UNIVERSIDADE ESTADUAL PAULISTA
"JÚLIO DE MESQUITA FILHO"
Campus de Ilha Solteira

ISABELLA CRISTINA BOARO FERNANDES

DESENVOLVIMENTO E IMPLEMENTAÇÃO DE LINGUAGEM DE PROGRAMAÇÃO EM PYTHON PARA PARAMETRIZAR E MONITORAR UM CONTROLE DE VELOCIDADE COM CICLOCONVERSOR DE FREQUÊNCIA

ILHA SOLTEIRA SP
SETEMBRO DE 2021

UNIVERSIDADE ESTADUAL PAULISTA
"JÚLIO DE MESQUITA FILHO"
Campus de Ilha Solteira

ISABELLA CRISTINA BOARO FERNANDES

DESENVOLVIMENTO E IMPLEMENTAÇÃO DE LINGUAGEM DE PROGRAMAÇÃO EM PYTHON PARA PARAMETRIZAR E MONITORAR UM CONTROLE DE VELOCIDADE COM CICLOCONVERSOR DE FEQUÊNCIA

Trabalho de Graduação apresentado à Faculdade de Engenharia do Campus de Ilha Solteira – UNESP como parte dos requisitos para obtenção do grau de Engenheira Eletricista.

Prof. Dr. Rodrigo Cardim
Orientador

ILHA SOLTEIRA SP
SETEMBRO DE 2021

FICHA CATALOGRÁFICA
Desenvolvido pelo Serviço Técnico de Biblioteca e Documentação

F363d

Fernandes, Isabella Cristina Boaro.
Desenvolvimento e implementação de linguagem de programação em Python para parametrizar e monitorar um controle de velocidade com cicloconversor de frequência / Isabella Cristina Boaro Fernandes. -- Ilha Solteira: [s.n.], 2021
96 f. : il.

Trabalho de conclusão de curso (Graduação em Engenharia Elétrica) - Universidade Estadual Paulista. Faculdade de Engenharia de Ilha Solteira, 2021

Orientador: Rodrigo Cardim
Inclui bibliografia

1. Cicloconversor de frequência. 2. Python. 3. Comissionamento.

Raiane da Silva Santos
Supervisora Técnica de Seção
Seção Técnica de Referência, Atendimento ao usuário e Documentação
Diretoria Técnica de Biblioteca e Documentação
CRB/8 - 9999

UNIVERSIDADE ESTADUAL PAULISTA
"JÚLIO DE MESQUITA FILHO"
Câmpus de Ilha Solteira

ATA DE DEFESA DE TRABALHO DE GRADUAÇÃO

Aos vinte e um dias do mês de setembro do ano de dois mil e vinte e um, a discente ***Isabella Cristina Boaro Fernandes*** matriculada sob o nº 142054526, tendo como banca examinadora o seu orientador, o *Prof. Dr. Rodrigo Cardim*, *o Prof. Dr. Carlos Antonio Alves* e o *Dr. Adalberto Zanatta Neder Lazarini*, apresentou o Trabalho de Graduação intitulado **"Desenvolvimento e Implementação de Linguagem de Programação em Python para Parametrizar e Monitorar um Controle de Velocidade com Cicloconversor de Frequência"** obtendo a nota 9,5 (nove e meio) e conceito Aprovado .

Prof. Dr. Rodrigo Cardim
- orientador -

Fernandes Isabella Cristina Boaro
Assinado digitalmente por Fernandes Isabella Cristina Boaro
DN: cn=Fernandes Isabella Cristina Boaro, o=Siemens, email=isabella.fernandes@siemens.com
Data: 2021.11.09 11:21:40 -03'00'

Isabella Cristina Boaro Fernandes
- discente -

Prof. Dr. Carlos Antonio Alves
- Membro da Banca -

Dr. Adalberto Zanatta Neder Lazarini
- Membro da Banca -

Faculdade de Engenharia – Departamento de Engenharia Elétrica
Avenida Professor José Carlos Rossi, 1370 - CEP 15.385-000, Ilha Solteira - SP - Brasil
Tel (18) 37431150 - Fax (18) 3742-2735 - www.feis.unesp.br

Dedico à Deus, pela misericórdia e por ser essencial em minha vida, à minha família e meu namorado, por sempre me apoiarem e acreditarem em mim.

AGRADECIMENTOS

À Deus pela saúde e pelos caminhos que me fez trilhar durante todo curso de Engenharia Elétrica na Universidade Estadual Paulista – Julio de Mesquita Filho. Aos meus pais, Geraldo Fernandes Filho e Maria Christina Martines Boaro Fernandes, por todo o apoio, pelos esforços imensuráveis e amor transferidos a mim durante toda a minha vida e principalmente durante a minha graduação. Á minha madrinha, Rosangela Luzia Martines Boaro Belucci pelo incentivo, amor e esforços dedicados a mim nos momentos difíceis e por sempre acreditar em mim. Ao meu namorado Eduardo Ribeiro Silva Abrahão pelo constante apoio, companheirismo, amizade e amor.

A todos meus professores da UNESP - Universidade Estadual Paulista "Júlio de Mesquita Filho" – Ilha Solteira, pelos ensinamentos, conselhos nas aulas ministradas, que contribuíram com a minha formação. Em especial, agradeço ao Prof. Dr. Rodrigo Cardim, pelo tempo, apoio e atenção nas orientações concedidas durante toda a graduação e em especial para este trabalho.

As amizades que construí durante toda a graduação, agradeço a Elisângela Moya Viani, Gabriela de Castro Araújo, Felipe de Carvalho Forti, José Victor Boaretto Guissoni, Jorge Luis Peredo Junior, Fernando Hiratsuka Estavare, Renan Suman, Luiz Pedro Miasato Nogueira de Assis, Igor Holtz, Jefferson de Almeida Carvalho, Bruno Gomes Aranha, Vinicius de Almeida Longo, Marcelo Henrique Carnio e Marcos Henrique do Nascimento Sousa pela amizade, conselhos, estudos e companheirismo durante todos esses anos.

A André Braga e Eduardo Golfeto pela amizade e principalmente pela ajuda por terem possibilitado a realização dos testes finais da simulação desse projeto.

Agradeço as amizades que construí na Siemens Infraestrutura e Industria Ltda. pelos ensinamentos durante todo o período de estágio e por terem possibilitado o contato com os equipamentos de bancada, com os manuais e catálogos.

LISTA DE FIGURAS

Figura 1 – Motor de Indução Trifásico 14
Figura 2 – Diagrama simplificado dos principais blocos do Inversor de Frequência 17
Figura 3 – Aplicação de uma fonte de tensão trifásica em uma ponte retificadora 17
Figura 4 – Topologia do conversor de frequência AC/AC 18
Figura 5 – Diagrama esquemático do inversor de frequência 19
Figura 6 – Princípio da geração do PWM 21
Figura 7 – Formas de onda de tensão e de corrente na saída do inversor 21
Figura 8 – Protocolo S7 no modelo de referência ISSO-OSI 28
Figura 9 – Encapsulamento de Protocolos 29
Figura 10 – Topologia do Sistema que será simulado 36
Figura 11 – Kit de bancada utilizado para testes 36
Figura 12 – Tela intitulada Menu 38
Figura 13 – Tela intitulada Conexão 43
Figura 14 – Tela intitulada Comissionamento 51
Figura 15 – Tela para a criação dos gráficos 63
Figura 16 – Tela de monitoramento 66
Figura 17 – Simulação conexão da Interface com o SINAMICS G120 68
Figura 18 – Simulação da realização do Reset de fábrica 69
Figura 19 – Simulação da realização do Comissionamento 70
Figura 20 – Simulação da realização do Monitoramento V1 71
Figura 21 – Simulação da realização do Monitoramento V2 72
Figura 22 – Simulação da realização do Monitoramento V3 73
Figura 23 – Simulação da realização do Monitoramento da Falha ativa 74
Figura 24 – Análise gráfica I 75
Figura 25 – Análise gráfica II Parada OFF1 76
Figura 26 – Análise gráfica III Parada OFF3 77

LISTA DE TABELAS

Tabela 1 – Combinação para acionamento dos IGBTs de um Conversor de Frequência trifásico 19

Tabela 2 – Tensões aplicadas no motor 20

Tabela 3 – Data Type: Tipos de dados que os parâmetros podem assumir 23

Tabela 4 - Estrutura de caracteres 34

RESUMO

Dada a importância do controle de velocidade visto que é uma parte essencial de muitas tarefas de automação, se faz necessário manter a confiabilidade e operabilidade dos equipamentos que compõe estes sistemas. Este trabalho propõe o desenvolvimento de uma interface gráfica utilizando a linguagem de programação Python para a realização do comissionamento de um cicloconvesor, bem como implementação e monitoramento gráfico de algumas variáveis de operação de um controle de velocidade "autônomo" com um cicloconversor de frequência da família SINAMICS G120 da marca SIEMENS, o foco principal é a aplicação de ventilação.

Palavras–chave: Cicloconversor de Frequência, Python, Controle de Velocidade, Comissionamento, Monitoramento de Variáveis.

ABSTRACT

Given the relevance of speed control, as it is an essential part of many automation tasks, it is necessary to maintain the reliability and operability that make up these systems. This work proposes the development of a graphical interface using a Python programing language to carry out the commissioning of a cycloconverter as well as the implementation and graphical monitoring of some operating variable of a simple “autonomous” speed control with a frequency cycloconverter from the family SINAMICS G120 of the SIEMENS brand, the main focus is the application of ”ventilation”.

Keywords: Frequency Cycloconverter, Python, Speed Control, Commissioning, Variable Monitoring.

SUMÁRIO

1.INTRODUÇÃO **10**

2.CONVERSORES DE FREQUÊNCIA **13**

2.1.MOTOR DE INDUÇÃO 14

2.1.1.Constituição do Motor Elétrico de Indução **14**

2.1.2.Princípio de funcionamento do Motor de Indução **15**

2.2.CONVERSORES DE FREQUÊNCIA 16

2.2.1.Controle de chaveamento dos tiristores do Inversor de frequência **18**

3.SINAMICS G120 **22**

3.1.CONTROLE DE VELOCIDADE 24

3.2.PROTOCOLO DE COMUNICAÇÃO S7 26

4.PYTHON **30**

4.1.BIBLIOTECAS 32

4.1.1.Biblioteca Python-Snap7 **32**

4.1.2.Biblioteca Python-Sinamics **33**

4.1.3.Biblioteca Matplotlib **34**

4.1.4.Biblioteca kivy **35**

4.2.DESENVOLVIMENTO DA INTERFACE PARA COMISSIONAMENTO E MONITORAMENTO 35

4.2.1.Classe Menu 38

4.2.2.Classe Conectador **40**

4.2.3.Classe Comissionamento **43**

4.2.4.Classe Drives 51

4.2.5.Classe Gerenciador e Classe G120 **66**

5.RESULTADOS E DISCUSSÃO **68**

6.CONCLUSÃO **78**

REFERÊNCIAS BIBLIOGRÁFICAS **79**

ANEXOS **81**

Anexo I – Código em linguagem Python intitulado main.py **81**

Anexo II – Código em linguagem Kivy intitulado G120.kv **90**

1. INTRODUÇÃO

O conceito automação industrial vem mutando-se constantemente, principalmente devido as mudanças e otimizações que a digitalização trouxe consigo. A atualidade está imersa em meio da quarta revolução industrial, que baseia-se na revolução digital, segundo o engenheiro e economista alemão Klaus Martin Schwab, a transformação que esta revolução tecnológica está trazendo consigo é diferente de tudo aquilo que o ser humano tem experimentado.

A segunda era da máquina se caracteriza por uma internet mais ubíqua, ou seja, “aquela que está presente em toda parte”, pela inteligência artificial e aprendizagem automática, características importantes da Indústria 4.0, conceito introduzido com a quarta revolução industrial. Para os professores Erik Brynjolfsson e Andrew McAfee do *Massachusetts Institute of Technology (MIT)* o efeito dessas novas tecnologias digitais está se manifestando intensamente por meio da automação.(MIT New Office, 2017)

O termo indústria 4.0 surgiu na Alemanha em 2011 na Feira de Hannover, onde foi introduzido o conceito de automação industrial com a integração de soluções tecnológicas inovadoras como a inteligência artificial e internet das coisas. Um ambiente com foco no desenvolvimento de alta tecnologia para o processo industrial, onde a forma de produção do mundo físico e virtual fundem-se integrando assim toda a planta operacional, dinamizando a produção.

Esse novo conceito propõe uma nova visão para a operação das fábricas um modelo com o controle dos processos produtivos descentralizados, contando com dispositivos inteligentes capazes de interconectar toda a cadeia de produção e logística, transformando a maneira como as máquinas se comunicam otimizando assim a produção e tornando os processos mais eficientes. A Indústria 4.0 possui como fundamentos a leitura e análise dos dados em tempo real, possibilitando assim a virtualização das fábricas, simulações e rastreamento dos processos produtivos e troca de informações de maneira inteligente e constante, culminando em tomadas de decisões mais ágeis e com maior exatidão (SIEMENS, 2021).

Segundo o economista Mauricio Canêdo, professor no Instituto Brasileiro de Economia da Fundação Getúlio Vargas do Rio de Janeiro, o Brasil apresenta

diferentes níveis da digitalização incorporada nos processos fabris, segundo Canêdo a inovação é mais propícia em países que contam com além de um mercado desenvolvido incentivos governamentais para fomentar esse desenvolvimento, como é o caso dos Estados Unidos.

Porém para ampliar a competitividade, as indústrias precisam reduzir tempos de máquina parada e de produção dos produtos, aumentar qualidade, flexibilizar a produção alinhando-se à demanda do mercado e com um valor compatível com seus consumidores. Para que essas exigências sejam possíveis entram nesse cenário além de equipamentos que possibilitem a automação da operação como cicloconversores de frequência e controladores lógicos programáveis, mas também a análise de dados e o monitoramento contínuo de variáveis pertinentes para a operação.

Com a necessidade do controle de velocidade e torque na maioria das aplicações industriais ou até mesmo focando em economia de energia, os cicloconversores de frequência vêm ganhando cada vez mais importância no setor industrial. São dispositivos capazes de variar simultaneamente a tensão e a frequência aplicado no estator do motor.

Para algumas aplicações mais complexas é raro encontrar o cicloconversor trabalhando de forma independente, nesses casos é necessária uma combinação como por exemplo um controlador lógico programável, para essa comunicação ser possível é necessário a troca de telegramas entre o drive e o controlador lógico programável.

Há também a possibilidade de comunicação entre o cicloconversor e o computador que se dá por meio de um software que se comunica no mesmo protocolo de comunicação, um exemplo seria o protocolo Ethernet. Assim sendo, para que seja possível a troca de telegramas é mandatório a utilização de linguagem de alto nível para assim conseguir administrar essas informações trocadas entre cicloconversor e computador.

Uma linguagem de alto nível que se destaca é o Python, criada sob o ideal de *“Programação de Computadores para todos”.* Essa linguagem possui código aberto, disponibilidade em Windows, Linux, Mac, Palm, entre outros sistemas, possui clareza de sintaxe, robustez tendo inúmeras funcionalidades mitigando as possibilidades de erros humanos. Tem como princípio a interoperabilidade, ou seja, comunicação, digitalização e

a integração entre sistema, tendo uma enorme gama de bibliotecas de computação, uma coleção de subprogramas que podem ser aliadas no desenvolvimento de um programa maior.

Este trabalho terá o foco de propor uma forma de monitorar os principais parâmetros de um cicloconversor de frequência em aplicações com controle de velocidade autônomo, o foco principal é aplicações de ventilação através da linguagem de programação Python. Será desenvolvido um sistema capaz de monitorar parâmetros básicos de operação bem como a análise dos mesmos.

Dada a importância da aplicação ventilação nos processos industriais como alto-fornos nas siderúrgicas, instalações de caldeiras, transportes pneumáticos e refrigeração faz-se necessário manter a confiabilidade e operabilidade dos equipamentos que compõe esses sistemas, monitorando variáveis e condições de operação do acionamento dessas aplicações. Além da manutenção térmica, os ventiladores são utilizados também na renovação de ar, facilita a filtragem de fumaças, poeiras, gases antes que sejam lançados no ar, evitando a degradação do meio ambiente.

2. CONVERSORES DE FREQUÊNCIA

Há alguns anos, quando se precisava de um controle de velocidade preciso eram utilizados motores de corrente contínua ou variadores que podiam ser mecânicos com polias e redutores, ou hidráulicos. Com o desenvolvimento da eletrônica de potência e a busca pelo aumento da produção com a necessidade da diminuição de custos passou-se a utilizar dispositivos eletrônicos de variação de velocidade, os que tiveram mais destaques foram os cicloconversor de frequência, equipamento versátil que permitiu a substituição dos motores de corrente contínua para os motores de indução em aplicações de controle de velocidade.

Essa foi uma mudança importante pois o motor de corrente alternada apresenta algumas vantagens em relação ao motor de corrente contínua como possuir o custo mais reduzido, um tamanho reduzido, oferecer melhor rendimento proporcionando redução no consumo de energia e menor aquecimento do motor, redução no custo de manutenção, devido ao motor de corrente alternada possuir uma topologia mais simples e com menos componentes, o motor de corrente contínua apresenta enrolamentos de campo e armadura, escovas comutadoras, entre outros, o que ocasiona maior custo durante a manutenção.

Os cicloconversores de frequência ou como também são conhecidos, drives, inversor de frequência vem ganhando cada vez mais importância nas indústrias, proporcionando melhorias nos processos de produção e economia de energia com o controle de velocidade e torque. Há algumas indústrias como de papel e celulose, na qual não se opera sem o controle de velocidade, há outras como bombas, ventiladores e centrífugas que podem se beneficiar com a redução de energia e aumento de eficiência.

Para melhor entender o inversor de frequência o item 2.1 irá abordar o funcionamento de um motor de indução.

2.1. MOTOR DE INDUÇÃO

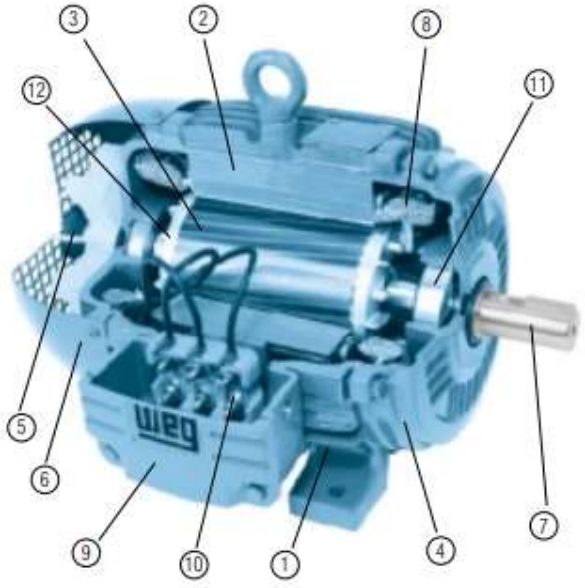
Motores trifásicos de corrente alternada são conversores eletromagnéticos de energia, ou seja, convertem energia elétrica em energia mecânica através da indução eletromagnética.

2.1.1. Constituição do Motor Elétrico de Indução

O motor assíncrono é constituído basicamente pelos seguintes elementos:

- ✓ Circuito magnético estático que é composto por chapas ferromagnéticas empilhadas e isoladas entre si, conhecida como estator, por bobinas localizadas nas cavidades abertas do estator e alimentadas pela rede de correntes alternadas, o número de grupos caracterizam o motor em monofásico ou polifásico;
- ✓ O rotor que é constituído por um núcleo ferromagnético sobre o qual encontra-se um enrolamento ou um conjunto de condutores paralelos, nos quais é induzida a corrente devido ao campo magnético rotativo que é resultado de uma corrente trifásica alternada aplicada no circuito magnético estático. O rotor é apoiado em uma cavidade que transmite energia mecânica à carga.

Figura 1 – Motor de Indução Trifásico.



Fonte: WEG, Catálogo Geral de Motores WEG,2020.

Elementos referente ao estator:

- Carcaça (1) – Estrutura suporte do conjunto, resistente à corrosão, possui construção robusta em ferro fundido, aço ou alumínio injetado;
- Núcleo de chapas (2) – Chapas de aço magnético, tratadas termicamente com o objetivo de reduzir as perdas no ferro;
- Enrolamento trifásico (8) – Conjunto de bobinas, uma para cada fase, formando um sistema trifásico alimentado por uma rede de corrente alternada trifásica.

Elementos referentes ao Rotor:

- Eixo (7) – Transmite a potência mecânica desenvolvida pelo motor para a carga;
- Núcleo de chapas (3) – Possuem as mesmas características das chapas do estator;
- Barras e anéis de curto-circuito (12) – Onde são induzidas correntes.

Outras partes do motor de indução trifásico:

- Tampa (4);
- Ventilador (5);
- Tampa defletora (6);
- Caixa de ligação (9);
- Terminais (10);
- Rolamentos (11).

2.1.2. Princípio de funcionamento do Motor de Indução

O princípio de funcionamento de um motor de indução é baseado na Lei de Faraday, ou seja, quando uma corrente trifásica alternada (cada um dos 03 (três) enrolamentos é percorrido por uma corrente elétrica defasadas de 120° entre si) passa pelo enrolamento do estator é gerado um campo magnético rotativo de intensidade constante, a velocidade do campo magnético é conhecida como velocidade síncrona

do motor. As barras do rotor ligadas aos anéis funcionam como condutores fechados imersos em um campo magnético girante, dessa forma pela Lei de Faraday será induzida uma força eletromotriz que produzirá uma corrente induzida nas barras do rotor e consequentemente um campo de polaridade oposta à do campo girante. Dessa maneira surge forças magnéticas no rotor com mesmo sentido do campo eletromagnético do estator desenvolvendo assim um conjugado motor que fará com que o rotor gire.

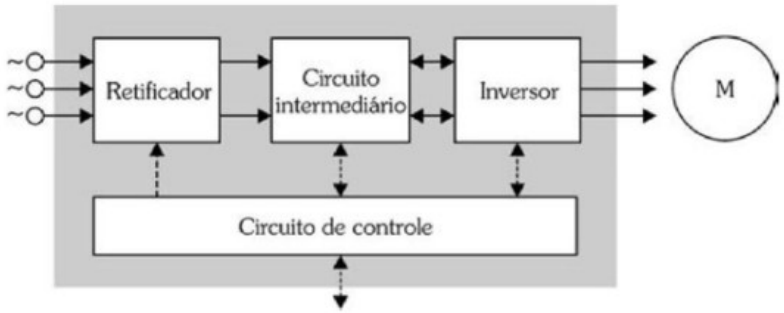
No Brasil as concessionárias de energia elétrica fornecem tensão alternada com uma frequência de 60Hz e em oposição a maior parte dos motores elétricos encontrados nas indústrias e que consomem essa energia rodam com velocidade nominal de 3600rpm. A maioria das aplicações não funcionam nessa velocidade, sendo assim ou utiliza-se um redutor mecânico ou, o que a maioria opta, utiliza-se o inversor de frequência que é capaz de gerar uma rampa de desaceleração, reduzir a frequência e a tensão fazendo assim com que o motor opere conforme os requisitos da carga ou seja, velocidade e tensão desejada.

2.2. CONVERSORES DE FREQUÊNCIA

Com o advento da Eletrônica de Potência foi possível o surgimento de dispositivos eletrônicos de variação de velocidade. O método que vem ganhando mais destaque e se mostrando mais completo e eficiente para o controle de velocidade de motores de indução trifásico consiste na variação da frequência da fonte de alimentação através de cicloconversores de frequência.

A topologia dos inversores de frequência é dividida em 03 (três) partes, a primeira é a retificação de entrada, a segunda é o circuito intermediário do link DC e a terceira a saída modulada.

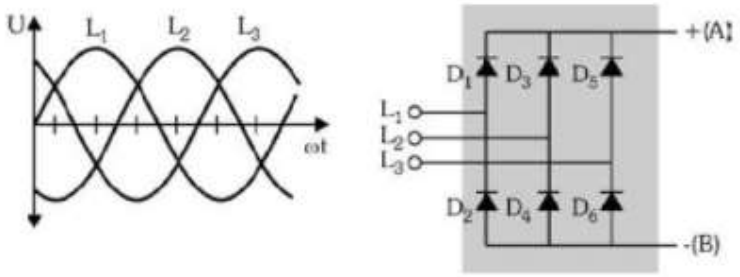
Figura 2 – Diagrama simplificado dos principais blocos do Inversor de Frequência.



Fonte: FRANCHI, 2009.

O retificador é responsável pela retificação do sinal alternado de entrada cujo módulo da tensão e a frequência são constantes. O retificador produz em sua saída uma tensão constante (DC). O modelo mais encontrado no mercado é o de 06 (seis) pulsos, há retificadores não regenerativos, passivos, na qual utilizam diodo ou tiristores em sua composição, permitindo assim um único sentido para o fluxo de corrente, e há retificadores regenerativos, que utilizam em sua composição IGBTs com diodos de roda livre, permitindo assim a regeneração da energia.

Figura 3 – Aplicação de uma fonte de tensão trifásica em uma ponte retificadora.



Fonte: FRANCHI, 2009.

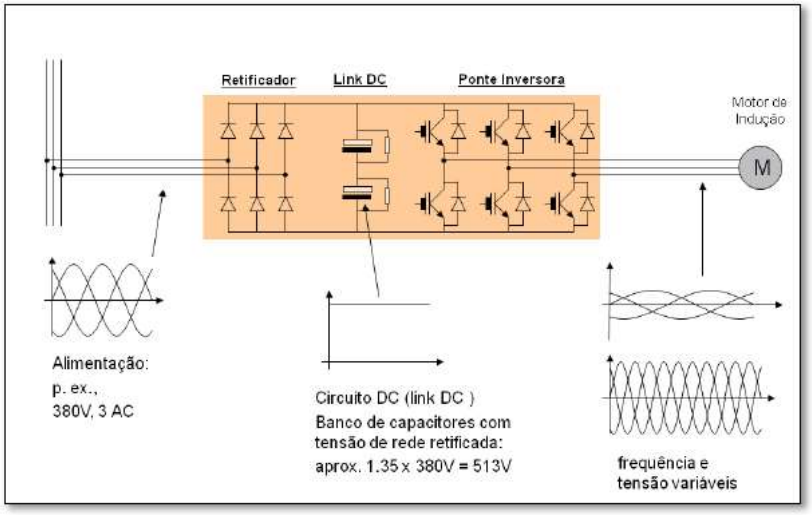
Logo após o retificador o conversor de frequência apresenta o circuito intermediário ou filtro ou ainda link DC, que é responsável por suavizar o sinal através de seu banco de capacitores. Chama-se a tensão de saída do barramento CC de tensão de link DC, quando o motor está a vazio espera-se uma tensão de link DC de 1,41 vezes a tensão de linha, com carga parcial espera-se 1,35 vezes a tensão de linha e quando está com carga total espera-se uma tensão de link DC 1,32 vezes a tensão de linha.

Após a retificação o sinal deve ser transformado novamente em alternado sendo então transferido para o terceiro ponto, a ponte inversora, essa conversão é feita mais comumente através de um conjunto de IGBTs que são chaveados de maneira a obter a forma de onda de tensão de saída desejada.

A seção inversora recebe então a alimentação do barramento CC, através de um controle específico e vital para o funcionamento do conversor de frequência, modula a tensão através da largura de pulso (*PWM – Pulse Width Modulation*) transformação essa que utiliza-se do princípio da modulação por largura de pulso PWM.

A figura a seguir ilustra as formas de ondas e etapas de funcionamento de um conversor de frequência.

Figura 04 – Topologia do conversor de frequência AC/AC.

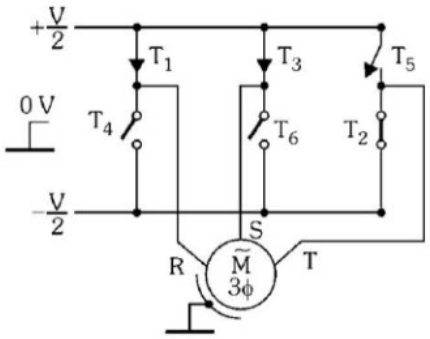


Fonte: Godinho, 2017.

2.2.1. Controle de chaveamento dos tiristores do Inversor de frequência

Para ilustrar melhor o chaveamento dos tiristores em um inversor de frequência segue abaixo a Figura 05 com um esquemático:

Figura 5 – Diagrama esquemático do inversor de frequência.



Fonte: FRANCHI, 2009.

O controle desses circuitos é feito de maneira a obter um sistema de tensão alternada defasadas em 120º uma das outras. O controle dos transistores de potência é feito através de microcontroladores digitais responsável pela geração de pulsos. Atuando sobre a variação do chaveamento dos transistores controla-se a frequência do sinal trifásico gerado.

Como a maioria das aplicações em indústrias atualmente são acionadas por inversores trifásicos será mostrado o princípio de funcionamento para esses casos. Por se tratar de um sistema de seis transistores, serão ligados três a três, com seis combinações validas. Por meio do controle serão realizadas as combinações presentes na Tabela 1.

Tabela 1 – Combinação para acionamento dos IGBTs de um Conversor de Frequência trifásico.

1° tempo	2° tempo	3° tempo	4° tempo	5° tempo	6° tempo
T_1, T_2, T_3	T_2, T_3, T_4	T_3, T_4, T_5	T_4, T_5, T_6	T_5, T_6, T_1	T_6, T_1, T_2

Fonte: FRANCHI, 2009.

Segue abaixo a análise para o primeiro tempo (1° tempo) referente ao chaveamento dos IGBTs os demais tempos seguem da mesma forma.

(I) Vrs = (+V/2) – (+V/2) = 0

(II) Vst = (+V/2) – (-V/2) = +V

(III) Vtr = (-V/2) – (+V/2) = -V

O barramento CC alimenta o circuito inversor com +V/2 e -V/2 pois possui referência central (terra), para alimentar o motor são necessárias as tensões de linha, aqui denominadas Vrs (representa a diferença de potencial entre a fase R e S), Vst (representa a diferença de potencial entre a fase S e T) e Vtr (representa a diferença de potencial entre a fase T e R), defasadas de 120°.

A tabela seguinte ilustra as tensões de linha aplicada no motor para cada tempo do chaveamento dos IGBT.

Tabela 2 – Tensões aplicadas no motor.

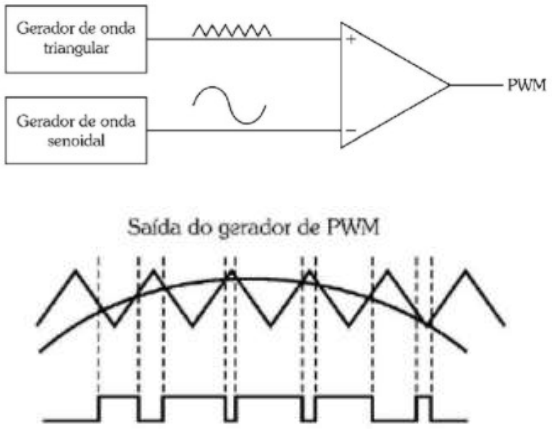
IGBTs	Vrs	Vst	Vtr	tempos
T_1, T_2, T_3	0	+V	-V	1° tempo
T_2, T_3, T_4	-V	+V	0	2° tempo
T_3, T_4, T_5	-V	0	+V	3° tempo
T_4, T_5, T_6	0	-V	+V	4° tempo
T_5, T_6, T_1	+V	-V	0	5° tempo
T_6, T_1, T_2	+V	0	-V	6° tempo

Fonte: FRANCHI, 2009.

Como os IGBTs trabalham como chaves, a forma de onda de tensão de saída do inversor de frequência é sempre quadrada, para obter a tensão de saída mais próxima de uma senoide, os transistores chaveiam modulando sua largura de pulso denominando tal método de PWM (*Pulse Width Modulation*).

A Figura 06 (seis) aborda o funcionamento do chaveamento PWM, modulação por largura de pulso.

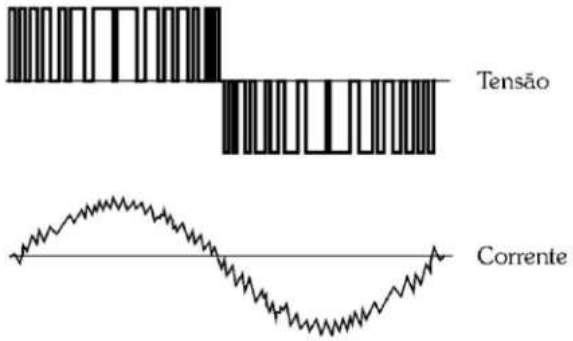
Figura 6 – Princípio da geração do PWM.



Fonte: FRANCHI, 2009.

Utilizando o microprocessador as funções de controle do PWM são efetivamente realizadas pela combinação de uma onda triangular, que determina a frequência de chaveamento, e uma senoidal que determinam a largura dos pulsos e, portanto, a tensão rms (*root mean square*) na saída do inversor.

Figura 7 – Formas de onda de tensão e de corrente na saída do inversor.



Fonte: FRANCHI, 2009.

Inicialmente o IGBT é chaveado por um curto período permitindo apenas que uma pequena parcela da corrente chegue até o motor, e vai aumentando a largura do pulso conforme vai chegando próximo à corrente nominal do motor ao alcançar tal valor a largura do pulso vai diminuindo novamente. Para a parcela negativa das ondas senoidais esse chaveamento ocorre da mesma forma, porém com o valor negativo da tensão contínua.

3. SINAMICS G120

Esse capítulo abordará uma apresentação sobre uma família de conversores de frequência difundida no mercado e que será usada para a elaboração dos testes e simulações objeto deste trabalho.

A família de drives SINAMICS G120 atendem perfeitamente aplicações com demandas de baixa e média resposta dinâmica, como bombas, ventiladores, compressores, correias transportadoras, misturadores e extrusoras. É projetado de forma a atender controle preciso e econômico de velocidade e torque para motores trifásicos. Possuem um range de potência que varia de 0,37 kW a 250 kW.

O SINAMICS G120 é um conversor modular que essencialmente compreende 02 (duas) unidades de função a unidade de controle intitulada CU e o módulo de potência intitulado PM. A unidade de potência possui tecnologia IGBT fornecendo tensão modulada através da largura de pulso com frequência selecionável ao motor, a unidade de controle tem função de monitorar e controlar o módulo de potência e o motor conectado. Possui capacidade de comunicação via PROFIBUS ou PROFINET, nesse trabalho iremos utilizar para simulação o padrão PROFINET, Ethernet industrial, que destaca-se pela troca rápida e segura dos dados, sua flexibilidade e eficiência criam condições para um ambiente otimizado, culminando em aumento de produtividade e portanto culminando em uma maior competitividade.

O SINAMICS G120 pode ser parametrizado de duas maneiras diferentes, a primeira seria inserindo os parâmetros diretamente no drive através de ferramentas de interface/ painéis de operação, que seria o BOP - *Basic Operator Panel* ou o IOP - *Intelligent Operator Panel*, ou através de softwares de edição que possui uma gama enorme de possibilidades, a SIEMENS disponibiliza dois softwares, que antes de qualquer iteração recomenda-se a realização de treinamentos para capacitação, O STARTER e o STARTDRIVE. A primeira permite comandos com uma interface de comissionamento, parametrização de entradas e saídas e diagnóstico mais otimizada que economiza tempo e mitiga erros. O STARTDIVE pertence ao ambiente integrado do TIA PORTAL (*Totally Integrated Automation Portal*) que consiste em uma plataforma integrada de softwares de programação, que depende da aquisição de

licenças pelo usuário, para a configuração de PLC (*Programmable Logic Controller*), IHM (*Human-Machine Interface*) e Conversores de frequência.

No Conversor SINAMICS G120 há dois tipos de parâmetros os de leitura, iniciados pela letra “r” e os de escrita iniciados pela letra “p”, ambos são seguidos do número que compõe o parâmetro e o índice ou o campo de bits. Por exemplo o r0026 é um parâmetro de leitura que exibe o valor real da tensão de link DC, o r0052.0...15 exibe uma palavra (*word*) e cada bit dessa palavra tem um significado por exemplo o r0052.2 ou bit 2 exibe 1 se o drive está em operação ou 0 se o mesmo não está em operação, já o parâmetro r0720 [0...4] exibe o número de entradas e saídas, o r0720[0] ou índice 0 exibe o número de entradas digitais já o índice 1 exibe o número de saídas digitais e assim para cada índice existente, o p0305 por exemplo é um parâmetro de escrita para informar ao drive qual o valor da corrente nominal do motor.

Cada parâmetro do SINAMICS G120 possui um tipo de dado específico (*Data Type*) que deve ser respeitado durante a parametrização para uma eficiente transmissão de informação e assim consequentemente a aplicação acionada funcionar como espera-se.

Tabela 3 – Data Type: Tipos de dados que os parâmetros podem assumir.

Data Type	Descrição
Integer8	Número inteiro de 8 bits
Integer16	Número inteiro de 16 bits
Integer32	Número inteiro de 32 bits
Unsigned8	Número sem sinal de 8 bits
Unsigned16	Número sem sinal de 16 bits
Unsigned32	Número sem sinal de 32 bits
FloatingPoint32	Número com ponto flutuante de 32 bits

Fonte: Elaborada pela autora.

3.1. CONTROLE DE VELOCIDADE

Após entender um pouco do conversor de frequência da família SINAMICS G120 é possível compreender como se dá a parametrização de um controle de velocidade simples, muito utilizado nas aplicações industriais modernas.

O controle de velocidade é uma parte essencial de muitas aplicações de automação industrial, os drives da família SINAMICS G120 podem ser usados para resolver um grande número dessas tarefas, esse item tem o objetivo de descrever a implementação de um controle de velocidade simples, aplicando o conceito de velocidades fixas. O foco principal desse controle de velocidade simples será aplicações de “ventilação”.

O princípio da operação dessa aplicação é que através do controle da velocidade controla-se a potência de um ventilador, essa velocidade é então definida por um inversor, 03 (três) entradas digitais funcionaram como chaves seletoras, essa seleção possibilitará ligar o motor com velocidade baixa, ligar o motor com velocidade média e ligar o motor com velocidade alta, o conversor de frequência então controla o motor na velocidade selecionada.

Antes do início de qualquer parametrização no drive é sempre recomendado realizar um *reset* de fábrica no equipamento, este procedimento restaurará todos os parâmetros para o padrão de fábrica mitigando qualquer inconsistência com parametrizações anteriores, para realizar tal processo basta fazer o parâmetro p0010, parâmetro de comissionamento do inversor, assumir o valor 30 (trinta) indicando que será realizado o reset de fábrica no drive, e o p0970, parâmetro de reinicialização dos parâmetros do drive, assumir o valor 01 (um), após alguns segundos os parâmetros P0010 e p0970 assumiram o valo 0 (zero) e o drive então estará no padrão de fábrica e pronto para reconhecimento dos dados do motor, ou seja, o comissionamento.

Após o *reset* de fábrica, antes de iniciar a parametrização, é necessário ainda realizar o comissionamento básico do drive, ou seja, fazer o drive reconhecer o motor que ele irá acionar. Para iniciar tal procedimento será necessário fazer o parâmetro p0010, parâmetro de comissionamento do drive, assumir o valor 01 (um) informando o drive que serão inseridos os dados do comissionamento rápido, assim sendo no parâmetro p0304 será inserido o valor da tensão nominal do motor, no p0305 o valor da corrente nominal do motor, no p0307 o valor da potência nominal do motor, no

p0310 será inserido o valor da frequência do motor, no p0311 a velocidade nominal do motor em rpm, no p0400 a tensão de alimentação, no p1080 a velocidade máxima em rpm, no p1082 a velocidade mínima, no p1120 o tempo de rampa de aceleração em segundos, no p1121 o valor da rampa de desaceleração em segundos, no p1300 será selecionado o modo de controle do inversor, e por fim o p1900 assumira o valor 02 (dois) e fará o reconhecimento do motor, após essa sequência o p3900 assumirá o valor 03 (três) o que indicará que o drive foi comissionado e está pronto para receber a parametrização da aplicação.

Após o reset de fábrica e o comissionamento o drive está pronto para receber a parametrização do controle de velocidade a partir de velocidades fixas. O drive da família SINAMICS G120 contempla macros, que quando selecionadas pelo parâmetro p0015, pré configura alguns parâmetros tornando assim mais simples e fácil a parametrização de algumas aplicações. Esse será o caso para a configuração das velocidades fixas, será utilizado então a macro 12 (doze), assim o p0015 assumira o valor 12 (doze), para funcionar como espera será necessário ainda fazer algumas configurações como segue a seguir:

- ✓ p0840 (Parâmetro ON/OFF1) deverá assumir a leitura do status do *setpoint* da velocidade fixa o parâmetro r1025.0 fazendo com que qualquer entrada digital seletora de velocidade fixa possa acionar ou desligar o motor.
- ✓ p1000 (parâmetro de seleção de *setpoint* de velocidade) deverá assumir o valor 03 (três) que indica que o *setpoint* de velocidade será realizado por velocidade fixa.
- ✓ p1001 (parâmetro de velocidade fixa de *setpoint* 01) deverá receber o valor da velocidade baixa de *setpoint.*
- ✓ p1002 (parâmetro de velocidade fixa de *setpoint* 02) deverá receber o valor da velocidade média de *setpoint.*
- ✓ p1003 (parâmetro de velocidade fixa de *setpoint* 03) deverá receber o valor da velocidade alta de *setpoint.*
- ✓ p1020 (parâmetro que indica por onde será realizado a seleção da velocidade fixa 01) deverá receber o parâmetro r0722.0 leitura do status da entrada digital DI0.
- ✓ p1021 (parâmetro que indica por onde será realizado a seleção da velocidade fixa 02) deverá receber o parâmetro r0722.1 leitura do status da entrada digital DI1.
- ✓ p1022 (parâmetro que indica por onde será realizado a seleção da velocidade fixa 03) deverá receber o parâmetro r0722.2 leitura do status da entrada digital DI2.

✓ p1113 (parâmetro de *setpoint* de inversão) deverá ser desabilitado pois é um parâmetro pré configurado pela macro 12 (doze), sendo assim esse parâmetro deverá receber 0 (zero).

✓ p2103 (parâmetro de reconhecimento de falha) deverá receber o parâmetro r0722.3, ou seja, o reconhecimento de falhas será realizado pela entrada digital DI3.

✓ p0848 (parâmetro de emergência OFF3) deverá receber o parâmetro r0722.4, ou seja, a parada de emergência será realizada pela entrada digital DI4.

✓ p1135 (parâmetro que informa o tempo da rampa de desaceleração da parada de emergência OFF3) esse parâmetro deverá receber o valor em segundos da rampa de desaceleração da parada de emergência.

A parametrização do SINAMICS G120 como já dito anteriormente pode se dar de duas formas ou inserindo os parâmetros diretamente no drive através de interfaces de operação como o BOP (*Basic Operator Panel*) ou o IOP (*Intelligent Operator Panel*) ou através de softwares de edição como o STARTER e o STARTDRIVE. Algumas vezes um controlador logico programável PLC consegue ditar regras para o inversor seguir e dessa forma acionar o motor, mas para todas essas configurações é necessário compreender como que esses equipamentos conseguem se comunicar, trocar informações e dado, o próximo tópico deste trabalho irá abordar esse conceito.

3.2. PROTOCOLO DE COMUNICAÇÃO S7

Redes de comunicação permitem que sejam enviadas e recebidas informações, em um ambiente industrial a comunicação através de redes é essencial para que este se inclua na nova era da digitalização e permaneça cada vez mais competitiva.

Atualmente, existem diversos protocolos de comunicação industrial disponível, para integrar plantas industriais e processos, o que em grande parte significa colocar os dispositivos de acionamento e automação em uma mesma rede, é exigido que os computadores industriais se comuniquem de maneira correta, eficiente e segura, com isso um ponto chave é a escolha do protocolo de comunicação.

Quando se fala em Protocolos de comunicação refere-se ao conjunto de regras que regem a comunicação das redes, sendo assim para dois equipamentos se comunicarem é necessário que ambos compreendam o mesmo protocolo. Para

compreender melhor essa comunicação é preciso entender o modelo OSI (*Open System Interconnection Reference Model*) que rege a base de como as redes modernas são estruturadas.

O modelo OSI separa em 07 (sete) camadas a operação das redes, onde as camadas sucessoras vão envelopando as camadas antecessoras, segue uma breve descrição de cada camada da camada mais externa para a mais interna:

✓ Camada Física (*Physical Layer*): Essa camada é responsável por enviar e receber os bits através de um meio físico. Nela são especificadas as interfaces elétricas/óticas e mecânicas.

✓ Camada de Enlace de Dados (*Data Link Layer*): Essa camada estabelece endereços para identificar os nós da rede além de dividir os pacotes a serem enviados pela camada física de acordo com a arquitetura utilizada.

✓ Camada de Rede (*Network Layer*): Essa é a camada responsável pelo endereçamento do dado, ou seja, leva-o do emissor até o receptor, traduzir o endereço físico para os endereços de rede.

✓ Camada de Transporte (*Transport Layer*): Essa camada é responsável pela movimentação/transporte dos dados da camada de sessão a sua antecessora para a camada de rede a sua sucessora.

✓ Camada de Sessão (*Session Layer*): Essa camada tem função de permitir que diferentes dispositivos se comuniquem. É responsável também por garantir a conexão até que toda a transmissão de pacotes de dados sejam realizadas, por realizar tarefas de segurança, reconhecimento de endereços.

✓ Camada de Apresentação (*Presentation Layer*): Essa camada é responsável por apresentar a informação de maneira que as aplicações consigam interpretá-las, ou seja, convertê-las no formato do protocolo utilizado.

✓ Camada de Aplicação (*Application Layer*): Essa camada é responsável por permitir que a comunicação seja possível, é capaz de fornecer acesso à rede às aplicações.

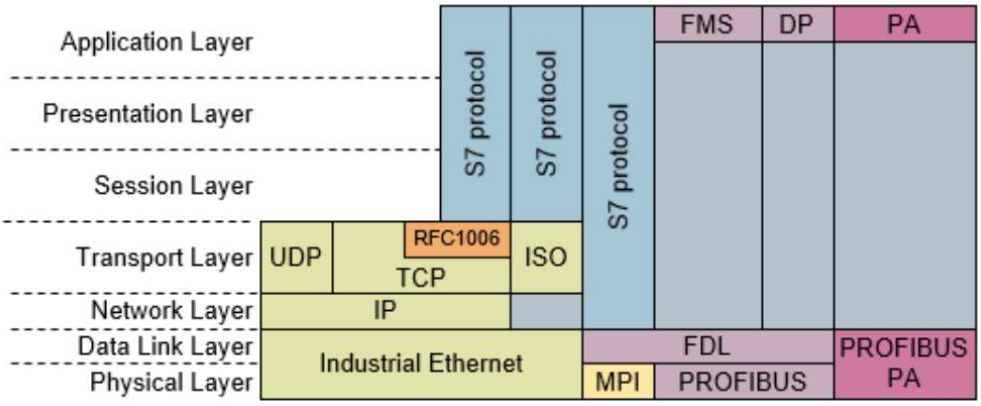
O Protocolo Ethernet é o protocolo de comunicação mais utilizado em redes locais (*LAN – Local Area Network*), é basicamente o padrão responsável por definir como os dados são transmitidos pelo meio físico, esse protocolo opera nas camadas físicas e de enlace.

O endereçamento da rede é realizado através de um número único de 06 (seis) bytes, atrelado a placa de rede de cada computador, em uma rede não é aceitável dois dispositivos com o mesmo número *MAC- Media Acess Control*, já o *IP- internet protocol*, protocolo de comunicação aberto utilizado para encaminhamento de dados que está na camada intitulada rede do modelo OSI, possui um endereço de rede variável e a mudança de uma rede para outra leva à um IP diferente.

O Protocolo de Controle de Transmissão, do inglês *Transmission Control Protocol – TCP*, é um dos protocolos de comunicação referente a camada de transporte do modelo OSI, verifica se os dados são enviados na sequência correta e sem erros via rede.

No âmbito da comunicação entre equipamentos da marca SIEMENS é utilizado um protocolo fechado intitulado *S7 Comunication* lançado em 1994 juntamente com os controladores da família S7. Esse protocolo opera com diversos meios físicos, nesse trabalho iremos focar no trabalhado com o protocolo Ethernet. Através desse protocolo pode-se escrever e ler dados dos dispositivos SIEMENS além da coleta de diagnósticos dos mesmos. Por exemplo os parâmetros e variáveis da unidade de controle do conversor de frequência SINAMICS G120 pode ser acessado de forma direta através do protocolo *S7 comunications*.

Figura 8 – Protocolo S7 no modelo de referência ISO-OSI.



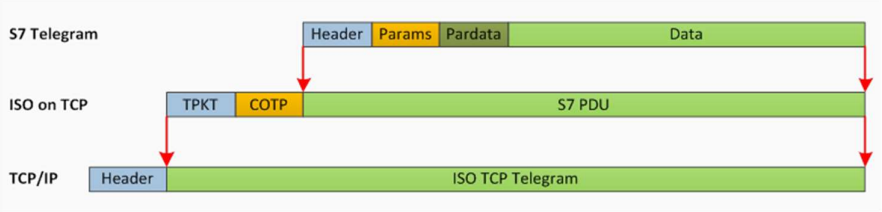
Fonte: SIEMENS AG, 2019.

A Figura 08 (oito) apresenta como o protocolo de comunicação S7 se referência ao modelo padrão da estrutura das redes de comunicação modernas.

O Protocolo *S7 comunications* possui orientação em blocos, cada bloco, denominado PDU – *Protocol Data Unit*, contém uma informação ou comando, inicia-

se com um cabeçalho que contém o tamanho da informação transmitida, posteriormente tem-se a definição do parâmetro, depois vem o dado do parâmetro e por fim o bloco de dados. Para uma comunicação efetiva há obrigatoriedade apenas dos dois primeiros elementos, os dois últimos irão depender se a transmissão da mensagem está ocorrendo de um– Controlador Logico Programável - PLC ou de uma Interface Homem Máquina – IHM ou de um Conversor de frequência. Caso a transferência de dados ocorra através de um conversor de frequência, objeto deste trabalho, será necessário informar o tamanho da informação, o parâmetro e o tipo de dado que será transmitido, escrito ou lido.

Figura 9 – Encapsulamento de Protocolos.



Fonte: SIEMENS AG.

A figura acima contempla o encapsulamento do protocolo de comunicação S7, ISSO on TCP e TCP/IP para melhor ilustração da explicação.

4. PYTHON

Após o detalhamento sobre o protocolo de comunicação S7, este capítulo trará uma descrição da linguagem de programação Python e da interface que foi construída para comunicação e monitoramento de um SINAMICS G120, detalhando as principais bibliotecas que foram utilizadas.

Python é uma linguagem de programação de alto nível, poderosa e com uma abordagem simples mas efetiva orientada a objeto, o que facilita a organização e torna o programa mais estável mitigando erros. Essa é uma linguagem de código aberto com uma ampla gama de bibliotecas, programas, diversos módulos e ferramentas disponível gratuitamente, ou seja, o usuário pode utilizar módulos, partes de outros programas em um novo projeto, trabalhando como uma linguagem modular e tornando o programa menor e menos suscetível ao erro, é ideal para as mais diversas aplicações.

Para desenvolver o código em linguagem Python é necessário uma interface que consolide as ferramentas básicas necessárias para escrever e testar o mesmo, esse é o IDE, do inglês *Integrated Development Environment*, ou seja Ambiente de Desenvolvimento Integrado. Esse é um software que reúne ferramentas de desenvolvimento em uma única interface gráfica do usuário, ele possui um editor de texto que auxilia na elaboração do código através de funcionalidades, como destaque da sintaxe e recursos de preenchimento automático, realiza testes e aponta posição de erros.

Para esse trabalho utilizou-se para construção do código em Python o ambiente de desenvolvimento integrado Pycharm, que é uma IDE dedicada à linguagem Python que fornece além do ambiente editor uma análise de código, ferramenta de *debugging* e auto complemento de códigos.

Para a construção de um código em Python é necessário primeiro entender os principais tipos de variáveis que são possíveis a utilização, sendo assim essa linguagem comporta variáveis do tipo Números podendo ser inteiras especificadas por “int” ou reais ou com parte fracionária especificados por “float”, além dos números essa linguagem também pode manipular sequências de caracteres conhecidas por *Strings*, essas podem ser delimitadas por aspas simples ou duplas, outro tipo de variável são as Listas, estruturas de dados compostas, usadas para agrupar outros

valores, os dados que a compõe são mutáveis, ou seja é possível aumentar ou diminuir os dados de uma lista, bem como modifica-los.

Um código em sua maioria também é composto por funções que carregam consigo condições que controlam o fluxo do programa como o laço de repetição "*while*" que executa uma série de comandos enquanto a condição contida nele é satisfeita, outro comando de controle de fluxo é o "*if*", "*elif*" e *"else*" , o comando *"if"* é responsável por apresentar a primeira opção de condição, o *"elif"* que vem do inglês *"else if"* (mais se / caso contrário) e apresenta uma segunda opção diferente da primeira e o *"else"* que traz uma terceira opção diferente das duas primeiras. Uma terceira maneira de conseguir controlar o fluxo do código é utilizando o comando *"for",* que é um ciclo de repetição do código que está identado à ele até que toda a condição seja satisfeita.

Para definir essas funções, que podem ser muito úteis para executar um bloco que será repetido várias vezes, basta digitar "def' e em seguida o nome da função.

Para os códigos em linguagem Python uma forma de organizar dados e funcionalidades é através de classes, criando uma classe é possível criar um novo "tipo" de objeto permitindo que novas instâncias desse tipo sejam produzidas, assim cada instância da classe pode ter atributos anexados a ela, dessa maneira teremos a programação orientada a objetos. Para criar uma classe basta digitar "class" e em seguida o nome dado a classe.

As bibliotecas são outro ponto importante para a construção de um código, elas são basicamente um conjunto de novos módulos/códigos que podem ser uteis para a criação de um novo programa minimizando linhas de código. Quando se instala o Python várias bibliotecas já são instaladas em conjunto, são as bibliotecas padrão, porém além das bibliotecas básicas pode ser necessário a instalação de outras, para isso é necessário acessar o site da biblioteca e baixar e instalar a mesma, na maioria das vezes esse processo pode ser realizado pelo prompt de comando. Após a instalação da biblioteca para utilizar a mesma no código basta importá-la para a programação utilizando o comando "import" seguido do nome da biblioteca.

Para a elaboração da interface de monitoramento e criação de gráficos para posterior análise de uma aplicação de velocidades fixas no SINAMICS G120 foram utilizadas além das funções das bibliotecas básicas do Python as bibliotecas Sinamics, Snap7, Matplotlib e o *framework* Kivy, essas serão mais detalhadas nos itens a seguir para melhor compreensão do código que foi desenvolvido.

4.1. BIBLIOTECAS

4.1.1. Biblioteca Python-Snap7

A biblioteca Snap7, criada por Gijs Molenaar e Stephan Preeker foi baseada na Snap7 de Davide Nardella, apresenta uma solução estrutural contemplando pacote de comunicação Ethernet multiplataforma de código aberto para sistemas operacionais de 32 e 64 bits para possibilitar o trabalho com interfaces nativa com protocolo de comunicação S7, ou seja possibilita a construção de códigos que permitam a comunicação com drives da família SINAMICS e PLC da família S7 1500 e S7 1200, bem como o módulo lógico da SIEMENS LOGO, dispositivos SIEMENS que se comunicam através do protocolo de comunicação S7.

Como já abordado anteriormente o Protocolo S7 é a espinha dorsal das comunicações da SIEMENS. O formato mais usual de armazenamento de dados é o LITTLE-Endian onde a sequência de informações dá-se em ordem crescente do byte menos significativo para o mais significativo, já o formato interno dos dados de dispositivos SIEMENS segue o padrão BIG-Endian dá-se na ordem contraria do byte mais significativo para o menos significativo.

A ideia por trás desse método é poder ler ou escrever um buffer de bytes, de um PLC ou Drive SIEMENS que se comunica através do protocolo S7 utilizando o comando *read* ou o comando *write*. Abaixo segue os as classes de dados que essas informações podem assumir:

- ✓ BIT – Sem sinal com 01 bit;
- ✓ WORD – É representado por um número inteiro, sem sinal com 16 bits;
- ✓ INT – É representado por um número inteiro com sinal e 16 bits;
- ✓ DWORD – É representado por um número inteiro, sem sinal 32 bits;
- ✓ DINT – É representado por um número inteiro, com sinal 32 bits;
- ✓ REAL – É representado por um número inteiro com ponto flutuante 32 bits;
- ✓ S7 Strings;
- ✓ S7 Array.

4.1.2. Biblioteca Python-Sinamics

A biblioteca python-sinamics contém a definição de classe para controle e monitoramento de drives SINAMICS utilizando método para leitura e escrita de parâmetros, é realizado via TCP/IP utilizando o protocolo da camada de aplicação S7. A biblioteca python-sinamics usufrui da python-snap7, utilizando a estrutura já definida para a comunicação via protocolo S7.

Essa biblioteca apresenta todas as definições para conexão, escrita e leitura de parâmetros além de retornar possíveis erros do usuário. Para iniciar a conexão, responsável por possibilitar a troca de informações entre o computador e o drive, é definida uma função que solicita como atributos o *IP -Internet Protocol* e informações de rack e slot, para os drives essas duas últimas informações são atribuídas o valor 0 (zero), para a conexão com os controladores lógicos programáveis esses itens podem assumir valores diferentes.

Após estabelecer a conexão com o drive inicia-se a escrita e leitura dos parâmetros do drive da família SINAMICS G120. Para realizar a leitura será necessário informar algumas definições do parâmetro a ser lido, o *number* que é o número do parâmetro que será acessado, *data_type* que será o tipo de dado do parâmetro se é inteiro, real, binário e assim por diante, o index do parâmetro quando o mesmo permitir configuração, para alguns parâmetros esse item não é configurado, o *drive_object* se refere ao número do drive, para o objetivo deste trabalho esse item não será configurado pois será sempre igual a um como a configuração padrão e o *no_unpack* caso esse item assumir como verdadeiro, diferente do padrão que é falso, esse item fará a informação do parâmetro ser lida em hexadecimal. Segue a definição para leitura de parâmetros no drive através do uso da biblioteca:

```
      def read_parameter(self, number, data_type, index=0, drive_object=1,
no_unpack=False):
              """ This method can be used to read any parameter in the inve
rter
```

Para a escrita de dados no inversor também será necessário informar o *number, data_type* e o *drive_object* da mesma forma que a leitura dos parâmetros. Segue a definição para a escrita de parâmetros no drive através do uso da biblioteca:

```
      def
write_parameter(self, number, data_type, index=0, drive_object=1):
```

""" Este método pode ser usado para escrever um parâmetro do inversor SINAMICS. O tipo de dados só parâmetro a ser escrito deve ser conhecido. Se a escrita falha, uma exceção é gerada por *python-sinamics* snap7.

Um ponto importante é a estrutura de caracteres e como é definido o *data-type* de cada parâmetro que segue uma regra para formatação dessas informações dependendo do tipo de dado de cada parâmetro. Essa regra é formalizada na biblioteca *python-snap7* e organizada e definida no início da biblioteca *python-sinamics*, na tabela a seguir cada letra refere-se a um número com o tamanho de cada estrutura:

Tabela 4 - Estrutura de caracteres.

Formato Python	Tamanho	Data Type
B	8 bits – 1 byte	Integer8 (Byte)
I h H	32 bits – 4 byte 16 bits – 2 byte 16 bits – 2 byte	Integer16 (Int/Word)
i H	32 bits – 4 bytes 16 bits – 2 byte	Unsigned16 (Int/Word)
B	8 bits – 1 byte	Unsigned8 (Byte)
I L L	32 bits – 4 byte	Unsigned32(DInt/DWord)
f	32 bits – 4 byte	FloatingPoint32 (real)

Fonte: Elaborada pela autora.

4.1.3. Biblioteca Matplotlib

A biblioteca Matplotlib criada pelo biólogo e neurocientista americano John D. Hunter é voltada para a criação de gráficos e visualização de dados em geral, sejam elas estáticas animadas ou iterativas, feita com a linguagem de programação Python e sua extensão de matemática NumPy.

O método interno à biblioteca Matplotlib "*plot"* possibilita a criação de gráficos de linhas, que define o atributo/variável do eixo x horizontal e do eixo y vertical. Assim sendo segue uma maneira que possibilita a criação de gráficos com essa ferramenta:

```
fig, ax = plt.subplots()
ax.plot(x,y)
plt.show()
```

Para a demonstração em gráficos de diferentes variáveis basta utilizar o seguinte comando antes de solicitar a plotagem dos gráficos:

```
fig,axs = plt.subplots (x,y, constrained_layout=True)
```

O comando "constrained_layout" é necessário para ajustar os gráficos de forma limpa à sua figura, ajustando automaticamente legendas e barras de cores para que caibam na janela da figura enquanto preserva, da melhor maneira possível, o layout lógico solicitado pelo usuário.

4.1.4. Biblioteca kivy

A biblioteca kivy é um framework para desenvolvimento multiplataforma, ou seja, seu código nativo é executável em Linux, Windows, OSX, Android, iOS e *Raspberry Pi*. Aplicações desenvolvidas com essa biblioteca são executadas pela biblioteca gráfica OpenGL – *Open Graphics Library* que normalmente é executada em cima da placa gráfica, ou seja, o código desenvolvido com o kivy será executado pelos processadores da placa de vídeo.

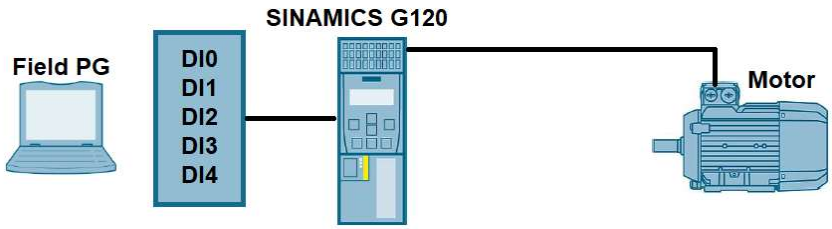
Para a construção das interfaces gráficas como telas, botões, campos de entrada de textos da tela é utilizado um segundo código, semelhante a linguagem Python, com a linguagem kvlang, essa separação do código principal em Python do código para a construção gráfica tem o objetivo de facilitar a adição ou remoção de *Widgets* (componentes visuais)

4.2. DESENVOLVIMENTO DA INTERFACE PARA COMISSIONAMENTO E MONITORAMENTO

Esse trabalho tem o foco de desenvolvimento de uma interface, através da linguagem Python e da utilização da biblioteca kivy, para realizar a restauração de

fábrica, o comissionamento, a parametrização de 3 (três) velocidades fixas, o monitoramento e posteriormente a criação de 3 (três) gráficos importantes para verificar o correto funcionamento da aplicação. Segue abaixo a topologia do sistema que será implementado:

Figura 10 – Topologia do Sistema que será simulado.



Fonte: Elaborado pelo autor.

Para a criação e simulação dessa interface foi utilizado um kit de bancada composto pelos seguintes equipamentos:

- ✓ Unidade de potência do SINAMICS G120 de 0,37 kW PM240;
- ✓ Unidade de controle do SINAMICS G120 CU240E-2;
- ✓ Um painel do operador básico do SINAMICS G120 BOP-2;
- ✓ Um motor da família SIMOTICS GP 1LA7 440V de 0,14 kW;
- ✓ Uma SIMATIC Field PG

Figura 11 – Kit de bancada utilizado para testes.



Fonte: Elaborado pelo autor.

Com a SIMATIC Field PG foi possível a instalação do Python e do IDE, do inglês *Integrated Development Environment*, ou seja, ambiente de desenvolvimento

integrado, para este trabalho foi utilizado o Pycharm, o código foi desenvolvido no sistema operacional Windows. Para a criação da interface iniciou-se baixando todas as bibliotecas necessárias, iniciou-se pela instalação no Pycharm da biblioteca Snap7 e Sinamics que possibilitam a comunicação com o drive SINAMICS G120 e a comunicação para leitura e escrita de parâmetros através do protocolo de comunicação S7, posteriormente instalou-se o Kivy, para a contrução da interface de monitoramento e por fim instalou-se a biblioteca Matplotilib e time para a criação dos gráficos com os dados sendo coletados a cada 01 (um) segundo.

A seguir são apresentados as principais importações que foram realizadas para o correto funcionamento do código, a biblioteca Snap7 é importada internamente pela biblioteca Sinamics, dessa maneira não há a necessidade de importar novamente no código principal.

```
# Comando para importar a biblioteca time necessária para dar as
# pausas necessárias no programa
      import time
# Comando para importar a biblioteca threading na qual executa Threads # que
são fluxos de programas que são executados em paralelo dentro de
# uma aplicação, ou seja, uma ramificação, para esse código é
# utilizado para esperar os 30 segundos do pop-up para o usuário
# aguardar o tempo necessário para a realização da restauração de
#                 fábrica                  dos                  parâmetros.
import                                                                threading
# Para criar a interface gráfica temos que criar um aplicativo, sendo # assim
necessário importar a classe App que é responsável pela
      # criação de aplicativos kivy e para iniciar seu aplicativo basta
      # chamar o método App.run().
      from                kivy.app                import                App
# O screenmanager é um gerenciador de telas, ou seja, permite a criação # de
várias telas para a aplicação, criando um menu, é necessário
      # importar ele e a classe Screen, referente a criação de cada telas.
      from kivy.uix.screenmanager import ScreenManager, Screen
      # Factory é utilizado para instanciar, referenciar qualquer classe ou
# método no Python, neste projeto é utilizado para fazer o pop-up
      # apresentar a mensagem de aguarde na tela do kivy.
      from              kivy.factory              import              Factory
# Para mitigar possíveis erros na etapa do código referente ao
# monitoramento foi utilizado o kivy.properties para garantir que os valores
#lidos e escritos assumam apenas as propriedades pertinentes # as variáveis.
from kivy.properties import NumericProperty,BooleanProperty, ListProperty,
ColorProperty,                                                  StringProperty
# para a construção dos gráficos é necessário importar a biblioteca
# matiplotlib
      import matplotlib.pyplot as plt

# Faz referência ao sinamics file (arquivo) e importando a biblioteca

      from sinamics import Sinamics
```

4.2.1. Classe Menu

Após importar todas as bibliotecas e classes necessárias para a construção da aplicação é necessário instanciar a classe Sinamics e demonstrar qual será o objeto que receberá os atributos da classe importada, para isso nomeamos o drive que será objeto da simulação aqui proposta, no código construído o mesmo foi nomeado como g120.

Prosseguindo a construção do código foi realizado a criação das classes necessárias, inicialmente criou-se a classe Menu que é responsável por possibilitar a navegação inicial de iniciar a conexão com o drive ou de sair do aplicativo como é mostrado na Figura 12.

Figura 12 – Tela intitulada Menu.



Figura: Elaborada pela autora.

Segue a criação da classe Menu no código principal em linguagem Python.

```
#todas as classes herdarão métodos da Classe Screen
class Menu(Screen):
    pass
```

No código principal, para esse trabalho foi intitulado main.py, a classe foi apenas definida pois a funcionalidade desta é apenas selecionar uma outra tela ou sair da interface, dessa forma a programação em si é referente à criação de Widgets,

assim sendo, é tratada dentro do código de linguagem kvlang, para esse trabalho esse arquivo foi intitulado G120.kv, nesse código é utilizado a classe do kivy BoxLayout responsável pela organização de telas e botões com textos, o comando canvas responsável pela definição da cor de fundo da tela e papel de parede, comandos de espaçamentos (padding e spacing), comandos de orientação e posição (orientation e pos), comando para inserir imagem na tela em formato *PNG* e texto nos botões (source e text) e de comandos de realização de função como on_release. Abaixo segue o código referente a classe Menu em linguagem kvlang.

```
      <Menu>:
    BoxLayout:
        #Faz a definição da cor do fundo de tela
        canvas:
            Color:
                rgba:1,1,1,1
            Rectangle:
                #para definir que o fundo cobrirá a tela toda
                size: self.size
                pos:self.pos
                source:'siemens_logo.png'
        # É definido a orientação vertical, caso queira a horizontal tal
linha de código não é necessaria.
        orientation:'vertical'
        # Essa propriedade adiciona uma margem ao layout
        padding: 200
        # Essa propriedade adiciona um espaço entre os widgets
        spacing: 50
        Button:
            # Defini o nome do widget botão:
            text: 'Iniciar a conexão com o drive'
            #Definição da função do botão, quando usamos root.current
indicamos qual é a tela que queremos como atual.
            on_release: app.root.current = 'conectador'
        Button:
            text: 'Sair'
            # Comando para sair do aplicativo.
            on_release: app.stop()
```

4.2.2. Classe Conectador

A próxima classe criada foi a Conectador com a função de realizar a conexão com o drive G120, essa classe utiliza o método *connect* da biblioteca Sinamics que permitirá a interface conectar com o drive SINAMICS G120 desde que o usuário apresente o IP (Internet Protocol) valido, dessa maneira é definida uma função intitulada conectar que solicita como entrada o número de IP do drive através do comando *self.ids.ip.text*, ou seja, o self referência ao widget da classe Conectador, o ids é responsável por acessar a sua identidade no código G120.kv dessa forma está sendo acessado a variável ip do código em linguagem kivy, valor referente ao IP do drive que será completado na interface pelo usuário e assim realiza-se a conexão com o mesmo.

```
class Conectador(Screen): # Classe que permite a criação de regras
para o widgets herdado do arquivo kivy.

  def conectar(self):
      if self.ids.ip.text != '':
          ip_g120 = self.ids.ip.text
          # Chama-se a função connect do método sinamics para
# conectar o inversor.
          g120.connect(ip_g120)
          print(ip_g120)
                self.ids.conectar.text = 'Conectado!'
      else:
          self.ids.conectar.text = 'Insira um IP válido!'
```

Para a construção da tela com os *widgets* foi necessário como na classe anterior definir todo layout no código G120.kv. Inicialmente definiu-se orientação vertical como uma hierarquia para concepção dos itens da tela, posteriormente definiu-se o primeiro *BoxLayout* dessa tela, responsável por escolher a cor de fundo da tela com o comando *canvas*, todas as telas seguirão o mesmo padrão.

```
  <Conectador>:
orientation:'vertical'
BoxLayout:
    canvas.before:
        Color:
            #rgb da Siemens = 0, 153, 153 (rgb vale de 0 a 256 e o rgba
```

```
0 a 1, fez então uma regra de três para inserir a cor desejada)
                rgba:0,0.59,0.59,1
            Rectangle:
                size: self.size
                pos: self.pos
        orientation: 'vertical'
```

O próximo item desta tela que foi atribuído foi a classe *ActionBar* esta é responsável por definir na tela uma barra superior com as opções sair da interface ou voltar para a tela anterior, todas as outras interfaces também possuem tal atributo.

```
        # Criação da barra superior do programa menu superior
        ActionBar:
            # O ActionView contém os widges especificos da ActionBar.
            ActionView:
                # Item para voltar ao menu, ele insere o ícone do próprio
kivy
                ActionPrevious:
                    title: 'Menu Principal'
                    on_release: app.root.current = 'menu'
                # Criando o Botão para sair do aplicativo
                ActionButton:
                    text: 'Sair'
                    on_release: app.stop()
```

O próximo passo foi acrescentar uma imagem de drives da família G120 na tela de conectar entre a barra superior e o campo de inserir o IP do drive para conexão, para isso foi criado um novo *BoxLayout* com o objetivo de inserir uma imagem em formato *PNG* através do comando *source*.

```
      BoxLayout:
            # Acrescentar uma imagem png que esteja na mesma pasta do
arquivo .kv
            Image:
                source: 'g120.png'
                # size_hint é uma função para definir o tamanho da tela
aqui no caso ela foi desabilitada para a vertical.
                size_hint_y: None
                # Especificar a altura em pixels
                height: 180
```

```
                # Permitir alterar o tamanho da imagem:
                allow_stretch: True
```

Como próxima etapa foi definido mais um *BoxLayout* referente ao campo para inserir o endereço de IP do drive, para isso foi utilizado o comando *Label* para inserir o texto solicitando inserir essa informação e o comando *TextInput* para o usuário inserir essa informação.

```
      BoxLayout:
            Label:
                text:'Conectar com o IP adress da Siemens:'
                size_hint_y: None
                height: 25
            TextInput:
                # O id é um identificador para acessar esse widget
                id:ip
                size_hint_y: None
                # self.minimum_height é uma função para adequar todos
widgets desse BoxLayout ao tamanho da tela
                height:self.minimum_height
```

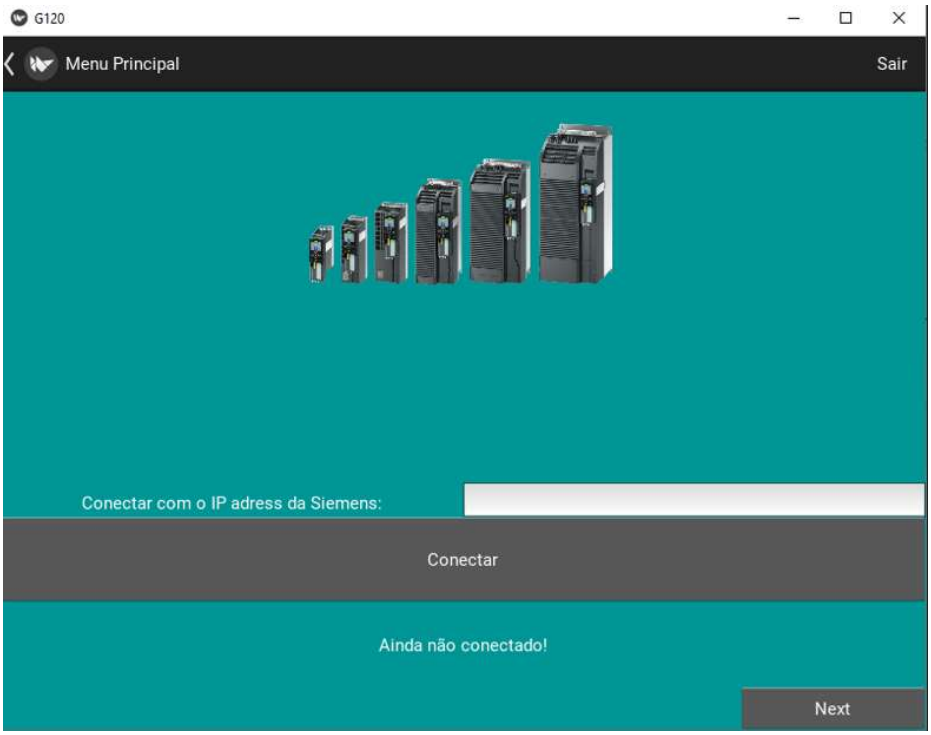
Por fim é criado o último *BoxLayout* dessa classe, se trata de um botão responsável por chamar a função conectar do programa em main.py e devolver se o drive está conectado ou não.

```
       BoxLayout:
            orientation: 'vertical'
            Button:
                text: 'Conectar'
                # root é uma referência ao método da classe Conectador
                on_release: root.conectar()
            Label:
                id: conectar
                text: 'Ainda não conectado!'
            Button:
                text: 'Next'
                size_hint: .2,.5
                pos_hint: {'right': 1, 'y': 0}
                # app é uma referência ao aplicativo que está rodando
(G120)
```

```
                # app.root faz uma referência ao gerenciador e chama o
método current
                on_release: app.root.current = 'comissionamento'
```

Após a apresentação da construção dos códigos é possível verificar a tela que estabelece a comunicação entre o drive e a interface criada, segue para esse propósito a Figura 13.

Figura 13 – Tela intitulada Conexão.



Fonte: Elaborada pela autora.

4.2.3. Classe Comissionamento

A classe comissionamento é dividida em duas partes principais a primeira é referente ao reset de fábrica, ou seja, a restauração dos parâmetros conforme parâmetro de fábrica, e a segunda referente ao comissionamento do drive, a parametrização segue conforme já explanado no tópico 3.1 deste trabalho.

O código a seguir refere-se à parte do programa em Python, intitulado main.py referente a primeira parte da classe de comissionamento, o reset de fábrica. É definido então uma função chamada *reset_fabrica* na qual utilizando a biblioteca sinamics

escreve no drive através do parâmetro *write_parameter* as alterações necessárias. Como esse processo de restauração para os dados padrão requer um tempo é inserido um pop-up na qual permanece na tela durante 30 (trinta) segundos através do comando *show_popup,* para a criação do pop-up, e da biblioteca *threading* que é utilizada para manter o *pop-up* rodando pelo tempo necessário impedindo o usuário de ter acesso à interface nesse período.

```
      class Comissionamento(Screen):
    # Realização do reset de fábrica
    def reset_fabrica(self, *args):
        print("Reset de fábrica P0010=30 P0970=1\n Espera 30 segundos para
resetar os parâmetros e retorno de P0010=1")
        # P0010 - Data type: Integer16 -> h (short)
        g120.write_parameter(10, 30,"h")
        p0010 = 30
        print(p0010)
        # P0970 - Data type: Unsigned16 -> H (Unsigned short)
        g120.write_parameter(970, 1,"H")
        p0970 = 1
        print(p0970)
        self.show_popup()
        # mythread é usado para trabalhar com tempo, ou seja para o popup
aparecer durante 30 segundos.
        mythread = threading.Thread(target=self.espera30)
        mythread.start()

    def show_popup(self):
        # Factory é utilizado para estanciar qualquer classe no python.
        self.pop_up = Factory.Popup(title='Aguarde 30 segundos estamos
resetando o drive!', size_hint=(None, None), size=(300, 180))
        self.pop_up.open()

    def espera30(self):
        thistime = time.time()
        while thistime + 30 > time.time():
            time.sleep(1)
        self.pop_up.dismiss()
        # Iniciar o comissionamento:
        # P0010 - Data type: Integer16 -> h (short)
        g120.write_parameter(10, 1, "h")
        p0010 = 1
        print(p0010)
        self.ids.info_reset.text = 'Reset de fábrica realizado com
sucesso!'
```

Para a segunda parte da classe Comissionamento é realizado a inserção dos dados do motor, os parâmetros referente ao comissionamento que devem ser alterados segue conforme mencionado no item 3.1 deste trabalho, dessa forma utiliza-se da biblioteca Sinamics para conseguir imputar esses dados no drive através do

comando *write_parameter,* após inserir o parâmetro P1900, referente a identificação dos dados do motor, é necessário aguardar pelo menos 10 segundos para que essa ação seja realizada, então é utilizado o comando *time.sleep* com o objetivo de pausar o fluxo do programa até que essa funcionalidade seja realizada, isso se repete para a alteração dos parâmetros P0015, referente a configuração da macro 12 e P3900, referente ao fim do comissionamento porem o tempo de pausa para eles será de pelo menos 03 (três) segundo, para essa interrupção do fluxo do programa também será utilizado o comando *time.sleep.* Segue o código referente a segunda etapa da classe comissionamento do programa *main.py*.

```
        # Iniciar o Comissionamento
    def inicio_comissionamento(self):
        print("Definindo as configurações de potência do inversor (kW ou hp
- 02:NEMA 60Hz line, SI units) - P0100 = 2")
        # P0100 - Data type: Integer16 -> h (short)
        g120.write_parameter(100, 2, "h")
        p0100 = 2
        print(p0100)
        print("Definindo os dados nominais do motor:")
        print("Definir o valor da tensão de alimentação (rede): P0210 =
Vrms")
        w_p210 = float(self.ids.v_alimentação.text)
        print(w_p210)
        # g120.write_parameter(210, w_p210, "f")
        print("Definir o valor de tensão nominal do motor : P0304 = V")
        w_p304 = float(self.ids.v_motor.text)
        print(w_p304)
        g120.write_parameter(304, w_p304, "f")
        print("Definir o valor da corrente nominal do motor : P0305 = A")
        w_p305 = float(self.ids.i_motor.text)
        print(w_p305)
        g120.write_parameter(305, w_p305, "f")
        print("Definir o valor da potência nominal do motor : P0307 = kW")
        w_p307 = float(self.ids.p_motor.text)
        print(w_p307)
        g120.write_parameter(307, w_p307, "f")
        print("Definir o valor da frequência do motor : P0310 = Hz")
        w_p310 = float(self.ids.f_motor.text)
        print(w_p310)
```

```
        g120.write_parameter(310, w_p310, "f")
        time.sleep(1)
        print("Definir o valor da velocidade nominal do motor : P0311 =
rpm")
        w_p311 = float(self.ids.vel_motor.text)
        print(w_p311)
        g120.write_parameter(311, w_p311, "f")
        time.sleep(1)
        print("Definir o valor da velocidade mínima do motor: P1080 = rpm")
        w_p1080 = float(self.ids.vmin_motor.text)
        print(w_p1080)
        g120.write_parameter(1080, w_p1080, "f")
        time.sleep(1)
        print("Definir o valor da velocidade máxima do motor: P1082 = rpm")
        w_p1082 = float(self.ids.vmax_motor.text)
        print(w_p1082)
        g120.write_parameter(1082, w_p1082, "f")
        time.sleep(1)
        print("Definir o tempo de rampa de aceleração do motor: P1120 = s")
        w_p1120 = float(self.ids.ta_motor.text)
        print(w_p1120)
        g120.write_parameter(1120, w_p1120, "f")
        time.sleep(1)
        print("Definir o tempo de rampa de desaceleração do motor: P1121 =
s")
        w_p1121 = float(self.ids.td_motor.text)
        print(w_p1121)
        g120.write_parameter(1121, w_p1121, "f")
        time.sleep(1)
        print("O modo de controle será definido como U/f característica
linear: P1300 =  0")
        p1300 = 0
        print(p1300)
        # P1300 - data type: Integer16 -> h (short)
        g120.write_parameter(1300, 0, "h")
        time.sleep(1)
        print("Identificar os dados do motor: P1900 = 2")
        p1900 = 2
        print(p1900)
        # P1900 - data type: Integer16 -> h (short)
```

```
        g120.write_parameter(1900, 2, "h")
        time.sleep(10)
        print("Definir a macro de funcionamento igual à 12: P0015 = 12")
        p0015 = 12
        print(p0015)
        # P0015  - Data type: Unsigned32 -> I
        g120.write_parameter(15, 12, "I")
        time.sleep(3)
        print("Fim de comissionamento : P3900 = 3")
        p3900 = 3
        print(p3900)
        # P3900 data type: Integer16 -> h (short)
        g120.write_parameter(3900, 3, "H")
        self.ids.fim_comissionamento.text = "Comissionamento realizado com
sucesso!"
```

Para a construção da tela com os widgets foi necessário, como na classe anterior, definir todo layout no código G120.kv, para essa definição iniciou-se definindo a cor padrão e a barra superior conforme o item anterior, posteriormente foi definido através da classe *BoxLayout* o botão intitulado “Reset de Fábrica” através da função Button foi possível criar um botão para quando for acionado rodar a função *reset_fabrica* do programa principal *main.py*, assim que finalizar a restauração padrão irá aparecer logo abaixo do botão a mensagem definida no código principal de “Reset de fábrica realizado com sucesso”, para isso é utilizado a função Label para a criação de um campo que assumira a variável *info_reset* do programa principal main.py após a função ser realizada. Segue abaixo o código referente a esse primeiro *BoxLayout* da classe Comissionamento.

```
  BoxLayout:
#Para controlar o tamanho do botão e do label.
size_hint_y: None
height: 75
orientation: 'vertical'
Button:
    text: 'Reset de Fábrica'
    on_release: root.reset_fabrica()
Label:
    id: info_reset
```

```
                text:''
```

Posteriormente é definido a segunda etapa da construção dos widgets da classe de comissionamento, logo abaixo do *Label* mencionado anteriormente é criado um segundo *Label* com o título Comissionamento, logo abaixo é então definido duas colunas com a classe *GridLayout* e a função *cols,* para cada coluna foi inserido 05 linhas para criar os campos de entrada dos dados do motor na qual o usuário final completará, para isso então foi criado um *BoxLayout* para cada parâmetro referente aos dados do motor onde criou-se um *Label* que dividia-se em dois, um com a função *text* na qual é inserido qual o parâmetro que está sendo solicitado para o usuário e sua unidade e logo a frente é inserido uma caixa de texto onde o usuário ira completar com o valor solicitado, para isso é utilizado o comando *TextInput,* esse código é repetido para os 10 (dez) parâmetros referente aos dados do motor e tempos de aceleração e desaceleração, segue abaixo o código referente aos campos de solicitação dos dados do motor para a realização do comissionamento.

```
      BoxLayout:
                size_hint_y: None
                height: 75
                Label:
                    text: 'Comissionamento:'
                    font_size: 20
            GridLayout:
                cols: 2
                BoxLayout:
                    Label:
                        # Define o tamanho da fonte do widget no caso do
Label:
                        font_size: 14
                        text: 'Inserir o valor de Tensão \n de Alimentação
[Vrms]:'
                    TextInput:
                        id: v_alimentação
                BoxLayout:
                    Label:
                        font_size: 14
                        text: 'Inserir o valor de Tensão \n Nominal do
Motor [V]:'
```

```
                TextInput:
                    id: v_motor
            BoxLayout:
                Label:
                    font_size: 14
                    text: 'Inserir o valor de Corrente \n Nominal do
Motor [A]:'
                TextInput:
                    id: i_motor
            BoxLayout:
                Label:
                    font_size: 14
                    text: 'Inserir o valor de Potência \n Nominal do
Motor [kW]:'
                TextInput:
                    id: p_motor
            BoxLayout:
                Label:
                    font_size: 14
                    text: 'Inserir o valor de Frequência \n do Motor
[Hz]:'
                TextInput:
                    id: f_motor
            BoxLayout:
                Label:
                    font_size: 14
                    text: 'Inserir o valor de Velocidade \n nominal do
Motor [rpm]:'
                TextInput:
                    id: vel_motor
            BoxLayout:
                Label:
                    font_size: 14
                    text: 'Inserir o valor de Velocidade \n Mínima do
Motor [rpm]:'
                TextInput:
                    id: vmin_motor
            BoxLayout:
                Label:
                    font_size: 14
```

```
                        text: 'Inserir o valor de Velocidade \n Máxima do
Motor [rpm]:'
                    TextInput:
                        id: vmax_motor
                BoxLayout:
                    Label:
                        font_size: 14
                        text: 'Inserir o valor da rampa \n de aceleração do
Motor [s]:'
                    TextInput:
                        id: ta_motor
                BoxLayout:
                    Label:
                        font_size: 14
                        text: 'Inserir o valor da rampa de \n desaceleração
do Motor [s]:'
                    TextInput:
                        id: td_motor
```

Logo após a construção dos campos de inserção dos valores dos parâmetros referentes aos dados do motor criou-se através da classe *BoxLayout* e da função *Button* o botão para identificar os dados do motor, através do comando *on_release* é possível fazer com que ao ser pressionado o botão criado esse realiza a função *inicio_comissionamento* do programa principal *main.py*, abaixo segue o código referente a criação do botão mencionado.

```
BoxLayout:
                size_hint_y: None
                height: 150
                orientation: 'vertical'
                Button:
                    text: 'Identificar os dados do motor'
                    on_release: root.inicio_comissionamento()
                Label:
                    id: fim_comissionamento
                    text: ''
```

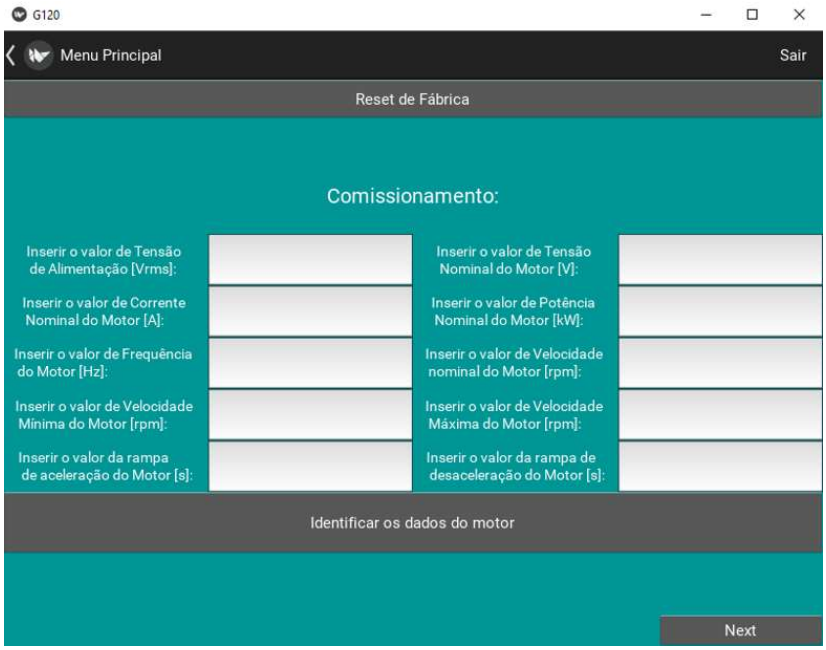
Por fim criou-se um último botão na interface de comissionamento que apenas realiza a troca de telas, para isso utiliza-se além do comando *on_release* o comando

app.root.current e indica o nome da classe referente a próxima tela. A seguir é apresentado o código referente a criação do botão “Next”.

```
Button:
    text: 'Next'
    size_hint: .2,.5
    pos_hint: {'right': 1, 'y': 0}
    on_release: app.root.current = 'drive'
```

Após entender um pouco mais sobre a construção dos códigos referente a classe “Comissionamento” é possível verificar a tela que realiza a restauração de fábrica e a identificação dos dados do motor, segue para esse propósito a Figura 14.

Figura 14 – Tela intitulada Comissionamento.



Fonte: Elaborada pela autora.

4.2.4. Classe Drives

A classe Drives é dividida em 03 (três) partes, a primeira refere-se à parametrização da aplicação de controle de velocidade através da parametrização das velocidades fixas como disposto no item 3.1 deste trabalho, inicialmente é definido

todas as propriedades das variáveis utilizadas, esse ponto é importante para mitigar possíveis erros restringindo-se as propriedades pertinentes a cada variável, para esse procedimento utilizou-se a classe *Properties* da biblioteca kivy. Foi atribuído então *NumericProperty* para as variáveis que assumiram valores com propriedades numéricas, *StringProperty* para as variáveis que assumiram características de *String*, *ColorProperty* para as variáveis que assumiram cores e *ListProperty* para as variáveis que serão listas. Segue o código, retirado do programa *main.py*, referente a atribuição de propriedades para cada variável.

```
class Drive(Screen):


    r0052 = NumericProperty()
    r0052_word = StringProperty()
    r2139 = NumericProperty()
    r2139_word = StringProperty()
    r722 = NumericProperty()
    r722_word = StringProperty()

    r1025 = NumericProperty()
    r00522 = NumericProperty()
    r21397 = NumericProperty()
    r21393 = NumericProperty()
    r7220 = NumericProperty()
    r7221 = NumericProperty()
    r7222 = NumericProperty()
    r0022 = NumericProperty()
    r0027 = NumericProperty()
    r0026 = NumericProperty()
    r0052_hex = NumericProperty()
    r2139_hex = NumericProperty()
    r722_hex = NumericProperty()

    r00522_color = ColorProperty()
    r21397_color = ColorProperty()
    r21393_color = ColorProperty()
    r7220_color = ColorProperty()
    r7221_color = ColorProperty()
    r7222_color = ColorProperty()
```

```
status_monitorar = BooleanProperty()

tempo = ListProperty()
velocidade = ListProperty()
corrente = ListProperty()
tensao = ListProperty()
```

Após a definição da propriedade que as variáveis podem assumir realiza-se a definição “init”, esse é um método na qual sempre que for chamado a classe, no caso Drive, ele será iniciado antes de qualquer função. A função *super* permite não só manter os atributos de inicialização da classe Drive, mas também manter os atributos da classe Screen, classe referenciada, que é necessária para manter a interface em execução, quando inicializar a classe essa atribuirá a modificação nas variáveis de acordo com o que a função *reset_monitorar* propõe.

```
    def __init__(self,*args,**kwargs):
  super(Drive, self).__init__(*args,**kwargs)
  self.reset_monitorar()
```

Segue as mudanças que a função *reset_monitorar* propõe.

```
def reset_monitorar(self):
  self.r00522_color = [1, 1, 1, 1]
  self.r21397_color = [1, 1, 1, 1]
  self.r21393_color = [1, 1, 1, 1]
  self.r7220_color = [1, 1, 1, 1]
  self.r7221_color = [1, 1, 1, 1]
  self.r7222_color = [1, 1, 1, 1]
  self.r1025 = 0.1
  self.r00522 = 0.1
  self.r21397 = 0.1
  self.r21393 = 0.1
  self.r7220 = 0.1
  self.r7221 = 0.1
  self.r7222 = 0.1
  self.status_monitorar = False
  self.tempo = []
  self.r0022 = 0.1
  self.velocidade = []
  self.r0027 = 0.1
  self.corrente = []
```

```
        self.r0026 = 0.1
        self.tensao = []
        self.r0052 = 0.1
        self.r0052_word = '0'
        self.r2139 = 0.1
        self.r2139_word = '0'
        self.r722 = 0.1
        self.r722_word = '0'
        self.r0052_hex = 0
        self.r2139_hex = 0
        self.r722_hex = 0
```

Após a definição das variáveis e a definição do método de inicialização foi então definida a função intitulada *iniciar_app* para parametrizar a aplicação do controle de velocidades através de velocidades fixas, com isso foi utilizado novamente a biblioteca Sinamics com o comando *write_parameter*, para o bom funcionamento da aplicação foi inserido também alguns *time.sleep* para atribuir uma pausa no fluxo do código para escrever os parâmetros no drive de forma mais organizada, mitigando possíveis erros. Para as velocidades que serão completadas pelo usuário na interface criada utilizou-se novamente da função *self.ids* sendo assim o comando utilizado foi *self.ids."nome da variável que a velocidade inserida pelo usuário irá assumir".text* assim sendo o self faz referência ao widget atual, o ids é responsável por acessar a identidade atribuída no código G120.kv, ou seja, a variável que foi atribuída a cada velocidade que o usuário irá preencher. Segue dessa forma o código referente à parametrização da aplicação construído no programa principal *main.py*.

```
    def iniciar_app(self):
        # Parametrizar o controle de velocidade no drive:
        print("Configurar que será dado o comando ON/OFF (P0840) pelas
entradas digitais configuradas com velocidades fixas (r1025.0)")
        # P0840 - Data type: U32/Binary -> "I/B"
        g120.write_parameter(840, '1025.0', "I/B")
        time.sleep(1)
        print("Define que a variação de velocidade será dado por
velocidades fixa P1000 = 3")
        # P1000 - Data type: Integer16 -> h (short - interger)
        g120.write_parameter(1000, 3, "h")
        time.sleep(1)
```

```
        print("Definição dos valores das 03 velocidade fixa (P1001, P1002, P1003)")
        # P1001/ P1002/ P1003 - Data type: FloatingPoint32 -> f (float)refere-se aos valores que serão inseridos pelo usuário para as 03 velocidades fixas, no P1001 é atribuído a velocidade mais baixa, no P1002 é atribuída a velocidade média e no P1003 será atribuída a velocidade mais alta. O comando ids faz referência ao código .kv lincando valores que seão inseridos na tela ao programa principal.
        w_p1001 = float(self.ids.v1.text)
        print(w_p1001)
        time.sleep(1)
        g120.write_parameter(1001, w_p1001, "f")
        w_p1002 = float(self.ids.v2.text)
        print(w_p1002)
        time.sleep(1)
        g120.write_parameter(1002, w_p1002, "f")
        w_p1003 = float(self.ids.v3.text)
        print(w_p1003)
        g120.write_parameter(1003, w_p1003, "f")
        time.sleep(1)
        print("Definir que o drive fará a seleção da velocidade fixa pelas entradas digitais")
        print("P1020 = r7221.0 / P1021 = r722.1 / P1022 = r722.2")
        # r722.0/ r222.1/ r222.2 - Data type: Unsigned32 -> I (Unsigned int)
        # P1020/ P1021/ P1022 - Data type: U32/Binary -> I/B
        g120.write_parameter(1020, '722.0', "I/B")
        g120.write_parameter(1021, '722.1', "I/B")
        g120.write_parameter(1022, '722.2', "I/B")
        time.sleep(1)
        print("Desabilitar o parametro de reversão de sentido habilitado pela macro 12 P1113=0")
        # P1113 - Data type: U32/Binary -> I/B
        g120.write_parameter(1113, 0 , "I/B")
        time.sleep(1)
        print("Definir que o reconhecimento de falha será dado pela DI3 P2103=r722.3")
        # P2103 - Data type: U32/Binary -> I/B
        g120.write_parameter(2103, '722.3', "I/B")
        time.sleep(1)
```

```
        print("Definir que a parada de emergência será dada pela DI4
P0848=r722.4")
        # P0848 - Data Type: U32/Binary -> I/B
        g120.write_parameter(848, '722.4', "I/B")
        print("Definir o tempo da rampa de desaceleração da parada de
emergência P1135 = Tempo de parada (s)")
        # P1135 - Data type: FloatingPoint32 - f (float)
        g120.write_parameter(1135, 3, "f")
        self.ids.fim_parametriza.text = "Parametrização Realizada com
Sucesso!"
```

Posteriormente a realização da parametrização do drive é necessário definir as funções que irão iniciar ou finalizar o monitoramento, a função responsável por atualizar a interface de acordo com o funcionamento do drive. O código a seguir define apenas as funções responsáveis por atualizar a variável *status_monitorar* em verdadeiro (“*True*”) para monitorar ou falso (“*False*”) para parar de monitorar, sendo assim responsável por chamar ou interromper a função de atualização da interface de monitoramento, no código nomeamos *update*.

```
            def monitorar (self):
            # Atualizar os valores na tela durante o monitoramento.
            self.status_monitorar = True
                  # Utilização da biblioteca threading responsável por
executar     Threads, ou seja, fluxo de programas em paralelo dentro de uma
aplicação, ramificação, aqui é responsável por chamar a função update
enquanto a variável self.status_monitorar é verdadeiro.
                  threading.Thread(target=self.update).start()

    # Função responsável por pausar a função de atualização (update) do
status do drive presentes na interface de monitoramento.
          def parar_monitorar(self):
            self.status_monitorar = False
```

Após a definição das funções de monitorar e parar de monitorar é necessário a criação da função de atualização, intitulada *update* no programa *main.py*, inicialmente verifica se a função *self.status_monitorar* encontra-se em “True” para essa verificação é utilizado o comando *if* caso essa condição seja verdadeira inicia-se a leitura dos parâmetros que serão monitorados, essa leitura é realizada pelo comando *read_parameter* , comando herdado pela biblioteca Sinamics. O primeiro parâmetro monitorado é o r0052 que trás a informação sobre o a palavra de status, porem para o monitoramento iremos necessitar apenas do bit 02 (dois), no código *main.py* a *variável foi nomeada como self.r00522*, que quando em nível logico alto demonstra se

o drive está em operação, na interface enquanto essa variável estiver retornando 01 (um) o led intitulado *run* deverá assumir a cor verde.

Como foi possível perceber esse parâmetro ao ser lido deverá retornar um valor hexadecimal, para realizar o monitoramento dele então é necessário tratá-lo, após realizar a leitura e armazenamento deste valor em uma variável intitulada *self."nome do parâmetro"* é necessário transforma-lo em binário através da função *bin*, esse valor é inserido em uma variável com propriedade de string que no código desenvolvido foi intitulado *self."nome do parâmetro"_word* após essa transformação é necessário acrescentar zeros no lado mais significativo (lado esquerdo) até que o número apresente o comprimento total disponível, para isso é verificado com o comando de condição *if* se esse acréscimo é realmente necessário. Após a verificação e se necessário o acréscimo de zeros o número binário é invertido pois o drive apresenta esse valor da forma oposta como a linguagem Python entende, assim é utilizado o comando [:: -1] responsável por inverter esse valor e dessa forma é possível ler o bit que será monitorado.

Segue abaixo o código referente ao monitoramento do bit 02 (dois) da palavra de status r0052, responsável por retornar se o drive encontra-se ou não em operação.

```
    def update(self,*args):
      for i in range (1,200):
          if self.status_monitorar == True:
              # Leitura de Parâmetro da operação r0052 bit2 = 1 em
operação
              # self.r0052 = input("Insira uma suposição de valor
hexadecimal para o r0052: ")
              # r0052.2 - Data type: Unsigned16 -> H (Unsigned Short)
              self.r0052 = g120.read_parameter(52,"H")
              print(self.r0052)
              self.r0052_word = bin(self.r0052)[2:]# Necessário o [2:]
pois o número binário no Python sempre inicia com 0b.
              print(self.r0052_word)
              if len(self.r0052_word) < 16:
                  self.r0052_word = ('0' * (16 - len(self.r0052_word)) +
self.r0052_word) # acrescenta-se 0s até completar o total de bits, no caso
16, acrescenta-se sempre no lado mais significativo (lado esquerdo)
              print(self.r0052_word)
              self.r0052_word = self.r0052_word[::-1] # posteriormente
após completar o número de bits é necessário inverter o número para
```

```
conseguir selecionar o bit que quer-se ler corretamente.
                print(self.r0052_word)
                self.r00522 = self.r0052_word[2]
                print(self.r00522)
```

Após a verificação do parâmetro r0052, foi verificado o parâmetro r2139 bit 03 (três) e bit 07 (sete) que respectivamente representam se o drive está em falha e alarme e o parâmetro r0722 bit 00 (zero), bit 01 (um) e bit 02 (dois) que representam respectivamente a entrada digital 01 a entrada digital 02 e a entrada digital 03, para o processo de construção do código em linguagem Python seguiu o mesmo princípio do anterior como pode-se ver a seguir.

```
      # Leitura do Parâmetro de falha/alarme r2139 bit 7 = 1 em alarme
                # self.r21397 = int(input("Insira uma suposição para o
valor de r2139.7 (0 sem Alarme ou 1 com Alarme)"))
                # r2139.7 - Data type: Unsigned16 -> H (Unsigned Short)
                self.r2139 = g120.read_parameter(2139 , "H")
                # self.r2139 = input("insira o valor em hexadecimal para
r2139: ")
                # self.r2139_hex = (self.r2139, 16)
                self.r2139_word = bin(self.r2139)[2:]
                # self.r2139_word = bin(int(self.r2139, 16))[2:]
                print(self.r2139_word)
                if len(self.r2139_word) < 16:
                    self.r2139_word = ('0' * (16 - len(self.r2139_word)) +
self.r2139_word)
                print(self.r2139_word)
                self.r2139_word = self.r2139_word[::-1]
                print(self.r2139_word)
                self.r21397 = self.r2139_word[7]
                print(self.r21397)
                # Leitura do Parâmetro de falha/alarme r2139 bit 3 = 1 em
falha
                # r2139.3 - Data type: Unsigned16 -> H (Unsigned Short)
                self.r21393 = self.r2139_word[3]
                print(self.r21393)
                # r722.0/ r222.1/ r222.2 - Data type: Unsigned32 -> I
(Unsigned int)
                # Leitura do Parâmetro das velocidades r722.0 , r722.1,
r722.2
```

```
                time.sleep(1)
                self.r722 = g120.read_parameter(722, "I")
                # self.r722 = (input("insira o valor em hexadecimal para
r0722: "))
                # self.r722_hex = (self.r722, 16)
                # print(self.r722_hex)
                # self.r0052_word = bin(int(r0052, scale))[2:]
                self.r722_word = bin(self.r722)[2:]
                print(self.r722_word)
                if len(self.r722_word) < 32: # o parâmetro referente ao
r722 possui 32 bit de acordo com o manual do SINAMICS G120.
                    self.r722_word = ('0' * (32 - len(self.r722_word)) +
self.r722_word)
                print(self.r722_word)
                self.r722_word = self.r722_word[::-1]
                self.r7220 = self.r722_word[0]
                print(self.r7220)
                self.r7221 = self.r722_word[1]
                print(self.r7221)
                self.r7222 = self.r722_word[2]
                print(self.r7222)
```

Após monitorar os valores no código em linguagem python é necessário definir cada cor que aparecera nos leds da interface de monitoramento quando cada uma das possibilidades lidas ocorrer. Para a definição das cores em cada led é importante verificar cada uma das possibilidades de retorno em cada bit dos parâmetros monitorados, assim sendo são utilizados os comandos de condição *if* e *else* pois para cada bit lido do inversor é possível assumir apenas 02 (dois) valores ou nível logico alto ou nível logico baixo, no código principal em linguagem Python é definido uma lista, para cada uma das variáveis que assumiram os valores dos bits monitorados, com quatro elementos numéricos, podendo assumir entre zero e um, esses valores posteriormente no código kivy serão interpretados como cores após a utilização da classe *Color* como será possível entender mais a frente.

```
                    # definição de cores para o led run da interface de
monitoramento, ou seja, ira aparecer no led da interface de monitoramento
intitulado RUN a cor verde quando o bit 02 da palavra de controle, no
código intitulado self.r00522, assumir nível logico alto e a cor branca
quando for nível logico baixo, ou seja quando o motor não estiver em
operação.
```

```
                if self.r00522 == 1:
                    self.r00522_color = [0,1,0,1]
                else:
                    self.r00522_color = [1,1,1,1]
                      # definição de cores para o led Alarme da interface
de monitoramento, se o bit 07 (sete) do parâmetro r2139 estiver em nível
lógico alto, ou seja, a variável self.r21397 assumir 01 (um) o led Alarme
deverá assumir a cor laranja, já se ele estiver em nível lógico baixo
deverá assumir a cor branca.
                if self.r21397 == 1:
                    self.r21397_color = [1,0.5,0,1]
                else:
                    self.r21397_color = [1,1,1,1]
                      # definição de cores para o led Falha da interface de
monitoramento, se o bit 03 (três) do parâmetro r2139 estiver em nível
lógico alto, ou seja, a variável self.r21393 assumir 01 (um) o led Falha
deverá assumir a cor vermelha, já se ele estiver em nível lógico baixo
deverá assumir a cor branca.
                if self.r21393 == 1:
                    self.r21393_color = [1,0,0,1]
                else:
                    self.r21393_color = [1,1,1,1]
                      # definição de cores para o led v1 da interface de
monitoramento, se o bit 00 (zero) do parâmetro r0722 estiver em nível
lógico alto, ou seja, a variável self.r7220 assumir 01 (um) o led v1 deverá
assumir a cor verde, já se ele estiver em nível lógico baixo deverá assumir
a cor branca.
                if self.r7220 == 1:
                    self.r7220_color = [0,1,0,1]
                else:
                    self.r7220_color = [1,1,1,1]
                      # definição de cores para o led v2 da interface de
monitoramento, se o bit 01 (um) do parâmetro r0722 estiver em nível lógico
alto, ou seja, a variável self.r7221 assumir 01 (um) o led v2 deverá
assumir a cor verde, já se ele estiver em nível lógico baixo deverá assumir
a cor branca.
                if self.r7221 == 1:
                    self.r7221_color = [0,1,0,1]
                else:
                    self.r7221_color = [1,1,1,1]
```

```
                    # definição de cores para o led v3 da interface de
monitoramento, se o bit 02 (dois) do parâmetro r0722 estiver em nível
lógico alto, ou seja, a variável self.r7222 assumir 01 (um) o led v3 deverá
assumir a cor verde, já se ele estiver em nível lógico baixo deverá assumir
a cor branca.
                if self.r7222 == 1:
                    self.r7222_color = [0,1,0,1]
                else:
                    self.r7222_color = [1,1,1,1]
```

Para finalizar a classe Drives do programa principal, *main.py*, é elaborado o código referente a criação dos gráficos, inicialmente cria-se as listas com a leitura dos parâmetros que serão objeto da criação dos gráficos, para isso utiliza-se o comando *self.tempo.append(i-1)*, esse comando é responsável por adicionar um valor das variáveis monitoradas na lista a cada incremento de tempo de 01 (um) segundo. As variáveis que serão monitoradas graficamente são a velocidade atual do motor, parâmetro r0022, corrente atual do motor, parâmetro r0027 e tensão de link DC, parâmetro r0026. Para a leitura dos parâmetros é utilizado o comando *read.parameter*. Para a criação dos gráficos é definida uma função intitulada *graficos* que utiliza a função plot para criar os gráficos, em uma mesma imagem será apresentado os três gráficos. Segue abaixo o código referente a criação dos gráficos.

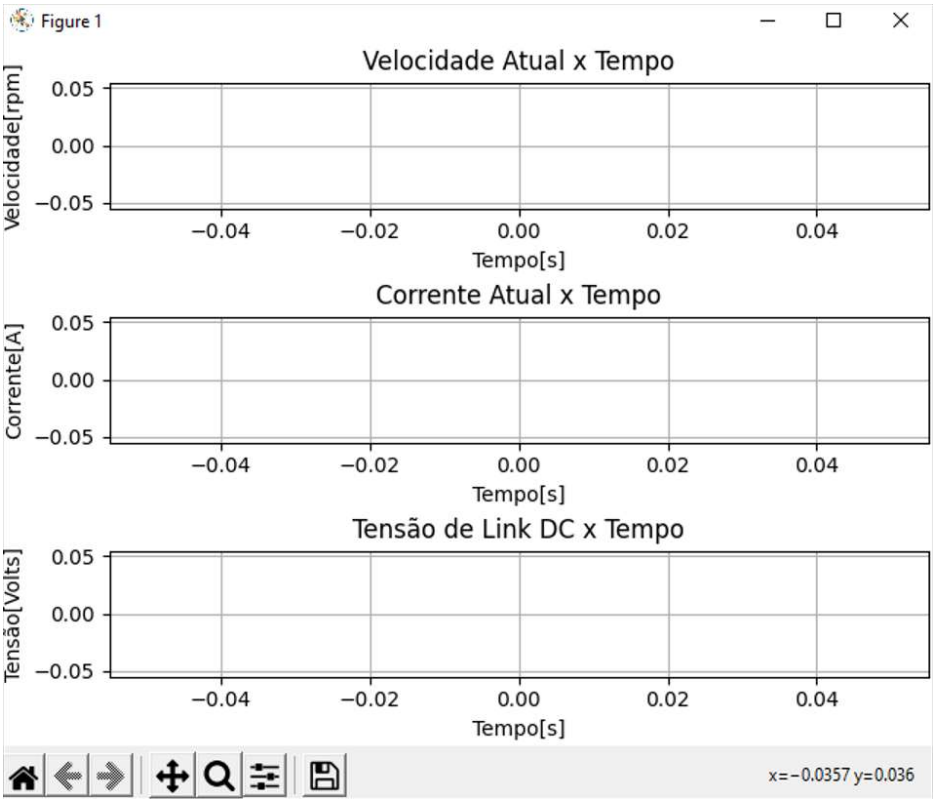
```
      # Monitoramento das variáveis objetos da construção dos gráficos
                self.tempo.append(i-1)# a função append para cada
incremento de tempo adiciona um valor das variáveis abaixo lidas do drive
em uma lista.
                print(self.tempo)
                # r0022 - Data type: FloatingPoint32 - f (float)
                self.r0022 = float(g120.read_parameter(22,"f"))
                print("r0022 - leitura da velocidade atual do motor")
                # self.r0022 = float(input("insira a suposta velocidade do
motor:"))
                self.velocidade.append(self.r0022)
                print(self.velocidade)
                # r0027 - Data type: FloatingPoint32 - f (float)
                self.r0027 = float(g120.read_parameter(27,"f"))
                print("r0027 - leitura da corrente atual do motor")
                # self.r0027 = float(input("insira a suposta corrente do
motor:"))
                self.corrente.append(self.r0027)
```

```
                print(self.corrente)
                # r0026 - Data type: FloatingPoint32 - f (float)
                self.r0026 = float(g120.read_parameter(26,"f"))
                print("r0026 - leitura da tensão de link DC")
                # self.r0026 = float(input("insira a suposta tensão de link
DC:"))
                self.tensao.append(self.r0026)
                print(self.tensao)


    # Função para definir gráficos:
    def graficos(self):
        fig, axs = plt.subplots(3, constrained_layout=True) # Comando para
definir os 03 (três) gráficos em uma mesma imagem.
        axs[0].plot(self.tempo, self.velocidade)
        axs[0].set_title('Velocidade Atual x Tempo')
        axs[0].set(xlabel='Tempo[s]', ylabel='Velocidade[rpm]')
        axs[0].grid()
        axs[1].plot(self.tempo, self.corrente, 'tab:orange')
        axs[1].set_title('Corrente Atual x Tempo')
        axs[1].set(xlabel='Tempo[s]', ylabel='Corrente[A]')
        axs[1].grid()
        axs[2].plot(self.tempo, self.tensao, 'tab:green')
        axs[2].set_title('Tensão de Link DC x Tempo')
        axs[2].set(xlabel='Tempo[s]', ylabel='Tensão[Volts]')
        axs[2].grid()
        plt.show()
```

Segue na Figura 15 a tela que serão criados os gráficos referente às 03 (três) variáveis monitoradas.

Figura 15 – Tela para a criação dos gráficos.



Fonte: Elaborada pela autora.

Para a construção da tela com os widgets foi necessário, como na classe anterior, definir todo layout no código G120.kv, para essa definição iniciou-se definindo a cor padrão e a barra superior conforme o item anterior, posteriormente foi definido as caixas de texto na qual o usuário iria inserir as velocidades e os leds que iriam mostrar o monitoramento ativo, assim para cada velocidade foi criado um *BoxLayout* e definiu-se 02 (dois) *Label* e um *TextInput,* o primeiro *Label* utiliza a função Color para atribuir a cor já definida no código principal e a função *Ellipse* para a criação do formato redondo do led, no segundo *Label* foi inserido de qual velocidade se trata através do comando text e no *TextInput* criou-se um espaço para o usuário inserir qual será o valor da velocidade que será parametrizado. Abaixo segue o código referente a velocidade 01 (um) para as demais velocidades segue o mesmo procedimento de criação.

```
  BoxLayout:
Label:
    id: status_v1
```

```
            canvas:
                Color:
                    rgba: root.r7220_color
                Ellipse:
                    size: [30,30]
                    pos: [125,470]
        Label:
            text:'v1'
        TextInput:
            id:v1
```

Após a construção do campo de entrada das velocidades fixas é criado um botão com a função *Button*, intitulado “Iniciar a Parametrização da Aplicação”, na qual ao ser pressionado é responsável por rodar a função *iniciar_app* definida no programa principal em linguagem Python, após finalizar a parametrização na interface de monitoramento irá aparecer o texto “Parametrização Realizada com Sucesso”, na qual a variável fim_parametriza assume no programa main.py. Abaixo é possível verificar a parametrização no G120.kv referente a esse botão.

```
BoxLayout:
    padding: 50
    spacing: 25
    orientation: 'vertical'
    Button:
        text: 'Iniciar a Parametrização da Aplicação'
        on_release: root.iniciar_app()
    Label:
        id: fim_parametriza
        text: ''
```

Com o botão da parametrização pronto na interface de monitoramento é possível construir os leds que irão mostrar o status do drive, assim como os leds das velocidades é definido um *BoxLayout* com dois *Label* o primeiro referente a construção do led utilizou a função *Color* e assim atrelou-se à cor já atribuída de acordo com a condição verificada no código *main.py* utilizou-se também a função *Ellipse* para atrelar o formato redondo ao led. O segundo *Label* foi utilizado para atribuir o nome ao led. Esse procedimento repetiu-se para os três leds o intitulado “Run”, o intitulado “Falha” e o intitulado “Alarme”, dessa forma segue o código referente a construção do led “Run”, o código completo será anexado ao fim do trabalho.

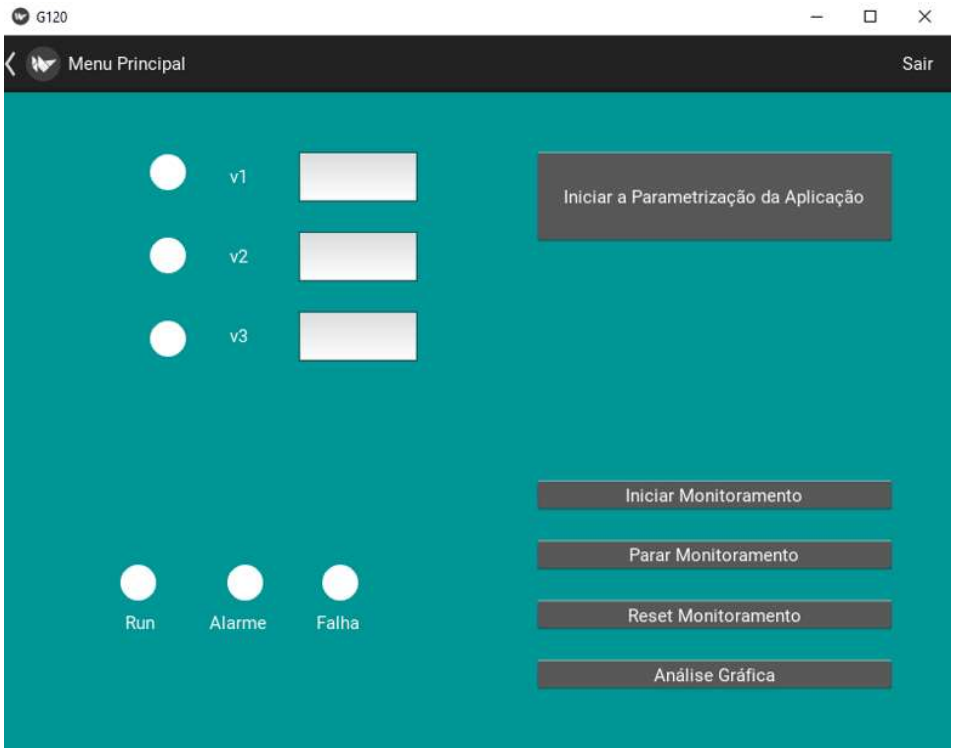
```
  BoxLayout:
padding: 75
BoxLayout:
    orientation:'vertical'
    Label:
        id: run
        canvas:
            Color:
                rgba: root.r00522_color
            Ellipse:
                size: [30,30]
                pos: [100,125]
    Label:
        text: 'Run'
```

Por fim foi construído os botões “Iniciar Monitoramento”, “Parar Monitoramento”, “Reset monitoramento” e “Análise Gráfica” todos atrelados as suas respectivas funções no programa principal em linguagem Python, *main.py*. Segue abaixo a construção dos botões utilizando a função *Button*.

```
  Button:
    text: 'Iniciar Monitoramento'
    on_release: root.monitorar()
Button:
    text: 'Parar Monitoramento'
    on_release: root.parar_monitorar()
Button:
    text: 'Reset Monitoramento'
    on_release: root.reset_monitorar()
Button:
    text: 'Análise Gráfica'
    on_release: root.graficos()
```

Após a conclusão do código referente a interface segue na Figura 16 a interface de monitoramento.

Figura 16 – Tela de monitoramento.



Fonte: Elaborada pela autora.

4.2.5. Classe Gerenciador e Classe G120

Quando se trata de mais de uma interface de trabalho é necessário estabelecer uma hierarquia entre as telas, ou seja, uma ordem de apresentação, quem realiza essa função é o gerenciador de telas, para esse trabalho é intitulado Gerenciador. Abaixo segue o código em linguagem Python onde a classe é apenas criada.

```
    # Gerência as telas:
class Gerenciador(ScreenManager):
    pass
```

Abaixo segue a configuração da classe Gerenciador em linguagem kivy, pode-se notar que ela é responsável por gerenciar as demais telas ou seja as telas apresentadas anteriormente estão todas sobre o domínio da classe gerenciador.

```
    #O Gerenciador é utilizado para gerenciar e organizar as demais telas
do aplicativo.
<Gerenciador>:
    Menu:
        name: 'menu'
```

```
    Conectador:
        name: 'conectador'
    Comissionamento:
        name: 'comissionamento'
    Drive:
        name: 'drive'
```

O código em linguagem kivy responsável por criar as funcionalidades dos widgets foi então criado com o nome G120.kv, dessa forma é criado a classe G120 que levara a atribuição *App* de aplicação kivy no código *main.py*, segue abaixo a definição da classe G120.

```
#Chama o arquivo G120 no kivy para assim criar uma interface gráfica:
class G120(App):
    # Definindo o método que inicializa e constrói o aplicativo.
    def build(self):
        return Gerenciador()
# Método responsável por rodar o aplicativo.
if __name__ == '__main__':
    G120().run()
```

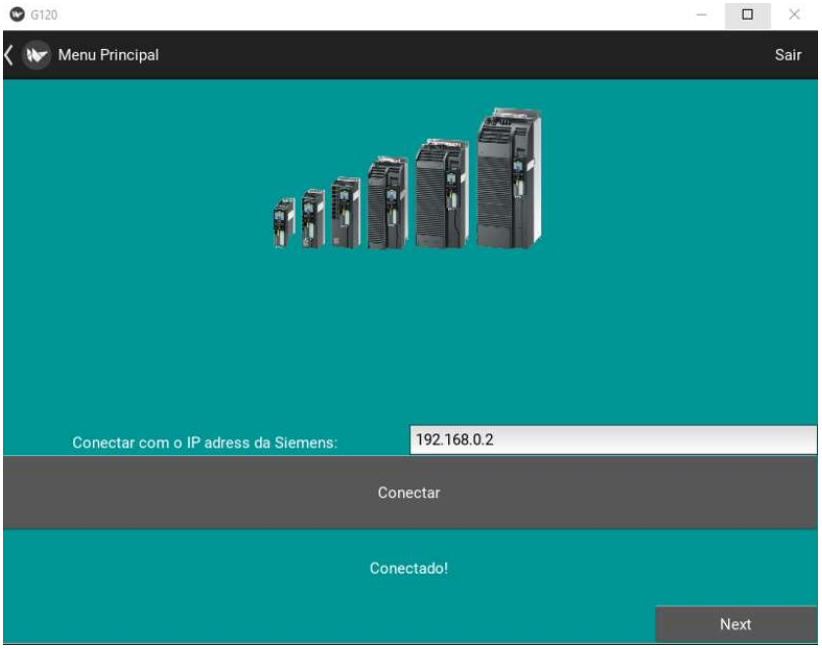
Essa classe então é responsável por criar o arquivo kivy de mesmo nome e define a construção através da função *build* da interface da aplicação retornando a classe “Gerenciador” e pôr fim a aplicação é executada com o comando *run*.

5. RESULTADOS E DISCUSSÃO

Este capítulo consiste na simulação da metodologia descrita no Capítulo 4, essa simulação tem o objetivo de verificar se a interface construída é capaz de comissionar corretamente o kit de bancada, bem como parametrizar o mesmo com a aplicação de controle de velocidade através da velocidade fixa e realizar a análise gráfica da aplicação parametrizada.

Iniciou-se estabelecendo a conexão entre a interface criada através da linguagem de programação Python e o drive SINAMICS G120, o endereço de IP (*Internet Protocol*) utilizado foi 192.168.0.2 e a interface conseguiu realizar a comunicação com drive.

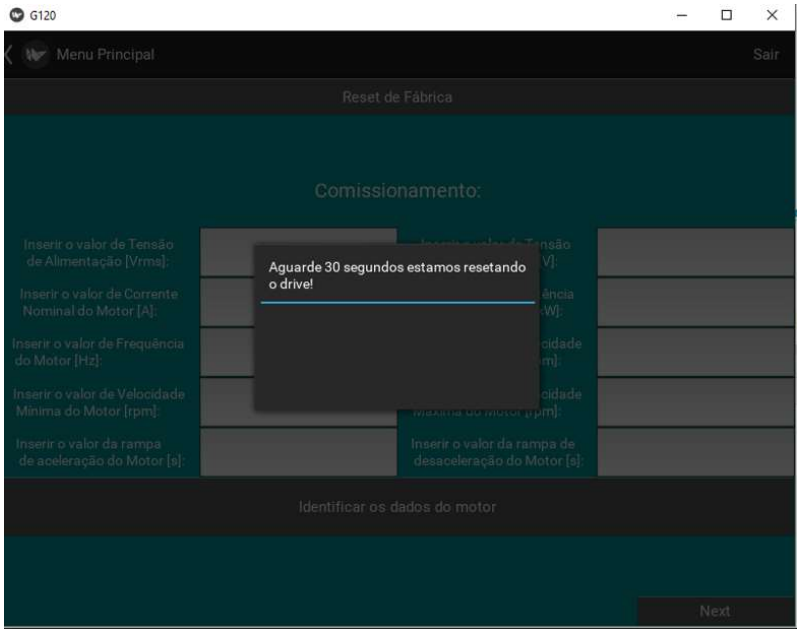
Figura 17 – Simulação conexão da Interface com o SINAMICS G120.



Fonte: Elaborado pela autora.

Seguindo o botão “Next” da interface foi possível seguir para a tela de restauração padrão dos parâmetros e comissionamento. Ao clicar no botão “Reset de Fábrica” como esperado o pop-up subiu na tela impedindo o acesso ao restante da tela como segue na Figura 18.

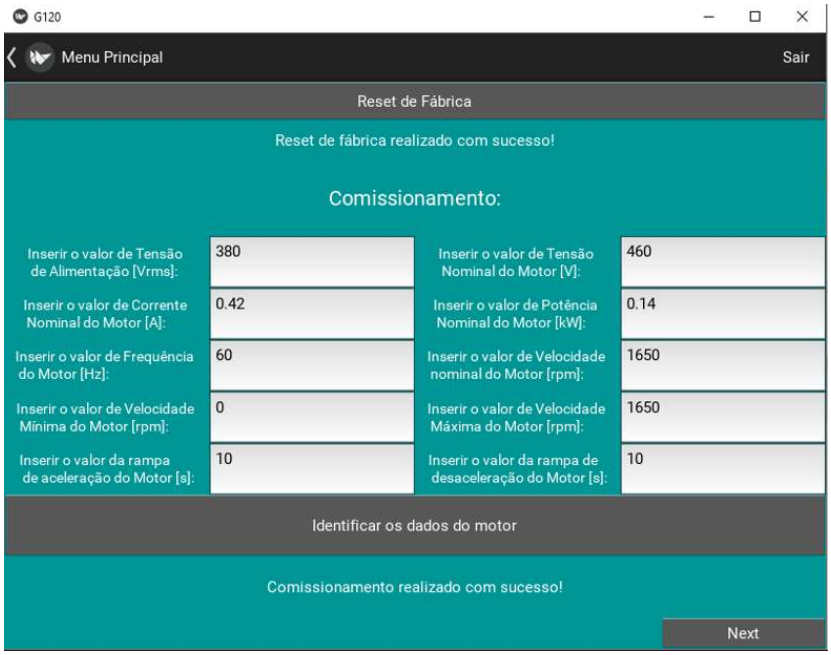
Figura 18 – Simulação da realização do Reset de fábrica.



Fonte: Elaborado pela autora.

Após os 30 segundos foi possível notar no drive que o reset de fábrica ocorreu como esperado. Posteriormente, quando o pop-up sumiu da tela foi possível realizar o comissionamento do drive, para o mesmo reconhecer os dados do motor, foi então inserido os dados nominais do motor. Na Figura 19 pode-se verificar que o comissionamento foi realizado como esperado e então o drive estaria pronto para receber a parametrização da aplicação de controle de velocidade simples.

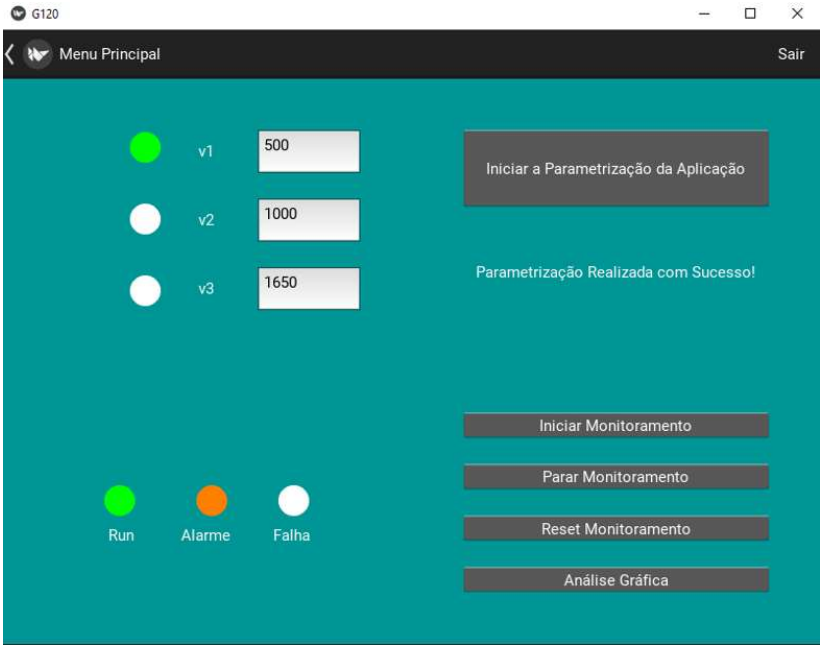
Figura 19 – Simulação da realização do Comissionamento.



Fonte: Elaborado pela Autora.

Ao selecionar o botão “Next” a interface irá direcionar o usuário para a tele de Parametrização e Monitoramento, antes de iniciar a parametrização da aplicação foi inserido os valores das 03 (três) velocidades fixas, para a velocidade V1 parametrizou-se como a mais baixa em 500 rpm, para a velocidade V2 atribuiu um valor intermediário de 1000 rpm e para a velocidade V3 configurou como a velocidade mais alta 1650 rpm, ou seja, a nominal do motor. Posteriormente clicou-se no botão “Iniciar a Parametrização da Aplicação” e após alguns segundos apareceu a mensagem de parametrização realizada com sucesso, assim iniciou o monitoramento clicando no botão “Iniciar Monitoramento”, acionou-se inicialmente a entrada digital zero e verificou o drive rodando a 500 rpm bem como a interface trocou o led de V1 para a cor verde o led de “*Run*” para a cor verde e curiosamente o led de Alarme para a cor laranja como pode-se observar na Figura 20.

Figura 20 – Simulação da realização do Monitoramento V1.

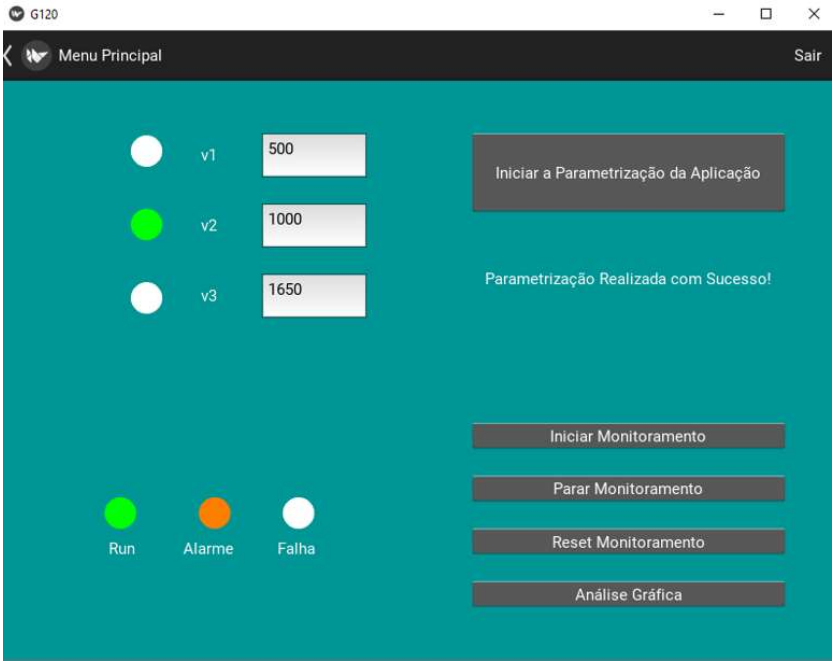


Fonte: Elaborado pela Autora.

O led "*Run*" e o da velocidade V1 apresentando a cor verde já era esperado, porem o led de Alarme apresentando a cor laranja não era esperado, porem ao verificar qual Alarme que o drive estava apresentando pode-se notar que era o A8526 pelo manual do SINAMICS G120 pode-se notar que era um Alarme de PROFINET sem conexão cíclica, ou seja não há nenhuma conexão com um controlador PROFINET sendo assim não era nenhum alarme preocupante ou que indicava que a parametrização não tinha ocorrido como o esperado.

Posteriormente simulou-se a entrada digital um em nível lógico alto e pode-se notar que o led de V1 voltou para a cor branca e que o led V2 assumiu a cor verde como pode-se notar na Figura 21.

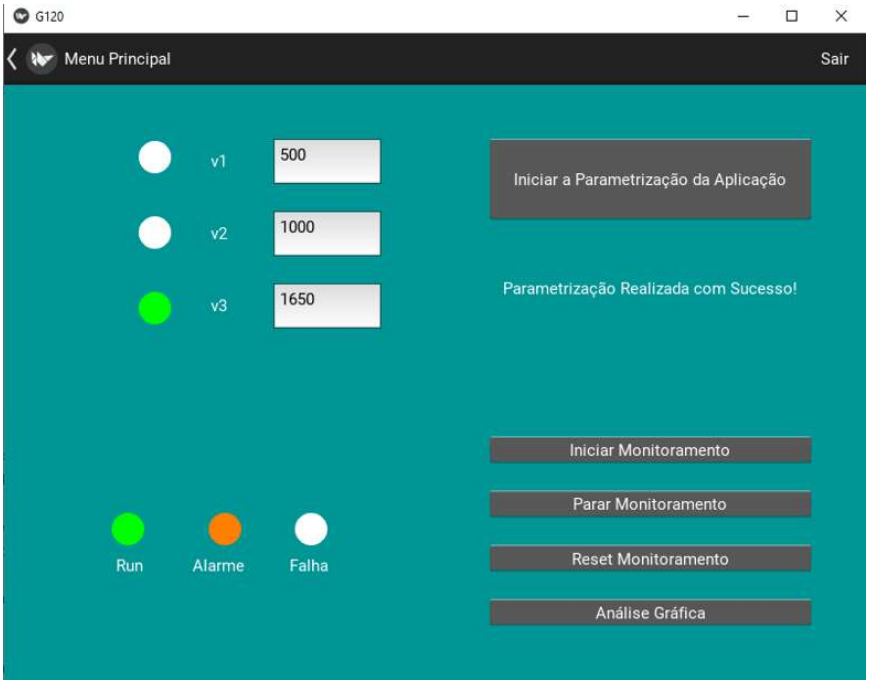
Figura 21 – Simulação da realização do Monitoramento V2.



Fonte: Elaborado pela Autora.

Posteriormente simulou-se a entrada digital dois em nível lógico alto e pode-se notar que o led de V2 voltou para a cor branca e que o led V3 assumiu a cor verde como pode-se verificar na Figura 22.

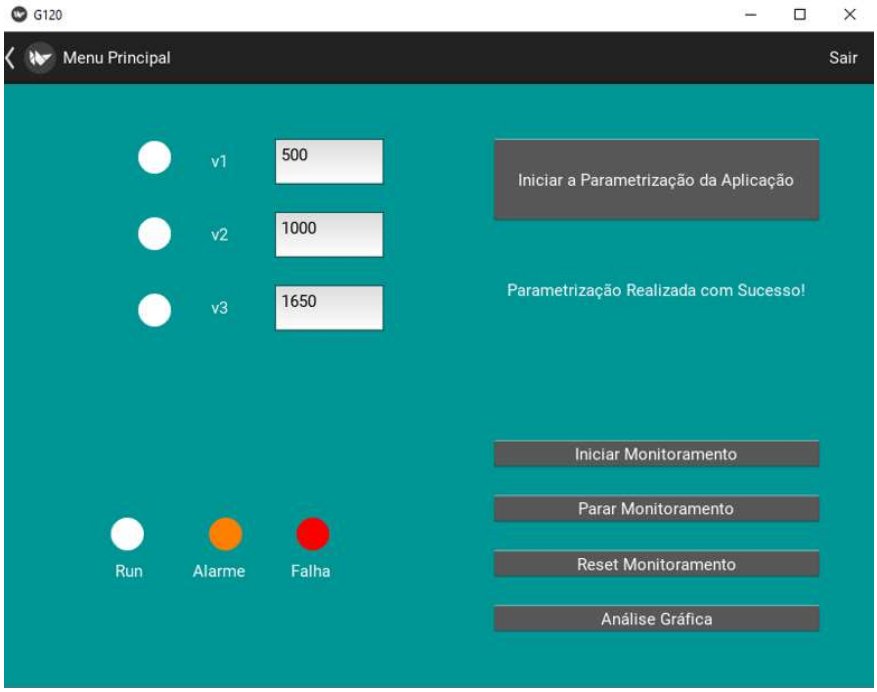
Figura 22 – Simulação da realização do Monitoramento V3.



Fonte: Elaborado pela Autora.

Posteriormente atribuiu uma falha externa (P2106 – External Fault) na entrada digital 05 (cinco) (r722.5) apenas com o objetivo de testar o led da falha para verificar se estava monitorando corretamente, após a simulação pode verificar que o monitoramento da falha ocorreu como esperado como segue na Figura 23.

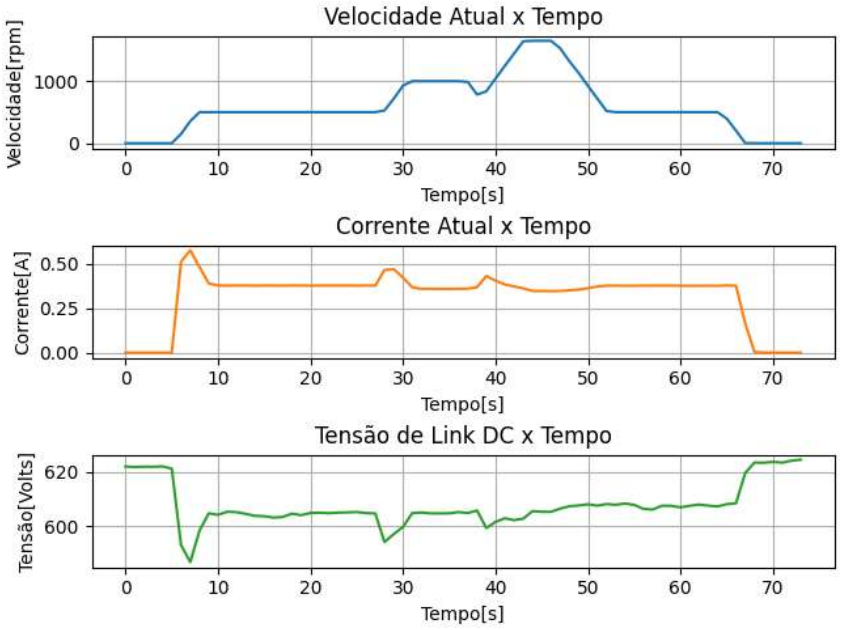
Figura 23 – Simulação da realização do Monitoramento da Falha ativa.



Fonte: Elaborada pela Autora.

Após as simulações das Interfaces passou-se para análise gráfica, inicialmente simulou uma aplicação rodando inicialmente em velocidade baixa, posteriormente em velocidade intermediária e por fim em velocidade alta, após alguns segundos retornou-se para a velocidade mínima e desligou-se. Segue na Figura 24 o gráfico que foi construído pela interface elaborada.

Figura 24 – Análise gráfica I.



Fonte: Elaborada pela autora.

A primeira análise que se pode realizar é verificando o comportamento da velocidade comparado com a demanda de *setpoint*, o primeiro *setpoint* solicitado foi referente a velocidade de 500 rpm, pode-se verificar que o drive respeitou, posteriormente a velocidade intermediária inserida foi de 1000 rpm o drive também respeitou e por fim a velocidade mais alta de 1650 rpm pode-se notar que também foi respeitada, antes de desligar o drive recebeu o *setpoint* da velocidade mínima que novamente foi respeitado e posteriormente desligou-se.

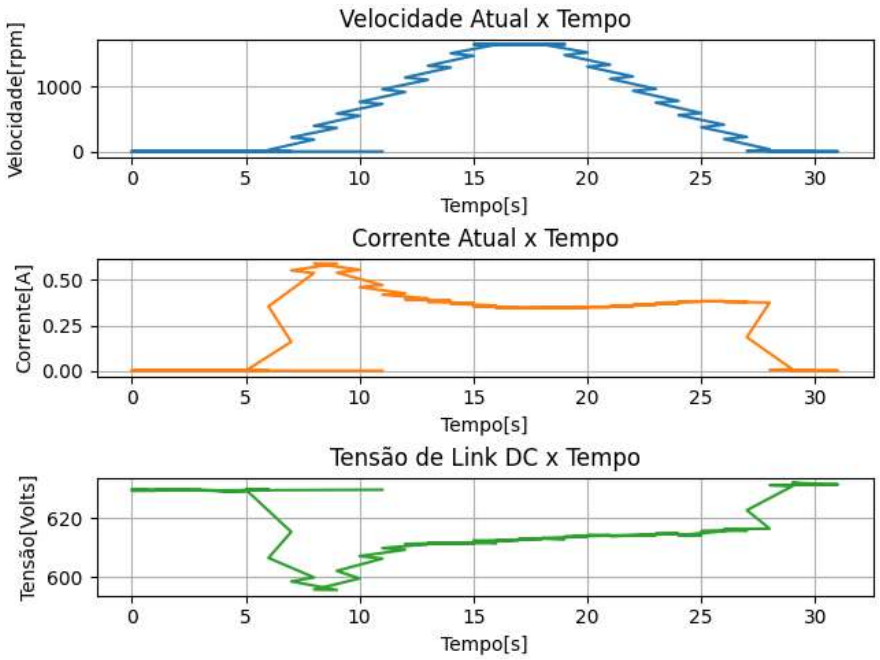
Outra análise que se pode fazer é referente a curva de corrente, verificando assim se a aplicação está trabalhando com sobrecarga ou não, pela curva de corrente pode-se perceber que que a aplicação está bem dimensionada em relação a aplicação e que não trabalha-se em sobrecarga uma vez que a corrente permanece abaixo da nominal, nota-se pequenas oscilações de corrente apenas na variação de velocidade o que é normal, a curva de corrente não apresentou nenhuma anomalia significativa.

Por fim tem-se a análise pertinente é sobre a tensão do banco de capacitores, foi observado que os valores encontrados foram satisfatórios. Pode-se notar que o valor médio da tensão do link DC (605 Volts), dividido pelo fator de retificação (1,35)

se aproxima da tensão de alimentação do drive chegou-se em um valor de 448 Volts, o valor da tensão de alimentação é de 480 volts.

Posteriormente a outra simulação que foi realizada é referente as duas paradas parametrizadas para a aplicação, foi parametrizada a parada de emergência, que costuma-se chamar OFF3, com uma rampa de desaceleração de 3 (três) segundos e a parada normal, que costuma-se chamar de OFF1, com uma rampa de desaceleração de 10 (dez) segundos.

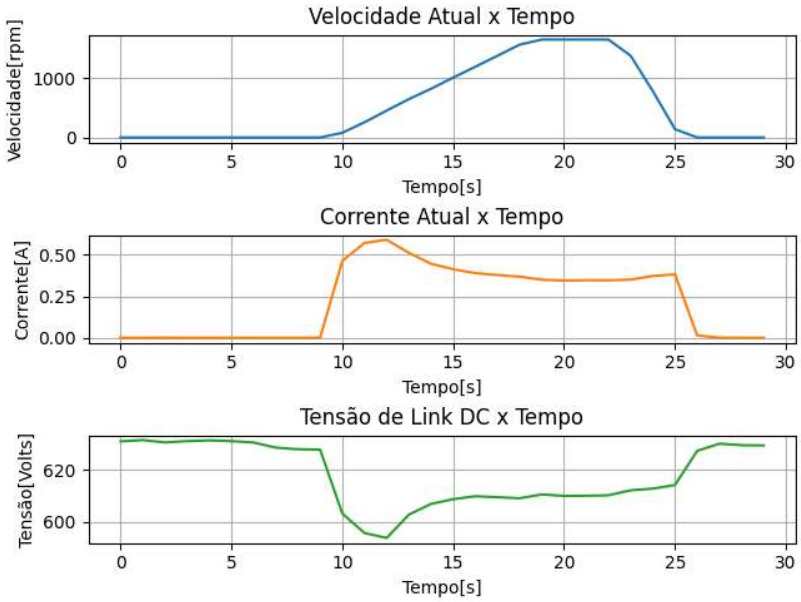
Figura 25 – Análise gráfica II Parada OFF1.



Fonte: Elaborada pela autora.

Pela Figura 25 pode-se notar que inicialmente colocou-se a velocidade nominal de 1650 rpm a rampa de aceleração está respeitando o que foi parametrizado em 10 (dez) segundos assim como a rampa de desaceleração pela parada OFF1 que está em torno de 10 segundos também.

Figura 26 – Análise gráfica III Parada OFF3.



Fonte: Elaborada pela autora.

Pela Figura 26 pode-se notar que inicialmente colocou-se a velocidade nominal de 1650 rpm a rampa de aceleração está respeitando o que foi parametrizado em 10 (dez) segundos assim como a rampa de desaceleração pela parada OFF3 que está em torno de 03 (três) segundos.

6. CONCLUSÃO

Pode-se concluir que o objetivo central deste trabalho consistia em elaborar uma interface de comunicação, comissionamento, parametrização de um controle de velocidade simples e monitoramento construída com linguagem de programação em Python para o drive da família SINAMICS G120 com conceitos básicos da linguagem de python e principalmente herdando métodos, funções e classes de bibliotecas existentes. Inicialmente foi desenvolvida a interface para estabelecer a comunicação com o drive posteriormente foi criada mais uma interface destinada a restauração dos parâmetros conforme padrão de fábrica e comissionamento, em seguida criou-se a interface destinada a parametrização do controle de velocidade com velocidades fixas e o monitoramento e por fim o a interface da construção dos gráficos.

As Interfaces criadas foram testadas e pode-se verificar que trabalharam em harmonia com o drive, registrando os principais status da operação em tempo real como drive em operação, alarme ou falha, além da exibição em gráficos de comportamento de parâmetros primordiais para qualquer aplicação, através da análise gráfica pode-se verificar que o drive estava respeitando setpoint de velocidade definido pelo usuário na interface de parametrização e monitoramento, pode-se verificar também que a aplicação não estava trabalhando com sobrecarga uma vez que a corrente estava dentro da nominal, pode-se notar que os valores de link DC estão dentro dos satisfatórios. Outro ponto que se pode verificar é que a aplicação respeitava os tempos de desaceleração principalmente da parada de emergência dentro dos 03 segundos estipulados.

Isto posto pode-se concluir que os objetivos propostos nesse trabalho foram atingidos, podendo ser comprovado através da simulação com o kit de bancada e dos testes que foram realizados, melhorias como o envio dessas informações para a nuvem que posteriormente consiga ser acessados por uma inteligência artificial para gerar diagnósticos mais completos seria uma solução mais próxima do conceito de indústria 4.0.

REFERÊNCIAS BIBLIOGRÁFICAS

BARBI, I. Eletrônica de Potência. Florianópolis: Ed. Do Autor, 6° Edição, 2006.

CARVALHO, Rogério Cláudio. Biblioteca Kivy. eXcript, 21, Jun. 2018. Disponível em: < http://excript.com/python/kivy.html>. Acesso em: 29, Ago.2021.

CONCEIÇÃO, M. MONTEIRO, S. Mesmo sob escassez de recursos, há muito a ser feito para estimular a inovação empresarial. **Blog da Conjuntura Econômica,** 30, março. 2021. Disponível em: https://ibre.fgv.br/blog-da-conjuntura-economica/artigos/mesmo-sob-escassez-de-recursos-ha-muito-ser-feito-para. Acesso em: 29, Jun. 2021.

FRANCHI, C. Inversores de Frequência: Teoria e Aplicações. Editora Érica; 2ª edição, 2009.

GODINHO, Renan Carlos Ferreira. PROJETO DE AUTOMAÇÃO INDUSTRIAL INTEGRADO: REPRESENTAÇÃO PRÁTICA DOS DIVERSOS NÍVEIS DE AUTOMAÇÃO ATRAVÉS DE DISPOSITIVOS E REDES DE COMUNICAÇÃO INDUSTRIAIS. 2017. 93 p. Monografia (Graduação em Engenharia Elétrica)- Faculdade de Engenharia de Ilha Solteira, Universidade Estadual Paulista 'Júlio de Mesquita Filho', Ilha Solteira - SP, 2017

KIVY (Brasil) (ed.). Kivy Documentation: release 2.1.0.dev0. 2020. Disponível em: <https://buildmedia.readthedocs.org/media/pdf/kivy/latest/kivy.pdf.> Acesso em: 29, Ago. 2021.

MATPLOTLIB: VISUALIZATION WITH PYTHON. Matplotlib version 3.4.3. 2012. Disponível em: < https://matplotlib.org/>. Acesso em 29 Ago. 2021

OLIVEIRA, Vinicius de Souza Lima. PROTOCOLO DE COMUNICAÇÃO PROFINET PARA REDES DE AUTOMAÇÃO. 2016. 77 p. Monografia (Graduação em Engenharia Elétrica)- Universidade Federal do Rio de Janeiro, Rio de Janeiro-RJ, 2016.

O TUTORIAL DE PYTHON. Python 3.9.7 Documentation, 2021. Disponível em: < https://docs.python.org/pt-br/3/tutorial/index.html >. Acesso em 03 Ago. 2021.

SCHWAB, K. A Quarta Revolução Industrial. Editora Edipro; 1ª edição, 2018.

SIEMENS (AG) (ed.). SINAMICS Inverters for Single-Axis Drives Built-In Units. Siemens AG, Digital Factory Division, Motion Control. 2018. Disponível em:<file:///C:/Users/z004399k/Downloads/motion-control-drives-D31-1-complete-English-06-2018%20(20).pdf >. Acesso em 07, Ago. 2021.

SIEMENS AG 2009-2021 (Brasil) (ed.). Simple speed control with SINAMICS G120. Industry Online Support Product Support. 2019. Disponível em: <https://support.industry.siemens.com/cs/br/pt/view/109759700/en> Acesso em: 03 Ago.2021

SIEMENS AG 2009-2021 (Brasil) (ed.). What properties, advantages and special features does the S7 protocol offer? Industry Online Support Product Support. 2019. Disponível em: <https://support.industry.siemens.com/cs/document/26483647/what-properties-advantages-and-special-features-does-the-s7-protocol-offer-?dti=0&dl=en&lc=pt-BR.> Acesso em: 29 Ago. 2021.

SIEMENS AG 2014 - 2017. (Brasil) (ed.). List Manual: SINAMICS G120 Control Units CU240B-2/CU240E-2. 09/2020. Siemens AG, Division Digital Industries, P.O. Box 48 48 90026 Nuremberg Germany. Disponível em: <file:///C:/Users/z004399k/Downloads/G120_CU240BE 2_List_Manual_0920_en_US.pdf> Acesso em: 22 Ago. 2021.

SIEMENS 2021 (Brasil). Um guia prático sobre a Indústria 4.0. 2021. Disponível em: https://new.siemens.com/br/pt/empresa/stories/industria/industria-4-0.html. Acesso em: 29, Jun. 2021.

STEP7 OPEN SOURCE ETHERNET COMMUNICATION SUITE. Snap7. Disponível em: <http://snap7.sourceforge.net/> . Acesso em 03 Ago. 2021

WEG. Catálogo Geral de Motores WEG, Disponível em: < https://www.hd.ind.br/PDF/manual_TECNICO-motores.pdf>. Acesso em 29, Jun. 2021.

ANEXOS

Anexo I – Código em linguagem Python intitulado main.py

```
# Comando para importar a biblioteca time necessária para dar as
# pausas necessárias no programa
import time
# Comando para importar a biblioteca threading na qual executa Threads #
que são fluxos de programas que são executados em paralelo dentro de
# uma aplicação, ou seja, uma ramificação, para esse código é
# utilizado para esperar os 30 segundos do pop-up para o usuário
# aguardar o tempo necessário para a realização da restauração de
# fábrica dos parâmetros.
import threading
# Para criar a interface gráfica temos que criar um App, sendo assim
# necessário importar o App.
from kivy.app import App
# O Screen Manager é um gerenciador de telas
from kivy.uix.screenmanager import ScreenManager, Screen
from kivy.factory import Factory
# teste de bibliotecas
from kivy.properties import NumericProperty,BooleanProperty, ListProperty,
ColorProperty, StringProperty
# para a construção dos gráficos é necessário importar a biblioteca
# matiplotlib
import matplotlib.pyplot as plt
# Faz referência ao sinamics file (arquivo) e importando a biblioteca #
Sinamics
from sinamics import Sinamics

g120 = Sinamics()


# Aplicativo desenvolvido para conectar, comissionar, parametrizar,
monitorar e analisar variáveis do G120:


#todas as classes herdarão métodos da Classe Screen
class Menu(Screen):
    pass


class Conectador(Screen): # Classe que permite a criação de regras para o
widgets herdado do arquivo kivy.

    def conectar(self):
        if self.ids.ip.text != '':
            ip_g120 = self.ids.ip.text
            # Estou chamando a função connect do método sinamics para
conectar o inversor.
            g120.connect(ip_g120)
            print(ip_g120)
            self.ids.conectar.text = 'Conectado!'
        else:
            self.ids.conectar.text = 'Insira um IP válido!'


class Comissionamento(Screen):
    # Realização do reset de fábrica
```

```
    def reset_fabrica(self, *args):
        print("Reset de fábrica P0010=30 P0970=1\n Espera 30 segundos para
resetar os parâmetros e retorno de P0010=1")
        # P0010 - Data type: Integer16 -> h (short)
        g120.write_parameter(10, 30,"h")
        p0010 = 30
        print(p0010)
        # P0970 - Data type: Unsigned16 -> H (Unsigned short)
        g120.write_parameter(970, 1,"H")
        p0970 = 1
        print(p0970)
        self.show_popup()
        # mythread é usado para trabalhar com tempo, ou seja para o popup
aparecer durante 30 segundos.
        mythread = threading.Thread(target=self.espera30)
        mythread.start()

    def show_popup(self):
        # Factory é utilizado para estanciar qualquer classe no python.
        self.pop_up = Factory.Popup(title='Aguarde 30 segundos estamos
resetando o drive!', size_hint=(None, None), size=(300, 180))
        self.pop_up.open()

    def espera30(self):
        thistime = time.time()
        while thistime + 30 > time.time():
            time.sleep(1)
        self.pop_up.dismiss()
        # Iniciar o comissionamento:
        # P0010 - Data type: Integer16 -> h (short)
        g120.write_parameter(10, 1, "h")
        p0010 = 1
        print(p0010)
        self.ids.info_reset.text = 'Reset de fábrica realizado com
sucesso!'

    # Iniciar o Comissionamento
    def inicio_comissionamento(self):
        print("Definindo as configurações de potência do inversor (kW ou hp
- 02:NEMA 60Hz line, SI units) - P0100 = 2")
        # P0100 - Data type: Integer16 -> h (short)
        g120.write_parameter(100, 2, "h")
        p0100 = 2
        print(p0100)
        print("Definindo os dados nominais do motor:")
        print("Definir o valor da tensão de alimentação (rede): P0210 =
Vrms")
        w_p210 = float(self.ids.v_alimentação.text)
        print(w_p210)
        # g120.write_parameter(210, w_p210, "f")
        print("Definir o valor de tensão nominal do motor : P0304 = V")
        w_p304 = float(self.ids.v_motor.text)
        print(w_p304)
        g120.write_parameter(304, w_p304, "f")
        print("Definir o valor da corrente nominal do motor : P0305 = A")
        w_p305 = float(self.ids.i_motor.text)
        print(w_p305)
        g120.write_parameter(305, w_p305, "f")
        print("Definir o valor da potência nominal do motor : P0307 = kW")
        w_p307 = float(self.ids.p_motor.text)
        print(w_p307)
```

```
        g120.write_parameter(307, w_p307, "f")
        print("Definir o valor da frequência do motor : P0310 = Hz")
        w_p310 = float(self.ids.f_motor.text)
        print(w_p310)
        g120.write_parameter(310, w_p310, "f")
        time.sleep(1)
        print("Definir o valor da velocidade nominal do motor : P0311 =
rpm")
        w_p311 = float(self.ids.vel_motor.text)
        print(w_p311)
        g120.write_parameter(311, w_p311, "f")
        time.sleep(1)
        print("Definir o valor da velocidade mínima do motor: P1080 = rpm")
        w_p1080 = float(self.ids.vmin_motor.text)
        print(w_p1080)
        g120.write_parameter(1080, w_p1080, "f")
        time.sleep(1)
        print("Definir o valor da velocidade máxima do motor: P1082 = rpm")
        w_p1082 = float(self.ids.vmax_motor.text)
        print(w_p1082)
        g120.write_parameter(1082, w_p1082, "f")
        time.sleep(1)
        print("Definir o tempo de rampa de aceleração do motor: P1120 = s")
        w_p1120 = float(self.ids.ta_motor.text)
        print(w_p1120)
        g120.write_parameter(1120, w_p1120, "f")
        time.sleep(1)
        print("Definir o tempo de rampa de desaceleração do motor: P1121 =
s")
        w_p1121 = float(self.ids.td_motor.text)
        print(w_p1121)
        g120.write_parameter(1121, w_p1121, "f")
        time.sleep(1)
        print("O modo de controle será definido como U/f característica
linear: P1300 =  0")
        p1300 = 0
        print(p1300)
        # P1300 - data type: Integer16 -> h (short)
        g120.write_parameter(1300, 0, "h")
        time.sleep(1)
        print("Identificar os dados do motor: P1900 = 2")
        p1900 = 2
        print(p1900)
        # P1900 - data type: Integer16 -> h (short)
        g120.write_parameter(1900, 2, "h")
        time.sleep(10)
        print("Definir a macro de funcionamento igual à 12: P0015 = 12")
        p0015 = 12
        print(p0015)
        # P0015  - Data type: Unsigned32 -> I
        g120.write_parameter(15, 12, "I")
        time.sleep(3)
        print("Fim de comissionamento : P3900 = 3")
        p3900 = 3
        print(p3900)
        # P3900 data type: Integer16 -> h (short)
        g120.write_parameter(3900, 3, "H")
        self.ids.fim_comissionamento.text = "Comissionamento realizado com
sucesso!"
```

```
class Drive(Screen):


    r0052 = NumericProperty()
    r0052_word = StringProperty()
    r2139 = NumericProperty()
    r2139_word = StringProperty()
    r722 = NumericProperty()
    r722_word = StringProperty()

    r1025 = NumericProperty()
    r00522 = NumericProperty()
    r21397 = NumericProperty()
    r21393 = NumericProperty()
    r7220 = NumericProperty()
    r7221 = NumericProperty()
    r7222 = NumericProperty()
    r0022 = NumericProperty()
    r0027 = NumericProperty()
    r0026 = NumericProperty()
    r0052_hex = NumericProperty()
    r2139_hex = NumericProperty()
    r722_hex = NumericProperty()

    r00522_color = ColorProperty()
    r21397_color = ColorProperty()
    r21393_color = ColorProperty()
    r7220_color = ColorProperty()
    r7221_color = ColorProperty()
    r7222_color = ColorProperty()

    status_monitorar = BooleanProperty()

    tempo = ListProperty()
    velocidade = ListProperty()
    corrente = ListProperty()
    tensao = ListProperty()

    def __init__(self,*args,**kwargs):
        super(Drive, self).__init__(*args,**kwargs)
        self.reset_monitorar()

    def iniciar_app(self):
        # Parametrizar o controle de velocidade no drive:
        print("Configurar que será dado o comando ON/OFF (P0840) pelas
entradas digitais configuradas com velocidades fixas (r1025.0)")
        # P0840 - Data type: U32/Binary -> "I/B"
        g120.write_parameter(840, '1025.0', "I/B")
        time.sleep(1)
        print("Define que a variação de velocidade será dado por
velocidades fixa P1000 = 3")
        # P1000 - Data type: Integer16 -> h (short - interger)
        g120.write_parameter(1000, 3, "h")
        time.sleep(1)
        print("Definição dos valores das 03 velocidade fixa (P1001, P1002,
P1003)")
        # P1001/ P1002/ P1003 - Data type: FloatingPoint32 -> f
(float)refere-se aos valores que serão inseridos pelo usuário para as 03
velocidades fixas, no P1001 é atribuído a velocidade mais baixa, no P1002 é
atribuída a velocidade média e no P1003 será atribuída a velocidade mais
alta. O comando ids faz referência ao código .kv lincando valores que seão
```

```
inseridos na tela ao programa principal.
        w_p1001 = float(self.ids.v1.text)
        print(w_p1001)
        time.sleep(1)
        g120.write_parameter(1001, w_p1001, "f")
        w_p1002 = float(self.ids.v2.text)
        print(w_p1002)
        time.sleep(1)
        g120.write_parameter(1002, w_p1002, "f")
        w_p1003 = float(self.ids.v3.text)
        print(w_p1003)
        g120.write_parameter(1003, w_p1003, "f")
        time.sleep(1)
        print("Definir que o drive fará a seleção da velocidade fixa pelas
entradas digitais")
        print("P1020 = r7221.0 / P1021 = r722.1 / P1022 = r722.2")
        # r722.0/ r222.1/ r222.2 - Data type: Unsigned32 -> I (Unsigned
int)
        # P1020/ P1021/ P1022 - Data type: U32/Binary -> I/B
        g120.write_parameter(1020, '722.0', "I/B")
        g120.write_parameter(1021, '722.1', "I/B")
        g120.write_parameter(1022, '722.2', "I/B")
        time.sleep(1)
        print("Desabilitar o parametro de reversão de sentido habilitado
pela macro 12 P1113=0")
        # P1113 - Data type: U32/Binary -> I/B
        g120.write_parameter(1113, 0 , "I/B")
        time.sleep(1)
        print("Definir que o reconhecimento de falha será dado pela DI3
P2103=r722.3")
        # P2103 - Data type: U32/Binary -> I/B
        g120.write_parameter(2103, '722.3', "I/B")
        time.sleep(1)
        print("Definir que a parada de emergência será dada pela DI4
P0848=r722.4")
        # P0848 - Data Type: U32/Binary -> I/B
        g120.write_parameter(848, '722.4', "I/B")
        print("Definir o tempo da rampa de desaceleração da parada de
emergência P1135 = Tempo de parada (s)")
        # P1135 - Data type: FloatingPoint32 - f (float)
        g120.write_parameter(1135, 3, "f")
        self.ids.fim_parametriza.text = "Parametrização Realizada com
Sucesso!"

    def monitorar (self):
        # Atualizar os valores na tela durante o monitoramento.
        self.status_monitorar = True
        # Utilização da biblioteca threading responsável por executar
Threads, ou seja, fluxo de programas em paralelo dentro de uma aplicação,
ramificação, aqui é responsável por chamar a função update enquanto a
variável self.status_monitorar é verdadeiro.
        threading.Thread(target=self.update).start()

    # Função responsável por pausar a função de atualização (update) do
status do drive presentes na interface de monitoramento.
    def parar_monitorar(self):
        self.status_monitorar = False

    def reset_monitorar(self):
        self.r00522_color = [1, 1, 1, 1]
        self.r21397_color = [1, 1, 1, 1]
```

```
        self.r21393_color = [1, 1, 1, 1]
        self.r7220_color = [1, 1, 1, 1]
        self.r7221_color = [1, 1, 1, 1]
        self.r7222_color = [1, 1, 1, 1]
        self.r1025 = 0.1
        self.r00522 = 0.1
        self.r21397 = 0.1
        self.r21393 = 0.1
        self.r7220 = 0.1
        self.r7221 = 0.1
        self.r7222 = 0.1
        self.status_monitorar = False
        self.tempo = []
        self.r0022 = 0.1
        self.velocidade = []
        self.r0027 = 0.1
        self.corrente = []
        self.r0026 = 0.1
        self.tensao = []
        self.r0052 = 0.1
        self.r0052_word = '0'
        self.r2139 = 0.1
        self.r2139_word = '0'
        self.r722 = 0.1
        self.r722_word = '0'
        self.r0052_hex = 0
        self.r2139_hex = 0
        self.r722_hex = 0

    def update(self,*args):
        for i in range (1,200):
            if self.status_monitorar == True:
                # Leitura de Parâmetro da operação r0052 bit2 = 1 em
operação
                # self.r0052 = input("Insira uma suposição de valor
hexadecimal para o r0052: ")
                # r0052.2 - Data type: Unsigned16 -> H (Unsigned Short)
                self.r0052 = g120.read_parameter(52,"H")
                print(self.r0052)
                self.r0052_word = bin(self.r0052)[2:]# Necessário o [2:]
pois o número binário no Python sempre inicia com 0b.
                print(self.r0052_word)
                if len(self.r0052_word) < 16:
                    self.r0052_word = ('0' * (16 - len(self.r0052_word)) +
self.r0052_word) # acrescenta-se 0s até completar o total de bits, no caso
16, acrescenta-se sempre no lado mais significativo (lado esquerdo)
                print(self.r0052_word)
                self.r0052_word = self.r0052_word[::-1] # posteriormente
após completar o número de bits é necessário inverter o número para
conseguir selecionar o bit que quer-se ler corretamente.
                print(self.r0052_word)
                self.r00522 = self.r0052_word[2]
                print(self.r00522)
                # Leitura do Parâmetro de falha/alarme r2139 bit 7 = 1 em
alarme
                # self.r21397 = int(input("Insira uma suposição para o
valor de r2139.7 (0 sem Alarme ou 1 com Alarme)"))
                # r2139.7 - Data type: Unsigned16 -> H (Unsigned Short)
                self.r2139 = g120.read_parameter(2139 , "H")
                # self.r2139 = input("insira o valor em hexadecimal para
r2139: ")
```

```
                # self.r2139_hex = (self.r2139, 16)
                self.r2139_word = bin(self.r2139)[2:]
                # self.r2139_word = bin(int(self.r2139, 16))[2:]
                print(self.r2139_word)
                if len(self.r2139_word) < 16:
                    self.r2139_word = ('0' * (16 - len(self.r2139_word)) +
self.r2139_word)
                print(self.r2139_word)
                self.r2139_word = self.r2139_word[::-1]
                print(self.r2139_word)
                self.r21397 = self.r2139_word[7]
                print(self.r21397)
                # Leitura do Parâmetro de falha/alarme r2139 bit 3 = 1 em
falha
                # r2139.3 - Data type: Unsigned16 -> H (Unsigned Short)
                self.r21393 = self.r2139_word[3]
                print(self.r21393)
                # r722.0/ r222.1/ r222.2 - Data type: Unsigned32 -> I
(Unsigned int)
                # Leitura do Parâmetro das velocidades r722.0 , r722.1,
r722.2
                time.sleep(1)
                self.r722 = g120.read_parameter(722, "I")
                # self.r722 = (input("insira o valor em hexadecimal para
r0722: "))
                # self.r722_hex = (self.r722, 16)
                # print(self.r722_hex)
                # self.r0052_word = bin(int(r0052, scale))[2:]
                self.r722_word = bin(self.r722)[2:]
                print(self.r722_word)
                if len(self.r722_word) < 32: # o parâmetro referente ao
r722 possui 32 bit de acordo com o manual do SINAMICS G120.
                    self.r722_word = ('0' * (32 - len(self.r722_word)) +
self.r722_word)
                print(self.r722_word)
                self.r722_word = self.r722_word[::-1]
                self.r7220 = self.r722_word[0]
                print(self.r7220)
                self.r7221 = self.r722_word[1]
                print(self.r7221)
                self.r7222 = self.r722_word[2]
                print(self.r7222)
                # definição de cores para o led run da interface de
monitoramento, ou seja, ira aparecer no led da interface de monitoramento
intitulado RUN a cor verde quando o bit 02 da palavra de controle, no
código intitulado self.r00522, assumir nível logico alto e a cor branca
quando for nível logico baixo, ou seja quando o motor não estiver em
operação.
                if self.r00522 == 1:
                    self.r00522_color = [0,1,0,1]
                else:
                    self.r00522_color = [1,1,1,1]
                # definição de cores para o led Alarme da interface de
monitoramento, se o bit 07 (sete) do parâmetro r2139 estiver em nível
lógico alto, ou seja, a variável self.r21397 assumir 01 (um) o led Alarme
deverá assumir a cor laranja, já se ele estiver em nível lógico baixo
deverá assumir a cor branca.
                if self.r21397 == 1:
                    self.r21397_color = [1,0.5,0,1]
                else:
                    self.r21397_color = [1,1,1,1]
```

```
                # definição de cores para o led Falha da interface de
monitoramento, se o bit 03 (três) do parâmetro r2139 estiver em nível
lógico alto, ou seja, a variável self.r21393 assumir 01 (um) o led Falha
deverá assumir a cor vermelha, já se ele estiver em nível lógico baixo
deverá assumir a cor branca.
                if self.r21393 == 1:
                    self.r21393_color = [1,0,0,1]
                else:
                    self.r21393_color = [1,1,1,1]
                # definição de cores para o led v1 da interface de
monitoramento, se o bit 00 (zero) do parâmetro r0722 estiver em nível
lógico alto, ou seja, a variável self.r7220 assumir 01 (um) o led v1 deverá
assumir a cor verde, já se ele estiver em nível lógico baixo deverá assumir
a cor branca.
                if self.r7220 == 1:
                    self.r7220_color = [0,1,0,1]
                else:
                    self.r7220_color = [1,1,1,1]
                # definição de cores para o led v2 da interface de
monitoramento, se o bit 01 (um) do parâmetro r0722 estiver em nível lógico
alto, ou seja, a variável self.r7221 assumir 01 (um) o led v2 deverá
assumir a cor verde, já se ele estiver em nível lógico baixo deverá assumir
a cor branca.
                if self.r7221 == 1:
                    self.r7221_color = [0,1,0,1]
                else:
                    self.r7221_color = [1,1,1,1]
                # definição de cores para o led v3 da interface de
monitoramento, se o bit 02 (dois) do parâmetro r0722 estiver em nível
lógico alto, ou seja, a variável self.r7222 assumir 01 (um) o led v3 deverá
assumir a cor verde, já se ele estiver em nível lógico baixo deverá assumir
a cor branca.
                if self.r7222 == 1:
                    self.r7222_color = [0,1,0,1]
                else:
                    self.r7222_color = [1,1,1,1]
# Monitoramento das variáveis objetos da construção dos gráficos
                self.tempo.append(i-1)# a função append para cada
incremento de tempo adiciona um valor das variáveis abaixo lidas do drive
em uma lista.
                print(self.tempo)
                # r0022 - Data type: FloatingPoint32 - f (float)
                self.r0022 = float(g120.read_parameter(22,"f"))
                print("r0022 - leitura da velocidade atual do motor")
                # self.r0022 = float(input("insira a suposta velocidade do
motor:"))
                self.velocidade.append(self.r0022)
                print(self.velocidade)
                # r0027 - Data type: FloatingPoint32 - f (float)
                self.r0027 = float(g120.read_parameter(27,"f"))
                print("r0027 - leitura da corrente atual do motor")
                # self.r0027 = float(input("insira a suposta corrente do
motor:"))
                self.corrente.append(self.r0027)
                print(self.corrente)
                # r0026 - Data type: FloatingPoint32 - f (float)
                self.r0026 = float(g120.read_parameter(26,"f"))
                print("r0026 - leitura da tensão de link DC")
                # self.r0026 = float(input("insira a suposta tensão de link
DC:"))
                self.tensao.append(self.r0026)
```

```
                print(self.tensao)


    # Função para definir gráficos:
    def graficos(self):
        fig, axs = plt.subplots(3, constrained_layout=True) # Comando para
definir os 03 (três) gráficos em uma mesma imagem.
        axs[0].plot(self.tempo, self.velocidade)
        axs[0].set_title('Velocidade Atual x Tempo')
        axs[0].set(xlabel='Tempo[s]', ylabel='Velocidade[rpm]')
        axs[0].grid()
        axs[1].plot(self.tempo, self.corrente, 'tab:orange')
        axs[1].set_title('Corrente Atual x Tempo')
        axs[1].set(xlabel='Tempo[s]', ylabel='Corrente[A]')
        axs[1].grid()
        axs[2].plot(self.tempo, self.tensao, 'tab:green')
        axs[2].set_title('Tensão de Link DC x Tempo')
        axs[2].set(xlabel='Tempo[s]', ylabel='Tensão[Volts]')
        axs[2].grid()
        plt.show()

# Gerência as telas:
class Gerenciador(ScreenManager):
    pass

#Chama o arquivo G120 no kivy para assim criar uma interface gráfica:
class G120(App):
    # Definindo o método que inicializa e constrói o aplicativo.
    def build(self):
        return Gerenciador()



# Método responsável por rodar o aplicativo.
if __name__ == '__main__':
    G120().run()
```

Anexo II – Código em linguagem Kivy intitulado G120.kv

```
#Arquivo kivy language responsável por criar as funcionalidades dos widgets
#O Gerenciador é utilizado para gerenciar e organizar as demais telas do
aplicativo.
<Gerenciador>:
    Menu:
        name: 'menu'
    Conectador:
        name: 'conectador'
    Comissionamento:
        name: 'comissionamento'
    Drive:
        name: 'drive'

<Menu>:
    BoxLayout:
        #Faz a definição do fundo de tela (cor)
        canvas:
            Color:
                rgba:1,1,1,1
            Rectangle:
                #para definir que o fundo cobrirá a tela toda
                size: self.size
                pos:self.pos
                source:'siemens_logo.png'
        # É definido a orientação vertical, caso queira a horizontal tal
linha de código não é necessaria.
        orientation:'vertical'
        # Essa propriedade adiciona uma margem ao layout
        padding: 200
        # Essa propriedade adiciona um espaço entre os widgets
        spacing: 50
        Button:
            # Defini o nome do widget botão:
            text: 'Iniciar a conexão com o drive'
            #Definição da função do botão, quando usamos root.current
indicamos qual é a tela que queremos como atual.
            on_release: app.root.current = 'conectador'
        Button:
            text: 'Sair'
            # Comando para sair do aplicativo.
            on_release: app.stop()

<Conectador>:
    orientation:'vertical'
    BoxLayout:
        canvas.before:
            Color:
                #rgb da Siemens = 0, 153, 153 (rgb vale de 0 a 256 e o rgba
0 a 1)
                rgba:0,0.59,0.59,1
            Rectangle:
                size: self.size
                pos: self.pos
        orientation: 'vertical'
        # Criação da barra superior do programa menu superior
```

```
        ActionBar:
            # O ActionView contem os widges especificos da ActionBar.
            ActionView:
                # Item para voltar ao menu, ele insere o ícone do própio
kivy
                ActionPrevious:
                    title: 'Menu Principal'
                    on_release: app.root.current = 'menu'
                # Criando o Botão para sair do aplicativo
                ActionButton:
                    text: 'Sair'
                    on_release: app.stop()
        BoxLayout:
            # Acrescentar uma imagem png que esteja na mesma pasta do
arquivo .kv
            Image:
                source: 'g120.png'
                # size_hint é uma função para definir o tamanho da tela
aqui no caso ela foi desabilitada para a vertical.
                size_hint_y: None
                # Especificar a altura em pixels
                height: 180
                # Permitir alterar o tamanho da imagem:
                allow_stretch: True
        BoxLayout:
            Label:
                text:'Conectar com o IP adress da Siemens:'
                size_hint_y: None
                height: 25
            TextInput:
                # O id é um identificador para acessar esse widget Label:
                id:ip
                size_hint_y: None
                # self.minimum_height é uma função para adequar todos
widgets desse BoxLayout ao tamanho da tela
                height:self.minimum_height
        BoxLayout:
            orientation: 'vertical'
            Button:
                text: 'Conectar'
                # root é uma referência ao método da classe Conectador
                on_release: root.conectar()
            Label:
                id: conectar
                text: 'Ainda não conectado!'
            Button:
                text: 'Next'
                size_hint: .2,.5
                pos_hint: {'right': 1, 'y': 0}
                # app é uma referência ao aplicativo que esta rodando
(G120)
                # app.root faz uma referência ao gerenciador e chama o
método current
                on_release: app.root.current = 'comissionamento'

<Comissionamento>
    orientation:'vertical'
    BoxLayout:
        canvas.before:
            Color:
                rgba:0,0.59,0.59,1
```

```
            Rectangle:
                size: self.size
                pos: self.pos
        orientation: 'vertical'
        ActionBar:
            ActionView:
                ActionPrevious:
                    title: 'Menu Principal'
                    on_release: app.root.current = 'menu'
                ActionButton:
                    text: 'Sair'
                    on_release: app.stop()
        BoxLayout:
            orientation:'vertical'
            BoxLayout:
                #Para controlar o tamanho do botão e para o label.
                size_hint_y: None
                height: 75
                orientation: 'vertical'
                Button:
                    text: 'Reset de Fábrica'
                    on_release: root.reset_fabrica()
                Label:
                    id: info_reset
                    text:''
            BoxLayout:
                size_hint_y: None
                height: 75
                Label:
                    text: 'Comissionamento:'
                    font_size: 20
            GridLayout:
                cols: 2
                BoxLayout:
                    Label:
                        # Define o tamanho da fonte do widget no caso do
Label:
                        font_size: 14
                        text: 'Inserir o valor de Tensão \n de Alimentação
[Vrms]:'
                    TextInput:
                        id: v_alimentação
                BoxLayout:
                    Label:
                        font_size: 14
                        text: 'Inserir o valor de Tensão \n Nominal do
Motor [V]:'
                    TextInput:
                        id: v_motor
                BoxLayout:
                    Label:
                        font_size: 14
                        text: 'Inserir o valor de Corrente \n Nominal do
Motor [A]:'
                    TextInput:
                        id: i_motor
                BoxLayout:
                    Label:
                        font_size: 14
                        text: 'Inserir o valor de Potência \n Nominal do
Motor [kW]:'
```

```
                    TextInput:
                        id: p_motor
                BoxLayout:
                    Label:
                        font_size: 14
                        text: 'Inserir o valor de Frequência \n do Motor
[Hz]:'
                    TextInput:
                        id: f_motor
                BoxLayout:
                    Label:
                        font_size: 14
                        text: 'Inserir o valor de Velocidade \n nominal do
Motor [rpm]:'
                    TextInput:
                        id: vel_motor
                BoxLayout:
                    Label:
                        font_size: 14
                        text: 'Inserir o valor de Velocidade \n Mínima do
Motor [rpm]:'
                    TextInput:
                        id: vmin_motor
                BoxLayout:
                    Label:
                        font_size: 14
                        text: 'Inserir o valor de Velocidade \n Máxima do
Motor [rpm]:'
                    TextInput:
                        id: vmax_motor
                BoxLayout:
                    Label:
                        font_size: 14
                        text: 'Inserir o valor da rampa \n de aceleração do
Motor [s]:'
                    TextInput:
                        id: ta_motor
                BoxLayout:
                    Label:
                        font_size: 14
                        text: 'Inserir o valor da rampa de \n desaceleração
do Motor [s]:'
                    TextInput:
                        id: td_motor
            BoxLayout:
                size_hint_y: None
                height: 150
                orientation: 'vertical'
                Button:
                    text: 'Identificar os dados do motor'
                    on_release: root.inicio_comissionamento()
                Label:
                    id: fim_comissionamento
                    text: ''
                Button:
                    text: 'Next'
                    size_hint: .2,.5
                    pos_hint: {'right': 1, 'y': 0}
                    on_release: app.root.current = 'drive'

<Drive>
```

```
orientation:'vertical'
BoxLayout:
    canvas.before:
        Color:
            rgba:0,0.59,0.59,1
        Rectangle:
            size: self.size
            pos: self.pos
    orientation: 'vertical'
    ActionBar:
        ActionView:
            ActionPrevious:
                title: 'Menu Principal'
                on_release: app.root.current = 'menu'
            ActionButton:
                text: 'Sair'
                on_release: app.stop()
    GridLayout:
        cols:2
        BoxLayout:
            orientation:'vertical'
            padding: 50
            spacing: 25
            BoxLayout:
                Label:
                    id: status_v1
                    canvas:
                        Color:
                            rgba: root.r7220_color
                        Ellipse:
                            size: [30,30]
                            pos: [125,470]
                Label:
                    text:'v1'
                TextInput:
                    id:v1
            BoxLayout:
                Label:
                    id: status_v2
                    canvas:
                        Color:
                            rgba: root.r7221_color
                        Ellipse:
                            size: [30,30]
                            pos: [125,400]
                Label:
                    text:'v2'
                TextInput:
                    id:v2
            BoxLayout:
                Label:
                    id: status_v3
                    canvas:
                        Color:
                            rgba: root.r7222_color
                        Ellipse:
                            size: [30,30]
                            pos: [125,330]
                Label:
                    text:'v3'
                TextInput:
```

```
            id:v3
BoxLayout:
    padding: 50
    spacing: 25
    orientation: 'vertical'
    Button:
        text: 'Iniciar a Parametrização da Aplicação'
        on_release: root.iniciar_app()
    Label:
        id: fim_parametriza
        text: ''
BoxLayout:
    padding: 75
    BoxLayout:
        orientation:'vertical'
        Label:
            id: run
            canvas:
                Color:
                    rgba: root.r00522_color
                Ellipse:
                    size: [30,30]
                    pos: [100,125]
        Label:
            text: 'Run'
    BoxLayout:
        orientation:'vertical'
        Label:
            id: alarme
            canvas:
                Color:
                    rgba: root.r21397_color
                Ellipse:
                    size: [30,30]
                    pos: [190,125]
        Label:
            text: 'Alarme'
    BoxLayout:
        orientation:'vertical'
        Label:
            id: falha
            canvas:
                Color:
                    rgba: root.r21393_color
                Ellipse:
                    size: [30,30]
                    pos: [270,125]
        Label:
            text: 'Falha'
BoxLayout:
    orientation:'vertical'
    padding: 50
    spacing: 25
    Button:
        text: 'Iniciar Monitoramento'
        on_release: root.monitorar()
    Button:
        text: 'Parar Monitoramento'
        on_release: root.parar_monitorar()
    Button:
        text: 'Reset Monitoramento'
```

```
    on_release: root.reset_monitorar()
Button:
    text: 'Análise Gráfica'
    on_release: root.graficos()
```