IMPLEMENTAÇÃO DE SISTEMA PLL PARA REDES TRIFÁSICAS BASEADO EM FILTRO SOGI

PEDRO VINÍCIUS GUIMARÃES SOUZA

TRABALHO DE GRADUAÇÃO
DEPARTAMENTO DE ENGENHARIA ELÉTRICA

FACULDADE DE TECNOLOGIA

UNIVERSIDADE DE BRASÍLIA

Universidade de Brasília
Faculdade de Tecnologia
Departamento de Engenharia Elétrica

IMPLEMENTAÇÃO DE SISTEMA PLL PARA REDES TRIFÁSICAS BASEADO EM FILTRO SOGI

PEDRO VINÍCIUS GUIMARÃES SOUZA

Trabalho final de graduação submetido ao Departamento de Engenharia Elétrica da Faculdade de Tecnologia da Universidade de Brasília, como parte dos requisitos necessários para a obtenção do grau de Engenheiro Eletricista.

APROVADA POR:

Prof. Lélio Ribeiro Soares Júnior (ENE-UnB)
(Orientador)

Prof. Adolfo Bauchspiess, D.Sc. (ENE-UnB)
(Examinador Interno)

Prof. Alex Reis, D.Sc. (FGA-UnB)
(Examinador Interno)

Brasília/DF, 10 de julho de 2018.

FICHA CATALOGRÁFICA

SOUZA, PEDRO VINÍCIUS GUIMARÃES

Implementação de Sistema PLL para Redes Trifásicas Baseado em Filtro SOGI. [Distrito Federal] 2018.

xiii, 53p., 210 x 297 mm (ENE/FT/UnB, Engenheiro Eletricista, Engenharia Elétrica, 2018).

Trabalho de Graduação – Universidade de Brasília, Faculdade de Tecnologia.

Departamento de Engenharia Elétrica

1. Phase-Locked Loop (PLL)
2. SRF-PLL
3. DSOGI-PLL
4. Controle Dinâmico
5. Energia Eólica
6. Geração Distribuída

I. ENE/FT/UnB

II. Título (série)

REFERÊNCIA BIBLIOGRÁFICA

SOUZA, P. V. G. (2018). Implementação de Sistema PLL para Redes Trifásicas Baseado em Filtro SOGI, Trabalho de Graduação em Engenharia Elétrica, Publicação 2018, Departamento de Engenharia Elétrica, Universidade de Brasília, Brasília, DF, 53p.

CESSÃO DE DIREITOS

AUTOR: Pedro Vinícius Guimarães Souza.

TÍTULO: Implementação de Sistema PLL para Redes Trifásicas Baseado em Filtro SOGI.

GRAU: Engenheiro Eletricista. ANO: 2018.

Pedro Vinícius Guimarães Souza

Departamento de Eng. Elétrica

Universidade de Brasília

Campus Darcy Ribeiro

Para minha mãe, que é meu maior exemplo de altruísmo e perseverança.

AGRADECIMENTOS

Agradeço, primeiramente, a minha família: meus pais Cida e Valtercides, e minha irmã Thaysa. O suporte e as palavras de incentivo a mim concedidas, durante toda a graduação, foram fundamentais nesta jornada.

Agradeço aos amigos Louise Cugula, Pedro H. Rocha, Rebeca Carvalho e Tainara Santana, os quais representam uma grande fonte de inspiração pessoal. É com extrema gratidão e carinho que guardo os conselhos e os momentos de alegria compartilhados.

Agradeço também ao Carlos Henrique S. Mendonça, que generosamente se dispôs a me auxiliar com a estrutura laboratorial montada no Laboratório de Conversão de Energia.

Por fim, agradeço a todos os professores que contribuíram positivamente com a minha formação; que me apresentaram novas perspectivas e que reacenderam meu interesse nos momentos de desânimo. Em especial, ao professor Anésio de Leles Ferreira Filho, que me disponibilizou um espaço de trabalho no LQEE, e ao professor Lélio, com quem pude aprender grandemente no último ano. Sua paciência e disponibilidade foram determinantes para a conclusão deste trabalho.

RESUMO

A expansão da matriz eólica brasileira demanda técnicas robustas de sincronismo com a rede elétrica. Independentemente de quaisquer pertubações, espera-se que tais técnicas possam realizar a medida do ângulo de fase dessa rede. Os sistemas de controle responsáveis por essa medição são denominados *Phase-Locked Loop* (PLL). Este trabalho compara, por meio de simulações computacionais, duas topologias de PLL: o *Synchronous Reference Frame* (SRF-PLL) e o *Dual Second Order Generalized Integrator* (DSOGI-PLL). A maior adaptabilidade do DSOGI-PLL a pertubações na rede, como harmônicas e desequilíbrios de fase, é ratificada. Em seguida, implementa-se o DSOGI-PLL na plataforma *Arduino Due*. É descrita a estrutura laboratorial necessária para converter um sinal de 127 V_{rms} para um valor de tensão aceito pelo microcontrolador. Utiliza-se, também, a transformação de Tustin e a transformada Z para a implementação digital das funções de transferência do DSOGI-PLL. O sistema é executado a uma taxa de amostragem de 2500 Hz e são obtidas, a partir da tensão trifásica de entrada: as componentes alfa e beta de sequência positiva, as componentes *dq*, do eixo de referência síncrono e o ângulo de fase.

Palavras-chave: *Phase-Locked Loop*, SRF-PLL, DSOGI-PLL, *Arduino Due*.

ABSTRACT

The growth of Brazil's wind power capacity demands strong synchronization techniques, able to track the phase angle of the grid voltage, despite any disturbances that may affect that grid. The control system responsible for that are called Phase-Locked Loop (PLL). Simulations are performed in order to compare two PLL topologies: The Synchronous Reference Frame (SRF-PLL) and the Dual Second Order Generalized Integrator (DSOGI-PLL). The latter shows a better performance, when facing grid disturbances, like harmonics and voltage unbalances. Later on, this work aims to implement the DSOGI-PLL on the electronic prototyping platform Arduino Due. It describes the structure needed to convert a 127 V_{rms} signal to a voltage level accepted by the board. Then, by means of the Tustin transformation and the Z transform, it implements the DSOGI-PLL digitally. Finally, the system runs with a 2500 Hz sample rate. The results obtained from the input voltage are: the alpha and beta positive sequence components, the dq components, from the synchronous reference frame and the phase angle.

Keywords: Phase-Locked Loop, SRF-PLL, DSOGI-PLL, Arduino Due.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

CAPÍTULO 1

INTRODUÇÃO

O aumento da utilização de fontes de energia renováveis mostra-se um fator significativo na transformação da matriz energética brasileira. Em 2017, a energia eólica destacou-se por representar 30,2% da expansão líquida da capacidade instalada nacional de geração elétrica, em relação ao ano anterior (MME, 2018).

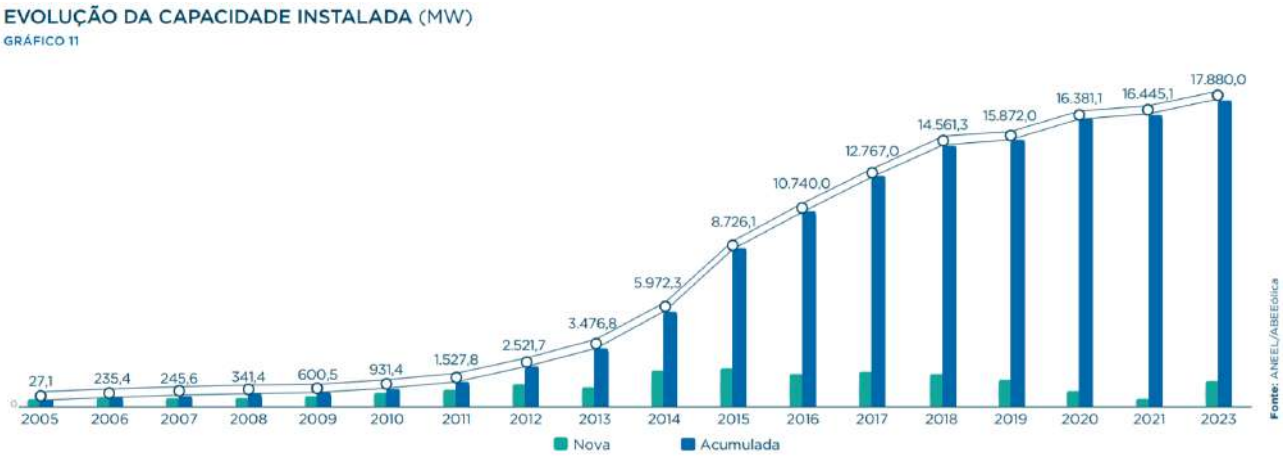

Figura 1.1: Evolução da capacidade eólica instalada nacional
Fonte: (ABEEOLICA, 2018)

Além de previsões para anos futuros, a figura 1.1 mostra o crescimento da capacidade instalada de energia eólica até o ano de 2017, no qual o Brasil alcançou o sexto lugar no ranking mundial de capacidade eólica nova (ABEEOLICA, 2018). Frente a esta inegável expansão da matriz eólica brasileira, entende-se a importância de aperfeiçoar as tecnologias referentes à mesma.

Neste contexto, os conversores de potência utilizados para o acoplamento da geração distribuída ao sistema elétrico apresentam uma grande variedade de temas para estudo. "Um dos aspectos mais importantes a ser considerado no controle de conversores de potência é a correta sincronização com a tensão trifásica da rede elétrica"(TEODORESCU *et al.*, 2011, p. 169). Em busca deste objetivo, faz-se necessária uma detecção precisa do ângulo de fase desta tensão

(CHUNG, 2000).

Tal tarefa é realizada pelo sistema de controle denominado *Phase-Locked Loop* (PLL), ou Malha de Captura de Fase. Dentre as topologias utilizadas em sistemas eólicos, o DSOGI-PLL destaca-se por sua simplicidade (RODRIGUEZ *et al.*, 2006), bem como sua capacidade de resistir a distorções na rede (BOBROWSKA-RAFAL *et al.*, 2011).

1.1 OBJETIVOS

O presente trabalho tem como objetivo analisar e comparar duas topologias de PLL, o *Synchronous Reference Frame* (SRF-PLL) e o *Dual Second Order Generalized Integrator* (DSOGI-PLL), por meio de simulações. Deseja-se comprovar a superioridade do DSOGI-PLL perante pertubações na rede elétrica. Para tanto, ambos os sistemas são simulados com sinais de tensão i) desbalanceados e ii) afetados por distorções harmônicas.

Objetiva-se também implementar o DSOGI-PLL na rede trifásica disponibilizada no Laboratório de Conversão de Energia da Universidade de Brasília (UnB), por meio de três etapas principais:

- Implementação no software *MATLAB*
- Elaboração de código em linguagem C
- Implementação final na plataforma de prototipagem eletrônica *Arduino Due*.

Ademais, espera-se que o código em *C* seja utilizado em estudos futuros do Laboratório de Qualidade de Energia Elétrica (LQEE), também da UnB, o qual possui uma bancada com gerador síncrono, que simula uma turbina eólica.

1.2 ESTRUTURA DO TRABALHO

O capítulo 2, Fundamentação Teórica, apresenta a estrutura de um PLL e a modelagem matemática que o define. Nesta etapa, simulações computacionais são efetuadas no *software Simulink*, a fim de se comparar o SRF-PLL e o DSOGI-PLL e ratificar a maior robustez do último.

O capítulo 3, Materiais e Métodos, introduz as ferramentas necessárias para a implementação digital do DSOGI-PLL em um microcontrolador *Arduino Due*. Detalham-se todos o códigos desenvolvidos, bem como a estrutura laboratorial necessária para a aquisição de dados.

Em seguida, o capítulo 4 apresenta os resultados obtidos e tem-se, por fim, no capítulo 5 as conclusões deste trabalho, juntamente com sugestões de trabalhos futuros.

CAPÍTULO 2

FUNDAMENTAÇÃO TEÓRICA

2.1 COMPONENTES SIMÉTRICAS

O método de componentes simétricas, proposto por Fortescue, é uma transformação linear de componentes de fase para outros três conjuntos de componentes, denominados sequência positiva (1), sequência negativa (2) e sequencia zero (0) (GLOVER *et al.*, 2012):

$$\begin{bmatrix} V_0 \\ V_1 \\ V_2 \end{bmatrix} = \frac{1}{3} \begin{bmatrix} 1 & 1 & 1 \\ 1 & a & a^2 \\ 1 & a^2 & a \end{bmatrix} \begin{bmatrix} V_a \\ V_b \\ V_c \end{bmatrix} \tag{2.1}$$

Onde: $a = 1\angle 120°$.

Em sistemas trifásicos equilibrados, de componentes *abc*, apenas a componente de sequência positiva é observada após a transformação. De forma similar, analisando-se pertubações na rede elétrica, quando uma "falta trifásica equilibrada ocorre em um sistema trifásico equilibrado, há apenas corrente de falta de sequência positiva"(GLOVER *et al.*, 2012, p. 471). Abaixo, aplicou-se o método a um vetor de tensões trifásicas equilibradas unitárias para exemplificar esta situação:

$$\begin{bmatrix} V_a \\ V_b \\ V_c \end{bmatrix} = \begin{bmatrix} 1\angle 0° \\ 1\angle -120° \\ 1\angle +120° \end{bmatrix} \tag{2.2}$$

Aplicando-se o método de componentes simétricas:

$$V_0 = \frac{1}{3}[1\angle 0° + 1\angle -120° + 1\angle +120°] = 0 \tag{2.3}$$

$$V_1 = \frac{1}{3}[1\angle 0° + (1\angle -120°)1\angle 120° + (1\angle +120°)1\angle 240°] = 1\angle 0° \tag{2.4}$$

$$V_2 = \frac{1}{3}[1\angle 0° + (1\angle -120°)1\angle 240° + (1\angle +120°)1\angle 120°] = 0 \tag{2.5}$$

Conversores de potência geralmente injetam correntes de sequência positiva na rede elétrica. Portanto, "a correta detecção da componente de sequência positiva, na frequência fundamental,

da tensão trifásica da rede elétrica pode ser considerada a tarefa principal de um sistema de sincronismo de um conversor de potência conectado a essa rede"(TEODORESCU *et al.*, 2011, p. 171). Na seção 2.4, explica-se porque o cálculo da componente de sequência positiva é essencial para o adequado funcionamento do DSOGI-PLL.

2.2 PLL - PHASE-LOCKED LOOP

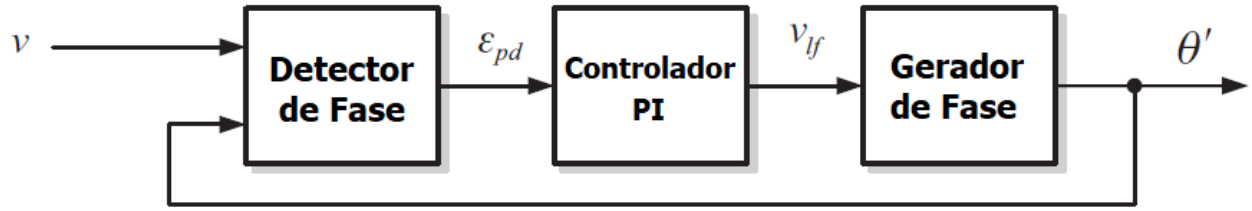

Figura 2.1: Estrutura básica de um PLL
Fonte: (TEODORESCU et al, 2011)

O PLL é um sistema em malha fechada utilizado para se obter sincronismo entre o conversor de potência e a rede elétrica. A fim de se alcançar o objetivo desejado, deve-se monitorar em tempo real o comportamento dessa rede. Sendo os valores de tensão elétrica as variáveis de entrada do sistema, espera-se, na sua saída, a detecção da frequência e da fase no instante de análise. A aquisição das componentes de sequência positiva é uma das etapas intermediárias realizadas pelo PLL. Este sistema é caracterizado por três partes principais:

- Detector de fase
- Controlador PI
- Gerador de frequência e fase

De acordo com a figura 2.1, a fase é o parâmetro que retorna por meio da malha fechada. Na saída do detector de fase, tem-se um sinal proporcional à diferença da fase do novo sinal de entrada e da fase gerada na saída do sistema. Desta forma, é possível monitorar a variação deste parâmetro em relação ao seu último valor medido.

2.3 SRF-PLL

O *Synchronous Reference Frame* (SRF-PLL) é uma das configurações mais populares desta técnica de medição. Portanto, sua performance deve ser avaliada. Um PLL monofásico com detector de fase em quadratura será analisado primeiramente, pois o entendimento do SRF-PLL deriva-se naturalmente desta configuração. Como visto na figura 2.2, esta técnica implementa uma modulação de amplitude em quadratura. A fase (posição angular) é calculada na saída do sistema e retorna pela malha fechada como argumento das portadoras da modulação. No equacionamento que se segue, θ' é o último valor de fase medido, enquanto θ é a fase do sinal a ser analisado na entrada do sistema.

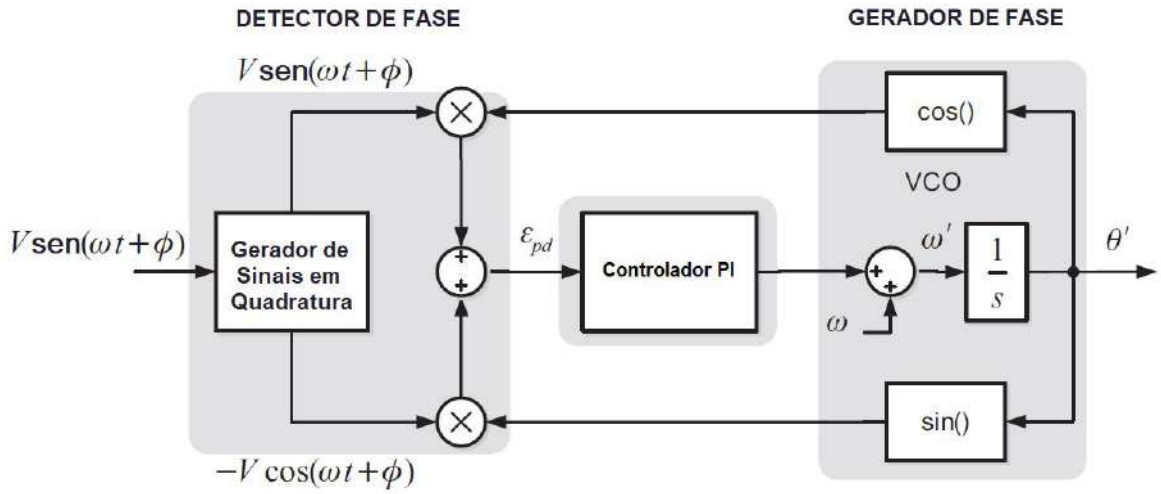

Figura 2.2: PLL com Detector de Fase em Quadratura
Fonte: (TEODORESCU et al, 2011)

Uma vez que θ' é igual a $(\omega' t + \phi')$, tem-se na saída do detector de fase:

$$\varepsilon_{pd} = V sen(\omega t + \phi) cos(\omega' t + \phi') - V cos(\omega t + \phi) sen(\omega' t + \phi') \tag{2.6}$$

Aqui, deve-se lembrar da seguinte relação trigonométrica:

$$sen(A)cos(B) = \frac{1}{2}[sen(A - B) + sen(A + B)] \tag{2.7}$$

Logo:

$$\varepsilon_{pd} = V sen(\theta) cos(\theta') - V cos(\theta) sen(\theta') \tag{2.8}$$

$$\varepsilon_{pd} = \frac{V}{2}[sen(\theta - \theta') + sen(\theta + \theta')] - \frac{V}{2}[sen(\theta' - \theta) + sen(\theta' + \theta)] \tag{2.9}$$

$$\varepsilon_{pd} = \frac{V}{2}[sen(\theta - \theta') - sen(\theta' - \theta) + sen(\theta + \theta') - sen(\theta' + \theta)] \tag{2.10}$$

Agora, θ será escrito novamente em função de ω e ϕ, para evidenciar que surgem componentes na frequência $(\omega + \omega')$:

$$\varepsilon_{pd} = \frac{V}{2}[sen((\omega - \omega')t + (\phi - \phi')) - sen((\omega' - \omega)t + (\phi' - \phi)) \tag{2.11}$$

$$+sen((\omega + \omega')t + (\phi + \phi')) - sen((\omega + \omega')t + (\phi + \phi'))] \tag{2.12}$$

Estas componentes com o dobro da frequência do sinal de entrada (caso $\omega = \omega'$) se anulam. Isso não ocorreria no caso mais simples de detecção de erro, que utiliza simplesmente o produto do sinais senoidais. Logo, fica claro o porquê da implementação de sinais em quadratura.

Por fim:

$$\varepsilon_{pd} = \frac{V}{2}[sen(\theta - \theta') - sen(\theta' - \theta)] \tag{2.13}$$

$$\varepsilon_{pd} = Vsen(\theta - \theta') \tag{2.14}$$

Quando $\theta = \theta'$:

$$\varepsilon_{pd} = 0 \tag{2.15}$$

O equacionamento evidencia que se o PLL estiver bem sincronizado ($\theta = \theta'$), o sinal resultante ε_{pd} não apresentará termo oscilatório em sua resposta de regime permanente. Se não há termos oscilatórios na entrada do controlador, a frequência estimada pelo PLL será mais precisa.

Analisando-se novamente o equacionamento do sinal de erro na saída do detector de fase da figura 2.2, percebe-se que o modelo estruturado pode ser matematicamente representado pela transformada de Park:

$$\begin{bmatrix} vd \\ vq \\ v0 \end{bmatrix} = \begin{bmatrix} cos(\theta) & sen(\theta) & 0 \\ -sen(\theta) & cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} v\alpha \\ v\beta \\ v0 \end{bmatrix} \tag{2.16}$$

A transformada de Park consiste na transposição de componentes em quadratura, representadas no plano $\alpha\beta 0$, para um plano de referência síncrono chamado plano *dq0*. O sinal de erro desejado na saída do detector de fase é análogo à componente vq desta transformada:

$$\varepsilon_{pd} = Vsen(\omega t + \phi)cos(\omega' t + \phi') - Vcos(\omega t + \phi)sen(\omega' t + \phi') \tag{2.17}$$

$$vq = v\beta cos(\theta') - v\alpha sen(\theta') \tag{2.18}$$

Neste ponto, será aproveitada a linha de raciocínio desenvolvida para a introdução do caso trifásico, analisando-se a configuração do SRF-PLL.

"Sistemas de geração distribuída são geralmente conectados à rede trifásica por meio de uma conexão de três fios e, portanto, não injetam corrente de sequência zero na rede"(TEODORESCU *et al.*, 2011, p. 171). Dessa forma, a componente de sequência zero do vetor de tensão será ignorada, simplificando o modelo adotado da transformada de Park:

$$\begin{bmatrix} vd \\ vq \end{bmatrix} = \begin{bmatrix} cos(\theta) & sen(\theta) \\ -sen(\theta) & cos(\theta) \end{bmatrix} \begin{bmatrix} v\alpha \\ v\beta \end{bmatrix} \tag{2.19}$$

No caso monofásico da figura 2.2, foi necessário gerar uma cópia do sinal de entrada, deslocada de 90 graus, para se obter os sinais em quadratura. Contudo, um sinal trifásico pode ser matematicamente representado por duas componentes ortogonais, por meio da transformada de Clarke, a qual transporta um vetor do plano *abc* para o plano $\alpha\beta$:

$$\begin{bmatrix} v\alpha \\ v\beta \end{bmatrix} = \sqrt{\frac{2}{3}} \begin{bmatrix} 1 & -\frac{1}{2} & -\frac{1}{2} \\ 0 & \frac{\sqrt{3}}{2} & -\frac{\sqrt{3}}{2} \end{bmatrix} \begin{bmatrix} va \\ vb \\ vc \end{bmatrix} \tag{2.20}$$

Na figura 2.3, o caso trifásico está representado. Esta configuração recebe o nome de *Synchronous Reference Frame* (SRF-PLL).

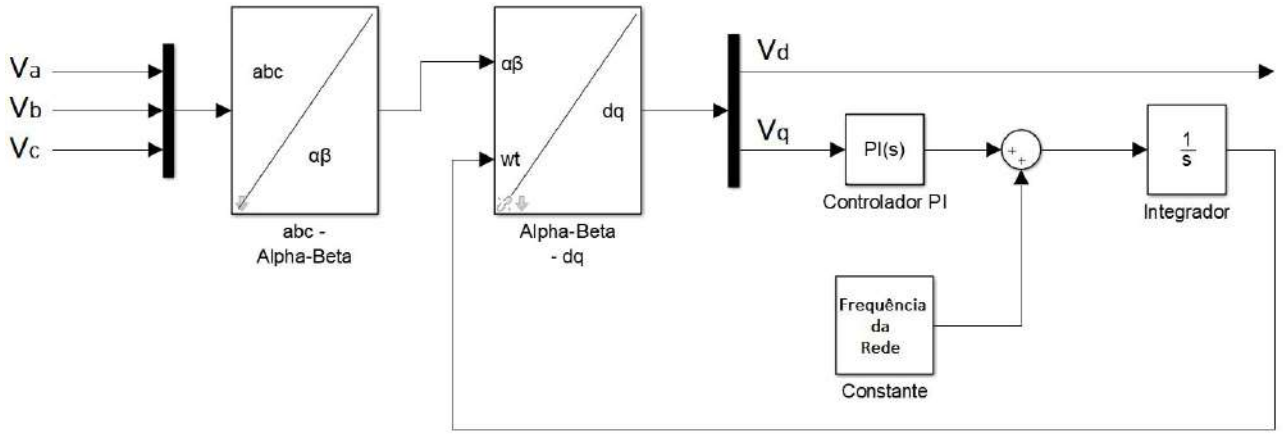

Figura 2.3: SRF-PLL

Transformar o sinal trifásico, primeiramente colocando-o em um novo sistema de coordenadas $\alpha\beta$, por meio da transformada de Clarke, e depois no plano *dq*, por meio da transformada de Park, facilita o projeto do sistema de controle. O plano *dq* é sincrono e move-se com a mesma

velocidade angular da tensão de entrada. Portanto, em um sistema equilibrado, as componentes deste plano serão sempre constantes, como explicitado no gráfico da figura 2.4. O que possibilita, na sequência do diagrama de blocos, a utilização de um controlador PI projetado para seguir referências constantes e não senoidais.

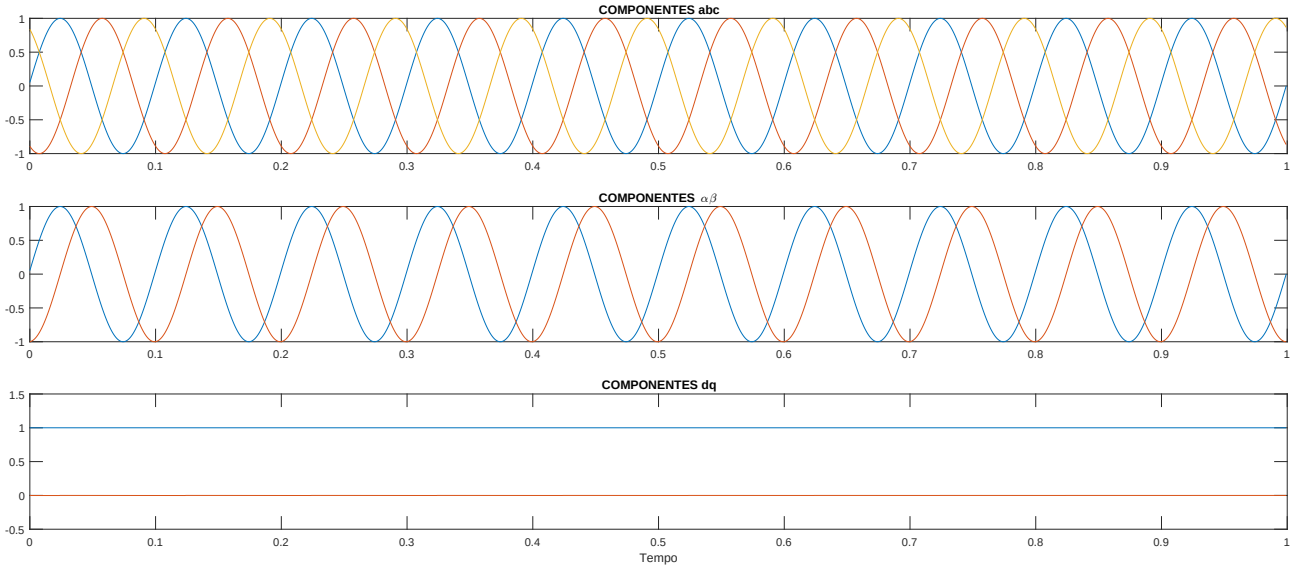

Figura 2.4: Componentes abc, $\alpha\beta$ e dq de um sistema equilibrado no domínio do tempo

Por meio da transformada de Park, pode-se manter a componente q em zero. Consequentemente, a componente d será igual ao módulo do vetor resultante das tensões de entrada, representado no diagrama fasorial da figura 2.5 e calculado como:

$$|v| = \sqrt{\frac{3}{2}}V \tag{2.21}$$

Onde: V é a amplitude dos sinais v_a, v_b e v_c.

Para que a componente d represente a amplitude de sequência positiva V, modifica-se a transformada de Clarke, multiplicando-a por $\sqrt{2/3}$. Logo, a transformada de Clarke efetivamente utilizada é:

$$\begin{bmatrix} v\alpha \\ v\beta \end{bmatrix} = \frac{2}{3}\begin{bmatrix} 1 & -\frac{1}{2} & -\frac{1}{2} \\ 0 & \frac{\sqrt{3}}{2} & -\frac{\sqrt{3}}{2} \end{bmatrix}\begin{bmatrix} va \\ vb \\ vc \end{bmatrix} \tag{2.22}$$

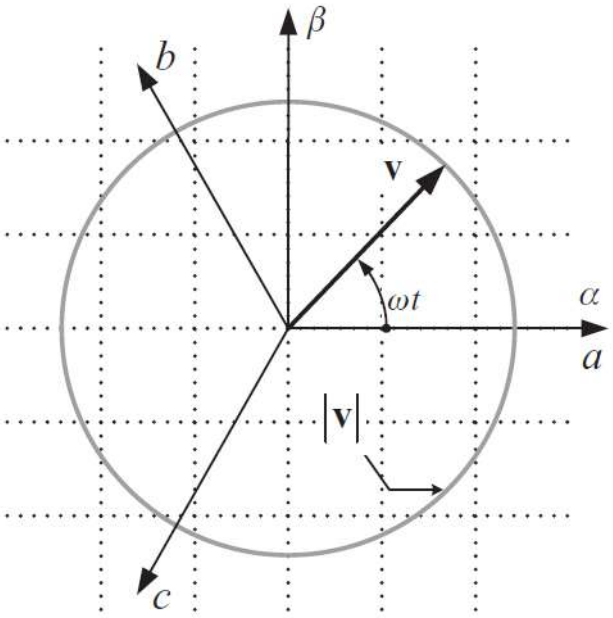

Figura 2.5: Módulo do vetor resultante das tensões *abc*
Fonte: (TEODORESCU et al, 2011)

No gráfico da figura 2.4, com as componetes representadas no tempo, já se utilizou a transformada de Clarke modificada. Percebe-se que a componente d representa a amplitude da tensão v_a, v_b e v_c e, consequentemente, a amplitude de sequência positiva.

A determinação do módulo de sequência positiva da tensão de entrada, no SRF-PLL, é portanto uma consequência de se trabalhar com um sistema equilibrado. Contudo, segundo Rodriguez *et al.* (2006) , sistemas de controle de conversores conectados à rede devem garantir uma adaptação rápida a pertubações, como desequilíbrios e harmônicas.

O surgimento de componentes fora da frequência fundamental, consequência destas pertubações, afeta significativamente o módulo do vetor resultante, como mostrado por Teodorescu *et al.* (2011) na equação 2.23:

$$|v| = \sqrt{\frac{3}{2}[(V^{+1})^2 + (V^n)^2 + 2V^{+1}V^n cos((n-1)\omega t)]} \tag{2.23}$$

$$\theta = \omega t + tan^{-1}\left[\frac{V^n sen((n-1)\omega t)}{V^{+1} + V^n cos((n-1)\omega t)}\right] \tag{2.24}$$

Onde: n representa a enésima harmônica presente no sistema, a qual pode ser tanto de sequência positiva $(+n)$ como negativa $(-n)$.

Por meio da equação 2.24, percebe-se que o calculo da fase é também severamente afetado. Em um sistema trifásico equilibrado ideal, espera-se que a fase varie linearmente de 0 a 2π rad/s. Consequentemente, a frequência angular do sistema é constante, uma vez que:

$$\omega = \frac{d\theta}{dt} \tag{2.25}$$

A equação 2.24 confirma, portanto, as limitações do SRF-PLL neste cenário de pertubação. Esta topologia falha em rastrear um valor constante de frequência angular, uma vez que o valor de tensão recebido na entrada do sistema está também variando.

2.3.1 Simulações do SRF-PLL

Pertubações propostas por Teodorescu *et al.* (2011) foram simulados em ambiente *Simulink*, a fim de se registrar essa limitação no comportamento do SRF-PLL. Utilizou-se um tempo total de simulação de 0,2 segundos. Nos casos 2 e 3 as pertubações são introduzidos a partir de 0,1 segundos.

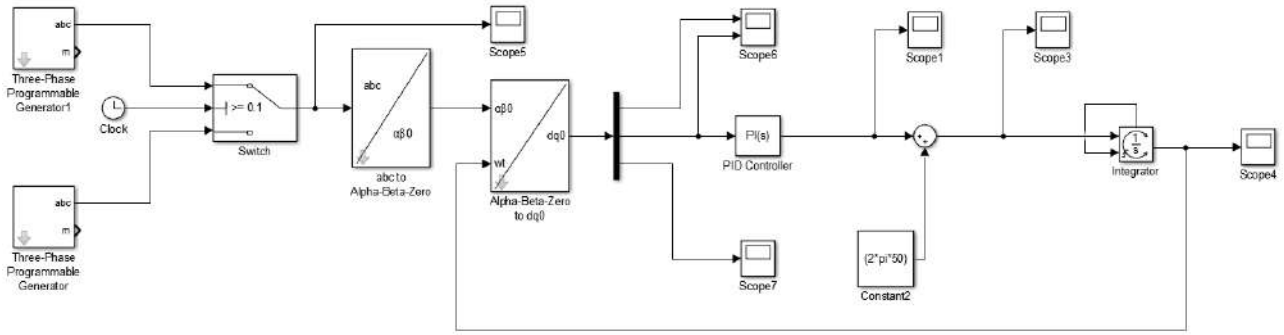

Figura 2.6: Diagrama de blocos do SRF-PLL implementado no Simulink

- Caso 1: Tensão trifásica de entrada equilibrada

Intervalo de Tempo (s)	Tensão de Entrada
0 a 0,2	v^{+1}

Tabela 2.1: SRF-PLL, Caso 1 - Condições de simulação

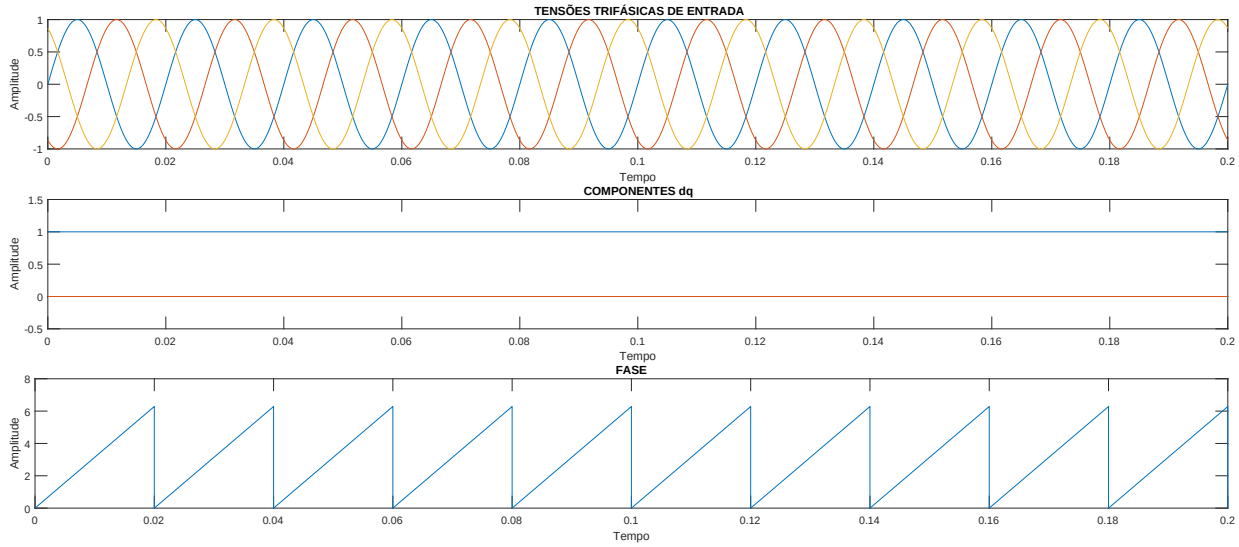

Figura 2.7: Caso 1 - Componentes *abc*, *dq* e fase

- Caso 2: Curto-circuito trifásico com harmônicas de quinta ordem

Intervalo de Tempo (s)	Tensão de Entrada
0 a 0,1	v^{+1}
0,1 a 0,2	$0,75v^{+1} + 0,075v^{-5}$

Tabela 2.2: SRF-PLL, Caso 2 - Condições de simulação

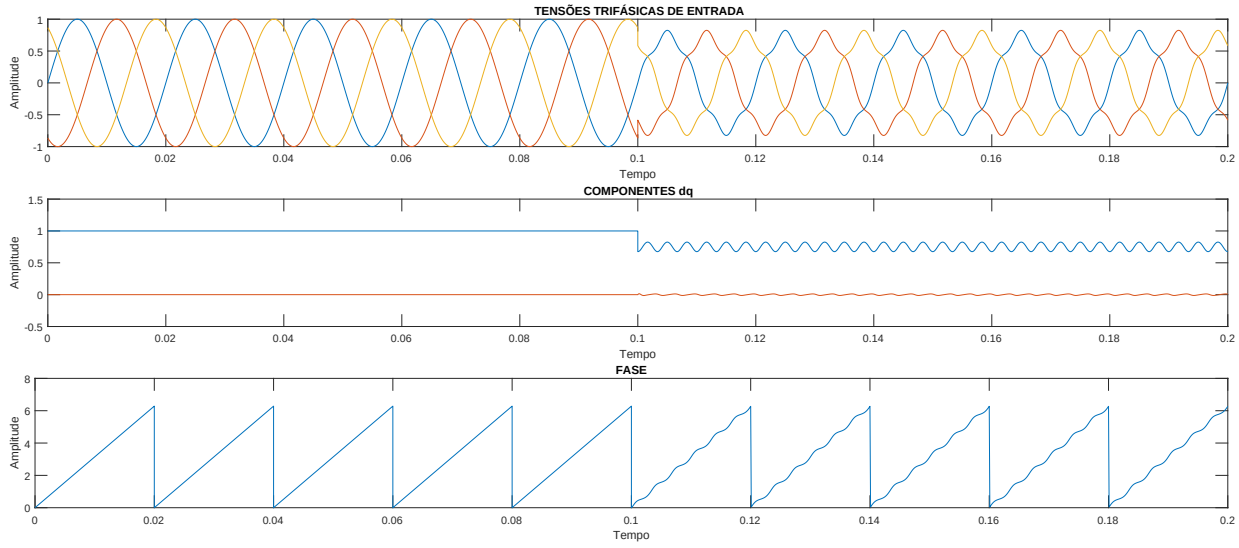

Figura 2.8: Caso 2 - Componentes *abc*, *dq* e fase

A figura 2.8 mostra as distorções causadas por harmônicas na medição da posição angular pelo SRF-PLL. Embora este problema possa ser resolvido por meio da redução da largura de banda do controlador PI, esta não é uma prática recomendada (TEODORESCU *et al.*, 2011).

- Caso 3: Curto-circuito fase-fase

Intervalo de Tempo (s)	Tensão de Entrada
0 a 0,1	v^{+1}
0,1 a 0,2	$0,75v^{+1} + 0,25v^{-1}$

Tabela 2.3: SRF-PLL, Caso 3 - Condições de simulação

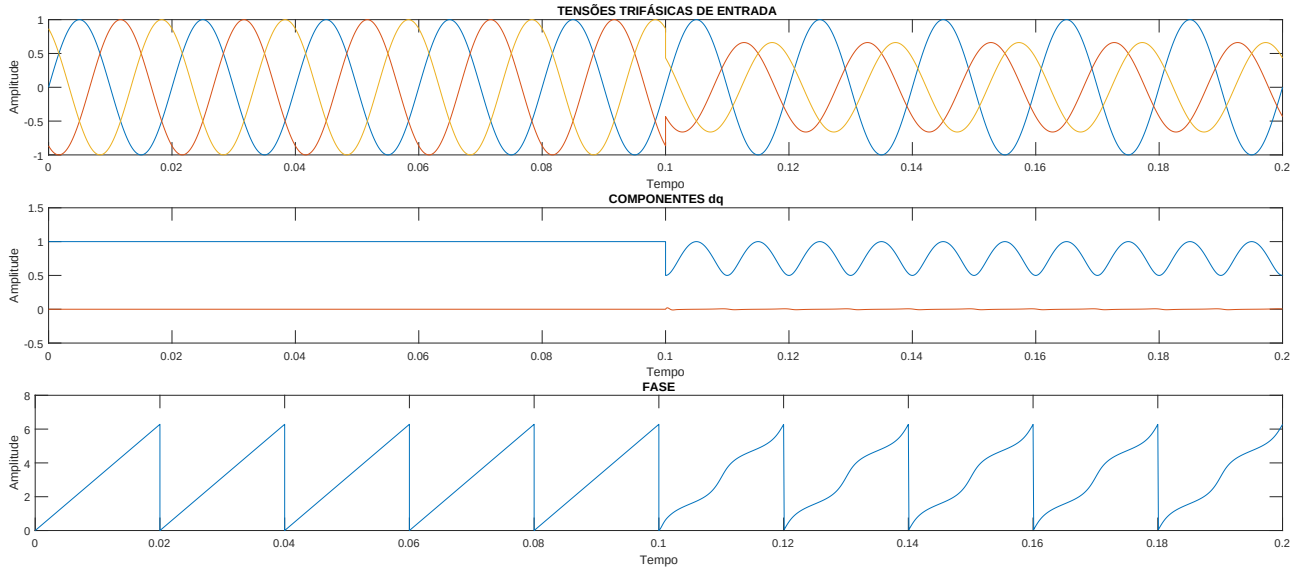

Figura 2.9: Caso 3 - Componentes *abc*, *dq* e fase

No caso de um desequilíbrio de fases, como simulado na figura 2.9, notam-se pertubações ainda mais acentuadas. De acordo com as equações 2.23 e 2.24:

$$|v| = \sqrt{0,9375 + 0,5625cos(2\omega t)} \tag{2.26}$$

$$\theta = \omega t + tan^{-1}\left[\frac{0,25sen(-2\omega t)}{0,75 + 0,25cos(2\omega t)}\right] \tag{2.27}$$

Ambas as equações mostram que a informação referente à componente de sequência positiva não pode ser adquirida por técnicas convencionais de filtragem (TEODORESCU *et al.*, 2011), caracterizando uma resposta insatisfatória do SRF-PLL.

2.4 DSOGI-PLL

Com o objetivo de corrigir a deficiência do SRF-PLL, deve-se buscar uma nova técnica para a geração de sinais em quadratura. Rodriguez *et al.* (2006) sugere o *Dual Second Order Generalized Integrator* (DSOGI-PLL), pela simplificação proporcionada pelo método, o qual pode ser visto como uma expansão da topologia anterior.

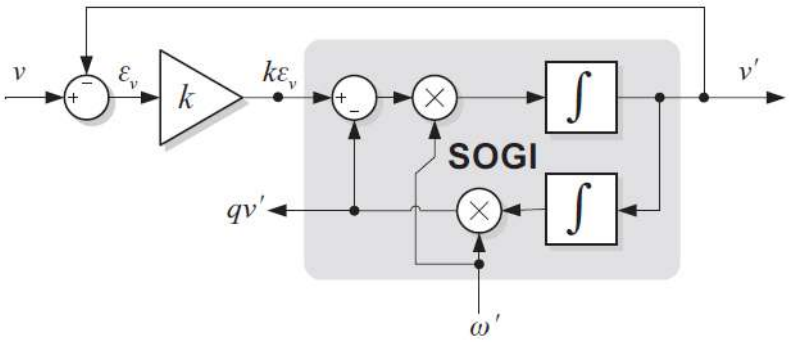

Figura 2.10: SOGI (Second Order Generalized Integrator)
Fonte: (TEODORESCU et al, 2011)

Aqui, faz-se uso de filtros adaptativos, baseados em integradores generalizados de segunda ordem, como o da figura 2.10. Este modelo apresenta a seguinte função de transferência:

$$SOGI(s) = \frac{v'}{ke_v}(s) = \frac{\omega' s}{s^2 + \omega'^2} \tag{2.28}$$

A qual, por sua vez, é desmembrada em outras duas funções de transferência de malha fechada de segunda ordem:

$$D(s) = \frac{v'}{v}(s) = \frac{k\omega' s}{s^2 + k\omega' s + \omega'^2} \tag{2.29}$$

$$Q(s) = \frac{qv'}{v}(s) = \frac{k\omega'^2}{s^2 + +k\omega' s + \omega'^2} \tag{2.30}$$

A função de transferência $D(s)$ funciona como um filtro passa-faixa, enquanto $Q(s)$ funciona como um filtro passa-baixas. Portanto, o DSOGI-PLL apresenta a capacidade de atenuar as componentes harmônicas de alta frequência. "A característica de passa-faixa sugere que é possível extrair uma componente particular na frequência de interesse ω', mesmo que o sinal de entrada esteja afetado por distorções"(TEODORESCU *et al.*, 2011, p. 74).

Por fim deve-se ressaltar que, independentemente de k ou ω', v' está sempre 90 graus a frente de qv', o que permite a geração dos sinais em quadratura (RODRIGUEZ *et al.*, 2006).

Antes dos sinais passarem pela transformada de Park, faz-se o cálculo da sequência positiva das componentes α e β, como mostrado na figura 2.11:

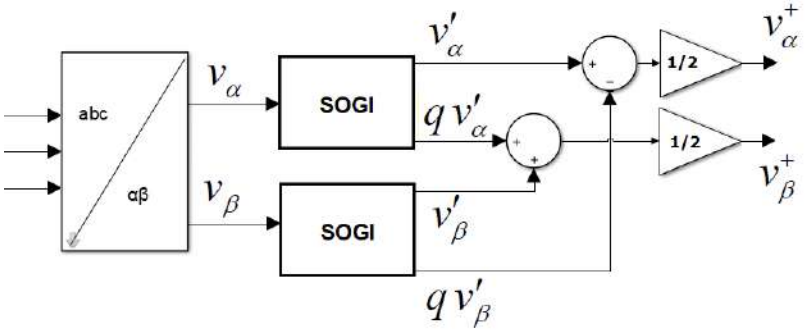

Figura 2.11: Calculadora de sequência positiva

Sendo assim, a utilização do SOGI para a geração de sinais em quadratura possibilita um sistema que calcule as componentes de sequência positiva independentemente da posição angular gerada na saída do sistema:

$$v^+_\alpha = \frac{1}{2}(v'_\alpha - qv'_\beta) \tag{2.31}$$

$$v^+_\beta = \frac{1}{2}(qv'_\alpha + v'_\beta) \tag{2.32}$$

Após o cálculo das sequências positivas, utiliza-se a estrutura conhecida do SRF-PLL. Apesar de qualquer desequilíbrio de fases que possa haver nas tensões *abc* de entrada, os sinais a serem processados pela transformada de Park serão sempre equilibrados, dado que já são as componentes de sequência positiva de α e β. A configuração final do DSOGI-PLL é apresentada na figura 2.12, na qual os fatores de 1/2 encontram-se inclusos nos blocos *SOGI-QSG*:

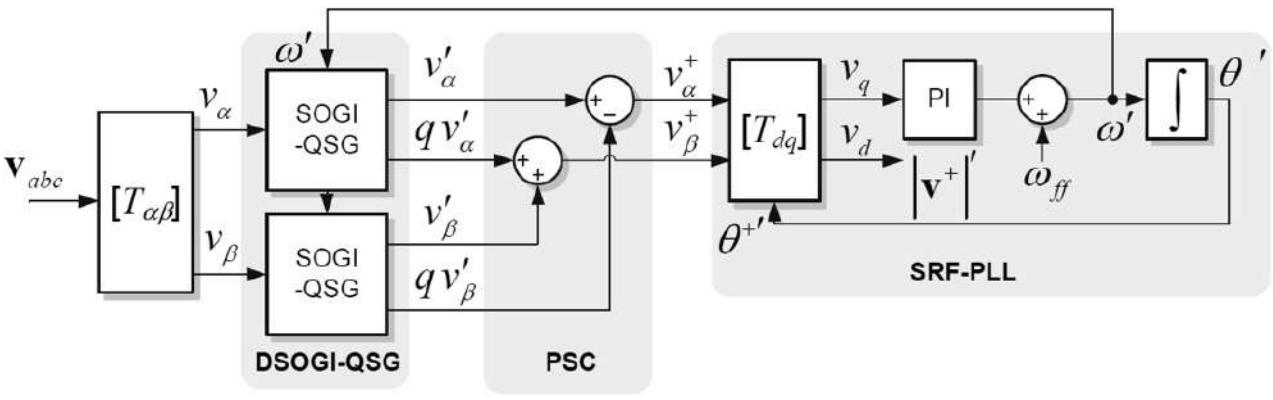

Figura 2.12: DSOGI-PLL
Fonte: (RODRIGUEZ et al, 2006)

2.4.1 Simulações do DSOGI-PLL

As mesmas pertubações da seção anterior foram aplicadas, também em ambiente *Simulink*, ao DSOGI-PLL, como forma de evidenciar o aperfeiçoamento nos resultados obtidos. A figura 2.13 apresenta o diagrama implementado no *software*.

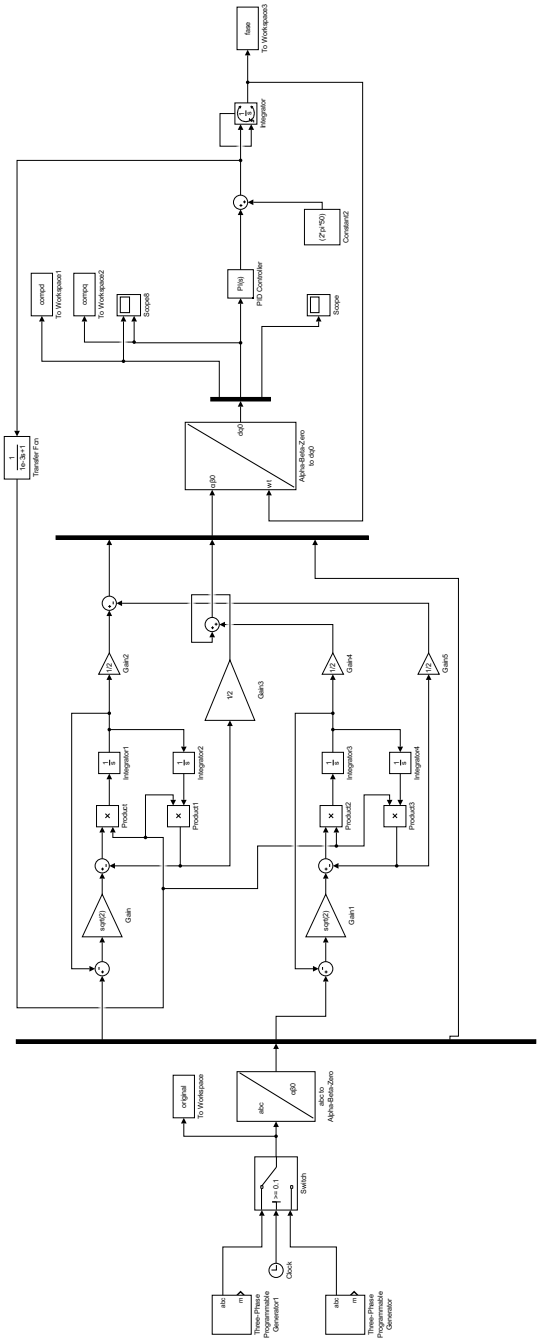

Figura 2.13: Diagrama de blocos do DSOGI-PLL implementado no *Simulink*

- Caso 1: Tensão trifásica de entrada equilibrada

Intervalo de Tempo (s)	Tensão de Entrada
0 a 0,2	v^{+1}

Tabela 2.4: DSOGI-PLL, Caso 1 - Condições de simulação

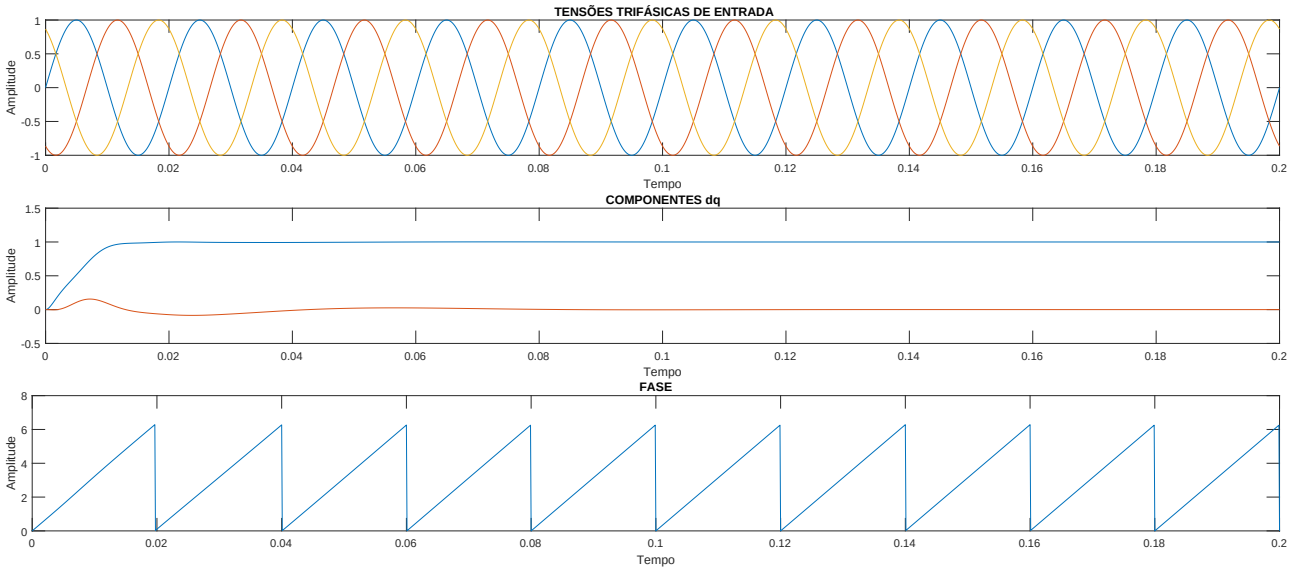

Figura 2.14: Caso 1 - Componentes *abc*, *dq* e fase

- Caso 2: Curto-circuito trifásico com harmônicas de quinta ordem

Intervalo de Tempo (s)	Tensão de Entrada
0 a 0,1	v^{+1}
0,1 a 0,2	$0,75v^{+1} + 0,075v^{-5}$

Tabela 2.5: DSOGI-PLL, Caso 2 - Condições de simulação

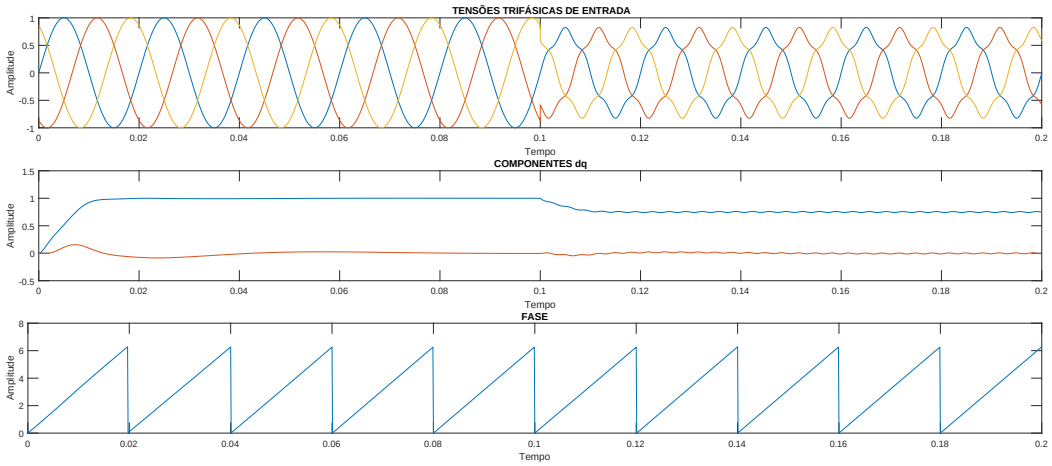

Figura 2.15: Caso 2 - Componentes *abc*, *dq* e fase

- Caso 3: Curto-circuito fase-fase

Intervalo de Tempo (s)	Tensão de Entrada
0 a 0,1	v^{+1}
0,1 a 0,2	$0,75v^{+1} + 0,25v^{-1}$

Tabela 2.6: DSOGI-PLL, Caso 3 - Condições de simulação

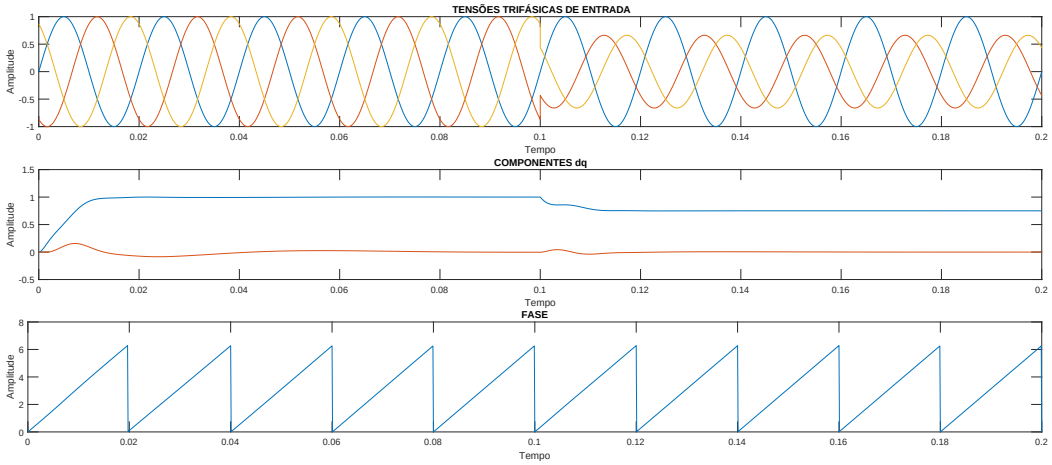

Figura 2.16: Caso 3 - Componentes *abc*, *dq* e fase

A figura 2.15 demonstra a atuação do filtro SOGI perante harmônicas, as quais tem seu efeito praticamente eliminado na componentes d e q. De forma similar, a figura 2.16 exibe como esta topologia calcula eficazmente as componentes α e β de sequência positiva. Ao fazer isso, mantém-se um cálculo preciso da fase, mesmo em situações de desequilíbrio.

CAPÍTULO 3

MATERIAIS E MÉTODOS

Neste capítulo são apresentados os materiais necessários e as etapas desenvolvidas para a aquisição dos dados desejados. Inicialmente, descrevem-se os passos necessários para a implementação digital da técnica de controle DSOGI-PLL, assim como os códigos elaborados para as diferentes etapas de teste. Por fim, nas seções 3.5 e 3.6, é caracterizada a montagem final do experimento para obtenção dos dados por meio da plataforma de prototipagem eletrônica *Arduino Due*.

3.1 FUNÇÕES DE TRANSFERÊNCIA DO DSOGI-PLL

A fim de se analisar e dimensionar o sistema, é necessário ter uma função de transferência que o represente da entrada à saída. Esta função é denominada função de transferência de malha fechada.

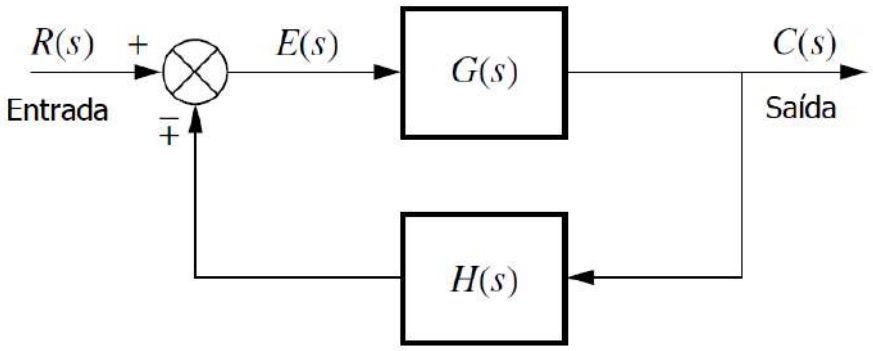

Figura 3.1: Diagrama de blocos de sistema em malha fechada
Fonte: (Nise, 2011)

Segundo Nise (2011), para um diagrama de blocos como o da figura 3.1, define-se a função de transferência de malha fechada como:

$$G_e(s) = \frac{G(s)}{1 \pm G(s)H(s)} \tag{3.1}$$

Tratando-se de configurações mais complexas como o DSOGI-PLL, deve-se analisar individualmente cada uma das malhas fechadas que compõe o sistema. Neste caso, as malhas fechadas são os dois filtros SOGI, do Gerador de Sinais em Quadratura, e o próprio SRF-PLL, composto pelo controlador PI em cascata com o integrador.

3.1.1 Gerador de Sinais em Quadratura

Como definido no capitulo 2, os filtros SOGI são caracterizados pelas seguintes funções de transferência de malha fechada:

$$D(s) = \frac{k\omega' s}{s^2 + k\omega' s + \omega'^2} \tag{3.2}$$

$$Q(s) = \frac{k\omega'^2}{s^2 + +k\omega' s + \omega'^2} \tag{3.3}$$

Estas funções de transferência de segunda ordem podem ser reescritas de forma normalizada:

$$D(s) = \frac{2\zeta\omega_n s}{s^2 + 2\zeta\omega_n s + \omega_n^2} \tag{3.4}$$

$$Q(s) = \frac{2\zeta\omega_n^2}{s^2 + +2\zeta\omega_n s + \omega_n^2} \tag{3.5}$$

Onde: ζ é o fator de amortecimento do sistema e ω_n sua frequência natural.

A tabela 3.1 apresenta os valores definidos para os parâmetros:

	Parâmetro	Valor
Ganho do SOGI	k	$\sqrt{2}$
Frequência central (rad/s)	ω'	$2\pi 60$

Tabela 3.1: Ganhos do filtro SOGI

Por meio da normalização, percebe-se que $\omega' = \omega_n$ e $k = 2\zeta$. Um ganho $k = \sqrt{2}$ implica em um fator de amortecimento $\zeta = 1/\sqrt{2}$, o que, por sua vez, corresponde a uma boa relação entre tempo de amortecimento e amplitude de sobressinal na resposta dinâmica (TEODORESCU *et al.*, 2011).

Quanto à frequência central do filtro ω', espera-se que ela esteja sincronizada com a frequência da rede.

3.1.2 SRF-PLL

Na malha do SRF-PLL, utiliza-se um controlador PI com o propósito de aumentar o tipo do sistema e, assim, reduzir o erro de regime permanente a zero:

$$C(s) = P + \frac{I}{s} \tag{3.6}$$

Onde: P é o ganho proporcional do controlador e I seu ganho integral.

Em seguida, tem-se o Gerador de Frequência e Fase, composto apenas por um integrador:

$$F(s) = \frac{1}{s} \tag{3.7}$$

Aplicando-se a equação 3.1 ao sistema composto por estes dois blocos, chega-se à função de transferência de malha fechada do SRF-PLL:

$$H_\theta(s) = \frac{Ps + I}{s^2 + Ps + I} \tag{3.8}$$

A qual é reescrita em sua forma normalizada como:

$$H_\theta(s) = \frac{2\zeta\omega_n s + \omega_n^2}{s^2 + 2\zeta\omega_n s + \omega_n^2} \tag{3.9}$$

Onde: ζ é o fator de amortecimento do sistema e ω_n sua frequência natural.

Pelas mesmas razões descritas na seção 3.1.1, $\zeta = 1/\sqrt{2}$. Já a frequência natural do sistema ω_n representa a frequência de corte do controlador PI. Baseando-se em Rodriguez et al (2006), escolheu-se $\omega_n = 2\pi 12,5$. A partir desta definição, os ganhos do controlador podem ser calculados. Ressalta-se que os mesmos devem ser compensados em relação à amplitude E_m do sinal de entrada:

$$P = \frac{2\zeta\omega_n}{E_m} \tag{3.10}$$

$$I = \frac{\omega^2}{E_m} \tag{3.11}$$

3.2 TRANSFORMAÇÃO BILINEAR

Toda a implementação do DSOGI-PLL será feita por meio de *software*, o que implica no processamento de sinais digitais. A conversão de um sinal analógico, que vem da placa de condicionamento, para um sinal digital passa pela amostragem desse sinal. "O fato dos sinais serem amostrados em intervalos específicos de tempo, e manterem este valor durante aquele intervalo, faz com que a performance do sistema mude com mudanças na taxa de amostragem" (NISE, 2011, p.728).

A modelagem matemática de sinais amostrados demanda uma abordagem diferente daquela utilizada para sinais contínuos. Funções de transferência que lidam com sinais amostrados são melhor representadas no plano complexo denominado plano Z. Este plano possui uma região de estabilidade diferente da do plano S (domínio da frequência).

Neste ponto, as transformações bilineares surgem como uma ferramenta fundamental na análise de sistemas digitais. Pode-se analisar as funções de transferência do sistema no domínio da frequência, o qual é mais familiar, como se os sinais de entrada fossem contínuos. Posteriormente, por meio da transformação bilinear, estas funções são levadas para o plano Z.

Segundo Nise (2011), a transformação de Tustin é uma transformação bilinear que resulta em um função de transferência digital capaz de preservar, nos instantes de amostragem, a resposta de um sistema contínuo:

$$s = \frac{2}{dt}\frac{(z-1)}{(z+1)} \tag{3.12}$$

Onde: dt é o intervalo de amostragem do sinal em questão.

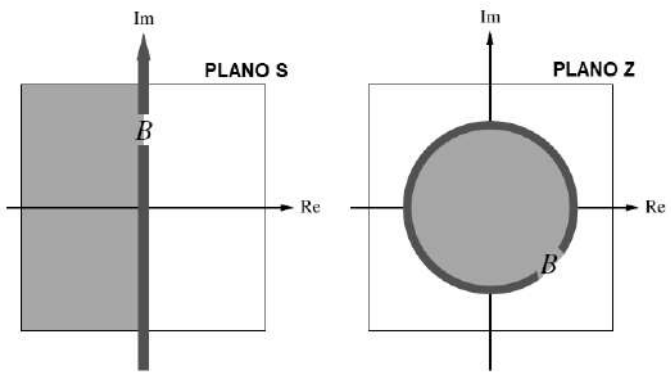

Figura 3.2: Planos S e Z, com regiões de estabilidade em destaque
Fonte: (Nise, 2011)

Como resultado da aplicação da transformação de Tustin, tem-se uma variável que vai do plano S para o plano Z, como mostrado na figura 3.2.

Contudo, para viabilizar a implementação por *software*, as funções de transferência precisam estar representadas no tempo discreto. Para que isso ocorra, são utilizadas propriedades da transformada Z, onde n representa a amostra utilizada no tempo:

Linearidade	$z(af(n)) = aF(z)$
Translação no tempo	$z(f[n-k]) = z^{-k}F(z)$

Tabela 3.2: Propriedades da transformada Z, (Nise, 2010)

A transformada Z permite que uma função no plano Z seja representada no domínio do tempo discreto. Essa representação permitirá ao microcontrolador ler e executar tais funções para cada uma das amostras processadas.

A seguir, detalhe-se a aplicação das transformadas para cada função de transferência presente no sistema. Deseja-se definir uma saída $y(n)$ para cada um dos blocos do DSOGI-PLL.

3.2.1 Controlador PI

$$C(s) = P + \frac{I}{s} \tag{3.13}$$

Aplicando-se a transformação de Tustin:

$$C(z) = \frac{Y(z)}{X(z)} = P + \frac{Idt(z+1)}{2(z-1)} \tag{3.14}$$

A equação é rearranjada multiplicando-se por z^{-1} no numerador e no denominador:

$$\frac{Y(z)}{X(z)} = \left[P + \frac{Idt(z+1)}{2(z-1)}\right]\frac{z^{-1}}{z^{-1}} = \frac{2P(1-z^{-1}) + Idt(1+z^{-1})}{2(1-z^{-1})} \tag{3.15}$$

$$Y(z) = \frac{2Y(z)z^{-1} + 2PX(z) - 2PX(z)z^{-1} + IdtX(z) + IdtX(z)z^{-1}}{2} \tag{3.16}$$

Por fim, aplicando-se as propriedades da transformada Z:

$$y(n) = \frac{2y(n-1) + (2P + Idt)x(n) + (Idt - 2P)x(n-1)}{2} \tag{3.17}$$

3.2.2 Integrador

$$F(s) = \frac{1}{s} \tag{3.18}$$

Aplicando-se a transformação de Tustin:

$$F(z) = \frac{Y(z)}{X(z)} = \frac{dt(z+1)}{2(z-1)} \tag{3.19}$$

A equação é rearranjada multiplicando-se por z^{-1} no numerador e no denominador:

$$\frac{Y(z)}{X(z)} = \left[\frac{dt(z+1)}{2(z-1)}\right]\frac{z^{-1}}{z^{-1}} = \frac{dt(1+z^{-1})}{2(1-z^{-1})} \tag{3.20}$$

$$Y(z) = \frac{2Y(z)z^{-1} + dtX(z) + dtX(z)z^{-1}}{2} \tag{3.21}$$

Por fim, aplicando-se as propriedades da transformada Z:

$$y(n) = \frac{2y(n-1) + dtx(n) + dtx(n-1)}{2} \tag{3.22}$$

3.2.3 SOGI

- $D(s)$:

$$D(s) = \frac{k\omega' s}{s^2 + k\omega' s + \omega'^2} \tag{3.23}$$

Aplicando-se a transformação de Tustin:

$$D(z) = \frac{Y(z)}{X(z)} = \frac{k\omega' \dfrac{2}{dt}\dfrac{(z-1)}{z+1}}{\left[\dfrac{2}{dt}\dfrac{(z-1)}{z+1}\right]^2 + k\omega'\left[\dfrac{2}{dt}\dfrac{(z-1)}{z+1}\right] + \omega'^2} \tag{3.24}$$

Repete-se a metodologia aplicada nas seções 3.41 e 3.42: multiplica-se por z^{-1} no numerador e no denominador e aplica-se as propriedades da transformada Z:

$$y(n) = \frac{-(4 - C1 + C2)y(n-2) - (2C2 - 8)y(n-1) + C1x(n) - C1x(n-2)}{(4 + C1 + C2)} \tag{3.25}$$

Onde:

$$C1 = 2\omega(n-1)\sqrt{(2)}dt$$
$$C2 = (\omega(n-1))^2 dt^2\sqrt{2}$$

- $Q(s)$:

$$Q(s) = \frac{k\omega'^2}{s^2 + +k\omega' s + \omega'^2} \tag{3.26}$$

Aplicando-se a transformação de Tustin:

$$D(z) = \frac{Y(z)}{X(z)} = \frac{k\omega'^2}{\left[\frac{2}{dt}\frac{(z-1)}{z+1}\right]^2 + k\omega'\left[\frac{2}{dt}\frac{(z-1)}{z+1}\right] + \omega'^2} \tag{3.27}$$

Repete-se a metodologia aplicada nas seções 3.41 e 3.42: multiplica-se por z^{-1} no numerador e no denominador e aplica-se as propriedades da transformada Z:

$$y(n) = \frac{C3x(n) + 2C3x(n-1) + C3x(n-2) - (2C2 - 8)y(n-1) - (4 - C1 + C2)y(n-2)}{4 + C1 + C2} \tag{3.28}$$

Onde:

$$C1 = 2\omega(n-1)\sqrt{(2)}dt$$
$$C2 = (\omega(n-1))^2 dt^2\sqrt{2}$$
$$C3 = (\omega(n-1))^2 dt^2$$

3.3 CÓDIGO *MATLAB*

Primeiramente, o DSOGI-PLL foi implementado em *MATLAB*, pela praticidade proporcionada pelo *software*.

O código baseia-se na leitura de um arquivo texto, que contém as amostras de tensão trifásica. O número de amostras no arquivo e o tempo de simulação do código devem respeitar a taxa de amostragem escolhida. Neste caso, decidiu-se trabalhar com uma taxa de amostragem de 6000 Hz. Consequentemente, gerou-se um arquivo com 6000 amostras e definiu-se o tempo de simulação de 01 segundo.

As variáveis de interesse são salvas em vetores de 6000 posições de forma que seu comportamento ao longo do tempo possa ser observado.

O Apêndice A contém um exemplo de como o arquivo texto deve ser estruturado, enquanto o Apêndice B apresenta o código escrito em *MATLAB*.

3.4 CÓDIGO EM C

Após a elaboração do código *MATLAB*, o mesmo foi reestruturado para a linguagem de programação *C*. A linguagem *C* é extremamente difundida entre os diversos tipos de microcontroladores, incluindo o *Arduino* e sistemas DSP.

O LQEE utiliza o kit de controle ezDSP f28335, que por sua vez, baseia-se em um microprocessador de ponto flutuante DSP. Sendo assim, pensando-se em sua utilização futura no laboratório, o código elaborado em *C* foi escrito com operações de ponto flutuante.

Ressalta-se que a implementação deste código em um Arduino, como feita neste trabalho, certamente custará uma quantidade significativa de tempo de processamento, uma vez que o mesmo não possui uma unidade dedicada a operações de ponto flutuante, realizando-as por meio de *software*. Contudo, apesar do custo de processamento, esta implementação mostra-se válida para confirmar o funcionamento do DSOGI-PLL.

Neste código, cada um dos blocos do diagrama do DSOGI-PLL foi transformado em uma função. Segundo Kernighan & Ritchie (2017), a linguagem *C* foi criada de modo a incentivar o uso de funções, as quais não apenas descomplicam o entendimento do programa, como também

facilitam alterações futuras no mesmo.

Assim como o código anterior, trabalha-se com base na leitura de um arquivo texto e uma taxa de amostragem de 6000 Hz.

O código escrito em *C* encontra-se no Apêndice C.

3.5 ESTRUTURA LABORATORIAL

Utilizou-se a estrutura desenvolvida no Laboratório de Qualidade de Energia Elétrica (LQEE) da Universidade de Brasília (UnB) para a aquisição dos sinais de tensão da rede elétrica.

O sinal de tensão trifásico a ser monitorado foi gerado por meio da bancada existente no Laboratório de Conversão de Energia, caracterizada conforme a tabela 3.3:

Bancada de Tensão Ajustável	
Fabricante	Equacional
Tensão de Linha (V)	220
Tensão de Fase (V)	127

Tabela 3.3: Características da bancada de tensão ajustável

A medição da tensão AC da rede é feita por meio de um transdutor, caracterizado na tabela 3.4,o qual opera por meio de sensores de efeito hall. Utiliza-se um para cada fase. Este dispositivo será a interface entre a rede de alta tensão e um circuito eletrônico. Na saída do transdutor, tem-se uma corrente, em miliampère, proporcional à tensão medida, em Volts.

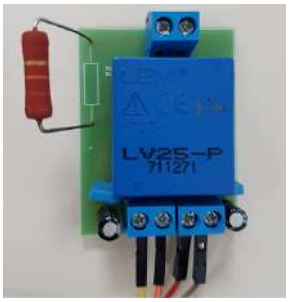

Figura 3.3: Transdutor de tensão

Modelo	LV 25-P
Tensão de Entrada (V)	10 a 1500
Tensão de Alimentação (V)	±15

Tabela 3.4: Características do transdutor

A tensão de alimentação de ±15 V foi gerada por uma fonte do fabricante Minipa, modelo *MPL-3303M*.

Os sinais de corrente gerados pelos transdutores são, então, levados à uma placa de condicionamento de sinais analógicos, construída de acordo com o diagrama esquemático da figura 3.4 (SILVEIRA, 2016). A placa converte o sinal de entrada em uma saída senoidal que varia de 0 V a 3,3 V. Dessa forma, um sinal de alta tensão variando entre $-127\sqrt{2}$ V e $127\sqrt{2}$ V passa a ser representado por um sinal de tensão variando entre 0 V e 3,3 V, o que permite o seu processamento por um microcontrolador.

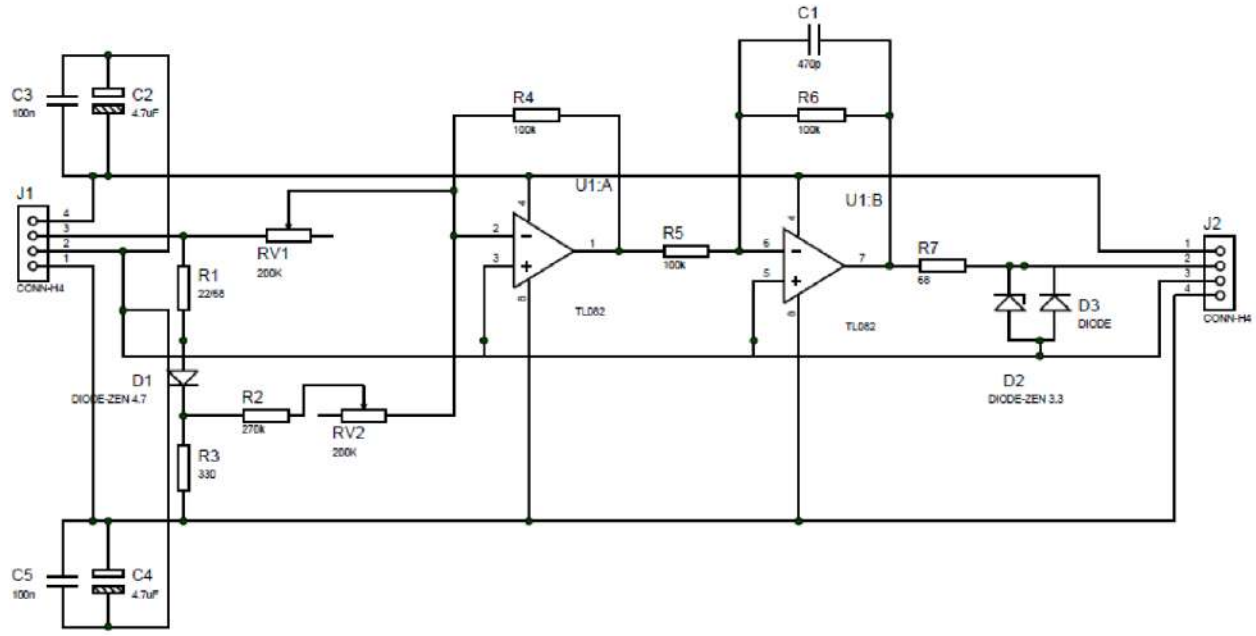

Figura 3.4: Diagrama esquemático da placa de condicionamento de sinais
Fonte: (Silveira, 2016)

3.6 CÓDIGO DE IMPLEMENTAÇÃO NO ARDUINO DUE

O *Arduino Due* é uma plataforma de prototipagem eletrônica que utiliza um microcontrolador ARM de 32 bits. Esta placa conta com doze entradas analógicas, das quais três serão utilizadas para receber o sinal trifásico. Há também duas portas 'Digital para Analógico' (DAC), que serão utilizadas como saída das variáveis de interesse do usuário.

Além de altamente acessível, o *Arduino* foi escolhido por sua interface de programação amigável e por seu extenso material de referência disponível, tanto pelo fabricante, quanto

pelos inúmeros usuários da plataforma.

A linguagem de programação da placa é baseada nas linguagens C e C++. Portanto, o código em C descrito na seção 3.4 pode ser utilizado. Alterações serão efetuadas para adaptá-lo ao *software* da placa, o *open-source Arduino Software (IDE)*, além de comandos adicionais para configuração do *hardware*.

O código de implementação no *Arduino Due* encontra-se no Apêndice D.

CAPÍTULO 4

APRESENTAÇÃO E ANÁLISE DE RESULTADOS

Neste capítulo são apresentados os resultados obtidos a partir da montagem laboratorial detalhada na seção 3.5. Implementou-se o DSOGI-PLL em um *Arduino Due* por meio do código transcrito no Apêndice D.

Na aquisição de resultados, investiga-se, primeiramente, o tempo de processamento necessário para tratar uma amostra. Em seguida, é feita a leitura das portas DAC0 e DAC1 do *Arduino Due*.

As seguintes variáveis foram registradas simultaneamente:

- Fase a da tensão de entrada e θ'
- Componentes alfa e beta de sequência positiva: α_+ e β_+
- Fase a da tensão de entrada e α_+
- Fase a da tensão de entrada e componente d
- Fase a da tensão de entrada e componente q

Para a visualização dos dados, foi utilizado o osciloscópio *54621A*, do fabricante *Agilent*.

4.1 TEMPO DE PROCESSAMENTO

O tempo de processamento gasto pelo *Arduino* é um limitador do intervalo de amostragem definido pelo usuário. Naturalmente, deve-se colher uma nova amostra apenas quando a anterior já tiver sido processada.

Como descrito por Lathi (1998), para que a informação do sinal analógica seja preservada em um sinal amostrado, o teorema da amostragem de Nyquist deve ser respeitado. Este teorema afirma que:

$$f_s > 2B \tag{4.1}$$

Onde: f_s é taxa de amostragem necessária e B a largura de banda do sinal.

Considerando-se que a componente fundamental da tensão da rede elétrica está em 60 Hertz, deve-se utilizar uma taxa de amostragem mínima de 120 Hertz.

O código do Apêndice D mantém a saída digital *13* do *Arduino Due* em *HIGH* durante toda a implementação do DSOGI-PLL. A figura 4.1 apresenta a forma de onda obtida nessa porta.

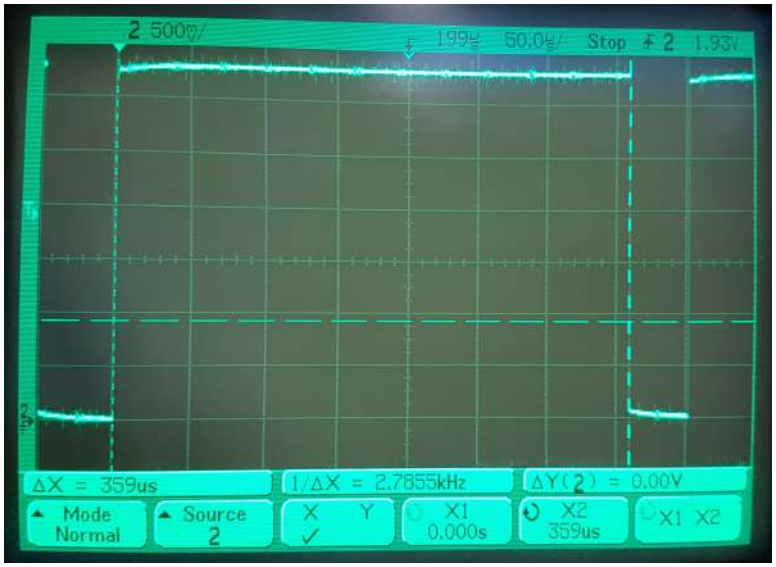

Figura 4.1: Tempo de processamento do DSOGI-PLL no *Arduino Due*

O osciloscópio exibe um tempo de processamento de $359\mu s$. Alerta-se para o fato de que ao se repetir o procedimento, este valor não se manteve constante. O tempo de processamento de $359\mu s$ foi obtido com mais frequência, entretanto foram registrados também tempos variando entre 300 e $380\mu s$.

O código implementado utilizou funções próprias do *open-source Arduino Software (IDE)*, como a função *map()*, que efetua a quantização da amostra. Recomenda-se substituir esta função por comandos que efetuem a mesma tarefa, em busca de tempos de processamento mais precisos.

De forma a garantir um tempo suficiente para a implementação do DSOGI-PLL, definiu-se um intervalo de amostragem de $400\mu s$, o qual implica uma taxa de amostragem de $2500Hz$.

4.2 VARIÁVEIS ADQUIRIDAS

4.2.1 Fase *a* da tensão de entrada e θ'

Primeiramente, deseja-se observar a resposta do sistema em relação a sua entrada. Portanto, a saída θ' do DSOGI-PLL foi plotada juntamente com a fase a da tensão de entrada, como mostrado na figura 4.2. Ressalta-se que esta fase a já é o sinal gerado pelo *Arduino* após a conversão analógico para digital.

A posição angular θ' varia linearmente de 0 a 2π, formando uma onda dente de serra, como previsto pelas simulações apresentadas na Fundamentação Teórica.

Figura 4.2: Osciloscópio - Fase a da tensão de entrada e θ'

Mediu-se o atraso entre o pico da fase a e o início da rampa de θ' e o tempo de $520\mu s$ foi obtido. Resultado esperado, uma vez que apenas para processar a amostra são necessários $359\mu s$.

4.2.2 Componentes alfa e beta de sequência positiva

As componentes α_+ e β_+ também foram analisadas, conforme figura 4.3.

Pode-se notar uma definição melhor da senoide, especialmente em seus picos e vales. Este efeito é uma consequência da utilização dos filtros SOGI, que eliminaram quaisquer distorções

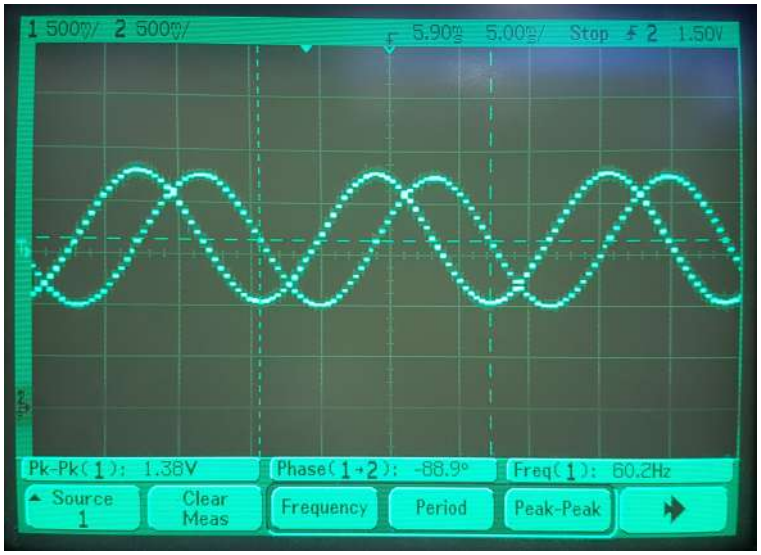

Figura 4.3: Osciloscópio - α_+ e β_+

presentes na rede durante a medição.

Plotou-se também o sinal α_+ em relação à fase a, com o intuito de ratificar a conservação da amplitude e calcular o atraso entre as ondas:

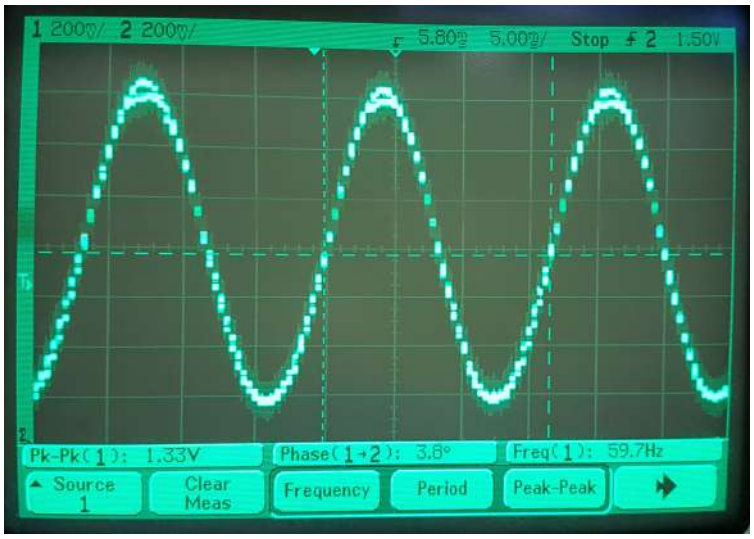

Figura 4.4: Osciloscópio - Fase a da tensão de entrada e α_+

Por meio da observação da figura 4.4, percebe-se a senoide de α_+ melhor definida, como ressaltado anteriormente. No topo esquerdo da tela, é possível notar que ambos os canais estão configurados em $200mV$ por divisão, confirmando que os sinais possuem a mesma amplitude. O atraso entre as duas ondas foi de $180\mu s$, que pode ser interpretado como o tempo necessário para a implementação da transformada de Clarke até a calculadora de sequência positiva.

4.2.3 Componentes d e q

Por fim, foram plotadas as componentes d e q geradas pela transformada de Park:

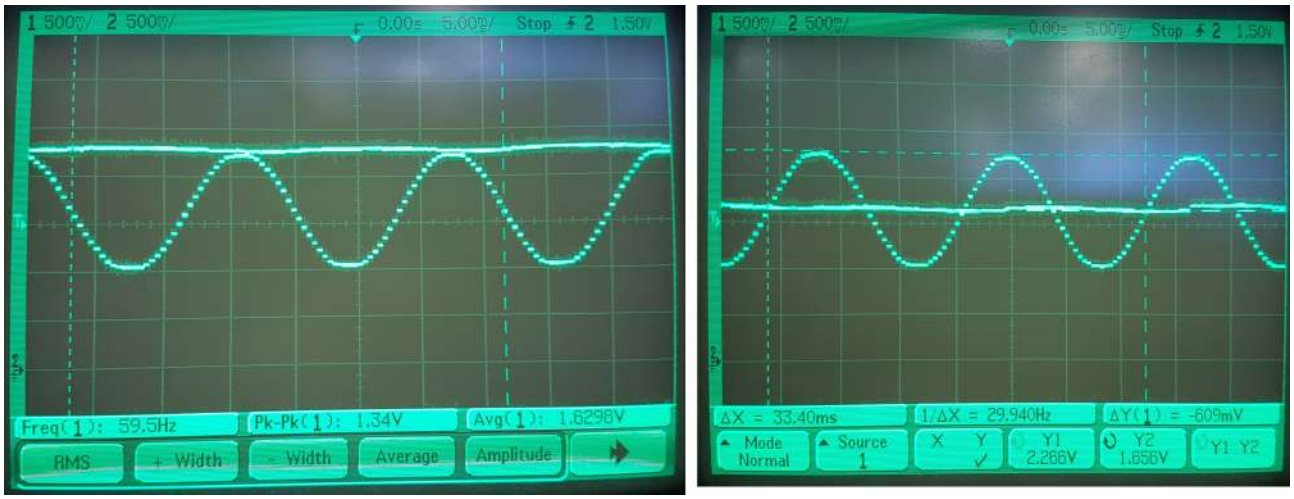

Figura 4.5: Osciloscópio - Componentes d e q

Percebe-se pela figura 4.5 que as componentes d e q apresentam o comportamento esperado: q é mantida em zero, enquanto d segue a amplitude das componentes de sequência positiva, que, no caso equilibrado, é também a própria amplitude das tensões abc de entrada.

CAPÍTULO 5

CONCLUSÃO

Este trabalho apresentou os resultados da implementação digital do sistema de controle denominado *Dual Second Order Generalized Integrator* (DSOGI-PLL). Como entrada do sistema, utilizou-se a tensão de linha trifásica de 220 V disponibilizada pela bancada de tensão ajustável do Laboratório de Conversão de Energia da UnB. Por meio de um algoritmo executado pela plataforma de prototipagem eletrônica *Arduino Due*, a fase da tensão de entrada foi devidamente monitorada. Outras variáveis calculadas pelo sistema foram também adquiridas, atestando a implementação bem-sucedida do DSOGI-PLL. São elas: componentes α e β de sequência positiva, α_+ e β_+, e as componentes d e q, resultantes da transformada de Park.

Destaca-se também, no capítulo 2, as simulações efetuadas no software *Simulink*. Duas topologias de PLL foram comparadas: o *Synchronous Reference Frame* (SRF-PLL) e o DSOGI-PLL. Por meio destas simulações pôde-se ratificar a maior robustez do último, representada por duas características principais: Sua capacidade de bloquear harmônicas e sua adaptabilidade a desequilíbrios de fase.

No capítulo 3 foram descritos os materiais e métodos aplicados neste trabalho. Apresentou-se toda a modelagem matemática necessária para a utilização do sistema no domínio do tempo discreto. Em sequência, detalhou-se a estrutura laboratorial necessária para a aquisição de dados pelo *Arduino Due*, bem como o código desenvolvido para execução do DSOGI-PLL nesta plataforma.

Apesar de respeitar o teorema de Nyquist com larga vantagem, acredita-se que a taxa de amostragem obtida de 2500 Hz pode ser melhorada. Neste sentido, destacam-se nesta seção duas estratégias principais para futuras otimizações:

- Substituição da função *map()* do *Arduino*. Comandos que implementem a quantização das amostras talvez sejam mais rápidos que a função disponibilizada pelo *open-source Arduino Software (IDE)*;

- Utilização de *Look up Tables* senoidais para substituir as funções seno e cosseno. *Look up Tables* são conjuntos de dados pré-definidos que podem representar as senoides por meio de uma interpolação linear. De forma semelhante ao item anterior, almeja-se ganhar tempo de processamento empregando-se este método.

Por fim, espera-se que os resultados apresentados sejam de valia para o Laboratório de Qualidade de Energia Elétrica. O conversor de potência presente no LQEE faz uso do kit de controle *ezDSP f28335*. Portanto, sugere-se, como trabalho futuro, a configuração da placa deste kit para a implementação do código em C aqui apresentado.

REFERÊNCIAS BIBLIOGRÁFICAS

ABEEOLICA. Boletim anual de geração 2017. Associação Brasileira de Energia Eólica, 2018. Citado na página 2.

BOBROWSKA-RAFAL, M.; RAFAL, K.; JASINSKI, M.; KAZMIERKOWSKI, M. Grid synchronization and symmetrical components extraction with pll algorithm for grid connected power electronic converters-a review. *Bulletin of the Polish Academy of Sciences: Technical Sciences*, v. 59, n. 4, 2011. Citado na página 3.

CHUNG, S.-K. A phase tracking system for three phase utility interface inverters. *IEEE Transactions on Power Electronics*, v. 15, n. 3, MAY 2000. Citado na página 3.

GLOVER, J. D.; SARMA, M. S.; OVERBYE, T. *Power System Analysis & Design*. [S.l.]: Cengage Learning, 2012. Citado na página 5.

KERNIGHAN, B.; RITCHIE, D. M. *The C programming language*. [S.l.]: Prentice hall, 2017. Citado na página 27.

LATHI, B. P. *Control Systems Engineering*. [S.l.]: Oxford university press, 1998. Citado na página 31.

MME. Resenha energética 2018 - ano base 2017. Ministério de Minas e Energia, 2018. Citado na página 2.

NISE, N. S. *Control Systems Engineering*. [S.l.]: John Wiley & Sons, Inc, 2011. Citado 3 vezes nas páginas 20, 21, and 23.

RODRIGUEZ, P.; TEODORESCU, R.; CANDELA, I.; TIMBUS, A.; LISERRE, M.; BLAABJERG, F. New positive-sequence voltage detector for grid synchronization of power converters under faulty grid conditions. *Power Electronics Specialists Conference, 2006*, 2006. Citado 3 vezes nas páginas 3, 11, and 15.

SILVEIRA, J. P. C. Avaliação de distorções harmônicas e inter-harmônicas em um sistema de conversão de energia eólica a geração síncrona. 2016. Citado na página 29.

TEODORESCU, J. R.; LISERRE, M.; RODRíGUEZ, P. *Grid Converters for Photovoltaic and Wind Power Systems*. [S.l.]: John Wiley & Sons, Ltd, 2011. Citado 7 vezes nas páginas 2, 6, 9, 11, 12, 14, and 15.

APÊNDICE A

EXEMPLO DE ARQUIVO TEXTO

O arquivo texto para leitura pelos códigos *MATLAB* e *C* deve possuir a estrutura que se segue. Uma coluna por fase, contendo os valores de amplitude da tensão de entrada, v_a, v_b e v_c:

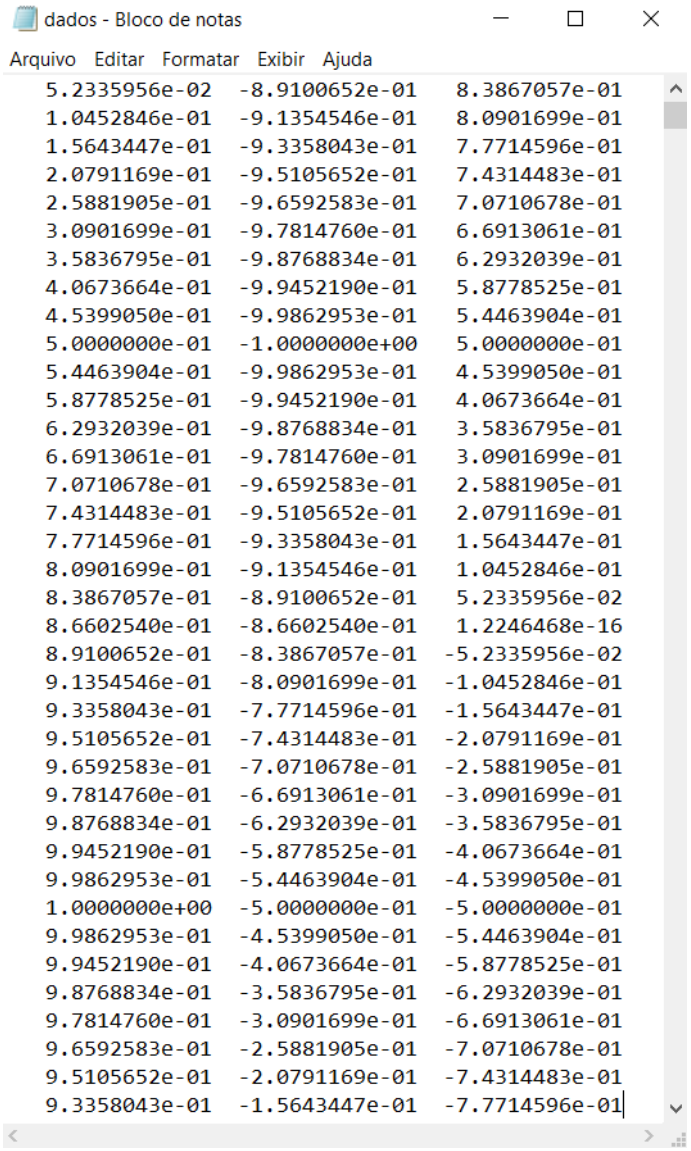

Figura A.1: Exemplo de arquivo texto utilizado para leitura pelos códigos *MATLAB* e *C*

APÊNDICE B

CÓDIGO *MATLAB*

Listing B.1: DSOGI-PLL implementado em *MATLAB*

```
Fs = 6000;                              % Taxa de amostragem
dt = 1/Fs;                              % amostras por segundo

End = 1;                                % Tempo de simulacao: 1s
t = (0:dt:End-dt);                      % Tempo de simulacao, definido pelos segundos
    em que tenho amostras

Vreal = load('dados.txt', '-ascii');    % Carregar arquivos de texto com 6000
   amostras
Vsim = Vreal.';                         % Armazenar dados de arquivo texto

% Transformada ABC -> Alpha-Beta
T_al_be = 2/3*[1 -1/2 -1/2; 0 sqrt(3)/2 -sqrt(3)/2; 1/sqrt(2) 1/sqrt(2) 1/sqrt
   (2)];

V_al_be = (T_al_be)*Vsim;
A = V_al_be(1,:);
B = V_al_be(2,:);

% Constante do SOGI:
k = sqrt(2);

% Constantes do controlador:

Ksi = 1/sqrt(2);                % Fator de amortecimento
Wn = 25*pi;                     % Frequencia natural

Kp = (2*Ksi*Wn)/220*sqrt(2);
Ki = (Wn*Wn)/220*sqrt(2);

%P = Kp;
%E = Kp/Ti;

K1 = 2*Kp+Ki*dt;
K2 = Ki*dt-2*Kp;

% Declaracao de condicoes iniciais (nulas):
v_al(1) = 0; v_al(2) = 0; qv_al(1) = 0; qv_al(2) = 0;
v_be(1) = 0; v_be(2) = 0; qv_be(1) = 0; qv_be(2) = 0;
w(1) = 0; w(2) = 0;
Teta(1) = 0; Teta(2) = 0;

% Comeca o Loop! Amostra por amostra!
for i = 3:length(t)

C1 = 2*w(i-1)*k*dt;
```

```
C2 = ((w(i-1))^2)*((dt)^2);
C3 = ((w(i-1))^2)*((dt)^2)*k;

v_al(i) = (C1*A(i) - C1*A(i-2) - (2*C2-8)*v_al(i-1) - (4-C1+C2)*v_al(i-2))/(4+C1+C2);
qv_al(i) = (C3*A(i) + 2*C3*A(i-1) + C3*A(i-2) - ((2*C2)-8)*qv_al(i-1) - (4-C1+C2)*qv_al(i-2))/(4+C1+C2);

v_be(i) = (C1*B(i) - C1*B(i-2) - (2*C2-8)*v_be(i-1) - (4-C1+C2)*v_be(i-2))/(4+C1+C2);
qv_be(i) = (C3*B(i) + 2*C3*B(i-1) + C3*B(i-2) - ((2*C2)-8)*qv_be(i-1) - (4-C1+C2)*qv_be(i-2))/(4+C1+C2);

Apos(i) = v_al(i)/2 - qv_be(i)/2;
Bpos(i) = qv_al(i)/2 + v_be(i)/2;

d(i) = Apos(i)*cos(Teta(i-1)) + Bpos(i)*sin(Teta(i-1));
q(i) = -Apos(i)*sin(Teta(i-1)) + Bpos(i)*cos(Teta(i-1));

Vlf(1) = 0; Vlf(2) = 0;
Vlf(i) = (2*Vlf(i-1) + K1*q(i) + K2*q(i-1))/2;

w(i) = Vlf(i) + (2*pi*50);

Teta(i) = (2*Teta(i-1) + dt*w(i) + dt*w(i-1))/2;
if Teta(i) > 2*pi
Teta(i) = 0;
end

end

% Plotar resultados:
ax1 = subplot(3,1,1);
plot(ax1,t,Vsim)
title(ax1,'Sinal Original')

ax2 = subplot(3,1,2);
plot(ax2,t,w)
title(ax2,'Frequencia')

ax3 = subplot(3,1,3);
plot(ax3,t,Teta)
title(ax3,'Fase')

figure
plot(t, Apos, t, Bpos)
title('Componentes alfa e beta de sequencia positiva')

figure
plot(t, d, t, q)
title('Componentes dq')
```

APÊNDICE C

CÓDIGO EM C

Listing C.1: DSOGI-PLL implementado em *C*

```
// **** CODIGO DE IMPLEMENTACAO DA TECNICA DE CONTROLE DSOGI-PLL ****

#include <stdio.h>
#include <math.h>
#include "transf.h"

// Prototipos das funcoes utilizadas:

int Detectar_Amostras (void);
void Armazenar_Amostras (float *Va, float *Vb, float *Vc);
void init_TClarke(typeM3x3 *M);
void Transf(typeV3x1 *X, typeM3x3 *T, typeV3x1 *Y);

typeV4x1 Quad_Signal_Generator(int k, float dt,
float *w, float *alpha, float *beta,
float *v_al, float *qv_al, float *v_be, float *qv_be);

void SeqPos_Generator(typeV4x1 *X, typeV2x1 *Y);
void TPark(int k, typeV2x1 *X, float *th, typeV2x1 *Y);
void Controlador_PI(int k, float dt, float *q, float *Vlf);
void Freq_Generator(int k, float *Vlf, float *w);
void Phase_Generator(int k, float dt, float *w, float *Theta);

main (){

int c = Detectar_Amostras();                    // Deteccao do numero de amostras
    contidas em arquivo de texto

int i = 3;                                      // Cada variavel tera tres
    posicoes. A amostra atual e duas passadas.
int k = 0;                                      // Amostra atual
int t;

float dt;                                       // Intervalo de amostragem

float  A[c], B[c], C[c];                        // Tensao de entrada
float alpha[i], beta[i], zero[i];               // Tensao alfa-beta-zero
float wold[i];                                  // Frequencia calculada
float v_al[i], qv_al[i], v_be[i], qv_be[i];     // Sinais em quadratura das
    componentes alfa e beta
float d[i], q[i];                               // Componentes d e q
float theta[i];                                 // Fase calculada
float Vlf[i];                                   // Sinal na saida do controlador
    PI

typeM3x3 TC;                                    // Tranformacao direta de Clarke
```

```
43 typeV3x1 Vabc;                                    // V abc
44 typeV3x1 VCl;                                     // V alpha, beta, 0
45 typeV4x1 Quad_Signals;                            // Sinais em quadratura das
       componentes alfa e beta
46 typeV2x1 SeqPos;                                  // V alpha, beta de sequencia
       positiva
47 typeV2x1 Vdq;                                     // V dq
48
49 Armazenar_Amostras(A, B, C);                      // Ler e armazenar amostras
       contidas em arquivo de texto
50 init_TClarke(&TC);
51
52 dt = 1.0/6000.0;                                  // Intervalo de amostragem. Taxa
       de amostragem escolhida = 6000 Hz
53
54 // ******** COMECA O LOOP! ********
55 for (t = 0; t < 6000; t++){
56
57 // Alocacao dos vetores de tensao ABC no modelo de struct 3x1
58 Vabc.A1 = A[t];
59 Vabc.A2 = B[t];
60 Vabc.A3 = C[t];
61
62 // Chamada da funcao que realiza a transformada de Clarke
63 Transf(&Vabc, &TC, &VCl);
64
65 alpha[k] = VCl.A1;
66 beta[k] = VCl.A2;
67 zero[k] = VCl.A3;
68
69 // Chamada da funcao que gera os sinais em quadratura
70 Quad_Signals = Quad_Signal_Generator(k, dt, wold, alpha, beta, v_al, qv_al, v_be
       , qv_be);
71
72 v_al[k] = Quad_Signals.A1;
73 qv_al[k] = Quad_Signals.A2;
74 v_be[k] = Quad_Signals.A3;
75 qv_be[k] = Quad_Signals.A4;
76
77 // Chamada da funcao que calcula as componentes alfa e beta de sequencia
       positiva
78 SeqPos_Generator(&Quad_Signals, &SeqPos);
79
80 // Chamada da funcao que realiza a transformada de Park
81 TPark(k, &SeqPos, theta, &Vdq);
82
83 d[k] = Vdq.A1;
84 q[k] = Vdq.A2;
85
86 // Chamada da funcao que implementa o controlador PI
87 Controlador_PI(k, dt, q, Vlf);
88
89 // Chamada da funcao que implementa o gerador de frequencia
90 Freq_Generator(k, Vlf, wold);
91
92 // Chamada da funcao que implementa o gerador de fase
93 Phase_Generator(k, dt, wold, theta);
94
95 // Imprimir variavel de interesse. Ex: frequencia
```

```
96  printf("%d  %.10f\n", k, wold[k]);
97
98  k = k + 1;
99
100 // Transformacao de Tustin requer a utilizacao dos valores de duas amostras
        passadas
101 // Ao fim de cada amostra, estes valores sao atualizados
102
103 if (k > 2){
104
105 alpha[0] = alpha[1]; alpha[1] = alpha[2];
106 beta[0] = beta[1]; beta[1] = beta[2];
107 zero[0] = zero[1]; zero[1] = zero[2];
108
109 wold[0] = wold[1]; wold[1] = wold[2];
110
111 v_al[0] = v_al[1]; v_al[1] = v_al[2];
112 qv_al[0] = qv_al[1]; qv_al[1] = qv_al[2];
113 v_be[0] = v_be[1]; v_be[1] = v_be[2];
114 qv_be[0] = qv_be[1]; qv_be[1] = qv_be[2];
115
116 d[0] = d[1]; d[1] = d[2];
117 q[0] = q[1]; q[1] = q[2];
118
119 theta[0] = theta[1]; theta[1] = theta[2];
120 Vlf[0] = Vlf[1]; Vlf[1] = Vlf[2];
121
122 k = 2;
123
124 }
125 }
126
127 return 0;
128
129 }
130
131 int Detectar_Amostras (void){
132
133 int c;                                          // Variaveis auxiliares:
        contadores
134 char contador;                                  // Identificador de nova
        linha
135
136 /*Descobrir o numero de linhas do arquivo (igual ao numero de amostras)*/
137
138 FILE *fp;                                       // Abrir arquivo
139 fp = fopen("dados.txt", "r");                   // Ponteiro que aponta para
        o arquivo
140
141 c = 0;
142 if(fp == NULL)
143 printf("Erro, nao foi possivel abrir o arquivo\n");
144 else{
145 while((contador = fgetc(fp)) != EOF)            //Equanto nao for Fim De Arquivo
        , contar a quantidade de linhas
146 if(contador == '\n')
147 c++;
148 }
149 fclose(fp);
```

```
150
151 return c;
152 }
153
154 void Armazenar_Amostras (float *Va, float *Vb, float *Vc){
155
156 int j;                                          // Variaveis auxiliares:
    contadores
157 char contador;                                  // Identificador de nova
    linha
158
159 /* Ler dados e armazena-los nos vetores de tensao*/
160
161 FILE *fp;
162 fp = fopen("dados.txt", "r");                   //Abrir arquivo novamente
163
164 j = 0;
165 if(fp == NULL)
166 printf("Erro, nao foi possivel abrir o arquivo\n");
167 else{
168
169 while((fscanf(fp, "%f  %f  %f", &Va[j], &Vb[j], &Vc[j]))!=EOF ){
170
171 contador = fgetc(fp);
172 if (contador == '\n')
173 j++;
174 }
175 }
176
177 fclose(fp);
178 return;
179 }
180
181 //      Inicializacao da matriz de Clarke direta
182
183 void init_TClarke(typeM3x3 *M)
184 {
185 M->a11 = 2.0/3.0;    // Nao esquecer dos pontos
186 M->a12 = -1.0/3.0;
187 M->a13 = -1.0/3.0;
188 M->a21 = 0;
189 M->a22 = sqrt(3.0)/3.0;
190 M->a23 = -sqrt(3.0)/3.0;
191 M->a31 = 1.0/3.0;
192 M->a32 = 1.0/3.0;
193 M->a33 = 1.0/3.0;
194
195 return;
196 }
197
198 //      Transformacao de Clarke
199 // [3 por 3] x [3 por 1]
200 // X = corrente ou tensao a ser transformada
201 // T = matriz de transformacao
202 // Y = corrente ou tensao transformada
203
204 void Transf(typeV3x1 *X, typeM3x3 *T, typeV3x1 *Y)
205 {
206 Y->A1 = T->a11*X->A1 + T->a12*X->A2 + T->a13*X->A3;
```

```
207 Y->A2 = T->a21*X->A1 + T->a22*X->A2 + T->a23*X->A3;
208 Y->A3 = T->a31*X->A1 + T->a32*X->A2 + T->a33*X->A3;
209 return;
210 }
211
212
213 // Funcao que implementa o gerador de sinais em quadratura
214 // Equacoes obtidas a partir da aplicacao da transf. de Tustin e transf. Z
215 // al, qal: Sinais em quadratura da componente alfa
216 // be, qbe: Sinais em quadratura da componente beta
217
218 typeV4x1 Quad_Signal_Generator(int k, float dt,
219 float *w, float *alpha, float *beta,
220 float *v_al, float *qv_al, float *v_be, float *qv_be)
221 {
222 float kqsg;
223 float C1, C2, C3, C4;
224 float al, qal, be, qbe;
225 kqsg = sqrt(2.0);
226
227 if(k < 2){
228
229 al = 0;
230 qal = 0;
231 be = 0;
232 qbe = 0;
233
234 }
235 else{
236
237 C1 = 2.0*w[k-1]*kqsg*dt;
238 C2 = pow(w[k-1], 2.0)*pow(dt, 2.0);
239 C3 = pow(w[k-1], 2.0)*pow(dt, 2.0)*kqsg;
240 C4 = pow(w[k-1], 2.0)*pow(dt, 2.0);
241
242 al = (C1*alpha[k] - C1*alpha[k-2] - (2.0*C2-8.0)*v_al[k-1] - (4.0-C1+C2)*v_al[k
    -2])/(4.0+C1+C2);
243 qal = (C3*alpha[k] + 2.0*C3*alpha[k-1] + C3*alpha[k-2] - ((2.0*C4)-8.0)*qv_al[k
    -1] - (4.0-C1+C4)*qv_al[k-2])/(4.0+C1+C4);
244
245 be = (C1*beta[k] - C1*beta[k-2] - ((2.0*C2)-8.0)*v_be[k-1] - (4.0-C1+C2)*v_be[k
    -2])/(4.0+C1+C2);
246 qbe = (C3*beta[k] + 2.0*C3*beta[k-1] + C3*beta[k-2] - ((2.0*C4)-8.0)*qv_be[k-1]
    - (4.0-C1+C4)*qv_be[k-2])/(4.0+C1+C4);
247
248 }
249
250 typeV4x1 Quad_signals;
251
252 Quad_signals.A1 = al;
253 Quad_signals.A2 = qal;
254 Quad_signals.A3 = be;
255 Quad_signals.A4 = qbe;
256
257 return Quad_signals;
258 }
259
260 //      Gerador de Sequencia Positiva
261 // X = Sinais em quadratura gerados a partir das componentes alfa e beta
```

```
262 // Y = Componentes alfa e beta de seq. positiva
263
264 void SeqPos_Generator(typeV4x1 *X, typeV2x1 *Y)
265 {
266 Y->A1 = (X->A1 - X->A4)/2;
267 Y->A2 = (X->A2 + X->A3)/2;
268 return;
269 }
270
271 //      Transformacao de Park direta
272 // X = corrente ou tensao (alpha beta) a ser trasf.
273 // th = angulo de sincronismo (radianos)
274 // Y = corrente ou tensao (d q) obtida
275 // Componente '0' foi descartada
276
277 void TPark(int k, typeV2x1 *X, float *th, typeV2x1 *Y)
278 {
279 if(k < 2){
280 Y->A1 = 0.0;
281 Y->A2 = 0.0;
282 }
283 else {
284 Y->A1 = cos(th[k-1])*X->A1 + sin(th[k-1])*X->A2;
285 Y->A2 = -sin(th[k-1])*X->A1 + cos(th[k-1])*X->A2;
286 }
287 return;
288 }
289
290 // Funcao qie implementa o controlador PI
291 // Equacao de Vlf obtida a partir da aplicacao da transf. de Tustin e transf. Z
292
293 void Controlador_PI(int k, float dt, float *q, float *Vlf)
294 {
295 //Constantes do controlador:
296
297 float Ksi, Wn;
298 float P, I;
299 float K1, K2;
300
301
302 if(k < 2){
303 Vlf[k] = 0.0;
304 }
305 else{
306 Ksi = 1/sqrt(2);                    // Fator de amortecimento
307 Wn = 25*M_PI;                       // Frequencia de corte
308
309 P = (2*Ksi*Wn)/220*sqrt(2);         // Termo proporcional
310 I = (Wn*Wn)/220*sqrt(2);            // Termo integral
311
312 K1 = 2*P+I*dt;
313 K2 = I*dt-2*P;
314
315 Vlf[k] = (2*Vlf[k-1] + K1*q[k] + K2*q[k-1])/2;
316 }
317
318 return;
319 }
320
```

```
321 // Gerador de Frequencia
322
323 void Freq_Generator(int k, float *Vlf, float *w){
324
325 if(k < 2){
326 w[k] = 0.0;
327 }
328 else{
329 w[k] = Vlf[k] + (2*(M_PI)*50);
330 }
331 return;
332 }
333
334 // Gerador de Fase
335
336 void Phase_Generator(int k, float dt, float *w, float *Theta){
337
338 if(k < 2){
339 Theta[k] = 0.0;
340 }
341 else{
342 Theta[k] = (2*Theta[k-1] + dt*w[k] + dt*w[k-1])/2;
343 if (Theta[k] > 2*M_PI){
344 Theta[k] = 0;
345 }
346 }
347 return;
348 }
```

APÊNDICE D

CÓDIGO DE IMPLEMENTAÇÃO NO *ARDUINO DUE*

Listing D.1: DSOGI-PLL implementado no *Arduino Due*

```
1  // Amostrar tensao constantemente e implementar o DSOGI-PLL
2
3  #include <stdio.h>
4  #include <math.h>
5  #include "transf.h"
6
7  #define dtm (1000000/2500)  // Intervalo de amostragem em us (Variavel global)
8  float Fs = 2500;
9  float dt = 1/Fs;            // Intervalo de amostragem (Variavel global)
10 unsigned long somador = 0L; // Variavel auxiliar para implementacao da
       amostragem
11
12 int input1 = A0;                          // Entrada Tensao A
13 int input2 = A1;                          // Entrada Tensao B
14 int input3 = A2;                          // Entrada Tensao C
15
16 int val1, val2, val3;
17 int led = 13;
18
19 void setup()
20 {
21 pinMode(input1,INPUT);
22 pinMode(input2,INPUT);
23 pinMode(input3,INPUT);
24
25 pinMode(led,OUTPUT);
26
27 // Configurar registradores diretamente, para leitura mais rapida dos canais de
       entrada:
28
29 pmc_enable_periph_clk(ID_ADC);
30 adc_init(ADC, SystemCoreClock, ADC_FREQ_MAX, ADC_STARTUP_FAST);
31 adc_disable_interrupt(ADC, 0xFFFFFFFF);
32 adc_set_resolution(ADC, ADC_12_BITS);
33 adc_configure_power_save(ADC, 0, 0);
34 adc_configure_timing(ADC, 0, ADC_SETTLING_TIME_3, 1);
35 adc_set_bias_current(ADC, 1);
36 adc_stop_sequencer(ADC);
37 adc_disable_tag(ADC);
38 adc_disable_ts(ADC);
39 adc_disable_channel_differential_input(ADC, ADC_CHANNEL_7);
40 adc_configure_trigger(ADC, ADC_TRIG_SW, 1);
41 adc_disable_all_channel(ADC);
42 adc_enable_channel(ADC, ADC_CHANNEL_7);
43 adc_enable_channel(ADC, ADC_CHANNEL_6);
44 adc_enable_channel(ADC, ADC_CHANNEL_5);
```

```
45 adc_start(ADC);
46 }
47
48 void loop()
49 {
50
51 // ************** DECLARACAO DE VARIAVEIS ********************
52
53 int i = 3;                                    // i, k: Variaveis necessarias p/
       armazenamento de...
54 int k = 0;                                    // ... 2 amostras passadas.
55
56 float  A = 0;                                 // Tensao de entrada
57 float  B = 0;                                 // Tensao de entrada
58 float  C = 0;                                 // Tensao de entrada
59
60 float alpha[i], beta[i], zero[i];             // Tensao alfa-beta-zero
61 float wold[i];                                // Frequencia calculada
62 float v_al[i], qv_al[i], v_be[i], qv_be[i];   // Sinais em quadratura das
       componentes alfa e beta
63 float d[i], q[i];                            // Componentes d e q
64 float theta[i];                              // Fase calculada
65 float Vlf[i];                                // Sinal na saida do controlador
       PI
66
67 float alp, bep;                              // Alfa e beta de seq. positiva
68
69 float n1, n2, ntheta;                        // Variaveis a serem jogadas na
       saida ADC
70
71 typeM3x3 TC;                                 // Tranformacao direta de Clarke
72 typeV3x1 Vabc;                               // V abc
73 typeV3x1 VCl;                                // V alpha, beta, 0
74 typeV4x1 Quad_Signals;                       // Sinais em quadratura das
       componentes alfa e beta
75 typeV2x1 SeqPos;                            // V alpha, beta de sequencia
       positiva
76 typeV2x1 Vdq;                               // V dq
77
78 init_TClarke(&TC);
79
80 analogWriteResolution(12);                   // Capacidade maxima da saida DAC
81 // Entrada sera convertida p/ valores entre 0 e 4095
82 while(1)                                     // Repetir processo
       indefinidamente
83 {
84 if (micros() - somador > dtm)               // Respeitar intervalo de amostragem
85 {
86 somador += dtm;
87
88 PIO_Set(PIOB,PIO_PB27B_TIOB0);
89
90 while ((adc_get_status(ADC) & ADC_ISR_DRDY) != ADC_ISR_DRDY)
91 {}; //Wait for end of conversion
92
93 val1 = adc_get_channel_value(ADC, ADC_CHANNEL_7); // Ler ADC, canal 7 (A0)
94 val2 = adc_get_channel_value(ADC, ADC_CHANNEL_6); // Ler ADC, canal 6 (A1)
95 val3 = adc_get_channel_value(ADC, ADC_CHANNEL_5); // Ler ADC, canal 5 (A2)
96
```

```
97  A = map(val1, 0, 4095, -127*sqrt(2), 127*sqrt(2));
98  B = map(val2, 0, 4095, -127*sqrt(2), 127*sqrt(2));
99  C = map(val3, 0, 4095, -127*sqrt(2), 127*sqrt(2));
100
101 Vabc.A1 = A;
102 Vabc.A2 = B;
103 Vabc.A3 = C;
104
105 Transf(&Vabc, &TC, &VCl);
106
107 alpha[k] = VCl.A1;
108 beta[k] = VCl.A2;
109 zero[k] = VCl.A3;
110
111 Quad_Signals = Quad_Signal_Generator(k, dt, wold, alpha, beta, v_al, qv_al, v_be
        , qv_be);
112
113 v_al[k] = Quad_Signals.A1;
114 qv_al[k] = Quad_Signals.A2;
115 v_be[k] = Quad_Signals.A3;
116 qv_be[k] = Quad_Signals.A4;
117
118 SeqPos_Generator(&Quad_Signals, &SeqPos);
119
120 alp = SeqPos.A1;
121 bep = SeqPos.A2;
122
123 TPark(k, &SeqPos, theta, &Vdq);
124
125 d[k] = Vdq.A1;
126 q[k] = Vdq.A2;
127
128 Controlador_PI(k, q, Vlf);
129
130 Freq_Generator(k, Vlf, wold);
131
132 Phase_Generator(k, wold, theta);
133
134 n1 = map(alp, -130*sqrt(2), 130*sqrt(2), 0, 4095);
135 n2 = map(bep, -130*sqrt(2), 130*sqrt(2), 0, 4095);
136 ntheta = theta[k]*(4095/(2*M_PI));
137
138 analogWrite(DAC0, val1);
139 analogWrite(DAC1, val2);
140
141 k = k + 1;
142
143 if (k > 2){
144
145 alpha[0] = alpha[1]; alpha[1] = alpha[2];
146 beta[0] = beta[1]; beta[1] = beta[2];
147 zero[0] = zero[1]; zero[1] = zero[2];
148
149 wold[0] = wold[1]; wold[1] = wold[2];
150
151 v_al[0] = v_al[1]; v_al[1] = v_al[2];
152 qv_al[0] = qv_al[1]; qv_al[1] = qv_al[2];
153 v_be[0] = v_be[1]; v_be[1] = v_be[2];
154 qv_be[0] = qv_be[1]; qv_be[1] = qv_be[2];
```

```
155
156 d[0] = d[1]; d[1] = d[2];
157 q[0] = q[1]; q[1] = q[2];
158
159 theta[0] = theta[1]; theta[1] = theta[2];
160 Vlf[0] = Vlf[1]; Vlf[1] = Vlf[2];
161
162 k = 2;
163
164 }
165
166 PIO_Clear(PIOB,PIO_PB27B_TIOB0);
167 }
168 }
169 }
170
171 void init_TClarke(typeM3x3 *M)
172 {
173 M->a11 = 2.0/3.0;    // Nao esquecer dos pontos
174 M->a12 = -1.0/3.0;
175 M->a13 = -1.0/3.0;
176 M->a21 = 0;
177 M->a22 = sqrt(3.0)/3.0;
178 M->a23 = -sqrt(3.0)/3.0;
179 M->a31 = 1.0/3.0;
180 M->a32 = 1.0/3.0;
181 M->a33 = 1.0/3.0;
182
183 return;
184 }
185
186 void Transf(typeV3x1 *X, typeM3x3 *T, typeV3x1 *Y)
187 {
188 Y->A1 = T->a11*X->A1 + T->a12*X->A2 + T->a13*X->A3;
189 Y->A2 = T->a21*X->A1 + T->a22*X->A2 + T->a23*X->A3;
190 Y->A3 = T->a31*X->A1 + T->a32*X->A2 + T->a33*X->A3;
191 return;
192 }
193
194 typeV4x1 Quad_Signal_Generator(int k, float dt,
195 float *w, float *alpha, float *beta,
196 float *v_al, float *qv_al, float *v_be, float *qv_be)
197 {
198 float kqsg;
199 float C1, C2, C3, C4;
200 float al, qal, be, qbe;
201 kqsg = sqrt(2.0);
202
203 if(k < 2){
204
205 al = 0;
206 qal = 0;
207 be = 0;
208 qbe = 0;
209
210 }
211 else{
212
213 C1 = 2.0*w[k-1]*kqsg*dt;
```

```
214 C2 = pow(w[k-1], 2.0)*pow(dt, 2.0);
215 C3 = pow(w[k-1], 2.0)*pow(dt, 2.0)*kqsg;
216 C4 = pow(w[k-1], 2.0)*pow(dt, 2.0);
217
218 al = (C1*alpha[k] - C1*alpha[k-2] - (2.0*C2-8.0)*v_al[k-1] - (4.0-C1+C2)*v_al[k
    -2])/(4.0+C1+C2);
219 qal = (C3*alpha[k] + 2.0*C3*alpha[k-1] + C3*alpha[k-2] - ((2.0*C4)-8.0)*qv_al[k
    -1] - (4.0-C1+C4)*qv_al[k-2])/(4.0+C1+C4);
220
221 be = (C1*beta[k] - C1*beta[k-2] - (2.0*C2-8.0)*v_be[k-1] - (4.0-C1+C2)*v_be[k
    -2])/(4.0+C1+C2);
222 qbe = (C3*beta[k] + 2.0*C3*beta[k-1] + C3*beta[k-2] - ((2.0*C4)-8.0)*qv_be[k-1]
    - (4.0-C1+C4)*qv_be[k-2])/(4.0+C1+C4);
223
224 }
225
226 typeV4x1 Quad_signals;
227
228 Quad_signals.A1 = al;
229 Quad_signals.A2 = qal;
230 Quad_signals.A3 = be;
231 Quad_signals.A4 = qbe;
232
233 return Quad_signals;
234 }
235
236 void SeqPos_Generator(typeV4x1 *X, typeV2x1 *Y)
237 {
238 Y->A1 = (X->A1 - X->A4)/2;
239 Y->A2 = (X->A2 + X->A3)/2;
240 return;
241 }
242
243 void TPark(int k, typeV2x1 *X, float *th, typeV2x1 *Y)
244 {
245 if(k < 2){
246 Y->A1 = 0.0;
247 Y->A2 = 0.0;
248 }
249 else {
250 Y->A1 = cos(th[k-1])*X->A1 + sin(th[k-1])*X->A2;
251 Y->A2 = -sin(th[k-1])*X->A1 + cos(th[k-1])*X->A2;
252 }
253 return;
254 }
255
256 void Controlador_PI(int k, float *q, float *Vlf)
257 {
258 //Constantes do controlador:
259
260 float Ksi, Wn;
261 float P, I;
262 float K1, K2;
263
264 if(k < 2){
265 Vlf[k] = 0.0;
266 }
267 else{
268 Ksi = sqrt(2);                              // Damping ratio
```

```
269 Wn = 25*M_PI;                                   // Cut-off frequency
270
271 P = (2*Ksi*Wn)/127*sqrt(2);                     // Termo proporcional
272 I = (Wn*Wn)/127*sqrt(2);                        // Termo integral
273
274 K1 = (2*P)+(I*dt);
275 K2 = (I*dt)-(2*P);
276
277 Vlf[k] = (2*Vlf[k-1] + K1*q[k] + K2*q[k-1])/2;
278 }
279
280 return;
281 }
282
283 void Freq_Generator(int k, float *Vlf, float *w){
284
285 if(k < 2){
286 w[k] = 0.0;
287 }
288 else{
289 w[k] = Vlf[k] + (2*(M_PI)*60);
290 }
291 return;
292 }
293
294 void Phase_Generator(int k, float *w, float *Theta){
295
296 if(k < 2){
297 Theta[k] = 0.0;
298 }
299 else{
300 Theta[k] = (2*Theta[k-1] + dt*w[k] + dt*w[k-1])/2;
301 if (Theta[k] > 2*M_PI){
302 Theta[k] = 0;
303 }
304 }
305 return;
306 }
```