TRABALHO DE GRADUAÇÃO

DESENVOLVIMENTO DE API PARA SISTEMA DE LOCALIZAÇÃO E MAPEAMENTO

Natalia Oliveira Borges

Brasília, Maio de 2021

ENGENHARIA
MECATRÔNICA
UNIVERSIDADE DE BRASÍLIA

UNIVERSIDADE DE BRASILIA
Faculdade de Tecnologia
Curso de Graduação em Engenharia de Controle e Automação

TRABALHO DE GRADUAÇÃO

DESENVOLVIMENTO DE API PARA SISTEMA DE LOCALIZAÇÃO E MAPEAMENTO

Natalia Oliveira Borges

Relatório submetido como requisito parcial de obtenção de grau de Engenheiro de Controle e Automação

Banca Examinadora

Profa. Mariana Costa Bernardes Matias, FGA/UnB
Orientadora _____

Prof. Geovany Araújo Borges, ENE/FT/UnB
Orientador _____

Prof. Roberto de Souza Baptista, FGA/UnB
Examinador Interno _____

Prof. Renato Alves Borges, ENE/FT/UnB
Examinador Interno _____

Brasília, Maio de 2021

FICHA CATALOGRÁFICA

NATALIA, OLIVEIRA BORGES
Desenvolvimento de API para Sistema de Localização e Mapeamento
[Distrito Federal] 2021.

x, 101p., 210 x 297 mm (FT/UnB, Engenheiro, Controle e Automação, 2021). Trabalho de Graduação – Universidade de Brasília, Faculdade de Tecnologia.

1. Localização 2.Rastreamento de Agentes
3. Reconhecimento Facial

I. Mecatrônica/FT/UnB II. Título (Série)

REFERÊNCIA BIBLIOGRÁFICA

Borges, Natalia. Oliveira., (2021). Desenvolvimento de API para Sistema de Localização e Mapeamento. Trabalho de Graduação em Engenharia de Controle e Automação, Publicação FT.TG-n°13, Faculdade de Tecnologia, Universidade de Brasília, Brasília, DF, 101p.

CESSÃO DE DIREITOS

AUTOR: Natalia Oliveira Borges

TÍTULO DO TRABALHO DE GRADUAÇÃO: Desenvolvimento de API para Sistema de Localização e Mapeamento

GRAU: Engenheiro ANO: 2021

É concedida à Universidade de Brasília permissão para reproduzir cópias deste Trabalho de Graduação e para emprestar ou vender tais cópias somente para propósitos acadêmicos e científicos. O autor reserva outros direitos de publicação e nenhuma parte desse Trabalho de Graduação pode ser reproduzida sem autorização por escrito do autor.

Natalia Oliveira Borges

Universidade de Brasília (UnB) – Campus Darcy Ribeiro

70919-970 Brasília – DF – Brasil.

Dedicatória

Dedico este trabalho a Deus e aos meus pais, que tanto se dedicam a mim.

Natalia Oliveira Borges

Agradecimentos

Gostaria de agradecer a todos os professores que participaram da minha formação, em especial à minha orientadora, Mariana, que sempre foi atenciosa e gentil, mesmo em períodos difíceis.

Aos meus pais, todo meu amor e carinho, por sempre se esforçarem para me proporcionar a melhor educação e sempre apoiarem minhas decisões. À toda minha família, agradeço pelo exemplo e incentivo.

Aos meus amigos da equipe Unbeatables, agradeço pelos momentos de aprendizado e companheirismo. Em especial à Débora, que tanto me orientou e ajudou neste trabalho, e à Lívia, minha dupla de todos os momentos.

A Marcus e Giselle, obrigada por acompanhar todos os meus momentos de alegrias, tristezas, descobertas e dificuldades.

Por fim, agradeço a Deus, por guiar meus passos e me trazer até aqui.

Natalia Oliveira Borges

RESUMO

A crescente inserção de robôs nos mais diversos ambientes trouxe a necessidade de desenvolvimento de uma estratégia de mapeamento e localização por meio da percepção visual. Este trabalho propõe soluções para a implementação de uma *API* de fácil utilização que ofereça funcionalidades de mapeamento de ambientes por uma rede de câmeras e rastreamento de agentes móveis, tanto robôs quanto humanos. A estratégia de mapeamento de câmeras e agentes robóticos móveis será feita através de marcadores, filtrados usando Filtro de Kalman Estendido. O rastreamento de agentes humanos será feito usando *Skeletons* fornecidos por uma câmera *Kinect* e identificados por meio de reconhecimento facial dos indivíduos, fornecido por modelos pré-treinados de rede neural.

Palavras Chave: Localização, Rastreamento de Marcadores, Filtro de Kalman Estendido, Skeleton, Câmera Kinect, Reconhecimento Facial

ABSTRACT

The rising of robot insertion in the most diverse environments brought the need to develop a mapping and localization strategy through visual perception. This work proposes solutions for the implementation of an easy-to-use *API* that offers features of mapping environments through a network of cameras and tracking of mobile agents, both robots and humans. The strategy of mapping cameras and mobile robotic agents will be done using markers, filtered using an Extended Kalman Filter. The tracking of human agents will be done using Skelentons provided by a *Kinect* camera and identified by facial recognition, provided through pre-trained neural network models.

Keywords: Localization, Marker Traking, Extended Kalman Filter, Skeleton, Kinect Camera, Facial Recognition

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SÍMBOLOS

Símbolos Latinos

p	coordenadas de um ponto	[m]
t	coordenadas de translação de um ponto	[m]
v	vetor de velocidade	[m/s]
q	coordenadas de rotação em quatérnio de um ponto	
x	coordenada x de um ponto na imagem	[pixel]
y	coordenada y de um ponto na imagem	[pixel]
X	coordenada x de um ponto no sistema global	[m]
Y	coordenada y de um ponto no sistema global	[m]
Z	coordenada z de um ponto no sistema global	[m]
f	distância focal da câmera	[m]

Símbolos Gregos

θ	Ângulo de rotação	[rad]

Subscritos

n	referente ao sistema de coordenadas de origem
k	referente ao instante de tempo
x	referente à coordenada x
y	referente à coordenada y
z	referente à coordenada z

Sobrescritos

n	referente ao sistema de coordenadas de destino
i	contador

Siglas

API	Interface de Programação de Aplicação
EKF	Filtro de Kalman Estendido
LARA	Laboratório de Automação e Robótica
GPS	Sistema de Posicionamento Global
ROS	*Robot Operating System*

Capítulo 1

Introdução

O campo da robótica vem crescendo e se tornando cada vez mais uma realidade em vários setores da sociedade. Seja na indústria, com braços robóticos que são capazes de construir produtos desde a fabricação das peças até a montagem final [1]. Seja na medicina, com o desenvolvimento de próteses mecânicas sofisticadas e robôs que auxiliam os médicos em cirurgias [2]. Ou até mesmo desbravando lugares hostis para os seres humanos, como os robôs em missões espaciais [3].

À medida que as tarefas vão se tornando mais complexas, é importante que os robôs consigam trabalhar em diferentes condições e cenários. Os ambientes nem sempre serão fixos e conhecidos, mas sim dinâmicos e desconhecidos. As tarefas não serão mais predefinidas para cada agente, mas sim guiadas pelo estado atual do progresso e o objetivo final [4]. Nesse contexto, surge a robótica cooperativa.

Robôs Cooperativos, tanto na indústria como em outros ambientes, devem conseguir realizar tarefas em conjunto e muitas vezes interagir com seres humanos. Essa tarefa exige que os robôs consigam planejar seus movimentos e ações [5, 6], que percebam o ambiente a sua volta, utilizando seus sensores, e que compartilhem informações, uns com os outros e/ou com a entidade controladora.

Para que o sistema interaja em harmonia, é importante que o robô tenha uma boa noção de sua localização e da localização dos outros agentes do ambiente. Nesse contexto, é relevante o estudo e o desenvolvimento de sistemas de percepção visual que utilizem informações providas por diversas câmeras.

1.1 Motivação

A Universidade de Brasília (UnB) e o Laboratório de Automação e Robótica (LARA) foram berço de várias pesquisas no desenvolvimento de novas tecnologias em robótica cooperativa. Tanto no controle de robôs manipuladores [7, 8] quanto no controle de robôs móveis [9].

Diversos trabalhos também pesquisaram abordagens de navegação e localização de robôs. Brito, em [10], estuda a realização de fusão sensorial de unidade de navegação inercial e *GPS*

para localização de robôs em ambientes externos. Já Santos, em [11], faz uso de filtragem bayesiana para localizar um robô usando informações providas pela câmera e sensores inerciais no contexto de futebol de robôs, tarefa que exige alto nível de cooperação.

Após a montagem de uma célula de indústria 4.0 no LARA, que conta com três robôs manipuladores e três robôs móveis sobre rodas, surgiu a necessidade de implementar um sistema que pudesse fazer o monitoramento por câmeras do ambiente e prover informações de posição e orientação, tanto dos agentes móveis, quanto de pessoas que frequentam o local. Esse sistema deve prover informações em uma linguagem que possa ser compreendida pelos controladores dos robôs, para auxiliar no planejamento de trajetória e reação.

1.2 Contextualização

Trabalhos com temática semelhante já implementaram estratégias de localização em que as câmeras ficam localizadas nos próprios agentes móveis, como estudado em [12] e [13]. Neste caso a câmera colocada no robô provê informação de pose utilizando técnicas de SLAM [1], por meio das características do ambiente. Neste trabalho, as câmeras não serão acopladas aos agentes, sendo posicionadas estaticamente no ambiente, como em [14], que desenvolve um sistema de estimação de pose de objetos rígidos por meio de múltiplas câmeras fixas de poses conhecidas.

Algumas ferramentas já são bastante consolidadas na área de localização, como o uso de marcadores para mapeamento e localização em ambientes internos, estudado em [15], e o uso de skeleton para rastreamento 3D de pose de agentes humanos [16]. Porém as soluções para os diferentes problemas não foram planejadas para interagir entre si e exigem um grande esforço para serem reunidas em um mesmo sistema. Além disso, os ambientes são frequentemente mutáveis, exigindo um dispendioso processo de configuração e calibração cada vez que um sensor é alterado ou acrescentado ao sistema.

O presente trabalho tem o objetivo de propor soluções para o problema de localização e mapeamento em ambientes internos, provendo funcionalidades de rastreamento de agentes móveis e agentes humanos e facilitando a sua utilização por meio do desenvolvimento de uma *API* [2] de fácil utilização, que siga os pilares da eficiência, escalabilidade, flexibilidade e modularidade. Além disso, este sistema também foi projetado para ser adaptável a qualquer ambiente e configurável a qualquer disposição de câmeras.

[1] Localização e Mapeamento Simultâneos

[2] *Application Programming Interface*: conjunto de rotinas e padrões de programação para acesso a um aplicativo de *software* ou plataforma.

1.3 Definição do Problema

Atualmente, o Laboratório de Automação e Robótica conta com uma célula de indústria 4.0 formada por três robôs móveis, do tipo *Pioneer* (Figura 1.1), e três robôs manipuladores, um *UR3* (Figura 1.2a), um *Meka* (Figura 1.2b) e um *Schunk* (Figura 1.2c).

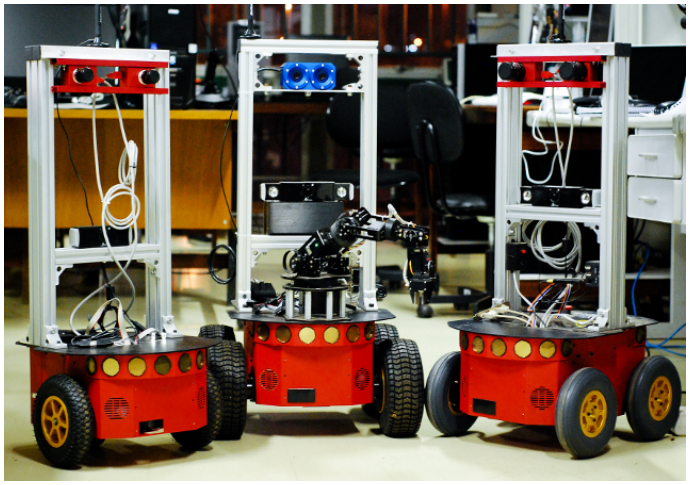
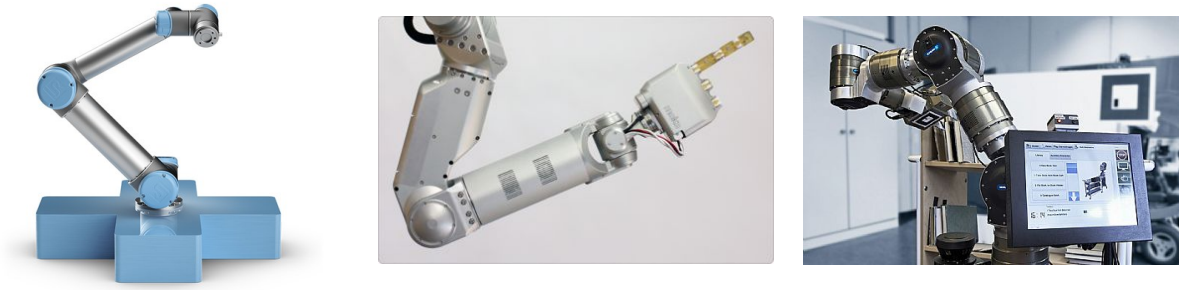


Figura 1.1: Robôs Pioneer. Fonte
`https://cooprobo.readthedocs.io/en/latest/mobile/pioneer.html`



(a) Robô *UR3* (b) Robô *Meka* (c) Robô *Schunk*

Figura 1.2: Robôs Manipuladores. Fonte:
`https://cooprobo.readthedocs.io/en/latest/manipulators.html`

Os robôs manipuladores representam as máquinas dessa célula e os robôs móveis, o transporte entre as máquinas. Todos os robôs devem contar com a interferência humana e agir de forma cooperativa. Se, por exemplo, uma pessoa estiver bloqueando a passagem de um robô móvel, ele deve perceber e evitar uma colisão. O mesmo vale para os robôs manipuladores, que devem parar ou mudar a sua trajetória caso haja risco de atingir alguém. Com isso em vista, o sistema de mapeamento e localização deve conseguir rastrear tanto os robôs, manipuladores e móveis, quanto as pessoas nesse local. Isso irá auxiliar tanto na execução correta das tarefas quanto na garantia

de segurança das partes envolvidas.

A percepção será feita usando múltiplas câmeras. O LARA conta com cinco câmeras do tipo *Microsoft Kinect XBox 360* (Figura 1.3a), que serão utilizadas para fazer o rastreamento de agentes móveis e humanos, e cinco câmeras *Videre Stereo* (Figura 1.3b), que serão usadas preferencialmente para fazer o rastreamento de agentes móveis. Essas câmeras são fixas, mas as suas posições não precisam ser conhecidas inicialmente.

(a) Microsoft Kinect XBox 360 (b) Videre Stereo

Figura 1.3: Câmeras. Fonte: `https://cooprobo.readthedocs.io/pt/latest/mobile/pioneer/hard/pioneer-cameras.html`

A solução proposta para o problema será dividida nas etapas: ambiente do sistema, rastreamento de agentes móveis, rastreamento de agentes humanos, reconhecimento facial, mapeamento, integração do rastreamento em referencial global e desenvolvimento da *API*.

1.3.1 Ambiente do Sistema

O ambiente de desenvolvimento do sistema proposto será o *ROS (Robot Operating System)* [17]. Ele é composto por um conjunto de *frameworks* de *software* de código aberto desenvolvido para facilitar e unificar o acesso a ferramentas para desenvolvimento em robótica. Basicamente, é um sistema dividido em nós em que mensagens são publicadas e recebidas em tópicos.

O *ROS* foi escolhido para esse projeto por possuir suporte a muitas ferramentas e *drivers* já consolidados, como acesso a diversos tipos de câmeras, módulo de detecção de marcadores e rastreamento de *skeleton* humano. Por ser de natureza *open source*[3] os módulos podem ser usados e modificados livremente.

Além disso, por meio do *ROS*, é possível fazer o processamento distribuído em diversas máquinas de forma escalável. Isso é essencial neste projeto, que conta com processamento de imagens de uma rede de câmeras. Dependendo da complexidade dos algoritmos e da quantidade de câmeras conectadas ao sistema, é importante poder dividir o trabalho e ainda manter a comunicação entre todas as partes.

Por fim, uma outra grande vantagem do *ROS* é a possibilidade de trabalhar com diferentes

[3]Código livre.

linguagens de programação. As bibliotecas cliente *rospy* [18] e *roscpp* [19] implementam o interfaceamento entre códigos em *python* e em *C++*, respectivamente, com o ambiente *ROS*. Assim, é possível utilizar funcionalidades e bibliotecas que são exclusivas de uma linguagem ou outra, permitindo um maior aproveitamento de recursos.

1.3.2 Rastreamento de Agentes Móveis

O rastreamento de agentes móveis é comumente realizado através de marcadores, como é o caso dos estudos feitos em [20] e [21], que se provaram eficientes. Os marcadores *ARUCO* [22] (Figura 1.4) foram escolhidos para fazer a estimação de posição e orientação dos robôs e o mapeamento do ambiente proposto nesse projeto.

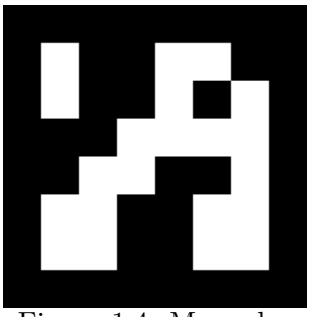


Figura 1.4: Marcador *ARUCO*

O marcador é um quadrado binário que possui um padrão único que o identifica. Ele pode ser encontrado em diversos tamanhos e sua borda preta ajuda na sua rápida detecção por algoritmos de visão computacional. Além disso, já existem bibliotecas que trazem as ferramentas para tratar esse problema, como a biblioteca *OpenCV* [23], que possui um módulo de detecção e estimação de pose desses marcadores.

Os robôs serão identificados por marcadores colados em seu exterior. A localização global dos marcadores será disponibilizada no ambiente *ROS* de forma que, posteriormente, a posição absoluta de cada parte do robô possa ser obtida através de sua cinemática.

1.3.3 Rastreamento de Agentes Humanos

A tarefa de rastreamento de agentes humanos é um pouco mais complexa que a tarefa de rastreamento de agentes móveis, porque uma pessoa pode ocupar o mesmo lugar no espaço de diversas maneiras. Por isso, o rastreamento será feito utilizando um modelo de corpo articulado *skeleton*, provido pelo *Kinect* [24, 25] e pelas bibliotecas *OpenNI* [26] e *NITE* [27]. O módulo *ROS openni_tracker* [28] utiliza a imagem da câmera e do sensor infravermelho para estimar as poses das juntas dos indivíduos em relação à câmera.

São retornadas as poses no espaço tridimensional de 15 juntas do corpo humano: cabeça, base do pescoço, ombros, cotovelos, mãos, torso, quadris, joelhos e pés. A posição 3D é dada em metros e a orientação de cada junta, em quatérnios. Unindo as poses encontradas, é montado um esqueleto para cada agente, como mostrado na Figura 1.5.

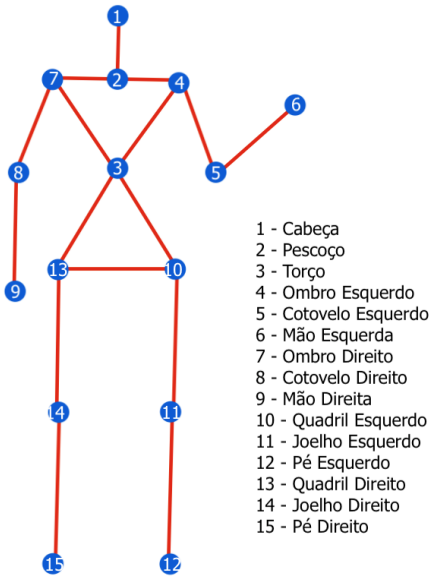


Figura 1.5: Skeleton Humano

1.3.4 Reconhecimento Facial

Apesar de o modelo de rastreamento de agentes humanos através de *skeleton* ser muito bom em diferenciar um indivíduo do outro, ele não é capaz de identificar as pessoas envolvidas na cena. Por isso será desenvolvido um módulo auxiliar que realizará o reconhecimento facial dos agentes humanos.

O desenvolvimento de algoritmos de reconhecimento facial sugerido em [29] divide o problema de reconhecimento nas seguintes etapas:

1. Captura da Imagem

   A imagem será capturada pelas câmeras que farão o rastreamento de agentes humanos.

2. Detecção de Face

   Antes de realizar o reconhecimento facial, é necessário realizar a detecção de rostos na imagem. Nesta etapa, existe um sistema que terá como entrada toda a imagem e, como saída, uma lista com as *bounding boxes*[4] das faces detectadas nessa imagem.

3. Extração de *Features*[5]

   Após a detecção de faces, as *bounding boxes* irão definir subimagens que contêm apenas um rosto. Essa subimagem será analisada por um algoritmo de detecção de *features* que identifica uma sequência de pontos chave da imagem e gera um identificador para cada um.

[4] *Bounding box:* caixa com a menor medida possível que englobe a área de interesse. Pode ser dada pelas coordenadas do canto superior esquerdo e do canto inferior direito.

[5] *Feature*: neste contexto, pode ser entendido como característica marcante.

4. Correspondência de *Features* da Base de Dados

   Para um conjunto de imagens de uma pessoa cadastrada, serão extraídas as *features* e armazenadas em uma base de dados. Ao realizar uma nova detecção, as *features* extraídas serão comparadas com as armazenadas na base de dados para encontrar correspondências.

5. Identificação

   Se ocorrer uma correspondência acima de um limiar preestabelecido de confiança, ocorre a identificação da face, finalizando o reconhecimento facial.

O módulo de reconhecimento facial proposto nesse trabalho tem o objetivo de identificar os agentes detectados pelo algoritmo de rastreamento. Para isso, será necessário verificar a base de dados de pessoas autorizadas a estarem no espaço e identificar o agente com o devido nome. Essa solução poderá, posteriormente, estabelecer o controle de acesso ao laboratório e criar alertas de segurança. Esse reconhecimento contará com um módulo de cadastro para que seja criada a base de dados de pessoas conhecidas no ambiente.

1.3.5 Mapeamento do Ambiente

O sistema proposto nesse trabalho não será atrelado ao ambiente do LARA, sendo uma solução geral que poderá ser reutilizada em qualquer ambiente. As câmeras, inicialmente, não têm suas posições conhecidas, mas precisarão dessa informação para realizar a etapa de integração em referencial global, por isso foi proposto o módulo de mapeamento.

Utilizando a mesma estratégia do rastreamento de agentes móveis, o mapeamento será feito com o auxílio dos marcadores *ARUCO*. As câmeras deverão possuir interseções nos seus campos de visão e os marcadores deverão ser colocados, prioritariamente, nesses espaços. Um desses marcadores, pelo menos, deverá ter a sua posição conhecida.

O módulo de mapeamento realizará iterações para tentar localizar o máximo de câmeras possível. O algoritmo que implementará essa solução será explicado em detalhes na Seção 3.3. A ideia é utilizar o marcador com posição conhecida para localizar a câmera que o visualiza e, em seguida, localizar os outros marcadores detectados por essa câmera. As próximas câmeras serão localizadas iterativamente.

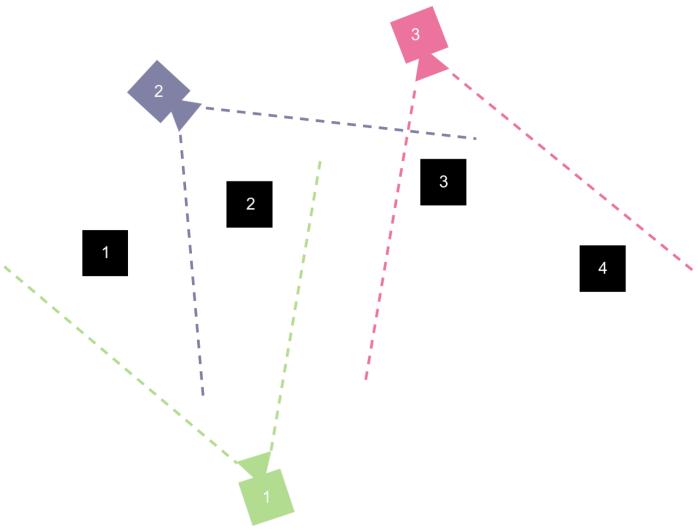


Figura 1.6: Módulo de Mapeamento

Tendo como exemplo o esquema da Figura 1.6 e sabendo que o marcador 1 possui uma pose definida no sistema de referência global, a câmera 1 visualiza o marcador 1, que é conhecido, logo ela consegue calcular a sua posição e se tornar uma câmera conhecida. A câmera 1 também visualiza o marcador 2, logo, como a pose dela é conhecida, ela consegue calcular a pose do marcador 2 e torná-lo conhecido. A câmera 2 visualiza o marcador 2, que é conhecido, portanto, ela consegue se localizar e localizar o marcador 3. A câmera 3 visualiza o marcador 3, que é conhecido, logo, consegue se localizar e localizar o marcador 4. Como não há mais câmeras no sistema, o algoritmo é finalizado com todas as câmeras localizadas.

1.3.6 Integração do Rastreamento em Referencial Global

Para cada câmera presente no sistema, as detecções e devidas estimações de pose são realizadas em relação ao sistema de coordenadas da própria câmera, que tem o seu centro como origem. Em um ambiente com múltiplas câmeras com interseções em seus campos de visão, é necessário que exista uma união dessas informações em um só referencial.

O referencial global é estabelecido pelo próprio usuário e é passado ao sistema através do módulo de mapeamento, quando a pose de pelo menos um marcador é dada como entrada. Todas as câmeras são mapeadas a partir do referencial global e, tendo suas poses como conhecidas, podem fazer a transformação de coordenadas do seu sistema para o sistema de referência global, tanto de detecções de marcadores quanto de juntas humanas.

Para fazer a fusão sensorial das detecções providas pelas diversas câmeras, será implementado um filtro bayesiano, que reunirá as informações de um mesmo objeto visualizado por mais de uma câmera em um mesmo identificador.

1.3.7 Desenvolvimento da *API*

A *API* proposta nesse projeto foi pensada para aproveitar ao máximo ferramentas *open source* já implementadas e ser de fácil utilização e manutenção. Para isso nada melhor que utilizar a linguagem *python*, que é de fácil compreensão e aprendizado, conta com várias bibliotecas e possui uma comunidade ativa que auxilia no desenvolvimento.

As soluções para os problemas de localização propostas até o momento foram implementadas separadamente, usando nós de *ROS*. A detecção de marcadores é feita pelo módulo *aruco_detect*, o rastreamento de pose pelo módulo *openni_tracker*, o reconhecimento facial, o mapeamento das câmeras e a integração em referencial global também possuem seus devidos nós de *ROS*. Cada nó pode ser implementado individualmente tanto em python quanto em C++, aproveitando o que há de melhor em cada linguagem.

Um nó do *ROS* pode ser executado por linha de comando através da instrução *rosrun*. Vários nós podem ser reunidos em um arquivo *ROS .launch* e executados de uma só vez por meio da instrução *roslaunch*. A biblioteca *rospy* possui funções que permitem chamar a execução tanto de nós individuais quanto de vários nós de um arquivo *.launch* por meio do *script* python. Essa foi a estratégia utilizada para sintetizar todas as ferramentas em uma só *API*.

Para cada módulo foi criada uma classe em python que, ao ser inicializada com os parâmetros de configuração passados pelo usuário, chama o nó ou nós de *ROS* através de um comando interno encapsulado, transparente para o usuário. Dessa forma, com apenas um *script* é possível controlar todas as câmeras, indicar quais tarefas cada uma irá realizar e colocar o sistema para funcionar.

1.4 Objetivos

O objetivo desse projeto é escrever uma *API* de fácil utilização, em *python*, que forneça informações de posição e orientação de marcadores e de juntas humanas em referencial global, reunindo informações providas por diferentes câmeras. Além disso, a *API* também trará um módulo de reconhecimento facial para identificação de indivíduos.

O sistema irá prover todas as informações usando o ambiente *ROS*, compatível com os robôs do LARA, para que possam ser utilizadas pelos seus controladores a fim de realizar tarefas e garantir a segurança do laboratório.

Capítulo 2

Fundamentação

Este capítulo apresenta os conceitos e fundamentos matemáticos que serão utilizados no desenvolvimento do trabalho. Primeiramente serão descritas as transformações de coordenadas em espaço 3D e a matemática dos quatérnios. Em seguida, será apresentada a teoria de calibração e estimação de pose a partir de câmeras pinhole. E por fim, será abordado o funcionamento do Filtro de Kalman Estendido que será utilizado para fazer a fusão sensorial e estimação de pose em referencial global.

2.1 Transformação de Coordenadas 3D

Um ponto p representado em um sistema de coordenadas $o_2x_2y_2z_2$, mostrado na Figura 2.1, pode ser representado no sistema $o_1x_1y_1z_1$ por meio da transformação

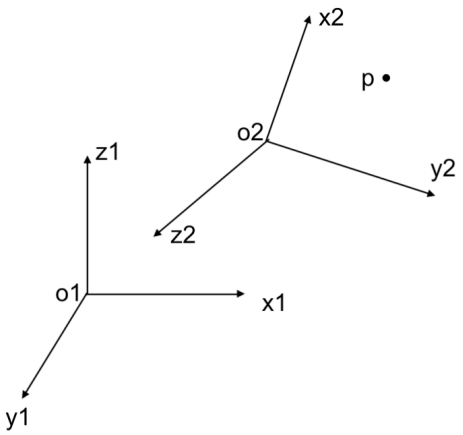


Figura 2.1: Representação em diferentes sistemas de coordenadas

$$p^1 = R_2^1 p^2 + t_2^1, \tag{2.1}$$

sendo R_2^1 a matriz de rotação (3x3) do sistema 2 para o sistema 1 e t_2^1 o vetor de translação (3x1) da origem do sistema 2 para o sistema 1.

Consequentemente, também é possível fazer a transformação inversa

$$p^2 = R_1^2 p^1 + t_1^2, \tag{2.2}$$

em que $R_1^2 = {R_2^1}^{-1}$ é matriz de rotação (3x3) do sistema 1 para o sistema 2 e $t_1^2 = -{R_2^1}^{-1} t_2^1$ é o vetor de translação (3x1) da origem do sistema 1 para o sistema 2. Essas relações são obtidas realizando a multiplicação da Equação 2.1 por ${R_2^1}^{-1}$ e manipulando os termos.

2.1.1 Representação de rotação por meio de quatérnios

Quatérnios, descritos pelo matemático William Rowan Hamilton em [30], são uma extensão dos números complexos desenvolvida especialmente para representação de rotações em espaço tridimensional.

Um quatérnio unitário é um número do tipo

$$q_w + q_x i + q_y j + q_z k, \tag{2.3}$$

que respeita a propriedade

$$q_w^2 + q_x^2 + q_y^2 + q_z^2 = 1. \tag{2.4}$$

Uma rotação θ em torno de um eixo dado pelo vetor unitário $n = [n_x \quad n_y \quad n_z]^T$ pode ser representada pelo quatérnio unitário

$$q = cos(\theta) + n_x sen(\theta) i + n_y sen(\theta) j + n_z sen(\theta) k. \tag{2.5}$$

Para realizar a rotação expressa em q de um vetor qualquer v, dado na forma $v = v_x i + v_y j + v_z k$, faz-se a multiplicação

$$u = q \cdot v \cdot q^*, \tag{2.6}$$

sendo q^* o conjugado de q, dado por $q^* = q_w - q_x i - q_y j - q_z k$.

A multiplicação deve ser realizada respeitando as definições:

$$ii = jj = kk = -1 \tag{2.7}$$
$$ij = -ji = k \tag{2.8}$$
$$jk = -kj = i \tag{2.9}$$
$$ki = -ik = j. \tag{2.10}$$

Uma matriz de rotação R na forma

$$R = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \tag{2.11}$$

pode ser representada pelo quatérnio $q = q_w + q_x i + q_y j + q_z k$ [31], tal que

$$q_w = \frac{r_{11} + r_{22} + r_{33} + 1}{4}, \tag{2.12}$$

$$q_x = \frac{r_{32} - r_{23}}{r_{11} + r_{22} + r_{33} + 1}, \tag{2.13}$$

$$q_y = \frac{r_{13} - r_{31}}{r_{11} + r_{22} + r_{33} + 1}, \tag{2.14}$$

$$q_z = \frac{r_{21} - r_{12}}{r_{11} + r_{22} + r_{33} + 1}. \tag{2.15}$$

Logo, a equação de transformação de sistemas de coordenadas expressa em 2.1 pode ser reescrita, utilizando o quatérnio de rotação, como

$$p^1 = q_2^1 \cdot p^2 \cdot q_2^{1*} + t_2^1, \tag{2.16}$$

em que p^1, p^2 e t_2^1 são vetores v expressos na forma $v = v_x i + v_y j + v_z k$.

É importante notar que um quatérnio unitário varre o espaço de orientação 3D duas vezes, logo, uma rotação expressa por q é a mesma expressa por $-q$.

2.2 Câmeras *Pinhole*

Câmeras são sensores capazes de transformar a perspectiva 3D do mundo em uma perspectiva 2D da imagem. O modelo mais genérico e simples que especifica essa transformação é o modelo de câmera *pinhole* [32]. Para esse tipo de câmera, é possível determinar os seus parâmetros intrínsecos e extrínsecos e, a partir deles, usar as informações providas pela câmera para mapear o mundo real.

2.2.1 Parâmetros Intrínsecos

Os parâmetros intrínsecos da câmera são aqueles que são inerentes a ela. Com um modelo de câmera pinhole, em que não há distorções, é possível mapear um ponto $Q = (X, Y, Z)^T$ para um ponto $(x, y)^T$ no plano da imagem. Escrevendo as coordenadas dos pontos em coordenadas homogêneas, temos a equação

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} f\eta_x & 0 & c_x & 0 \\ 0 & f\eta_y & c_y & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} = [K \mid 0_3] \cdot \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}. \tag{2.17}$$

Na Equação (2.17) f é a distância focal, $(c_x, c_y)^T$ são as coordenadas do ponto focal da câmera, e η_x e η_y são o número de pixels por unidade de comprimento na direção x e na direção y. A matriz K é a matriz de parâmetros intrínsecos e O_3 é um vetor nulo concatenado a K.

Esses parâmetros de câmera podem ser fornecidos diretamente pelo fabricante, mas, caso não sejam, podem ser estimados por meio de calibração.

Como a maioria das câmeras possui lentes, o uso direto de matriz de parâmetros intrínsecos pode acarretar em erros de distorção. Um ponto (x, y) na imagem distorcida é mapeado para um ponto $(\hat{x}, \hat{y})$ na imagem não distorcida a partir do centro de distorção x_c, y_c pela relação

$$\hat{x} = x_c + L(r) \cdot (x - \hat{x}) \qquad \hat{y} = y_c + L(r) \cdot (y - \hat{y}), \tag{2.18}$$

em que $L(r)$ é uma função que aproxima a distorção por uma série de Taylor, dada por

$$L(r) = 1 + k_1 r + k_2 r^2 + k_3 r^3 + ... \tag{2.19}$$

Os parâmetros $k_1, k_2, k_3, ..., x_c, y_c$ são chamados de parâmetros de distorção e também podem ser estimados por calibração.

2.2.2 Parâmetros Extrínsecos

Os parâmetros extrínsecos da câmera se referem a sua pose em relação a um sistema fixo de coordenadas do mundo. Essa propriedade se dá pela relação

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} + \begin{bmatrix} t_1 \\ t_2 \\ t_3 \end{bmatrix} = R \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} + t = \Big[R|t\Big] \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}. \tag{2.20}$$

Conhecendo a posição de certos pontos no sistema de coordenadas global e visualizando estes mesmos pontos na câmera, é possível obter a matriz de rotação e o vetor de translação da pose da câmera em relação ao sistema de coordenadas global.

A matriz R pode ser representada por meio de quatérnio através das relações expressas em 2.12, 2.13, 2.14 e 2.15.

2.3 Filtro de Kalman Estendido

O Filtro de Kalman [33] é um modelo matemático que propõe gerar estimativas de medidas contaminadas por ruídos que sejam mais próximas do valor real, ou seja, com um erro menor. O sistema realiza estimativas por meio de uma média ponderada dos valores medidos por sensores e preditos pelo modelo dinâmico.

O Filtro de Kalman Estendido (EKF) é a variação desse filtro que lineariza um sistema não linear em torno de um certo ponto de operação.

O sistema discreto do EKF pode ser representado por

$$\begin{aligned} x_k &= f\left(x_{k-1}, u_k\right) + \omega_k \\ z_k &= h(x_k) + \upsilon_k \end{aligned}, \tag{2.21}$$

em que f é a função de transição do sistema e h é a função de observação, essas funções não precisam ser lineares. Já ω_k e υ_k são, respectivamente o ruído de processo e o ruído de observação, modelados como um erro gaussiano de média zero. A covariância desses erros é representada, respectivamente, pelas matrizes $\boldsymbol{Q}_k$ e $\boldsymbol{R}_k$.

O Filtro de Kalman Estendido possui 3 etapas, são elas:

- Predição

  Nessa etapa, é calculada a predição da estimação do próximo estado, usando a estimativa anterior e as equações do modelo

  $$\hat{\boldsymbol{x}}_{k|k-1} = \boldsymbol{F}_k \hat{\boldsymbol{x}}_{k-1|k-1} + \boldsymbol{G}_k \hat{\boldsymbol{u}}_{k-1|k-1}, \tag{2.22}$$

  em que $\hat{x}_{k|k-1}$ é a predição da estimação do próximo estado, F_k é a matriz jacobiana da função de transição e G_k é a matriz jacobiana da função de controle. Como não ocorrerá sinal de controle, o sistema será simplificado para

  $$\hat{\boldsymbol{x}}_{k|k-1} = \boldsymbol{F}_k \hat{\boldsymbol{x}}_{k-1|k-1}. \tag{2.23}$$

  Também é calculada a matriz de covariância $\boldsymbol{P}$, dada pela fórmula

  $$\boldsymbol{P}_{k|k-1} = \boldsymbol{F}_k \boldsymbol{P}_{k-1|k-1} \boldsymbol{F}_k^T + \boldsymbol{Q}_k. \tag{2.24}$$

- Medição

  Aquisição do vetor de medições $\boldsymbol{z}_k$.

- Correção

  Nessa etapa o valor medido e o valor predito são comparados. O valor final da estimativa levará em conta a covariância e também o ruído de processo e o ruído de medição.

  Primeiramente, calcula-se o Ganho de Kalman $\boldsymbol{K}_k$

  $$\boldsymbol{K}_k = \boldsymbol{P}_{k|k-1} \boldsymbol{H}_k^T \left(\boldsymbol{H}_k \boldsymbol{P}_{k|k-1} \boldsymbol{H}_k^T + \boldsymbol{R}_k \right)^{-1}, \tag{2.25}$$

  em que $\boldsymbol{H}_k$ é a matriz jacobiana da função de observação h.

  Após calcular o Ganho de Kalman, ele é usado junto com a medição para calcular o vetor de estados estimado

  $$\hat{\boldsymbol{x}}_{k|k} = \hat{\boldsymbol{x}}_{k|k-1} + \boldsymbol{K}_k \left(\boldsymbol{z}_k - \boldsymbol{H}_k \hat{\boldsymbol{x}}_{k|k-1} \right). \tag{2.26}$$

Por fim, ocorre a correção da matriz de covariância, dada por

$$\boldsymbol{P}_{k|k} = (\boldsymbol{I} - \boldsymbol{K}_k\boldsymbol{H}_k)\boldsymbol{P}_{k|k-1}(\boldsymbol{I} - \boldsymbol{K}_k\boldsymbol{H}_k)\boldsymbol{P}^T + (\boldsymbol{K}_k\boldsymbol{R}_k\boldsymbol{K}_k^T). \tag{2.27}$$

Capítulo 3

Desenvolvimento

Este capítulo apresenta como foram desenvolvidas as ferramentas que compõem cada módulo do sistema e como foram feitos os experimentos que validam a solução. Primeiramente, é apresentada a configuração do ambiente e adaptação necessária para realização dos experimentos. Em seguida, são desenvolvidos os temas relacionados à detecção e rastreamento dos marcadores, incluindo o mapeamento do ambiente. Logo após, são expostos os temas relacionados à detecção, rastreamento e reconhecimento de agentes humanos. Por fim, é descrita a utilização da API.

3.1 Configuração do Ambiente

O ambiente para qual esse sistema foi pensado é o LARA. Como já comentado na Seção 1.3, o LARA possui cinco câmeras *Microsoft Kinect XBox 360* e cinco câmeras *Videre Stereo*.

1. ***Microsoft Kinect Xbox 360***

   Microsoft Kinect Xbox 360 é um sensor que detecta movimentos e pose de pessoas, e foi originalmente projetado para permitir interatividade com jogos através do próprio corpo. Ele possui uma câmera RGB e uma câmera infravermelho para estimação de distância.

   Suas especificações são:

   - Resolução da câmera RGB: 640 x 480
   - Resolução da câmera de profundidade (infravermelho): 320 x 240
   - Mínima profundidade: 0.04 m
   - Máxima profundidade: 4.50 m
   - Campo de visão: 43°-Vertical 57°-Horizontal
   - Ângulo de ajuste vertical: ±27°

- Frame rate: 30 fps

Para conseguir acessar as imagens e informações das câmeras *Kinect*, é necessária a instalação do *driver SensorKinect* [34] e da biblioteca *OpenNI* [26].

2. **Câmera *Videre Stereo***

   As câmeras *Videre Stereo* são muito antigas e os *drivers* fornecidos pelo fabricante foram descontinuados. Para capturar as imagens dessa câmera, foi necessário o uso do *driver Camera1394* [35] para câmeras *firewire*. Esse *driver* não é específico para essa câmera e as imagens capturadas possuíam linhas pretas horizontais que precisavam ser removidas para não prejudicar os algoritmos de visão computacional.

   Por isso foi feito um módulo *ROS* que escuta o tópico do *frame* da câmera e faz o processamento de remoção de linhas. A imagem resultante possui resolução 640x320 e é monocromática, como pode ser visto na Figura 3.1.

(a) Imagem com linhas (b) Imagem filtrada

Figura 3.1: Imagem da câmera Videre Stereo

Por conta da indisponibilidade de acessar o laboratório, algumas adequações precisaram ser feitas ao projeto inicial. A principal foi a adoção de um sistema de testes em simulação. Foi escolhido o simulador *CoppeliaSim* [36], antigo *V-REP*, para realização de testes no sistema. Esse simulador usa a arquitetura de controle distribuído, possui suporte a controladores de diferentes plataformas e possibilita a adaptação de diferentes cenários, já provendo várias ferramentas, como os sensores visuais.

Para poder se aproximar ao máximo da realidade, foi projetado um ambiente no simulador baseado no LARA, mostrado na Figura 3.2. Nesse ambiente simulado, foram posicionadas câmeras que serão utilizadas para fazer o mapeamento do ambiente e o posterior rastreamento de agentes móveis e humanos.

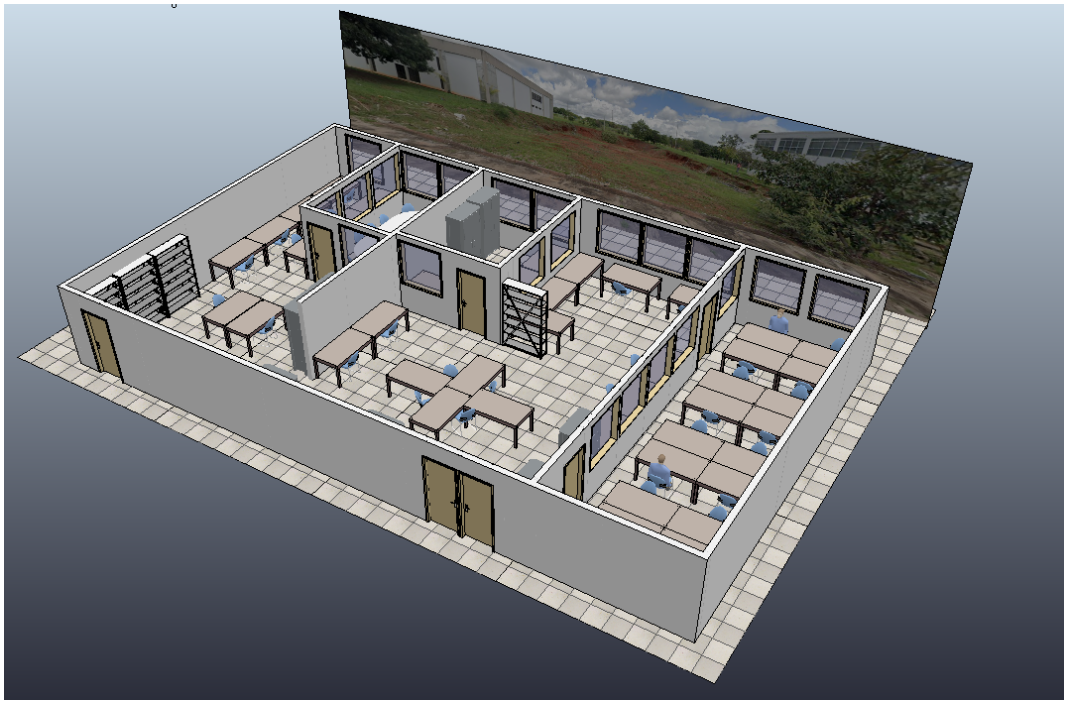


Figura 3.2: Ambiente do LARA montado no simulador

O simulador *CoppeliaSim* possui um sensor visual, como o mostrado na Figura 3.3, que será configurado e utilizado como câmera durante a simulação. Essa câmera deve disponibilizar via *ROS* tanto as imagens quanto as informações de calibração interna por meio de um *script* individual.

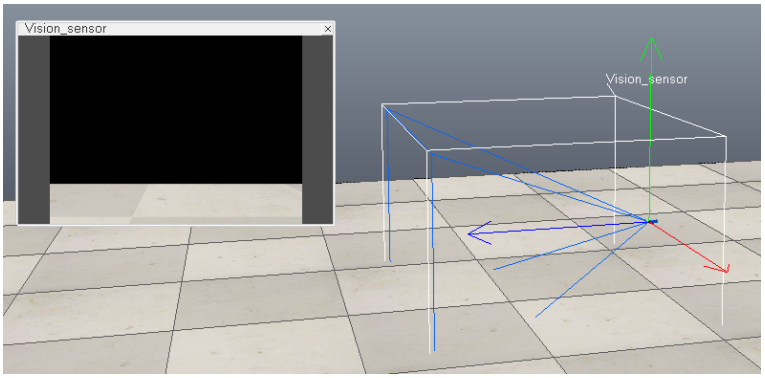


Figura 3.3: Câmera simulada

Para essa câmera são definidas a largura (w) e altura (h) em pixels da imagem e o ângulo de abertura (α). São calculados os parâmetros de distância focal f_x e f_y e centro c_x e c_y, que compõem a matriz de parâmetros intrínsecos, apresentada na Seção 2.2.1, pelas relações

$$f_x = \frac{w}{2} tan\left(\frac{\alpha}{2}\right), \tag{3.1}$$

$$f_y = f_x, \tag{3.2}$$

$$c_x = \frac{w}{2}, \tag{3.3}$$

$$c_y = \frac{h}{2}. \tag{3.4}$$

A matriz de parâmetros intrínsecos é publicada, juntamente com as imagens, para a devida câmera. A imagem proveniente deste sensor simulado não possui distorção.

3.2 Detecção e Estimação de Pose de Marcadores *ARUCO*

Os marcadores *ARUCO*, já comentados na Seção 1.3.2, possuem bordas pretas que geralmente promovem um forte contraste com o ambiente, facilitando a detecção por extração simples de *features*. Além disso, cada marcador conta com um código binário único, que é capaz de o identificar, como também identificar ordenadamente seus vértices.

O pacote ROS *aruco_detect* é *open source* e realiza detecção, estimação de pose e publicação de posição 3D, em metros, e orientação, em quatérnios, dos marcadores *ARUCO* no ambiente *ROS*.

A operação realizada por esse módulo considera que o centro do marcador é a origem de um sistema de coordenadas, como mostrado na Figura 3.4.

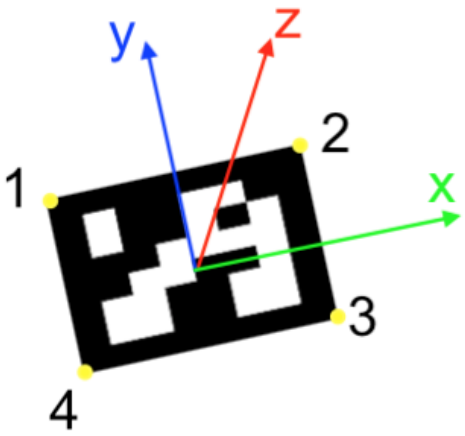


Figura 3.4: Sistema de Coordenadas do Marcador *ARUCO*

Logo, os seus 4 vértices podem ser descritos pelas coordenadas tridimensionais

$$p_1 = \left(-\frac{l}{2}, \frac{l}{2}, 0\right), \tag{3.5}$$

$$p_2 = \left(\frac{l}{2}, \frac{l}{2}, 0\right), \tag{3.6}$$

$$p_3 = \left(\frac{l}{2}, -\frac{l}{2}, 0\right), \tag{3.7}$$

$$p_4 = \left(-\frac{l}{2}, -\frac{l}{2}, 0\right), \tag{3.8}$$

em que l é a largura, em metros, do marcador.

Por meio do algoritmo $Efficient\ Perspective{-}n{-}Point$, descrito por Lepetit em [37], e os parâmetros de calibração intrínsecos da câmera, é possível encontrar a matriz R e o vetor t de parâmetros extrínsecos da câmera em relação à origem do centro do marcador. Esse processo é repetido para todos os marcadores detectados. Fazendo a transformação inversa, dada pela Equação 2.2, é possível encontrar a pose dos marcadores em relação à câmera. São fornecidas sete variáveis de pose para cada marcador: as coordenadas x, y e z de translação e as componentes q_x, q_y, q_z, q_w do quatérnio de orientação.

3.2.1 Validação Experimental

Para mostrar como é realizada a detecção de marcadores *ARUCO* e como os dados são recebidos, foi realizado um experimento de detecção utilizando uma câmera *Kinect* real e um marcador em movimento seguindo uma trajetória determinada pela Figura 3.5. O marcador faz um movimento de translação retangular na imagem e em seguida uma rotação no sentido horário no centro da imagem. Os resultados obtidos podem ser visualizados na Seção 4.1.

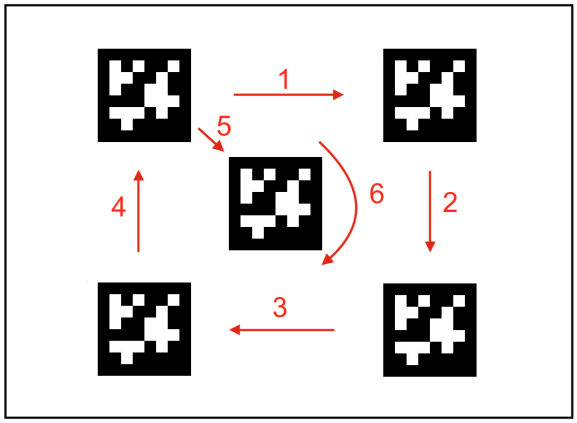


Figura 3.5: Trajetória seguida pelo Marcador

3.3 Mapeamento

O mapeamento do ambiente se resume a encontrar os parâmetros intrínsecos e extrínsecos de todas as câmeras do sistema. Reunindo esses parâmetros é possível obter a localização de um

objeto se ele estiver no campo de visão de alguma das câmeras.

Primeiramente, deve-se obter os parâmetros intrínsecos de cada câmera. Eles podem ser fornecidos pelo fabricante ou encontrados experimentalmente usando a ferramenta de calibração *camera_calibration* [38]. Essa ferramenta captura várias imagens contendo um tabuleiro de xadrez com dimensões conhecidas para calcular a matriz de parâmetros intrínsecos e os coeficientes de distorção da câmera.

A API proposta nesse projeto contará com um módulo de autocalibração dos parâmetros extrínsecos de todas as câmeras. O algoritmo de solução proposto tem o objetivo de minimizar ao máximo as informações iniciais que devem ser passadas pelo usuário. As entradas do algoritmo são apenas os nomes das câmeras que fazem parte do ambiente a ser mapeado, o nome dos marcadores que são fixos no ambiente e uma lista contendo a pose conhecida de pelo menos um desses marcadores.

A ideia é que um marcador seja escolhido como a origem do sistema de coordenadas do mundo e as câmeras possuam interseções nos seus campos de visão. Nessas interseções, serão colocados marcadores fixos que não precisam, necessariamente, possuir as suas poses conhecidas.

O algoritmo foi separado nas seguintes fases:

1. **Medição**

   Nesta etapa, as câmeras fazem a captura da imagem, e o algoritmo de detecção e estimação de pose dos marcadores é executado por um intervalo de tempo, armazenando as informações de posição e orientação, em quatérnios, de cada medição. Em seguida é calculada uma média entre as várias medições para minimizar o efeito de *outliers*[1].

   Para posição 3D, foi realizada a média aritmética dos vetores tridimensionais, dada por

   $$p = \sum_{i=1}^{N} p_i. \tag{3.9}$$

   Já para os quatérnios, foi utilizada uma média adaptada para o seu domínio, baseada no procedimento descrito por [39]. Essa média garante que o quatérnio resultante seja unitário.

2. **Mapeamento**

   No começo dessa etapa, todas as câmeras têm a sua posição desconhecida e deseja-se encontrar a pose das câmeras em relação ao sistema de coordenadas fixo global.

   Cada câmera possui um conjunto de medições de pose de marcadores em relação ao sistema de coordenadas da própria câmera e existe um vetor com a pose de pelo menos um marcador que é conhecida.

   O algoritmo realizará iterações enquanto existirem câmeras ainda não mapeadas e o número de câmeras não mapeadas de uma iteração para a outra diminui. Nessas iterações, as medições realizadas na etapa passada são verificadas. Caso a câmera visualize um marcador

[1] São valores que fogem da normalidade e podem causar desequilíbrio nos resultados obtidos

conhecido, deve realizar uma transformação de coordenadas que a leve para o sistema de coordenadas global.

O $q_{marcador\_medido}$ e o $t_{marcador\_medido}$ são, respectivamente, o quatérnio que representa a orientação do marcador em relação a câmera e o vetor de translação do marcador em relação a câmera. Já o $q_{marcador\_real}$ e o $t_{marcador\_real}$ são, respectivamente, o quatérnio que representa a orientação global do marcador e o vetor de translação global do marcador. É possível encontrar q_{camera} e t_{camera} da câmera em relação ao sistema de coordenadas global a partir de

$$q_{camera} = (q_{marcador\_medido}{}^{*})(q_{marcador\_real}), \tag{3.10}$$

$$t_{camera} = (q_{marcador\_medido}{}^{*})(t_{marcador\_medido})(q_{marcador\_medido}) + t_{marcador\_real}. \tag{3.11}$$

Após ter sua pose conhecida, a câmera é retirada da lista de câmeras desconhecidas e todos os outros marcadores vistos por ela tem a sua pose global calculada por

$$q_{marcador\_real} = (q_{camera})(q_{marcador\_medido}), \tag{3.12}$$

$$t_{marcador\_real} = (q_{camera})(t_{marcador\_medido})(q_{camera}{}^{*}) + t_{camera}. \tag{3.13}$$

O algoritmo de mapeamento pode ser visualizado no fluxograma da Figura 3.6.

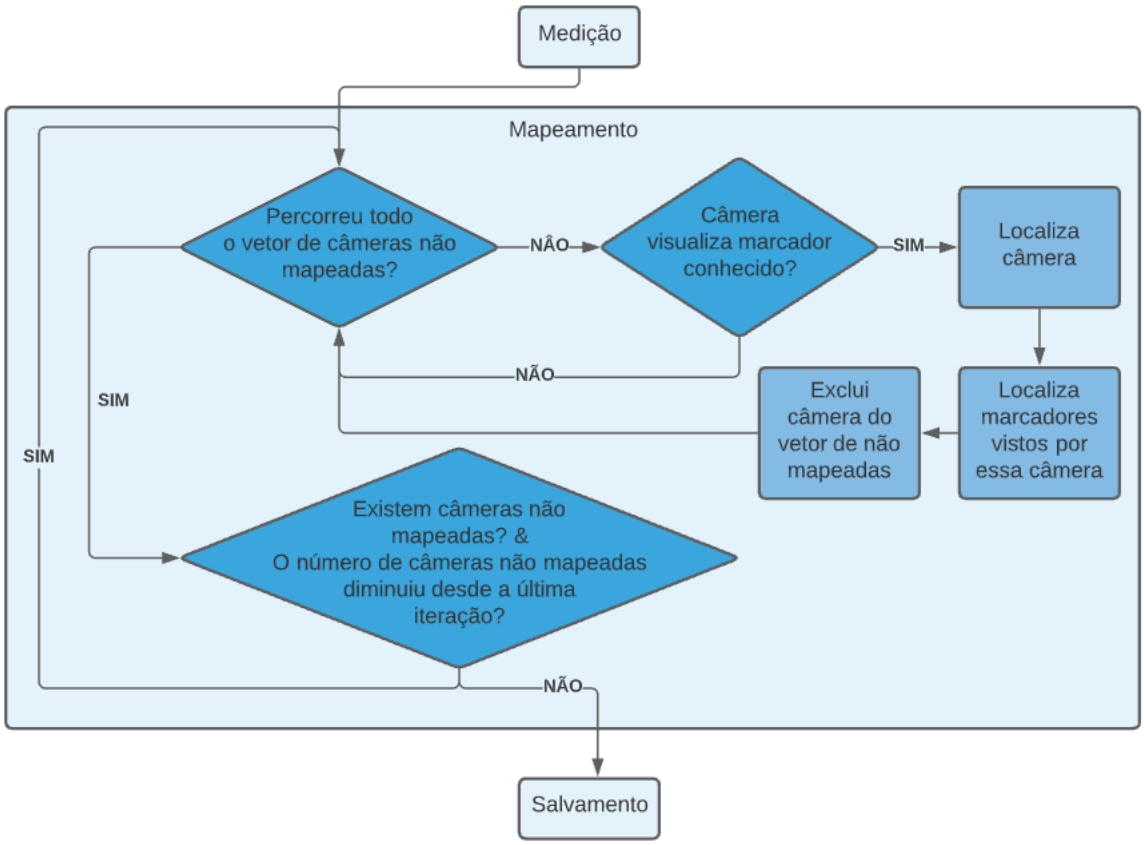


Figura 3.6: Algoritmo de Mapeamento de Câmeras

3. **Salvamento**

   Ao final, o algoritmo deve salvar um arquivo *json* com os identificadores e poses das câmeras que foram localizadas. Esse arquivo será utilizado para fazer a integração em referencial global dos rastreamentos de marcadores e pessoas. Caso as câmeras já possuam poses conhecidas, a sintaxe do arquivo *json* também facilita o preenchimento manual das informações. O algoritmo de mapeamento também produz um arquivo de *log*, que informa as câmeras que conseguiram, ou não, ser localizadas. Isso auxilia o usuário caso seja necessário rearranjar as posições dos marcadores e refazer o mapeamento.

3.3.1 Validação experimental

Para avaliar a performance desse algoritmo, foi proposto um experimento com o objetivo de observar como o erro de estimativa de posição e orientação se comporta ao longo das iterações, já que a pose de uma câmera é estimada a partir da pose, também estimada, de marcadores, acumulando erros.

Para isso foi criada uma cena no simulador *CoppeliaSim* com uma cadeia de dez câmeras. Essas câmeras foram posicionadas a 50 cm de altura e aproximadamente 40 cm de distância uma da outra, em uma angulação não retilínea com o plano do chão. Elas possuem interseções nos campos de visão e em cada interseção foi colocado um marcador *ARUCO*, como pode ser verificado na Figura 3.7.

O mapeamento foi executado e as estimativas de pose foram comparadas com as poses conhecidas fornecidas pelo simulador. Os resultados desse experimento podem ser observados na Seção 4.2.

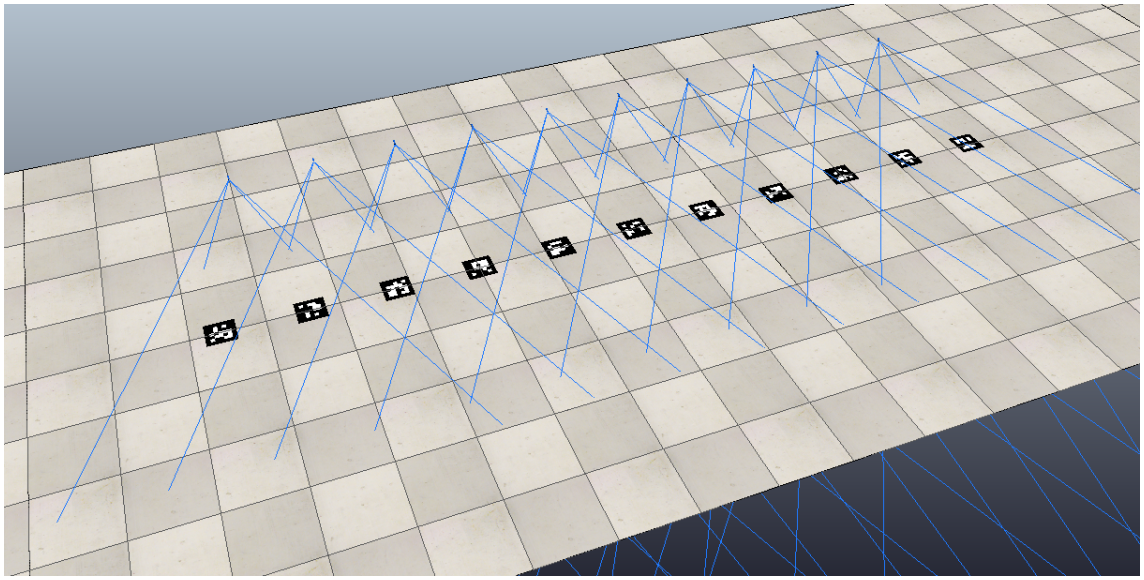


Figura 3.7: Simulação de Mapeamento

3.4 Rastreamento em Referencial Global das Detecções de Marcadores

A integração em referencial global dos marcadores deve fazer a transformação de coordenadas das estimativas realizadas em relação a cada câmera para o referencial global, determinado pelo mapeamento. Além disso, deve realizar a fusão sensorial por meio da filtragem bayesiana, usando Filtro de Kalman Estendido (EKF).

3.4.1 Transformação de Coordenadas

O módulo de detecção dos marcadores publica em um tópico *ROS*, para cada detecção, o identificador do marcador, a câmera que realizou a detecção e a pose estimada de cada marcador em relação à câmera. O módulo de rastreamento em referencial global dos marcadores, chamado de *Global_Marker_Tracker*, escuta este tópico e realiza, primeiramente, a transformação de coordenadas

$$q_{marcador\_referencial\_global} = (q_{camera})(q_{marcador\_referencial\_camera}), \tag{3.14}$$

$$t_{marcador\_referencial\_global} = (q_{camera})(t_{marcador\_referencial\_camera})({q_{camera}}^*) + t_{camera}, \tag{3.15}$$

usando o arquivo de configuração obtido durante o mapeamento com as poses de todas as câmeras e a pose do objeto visualizado.

Em seguida, é realizada a fusão sensorial e filtragem por meio de um EKF.

3.4.2 Modelagem do EKF

A modelagem do Filtro de Kalman Estendido exige que seja definida a dinâmica do sistema, que envolve tanto posição quanto orientação. Para posição, o modelo dinâmico escolhido para representar o sistema foi um modelo de aceleração zero, ou seja, de velocidade constante.

Se a posição em x, y e z varia a uma velocidade constante entre cada medição, é possível modelar o seu comportamento por

$$p_k = p_{k-1} + v_{k-1}\Delta t, \tag{3.16}$$

em que p representa os estados de posição x, y e z, e v representa os estados de velocidade v_x, v_y e v_z.

Para a modelagem da dinâmica dos quatérnios, decidiu-se adotar um sistema de velocidade angular zero. Logo, o comportamento da dinâmica dos quatérnios é dado por

$$q_k = q_{k-1}. \tag{3.17}$$

Com isso, o vetor de estados do sistema é

$$\begin{bmatrix} x & y & z & v_x & v_y & v_z & q_x & q_y & q_z & q_w \end{bmatrix}^T. \tag{3.18}$$

Já o jacobiano da função de transição é

$$\boldsymbol{F}_k = \begin{bmatrix} 1 & 0 & 0 & \Delta t_k & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & \Delta t_k & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & \Delta t_k & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}. \tag{3.19}$$

E por fim, o jacobiano da matriz de observação é

$$\boldsymbol{H}_k = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}. \tag{3.20}$$

Até o momento, o filtro tratou apenas de funções lineares, porém, os quatérnios não podem ser apenas filtrados sem levar em conta o fato de que são variáveis dependentes umas das outras. A equação que explicita a dependência entre as coordenadas dos quatérnios é

$$q_x^2 + q_y^2 + q_z^2 + q_w^2 = 1. \tag{3.21}$$

Para garantir a unicidade do quatérnio, a correção da estimação do filtro é feita usando uma segunda etapa de correção em que uma pseudo-medição é simulada [40].

Essa pseudo-medição recebe como medida a variável escalar de unicidade $y = 1$, com um erro de medição muito pequeno.

A matriz de observação da pseudo-medição é dada pelo jacobiano da norma do vetor

$$\boldsymbol{H2}_k = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 2q_x & 2q_y & 2q_z & 2q_w \end{bmatrix}. \tag{3.22}$$

Logo, a cada medição, duas etapas de correção são realizadas. Uma para estimar os estados do filtro e outra para corrigir a unicidade dos quatérnios. Em cada uma, é calculado um ganho de Kalman e atualizada a matriz de covariância.

3.4.3 Implementação do EKF

O EKF realiza as três etapas já explicadas na Seção 2.3, mas, antes disso, precisa ser inicializado. Foi escolhido usar a primeira medição do sensor como parâmetro de inicialização do filtro. As matrizes de erro de processo e erro de medição são inicializadas como matrizes diagonais com variâncias estabelecidas para cada componente. Esses valores podem ser alterados dependendo da configuração realizada.

A etapa de medição é composta pelo recebimento da detecção e a transformação de coordenadas para o referencial global. Além disso, por causa da representação equivalente entre o quatérnio e o seu oposto, é necessário verificar se houve mudança mudança no sinal da medição.

São calculadas as distâncias euclidianas entre o quatérnio medido e a última estimativa, e entre o oposto do quatérnio medido e a última estimativa

$$d_1 = |q_{medido} - q_{estimado}|\,, \tag{3.23}$$

$$d_2 = |-q_{medido} - q_{estimado}|\,. \tag{3.24}$$

Se $d_1 < d_2$, quer dizer que houve uma mudança de sinal do quatérnio de rotação, que precisará ser corrigido antes de ser usado como medição.

Usando o instante de tempo da medição, é feita uma predição para os dez estados do sistema e para a matriz de covariância pelas Equações 2.23 e 2.24, respectivamente.

Em seguida ocorre a correção, em que é calculado o ganho de Kalman pela Equação 2.25, a estimativa de estado pela Equação 2.26 e a covariância pela Equação 2.27. Por fim, ocorre uma segunda etapa de correção, para garantir a unicidade do quatérnio. Ela é feita supondo uma pseudo-medição e repete os mesmos passos da primeira correção.

3.4.4 Validação Experimental

Para validar o algoritmo proposto, foi montada uma cena no simulador *CoppeliaSim* que pretende se aproximar ao máximo do ambiente do LARA. Foram necessárias 13 câmeras para monitorar completamente os espaços livres do salão principal do laboratório, como pode ser verificado pela Figura 3.8.

Foi executado o módulo de mapeamento e as câmeras foram localizadas de acordo com o referencial global. Em seguida, apenas um marcador foi movimentado pelo campo de visão das câmeras e sua posição foi rastreada e comparada com a posição real. Foram realizados diferentes tipos de movimento para verificar o comportamento do filtro. Os resultados desse experimento podem ser verificados na Seção 4.3.

Figura 3.8: Simulação de Rastreamento

3.5 Rastreamento de Agentes Humanos

Como já mencionado na Seção 1.3.3, o rastreamento de agentes humanos é feito pelo pacote *ROS openni_tracker* com algumas alterações para prover suporte a múltiplos *Kinects*. São retornadas as poses no espaço tridimensional de 15 juntas do corpo humano, como mostrado na Figura 1.5, em relação à câmera que as visualiza. Cada pessoa detectada recebe um número identificador para se diferenciar das outras.

A autocalibração inclusa na biblioteca *OpenNI* evita que seja necessário realizar uma pose específica para o início do rastreamento. Logo, assim que o usuário entra em cena, já pode ser rastreado pelo algoritmo.

3.5.1 Validação Experimental

Para exemplificar os resultados de rastreamento deste módulo, foi proposto um simples experimento com o objetivo de verificar se o algoritmo era capaz de seguir movimentos e como são gerados e transmitidos os dados.

Uma pessoa se posicionou em frente à câmera a aproximadamente 2m de distância e movimentou os braços para cima em forma de arco, como mostrado na Figura 3.9. Os resultados desse experimento podem ser visualizados na Seção 4.4.

Figura 3.9: Movimento Realizado no Experimento de Rastreamento de Agentes Humanos

3.6 Módulo de Reconhecimento Facial

Na Seção 1.3.4, foram propostos os cinco passos para realizar o reconhecimento facial em imagens. Para implementar o módulo de reconhecimento facial que fará parte do sistema, cada passo exigiu pesquisa e testes antes que pudesse compor o sistema. Na presente seção, serão desenvolvidos a implementação e os experimentos realizados para concretizar o módulo de reconhecimento facial e validar a solução.

3.6.1 Captura da Imagem

A tarefa de reconhecimento facial é computacionalmente custosa e exige que a unidade de processamento realize uma série de operações para chegar ao resultado. Isso é um ponto importante quando se trata de aplicações que devem funcionar em tempo real, já que o processador pode não conseguir terminar um *frame* da câmera antes que outro *frame* chegue para ser processado. Por isso é interessante reduzir a quantidade de imagens que devem ser analisadas.

Como nesta solução o módulo de reconhecimento facial está atrelado ao módulo de detecção de *skeleton*, não faz sentido que ele seja executado o tempo inteiro. Logo, serão tratadas apenas

as imagens vindas da câmera que identificar um agente que ainda não foi reconhecido. Isso reduz a necessidade de processar frames quando não há ninguém em cena ou quando as pessoas em cena já foram reconhecidas.

3.6.2 Detecção de Faces

A detecção de rostos em imagens é tratada como um problema clássico em visão computacional e já é utilizada em algumas aplicações do dia a dia, como por exemplo, no foco automático de câmera [41], que está presente em grande parte dos celulares.

A pesquisa realizada nesse trabalho não tinha o objetivo de desenvolver um algoritmo de detecção facial, mas sim, de testar e avaliar diferentes abordagens e bibliotecas *open source* que já realizam essa tarefa e escolher a solução com o melhor custo benefício em qualidade e tempo de processamento. Ao final, quatro algoritmos foram escolhidos para realizar os experimentos de detecção, são eles: o detector de faces da biblioteca *cvlib* [42], os detectores *HOG* e *CNN* da biblioteca *dlib* [43] e o detector *open source Ultra-Light-Fast-Generic-Face-Detector* [44].

3.6.2.1 Algoritmos

- ***Cvlib Face Detector***

  A biblioteca *cvlib* foi construída visando facilitar a implementação de aplicações de visão computacional e possui diversos modelos pré-treinados de inteligência artificial para soluções de problemas comuns em visão, com suporte a processamento em GPU.

  O detector de faces da biblioteca *cvlib* recebe uma imagem como argumento e retorna as *bounding boxes* com os rostos encontrados e as correspondentes confianças para cada rosto.

- ***Dlib Face Detector***

  A biblioteca *dlib* é uma ótima solução para quem procura resolver problemas de aprendizagem de máquina em *C++*, e também possui uma *API* em *python*. Para detecção de rostos, essa biblioteca possui dois modelos: o *HOG* e o *CNN*.

  O modelo *HOG* utiliza histogramas de gradientes orientados como extrator de *features* para treinar um modelo máquina de vetores de suporte (*SVM*) que consegue detectar rostos em imagens. Este modelo é leve computacionalmente, porém, a maneira como foi implementado não suporta processamento em multicores de CPU nem em GPU. Ele é um bom detector de rostos frontais, contudo não funciona bem para rostos não frontais, pequenos ou parcialmente ocultos.

  O modelo *CNN* é baseado no treinamento de rede neural convolucional e consegue detectar rostos melhor que o modelo *HOG*, porém com um custo computacional mais alto. Esse problema, contudo, é amenizado pela forma como o algoritmo foi implementado, sendo capaz de utilizar uma CPU multicore ao máximo e também GPU. Logo, dependendo do hardware disponível, pode se tornar mais rápido que o modelo *HOG*.

- ***Ultra-Light-Fast-Generic-Face-Detector***

  O modelo Ultra-Light-Fast-Generic-Face-Detector é um modelo pré-treinado para detecção de rostos que foi criado com o objetivo de ser leve, capaz de ser executado até em dispositivos móveis com baixa capacidade computacional. Ele possui o código aberto e promete ser eficiente para reconhecer rostos de diversos tamanhos, orientações e até mesmo com oclusão parcial. O modelo é disponibilizado no formato *.pth* (natural do *PyTorch*) [2] e possui suporte a CPU multicore e GPU.

3.6.2.2 Validação Experimental

Para avaliar o desempenho dos algoritmos na detecção de rostos, foi utilizado o dataset *WIDER FACE* [45]. Este dataset possui 32203 imagens com pessoas em várias situações, separadas em treinamento, validação e teste. Também possui anotações bem completas que indicam a *bounding box* de cada rosto e ainda detalham se o rosto está borrado, possui alguma expressão muito forte, está mal iluminado, com oclusão ou com uma pose atípica.

Os algoritmos foram avaliados na execução do dataset de validação, que possui 3226 imagens. Para cada um, é calculado o *mean Average Precision score (mAP)* das detecções [46]. Essa métrica utiliza a medida de interseção sobre união das *bounding box* para plotar um gráfico de precisão por revocação. Em seguida, é calculada a área abaixo desse gráfico. Essa área resulta em um valor entre 0 e 1, quanto maior o valor, maior é o *mAP score* e melhor é o algoritmo.

Os resultados dos teste de detectores de faces estão detalhados na Seção 4.5.1.

3.6.3 Extração de *features*

Após obter as *bounding boxes* contendo as faces, é necessário realizar a extração de *features* da subimagem gerada. Isso será feito usando a biblioteca *dlib*, que possui uma função que recebe a imagem e as localizações das *bounding boxes* e retorna um *encoding* da face especificada.

O *encoding* da face é um *array* de dimensão 128 que representa essa face e é obtido através de um modelo de rede neural pré-treinado. Essa rede possui como entrada a imagem e a *bounding box*, e como saída, um vetor que representa as *features* do rosto contido na *bounding box*. Essas informações podem significar a distância entre os olhos, o formato da boca, o tamanho do nariz ou talvez uma combinação de todas essas informações.

3.6.4 Correspondência de *Features*

Para que uma pessoa possa ser reconhecida, o *encoding* da sua face deve ser comparado com os *encodings* da base de dados de rostos conhecidos. O resultado dessa comparação é uma distância entre os elementos do *array* de 128 posições.

[2] PyTorch é uma biblioteca de aprendizado de máquina de código aberto usada para aplicações em visão computacional e processamento de linguagem natural, desenvolvida principalmente pelo laboratório AI Research do Facebook.

Logo, antes do sistema ser colocado em prática, é necessário que as pessoas que terão acesso ao laboratório cedam algumas imagens. Todas as imagens terão seus *encodings* extraídos e será criado um arquivo com *encodings* mapeados para cada pessoa.

3.6.5 Identificação

Quando o sistema estiver funcionando, ao capturar uma nova imagem e detectar os rostos existentes, cada rosto terá seu *encoding* extraído e comparado com todos os *encodings* do banco de dados. Se a distância entre eles for menor que um determinado limiar, o rosto é identificado com o nome da pessoa que possui o *encoding* semelhante.

Para evitar falsos reconhecimentos, é aconselhável que cada pessoa cadastrada possua várias imagens para comparação. Dessa forma, é possível estabelecer um limiar mínimo de correspondências para aumentar a confiança do sistema.

3.6.5.1 Validação Experimental

Para verificar a qualidade do algoritmo de reconhecimento facial, foi utilizado um *subset* do dataset *Celebrity-Face-Recognition-Dataset* [47], formado por 1035 imagens, contendo 20 pessoas famosas e um grupo de pessoas desconhecidas.

Para realizar o experimento, 30% das imagens de pessoas famosas foram reservadas para compor o registro de pessoas conhecidas. O resto das imagens de pessoas famosas e desconhecidas foi testado, realizando detecção pelo método *Ultra-Light-Fast-Generic-Face-Detector*, extração de *features* por *encodings*, comparação e cálculo de distância com *encodings* cadastrados e realização de identificação.

Os resultados do experimento podem ser verificados na Seção 4.5.2.

3.7 Integração em Referencial Global das Detecções de Agentes Humanos

Até o momento, foi apresentado como rastrear agentes humanos usando câmeras *Kinect* e como reconhecer rostos em imagens, por meio de algoritmos de inteligência artificial. Porém, na atual solução, as estimativas de pose ainda são dadas em relação a câmera que realiza a detecção e os usuários ainda são identificados apenas por um número, que pode ser diferente para um mesmo usuário visto por diversas câmeras.

O módulo que irá unir as informações em referencial global e aplicar o reconhecimento facial nos agentes detectados é chamado de *Global_Human_Tracker*. Ele irá receber informações do tópico de estimação de pose das juntas e, por meio das poses conhecidas das câmeras, obtidas pelo mapeamento, fazer a transformação de coordenadas do sistema da câmera para o sistema global, aplicando o resultado em um EKF para cada junta.

Para entender como este módulo funciona, as seguintes etapas foram propostas:

- **Medição**

  Na etapa de medição, o módulo escuta o tópico *ROS* que publica as estimativas de pose de cada junta. Ao receber uma nova estimativa, são extraídas as posições x, y e z da junta, a orientação x, y, z e w em quatérnios, o identificador da câmera que realizou a detecção, o nome da junta e de qual agente é essa junta, por meio do número identificador.

- **Inicialização**

  O módulo possui dois registros de agentes, um para os agentes que já foram reconhecidos e um para os agentes ainda desconhecidos. Esses registros têm a mesma estrutura e servem para realizar o casamento de informações das múltiplas câmeras.

  Um exemplo de estrutura de registro pode ser dado pelo código

```
{"camera1":{{"1":"Maria"},
            {"2":"Joao"},
            {"3":"Jose"}},
 "camera2":{{"1":"Joao"},
            {"2":"Maria"}},
 "camera3":{{"1":"Jose"}}}
```

  em que existem três câmeras e três agentes. Os agentes nem sempre são visualizados por todas as câmeras e podem ter números identificadores diferentes de uma câmera para a outra. Esse registro irá auxiliar a concentrar as medições de um único agente em apenas um filtro.

  Quando os agentes ainda não foram reconhecidos, são inseridos em um registro temporário, em que é atribuído um nome que contém as informações de câmera e usuário.

- **Reconhecimento Facial**

  Quando uma medição chega no algoritmo, é verificado se o identificador do agente dessa câmera está em algum dos registros de agentes. Se não estiver em nenhum registro ou se estiver no registro temporário, é requisitado o reconhecimento facial para o módulo de reconhecimento, por meio de uma mensagem *ROS*.

  O módulo de reconhecimento facial recebe a requisição que indica qual câmera precisa ser capturada, qual agente precisa ser identificado e qual é a posição da cabeça desse agente no sistema de coordenadas da câmera.

  É realizado o reconhecimento facial para essa câmera, em que são retornadas uma lista de *bounding boxes* e uma lista correspondente de nomes das pessoas identificadas em cada *bounding box*.

  Por fim, a informação de posição da cabeça é mapeada para coordenadas da imagem por meio da matriz de parâmetros intrínsecos da câmera e é verificado se esta coordenada está contida em alguma das *bounding boxes* da lista resultante. Se estiver, quer dizer que o agente foi reconhecido.

O módulo de reconhecimento facial retorna uma mensagem indicando a câmera, o agente, o resultado do reconhecimento e o nome do agente, caso tenha sido bem sucedido. Os resultados de reconhecimento possíveis são: pessoa reconhecida, pessoa desconhecida (quando o rosto foi detectado, mas não identificado) e pessoa não detectada (quando não foi possível detectar o rosto).

- **Filtragem das Detecções**

  Cada agente presente no sistema, tanto os identificados quanto os não identificados, tem as suas medições passadas para o sistema global através de um EKF. Cada junta possui o seu próprio filtro, que foi implementado usando a mesma modelagem usada para os marcadores, proposta na Seção 3.4.2, com um modelo de velocidade constante para posição e velocidade angular nula para rotação. Quando um mesmo agente é reconhecido em mais de uma câmera, as medições resultantes são sempre atualizadas em apenas um filtro.

- **Limpeza**

  Por fim, uma *thread* paralela fica supervisionando todos os usuários identificados no sistema verificando o tempo da última medição realizada. Caso esse tempo ultrapasse um certo limiar, o agente é removido dos registros e os filtros de suas juntas são excluídos. Isso garante que futuramente um agente que receba o mesmo identificador passe por todas as etapas corretamente.

3.7.1 Validação Experimental

Para verificar o funcionamento do algoritmo, foram propostos dois experimentos. O primeiro tem o objetivo de mostrar o resultado do rastreamento por filtragem bayesiana e o segundo tem o objetivo de mostrar o resultado do reconhecimento. Para os dois, o sistema de coordenadas global foi manualmente colocado no mesmo sistema de coordenadas do *Kinect*, já que só foi possível utilizar um *Kinect* nesse experimento.

3.7.2 Experimento de Verificação do Rastreamento

No experimento de verificação do rastreamento, o *Kinect* foi posicionado a uma altura de aproximadamente 95 cm e foram colocados dois marcadores (cruzes azuis) no chão distantes 370 cm e 80 cm do *Kinect*, destacado com uma *bounding box* verde na Figura 3.10. Uma pessoa realizou o caminho andando de um marcador para o outro, como indicado pela seta vermelha.

Os dados de estimação de pose foram coletados e os resultados deste experimento podem ser conferidos na Seção 4.6.1.

Figura 3.10: Ambiente do Experimento

3.7.3 Experimento de Verificação do Reconhecimento

O experimento de verificação de reconhecimento tem o objetivo de validar o comportamento do algoritmo em identificar corretamente os *skeletons* de pessoas com o seu nome, a partir de reconhecimento facial. Para isso, contou com a participação de quatro pessoas, três delas cadastradas no sistema e uma desconhecida.

Os resultados do experimento podem ser vistos em detalhes na Seção 4.6.2.

3.8 *API* de Localização de Agentes Móveis e Humanos

Todos os módulos tratados nesse projeto resultaram em uma *API* com 8 classes em *python*, são elas:

```
SimCamera(camera_name)

KinectCamera(camera_name, device_id, rgb_camera_info_url, depth_camera_info_url)

MapCamera(cameras, markers, located_markers, logfilename)

ArucoDetect(camera_name, image_name, fiducial_len, dictionary)

GlobalMarkerTracker(conf_file)

KinectTracker(camera_name, device_id)

GlobalHumanTracker(conf_file)

FaceRecognitionModule(cameras, device_ids, model_path, register_path)
```

A classe *SimCamera* inicia captura de uma câmera simulada no *CoppeliaSim* e deve ser executada junto com a simulação. A classe *KinectCamera* inicia a captura de imagem de uma câmera *Kinect*. A classe *MapCamera* recebe a lista de câmeras do sistema e realiza o mapeamento no ambiente, fazendo os passos explicados na Seção 3.3. A classe *ArucoDetect* inicia a detecção de marcadores em relação ao sistema de coordenadas da câmera, como foi explicado na Seção 3.2. A classe *GlobalMarkerTracker* faz a transformação de coordenadas dos marcadores do sistema das câmeras para o sistema global realizando a filtragem, como mostrado na Seção 3.4. A classe *KinectTracker* inicia o módulo de detecção de *skeleton* relatado na Seção 3.5. A classe *GlobalHumanTracker* realiza a transformação para o sistema global das detecções de juntas dos *skeletons*, aplicando o reconhecimento facial, como foi explicado na Seção 3.7. E por fim, a classe *FaceRecognitionModule* responde as requisições de reconhecimento facial aplicando os algoritmos testados na Seção 3.6.

Além dessas classes, foram escritos outros três *scripts* auxiliares, um para capturar e armazenar fotos das pessoas cadastradas, um para gerar o arquivo de registro de pessoas cadastradas com os *encodings* das faces e um para mostrar o resultado da detecção dos *skeletons* no ambiente *rviz* [48].

Exemplos de utilização da *API* foram apresentados no Anexo I.

Capítulo 4

Resultados

4.1 Detecção de Marcadores *ARUCO*

O experimento descrito na Seção 3.2 foi realizado usando uma câmera *Kinect* real com um marcador impresso e colado em uma superfície plana, para não gerar alterações causadas por dobra ou ondulações no papel (Figura 4.1).

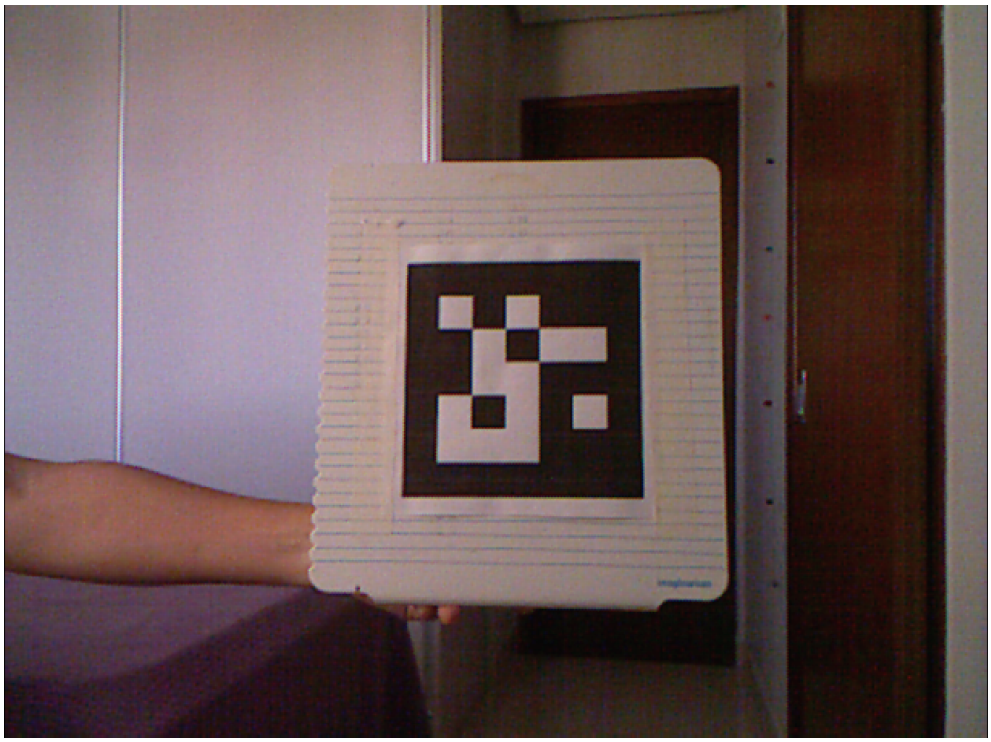


Figura 4.1: Preparação do Experimento de Detecção de Marcadores

O marcador foi movimentado seguindo a trajetória proposta na Figura 3.5 e foram obtidos os dados de posição 3D e orientação do marcador em relação a câmera, mostrados nos gráficos da Figuras 4.2 e 4.3, respectivamente.

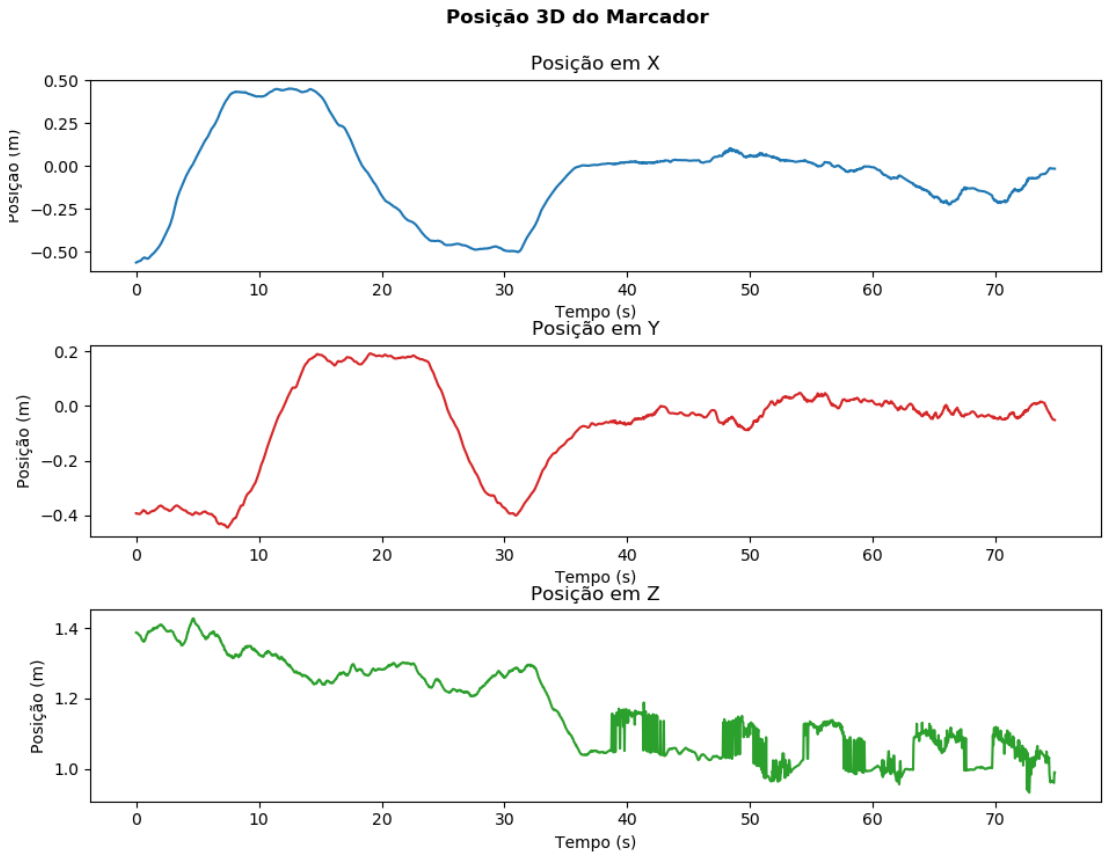


Figura 4.2: Posição 3D do Marcador

No gráfico da Figura 4.2 é possível perceber que do instante $0s$ até aproximadamente o instante $30s$, houve variação nas coordenadas x e y, indicando a realização do movimento retangular. Já no gráfico da Figura 4.3, percebe-se, pela variação nas coordenadas x e y do quatérnio, depois do instante $30s$, que houve uma rotação do marcador, indicando que os resultados observados estão de acordo com o movimento realizado.

O gráfico de posição da Figura 4.2 mostra que o detector acompanhou bem a as coordenadas x e y do marcador, porém, durante o movimento de rotação, houve uma certa instabilidade na coordenada z. Essa instabilidade deverá ser tratada pela filtragem bayesiana do EKF.

O gráfico de componentes do quatérnio de rotação da Figura 4.3 parece muito instável, com variações repentinas de valor, especialmente nas coordenadas x e y. Contudo, esses valores estão corretos e são consequência da representação de orientação por quatérnios varrer o espaço 3D duas vezes. A orientação representada pelo quatérnio q é a mesma representada pelo quatérnio $-q$, logo, a instabilidade observada é provocada por essa mudança de sinal, que, apesar de não estar errada, pode afetar a filtragem. Por isso, a medição realizada pelo detector de marcadores deverá ser tratada antes de ser dada como entrada para o EKF.

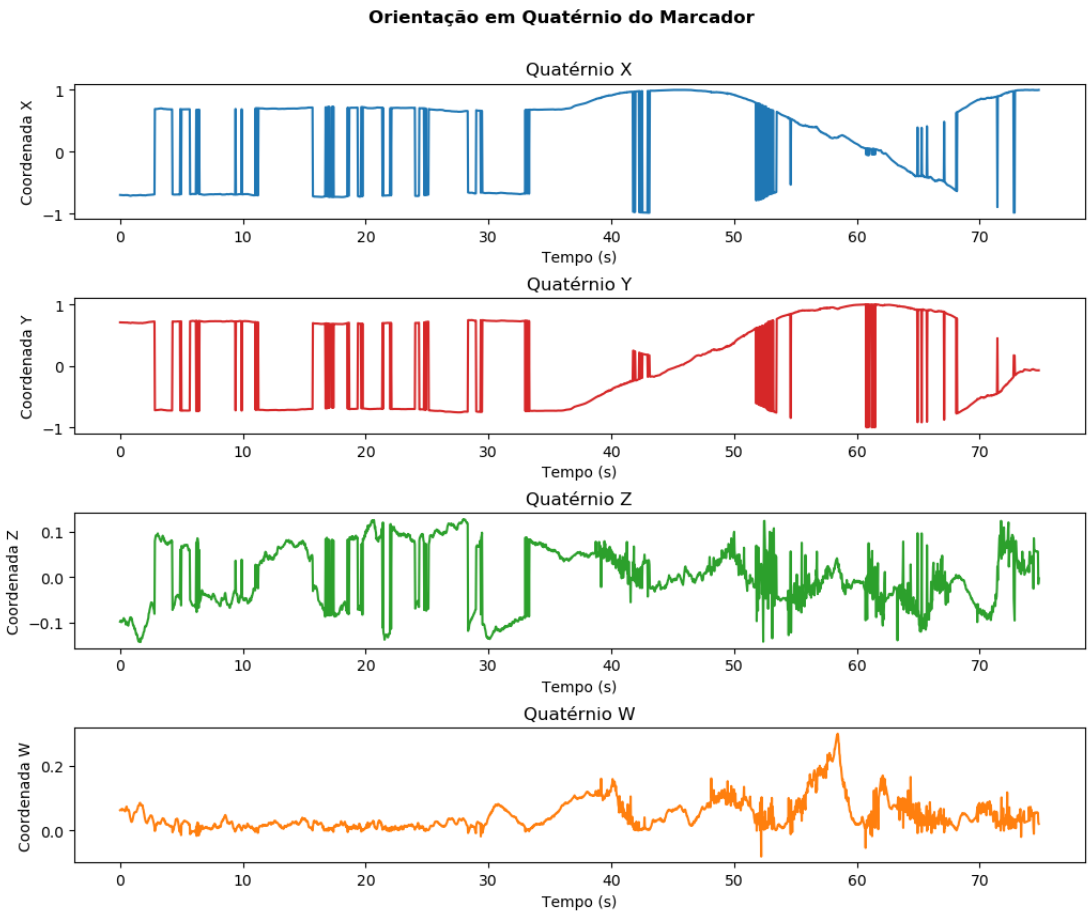


Figura 4.3: Orientação do Marcador em Quatérnios

4.2 Mapeamento

O experimento descrito na Seção 3.3 coloca uma cadeia de câmeras para testar o erro de estimação de pose do algoritmo de mapeamento. A execução do experimento gerou como saída dados de posição 3D e orientação em quatérnios da localização real e da localização estimada de cada câmera.

Para avaliar o erro de posição da estimação, foi utilizada a distância euclidiana entre os pontos no espaço, dada por

$$erro_{pos} = \sqrt{(p_{real} - p_{est})^2}. \tag{4.1}$$

Já para avaliar o erro de orientação dos quatérnios estimados, foi utilizada a distância geodésica intrínseca, que corresponde ao comprimento do arco que liga um quatérnio q_1 a um quatérnio q_2, desenvolvida em [49].

Para posição e orientação, foram plotados os gráficos das Figuras 4.4 e 4.5, respectivamente, que mostram o comportamento do erro de estimação à medida que a câmera se distancia do marcador inicial, de pose conhecida.

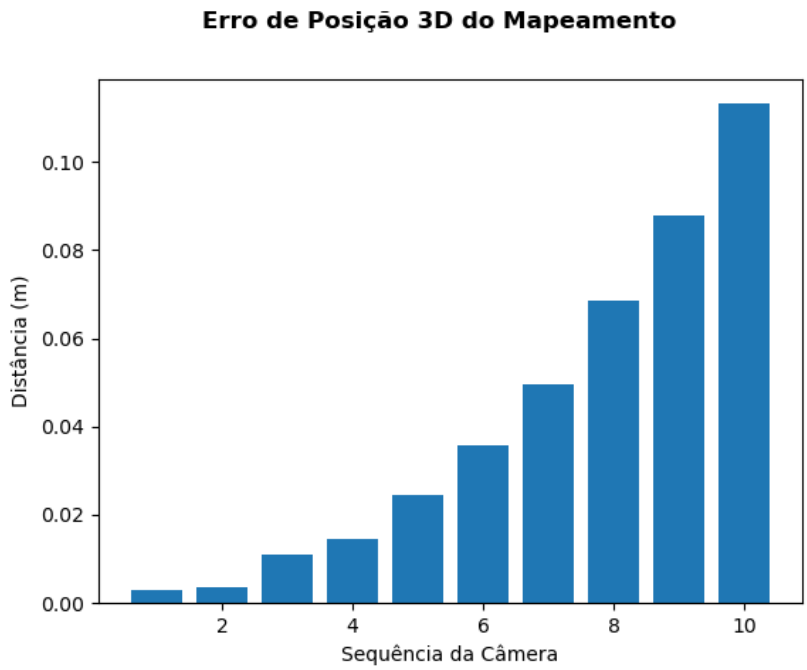


Figura 4.4: Erro de Posição do Mapeamento

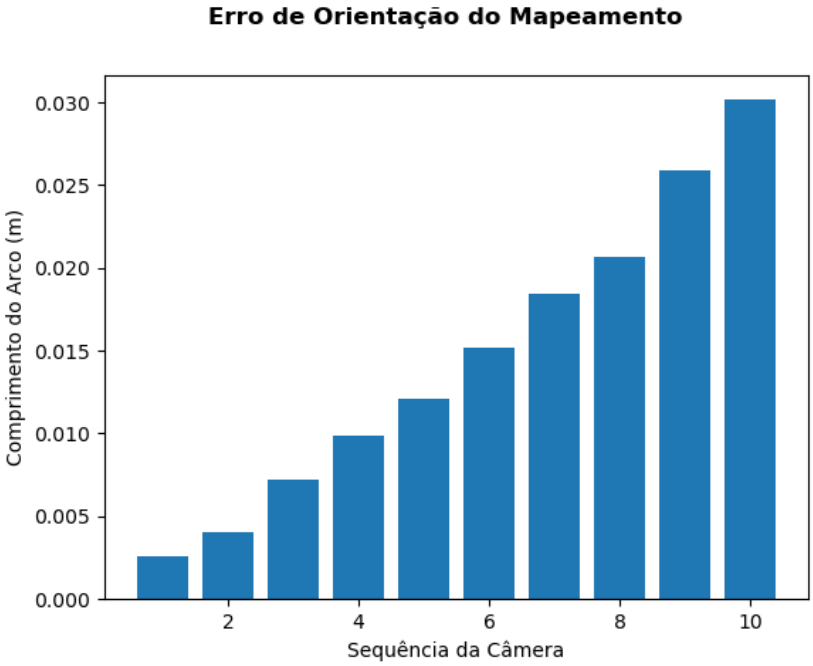


Figura 4.5: Erro de Orientação do Mapeamento

É possível visualizar que tanto o erro de posição quanto o de orientação crescem à medida que mais câmeras são encadeadas, o que já era esperado. Por isso é necessário analisar qual aplicação o sistema irá atender e quais são os requisitos de precisão e exatidão necessários, para poder alterar as condições de entrada e reduzir esse efeito. Uma maneira de reduzir o erro de estimação de pose no mapeamento é fornecer mais posições conhecidas para marcadores, evitando formar cadeias

muito grandes de câmeras.

Aqui é importante frisar que o erro de estimativa observado neste experimento não depende apenas do número de câmeras e não deve se repetir em uma conformação diferente da realizada. Isso porque o erro depende também do erro na detecção dos marcadores, que é altamente influenciado por condições como: iluminação, nitidez da câmera, tamanho do marcador, perpendicularidade do marcador em relação à câmera e etc.

4.3 Integração em Referencial Global das Detecções dos Marcadores

O algoritmo de integração em referencial global das detecções de marcadores utiliza um Filtro de Kalman Estendido para fazer a fusão sensorial das estimativas de pose geradas por múltiplas câmeras. Para verificar a performance desse algoritmo, foi montada a cena da Figura 3.8 e planejados os seguintes experimentos:

4.3.1 Seguimento de Trajetória com Velocidade Constante

No primeiro experimento, foi desenhada uma trajetória pré-definida no simulador que é percorrida pelo marcador em velocidade constante $v = 0.5m/s$. Essa velocidade é constante em módulo, porém varia a direção para seguir a trajetória. O marcador não foi rotacionado durante o movimento.

O simulador é capaz de fornecer os dados de pose 3D real do marcador e a saída do algoritmo fornece a pose 3D estimada. Foi plotado o gráfico de trajetória no plano XY (Figura 4.6) e o gráfico de posição 3D para cada componente (Figura 4.7).

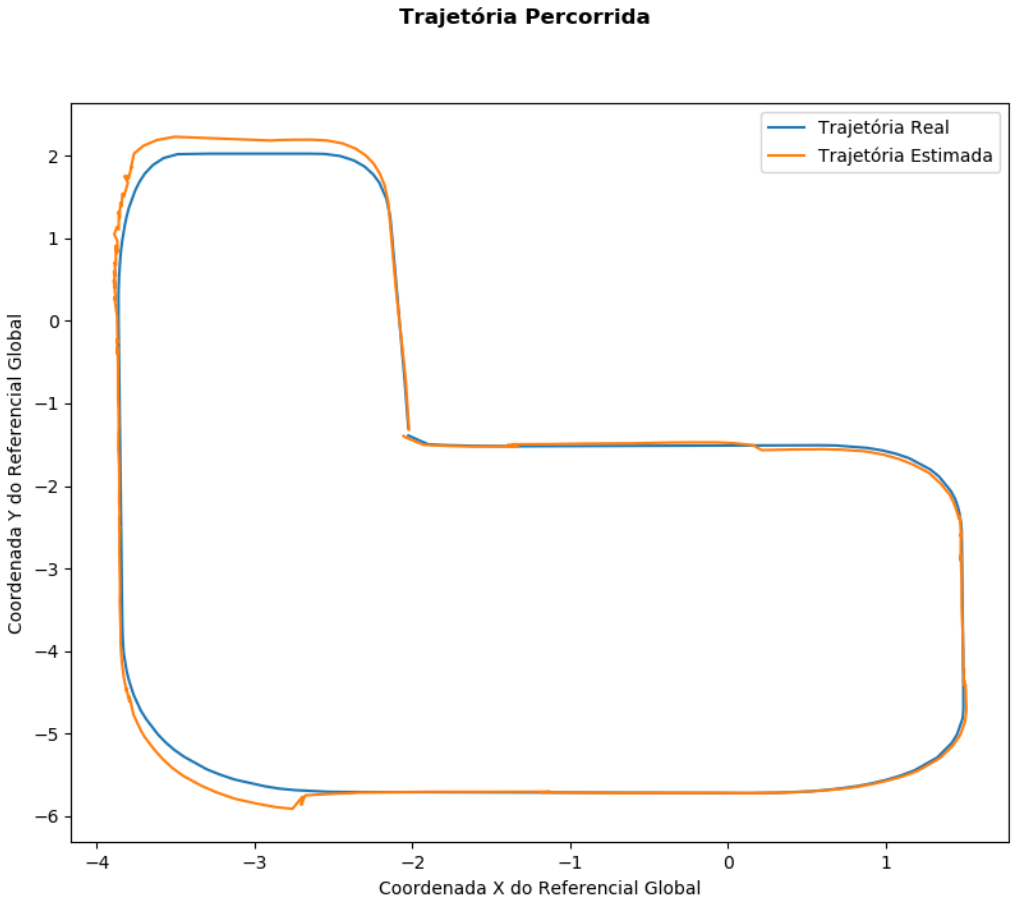


Figura 4.6: Trajetória Realizada para Velocidade Constante

Analisando o gráfico da Figura 4.6, é possível perceber que o EKF conseguiu acompanhar a referência de posição e seguir a trajetória de forma satisfatória. Porém, em algumas regiões é possível perceber um leve desvio, como no canto inferior e superior esquerdo do gráfico. Este erro pode ter sido gerado por um mal mapeamento de câmera na região. O gráfico da Figura 4.7 mostra que o filtro seguiu bem a referência, não gerou grandes saltos de estimativa ao passar de uma câmera para outra e não apresentou atraso notável na estimativa.

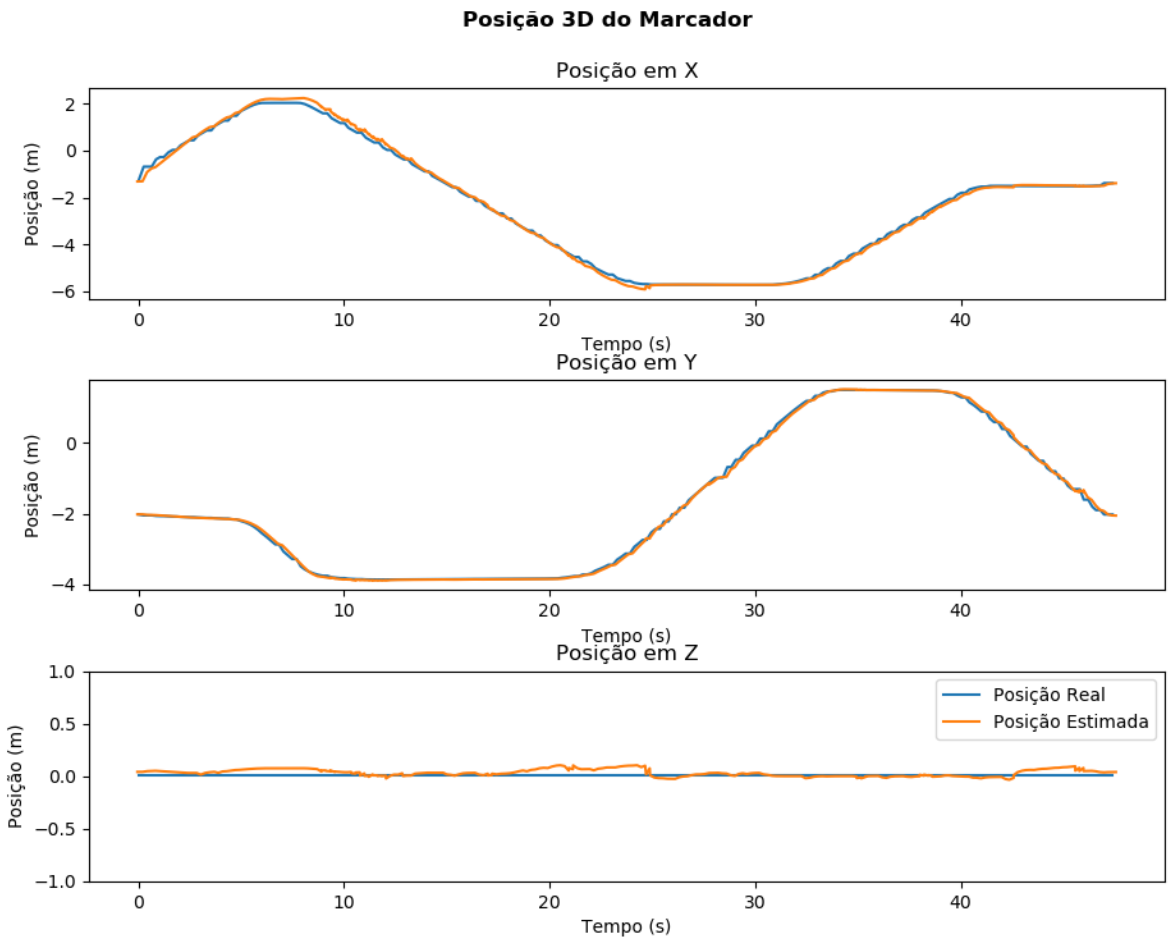


Figura 4.7: Rastreamento de posição 3D para Velocidade Constante

4.3.2 Seguimento de Trajetória com Aceleração Constante

O segundo experimento foi projetado para verificar o comportamento do EKF ao tratar uma situação para qual não foi modelado. Como explicado na Seção 3.4.2, o sistema considera que o movimento tem aceleração linear nula e velocidade angular nula. Para isso, foi utilizada a mesma trajetória do experimento anterior, porém o marcador possui uma velocidade inicial $v = 0.5m/s$ e aceleração constante $a = 0.2m/s$. Novamente, o marcador não foi rotacionado durante o movimento.

Foram armazenados os dados de pose 3D real e pose 3D estimada. Para visualizar os dados, foram plotados os gráficos de trajetória no plano XY (Figura 4.8) e de posição 3D (Figura 4.9).

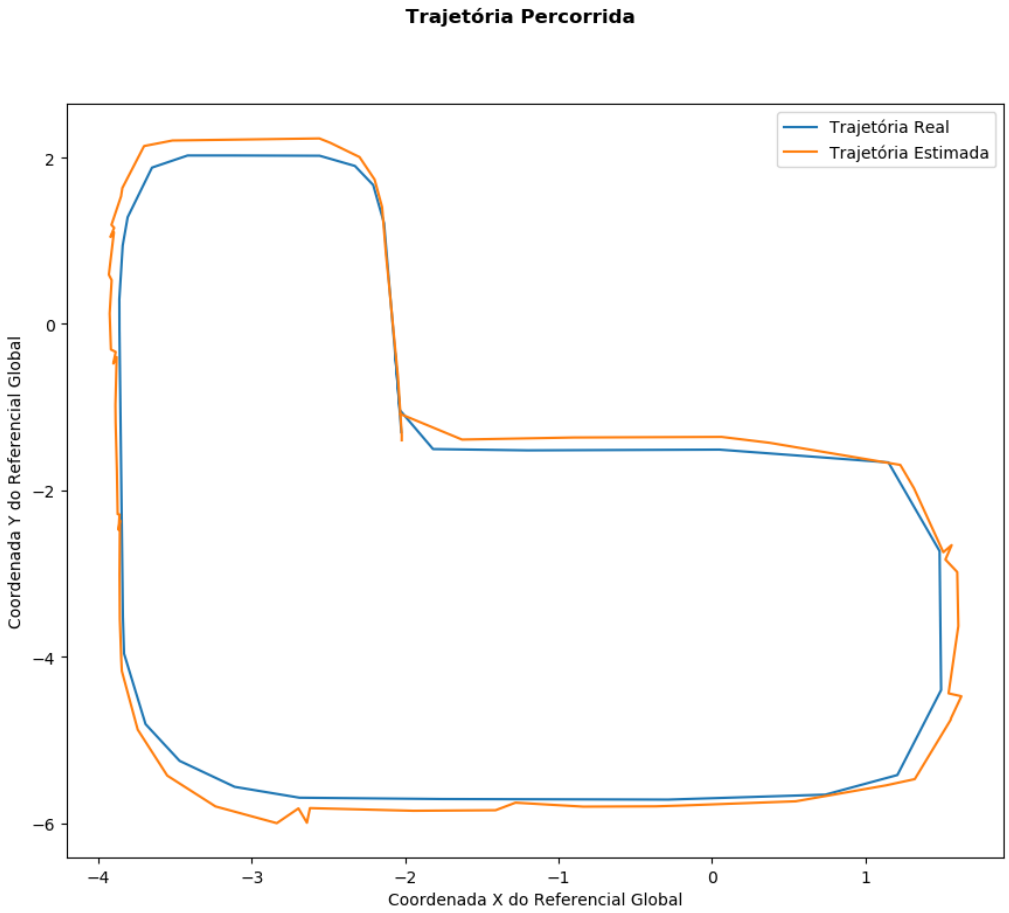


Figura 4.8: Trajetória Realizada para Aceleração Constante

Comparando o gráfico da Figura 4.6 e o gráfico da Figura 4.8 percebe-se um comportamento inferior no segundo, isso se deve ao fato de o sistema não estar preparado para esse tipo de movimento. Percebe-se que novamente ocorreram desvios na trajetória nos cantos superior e inferior esquerdo e na lateral direita.

O gráfico da Figura 4.9 mostra que o EKF atenuou a curva medida, porém com um certo atraso, causado pela aceleração. Apesar do resultado inferior, é possível concluir que o comportamento foi satisfatório e uma remodelagem do filtro não é necessária, dependendo da aplicação.

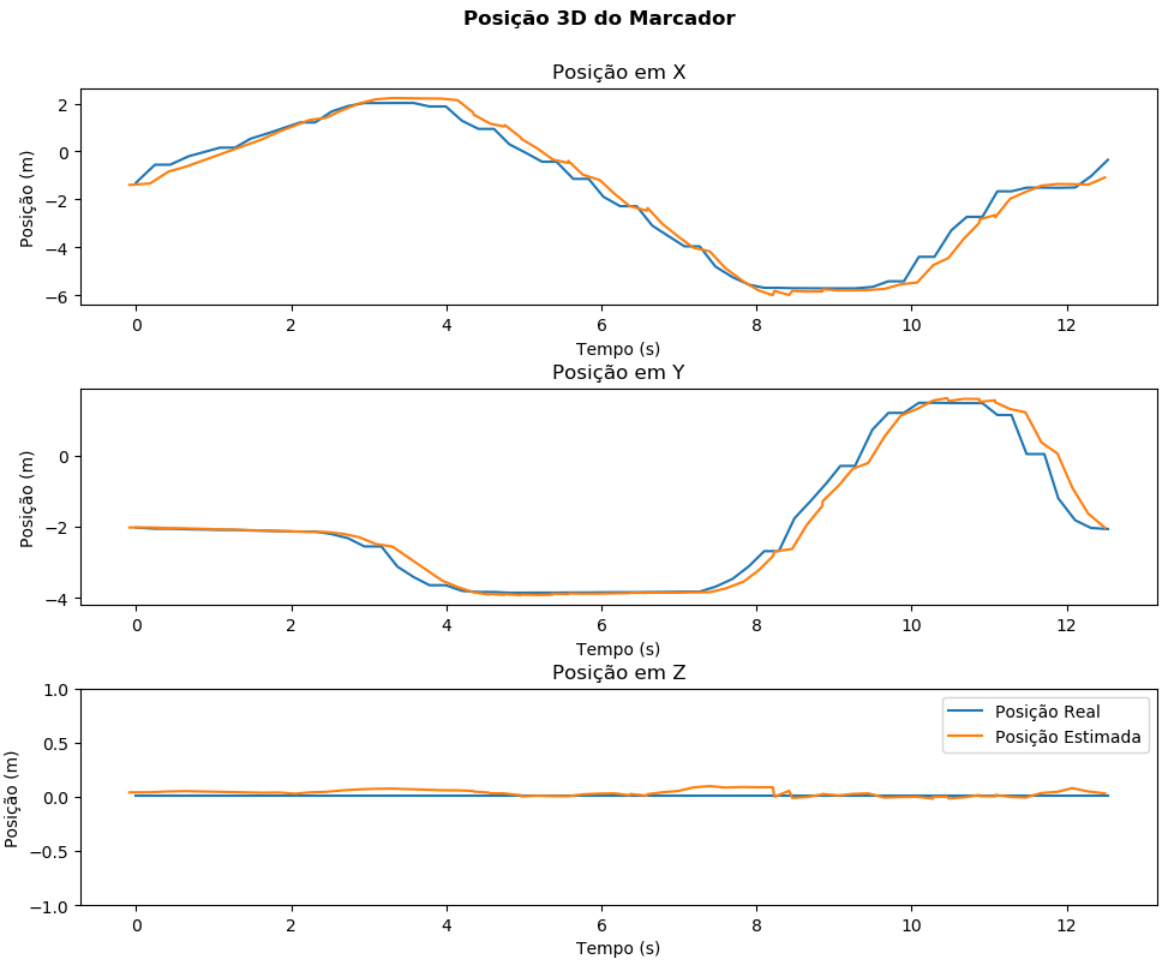


Figura 4.9: Rastreamento de posição 3D para Aceleração Constante

4.3.3 Movimento com Comportamento Não Definido

O terceiro experimento, tem o objetivo de simular uma situação mais próxima da realidade, com um movimento que não segue uma trajetória pré-definida e pode acelerar, desacelerar e mudar rapidamente de direção. Para isso foi utilizada a ferramenta de movimentação manual por *mouse* do *CoppeliaSim*. O marcador foi movimentado de forma aleatória variando a velocidade, tanto linear quanto angular, de forma não uniforme. Para visualizar os dados, foram plotados os gráficos de trajetória no plano XY (Figura 4.10) e posição 3D (Figura 4.11).

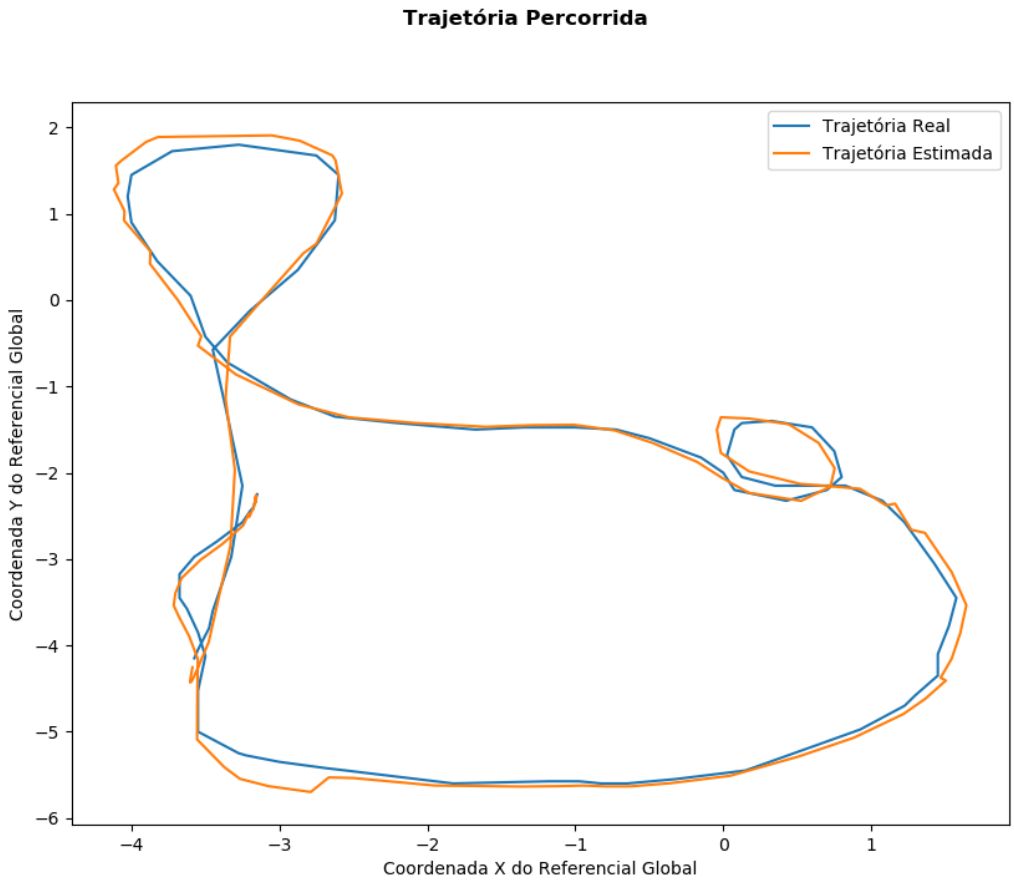


Figura 4.10: Trajetória Realizada para Movimento Aleatório

O gráfico da Figura 4.10 mostra que o filtro conseguiu seguir a referência e não apresentou grandes discrepâncias em relação a trajetória real. Após os três experimentos, verificou-se que os desvios de trajetória aconteceram aproximadamente no mesmo local, o que indica um mal mapeamento das câmeras dessas regiões. Não foram observados saltos bruscos de posição quando o marcador passa da visão de uma câmera para a outra, o que indica que o filtro foi bem sucedido na sua principal tarefa, que é a fusão sensorial das medições realizadas pelas múltiplas câmeras.

O gráfico da Figura 4.11 mostra o comportamento de EKF nas três componentes de posição, que acompanha bem a referência com um pequeno atraso, gerado pela variação da velocidade, não prevista pelo filtro.

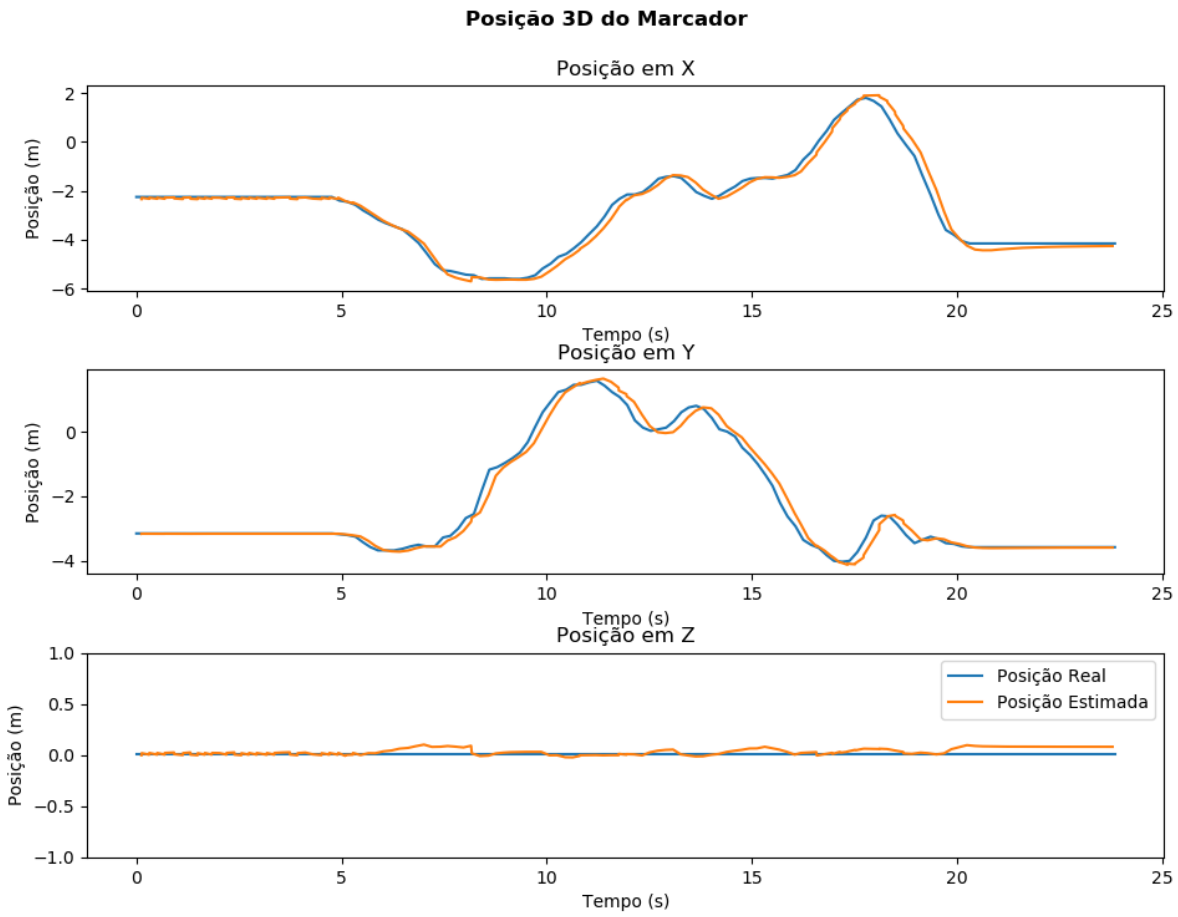


Figura 4.11: Rastreamento de posição 3D para Movimento Aleatório

4.4 Rastreamento de Agentes Humanos

Para validar a solução do módulo de rastreamento, foi proposto um simples experimento em que uma pessoa é posicionada em frente à câmera e realiza um movimento com os braços, como indicado na Figura 3.9.

O pacote *openni_tracker* verifica se há algum usuário em frente à câmera e, caso exista, publica em um tópico *ROS* as seguintes informações:

- *stamp*: Tempo *ROS* em milissegundos.
- *frame id*: Nome do sistema de coordenadas em que a pose foi medida.
- *child frame id*: Nome do sistema de coordenadas do objeto medido.
- *translation*: Posição 3D do *child frame id* em relação ao *frame id* em metros.
- *rotation*: Orientação do *child frame id* em relação ao *frame id* em quatérnios.

No experimento realizado, o movimento acontece no plano XY da câmera, por isso, foram coletadas as posições x e y de cada junta e plotado o gráfico da Figura 4.12, que mostra a trajetória de cada junta no plano XY.

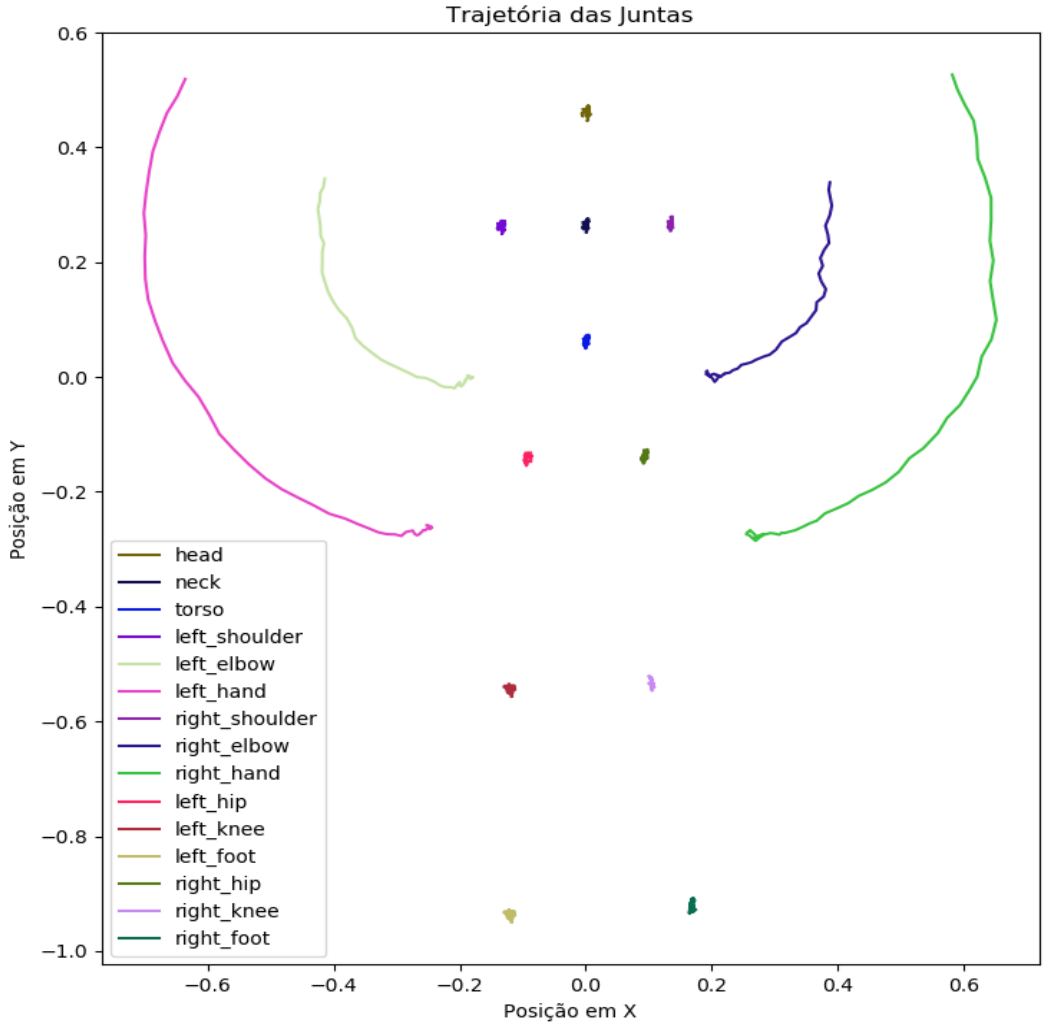


Figura 4.12: Trajetória das Juntas no plano X vs Y

O gráfico da Figura 4.12 mostra a trajetória percorrida pelas juntas das mãos e dos cotovelos, enquanto as outras juntas permanecem na mesma posição. Essa trajetória condiz com o movimento realizado, validando o modelo de rastreamento de agentes humanos pelo pacote *ROS openni_tracker*.

4.5 Módulo de Reconhecimento Facial

4.5.1 Detecção de Rostos

O experimento de detecção de rostos em imagens avaliou o desempenho dos algoritmos de detecção no *dataset* de validação *WIDER FACE*. Como esse *dataset* possui várias imagens em que os rostos não estão em condições ideais, foram criadas duas categorias de *ground truth*[1]. O primeiro leva em conta apenas os rostos não borrados, em posição típica, sem oclusão, sem expressões fortes e com boa iluminação, chamado de *dataset* fácil. O segundo leva em conta todas as anotações, chamado dataset difícil. Para cada algoritmo foi avaliado o *mAP score* para o *ground truth* fácil e difícil, assim como o tempo de execução total, para o *dataset*, e médio, por imagem.

A máquina que realizou o experimento possui uma placa-mãe Asus Prime B450M Gaming/BR, um processador AMD Ryzen 7 3700x com 8 núcleos e 16 *threads*, uma placa de vídeo NVIDIA RTX 3060Ti com 8GB de VRAM, 16GB de memória RAM, 256GB de SSD e 1T de HD. Todas as bibliotecas foram instaladas de forma otimizada com os aceleradores disponíveis para esse hardware.

Os resultados para cada algoritmo foram resumidos na Tabela 4.1. Um exemplo de detecção para cada algoritmo é dado pela Figura 4.13.

Algoritmo	mAP score		Tempo de Processamento	
	Dataset fácil	*Dataset* difícil	Total	Médio
cvlib Face Detector	0.681	0.281	59.79s	18.5ms
dlib HOG Face Detector	0.411	0.072	1000.19s	310ms
dlib CNN Face Detector	0.720	0.163	456.38s	141ms
Ultra-Light-Face-Detector	0.863	0.381	38.30s	11.9ms

Tabela 4.1: Avaliação dos Algoritmos de Detecção

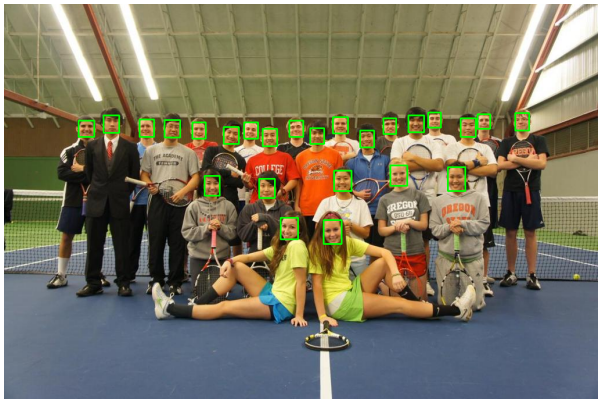
Analisando a Tabela 4.1 e a Figura 4.13 é possível concluir que o *Ultra-Light-Fast-Generic-Face-Detector* possui o melhor desempenho, tanto em tempo de processamento, quanto em *mAP score*, por isso, foi o modelo escolhido para compor o módulo de reconhecimento facial. O *cvlib Face Detector* mostrou um bom tempo de execução, porém apresentou uma qualidade mais baixa, especialmente quando os rostos são pequenos na imagem, como pode ser visto pela Figura 4.13a. O *dlib CNN Face Detector* apresentou um bom *mAP score*, porém um alto tempo de execução. Por fim, o algoritmo *HOG* da biblioteca *dlib* pode ser considerado o pior, com um alto tempo de execução e um baixo *mAP score*.

[1]Termo usado em vários campos para se referir às informações fornecidas por observação direta, em oposição às informações fornecidas por inferência.

(a) cvlib Face Detector

(b) dlib HOG Face Detector

(c) dlib CNN Face Detector



(d) Ultra-Light-Face-Detector

Figura 4.13: Algoritmos de Detecção de Faces

Vale lembrar que esses resultados foram obtidos com o *hardware* disponível, que contou com um processador *multicore* e uma placa de vídeo de alto desempenho, caso fosse necessário realizar a detecção facial em computadores *singlecore*, sem acelerador de GPU, provavelmente o modelo HOG seria mais rápido, pois exige menos operações.

4.5.2 Identificação

O experimento realizado para verificar o desempenho de todo o reconhecimento facial usou como base um *dataset* com imagens de 20 pessoas famosas e também pessoas desconhecidas. Após a execução de todo o algoritmo, foram medidas as acurácias total e por classe das inferências, e também o tempo de execução total e por imagem. Vale lembrar que esse tempo de execução inclui todo o processo, desde detecção de faces até a identificação final.

Os resultados de acurácia foram apresentados na forma de matriz de confusão pela Figura 4.14. Já o tempo de processamento pode ser verificado na Tabela 4.2.

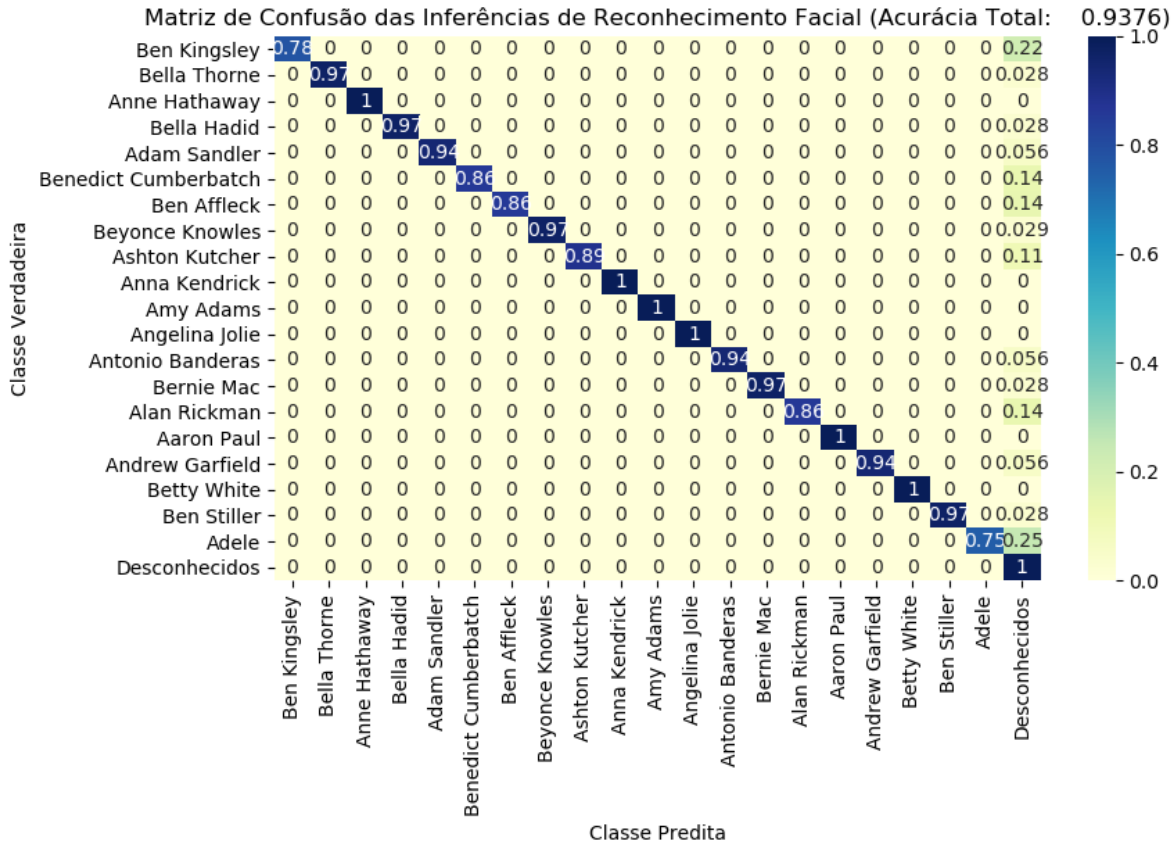


Figura 4.14: Matriz de Confusão

Tempo Total	Tempo Médio
16.84s	22.2ms

Tabela 4.2: Tempo de Execução

Por meio desse experimento, foi possível verificar que o algoritmo de reconhecimento tem um desempenho bastante satisfatório, atingindo uma acurácia total de 93,7%. Analisando a matriz de confusão da Figura 4.14 percebe-se que em algumas situações, algumas pessoas que estavam registradas no sistema não foram reconhecidas, porém nenhuma pessoa foi confundida com outra e todos os desconhecidos foram corretamente marcados como desconhecidos. Isso mostra que o sistema é bastante confiável para realizar o controle de acesso do laboratório.

O tempo de execução do algoritmo de reconhecimento também foi bastante satisfatório, permitindo a aplicação do sistema em tempo real.

4.6 Integração em Referencial Global das Detecções de Pessoas

A integração em referencial global das detecções de agentes humanos faz tanto a transformação de coordenadas do espaço da câmera para o referencial global, por meio de fusão sensorial de

múltiplas câmeras, quanto a identificação dos agentes por reconhecimento facial. Para validar a solução implementada foram propostos dois experimentos.

4.6.1 Experimento de Validação do Rastreamento

O experimento de validação do rastreamento tem o objetivo de avaliar o desempenho do Filtro de Kalman Estendido na estimação de posição. Infelizmente, não foi possível fazer a filtragem de dados de múltiplas câmeras, pois só havia um *Kinect* disponível. Por isso, o sistema de coordenadas global foi colocado no centro da câmera.

Uma pessoa caminhou entre os marcadores, que estavam posicionados a aproximadamente 3,70 m e 0,80 m da câmera, como indicado na Figura 3.10. Foram armazenados os dados de posição 3D de todas as juntas, porém, para facilitar a visualização, apenas os dados de posição 3D da cabeça foram plotados no gráfico da Figura 4.15.

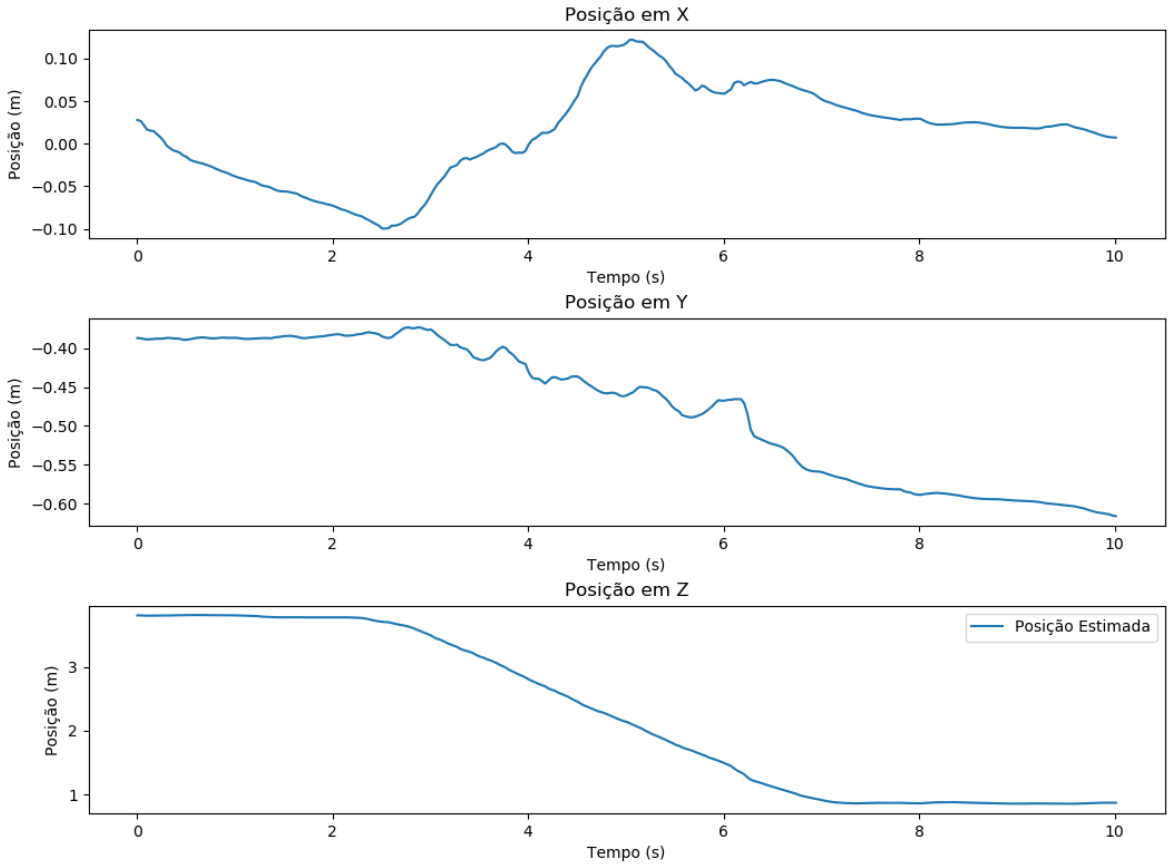


Figura 4.15: Posição 3D da Junta da Cabeça Durante o Movimento

O gráfico mostra muito bem o acompanhamento do movimento por meio da coordenada z que sai da posição inicial do primeiro marcador por volta de 3,70 m, realiza um movimento com velocidade praticamente constante até o segundo marcador, de coordenada 0,80 m. As componentes

x e y apresentaram uma variação em um intervalo pequeno de 20 cm, causado pelo movimento ao andar. Esses resultados mostram que a estimativa de posição provida pelo *Kinect* e a filtragem foram eficazes, produzindo valores próximos do esperado.

4.6.2 Experimento de Validação do Reconhecimento

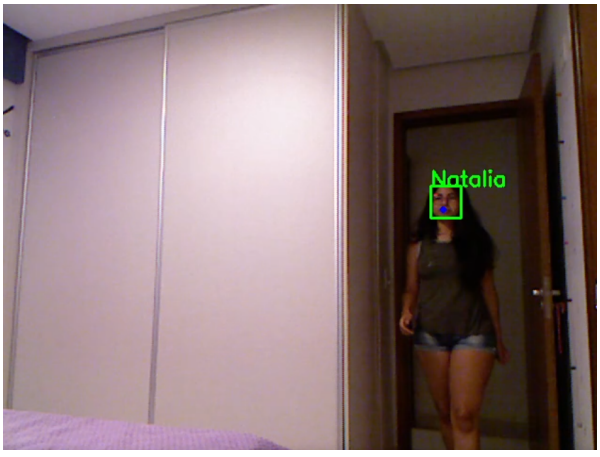
O experimento de validação do reconhecimento facial tem o objetivo de avaliar a capacidade do algoritmo em reconhecer os agentes em cena e atribuir um nome ao *skeleton* rastreado em referencial global.

Para isso, quatro pessoas foram colocadas em cena e o algoritmo foi executado. Para poder visualizar melhor, as juntas foram plotadas em espaço tridimensional utilizando o *software* de visualização *rviz*. Para cada pessoa registrada foi associada uma cor, como mostrado na Tabela 4.3

Nome	Cor
Desconhecidos	Verde
Karine	Azul
Natalia	Rosa
Renata	Amarelo

Tabela 4.3: Relação de Cores e Nomes no Reconhecimento Facial

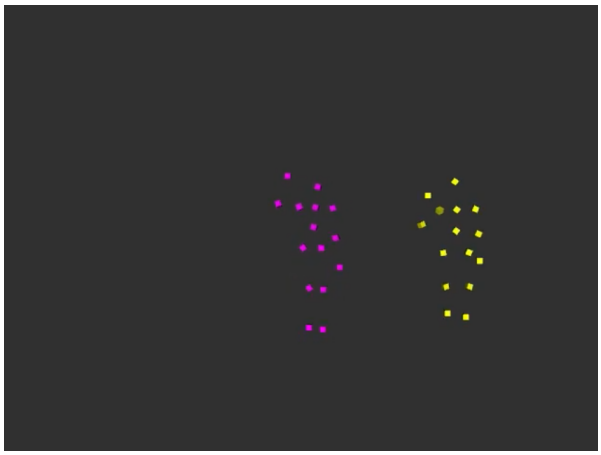
As pessoas entraram em cena de forma sequencial e o *skeleton*, quando reconhecido, mudou de cor para identificar corretamente os agentes, como pode ser visto nas Figuras 4.15. Os resultados mostram que o algoritmo foi eficiente no reconhecimento facial dos agentes envolvidos, sendo capaz de identificar corretamente as pessoas cadastradas e não identificar as pessoas desconhecidas.



(a) Imagem da Câmera

(b) Visualização do *Skeleton*

(c) Imagem da Câmera

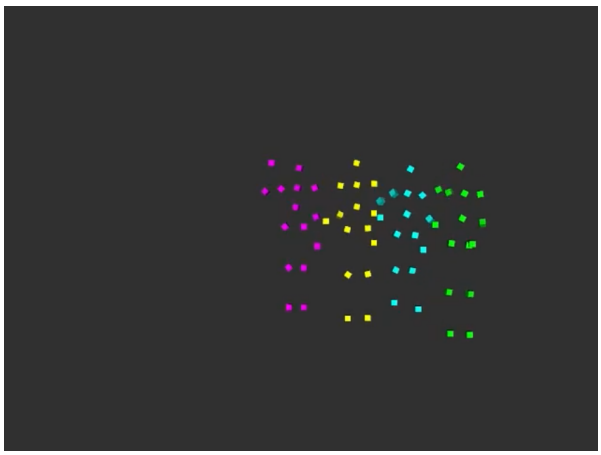


(d) Visualização do *Skeleton*

(e) Imagem da Câmera

(f) Visualização do *Skeleton*

(g) Imagem da Câmera

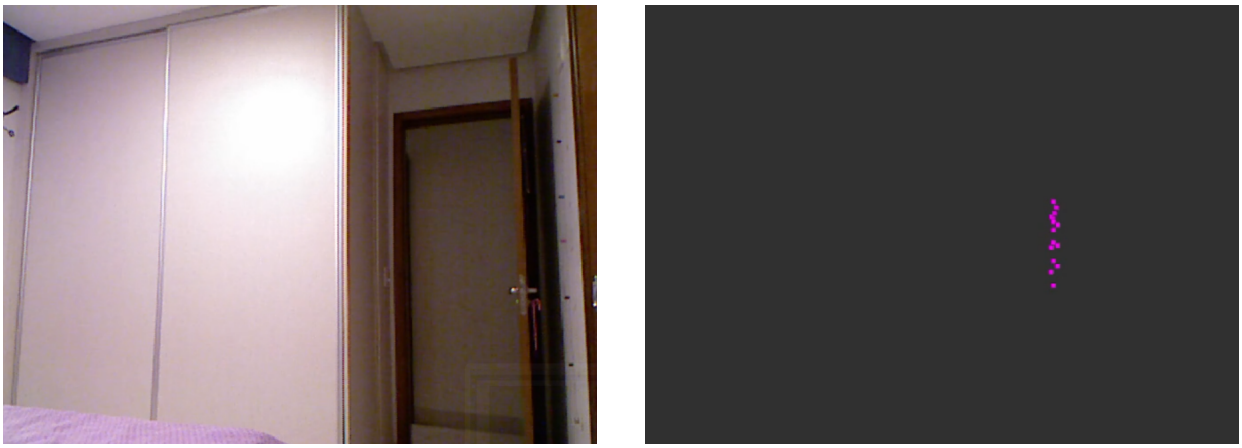


(h) Visualização do *Skeleton*

Figura 4.15: Desempenho do Reconhecimento Facial

Porém, alguns problemas foram observados e identificados durante a realização dos testes. O principal deles é uma limitação do pacote *openni_tracker*, que guarda a última posição de um usuário, mesmo quando este já saiu da cena por um período de tempo, criando um efeito fantasma no sistema, como pode ser visto na Figura 4.16. Este problema não é simples de corrigir, pois a

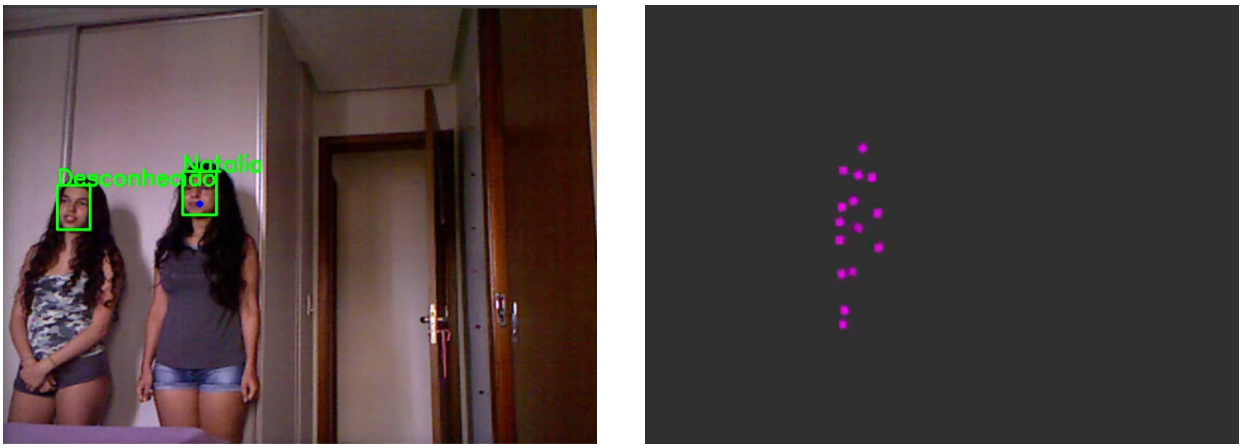
biblioteca utilizada não possui uma boa documentação, dificultando o trabalho de manutenção.



(i) Imagem da Câmera (j) Visualização do *Skeleton*

Figura 4.16: Falha de Agente Fantasma

Outro problema que pode ocorrer é o falso reconhecimento, que não é causado por falha no reconhecimento facial, mas sim pela atribuição errada da identificação ao agente. Isso porque, como explicado na Seção 3.7, o algoritmo usa a estimativa de posição do rosto para identificar qual *bounding box* pertence ao usuário que realizou a requisição, porém, existe um atraso entre a imagem que gerou a requisição e a imagem que chega no algoritmo de reconhecimento. Assim imagens com pessoas em movimento têm uma probabilidade maior de gerar falha no casamento da estimação de pose do rosto com a *bounding box*, gerando o fenômeno mostrado na Figura 4.17, em que o mesmo rótulo é atribuído para dois agentes, fazendo com que o filtro concentre as estimações em referencial global em apenas um agente. Esse problema pode ser enfrentado criando mais condições para atribuição de reconhecimento, como limitar o número de agentes por rótulo para apenas um por câmera.

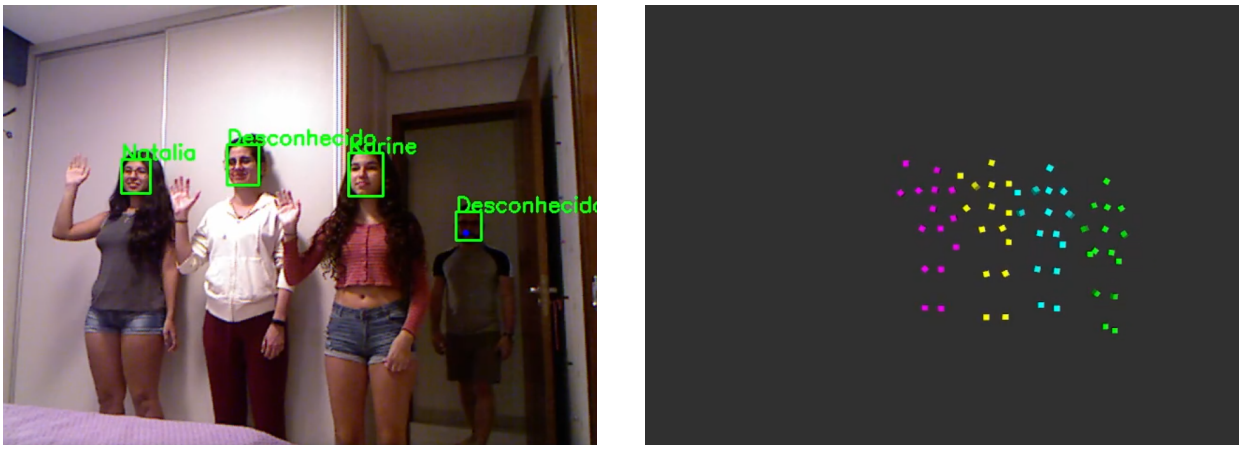


(a) Imagem da Câmera (b) Visualização do *Skeleton*

Figura 4.17: Falha de Falso Reconhecimento

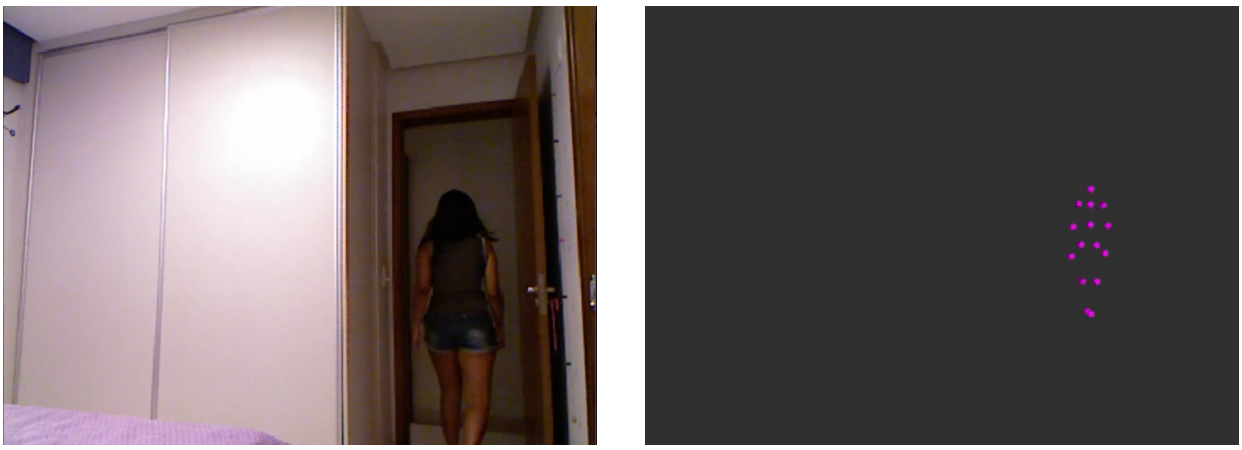
Além disso, no caso de múltiplas câmeras, os usuários só são rastreados em um mesmo agente caso sejam reconhecidos. Uma estratégia para resolver esse problema é adotar a fusão de detecções baseada também na interseção de porção espacial ocupada pelos agentes. Assim, caso um agente seja reconhecido em uma câmera, mas esteja de costas para outra câmera, ainda poderá ter suas medições filtradas pelos mesmos filtros.

O experimento também mostrou a robustez do sistema na persistência do reconhecimento quando o usuário por acaso não é reconhecido em dado *frame*, como mostrado na Figura 4.18 ou não possui o rosto visível, como pode ser visto na Figura 4.19.



(a) Imagem da Câmera (b) Visualização do *Skeleton*

Figura 4.18: Persistência de Rastreamento com Falha de Reconhecimento



(a) Imagem da Câmera (b) Visualização do *Skeleton*

Figura 4.19: Persistência de Rastreamento com Falha de Detecção

Capítulo 5

Conclusões

A crescente utilização de robôs nos mais diversos cenários trouxe a necessidade de novas pesquisas na área percepção visual que pudessem auxiliar no mapeamento de ambientes e localização de agentes. A pesquisa realizada neste trabalho propôs um sistema de mapeamento e localização por múltiplas câmeras, capaz de rastrear agentes humanos e agentes móveis. Esse sistema deve integrar o ambiente do Laboratório de Automação e Robótica da UnB, proporcionando o desenvolvimento de novas tecnologias em robótica cooperativa.

Para fazer a localização e rastreamento de robôs, o sistema contou com o auxílio de marcadores *ARUCO*, que são detectados por algoritmos de visão computacional pelo seu forte contraste com o ambiente e seu código binário único. Esses marcadores devem ser colados na superfície dos robôs móveis como forma de os identificar. As estimativas de pose dos marcadores são fornecidas no ambiente *ROS* em relação à câmera que os identifica, por isso, foi desenvolvido o módulo que realiza a transformação de coordenadas em referencial global e a fusão de estimativa providas por diversas câmeras, por meio de um Filtro de Kalman Estendido. Essa parte do sistema foi avaliada em simulação em um ambiente semelhante ao LARA, atestando o seu funcionamento.

A localização e rastreamento de agentes humanos foi feita utilizando câmeras *Kinect* e modelo de corpo articulado *skeleton*, capaz de identificar e rastrear 15 juntas do corpo humano. Além de desenvolver um módulo que realiza a integração em referencial global das juntas com filtragem por EKF, foi desenvolvido um módulo de reconhecimento facial, que tem o objetivo de servir como método de controle de acesso do laboratório.

Para concretização do módulo de reconhecimento facial, foram testados diferentes algoritmos de detecção de faces, com avaliação de qualidade de detecção e tempo de processamento. Concluiu-se que o *Ultra-Light-Fast-Generic-Face-Detector* foi mais performático, sendo o escolhido para compor o módulo de reconhecimento. A identificação por correspondência de *features* foi feita utilizando distância de *encodings* provida pela biblioteca *dlib*. O sistema de rastreamento e reconhecimento de agentes humanos foi testado em ambiente real com um grupo de pessoas, apresentando um bom comportamento.

Para que fosse possível prover as estimativas de localização de marcadores e pessoas, as câmeras do sistema precisaram ser localizadas no sistema de referência global. Por isso, foi desenvolvido

um algoritmo de mapeamento de câmeras com o auxílio dos marcadores *ARUCO*. Um experimento de simulação foi realizado para medir a qualidade do sistema e suas possíveis limitações.

Todo o sistema foi disponibilizado na forma de *API* em *python*, com o objetivo de ser de fácil utilização e manutenção. Dessa forma, a solução não fica atrelada ao ambiente para qual foi projetada, podendo ser adaptada e alterada com pequena necessidade de configuração.

5.1 Trabalhos Futuros e Sugestões de Melhorias

Um dos principais desafios no desenvolvimento deste trabalho foi a reunião das ferramentas e bibliotecas, muitas delas antigas e sem suporte, outras muito recentes, exigindo um ambiente com as configurações mais atualizadas. Isso torna o esforço de instalação muito dispendioso e complicado. Por isso, a principal melhoria que pode ser provida a esse sistema é a criação de um conjunto de *containers Docker* [50] com todas as dependências necessárias para executar o sistema. Os *containers* podem ser configurados automaticamente e são capazes de serem executados em qualquer sistema operacional que suporte *Docker*.

Outra grande necessidade do sistema é a disponibilização de uma documentação voltada aos usuários da *API*. Mesmo que existam poucas funções, um usuário ou outro desenvolvedor precisam entender qual é o propósito de cada módulo e quais são os formatos dos parâmetros requeridos.

Em relação ao rastreamento de agentes humanos, alguns problemas identificados poderiam ser corrigidos aplicando mais condições para correspondência de agentes, incluindo soluções que não dependessem de reconhecimento facial.

Outra melhoria interessante seria a adição de mais parâmetros de configuração que trouxessem personalização ao sistema. Por exemplo, a escolha de um algoritmo de detecção de faces que o usuário pudesse escolher dependendo do hardware disponível.

Por fim, para realmente consolidar o sistema desenvolvido, é necessária a instalação de todas as câmeras *Kinect* e *Videre* no laboratório LARA e a configuração do sistema de localização e mapeamento. A solução poderá ser validada por meio de testes e demonstrações práticas, com robôs identificados por marcadores *ARUCO* e pessoas cadastradas e identificadas por reconhecimento facial.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] BLANCO, P.; ABU-DAKKA, F.; ABDERRAHIM, M. Practical use of robot manipulators as intelligent manufacturing systems. *Sensors*, v. 18, p. 2877, 08 2018.

[2] BERNARDES, M. C.; ADORNO, B. V.; POIGNET, P.; BORGES, G. A. Semi-automatic needle steering system with robotic manipulator. *IEEE International Conference on Robotics and Automation ICRA*, p. 1595–1600, 05 2012.

[3] SASIADEK, J. Space robotics - present and past challenges. *2014 19th International Conference on Methods and Models in Automation and Robotics, MMAR 2014*, p. 926–929, 11 2014.

[4] THRUN, S.; BURGARD, W.; FOX, D. *Probabilistic Robotics (Intelligent Robotics and Autonomous Agents)*. [S.l.]: The MIT Press, 2005.

[5] CHERUBINI, A.; PASSAMA, R.; MELINE, A.; CROSNIER, A.; FRAISSE, P. Multimodal control for human-robot cooperation. *IEEE International Conference on Intelligent Robots and Systems*, 10 2013.

[6] NIKOLAIDIS, S.; HSU, D.; SRINIVASA, S. Human-robot mutual adaptation in collaborative tasks: Models and experiments. *International Journal of Robotics Research*, 36, n. 5-7, p. 618–634, 06 2017.

[7] FIGUEREDO, L. F. C.; ADORNO, B. V.; ISHIHARA, J. Y.; BORGES, G. A. Switching Strategy for Flexible Task Execution using the Cooperative Dual Task-Space Framework. *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, p. 1703–1709, 2014.

[8] FARIAS, C. M.; .ROCHA, Y. G.; FIGUEREDO, L. F. C.; BERNARDES, M. C. Design of singularity-robust and task-priority primitive controllers for cooperative manipulation using dual quaternion representation. *2017 IEEE Conference on Control Technology and Applications (CCTA)*, p. 740–745, 2017.

[9] LOYOLA, R. B.; SILVA, B. M. C.; ARAUJO, G. F. P.; ISHIHARA, J. Y.; BORGES, G. A. Controle de formação de sistemas multiagentes não-holonômicos em topologia fixa com prevenção de colisão. *São Carlos. Anais da XIV Conferência Brasileira de Dinâmica, Controle e Aplicações*, 2019.

[10] BRITO, C. G. *Desenvolvimento de um sistema de localização para robôs móveis baseado em filtragem Bayesiana não-linear*. 2017. Trabalho de conclusão de curso (Bacharelado em Engenharia Mecatrônica) — Universidade de Brasília, Brasília.

[11] SANTOS, D. F. *Implementação de Sistema de Localização para Futebol de Robôs Baseado em Sensores Inerciais*. 2020. Trabalho de conclusão de curso (Bacharelado em Engenharia Mecatrônica) — Universidade de Brasília, Brasília.

[12] HAMZEH, O.; ELNAGAR, A. A kinect-based indoor mobile robot localization. p. 1–6, 2015.

[13] GARCIA, R. O.; VALENTíN, L.; MARTINEZ-CARRANZA, J.; SUCAR, L. A fast algorithm for robot localization using multiple sensing units. p. 248–257, 01 2018.

[14] FRAHM, J.-M.; KöSER, K.; KOCH, R. Pose estimation for multi-camera systems. v. 3175, p. 286–293, 08 2004.

[15] KORTHALS, T.; WOLF, D.; RUDOLPH, D.; M., H.; RüCKERT, U. Fiducial marker based extrinsic camera calibration for a robot benchmarking platform. *2019 European Conference on Mobile Robots (ECMR)*, p. 1–6, 2019.

[16] SINHA, A.; CHAKRAVARTY, K. Pose Based Person Identification Using Kinect. *IEEE International Conference on Systems Man and Cybernetics Conference Proceedings*, p. 497–503, 2013.

[17] STANFORD-ARTIFICIAL-INTELLIGENCE-LABORATORY. *Robotic Operating System*. `https://www.ros.org`.

[18] CONLEY, K.; THOMAS, D. *rospy*. Disponível em: `http://wiki.ros.org/rospy`.

[19] QUIGLEY, M.; FAUST, J.; GERKEY, B. *roscpp*. Disponível em: `http://wiki.ros.org/roscpp`.

[20] SZALóKI, D.; KOSZó, N.; CSORBA, K.; TEVESZ, G. Marker localization with a multi-camera system. *ICSSE 2013 - IEEE International Conference on System Science and Engineering, Proceedings*, 07 2013.

[21] MOLDAGALIEVA, A. et al. Computer Vision-Based Pose Estimation of Tensegrity Robots Using Fiducial Markers. *IEEE/SICE International Symposium on System Integration*, p. 478–483, 01 2019.

[22] GARRIDO-JURADO, S.; MUNOZ-SALINAS, R.; MADRID-CUEVAS, F. J.; MARIN-JIMENEZ, M. J. Automatic generation and detection of highly reliable fiducial markers under occlusion. *Rattern Recognition*, 47, n. 6, p. 2280–2292, 06 2014.

[23] BRADSKI, G. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[24] ANJUM, M. L.; AHMAD, O.; ROSA, S.; YIN, J.; BONA, B. Skeleton Tracking Based Complex Human Activity Recognition Using Kinect Camera. *Lecture Notes in Artificial Intelligence*, 8755, p. 23–33, 2014.

[25] RYSELIS, K.; PETKUS, T.; BLAZAUSKAS, T.; MASKELIUNAS, R.; DAMASEVICIUS, R. Multiple kinect based system to monitor and analyze key performance indicators of physical training. *Human-centric Computing and Information Sciences*, v. 10, 12 2020.

[26] OPENNI. *OpenNI Library*. 2012. Disponível em: https://github.com/OpenNI/OpenNI/tree/unstable.

[27] PRIMESENSE. *Nite Framework*. 2016. Disponível em: https://github.com/arnaud-ramey/NITE-Bin-Dev-Linux-v1.5.2.23.

[28] TRACKER, O. *OpenNI Tracker*. 2014. Disponível em: https://github.com/ros-drivers/openni_tracker.

[29] PARAMJIT, K.; KEWAL, K.; SURESH, K. S.; TANUJ, K. Facial-recognition algorithms: A literature review. *Medicine, Science and the Law*, v. 60, n. 2, p. 131–139, 2020.

[30] HAMILTON, W. R. Xxii. on quaternions; or on a new system of imaginaries in algebra. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, Taylor & Francis, v. 29, n. 192, p. 113–122, 1846.

[31] SPONG, M.; HUTCHINSON, S.; VIDYASAGAR, M. *Robot Modeling and Control*. Wiley, 2005. ISBN 9780471649908. Disponível em: <https://books.google.com.br/books?id=jyD3xQEACAAJ>.

[32] HARTLEY, R.; ZISSERMAN, A. *Multiple View Geometry in Computer Vision*. [S.l.]: Cambridge University Press, 2004.

[33] GREWAL, M.; ANDREWS, A. Kalman filtering: theory and practice using matlab. *New York: John Wiley and Sons*, v. 14, 01 2001.

[34] SENSORKINECT. *PrimeSense Sensor Module for OpenNI* . 2012. Disponível em: https://github.com/avin2/SensorKinect.

[35] CAMERA1394. *ROS driver for devices supporting the IEEE 1394 Digital Camera (IIDC) protocol*. 2017. Disponível em: https://github.com/ros-drivers/camera1394.

[36] ROHMER, E.; SINGH, S. P. N.; FREESE, M. Coppeliasim (formerly v-rep): a versatile and scalable robot simulation framework. *Proc. of The International Conference on Intelligent Robots and Systems (IROS)*, 2013. Disponível em: www.coppeliarobotics.com.

[37] LEPETIT, V.; MORENO-NOGUER, F.; FUA, P. Epnp: An accurate o(n) solution to the pnp problem. *International Journal of Computer Vision*, v. 81, 02 2009.

[38] RABAUD, V. *image_pipeline*. Disponível em: https://github.com/ros-perception/image_pipeline.

[39] MARKLEY, L.; CHENG, Y.; CRASSIDIS, J.; OSHMAN, Y. Averaging quaternions. *Journal of Guidance, Control, and Dynamics*, v. 30, p. 1193–1196, 07 2007.

[40] Bó, A. P. L. *Desenvolvimento de um Sistema de Localização 3D para Aplicação em Robôs Aéreos*. 2007. Dissertação de Mestrado em Engenharia Elétrica — Universidade de Brasília.

[41] RAHMAN, M.; GAMADIA, M.; KEHTARNAVAZ, N. Real-time face-based auto-focus for digital still and cell-phone cameras. *Proceedings of the IEEE Southwest Symposium on Image Analysis and Interpretation*, p. 177–180, 04 2008.

[42] PONNUSAMY, A. *cvlib - high level Computer Vision library for Python*. 2018. Disponível em: `https://github.com/arunponnusamy/cvlib`.

[43] KING, D. E. Dlib-ml: A machine learning toolkit. *Journal of Machine Learning Research*, v. 10, p. 1755–1758, 2009.

[44] LINZAER. *Ultra-Light-Fast-Generic-Face-Detector-1MB*. 2019. Disponível em: `https://github.com/Linzaer/Ultra-Light-Fast-Generic-Face-Detector-1MB`.

[45] YANG, S.; LUO, P.; LOY, C. C.; TANG, X. Wider face: A face detection benchmark. *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[46] Cartucho, J.; Ventura, R.; Veloso, M. Robust object recognition through symbiotic deep learning in mobile robots. *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, p. 2336–2341, 2018.

[47] PRATEEKMEHTA59. *Celebrity-Face-Recognition-Dataset*. 2017. Disponível em: `https://github.com/prateekmehta59/Celebrity-Face-Recognition-Dataset`.

[48] Hershberger , Dave and Gossow, David and Faust, Josh and Woodall, William. *rviz*. 2021. Disponível em: `https://github.com/ros-visualization/rviz`.

[49] HUYNH, D. Metrics for 3d rotations: Comparison and analysis. *Journal of Mathematical Imaging and Vision*, v. 35, p. 155–164, 10 2009.

[50] MERKEL, D. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, v. 2014, 03 2014.

ANEXOS

I. EXEMPLOS DE UTILIZAÇÃO DA API

1. **Exemplo de Rastreamento de Marcadores**

   Exemplo de código utilizado para iniciar as câmeras do simulador *CoppeliaSim* e realizar a integração em referencial global do rastreamento. Este código foi responsável pelo experimento da Seção 4.3.

```
import rospy
from API.aruco_detect import ArucoDetect
from API.sim_camera import SimCamera
from API.marker_global_tracker import GlobalMarkerTracker

rospy.init_node("tracker", anonymous=True)

# Inclui 13 cameras simuladas no sistema e executa rastreador de marcadores
N = 13
for i in range(1,N+1):
    SimCamera("camera"+str(i)).start()
    ArucoDetect(camera_name="/sim_ros_interface/camera"+str(i),
                image_name='image_raw', dictionary=11).start()

# Inicia modulo de integracao em referencial global
GlobalMarkerTracker("cameras.json")

rospy.spin()
```

2. **Exemplo de Rastreamento de Agentes Humanos**

Exemplo de código utilizado para fazer a integração em referencial global das detecções de agentes humanos e reconhecimento facial. Este código foi responsável por executar os experimentos da Seção 4.6.

```
import rospy
from API.kinect_tracker import KinectTracker
from API.face_recognition_module import FaceRecognitionModule
from API.human_global_traker import GlobalHumanTracker

rospy.init_node("tracker", anonymous=True)

# Inicia rastreador de skeleton
kinect = KinectTracker(camera_name="kinect1", device_id='A00363904314053A')
kinect.start()
# Inicia integracao em referencial global de skeleton
ght = GlobalHumanTracker("kinects.json")
# Inicia reconhecimento facial
frm = FaceRecognitionModule(cameras=["kinect1"], device_ids=['
    A00363904314053A'], register_path="registro.pickle")

rospy.spin()
```