

Modelagem Comportamental de Amplificadores de Potência Usando Polinômios com Memória

Esse trabalho tem como objetivo a reprodução e análise de modelos baseados em polinômios com memória (MP) e suas variações, focando em diferentes técnicas de identificação e implementação

André Corso Pozzan

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

INTRODUÇÃO

Os sistemas de telecomunicações modernos exigem **altas taxas de transferência de dados** e, neste contexto, o Amplificador de Potência (PA) é um componente crítico na cadeia de transmissão, sendo responsável por amplificar a potência do sinal de entrada. Contudo, o PA é também o dispositivo que mais consome energia e o que mais gera distorções no sinal.

Para contornar este problema e manter a linearidade, é amplamente utilizado **técnicas de modelagem de comportamento** para os amplificadores, que buscam **representar matematicamente a relação entre o sinal de entrada e o sinal de saída**. Entre os diversos modelos existentes, os Modelos Polinomiais com Memória (MP) se destacam por sua capacidade de capturar tanto as não linearidades quanto os efeitos de memória.

MODELAGEM MATEMÁTICA

Deseja-se criar um **modelo matemático para o amplificador**.

O modelo escolhido é o polinômio com memória, um caso particular da série de **Volterra** como descrito em [1], cuja equação é:

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

Onde $\tilde{y}(n)$ é a saída estimada do modelo no instante n , $x(n-m)$ é a entrada no instante $(n-m)$, $b_{2p-1,m}$ são os coeficientes do modelo, P é o grau máximo do polinômio, e M é a profundidade de memória

MODELAGEM MATEMÁTICA

Em muitos casos é necessário trabalhar com **números complexos** com os polinômios, no entanto, os métodos de otimização convencionais **não são aplicáveis diretamente**, como solução é utilizada uma abstração dos complexos para que o algoritmo otimizador não utilize números complexos diretamente.

Dessa forma, para tal técnica são empregadas métodos para avaliar a precisão do modelo, o NMSE é dado por:

$$NMSE = 10 \log_{10} \left[\frac{\sum_{n=1}^N |e(n)|^2}{\sum_{n=1}^N |y_{\text{ref}}(n)|^2} \right]$$

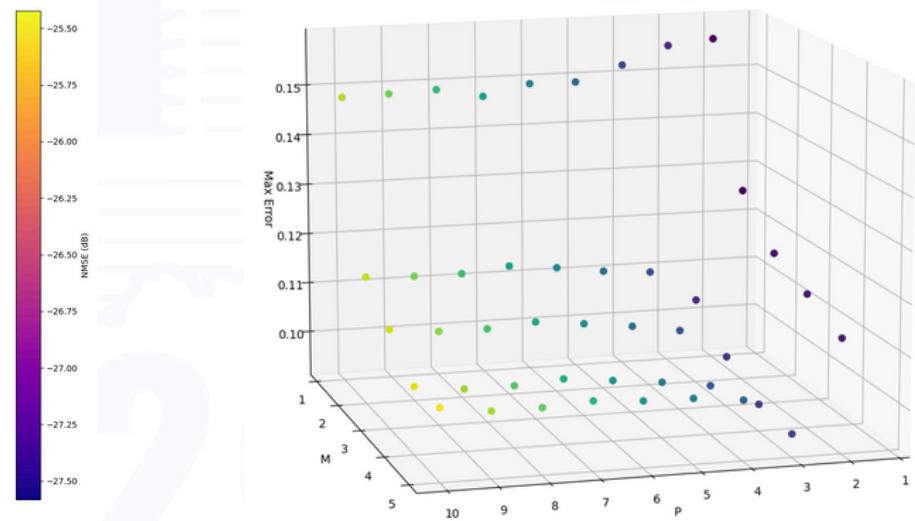
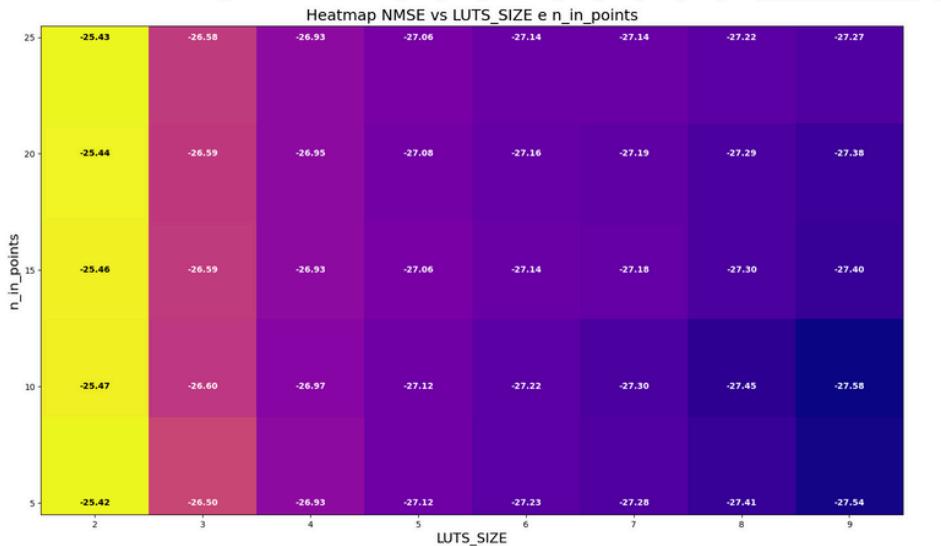
Normalized Mean Square Error (NMSE)

$$NMSE = 10 \log_{10} \left(\frac{\sum(\text{quadrados dos erros})}{\sum(\text{quadrados dos valores reais})} \right)$$

MODELAGEM MATEMÁTICA

Onde $e(n) = y_{\text{ref}}(n) - \hat{y}(n)$ é o erro entre a saída de referência e a estimada, e N é o número total de amostras. O **NMSE** é frequentemente expresso em decibéis (dB). Um valor de **NMSE menor** (mais negativo em dB) **indica uma melhor precisão do modelo**, ou seja, uma menor diferença entre os sinais medidos e estimados.

Representação de combinações dos parâmetros dos modelos com o NMSE e erro máximo:



MODELAGEM MATEMÁTICA

Nesse contexto, como já citado anteriormente, a complexidade computacional dos modelos polinomiais é um fator crucial em sua análise e aplicação. Assim, o estudo em questão abordou **técnicas para reduzir essa complexidade**, com destaque para o uso de **Lookup Tables (LUTs)**.

As LUTs são estruturas de dados que **armazenam valores pré-calculados** de funções ou expressões matemáticas, permitindo **acesso rápido a esses valores durante a execução do modelo**, em vez de recalculará-los repetidamente.

$$f_{\text{LUT}_m}(|\tilde{x}(n - m)|) = \tilde{s}_m(q - 1) + \left[\frac{\tilde{s}_m(q) - \tilde{s}_m(q - 1)}{e_m(q) - e_m(q - 1)} \right] \cdot (|\tilde{x}(n - m)| - e_m(q - 1))$$

MODELAGEM MATEMÁTICA

$$f_{\text{LUT}_m}(|\tilde{x}(n-m)|) = \tilde{s}_m(q-1) + \left[\frac{\tilde{s}_m(q) - \tilde{s}_m(q-1)}{e_m(q) - e_m(q-1)} \right] \cdot (|\tilde{x}(n-m)| - e_m(q-1))$$

- $\tilde{s}_m(q-1)$: saída complexa correspondente à entrada $e_m(q-1)$ (ponto inferior);
- $\tilde{s}_m(q)$: saída complexa correspondente à entrada $e_m(q)$ (ponto superior);
- $e_m(q-1)$: entrada real do limite inferior do intervalo de interpolação;
- $e_m(q)$: entrada real do limite superior do intervalo de interpolação;
- $|\tilde{x}(n-m)|$: amplitude da entrada para a qual se deseja estimar a saída.

Comparar com a fórmula padrão para interpolação linear:

$$P_1(x) = y_0 + \frac{y_1 - y_0}{x_1 - x_0}(x - x_0)$$

A interpolação será usada em uma função que auxilia na convergência do método que calcula os coeficientes.

MODELAGEM MATEMÁTICA

Com essa formulação, a implementação da matriz **LUT** permite estimar eficientemente os valores de saída durante a validação do modelo, utilizando **interpolação linear** entre os **pontos pré-calculados**.

Dessa forma, outro recurso importante utilizado para **otimizar o tempo de processamento** é a **estimativa inicial dos valores complexos**, que podem **reduzir o número de iterações** necessárias para a convergência do algoritmo de otimização. Assim, fizeram parte dos testes a inicialização unitária, com zeros e números aleatórios.

MODELAGEM MATEMÁTICA

Resumo das técnicas abordadas no trabalho:

- Uso da série de **Volterra** (polinômio com memória) ou **LUTs com interpolação linear** para cálculo dos coeficientes.
- Cálculo com números complexos.
- Estimar valores iniciais para os coeficientes.
- Análise de erros com **NMSE**.

DADOS

Os sinais analisados neste trabalho foram obtidos através de um amplificador de potência classe AB produzido com a tecnologia **GaN** (nitreto de gálio). Empregamos uma portadora centrada em **900 MHz**, modulada por um sinal de envoltória adequado para WCDMA (com uma largura de banda próxima a 3,84 MHz).

As ondas de entrada e saída foram documentadas utilizando um analisador vetorial de sinais Rohde Schwarz FSQ, operando com uma taxa de amostragem de **61,44 MHz**. Para os experimentos, os dados foram divididos em dois conjuntos: **3.221** amostras para extração (**treinamento**) e **2.001** amostras para **validação** do modelo.

CÓDIGOS EM PYTHON

Em todas as etapas do trabalho foram utilizados códigos em Python para realizar os cálculos matemáticos, simulações e geração de gráficos.

O **método principal** de otimização **não trabalha com números complexos**, então é necessário abstrair seu uso. O código que realiza a conversão entre vetores reais e complexos é mostrado na Figura:

```
def unpackComplexCoefficients(real_coef):
    # Metade do vetor são os componentes reais e a outra
    # metade são os imaginários
    real_parts = real_coef[:len(real_coef)//2]
    imag_parts = real_coef[len(real_coef)//2:]
    complex_coef = real_parts + 1j * imag_parts
    return complex_coef # Retorna vetor 1D de coeficientes
    # complexos

def packComplexCoefficients(complex_coef):
    # Metade do vetor são os componentes reais e a outra
    # metade são os imaginários
    real_parts = complex_coef.real
    imag_parts = complex_coef.imag
    real_coef = np.concatenate([real_parts, imag_parts])
    return real_coef
```

Para o uso da série de **Volterra**:

Iniciar os coeficientes com valores arbitrários:

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

# Inicialização com valores unitários
initial_complex = np.ones(P*(M+1)) + 1j * np.ones(P*(M+1))

# Inicialização com valores aleatórios
initial_complex = np.random.randn(P*(M+1)) + 1j *
                  np.random.randn(P*(M+1))
```

Desse modo, é possível reduzir o número de iterações necessárias para a convergência do algoritmo de otimização com as técnicas de inicialização com coeficientes estimados adequadamente escolhidos.

Implementação da Série de Volterra no cálculo dos coeficientes:

```
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 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

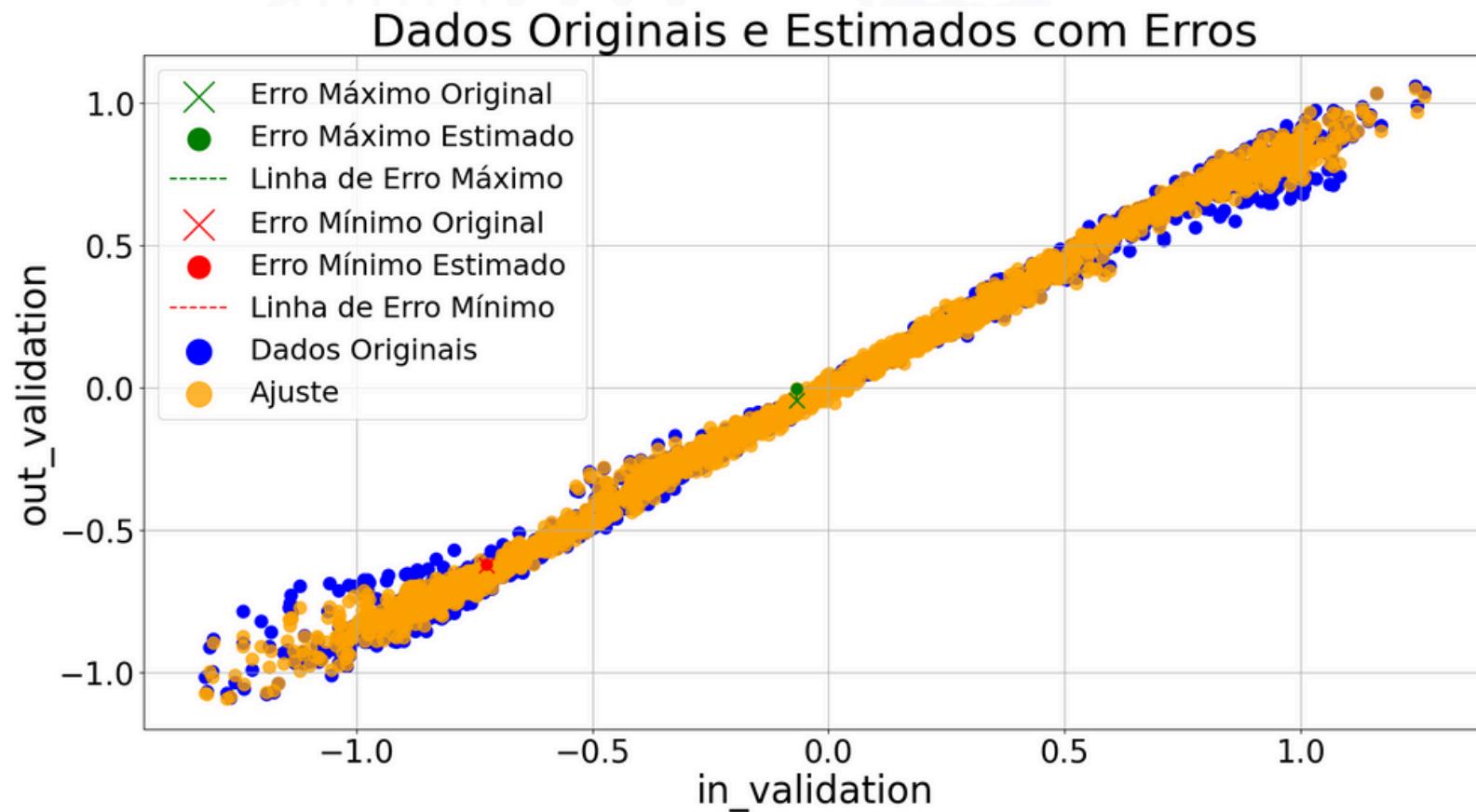
    return np.concatenate((resid.real, resid.imag))

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
```

Resultados: P=5, M=3



P: 5, M: 3

initial_estimated: [-0.37450898 0.81497983 -0.90908662 0.19970488 0.90943771 -0.76278512
 0.77425845 -0.08770478 -0.14102224 -1.10249407 1.23996207 0.10042498
 0.57148603 -2.15734699 -0.37882977 -0.13564024 -1.1426927 -1.01924581
 -0.00469158 -0.2729369 1.17304516 0.41898025 -0.38273821 1.21482358
 -1.22851314 0.19665682 -1.26649889 1.50382239 -0.75498267 0.92241675
 -1.06915919 -1.86455075 3.01734548 1.56737694 -0.66271964 -0.21340644
 1.23805091 -0.91843323 0.19075295 -1.50563541]

Iteration	Total nfev	Cost	Cost reduction	Step norm	Optimality
0	1	1.6019e+05		5.53e+04	
1	2	1.8113e+00	1.60e+05	6.74e+00	1.60e+01
2	3	1.7978e+00	1.34e-02	3.66e-02	2.90e-08
3	6	1.7978e+00	8.88e-16	5.64e-08	3.21e-08

`ftol` termination condition is satisfied.

Function evaluations 6, initial cost 1.6019e+05, final cost 1.7978e+00, first-order optimality 3.21e-08.

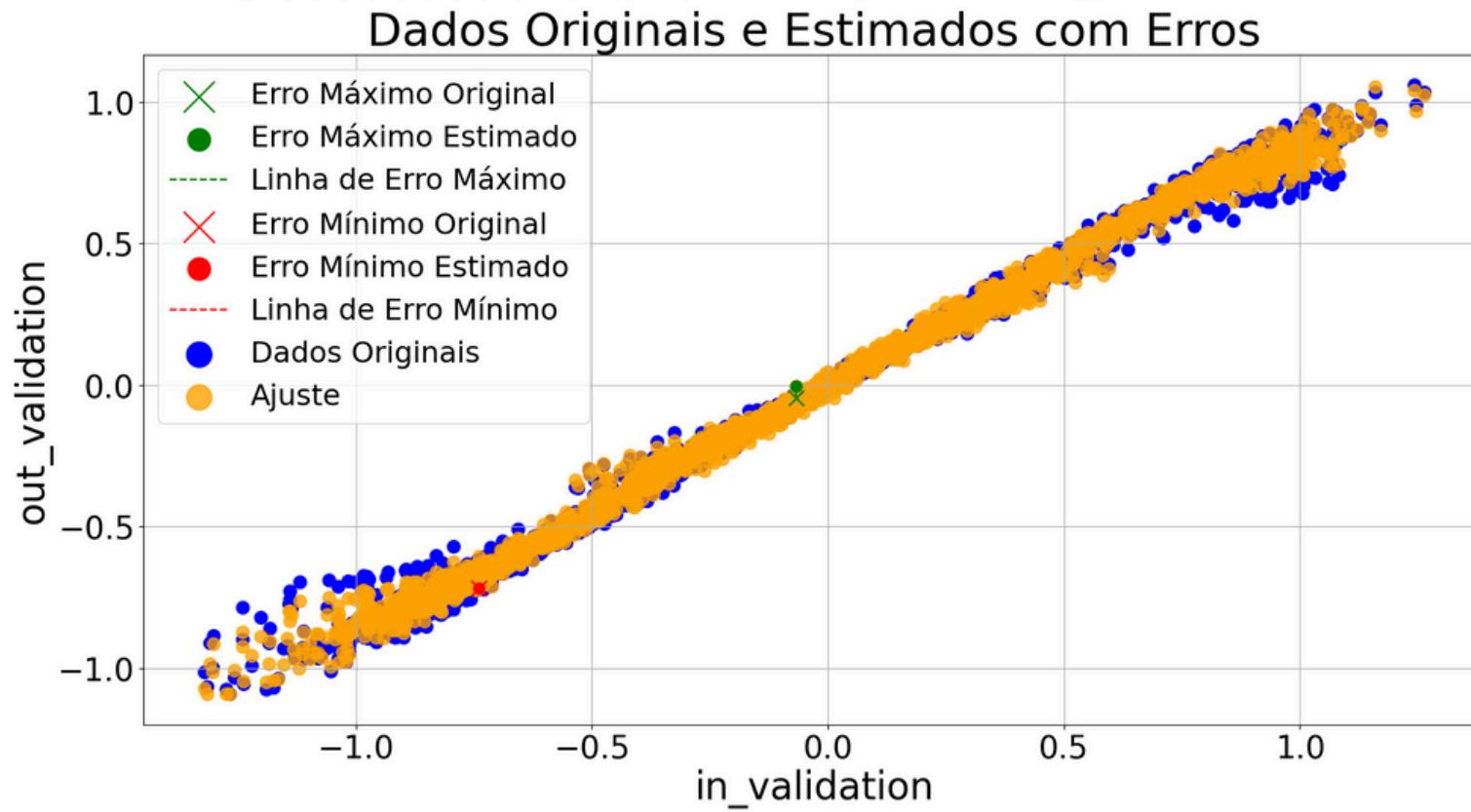
Resultado da otimização: [[1.03188629+0.21680638j 0.05961688-0.2975689j -0.64380925+0.04906722j
 0.28251041+0.00563618j]
 [0.20381252-0.02739967j -0.08882075-0.02370267j 0.34074468+0.11716156j
 0.15272088+0.0351565j]
 [-0.24905735-0.03172513j -0.26359023+0.07571073j 0.00928031-0.1408205j
 -0.19421885-0.04670957j]
 [0.01193239+0.03890497j 0.27291567-0.03166398j -0.07059931+0.07066077j
 0.08281843+0.02806583j]
 [0.01686848-0.00925963j -0.06717479+0.00461015j 0.02015225-0.01318752j
 -0.01506207-0.00641235j]]

Erro médio: 0.027326988325931376

Erro máximo em módulo: 0.1081032721546871

Erro mínimo em módulo: 0.0008754576788808584

Resultados: P=10, M=5



VOLTERRA

P: 10, M: 5

initial_estimated: [1.59045799 0.91679462 -0.34441474 -1.00267239 0.15119973 1.75279598
1.41393757 0.23116367 0.35094458 1.07665641 -0.46586614 -0.18589407
-0.38796401 0.16058817 -0.10583947 0.1690599 0.58883541 0.93991463
0.15414263 -0.13392873 1.9465728 1.51652163 1.37737172 -1.25095417
0.62056021 0.56743923 1.05461497 -1.15215268 1.33785954 0.25995072
0.19716398 1.0082755 1.27053404 -0.56978566 1.72069731 -0.94081286

.....

Iteration	Total nfev	Cost	Cost reduction	Step norm	Optimality
0	1	2.9713e+07		1.35e+07	
1	2	1.9180e+00	2.97e+07	9.93e+00	1.18e+03
2	3	1.6857e+00	2.32e-01	1.99e+01	7.13e-02
3	4	1.6844e+00	1.24e-03	3.97e+01	1.13e+00
4	5	1.6833e+00	1.17e-03	7.94e+01	2.39e+00
5	6	1.6832e+00	2.21e-05	1.55e+01	7.79e-01
6	8	1.6832e+00	2.74e-06	2.73e+00	1.24e-01
7	11	1.6832e+00	2.01e-09	1.71e-01	9.67e-03
8	12	1.6832e+00	1.23e-08	4.27e-02	8.52e-04
9	13	1.6832e+00	1.60e-08	1.07e-02	1.07e-03

`ftol` termination condition is satisfied.

Function evaluations 13, initial cost 2.9713e+07, final cost 1.6832e+00, first-order optimality 1.07e-03.

Resultado da otimização: [[1.09715810e+00+2.28062765e-01j -1.85414399e-03-2.23448709e-01j
-2.38131124e-01-2.41186676e-02j 2.50392651e-01+8.90953853e-02j
-2.52589164e-01+3.58670256e-02j 3.02222019e-01+1.42467116e-02j]

.....

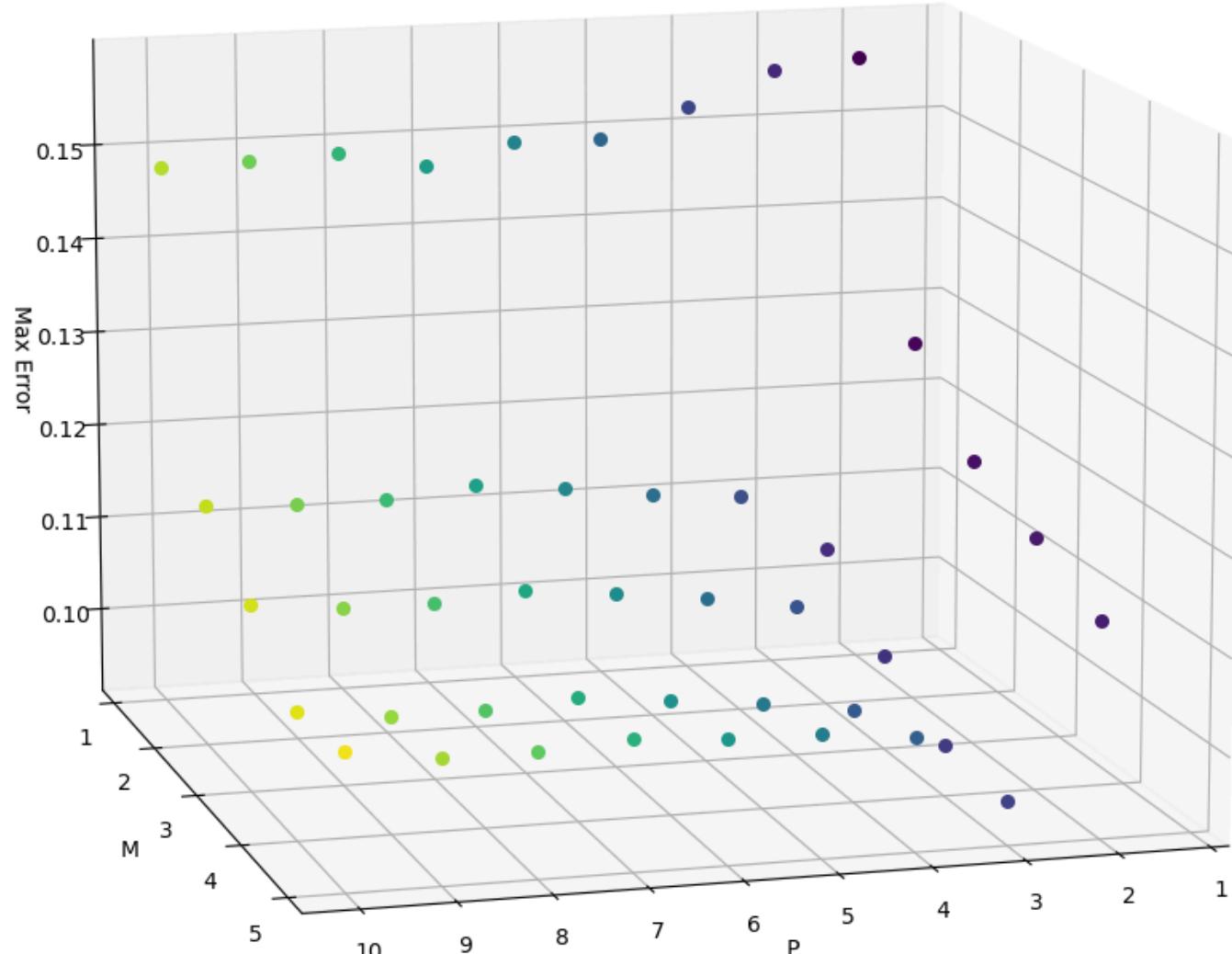
Erro médio: 0.027195941387310376

Erro máximo em módulo: 0.10516379271404204

Erro mínimo em módulo: 0.00048477668493392996

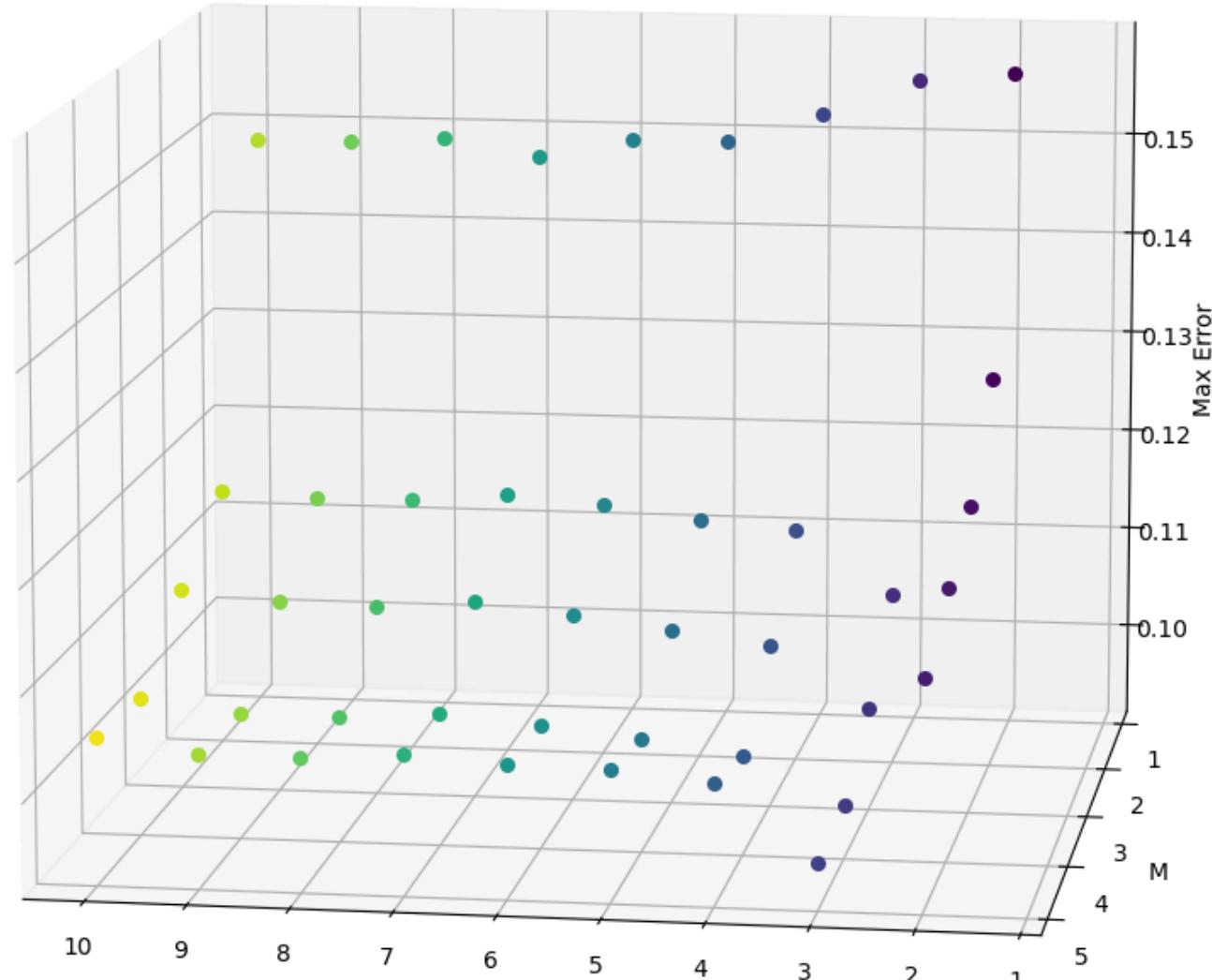
VOLTERRA

Combinações de P,M comparado com erro máximo:



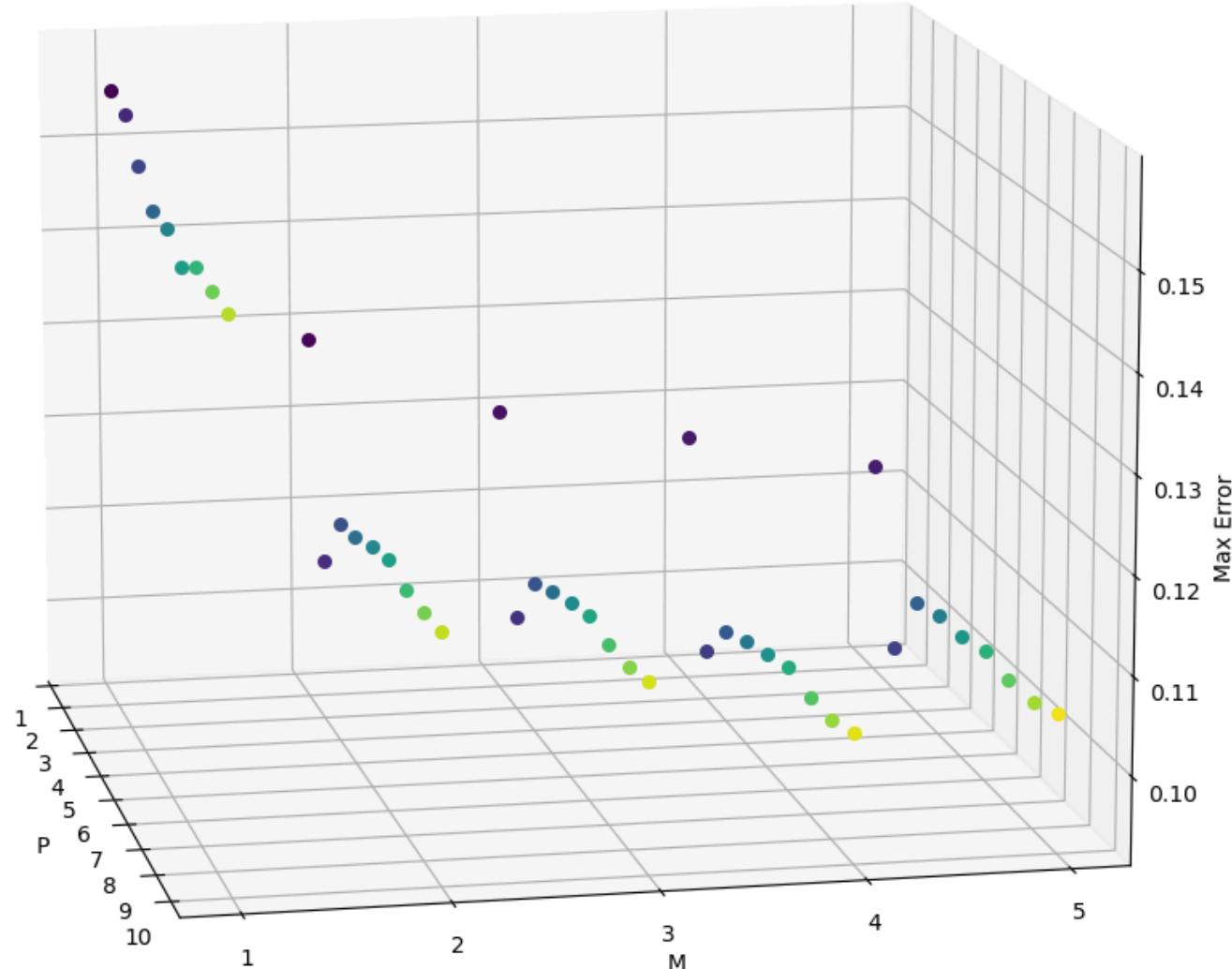
VOLTERRA

Combinações de P,M comparado com erro máximo:



VOLTERRA

Combinações de P,M comparado com erro máximo:



LUTs com Interpolação Linear

Intervalos igualmente espaçados e valores crescentes com início em zero indo até o valor máximo dos dados de treinamento, para garantir uma interpolação melhor.

```
n_in_points = 80
lut_in = np.linspace(0.0, np.max(np.abs(in_training)), n_in_points)
lut_out = out_training
```

A estimativa foi feita com o método `np.interp` para executar a interpolação linear.

```
def estimatedValueWithLUTOptimized(x_data, lut_out):
    x_abs = np.abs(x_data)
    real_interp = np.interp(x_abs, lut_in, lut_out.real)
    imag_interp = np.interp(x_abs, lut_in, lut_out.imag)
    result = x_data * (real_interp + 1j * imag_interp)
    return result
```

LUTs

```
def residuals(lut_out_real, x_data, y_data):
    lut_out_complex = unpackComplexCoefficients(lut_out_real)
    y_est = estimatedValueWithLUTOptimized(x_data, lut_out_complex)
    res = y_data - y_est
    res_vec = np.concatenate([res.real, res.imag])
    return res_vec

def calcCoef(in_data, out_data, n_in_points):
    initial_complex_coef = np.random.randn(n_in_points) + 1j * np.random.randn(n_in_points)
    initial_real_coef = packComplexCoefficients(initial_complex_coef)
    result = least_squares(residuals, initial_real_coef, args=(in_data, out_data), verbose=2)
    return unpackComplexCoefficients(result.x)

def calcOutOptimized(in_data, coef):
    # Interpolação separada para parte real e imaginária
    coef_real_interp = np.interp(np.abs(in_data), lut_in, coef.real)
    coef_imag_interp = np.interp(np.abs(in_data), lut_in, coef.imag)
    coef_interp = coef_real_interp + 1j * coef_imag_interp

    # Calcula saída
    calcResult = in_data * coef_interp

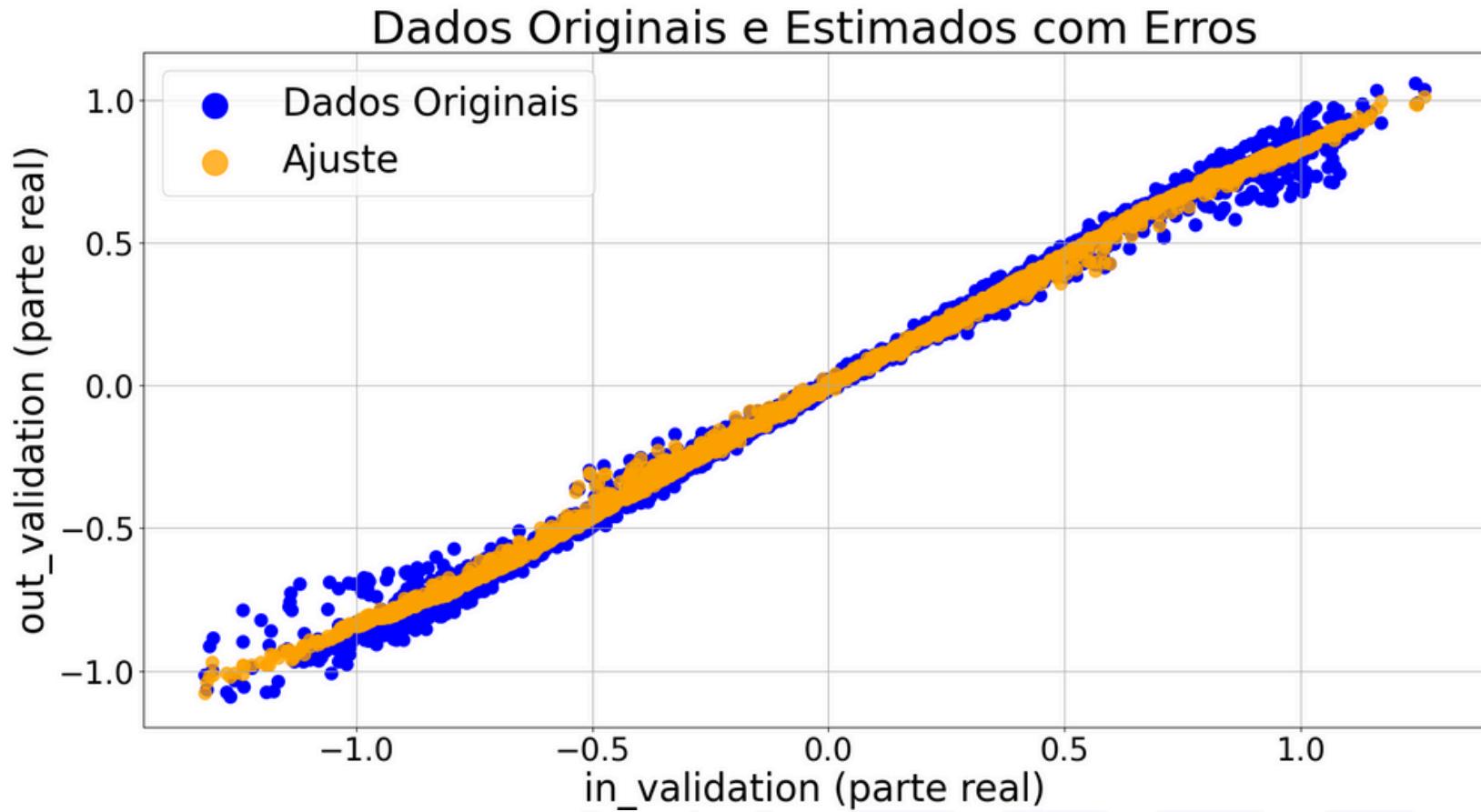
    # Calcula coeficientes polares (apenas para debug)
    coefComplexA = np.abs(coef_interp)
    coefComplexTeta = np.angle(coef_interp)
    polarCoefs = list(zip(coefComplexA, coefComplexTeta))

    print("Representação polar do primeiro coeficiente:", polarCoefs[0], "...\\n")
    print("Calculando saída otimizada com", len(in_data), "amostras e LUT de", len(coef), "pontos")

    return calcResult
```

LUTs

Resultado LUTs:



LUTs

Saída no terminal:

Tamanho dos dados de treinamento: 3221 x 3221

Iteration	Total nfev	Cost	Cost reduction	Step norm	Optimality
0	1	2.0390e+03		1.98e+02	
1	2	8.4286e+00	2.03e+03	1.23e+01	9.79e-01
2	3	4.9577e+00	3.47e+00	8.43e+00	2.22e-08
3	4	4.9577e+00	8.88e-16	1.19e-07	1.28e-08

`ftol` termination condition is satisfied.

Function evaluations 4, initial cost 2.0390e+03, final cost 4.9577e+00, first-order optimality 1.28e-08.

Representação polar do primeiro coeficiente: (np.float64(0.8813807236767855), np.float64(0.002828631378042488)) ...

Calculando saída otimizada com 2001 amostras e LUT de 80 pontos

NMSE: -22.693230 dB

LUTs

LUTs com tamanho variável:

Identificar os coeficientes de um MP já considerando a sua descrição através de LUTs. Em específico, ainda partindo do MP, rearranjar a matriz de regressão conforme detalhado na Seção 3.4.2 da dissertação da Carolina. A ideia aqui é usar LUTs de tamanho pequeno (sugestão: 4 ou 8 valores por LUT).

“Como forma de contornar as limitações da identificação tradicional descrita pela Seção 3.4.1, este trabalho propõe uma nova abordagem para obtenção dos valores a serem inseridos na LUTs. A proposta é obter diretamente estes valores, sem conhecimento prévio dos coeficientes das funções polinomiais unidimensionais. Assim sendo, neste caso os $(Q)(M + 1)$ valores complexos à serem inseridos nas $(M + 1)$ LUTs são obtidos diretamente através do MMQ, de acordo com:

$$s = (X_{LUT}^* X_{LUT})^{-1} (X_{LUT}^* Y) \quad (1)$$

Primeiramente, para a implementação de LUTs de tamanho variável foi criada a variável **LUTS_SIZE**, que define a profundidade de memória do modelo. Assim, as funções foram modificadas para trabalhar com a nova dimensão.

LUTs

Na sequência, a função de estimativa foi alterada para implementar um somatório que representa a contribuição de cada LUT associada ao seu atraso M:

```
def estimatedValueWithLUTS_SIZE(x_data, lut_out_matrix):
    n = len(x_data)
    result = np.zeros(n, dtype=complex)

    for M in range(LUTS_SIZE):
        delayed = np.roll(x_data, M)
        delayed[:M] = 0 # zera inicio (sem histrico real)
        print("DELAYED: ",delayed)

        x_abs = np.abs(delayed)
        real_interp = np.interp(x_abs, lut_in, lut_out_matrix[M].real)
        imag_interp = np.interp(x_abs, lut_in, lut_out_matrix[M].imag)
        result += delayed * (real_interp + 1j * imag_interp)
    return result
```

LUTs

O vetor delayed é a base do conceito de memória no modelo LUT-MP. No modelo, o sinal de entrada é deslocado (ou atrasado) de forma controlada para representar amostras anteriores; isso é equivalente a criar os termos de memória $x[n-M]$ da modelagem comportamental de amplificadores não lineares.

Assim, o modelo com LUTs implementa a seguinte expressão:

$$y_{est}[n] = \sum_{M=0}^{\text{LUTS_SIZE}-1} \tilde{x}[n - M] f_{\text{LUT},M}(|\tilde{x}[n - M]|)$$

3.1 Processo para LUT de tamanho 2

Para LUTS_SIZE = 2, executamos o script de avaliação e, conforme a Figura 3, obtemos o gráfico com o ajuste do modelo por LUT com memória. O desempenho é quantificado usando o NMSE definido na Equação 4.

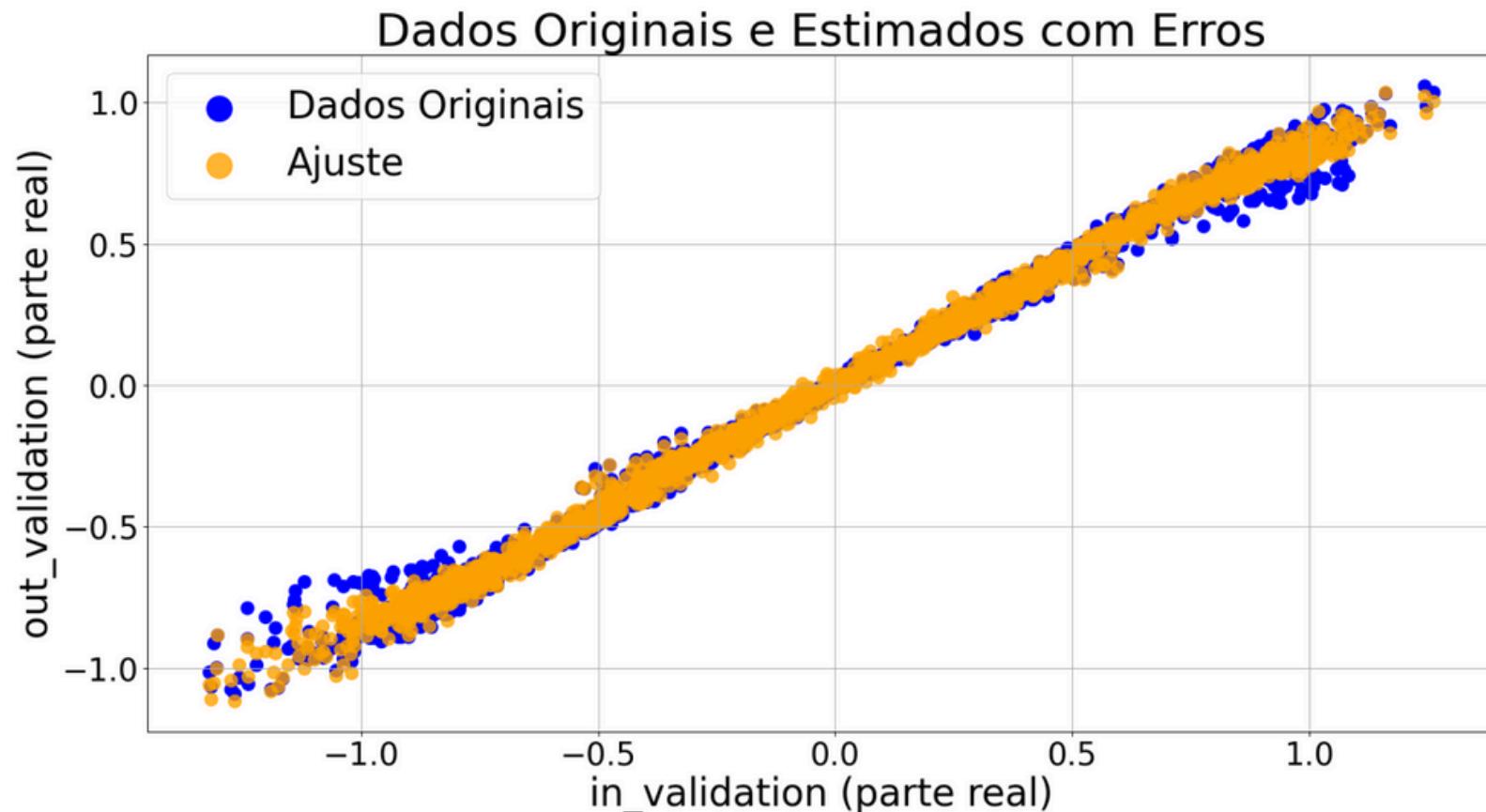


Figura 3: Resultados para LUT de tamanho 2: saída estimada pelo modelo LUT-MP versus a saída de referência.

LUTs

Saída no terminal:

```
Tamanho dos dados de treinamento: 3221 x 3221
Iteration      Total nfev      Cost      Cost reduction      Step norm      Optimality
 0              1             3.0685e+03
 1              2             2.8908e+00    3.07e+03      1.96e+01      1.75e-01
 2              3             2.5076e+00    3.83e-01      1.55e+01      3.37e-08
 3              6             2.5076e+00    4.44e-16      2.27e-07      8.08e-09
|gtol| termination condition is satisfied.
Function evaluations 6, initial cost 3.0685e+03, final cost 2.5076e+00, first-order optimality 8.08e-09.
NMSE: -25.307787 dB
```

3.2 Heatmap com diferentes tamanhos de LUT

Para testes, calculamos o NMSE (Equação 4) para diferentes tamanhos de LUTS_SIZE e diferentes números de pontos de entrada n_in_points. As Figuras 5–7 mostram mapas de calor relacionando essas variáveis ao desempenho.

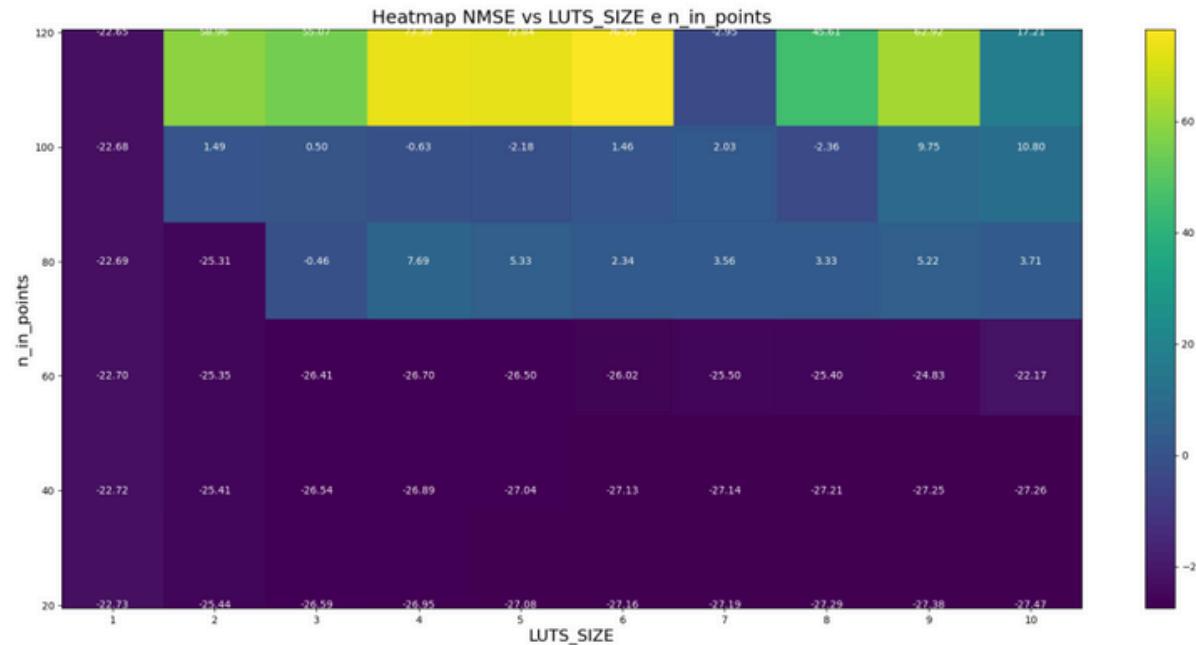
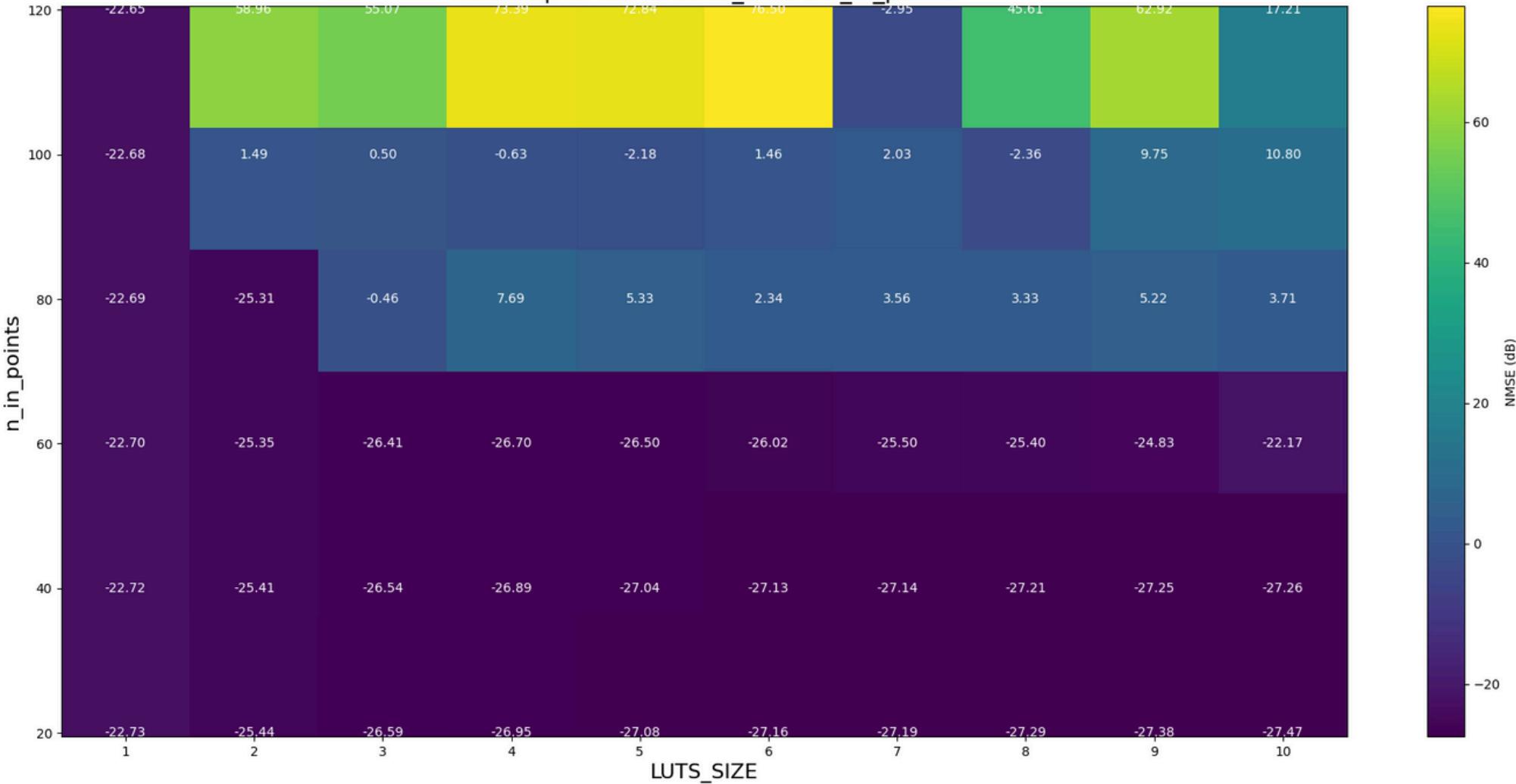


Figura 5: Heatmap do NMSE em função do tamanho da LUT (LUTS_SIZE) e do número de pontos de entrada (n_in_points).

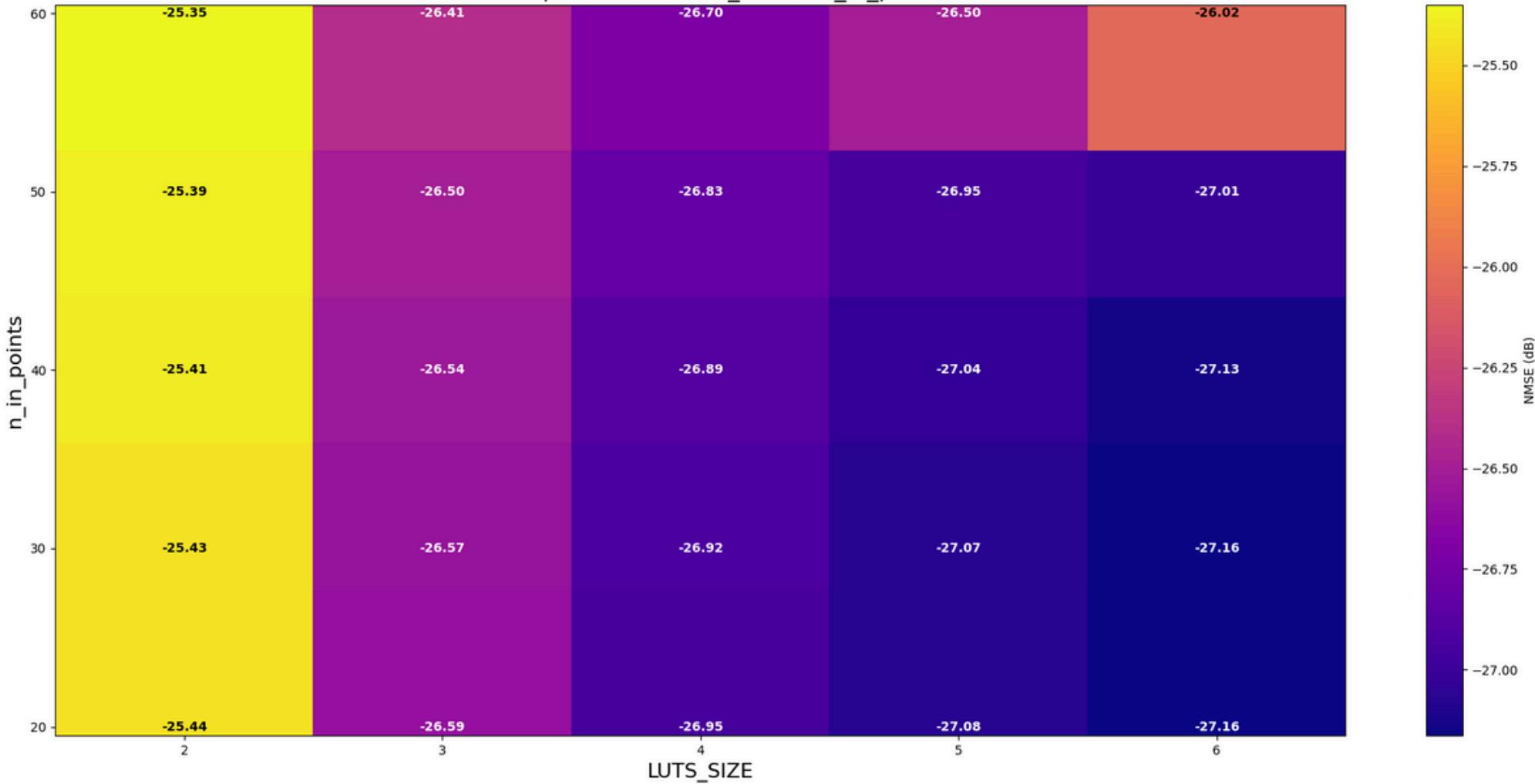
LUTs

Heatmap NMSE vs LUTS_SIZE e n_in_points



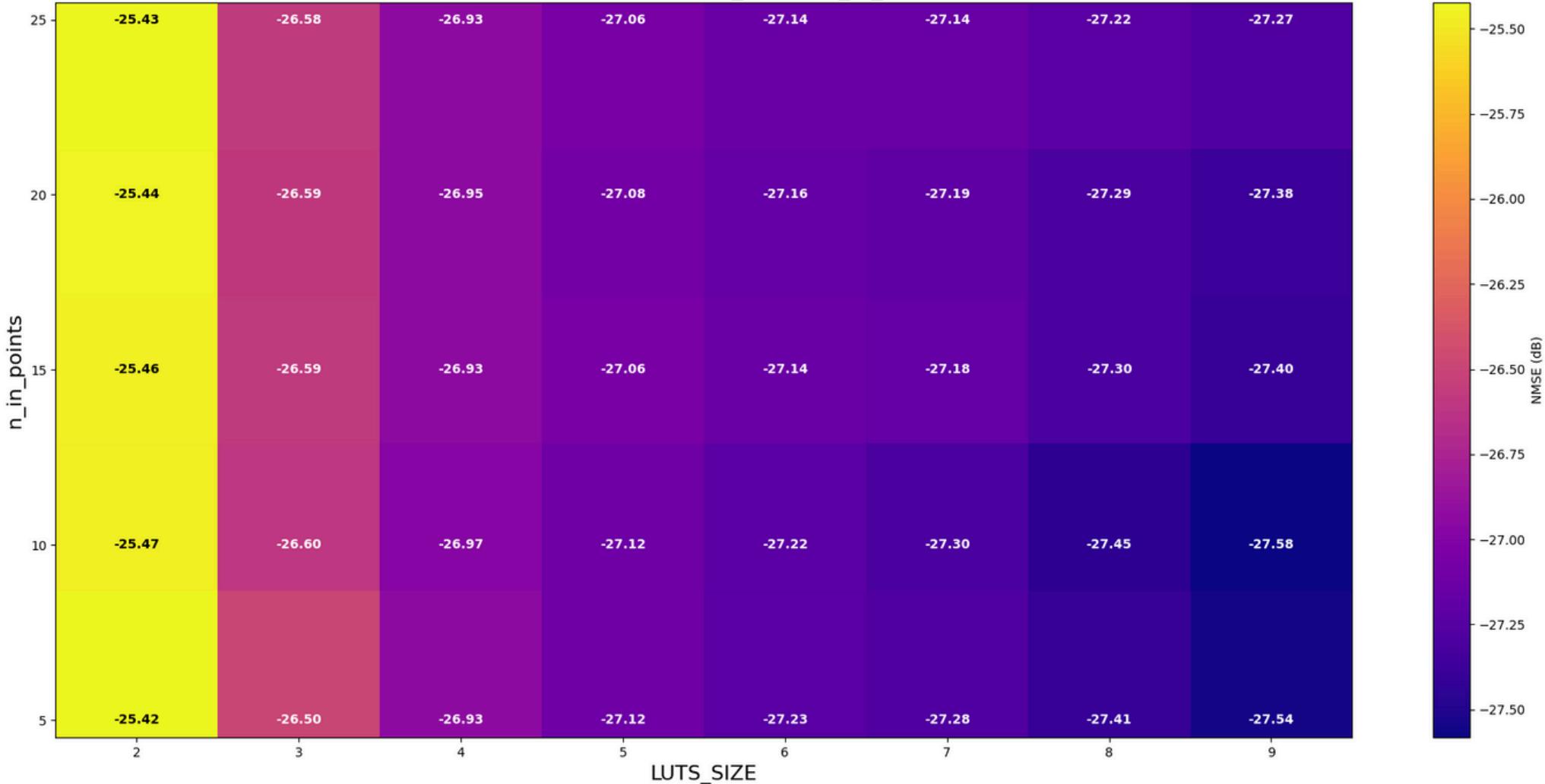
LUTs

Heatmap NMSE vs LUTS_SIZE e n_in_points

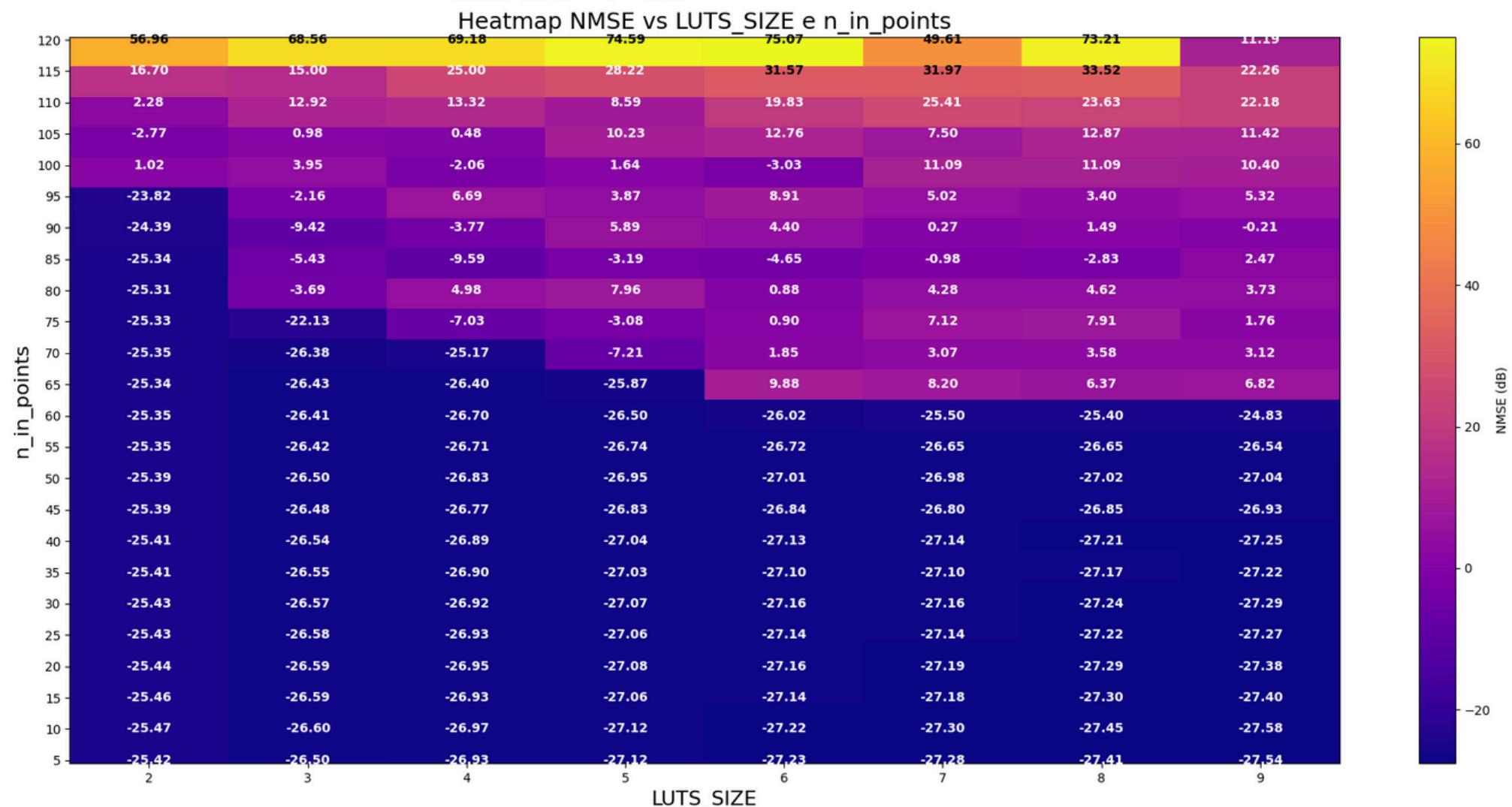


LUTs

Heatmap NMSE vs LUTS_SIZE e n_in_points



LUTs



FIM

Muito Obrigado!

Todos os **códigos em python** e **relatórios** estão presentes em um repositório no GitHub para consulta:

GitHub:



André Corso Pozzan
Prof. Ph.D. Eduardo Gonçalves de Lima