UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
DEPARTAMENTO ACADÊMICO DE ELETROTÉCNICA
ENGENHARIA ELÉTRICA COM ÊNFASE EM AUTOMAÇÃO

ADRIEL LIMA AZEVEDO
ANDERSON LUIZ CORREA

VISÃO COMPUTACIONAL APLICADA A MANIPULADOR ROBÓTICO

CURITIBA
2019

ADRIEL LIMA AZEVEDO

ANDERSON LUIZ CORREA

VISÃO COMPUTACIONAL APLICADA A MANIPULADOR ROBÓTICO

TCC2 apresentada ao curso de Graduação em Engenharia Elétrica com Ênfase em Automação, apresentado à disciplina de Trabalho de conclusão de curso 2, do Departamento Acadêmico de Eletrotécnica (DAELT) da Universidade Tecnológica Federal do Paraná (UTFPR) como requisito para obtenção do título de Engenheiro Eletricista.

Orientador: Prof. Dr. Roberto Cesar Betini

Coorientador: Prof. Me. Carlos André

Barbosa de Almeida

CURITIBA

2019

Adriel Lima Azevedo
Anderson Luiz Correa

Visão Computacional aplicada a Manipulador Robótico

Este Trabalho de Conclusão de Curso de Graduação foi julgado e aprovado como requisito parcial para a obtenção do Título de Engenheiro Eletricista, do curso de Engenharia Elétrica com ênfase em Automação do Departamento Acadêmico de Eletrotécnica (DAELT) da Universidade Tecnológica Federal do Paraná (UTFPR).

Curitiba, 21 de novembro de 2019.

_____

Prof. Paulo Sérgio Walênia, Dr.
Coordenador de Curso
Engenharia Elétrica

_____

Profa. Annemarlen Gehrke Castagna, Mestre
Responsável pelos Trabalhos de Conclusão de Curso de
Engenharia Elétrica do DAELT

ORIENTAÇÃO

_____

Roberto Cesar Betini, Dr.
Universidade Tecnológica Federal do Paraná
Orientador

_____

Carlos André Barbosa de Almeida, Me.
Faculdade Estácio de Curitiba
Coorientador

BANCA EXAMINADORA

_____

Mariana Antonia Aguiar Furucho, Dra.
Universidade Tecnológica Federal do Paraná

_____

Vilmair Ermenio Wirmond, Me.
Universidade Tecnológica Federal do Paraná

A folha de aprovação assinada encontra-se na Coordenação do Curso de Engenharia Elétrica

AGRADECIMENTOS

Primeiramente à Deus, pela vida, saúde e oportunidade dispensada. Por ter guiado nossos passos até aqui, possibilitando-nos atingir grandes objetivos, trazendo até nós pessoas gentis e auxiliadoras neste caminho de conhecimento.

A nossa família e amigos por todo amor e apoio incondicional fornecido. Mesmo nos momentos de maior dificuldade, respeitaram nossos silêncios e compreenderam nossas ausências.

Ao orientador, Professor Doutor Roberto Cesar Betini, e ao coorientador, Professor Mestre Carlos Andre Barbosa de Almeida, que além de apoiarem nossa idéia, incentivaram e orientaram para que atingíssemos nosso objetivo.

Aos professores que aceitaram nosso convite para compor a banca examinadora, e compartilharam seus conhecimentos e sugestões, até mesmo as críticas construtivas que muito colaboraram para o nosso melhor desempenho.

À nossa instituição de ensino, pelo apoio e disposição dos recursos técnicos e pedagógicos, administrativos, e a todo corpo docente pela dedicação para que tivéssemos uma formação justa e com muita qualidade.

RESUMO

AZEVEDO, Adriel Lima; CORREA, Anderson Luiz. **Visão Computacional aplicada a Manipulador Robótico.** 2019. 97f. Trabalho de Conclusão de Curso (Graduação em Engenharia Elétrica), Departamento de Eletrotécnica, Universidade Tecnológica Federal do Paraná, Curitiba, 2019.

Com o propósito de buscar melhorias para os sistemas industrias, obtendo interação entre equipamentos automatizados. Este trabalho apresenta algumas ferramentas que dá acesso ao desenvolvimento de programas para um sistema de visão computacional, com o objetivo de detectar obstáculos na área de trabalho do manipulador robótico, a partir de um sistema de comunicação estruturado, sendo capaz de captar informações e enviar para processamento e tomada de decisões rápidas, gerando o desvio de trajetória do manipulador. Para isso são abordados alguns tópicos de embasamento teórico sobre o manipulador robótico, métodos de funcionamento e movimentação, assim como as ferramentas de Visão Computacional, utilizando o OpenCV, para processamento de imagens e reconhecimento de objetos e o ROS (Sistema Operacional Robótico), que dá suporte as diversidades de ferramentas para manipulação e sensoriamento de um robô. São apresentados os materiais e métodos utilizados, que são os softwares e hardwares necessários para a implementação do sistema. Ao final são descritas todas as etapas de desenvolvimento do trabalho, finalizando com a apresentação dos resultados e comentários conclusivos.

Palavras-chave: Manipulador robótico, Visão computacional, Sistemas embarcados, Sistema Operacional Robótico (ROS), OpenCV.

ABSTRACT

AZEVEDO, Adriel Lima; CORREA, Anderson Luiz. Title of work **(Computational Vision Applied to Robotic Manipulator).** 2019. 97f. Trabalho de Conclusão de Curso (Graduação em Engenharia Elétrica), Departamento de Eletrotécnica, Universidade Tecnológica Federal do Paraná, Curitiba, 2019.

With the objective of seeking improvements in industrial systems, obtaining interaction between automated equipment. This article presents some tools that give access to the development of programs for a computer vision system, aiming to detect obstacles in the robotic manipulator's desktop, from a structured communication system, capable of capturing information and sending it to fast, processing and decision making, leading to the deviation of the manipulator's trajectory. To this end, some theoretical topics on robotic manipulator, operation and movement methods, as well as Computer Vision's tools using OpenCV for image processing and object recognition and ROS (Robotic Operating System) are discussed

. Supports the diversity of tools to manipulate and detect a robot. The materials and methods used, which are the software and hardware necessary for the system implementation, are presented. At the end are described all the stages of development of the work, ending with the presentation of results and final comments.

Keywords: Robotic Manipulator, Computer Vision, Embedded Systems, Robotic Operating System (ROS), OpenCV.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS OU SIGLAS

ROS: Robot Operation System (Sistema Operacional Robótico);

ISO: International Organization for Standardization (Organização Internacional para Padronização);

OpenCV: Open Source Computer Vision Library (Biblioteca de Visão Computacional de Código Aberto);

CAD: Computer Aided Design (Desenho Assistido por Computador);

RGB: Red, Green, Blue (Vermelho, Verde, Azul);

SUMÁRIO

1 INTRODUÇÃO

1.1 TEMA

A utilização de ferramentas e utensílios para o auxílio nas realizações das atividades cotidianas, já é uma prática desde a origem do ser humano. Pois está diretamente relacionada às necessidades de sobrevivência.

O conceito de evolução humana, para a civilização ocidental, associa-se ao grau que a tecnologia se desenvolve ao longo do tempo, justificando a motivação da invenção de máquinas com o objetivo de substituir o homem na realização de suas tarefas.

Segundo Romano e Dutra (2002), Aristóteles (séc. IV a.C.) foi quem escreveu a primeira referência explícita a este conceito de automação: “se os instrumentos pudessem realizar suas próprias tarefas, obedecendo ou antecipando o desejo de pessoas . . .”.

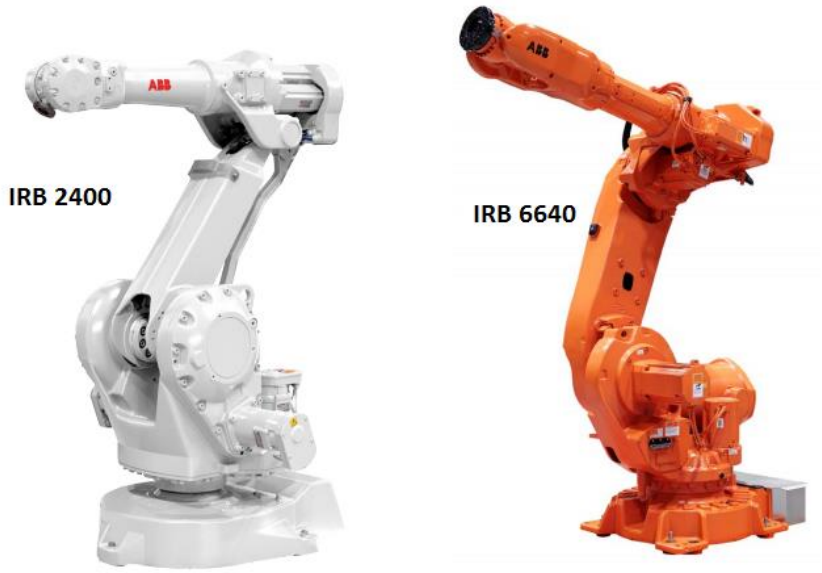
Após o desenvolvimento da máquina a vapor, por James Watt em 1769, a automação de processos produtivos teve um grande avanço, resultando na revolução sócio-econômica das relações humanas, através do crescimento na produção industrial (em larga escala) e o progresso nos meios de transporte.

Com a grande revolução industrial, a partir do século XVIII, a indústria tem demonstrado gradativamente, a necessidade do aprimoramento tecnológico, visando a melhoria da produtividade, da qualidade dos processos, diminuição de desperdícios e o cumprimento das exigências dos prazos, que são relativamente curtos.

Portanto, há um crescimento considerável da aplicação da robótica no setor industrial, especificamente os manipuladores robóticos, que têm capacidade de executar tarefas inadequadas ao ser humano, em questão da segurança, agilidade, precisão, repetitividade, alcance, entre outras.

Romano e Dutra (2002), afirma que a norma ISO 10218, tem uma definição mais completa ao robô industrial, como sendo: "uma máquina manipuladora com vários graus de liberdade controlada automaticamente, reprogramável, multifuncional, que pode ter base fixa ou móvel para utilização em aplicações de automação industrial".

Figura 1- Manipuladores Robóticos.



Fonte: ABB Robotics (2019).

Na Figura 1, são apresentados dois modelos de robôs manipuladores fabricado pela ABB Robotics, uma das fornecedoras de robôs industriais.

Há uma grande parte de robôs, que executam suas tarefas ao lado dos operários nas plataformas industriais. No setor logístico, transportando caixas; nos setores automotivos, na montagem, soldagem ou pintura de peças automotivas e no setor alimentício, no transporte de alimento de uma esteira para uma outra célula da produção. Especialistas ainda se preocupam com a segurança entre manipulador robótico e o ser humano, trabalhando no mesmo ambiente.

Atualmente, lidar com a interação entre humanos e robôs, tornou-se um grande desafio na robótica, a fim de possibilitar que ambos trabalhem juntos de forma natural, efetiva e segura. Para isso, é preciso que o ambiente ao redor do robô seja, pelo menos parcialmente, conhecido. Visando a garantia de que, na movimentação, não cause danos aos operadores próximos a ele.” O robô deve ser capaz de evitar colisões com obstáculos, localizados no seu espaço de trabalho, e alcançar os objetivos de controle simultaneamente” (ANTONIO, 2014).

Um sistema de visão artificial ao manipulador robótico, seria uma maneira de lidar com tal dificuldade. Para isso, deverá ser capaz de, além de captar a imagem, interpretar e transmitir um comando, de modo que o manipulador possa executar a tarefa adaptando-se a condições operacionais. É imprescindível um sistema de

comunicação eficaz na transmissão das informações ao robô, para que haja resposta imediatas à detecção de obstáculos (GRASSI, 2005).

Esse sistema de visão robótica, proporciona à empresa mais agilidade em seu processo de verificação e qualidade do produto fabricado. “O sistema de visão robótica trabalha incansavelmente, sem fadiga e com a precisão que a indústria necessita” (ECOS, 2018).

O sistema de visão para robôs pode ser desenvolvido no processo off-line e on-line. Por exemplo, “para inspeção de produtos”. No processo off-line é gerado um banco de dados com informações sobre os produtos que serão inspecionados. Enquanto que, no processo on-line, tem a função de identificar os objetos com dados cadastrais. Por meio destes processos, é possível determinar posições e orientações do produto, através de comparação de modelos previamente armazenados em um banco de dados. (ECOS, 2018).

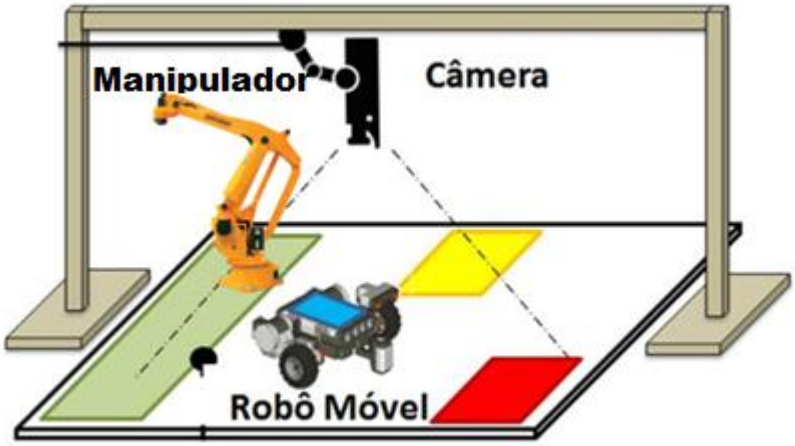
Considera-se que, quanto maior a autonomia do robô, mais eficiente será em termos produtivos. Porém, mais perigosos se tornam, aumentando o risco de acidente quando interagindo com o ser humano em um mesmo ambiente de trabalho. No caso de um incidente, com pessoas ou obstáculos, pode haver danos irreparáveis, seja com as pessoas, obstáculos ou com o próprio robô.

Este projeto, emprega o sistema de visão computacional interagindo com o ROS (Sistema Operacional Robótico), cuja função é comandar o manipulador robótico. O trabalho proposto faz uso de software livre e equipamentos de baixo custo, como a plataforma Raspberry Pi e a estrutura física do braço robótico, manufaturada com o uso da impressora 3D.

1.2 DELIMITAÇÃO DO TEMA

O foco deste trabalho é o sistema de processamento de imagem computacional para correção de trajetória. Para tanto, será necessário a implementação de um software em um manipulador robótico, que tem por finalidade desviar barreiras (obstáculos) e reconhecer objetos que possam adentrar em seu raio de ação. A presente proposta fará uso de protótipo, com a possibilidade de implementação na indústria.

Figura 2 – Representação do ambiente de trabalho.



Fonte: Autoria própria (2018).

A Figura 2 apresenta um modelo de ambiente de trabalho, monitorado por uma câmera, com um braço manipulador executando uma tarefa em um mesmo ambiente que um (AGV) veículo autoguiado.

1.3 PROBLEMAS E PREMISSAS

Com a evolução da tecnologia na área de automação industrial, os trabalhos perigosos, tediosos e insalubres, também de esforços repetitivos que possam causar lesões nos operadores humanos devido ao trabalho, estes podem ser substituídos por robôs. Onde após avaliação de risco, podem trabalhar ao lado dos funcionários sem proteções de segurança (UNIVERSAL, 2018). Com isso, torna-se necessário o uso de um sistema de segurança e monitoramento mais eficientes, para que homens e máquinas possam trabalhar lado a lado.

É observado que, cada vez mais, robôs móveis circulam autonomamente dentro das fábricas, podendo cruzar o caminho de um manipulador, seja para reduzir seu percurso em atividades paralelas, ou para realimentação da linha de produção.

1.4 OBJETIVOS

1.4.1 Objetivo Geral

Utilizar técnicas de Visão Computacional para desenvolver um sistema de desvio da trajetória do manipulador robótico mediante a detecção e reconhecimento de obstáculo.

1.4.2 Objetivos Específicos

- Definir a estrutura do sistema: dimensionamento dos servo motores, programação do software, responsável pelo movimento do robô, e a integração do hardware e sensores.
- Desenvolver os circuitos elétricos e eletrônicos do sistema de proteção do hardware.
- Elaborar o reconhecimento de obstáculos, através do processamento de imagem, captada por uma câmera de monitoramento.
- Criar o modelo do manipulador em 3D, através do software de simulação Gazebo, com movimentação em tempo real. Desta forma será possível acompanhar sua trajetória de movimentação e registrar desvios ou problemas na programação.
- Testar o funcionamento do sistema automatizado.
- Desenvolver um programa de sensoriamento e atuação na correção autônoma da trajetória de um braço robótico.
- Ampliar o conhecimento dos envolvidos com essas tecnologias.

1.5 JUSTIFICATIVA

Quanto mais eficiente e confiável o funcionamento do robô na indústria, mais sua autonomia pode ser aumentada, podendo gerar novas possibilidades de aplicação.

Elaborando o estudo sobre o sistema de segurança aplicado a robótica, abre-se um vasto campo de estudo sobre as ferramentas já existentes, que poderão ser integradas a todo o sistema.

O baixo custo de equipamentos de captura de vídeo e de computadores, além do avanço na tecnologia de processamentos;

A disponibilidade de bibliotecas de funções para o processamento de imagens com código aberto e uso livre.

O aprofundamento sobre a robótica, aprofundar os conhecimentos aplicado neste campo, para descobrir as variadas possibilidades de automatização, de um modo geral.

1.6 PROCEDIMENTOS METODOLÓGICOS

Para a execução do presente trabalho, fez-se necessário a divisão em seis etapas. A primeira etapa, consistiu na elaboração e apresentação da proposta do trabalho, a segunda foi o estudo de formas de movimentação, voltado a característica do manipulador robótico.

A terceira etapa, consistiu no estudo do reconhecimento de imagem, através do processamento de imagem digital. Já, a quarta etapa, realizou-se a programação do Raspberry Pi, com a utilização das ferramentas Open Source, como o Gazebo[1], para simulação. O ROS[2], (Robot Operation System) para programação do manipulador robótico. A biblioteca de Visão Computacional OpenCV e demais bibliotecas e pacotes necessários.

Na quinta etapa, realizou-se os testes de funcionamento do manipulador robótico, com a integração do sistema de visão computacional e comunicação com o ROS, monitorado e coletado dados por diferentes estados, a fim de análises e avaliações de resultados, que possibilitaram algumas correções e melhorias.

Na sexta etapa, são realizadas as considerações finais, e conclusão do projeto, com a elaboração deste relatório, com as descrições dos procedimentos e

[1] **GAZEBO -** Disponível em: < http://gazebosim.org>. Acesso em 11 nov. 2018.

[2] **ROS -** Disponível em: < https://www.ros.org/>. Acesso em 13 nov. 2018.

métodos utilizados, alguns dados relevantes, vinculado às expectativas da proposta e seus objetivos.

2 REVISÃO TEÓRICA

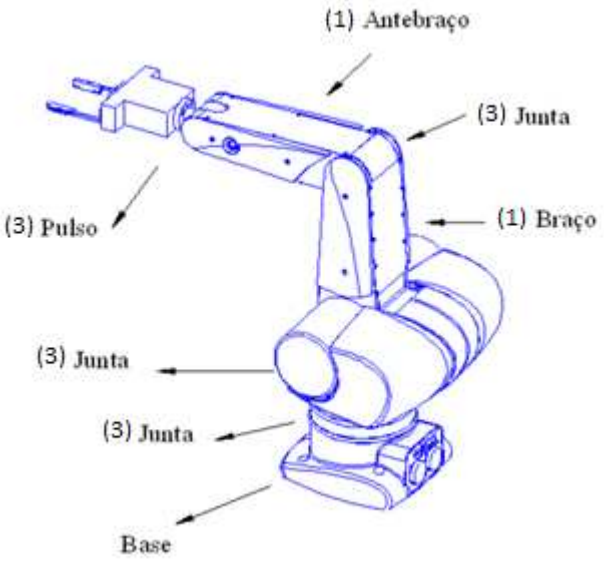
2.1 ROBÓTICA E MOVIMENTAÇÃO

Há diversidades de aplicação de robôs com a finalidade de executar tarefas autônomas ou pré-programadas. Tais tarefas são executadas através de atuadores, (elétricos, pneumáticos, hidráulicos, sonoros, luminosos, etc.), que produzem sons, acendem elementos luminosos ou displays, na articulação de um braço, com abertura ou fechamento de uma garra, ou seja, realização de seu próprio deslocamento.

Especificamente, o robô com maior aplicação na indústria, é o robô industrial, que “consiste em um braço mecânico motorizado programável que apresenta algumas características antropomórficas e um cérebro na forma de um computador que controla seus movimentos” (Rosário, João Maurício, 2005 pag.148).

O braço do robô executa movimentos no espaço, transferindo objetos e ferramentas de um ponto para outro, instruído pelo controlador e informado sobre o ambiente por sensores. Em sua extremidade existe um atuador, análogo a mão humana, que tem por finalidade a execução de suas tarefas. Essa analogia é mostrada na Figura 3, onde as estruturas que representam o braço e antebraço, são denominadas de elos (1); a região do ombro (2), o cotovelo e o “pulso” ou punho são as juntas (3), responsáveis pelas articulações do robô.

Figura 3 – Braço robótico



Fonte: Adaptado de AMARAL (2018).

Segundo (CRAIG, 2012), o estudo da mecânica e controle de manipuladores não é uma nova ciência, mas apenas uma coleção de tópicos extraídos de campos “clássicos”. Pois, enquanto a engenharia mecânica contribui com metodologias para o estudo de máquinas em termos estáticos e dinâmicos. A matemática fornece ferramentas para a descrição de movimentos espaciais e outros atributos dos manipuladores.

A teoria de controle provê ferramentas para projetar e avaliar algoritmos, que realizam os movimentos ou a aplicação de força, desejados. A engenharia elétrica, aplicam suas técnicas no projeto de sensores, atuadores e interfaces para robôs industriais. Por fim, a ciência da computação contribui com a base para a programação desses dispositivos a fim de que desempenhem a tarefa desejada.

2.2 MOVIMENTOS DO MANIPULADOR ROBÓTICO

Para que a manipulação robótica cumpra sua definição, é necessário um mecanismo que representam a posição e orientação das peças, ferramentas e do próprio mecanismo. Logo, para definir e manipular quantidades matemáticas que representam posição e orientação, são estabelecidos sistemas de coordenadas e desenvolvidas convenções de representação, que formam base para posteriores

considerações do movimento, relacionado a velocidades, forças, torques lineares e rotacionais.

No estudo da cinemática de robôs, denomina-se a posição do efetuador, como o conjunto das coordenadas de juntas que o robô apresenta. Isso é possível pela geometria dos elos do robô, tornando-se possível elaborar o equacionamento da cinemática direta[3], utilizada para encontrar as coordenadas de posição, e a cinemática inversa[4], responsável por encontrar os ângulos de orientação, ou seja, os ângulos das juntas, através de equacionamentos. A cinemática direta possibilita que os ângulos de orientação sejam encontrados (SANTOS, 2015).

Figura 4 – Transformação direta de coordenadas

Fonte: ROSÁRIO (2005).

Na Figura 4 é mostrado uma caixa de transformação, que ao realizar o processo da esquerda para direita, conforme indica o fluxo, é realizada a transformação direta, e invertendo o sentido, é realizado a transformação inversa.

Já a dinâmica de manipuladores estuda a relação entre o movimento dos corpos que formam a cadeia cinemática e as forças e torques aplicados nas juntas através dos atuadores a esses associados.

Os conceitos de Tensor de Inércia, Energia Cinética e Potencial são essenciais para o desenvolvimento das equações de movimentos de Lagrange. Porém, num estudo mais simplificado, aplica-se a Segunda Lei de Newton, que trata do efeito que uma força produz nos objetos materiais. É importante ressaltar que como na cinemática, existe a dinâmica direta e a dinâmica inversa, com seus respectivos equacionamentos.

[3] **Cinemática direta** – Relaciona-se com a determinação da trajetória do manipulador conhecendo-se os deslocamentos das juntas.

[4] **Cinemática inversa** – Relaciona-se com a determinação dos deslocamentos das juntas a partir do conhecimento da trajetória do manipulador.

2.2.1 Cinemática direta e inversa - Modelo D-H

O modelo DH (também chamado de parâmetros de Denavit-Hantenberg), refere-se a uma metodologia desenvolvida por Jacques Denavit e Richard Hartenberg, em 1955. Essa metodologia é voltada ao estudo da cinemática direta e inversa, com uma abordagem popular. Embora, haja outras convenções para sistemas de referências, o modelo D-H, é caracterizado por quatro parâmetros convencionados, que são essenciais a fixação do sistema de referência aos elos de um manipulador robótico.

Segundo (CRAIG, 2012), o método D-H é um algoritmo de cinemática direta que permite obter os sistemas de coordenadas e as transformações associadas a cada elo de um manipulador, e o principal objetivo é utilizá-lo em caso de complexidade analítica de um manipulador robótico, devido a quantidade de graus de liberdade.

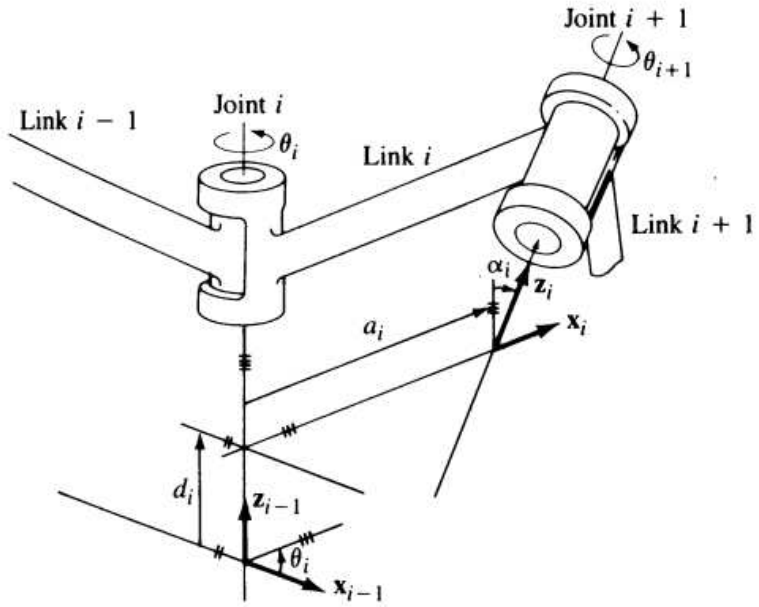
Para aplicação do método DH, são estabelecidos alguns critérios e limitações, para a descrição da cinemática do manipulador:

- É somente para cadeias cinemáticas abertas de corpos rígidos;
- Cada junta do manipulador robótico apresenta um único grau de liberdade de translação ou rotação;
- Os diferentes elos do robô são numerados em ordem crescente, isto é, (Base= “Elo 0” e a Ferramenta= “Elo N”);
- Convenção rigorosa para a definição dos Sistemas de Coordenadas adotados, como também para as coordenadas de posição e orientação.

Pelo fato do método DH obter apenas as coordenadas do elemento terminal do robô, utiliza-se em programas de geração de trajetórias e de identificação de erros, entre outros, requerendo apenas tais coordenadas (ROSÁRIO, 2005).

Os quatro parâmetros de D-H, ou seja, os parâmetros ai, αi, θi e di são associados com cada elo do manipulador. No momento uma convenção de sinais é estabelecida para cada um desses parâmetros. Eles constituem um conjunto suficiente para determinar a configuração cinemática de cada elo do manipulador.

Figura 5 – Parâmetros de Denavit-Hartenberg



Fonte: CRAIG (2013).

Na Figura 5 são apresentados esses parâmetros em suas devidas posições, pode-se verificar que tais parâmetros aparecem em pares, onde ai, αi determinam a estrutura do elo e os parâmetros da junta, respectivamente, e θi, di determinam a posição relativa de elos vizinhos.

Para descrição da translação e a rotação entre dois elos, Denavit-Hartenberg propuseram um método matricial para um estabelecimento sistemático de um sistema de coordenadas fixo para cada elo de um manipulador robótico, resultando na obtenção de uma matriz 4 x 4, mostrado na Figura 6. Tal matriz representa cada sistema de coordenadas do elo na junta, em relação ao sistema de coordenadas do elo anterior. Portanto, a partir de sucessivas transformações, podem ser obtidas as coordenadas do elemento terminal do robô, ou seja, o último elo, expressas matematicamente no sistema de coordenadas fixo à base.

Figura 6 - Matriz DH

$$\mathrm{DH}_{i-1}^{i} = \begin{bmatrix} \cos(\theta_i) & -\sin(\theta_i)\cos(\alpha_i) & \sin(\theta_i)\sin(\alpha_i) & a_i\cos(\theta_i) \\ \sin(\theta_i) & \cos(\theta_i)\cos(\alpha_i) & -\cos(\theta_i)\sin(\alpha_i) & a_i\sin(\theta_i) \\ 0 & \sin(\alpha_i) & \cos(\alpha_i) & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

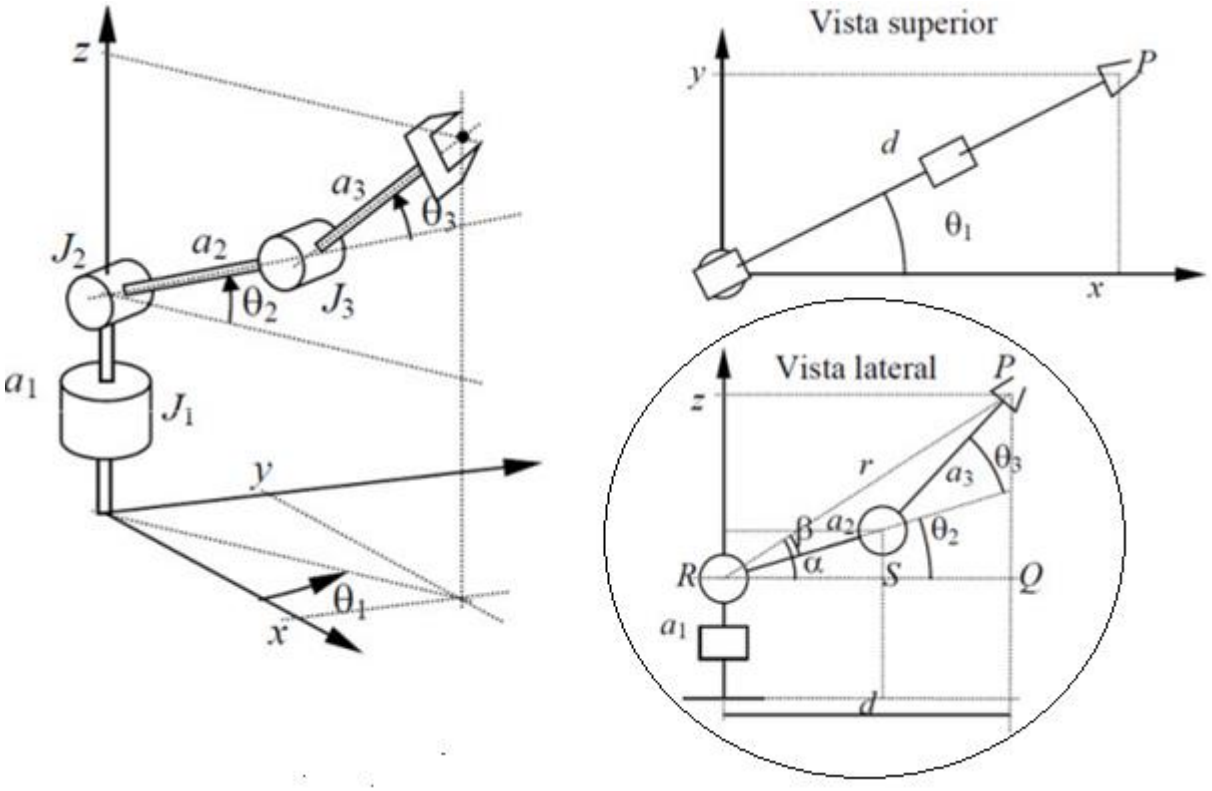
Fonte: Autoria própria (2018).

2.2.2 Coordenadas Generalizadas

Conforme Pazos (2002), as variáveis características das juntas são aquelas grandezas físicas que permitem representar este movimento relativo de um elo com respeito ao anterior. No caso das juntas de revolução, serão os ângulos de rotação entre um elo e o anterior. Observando a Figura 7, é visto que o estado dessas variáveis é suficiente para determinar a posição do efetuador (elemento terminal ou ferramenta do manipulador robótico), pois, se conhecida a posição de cada uma das juntas a partir da primeira, e os comprimentos dos elos, é possível conhecer a posição do efetuador. Na vista, na projeção do ponto P (efetuador) sobre o plano XY, é fornecido a distância horizontal, através da Equação 1, é encontrado a distância entre o centro da base até o efetuador.

$$d = a_2.cos\theta_2 + a_3.cos(\theta_2 + \theta_3) \qquad \text{(Equação 1)}$$

Figura 7 – Manipulador com 3 juntas de revolução.



Fonte: TRONCO (2017).

Em geral, se representam por meio de um vetor de tantas componentes, sejam juntas, representadas por revoluções (Θ) ou distâncias, representadas pelos comprimentos dos elos (a).

2.2.3 Graus de Liberdade

É definido pelo número total de juntas no manipulador. Um manipulador típico tem 6 graus de liberdade, sendo três voltado ao posicionamento do efetuador dentro do espaço de trabalho, e outros três para a obtenção de uma orientação adequada do efetuador ao objeto a ser seguro, ou o ponto à ser efetuado alguma operação (PAZOS, 2002).

O número de grau de liberdade é dependente da necessidade de manipulação, sendo que quanto menor, mais limitados em questão de posicionamento ou alcance do efetuador. Para aplicações onde o espaço de trabalho é composto de

muito obstáculos, pode haver a necessidade de um número maior que 6 graus de liberdade.

Pazos (2002) ressalta que, quanto maior o número de elos do braço, maior será o grau de dificuldade em controlar o movimento.

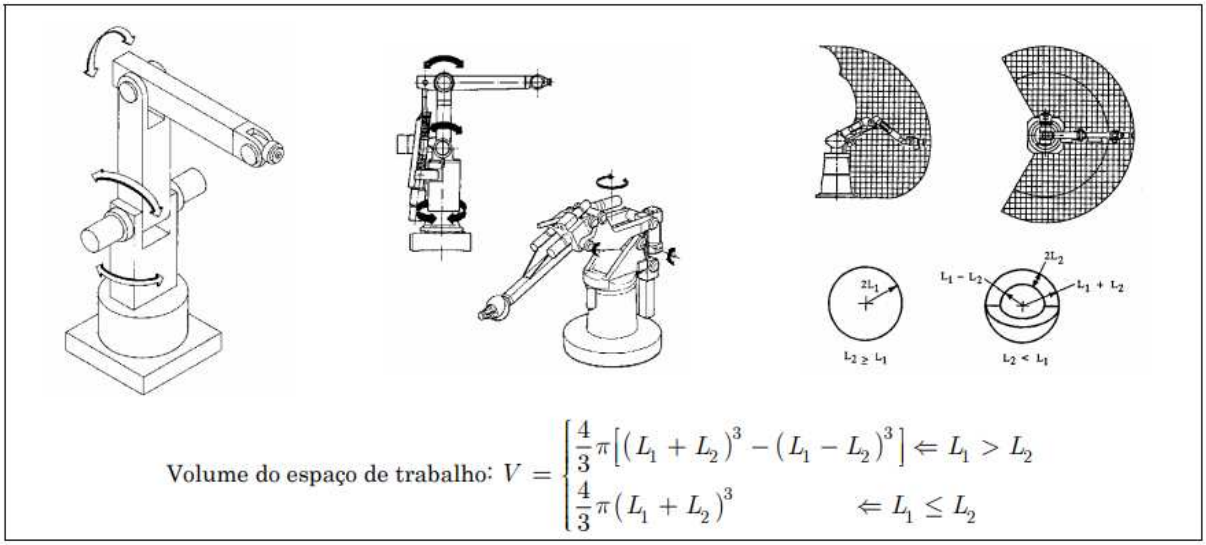
2.2.4 Espaço de Trabalho

É definido como o volume total em conformidade pelo percurso do extremo do último elo (punho), ao manipulador efetuar todas as trajetórias possíveis. O volume dependerá da anatomia do robô, do tamanho dos elos, assim como os limites dos movimentos das juntas. Existe um ângulo máximo de giro, determinados por limites mecânicos (PAZOS,2002).

A estrutura cinemática do manipulador robótico pode ser determinante para o estudo do espaço de trabalho. Para isso são destacadas 5 categorias:

- Cartesiano (PPP): há movimentos lineares em 3 dimensões, e o volume de trabalho é representado pela área cúbica de alcance dos elos.
- Cilíndrico (RPP): movimento da base é rotacional e o outros elos com movimentos lineares. Logo seu volume de trabalho é calculado pela equação de um cilindro.
- Esférico (RRP): composto por duas juntas rotacionais, a junta da base causando uma rotação em torno da base no plano horizontal, e a outra no plano vertical, e uma junta prismática que pode variar o raio da esfera. Podendo assim, representar seu espaço de trabalho como o volume de uma esfera, cujo cálculo é realizado através do volume esférico.
- Articulado Horizontal – SCARA (RRP): composto por duas juntas rotacionais, ambas tem rotações na horizontal, uma em torno do centro da base, e outra entre dois elos, e o movimento da vertical é linear através da junta prismática. Logo o volume de alcance é representado através da equação cilíndrica. Porém, o raio será variável, dependendo do ângulo da junta (cotovelo).
- Articulado Vertical – Braço Antropomórfico (RRR): é composto por 3 juntas que tem rotações no plano vertical e horizontal. Portando o raio de alcance é variado, dependendo do ângulo da junta (cotovelo).

Figura 8 - Articulado vertical (Antropomórfico) RRR



Fonte: SANTOS, Vítor (2004).

A categoria em estudo neste projeto é o Articulado Vertical (RRR). Através de seu raio de alcance é possível calcular o seu volume de trabalho, comparado ao volume de uma esfera. Na Figura 8 é mostrado os detalhes destes movimentos e a expressão para o cálculo do volume de trabalho.

2.3 TIPOS DE MOVIMENTOS

Segundo Santos (2015), os manipuladores robóticos podem ser classificados em quatro categorias, de acordo com o tipo de algoritmo empregado nos seus softwares de movimento:

- **Robôs de sequência fixa:** não empregam servo-controle para indicar as posições relativas das juntas, isto é, são controlados através de chaves-limite e/ou batentes mecânicos.
- **Robôs de repetição com controle ponto-a-ponto:** composto por uma unidade de controle mais sofisticada, onde o manipulador é comandado através de uma série de posições ou movimentos, gravados na memória, e após memorizados, podem ser executados repetitivamente. O fato de ser ponto-a-ponto está nas instruções determinadas pelo programador, em que é estabelecida a movimentação de um ponto a outro, visando o trajeto do manipulador para execução da tarefa.

- **Robôs de repetição com controle de trajetória contínua:** os pontos individuais são definidos pela unidade de controle, e não pelo programador. O programador especifica apenas o ponto de partida e o ponto final da trajetória e a unidade de controle calcula a sequência de pontos individuais. Esse tipo de movimentação é necessário para certos tipos de aplicações industriais, tais como pintura, polimento, soldagem a arco, etc.

Robôs Inteligentes: além de possuir capacidade de repetições de movimento programado, interage com seu ambiente de trabalho de modo inteligentes. Esses robôs podem ter seu ciclo de trabalho alterado, em resposta a condições que possa ocorrer no local de trabalho. Isso acontece através da comunicação tanto com operadores humanos, ou com sistemas computadorizados, utilizando algoritmos de inteligência artificial. Os tipos de aplicações realizadas por robôs inteligentes são baseado no uso de uma linguagem de alto nível para realizar atividades complexas e sofisticadas. Tem aplicações diversas nas tarefas de montagem, seleção de objetos, identificação de padrões, operações complexas de soldagem a arco, entre outras aplicações.

2.4 SISTEMAS DE VISÃO

Em projetos de sistemas automatizados, os sensores não oferecem os requisitos adequados para os requerimentos da aplicação, pois não fornecem informação suficiente para a implementação do algoritmo de controle. O controlador pode precisar de informações além da medição, uma “visão mais geral” da situação da planta ou do ambiente de trabalho, por exemplo, quando um robô manipulador se movimenta em um ambiente de trabalho com possibilidades de deparar com obstáculos imprevisíveis ou desconhecidos (PAZOS, 2002).

Conforme Pazos (2002), um sistema de visão é constituído de uma câmera, a qual ao captar a imagem, entrega um sinal elétrico representativo dessa imagem a uma interface adequada. Na maioria dos casos, esta interface é um dispositivo eletrônico chamado de pré-processador. Cuja função, é analisar os sinais que representam a imagem e gerar informação requerida pelo dispositivo de controle do sistema, o controlador principal, que será utilizada no algoritmo de controle.

Os sistemas de visão estão presentes no campo da robótica nas principais aplicações:

- **Inspeção:** destinado a inspecionar peças com o intuito de assistir um processo de controle de qualidade como: verificar defeitos superficiais; erro de classificação de peças; presença de componente de montagem e dispositivos que fazem parte da montagem; medição da precisão quanto as dimensões das peças e a verificação da presença de furos; assim como outros detalhes de inspeção.
- **Identificação:** referida ao reconhecimento da figura captada na imagem pelo controlador, através dos parâmetros característicos da figura. Geralmente, são aplicados em classificação e seleção de peças, paletização e despaletização e a manipulação de peças posicionadas e orientadas aleatoriamente dentro de um escaninho ou esteira transportadora.

2.4.1 Visão Computacional

Segundo Bradisk e Kaehler (2008), visão computacional é a transformação de dados oriundos de uma máquina fotográfica ou câmera de vídeo em qualquer decisão ou em uma nova representação.

Em outra definição, Shapiro e Stockman (2000) afirmam:

> "A meta da visão computacional é tomar decisões sobre objetos físicos reais a partir de cenas baseadas em imagens captadas".

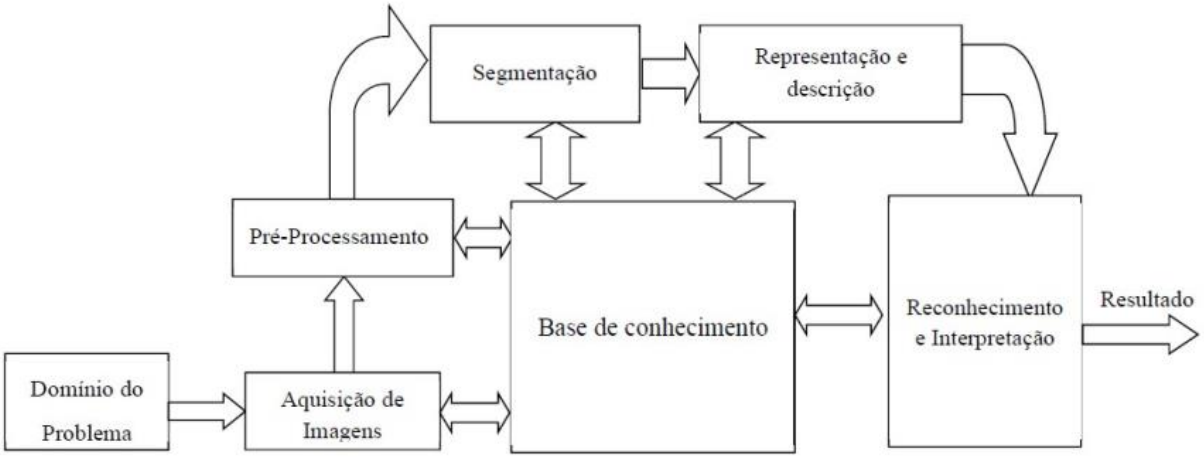
Shapiro e Stockman (2001) afirmam que, a fim de efetuar tomadas de decisões sobre objetos físicos reais, é quase sempre necessário construir alguma descrição ou modelo destes objetos. Por este motivo, alguns especialistas em visão computacional afirmam que a visão computacional tem como meta a "construção de descrições de cenas a partir de imagens".

De acordo com Santos (2014), a visão computacional é responsável por fazer o robô "enxergar" o ambiente. Portanto, são necessárias três características para realização desta função: uma base de dados; velocidade de processamento (análise em tempo real); e a capacidade de trabalho sob condições variadas.

Um sistema de visão computacional tem como principal objetivo encontrar um resultado para um problema. Geralmente ele é dividido em etapas, sendo elas:

aquisição, pré-processamento, segmentação, extração de características, reconhecimento e interpretação, cada etapa estando associada a uma base de conhecimento.

Figura 9 – Diagrama de fluxo dos passos fundamentais no processo de imagens

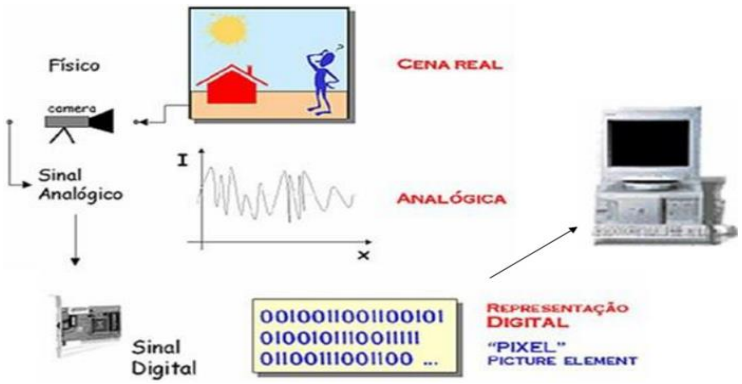


Fonte: AREND (2015).

Na Figura 9 é demonstrado os passos fundamentais para a execução de uma tarefa de processamento de imagens em visão computacional. Conforme indicam as setas, dependendo da necessidade ou disponibilidade de dados do objeto, alguns passos podem ser descartados. Mas, haverá casos em que será necessário que a imagem passe por todas etapas do processamento, para que haja o reconhecimento e interpretação.

- **Aquisição de Imagens:** etapa realizada por uma câmera que transforma a imagem analógica em digital, apresentado na Figura 10. Nesta etapa, os aspectos envolvidos são: Condições de iluminação, velocidade de aquisição, resolução, dentre outros.

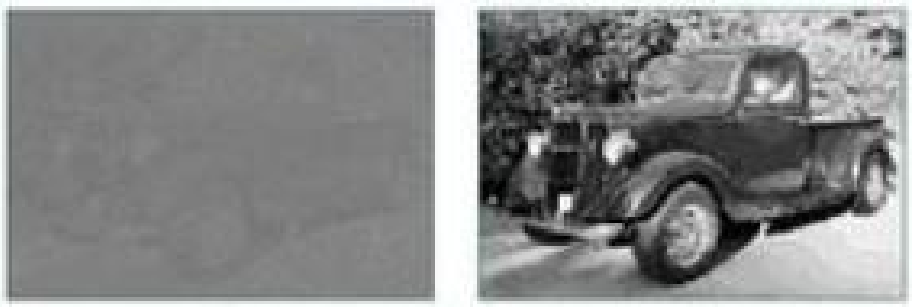
Figura 10 - Aquisição de imagem



Fonte: Marotta (2007).

- **Pré-processamento:** consiste em corrigir imperfeições e aprimorar a qualidade, a fim de garantir o sucesso das etapas seguintes, na
- Figura 11 é apresentado um exemplo do processo. As técnicas mais utilizadas nesta etapa, são a limiarização e a suavização, a partir de análise em histogramas.

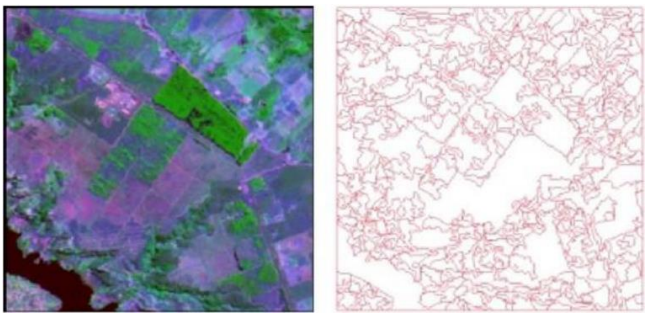
Figura 11 - Pré-processamento de imagem.



Fonte: Marotta (2007).

- **Segmentação:** responsável por dividir a imagem em unidades, ou seja, em objetos de interesse que a compõem. As técnicas utilizadas neste processamento são: a detecção de bordas, afinamento, operadores morfológicos e detecção de regiões, conforme a Figura 12. Para conduzir o processo de extração dos objetos de interesse da imagem, também são utilizados algoritmos específicos que identificam cores, intensidades e formas (linhas, curvas, e outras formas que podem ser parametrizadas) na imagem.

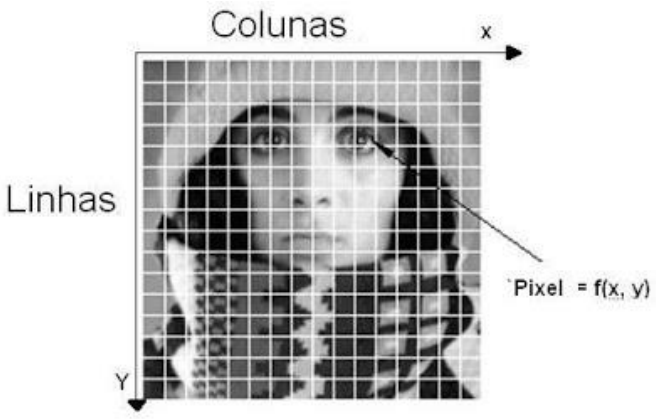
Figura 12 - Segmentação



Fonte: Marotta (2007).

- **Representação e descrição:** as características são extraídas através do descritor, transformando a imagem em um conjunto de dados, denominado vetor de características. Na Figura 13 é apresentada uma matriz no qual os índices de linhas e colunas identificam um ponto na imagem.

Figura 13 - Representação e descrição de imagem.



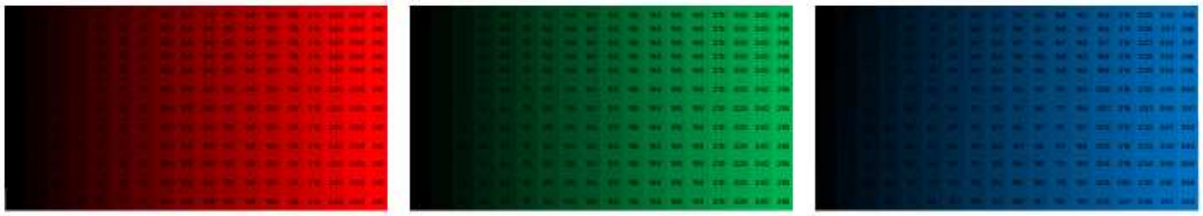
Fonte: Neves (2009).

- **Reconhecimento e interpretação:** identifica e determina a posição e orientação de cada objeto na cena em relação à câmera, e com as informações adquiridas na fase de calibração, determina sua localização em relação a um sistema de coordenadas do ambiente.
- **Base de Conhecimento:** tem armazenado em uma base de dados, o conhecimento do problema a ser resolvido. Necessário para a execução de todas as etapas anteriores.

2.4.1.1 Espaço de cores RGB

Para a detecção de uma imagem colorida, a imagem é lida e armazenada em ‘imagem’, que é uma variável que dará acesso ao objeto da imagem, isto é, uma matriz de 3 dimensões (3 canais), que contém em cada dimensão uma das 3 cores do padrão RGB, que são representadas pelas cores vermelha, verde e azul, que do inglês tem-se respectivamente: red (R), green (G) e blue (B). No caso de uma imagem preto e branca, possui apenas um canal, ou seja, apenas uma matriz de 2 dimensões (ANTONELLO, 2017).

Figura 14 - Matrizes que compõem o sistema RGB.



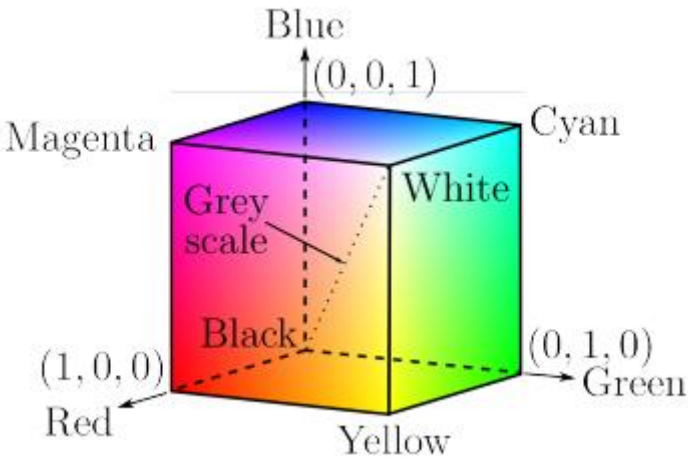
Fonte: Antonello (2017).

Na Figura 14 temos um exemplo das 3 matrizes que compõe o sistema RGB. Cada pixel da imagem, portanto, é composto por 3 componentes de 8 bits cada, sem sinal, gerando 256 combinações por cor. Portanto, a representação é de 256 vezes 256 vezes 256 ou 256^3 que é igual a 16,7 milhões de cores.

No caso de uma matriz de 2 dimensões, cada célula dessa matriz é um pixel, que para imagens preto e brancas possuem um valor de 0 a 255, sendo 0 para preto e 255 para branco. Já para as matrizes com 3 dimensões, no sistema RGB, (0,0,0) representa o preto e (255,255,255) o branco.

Uma representação tradicional para o espaço de cores RGB, é mostrada na Figura 15, através de um cubo de cores.

Figura 15 - Cubo de cores RGB.



Fonte: OLIVEIRA (2013).

Na Figura 15, é mostra um cubo formado por 3 eixos representados pelas letras R (Red), G (Green) e B (Blue). Esses eixos estão normalizados, variando de 0 até 1. É observável que nos vértices do cubo são encontradas as cores aditivas primárias e subtrativas, exceto os dois vértices, que se refere ao preto (black) e ao branco (while), cuja diagonal formada por estes vértices apresentam os tons de cinza (gray scale).

Considerando a escala de 0 a 1, onde 1 representa o pixel de maior valor (255), as seguintes cores podem ser verificadas:

- Branco - RGB (255,255,255);
- Azul - RGB (0,0,255);
- Vermelho - RGB (255,0,0);
- Verde - RGB (0,255,0);
- Amarelo - RGB (255,255,0);
- Magenta - RGB (255,0,255);
- Ciano - RGB (0,255,255);
- Preto - RGB (0,0,0).

Com esses conceitos é possível imaginar o grande número de combinações possíveis que resultarão em 16,7 milhões de cores, conforme mencionado anteriormente.

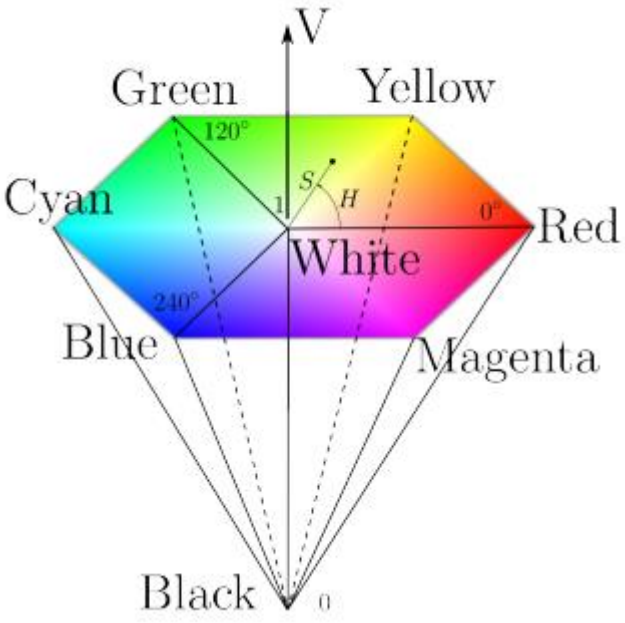
2.4.1.2 Atributo de cor HSV

Na secção 2.4.1.1 foi apresentado a definição de espaço de cores no sistema RGB, representado pelo cubo de cores (Figura 15). Portando, em qualquer ponto interior do cubo representará uma cor a partir das combinações das cores RGB.

Dependendo dos valores dos componentes selecionados tem como resultado uma intensidade (brilho) associada, uma quantidade de luz branca, que determina sua saturação e uma cor predominante matiz, mais conhecida como tonalidade (OLIVEIRA, 2013).

Essas novas referências são conhecidas como atributos, e conhecendo essas atribuições é criado uma representação espacial, chamada de atributos de cor para a mesma cor obtida pelo espaço de cor. Uma representação completa do espaço de atributos pode ser visualizada na Figura 16.

Figura 16 - Representação de atributos de cor com cone hexagonal.



Fonte: Adaptado de OLIVEIRA (2013).

Observando a Figura 16, pode-se imaginar um conde invertido, onde a base representa o hexágono, gerado pela vista 3D do cubo de cores, cujos vértices representam os ângulos de uma circunferência circunscrita, esses ângulos, em revolução de 0 a 360 graus, representam o valor da matiz (H). O eixo (diagonal entre os vértices do branco e do preto no cubo de cores), representa o valor da intensidade

(V) que variam de 0 a 1, e o raio representa a saturação (S). Nos ângulos 0º, 120º e 240º, são apresentadas as tonalidades ou matizes RGB, respectivamente.

É possível a conversão dos sistemas RGB para o sistema HSV, com equacionamentos gerados a partir de transformações geométricas.

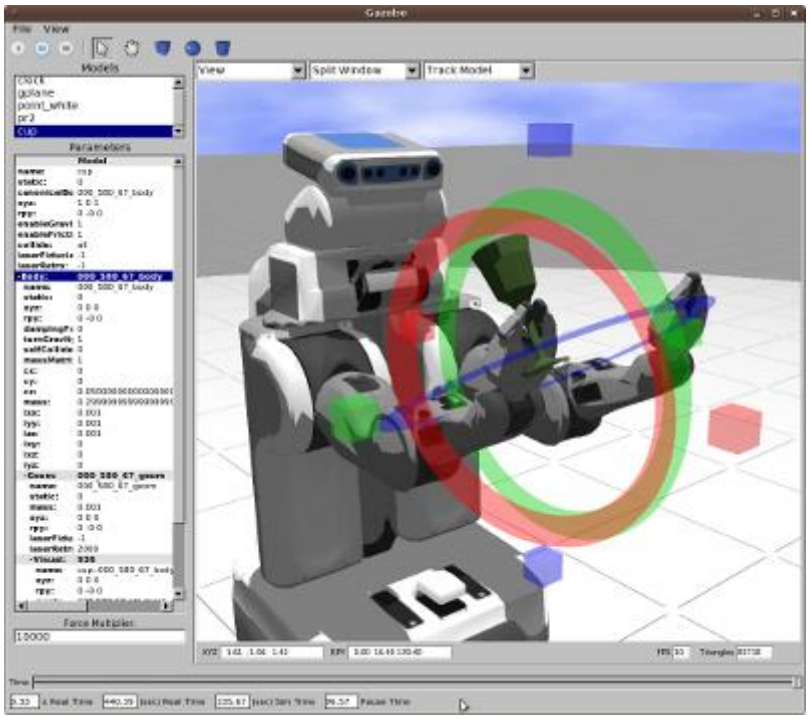
2.5 SIMULADOR ROBÓTICO

Um simulador de robô é um programa de computador que permite criar um ambiente em que o robô se desloca em ambientes fechados ou em campo aberto sem a necessidade de usar o robô propriamente dito, possibilitando que se poupem recursos financeiros.

Nesse ambiente são simulados os comportamentos do robô em um mundo virtual em que podem interagir com outros robôs e/ou com corpos rígidos. Baseado nessas simulações que os robôs são programados.

A simulação em robótica permite a quem faz o desenvolvimento testar o seu código de programação e verificar se o design mecânico está de acordo com o design proposto nos requisitos (Souza, 2015).

Figura 17 - Simulador Robótico em 3D



Fonte: GAZEBOSIM (2018).

Na Figura 17 é mostrado um ambiente de simulação em 3 dimensões de um robô, que através do programa pode simular tanto o movimento de seus membros, como o deslocamento pela área de simulação construída pelo software, que fornece esta opção de desenvolvimento do ambiente.

2.6 PROGRAMAÇÃO

A programação é o ato de programar, mas não está somente ligado a computadores, também faz parte de seguir rotinas sequenciadas para o desenvolvimento de diversas atividades (VANÇAN, 2017). É como uma receita, necessita seguir uma ordem adequada para o correto preparo, ou ainda pode-se comparar com a programação de atividades do dia a dia.

Na programação de computadores e/ou sistemas embarcados, se faz necessário a utilização de linhas de comandos, escritas em uma cadeia de sequencias de acontecimentos.

2.6.1 Linguagem de Programação

De acordo com Vançan (2017), devido ao computador compreender apenas números que atinjam valores de 0 e 1, ouve a necessidade do desenvolver uma forma de comunicação, de forma a facilitar a compreensão de quem programa.

Dito isto, tem-se que a linguagem de programação é tido uma escrita formal, a qual é capaz de passar instruções para o computador. Semelhante as frases usadas na linguagem humana como citado por Haverbeke (2014), porém tem particularidades encontradas nos diferentes tipos de linguagens existentes, tais como interface de usuário, forma de declarar uma variável, entre outras.

2.6.2 Python

Segundo Ebermam et al. (2017), Python é uma linguagem de alto nível, que se aproxima da escrita humana. Onde a linguagem é objetiva e clara, não sendo necessário o uso de caracteres especiais para o seu funcionamento. As variáveis não

necessitam de pré declaração de tipo, portanto são de tipagem dinâmica. Também é observado a forma correta de indentação do código, pois se a forma a qual o texto é organizado, estiver errado, não irá funcionar.

2.7 ROBOTIC OPERATION SYSTEM

O ROS é um framework para desenvolvimento de robôs que disponibiliza uma estrutura para comunicação entre módulos, ferramentas de software, simuladores, implementações de sistemas de visão, reconhecimento de voz, navegação e muito mais (ROS, 2014).

Segundo Augusto (2013), Robotic Operation System (ROS) é um software que teve seu desenvolvimento em 2007, para implementação em qualquer tipo de robô. Como é para a área da robótica, ele funciona como um sistema operativo, com o objetivo principal da criação de uma plataforma global de partilha e desenvolvimento de trabalhos da robótica. O ROS permite utilizar projetos de outros, que usem código aberto e também disponibiliza ferramentas e bibliotecas próprias. Estas ferramentas são exclusivas para algumas linguagens de programação, como o Python, Gazebo e OpenCV.

O sistema operacional do robô (ROS) é uma estrutura flexível para escrever software robótico. É uma coleção de ferramentas, bibliotecas abstração de hardware, device drivers, visualizadores, transmissão de mensagens, gerenciamento de pacotes e convenções que visam simplificar a tarefa de criar um comportamento de robô complexo e robusto em uma ampla variedade de plataformas robóticas (ROS, 2014).

Como resultado, o ROS foi construído desde o início para incentivar o desenvolvimento de software de robótica colaborativa. Por exemplo, um laboratório pode ter especialistas em mapear ambientes internos e contribuir com um sistema de classe mundial para produzir mapas.

Outro grupo pode ter especialistas em usar mapas para navegar e, no entanto, outro grupo pode ter descoberto uma abordagem de visão computacional que funciona bem para reconhecer pequenos objetos em desordem. O ROS foi projetado especificamente para grupos como esses para colaborar e desenvolver o trabalho uns dos outros (ROS, 2014).

Figura 18 - Aplicação do ROS.

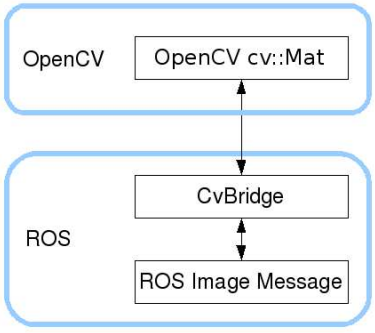
Fonte: ROS (2014).

Na Figura 18 são apresentadas as possíveis junções que os softwares podem realizar para executar um projeto. Neste caso é apresentado o estudo de um cientista, em que necessitará do uso de softwares que auxilie na pesquisa geográfica, outro para cálculo estatísticos, ao mesmo tempo um outro software voltado aos estudos dos planetas e satélites, e um para impressão e verificação de base de dados, visando atender um único objetivo.

2.7.1 Interação com o ROS

A interação entre o OpenCV e o ROS é possível através do CvBridge, que é uma biblioteca do ROS que fornece uma interface entre o ROS e o OpenCV, isto é, converte uma imagem do ROS para o OpenCV e vice-versa. Conforme o esquema apresentado na Figura 19.

Figura 19 - Interação entre OpenCV e ROS



Fonte: ROS (2014).

A imagem produzida no ROS, pode ter acesso ao OpenCV para exploração das ferramentas de visão computacional, assim como as mensagens produzidas pelo

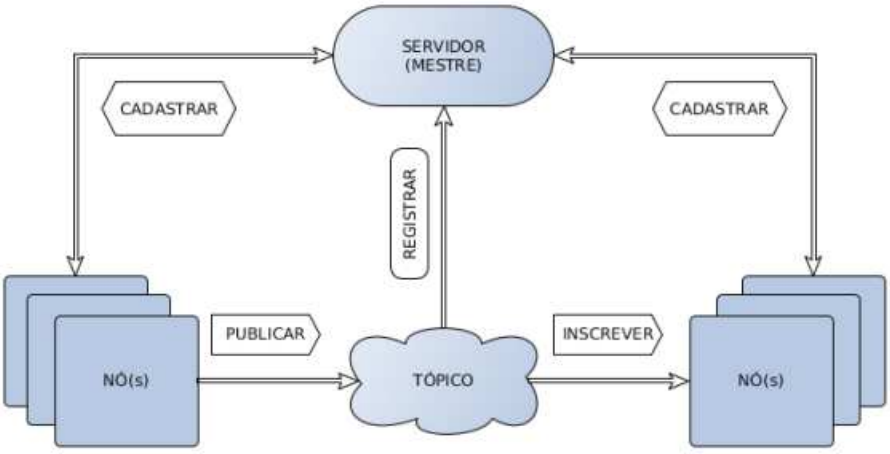
OpenCV pode ser transmitida para ROS. Esta biblioteca pode ser encontrado no pacote cv_bridge na pilha vision_opencv.

Conforme Rocha (2019), os componentes básicos da estrutura de comunicação do ROS são:

- **Servidor (mestre):** é o responsável por inicializar as dependências, bibliotecas do ROS e o sistema de troca de mensagens.
- **Nós:** são arquivos executáveis ou rotinas desenvolvidas para realizar determinadas tarefas, os quais utilizando as bibliotecas do ROS para estabelecer a comunicação com outros nós inscritos no mesmo tópico, publicar ou assinar tópicos, bem como prover ou usar serviços.
- **Tópicos:** são estruturas que armazenam tipos específicos de mensagens publicadas pelos nós, as quais podem ser vistas por toda a aplicação. Dessa forma, sempre que o nó publicar novas mensagens em determinado tópico, os nós que estejam inscritos irão receber a informação referente aquela mensagem.
- **Serviços:** podem ser vistos como outra forma pela qual os nós podem se comunicar uns com os outros, porquanto permitem que os nós enviem e recebam solicitações de mensagens e serviços.

A comunicação entre os dados de imagem processados no OpenCV com o ROS, simultaneamente com o simulador Gazebo, é realizada através de uma estrutura de comunicação, apresentado na Figura 20.

Figura 20 - Estrutura de Comunicação com o ROS



Fonte: Adaptado de Rocha (2019).

3 MATERIAIS E MÉTODOS

Para viabilizar nosso trabalho, foram necessários alguns hardwares como a placa Raspberry PI, o Arduino Uno, o manipulador robótico, a microcâmera webcam, notebook e outros acessórios. Além do Sistema Operacional Robótico (ROS), a biblioteca OpenCV, a plataforma de virtualização RVIZ, com o suporte do Moveit, e a plataforma de simulação Gazebo.

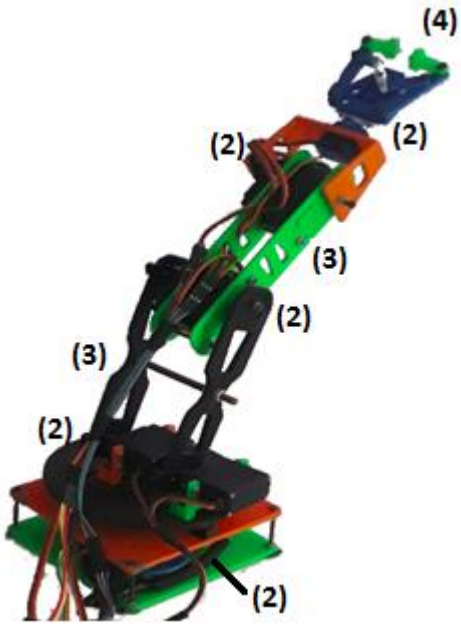
3.1 HARDWARE

As partes essenciais de hardware, necessário para execução dos experimentos e funcionalidade do projeto se dá pelos seguintes itens:

3.1.1 Manipulador Robótico

A principal finalidade do manipulador robótico é dar suporte ao estudo do sistema de visão computacional, e através da plataforma ROS, controlar seus posicionamentos. O manipulador utilizado neste trabalho é mostrado na Figura 21, com 5 graus de liberdade, impresso em 3D, modelo reformulado do robô ARM5 da WR kits.

Figura 21 - Manipulador Robótico.



Fonte: Autoria própria (2019).

O braço é constituído de uma base (1), juntas (2) e elos (3), e um efetuador com função de pinça (4), possuindo então 5 graus de liberdades, consistindo na rotação da base em torno do eixo Z, as articulações com rotação em torno do eixo Y, para estas quatro articulações mencionadas, os servomotores responsáveis pelo funcionamento das articulações são apresentados na seção 3.1.6.

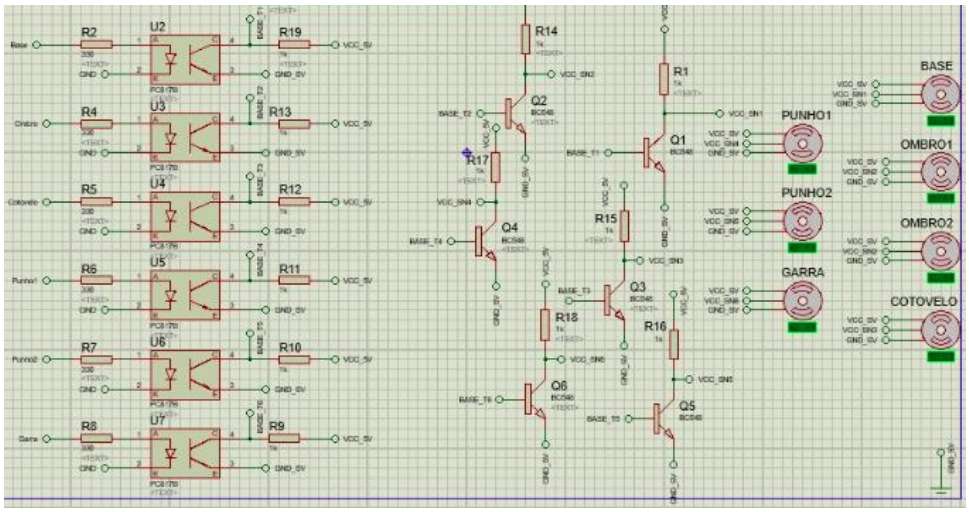
3.1.2 Computador

Notebook Acer Aspire E1 Series, equipado com processador Intel i3-4010U CPU @ 1.70GHz × 4 e 8GB de memória RAM;

3.1.3 Placa de Comunicação com os Servomotores

Para as saídas dos canais de comunicação, desenvolvemos um circuito, usando optoacopladores, com a finalidade de isolar o sinal PWM que o microcontrolador Raspberry Pi 3 B, que será apresentado na seção 3.1.5. Pois é ele quem enviará essas posições, em seus respectivos tempos, para cada servomotor, não permitindo que tenha algum retorno para os pinos do controlador e fornecendo alimentação própria para os motores. O circuito para este isolamento é mostrado Figura 22.

Figura 22 - Circuito Simulado pelo Proteus.



Fonte: Autoria própria.

3.1.4 Webcam

Webcam é uma micro câmera que deve ser ligada a um computador, criada para uso da internet, pode ser usada em videoconferências, produção de vídeos e imagens. Com exceção das que são embutidas nos notebooks, são portáteis e na grande maioria são conectados através de um cabo USB. Alguns itens são altamente relevantes na escolha de uma webcam (CARVALHO, 2011):

- Webcam com sensor CCD;
- Possibilidade de usar altas taxas de gravação de vídeo (ideal entre 5 e 30fps);
- Baixa compressão de imagens (alta qualidade), formatos de vídeo diversificados (.AVI, sequência .BMP, RAW, etc.) e boa resolução (ideal 640x480px);
- Que tenha lente de fácil remoção ou que se tenha facilidade em desmontar e montar a caixa de lentes;
- Que tenha, ou permita, o uso de softwares de fácil manuseio com funções de ajuste dos parâmetros básicos da câmera (exposição, ganho e controle de cores) plenamente funcionais.

Figura 23 - Webcam PISE.

Fonte: Gigatech (2019).

A câmera utilizada neste projeto é a webcam a PISE webcam, mostrada na Figura 23, com as seguintes especificações:

- Resolução máxima de 1.3 Megapixels
- Foco = 3.85mm.
- Interface de conexão USB 2.0.
- Taxa de quadros de 320 x 240 a 30 qps (CIF) ou 640 x 480 a 15 qps (VGA).

3.1.5 Raspberry Pi

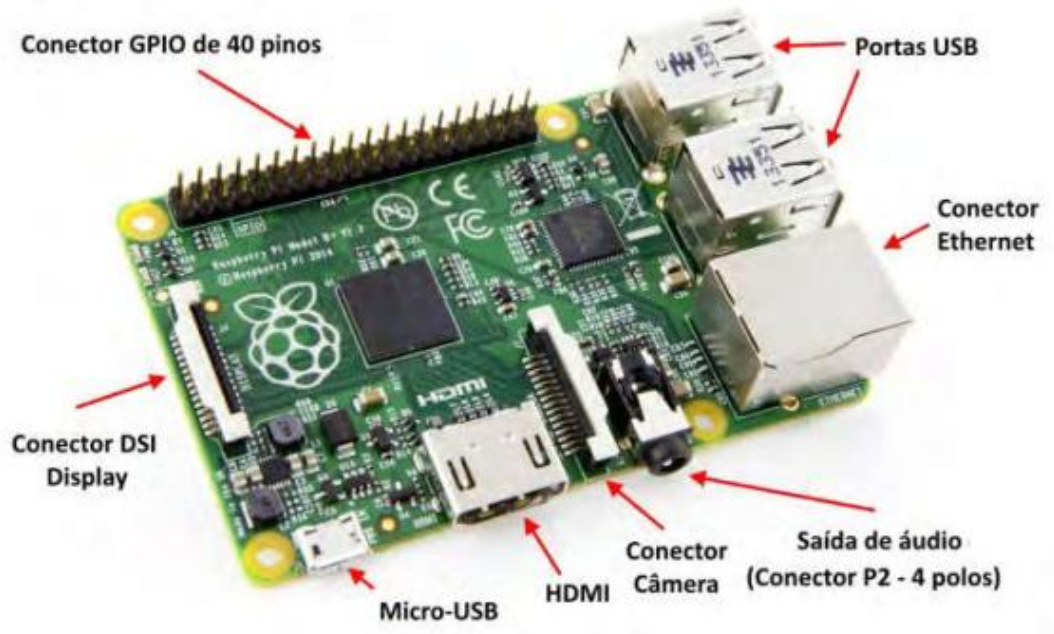
O Raspberry Pi é um computador do tamanho de um cartão de crédito, que se conecta a um monitor, e usa um teclado e um mouse padrão. Foi desenvolvido no Reino Unido pela Fundação Raspberry Pi. Possui os mesmos recursos que um computador desktop, como navegação na internet, reprodução de vídeo, fazer planilhas, processamento de texto, etc. Além do mais, o Raspberry Pi tem a capacidade de interagir com o mundo exterior através da conexão GPIO (General Purpose Input/Output), e tem sido usado numa ampla gama de projetos digitais como um sistema embarcado (EBERMAM; et al., 2017).

Conforme relatos de Ebermam [et al.] (2017), o nome Raspberry foi escolhido por uma equipe. “Raspberry” é a fruta framboesa; a escolha segue a tradição da época, de colocar nome de frutas em empresas e em computadores, como a Apple, Tangerine, Apricot, entre outras. “PI” é uma abreviação de Python, a linguagem de programação mais indicada para o aprendizado em programação no Raspberry Pi.

As linguagens de programação que acompanham o Raspberry são: Scratch, ferramenta para o ensino inicial de lógica de programação de forma lúdica, o Python que é uma linguagem mais avançada, voltada ao desenvolvimento de projetos robóticos e de cluster, e o Pygames, que é uma biblioteca de rotinas em Python que trabalha com orientação a objetos desenvolvida para facilitar a criação de jogos. As linguagens C, Ruby, Java e Perl, também podem ser utilizadas para programação no Raspberry.

Na Figura 24 são apresentados os principais componentes da placa de um Raspberry Pi B+, como as diversas portas para áudio, vídeo e dados: HDMI, USB, Ethernet e GPIO.

Figura 24 – Conectividade do Raspberry Pi Modelo B+.



Fonte: Aplicações Práticas de Sistemas Embarcados Linux utilizando Raspberry Pi (JUCA, 2018).

Slot para cartão de memória SD (Secure Digital): Não há disco rígido no Pi, portanto, tudo é armazenado em um cartão de memória SD.

Fonte de alimentação (entrada de energia): O Raspberry não trabalha com interruptores de alimentação de energia. Para seu funcionamento é utilizado um cabo USB de 5V, compatível a um carregador de celular, através de uma porta USB de entrada na placa, que deverá funcionar apenas para receber a carga para o funcionamento do Raspberry Pi.

Processador: o mesmo processador de um iPhone 3G da Apple. Um processador de 700 MHz e de 32 bits, construído sobre a arquitetura ARM11 (sistema microprocessado no qual o computador é completamente encapsulado ou dedicado ao dispositivo ou sistema que ele controla). Chips ARM apresentam-se em uma variedade de arquiteturas com diferentes núcleos configurados para fornecer diferentes capacidades, com preços diferentes. O modelo B tem 512 MB de memória RAM e o modelo A tem 256 MB.

Porta HDMI (High-Definition Multimedia Interface): Por meio da porta HDMI se consegue a transmissão, com alta qualidade, áudio e vídeo por um só cabo até um monitor. O Raspberry Pi suporta cerca de catorze tipo de resoluções de vídeo. Através

de adaptadores externos pode-se converter vídeo em DVI e jogar a imagem para monitores de modelo antigo.

Conector de Interface Serial para Câmera: Por meio dessa porta é possível a conexão de um cabo serial de câmera para transmitir a imagem a um monitor.

Porta Ethernet: Diferentemente do modelo A, e o modelo B do Raspberry Pi possui porta Ethernet (arquitetura para interconexão para redes locais) para o padrão RJ-45. Pode-se também utilizar redes Wi-Fi, mas para isso deve-se utilizar uma das portas USB e nela conectar o "Dongle Wi-Pi (Wireless Internet Platform for Interoperability)".

Portas USB (Universal Serial Bus): O modelo A possui apenas uma porta USB; o modelo B possui duas portas UBS 2.0; a versão mais atual, o modelo B+, possui quatro portas USB 2.0. A quantidade de portas USB ainda pode ser expandida com um hub USB para a utilização de mais periféricos.

Conectores P2 e P3: Essas duas linhas perfuradas na placa são os conectores JTAG (Joint Test Action Group), utilizados para testes de chips Broadcom (P2) e o de rede LAN9512 (P3). Por serem de natureza proprietária, esses conectores dificilmente serão utilizados em seus projetos.

LED: O status de funcionamento da placa é mostrado em cinco LEDs, cujos significados estão detalhados na Tabela 1.

Tabela 1 - Descrição dos leds da placa Raspberry Pi.

LED	Cor	Descrição
ACT	Verde	Acende quando o cartão SD é acessado.
PWR	Vermelho	Conectado à alimentação de 3,3 V.
FDX	Verde	ON (ligado) se o adaptador de rede é full-duplex.
LNK	Verde	Luz indicando atividade de rede.
100	Amarelo	ON (ligado) se a conexão de rede for de

Fonte: EBERMAN (2017).

Saída de áudio analógico: Destina-se aos reprodutores de áudio (amplificadores, caixas de som etc.). Essa saída possui apenas 3,5mm e conduz cargas de alta impedância. Caso utilize fone de ouvido ou alto-falante, nessa porta

sem alimentação elétrica independente haverá queda na qualidade do áudio, diferentemente do cabo HDMI, que, ao transmitir o áudio para o monitor, não apresenta nenhuma perda de qualidade.

Saída de vídeo: Trata-se de um conector do tipo RCA para fornecer sinais de vídeo composto NTSC (Sistema de Televisão Analógico) ou PAL (Padrão de codificação de cores para televisão analógica). Essa saída de vídeo tem baixa qualidade, sendo preferível usar a porta HDMI se possível.

Pinos de Entrada e Saída (GPIO) para uso geral: São 26 pinos GPIO (General Purpose Input/Output) para comunicação com outros dispositivos externos; podem ser usados, por exemplo, para controle de equipamentos de automação.

Conector de Interface Serial do Display (DSI): Esse conector foi projetado para a utilização de um cabo flat de 15 pinos para a comunicação com uma tela LCD ou OLED ou uma WebCam.

Pode ser conectado a um monitor de computador ou TV, utilizando um teclado padrão e um mouse. Além de oferecer suporte a outros periféricos.

O *Raspberry* Pi utiliza o sistema operacional *Raspibian ou Ubiquity Robotics*, uma versão não oficial do *Debian Wheezy5* otimizada para funcionar com instruções avançadas da arquitetura ARM v6 presente no processador do *Raspberry Pi*.

3.1.6 Servomotores

Servos motores são máquinas elétricas, que tem como característica principal a precisão de posicionamento de seu eixo. Essas máquinas possuem sistemas de controle interno e por terem tamanhos reduzidos são muito utilizadas em aplicações didáticas, robótica e diversas outras aplicações cotidianas (SANTOS, 2015).

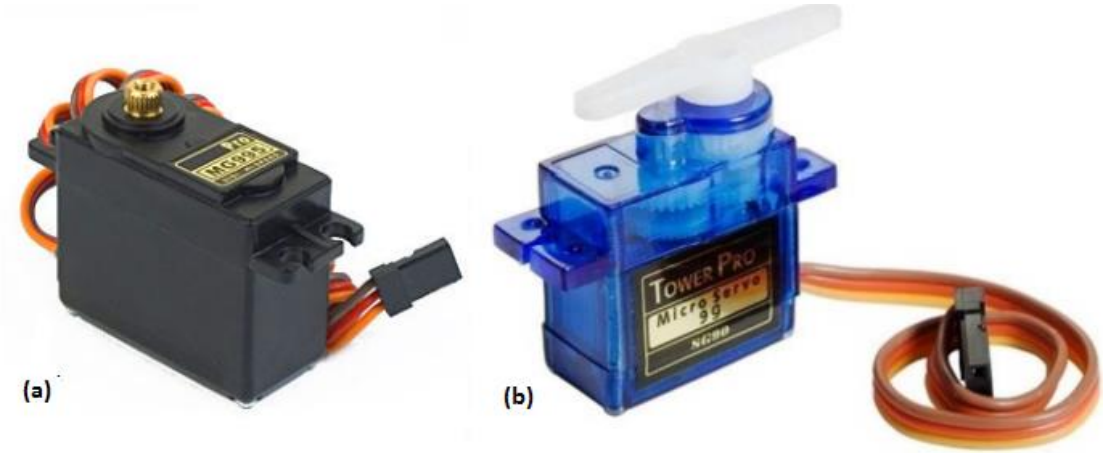
Existem vários tipos de servos motores, cada um correspondente à uma aplicação, conforme suas especificações técnicas, são eles: servo motor assíncrono de corrente alternada, servo motor síncrono de corrente alternada, servo motor síncrono de corrente continua, servo motor de passo, servo motor de relutância chaveada e servo motor de indução (MATOS, 2012).

Um servo motor possui um potenciômetro que possibilita a um sistema de controle interno o monitoramento do seu eixo de saída. O sistema de controle de um

servo é em malha fechada, isto é, compara a saída com a entrada e se necessário faz a compensação até atingir o posicionamento determinado (SANTOS, 2015).

Neste projeto foram utilizados os servomotores MG995 e SG90, conforme mostrado na Figura 25.

Figura 25 - Servomotores (a) MG995 e (b) SG90.



Fonte: ALLDATASHEET (2019).

As especificações destes servos motores são apresentadas a seguir (ALLDATASHEET, 2019):

Servo motor MG995

- Peso: 55 g.
- Dimensão: 40,7 x 19,7 x 42,9 mm aprox.
- Torque de parada: 8,5 kgf · cm (4,8 V), 10 kgf · cm (6 V).
- Velocidade de operação: 0,2 s / 60º (4,8 V), 0,16 s / 60º (6 V).
- Tensão de operação: 4,8 V a 7,2 V.
- Largura da banda morta: 5 µs.
- Projeto de rolamento de esferas duplo estável e à prova de choque.
- Faixa de temperatura: 0 ºC - 55 ºC.

Servo motor SG90

- Peso: 9 g.
- Dimensão: 22.2 x 11,8 x 31 mm aprox.
- Torque de parada: 1,8 kgf.cm (4,8 V), 10 kgf.cm (6 V).
- Velocidade de operação: 0,1 s / 60º (4,8 V), 0,16 s / 60º (6 V).

- Tensão de operação: 4,8 V (~5V).
- Largura da banda morta: 10 µs.
- Projeto de rolamento de esferas duplo estável e à prova de choque.
- Faixa de temperatura: 0ºC - 55ºC.

Para o funcionamento do manipulador robótico, foram utilizados 6 servo motores. Um MG995 mostrado na Figura 25 (a), para a rotação da base. Devido ao peso da estrutura do manipulador, que exige um torque maior, a junta posterior a junta da base, que representa o ombro, fez-se necessário 2 servos motores MG995. Para as juntas que representam o cotovelo e punho, foram utilizados 1 servo motor MG995 para cada junta.

Já para a rotação do efetuador pinça e para abertura ou fechamento da mesma foram utilizados um servo motor SG90 Figura 25(b) para cada função.

Como estes servos são modelos estilo hobby, ele opera com variação de PWM (Modulação por Largura de Pulso) não fornecendo informações de posicionamento, velocidade ou torque.

3.2 SOFTWARES

Para execução do projeto, serão utilizados os softwares voltados as práticas de robótica em um nível didático, de fácil compreensão e implementação:

3.2.1 ROS Controlador

Controlador para executar trajetórias de espaço conjunto em um grupo de juntas. As trajetórias são especificadas como um conjunto de pontos de referência a serem alcançados em instantes de tempo específicos, que o controlador tenta executar, assim como o mecanismo permite. Os pontos de referência consistem em posições e, opcionalmente, em velocidades e acelerações (ROS, 2014). Neste trabalho será utilizado trajetória por posição, a qual irá trabalhar com a representação Linear (somente a posição é especificada, garantindo a continuidade no nível de posição).

O controlador é modelado para trabalhar com vários tipos de hardware. Atualmente, as articulações com interfaces de posição, velocidade e esforço são

suportadas. Para juntas de posição controlada, as posições desejadas são simplesmente encaminhadas para as articulações (ROS, 2014).

Existem dois mecanismos para enviar trajetórias para o controlador: por meio de interface de ação ou de interface de tópico. Ambos usam mensagem trajectory_msgs/JointTrajectory para especificar trajetórias e exigem a especificação de valores para todas as juntas do controlador (em oposição a apenas um subconjunto) se “allow_partial_joints_goal” não estiver definido como *True* (ROS,2014).

Figura 26 - Descrição mínima, interface de posição

```
head_controller:
  tipo: "position_controllers / JointTrajectoryController"
  articulações:
    - head_1_joint
    - head_2_joint
```

Fonte: ROS, 2014.

A Figura 26 mostra um exemplo de descrição de comando para a configuração do controlador.

3.2.1.1 RVIZ

O ROS possui uma ferramenta de visualização 3D chamada RVIZ que permite à representação em imagem 3D, de praticamente qualquer plataforma robótica, respondendo em tempo real ao que acontece no mundo real.

O RVIZ pode ser usado para exibir leituras de sensores, dados retornados por visão estereoscópica (Cloud Point), SLAM (localização e mapeamento simultâneos), evitando obstáculos, etc. Essa ferramenta também possui muitas opções de configuração.

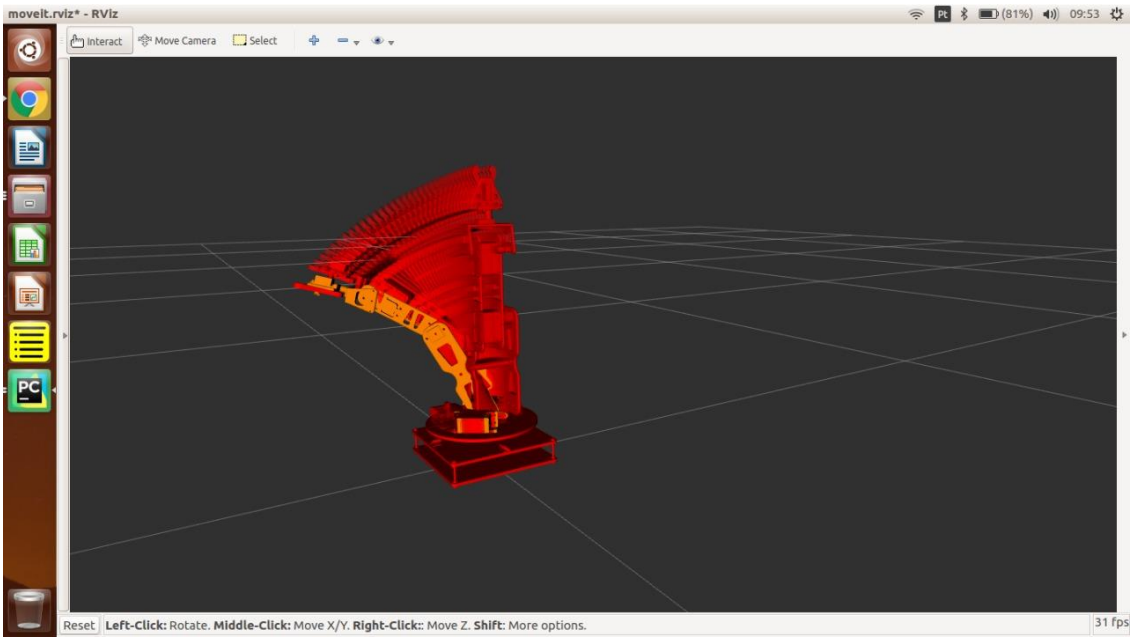
Para gerar cada modelo de robô específico, é necessário editar em um arquivo XML e escrever em URDF (Unified Robot Description Format, traduzido: Formato de Descrição do Robô Unificado), onde especifica as dimensões do robô, os movimentos das articulações, parâmetros físicos como massa e inércia, etc.

No caso de robôs com muitas articulações, o ROS possui outra ferramenta chamada TF[5], que é uma biblioteca que facilita a elaboração nesses casos. De qualquer forma, o ROS já criou modelos para testar com o RVIZ.

3.2.1.2 Moveit

O Moveit é uma biblioteca que gera um script de planejamento de trajetória do ROS, o código gerado serve para a movimentação de um manipulador robótico, dando suporte ao visualizador RVIZ, ou seja, sua tarefa básica é fornecer as trajetórias necessárias para que o robô coloque seu efetuador final em um determinado local.

Figura 27 - Tela do RVIZ com aplicação do Moveit



Fonte: Autoria própria (2019).

O resultado de um planejamento de movimento executado pelo Moveit, é a sequência de movimentos que todas as articulações do manipulador devem realizar para passar do local atual para o desejado. Na Figura 27, é apresentado uma janela com a utilização do moveit, demonstrando sua aplicação em um manipulador robótico.

[5] **TF –** Significa transformações, quadros de coordenadas.

3.2.2 OpenCV

Open Source Computer Vision Library (OpenCV) é uma biblioteca multiplataforma de livre uso, tanto acadêmico e comercial. Originalmente teve seu desenvolvimento iniciado pela Intel em 1999 (OPENCV, 2015).

A OpenCV é um conjunto de ferramentas de programação para desenvolvimento de aplicações em Visão Computacional. Ela engloba também outro conceito importante, especialmente no meio acadêmico, o software livre. A biblioteca é completamente OpenSource, e é distribuída gratuitamente, aberta a colaborações de qualquer indivíduo ou empresa voluntários. A gratuidade da OpenCV, o baixo custo do poder de máquina e a crescente qualidade das câmeras torna possível o desenvolvimento de sistemas sofisticados de Visão, com baixo investimento e custo de operação. A utilização de câmeras pode inclusive substituir outros sensores e sistemas mais caros, complexos e menos genéricos. Isso evidencia não só o crescimento da área da Visão Robótica, mas também a tendência de aproximação dos computadores à forma como os humanos entendem o ambiente ao seu redor (DELAI; COELHO, 2012).

Conforme Orlandini (2012), a OpenCV possui cerca de 500 funções relacionadas a várias áreas da visão computacional, como: funções de processamento de imagens, detecção de movimento e rastreamento, reconhecimento de padrões em imagens e calibração de câmera. Essas funções, são de alto-nível e tornam a resolução de problemas complexos em visão computacional mais fácil. O download dessa biblioteca está disponível em <http://opencv.willowgarage.com/wiki>.

A biblioteca tem mais de 2500 algoritmos otimizados, o que inclui um conjunto abrangente de algoritmos de visão computacional e de aprendizado de máquina clássicos e de última geração.

Esses algoritmos podem ser usados para detectar e reconhecer rostos, identificar objetos, classificar ações humanas em vídeos, rastrear movimentos de câmera, rastrear objetos em movimento, extrair modelos 3D de objetos, produzir nuvens de pontos 3D a partir de câmeras estéreo, unir imagens para produzir alta resolução imagem de uma cena inteira, encontrar imagens semelhantes de um banco de dados de imagens, remover olhos vermelhos de imagens tiradas usando flash,

acompanhar movimentos dos olhos, reconhecer paisagens e estabelecer marcadores para sobrepô-lo com realidade aumentada, etc. (OPENCV, 2015).

3.2.3 Gazebo

Conforme Gazebosim (2018), o gazebo é um simulador 3D de robô, com a capacidade de projetar robôs, rodar algoritmos, fazer testes de regressão e treinar sistemas de inteligência artificial com cenários da realidade. Estes cenários podem ser desenvolvidos com diferentes formas de obstáculos.

A simulação de robô é uma ferramenta essencial na caixa de ferramentas de todos os roboticistas. Um simulador bem projetado torna possível testar rapidamente algoritmos, projetar dentre outras funcionalidades, usando cenários realistas. O Gazebo oferece a capacidade de simular com precisão e eficiência as populações de robôs em ambientes internos e externos complexos. Ao seu alcance está um motor robusto de física, gráficos de alta qualidade e interfaces programáticas e gráficas convenientes (Gazebosim, 2018).

Conforme Vagallis (2016), a Open Source Robotics Foundation (OSRF) oferece a solução para ambos os problemas com o seu simulador de robô Gazebo gratuito e de código aberto.

> "Usado tanto por amadores quanto por profissionais, é possível construir modelos que agem como robôs reais e se movem em seu próprio mundo, governados por seus quatro motores de física de última geração (sendo o padrão *ODE Open Dynamics Engine*, para simulação de dinâmica de corpo rígido)."

O Gazebo é como uma ferramenta de CAD, capaz de criar robôs. Um pré-requisito é o Sistema Operacional de Robô (ROS) no qual o Gazebo é executado. O ROS é uma estrutura para escrever software robótico e fornece ferramentas, bibliotecas, drivers e convenções para criar um comportamento robusto e complexo de robôs. Sua integração é explicada com mais detalhes no *white paper* de visão geral de integração ROS to Gazebo.

4 DESENVOLVIMENTO

Neste tópico são apresentados os procedimentos realizados para a construção do projeto, desde a instalação dos softwares, descrição dos comandos de execução, a montagem e composição dos hardwares, até a funcionalidade do projeto visando atender os requisitos propostos.

4.1 INSTALAÇÃO DOS SOFTWARES

Para execução do projeto, fez-se necessário a preparação do notebook e a configuração do sistema operacional em Linux, sugerido para a instalação do ROS, e os demais softwares composto na plataforma.

4.1.1 ROS

O ROS é a primeira plataforma a ser instalada, e por ser uma plataforma *OpenSource*, sua instalação é realizada através do acesso no link[6] de execução do download. Após entrar no site, são disponibilizadas diversas instruções de instalações e configurações do sistema.

Para execução do projeto, foi instalado o ROS *Kinetic*, com o objetivo de melhor atender a versão do Sistema Operacional Ubuntu 16.04 (*Xenial*), pela qual é mais direcionado.

Além da praticidade, o ROS disponibiliza um tutorial riquíssimo em informações e instruções, desde a criação do ambiente de trabalho, como a descrição de comandos e bibliotecas.

4.1.2 Python

O Python geralmente já vem instalado no Ubuntu. Caso haja a necessidade de instalação ou atualização de versão, são disponibilizadas informações importantes no site do Python, através do link[7] e seguir os procedimentos de instalação.

---

[6] ROS <http://wiki.ros.org/pt_BR/ROS/Tutorials/InstallingandConfiguringROSEnvironment>.
[7] PYTHON <https://python.org.br/instalacao-linux/>

Instruções para o uso desta ferramenta também podem ser facilmente encontradas em livros didáticos, artigos em sites, vídeo-aulas demonstrativas de aplicações práticas (PYTHON, 2019). Por ser também uma ferramenta de código aberto. São realizadas constantes melhorias e atualizações de versões e disponibilizadas para melhoria do desempenho nas programações.

4.1.3 OpenCV

O OpenCV é uma biblioteca que já faz parte do pacote ROS, sendo a versão usada no trabalho a OpenCV 2.4.8. No Sistema Operacional de código aberto Ubuntu 16.04.6 LTS, com o Python 2.7 já instalado, basta baixar a biblioteca OpenCV. Os detalhes para a instalação das bibliotecas e pacotes necessários são disponibilizados acessando o link[8].

Caso haja necessidade da instalação de alguma outra biblioteca ou pacotes para execução de programas, basta solicitar a instalação com a descrição e seguir o mesmo procedimento que a instalação do OpenCV, descrito anteriormente.

Existem vários ambientes de desenvolvimento integrado (*IDE-Integrated Development Environment*), que são conjuntos de ferramentas que tem o objetivo de oferecer ao desenvolvedor tudo o que é necessário para seu trabalho: editor, corretor, *debugger*, compilador/interpretador, etc. O principal objetivo do IDE é proporcionar conforto, eficiência e desempenho a projetos de desenvolvimento de todos os tipos e tamanhos.

4.1.4 Gazebo

O Gazebo é um ambiente de simulação integrado ao ROS, e sua instalação pode ser realizada acessando o link[9] para instalação a partir da fonte ou a partir de pré-compilar o Ubuntu seguindo as instruções disponibilizadas no link[10]. O mais fácil e rápido é instalá-lo a partir de pacotes, mas instalar a partir do código significa que você pode depurar e enviar correções de bugs com maior facilidade.

---

[8] OPENCV <https://docs.opencv.org/master/d2/de6/tutorial_py_setup_in_ubuntu.html>.
[9] GAZEBOSIM <http://gazebosim.org/>.
[10] GAZEBOSIM <http://gazebosim.org/tutorials?cat=install>.

Na página do Gazebo são disponibilizados tutoriais e diversas instruções para instalação e configurações (GAZEBOSIM, 2018). O conjunto de pacotes ROS para interface com o Gazebo está contido em um novo meta pacote (versão catkin das pilhas) chamado gazebo_ros_pkgs.

4.1.5 RVIZ

O RVIZ é um visualizador em 3D para a estrutura do ROS, logo para sua instalação, já com o ROS instalado, pode-se fazer o *download* das fontes RVIZ em sua área de trabalho ou na sobreposição.

Primeiramente é necessário satisfazer as dependências do sistema, para isso é digitado o comando: *Rosdep install rviz.*

Em seguida é construído o visualizador com o comando: *rosmake rviz.*

Com o visualizador criado, o próximo passo é iniciá-lo como o seguinte comando: *rosrun rviz rviz.*

Por ser iniciado pela primeira vez, a janela do RVIZ estará vazia. Mas, através desta janela, já é possível criar um ambiente de visualização, que são explicados passo a passo, acessando o link[11] do ROS.

4.1.5.1 Moveit

No link[12] é disponibilizado o passo a passo de instalação para o sistema operacional Linux (nas diferentes versões do Ubuntu), desde a instalação, configuração e até os passos de utilização, inclusive as instruções para utilização dos comandos.

No caso do ROS Kinetic já instalado e a versão do Ubuntu 16.04, basta satisfazer as dependências do sistema com o comando: *rosdep*

Em seguida solicitar a instalação do moveit para esta versão do ROS, com o comando: *sudo apt-get install ros-melodic-moveit*.

Na área de trabalho do RVIZ, essa biblioteca já pode ser explorada.

---

[11] RVIZ< http://wiki.ros.org/rviz/UserGuide >.
[12] MOVEIT <https://moveit.ros.org/install/source/>.

Além do programa acima já mencionado, se faz necessário a instalação de bibliotecas para geração de um solucionador cinemático para robôs com menos de 6 graus de liberdade. Para seguiu-se o tutorial de instalação do link[13].

4.2 SISTEMA INTEGRADO AO ROS

Como apresentado nas seções 2.7.1, o ROS é a plataforma responsável pela interação entre os laços de informações simultâneos entre o detector de obstáculos (OpenCV), o manipulador robótico e o simulador Gazebo. O objetivo é a tomada de decisões e execução dos comandos correspondentes às condições propostas para as situações captadas em cada instante.

4.2.1 Integração do OpenCV com o ROS

Para isso um nó precisa publicar e assinar uma informação em um tópico específico no ROS, necessitando que o mesmo apresente uma estrutura padrão. Desta forma o código do nó do ROS para o reconhecimento de objeto, deve conter as instruções seguintes:

- *rospy.init_node('follower_p'):* esta instrução inicializa o nó ROS para o processo. Os nomes têm propriedades importantes no ROS, portanto, eles devem ser únicos**.**
- *rospy.Subscriber*("/image_view/output", Image, image_callback): este é uma classe assinante, que se inscreve em um tópico, para receber os dados relativos ao tópico.
- *rospy.Publisher* ('detected, Point, queue_size=1): esta é uma classe publicadora, que envia mensagens ao tópico, onde se tem dentro do parênteses, o nome do tópico, o tipo de mensagem e o tamanho da mesma.

[13] TUTORIAL OpenRave: <https://github.com/yijiangh/Choreo/blob/7c98fd29120e5ce75d2b8ed17bc49488ad983cb6/framefab_robot/abb/framefab_irb6600/framefab_irb6600_support/doc/ikfast_tutorial.rst>.

4.3 ELABORAÇÃO DOS PROGRAMAS

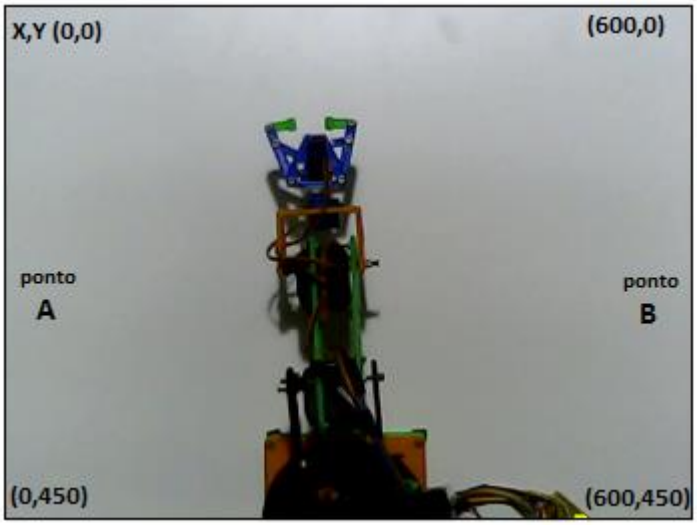
Após a instalação de todas as ferramentas necessárias, são realizadas as programações, utilizando as diferentes plataformas para atender todos os processos do trabalho. Primeiramente, é encontrado um programa para o estudo do sistema de visão computacional, para a detecção de obstáculo utilizando a biblioteca OpenCV, posteriormente a programação da movimentação do robô no ROS, com o auxílio da plataforma de visualização RVIZ e manipulação Moveit, e por último a simulação dos movimentos do robô pelo Gazebo.

4.3.1 Detecção de obstáculo no OpenCV

Após o estudo da visão computacional alguns métodos para processamento de imagem foram verificados, para obter o reconhecimento de objeto. Visto que a biblioteca OpenCV dispõe de algoritmos prontos, os quais variados métodos de alta complexidades são explorados dentro de pacotes.

A primeira etapa a ser realizada para obtenção das imagens desejadas é a calibração do sistema de visão computacional, fundamental para encontrar a correlação entre as medidas dos objetos e/ou as distâncias entre eles, no mundo real, e as respectivas dimensões das imagens adquiridas. Para isso foram estabelecidos os parâmetros de distância entre a câmera e a planta (solo), como o trabalho dá continuidade ao trabalho de Matos (2018), utilizou-se de uma parte de seu código de calibração para o OpenCV, ao qual considerando a área de captação da câmera, como sendo a área de monitoramento, determinando então as coordenadas do sistema, apresentado na Figura 28.

Figura 28 - Área de captura da webcam.



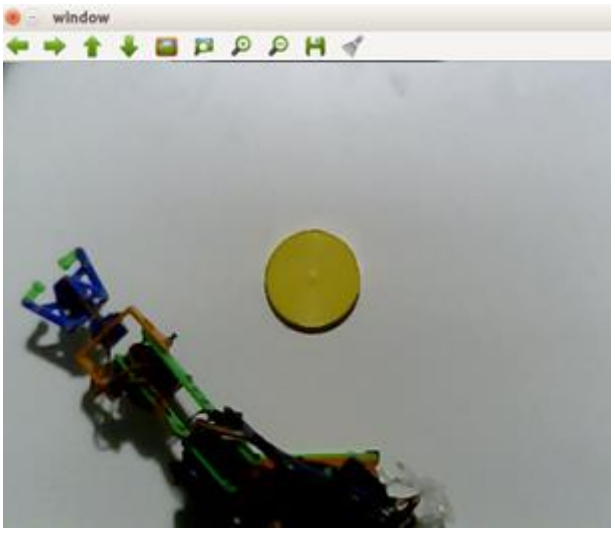
Fonte: Autoria própria (2019).

Conforme mostra a Figura 28, foi estabelecida uma área (600x450 pontos) que podem ser representados por um sistema de medida de comprimento ou pixels. Consequentemente a localização do objeto já pode ser informada através de um algoritmo que solicita a impressão dos posicionamentos do obstáculo detectado.

Para tanto, como o objetivo é detectar um obstáculo. Uma imagem típica da câmera é mostrada na Figura 29.

A técnica desenvolvida para detecção do obstáculo foi encontrar no fluxo de imagens parâmetros próximos da realidade, como o estabelecido em fabricas, que consiste em uma identificação por meio de sinalização nas pessoas ou equipamentos, que circulam meio ao ambiente de trabalho, máquinas e robôs. Logo, para tal identificação foi estabelecida uma cor (amarelo), devido à proximidade das cores já existentes nos sistemas de segurança.

Figura 29 - Uma visão típica da câmera ao encontrar um obstáculo



Fonte: Autoria própria (2019).

Com esses parâmetros já estabelecidos, desenvolveu-se os algoritmos, baseados em algoritmos já existentes, seguindo a programação dentro do ambiente de desenvolvimento no Python.

O primeiro passo para a descrição do programa de detecção é a importação dos pacotes necessários para o funcionamento do programa, conforme descrito a seguir:

import rospy - para que o ROS reconheça a linguagem Python.

from sensor_msgs.msg import Image - para aceitar a imagem da câmera.

from cv_bridge import CvBridge, CvBridgeError – converter a imagem do ROS ao OpenCV

import cv2 – importar a biblioteca OpenCV

import numpy - provê as diversas funções e operações matemáticas sofisticadas.

import Imutils – funções de conveniências para facilitar as funções básicas de processamento de imagem, como conversão, rotação, redimensionamento, esqueletização e exibição de imagens do Matplotlib.

Para enviar mensagem de reconhecimento, foram necessários os seguintes comandos:

from geometry_msgs.msg – mensagem de um ponto;

import Point

bridge = CvBridge( ) – aloca espaço memória para posteriormente utilizar os atributos que o *CvBridge* dispor.

Para o reconhecimento do objeto pela tonalidade definida no projeto, encontrou-se na biblioteca o comando com os limites utilizados para cor amarela. Baseado no estudo do espaço das cores RGB, na seção 2.4.1.1, o range foi ajustado para a tonalidade desejada, estabelecendo os limites inferiores e superiores a seguir:

```
yellowLower = numpy.array([25, 50, 50], numpy.uint8)
yellowUpper = numpy.array([32, 255, 255], numpy.uint8)
```

Após fazer as importações e atribuições, foi desenvolvido um algoritmo para o reconhecimento do objeto gerar um contorno retangular sobre ele.

Calculado o centróide, para que a localização seja impressa e enviada ao ROS, isto é, a partir do estudo da imagem pelo OpenCV, é enviada como resposta ao ROS através do CvBridge, por isso, foi solicitado anteriormente um espaço na memória.

É convertido a mensagem do Ros para o openCV, com a função:

```
cv2_img = bridge.imgmsg_to_cv2(Image, desired_encoding='bgr8')
```

Definindo a largura do quadro:

```
cv2_img = imutils.resize(cv2_img, width=600)
```

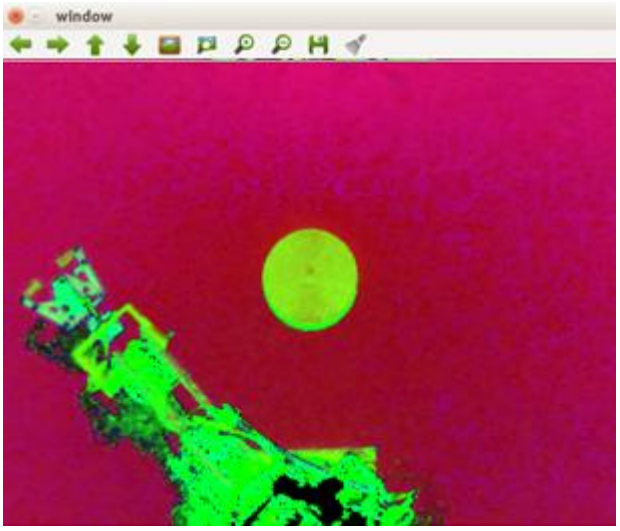
Convertendo espaços de cores de BGR para HSV:

```
hsv = cv2.cvtColor(cv2_img, cv2.COLOR_BGR2HSV)
```

A função cvtColor() produzirá uma imagem HSV mostrada na Figura 30*,* quando apresentada com a imagem RGB, da figura 14 anteriormente apresentada. Limitando a imagem HSV para obter apenas as cores amarelas, também usando a função inRange() do OpenCV para gerar a imagem binária conforme mostrado na Figura 31**.**

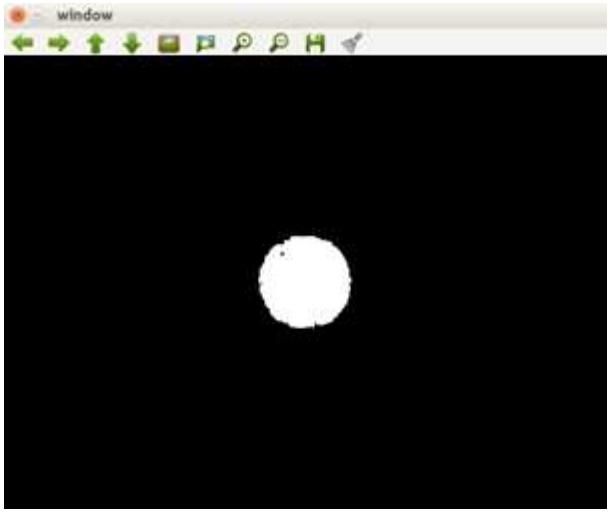
```
maskYellow = cv2.inRange(hsv, yellowLower, yellowUpper)
```

Figura 30 - Representação HSV de uma imagem da câmera com o objeto no centro



Fonte: Autoria própria (2019).

Figura 31 - Imagem binária obtida por filtro de matiz na imagem HSV.



Fonte: Autoria própria (2019).

Visando uma maior definição da cor, utilizou-se a função para erosão (*erode*), cuja função é remover os ruídos brancos. Porém, essa etapa reduz o objeto. Para que a área do objeto volte ao tamanho inicial, utilizou-se o comando de dilatação (*dilate*), conforme abaixo:

```
maskYellow = cv2.erode(maskYellow, None, iterations=2)
maskYellow = cv2.dilate(maskYellow, None, iterations=2)
```

Encontrando contornos para a parte detectada da imagem:

cntYellow = cv2.findContours(maskYellow.copy(), cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)[-2]

Para iniciar os cálculos do centróide, é necessário a atribuição inicial em zero:

centerYellow = 0

O próximo passo do algoritmo é o condicionamento da detecção ou não do obstáculo. Conforme descrito anteriormente é solicitado um contorno sobre o objeto com a tonalidade especificada. Logo, as condições a seguir são baseadas neste contorno gerado.

Portanto, o contorno inicia zerado. 'Se' houve a detecção, é gerado o contorno e a variável *cnt.Yelow* será maior que zero. 'Senão' a variável continua zerada e volta ao início, fazendo sempre a leitura dessas informações.

Para a condição de cnt.Yellow > 0, o programa tem as seguintes execuções:

```
print len(cntYellow)
cYellow = max(cntYellow, key=cv2.contourArea)
rectYellow = cv2.minAreaRect(cYellow)
boxYellow = cv2.boxPoints(rectYellow)
boxYellow = numpy.int0(boxYellow)
```

As funções acima tiveram como objetivo a criação de um retângulo mínimo para o contorno e imprimir na tela de visualização.

Após a criação do retângulo, encontra-se o centroide, calculando o momento da imagem, que é definida como uma média ponderada específica das intensidades de pixel da imagem determinada.

O centróide é determinado através das seguintes fórmulas:

$$C_x = \frac{M_{10}}{M_{00}} \quad \text{e} \quad C_y = \frac{M_{10}}{M_{00}}$$

Onde:

C_x e C_y as respectivas coordenadas **x** e **y** do centróide.

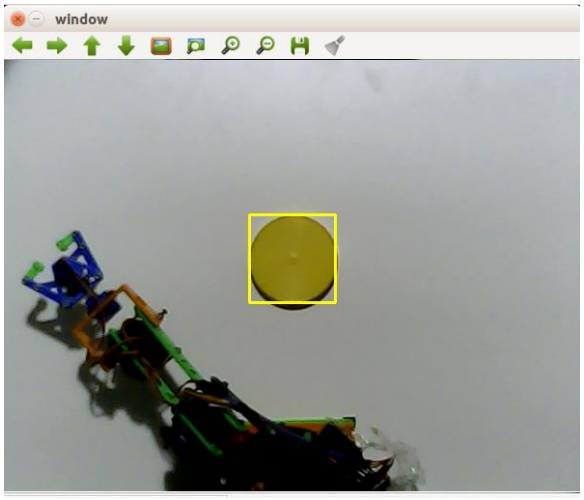
M : momento da imagem.

Partindo dessas considerações, é descrito para o programa:

```
MYellow = cv2.moments(cYellow)
   if MYellow['m00'] > 0:
       M = MYellow
       cx = int(M['m10'] / M['m00'])
       cy = int(M['m01'] / M['m00'])
       centerYellow = (cx, cy)
        cv2.drawContours(cv2_img, [boxYellow], 0, (0, 255, 255), 2)
```

Na Figura 32, podemos visualizar o obstáculo reconhecido e contornado pelo retângulo amarelo.

Figura 32 - Imagem do obstáculo reconhecido no OpenCV ROS.



Fonte: Autoria própria (2019).

Com estas etapas, tem-se todas as informações desejadas sobre a visão computacional. Os passos seguintes foram estabelecer a função de comunicação com o ROS, para o envio das coordenadas do objeto.

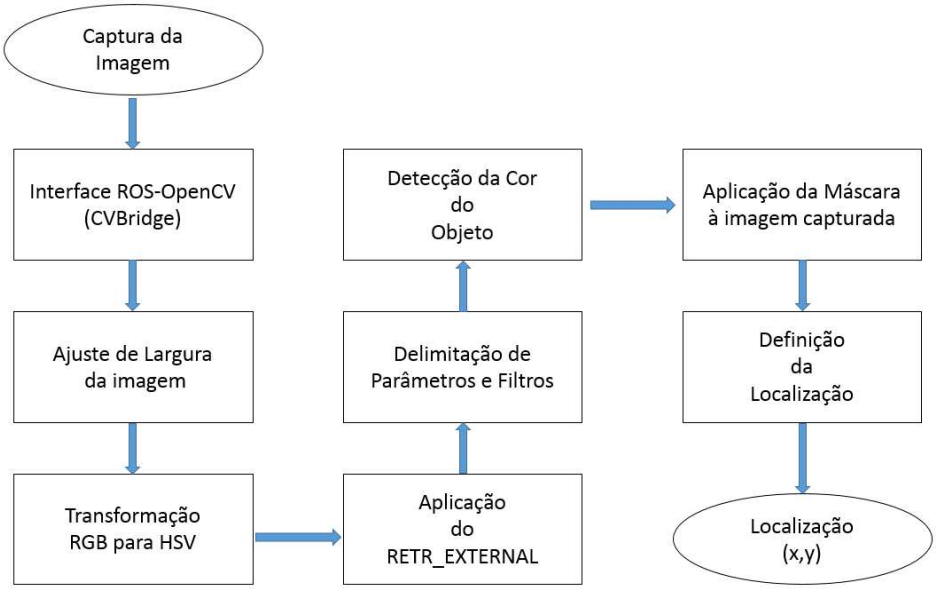
```
pub = rospy.Publisher('detected', Point, queue_size=1)
rate = rospy.Rate(10)  # 10hz
msg = Point()
```

```
msg.x = cx * 0.11666
msg.y = cy
#hello_str = Point()
print (centerYellow)
rospy.loginfo(msg)
pub.publish(msg)
rate.sleep()
```

Identificado e localizado o obstáculo, suas posições são gravadas como mensagem, atualizadas e enviadas ao ROS em uma frequência de 10 Hz, determinada no programa.

Na Figura 33, é demonstrado o processo de Visão Robótica, por meio do fluxograma, de que possa apresentar uma perspectiva sequencial das etapas executadas.

Figura 33 - Fluxograma da Programação de Visão Robótica



Fonte: Autoria própria (2019).

4.3.1.1 Executando o OpenCV

Com o programa de detecção já desenvolvido, o próximo passo é a execução do programa. Para isso, foi necessário importar o pacote do driver para interface da câmera USB, possibilitando a captura de imagem. Realizou-se aquisição do driver através do comando:

$ git clone https://github.com/ros-drivers/usb_cam.git

Sendo este pacote descarregado na pasta src da área de trabalho do ROS. Agora utilizando o comando abaixo, executamos a inicialização do pacote, carregando a imagem.

$ roslaunch usb_cam usb_cam-test.launch

Para a correta execução do nó e inicialização da publicação das imagens, outros parâmetros essenciais foram necessários, como: *video_device, image_width, image_height, pixel_format e camera_frame_id*. Sendo respectivamente, as portas de conexão e o formato do vídeo. Todos esses parâmetros e outros são passados via arquivo *usb_cam-test.launch* do tipo XML, o qual está parcialmente apresentado a seguir:

```
<launch>
        < node name="usb_cam"  pkg="usb_cam" type="usb_cam_node" output="screen" >
                <param name="vídeo_device" value="/dev/vídeo" />
                <param name="image_width" value="640" />
                <param name="image_height" value="480" />
                <param name="pixel_format" value="yuyv" />
                <param name="camera_frame_id" value="usb_cam" />
        </node>
</launch>
```

A publicação e/ou assinatura de qualquer nó em um tópico determinado foi vinculada a execução de uma estrutura básica do ROS. Logo o ROS mestre que é o servidor de parâmetro e um nó de *logging rosout*, é a estrutura básica.

Para a execução desta estrutura, normalmente carrega-se com o comando *roscore*. No entanto para inicializar o arquivo *usb_cam-test.launch*, todos os componentes foram iniciados automaticamente, sem que o uso do comando roscore anteriormente mencionado, fosse necessário.

Para confirmar a real publicação da imagem da câmera, executou-se o comando *rostopic list*, para retornar todos os tópicos ativos. Com isso, o *tópico /usb_cam/image_raw* está pronto e aberto na janela do monitor para visualização dos eventos no sistema de visão.

Na Figura 34 é mostrado o Processamento Grafo do ROS que faz parte deste trabalho, representando aproximadamente 2/8 de todo o Grafo.

Figura 34 - Representação do Processamento Grafo do ROS para a Visão Computacional

Fonte: Autoria própria (2019).

Após a inicialização do nó da visão computacional, iniciamos o nó de detecção de obstáculo, mediante o nó Follower_p. Com isso este nó assina o anterior para receber as imagens e gera o tópico /detected com as coordenadas de saída do obstáculo. O comando abaixo, carrega o nó Follower_p.

$ rosrun usb_cam Follower_p.py

Na Figura 35 é mostrado o Processamento Grafo do ROS que faz parte deste trabalho, representando aproximadamente 2/8 de todo o Grafo.

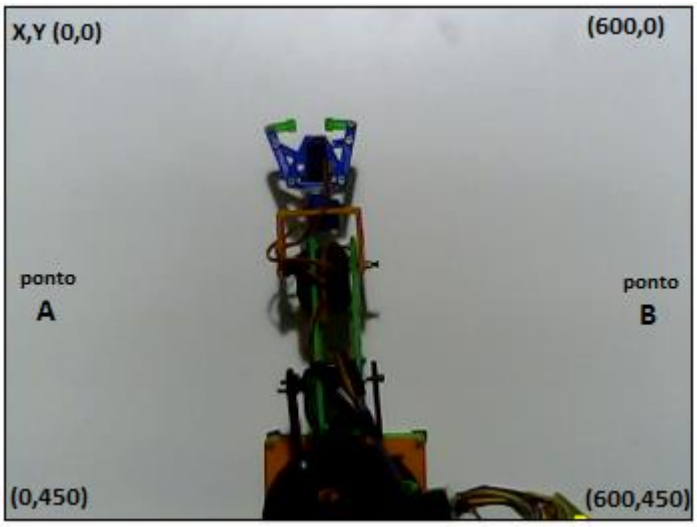
Figura 35 - Representação do Processamento Grafo do ROS para o Reconhecimento de Obstáculo.

Fonte: Autoria própria (2019).

4.3.2 Programa de Manipulação e Desvio.

Para a programação das trajetórias do manipulador, fez-se necessário a avaliação de espaços e possibilidades em que o objeto pode adentrar ao seu ambiente de atuação. Pois, sabendo que sua trajetória fixa, é simulado a manipulação de peças do ponto A ao ponto B (Figura 36), com a missão de carregar ou descarregar algum pallet.

Figura 36 - Ambiente de trabalho do Manipulador.



Fonte: Autoria própria (2019).

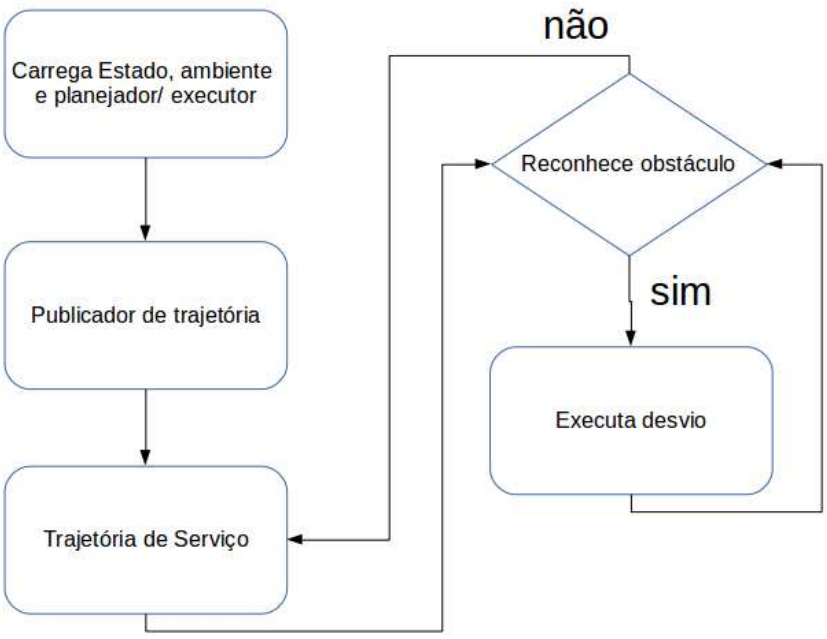
Caso seja detectado algum obstáculo na área de trabalho do manipulador, via sistema de visão computacional. O manipulador deverá executar uma trajetória desviando deste obstáculo, evitando uma consequente colisão.

Para isso, foi criado a virtualização do braço robótico com o uso do arquivo (arm.urdf) que segue no apêndice c, logo em seguida foi gerado os arquivos correspondentes aos cálculos cinemáticos, seguindo o tutorial anteriormente descrito. Para finalizar, utilizou-se o assistente de configuração do moveit usando o código abaixo:

roslaunch moveit_setup_assistant setup_assistant.launch

Dando continuidade no desenvolvimento, na Figura 37 é apresentado o fluxograma sequencial dos eventos de funcionamento do programa de desvio.

Figura 37 - Fluxograma do Manipulador.



Fonte: Autoria própria (2019).

Para ocorrência destes eventos, fez-se necessário um algoritmo estruturado com as seguintes condicionalidades:

Iniciamos com a importação de bibliotecas para o uso da interface Python Moveit, necessita-se do *moveit_commander,* o qual nos fornece as classes *MoveGroupCommander, PlanningSceneInterface* e *RobotCommander*. Também a *geometry_msg*, que fazemos uso para recepção de mensagem da detecção do objeto pelo Opencv pela classe Point e para inclusão do objeto 3D na virtualização do Rviz com o *PoseStamped.*

```
import moveit_commander
import moveit_msgs.msg
from geometry_msgs.msg import Point
from geometry_msgs.msg import PoseStamped
```

Primeiramente inicializa o *moveit_commander* e o nó rospy:

moveit_commander.roscpp_initialize(sys.argv)

rospy.init_node('move_group_python_interface_tutorial', anonymous=True)

Na sequência é instanciado o objeto *RobotCommander*, cuja função é fornecer informações cinemática do robô e estado das juntas atuais do manipulador.

robot = moveit_commander.RobotCommander()

O *PlanningSceneInterface*, instancia um objeto. Fornecendo uma interface remota para configurar, obter e atualizar o entendimento do robô sobre o ambiente ao seu redor:

scene = moveit_commander.PlanningSceneInterface()

Instanciando o objeto *MoveGroupCommander*, que é uma interface para um grupo de planejamento (grupo de juntas). Pode ser usada essa interface para planejar e executar movimentos.

move_group = moveit_commander.MoveGroupCommander(“arm”)

Ao criar um editor com *DisplayTrajectory* publicador do ROS, pode ser usado para exibir trajetórias no Rviz:

display_trajectory_publisher =
rospy.Publisher('/move_group/display_planned_path',
moveit_msgs.msg.DisplayTrajectory, queue_size=20)

Pode-se obter os valores do grupo de juntas, que puderam ser ajustados. No código foi criado funções para determinar estas coordenadas alvo do manipulador, como segue nas linhas de códigos abaixo:

```
group_variable_values = group.get_current_joint_values()
def direito():
    group_variable_values[0] = -1.6
    group_variable_values[1] = 1.45
    group_variable_values[2] = -0.7
    group_variable_values[3] = -0.55
```

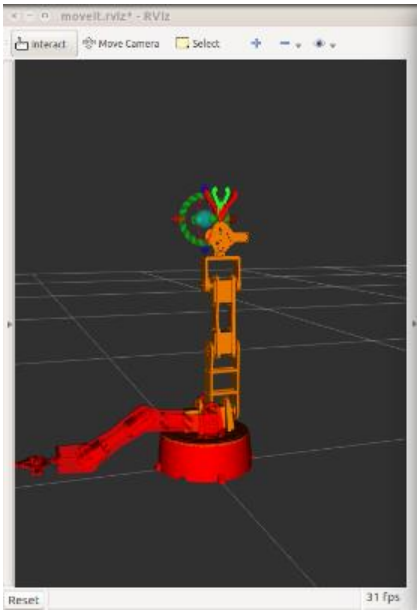
group.set_joint_value_target(group_variable_values)

O comando ***go*** no código a seguir pode ser chamado com valores, poses ou sem nenhum parâmetro, caso já se tenha definido a pose ou o destino conjunto para o grupo:

move_group.go(wait=True)

Na Figura 38, é demonstrado a posição efetuada no Rviz das coordenadas acima mencionadas.

Figura 38 - Alvo "direito" conforme as coordenadas.



Fonte: Autoria própria 2019.

4.3.2.1 Executando o Manipulador

Com o programa de movimentação e desvio já desenvolvido, o próximo passo é a execução do programa. Para isso, foi realizado o download de um pacote de arquivos de exemplo e remodelado para o nosso modelo de Robô, usando o código abaixo:

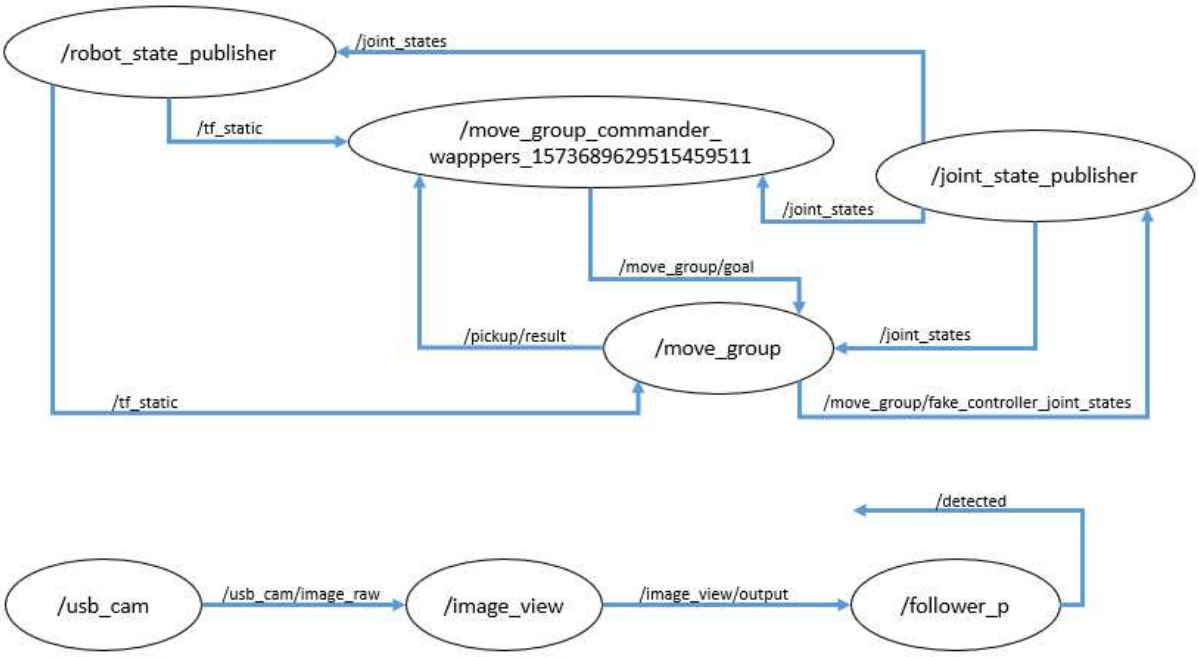
$ git clone https://github.com/AIWintermuteAI/ros-moveit-arm.git

Para a execução do manipulador virtualizado fez uso do comando:

$ roslaunch meu_robo_xacro demo.launch

Com a execução do comando (nó) acima e inicialização da publicação dos valores das juntas e recepção das coordenadas de reconhecimento. Podemos ver o caminho dos dados mostrado na Figura 39 é mostrado o Processamento Grafo do ROS.

Figura 39 - Representação do processamento grafo do ROS para manipulador e visão inicializados.

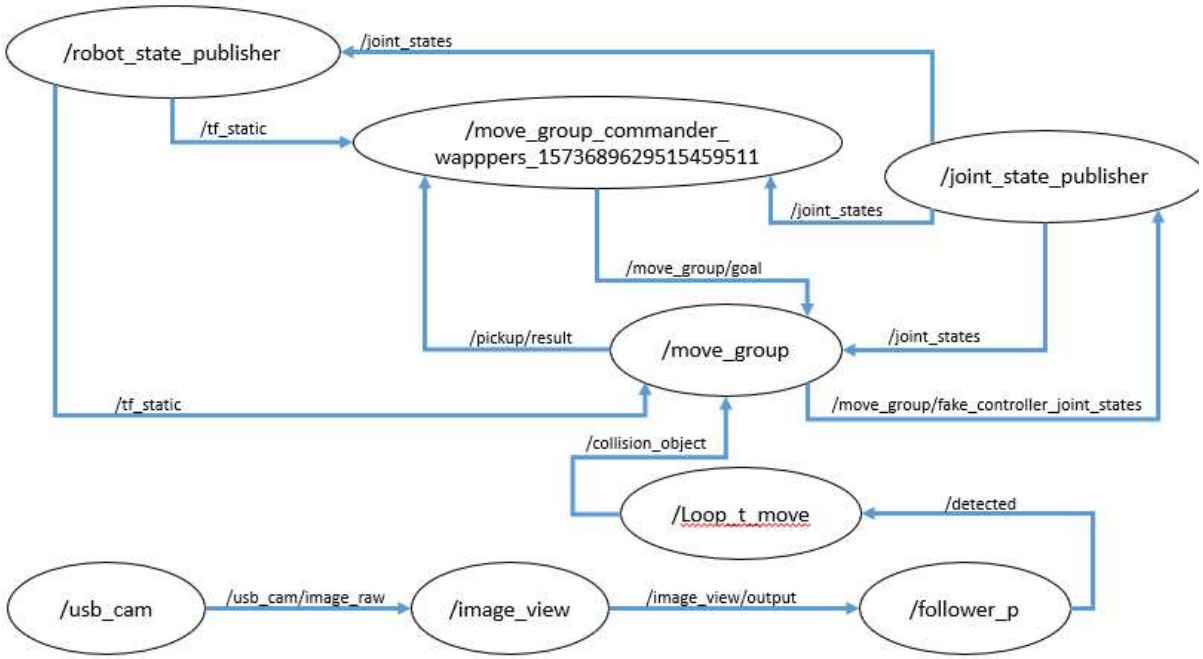


Fonte: Autoria própria (2019).

Após a inicialização do nó do manipulador e da visão computacional, com a detecção de obstáculo, mediante o nó */move_group_commander* e os outros já mencionados anteriormente. Verificamos que este nó assina e pública nos nós de cálculo e estados das juntas. O comando abaixo, carrega o nó */loop_to_move* e na Figura 40 é mostrado o Processamento Grafo do ROS do fluxo de mensagens do processo já descrito anteriormente.

$ rosrun meu_robo_xacro pick/pick_3.py

Figura 40 - Representação do processamento grafo do ROS para o reconhecimento de obstáculo.



Fonte: Autoria própria (2019).

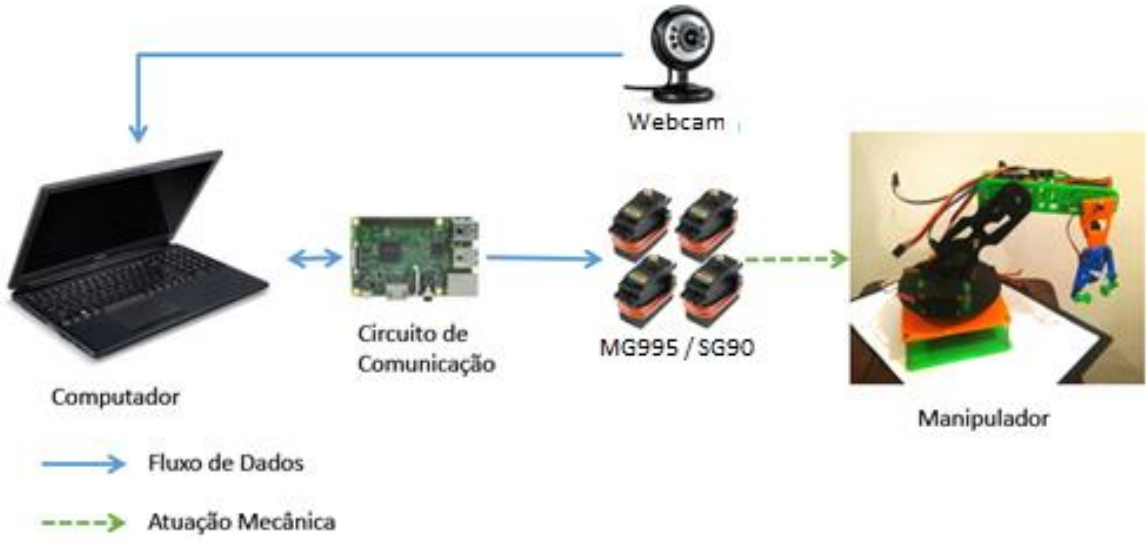
Finalmente, é inicializado a conexão do computador com o controlador por uso de conexão SSH. Mediante o código abaixo:

$ ssh pi@raspberry

4.4 MONTAGEM DO SISTEMA

Para os testes de reconhecimento de objeto e de movimento, montou-se a estrutura, ilustrada na Figura 41, da arquitetura de hardware dos componentes apresentados.

Figura 41 - Arquitetura de hardware



Fonte: Autoria própria 2019.

Para obter a visão de cima da área de atuação do manipulador, é fixado a câmera no centro da área de monitoramento, mostrada na Figura 42, para a identificação das coordenadas de detecção com maior exatidão. Para isso, foi referenciada a câmera à base do manipulador em uma altura de 1,40 metros, suficiente para visualização da área de (0,6 m x 0,45 m) e maior que a altura do espaço de trabalho que pode ser ocupada pelo manipulador, quando em execução.

Figura 42 - Posicionamento da webcam para detecção de obstáculos.

Fonte: Autoria própria (2019).

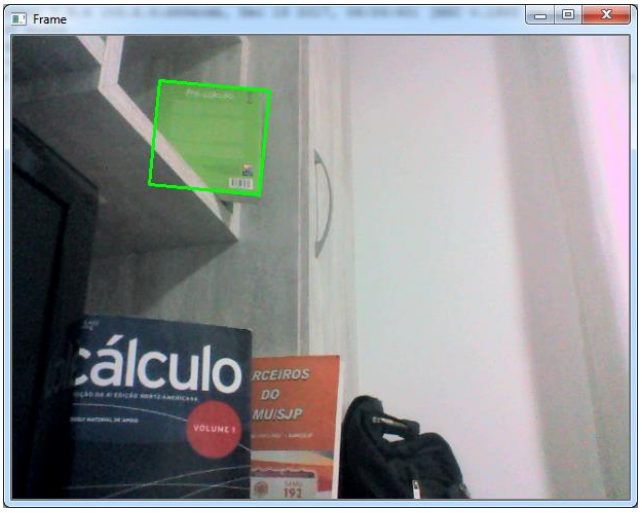
Na Figura 42 é apresentada a instalação do sistema, utilizando um pedestal, com regulagem de posição, com objetivo de atender os parâmetros já estabelecidos ao projeto.

5 APRESENTAÇÃO E ANÁLISE DE RESULTADOS

Nesta etapa do trabalho, são apresentados os resultados obtidos. Primeiramente apresentamos alguns testes realizados com o programa de detecção de obstáculos. Em seguida, as etapas de funcionamento com os módulos integrado ao ROS, junto ao funcionamento do manipulador robótico.

Na Figura 43 é mostrado o contorno, comprovando a detecção do objeto próxima a câmera. Para fins de teste, a cor utilizada foi o verde e a detecção mostrou-se compatível com a cor determinada no programa do sistema computacional.

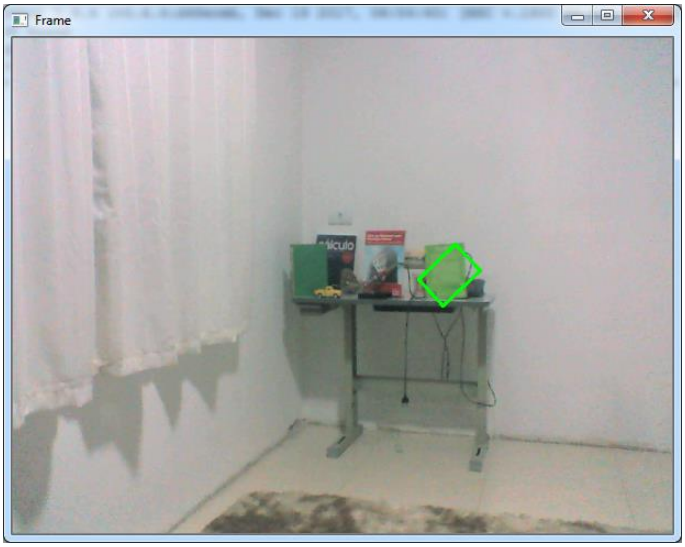
Figura 43 - Detecção próxima de um livro verde.



Fonte: Autoria própria (2019).

O mesmo experimento foi realizado, distanciando o objeto em questão, e inserindo alguns objetos de cores próximas e outras bem distantes. Conforme mostrado na Figura 44, o resultado de detecção foi satisfatório, isto é, nenhuma interferência foi comprovada, com relação à distância e aproximação de tonalidades.

Figura 44 - Detecção distante de um livro verde.

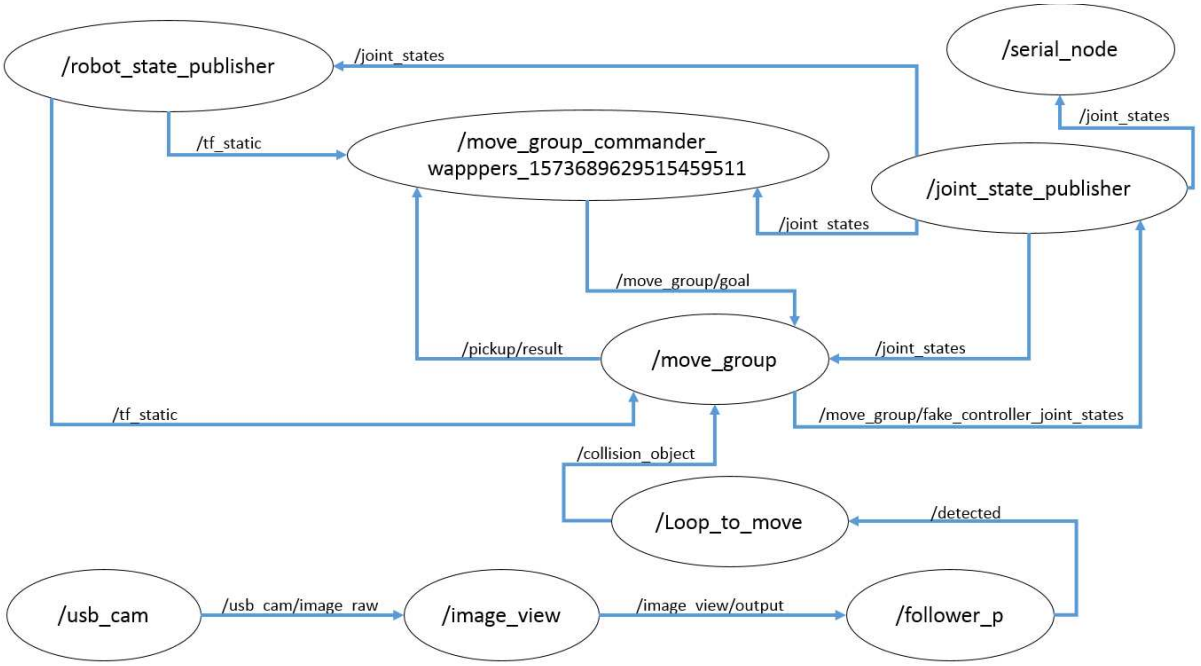


Fonte: Autoria própria (2019).

Esses experimentos realizados, foram necessários para o desenvolvimento e implementações do programa no OpenCV. Sem a necessidade da integração com os softwares de manipulação e simulação.

Após a comprovação do funcionamento do sistema de visão computacional com o OpenCV, integrou-se as outras plataformas, para comprovar o funcionamento do sistema. Para tanto, imprimiu-se na tela a estrutura de processamento Grafo, para fins de conferência da estrutura de comunicação entre os nós, e os tópicos.

Figura 45 - Diagrama de comunicação gerado pelo ROS



Fonte: Autoria própria (2019).

A Figura 45 mostra a interação dos nós, com o fluxo completo entre tópicos e nós, responsáveis pelo funcionamento do sistema. O processamento é iniciado pelo nó */usb_cam* responsável pelo reconhecimento da câmera na porta USB, percorrendo por todas etapas de comunicação necessárias e terminando com o nó */serial_node*, que é a etapa final, onde já são enviados os sinais de comando para os atuadores do manipulador robótico.

Com o funcionamento do sistema, pode-se assistir os eventos realizados pelo manipulador robótico, observando as respostas e tomada de decisões, determinadas por meio dos programas de detecção e desvio de trajetória desenvolvidos.

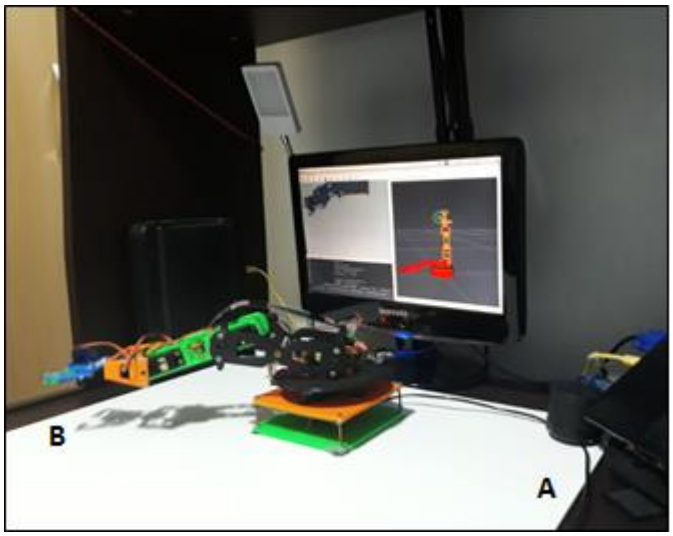
Na Figura 46 é mostrado os equipamentos ligados e prontos para serem inicializados. Na tela do notebook pode ser observado o sincronismo nos movimentos entre os manipuladores físico e o simulado no RVIZ.

Figura 46 - Simulação do sistema ROS integrado.

Fonte: Autoria própria (2019).

Após iniciar o robô para execução da trajetória programada para manipulação. Inicialmente o manipulador da posição de repouso e inicia as tarefas de um ponto a outro. Com uma posição de trabalho estabelecido. O trajeto para a execução da tarefa do manipulador foi convencionado do ponto A ao ponto B ou vice-versa. A Figura 47 mostra, o manipulador no ponto B, preparando-se para o deslocamento ao ponto A.

Figura 47 - Análise do percurso 1.

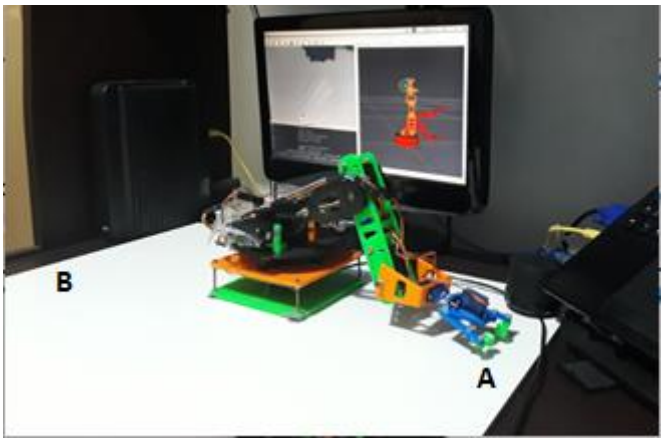


Fonte: Autoria própria (2019).

É visto que existe três opções para o monitoramento da área de trabalho do manipulador, que é na tela de imagem capturada pela câmera, na tela do visualizador RVIZ e visualizando o espaço físico (olho humano).

Observa-se na Figura 48, que não há obstáculo na área de trabalho do manipulador. Portando não há sinalização para que o desvio seja executado. Então o manipulador realizou sua trajetória normal até chegar ao ponto A.

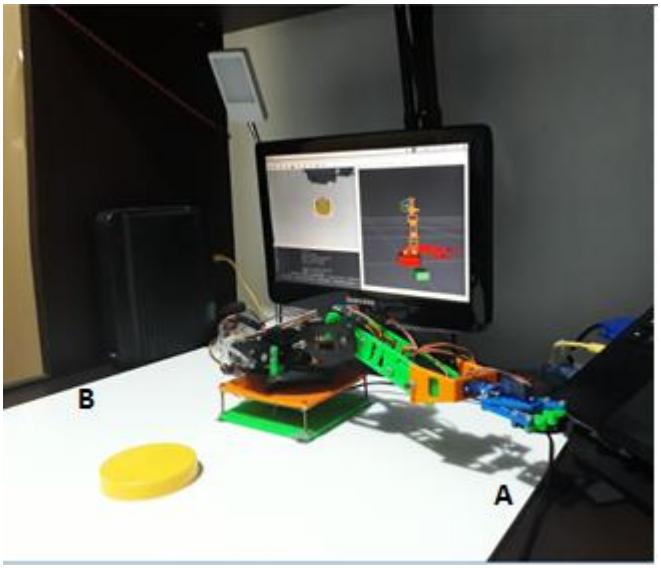
Figura 48 - Análise do percurso 2.



Fonte: Autoria própria (2019).

Na Figura 49 o manipulador já realizou o percurso com a posição normal, chegando ao ponto A. O próximo evento será o retorno ao ponto B.

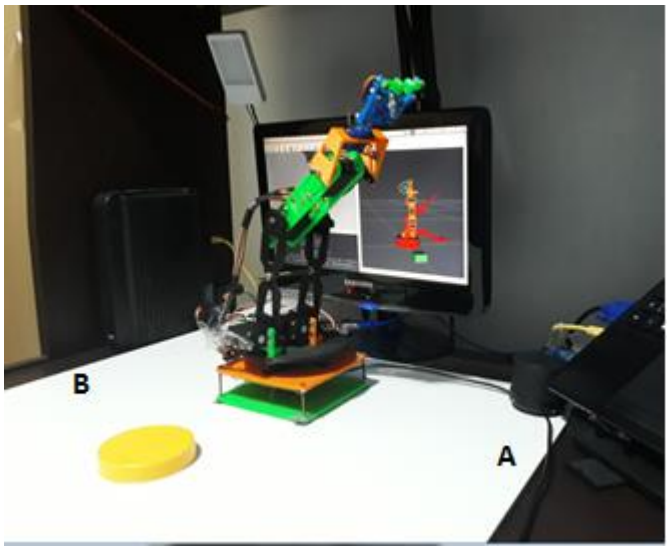
Figura 49 - Análise do percurso 3.



Fonte: Autoria própria (2019).

É observado na Figura 50 o surgimento de um obstáculo na área de trabalho, que pode ocasionar uma colisão. Para isso será necessário que o manipulador tenha uma trajetória alternativa, para que a colisão seja evitada.

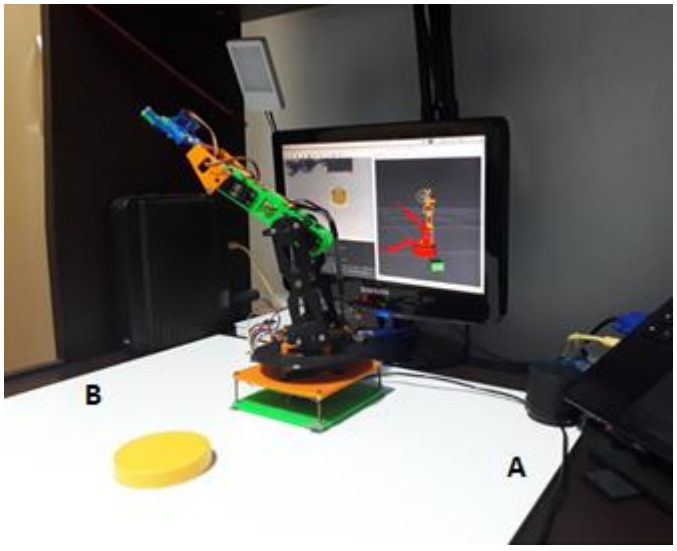
Figura 50 - Análise do percurso 4.



Fonte: Autoria própria (2019).

Obedecendo as condições determinadas no algoritmo, o manipulador foi informado sobre o obstáculo, e trafegará a uma altura maior que a normal, evitando a colisão. A Figura 51, mostra o manipulador com o posicionamento alternativo.

Figura 51 - Análise do percurso 5.



Fonte: Autoria própria (2019).

Na Figura 51, é observado que o obstáculo ainda permanece na zona de risco. Logo o manipulador manterá seu movimento alternativo. Até que o percurso normal seja liberado. Com isso, é comprovado, o funcionamento do sistema e os resultados esperados, na interação do robô com os obstáculos.

6 CONCLUSÃO

As ferramentas OpenSource tem propiciado uma interação considerável entre estudiosos de diferentes áreas tecnológica. Pois, além das disponibilidades de recursos técnicos-didáticos, há uma reciprocidade nas informações e idéias, resultando em um avanço de conhecimento nas diversas áreas. Principalmente na área robótica, que necessita de programas e softwares avançados e inteligentes, requerendo sempre a criatividade e raciocínios lógicos.

Conhecendo as aplicações e ferramentas disponibilizadas na biblioteca OpenCV, é percebido que houve um grande avanço no sistema de visão computacional, e o fato de possuir vários pacotes prontos, além de facilitar o processo de programação, dá-nos a oportunidade de tomar base das ferramentas já existentes e explorar tais recursos com maior criatividade.

Com a utilização do ROS, as plataformas para o desenvolvimento de sistemas inteligentes são viabilizadas, pois há uma gama de frameworks que dá suporte aos sistemas mais avançados da robótica, como neste trabalho, todos os recursos para o bom funcionamento do manipulador robótico, foi desenvolvido através de plataformas sofisticadas, que interagem facilmente com o Sistema Operacional Robótico.

Embora programas com funções de detecções mais complexos não foram aplicados neste trabalho, ao consultar artigos, monografias, teses, publicações, e tutoriais, foi visto que há possibilidade de projetar um sistema mais preciso e inteligente, capaz de tomada de decisões, selecionando ou reconhecimento objeto pelo formato, cores, etc. Além do reconhecimento de pessoas, por detecção de face, a detecção de movimentos a partir de algumas características.

Essa mesma tecnologia aplicada para o manipulador robótico, pode ser utilizada para outros sistemas de segurança que requer precisão na captação de imagem e informações mais apuradas sobre características ou posicionamento de objetos, equipamentos, pessoas ou animais.

Conhecendo os diversos recursos disponíveis no campo de visão computacional, é comprovado o quanto ainda se pode avançar em pesquisas e aplicações. E quanto essa busca por melhorias pode atribuir para o melhor desempenho na automatização dos sistemas. Esses recursos são de extrema importância para alavancar cada vez mais o sistema de indústria 4.0.

Os objetivos específicos propostos foram alcançados no desenvolvimento do trabalho, ainda que o sistema funcione de forma satisfatória, o *hardware* e *software* apresentam um *delay*, tanto de aquisição de imagem quanto na resposta após a retirada do objeto. Esse problema de hardware é observado pela utilização de equipamentos de baixa resolução de envio de dados.

Uma outra dificuldade encontrada na execução e estudo do projeto, considerado a de maior relevância, foi o tempo necessário para o aprendizado das ferramentas de programação, que consequentemente interferiu no desenvolvimento mais aprimorado do código de desvio.

TRABALHOS FUTUROS

Este trabalho pode ter continuidade com implementações e melhorias nos seguintes pontos:

- Sincronismo da trajetória com obstáculo;
- Velocidade e aceleração dos movimentos do Braço Robótico;
- Cadastro de obstáculos e reconhecimento dos mesmos.

REFERÊNCIAS

ABB Robotics. **Fabricante de Robôs Industriais.** Disponível em: <https://new.abb.com/products/robotics/pt>. Acesso em: 5 nov. 2019.

ACHMAD, M. H. et al. **Tele-Operated Mobile Robot for 3D Visual Inspection Utilizing Distributed Operating System Platform.** International Journal of Vehicle Structures & Systems, MechAero Foundation for Technical Research & Education Excellence, v. 9, n. 3, p. 190–194, 2017. Citado na página 44.

ALLDATASHEET. **All datasheet catalog Search Engine**. Disponível em: <https://www.alldatasheet.com/datasheetpdf/pdf/1132435/ETC2/MG995.html>. Acesso em: 13 out. 2019.

AMARAL, Silas. Apostila de Fundamentos de Robótica Industrial. UDESC. <http://www.joinville.udesc.br/portal/professores/silas/materiais/Apostila_de_Robotica.pdf>. Disponível em: 13 agosto de 2018.

ANTONELLO, Ricardo. **Introdução a Visão Computacional com Python e OpenCV.** Mestre em Ciência da Computação pela UFSC. Publicado em 2017. Disponível em: <http://professor.luzerna.ifc.edu.br/ricardo-antonello/wp-content/uploads/sites/8/2017/02/Livro-Introdu%C3%A7%C3%A3o-a-Vis%C3%A3o-Computacional-com-Python-e-OpenCV-3.pdf> Acesso: nov. 2019.

ANTONIO, Thiago Braga de Almeida. **Método de Controle e Detecção de Obstáculos para Robôs Manipuladores Aplicado a Interação ao Humano-Robô.** Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia Elétrica, COPPE, da Universidade Federal do Rio de Janeiro, Rio de Janeiro, 2014.

AREND, Renan F. **Detecção e Histórico de Nevos em Imagens da Pele Humana.** 2015. 77 f. Monografia (Trabalho de Conclusão de Curso) - Curso de Engenharia de Computação, Universidade Tecnológica Federal do Paraná, Campus Pato Branco. Pato Branco, 2015.

AUGUSTO, Gonçalo Nuno dos Santos; **RobotTeamSim – 3D Visualization of Cooperative Mobile Robots Missions in Gazebo Virtual Environment.** Mestrado em Ciências Elétricas e Engenharia da Computação; Universidade de Coimbra, PORTUGAL, 2013.

BALL, Stuart. – “**Embedded Microprocessor Systems: Real World Design**”, 3rd edition, Editora: MCPros, EUA, 2005.

BRADISK, G.; KAEHLER, A. **Learning OpenCV: Computer Vision with the OpenCV Library**. O'Reilly, 2008.

CARVALHO, Fabio. **Astrofotografia com Webcams.** Escrito em 2011. Disponível em: <http://www.cyberplocos.com.br/artigos/astrofotografia-com-webcams.html>. Acesso em: Novembro de 2018.

CRAIG. John J., **Robótica**; tradução Heloísa Coimbra de Souza; revisão técnica Reinaldo Augusto de Costa Bianchi, 3ª edição, São Paulo, Pearson Education do Brasil, 2012.

DELAI, Riccardo Luigi (1); COELHO, Alessandra Dutra (2). **Visão Computacional com a OpenCV – Material Apostilado e Veículo Autônomo.** Edição: 2014. (1) Aluno de Iniciação Científica da Escola de Engenharia Mauá (EEM/CEUN-IMT); (2) Professora da Escola de Engenharia Mauá (EEM/CEUN-IMT). Disponível em: < https://maua.br/files/082014/visao-computacional-opencv-material-apostilado-veiculo-seguidor-autonomo.pdf > Acesso em Julho de 2018.

EBERMAM, Elivelto; PESENTE, Guilherme M.; RIOS, Renan O.; PULINI, Igor C. **Programação para Leigos com Raspberry Pi.** 1ª edição, Espírito Santo, Editora IFPB, 2017.

ECOS, Engenharia e Automação. **Sistema de Visão Robótica.** Disponível em: <https://www.ecos.eng.br/sistema-visao-robotica> Acesso em: 5 maio de 2018.

GAZEBOSIM. Disponível em: < http://gazebosim.org>. Acesso em 11 nov. 2018.

GRASSI, Maurício Velloso (2005). **Desenvolvimento e aplicação de um sistema de visão para robô industrial de manipulação.** Trabalho de Conclusão de Curso em pós graduação em Engenharia Mecânica. UFRS, 2005. Disponível em: <https://lume.ufrgs.br/bitstream/handle/10183/6202/000482391.pdf?sequence=1 >. Acesso em: 11 set. 2018.

HAVERBEKE, Marijn. **Eloquent JavaScript,** 2ª ed. No Starch Press, 2014. Disponível em: <http://eloquentjavascript.net/>. Acesso em: novembro. 2018.

JUCA, Sandro. **Aplicações Práticas de Sistemas Embarcados Linux utilizando Raspberry Pi** [recurso eletrônico] / Sandro Jucá e Renata Pereira. 1ª ed. Rio de Janeiro; PoD, 2018. Recurso digital; 21MB.

MATOS, N. M. R. de. **Análise do Funcionamento de um Servomotor de Corrente Alternada com Ímãs Permanentes.** Trabalho de Conclusão de Curso — Universidade Regional de Blumenau, Blumenau, SC, Brasil, 2012.

MATOS, Wesley C. P. de; KLEM, Christofer dos Reis. **Braço Robótico com Visão Computacional.** Trabalho de Conclusão de Curso — Faculdade Estácio de Curitiba, Curitiba, PR, Brasil, 2018.

MATOTTA, Giuliano Sant'Anna. **Processamento Digital de Imagens**, UNIPAC UBÁ, 2007 Disponível em:<https://slideplayer.com.br/slide/3360786> Acesso em Dezembro de 2018.

NEVES, Vinicius. **Visão Computacional**,2009. Disponível em: <http://sdbmcc.blogspot.com/2009/06/imagem-digital-teoria.html>. Acesso em Dezembro de 2018.

OLIVEIRA, Leandro Luiz Rezende. **Controle de Trajetória Baseado em Visão Computacional Utilizando o Framework ROS.** Dissertação de Mestrado em Engenharia Elétrica na Universidade Federal de Juiz de Fora, Minas Gerais, 2013.

OPENCV, About. Disponível em: <http://opencv.org/about.html>. Acesso em: 1 set. 2015.

ORLANDINI, Guilherme. **Desenvolvimento de Aplicativos Baseados em Técnicas de Visão Computacional para Robô Móvel Autônomo**, Mestrado em Ciência da Computação; UMP; Piracicaba, 2012.

PAZOS, Fernando, **Automação de Sistemas & Robótica**, Rio de Janeiro, Axcel Books, 2002.

PYTHON. **Python Brasil.** Disponível em: < https://python.org.br/instalacao-linux > Acesso em: Março 2019.

ROCHA, Walber Conceição de Jesus, **Desenvolvimento de um Sistema de Medição de Distância Baseado em Visão Computacional Utilizando Laser de Linha**, Trabalho de conclusão de curso - Centro de Ciências Exatas e Tecnológicas da Universidade Federal do Recôncavo da Bahia. Bahia, 2019.

ROMANO, Vitor Ferreira; DUTRA, Max Suell**. Introdução à robótica industrial.** In: ROMANO, Vitor Ferreira; DUTRA, Max Suell. **Robótica.** [S.l.: s.n.], [2002]. cap. 1, p. 1-21.Disponível em:<http://www.fem.unicamp.br/~hermini/Robotica/livro/cap.1.pdf>. Acesso em outubro. 2018.

ROS.org. **Tutoriais sobre o ROS.** Artigo escrito em 2014 – Disponível em: <http://wiki.ros.org/pt_BR/ROS/Tutorials>. Acesso em: 10 out. 2018.

ROSÁRIO, João Maurício, **Princípios de Mecatrônica**, São Paulo, Prentice Hall, 2005.

SANTOS, Eduardo dos. **O uso da visão computacional para o controle de um manipulador robótico.** 2014. 60 f. Monografia (Trabalho de Conclusão de Curso) - Curso de Engenharia de Computação, Universidade Tecnológica Federal do Paraná, Câmpus Pato Branco. Pato Branco, 2014.

SANTOS, Vítor. M. Ferreira. **Robótica industrial**. 2004. Disponível em: <http://www.ece.ufrgs.br/~rventura/RoboticaIndustrial.pdf> Acesso: set. 2017.

SANTOS, Winderson Eugênio dos. **Robótica Industrial: fundamentos, tecnologias, programação e simulação.** 1ª edição, São Paulo, Érica, 2015.
SEBESTA, Robert W. **Concepts of Programming Languages**, 11ª ed. Pearson, 2015.

SHAPIRO, Linda G.; STOCKMAN, George C.; **Computer Vision.** University of Washington, Editora Pearson; EUA, 2001.

SOUZA, Marcelo A. R. C.C., **Desenvolvimento de um Simulador de Robô Móvel.** Mestrado em Ciências Eletrotécnica e Engenharia da Computação; Universidade de Coimbra, PORTUGAL, 2015.

TRONCO, Mário Luiz. **Modelagem Cinemática de Robôs Industriais**, Engenharia Mecânica da Universidade de São Paulo – São Carlos, 2017. Disponível em: <http://www.simulacao.eesc.usp.br/sem406/material/Modelagem_Cinematica.pdf> Acesso em Setembro, 2018.

UNIVERSAL, Robots. Robótica de Linha de Montagem com um Braço Robótico Colaborativo. Disponível em: <https://www.universal-robots.com/br/aplicac%C3%B5es/montagem/> Acesso: Jul. 2018.

VAGALLIS, Nikos. **Gazebo Robotics Simulator.** Artigo escrito em 2016 - Disponível em: <https://www.i-programmer.info/news/169-robotics/9439-gazebo-robot-simulator.html>. Acesso em: 12 nov. 2018.

VANÇAN, Nícolas B. Montanha. **PROTOBOT - Protótipo de Braço Robótico comandado por comunicação sem fio**. 124 p. Trabalho de Conclusão de Curso em Engenharia Elétrica - Universidade Estadual de Londrina, Londrina, 2017.

WILLRICH, Roberto, **Introdução à Informática**; Apostila de Graduação em Informática e Estatística. Santa Catarina, 2000.

APÊNDICE A

Follower_p.py

```
#! /usr/bin/python

# rospy for the subscriber
import rospy
# ROS Image message
from sensor_msgs.msg import Image
# ROS Image message -> OpenCV2 image converter
from cv_bridge import CvBridge, CvBridgeError
# OpenCV2 for saving an image
import cv2

import numpy
import imutils

# Envio de mensagem de reconhecimento
#from std_msgs.msg import String
from geometry_msgs.msg import Point
# Instantiate CvBridge
bridge = CvBridge()

# Definir os limites inferiores e superiores da cor
#amarelo
#yellowLower = (25, 50, 50)
#yellowUpper = (32, 255, 255)

yellowLower = numpy.array([25, 50, 50], numpy.uint8)
yellowUpper = numpy.array([32, 255, 255], numpy.uint8)

#yellowLower = numpy.array([20,255, 255], numpy.uint8)
#yellowUpper = numpy.array([0,0, 255], numpy.uint8)

def image_callback(Image):

    try:
        cv2_img = bridge.imgmsg_to_cv2(Image, desired_encoding='bgr8')
        cv2_img = imutils.resize(cv2_img, width=600)
        hsv = cv2.cvtColor(cv2_img, cv2.COLOR_BGR2HSV)
        maskYellow = cv2.inRange(hsv, yellowLower, yellowUpper)
        maskYellow = cv2.erode(maskYellow, None, iterations=2)
        maskYellow = cv2.dilate(maskYellow, None, iterations=2)
        cntYellow = cv2.findContours(maskYellow.copy(), cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)[-2]
        centerYellow = 0
        if len(cntYellow) > 0:
            print len(cntYellow)
            cYellow = max(cntYellow, key=cv2.contourArea)
            rectYellow = cv2.minAreaRect(cYellow)
            boxYellow = cv2.boxPoints(rectYellow)
            boxYellow = numpy.int0(boxYellow)
            MYellow = cv2.moments(cYellow)
            if MYellow['m00'] > 0:
                M = MYellow
```

```
                cx = int(M['m10'] / M['m00'])
                cy = int(M['m01'] / M['m00'])
                centerYellow = (cx, cy)
            cv2.drawContours(cv2_img, [boxYellow], 0, (0, 255, 255), 2)
            pub = rospy.Publisher('detected', Point, queue_size=1)
            rate = rospy.Rate(10)  # 10hz
            msg = Point()
            msg.x = cx * 0.11666
            msg.y = cy * 0.09777
            print (centerYellow)
            rospy.loginfo(msg)
            pub.publish(msg)
            rate.sleep()
        else:
            pub2 = rospy.Publisher('detected', Point, queue_size=1)
            msg = Point()
            msg.x = 0
            msg.y = 0
            rospy.loginfo(msg)
            pub2.publish(msg)

        cv2.imshow("window", cv2_img)
        cv2.waitKey(1) & 0xFF
    except CvBridgeError, e:
        print(e)

def main():
    rospy.init_node('follower_p')
    # Define your image topic
    image_topic = "/image_view/output"
    # Set up your subscriber and define its callback
    rospy.Subscriber(image_topic, Image, image_callback)
    # Spin until ctrl + c
    rospy.spin()

if __name__ == '__main__':
    main()
```

APÊNDICE B

Pick_3.py

```
#! /usr/bin/env python

import sys
import copy
import rospy
import moveit_commander
import moveit_msgs.msg
from geometry_msgs.msg import Point
from geometry_msgs.msg import PoseStamped

resultx = Point()
resulty = Point()

var_x = 0

"""Primeiro inicialize moveit_commander e rospy."""
moveit_commander.roscpp_initialize(sys.argv)
rospy.init_node('Loop_to_move')
robot = moveit_commander.RobotCommander()

scene = moveit_commander.PlanningSceneInterface()

group = moveit_commander.MoveGroupCommander("arm")
garra = moveit_commander.MoveGroupCommander("gripper")

display_trajectory_publisher =
rospy.Publisher('/move_group/display_planned_path',
moveit_msgs.msg.DisplayTrajectory)

group.clear_pose_targets()

""" obteremos o conjunto atual de valores conjuntos para o grupo"""
group_variable_values = group.get_current_joint_values()

p = PoseStamped()
p.header.frame_id = robot.get_planning_frame()

def objeto():
    p.pose.position.x = var_x/100
    p.pose.position.y = 0.2
    print p
    p.pose.position.z = 0.0
    scene.add_box("part", p, (0.07, 0.07, 0.05))

garra.set_named_target("open")
garra.go(wait=True)
rospy.sleep(1)

garra.set_named_target("close")
garra.go(wait=True)
rospy.sleep(1)

garra.set_named_target("open")
```

```
garra.go(wait=True)
rospy.sleep(1)
def aproximacao_direita():

    group_variable_values[0] = -1.6
    group_variable_values[1] = 1.15
    group_variable_values[2] = -1.5
    group_variable_values[3] = -1.15
    group.set_joint_value_target(group_variable_values)

def aproximacao_esquerda():

    group_variable_values[0] = 1.5
    group_variable_values[1] = 1.15
    group_variable_values[2] = -1.5
    group_variable_values[3] = -1.0
    group.set_joint_value_target(group_variable_values)

def elevacao_direita_desvio():

    group_variable_values[0] = -1.3
    group_variable_values[1] = 0.1
    group_variable_values[2] = -0.8
    group_variable_values[3] = -0.2
    group.set_joint_value_target(group_variable_values)

def elevacao_esquerda_desvio():

    group_variable_values[0] = 1.3
    group_variable_values[1] = 0.1
    group_variable_values[2] = -0.8
    group_variable_values[3] = -0.2
    group.set_joint_value_target(group_variable_values)

def elevacao_direita():

    group_variable_values[0] = -1.3
    group_variable_values[1] = 1.3
    group_variable_values[2] = -0.5
    group_variable_values[3] = -0.3
    group.set_joint_value_target(group_variable_values)

def elevacao_esquerda():

    group_variable_values[0] = 1.3
    group_variable_values[1] = 1.3
    group_variable_values[2] = -0.5
    group_variable_values[3] = -0.3
    group.set_joint_value_target(group_variable_values)

def direito():

    group_variable_values[0] = -1.6
    group_variable_values[1] = 1.45
    group_variable_values[2] = -0.7
    group_variable_values[3] = -0.6
    group.set_joint_value_target(group_variable_values)

def esquerdo():

    group_variable_values[0] = 1.5
```

```
    group_variable_values[1] = 1.5
    group_variable_values[2] = -0.7
    group_variable_values[3] = -0.65
    group.set_joint_value_target(group_variable_values)

def callback(msg):

    global resultx
    global resulty
    resultx = int(msg.x)
    resulty = int(msg.y)

while not rospy.is_shutdown():

    rospy.Subscriber("detected", Point, callback)
    rate = rospy.Rate(1)
    global var_x
    var_x = resultx
    var_y = resulty
    print ("verif_x =", var_x)

    aproximacao_esquerda()
    plan2 = group.plan()
    group.go(wait=True)

    esquerdo()
    plan2 = group.plan()
    group.go(wait=True)
    rospy.sleep(1)

    garra.set_named_target("close")
    garra.go(wait=True)
    rospy.sleep(2)

    if var_x == 0:
        scene.remove_world_object("part")
        elevacao_esquerda()
        plan2 = group.plan()
        group.go(wait=True)

        rospy.Subscriber("detected", Point, callback)
        rate = rospy.Rate(1)
        global var_x
        var_x = resultx
        var_y = resulty
        print ("verif_x =", var_x)

        if var_x == 0:
            scene.remove_world_object("part")
            elevacao_direita()
            plan2 = group.plan()
            group.go(wait=True)
        #else:
    if var_x >= 30:
        if var_x <= 38:
            objeto()
            rospy.sleep(2)
            elevacao_esquerda_desvio()
            plan2 = group.plan()
            group.go(wait=True)
```

```
            elevacao_direita_desvio()
            plan2 = group.plan()
            group.go(wait=True)

    rospy.Subscriber("detected", Point, callback)
    rate = rospy.Rate(1)
    global var_x
    var_x = resultx
    var_y = resulty

    aproximacao_direita()
    plan2 = group.plan()
    group.go(wait=True)

    direito()
    plan2 = group.plan()
    group.go(wait=True)
    rospy.sleep(1)

    garra.set_named_target("open")
    garra.go(wait=True)
    rospy.sleep(2)

    aproximacao_direita()
    plan2 = group.plan()
    group.go(wait=True)
    if var_x == 0:
        scene.remove_world_object("part")
        elevacao_direita()
        plan2 = group.plan()
        group.go(wait=True)

        rospy.Subscriber("detected", Point, callback)
        rate = rospy.Rate(1)
        global var_x
        var_x = resultx
        var_y = resulty

        if var_x == 0:
            scene.remove_world_object("part")
            elevacao_esquerda()
            plan2 = group.plan()
            group.go(wait=True)
        #else:
    if var_x >= 30:
        if var_x <= 38:
            objeto()
            rospy.sleep(1)
            elevacao_direita_desvio()
            plan2 = group.plan()
            group.go(wait=True)

            elevacao_esquerda_desvio()
            plan2 = group.plan()
            group.go(wait=True)

    var_x = 0
    var_y = 0
moveit_commander.roscpp.shutdown()
# Spin until ctrl + c
rospy.spin()
```

APÊNDICE C

arm.urdf

```
      <robot name="robot_arm">

        <link name="base_link">
            <visual>
               <geometry>
                <mesh
filename="package://braco_xacro/meshes/base_cylinder.stl" scale ="0.1 0.1
0.1"/>
               </geometry>
              </visual>
              <collision>
               <geometry>
                <mesh
filename="package://braco_xacro/meshes/base_cylinder.stl" scale ="0.1 0.1
0.1"/>
               </geometry>
              </collision>
        </link>

        <link name="platform">
            <visual>
               <geometry>
                <mesh filename="package://braco_xacro/meshes/top_plate.stl"
scale ="0.1 0.1 0.1"/>
               </geometry>
              </visual>
              <collision>
               <geometry>
                <mesh filename="package://braco_xacro/meshes/top_plate.stl"
scale ="0.1 0.1 0.1"/>
               </geometry>
              </collision>
        </link>

        <link name="lower_arm">
            <visual>
               <geometry>
                <mesh filename="package://braco_xacro/meshes/lower_arm.stl"
scale ="0.1 0.1 0.1"/>
               </geometry>
              </visual>
              <collision>
               <geometry>
                <mesh filename="package://braco_xacro/meshes/lower_arm.stl"
scale ="0.1 0.1 0.1"/>
               </geometry>
              </collision>
        </link>

        <link name="upper_arm">
            <visual>
               <geometry>
```

```
                <mesh filename="package://braco_xacro/meshes/top_arm.stl"
scale ="0.1 0.1 0.1"/>
               </geometry>
              </visual>
              <collision>
               <geometry>
                <mesh filename="package://braco_xacro/meshes/top_arm.stl"
scale ="0.1 0.1 0.1"/>
               </geometry>
              </collision>
        </link>

        <link name="wrist">
            <visual>
               <geometry>
                <mesh filename="package://braco_xacro/meshes/wrist.stl"
scale ="0.1 0.1 0.1"/>
               </geometry>
              </visual>
              <collision>
               <geometry>
                <mesh filename="package://braco_xacro/meshes/wrist.stl"
scale ="0.1 0.1 0.1"/>
               </geometry>
              </collision>
        </link>

        <link name="claw_base">
            <visual>
               <geometry>
                <mesh filename="package://braco_xacro/meshes/clawbase.stl"
scale ="0.1 0.1 0.1"/>
               </geometry>
              </visual>
              <collision>
               <geometry>
                <mesh filename="package://braco_xacro/meshes/clawbase.stl"
scale ="0.1 0.1 0.1"/>
               </geometry>
              </collision>
        </link>

        <link name="claw_l">
            <visual>
               <geometry>
                <mesh filename="package://braco_xacro/meshes/clawl.stl"
scale ="7 7 7"/>
               </geometry>
              </visual>
              <collision>
               <geometry>
                <mesh filename="package://braco_xacro/meshes/clawl.stl"
scale ="7 7 7"/>
               </geometry>
              </collision>
        </link>

        <link name="claw_r">
            <visual>
               <geometry>
```

```
                <mesh filename="package://braco_xacro/meshes/clawr.stl"
scale ="7 7 7"/>
               </geometry>
              </visual>
              <collision>
               <geometry>
                <mesh filename="package://braco_xacro/meshes/clawr.stl"
scale ="7 7 7"/>
               </geometry>
              </collision>
        </link>

        <joint name="joint1" type="revolute">
          <axis xyz="0 0 1" />
          <limit effort="1000.0" lower="-1.708" upper="1.708"
velocity="1.0" />
          <origin xyz="0.0 -0.001 0.507" rpy="0.0 0.0 0.0" />
          <parent link="base_link"/>
          <child link="platform"/>
        </joint>

        <joint name="joint2" type="revolute">
          <axis xyz="0 1 0" />
          <limit effort="1000.0" lower="-1.708" upper="1.708"
velocity="1.0" />
          <origin xyz="0.088 0.121 0.222" rpy="0 0 0" />
          <parent link="platform"/>
          <child link="lower_arm"/>
        </joint>

        <joint name="joint3" type="revolute">
          <axis xyz="0 1 0" />
          <limit effort="1000.0" lower="-1.708" upper="1.708"
velocity="1.0" />
          <origin xyz="0 0 1.235" rpy="0 0 0" />
          <parent link="lower_arm"/>
          <child link="upper_arm"/>
        </joint>

        <joint name="joint4" type="revolute">
          <axis xyz="0 1 0" />
          <limit effort="1000.0" lower="-1.708" upper="1.708"
velocity="1.0" />
          <origin xyz="0 0 0.89" rpy="0 0 0" />
          <parent link="upper_arm"/>
          <child link="wrist"/>
        </joint>

        <joint name="joint5" type="revolute">
          <axis xyz="0 0 -1" />
          <limit effort="1000.0" lower="-1.708" upper="1.708"
velocity="1.0" />
          <origin xyz="0.002 0.054 0.492" rpy="0 0 0" />
          <parent link="wrist"/>
          <child link="claw_base"/>
        </joint>

        <joint name="joint6" type="revolute">
          <axis xyz="0 1 0" />
          <limit effort="1000.0" lower="-0.15" upper="0.30" velocity="1.0"
/>
```

```
    <origin xyz="-0.107 0.1 0.150" rpy="0 -0.3 1.55" />
    <parent link="claw_base"/>
    <child link="claw_l"/>
  </joint>

  <joint name="joint7" type="revolute">
    <axis xyz="0 -1 0" />
    <limit effort="1000.0" lower="-0.15" upper="0.30" velocity="1.0"
/>
    <origin xyz="-0.107 -0.1 0.150" rpy="0 0.3 1.55" />
    <parent link="claw_base"/>
    <child link="claw_r"/>
  </joint>

</robot>
```