Departamento de Engenharia Elétrica
Universidade Federal do Paraná

Método de Otimização para Sistemas Complexos Usando Lookup Tables e Interpolação Linear

André Corso Pozzan

Atividade **6** – Iniciação Científica **GICS**

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

UFPR – Centro Politécnico – Curitiba, PR
8 de agosto de 2025

Conteúdo

1 Atividade 6

Mantendo os coeficientes já identificados na Etapa 3 ou 5, modificar apenas o código de validação. Em específico, na validação, obter a saída estimada substituindo os polinômios por LUTs com interpolação linear, conforme detalhado na Seção 3.3 da dissertação:

> *"Com o intuito de diminuir a complexidade computacional da implementação de modelos polinomiais, a substituição de polinômios por Lookup Tables (LUTs), ou tabelas de busca unidimensionais, foi proposta em [2]. LUTs são representações matriciais que armazenam na memória dados pré-calculados que serão utilizados como referência para outros valores e/ou cálculos. Portanto, a implementação com LUTs ajuda a diminuir a complexidade computacional da resolução do problema pois substitui a execução de cálculos por operações de indexação de matrizes."*

(MACHADO, 2016)

2 Resolução

Uma Look-Up Table (LUT), ou tabela de busca unidimensional, é uma estrutura de dados (como uma matriz ou vetor) que armazena pares de valores de entrada e suas saídas correspondentes que foram previamente calculadas.

Esses dados na memória pré-calculados serão utilizados como referência para outros valores e/ou cálculos por meio da interpolação que nesse caso é linear.

2.1 Interpolação linear

2.1.1 Definição

Dados dois pontos distintos, (x_0, y_0) e (x_1, y_1), de uma função $y = f(x)$, deseja-se calcular o valor estimado $\bar{y}$ para um valor não amostrado, entre x_0 e x_1, usando a interpolação polinomial.

O polinômio interpolador é uma unidade menor do que o número de pontos conhecidos.

Assim, o polinômio interpolador, nesse caso, terá grau 1, isto é:

$$P_1(x) = a_1 x + a_0$$

Para determinar este polinômio, os coeficientes a_0 e a_1 devem ser calculados de forma que se tenha:

$$P_1(x_0) = f(x_0) = y_0 \quad \text{e} \quad P_1(x_1) = f(x_1) = y_1$$

Ou seja, basta resolver o sistema linear abaixo:

$$\begin{cases} a_1 x_0 + a_0 = y_0 \\ a_1 x_1 + a_0 = y_1 \end{cases} \qquad \text{onde } a_1 \text{ e } a_0 \text{ são as incógnitas, e}$$

$$A = \begin{bmatrix} x_0 & 1 \\ x_1 & 1 \end{bmatrix} \qquad \text{é a matriz dos coeficientes.}$$

O determinante da matriz A é diferente de zero sempre que $x_0 \neq x_1$, logo, para pontos distintos, o sistema tem solução única.

2.1.2 Dedução da equação para Interpolação Linear

Queremos encontrar o polinômio de grau 1 na forma:

$$P_1(x) = a_1 x + a_0$$

Esse polinômio deve passar por dois pontos conhecidos:

$$P_1(x_0) = y_0 \Rightarrow a_1 x_0 + a_0 = y_0$$

$$P_1(x_1) = y_1 \Rightarrow a_1 x_1 + a_0 = y_1$$

Subtraindo as duas equações para eliminar a_0:

$$(a_1 x_1 + a_0) - (a_1 x_0 + a_0) = y_1 - y_0$$

$$a_1(x_1 - x_0) = y_1 - y_0$$

Isolando a_1:

$$\boxed{a_1 = \frac{y_1 - y_0}{x_1 - x_0}}$$

Substituindo a_1 em uma das equações para encontrar a_0:

$$a_1 x_0 + a_0 = y_0 \Rightarrow a_0 = y_0 - a_1 x_0$$

$$\boxed{a_0 = y_0 - \frac{y_1 - y_0}{x_1 - x_0} \cdot x_0}$$

Substituindo a_1 e a_0 na expressão original do polinômio:

$$P_1(x) = a_1 x + a_0$$

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

Colocando em evidência:

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

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

2.1.3 Aplicação para LUTs

Para o caso das LUTs ao invés de x_n e y_n vamos considerar as entradas e saídas dos dados com números complexos, dessa forma a seguinte equação demostra a interpolação linear usando os elementos calculados na matriz LUT:

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

(MACHADO, 2016)
Onde:

- $\tilde{s}_m(q-1)$: A saída complexa correspondente à entrada $e_m(q-1)$, $\Rightarrow y_0$.
- $\tilde{s}_m(q)$: A saída complexa correspondente à entrada $e_m(q)$, $\Rightarrow y_1$.
- $e_m(q-1)$: A entrada real inferior do intervalo, $\Rightarrow x_0$.
- $e_m(q)$: A entrada real superior do intervalo, $\Rightarrow y_1$.
- $|\tilde{x}(n-m)|$: A amplitude de entrada para a qual se deseja estimar a saída, $\Rightarrow x$.

Agora basta montar a matriz LUT no código e utilizar a fórmula da interpolação para estimar os valores na função de validação.

O cálculo estimado dos erros foi alterado para utilizar o NMSE(Normalized Mean Square Error) e com isso melhor representar os resultados nos gráficos.

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

2.2 Código em Python

O código desta atividade apresenta o mesmo fluxo de operação do anterior, porém foram feitas as modificações para o uso de LUTs com interpolação linear na função de resíduos da otimização e também para obtenção dos resultados.

2.2.1 Definições iniciais

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
```

2.2.2 Compactação e descompactação de números complexos

Utilizado por conta de que o método principal de otimização não trabalha com números complexos, então é necessário abstrair seu uso.

```
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
```

2.2.3 Estimativa com interpolação

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

```python
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
```

2.2.4 Cálculos

As funções foram adaptadas para o nomo método, uma interpolação final for necessária para encontrar os resultados baseado nos dados do intervalo igualmente espaçado definido inicialmente.

```python
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
```

Chamada das funções que realizam os cálculos

```python
optimized_coef = calcCoef(in_training, out_training, n_in_points)

out_estimated = calcOutOptimized(in_validation, optimized_coef)
```

2.2.5 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')
in_training = mat['in_extraction'].flatten()
out_training = mat['out_extraction'].flatten()
in_validation = mat['in_validation'].flatten()
```

```python
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

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

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

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

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

O código completo está disponível no seguinte endereço do GitHub:

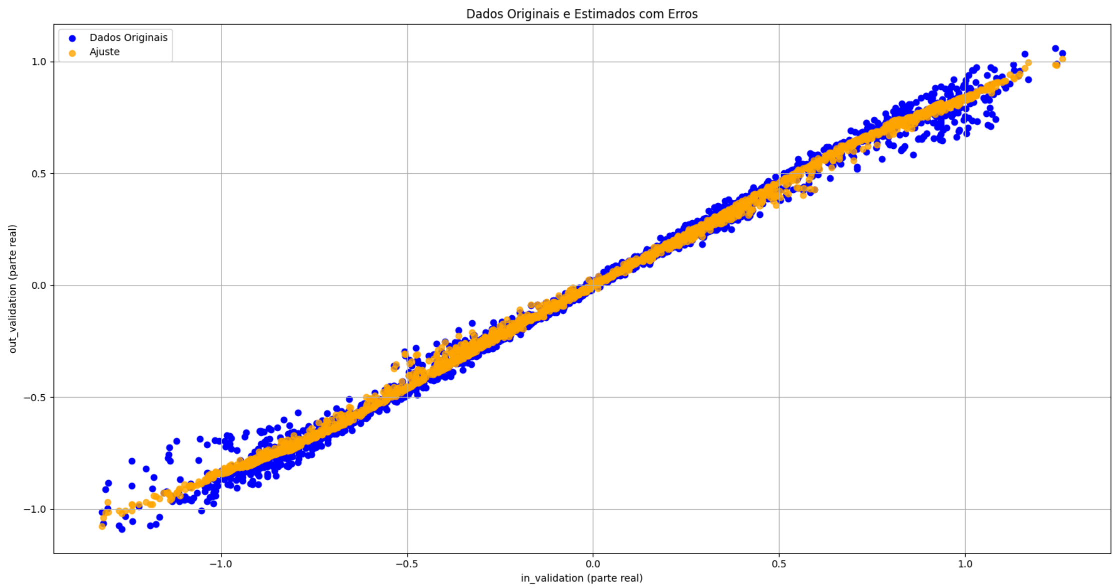
`script-6-lut.py` — Repositório no GitHub

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

github.com/andrepozzan/ic-gics

3 Resultados

Executando `python3 script-6-lut.py > terminal-out.txt` obtemos o gráfico:



e a saída no arquivo:

```
Tamanho dos dados de treinamento: 3221 x 3221
   Iteration     Total nfev        Cost      Cost reduction    Step norm     Optimality
       0              1         3.4220e+03                                    2.72e+02
       1              2         3.2063e+01      3.39e+03       1.17e+01       5.21e+00
       2              3         4.9577e+00      2.71e+01       8.73e+00       4.14e-08
       3              4         4.9577e+00      8.88e-16       1.08e-07       1.20e-08
`ftol` termination condition is satisfied.
Function evaluations 4, initial cost 3.4220e+03, final cost 4.9577e+00, first-order optimality 1.20e-08.
Representação polar do primeiro coeficiente: (np.float64(0.8813807236646474), np.float64(0.0028286314169537045)) ...

Calculando saída otimizada com 2001 amostras e LUT de 80 pontos
NMSE: -22.693230 dB
```

Observa-se que o NMSE alcançou um valor aceitável e que o custo computacional dessa operação foi relativamente baixo o que demostra a principal vantagem da utilização de LUTs em relação a métodos mais avançados.

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¿.