UNIVERSIDADE FEDERAL DO PARÁ
INSTITUTO DE TECNOLOGIA
FACULDADE DE ENGENHARIA DA COMPUTAÇÃO E TELECOMUNICAÇÕES
CURSO DE ENGENHARIA DA COMPUTAÇÃO

MERCEDES MARIA BARBOSA DINIZ

AQUISIÇÃO E COMPRESSÃO DE SINAIS DE CORRENTE UTILIZANDO DISPOSITIVOS IoT

BELÉM-PA
2024

UNIVERSIDADE FEDERAL DO PARÁ
INSTITUTO DE TECNOLOGIA
FACULDADE DE ENGENHARIA DA COMPUTAÇÃO E TELECOMUNICAÇÕES
CURSO DE ENGENHARIA DA COMPUTAÇÃO

MERCEDES MARIA BARBOSA DINIZ

AQUISIÇÃO E COMPRESSÃO DE SINAIS DE CORRENTE UTILIZANDO DISPOSITIVOS IoT

Trabalho de Conclusão de Curso apresentado à Faculdade de Engenharia da Computação e Telecomunicações da Universidade Federal do Pará, como requisito parcial para a obtenção do Grau de Bacharel em Engenharia da Computação.

Orientador: Prof. Dr. Leonardo Lira Ramalho

BELÉM-PA
2024

Mercedes Maria Barbosa Diniz

AQUISIÇÃO E COMPRESSÃO DE SINAIS DE CORRENTE UTILIZANDO DISPOSITIVOS IoT

Trabalho de Conclusão de Curso apresentado à Faculdade de Engenharia da Computação e Telecomunicações da Universidade Federal do Pará, sendo julgada adequada para a obtenção do Grau de Bacharel em Engenharia da Computação.

Data da Defesa: 30 de Outubro de 2024

BANCA EXAMINADORA

Prof. Dr. Leonardo Lira Ramalho

(Orientador – FCT/ITEC/UFPA)

Prof. Dr. Adalbery Rodrigues Castro

(Membro - FCT/ITEC/UFPA)

Prof. Dr. Ilan Sousa Correa

(Membro - FCT/ITEC/UFPA)

Prof. Dr. Ilan Sousa Correa

(Diretor da FCT/ITEC/UFPA)

Dedico este trabalho à Rosineide de Oliveira Diniz, minha mãe e fonte de inspiração, cuja crença inabalável em mim me mostrou que sou capaz de alcançar muito mais do que um dia julguei possível.

AGRADECIMENTOS

Primeiramente, agradeço a Deus, ou a qualquer entidade cósmica que se importe com a minha existência, pela oportunidade de viver.

Aos meus pais, expresso meus mais sinceros agradecimentos. À minha mãe, por seu amor e cuidados constantes; e ao meu pai, por sempre me incentivar a buscar conhecimento e a perseguir meus objetivos.

Aos meus amigos, sou grato por tornarem essa jornada mais leve e divertida. Um agradecimento especial a Pedro Rendeiro, Caio Brasil, Madjer Costa, cujo as ideias foram de grade ajuda para superar os desafios da pesquisa. Agradeço também a Frank Bruno e Paulo Anjos, por sempre estarem dispostos a me ouvir manifestar meu descontentamento e me oferecer apoio.

Também quero expressar minha gratidão ao meu orientador, Leonardo Ramalho, pela paciência e dedicação em me ensinar o que, muitas vezes eu já deveria saber, mas esqueci. Sua orientação foi fundamental para a conclusão deste trabalho.

Agradeço ainda ao INESC P&D Brasil e à ENEVA S.A. pelo financiamento deste projeto, através do GImpSI - Gestão dos Impactos da Salinidade em Isolamentos, no âmbito do Programa de P&D da Agência Nacional de Energia Elétrica (ANEEL), código PD-11278-0001-2021.

“The most exciting phrase to hear in science, the one that heralds the most discoveries, is not "Eureka!"(I found it!) but ’That’s funny...”

(Isaac Asimov)

RESUMO

A manutenção de isoladores de alta tensão em subestações elétricas, especialmente em regiões costeiras, apresenta desafios significativos para a confiabilidade das redes elétricas. A exposição constante a ambientes com alta salinidade contribui para a corrosão e o acúmulo de poluentes nos isoladores, o que pode comprometer a integridade do sistema de transmissão. A corrente de fuga — um fenômeno que ocorre quando há passagem indesejada de corrente elétrica pela superfície dos isoladores devido à contaminação acumulada — é comumente usada para avaliar o nível de poluição nesses equipamentos. O monitoramento continuo desses sinais permite a implementação de medidas preventivas, garantindo a segurança e a eficiência operacional das redes elétricas. Este trabalho aborda o desenvolvimento de um sistema para aquisição e compressão de sinais de corrente de fuga utilizando dispositivos IoT. A solução foi aplicada em um sistema embarcados que coleta dados de sensores instalados em um usina termoelétrica localizada em um ambiente com alta salinidade para monitorar a degradação de isoladores de alta tensão. O objetivo principal é otimizar a transmissão e o armazenamento de dados, prolongando a vida útil dos dispositivos e assegurando a integridade dos dados monitorados. O algoritmo de compressão proposto, baseado na detecção de variação espectral, foi implementado em microcontroladores da família ESP32, utilizando técnicas de compressão com perdas para reduzir o volume de dados transmitidos. Os experimentos realizados validaram o sistema em condições controladas e em campo, destacando uma eficiente compressão dos sinais com baixa perda de qualidade, conforme demonstrado pela taxa de compressão de $38,76$ e NMSE de $-44,56$ dB alcançados em um dos melhores casos testados.

Palavras-chave: Internet of Things, Sistemas Embarcadas, Rede de Sensores, Compressão de Sinais, Corrente de Fuga.

ABSTRACT

The maintenance of high-voltage insulators in electrical substations, especially in coastal regions, presents significant challenges for the reliability of electrical networks. Constant exposure to environments with high salinity contributes to corrosion and the accumulation of pollutants in the insulators, which can compromise the integrity of the transmission system. Leakage current — a phenomenon that occurs when there is an unwanted passage of electric current through the surface of insulators due to accumulated contamination — is commonly used to assess the level of pollution in these equipment. Continuous monitoring of these signals allows the implementation of preventive measures, ensuring the safety and operational efficiency of electrical networks. This work addresses the development of a system for acquisition and compression of leakage current signals using IoT devices. The solution was applied in an embedded system that collects data from sensors installed in a thermoelectric plant located in an environment with high salinity to monitor the degradation of high-voltage insulators. The main objective is to optimize data transmission and storage, extending the life of the devices and ensuring the integrity of the monitored data. The proposed compression algorithm, based on spectral variation detection, was implemented in ESP32 family microcontrollers, using lossy compression techniques to reduce the volume of transmitted data. The experiments performed validated the system under controlled and field conditions, highlighting an efficient signal compression with low quality loss, as demonstrated by the compression ratio of 38.76 and NMSE of -44.56 dB achieved in one of the best tested cases.

Keywords: Internet of Things, Embedded Systems, Sensor Network, Digitization, Signal Compression, Leakage Current.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ACRÔNIMOS

CR *Compression Ratio*. 22, 44–46, 48
CRE *Coefficient of Reconstruction*. 22, 23, 44, 55

DAQ Placa De Aquisição De Dados. 8, 26–28, 36
DFT *Discrete Fourier Transforms*. 8, 18–21, 36, 37, 41

ESDD *Equivalent Salt Deposit Density*. 21
ESP32 ESP32-WROOM-32UE. 27–30, 38, 55

FFT *Fast Fourier Transform*. 8, 18, 19, 24, 35
FPGA *Field Programmable Gate Array*. 24

GImpSI Gestão Dos Impactos Da Salinidade Em Isolamentos. 13, 14, 55
GUI *Graphical User Interface*. 49

IoT *Internet of Things*. 13, 14, 26, 55
ISR *Interrupt Service Routines*. 32, 33

LZW *Lempel-Ziv-Welch*. 24

MCU Microcontrolador. 28

NMSE *Normalized Mean Squared Error*. 9, 22, 44, 46, 54, 55

PCI Placa De Circuito Impresso. 26
POWER Placa De Alimentação. 26, 27
PQ *Power Quality*. 23

RMS *Root Mean Square*. 21
ROC *Receiver Operating Characteristic*. 23
RTE *Percentage of Retained Energy*. 9, 22, 23, 44, 54, 55

SNR *Signal-to-noise Ratio*. 23
SPI *Serial Peripheral Interface*. 32
STM32 STM32F103C8T6. 27, 28, 31
SVDA *Spectral Variation Detection Algorithm*. 35, 44, 46

SVDCS *Spectral Variation Detection and Compression System*. 10, 24, 46

TDF Transformada Discreta De Fourier. 55

TF Transformada De Fourier. 16–18

THD *Total Harmonic Distortion* . 21

UML *Unified Modeling Language*. 29

UMSCF Unidade Microprocessada Para Sensores De Corrente De Fuga. 9, 14, 26–28, 33, 35–38, 42, 45, 46, 48, 52, 55

UTEPSI Usina Termoelétrica Porto De Sergipe I. 22, 35, 42

SUMÁRIO

1 INTRODUÇÃO

O crescimento das tecnologias de *Internet of Things* (IoT) tem impulsionado a criação de soluções inovadoras para o monitoramento de sistemas de energia, sendo essenciais para otimizar a operação e manutenção de redes elétricas. Dispositivos embarcados são amplamente utilizados para a coleta e processamento dos dados gerados por sensores, desempenhando um papel crucial na infraestrutura de monitoramento. Entretanto, esses dispositivos enfrentam limitações de recursos computacionais e de armazenamento, tornando necessária a implementação de técnicas eficientes para processar e comprimir os dados, de forma a otimizar o gerenciamento desses recursos.

Diversos estudos têm explorado o uso de dispositivos IoT e redes de sensores em sistemas de monitoramento de energia. Em [1], uma plataforma foi desenvolvida para o monitoramento de redes inteligentes, com o objetivo de melhorar a eficiência operacional. De forma semelhante, [2] apresentou um sistema de detecção de falhas em redes de distribuição de energia usando dispositivos autossuficientes, demonstrando que esses sistemas podem detectar e localizar falhas rapidamente, minimizando o impacto de interrupções. Esses estudos destacam o potencial das soluções IoT para monitoramento em redes de grande escala, mesmo em condições de recursos limitados. Neste contexto, o uso de técnicas de compressão é uma alternativa promissora para reduzir o volume de dados transferidos e armazenados, sem comprometer a qualidade das informações coletadas [3].

As técnicas de compressão aplicadas a dispositivos IoT são abordadas por autores como [4], que propõem uma estrutura de compressão de dados para reduzir o consumo energético em dispositivos de monitoramento, e [5], que compara métodos de compressão sem perdas para sistemas IoT de baixo custo e baixo consumo energético. Por sua vez, [6] apresenta um estudo comparativo das técnicas de compressão aplicáveis ao contexto de IoT, explorando os tipos de compressão, suas limitações e as adequações para diferentes aplicações. A compressão de dados surge, portanto, como uma solução essencial para lidar com os desafios de armazenamento e transmissão de dados em sistemas IoT.

Nesse contexto, este trabalho é desenvolvido no âmbito do projeto Gestão dos Impactos da Salinidade em Isolamentos (GImpSI) — uma colaboração entre o INESC P&D Brasil, a Agência Nacional de Energia Elétrica (ANEEL), a Eneva S.A. e diversas instituições de ensino superior, incluindo a Universidade Federal do Pará (UFPA). O projeto tem como objetivo criar soluções para mitigar os efeitos da salinidade nos isoladores de subestações de alta tensão localizadas em regiões costeiras. A exposição à salinidade causa degradação dos isoladores, o que pode levar à falha em sistemas de transmissão de energia. O monitoramento contínuo das correntes de fuga que fluem pelos isoladores é um método eficaz para avaliar o nível de poluição e o estado dos isoladores, auxiliando na tomada de decisões sobre a manutenção preventiva, como a lavagem dos isoladores, e prevenindo falhas que podem comprometer a segurança e a

confiabilidade do sistema de energia [7, 8].

Pesquisas anteriores realizadas no projeto GImpSI abordaram diferentes aspectos relacionados ao sistema monitoramento das correntes de fuga em subestações. Em [9], foi desenvolvida a arquitetura da rede de sensores para esse monitoramento utilizando dispositivos IoT, denominado Unidades Microprocessadas para Sensores de Corrente de Fuga (UMSCFs), enquanto [10] e [11] apresentaram sistemas de medição de correntes de fuga que auxiliaram na caracterização desses sinais em condições reais de operação. Este estudo complementa esses esforços ao focar na compressão dos dados coletados por essas unidades, utilizando um algoritmo de compressão com perdas baseado na análise espectral. A compressão dos dados é especialmente relevante devido às limitações características dos dispositivos IoT, mencionadas anteriores.

O sistema proposto neste trabalho é capaz de condicionar, digitalizar e comprimir os sinais de corrente de fuga capturados pelas UMSCFs. A principal contribuição deste estudo é a adaptação de um algoritmo de compressão, que foi implementado em dispositivos de baixo custo e capacidade computacional e de armazenamento limitadas, como os microcontroladores da família ESP32. Este sistema visa garantir uma gestão eficiente dos dados de corrente de fuga, otimizando a transmissão e armazenamento de grandes volumes de dados para análise posterior.

1.1 Escopo, Objetivos e Contribuição

O escopo deste trabalho envolve o desenvolvimento de um sistema embarcado capaz de realizar a compressão dos sinais de corrente de fuga coletados por sensores instalados em isoladores de subestações de alta tensão. O objetivo principal é adaptar algoritmos de compressão para diminuir a necessidade de armazenamento para esses sinais, mantendo a integridade e a qualidade dos dados após a compressão. O estudo foca nos seguintes objetivos específicos:

- Investigar um método de compressão adequado para sinais de corrente de fuga;
- Avaliar a eficiência do algoritmo de compressão em termos de taxa de compressão e qualidade de reconstrução do sinal, utilizando métricas como o Erro Médio Quadrático (NMSE, do inglês *Normalized Mean Squared Error*), retenção de energia (RTE) e a correlação cruzada (COR);
- Adaptar a complexidade do algoritmo de compressão para implementação no sistema embarcado;
- Integrar o algoritmo de compressão nos sistemas de monitoramento desenvolvidos anteriormente no projeto GImpSI, garantindo uma solução que possa ser implantada em campo;
- Implementar a o algoritmo de reconstrução em uma linguagem compatível com o servidor da aplicação.

Além disso, um artigo baseado neste estudo foi submetido ao Workshop on Communication Networks and Power Systems 2024 (WCNPS'24), com o título "Spectral Variation-Based Compression: A Case Study with Leakage Current Monitoring". Este trabalho busca divulgar os resultados preliminares obtidos com a implementação do algoritmo de compressão e discutir suas aplicações em sistemas de monitoramento de redes elétricas.

1.2 Estrutura do Trabalho

Este trabalho está organizado em capítulos que guiam o leitor através dos fundamentos teóricos, metodologias e resultados alcançados. O Capítulo 2 revisa os conceitos essenciais que embasam o desenvolvimento deste estudo, abordando o processo de digitalização dos sinais, a análise espectral através da Transformada Discreta de Fourier (TDF), e o levantamento das principais técnicas de compressão baseadas em transformadas aplicadas a sinais elétricos. No Capítulo 3, detalha-se o processo completo de condicionamento, digitalização, compressão e descompressão dos sinais de corrente de fuga, discutindo as etapas práticas e adaptações feitas no sistema embarcado. O Capítulo 4 apresenta os experimentos conduzidos e os resultados obtidos, analisando a eficiência do sistema proposto. Finalmente, o Capítulo 5 encerra o trabalho com as conclusões e aponta direções para melhorias futuras. Nos Apêndices A e B, disponibilizam-se os códigos-fonte utilizados para a compressão e descompressão, implementados em C/C++, Python e Matlab, oferecendo ao leitor a possibilidade de replicar ou estender os experimentos descritos.

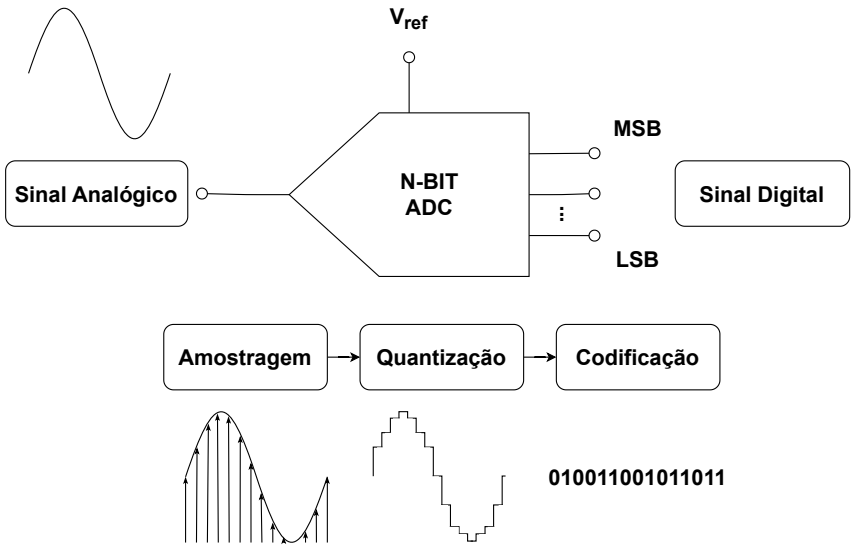
2 FUNDAMENTOS TEÓRICOS

Este capítulo apresenta os conceitos fundamentais que sustentam o desenvolvimento deste trabalho. Inicialmente, discute-se a digitalização de sinais, com ênfase no processo de amostragem. Em seguida, a Transformada Discreta de Fourier é detalhada, ressaltando sua importância para a análise de sinais no domínio da frequência. São caracterizados os sinais de interesse, no caso, os sinais de corrente de fuga, e descritos métodos de compressão por transformada encontrados na literatura, com foco na explicação do método de referência utilizado neste estudo.

2.1 Digitalização de Sinais

Os sinais podem ser representados matematicamente como funções de uma ou mais variáveis independentes. Os sinais de tempo contínuo têm como variável independente uma variável contínua, ou seja, são definidos em um conjunto contínuo de valores. Por outro lado, os sinais de tempo discreto são definidos apenas em instantes discretos, onde a variável independente assume um conjunto de valores discretos [12].

Figura 1 – Processo de transformação de um sinal analógico de tempo contínuo para um sinal digital de tempo discreto.



Fonte: A autora.

Sob certas condições ideais, um sinal de tempo contínuo pode ser completamente representado por seus valores ou amostras uniformemente espaçadas no tempo, como ilustrado no primeiro dos processos listados na Fig.1. Se um sinal for limitado em banda, ou seja, se sua Transformada de Fourier (TF) for nula fora de um intervalo finito de frequências, e se as amostras forem tomadas suficientemente próximas em relação à frequência mais alta presente

no sinal, então essas amostras especificam unicamente tal sinal, permitindo sua reconstrução perfeita. Esse princípio é conhecido como teorema da amostragem [12].

Na prática, entretanto, sinais com componentes de frequência absolutamente nula não existem, o que significa que nenhum sinal obedece rigorosamente às condições do teorema. Além disso, o filtro ideal de reconstrução também não pode ser implementado em sistemas reais, o que impede uma reconstrução perfeita. Apesar dessas limitações, o desvio das condições ideais introduz apenas ruídos ou distorções leves, geralmente aceitáveis na maioria das aplicações.

A amostragem é um processo fundamental na digitalização de sinais, pois permite a conversão de um sinal contínuo em um sinal discreto. No entanto, para que esses sinais possam ser processados por sistemas digitais, é necessário dois passos adicionais chamados quantização e codificação, sendo ilustrado na Fig.1 o resultado esperado após nessas etapas serem realizadas.

A quantização é crucial porque sistemas digitais utilizam um número limitado de bits para representar números. Um quantizador mapeia valores de entrada de um conjunto (eventualmente com um número infinito de elementos) para um conjunto menor com um número finito de elementos [13]. Em outras palavras, a quantização envolve a aproximação dos valores amostrados para um conjunto finito de níveis discretos. Já a codificação transforma essas amostras quantizadas em um formato adequado para armazenamento ou transmissão.

Para converter um sinal analógico em um formato adequado para processamento digital, os processos de amostragem, quantização e codificação são essenciais. Na Fig. 1 resume de forma simplificada cada um desses processos, com a amostragem capturando os valores do sinal em intervalos regulares de tempo, enquanto a quantização e a codificação aproximam esses valores para uma representação discreta que pode ser manipulada por sistemas digitais.

2.2 Transformada Discreta de Fourier

Transformadas lineares são ferramentas matemáticas que permitem a conversão de sinais ou funções de um domínio para outro, associando um vetor a outro vetor enquanto preservam a informação. Um exemplo comum de transformada linear é a TF, que converte sinais representados no domínio do tempo para o domínio da frequência.

Um sinal discreto $x[n]$, definido como $x : \{1,2,...,N-1\} \rightarrow \mathbb{R}$, pode ser decomposto em uma série de funções base $b_k[n]$, da seguinte forma:

$$x[n] = \sum_{k=0}^{N-1} \hat{X}[k].b_k[n], \tag{2.1}$$

onde $\hat{X}[k]$ são os coeficientes transformados que representam o sinal no novo domínio.

Quando as funções-base são ortogonais, a decomposição é particularmente eficiente. No caso da Transformada Discreta de Fourier (DFT, do inglês *Discrete Fourier Transforms*), as funções base são senos e cossenos, ou exponenciais complexas, que são ortogonais.

A DFT é uma versão discretizada da TF, apropriada para sinais digitais e sistemas computacionais. A DFT de um sinal $x[n]$ é definida como:

$$X[k] = \sum_{n=0}^{N-1} x[n].e^{-j\left(\frac{2\pi}{N}\right)kn} \text{ para, } k = 0, \cdots, N-1, \tag{2.2}$$

onde os coeficientes $X[k]$ representam a amplitude e fase das componentes de frequência do sinal original. Para reconstruir o sinal original $x[n]$ a partir dos coeficientes $X[k]$, utiliza-se a DFT inversa:

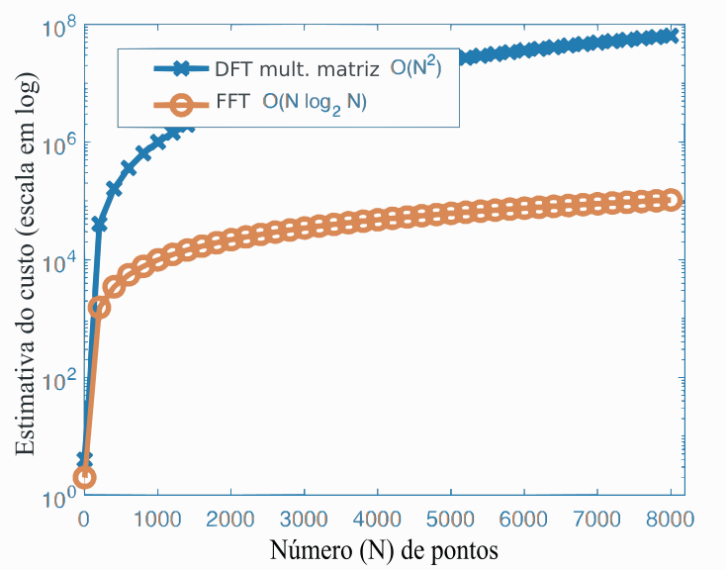
$$x[n] = \frac{1}{N} \sum_{k=0}^{N-1} X[k].e^{j\left(\frac{2\pi}{N}\right)kn} \text{ para, } n = 0, \cdots, N-1. \tag{2.3}$$

A DFT e sua inversa são ferramentas fundamentais no processamento de sinais, permitindo a análise e manipulação de sinais em sistemas digitais. Elas são particularmente úteis em aplicações onde é necessário entender o conteúdo de frequência de um sinal, como em telecomunicações, processamento de áudio e análise de sistemas de controle.

2.2.1 Algoritmo Cooley-Tukey

Continuando a discussão sobre as transformadas, a Transformada Rápida de Fourier (FFT, do inglês *Fast Fourier Transform*) é um algoritmo eficiente para calcular a DFT. O método Cooley-Tukey, introduzido em [14], é a abordagem mais amplamente utilizada para sua implementação.

Figura 2 – Custo computacional da DFT por multiplicação de matrizes versus FFT.



Fonte: Adaptado de [13].

A FFT, como é mais conhecida, se destaca por reduzir a complexidade computacional de $O(N^2)$ para $O(N \log N)$, onde N é o número de pontos do sinal. Essa eficiência é crucial para o processamento de grandes conjuntos de dados, na Fig. 2 há a estimativa do custo (relacionado

ao número de produtos internos) para executar esta operação com dois métodos diferentes, em relação a quantidades de pontos do sinal.

O método de Cooley-Tukey implementa uma abordagem recursiva para dividir o cálculo da DFT de um sinal de comprimento N em múltiplas DFTs menores. Inicialmente, o vetor de entrada é separado em dois vetores de comprimento $N/2$. Esse processo de divisão se repete de maneira recursiva, até que o comprimento dos vetores seja igual a 1, tornando os cálculos triviais.

De forma geral, o algoritmo reformula a soma da DFT para dividir o problema em partes menores, aproveitando as simetrias e periodicidades das funções exponenciais. Em cada nível da recursão, o problema é dividido em dois subproblemas de metade do tamanho original. Após calcular essas DFTs menores, os resultados são combinados para formar a DFT do sinal original, resultando em um cálculo eficiente da transformada. Esse processo "divide e conquista"acelera consideravelmente a FFT em comparação com a abordagem direta.

2.2.2 Simetria Hermitiana dos Sinais Reais

Ao aplicar a DFT a um sinal composto apenas por valores reais, os coeficientes resultantes apresentam uma propriedade conhecida como simetria Hermitiana. Em termos simples, isso significa que os coeficientes da DFT apresentam uma relação especial: para cada valor $X[k]$, existe um coeficiente correspondente em $X[N-k]$, que é o conjugado complexo de $X[k]$. Onde, N representa o número total de coeficientes da DFT, e k é o índice do k-ésimo coeficiente, variando de 0 a $N-1$.

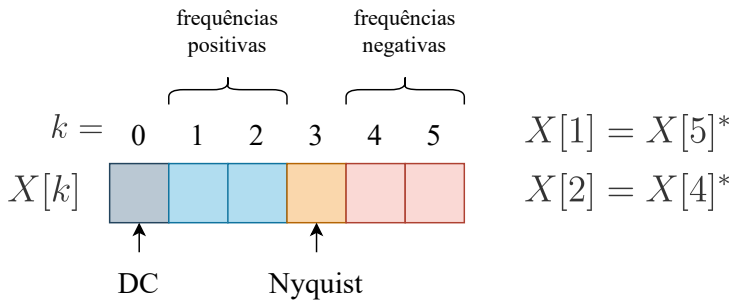
Na Fig. 3, dois exemplos ilustram essa propriedade. O primeiro coeficiente sempre corresponde ao nível DC (componente de frequência zero), enquanto a frequência de Nyquist, que representa a frequência mais alta ($F_s/2$), aparece explicitamente quando o número de coeficientes é par. Essa simetria implica que os coeficientes de frequência negativa são apenas o espelho conjugado dos coeficientes de frequência positiva. Por isso, somente metade dos coeficientes da DFT precisam ser armazenados ou processados, pois o conhecimento de uma metade já permite deduzir a outra.

Essa propriedade é essencial para melhorar a eficiência no processamento de sinais reais, uma vez que reduz o número de cálculos e a quantidade de dados que precisam ser manipulados, economizando tanto memória quanto tempo de processamento.

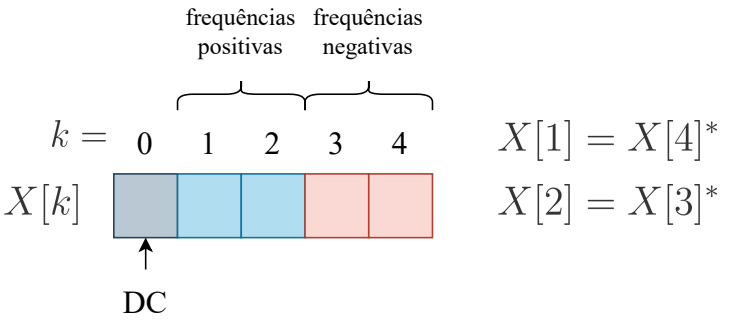
2.2.3 Resolução de Frequência da DFT

A resolução em frequência é um aspecto crucial DFT, pois determina a capacidade de distinguir diferentes componentes de frequência em um sinal. Uma resolução alta permite a separação precisa das componentes de frequência, enquanto uma resolução baixa pode levar à perda de detalhes importantes no espectro.

Figura 3 – Simetria Hermitiana dos coeficiente da transformadas.



(a) Número par de coeficiente.



(b) Número impar de coeficiente.

Fonte: A autora.

A resolução de frequência da DFT é definida como:

$$\Delta\Omega = \frac{2\pi}{N}, \tag{2.4}$$

onde N é o número de pontos. Já a frequência Ω_k correspondente ao k-ésimo coeficiente da transformada e é dado por:

$$\Omega_k = \Delta\Omega k = \frac{2\pi}{N}k \tag{2.5}$$

Dois fenômenos importantes relacionados à este conceito são o *picket-fence effect* e o *leakage spectral*, detalhados em [13]. O primeiro ocorre quando frequências próximas não são distinguidas corretamente, resultando em uma mistura das componentes de frequência. Isso acontece quando a resolução é insuficiente para separar componentes que estão muito próximas umas das outras, causando um "borramento" no espectro.

O *leakage spectral*, por outro lado, ocorre quando a energia de uma frequência se espalha para outras. Este fenômeno é especialmente comum se o sinal não for perfeitamente periódico dentro do intervalo de amostragem, levando a um espectro que mostra energia em frequências onde, teoricamente, não deveria haver. Isso pode distorcer significativamente a interpretação do espectro.

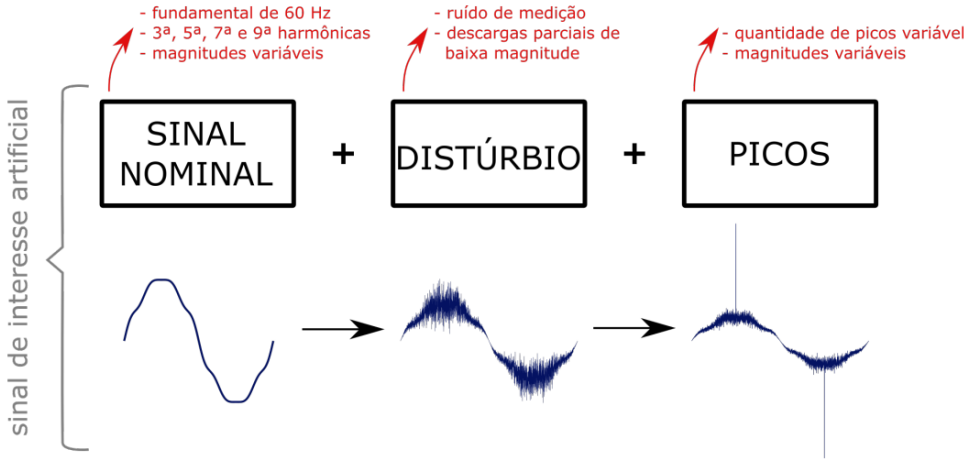
A resolução de frequência determina a menor diferença de frequência que pode ser distinguida na análise espectral. Quando N é pequeno, $\Delta\Omega$ se torna grande, resultando em uma

perda de detalhes finos no espectro de frequência e aumentando a probabilidade de *picket-fence effect* e *leakage spectral*. Para garantir uma boa resolução de frequência e evitar a distorção do sinal na DFT inversa, é essencial que N seja suficientemente grande para capturar os detalhes do sinal.

2.3 Aquisição de Sinais de Corrente de Fuga em Isoladores

O sinal de interesse neste trabalho é o de corrente de fuga, que se refere à um fenômeno que ocorre quando há passagem indesejada de corrente elétrica em um circuito. O mesmo é comum ao longo da superfície dos isoladores, especialmente sob condições de poluição e alta umidade. Esses sinais, que tipicamente tem uma magnitudes relativamente pequenas (inferiores a 10 mA), possuem como componentes principais a fundamental em 60 Hz e as terceira e quinta harmônicas, conforme descrito em [11] e [15].

Figura 4 – Ilustração das características do sinal de corrente de fuga.



Fonte: [11].

Além do estudo realizado neste trabalho, a literatura apresenta diversas abordagens para aquisição e simulação de sinais de corrente de fuga, cada uma buscando replicar condições de poluição em isoladores elétricos. Em [11], os sinais artificiais foram gerados com base em um procedimento que simula a variação das componentes harmônicas conforme o grau de poluição do isolador, conforme representado na Fig. 4. Já em [15], os autores realizaram experimentos em uma câmara de névoa para simular diferentes níveis de poluição, aplicando soluções com diferentes *Equivalent Salt Deposit Density* (ESDD) em isoladores de vidro. A corrente de fuga foi então adquirida e analisada em termos de parâmetros específicos, como valor *Root Mean Square* (RMS), valor de pico, *Total Harmonic Distortion* (THD) e componentes harmônicas (fundamental, 3ª, 5ª, 7ª e 9ª), para estimar o nível crítico de poluição. Em [7], foi explorada a influência de contaminantes específicos, como NaCl e KCl, na geração de harmônicas nos sinais de corrente de fuga, destacando a sensibilidade do sinal às diferentes composições químicas.

Esses estudos ressaltam como as características harmônicas da corrente de fuga são amplamente utilizadas para avaliar a gravidade da poluição e o risco de falhas nos isoladores.

Neste trabalho, os sinais de corrente de fuga foram adquiridos de duas maneiras. Primeiramente, utilizou-se um gerador de sinais para simular o comportamento do sensor com uma senoide de 10 mVpp a 60 Hz, com o intuito de criar um sinal de corrente de fuga idealizado. Além disso, também foram coletados sinais diretamente dos sensores de corrente de fuga instalados na Usina Termoelétrica Porto de Sergipe I (UTEPSI), conforme detalhado na Seção 4.1.

2.4 Compressão de Sinais Elétricos Utilizando Transformadas

A compressão é uma operação que pode ser realizada seguindo variados métodos, com o objetivo final de reduzir a quantidade de bytes necessários para representar os dados. Utilizando um conhecimento prévio das características do sinal de origem, a compressão faz uso eficiente de simplificações baseadas nas redundâncias presentes no sinal, sempre preservando, ou no caso de compressão com perdas, minimizando a perda das informações essenciais do sinal original.

Os métodos de compressão por transformada são técnicas que convertem dados no domínio original para um outro que maximize a redundâncias nos dados. A ideia central é que, para alguns sinais a informação relevante tende a se concentrar em um menor número de coeficientes no domínio transformado.

Na Seção 2.3 foi apresentado as características do sinal explorado este trabalho, e na Seção 2.4.2 é discutido algumas técnicas de compressão por transformada aplicadas a sinais elétricos, cujo as métricas utilizadas para avalia-los estão definidas na Seção 2.4.1.

2.4.1 Métricas de Desempenho

Para avaliar o desempenho da maioria dos algoritmos de compressão e reconstrução, foram adotadas métricas que analisam tanto a razão entre o tamanho do arquivo original e sua versão comprimida, quanto a distorção entre o sinal original e o sinal reconstruído. As métricas selecionadas para este trabalho incluem a *Compression Ratio* (CR), o NMSE, a RTE e a *Coefficient of Reconstruction* (CRE). A seguir, são apresentadas as definições e as equações utilizadas para o cálculo dessas métricas, proporcionando uma base objetiva para a análise dos resultados experimentais.

A qualidade do sinal reconstruído foi avaliada por meio do NMSE, definido como:

$$NMSE = \sum_{n=1}^{N_s} |x_n - \hat{x}_n|^2 \,/\, \sum_{n=1}^{N_s} |x_n|^2 . \tag{2.6}$$

Essa métrica quantifica a diferença entre o sinal original e o sinal reconstruído, normalizando o erro quadrático em relação à energia do sinal original. Um NMSE próximo de zero indica uma alta fidelidade na reconstrução do sinal.

Além disso, é utilizado a porcentagem de energia retida (RTE), descrita como:

$$RTE(\%) = 100\% \times \sum_{n=1}^{N_s} \hat{x}_n^2 / \sum_{n=1}^{N_s} x_n^2 \ , \tag{2.7}$$

que mede a quantidade de energia do sinal original preservada no sinal reconstruído, expressa em porcentagem. Valores de RTE próximos de 100% indicam uma preservação eficiente da energia do sinal.

O coeficiente de reconstrução (CRE), introduzido em [16], é dado por:

$$CRE = \left(x(n)^T \cdot \hat{x}(n)\right) / \left(x(n)^T \cdot x(n)\right) \ , \tag{2.8}$$

onde o operador "." é o produto interno entre dois vetores. A mesma avalia a similaridade entre o sinal original e o sinal reconstruído ao longo do tempo. Valores de CRE próximos de 1 indicam uma alta correlação e, portanto, uma boa preservação da estrutura temporal do sinal.

2.4.2 Técnicas de Compressão Presentes na Literatura

Diversas técnicas de compressão de sinais em sistemas elétricos têm sido discutidas na literatura, com ênfase em abordagens baseadas em transformadas. Embora este estudo se concentre nos sinais de corrente de fuga, é importante destacar que grande parte das pesquisas sobre compressão de sinais elétricos está voltada para sinais de *Power Quality* (PQ), ou "qualidade de energia", especialmente sinais com distúrbios que afetam a tensão, corrente ou frequência. Tais distúrbios são mais comuns em redes de energia elétrica de grande porte e, embora compartilhem algumas semelhanças com os sinais de interesse neste trabalho, os desafios específicos para o mesmo ainda não foram amplamente abordados na literatura.

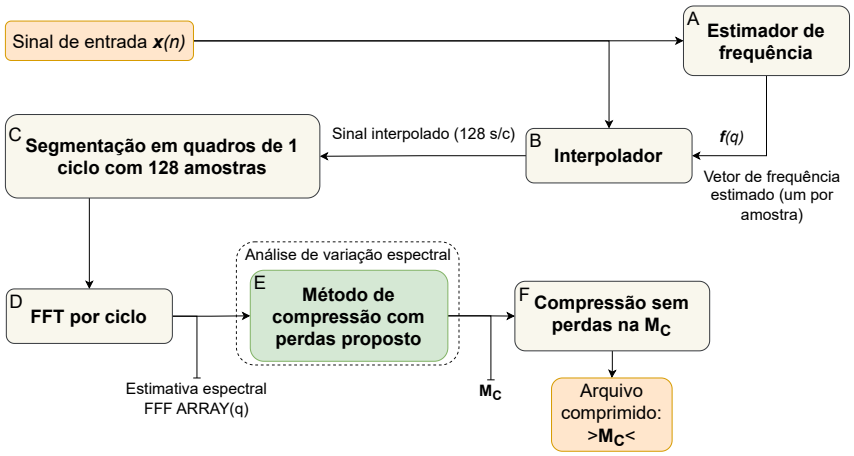
Em [17], foi usando a Wavelet db4 com etapas de quantização, alocação de bits, codificação. Com 8 sub-bandas e 1024 amostras, foi obtido um NMSE de -42,5 dB e uma taxa de compressão de 73,2. Em um estudo semelhante, senso este o [18], foi aplicada a Wavelet symlet 4 com decomposição de 3 níveis. As taxas de compressão variaram entre 1 e 5, destacando a importância de selecionar adequadamente a função wavelet e o número de níveis de decomposição. Em [19], foi apresentado um algoritmo de compressão baseado na Wavelet biortogonal de nível 6, obtendo uma alta taxa de compressão de 99,803%, com um NMSE de 0,000434, uma retenção de energia (RTE) de 99,9479% e correlação cruzada de 0,999925.

Diferenciando-se das abordagens anteriores, [20] aplicou uma métrica de similaridade para detectar eventos em sinais de qualidade de energia no domínio da frequência, avaliando a eficácia pela curva *Receiver Operating Characteristic* (ROC), com uma área sob a curva (AUC) de 0,99128 para um *Signal-to-noise Ratio* (SNR) de 60 dB. Esse estudo serviu de base para [16], que introduziu um sistema de compressão em tempo real com análise de variação espectral.

Embora esses estudos tenham alcançado altas taxas de compressão, todos eles foram implementados no ambiente MATLAB, o que permite um processamento mais flexível e com

mais recursos de hardware. A principal referência para o trabalho atual é [16], que propôs o SVDCS, implementado em um *Field Programmable Gate Array* (FPGA). Neste contexto, o presente trabalho propõe uma versão adaptada do SVDCS apresentado, com a escolha de um método baseado em Transformada de Fourier e da análise de variação espectral motivada pela simplicidade computacional.

Figura 5 – Sistema de compressão de proposto pelo trabalho de referência.



Fonte: Adaptado de [16].

No diagrama da Fig. 5 há o resumo das etapas do algoritmo de referência. Nele observa-se que o sistema estima a frequência fundamental do sinal usando a técnica de cruzamento por zero, proposta em [21], seguida de uma interpolação que garante o número correto de amostras por ciclo fundamental. Cada quadro de sinal é então processado pela FFT para análise no domínio da frequência. O núcleo do algoritmo é a detecção de novidades espectrais, onde quadros consecutivos são comparados para identificar variações. Quando uma mudança significativa é detectada, o quadro é armazenado, o que resulta em compressão. Por fim, após a compressão com perdas, o sistema aplica o algoritmo *Lempel-Ziv-Welch* (LZW) para compressão sem perdas.

Tabela 2 – Testes e resultados do SVDCS.

Características do sinal	**Duração**	**CR**	**NMSE**
Com variados distúrbios	13min	8460	$1{,}74 \times 10^{-4}$
Com distorção harmonica	10min	10780	$1{,}96 \times 10^{-4}$
Com variação na magnitude da rampa	10min	11700	1.39×10^{-4}
Com variação da frequência fundamental	82s	1807	$4{,}48 \times 10^{-4}$
De uma subestação (tensão nominal de 230kV)	1,267s (76 ciclos)	43	$4{,}94 \times 10^{-5}$
De um gerador eólico (tensão)	34h	9470	-
De um gerador eólico (corrente)	34h	5493	-

Em [16], o SVDCS foi testado em sinais de diferentes durações e com variados distúrbios, amostrados em frequências superiores às utilizadas neste trabalho. Para fins de comparação, a Tabela 2 resume as características dos sinais de entrada e os resultados obtidos. Nela, observa-se

que o cenário mais próximo do escopo deste trabalho é o do sinal de curta duração amostrado em uma subestação.

3 METODOLOGIA

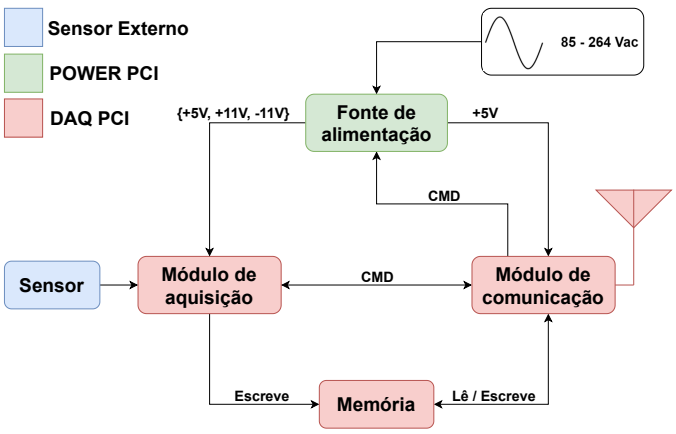
Neste capítulo, é apresentada a metodologia proposta para o processo de condicionamento, digitalização e compressão de sinais de corrente de fuga, implementados no sistema embarcado das UMSCFs. Primeiramente, é descrito o módulo de condicionamento das UMSCFs, seguido pelos detalhes do firmware que executa a digitalização dos sinais. Em seguida, discute-se a compressão baseada na detecção de variação espectral e, por fim, é explicado o processo de reconstrução do sinal comprimido.

3.1 Unidade Microprocessada para Sensores de Corrente de Fuga

As UMSCFs integram um sistema de IoT destinado ao monitoramento de correntes de fuga em isoladores de alta tensão. Este dispositivo é responsável pela coleta de dados provenientes dos sensores e pelo envio dessas informações a um servidor, onde serão analisadas para determinar o momento ideal para a lavagem dos equipamentos da subestação.

A medição da corrente de fuga é a técnica adotada por este sistema para mensurar o nível de poluição na superfície dos isoladores. Uma discussão mais aprofundada sobre a natureza dessa relação e as características do projeto do sensor pode ser encontrada em [11], [10] e [15].

Figura 6 – Visão geral do hardware da Unidade Microprocessada para Sensores de Corrente de Fuga.



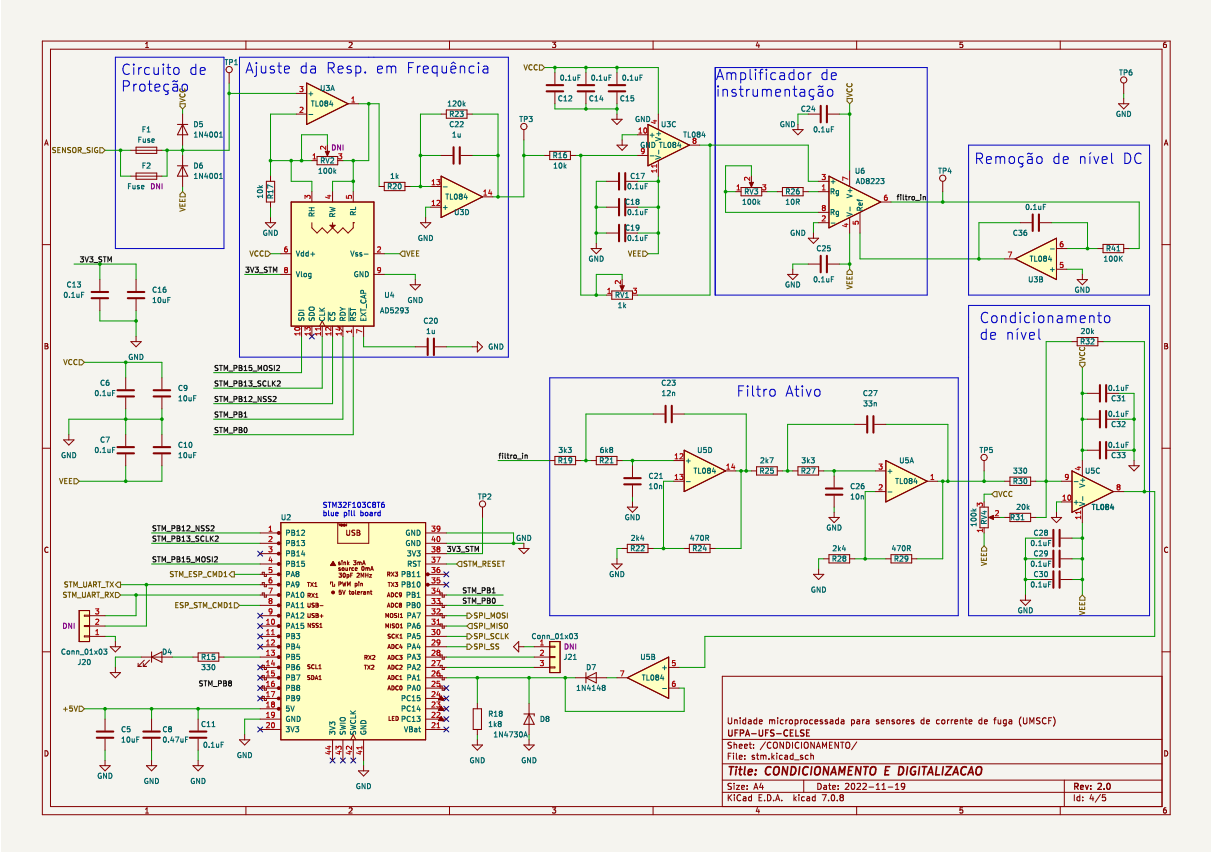
Fonte: Adaptado de [9].

O sistema completo da UMSCF é composto por duas Placa de Circuito Impressos (PCIs): uma plataforma de aquisição e comunicação de dados, e outra responsável pela alimentação da UMSCF, denominadas respectivamente, DAQ e Placa de Alimentação (POWER), como mostrado na Fig. 6. A DAQ integra dois módulos principais: o de aquisição de sinal e o de comunicação. Entre esses módulos, há um cartão de memória microSD, acessível por ambos os módulos.

O módulo de aquisição inclui o circuito de condicionamento do sinal e um microcontrolador STM32F103C8T6 (STM32), que é responsável pela digitalização e pelo pré-processamento do sinal. Já o módulo de comunicação utiliza um ESP32-WROOM-32UE (ESP32), que realiza o controle geral do sistema e a transmissão dos dados. A placa POWER, por sua vez, é responsável por gerar as tensões de alimentação DC necessários para a operação da DAQ, a partir de uma fonte AC (100V - 240V).

Atualmente, as UMSCFs possuem duas versões já produzidas: a primeira versão, abordada em [9], e a nova atualização discutida neste trabalho. As modificações no hardware concentram-se no circuito de condicionamento do módulo de aquisição da DAQ, com a nova do esquemático apresentada na Fig. 7. Uma adição importante na versão atual é o uso de um potenciômetro digital AD5293, recurso que não estava presente anteriormente. Esse componente agora desempenha um papel essencial no ajuste dinâmico do ganho durante o ciclo de amostragem, processo detalhado na Seção 3.2.1.

Figura 7 – Versão atualizada do módulo de condicionamento da plataforma de aquisição.



Fonte: A autora.

O circuito de condicionamento foi projetado para garantir a integridade e a qualidade do sinal proveniente dos sensores de corrente de fuga. Inicialmente, há uma proteção composta por um fusível e um diodo 1N4001, que atuam para impedir que tensões excessivas oriundas do sensor danifiquem o restante do circuito. O primeiro estágio do circuito é responsável por

ajustar a resposta em frequência e ajustar o ganho automático controlado dinamicamente pelo potenciômetro digital, que é regulado pelo STM32. Em seguida, o sinal passa por um amplificador de instrumentação AD8223, cujo ganho é ajustado manualmente, seguido da remoção do nível DC.

Após essa etapa, o sinal é filtrado para garantir que apenas as frequências de interesse sejam preservadas, e o nível do sinal é ajustado para estar dentro da faixa dinâmica do conversor ADC do STM32. Antes da digitalização, há uma segunda proteção com um diodo 1N4148, seguida por um amplificador que compensa a queda de tensão introduzida pelo diodo, assegurando que o sinal permaneça dentro dos limites operacionais.

Para um entendimento mais abrangente sobre o projeto e desenvolvimento destas unidades, recomenda-se a leitura de [9], onde são apresentados detalhes técnicos sobre o firmware do módulo de comunicação. Neste trabalho, o foco está na aquisição e compressão dos sinais de corrente de fuga. A seguir, serão apresentados detalhes sobre o firmware do Microcontrolador (MCU) do módulo de condicionamento.

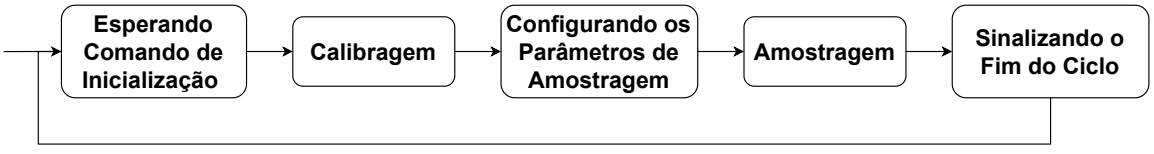
3.2 Digitalização de Sinais de Corrente de Fuga

O núcleo do sistema das UMSCFs reside na DAQ, especialmente no módulo de comunicação (ESP32). Esse módulo, considerando só as etapas relacionadas à digitalização do sinal, é responsável por definir os parâmetros da amostragem, coordenar o início de cada ciclo de amostragem, gerenciar o acesso à memória externa (microSD) para armazenamento temporário das amostras, e interromper o ciclo de amostragem caso o tempo limite seja atingido.

O módulo de condicionamento, por outro lado, encarrega-se de processar os sinais recebidos do sensor, com a Fig.8 ilustrando as etapas executadas. O MCU desse módulo STM32 executa a calibração do ganho, crucial para maximizar o sinal na entrada do ADC do MCU e evitar saturação do mesmo. Além disso, realiza a amostragem do sinal conforme os parâmetros previamente configurados.

Esse processo integrado garante que os sinais de corrente de fuga sejam adequadamente digitalizados e armazenados, permitindo que posteriormente sejam enviados para o servidor que processa os dados coletados pela rede de sensores.

Figura 8 – Visão geral da lógica do firmware do módulo de condicionamento.



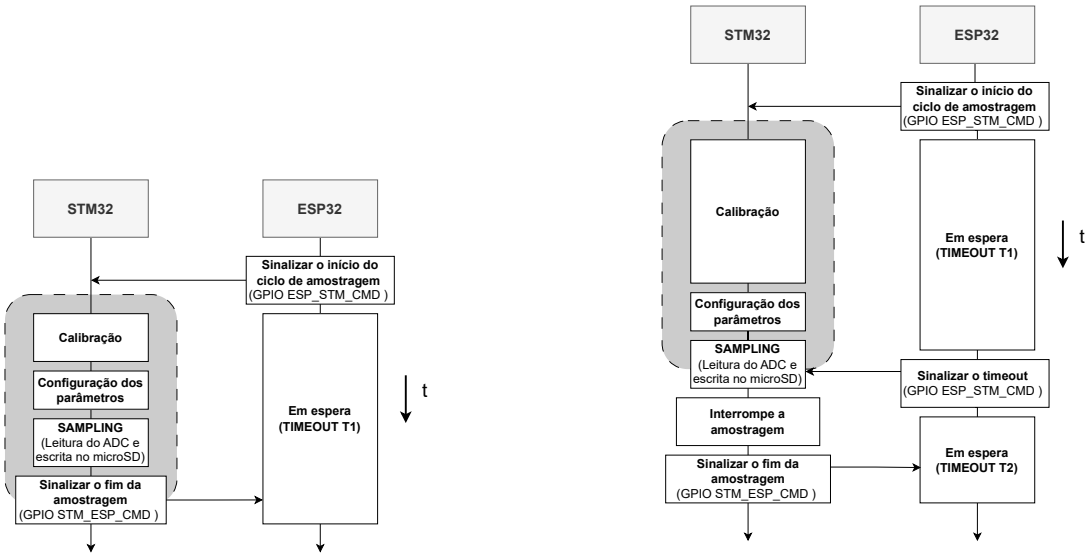
Fonte: A autora.

3.2.1 Ganho Adaptativo do Sinal de Entrada

Para garantir a precisão na digitalização dos sinais de corrente de fuga, o ganho do sinal de entrada precisa ser ajustado adequadamente antes de ser processado pelo ADC do STM32. O sinal do sensor passa por várias etapas de condicionamento, como apresentado do esquemático da Fig. 7, sendo o ajuste ganho automático uma das primeiras etapas, onde a rotina de calibração é executada a cada novo ciclo de digitalização do sinal.

O processo de calibração é baseado no sinal capturado pelo ADC e ocorre antes da leitura das amostras que realmente compõem a versão digital do sinal. Esse ajuste é essencial, pois, se o sinal estiver saturado, as informações contidas nele podem ser interpretadas incorretamente ou até perdidas. A duração desse processo é um fator crítico, pois afeta diretamente o tempo total disponível para a execução das demais rotinas dentro do ciclo de amostragem, conforme ilustrado na Fig. 8. Caso o tempo de calibração seja excessivo, o ciclo de amostragem pode ultrapassar o limite de tempo estabelecido pelo módulo de comunicação, resultando em uma interrupção do processo, conforme indicado na Fig. 9b.

Figura 9 – Impacto do tempo de calibração no ciclo de amostragem.



(a) Amostragem finalizada dentro do timeout estabelecido.

(b) Amostragem interrompida após atingir o timeout.

Fonte: A autora.

O primeiro intervalo de tempo (*timeout 1*), indicado nos diagramas de sequência *Unified Modeling Language* (UML) na Fig. 9, representa uma estimativa do tempo máximo necessário para coletar N_S amostras dado uma frequência de amostragem especifica. Esse valor é calculado pelo módulo de comunicação (ESP32) com base nos parâmetros de amostragem definidos, mais um tempo extra arbitrário, e determina quanto tempo o módulo de comunicação concederá ao módulo de condicionamento para realizar todo o processo de digitalização do sinal e armazenamento

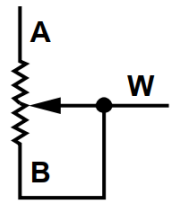
na memória externa.

O segundo intervalo de tempo (*timeout 2*), indicado no diagrama da Fig. 9b, é iniciado após o término do *timeout 1*. Esse período é um tempo fixo concedido ao módulo de condicionamento para finalizar a comunicação SPI com a memória externa. Esse processo é necessário porque ambos os módulos, de comunicação e de condicionamento, não podem acessar a memória externa simultaneamente, pois isso poderia corromper o microSD. Portanto, o módulo de comunicação deve aguardar até que o módulo de condicionamento conclua sua comunicação com a memória antes de assumir o controle. Esse controle de acesso é gerenciado tanto pela lógica da aplicação quanto pelo multiplexador no circuito, que é controlado pelo ESP32.

A lógica por trás deste método reside no fato de que, ao realizar a calibração a cada ciclo de amostragem, o valor do ganho no ciclo subsequente estará mais próximo do ideal para o sinal específico em análise. Dessa forma, o tempo necessário para a calibração seguinte é reduzido, assumindo que não ocorram variações bruscas no sinal entre os ciclos. Dessa forma, o sistema se torna mais eficiente, adaptando-se continuamente às variações do sinal de entrada e garantindo a integridade e precisão dos dados coletados.

Tal processo é realizado através do uso de um potenciômetro digital, o AD5293, presente no circuito. Este componente opera no modo *rheostat* com apenas dois terminais sendo utilizados, conforme mostrado na Fig. 10. O mesmo possui 1024 configurações possíveis de ajuste do terminal do cursor, com o valor de 10 bits armazenado no registrador RDAC, conforme definido em [22].

Figura 10 – Configuração do modo *rheostat*.



Fonte: Adaptado de [22].

No AD5293, a resistência entre o terminal W e o A (R_{WA}), e a resistência entre o terminal W e o B (R_{WB}), são projetadas para garantir um erro máximo absoluto de resistência de ±1% sobre toda a faixa de fornecimento e temperatura. As relações gerais para determinar esses valores são:

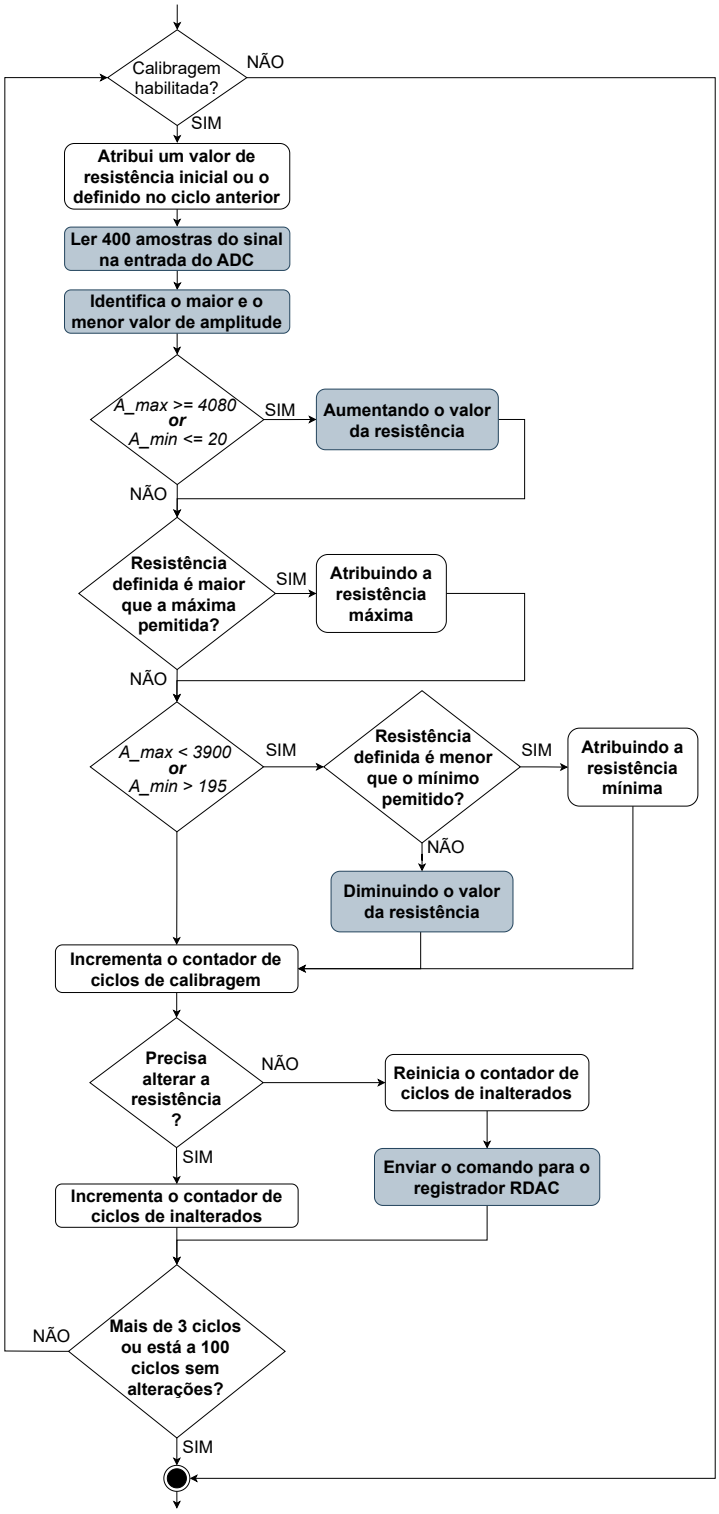
$$R_{WB}(D) = \frac{D}{1024} \times R_{AB} \tag{3.1}$$

e

$$R_{WA}(D) = \frac{1024 - D}{1024} \times R_{AB} \quad , \tag{3.2}$$

onde D é o equivalente decimal do código binário carregado no registrador RDAC e R_{AB} é a resistência nominal entre os terminais A e B, cujo valor nesse projeto é de 100kΩ.

Figura 11 – Diagrama da rotina de calibração.



Fonte: A autora.

O diagrama na Fig. 11 ilustra a lógica do algoritmo de calibração. Durante essa etapa, o ADC do STM32 realiza a leitura de algumas amostras para avaliar se o sinal se encontra dentro de uma faixa de valores desejada. Com base nos valores máximo e mínimo de amplitude do sinal, o algoritmo determina se a resistência do AD5293 precisa ser aumentada ou diminuída.

Esse ajuste é feito variando o valor de D em incrementos de 10, que são escritos no registrador RDAC do potenciômetro digital via comunicação *Serial Peripheral Interface* (SPI).

Para controlar o número de vezes que o ciclo de calibração ocorre dentro de um único ciclo de amostragem, são utilizados algumas variáveis (como o contador de ciclos de calibração e o contador de ciclos sem alteração), especificados no último bloco condicional do diagrama da Fig. 11. Esses contadores asseguram que o módulo de condicionamento não fique indefinidamente preso na calibração caso o sinal de entrada apresente muita variação.

3.2.2 Amostragem Baseada em Interrupções do Temporizador

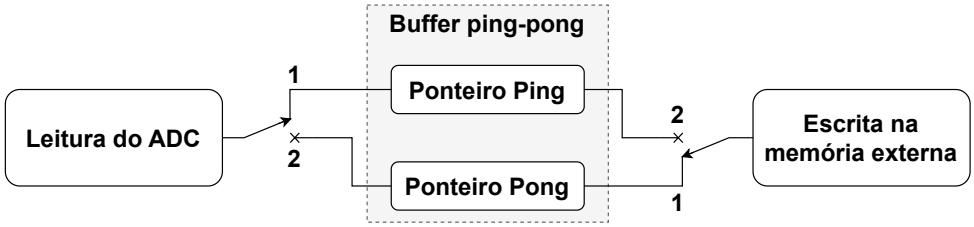
Após a etapa de calibração, conforme indicado no fluxograma da Fig. 8, são verificados os parâmetros de amostragem recebidos juntamente com o comando de inicialização. Esses parâmetros incluem o período de amostragem (T_s) e o número de amostras (N_s) para o ciclo em questão. Caso esses parâmetros não tenham sido alterados, a rotina segue para a etapa de amostragem. Caso contrário, antes de seguir os parâmetros são atualizados no escopo do firmware em execução e a interrupção do temporizador é redefinida com o novo valor do T_s. Com esses valores definidos, a rotina de amostragem lê os valores do ADC a cada T_s segundos até que N_s amostras sejam coletadas, formando a representação digital do sinal.

A estrutura de controle de memória denominada "buffer ping-pong" foi empregada conforme é apresentada na literatura em [23], [24], [25] e [26]. Esta técnica divide o tratamento dos dados em dois grupos: dados em captura e dados em processamento. A ideia é possuir duas áreas de memória de mesmo tamanho, com respectivos ponteiros para o endereço de cada uma dessas áreas. O ponteiro "ping" aponta para o buffer que recebe os dados lidos do ADC até que seja totalmente preenchido. Já o ponteiro "pong" é associado ao buffer que lê a memória reservada previamente, preenchida pelo canal "ping" e realiza a escrita desses dados no cartão de memória.

A Fig. 12 apresenta uma representação dessa política, na qual no estado 1 escreve-se no espaço de memória indicado pelo ponteiro "ping" e lê-se no "pong" e no estado 2 a lógica é invertida. Quando o buffer de leitura do ADC está cheio e os dados no buffer de escrita na memória já foram processados, uma operação de chaveamento é realizada. O canal "ping" passa a apontar para o endereço de memória antes destinada ao canal "pong" e este último passa a apontar para o canal anteriormente utilizado pelo "ping" com novos dados disponíveis para processamento.

Para garantir que o período de amostragem esteja de acordo com o definido, é essencial o uso de interrupções do temporizador. Essa interrupção é utilizada para executar uma função específica, chamada de *Interrupt Service Routines* (ISR), imediatamente quando o temporizador atinge um valor específico, que neste caso é o valor de T_s atribuído de acordo com as limitações

Figura 12 – Política da estrutura de controle de memória, "ping-pong".



Fonte: A autora.

da biblioteca *STM32_TimerInterrupt*[1].

No laço do escopo principal, representado no diagrama da Fig. 13, ocorre a habilitação e desabilitação da interrupção, o chaveamento dos ponteiros dos buffers, o controle da quantidade total de amostras coletadas e o armazenamento dos dados na memória externa. Essas operações são realizadas de forma sincronizada com a ISR para que não haja perda de informações.

A ISR dessa aplicação está definida no diagrama da Fig. 14, onde ocorre a coleta dos dados a partir do ADC e o preenchimento do buffer de leitura.

3.3 Compressão Baseada em Detecção de Variação Espectral

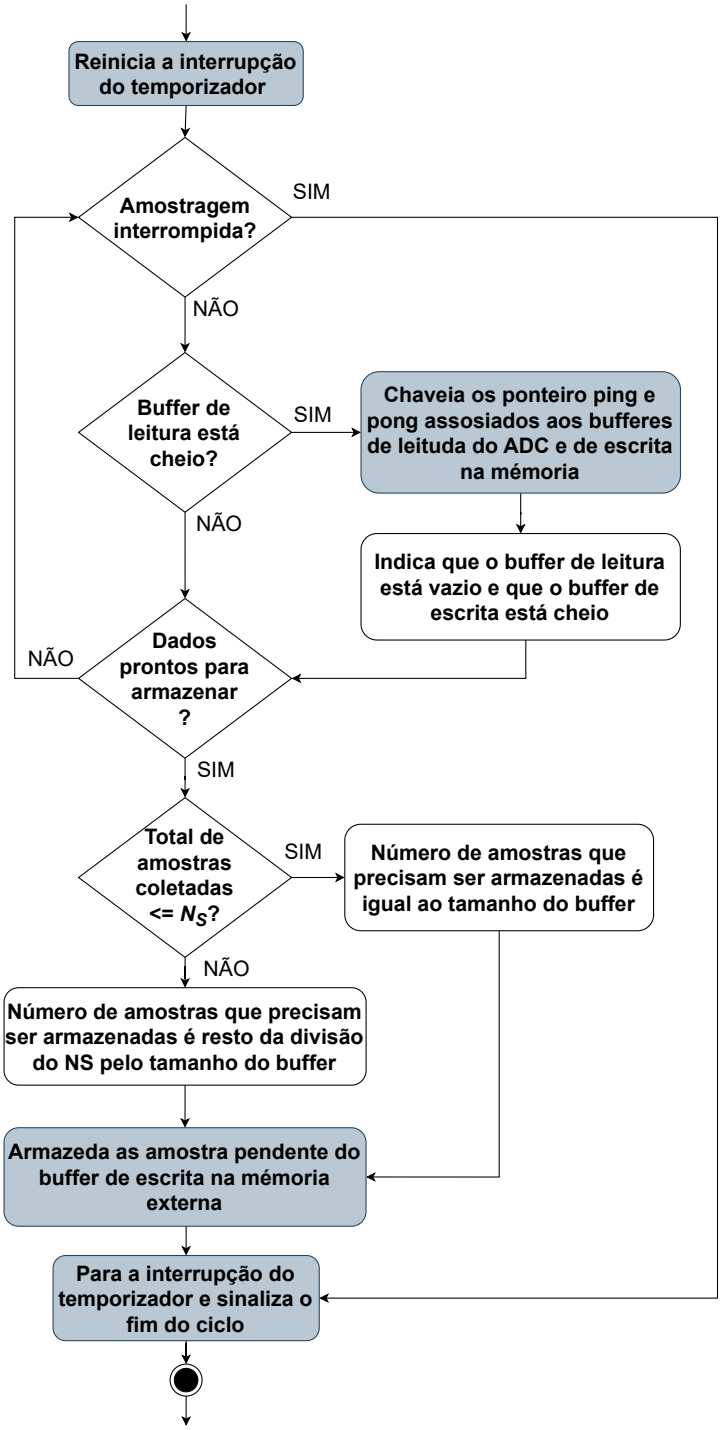
Após o processo de digitalização, abordado na Seção 3.2, o sinal amostrado fica armazenado na memória externa da UMSCF até que o módulo de comunicação consiga transmiti-lo para o servidor que concentra os dados desta rede de sensores. Neste trabalho, propomos uma nova etapa no sistema para a compressão desses dados, com o objetivo de otimizar o armazenamento e a transmissão. No cenário em questão, onde há limitações de processamento e memória, característicos dos sistemas embarcados, a eficiência do algoritmo deve estar balanceada com o custo da execução.

O algoritmo de compressão aplicado é baseado na análise de variação espectral do sinal de corrente de fuga, com a principal referência sendo o trabalho [16], discutido na Seção 2.4.2. A essência do mesmo é a detecção de alterações no domínio da frequência entre os segmentos do sinal, a fim de representá-lo de forma simplificada apenas com nesses casos. Para que essa análise seja executada, um pré-processamento é necessário, e na Seção 3.3.1 pode-se compreender melhor os passos dessa operação. Em seguida, na Seção 3.3.2 e no Apêndice A serão apresentados os detalhes essa implementação.

Foram realizadas adaptações para ajustar o algoritmo do trabalho de referência ao ambiente de desenvolvimento das UMSCF e às características dos sinais de corrente de fuga. No trabalho original, o algoritmo foi implementado em um FPGA com 13.860 elementos lógicos operando a 25 MHz, processando sinais de potência com distúrbios e tensão nominal de 230

[1] Hoang, Khoi. *STM32_TimerInterrupt*. Disponível em: arduino.cc/reference/en/libraries/stm32_timerinterrupt/

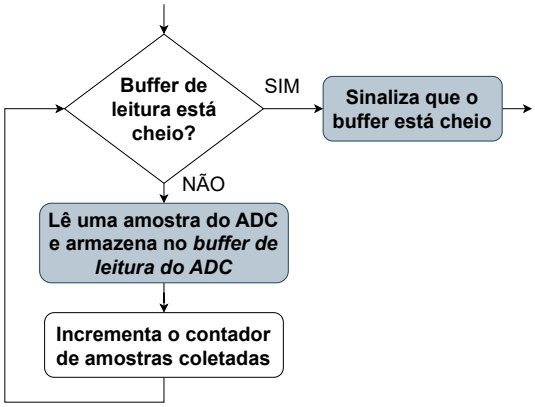
Figura 13 – Diagrama da rotina de amostragem executada no loop principal.



Fonte: A autora.

kV, utilizando altas taxas de amostragem, como $F_s = 15.360$ kHz para sinais reais e $F_s = 7.680$ kHz, 15.360 kHz, 30.720 kHz e 61.440 kHz para sinais sintéticos. Em contraste, neste estudo, o algoritmo foi implementado em um ESP32, processando sinais de corrente de fuga com magnitudes na escala de miliamperes, condicionados a uma faixa de 0 a 3.3V. As taxas de amostragem adotadas foram ajustadas às limitações do hardware, não excedendo 8 kHz.

Figura 14 – Diagrama da rotina de amostragem executada na interrupção do temporizador.



Fonte: A autora.

Adicionalmente, algumas etapas do algoritmo original, comentados na Seção 2.4.2, foram modificadas ou descartadas para melhor se adaptar às restrições do sistema embarcado utilizado. As etapas de estimativa de frequência e interpolação, correspondentes às etapas A e B da Fig. 5, foram eliminadas. Isso ocorreu devido às simplificações adotadas de que a frequência fundamental do sinal é sempre 60 Hz, e que apenas uma faixa de valores específicos de taxa de amostragem são utilizados para dispensar a necessidade da reamostragem. Dessa forma, garantiu-se que, na etapa seguinte (C), onde ocorre a segmentação em frames com um número específico de amostras por ciclo fundamental, o sinal segmentado resultasse em uma quantidade inteira de frames, cada um com um número inteiro de elementos.

A segmentação em frames, correspondente à etapa C, foi mantida. No entanto, foi necessário implementar um gerenciamento de memória que utilizasse armazenamento externo, para lidar com as limitações de memória do microcontrolador. A execução da FFT por ciclo, etapa D, foi otimizada utilizando a propriedade da simetria hermitiana dos sinais reais, reduzindo pela metade o número de componentes da FFT a serem processados e armazenados. A análise de variação espectral (compressão com perdas), seguiu a lógica original do trabalho de referência, enquanto a etapa de compressão sem perdas, foi descartada pois exigia uma fase de treinamento para gerar dicionários Huffman.

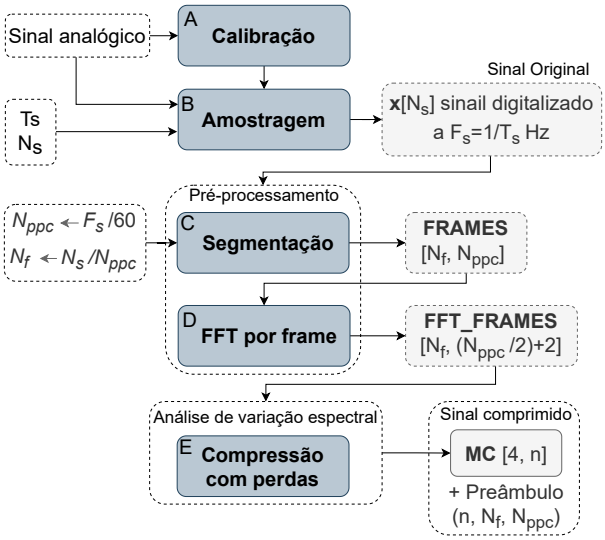
Para a validação, a compressão através do *Spectral Variation Detection Algorithm* (SVDA) foi implementada no MATLAB [2] e em uma UMSCF com uma versão simplificada do sistema, onde o arquivo com as amostras do sinal original encontra-se de forma estática na memória externa, e no final, a versões comprimida do mesmo é armazenada no cartão para fins de comparação. Nos testes foi utilizado o dataset dos sinais de corrente digitalizados pelas UMSCFs em um ambiente controlado, com um gerador de sinais simulando o sinal do sensor, e em campo, com as unidades coletando o sinal real dos sensores nos isoladores da UTEPSI.

[2] MATLAB, desenvolvido pela MathWorks. Disponível em: <https://www.mathworks.com/products/matlab.html>

3.3.1 Etapas do Pré-processamento

A Fig. 15 apresenta uma representação geral do fluxo de processamento do sinal pelo módulo de aquisição. Nas etapas A e B, ocorre a leitura do sinal de corrente de fuga em dois momentos do processamento: durante a calibração e a amostragem, respectivamente. O sinal digitalizado pela UMSCF a uma frequência de amostragem de $1/T_s$ Hz é representado como um vetor de números reais de tamanho N_s, onde $\mathbf{x}[q]$ indica a q-ésima amostra do vetor $\mathbf{x}$, que contém o sinal original, com q variando de 1 a N_s. Essa sequência de valores é armazenada em um arquivo binário, acompanhada de um preâmbulo que especifica os parâmetros de amostragem, como a frequência de amostragem (F_s) e o número total de amostras (N_s).

Figura 15 – Visão geral do processamento do sinal no módulo de aquisição da DAQ.



Fonte: A autora.

Nas etapas subsequentes, de C a E, representadas na Fig. 15, ocorre a aplicação do algoritmo de compressão. As duas primeiras etapas constituem o pré-processamento do sinal digitalizado, armazenado na memória externa. Após abrir o arquivo que contem o sinal original, realiza-se o primeiro passo do processo, que consiste em segmentá-lo, conforme ilustrado na Fig. 16a, em frames de tamanho igual a:

$$N_{ppc} = \frac{2\pi}{\Omega} = \frac{2\pi}{(2\pi f)/F_s} = \frac{1}{f/F_s} = \frac{F_s}{f}, \tag{3.3}$$

onde N_{ppc} é o número de amostras que compõem um ciclo fundamental.

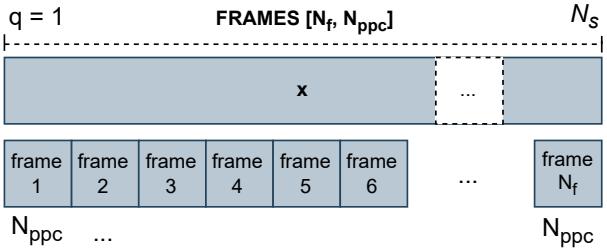
É preferível que este valor seja uma potência de 2 para maximizar a eficiência do cálculo da DFT pelo algoritmo de Cooley-Tukey. Conforme determinado na Equação 3.3, o valor do grau de liberdade é dado por F_s para definir o N_{ppc}, já que f é 60 Hz devido às características do sinal de interesse, discutido na Seção 2.3. Outro ponto importante na definição deste valor é que ele deve garantir que a segmentação do sinal original resulte em um número inteiro de frames

(N_f), como especificado em:

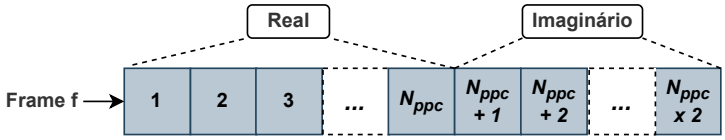
$$N_f = \frac{N_s}{N_{ppc}}. \tag{3.4}$$

Caso contrário, será necessário reamostrar o sinal original, algo realizado nos testes executados no MATLAB, mas não implementado nas UMSCF neste trabalho. Do arquivo original, são obtidos, além do sinal amostrado **x**, o F_s e o N_s, utilizados para calcular os valores de N_{ppc} e N_f. Conforme indicado na Fig. 15, após a segmentação do sinal, segue-se para a etapa mais importante do pré-processamento, a aplicação da DFT em cada um dos frames.

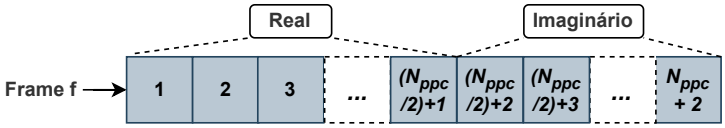
Figura 16 – Estrutura dos dados em diferentes etapas do pré-processamento.



(a) Segmentação do sinal original em frames.



(b) Resultado complexo do cálculo da DFT para um frame.



(c) Coeficientes da DFT de um frame que são realmente utilizados.

Fonte: A autora.

Esta operação gera os coeficientes da DFT de cada um dos frames. Como esses coeficientes são números complexos, um frame com N_{ppc} elementos, após essa operação, será um frame de coeficientes com $2 \times N_{ppc}$ elementos, devido às partes real e imaginária, conforme ilustrado na Fig. 16b. Esse aumento de dimensão é tratado utilizando a propriedade da simetria conjugada, abordada na Seção 2.2.2, que permite recuperar os coeficientes de frequência negativa apenas realizando a operação de conjugado dos coeficientes de frequência positiva. Ou seja, apenas parte dos coeficientes da saída da FFT são utilizados, devido à simetria Hermitiana.

Dessa forma, apenas os valores complexos do componente DC, dos elementos das frequência positivas e a de Nyquist (N_{ppc} é par) precisam ser considerados, resultando em frames de coeficientes de dimensão $N_{ppc}+2$ elementos, como indicado na Fig. 16c. Esta abordagem representa uma contribuição significativa do presente trabalho, uma vez que o trabalho de referencia [16] não aplicava essa propriedade.

Na implementação no MATLAB, a função *fft* [3] foi utilizada para realizar a operação, com os resultados armazenados em uma estrutura matricial. Já na implementação embarcada, foi utilizada a biblioteca *arduinoFFT* [4] e, devido às limitações de memória, os resultados dessa operação foram armazenados em um arquivo binário temporário no microSD da UMSCF.

3.3.2 Algoritmo de Detecção de Variação Espectral

Com o sinal segmentado e transformado para o domínio da frequência, a operação de compressão é iniciada conforme o algoritmo apresentado no diagrama da Fig. 17.

O processamento é realizado percorrendo cada elemento q de cada frame f, onde q varia do primeiro ao último elemento do frame ($N_{ppc}+2$) e f vai do primeiro ao último frame (N_f). Além disso, foi utilizado quatro vetores auxiliares de tamanho igual ao do frame, sendo eles: MIN, que contém o valor mínimo dos frames em cada posição; MAX, que contém o valor máximo dos frames em cada posição; SUM, que contém a soma acumulada; e CNT, que registra a contagem de repetições.

No primeiro ciclo do loop, o primeiro frame inicializa os vetores MIN, MAX e SUM, enquanto o CNT recebe o valor 1 em cada posição, indicando a primeira ocorrência daqueles valores. Seguindo o fluxo presente na Fig. 17, verifica-se se o elemento de índice q do frame f é o máximo ou o mínimo para aquela posição. Esses valores são utilizados para determinar se a variação é significativa o suficiente para ser considerada uma "novidade", dado o limiar:

$$\gamma(q) = Gq^{-\beta}\ , \tag{3.5}$$

com G (ganho) e β sendo parâmetros configuráveis que impactam a sensibilidade do algoritmo a variações.

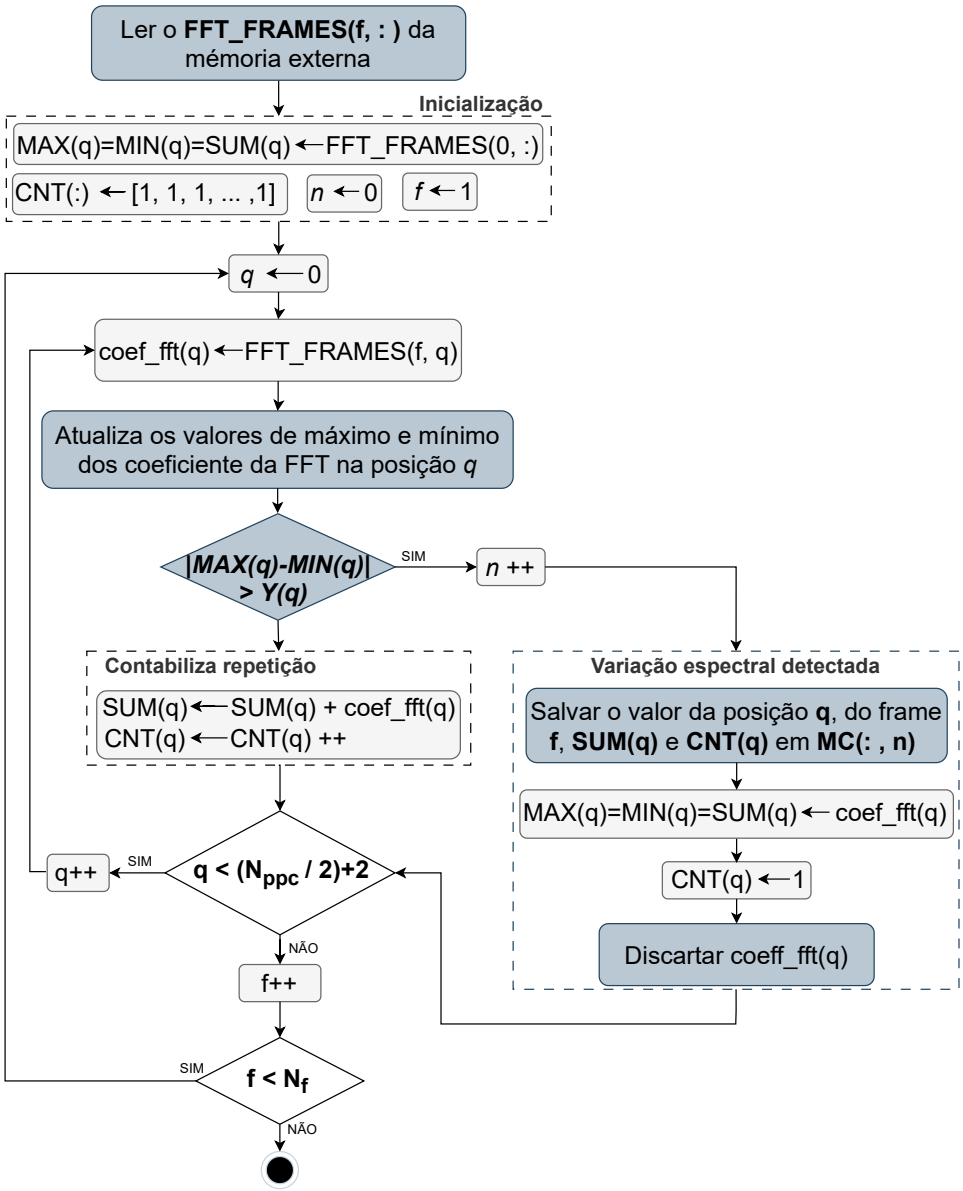
Caso o módulo da diferença entre $MAX(q)$ e $MIN(q)$ seja maior que $\gamma(q)$, uma novidade é detectada e todas as informações necessárias para recuperar essa informação posteriormente (a posição q onde ela foi detectada, o frame f da última ocorrência da repetição, a soma acumulada $SUM(q)$ e a quantidade de repetições $CNT(q)$) são salvas em uma estrutura denominada "matriz de compressão" (MC), descrita na Tabela. 3. Por ter dimensão inicialmente indeterminada (pois depende do número de novidades detectadas), essa estrutura é implementada no ESP32 como uma lista encadeada através da biblioteca *LinkedList*[5], onde a ordem dos elementos é indicada por um ponteiro, conforme discutido em [27, p. 258].

Se o módulo da diferença entre $MAX(q)$ e $MIN(q)$ for menor que $\gamma(q)$, a variação não é considerada relevante, então $CNT(q)$ contabiliza mais uma repetição e $SUM(q)$ é incrementado com o valor do frame na posição q. O ciclo se repete até que todos os elementos de cada frame

[3] Função *fft* do MATLAB. Disponível em: <https://www.mathworks.com/help/matlab/ref/fft.html>
[4] Condes, Enrique. *arduinoFFT*. Disponível em: <https://www.arduino.cc/reference/en/libraries/arduinofft/>
[5] Seidel, Ivan. *LinkedList*. Disponível em: <https://www.arduino.cc/reference/en/libraries/linkedlist/>

sejam analisados. Como o salvamento das informações só ocorre quando há uma "mudança" (detecção de novidade), os dados do último frame precisam ser tratados para não serem descartados indevidamente.

Figura 17 – Diagrama do algoritmo de compressão.



Fonte: A autora.

A estrutura que contém as informações de saída desta etapa constitui o dado comprimido. Para formar o arquivo binário final, é adicionado um preâmbulo contendo informações importantes, como o número de novidades detectadas (n), o número de pontos por ciclo (N_{ppc}) e o número de frames (N_f). Na Tabela 3, é apresentada a descrição completa dos elementos que compõem o arquivo comprimido, além da relação entre a quantidade de variações espectrais detectadas (n) e o tamanho final do arquivo. Vale destacar que a informação de n é repetida quatro vezes no preâmbulo. Isso ocorre porque, inicialmente, seria aplicada uma compressão sem perdas em cada um dos vetores q, f, SUM e CNT, o que alteraria o tamanho individual de cada um e exigiria o

Tabela 3 – Descrição das variáveis que compõem o arquivo comprimido.

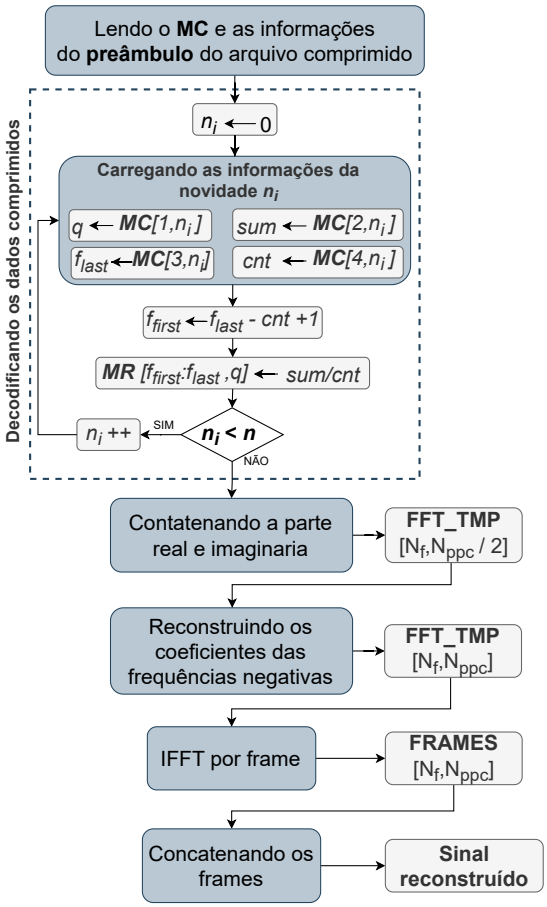
	Descrição	**Tamanho**	**Bytes**	**Classe**
MC $(n \times 4)$	q	$n \times 1$	$n \times 2$	uint16
	f	$n \times 1$	$n \times 4$	uint32
	SUM	$n \times 1$	$n \times 8$	double
	CNT	$n \times 1$	$n \times 4$	uint32
Preâmbulo	$n \times 4$, N_{ppc} e N_f	6x1	24	uint32
	Total (em bytes):	$(n \times 18) + 24$		

armazenamento dessa informação para cada vetor. Como essa etapa de compressão sem perdas não foi implementada, os tamanhos dos vetores permanecem fixos e iguais a n, gerando essa redundância no preâmbulo.

3.4 Reconstrução do Sinal Comprimido

As etapas do algoritmo de reconstrução do sinal são basicamente o caminho inverso das etapas da compressão, conforme apresentado no diagrama da Fig. 18 e detalhado na implementação presente no Apêndice B.

Figura 18 – Diagrama do algoritmo de reconstrução.



Fonte: A autora.

Primeiramente, os dados armazenados na matriz de compressão são decodificados para gerar a versão reconstruída de parte dos coeficientes da DFT de cada frame. Os mesmos são obtidos a partir das informações de cada uma das novidades, sendo preenchidos com o valor médio. Por exemplo, dada a novidade de número 7 com $q = 3$, $f = 18$, $SUM(q) = 125$ e $CNT(q) = 8$, obtém-se como resultado que, na posição 3 dos frames de 10 a 18, o valor médio dos coeficientes será $125/8$.

Em seguida, as partes real e imaginária são concatenadas para formar os coeficientes na forma cartesiana complexa. A reconstrução dos coeficientes das frequências negativas é realizada a partir do conjugado complexo dos coeficientes das frequências positivas. Com as informações completas da DFT, aplica-se a transformada inversa, e os *frames* são concatenados para compor o sinal reconstruído.

4 EXPERIMENTOS E RESULTADOS

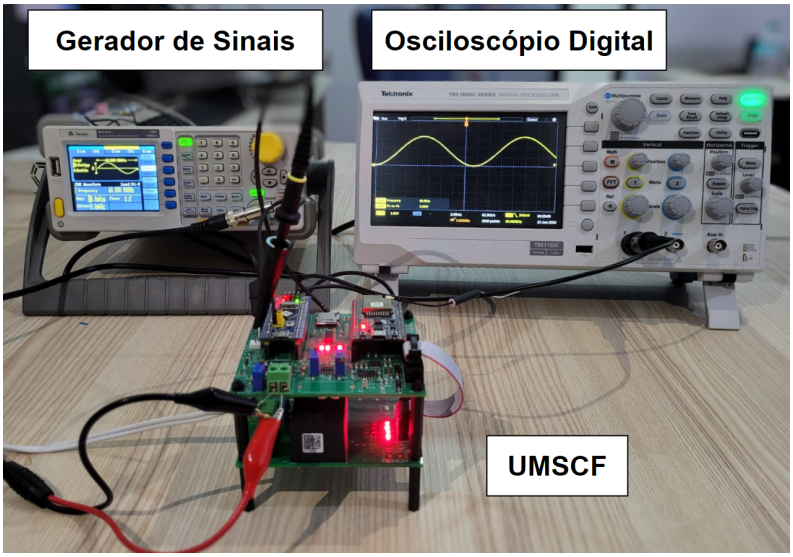
Neste capítulo, são descritos os experimentos realizados e os resultados obtidos com a aplicação do sistema proposto. Detalham-se os procedimentos de captura e armazenamento dos dados, seguidos pela apresentação dos resultados da compressão e reconstrução do sinal.

4.1 Aquisição do Sinal Original

Os sinais digitalizados pelas UMSCFs foram adquiridos de duas maneiras: utilizando um gerador de sinais para simular o sinal do sensor como uma senoide de 10 mVpp e 60 Hz, e diretamente dos sensores de corrente de fuga instalados na UTEPSI.

Para avaliar a digitalização e compressão dos sinais em um ambiente controlado, foi utilizado o gerador de sinais MFG-4225, o osciloscópio TBS1102C e a UMSCF, conforme mostrado na Fig. 19.

Figura 19 – Testbed usado na aquisição de sinais em um ambiente controlado.



Fonte: A autora.

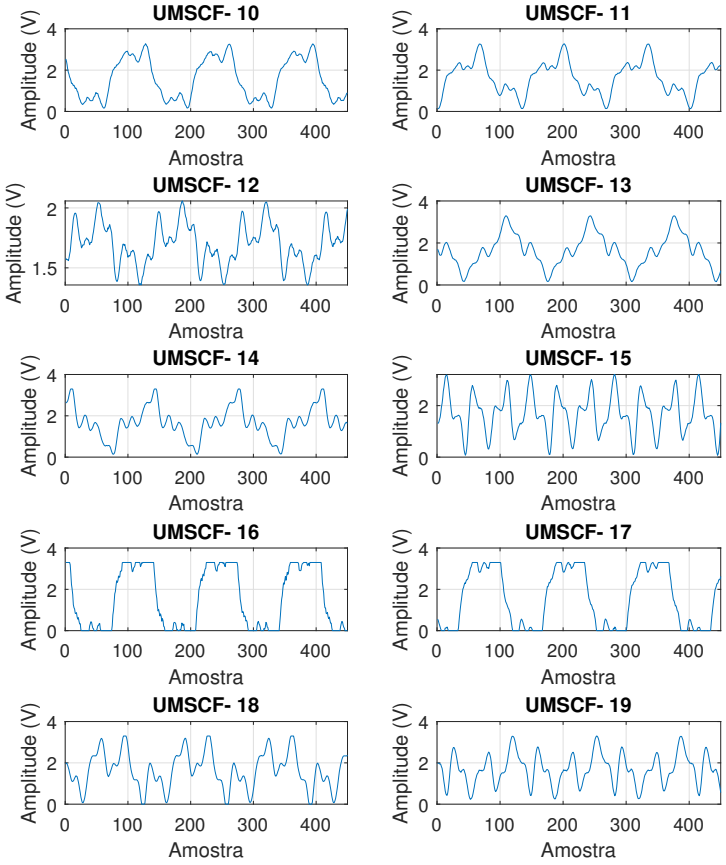
Já os sinais de corrente de fuga reais, foram coletados das UMSCFs dispostas nos pontos de monitoramento indicados na Fig. 20. Para este estudo, um arquivo contendo o sinal digitalizado por cada uma das dez unidades foram utilizados nos testes do algoritmo de compressão, nos gráficos da Fig. 21 são mostrados as primeiras 450 amostras de cada um deles. Os parâmetros de amostragem dessas unidades foram definidos como F_s de 8000Hz (reamostrada posteriormente para 7680 Hz) e um N_s de 8000 amostras, gerando arquivos de 156kB.

Figura 20 – Visão geral da UTEPSI e o mapa da disposição das UMSCF instaladas.

(a) Usina Termoelétrica Porto de Sergipe I. **(b) Disposição das unidades.**

Fonte: A autora.

Figura 21 – Exemplos de sinais de corrente de fuga capturados pelas UMSCFs (tensão condicionada na entrada do ADC) no dia 27/06/2024 na UTEPSI.



Fonte: A autora.

4.2 Taxas de Compressão e Qualidade do Sinal Reconstruído

Os valores de RTE em todos os experimentos ficaram em torno de 100% e o CRE em torno de 1, sugerindo uma alta preservação da energia e da estrutura do sinal original. Devido à baixa variação observada nessas métricas, a análise nas seções subsequentes será focada no NMSE e na taxa de compressão.

Para comparar as taxas de compressão (CR) obtidas com o SVDA, foram geradas versões comprimidas dos arquivos originais nos formatos .zip e .rar utilizando o WinRAR. É importante destacar que há diferenças significativas nos algoritmos de compressão utilizados por esses formatos.

O formato .zip utiliza principalmente o algoritmo DEFLATE, que combina os métodos LZ77 (Lempel-Ziv 1977) e Huffman. O método LZ77 trabalha com a substituição de cadeias de caracteres repetidas por referências a ocorrências anteriores dentro do mesmo arquivo, enquanto a codificação de Huffman é um algoritmo de compressão sem perdas que usa uma técnica de codificação variável baseada na frequência de ocorrência dos símbolos.

Por outro lado, o formato .rar usa uma combinação de algoritmos de compressão, incluindo LZSS (Lempel-Ziv-Storer-Szymanski) e outros algoritmos proprietários desenvolvidos pela RARLAB. O método LZSS é uma variante do LZ77 que usa uma técnica similar de substituição de cadeias repetidas, mas com algumas melhorias que permitem uma compressão mais eficiente. Além disso, o .rar pode utilizar diferentes métodos de compressão em conjunto, como a codificação de aritmética, filtros delta e técnicas de compressão de áudio e imagem específicas. Esses aprimoramentos permitem que o formato .rar atinja uma taxa de compressão maior em comparação com o formato .zip, especialmente em arquivos que contêm dados mais complexos ou menos redundantes.

Na Tabela 4, são apresentadas as taxas de compressão obtidas com o algoritmo de proposto (usando o *threshold* padrão), o RAR e o ZIP.

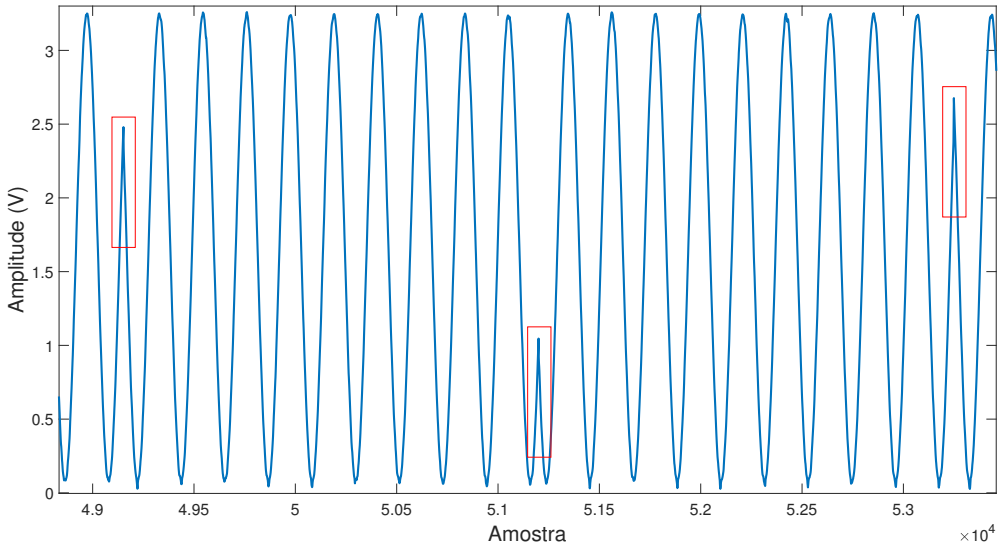
Tabela 4 – Comparação das CR atingidas pelo algoritmo proposto, pelo RAR e pelo ZIP, nos sinais digitalizados pela UMSCF em um ambiente controlado.

Arquivo original		**Segmentação**		**ZIP**	**RAR**	**SVDA**	
#	**Fs (Hz)**	**Nppc**	**Nf**			**CR**	**NMSE (dB)**
1	4800	80	800	1,25	1,96	10,89	-44,20
2	6000	100	640	1,26	1,93	6,12	-44,56
3	7680	128	500	1,27	1,98	9,41	-45,29
4	9600	160	400	1,28	2,06	7,51	-45,69
5	**12000**	200	320	1,29	2,17	**2,69**	-44,81
6	**15360**	256	250	1,31	2,26	**0,26**	-38,79
7	**19200**	320	200	1,31	2,27	**0,15**	-44,09
8	**30720**	512	125	1,30	2,28	**0,16**	-44,44

Observa-se que a maioria das taxas de compressão alcançadas pelo SVDA são superiores

em comparação com as versões compactadas com o ZIP e o RAR. No entanto, há uma grande influência dos parâmetros de amostragem usados na digitalização do sinal original sobre o desempenho do algoritmo. Nos arquivos 4 a 8, observou-se perdas de amostras, conforme mostrado na Fig. 22. Tal perda cria grandes variações no domínio da frequência e consequentemente um CR baixo. Isso ocorre devido a uma limitação do hardware em processar corretamente sinais com frequências de amostragem tão altas. Mais especificamente, o buffer ping-pong no STM32 não consegue escrever na memória externa em taxa maior do que a taxa de leitura de amostras, quando a frequência de amostragem é maior que $8k$Hz.

Figura 22 – Sinal original (arquivo 8) amostrado com uma $F_s = 30720$Hz.



Fonte: A autora.

Nos resultados obtidos com os testes utilizando os sinais vindos da subestação, fica claro que os parâmetros de amostragem não são os únicos fatores que influenciam o desempenho. Todos os sinais listados na Tabela 5 foram digitalizados com a mesma Fs (8kHz e posteriormente reamostrada para 7680Hz), mas o algoritmo atingiu taxas de compressão bastante distintas. Tal comportamento é oposto ao observado nas versões compactadas com o ZIP e o RAR, que apresentam taxas bem constantes.

Nos gráficos da Fig. 23, há em azul o gráfico do sinal original e em vermelho o sinal reconstruído a partir da versão comprimida dos sinais vindos da UMSCF-12 e da UMSCF-16. Dado que foi utilizado o *threshold* padrão, a primeira atingiu a maior taxa de compressão e a segunda a menor, ficando claro pelo resultado obtido com o signal saturado da UMSCF-16, que as distorções refletem na eficiência do algoritmo.

Tabela 5 – Comparação das CR atingidas pelo algoritmo proposto, pelo RAR e pelo ZIP, nos sinais digitalizados pelas UMSCFs em campo.

UMSCF	Threshold padrão (G=1 e β = 0.001)			ZIP	RAR
	n	CR	NMSE(dB)		
10	1600	5,55	-38,21	1,25	1,82
11	1091	8,14	-39,28	1,26	1,86
12	228	**38,76**	-44,56	1,50	2,24
13	393	22,54	-40,76	1,29	1,83
14	981	9,05	-39,67	2,21	2,51
15	1703	5,22	-39,21	1,24	1,59
16	6160	**1,44**	-37,42	2,81	2,81
17	3365	2,64	-37,80	2,74	2,87
18	923	9,62	-38,66	2,07	2,32
19	1447	6,14	-38,93	1,24	1,65

Em [16], o algoritmo SVDCS testado em sinais de de curta duração de um subestação, obteve CR de 43, enquanto a melhor taxa registrada pelo SVDA implementado nas UMSCFs com o threshold padrão foi de 38,76. O autor disponibilizou em [28], o dataset de um sinal de corrente adquirido de um gerador eólico, com uma taxa de amostragem de 7680 kHz e duração de 3640 segundos. O arquivo fornecido é um *.mat* contendo 27.955.200 amostras do sinal no formato *double* (8 bytes por amostra), sendo que não foi informado o CR obtido para o sinal dessa duração especifica, nem o tamanho do arquivo que o contém. O tamanho total do arquivo é estimado em 223.641.600 bytes, o que foi utilizado para o cálculo da CR. Ao testar o algoritmo SVDA nesse dataset, utilizando diferentes combinações arbitrárias de *threshold*, obtivemos uma CR de até 644,25 com um NMSE de -33,8 dB (com G = 125,95 e $\beta = 0,001$).

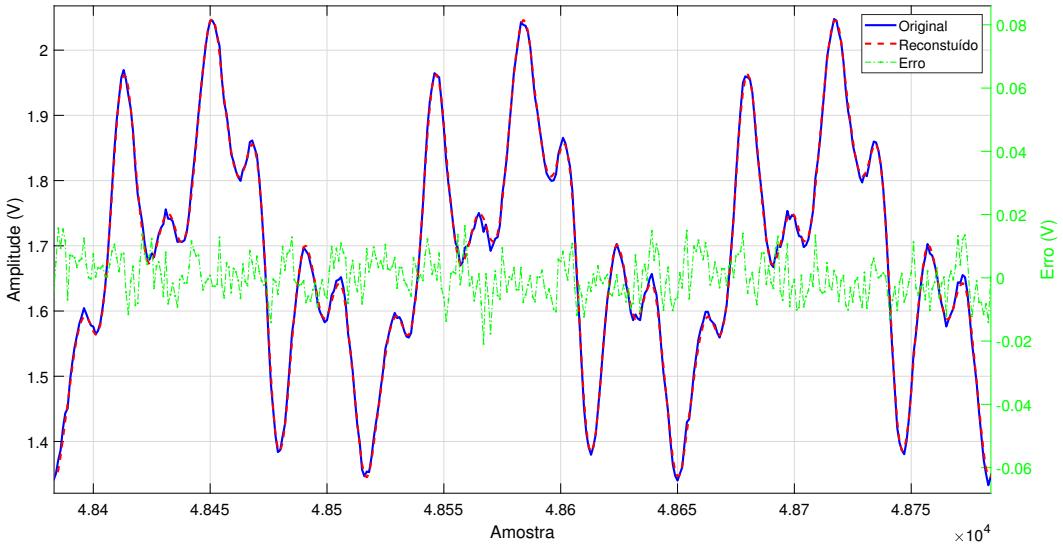
4.2.1 Impacto do Threshold

A definição dos parâmetros do *threshold* é crucial para a eficiência do algoritmo, especialmente quando se lida com os sinais reais que possuem aspectos distintos, embora com a mesma frequência fundamental.

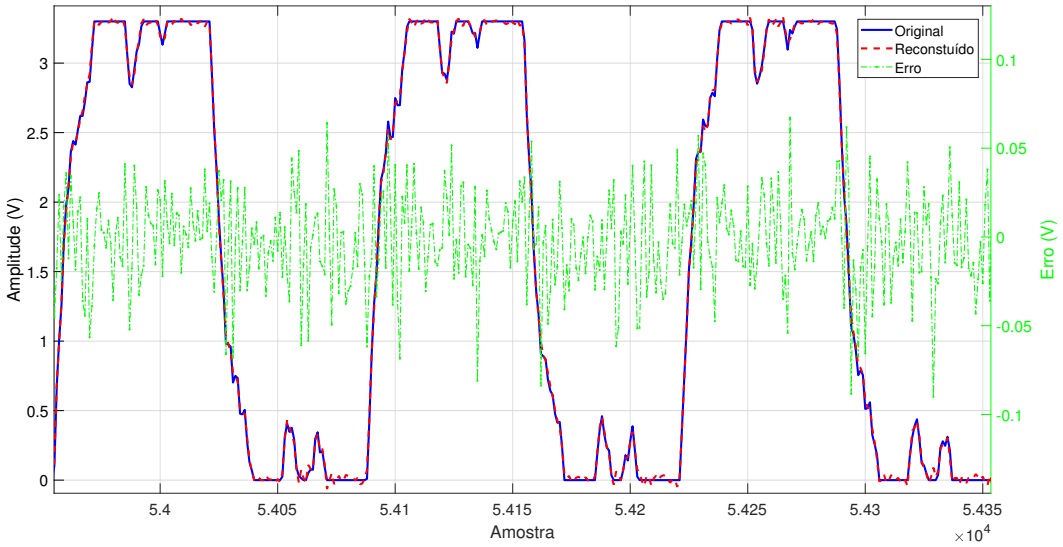
Conforme discutido na Seção 3.3.2, o número de variações espectrais detectadas (n) pelo SVDA está diretamente relacionado à sensibilidade do algoritmo, configurada pelo *threshold*. A quantidade de "novidades" detectadas é inversamente proporcional à taxa de compressão e ao NMSE, como expresso pelo gráfico da Fig. 24.

Para equilibrar a compressão e a distorção do sinal é fundamental compreender a relação direta entre essas métricas. Como mostrado nos gráficos da Fig. 25 e Fig. 26, aumentar o CR eleva o NMSE, resultando em maior distorção no sinal reconstruído. Observando a variação dos dois parâmetros do *threshold* (β e G) em uma mesma faixa de valores, verifica-se que G exerce uma influência mais rápida e significativa sobre a compressão e a distorção no intervalo analisado, conforme evidenciado nos testes realizados com o arquivo 3 dos sinais simulados

Figura 23 – Comparação do sinal de corrente de fuga real com sua versão reconstruída.



(a) Sinal digitalizado pela a UMSCF-12 e comprimido à uma taxa $38,76$**.**



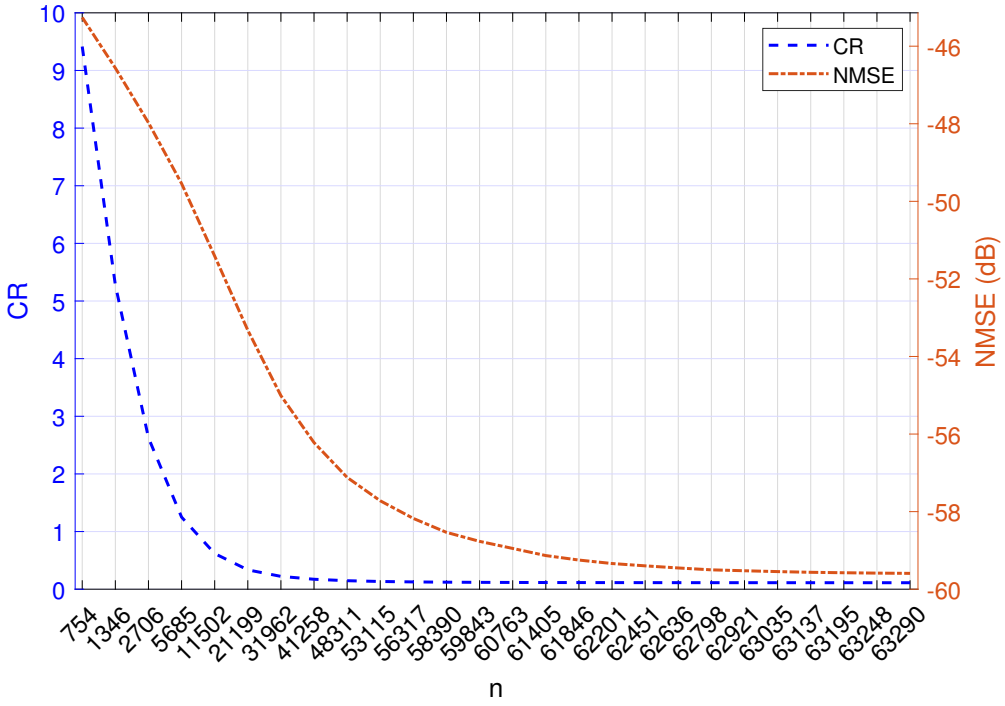
(b) Sinal digitalizado pela a UMSCF-16 e comprimido à uma taxa $1,44$**.**

Fonte: A autora.

($F_s = 7680\,\text{Hz}$, $N_s = 64000$).

Como definido na Equação 3.5, o *threshold* possui duas variáveis que podem ser ajustadas, com os valores padrão usados sendo $\beta = 0.001$ e $G = 1$. Para entender a influência desses parâmetros, testes foram realizados variando os valores desse dois parâmetros de forma combinada na faixa de 0.0001 e 3.3, utilizando o arquivo 3. Os resultados mostram que valores mais altos de G e mais baixos de β proporcionam combinações mais interessantes para o sinal testado, indicado nas Figuras 27 e 28.

Figura 24 – Impacto da quantidade de variações detectadas na CR e no NMSE, ao variar os valores de β na faixa de 0.001 a 3.3, mantendo $G = 1$.



Fonte: A autora.

Para os testes práticos, optou-se por manter o valor padrão de β e ajustar o valor de G com base nos valores de pico do sinal original usando as relações:

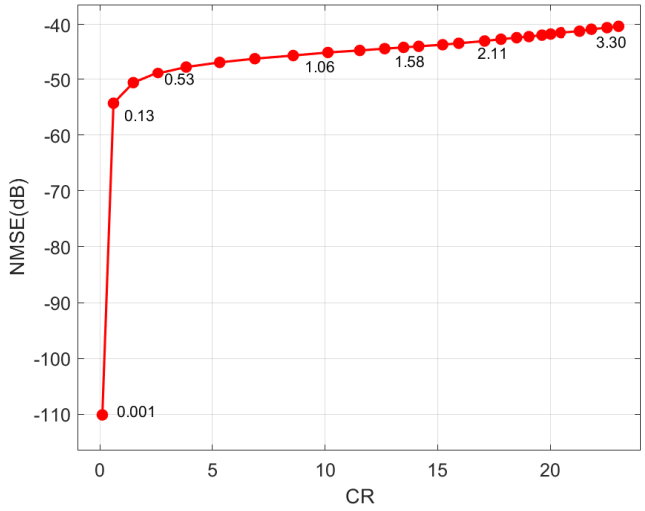
$$G = \frac{|\max(x) - \min(x)|}{10} + \frac{|\max(x) - \min(x)|}{2} \tag{4.1}$$

e

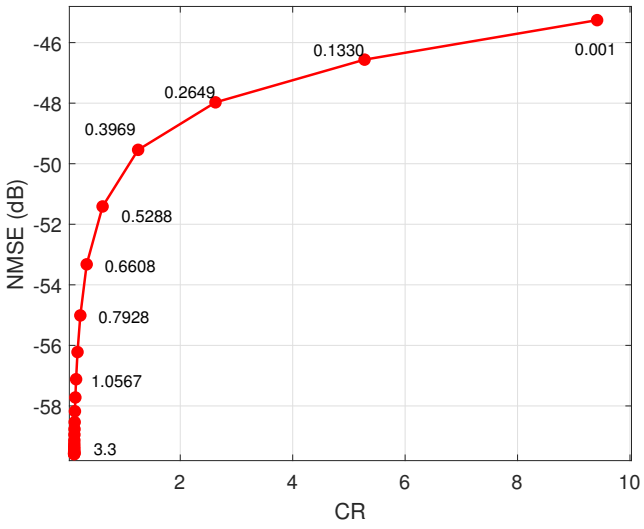
$$G = \frac{|\max(x) - \min(x)|}{5} \quad . \tag{4.2}$$

Os resultados listados na Tabela 6, onde na versão que prioriza a menor distorção, o tamanho do arquivo comprimido do sinal digitalizado pela UMSCF-16 ficou maior que o original, resultando em uma taxa de compressão menor que 1. Já na versão que busca manter um equilíbrio entre compressão e distorção, obteve-se um CR de 4. Nesse teste, o sinal que alcançou a maior compressão foi o digitalizado pela UMSCF-13 e o gráfico correspondente pode ser visualizado na Fig. 29.

Figura 25 – Relação entre a CR e o NMSE ao variar um dos parâmetros do *threshold* ao comprimir o sinal do arquivo 3.



(a) Variando G na faixa de 0.001 à 3.3, mantendo $\beta = 0.001$.



(b) Variando β na faixa de 0.001 à 3.3, mantendo $G = 1$.

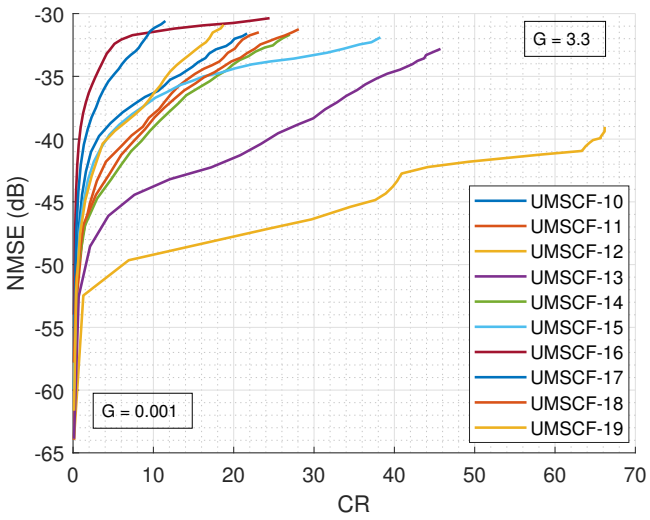
Fonte: A autora.

4.3 Interface Gráfica do Usuário

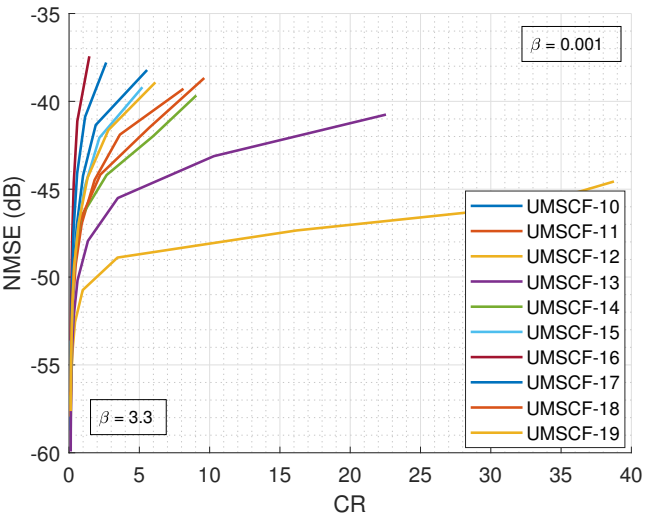
Para facilitar a execução do algoritmo foi desenvolvida uma *Graphical User Interface* (GUI) utilizando a biblioteca Tkinter em Python. Esta interface, ilustrada na Fig. 30, tem como objetivo proporcionar uma experiência simplificada e intuitiva, similar à do WinRAR, um software amplamente conhecido para compressão e descompressão de arquivos.

Os algoritmos de compressão e reconstrução foram reimplementados em *python* e a aplicação convertida para um executável (.exe) utilizando o *PyInstaller*, permitindo uma distribuição mais fácil e prática. Com essa interface, os usuários podem definir parâmetros como

Figura 26 – Relação entre a CR e o NMSE ao variar um dos parâmetros do *threshold* ao comprimir nos sinais coletados em campo.



(a) Variando G na faixa de 0.001 a 3.3, mantendo $\beta = 0.001$.



(b) Variando β na faixa de 0.001 à 3.3, mantendo $G = 1$.

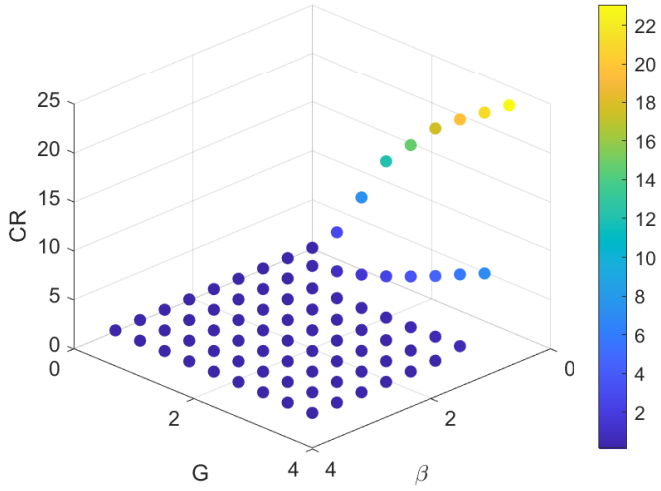
Fonte: A autora.

o uso de propriedades de simetria na compressão, a conversão do sinal para volts, a habilitação da reamostragem e a configuração do *threshold*, sem a necessidade de interagir diretamente com o código.

Na aba "Logs", são apresentadas informações como o caminho do arquivo, o preâmbulo e as métricas, com a Fig. 31 mostrando a saída após a execução da opção de "comprimir".

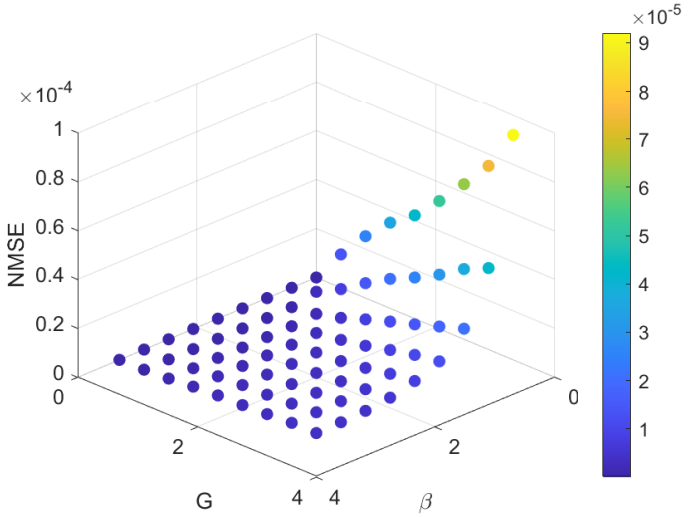
Além das funcionalidades de compressão e descompressão de arquivos binários específicos do projeto GImpSI, a interface também oferece as opções "Abrir"e "Avaliar". A opção "Abrir"lê as informações do preâmbulo dos arquivos e plota o sinal, sendo essa uma forma do usuário determinar se é necessário reamostrar se quiser comprimir o arquivo posteriormente,

Figura 27 – Impacto do *threshold* na CR alcançada pelo SVDA.



Fonte: A autora.

Figura 28 – Impacto do *threshold* na distorção (NMSE) atingida pelo SVDA.



Fonte: A autora.

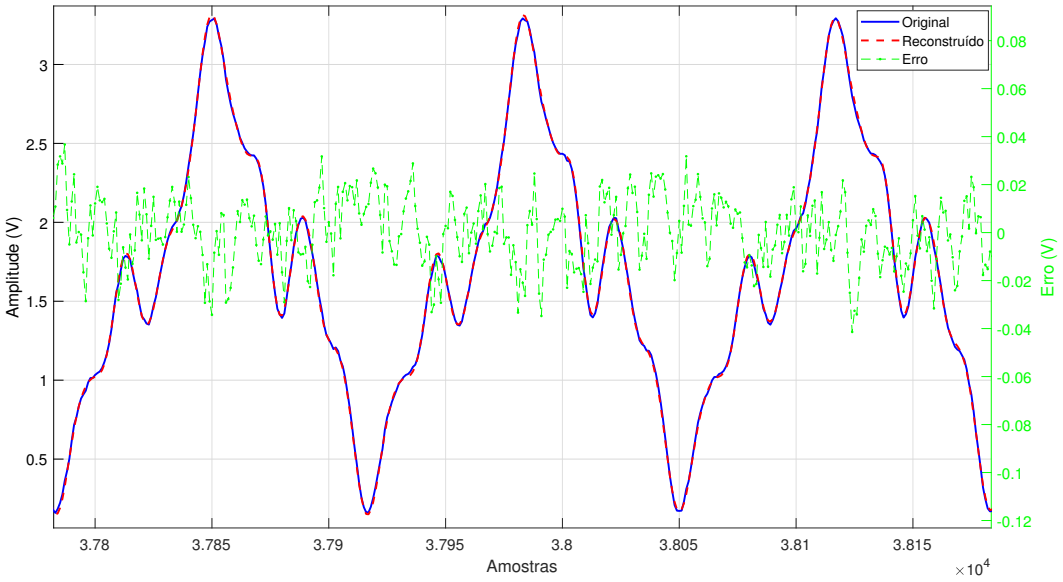
enquanto a opção "Avaliar"permite a análise da qualidade da reconstrução, comparando o arquivo original e sua versão reconstruída se as duas estiverem disponíveis, conforme exibido na Fig. 32.

A interface permite ao usuário navegar pelo explorador de arquivos para selecionar arquivos ou diretórios para abrir ou salvar.

Tabela 6 – Comparação da CR atingidas pelo algoritmo proposto com $\beta = 0.001$ e G variados, dos sinais digitalizados pelas UMSCFs em campo.

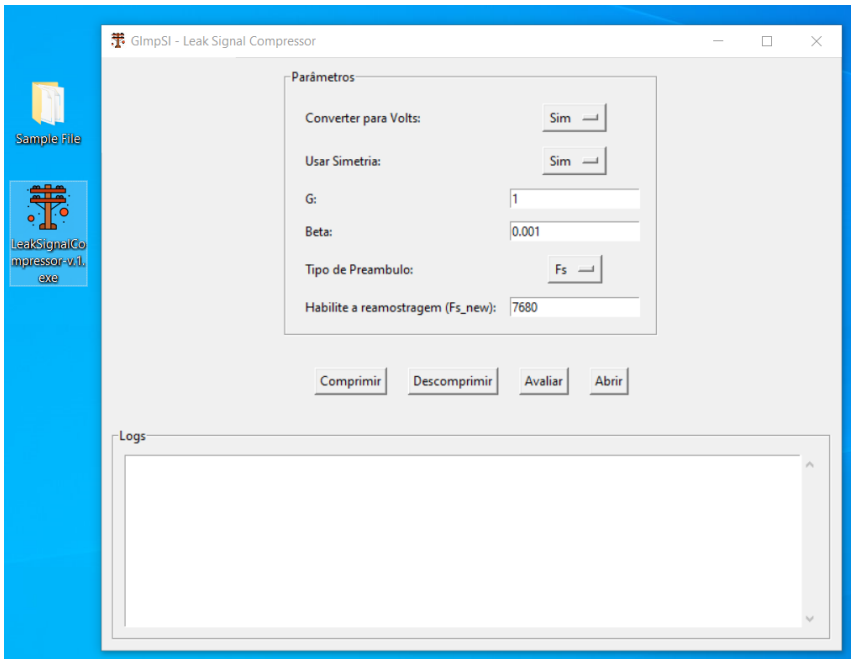
UMSCF	Menor distorções			Equilíbrio		
	G	CR	NMSE(dB)	G	CR	NMSE(dB)
10	0,635824	2,10	-41,31	1,907473	13,71	-34,76
11	0,647429	3,95	-41,94	1,942286	14,96	-34,93
12	0,149245	1,66	-52,22	0,447736	20,33	-47,96
13	0,631795	**11,05**	-43,47	1,895385	**34,95**	-36,38
14	0,636952	5,77	-42,29	1,910857	17,70	-35,13
15	0,654842	2,56	-41,87	1,964527	16,12	-35,02
16	0,660000	**0,70**	-40,46	1,980000	**4,20**	-33,18
17	0,660000	1,34	-40,46	1,980000	6,62	-33,62
18	0,660000	5,98	-41,25	1,980000	18,58	-34,28
19	0,621480	2,84	-42,01	1,864440	11,48	-35,14

Figura 29 – Sinal original digitalizado pela UMSCF-13 comparado com o sinal reconstruído a partir da vesão comprimida ($G = 1,895$ e $\beta = 0,001$).



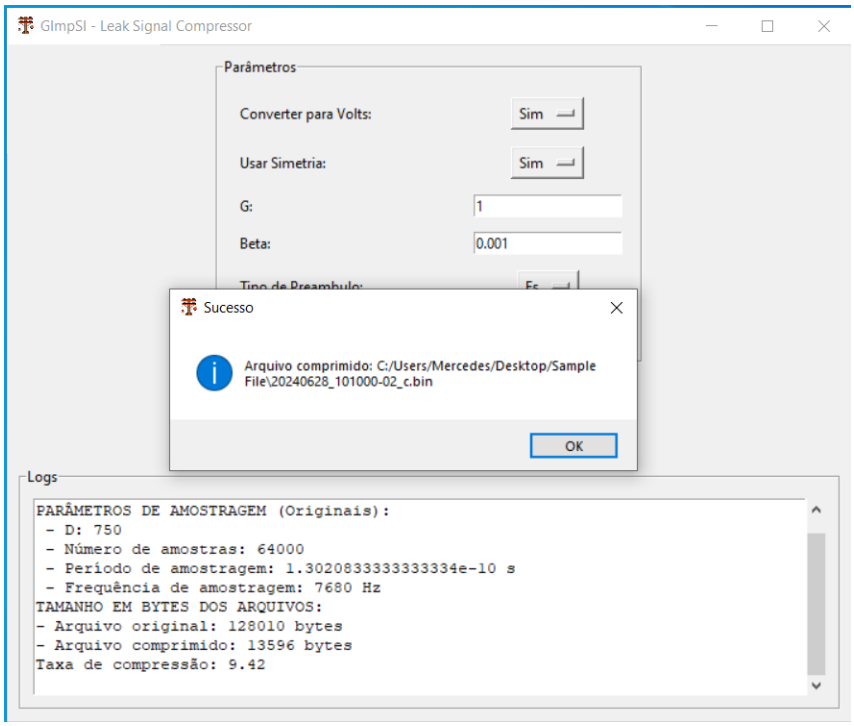
Fonte: A autora.

Figura 30 – Tela inicial da interface gráfica do usuário desenvolvida em Tkinter.



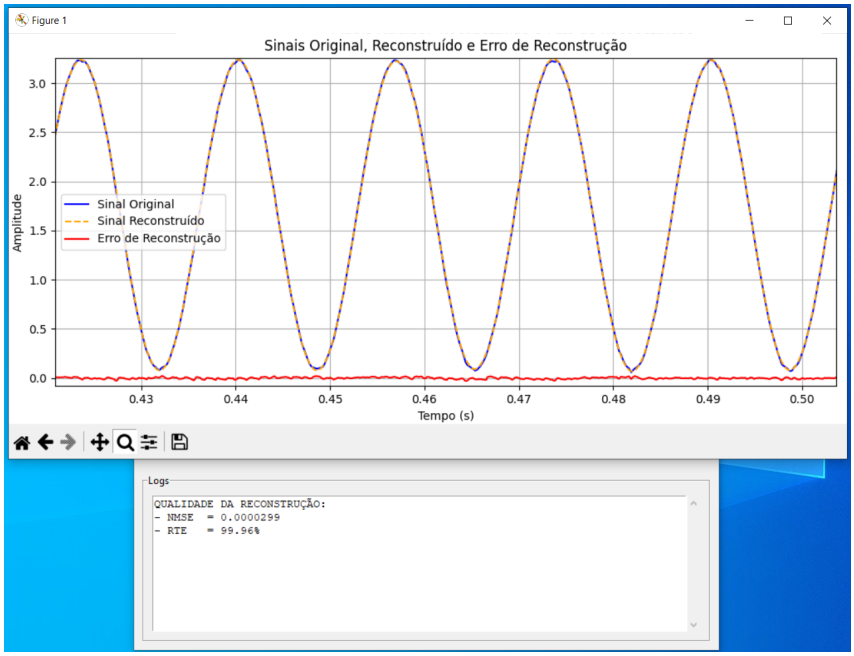
Fonte: A autora.

Figura 31 – Tela "Comprimir"exibindo nos logs o preâmbulo do arquivo original e a taxa de compressão.



Fonte: A autora.

Figura 32 – Tela de "Avaliar"exibindo nos logs as métricas de NMSE e RTE e o plot da comparação dos sinais.



Fonte: A autora.

5 CONCLUSÃO

Este trabalho apresentou uma proposta para a digitalização e compressão de sinais de corrente de fuga implementado em dispositivos IoT em funcionamento em uma subestação de alta tensão. Sendo essas, as Unidades Microprocessadas para Sensores de Corrente de Fuga (UMSCF), desenvolvidas no âmbito do projeto GImpSI. A implementação de técnicas de compressão de dados em sistemas embarcados com recursos limitados, mostrou-se eficaz para otimizar o armazenamento e a transmissão de grandes volumes de dados, preservando a integridade dos sinais monitorados.

Nos capítulos iniciais, foram abordados os fundamentos teóricos essenciais para o projeto, incluindo a Transformada Discreta de Fourier (Transformada Discreta de Fourier (TDF)) e métodos de compressão baseados em transformadas. A metodologia proposta foi detalhada, abrangendo etapas como o condicionamento, digitalização, compressão e reconstrução dos sinais de corrente de fuga. A implementação do algoritmo foi realizada em diversas linguagem de programação: C++ para a implementação do algoritmo de compressão no ESP32, Matlab para implementação do algoritmo de reconstrução, e Python para a interface gráfica do usuário.

Os resultados experimentais evidenciaram que a configuração adequada dos parâmetros de *threshold* é fundamental para equilibrar a taxa de compressão e a qualidade do sinal reconstruído. A análise das métricas de desempenho, incluindo NMSE, RTE, e CRE, comprovou que a abordagem proposta pode preservar a qualidade dos sinais enquanto reduz significativamente o volume de dados transmitidos e armazenados.

Para futuros trabalhos, sugere-se explorar a otimização dos parâmetros de *threshold* por meio de técnicas de aprendizado por reforço, que podem ajustar dinamicamente o algoritmo às condições variáveis dos sinais de corrente de fuga.

O sistema proposto permite uma gestão de dados mais eficiente, facilitando a manutenção preditiva dos isoladores de subestações e reduzindo o risco de falhas. A continuidade deste trabalho, juntamente com os avanços em tecnologias de IoT, poderá promover melhorias significativas no monitoramento e na gestão de redes elétricas inteligentes, garantindo maior confiabilidade e segurança.

REFERÊNCIAS

1 KHAN, F. et al. Iot based power monitoring system for smart grid applications. In: **2020 International Conference on Engineering and Emerging Technologies (ICEET)**. [S.l.: s.n.], 2020. p. 1–5.

2 ODONGO, G. Y. et al. An efficient lora-enabled smart fault detection and monitoring platform for the power distribution system using self-powered iot devices. **IEEE Access**, v. 10, p. 73403–73420, 2022.

3 KARTHICK, A. et al. Low-cost energy monitoring of gird connected solar photovoltaic systems. In: **2024 IEEE Third International Conference on Power Electronics, Intelligent Control and Energy Systems (ICPEICES)**. [S.l.: s.n.], 2024. p. 891–894.

4 VADREVU, S.; MANIKANDAN, M. S. A new quality-aware quality-control data compression framework for power reduction in iot and smartphone ppg monitoring devices. **IEEE Sensors Letters**, v. 3, n. 7, p. 1–4, 2019.

5 HANUMANTHAIAH, A. et al. Comparison of lossless data compression techniques in low-cost low-power (lclp) iot systems. In: **2019 9th International Symposium on Embedded Computing and System Design (ISED)**. [S.l.: s.n.], 2019. p. 1–5.

6 HAMDAN, S.; AWAIAN, A.; ALMAJALI, S. Compression techniques used in iot: A comparitive study. In: **2019 2nd International Conference on new Trends in Computing Sciences (ICTCS)**. [S.l.: s.n.], 2019. p. 1–5.

7 BANIK, A. et al. A comparative study on leakage current harmonics of porcelain disc insulator contaminated with nacl and kcl. In: **2015 1st Conference on Power, Dielectric and Energy Management at NERIST (ICPDEN)**. [S.l.: s.n.], 2015. p. 1–4.

8 ABEYSEKARA, A. H. A. D. et al. Remote leakage current detector for identification of insulators discharges. **IEEE Transactions on Dielectrics and Electrical Insulation**, v. 24, n. 4, p. 2449–2458, 2017.

9 RENDEIRO, P. H. F. **Development of an Iot System for Leakage Current Monitoring in High Voltage Insulators**. Trabalho de Conclusão de Curso (Bacharelado em Engenharia da Computação) — Universidade Federal do Pará, Belém, PA, Brasil, 2023.

10 CARVALHO, S. S. **Desenvolvimento de um Sistema de Medição de Corrente de Fuga para Isoladores de Alta Tensão**. Dissertação (Electronic Engineering) — Universidade Federal de Sergipe, São Cristóvão, 2022.

11 OLIVEIRA, G. V. S. **Sistema de Medição de Picos de Corrente de Fuga para Isoladores de Alta Tensão**. Dissertação (Master's thesis in Electrical Engineering) — Universidade Federal de Sergipe, São Cristóvão, 2023.

12 OPPENHEIM, A. V.; WILLSKY, A. S. **Sinais e Sistemas**. 2. ed. São Paulo, Brasil: Pearson Universidades, 2010. ISBN 978-8576055044.

13 KLAUTAU, A. **Digital Signal Processing with Python, Matlab or Octave: Applications in Machine Learning**. <https://ai6g.org/books/dsp/ak_dsp_book.html>. Accessed: 2024-05-18.

14 COOLEY, J. W.; TUKEY, J. W. An algorithm for the machine calculation of complex fourier series. **Mathematics of Computation**, American Mathematical Society, v. 19, n. 90, p. 297–301, 1965.

15 BARBOSA, V. R. N. et al. Estimation of the pollution critical level on the surface of glass insulators based on leakage current. **IEEE Transactions on Power Delivery**, v. 39, n. 2, p. 1222–1232, 2024.

16 KAPISCH, E. B. et al. Spectral variation-based signal compression technique for gapless power quality waveform recording in smart grids. **IEEE Transactions on Industrial Informatics**, v. 18, n. 7, p. 4488–4498, 2022.

17 ALVES, J. de S. **Compressão de sinais para smart grid**. Trabalho de Conclusão de Curso (Bacharelado em Engenharia Eletrônica) — Universidade de Brasília, Brasília, Brasil, 2019. Xiii, 43 f.

18 JOSE, K. M.; MORSI, W. G. Smart grid data compression of power quality events using wavelet transform. In: **2022 IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)**. [S.l.: s.n.], 2022. p. 159–164.

19 RUIZ, M. et al. A novel algorithm for high compression rates focalized on electrical power quality signals. **Heliyon**, v. 7, n. 3, p. e06475, 2021. ISSN 2405-8440. Disponível em: <https://www.sciencedirect.com/science/article/pii/S2405844021005806>.

20 TRAVASSOS, F. C. et al. Detecção adaptativa de novidades baseada em métrica de similaridade aplicada a sinais de qualidade de energia. **Anais do Congresso Brasileiro de Automática 2020**, 12 2020.

21 KAPISCH, E. B.; DUARTE, J. L.; DUQUE, C. A. Single-phase fundamental component estimation and frequency tracking under time-varying harmonic distortion operation. In: **2018 20th European Conference on Power Electronics and Applications (EPE'18 ECCE Europe)**. [S.l.: s.n.], 2018. p. P.1–P.9.

22 Analog Devices. **AD5293: Single-Channel, 1024-Position, 1% R-Tolerance Digital Potentiometer Datasheet**. 2023. [Online; accessed 20-01-2024]. Disponível em: <https://www.analog.com/en/products/ad5293.html?doc=AD5293.pdf>.

23 LAB, A. D. **Ping-Pong Buffer Audio Stream**. n.d. Accessed: 2024-05-15. Disponível em: <https://audiodsplab.wordpress.com/ping-pong-buffer-audio-stream/>.

24 INSTRUMENTS, T. **AN-1222 USBN9603/4 - Increased Data Transfer Rate Using Ping-Pong Buffering**. [S.l.], n.d. Accessed: 2024-06-01. Disponível em: <https://www.ti.com/lit/an/snoa417/snoa417.pdf?ts=1721933516490>.

25 INC., M. T. **Ping-Pong Buffers - Online Documentation**. [S.l.], n.d. Accessed: 2024-07-13. Disponível em: <https://onlinedocs.microchip.com/pr/GUID-324A966D-1464-4B35-A7D1-DCAE052AC22C-en-US-3/index.html?GUID-B6995A5F-E06B-4071-893E-BBC60082F576>.

26 INC., M. T. **How to Use XDMAC on Cortex-M7 Microcontrollers to Implement Ping-Pong Buffering in Audio Applications**. n.d. Accessed: 2024-06-26. Disponível em: <https://ww1.microchip.com/downloads/aemDocuments/documents/OTH/ProductDocuments/SupportingCollateral/90003172A.pdf>.

27 CORMEN, T. H. et al. **Introduction to Algorithms**. 3rd. ed. [S.l.]: MIT Press, 2009. 258–264 p.

28 KAPISCH, E. et al. **Real-life current signal from a wind generator**. IEEE Dataport, 2019. Disponível em: <https://dx.doi.org/10.21227/pvkz-nm74>.

A CÓDIGO-FONTE DA COMPRESSÃO

A.1 Implementação em C/C++

Listing A.1 – Escopo principal (svda.ino).

```
1  // Spectral Variation Detection Algorithm
2  #include "config.h" // General
3
4  #define TAG "STATES MACHINE"
5  #define SPI1_CLOCK_MHZ 8
6  //*****************************************
7  // GLOBAL VARIABLES AND FUNCTION PROTOTYPES
8  //*****************************************
9  // Controls the states machine (switch case)
10 enum State state = READ_SAMPLING_PARAMETERS;
11
12 // Path
13 #define SAMPLE_DIRECTORY "/samples"
14 #define EXTENSION_BIN ".bin"
15 #define EXTENSION_TXT ".txt"
16 String full_path_original_file, full_path_FFT, full_path_MC;
17 String sample_file = "20240423_132630-02";
18
19 // PARAMETERS
20 SampleParameters parameters;
21 uint8_t Nppc;
22 uint32_t num_frames = 0;
23
24 // BUFFERS
25 uint16_t* sample_adc;
26 double* vReal;
27 double* vImag;
28 uint8_t gCard_access_failure_counter = 0;
29
30 // Compression Matrix
31 CompressionMatrix mc;
32 Preambulo p;
33 uint32_t novelty = 0;
34
35 // Functions prototypes
36 static void SetupSPI(void);
```

```
37 static void SetupLEDs(void);
38 static void SetupPins(void);
39 static SampleParameters ReadSamplingParameters(String path);
40 void MallocAuxiliaryBuffers(uint16_t** sample_adc, double** vReal, double** vImag, uint8_t size
       );
41 void ReleaseAuxiliaryBuffers(uint16_t* sample_adc, double* vReal, double* vImag);
42 void PrintPreambulo(Preambulo p);
43 void PrintComplexVector(double *vReal, double *vImag, uint16_t bufferSize, bool close_key);
44 void Gamma(double gamma[], double gain, double beta, uint8_t length_frame);
45 void PrintGamma(double gain, double beta, uint8_t length_frame);
46 String FormatFullPathOriginalFile(String sample_file);
47 String CreateTemporaryFFTfile(String sample_file);
48 String CreateMCfile(String sample_file);
49
50 //************************
51 // MAIN CODE
52 //************************
53 void setup() {
54 Serial.begin(115200);
55
56 // Setup do pinos
57 DisableSPI();
58 SetupPins();
59 SPI_SelectESP();
60
61 full_path_original_file = FormatFullPathOriginalFile(sample_file);
62 }
63 // Infinite loop
64 void loop() {
65 // Finite-state machine (FSM)
66 switch (state) {
67     case READ_SAMPLING_PARAMETERS:
68     {
69         ESP_LOGI(TAG, "\nEntering READ_SAMPLING_PARAMETERS state");
70
71         // Reading sampling parameters stored in the source file
72         parameters = ReadSamplingParameters(full_path_original_file);
73         Serial.printf("SAMPLING PARAMETERS (Original):\n");
74         Serial.printf("- D: %d\n", parameters.d);
75         Serial.printf("- Number of samples: %d\n", parameters.sample_size);
76         Serial.printf("- Sampling frequency: %d\n", parameters.sample_freq);
77
```

```
78          // Estimating the number of samples per cycle (Nppc) and the number of frames needed
                to segment the original signal
79          Nppc = round(parameters.sample_freq / FUNDAMENTAL_FREQUENCY);
80          num_frames = round(parameters.sample_size / Nppc);
81          Serial.printf("- Nppc: %d\n", Nppc);
82          Serial.printf("- Frame number: %d\n", num_frames);
83
84          state = DFT_PER_CYCLE; //DFT_PER_CYCLE;
85          break;
86      }
87      case DFT_PER_CYCLE:
88      {
89          ESP_LOGI(TAG, "\nEntering DFT_PER_CYCLE state");
90          uint8_t length_frame = (Nppc/2) + 1; // simetria
91          Serial.printf("length_frame: %d\n", length_frame);
92
93          // Create temporary files to store each frame's FFT and MC
94          full_path_FFT = CreateTemporaryFFTfile(sample_file);
95
96          // Dynamic buffer allocation
97          MallocAuxiliaryBuffers(&sample_adc, &vReal, &vImag, Nppc);
98          // Create FFT object
99          ArduinoFFT<double> FFT = ArduinoFFT<double>(vReal, vImag, Nppc, parameters.
                sample_freq);
100
101         // Read one frame at a time
102         SD_GimpsiSetup(&gCard_access_failure_counter);
103         File original_file = SD.open(full_path_original_file, FILE_READ);
104         if (!original_file) {
105         ESP_LOGE(TAG, "Failed when trying to access files.");
106         } else {
107         // Read the file in blocks of Nppc elements until the end
108         ESP_LOGI(TAG, "Processing frames...");
109
110         for (uint32_t frame_index = 0; frame_index < num_frames; frame_index++) {
111         // for (uint32_t frame_index = 0; frame_index < 1; frame_index++) {
112             // Position the pointer in the file to the beginning of the next frame
113             uint32_t byte_offset_1 = frame_index * sizeof(uint16_t) * Nppc + 10;
114             original_file.seek(byte_offset_1);
115
116             // Read a frame of Nppc elements from the file;
117             size_t bytesRead = original_file.read((uint8_t*)sample_adc, Nppc*sizeof(uint16_t));
```

```
118             double dcComponet = 0;
119             if (bytesRead == Nppc*sizeof(uint16_t)) {
120             // Convert the value to Volts
121             for (uint16_t q = 0; q < Nppc; q++) {
122                 vReal[q] = ((uint16_t*)sample_adc)[q] * DIG_TO_VOLT;
123                 vImag[q] = 0.0;
124                 dcComponet += vReal[q];
125             }
126             // Calculate the DFT of each frame
127             if (frame_index == 0) PrintComplexVector(vReal, vImag, Nppc, true);
128             FFT.windowing(FFTWindow::Rectangle, FFTDirection::Forward);
129             FFT.compute(FFTDirection::Forward); // Compute FFT
130             vReal[0] = dcComponet; // 1/N not implemented in Cooley-Tukey FFT algorithm
131             if (frame_index == 0) PrintComplexVector(vReal, vImag, Nppc, true);
132
133             // Stores the frame's FFT in an auxiliary temporary file.
134             SD_GimpsiSetup(&gCard_access_failure_counter);
135             File file_frames = SD.open(full_path_FFT, FILE_WRITE);
136             if (file_frames) {
137                 double vReal_aux[length_frame];
138                 double vImag_aux[length_frame];
139                 for (uint16_t q = 0; q < (length_frame); q++) {
140                 vReal_aux[q] = vReal[q];
141                 vImag_aux[q] = vImag[q];
142                 }
143                 uint32_t byte_offset_2 = frame_index * sizeof(double) * length_frame*2;
144                 file_frames.seek(byte_offset_2);
145                 file_frames.write((uint8_t*)vReal_aux, sizeof(double) * length_frame);
146                 file_frames.write((uint8_t*)vImag_aux, sizeof(double) * length_frame);
147                 file_frames.close();
148             }
149
150             } else {
151             ESP_LOGE(TAG, "Error reading the frame.");
152             state = FINISHED;
153             break;
154             }
155         }
156         }
157         original_file.close();
158         ReleaseAuxiliaryBuffers(sample_adc, vReal, vImag);
159         state = SPECTRAL_VARIATION_ANALYSIS;
```

```
160         break;
161     }
162     case SPECTRAL_VARIATION_ANALYSIS:
163     {
164         ESP_LOGI(TAG, "\nEntering SPECTRAL_VARIATION_ANALYSIS state");
165         novelty=0;
166         uint8_t length_frame_fft = ((Nppc/2)+1)*2; // simetria
167         Serial.printf("length_frame_fft: %d\n", length_frame_fft);
168         // Read the FFT of Frames stored in the auxiliary file
169 
170         // Tolerance Value
171         PrintGamma(GAIN, BETA, length_frame_fft);
172 
173         // Auxiliary vectors
174         double MAX[length_frame_fft], MIN[length_frame_fft], SUM[length_frame_fft],
                FFT_ARRAY[length_frame_fft];
175         uint32_t CNT[length_frame_fft];
176 
177         SD_GimpsiSetup(&gCard_access_failure_counter);
178 
179         File file_frames = SD.open(full_path_FFT, FILE_READ);
180         // Cycling through each frame
181         for (uint32_t f = 0; f < num_frames; f++) {
182         uint32_t byte_offset = f * sizeof(double) * length_frame_fft;
183         file_frames.seek(byte_offset);
184         file_frames.read((uint8_t*)FFT_ARRAY, length_frame_fft * sizeof(double));
185 
186         if (f == 0) {
187             // Initialization of auxiliary vectors (First frame)
188             for (uint8_t q = 0; q < length_frame_fft; q++) {
189             MAX[q] = FFT_ARRAY[q]; MIN[q] = FFT_ARRAY[q]; SUM[q] = FFT_ARRAY[q];
                    CNT[q] = 1;
190             }
191         } else {
192             // Cycling through each element of the current frame
193             for (uint8_t q = 0; q < length_frame_fft; q++) {
194             if (FFT_ARRAY[q] > MAX[q]) MAX[q] = FFT_ARRAY[q];
195             if (FFT_ARRAY[q] < MIN[q]) MIN[q] = FFT_ARRAY[q];
196 
197             // Checking if the change is relevant (greater than tolerance)
198 
199             double threshold = GAIN * pow((q+1), -BETA);
```

```
200             if (fabs(MAX[q] - MIN[q]) > threshold) {
201                 // Variation identified
202                 novelty++;
203                 // Save relevant information for reconstruction
204                 uint32_t f_tmp = f - 1;
205                 mc.q.add(q); mc.f.add(f_tmp); mc.sum.add(SUM[q]); mc.cnt.add(CNT[q]);
206                 Serial.printf("MC[%d]: q = %d, f = %d, sum = %f, cnt = %d\n", novelty, (mc.
                        q.get(novelty-1)), (mc.f.get(novelty-1)), (mc.sum.get(novelty-1)), (mc.cnt.
                        get(novelty-1)));
207
208                 // Reset
209                 MAX[q] = FFT_ARRAY[q]; MIN[q] = FFT_ARRAY[q]; SUM[q] = FFT_ARRAY
                        [q]; CNT[q] = 1;
210             } else {
211                 // Nothing new (we count repetition and discard)
212                 SUM[q] = SUM[q] + FFT_ARRAY[q];
213                 CNT[q] = CNT[q] + 1;
214             }
215             }
216         }
217         }
218         file_frames.close();
219
220         // Stores the latest novlty/frame data
221
222         for (uint8_t q = 0; q < length_frame_fft; q++) {
223         novelty++;
224         uint32_t f_tmp = num_frames;
225         mc.q.add(q); mc.f.add(f_tmp); mc.sum.add(SUM[q]); mc.cnt.add(CNT[q]);
226         Serial.printf("MC[%d]: q = %d, f = %d, sum = %f, cnt = %d\n", novelty, (mc.q.get(
                novelty-1)), (mc.f.get(novelty-1)), (mc.sum.get(novelty-1)), (mc.cnt.get(novelty
                -1)));
227         }
228         Serial.printf("novelty = %d\n", novelty);
229         state = LZW;
230         break;
231     }
232     case LZW: // TODO: not implemented
233     {
234         ESP_LOGI(TAG, "\nEntering LZW state");
235         state = GENERATE_COMPRESSED_FILE;
236         break;
```

```
237     }
238     case GENERATE_COMPRESSED_FILE:
239     {
240         ESP_LOGI(TAG, "\nEntering GENERATE_COMPRESSED_FILE state");
241         // Create temporary files to store MC
242         full_path_MC = CreateMCfile(sample_file);
243         SD_GimpsiSetup(&gCard_access_failure_counter);
244         File file_MC = SD.open(full_path_MC, FILE_WRITE);
245         if (file_MC) {
246         // Preanbulo
247         uint32_t qP1 = p.length_mc_q; uint32_t qP2 = p.length_mc_f; uint32_t qP3 = p.
                length_mc_sum;
248         uint32_t qP4 = p.length_mc_cnt; uint32_t qP5= p.nppc; uint32_t qP6 = p.
                num_frames;
249         file_MC.write((uint8_t*)&qP1, sizeof(uint32_t));
250         file_MC.write((uint8_t*)&qP2, sizeof(uint32_t));
251         file_MC.write((uint8_t*)&qP3, sizeof(uint32_t));
252         file_MC.write((uint8_t*)&qP4, sizeof(uint32_t));
253         file_MC.write((uint8_t*)&qP5, sizeof(uint32_t));
254         file_MC.write((uint8_t*)&qP6, sizeof(uint32_t));
255 
256         // MC
257         for (uint32_t i = 0; i < mc.q.size(); i++) {
258             uint8_t qValue = mc.q.get(i);
259             file_MC.write((uint8_t*)&qValue, sizeof(qValue));
260         }
261         for (uint32_t i = 0; i < mc.f.size(); i++) {
262             uint32_t fValue = mc.f.get(i);
263             file_MC.write((uint8_t*)&fValue, sizeof(fValue));
264         }
265         for(uint32_t i=0; i<mc.sum.size(); i++){
266             double sumValue = mc.sum.get(i);
267             file_MC.write((uint8_t*)&sumValue, sizeof(sumValue));
268         }
269         for (uint32_t i = 0; i < mc.cnt.size(); i++) {
270             uint32_t cntValue = mc.cnt.get(i);
271             file_MC.write((uint8_t*)&cntValue, sizeof(cntValue));
272         }
273         ESP_LOGI(TAG, "Compressed signal saved in: %s\n", file_MC.name());
274         file_MC.close();
275         }
276         state = FINISHED;
```

```
277             break;
278         }
279         case FINISHED:
280         {
281             state = FINISHED;
282             break;
283         }
284 }
285 }
```

Listing A.2 – Funções Auxiliares (svda.ino).

```
1  //*****************************
2  // AUXILIARY FUNCTIONS
3  //*****************************
4
5  void Gamma(double gamma[], double gain, double beta, uint8_t length_frame) {
6  for (uint8_t q = 0; q < length_frame; q++) {
7      gamma[q] = gain * pow(q, -beta);
8  }
9  }
10 void PrintGamma(double gain, double beta, uint8_t length_frame){
11 Serial.print("threshold = [");
12 for (uint8_t q = 0; q < length_frame; q++) {
13     Serial.printf("%f, ", (gain * pow((q+1), -beta)));
14 }
15 Serial.print("];\n");
16 }
17
18 void PrintComplexVector(double *vReal, double *vImag, uint16_t bufferSize, bool close_key){
19 if(close_key) Serial.print("v = [");
20 for(uint8_t q=0; q < bufferSize; q++){
21     Serial.printf("%f%+fj, ", vReal[q], vImag[q]);
22 }
23 if(close_key) Serial.print("];\n");
24 }
25
26 void PrintPreambulo(Preambulo p) {
27 Serial.print("PREANBULO: \n");
28 Serial.printf("- length_mc_q = %d\n", p.length_mc_q);
29 Serial.printf("- length_mc_f = %d\n", p.length_mc_f);
30 Serial.printf("- length_mc_sum = %d\n", p.length_mc_sum);
31 Serial.printf("- length_mc_cnt = %d\n", p.length_mc_cnt);
```

```
32 Serial.printf("- Nppc = %d\n", p.nppc);
33 Serial.printf("- num_frames = %d\n", p.num_frames);
34 }
35
36 static SampleParameters ReadSamplingParameters(String path) {
37 SampleParameters parameters = { 0, 0, 0 };
38 SD_GimpsiSetup(&gCard_access_failure_counter);
39 File file = SD.open(path, FILE_READ);
40 if (!file) {
41     ESP_LOGE(TAG, "Failed when trying to read the file %s", file.name());
42 } else {
43     ESP_LOGI(TAG, "%s file opened successfully.", file.name());
44     // Read sampling parameters from file
45     file.read((uint8_t*)&(parameters.sample_freq), sizeof(parameters.sample_freq));
46     uint16_t number_of_samples_low, number_of_samples_high;
47     file.read((uint8_t*)&number_of_samples_low, sizeof(number_of_samples_low));
48     file.read((uint8_t*)&number_of_samples_high, sizeof(number_of_samples_high));
49     parameters.sample_size = (uint32_t)number_of_samples_low + ((uint32_t)
           number_of_samples_high);
50     file.read((uint8_t*)&(parameters.d), sizeof(parameters.d));
51     file.close();
52 }
53 return parameters;
54 }
55
56 String FormatFullPathOriginalFile(String sample_file) {
57 // Original file
58 String full_path = "/";
59 String full_path_original_file = SAMPLE_DIRECTORY "/";
60 full_path.concat(sample_file);
61 full_path_original_file.concat(sample_file);
62 full_path_original_file.concat(EXTENSION_BIN);
63 ESP_LOGI(TAG, "Full path to the original file: %s", full_path_original_file.c_str());
64 return full_path_original_file;
65 }
66
67 String CreateTemporaryFFTfile(String sample_file) {
68 String full_path_FFT;
69 String full_path = "/";
70 String sample_file_aux = "*";
71 sample_file_aux.concat(sample_file);
72 String timestamp = sample_file.substring(sample_file_aux.indexOf("*"));
```

```
73
74  // DFT of each frame
75  full_path_FFT = SAMPLE_DIRECTORY "/";
76  full_path_FFT.concat(timestamp);
77  full_path_FFT.concat("_FFT");
78  full_path_FFT.concat(EXTENSION_BIN);
79
80  SD_GimpsiSetup(&gCard_access_failure_counter);
81  File file_fft = SD.open(full_path_FFT, FILE_WRITE);
82  if (!file_fft) {
83      ESP_LOGE(TAG, "File creation failed %s", file_fft.name());
84  } else {
85      ESP_LOGI(TAG, "Full path to the DFT of each frame: %s", full_path_FFT.c_str());
86  }
87  file_fft.close();
88  return full_path_FFT;
89  }
90
91  String CreateMCfile(String sample_file) {
92  String full_path_MC;
93  String full_path = "/";
94  String sample_file_aux = "*";
95  sample_file_aux.concat(sample_file);
96  String timestamp = sample_file.substring(sample_file_aux.indexOf("*"));
97
98  // Compression matrix
99  full_path_MC = SAMPLE_DIRECTORY "/";
100 full_path_MC.concat(timestamp);
101 full_path_MC.concat("_MC");
102 full_path_MC.concat(EXTENSION_BIN);
103
104 SD_GimpsiSetup(&gCard_access_failure_counter);
105 File file_mc = SD.open(full_path_MC, FILE_WRITE);
106 if (!file_mc) {
107     ESP_LOGE(TAG, "File creation failed %s", full_path_MC.c_str());
108 } else {
109     ESP_LOGI(TAG, "Full path to the compression matrix: %s", full_path_MC.c_str());
110 }
111 file_mc.close();
112 return full_path_MC;
113 }
114
```

```
115 void MallocAuxiliaryBuffers(uint16_t** sample_adc, double** vReal, double** vImag, uint8_t size
        ) {
116 *sample_adc = (uint16_t*)malloc(sizeof(uint16_t) * size);
117 *vReal = (double*)malloc(sizeof(double) * size);
118 *vImag = (double*)malloc(sizeof(double) * size);
119 ESP_LOGI(TAG, "Allocation of auxiliary buffers carried out successfully.");
120 }
121 void ReleaseAuxiliaryBuffers(uint16_t* sample_adc, double* vReal, double* vImag) {
122 free(sample_adc);
123 free(vReal);
124 free(vImag);
125 ESP_LOGI(TAG, "The deallocation of memory from auxiliary buffers was successful.");
126 }
127
128 static void SetupSPI(void) {
129 pinMode(MISO, INPUT);
130 pinMode(MOSI, OUTPUT);
131 pinMode(SCK, OUTPUT);
132 pinMode(CS_PIN, OUTPUT);
133 digitalWrite(MOSI, HIGH);
134 digitalWrite(SCK, HIGH);
135 digitalWrite(CS_PIN, HIGH);
136 }
137
138 static void SetupPins(void) {
139 pinMode(SPI_SEL, OUTPUT);
140 digitalWrite(SPI_SEL, HIGH);
141 }
142
143 static void SetupLEDs(void) {
144 pinMode(LEDR, OUTPUT);
145 pinMode(LEDY, OUTPUT);
146 pinMode(LEDG, OUTPUT);
147 digitalWrite(LEDR, LOW);
148 digitalWrite(LEDY, LOW);
149 digitalWrite(LEDG, HIGH);
150 }
```

Listing A.3 – Configurações (config.h)

```
1 #ifndef _CONFIG_H
2 #define _CONFIG_H
3
```

```
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <string.h>
7  #include <time.h>
8  #include <LinkedList.h>
9  #include <math.h>
10 #include "sd_gimpsi.h"
11 #include "arduinoFFT.h"
12
13 #define FUNDAMENTAL_FREQUENCY 60 // in HZ
14 #define DIG_TO_VOLT 0.0008058608 // 3.3 / 4095.0
15
16 // Threshold
17 #define GAIN 1
18 #define BETA 0.001
19
20 // Finite States Machine
21 enum State {
22     READ_SAMPLING_PARAMETERS,
23     SEGMENTATION,
24     DFT_PER_CYCLE,
25     SPECTRAL_VARIATION_ANALYSIS,
26     LZW,
27     GENERATE_COMPRESSED_FILE,
28     FINISHED,
29 };
30
31 struct SampleParameters {
32     uint16_t d;
33     uint16_t sample_size;
34     uint32_t sample_freq;
35 };
36
37 struct Preambulo{
38     uint32_t length_mc_q;
39     uint32_t length_mc_f;
40     uint32_t length_mc_sum;
41     uint32_t length_mc_cnt;
42     uint32_t nppc;
43     uint32_t num_frames;
44 };
45
```

```
// Compression Matrix
struct CompressionMatrix {
    LinkedList<uint8_t> q;
    LinkedList<uint32_t> f;
    LinkedList<double> sum;
    LinkedList<uint32_t> cnt;
};

struct CompressedData{
    Preambulo p;
    CompressionMatrix mc;
};

// GPIOs
#define LEDR 27
#define LEDY 12
#define LEDG 2
#define SPI_SEL 25

#endif
```

A.2 Implementação no MATLAB

Listing A.4 – Escopo principal (svda.m).

```
% SPECTRAL VARIATION DETECTION ALGORITHM
clc
close all
clear

%% INITIAL SETTINGS
original_file_name = 'samples_utepsi\AAAAMMDD_HHMMSS-ID.bin';
type_compression = 'matlab'; % embedded ou matlab
using_symmetry = 1; % 1 - com simetria | 2 - sem simetria
preamble_type = 2; % 1 - Fs | 2 - Ts
converter_volts = 1; % 1 - sim | 2- Não

%% READING SAMPLES FROM A GIMPSI-TYPE FILE
switch type_compression
    case 'matlab'
        path = 'C:\Users\...\svda\sample_files\';
    case 'embedded'
```

```
18          path = 'E:\samples\';
19      otherwise
20          error('Unexpected value: type_compression = %s. Execution is stopped.',
                type_compression);
21  end
22
23  % ORIGINAL FILE NAME
24  full_path_original_file = strcat(path,original_file_name);
25
26  % OPEN ORIGINAL FILE
27  [signal, D, number_of_samples, sampling_period_us, Fs_orig] = open_gimpsi_file(
        full_path_original_file, converter_volts, preamble_type);
28
29  %% RUN COMPRESSION (IN MATLAB)
30  if (strcmpi(type_compression,'matlab'))
31      G = 1;
32      beta = 0.001;
33      Fs_new = 7680; %Fs_new = Fs_orig (does not enable resampling)
34      [MC, Nppc, num_frames, gamma] = compression(signal, G, beta, Fs_orig, Fs_new,
            using_symmetry);
35
36      % COMPRESSED FILE NAME
37      compressed_file_name = strcat(original_file_name(1:end-4), '_MC-MATLAB.bin');
38      full_path_compressed_file = strcat(path ,compressed_file_name);
39
40      % Saving compressed data to a .bin file
41      preambulo = [length(MC(1,:)) length(MC(2,:)) length(MC(3,:)) length(MC(4,:)) Nppc
            num_frames];
42
43      fprintf('Compressed signal saved in: %s\n',full_path_compressed_file);
44      file = fopen(full_path_compressed_file, 'w+b');
45      if file == -1
46          error('Unable to open file for writing.');
47      end
48
49      fwrite(file, preambulo, 'uint32');
50      fwrite(file, MC(1,:), 'uint16');
51      fwrite(file, MC(2,:), 'uint32');
52      fwrite(file, MC(3,:), 'double');
53      fwrite(file, MC(4,:), 'uint32');
54      fclose(file);
55
```

```
elseif (strcmpi(type_compression,'embedded'))
    % COMPRESSED FILE NAME
    compressed_file_name = strcat(original_file_name(1:end-4), '_MC.bin');
    full_path_compressed_file = strcat(path ,compressed_file_name);
else
    error('Unexpected value: type_compression = %s. Execution is stopped.', type_compression);
end

%% RECINSTRUCTION
if (strcmpi(type_compression,'matlab'))
    [reconstructed_signal, FFT_ARRAY_r, MR, Nppc, num_frames] =
        decompress_signal_matlab(full_path_compressed_file, Fs_orig, using_symmetry);
elseif (strcmpi(type_compression,'embedded'))
    [reconstructed_signal, FFT_ARRAY_r, MR, Nppc, num_frames] = decompress_signal_esp32
        (full_path_compressed_file, using_symmetry);
end
```

Listing A.5 – Função Auxiliar (compression.m).

```
function [MC, Nppc, num_frames, gamma] = compression(original_signal, G, beta, Fs_orig,
    Fs_new, using_symmetry)
%% Processing the original signal so that it can be segmented into Nppc sample-sized frames:
    if (Fs_orig ~= Fs_new)
        disp('Resampling the signal ...')
        signal = resample(original_signal, round(Fs_new), round(Fs_orig));
        % Checking if the signal was resampled correctly
        signal_size = length(signal);
        t = signal_size/Fs_new;
        Fs_calculated_resampled_signal = signal_size / t;

        % Check whether the new calculated sampling frequency is equal to the new desired
            sampling frequency
        if abs(Fs_calculated_resampled_signal - Fs_new) > eps
            fprintf('Error: The sampling frequency was not changed correctly [Fs = %d Hz].\n',
                Fs_calculated_resampled_signal);
        else
            fprintf('Sampling frequency successfully changed to Fs = %d Hz.\n',
                Fs_calculated_resampled_signal);
        end
    else
        signal = original_signal;
        signal_size = length(signal);
    end
```

```
21
22 %% Segmentation of the original signal into frames:
23     f_estimated = 60; % frequency_estimator(original_signal, Fs_orig);
24     Nppc = Fs_new*(1/f_estimated);
25     num_frames = floor(signal_size / Nppc);
26     fprintf('Nf = %d/%d = %d\n', signal_size, Nppc, num_frames);
27
28     frames = zeros(num_frames, Nppc); % linhas: frame; colunas: N_ppc elementos
29
30     % Segment the signal into frames of size N_ppc
31     next_frame_start = 1;
32     for i = 1:num_frames
33         frames(i, :) = signal(next_frame_start:next_frame_start + Nppc - 1);
34         next_frame_start = next_frame_start + Nppc;
35     end
36     fprintf('Segmented into %d frames of %d samples.\n', size(frames));
37
38 %% FFT per Cycle:
39     % FFT of each line/frame of the frames a matrix separately = symmetry with respect to
           Nppc
40     frames_fft = fft(frames, [], 2);
41
42     switch using_symmetry
43         case 1 % with the use of symmetry
44             FFT_ARRAY_TMP = [real(frames_fft(:, 1:((Nppc/2)+1))), imag(frames_fft(:, 1:((
                   Nppc/2)+1)))];
45         case 2 % without the use of symmetry
46             FFT_ARRAY_TMP = [real(frames_fft), imag(frames_fft)];
47         otherwise
48             error('Unexpected value: using_symmetry = %d. Execution is stopped.',
                   using_symmetry);
49     end
50     FFT_ARRAY = FFT_ARRAY_TMP;
51     fprintf('size(frames_fft): [%d, %d]\n', size(frames_fft));
52     fprintf('size(FFT_ARRAY): [%d, %d]\n', size(FFT_ARRAY));
53
54 %% Spectral Variation Analysis:
55     fprintf('Threshold: G = %f ; beta = %f\n', G, beta);
56     [MC, gamma] = spectral_variation_compression(FFT_ARRAY, G, beta);
57
58 end
```

Listing A.6 – Função Auxiliar (spectral_variation_compression.m).

```
function [MC, gamma] = spectral_variation_compression(FFT_ARRAY, G, beta)
    [num_frames, fft_array_length] = size(FFT_ARRAY);
    fprintf('Spectral variation compression of %d frames with %d samples ...\n', num_frames,
        fft_array_length);
    % Compression matrix initialization
    MC = [];
    novelty = 0;

    % Threshold
    gamma = G * (1:fft_array_length).^(-beta);

    % Initialization of auxiliary vectors (First frame)
    MAX = FFT_ARRAY(1, :);
    MIN = FFT_ARRAY(1, :);
    SUM = FFT_ARRAY(1, :);
    CNT = ones(1, fft_array_length);

    % Cycling through each frame
    for f = 2:num_frames
        NEW_FFT_ARRAY = FFT_ARRAY(f, :);
        % Cycling through each element of the frame
        for q = 1:fft_array_length
            new_fft_array = NEW_FFT_ARRAY(q);
            if new_fft_array > MAX(q)
                MAX(q) = new_fft_array;
            end
            if new_fft_array < MIN(q)
                MIN(q) = new_fft_array;
            end
            % Checking if the change is relevant (greater than tolerance)
            if abs(MAX(q)-MIN(q)) > (G * power(q, -beta))
                % Novlty detectio
                novelty = novelty + 1;
                % Save relevant information for reconstruction
                MC(1, novelty) = q; % position where the novelty was detected
                MC(2, novelty) = f-1; % frame of the last occurrence
                MC(3, novelty) = SUM(q); % cumulative sum
                MC(4, novelty) = CNT(q); % number of repetitions of this novelty

                % Reset
                MAX(q) = new_fft_array; MIN(q) = new_fft_array;
```

```
                SUM(q) = new_fft_array; CNT(q) = 1;
            else
                % Nothing novlty (we count repetition and discard)
                SUM(q) = SUM(q) + new_fft_array;
                CNT(q) = CNT(q)+1; % one more repetition
            end
        end
    end

    % Stores latest novelty data
    for q = 1:fft_array_length
            novelty = novelty + 1;
            MC(:, end+1) = [q; f; SUM(q); CNT(q)];
    end

    fprintf('> %d novelty were detected.\n', novelty);
    fprintf('size(MC) = [%d, %d]\n', size(MC));
end
```

A.3 Implementação em Python

Listing A.7 – Escopo principal (app.py).

```
import sys
import os
import tempfile
# Redirect stderr to a temporary file to capture errors
stderr_file = tempfile.NamedTemporaryFile(delete=False)
sys.stderr = open(stderr_file.name, 'w')

# Configure Matplotlib backend
import matplotlib
matplotlib.use('TkAgg')

import tkinter as tk
from tkinter import filedialog
from tkinter import messagebox
from tkinter.scrolledtext import ScrolledText
import numpy as np
import matplotlib.pyplot as plt
from open_gimpsi_file import open_gimpsi_file
from compression import compression, save_compressed_file
```

```
20 from decompress_signal import decompress_signal_local, decompress_signal_embedded,
       save_reconstructed_signal
21
22 class CompressionApp:
23     def __init__(self, root):
24         self.root = root
25         self.root.title("GImpSI – Leak Signal Compressor")
26         # Determine the icon path
27         if hasattr(sys, '_MEIPASS'):
28             icon_path = os.path.join(sys._MEIPASS, "icon.ico")
29         else:
30             icon_path = os.path.join(os.path.dirname(__file__), "icon.ico")
31
32         # Set window icon
33         self.root.iconbitmap(icon_path)
34
35         self.Fs_orig = 0 # Initialize Fs_orig as an instance variable
36
37         self.create_widgets()
38
39     def create_widgets(self):
40         # Parameters frame
41         param_frame = tk.LabelFrame(self.root, text="Parametros", padx=10, pady=10)
42         param_frame.pack(padx=10, pady=10)
43
44         # User-selectable parameters
45         self.create_label_dropdown(param_frame, "converter_para_volts", "Converter para Volts:
               ", ["Sim", "Não"], "Sim")
46         self.create_label_dropdown(param_frame, "usar_simetria", "Usar Simetria:", ["Sim", "Nã
               o"], "Sim")
47         self.create_label_entry(param_frame, "g", "G:", 1)
48         self.create_label_entry(param_frame, "beta", "Beta:", 0.001)
49         self.create_label_dropdown(param_frame, "tipo_de_preambulo", "Tipo de Preambulo:",
               ["Fs", "Ts"], "Fs")
50         self.create_label_entry(param_frame, "nova_freq_de_amostragem_fs_new", "Habilite a
               reamostragem (Fs_new):", 7680)
51
52         # Action buttons
53         action_frame = tk.Frame(self.root)
54         action_frame.pack(pady=20)
55
```

```
56          compress_button = tk.Button(action_frame, text="Comprimir", command=self.
                compress_file)
57          compress_button.grid(row=0, column=0, padx=10)
58
59          decompress_button = tk.Button(action_frame, text="Descomprimir", command=self.
                decompress_file)
60          decompress_button.grid(row=0, column=1, padx=10)
61
62          evaluate_button = tk.Button(action_frame, text="Avaliar", command=self.
                evaluate_reconstruction)
63          evaluate_button.grid(row=0, column=2, padx=10)
64
65          open_button = tk.Button(action_frame, text="Abrir", command=self.open_file)
66          open_button.grid(row=0, column=3, padx=10)
67
68          # Log display
69          log_frame = tk.LabelFrame(self.root, text="Logs", padx=10, pady=10)
70          log_frame.pack(padx=10, pady=10, fill=tk.BOTH, expand=True)
71
72          self.log_text = ScrolledText(log_frame, wrap=tk.WORD, height=10)
73          self.log_text.pack(fill=tk.BOTH, expand=True)
74
75      def create_label_entry(self, frame, attr_name, label_text, default_value):
76          row = len(frame.winfo_children()) // 2
77          label = tk.Label(frame, text=label_text)
78          label.grid(row=row, column=0, padx=5, pady=5, sticky=tk.W)
79
80          entry = tk.Entry(frame)
81          entry.grid(row=row, column=1, padx=5, pady=5)
82          entry.insert(0, default_value)
83          setattr(self, attr_name, entry)
84
85      def create_label_dropdown(self, frame, attr_name, label_text, options, default_value):
86          row = len(frame.winfo_children()) // 2
87          label = tk.Label(frame, text=label_text)
88          label.grid(row=row, column=0, padx=5, pady=5, sticky=tk.W)
89
90          variable = tk.StringVar(frame)
91          dropdown = tk.OptionMenu(frame, variable, *options)
92          variable.set(default_value)
93          dropdown.grid(row=row, column=1, padx=5, pady=5)
94          setattr(self, attr_name, variable)
```

```
95
96      def compress_file(self):
97          filepath = filedialog.askopenfilename()
98          if not filepath:
99              return
100
101         converter_volts = 1 if self.converter_para_volts.get() == "Sim" else 0
102         using_symmetry = 1 if self.usar_simetria.get() == "Sim" else 0
103         G = float(self.g.get())
104         beta = float(self.beta.get())
105         preamble_type = 1 if self.tipo_de_preambulo.get() == "Fs" else 2
106         Fs_new = int(self.nova_freq_de_amostragem_fs_new.get())
107
108         samples, D, number_of_samples, sampling_period_us, self.Fs_orig = open_gimpsi_file(
                filepath, converter_volts, preamble_type)
109
110         self.log(f"Arquivo original: {filepath}")
111         self.log(f"PARAMETROS DE AMOSTRAGEM (Originais):")
112         self.log(f" - D: {D}")
113         self.log(f" - Ns: {number_of_samples}")
114         self.log(f" - Ts: {sampling_period_us * 1e-6} s")
115         self.log(f" - Fs: {self.Fs_orig} Hz")
116
117         MC, Nppc, num_frames, gamma = compression(samples, G, beta, self.Fs_orig, Fs_new,
                using_symmetry)
118
119         # Ask the user where to save the compressed file
120         save_path = filedialog.askdirectory()
121         if not save_path:
122             return
123
124         full_path_compressed_file = save_compressed_file(MC, Nppc, num_frames, filepath,
                save_path)
125
126         # Checking the size in bytes of the original and compressed files
127         fileInfo_org = os.stat(filepath)
128         fileInfo_comp = os.stat(full_path_compressed_file)
129
130         if not fileInfo_org or not fileInfo_comp:
131             messagebox.showerror("Erro", "O arquivo não existe.")
132         else:
133             self.log('TAMANHO EM BYTES DOS ARQUIVOS:')
```

```
134             self.log(f'— Arquivo original: {fileInfo_org.st_size} bytes')
135             self.log(f'— Arquivo comprimido: {fileInfo_comp.st_size} bytes')
136             taxa = fileInfo_org.st_size / fileInfo_comp.st_size
137             self.log(f'Taxa de compressão: {taxa:.2f}')
138
139         messagebox.showinfo("Sucesso", f"Arquivo comprimido: {full_path_compressed_file}")
140         self.log(f"Arquivo comprimido salvo em: {full_path_compressed_file}")
141
142     def decompress_file(self):
143         filepath = filedialog.askopenfilename()
144         if not filepath:
145             return
146
147         decompression_method = messagebox.askyesno("Método de Descompressão", "Este
                arquivo foi comprimido pela UMSCF?")
148         using_symmetry = 1 if self.usar_simetria.get() == "Sim" else 0
149
150         if decompression_method:
151             reconstructed_signal, FFT_ARRAY, MR, Nppc, num_frames =
                    decompress_signal_embedded(filepath, using_symmetry)
152         else:
153             reconstructed_signal, FFT_ARRAY, MR, Nppc, num_frames =
                    decompress_signal_local(filepath, self.Fs_orig, using_symmetry)
154
155         # Ask the user where to save the reconstructed file
156         save_path = filedialog.askdirectory()
157         if not save_path:
158             return
159
160         Fs_current = Nppc * 60
161         full_path_reconstructed_file = save_reconstructed_signal(reconstructed_signal, filepath,
                Fs_current, save_path)
162         messagebox.showinfo("Sucesso", f"Arquivo descomprimido: {full_path_reconstructed_file}
                ")
163         self.log(f"Arquivo descomprimido salvo em: {full_path_reconstructed_file}")
164
165     def open_file(self):
166         filepath = filedialog.askopenfilename()
167         if not filepath:
168             return
169
170         converter_volts = 1 if self.converter_para_volts.get() == "Sim" else 0
```

```
171         preamble_type = 1 if self.tipo_de_preambulo.get() == "Fs" else 2
172
173         samples, D, number_of_samples, sampling_period_us, self.Fs_orig = open_gimpsi_file(
                filepath, converter_volts, preamble_type)
174
175         self.log(f"Arquivo: {filepath}")
176         self.log(f"PARAMETROS DE AMOSTRAGEM:")
177         self.log(f" - D: {D}")
178         self.log(f" - Ns: {number_of_samples}")
179         self.log(f" - Ts: {sampling_period_us * 1e-6} s")
180         self.log(f" - Fs: {self.Fs_orig} Hz")
181
182         # Preview
183         plt.plot(samples)
184         plt.title('Sinal')
185         plt.xlabel('Sample')
186         plt.ylabel('Amplitude')
187         plt.grid(True)
188         plt.show()
189
190
191
192     def evaluate_reconstruction(self):
193         original_file = filedialog.askopenfilename(title="Selecione o arquivo original")
194         reconstructed_file = filedialog.askopenfilename(title="Selecione o arquivo reconstruido")
195         if not original_file or not reconstructed_file:
196             return
197
198         # Open the files
199         converter_volts = 1 if self.converter_para_volts.get() == "Sim" else 0
200         preamble_type = 1 if self.tipo_de_preambulo.get() == "Fs" else 2
201
202         original_signal, _, _, _, Fs_orig = open_gimpsi_file(original_file, converter_volts,
                preamble_type)
203         reconstructed_signal, _, _, _, _ = open_gimpsi_file(reconstructed_file, converter_volts,
                 preamble_type)
204
205         # Assessment of Reconstruction Quality Metrics
206         NMSE = (np.sum((original_signal - reconstructed_signal) ** 2)) / np.sum(
                original_signal ** 2)
207         RTE = (np.sum(reconstructed_signal ** 2) / np.sum(original_signal ** 2)) * 100
208
```

```
209             self.log("QUALIDADE DA RECONSTRUCAO:")
210             self.log(f"- NMSE = {NMSE:.7f}")
211             self.log(f"- RTE = {RTE:.2f}%")
212 
213             # Plot graphs for visual comparison
214             fig, ax = plt.subplots(figsize=(12, 6))
215             time_orig = np.linspace(0, len(original_signal) / Fs_orig, len(original_signal))
216             time_rec = np.linspace(0, len(reconstructed_signal) / Fs_orig, len(reconstructed_signal))
217 
218             ax.plot(time_orig, original_signal, label='Sinal Original', color='blue')
219             ax.plot(time_rec, reconstructed_signal, label='Sinal Reconstruido', linestyle='--', color
                    ='orange')
220             ax.plot(time_orig, original_signal - reconstructed_signal, label='Erro', color='red')
221             ax.set_title("Sinais Original, Reconstruido e Erro")
222             ax.set_xlabel("Tempo (s)")
223             ax.set_ylabel("Amplitude")
224             ax.legend()
225             ax.grid()
226 
227             plt.tight_layout()
228             plt.show()
229 
230         def log(self, message):
231             self.log_text.insert(tk.END, message + "\n")
232             self.log_text.see(tk.END)
233 
234 
235 if __name__ == "__main__":
236     root = tk.Tk()
237     app = CompressionApp(root)
238     root.mainloop()
```

Listing A.8 – Função Auxiliar (compression.py).

```
1 import sys
2 import os
3 import tempfile
4 # Redirect stderr to a temporary file to capture errors
5 stderr_file = tempfile.NamedTemporaryFile(delete=False)
6 sys.stderr = open(stderr_file.name, 'w')
7 
8 import numpy as np
9 import struct
```

```
10 from scipy.fft import fft
11 from scipy.signal import resample
12 from spectral_variation_compression import spectral_variation_compression
13
14 def compression(original_signal, G, beta, Fs_orig, Fs_new, using_symmetry):
15     if Fs_orig != Fs_new:
16         print('Resampling the signal ...')
17         signal = resample(original_signal, int(len(original_signal) * Fs_new / Fs_orig))
18         signal_size = len(signal)
19         t = signal_size / Fs_new
20         Fs_calculated_resampled_signal = signal_size / t
21
22         if abs(Fs_calculated_resampled_signal - Fs_new) > np.finfo(float).eps:
23             print(f'Error: The sampling frequency was not changed correctly [Fs = {
                   Fs_calculated_resampled_signal} Hz].')
24         else:
25             print(f'Sampling frequency successfully changed to Fs = {
                   Fs_calculated_resampled_signal} Hz.')
26     else:
27         print('Resampling Desabilitado.')
28         signal = original_signal
29         signal_size = len(signal)
30
31     f_estimated = 60
32     Nppc = int(Fs_new * (1 / f_estimated))
33     num_frames = signal_size // Nppc
34     print(f'Frames = {signal_size}/{Nppc} = {num_frames}')
35
36     frames = np.array([signal[i * Nppc:(i + 1) * Nppc] for i in range(num_frames)])
37     print(f'Segmented into {frames.shape[0]} frames of {frames.shape[1]} samples.')
38
39     frames_fft = fft(frames, axis=1)
40
41     if using_symmetry == 1:
42         FFT_ARRAY = np.concatenate((frames_fft[:, :(Nppc // 2 + 1)].real,
43                                     frames_fft[:, :(Nppc // 2 + 1)].imag), axis=1)
44     elif using_symmetry == 2:
45         FFT_ARRAY = np.concatenate((frames_fft.real, frames_fft.imag), axis=1)
46     else:
47         raise ValueError(f'Unexpected value: using_symmetry = {using_symmetry}. Execution is
               stopped.')
48
```

```
49      print(f'size(frames_fft): {frames_fft.shape}\nsize(FFT_ARRAY): {FFT_ARRAY.shape}')
50
51      print(f'Threshold: G = {G} ; beta = {beta}')
52      MC, gamma = spectral_variation_compression(FFT_ARRAY, G, beta)
53
54      return MC, Nppc, num_frames, gamma
55
56
57  def save_compressed_file(MC, Nppc, num_frames, original_file_name, path):
58      compressed_file_name = os.path.basename(original_file_name)[:-4] + '_c.bin'
59      full_path_compressed_file = os.path.join(path, compressed_file_name)
60
61      preambulo = [len(MC[0]), len(MC[1]), len(MC[2]), len(MC[3]), Nppc, num_frames]
62
63      print(f'Compressed signal saved in: {full_path_compressed_file}')
64      with open(full_path_compressed_file, 'wb') as file:
65          file.write(struct.pack('I' * len(preambulo), *preambulo))
66          file.write(struct.pack('H' * len(MC[0]), *MC[0].astype(np.uint16)))
67          file.write(struct.pack('I' * len(MC[1]), *MC[1].astype(np.uint32)))
68          file.write(struct.pack('d' * len(MC[2]), *MC[2].astype(np.float64)))
69          file.write(struct.pack('I' * len(MC[3]), *MC[3].astype(np.uint32)))
70      return full_path_compressed_file
```

Listing A.9 – Função Auxiliar (spectral_variation_compression.py).

```
1   import numpy as np
2
3   def spectral_variation_compression(FFT_ARRAY, G, beta):
4       num_frames, fft_array_length = FFT_ARRAY.shape
5       print(f'Spectral variation compression of {num_frames} frames with {fft_array_length}
            samples ...')
6
7       MC = []
8       novelty = 0
9
10      gamma = G * np.arange(1, fft_array_length + 1) ** -beta
11
12      MAX = np.copy(FFT_ARRAY[0, :])
13      MIN = np.copy(FFT_ARRAY[0, :])
14      SUM = np.copy(FFT_ARRAY[0, :])
15      CNT = np.ones(fft_array_length)
16
17      for f in range(1, num_frames):
```

```
18          NEW_FFT_ARRAY = FFT_ARRAY[f, :]
19          for q in range(fft_array_length):
20              new_fft_array = NEW_FFT_ARRAY[q]
21              if new_fft_array > MAX[q]:
22                  MAX[q] = new_fft_array
23              if new_fft_array < MIN[q]:
24                  MIN[q] = new_fft_array
25
26              current_gamma = G * (q + 1) ** -beta
27
28              if abs(MAX[q] - MIN[q]) > current_gamma:
29                  novelty += 1
30                  MC.append([q + 1, f, SUM[q], CNT[q]])
31
32                  MAX[q] = new_fft_array
33                  MIN[q] = new_fft_array
34                  SUM[q] = new_fft_array
35                  CNT[q] = 1
36              else:
37                  SUM[q] += new_fft_array
38                  CNT[q] += 1
39
40      for q in range(fft_array_length):
41          novelty += 1
42          MC.append([q + 1, num_frames, SUM[q], CNT[q]])
43
44      MC = np.array(MC).T
45      print(f'> {novelty} novelty were detected.\nsize(MC) = {MC.shape}')
46
47      return MC, gamma
```

B CÓDIGO-FONTE DA DESCOMPRESSÃO

B.1 Implementação no MATLAB

Listing B.1 – Função Auxiliar (decompress_signal_matlab.m).

```
1
2  function [reconstructed_signal, FFT_ARRAY, MR, Nppc, num_frames] =
       decompress_signal_matlab(full_path_mc_file, Fs_orig, using_symmetry)
3      file_mc = fopen(full_path_mc_file, 'rb');
4      if file_mc == -1
5          error('Error when trying to open the file %s. Execution is stopped.', full_path_mc_file);
6      else
7          fprintf('%s file opened successfully\n', full_path_mc_file);
8      end
9
10     % Read the preamble
11     length_mc_q = fread(file_mc, 1, 'uint32');
12     length_mc_f = fread(file_mc, 1, 'uint32');
13     length_mc_sum = fread(file_mc, 1, 'uint32');
14     length_mc_cnt = fread(file_mc, 1, 'uint32');
15     Nppc = fread(file_mc, 1, 'uint32');
16     num_frames = fread(file_mc, 1, 'uint32');
17
18     fprintf('Preambulo: [q = %d, f = %d, sum = %d, cnt = %d, Nppc = %d, Nf = %d]\n',
           length_mc_q, length_mc_f, length_mc_sum, length_mc_cnt, Nppc, num_frames);
19
20 %% Compression matrix recovery (MC)
21     mc_q = fread(file_mc, length_mc_q, 'uint16'); % uint8 for embedded implementation (
           decompress_signal_esp32.m)
22     mc_f = fread(file_mc, length_mc_f, 'uint32');
23     mc_sum = fread(file_mc, length_mc_sum, 'double');
24     mc_cnt = fread(file_mc, length_mc_cnt, 'uint32');
25     fclose(file_mc);
26
27 %% Creation of the Reconstruction Matrix (MR)
28     % Decoding data stored in the MC compression matrix
29     MR = []; % zeros(num_frames, 2*Nppc);
30     for novelty = 1:length(mc_q)
31                 % In the embedded implementation (decompress_signal_esp32.m) we have +1
                       to the value of q and f_b (vectors start at index 0)
32         q = mc_q(novelty);  % position where the novelty was detected
```

```
33          f_b = mc_f(novelty);  % frame of the last occurrence
34          sum_q = mc_sum(novelty); % cumulative sum
35          cnt_q = mc_cnt(novelty); % number of repetitions of this novelty
36
37          % Assigning the estimated value based on the accumulated sum and number of
                repetitions
38          f_a = f_b - cnt_q+1;
39
40                  % In the embedded implementation (decompress_signal_esp32.m):
41                  %if (f_b>= num_frames)
42          % f_a = f_b - cnt_q;
43          %else
44          % f_a = f_b - cnt_q+1;
45          end
46
47          for frame = f_a:f_b
48              MR(frame, q) = sum_q/cnt_q;
49          end
50      end
51      fprintf('size(MR) = [%d, %d]\n', size(MR));
52
53  %% Signal Reconstruction
54       switch using_symmetry
55          case 1 % with the use of symmetry
56              %Concatenating the real and imaginary parts
57              length_frame = ((Nppc/2)+1)*2;
58              FFT_ARRAY_TMP_1 = complex(MR(:, 1:(length_frame/2)), MR(:, (length_frame
                    /2)+1:end));
59
60              % Reconstructing the DFT of a signal based on symmetry
61              FFT_ARRAY_TMP_2 = FFT_ARRAY_TMP_1(:, 1:end);
62              [~ , c] = size(FFT_ARRAY_TMP_2); % numero de colunas
63              ind_invertido = c-1:-1:2;
64              FFT_ARRAY_TMP_2_INV = FFT_ARRAY_TMP_2(:, ind_invertido); % mirroring
65              FFT_ARRAY = [FFT_ARRAY_TMP_1, conj(FFT_ARRAY_TMP_2_INV)];
66
67              % Applying IFFT to recover each frame
68              IFFT_ARRAY_AUX = ifft(FFT_ARRAY,[],2);
69          case 2 % without the use of symmetry
70              length_frame = Nppc*2;
71              FFT_ARRAY = complex(MR(:, 1:(length_frame/2)), MR(:, (length_frame/2)+1:end
                    ));
```

```
72          otherwise
73              error('Unexpected value: using_symmetry = %d. Execution is stopped.',
                    using_symmetry);
74      end
75
76      fprintf('size(FFT_ARRAY): [%d, %d]\n', size(FFT_ARRAY));
77      fprintf('size(IFFT_ARRAY_AUX): [%d, %d]\n', size(IFFT_ARRAY_AUX));
78
79      % Concatenating the frames to recover the signal
80      reconstructed_signal = [];
81      for f = 1:num_frames
82          reconstructed_signal = [reconstructed_signal real(IFFT_ARRAY_AUX(f, :))];
83      end
84
85      % Resampling to return to the original number of samples
86      f1 = 60;
87      Fs_current = Nppc*f1;
88      if (Fs_orig ~= Fs_current)
89          fprintf('Fs_current = %d\n',Fs_current);
90          reconstructed_signal = resample(reconstructed_signal, round(Fs_orig), round(Fs_current)
                );
91      end
92  end
```

B.2 Implementação em Python

Listing B.2 – Função Auxiliar (decompress_signal.py).

```
1   import sys
2   import os
3   import tempfile
4   # Redirect stderr to a temporary file to capture errors
5   stderr_file = tempfile.NamedTemporaryFile(delete=False)
6   sys.stderr = open(stderr_file.name, 'w')
7
8   # Configure Matplotlib backend
9   import matplotlib
10  matplotlib.use('TkAgg')
11
12  import numpy as np
13  import matplotlib.pyplot as plt
14  from scipy.fft import ifft
```

```
15 from scipy.signal import resample
16 import struct
17
18 def decompress_signal_local(full_path_mc_file, Fs_orig, using_symmetry):
19     with open(full_path_mc_file, 'rb') as file:
20         # Read the preamble
21         length_mc_q = struct.unpack('I', file.read(4))[0]
22         length_mc_f = struct.unpack('I', file.read(4))[0]
23         length_mc_sum = struct.unpack('I', file.read(4))[0]
24         length_mc_cnt = struct.unpack('I', file.read(4))[0]
25         Nppc = struct.unpack('I', file.read(4))[0]
26         num_frames = struct.unpack('I', file.read(4))[0]
27
28         # Read the compression matrix
29         mc_q = np.fromfile(file, dtype=np.uint16, count=length_mc_q)
30         mc_f = np.fromfile(file, dtype=np.uint32, count=length_mc_f)
31         mc_sum = np.fromfile(file, dtype=np.float64, count=length_mc_sum)
32         mc_cnt = np.fromfile(file, dtype=np.uint32, count=length_mc_cnt)
33     print(length_mc_q == len(mc_q))
34     print(f"length_mc_q: {length_mc_q}, length_mc_f: {length_mc_f}, length_mc_sum: {
           length_mc_sum}, length_mc_cnt: {length_mc_cnt}")
35     print(f"mc_q[:10]: {mc_q[:10]}")
36     print(f"mc_f[:10]: {mc_f[:10]}")
37     print(f"mc_sum[:10]: {mc_sum[:10]}")
38     print(f"mc_cnt[:10]: {mc_cnt[:10]}")
39
40     # Creation of the Reconstruction Matrix (RM)
41     if using_symmetry == 1:
42         MR = np.zeros((num_frames, Nppc+2))
43     elif using_symmetry == 2:
44         MR = np.zeros((num_frames, Nppc*2))
45     else:
46         raise ValueError(f'Unexpected value: using_symmetry = {using_symmetry}. Execution is
               stopped.')
47
48     for novelty in range(len(mc_q)):
49         q = mc_q[novelty]-1
50         f_b = mc_f[novelty] -1
51         sum_q = mc_sum[novelty]
52         cnt_q = mc_cnt[novelty]
53
54         f_a = f_b - cnt_q+1
```

```
55          if q == 1 and f_b%2 !=0 :
56              print(f'#{novelty}: q={q}, f={f_b}:{f_a}, sum={sum_q}, cnt={cnt_q}\n')
57
58          MR[f_a:f_b+1, q] = sum_q / cnt_q
59
60      print(f'size(MR) = {MR.shape}')
61
62
63      # Signal reconstruction
64      if using_symmetry == 1:
65          length_frame = ((Nppc // 2) + 1) * 2
66          FFT_ARRAY_TMP_1 = np.zeros((num_frames, length_frame // 2), dtype=np.
                complex128)
67          FFT_ARRAY_TMP_1.real = MR[:, :length_frame // 2]
68          FFT_ARRAY_TMP_1.imag = MR[:, length_frame // 2:2 * (length_frame // 2)]
69
70          FFT_ARRAY_TMP_2 = np.conjugate(np.flip(FFT_ARRAY_TMP_1[:, 1:Nppc // 2],
                axis=1))
71          FFT_ARRAY = np.concatenate((FFT_ARRAY_TMP_1, FFT_ARRAY_TMP_2), axis
                =1)
72      elif using_symmetry == 2:
73          FFT_ARRAY = MR[:, :Nppc] + 1j * MR[:, Nppc:]
74      else:
75          raise ValueError(f'Unexpected value: using_symmetry = {using_symmetry}. Execution is
                stopped.')
76
77      print(f'size(FFT_ARRAY) = {FFT_ARRAY.shape}')
78
79
80      reconstructed_frames = ifft(FFT_ARRAY, axis=1).real
81      reconstructed_signal = reconstructed_frames.flatten()
82
83      if Fs_orig != Nppc * 60:
84          reconstructed_signal = resample(reconstructed_signal, int(len(reconstructed_signal) *
                Fs_orig / (Nppc * 60)))
85
86      # Preview
87      plt.plot(reconstructed_signal)
88      plt.title('Sinal Reconstruido')
89      plt.xlabel('Sample')
90      plt.ylabel('Amplitude')
91      plt.grid(True)
```

```
92      plt.show()
93
94      return reconstructed_signal, FFT_ARRAY, MR, Nppc, num_frames
95
96  def decompress_signal_embedded(full_path_mc_file, using_symmetry):
97      with open(full_path_mc_file, 'rb') as file:
98          length_mc_q = struct.unpack('I', file.read(4))[0]
99          length_mc_f = struct.unpack('I', file.read(4))[0]
100         length_mc_sum = struct.unpack('I', file.read(4))[0]
101         length_mc_cnt = struct.unpack('I', file.read(4))[0]
102         Nppc = struct.unpack('I', file.read(4))[0]
103         num_frames = struct.unpack('I', file.read(4))[0]
104
105         mc_q = np.fromfile(file, dtype=np.uint8, count=length_mc_q)
106         mc_f = np.fromfile(file, dtype=np.uint32, count=length_mc_f)
107         mc_sum = np.fromfile(file, dtype=np.float64, count=length_mc_sum)
108         mc_cnt = np.fromfile(file, dtype=np.uint32, count=length_mc_cnt)
109
110     # Check read data
111     print(f"length_mc_q: {length_mc_q}, length_mc_f: {length_mc_f}, length_mc_sum: {
            length_mc_sum}, length_mc_cnt: {length_mc_cnt}")
112     print(f"mc_q[:10]: {mc_q[:10]}")
113     print(f"mc_f[:10]: {mc_f[:10]}")
114     print(f"mc_sum[:10]: {mc_sum[:10]}")
115     print(f"mc_cnt[:10]: {mc_cnt[:10]}")
116
117     # Creation of the Reconstruction Matrix (RM)
118     MR = np.zeros((num_frames, 2 * Nppc))
119     for novelty in range(len(mc_q)):
120         q = mc_q[novelty] + 1
121         f_b = mc_f[novelty] + 1
122         sum_q = mc_sum[novelty]
123         cnt_q = mc_cnt[novelty]
124
125         f_a = max(0, int(f_b - cnt_q + 1))
126         if f_b >= num_frames:
127             f_b = num_frames - 1
128
129         for frame in range(f_a, f_b + 1):
130             MR[frame, q] = sum_q / cnt_q
131
132     print(f'size(MR) = {MR.shape}')
```

```
133     print(f'MR[:10, :10]: {MR[:10, :10]}')
134
135     # Signal reconstruction
136     if using_symmetry == 1:
137         length_frame = ((Nppc // 2) + 1) * 2
138         FFT_ARRAY_TMP_1 = np.zeros((num_frames, length_frame // 2), dtype=np.
                complex128)
139         FFT_ARRAY_TMP_1.real = MR[:, :length_frame // 2]
140         FFT_ARRAY_TMP_1.imag = MR[:, length_frame // 2:2 * (length_frame // 2)]
141
142         FFT_ARRAY_TMP_2 = FFT_ARRAY_TMP_1[:, 1:-1]
143         FFT_ARRAY_TMP_2_INV = FFT_ARRAY_TMP_2[:, ::-1].conj()
144         FFT_ARRAY = np.concatenate((FFT_ARRAY_TMP_1, FFT_ARRAY_TMP_2_INV),
                axis=1)
145     elif using_symmetry == 2:
146         length_frame = Nppc * 2
147         FFT_ARRAY = np.zeros((num_frames, Nppc), dtype=np.complex128)
148         FFT_ARRAY.real = MR[:, :Nppc]
149         FFT_ARRAY.imag = MR[:, Nppc:]
150     else:
151         raise ValueError(f'Unexpected value: using_symmetry = {using_symmetry}. Execution is
                stopped.')
152
153     IFFT_ARRAY_AUX = ifft(FFT_ARRAY, axis=1)
154
155     print(f'size(FFT_ARRAY): {FFT_ARRAY.shape}\nsize(IFFT_ARRAY_AUX): {
        IFFT_ARRAY_AUX.shape}')
156
157     reconstructed_signal = np.concatenate([frame.real for frame in IFFT_ARRAY_AUX])
158
159     return reconstructed_signal, FFT_ARRAY, MR, Nppc, num_frames
160
161 def save_reconstructed_signal(signal, original_file_name, Fs_current, path):
162     reconstructed_file_name = os.path.basename(original_file_name)[:-4] + '_r.bin'
163     full_path_reconstructed_file = os.path.join(path, reconstructed_file_name)
164
165     preambulo = ['RECO', Fs_current, len(signal)]
166     volt_to_dig = 4095/3.3
167     signal = np.uint16(signal*volt_to_dig)
168
169     with open(full_path_reconstructed_file, 'wb') as file:
170         file.write(struct.pack('4s f I', preambulo[0].encode(), preambulo[1], preambulo[2]))
```

```
171             file.write(signal.astype(np.uint16).tobytes())
172
173     print(f'Reconstructed signal saved in: {full_path_reconstructed_file}')
174     return full_path_reconstructed_file
```