THALES ROGER ALVES DE PAULA

SIMULAÇÃO E IMPLEMENTAÇÃO EM DSP DE ALGORITMOS AUXILIARES PARA CÁLCULO DE POTÊNCIA EM CONVERSORES COM MODELO DROOP

LAVRAS – MG

2021

THALES ROGER ALVES DE PAULA

SIMULAÇÃO E IMPLEMENTAÇÃO EM DSP DE ALGORITMOS AUXILIARES PARA CÁLCULO DE POTÊNCIA EM CONVERSORES COM MODELO DROOP

Trabalho de Conclusão de Curso apresentado à Escola de Engenharia da Universidade Federal de Lavras, como parte das exigências para graduação nível Bacharelado em Engenharia de Controle e Automação.

Dra. Sívia Costa Ferreira
Orientadora

LAVRAS – MG

2021

Ficha catalográfica elaborada pelo Sistema de Geração de Ficha Catalográfica da Biblioteca Universitária da UFLA, com dados informados pelo(a) próprio(a) autor(a).

Paula, Thales Roger Alves de.
SIMULAÇÃO E IMPLEMENTAÇÃO EM DSP DE ALGORITMOS AUXILIARES PARA CÁLCULO DE POTÊNCIA EM CONVERSORES COM MODELO DROOP / Thales Roger Alves de Paula. – Lavras : UFLA, 2021.
100 p. : il.

TCC(graduação)–Universidade Federal de Lavras, 2021.
Orientadora: Dra. Sívia Costa Ferreira.
Bibliografia.

1. Cálculo de potências. 2. Referência Síncrona. 3. Filtro Adaptativo Sintonizado. I. Ferreira, Sílvia Costa. II. Título.

THALES ROGER ALVES DE PAULA

SIMULAÇÃO E IMPLEMENTAÇÃO EM DSP DE ALGORITMOS AUXILIARES PARA CÁLCULO DE POTÊNCIA EM CONVERSORES COM MODELO DROOP

Trabalho de Conclusão de Curso apresentado à Escola de Engenharia da Universidade Federal de Lavras, como parte das exigências para graduação nível Bacharelado em Engenharia de Controle e Automação.

APROVADA em 10 de junho de 2021.

Dra. Sívia Costa Ferreira	UFLA
Dr. Vinícius Miranda Pacheco	UFLA
Eng. João Paulo C. Pedroso	UFLA

Dra. Sívia Costa Ferreira
Orientadora

LAVRAS – MG
2021

AGRADECIMENTOS

Agradeço primeiramente a Deus, que em sua bondade e misericórdia guiou cada um dos meus passos, dando sabedoria e discernimento, e me permitindo alcançar mais essa vitória.

Aos meus pais, Bertolino e Sandra, que me forneceram todo o suporte, apoio e incentivo essenciais durante essa jornada. À minha namorada Camila, que sempre esteve ao meu lado nas lutas e nas conquistas.

Ao meu colega de casa e amigo Nicolás. A cada familiar, amigos da igreja, da universidade e da vida, que me acompanharam, ajudaram e fizeram parte dessa história.

À Igreja Adventista do Sétimo Dia que me apoiou. À Equipe TROIA de Robótica por cada ensinamento e cada experiência.

À Universidade Federal de Lavras (UFLA) e ao Governo Brasileiro pela oportunidade.

Todas as coisas foram feitas por intermédio dEle, e, sem Ele, nada do que foi feito se fez.
(João 1:3)

RESUMO

As tecnologias aplicadas no gerenciamento de redes elétricas estão em constante evolução. Em locais onde existe a utilização de uma ou mais fontes de energia independentes da fonte geradora comum, existe a necessidade de realizar um bom gerenciamento da potência fornecida por cada uma delas. Para tanto, faz-se necessário realizar um monitoramento preciso e em tempo real da potência demandada pelos aparelhos conectados, o que pode ser feito por diferentes técnicas. Este trabalho tem por objetivo aplicar duas técnicas para estimação de potência utilizando microcontrolador DSP e compará-las. A primeira se baseia na Referência Síncrona, convertendo os sinais de tensão e corrente em componentes dq, a partir das quais é possível calcular a potência. A segunda técnica é baseada em um Filtro Adaptativo sintonizado, que extrai a componente fundamental dos sinais de tensão e corrente e calcula as parcelas ativa e reativa da corrente, sendo possível então calcular a potência. Os códigos foram implementados em linguagem C e seu comportamento foi simulado através do ambiente Simulink do software Matlab. O algoritmos foram implementados no DSP TMS320F28379D, da Texas Instruments, e ambos apresentaram resultados satisfatórios, sendo capazes de apresentar um valor de potência estimada com boa precisão em curto período de tempo.

Palavras-chave: Cálculo de potência, Alternada, DSP, Referência Síncrona, Filtro Adaptativo Sintonizado

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

Uma microrrede pode ser definida como um sistema que engloba unidades de geração distribuída, cargas e sistemas de armazenamento que devem entregar energia confiável e de qualidade. Independente de a microrrede estar operando em condição ilhada ou não, isto é, desconectada ou conectada à rede, é necessário que o gerenciamento de potência fornecida por cada fonte seja feito de forma confiável e precisa.

Um dos métodos para realizar essa função é o Modelo Droop, que atualmente é um dos mais utilizados. Este método se trata de um sistema de gerenciamento distribuído em que cada conversor do tipo fonte de tensão determina a potência a ser fornecida baseado na frequência da rede, a qual varia de acordo com a quantidade de potência demandada. Para que esse modelo seja aplicado é necessário uma medição precisa e veloz da potência demandada pela rede.

A leitura de potência elétrica demandada, quando se trata de uma rede elétrica alternada, não pode ser feita de forma trivial como no caso contínuo, onde basta se multiplicar a tensão pela corrente a cada instante. No caso alternado o conceito se mantém, ou seja, a potência é obtida por um produto da tensão pela corrente, mas são necessárias adaptações, principalmente para situações em que há variações na frequência.

Dado que os sinais oscilam na forma de uma senoide, e possuem valor médio nulo, é necessário que se utilize métodos de obtenção da amplitude dessas ondas. Uma vez obtidas as amplitudes, o cálculo de potência se torna quase tão simples quanto em circuitos de corrente contínua, pois são sinais contínuos. Dois desses métodos são a Referência Síncrona e o Filtro Adaptativo Sintonizado, os quais utilizam um algoritmo de PLL (*Phase Locked Loop*) para detectar a fase e a frequência da rede.

A técnica da Referência Síncrona consiste em utilizar um algoritmo PLL para identificar a frequência e a fase da rede e utilizar essas informações para aplicar a Transformada de Park adaptada para sinais monofásicos, convertendo os sinais de tensão e corrente em suas respectivas componentes dq. A partir dessas componentes é possível calcular a potência.

A segunda técnica é baseada em um Filtro Adaptativo Sintonizado. O filtro possui um estimador de frequência, que faz com que ele seja sintonizado com a frequência do sinal de entrada independente das variações de frequência. O algoritmo de filtragem calcula duas componentes ortogonais para o sinal de entrada, sendo possível então aplicá-lo nos sinais de tensão e corrente, e os resultados podem ser utilizados para calcular a potência de forma análoga à *teoria dq*.

Ambas as técnicas tem sido amplamente estudadas e aplicadas. Em Silva (2019) foi realizado um estudo comparativo entre quatro algoritmos de PLL, dentre os quais se encontram um baseado em Referência Síncrona e outro em Filtros Adaptativos Sintonizados. Em Ferreira (2012) Filtros Adaptativos também foram utilizados em sistemas compensadores de reativos, e em Ferreira et al. (2015) para estimação de potência reativa. Aplicações com algoritmos mais avançados foram realizadas em Rauth, Kumar e Srinivas (2018) e Yin, Guo e Li (2013), onde foram utilizadas variações do algoritmo, implementando um parâmetro de ajuste (γ) adaptativo de acordo com o sinal de entrada.

Tendo em vista a ampla aplicação das técnicas de estimação de potência em redes elétricas alternadas, o presente trabalho visa implementar os métodos de estimação por Referência Síncrona e por Filtro Adaptativo Sintonizado e comparar seus desempenhos através de simulações e testes, para que posteriormente o melhor método seja utilizado no gerenciamento de microrredes.

1.1 Objetivo geral

O objetivo principal deste trabalho é aplicar, simular, testar e avaliar técnicas para cálculo de potência demandada da rede elétrica de corrente alternada, utilizando o software Matlab/Simulink e o DPS TMS320F28379D. As técnicas estudadas são Referência Síncrona e Filtro Adaptativo com Estimador de Frequência.

1.2 Objetivos específicos

Como objetivos específicos deste trabalho, podem ser listados:

• Simulação em ambiente Matlab/Simulink, utilizando o bloco S-function para implementar, em linguagem C, dos algoritmos de cálculo de potência;

• Configuração do DSP TMS320F28379D para implementação dos códigos previamente desenvolvidos no Matlab/Simulink, bem como os ajustes necessários em seus conversores analógicos-digitais (ADC) para leitura em tempo real dos sinais de tensão e corrente da rede elétrica;

• Testes práticos do sistema para análise de desempenho em aplicações reais.

2 REFERENCIAL TEÓRICO

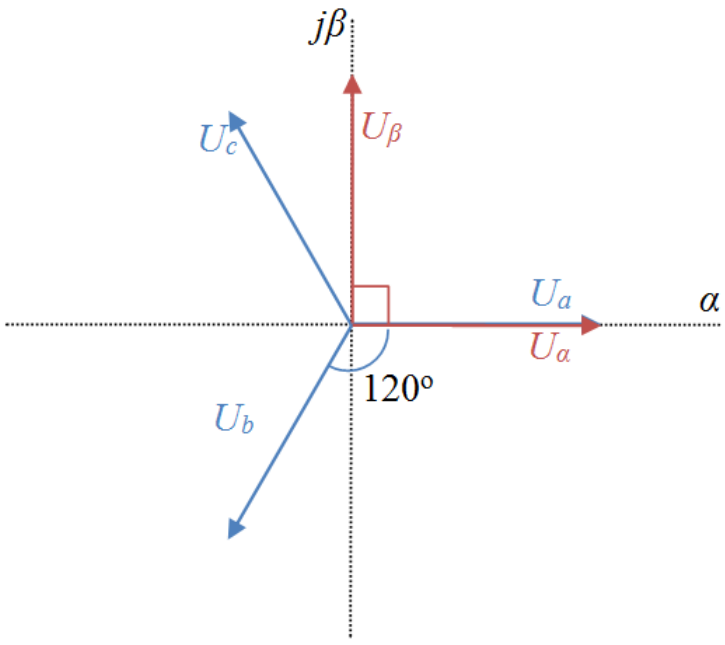
2.1 Referência Síncrona

A Referência Síncrona foi inicialmente desenvolvida para sistemas trifásicos, com o objetivo de ser utilizada no controle de filtros ativos de potência. Tendo sido apresentada por Peng, Akagi e Nabae (1988), a técnica passou a ser aplicada em diversos trabalhos e pesquisas, sendo muito utilizada pra controlar conversores conectados à rede (ASIMINOAEL; BLAABJERG; HANSEN, 2007).

A técnica consiste na aplicação de duas transformadas aos sinais da rede elétrica. Primeiramente aplica-se a Transformada de Clarke, que converte os sinais a, b e c em sinais chamados de α e β. Através dessa transformação, o sinal inicialmente representado por três fasores passa a ser representado por apenas dois, que correspondem ao somatório das suas projeções em um sistema de coordenadas estacionário.

A transformação, portanto, tem como saída duas componentes perpendiculares, e terá também uma componente de sequência zero chamada x_0, caso o sistema seja desequilibrado e a quatro fios. A Figura 2.1 apresenta um diagrama fasorial correspondente à transformação $abc - \alpha\beta$, onde U_a, U_b e U_c representam a tensão nas fases A, B e C, e U_α e U_β representam os sinais de saída da Transformada de Clarke.

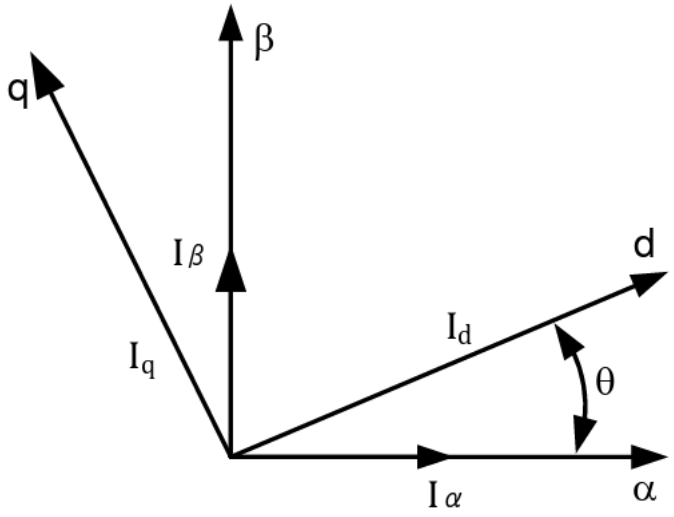
Figura 2.1 – Diagrama fasorial da Transformada de Clarke.



Fonte: Dobrucký et al. (2016)

Após essa transformação aplica-se a transformada de Park. Ela converte os sinais $\alpha\beta$ em dq, e consiste na projeção dos sinais de entrada em um sistema de coordenadas que gira com a mesma frequência do sinal de entrada, tornando-se estacionário em relação aos fasores de entrada. A Figura 2.2 apresenta um diagrama fasorial correspondente à transformação $\alpha\beta - dq$, onde θ corresponde à fase do sinal de entrada, α e β representam os eixos do sistema de coordenadas estacionário, I_α e I_β representam os sinais de saída da Transformada de Clarke, d e q representam os eixos do sistema de coordenadas rotacional, e I_d e I_q representam os sinais de saída da Transformada de Park.

Figura 2.2 – Diagrama fasorial da Transformada de Park.



Fonte: Keysan (2017).

2.1.1 Cálculo de Potências em dq

Após a transformação síncrona os sinais de tensão e corrente são convertidos em valores contínuos. Ao se aplicar ambas as transformações nos sinais de tensão e corrente, torna-se viável o cálculo de potência de forma mais simples. Foi demonstrado por Peng, Akagi e Nabae (1988) que os valores de potência instantânea podem ser calculados da seguinte forma, onde P_{dq}, Q_{dq} e S_{dq} representam respectivamente os valores de potência ativa, reativa e aparente calculados utilizando os valores de eixo direto e quadratura da tensão e da corrente, v_d e v_q representam os valores de eixo direto e quadratura da tensão, i_d e i_q representam os valores de eixo direto e quadratura da corrente, V_{rms} e I_{rms} representam os valores eficazes da tensão e da corrente e ϕ representa a defasagem entre os sinais de tensão e corrente.

$$P_{dq} = (3/2)\left(v_d i_d + v_q i_q\right) = 3V_{rms}I_{rms}cos(\phi) \tag{2.1}$$

$$Q_{dq} = (3/2)\left(v_d i_q - v_q i_d\right) = 3V_{rms}I_{rms}sen(\phi) \tag{2.2}$$

$$S_{dq} = \sqrt{P_{dq}^2 + Q_{dq}^2} \tag{2.3}$$

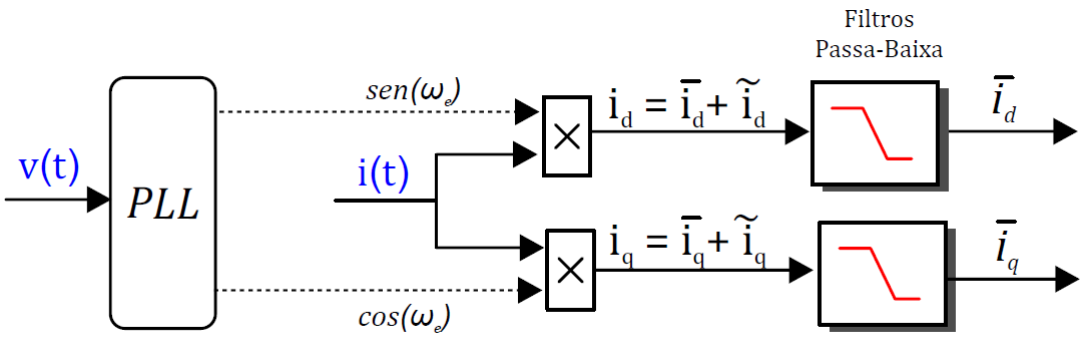
2.1.2 Referência Síncrona Monofásica

Para sistemas monofásicos as equações de transformação são diferentes dos sistemas trifásicos. Nesse caso, a utilização da Referência Síncrona se dá pela multiplicação da entrada pelos sinais ortogonais $sen(\omega t)$ e $cos(\omega t)$. Os valores obtidos devem então ser filtrados para remoção de uma oscilação resultante da operação, com frequência igual ao dobro da frequência do sinal de entrada, e para remoção das componentes harmônicas dos sinais, de forma que os sinais resultantes representem apenas a frequência fundamental.

A Figura 2.3 apresenta um diagrama de blocos para cálculo das componentes dq da corrente monofásica. Neste diagrama, o bloco PLL representa um algoritmo de Malha de Controle de Fase, que recebe um sinal senoidal como entrada e fornece na saída a informação de sua fase instantânea, a qual é utilizada para calcular os valores de seno e cosseno que são multiplicados pelo sinal que se deseja analisar, no caso a corrente. Esse algoritmo será discutido de forma mais aprofundada posteriormente.

O resultado desse produto são os sinais i_d e i_q, cujos valores médios ($\bar{i}_d$ e $\bar{i}_q$) são proporcionais à amplitude das parcelas de corrente ativa e reativa. O filtro passa-baixas é então utilizado para retirar a parcela oscilante do sinal ($\tilde{i}_d$ e $\tilde{i}_q$), de forma que o sinal resultante apresente apenas o valor médio.

Figura 2.3 – Diagrama de blocos do cálculo de componentes dq para corrente alternada monofásica.



Fonte: Ferreira (2016)

Para este caso Silva et al. (2010) demonstra que as componentes contínuas dos sinais obtidos a partir desse cálculo são dadas pelas seguintes equações:

$$\bar{i}_d = \frac{I_1}{2}\cos(\theta_1) \tag{2.4}$$

$$\bar{i}_q = \frac{I_1}{2}\,\text{sen}(\theta_1) \tag{2.5}$$

onde, $I1$ é o valor de pico da corrente fundamental e θ_1 é o ângulo de defasagem entre tensão e o sinal analizado. Dessa forma, basta que o sinal obtido seja multiplicado por 2 para obter o valor correspondente à amplitude total da onda de corrente.

Por se tratar de uma única fase, as equações de potência sofrem uma pequena alteração, onde o ganho de $\frac{3}{2}$ passa a ter o valor de $\frac{1}{2}$.

$$P_{dq} = (1/2)\left(\bar{v}_d\,\bar{i}_d + \bar{v}_q\,\bar{i}_q\right) = V_{rms}I_{rms}cos(\phi) \tag{2.6}$$

$$Q_{dq} = (1/2)\left(\bar{v}_d\,\bar{i}_q - \bar{v}_q\,\bar{i}_d\right) = V_{rms}I_{rms}sen(\phi) \tag{2.7}$$

$$S_{dq} = \sqrt{P_{dq}^2 + Q_{dq}^2} \tag{2.8}$$

2.2 Filtro Adaptativo Sintonizado com Estimador de Frequência

Além da Referência Síncrona, existem também outras técnicas na literatura com o objetivo de calcular a potência instantânea de forma a auxiliar o controle de conversores eletrônicos. Entre elas se destaca o algoritmo baseado em um Filtro Adaptativo Sintonizado, apresentado em diversos trabalhos, tais como Yazdani, Bakhshai e Jain (2010) e Ferreira et al. (2015).

Um Filtro Adaptativo Sintonizado com Estimador de Frequência (FAS-estimador) se trata de um filtro *notch* do tipo IIR, que se diferencia de um filtro convencional por possuir um algoritmo que ajusta dinamicamente os seus parâmetros, permitindo que ele se adapte ao sinal de entrada.

Essa característica é essencial para a aplicação em estimação de potência em microrredes isoladas gerenciadas por Modelo Droop, pois esse modelo utiliza a frequência de oscilação da

rede como meio de comunicação. Nele, a frequência é controlada de forma a comunicar aos conversores a potência que está sendo demandada, permitindo o controle da quantidade de potência que cada fonte proverá e evitando o desbalanço entre a energia gerada e a consumida pelas cargas.

A utilização de um filtro convencional, cuja frequência sintonizada seja fixa, nesse cenário certamente levará ao acumulo de erros. Por esse motivo também, o filtro utiliza um algoritmo para estimar a frquência do sinal de entrada, utilizando essa informação em seu algoritmo de adaptação.

O estimador de frequência em questão foi inicialmente desenvolvido para ser utilizado em um Filtro Adaptativo, proposto no domínio do tempo por *Bodson e Douglas* (BODSON; DOUGLAS, 1996), e modificado por *Hsu et al.* (HSU; ORTEGA; DAMM, 1999). O algoritmo foi posteriormente estendido para um arranjo capaz de extrair individualmente as componentes senoidais de um sinal e estimar a componente fundamental e os harmônicos.

O filtro é definido pelo seguinte conjunto de equações:

$$\dot{\omega}(t) = -\gamma\, \omega(t)\, x_1(t)\, e(t) \tag{2.9}$$

$$\ddot{x}_1(t) = 2\, \zeta \omega(t)\, e(t) - i^2 \omega^2(t)\, x_1(t) \tag{2.10}$$

$$e(t) = d(t) - \dot{x}_1(t) \tag{2.11}$$

em que $d(t)$ é o sinal observado, $x_1(t)$ é saída do filtro, $\omega_1(t)$ é a freqûencia fundamental estimada, $e(t)$ é o erro e i a ordem harmônica de interesse; γ e ζ são parâmetros que determinam a dinâmica do estimador, sua velocidade e precisão, eles correspondem ao coefciente de adaptação e ao fator de amortecimento, respectivamente.

O conjunto de equações que define esse algoritmo pode ser dividido em duas variáveis de estado (x_i e $\dot{x}_i$), e reescrito:

$$\omega(t) = -\int \gamma\, x_1(t)\; \omega(t) e(t)\; \cdot dt \tag{2.12}$$

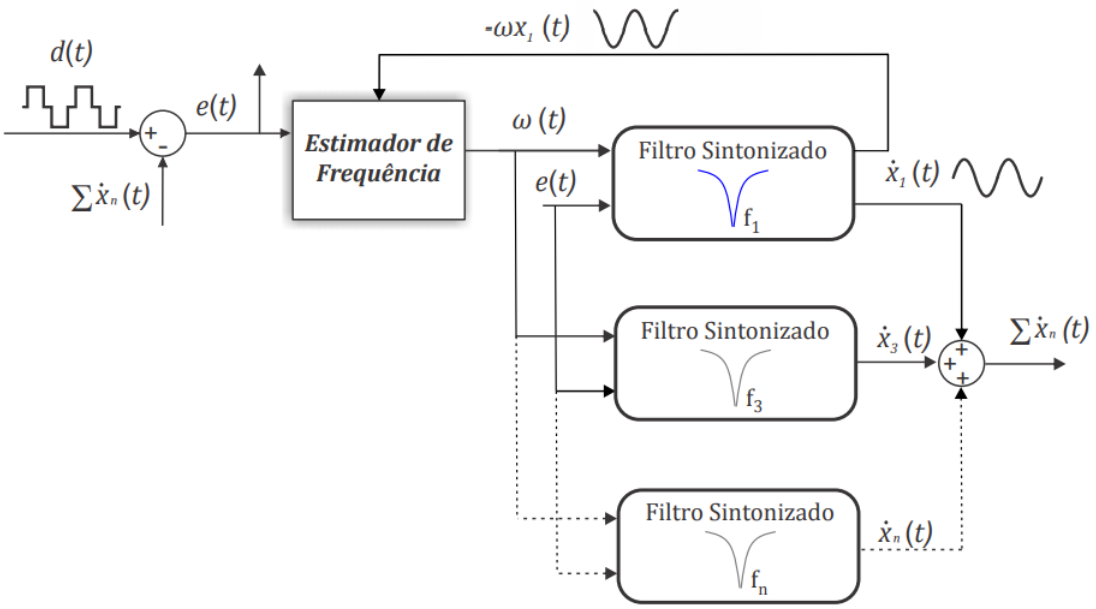
$$\dot{x}_i(t) = \int 2\; \zeta_i\; \omega(t)\; e(t)\; -\; i^2 \omega^2(t)\; x_i(t) \cdot dt \tag{2.13}$$

$$x_i(t) = \int \dot{x}_i(t) \cdot dt\; \rightarrow\; i = 1,3,5,...,n \tag{2.14}$$

$$e(t) = d(t) - \sum_{i=0}^{n} \dot{x}_i(t). \tag{2.15}$$

A Figura 2.4 apresenta um diagrama de blocos representando o funcionamento desse filtro de forma genérica.

Figura 2.4 – Diagrama do Filtro Adaptativo Sintonizado com Estimador de Frequência.



Fonte: Mojiri e Bakhshai (2004)

A utilização de subfiltros para os harmônicos pode ser útil mesmo houver interesse apenas na componente fundamental, pois melhora a precisão do estimador de frequência.

2.2.1 Cálculo de potência com vetores ortogonais

No contexto da estimação de potência, o Filtro Adaptativo Sintonizado (FAS-estimador) descrito anteriormente pode ser utilizado pois suas variáveis de estado x_i e $\dot{x}_i$ representam dois sinais ortogonais cujas amplitudes se relacionam à frequência fundamental do sinal de entrada da seguinte forma (FERREIRA, 2012):

$$d_1 = \dot{x}_{1_d} = D_1 sen(\omega t + \theta_1) \tag{2.16}$$

$$d_{1_{90^\circ}} = -\omega x_{1_d} = D_1 cos(\omega t + \theta_1) \tag{2.17}$$

Onde d_1 e $d_{1_{90^\circ}}$ são as duas componentes ortogonais, que podem representar tensão (v_1 e $v_{1_{90^\circ}}$) e corrente (i_1 e $i_{1_{90^\circ}}$) dependendo do sinal de entrada do filtro. $D1$ representa a amplitude do sinal de entrada, x_{i_d} e $\dot{x}_{i_d}$ são as variáveis de estado do filtro, t representa o tempo, ω a frequência do sinal e θ a defasagem do sinal de entrada.

Assim sendo, pode-se aplicar este filtro duas vezes, tendo como entradas os sinais de tensão e corrente a serem analisados. Uma vez obtidos os quatro sinais, sendo dois pares de sinais ortogonais, a potência pode ser cálculada de forma análoga ao método *dq*.

$$P_1 = \left(v_1\, i_1 + v_{1_{90^\circ}}\, i_{1_{90^\circ}}\right) \tag{2.18}$$

$$Q_1 = \left(v_1\, i_{1_{90^\circ}} - v_{1_{90^\circ}}\, i_1\right) \tag{2.19}$$

É possível observar que nesse método não se faz necessária a utilização do valor de defasagem entre a tensão e a corrente, o que elimina a necessidade de implementação de um algoritmo PLL. No entanto, para suprir essa informação foi necessário implementar o FAS-estimador uma vez para cada sinal.

2.2.2 Malha de controle de Fase baseada em Filtro Adaptativo (PLL - Phase Locked Loop)

A estimação de frequência e fase da rede elétrica é uma etapa essencial para a obtenção de um bom resultado do cálculo de potência por Referência Síncrona, e pode ser realizada através de um agoritmo de PLL (*Phase Locked Loop*). Essa malha de controle pode ser implementada através de um Filtro Adaptativo Sintonizado com Estimador de Frequência (FAS-estimador), que foi apresentado anteriormente.

Uma vez estimada a frequência por meio desse algoritmo, a fase *wt* pode ser determinada de acordo com a Equação 2.20.

$$\omega t = \arctan\left(\frac{\dot{x}_1}{\omega x_1}\right) = \arctan\left(\frac{v_1}{v_{1_{90^\circ}}}\right) \tag{2.20}$$

Dessa forma, esse valor calculado de fase pode ser utilizado para obter os sinais ortogonais proporcionais à amplitude do sinal de entrada. O PLL apresenta melhores resultados quando aplicado ao sinal de tensão, pois este apresenta maior estabilidade que o sinal de corrente.

2.3 Processador Digital de Sinais (DSP - Digital Signal Processor)

A crescente integração dos computadores e máquinas com o mundo real criou, nas últimas décadas, uma grande demanda de maior eficiência na análise de sinais. Muitas aplicações,

tais como o cálculo de potência para controle de conversores, demandam execução de algoritmos com tempo de processamento mínimo, para que seja possível a análise e tomada de decisão em tempo real.

Nesse contexto, a sigla DSP significa *Digital Signal Processor*, ou Processador Digital de Sinais, e se refere a unidades de processamento que são otimizadas para realizar operações de multiplicação, as quais são uma das operações de maior custo computacional. Como consequência dessa otimização, essas unidades se tornam capazes de realizar operações de processamento de sinais digitais com maior velocidade e menor consumo energético que as unidades de processamento comuns.

Transformadas de Fourier, filtros e convolução são exemplos desses algoritmos de processamento de sinais, que são baseados em diversas multiplicações. Além disso, costumam apresentar acesso de dados em alta velocidade, maior fidelidade numérica e controle de tempo de execução de código com maior precisão (EYRE; BIER, 2000).

Todas essas características tornam os DSPs ideais para diversas aplicações, como comunicações, imagens médicas, radares, sonares, processamento e análise de áudio, entre outras (SMITH, 1997).

O DSP opera em conjunto com outros circuitos auxiliares, chamados periféricos, tais como geradores de PWM, conversores analógico-digitais, conversores digital-analógicos, entre outros. Devido à sua proposta de fornecer alta performance e ser versátil, configurável para diferentes aplicações, existe uma maior dificuldade no processo de programação. Essa programação é feita em mais baixo nível, na maioria das vezes através da alteração de valores diretamente em seus registradores, o que permite ao programador ter um controle profundo sobre a operação do dispositivo.

Para fornecer maior versatilidade, diversos pinos de entrada e saída do DSP podem ser remapeados para que estejam associados a periféricos diferentes, de forma que essa seleção deve ser realizada em todos os pinos utilizados. Além disso, cada periférico possui uma grande quantidade de recursos a serem configurados.

O módulo de PWM permite configurar não somente a frequência de chaveamento e o ciclo de serviço, mas também a frequência de clock do módulo, o modo de contagem do registrador, que pode ser feita de forma crescente, decrescente, ou triangular, se serão sincronizados a outros sinais de PWM ou associados a outros eventos, a habilitação da função PWM Chopper para que quando em sinal alto ele apresente um sinal chaveado de alta frequência, o modo de

atualização dos seus registradores, que pode ser diretamente mediante comando de gravação ou de forma sincronizada, entre vários outros.

O módulo de conversão analógica-digital permite configurar a frequência do clock, a resolução do sinal digital de saída, o modo de conversão que pode ser referenciado no terra ou a dois fios para sinais balanceados, o instante de início do processo de conversão, o que será feito após o fim da conversão, entre outros parâmetros.

Apesar de aumentar a complexidade e o nível de dificuldade para se programar um DSP, a possibilidade de trabalhar com todos esses diferentes parâmetros permite que o processador forneça o melhor desempenho independente da aplicação.

3 METODOLOGIA

Para a realização deste projeto, inicialmente, foi necessário realizar uma revisão da literatura sobre as principais técnicas que são empregadas para cálculo de potência em corrente alternada, com a intenção de fazer toda elaboração dos códigos e, posteriormente, ser possível implementá-los no DSP em questão, realizando os testes e validações.

Em razão disto, pode-se dividir a metodologia em duas grandes etapas, sendo elas: elaboração do código que implementa o cálculo de potências utilizando a Referência Síncrona e a elaboração do código que implementa o mesmo cálculo utilizando Filtros Adaptativos Sintonizados, cada uma com sua respectiva etapa de aplicação e testes. Para ambas as técnicas o programa foi primeiramente desenvolvido através do S-Function, extensão do Simulink que realizou a simulação do código em linguagem C, o qual posteriormente foi aplicado no DSP TMS320F28379D.

3.1 Cálculo de potências utilizando a Referência Síncrona

O desenvolvimento e implementação do algoritmo para cálculo de potências utilizando a Referência Síncrona foi subdividido em elementos e operações que o compõem, para simplificar o processo, sendo essas divisões o PLL, a Referência Síncrona monofásica, o Filtro Passa-Baixas de terceira ordem e o cálculo de potências propriamente dito. A metodologia utilizada em cada seção é descrita nos seguintes tópicos.

3.1.1 Implementação do PLL

No cálculo referente à determinação da fase instantânea do sinal de entrada foi utilizado o algoritmo de Yazdani, Bakhshai e Jain (2010) para Filtro Adaptativo Sintonizado com Estimador de Frequência (FAS-estimador). Para este caso, em que serão analisados dados da rede elétrica, foi implementado além cálculo da componente fundamental somente o 5º harmônico, pois é o que tem maior relevância, o que levou a maior precisão do estimador sem aumentar muito o custo computacional.

O conjunto de equações que define esse algoritmo foi então discretizado. Dentre os vários métodos disponíveis para realizar esse processo, foram avaliados a Aproximação de Euler e o método de Tustin. Dentre estes, optou-se por utilizar o primeiro, pois apresenta resulta-

dos muito próximos aos obtidos com outros métodos, porém com menor custo computacional, utilizando menor número de operações de multiplicação e divisão (SOARES, 1996).

Ao discretizar o algoritmo utilizando a aproximação de Euler, as equações obtidas foram as seguintes:

$$\omega(n+1) = \omega(n) \; - \; T \; \gamma \, x_1(n) \; \omega(n) \; e(n) \tag{3.1}$$

$$\dot{x}_i(n+1) = \dot{x}_i(n) + T \; \left[2\zeta_i \; \omega(n) \; e(n) - i^2 \omega^2(n) \; x_i(n) \right] \tag{3.2}$$

$$x_i(n+1) = x_i(n) \; + \; T \; \dot{x}_i(n) \; \rightarrow \; i = 1, 2, ..., N \tag{3.3}$$

$$e(n+1) = d(n) - \sum_{l=0}^{N} x_l(n) \tag{3.4}$$

onde, T é o período de amostragem ($t = nT$), d é o sinal observado normalizado, $x1$ é a componente fundamental, ω é a frequência fundamental estimada, e é o erro e i a ordem harmônica de interesse. γ e ζ são parâmetros que determinam a dinâmica do estimador, sua velocidade e precisão, eles correspondem ao coeficiente de adaptação e ao fator de amortecimento, respectivamente.

Para determinar os parâmetros do algoritmo, utilizou-se como ponto inicial os parâmetros definidos por Guimarães (2019). Esses parâmetros foram então alterados em pequena escala por meio de iterações com o objetivo de melhorar a resposta dos filtros. Os valores determinados para melhor desempenho da estimação foram $\gamma = 4000$, $\zeta_1 = 0.19$, $\zeta_5 = 0.3$ e $T = 4.99 \times 10^{-5}$. Além disso os estados foram iniciados como nulos, com exceção da frequência, que teve 366,99 rad/s como valor inicial.

A Figura 3.1 apresenta o código do algoritmo de PLL discretizado por Aproximação de Euler implementado no S-Function. As variáveis utilizadas correspondem às variáveis do conjunto de equações apresentado anteriormente. GAMA representa γ, ZETA representa ζ, w representa ω, x representa o sinal de entrada, xd representa $\dot{x}$, e representa o erro e wt representa ωt que é a fase calculada, sendo que algumas dessas variáveis são acompanhadas um número representando a ordem do harmônico a que corresponde.

3.1.2 Referência Síncrona monofásica

A Referência Síncrona monofásica foi implementada utilizando as funções *sin* e *cos* da biblioteca *math.h*. Elas recebem como parâmetro a fase estimada pelo PLL (variável wt[0]) e

Figura 3.1 – Código PLL por Aproximação de Euler

```
//Inicializando
#define T 4.99e-05
#define GAMA 4000
#define ZETA 0.19
#define ZETA2 0.3

double w1=377, x1=0, xd1=0, x5=0, xd5=0, e1=0;



//Loop
w1 = w1 - T*GAMA*x1*w1*e1;

xd1 = xd1 + T*(2*ZETA*w1*e1 - w1*w1*x1);
x1 = x1 + T*xd1;

xd5 = xd5 + T*(2*ZETA2*w1*e1 - 25*w1*w1*x5);
x5 = x5 + T*xd5;

e1 = d[0]/180 - xd1 - xd5;

wt[0] = atan2(xd1,(-w1*x1));
```

são multiplicadas pelo sinal de entrada (variável in[0]), sendo obtidos os sinais direto e quadratura deste sinal (variáveis d[0] e q[0]).

Figura 3.2 – Código Transformada de Park

```
d[0] = 2*in[0]*sin(wt[0]);
q[0] = -2*in[0]*cos(wt[0]);
```

3.1.3 Filtro Passa-Baixas de terceira ordem

Diversos algoritmos de filtros passa-baixas poderiam ser utilizadas para filtrar os sinais obtidos pela transformação e obter o seu valor médio. Neste caso utilizou-se uma implementação de primeira ordem recursiva em cascata, que se trata de filtro fácil de projetar, com um algoritmo simples e de baixo custo computacional.

Cada estágio consiste em multiplicar a saída do filtro no instante anterior por um coeficiente α e o sinal a ser filtrado (entrada) pelo coeficiente $1-\alpha$, como indicado na Equação 3.5, onde $x[n]$ representa o sinal de entrada e $y[n]$ representa o sinal de saída.

$$y[n] = \alpha\, y[n-1] + (1-\alpha)\, x[n] \tag{3.5}$$

Utilizando 3 estágios em cascata obtém-se um filtro de terceira ordem. Isso leva a uma maior inclinação do ganho na resposta em frequência, permitindo que haja maior filtragem do sinal, sendo possível aumentar a frequência de corte para se obter dinâmica mais rápida.

A relação entre α e a frequência de corte para cada estágio se dá de acordo com a Equação 3.6 (SMITH, 1997), onde f_c se refere à frequência de corte como fração da frequência de amostragem. Assim foi possível determinar o valor 0.9912595152854919 para esse parâmetro, que define uma frequência de corte em 28 Hz para 20040 Hz de amostragem.

$$\alpha = e^{-2\pi f_c} \tag{3.6}$$

A Figura 3.3 apresenta o código implementado, onde alpha representa o coeficiente α, alpha1 representa o coeficiente $(1-\alpha)$, x[0] representa o sinal de entrada, y[0] o sinal de saída filtrado, e y1, y2 e y3 representam os sinais de saída nos instantes $n-1$, $n-2$ e $n-3$, respectivamente.

Figura 3.3 – Código Filtro Passa-Baixas Recursivo Cascata

```
//Inicializando
double alpha = 0.99125951528549194;
double alpha1 = 1 - alpha;

double y1 = 0, y2 = 0, y3 = 0;

//Loop
y1 = alpha*y1 + alpha1*x[0];
y2 = alpha*y2 + alpha1*y1;
y3 = alpha*y3 + alpha1*y2;
y[0] = y3;
```

3.1.4 Cálculo de potências

Uma vez determinados os sinais diretos e de quadratura da tensão e da corrente, a potência foi calculada como apresentado por Ferreira et al. (2015), de acordo com as Equações 2.6 e 2.7.

Foi possível observar que ao finalizar o cálculo utilizando essas equações, o valor reportado de potência ativa apresenta uma oscilação relevante, com aproximadamente 5% do valor

de potência calculado. A potência reativa calculada também apresentou oscilações, porém com amplitude tolerável. Esse foi o motivo pelo qual optou-se por processar o sinal de saída da potência ativa utilizando um filtro passa-baixas de cascata, de forma que não aumentasse muito o tempo de estabilização.

A Figura 3.4 apresenta a implementação em C do cálculo de potência após a cálculo dos sinais direto e quadratura da tensão e da corrente utilizando a transformada de Park, onde Vd[0] representa o sinal de eixo direto filtrado da tensão, Vq[0] representa o sinal de quadratura filtrado da tensão, Id[0] representa o sinal de eixo direto filtrado da corrente, Iq[0] representa o sinal de quadratura filtrado da corrente, Q[0] representa a potência reativa calculada, pp representa a potência ativa calculada, P[0] representa a potência ativa calculada filtrada, P1, P2 e P3 representam os valores de potência ativa filtrada nos tempos de amostragem anteriores.

Figura 3.4 – Código Cálculo de Potências Ativa e Reativa

```
pp = (Vd[0]*Id[0]+Vq[0]*Iq[0])/2;

P1 = alpha*P1 + alpha1*pp;
P2 = alpha*P2 + alpha1*P1;
P3 = alpha*P3 + alpha1*P2;

P[0] = P3;

Q[0] = (Vd[0]*Iq[0]-Vq[0]*Id[0])/2;
```

3.2 Filtro Adaptativo Sintonizado com Estimador de Frequência

Para realizar o cálculo das potências utilizando este método, é necessário obter os sinais ortogonais relativos à tensão e à corrente, os quais são utilizados para calcular a potência.

Assim como no caso anterior, o desenvolvimento e implementação do algoritmo para cálculo de potências utilizando o Filtro Adaptativo foi subdividido em elementos e operações que o compõem, para simplificar o processo, sendo essas divisões a implementação do filtro para tensão, a implementação do filtro para corrente e o cálculo de potências propriamente dito. A metodologia utilizada em cada seção é descrita nos tópicos a seguir.

3.2.1 Filtro Adaptativo Sintonizado aplicado à tensão

Os sinais ortogonais da tensão são obtidos passando o sinal pelo Filtro Adaptativo Sintonizado com Estimador de Frequência. A implementação foi a mesma do PLL, com a diferença de que não foi necessária a operação de arco tangente para calcular a fase do sinal. Neste caso utiliza-se as duas variáveis de estado para calcular os sinais ortogonais da tensão.

A Figura 3.5 apresenta o código desenvolvido para esta etapa do algoritmo de cálculo de potências utilizando Filtro Adaptativo. Ele recebe como entrada o sinal de tensão e tem como saídas os sinais ortogonais da tensão na frequência fundamental, além de estimar a frequência do sinal de entrada.

Nesse código, V representa o sinal de tensão, Vd representa $\dot{V}$, Ve representa o sinal de erro, V[0] e V90[0] representam os sinais ortogonais calculados, sendo que algumas dessas variáveis são acompanhadas um número representando a ordem do harmônico a que corresponde.

Figura 3.5 – Código do Filtro Adaptativo aplicado à Tensão

```
//---------- FAS TENSAO COM ESTIMADOR DE FREQUENCIA
w1 = w1 - T*GAMA*V1*w1*Ve1;

Vd1 = Vd1 + T*(2*ZETA*w1*Ve1 - w1*w1*V1);
V1 = V1 + T*Vd1;

Vd5 = Vd5 + T*(2*ZETA2*w1*Ve1 - 25*w1*w1*V5);
V5 = V5 + T*Vd5;

Ve1 = Vin[0] - Vd1 - Vd5;

V[0] = Vd1;
V90[0] = -w1*V1;
```

3.2.2 Filtro Adaptativo Sintonizado aplicado à corrente

O procedimento para obtenção dos sinais ortogonais da corrente é idêntico ao utilizado na análise da tensão, diferenciando-se apenas pelo fato de que neste não é necessário implementar a fração do código responsável pela estimação da frequência, uma vez que esta será a mesma da tensão. Essa estimação poderia ser realizada em qualquer dos sinais, porém optou-se por utilizar o sinal de tensão por ser menos suscetível a distúrbios e deformações.

A Figura 3.6 apresenta o código desenvolvido para esta etapa do algoritmo de cálculo de potências utilizando Filtro Adaptativo. Ele recebe como entrada o sinal de corrente e tem como saídas os sinais ortogonais da corrente.

Nesse código, I representa o sinal de tensão, Id representa $\dot{I}$, Ie representa o sinal de erro, I[0] e I90[0] representam os sinais ortogonais calculados, sendo que algumas dessas variáveis são acompanhadas um número representando a ordem do harmônico a que corresponde.

Figura 3.6 – Código do Filtro Adaptativo aplicado à Corrente

```
//----------FAS CORRENTE
Id1 = Id1 + T*(2*ZETA*w1*Ie1 - w1*w1*I1);
I1 = I1 + T*Id1;

Id5 = Id5 + T*(2*ZETA2*w1*Ie1 - 25*w1*w1*I5);
I5 = I5 + T*Id5;

Ie1 = Iin[0] - Id1 - Id5;

I[0] = Id1;
I90[0] = -w1*I1;
```

3.2.3 Cálculo de potências

Uma vez obtidos os sinais de tensão e corrente filtrados, os valores de potência ativa e reativa são calculados de forma análoga à teoria dq, utilizando no lugar dos sinais direto e quadratura os sinais ortogonais calculados.

Assim como no cálculo de potências por Referência Síncrona, foi necessário filtrar os valores obtidos para a potência ativa para diminuir a oscilação presente na saída. Portanto foi utilizada a mesma implementação de três filtros de primeira ordem em cascata.

A Figura 3.7 apresenta o código desenvolvido para cálculo das potências utilizando Filtro Adaptativo, onde V[0] e V90[0] representam os sinais ortogonais calculados para a tensão, I[0] e I90[0] representam os sinais ortogonais calculados para a corrente, Q[0] representa a potência reativa calculada, pp representa a potência ativa calculada, P[0] representa a potência ativa calculada filtrada, P1, P2 e P3 representam os valores de potência ativa filtrada nos tempos de amostragem anteriores.

Figura 3.7 – Código da Estimação de Potência utilizando Filtro Adaptativo

```
//----------POTENCIAS
pp = (V[0]*I[0] + V90[0]*I90[0])/2;

P1 = alpha*P1 + alpha1*pp;
P2 = alpha*P2 + alpha1*P1;
P3 = alpha*P3 + alpha1*P2;

P[0] = P3;

Q[0] = (V[0]*I90[0] - V90[0]*I[0])/2;
```

3.3 Implementação no DSP

Os algoritmos desenvolvidos foram implementados no DSP TMS320F 28379D, da Texas instruments, utilizando um kit de desenvolvimento da própria fabricante. Foi utilizado o compilador indicado também pela fabricante, Code Composer Studio (CCS).

Uma vez que a os códigos implementados para simulação foram escritos em linguagem C, sua transferência para a IDE do compilador foi simples, sendo necessário apenas ajustar o nome de algumas variáveis e outros pequenos ajustes em relação ao código implementado no Matlab/Simulink utilizando o S-function.

As operações para cálculo de potência, no entanto, não são suficientes para alcançar esse objetivo. Foi necessário também configurar o sistema para realizar as leituras de tensão e corrente utilizando conversores analógico-digitais, conversores digital-analógicos baseados em PWM para utilizar como saída de dados para análise, e GPIOs para comando dos relés utilizados nos testes de regime transiente.

A parametrização desses periféricos do DSP foi baseada em códigos de prática de laboratório fornecidos como parte de um curso de apresentação do microcontrolador disponibilizado online pela Texas Instruments (INSTRUMENTS, 2016), e os códigos implementados contendo as configurações de periféricos e os algoritmos para cálculo de potências podem ser encontrados nos Apêndices A e B.

Os dois conversores analógico-digitais utilizados para leitura dos sinais de tensão e corrente foram configurados para iniciar a conversão após um sinal de disparo gerado por um PWM com frequência de 20040 Hz. Uma vez que foi utilizada a mesma configuração para ambos os conversores, o tempo de que levam para finalizar o processamento dos sinais é muito próximo.

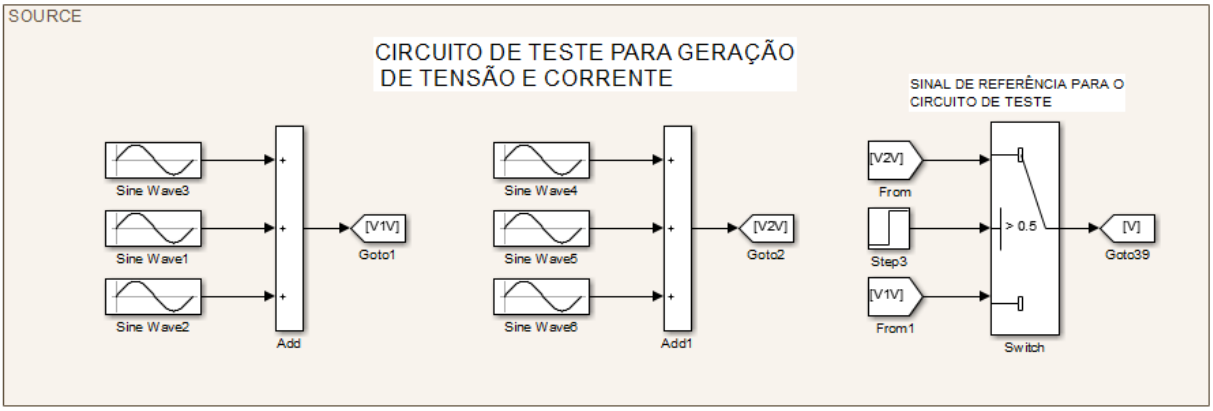
Assim, um sinal de fim de conversão do ADC é utilizado como disparo para execução de uma interrupção que contém o algoritmo de processamento dos valores entrada e cálculo da potência.

3.4 Simulação e testes

Com o objetivo de analisar a dinâmica de operação do sistema de cálculo de potências e validar os resultados obtidos, foram realizados testes por meio de simulação e em laboratório. Os testes foram divididos em duas etapas, sendo elas análise de resposta em regime estacionário e em regime transiente, e foram executadas igualmente para avaliação de ambos os algoritmos.

As simulações foram executadas por meio do Simulink, ambiente de simulação do *software* Matlab. Nele foram montados da forma mais fiel possível os circuitos que seriam usados nos testes de laboratório, através do diagrama de blocos que o programa oferece. A onda de tensão utilizada como fonte foi gerada através do somatório de três ondas senoidais, representando a frequência fundamental, o terceiro e o quinto harmônico do sinal de tensão da rede, como apresentado na Figura 3.8.

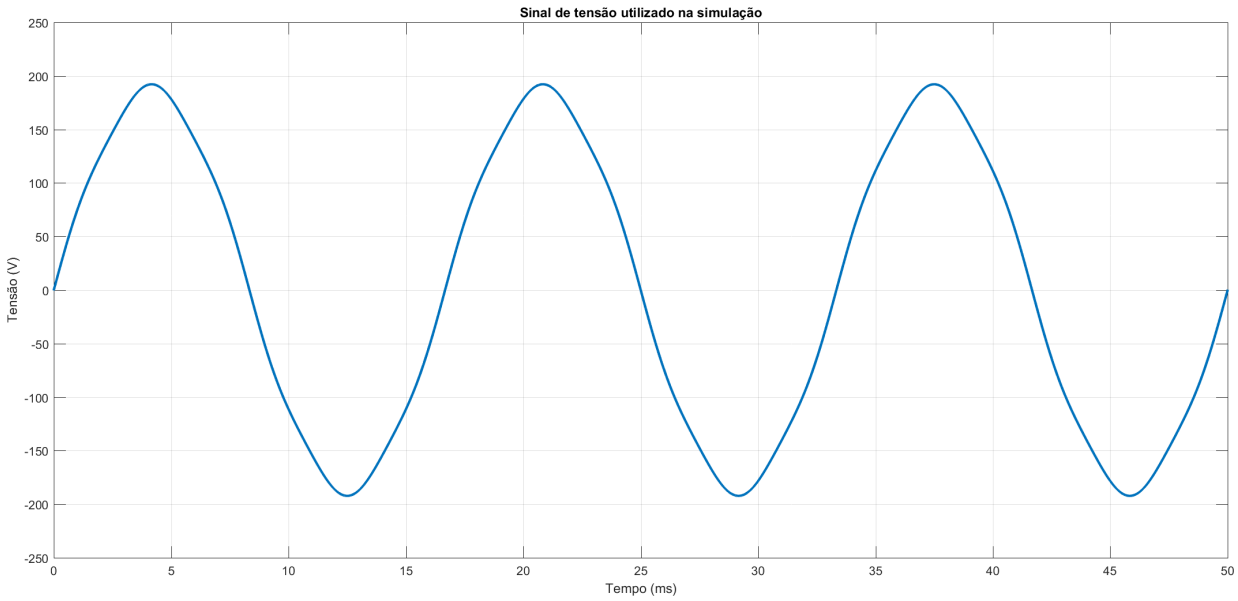
Figura 3.8 – Diagrama de simulação para gerar o sinal de tensão.



Fonte: Do autor (2021).

A função FFT (Transformada rápida de Fourier) do osciloscópio foi utilizada para obter a amplitude de cada um desses sinais na rede elétrica do laboratório, e os valores utilizados para simulação foram 188 V, 0,8 V e 5 V para a fundamental, terceiro e quinto harmônicos, respectivamente. O sinal *V2V* foi gerado da mesma forma, porém utilizando uma frequência diferente e, juntamente com o bloco Switch, foi utilizado para simulação de degraus de frequência. A Figura 3.9 apresenta o sinal obtido através desse método, sem o degrau de frequência.

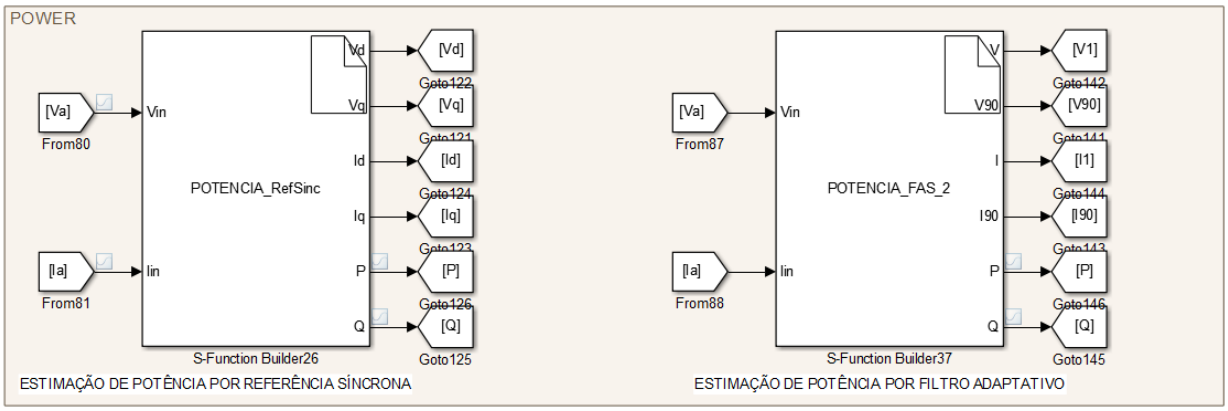
Figura 3.9 – Sinal de tensão utilizado na simulação.



Fonte: Do autor (2021).

Os sinais de tensão e corrente dos circuitos foram obtidos através dos blocos *Voltage Measurement* e *Current Measurement*. Por fim os algoritmos de cálculo de potência foram executados no ambiente de simulação através do S-Function, uma extensão do Matlab que permite ao Simulink compilar e executar códigos de linguagem C, ou até escrevê-los diretamente, sem precisar utilizar outro *software*, conforme apresentado na Figura 3.10.

Figura 3.10 – Diagrama de simulação dos blocos de cálculo de potências.



Fonte: Do autor (2021).

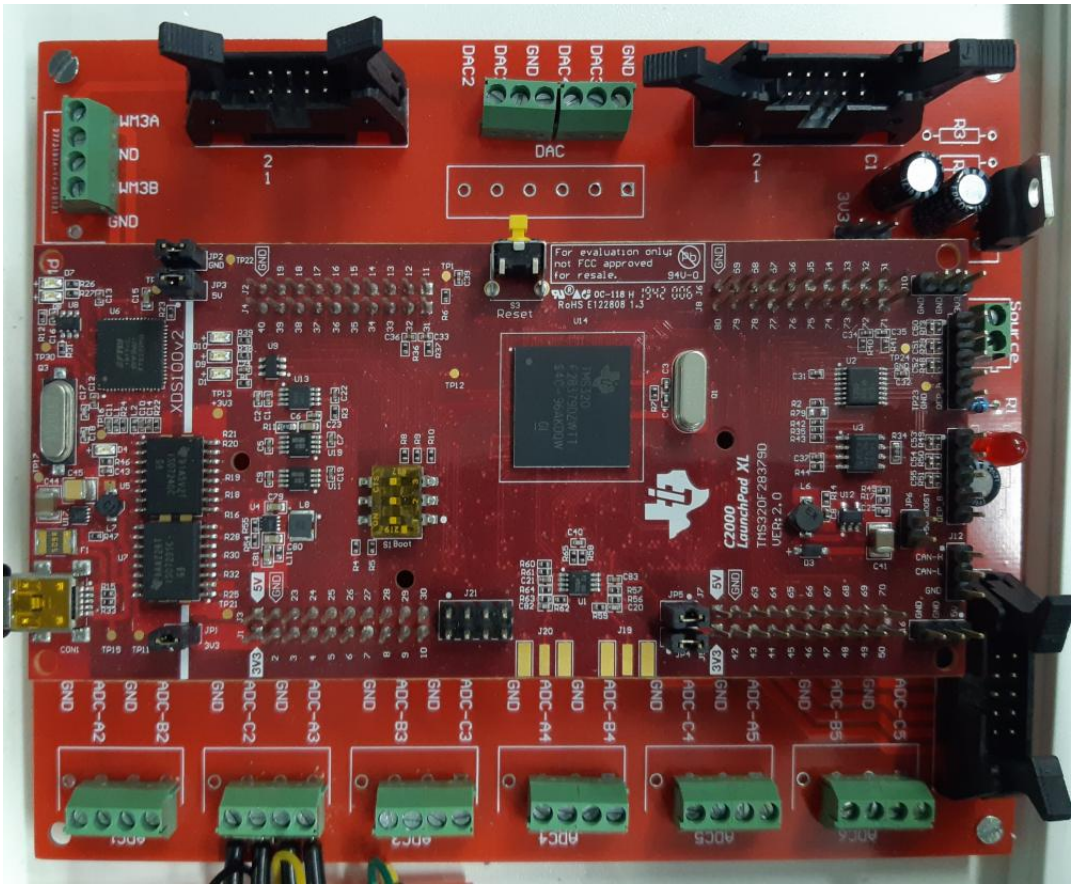
Os testes de laboratório foram realizados utilizando o o DSP TMS320F28379D contendo o algoritmo de cálculo da potências e os periféricos configurados, um sensor de corrente tipo Hall LV 25, um sensor de tensão tipo Hall LA 55, circuitos de condicionamento para ajuste dos sinais dos sensores, um relé de estado sólido SSR-40 DA e um osciloscópio Tektronix TPS

2024 de quatro canais isolados para captura e análise dos dados de saída através dos DACs. As cargas foram montadas utilizando módulos de componentes passivos do laboratório, e todos apresentam valores de resistência, indutância e capacitância iguais, sendo $125\,\Omega$, $0,6H$ e $15\mu F$ respectivamente.

Os valores calculados pelo DSP foram capturados utilizando o osciloscópio, conectado às saídas dos conversores digital-analógicos baseados em PWM. Essa saída gera um sinal de tensão entre 0 e 3,3V, portanto os valores passados para os DACs foram convertidos para essa faixa através de uma escala linear personalizada para cada grandeza, considerando os valores mínimo e máximo que seriam medidos. Após as capturas dos dados, esses valores foram convertidos novamente para a escala inicial, permitindo sua análise.

A Figura 3.11 apresenta a placa do kit de desenvolvimento do DSP TMS320F28379D utilizado nos testes de laboratório, desenvolvido pela Texas Instruments, montado em uma placa de circuito impresso de expansão desenvolvida para facilitar o manuseio das conexões de entrada e saída.

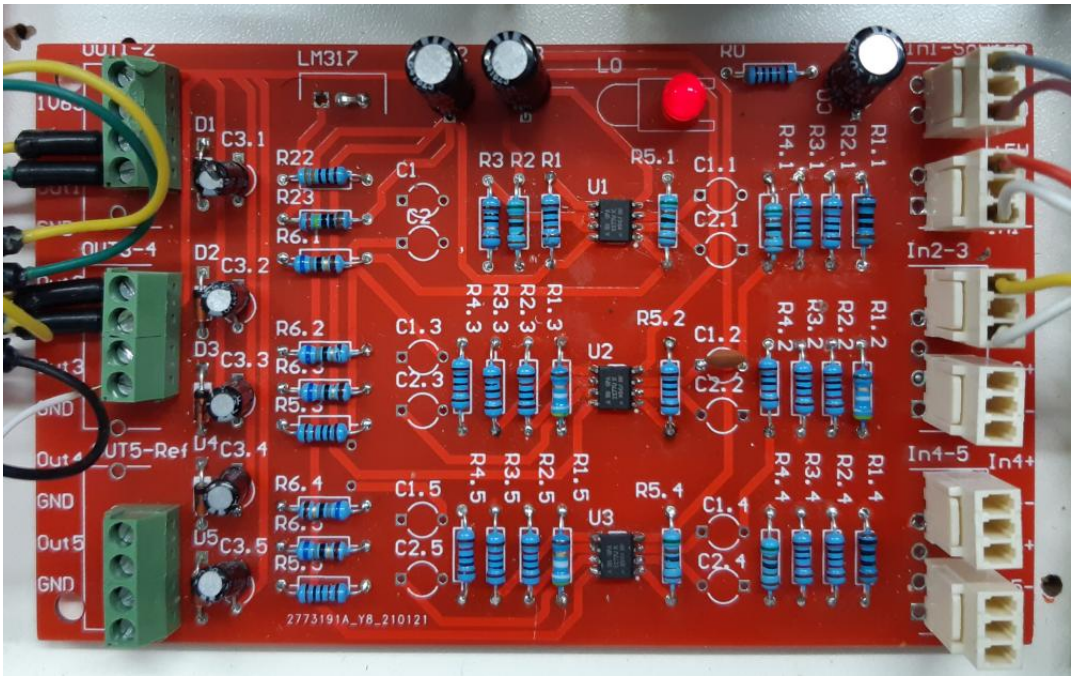
Figura 3.11 – Kit de desenvolvimento do DSP TMS320F28379D e placa de expansão.



Fonte: Do autor (2021)

A Figura 3.12 apresenta a placa de o circuito impresso desenvolvida contendo o circuito de condicionamento do sinal dos sensores.Esse circuito é responsável por compatibilizar os níveis de tensão de saída dos sensores com os níveis de tensão de leitura do conversor ADC do DSP. À direita na imagem estão os conectores de entrada dos sinais provenientes dos sensores, e à esquerda os conectores de saída dos sinais condicionados que serão conectados ao DSP.

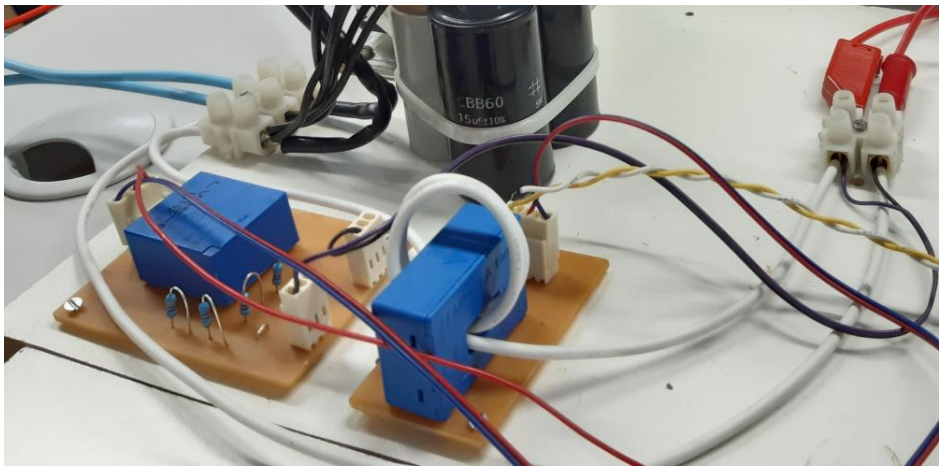
Figura 3.12 – Placa de circuito impresso do circuito de condicionamento de sinais.



Fonte: Do autor (2021).

A Figura 3.13 apresenta os sensores de efeito hall (componentes azuis) utilizados nos testes em laboratório, sendo o sensor de tensão da marca LEM modelo LV 25-P (à esquerda) e o sensor de corrente da marca LEM modelo LA 100-P (à direita).

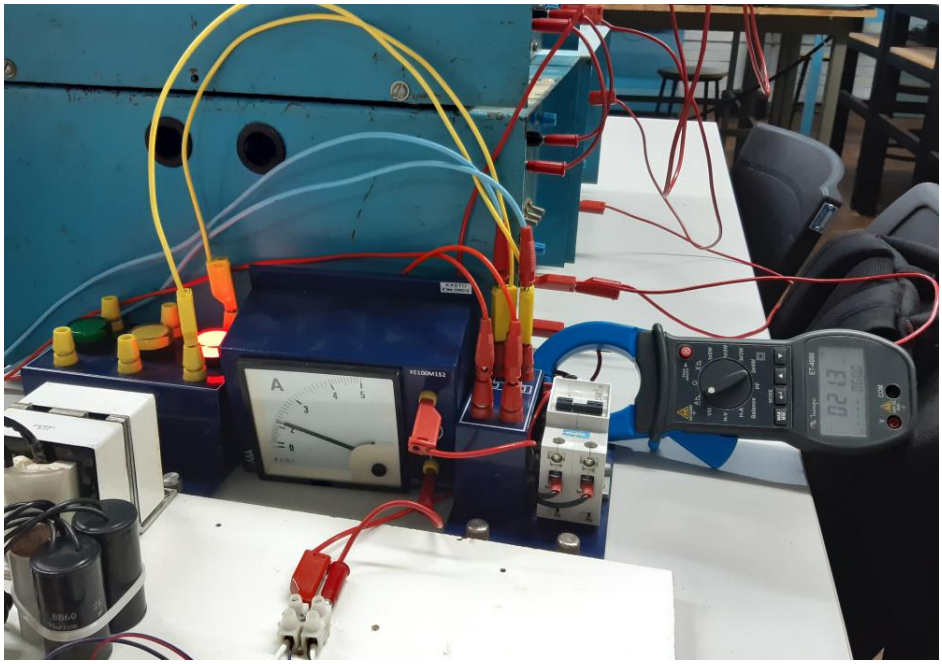
Figura 3.13 – Sensores utilizados nos testes em laboratório.



Fonte: Do autor (2021).

A Figura 3.14 apresenta alguns elementos adicionados ao circuito de testes, utilizados para proteção e para facilitar a observação de status de operação do circuito. Pode-se observar um disjuntor utilizado para ligar e desligar o circuito, bem como proteger contra alta corrente devido a uma possível montagem incorreta das cargas, um amperímetro para leitura rápida da corrente demandada e LEDs indicadores de status.

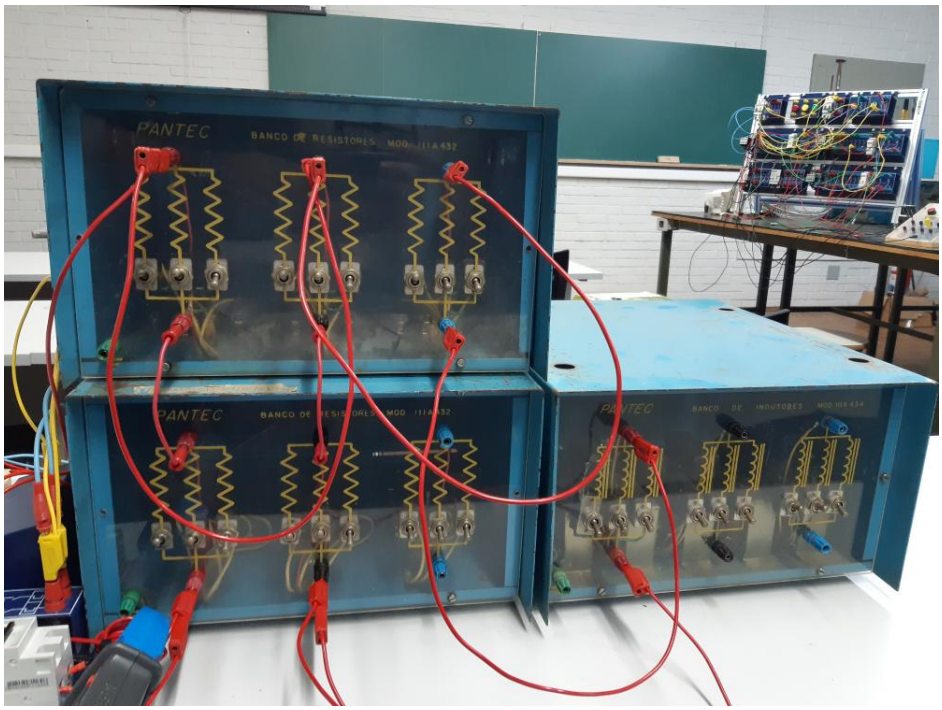
Figura 3.14 – Elementos de proteção e indicadores de status de operação do circuito.



Fonte: Do autor (2021).

A Figura 3.15 apresenta os módulos de resistores e indutores utilizados como cargas nos testes de laboratório.

Figura 3.15 – Módulos de resistores e indutores.



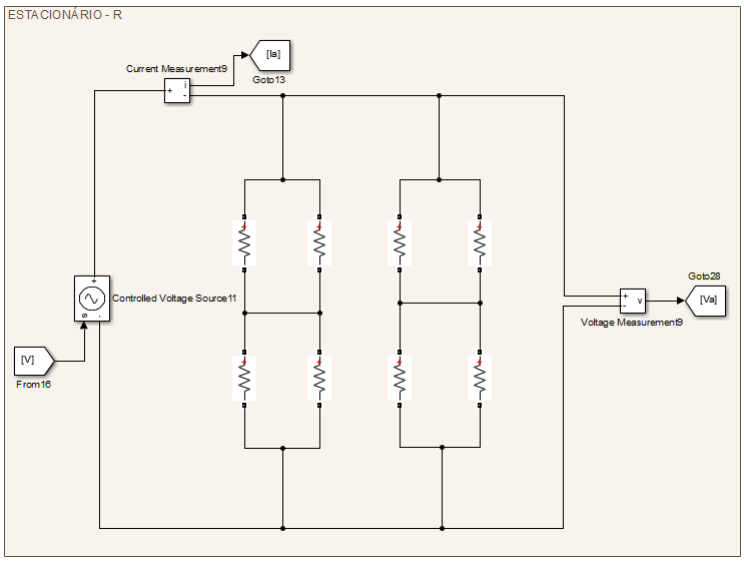
Fonte: Do autor (2021).

3.4.1 Regime estacionário

Para avaliar o comportamento em regime estacionário, como erros nos valores de potências calculados e existência de oscilações, foram realizados testes com cargas lineares R, RL e RC conectadas à rede. As potências também foram medidas através de um alicate wattímetro ET 4080 da Minipa, para fins de comparação com os resultados obtidos. As figuras a seguir apresentam os diagramas utilizados para simulação, os quais também representam os circuitos utilizados nos testes práticos.

O circuito utilizado para análise de regime estacionário com carga R é apresentado na Figura 3.16, onde se obteve impedância equivalente $Z_{eq} = 62,5\,\Omega$.

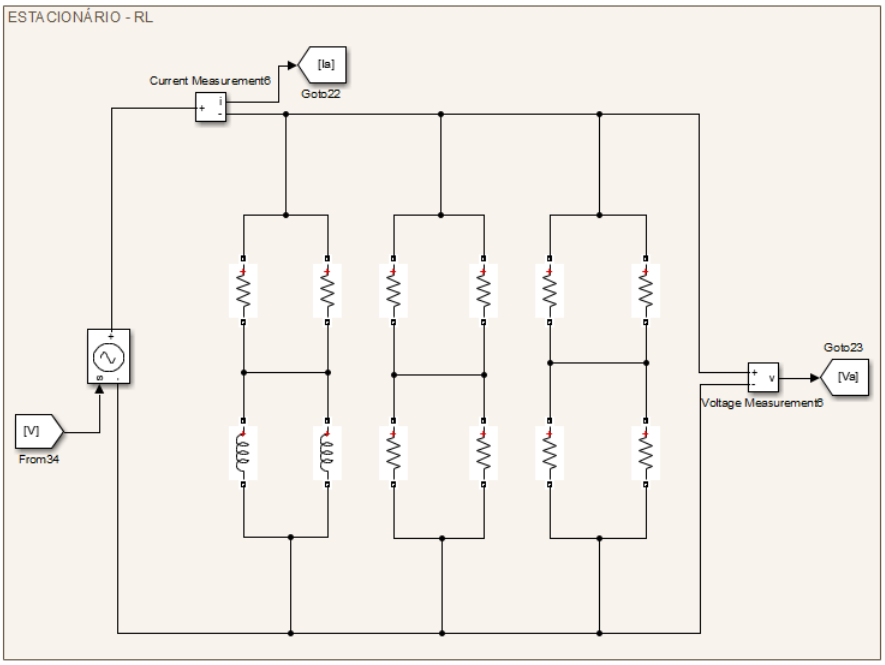
Figura 3.16 – Diagrama de simulação de regime estacionário com carga R.



Fonte: Do autor (2021).

O circuito utilizado para análise de regime estacionário com carga RL é apresentado na Figura 3.17, onde se obteve impedância equivalente $Z_{eq} = 45,3 + j15,5\,\Omega$.

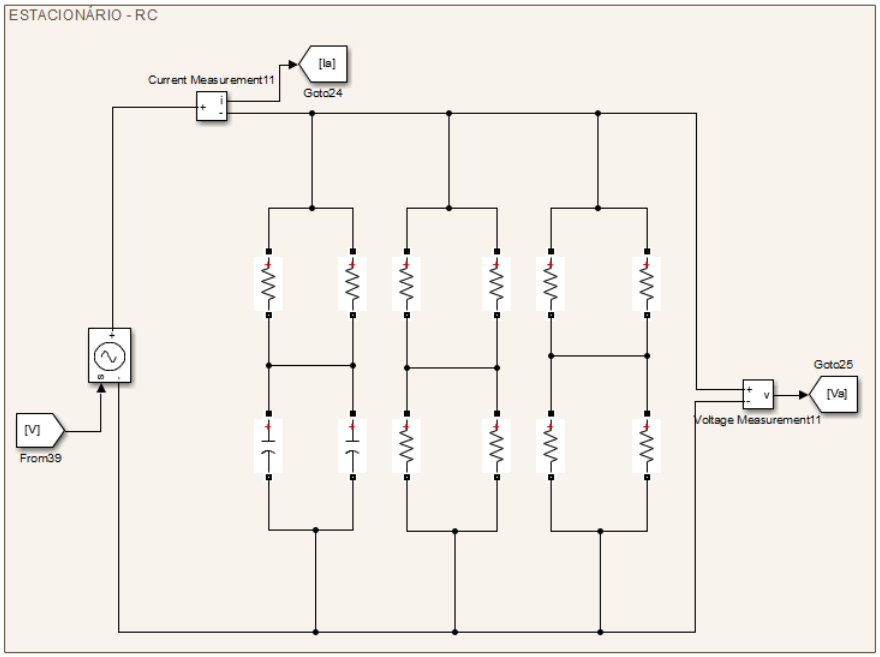
Figura 3.17 – Diagrama de simulação de regime estacionário com carga RL.



Fonte: Do autor (2021).

O circuito utilizado para análise de regime estacionário com carga RC é apresentado na Figura 3.18, onde se obteve impedância equivalente $Z_{eq} = 41,7 - j14,5\,\Omega$.

Figura 3.18 – Diagrama de simulação de regime estacionário com carga RC.



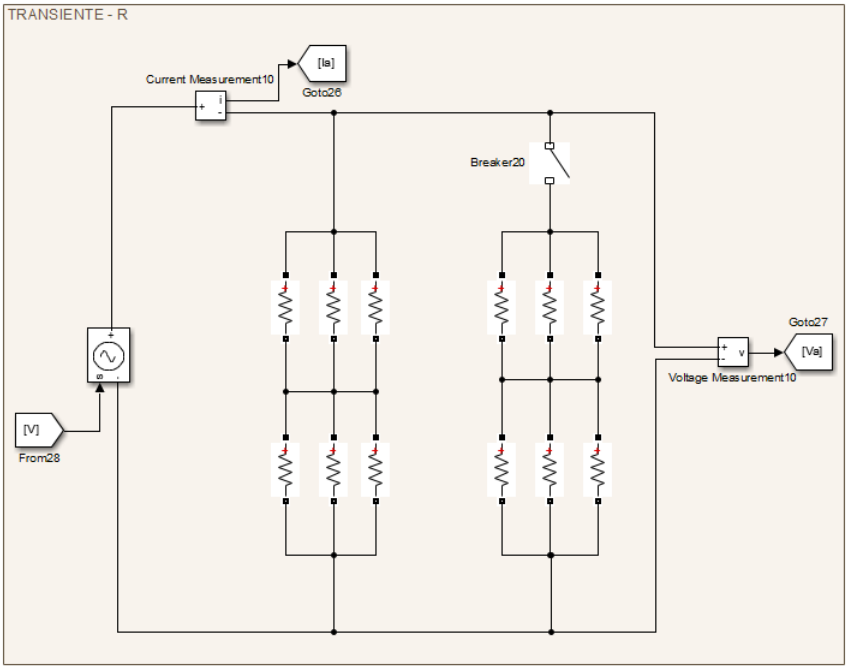
Fonte: Do autor (2021).

3.4.2 Regime transiente

Para avaliar o comportamento em regime transiente, como tempo de convergência e máximo pico, foram realizados testes com degraus de cargas R, RL e RC, bem como degrau de tensão com carga RL. Utilizou-se um relé de estado sólido, representado pelo bloco *Breaker* nos diagramas de simulação apresentados a seguir nas Figuras 3.19 a 3.23, para interromper a conexão de metade da carga total. Na implementação prática, o relé foi acionado por uma saída GPIO do DSP, e o comando de acionamento foi enviado via USB pelo computador. A captura do osciloscópio foi sincronizada utilizando o sinal de saída GPIO do relé como trigger.

O circuito utilizado para análise de regime transiente com carga R é apresentado na Figura 3.19, onde se obteve impedância equivalente $Z_{eq} = 83,3\,\Omega$ para cada associação de cargas, ou seja, $Z_{eq} = 41,7\,\Omega$ após o acionamento do segundo conjunto.

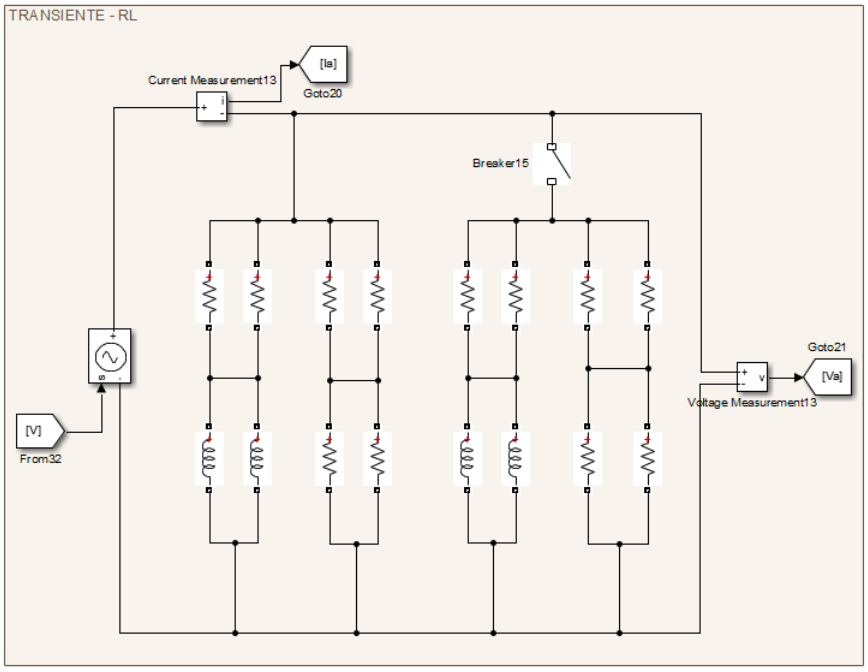
Figura 3.19 – Diagrama de simulação de regime transiente com degrau de carga R.



Fonte: Do autor (2021).

O circuito utilizado para análise de regime transiente com carga RL é apresentado na Figura 3.20, onde se obteve impedância equivalente $Z_{eq} = 63,9 + j36,9\,\Omega$ para cada associação de cargas, ou seja, $Z_{eq} = 32 + j18,4\,\Omega$ após o acionamento do segundo conjunto.

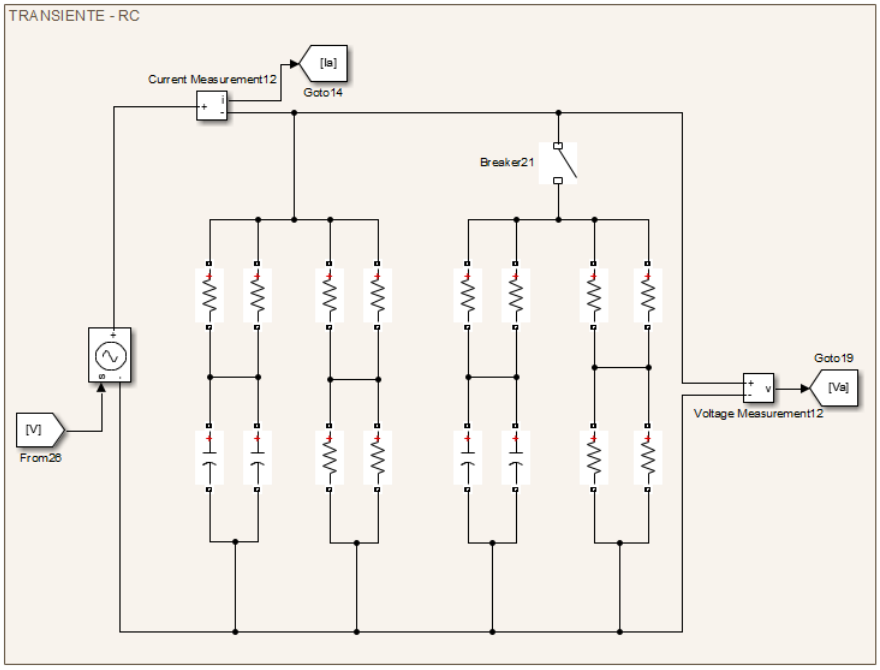
Figura 3.20 – Diagrama de simulação de regime transiente com degrau de carga RL.



Fonte: Do autor (2021).

O circuito utilizado para análise de regime transiente com carga RC é apresentado na Figura 3.21, onde se obteve impedância equivalente $Z_{eq} = 56,8 - j32,2\,\Omega$ para cada associação de cargas, ou seja, $Z_{eq} = 28,4 - j16,1\,\Omega$ após o acionamento do segundo conjunto.

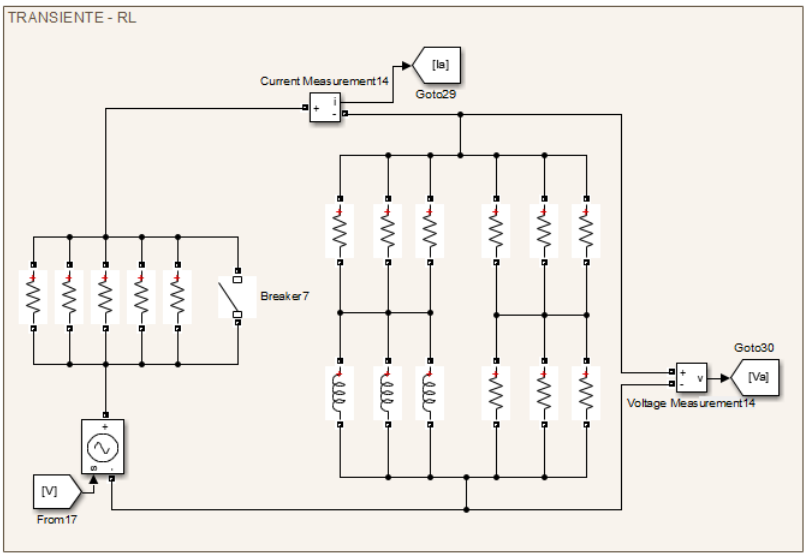
Figura 3.21 – Diagrama de simulação de regime transiente com degrau de carga RC.



Fonte: Do autor (2021).

O degrau de tensão foi obtido conectando uma associação paralela de resistores em série com a carga, causando uma queda de tensão. O relé conectado em paralelo com esses resistores, ao ser acionado fecha um curto circuito entre seus terminais, permitindo que a carga recebe a tensão total da fonte. Dessa forma foi possível aplicar um degrau de $92,6\,VRMS$ para $135,8\,VRMS$. O circuito utilizado para análise de regime transiente com tensão é apresentado na Figura 3.22, onde se obteve impedância equivalente $Z_{eq} = 42,6 + j24,6\,\Omega$.

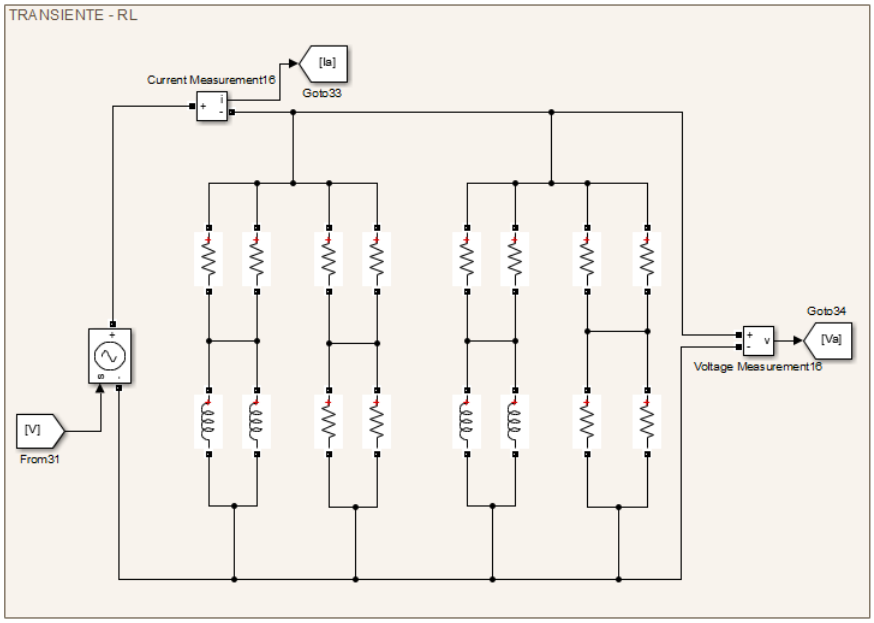
Figura 3.22 – Diagrama de simulação de regime transiente com degrau de tensão.



Fonte: Do autor (2021).

Foi testado em simulação o comportamento mediante degrau de frequência, este porém não foi realizado em laboratório devido à dificuldade de implementação com os recursos disponíveis. Sua simulação foi realizada gerando o sinal de tensão como mostrado na Figura 3.8, programada para gerar um degrau de 60 para $63Hz$ ($376,99$ para $395,84 rad/s$). O circuito utilizado para análise de regime transiente com tensão é apresentado na Figura 3.23, onde se obteve impedância equivalente $Z_{eq} = 32 + j18,4\,\Omega$.

Figura 3.23 – Diagrama de simulação de regime transiente com degrau de frequência.



Fonte: Do autor (2021).

4 RESULTADOS E DISCUSSÕES

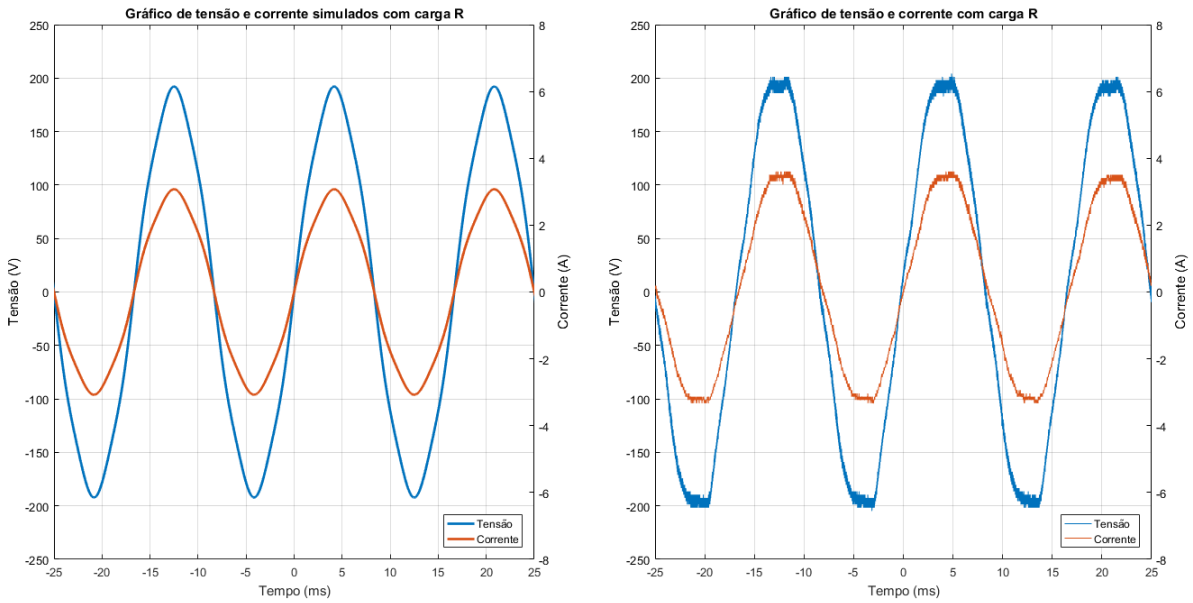
Os tópicos a seguir apresentam os resultados obtidos através das simulações e dos testes de laboratório, bem como as análises realizadas para a implementação de cálculo de potência utilizando os dois métodos analisados: Referência Síncrona e Filtro Adaptativo com Estimador de Frequência.

Em seguida serão apresentados os resultados obtidos relativos ao comportamento em regime estacionário.

4.1 Regime Estacionário

A Figura 4.1 apresenta os sinais de tensão e corrente dos testes em ambiente simulado e em laboratório com carga R em regime permanente. Conforme descrito na metodologia, este arranjo possui uma resistência de $62,5\,\Omega$, sendo alimentada por uma tensão com $136\,VRMS$ na frequência fundamental em simulação. Isso resulta em um corrente simulada de $2,2\,A\,RMS$. Valores similares foram encontrados para tensão e corrente medidos em laboratório que possuem os seguintes valores: $138,7\,VRMS$ e $2,2\,A\,RMS$.

Figura 4.1 – Gráficos de tensão e corrente para carga R em regime estacionário, obtidos por simulação e em laboratório .

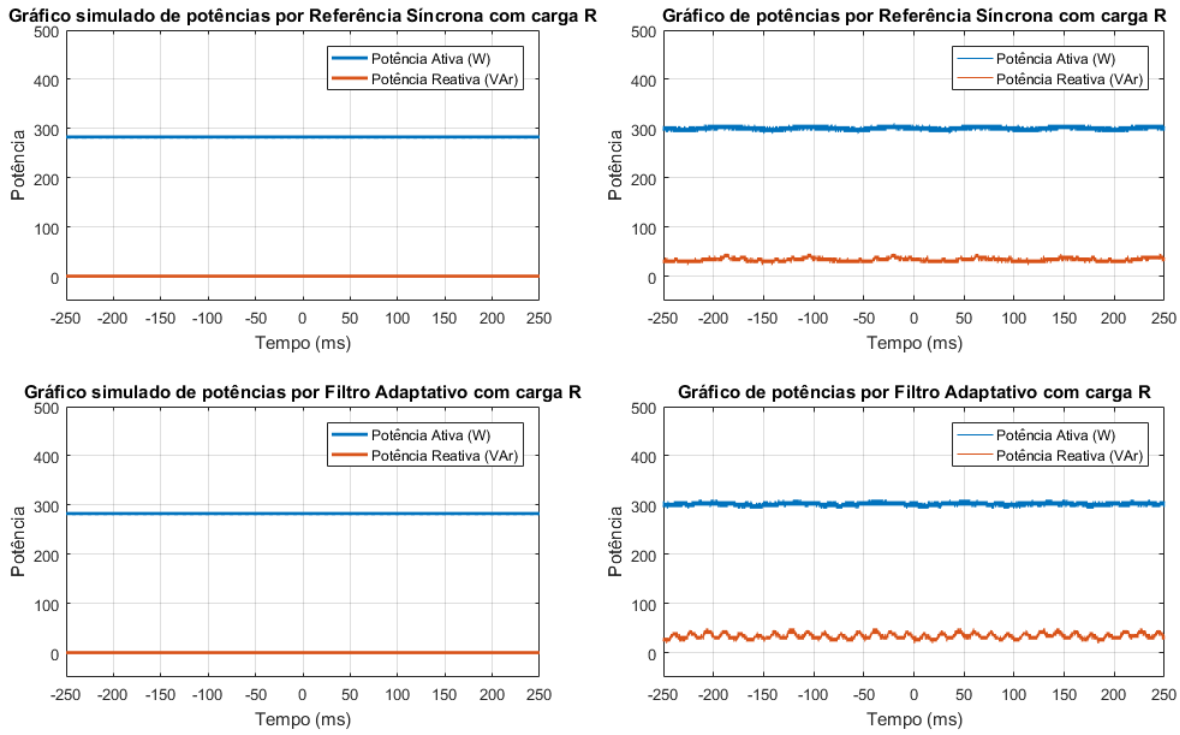


Fonte: Do autor (2021).

A Figura 4.2 apresenta os resultados obtidos de potência ativa e reativa nas mesmas condições, por meio de ambos os algoritmos analisados. Para a Referência Síncrona os valores de potência apresentados foram de $283W$ e $0VAr$ na simulação, e $300W$ e $33VAr$ em laboratório.

Por fim para o Filtro Adaptativo foram de $282W$ e $0VAr$ na simulação, e $302W$ e $34VAr$ em laboratório, neste último caso com oscilações de amplitude $15VAr$ no cálculo da potência reativa. A diferença entre os valores obtidos serão discutidas posteriormente.

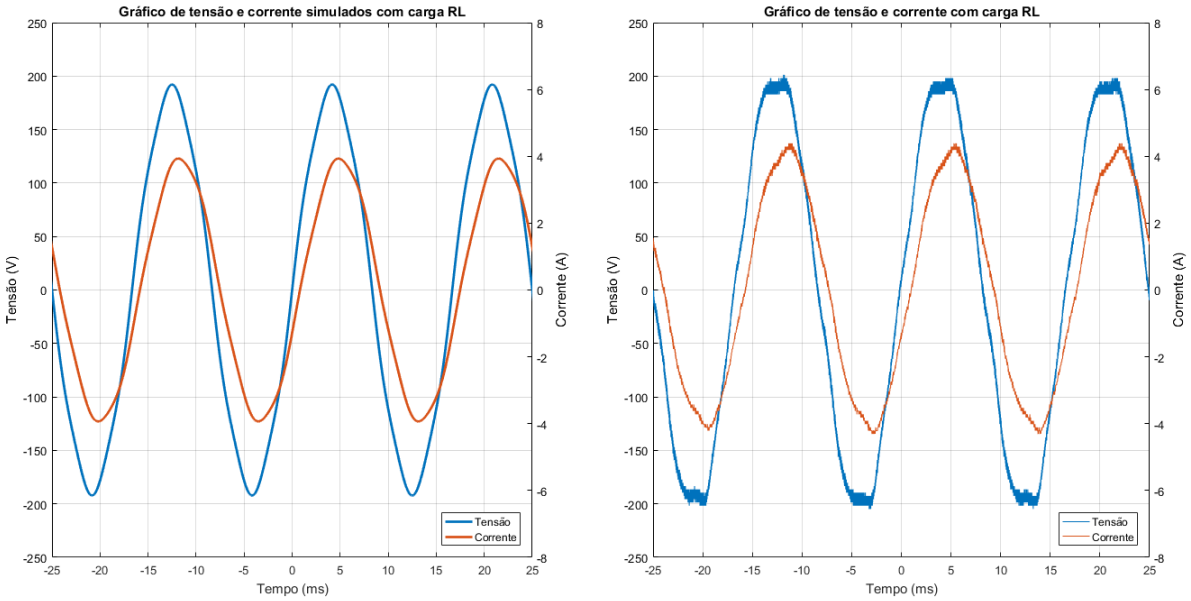
Figura 4.2 – Gráficos de potência ativa e reativa para carga R em regime estacionário, obtidos por simulação e em laboratório .



Fonte: Do autor (2021).

A Figura 4.3 apresenta os sinais de tensão e corrente dos testes em ambiente simulado e em laboratório com carga RL em regime permanente. Conforme descrito na metodologia, este arranjo possui uma resistência de $45,3 + j15,5\,\Omega$, sendo alimentada por uma tensão com $136VRMS$ na frequência fundamental em simulação. Isso resulta em um corrente simulada de $2,8ARMS$. Valores similares foram encontrados para tensão e corrente medidos em laboratório que possuem os seguintes valores: $136,5VRMS$ e $3,0ARMS$.

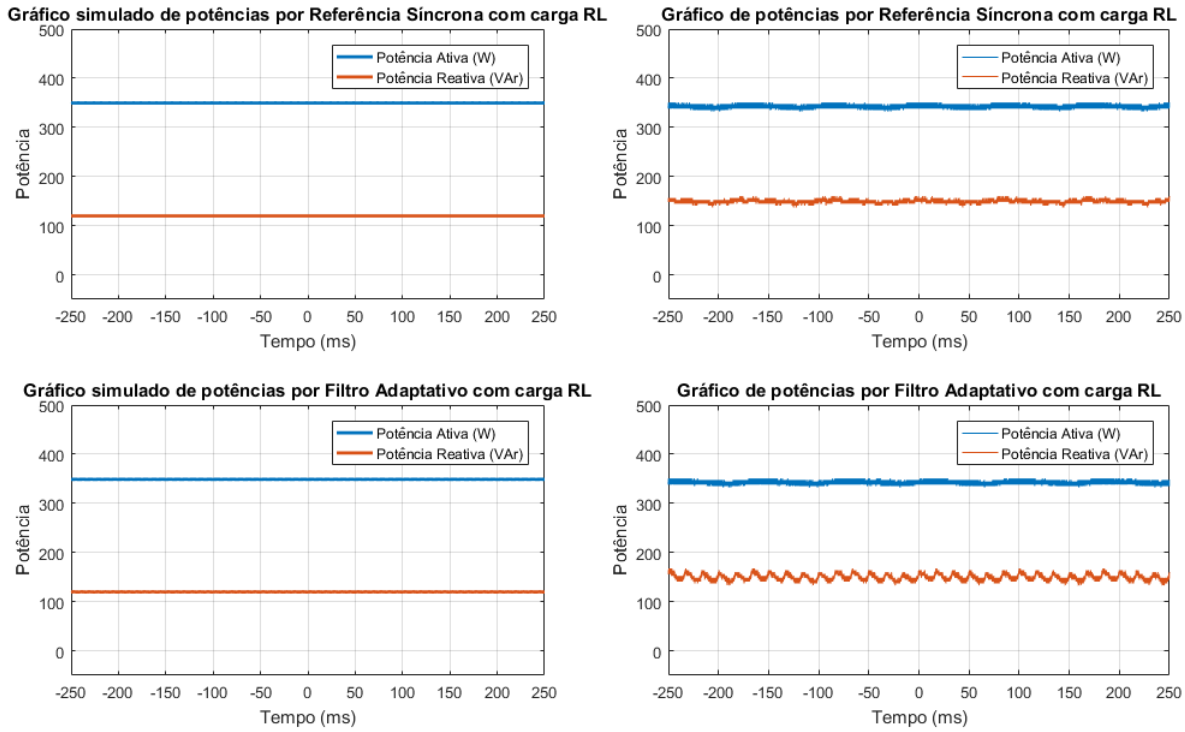
Figura 4.3 – Gráficos de tensão e corrente para carga RL em regime estacionário, obtidos por simulação e em laboratório .



Fonte: Do autor (2021).

A Figura 4.4 apresenta os resultados obtidos de potência ativa e reativa nas mesmas condições, por meio de ambos os algoritmos analisados. Para a Referência Síncrona os valores de potência apresentados foram de $350W$ e $120VAr$ na simulação, e $342W$ e $149VAr$ em laboratório. Por fim para o Filtro Adaptativo foram de $349W$ e $120VAr$ na simulação, e $343W$ e $150VAr$ em laboratório, neste último caso com oscilações de amplitude $15VAr$ no cálculo da potência reativa. A diferença entre os valores obtidos serão discutidas posteriormente.

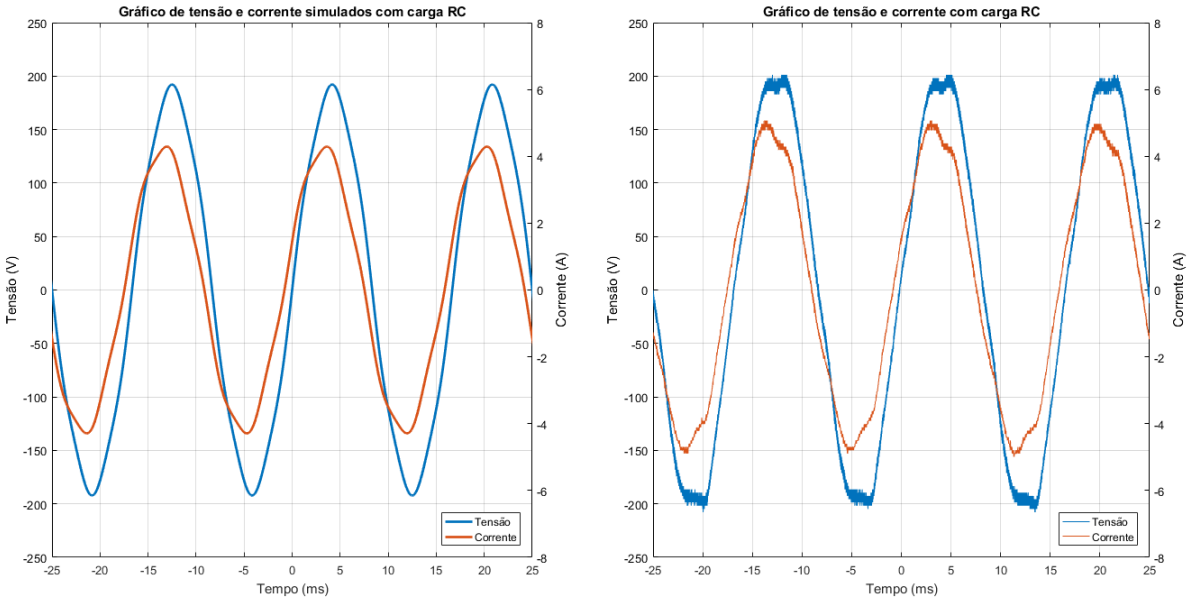
Figura 4.4 – Gráficos de potência ativa e reativa para carga RL em regime estacionário, obtidos por simulação e em laboratório .



Fonte: Do autor (2021).

A Figura 4.5 apresenta os sinais de tensão e corrente dos testes em ambiente simulado e em laboratório com carga RC em regime permanente. Conforme descrito na metodologia, este arranjo possui uma resistência de $41,7 - j14,5\,\Omega$, sendo alimentada por uma tensão com $136\,VRMS$ na frequência fundamental em simulação. Isso resulta em um corrente simulada de $3,0\,ARMS$. Valores similares foram encontrados para tensão e corrente medidos em laboratório que possuem os seguintes valores: $137,2\,VRMS$ e $3,5\,ARMS$.

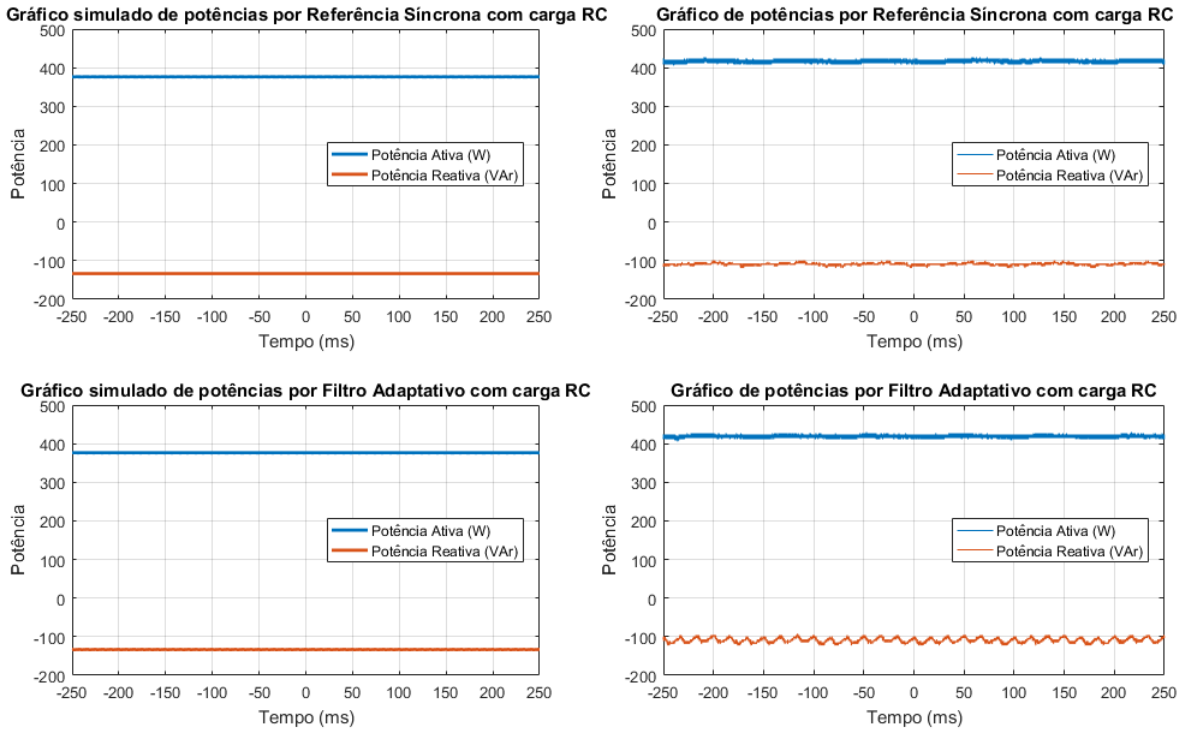
Figura 4.5 – Gráficos de tensão e corrente para carga RC em regime estacionário, obtidos por simulação e em laboratório .



Fonte: Do autor (2021).

A Figura 4.6 apresenta os resultados obtidos de potência ativa e reativa nas mesmas condições, por meio de ambos os algoritmos analisados. Para a Referência Síncrona os valores de potência apresentados foram de $376W$ e $-133VAr$ na simulação, e $416W$ e $-109VAr$ em laboratório. Por fim para o Filtro Adaptativo foram de $377W$ e $-133VAr$ na simulação, e $419W$ e $-109VAr$ em laboratório, neste último caso com oscilações de amplitude $15VAr$ no cálculo da potência reativa. A diferença entre os valores obtidos serão discutidas posteriormente.

Figura 4.6 – Gráficos de potência ativa e reativa para carga RC em regime estacionário, obtidos por simulação e em laboratório .



Fonte: Do autor (2021).

A Tabela 4.1 apresenta os valores de potência medidos através do wattímetro em comparação com os valores médios obtidos pela simulação e pelos testes de laboratório através do algoritmo de Referência Síncrona. São apresentados também os valores de erro da potência calculada pelo DSP em relação àquela medida com o wattímetro.

Tabela 4.1 – Potências calculadas pela Referência Síncrona e medidas pelo wattímetro.

		Simulação	Wattímetro	DSP	Erro (%)
Carga R	Potência Ativa (W)	283	295	300	+1,7
	Potência Reativa (VAr)	0	19	33	+73,7
Carga RL	Potência Ativa (W)	350	340	342	+0,6
	Potência Reativa (VAr)	120	105	149	+41,9
Carga RC	Potência Ativa (W)	376	393	416	+5,8
	Potência Reativa (VAr)	-133	-135	-109	-19,2

Diferentemente da Referência Síncrona e dos valores obtidos por simulação, os resultados obtidos por meio do algoritmo de Filtro Adaptativo deixam evidente a existência de uma oscilação considerável no valor de potência reativa calculado pelo algoritmo de Filtro Adaptativo. Essa oscilação ocorre na mesma frequência que o sinal de tensão, e com amplitude média de aproximadamente 15 VAr. Por esse motivo foram utilizados os valores médios desses sinais

para esta análise. A Tabela 4.2 apresenta uma análise semelhante àquela feita na tabela acima, porém relativa aos resultados obtidos com o algoritmo baseado em Filtro Adaptativo.

Tabela 4.2 – Potências calculadas pelo Filtro Adaptativo e medidas pelo wattímetro.

		Simulação	Wattímetro	DSP	Erro (%)
Carga R	Potência Ativa (W)	282	295	302	+2,4
	Potência Reativa (VAr)	0	19	34	+78,9
Carga RL	Potência Ativa (W)	349	340	343	+0,9
	Potência Reativa (VAr)	120	105	150	+42,8
Carga RC	Potência Ativa (W)	377	393	419	+6,4
	Potência Reativa (VAr)	-133	-135	-109	-19,2

Ao comparar os valores de potência ativa calculados pelos algoritmos com os valores medidos pelo wattímetro é possível observar uma relação de grande proximidade, sendo que a maior diferença entre eles foi no caso da carga capacitiva, com erro de $5,8\%$ no algoritmo de Referência Síncrona, e $6,4\%$ no algoritmo de Filtro Adaptativo. Essa diferença pode ser atribuída a fatores como precisão do wattímetro, calibração imperfeita dos sensores utilizados com o DSP e perda de exatidão dos valores calculados pelo DSP ao realizar a leitura através dos conversores digital-analógicos, entre outros.

Tratando-se dos valores de potência reativa, no entanto, essa diferença se mostra considerável. Isso pode ter acontecido devido a erros tanto no sensor de corrente utilizado em conjunto com o DSP como no próprio Wattímetro. Não foi possível verificar a fonte desses erros com mais detalhes devido à indisponibilidade de instrumentos de medição de corrente e de potência com maior precisão.

Além disso, é possível observar uma diferença entre os valores calculados e os teóricos obtidos por meio de simulação, porém isso já era esperado, uma vez que vários aspectos foram desconsiderados na simulação, como por exemplo faixa de precisão dos componentes utilizados, capacitâncias, indutâncias e resistências indesejadas nos fios e nos componentes, etc.

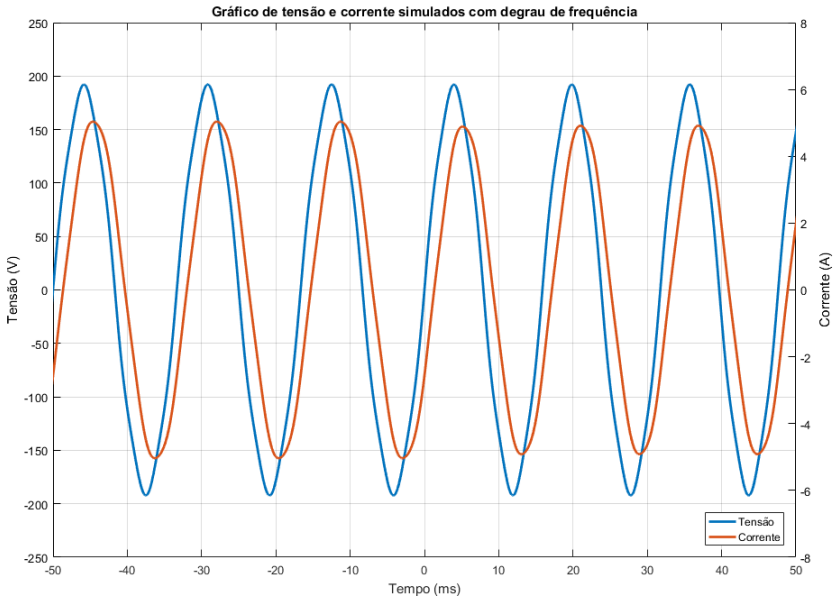
Em seguida serão apresentados os resultados obtidos relativos ao comportamento em regime transiente.

4.2 Regime Transiente

Os gráficos apresentados neste tópico, referentes ao comportamento dos algoritmos em regime transiente, foram obtidos mediante a aplicação de degraus de frequência, carga e tensão. Em todos eles o instante de aplicação do degrau corresponde ao tempo 0 s.

A Figura 4.7 apresenta os sinais de tensão e corrente dos testes em ambiente simulado com degrau de frequência. Conforme descrito na metodologia, o degrau aplicado foi de 60 para $63Hz$, ou seja $376,99$ para $395,84rad/s$. Isso pode ser observado no sinal apresentado pois no período de -50 a 0 ms existem 3 ondas completas de cada sinal, enquanto que no período de 0 a 50 ms existem mais de 3 ondas.

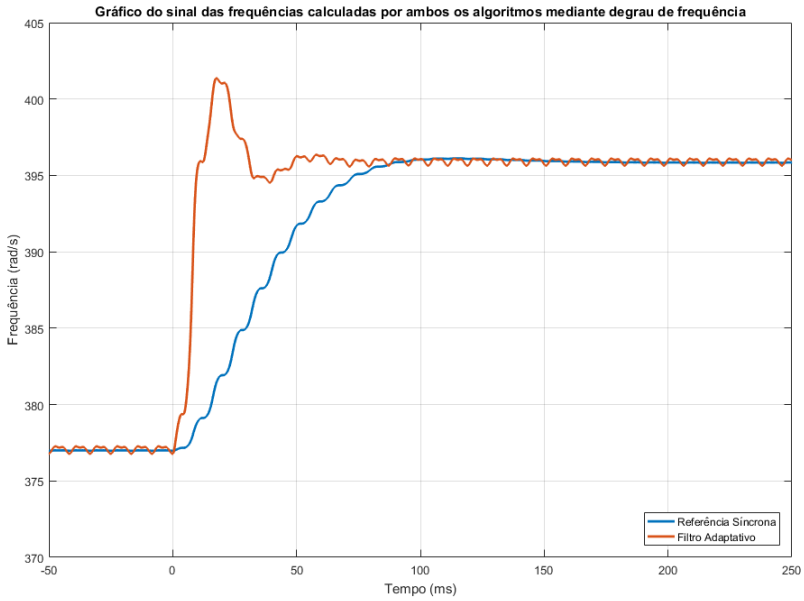
Figura 4.7 – Gráficos de tensão e corrente mediante degrau de frequência, obtidos por simulação.



Fonte: Do autor (2021).

A frequência estimada pelos algoritmos é apresentada na Figura 4.8. Ambos os algoritmos estimam uma frequência de $377rad/s$ antes do degrau aplicado, e convergem para $395,8rad/s$ após o degrau. O algoritmo de Referência Síncrona converge em um período de aproximadamente $82ms$, e o do Filtro Adaptativo em aproximadamente $60ms$, porém com maior ultrapassagem.

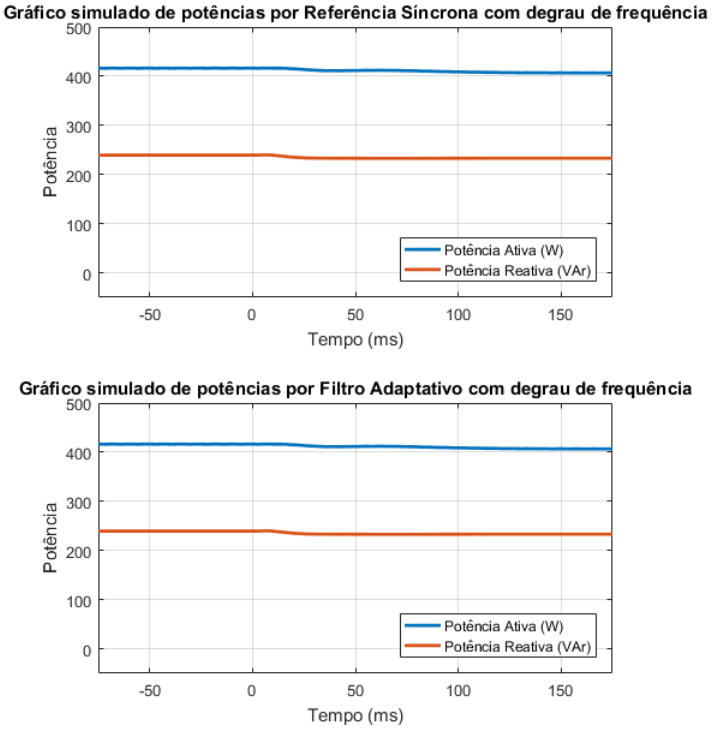
Figura 4.8 – Gráfico das frequências estimadas pelos algoritmos mediante degrau de frequência, obtido por simulação.



Fonte: Do autor (2021).

A Figura 4.9 apresenta os resultados obtidos de potência ativa e reativa nessas condições, e mostram que tanto ambos os algoritmos convergiram em um período menor que 50ms. Como descrito na metodologia esta etapa não foi realizada em laboratório. Além disso o degrau aplicado não causou perturbações nos valores calculados de potência.

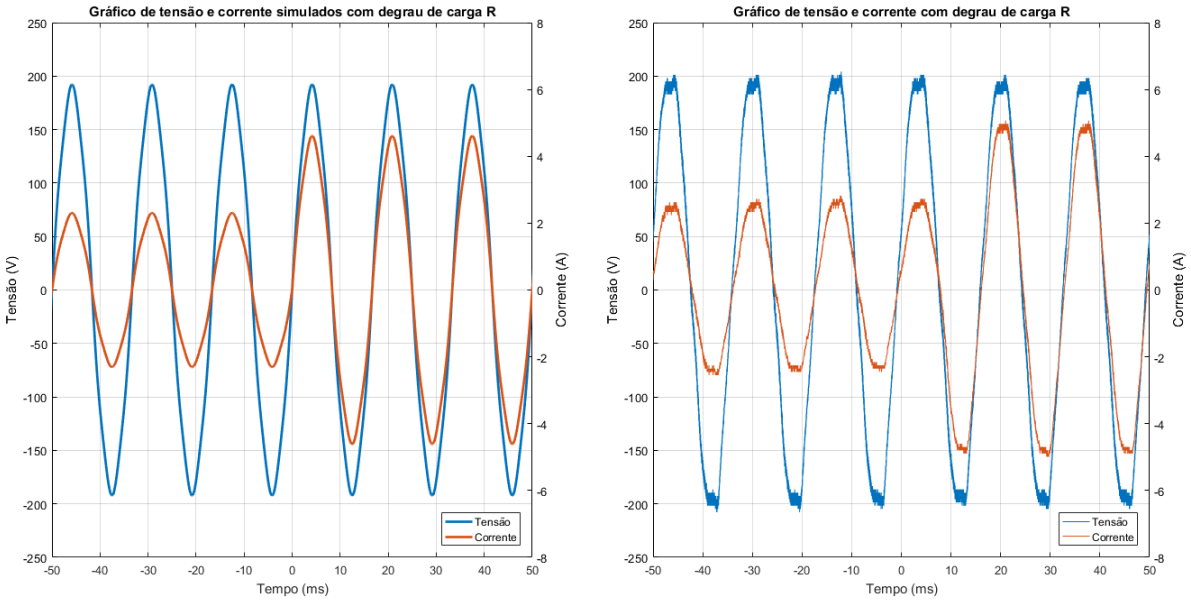
Figura 4.9 – Gráficos de potência ativa e reativa mediante degrau de frequência, obtidos por simulação.



Fonte: Do autor (2021).

A Figura 4.10 apresenta os sinais de tensão e corrente dos testes em ambiente simulado e em laboratório com degrau de carga R. Como descrito na metodologia a carga foi de $Z_{eq} = 83,3\,\Omega$ para $Z_{eq} = 41,7\,\Omega$, fazendo com que a corrente passasse de $1,6\,A\,RMS$ para $3,2\,A\,RMS$ na simulação, e de $1,8\,A\,RMS$ para $3,4\,A\,RMS$.

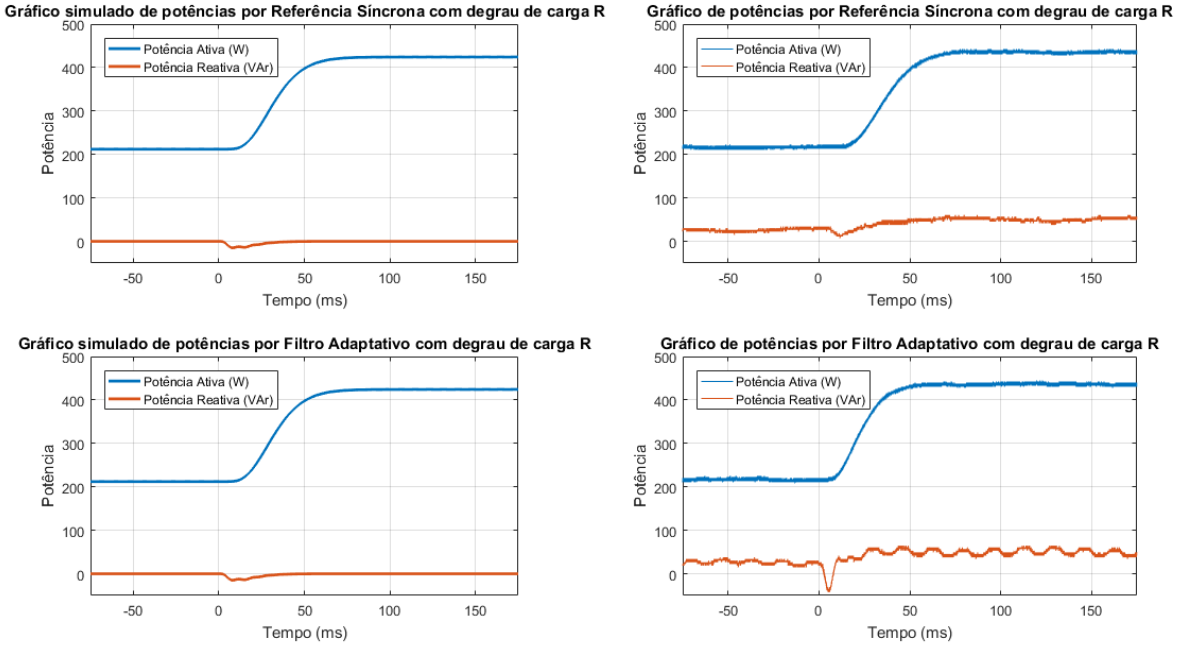
Figura 4.10 – Gráficos de tensão e corrente mediante degrau de carga R, obtidos por simulação e em laboratório .



Fonte: Do autor (2021).

A Figura 4.11 apresenta os resultados obtidos de potência ativa e reativa nas mesmas condições, por meio de ambos os algoritmos analisados. No caso do algoritmo de Referência Síncrona os tempos de convergência das potências ativa e reativa foram de 61*ms* e 50*ms* em simulação, e 65*ms* e 50*ms* em laboratório. Já com o algoritmo de Filtro Adaptativo esses tempos foram 61*ms* e 30*ms* em simulação, e 47*ms* e 24*ms* em laboratório.

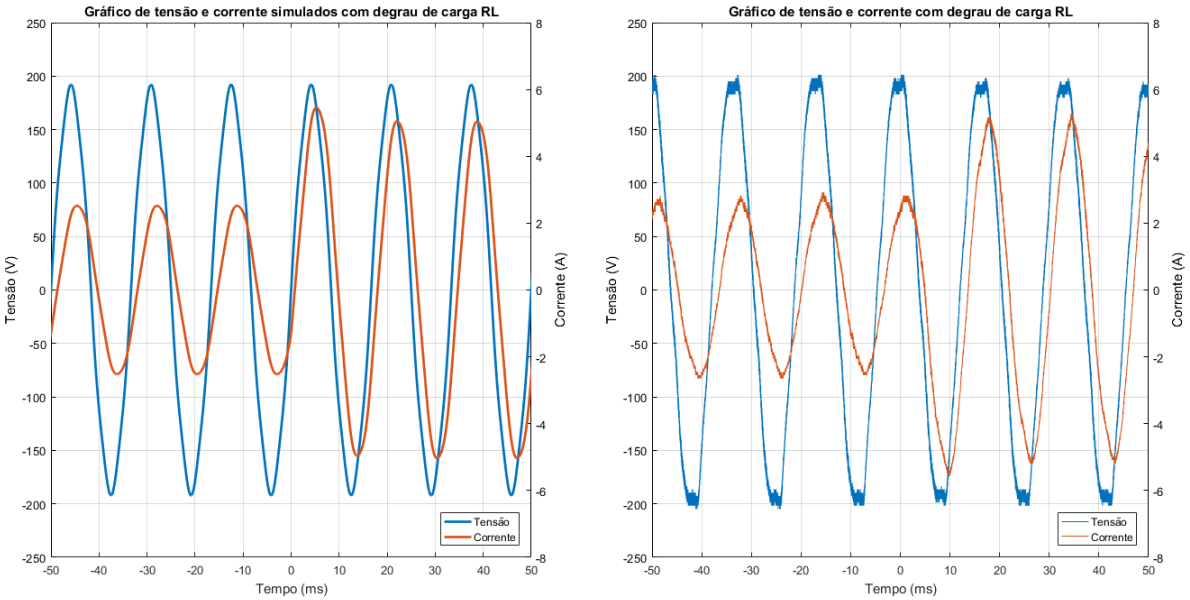
Figura 4.11 – Gráficos de potência ativa e reativa mediante degrau de carga R, obtidos por simulação e em laboratório .



Fonte: Do autor (2021).

A Figura 4.12 apresenta os sinais de tensão e corrente dos testes em ambiente simulado e em laboratório com degrau de carga RL. Como descrito na metodologia a carga foi de $Z_{eq} = 63,9 + j36,9\,\Omega$ para $Z_{eq} = 32 + j18,4\,\Omega$, fazendo com que a corrente passasse de $1,8\,A\,RMS$ para $3,6\,A\,RMS$ na simulação, e de $2,0\,A\,RMS$ para $3,7\,A\,RMS$.

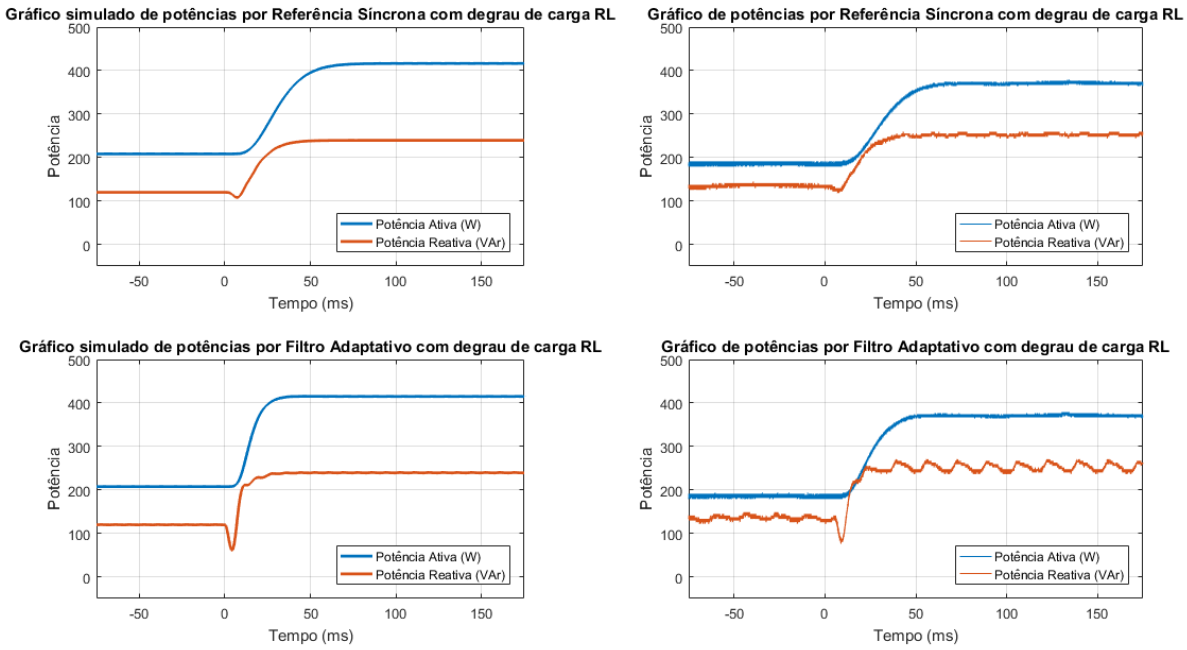
Figura 4.12 – Gráficos de tensão e corrente mediante degrau de carga RL, obtidos por simulação e em laboratório .



Fonte: Do autor (2021).

A Figura 4.13 apresenta os resultados obtidos de potência ativa e reativa nas mesmas condições, por meio de ambos os algoritmos analisados. No caso do algoritmo de Referência Síncrona os tempos de convergência das potências ativa e reativa foram de 59*ms* e 40*ms* em simulação, e 60*ms* e 40*ms* em laboratório. Já com o algoritmo de Filtro Adaptativo esses tempos foram 29*ms* e 26*ms* em simulação, e 44*ms* e 26*ms* em laboratório.

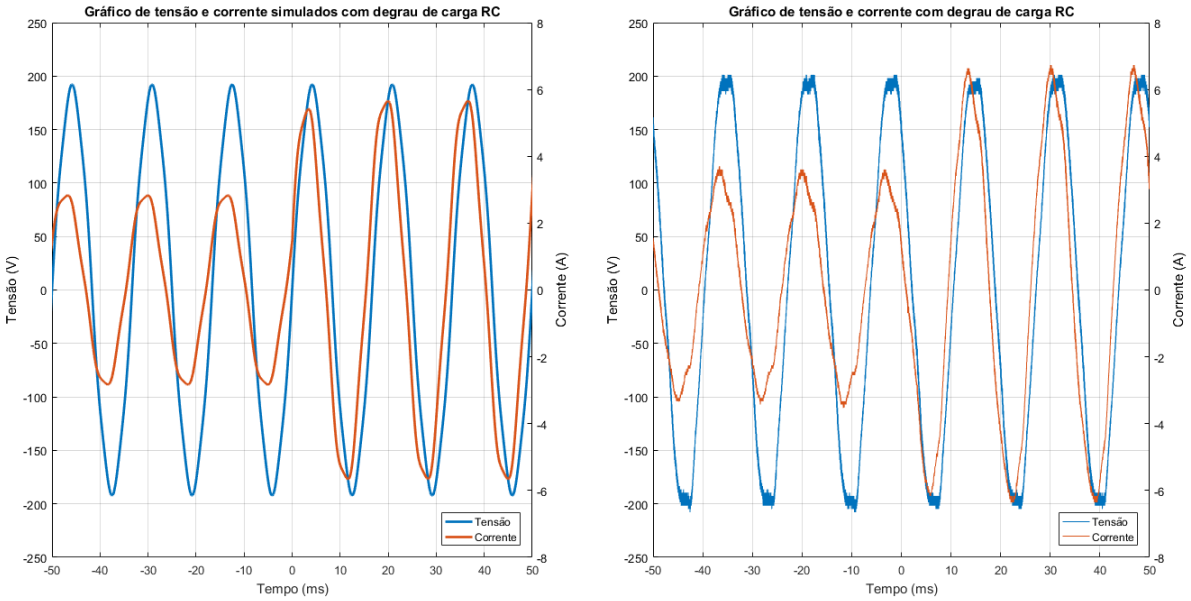
Figura 4.13 – Gráficos de potência ativa e reativa mediante degrau de carga RL, obtidos por simulação e em laboratório .



Fonte: Do autor (2021).

A Figura 4.14 apresenta os sinais de tensão e corrente dos testes em ambiente simulado e em laboratório com degrau de carga RC. Como descrito na metodologia a carga foi de $Z_{eq} = 56,8 - j32,2\,\Omega$ para $Z_{eq} = 28,4 - j16,1\,\Omega$, fazendo com que a corrente passasse de $2,0\,A\,RMS$ para $3,8\,A\,RMS$ na simulação, e de $2,5\,A\,RMS$ para $4,6\,A\,RMS$.

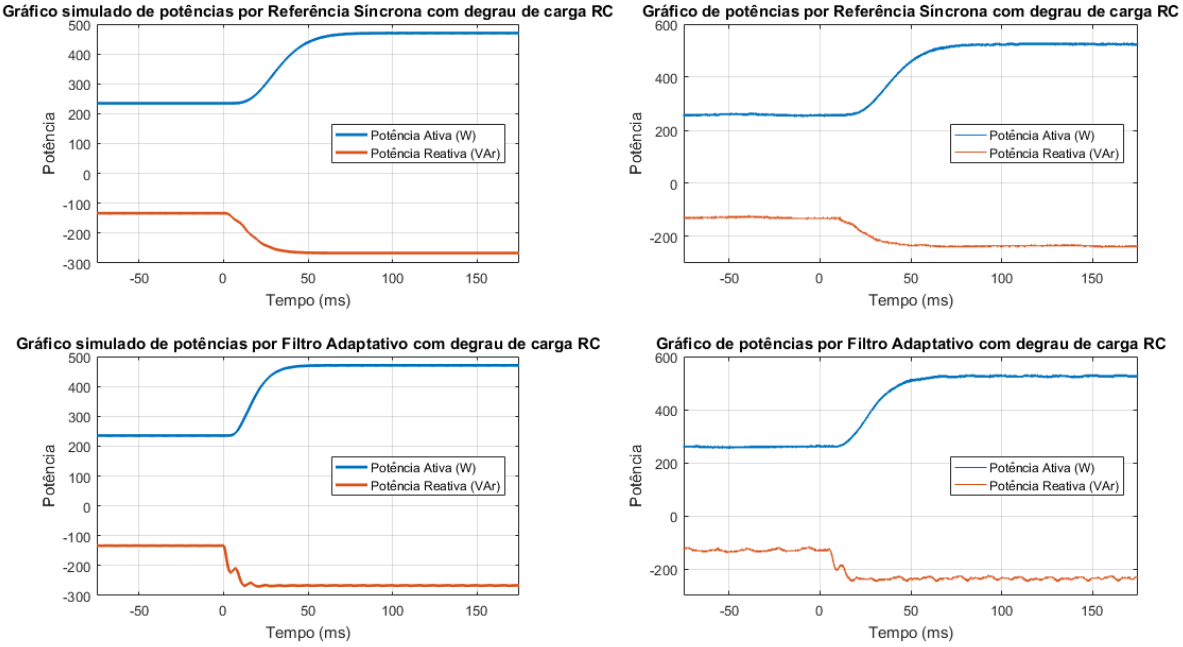
Figura 4.14 – Gráficos de tensão e corrente mediante degrau de carga RC, obtidos por simulação e em laboratório .



Fonte: Do autor (2021).

A Figura 4.15 apresenta os resultados obtidos de potência ativa e reativa nas mesmas condições, por meio de ambos os algoritmos analisados. No caso do algoritmo de Referência Síncrona os tempos de convergência das potências ativa e reativa foram de 62*ms* e 39*ms* em simulação, e 72*ms* e 44*ms* em laboratório. Já com o algoritmo de Filtro Adaptativo esses tempos foram 37*ms* e 17*ms* em simulação, e 54*ms* e 20*ms* em laboratório.

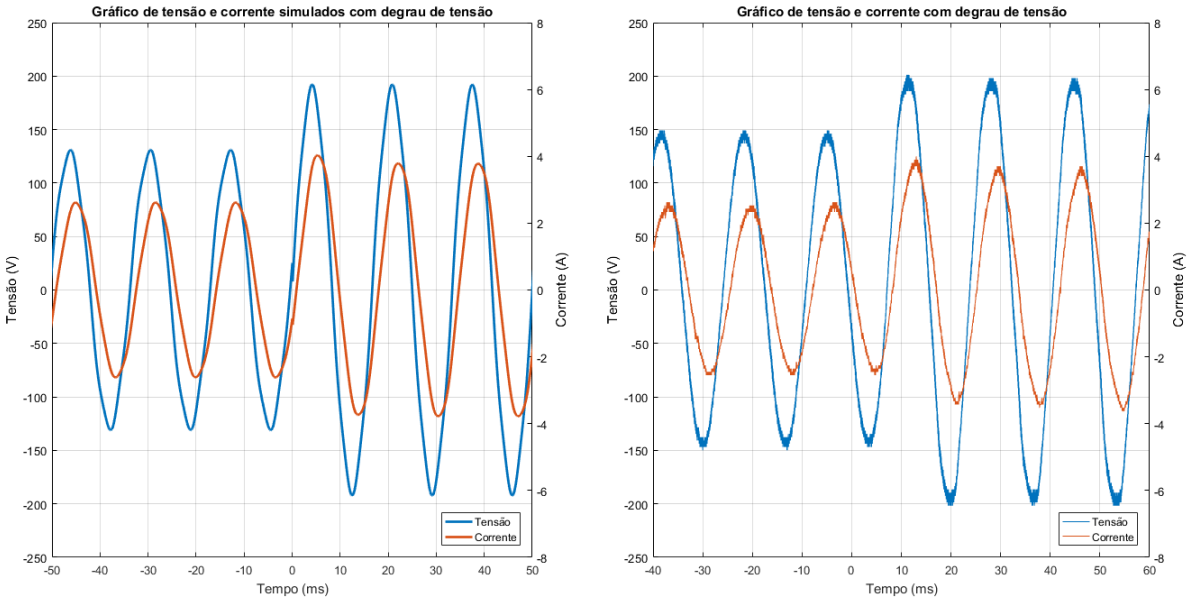
Figura 4.15 – Gráficos de potência ativa e reativa mediante degrau de carga RC, obtidos por simulação e em laboratório .



Fonte: Do autor (2021).

A Figura 4.16 apresenta os sinais de tensão e corrente dos testes em ambiente simulado e em laboratório com degrau de tensão. Como descrito na metodologia a tensão foi de $92,6\,VRMS$ para $135,8\,VRMS$, fazendo com que a corrente passasse de $1,8\,ARMS$ para $2,7\,ARMS$ na simulação, e de $1,8\,ARMS$ para $2,5\,ARMS$.

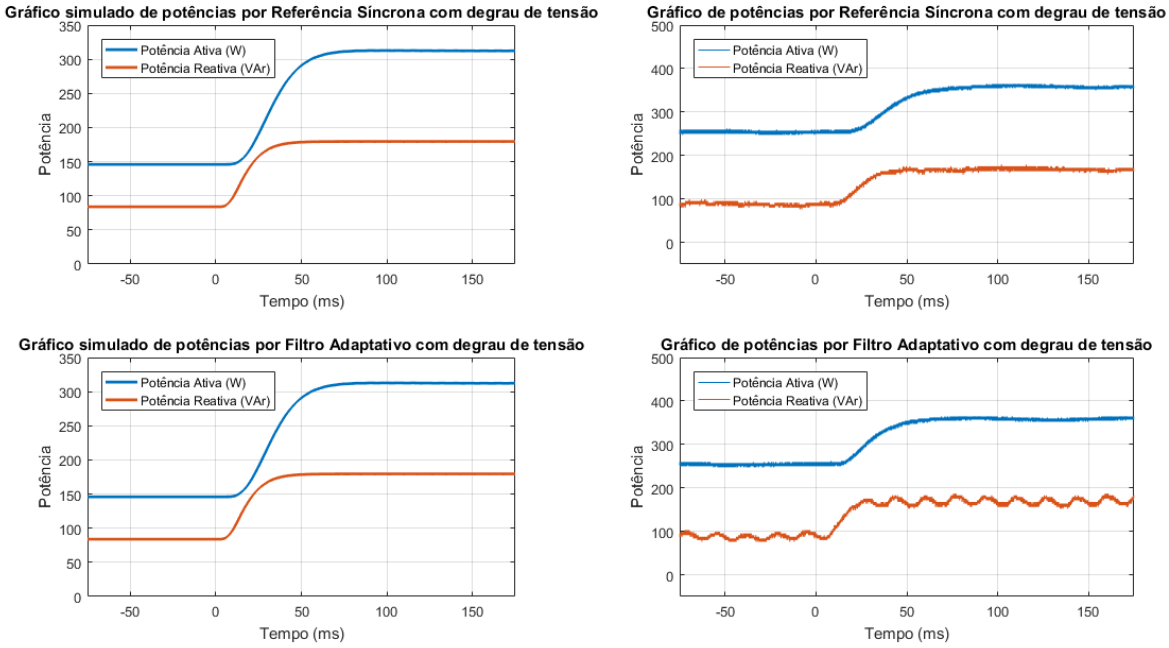
Figura 4.16 – Gráficos de tensão e corrente mediante degrau de tensão, obtidos por simulação e em laboratório .



Fonte: Do autor (2021).

A Figura 4.17 apresenta os resultados obtidos de potência ativa e reativa nas mesmas condições, por meio de ambos os algoritmos analisados. No caso do algoritmo de Referência Síncrona os tempos de convergência das potências ativa e reativa foram de 68*ms* e 39*ms* em simulação, e 71*ms* e 43*ms* em laboratório. Já com o algoritmo de Filtro Adaptativo esses tempos foram 61*ms* e 39*ms* em simulação, e 53*ms* e 22*ms* em laboratório.

Figura 4.17 – Gráficos de potência ativa e reativa mediante degrau de tensão, obtidos por simulação e em laboratório .



Fonte: Do autor (2021).

Observa-se que os resultados obtidos em laboratório apresentam alto grau de proximidade com aqueles obtidos em simulação, sendo que a maior diferença entre eles é a maior amplitude da oscilação com frequência de 60 Hz nos valores de potência reativa calculados pelo algoritmo de Filtro Adaptativo. Os valores calculados de potência ativa não apresentaram oscilações, tampouco máximo pico. Já os valores de potência reativa apresentaram um breve pico inferior no caso dos degraus de cargas resistivas, e indutivas.

As Tabelas 4.3 e 4.4 apresentam um comparativo dos tempos de acomodação para cada sinal de potência, calculados a partir do instante de acionamento do degrau, para os algoritmos baseados em Referência Síncrona e em Filtro Adaptativo.

Tabela 4.3 – Tempo de acomodação das potências calculadas por Referência Síncrona mediante degraus de carga e tensão.

		Simulação (ms)	Laboratório (ms)
Potência ativa	Carga R	61	65
	Carga RL	59	60
	Carga RC	62	72
	Tensão	68	71
Potência reativa	Carga R	50	50
	Carga RL	40	40
	Carga RC	39	44
	Tensão	39	43

Tabela 4.4 – Tempo de acomodação das potências calculadas por Filtro Adaptativo mediante degraus de carga e tensão.

		Simulação (ms)	Laboratório (ms)
Potência ativa	Carga R	61	47
	Carga RL	29	44
	Carga RC	37	54
	Tensão	61	53
Potência reativa	Carga R	30	24
	Carga RL	26	26
	Carga RC	17	20
	Tensão	39	22

É possível perceber que o período compreendido entre o instante de acionamento do degrau e o momento em que o controlador alcança a faixa de 2% do valor final é muito curto em todas as situações. Além disso os dados indicam consistência dos resultados, pois não apresentam grandes variações, e aqueles obtidos em laboratório são muito próximos aos obtidos por meio de simulação.

Observa-se também que o tempo de acomodação para a potência ativa é ligeiramente maior que o da potência reativa, devido à atuação do filtro passa baixas implementado para pós-processamento dos valores calculados.

5 CONCLUSÃO

Este trabalho demonstrou o processo de implementação e a dinâmica de operação de algoritmos utilizados para cálculo de potência demandada ou fornecida por uma rede elétrica de corrente alternada. Foram utilizados dois algoritmos, sendo um baseado em Referência Síncrona e outro em um Filtro Adaptativo sintonizado com estimador de frequência.

Este estudo têm importância primária quando se trata do gerenciamento de microrredes de geração distribuída através do Modelo Droop, pois é necessário obter uma informação precisa e em tempo real do valor de potência fornecida por cada fonte, para que então seja possível controlar esse valor tendo como base um modelo de despacho econômico.

A utilização de algoritmos especializados se torna imprescindível nesse contexto, uma vez que além da variação de potência que mudanças nas cargas podem ocasionar, o Modelo Droop se baseia na variação da frequência de oscilação da rede para comunicação desse valor de potência demandada a cada fonte.

A maior dificuldade encontrada durante o desenvolvimento deste trabalho foi a implementação dos algoritmos no DSP, devido à grande quantidade de parâmetros a serem programados, à sua complexidade e à dificuldade de encontrar informações claras entre as diversas documentações fornecidas pelo fabricante. Foi possível superar essas dificuldades estudando códigos funcionais fornecidos como parte de um workshop sobre a programação de periféricos do DSP, também do fabricante, e replicando fragmentos desses códigos.

Os resultados obtidos indicam que ambos os algoritmos apresentam precisão satisfatória nos valores calculados de potência e boa estabilidade do sinais em regime estacionário, com exceção do valor de potência reativa calculado pelo algoritmo do Filtro Adaptativo, pois apresentou muita oscilação. Pode-se concluir que o algoritmo de Referência Síncrona teve desempenho superior nesse quesito, e que esse problema precisa ser tratado no algoritmo de Filtro Adaptativo antes que possa ser utilizado no gerenciamento de microrredes.

Na análise de comportamento dos algoritmos em regime transiente, os resultados apontam também que ambos foram capazes de controlar os valores de potência de forma que convergissem em um curto período de tempo, o suficiente para que apresentem bom comportamento no gerenciamento de microrredes.

Vale notar, porém, que os valores de potência calculados por Referência Síncrona sofreram menos oscilação mediante os degraus aplicados do que aqueles calculados utilizando Filtro Adaptativo, principalmente undershoot. Já no que se refere ao tempo de resposta a situação

se inverte, pois os valores calculados por Filtro Adaptativo convergiram em menor período de tempo, apresentando assim desempenho superior à Referência Síncrona nesse quesito.

Assim foi possível concluir este trabalho, tendo sido alcançados todos os objetivos propostos. Depreende-se que cada algoritmo apresenta vantagens e desvantagens em relação ao outro, cabendo àquele que irá realizar a implementação do sistema de gerenciamento tomar a decisão de qual utilizar, levando em consideração o projeto como um todo.

Além disso esse estudo pode ser estendido, entre outros, nos seguintes aspectos, a fim de alcançar melhores resultados na implementação do sistema de gerenciamento.

• Realizar um estudo focado no ajuste e refinamento dos parâmetros dos filtros utilizados, a fim de melhorar suas respostas estáticas e dinâmicas.

• Implementação de um método de pós-processamento do sinal de potência reativa calculada por Filtro Adaptativo, tal como um filtro passa baixas, para que apresente menos oscilação.

• Realizar um estudo ampliado considerando outros algoritmos para cálculo de potência em rede de corrente alternada.

• Otimizar os algoritmos implementados para que demandem menor custo computacional, liberando recursos para outros algoritmos necessários no sistema de gerenciamento.

REFERÊNCIAS

ASIMINOAEL, L.; BLAABJERG, F.; HANSEN, S. Detection is key - harmonic detection methods for active power filter applications. **IEEE Industry Applications Magazine**, v. 13, n. 4, p. 22–33, 2007.

BODSON, M.; DOUGLAS, S. C. Adaptive algorithms for the rejection of sinusoidal disturbances with unknown frequency. **IFAC Proceedings Volumes**, v. 29, n. 1, p. 5168 – 5173, 1996. ISSN 1474-6670. 13th World Congress of IFAC, 1996, San Francisco USA, 30 June - 5 July. Disponível em: <http://www.sciencedirect.com/science/article/pii/S1474667017585018>.

Dobrucký, B. et al. Measurement of multi-phase clarke-transformed waveforms using labview virtual instrumentation. In: **2016 International Siberian Conference on Control and Communications (SIBCON)**. [S.l.: s.n.], 2016. p. 1–5.

EYRE, J.; BIER, J. The evolution of dsp processors. **IEEE Signal Processing Magazine**, Citeseer, v. 17, n. 2, p. 43–51, 2000.

FERREIRA, S. C. Aplicação de filtros adaptativos em compensadores híbridos de reativo. In: **Dissertação (Mestrado Acadêmico) - Universidade Federal de Itajubá**. [S.l.: s.n.], 2012.

FERREIRA, S. C. Controle preditivo baseado em modelo na compensação dinâmica do reativo com filtro híbrido. In: **Tese (Doutorado em Engenharia Elétrica) – Universidade Federal de Itajubá**. [S.l.: s.n.], 2016.

FERREIRA, S. C. et al. Adaptive real-time power measurement based on ieee standard 1459-2010. **Electric Power Components and Systems**, Taylor & Francis, v. 43, n. 11, p. 1307–1317, 2015. Disponível em: <https://doi.org/10.1080/15325008.2015.1027425>.

GUIMARãES, R. A. Controle preditivo baseado em modelo para conversores formadores de rede em operação ilhada. In: **Dissertação (Mestrado Acadêmico) - Universidade Federal de Lavras**. [S.l.: s.n.], 2019.

HSU, L.; ORTEGA, R.; DAMM, G. A globally convergent frequency estimator. **IEEE Transactions on Automatic Control**, v. 44, n. 4, p. 698–713, 1999.

INSTRUMENTS, T. **C2000™ F2837xD microcontroller one-day workshop series**. 2016. Disponível em: <https://training.ti.com/c2000-f2837xd-microcontroller-one-day-workshop-series>.

KEYSAN, O. Basics of the field oriented control. In: . [S.l.: s.n.], 2017.

MOJIRI, M.; BAKHSHAI, A. An adaptive notch filter for frequency estimation of a periodic signal. **IEEE Transactions on Automatic Control**, v. 49, n. 2, p. 314–318, 2004.

PENG, F. Z.; AKAGI, H.; NABAE, A. A new approach to harmonic compensation in power systems. In: **Conference Record of the 1988 IEEE Industry Applications Society Annual Meeting**. [S.l.: s.n.], 1988. v. 1, p. 874–880.

RAUTH, S. S.; KUMAR, M.; SRINIVAS, K. A proportional resonant power controller and a combined amplitude adaptive notch filter with pll for better power control and synchronization of single phase on grid solar photovoltaic system. In: **2018 International Conference on Smart Systems and Inventive Technology (ICSSIT)**. [S.l.: s.n.], 2018. p. 378–384.

SILVA, C. et al. A digital pll scheme for three-phase system using modified synchronous reference frame. **Industrial Electronics, IEEE Transactions on**, v. 57, p. 3814 – 3821, 11 2010.

SILVA, M. J. Avaliação de desempenho de algoritmos de sincronismo com a rede elétrica. In: **Dissertação (Mestrado Acadêmico) - Universidade Federal de Lavras**. [S.l.: s.n.], 2019.

SMITH, S. **The Scientist and Engineer's Guide to Digital Signal Processing**. California Technical Pub., 1997. ISBN 9780966017632. Disponível em: <https://books.google.com.br/books?id=rp2VQgAACAAJ>.

SOARES, P. M. O. R. Discretização de controladores contínuos. **Faculdade de Engenharia da Universidade do Porto**, I, n. 1, p. 28–46, out. 1996. Disponível em: <http://hdl.handle.net/10216/11227>.

YAZDANI, D.; BAKHSHAI, A.; JAIN, P. K. A three-phase adaptive notch filter-based approach to harmonic/reactive current extraction and harmonic decomposition. **IEEE Transactions on Power Electronics**, v. 25, n. 4, p. 914–923, 2010.

YIN, G.; GUO, L.; LI, X. An amplitude adaptive notch filter for grid signal processing. **IEEE Transactions on Power Electronics**, v. 28, n. 6, p. 2638–2641, 2013.

APÊNDICE A – Códigos implementados no DSP do algoritmo baseado em Referência Síncrona

Figura 1 – Código principal do programa - Referência Síncrona

```
// FILE:   DSP_Fas_1_cpu01.c

#include "F28x_Project.h"     // Device Header File and Examples Include File

#include "DSP_RefSinc.h"

// Function Prototypes
void ConfigureADC(void);
void ConfigureEPWM(void);
void SetupADCEpwm(void);
interrupt void adca1_isr(void);

extern void InitEPwmMode(void);
extern void InitGPIOMode(void);

void main(void)
{
    // Calcula parametro do filtro passa baixas recursivo
     alpha1 = 1-alpha;

    // Initialize System Control
    InitSysCtrl();
    EALLOW;
    ClkCfgRegs.PERCLKDIVSEL.bit.EPWMCLKDIV = 1;
    EDIS;

    // Initialize GPIO
    InitGpio();                              // Configure default GPIO
    EALLOW;
    GpioCtrlRegs.GPADIR.bit.GPIO31 = 1;      // Drives LED LD2 on controlCARD
    EDIS;
    GpioDataRegs.GPADAT.bit.GPIO31 = 1;      // Turn off LED

    // Clear all interrupts and initialize PIE vector table
    DINT;
    InitPieCtrl();
    IER = 0x0000;
    IFR = 0x0000;
    InitPieVectTable();

    // Map ISR functions
    EALLOW;
    PieVectTable.ADCA1_INT = &adca1_isr;     // Function for ADCA interrupt 1
    EDIS;
```

```
    // Init the output ePWMm
    InitEPwmMode();
    //InitGPIOMode();

    // Configure the ADC and power it up
    ConfigureADC();

    // Configure the ePWM
    ConfigureEPWM();

    // Setup the ADC for ePWM triggered conversions on channel 0
    SetupADCEpwm();

    // Initialize results buffer
    for(graphValueIndex = 0; graphValueIndex < RESULTS_BUFFER_SIZE; graphValue
    {
        graphValue[graphValueIndex] = 0;
    }
    graphValueIndex = 0;

    // Enable PIE interrupt
    PieCtrlRegs.PIEIER1.bit.INTx1 = 1;

    // Enable global interrupts and higher priority real-time debug events
    IER |= M_INT1;          // Enable group 1 interrupts
    EINT;                   // Enable Global interrupt INTM
    ERTM;                   // Enable Global real-time interrupt DBGM

    // Sync ePWM
    EALLOW;
    CpuSysRegs.PCLKCR0.bit.TBCLKSYNC = 1;
    EDIS;

    // Start ePWM
    EPwm3Regs.TBCTL.bit.CTRMODE = 0;
↪ // Un-freeze and enter up-count mode
```

```
    do {
        //GpioDataRegs.GPADAT.bit.GPIO31 = 0;     // Turn on LED
        GpioDataRegs.GPASET.bit.GPIO31 = 1;     // Turn on LED
        DELAY_US(1000 * 500);                   // ON delay
        //GpioDataRegs.GPADAT.bit.GPIO31 = 1;     // Turn off LED
        GpioDataRegs.GPACLEAR.bit.GPIO31 = 1;     // Turn off LED
        DELAY_US(1000 * 500);                   // OFF delay

    } while(1);
}


// Write ADC configurations and power up the ADC for both ADC A and ADC C
void ConfigureADC(void)
{
    EALLOW;

    // ADC-A (Tensao)
    AdcaRegs.ADCCTL2.bit.PRESCALE = 6;
↪ // Set ADCCLK divider to /4
    AdcaRegs.ADCCTL2.bit.RESOLUTION =  0;       // 12-bit resolution
    AdcaRegs.ADCCTL2.bit.SIGNALMODE = 0;
↪ // Single-ended channel conversions (12-bit mode only)
    AdcaRegs.ADCCTL1.bit.INTPULSEPOS = 1;
↪ // Set pulse positions to late
    AdcaRegs.ADCCTL1.bit.ADCPWDNZ = 1;          // Power up the ADC

    // ADC-C (Corrente)
    AdccRegs.ADCCTL2.bit.PRESCALE = 6;
↪ // Set ADCCLK divider to /4
    AdccRegs.ADCCTL2.bit.RESOLUTION =  0;
↪ // 12-bit resolution RESOLUTION_12BIT;
    AdccRegs.ADCCTL2.bit.SIGNALMODE = 0;
↪ // Single-ended channel conversions (12-bit mode only)
    AdccRegs.ADCCTL1.bit.INTPULSEPOS = 1;
↪ // Set pulse positions to late
    AdccRegs.ADCCTL1.bit.ADCPWDNZ = 1;          // Power up the ADC
    EDIS;

    DELAY_US(1000);
↪ // Delay for 1ms to allow ADC time to power up
}
```

```
void ConfigureEPWM(void)
{
    EALLOW;
    // Assumes ePWM clock is already enabled
    EPwm3Regs.TBCTL.bit.CTRMODE = 3;            // Freeze counter
    EPwm3Regs.TBCTL.bit.HSPCLKDIV = 0;
↪ // TBCLK pre-scaler = /1
    EPwm3Regs.TBPRD = 0x137e;
↪ // Set period to 4990 counts (20040 Hz)
    EPwm3Regs.ETSEL.bit.SOCAEN  = 0;
↪ // Disable SOC on A group
    EPwm3Regs.ETSEL.bit.SOCASEL = 2;
↪ // Select SOCA on period match
    EPwm3Regs.ETSEL.bit.SOCAEN = 1;             // Enable SOCA
    EPwm3Regs.ETPS.bit.SOCAPRD = 1;
↪ // Generate pulse on 1st event
    EDIS;
}


void SetupADCEpwm(void)
{
    // Select the channels to convert and end of conversion flag
    EALLOW;
    AdcaRegs.ADCSOC0CTL.bit.CHSEL = 3;
↪ // SOC0 will convert pin A3
    AdcaRegs.ADCSOC0CTL.bit.ACQPS = 14;
↪ // Sample window is 100 SYSCLK cycles
    AdcaRegs.ADCSOC0CTL.bit.TRIGSEL = 9;
↪ // Trigger on ePWM3 SOCA/C
    AdcaRegs.ADCINTSEL1N2.bit.INT1SEL = 0;
↪ // End of SOC0 will set INT1 flag
    AdcaRegs.ADCINTSEL1N2.bit.INT1E = 1;        // Enable INT1 flag
    AdcaRegs.ADCINTFLGCLR.bit.ADCINT1 = 1;
↪ // Make sure INT1 flag is cleared

    // Also setup ADC-C2 in this example
    AdccRegs.ADCSOC0CTL.bit.CHSEL = 2;
↪ // SOC0 will convert pin C2
    AdccRegs.ADCSOC0CTL.bit.ACQPS = 14;
↪ // Sample window is 100 SYSCLK cycles
    AdccRegs.ADCSOC0CTL.bit.TRIGSEL = 9;
↪ // Trigger on ePWM2 SOCA/C
    EDIS;
}
```

```
interrupt void adca1_isr(void)
{
    // Read the ADC result and store in circular buffer
    input();

    // Estima a frequencia e fase da tensao de entrada
    pll();

    // Estima a potencia
    potencia();

    // Calcula e grava o duty cycle nos registradores dos PWMs de saida para D
    dac();

    // Controla status dos reles
    rele();

    // Armazena uma variavel selecionada em um buffer para ser exibido em graf
    grafico();

    // Return from interrupt
    AdcaRegs.ADCINTFLGCLR.bit.ADCINT1 = 1;
↪ // Clear ADC INT1 flag
    PieCtrlRegs.PIEACK.all = PIEACK_GROUP1;
↪ // Acknowledge PIE group 1 to enable further interrupts
}
 // end of file
```

Fonte: Do autor (2021)

Figura 2 – Código de configuração dos PWMs

```
/**********************************************************************
* File: DSP_EPwm.c
* Devices: TMS320F28x7x
* Author: Thales Roger, Silvia Ferreira, UFLA, 2020
**********************************************************************/

#include "F2837xD_device.h"
#include "F2837xD_Examples.h"

/**********************************************************************
* Function: InitEPwmMode()
*
* Description: Initializes the Enhanced PWM modules on the F28x7x for the IGBT
**********************************************************************/
void InitEPwmMode(void)
{
    asm(" EALLOW");                 // Enable EALLOW protected register access

   // Configure the prescaler to the ePWM modules.
↪ Max ePWM input clock is 100 MHz.
   ClkCfgRegs.PERCLKDIVSEL.bit.EPWMCLKDIV = 1;     // EPWMCLK divider from PLL
↪ 0=/1, 1=/2

   // Must disable the clock to the ePWM modules if you want all ePWM modules
   CpuSysRegs.PCLKCR0.bit.TBCLKSYNC = 0;

   asm(" EDIS");                   // Disable EALLOW protected register access


//--------------------------------------------------------------------------
//--- Configure GPIOs 157, 158, 159, 160 for EPwm output
//--------------------------------------------------------------------------
   EALLOW;

    //
    // Disable internal pull-up for the selected output pins
    // for reduced power consumption
    //
```

```
    GpioCtrlRegs.GPEPUD.bit.GPIO157 = 1;    // Disable pull-up on GPIO157 (EPW
    GpioCtrlRegs.GPEPUD.bit.GPIO158 = 1;    // Disable pull-up on GPIO158 (EPW
    GpioCtrlRegs.GPEPUD.bit.GPIO159 = 1;    // Disable pull-up on GPIO159 (EPW
    GpioCtrlRegs.GPFPUD.bit.GPIO160 = 1;    // Disable pull-up on GPIO160 (EPW

    //
    // Configure EPWM-1, EPWM-2 and EPWM-7 and EPWM-8 pins using GPIO regs
    // This specifies which of the possible GPIO pins will be EPWM functional
    // pins.
    //

    GpioCtrlRegs.GPEMUX2.bit.GPIO157 = 1;   // Configure GPIO157 as EPWM8A
    GpioCtrlRegs.GPEMUX2.bit.GPIO158 = 1;   // Configure GPIO158 as EPWM7B
    GpioCtrlRegs.GPEMUX2.bit.GPIO159 = 1;   // Configure GPIO159 as EPWM8A
    GpioCtrlRegs.GPFMUX1.bit.GPIO160 = 1;   // Configure GPIO160 as EPWM8B

    EDIS;

//-----------------------------------------------------------------------
//--- Configure ePWM7 for 250 kHz asymmetric PWM on EPWM7A and EPWM7B pins
//-----------------------------------------------------------------------
    asm(" EALLOW");                         // Enable EALLOW protected registe
    DevCfgRegs.SOFTPRES2.bit.EPWM7 = 1;     // ePWM8 is reset
    DevCfgRegs.SOFTPRES2.bit.EPWM7 = 0;     // ePWM8 is released from reset
    asm(" EDIS");                           // Disable EALLOW protected regist

    EPwm7Regs.TBCTL.bit.CTRMODE = 0x3;      // Disable the timer

    EPwm7Regs.TBCTL.all = 0xC033;           // Configure timer control registe
// bit 15-14     11:     FREE/SOFT, 11 = ignore emulation suspend
// bit 13        0:      PHSDIR, 0 = count down after sync event
// bit 12-10     000:    CLKDIV, 000 => TBCLK = HSPCLK/1
// bit 9-7       000:    HSPCLKDIV,     000 => HSPCLK = EPWMCLK/1
// bit 6         0:      SWFSYNC, 0 = no software sync produced
// bit 5-4       11:     SYNCOSEL, 11 = sync-out disabled
// bit 3         0:      PRDLD, 0 = reload PRD on counter=0
// bit 2         0:      PHSEN, 0 = phase control disabled
// bit 1-0       11:     CTRMODE, 11 = timer stopped (disabled)
```

```
    EPwm7Regs.TBCTR = 0x0000;           // Clear timer counter
    EPwm7Regs.TBPRD = 0x00C8;           // Set timer period to 500 counts (100
    EPwm7Regs.TBPHS.bit.TBPHS = 0x0000; // Set timer phase

    EPwm7Regs.CMPA.bit.CMPA = 0;     // Set PWM duty cycle 0%
    EPwm7Regs.CMPB.bit.CMPB = 0;     // Set PWM duty cycle 0%

    EPwm7Regs.CMPCTL.all = 0x000A;      // Compare control register
// bit 15-10     0's:    reserved
// bit 9         0:      SHDWBFULL, read-only
// bit 8         0:      SHDWAFULL, read-only
// bit 7         0:      reserved
// bit 6         0:      SHDWBMODE, 0 = shadow mode
// bit 5         0:      reserved
// bit 4         0:      SHDWAMODE, 0 = shadow mode
// bit 3-2       10:     LOADBMODE, 10 = load on zero or PRD match
// bit 1-0       10:     LOADAMODE, 10 = load on zero or PRD match

    EPwm7Regs.AQCTLA.all = 0x0060;      // Action-qualifier control register A
 // bit 15-12     0000:   reserved
 // bit 11-10     00:     CBD, 00 = do nothing
 // bit 9-8       00:     CBU, 00 = do nothing
 // bit 7-6       01:     CAD, 01 = clear
 // bit 5-4       10:     CAU, 10 = set
 // bit 3-2       00:     PRD, 00 = do nothing
 // bit 1-0       00:     ZRO, 00 = do nothing

     EPwm7Regs.AQCTLB.all = 0x0600;      // Action-qualifier control register
 // bit 15-12     0000:   reserved
 // bit 11-10     01:     CBD, 01 = clear
 // bit 9-8       10:     CBU, 10 = set
 // bit 7-6       00:     CAD, 00 = do nothing
 // bit 5-4       00:     CAU, 00 = do nothing
 // bit 3-2       00:     PRD, 00 = do nothing
 // bit 1-0       00:     ZRO, 00 = do nothing
```

```
    EPwm7Regs.AQSFRC.all = 0x0000;      // Action-qualifier s/w force register
// bit 15-8      0's:    reserved
// bit 7-6       00:     RLDCSF, 00 = reload AQCSFRC on zero
// bit 5         0:      OTSFB, 0 = do not initiate a s/w forced event on outp
// bit 4-3       00:     ACTSFB, don't care
// bit 2         0:      OTSFA, 0 = do not initiate a s/w forced event on outp
// bit 1-0       00:     ACTSFA, don't care

    EPwm7Regs.AQCSFRC.all = 0x0000;     // Action-qualifier continuous s/w for
// bit 15-4      0's:    reserved
// bit 3-2       00:     CSFB, 00 = forcing disabled
// bit 1-0       00:     CSFA, 00 = forcing disabled

    EPwm7Regs.DBCTL.bit.OUT_MODE = 0;   // Deadband disabled
    EPwm7Regs.PCCTL.bit.CHPEN = 0;      // PWM chopper unit disabled
    EPwm7Regs.TZDCSEL.all = 0x0000;     // All trip zone and DC compare action

    EPwm7Regs.TBCTL.bit.CTRMODE = 0x2;  // Enable the timer in up-down count m

//-----------------------------------------------------------------------
//--- Configure ePWM8 for 250 kHz asymmetric PWM on EPWM8A and EPWM8B pins
//-----------------------------------------------------------------------
    asm(" EALLOW");                             // Enable EALLOW protected registe
    DevCfgRegs.SOFTPRES2.bit.EPWM8 = 1;     // ePWM8 is reset
    DevCfgRegs.SOFTPRES2.bit.EPWM8 = 0;     // ePWM8 is released from reset
    asm(" EDIS");                               // Disable EALLOW protected regist

    EPwm8Regs.TBCTL.bit.CTRMODE = 0x3;      // Disable the timer
```

```
    EPwm8Regs.TBCTL.all = 0xC033;           // Configure timer control registe
// bit 15-14     11:     FREE/SOFT, 11 = ignore emulation suspend
// bit 13        0:      PHSDIR, 0 = count down after sync event
// bit 12-10     000:    CLKDIV, 000 => TBCLK = HSPCLK/1
// bit 9-7       000:    HSPCLKDIV,     000 => HSPCLK = EPWMCLK/1
// bit 6         0:      SWFSYNC, 0 = no software sync produced
// bit 5-4       11:     SYNCOSEL, 11 = sync-out disabled
// bit 3         0:      PRDLD, 0 = reload PRD on counter=0
// bit 2         0:      PHSEN, 0 = phase control disabled
// bit 1-0       11:     CTRMODE, 11 = timer stopped (disabled)

    EPwm8Regs.TBCTR = 0x0000;           // Clear timer counter
    EPwm8Regs.TBPRD = 0x00C8;           // Set timer period to 500 counts (100
    EPwm8Regs.TBPHS.bit.TBPHS = 0x0000; // Set timer phase

    EPwm8Regs.CMPA.bit.CMPA = 0;     // Set PWM duty cycle 0%
    EPwm8Regs.CMPB.bit.CMPB = 0;     // Set PWM duty cycle 0%

    EPwm8Regs.CMPCTL.all = 0x000A;      // Compare control register
// bit 15-10     0's:    reserved
// bit 9         0:      SHDWBFULL, read-only
// bit 8         0:      SHDWAFULL, read-only
// bit 7         0:      reserved
// bit 6         0:      SHDWBMODE, 0 = shadow mode
// bit 5         0:      reserved
// bit 4         0:      SHDWAMODE, 0 = shadow mode
// bit 3-2       10:     LOADBMODE, 10 = load on zero or PRD match
// bit 1-0       10:     LOADAMODE, 10 = load on zero or PRD match

    EPwm8Regs.AQCTLA.all = 0x0060;      // Action-qualifier control register A
 // bit 15-12     0000:   reserved
 // bit 11-10     00:     CBD, 00 = do nothing
 // bit 9-8       00:     CBU, 00 = do nothing
 // bit 7-6       01:     CAD, 01 = clear
 // bit 5-4       10:     CAU, 10 = set
 // bit 3-2       00:     PRD, 00 = do nothing
 // bit 1-0       00:     ZRO, 00 = do nothing
```

```
     EPwm8Regs.AQCTLB.all = 0x0600;      // Action-qualifier control register
 // bit 15-12     0000:   reserved
 // bit 11-10     01:     CBD, 01 = clear
 // bit 9-8       10:     CBU, 10 = set
 // bit 7-6       00:     CAD, 00 = do nothing
 // bit 5-4       00:     CAU, 00 = do nothing
 // bit 3-2       00:     PRD, 00 = do nothing
 // bit 1-0       00:     ZRO, 00 = do nothing

    EPwm8Regs.AQSFRC.all = 0x0000;      // Action-qualifier s/w force register
// bit 15-8      0's:    reserved
// bit 7-6       00:     RLDCSF, 00 = reload AQCSFRC on zero
// bit 5         0:      OTSFB, 0 = do not initiate a s/w forced event on outp
// bit 4-3       00:     ACTSFB, don't care
// bit 2         0:      OTSFA, 0 = do not initiate a s/w forced event on outp
// bit 1-0       00:     ACTSFA, don't care

    EPwm8Regs.AQCSFRC.all = 0x0000;     // Action-qualifier continuous s/w for
// bit 15-4      0's:    reserved
// bit 3-2       00:     CSFB, 00 = forcing disabled
// bit 1-0       00:     CSFA, 00 = forcing disabled

    EPwm8Regs.DBCTL.bit.OUT_MODE = 0;   // Deadband disabled
    EPwm8Regs.PCCTL.bit.CHPEN = 0;      // PWM chopper unit disabled
    EPwm8Regs.TZDCSEL.all = 0x0000;     // All trip zone and DC compare action

    EPwm8Regs.TBCTL.bit.CTRMODE = 0x2;  // Enable the timer in up-down count m


//-----------------------------------------------------------------------
//--- Enable the clocks to the ePWM module.
//--- Note: this should be done after all ePWM modules are configured
//--- to ensure synchronization between the ePWM modules.
//-----------------------------------------------------------------------
   asm(" EALLOW");                      // Enable EALLOW protected register acc
   CpuSysRegs.PCLKCR0.bit.TBCLKSYNC = 1;  // TBCLK to ePWM modules enabled
   asm(" EDIS");                       // Disable EALLOW protected register acces
```

```
   // Initialize GPIO for relay control
    EALLOW;
    GpioCtrlRegs.GPCDIR.bit.GPIO95 = 1;     // Drives LED LD2 on controlCARD
    EDIS;
    GpioDataRegs.GPCDAT.bit.GPIO95 = 0;     // Turn off output

} // end InitEPwmMode()

/*************************************************************************
* Function: InitGPIOMode()
*
* Description: Initializes the outputs as GPIO fot the IGBT control
*************************************************************************/

void InitGPIOMode(void)
{
    // Set GPIO pins as output
    EALLOW;
    GpioCtrlRegs.GPADIR.bit.GPIO0 = 1;
    GpioCtrlRegs.GPADIR.bit.GPIO1 = 1;
    GpioCtrlRegs.GPADIR.bit.GPIO2 = 1;
    GpioCtrlRegs.GPADIR.bit.GPIO3 = 1;
    EDIS;

    // Set GPIOs to low state
    GpioDataRegs.GPADAT.bit.GPIO0 = 0;
    GpioDataRegs.GPADAT.bit.GPIO1 = 0;
    GpioDataRegs.GPADAT.bit.GPIO2 = 0;
    GpioDataRegs.GPADAT.bit.GPIO3 = 0;

} // end InitGPIOMode()

//--- end of file -----------------------------------------------------
```

Fonte: Do autor (2021)

Figura 3 – Código contendo as funções que implementam o algoritmo baseado em Referência Síncrona

```
/**********************************************************************
* File: DSP_FAS_estimador.c
* Devices: TMS320F28x7x
* Author: Thales Roger, Silvia Ferreira, UFLA, 2020
**********************************************************************/

#include "F2837xD_device.h"
#include "F2837xD_Examples.h"
#include <math.h>

// Defines
#define T 4.99e-05
#define GAMA 4000
#define ZETA 0.19
#define ZETA2 0.3

// Variables
#define RESULTS_BUFFER_SIZE 334
float32 graphValue[RESULTS_BUFFER_SIZE];
Uint16 graphValueIndex;
Uint16 selectGraph = 2;

float32 Vin, Iin, Vd, Vq, Id, Iq, P, Q;

float32 w1=376.991118431;
float32 x1=0.0, xd1=0.0, x5=0.0, xd5=0.0, e1=0.0;

float32 wt=0.0, vd=0.0, vq=0.0, id = 0.0, iq = 0.0, pp = 0.0;

float32 alpha = 0.99125951528549194;
float32 alpha1;

float32 Vd1 = 0.0, Vd2 = 0.0, Vd3 = 0.0;
float32 Vq1 = 0.0, Vq2 = 0.0, Vq3 = 0.0;
float32 Id1 = 0.0, Id2 = 0.0, Id3 = 0.0;
float32 Iq1 = 0.0, Iq2 = 0.0, Iq3 = 0.0;
float32 P1 = 0.0, P2 = 0.0, P3 = 0.0;
```

```
float32 dac1duty = 0.0;
float32 dac2duty = 0.0;
float32 dac3duty = 0.0;
float32 dac4duty = 0.0;
float32 dac1max = 0.002;  // = 1/500 Potencia Ativa -
↪ Escala 0 a 100% de duty = 0 a 500 W
//float32 dac1max = 0.0025;  // = 1/400 Potencia Ativa -
↪ Escala 0 a 100% de duty = 150 a 550 W
float32 dac2max = 0.0016; // = 1/625 Potencia Reativa - Escala 0 a 100% de dut
float32 dac3max = 0.002;  // = 1/500 Tensao -
↪ Escala 0 a 100% de duty = -250 a 250 V
float32 dac4max = 0.0625; // = 1/16  Corrente -
↪ Escala 0 a 100% de duty = -8 a 8 A

bool rele1;

/**************************************************************************
* Function: input()
*
* Description: Read the ADC result and store in circular buffer
**************************************************************************/
void input(void)
{
    //Vin = 0.000759781*((float32)AdcaResultRegs.ADCRESULT0); // Valor direto
    Vin = 0.204075192*((float32)AdcaResultRegs.ADCRESULT0 - 2305.1); // Valor
    //Iin = 0.000754128*((float32)AdccResultRegs.ADCRESULT0); // Valor direto
    Iin = 0.01002206*((float32)AdccResultRegs.ADCRESULT0 - 2337.6); // Valor d

} // end input()


/**************************************************************************
* Function: pll()
*
* Description: Estima a frequencia e fase da tensao de entrada
**************************************************************************/
extern void pll(void)
{
    //----------PLL
    w1 = w1 - T*GAMA*x1*w1*e1;

    xd1 = xd1 + T*(2*ZETA*w1*e1 - w1*w1*x1);
    x1 = x1 + T*xd1;
```

```
    xd5 = xd5 + T*(2*ZETA2*w1*e1 - 25*w1*w1*x5);
    x5 = x5 + T*xd5;

    e1 = Vin/180 - xd1 - xd5;

    wt = atan2(xd1,(-w1*x1));

} // end pll()

/*************************************************************************
* Function: potencia()
*
* Description: Estima a potencia
*************************************************************************/
void potencia(void)
{
    //---------- TENSAO
    //Park
    vd = 2*Vin*sin(wt);
    vq = -2*Vin*cos(wt);

    //Filtro LPF3
    Vd1 = alpha*Vd1 + alpha1*vd;
    Vd2 = alpha*Vd2 + alpha1*Vd1;
    Vd3 = alpha*Vd3 + alpha1*Vd2;
    Vd = Vd3;

    Vq1 = alpha*Vq1 + alpha1*vq;
    Vq2 = alpha*Vq2 + alpha1*Vq1;
    Vq3 = alpha*Vq3 + alpha1*Vq2;
    Vq = Vq3;


    //----------CORRENTE
    //Park
    id = 2*Iin*sin(wt);
    iq = -2*Iin*cos(wt);
```

```
    //Filtro LPF3
    Id1 = alpha*Id1 + alpha1*id;
    Id2 = alpha*Id2 + alpha1*Id1;
    Id3 = alpha*Id3 + alpha1*Id2;
    Id = Id3;

    Iq1 = alpha*Iq1 + alpha1*iq;
    Iq2 = alpha*Iq2 + alpha1*Iq1;
    Iq3 = alpha*Iq3 + alpha1*Iq2;
    Iq = Iq3;


    //----------POTENCIAS
    pp = (Vd*Id+Vq*Iq)/2;

    P1 = alpha*P1 + alpha1*pp;
    P2 = alpha*P2 + alpha1*P1;
    P3 = alpha*P3 + alpha1*P2;
    P = P3;

    Q = (Vd*Iq-Vq*Id)/2;

} //end potencia()


/*************************************************************************
* Function: dac()
*
* Description: Calcula e grava o duty cycle nos registradores dos PWMs de said
*************************************************************************/
void dac(void)
{
    dac1duty = P*dac1max;
    //dac1duty = (P-150)*dac1max;
    dac2duty = (Q+312.5)*dac2max;
    dac3duty = (Vin+250)*dac3max;
    dac4duty = (Iin+8)*dac4max;
```

```
    //dac1duty = 0.0;
    //dac2duty = 0.5;
    //dac3duty = 0.5;
    //dac4duty = 1;

    EPwm8Regs.CMPA.bit.CMPA = (1-dac1duty)*200;
↪ // Set PWM duty cycle
    EPwm8Regs.CMPB.bit.CMPB = (1-dac2duty)*200;
↪ // Set PWM duty cycle
    EPwm7Regs.CMPA.bit.CMPA = (1-dac3duty)*200;
↪ // Set PWM duty cycle
    EPwm7Regs.CMPB.bit.CMPB = (1-dac4duty)*200;
↪ // Set PWM duty cycle

} // end dac()


/*************************************************************************
* Function: rele()
*
* Description: Controla status dos GPIOs dos reles
*************************************************************************/
extern void rele(void){
    if (rele1 == 1){
        GpioDataRegs.GPCTOGGLE.bit.GPIO95 = 1;  // Toggle LED
        rele1 = 0;
    }

} // end rele()


/*************************************************************************
* Function: grafico()
*
* Description: Armazena uma variavel selecionada em um buffer para ser exibido
*************************************************************************/
void grafico(void)
{
    // GRAPH
    switch(selectGraph){
        case 0:
            graphValue[graphValueIndex++] = 0;
            break;
```

```
case 1:
    graphValue[graphValueIndex++] = Vin;
    break;

case 2:
    graphValue[graphValueIndex++] = Iin;
    break;

case 3:
    graphValue[graphValueIndex++] = w1;
    break;

case 4:
    graphValue[graphValueIndex++] = xd1;
    break;

case 5:
    graphValue[graphValueIndex++] = x1;
    break;

case 6:
    graphValue[graphValueIndex++] = xd5;
    break;

case 7:
    graphValue[graphValueIndex++] = x5;
    break;

case 8:
    graphValue[graphValueIndex++] = e1;
    break;

case 9:
    graphValue[graphValueIndex++] = wt;
    break;

case 10:
    graphValue[graphValueIndex++] = vd;
    break;
```

```
case 11:
    graphValue[graphValueIndex++] = Vd;
    break;

case 12:
    graphValue[graphValueIndex++] = vq;
    break;

case 13:
    graphValue[graphValueIndex++] = Vq;
    break;

case 14:
    graphValue[graphValueIndex++] = id;
    break;

case 15:
    graphValue[graphValueIndex++] = Id;
    break;

case 16:
    graphValue[graphValueIndex++] = iq;
    break;

case 17:
    graphValue[graphValueIndex++] = Iq;
    break;
```

```
        case 18:
            graphValue[graphValueIndex++] = pp;
            break;

        case 19:
            graphValue[graphValueIndex++] = P;
            break;

        case 20:
            graphValue[graphValueIndex++] = Q;
            break;
    }

    if(RESULTS_BUFFER_SIZE <= graphValueIndex)
    {
        graphValueIndex = 0;
    }
} // end grafico()

//--- end of file -----------------------------------------------------
```

Fonte: Do autor (2021)

APÊNDICE B – Códigos implementados no DSP do algoritmo baseado em Filtro Adaptativo Sintonizado

Figura 4 – Código principal do programa - Filtro Adaptativo

```
// FILE:   DSP_Fas_1_cpu01.c

#include "F28x_Project.h"     // Device Header File and Examples Include File

#include "DSP_FAS_estimador.h"

// Function Prototypes
void ConfigureADC(void);
void ConfigureEPWM(void);
void SetupADCEpwm(void);
interrupt void adca1_isr(void);

extern void InitEPwmMode(void);
extern void InitGPIOMode(void);

void main(void)
{
    // Calcula parametro do filtro passa baixas recursivo
    alpha1 = 1-alpha;

    // Initialize System Control
    InitSysCtrl();
    EALLOW;
    ClkCfgRegs.PERCLKDIVSEL.bit.EPWMCLKDIV = 1;
    EDIS;

    // Initialize GPIO
    InitGpio();                          // Configure default GPIO
    EALLOW;
    GpioCtrlRegs.GPADIR.bit.GPIO31 = 1;     // Drives LED LD2 on controlCARD
    EDIS;
    GpioDataRegs.GPADAT.bit.GPIO31 = 1;     // Turn off LED

    // Clear all interrupts and initialize PIE vector table
    DINT;
    InitPieCtrl();
    IER = 0x0000;
    IFR = 0x0000;
    InitPieVectTable();
```

```
    // Map ISR functions
    EALLOW;
    PieVectTable.ADCA1_INT = &adca1_isr;    // Function for ADCA interrupt 1
    EDIS;

    // Init the output ePWMm
    InitEPwmMode();
    //InitGPIOMode();

    // Configure the ADC and power it up
    ConfigureADC();

    // Configure the ePWM
    ConfigureEPWM();

    // Setup the ADC for ePWM triggered conversions on channel 0
    SetupADCEpwm();

    // Initialize results buffer
    for(graphValueIndex = 0; graphValueIndex < RESULTS_BUFFER_SIZE; graphValue
    {
        graphValue[graphValueIndex] = 0;
    }
    graphValueIndex = 0;

    // Enable PIE interrupt
    PieCtrlRegs.PIEIER1.bit.INTx1 = 1;

    // Enable global interrupts and higher priority real-time debug events
    IER |= M_INT1;          // Enable group 1 interrupts
    EINT;                   // Enable Global interrupt INTM
    ERTM;                   // Enable Global real-time interrupt DBGM

    // Sync ePWM
    EALLOW;
    CpuSysRegs.PCLKCR0.bit.TBCLKSYNC = 1;
    EDIS;

    // Start ePWM
    EPwm3Regs.TBCTL.bit.CTRMODE = 0;
↪ // Un-freeze and enter up-count mode
```

```
    do {
        //GpioDataRegs.GPADAT.bit.GPIO31 = 0;     // Turn on LED
        GpioDataRegs.GPASET.bit.GPIO31 = 1;     // Turn on LED
        DELAY_US(1000 * 500);                   // ON delay
        //GpioDataRegs.GPADAT.bit.GPIO31 = 1;     // Turn off LED
        GpioDataRegs.GPACLEAR.bit.GPIO31 = 1;     // Turn off LED
        DELAY_US(1000 * 500);                   // OFF delay

    } while(1);
}


// Write ADC configurations and power up the ADC for both ADC A and ADC C
void ConfigureADC(void)
{
    EALLOW;

    // ADC-A (Tensao)
    AdcaRegs.ADCCTL2.bit.PRESCALE = 6;
↪ // Set ADCCLK divider to /4
    AdcaRegs.ADCCTL2.bit.RESOLUTION =  0;       // 12-bit resolution
    AdcaRegs.ADCCTL2.bit.SIGNALMODE = 0;
↪ // Single-ended channel conversions (12-bit mode only)
    AdcaRegs.ADCCTL1.bit.INTPULSEPOS = 1;
↪ // Set pulse positions to late
    AdcaRegs.ADCCTL1.bit.ADCPWDNZ = 1;          // Power up the ADC

    // ADC-C (Corrente)
    AdccRegs.ADCCTL2.bit.PRESCALE = 6;
↪ // Set ADCCLK divider to /4
    AdccRegs.ADCCTL2.bit.RESOLUTION =  0;
↪ // 12-bit resolution RESOLUTION_12BIT;
    AdccRegs.ADCCTL2.bit.SIGNALMODE = 0;
↪ // Single-ended channel conversions (12-bit mode only)
    AdccRegs.ADCCTL1.bit.INTPULSEPOS = 1;
↪ // Set pulse positions to late
    AdccRegs.ADCCTL1.bit.ADCPWDNZ = 1;          // Power up the ADC
    EDIS;

    DELAY_US(1000);
↪ // Delay for 1ms to allow ADC time to power up
}
```

```
void ConfigureEPWM(void)
{
    EALLOW;
    // Assumes ePWM clock is already enabled
    EPwm3Regs.TBCTL.bit.CTRMODE = 3;            // Freeze counter
    EPwm3Regs.TBCTL.bit.HSPCLKDIV = 0;
↪ // TBCLK pre-scaler = /1
    EPwm3Regs.TBPRD = 0x137e;
↪ // Set period to 4990 counts (20040 Hz)
    EPwm3Regs.ETSEL.bit.SOCAEN  = 0;
↪ // Disable SOC on A group
    EPwm3Regs.ETSEL.bit.SOCASEL = 2;
↪ // Select SOCA on period match
    EPwm3Regs.ETSEL.bit.SOCAEN = 1;             // Enable SOCA
    EPwm3Regs.ETPS.bit.SOCAPRD = 1;
↪ // Generate pulse on 1st event
    EDIS;
}


void SetupADCEpwm(void)
{
    // Select the channels to convert and end of conversion flag
    EALLOW;
    AdcaRegs.ADCSOC0CTL.bit.CHSEL = 3;
↪ // SOC0 will convert pin A3
    AdcaRegs.ADCSOC0CTL.bit.ACQPS = 14;
↪ // Sample window is 100 SYSCLK cycles
    AdcaRegs.ADCSOC0CTL.bit.TRIGSEL = 9;
↪ // Trigger on ePWM3 SOCA/C
    AdcaRegs.ADCINTSEL1N2.bit.INT1SEL = 0;
↪ // End of SOC0 will set INT1 flag
    AdcaRegs.ADCINTSEL1N2.bit.INT1E = 1;        // Enable INT1 flag
    AdcaRegs.ADCINTFLGCLR.bit.ADCINT1 = 1;
↪ // Make sure INT1 flag is cleared

    // Also setup ADC-C2 in this example
    AdccRegs.ADCSOC0CTL.bit.CHSEL = 2;
↪ // SOC0 will convert pin C2
    AdccRegs.ADCSOC0CTL.bit.ACQPS = 14;
↪ // Sample window is 100 SYSCLK cycles
    AdccRegs.ADCSOC0CTL.bit.TRIGSEL = 9;
↪ // Trigger on ePWM2 SOCA/C
    EDIS;
}


interrupt void adca1_isr(void)
{
    // Read the ADC result and store in circular buffer
    input();
```

```
    // Estima a potencia
    potencia();

    // Calcula e grava o duty cycle nos registradores dos PWMs de saida para D
    dac();

    // Controla status dos reles
    rele();

    // Armazena uma variavel selecionada em um buffer para ser exibido em graf
    grafico();

    // Return from interrupt
    AdcaRegs.ADCINTFLGCLR.bit.ADCINT1 = 1;
↪ // Clear ADC INT1 flag
    PieCtrlRegs.PIEACK.all = PIEACK_GROUP1;
↪ // Acknowledge PIE group 1 to enable further interrupts
}
 // end of file
```

Fonte: Do autor (2021)

Figura 5 – Código de configuração dos PWMs

```
/**********************************************************************
* File: DSP_EPwm.c
* Devices: TMS320F28x7x
* Author: Thales Roger, Silvia Ferreira, UFLA, 2020
**********************************************************************/

#include "F2837xD_device.h"
#include "F2837xD_Examples.h"

/**********************************************************************
* Function: InitEPwmMode()
*
* Description: Initializes the Enhanced PWM modules on the F28x7x for the IGBT
**********************************************************************/
void InitEPwmMode(void)
{
    asm(" EALLOW");                 // Enable EALLOW protected register access

   // Configure the prescaler to the ePWM modules.
↪ Max ePWM input clock is 100 MHz.
   ClkCfgRegs.PERCLKDIVSEL.bit.EPWMCLKDIV = 1;     // EPWMCLK divider from PLL
↪ 0=/1, 1=/2

   // Must disable the clock to the ePWM modules if you want all ePWM modules
   CpuSysRegs.PCLKCR0.bit.TBCLKSYNC = 0;

   asm(" EDIS");                   // Disable EALLOW protected register access


//--------------------------------------------------------------------------
//--- Configure GPIOs 157, 158, 159, 160 for EPwm output
//--------------------------------------------------------------------------
   EALLOW;

    //
    // Disable internal pull-up for the selected output pins
    // for reduced power consumption
    //
```

```
    GpioCtrlRegs.GPEPUD.bit.GPIO157 = 1;    // Disable pull-up on GPIO157 (EPW
    GpioCtrlRegs.GPEPUD.bit.GPIO158 = 1;    // Disable pull-up on GPIO158 (EPW
    GpioCtrlRegs.GPEPUD.bit.GPIO159 = 1;    // Disable pull-up on GPIO159 (EPW
    GpioCtrlRegs.GPFPUD.bit.GPIO160 = 1;    // Disable pull-up on GPIO160 (EPW

    //
    // Configure EPWM-1, EPWM-2 and EPWM-7 and EPWM-8 pins using GPIO regs
    // This specifies which of the possible GPIO pins will be EPWM functional
    // pins.
    //

    GpioCtrlRegs.GPEMUX2.bit.GPIO157 = 1;   // Configure GPIO157 as EPWM8A
    GpioCtrlRegs.GPEMUX2.bit.GPIO158 = 1;   // Configure GPIO158 as EPWM7B
    GpioCtrlRegs.GPEMUX2.bit.GPIO159 = 1;   // Configure GPIO159 as EPWM8A
    GpioCtrlRegs.GPFMUX1.bit.GPIO160 = 1;   // Configure GPIO160 as EPWM8B

    EDIS;

//-----------------------------------------------------------------------
//--- Configure ePWM7 for 250 kHz asymmetric PWM on EPWM7A and EPWM7B pins
//-----------------------------------------------------------------------
    asm(" EALLOW");                         // Enable EALLOW protected registe
    DevCfgRegs.SOFTPRES2.bit.EPWM7 = 1;     // ePWM8 is reset
    DevCfgRegs.SOFTPRES2.bit.EPWM7 = 0;     // ePWM8 is released from reset
    asm(" EDIS");                           // Disable EALLOW protected regist

    EPwm7Regs.TBCTL.bit.CTRMODE = 0x3;      // Disable the timer

    EPwm7Regs.TBCTL.all = 0xC033;           // Configure timer control registe
// bit 15-14     11:     FREE/SOFT, 11 = ignore emulation suspend
// bit 13        0:      PHSDIR, 0 = count down after sync event
// bit 12-10     000:    CLKDIV, 000 => TBCLK = HSPCLK/1
// bit 9-7       000:    HSPCLKDIV,     000 => HSPCLK = EPWMCLK/1
// bit 6         0:      SWFSYNC, 0 = no software sync produced
// bit 5-4       11:     SYNCOSEL, 11 = sync-out disabled
// bit 3         0:      PRDLD, 0 = reload PRD on counter=0
// bit 2         0:      PHSEN, 0 = phase control disabled
// bit 1-0       11:     CTRMODE, 11 = timer stopped (disabled)
```

```
    EPwm7Regs.TBCTR = 0x0000;           // Clear timer counter
    EPwm7Regs.TBPRD = 0x00C8;           // Set timer period to 500 counts (100
    EPwm7Regs.TBPHS.bit.TBPHS = 0x0000; // Set timer phase

    EPwm7Regs.CMPA.bit.CMPA = 0;     // Set PWM duty cycle 0%
    EPwm7Regs.CMPB.bit.CMPB = 0;     // Set PWM duty cycle 0%

    EPwm7Regs.CMPCTL.all = 0x000A;      // Compare control register
// bit 15-10     0's:    reserved
// bit 9         0:      SHDWBFULL, read-only
// bit 8         0:      SHDWAFULL, read-only
// bit 7         0:      reserved
// bit 6         0:      SHDWBMODE, 0 = shadow mode
// bit 5         0:      reserved
// bit 4         0:      SHDWAMODE, 0 = shadow mode
// bit 3-2       10:     LOADBMODE, 10 = load on zero or PRD match
// bit 1-0       10:     LOADAMODE, 10 = load on zero or PRD match

    EPwm7Regs.AQCTLA.all = 0x0060;      // Action-qualifier control register A
 // bit 15-12     0000:   reserved
 // bit 11-10     00:     CBD, 00 = do nothing
 // bit 9-8       00:     CBU, 00 = do nothing
 // bit 7-6       01:     CAD, 01 = clear
 // bit 5-4       10:     CAU, 10 = set
 // bit 3-2       00:     PRD, 00 = do nothing
 // bit 1-0       00:     ZRO, 00 = do nothing

     EPwm7Regs.AQCTLB.all = 0x0600;      // Action-qualifier control register
 // bit 15-12     0000:   reserved
 // bit 11-10     01:     CBD, 01 = clear
 // bit 9-8       10:     CBU, 10 = set
 // bit 7-6       00:     CAD, 00 = do nothing
 // bit 5-4       00:     CAU, 00 = do nothing
 // bit 3-2       00:     PRD, 00 = do nothing
 // bit 1-0       00:     ZRO, 00 = do nothing
```

```
    EPwm7Regs.AQSFRC.all = 0x0000;      // Action-qualifier s/w force register
// bit 15-8      0's:    reserved
// bit 7-6       00:     RLDCSF, 00 = reload AQCSFRC on zero
// bit 5         0:      OTSFB, 0 = do not initiate a s/w forced event on outp
// bit 4-3       00:     ACTSFB, don't care
// bit 2         0:      OTSFA, 0 = do not initiate a s/w forced event on outp
// bit 1-0       00:     ACTSFA, don't care

    EPwm7Regs.AQCSFRC.all = 0x0000;     // Action-qualifier continuous s/w for
// bit 15-4      0's:    reserved
// bit 3-2       00:     CSFB, 00 = forcing disabled
// bit 1-0       00:     CSFA, 00 = forcing disabled

    EPwm7Regs.DBCTL.bit.OUT_MODE = 0;   // Deadband disabled
    EPwm7Regs.PCCTL.bit.CHPEN = 0;      // PWM chopper unit disabled
    EPwm7Regs.TZDCSEL.all = 0x0000;     // All trip zone and DC compare action

    EPwm7Regs.TBCTL.bit.CTRMODE = 0x2;  // Enable the timer in up-down count m

//-----------------------------------------------------------------------
//--- Configure ePWM8 for 250 kHz asymmetric PWM on EPWM8A and EPWM8B pins
//-----------------------------------------------------------------------
    asm(" EALLOW");                             // Enable EALLOW protected registe
    DevCfgRegs.SOFTPRES2.bit.EPWM8 = 1;     // ePWM8 is reset
    DevCfgRegs.SOFTPRES2.bit.EPWM8 = 0;     // ePWM8 is released from reset
    asm(" EDIS");                               // Disable EALLOW protected regist

    EPwm8Regs.TBCTL.bit.CTRMODE = 0x3;      // Disable the timer
```

```
    EPwm8Regs.TBCTL.all = 0xC033;           // Configure timer control registe
// bit 15-14     11:     FREE/SOFT, 11 = ignore emulation suspend
// bit 13        0:      PHSDIR, 0 = count down after sync event
// bit 12-10     000:    CLKDIV, 000 => TBCLK = HSPCLK/1
// bit 9-7       000:    HSPCLKDIV,     000 => HSPCLK = EPWMCLK/1
// bit 6         0:      SWFSYNC, 0 = no software sync produced
// bit 5-4       11:     SYNCOSEL, 11 = sync-out disabled
// bit 3         0:      PRDLD, 0 = reload PRD on counter=0
// bit 2         0:      PHSEN, 0 = phase control disabled
// bit 1-0       11:     CTRMODE, 11 = timer stopped (disabled)

    EPwm8Regs.TBCTR = 0x0000;           // Clear timer counter
    EPwm8Regs.TBPRD = 0x00C8;           // Set timer period to 500 counts (100
    EPwm8Regs.TBPHS.bit.TBPHS = 0x0000; // Set timer phase

    EPwm8Regs.CMPA.bit.CMPA = 0;     // Set PWM duty cycle 0%
    EPwm8Regs.CMPB.bit.CMPB = 0;     // Set PWM duty cycle 0%

    EPwm8Regs.CMPCTL.all = 0x000A;      // Compare control register
// bit 15-10     0's:    reserved
// bit 9         0:      SHDWBFULL, read-only
// bit 8         0:      SHDWAFULL, read-only
// bit 7         0:      reserved
// bit 6         0:      SHDWBMODE, 0 = shadow mode
// bit 5         0:      reserved
// bit 4         0:      SHDWAMODE, 0 = shadow mode
// bit 3-2       10:     LOADBMODE, 10 = load on zero or PRD match
// bit 1-0       10:     LOADAMODE, 10 = load on zero or PRD match

    EPwm8Regs.AQCTLA.all = 0x0060;      // Action-qualifier control register A
 // bit 15-12     0000:   reserved
 // bit 11-10     00:     CBD, 00 = do nothing
 // bit 9-8       00:     CBU, 00 = do nothing
 // bit 7-6       01:     CAD, 01 = clear
 // bit 5-4       10:     CAU, 10 = set
 // bit 3-2       00:     PRD, 00 = do nothing
 // bit 1-0       00:     ZRO, 00 = do nothing
```

```
     EPwm8Regs.AQCTLB.all = 0x0600;      // Action-qualifier control register
 // bit 15-12     0000:   reserved
 // bit 11-10     01:     CBD, 01 = clear
 // bit 9-8       10:     CBU, 10 = set
 // bit 7-6       00:     CAD, 00 = do nothing
 // bit 5-4       00:     CAU, 00 = do nothing
 // bit 3-2       00:     PRD, 00 = do nothing
 // bit 1-0       00:     ZRO, 00 = do nothing

    EPwm8Regs.AQSFRC.all = 0x0000;      // Action-qualifier s/w force register
// bit 15-8      0's:    reserved
// bit 7-6       00:     RLDCSF, 00 = reload AQCSFRC on zero
// bit 5         0:      OTSFB, 0 = do not initiate a s/w forced event on outp
// bit 4-3       00:     ACTSFB, don't care
// bit 2         0:      OTSFA, 0 = do not initiate a s/w forced event on outp
// bit 1-0       00:     ACTSFA, don't care

    EPwm8Regs.AQCSFRC.all = 0x0000;     // Action-qualifier continuous s/w for
// bit 15-4      0's:    reserved
// bit 3-2       00:     CSFB, 00 = forcing disabled
// bit 1-0       00:     CSFA, 00 = forcing disabled

    EPwm8Regs.DBCTL.bit.OUT_MODE = 0;   // Deadband disabled
    EPwm8Regs.PCCTL.bit.CHPEN = 0;      // PWM chopper unit disabled
    EPwm8Regs.TZDCSEL.all = 0x0000;     // All trip zone and DC compare action

    EPwm8Regs.TBCTL.bit.CTRMODE = 0x2;  // Enable the timer in up-down count m


//-------------------------------------------------------------------
//--- Enable the clocks to the ePWM module.
//--- Note: this should be done after all ePWM modules are configured
//--- to ensure synchronization between the ePWM modules.
//-------------------------------------------------------------------
   asm(" EALLOW");                      // Enable EALLOW protected register acc
   CpuSysRegs.PCLKCR0.bit.TBCLKSYNC = 1;  // TBCLK to ePWM modules enabled
   asm(" EDIS");                       // Disable EALLOW protected register acces
```

```
   // Initialize GPIO for relay control
    EALLOW;
    GpioCtrlRegs.GPCDIR.bit.GPIO95 = 1;     // Drives LED LD2 on controlCARD
    EDIS;
    GpioDataRegs.GPCDAT.bit.GPIO95 = 0;     // Turn off output

} // end InitEPwmMode()

/*************************************************************************
* Function: InitGPIOMode()
*
* Description: Initializes the outputs as GPIO fot the IGBT control
*************************************************************************/

void InitGPIOMode(void)
{
    // Set GPIO pins as output
    EALLOW;
    GpioCtrlRegs.GPADIR.bit.GPIO0 = 1;
    GpioCtrlRegs.GPADIR.bit.GPIO1 = 1;
    GpioCtrlRegs.GPADIR.bit.GPIO2 = 1;
    GpioCtrlRegs.GPADIR.bit.GPIO3 = 1;
    EDIS;

    // Set GPIOs to low state
    GpioDataRegs.GPADAT.bit.GPIO0 = 0;
    GpioDataRegs.GPADAT.bit.GPIO1 = 0;
    GpioDataRegs.GPADAT.bit.GPIO2 = 0;
    GpioDataRegs.GPADAT.bit.GPIO3 = 0;

} // end InitGPIOMode()

//--- end of file -----------------------------------------------------
```

Fonte: Do autor (2021)

Figura 6 – Código contendo as funções que implementam o algoritmo baseado em Filtro Adaptativo

```
/**********************************************************************
* File: DSP_FAS_estimador.c
* Devices: TMS320F28x7x
* Author: Thales Roger, Silvia Ferreira, UFLA, 2020
**********************************************************************/

#include "F2837xD_device.h"
#include "F2837xD_Examples.h"
#include <math.h>

// Defines
#define T 4.99e-05
#define GAMA 2
#define ZETA 0.4
#define ZETA2 0.3

// Variables
#define RESULTS_BUFFER_SIZE 334
float32 graphValue[RESULTS_BUFFER_SIZE];
Uint16 graphValueIndex;
Uint16 selectGraph = 2;

float32 Vin, Iin, V, V90, I, I90, P, Q;

float32 w1=376.991118431;
float32 V1=0.0, Vd1=0.0, V5=0.0, Vd5=0.0, Ve1=0.0;
float32 I1=0.0, Id1=0.0, I5=0.0, Id5=0.0, Ie1=0.0;

float32 alpha = 0.99125951528549194;
float32 alpha1;

float32 pp = 0.0, P1 = 0.0, P2 = 0.0, P3 = 0.0;
```

```
float32 dac1duty = 0.0;
float32 dac2duty = 0.0;
float32 dac3duty = 0.0;
float32 dac4duty = 0.0;
float32 dac1max = 0.002;  // = 1/500 Potencia Ativa -
↪ Escala 0 a 100% de duty = 0 a 500 W
//float32 dac1max = 0.0025;  // = 1/400 Potencia Ativa -
↪ Escala 0 a 100% de duty = 150 a 550 W
float32 dac2max = 0.0016; // = 1/625 Potencia Reativa - Escala 0 a 100% de dut
float32 dac3max = 0.002;  // = 1/500 Tensao -
↪ Escala 0 a 100% de duty = -250 a 250 V
float32 dac4max = 0.0625; // = 1/16  Corrente -
↪ Escala 0 a 100% de duty = -8 a 8 A

bool rele1;

/************************************************************************
* Function: input()
*
* Description: Read the ADC result and store in circular buffer
************************************************************************/
void input(void)
{
    //Vin = 0.000759781*((float32)AdcaResultRegs.ADCRESULT0); // Valor direto
    Vin = 0.204075192*((float32)AdcaResultRegs.ADCRESULT0 - 2305.1); // Valor
    //Iin = 0.000754128*((float32)AdccResultRegs.ADCRESULT0); // Valor direto
    Iin = 0.01002206*((float32)AdccResultRegs.ADCRESULT0 - 2337.6); // Valor d

} // end input()


/************************************************************************
* Function: potencia()
*
* Description: Estima a potencia
************************************************************************/
void potencia(void)
{
    //----------FAS TENSAO + ESTIMADOR DE FREQUENCIA
    w1 = w1 - T*GAMA*V1*w1*Ve1;
```

```
    Vd1 = Vd1 + T*(2*ZETA*w1*Ve1 - w1*w1*V1);
    V1 = V1 + T*Vd1;

    Vd5 = Vd5 + T*(2*ZETA2*w1*Ve1 - 25*w1*w1*V5);
    V5 = V5 + T*Vd5;

    Ve1 = Vin - Vd1 - Vd5;

    V = Vd1;
    V90 = -w1*V1;

    //----------FAS CORRENTE
    Id1 = Id1 + T*(2*ZETA*w1*Ie1 - w1*w1*I1);
    I1 = I1 + T*Id1;

    Id5 = Id5 + T*(2*ZETA2*w1*Ie1 - 25*w1*w1*I5);
    I5 = I5 + T*Id5;

    Ie1 = Iin - Id1 - Id5;

    I = Id1;
    I90 = -w1*I1;

    //----------POTENCIAS
    pp = (V*I + V90*I90)/2;

    P1 = alpha*P1 + alpha1*pp;
    P2 = alpha*P2 + alpha1*P1;
    P3 = alpha*P3 + alpha1*P2;
    P = P3;

    Q = (V*I90 - V90*I)/2;

} //end potencia()
```

```
/**********************************************************************
* Function: dac()
*
* Description: Calcula e grava o duty cycle nos registradores dos PWMs de said
**********************************************************************/
void dac(void)
{
    dac1duty = P*dac1max;
    //dac1duty = (P-150)*dac1max;
    dac2duty = (Q+312.5)*dac2max;
    dac3duty = (Vin+250)*dac3max;
    dac4duty = (Iin+8)*dac4max;

    //dac1duty = 0.0;
    //dac2duty = 0.5;
    //dac3duty = 0.5;
    //dac4duty = 1;

    EPwm8Regs.CMPA.bit.CMPA = (1-dac1duty)*200;
↪ // Set PWM duty cycle
    EPwm8Regs.CMPB.bit.CMPB = (1-dac2duty)*200;
↪ // Set PWM duty cycle
    EPwm7Regs.CMPA.bit.CMPA = (1-dac3duty)*200;
↪ // Set PWM duty cycle
    EPwm7Regs.CMPB.bit.CMPB = (1-dac4duty)*200;
↪ // Set PWM duty cycle

} // end dac()


/**********************************************************************
* Function: rele()
*
* Description: Controla status dos GPIOs dos reles
**********************************************************************/
extern void rele(void){
    if (rele1 == 1){
        GpioDataRegs.GPCTOGGLE.bit.GPIO95 = 1;  // Toggle LED
        rele1 = 0;
    }

} // end rele()
```

```
/*********************************************************************
* Function: grafico()
*
* Description: Armazena uma variavel selecionada em um buffer para ser exibido
*********************************************************************/
void grafico(void)
{
    // GRAPH
    switch(selectGraph){
        case 0:
            graphValue[graphValueIndex++] = 0;
            break;

        case 1:
            graphValue[graphValueIndex++] = Vin;
            break;

        case 2:
            graphValue[graphValueIndex++] = Iin;
            break;

        case 3:
            graphValue[graphValueIndex++] = w1;
            break;

        case 4:
            graphValue[graphValueIndex++] = Vd1;
            break;

        case 5:
            graphValue[graphValueIndex++] = V1;
            break;

        case 6:
            graphValue[graphValueIndex++] = Vd5;
            break;

        case 7:
            graphValue[graphValueIndex++] = V5;
            break;
```

```
case 8:
    graphValue[graphValueIndex++] = Ve1;
    break;

case 9:
    graphValue[graphValueIndex++] = Id1;
    break;

case 10:
    graphValue[graphValueIndex++] = I1;
    break;

case 11:
    graphValue[graphValueIndex++] = Id5;
    break;

case 12:
    graphValue[graphValueIndex++] = I5;
    break;

case 13:
    graphValue[graphValueIndex++] = Ie1;
    break;

case 14:
    graphValue[graphValueIndex++] = V;
    break;

case 15:
    graphValue[graphValueIndex++] = V90;
    break;

case 16:
    graphValue[graphValueIndex++] = I;
    break;

case 17:
    graphValue[graphValueIndex++] = I90;
    break;
```

```
        case 18:
            graphValue[graphValueIndex++] = pp;
            break;

        case 19:
            graphValue[graphValueIndex++] = P;
            break;

        case 20:
            graphValue[graphValueIndex++] = Q;
            break;
    }

    if(RESULTS_BUFFER_SIZE <= graphValueIndex)
    {
        graphValueIndex = 0;
    }
} // end grafico()

//--- end of file -----------------------------------------------------
```

Fonte: Do autor (2021)