Universidade de Brasília – UnB

Campus UnB Gama: Faculdade de Ciências e Tecnologias em Engenharia – FCTE

Engenharia de Software

Implementação do Signalize, um visualizador de métodos de aproximações de sinais

Autor: Gabriel Costa de Oliveira e Thalisson Alves Gonçalves de Jesus

Orientador: Prof. Dr. Edson Alves da Costa Júnior

Brasília, DF

2025

Gabriel Costa de Oliveira e
Thalisson Alves Gonçalves de Jesus

Implementação do Signalize, um visualizador de métodos de aproximações de sinais

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Universidade de Brasília – UnB

Campus UnB Gama: Faculdade de Ciências e Tecnologias em Engenharia – FCTE

Orientador: Prof. Dr. Edson Alves da Costa Júnior

Brasília, DF
2025

Gabriel Costa de Oliveira e
Thalisson Alves Gonçalves de Jesus
Implementação do Signalize, um visualizador de métodos de aproximações de sinais/ Gabriel Costa de Oliveira e
Thalisson Alves Gonçalves de Jesus. – Brasília, DF, 2025-
57 p. : il. (algumas color.) ; 30 cm.

Orientador: Prof. Dr. Edson Alves da Costa Júnior

Trabalho de Conclusão de Curso – Universidade de Brasília – UnB
Campus UnB Gama: Faculdade de Ciências e Tecnologias em Engenharia – FCTE
, 2025.

1. métodos de aproximação. 2. sinais. I. Prof. Dr. Edson Alves da Costa Júnior. II. Universidade de Brasília. III. Faculdade UnB Gama. IV. Implementação do Signalize, um visualizador de métodos de aproximações de sinais

CDU 02:141:005.6

Gabriel Costa de Oliveira e
Thalisson Alves Gonçalves de Jesus

Implementação do Signalize, um visualizador de métodos de aproximações de sinais

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Trabalho aprovado. Brasília, DF, 22 de julho de 2025:

Prof. Dr. Edson Alves da Costa Júnior
Orientador

Prof. Dr. Luis Filomeno de Jesus Fernandes
Convidado 1

Prof. Dr. Renato Coral Sampaio
Convidado 2

Brasília, DF
2025

Agradecimentos

Eu, Thalisson Alves, agradeço à minha família por todo o suporte durante a minha trajetória acadêmica. Em especial, agradeço aos meus pais, Heloisa Alves e Carlos Roberto, e à minha irmã, Thais Alves, pelo incentivo e por proporcionarem as condições necessárias para que eu pudesse concluir a minha graduação. Agradeço a todos os professores que fizeram parte da minha formação, em especial aos Profs. Drs. Edson Alves da Costa Júnior e Bruno César Ribas, que me apresentaram à programação competitiva, uma experiência que ampliou meus horizontes e impulsionou meu crescimento pessoal e profissional. Agradeço à banca examinadora, composta pelos Profs. Drs. Luis Filomeno de Jesus Fernandes e Renato Coral Sampaio, pela disponibilidade e pelas sugestões que contribuíram para tornar este trabalho ainda melhor. Por fim, agradeço aos meus colegas de curso por me acompanharem nessa jornada, compartilhando experiências, conhecimentos e momentos de descontração.

Eu, Gabriel Costa, agradeço aos meus pais, Rita Paula e Josué de Oliveira; a todos os meus professores e professoras; aos meus amigos que fiz durante a minha graduação; e ao Brasil, por ter me proporcionado uma educação gratuita e de qualidade.

Resumo

Este trabalho descreve o desenvolvimento de uma aplicação *desktop* para a análise comparativa de métodos de aproximação de sinais. Sua principal característica é uma arquitetura extensível baseada em *plugins*, que permite a integração de novos algoritmos sem a necessidade de recompilar o sistema. A plataforma permite carregar sinais de arquivos CSV e visualizar simultaneamente o sinal original e múltiplas aproximações. A interface, desenvolvida em C++ com o *framework* Qt 6, integra métodos implementados em Python, como Prony, Matriz Pencil, Algoritmo de Realização de Autosistema e Identificação de Subespaço Estocástico. O usuário pode ajustar os parâmetros de cada método e personalizar a visualização dos gráficos, alterando cores, estilos e visibilidade das curvas. A ferramenta gera relatórios com métricas de erro, como o Erro Médio Absoluto (MAE) e o Erro Quadrático Médio (MSE), e apresenta um gráfico com os autovalores associados a cada modelo. O resultado é uma plataforma integrada e flexível para a análise e comparação de desempenho de algoritmos de aproximação de sinais.

Palavras-chave: visualizador. métodos de aproximação. sinais. análise de sinais. interpolação. C++. Python. Qt6. plugin.

Abstract

This work describes the development of a desktop application for the comparative analysis of signal approximation methods. Its main feature is an extensible plugin-based architecture, which allows the integration of new algorithms without recompiling the system. The platform can load signals from CSV files and simultaneously visualize the original signal alongside multiple approximations. The interface, developed in C++ with the Qt 6 framework, integrates methods implemented in Python, such as the Prony method, Matrix Pencil, Eigensystem Realization Algorithm (ERA), and Stochastic Subspace Identification. The user can adjust the parameters of each method and customize the plot visualization by changing colors, line styles, and curve visibility. The tool generates reports with error metrics, such as Mean Absolute Error (MAE) and Mean Squared Error (MSE), and displays a plot of the eigenvalues associated with each model. The result is an integrated and flexible platform for the analysis and performance comparison of signal approximation algorithms.

Key-words: visualizer. approximation methods. signals. signal analysis. interpolation. C++. Python. Qt6.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

CSV	*Comma-separated values*
SVD	*Singular value decomposition*
PDF	*Portable Document Format*
GUI	*Graphical User Interface*
UnB	Universidade de Brasília
FCTE	Faculdade de Ciências e Tecnologias em Engenharia
MPM	*Matrix Pencil Method*
SVD	*Singular value decomposition*
ERA	*Eigensystem Realization Algorithm*
SSI	*Stochastic Subspace Identification*
MAE	*Mean Average Error*
MSE	*Mean Square Error*

Lista de símbolos

λ	Letra grega minúscula lambda; Fator de amortecimento; Autovalor
Λ	Letra grega maiúscula lambda; Matriz diagonal dos autovalores
θ	Letra grega minúscula theta; Ângulo de fase
π	Letra grega minúscula pi; Constante
Σ	Letra grega maiúscula sigma; Somatório; Matriz diagonal dos valores singulares
σ	Letra grega minúscula sigma; Valor singular; Constante
α	Letra grega minúscula alfa
ω	Letra grega minúscula omega; Frequência angular
ϵ	Letra grega minúscula épsilon; tolerância
$\mathbb{R}$	Conjunto dos números reais
$\mathbb{C}$	Conjunto dos números complexos

Sumário

1 Introdução

1.1 Contexto

A aproximação de curvas é uma técnica fundamental em teoria de sinais e sistemas, com aplicações em matemática aplicada, engenharias e computação científica (COSTA; ANDRADE; FERREIRA, 2020). Seu objetivo é ajustar funções matemáticas a conjuntos de dados, estimando parâmetros que minimizem o erro em relação ao sinal original. A eficácia dessa abordagem depende tanto da seleção adequada do modelo matemático quanto de uma estimativa adequada dos parâmetros do modelo.

A visualização das curvas, dos parâmetros e das medidas de erro gerados por esses algoritmos consiste em um importante vetor que norteia o ensino e a pesquisa. Existem diversos softwares que têm suporte para essa finalidade, porém a exigência de conhecimento em programação e em matemática, e as restrições de licença de software, consistem uma importante barreiras para o ensino e pesquisa em processamento de sinais (COLOMBET et al., 2015).

A programação consiste na ferramenta mais poderosa para a visualização das curvas aproximadas por métodos de processamento de sinal. Ferramentas como NumPy e Matplotlib permitem o processamento de dados e a geração de gráficos complexos (COHEN, 2022), porém o rigor técnico exigido pela programação e as lacunas em matemática podem criar uma barreira na visualização das informações geradas pelo modelo.

Essa dificuldade é agravada por deficiências na formação matemática de muitos estudantes de engenharia, que enfrentam desafios na compreensão de conceitos fundamentais, como Cálculo e Álgebra Linear. Tais disciplinas são essenciais para as técnicas de processamento de sinais, e a falta de domínio desses tópicos pode levar à desmotivação, e até à evasão, nos cursos de engenharia (CARVALHO, 2022).

Embora já existam diversas ferramentas disponíveis, a ausência de uma que seja adequada à realidade dos estudantes do Campus UnB Gama: Faculdade de Ciência e Tecnologia em Engenharia (FCTE) e que seja *open source*, livre e extensível a projetos, como trabalho de conclusão de curso (TCC) e programa institucional de bolsas de iniciação científica (PIBIC), motivou a realização desse trabalho.

1.2 Objetivos

Este trabalho propõe o desenvolvimento do **Signalyze**, uma ferramenta gráfica para a visualização da aproximação numérica de curvas. Para atingir este objetivo, foram

estabelecidos os seguintes objetivos específicos:

(a) implementar uma interface interativa que integre múltiplos métodos de aproximação;

(b) permitir que as informações dos sinais possam ser carregadas por meio de arquivos CSV;

(c) permitir a visualização simultânea de várias aproximações para um sinal;

(d) desenvolver uma arquitetura extensível baseada em *plugins*;

(e) implementar a visualização dos autovalores associados aos modelos de aproximação.

1.3 Estrutura do Trabalho

Este trabalho está dividido em cinco capítulos, incluindo o presente capítulo. No Referencial Teórico apresentamos conceitos elementares sobre sinais, métodos de aproximação e interfaces gráficas. Em Materiais e Métodos descrevemos os procedimentos metodológicos adotados. Nos Resultados demonstramos as funcionalidades implementadas e os desafios encontrados. Por fim, nas Considerações Finais avaliamos o trabalho realizado e indicamos possíveis trabalhos futuros.

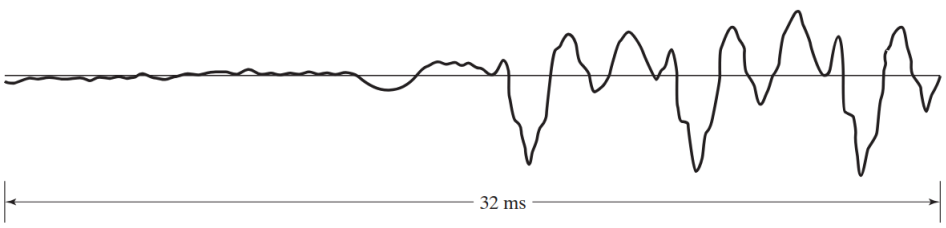
2 Referencial Teórico

Este capítulo apresenta os fundamentos teóricos que sustentam o desenvolvimento do presente trabalho, abordando conceitos elementares de processamento de sinais e métodos de aproximação, além de introduzir os conceitos básicos de interfaces gráficas.

2.1 Conceitos elementares de processamento de sinais

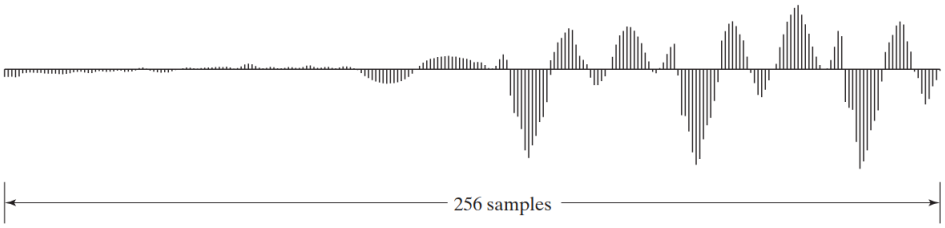
Sinais são representações matemáticas de grandezas físicas que variam em função do tempo, como a temperatura de um ambiente ou a intensidade de um sinal de áudio. Eles podem ser classificados em contínuos e discretos. Sinais contínuos são aqueles que assumem valores em qualquer instante de tempo, como a temperatura de um ambiente que varia suavemente. Por outro lado, os sinais discretos são aqueles que assumem valores apenas em instantes de tempo específicos, como a temperatura medida por um termômetro a cada segundo, Podemos ver um exemplos de um sinal contínuos e discretos, respectivamente nas Figuras 1 e 2 (OPPENHEIM; SCHAFER, 2010).

Figura 1 – Representação de um sinal contínuo.



Fonte: Oppenheim e Schafer (2010)

Figura 2 – Representação de um sinal discreto.



Fonte: Oppenheim e Schafer (2010)

A conversão de um sinal contínuo para um sinal discreto é realizada através de um processo chamado amostragem, que consiste em medir o valor do sinal em intervalos regulares de tempo (ROBERTS, 2008). A amostragem é crucial para o processamento digital de sinais, já que computadores trabalham com dados discretos.

O resultado da amostragem é uma sequência de valores que representam o sinal original em pontos discretos no tempo. Essa sequência de valores precisa ser cuidadosamente coletada para evitar a perda de informações importantes do sinal original. A frequência da amostragem, que representa o número de amostras coletadas por unidade de tempo, é um fator crucial nesse processo. Uma frequência de amostragem muito baixa pode resultar na perda de informações importantes do sinal, enquanto uma frequência muito alta pode levar a necessidade de uma grande área de armazenamento e a um tempo de processamento demasiadamente longo (ROBERTS, 2008).

Fundamental para o processamento de sinais, a teoria da amostragem estabelece as condições necessárias para uma reconstrução perfeita do sinal original a partir de suas amostras discretas. Formalmente, o teorema afirma que um sinal contínuo, limitado em banda, com frequência máxima B Hz, pode ser reconstruído integralmente desde que a taxa de amostragem f_s seja pelo menos o dobro de B, ou seja, $f_s \geq 2B$. Essa frequência crítica, $2B$, é denominada taxa de Nyquist. Se essa condição não for atendida, ocorre o fenômeno de *aliasing*, em que componentes de alta frequência do sinal original se sobrepõem às componentes de baixa frequência no sinal amostrado, distorcendo irreversivelmente a informação original (PROAKIS, 1995).

Uma exponencial amortecida é uma função que descreve um sinal que decai exponencialmente com o tempo. Essa exponencial é caracterizada por parâmetros que definem seu comportamento: amplitude, frequência, coeficiente de amortecimento e fase. A amplitude (A) representa o valor máximo do sinal; a frequência (f) determina a periodicidade das oscilações; o coeficiente de amortecimento (λ) indica a taxa de decaimento exponencial (se $\lambda < 0$, o sinal decai; se $\lambda > 0$, o sinal cresce); e a fase (θ) define o deslocamento temporal inicial das oscilações. A Figura 3 exemplifica uma destas exponenciais amortecidas. Nas engenharias, a representação de sinais através de exponenciais complexas amortecidas compõe uma importante ferramenta para a análise e o processamento de sinais.

Um número complexo pode ser representado na sua forma retangular por dois números reais, que compõem a parte real e imaginária, respectivamente, conforme mostra a Equação 2.1.

$$w = a + bj, \quad a, b \in \mathbb{R}, \quad j^2 = -1 \tag{2.1}$$

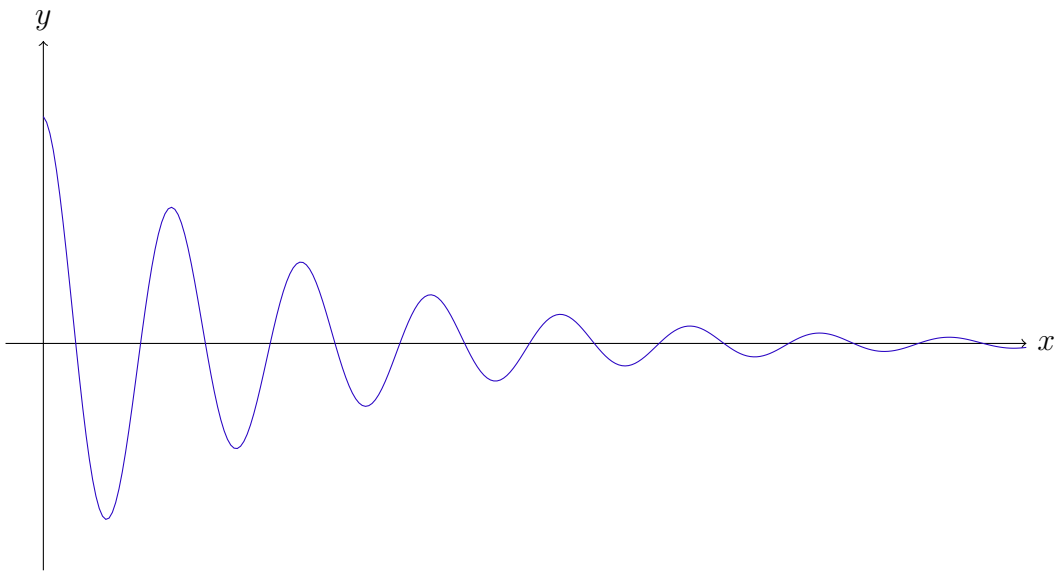
Exponenciais complexas estão diretamente relacionadas com funções trigonométricas através da equação de Euler, apresentada na Equação 2.2 (STEIN; SHAKARCHI, 2003).

$$e^{ja} = \cos(a) + j\sin(a) \tag{2.2}$$

Essa relação permite representar números complexos em sua forma polar, combinando módulo $|z| = r$ e fase $\angle z = \theta$, conforme mostra a Equação 2.3 e Figura 4 (KREYSZIG, 2011).

$$z = e^{a+jb} = e^a(\cos b + j\sin b) = r(\cos\theta + j\sin\theta) = r\angle\theta \tag{2.3}$$

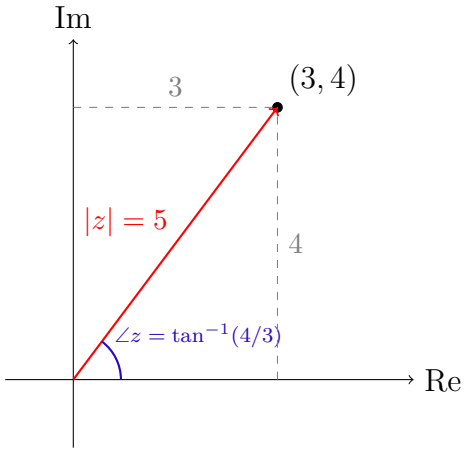
Figura 3 – Visualização da função contínua $y(t) = 3e^{-0.3t}\cos(0.58t)$



Fonte: os Autores

Essa relação permite representar números complexos em sua forma polar, combinando módulo $|z| = r$ e fase $\angle z = \theta$, conforme mostra a Figura 4.

Figura 4 – Representação do número complexo $z = 3 + 4j$ e $z = 5e^{j\tan^{-1}(4/3)}$, nas formas retangular e polar, respectivamente.



Fonte: os Autores

De forma trivial, pode-se a partir da Equação 2.2 deduzir as Equações 2.4 e 2.5.

$$\cos(a) = \frac{e^{ja} + e^{-ja}}{2} \tag{2.4}$$

$$\sin(a) = \frac{e^{ja} - e^{-ja}}{2j} \tag{2.5}$$

2.2 Conceitos elementares de Álgebra Linear

Seja $L : V \to W$ uma transformação linear entre dois espaços vetoriais. O núcleo ou espaço nulo de L é o conjunto definido pela Equação 2.6 (LEON, 2018).

$$\ker(L) = \{v \in V \mid L(v) = 0\} \tag{2.6}$$

Se representarmos uma transformação linear por meio de uma matriz real quadrada A_{nxn}, então o núcleo desta transformação é definido por um sistema homogêneo, conforme mostra a Equação 2.7 (BOLDRINI, 1984).

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & x_2 & \cdots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_1 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \tag{2.7}$$

O sistema homogêneo descrito na Equação 2.7 pode ser simplificado pela expressão apresentada na Equação 2.8.

$$Ax = 0 \tag{2.8}$$

Se a matriz A é invertível, o sistema só admite como resposta o vetor nulo; caso contrário, se A for singular, o conjunto de resposta será infinito (ANTON; RORRES, 2011).

Se $T(v) = \lambda v$ então v é um autovetor de T e λ e seu autovalor correspondente. Em outras palavras, se multiplicarmos $A_{n\times n}$ por $v_{n\times 1}$ o vetor resultante conservará a direção de v mudando, possivelmente, o seu tamanho e/ou sentido, conforme mostra a Equação 2.9. Além disso, matrizes reais eventualmente podem ter autovalores complexos, especialmente em matrizes assimétricas (STRANG, 2006).

$$Av = \lambda v \quad \lambda \in \mathbb{C} \tag{2.9}$$

Manipulando a Equação 2.9 podemos encontrar um sistema de equações homogêneas, conforme mostra a Equação 2.10.

$$(A - \lambda I)v = 0 \tag{2.10}$$

Conforme discutimos anteriormente, esse sistema homogêneo terá resultados além do trivial se, e somente se, a matriz $A - \lambda I$ for singular, ou seja, se $\det(A - \lambda I) = 0$. Essa condição define o polinômio característico de A, cujas raízes são os autovalores da transformação linear, conforme mostra a Equação 2.11 (GOLUB; LOAN, 2013).

$$\det(A - \lambda I) = (\lambda - \lambda_1)(\lambda - \lambda_2) \cdots (\lambda - \lambda_n) = 0 \tag{2.11}$$

Seja v_i o autovetor associado ao autovalor λ_i. Quando A é diagonalizável, isto é, possui n autovetores linearmente independentes, podemos escrever a decomposição espectral de A na forma apresentada na Equação 2.12.

$$AV = A\begin{bmatrix} v_1 & \dots & v_n \end{bmatrix} = \begin{bmatrix} Av_1 & \dots & Av_n \end{bmatrix} = \begin{bmatrix} \lambda_i v_1 & \dots & \lambda_n v_n \end{bmatrix} \tag{2.12}$$

Reescrevendo a Equação 2.12 e multiplicando ambos lados da equação à esquerda por V^{-1} obtemos a Equação 2.13 (LEON, 2018).

$$A = V\Lambda V^{-1}, \quad \Lambda = \begin{bmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \lambda_n \end{bmatrix} \tag{2.13}$$

No processamento de sinais, os autovalores estão associados a exponenciais complexas amortecidas da forma $e^{\lambda t}$. Conforme mostra a Equação 2.3, a parte real de λ está relacionada ao invólucro da função. Assim, temos que $\lim_{t\to\infty} e^{\lambda t} = 0$ se $\mathrm{R}\{y\} < 0$, indicando que a exponencial é estável. Por outro lado, se $\mathrm{R}\{y\} > 0$, então $\lim_{t\to\infty} e^{\lambda t} = \infty$, indicando que a exponencial é instável.

Uma matriz $B \in \mathbb{R}^{n\times n}$ é definida positiva se for simétrica ($B = B^T$) e possuir todos os autovalores estritamente positivos e reais. Além disso, como B é uma matriz simétrica, podemos escolher, na sua decomposição espectral, autovetores ortonormais, conforme mostra a Equação 2.14 (STRANG, 2006).

$$B = Q\Lambda Q^T, \quad \text{onde} \quad Q^T Q = I \tag{2.14}$$

A decomposição em valores singulares está intimamente associada à decomposição espectral de matrizes definidas positivas. Uma vez que, para qualquer matriz $A \in \mathbb{R}^{n\times m}$, as matrizes $A^T A$ e AA^T são definidas positivas e possuem todos os autovalores não negativos iguais, como pode-se observar nas Equações 2.16 e 2.15 (STRANG, 2006).

$$AA^T = U\Sigma\Sigma^T U^T, \quad U^T U = I \tag{2.15}$$

$$A^T A = V\Sigma\Sigma^T V^T, \quad V^T V = I \tag{2.16}$$

A decomposição em valores singulares de A sintetiza as decomposições espectrais mostradas nas Equações 2.15 e 2.16, conforme mostra a Equação 2.38.

$$A = U\Sigma V^T \tag{2.17}$$

A decomposição em valores singulares é fundamental para o processamento de sinais, pois ela é aplicável em matrizes retangulares. Duas de suas principais aplicações

são aproximação de posto baixo $\tilde{\mathrm{A}}$ e matriz inversa de Moore-Penrose $A^{\dagger}$. A matriz $\tilde{\mathrm{A}}$ pode ser construída com os k maiores valores singulares, conforme mostra a Equação 2.18 (GOLUB; HOFFMAN; STEWART, 1987).

$$\tilde{\mathrm{A}} = U\tilde{\Lambda}V^T = \sum_{i=1}^{k} \sigma_i u_i v_i^T \tag{2.18}$$

A matriz inversa de Moore-Penrose $A^{\dagger}$ fornece a solução ótima de mínimos quadrados para sistemas lineares e pode ser calculada por meio da decomposição de valores singulares, conforme mostra a Equação 2.19 (GOLUB; LOAN, 2013).

$$A^{\dagger} = U\Sigma^{-1}V^T \tag{2.19}$$

2.3 Método de Prony

O Método de Prony foi desenvolvido no século XVIII pelo matemático francês Gaspard de Prony (COSTA, 2014) tem o objetivo de encontrar uma aproximação para a Equação 2.20

$$y(t) = \sum_{i=1}^{d} A_i e^{\lambda_i t} \cos(2\pi f_i t + \theta_i), \tag{2.20}$$

onde $A_i, \lambda_i, f_i, \theta_i$ são a amplitude, coeficiente de amortecimento, frequência e fase da i-ésima componente, com $i = 1, 2, \ldots, d$.

Quando os sinais são sincronizados e discretos, podemos expressar a série de amostras y_k do sinal pro meio da Equação 2.21 (COSTA, 2014; OPPENHEIM; SCHAFER, 2010),

$$y_k = \sum_{i=1}^{d} R_i z_i^{kT}, \tag{2.21}$$

onde $R_i = \frac{1}{2}A_i e^{j\theta_i}$ e $z_i = e^{(\lambda_i + j2\pi f_i)T}$. Reescrevendo a Equação 2.21 na forma matricial, obtemos a Equação 2.22.

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{N-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 & \cdots & 1 \\ z_1 & z_2 & \cdots & z_d \\ z_1^2 & z_2^2 & \cdots & z_d^2 \\ \vdots & \vdots & \ddots & \vdots \\ z_1^{N-1} & z_2^{N-1} & \cdots & z_d^{N-1} \end{bmatrix} \begin{bmatrix} R_1 \\ R_2 \\ \vdots \\ R_d \end{bmatrix} \tag{2.22}$$

Para determinar os valores dos termo z_i, observamos que eles são raízes do polinômio característico $P(q)$ associado à Equação 2.23, onde os coeficientes c_i são desconhecidos.

$$P(z) = z^d - c_1 z^{d-1} - \cdots - c_{d-1} z^1 - c_d = z^d - \sum_{i=1}^{d} c_i z^{d-i} \tag{2.23}$$

Multiplicando a Equação 2.22 pelo vetor

$$[-c_d, -c_{d-1}, \ldots, -c_1, 1, 0, \ldots, 0]$$

obtemos a Equação 2.24 no lado esquerdo,

$$\begin{bmatrix} -c_d & -c_{d-1} & \cdots & -c_1 & 1 & 0 & \cdots & 0 \end{bmatrix} \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{N-1} \end{bmatrix} \tag{2.24}$$

e obtemos a Equação 2.25 no lado direito 2.25.

$$\begin{bmatrix} -c_d & -c_{d-1} & \cdots & -c_1 & 1 & 0 & \cdots & 0 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ z_1 & z_2 & z_3 & \cdots & z_d \\ z_1^2 & z_2^2 & z_3^2 & \cdots & z_d^2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ z_1^{N-1} & z_2^{N-1} & z_3^{N-1} & \cdots & z_d^{N-1} \end{bmatrix} \begin{bmatrix} R_1 \\ R_2 \\ \vdots \\ R_d \end{bmatrix} \tag{2.25}$$

Note que o resultado da Equação 2.25 será a matriz nula $0_{1\times d}$, pois a i–ésima coluna da matriz será igual ao polinômio característico aplicado em z_i, conforme mostra a Equação 2.26.

$$\begin{bmatrix} -c_d & \cdots & -c_1 & 1 & \cdots & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ z_i \\ z_i^2 \\ z_i^{N-1} \end{bmatrix} = P(z_i) = 0 \tag{2.26}$$

Desenvolvendo a Equação 2.24 e igualando ao resultado obtido na Equação 2.26, obtemos a Equação 2.27

$$-c_d y_0 - c_{d-1} y_1 - \cdots - c_1 y_{d-1} + y_d = 0, \tag{2.27}$$

a qual compõe o núcleo do método de Prony, pois se multiplicamos novamente a Equação 2.22 pelo deslocamento (*shift*) do vetor de coeficientes, obtemos a matriz nula novamente. Isso pode ser observado desenvolvendo a Equação 2.23 e usando a Equação 2.28.

$$P(z_i) = z_i^d - \sum_{m=1}^{d} c_m z_i^{d-m} = 0 \Rightarrow z_i^d = \sum_{i}^{d} c_m z_i^{d-m} \tag{2.28}$$

Multiplicando ambos os lados por z_i^k obtemos a Equação 2.29.

$$z_i^{k+d} = \sum_{m=i}^{d} c_m z_i^{k+d-m} \tag{2.29}$$

A Equação 2.29 implica que qualquer potência z_i^k pode ser escrita com a soma das d parcelas anteriores. Esse fato nos permite escrever a Equação 2.30.

$$\begin{bmatrix} y_d \\ y_{d+1} \\ \vdots \\ y_{N-1} \end{bmatrix} = \begin{bmatrix} y_{d-1} & y_{d-2} & \cdots & y_0 \\ y_d & y_{d-1} & \cdots & y_1 \\ \vdots & \vdots & \ddots & \vdots \\ y_{N-2} & y_{N-3} & \cdots & y_{N-d-1} \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_d \end{bmatrix} \tag{2.30}$$

Segundo Costa (2014), o método de Prony pode ser resumido nas seguintes etapas:

a) calcule os coeficientes c_i resolvendo o sistema linear da Equação 2.30;

b) encontre as raízes z_i do polinômio característico da Equação 2.23;

c) calcule os coeficientes R_i solucionando a Equação 2.22;

d) obter as variáveis físicas a partir dos números complexos z_i e R_i com as Equações 2.31, 2.32 e 2.33.

$$\lambda_i = \frac{\mathrm{Re}\{\ln(z_i)\}}{T} \tag{2.31}$$

$$f_i = \frac{\mathrm{Imag}\{\ln(z_i)\}}{2\pi T} \tag{2.32}$$

$$\theta_i = \arctan\left(\frac{\mathrm{imag}\{R_i\}}{\mathrm{Re}\{R_i\}}\right) \tag{2.33}$$

2.4 Método Matrix Pencil

O Método Matrix Pencil foi desenvolvido por Hua e Sarkar em 1990 (SARKAR; PEREIRA, 1995). O método utiliza o modelo de sinal apresentado na Equação 2.21, onde y_k é expresso como uma combinação de exponenciais complexas. Assim como no método de Prony, o objetivo é determinar os parâmetros z_i e R_i, porém através de uma abordagem baseada em autovalores.

Sejam as matrizes de Hankel (RANE; PANDEY; KAZI, 2021) Y_1 e Y_2, construídas a partir das amostras, conforme mostram as Equações 2.34 e 2.35,

$$Y_1 = \begin{bmatrix} y_0 & y_1 & \cdots & y_{L-1} \\ y_1 & y_2 & \cdots & y_L \\ \vdots & \vdots & \ddots & \vdots \\ y_{N-L-1} & y_{N-L} & \cdots & y_{N-2} \end{bmatrix} \tag{2.34}$$

$$Y_2 = \begin{bmatrix} y_1 & y_2 & \cdots & y_L \\ y_2 & y_3 & \cdots & y_{L+1} \\ \vdots & \vdots & \ddots & \vdots \\ y_{N-L} & y_{N-L+1} & \cdots & y_{N-1} \end{bmatrix} \tag{2.35}$$

onde L é o parâmetro *pencil* (geralmente $L \approx N/3$). A matrix Pencil X é definida como a combinação linear, conforme mostra a Equação, 2.36

$$X = Y_2 - \lambda Y_1 \tag{2.36}$$

e a matriz aumentada Y é definida na Equação 2.37.

$$Y = [Y_1|Y_2] \tag{2.37}$$

A decomposição em valores singulares (SVD) (MEYER, 2000) de Y é dada pela Equação 2.38,

$$Y = USV^* \tag{2.38}$$

onde U e V são matrizes unitárias, S é uma matriz diagonal com os valores singulares em ordem decrescente e V^* é a transposta conjugada de V.

A ordem do sistema d é determinada pelo número de valores singulares $\alpha > \alpha_{\max} \cdot \epsilon$, onde $\alpha_{\max}$ é o maior valor singular e ϵ é uma tolerância (tipicamente 10^{-3}). As colunas correspondentes de V são usadas para construir a matriz filtrada (RANE; PANDEY; KAZI, 2021), conforme apresenta a Equação 2.39.

$$V_{\text{filtrada}} = [V_1 \; V_2 \; \cdots \; V_k] \tag{2.39}$$

As matrizes B_1 e B_2 são construídas a partir de V_{filtrada}, onde B_1 elimina a última linha e B_2 elimina a primeira linha de V_{filtrada}.

Os autovalores z_i são calculados a partir da Equação 2.40,

$$z_i = \text{eig}(B_1^\dagger B_2) \tag{2.40}$$

onde $B_1^\dagger$ é a pseudo-inversa de Moore-Penrose de B_1 (BEN-ISRAEL; GREVILLE, 2003). Uma vez determinados os z_i, os coeficientes R_i são obtidos resolvendo o sistema linear da Equação 2.22 (SARKAR; PEREIRA, 1995).

O Método Matrix Pencil pode ser resumido nas seguintes etapas:

a) construa as matrizes de Hankel Y_1 e Y_2 de acordo com as Equações (2.34) e (2.35);

b) use a decomposição de valores singulares para tratamento de ruído;

c) resolva o problema de autovalores $B_1^\dagger B_2$ para obter z_i, (Equação 2.40);

d) calcule os coeficientes R_i solucionando a Equação 2.22;

e) obtenha as variáveis físicas a partir dos números complexos z_i e R_i usando as Equações 2.31, 2.32 e 2.33.

2.5 Algoritmo de Realização de Autosistemas

O algoritmo de realização de autosistemas ou *Eigensystem Realization Algorithm* (ERA) é um algoritmo que foi desenvolvido por Juang e Pappa (ALMUNIF; FAN; MIAO, 2020). O método analisa dados de vibrações amortecidas gerados por um impulso em estruturas flexíveis. Como nos outros métodos, ele também fornece uma aproximação para a Equação 2.21.

O ERA considera o sistema linear invariante no tempo no domínio discreto, conforme mostram as Equações 2.41 e 2.42.

$$x_{k+1} = Ax_k + Bu_k, \quad A \in \mathbb{R}^{n\times n}, B \in \mathbb{R}^{n\times m} \tag{2.41}$$

$$y_k = Cx_k + Du_k, \quad C \in \mathbb{R}^{p\times n}, D \in \mathbb{R}^{p\times m} \tag{2.42}$$

Assumindo condições iniciais nulas ($x_0 = 0$) e entrada do tipo impulso ($u_0 = 1$, $u_k = 0$ para $k \geq 1$), obtemos a Equação 2.43,

$$x = \begin{bmatrix} 0 \\ B \\ AB \\ \vdots \\ A^{k-1}B \end{bmatrix}, \quad y = \begin{bmatrix} D \\ CB \\ CAB \\ \vdots \\ CA^{k-1]}B \end{bmatrix} \tag{2.43}$$

Pode ser demonstrado que a matriz Hankel (Equação 2.34) pode ser decomposta na forma apresentada na Equação 2.44,

$$Y_1 = \begin{bmatrix} CB & CAB & \dots & CA^{l-1}B \\ CAB & CA^2B & \dots & CA^LB \\ \vdots & \vdots & \ddots & \vdots \\ CA^{N-L}B & CA^{N-L+1}B & \dots & CA^{n-1}B \end{bmatrix} = \begin{bmatrix} C \\ CA \\ \dots \\ CA^{N-L} \end{bmatrix} \begin{bmatrix} B & AB & \dots & A^{L-1}B \end{bmatrix} = \mathcal{O}C \tag{2.44}$$

$$Y_2 = \mathcal{O}A\mathcal{C} \tag{2.45}$$

onde $\mathcal{O}$ é a matriz de observabilidade e $\mathcal{C}$ é a matriz de controlabilidade.

Aplicando a decomposição em valores singulares na Equação 2.34 podemos representar sua matriz de baixo posto (Equação 2.18), conforme mostra a Equação 2.46.

$$\tilde{Y}_1 = U\tilde{\Lambda}V^T \tag{2.46}$$

Uma redução de posto similar também é aplicada em Y_2 (Equação 2.35).

Os valores de $\mathcal{O}$, $\mathcal{C}$ e A podem ser determinados a partir das Equações 2.47, 2.48 e 2.49.

$$\mathcal{O} = U\tilde{\Lambda}^{\frac{1}{2}} \tag{2.47}$$

$$\mathcal{C} = \tilde{\Lambda}^{\frac{1}{2}}V^T \tag{2.48}$$

$$A = \tilde{\Lambda}U\tilde{Y_2}V\tilde{\Lambda}^{\frac{1}{2}} \tag{2.49}$$

Os valores de z da Equação 2.21 podem ser encontrados extraindo os autovalores da matriz A (Equação 2.49)

O algoritmo ERA pode ser sintetizado nas seguintes etapas:

a) construa as matrizes de Hankel Y_1 e Y_2 conforme a (Equações (2.34) e (2.35));

b) use a decomposição de valores singulares (SVD) para tratamento de ruído;

c) resolva o sistema de matrizes A descrito na Equação 2.49

d) obtenha as variáveis físicas a partir dos números complexos z_i e R_i usando as Equações 2.31, 2.32 e 2.33.

2.6 Identificação Estocástica de Subespaço

Inicialmente apresentado pro Van Overschee Moor. O *Stochastic Subspace Identification* (SSI) calcula o modelo em espaço de estados a partir dos dados de saída do sistema. Esse método é capaz de determinar as características dinâmicas de um sistema com base na entrada e saída de valores. Desta forma, é amplamente utilizado na engenharia, já que fornece uma solução suficientemente precisa para diversos problemas práticos. Embora seja possível modelar esses problemas por meio de equações diferenciais parciais, a abordagem fisicamente rigorosa, apesar de mais precisa, tende a ser mais complexa. (OVERSCHEE; MOOR, 1996)

s matrizes de Hankel para saídas passadas (Y_p) e futuras (Y_f) são definidas respectivamente conforme mostram as Equações 2.50 e 2.51. onde $q = \lfloor \frac{n}{3} \rfloor$ e $M = n - 2q - 1$

$$Y_p = \begin{bmatrix} y_0 & y_1 & \cdots & y_{M-1} \\ y_1 & y_2 & \cdots & y_M \\ \vdots & \vdots & \ddots & \vdots \\ y_{q-1} & y_q & \cdots & y_{q+M-2} \end{bmatrix} \tag{2.50}$$

$$Y_f = \begin{bmatrix} y_q & y_{q+1} & \cdots & y_{q+M-1} \\ y_{q+1} & y_{q+2} & \cdots & y_{q+M} \\ \vdots & \vdots & \ddots & \vdots \\ y_{q+p} & y_{q+p+1} & \cdots & y_{q+p+M-1} \end{bmatrix} \tag{2.51}$$

A projeção oblíqua é calculada conforme mostra a Equação 2.52

$$O = Y_f/Y_p^T = Y_f Y_p^T (Y_p Y_p^T)^{-1} Y_p \tag{2.52}$$

Aplicando a decomposição em valores singulares na Equação 2.52 podemos representar sua matriz de baixo posto (Equação 2.18), conforme mostra a Equação 2.46.

$$O = U\tilde{\Lambda}V^T \tag{2.53}$$

Usando a Equação 2.53 obtemos a matriz de observabilidade, conforme mostra a Equação 2.54

$$\mathcal{O} = U\tilde{\Lambda}^{\frac{1}{2}} \tag{2.54}$$

$\mathcal{O}_{up}$ e $\mathcal{O}_{down}$ é a matriz de observabilidade (Equação 2.52 removendo a última e a primeira linha, respectivamente.

$$A = \mathcal{O}_{up}^{\dagger}\mathcal{O}_{down} \tag{2.55}$$

Os valores de z da Equação 2.21 podem ser encontrados extraindo os autovalores da matriz A (Equação 2.55)

O algoritmo SSI pode ser sintetizado nas seguintes etapas:

a) construa as matrizes de Hankel Y_p e Y_f de acordo com as Equações (2.50) e (2.51);

b) obtenha a projeção oblíqua O conforme mostra a Equação 2.52;

c) use a decomposição de valores singulares na matriz O (Equação 2.53);

d) obtenha a matriz A (2.55) a partir da matriz de observabilidade O (Equação 2.52);

e) obtenha os autovalores z_i da matriz A (Equação 2.55)

f) obtenha as variáveis físicas a partir dos números complexos z_i usando as Equações 2.31, 2.32 e 2.33.

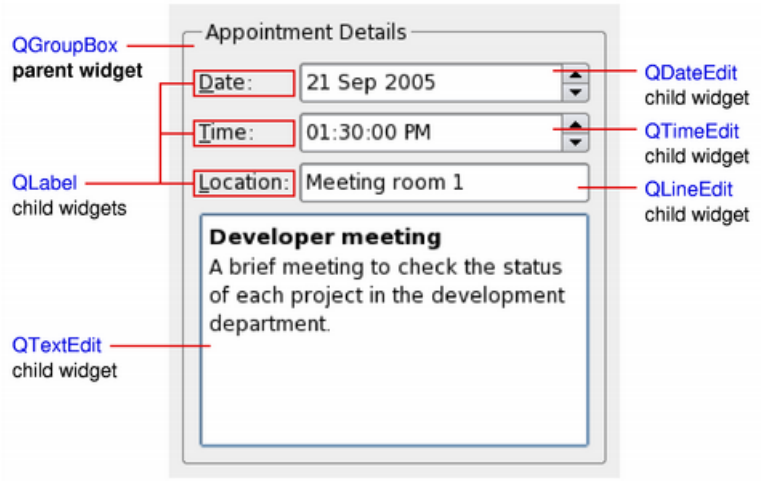
2.7 Interfaces Gráficas

As interfaces gráficas (*Graphical User Interfaces* – GUIs) tornam a interação entre usuários e softwares mais intuitiva e acessível. Diferente de interfaces baseadas em linha

de comando, as GUIs permitem a visualização e manipulação de informações por meio de elementos gráficos, como botões, menus e janelas, facilitando a navegação e o uso da aplicação (MARCUS, 1997).

A construção de uma GUI envolve a combinação de diversos elementos, sendo os *widgets* os blocos de construção básicos da interface (PéREZ; SIBERT, 1993). Cada *widget* representa um elemento visual, como botões, caixas de texto, menus e listas, que permite a interação do usuário com a aplicação. A organização desses *widgets* em uma estrutura visualmente agradável e funcional é realizada através de *layouts*, que definem o posicionamento e o tamanho dos elementos na tela. Para garantir a interatividade da interface, eventos são disparados quando o usuário interage com os *widgets*, como clicar em um botão ou digitar em uma caixa de texto. Esses eventos são tratados pelo código da aplicação, que executa as ações correspondentes. A Figura 5 exemplifica a estrutura de uma interface gráfica no `Qt6`, ilustrando a hierarquia entre *widgets* e a organização dos elementos pelo *layout*.

Figura 5 – Exemplo de interface com *widgets* no `Qt6`.



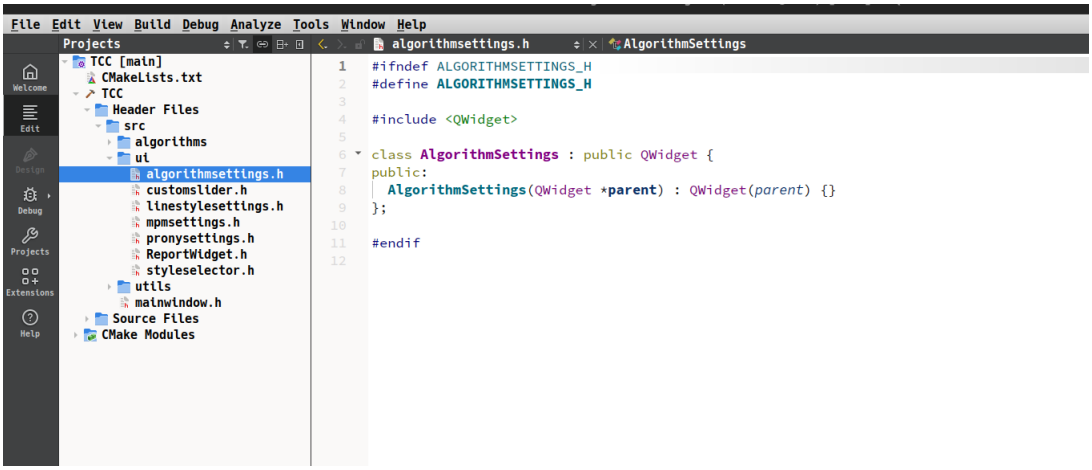
Fonte: Qt Documentation.

O *framework* Qt6 fornece um conjunto abrangente de ferramentas para o desenvolvimento de GUIs modernas, permitindo a construção de interfaces responsivas com suporte a eventos, gráficos interativos e *layouts* flexíveis (DEY, 2021). Além disso, seu suporte *cross-platform* garante compatibilidade com diferentes sistemas operacionais sem a necessidade de grandes adaptações no código.

O QtCreator é o ambiente de desenvolvimento oficial e figura como um excelente ponto de partida para criar um ambiente de desenvolvimento C++ para Qt (Figura 6). No entanto, isso não impede que o Qt seja desenvolvido em outras plataformas ou editores de texto, como o Neovim.

Para estender as funcionalidades do C++ padrão e permitir o uso de recursos

Figura 6 – Interface do QtCreator, mostrando o ambiente de desenvolvimento.



Fonte: Os Autores.

como *signals* e *slots*, o Qt6 emprega um pré-processador chamado *Meta-Object Compiler* – MOC. Este pré-processador analisa os arquivos-fonte C++ do projeto que utilizam a sintaxe especial do Qt (como *signals* e *slots* e a macro `Q_OBJECT`). Durante a compilação, o MOC gera código C++ adicional que implementa a infraestrutura de meta-objetos. Esse código permite a introspecção de objetos em tempo de execução, a conexão de *signals* e *slots* e o uso do sistema de propriedades do Qt, facilitando o desenvolvimento de aplicações reativas e modulares sem a complexidade manual das funções de *callback* (COMPANY, 2025).

A comunicação entre os objetos na arquitetura Qt é gerenciada por um sistema de *signals* e *slots*, que proporciona uma comunicação segura e com alto nível de desacoplamento entre componentes. Um *signal* é emitido por um objeto quando um evento específico ocorre, como o clique de um botão. Já um *slot* é uma função que é chamada em resposta a um *signal*. Diferente das funções de *callback* tradicionais, que exigem conhecimento explícito entre as partes que se comunicam, o sistema de *signals* e *slots* permite que um objeto emita um *signal* sem saber quais *slots* estão conectados a ele, e um *slot* pode ser acionado sem saber qual *signal* foi emitido. Essa abordagem garante a compatibilidade dos tipos dos argumentos esperados entre os objetos, mantendo um baixo acoplamento entre eles.

2.8 Bibliotecas Dinâmicas

A extensibilidade e a modularidade são cruciais para softwares complexos, pois permitem adicionar e atualizar funcionalidades sem a necessidade de alterar a aplicação principal. Uma forma comum de alcançar esses objetivos é através de uma arquitetura de *plugins* (ou módulos), que possibilita o carregamento dinâmico de componentes de

software pela aplicação em tempo de execução. Neste trabalho, essa arquitetura é utilizada para incorporar diferentes métodos de interpolação como módulos independentes.

A base para a implementação de uma arquitetura de *plugins* está no conceito de bibliotecas dinâmicas, também conhecidas como bibliotecas compartilhadas (*shared libraries*). Estas, representadas por arquivos com extensões `.so` em sistemas Unix-like e `.dll` em sistemas Windows, consistem em código e dados que podem ser carregados na memória e utilizados por múltiplos processos simultaneamente em tempo de execução (ZHIRKOV, 2017a). A utilização de bibliotecas dinâmicas oferece algumas vantagens em relação às bibliotecas estáticas, incluindo a redução do tamanho do executável principal, compartilhamento de código em memória e a capacidade de atualizar ou substituir módulos individuais sem recompilar a aplicação como um todo. O processo de vinculação (*linking*) das bibliotecas dinâmicas ocorre durante a execução do programa, onde o sistema operacional resolve os símbolos (funções e variáveis) exportados pela biblioteca e os conecta às chamadas correspondentes na aplicação.

Um aspecto fundamental ao trabalhar com *plugins* e bibliotecas dinâmicas em C++ é o gerenciamento de memória. Cada processo de software possui seu próprio espaço de endereço virtual, uma abstração que fornece ao programa uma visão isolada da memória do sistema (ZHIRKOV, 2017b). Quando um *plugin* (carregado como uma biblioteca dinâmica) é utilizado pela aplicação principal, ele opera dentro do mesmo espaço de endereço do processo hospedeiro, porém com seu próprio gerenciador de memória (*heap*). Dessa forma, a alocação de memória realizada dentro do contexto do *plugin* deve ser correspondente à sua desalocação também dentro desse contexto. Tentar desalocar memória alocada em um *plugin* utilizando o gerenciador de memória da aplicação principal (ou vice-versa) pode levar a comportamentos indefinidos, como a corrupção da memória e falhas na execução. Uma solução para este problema envolve a definição de interfaces bem estabelecidas entre a aplicação e os *plugins*. Estas interfaces podem incluir métodos específicos para a alocação e liberação de memória dos objetos gerenciados pelo *plugin*, garantindo que o ciclo de vida da memória seja consistentemente controlado dentro do módulo responsável por sua alocação.

Outro conceito relevante no contexto da interoperabilidade entre a aplicação e os *plugins* implementados em C++ é o *name mangling* (ou decoração de nomes). O compilador C++ altera os nomes das funções e variáveis durante o processo de compilação, incorporando informações sobre seus tipos, *namespaces* e outros atributos. Essa técnica é essencial para suportar características da linguagem, como a sobrecarga de funções e a resolução de nomes em diferentes escopos (KEFALLONITIS, 2007). Contudo, o *name mangling* dificulta a vinculação de símbolos exportados por bibliotecas dinâmicas, especialmente quando a aplicação principal ou outros *plugins* são compilados com diferentes compiladores ou configurações, situações nas quais o nome do símbolo esperado pela apli-

cação pode não corresponder ao nome decorado presente na biblioteca. Para mitigar esse problema, a linguagem C++ oferece o mecanismo de *linkage specification* por meio da diretiva `extern "C"`. Ao declarar uma função ou um bloco de funções utilizando `extern "C"`, instrui-se o compilador a utilizar a convenção de ligação da linguagem C, que não realiza o *name mangling*. Isso assegura que o nome do símbolo na biblioteca dinâmica seja preservado, permitindo que a aplicação o encontre e o utilize corretamente. A aplicação desta diretiva nas funções que compõem a interface do *plugin* é crucial para garantir a comunicação eficaz entre a aplicação principal e os *plugins*.

3 Materiais e Métodos

Este capítulo apresenta os processos realizados na execução deste trabalho. A seção de pesquisa bibliográfica descreve as fontes consultadas e o processo utilizado para identificar e selecionar as referências mais relevantes. Em método de desenvolvimento de software, descrevemos a metodologia adotada para a construção da aplicação, detalhando as abordagens utilizadas para garantir eficiência e organização no processo de desenvolvimento. A seção de ferramentas apresenta os recursos empregados ao longo do projeto, justificando a escolha de cada um com base em sua adequação às necessidades do trabalho. A seção de arquitetura de plugins detalha o sistema modular implementado, que permite a extensão das funcionalidades da aplicação sem a necessidade de recompilação. Por fim, a seção de trabalhos correlatos analisa pesquisas e soluções semelhantes, contextualizando este estudo dentro do cenário existente.

3.1 Pesquisa bibliográfica

A pesquisa bibliográfica baseou-se em fontes diversas, incluindo materiais fornecidos pelo orientador e pesquisas realizadas em plataformas acadêmicas como Google Acadêmico[1] e Periódicos CAPES[2]. O acesso a essas plataformas foi viabilizado por meio do login institucional da Universidade de Brasília (UnB), garantindo a consulta a fontes confiáveis e relevantes.

O estudo teve como foco a análise de métodos de aproximação aplicados à análise de sinais e a identificação das melhores práticas para o desenvolvimento de aplicações em Qt 6, C++ e Python.

Para otimizar a busca e direcionar os resultados para fontes pertinentes, foram utilizadas palavras-chave como "*Prony method*", "*Graphical user interfaces*" e "*Qt 6 and modern C++*", entre outros.

Também foram consultadas referências sobre as ferramentas utilizadas, como o *framework* Qt6 e as linguagens C++ e Python, visando garantir a eficiência e flexibilidade na implementação da interface gráfica e dos algoritmos de interpolação.

A pesquisa bibliográfica resultou em um conjunto significativo de publicações. Visando refinar a seleção, aplicamos critérios de seleção, priorizando artigos com acesso integral e disponíveis em português ou inglês, cujos títulos e resumos demonstrassem relevância para o tema em questão. Após análise inicial, cerca de 57 obras foram consideradas

[1] <https://scholar.google.com>
[2] <https://www.periodicos.capes.gov.br>

relevantes, das quais 31 foram selecionadas para leitura e aprofundamento.

3.2 Método de desenvolvimento de software

O desenvolvimento da aplicação adotou um modelo de desenvolvimento de software ágil, com inspiração no Scrum, adaptado às necessidades específicas do projeto. O Scrum é um *framework* ágil para gerenciamento de projetos, que enfatiza a colaboração, a flexibilidade e a entrega iterativa de valor. Ele se baseia em ciclos de desenvolvimento chamados *sprints*, nos quais a equipe trabalha em conjunto para alcançar metas de curto prazo e entregar incrementos do produto (KADENIC; KOUMADITIS; JUNKER-JENSEN, 2023). Para garantir organização e eficiência, adotamos *sprints* semanais, permitindo a definição de metas de curto prazo e a avaliação constante do desenvolvimento. O planejamento detalhado das *sprints* para o `TCC 1` e `TCC 2`, incluindo datas de início e fim, bem como seus objetivos específicos, pode ser encontrado nos Quadros 1 e 2. Apesar do planejamento semanal, algumas *sprints* tiveram duração variável devido a feriados, exigindo adaptações no cronograma.

Quadro 1: Planejamento de sprints para o TCC 1

Sprint	**Data Início**	**Data Fim**	**Objetivos**
1	2024-10-15	2024-10-22	Definição do tema
2	2024-10-22	2024-10-29	Definição dos objetivos
3	2024-10-29	2024-11-12	Estudo do Qt 6
4	2024-11-12	2024-11-19	Menu de seleção de arquivos e plot do sinal
5	2024-11-19	2024-11-26	Interface de comunicação entre C++ e Python, integração do método de Prony, estilização de curvas e visibilidade de curvas aproximadas
6	2024-11-26	2024-12-03	Melhorias na estilização e na interface de comunicação entre C++ e Python, adição da aba de configuração dos métodos e geração de relatório
7	2024-12-03	2024-12-10	Correções, refatorações e integração do método MPM
8	2024-12-10	2024-12-17	Apresentação do relatório e melhorias de usabilidade
9	2024-12-17	2025-01-07	Estudo sobre escrita de documento científico
10	2025-01-07	2025-01-14	Estudo sobre os métodos de aproximação utilizados
11	2025-01-14	2025-01-21	Escrita do cronograma e dos resultados parciais
12	2025-01-21	2025-01-28	Escrita da metodologia e fundamentação teórica
13	2025-01-28	2025-02-04	Escrita da introdução e considerações finais
14	2025-02-04	2025-02-11	Finalização da escrita do documento
15	2025-02-11	2025-02-18	Preparação para a defesa

Fonte: os Autores

Quadro 2: Planejamento de sprints para o TCC 2

Sprint	Data Início	Data Fim	Objetivos
1	2025-04-08	2025-04-15	Correção da funcionalidade de estilização
2	2025-04-15	2025-04-22	Integração dos métodos `ERA` e `SSI`
3	2025-04-22	2025-04-29	Implementação do sistema de *plugins* e migração dos métodos existentes para o novo sistema
4	2025-04-29	2025-05-06	Implementação e integração do arquivo de configurações
5	2025-05-06	2025-05-13	Estudo sobre o método `ERA`
6	2025-05-13	2025-05-20	Escrita do Referencial Teórico sobre o método `ERA`
7	2025-05-20	2025-05-27	Estudo sobre o método `SSI`
8	2025-05-27	2025-06-03	Escrita do Referencial Teórico sobre o método `SSI`
9	2025-06-03	2025-06-10	Visualização dos autovalores gerados pelo método de interpolação
10	2025-06-10	2025-06-17	Refatorações no código e página de configurações para a visualização dos autovalores
11	2025-06-17	2025-06-24	Funcionalidade de zoom e melhorias no fluxo de interação do usuário
12	2025-06-24	2025-07-01	Ajustes no documento texto
13	2025-07-01	2025-07-08	Ajustes no documento texto
14	2025-07-08	2025-07-15	Finalização da escrita do documento
15	2025-07-15	2025-07-22	Preparação para a defesa

Fonte: os Autores

Semanalmente, realizamos reuniões com o orientador para apresentar os avanços e alinhar o direcionamento do projeto. Além dessas reuniões fixas, os autores também se reuniam periodicamente para planejamento e acompanhamento interno, garantindo alinhamento com os objetivos da *sprint* em andamento. A comunicação ao longo do desenvolvimento foi facilitada pelo uso do Telegram[3] para trocas assíncronas de mensagens e do Microsoft Teams[4] para reuniões síncronas.

As responsabilidades foram divididas entre os autores: um ficou responsável pela integração do Python e apresentação das curvas dos sinais, enquanto o outro se concentrou nos recursos visuais, incluindo a estilização das curvas e a geração do relatório de desempenho dos métodos de interpolação.

O controle de versão foi realizado com Git, utilizando *branches* separadas para cada funcionalidade, minimizando conflitos e facilitando a integração do código. Ao final de cada *sprint*, realizávamos uma reunião para integrar as funcionalidades desenvolvidas, discutir possíveis pendências e definir os próximos passos.

[3] <https://web.telegram.org>
[4] <https://teams.microsoft.com/v2/>

3.3 Ferramentas

Nesta seção são apresentadas as ferramentas utilizadas no desenvolvimento da aplicação, incluindo linguagens de programação, bibliotecas e softwares de suporte. Para cada ferramenta, são detalhadas suas versões e a justificativa para sua escolha no contexto do projeto.

O sistema de controle de versão Git[5], na versão 2.43.0, foi utilizado para gerenciar o código-fonte do projeto, permitindo rastreamento de alterações, colaboração eficiente e versionamento seguro do software. A escolha do Git se deve à sua ampla adoção na indústria e à robustez que oferece, especialmente em projetos com múltiplas iterações e experimentações.

A linguagem de programação C++, na versão C++20[6], foi utilizada na implementação principal da aplicação. Sua escolha se deve à alta performance, garantindo eficiência no processamento e manipulação de dados do sinal. Além disso, o C++ conta com bibliotecas consolidadas para o desenvolvimento de interfaces gráficas modernas, como o Qt.

Para a construção da interface gráfica foi utilizado o *framework* Qt6[7]. Ele foi escolhido por sua versatilidade, suporte a múltiplas plataformas e integração eficiente com C++. Além disso, o Qt6 oferece ferramentas avançadas para visualização de sinais, facilitando a implementação dos requisitos gráficos do projeto.

Os métodos de interpolação utilizados já estavam previamente implementados em Python[8] (versão 3.12), aproveitando bibliotecas científicas como NumPy[9] e SciPy[10]. No contexto deste trabalho, não foram desenvolvidos novos algoritmos de interpolação; a principal contribuição foi a implementação da interface de comunicação entre a ferramenta e esses algoritmos já existentes, permitindo sua integração e uso na aplicação.

3.4 Arquitetura de Plugins

O sistema de *plugins* da aplicação foi projetado para permitir a adição de novos métodos de interpolação sem a necessidade de modificar ou recompilar o código principal. Os *plugins* são implementados como bibliotecas dinâmicas e carregados em tempo de execução a partir de um diretório indicado no arquivo de configurações da aplicação.

Para que um *plugin* seja reconhecido, ele deve implementar uma interface de-

[5] <https://git-scm.com>
[6] <https://isocpp.org/>
[7] <https://www.qt.io>
[8] <https://www.python.org>
[9] <https://numpy.org>
[10] <https://scipy.org>

finida em um arquivo de cabeçalho. Essa interface, representada pela classe abstrata `ApproximationMethod`, define os métodos obrigatórios que os *plugins* devem implementar, como `approximate()`, responsável por calcular o sinal aproximado, e `getName()`, que retorna o nome do *plugin*. Além disso, os *plugins* devem fornecer as funções `create()` e `destroy()`, utilizadas para criar instâncias do *plugin* e liberar os recursos alocados, respectivamente.

O método `approximate()` retorna um objeto do tipo `ApproximationResult`, que contém tanto o sinal aproximado quanto um relatório sobre o desempenho do método. Esse relatório inclui métricas como o erro médio quadrático (MSE) e o erro absoluto médio (MAI), além de informações sobre a decomposição do sinal em soma de exponenciais. Esses dados permitem ao usuário avaliar a qualidade da interpolação e compreender melhor como o método processou o sinal original.

Os parâmetros necessários para a execução dos métodos de interpolação devem ser inicializados pelos *plugins* em seus construtores. Cada parâmetro é representado pela classe `Parameter`, que inclui um nome, um valor padrão e, opcionalmente, metadados associados. Esses metadados, definidos pela estrutura `ParameterMetadata`, permitem especificar informações como valores mínimos, máximos e incrementos, dependendo do tipo do parâmetro. Durante a execução, a aplicação utiliza os metadados para validar os valores definidos pelo usuário, garantindo que estejam dentro dos limites aceitáveis. Além disso, os metadados são aproveitados para gerar dinamicamente a interface gráfica de configuração, facilitando o ajuste dos parâmetros pelo usuário.

A aplicação utiliza a classe `QLibrary` da biblioteca Qt para carregar os *plugins* em tempo de execução, independentemente do sistema operacional. Durante o carregamento, verifica-se se o *plugin* implementa corretamente a interface e fornece as funções necessárias. Caso a verificação seja bem-sucedida, o *plugin* é adicionado à lista de métodos disponíveis, permitindo que o usuário o utilize diretamente.

Para manter a interface responsiva durante a execução dos métodos de interpolação, a aplicação realiza os cálculos de forma assíncrona utilizando as classes `QFuture` e `QFutureWatcher`. Essa abordagem permite que operações computacionalmente intensivas sejam executadas em segundo plano, enquanto o usuário continua interagindo com a aplicação normalmente. Quando o cálculo é concluído, o `QFutureWatcher` notifica a aplicação, que, então, atualiza a interface com os resultados.

3.5 Trabalhos correlatos

Costa (2018), em seu projeto, apresenta o desenvolvimento de uma interface gráfica em Qt/C++ para otimizar o uso do algoritmo de inversão de forma de onda (FWI) em imageamento sísmico, utilizado na indústria de óleo e gás. A interface permite ao

usuário configurar projetos, inserir parâmetros validados, executar simulações e monitorar resultados em tempo real facilitando a análise iterativa. O trabalho visa transformar um código científico complexo em um produto comercial acessível, agregando valor técnico e competitivo para pequenas e médias empresas do setor.

Colombet et al. (2015) descreve, em seu artigo, o desenvolvimento de um software chamado AnyWave, criado em Qt/C++, e tem como objetivo oferecer uma interface gráfica acessível para visualização e processamento de sinais eletrofisiológicos. A ferramenta busca facilitar a aplicação de algoritmos complexos, permitindo que usuários sem expertise técnica interajam com dados neurológicos de forma eficiente.

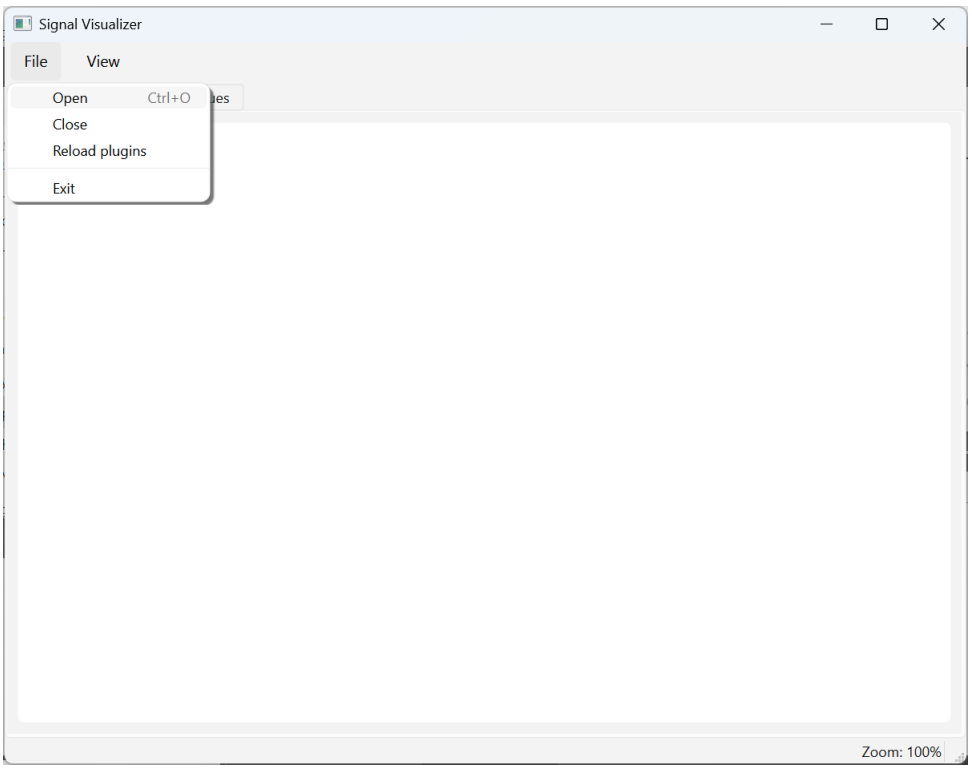
4 Resultados

Este capítulo apresenta o progresso no desenvolvimento da aplicação desktop construída utilizando Qt 6, destacando suas principais funcionalidades e o estado atual. A aplicação tem como objetivo visualizar sinais e executar diferentes métodos de aproximação. Serão detalhadas as funcionalidades implementadas até o momento, juntamente com os desafios enfrentados durante o desenvolvimento. Capturas de tela da interface da aplicação serão incluídas para ilustrar seu estado atual.

4.1 Funcionalidades

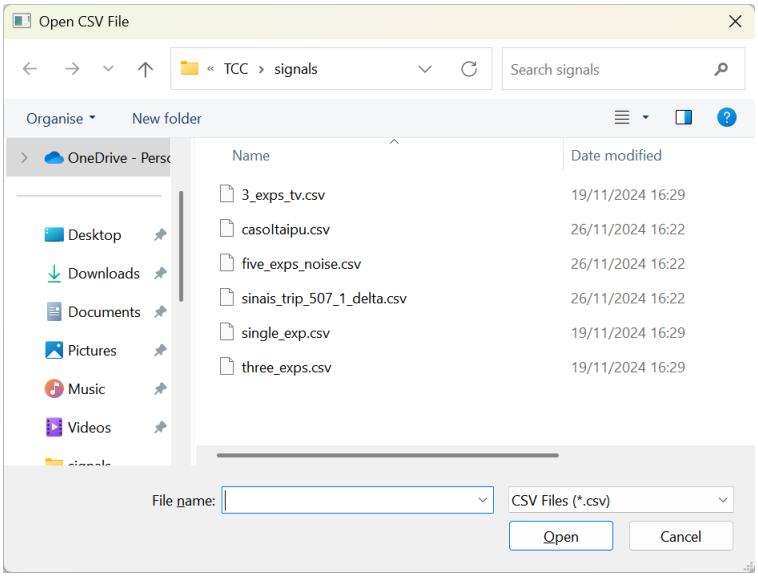
O usuário pode carregar sinais na aplicação por meio de arquivos CSV, que contêm dados tabulares representando amostras de sinais em uma série temporal. Para isso, a aplicação oferece a opção de carregar arquivos através do menu `File` (Figura 7), o qual abre o sistema de arquivos configurado no sistema operacional, já com um filtro para apresentar apenas diretórios e arquivos CSV, a fim de facilitar a busca do sinal desejado (Figura 8). Ao carregar esses dados, a aplicação transforma as amostras contidas no arquivo em curvas, utilizando o algoritmo Spline (Figura 9).

Figura 7 – Menu *File* da aplicação, exibindo as opções `Open`, `Close` e `Exit`.



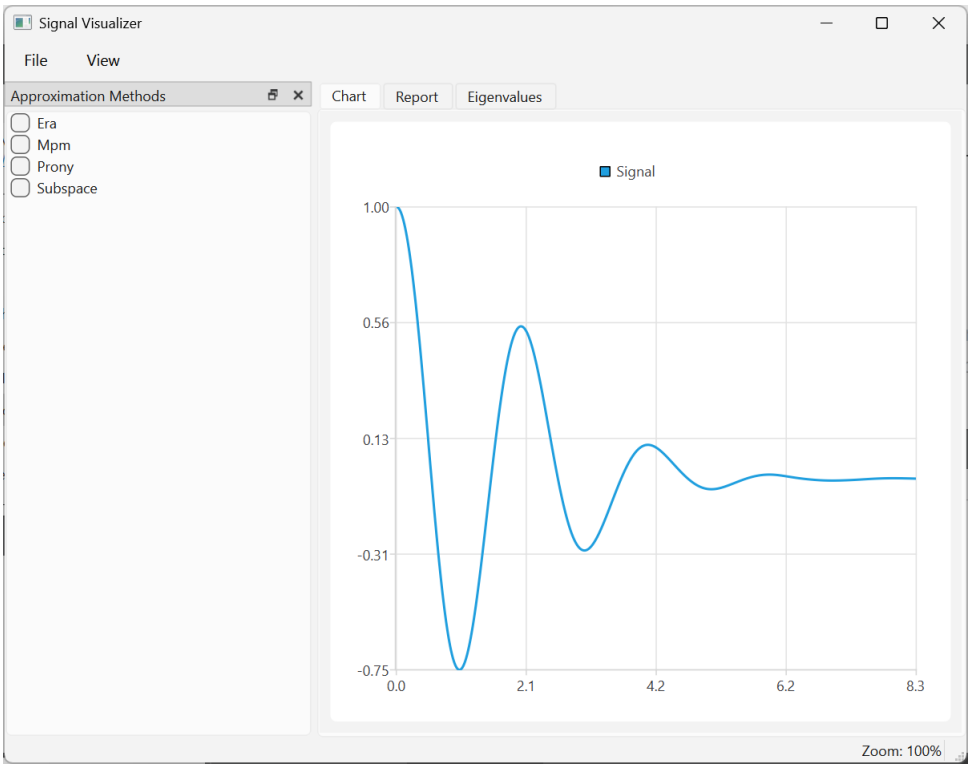
Fonte: os Autores.

Figura 8 – Gerenciador de arquivos com filtro de CSV aplicado.



Fonte: os Autores.

Figura 9 – Interface da aplicação exibindo o gráfico com um sinal.

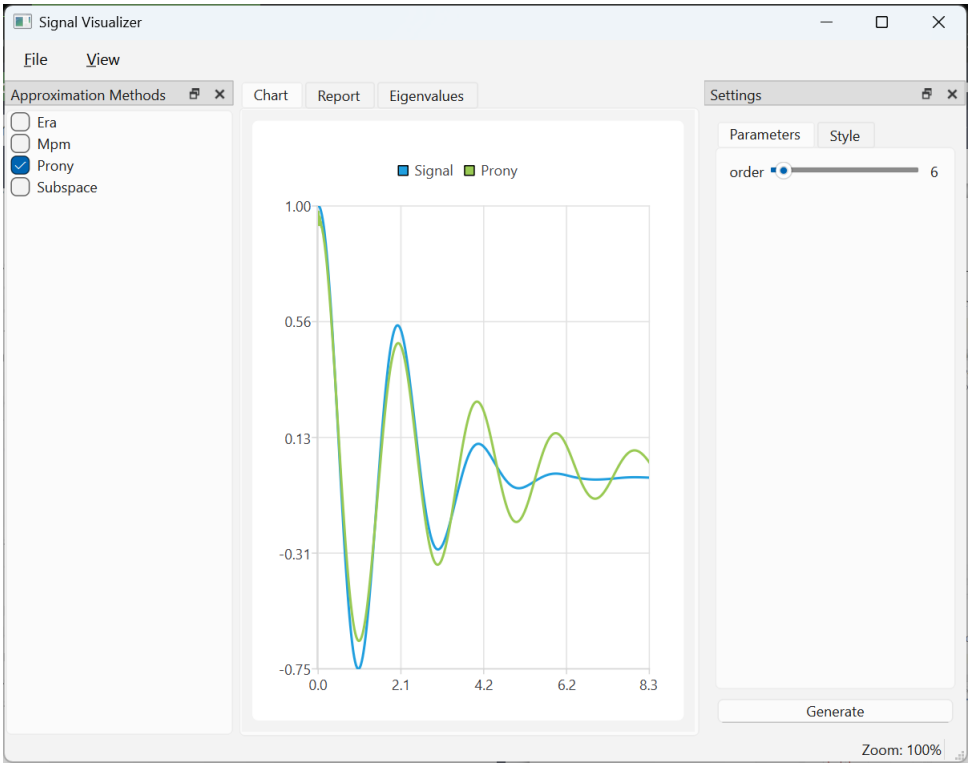


Fonte: os Autores.

Após o carregamento do sinal, o usuário pode configurar o método que deseja aplicar. Cada método possui um conjunto de parâmetros específicos que podem ser ajustados a fim de tentar obter a melhor aproximação possível do sinal original. Por exemplo, no Método de Prony é possível configurar a ordem do modelo de exponenciais através do parâmetro `order`. A aproximação apresentada na Figura 10 foi gerada utilizando Prony com ordem igual a 6 e, como podemos notar, não se aproximou tão bem da curva original

quanto à aproximação gerada com ordem igual a 56 (Figura 11).

Figura 10 – Aproximação gerada pelo método Prony com ordem 6.



Fonte: os Autores.

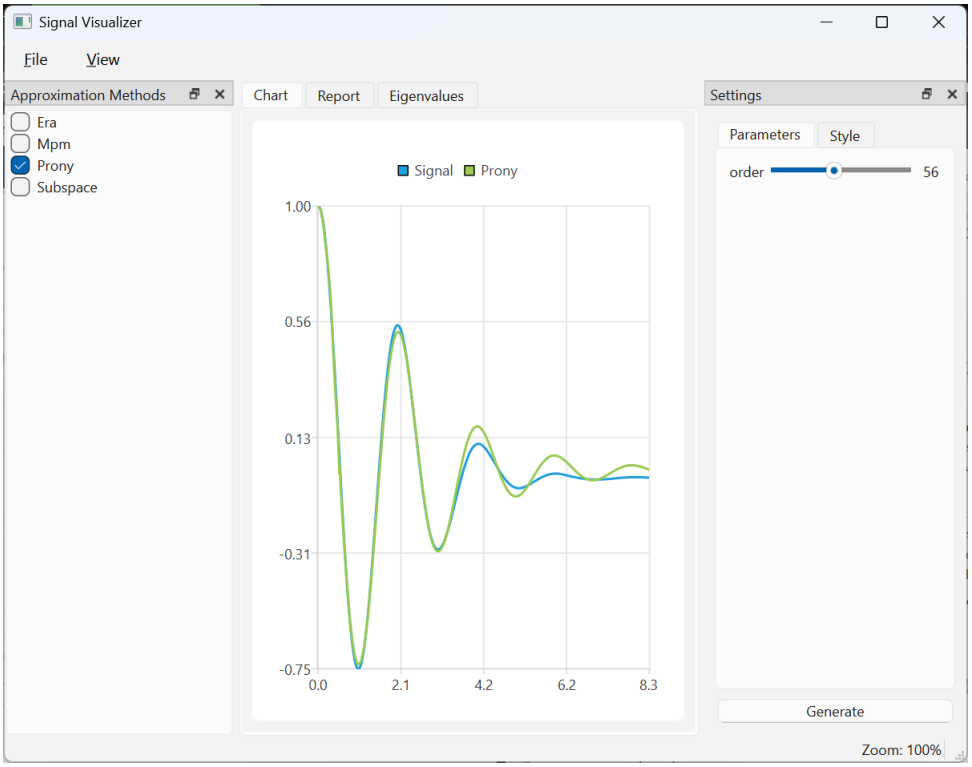
Os métodos foram implementados utilizando a linguagem de programação Python 3 e executados pela aplicação C++ por meio da instanciação de processos filhos gerenciados pelo *framework* Qt 6. Os *scripts* Python produzem um conjunto de coordenadas de mesmo tamanho das amostras do sinal original. Essas coordenadas correspondem às aproximações e são utilizadas para a plotagem dos gráficos e para calcular algumas métricas de desempenho do método.

Além disso, a aplicação oferece opções de personalização visual para as curvas plotadas, onde o usuário pode modificar aspectos como cor, espessura e formato das linhas. A Figura 12 apresenta a aba de estilização de uma curva gerada pelo método de Prony, destacando como essas configurações afetam a visualização dos gráficos.

No painel esquerdo há uma lista de todos os métodos disponíveis para aproximação e, através dele, podemos escolher quais curvas são visíveis, apenas marcando as caixas de seleção ao lado dos nomes dos métodos que desejamos visualizar. Essa funcionalidade é ilustrada através das Figuras 13 e 14, onde a Figura 13 apresenta um gráfico com quatro aproximações visíveis, utilizando os métodos Prony, MPM, ERA e SSI. Já na Figura 14 as aproximações dos métodos MPM e SSI não foram selecionadas e, consequentemente, apenas as aproximações geradas pelos métodos Prony e ERA estão visíveis.

Para avaliar a eficácia dos métodos de aproximação, a aplicação gera relatórios que

Figura 11 – Aproximação gerada pelo método Prony com ordem 56.



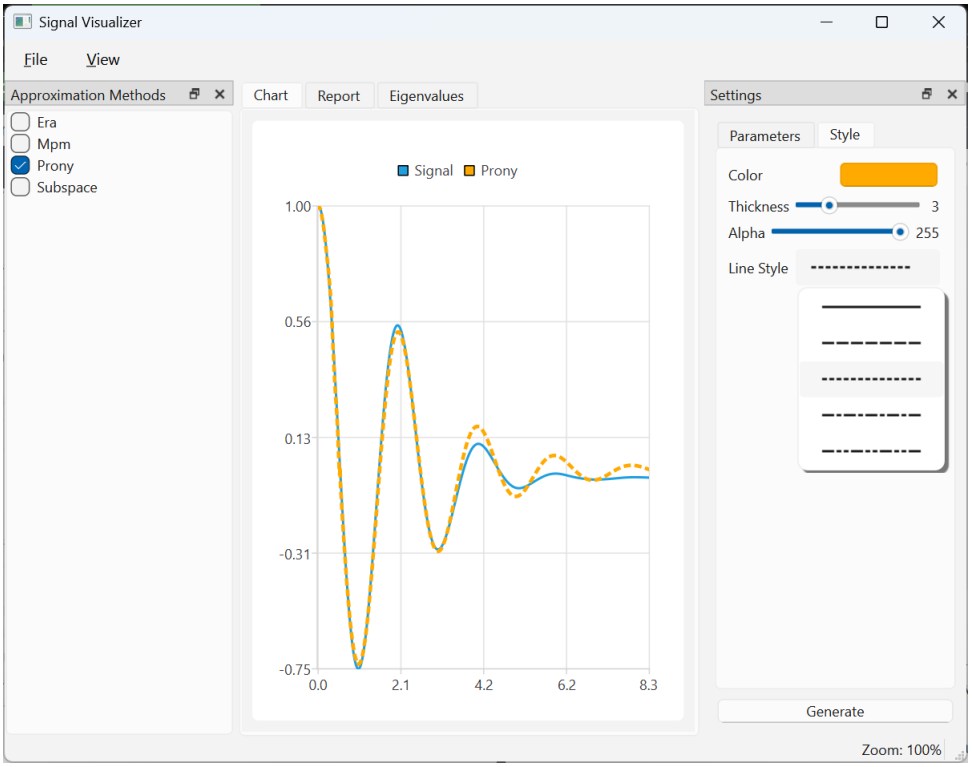
Fonte: os Autores.

incluem os parâmetros resultantes da decomposição do sinal em soma de exponenciais, obtidos pelo método aplicado, além de métricas comumente utilizadas na análise de erros. Os parâmetros de decomposição incluem A, que representa o coeficiente de amplitude e indica a intensidade de cada componente; λ, o fator de amortecimento, que determina a taxa de decaimento do sinal ao longo do tempo; ω, a frequência angular, que descreve a periodicidade das oscilações; θ, o ângulo de fase, que define a posição inicial das ondas; e f, a frequência em Hertz, calculada pela expressão $f = \frac{\omega}{2*\pi}$.

As métricas de análise de erros fornecidas incluem o Erro Médio Absoluto (*Mean Average Error* – MAE), que mede a magnitude média das diferenças entre os valores aproximados e os valores reais, e o Erro Quadrático Médio (*Mean Square Error* – MSE), que calcula o quadrado da diferença média entre os valores reais e os interpolados, sendo mais sensível a grandes discrepâncias. Essas métricas são calculadas automaticamente após a execução do processo de aproximação, e os resultados são apresentados ao usuário no formato de tabela, ordenados pela frequência, como apresentado na Figura 15.

Adicionalmente, para uma análise mais visual do comportamento dos métodos, a aplicação oferece a visualização dos autovalores ($\lambda = \alpha + j\omega$) resultantes da aproximação. Estes são extraídos do relatório do método, onde a parte real α representa a taxa de amortecimento e a parte imaginária está relacionada à frequência. A aplicação plota estes autovalores como pontos em um gráfico de dispersão no plano complexo (Figura 16). O eixo horizontal corresponde à parte real α e o eixo vertical à frequência f (em Hertz).

Figura 12 – Interface de estilização de curvas.



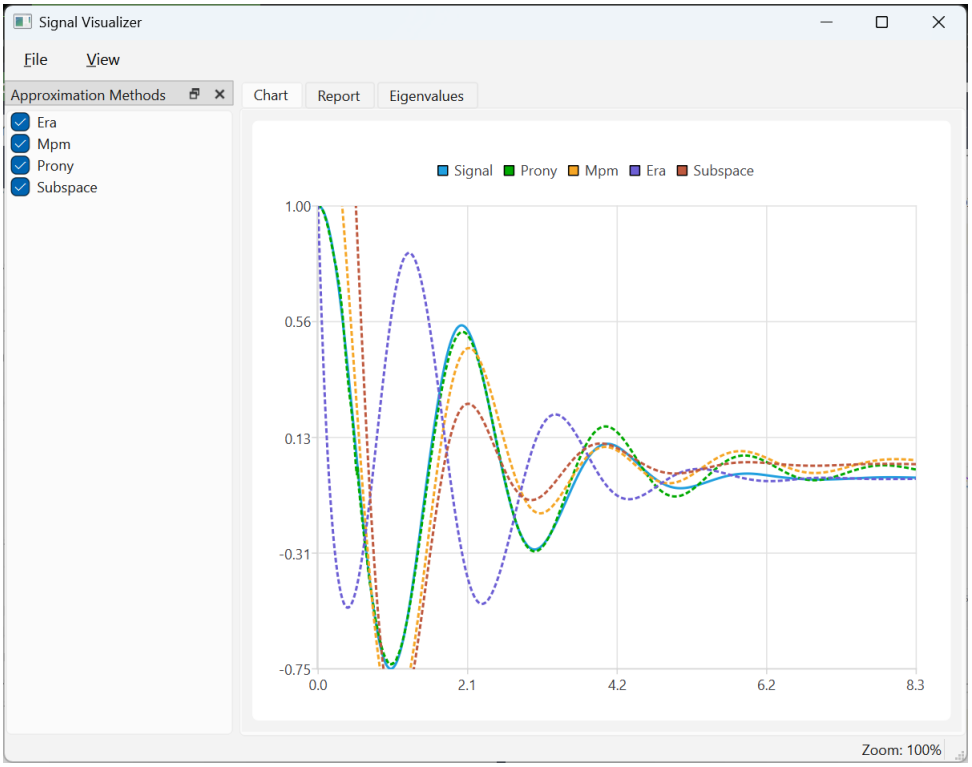
Fonte: os Autores.

A aplicação oferece opções de configuração para esta visualização, acessíveis através de um painel dedicado. O usuário pode optar por exibir todos os autovalores ou filtrar aqueles cuja parte real, em valor absoluto, não exceda um limiar $\epsilon > 0$ ajustável. Esta filtragem é útil para focar em autovalores mais próximos ao eixo imaginário, que geralmente correspondem aos componentes mais persistentes e significativos do sinal. A interface de configuração permite ajustar o valor de ϵ e alternar a exibição de todos os pontos, atualizando o gráfico dinamicamente (Figura 17). Esta funcionalidade auxilia na compreensão de aspectos como a estabilidade e as características de frequência dos modelos exponenciais gerados pelos métodos.

A extensibilidade da aplicação é assegurada por um sistema de *plugins*, que viabiliza a incorporação de novos métodos de interpolação sem a necessidade de recompilar o programa principal. O processo de gerenciamento de *plugins* inicia-se com a descoberta dos *plugins* disponíveis. A aplicação varre um diretório específico, configurável pelo usuário via arquivo de configurações, em busca de arquivos que possam ser bibliotecas dinâmicas. Para cada arquivo encontrado nesse diretório, a aplicação tenta carregá-lo como um *plugin*, conforme ilustrado no Código 1.

Para cada arquivo identificado, a aplicação utiliza a classe `QLibrary` do *framework* Qt para carregar dinamicamente os *plugins*. O processo inicia-se com a tentativa de carregar a biblioteca a partir de um caminho especificado, como demonstrado no Código 2:

Figura 13 – Gráfico com quatro aproximações.



Fonte: os Autores.

Código 1: Método que carrega todos os *plugins* do diretório indicado.

```
1 void PluginManager::loadAllPlugins(const QString &pluginPath) {
2   QDir pluginsDir(pluginPath);
3   if (!pluginsDir.exists()) {
4     qWarning() << "Plugin directory does not exist:" << pluginPath;
5     return;
6   }
7
8   QStringList entrys = pluginsDir.entryList(QDir::Files, QDir::Name);
9   for (const QString &fileName : entrys) {
10    QString filePath = pluginsDir.absoluteFilePath(fileName);
11    loadPlugin(filePath);
12  }
13 }
```

Fonte: os Autores.

Uma vez que a biblioteca é carregada com sucesso, a aplicação busca por funções específicas exportadas por ela: a função `create()`, responsável por retornar uma instância do método de aproximação (Código 3). e a função `destroy()`, responsável por liberar os recursos da instância recebida (Código 4). Com ambas as funções resolvidas, a aplicação invoca a função `create()` para obter a instância do *plugin* (Código 5)

Se a instância do método for criada com sucesso e não houver conflitos de nome, o *plugin* é registrado internamente. A aplicação armazena tanto a instância do método quanto as informações da biblioteca carregada (incluindo a referência à biblioteca e à

Código 2: Trecho que carrega uma biblioteca dinâmica em memória.

```
QLibrary *library = new QLibrary(filePath, this);
if (!library->load()) {
  qWarning() << "Failed to load library:" << library->errorString();
  delete library;
  return;
}
```

Fonte: os Autores.

Código 3: Trecho que resolve a função `create` exportada pelo *plugin*.

```
void PluginManager::loadPlugin(const QString &filePath) {
  // ...
  using CreatePluginFunc = ApproximationMethod *(*)();
  CreatePluginFunc createPlugin = (CreatePluginFunc)library->resolve("create");
  if (!createPlugin) {
    qWarning() << "Failed to resolve create function:"
               << library->errorString();
    library->unload();
    delete library;
    return;
  }
  // ...
}
```

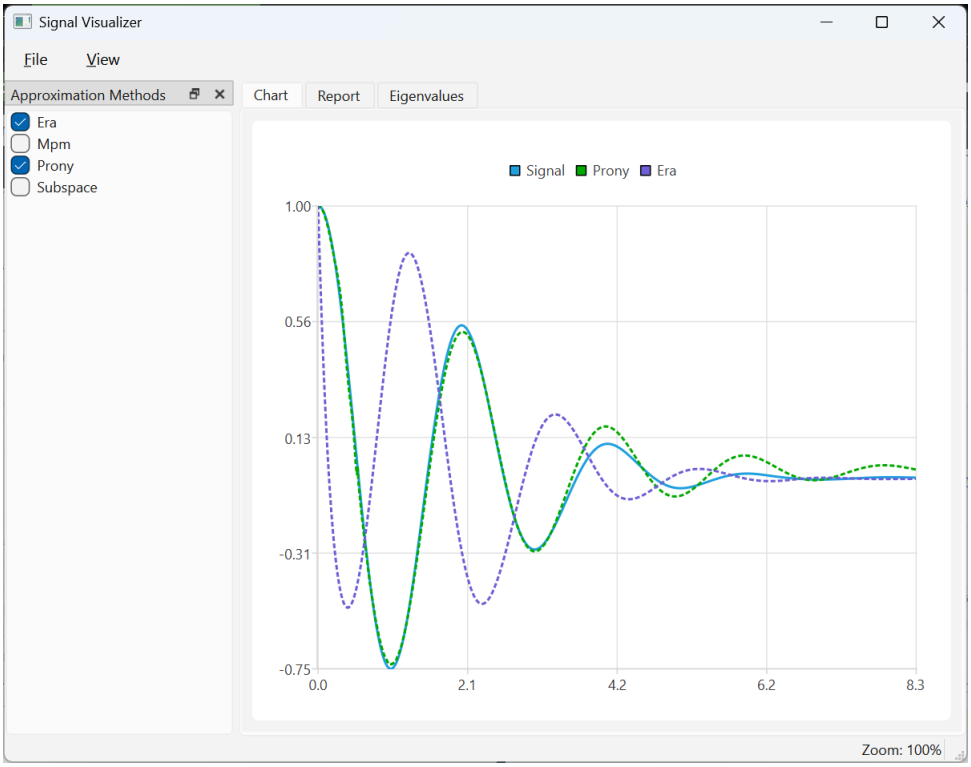
Fonte: os Autores.

Código 4: Trecho que resolve a função `destroy` exportada pelo *plugin*.

```
void PluginManager::loadPlugin(const QString &filePath) {
  // ...
  using DestroyPluginFunc = void (*)(ApproximationMethod *);
  DestroyPluginFunc destroyPlugin =
      (DestroyPluginFunc)library->resolve("destroy");
  if (!destroyPlugin) {
    qWarning() << "Failed to resolve destroy function:"
               << library->errorString();
    library->unload();
    delete library;
    return;
  }
  // ...
}
```

Fonte: os Autores.

Figura 14 – Aproximação gerada pelos métodos Prony e ERA.



Fonte: os Autores.

Código 5: Trecho invocando a função `create` para instanciar o *plugin*.

```
void PluginManager::loadPlugin(const QString &filePath) {
  // ...
  ApproximationMethod *plugin = createPlugin();
  if (!plugin) {
    qWarning() << "Failed to create plugin instance.";
    library->unload();
    delete library;
    return;
  }
  // ...
}
```

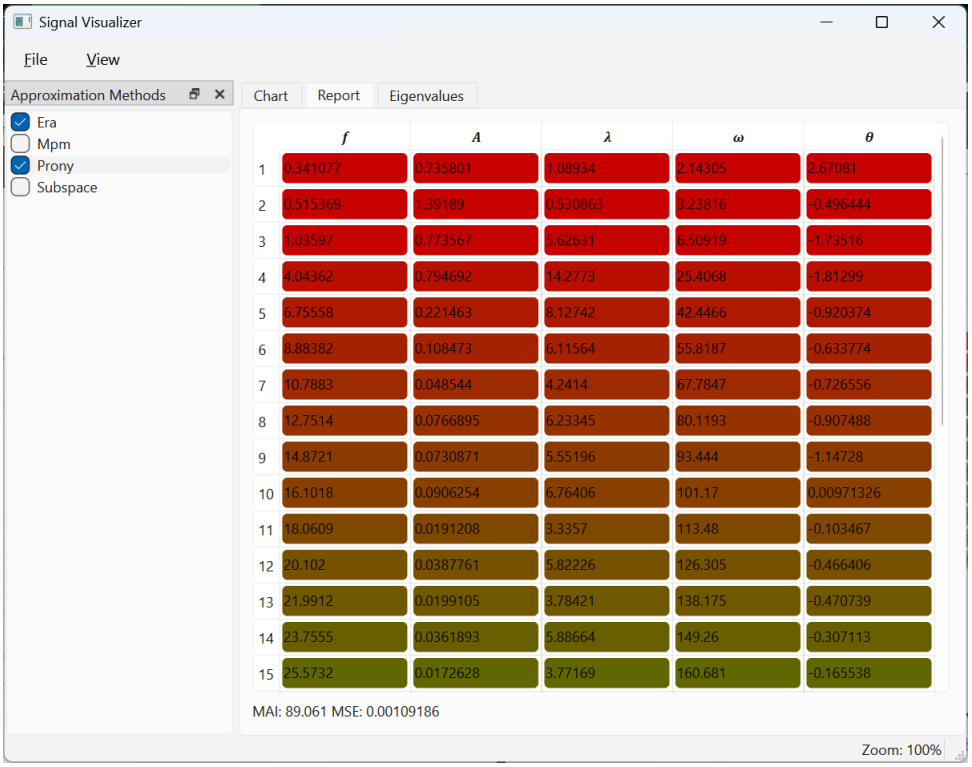
Fonte: os Autores.

função `destroy()`), associando-as ao nome do *plugin*, conforme apresentado no Código 6.

Com isso, o novo método de interpolação torna-se disponível na interface do usuário. O processo de descarregamento de um *plugin*, tipicamente realizado ao fechar a aplicação ou ao recarregar os *plugins*, envolve chamar a função `destroy()` correspondente para liberar a instância do método e, subsequentemente, descarregar a biblioteca da memória utilizando `QLibrary::unload()` (7)

A aplicação é projetada para lidar com possíveis falhas durante essas etapas. Caso um arquivo de *plugin* não seja encontrado, não possa ser carregado, ou se as funções `create()` ou `destroy()` não forem localizadas, ou ainda se a criação da instância do

Figura 15 – Tabela de relatório de desempenho do método de Prony.



Fonte: os Autores.

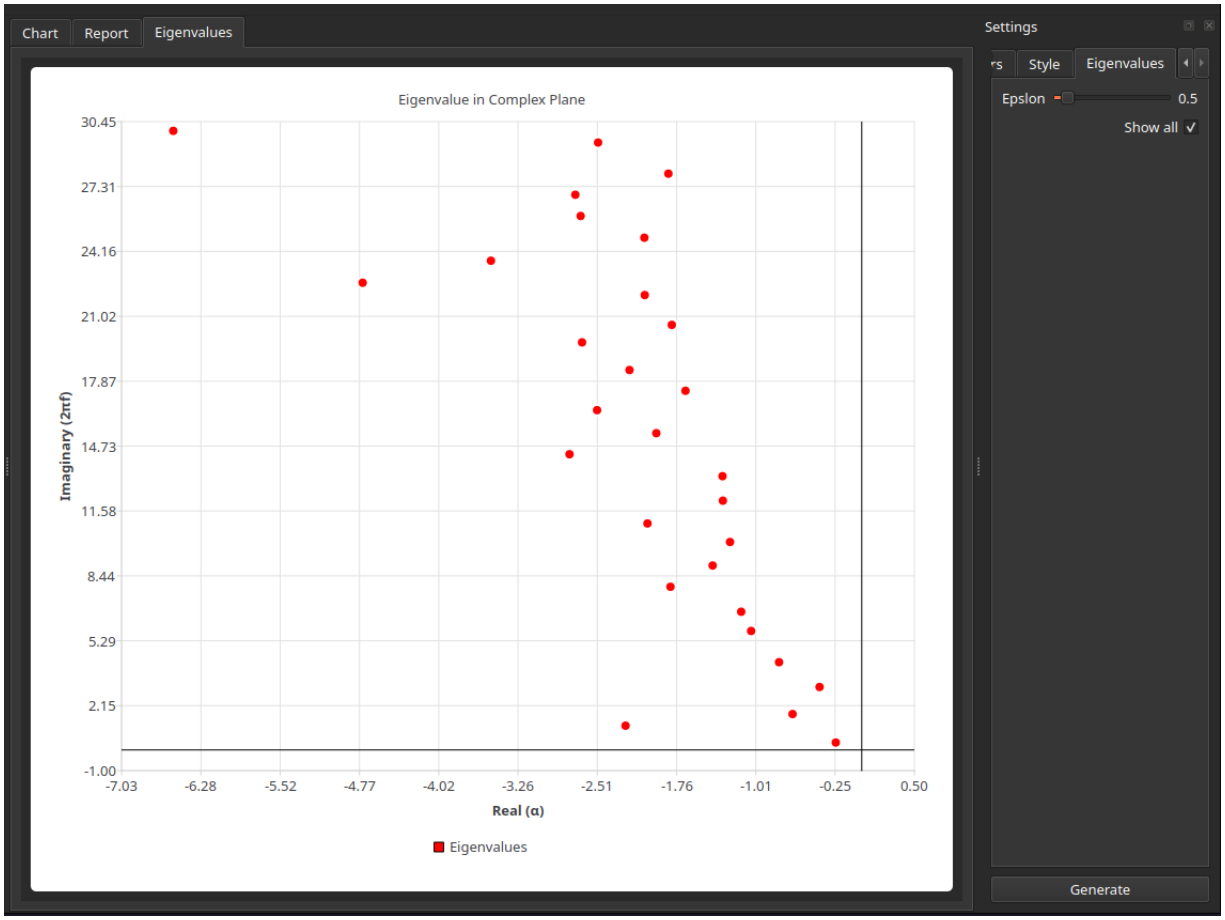
Código 6: Trecho responsável por assegurar que os *plugins* tenham nomes distintos e por armazena-los em memória.

```
1 void PluginManager::loadPlugin(const QString &filePath) {
2   // ...
3   QString pluginName = QString::fromUtf8(plugin->getName());
4   if (plugins.contains(pluginName)) {
5     destroyPlugin(plugin);
6     library->unload();
7     delete library;
8     return;
9   }
10  plugins.insert(pluginName, plugin);
11  loadedPlugins.insert(pluginName, {library, destroyPlugin});
12  return;
13 }
```

Fonte: os Autores.

plugin falhar, mensagens de aviso são registradas utilizando `qWarning()`, como visto nos exemplos anteriores. O *plugin* problemático é então ignorado, permitindo que a aplicação continue funcionando com os demais *plugins* carregados corretamente.

Figura 16 – Gráfico com uma representação dos autovalores sem filtro.



Fonte: os Autores.

Código 7: Trecho responsável por liberar os recursos dos *plugins* alocados.

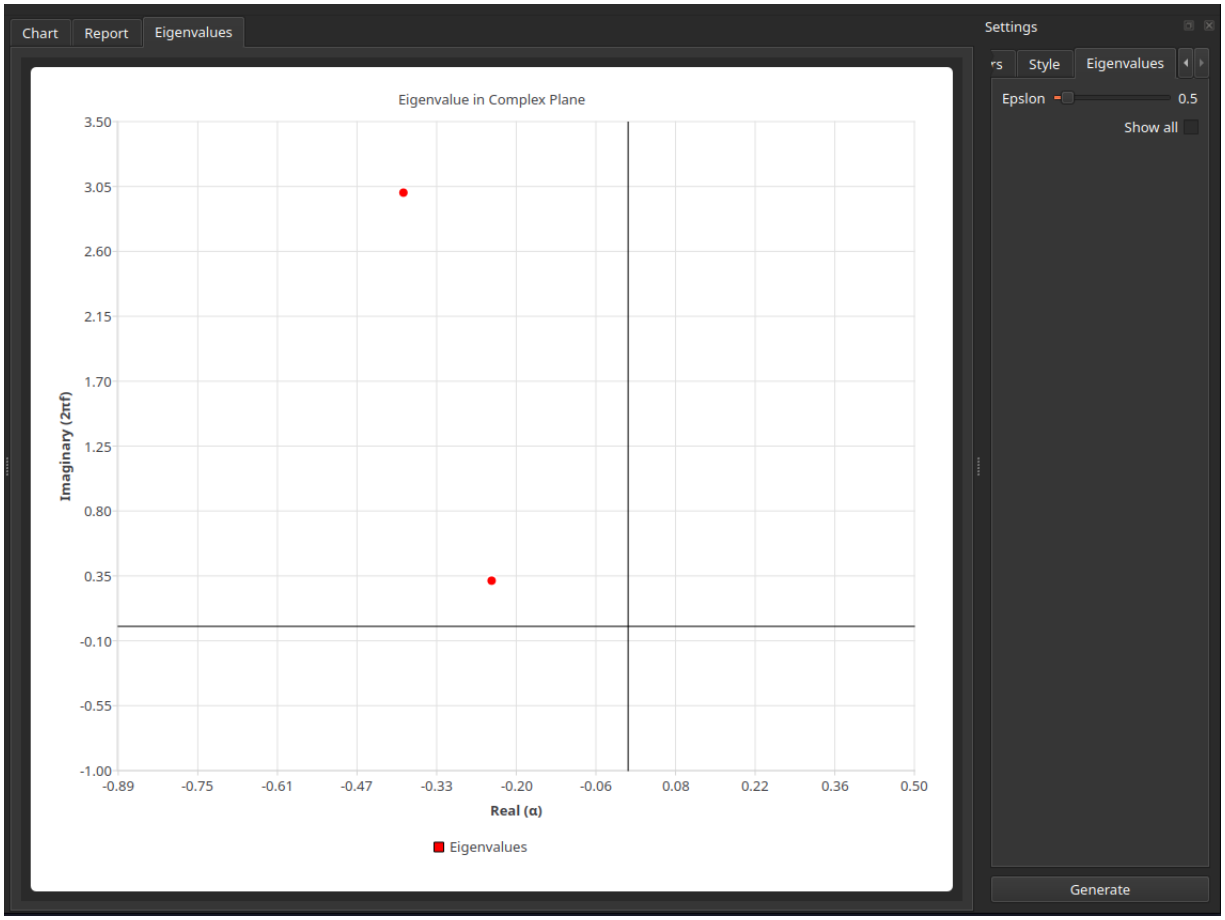
```
void PluginManager::unloadAllPlugins() {
  for (auto it = loadedPlugins.begin(); it != loadedPlugins.end(); ++it) {
    const QString &pluginName = it.key();
    LoadedPlugin &loadedPlugin = it.value();
    if (auto *plugin = plugins.take(pluginName))
      loadedPlugin.destroy(plugin);
    loadedPlugin.library->unload();
    delete loadedPlugin.library;
  }
  loadedPlugins.clear();
}
```

Fonte: os Autores.

4.2 Desafios encontrados

Durante o desenvolvimento da aplicação, enfrentamos uma série de desafios. A complexidade do projeto, combinada com as especificidades da linguagem C++ e das bibliotecas utilizadas, trouxe à tona dificuldades em diferentes áreas, desde a gestão de memória até a apresentação de dados. A seguir, detalhamos alguns dos principais obstáculos que encontramos e as abordagens que adotamos para superá-los.

Figura 17 – Painel de configurações do gráfico de autovalores com *epslon* igual a 0.5



Fonte: os Autores

Utilizar padrões de projeto consagrados para construções de interfaces em C++ foi um grande desafio. Diferentemente de linguagens que oferecem gerenciamento automático de memória, como Python ou Java, C++ exige que o desenvolvedor gerencie manualmente a alocação e liberação de memória. Isso requer atenção constante para evitar vazamentos e garantir a correta destruição de objetos, especialmente quando lidamos com dependências entre as várias camadas da aplicação. Essa complexidade pode resultar em erros sutis e difíceis de depurar.

Outro desafio significativo foi a exportação da tabela de relatório no formato PDF utilizando Qt. Embora o Qt ofereça ferramentas para criar documentos PDF, garantir que a formatação da tabela fosse clara se mostrou complicado. A representação e a disposição dos dados exigem ajustes cuidadosos, especialmente para manter a legibilidade e a organização das informações apresentadas. A necessidade de formatar corretamente colunas, alinhar textos e escolher fontes apropriadas adicionou uma camada extra de complexidade ao processo, tornando essencial realizar múltiplos testes para assegurar que o resultado final atendesse às expectativas.

Além disso, garantir que o gráfico do sinal seja adequadamente dimensionado em

relação às diversas curvas apresentadas, mesmo em presença de *outliers*, foi um desafio. Quando os dados contêm valores extremos, como picos inesperados, a visualização pode se tornar confusa e enganosa. Por exemplo, em alguns resultados da interpolação de um sinal ruidoso, pontos extremos faziam com que as curvas principais ficassem quase invisíveis. Para resolver isso, desenvolvemos um algoritmo para assegurar que os eixos do gráfico fossem sempre bem dimensionados em relação ao sinal principal, evitando que os *outliers* impactassem a visualização.

5 Considerações Finais

A aplicação desenvolvida alcançou os seguintes objetivos específicos: (a) implementar uma interface interativa que integre múltiplos métodos de aproximação, (b) permitir que as informações dos sinais possam ser carregadas por meio de arquivos CSV, (c) permitir a visualização simultânea de várias aproximações para um sinal original, (d) desenvolver uma arquitetura extensível baseada em *plugins* e (e) implementar a visualização dos autovalores associados aos modelos de aproximação.

A implementação da interface de comunicação entre C++ e Python, viabilizada pelo *framework* Qt 6, possibilitou a integração dos métodos de aproximação de Prony, Matriz Pencil, Algoritmo de Realização de Autosistema e Identificação de Subespaço Estocástico, além do controle visual, que permite a personalização de cores e estilos das curvas e a alternância de sua visibilidade, a ferramenta oferece ao usuário a capacidade de ajustar os parâmetros de cada método.

Para complementar a análise, a aplicação gera relatórios com métricas de desempenho, como MAE e MSE, permitindo uma comparação quantitativa e objetiva entre os diferentes modelos de aproximação.

Alguns dos desafios enfrentados, como a gestão de memória em C++ e o ajuste dinâmico da escala dos gráficos, foram superados. Entretanto, a formatação das tabelas exportadas para PDF ainda requer ajustes para garantir sua correta apresentação. Apesar disso, o sistema demonstrou ser funcional, intuitivo e alinhado às expectativas do projeto.

5.1 Trabalhos Futuros

Uma futura melhoria para o sistema de *plugins* consiste em distribuí-los em repositórios separados, permitindo que os usuários baixem apenas os métodos de seu interesse. Sugere-se também a criação de uma central de extensões, que possibilitaria à comunidade compartilhar e distribuir novos algoritmos.

Para simplificar a distribuição e a instalação, recomenda-se automatizar o processo de lançamento da aplicação, gerando pacotes para os principais sistemas operacionais.

A usabilidade também seria aprimorada com uma tela de configurações para gerenciar opções, como o diretório dos *plugins*, eliminando a necessidade de editar arquivos manualmente.

Por fim, o desenvolvimento de um módulo para gerar sinais a partir de expressões matemáticas, como senoides e exponenciais, ofereceria um ambiente controlado para

validar e testar os algoritmos.

Referências

ALMUNIF, A.; FAN, L.; MIAO, Z. A tutorial on data-driven eigenvalue identification: Prony analysis, matrix pencil, and eigensystem realization algorithm. *International Transactions on Electrical Energy Systems*, Wiley Online Library, v. 30, n. 4, p. e12283, 2020. Disponível em: <https://doi.org/10.1002/2050-7038.12283>. Citado na página 24.

ANTON, H.; RORRES, C. *Álgebra Linear com Aplicações*. 10. ed. Porto Alegre: Bookman, 2011. Citado na página 18.

BEN-ISRAEL, A.; GREVILLE, T. N. E. *Generalized Inverses: Theory and Applications*. 2. ed. Springer, 2003. (CMS Books in Mathematics). ISBN 978-0-387-00293-4. Disponível em: <https://link.springer.com/book/10.1007/b97366>. Citado na página 23.

BOLDRINI, J. L. *Álgebra Linear*. 3. ed. Campinas, SP: Harbra, 1984. 424 p. ISBN 978-85-294-0202-4. Citado na página 18.

CARVALHO, G. C. A. de. Trabalho de Conclusão de Curso, *Desafios no Ensino de Engenharia*. Formosa, GO: [s.n.], 2022. Disponível em: <<adicioneolinksedisponÃnvel>>. Citado na página 13.

COHEN, M. X. *Practical linear algebra for data science*. Sebastopol, CA: O'Reilly Media, 2022. Citado na página 13.

COLOMBET, B. et al. Anywave: A cross-platform and modular software for visualizing and processing electrophysiological signals. *Journal of Neuroscience Methods*, v. 242, p. 118–126, 2015. Citado 2 vezes nas páginas 13 e 36.

COMPANY, T. Q. *Signals & Slots | Qt Core | Qt 6.9.1 — doc.qt.io*. 2025. <https://doc.qt.io/qt-6/signalsandslots.html>. [Accessed 13-07-2025]. Citado na página 28.

COSTA, D. R. S.; ANDRADE, B. A. R. de; FERREIRA, N. R. Análise dos métodos prony e mínimos quadrados na estimação de parâmetros de geradores síncronos de pólos salientes. *Programa de Pós-Graduação em Engenharia Elétrica – PPGEE UFBA*, 2020. Citado na página 13.

COSTA, R. M. B. Projeto de Graduação, *Desenvolvimento de uma Interface Gráfica para uma Solução de Imageamento Sísmico de Inversão FWI*. Rio de Janeiro, Brasil: [s.n.], 2018. Citado na página 35.

COSTA, T. B. *Identificação de Modos Eletromecânicos e Formas Modais Utilizando Dados Sincronizados*. Dissertação (Mestrado) — Universidade Federal do Rio de Janeiro, 2014. Citado 2 vezes nas páginas 20 e 22.

DEY, N. *Cross-Platform Development with Qt 6 and Modern C++*. Birmingham, England: Packt Publishing, 2021. Citado na página 27.

GOLUB, G. H.; HOFFMAN, A.; STEWART, G. W. A generalization of the eckart-young-mirsky matrix approximation theorem. *Linear Algebra and its Applications*, v. 88-89, p. 317–327, 1987. Citado na página 20.

GOLUB, G. H.; LOAN, C. F. V. *Matrix Computations*. 4th. ed. Baltimore, MD: Johns Hopkins University Press, 2013. ISBN 978-1421407944. Citado 2 vezes nas páginas 18 e 20.

KADENIC, M. D.; KOUMADITIS, K.; JUNKER-JENSEN, L. Mastering scrum with a focus on team maturity and key components of scrum. *Information and Software Technology*, Elsevier BV, v. 153, p. 107079, jan. 2023. ISSN 0950-5849. Disponível em: <http://dx.doi.org/10.1016/j.infsof.2022.107079>. Citado na página 32.

KEFALLONITIS, F. Name mangling demystified. *int 0x80, White Paper*, 2007. Citado na página 29.

KREYSZIG, E. *Advanced Engineering Mathematics*. 10. ed. Hoboken, NJ: Wiley, 2011. Citado na página 16.

LEON, S. J. *Álgebra Linear com Aplicações*. 9. ed. Rio de Janeiro: LTC, 2018. Citado 2 vezes nas páginas 18 e 19.

MARCUS, A. Graphical user interfaces. In: _____. *Handbook of Human-Computer Interaction*. Elsevier, 1997. p. 423–440. ISBN 9780444818621. Disponível em: <http://dx.doi.org/10.1016/B978-044481862-1.50085-6>. Citado na página 27.

MEYER, C. D. *Matrix Analysis and Applied Linear Algebra*. Society for Industrial and Applied Mathematics (SIAM), 2000. ISBN 978-0-89871-454-8. Disponível em: <https://doi.org/10.1137/1.9780898719505>. Citado na página 23.

OPPENHEIM, A. V.; SCHAFER, R. W. *Discrete-Time Signal Processing*. 3. ed. Pearson, 2010. ISBN 978-0-13-198842-2. Disponível em: <https://books.google.com/books/about/Discrete_Time_Signal_Processing.html?id=EaMuAAAAQBAJ>. Citado 2 vezes nas páginas 15 e 20.

OVERSCHEE, P. V.; MOOR, B. D. *Subspace Identification for Linear Systems: Theory - Implementation - Applications*. Boston, Dordrecht, London: Kluwer Academic Publishers, 1996. ISBN 9780792337581. Citado na página 25.

PROAKIS, M. *Digital signal processing*. 3. ed. Upper Saddle River, NJ: Pearson, 1995. Citado na página 16.

PéREZ, M. A.; SIBERT, J. L. Focus in graphical user interfaces. In: *Proceedings of the 1st international conference on Intelligent user interfaces - IUI '93*. ACM Press, 1993. (IUI '93), p. 255–257. Disponível em: <http://dx.doi.org/10.1145/169891.170022>. Citado na página 27.

RANE, R.; PANDEY, A.; KAZI, F. Real-time electromechanical mode identification through energy-sorted matrix pencil method. In: *2021 56th International Universities Power Engineering Conference (UPEC)*. [S.l.: s.n.], 2021. p. 1–6. Citado 2 vezes nas páginas 22 e 23.

ROBERTS, M. J. *Fundamentals of Signals and Systems*. 1. ed. New York: McGraw-Hill, 2008. Citado 2 vezes nas páginas 15 e 16.

SARKAR, T.; PEREIRA, O. Using the matrix pencil method to estimate the parameters of a sum of complex exponentials. *IEEE Antennas and Propagation Magazine*, v. 37, n. 1, p. 48–55, 1995. Citado 2 vezes nas páginas 22 e 23.

STEIN, E. M.; SHAKARCHI, R. *Complex Analysis*. [S.l.]: Princeton University Press, 2003. v. 2. (Princeton Lectures in Analysis, v. 2). ISBN 978-0691113852. Citado na página 16.

STRANG, G. *Linear Algebra and Its Applications*. 4. ed. [S.l.]: Cengage Learning, 2006. ISBN 978-0030105678. Citado 2 vezes nas páginas 18 e 19.

ZHIRKOV, I. Compilation pipeline. In: _____. *Low-Level Programming: C, Assembly, and Program Execution on Intel® 64 Architecture*. Berkeley, CA: Apress, 2017. p. 63–90. ISBN 978-1-4842-2403-8. Disponível em: <https://doi.org/10.1007/978-1-4842-2403-8_5>. Citado na página 29.

ZHIRKOV, I. Shared objects and code models. In: _____. *Low-Level Programming: C, Assembly, and Program Execution on Intel® 64 Architecture*. Berkeley, CA: Apress, 2017. p. 291–325. ISBN 978-1-4842-2403-8. Disponível em: <https://doi.org/10.1007/978-1-4842-2403-8_15>. Citado na página 29.

Apêndices

APÊNDICE A – Tutorial de Criação de Plugins

Este tutorial apresenta o processo de criação de um *plugin* para a aplicação, utilizando como exemplo um método simples que escala o sinal original por um fator configurável. O objetivo é demonstrar, de forma prática, como implementar um *plugin*, configurar seus parâmetros e integrá-lo à aplicação.

O *plugin* desenvolvido, chamado `ScalePlugin`, multiplica o valor de cada ponto do sinal original por um fator de escala definido pelo usuário como um parâmetro. O parâmetro será inicializado com um valor padrão de 1.0 e permitirá valores entre 0.1 e 2.0, com incrementos de 0.01.

A implementação do *plugin* é apresentada no Código 8. No construtor, o parâmetro `scale factor` é configurado com seus valores padrão e limites. O método `approximate()` percorre o sinal original, aplicando o fator de escala a cada ponto, e retorna o sinal resultante. As funções `create()` e `destroy()` são utilizadas para gerenciar o ciclo de vida do *plugin*.

Para compilar o *plugin* é necessário gerar uma biblioteca dinâmica. No Linux, o comando seria `g++ -shared -fPIC -o libScalePlugin.so ScalePlugin.cpp`, enquanto no Windows o comando deve ser ajustado para gerar um arquivo `.dll`. Certifique-se de incluir o diretório onde o arquivo `ApproximationMethod.h` está localizado, utilizando a *flag* `-I`. Após a compilação, copie o arquivo gerado (`ScalePlugin.so` ou `ScalePlugin.dll`) para o diretório de *plugins* indicado no arquivo de configurações da aplicação.

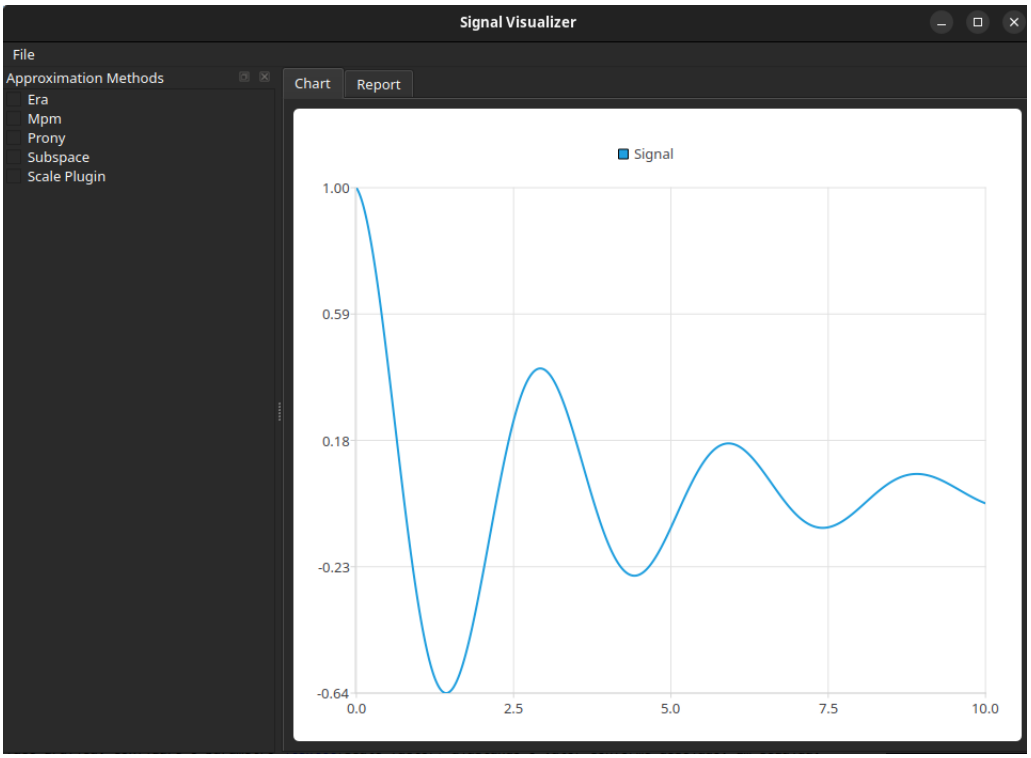
Com o *plugin* instalado, abra a aplicação e verifique se o método "`Scale Plugin`" aparece na lista de métodos disponíveis (Figura 18). Na interface gráfica, configure o parâmetro `scale factor` ajustando o valor conforme desejado. Em seguida, execute o método e observe o sinal escalado no gráfico. O resultado deve refletir o sinal original multiplicado pelo fator configurado, como mostra a Figura 19.

Código 8: Código fonte do *plugin* `ScalePlugin`

```cpp
#include "Plugin.h"

class ScalePlugin : public ApproximationMethod {
public:
  ScalePlugin() {
    parameters.emplace_back("scale factor", 1.0,
                            RangeMetadata<double>{0.1, 2.0, 0.01});
  }

  ApproximationResult approximate(const Signal &input) override {
    double scaleFactor =
        getParameterByName("scale factor").getValueAs<double>();
    Signal scaledSignal;
    for (const auto &[x, y] : input) {
      scaledSignal.emplace_back(x, y * scaleFactor);
    }
    Errors errors = Errors::fromSignals(scaledSignal, input);
    return {scaledSignal, {errors, {}}};
  }

  std::string_view getName() const override { return "Scale Plugin"; }
};

extern "C" ApproximationMethod *create() { return new ScalePlugin(); }

extern "C" void destroy(ApproximationMethod *plugin) { delete plugin; }
```

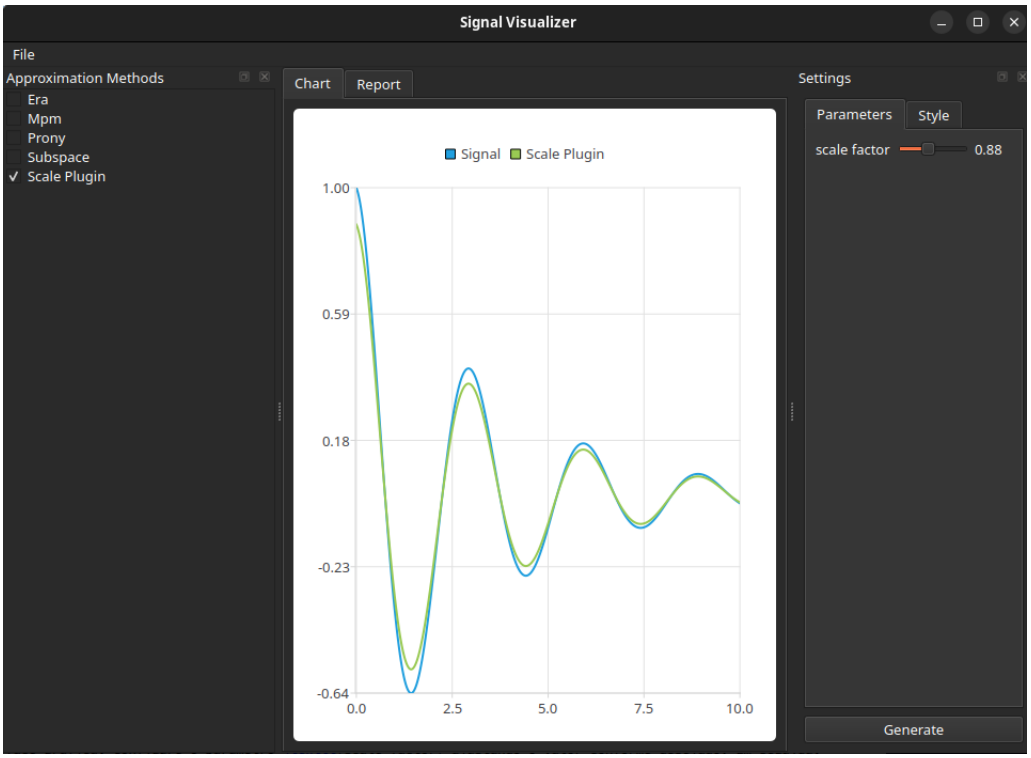
Fonte: os Autores.

Figura 18 – Lista de métodos disponíveis com Scale Plugin entre eles.



Fonte: os Autores.

Figura 19 – Curva gerada pelo método Scale Plugin.



Fonte: os Autores.