SeMicro-PR 2025

Modelagem Comportamental de Amplificadores de Potência Usando Polinômios com Memória

André Corso Pozzan[1], Eduardo Gonçalves de Lima[1]
[1]Departamento de Engenharia Elétrica, Universidade Federal do Paraná, Curitiba, PR, Brasil
E-mails de contato: andrepozzan@ufpr.br, eduardo.lima@ufpr.br

Resumo: ***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. Tais modelos são cruciais na engenharia, aplicados notavelmente na modelagem comportamental de amplificadores de potência (PAs) e na previsibilidade operacional de componentes. Dada a ausência de um modelo universalmente superior, o desempenho ótimo depende das características específicas do PA e dos sinais de entrada. O trabalho envolveu variações na topologia, como o uso de séries de Volterra, a implementação de otimização de erros com números complexos, e técnicas para estimar coeficientes iniciais. Buscou-se também a redução da complexidade computacional com Lookup Tables (LUTs). Para isso, foram utilizadas ferramentas matemáticas e simulações em Python. Foi possível observar diferentes níveis de precisão e demandas computacionais entre os modelos, sendo o objetivo aplicar aquele que ofereça o melhor custo-benefício entre precisão e complexidade. A precisão foi avaliada por métricas como o Erro Quadrático Médio Normalizado (NMSE), e por análises AM-AM e AM-PM que comparam a relação de amplitude e fase entre o PA real e o modelo estimado. Ao representar um componente com a precisão necessária, é possível estipular seu comportamento de saída sem a necessidade de testes laboratoriais diretos, otimizando tempo e recursos.***

I. 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 [2].

O projeto de PAs enfrenta um compromisso entre linearidade e eficiência. Esta operação não linear com efeitos de memória causados por consequências da topologia e fatores físicos como o auto aquecimento do transistor, resulta em distorções que comprometem a qualidade do sinal transmitido [1, 3].

Para contornar este problema e manter a linearidade, é amplamente utilizado técnicas de modelagem de comportamento para os amplificadores como descrevem os trabalhos [1, 2, 3], 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.

II. MODELAGEM MATEMÁTICA

Inicialmente foram realizados cálculos com otimização linear por meio do método dos mínimos quadrados (MMQ) para encontrar os coeficientes do modelo polinomial. O objetivo é minimizar a soma dos quadrados dos erros entre a saída real do PA e a saída estimada pelo modelo.

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:

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

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

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 como a função de resíduo que é responsável por auxiliar o otimizador para a conversão.

Para avaliar a precisão do modelo, o NMSE é dado por:

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

onde $e(n) = y_{\text{ref}}(n) - \tilde{y}(n)$ é o erro entre a saída de referência e a estimada, e N é o número total de amostras conforme [2]. 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.

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 recalculá-los repetidamente.

Para a implementação das LUTs, considera-se que as entradas e saídas dos dados são números complexos. A interpolação

linear entre os elementos armazenados na matriz LUT é realizada conforme [3], sendo expressa pela seguinte equação:

$$f_{\mathrm{LUT}_m}(|\tilde{x}(n-m)|) = \tilde{s}_m(q-1) + \left[\frac{\tilde{s}_m(q) - \tilde{s}_m(q-1)}{e_m(q) - e_m(q-1)}\right] \cdot (|\tilde{x}(n-m)| - e_m(q-1)) \quad (3)$$

onde as variáveis são definidas como:

- $\tilde{s}_m(q-1)$: 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.

Considera-se o *ponto inferior* como aquele imediatamente abaixo do valor desejado de $|x(\tilde{n}-m)|$ na tabela, representado por $e_m(q-1)$, e o *ponto superior* como o imediatamente acima, representado por $e_m(q)$. Dessa forma, a interpolação linear é realizada com as condições necessárias de continuidade atendidas.

Desse modo, $x(\tilde{n}-m)$ representa o sinal de entrada complexo no instante de tempo discreto $\tilde{n}$, considerando o atraso m. Seu módulo $|x(\tilde{n}-m)|$ é utilizado como referência para indexar a LUT, enquanto $e_m(q)$ corresponde aos valores reais de entrada previamente amostrados que formam o eixo da tabela. Já $\tilde{s}_m(q)$ representa a saída complexa esperada associada a cada ponto $e_m(q)$, de modo que a LUT descreve a relação entre a amplitude de entrada e a resposta complexa do sistema.

Em outras palavras, esses dois pontos definem o intervalo de interpolação dentro do qual o valor de entrada atual está contido.

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.

III. CÓDIGOS EM PYTHON

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

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 1.

```python
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
```

Figura 1: *Funções para empacotar e desempacotar coeficientes complexos*

Para a inicialização dos coeficientes complexos, foram testadas três estratégias diferentes: inicialização com zeros, com valores unitários e com números aleatórios. O código mostrado na Figura 2 ilustra essas opções:

```python
# 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))
```

Figura 2: *Estratégias de inicialização dos coeficientes complexos*

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.

A combinação de LUTs com estratégias de inicialização resulta em métodos com boa precisão e rápida convergência. Nesse contexto, na Figura 3 é apresentada a saída no terminal das iterações do otimizador resultante de um processo que utilizou mínimos quadrados, números complexos, LUTs e inicialização aleatória dos coeficientes complexos.

```
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:
↪  (0.8813807236646474, 0.0028286314169537045) ...

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

Figura 3: *Saída do terminal mostrando a convergência do algoritmo de otimização*

Observa-se que o NMSE alcançou um valor aceitável (-22,69 dB) e que o custo computacional dessa operação foi relativamente baixo, com apenas 4 iterações para convergência. Isso demonstra a principal vantagem da utilização de LUTs em relação a métodos mais complexos, proporcionando um bom equilíbrio entre precisão e eficiência computacional.

A estimativa para a função de resíduos que auxilia na convergência do método foi feita com o método `np.interp` para executar a interpolação linear (veja Figura 4).

```
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
```

Figura 4: *Estimativa otimizada usando LUTs e interpolação*

As funções que estimam os coeficientes e calculam a saída otimizada são mostradas na Figura 5. A otimização é realizada com um solucionador de mínimos quadrados não linear (implementado aqui via SciPy), conforme documentado em [5] e [4].

```
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
```

Figura 5: *Funções de cálculo de coeficientes e saída otimizada*

Chamada das funções que realizam os cálculos com a passagem dos dados de treinamento e validação, também já é retornado os valores estimados do modelo (veja Figura 6).

```
optimized_coef = calcCoef(in_training, out_training,
↪  n_in_points)
# ... carregamento e pré-processamento dos dados ...
out_estimated = calcOutOptimized(in_validation,
↪  optimized_coef)
```

Figura 6: *Chamada das funções que realizam os cálculos*

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

github.com/andrepozzan/ic-gics

IV. RESULTADOS

Os sinais analisados neste trabalho foram obtidos através de um amplificador de potência classe AB produzido com a tecnologia GaN. 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.

A Figura 7 apresenta a característica AM-AM (amplitude-amplitude) do amplificador, onde os pontos em azul representam o comportamento real do amplificador, indicando como a amplitude do sinal de saída varia em resposta às mudanças na amplitude do sinal de entrada. Em um amplificador idealmente linear, essa relação seria diretamente proporcional. Do mesmo modo, os pontos em laranja representam o resultado gerado pelo polinômio aplicado nas mesmas entradas que o amplificador, assim, é possível verificar se o modelo matemático está representado fielmente e analisar as regiões de cobertura.

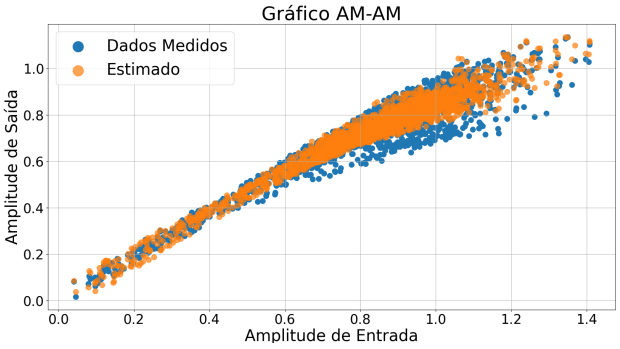

Figura 7: *Característica AM-AM do amplificador de potência*

A Figura 8 mostra a característica AM-PM (amplitude-fase) do amplificador, que representa a variação da fase do sinal de saída em função da amplitude do sinal de entrada. Esta análise é crucial para avaliar as distorções não lineares introduzidas pelo amplificador.

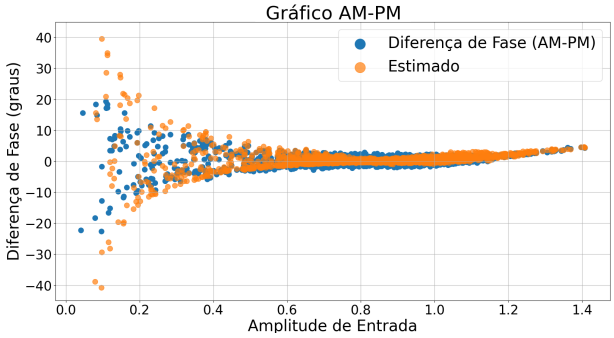

Figura 8: *Característica AM-PM do amplificador de potência*

A Figura 9 apresenta os resultados obtidos com o modelo polinomial usando números complexos, com grau P = 3 e memória M = 2. Este gráfico demonstra a capacidade do modelo em capturar as características não lineares do amplificador quando se utiliza otimização com números complexos.

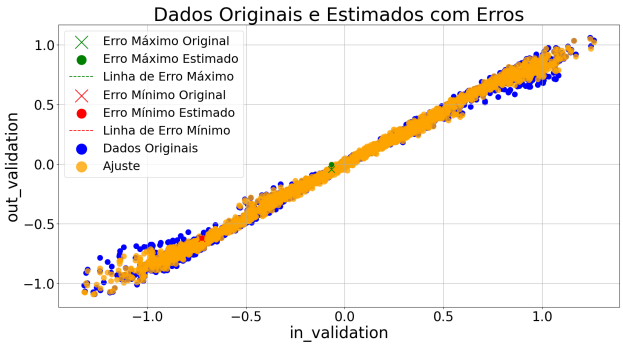

Figura 9: *Resultado do modelo polinomial com $P = 3$, $M = 2$ usando números complexos*

Para analisar o comportamento do erro em função dos parâmetros do modelo, foram gerados gráficos tridimensionais. A Figura 10 mostra a relação entre o erro máximo (eixo z), o grau do polinômio P e a profundidade de memória M. Esta visualização permite identificar qual combinação de P e M resulta no maior erro, auxiliando na seleção dos parâmetros ótimos do modelo.

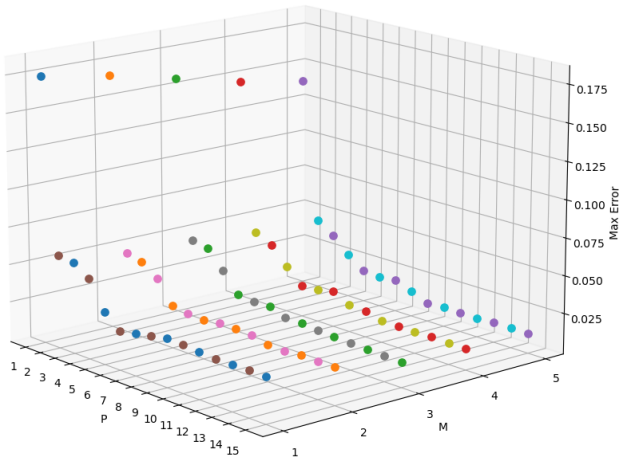

Figura 10: *Erro máximo em função do grau P e memória M*

A Figura 11 apresenta uma análise similar à anterior, porém considerando a implementação com números complexos. Esta abordagem permite uma representação mais precisa das características do amplificador, especialmente em relação às distorções de fase.

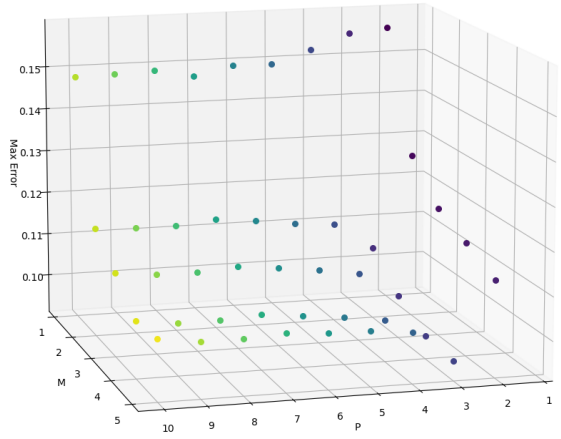

Figura 11: *Erro máximo para $P = 10$, $M = 5$ com implementação em números complexos*

Os resultados demonstram que o modelo polinomial com memória é eficaz na representação do comportamento não linear do amplificador de potência. A análise das características AM-AM e AM-PM confirma a capacidade do modelo em capturar tanto as distorções de amplitude quanto as de fase. A implementação com números complexos oferece maior precisão, especialmente para aplicações que requerem alta fidelidade na reprodução das características do amplificador.

V. CONCLUSÃO

Este trabalho demonstrou a eficácia dos modelos polinomiais com memória (MP) baseados nas séries de Volterra na modelagem comportamental de amplificadores de potência. A implementação de Lookup Tables (LUTs) reduziu significativamente a complexidade computacional, proporcionando convergência rápida e eficiente. A utilização de números complexos e estratégias de inicialização aleatória dos coeficientes mostraram-se fundamentais para a precisão e estabilidade do algoritmo de otimização.

Os testes experimentais com amplificador GaN classe AB confirmaram a capacidade do modelo baseado em séries de Volterra em reproduzir fielmente o comportamento real do dispositivo. As análises AM-AM e AM-PM validaram a precisão tanto para distorções de amplitude quanto de fase, oferecendo uma ferramenta robusta para predição do comportamento de amplificadores sem necessidade de testes laboratoriais diretos.

REFERÊNCIAS

[1] Elton John Bonfim. "Modelagem Comportamental de Amplificadores de Potência de Radiofrequência Usando Termos Unidimensionais e Bidimensionais de Séries de Volterra". Diss. de mestr. Curitiba: Universidade Federal do Paraná, 2016. URL: https://bdtd.ibict.br/vufind/Record/UFPR_54e3f81abd9cae4e13508c3e91186f38.

[2] Eduardo Gonçalves de Lima. "Behavioral modeling and digital base-band predistortion of RF power amplifiers". Tese de dout. POLITECNICO DI TORINO, jan. de 2009.

[3] Carolina Luiza Rizental Machado. "Modelagem comportamental de amplificadores de potência usando soma de produtos entre filtros digitais de resposta ao impulso finita e tabelas de busca unidimensionais". Diss. de mestr. Curitiba: Universidade Federal do Paraná, 2016. URL: https://acervodigital.ufpr.br/handle/1884/46250.

[4] MathWorks. *lsqnonlin: Solve nonlinear least-squares (nonlinear data-fitting) problems*. Acessado em: 1 abr. 2025. 2025. URL: https://www.mathworks.com/help/optim/ug/lsqnonlin.html.

[5] SciPy. *scipy.optimize.least_squares: Solve a nonlinear least-squares problem with bounds on the variables*. Acessado em: 20 abr. 2025. 2025. URL: https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.least_squares.html.

GitHub: