(P*(M+1)) + 1j * np.zeros(P*(M+1))
# initial_complex = np.ones(P*(M+1)) + 1j * np.ones(P*(M+1))
# initial_complex = np.random.randn(P*(M+1)) + 1j * np.random.randn(P*(M+1))

# Vetor real concatenando parte real e imaginária
initial_real = np.concatenate((initial_complex.real, initial_complex.imag))

print(f"P: {P}, M: {M}  \ninitial_estimated: ", np.array(initial_real), "\n")

def estimatedValueWithComplex(x_data, coef_matrix, P, M):
    y_est = []

    for n in range(len(x_data)):
        estimated_value = 0.0 + 0.0j
        for p in range(1, P + 1):
            for m in range(M + 1):
                dataIndex = max(0, n - m)
                power = 2*p - 2

                term = (np.abs(x_data[dataIndex])**power) * x_data[dataIndex]
                estimated_value += term * coef_matrix[p-1, m]
        y_est.append(estimated_value)

    return np.array(y_est)

def unpackComplexCoefficients(x_real, P, M):
    # Metade do vetor são os componentes reais e a outra metade são os imaginários
    real_parts = x_real[:len(x_real)//2]
    imag_parts = x_real[len(x_real)//2:]

    complex_coef = real_parts + 1j * imag_parts
    return complex_coef.reshape((P, M+1))


def mpResiduals(x_real, x_data, y_data, P, M):
    coef_matrix = unpackComplexCoefficients(x_real, P, M)

    y_est = estimatedValueWithComplex(x_data, coef_matrix, P, M)

    resid = y_data.ravel() - y_est

    return np.concatenate((resid.real, resid.imag))

def calcCoefByLeastSquares(in_data, out_data):
    result = least_squares(mpResiduals, initial_real, args=(in_data.ravel(), out_data.ravel(), P, M),verbose=1)
    final_coef = unpackComplexCoefficients(result.x, P, M)

    print("\nResultado da otimização:", final_coef)

    return final_coef

def calcVectorError(out_estimated, out_data):
    vector_error = []
    for i in range(len(out_data)):
        error = out_estimated[i] - out_data[i]
        vector_error.append(error)

    return np.abs(vector_error)

def checkError(vector_error, in_data, out_data, out_estimated_data):
    """Exibe métricas de erro e retorna os pontos de maior e menor erro, tanto real quanto estimado."""

    mean_error = np.mean(vector_error)
    max_abs_error = np.max(np.abs(vector_error))
    min_abs_error = np.min(np.abs(vector_error))

    print("\nErro médio:", mean_error)
```

```
    print("Erro máximo em módulo:", max_abs_error)
    print("Erro mínimo em módulo:", min_abs_error)

    # Índices dos maiores e menores erros em módulo
    max_idx = np.argmax(np.abs(vector_error))
    min_idx = np.argmin(np.abs(vector_error))

    # Pontos de erro máximo
    max_error_point_original = (in_data[max_idx], out_data[max_idx])
    max_error_point_estimated = (in_data[max_idx], out_estimated_data[max_idx])

    # Pontos de erro mínimo
    min_error_point_original = (in_data[min_idx], out_data[min_idx])
    min_error_point_estimated = (in_data[min_idx], out_estimated_data[min_idx])

    return (
        max_error_point_original,
        max_error_point_estimated,
        min_error_point_original,
        min_error_point_estimated
    )

def plotErrorPair(p1, p2, color, label_base, marker1="x", marker2="o", zorder=5):
    x1, y1 = p1[0].item(), p1[1].item()
    x2, y2 = p2[0].item(), p2[1].item()

    plt.scatter(x1, y1, color=color, label=f'{label_base} Original', marker=marker1, s=150, zorder=zorder)
    plt.scatter(x2, y2, color=color, label=f'{label_base} Estimado', marker=marker2, s=80, zorder=zorder)

    plt.plot([x1, x2], [y1, y2], color=color, linestyle='--', linewidth=1.5, label=f'Linha de {label_base}',
    ↪  zorder=zorder-1)


coef = calcCoefByLeastSquares(in_training, out_training)

out_estimated = estimatedValueWithComplex(in_validation.ravel(), coef, P, M)

vector_error = calcVectorError(out_estimated, out_validation)

max_error_point_original, max_error_point_estimated, min_error_point_original, min_error_point_estimated =
↪  checkError(vector_error, in_validation, out_validation, out_estimated)

plotErrorPair(max_error_point_original, max_error_point_estimated, color='green', label_base='Erro Máximo')
plotErrorPair(min_error_point_original, min_error_point_estimated, color='red', label_base='Erro Mínimo')

plt.scatter(in_validation, out_validation, label='Dados Originais', color='blue')
plt.scatter(in_validation, out_estimated, color='orange', label='Ajuste',alpha=0.8)
plt.xlabel('in_validation')
plt.ylabel('out_validation')
plt.title('Dados Originais e Estimados com Erros')
plt.legend()
plt.grid()
plt.show()
```

O código completo está disponível no seguinte endereço do GitHub:

`script-5.py` — Repositório no GitHub

`3D-model.py` — Repositório no GitHub

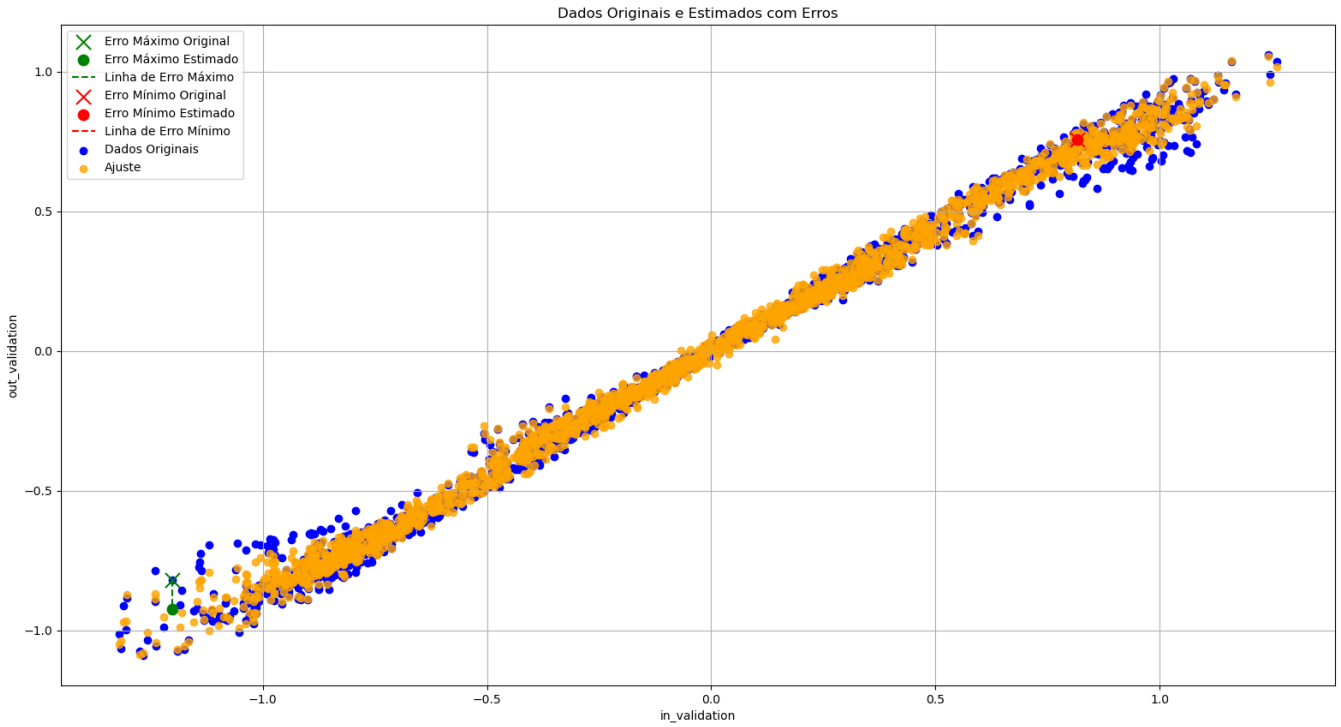
Repositório no GitHub com todos os códigos feitos na iniciação científica:

github.com/andrepozzan/ic-gics

3 Resultados

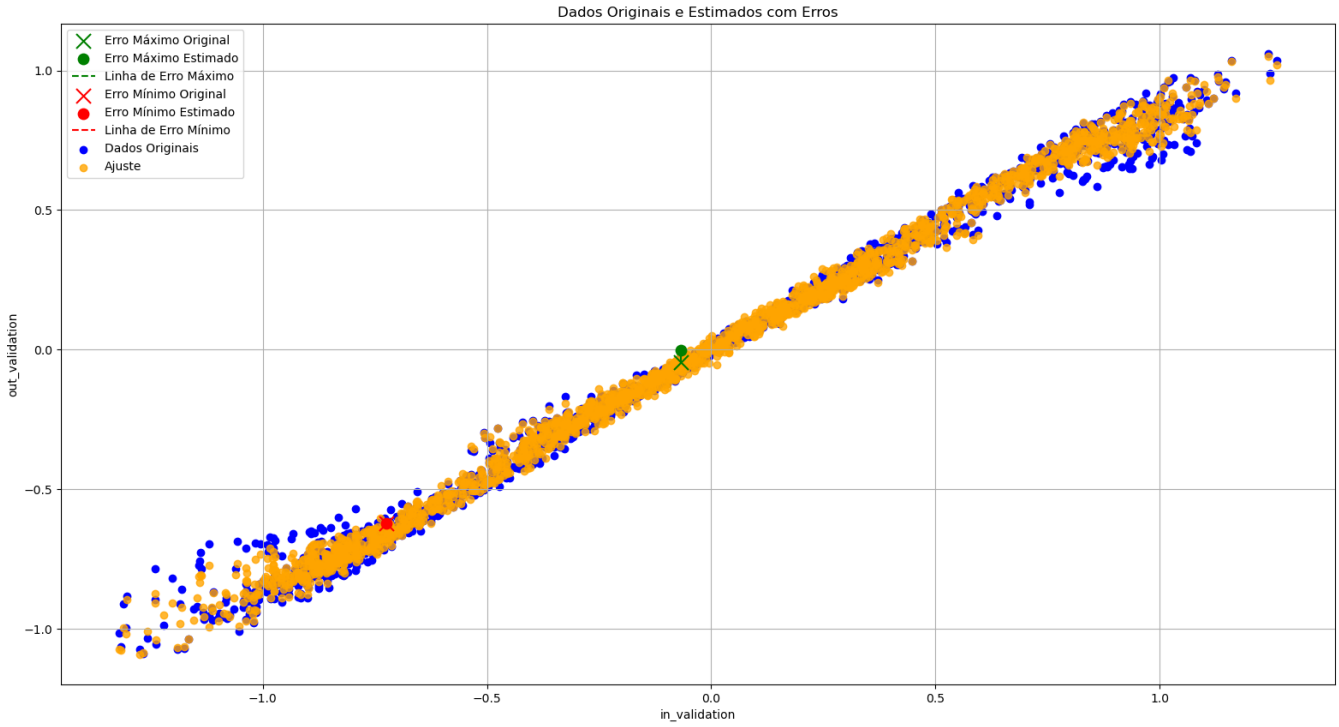
Nesse sentido, os gráficos produzidos pelos códigos apresentados são:

3.1 P=3, M=2




```
P: 3, M: 2
initial_estimated:  [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]

   Iteration     Total nfev        Cost      Cost reduction    Step norm     Optimality
       0              1         8.8318e+02                                    2.03e+03
       1              2         2.4693e+00      8.81e+02       1.00e+00       2.26e+00
       2              3         1.9424e+00      5.27e-01       1.09e+00       2.72e-07
       3              4         1.9424e+00      4.44e-16       6.45e-08       1.91e-08
`ftol` termination condition is satisfied.
Function evaluations 4, initial cost 8.8318e+02, final cost 1.9424e+00, first-order optimality 1.91e-08.

Resultado da otimização: [[ 1.3881898 +0.23386603j -0.80520357-0.34159084j  0.26636217+0.09909635j]
 [ 0.01823281-0.04889777j -0.28509185-0.00041084j  0.39025178+0.05731193j]
 [-0.11365363+0.0190743j   0.16636963+0.02282542j -0.13372659-0.02721599j]]

Erro médio: 0.027743779427684814
Erro máximo em módulo: 0.10791023961430596
Erro mínimo em módulo: 0.00013503308669843734
```

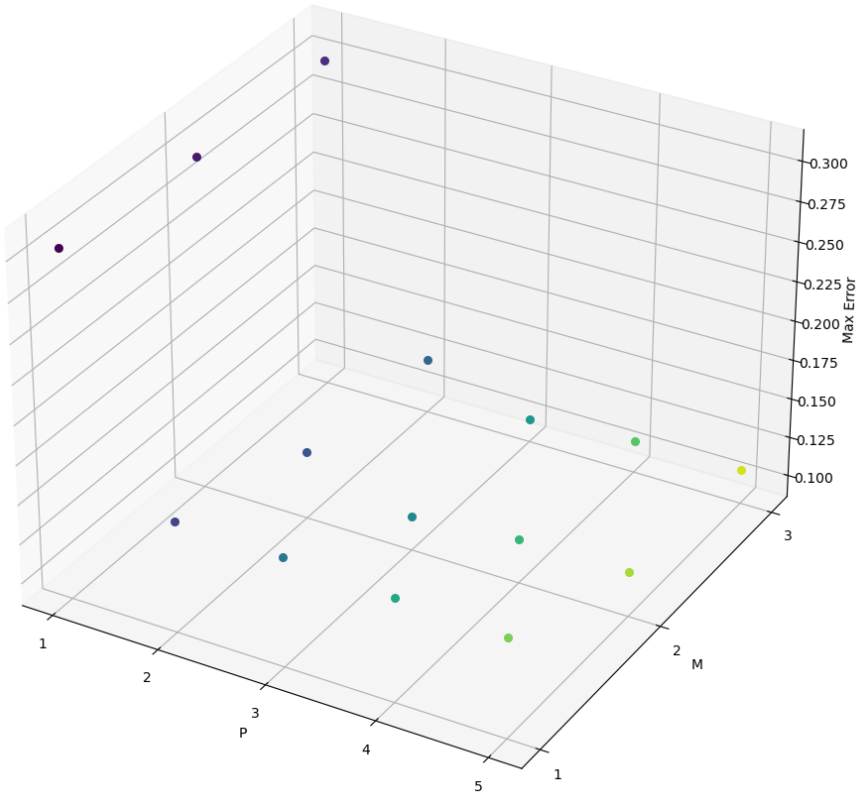
3.2 P=5, M = 3

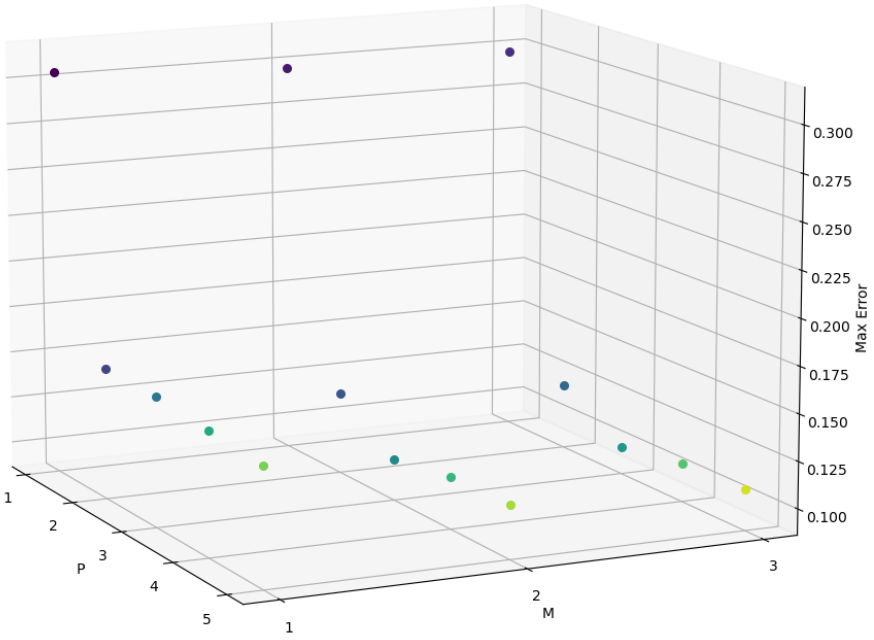



```
P: 5, M: 3
initial_estimated:  [ 4.20311689e-02 -1.16634983e+00 -8.64828527e-01 -4.50722999e-01
 -3.59842515e-04  5.10921196e-01  5.05405894e-01  1.02429279e+00
 -6.08385641e-01 -1.58753551e+00 -1.30134562e+00 -5.71861716e-01
  9.94360929e-01  1.57443334e+00  8.96612702e-01  1.11300582e+00
  1.35050601e+00  5.67717085e-01 -1.60727328e+00  4.08265539e-01
 -2.95259731e-01 -2.60258910e-01  1.19007530e+00 -1.20236186e+00
  8.89572164e-01  1.12505304e+00 -7.15433762e-01 -4.96304193e-01
 -3.35790353e-01 -6.23247149e-01  1.70590196e+00 -3.20031311e-01
  6.31961169e-01  5.72035686e-01  6.11755585e-01  1.19482376e-01
 -1.66327955e-02 -1.46499045e-01 -1.35471303e+00  3.23815228e-02]

   Iteration     Total nfev        Cost      Cost reduction    Step norm     Optimality
       0              1         2.8206e+04                                    2.50e+04
       1              2         1.7978e+00      2.82e+04       5.54e+00       6.33e-04
       2              3         1.7978e+00      8.78e-11       1.87e-05       2.57e-08
`ftol` termination condition is satisfied.
Function evaluations 3, initial cost 2.8206e+04, final cost 1.7978e+00, first-order optimality 2.57e-08.

Resultado da otimização: [[ 1.03188634+0.21680646j  0.05961678-0.29756912j -0.64380917+0.04906743j
   0.28251039+0.0056361j ]
 [ 0.20381247-0.02739972j -0.08882071-0.02370251j  0.34074469+0.11716144j
   0.15272078+0.03515654j]
 [-0.24905736-0.03172511j -0.26359009+0.07571063j  0.00928009-0.14082046j
  -0.19421859-0.04670955j]
 [ 0.01193243+0.03890497j  0.27291553-0.03166394j -0.07059911+0.07066077j
   0.08281823+0.0280658j ]
 [ 0.01686846-0.00925963j -0.06717475+0.00461014j  0.0201522 -0.01318752j
  -0.01506202-0.00641234j]]
Erro médio: 0.02732698825727856
Erro máximo em módulo: 0.10810327293910689
Erro mínimo em módulo: 0.0008754589893070488
```

3.3 3D, P=5, M = 3

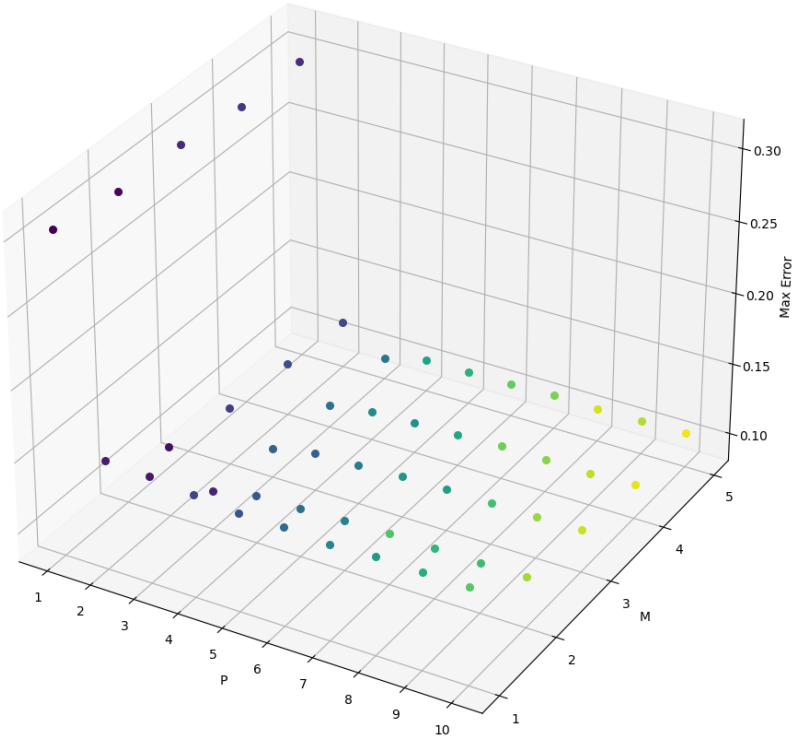
Agora o gráfico 3D tem os pontos com a coloração conforme seu valor:

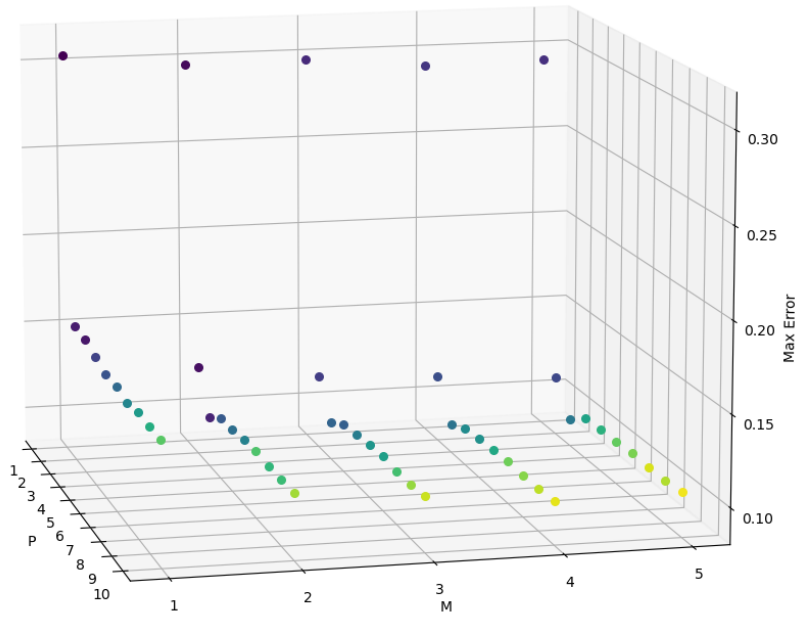


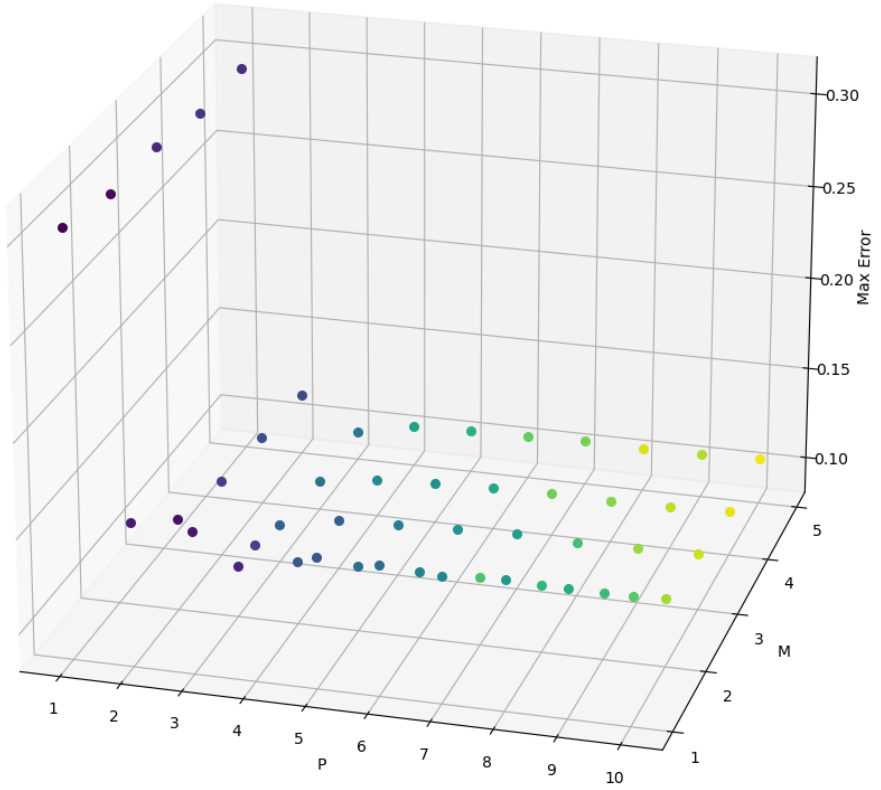


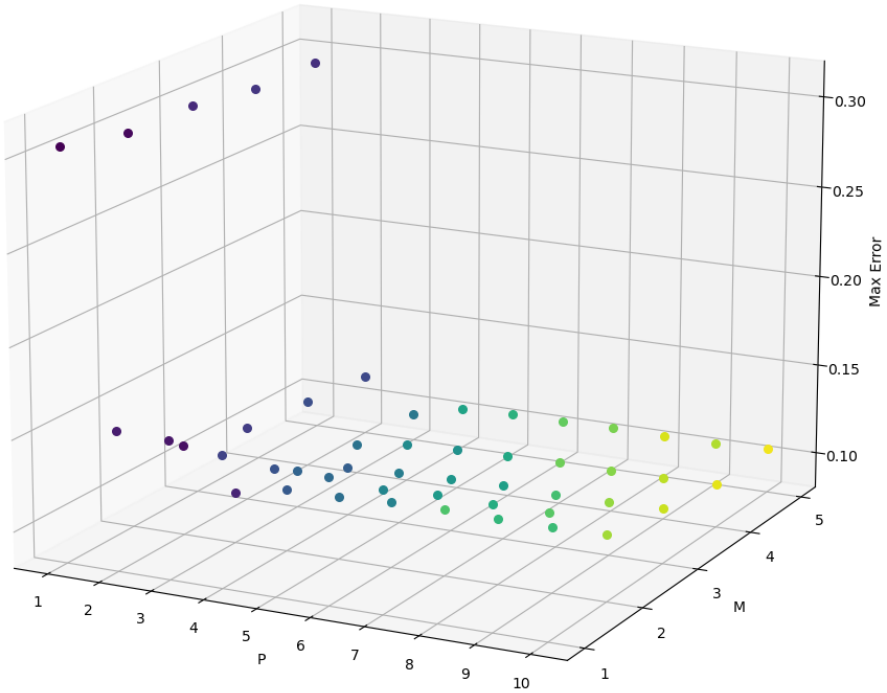
3.4 3D, P=10, M = 5

Neste contexto, os resultados observados para a combinação de P,M foram semelhantes ao da atividade 4 onde a ordem polinomial P ganha uma quantidade relevante de precisão e é impulsionada pela memória.



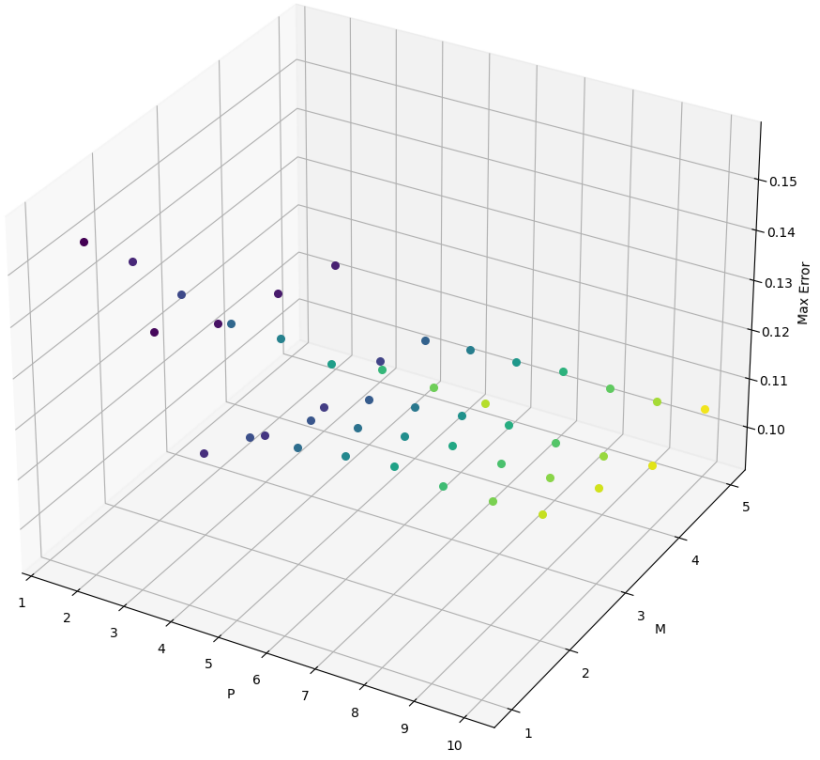


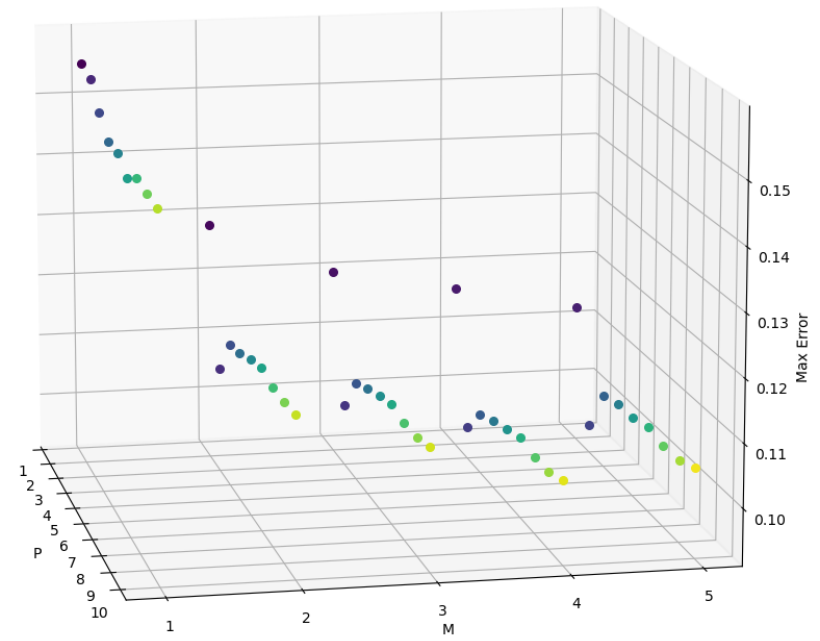

0.30
0.25
0.20
0.15
0.10
Max Error
1
2
3
4
5
6
7
8
9
10
P
1
2
3
4
5
M

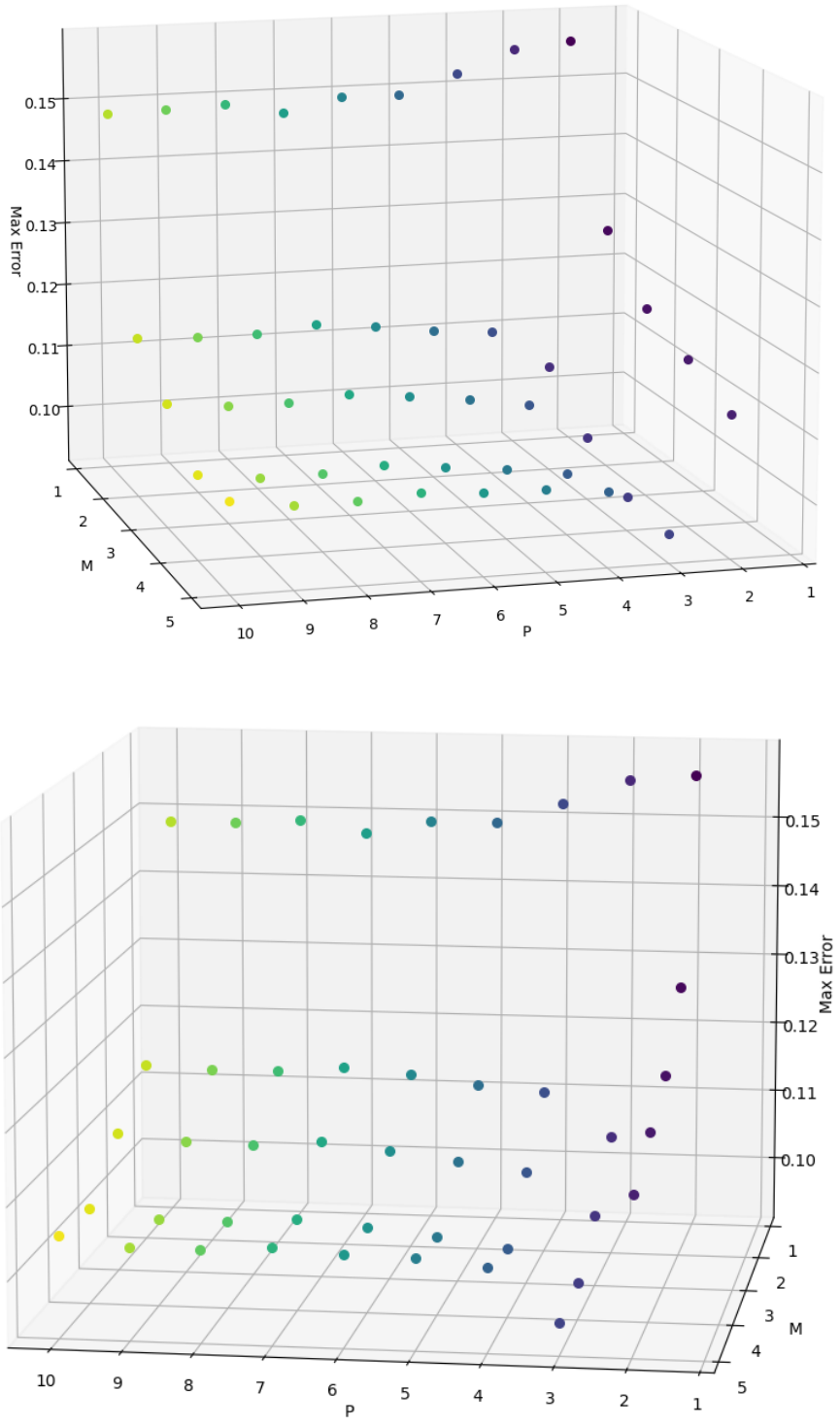

0.30
0.25
0.20
0.15
0.10
Max Error
1
2
3
4
5
6
7
8
9
10
P
1
2
3
4
5
M

3.5 3D, P=10, M = 5, desconsiderando P=1

Com base na seção anterior, observou-se que a escolha de P = 1 comprometia a escala do gráfico, dificultando a representação adequada dos valores subsequentes. Além disso, na maioria dos casos, a precisão obtida com P = 1 mostrou-se insuficiente para uma análise relevante.







Portanto, a combinação que apresentou o melhor desempenho nessas condições foi P = 3 e M = 5. Observa-se que o aumento no grau do polinômio, além desse ponto, resultou apenas em maior complexidade computacional, sem ganhos em precisão.

Referências

LIMA, Eduardo Gonçalves de. **Behavioral modeling and digital base-band predistortion of RF power amplifiers**. Jan. 2009. Tese (Doutorado) – POLITECNICO DI TORINO.

MATHWORKS. **lsqnonlin: Solve nonlinear least-squares (nonlinear data-fitting) problems**. [S.l.: s.n.], 2025. Acessado em: 1 abr. 2025. Disponível em: ¡https://www.mathworks.com/help/optim/ug/lsqnonlin.html¿.

SCIPY. **scipy.optimize.least_squares: Solve a nonlinear least-squares problem with bounds on the variables**. [S.l.: s.n.], 2025. Acessado em: 20 abr. 2025. Disponível em: ¡https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.least_squares.html¿.