Universidade de Brasília
Faculdade de Tecnologia

Estimação de pose com marcadores Aruco e ambiente virtual para robótica cooperativa

Gabriel Tambara Rabelo

Marcos Eduardo Monteiro Junqueira

PROJETO FINAL DE CURSO

ENGENHARIA DE CONTROLE E AUTOMAÇÃO

Brasília

2023

Universidade de Brasília
Faculdade de Tecnologia

Estimação de pose com marcadores Aruco e ambiente virtual para robótica cooperativa

Gabriel Tambara Rabelo
Marcos Eduardo Monteiro Junqueira

Projeto Final de Curso submetido como requisito parcial para obtenção do grau de Engenheiro de Controle e Automação

Orientador: Prof. Dr. Geovany Araujo Borges
Coorientador: Prof. Dr. João Yoshiyuki Ishihara

Brasília
2023

Tambara Rabelo, Gabriel.
T154e Estimação de pose com marcadores Aruco e ambiente virtual para robótica cooperativa / Gabriel Tambara Rabelo; Marcos Eduardo Monteiro Junqueira; orientador Geovany Araujo Borges; coorientador João Yoshiyuki Ishihara. -- Brasília, 2023.
121 p.

Projeto Final de Curso (Engenharia de Controle e Automação) -- , 2023.

1. Estimação de pose. 2. Marcadores Fiduciais. 3. Filtro de Kalman. 4. ArUco. I. Eduardo Monteiro Junqueira, Marcos. II. Borges, Geovany Araujo, orient. III. Ishihara, João Yoshiyuki, coorient. IV. Título

Universidade de Brasília
Faculdade de Tecnologia

Estimação de pose com marcadores Aruco e ambiente virtual para robótica cooperativa

Gabriel Tambara Rabelo
Marcos Eduardo Monteiro Junqueira

Projeto Final de Curso submetido como requisito parcial para obtenção do grau de Engenheiro de Controle e Automação

Trabalho aprovado. Brasília, 21 de Dezembro de 2023:

Prof. Dr. Geovany Araujo Borges,
UnB/FT/ENE
Orientador

Prof. Dr. João Yoshiyuki Ishihara,
UnB/FT/ENE
Coorientador

Prof. Dr. Adolfo Bauchspiess
Examinador interno

Prof. Dr. Roberto de Souza Baptista
Examinador interno

Brasília
2023

Agradecimentos

Meus sinceros agradecimentos aos meus pais, Alessandra e Allan que, com muito esforço e apreço, permitiram que me tornasse quem sou, e pudesse conquistar o que considero uma grande realização em minha vida. Também às minhas irmãs, Brenda e Natália, por me acompanharem na juventude até o presente marco, em muitas das felicidades e adversidades encontradas. Também expresso meu grande carinho a todos os meus amigos da Mecajun e colegas de turma, pela companhia desde o começo desta longa jornada, e sem eles tudo seria muito mais difícil, principalmente ao André, Thi, Alex, David, Ema, Beto e, não menos importante, ao Marcos, pela sua companhia resiliente, e bagagem intelectual e virtuosa. Aprecio também a companhia e sabedoria dos presentes no LARA, como o Eric que através do projeto Edubot me cativou a seguir este rumo, e como os professores Geovany e Roberto, que observaram os instantes finais desta caminhada. Por fim, destaco meus agradecimentos principalmente aos meus primos, Yuri e Fábio, por me cativarem e me mostrarem a beleza na engenharia, quando ainda mal calculava, ao serem grandes exemplos.

Gabriel Tambara Rabelo

Gostaria de expressar minha profunda gratidão a todos que contribuíram para a realização deste trabalho. Este projeto não teria sido possível sem o apoio e a colaboração de várias pessoas incríveis. Em primeiro lugar, quero agradecer aos professores Geovany Araujo Borges e Roberto de Souza Baptista, que apesar de muito ocupados, sempre dedicaram o máximo de sua disponibilidade para nos auxiliar em cada etapa. Suas orientações, recomendações e discussões foram fundamentais para o desenvolvimento deste trabalho. Além disso, gostaria de agradecer ao colegas do LARA que forneceram companhia e suporte ao longo desta caminhada. Quero também expressar minha gratidão aos meus colegas de curso e grandes amigos: Alexandre, David, Emanuel e Luis. Suas contribuições, discussões e companhia foram inestimáveis e tornaram esta jornada mais significativa e prazerosa. Também gostaria agradecer ao meu grande amigo Gabriel, com o qual estou compartilhando este projeto, pela sua infinda dedicação e esforço que tornaram este projeto possível. Por fim, um agradecimento especial aos meus familiares, sobretudo meus pais Marcos e Valderlândia e minha namorada Ágatha, pelo constante apoio emocional e encorajamento. Sua compreensão e paciência foram fundamentais para superar desafios e alcançar este marco.

Marcos Eduardo Monteiro Junqueira

Resumo

A robótica cooperativa, área que integra sistemas robóticos em sinergia, utiliza do advento da visão computacional para identificação de pose, que é crucial para permitir que múltiplos robôs atuem em conjunto na realização de tarefas complexas. Ao compartilhar informações de câmeras, os robôs podem determinar com certa precisão a posição e orientação de objetos ou agentes no ambiente. No presente documento, é apresentado o projeto de um ambiente de identificação por múltiplas câmeras de diferentes tipos, dentre as previamente disponíveis no laboratório de robótica e automação da UnB, o LARA, utilizando marcadores ArUco, com auxílio do ambiente ROS, objetivando a fundamentação de um sistema para controle de cenários multi-robôs, através da identificação de poses de objetos ou pontos de interesse com o uso de bibliotecas de identificação compatíveis com os sistemas de robótica cooperativa previamente utilizadas no laboratório, com a utilização de filtro de Kalman para melhorar os resultados de estimação iterativa das poses, tratando incertezas e ruído de medições, buscando permitir uma futura integração com sistemas de controle entre manipuladores. Além do processo de coleta e filtragem dos dados posicionais, também é apresentado um gêmeo virtual do ambiente detectado, visando melhor analisar o comportamento dos dados coletados bem como compreender a fundo a filtragem estocástica realizada. O projeto foi desenvolvido com uma modelagem de posição constante para os objetos detectados e apresentou localizações condizentes com o ambiente real, denotando uma recriação de situações reais em ambiente virtualizado. São destacadas possíveis melhorias e adaptações no sistema para os próximos passos na criação de um ambiente de robótica cooperativa bem como também são avaliadas diferentes formas de *tuning* do filtro para adequação à ruídos.

Palavras-chave: Estimação de pose. Marcadores Fiduciais. Filtro de Kalman. ArUco.

Abstract

Cooperative robotics, an area that integrates robotic systems synergistically, utilizes the advent of computer vision for pose identification, which is crucial for enabling multiple robots to work together in performing complex tasks. By sharing camera information, robots can accurately determine the position and orientation of objects or agents in the environment. This document presents the project of a multiple camera identification environment using different types of cameras previously available in the robotics and automation laboratory at UnB, known as LARA, using ArUco markers with the assistance of the ROS environment. The goal is to establish a foundation for a system to control multi-robot scenarios by identifying poses of objects or points of interest using identification libraries compatible with cooperative robotics systems previously used in the laboratory. The Kalman filter is employed to improve the iterative estimation results of poses, addressing uncertainties and measurement noise, aiming to enable future integration with control systems between manipulators. In addition to the process of collecting and filtering positional data, a virtual twin of the detected environment is also presented to better analyze the behavior of the collected data and to thoroughly understand the stochastic filtering performed. The project was developed with a constant position modeling for the detected objects and showed consistent locations with the real environment, indicating a recreation of real situations in a virtualized environment. Possible improvements and adaptations to the system for the next steps in creating a cooperative robotics environment are highlighted, and different tuning methods for the filter to accommodate noise are evaluated.

Keywords: Pose estimation. Fiducial Markers. Kalman Filter. ArUco.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

API	Application Programming Interface	19
ArUco	Augmented Reality University of Cordoba	32
CCD	Charge-Coupled Device	27
CMOS	Complementary Metal-Oxide-Semiconductor	27
LARA	Laboratório de Robótica e Automação	17
MIT	Massachussets Institute of Technology	16
OpenCV	Open Source Computer Vision	32
PnP	Perspective-n-Point	32
ROS	Robot Operating System	21
RPC	Remote Procedure Call	21
Rviz	ROS visualization	50
SLAM	Simultaneous Localization And Mapping	16
UKF	Unscented Kalman Filter	19
UnB	Universidade de Brasília	5
USB	Universal Serial Bus	31
VIO	Visual Inertial Odometer	16
VTNT	Veículo Terrestre Não Tripulado	17

Lista de símbolos

Sumário

1 Introdução

1.1 Motivação

É de grande repercussão social e econômica que os efeitos da atuação dos robôs na sociedade impõem condições das mais diversas sobre os indivíduos e os processos a que estão submetidos. Historicamente, os robôs demonstram sua importância na sociedade desde o primeiro robô industrial autônomo, o Unimate, cuja função era de atuar em uma linha de produção da General Motors, em 1961, carregando e soldando peças metálicas para montagens de veículos automotivos, (AUTOMATE, 2021). A linha de produção com a atuação do Unimate hoje pode ser considerada perigosa para operários da época, bem como também ineficiente diante da possibilidade de um cenário puramente automatizado, o que traz à tona a importância de se abordar o impacto na sociedade de tal aparato tecnológico, bem como a necessidade por uma busca pela automação e integração maior de tais máquinas em ambiente de produção.

Contudo, a atuação da robótica se vê atrelada à um setor cujo índice de acidentes no Brasil está diretamente relacionada à presença de força de trabalho atuante no mesmo, o que levou a uma necessidade por busca de oportunidades mais seguras, seja através de mudanças dentro do ambiente de produção, ou simplesmente por uma migração para outros setores de trabalho (FILHO, 1999). É natural que essa migração envolva uma redução do potencial de desenvolvimento nas mais diversas áreas de atuação industrial, além do acréscimo aos índices de desemprego.

Em contrapartida, o avanço na área de robótica cooperativa visa atuar na integração dos robôs nos ambientes de trabalho de forma conjunta entre si e entre indivíduos, com segurança e eficiência, utilizando de normas como a (ISO/TS, 2016) e da tecnologia disponível para tal. Um dos exemplos de aplicação tecnológica que corrobora para esta integração é a visão computacional, responsável por permitir e/ou auxiliar ambientes robóticos na sua localização e noção espacial de modo a poder identificar pontos de interesse, bem como possivelmente orientar outros sistemas computacionais atuantes no local, das informações espaciais obtidas, criando assim, um ambiente integrado com mais informações e mais responsivo ao meio no qual está inserido. A redução dos riscos quanto aos robôs em uma linha de produção, portanto, não apenas se mostra de grande importância para a melhoria da qualidade de vida daqueles que atuam no setor, como também representa um avanço nacional no contexto tecnológico. Contudo, os benefícios das tecnologias atreladas não se resumem a tal. Pode-se comentar, por exemplo, do uso de visão computacional para atuação em treinamentos de controle de manipuladores por pacientes com lesão cervical (YOON

JAE KIM HYUNG SEOK NAM, 2019).

1.2 Contextualização

Há uma grande gama de aplicações da visão computacional aplicada em localização e estimação de posicionamento de entidades em ambientes dos mais diversos. Essas aplicações não apenas contextualizam um cenário tecnológico no que tange o presente projeto mas também demonstram o caráter inovador e o potencial científico, bem como abordam possibilidades de aprimoramento. Essa área é composta por um campo interdisciplinar que combina técnicas de processamento de imagem, aprendizado de máquina, geometria computacional, entre outros, e desempenha um papel crucial na interpretação de ambientes por computadores.

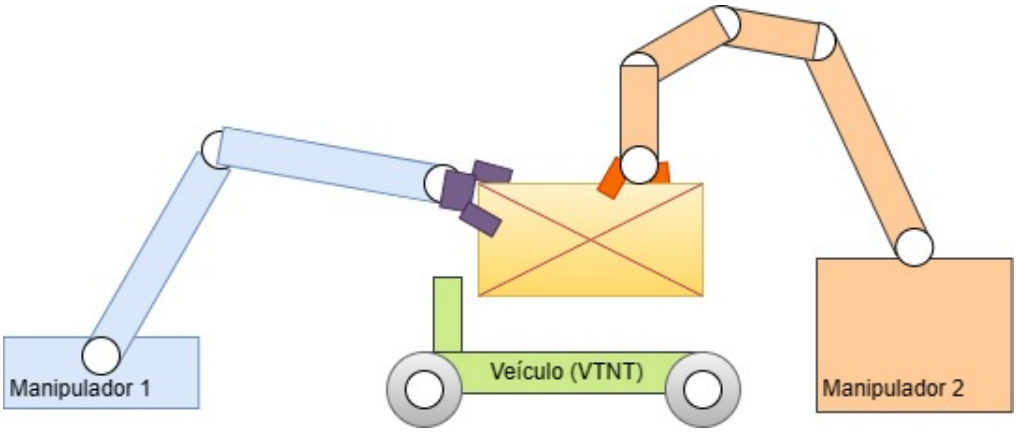
Lawrence Gilman Roberts, um engenheiro norte americano do MIT, foi responsável por iniciar o estudo na área visão computacional e percepção com sua tese de "Percepção de máquinas de sólidos tridimensionais", vide (ROBERTS, 1963), que abordou desde a captação de imagens à localização de bordas e reconhecimento de objetos baseado em modelos, (SHAPIRO, 2020). Desde então, houveram diversos avanços tecnológicos no ramo que implicaram na utilização de novas ferramentas bem como no desenvolvimento de novas aplicações. Uns dos maiores problemas na área é o de localização. Tal problema foi inicialmente abordado em (ZHANG; NAVAB, 2000), com a publicação de nome "Rastreamento e Estimação de Pose para Localização Assistida por Computador em Ambientes Industriais". Este artigo já abordava a necessidade de reconhecimento de características específicas bem como a necessidade de acompanhá-las.

Há diversas formas de se realizar o processo de localização espacial, e alguns métodos envolvem tecnologias como SLAM e VIO, contudo, esses métodos, apesar de se adequadamente implementados performarem bem em condições ideais, para a maioria dos cenários, existem fatores como baixa luminosidade e/ou inadequada qualidade de imagem. Para esses cenários, são utilizados marcadores fiduciais, o que compõe um grande grupo de marcadores, cada um com suas características e aplicabilidades, vide (KALAITZAKIS et al., 2021). Trabalhos foram realizados utilizando marcadores em diversas aplicações, como por exemplo, na detecção e captura de agentes semi colaborativos por robôs espaciais com manipuladores, vide (OPROMOLLA et al., 2022). Projetos como esse abordam aspectos comuns à maioria das soluções propostas para problemas de localização, que envolvem a criação de arquiteturas para estimação de pose, considerando certa limitação de *hardware* e limitações diversas causadas pelo ambiente de aplicação; envolvem a solução numérica do problema de detecção relativas aos sensores utilizados, como solução do problema PnP com critérios pre-definidos; análise de fidelidade do sistema observado/detectado, relacionados à calibração das câmeras e do processo de determinação da pose final.

1.2.1 Definição do problema

O LARA possui um ambiente de robótica colaborativa que compõe diversos manipuladores robóticos e VTNTs a disposição para diversas atuações no ramo da engenharia. Necessita-se que esse ambiente seja integrado de modo que os manipuladores possam interagir entre si e com outros robôs de modo a criar um espaço interconectado de integração via internet, que compartilha informações posicionais e permita que usuários possam comandar os robôs para executar tarefas complexas e que exijam atuação conjunta. Faz-se necessário, portanto, a detecção destas informações posicionais além da centralização e publicação destas informações de modo que os sistemas robóticos possam utilizá-las. A figura 1 retrata um exemplo de atuação conjunta que envolve dois manipuladores e um VTNT, de modo que para a sustentação da caixa a ser carregada, dois manipuladores trabalham coordenadamente para garantir uma trajetória específica até a posição do VTNT, que deve mover a caixa para outro destino, cenário comum em ambientes de controle de estoque como armazéns.

Figura 1 – Representação de vista lateral de trabalho conjunto de robótica cooperativa

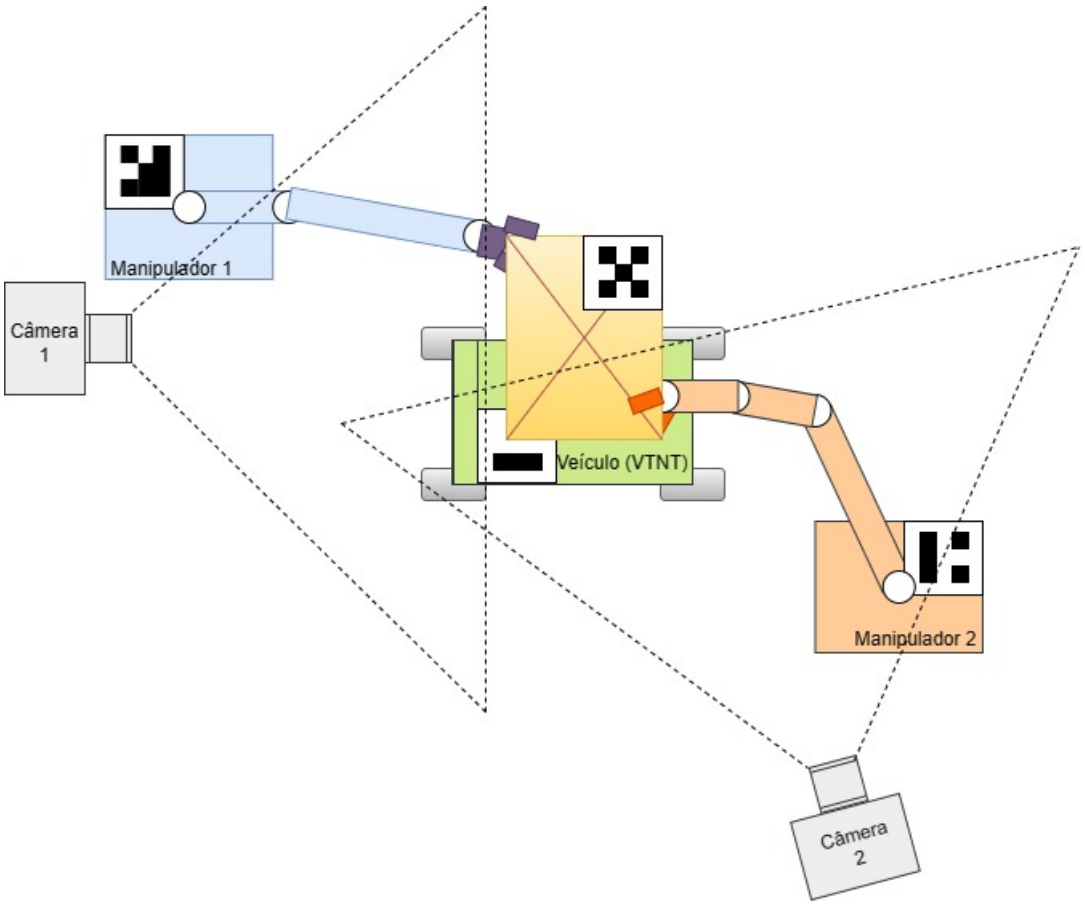


Uma das formas de se obter tais informações posicionais dos elementos presentes em um ambiente de interação de robótica cooperativa é com o uso de identificadores únicos e sensores, como retratado pela figura 2. Nesta ocasião, as câmeras conseguem obter, individualmente a posição dos marcadores relativa às mesmas, e caso exista um sistema centralizador capaz de recolher e integrar os dados de todas as câmeras, pode-se garantir que o computador compreenda as posições de todos os marcadores, e portanto das entidades relevantes, observadas por qualquer câmera dentro do ambiente, permitindo assim, a ação conjunta.

1.2.2 Objetivos e metas

Para a realização da interação entre manipuladores e VTNTs, se faz necessário o uso de um sistema de captação de dados do ambiente para localização de pontos de interesse ou simplesmente pontos de referência para que os sistemas de posicionamento de cada um dos agentes integrados possa se localizar. Objetivou-se para o presente projeto, portanto, a criação

Figura 2 – Representação de vista superior de cenário com marcadores fiduciais e câmeras



de um sistema de visualização descentralizado (com múltiplas fontes de detecção de posição fixas), capaz de adicionar novos agentes de coleta de dados, aqui definidos como câmeras, e que esses dados (posição relativa dos objetos observados para as câmeras) passassem por fusão sensorial para melhorar a qualidade de informações obtidas, permitindo seu uso em aplicações de menor erro entre o valor posicional real e o valor obtido entre as câmeras. Também definiu-se a utilização de marcadores fiduciais como os designadores únicos para as entidades móveis e fixas.

Dentre os requisitos iniciais definidos do projeto, foi definida a necessidade de razoabilidade posicional relacional entre os objetos detectados, buscando inicialmente apenas resultados condizentes de posição e orientação com o ambiente físico de testes. Definiu-se que seria possibilitada a utilização de múltiplas formas de coleta de dados visuais, especificamente com diferentes tipos de câmera, de preferência câmeras usualmente utilizadas em projetos de tal aplicação, conforme disponibilidade, para uso no sistema. Além destes, também foi definido que o sistema deveria ser capaz de integrar com os sistemas vigentes e funcionais atualmente no LARA, o que envolveria compatibilidade tecnológica para com os sistemas de controle dos manipuladores e dos veículos.

1.3 Trabalhos anteriores

Previamente, foi realizado um trabalho no LARA com localização e mapeamento, também utilizando marcadores ArUco, vide (BORGES, 2021). Este trabalho desenvolveu um sistema de localização para robôs, com marcadores fiduciais, e também de localização para humanos, com detecção de face, via treinamento de rede neural. Seu foco foi em desenvolver uma API de fácil uso, na linguagem Python. O trabalho demonstrou grande potencial de desenvolvimento de sistemas de localização e retratou a facilidade com que a linguagem Python se mostra para implementação de sistemas complexos. Diante disso, pode-se atuar em uma reimplementação em C++ buscando resultados mais rápidos, bem como também pode-se estender a implementação através de novas formas de detecção e recriação de pose. O sistema também se limita ao retorno dos dados de posicionamento, a princípio de um em um, diferente do presente projeto, que visa criar um ambiente virtualizado para localizar múltiplos objetos marcados publicados e captados por múltiplas câmeras. Algumas outras recomendações pela autora do projeto são a facilitação do preparo do ambiente, tendo em vista a ampla quantidade de sistemas a serem instalados, o que viria a tornar isto um processo tedioso e sujeito a problemas, principalmente conforme os sistemas se tornam obsoletos.

Houveram trabalhos realizados também no laboratório, com o UR3, o manipulador cooperativo complacente da empresa Universal Robots, como o realizado por Garcia (2019), que envolveu a estimação das forças de interação do UR3 com o uso de estimadores de estados, o UKF, juntamente à identificação do modelo dinâmico do robô. Esse trabalho realizou a estimativa e obteve resultados condizentes, contudo ainda haveria a necessidade da busca por resultados de maior acurácia do esforço do manipulador. Os modelos cinemático e dinâmico desenvolvidos apresentaram singularidades que vieram a gerar respostas não ideais em certas trajetórias. Outro fato pertinente é que o estimador de esforços externos foi realizado majoritariamente no software Matlab, e dadas as suas limitações temporais, não foi possível a realização de cálculos em tempo real para a estimação com a taxa de amostragem do manipulador. Os desenvolvimentos matemáticos realizados foram simbolicamente simplificados utilizando da ferramenta Wolfram Mathematica, e suas soluções numéricas realizadas também pelo Matlab. Essas informações denotam um sistema capaz de realizar certa complacência quando em malha de controle, contudo, ainda seria necessária a utilização de mais tecnologias para garantir segurança e acurácia nas aplicações do manipulador, como por exemplo, na integração deste sistema com um ambiente de localização.

Já o trabalho de Cesarino (2020) envolveu o uso de softwares como Maple e Matlab visando uma remodelagem matemática do UR3, não abordando a estimação de esforços externos. O processo se resumiu à parametrização de Newton-Euler, diferentemente do modelo anterior realizado com a parametrização de Euler-Lagrange, seguida pela linearização e então a busca pelos parâmetros desconhecidos se deu pelos método dos mínimos quadrados. O resultado obtido foi de 73.14% de *fitness*. Contudo, houve um problema no processo de

identificação da orientação da ferramenta dada pela escolha da orientação em termos de rolamento, arfagem e guinada, cuja mudança recomendada seria a de representação via quatérnions duais, que impactariam na mudança dos jacobianos do modelo, o que não pôde ser feito. Uma diretriz para a continuação de seu trabalho envolveria a adaptação também do controlador para a linguagem C, que desempenhou bem, somado à realização de maiores testes da implementação, o que não foi tão viável dada a inacessibilidade ao laboratório na época de desenvolvimento. O problema enfrentado neste projeto de determinação da orientação da ferramenta, quando unido a um sistema de localização, poderia possivelmente ser resolvido, ao criar referenciais visuais para a ferramenta e sua posição no espaço tridimensional do laboratório.

Para a interação dos projetos com o controlador do UR3, foi utilizado um programa desenvolvido por Rafael, vide Matos (2020), que atuou no LARA e permitiu essa interação de forma simplificada entre as máquinas. A análise desse ambiente desenvolvido se mostra essencial para determinar critérios de ambientes do sistema de localização, visto que a integração entre estes seria fundamental para a criação de um ambiente de robótica colaborativa.

1.4 Apresentação do documento

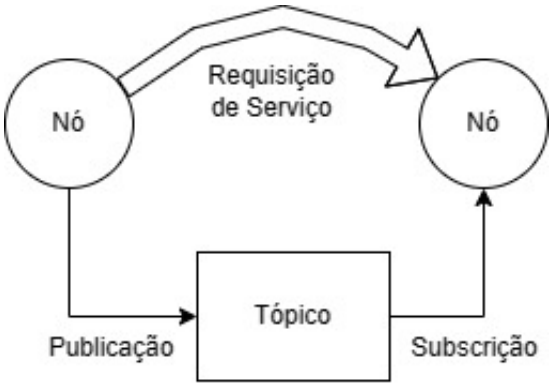
O presente documento se organiza de forma a apresentar os fundamentos utilizados para o projeto na seção 2, retratando o funcionamento geral de câmeras e motivando uma análise mais aprofundada de como tais equipamentos podem influenciar nos resultados obtidos; além de apresentar como funciona o ambiente ROS, o qual já é presente no LARA e do qual se fez uso para o desenvolvimento do projeto, bem como também aborda a teoria por trás da representação matemática de posição e orientação de marcadores, seguido da fundamentação teórica de processos estocásticos e filtragem estocástica, com ênfase no filtro de Kalman, empregado no projeto. A seção 3 aborda o desenvolvimento do projeto, detalhando os passos tomados pelo grupo para a realização e implementação do código a fim de destacar os principais elementos que influenciaram nas escolhas de projeto e na arquitetura do sistema. Na seção 4 são apresentados os resultados obtidos da implementação realizada, conjuntamente com uma análise destes. Já na seção 5 há uma breve conclusão sobre o funcionamento do sistema, destacando possíveis melhorias e próximos passos para desenvolvimento do projeto. Por fim, o documento aborda as referências bibliográficas utilizadas para fundamentar toda a teoria empregada e segue com os apêndices que compõem os códigos desenvolvidos pertencentes ao projeto.

2 Fundamentação

2.1 ROS

O ROS é um meta sistema operacional de código aberto voltado para o controle de robôs e integração deste com outros sistemas, fornecendo funcionalidades como abstração de hardware, sistema de comunicação entre processos por mensagens, controle de dispositivos de baixo nível, gerenciamento de pacotes, entre outras ferramentas. Além disso, o ROS também fornece ferramentas para desenvolvimento e execução de código entre múltiplos computadores. Apresenta também a capacidade de ser integrado com módulos, bibliotecas e *toolkits* para realização de aplicações em tempo real. [1]

Figura 3 – Grafo de Computação ROS com dois nós



Fonte: Adaptado de http://wiki.ros.org/ROS/Concepts

Em adição, o ROS trabalha utilizando um sistema de Grafo de Computação[2],como observado na Figura 3, composto por uma rede *peer-to-peer* de processos ROS utilizado para o processamento de dados. Estes processos podem ser implementados em máquinas distintas e por diferentes estilos de comunicação, como RPC, *streaming* de dados, entre outros estilos. Um sistema ROS pode ser dividido no seguintes elementos:

- **Nós:** São processos que realizam algum tipo de computação, geralmente desenvolvidos para serem modulares e especializados, de forma que um sistema de controle será composto por diversos nós.
- **Mestre:** O Mestre ROS é responsável por gerenciar o registro de nomes e consultas para o resto do grafo de computação, de forma a possibilitar a comunicação entre nós, troca de mensagens e invocação de serviços, além de ser responsável pela parte de armazenamento de dados.

[1] Texto adaptado de: http://wiki.ros.org/ROS/Introduction
[2] Texto adaptado de: http://wiki.ros.org/ROS/Concepts

- **Mensagens:** Mensagens são estruturas de dados utilizadas nas comunicações entre nós. Além de suportarem os tipos primitivos de dados e *arrays*, é também possível aninhá-los e combiná-los semelhantemente a *structs* em C.
- **Tópicos:** Um tópico é um identificador para o conteúdo de um mensagem, de forma a seguir o padrão *publisher*/*subscriber*. Nesse padrão, um nó *publisher* é capaz de enviar mensagens, publicando-as sob um determinado tópico, de forma que todos os nós *subscribers* têm acesso a informação publicada. Além disso, cada nó pode ser *subcriber* ou *publisher* de múltiplos tópicos distintos simultaneamente.
- **Serviços:** Serviços servem como uma alternativa na comunicação entre nós, utilizando um padrão requisição/resposta. Nesse contexto, um nó disponibiliza um serviço sob um nome específico e um nó cliente pode utilizar este serviço mandando uma mensagem de requisição e esperando por uma mensagem de resposta.
- **Bags:** *Bags* é um formato utilizados para guardar e reutilizar dados de mensagens ROS, podendo ser utilizado para guardar dados de sensoreamento, por exemplo.

Nessa arquitetura, o mestre ROS serve como índice e gerenciador do Grafo de Computação, armazenando informações dos tópicos e serviços para os nós. Dessa forma, os nós conectam-se com o mestre para reportar suas informações de registro, juntamente com a possibilidade de obter informações e se conectar com os demais nós presentes no grafo. Em adição, o Mestre também é responsável por informar mudanças nas informações de registro, o que possibilita a criação de conexões dinamicamente de acordo com o surgimento de novos nós.

Por fim, vale ressaltar que os nós realizam conexões diretamente entre si, de forma que o mestre simplesmente serve como uma tabela de consulta para os nós registrados na rede. Dessa forma, os nós conectam-se entre si por meio do sistema de tópicos e serviços citado anteriormente, garantindo que haja um desacoplamento entre nós, de forma garantir maior versatilidade para o sistema.

2.2 Representação de posição e orientação

2.2.1 Pose e movimento

Uma das formas mais comuns de representar a localização de um objeto em um determinado ambiente é pelo conceito de pose relativa, também referida apenas como pose. De acordo com a definição encontrada em (CORKE, 2023), pose é uma medida relativa composta por duas informações: translação e orientação em relação a um determinado referencial, as quais serão definidas futuramente nessa seção. Tal definição permite então descrever a configuração de um objeto no espaço, além de fornecer uma forma de mensurar

movimento de um determinado objeto, dada pela diferença entre as poses. A partir dessa premissa, é possível definir um movimento como uma transformação que tem como objetivo levar um objeto de uma pose inicial para uma pose final.

Adicionalmente, (CORKE, 2023) define que um movimento arbitrário qualquer que leva de uma pose inicial P_i para uma pose final P_f, representado pelo símbolo ${}^{P_i}\xi_{P_f}$, pode ser definido como uma junção de outros movimentos. Considerando quaisquer poses P_1, P_2 e P_3, é possível descrever o movimento ${}^{P_1}\xi_{P_3}$ como retratado pela equação 2.1.

$$ {}^{P_1}\xi_{P_3} = {}^{P_1}\xi_{P_2} \oplus {}^{P_2}\xi_{P_3} \tag{2.1} $$

No qual $\oplus$ é definido como o operador de *composição* de movimentos. Disso, postula-se que qualquer movimento pode ser então decomposto em um série de movimentos menores ou mais simples, aplicados em sucessão. Vale ressaltar que neste contexto, a ordem em que os movimentos ocorrem interfere na pose final do movimento, de forma que o operador $\oplus$ não é comutativo. Analogamente, um movimento sempre apresenta um movimento inverso equivalente, como abordado pela equação (2.2), em que $\oslash$ representa um movimento nulo, o qual não gera mudança na pose do objeto. Disso, (CORKE, 2023) define o operador inverso $\ominus$, como retratado na equação (2.3).

$$ {}^{P_1}\xi_{P_2} \oplus {}^{P_2}\xi_{P_1} = {}^{P_1}\xi_{P_1} = \oslash \tag{2.2} $$

$$ {}^{P_1}\xi_{P_2} = \ominus {}^{P_2}\xi_{P_1} \tag{2.3} $$

Desta equação, pode-se inferir o retratado pela equação (2.4), apresentada a seguir.

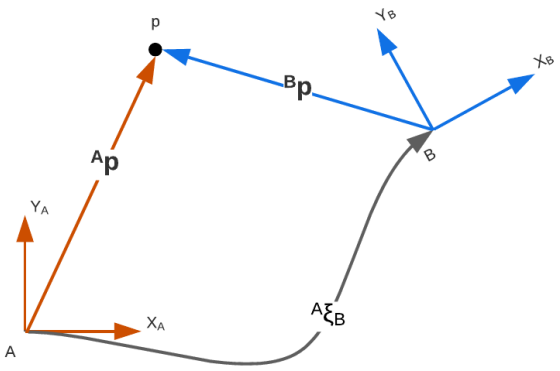
$$ {}^{P_1}\xi_{P_2} \ominus {}^{P_1}\xi_{P_2} = {}^{P_1}\xi_{P_2} \oplus {}^{P_2}\xi_{P_1} = \oslash \tag{2.4} $$

Com isso, é possível então definir os conceitos de translação e orientação citados anteriormente. Observando a figura 4, pode-se perceber a existência de dois sistemas de coordenadas distintos: A e B. Nesse contexto, a translação entre A e B é dada pela distância entre as origens, enquanto que a orientação entre A e B é descrita pelo ângulo entre cada um dos eixos desses sistemas de coordenadas. Dessa forma, a pose relativa entre A e B, representada pelo movimento ${}^{A}\xi_{B}$, permite que um ponto P qualquer representado em um sistema de coordenadas possa ser convertido para seu equivalente no outro sistema apenas aplicando a relação da equação (2.5), definida por (CORKE, 2023):

$$ {}^{A}p = {}^{A}\xi_{B} \cdot {}^{B}p \tag{2.5} $$

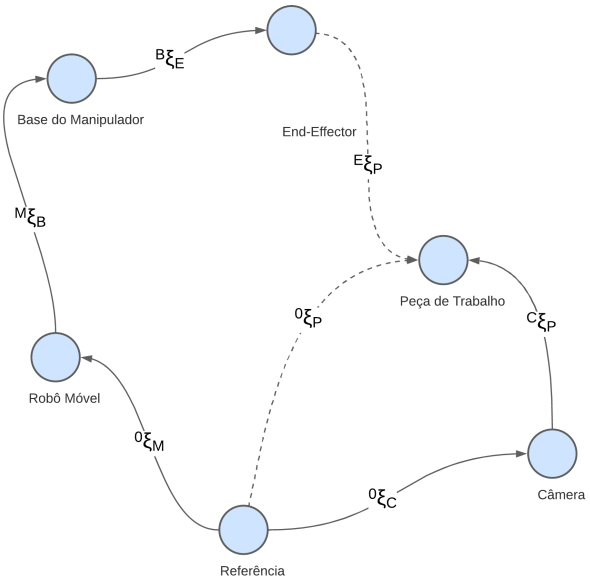
Nos quais ${}^{A}p$ e ${}^{B}p$ representam o ponto p descrito nos sistemas de coordenadas A e B, respectivamente, e o operador $\cdot$ representa a transformação que traz o ponto P de um sistema de coordenadas para outro.

Figura 4 – Representação de um ponto em dois sistemas de coordenadas



Fonte: Adaptado de (CORKE, 2023)

Figura 5 – Grafo de poses de um manipulador robótico



Fonte: Adaptado de (CORKE, 2023)

Consequentemente, é possível agora descrever relações espaciais entre objetos e inferir relações por meio das operações definidas na equação (2.1), e na equação (2.5). No exemplo dado pela figura 5, retirado de (CORKE, 2023), é necessário descobrir a pose relativa do *end-effector* do manipulador e a peça de trabalho, representada pelo movimento ${}^{E}\xi_{P}$. Nessa situação, é possível obter a pose desejada observando as poses que se têm conhecimento, marcadas pelas setas escuras no grafo da imagem, via inspeção. Do grafo, tem-se a figura 5.

$$ {}^{0}\xi_{M} \oplus {}^{M}\xi_{B} \oplus {}^{B}\xi_{E} \oplus {}^{E}\xi_{P} = {}^{0}\xi_{C} \oplus {}^{C}\xi_{P} \tag{2.6} $$

Isolando ${}^{E}\xi_{P}$ da equação (2.6), obtém-se ${}^{E}\xi_{P}$ em função das poses já conhecidas pelo sistema, como representado na equação (2.7).

$$ {}^{E}\xi_{P} = \ominus {}^{B}\xi_{E} \ominus {}^{M}\xi_{B} \ominus {}^{0}\xi_{M} \oplus {}^{0}\xi_{C} \oplus {}^{C}\xi_{P} \tag{2.7} $$

Finalmente, a partir do que foi apresentado anteriormente, é possível definir o movimento ξ como um grupo matemático, com os seguintes operadores apresentados na lista a seguir:

- $\oplus$ Operador de Composição
- $\ominus$ Operador Inverso
- $\oslash$ Elemento Identidade
- ${}^{A}p = {}^{A}\xi_{B} \cdot {}^{B}p$ Transformação de Sistema de Coordenadas do ponto

No entanto, os conceitos ao longo desta seção foram definidos apenas abstratamente, sem uma implementação desses operadores de forma concreta. Para isso, é necessário, primeiramente se definir como serão representadas as translações e orientações dentro do escopo da aplicação. As abordagens escolhidas para tal tarefa serão discutidas nas seções a seguir.

2.2.2 Representação da translação no espaço $\mathbb{R}^3$

Considerando que um ambiente 3D pode ser formulado pelo espaço $\mathbb{R}^3$, tem-se que a translação entre dois sistemas de coordenadas A e B pode ser modelada da seguinte forma:

$$ {}^{A}t_{B} = (t_x, t_y, t_z), {}^{A}t_{B} \in \mathbb{R}^3 \tag{2.8} $$

Onde ${}^{A}t_{B}$ representa a translação entre os sistemas de coordenadas A e B, respectivamente, e as coordenada t_x, t_y e t_z representam as distâncias entre os sistemas de coordenadas para cada eixo em relação ao referencial estipulado.

2.2.3 Representação da orientação por quatérnions unitários

Descobertos pelo matemático William Rowan Hamilton, vide (CORKE, 2023), os quatérnions são um conjunto de números definidos como uma extensão dos números complexos, com um componente real e três componentes imaginários como representado pela equação (2.9). Essa representação matemática é valiosa em aplicações que envolvem rotações tridimensionais, como em gráficos 3D, visão computacional e, especialmente, em evitar o problema conhecido como *Gimbal Lock*, fenômeno responsável pela redução graus de liberdade em orientações, podendo levar à perda de informações, ou seja, limitações na representação da orientação espacial. Quatérnions contornam tal problemática ao evitar singularidades pela sua continuidade de representação de rotações, especificamente.

$$q = a + bi + cj + dk,\ a, b, c, d \in \mathbb{R} \tag{2.9}$$

Os números complexos i, j e k, que podem ser compreendidos como versores, ou vetores unitários ortogonais, são dados pela relação apresentada pela equação (2.10).

$$i^2 = j^2 = k^2 = ijk = -1 \tag{2.10}$$

Nesse contexto, quatérnions possibilitam adição e substração, realizada elemento a elemento, além de multiplicação por um escalar. Assim como números complexos, os quatérnions também apresentam um conjugado, de formato tal que suas partes imaginárias possuam o sentido negativo, como retratado pela equação (2.11).

$$q^* = a - bi - cj - dk \tag{2.11}$$

O produto entre dois quatérnions q_1 e q_2 é então definido pela equação (2.12), na forma de produto matriz-vetor, conhecido como produto de Hamilton:

$$q_1 \circ q_2 = \begin{bmatrix} a_1 & -b_1 & -c_1 & -d_1 \\ b_1 & a_1 & d_1 & -c_1 \\ c_1 & d_1 & a_1 & b_1 \\ d_1 & c_1 & -b_1 & a_1 \end{bmatrix} \begin{bmatrix} a_2 \\ b_2 \\ c_2 \\ d_2 \end{bmatrix} \tag{2.12}$$

Adicionalmente, o produto interno entre dois quatérnions q_1 e q_2 é dado por pela equação (2.13). Complementarmente, a norma de um quatérnion é dada pela equação (2.14).

$$q_1 \cdot q_2 = a_1a_2 + b_1b_2 + c_1c_2 + d_1d_2 \tag{2.13}$$

$$\|q\| = \sqrt{q \cdot q} = \sqrt{a^2 + b^2 + c^2 + d^2} \tag{2.14}$$

Finalmente, (CORKE, 2023) então afirma que um quatérnion $\hat{q}$, de norma $\|\hat{q}\| = 1$ pode ser utilizado para representação de uma rotação θ sobre um vetor unitário $\hat{v}$, de forma que vale o apresentado pela equação (2.15).

$$\hat{q} = \cos\frac{\theta}{2} + \hat{v}_x \sin\frac{\theta}{2}i + \hat{v}_y \sin\frac{\theta}{2}j + \hat{v}_z \sin\frac{\theta}{2}k \tag{2.15}$$

Nesse contexto, a transformação de coordenadas de um vetor v por meio de um quatérnion unitário $\hat{q}$ é definida da seguinte forma, apresentada pela equação (2.16):

$$\hat{q} \cdot v = \hat{q} \circ \hat{q}_v \circ \hat{q}^*,\ onde\ \hat{q}_v = 0 + v_x i + v_y j + v_z k \tag{2.16}$$

Vale ressaltar que, para este tipo de representação de rotação, há ocorrência de mapeamento duplo, no qual os quatérnions $\hat{q}$ e $-\hat{q}$ representam a mesma orientação, de forma que é necessário impor a convenção escolhida na aplicação a fim de evitar ambiguidade.

2.2.4 Representação de pose por par vetor-quatérnion

Utilizando a definições apresentadas nas seções 2.2.1, 2.2.2 e 2.2.3, (CORKE, 2023) afirma que ξ pode então ser definido como um par ordenado $(t, \hat{q}) \in \mathbb{R}^3 \times S^3$, no quais os operadores definidos na seção 2.2.1 são mapeados da seguinte forma:

$$
\begin{aligned}
\xi_1 \oplus \xi_2 &\longmapsto (t_1 + \hat{q}_1 \cdot t_2, \hat{q}_1 \circ \hat{q}_2) \\
\ominus \xi &\longmapsto (-\hat{q}^* \cdot t, \hat{q}^*) \\
\oslash &\longmapsto (t_0, \hat{q}_0), onde\ t_0 = (0,0,0)\ e\ \hat{q}_0 = 1 + 0i + 0j + 0k \\
\xi \cdot v &\longmapsto \hat{q} \cdot v + t
\end{aligned}
$$

2.3 Câmeras

As câmeras, especificamente as digitais, possuem um método específico para captar informações do ambiente e perceberem o suficiente para recriar imagens que, a depender de diversos fatores relacionados à sua construção, influenciam diretamente no resultado. Como descrito por (STEPHEN SAGERS, 2010), durante a captura da imagem, o obturador, em muitos casos eletrônico, age para controlar o tempo de exposição da luz ao sensor sensível à luz, geralmente do tipo CCD (*Charge-Coupled Device*) ou CMOS (*Complementary Metal-Oxide-Semiconductor*). Este sensor converte a luz em um sinal elétrico, o qual é processado por um processador de imagem incorporado, que pode ser armazenado, e a informação pode ser carregada para um computador, por exemplo.

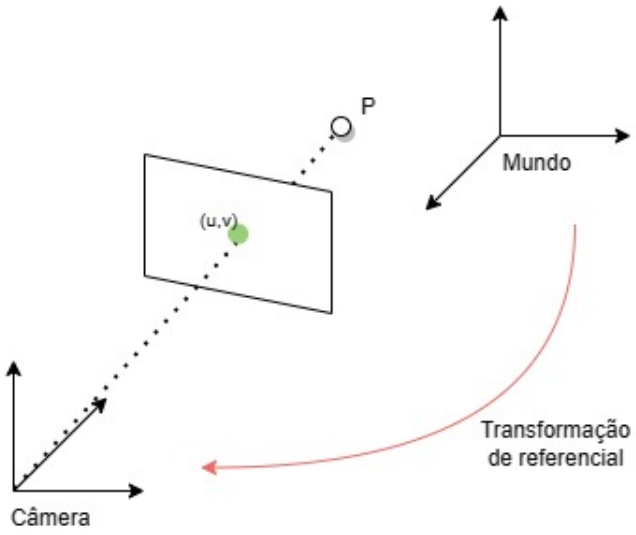
A capacidade de capturar imagens em tempo real e processá-las para extrair informações relevantes, como a posição e orientação de objetos, é crucial para a manipulação de objetos e interação eficiente com o ambiente em sistemas robóticos. Câmeras específicas para robótica são projetadas levando em consideração as necessidades específicas desses sistemas. Elas são otimizadas para operar em ambientes desafiadores, como baixa luminosidade, variações de temperatura e condições adversas, garantindo desempenho consistente e confiável. Além disso, essas câmeras são frequentemente compactas e leves, adequadas para a integração em estruturas robóticas sem comprometer a mobilidade ou a eficiência do robô. Um exemplo são as câmeras de obturação global, ou *global shutter*, capazes de processar toda a imagem de forma paralela, evitando efeitos de distorção na imagem. Em câmeras de obturador rolante, por exemplo, se a imagem retratar um objeto em movimento, é possível que para diferentes partes da imagem, o resultado retrate instantes diferentes na realidade do objeto, já que o obturador ter um tempo de movimento. Para obturadores globais, a coleta é feita no mesmo instante para todos os pixeis.

A utilização de câmeras genéricas ou inadequadas pode levar a uma série de problemas. A precisão na identificação de objetos e na estimativa de poses pode ser comprometida, prejudicando a eficácia do robô em suas tarefas. Câmeras com latência inadequada podem resultar em atrasos críticos na tomada de decisões, afetando a resposta em tempo real necessária para muitas aplicações robóticas. Além disso, a falta de robustez em condições desafiadoras pode levar a falhas frequentes, aumentando a probabilidade de erros na execução de tarefas. Câmeras de baixa resolução podem limitar a capacidade do robô de discernir detalhes importantes no ambiente, impactando negativamente a qualidade das informações visuais capturadas.

2.3.1 Observação e parâmetros

As câmeras possuem uma forma de transformar os dados obtidos em pixeis, e esta forma depende de parâmetros que variam de câmera para câmera. A equação (2.17) descreve como a coordenada $\tilde{p}$ em pixeis, vide (CORKE, 2023), de um ponto arbitrário no espaço p pode ser representada de acordo com parâmetros ditos intrínsecos e extrínsecos. O cenário pode ser observado pela figura 6.

Figura 6 – Ilustração de funcionamento de câmera com eixos



Fonte: Adaptado de Solve PnP

$$\tilde{p} = \underbrace{\begin{bmatrix} f/\rho_w & 0 & u_0 \\ 0 & f/\rho_h & v_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}}_{\text{intrínseco}} \underbrace{Tp}_{\text{extrínseco}} \tag{2.17}$$

Os parâmetros intrínsecos estão relacionados à qualidades da composição do mecanismo da câmera e envolvem, dentre outros fatores, ρ_w que representa a largura dos pixeis, ρ_h, que representa a altura dos pixeis, e f um fator de escala entre coordenadas de um plano e uma imagem de retina, sendo $f = 1$ o caso para câmeras normalizadas, ou de coordenadas imagem-plano canônicas. Para os parâmetros extrínsecos, têm-se a própria posição do ponto em relação a um referencial global, ou seja, do mundo, e uma matriz de transformação T para representar esse ponto sob o referencial da própria câmera, como ilustrado pela figura 6. A matriz de transformação T toma a forma, portanto, de uma transformação linear, capaz de transladar e rotacionar o ponto no referencial global para o plano de perspectiva da câmera, de modo como segue a equação (2.18).

$$T = \begin{bmatrix} r_{aa} & r_{ab} & r_{ac} & t_x \\ r_{ba} & r_{bb} & r_{bc} & t_y \\ r_{ca} & r_{cb} & r_{cc} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.18}$$

Por vezes, a obtenção destes parâmetros não é simplificada por fabricantes, principalmente pelo fato de que os mesmos costumam variar bastante por depender de muitos fatores físicos atrelados à montagem da estrutura e seus componentes, bem como no uso recorrente do produto. Portanto, se faz necessário o uso da calibração da câmera através da aquisição de múltiplas imagens com a mesma referência, usualmente de um tabuleiro, em ângulos e posições diferentes, de modo que seja possível identificar distorções, incluindo distorções geométricas, entre o que a referência deve ter e o que é obtido pela câmera. Esse processo costuma ocorrer com correspondência de pontos de interesse na referência, seguido do uso de métodos de otimização variados para estimar os parâmetros destacados na Equação [2.17].

A modelagem das distorções geométricas pode ser realizada por meio de funções polinomiais. Uma forma comum é a utilização do modelo de distorção radial, onde a coordenada $\tilde{p}$ distorcida é obtida pela seguinte equação:

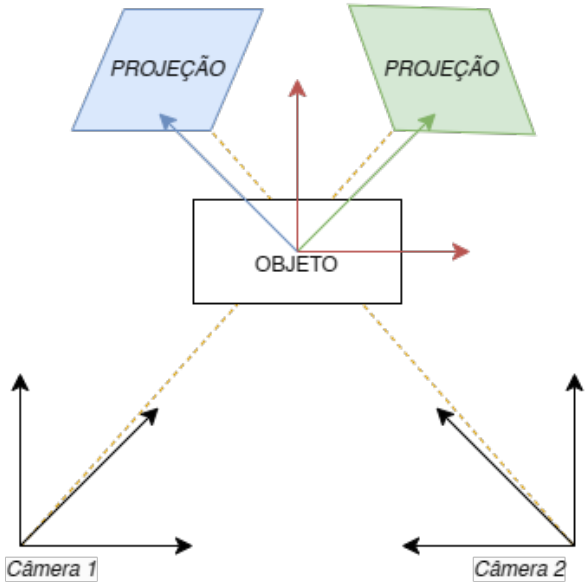
$$\tilde{p}_{\text{distorcido}} = \tilde{p}\left(1 + k_1 r^2 + k_2 r^4 + ...\right) \tag{2.19}$$

Onde r representa a distância radial do ponto distorcido ao ponto principal da imagem. Os coeficientes $k_1, k_2, ...$ são parâmetros que caracterizam as características específicas da distorção radial.

Há também a necessidade de se abordar a problemática de ambiguidade de poses para a câmera, dada uma observação, ou projeção de um objeto num plano 2D, com simetria planar. Utilizando-se novamente da figura 6, percebe-se que, caso duas câmeras se posicionem simetricamente refletidas e igualmente espaçadas de um ponto observado, e caso não hajam correspondências de pontos suficientes, ambas obterão a mesma projeção, de modo que

através dos dados coletados não se pode definir absolutamente a pose relativa da câmera em relação ao marcador. Essa problemática é melhor ilustrada pela figura [7.

Figura 7 – Ilustração de problema de ambiguidade de pose de câmera



2.3.2 Câmera STH-MDCS3-VAR COLOR

Um dos modelos de câmera presentes no LARA é a STH-MDCS3-VAR COLOR, da Videre Design. Esta câmera estéreo, ou seja, trabalha com duas câmeras por coleta, de obturador global é ideal para coleta de dados de alta sensibilidade com baixo ruído. Através do protocolo *firewire*, pode coletar dados a 30 Hz de frequência com resolução de 640 x 480 pixeis, a 15 Hz com 1024 x 768 pixeis, ou 7.5 Hz com 1280 x 960 pixeis. Possui controle de exposição e ganho de imagem automático. Este modelo precisa de câmeras conectadas em uma interface *firewire* instalada em computadores *desktop*.

Figura 8 – Câmera estéreo STH-MDCS3-VAR

Fonte: (STH-MDCS3-VAR(X)(-C)..., 2007)

2.3.3 Câmera Kinect

Outro dos modelos de câmera presentes no laboratório é o Microsoft Kinect Xbox 360. Projetado para aplicações de video-games com a plataforma Xbox, também se mostrou como uma poderosa ferramenta, dadas as suas capacidades de captação de dados de ambientes em 2D com múltiplas câmeras RGB e câmera infravermelho, e 3D, com nuvem de pontos. É capaz de gerar imagens de 640 x 480 pixeis com profundidade máxima de 4.5 metros, tudo a 30 quadros por segundo. Esta câmera precisa ser alimentada externamente com fonte própria, e se conecta ao computador por entrada USB.

Figura 9 – Câmera Kinect Xbox

Fonte: Fickr Xbox Kinect

2.3.4 Câmera C270 HD Webcam Logitech

Um modelo de câmera capaz de gerar imagens em HD, 1280 x 720 pixeis a uma taxa de 30 quadros por segundo, com correção automática de luz, facilitando a visualização em ambientes de pior luminosidade. Também utiliza entrada USB para comunicação, porém não exige alimentação externa.

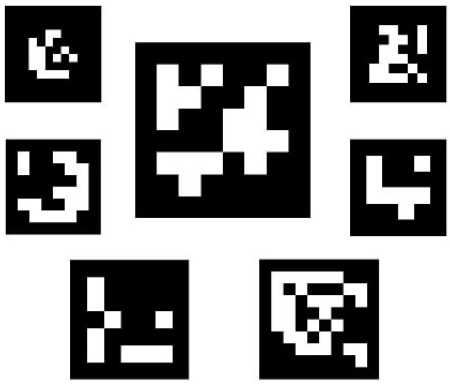
Figura 10 – Câmera Logitech C270 HD

Fonte: https://www.logitechstore.com.br/camera-webcam-hd-logitech-c270

2.4 Marcadores fiduciais ArUco

Segundo (KALAITZAKIS et al., 2021), a utilização de marcadores fiduciais é uma prática comum em aplicações de rastreamento e localização, servindo como complemento ou até mesmo substituindo a utilização de algoritmos de visão computacional para esta finalidade, como SLAM, por exemplo. Um marcador fiducial pode ser definido como um referencial a ser colocado em um ambiente com o propósito de ser facilmente identificado e rastreado, apresentando um padrão altamente reconhecível com características visuais fortes e bem definidas, promovendo um forte contraste em relação ao ambiente, como relatam (KALAITZAKIS et al., 2021) e (BORGES, 2021). Adicionalmente, muitos deles empregam um sistema de codificação específico a fim de identificar unicamente o marcador e prevenir erros de detecção. Nesse contexto, (KALAITZAKIS et al., 2021) cita alguns do modelos de marcadores mais utilizados amplamente e entre eles está o modelo ArUco.

Figura 11 – Exemplos de marcadores ArUco



Fonte: ArUco OpenCV

O modelo de marcadores ArUco, definido em (GARRIDO-JURADO et al., 2014) e (MUÑOZ-SALINAS et al., 2017), estabelece o marcador como uma imagem quadrada binária, especificamente preto e branco, atrelada a um identificador único do marcador, como pode ser observada nos exemplos da figura 11. Nesse contexto, a geometria do marcador fornece pontos bem definidos, dado pelos quatro vértices do marcador, que são utilizados na estimação da pose do marcador em relação à câmera.

Disso, (CORKE, 2023) descreve então que a estimação dessa pose é definida então como um problema *Perspective-n-Point*, ou PnP, no qual deseja-se estimar a pose relativa do marcador em três dimensões por meio da comparação de um conjunto n de ponto de referência na imagem, dada uma câmera previamente calibrada e de parâmetros intrínsecos conhecidos. Nesse contexto, (MUÑOZ-SALINAS et al., 2017) estipula que este problema pode então ser escrito na forma de um problema de otimização não-linear do erro de re-projeção dos vértices do marcador, de forma que a minimização utiliza o algoritmo de *Levenberg Marquardt*. Ademais, a biblioteca de *software* cuja aplicação é mais utilizada para solução

de problemas PnP é a OpenCV, e diversos métodos de solução são implementados, como o retratado em (LEPETIT; MORENO-NOGUER; FUA, 2009). De modo mais objetivo, a solução do problema PnP envolve encontrar a matriz de rotação e translação, retratada pela equação (2.18), que transforma um ponto expresso no mundo, em um plano 2D na perspectiva da respectiva câmera, de forma a minimizar o erro de re-projeção.

2.5 Variáveis aleatórias

Muitos fenômenos que ocorrem na natureza são aleatórios ou, de certa forma, incertos. Surge, portanto, a necessidade da modelagem matemática do fenômeno relacionado à essa incerteza, e para tal, existem as variáveis aleatórias. Nesse contexto, uma função $f(\omega)$ definida em Ω, tal que $\omega \in \Omega$, é denominada uma variável aleatória se, para cada número real x, também chamado de realização da variável aleatória, vale que $f(\omega) \leq x$, de modo que a probabilidade do conjunto de ω é definida. Também, diz-se que o conjunto de ω é um evento. A função $F_f(x)$ é dita função de distribuição da variável aleatória f, e vale a equação (2.20).

$$F_f(x) \triangleq Pr\{f(\omega) \leq x\} \; \forall \; x \in \mathbb{R} \tag{2.20}$$

Para os cenários em que $f(\omega)$ for uma função contínua, pode-se escrever F_f em função da denominada função de densidade de probabilidade p_f, como definido pela equação (2.21). Essa equação denota que a probabilidade de uma variável estar em um intervalo é tal qual o numericamente representado pela integral sob a curva da função de densidade de probabilidade entre o referido intervalo de interesse.

$$F_f(x) = \int_{-\infty}^{x} p_f(\xi) d\xi \mid -\infty \leq x \leq \infty \tag{2.21}$$

Para fins estatísticos, vale a definição do momentos das variáveis aleatórias. A função geral para o enésimo momento em torno da origem de f é dada pela equação (2.22). Os momentos são relevantes pois descrevem informações pertinentes sobre a forma, a centralidade e a variabilidade de uma distribuição aleatória. É definido também que o primeiro momento, com $n = 1$, é chamado de média, ou esperança, representando uma tendência de um conjunto de dados. O segundo momento, com $n = 2$, é a variância, ou simplesmente $var\{x\}$, e representa a dispersão de um conjunto de dados da média.

$$\mathbb{E}\{(f - \mathbb{f})^n\} \triangleq \int (f - \mathbb{f})^n p_f(x) dx \tag{2.22}$$

2.5.1 Distribuições Gaussianas

De grande importância na ciência e na engenharia, a função de distribuição Gaussiana, cujas variáveis aleatórias são ditas de distribuição normal, é definida pela equação (2.23). Para essa distribuição, percebe-se a constância de dois parâmetros. $m = \mathbb{E}\{f\}$ e $\sigma^2 = var\{f\}$, portanto, pode-se definir uma função de distribuição Gaussiana f apenas com estes parâmetros a utilizar da seguinte notação: $f \sim N\{m,\sigma^2\}$.

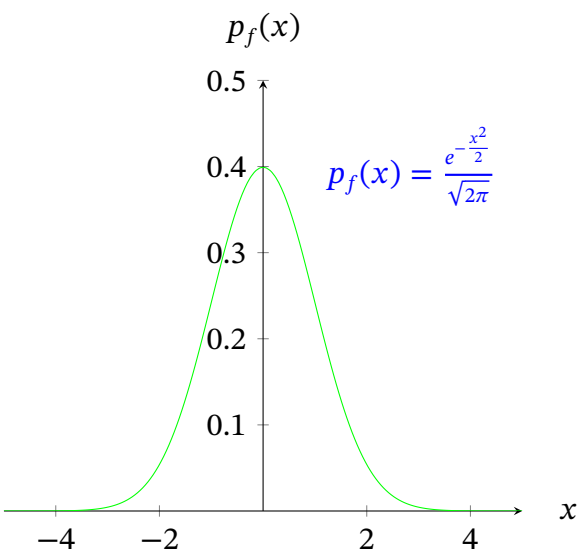
$$p_f(x|m,\sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-m)^2}{2\sigma^2}} \tag{2.23}$$

Muitos dos fenômenos encontrados na natureza que possuem certo caráter aleatório podem ser representados matematicamente pela função de distribuição normal. Essa função possui diversas características de interesse para modelagem, como o fato de seu domínio se estender do menos infinito ao infinito através de uma curva assintótica; A área compreendia pela curva corresponde a 1, ou seja, 100% de probabilidade; Possui dois pontos de inflexão simétricos à média, denominados pontos de desvio padrão; E a área sob a curva indica probabilidade da ocorrência de x.

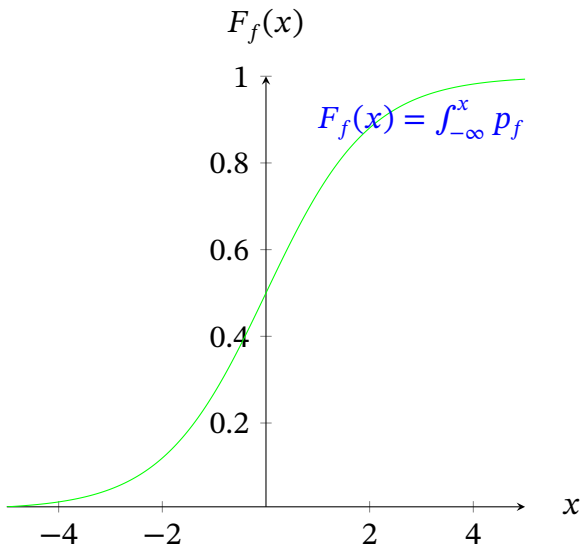
Para fins de ilustração, a figura 12a destaca um cenário específico para a função de densidade da distribuição normal. Percebe-se que seu caráter visual simplifica a visualização de que a média dos pontos se encontra centrada no ponto em que $x = 0$. Para a função cumulativa, resultado da integral da função de densidade, ou seja, F_f, o gráfico é representado pela figura 12b, e ilustra, como o seu valor tende e 0 a 1, como representante da probabilidade do evento relacionado a x.

Figura 12 – Distribuição normal com média nula e variância unitária

(a) Gráfico de função de densidade



(b) Gráfico de função cumulativa



2.6 Processos estocásticos

Supondo um cenário que compõe um sistema não conhecido, ao menos completamente, espera-se que para cada condição inicial definida unida à certas condições impostas sobre o mesmo sistema, o levem para uma condição não conhecida. Portanto, pode-se modelar tal sistema como uma função arbitrária $F(x_t(\omega), t) \mid F : \mathbb{R} \to \mathbb{R}$ sendo $x_t(\omega)$ uma variável aleatória dependente de $\omega \in \Omega$, um parâmetro aleatório pertencente a um conjunto de eventos, e também do instante de tempo $t \in \mathbb{R} \mid t_i \leq t \leq t_f$, sendo t_i um instante de tempo inicial e t_f um final. Para cada combinação de instante arbitrário e parâmetro aleatório, denomina-se a realização do processo a condição $x_t(\omega)$, ou seja, a realização é o valor específico tomado pela variável aleatória em uma observação específica. Essa realização pode ser uma função de amostragem, caso o conjunto dos parâmetros seja contínuo, ou uma sequência de amostragem, caso seja discreto. Aqui, destaca-se o fato de que para um mesmo instante e condições de ambiente, ainda é possível que a informação obtida de uma variável aleatória varie, tendo em vista a ausência de conhecimento sobre a realidade a ser modelada e os efeitos aleatórios que nela impactam.

Para cada situação, dependente do instante e do parâmetro aleatório, a função F retornará um valor diferente, de modo que presume-se uma probabilidade de ocorrer cada um dos cenários previstos pelos possíveis valores assumidos por ω, e.g., para uma aposta realizada entre indivíduos dependente apenas do resultado do lançamento de uma moeda e da sua face superior após o lançamento, aqui abstraído como ω, presume-se uma probabilidade complementar para cada um dos cenários, $Pr(\eta(cara)) = 0.5$, $Pr(\eta(coroa)) = 0.5$, que compõem uma variável aleatória $\eta(\omega)$, o que resulta em ações diferentes a serem tomadas pelos indivíduos que impactarão o estado do sistema, aqui abstraído pela realização x_t.

É evidente que se o objetivo for a compreensão do comportamento do sistema a longo prazo, se faz necessária uma análise que considere uma condição inicial como $x_0 = 0$, a ser afetada por n instantes, ou atuações por $x_t(\omega)$. Esse modelo matemático toma a forma de uma equação de diferenças se o efeito da variável aleatória do sistema for um acréscimo ou decréscimo do estado, já que o ato de jogar as moedas para cima, mesmo que considerado um evento instantâneo, ocorre serializadamente, sendo apenas uma moeda jogada por vez, denotando x_t como uma série de parâmetros discretizados, o que permite modelar $x_n = \sum_{n=t_i}^{n} \eta_t(\omega)$. Alternativamente, x_t pode tomar a forma de uma sequência aleatória caso η seja uma variável independente e contínua, mantendo o atual espaço de estados contínuo. Nesse novo cenário, para cada instante de tempo n, haveria uma variável aleatória contínua x_t. Se os parâmetros aleatórios forem contínuos e o espaço de estados do sistema for discreto, x_t terá a forma de uma sequência de parâmetros contínuos. Contudo, para tanto o espaço de estados quanto o conjunto de parâmetros contínuos, têm-se que x_t é denominada uma função de processo aleatório (estocástico).

2.6.1 Sistema dinâmico estocástico

Muitos dos processos dinâmicos trabalhados na engenharia são complexos a ponto de que sua modelagem analítica não satisfaz necessariamente certa quantidade de requisitos que caracterizariam tal modelagem acurada ou precisa o suficiente. Essa discrepância entre modelo e realidade por vezes é tratada como distúrbios ou ruído inserido no sistema, de modo que é possível, então, modelar certa parte do desconhecimento do sistema através de variáveis aleatórias. Muitos desses sistemas possuem estados que podem ser representados por vetores de dimensão finita, o que permite que o sistema em si possa ser representado por uma equação diferencial vetorial.

2.6.1.1 Equações diferenciais estocásticas

Seja x_k um vetor de estados de dimensão n no instante t_k, e w_k um vetor de dimensão m de ruído aleatório, tal que $m \leq n$, pode-se definir a equação (2.24) de modo que f se torna uma equação diferencial vetorial estocástica, que representa um sistema dinâmico estocástico discreto. Aqui, f é uma função presumidamente contínua e diferenciável em todos os seus pontos, para todos os seus argumentos.

$$x_{k+1} = f(x_k, w_{k+1}, t_k) \mid k \in \{0,1,2,3,...\} \tag{2.24}$$

Como consequência da existência de w_k, a solução do sistema não se define apenas com o conhecimento de x_k, mas também se faz necessário o conhecimento da densidade de probabilidade de x_k, ou simplesmente $p(x_k)$. Com esta, é possível computar a esperança e a variância do estado do sistema, bem como permitir a análise estatística do comportamento de x_k. Não obstante, um dos maiores objetivos na análise de sistemas estocásticos é a definição da função de densidade de probabilidade da realização.

Assumindo que w_k seja uma variável de ruído branco gaussiano, que independe de uma condição inicial x_0 e que cada termo é independente entre si, então, como apresentado por (JAZWINSKI, 1970), o estado seguinte do sistema, x_{k+1}, depende apenas de w_{k+1} e do estado atual x_k, o que torna a solução da equação (2.24) uma cadeia de Markov, ou seja, uma cadeia cujo estado posterior depende apenas do estado atual e não de como o sistema chegou ao estado atual. Para um cenário cujas dimensões dos vetores de estado x_k e de ruído w_k possuam a mesma dimensão, ou seja, $n = m$, pode-se assumir uma solução para a equação (2.24) para w_{k+1} se a função f possuir uma inversa. Como consequência, seguiria a definição da função de transição de densidade de probabilidade denotado pela equação (2.25) obtida de (JAZWINSKI, 1970, Teorema 2.7).

$$p(x_{k+1}|x_k) = p_{w_{k+1}}(f^{-1}(x_k, x_{k+1}, t_k)) \left|\left| \frac{\partial f^{-1}}{\partial x_{k+1}} \right|\right| \tag{2.25}$$

Pelo fato do sistema possuir uma variável de ruído branco Gaussiano, segue da equação (2.25) o que está determinado pela equação (2.26), também chamada de Equação de Chapman-Kolmogorov, como apresentado em (JAZWINSKI, 1970, Teorema 2.8). Essa equação aborda o fato de que para que o sistema alcance a realização de x_{k+1}, deve-se passar pelo estado anterior x_k, contudo, há múltiplos casos possíveis para o estado anterior que leva o sistema para tal realização. A problemática, portanto, envolve a avaliação da probabilidade de todos estes casos. Assim, sabendo o valor inicial para a função de probabilidade do sistema aleatório, pode-se calcular seus efeitos subsequentes em série, ao definir a chamada lei de propagação de estados adequada, que utiliza de princípios de modelagem em equações diferenciais para se obter tais transições de estados.

$$p(x_{k+1}) = \int p(x_{k+1}|x_k)p(x_k)dx_k \tag{2.26}$$

Pertinente destacar que para uma variável inicial constante como $x_0 = c$, sendo c uma constante arbitrária, a sua função de densidade de probabilidade será representada por um Delta de Dirac como $p(x_0) = \delta(x_0 - c)$, denotando uma probabilidade máxima e definindo o início da cadeia de Markov. Assumindo que $w_{k+1} \sim N(0, Q_{k+1}) \mid Q_{k+1} > 0$, e portanto vale a equação (2.23), então, segue a relação detalhada pela equação (2.27). Neste cenário, define-se, a partir de termos já obtidos do instante anterior, a função de probabilidade da variável aleatória, responsável pelo termo não determinístico da equação (2.24).

$$p_{w_{k+1}}(f^{-1}(x_l, x_{k+1}, t_k)) = \frac{1}{(2\pi)^{n/2}\sqrt{|Q_{k+1}|}} \cdot e^{-\frac{1}{2}(f^{-1}(x_l,x_{k+1},t_k))^T Q_{k+1}^{-1}(f^{-1}(x_l,x_{k+1},t_k))} \tag{2.27}$$

É pertinente também a análise para casos em que a matriz de covariâncias não possua posto completo, de modo que exista alguma dependência linear entre as variáveis. Para o cenário no qual Q_{k+1} é uma matriz semi-definida positiva, de posto r, então existe uma matriz não singular A_{k+1} de dimensões $n \times n$, tal que vale o definido pela equação (2.28). Aqui, a matriz A é uma transformação linear que diagonaliza em blocos a covariância de w_k.

$$A_{k+1}Q_{k+1}A_{k+1}^T = \begin{bmatrix} I_{r\times r} & 0 \\ 0 & 0 \end{bmatrix} \tag{2.28}$$

Ao se definir a variável auxiliar $v_{k+1} = A_{k+1}w_{k+1}$, é possível separar os termos dependentes da variável aleatória e independentes através de algumas manipulações. Primeiro, define-se pela equação (2.29), e equação (2.30), a transposta da variável auxiliar e a inversa da matriz A, sendo a primeira segmentada pelos primeiros r elementos em termos de (1) e (2). Essa nova caracterização é pertinente tendo em vista certas facilidades de manipulação e de definição de propriedades.

$$v_{k+1}^{T} = \overset{(1\times r)}{[v_{k+1}^{(1)T}, v_{k+1}^{(2)T}]} \tag{2.29}$$

$$A_{k+1}^{-1} = \overset{(n\times r)}{[B_{k+1}, C_{k+1}]} \tag{2.30}$$

Seguindo dos resultados da equação (2.28), $v_{k+1}^{(1)} \sim N(0, I_{r\times r})$, portanto, pode-se dizer que v_k terá uma distribuição multivariada normal com média zero e covariância igual à identidade, já que sua segunda parte seria tal que $v_{k+1}^{(2)} = 0\ wp\ 1$, ou seja, essa equação é válida com probabilidade 1, pela variância nula. Por consequência, vale o definido pela equação (2.31), ao se aplicar a transformação de forma inversa, sem a necessidade da parte segmentada, já que o termo equivalente de variância para $v_{k+1}^{(2)}$ seria anulado pela transformação.

$$w_{k+1} = B_{k+1} v_{k+1}^{(1)} \tag{2.31}$$

Portanto, ao substituir a equação (2.31) na equação (2.24), têm-se a formulação de um problema de ruído de dimensão abaixo do da variável de estado, com a matriz de covariância do ruído sendo positiva definida, o que seria um caso especial da análise do sistema para caso $m < n$, que implicaria na necessidade de uma divisão do vetor de estados x_{k+1} como se segue pela equação (2.32), e equação (2.33).

$$\overset{(1)}{x}_{k+1} = \overset{(1)}{f}(x_k, w_{k+1}, t_k) \tag{2.32}$$

$$\overset{(2)}{x}_{k+1} = \overset{(2)}{f}(x_k, \overset{(1)}{x}_{k+1}, t_k) \tag{2.33}$$

Para tal situação, em que $\partial f^{(1)}/\partial w_{k+1}$ é não singular e de posto m, então segue que o sistema pode ser resolvido para $p(x_{k+1})$ pela equação (2.34), de acordo com (JAZWINSKI, 1970, Teorema 2.7).

$$p(x_{k+1}|x_k) = p(x_{k+1}^{(1)}, x_{k+1}^{(2)}|x_k) = p(x_{k+1}^{(2)}, x_{k+1}^{(1)}|x_k) p(x_{k+1}^{(1)}|x_k) \tag{2.34}$$

Como $f^{(1)}$ é invertível, pode-se realizar o processo análogo ao realizado para obter a equação (2.27), o que resultará na equação (2.35). Este resultado, fazendo uso da função Delta de Dirac, denota uma probabilidade tão alta quanto possível, do valor obtido $x_{k+1}^{(2)}$ ser igual ao valor calculado através de $f^{(2)}$. Portanto, um comportamento majoritariamente determinístico dentro deste contexto já que a probabilidade, sendo a integral da função Delta de Dirac, retornará o valor de 1. Tal processo determina completamente a densidade de

probabilidade de x_{k+1} para as condições definidas em x_k.

$$p(x_{k+1}^{(2)}, x_{k+1}^{(1)}|x_k) = \delta(x_{k+1}^{(2)} - f^{(2)}) \tag{2.35}$$

2.6.1.2 Modelagem de sistema estocástico

Até então, apresentou-se como correlacionar variáveis aleatórias e suas probabilidades, assumindo uma função f, contudo, surge a necessidade de avaliação do avanço de um sistema conforme um modelo no espaço de estados predefinido. Para um sistema linear discretizado arbitrário que possa ser modelado conforme a equação (2.36), têm-se $\bar{\Phi}$, a matriz de transição de estados de dimensão $n \times n$, $\bar{\Gamma}$, uma matriz de dimensão $n \times m$, e $\bar{w}$ de dimensão $m \times 1$.

$$x_{k+1} = \Phi_{k+1,k} x_k + \bar{\Gamma}_{k+1} \bar{w}_{k+1} \tag{2.36}$$

Considerando que o segundo termo desta equação possua uma matriz de covariância singular $\bar{\Gamma}_{k+1} \bar{Q}_{k+1} \bar{\Gamma}_{k+1}^T$, ou seja, cujo determinante é zero e portanto a torna não invertível e diagonalizada, analogamente a como abordado pela equação (2.27), vale a igualdade da equação (2.37). Contudo, deve-se notar que a dimensão de w_k, r é tal que $r \leq m$ e que a dimensão de Φ se torna $n \times r$. Realizando adaptações algébricas, pode-se encontrar uma matriz T que mapeia x_k com suas singularidades para um vetor y que não contenha singularidades.

$$\Gamma_{k+1} w_{k+1} = \bar{\Gamma}_{k+1} \bar{w}_{k+1} \tag{2.37}$$

Contudo, para o caso desta transformação não ser necessária, e portanto, o caso mais genérico, com as matrizes não singulares, segue a equação (2.38).

$$x_{k+1} = \Phi_{k+1,k} x_k + \Gamma_{k+1} w_{k+1} \tag{2.38}$$

Assumindo o sistema como discreto, bem como assumindo uma variável w_k como Gaussiana de ruído branco, média 0 e com matriz de covariâncias igual a Q_k, pode-se definir e média e a covariância do sistema como definido pela equação (2.39), e equação (2.40) a partir da equação (2.26).

$$\hat{x}_k = \mathbb{E}(x_k) \tag{2.39}$$

$$P_k = \mathbb{E}\{(x_k - \hat{x}_k)(x_k - \hat{x}_k)^T\} \tag{2.40}$$

Tomando a esperança, ou média, da equação (2.38)], conclui-se que $\hat{x}_{k+1} = \Phi_{k+1} \hat{x}_k$. Tomando o resto da equação (2.38) por esta, chega-se à equação de diferenças equação (2.41).

$$x_{k+1} - \hat{x}_k = \Phi_{k+1,k}(x_k - \hat{x}_k) + \Gamma_{k+1} w_{k+1} \tag{2.41}$$

Elevando esta expressão ao quadrado e operando o primeiro momento em ambos os lados, obtêm-se a variância, que pode ser observada pela equação (2.42).

$$P_{k+1} = \Phi_{k+1,k} P_k {\Phi^T}_{k+1,k} + \Gamma_{k+1,k} Q_{k+1} {\Gamma^T}_{k+1,k} \tag{2.42}$$

2.7 Filtragem Estocástica

O processo de filtragem de dados objetiva obter, ou estimar, um valor para o estado do sistema dado um certo número de observações e de informações relacionadas ao sistema em momentos anteriores. Este é um problema que data de séculos atrás e envolveu diversos matemáticos e aplicações diferentes. O início das tentativas de se realizar um processo que envolvesse a estimação e tratamento de dados que visasse minimizar funções de erros começa com Galileu em 1632, vide (KAILATH, 1974), e é tratado futuramente por Euler, Lagrange, Laplace, Bernoulli, entre outros. Tempos depois, destaca-se o método dos mínimos quadrados, em 1975, por Gauss, uma ferramenta precursora do comportamento de muitos filtros a serem desenvolvidos desde então. Contudo, a abordagem do mesmo não envolveu uma análise estatística probabilística mas sim uma análise de optimização pura. A utilização de técnicas estocásticas, ou seja, do uso da probabilidade como forma de colaborar na solução dos problemas de estimação e filtragem começaram com o trabalho de, dentre vários outros, Norbert Wiener, que propôs o conceito de um filtro linear no domínio da frequência cujo principal objetivo era a remoção de ruídos de sinais, através do que hoje é conhecido como filtro de Wiener, que minimiza o erro médio quadrático entre o sinal original e o sinal filtrado. Seus trabalhos foram posteriormente utilizados por outros, em especial Rudolf Kalman, que utiliza da teoria de probabilidade Bayesiana para a criação de um filtro mais adaptativo e robusto.

Surge na análise, portanto, a problemática da necessidade de estimar um estado a partir de uma observação. Considerando um sistema cujo formato é retratado pela equação (2.38), para variáveis de ruído branco Gaussiano tais que $w_k \sim N(0,Q_k)$, com um valor inicial pré-definido x_0 independente de w, portanto, para uma sequência de dados do processo gerados tomando a forma de uma cadeia de Markov, assume-se a existência de um vetor de observações ruidosas e, retratando um processo de medição discretizado y_k, vale a equação (2.43).

$$y_k = h(x_k, t_k) + v_k \mid k \in \{0,1,2,3,...\} \tag{2.43}$$

Aqui, h é uma função vetorial de dimensão m denominada matriz de medição ou observação e v_k é uma sequência gaussiana vetorial, também de dimensão m, tal que $v_k \sim N(0, R_k)$, com $R_k > 0$, análogo ao definido para as variáveis de estado do sistema na seção 2.6.1.1. Também define-se Y_l o conjunto de l observações realizadas.

Para uma sequência de realizações de observações, o problema de se encontrar x_k com base em Y_l é denominado estimação discreta, já que o sistema possui observações que ocorrem em instantes discretizados. Caso $k < l$, ou seja, caso hajam mais observações do que estados definidos, o problema é denominado suavização discreta. Caso $k = l$, ou seja, caso haja o mesmo número de observações que se têm de estados, o problema é denominado filtragem discreta, e caso $k > l$, o problema é denominado predição discreta. Portanto, resolvendo o mesmo problema de tempo real, pode-se, a depender do contexto do sistema, aprimorar dados estimados com base em um acúmulo de observações ou simplesmente realizar predições.

Assumindo a linearidade, pode-se determinar que $h(x_k, t_k) = M(t_k)x_k$ com $M(t_k)$ possuindo as dimensões de $m \times n$. Nesse contexto, um problema de modelagem do sistema com base nas matrizes, ou funções, M, Γ, Q, R, e F, poderia ser responsável por realizar a filtragem e resolver o problema da estimação discreta, ou até contínuo para além das demonstrações aqui apresentadas. É importante ressaltar, no entanto, que para esta modelagem de sistema, e portanto para suas soluções, as variáveis x_k, w_k e v_k são independentes, ou seja, vale que $\mathbb{E}\{w_k v_k^T\} = 0$, $\mathbb{E}\{w_k x_k^T\} = 0$, e $\mathbb{E}\{v_k x_k^T\} = 0$.

Diante dessa problemática, se faz necessária a escolha de um critério que auxilie na definição do modelo do filtro, portanto, um critério que permita a seleção, dentre diversas possíveis soluções para $\hat{x}_k$, a estimação do estado do sistema. Este critério, ao ser minimizado, idealmente reduziria o traço da matriz de variâncias, o que significaria uma redução da discrepância entre os dados estimados pelo sistema e os dados reais obtidos através de medição. Para a definição de tal critério, é necessário a escolha de uma função de perda L, responsável por mapear para números que representem um custo, ou perda, propriamente dita, e precisa ser não negativa, e real. Portanto, busca-se minimizar a média, ou, perda esperada $\mathbb{E}\{L(x_k - \hat{x}_k)\}$.

O primeiro dos critérios mais comuns é o critério L1, que minimiza o somatório da norma dos erros de predição/estimação do modelo, portanto, minimizando $\sum_i |x_k - \hat{x}_k|$, com $\hat{x}$ sendo o valor estimado para o estado x_k. Este critério é pouco sensível a valores muito discrepantes, o que o torna menos aplicável para diversas situações na engenharia. Contudo, o critério L2, ou critério dos mínimos quadrados, busca minimizar a soma dos quadrados dos resíduos, portanto, minimizar $\sum_i (x_k - \hat{x}_k)^2$, ou da forma generalizada em notação matricial, $x_k^T S x_k$, sendo S uma matriz definida positiva, para permitir a sua inversão. Uma consequência pertinente é que $\mathbb{E}\{L2(x_k - \hat{x}_k)\} = tr\{\mathbb{E}\{P_k\}\}$, sendo $P_k = \mathbb{E}\{(x_k - \hat{x}_k)(x_k - \hat{x}_k)^T | Y_k\}$, ou seja, sua escolha como critério a se minimizar objetivamente minimiza o traço da matriz de

covariâncias do sistema. Há também a possibilidade de utilização de ambos os critérios juntos ao se utilizar o menor dos dois valores apresentados, contudo, a escolha depende diretamente da problemática e envolve parâmetros de projeto diversos. O conhecimento das informações relacionadas à variância são de igual importância às estimativas para o filtro, tendo em vista que servem de parâmetro para avaliar a qualidade dos dados obtidos.

2.7.1 Filtro de Kalman

O filtro de Kalman é um estimador Bayesiano, ou probabilístico, para sistemas lineares. Sua formulação se dá pela correção de parâmetros, como os detalhados na seção 2.7, de forma recursiva com base em novas medições. Seu processo se resume em duas etapas principais, a predição, e a correção. Para a etapa de predição, estima-se $p(x_k|y_0, y_1...y_{k-1})$, ou seja, estima-se a densidade de probabilidade do estado atual dadas as medições até o instante anterior, ou simplesmente Y_{k-1}. Para a etapa de correção, se adequa o modelo de forma a minimizar o erro quadrático médio com base na observação também do presente instante, ou seja, Y_k.

Seguindo a regra de Bayes, definida pela equação (2.44), pode-se correlacionar a função de densidade de probabilidade do estado do sistema, dada a probabilidade da saída conforme o estado, e as funções de densidade das variáveis individuais. Percebe-se que a saída do sistema no numerador depende dos estados do sistema e portanto pode-se representá-lo como $p(y_k|x_k, y_1, \dots, y_{k-1}) = p_{v_k}(y_k - C_k x_k)$, dada a sua dependência com o ruído branco v_k, a partir da equação (2.43).

$$p(x_k|y_k) = \frac{p(y_k|x_k, y_1, \dots, y_{k-1})\, p(x_k|y_1, \dots, y_{k-1})}{p(y_k|y_1, \dots, y_{k-1})} \tag{2.44}$$

O termo $p(x_k|y_1, \dots, y_{k-1})$, representa o estado *a-posteriori*, ou antes da obtenção de informações atuais do sistema, e portanto, tem a forma de uma distribuição normal que depende do estado do sistema anterior e da sua matriz de covariâncias, enquanto que o termo $p(y|x)$, retrataria a medição do sistema.

$$p\left(x_0, \dots, x_N \mid y_1, \dots, y_N\right) = c\, p\left(x_0\right) \prod_{k=1}^{N} p_{v_k}\left(y_k - h\left(x_k, t_k\right)\right) \times \prod_{k=1}^{N} p_{\Gamma w_k}\left(x_k - \varphi\left(x_{k-1}, t_k, t_{k-1}\right)\right) \tag{2.45}$$

Utilizando do seguinte teorema, apresentado por (JAZWINSKI, 1970), representado na equação (2.46), vale a relação apresentada pela equação (2.45), por um processo análogo, sendo c uma constante independente das variáveis.

$$p\left(y_1, \dots, y_N \mid x_0, \dots, x_N\right) = \prod_{k=1}^{N} p_{v_k}\left(y_k - h\left(x_k, t_k\right)\right) \tag{2.46}$$

Por fim, para determinar $p(y_k|y_1 \dots, y_{k-1})$, considera-se que o processo é análogo ao apresentado para $p(y_k|x_k, y_1 \dots y_{k-1})$, porém, sem a informação do estado x_k e tomando-se a média, portanto, $\mathbb{E}\{p(y_k|y_1 \dots, y_{k-1})\} = \mathbb{E}\{p_{v_k}(y_k - C_k x_k))\}$, ou seja, $C_k \hat{x}_k$. Para a covariância, considerando o sistema utilizando a função de perda L2, pelo critério dos mínimos quadrados, seu valor é $\mathbb{E}\{(y_k - C_k \hat{x}_{k|k-1})(y_k - C_k \hat{x}_{k|k-1})^T \mid y_1, \cdots, y_{k-1}\}$, que utilizando da equação (2.42), pode-se simplificar para $C_k P_{k|k-1} C_k^T + R_k$. Ao fim, pode-se definir $p(y_k|y_1 \dots, y_{k-1}) \sim N(C_k \hat{x}_k, C_k P_{k|k-1} C_k^T + R_k)$. Na prática, o termo $p(y_k|y_1 \dots, y_{k-1})$ representa um fator de escala que garante que $\int_{\infty}^{\infty} p(x_k|y_1, \dots, y_k)dx = 1$, e pode ser simplificado por uma constante η ou modelado conforme a necessidade do sistema.

Considerando estas informações, é possível desenvolver, como apresentado por (JAZWINSKI, 1970), a solução que leve à equação (2.47), equação (2.48), e à equação (2.49), e que exige linearidade nas variações com o tempo do sistema, bem como exige a distribuição normal para as variáveis aleatórias. O resultado é o denominado filtro de Kalman, que é obtido ao buscar uma representação de estimativa do estado que minimize o erro médio quadrático de predição considerando todas as medidas anteriores. A variável a se minimizar, portanto, é dada por $E\sum_i (x_k - \hat{x}_k)^2$, pelo critério L2, e é igual à média dos estados estimados *a-posteriori*.

Aqui, define-se o uso do termo inovação para a expressão $y_k - C_k X_{k|k-1}$ e ganho de Kalman para a matriz G_k.

$$\hat{x}_k = \hat{x}_{k|k-1} + G_k(y_k - C_k \hat{x}_{k|k-1}) \tag{2.47}$$

$$P_k = (I - G_k C_k) P_{k|k-1} (I - G_k C_k)^T + G_k R_k G_k^T \tag{2.48}$$

$$G_k = P_{k|k-1} C_k^T (C_k P_{k|k-1} C_k^T + R_k)^{-1} \tag{2.49}$$

Percebe-se que estes resultados determinam uma solução para o problema de correção, enquanto as Eqs[2.38, 2.42] resolvem a etapa de predição com uma pequena adaptação, definindo a escolha de parâmetros como todos pertencentes ao instante k, gerando como resultado um valor preditivo para $k + 1$. Vale o destaque que, para sistemas invariantes no tempo, o filtro de Kalman atuaria normalmente, porém de forma que suas matrizes permanecessem constantes. Também é notório ressaltar que as medições realizadas y_k não afetam a correção da covariância do sistema, já que suas variações são modeladas pela matriz R_k.

O filtro de Kalman, pela sua característica adaptativa e recursiva, consegue, com base nos dados estatísticos relacionados à variância do processo de medição, contornar possíveis problemas com amostras de dados muito irregulares. Percebe-se que, para uma variância

$R_k \to \infty$, que o termo que influenciaria na correção da predição com base na observação, o ganho de Kalman G_k, se aproxima de zero. Analogamente, o contrário também é válido e, caso as amostras sejam acuradas e precisas, o sistema tende a corrigir com mais fidelidade as predições realizadas.

2.7.2 Fusão sensorial

Por vezes um sistema possui uma ambiente heterogêneo de coleta de dados, em que cada sensor utilizado contribui com uma diferente forma de erro à aferição realizada. A fusão sensorial envolve a integração de informações provenientes desses diferentes sensores para obter medidas mais precisas e confiáveis de uma variável de interesse.

Nesse contexto, o Filtro de Kalman se destaca como uma ferramenta eficaz devido à sua capacidade de lidar com erros e incertezas inerentes às medições, como observado. Portanto- aplicá-lo de modo a integrar no vetor de estados os diferentes valores medidos, irá, através do comportamento do filtro, convergir para um termo que compreenda melhor o estado do sistema, que, ao utilizar um maior conjunto de dados, permitirá reduzir o erro, permitindo uma solução paliativa a uma possível impossibilidade de aquisição de sensores mais caros ou robustos, ou simplesmente uma melhora do sistema. Assim, a aplicação bem sucedida do Filtro de Kalman, na fusão sensorial, proporciona uma solução robusta para diversas aplicações, em específico, rastreamento de objetos.

3 Desenvolvimento do projeto

3.1 Preparação do ambiente

Figura 13 – Foto tirada do ambiente no laboratório

Para a execução do trabalho utilizou-se uma combinação do computador fornecido pelo laboratório, retratado na figura 13, em adição ao uso de máquinas virtuais, via VMware Workstation 17, para testes em ambientes fora do laboratório. Em ambos os casos, utilizou-se o ROS Noetic e o sistema operacional Ubuntu 20.04 como principais ambientes de desenvolvimento, de forma a manter a compatibilidade do sistema desenvolvido com os manipuladores presentes no laboratório para aplicações futuras. Complementarmente, foi selecionada a implementação do ROS na linguagem C++, em oposição à implementação em python, a fim de obter um melhor desempenho para aplicação em cenários de tempo real. Os detalhes da implementação e bibliotecas adicionais utilizadas podem ser observados no repositório virtual do projeto.

3.1.1 Escolha de câmeras

Em relação à utilização de câmeras, com base no desenvolvido na seção 2.3 e nos recursos disponíveis no LARA, optou-se pelo uso de um conjunto de câmeras estéreo STH-MDCS3-VAR, da Videre Design, que se comunicam pelo protocolo firewire. Este conjunto de câmeras a ser utilizado em pares já foi utilizado para tarefas de localização em projetos prévios no laboratório há anos atrás, porém tentativas recentes de seu uso, como a de (BORGES, 2021), envolveram resultados frustrados pela integração com os sistemas atuais dada a obsolescência dos drivers das câmeras Videre. Como tentativa de contornar tais

problemas, foi realizada uma análise de comportamento dentro do ambiente linux atual presente nos computadores do laboratório. Com a utilização de diversos utilitários linux para acesso de dados publicados por câmeras firewire ao computador, não foi possível a obtenção de qualquer resposta de vídeo inicialmente. Contudo, com interfaces de ROS como a camera1394stereo foi possível obter resultados cuja imagem da câmera se apresentou avariada, com apenas metade das informações relevantes. Esse comportamento pareceu estar relacionado a um problema de codificação ou um problema de sincronismo, em que o ambiente não consegue reconhecer a imagem da câmera como uma imagem mono e apenas estéreo, exigindo, portanto, os dados de uma outra câmera para integrar e compor uma imagem completa. Tentativas foram feitas buscando explorar as combinações de formatação das imagens com diferentes codificações e diferentes frequências, contudo, apenas um padrão era reconhecido, e este gerava o fenômeno a ser observado pela figura 14, também observado por (BORGES, 2021).

Figura 14 – Imagem capturada em computador do LARA com tentativa de uso das câmeras estéreo Videre

Tentativas de contactar a fabricante para obter suporte técnico foram infrutíferas tendo em vista que sessaram sua fabricação. Portanto, entre as alternativas restantes estavam a utilização de softwares como o os recursos disponíveis da OpenCV, para realizar a limpeza da imagem, removendo as linhas pretas e reduzindo os dados obtidos para algo mais palatável, ou a busca por trabalhar com os drivers e tentar adaptá-los para os sistemas atuais, encontrando as mudanças cruciais entre as versões e buscando realizar um *retro fitting* do ambiente.

De acordo com a documentação do LARA sobre os sistemas de câmeras, era necessário que automaticamente os drivers *video1394* e *raw1394* fossem montados nos sistema, contudo apenas o último o estava sendo feito. Então, os drivers foram forçados a serem

montados, com as devidas autorizações ao sistema operacional linux toda vez que o sistema ligasse através do arquivo /etc/modules. Essa medida provavelmente resultaria em uma falha, porém, poderia conceder mais informações quanto ao problema. Após novas tentavivas de uso das câmeras no ambiente de teste da própria Videre, ainda não foi possível uma melhoria das imagens. Seguido de buscas na internet, a documentação presente de suporte à câmeras firewire do linux relatou mudanças no kernel do sistema operacional após a versão 5.0 que garantiam que os drivers video1394 e raw1394 fossem substituídos por firewire-core, similares porem não compatíveis binariamente. Essa mudança, portanto, inviabilizaria qualquer uso de recursos que precisassem dos drivers das versões de kernel anteriores.

Em função disso, escolheu-se então a utilização das câmeras C270 HD Webcam Logitech e câmeras Kinect, as quais já foram utilizadas em trabalho anteriores, dos quais ambos os modelos foram empregados intercaladamente, com a implementação utilizando duas câmeras no sistema, apesar do sistema ter sido projetado para múltiplas câmeras. Essa decisão foi motivada pelas limitações do hardware utilizado, já que para o funcionamento adequado do sistema cada câmera deve estar um barramento USB individual. Para a calibração das câmeras utilizou-se o pacote *camera_calibration* do ROS.

Por fim, para o reconhecimento e estimação da pose dos marcadores fiduciais utilizou-se um pacote do ROS para reconhecimento de marcadores ArUco, o *aruco_ros*, por causa deste de ser um modelo de marcador amplamente utilizado e com bons resultados prévios, como visto na seção 2.4 e no trabalho de (BORGES, 2021). Os marcadores em si foram obtidos pelo emprego de ferramentas online de geração de marcadores fiduciais e depois impressos.

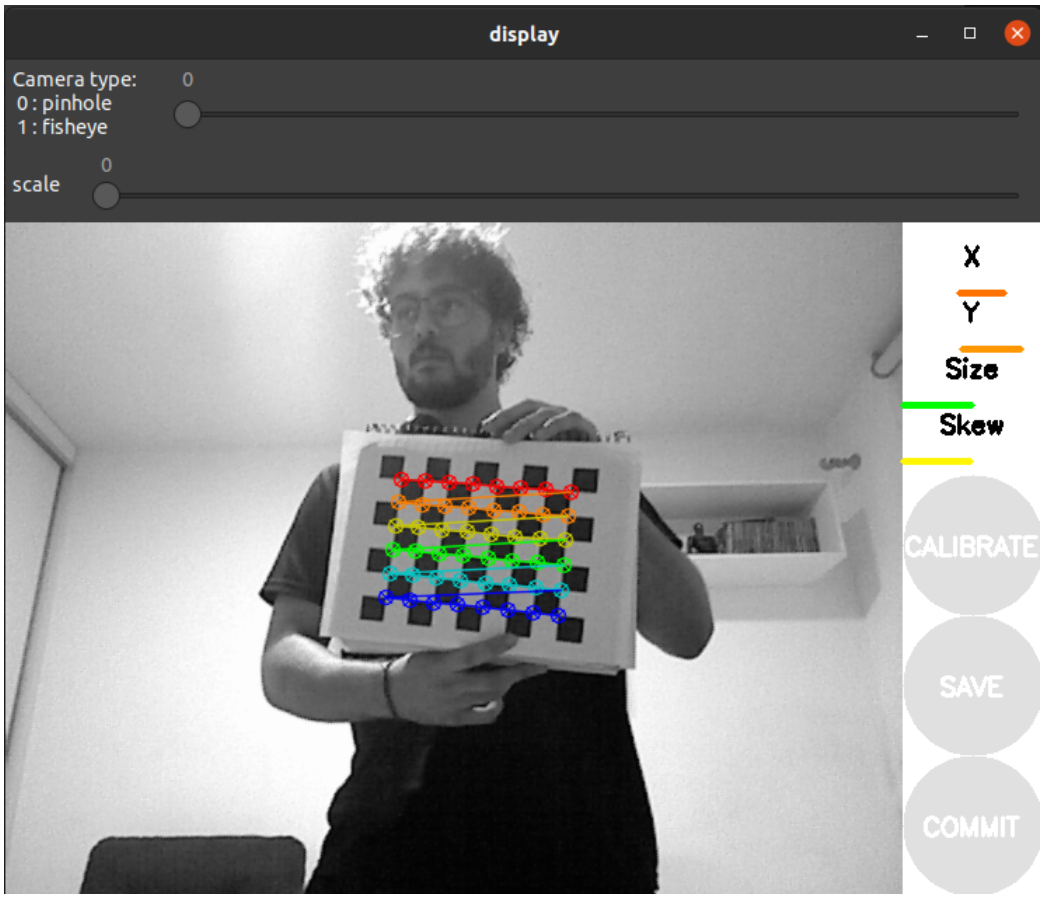
3.1.2 Calibração de câmeras

Como abordado na secção 2.3, as câmeras possuem parâmetros intrínsecos que são fundamentais para que o computador possa corrigir distorções e recriar uma imagem 2D com base nos dados obtidos. Esses parâmetros são salvos em arquivos de texto no ambiente Linux e podem ser gerados através de softwares especializados. Para a calibração das câmeras a serem utilizadas neste projeto, o processo se deu pelo uso do pacote do ROS camera_calibration, que funciona a base da biblioteca OpenCV, de fácil utilização e integração para diversos tipos de câmeras monoculares ou estéreo.

O processo de calibração, apresentado na seção 2.3.1, envolveu a impressão em uma folha A4 de um tabuleiro quadriculado preto e branco de proporções 8x6 de 108mm^2. Os pontos de referência para a calibração são os vértices de intersecção internos do tabuleiro. Ao inicializar a interface gráfica do ambiente de calibração ROS, conferiu-se que as imagens das câmeras estavam adequadas e então movimentou-se o tabuleiro de modo a gerar variação em todos os 6 movimentos principais em cada eixo, de deslocamento e rotação, positivos e negativos. Esse processo foi realizado múltiplas vezes, comparando com a geração de poses

no sistema, para garantir que o resultado era robusto e não sofria de valores discrepantes causados por interferências consideráveis. A figura 15 retrata o procedimento de calibração.

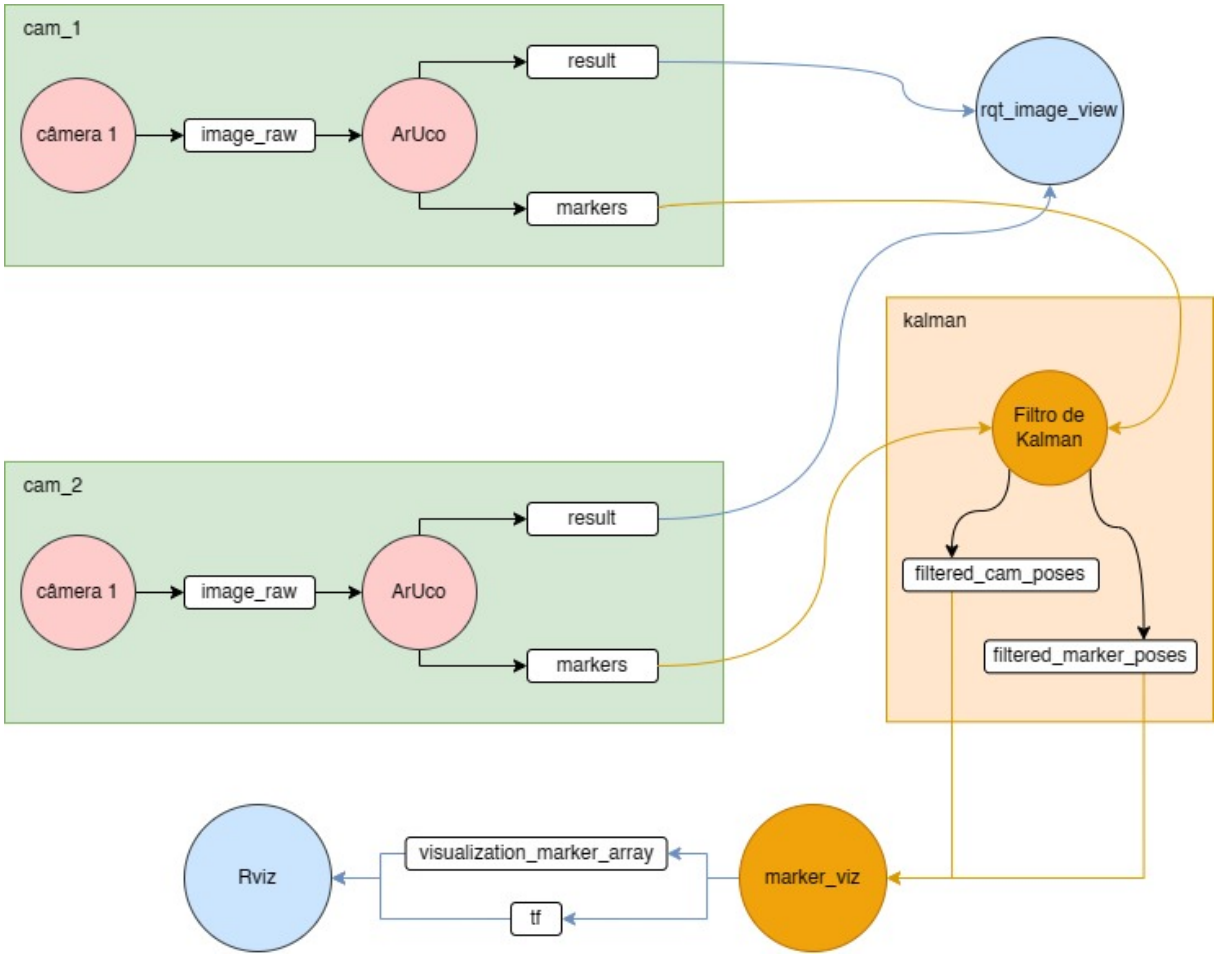
Figura 15 – Calibração com tabuleiro



Percebe-se que, na figura 15, está retratado o andamento de um processo de calibração, com 4 indicadores na lateral da interface gráfica avaliando se há amostras de dados de imagens suficientes representando o posicionamento do tabuleiro ao longo dos eixos X, Y e Z (*Size*), bem como também com as rotações X e Y pelo parâmetro *Skew*.

3.2 Organização geral do sistema

Figura 16 – Diagrama da estrutura do projeto em ROS



Observando a figura [16], tem-se a organização básica do sistema dentro do ROS no quais os círculos representam os nós do sistema, os retângulos coloridos representam os *namespaces* empregados, os retângulos brancos representam os tópicos criados. Complementarmente as setas representam os relacionamentos de *publisher*/*subscriber* para cada tópico, em que setas entrando no tópico significam que o respectivo nó publica no tópico em questão e as setas saindo do tópico significam que o respectivo nó se inscreveu no tópico em questão. Por fim, setas azuis estão relacionadas aos processos de visualização dos dados e as laranjas estão relacionadas ao processo de tratamento de dados/filtragem.

Seguindo então da estrutura apresentada anteriormente, tem-se que as duas câmeras utilizadas para o sistema publicam as suas respectivas imagens no tópico *image_raw*, formato de um vetor de inteiros. Em seguida, essas imagens são processadas nó ArUco, responsável pela identificação e estimação da pose do marcador em relação a câmera que o observou, dada pelo movimento ${}^{c}\xi_{m}$. Dessa pose obtida, no tópico *result* é publicada a imagem captada pela câmera com o sistemas de coordenadas dos marcadores observados sobrepostos, que por ser visto por meio do pacote ROS *rqt_image_view* e foi utilizado como uma ferramenta de

verificação da detecção dos marcadores. Já no tópico *markers* são publicadas as poses ${}^{c}\xi_{m}$ de todos os marcadores observados pela respectiva câmera, na forma de um vetor de poses, dadas na representação par vetor-quatérnion, definida na seção 2.2.4, juntamente de metadados acerca do marcador observado. Optou-se por esta representação de pose me função desta ser mais compacta que a representação utilizando matiz de rotação e diferentemente da representação de rotação em ângulos de Euler, a utilização de quatérnions não apresenta singularidades, como descrito em (CORKE, 2023).

Em seguida, essas poses então são utilizadas pelo nó ROS do Filtro de Kalman, que após processá-las é capaz de determinar as poses de todas as câmeras e todos os marcadores em relação a um referencial fixo. As poses das câmeras e marcadores que compõem o sistema então são publicas nos tópicos *filtered_cam_poses* e *filtered_marker_poses*, respectivamente, também na forma de um vetor de poses na representação vetor-quatérnion.

Finalmente, as poses obtidas pelo filtro são utilizadas pelo nó ROS *marker_viz*, que tem o objetivo de convertê-las em objetos visualizáveis pelo pacote de simulação e visualização de ambientes 3D do ROS, o *Rviz*. Para isso, informação das poses de cada componente do sistema são encapsuladas e publicadas no tópico *tf*, o qual o Rviz utiliza para determinação do posicionamento de cada item, além de ser capaz de definir e plotar o sistema de coordenadas de cada um desses. Complementarmente, o tópico *visualization_marker_array* tem a funcionalidade por carregar as informações visuais de cada marcador do sistemas que podem ser plotadas pelo Rviz, tais como formato, cor e tamanho.

Todo o processo aqui descrito pode ser analisado diretamente em sua implementação nas linguagens C++ e Python através dos apêndices A.1 (Código do nó do ROS para o filtro de Kalman), A.2 (Código de integração entre visão e filtrage), A.3 (Código de filtragem de Kalman), A.4 (Código de organização das variáveis do filtro) e A.5 (Script de inicialização do sistema de visualização).

3.3 Sistema de captação de imagens

O sistema ROS deve ser capaz de obter os dados publicas pelos nós ROS das câmeras e unificá-los de modo que possam ser tratados como o mesmo tipo de dado e processados de acordo. Para isso, é necessário que os nós sejam inicializados conforme as câmeras são detectadas no computador, e um mapeamento deve ser feito. Um script em Python retratado no apêndice A.5 visa facilitar a configuração e inicialização de câmeras nesse contexto do ROS (Sistema Operacional de Robôs) em ambientes Linux.

Primeiramente, é definida a função "get_device_type(tokens)", a qual, dado um conjunto de tokens representando informações de um dispositivo USB obtidas por meio do comando "lsusb | grep Camera|Webcam", determina o tipo de dispositivo com base na condição de o terceiro token a partir do final ser ou não igual a 'Xbox'. Posteriormente, o

script executa o comando "lsusb | grep Camera|Webcam"no terminal utilizando os.popen(), visando obter informações relativas a dispositivos USB associados a câmeras ou webcams. O resultado é processado e organizado em linhas, constituindo uma lista denominada devices. Os dispositivos são então segregados conforme o identificador do barramento USB "bus_id". O script verifica se mais de um dispositivo está conectado ao mesmo barramento, o que seria considerado um erro. Em caso de detecção de erro, a variável error_flag é ativada, e uma mensagem de erro é emitida. Para cada dispositivo, o tipo (kinect ou usb_cam) é determinado pela função "get_device_type()". Além disso, são contabilizados o número de dispositivos Kinect (num_kinect) e o número de dispositivos USB "num_usb_cam". Uma estrutura de dados denominada "all_usb_bus"associa o identificador do barramento USB ao tipo de dispositivo, com propósitos de depuração. Caso não haja erros ("error_flag"é falso), o script inicializa os nós ROS relativos às câmeras em terminais GNOME distintos. Inicialmente, é lançado o nó mestre ROS (roscore). Subsequentemente, para cada dispositivo Kinect e USB, são lançados nós específicos para cada câmera mediante o comando "roslaunch". Cada nó é executado em um novo terminal GNOME, e é introduzido um intervalo entre os lançamentos para assegurar uma inicialização adequada.

O resultado de tal sistema é um ambiente em que câmeras padrão USB Logitech e câmeras Kinect publicam dados através de nós com nome no formato *cam_x*, sendo *x* o identificador único serializado, iniciado em 1. Para cada nó, observa-se as mensagens e os tópicos respectivos de cada tipo de câmera. Algumas informações de configuração relacionadas à resolução, aos *namespaces* e a demais parâmetros, podem ser observados em arquivos denominados *launch files*, que podem ser encontrados nas pastas do ambiente ROS e são referenciadas no script Python.

3.4 Integração da visão com a filtragem

3.4.1 Visão geral e processos

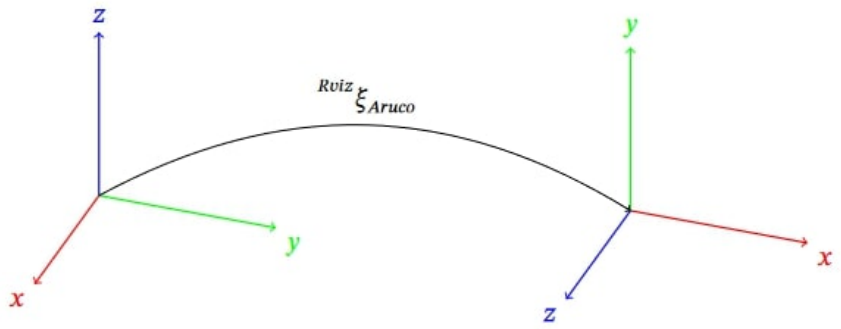
Como observado na seção 3.2, tem-se que o nó ROS responsável pela implementação do filtro de kalman pode ser dividido em duas etapas: processamento das poses dos marcadores vindo de cada uma das câmeras e a implementação do filtro propriamente dito. Nesse contexto, a primeira etapa tem o objetivo de gerenciar e agregar as informações obtidas por cada câmera de forma a construir o estado do sistema que será incorporado com uma observação para o filtro de kalman.

Primeiramente, o nó é inicializado de forma a definir o estado inicial do sistema, composto apenas por uma lista das câmeras inicializadas e um conjunto de lista vazias, que representam a lista de marcadores observados por cada câmera. Neste momento, a referência do sistema inteiro será dada pela pose da primeira câmera inicializada no sistema, também chamada de mundo, representado por ξ_{c_1}, no qual seu valor será fixado e dado por:

$$\xi_{c_1} = \oslash \tag{3.1}$$

As demais câmeras também serão definidas com o mesmo valor de pose em relação ao referencial, já que ainda não é possível inferir seu respectivos valores.

Figura 17 – Sistemas de Coordenadas usados pelo Rviz e pelo Aruco



No entanto, ao longo da implementação do projeto percebeu-se uma discrepância entre os sistemas de coordenadas para as câmeras utilizados entre o ambiente de visualização Rviz e o pacote ArUco, resultando em uma representação errônea do sistema no simulador. Essas discrepâncias podem ser observadas na figura 17, com a representação do Rviz à esquerda da imagem e a representação do ArUco à direita. Este problema foi relatado por outros usuários do software, e os desenvolvedores esclareceram que se tratava de uma definição arbitrária para adequação à outras bibliotecas e referências utilizadas. Ademais, como observado na seção 2.2.1, essa discrepância pode ser resolvida apenas aplicando um movimento ${}^{Rviz}\xi_{Aruco}$ de forma a trazer o sistema de coordenadas do Rviz para o utilizado pelo ArUco, dado por apenas por uma rotação dos eixos x e y em +90°, respectivamente. Tal rotação pode ser escrita na notação par vetor-quatérnion da seguinte forma:

$${}^{Rviz}\xi_{Aruco} = (t = (0{,}0{,}0),\ \hat{q} = 0{,}5 + 0{,}5i + 0{,}5j + 0{,}5k) \tag{3.2}$$

Com isto, é aplicado então a composição deste movimento com a pose da câmera *cam_1*, a fim de trazê-la para o sistema de coordenadas utilizado pelo ArUco, com as demais câmeras definindo suas poses com o mesmo valor similarmente ao processo descrito anteriormente. Essa composição é dada então pela expressão a seguir:

$$\xi_{c_1} = \xi_{c_1} \oplus {}^{Rviz}\xi_{Aruco} = \oslash \oplus {}^{Rviz}\xi_{Aruco} = {}^{Rviz}\xi_{Aruco} \tag{3.3}$$

Adicionalmente, percebeu-se que os dados entregues pelo nó ArUco apresentavam uma rotação de 180° em torno do eixo Z, de forma que foi-se necessário aplicar uma correção nas medições provenientes deste nó. Para isso, foi aplicada uma rotação adicional de 180° em torno do eixo Z, de forma análoga ao aplicado na equação (3.3), por meio da transformação:

$$\xi = (t = (0,0,0),\ \hat{q} = 0 + 0i + 0j + 1k) \tag{3.4}$$

Além disso, cada câmera também possui associada uma pose relativa à uma outra câmera do sistema, chamada de *câmera anterior*, conceito este que será melhor definido futuramente ao longo desta seção. Inicialmente, esta pose será dada pela seguinte expressão, com c_i sendo a i-ésima câmera do sistema e $c_{anterior}$ a câmera anterior em relação à c_i:

$$^{c_i}\xi_{c_{anterior}} = \oslash \tag{3.5}$$

Em seguida, o nó realiza o processo de atualização do sistema, no qual ele checa por novas atualizações no tópico *markers* para cada câmera, iterativamente e em ordem de inicialização no sistema, com uma período de 5ms. Este intervalo foi escolhido de forma que filtro seja capaz de se adaptar rapidamente a mudanças bruscas no sistema observado, porém ainda valendo-se de um período de amostragem constante a fim de facilitar a modelagem do filtro e reduzir a demanda de processamento. Quando uma atualização é observada em uma ou mais câmeras, incia-se então processo de integração das informações obtidas no sistema.

Considerando c_i a i-ésima câmera do sistema, é observado então os identificadores de cada marcador observado por c_i. Em seguida observa-se se existe uma câmera c_j que observou um marcador com o mesmo identificador no mesmo instante, com $j = 0 \dots (i-1)$. Caso c_j não exista, é então comparado o identificador do marcador com a lista de marcadores já observados anteriormente por c_i. No caso do marcador em questão já ter sido observado anteriormente, a sua informação de pose é sobrescrita pela nova pose observada. Já no caso do marcador não haver sido observado anteriormente, este é apenas adicionado na lista de marcadores observados por c_i. No entanto, caso exista uma câmera c_j que cumpra as condições dadas acima, ela é definida como a câmera anterior e a pose relativa entre c_i e $c_{anterior}$ é atualizada, de forma que:

$$^{c_i}\xi_{c_{anterior}} = {}^{c_i}\xi_m \ominus {}^{c_j}\xi_m = {}^{c_i}\xi_m \oplus {}^{m}\xi_{c_j} \tag{3.6}$$

No quais $^{c_i}\xi_m$ e $^{c_j}\xi_m$ representam, respectivamente, as poses relativa do marcador m observado em relação às c_i e c_j, obtidas diretamente pelo nó ArUco no tópico *markers*. Em seguida, o processo de atualização da pose do marcador para c_i segue o mesmo procedimento descrito anteriormente para o caso de c_j não existir. Por fim, o procedimento citado acima é repetido para cada um dos marcadores observados por c_i.

Após isso, incia-se então o procedimento de consolidação da informações obtidas por c_i em um vetor de estado do sistema a fim de ser integrado ao filtro como uma observação. Para isso, é necessário primeiramente definir a nova pose de c_i em relação ao referencial, por meio da seguinte relação:

$$\xi_{c_i} = \xi_{c_{anterior}} \ominus {}^{c_i}\xi_{c_{anterior}} = \xi_{c_{anterior}} \oplus {}^{c_{anterior}}\xi_{c_i} \tag{3.7}$$

Com a pose de c_i em mãos, é definida a pose de um marcador m observado por c_i em relação ao referencial será dada pela seguinte expressão:

$$\xi = \xi_{c_i} \oplus {}^{c_i}\xi_m \tag{3.8}$$

De forma complementar, a poses das demais câmeras e marcadores não observados por c_i pode ser obtida de forma análoga a demonstrada na equação (3.7),e equação (3.8), porém utilizando os valores do último instante observado para cada câmera. Com isso, é possível montar então o vetor de estados do sistema, de forma que:

$$x_{k+1} = \begin{bmatrix} \xi_{c_1} \\ \xi_{c_2} \\ \vdots \\ \xi_{c_m} \\ \xi_{m_1} \\ \xi_{m_2} \\ \vdots \\ \xi_{m_n} \end{bmatrix} \tag{3.9}$$

Nos quais m e n representam o número de câmeras e marcadores presentes nesta observação, respectivamente. Finalmente, o processo inteiro descrito para c_i repete-se para cada uma das câmeras do sistema que apresentarem novas atualizações no tópico *markers* para o dado instante.

3.4.2 Detalhes de implementação

A classe *RosFilter* em C++, como apresentado pelo apêndice A.2 implementa diversos processos realizados para a adequada comunicação com o filtro e com os dados publicados pelos nós das câmeras. De modo geral, há cálculos de mudança de estruturas de dados, adaptando tipos de mensagens de ROS para tipos de estruturas de "tf", ou funções de transferência, permitindo a aplicação de rotações e outros processos de álgebra linear, por exemplo. Também são implementados todos os processos descritos na seção anterior.

De modo mais específico, o construtor *RosFilter* realiza a inicialização de membros essenciais, incluindo um objeto do tipo "filter"para representar o filtro de Kalman e um manipulador de nó "ros::NodeHandle"para facilitar a comunicação com os processos do ROS. Entre os métodos públicos, "getCameraTopics()"identifica os tópicos de publicação associados a marcadores ArUco para cada câmera, enquanto "subscribeTopics()"realiza a subscrição aos tópicos identificados, além de criar os respectivos publicadores. A função "createTimer(ros::Duration period)"gera um temporizador ROS para a atualização periódica do estado do sistema, e "insertCameraBasis(int camera_id)"insere informações específicas de uma câmera para uso futuro no filtro. A classe possui atributos privados fundamentais, como

o objeto "filter kf"representando o filtro Kalman, o manipulador de nó "ros::NodeHandle nh"para comunicação ROS, e diversas estruturas de dados para armazenamento eficiente de informações sobre marcadores, publicadores, tópicos, entre outros. Os métodos privados incluem operações especializadas, como a criação de publicadores para dados filtrados ("createPublisherFiltered") e poses de câmeras ("createPublisherCameras"), a extração do identificador de câmera a partir do nome do tópico ("extractCameraID"), além de outras funções dedicadas à manipulação de dados do filtro Kalman. Callbacks são implementados para lidar com dados provenientes dos tópicos, como "cameraCallback"para manipular informações de marcadores e "timerCallback"para tratar eventos de atualização do filtro.

3.5 Implementação do filtro de Kalman

3.5.1 Visão geral e processos

Buscando uma boa modelagem do sistema de visualização no espaço de estados, conforme a equação (2.38) e os fundamentos desenvolvidos na seção 2.7, foi necessário presumir certas linearidades e simplificações no modelo, portando, atribuindo à variável aleatória um maior peso quanto a possíveis imprecisões de modelagem e efeitos de ruído relacionados aos parâmetros intrínsecos e extrínsecos das câmeras.

Pela arquitetura de projeto, o nó do ROS responsável pela filtragem deve recebe os dados publicados pelos nós da ArUco, portanto, já tendo a relação de pose com o marcador. Essa informação deve ser armazenada em um vetor que correlaciona a este um índice específico para o identificar, bem como também o atrela a um índice do vetor de estados. Esse direcionamento é fundamental dado o caráter dinâmico do sistema de aumentar o vetor de estados conforme novas informações são obtidas. Como escolha de projeto, definiu-se que estados já inseridos se manteriam no vetor independente de não mais observados.

Quanto à paradigmas de programação deste ambiente, definiu-se o uso da orientação a objetos, como um forte aspecto da linguagem a ser utilizada, C++, e também pela simplicidade em lidar com diferentes instâncias de filtros, para caso seja de interesse o trabalho com herança de funções para diferentes variantes do filtro, como o filtro estendido ou *unscented*. Para a implementação das funções e operações matemáticas necessárias para o filtro, foi utilizada a biblioteca Eigen, de versão 3.4, capaz de realizar operações com matrizes enormes, especialmente matrizes cuja maioria de seus elementos são nulos, como é o caso, se adequando às características do presente projeto, tendo em vista sua escalabilidade.

A princípio, arquitetou-se uma modelagem do sistema simples, para ser desenvolvida futuramente com modelos mais complexos. Essa modelagem, denominada modelagem de velocidade nula, ou de posição constante, assume que os pontos observados pelas câmeras tenderão a se manter na posição que se encontram no momento de medição. A consequência

disto é que para movimentações mais bruscas, o filtro se comportará com mais dificuldade de prever sua posição de forma acurada, porém, realizando menos operações e consumindo menos espaço na memória. Toda e qualquer variação observada entre a posição estimada $x_{k+1|k}$ e a posição mensurada $\hat{x}_{k+1}$ será modelada como um erro de imprecisão no processo de observação das câmeras e por distúrbios no sistema, ou seja, representado por v_{k+1} e w_{k+1}, respectivamente. A representação desta modelagem é dada pela equação (3.10), e equação (3.11). Para o sistema descrito, espera-se que o mesmo se mantenha, portanto, a matriz de transição de estados Φ toma a forma da matriz identidade. A matriz de observação de estados C também toma a forma da identidade já que a informação obtida pelas câmeras e pelo sistema ArUco também é o estado a ser avaliado. Essas matrizes, pelas características do sistema, se manterão constantes, já que o mesmo não mudará seu caráter ao longo do tempo de execução. Também não são realizadas transformações lineares nos vetores aleatórios w_k e v_k, já que sua incerteza simplesmente será modelada de forma a adaptar estas modificações lineares, mantendo suas propriedades que respeitam as distribuições Gaussianas de ruído branco.

$$x_k = \Phi_k x_{k-1} + w_k \tag{3.10}$$

$$y_k = C_k x_k + v_k \tag{3.11}$$

O filtro foi desenvolvido para ser escalável de modo que a adição de novos estados permitisse a não alteração de dados previamente adicionados e tratados, portanto, o formato da variável de estado se dá pelo concatenamento dos vetores ξ de pose obtidos, ou seja, $x_k = [\xi_1 \xi_2 \dots]$, sendo $\xi = [x_t, y_t, z_t, x_q, y_q, z_q, w_q]$. Ademais, para a implementação do algoritmo de filtragem apresentado na seção 2.7.1, se faz necessária a definição arbitrária das demais matrizes.

As matrizes relacionadas à distribuição das variáveis aleatórias precisa conter um formato tal que separa as imprecisões de variáveis lineares de variáveis angulares, levando em consideração o formato de representação par vetor-quatérnion e as definições de publicação da ArUco, definiu-se que o vetor de estados seria composto por 3 elementos lineares, ou seja, o comum vetor xyz, e 4 elementos angulares do quatérnion, ou seja, xyzw, compondo ao todo 7 elementos. A matriz Q e a matriz R, portanto, sabendo que devem ser diagonalizadas, de forma que se presume que suas variáveis não possuam relação entre si, deve possuir em ordem, 3 variâncias definidas para os elementos lineares e 4 para os angulares, como apresentado pela equação (3.12).

$$Q,R = \begin{bmatrix} var_{lin} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & var_{lin} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & var_{lin} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & var_{ang} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & var_{ang} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & var_{ang} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & var_{ang} \end{bmatrix} \tag{3.12}$$

Não há valores ideais a serem pré-definidos para estas constantes, portanto, estimou-se $var_{lin} = 0.1m$, ou seja, 10cm de variância, sendo considerada uma boa faixa inicial para testes e para um protótipo, e de forma comparativa, estimou-se $var_{ang} = 0.001rad$. Esses valores são definidos baixos para garantir que o filtro considere as observações imprecisas o suficiente para realizar as correções mas não tanto a ponto de levar o sistema à alguma instabilidade temporária, ou demorando demais para convergir. Para a matriz R, presume-se a identidade, portanto, uma variância moderada, não muito alta ou baixa.

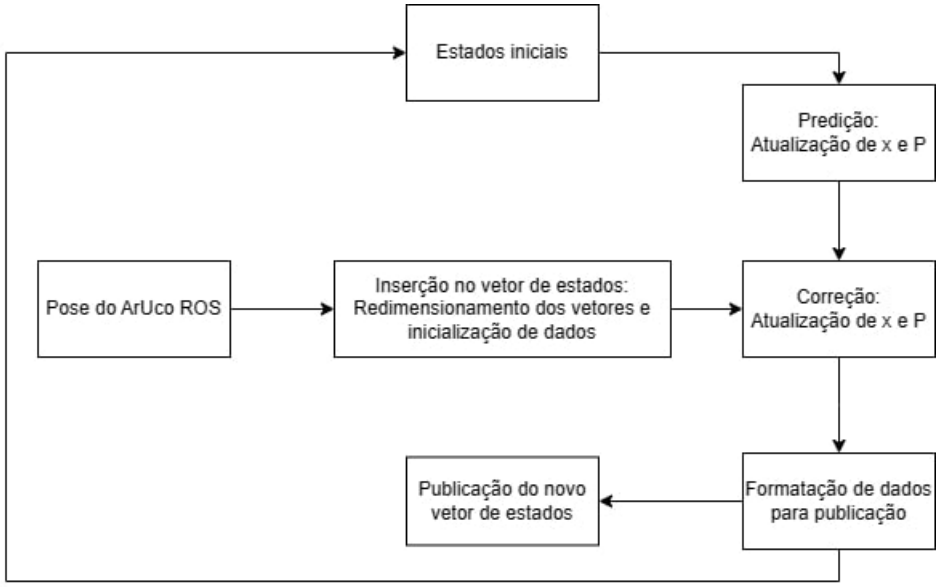
O comportamento do filtro, se resume em inicializar as matrizes todas que compõe e irão fazer parte de equações *a priori* e *a posteriori*. Elas são inicializadas para realizar o tratamento de apenas um elemento de pose no vetor de estados, ou seja, um vetor de 7 elementos, que representará o mundo. Também são implementadas funções para adição de novos vetores no vetor de estados, alterando as matrizes aleatórias com os valores previamente definidos até então, sem alterar os valores gerados pelo sistema. Por exemplo, a matriz de covariâncias dos estados P_k é inicializada como uma matriz diagonal cujos termos possuem um valor de variância iguais a 0.1, e não é alterada ao se inserir novos elementos, apenas redimensionada, com os novos elementos também compondo 0.1. Este valor também é definido arbitrariamente ao longo de diversas iterações de uso do filtro, se mostrando um bom valor para inicialização e facilitação da convergência dos dados.

Para a etapa de predição, o vetor de estados se mantém, tendo em vista o comportamento linear e simplificado na passagem entre os estados. Portanto, presume-se que o sistema tenderá a não se movimentar enquanto houver uma passagem de instantes. Então, uma nova matriz de covariância é calculada conforme a equação (2.42), com $\Gamma = I$.

Para a etapa de correção, são recebidos os dados obtidos pelas câmeras na ordem em que são listados no vetor de estados e a predição da saída com base nos estados presentes do sistema em relação à saída é feita diretamente, já que vale $C_k = I$. O erro de predição é, portanto, calculado e, junto ao ganho de Kalman, como representado pela equação (2.49), permite a estimação e filtragem do novo vetor de estados x_k. Por fim, a matriz de covariância é atualizada, e o novo erro de medição também é obtido. Funções que retornam cada uma dessas informações foram criadas para serem utilizadas pela interface de conexão entre o filtro e o publicador dos dados. O seguinte diagrama representado pela figura 18 resume a

estrutura do filtro conforme a implementação apresentada.

Figura 18 – Topologia do filtro de Kalman implementado



3.5.2 Detalhes de implementação

Todos os processos aqui descritos são documentados e apresentados em mais detalhes diretamente no código do programa, apresentado no apêndice A.3. O código implementa uma classe denominada "filter"que encapsula o Filtro de Kalman em si. Os principais atributos da classe incluem o vetor de estado (X), a matriz de covariância (P), as matrizes de transição de estado (A), de observação (H), e as covariâncias dos ruídos do processo (Q) e de medição (R). Além disso, há atributos para o ganho do filtro de Kalman (K) e para as saídas previstas e corrigidas (Y_pred e Y_cor), bem como as inovações ou resíduos associados (I_pred e I_cor). Os métodos principais, como predict() e correct(), executam as etapas essenciais do filtro de Kalman, realizando previsões e correções com base nas medições. As funções getState() e getCovariance() fornecem acesso aos estados estimados e às matrizes de covariância, respectivamente. A classe filter é configurável para lidar com modelos não lineares por meio de protótipos de funções e Jacobianas. Adicionalmente, ela oferece funcionalidades para redimensionamento dinâmico do vetor de estado e para reinicialização do sistema, facilitando a adaptação a diferentes contextos e configurações.

O código apresenta um conjunto de macros e constantes definidas para facilitar a configuração e personalização do filtro de Kalman implementado pela classe "filter". Estas macros e constantes desempenham um papel crucial na flexibilidade do código, permitindo que parâmetros essenciais, como o tamanho do vetor de estados, as covariâncias dos ruídos do processo e de medição, sejam ajustados de maneira rápida e cen-

tralizada. Por exemplo, a constante "POSE_VECTOR_SIZE"determina a dimensão do vetor de estado, enquanto as macros "PROCESS_LINEAR_NOISE_COVARIANCE"e "PROCESS_ANGULAR_NOISE_COVARIANCE"controlam as covariâncias específicas para componentes lineares e angulares. Essas definições tornam o código adaptável a diferentes contextos de aplicação, promovendo uma fácil configuração do filtro de Kalman para atender às exigências específicas de sistemas dinâmicos diversos. Essa abordagem modular e configurável contribui para a versatilidade e aplicabilidade do código em ambientes onde a estimação de estados é crucial.

Para mais detalhes, o próprio código fonte foi comentado de modo a permitir uma boa compreensão do fluxo de dados nas variáveis, bem como o significado de cada processo realizado dentro e fora das funções, a depender do escopo e da clareza do processo executado pelo computador.

3.6 Recriação virtual do ambiente detectado

O nó *marker_viz* tem como objetivo converter os dados obtidos do filtro para uma estrutura de dados interpretável pelo ambiente de visualização do ROS Rviz. Para isso, há três principais códigos responsáveis por tal tarefa e por processos atrelados, o "marker_viz.cpp", "visualization_node.cpp"e o "visualization_objects.cpp", retratados respectivamente, nos apêndices A.6, A.7, A.8.

O primeiro código apresenta uma implementação eficiente para o gerenciamento de marcadores e câmeras em um ambiente ROS. A classe central, denominada "VisualizationHandler", é concebida para facilitar a administração dinâmica e precisa desses elementos no sistema. A classe "VisualizationHandler"mantém listas internas para câmeras, marcadores e assinaturas ROS, garantindo uma organização estruturada. A função "find_marker"desempenha um papel fundamental ao possibilitar a localização eficaz de marcadores no sistema. Utilizando a função "std::find", ela retorna o índice do marcador desejado ou -1 caso não seja encontrado. Outro aspecto crucial é a implementação do método "clear_markers", que utiliza o idiom erase-remove para eliminar marcadores cujo tempo de vida tenha expirado. Essa abordagem dinâmica assegura que apenas informações relevantes sejam mantidas, otimizando o desempenho do sistema. A interação com o ambiente ROS é habilitada por meio de uma série de métodos especializados. A função "callback_markers"é invocada quando novos marcadores são recebidos, atualizando a lista interna de marcadores na classe. Da mesma forma, "callback_cameras"trata a chegada de informações sobre as câmeras, garantindo uma representação precisa. Além disso, a classe fornece métodos para enviar transformações tf,*Transform Frames*, ao RViz, possibilitando uma visualização dinâmica do ambiente. A função "send_tfs"gera transformações tanto para câmeras quanto para marcadores, mantendo a consistência no sistema. Por fim, a classe "Visualizati-

onHandler"se destaca pela sua versatilidade. Com métodos como "start", "publish_markers"e "get_camera_topics", ela proporciona uma interface completa para interagir com o ambiente ROS. Essa abordagem modular facilita a integração e manutenção do sistema, tornando-a uma peça essencial para a gestão eficaz de marcadores e câmeras em aplicações robóticas avançadas.

O segundo código implementa um nó de visualização no ROS, focado em proporcionar uma representação gráfica precisa de câmeras e marcadores em um ambiente tridimensional. O nó é configurado através da classe "VisualizationHandler", projetada para gerenciar eficientemente as informações de câmeras e marcadores. Na função "main", o nó ROS é inicializado e configurado com uma taxa de 30 Hz. A classe "VisualizationHandler"é instanciada, e os tópicos para câmeras e marcadores são definidos como "kalman/filtered_cam_poses"e "kalman/filtered_markers_poses", respectivamente. O loop principal do nó é composto pelas seguintes etapas. Primeiramente, os marcadores expirados são removidos através do método "clear_markers"da classe "VisualizationHandler". Em seguida, as transformações tf são enviadas ao RViz usando "send_tfs", permitindo uma atualização contínua da visualização tridimensional. Por fim, os marcadores são publicados utilizando "publish_markers". Esse nó exemplifica a importância do ciclo de vida do ROS, já que a função "ros::spinOnce"é essencial para processar eventos, garantindo uma comunicação assíncrona eficiente entre os diferentes componentes do sistema. O código destaca-se pela sua simplicidade e modularidade, tornando-o uma adição valiosa para projetos de robótica que requerem uma representação visual em tempo real de câmeras e marcadores. A classe "VisualizationHandler"desempenha um papel central, permitindo a fácil expansão e manutenção do sistema conforme novos requisitos surgem.

O terceiro código apresenta uma série de classes e métodos destinados a gerenciar dados relacionados à visualização em um ambiente ROS. Destacam-se duas principais classes: "VisualizationMarker"e Camera. A classe "VisualizationMarker"encapsula informações sobre marcadores visuais. No construtor padrão, são definidos parâmetros iniciais, como tempo de vida (lifetime), identificação (id), e a posição/orientação inicial. A classe oferece métodos para obter a representação do marcador em diferentes formatos, como a estrutura "visualization_msgs::Marker"e transformações tf. A classe "Camera"modela informações sobre câmeras no sistema. Pode ser inicializada com um identificador único ("frame_id") ou com posição e orientação específicas. O método "get_tf_stamped"retorna a transformação TF da câmera. O código também apresenta funções utilitárias, como "get_rot_from_quat"e "get_quat_from_rot", responsáveis por converter entre representações de rotação (quatérnion e matriz de rotação). Além disso, a função "test_quaternion"verifica a consistência da conversão entre quatérnions e matrizes de rotação. Essas funções são cruciais para garantir a precisão das transformações no ambiente visualizado.

A interação entre os programas é articulada pela classe "VisualizationHandler".

Essa classe age como uma ponte central, sendo invocada pelo nó principal "visualization_node"para coordenar as operações principais do sistema. Métodos como "clear_markers"e "publish_markers"são chamados no contexto do loop contínuo, garantindo uma atualização constante da visualização no RViz. A lista de assinantes ROS é gerenciada pela classe, facilitando a subscrição eficiente aos tópicos relevantes. A sincronia entre o "visualization_node"e a "VisualizationHandler"é essencial para manter a representação visual coerente e em tempo real dos marcadores. O nó principal utiliza intensivamente os métodos da classe, mantendo a manipulação eficiente e garantindo uma visualização precisa no ambiente ROS. A comunicação com os outros códigos é estabelecida por meio de tópicos ROS, com a "VisualizationHandler"atuando como uma camada intermediária para coordenar as informações entre as classes e o nó principal.

4 Validação e resultados

Para a avaliação do sistema, foram realizados diversos testes em ambientes diferentes, sendo estes o próprio LARA, como também residências, cujos ambientes possuíam diferentes intensidades luminosas, bem como diferentes infraestruturas.

4.1 Testes de captação de pose ArUco

Para a validação do sistema de captação de pose pelas câmeras, com o ambiente ArUco, foram realizados testes sistematizados comparando as tendências de movimento, ao longo dos três eixos, entre a referência, no caso o marcador, e a câmera, localizada em um ponto fixo como o ilustrado pela figura 19. Os testes foram realizados com ambos os modelos de câmera Logitech e Kinect, e os resultados foram condizentes com o esperado, após as correções de eixos detalhadas na seção 3.4. A figura 20 ilustra como se deram essas tentativas, bem como demonstra os resultados e as publicações pelo nó do ROS.

Figura 19 – Sequência de posições do marcador para teste

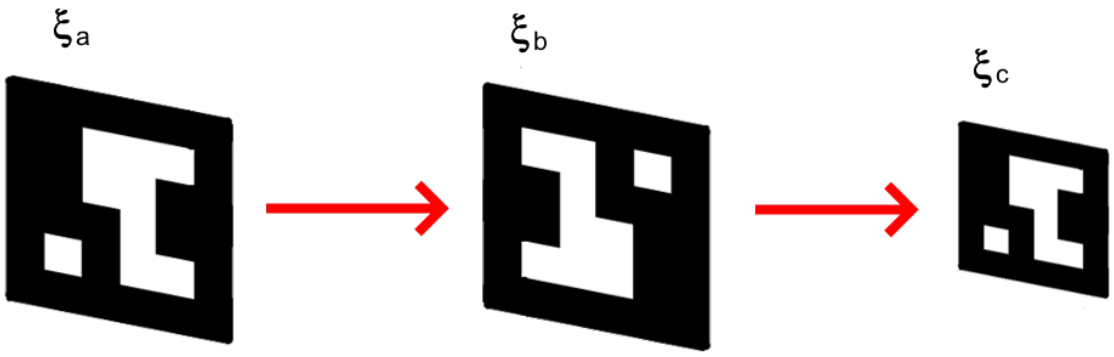
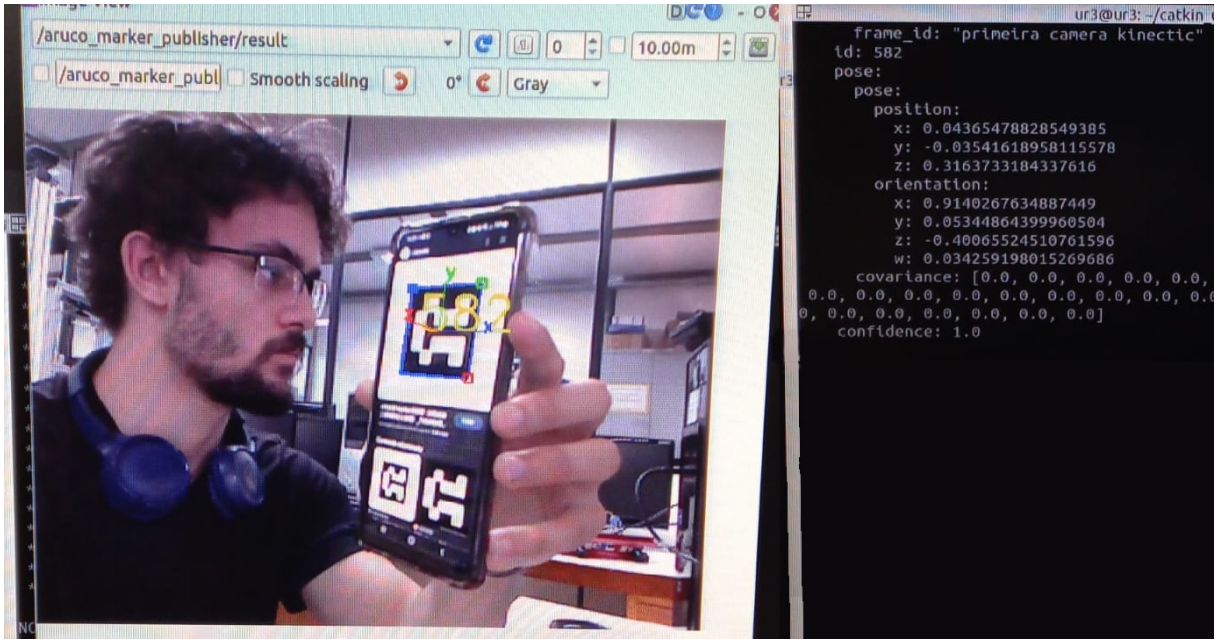
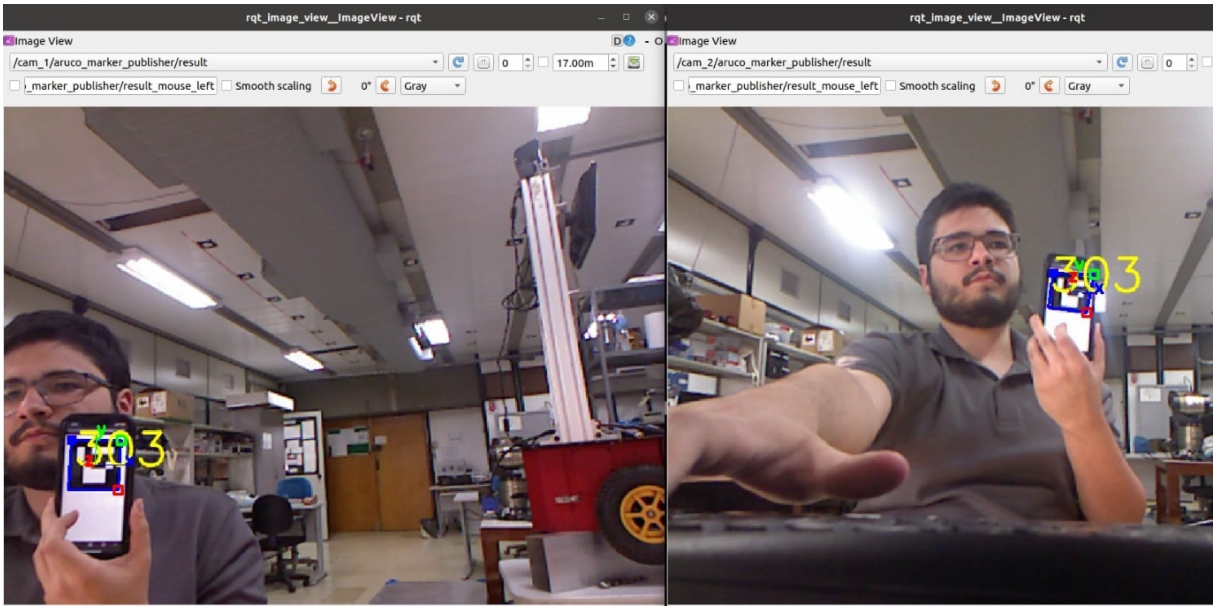


Figura 20 – Teste de publicação de mensagens pela ArUco



Testes semelhantes foram realizados para avaliar o comportamento do sistema para múltiplas câmeras. A figura 21 ilustra como se deram essas avaliações. Ao longo de ambos os testes, pôde-se concluir robustez no sistema, para distâncias de até aproximadamente 2 metros, quando utilizando dispositivos móveis como celulares para mostrar os marcadores fiduciais de forma móvel, contudo, marcadores maiores, dispostos em folhas A4, demonstraram melhor detecção para distâncias de até aproximadamente 7 metros. Para distâncias maiores, o sistema se comportou de forma a eventualmente parar a detecção dos marcadores, podendo-se visualizar tal comportamento pela interface rqt_image_view, integrada nativamente no ROS, quando a numeração em amarelo do código de identificação do marcador ArUco vêm a sumir da tela e a publicação dos dados cessa.

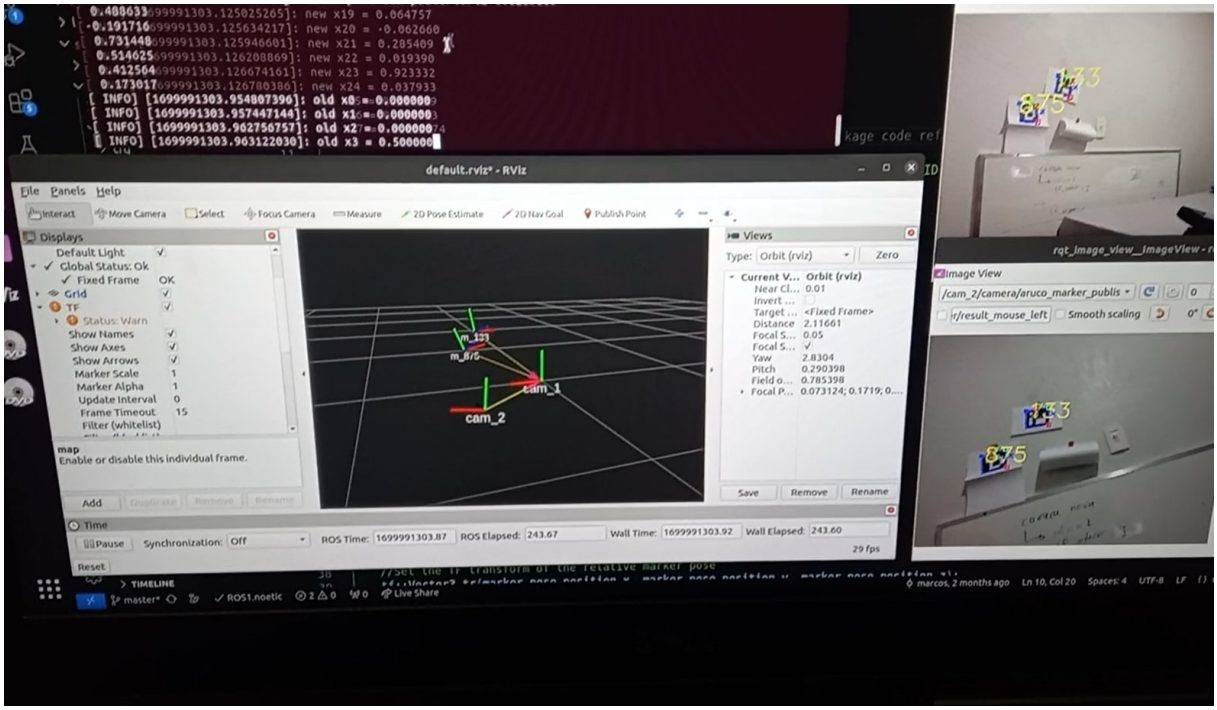
Figura 21 – Teste de detecção simultânea de marcadores no LARA com duas câmeras diferentes



4.2 Teste de ambiente virtual

A figura 22 ilustra um teste em ambiente com uma estrutura preparada previamente para permitir diferentes incidências luminosas em cada um dos marcadores, bem como permitindo a possibilidade de se tampar e destampar cada um dos marcadores para avaliar os efeitos no sistema. O objetivo foi a validação do quão condizentes as poses de marcadores e posições se posicionam. Apesar dos diversos testes apresentados até então, para se realizar a validação do ambiente virtual detectado pelo sistema, a única forma seria observando diretamente a recriação virtual pela interface Rviz. A figura 22 demonstra como se deu este teste, bem como também ilustra a interface gráfica com duas câmeras e dois marcadores sendo apresentados. Contudo, percebeu-se uma variação acentuada da posição da segunda câmera bem como de alguns dos marcadores, denotando, portanto, uma dificuldade de convergência de suas posições. Essa variação não se demonstrou exacerbada, contudo, pode significar um fator limitante para certas aplicações. Na figuram pode-se observar a imagem de cada uma das câmeras em tempo real, destacando sua posição, e à esquerda percebe-se essa distância sendo representada pela diferença posicional no ambiente virtualizado. No geral, a posição relativa entre os marcadores e as câmeras se mostraram adequadas e condizentes com as alterações no ambiente.

Figura 22 – Visualização do sistema com interface gráfica RVIZ



4.3 Testes de robustez e qualidade

Para a melhor validação da qualidade do filtro, se fez necessária a análise das formas de sinal referentes à cada um dos parâmetros posicionais dos marcadores e das câmeras, de modo que seja possível uma análise dos sinais e da qualidade, tanto do processo de filtragem, quanto do sistema ArUco ROS em detecção de pose relativa dos marcadores para as câmeras.

4.3.1 Teste de 1 câmera e 1 marcador

Neste teste, o sistema inicia com uma câmera e um marcador observável por esta câmera. Nas figuras 24, e 25 é retratada a posição do marcador em translação e rotação, respectivamente, e na figura 23 pode-se observar o ambiente de testes com a perspectiva das câmeras. Percebe-se a capacidade do filtro de seguir a rota e filtrar consideráveis inconstâncias posicionais advindas de imprecisões de medição das câmeras.

Figura 23 – Ambiente de teste observado

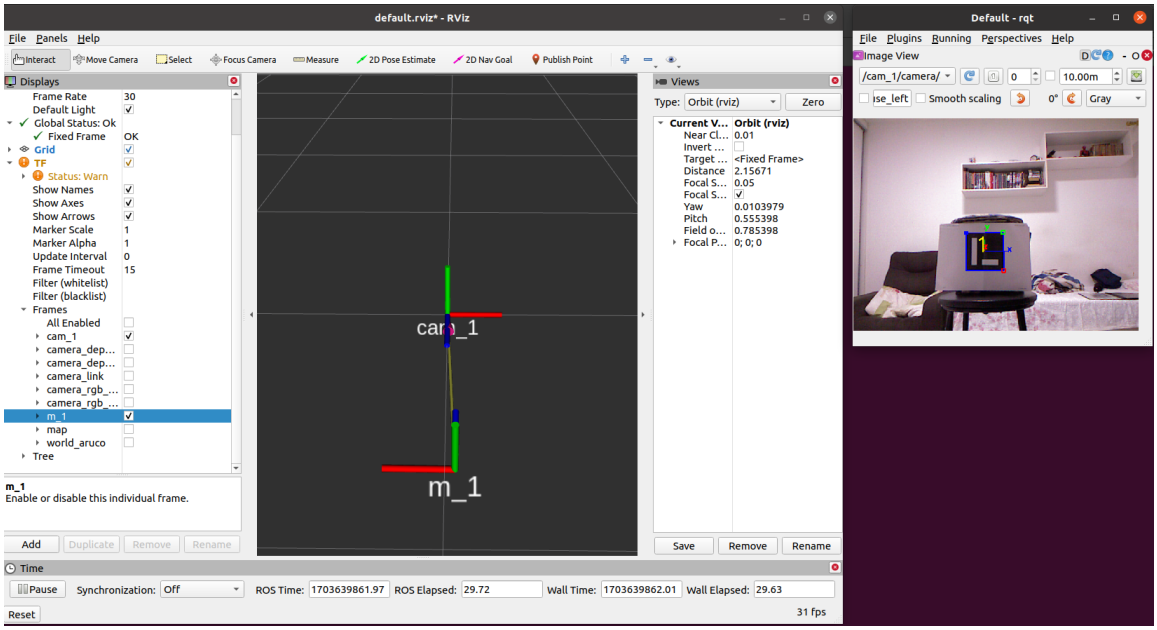


Figura 24 – Posição translacional do marcador observado

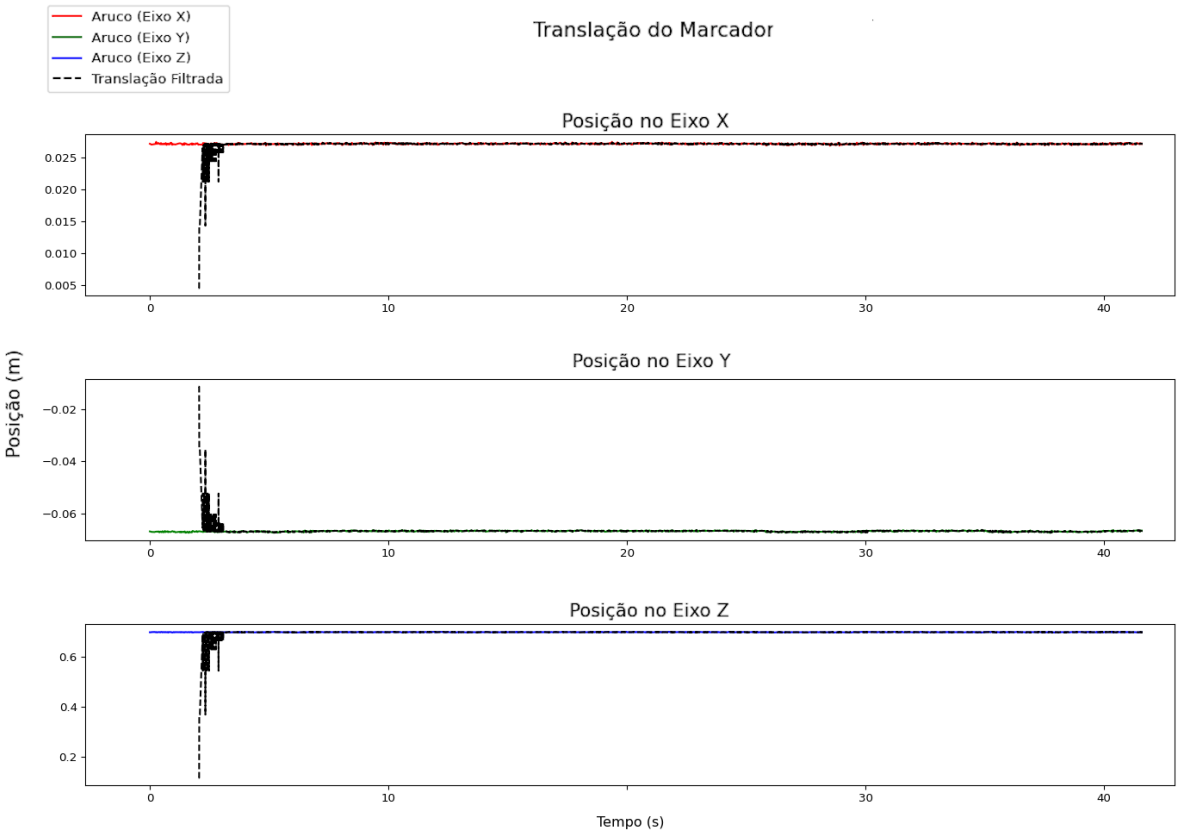
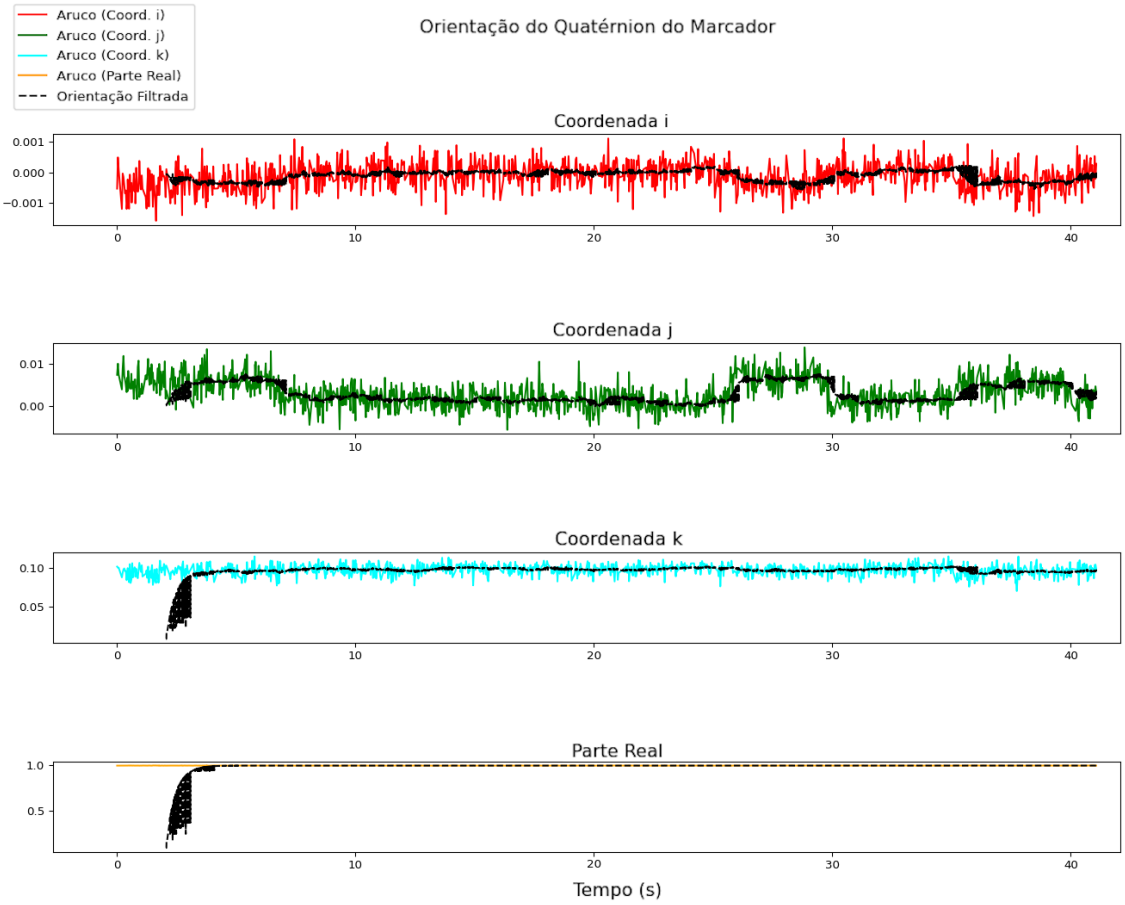
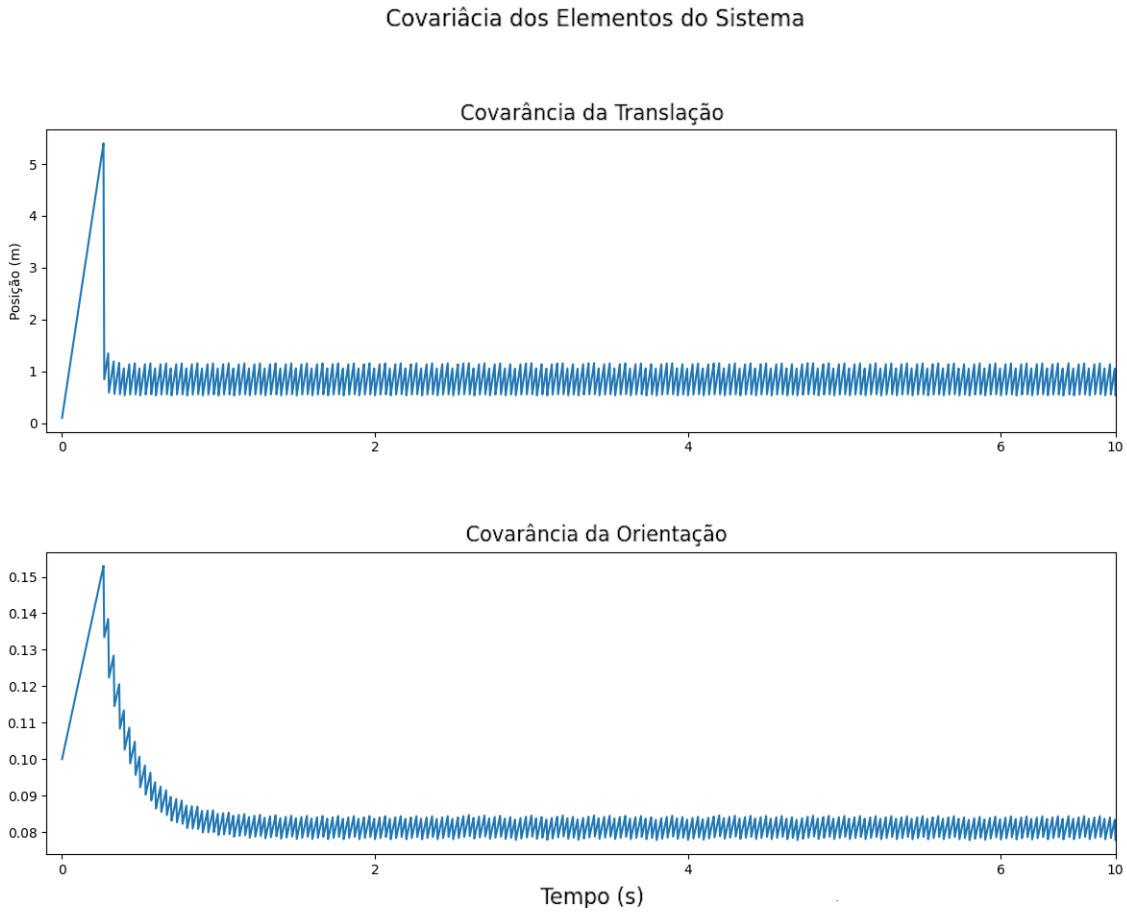


Figura 25 – Posição rotacional do marcador observado



A seguir, na figura 26, observa-se o comportamento dos elementos translacional e rotacional da matriz de covariâncias P do sistema. Nos testes realizados e apresentados neste documento, percebeu-se um padrão serializado do comportamento das variáveis de covariância ao longo do progresso do sistema, de modo que para todas as entidades, os elementos translacionais se comportaram da mesma forma, tanto para câmeras quanto marcadores, e o mesmo para os elementos rotacionais. Portanto, para simplificar a análise dos resultados. A variância translacional do sistema se mantém em uma faixa próxima de 1 metro, advinda majoritariamente como consequência dos elementos unitários das matrizes Q, multiplicada por um fator de 0.1 para os termos lineares e 0.001 para termos rotacionais e R, sendo considerado unitário.

Figura 26 – Covariâncias P translacional e rotacional do sistema



Por fim, os dados podem ser resumidos como apresentado na tabela 1. Aqui, são comparados os dados reais medidos com uma régua, com os dados obtidos pelo sistema e filtrados, bem como suas variâncias.

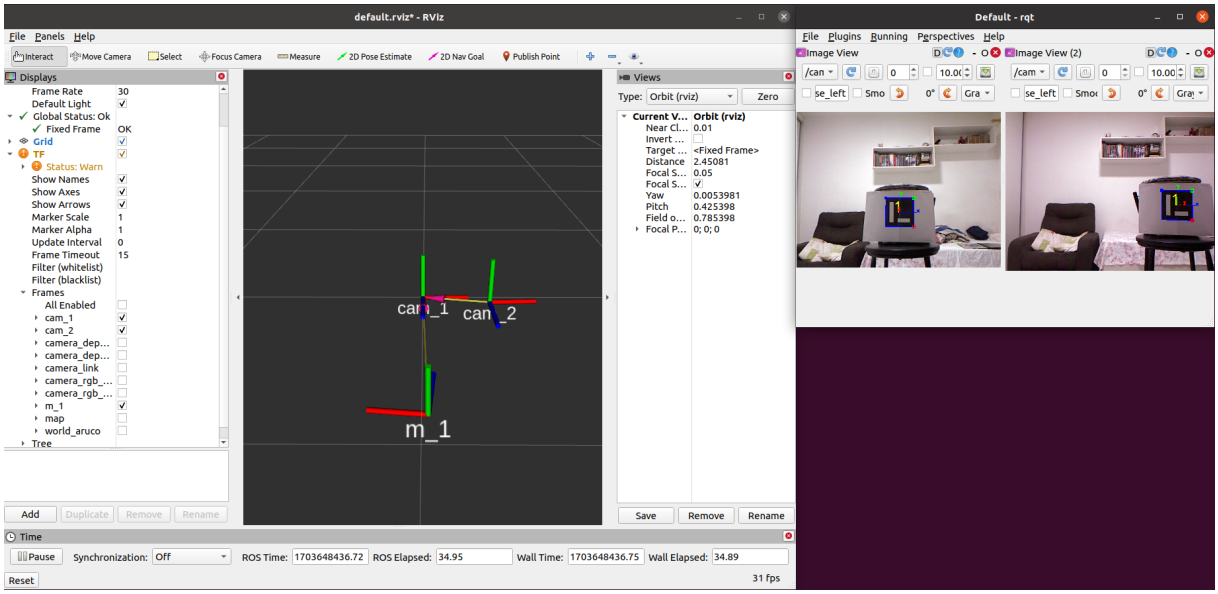
	Experimento 1 - 1 Câmera e 1 Marcador		
	Valor Medido	Valor Médio do Filtro	Covariância
t_x	0,03 m	0,027 m	0,843 m
t_y	-0,005 m	-0,067 m	0,843 m
t_z	0,75 m	0,696 m	0,843 m
q_x	0	0,0001	0,082
q_y	0	0,0029	0,082
q_z	0	0,0966	0,082
q_w	1	0,9881	0,082

Tabela 1 – Resumo de dados do experimento 1

4.3.2 Teste de 2 câmeras e 1 marcador, com a segunda câmera inserida e movida durante teste

Para este teste, o sistema foi inicializado com um marcador e após alguns instantes, uma segunda câmera a 30 cm da primeira câmera foi destampada e pôde observar o marcador. Por fim, após aproximadamente 38 segundos, a segunda câmera foi movimentada mais 20 cm a se distanciar da câmera 1 no eixo Z. O objetivo é avaliar como o sistema responde a alterações na posição e percepção de uma das câmeras, tanto na própria pose da câmera quanto na pose do marcador observado.

Figura 27 – Ambiente de teste observado



Percebe-se que o mesmo inicia com um sistema de comportamento bem estável. Também observa-se uma dificuldade do filtro de definir com a mesma acurácia que antes a posição do marcador a partir deste ponto, dado que agora há duas informações de posição para o marcador, enquanto o filtro busca manter o valor oscilando entre ambos tentando a convergência. A seguir, nas figuras 28 e 29, são apresentados os mesmos dados para a posição da segunda câmera, obtida a partir da pose do marcador coletado por ambas as câmeras. Por fim, a covariância do sistema é retratada pela figura 32. Destaca-se a resposta do sistema à movimentação no eixo Z da segunda câmera, e como o sistema conseguiu compreender a posição do marcador como parada durante esse processo, dado que a posição da câmera é calculada com base no marcador observado em comum entre as câmeras referencial e secundária. Também observa-se um maior número de dados pelo tempo de teste resultando em um gráfico de covariâncias mais denso, contudo, ainda convergindo para o mesmo valor.

Figura 28 – Posição translacional do marcador observado

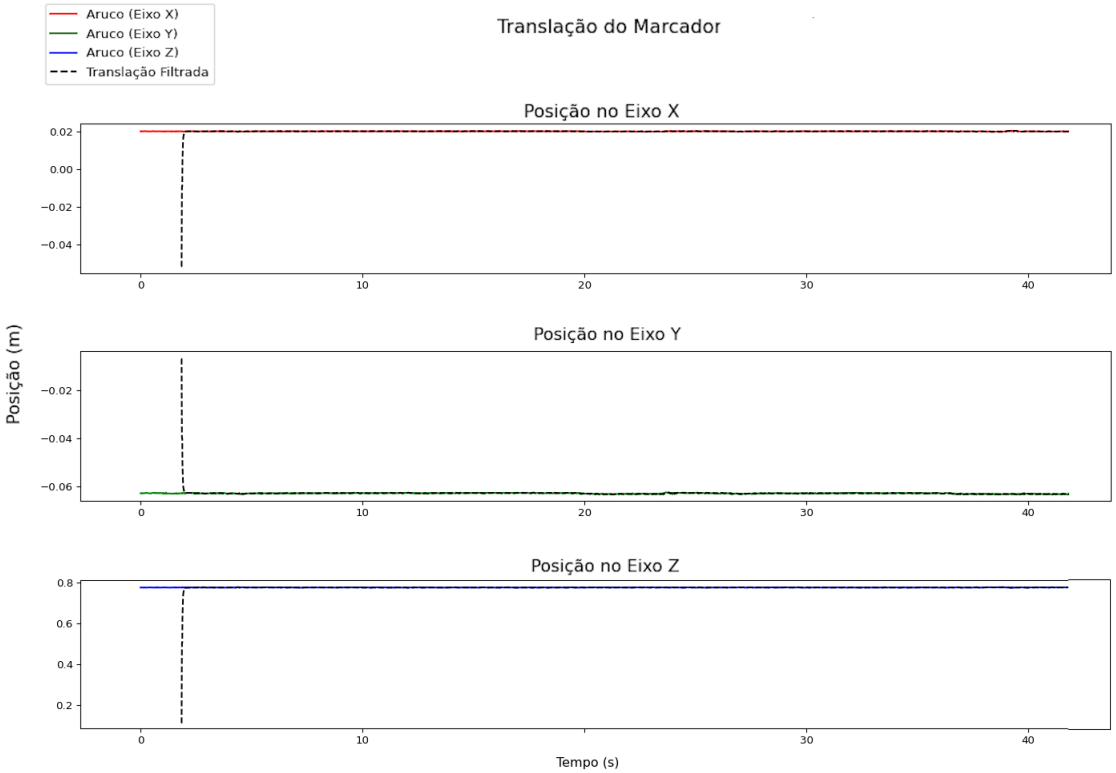


Figura 29 – Posição rotacional do marcador observado

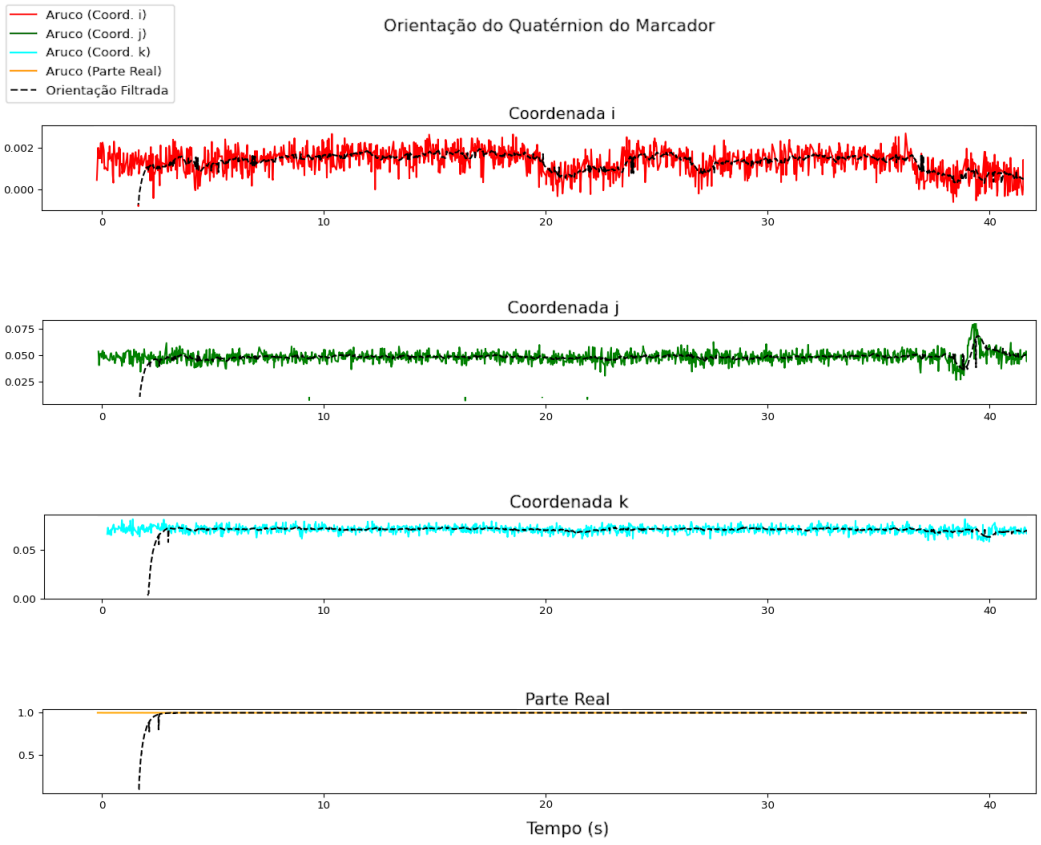


Figura 30 – Posição translacional da câmera calculada

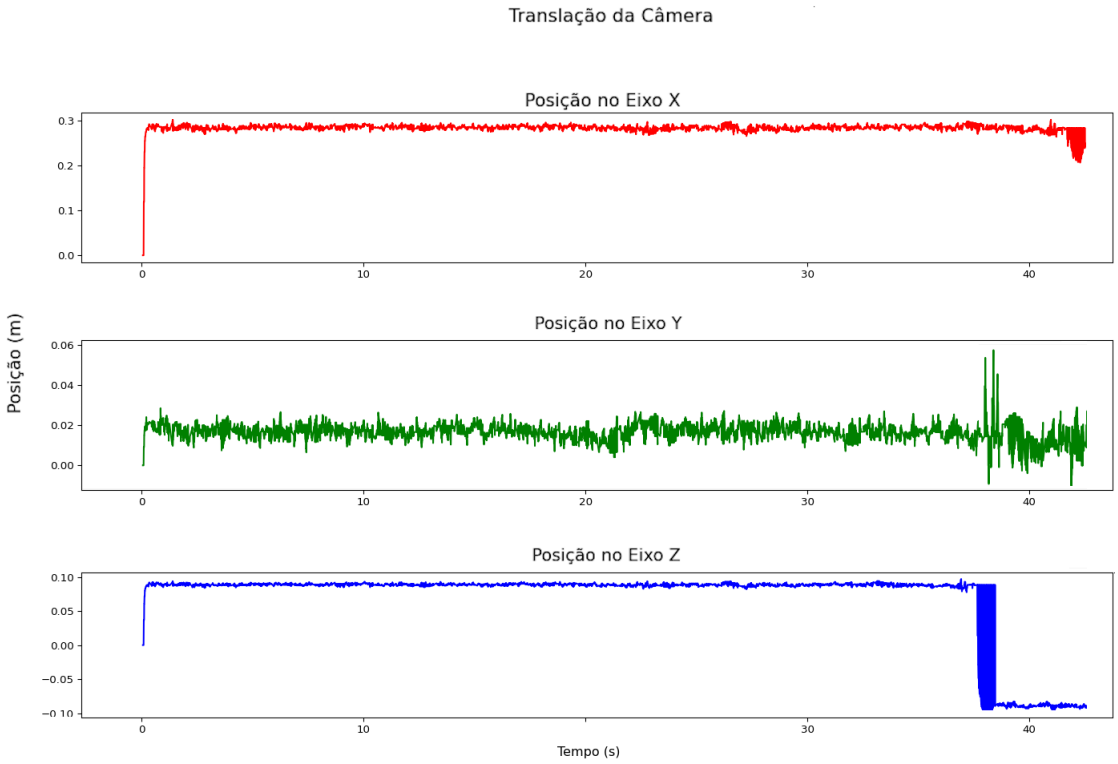
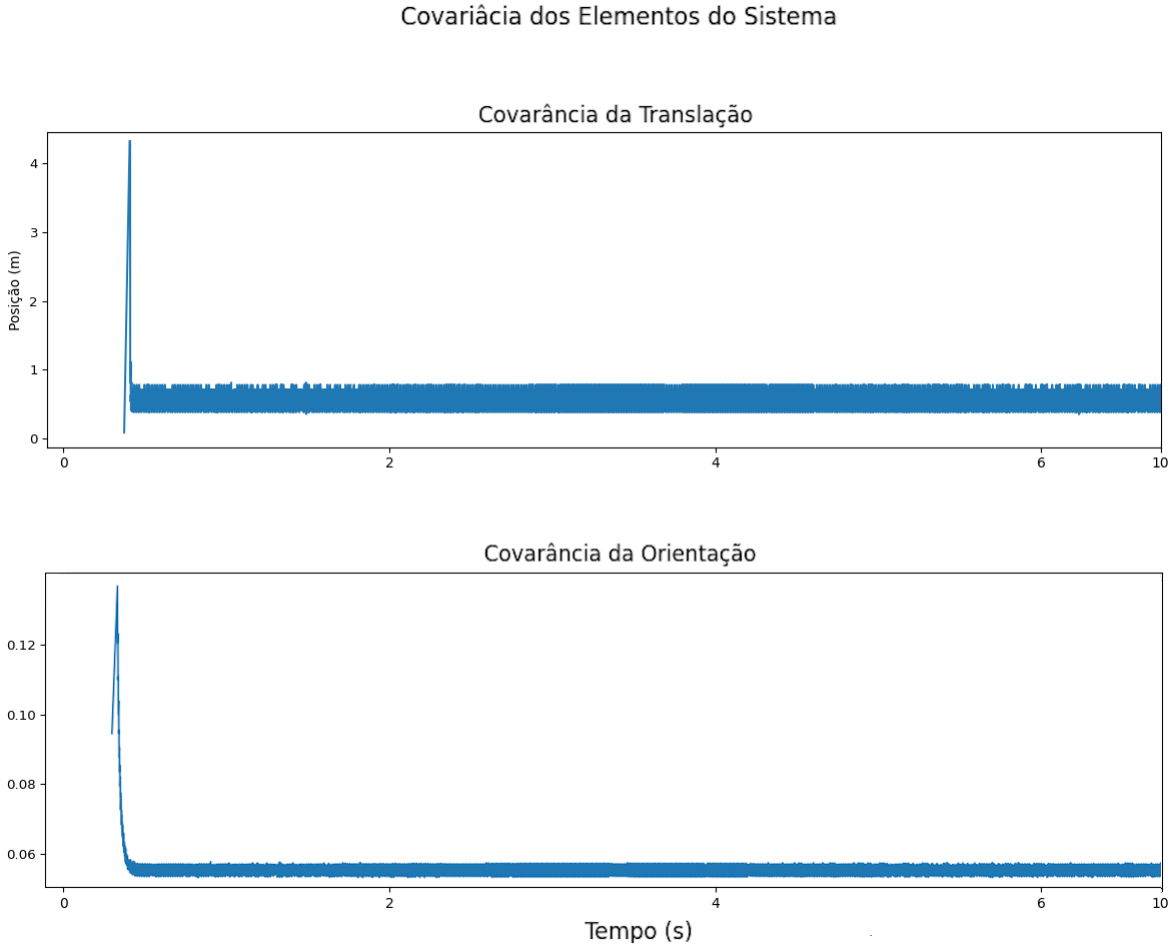


Figura 31 – Posição rotacional da câmera calculada

Figura 32 – Covariâncias P translacional e rotacional do sistema



Por fim, os dados podem ser resumidos como apresentado na tabela 2. Aqui são comparados os dados reais medidos com uma régua, com os dados obtidos pelo sistema e filtrados, bem como suas variâncias.

Experimento 2 - 2 Câmeras e 1 Marcador

	Marcador (ID 1)		Câmera 2		
	Valor Medido	Valor Filtrado	Valor Medido	Valor Filtrado	Covariância
t_x	0,03 m	0,020 m	0,29 m	0,248 m	0,570 m
t_y	-0,005 m	-0,063 m	0 m	0,015 m	0,570 m
t_z	0,72 m	0,772 m	-0,21 m	-0,069 m	0,570 m
q_x	0	0,0014	0	0,0197	0,058
q_y	0	0,00465	0	0,0353	0,058
q_z	0	0,0791	0	-0,0081	0,058
q_w	1	0,9886	1	0,9991	0,058

Tabela 2 – Resumo de dados do experimento 2

4.3.3 Teste de 2 câmeras e 2 marcadores

O próximo teste foi realizado para avaliar o comportamento de um sistema multi-câmeras e multi-marcadores, com duas de cada entidade. Analogamente ao experimento anterior, observa-se os dados para o primeiro marcador, de identificador 100, nas figuras 33 e 34. Para tais dados, percebe-se o mesmo comportamento para um sistema de uma única câmera e marcador, com os dados sendo filtrados e estabilizados, tanto para a parte translacional quanto rotacional.

Figura 33 – Posição translacional do marcador 100

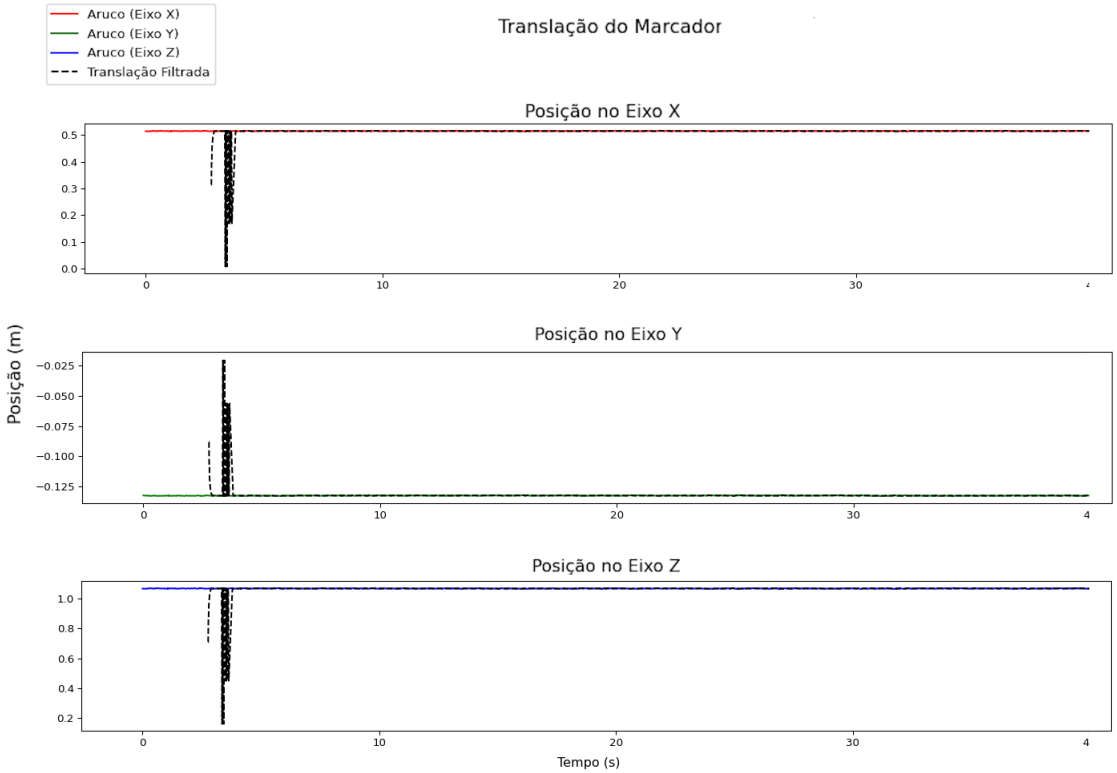
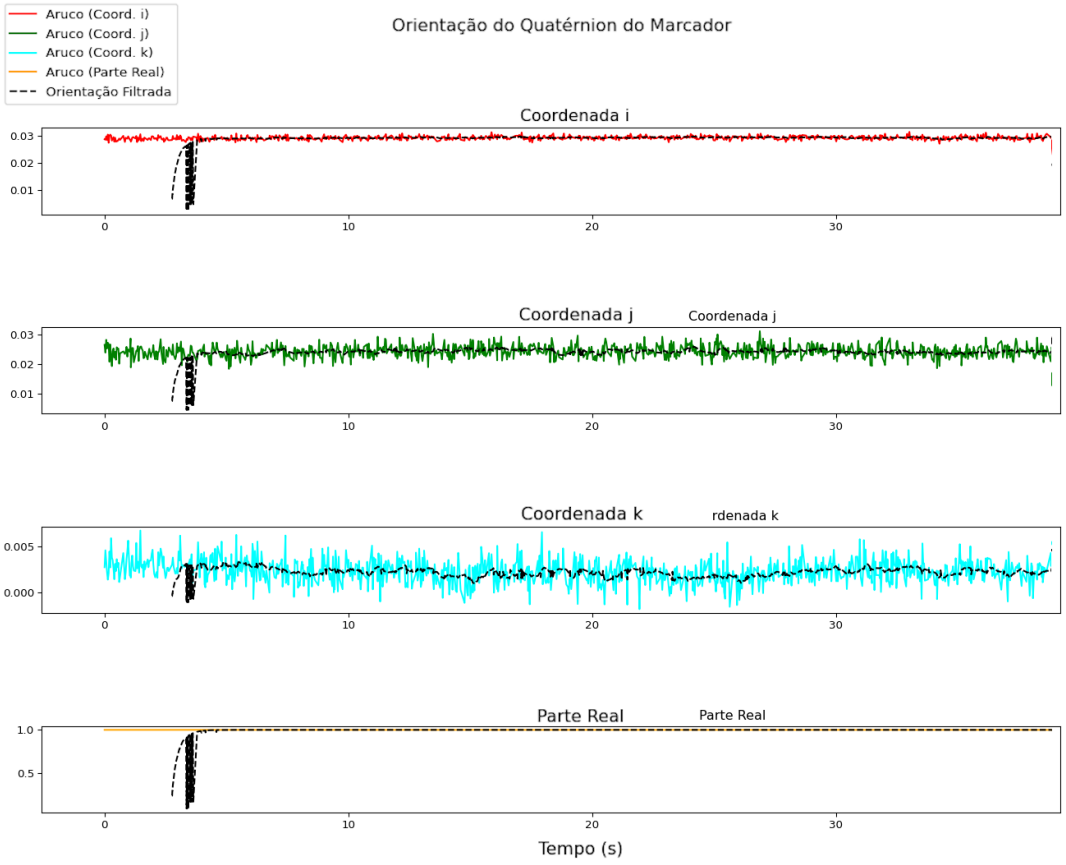


Figura 34 – Posição rotacional do marcador 100



A seguir, são apresentados os dados obtidos para o segundo marcador, de identificador 1, pelas figuras 36 e 37, bem como o ambiente de testes retratado pela figura 35.

Figura 35 – Ambiente de teste observado

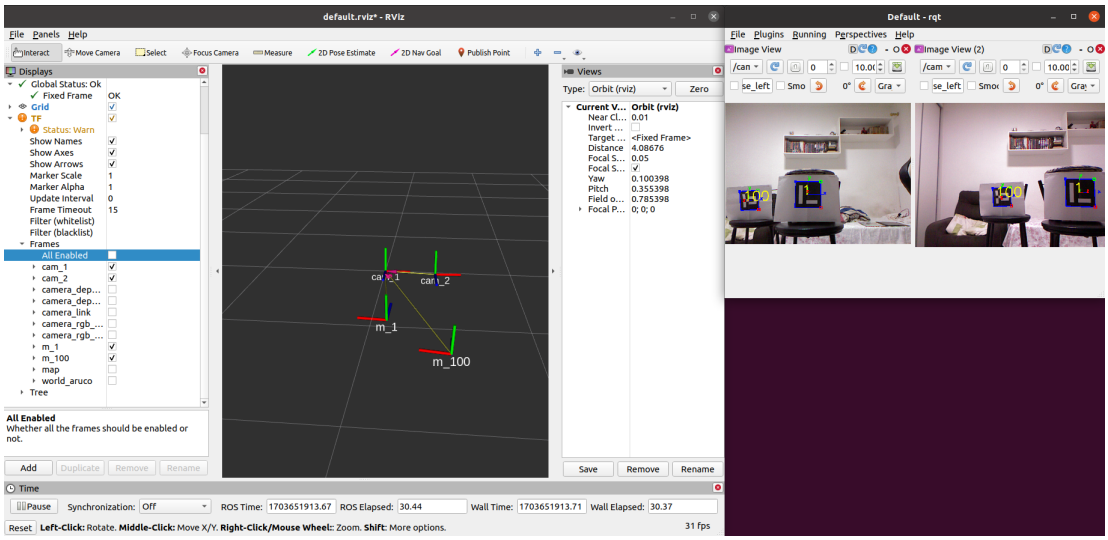


Figura 36 – Posição translacional do marcador 100

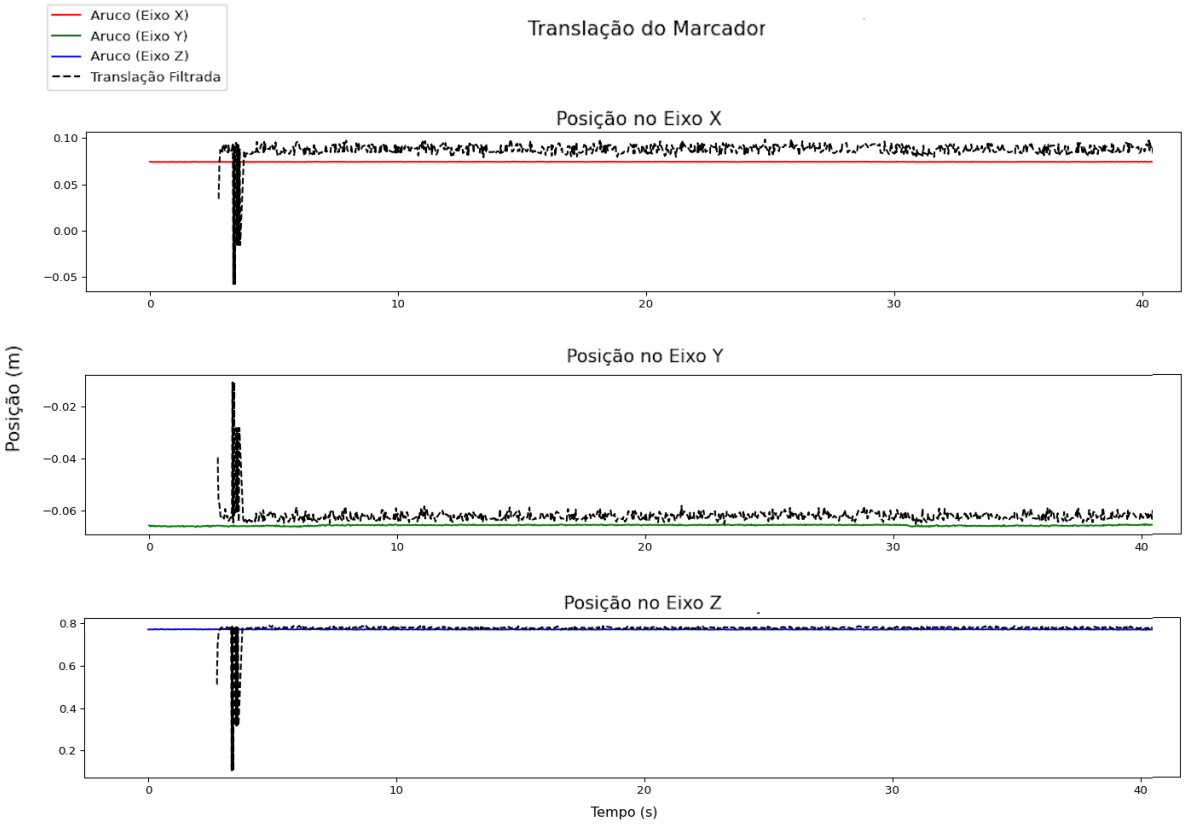
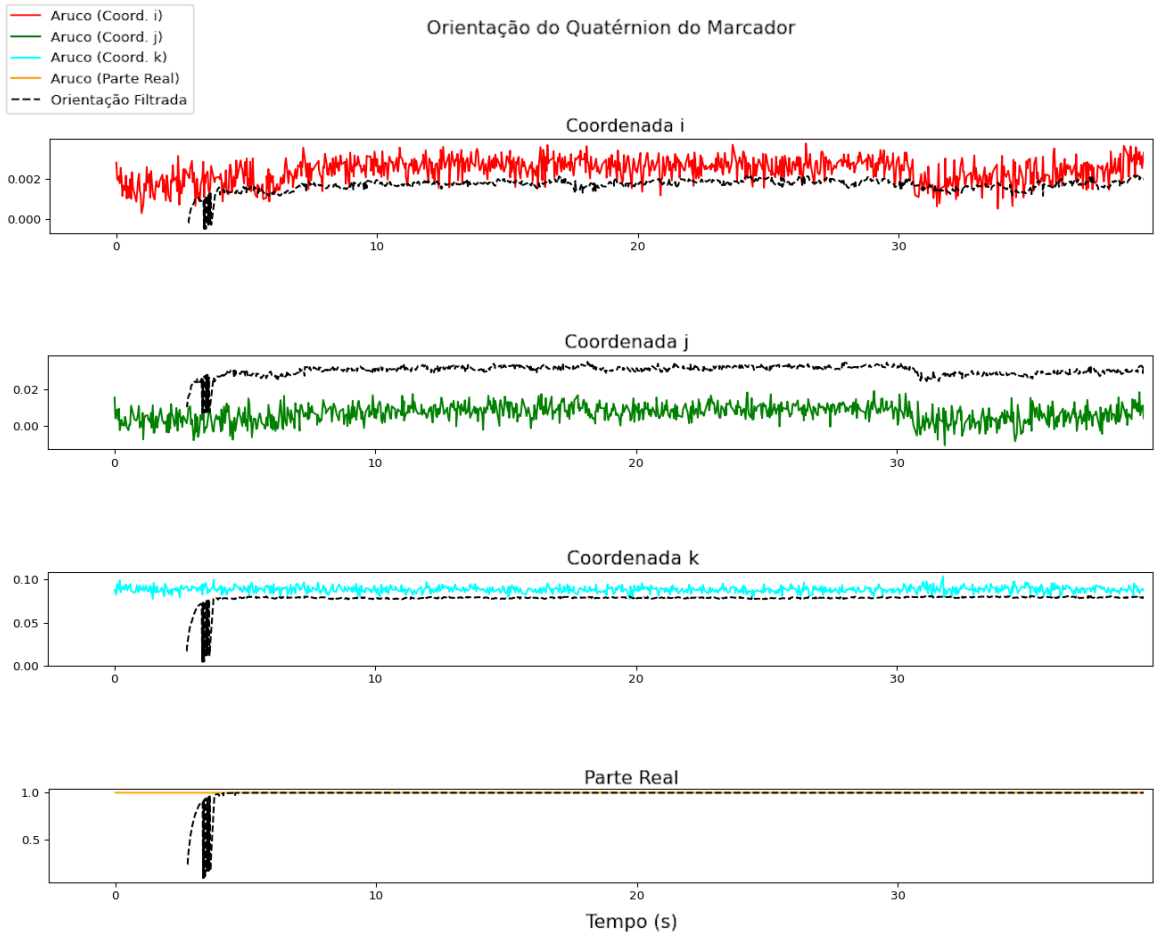


Figura 37 – Posição rotacional do marcador 1



Para este conjunto de dados, observa-se uma dificuldade do filtro de convergir, considerando que é demonstrado no gráfico apenas os dados obtidos de uma das câmeras, o comportamento observado é tal que a pose obtida pela outra câmera diferente consideravelmente da pose do marcador pela primeira câmera, localizada na origem. O filtro busca contornar essa diferença entre posições constante com um valor intermediário entre as medições, contudo, pela variância modelada, a confiança que se dá para os dados obtidos é alta a ponto de gerar instabilidade. Uma das formas de corrigir seria aumentando a variância de dados obtidos, modelado pela matriz Q.

A seguir, nas figuras 38 e 39, são apresentados os mesmos dados para a posição da segunda câmera, obtida a partir da pose do marcador coletado por ambas as câmeras. Percebe-se que a inconstância dos dados para a segunda câmera pro marcador, propagam também, como esperado, para a pose da câmera que, apesar de razoavelmente condizente com a sua localização física, observado pela interface RVIZ, ainda assim possui grande variação posicional. Percebe-se que esse erro é maior no eixo Y, podendo ser também causado por uma imperfeição do processo de calibração.

Figura 38 – Posição translacional da câmera calculada

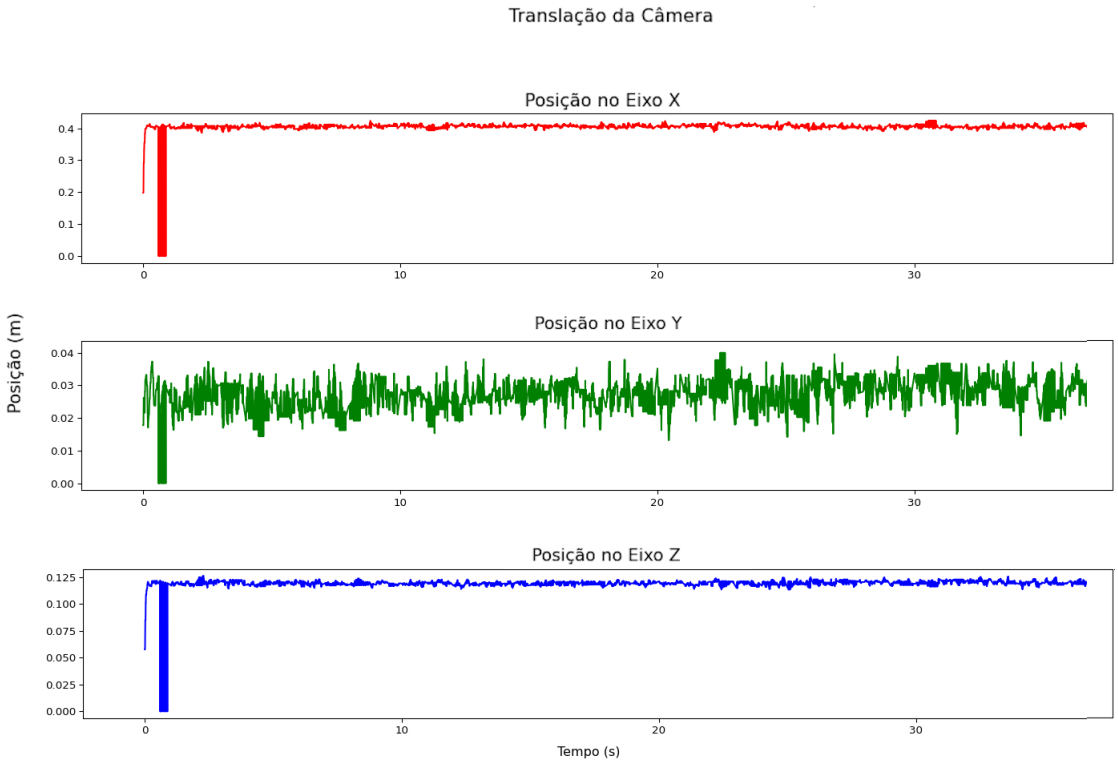
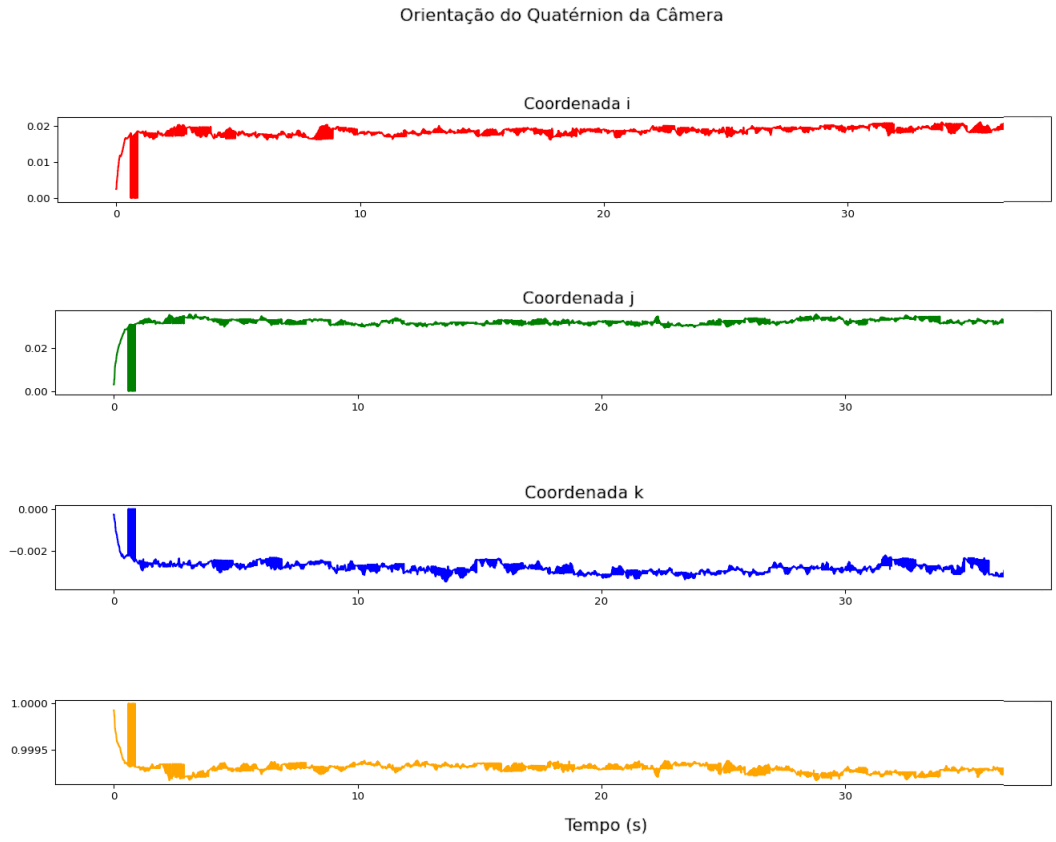
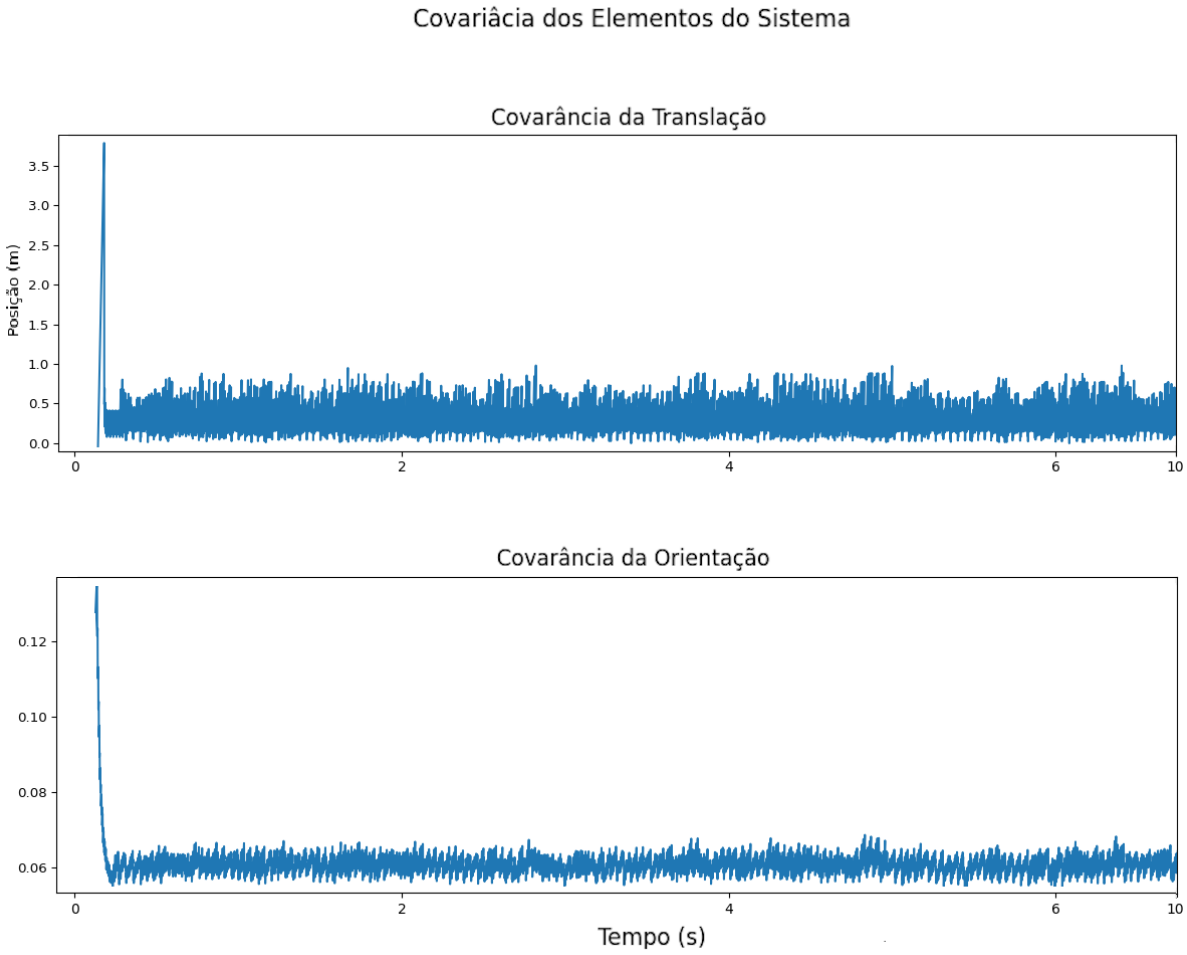


Figura 39 – Posição rotacional da câmera calculada



A covariância do sistema é retratada pela figura 40 e retrata uma oscilação levemente maior do que para os experimentos anteriores, denotando um pequeno efeito nela com base nos dados obtidos e calculados, mas se atendo majoritariamente à modelagem predita para P com base em Q e R. Por fim, os dados podem ser resumidos como apresentado na tabela 3.

Figura 40 – Covariâncias P translacional e rotacional do sistema



	Experimento 3 - 2 Câmeras e 2 Marcadores						
	Marcador (ID 1)		Marcador (ID 100)		Câmera 2		
	Medido	Filtrado	Medido	Filtrado	Medido	Filtrado	Covariância
t_x	0,03 m	0,0088 m	0,38 m	0,513 m	0,29 m	0,406 m	0,659 m
t_y	-0,005 m	-0,062 m	-0,005 m	-0,133 m	0 m	0,028 m	0,659 m
t_z	0,72 m	0,775 m	0,98 m	1,064 m	0 m	0,119 m	0,659 m
q_x	0	0,0017	0	0,0291	0	0,0185	0,063
q_y	0	0,0307	0	0,0242	0	0,0316	0,063
q_z	0	0,0781	0	-0,0021	0	-0,0028	0,063
q_w	1	0,9858	1	0,9890	1	0,9931	0,063

Tabela 3 – Resumo de dados do experimento 3

4.3.4 Teste de 2 câmeras e 2 marcadores, com alta covariância de erro de medição

O mesmo experimento foi realizado novamente, contudo, desta vez buscou-se aumentar a covariância R do sistema, buscando avaliar como o filtro tenta contornar as dificuldades de observação e filtragem de posições diferentes para um mesmo objeto. Nas figuras 41 e 42, percebe-se que houve uma melhora na estabilização da posição, como esperado, ao realizar o *tuning* do filtro, porém, o tempo de acomodação do sinal é considerado extenso dada a demora de aproximadamente 5 segundos para a estabilização da posição translacional de um marcador, e o dobro para rotacional.

Esse fenômeno de melhor estabilização pode ser percebido também na forma como o filtro lida com as posições concorrentes para o segundo marcador, como apresentado pelas figuras 43 e 44. Ademais, também percebe-se o mesmo tempo de acomodação para o sinal, apesar de sua convergência possuir um erro constante do início ao fim do experimento.

Por fim, seguem as posições para a segunda câmera, nas figuras 45 e 46, e o comportamento da covariância, em figura 47. Nestes casos, percebe-se um fenômeno de degraus para cada as posições de quatérnion, sendo causado possivelmente por uma oscilação frequente na orientação da câmera, como uma consequência do ruído apresentado pela filtragem dos marcadores.

Figura 41 – Posição translacional do marcador 100

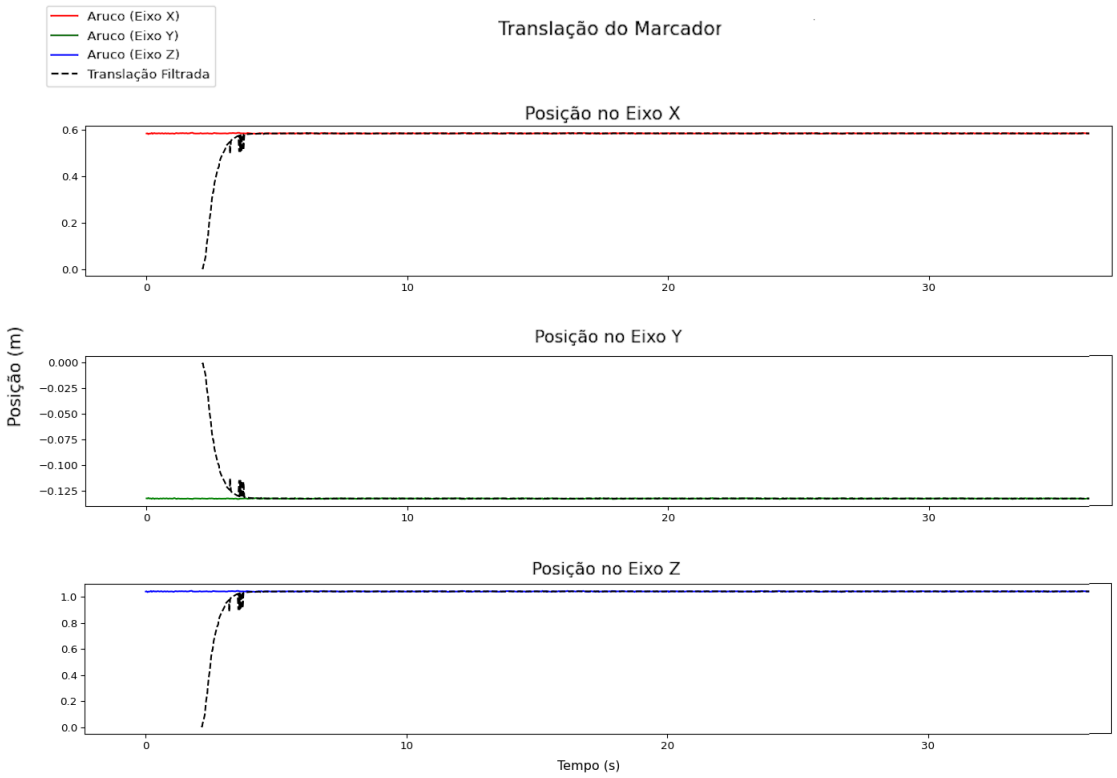


Figura 42 – Posição rotacional do marcador 100

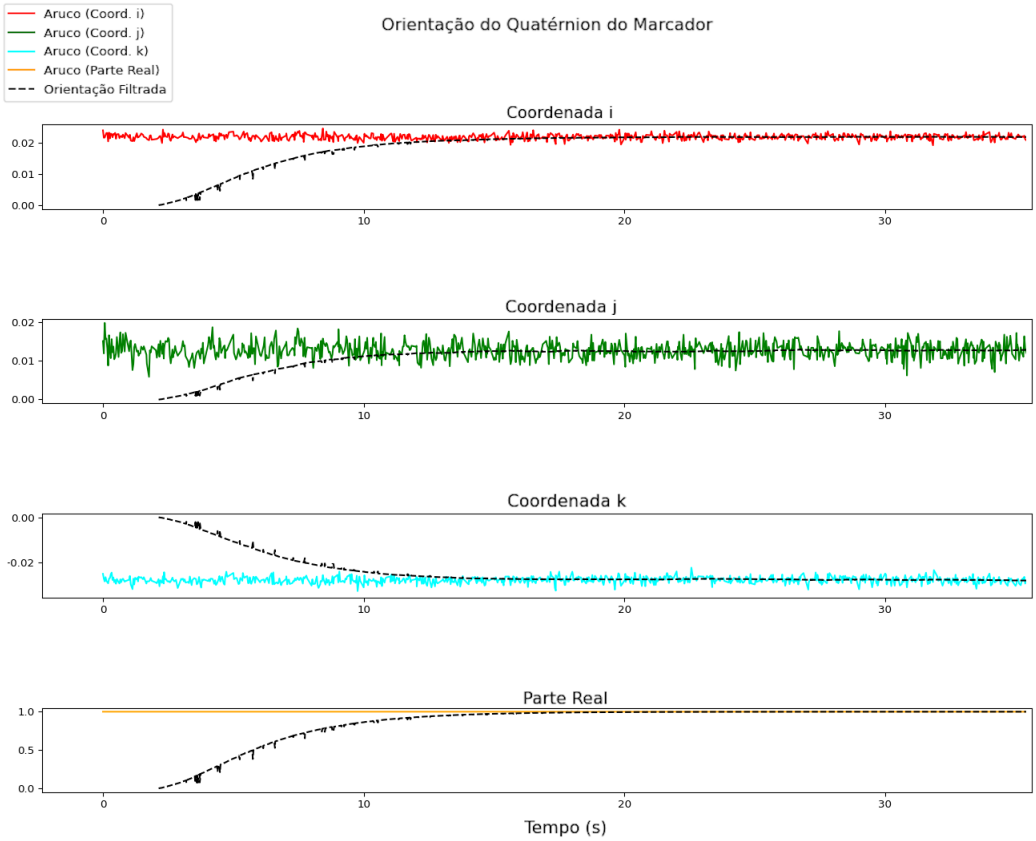


Figura 43 – Posição translacional do marcador 1

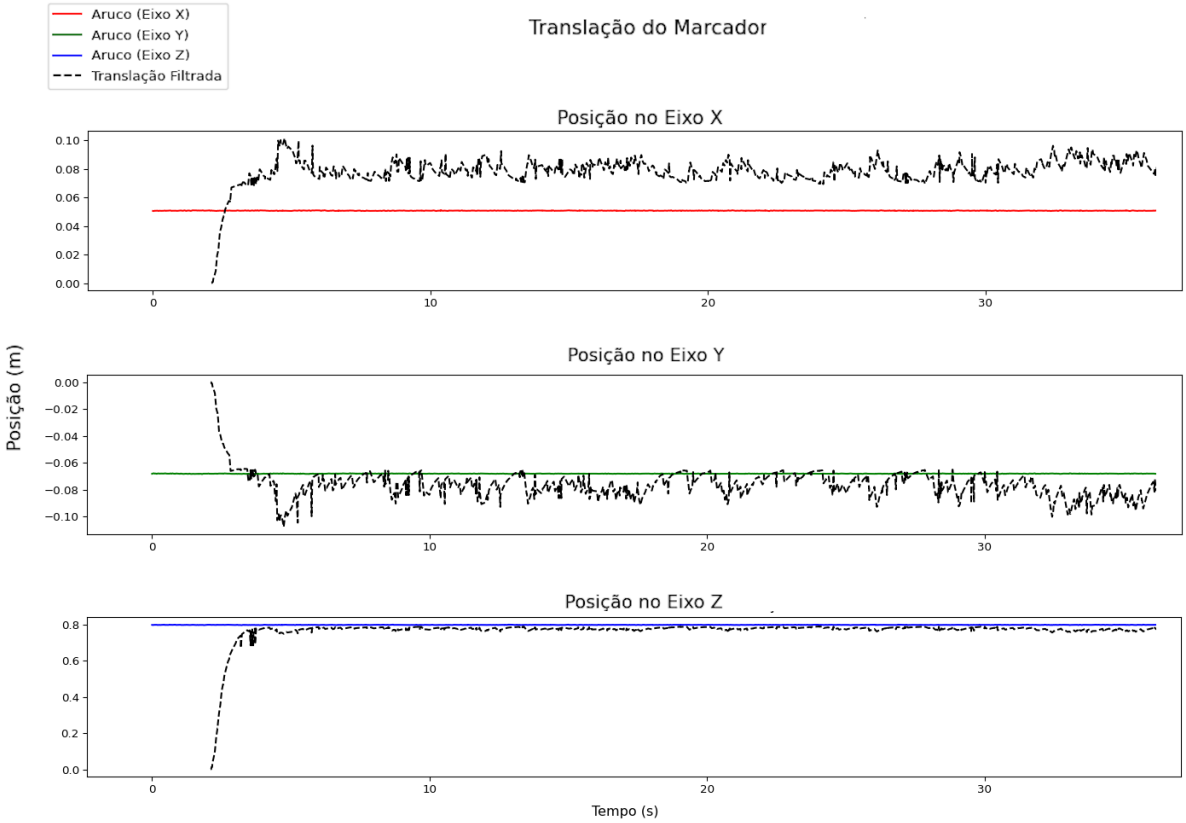


Figura 44 – Posição rotacional do marcador 1

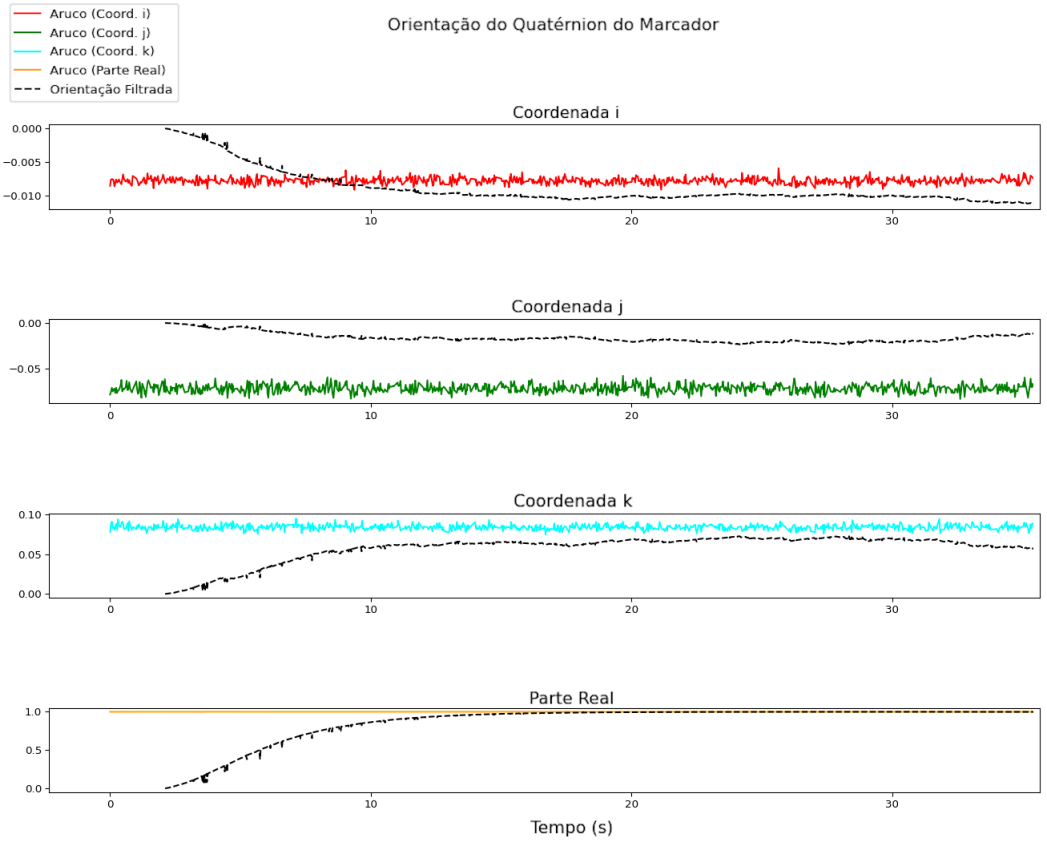


Figura 45 – Posição translacional da câmera calculada

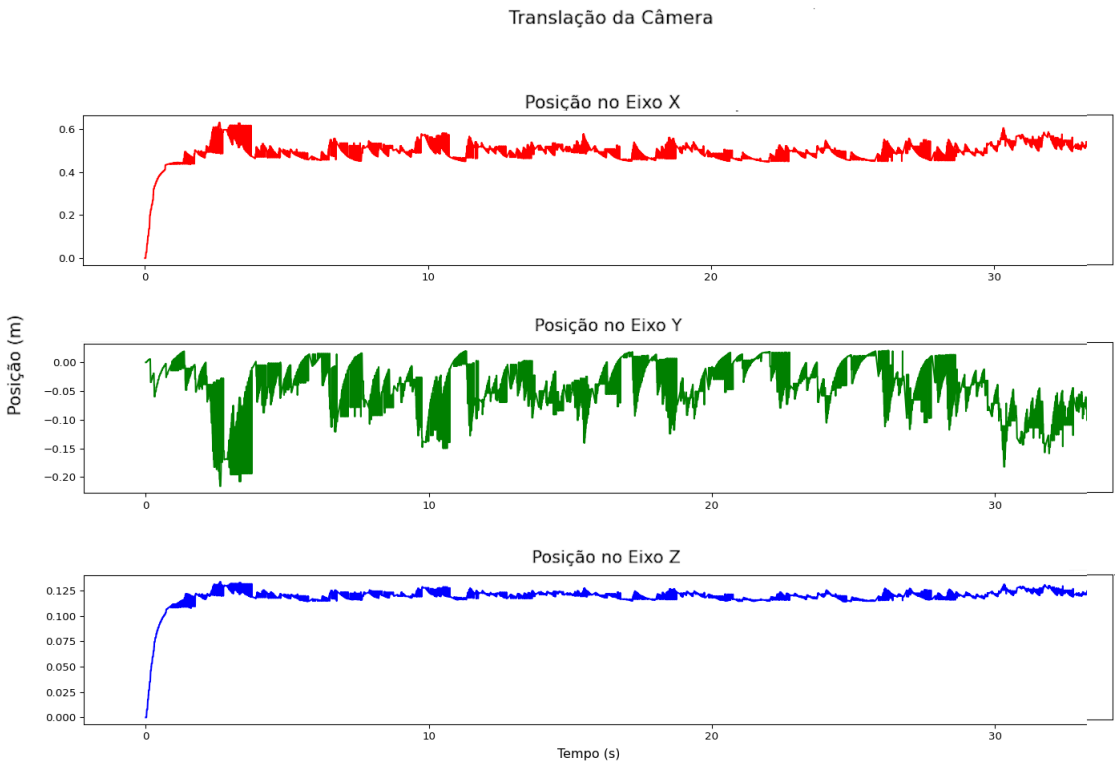
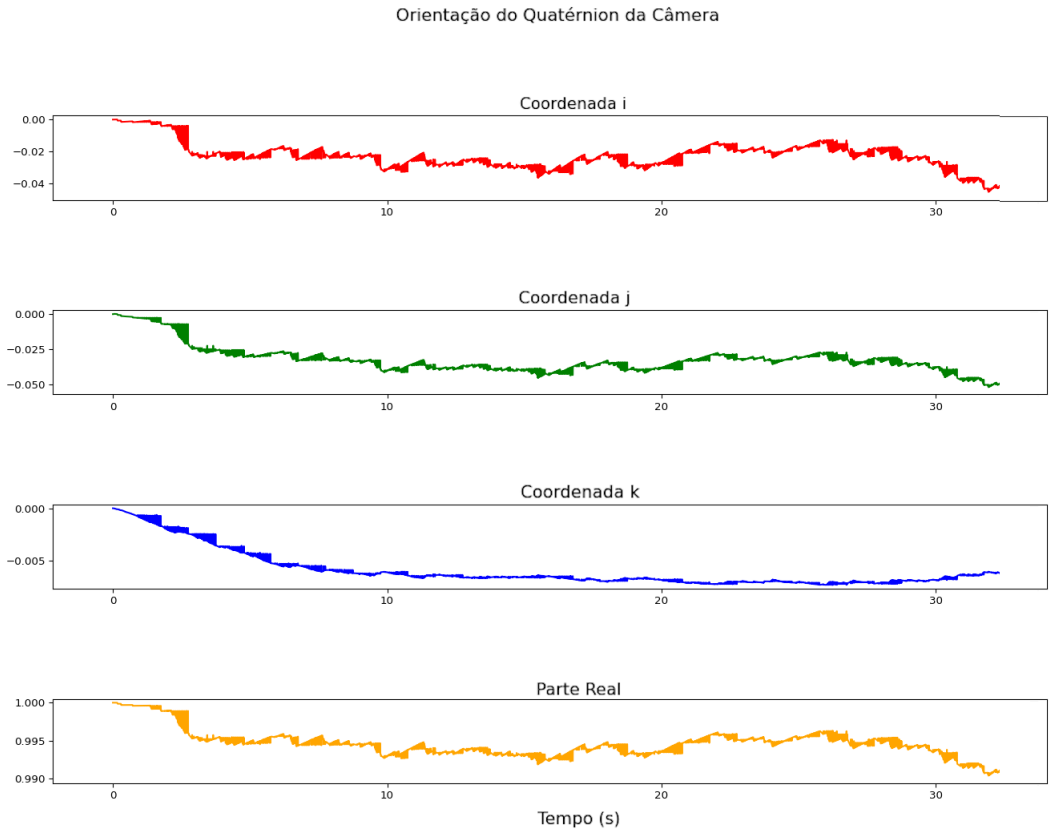
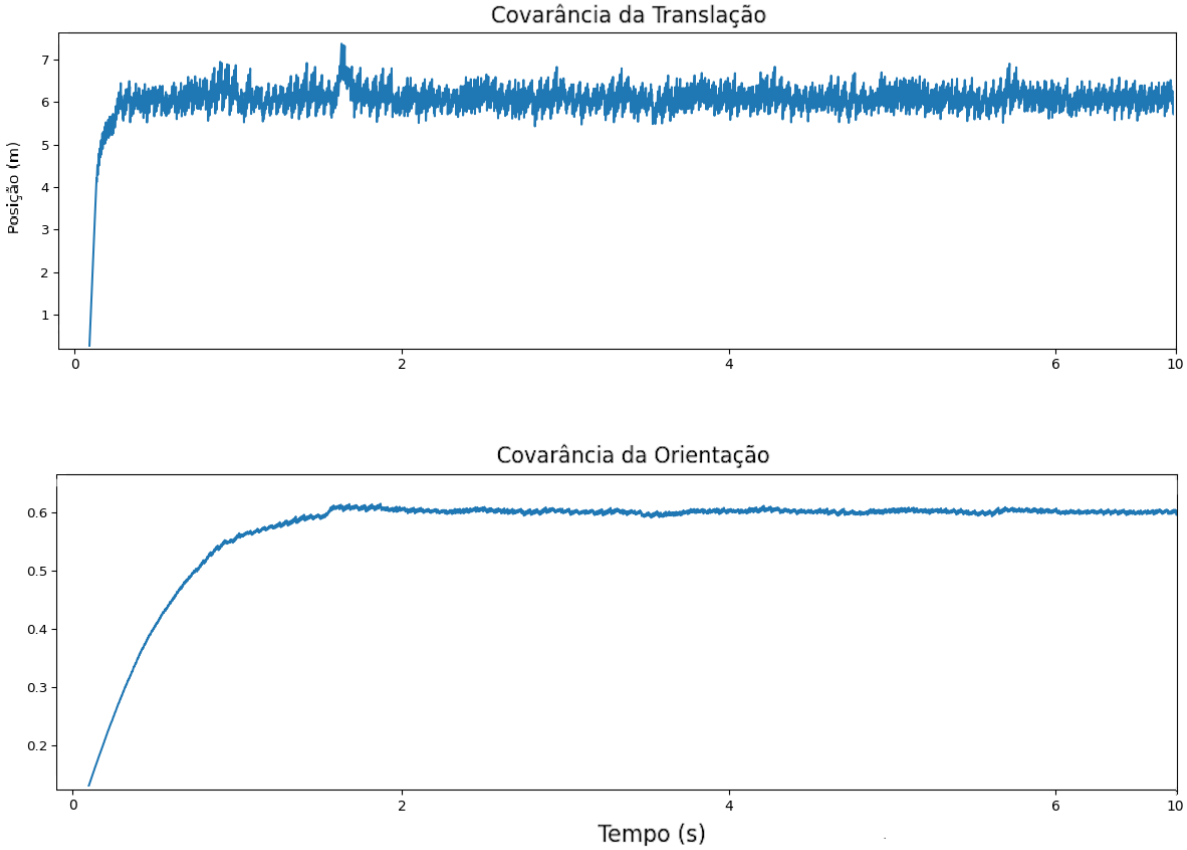


Figura 46 – Posição rotacional da câmera calculada



Para a figura 47 apresentada a seguir, percebe-se que o termo de P é incrementado consideravelmente, o que é esperado pela sua composição no processo de atualização conforme as novas matrizes e pelo efeito do ganho de Kalman, de modo a buscar uma representação mais acurada da variância dos estados.

Figura 47 – Covariâncias P translacional e rotacional do sistema



Por fim, os dados obtidos são resumidos e apresentados na tabela 4

	Experimento 4 - 2 Câmeras e 2 Marcadores (Alta Covariância)						
	Marcador (ID 1)		Marcador (ID 100)		Câmera 2		
	Medido	Filtrado	Medido	Filtrado	Medido	Filtrado	Covariância
t_x	0,03 m	0,078 m	0,37 m	0,577 m	0,29 m	0,495 m	6,254 m
t_y	-0,005 m	-0,076 m	-0,005 m	-0,131 m	0 m	-0,043 m	6,254 m
t_z	0,72 m	0,764 m	0,95 m	1,027 m	0 m	0,119 m	6,254 m
q_x	0	0,0093	0	0,0200	0	-0,0255	0,6074
q_y	0	0,0161	0	0,0110	0	-0,0354	0,6074
q_z	0	0,0588	0	-0,0250	0	-0,0061	0,6074
q_w	1	0,8876	1	0,8943	1	0,9940	0,6074

Tabela 4 – Resumo de dados do experimento 4

5 Conclusões

O projeto realizado e apresentado neste documento foi o de uma infraestrutura de localização e comunicação de informações espaciais entre robôs e sensores, visando a atuação integrada e automatizada, permitindo mais segurança e estabilidade. Essa interação se dá por câmeras do tipo *pinhole* do tipo Logitech e Kinect, e marcadores ArUco, e permitem uma maior capacidade de localização das instâncias envolvidas. O presente projeto de localização desenvolvido demonstrou coerência com a modelagem de velocidade nula para um sistema multi-câmeras, com capacidade de avaliar com erros máximos de 20cm (para o pior cenário de modelagem de variâncias na casa dos 6 metros) a posição de marcadores e de demais câmeras através de marcadores observados em comum, e replicar da forma condizente espacialmente dentro do ambiente virtual desenvolvido Rviz, contudo, para uma modelagem que gere sinais mais suaves e adequados para sistemas de controle, se faz necessária uma modelagem de velocidade ou aceleração do filtro de Kalman utilizado, tendo em vista que os sinais filtrados obtidos apresentaram dificuldade de se ater à mudanças bruscas na posição dos marcadores e câmeras.

Como possíveis próximos passos no desenvolvimento do sistema, os processos imediatos se dariam em duas vertentes: a primeira envolve a integração no sistema de controle de ambos os manipuladores atuantes no LARA, o UR3 e o Meka, de modo que ambos possam interagir entre si ao garantir que possuam nós em seus sistemas ROS internos que se inscrevam nos nós que publicam as poses obtidas pelas câmeras, e portanto, que possam reconhecer o ambiente que os contorna, e através de um sistema de compreensão que relacione a sua respectiva posição a um marcador de referência, possam publicar sua pose via cinemática direta; A segunda envolve a melhoria do presente sistema, alterando a modelagem dinâmica pela sua representação no espaço de estados e também pela complexidade do filtro. Para facilitar e melhorar medições relacionadas a objetos em movimento, faz-se necessário o uso de filtros como o Filtro de Kalman *Unscented*, ou o Filtro de Kalman Estendido, sendo o primeiro a melhor recomendação dada a sua menor carga computacional e menor dificuldade de implementação por não exigir o trabalho com elementos derivativos. Também pode-se otimizar o presente sistema com mudanças nas estruturas de dados para reduzir os dados armazenados e os processos realizados conforme a necessidade da aplicação. Além disso, pode-se buscar melhorias no sistema através de maior integração com diferentes tipos de câmera, bem como inserir a possibilidade de inserção e remoção de câmeras conectadas em tempo de execução, exigindo um trabalho de melhor integração com o sistema operacional.

Referências

AUTOMATE. **Unimate, The First Industrial Robot**. 2021. Disponível em: <https://www.automate.org/a3-content/joseph-engelberger-unimate>. Citado na p. 15.

BORGES, N. O. Desenvolvimento de API para sistema de localização e mapeamento, 2021. DOI: https://doi.org/10.1016/j.patcog.2014.01.005. Disponível em: <https://bdm.unb.br/handle/10483/30577>. Citado nas pp. 19, 32, 45–47.

CESARINO, P. S. D. N. MODELAGEM, IDENTIFICAÇÃO E DESIGN DE CONTROLE PARA O ROBÔ MANIPULADOR UR3. Dez. 2020. Citado na p. 19.

CORKE, P. Cham: Springer International Publishing, 2023. ISBN 978-3-031-06469-2. DOI: 10.1007/978-3-031-06469-2. Disponível em: <https://doi.org/10.1007/978-3-031-06469-2>. Citado nas pp. 22–28, 32, 50.

FILHO, V. W. **Reestruturação produtiva e acidentes de trabalho no Brasil: estrutura e tendências**. 1. ed. São Paulo, SP, 1999. Citado na p. 15.

GARCIA, P. ESTIMAÇÃO DE FORÇA DE INTERAÇÃO NO BRAÇO ROBÓTICO UR3. Dez. 2019. Citado na p. 19.

GARRIDO-JURADO, S.; MUÑOZ-SALINAS, R.; MADRID-CUEVAS, F.; MARÍN-JIMÉNEZ, M. Automatic generation and detection of highly reliable fiducial markers under occlusion. **Pattern Recognition**, v. 47, n. 6, p. 2280–2292, 2014. ISSN 0031-3203. DOI: https://doi.org/10.1016/j.patcog.2014.01.005. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0031320314000235>. Citado na p. 32.

INTERNATIONAL ORGANIZATION FOR STANDARDIZATION. **ISO/TS 15066:2016**: Robots and robotic devices — Collaborative robots. Fev. 2016. P. 33. Citado na p. 15.

JAZWINSKI, A. H. **Stochastic Processes and Filtering Theory**. New York, NY: Academic Press, 1970. Citado nas pp. 36–38, 42, 43.

KAILATH, T. A view of three decades of linear filtering theory. **IEEE Transactions on Information Theory**, v. 20, p. 146–181, 1974. Citado na p. 40.

KALAITZAKIS, M.; CAIN, B.; CARROLL, S.; AMBROSI, A.; WHITEHEAD, C.; VITZILAIOS, N. Fiducial Markers for Pose Estimation: Overview, Applications and Experimental Comparison of the ARTag, AprilTag, ArUco and STag Markers. eng. **Journal of intelligent robotic systems**, Springer Netherlands, Dordrecht, v. 101, n. 4, 2021. ISSN 0921-0296. Citado nas pp. 16, 32.

LEPETIT, V.; MORENO-NOGUER, F.; FUA, P. EPnP: An Accurate O(n) Solution to the PnP Problem. eng. **International journal of computer vision**, Springer US, Boston, v. 81, n. 2, p. 155–166, 2009. ISSN 0920-5691. Citado na p. 33.

MATOS, R. R. de. Interfaceamento amigável para robôs manipuladores. 2020. Citado na p. 20.

MUÑOZ-SALINAS, R.; MARÍN-JIMENEZ, M. J.; YEGUAS-BOLIVAR, E.; MEDINA-CARNICER, R. **Mapping and Localization from Planar Markers**. 2017. arXiv: 1606.00151 [cs.CV]. Citado na p. 32.

OPROMOLLA, R.; VELA, C.; NOCERINO, A.; LOMBARDI, C. Monocular-Based Pose Estimation Based on Fiducial Markers for Space Robotic Capture Operations in GEO. eng. **Remote sensing (Basel, Switzerland)**, MDPI AG, Basel, v. 14, n. 18, p. 4483, 2022. ISSN 2072-4292. Citado na p. 16.

ROBERTS, L. G. Computer vision: the last 50 years. eng. Massachusetts Institute of Technology, Massachusetts, 1963. Citado na p. 16.

SHAPIRO, L. G. Computer vision: the last 50 years. eng. **International journal of parallel, emergent and distributed systems**, Taylor Francis, Abingdon, v. 35, n. 2, p. 112–117, 2020. ISSN 1744-5760. Citado na p. 16.

STEPHEN SAGERS, R. P. Mechanics of a Digital Camera. 2010. Disponível em: <https://digitalcommons.usu.edu/cgi/viewcontent.cgi?article=2466&context=extension_curall>. Acesso em: 19 nov. 2023. Citado na p. 27.

VIDERE DESIGN. **STH-MDCS3-VAR(X)(-C) Stereo Head User's Manual**. English. Set. 2007. Citado na p. 30.

YOON JAE KIM HYUNG SEOK NAM, e. a. Vision-aided brain–machine interface training system for robotic arm control and clinical application on two patients with cervical spinal cord injury. **BioMed Eng OnLine 18**, p. 14, fev. 2019. Citado na p. 15.

ZHANG, X.; NAVAB, N. Tracking and pose estimation for computer assisted localization in industrial environments. eng. In: PROCEEDINGS Fifth IEEE Workshop on Applications of Computer Vision. IEEE, 2000. P. 214–221. ISBN 0769508138. Citado na p. 16.

Apêndices

APÊNDICE A – Códigos de programação

A.1 Código do nó do ROS para o filtro de Kalman

Código A.1 – Código em C++

```cpp
#include <ros/ros.h>
#include "ros_filter.h"
#include "filter_variables.h"

int main(int argc, char** argv) {
    ros::init(argc, argv, "kalman_aruco");

    RosFilter filter;

    ROS_INFO_STREAM("Kalman filter on!");
    filter.getCameraTopics();
    filter.subscribeTopics();
    //filter.insertCameraBasis(3); //is private function
    ros::Timer timer = filter.createTimer(ros::Duration(0.005));
    ros::spin();

    return 0;
}
```

A.2 Código de integração entre visão e filtragem

Código A.2 – Código em C++

```cpp
#include <tf/transform_broadcaster.h>
#include <vector>
#include <algorithm>
#include <iostream>
#include <fstream>
#include <cmath>
#include <filesystem>

#pragma once

#include <map>

#include <ros/ros.h>
#include <ros/package.h>
```

```
15
16 #include <aruco_msgs/MarkerArray.h>
17 #include <geometry_msgs/PoseArray.h>
18 #include <geometry_msgs/PoseWithCovariance.h>
19
20 #include "filter_variables.h"
21 #include "kalman_filter.hpp"
22
23 #define DEBUG true
24
25 class RosFilter {
26     public:
27         RosFilter();
28
29         // Find all aruco marker publisher topics for each camera
30         void getCameraTopics();
31
32         // Subscribe for all topics found by getCameraTopics and
               created the publishers
33         void subscribeTopics();
34
35         // Create Timer responsible for update the state of the
               system
36         ros::Timer createTimer(ros::Duration period);
37
38         // Insert a camera on the system for fututre use on the
               filter
39         void insertCameraBasis(int camera_id);
40
41     private:
42         // The Kalman filter itself
43         filter kf;
44
45         // Node handle responsible for communication with ROS
               processes
46         ros::NodeHandle nh;
47
48         // Map of marker list on the system with cameras on the
               system
49         std::map<int, std::vector<trackedMarker>> cam_markers;
50
51         // Publisher for filtered marker data
52         ros::Publisher filtered_marker_publisher;
53
54         // Publisher for camera data
55         ros::Publisher camera_publisher;
56
57         // List of camera topics
58         std::vector<std::string> topics;
59
60         // Subscribers for each camera topic
61         std::map<int, ros::Subscriber> camera_subs;
```

```
62
63          // Map of cameras tracked by the system and their info
64          std::map<int, cameraBasis> camera_poses;
65
66          // number of poses that are tracked in the system (there
                may be multiple version of the same aruco tag for
                different cameras)
67          // used for new indexes in state vector calculation
                without the need to iterate over cam_markers for all
                cameras
68          int tracked_poses = 0; // the same as stateVector size
                (from the filter)
69
70          // Saves the last observation published by aruco ros node
                of each camera and correlate to the camera that
                observes it
71          std::map<int, std::shared_ptr<aruco_msgs::MarkerArray>>
                last_msgs;
72
73          // Create the filtered markers publishers and store the
                references in a map
74          void createPublisherFiltered(std::string
                filtered_marker_topic, ros::NodeHandle nh);
75
76          // Create the camera poses publisher
77          void createPublisherCameras(std::string camera_topic,
                ros::NodeHandle nh);
78
79          // Extract the camera ID from the topic name (assuming
                "cam_x" format)
80          int extractCameraID(const std::string& camera_topic);
81
82          // Receives a msg with a pose, process it and convert it
                to eigen::vector
83          Eigen::VectorXd msgToPose(geometry_msgs::Pose pose);
84
85         // Converts a given pose from Eigen::VectorXd  to
              poseTransform
86          poseTransform poseToTf(Eigen::VectorXd pose);
87
88          // Converts a given pose from poseTransform to
                Eigen::VectorXd
89          Eigen::VectorXd tfToPose(poseTransform transform);
90
91          // Evaluate if the marker is already tracked by te system
                and updates it. If not, creates it
92          void insertUpdateMarker(aruco_msgs::Marker marker, int
                camera_id);
93
94          // Insert new marker state variables on the filter and
                updates camera poses
```

```
95          void insertPosesOnStateVector(Eigen::VectorXd& newState,
                int cam_id);
96
97          // Extracting a marker data from the state vector.
                Overload dedicated to markers
98          aruco_msgs::Marker extractDataFromState(trackedMarker
                data, Eigen::VectorXd filtered_states, Eigen::MatrixXd
                covariance_matrix);
99
100         // Extracting a camera data from the state vector.
                Overload dedicated to cameras which dont return
                covariances (since ros doesnt has a message for vector
                of poseWithCovariance)
101         geometry_msgs::Pose extractDataFromState(cameraBasis data,
                Eigen::VectorXd filtered_states);
102
103         // Prepare the data to publish them properly
104         void preparePublishData(geometry_msgs::PoseArray&
                camera_poses_msg, aruco_msgs::MarkerArray&
                filtered_markers_msg);
105
106         // Publish filter data on ROS topics
107         void publishData(const geometry_msgs::PoseArray
                camera_poses_msg, const aruco_msgs::MarkerArray
                filtered_markers_msg);
108
109         // Callback to handle marker data from cameras
110         void cameraCallback(const
                aruco_msgs::MarkerArray::ConstPtr& msg, const int&
                camera_id);
111
112         // Callback to handle filter update event
113         void timerCallback(const ros::TimerEvent& event);
114
115         // Resize the state vector based on the number of objects
                detected
116         void resizeState();
117
118         // Resize the state vector based on the number of objects
                detected, and inserts if there is a base data
119         void resizeState(Eigen::VectorXd newData);
120
121         // Saves the timestamp and filter covariance main diagonal
                on a csv file on the readings/filtered folder
122         void saveCovariance();
123
124 };
125
126 RosFilter::RosFilter() {
127     // Prepare the filter by defining the initial dimension
128     kf.set_dimensions(0);
```

```
129     std::filesystem::remove("./readings/filtered/covariance.csv");
           //Clears file covariance file
130     ROS_INFO("Starting kalman filter aruco node");
131 }
132
133 void RosFilter::getCameraTopics() {
134     ros::master::V_TopicInfo master_topics;
135     ros::master::getTopics(master_topics);
136     std::string
           marker_topic_name("/aruco_marker_publisher/markers");
137
138     for (ros::master::V_TopicInfo::iterator it =
           master_topics.begin() ; it != master_topics.end(); it++) {
139         const ros::master::TopicInfo& info = *it;
140         if(info.name.find(marker_topic_name) != std::string::npos){
141             if(info.name.find("list") == std::string::npos){  //
                   filters "markers_list" topics from the vector
142                 topics.push_back(info.name);
143             }
144         }
145     }
146 }
147
148 void RosFilter::subscribeTopics() {
149     // Create universal publisher for all camera poses
150     createPublisherCameras("kalman/filtered_cam_poses", nh);
151
152     // Create publisher for filtered marker data
153     createPublisherFiltered("kalman/filtered_markers_poses", nh);
154
155     for (const std::string& camera_topic : topics) {
156         int camera_id = extractCameraID(camera_topic);
157
158         insertCameraBasis(camera_id);
159
160         camera_subs[camera_id] =
               nh.subscribe<aruco_msgs::MarkerArray>(
161             camera_topic, 10,
                   boost::bind(&RosFilter::cameraCallback, this, _1,
                   camera_id));
162     }
163 }
164
165 ros::Timer RosFilter::createTimer(ros::Duration period) {
166     return nh.createTimer(period, &RosFilter::timerCallback, this);
167 }
168
169 void RosFilter::insertCameraBasis(int camera_id) {
170     // Verifying if this camera is already tracked. If not, insert
           in the system
171     poseTransform tf = poseTransform(tf::Quaternion(0,0,0,1),
           tf::Vector3(0,0,0)); // Transformation to convert rviz axis
```

```
        representation to aruco
172     if(!camera_poses.contains(camera_id)) {
173         cameraBasis aux((tracked_poses++) * POSE_VECTOR_SIZE,
174                         tfToPose(tf),
175                         Eigen::MatrixXd::Identity(POSE_VECTOR_SIZE,
                                POSE_VECTOR_SIZE),
176                         poseTransform());
177
178         if(camera_id != 1) {
179             aux.covariance *= HIGH_COVARIANCE_PRESET;
180         }
181         camera_poses[camera_id] = aux;
182         resizeState(tfToPose(tf));
183     }
184 }
185
186 void RosFilter::createPublisherFiltered(std::string
       filtered_marker_topic, ros::NodeHandle nh) {
187     filtered_marker_publisher =
           nh.advertise<aruco_msgs::MarkerArray>(filtered_marker_topic,
           5);
188 }
189
190 void RosFilter::createPublisherCameras(std::string camera_topic,
       ros::NodeHandle nh) {
191     camera_publisher =
           nh.advertise<geometry_msgs::PoseArray>(camera_topic, 5);
192 }
193
194 int  RosFilter::extractCameraID(const std::string& camera_topic) {
195     std::size_t found = camera_topic.find("cam_");
196     if (found != std::string::npos) {
197         std::string aux = camera_topic.substr(found + 4); //
              Extract the ID part
198         try {
199             return std::stoi(aux);
200         } catch(const std::invalid_argument& e){
201             return -1;
202         }
203     }
204     return -1;
205 }
206
207 Eigen::VectorXd RosFilter::msgToPose(geometry_msgs::Pose pose) {
208     poseTransform transform;
209     tf::Quaternion adjust_rotation, msg_orientation;
210     tf::Vector3 msg_pose;
211
212     msg_pose = tf::Vector3(pose.position.x,
213                            pose.position.y,
214                            pose.position.z);
215
```

```
216     msg_orientation = tf::Quaternion(pose.orientation.x,
217                                      pose.orientation.y,
218                                      pose.orientation.z,
219                                      pose.orientation.w);
220
221     transform = poseTransform(msg_orientation, msg_pose);
222     adjust_rotation = tf::Quaternion(0,0,1,0);   // Rotation on
           the z axis to adjust to our system from what we receive
           from aruco_ros
223     auto adjust_rotation_tf =  poseTransform(adjust_rotation);
224     transform = adjust_rotation_tf * transform;
225     return tfToPose(transform);
226 }
227
228 void RosFilter::insertUpdateMarker(aruco_msgs::Marker marker, int
       camera_id) {
229     int markerTagId = marker.id;
230     geometry_msgs::Pose pose = marker.pose.pose;
231
232     Eigen::VectorXd poseVector = msgToPose(pose);
233
234     // Verifying if this marker is already tracked by any camera,
           if not, insert in the system
235     int marker_index = -1;
236     int marker_found_camera_id;
237
238     for(const auto [id, marker_list] : cam_markers){
239         auto it = std::find_if(marker_list.begin(),
               marker_list.end(),
240                                [markerTagId](const trackedMarker&
                                   tracked_marker){
241                                    return tracked_marker.arucoId ==
                                       markerTagId;
242                                });
243         if(it != marker_list.end()) {
244             marker_index = std::distance(marker_list.begin(), it);
245             marker_found_camera_id = id;
246             break;
247         }
248     }
249     std::cout << " find camera " << camera_id << " found: " <<
           marker_found_camera_id << std::endl;
250
251     for(auto it : cam_markers){
252         std::cout << "map camera " << it.first << std::endl;
253         for(auto jt : it.second){
254             std::cout << "marker: " << jt.arucoId << std::endl;
255         }
256     }
257     if(marker_index == -1) {
258         // The marker is new
259         trackedMarker auxMarker;
```

```
260             auxMarker.arucoId = markerTagId;
261             auxMarker.stateVectorAddr = (tracked_poses++) *
                    POSE_VECTOR_SIZE;
262             auxMarker.pose = poseVector;
263             auxMarker.covariance =
                    Eigen::MatrixXd::Identity(POSE_VECTOR_SIZE,
                    POSE_VECTOR_SIZE) * HIGH_COVARIANCE_PRESET;
264
265             cam_markers[camera_id].push_back(auxMarker);
266         }
267         else if(marker_found_camera_id == camera_id) {
268             // The marker is already tracked by this camera
269             trackedMarker& auxMarker =
                    cam_markers[camera_id][marker_index];
270
271             auxMarker.pose = poseVector;
272             //auxMarker.covariance = Eigen::MatrixXd::Identity() *
                    HIGH_COVARIANCE_PRESET;
273
274         }
275         else if(marker_found_camera_id){
276             // Marker is being tracked by another camera
277             // Updates the tf_previous from the camera based on marker
                    pose of the another camera with id lesser than this one
                    (otherwise it would fall on the first if)
278             if(marker_found_camera_id < camera_id){
279                 camera_poses[camera_id].updateTfPrevious(
                        poseToTf(poseVector),
                        poseToTf(cam_markers[marker_found_camera_id]
                        [marker_index].pose), marker_found_camera_id);
280             }
281             // Checks if the marker already exists in the camera
282             auto it = std::find_if(cam_markers[camera_id].begin(),
                    cam_markers[camera_id].end(),
283                                    [markerTagId](const trackedMarker&
                                           tracked_marker){
284                                        return tracked_marker.arucoId ==
                                               markerTagId;
285                                    });
286
287             if(it != cam_markers[camera_id].end()) {
288                 // The marker existd previously in the camera
289                 trackedMarker& auxMarker =
                        cam_markers[camera_id][std::distance(
                        cam_markers[camera_id].begin(), it)];
290                 auxMarker.pose = poseVector;
291                 //auxMarker.covariance = Eigen::MatrixXd::Identity() *
                        HIGH_COVARIANCE_PRESET;
292
293             } else {
294                 // The marker didnt exist previously in the camera
```

```
295             trackedMarker auxMarker =
                    cam_markers[marker_found_camera_id][marker_index];
296             auxMarker.pose = poseVector;
297             cam_markers[camera_id].push_back(auxMarker);
298         }
299     }
300 }
301
302 void RosFilter::insertPosesOnStateVector(Eigen::VectorXd&
    newState, int cam_id) {
303     std::vector<int> inserted_markers;
304
305     // Inserts camera and all it's markers observations into the
        state
306     int previous_id =  camera_poses[cam_id].previous_id;
307     auto previous_tf = camera_poses[cam_id].previous_tf.inverse();
308
309     camera_poses[cam_id].pose = tfToPose(previous_tf *
        poseToTf(camera_poses[previous_id].pose));
310     std::cout << "camera " << cam_id << " POSE: " <<
        camera_poses[cam_id].pose << std::endl;
311     newState.segment(camera_poses[cam_id].stateVectorAddr,
        POSE_VECTOR_SIZE) = camera_poses[cam_id].pose;
312
313     for(const auto& marker : cam_markers[cam_id]){
314         newState.segment(marker.stateVectorAddr, POSE_VECTOR_SIZE)
            = tfToPose( poseToTf(camera_poses[cam_id].pose) *
            poseToTf(marker.pose) );
315         inserted_markers.push_back(marker.arucoId);
316     }
317
318     // Insert other cameras and their obsevations into the state
319     for(const auto& [id, marker_list] : cam_markers) {
320         if(id != cam_id) {
321
322             newState.segment(camera_poses[id].stateVectorAddr,
                POSE_VECTOR_SIZE) = camera_poses[id].pose;
323
324             for(const auto& marker : marker_list) {
325                 // Checks if the marker isn't already inserted by
                    the current camera
326                 if(std::find(inserted_markers.begin(),
327                              inserted_markers.end(),
328                              marker.arucoId) ==
                                 inserted_markers.end()) {
329                     newState.segment(marker.stateVectorAddr,
                        POSE_VECTOR_SIZE) = tfToPose(
                        poseToTf(camera_poses[id].pose) *
                        poseToTf(marker.pose));
330                 }
331             }
332         }
```

```
333     }
334 }
335
336 aruco_msgs::Marker RosFilter::extractDataFromState(trackedMarker
       data, Eigen::VectorXd filtered_states, Eigen::MatrixXd
       covariance_matrix) {
337     aruco_msgs::Marker filtered_data;
338     filtered_data.id = data.arucoId;
339
340     // Extract the filtered data corresponding to this marker
341     if (data.stateVectorAddr + POSE_VECTOR_SIZE <=
           filtered_states.size()) {
342         Eigen::VectorXd data_pose =
               filtered_states.segment(data.stateVectorAddr,
               POSE_VECTOR_SIZE);
343
344         // Create a geometry_msgs::Pose message from the filtered
               pose data
345         geometry_msgs::Pose filtered_pose;
346         filtered_pose.position.x = data_pose(0);
347         filtered_pose.position.y = data_pose(1);
348         filtered_pose.position.z = data_pose(2);
349         filtered_pose.orientation.x = data_pose(3);
350         filtered_pose.orientation.y = data_pose(4);
351         filtered_pose.orientation.z = data_pose(5);
352         filtered_pose.orientation.w = data_pose(6);
353
354         filtered_data.pose.pose = filtered_pose;
355
356         // Populate the filtered_data.pose.covariance with the
               covariance matrix
357         filtered_data.pose.covariance.fill(0.0); // Initialize to
               zero
358
359         // Extract the corresponding covariance matrix for this
               marker
360         // adding the '-1' part is important because the
               covariance matrix is 7x7 and the last row and cols
               relate to the
361         // imaginary 'w' rotation which is not really meaningful,
               and the output is a 6x6 matrix by the format
362         // specified in geometry_msgs
363         if (data.stateVectorAddr + POSE_VECTOR_SIZE <=
               covariance_matrix.rows()  &&
364             data.stateVectorAddr + POSE_VECTOR_SIZE <=
                   covariance_matrix.cols() ) {
365             for (int i = 0; i < POSE_VECTOR_SIZE -1; ++i) {
366                 for (int j = 0; j < POSE_VECTOR_SIZE -1; ++j) {
367                     filtered_data.pose.covariance[i *
                           (POSE_VECTOR_SIZE -1) + j] =
368                         covariance_matrix(data.stateVectorAddr +
                               i, data.stateVectorAddr + j);
```

```
369                 }
370             }
371         }
372     }
373     return filtered_data;
374 }
375
376 geometry_msgs::Pose RosFilter::extractDataFromState(cameraBasis
    data, Eigen::VectorXd filtered_states) {
377     // Extract the filtered data corresponding to this marker
378     geometry_msgs::Pose filtered_pose;
379     if (data.stateVectorAddr + POSE_VECTOR_SIZE <=
        filtered_states.size()) {
380         Eigen::VectorXd data_pose =
            filtered_states.segment(data.stateVectorAddr,
            POSE_VECTOR_SIZE);
381
382         // Create a geometry_msgs::Pose message from the filtered
            pose data
383         filtered_pose.position.x = data_pose(0);
384         filtered_pose.position.y = data_pose(1);
385         filtered_pose.position.z = data_pose(2);
386         filtered_pose.orientation.x = data_pose(3);
387         filtered_pose.orientation.y = data_pose(4);
388         filtered_pose.orientation.z = data_pose(5);
389         filtered_pose.orientation.w = data_pose(6);
390     }
391     return filtered_pose;
392 }
393
394 void RosFilter::preparePublishData(geometry_msgs::PoseArray&
    camera_poses_msg, aruco_msgs::MarkerArray&
    filtered_markers_msg){
395     // Retrieve the filtered data from the Kalman filter (kf)
396     Eigen::VectorXd filtered_states = kf.getState();
397     Eigen::MatrixXd covariance_matrix = kf.getCovariance();
398
399     geometry_msgs::Pose camera_msg;
400
401     aruco_msgs::Marker markerAux;
402
403     for(const auto& [camera_id, camera_basis] : camera_poses){
404         camera_msg = extractDataFromState(camera_basis,
            filtered_states);
405         camera_poses_msg.header.stamp = ros::Time::now();
406         camera_poses_msg.poses.push_back(camera_msg);
407
408         for (const auto& marker : cam_markers[camera_id]) {
409             markerAux = extractDataFromState(marker,
                filtered_states, covariance_matrix);
410             // markerAux.header = msg->header; // Use the same
                header as the input marker data
```

```
411             markerAux.header.frame_id = "cam_1"; // publish
                   everything related to origin (since the system is
                   suposed to have only same referenced data)
412             markerAux.header.stamp = ros::Time::now();
413             filtered_markers_msg.markers.push_back(markerAux);
414         }
415     }
416 }
417
418 void RosFilter::publishData(geometry_msgs::PoseArray
    camera_poses_msg, aruco_msgs::MarkerArray filtered_markers_msg){
419
420     // Publish the filtered marker data
421     filtered_marker_publisher.publish(filtered_markers_msg);
422
423     // Publish the filtered camera data
424     camera_publisher.publish(camera_poses_msg);
425 }
426
427 void RosFilter::cameraCallback(const
    aruco_msgs::MarkerArray::ConstPtr& msg, const int& camera_id) {
428     if(!last_msgs[camera_id]){
429         last_msgs[camera_id] =
               std::make_shared<aruco_msgs::MarkerArray>();
430     }
431     auto p  = last_msgs[camera_id];
432     *p = *msg;
433 }
434
435 void RosFilter::timerCallback(const ros::TimerEvent& event) {
436     geometry_msgs::PoseArray camera_poses_msg;
437     aruco_msgs::MarkerArray filtered_markers_msg;
438
439     if(last_msgs.empty()) {
440         // Filter the data
441         saveCovariance();
442         kf.predict();
443     }
444     else {
445         for (const auto& [camera_id, msg]: last_msgs) {
446             if(last_msgs[camera_id] != nullptr) {
447                 // Insert the markers detected in the system
448                 saveCovariance();
449                 for (const auto& marker : msg->markers) {
450                     insertUpdateMarker(marker, camera_id);
451                 }
452
453                 // Adapt the filter for the new data/change of
                       state vector size
454                 Eigen::VectorXd oldState = kf.getState();
455
```

```
456                 // Verifying if there is a need to extend the
                       state vector (new states)
457                 if(oldState.size() < tracked_poses *
                       POSE_VECTOR_SIZE) {
458                     resizeState();
459                 }
460 
461                 Eigen::VectorXd newState(tracked_poses *
                       POSE_VECTOR_SIZE);
462                 insertPosesOnStateVector(newState, camera_id);
463 
464                 // Filter the data
465                 kf.predict();
466                 kf.correct(newState);
467             }
468         }
469     }
470     // Construct the to-be-published data structure and publish
           them
471     preparePublishData(camera_poses_msg, filtered_markers_msg); //
           using the header of the first msg received only for
           simplification
472 
473     publishData(camera_poses_msg, filtered_markers_msg);
474 
475     last_msgs.clear();
476 }
477 
478 poseTransform RosFilter::poseToTf( Eigen::VectorXd pose) {
479     tf::Vector3 tr(pose(0), pose(1), pose(2));
480     tf::Quaternion rot(pose(3), pose(4), pose(5), pose(6));
481     return poseTransform(rot, tr);
482 }
483 
484 Eigen::VectorXd RosFilter::tfToPose(poseTransform transform) {
485 
486     Eigen::VectorXd pose(7);  // storing the poses in a vector
           with the format [xt,yt,zt,xr,yr,zr,wr]
487 
488     pose << transform.tf_tr.x(),
489                     transform.tf_tr.y(),
490                     transform.tf_tr.z(),
491                     transform.tf_q.x(),
492                     transform.tf_q.y(),
493                     transform.tf_q.z(),
494                     transform.tf_q.w();
495 
496     return pose;
497 }
498 
499 void RosFilter::resizeState(){
500     Eigen::VectorXd oldState = kf.getState();
```

```
501     int size_difference = (tracked_poses * POSE_VECTOR_SIZE) -
            oldState.size();
502     oldState.conservativeResize(tracked_poses * POSE_VECTOR_SIZE);
503     oldState.tail(size_difference).setZero();
504     kf.insertState(oldState);
505     // Dont think this is necessary but just in case, considering
            its being tested
506     // Makes so that the pose's covariance related to the world
            (initial camera) is 0.
507     // ...
508     // kf.resetWorld(camera_poses[1].stateVectorAddr);
509 }
510
511 void RosFilter::resizeState(Eigen::VectorXd newData){
512     Eigen::VectorXd oldState = kf.getState();
513     int size_difference = (tracked_poses * POSE_VECTOR_SIZE) -
            oldState.size();
514     oldState.conservativeResize(tracked_poses * POSE_VECTOR_SIZE);
515     oldState.tail(size_difference) = newData;
516     kf.insertState(oldState);
517     // Dont think this is necessary but just in case, considering
            its being tested
518     // Makes so that the pose's covariance related to the world
            (initial camera) is 0.
519     // ...
520     // kf.resetWorld(camera_poses[1].stateVectorAddr);
521 }
522
523 void RosFilter::saveCovariance(){
524     std::ofstream file;
525     std::string path("./readings/filtered/covariance.csv");
526     file.open(path, std::ios::out | std::ios::app);
527     auto covariance = kf.getCovariance();
528     file <<  ros::Time::now() << ",";
529     for (int i = 0; i < covariance.rows(); i++)
530     {
531         file << covariance(i,i);
532         if( i < covariance.rows() - 1){
533             file << ",";
534         }
535     }
536     file << "\n";
537     file.close();
538 }
```

A.3 Código de filtragem de Kalman

Código A.3 – Código em C++

```
1 #include <eigen3/Eigen/Dense>
```

```
2  #include <eigen3/Eigen/Eigen>
3  #include <ros/ros.h>
4
5  #ifndef KALMAN_FILTER_H
6  #define KALMAN_FILTER_H
7
8  #define POSE_VECTOR_SIZE 7
9  #define LINEAR_VECTOR_SIZE 3
10 #define ANGULAR_VECTOR_SIZE 4
11 #define HIGH_COVARIANCE_PRESET 0.1
12 #define PROCESS_LINEAR_NOISE_COVARIANCE 0.1
13 #define PROCESS_ANGULAR_NOISE_COVARIANCE 0.001
14
15 //  TIME UPDATE EQUATIONS
16 //  X[k] = A * X[k-1] + w[k1]             // Predict a state based
       on the process stochastic model
17 //  Y[k] = H * X[k] + v[k]                // Predict output based
       on the measurement stochastic model
18 //  P[k] = A * P[k-1] * A^{T} + Q           // Predict error
       covariance
19 //
20 //  p(w) ~ N(0,Q) // Normal distribution with mean of 0 and
       covariance of Q
21 //  p(v) ~ N(0,R) // Normal distribution with mean of 0 and
       covariance of R
22 //
23 //  MEASUREMENTS UPDATE EQUATIONS
24 //  K[k] = P[k] * H^{T} * ((H * P[k] * H^{T} + R)^{-1}) //
       Calculating Kalman gain
25 //  X[k] = X[k] + K[k] * (Y[k] - H * X[k])        // after
       measuring Y[k], calculating a posteriori state estimate
26 //  P[k] = (I - K[k] * H) * P[k]            // Calculate a
       posteriori error covariance estimate
27
28 class filter{
29
30   public:
31     Eigen::VectorXd X;    // State vector
32     Eigen::VectorXd X0;   // Initial state vector
33     Eigen::MatrixXd P;    // Covariance matrix
34     Eigen::MatrixXd P0;   // Initial covariance matrix
35     Eigen::MatrixXd A;    // State transition matrix, presumably
         constant and equal identity
36     Eigen::MatrixXd H;    // Observation/measurement matrix,
         presumably constant and equal identity
37     Eigen::MatrixXd Q;    // Process noise covariance
38     Eigen::MatrixXd Q0;   // Reference process noise covariance
39     Eigen::MatrixXd R;    // Measurement noise covariance
40     Eigen::MatrixXd R0;   // Reference measurement noise covariance
41     Eigen::MatrixXd K;    // Kalman filter gain
42     Eigen::VectorXd Y_pred; // Predicted output
43     Eigen::VectorXd Y_cor;  // Corrected output
```

```
44      Eigen::VectorXd I_cor;  // Corrected measurement
           innovation/residual
45      Eigen::VectorXd I_pred; // Predicterd measurement
           innovation/residual
46
47      // Function prototypes for non-linear models and their
           Jacobians
48      Eigen::VectorXd stateTransition(Eigen::VectorXd& state);
49      Eigen::MatrixXd stateTransitionJacobian(Eigen::VectorXd&
           state);
50      Eigen::VectorXd measurementModel(Eigen::VectorXd& state);
51      Eigen::MatrixXd measurementJacobian(Eigen::VectorXd& state);
52
53      void set_dimensions(int dim){
54        X.resize(dim);
55        X0.resize(dim);
56        P.resize(dim, dim);
57        P.setZero();
58        P0.resize(dim, dim);
59        P0.setIdentity() * HIGH_COVARIANCE_PRESET;
60        A.setIdentity(dim, dim);
61        H.setIdentity(dim, dim);
62        Q.resize(dim, dim);
63        Q0.resize(POSE_VECTOR_SIZE, POSE_VECTOR_SIZE);
64        Q0.setZero();
65
66        for(int i = 0; i < Q0.rows(); i++){
67          if(i % POSE_VECTOR_SIZE < 3){
68            Q0(i,i) = PROCESS_LINEAR_NOISE_COVARIANCE;
69          }else{
70            Q0(i,i) = PROCESS_ANGULAR_NOISE_COVARIANCE;
71          }
72        }
73
74        R.resize(dim, dim);
75        R0 = Eigen::MatrixXd::Identity(POSE_VECTOR_SIZE,
              POSE_VECTOR_SIZE);
76      }
77
78      void resetWorld(int worldAddr){
79        P.block(worldAddr, worldAddr, POSE_VECTOR_SIZE,
              POSE_VECTOR_SIZE) =
              Eigen::MatrixXd::Identity(POSE_VECTOR_SIZE,
              POSE_VECTOR_SIZE);
80        X.segment(worldAddr, POSE_VECTOR_SIZE).setZero();
81      }
82
83      void insertState(Eigen::VectorXd state){
84        // Regarding the state vector
85        int size_difference = state.size() - X.size(); // Number of
              new elements to be added = new poses * POSE_VECTOR_SIZE
86        X.conservativeResize(X.rows() + size_difference);
```

```
87        X.tail(state.size()) = state;
88
89        // Regarding the Covariance matrix
90        Eigen::MatrixXd auxMatrix(P.rows() + size_difference,
             P.rows() + size_difference);
91        auxMatrix.setZero();
92        auxMatrix.topLeftCorner(P.rows(), P.rows()) = P;
93        auxMatrix.bottomRightCorner(size_difference,
             size_difference) =
             Eigen::MatrixXd::Identity(size_difference,
             size_difference) * HIGH_COVARIANCE_PRESET;
94        P = auxMatrix;
95
96        // ROS_INFO("P-OLD:");
97        // for (int i = 0; i < P.rows(); ++i) {
98        //  std::string row_str = "[ ";
99        //  for (int j = 0; j < P.cols(); ++j) {
100       //    row_str += std::to_string(P(i, j)) + " ";
101       //  }
102       //  row_str += "]";
103       //  ROS_INFO_STREAM(row_str);
104       // }
105
106       // Regarding the observation matrix
107       A = Eigen::MatrixXd::Identity(A.rows() + size_difference,
             A.cols() + size_difference);
108
109       // Regarding the observation matrix
110       H = Eigen::MatrixXd::Identity(H.rows() + size_difference,
             H.cols() + size_difference);
111
112       // Regarding the process noise covariance matrix
113       Q.resize(Q.rows() + size_difference, Q.cols() +
             size_difference);
114       Q.setZero();
115
116       for(int i = 0; i < Q.rows(); i++){
117         if(i % POSE_VECTOR_SIZE < 3){
118           Q(i,i) = PROCESS_LINEAR_NOISE_COVARIANCE;
119         }else{
120           Q(i,i) = PROCESS_ANGULAR_NOISE_COVARIANCE;
121         }
122       }
123
124       // Regarding the measurement noise covariance matrix
125       R = Eigen::MatrixXd::Identity(R.rows() + size_difference,
             R.cols() + size_difference);
126
127       // if(X.rows() == POSE_VECTOR_SIZE){
128       //  X.setZero();
129       //  P.setIdentity();
130       //  Q = Q0;
```

```
131       // }
132
133       // ROS_INFO("Q:");
134       // for (int i = 0; i < Q.rows(); ++i) {
135       //  std::string row_str = "[ ";
136       //  for (int j = 0; j < Q.cols(); ++j) {
137       //    row_str += std::to_string(Q(i, j)) + " ";
138       //  }
139       //  row_str += "]";
140       //  ROS_INFO_STREAM(row_str);
141       // }
142     }
143
144     void reset(){
145       X = X0;
146       P = P0;
147       Q = Q0;
148       R = R0;
149     }
150
151     void predict(){
152       // Predicst state vector
153       X = A * X;
154
155       for(int i=0;i < X.size();i++){
156         // ROS_INFO("old x%d = %f",i, X[i]);
157       }
158
159       // Predicts Covariance error matrix
160       P = A * P * A.transpose() + Q;
161
162       // ROS_INFO("P-new:");
163       // for (int i = 0; i < P.rows(); ++i) {
164       //  std::string row_str = "[ ";
165       //  for (int j = 0; j < P.cols(); ++j) {
166       //    row_str += std::to_string(P(i, j)) + " ";
167       //  }
168       //  row_str += "]";
169       //  ROS_INFO_STREAM(row_str);
170       // }
171     }
172
173     void correct(Eigen::VectorXd Y){
174       // ROS_INFO_STREAM("PRE-CORRECT X:\n" << X);
175
176       // Predicts the output Y with the predicted X
177       Y_pred = H * X;
178
179       //Calculates prediction error (also called measurement
              innovation or residual)
180       I_pred = Y - Y_pred;
181
```

```
182         // ROS_INFO_STREAM("PRE-CORRECT Y:\n" << Y);
183         // ROS_INFO_STREAM("PRE-CORRECT Y_PRED:\n" << Y_pred);
184 
185         // Kalman gain
186         K = P * H.transpose() * (H * P * H.transpose() +
               R).inverse();
187 
188         // Updates estimated state matrix with Kalman gain and
               estimated output error
189         X = X + K * I_pred;
190 
191         // ROS_INFO_STREAM("POS-CORRECT X:\n" << X);
192 
193         // ROS_INFO("K:");
194         // for (int i = 0; i < K.rows(); ++i) {
195         //  std::string row_str = "[ ";
196         //  for (int j = 0; j < K.cols(); ++j) {
197         //    row_str += std::to_string(K(i, j)) + " ";
198         //  }
199         //  row_str += "]";
200         //  ROS_INFO_STREAM(row_str);
201         // }
202 
203         for(int i=0;i < I_pred.size();i++){
204           // ROS_INFO("i_pred[%d] = %f",i, I_pred[i]);
205         }
206 
207         for(int i=0;i < X.size();i++){
208           // ROS_INFO("new x%d = %f",i, X[i]);
209         }
210 
211         // Updating covariance error matrix with Kalman gaind and
               old covariance error matrix
212         P = (Eigen::MatrixXd::Identity(X.rows(), X.rows()) - K * H)
               * P;
213 
214         // New prediction of output
215         Y_cor = H * X;
216 
217         // New error of output
218         I_cor = Y - Y_cor;
219     }
220 
221     Eigen::VectorXd getState() { return X; }
222     Eigen::MatrixXd getCovariance() { return P; }
223     Eigen::VectorXd getPredictedOutput() { return Y_pred; }
224     Eigen::VectorXd getCorrectedOutput() { return Y_cor; }
225     Eigen::VectorXd getPredictedInnovation() { return I_pred; }
226     Eigen::VectorXd getCorrectedInnovation() { return I_cor; }
227 };
228 
229 #endif
```

A.4 Código de organização das variáveis do filtro

Código A.4 – Código em C++

```
#pragma once

#include <eigen3/Eigen/Dense>
#include <eigen3/Eigen/Eigen>
#include <tf/transform_broadcaster.h>

#define POSE_VECTOR_SIZE 7

// Structure to correlate marker poses with aruco id tags
typedef struct trackedMarker {
    int stateVectorAddr;    // index where its found in the state
       vector of the kalman filter
    int arucoId;            // aruco tag id
    Eigen::VectorXd pose;
    Eigen::MatrixXd covariance;
}trackedMarker;

// Class to implement pose as Vector-Quaternion pair
// Used because tf::Transfornm yielded unsatisfactory results
class poseTransform{
    public:
        tf::Vector3 tf_tr;
        tf::Quaternion tf_q;

        poseTransform(tf::Quaternion q, tf::Vector3 v);
        poseTransform(tf::Quaternion& q);
        poseTransform();

        static tf::Vector3 vector_transformation(tf::Quaternion q,
           tf::Vector3 tr);
        poseTransform inverse();
        poseTransform operator*(poseTransform const& tf); //
           Composition operator
};

// Class to save a camera's pose
class cameraBasis {
    public:
        int stateVectorAddr;        // index where its found in
           the state vector of the kalman filter
        Eigen::VectorXd pose;
        Eigen::MatrixXd covariance;
        poseTransform previous_tf;  //relate to the pose of the
           previous camera. ex.: if it's cam3, it will give the tf
           to the cam2 pose.
```

```
40          int previous_id;
41
42          cameraBasis();
43          cameraBasis(int address, Eigen::VectorXd pose,
               Eigen::MatrixXd cv, poseTransform tf);
44          void updateTfPrevious(poseTransform new_marker_tf,
               poseTransform  marker_tf_from_previous, int
               new_previous_id);
45  };
46
47  cameraBasis::cameraBasis():
48  stateVectorAddr(0),
49  pose(Eigen::VectorXd::Zero(7)),
50  covariance(Eigen::MatrixXd::Identity(7, 7)),
51  previous_tf(poseTransform()),
52  previous_id(1)
53  {
54  }
55
56  poseTransform::poseTransform(tf::Quaternion q, tf::Vector3 v):
57  tf_tr(v),
58  tf_q(q)
59  {
60
61  };
62
63  poseTransform::poseTransform(tf::Quaternion& q):
64  tf_tr(),
65  tf_q(q)
66  {
67      tf_tr = tf::Vector3(0,0,0);
68  };
69
70  poseTransform::poseTransform():
71  tf_tr(tf::Vector3(0,0,0)),
72  tf_q(tf::Quaternion(0,0,0,1))
73  {
74
75  };
76
77  tf::Vector3 poseTransform::vector_transformation(tf::Quaternion q,
       tf::Vector3 tr){
78      tf::Quaternion q_tr = tf::Quaternion(tr.x(),tr.y(), tr.z(), 0);
79      tf::Quaternion q_result = q*q_tr*q.inverse();
80      return tf::Vector3(q_result.x(),q_result.y(),q_result.z());
81  }
82
83  poseTransform poseTransform::inverse(){
84      tf::Vector3 tr_inverted =
           vector_transformation(tf_q.inverse(),tf_tr);
85      tr_inverted *= -1;
86      tf::Quaternion q_inverted = tf_q.inverse();
```

```
87      return poseTransform(q_inverted, tr_inverted);
88  }
89
90  poseTransform poseTransform::operator*(poseTransform const& tf2){
91      tf::Vector3 transformed_tr2 = vector_transformation(tf_q,
            tf2.tf_tr);
92      tf::Vector3 new_tr = tf::Vector3(tf_tr.x() +
            transformed_tr2.x(),
93                                       tf_tr.y() +
                                             transformed_tr2.y(),
94                                       tf_tr.z() +
                                             transformed_tr2.z());
95      tf::Quaternion new_q = tf_q*tf2.tf_q;
96      return poseTransform(new_q,new_tr);
97  }
98
99  cameraBasis::cameraBasis(int address, Eigen::VectorXd pose,
        Eigen::MatrixXd cv, poseTransform tf):
100 stateVectorAddr(address),
101 pose(pose),
102 covariance(cv),
103 previous_tf(tf),
104 previous_id(1)
105 {
106     previous_tf.tf_tr = tf::Vector3(0,0,0);
107     previous_tf.tf_q = tf::Quaternion(0,0,0,1);
108 }
109
110 void cameraBasis::updateTfPrevious(poseTransform new_marker_tf,
        poseTransform  marker_tf_from_previous, int new_previous_id) {
111     previous_id = new_previous_id;
112     previous_tf = new_marker_tf *
            marker_tf_from_previous.inverse();
113     previous_tf = previous_tf;
114 }
```

A.5 Script de inicialização do sistema de visualização

Código A.5 – Código em Python

```
1  import os
2  import time
3
4  def get_device_type(tokens):
5      if tokens[-3] == 'Xbox':
6          return 'kinect'
7      return 'usb_cam'
8
9  error_flag = False
10
```

```
# Run shell commnad and get output
output = os.popen("lsusb | grep 'Camera\|Webcam'").read()
devices = output.splitlines()

# Separate the devices by USB Bus
all_usb_bus = {}
num_kinect = 0
num_usb_cam = 0
for device in devices:
    tokens = device.split(' ')
    bus_id = tokens[1] #Checks Bus ID

    if bus_id in all_usb_bus:
        error_flag = True
        print("Error: Two Devices can't be connect in the same USB
           Bus")
        break
    else:
        device_type = get_device_type(tokens)
        if device_type == 'kinect':
            num_kinect = num_kinect + 1
        else:
            num_usb_cam = num_usb_cam + 1
            all_usb_bus[bus_id] = device_type #Association between
               device type and bus for debugging puposes

# Launching Camera Nodes based on camera type
if not error_flag:
    cam_id = 1
    os.system("gnome-terminal --tab --title='rosocore' -- roscore")
    time.sleep(2)
    for kinect_id in range(1, num_kinect + 1):
        os.system(f"gnome-terminal --tab -- roslaunch
           launch/launch_camera.launch cam_id:={cam_id}
           cam_type:=kinect device_id:={kinect_id}")
        cam_id = cam_id + 1
        time.sleep(1)
    for usb_cam_id in range(1, num_usb_cam + 1):
        os.system(f"gnome-terminal --tab -- roslaunch
           launch/launch_camera.launch cam_id:={cam_id}
           cam_type:=usb_cam device_id:={usb_cam_id}")
        time.sleep(1)
        cam_id = cam_id + 1
```

A.6 Código de visualização no Rviz

Código A.6 – Código em C++

```
#include <vector>
#include <iterator>
```

```
3  #include <algorithm>
4  #include <ros/ros.h>
5  #include "visualization_objects.h"
6  #include <tf/transform_broadcaster.h>
7  #include <aruco_msgs/MarkerArray.h>
8  #include <visualization_msgs/Marker.h>
9  #include <visualization_msgs/MarkerArray.h>
10 #include <geometry_msgs/PoseArray.h>
11
12 class VisualizationHandler{
13     protected:
14         std::vector<Camera> sys_cameras;
15         std::vector<VisualizationMarker> sys_markers;
16         std::vector<ros::Subscriber> sys_subs;
17         ros::NodeHandle nh;
18         tf::TransformBroadcaster br;
19         ros::Publisher marker_pub;
20
21         int find_marker(VisualizationMarker m);
22         void callback_markers(const aruco_msgs::MarkerArray& msg);
23         void callback_cameras(const geometry_msgs::PoseArray& msg);
24         void add_cameras(std::string topic);
25         void add_markers(std::string topic);
26
27     public:
28         VisualizationHandler(ros::NodeHandle& node_handle);
29         void start(std::string camera_topic, std::string
              markers_topic);
30         void send_tfs();
31         void clear_markers();
32         void publish_markers();
33 };
34
35 std::vector<std::string> get_camera_topics();
36 tf::Matrix3x3  get_rot_from_quat(tf::Quaternion q);
37 tf::Quaternion get_quat_from_rot(tf::Matrix3x3 r);
38 void test_quaternion(tf::Quaternion q);
39
40 VisualizationHandler::VisualizationHandler(ros::NodeHandle&
      node_handle){
41     sys_cameras = std::vector<Camera>();
42     sys_markers = std::vector<VisualizationMarker>();
43     std::vector<ros::Subscriber> sys_subs =
          std::vector<ros::Subscriber>();
44     nh = node_handle;
45     br = tf::TransformBroadcaster();
46     marker_pub =
          nh.advertise<visualization_msgs::MarkerArray>("visualization_marker_a
          1);
47 }
48
49 //Find marker on all system markers
```

```
50 // Returns marker index if found, else returns -1
51 int VisualizationHandler::find_marker(VisualizationMarker m){
52     auto marker_it = std::find(sys_markers.begin(),
          sys_markers.end(), m);
53     if(!sys_markers.empty() && marker_it != sys_markers.end()){ //
          Found the marker
54         return std::distance(sys_markers.begin(), marker_it);
55     }
56     else { // Marker not found or empty vector
57         return -1;
58     }
59 }
60
61 //Clears all markers which lifetime expired
62 //Uses the erase-remove idiom
63 void VisualizationHandler::clear_markers(){
64     sys_markers.erase(std::remove_if(sys_markers.begin(),
          sys_markers.end(),
65     [](const VisualizationMarker& m){return
          m.get_marker().header.stamp + m.get_marker().lifetime <=
          ros::Time::now();}), sys_markers.end());
66 }
67
68 // Callback method for the markers subscriber
69 // Responsible for update the system markers to be published
70 void VisualizationHandler::callback_markers(const
      aruco_msgs::MarkerArray& msg){
71     for(int i = 0; i < msg.markers.size(); i++){
72         VisualizationMarker marker_candidate =
              VisualizationMarker(msg.markers.at(i));
73         int marker_index = find_marker(marker_candidate);
74         if(marker_index != -1){ //Marker found: Update marker
75             sys_markers.at(marker_index).set_marker(marker_candidate);
76         }
77         else{ //Marker not found: Add to the list of markers
78             ROS_INFO_STREAM("Adding marker " <<
                  marker_candidate.get_marker().id);
79             sys_markers.push_back(marker_candidate);
80         }
81     }
82 }
83
84 // Callback method for the camera subscriber
85 // Responsible for update the system cameras to be published
86 void VisualizationHandler::callback_cameras(const
      geometry_msgs::PoseArray& msg){
87     for(int i = 0; i < msg.poses.size(); i++){
88
89         tf::Vector3 tr(msg.poses.at(i).position.x,
90                        msg.poses.at(i).position.y,
91                        msg.poses.at(i).position.z);
92
```

```
93          tf::Quaternion rot(msg.poses.at(i).orientation.x,
94                             msg.poses.at(i).orientation.y,
95                             msg.poses.at(i).orientation.z,
96                             msg.poses.at(i).orientation.w);
97
98          if(i >= sys_cameras.size()){
99              ROS_INFO_STREAM("Adding camera " << i+1);
100             sys_cameras.push_back(Camera(tr, rot, "cam_" +
                    std::to_string(i+1)));
101         }
102         else {
103             sys_cameras.at(i).set_tf(tr, rot);
104         }
105     }
106 }
107
108 void VisualizationHandler::start(std::string camera_topic,
    std::string markers_topic){
109     add_cameras(camera_topic);
110     add_markers(markers_topic);
111 }
112
113 // create the respective  ROS  camera subscriber
114 void VisualizationHandler::add_cameras(std::string topic){
115     ROS_INFO_STREAM("Adding cameras...");
116     ROS_INFO_STREAM("Subscribing to " << topic << std::endl);
117     sys_subs.push_back(nh.subscribe(topic, 5,
           &VisualizationHandler::callback_cameras, this));
118
119 }
120
121 void VisualizationHandler::add_markers(std::string topic){
122     ROS_INFO_STREAM("Adding markers ...");
123     ROS_INFO_STREAM("Subscribing to " << topic << std::endl);
124     sys_subs.push_back(nh.subscribe(topic, 5,
           &VisualizationHandler::callback_markers, this));
125 }
126
127
128 // Send all objects transformations as TF objects to use on the
    RViz environment
129 // TF_translation = [x,y,z]
130 // TF_rotation(quaternion) = [x,y,z,w]
131 void  VisualizationHandler::send_tfs(){
132     auto map =
           tf::StampedTransform(tf::Transform(tf::Quaternion(0.5,0.5,0.5,0.5),
           tf::Vector3(0,0,0)) , ros::Time::now(),"map",
           "world_aruco");
133     br.sendTransform(map);
134     for(auto it = sys_cameras.begin(); it != sys_cameras.end();
           it++){
135         br.sendTransform(it->get_tf_stamped());
```

```
136     }
137     for(auto it = sys_markers.begin(); it != sys_markers.end();
           it++){
138         br.sendTransform(it->get_tf_stamped());
139     }
140 }
141 
142 //Publish all markers to the RVIZ environment
143 void VisualizationHandler::publish_markers(){
144     visualization_msgs::MarkerArray arr;
145     for(int i = 0; i < sys_markers.size(); i++){
146         arr.markers.push_back(sys_markers.at(i).get_marker());
147     }
148     marker_pub.publish(arr);
149 }
150 
151 //Utility Functions
152 
153 // Find all aruco marker publisher topics for each camera
154 std::vector<std::string> get_camera_topics(){
155     ros::master::V_TopicInfo master_topics;
156     ros::master::getTopics(master_topics);
157     std::string marker_topic_name("/filtered_markers");
158     std::vector<std::string>topics;
159 
160     for (ros::master::V_TopicInfo::iterator it =
           master_topics.begin() ; it != master_topics.end(); it++) {
161         const ros::master::TopicInfo& info = *it;
162         if(info.name.find(marker_topic_name) != std::string::npos){
163             if(info.name.find("list") == std::string::npos){  //
                   filters "markers_list" topics from the vector
164                 topics.push_back(info.name);
165             }
166         }
167     }
168     return topics;
169 }
170 
171 // Converts Quaternion to Rotation Matrix (3x3)
172 // From Ken Shoemake's article "Quaternion Calculus and Fast
       Animation"
173 tf::Matrix3x3  get_rot_from_quat(tf::Quaternion q){
174     float norm = q.x()*q.x() + q.y()*q.y() + q.z()*q.z() +
           q.w()*q.w();
175     float s = (norm >= 0) ? 2.0/norm : 0.0;
176 
177     float r_xx = 1 - s*(q.y()*q.y() + q.z()*q.z());
178     float r_xy = s*(q.x()*q.y() - q.w()*q.z());
179     float r_xz = s*(q.x()*q.z() + q.w()*q.y());
180 
181     float r_yx = s*(q.x()*q.y() + q.w()*q.z());
182     float r_yy = 1 - s*(q.x()*q.x() + q.z()*q.z());
```

```
183     float r_yz = s*(q.y()*q.z() - q.w()*q.x());
184 
185     float r_zx = s*(q.x()*q.z() - q.w()*q.y());
186     float r_zy = s*(q.y()*q.z() + q.w()*q.x());
187     float r_zz = 1 - s*(q.x()*q.x() + q.y()*q.y());
188 
189     return tf::Matrix3x3(r_xx, r_xy, r_xz, r_yx, r_yy, r_yz, r_zx,
            r_zy, r_zz);
190 }
191 
192 // Converts Rotation Matrix (3x3) to Quaternion
193 // From Ken Shoemake's article "Quaternion Calculus and Fast
    Animation"
194 tf::Quaternion get_quat_from_rot(tf::Matrix3x3 r){
195     float rot[3][3];
196 
197     rot[0][0] = r[0].x(); // rxx
198     rot[0][1]= r[0].y(); // rxy
199     rot[0][2] = r[0].z();// rxz
200 
201     rot[1][0] = r[1].x(); // ryx
202     rot[1][1] = r[1].y(); // ryy
203     rot[1][2] = r[1].z(); // ryz
204 
205     rot[2][0] = r[2].x(); // rzx
206     rot[2][1] = r[2].y(); // rzy
207     rot[2][2] = r[2].z(); // rzz
208 
209     float tr = rot[0][0] + rot[1][1] + rot[2][2];
210 
211     if(tr >= 0.0){
212         float q_x,q_y,q_z,q_w;
213         float s = std::sqrt(tr + 1.0);
214         q_w = s*0.5;
215         s = 0.5/s;
216         q_x = (rot[2][1] - rot[1][2])*s;
217         q_y = (rot[0][2] - rot[2][0])*s;
218         q_z = (rot[1][0] - rot[0][1])*s;
219         return tf::Quaternion(q_x, q_y, q_z, q_w);
220     }
221     else {
222         int i = 0;
223         float q[4];
224         if(rot[1][1] > rot[0][0]){
225             i = 1;
226         }
227         if (rot[2][2] > rot[i][i]){
228             i = 2;
229         }
230         int j = (i+1)%3;
231         int k = (j+1)%3;
232         float s = sqrt(rot[i][i] - rot[j][j] - rot[k][k] + 1.0);
```

```
233         q[i] = s*0.5;
234         s = 0.5/s;
235         q[j] = (rot[i][j] + rot[j][i]) * s;
236         q[k] = (rot[k][i] + rot[i][k]) * s;
237         q[3] = (rot[k][j] - rot[j][k]) * s;
238         return tf::Quaternion(q[0], q[1], q[2], q[3]);
239     }
240 }
241
242 //Quaternion conversion verification routine
243 void test_quaternion(tf::Quaternion q){
244
245     auto converted_q = get_quat_from_rot(get_rot_from_quat(q));
246
247     std::cout << "q (Original):"
248     <<"              x= " << q.x()
249     << " y= " << q.y()
250     << " z = " << q.z()
251     << " w= " << q.w() << std::endl;
252
253     std::cout << "q (from Rotation Matrix):"
254     << " x= " << converted_q.x()
255     << " y= " << converted_q.y()
256     << " z = " << converted_q.z()
257     << " w= " << converted_q.w() << std::endl;
258
259 }
```

A.7 Código do nó de visualização no Rviz

Código A.7 – Código em C++

```
1  #include <ros/ros.h>
2  #include "marker_viz.h"
3  #include <vector>
4
5
6  int main(int argc, char** argv){
7
8    // ROS node configuration
9    ros::init(argc, argv, "visualization_node");
10   ros::NodeHandle n;
11   ros::Rate r(30);
12
13   ROS_INFO_STREAM("Starting Node...");
14
15   //Setting the system visualization handler
16   VisualizationHandler vh = VisualizationHandler(n);
17   std::string camera_topic = "kalman/filtered_cam_poses";
18   std::string markers_topic = "kalman/filtered_markers_poses";
```

```
19
20    vh.start(camera_topic, markers_topic);
21
22    // Node Loop
23    while (ros::ok()){
24      vh.clear_markers();
25      vh.send_tfs();
26      vh.publish_markers();
27      ros::spinOnce();
28      r.sleep();
29    }
30    return 0;
31  }
```

A.8 Código dos objetos de visualização no Rviz

Código A.8 – Código em C++

```
1  #include <ros/ros.h>
2  #include <vector>
3  #include <tf/transform_broadcaster.h>
4  #include <visualization_msgs/Marker.h>
5  #include <aruco_msgs/Marker.h>
6
7  class VisualizationMarker{
8      visualization_msgs::Marker marker;
9      tf::Transform tf;
10     std::string parent_frame;
11
12     public:
13         VisualizationMarker();
14         VisualizationMarker(aruco_msgs::Marker m);
15
16         tf::StampedTransform get_tf_stamped() const;
17         tf::Transform get_tf() const;
18         visualization_msgs::Marker get_marker() const;
19         bool operator==(const VisualizationMarker& obj);
20         void set_marker(VisualizationMarker m);
21         void set_pose(tf::Transform new_pose);
22 };
23
24 class Camera {
25     tf::Transform tf;
26     std::string frame_id;
27
28     public:
29         Camera(std::string id);
30         Camera(tf::Vector3 tr, tf::Quaternion rot, std::string id);
31
32         tf::StampedTransform get_tf_stamped() const ;
```

```
33          void set_tf(tf::Vector3 tr, tf::Quaternion rot);
34          bool operator==(const Camera& obj);
35          std::string get_frame_id() const ;
36
37 };
38
39 VisualizationMarker::VisualizationMarker(){
40     marker = visualization_msgs::Marker();
41
42     // Setting the marker initial parameters
43     marker.header.stamp = ros::Time::now();
44     marker.lifetime = ros::Duration(2);
45     int marker_id = 0;
46     parent_frame = "cam_1";
47
48     // Set the namespace and id for this marker.  This serves to
          create a unique ID
49     marker.header.frame_id = parent_frame;
50     marker.id = marker_id;
51     marker.ns = marker.header.frame_id + "_m_" +
          std::to_string(marker.id);
52     marker.type = visualization_msgs::Marker::CUBE;
53
54     // Set the pose of the marker.
55     marker.pose.position.x = 0;
56     marker.pose.position.y = 0;
57     marker.pose.position.z = 0;
58     marker.pose.orientation.x = 0.0;
59     marker.pose.orientation.y = 0.0;
60     marker.pose.orientation.z = 0.0;
61     marker.pose.orientation.w = 1.0;
62
63     // Set the scale of the marker -
64     marker.scale.x = 0.5;
65     marker.scale.y = 0.5;
66     marker.scale.z = 0.5;
67
68     // Set the color -- be sure to set alpha to something non-zero!
69     marker.color.r = 0.0f;
70     marker.color.g = 1.0f;
71     marker.color.b = 0.0f;
72     marker.color.a = 1.0;
73
74     //Set the TF transform of the relative marker pose
75     tf::Vector3 tr(marker.pose.position.x, marker.pose.position.y,
          marker.pose.position.z);
76     tf::Quaternion rot(marker.pose.orientation.x,
          marker.pose.orientation.y, marker.pose.orientation.z,
          marker.pose.orientation.w);
77     tf  = tf::Transform(rot, tr);
78 }
79
```

```
80  VisualizationMarker::VisualizationMarker(aruco_msgs::Marker m){
81      marker = visualization_msgs::Marker();
82
83      // Setting the marker initial parameters
84      marker.header.stamp = ros::Time::now();
85      marker.lifetime = ros::Duration(2);
86      int marker_id = m.id;
87      parent_frame = m.header.frame_id;
88
89      // Set the namespace and id for this marker.  This serves to
            create a unique ID
90      marker.header.frame_id = parent_frame;
91      marker.id = marker_id;
92      marker.ns = marker.header.frame_id + "_m_" +
            std::to_string(marker.id);
93      marker.type = visualization_msgs::Marker::CUBE;
94
95      // Set the pose of the marker.
96      marker.pose.position.x = m.pose.pose.position.x;
97      marker.pose.position.y = m.pose.pose.position.y;
98      marker.pose.position.z = m.pose.pose.position.z;
99      marker.pose.orientation.x = m.pose.pose.orientation.x;
100     marker.pose.orientation.y = m.pose.pose.orientation.y;
101     marker.pose.orientation.z = m.pose.pose.orientation.z;
102     marker.pose.orientation.w = m.pose.pose.orientation.w;
103
104     // Set the scale of the marker -
105     marker.scale.x = 0.5;
106     marker.scale.y = 0.5;
107     marker.scale.z = 0.5;
108
109     // Set the color -- be sure to set alpha to something non-zero!
110     marker.color.r = 0.0f;
111     marker.color.g = 1.0f;
112     marker.color.b = 0.0f;
113     marker.color.a = 1.0;
114
115     //Set the TF transfor of the relative marker pose
116     tf::Vector3 tr(marker.pose.position.x, marker.pose.position.y,
            marker.pose.position.z);
117     tf::Quaternion rot(marker.pose.orientation.x,
            marker.pose.orientation.y, marker.pose.orientation.z,
            marker.pose.orientation.w);
118     tf  = tf::Transform(rot, tr);
119 }
120
121 visualization_msgs::Marker VisualizationMarker::get_marker() const
        {
122     return marker;
123 }
124
125 tf::Transform VisualizationMarker::get_tf() const {
```

```
126     return tf;
127 }
128
129 tf::StampedTransform VisualizationMarker::get_tf_stamped() const {
130     return tf::StampedTransform(tf, ros::Time::now(),
            parent_frame, "m_" + std::to_string(marker.id));
131 }
132
133 bool VisualizationMarker::operator==(const VisualizationMarker&
        obj){
134     return (this->marker.ns == obj.marker.ns) ? true : false;
135 }
136
137 void VisualizationMarker::set_marker(VisualizationMarker m){
138     marker = m.marker;
139     tf.setOrigin(tf::Vector3(marker.pose.position.x,
            marker.pose.position.y, marker.pose.position.z));
140     tf.setRotation(tf::Quaternion(marker.pose.orientation.x,
            marker.pose.orientation.y, marker.pose.orientation.z,
            marker.pose.orientation.w));
141 }
142
143 void VisualizationMarker::set_pose(tf::Transform new_pose){
144     tf = new_pose;
145     marker.pose.position.x = new_pose.getOrigin().x();
146     marker.pose.position.y = new_pose.getOrigin().y();
147     marker.pose.position.z = new_pose.getOrigin().z();
148     marker.pose.orientation.x = new_pose.getRotation().x();
149     marker.pose.orientation.y = new_pose.getRotation().y();
150     marker.pose.orientation.z = new_pose.getRotation().z();
151     marker.pose.orientation.w = new_pose.getRotation().w();
152 }
153
154 Camera::Camera(std::string id){
155     tf = tf::Transform(tf::Quaternion(0,0,0,1),
            tf::Vector3(0,0,0));
156     frame_id = id;
157 }
158
159 Camera::Camera(tf::Vector3 tr, tf::Quaternion rot, std::string id){
160     tf = tf::Transform(rot, tr);
161     frame_id = id;
162 }
163
164 tf::StampedTransform Camera::get_tf_stamped() const {
165     return tf::StampedTransform(tf,
            ros::Time::now(),"world_aruco", frame_id);
166 }
167
168 bool Camera::operator==(const Camera& obj){
169     return (this->frame_id == obj.frame_id) ? true : false;
170 }
```

```
171
172 void Camera::set_tf(tf::Vector3 tr, tf::Quaternion rot){
173     tf.setOrigin(tr);
174     tf.setRotation(rot);
175 }
176
177 std::string Camera::get_frame_id() const {
178     return frame_id;
179 }
```