UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
ESCOLA DE ENGENHARIA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA

MATHEUS LADVIG BUDELON OLIVEIRA

PyBlock - interface para execução de códigos Python no *software* PSIM

Porto Alegre

2020

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
ESCOLA DE ENGENHARIA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA

MATHEUS LADVIG BUDELON OLIVEIRA

PyBlock - interface para execução de códigos Python no *software* PSIM

Projeto de Diplomação apresentado ao Departamento de Engenharia Elétrica da Escola de Engenharia da Universidade Federal do Rio Grande do Sul, como requisito parcial para Graduação em Engenharia Elétrica

Orientador: Prof. Dr. Jeferson Vieira Flores

Porto Alegre
2020

MATHEUS LADVIG BUDELON OLIVEIRA

PyBlock - interface para execução de códigos Python no *software* PSIM

Projeto de Diplomação apresentado ao Departamento de Engenharia Elétrica da Escola de Engenharia da Universidade Federal do Rio Grande do Sul, como requisito parcial para Graduação em Engenharia Elétrica

---

Prof. Dr. Jeferson Vieira Flores
Orientador - UFRGS

Aprovado em 30 de Novembro de 2020.

BANCA EXAMINADORA

Prof. Dr. Aurélio Tergolina Salton
UFRGS

Prof. Dr. Ivan Müller
UFRGS

Prof. Dr. Jeferson Vieira Flores
UFRGS

Dedico este trabalho à minha família.

AGRADECIMENTOS

Gostaria de agradecer primeiramente à minha família que desde sempre me apoiou em minhas escolhas, esteve presente em todos os momentos decisivos de minha vida e nunca mediu esforços para me fornecer a melhor educação possível. Também gostaria de agradecer à minha namorada, que me acompanhou durante este curso, foi uma companheira compreensível e me incentivou a sempre dar o melhor de mim. Também gostaria de agradecer ao professor Jeferson, que compartilhou todo seu conhecimento e fez este trabalho muito melhor. Por fim, gostaria de agradecer ao pagador de impostos que financiou este trabalho e firmar um compromisso com a sociedade de sempre dar o melhor de mim.

"Através dos séculos existiram homens que deram os primeiros passos, por novas estradas, armados com nada além de sua própria visão.".

Ayn Rand

RESUMO

O presente trabalho tem como objetivo o desenvolvimento de um novo bloco de simulação denominado PyBlock junto ao software de simulação PSIM para a execução de códigos descritos em linguagem Python. Além disso, é apresentado o estudo de aplicações desta ferramenta no controle de conversores CC-CA. O novo bloco de simulação apresentado neste trabalho pode ser acoplado ao PSIM apenas adicionando-se uma pasta auxiliar na raiz do projeto e checando se o interpretador de linguagem Python já está instalado na máquina. A partir do desenvolvimento do PyBlock, espera-se difundir o uso de algoritmos de controle mais complexos em aplicações da área de Eletrônica de Potência. Visando testar este novo software, dois estudos de caso foram propostos: i) controle de um inversor trifásico de dois níveis a partir de um controlador PI; ii) controle de um inversor trifásico multinível com neutro grampeado a partir de técnicas de controle chaveado com restrições na forma de desigualdades lineares matriciais(LMIs). Em ambos os casos, o desempenho do sistema utilizando o PyBlock foi comparado à implementação do mesmo controlador no bloco DLL nativo do PSIM, que permite a execução de códigos em linguagem C. Os resultados de simulação mostraram que o PyBlock possibilita uma implementação mais rápida que o atual bloco nativo do PSIM e também um ganho de performance se comparado à implementação com os blocos nativos do Matlab.

Palavras-chave: LMI; Funções de Lyapunov; Conversor multinível; Controlador PID; Sistemas chaveados; *Software* PSIM; *Dynamic Link Library*;

ABSTRACT

This work aims to develop a new simulation block denominated PyBlock, which can be linked with the PSIM simulation software to allow the execution of codes written in Python programming language. Besides that, it is presented the application study in DC-AC inverters. The new simulation block presented in this work can be linked with PSIM simply pasting the block folder inside the root project folder and installing Python in the same machine if it is not yet installed. From the PyBlock development, it is expected to spread the use of more complex control algorithms in the Power Eletronics field. Aiming to test the new software, two case studies were proposed: i) control of three phase inverter with two levels from a PI controller; ii) control of a three phase neutral point clampled inverter using switched control techniques with restrictions in the form of linear matrix inequalities(LMIs). In both cases, the system performance using the PyBlock were compared with the implementation of the same controller in the DLL native block from PSIM, which allows C programming language. The simulation results revealed that the PyBlock allows a faster implementation than DLL Block and with a simulation time performance better than Matlab software.

Keywords:LMI; Lyapunov functions; Multilevel inverter; PID controller; Switched systems; *Software* PSIM; *Dynamic Link Library*;

LISTA DE FIGURAS

Figura 1 – *DLLBlock*. 18
Figura 2 – Arquivo de definição de módulo. 18
Figura 3 – Circuito com uso de *DLLBlock*. 19
Figura 4 – Código da média do sinal na função exportada pela *DLL*. 19
Figura 5 – Resultado da simulação do circuito da Figura 3. 20
Figura 6 – Esquema do interpretador de linguagem Python. 21
Figura 7 – *PyBlock*. 23
Figura 8 – Algoritmo da função em linguagem C em blocos. 30
Figura 9 – Detalhes do bloco *Executar função*. 30
Figura 10 – Esquema de teste do PSIM com DLL. 32
Figura 11 – Teste de integração da DLL com Python. 32
Figura 12 – *PyBlock* no PSIM. 36
Figura 13 – Arquivos de compilação. 36
Figura 14 – Configurações do bloco no PSIM. 37
Figura 15 – Exemplo de uso do *PyBlock*. 37
Figura 16 – Arquivos de simulação. 38
Figura 17 – Resultado da simulação do exemplo. 38
Figura 18 – Inversor de trifásico de dois níveis. 39
Figura 19 – Circuito do inversor de 2 níveis. 40
Figura 20 – Lugar das raízes de C(S)G(S). 42
Figura 21 – Resposta ao degrau. 43
Figura 22 – Descrição do inversor no PSIM. 45
Figura 23 – Conversão de tipos e definição da referência. 46
Figura 24 – Saída do PI e definição do chaveamento. 46
Figura 25 – Corrente id e a referência. 47
Figura 26 – Corrente iq e a referência. 47
Figura 27 – Bloco PI nativo com circuito de controle. 48
Figura 28 – Resultado de I_d obtida com o bloco nativo PI. 48
Figura 29 – Resultado de I_q obtida com o bloco nativo PI. 48
Figura 30 – Diferença na tensão V_d. 49
Figura 31 – Diferença na tensão V_q. 49
Figura 32 – Família dos inversores multiníveis. 50
Figura 33 – Conversor de 3 níveis. 51
Figura 34 – Circuito do inversor multinível. 52
Figura 35 – Estados do inversor multinível. 53
Figura 36 – Diagrama de estados do conversor multinível. 53
Figura 37 – Simulação do controlador através de LMI's com Matlab. 63
Figura 38 – Conversor multinível. 64

Figura 39 – Bloco NPC. 64
Figura 40 – Id do conversor multinível e a referência. 65
Figura 41 – Iq do conversor multinível e a referência. 65
Figura 42 – Estados chaveados durante a simulação. 66
Figura 43 – Id do conversor multinível e a referência com código em C. 66
Figura 44 – Iq do conversor multinível e a referência com código em C. 67
Figura 45 – Estados chaveados durante a simulação com código em C. 67
Figura 46 – Resultado de integração numérica com matrizes solucionadas no *solver* de Python. 69
Figura 47 – Exemplo de *logs* amostrados em tempo real e disponibilizados ao usuário. 74

LISTA DE TABELAS

Tabela 1 – Estados do inversor de 2 níveis 39
Tabela 2 – Valores das constantes do conversor. 42
Tabela 3 – Comparação do desempenho entre Python e Matlab 63

LISTA DE ABREVIATURAS

API	*Application Programming Interface*
IHD	*Individual Harmonic Distortions*
LGR	Lugar Geométrico das Raízes
LMI	*Linear Matrix Inequalities*
LPF	*Low-Pass Filter*
PID	Proporcional Integral Derivativo
PWM	*Pulse Width Modulation*
RMS	*Root Mean Square*
THD	*Total Harmonic Distortions*

SUMÁRIO

1 INTRODUÇÃO 14

2 CONSTRUÇÃO DO PYBLOCK 17
2.1 CONCEITOS DE BASE 17
2.1.1 DLL E DLLBlock 17
2.1.2 INTERPRETADOR PYTHON 19
2.2 CONSTRUÇÃO DO PYBLOCK 21
2.2.1 VISÃO GERAL DO PyBlock 21
2.2.2 PYTHON EMBARCADO 24
2.2.3 CONVERSÂO DE TIPOS 26
2.2.4 GESTÃO DA MEMÓRIA E DETECÇÃO DE ERROS 28
2.2.5 CONSTRUÇÃO DOS BLOCOS DE CÓDIGO E COMUNICAÇÃO . . 29
2.3 TESTES 29
2.3.1 TESTES UNITÁRIOS 30
2.3.2 TESTES DE INTEGRAÇÃO 31
2.3.2.1 PSIM E DLL 31
2.3.2.2 DLL E PYTHON 31
2.3.2.3 TESTE INTEGRADO DE TODOS BLOCOS 34
2.3.2.4 ARMAZENAMENTO DE INFORMAÇÃO 35
2.4 INSTALAÇÃO E EXEMPLO 36
2.5 CONCLUSÃO DO CAPÍTULO 38

3 INVERSOR PWM TRIFÁSICO 39
3.1 TOPOLOGIA DO INVERSOR TRIFÁSICO DE DOIS NÍVEIS 39
3.2 PROJETO DO CONTROLADOR PI E IMPLEMENTAÇÃO PYTHON 40
3.2.1 FORMULAÇÃO GERAL 40
3.2.2 PROJETO DO CONTROLADOR 41
3.3 RESULTADOS DE SIMULAÇÃO 45
3.3.1 PYTHON COM PSIM 45
3.4 CONCLUSÃO DO CAPÍTULO 49

4 INVERSOR MULTINÍVEL 50
4.1 TOPOLOGIA DO INVERSOR 50
4.2 PROJETO DO CONTROLADOR 51
4.2.1 FORMULAÇÃO GERAL 52
4.2.2 IMPLEMENTAÇÃO DO CONTROLADOR 57
4.3 RESULTADOS DE SIMULAÇÃO 61
4.3.1 IMPLEMENTAÇÃO DO CONTROLADOR EM MATLAB 61

4.3.2 IMPLEMENTAÇÃO VIA *PyBlock* 64
4.4 CONCLUSÃO DO CAPÍTULO 69

5 CONCLUSÃO 70

REFERÊNCIAS BIBLIOGRÁFICAS 72

1 INTRODUÇÃO

O software PSIM é um dos softwares mais utilizados no projeto e simulação de circuitos de eletrônica de potência, pois permite a simulação de características intrínsecas dos componentes fornecendo um resultado de simulação com precisão e acurácia. Este software possui muitos blocos de simulação já previamente projetados e disponibilizados ao usuário, além de contar com uma comunidade de usuários em escala mundial que utilizam o software e compartilham conteúdo didático.

O PSIM também permite implementar algoritmos de controle previamente projetados. Estes algoritmos de controle podem ser implementados através de blocos já disponibilizados pelo software como o bloco PID, PI, filtros e etc. Outra maneira possível é através de algoritmos escritos em linguagens de programação. O software, até a atual versão, doze, disponibiliza o *C-Block* e o *DLL-Block* para a execução de algoritmos descritos em linguagem C. O bloco *DLL* permite que o usuário construa uma biblioteca de vínculo dinâmico e disponibilize ao bloco, enquanto o *C-Block* permite que o projetista descreva o algoritmo de controle diretamente na interface do PSIM. De maneira geral, ambos os blocos têm o mesmo objetivo de executar algoritmos em linguagem C.

Muitos dos projetos atualmente em fase de prototipação e simulação são primeiramente desenvolvidos através de linguagens de programação de mais alto nível e isto vem sendo uma tendência devido à diversos fatores, principalmente, o tempo de engenharia investido na validação de conceitos. A utilização de linguagem de alto nível permite, em regra, algumas vantagens como em tempo de prototipação, modularização, uso de bibliotecas de terceiros e facilidade de compreensão do código. Este fenômeno pode ser exemplificado pela grande aceitação do *software* Matlab dentro de empresas e da Academia. O mesmo é amplamente utilizado em simulações e prototipação devido à sua facilidade de compreender a linguagem, nenhuma gestão de memória a ser feita pelo programador e também uma grande quantidade de funções já implementadas.

Em (ROBINSON, 2017), resultados de pesquisa do crescimento de Python foram apresentados, identificando o crescimento exponencial dessa linguagem de programação nos últimos anos no ambiente de software industrial e acadêmico. Também em (GOLDEMBERG *et al.*, 2009) foi apresentado uma plataforma totalmente desenvolvida em Python para fins de simulação de circuitos de eletrônica de potência, no entanto esta plataforma não permite implementar métodos de controle utilizando uma linguagem de programação e também não possui a mesma aceitação na comunidade de eletrônica de potência que o PSIM. Logo, ambos trabalhos foram motivação para a proposição de uma interface capaz de unir um software amplamente aceito pela comunidade acadêmica como o PSIM e uma linguagem de programação crescente, aceita pela comunidade científica e de fácil prototipação como Python.

Assim, o objetivo principal deste trabalho é desenvolver um novo bloco no PSIM que permita a execução de códigos descritos na linguagem Python visando a geração de sinais de controle para aplicações de eletrônica de potência. Este novo bloco, denominado de PyBlock, até o limite do nosso conhecimento é inédito dentro das funcionalidades do PSIM. Esta nova ferramenta de simulação a ser acoplada ao PSIM possibilita que novos algoritmos de controle sejam implementados utilizando o *software*, pois a resolução de problemas de otimização ou qualquer outro problema matemático complexo computacionalmente pode ser resolvido através da ajuda de bibliotecas especializadas de Python. Embora a linguagem C, utilizada no PSIM até o momento, também permita construir algoritmos de resolução de problemas complexos, o baixo nível da linguagem C impõe desafios de implementação muito maiores que no caso de Python.

Com o intuito de testar e validar este novo bloco, foram projetados e implementados métodos de controle aplicados à conversores trifásicos CC-CA. Mais especificamente, foram projetados dois métodos de controle distintos aplicados a conversores também distintos. O controle através da aplicação de técnicas de sistemas chaveados foi projetado para um conversor multinível trifásico com neutro grampeado, muito utilizado em sistemas de alta potência e sistemas fotovoltaicos, e também projetou-se um controlador PI para um conversor trifásico não multinível com 3 chaves. Os algoritmos de controle foram implementados em linguagem Python e os circuitos dos conversores descritos no software PSIM.

O controle em malha fechada é fundamental em várias aplicações de eletrônica de potência visando garantir a regulação de saída e robustez de projeto. Entre essas aplicações se destacam os conversores CC-CA, principalmente em aplicações associadas a energia renováveis.

As técnicas de controle mais comuns aplicadas a conversores multiníveis foram apresentadas em (RODRIGUEZ; LAI; PENG, 2002) bem como suas vantagens e desvantagens, além disso são apresentadas topologias de circuito e os métodos de acionamento com destaque para os métodos *Pulse width modulation*(PWM) e *Space vector modulation*(SVM). Em (SILVA *et al.*, 2016), uma configuração de conversor multinível com neutro grampeado foi modelado e apresentado com as suas respectivas equações de chaveamento para aplicações de energia solar, para este conversor foi projetado um controlador MPC(*model predictive control*). Em (SILVA; RODRIGUES; COSTA, 2000) o controle deste conversor é feito usando-se de um controlador por modos deslizantes(*sliding model control*).

Os conversores de potência são controlados via o chaveamento de semicondutores, sendo então candidatos para a aplicação de técnicas de controle de sistemas chaveados. Em (DEAECTO *et al.*, 2010), um controlador projetado com LMI's foi abordado com o objetivo principal de garantir estabilidade global. Nessa referência, os autores utilizaram funções de *Lyapunov* para a garantia de estabilidade e o projeto da função de chaveamento

dos conversores. Este controlador foi testado em algumas topologias de conversores CC-CC como buck, boost e buck-boost.

Em (DEZUO; LUNARDI; TROFINO, 2017), utilizaram-se técnicas de controle chaveado para controlar um sistema fotovoltaico levando-se em conta os parâmetros não lineares do sistema, como a iluminação do painel solar e distribuição irregular de temperatura, com o objetivo de rastrear o ponto de máxima potência. Uma das vantagens da utilização da técnica de controle chaveado apresentada nessa referência é a garantia de estabilidade global assintótica do sistema em malha fechada, tratamento formal de não linearidades e a inclusão de objetivos de projeto como robustez a variações paramétricas.

Em ambos os casos, (DEAECTO *et al.*, 2010), (DEZUO; LUNARDI; TROFINO, 2017) deve-se resolver a cada amostra um problema de otimização com restrições na forma de desigualdades lineares matriciais(*LMI's*). Estes problemas de otimização podem ser resolvidos através de bibliotecas em Python e Matlab, que fazem isso de forma sistemática e eficiente. Também em ambos artigos, o período de chaveamento deve ser pequeno(na ordem de microssegundos) para que o sistema em malha fechada convirja para o ponto de equilíbrio desejado.

2 CONSTRUÇÃO DO PYBLOCK

Neste capítulo tem-se o objetivo de detalhar os aspectos da construção do *PyBlock*. Logo, estudou-se aspectos teóricos e de construção prática, alguns desafios de implementação e por fim apresentou-se os testes realizados para validar o bom funcionamento deste bloco.

2.1 CONCEITOS DE BASE

2.1.1 DLL E DLLBlock

Como mencionado anteriormente, o *software* PSIM possibilita a execução de códigos em linguagem C, os quais devem ser disponibilizados como uma função dentro de um módulo DLL no caso do bloco *DLLBlock* ou diretamente no PSIM através do bloco *CBlock*.

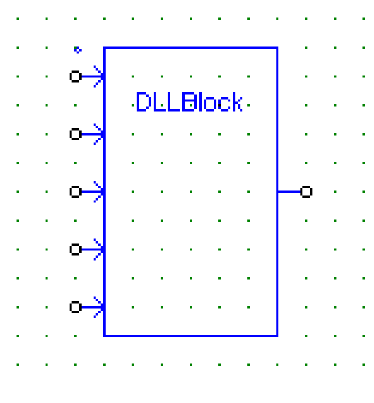
Através da *DLL* disponibilizada ao PSIM é permitido importar constantes ou variáveis de um código em C após os cálculos realizados. O objetivo de usar uma *DLL* é modularizar uma aplicação, ou seja, possibilita mais facilmente atualizar partes desta aplicação de maneira independente. A biblioteca de vínculo dinâmico é utilizada pois seu código é vinculado ao código executável apenas no momento de execução do arquivo, ou seja, são criados conectores no momento da construção da aplicação e antes da execução estes conectores são ativados. Este processo de execução de uma aplicação através de múltiplos arquivos é definido como ligação dinâmica(CHIOZO, 2007). Deve-se destacar que o tempo de execução de uma aplicação ligada dinamicamente é maior do que a de um programa ligado estaticamente.

O PSIM, a cada passo de simulação, faz a chamada da *DLL* pelo endereço fornecido pelo projetista através do *DLLBlock*. Primeiramente, a *DLL* é acessada e a função escrita na *DLL* é carregada, executada e após os cálculos retorna os valores numéricos ao PSIM, que então disponibiliza ao circuito pelo sinal de saída do *DLLBlock*.

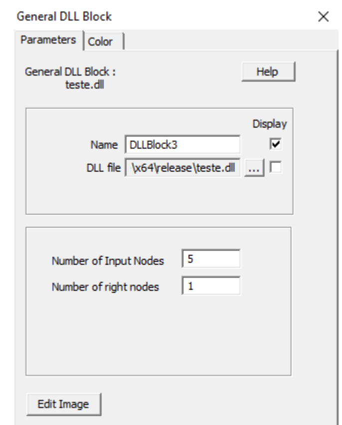
A seguir é apresentada a simulação de um circuito no PSIM usando-se do *DLLBlock*. O intuito é de se demonstrar a utilidade deste bloco para simulações onde parte dos sinais de interesse são gerados a partir de funções descritas em linguagem C.

A Figura 1a mostra o *DLLBlock*, neste caso com 5 entradas e uma saída, isto quer dizer que 5 sinais devem ser fornecidos como argumento para a função, a qual retornará para o circuito apenas um sinal de saída. Estas configurações são variáveis e estão disponíveis nos parâmetros do bloco como mostrado na Figura 1b.

Figura 1 – *DLLBlock*.



(a) Imagem do *DLLBlock*.



(b) Configurações do *DLLBlock*.

No mesmo bloco de configurações, o PSIM demanda o endereço do arquivo *.dll* compilado, ou seja, o caminho para a biblioteca de vinculo dinâmico com a função a ser executada. Alguns pontos devem ser levados em consideração na construção desta biblioteca. Primeiramente, o PSIM só aceita uma determinada configuração da função a ser exportada (POWERSYS, 2004), esta função deve ser escrita no formato *__declspec(dllexport) void simuser (t, delt, in, out)*, caso contrário o PSIM não aceita o bloco como válido e não executa a simulação. Outro ponto a ser observado e que pode resultar no não funcionamento do código é que o PSIM necessita obrigatoriamente de um arquivo de definição do módulo *.def* para fornecer ao vinculador informações sobre o programa a ser vinculado. Na Figura 2 é mostrado o código do arquivo *.def* escrito para uma função da *DLL* a ser compilada.

Figura 2 – Arquivo de definição de módulo.

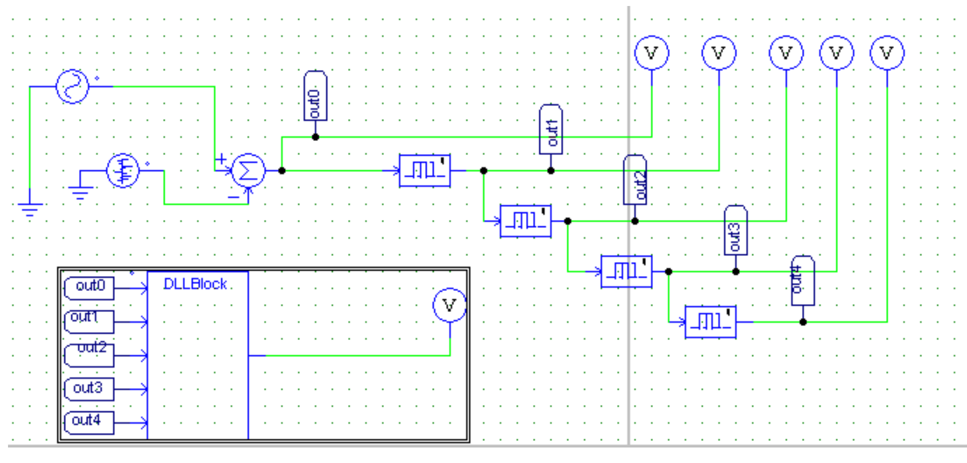
```
1  ; dllmain.def : Declares the module parameters for the DLL.
2
3  LIBRARY      "dllmain"
4  DESCRIPTION  'Test DLL Block For PSIM'
5
6  EXPORTS
7      simuser
```

Fonte: Autor

Para fins de ilustração, foi criada a simulação apresentada na Figura 3, onde um sinal de uma fonte de tensão sinusoidal de amplitude 100 Volts e período de 100 Hertz é somado com uma variável aleatória distribuída uniformemente no intervalo $[-10, 10]$ Volts. O sinal resultante, senoide somado ao ruído, é então filtrado pelo bloco DLL baseado na média das últimas 5 amostras, conforme apresentado na Figura 4.

Este exemplo ajuda a compreender alguns detalhes de implementação como: todas as variáveis devem ser do tipo *double*, a variável **in* é passada por referência sempre, a

Figura 3 – Circuito com uso de *DLLBlock*.



Fonte: Autor

Figura 4 – Código da média do sinal na função exportada pela *DLL*.

```
#include <math.h>

__declspec(dllexport) void simuser(double t, double delt, double* in, double* out)
{

    out[0] = (in[0] + in[1] + in[2] + in[3] + in[4]) / 5;


}
```

Fonte: Autor

DLL não tem permissão de escrita (POWERSYS, 2004) e também out[0] recebe o valor calculado que é disponibilizado ao PSIM.

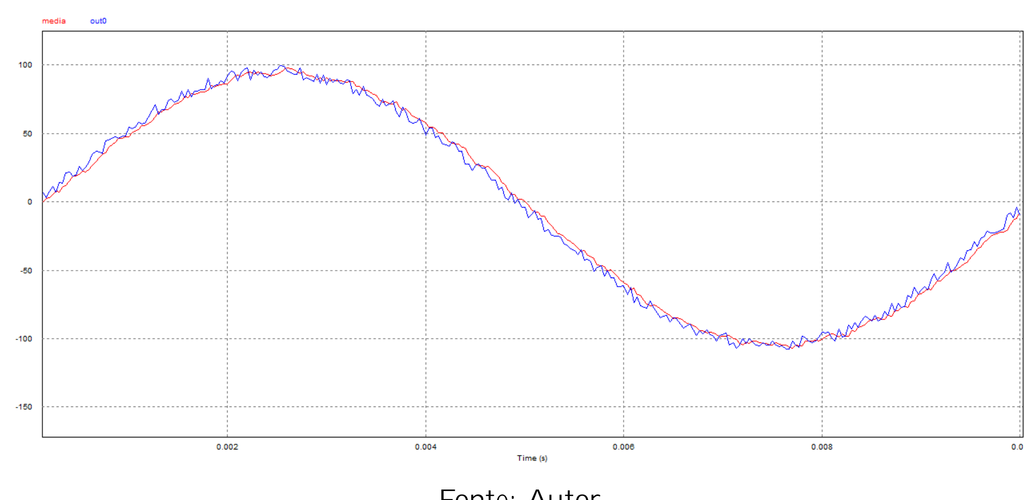
Simulando-se este circuito, obtém-se o resultado da Figura 5. Pode-se ver que o sinal de saída representado pela curva em vermelho de fato representa o sinal de entrada filtrado, com menor variabilidade na saída e um pequeno deslocamento de fase. Apesar de simples, este exemplo permite compreender detalhes importantes da utilização do bloco que serão úteis na construção do *PyBlock*, como será tratado posteriormente neste trabalho.

Neste trabalho, uma *DLL* foi programada com o objetivo de se utilizar o *DLLBlock* já existente no PSIM, para aplicações em código C, como bloco intermediador entre os sinais de entrada e saída do PSIM.

2.1.2 INTERPRETADOR PYTHON

Embarcar o compilador de linguagem Python em uma biblioteca dinâmica em C possibilita escrever códigos em Python e executá-los através de uma aplicação não compatível com a linguagem Python, através da biblioteca de vínculo dinâmico. Esta é exatamente uma necessidade de projeto deste trabalho, pois tem-se o objetivo de executar

Figura 5 – Resultado da simulação do circuito da Figura 3.



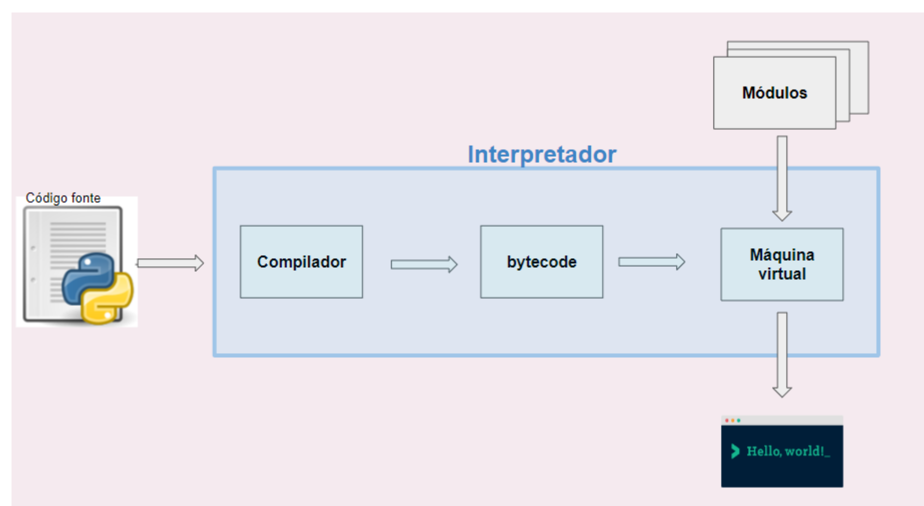
Fonte: Autor

um arquivo no programa PSIM e comunicá-lo com um código escrito em linguagem Python, embora o PSIM não possibilite esta comunicação como uma opção ao projetista.

O Python é uma linguagem compilada em alto nível, portanto os comandos de um programa escrito em linguagem Python não são diretamente transcritos para comandos em linguagem de máquina. Logo, após a compilação do código ainda existem mais passos de transcrição para que o programa de fato esteja escrito em linguagem de máquina. O código escrito em Python necessita de um interpretador, o qual é responsável, entre outras atribuições, por traduzir o código Python em *bytecode*, ou seja, compilar o código *.py*. A facilidade de se utilizar a linguagem Python do ponto de vista do usuário é devido a existência do interpretador, pois muitas tarefas complexas como gerir memória, alocar ponteiros e declarar tipo de variável são realizadas por ele. A compilação ocorre em mais alto nível, o que torna o código visível ao programador independente da plataforma operacional ou do processador, ademais este passo de compilação para *bytecode* permite ter uma linguagem em mais baixo nível a ser executada pela máquina virtual de Python. Pode-se utilizar este código em *bytecode* inclusive para um ganho de performance no tempo de execução, já que muitas aplicações já utilizam diretamente o arquivo em *bytecode* compilado como código fonte na execução do programa.

O processo de compilação pode ser dividido nas seguintes etapas: primeiramente, o código em Python é traduzido para uma árvore de comandos que é construída baseada em regras gramaticais próprias da linguagem. O segundo passo é transformar a árvore de comandos em uma *AST(Abstract Syntax Tree)*, a qual é fornecida ao código, chamado *Control Flow Graph*, para mapear o fluxo do programa e organizá-lo em blocos de *bytecodes*. Após o terceiro passo tem-se um grafo, onde cada nó representa um bloco de *bytecode*. Logo, o quarto e último passo é usar um algoritmo de *Busca em profundidade* (THULASIRAMAN; SWAMY, 2011) para construir a sequência de execução dos pontos, onde cada ponto possui um *bytecode*.

Figura 6 – Esquema do interpretador de linguagem Python.



Fonte: Autor

A etapa final do processo se dá na máquina virtual de Python, que recebe o código em linguagem *bytecode* e cria pilhas de dados e comandos. Através das pilhas se executa a sequência de operações em linguagem C. Ainda existem outras implementações do interpretador de Python como o Jython que faz a tradução do código para ser executado na máquina virtual de Java. Uma visão geral do funcionamento do interpretador é apresentado na Figura 6.

2.2 CONSTRUÇÃO DO PYBLOCK

Nesta seção se abordou os aspectos técnicos da construção do *PyBlock*.

2.2.1 VISÃO GERAL DO PyBlock

A disponibilização do interpretador de Python para códigos escritos em linguagem C é de extrema importância em algumas aplicações que necessitam de velocidade de prototipação ou da grande quantidade de bibliotecas disponíveis em linguagem Python. No caso deste trabalho, tem-se a necessidade de ambos, pois necessita-se de um bloco de programação no PSIM para se testar métodos de controle sem a necessidade de uma grande performance em tempo de simulação, ou seja, pretende-se com este trabalho diminuir o tempo de engenharia para validar o método de controle projetado, mesmo que aumentando o tempo de simulação. Adicionalmente, também tem-se o objetivo de fornecer a grande gama de bibliotecas de cálculo ao projetista.

Logo, combina-se uma biblioteca de vínculo dinâmico ao interpretador de Python, mais especificamente, uma API permite o interpretador ser inicializado dentro da biblioteca de vínculo dinâmico. Desta forma, a gestão dos códigos Python a serem carregados e

executados, as bibliotecas a serem incluídas e os dados a serem usados pelo Python são geridos diretamente pela biblioteca de vínculo dinâmico em linguagem C.

Dentro desta biblioteca deve-se realizar primeiramente a gestão do interpretador de Python, ou seja, a correta inicialização do interpretador o qual cria as tabelas dos módulos carregados e também carrega os módulos fundamentais para a execução de um arquivo *.py* como o *builtins*, ___ *main* ___ e *sys*.

O Python a ser usado deve ser indicado dentro da biblioteca de vínculo dinâmico, já que uma máquina pode possuir muitas versões de Python instaladas. Optou-se por utilizar o *software* Anaconda para o controle de versão, onde um ambiente especial foi configurado para abrigar o interpretador de Python e as bibliotecas disponíveis ao interpretador a serem usados pelo PSIM.

O Python é uma linguagem que não necessita de tipos na declaração das variáveis, mas durante a atribuição de valor, a variável recebe o tipo do valor numérico ou do objeto que lhe foi atribuído. O tipo de uma variável em Python pode ser numérico como *int*, *float* ou *complex*, mas também pode lhe ser atribuído um objeto oriundo de uma determinada classe uma vez que Python é uma linguagem orientada objeto.

A linguagem C, em contraponto, necessita que as variáveis sejam previamente declaradas, e valores posteriormente atribuídos devem ser do mesmo tipo. Além disso, C não é uma linguagem orientada objeto, não havendo possibilidade de definir classes e atribuir a uma variável um objeto. Em consequência, o compartilhamento de dados entre códigos em Python e C não pode ser feito sem uma prévia conversão de tipos. Por exemplo, neste trabalho os dados em memória armazenados como *double* na linguagem C foram transformados em um Python *float*, e ponteiros de *arrays* em C foram transformados em Python *tuples* e posteriormente em objetos da classe *Numpy* com o objetivo de se usufruir das funcionalidades desta classe. Logo isto mostra a necessidade de se trabalhar os tipos de dados de cada linguagem para se poder compartilhar informação entre ambas.

Uma das etapas fundamentais no desenvolvimento do *PyBlock* é comunicação entre as diferentes linguagens e programas utilizados. Logo os objetos Python declarados dentro da biblioteca de vínculo dinâmico, como os valores a serem enviados a uma determinada função Python, são armazenados no código C com o tipo *PyObject* disponibilizado pela API. Utilizando-se dos chamados ponteiros opacos pode-se construir objetos de Python dentro do código C.

A gestão da memória dos objetos Python criados em C também deve ser realizada dentro da biblioteca de vínculo dinâmico. O *garbage collector* usa um *reference counting*, ou seja, ele mantém uma lista de referências de um determinado objeto com o intuito de ter o controle da memória utilizada por esta variável. Através desta lista de referência à variáveis do tipo *PyObject* pode-se desalocar a memória. A desalocação implementada

Figura 7 – *PyBlock*.

Fonte: Autor

pelo *garbage collector* acontece quando o *reference count* de uma determinada variável é posto em zero.

A Figura 7 mostra os blocos que compõem o *PyBlock* e também a sequência dos passos de comunicação entre os mesmos a cada passo de simulação. Os pontos 1 a 4 são definidos como segue:

1. o PSIM simula a equação do circuito e envia os valores à biblioteca de vínculo dinâmico.
2. Na biblioteca de vínculo dinâmico, o interpretador de Python é inicializado e em seguida são executados todos processos de carregamento de código na memória, transformação de tipos, etc.
3. Os valores são calculados pela função em Python e em seguida são retornados ao código em C.
4. Os valores são salvos em um ponteiro de *array* que aponta para um endereço de memória o qual o PSIM tem acesso.

Uma das vantagens já mencionada da utilização de Python é a facilidade de depuração do código. Para programar o algoritmo utilizou-se do *software* Spyder que fornece ferramentas de correção de código e um interpretador Python para executar o código de maneira simples.

Antes de interligar o código Python à DLL, utilizou-se do Spyder para executar o código Python. Este primeiro passo possibilita isolar os possíveis erros na cadeia de simulação numérica. Esta prática de isolar os blocos possibilita se utilizar de *softwares* de desenvolvimento para diminuir a probabilidade de um erro durante a interligação. Os testes de todos os blocos foram abordados na próxima seção.

2.2.2 PYTHON EMBARCADO

Existem algumas maneiras de se executar um código Python em uma aplicação em linguagem C. Optou-se por fazer uma integração com o módulo Python, pois isto possibilita executar funções que estejam dentro deste módulo independente do tamanho deste código e também de maneira simples. Fez-se a escolha da API para embarcar Python. Alguns passos gerais da construção são:

1. Inicializar o interpretador de Python.
2. Executar o código Python.
3. Finalizar o interpretador de Python.

O CPython, implementação do interpretador de Python em linguagem C a ser utilizada neste trabalho, disponibiliza uma API. Esta API possibilita dois caminhos, podendo disponibilizar um interpretador de Python ao código C/C++ ou também criar *extension modules*, ou seja, disponibilizar código em linguagem C à programas escritos em Python.

A interação entre o código C e o código Python se dá por meio de ponteiros opacos. Este ponteiro opaco representa um objeto Python qualquer escrito em C e pode ser declarado no código C como *PyObject**.

A seguir, são detalhados os passos principais na construção do código principal do *PyBlock*. Primeiramente, a biblioteca *Python.h* deve ser adicionada, como mostra o código

```
#include <Python.h>
```

isto garante que o interpretador de linguagem Python poderá ser inicializado. No momento de compilação, um diretório de inclusão adicional deve ser configurado para que o compilador seja capaz de encontrar a biblioteca. Importante ressaltar que este diretório deve ser o da versão Python a ser usada no interpretador e que as bibliotecas a serem utilizadas posteriormente sejam compatíveis com a versão Python selecionada. O segundo passo é configurar os arquivos do vinculador, que deve receber o nome da biblioteca com extensão *.lib* nas dependências adicionais durante a compilação.

Por terceiro, um arquivo *.def* também deve ser escrito obrigatoriamente por exigência do PSIM, caso ele não esteja disponível o PSIM não executa a simulação. Dentro deste arquivo estabelece-se apenas a função que será exportada como mostra o código

```
; dll_simples.def : Declares the module parameters for the DLL.

LIBRARY     "dll_simples"
```

```
DESCRIPTION 'Test DLL Block For PSIM'

EXPORTS
   simuser
```

Também no momento de compilação, o arquivo de definição de módulo deve ser passado ao vinculador, caso este passo não seja realizado a comunicação não será executada pelo PSIM.

Um ponto importante antes de inicializar a biblioteca de vínculo dinâmico é a definição primeiramente de qual versão do Python será usada. Logo, antes da inicialização do interpretador foi definido os caminhos das bibliotecas para o interpretador de linguagem Python a ser inicializado. O código mostra a definição dos caminhos que serão usados pelo interpretador

```
const wchar_t* pylibs = L"C:\\Users\\mathe\\Anaconda3\\python37.zip";
const wchar_t* pypath = L"C:\\Users\\mathe\\Anaconda3\\python37.zip;
C:\\Users\\mathe\\Anaconda3\\DLLs;
C:\\Users\\mathe\\Anaconda3\\lib;C:\\Users\\mathe\\Anaconda3;
C:\\Users\\mathe\\Anaconda3\\lib\\site-packages;
C:\\Users\\mathe\\Anaconda3\\lib\\site-packages\\win32;
C:\\Users\\mathe\\Anaconda3\\lib\\site-packages\\win32\\lib;
C:\\Users\\mathe\\Anaconda3\\lib\\site-packages\\Pythonwin;";
```

A função exportada pela biblioteca de vínculo dinâmico obrigatoriamente deve ter o nome *simuser* e os tipos de dados também devem ser do tipo *double* da linguagem C como mostra o código abaixo

```
__declspec(dllexport) void simuser(double t, double delt, double* in, double*
    out){}
```

Antes de se inicializar o interpretador ainda deve-se setar as variáveis de ambiente de Python

```
Py_SetPythonHome(pylibs);
Py_SetPath(pypath);
```

Após este passo, pode-se inicializar o interpretador

```
Py_Initialize();
```

Após o interpretador estar inicializado, pode-se importar e usar qualquer módulo Python de forma equivalente pelo código C utilizando-se dos ponteiros opacos disponibili-

zados pela API. Para executar um código primeiramente necessita-se do módulo que está inserido o código, o qual deve ser carregado pelo interpretador. Considere o exemplo a seguir:

```
pName = PyUnicode_DecodeFSDefault(name_script);
pModule = PyImport_Import(pName);
Py_DECREF(pName);
```

Nesse código, definiu-se o *name_script* como sendo o nome do módulo a ser carregado, faz-se a devida conversão de *char* para um *Pyobject** e importa-se o módulo através da função *PyImport_Import(pName)*, a qual retorna um novo *Pyobject**. O *Py_DECREF()* será explicado posteriormente. Através do *Pyobject** pode-se acessar qualquer função que esteja dentro deste módulo. Criando-se um *Pyobject** para esta função pode-se ter a permissão, por parte do interpretador, de executá-la

```
pFunc = PyObject_GetAttrString(pModule, name_function);
```

Após este passo, pode-se executar a função, sendo *pArgs* os argumentos da função os quais terão sua construção explicada posteriormente. O *pValue* recebe o retorno dessa função, podendo-se realizar a conversão de tipos.

```
pValue = PyEval_CallObject(pFunc, pArgs);
```

2.2.3 CONVERSÂO DE TIPOS

A conversão de tipos neste trabalho deve ser feita do formato de C para o formato de Python e também ao contrário. Para o primeiro caso, usam-se funções que transformam os tipos de C em ponteiros opacos do tipo **PyObject*. Ressalta-se que no código C, o tipo de objeto no código Python não pode ser visto no código C, pois todos são do mesmo tipo vistos dentro da biblioteca dinâmica: **PyObject*. A distinção ocorre quando se chama a função que cria o tipo da variável em Python. Como no código:

```
PyObject* ob1 = PyFloat_FromDouble(in[0]);
```

Caso *ob1* seja enviado a uma função Python, ele será tratado pelo interpretador como do tipo Python *float*.

A conversão de tipos de Python para C também acontece passando por ponteiros opacos **PyObject*. Por exemplo, sabe-se que uma determinada função Python retorna uma Python *tuple* e quer-se transformar cada valor da tuple em *double* e armazenar em variáveis no código C. A função no código abaixo realiza esta operação usando o *pValue* que foi retornado da execução da função pelo interpretador

```
PyArg_Parse(pValue, "(dddddd)", &sa1, &sa2, &sb1, &sb2, &sc1, &sc2);
```

Com estas duas abordagens pode-se converter qualquer tipo de dados de Python para C e também ao contrário. A criação de tipos de dados Python compostos, como uma *tuple*, necessita de um passo adicional. Neste caso, deve-se criar um **PyObject* e posteriormente alocar os valores em cada posição da *tuple*, sendo estes valores também do mesmo tipo C

```
//Defini-se o tamanho da tuple
const Py_ssize_t len = 5;
//Inicializa o objeto tuple
pyTuple = PyTuple_New(len);

//Converte-se os valores de double para python float
PyObject* ob1 = PyFloat_FromDouble(in[0]);
PyObject* ob2 = PyFloat_FromDouble(in[1]);
PyObject* ob3 = PyFloat_FromDouble(in[2]);
PyObject* ob4 = PyFloat_FromDouble(in[3]);
PyObject* ob5 = PyFloat_FromDouble(in[4]);

//Aloca-se os valores na tuple na ordem desejada
PyTuple_SET_ITEM(pyTuple, 0, ob1);
PyTuple_SET_ITEM(pyTuple, 1, ob2);
PyTuple_SET_ITEM(pyTuple, 2, ob3);
PyTuple_SET_ITEM(pyTuple, 3, ob4);
PyTuple_SET_ITEM(pyTuple, 4, ob5);
```

Após a execução deste código tem-se uma *tuple* em Python.

Em seguida deve-se construir o argumento da função que será passado para a função em Python. Para a construção dos argumentos requisitados pela função deve-se conhecer os tipos definidos de cada argumento de entrada da mesma. Caso a função Python necessite apenas de uma *tuple* pode-se construir o argumento desta função com o código

```
pArgs = Py_BuildValue("(O)", pyTuple);
```

A função acima executada pela DLL constrói o argumento da função Python e define os tipos destes argumentos. O primeiro argumento requerido da função *Py_BuildValue* é o tipo de variável que será enviada a função Python. No caso deste exemplo, *"(O)"* significa que está se enviando um argumento do tipo **PyObject* puro e que não deve ser feita

nenhuma conversão prévia. O segundo argumento da função é o próprio objeto Python já construído através de ponteiros opacos.

Neste exemplo, quando a função *PyEval_CallObject* é executada pelo código em C envia-se juntamente os argumentos da função Python, neste caso *pArgs*.

2.2.4 GESTÃO DA MEMÓRIA E DETECÇÃO DE ERROS

O CPython oferece maneiras de se evitar vazamentos de memória. Através de um *reference counting* pode-se fazer a gestão da memória, ou seja, quando se tem uma nova referência identificada ao objeto o *reference counting* é incrementado. E quando o *reference counting* é decrementado uma referência ao objeto foi deletada. O *CPython* também oferece um detector de ciclos, executado de maneira periódica, caso a referência ao objeto for possuída por ele mesmo.

Quando utiliza-se a linguagem Python através de um interpretador não embarcado a gestão da memória através do *reference count* é realizada automaticamente pelo *software* Python. No entanto, quando o interpretador é embarcado em linguagem C a gestão da memória fica a cargo do código em C. Portanto, a API disponibiliza duas macros para que o programa em C tenha a possibilidade de fazer a gestão da memória, são elas: *Py_INCREF()* e *Py_DECREF()*.

Um detalhe importante na implementação desta gestão de memória é que cada função disponibilizada pela API para comunicação entre a biblioteca de vínculo dinâmico e o algoritmo em Python se comporta de maneira distinta em relação a referência do contador. Existem duas possibilidades no retorno de uma função da API chamada na *DLL*.

No primeiro tipo o objeto retornado recebe uma nova referência, e já no segundo tipo uma referência é emprestada, ou seja, o *reference count* do objeto é emprestado pela função. Portanto, deve-se estudar cada função caso a caso durante estas chamadas realizadas através da API e entender se o objeto recebe uma nova referência que deve ser decrementada posteriormente ou não.

Na chamada de uma função, o mesmo acontece, ou seja, existem duas possibilidades com os objetos passados como argumento, a função pode assumir a responsabilidade pela referência e se tornar responsável pelo *reference count* ou não manter a referência. Neste trabalho, a função utilizada que assume o referenciador dos objetos é a *PyTuple_SetItem()*.

Quando declara-se objetos do tipo *PyObject* no código, deve-se realizar a devida gestão da memória. No caso, deve-se informar ao Python para liberar a memória caso não se precise mais de um determinado objeto (IBM, 2010). Usa-se a função *Py_DECREF()* para realizar um decremento no *reference count* do objeto. Em Python, quando o *reference count* é zero, o objeto é automaticamente deletado como parte do processo sob responsabilidade do *garbage collector*.

Na construção do código em C, os objetos Python são sempre deletados. Para exemplificar este processo tem-se o código abaixo, onde o *pName* não será mais usado após o carregamento do módulo Python e a função de importação não se apropriou a referência do objeto.

```
pModule = PyImport_Import(pName);
Py_DECREF(pName);
```

No caso de haver algum erro de carregamento do módulo Python ou da função escrita em Python, as funções de chamada retornam *NULL*. Também é possível checar se a função pode ser carregada e executada com o intuito de se certificar de que uma função foi carregada corretamente, isto pode ser feito pela função abaixo chamada de dentro da DLL e disponibilizada pela API

```
PyCallable_Check(pFunc)
```

Além desta forma de se checar se o código está sendo executado da maneira correta, pode-se checar se ocorreu algum erro de execução durante o processo executado pelo interpretador do Python através das funções

```
PyErr_Occurred()
PyErr_Print()
```

2.2.5 CONSTRUÇÃO DOS BLOCOS DE CÓDIGO E COMUNICAÇÃO

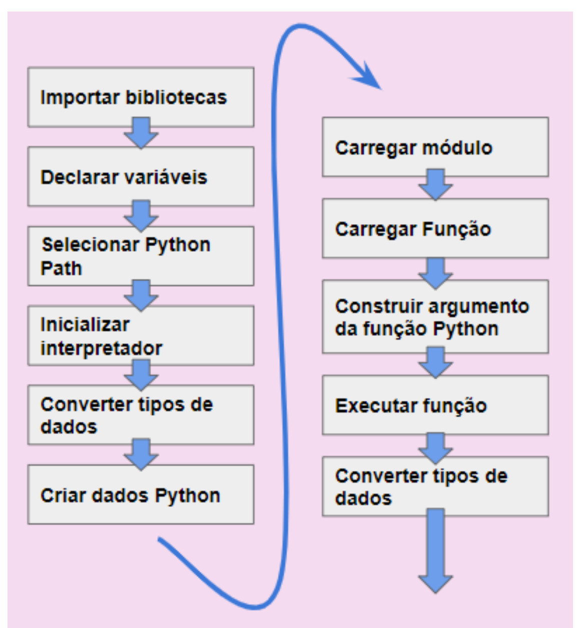
A construção do código C é feita levando-se em conta os pontos apresentados nas seções passadas. Na Figura 8 está descrita a sequência de operações do algoritmo, sendo cada bloco um passo executado pelo mesmo. Alguns pontos desta execução foram descritos em código nas seções passadas e ajudam a compreender o passo a passo.

No passo *Executar função* ocorre a execução do código em Python. Neste momento, o interpretador de Python inicializado na aplicação compila o código C e cria um *bytecode*, este *bytecode* é enviado à máquina virtual de Python, que acrescenta os módulos ao código e executa os comandos, então os resultados são salvos e armazenados em uma *tuple* enviada à biblioteca de vínculo dinâmico. Este processo está resumido na Figura 9.

2.3 TESTES

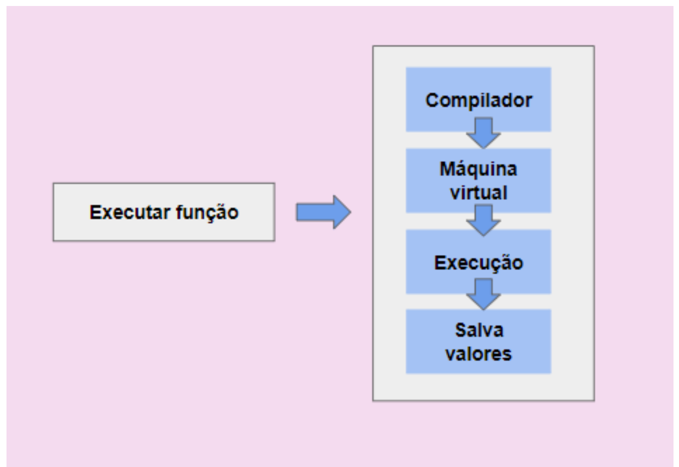
Nesta seção foram abordados todos os aspectos técnicos dos testes realizados com o *PyBlock* e os resultados de cada teste.

Figura 8 – Algoritmo da função em linguagem C em blocos.



Fonte: Autor

Figura 9 – Detalhes do bloco *Executar função*.



Fonte: Autor

2.3.1 TESTES UNITÁRIOS

Os testes unitários são responsáveis por verificar um pequeno pedaço de código, conhecido como unidade, de forma rápida e de forma isolada (KHORIKOV, 2019).

Neste trabalho se desenvolveu testes isolados para cada funcionalidade. Primeiramente executou-se testes unitários no código em Python. Posteriormente, procedeu-se comparando os resultados do algoritmo com resultados obtidos previamente com o algoritmo em Matlab. Checou-se se todas as saídas correspondiam aos valores esperados. A plataforma utilizada foi o programa *Spyder* que fornece ferramentas para testar códigos e uma fácil visualização das entradas, saídas e manipulações de dados.

O código da DLL foi validado e testado através da construção de um arquivo executável em C. Este arquivo executável carrega a DLL e aplica sinais de entrada e compara com os valores esperados da saída. O *Software Visual Studio* foi utilizado para escrever o código e compilação dos arquivos. Nenhum erro foi acusado pelo teste unitário realizado e todos os valores de saída estavam corretos.

2.3.2 TESTES DE INTEGRAÇÃO

O teste de integração é responsável por validar as interfaces que fazem parte do sistema. Este teste tem o objetivo geral de validar e simular as interações entre as unidades.

O teste de integração foi realizado primeiramente entre os blocos comunicantes dois a dois, posteriormente um teste de todo sistema foi realizado. Na construção do *PyBlock*, antes de se realizar o teste geral de integração, foram realizados testes com os seguintes blocos comunicantes dois a dois: PSIM com DLL e DLL com Python.

2.3.2.1 PSIM E DLL

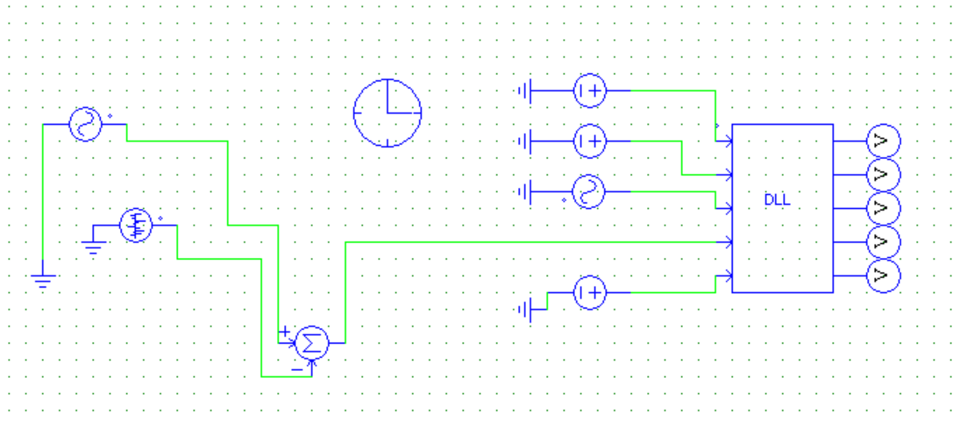
Para validação destes blocos deve-se isolar o sistema PSIM e DLL da parte em Python. Para isto se utilizou da DLL, todavia se comentou a parte do código que implementa o interpretador de linguagem Python e se implementou operações simples em linguagem C, que devolvem os valores alterados ao PSIM.

O Bloco PSIM com DLL foi validado através de um simples esquema no PSIM, o qual envia mensagens correspondentes à níveis de tensão para a DLL que realiza operações de divisão e soma e retorna os valores ao PSIM. O bloco pode ser visto na Figura 10. Diferentes níveis de tensão foram testados na entrada e a validação foi feita através da comparação do sinal de saída com o sinal de entrada. Neste teste o objetivo era validar se os blocos estavam comunicando-se de maneira adequada e se os valores mantinham-se coerentes e sem atraso entre a entrada e a saída. Testes de estresse também foram realizados como a implementação de um *loop for* com um contador apenas para atrasar a resposta ao PSIM. Todos os testes foram bem sucedidos entre o PSIM e o bloco DLL, sem atraso e os cálculos realizados foram retornados no tempo correto sem problemas numéricos.

2.3.2.2 DLL E PYTHON

Este teste de integração teve como objetivo testar a comunicação entre os blocos DLL e o código Python de maneira isolada, ou seja, a DLL não deve ser chamada pelo PSIM e sim por um outro código que faça uma chamada a DLL. Para realizar este teste foi implementado um código executável em linguagem C que faz a chamada a DLL. O código compilado para esta DLL está implementado juntamente com o interpretador de Python.

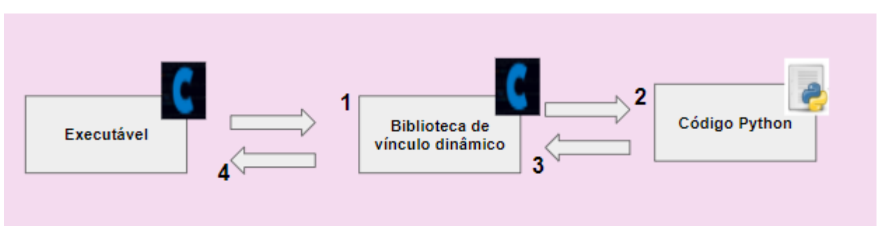
Figura 10 – Esquema de teste do PSIM com DLL.



Fonte: Autor

O algoritmo em linguagem C primeiramente carrega a DLL e envia um ponteiro com os valores definidos no executável, simulando uma chamada do *software* PSIM. Na sequência, a DLL envia os valores ao algoritmo Python que realiza pequenas operações de soma e divisão e retorna os valores à DLL que então retorna ao algoritmo em C. Um resumo desta sequência está na Figura 11.

Figura 11 – Teste de integração da DLL com Python.



Fonte: Autor

Primeiramente, importam-se as bibliotecas

```
#include <stdio.h>
#include <conio.h>
#include <math.h>
#include <dll_simples.h>
```

e posteriormente chama-se a DLL através da função disponibilizada pela biblioteca *dll_simples.h*.

```
int main() {
   double in[] = { 0,-57.54426992,352.26959034,-62.11181733,799.9997470514371
      };
   double in_2[] = { 1,-322.5,15,-5,550 };
   double out[4] = {};
```

```
double a = 0;
double b = 0;
int nb_repetitions = 250;

for(int i=0;i<nb_repetitions;i++){

   if (i%5 == 0){
      printf("\nsimulation step: %d", i);
      printf("\nEnviando %lf: ", *in);
      simuser(a, b, in, out);
      printf("\nEnviando %lf: ", *in);
      printf("\nsa1: %lf ", *out);
      printf("\nsa2: %lf ", *(out + 1));
      printf("\nsb1: %lf ", *(out + 2));
      printf("\nsb2: %lf ", *(out + 3));
      printf("\n xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx");
   }
   else {
      printf("\nsimulation step: %d", i);
      printf("\nEnviando %lf: ", *in_2);
      simuser(a, b, in_2, out);
      printf("\nEnviando %lf: ", *in_2);
      printf("\nsa1: %lf ", *out);
      printf("\nsa2: %lf ", *(out + 1));
      printf("\nsb1: %lf ", *(out + 2));
      printf("\nsb2: %lf ", *(out + 3));
      printf("\n xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx");

   }
}
printf("\nClosing C function...");
```

Neste teste, foi detectado o primeiro problema de implementação. Um erro de memória impedia o código de ser executado em sequência durante o teste de estresse do sistema.

O erro *AccessViolationException* acontece devido à uma biblioteca chamada dentro do código Python e que é compilada junto da DLL. A biblioteca *Numpy* ocasiona este erro pois em sua implementação ainda não está previsto múltiplas inicializações do interpretador. Este comportamento não é um comportamento comum e é considerado um *bug*. Este problema foi reportado aos desenvolvedores da biblioteca como indicado na própria página oficial da biblioteca (NUMPY. . . ,) e que até este momento não foi disponibilizado solução.

A biblioteca Numpy é uma biblioteca chave para uma série de aplicações e abrir mão do seu uso acarretaria em uma complexidade muito maior no algoritmo em Python. Portanto, optou-se por desenvolver uma solução para este problema. O erro é ocasionado apenas se o interpretador é aberto e fechado mais de uma vez pela mesma biblioteca de vínculo dinâmico em sequência, como neste caso onde a DLL faz uma chamada ao interpretador a cada passo de simulação. Desta forma, para que seja possível a utilização da biblioteca *Numpy*, inicializou-se apenas uma vez o interpretador durante a primeira chamada e posteriormente apenas se checou se o interpretador já estava inicializado.

```
if (!Py_IsInitialized()) {
    // Set Python Home and Python Path -> Escolhe o Python correto dentro do
       PC.
    Py_SetPythonHome(pylibs);
    Py_SetPath(pypath);
    Py_Initialize();
 }
```

O Interpretador é finalizado apenas no fim da simulação. Após este passo todos os testes foram bem sucedidos e nenhum erro adicional foi detectado.

2.3.2.3 TESTE INTEGRADO DE TODOS BLOCOS

Este teste visa testar todos os blocos internamente, a comunicação entre eles e o compartilhamento de informação. Este teste deve percorrer toda cadeia de comunicação começando no PSIM, passando pela biblioteca de vínculo dinâmico, o algoritmo em Python bem como a cadeia inversa após o cálculo do algoritmo.

Para este teste, se construiu um circuito no PSIM que gera tensões e posteriormente estas tensões são alteradas em Python no algoritmo que responde com as novas tensões. Para fins de validação compara-se as tensões de saída com as tensões de entrada. Utilizou-se o mesmo bloco da Figura 10, apenas substituiu-se o bloco DLL pelo *PyBlock*.

Neste teste de integração encontrou-se um dos problemas mais difíceis de se solucionar na construção do *PyBlock*, pois o PSIM não é um *software* livre, ou seja, não tem-se acesso a seu código nem às informações de erro produzidos internamente. Isto acarreta em uma dificuldade muito grande de depurar possíveis problemas de comunicação, já que um dos blocos do sistema é visto como uma caixa preta sem acesso externo.

No primeiro teste, com um algoritmo no PSIM e em Python que apenas realiza operações matemáticas simples, nenhum problema foi detectado e a primeira vista o sistema estava funcionando de maneira adequada.

No entanto, quando testou-se o sistema com um algoritmo de maior complexidade e com um circuito com um maior número de componentes, como um conversor multinível

com neutro grampeado e um algoritmo de otimização juntamente com o algoritmo *solver* de LMI's ocorreu um comportamento inesperado: o PSIM abortava a execução da simulação e fechava automaticamente sem nenhuma mensagem de erro.

Este comportamento anômalo não está documentado no manual do PSIM, e também nenhum erro é gerado por parte do PSIM ou mostrado ao usuário através da interface. Desta forma, teve-se primeiramente a ideia de se acessar os *logs* de erro e avisos gerados pelo *Windows* durante a execução do programa PSIM.

Percebeu-se através destes *logs* que o PSIM demorava muito tempo inicializando os blocos da simulação e inicializando o interpretador através da DLL. O fechamento do programa acontece devido à um *timeout* que fiscaliza a execução do programa e interrompe sua execução caso haja uma demora demasiada na resposta da DLL para prosseguir com a simulação.

Para sobrepor este obstáculo primeiramente criou-se um arquivo do PSIM que apenas carrega a DLL. Este arquivo deve ser executado pelo PSIM sempre antes de se executar a simulação de um circuito maior. Esta possível solução foi adotada por que a execução do código de carregamento dos blocos e inicialização da simulação por parte do PSIM sobrecarrega o programa, se executado juntamente com o carregamento da DLL com o Python embarcado. Logo, separou-se em dois arquivos PSIM distintos a simulação do arquivo e o carregamento da DLL. Este novo sistema resolveu o problema encontrado e não resultou em outros erros.

Através deste arquivo, pode-se testar toda a cadeia de comunicação entre os blocos. Testes de estresse também foram realizados afim de validar a comunicação como a inserção de *timesleep* no código em Python e *loop for* no código em C.

Todos os testes foram bem sucedidos. Logo, o *PyBlock* está finalizado e pronto para ser usado acoplado ao PSIM.

2.3.2.4 ARMAZENAMENTO DE INFORMAÇÃO

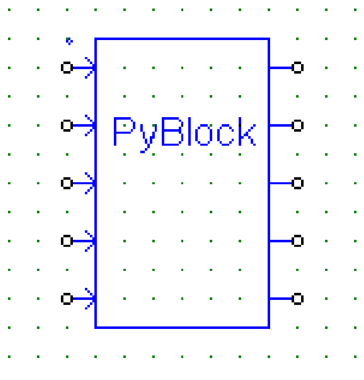
Armazenar informação a respeito da simulação é de extrema importância para validar os resultados ou entender o comportamento do circuito e do sistema de controle. Nos algoritmos implementados neste trabalho teve-se o cuidado de se gerar *logs* que descrevem valores de variáveis e comportamentos da simulação ao longo do tempo.

Estes *logs* trazem uma outra vantagem da utilização de Python, pois seu uso é simples e os mesmos podem ser conferidos ao longo da simulação por parte do usuário. Para isto utilizou-se da própria biblioteca de escrita da linguagem Python em formato *.txt*. Exemplos de arquivos gerados podem ser vistos no Apêndice A.

2.4 INSTALAÇÃO E EXEMPLO

A versão final do *PyBlock* é composta por 3 partes: i) O bloco *DLLBlock* do PSIM mostrado na Figura 12; ii) A biblioteca de vínculo dinâmico com o interpretador de Python; iii) O código Python com as funções de cálculo.

Figura 12 – *PyBlock* no PSIM.



Fonte: Autor

A instalação do *PyBlock* é composta de alguns passos. Primeiramente, o arquivo com o código raiz do bloco deve ser aberto e as seguintes configurações ajustadas de modo a ficar de acordo com o algoritmo e a implementação do PSIM:

1. Número de entradas do bloco PSIM.
2. Número de saídas do bloco PSIM.
3. Nome do arquivo Python.
4. Nome da função em Python.

Posteriormente, o arquivo raiz deve ser compilado na máquina para gerar um arquivo *.dll* como mostrado na Figura 13.

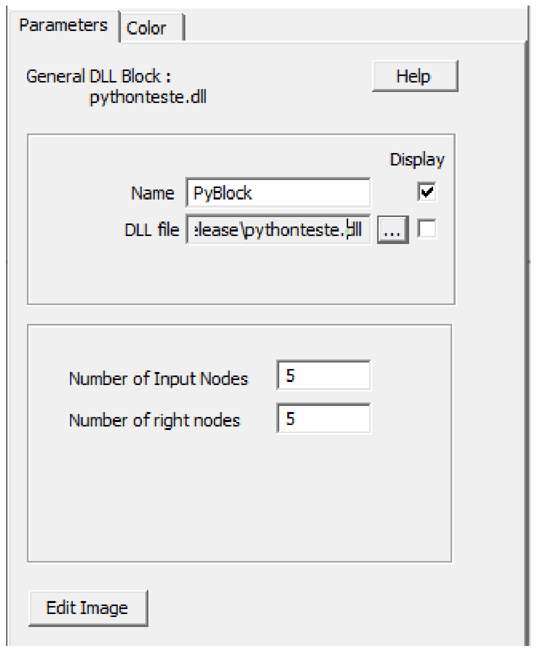
Figura 13 – Arquivos de compilação.

Nome	Data de modificação	Tipo	Tamanho
PYTHONTESTE.dll	19/08/2020 22:43	Extensão de aplica...	13 KB
PYTHONTESTE.exp	19/08/2020 22:43	Exports Library File	2 KB
PYTHONTESTE.iobj	19/08/2020 22:42	Arquivo IOBJ	21 KB
PYTHONTESTE.ipdb	19/08/2020 22:42	Arquivo IPDB	13 KB
PYTHONTESTE.lib	02/08/2020 20:50	Object File Library	2 KB
PYTHONTESTE.pdb	19/08/2020 22:43	Program Debug D...	340 KB

Fonte: Autor

O endereço do arquivo *.dll* é então passado para o bloco do PSIM no seletor *DLL file*, como mostrado na Figura 14

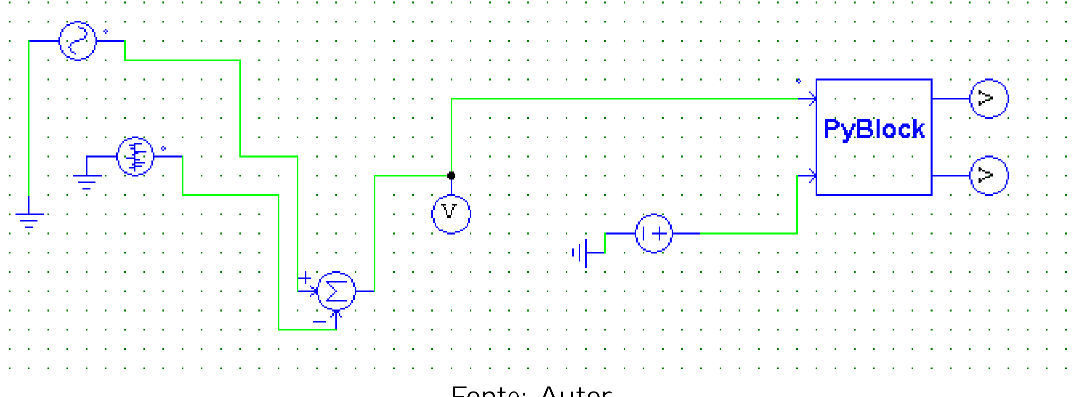
Figura 14 – Configurações do bloco no PSIM.



Fonte: Autor

Preparou-se um exemplo ilustrativo da utilização do *PyBlock*: o bloco recebe duas entradas, um sinal constante e uma senoide corrompida por ruído, e retorna duas saídas, o sinal constante original e a senoide com amplitude reduzida pela metade. A Figura 15 apresenta os blocos de simulação deste exemplo

Figura 15 – Exemplo de uso do *PyBlock*.



Fonte: Autor

O código em Python(descrito no arquivo teste.py) recebe os valores do PSIM e retorna no caso da senoide o valor de entrada dividido por 2 e no caso da constante retorna o mesmo valor

---

```
def teste(inputData):
```

```
    return (inputData[0]/2,inputData[1])
```

Este código deve ser posto na mesma pasta do arquivo PSIM como mostrado na Figura 16.

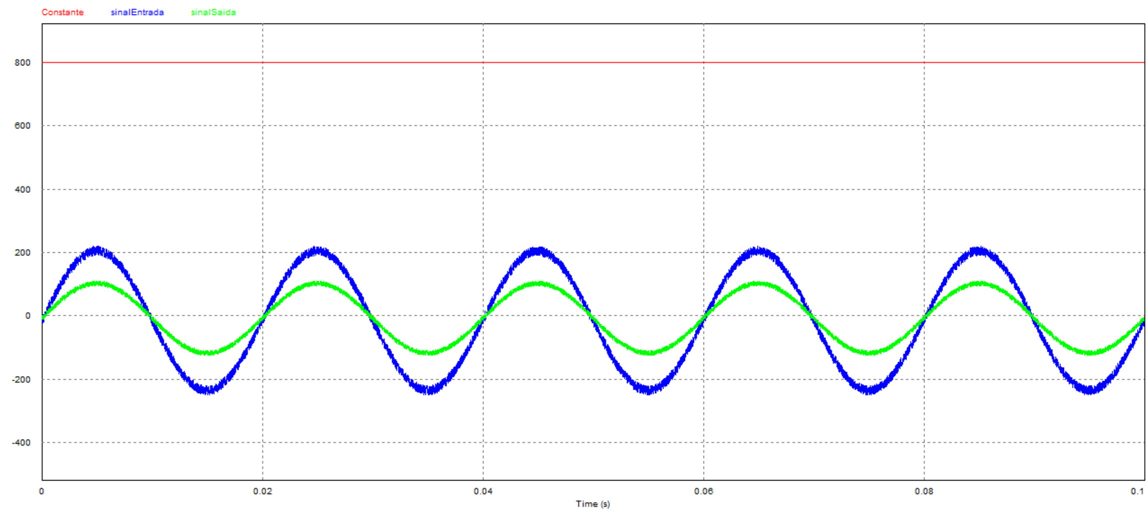
Figura 16 – Arquivos de simulação.

simples.psimsch	02/11/2020 18:30	Arquivo PSIMSCH	11 KB
simples.smv	02/11/2020 18:31	Arquivo SMV	287 KB
teste.py	02/11/2020 18:31	Python File	1 KB

Fonte: Autor

Por fim, apresenta-se o resultado da simulação do exemplo na Figura 17 com a curva azul representando o sinal de entrada, a curva verde o sinal de saída após divisão e a curva vermelha o sinal constante.

Figura 17 – Resultado da simulação do exemplo.



Fonte: Autor

2.5 CONCLUSÃO DO CAPÍTULO

Neste capítulo, foi definido a motivação da construção do *PyBlock*, bem como o desenvolvimento do mesmo e todas as fases da construção como a escolha da linguagem, escrita do código, integração dos blocos, testes unitários e de integração e problemas de construção. Logo, este capítulo apresenta de forma completa toda a construção do bloco. Os próximos capítulos definem os conversores a serem implementados bem como o projeto dos métodos de controle e a implementação através do *PyBlock* e por fim os resultados.

3 INVERSOR PWM TRIFÁSICO

Neste capítulo foi apresentado o projeto de um controlador proporcional-integral(PI) projetado através de LGR para o controle de um conversor trifásico de dois níveis. Os resultados da simulação deste circuito foram obtidos através do *software* PSIM através do blocos PI, que é nativo do PSIM, e do *PyBlock*.

3.1 TOPOLOGIA DO INVERSOR TRIFÁSICO DE DOIS NÍVEIS

Inversores trifásicos de dois níveis são utilizados em aplicações de todos os níveis de potência. A Figura 18 apresenta uma possível topologia de inversor trifásico, onde as chaves S_1 a S_6 representam dispositivos semicondutores como IGBTS. A tensão CC está representada pela fonte de tensão V_i e cada ponto a, b, c representa uma fase diferente do circuito trifásico.

Figura 18 – Inversor de trifásico de dois níveis.

Fonte: (RASHID, 2017)

Com esta topologia são possíveis 8 combinações de estados das chaves no total. No entanto, 7 delas produzem resultados de tensão distintos como mostra a Tabela 1.

Tabela 1 – Estados do inversor de 2 níveis

Chaves ligadas	Estado	V_{ab}	V_{bc}	V_{ca}
S_1 S_2 S_6	1	V_i	0	$-V_i$
S_2 S_3 S_1	2	0	V_i	$-V_i$
S_3 S_4 S_2	3	$-V_i$	V_i	0
S_4 S_5 S_3	4	$-V_i$	0	V_i
S_5 S_6 S_4	5	0	$-V_i$	V_i
S_6 S_1 S_5	6	V_i	$-V_i$	0
S_1 S_3 S_5	7	0	0	0
S_4 S_6 S_2	8	0	0	0

Fonte: Autor

Esta topologia foi estudada neste trabalho e implementada através do *software PSIM*.

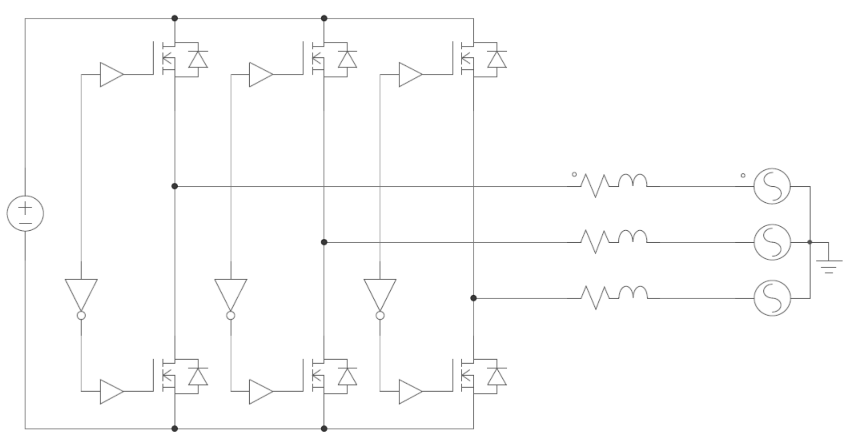
3.2 PROJETO DO CONTROLADOR PI E IMPLEMENTAÇÃO PYTHON

A seguir é apresentado o controle desse conversor no referencial síncrono a partir de um controlador proporcional-integral(PI).

3.2.1 FORMULAÇÃO GERAL

O sistema a ser controlado é um conversor de dois níveis trifásico CC-CA. Este conversor formado por seis chaves, sendo os transistores IGBT e MOSFET os mais comuns. Estes conversores são bastante comuns em aplicações industriais, como em projetos que demandam fornecimento ininterrupto de potência (TAHIR *et al.*, 2018). A Figura 19 mostra o circuito do inversor conectado à rede elétrica levando-se em conta o modelo RL série da linha.

Figura 19 – Circuito do inversor de 2 níveis.



Fonte: Autor

A transformação de *Park* do domínio *abc* para o domínio *dq*0 (KRAUSE *et al.*, 2002) foi definida inicialmente para a análise de máquinas elétricas e a modelagem do sistema. A transformação de domínio importante já que é possível transformar grandezas senoidais em sinais constantes.

Defini-se matematicamente esta transformação como

$$T_{abc}^{dq0} = \frac{2}{3} \begin{matrix} cos(\ ) & cos(\ -\frac{2}{3}) & cos(\ +\frac{2}{3}) \\ -sen(\ ) & -sen(\ -\frac{2}{3}) & -sen(\ +\frac{2}{3}) \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \end{matrix} \tag{3.1}$$

onde representa o angulo de rotação em cada instante de tempo da sinusoide t. A dinâmica da corrente injetada na rede em cada fase do sistema é definida majoritariamente pelo circuito RL. Assim, a equação diferencial deste circuito em uma fase no domínio *abc*

$$L\frac{di_a}{dt} + Ri_a(t) = V_a(t) - V_{Sa}(t) = \quad V_a(t) \tag{3.2}$$

aplicando-se a transformação *dq*0:

$$L\frac{di_d}{dt} + Ri_d(t) = \quad V_d(t) \tag{3.3}$$

$$L\frac{di_q}{dt} + Ri_q(t) = \quad V_q(t) \tag{3.4}$$

desta forma, através da aplicação da transformada de Laplace

$$I_d(s) = \frac{V_d(s)}{(Ls + R)} \tag{3.5}$$

$$I_q(s) = \frac{V_q(s)}{(Ls + R)} \tag{3.6}$$

regulando-se V_d e V_q é possível regular a injeção de corrente em cada uma das fases. Assim, o modelo genérico para as correntes independente se q ou d é dado por

$$G(s) = \frac{I(s)}{V(s)} = \frac{\frac{1}{L}}{(s + \frac{R}{L})} \tag{3.7}$$

Neste trabalho, escolheu-se o controlador PI(Proporcional Integral) para o controle do conversor, pois no domínio dq a referência de corrente é constante. Uma característica fundamental do controlador PI é a garantia de erro nulo de seguimento de referência em regime permanente para sinais do tipo salto. A função de transferência deste controlador se dá pela equação (3.8) (BAZANELLA; JUNIOR, 2005). Na próxima subseção foram projetados os ganhos do controlador PI para o conversor de dois níveis deste trabalho.

$$C(s) = k(1 + \frac{1}{T_i s}) \tag{3.8}$$

3.2.2 PROJETO DO CONTROLADOR

Baseado na referência (SILVA *et al.*, 2016) foram considerados os valores numéricos apresentados na Tabela 2, resultando em

$$G(s) = \frac{\frac{1}{L}}{(s + \frac{R}{L})} = \frac{10^3}{(s + 800)} \tag{3.9}$$

Tabela 2 – Valores das constantes do conversor.

Constantes	Componentes
L	1mH
R	0.8
f	50 Hertz
V_{dc}	[500,1000]
V^{saN}	220 V_{rms}

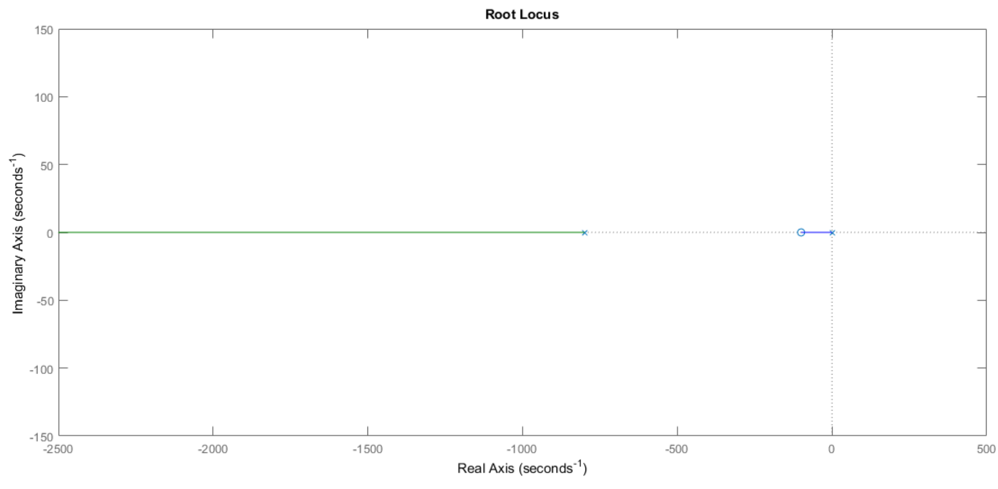
Fonte: autor

O projeto dos ganhos do controlador será baseado no método do lugar geométrico das raízes(LGR). Assumindo que o parâmetro variante é o ganho K_p, então segue que a função a ser analisada é

$$F(s) = \frac{1000(s + \frac{1}{T_i})}{s(s + 800)} \tag{3.10}$$

e que o zero do controlador esteja localizado em $s = -100$, tem-se o LGR mostrado na Figura 20.

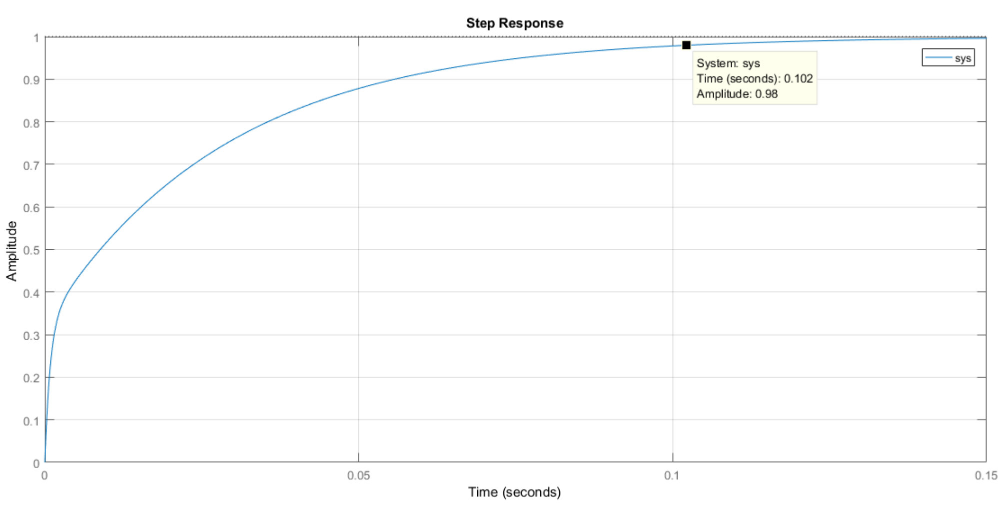
Figura 20 – Lugar das raízes de C(S)G(S).



Fonte: Autor

O ganho Kp foi definido como 0.4 tal que o tempo de acomodação em malha fechada seja de aproximadamente de 0.1 segundos. A Figura 21 mostra a resposta ao degrau do sistema com este controlador.

Figura 21 – Resposta ao degrau.



Fonte: Autor

O algoritmo do controlador PI está disponível em (BAZANELLA; JUNIOR, 2005). Sua sequência de passos pode ser resumida abaixo onde T é o período de amostragem

```
y = out_measure
error = ref - y
P = K * error
I = I_bef + K*T*(error+error_bef)/(2*Ti)

control_signal = P + I

error_bef = error
I_bef = I
```

O controlador foi implementado em linguagem Python. O algoritmo a seguir recebe os valores do PSIM, calcula o erro e em seguida o erro entra no controlador. Posteriormente, calcula-se a saída em forma de tensão para ambos os espaços V_d e V_q e devolve-se este valor ao PSIM.

```
import numpy as np
from os import listdir
import os.path
import os

def calculate_optimal_configuration(inputData):
    xe = np.array(list(inputData[:2]))[...,np.newaxis]
    x  = np.array(list(inputData[2:4]))[...,np.newaxis]
    vdc = inputData[-2]
```

```
delt = inputData[-1]

# Aproximando os arrays
xe = np.around(xe,decimals=4)
vdc = np.around(vdc,decimals=4)
x = np.around(x,decimals=4)
error = (xe-x)

mypath = os.path.dirname(os.path.realpath(__file__))
mypath = os.path.join(mypath, "data")

onlyfiles = [f for f in listdir(mypath) if
    os.path.isfile(os.path.join(mypath, f))]

if ("erroAnt.npy" in onlyfiles):
    erroAnt = np.load(os.path.join(mypath,"erroAnt.npy"))
    Iant    = np.load(os.path.join(mypath,"Iant.npy"))

else:
    erroAnt = np.array([[0],[0]])
    Iant = np.array([[0],[0]])

# Proporcional
kp1 = 0.4
kp2 = 0.4
vd_prop = kp1*error[0,0]
vq_prop = kp2*error[1,0]


# Integral
Ti  = 0.01
T = delt
ki1 = kp1
ki2 = kp2
vd_int = Iant[0,0] + ki1*T*(error[0,0]+erroAnt[0,0])/(2*Ti)
vq_int = Iant[1,0] + ki2*T*(error[1,0]+erroAnt[1,0])/(2*Ti)

text_file = open("code_optimization_notes.txt", "a")
text_file.write("ok")
text_file.close()

vd = vd_prop + vd_int
```

```
    vq = vq_prop + vq_int
    Vout = np.array([[vd],[vq]])

    np.save(os.path.join(mypath,"erroAnt"),error)
    np.save(os.path.join(mypath,"Iant"),np.array([[vd_int],[vq_int]]))

    return Vout
```

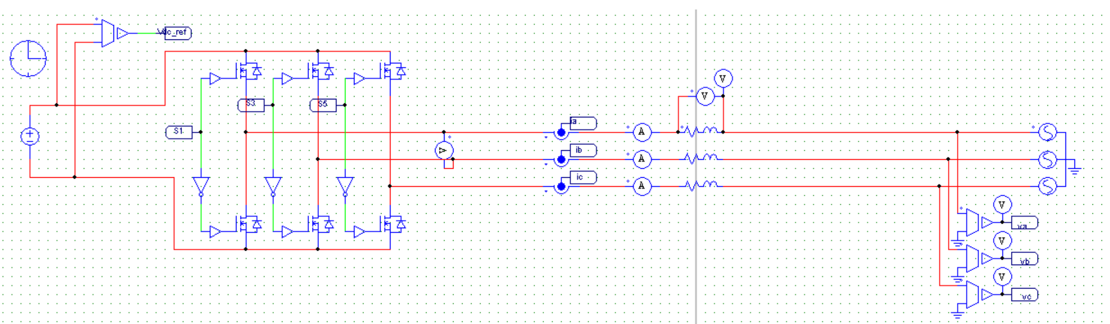
3.3 RESULTADOS DE SIMULAÇÃO

Nesta seção tem-se o objetivo de apresentar os resultados das simulações do controlador projetado na última seção para o inversor de dois níveis. O controlador PI foi implementado em código através de linguagem Python utilizando o bloco *PyBlock* e o bloco nativo PI.

3.3.1 PYTHON COM PSIM

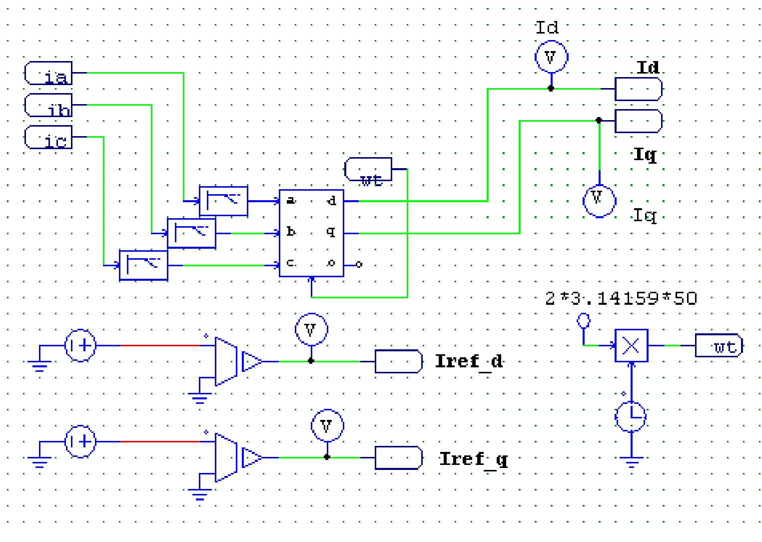
O circuito implementado consta na Figura 22 onde primeiramente implementou-se o conversor de dois níveis trifásico, ligou-se este conversor a uma rede trifásica com um indutor e uma resistência em série. Em adição, sensores de corrente e de tensão também foram utilizados para amostrar informação para o controlador. Logo abaixo, implementou-se o controlador com o *PyBlock*, juntamente com as conversões de tipos e a definição das chaves posteriormente ao PI como mostra as Figuras 23 e 24.

Figura 22 – Descrição do inversor no PSIM.



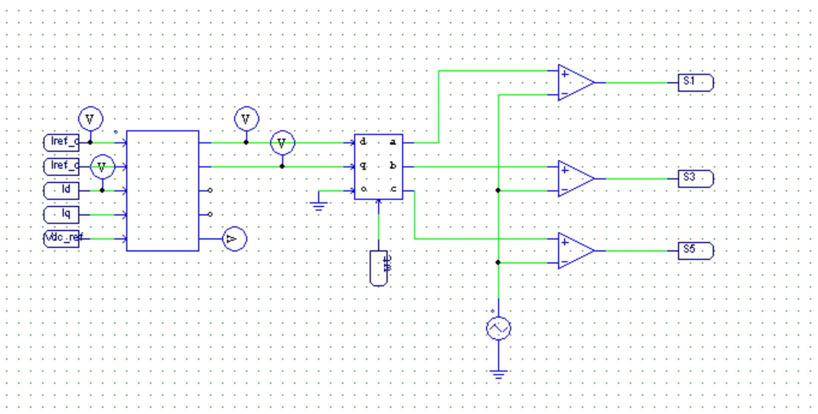
Fonte: Autor

Figura 23 – Conversão de tipos e definição da referência.



Fonte: Autor

Figura 24 – Saída do PI e definição do chaveamento.



Fonte: Autor

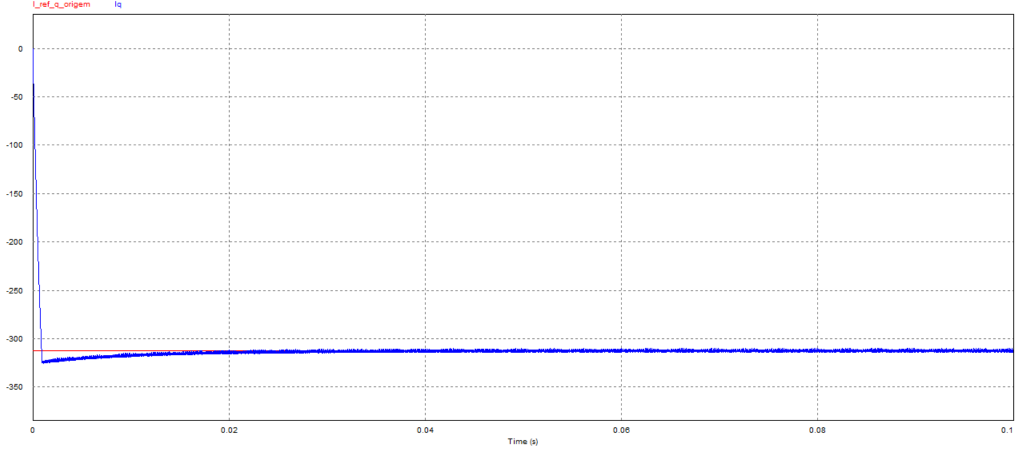
Na Figuras 25 e 26 está representado o resultado da simulação do circuito do conversor de dois níveis. Como referência foram escolhdas $i_{dRef} = 0A$ e $i_{qRef} = -312,5A$, correspondentes a uma injeção de potência de aproximadamente $84,2KVA$ na rede elétrica. Nas Figuras 25 e 26 são apresentadas as simulações das correntes i_d e i_q, respectivamente. Nota-se que o seguimento das referências com erro nulo é alcançado com um tempo de acomodação inferior ao estipulado na etapa de projeto.

O tempo de simulação foi de $100ms$, sendo este período equivalente a $5 \times T_{rede}$. O período de amostragem da simulação foi fixado em 10^{-5} segundos.

Figura 25 – Corrente id e a referência.

Fonte: Autor

Figura 26 – Corrente iq e a referência.

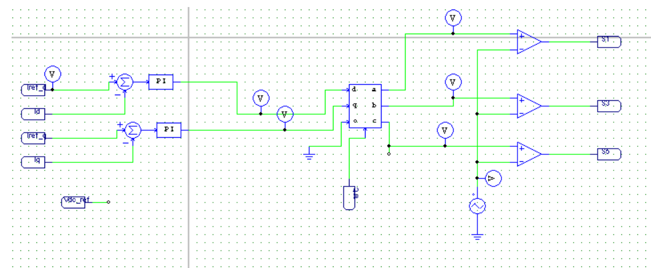


Fonte: Autor

O tempo de simulação necessário pelo PSIM para executar 0.1 segundos de simulação foi de 6.08 minutos. Este tempo pode ser diminuído abrindo-se mão de salvar os resultados da simulação para visualização durante a simulação. Esta funcionalidade está implementada em Python e basta comentar as linhas de código que implementam a visualização em tempo real.

O resultado obtido através do *PyBlock* foi comparado com o resultado obtido através do bloco de controle PI nativo do PSIM. Primeiramente, implementou-se o circuito da Figura 27. Pode-se notar que o *PyBlock* foi substituído por dois blocos PI que recebem o erro a cada passo de simulação.

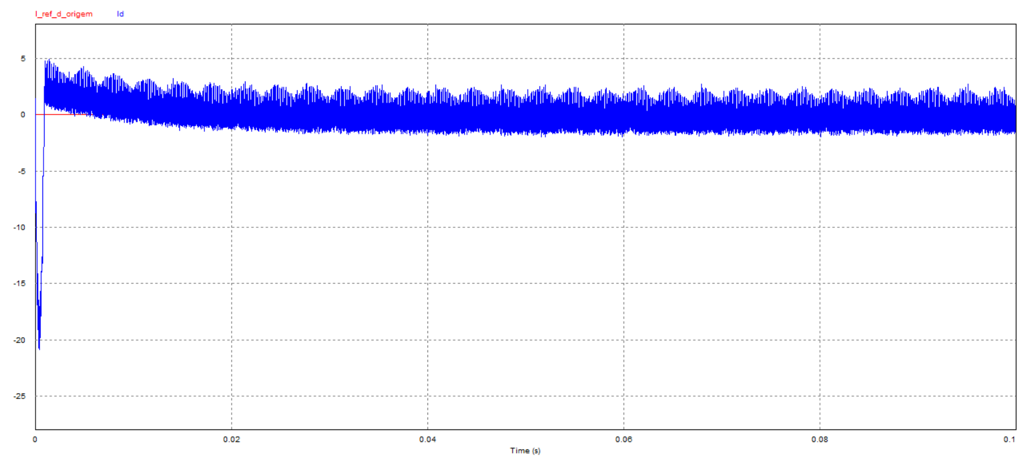
Figura 27 – Bloco PI nativo com circuito de controle.



Fonte: Autor

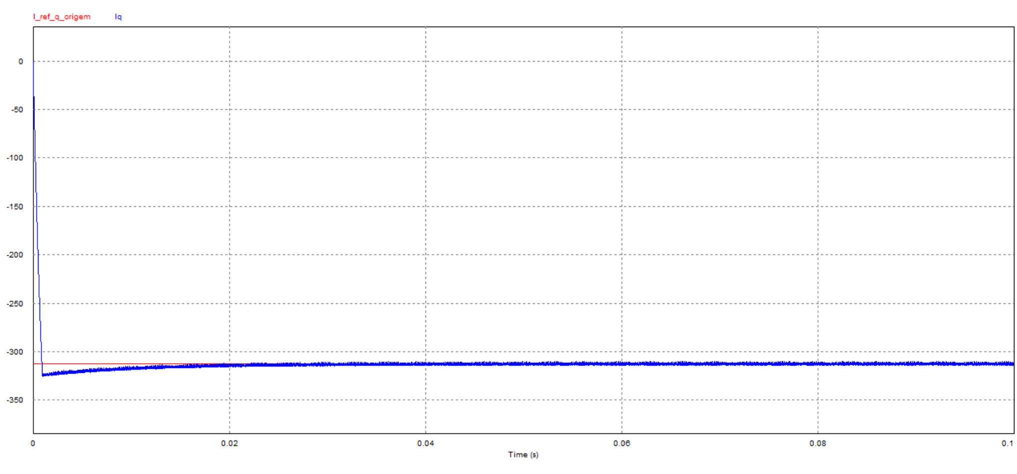
O resultado da simulação para as correntes i_d e i_q está na Figura 28 e 29.

Figura 28 – Resultado de I_d obtida com o bloco nativo PI.



Fonte: Autor

Figura 29 – Resultado de I_q obtida com o bloco nativo PI.



Fonte: Autor

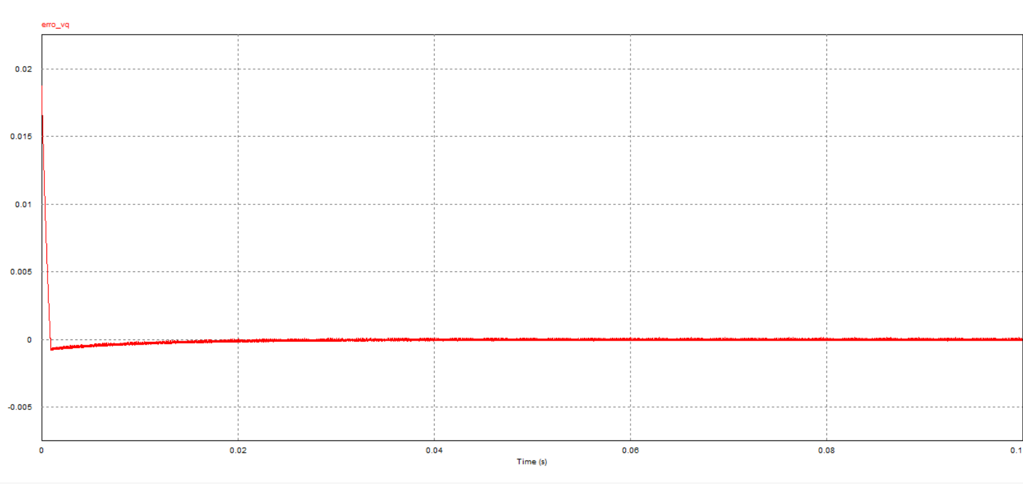
Pode-se inferir através destes gráficos que os resultados obtidos através do bloco original PI do PSIM e programando-se o controlador no *PyBlock* não acarretou em uma diferença significativa no comportamento do sistema em malha fechada.

Calculou-se a diferença entre as tensões de saída dos controladores, V_d e V_q, calculadas através do *PyBlock* com as mesmas tensões calculadas através do bloco nativo de controle PI. As Figuras 30, 31 mostram os resultados, os quais indicam que não existe uma diferença significativa entre ambas as tensões tanto no domínio d quanto em q. A pequena diferença entre ambas tensões é causada por erros de natureza numérica na casa de 10^{-5} volts.

Figura 30 – Diferença na tensão V_d.

Fonte: Autor

Figura 31 – Diferença na tensão V_q.



Fonte: Autor

3.4 CONCLUSÃO DO CAPÍTULO

Neste capítulo, o *PyBlock* foi utilizado para implementar um controlador proporcional-integral em um conversor trifásico. O resultado da simulação utilizando-se do bloco foi comparado ao resultado obtido utilizando-se o controlador implementado no bloco PI, que é nativo do PSIM. Os resultados mostraram que o *PyBlock* permite obter resultados consistentes com aqueles encontrados através do bloco nativo.

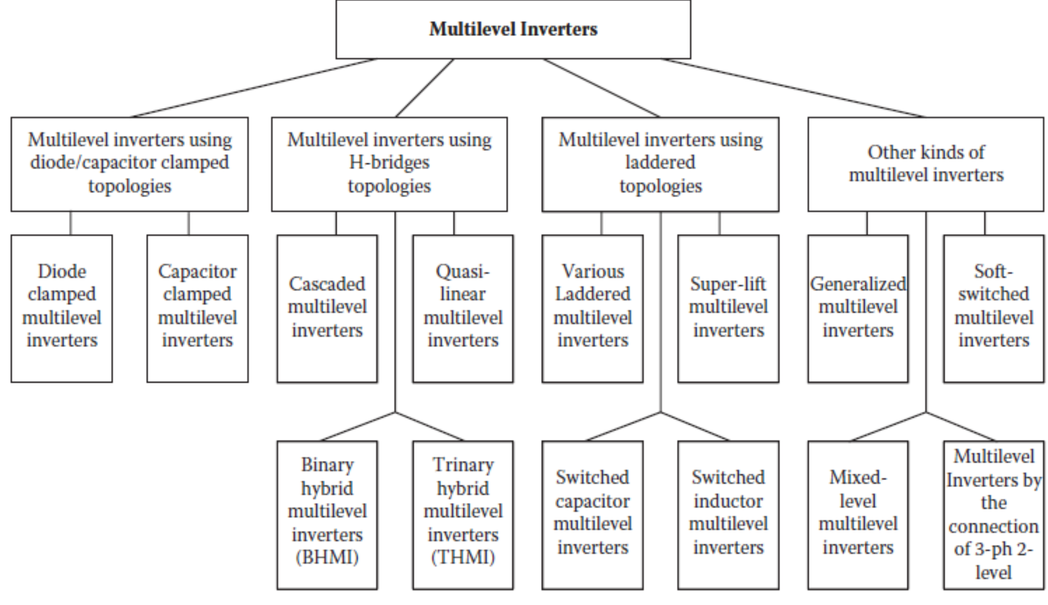
4 INVERSOR MULTINÍVEL

Neste capítulo foi apresentado o desenvolvimento de um controlador projetado com LMI's para o controle de um conversor trifásico multinível com neutro grampeado. Os resultados da simulação deste circuito foram obtidos através do *software* Matlab e também do *software* PSIM utilizando-se dos blocos *DLLBlock*, que é nativo do PSIM, e do *PyBlock*.

4.1 TOPOLOGIA DO INVERSOR

Existe uma grande quantidade de arquiteturas possíveis dentro da família de inversores multiníveis. A Figura 32 mostra as divisões internas dentro desta família

Figura 32 – Família dos inversores multiníveis.



Fonte: (YE, 2013)

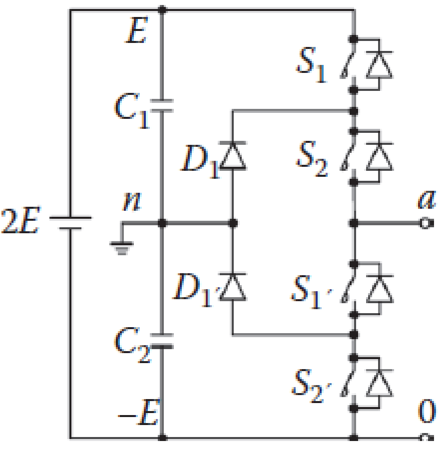
Os inversores multiníveis foram pensados para aplicações de média/alta potência (LIU; LUO, 2008), embora também sejam usados cada vez mais em sistemas de baixa potência, por exemplo, em sistemas fotovoltaicos, carros elétricos e outros. Seu uso se intensificou sobretudo devido à baixa distorção harmônica na tensão de saída (YE, 2013). O primeiro inversor multinível, que surgiu na década de 80, se chama NPC(*Neutral-point-clamped*). O inversor estudado neste trabalho possui 3 níveis de tensão possíveis por fase, também possui 3 pernas caracterizando então uma saída de tensão trifásica. Os conversores multiníveis necessitam do seguinte número de componentes, em função do número m de níveis

1. Chaves = 2(m-1)

2. Capacitores = (m-1)
3. Diodos = 2(m-2)
4. Tensão em cada capacitor = $\frac{V_{dc}}{m-1}$

O inversor a ser usado em cada perna do conversor trifásico está representado na Figura 33

Figura 33 – Conversor de 3 níveis.



Fonte: (YE, 2013)

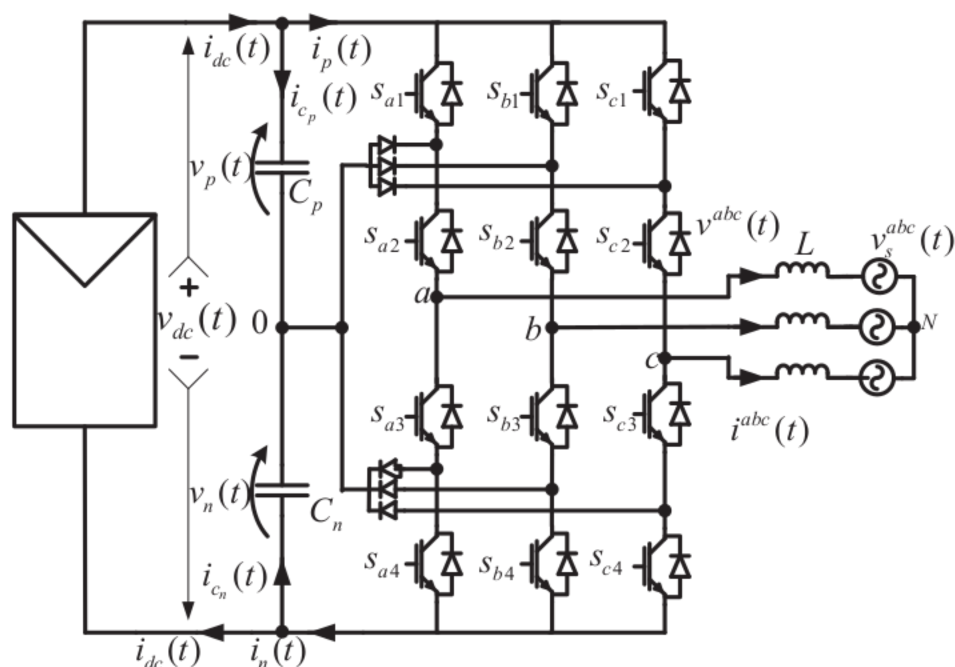
Escolheu-se o inversor multinível devido à sua necessidade de componentes ser menor em relação aos seus pares e devido a sua capacidade de trabalhar apenas com uma fonte CC (RODRIGUEZ; LAI; PENG, 2002) (NABAE; TAKAHASHI; AKAGI, 1981), além disso ele é amplamente utilizado em aplicações de energia fotovoltaica (SILVA *et al.*, 2016).

O circuito implementado neste trabalho foi considerado em (SILVA *et al.*, 2016), o objetivo era de implementar um algoritmo MPPT(*maximum power point tracking*) com tensão CC oriunda de painéis solares. Como colocado em (SILVA *et al.*, 2016), embora a topologia NPC tenha muitos méritos nestas aplicações fotovoltaicas, a flutuação do ponto neutro juntamente com altas tensões aplicadas às chaves deve ser abordada com estratégias de controle e também diminuir a distorção das tensões e correntes de saída. Soluções implementadas podem ser encontradas em (SEO; CHOI; HYUN, 2001) (YAMANAKA *et al.*, 2002) (MCGRATH; HOLMES; LIPO, 2003) (OGASAWARA; AKAGI, 1993) com enfase em *PWM space vector*.

4.2 PROJETO DO CONTROLADOR

Nesta seção, tem-se o objetivo de apresentar o projeto do controlador do inversor multinível com neutro grampeado.

Figura 34 – Circuito do inversor multinível.



Fonte: (SILVA *et al.*, 2016)

4.2.1 FORMULAÇÃO GERAL

Considere o inversor apresentado na Figura 34 onde cada chave S_{kn} é controlada através do *Gate* do transistor IGBT, sendo cada fase do conversor (a, b, c) acionada por 4 chaves. Estas chaves devem operar de maneira complementar duas a duas, ou seja, se S_{k1} está fechada então S_{k3} estará aberta, sendo válida esta relação de complementaridade entre S_{k2} e S_{k4}. Por conseguinte, apenas 3 configurações são possíveis para cada fase, definidas como segue

$$
U_k = k\frac{U_{dc}}{2}
\qquad
k = \begin{cases} 1 & \text{Se } [S_{k1},\ S_{k2}] = \text{ON e } [S_{k3},\ S_{k4}] = \text{OFF} \\ 0 & \text{Se } [S_{k2}, S_{k3}] = \text{ON e } [S_{k1},\ S_{k4}] = \text{OFF} \\ -1 & \text{Se } [S_{k3},\ S_{k4}] = \text{ON e } [S_{k1},\ S_{k2}] = \text{OFF} \end{cases}
\tag{4.1}
$$

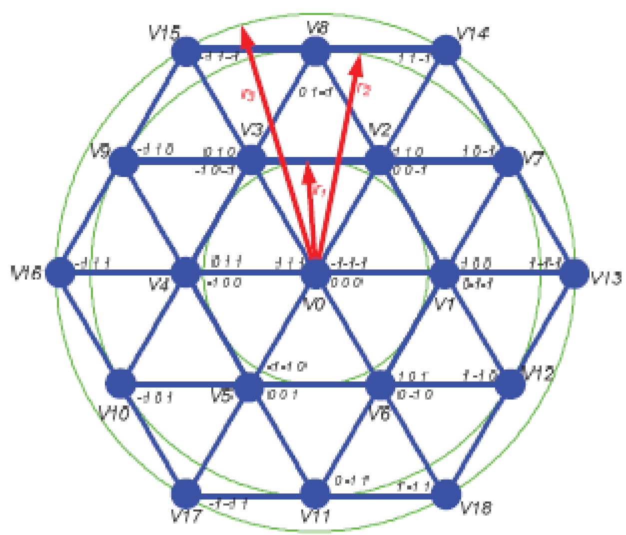
Na equação (4.1), apresentou-se os estados de chaveamento para cada perna do conversor multinível, então para se ter a tensão de saída condicionada à configuração das chaves deve-se combinar os estados de tensão de cada perna. Na Figura 35, tem-se os 27 estados de tensão possíveis, sendo que 19 combinações geram tensões diferentes de saída. Os valores das tensões de saída podem ser deduzidas através do diagrama de estados possíveis do conversor no domínio , que é uma transformação de domínio que converte

Figura 35 – Estados do inversor multinível.

v_j	ST	γ^a	γ^b	γ^c	$\|v_j^\alpha\|$	$\|v_j^\beta\|$
0	0	1	1	1	0	0
	1	0	0	0		
	2	-1	-1	-1		
1	3	1	0	0	1/2	0
	4	0	-1	-1		
2	5	1	1	0	1/4	$C_1/2$
	6	0	0	-1		
3	7	0	1	0	- 1/4	$C_1/2$
	8	-1	0	-1		
4	9	0	1	1	-0,5	0
	10	-1	0	0		
5	11	-1	-1	0	- 1/4	$- C_1/2$
	12	0	0	1		
6	13	1	0	1	1/4	$- C_1/2$
	14	0	-1	0		
7	15	1	0	-1	C_1	1/2
8	16	0	1	-1	0	1
9	17	-1	1	0	$- C_1$	1/2
10	18	-1	0	1	$- C_1$	- 1/2
11	19	0	-1	1	0	-1
12	20	1	-1	0	C_1	- 1/2
13	21	1	-1	-1	1	0
14	22	1	1	-1	1/2	C_1
15	23	-1	1	-1	- 1/2	C_1
16	24	-1	1	1	-1	0
17	25	-1	-1	1	- 1/2	$- C_1$
18	26	1	-1	1	1/2	$- C_1$

Fonte: (SILVA *et al.*, 2016)

Figura 36 – Diagrama de estados do conversor multinível.



Fonte: (SILVA *et al.*, 2016)

grandezas sinusoidais no plano de 3 dimensões para o plano de 2 dimensões, como mostra a Figura 36.

O conversor é conectado as fases *abc* através de um filtro indutivo de primeira ordem com um certo valor de resistência em série devido à não idealidade do indutor. O modelo matemático deste circuito é dado pela equação 4.2

$$L\frac{d}{dt}\mathbf{i}^{abc}(t) = \mathbf{v}^{abc} - R\mathbf{i}^{abc}(t) - \mathbf{v}_s^{abc}(t) \tag{4.2}$$

Através da transformação de *Park*, o modelo em *dq* é dado pela equação 4.3

$$L\frac{d}{dt}\mathbf{i}^{dq}(t) = \mathbf{v}^{dq} - R\mathbf{i}^{dq}(t) - \mathbf{v}_s^{dq}(t) \tag{4.3}$$

Ao contrário da abordagem por função de transferência utilizada no capítulo anterior, o sistema foi modelado utilizando a teoria de sistema chaveados apresentada em (DEZUO; LUNARDI; TROFINO, 2017).

Primeiramente, defini-se

$$x = \begin{matrix} i_q \\ i_d \end{matrix} \tag{4.4}$$

através da equação (4.3), defini-se a matriz A que multiplica x

$$A_{dq} = \begin{matrix} \frac{-R}{L} & 0 \\ 0 & \frac{-R}{L} \end{matrix} \tag{4.5}$$

e a partir da equação (4.3) e da Figura 35 segue que b_i são as colunas da matriz b_{dq} e $c_1 = \sqrt{\frac{2}{3}}$

$$B_{dq} = \begin{bmatrix} \frac{1}{L} & 0 \\ 0 & \frac{1}{L} \end{bmatrix}$$

$$b_{dq} = B_{dq} u_{dq} = B_{dq} V_{dc} \begin{bmatrix} 0 & 0 \\ 1/2 & 0 \\ 1/4 & c1/2 \\ -1/4 & c1/2 \\ -1/2 & 0 \\ -1/4 & -c1/2 \\ 1/4 & -c1/2 \\ c1 & 1/2 \\ 0 & 1 \\ -c1 & 1/2 \\ -c1 & -1/2 \\ 0 & -1 \\ c1 & -1/2 \\ 1 & 0 \\ 1/2 & c1 \\ -1/2 & c1 \\ -1 & 0 \\ -1/2 & -c1 \\ 1/2 & -c1 \end{bmatrix}^T \tag{4.6}$$

Tendo definido as matrizes de interesse do sistema, considere um sistema dinâmico não linear composto de m subsistemas representados na forma

$$\dot{x}(t) = Ax(t) + b_i, \quad i = 1, \cdots, m \tag{4.7}$$

O objetivo neste caso é projetar uma regra de chaveamento entre estes subsistemas(escolha de qual i está ativo em qual instante) que leva o sistema assimptoticamente ao estado de equilíbrio $\bar{x}$. Portanto tendo-se $\bar{x}$, segue que a dinâmica do erro entre os estados da planta e os estados desejados é dada por

$$\dot{e}(t) = Ae(t) + A\bar{x}(t) + b_i, \; i = 1, \cdots, m \tag{4.8}$$

onde $e = x - \bar{x}$.

Baseado em (FILIPPOV, 2013), assume-se que a dinâmica do sinal de erro pode ser representada como combinações convexas de subsistemas, logo o sistema chaveado global é representado por

$$\dot{e}(t) = \sum_{i=1}^{m} {}_i(e(t))(Ae(t) + A\bar{x} + b_i), \quad (e(t)) \tag{4.9}$$

com

$$:= \{ \quad R : \sum_{i=1}^{m} {}_i = 1, \quad {}_i \quad 0\} \tag{4.10}$$

Como o único parâmetro do modelo variante no tempo é b_i, de (4.8) e (4.9), segue que

$$A\bar{x} + \sum_{i=1}^{m} \bar{}_i b_i = 0 \tag{4.11}$$

Logo, deve-se primeiramente determinar qual a combinação linear dos parâmetros $\bar{}_i$ que leve ao ponto de equilíbrio $\bar{x}$. Além disso, o ponto de equilíbrio desejado deve ser uma combinação linear dos possíveis estados b_i.

Reescrevendo-se a dinâmica do erro como

$$\begin{aligned} \dot{e}(t) &= Ae + \quad , \quad (e(t)) \\ &= \sum_{i=1}^{m} {}_i(e(t)) \; {}_i = \sum_{i=1}^{m} {}_i(b_i - b^{-}) \\ b^{-} &= \sum_{i=1}^{m} \bar{}_i b_i \end{aligned} \tag{4.12}$$

Portanto, deve-se determinar a cada amostra qual o valor i deve ser utilizado tal que o erro converge para a origem. Com esse objetivo, utilizou-se uma versão adaptada do teorema apresentado em (DEZUO; LUNARDI; TROFINO, 2017).

Teorema 1: Assuma que $\bar{x}$ é um vetor constante representando o ponto de equilíbrio desejado para o sistema em malha fechada e que $x(t)$ está disponível para medição. Considere a dinâmica do erro definida em (4.7) e que o vetor $\bar{}$ de combinações lineares que leva ao equilíbrio é conhecido.

Suponha que existem matrizes $P = P^T > 0$, S, L_a satisfazendo as seguintes condições matriciais:

$$
\begin{aligned}
P &> 0 \\
+ L_a C_a + C_a^T L_a^T &< 0
\end{aligned}
\tag{4.13}
$$

$$
= \begin{matrix} A P + P A & \cdot \\ b P + S A + S & b S + S b \end{matrix}
\tag{4.14}
$$

$$
b = [b_1 \ldots b_m] \; , \; S = [S_1 \ldots S_m]
\tag{4.15}
$$

$$
C_a = [0_{(1 \times n)} \; \mathbf{0}_m \; 0] \; \mathbf{1}_m = [1 \ldots 1] \quad R^{1 \times m}
\tag{4.16}
$$

Então o sistema 4.7 é globalmente assintoticamente estável com lei de chaveamento

$$
\begin{aligned}
&:= argmax\{v_i(e)\}, v_i(e) = e^T P e + 2e^T (S_i - S^-) \\
S^- &= \sum_{i=1}^{m} {}^-_i S_i
\end{aligned}
\tag{4.17}
$$

e

$$
V(e) := \max_{i=1,\cdots,m}(v_i(e(t)))
\tag{4.18}
$$

é uma função de *Lyapunov* para o sistema chaveado.

4.2.2 IMPLEMENTAÇÃO DO CONTROLADOR

O primeiro passo para a implementação do método é determinar qual o vetor $^- = [\;_1 \;\; _2 \; \ldots \; _m]^T$ que resulta no ponto de equilíbrio $\bar{x}$. Para isso, deve-se resolver o seguinte problema de programação linear derivado de (4.11):

$$
\begin{aligned}
b_{dq} &= -A_{dq}\bar{x} \\
\sum_{i=1}^{m} {}^-_i &= 1
\end{aligned}
\tag{4.19}
$$

Este problema pode ser resolvido cada vez que se troca os parâmetros do circuito como L,R, V_{dc} ou quando se troca o valor do ponto de equilíbrio $\bar{x}$.

A solução de (4.19) é resolvida através da linguagem Python com base na biblioteca de otimização numérica *scipy*. O método numérico escolhido para esta otimização é o *IPM*, ou método dos pontos interiores, pois ele é utilizado em otimizações do tipo linear, quadrática e semi-definida positiva. Mais detalhes deste método podem ser encontrados em (SAVARD, 2001). O código a seguir apresenta o algoritmo para o cálculo da determinação de ¯ para valores de *xe* e *vdc* quaisquer escrito em linguagem Python.

```
from scipy import optimize
import numpy as np

def find_matrices(xe,vdc):
    L = 1*10**(-3)
    R = 0.8
    c1 = np.sqrt(2/3)
    u_albe = vdc*np.array([
    [0,1/2,1/4,-1/4,-1/2,-1/4,1/4,c1,0,-c1,-c1,0,c1,1,1/2,-1/2,-1,-1/2,1/2],
    [0,0,c1/2,c1/2,0,-c1/2,-c1/2,1/2,1,1/2,-1/2,-1,-1/2,0,c1,c1,0,-c1,-c1]
    ])

    u_dq = u_albe
    m = u_albe.shape[-1]
    A_dq = np.diag(np.array([-R/L,-R/L]))
    B_dq = np.diag(np.array([1/L,1/L]))
    b_dq = np.matmul(B_dq,u_dq)

    a_opt = np.concatenate((np.ones(shape=[1,m]),b_dq))
    b_opt = np.concatenate((np.array([[1]]),np.matmul(-A_dq,xe)),axis=0)
    coeffic_obj_func = np.ones(shape=[m])
    bound = ((0,1),)*m

    x = optimize.linprog(
    coeffic_obj_func,A_ub=None,b_ub=None,
    A_eq=a_opt,b_eq=b_opt,bounds=bound,
    method='interior-point',options={'maxiter': 200,'tol':1e-12,'disp': False}
    )
```

O vetor ¯ foi calculado primeiramente para os valores de $V_{dc} = 500V$ e as correntes de referência do circuito de $i_q = -312.5\,A$ e $i_d = 0\,A$. Estes valores resultaram no seguinte vetor

$$
= \begin{array}{c}
0.033784 \\
0.0164854 \\
0.0237447 \\
0.0487013 \\
0.0712657 \\
0.0487013 \\
0.0237447 \\
0.00900546 \\
0.033784 \\
0.116876 \\
0.116876 \\
0.033784 \\
0.00900546 \\
0.00477006 \\
0.0164854 \\
0.0712657 \\
0.233969 \\
0.0712657 \\
0.0164854
\end{array} \tag{4.20}
$$

Estes valores são recalculados automaticamente pelo código implementado sempre que V_{dc} ou X_e mudam no circuito.

O segundo passo para definir a função de chaveamento é o cálculo das matrizes através do problema de otimização com restrições LMI apresentado no Teorema 1. O cálculo das matrizes é realizado através da biblioteca *cvxpy* apresentada em (DIAMOND; BOYD, 2016) para a resolução do problema de otimização. Esta biblioteca oferece uma linguagem de modelagem matemática para problemas de otimização convexos. A *cvxpy* permite escolher um algoritmo *solver* para resolver o problema, optou-se pelo algoritmo apresentado em (O'DONOGHUE *et al.*, 2016) por ser amplamente utilizado em problemas de otimização deste tipo e receber amparo da comunidade cientifica na sua implementação. O código desenvolvido é apresentado a seguir

```
import cvxpy as cp

alfa = 10**7
epsilon = 10**-1
```

```
n,m = b_dq.shape

Ca = np.concatenate((np.zeros(shape=(1,n)),np.ones(shape=(1,m))),axis=1)
P = cp.Variable((n, n), PSD=True)
S = cp.Variable((n,m))
La = cp.Variable((int(n+m),1))
t = cp.Variable()

LMI = cp.bmat(
[[A_dq.T @ P + P @ A_dq.T , (b_dq.T @ P + S.T @ A_dq + S.T*alfa ).T],
[b_dq.T @ P + S.T @ A_dq + S.T*alfa , b_dq.T @ S + S.T @ b_dq]
])

cons1 = P >> 0
cons2 = LMI + La @ Ca + Ca.T @ La.T + epsilon * np.eye(n+m,n+m) <<
    t*np.eye((n+m),(n+m))

constraints = [cons1, cons2]

problem = cp.Problem(cp.Minimize(t), constraints)
solvers = ['SCS']
problem.solve(solver=solvers[0] )
```

Resolvendo-se este problema de otimização com o código acima, resulta nas seguintes matrizes

$$
P = \begin{matrix} 3.95241856e-08 & -6.56429153e-23 \\ -6.56429153e-23 & 3.95982011e-08 \end{matrix}
$$

$$
S^T = \begin{matrix} -5.38475e-21 & -2.60427e-20 \\ -9.89556e-10 & -2.60365e-20 \\ -4.94778e-10 & -8.09385e-10 \\ 4.94778e-10 & -8.09385e-10 \\ 9.89556e-10 & -2.60493e-20 \\ 4.94778e-10 & 8.09385e-10 \\ -4.94778e-10 & 8.09385e-10 \\ -1.61594e-09 & -9.9129e-10 \\ -5.37847e-21 & -1.98258e-09 \\ 1.61594e-09 & -9.9129e-10 \\ 1.61594e-09 & -9.9129e-10 \\ -5.39129e-21 & 1.98258e-09 \\ -1.61594e-09 & -9.9129e-10 \\ -1.97911e-09 & -2.60297e-20 \\ -9.89556e-10 & -1.61877e-09 \\ 9.89556e-10 & -1.61877e-09 \\ 1.97911e-09 & -2.60554e-20 \\ 9.89556e-10 & 1.61877e-09 \\ -9.89556e-10 & 1.61877e-09 \end{matrix} \quad (4.21)
$$

Logo, conhecidos as matrizes P e S e utilizando a relação (4.17), é possível calcular a cada amostra qual o estado i que leva o sistema ao ponto de equilíbrio $\bar{x}$.

4.3 RESULTADOS DE SIMULAÇÃO

Nesta seção serão apresentados os resultados do estudo do inversor multinível. Primeiramente, foi validado a aplicação de técnicas de sistemas chaveados no inversor através do *software* Matlab. Em seguida, o algoritmo em Python foi testado com o software PSIM. Por fim, realizaram-se testes com o código em C através do *DLLBlock*.

4.3.1 IMPLEMENTAÇÃO DO CONTROLADOR EM MATLAB

Para fins de comparação, o controlador projetado no capítulo anterior foi implementado no *software* Matlab e simulado através da integração numérica da equação diferencial que simula o comportamento do circuito. Logo, obteve-se o resultado da Figura 37 com

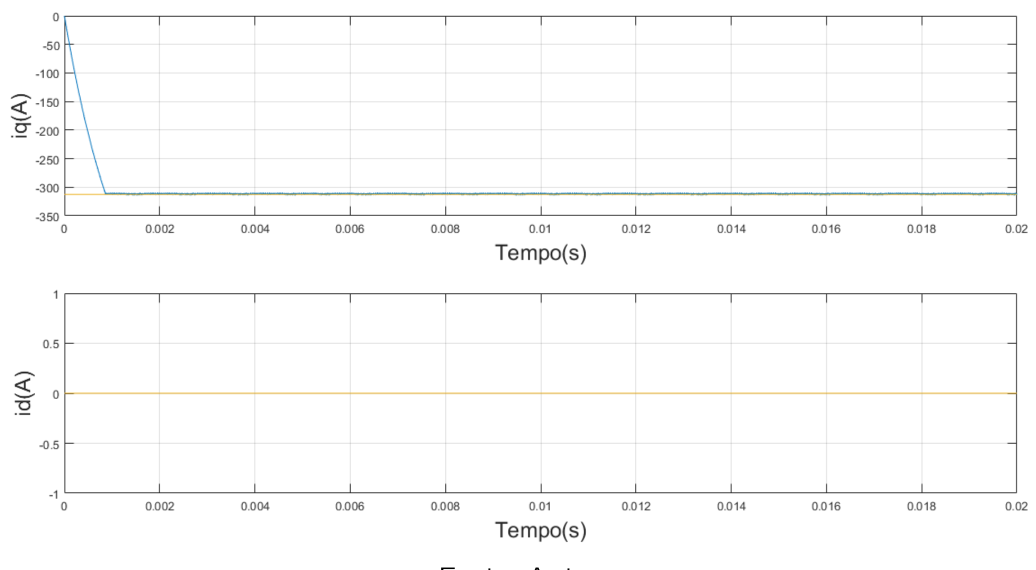
um período de amostragem de aproximadamente 1.8×10^{-6} segundos. A resolução da LMI foi realizada com as funções nativas do Matlab, resultando em matrizes P e S dadas por:

$$P = \begin{matrix} 1.63267604e-18 & -7.74171252e-26 \\ -7.74171252e-26 & 1.63267604e-18 \end{matrix}$$

$$S^T = \begin{matrix} -5.42289e-22 & 4.7266e-22 \\ -5.1020e-16 & 2.0103e-21 \\ -2.5510e-16 & -4.3016e-16 \\ 2.5510e-16 & -4.3016e-16 \\ 5.1020e-16 & -1.0650e-21 \\ 2.5510e-16 & 4.3016e-16 \\ -2.5510e-16 & 4.3016e-16 \\ -8.3315e-16 & -5.2683e-16 \\ -3.6181e-21 & -1.0537e-15 \\ 8.3315e-16 & -52683e-16 \\ 8.3315e-16 & 52683e-16 \\ 2.5323e-21 & 1.0537e-15 \\ -8.3315e-16 & 5.2683e-16 \\ -1.0204e-15 & 3.5480e-21 \\ -5.1020e-16 & -8.6031e-16 \\ 5.1020e-16 & -8.6031e-16 \\ 1.0204e-15 & -2.6027e-21 \\ 5.1020e-16 & 8.6031e-16 \\ -5.1020e-16 & 8.6032e-16 \end{matrix} \tag{4.22}$$

Note que estes valores são distintos dos obtidos via Python uma vez que os *solvers* LMI são diferentes.

Figura 37 – Simulação do controlador através de LMI's com Matlab.



Fonte: Autor

Nesta simulação a referência de corrente foi escolhida como em $iqe = \frac{-5V_{dv}}{8}$ e $ide = 0$. O estado da corrente convergiu para este valor em aproximadamente $0.87ms$. Portanto, tem-se o primeiro resultado da implementação do controlador projetado através de LMI's aplicado ao inversor multinível. Após este passo aplicou-se este controlador ao inversor descrito no PSIM com o *PyBlock* e os algoritmos em Python.

Para medir a capacidade do *software* Python no que tange seu desempenho em tempo utilizou-se do *software* Matlab como *benchmark*. O algoritmo de otimização implementado em ambas linguagens no controle de conversores multiníveis e apresentado nas seções acima é utilizado como comparativo de performance.

Utilizou-se da biblioteca *time*, disponível em código Python para calcular o tempo de cálculo do algoritmo e paralelamente utilizou-se do método *tic* *toc* no Matlab. O resultado em tempo para se calcular a otimização está disponível na Tabela 3.

Tabela 3 – Comparação do desempenho entre Python e Matlab

Linguagem	Tempo(s)
Matlab	3.063327
Python	0.007981

Fonte: Autor

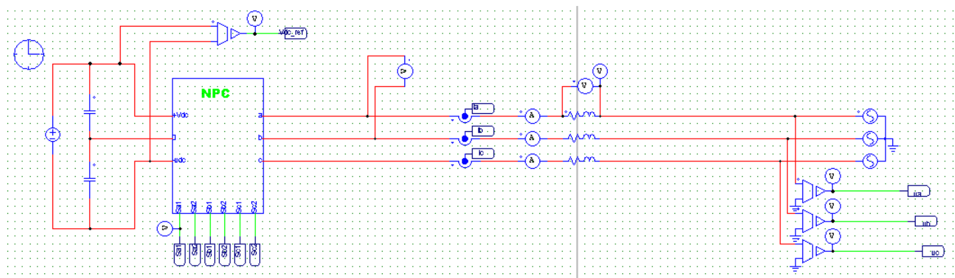
O algoritmo em Python teve uma performance $383,8\times$ mais rápida que o código em Matlab.

4.3.2 IMPLEMENTAÇÃO VIA *PyBlock*

Nesta seção, tem-se o objetivo de apresentar os resultados obtidos através do algoritmo em Python sendo executado pelo bloco do PSIM com o *PyBlock*.

O circuito implementado está representado na Figura 38.

Figura 38 – Conversor multinível.



Fonte: Autor

O bloco NPC representa o circuito do conversor multinível, como representado na Figura 39.

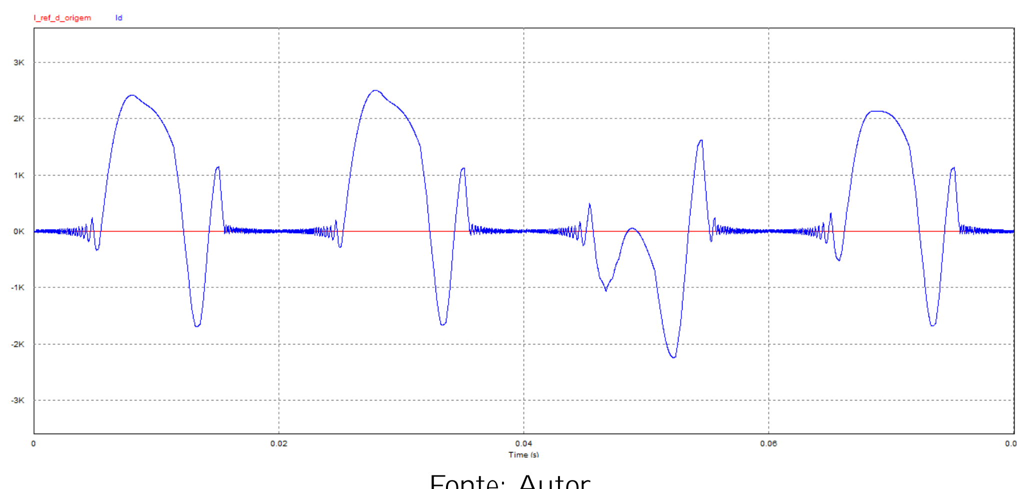
Figura 39 – Bloco NPC.

Fonte: Autor

O período necessário para um bom funcionamento do algoritmo de controle deve estar em torno de 10^{-6} segundos, definido de maneira empírica através do algoritmo em Matlab. Os primeiros testes de simulação indicaram problemas que não tinham sido encontrados no Matlab e também não encontrados nos testes de integração entre PSIM e *PyBlock*.

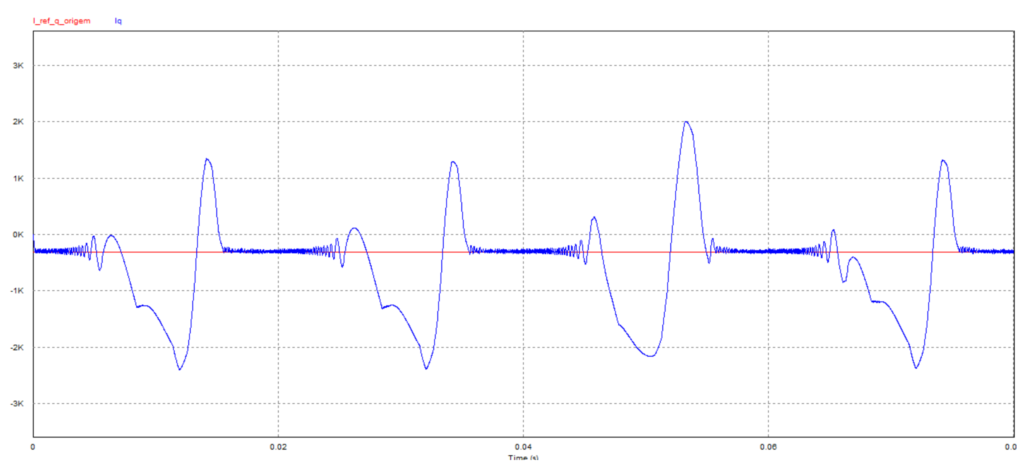
Simulou-se o circuito da Figura 38 e obteve-se os resultados da Figura 40 para I_d e da Figura 41 para I_q com período de amostragem de 10^{-6} segundos.

Figura 40 – *Id* do conversor multinível e a referência.



Fonte: Autor

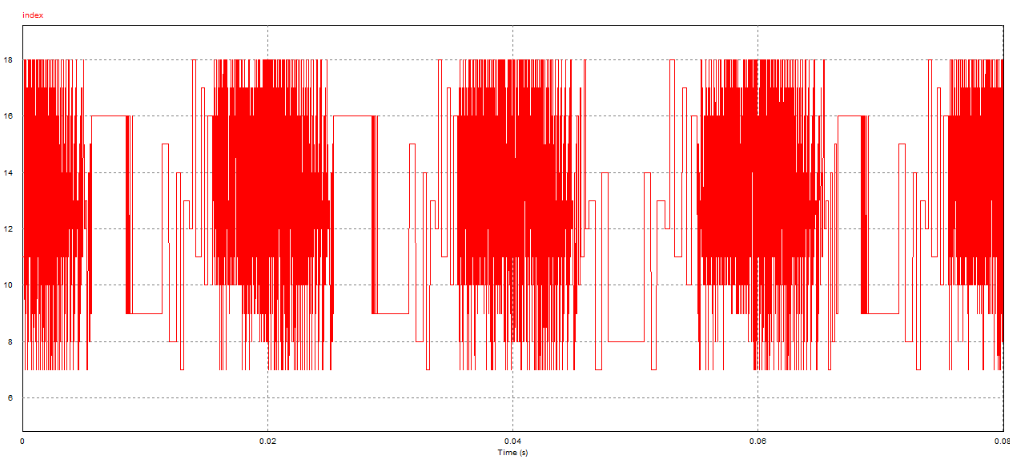
Figura 41 – *Iq* do conversor multinível e a referência.



Fonte: Autor

Pode-se constatar que as correntes seguem a referência durante um intervalo de tempo, no entanto quando a simulação atinge 25% de um ciclo da rede aproximadamente, as correntes divergem e acabam retornando, após pouco mais de um ciclo, de volta a referência. Com o intuito de se investigar este comportamento anômalo por parte do circuito plotou-se os estados do conversor que estavam sendo chaveados como mostra a Figura 42.

Figura 42 – Estados chaveados durante a simulação.

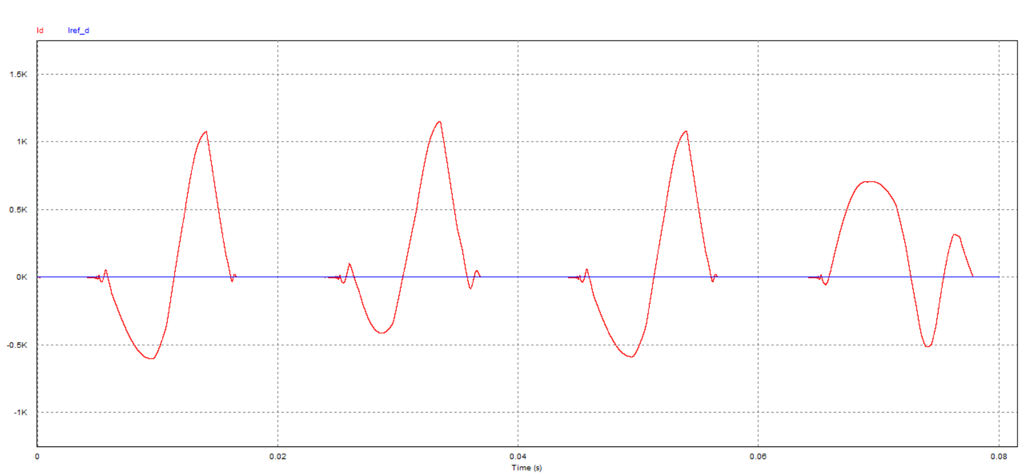


Fonte: Autor

Este gráfico expõe um problema de chaveamento por parte do PSIM. A frequência de chaveamento das chaves, ou seja, o período entre duas chamadas ao *PyBlock* está variando e aumentando no decorrer da simulação, logo o PSIM não consegue manter os 10^{-6} segundos de período de chaveamento de forma constante ao longo de toda simulação.

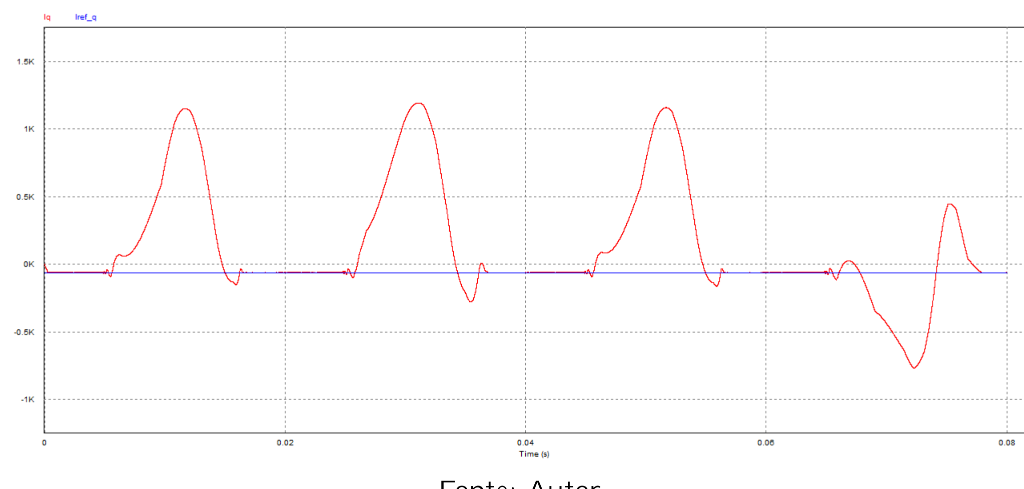
Pensou-se ser um problema do *PyBlock*, portanto implementou-se o controlador baseado em LMI através do bloco nativo do PSIM, o *DLLBlock* utilizando-se linguagem C com o vetor ¯ e as matrizes P e Q previamente calculadas. Nesse caso, não é necessário resolver o problema de otimização a cada amostra mas sim apenas calcular o chaveamento a partir da relação 4.17. Utilizou-se o mesmo período de amostragem no PSIM de 10^{-6} segundos.

Figura 43 – *Id* do conversor multinível e a referência com código em C.



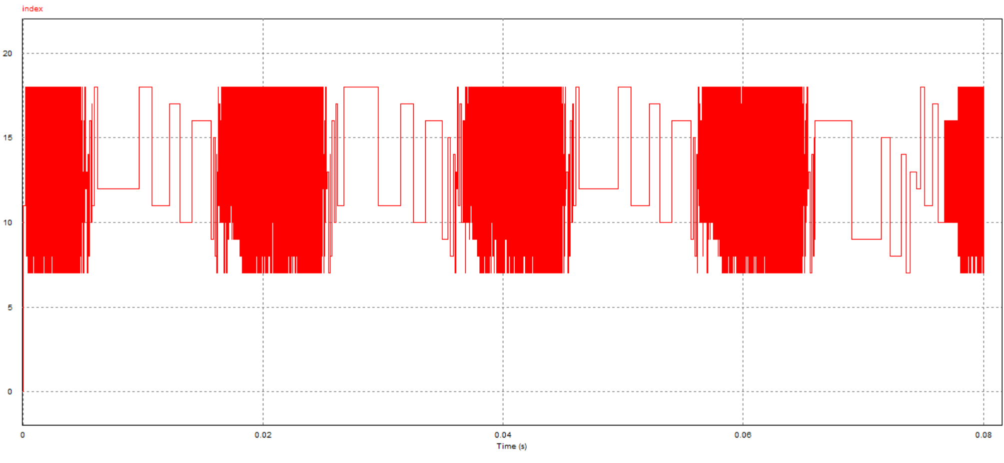
Fonte: Autor

Figura 44 – Iq do conversor multinível e a referência com código em C.



Fonte: Autor

Figura 45 – Estados chaveados durante a simulação com código em C.



Fonte: Autor

Através das Figuras 43, 44 e 45 pode-se constatar que o mesmo problema de chaveamento ocorre com o PSIM e seu bloco nativo para programação puramente em linguagem C, o *DLLBlock*. As correntes se acomodam na referência e após um determinado tempo ocorre um diminuição da frequência de chaveamento levando à instabilidade do sistema. É importante observar que enquanto o chaveamento entre os estados ocorre em altas frequências, a corrente é mantida no valor de equilíbrio. Por outro lado, quando as atualizações do sinal de corrente são menos frequentes, o sinal de corrente começa a divergir.

Para descartar hipóteses de erros decorrentes da solução do problema de otimização ligados aos valores de matrizes calculados através do *solver* em Python, foram utilizadas as matrizes P e S calculadas em Python na simulação numérica rodando em Matlab, conforme o código abaixo.

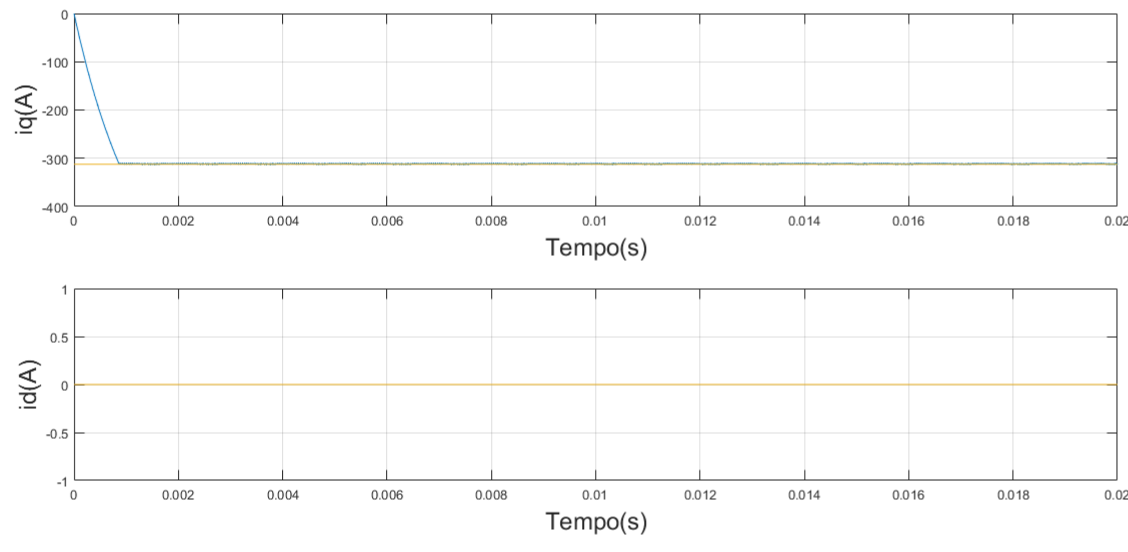
```
Popt = [ 1.63267604e-18 -7.74171252e-26;-7.74171252e-26 1.68593745e-18];
Sopt = [-5.42886741e-22,  -5.10199309e-16,  -2.55101181e-16,...
          2.55097585e-16,  5.10198223e-16,  2.55100096e-16,...
         -2.55098671e-16,  -8.33153176e-16,  -3.61806301e-21,...
          8.33149015e-16,  8.33152090e-16,  2.53228953e-21,...
         -8.33150101e-16,  -1.02039808e-15,  -5.10201820e-16,...
          5.10195713e-16,  1.02039699e-15,  5.10200734e-16,...
         -5.10196798e-16;  4.72663119e-22,  2.01033824e-21,  -4.30155295e-16,...
         -4.30156833e-16,  -1.06500588e-21,  4.30156240e-16,...
          4.30157778e-16,  -5.26829028e-16,  -1.05366354e-15,...
         -5.26834050e-16,  5.26829974e-16,  1.05366449e-15,...
          5.26834996e-16,  3.54800579e-21,  -8.60311059e-16,...
         -8.60314134e-16,  -2.60268120e-21,  8.60312004e-16,...
          8.60315080e-16];

S_thetabar = Sopt*theta_bar;
S_aux = Sopt-kron(ones(1,m),S_thetabar);

%% Simulation
toc
tspan=[0 0.02];
x0=[0;0];
opts = odeset('RelTol',1e-3,'AbsTol',1e-3);
[t,y]=ode45(@(t,X) dq_sys(t,X,A_dq,b_dq,Popt,S_aux,xe),tspan,x0,opts);
```

Logo, este caminho permite simular o circuito, atuar sobre ele através da função de chaveamento, medir a saída do circuito e acompanhar o desempenho do método de controle sem utilizar o PSIM. O resultado utilizando-se de um período de amostragem de aproximadamente 1.8×10^{-6} segundos está na Figura 46.

Figura 46 – Resultado de integração numérica com matrizes solucionadas no *solver* de Python.



Fonte: Autor

Obteve-se um resultado idêntico ao resultado obtido com as matrizes calculadas via *solver* do próprio Matlab. Em consequência, pode-se afirmar que o problema encontrado não é acarretado devido a problemas de otimização por parte do *solver* de Python quando este resolve o problema de LMI.

Com os testes realizados acima, foram analisadas as seguintes possibilidades: primeiramente, um possível causador de instabilidades poderia ser o *solver* de Python, mas os testes do Matlab mostraram que os valores oriundos da otimização são consistentes e resultam em resultados estáveis quando implementados em Matlab. Posteriormente, foi avaliada a possibilidade da instabilidade ser decorrente de um problema atrelado ao *PyBlock*, no entanto a mesma instabilidade foi detectada utilizando-se o *DLLBLock* nativo do PSIM. Não foi possível a utilização de períodos de amostragem maiores porque o controlador necessita de um período de amostragem na ordem de microssegundos.

4.4 CONCLUSÃO DO CAPÍTULO

Neste capítulo, o controlador foi implementado em Python e comparado ao Matlab, mostrando resultados consistentes. Posteriormente, o mesmo foi implementado no PSIM tanto com o *PyBlock* quanto com o *DLLBlock* e foi encontrada uma instabilidade na corrente, em uma faixa do período, que se reproduziu de maneira periódica indicando problemas de natureza de simulação.

5 CONCLUSÃO

Neste trabalho foi estudado o uso do software de simulação de circuitos de potência PSIM utilizando-se algoritmos de controle escritos em Python e a aplicação de controladores projetados através de LMI e LGR à conversores CC-CA. No bloco, que permite programação em linguagem Python, foi avaliada a possibilidade de se usar a abordagem de integração através de bibliotecas de vínculo dinâmico com o software PSIM. Posteriormente, implementou-se os dois métodos de controle em dois conversores CC-CA de topologias distintas. O primeiro um trifásico multinível com neutro grampeado utilizado em aplicações com painéis fotovoltaicos e o segundo um conversor trifásico de dois níveis utilizado em aplicações de menor potência. Projetou-se os controladores através de LMIs e LGR, respectivamente.

Os resultados demonstraram a efetividade da programação em Python, primeiramente devido ao tempo de simulação necessário comparado ao software Matlab. O novo bloco implementado neste trabalho também possibilita uma integração diretamente com o PSIM o que possibilitou um controle sobre as informações armazenadas durante a simulação. Além disso, a possibilidade de execução de códigos em python abre um leque de possibilidades de novas aplicações através do uso das bibliotecas desenvolvidas para essa linguagem de programação.

Os métodos de controle implementados para seguir o sinal de referência demonstraram bom desempenho em ambos os conversores. Através de LMI's chegou-se ao sinal de referência em 0,8ms com um período de chaveamento de 1µs e um período de rede de 20ms. Já no caso do conversor de dois níveis com controlador projetado por LGR, a frequência de chaveamento escolhida foi de 10µs e o sinal convergiu para a referência após 10ms de simulação.

Através deste trabalho também verificou-se a capacidade do *software* PSIM em simular circuitos com alta frequência de chaveamento e com otimizações numéricas implementadas em cada passo de simulação. No entanto, o *software* PSIM não apresentou bom desempenho em ambos os blocos, *DLLBlock* e *PyBlock*. Potenciais problemas numéricos na forma como o PSIM executa as chamadas do *DLLBlock* podem ser investigados com o objetivo de garantir a manutenção de um período de amostragem constante e suficientemente pequeno para a aplicação de técnicas de controle chaveado.

Sugere-se alguns possíveis trabalhos futuros envolvendo o bloco *PyBlock* que foi totalmente desenvolvido neste trabalho. Primeiramente, um possível melhoramento está nos vetores salvos à cada passo de simulação e que devem ser deletados da memória no próximo passo. Então, pode-se propor uma solução mais rápida salvando-se estes vetores na DLL.

Também sugere-se melhorar o modo como está sendo realizado o carregamento da DLL e a inicialização do interpretador de Python, pois neste trabalho montou-se um arquivo de inicialização que deve ser executado antes da simulação pelo usuário. Logo, este arquivo de carregamento poderia de alguma forma ser executado no início de cada simulação.

Por fim, sugere-se uma melhoria na finalização do interpretador, devido ao problema encontrado na biblioteca *Numpy*, relatado neste trabalho e ainda não solucionado pelos desenvolvedores da biblioteca. O interpretador não pode ser finalizado à cada chamada da DLL, portanto finalizar o interpretador de maneira eficiente e independente do usuário tornaria mais simples a utilização do *PyBlock* no PSIM.

REFERÊNCIAS BIBLIOGRÁFICAS

BAZANELLA, A. S.; JUNIOR, J. M. G. da S. **Sistemas de controle: Princípios e métodos de projeto**. [S.l.]: UFRGS, 2005.

CHIOZO, M. F. **Notas de aula em Ligacao Dinamica**. [S.l.]: UNICAMP, 2007.

DEAECTO, G. S. *et al.* Switched a ne systems control design with application to dc–dc converters. **IET control theory & applications**, IET, v. 4, n. 7, p. 1201–1210, 2010.

DEZUO, T.; LUNARDI, H.; TROFINO, A. Robust switching rule design for photovoltaic systems under non-uniform conditions. In: IEEE. **2017 IEEE 56th Annual Conference on Decision and Control (CDC)**. [S.l.], 2017. p. 2342–2347.

DIAMOND, S.; BOYD, S. CVXPY: A Python-embedded modeling language for convex optimization. **Journal of Machine Learning Research**, v. 17, n. 83, p. 1–5, 2016.

FILIPPOV, A. F. **Differential equations with discontinuous righthand sides: control systems**. [S.l.]: Springer Science & Business Media, 2013. v. 18.

GOLDEMBERG, C. *et al.* A python based power electronics e-learning tool. In: IEEE. **2009 Brazilian Power Electronics Conference**. [S.l.], 2009. p. 1088–1092.

IBM. **Embed Python scripting in C applications**. [S.l.], 2010. Rev. 1.

KHORIKOV, V. **Unit Testing Principles, Practices, and Patterns**. 1. ed. [S.l.]: Manning Publications, 2019. ISBN 1617296279, 978-1617296277.

KRAUSE, P. C. *et al.* **Analysis of electric machinery and drive systems**. [S.l.]: Wiley Online Library, 2002. v. 2.

LIU, Y.; LUO, F. L. Trinary hybrid 81-level multilevel inverter for motor drive with zero common-mode voltage. **IEEE Transactions on Industrial Electronics**, IEEE, v. 55, n. 3, p. 1014–1021, 2008.

MCGRATH, B. P.; HOLMES, D. G.; LIPO, T. Optimized space vector switching sequences for multilevel inverters. **IEEE Transactions on power electronics**, v. 18, n. 6, p. 1293–1301, 2003.

NABAE, A.; TAKAHASHI, I.; AKAGI, H. A new neutral-point-clamped pwm inverter. **IEEE Transactions on industry applications**, IEEE, n. 5, p. 518–523, 1981.

NUMPY, Issues reports. <https://github.com/numpy/numpy/issues/8097>. Acessado: 29-08-2020.

O'DONOGHUE, B. *et al.* Conic optimization via operator splitting and homogeneous self-dual embedding. **Journal of Optimization Theory and Applications**, v. 169, n. 3, p. 1042–1068, June 2016. Disponível em: <http://stanford.edu/~boyd/papers/scs.html>.

OGASAWARA, S.; AKAGI, H. Analysis of variation of neutral point potential in neutral-point-clamped voltage source pwm inverters. In: IEEE. **Conference Record of the 1993 IEEE Industry Applications Conference Twenty-Eighth IAS Annual Meeting**. [S.l.], 1993. p. 965–970.

POWERSYS. **How to use de DLL Block**. [S.l.], 2004. Rev. 1.

RASHID, M. H. **Power electronics handbook**. [S.l.]: Butterworth-Heinemann, 2017.

ROBINSON, D. **The Incredible Growth of Python.** 2017. <https://stackoverflow.blog/2017/09/06/incredible-growth-python/>. Acessado: 05-10-2019.

RODRIGUEZ, J.; LAI, J.-S.; PENG, F. Z. Multilevel inverters: a survey of topologies, controls, and applications. **IEEE Transactions on industrial electronics**, IEEE, v. 49, n. 4, p. 724–738, 2002.

SAVARD, G. **Introduction aux m´ethodes de points int´erieurs**. [S.l.]: Ecole Polytechnique de Montreal, 2001.

SEO, J. H.; CHOI, C. H.; HYUN, D. S. A new simplified space-vector pwm method for three-level inverters. **IEEE Transactions on power electronics**, IEEE, v. 16, n. 4, p. 545–550, 2001.

SILVA, J. *et al.* Extended-horizon finite-control-set predictive control of a multilevel inverter for grid-tie photovoltaic. In: IEEE. **2016 IEEE Energy Conversion Congress and Exposition (ECCE)**. [S.l.], 2016. p. 1–6.

SILVA, J. F.; RODRIGUES, N.; COSTA, J. Space vector alpha-beta sliding mode current controllers for three-phase multilevel inverters. In: IEEE. **2000 IEEE 31st Annual Power Electronics Specialists Conference. Conference Proceedings (Cat. No. 00CH37018)**. [S.l.], 2000. v. 1, p. 133–138.

TAHIR, S. *et al.* Digital control techniques based on voltage source inverters in renewable energy applications: A review. **Electronics**, Multidisciplinary Digital Publishing Institute, v. 7, n. 2, p. 18, 2018.

THULASIRAMAN, K.; SWAMY, M. N. **Graphs: theory and algorithms**. [S.l.]: John Wiley & Sons, 2011.

YAMANAKA, K. *et al.* A novel neutral point potential stabilization technique using the information of output current polarities and voltage vector. **IEEE Transactions on industry applications**, IEEE, v. 38, n. 6, p. 1572–1580, 2002.

YE, F. L. L. H. **Advanced DC/AC inverters : applications in renewable energy**. [S.l.]: Taylor Francis, 2013. (Power electronics, electrical engineering, energy, and nanotechnology). ISBN 9781466511385,1466511389.

APÊNDICE A

Figura 47 – Exemplo de *logs* amostrados em tempo real e disponibilizados ao usuário.

```
ok

Simulando mais uma vez:
[array([[   0.    ],
       [-125.0625]]), array([[   0. ],
       [-312.5]]), array([[0],
       [0]])]ok

Simulando mais uma vez:
[array([[  -0.67301634],
       [-123.32873108]]), array([[  -1.6817],
       [-307.8554]]), array([[ 0.    ],
       [-0.0625]])]ok

Simulando mais uma vez:
[array([[  -0.32379416],
       [-121.60414984]]), array([[  -0.8074],
       [-303.2384]]), array([[-0.00033634],
       [-0.18657108]])]ok

Simulando mais uma vez:
[array([[   0.3126411 ],
       [-119.88476532]]), array([[   0.7837],
       [-298.639 ]]), array([[-0.00083416],
       [-0.30878984]])]ok

Simulando mais uma vez:
[array([[   0.93594592],
       [-118.1704645 ]]), array([[   2.3404],
       [-294.0569]]), array([[-0.0008389 ],
       [-0.42916532]])]ok
```

Fonte: Autor

APÊNDICE B

CÓDIGO COMPLETO DA RESOLUÇÃO DE LMI's

```
import numpy as np
from scipy import optimize
import cvxpy as cp

def find_matrices(xe,vdc):

    L = 1*10**(-3)
    R = 0.8
    c1 = np.sqrt(2/3)

    u_albe = vdc*np.array([
    [0,1/2,1/4,-1/4,-1/2,-1/4,1/4,c1,0,-c1,-c1,0,c1,1,1/2,-1/2,-1,-1/2,1/2],
    [0,0,c1/2,c1/2,0,-c1/2,-c1/2,1/2,1,1/2,-1/2,-1,-1/2,0,c1,c1,0,-c1,-c1]])

    u_dq = u_albe
    m = u_albe.shape[-1]
    A_dq = np.diag(np.array([-R/L,-R/L]))
    B_dq = np.diag(np.array([1/L,1/L]))
    b_dq = np.matmul(B_dq,u_dq)

    # finding a linear combination theta_bar of input signals that results in
        the
    # desired equilibrium point
    a_opt = np.concatenate((np.ones(shape=[1,m]),b_dq))
    b_opt = np.concatenate((np.array([[1]]),np.matmul(-A_dq,xe)),axis=0)

    # arguments to linear optimization
    coeffic_obj_func = np.ones(shape=[m])
    bound = ((0,1),)*m

    try:
        x = optimize.linprog(coeffic_obj_func,
        A_ub=None,b_ub=None,A_eq=a_opt,b_eq=b_opt,
        bounds=bound,method='interior-point',options={'maxiter':
            200,'tol':1e-12,'disp': False})
    except:
        print('Optimization Failed!\n ')
        return 0,0,0
    theta_bar = x.x[...,np.newaxis]
```

```
#%% LMI feasability problem
alfa = 10**7
epsilon = 10**-1
n,m = b_dq.shape

Ca = np.concatenate((np.zeros(shape=(1,n)),np.ones(shape=(1,m))),axis=1)

P = cp.Variable((n, n), PSD=True)
S = cp.Variable((n,m))
La = cp.Variable((int(n+m),1))
t = cp.Variable()

LMI = cp.bmat([[A_dq.T @ P + P @ A_dq.T , (b_dq.T @ P + S.T @ A_dq +
    S.T*alfa ).T],
              [b_dq.T @ P + S.T @ A_dq + S.T*alfa , b_dq.T @ S + S.T @
                  b_dq]])

cons1 = P >> 0
cons2 = LMI + La @ Ca + Ca.T @ La.T + epsilon * np.eye(n+m,n+m) <<
    t*np.eye((n+m),(n+m))
constraints = [cons1, cons2]

problem = cp.Problem(cp.Minimize(t), constraints)
solvers = ['SCS']
for solver_i in solvers:
    try:
        problem.solve(solver=solver_i )#, verbose=True)
    except:
        print("ERROR Otimization PYTHON")
        print(problem.status)
        return 0,0,0

return P.value,S.value,theta_bar
```