Departamento de Engenharia Elétrica
Universidade Federal do Paraná

Método de Otimização para Sistemas Complexos Usando Lookup Tables de Tamanho Variável

André Corso Pozzan

Atividade **7** – Iniciação Científica **GICS**

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

UFPR – Centro Politécnico – Curitiba, PR
19 de outubro de 2025

Conteúdo

1 Atividade 7

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:*
>
> $$s = \left(X_{LUT}^{*} X_{LUT}\right)^{-1} \left(X_{LUT}^{*} Y\right) \tag{1}$$
>
> "

1.1 Operador G_m para cada modelo

Tabela 1: Valores assumidos pelo operador G_m para cada um dos modelos analisados por este trabalho

	Operador G_m	
MP	$G_m = \tilde{x}(n-m)$	(todos diferentes)
EMP	$G_m = \tilde{x}(n)$	(todos iguais)
Proposto 1	$G_m = \sum_{m_1=0}^{M} \tilde{a}_{m_1+1} \tilde{x}(n-m_1)$	(todos iguais)
Proposto 2	$G_m = \sum_{m_1=0}^{M} \tilde{a}_{m+1,m_1+1} \tilde{x}(n-m_1)$	(todos diferentes)

(MACHADO, 2016)

2 Resolução

O modelo escolhido é o polinômio com memória, um caso particular da série de Volterra, cuja equação é:

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

onde: $\tilde{x}[n]$ é a entrada complexa do sistema; $\tilde{y}[n]$ é a saída complexa; M é a profundidade de memória; P é a ordem polinomial; e $\tilde{b}_{p,m}$ são os coeficientes complexos do modelo que ponderam cada termo de atraso m e ordem p.

Primeiramente, para a implementação de LUTs de tamanho variável foi criada a variável `LUTS_SIZE`, que define a profundidade de memória do modelo LUT-MP. Assim, as funções foram modificadas para trabalhar com a nova dimensão, como mostra a Figura 1.

```
def unpackComplexCoefficients(real_coef):
# Para matriz LUTS_SIZE x n_in_points
n = LUTS_SIZE * n_in_points
real_parts = real_coef[:n].reshape(LUTS_SIZE, n_in_points)
imag_parts = real_coef[n:].reshape(LUTS_SIZE, n_in_points)
complex_coef = real_parts + 1j * imag_parts
return complex_coef
```

Figura 1: Função para desempacotar coeficientes complexos (conversão do vetor real em matriz complexa `LUTS_SIZE` × `n_in_points`).

Na sequência, a função de estimativa foi alterada para implementar um somatório que representa a contribuição de cada LUT associada ao seu atraso M, como mostra a Figura 2.

```
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 incio (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
```

Figura 2: Função para estimar valores com LUT de tamanho variável: para cada atraso M, interpola-se a LUT correspondente em função de $|\tilde{x}[n-M]|$ e acumula-se a contribuição.

O vetor `delayed` é a base do conceito de memória no modelo LUT-MP. No seu modelo, o sinal de entrada `x_data` é deslocado (ou atrasado) de forma controlada para representar amostras anteriores; isso é equivalente a criar os termos de memória $\tilde{x}[n-M]$ da modelagem comportamental de amplificadores não lineares, coerentes com a Equação 2.

Assim, o modelo com LUTs implementa a seguinte expressão:

$$y_{est}[n] = \sum_{M=0}^{\text{LUTS\_SIZE}-1} \tilde{x}[n-M]\, f_{\text{LUT},M}\big(|\tilde{x}[n-M]|\big) \tag{3}$$

onde $f_{\text{LUT},M}(\cdot)$ é a função (obtida por interpolação) da LUT associada à profundidade de memória M. Cada LUT, portanto, representa a resposta não linear do sistema para um determinado atraso.

Para avaliar a qualidade do ajuste entre $\tilde{y}[n]$ e $\tilde{y}_{est}[n]$, utilizamos o erro normalizado médio quadrático (NMSE). O NMSE é dado por:

$$\text{NMSE}\,(\text{dB}) = 10\,\log_{10}\left(\frac{\sum_n \left|\tilde{y}[n]-\tilde{y}_{est}[n]\right|^2}{\sum_n \left|\tilde{y}[n]\right|^2}\right) \tag{4}$$

Por fim, a chamada das funções que realizam os cálculos é feita conforme o código apresentado.

2.0.1 Código completo

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

mat = loadmat('in_out_SBRT2_direto.mat')
#Flatten para garantir que so vetores de uma dimenso
in_training = mat['in_extraction'].flatten()
out_training = mat['out_extraction'].flatten()
in_validation = mat['in_validation'].flatten()
out_validation = mat['out_validation'].flatten()

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

LUTS_SIZE = 2

def unpackComplexCoefficients(real_coef):
    # Para matriz LUTS_SIZE x n_in_points
    n = LUTS_SIZE * n_in_points
    real_parts = real_coef[:n].reshape(LUTS_SIZE, n_in_points)
    imag_parts = real_coef[n:].reshape(LUTS_SIZE, n_in_points)
    complex_coef = real_parts + 1j * imag_parts
    return complex_coef

def packComplexCoefficients(complex_coef):
    # Para matriz LUTS_SIZE x n_in_points
    real_parts = complex_coef.real.flatten()
    imag_parts = complex_coef.imag.flatten()
    real_coef = np.concatenate([real_parts, imag_parts])
    return real_coef

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 incio (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

def residuals(lut_out_real, x_data, y_data):
    lut_out_complex = unpackComplexCoefficients(lut_out_real)
    y_est = estimatedValueWithLUTS_SIZE(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(LUTS_SIZE, n_in_points) +
                            1j * np.random.randn(LUTS_SIZE, 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 calculate_nmse(out_validation, saida_estimada):
    erro = out_validation - saida_estimada
    nmse = 10 * np.log10(np.sum(np.abs(erro)**2) / np.sum(np.abs(out_validation)**2))
    return nmse

print("Tamanho dos dados de treinamento:",len(in_training),"x", len(out_training))

optimized_coef = calcCoef(in_training, out_training, n_in_points)

out_estimated = estimatedValueWithLUTS_SIZE(in_validation, optimized_coef)

nmse = calculate_nmse(out_validation, out_estimated)
print(f"NMSE: {nmse:.6f} dB")

plt.scatter(in_validation.real, out_validation.real, label='Dados Originais', color='blue', s=100)
plt.scatter(in_validation.real, out_estimated.real, color='orange', label='Ajuste', alpha=0.8, s=100)
plt.xlabel('in_validation (parte real)', fontsize=30)
plt.ylabel('out_validation (parte real)', fontsize=30)
plt.title('Dados Originais e Estimados com Erros', fontsize=36)
plt.legend(fontsize=30, markerscale=2)
plt.grid()
plt.tick_params(axis='both', which='major', labelsize=25)
plt.show()
```

O código completo está disponível no seguinte endereço do GitHub:

`script-7-lut.py` — Repositório no GitHub

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

github.com/andrepozzan/ic-gics

3 Resultados

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.

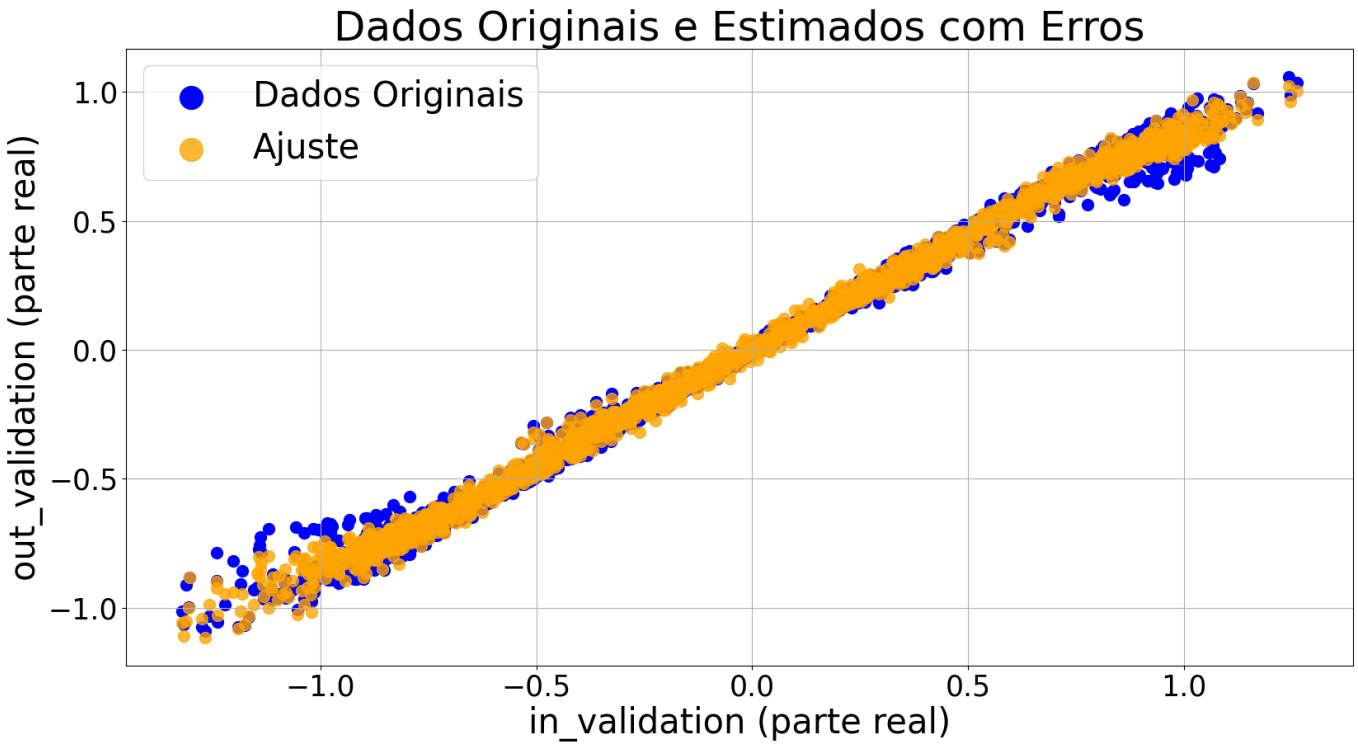

Figura 3: Resultados para LUT de tamanho 2: saída estimada pelo modelo LUT-MP versus a saída de referência.

Além disso, a Figura 4 apresenta a saída textual do processo de otimização (redução de custo e NMSE), em que observamos convergência rápida e erro baixo.

```
Tamanho dos dados de treinamento: 3221 x 3221
Iteration     Total nfev        Cost      Cost reduction    Step norm     Optimality
0                  1         3.0685e+03                                    1.63e+02
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
```

Figura 4: Log resumido da execução do método para `LUTS_SIZE = 2`, com iterações, custo e NMSE final (Equação 4).

Esses resultados indicam boa precisão com poucas iterações, demonstrando a eficiência do uso de LUTs com memória.

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.

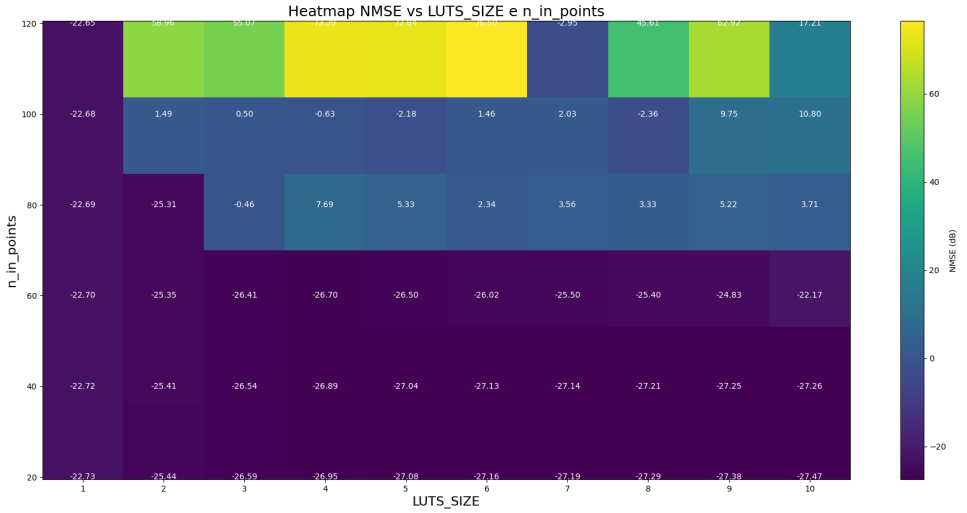

Figura 5: Heatmap do NMSE em função do tamanho da LUT (`LUTS_SIZE`) e do número de pontos de entrada (`n_in_points`).

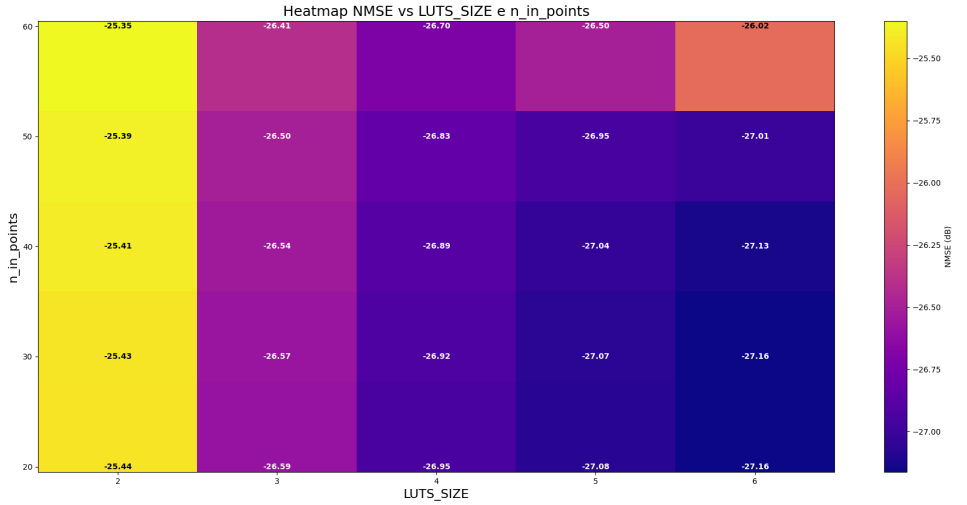

Figura 6: Heatmap detalhando os melhores intervalos para `LUTS_SIZE` até 6.

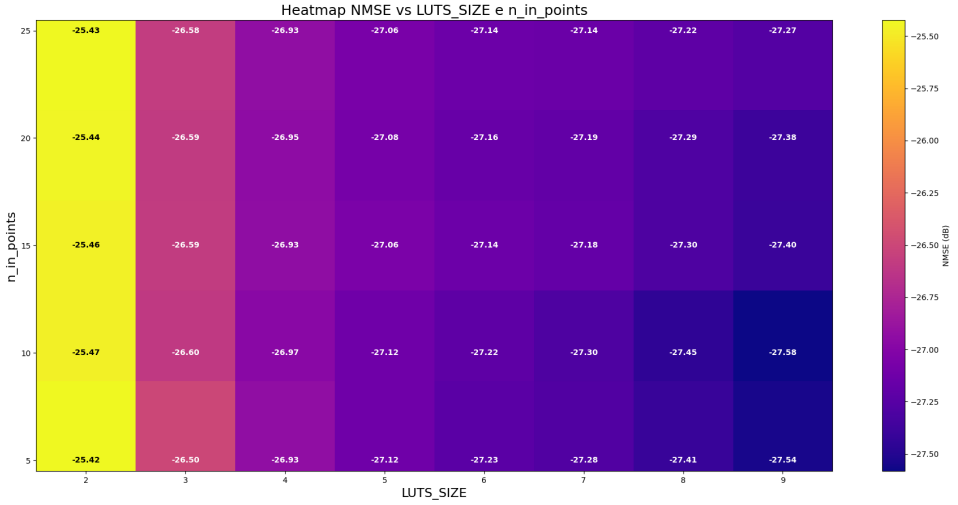

Figura 7: Heatmap detalhando os melhores intervalos para `LUTS_SIZE` até 9.

Onde `n_in_points` é o número de pontos de entrada (divisões) para cada dimensão da LUT, e `LUTS_SIZE` é a profundidade de memória (quantidade de atrasos considerados no somatório da Equação 3).

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

Observamos que, para valores moderados de `n_in_points`, o aumento de `LUTS_SIZE` tende a melhorar o NMSE, e há regiões favoráveis de compromisso entre resolução de magnitude e profundidade de memória.

Com maior quantidade:

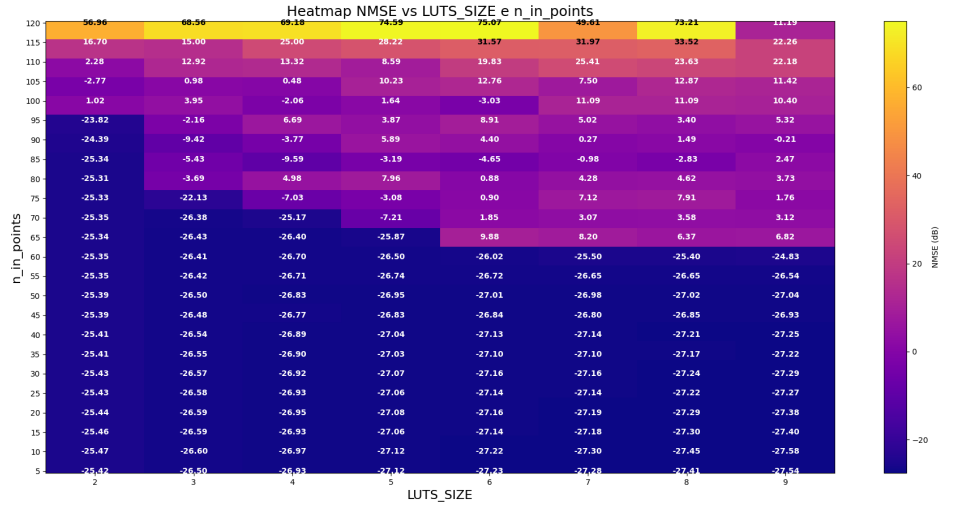

Figura 8: Visão global dos resultados envolvendo diferentes combinações de `LUTS_SIZE` e `n_in_points`, consolidados no NMSE (Equação 4).

Referências

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

MACHADO, Carolina Luiza Rizental. **Modelagem comportamental de amplificadores de potência usando soma de produtos entre filtros digitais de resposta ao impulso finita e tabelas de busca unidimensionais**. 2016. Diss. (Mestrado) – Universidade Federal do Paraná, Curitiba. Disponível em: ¡https://acervodigital.ufpr.br/handle/1884/46250¿.

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¿.