UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
CURSO DE GRADUAÇÃO EM ENGENHARIA DE CONTROLE E AUTOMAÇÃO

Mateus Osvaldo Klan Pereira

COMPARATIVO DE ALGORITMOS DE LOCALIZAÇÃO E MAPEAMENTO SIMULTÂNEOS MONOCULARES DYNASLAM E ORB-SLAM3 EM AMBIENTES INTERNOS E EXTERNOS

Santa Maria, RS
2023

Mateus Osvaldo Klan Pereira

COMPARATIVO DE ALGORITMOS DE LOCALIZAÇÃO E MAPEAMENTO SIMULTÂNEOS MONOCULARES DYNASLAM E ORB-SLAM3 EM AMBIENTES INTERNOS E EXTERNOS

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia de Controle e Automação da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para obtenção do grau de **Engenheiro de Controle e Automação**. Defesa realizada por videoconferência.

Orientador: Prof. Daniel Fernando Tello Gamarra

Santa Maria, RS
2023

Mateus Osvaldo Klan Pereira

COMPARATIVO DE ALGORITMOS DE LOCALIZAÇÃO E MAPEAMENTO SIMULTÂNEOS MONOCULARES DYNASLAM E ORB-SLAM3 EM AMBIENTES INTERNOS E EXTERNOS

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia de Controle e Automação da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para obtenção do grau de **Engenheiro de Controle e Automação**.

Aprovado em 14 de julho de 2023:

_____

Daniel Fernando Tello Gamarra, Dr. (UFSM)
(Presidente/Orientador)

_____

Anselmo Rafael Cukla, Dr. (UFSM)

_____

Adriano Quilião De Oliveira, Dr. (UFSM)

Santa Maria, RS
2023

RESUMO

COMPARATIVO DE ALGORITMOS DE LOCALIZAÇÃO E MAPEAMENTO SIMULTÂNEOS MONOCULARES DYNASLAM E ORB-SLAM3 EM AMBIENTES INTERNOS E EXTERNOS

AUTOR: Mateus Osvaldo Klan Pereira
Orientador: Daniel Fernando Tello Gamarra

Este trabalho tem como foco a comparação de dois algoritmos de *Visual Simultaneous Localization and Mapping* (*vSLAM*): *ORB-SLAM3* e *DynaSLAM*, utilizando simulações com uma câmera monocular. *ORB-SLAM3* é uma biblioteca de código aberto conhecida por seus recursos *vSLAM* monoculares e é uma evolução do conceituado *ORB-SLAM2*. Em contraste, o *DynaSLAM* estende o *ORB-SLAM2* incorporando *Mask R-CNN* para detecção dinâmica de objetos, filtragem e segmentação. Ambos os algoritmos foram testados e avaliados usando sequências de imagens monoculares de dois conjuntos de dados populares, *KITTI* e *TUM RGB-D*, bem como ambientes de simulação com *ROS* no *software Gazebo 3D*. Os experimentos demonstram a eficiência dos algoritmos *vSLAM*. Os resultados revelam que o *DynaSLAM* supera consistentemente o *ORB-SLAM3* na maioria dos casos. No geral, esta pesquisa contribui para a compreensão desses métodos de *vSLAM*, fornecendo informações sobre seu desempenho e destacando as vantagens e desvantagens do *DynaSLAM* sobre o *ORB-SLAM3* em diversos cenários.

Palavras-chave: vSLAM. ORB-SLAM3. DynaSLAM. Monocular.

ABSTRACT

A COMPARISON OF VISUAL SLAM ALGORITHMS ORB-SLAM3 AND DYNASLAM ON INTERNAL AND EXTERNAL ENVIRONMENTS

AUTHOR: Mateus Osvaldo Klan Pereira
ADVISOR: Daniel Fernando Tello Gamarra

This work focuses on comparing two visual Simultaneous Localization and Mapping (vSLAM) algorithms: ORB-SLAM3 and DynaSLAM, utilizing simulations with a monocular camera. ORB-SLAM3 is an open-source library known for its monocular vSLAM capabilities and is an evolution of the well-regarded ORB-SLAM2. In contrast, DynaSLAM extends ORB-SLAM2 by incorporating Mask R-CNN for dynamic object detection, filtering, and segmentation. Both algorithms were tested and evaluated using sequences of monocular images from two popular datasets, KITTI and TUM RGB-D as well as simulation environments with ROS on the Gazebo 3D software. The experiments demonstrate the efficiency of the vSLAM algorithms. The results reveal that DynaSLAM consistently outperforms ORB-SLAM3 in the majority of cases. Overall, this research contributes to the understanding of these vSLAM methods, providing insights into their performance and highlighting the advantages and disadvantages of DynaSLAM over ORB-SLAM3 in various scenarios.

Keywords: vSLAM. ORB-SLAM3. DynaSLAM. Monocular.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS

SLAM	Localização e Mapeamento Simultâneos
vSLAM	Localização e Mapeamento Visual Simultâneos
TUM	Universidade Técnica de Munique
KITTI	Instituto de Tecnologia de Chicago e Instituto Tecnológico Toyota
CNN	Rede Neural Convolucional
R-CNN	Rede Neural Convolucional Baseada em Região
RGB-D	Vermelho, Verde e Azul mais Profundidade
ORB	*Oriented FAST and Rotated BRIEF*
FAST	*Features from Accelerated Segment Test*
BRIEF	*Binary Robust Independent Elementary Features*
DynaSLAM	Localização e Mapeamento Simultâneos Dinâmico

SUMÁRIO

1 INTRODUÇÃO

Visual Simultaneous Localization and Mapping (*vSLAM*) é um tema que pode ser útil em diversas áreas como robôs com navegação autônoma, realidade aumentada, drones de mapeamento interno, realidade virtual e muito mais. Existem muitos algoritmos utilizando diferentes métodos, como *SLAM* baseado em recursos, direto, denso e baseado em gráfico (TOURANI et al., 2022). Os referidos métodos podem empregar o uso de uma variedade de dispositivos, como câmeras monoculares, RGB-D ou estéreo para capturar os dados necessários para a análise, com cada um deles mostrando vantagens e desvantagens (TOURANI et al., 2022). Este trabalho apresenta a experimentação e consequente comparação de dois algoritmos que utilizam um método SLAM baseado em recursos para imagens monoculares. Hoje em dia existem diversos métodos que prometem melhorar o rastreamento desses algoritmos agregando algum tipo de recurso ou sistema adicional. Dentre eles, um que têm tomado destaque no âmbito acadêmico ultimamente tem sido o *DynaSLAM*, que utiliza uma rede de detecção e segmentação de objetos chamada *Mask R-CNN* juntamente com um método de geometria de múltiplas visualizações.

Os algoritmos *vSLAM* utilizados neste trabalho usam o detector de características ORB para processar os *frames* antes de chegar ao seu rastreamento. Para um ambiente dinâmico, como pessoas andando, objetos em movimento ou algo semelhante, esse tipo de rastreamento tende a apresentar problemas quando alguns dos pontos ORB desses objetos dinâmicos e em movimento estão presentes, porque o sistema *vSLAM* os leva em consideração como dados ruidosos.

1.1 MOTIVAÇÃO

Com o crescente interesse na aplicação de algoritmos vSLAM em ambientes *indoor* e *outdoor*, surge a necessidade de avaliar e comparar o desempenho de diferentes abordagens. Neste contexto, a motivação para este trabalho foi comparar dois algoritmos vSLAM amplamente utilizados.

ORB-SLAM3 é uma evolução do ORB-SLAM2, uma biblioteca de código aberto com excelentes recursos monoculares vSLAM. No entanto, os avanços na tecnologia trouxeram novas possibilidades e o DynaSLAM é uma extensão do ORB-SLAM2 que inclui Mask R-CNN para detecção dinâmica de objetos, filtragem e segmentação. Essa adição permite que o DynaSLAM lide com a presença de objetos em movimento no ambiente com mais eficiência, o que pode ser importante em cenários do mundo real onde a detecção e o rastreamento dinâmico de objetos são essenciais.

A comparação desses dois algoritmos é necessária para entender suas vantagens

e desvantagens em diferentes cenários e condições. Esta informação é importante para a seleção e implementação de algoritmos vSLAM em aplicações reais. Além disso, a realização de experimentos usando diferentes conjuntos de dados e ambientes de simulação oferece uma visão abrangente do desempenho do algoritmo em diferentes situações.

1.2 JUSTIFICATIVA

A robótica é uma área em constante expansão que sobretudo nos últimos anos tem visto seu potencial aumentando veemente (BEGHDADI; MALLEM, 2022). Robôs móveis fazem parte de uma classe dessa área que explora a capacidade destas máquinas de se locomoverem em seus ambientes livremente, seja de maneira automatizada ou manual, com exemplos de modelos terrestres, aéreos e subaquáticos.

O mapeamento de ambientes por meio de robôs móveis já é utilizado em diversas áreas para agilizar e facilitar seu processo de obtenção, que se mostra uma alternativa viável a ser utilizada em situações que o processo e impossível de ser realizado manualmente devido a intempéries externas como calor ou pressão (TOURANI et al., 2022). Uma das principais técnicas para realizar tal mapeamento é através de *simultaneous localization and mapping* (*SLAM*), onde dados de odometria são aliados a leitura de sensores que detectam os arredores para produzir o resultado desejado.

O aprendizado de máquina vem se tornando um aspecto relevante no campo da robótica há algum tempo (BEGHDADI; MALLEM, 2022), avanços vêm sendo feitos e principalmente aprendizagem profunda é muito utilizada para tentar processar grandes volumes de dados advindos de sensores tentando interpretá-los e ajudar na solução de problemas. A integração destas tecnologias pode apresentar uma nova solução que se apresentaria como alternativa para técnicas já existentes, onde um método de odometria visual relativamente novo tal qual o *DynaSLAM* ou *ORB-SLAM3* seria utilizado para realizar a parte de localização de um sistema de *vSLAM*, tanto um robô móvel genérico quanto em qualquer outro sistema que necessite da funcionalidade, desde que o mesmo tenha uma câmera acoplada.

O estudo comparativo proposto neste trabalho busca contribuir para o avanço do campo de vSLAM, fornecendo uma análise detalhada e abrangente desses dois algoritmos amplamente utilizados. Compreender as características, limitações e benefícios de cada abordagem permitirá que pesquisadores, engenheiros e desenvolvedores tomem decisões mais informadas ao escolher o algoritmo mais adequado para suas aplicações específicas. Além disso, os resultados obtidos podem estimular o desenvolvimento de novas técnicas e melhorias nos algoritmos existentes, impulsionando ainda mais o progresso nessa área de pesquisa em rápido crescimento.

1.3 OBJETIVOS GERAIS E ESPECÍFICOS

O desenvolvimento deste trabalho tem como objetivo geral a comparação e avaliação de dois algoritmos de localização e mapeamento simultâneos, que utilizam técnicas de aprendizado profundo em ambientes internos e externos e condições de simulação e reais. Busca-se determinar qual é mais vantajoso em quais situações e apontar pontos positivos e negativos de cada um bem como suas limitações.

Os objetivos específicos incluem:

- Verificar a capacidade atual de técnicas de SLAM visual e suas limitações por meio de dados.
- Implementar *ORB-SLAM3* e *DynaSLAM* com as linguagens de programação Python e C++.
- Desenvolver ambientes de simulação variados com diferentes tipos de robôs móveis.
- Efetuar uma análise dos resultados e tentar identificar vantagens e desvantagens de cada implementação nas diferentes situações averiguadas.
- Testar o algoritmo ORB-SLAM3 em robôs móveis, utilizando ambientes de simulação.
- Testar o algoritmo DynaSLAM em robôs móveis, utilizando ambientes de simulação.

1.4 ESTRUTURA DO TRABALHO

Este trabalho está dividido em seis capítulos. Após esta breve introdução resumindo a ideia e a motivação por trás dos esforços, o Capítulo 2 apresenta trabalhos similares realizados na área nos últimos anos e o estado da arte, discutindo o embasamento teórico necessário para entender a dinâmica dos sistemas utilizados. O Capítulo 3 traz as ferramentas e a metodologia empregadas na experimentação. Já o Capítulo 4 mostra os resultados obtidos após os referidos julgamentos e discute as implicações deles e o que eles trazem para a conversa. Por fim, o Capítulo 5 reúne todos os estudos em uma conclusão com as considerações finais.

2 REFERENCIAL TEÓRICO

Neste capítulo são apresentados os trabalhos que inspiraram e influenciaram este trabalho. Também é realizada uma revisão teórica das técnicas empregadas no desenvolvimento .

2.1 TRABALHOS RELACIONADOS

O *ORB-SLAM* foi apresentado à comunidade por (MUR-ARTAL; MONTIEL; TARDOS, 2015). Inicialmente utilizando diversas funcionalidades e apresentando-se como um dos pioneiros capazes realizar SLAM em tempo real. Em (MUR-ARTAL; TARDOS, 2017), os autores publicados desenvolveram seus trabalhos apresentando o *ORB-SLAM2*, que teve um aumento de desempenho se comparado à versão anterior. Nele foram adicionadas camadas de *bundle adjustment* local e global para minimizar os erros de reprojeção. Em (CAMPOS; ELVIRA; RODRIGUEZ, 2021) foi proposto o *ORB-SLAM3*, trazendo consigo novas funcionalidades como associação de dados de múltiplos mapas. Em (BESCOS et al., 2018) foi apresentado o *DynaSLAM*, que foi basicamente criado usando o *ORB-SLAM2* como algoritmo base, mas aplicando um algoritmo de detecção de objetos e geometria *multi-view* para melhorar seu rastreamento. Foram realizadas comparações do *DynaSLAM* com o *ORB-SLAM2*, porém não com a sua versão mais recente, o *ORB-SLAM3*.

Em (JESUS et al., 2021) a comparação de três algoritmos *vSLAM* foi realizada pelos autores, todos eles com sensores de profundidade e incluindo o *ORB-SLAM2*. Em (BOBBE et al., 2017) foi concebido um *drone* capaz de digitalizar rapidamente um ambiente com imagens aéreas usando *ORB-SLAM2*, foi mostrado que os dados do parâmetro de limite de profundidade variam para cada câmera para obter os melhores resultados. Em (KLEIN; MURRAY, 2007) foi proposta uma implementação paralela de rastreamento e mapeamento visando uma aplicação de realidade aumentada. Em (NEWCOMBE; LOVEGROVE; DAVISON, 2011) o *DTAM* foi apresentado mostrando um avanço significativo na visão geométrica em tempo real.

(BOUAZZAOUI; FLOREZ; OUARDI, 2022) investiga o impacto dos modos de aquisição do sensor na localização *RGB D-SLAM* interna. Os autores comparam os métodos *IR* e *RGB SLAM* e atestam que uma câmera infra-vermelha melhora a localização em até 82,14 por cento. (LEE et al., 2023) apresenta um sistema *SLAM* monocular que usa segmentação semântica baseada em aprendizado profundo semelhante ao *DynaSLAM*. (CRAMARIUC et al., 2023) propõe uma plataforma de código aberto para mapeamento multimodal e multirobô e funciona com um design modular que permite o desenvolvimento

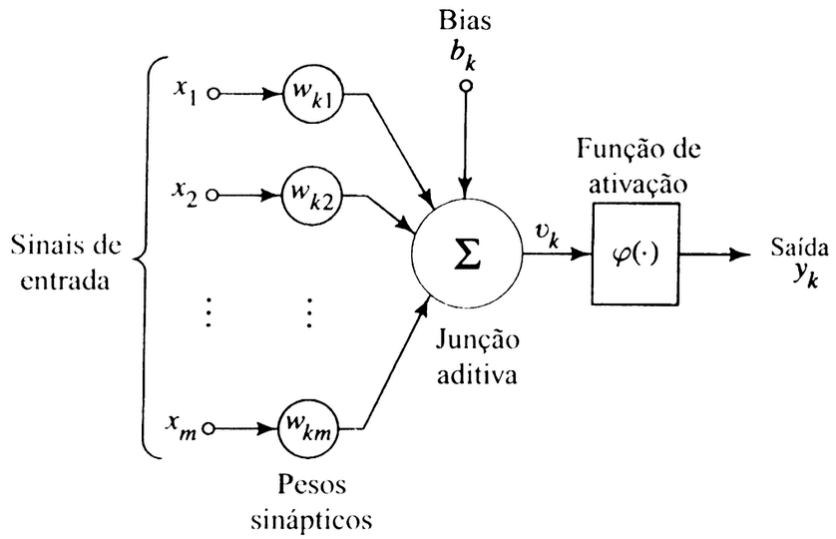
e integração de módulos em um sistema *SLAM*.

2.2 APRENDIZAGEM DE MÁQUINA

Aprendizagem de máquina é um campo da inteligência artificial que visa explorar maneiras de criar sistemas que possam processar dados e aprender a se adaptar sem seguir instruções específicas, utilizando algoritmos para analisar e extrair inferências de padrões presentes nos dados (BEGHDADI; MALLEM, 2022). Fundamentalmente, estes algoritmos extraem informações os representa através de algum modelo matemático.

Ao longo do tempo, algoritmos com diversas abordagens distintas acerca do tópico foram se consolidando, dentre eles pode-se destacar aprendizagem de árvore de decisão, análise de agrupamento de dados, aprendizagem por reforço e aprendizagem profunda. A aprendizagem de máquina é baseada na teoria de redes neurais artificiais, que tem como unidade básica o neurônio. Um neurônio tem como trabalho assimilar duas entradas, multiplicá-las por um peso, adicioná-las a um valor de bias e por fim passá-las por uma função de ativação.

Figura 1 – Estrutura de uma rede neural totalmente conectada



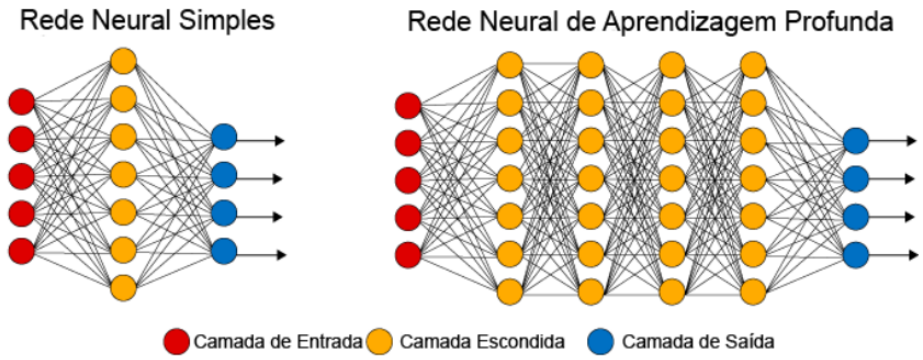
Fonte: (ARAUJO, 2015).

A figura 1 apresenta um modelo de neurônio simples com algumas entradas, a combinação de diferentes entradas sendo utilizadas em uma sequência de neurônios paralelamente em uma camada escondida e posteriormente uma junção aditiva e uma função de ativação é obtida uma saída, que configura uma rede neural.

2.3 APRENDIZAGEM PROFUNDA

Aprendizagem profunda é um ramo de aprendizagem de máquina que surgiu após o poder de processamento computacional médio expandir exponencialmente no início dos anos 2000 (ARAUJO, 2015). Ela emergiu como mecanismo principal de sistemas de inteligência artificial e utiliza uma série de camadas de neurônios para processar seus dados, se distinguindo das redes neurais simples. Seu uso torna possível o processamento de um conjunto de dados muito maior e, dada uma amostragem suficiente, poder de processamento e algoritmos suficientemente sofisticados a realização de tarefas que, até então, só poderiam ser realizadas por seres humanos.

Figura 2 – Comparação da estrutura de camadas de uma rede neural simples e uma de aprendizagem profunda.



Fonte: Traduzido de (DEEP. . . , 2022)

A figura 2 representa uma rede neural de aprendizagem profunda, nela as camadas podem se organizar de maneiras diferentes e possuir funções distintas, consequentemente afetando nos seus resultados.

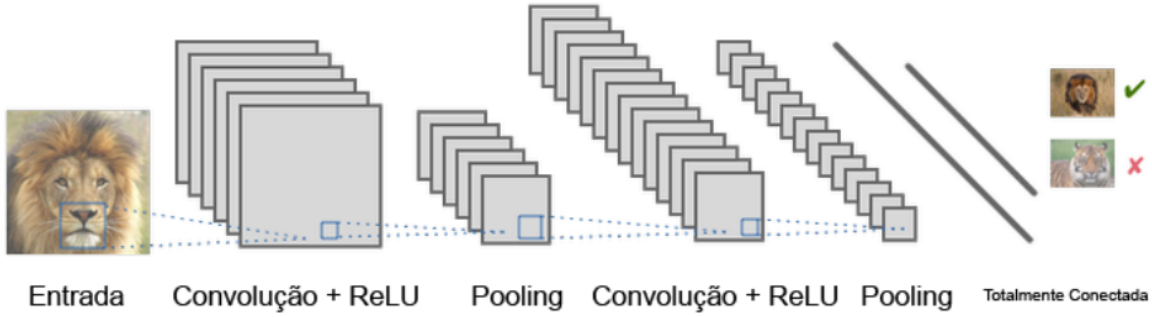
2.4 REDES NEURAIS CONVOLUCIONAIS

Uma arquitetura comum de rede neural é a convolucional, ou CNN, que são frequentemente utilizadas em tarefas que envolvem visão computacional, onde imagens precisam ser processadas para algum fim específico. São eficazes para reconhecer objetos individuais, como caracteres e podem ser replicadas sobre grandes volumes de imagens de entrada para reconhecimento de objetos facilmente (MISHRA, 2020).

Estas redes inferem imagens como volumes, extraindo de sua codificação (normalmente RGB) cada um dos canais e empilhando-os um acima do outro, recebendo a figura como um quadro retangular cujos altura e comprimento são os números de *pixels* de suas

respectivas medições e a profundidade o número de canais presentes no formato do arquivo. À medida que as imagens atravessam as camadas da rede, suas dimensões vão sendo alteradas até que são geradas séries de probabilidades na camada de saída, sendo que estas medidas servem como base para as operações algébricas lineares usadas no processamento das imagens (MISHRA, 2020).

Figura 3 – Exemplo de arquitetura de modelo de rede *CNN*.



Fonte: Traduzido de (MISHRA, 2020).

As camadas de agrupamento servem o propósito de simplificar a informação advinda da camada anterior, a fim de reduzir a carga computacional da rede como um todo. No caso da figura 3, são utilizadas diversas camadas de convolução aliadas a funções de ativação retificadoras e *pooling*, além de uma camada totalmente conectada tipicamente utilizada em problemas de classificação.

2.5 R-CNN

Redes convolucionais baseadas em regiões combinam os métodos de uma rede convolucional tradicional com propostas de regiões retangulares. Elas são compostas por dois estágios, sendo que o primeiro identifica sub-regiões em um *frame* que tem possibilidade de conter um objeto e a segunda classifica o objeto em cada região. Existem três tipos de R-CNNs:

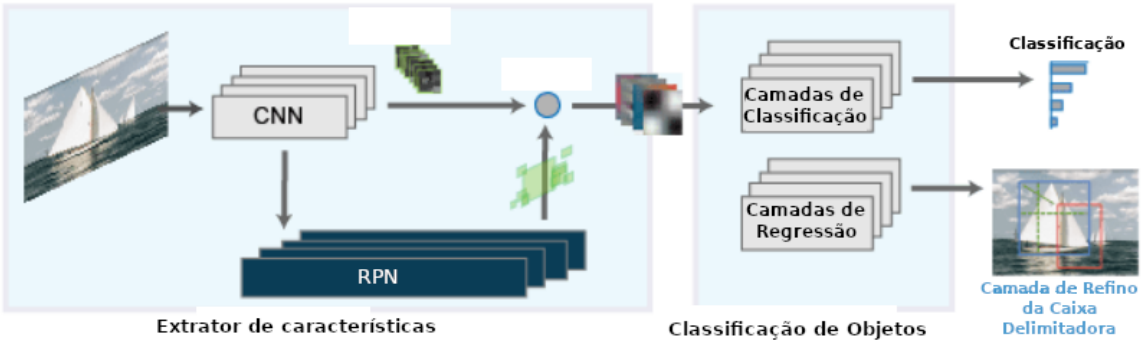
- *R-CNN*
- *Fast R-CNN*
- *Faster R-CNN*

R-CNN (GIRSHICK et al., 2014) se trata de um método que gera potenciais regiões com um algoritmo *Edge Boxes* (ZITNICK; DOLLÁR, 2014). Estas regiões são então cortadas das imagens e redimensionadas para que uma rede convolucional as avalie e averígue se existe mesmo um objeto ali e caso positivo ele seja classificado. Finalmente, as caixas delimitadoras geradas são aprimoradas por uma *support vector machine*.

A *Fast R-CNN* (GIRSHICK, 2015) utiliza o mesmo algoritmo para detectar as propostas. Diferentemente do *R-CNN* tradicional ela processa a imagem inteira ao invés de apenas regiões delimitadas, de forma a agregar características convolucionais correspondentes a cada região potencial, processando regiões concomitantemente e consequentemente melhorando seu desempenho.

O *Faster R-CNN* (REN et al., 2016) agrega uma rede de propostas de região (*RPN*) para que potenciais áreas sejam geradas diretamente na rede ao invés de utilizar um algoritmo como nos dois últimos exemplos, esta rede utiliza *anchor boxes* (uma serie de caixas delimitadoras com dimensões predefinidas) e é ainda mais eficiente, possuindo um melhor desempenho se comparada as suas versões anteriores.

Figura 4 – Exemplo de arquitetura de modelo de rede *Faster R-CNN*.



Fonte: Adaptado de (GETTING. . . , 2020).

A figura 4 representa a arquitetura de rede *Faster R-CNN* completo desde a entrada, passando pela rede convolucional, checagem de áreas potenciais com a rede de propostas de região, camadas de classificação e regressão e saída.

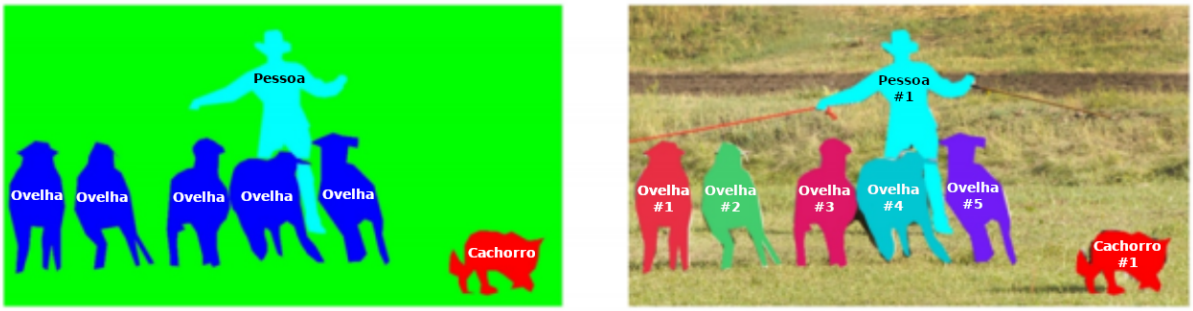
2.6 MASK R-CNN

Mask R-CNN (HE et al., 2017) se dá por um *framework* para detecção e segmentação de objetos que foi desenvolvido tendo como base o *Faster R-CNN*. Para assimilar o funcionamento desta rede, o conceito de segmentação de imagens é essencial. É uma técnica que divide uma imagem em múltiplas regiões visando localizar objetos ou limites

(linhas, curvas, cantos ou outros) e facilitar a sua análise. Este algoritmo realiza dois tipos de segmentação: Semântica e por Instância.

Segmentação semântica classifica cada pixel de uma imagem em uma classe ou objeto baseado em uma lista predefinida, como pode ser visto na figura 5, todos os objetos de cada classe foram agrupados como uma entidade única. Ele também é conhecido como segmentação de *background* já que pode separar os sujeitos da imagem de seu meio.

Figura 5 – Segmentação semântica vs. por instância.



Fonte: Adaptado de (DRAELOS, 2020).

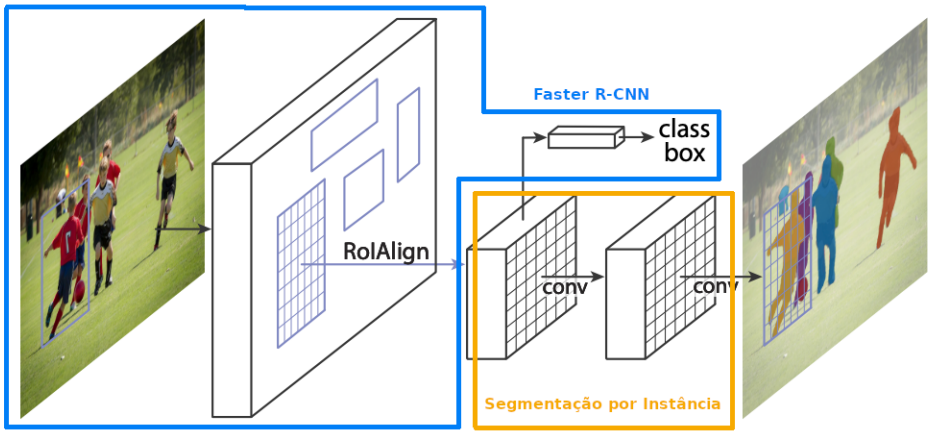
Segmentação por instância diferencia cada um dos objetos provenientes de cada classe por número e os separa com máscaras, portanto combina detecção, localização e classificação de objetos e vai além para distinguir cada objeto rotulado como instâncias similares. Percebe-se pela figura 5 que desta maneira cada ovelha é separada com sua própria máscara e numeração. Esta forma também é conhecida como segmentação de primeiro plano pois ela acentua os sujeitos da imagem ao invés de apenas separá-los do plano de fundo.

Faster R-CNN possui duas saídas de rede para cada potencial objeto detectado: Sua classe e as demarcações de sua caixa delimitadora. O *Mask R-CNN* adiciona uma terceira saída, caracterizada pela máscara do objeto que se diferencia das duas outras saídas pois requer uma extração mais refinada dos contornos de cada alvo. Isso é alcançado pois foi adicionado um recurso responsável por prever uma região de interesse (máscara) em paralelo com o ramo que reconhece a caixa delimitadora. A figura 6 representa a arquitetura deste tipo de rede.

2.7 SLAM VISUAL

SLAM em forma visual permite que um robô ou outro dispositivo mapeie um ambiente desconhecido enquanto mantém o controle de sua localização atual. É amplamente

Figura 6 – Arquitetura de rede *Mask R-CNN*.



Fonte: Adaptado de (HE et al., 2017).

utilizado em aplicações relacionadas à robótica, veículos autônomos, realidade aumentada e realidade virtual. Esta é uma abordagem que usa câmeras para estimativa de pose e geração de mapas e tem demonstrado um desempenho melhor do que as técnicas convencionais que usam apenas um tipo de sensor (TOURANI et al., 2022).

Existem diferentes métodos de *SLAM* visual, como estéreo e monocular. Este trabalho é focado em implementações do segundo, que geralmente recebe uma sequência de imagens de uma única câmera para estimar a posição tridimensional de pontos de referência e a pose da mesma. Essa técnica costuma ser menos precisa do que o estéreo porque faz suposições para produzir informações de profundidade (LEMAIRE et al., 2007).

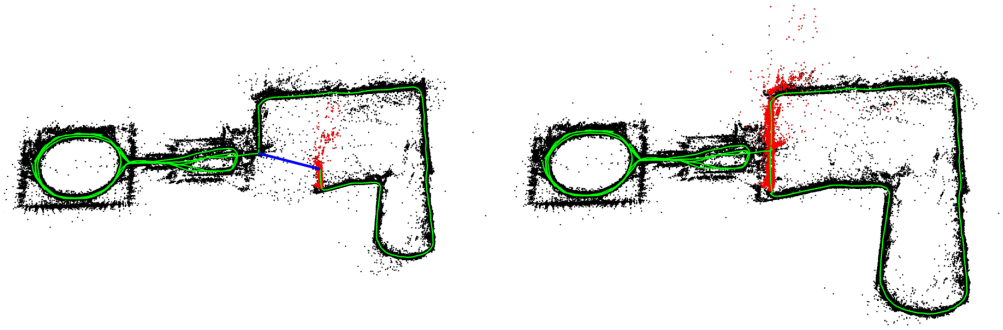
No entanto, algumas das limitações de métodos monoculares incluem sua complexidade computacional, precisão limitada em certas situações, como aquelas com superfícies de baixa textura (BEGHDADI; MALLEM, 2022) e sensibilidade a condições de iluminação, devido à natureza das câmeras e como elas normalmente usam um sensor que captura luz para gerar uma imagem (KUMAR et al., 2022).

2.7.1 Fechamento de Laço

Os algoritmos de vSLAM estimam o movimento contínuo e permitem alguma margem de erro no processamento do mapa local. Isso pode fazer com que os dados do mapa sejam recolhidos ou distorcidos, resultando em erros maiores (MUR-ARTAL; MONTIEL; TARDOS, 2015). Se a taxa de acumulação for alta o suficiente, os pontos iniciais dos mapas não coincidirão, comumente referido como problema de fechamento de laço (MUR-ARTAL; TARDOS, 2017). Tais erros de estimativa de pose são inevitáveis, então uma metodologia é necessária para lidar com eles e modificar ou, idealmente, negar seus

efeitos.

Figura 7 – Exemplo de fechamento de laço em um mapeamento por SLAM.



Fonte: (KRAMER, 2019).

Fechamento de laço se dá pela tarefa de decidir se o veículo, após se movimentar, retornou ou não para uma área que já foi previamente visitada e dada a natureza do trabalho que define um algoritmo de localização e mapeamento simultâneos um fechamento de laço confiável se torna essencial e árduo (MUR-ARTAL; TARDOS, 2017). A figura 7 apresenta um exemplo de fechamento de laço sendo realizado em uma trajetória.

2.7.2 Bundle Adjustment

O ajuste de pacote é utilizado para refinar pontos tridimensionais e poses de câmera para minimizar erros de reprojeção (MUR-ARTAL; MONTIEL; TARDOS, 2015). O procedimento de refinamento é uma variante do algoritmo de Levenberg-Marquadt (MORÉ, 1978) e usa o mesmo sistema de coordenada de referência global para retornar os referidos dados.

Este método consiste em diminuir a taxa de erro de reprojeção entre poses e pontos da imagem observada e é expresso como a soma de um grande número de funções não lineares de valores reais ao quadrado. Desta forma, a minimização é feita usando um algoritmo não linear de mínimos quadrados, sendo um dos de melhor desempenho o de Levenberg-Marquadt (MUR-ARTAL; TARDOS, 2017).

$$LM = \min_{a_j, b_i} \sum_{i=1}^{n} \sum_{i=1}^{m} v_{ij} d(Q(a_j, b_i), x_{ij})^2 \tag{2.1}$$

Na equação (2.1) n representa os pontos 3D vistos nas visualizações m. Desta forma x_{ij} é a projeção do ponto i em um determinado referencial j. v_{ij} denota variáveis binárias que têm valor igual a 1 se i for visível e 0 se não for. O ajuste do pacote minimiza o erro total de reprojeção proveniente de pontos 3D e parâmetros da câmera especifica-

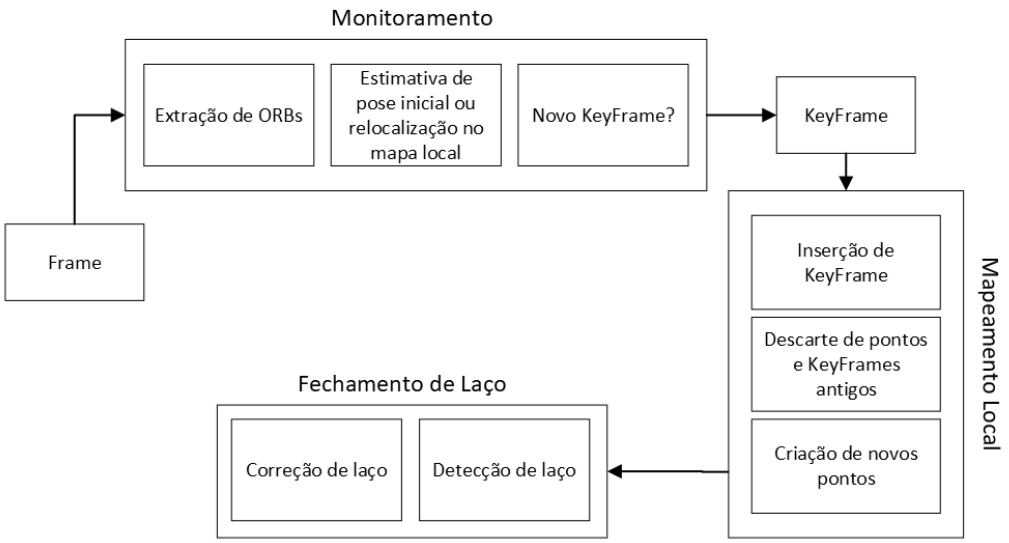
mente. $Q(a_j, b_i)$ é a visualização da projeção no ponto i e quadro j. $d(x, y)$ denota a distância euclidiana entre os pontos do referencial representados pelos vetores x e y.

Como o valor mínimo é calculado a partir de muitos pontos e *frames*, o ajuste do pacote é, por definição, tolerante a projeções de imagens ausentes e, se a métrica de distância for escolhida razoavelmente (por exemplo, distância euclidiana), o ajuste do pacote também minimizará um critério fisicamente significativo.

2.8 ORB-SLAM

ORB-SLAM é um sistema de *SLAM* monocular baseado em *features* que opera em tempo real e é desenhado para ambientes internos e externos pequenos e grandes (MUR-ARTAL; MONTIEL; TARDOS, 2015).

Figura 8 – Diagrama de sistema do *ORB-SLAM*.



Fonte: Autor.

O sistema gira em torno de utilizar as *features* extraídas de cada frame tanto para mapeamento quanto para localização e reconhecimento de espaço para realizar relocalização e detecção de laços, tornando o sistema robusto e eficiente. A figura 8 apresenta um diagrama representando cada um dos três *threads* que são executados paralelamente e suas funções.

São utilizados *ORB*s (RUBLEE et al., 2011), que se dão por cantos detectados por um algoritmo extrator de características por meio de algoritmos de testes segmentados acelerados (*FAST*, (LAVIN; GRAY, 2015)) com escala múltipla e descritores *BRIEF* (CALONDER et al., 2010) de 256 *bits* associados. Este método tem como vantagem ser

extremamente rápido para computar e assimilar.

2.9 ORB-SLAM2

Em sua segunda versão (MUR-ARTAL; TARDOS, 2017), os autores promoveram um aprimoramento do sistema anterior. *Bundle Adjustment* (*BA*) foi aplicado durante o mapeamento local e após o fechamento de laço para otimizar os *keyframes* locais e pontos do mapa local (*BA* local), bem como posteriormente a um fechamento de laço para que todas os *keyframes* restantes e pontos (*BA* total) também sejam otimizados. Ambas as metodologias fazem uso da Equação 2.1 por meio da implementação de (KüMMERLE et al., 2011).

BA local otimiza uma serie de *keyframes* bem como todos os pontos presentes neles, todos os restantes contribuem para uma função de custo mas permanecendo com valores inalterados. *BA* total se dá por um caso especifico de *BA* local onde todos os *keyframes* e pontos no mapa global são otimizados, com exceção do de origem que permanece fixo para eliminar parte da liberdade de calibração.

O sistema também conta com um módulo de reconhecimento de localização embutido baseado em *DBoW2* (GALVEZ-LóPEZ; TARDOS, 2012) para relocalização no caso de perda de rastreamento ou então para reinicializar o mapeamento em uma situação que já existe um mapa e fechamento de laços.

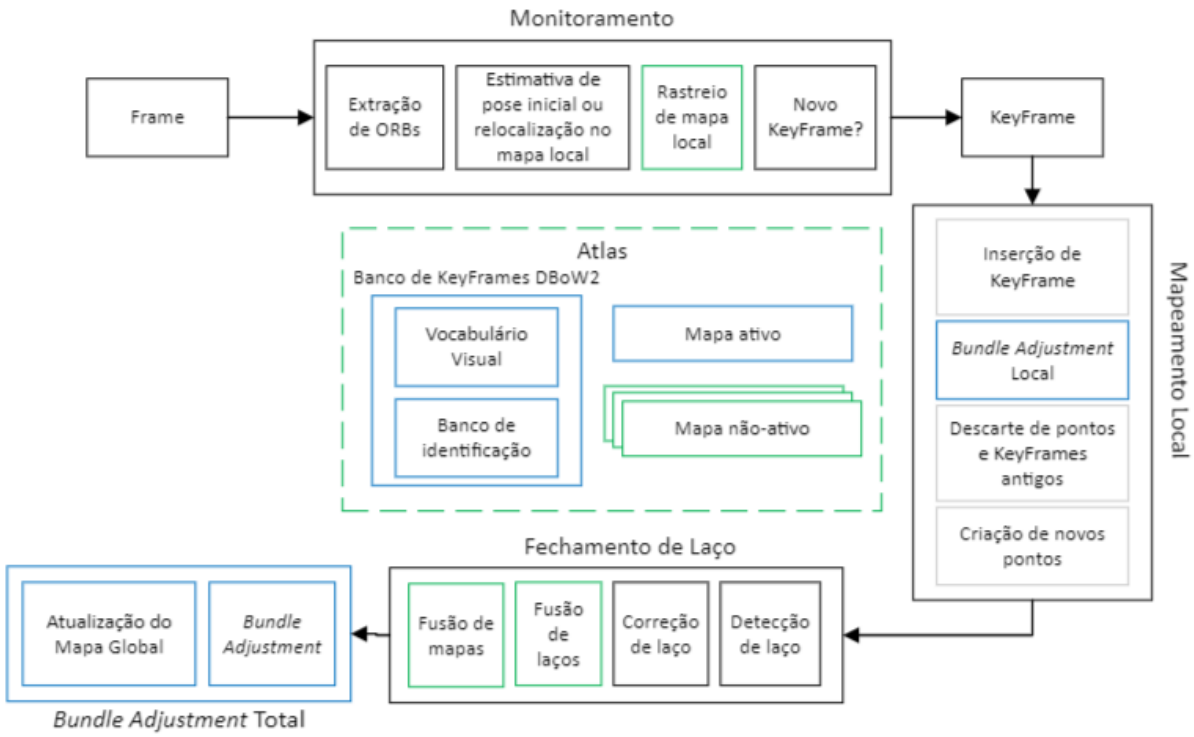
2.10 ORB-SLAM3

ORB-SLAM3 (CAMPOS; ELVIRA; RODRIGUEZ, 2021) baseia-se no *ORB-SLAM2* (MUR-ARTAL; TARDOS, 2017) fornecendo associação de dados multi-mapa na forma de Atlas, composto por um conjunto de mapas desconectados com um ativo que é continuamente desenvolvido e otimizado pelo encadeamento de rastreamento e mapeamento local, que permite usar elementos de mapa de ajuste de pacote de sessões de mapeamento anteriores.

Ele cria um mapa topológico que é representado por uma nuvem de pontos, sendo constituído por *keyframes* $\mathbf{K} = \{\mathbf{k_1}, \mathbf{k_2}, ..., \mathbf{k_r}\}$ que armazenam todas as *features* extraídas, a pose da câmera $\mathbf{T_a}$, um conjunto de pontos $\mathbf{P} = \{\mathbf{p_1}, \mathbf{p_2}, ..., \mathbf{p_s}\}$ que guardam as respectivas posições no ambiente tridimensional e seus descritores de *feature* ORB que o gerou. Os pontos do mapa são associados com os *keyframes* visíveis. Também é construído um grafo de co-visibilidade que adiciona uma aresta entre dois *keyframes* se estes possuírem dois pontos observáveis em comum. Como este grafo pode conter muitas ares-

tas, ele passa por uma otimização que é realizada após o fechamento de *loop*, gerando um grafo essencial que mantém todos os *keyframes* mas mantém apenas arestas com maiores pesos (pontos em comum).

Figura 9 – Diagrama de sistema do *ORB-SLAM3*. Com etapas em azul adicionadas na versão dois e em verde na versão três do algoritmo.



Fonte: Autor.

Apresenta um sistema *SLAM* visual-inercial monocular e estéreo que integra três tipos de associação de dados: recursos *ORB* para associação de dados de curto e médio prazo, um gráfico de co-visibilidade para limitar a complexidade de rastreamento e mapeamento, bem como fechamento de laço e relocalização com a biblioteca *DBoW2*. Isso significava que, para a época, era o sistema visual mais capaz até o momento, atingindo níveis de precisão além dos sistemas existentes (TOURANI et al., 2022). O principal caso de falha ocorre quando se apresenta com ambientes de baixa textura, quando seus métodos de associação de dados são limitados a curto e médio prazo.

A estrutura é composta por três *threads* que rodam em paralelo. O *thread* de rastreamento ou localização é responsável por processar as informações do sensor e calcular a pose do quadro atual em comparação com o mapa ativo em tempo real, ao mesmo tempo em que decide se ele se torna um quadro-chave. Ele obtém um mapa de visibilidade local a partir do gráfico de co-visibilidade de modo a apontar um quadro-chave de referência com a maior similaridade com o quadro atual k_{ref}. Os pontos deste mapa local são reprojetados e a pose é otimizada com os pares de pontos reprojetados e *features* obtidos.

Por fim é decidido se o *frame* atual será utilizado como um novo *keyframe* k_a se uma série de condições forem atendidas, como no mínimo 50 pontos rastreados, 20 *frames* desde a última seleção e um máximo de 90 por cento de pontos em comum com os da referência.

O *thread* de mapeamento refina o mapa usando o ajuste de pacote visual, adicionando *keyframes* e pontos ao mapa ativo e removendo os redundantes. Ele processa cada nova inserção do *thread* de rastreamento atualizando o gráfico de co-visibilidade ao adicionar um novo nó para k_a e as arestas detectadas entre ele e os vizinhos. É extraída uma representação de k_a com *DBoW2* para ajudar na triangulação de novos pontos e aplicado um filtro para remover pontos com baixa qualidade, sendo que para ser mantido um ponto ele deve ser rastreado em pelo menos um quarto dos *frames* onde é visível e ser rastreado em ao menos três *keyframes*. Finalmente, ela cria novos pontos no mapa através de *features* ORB e a triangulação de profundidade, erro de reprojeção e consistência de escala.

Por fim, a *thread* de fechamento de laços tenta detectar regiões comuns entre o Atlas e o mapa ativo atual, sendo responsável por correções de laço e fusão de diferentes mapas em um único. A busca por *loops* acontece a cada novo quadro-chave k_a inserido no mapa e é feita através de um cálculo de similaridade que faz uso de *DBoW2* (GALVEZ-LóPEZ; TARDOS, 2012) que busca o vizinho com a menor similaridade com k_a, para então considerar todos os outros *keyframes* que possuem uma similaridade maior que aquela como candidatos em conjuntos de três conectados pelo grafo de co-visibilidade. Para calcular o erro entre o candidato e k_a são inicialmente encontradas correspondências entre pontos nos mapas de ambos e realizado um método de consenso de amostra aleatória (*RANSAC*) para estimar a similaridade. Se estes forem suportados por pontos suficientes, o *loop* é efetuado (CAMPOS; ELVIRA; RODRIGUEZ, 2021).

O *Bundle Adjustment* é feito utilizando o método de Levenberg-Marquardt da Equação 2.1. A orientação da câmera R e a posição t são otimizadas para minimizar o erro de reprojeção entre os pontos correspondentes X^i e *keyframes* $X_{(\cdot)^i}$ monoculares e é representado pela Equação 2.2.

$$\{R,t\} = argmin_{R,t} \sum_{i \in X} \rho(\left\| X^i_{(\cdot)} - \pi_{(\cdot)}(RX^i + t) \right\|^2_{\sum}) \tag{2.2}$$

Onde $\sum$ é uma matriz de covariância associada ao ponto e ρ é uma função de custo de Huber. $\pi_{(\cdot)}$ é uma função de projeção monocular representada na Equação 2.3

$$\pi_{(\cdot)} \left(\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \right) = \begin{bmatrix} f_x \frac{X}{Z} + c_x \\ f_y \frac{Y}{Z} + c_y \end{bmatrix} \tag{2.3}$$

Onde c são os pontos principais e f o comprimento focal da câmera. O *ORB-SLAM3* se dá pela biblioteca de código aberto mais completa para *SLAM* visual disponível atual-

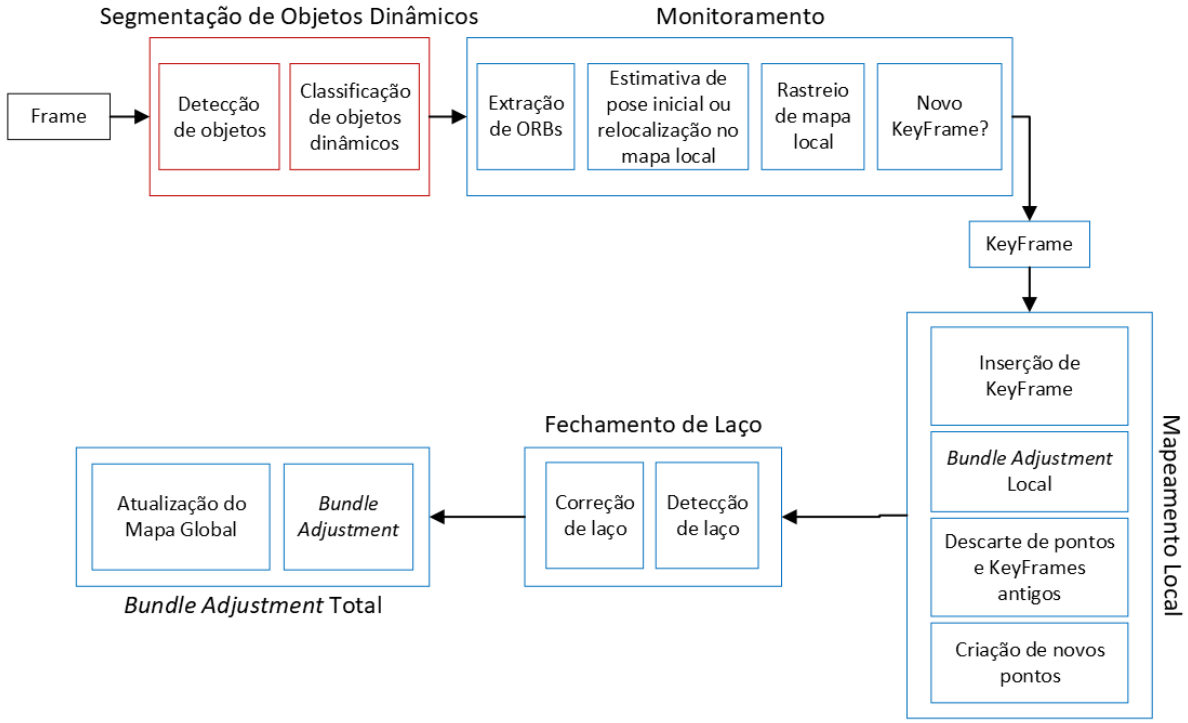
mente (TOURANI et al., 2022). Apesar do algoritmo possuir modos de câmera monocular, RGB-D, estenopeica (*pinhole*) e olho de peixe, para este trabalho foi utilizado apenas o modo monocular.

2.11 DYNASLAM

O *DynaSLAM* (BESCOS et al., 2018) apresenta um sistema que se baseia no *ORB-SLAM2*, adicionando os recursos de detecção dinâmica de objetos e pintura de fundo fornecida por uma instância da rede *Mask R-CNN* (HE et al., 2017) em execução juntamente com o sistema *SLAM* básico .

Ele pode ser usado em cenários dinâmicos para configurações monoculares, estéreo e RGB-D e é capaz de detectar objetos em movimento por geometria de multi-visualização, aprendizado profundo ou ambos. O *DynaSLAM* tende a superar a precisão das linhas de base visuais padrão do *SLAM* em cenários altamente dinâmicos (BESCOS et al., 2018).

Figura 10 – Diagrama de sistema do *DynaSLAM*. Com etapas em vermelho da parte do *Mask R-CNN* e em azul do *ORB-SLAM2*.



Fonte: Autor.

Ao usar o *Mask R-CNN*, a maioria dos objetos dinâmicos pode ser segmentado e não usado para rastreamento e mapeamento, mas nem todos devem ser considerados

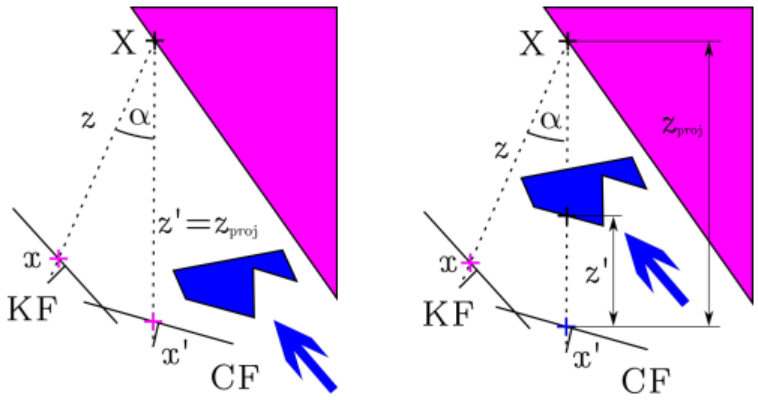
porque às vezes não são necessariamente dinâmicos, mas móveis, como uma pessoa parada no lugar ou um livro apoiado sobre uma mesa. Os autores lidaram com esse problema comparando *keyframes* sobrepostos anteriores, levando em conta a distância e a rotação entre eles e o quadro atual, projetando um valor de profundidade para cada objeto, rotulando-os como dinâmicos apenas se esse valor atingir um limite definido pelo movimento da câmera no determinado momento.

Algumas classes de objetos pertencentes ao algoritmo que possuem maior probabilidade de representar objetos dinâmicos são selecionadas (pessoa, bicicleta, carro, motocicleta, ônibus, caminhão, etc.) e é considerado que para a maioria dos ambientes, os objetos dinâmicos que podem surgir fazem parte destes grupos.

Após os objetos dinâmicos serem segmentados, a pose da câmera é rastreada utilizando a parte restante da imagem. Este rastreamento é uma versão modificada do *ORB-SLAM2* que possui um funcionamento similar ao exposto na seção anterior que projeta as *features* extraídas no quadro e busca correspondências nas áreas estáticas dos mesmos, buscando minimizar o erro de reprojeção.

Para cada *frame* processado, são selecionados até cinco *frames* anteriores com maiores sobreposições e similaridades, que é definido levando em conta a distância e rotação entre o eles. Posteriormente é computada a projeção de cada ponto x dos *keyframes* anteriores dentro do atual, assim obtendo os pontos x' bem como a profundidade projetada z_{proj}.

Figura 11 – Processo de decisão se um ponto x' pertence a um objeto estático ou dinâmico. À esquerda estático, à direita dinâmico.



Fonte: (BESCOS et al., 2018).

Na figura 11 é visto o ponto x sendo projetado no *frame* atual CF utilizando a pose da câmera e resultando no ponto x' com profundidade z'. O ângulo de paralaxe α entre x e x' é então calculado e se exceder o valor de 30^o, o ponto correspondente será ocultado. Um pixel é marcado como dinâmico se a diferença $\Delta z = z_{proj} - z'$ for maior que o limite definido por τ_z. Este limite é calculado manualmente tomando a distância medida

de diversos objetos dinâmicos diferentes tirados do *dataset*. Esta metodologia geométrica complementa a rede *Mask R-CNN*, de forma a que cada uma das duas possui diferentes vantagens e desvantagens. Se um objeto é detectado pelos dois métodos, a máscara de segmentação geométrica é utilizada, se ele for visto somente pelo método baseado em *deep learning*, a máscara de segmentação utilizada é a provinda dele mesmo.

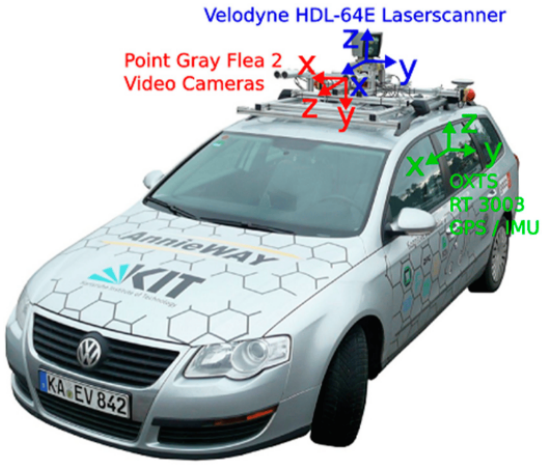
3 MATERIAIS E MÉTODOS

Neste capítulo serão apresentados as ferramentas utilizadas no desenvolvimento e experimentações deste trabalho, bem como a metodologia de testes e rotinas planejadas para que resultados consistentes fossem obtidos. Para as experimentações com ambientes realistas foram utilizadas sequências dos conjuntos de dados *KITTI* e *TUM RGB-D*, já para as simulações foram gravadas novas sequências a partir de triagens realizadas em *software Gazebo* e um sistema construído com *ROS*.

3.1 CONJUNTO DE DADOS KITTI

O conjunto de dados (do inglês *dataset*) *KITTI Vision Benchmark Suite* (GEIGER et al., 2013) provém uma série de dados extraídos através de um carro com uma plataforma de direção autônoma, gravando diversas sequências de rotas realizadas em ruas e rodovias nos arredores da cidade de Karlsruhe, Alemanha.

Figura 12 – Carro equipado com a plataforma sensorial utilizada para captura do data set.



Fonte: (GEIGER et al., 2013).

Dentre os dados coletados, câmeras RGB e *grayscale* foram utilizadas para capturar imagens e um sensor laser *Velodyne* aliado de um sistema de localização por *GPS* foram responsáveis por capturar os valores de referência ao *ground truth* (GEIGER et al., 2013).

Figura 13 – Sequência de imagens exemplo extraídas de uma sequência do KITTI Dataset.

Fonte: (GEIGER et al., 2013).

O *KITTI* foi escolhido pois, além de ser um dos mais completos e difundidos no meio acadêmico, representa situações reais de ambientes externos (ruas/estradas), onde a aplicação dos algoritmos pode ser relevante ao desenvolvimento de tecnologias tais quais direção autônoma ou assistências.

3.2 CONJUNTO DE DADOS TUM RGB-D

O *dataset TUM RGB-D* (STURM et al., 2012) contém sequências de ambientes internos provindas de sensores RGB-D agrupados em diversas categorias para avaliar reconstrução de objetos e métodos de odometria e *SLAM* sob diferentes condições de iluminação, textura e estruturais.

Figura 14 – Sequência de imagens exemplo extraídas de uma sequência do *dataset* TUM RGB-D.

Fonte: (STURM et al., 2012).

Esta ferramenta foi utilizada nos experimentos deste trabalho pois representa situações antagônicas ao conjunto de dados anterior, estas sendo ambientes internos com iluminação artificial de escritórios com objetos dinâmicos diferentes, constituídos principalmente de pessoas.

3.3 IMPLEMENTAÇÕES DE REDES

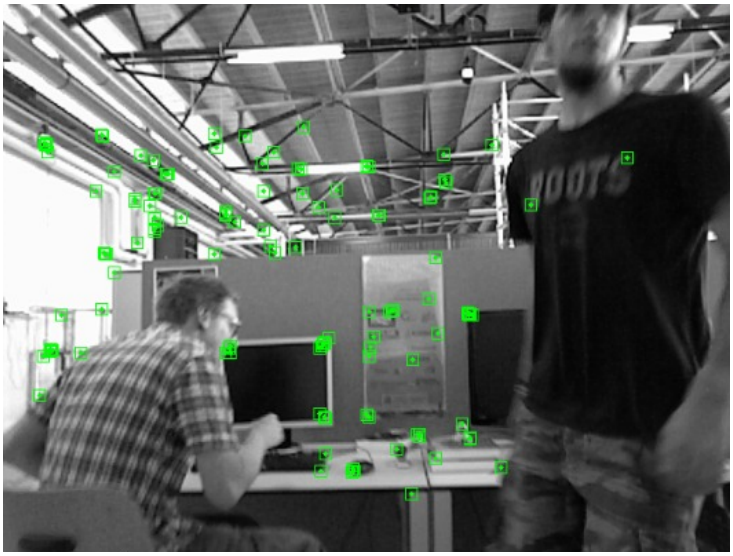
As redes *ORB-SLAM3* e *DynaSLAM* são de código aberto e possuem modelos autorais disponíveis em repositórios na internet, estes foram adaptados conforme necessidade e parâmetros foram modificados. O sistema operacional utilizado para a experimentação foi o Ubuntu 20.04.

3.4 ORB-SLAM3

Para a implementação do *ORB-SLAM3* foi clonado o repositório disponível em (CAMPOS et al., 2021) e modificados alguns parâmetros e aspectos do código como o número máximo de *ORBs* por *frame*, dados de parametrização das câmeras e limite para

a detecção de cantos *FAST*. O compilador utilizado foi o *C++11* e a biblioteca de visualização e interface de usuário *Pangolin* para o acompanhamento dos mapeamentos em tempo real. Para manipulação de imagens foi utilizada a biblioteca *OpenCV* versão 4.2, DBoW2 para reconhecimento de ambiente e g2o para otimizações não-lineares. Python foi utilizado para cálculo de alinhamento de trajetórias e plotagem dos resultados.

Figura 15 – *ORB-SLAM3* rodando no *TUM RGB-D*.



Fonte: Autor.

3.5 DYNASLAM

O *DynaSLAM* também possui um repositório público que os autores disponibilizaram (BESCOS; FACIL, 2019) e foi utilizado de ponto de partida para a implementação da mesma. Neste caso se atestou a necessidade de utilizar a plataforma *Docker* para isolar o sistema em um contêiner a parte do sistema, visto que a execução das aplicações necessitavam de algumas versões antigas de bibliotecas que já não possuem mais suporte ativo.

Como este algoritmo depende de uma instância de *ORB-SLAM2* e uma de *Mask R-CNN* para funcionar corretamente, foi necessário instalar os pré-requisitos de ambas. A primeira utilizou o compilador *C++11*, *Pangolin* para visualização, *OpenCV* 3.2 para manipulação de imagens. A segunda utilizou *Python* 2.7 e as bibliotecas *Keras* e *TensorFlow* para aprendizagem profunda, bem como um modelo pré-treinado com o conjunto de dados COCO (LIN et al., 2014). Python também foi utilizado para cálculo de alinhamento de trajetórias e plotagem dos resultados.

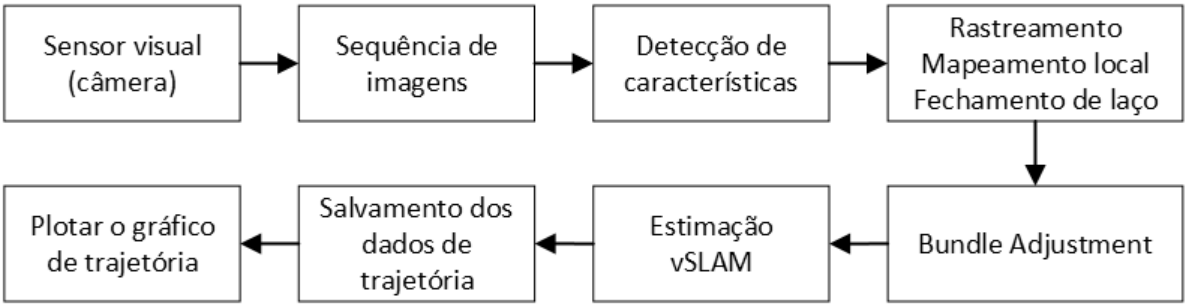
Figura 16 – *DynaSLAM* rodando no *TUM RGB-D*.

Fonte: Autor.

3.6 METODOLOGIA DE TESTES

Os testes foram conduzidos em um computador equipado com uma placa de vídeo *Geforce RTX 2070*, API para computação paralela CUDA 11, linguagens de programação *Python* 3.9 e *C++* e biblioteca de aprendizado de máquina *TensorFlow*. Na figura 17 é apresentado um diagrama que exemplifica o fluxo de trabalho realizado para adquirir os resultados que serão expostos posteriormente.

Figura 17 – Diagrama de metodologia de testes.



Fonte: Autor.

Para fins de comparação, foram realizados primeiramente testes com sequências pertencentes ao próprio conjunto de dados *KITTI* e *TUM RGB-D* que foram utilizados para treinamento de ambas as redes e análise para identificar quais suas vantagens e desvantagens. Posteriormente, foram criadas sequências em simulação utilizando os robôs móveis *Turtlebot* e *Husky* com uma câmera acoplada, onde o vídeo capturado por ela bem como

a odometria do robô será gravada através de *scripts* em *python* em forma de imagens e vetores de translação ao longo de um período de tempo, utilizando o software Gazebo em alguns ambientes diversos. Essas sequências foram alimentadas para as duas redes a fim de analisar a capacidade seus niveis de portabilidade, bem como se a mudança de sequências reais para simulação iria afetar muito na capacidade de predição de movimentação espacial delas.

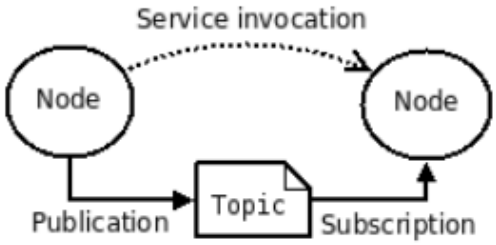
3.7 SIMULAÇÕES

Esta seção apresenta os recursos utilizados para a realização das simulações tanto em ambientes internos e externos. O sistema utilizado é o *ROS Noetic* com simulação tridimensional feita com o *software Gazebo*, cada um dos robôs utilizados possui um modelo dentro deste simulador que realizará diferentes trajetórias, que foram gravadas em forma de sequências de imagens *.png* provindas de uma câmera e os dados de odometria em forma de valores de coordenada x, y e z em cada momento que uma imagem é gerada.

3.8 ROBOT OPERATIONAL SYSTEM

O Sistema Operacional de Robôs ou ROS se trata de uma seleção de softwares para simulação e desenvolvimento de robôs, ele provê uma plataforma padrão para desde desenvolvedores amadores até indústrias, que ajudam com a pesquisa e prototipação desde projeto até produção. Começou a ser desenvolvido pouco antes de 2007 na Universidade de Stanford com o intuito de se tornar um ponto de partida para quem não necessariamente tinha conhecimento de todos os escopos necessários para realizar projetos de robótica tivesse mais agilidade no desenvolvimento deles (ROS. . . , 2023).

Figura 18 – Modelo de funcionamento de nós do ROS.



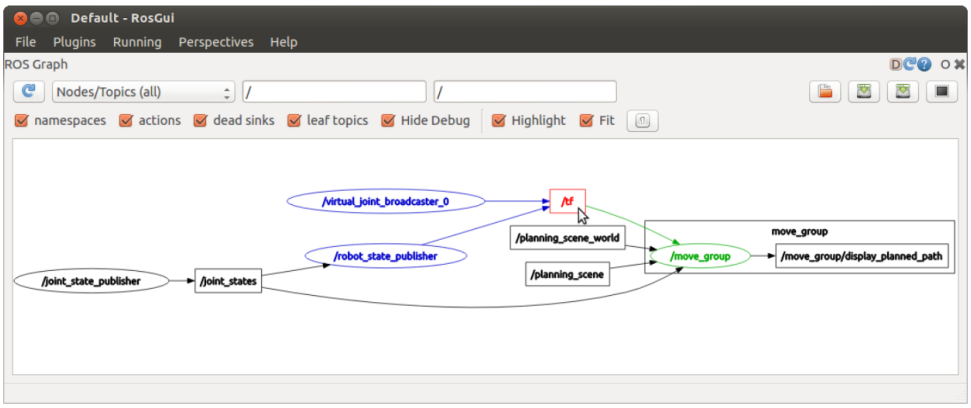
Fonte: (ROS. . . , 2023).

O ROS tem código aberto e uma comunidade muito ativa que mantém diversos *frameworks* e algoritmos que incrementam ainda mais a gama de possíveis aplicações. Também possui uma arquitetura simples, com uma estruturação em formato de nós.

Como representado pela figura 18, cada nó funciona essencialmente como um executável que realiza processos computacionais, sendo que eles podem estar rodando em uma só máquina. Uma máquina deverá exercer a função de MESTRE e irá executar o comando “ROSCORE” e inicializará todos os serviços necessários para que os nós se comuniquem por meio de tópicos (equivalentes a variáveis) que podem ser publicados e subscritos pelos nós, utilizando o protocolo de comunicação TCP/IP. Esta arquitetura traz alguns benefícios para o sistema, como uma complexidade de código reduzida e uma melhor tolerância a falhas devido a que eventuais panes são isoladas aos nós específicos.

A estrutura do sistema pode ser facilmente visualizada por meio do comando “RQT GRAPH”, que gera uma interface gráfica representativa que pode ser útil para a depuração tanto de tópicos individuais quanto de projetos inteiros (ROS. . . , 2023). Na figura 19 é possível observar um exemplo de saída da função.

Figura 19 – Exemplo de RQT GRAPH.



Fonte: (ROS. . . , 2023).

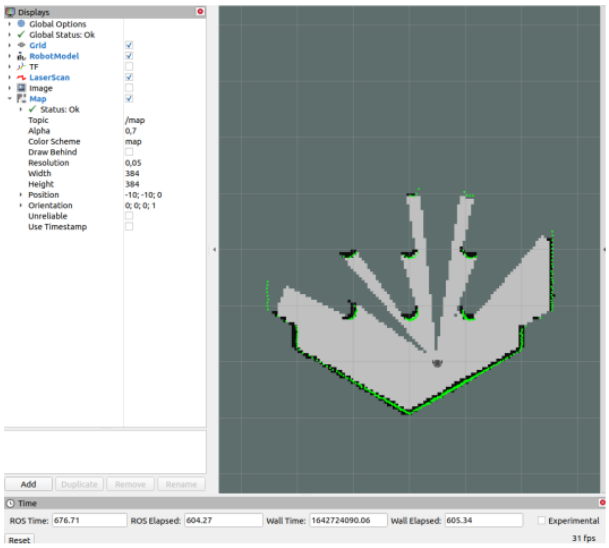
O ROS passou por diversas versões ao longo de seus mais de 15 anos de existência, e a escolha dela deve ser feita baseada nas compatibilidades necessárias para a utilização do hardware específico de cada trabalho, tendo isso em mente a versão utilizada neste trabalho foi a ROS Noetic no Ubuntu 20.04.

3.9 RVIZ

O RVIZ é uma interface gráfica que possibilita a visualização de diversas informações de tópicos disponíveis utilizando *plugins*. Com ele pode-se fazer a leitura da odometria do robô dentro de uma simulação tridimensional.

Um exemplo típico de aplicação deste software se dá por uma representação da simulação à direita acompanhada das leituras do sensor selecionado à esquerda, como visto na figura 20:

Figura 20 – Interface RVIZ.



Fonte: Autor.

3.10 GAZEBO

É um simulador tridimensional de robótica de código aberto muito utilizado na indústria e no meio acadêmico. A união dele com o ROS torna possível a criação de cenários com objetos, obstáculos e topografia customizados, bem como o teste de algoritmos, de regressão e treinamento de redes neurais.

Figura 21 – Interface Gazebo.

Fonte: Autor.

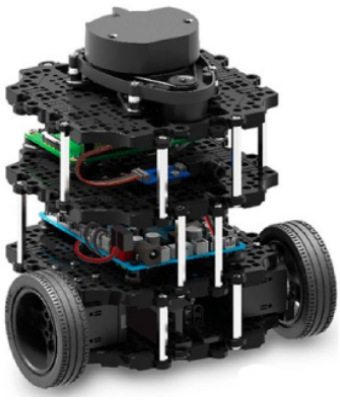
Neste trabalho foram utilizados um ambiente de simulação e um externo para os testes, visando uma maior variedade situacional. A figura 21 mostra a interface de usuário do *Gazebo* rodando a simulação de um robô móvel *Husky*.

3.11 TURTLEBOT

Normalmente os sensores embarcados são cruciais no momento da escolha do modelo com o qual se irá trabalhar e se tornam um dos principais critérios a se considerar, por isso é importante escolher algo que se adeque as condições projetadas. No caso deste trabalho, o único equipamento embarcado além dos *encoders* responsáveis pela odometria foi a câmera monocular, portanto, ambos os robôs escolhidos se tratam de plataformas que aceitam a ligação de um dispositivo do tipo em sua composição.

O TurtleBot se caracteriza por ser um robô móvel pessoal de baixo custo com software de código aberto que é produzido desde 2010 e possui diversos modelos com peculiaridades distintas. Para as simulações em ambientes internos, o robô escolhido foi o TurtleBot 3, modelo Burger. Ele pode ser visto na figura 22.

Figura 22 – Robô móvel Turtlebot 3 modelo Burger.



Fonte: (ROBOTIS, 2023)

3.12 HUSKY

O Husky é um robô móvel de médio porte robusto que é desenhado para pesquisas na área de robótica em ambientes adversos. Ele pode ser customizado com uma variedade de sensores e sistemas para atender diversas atividades e possui uma construção preparada para enfrentar ambientes hostis. A figura 23 apresenta um modelo do Husky.

Figura 23 – Robô móvel Husky.

Fonte: (CLEARPATH, 2023)

Uma câmera RGB Realsense D435 foi acoplada a cada um dos robôs e foi realizada uma calibração inicial para definir os pontos focais e valores de distorção da lente e o controle dos robôs foi feito remotamente por meio de um controle. Os dados da câmera e de odometria são postados de forma periódica em tópicos respectivos para cada um dos modelos de robô, e para acessá-los utilizou-se de um *script Python* com o pacote *rospy* que possibilita criar subscrições nesses locais.

3.13 AMBIENTES DE SIMULAÇÃO

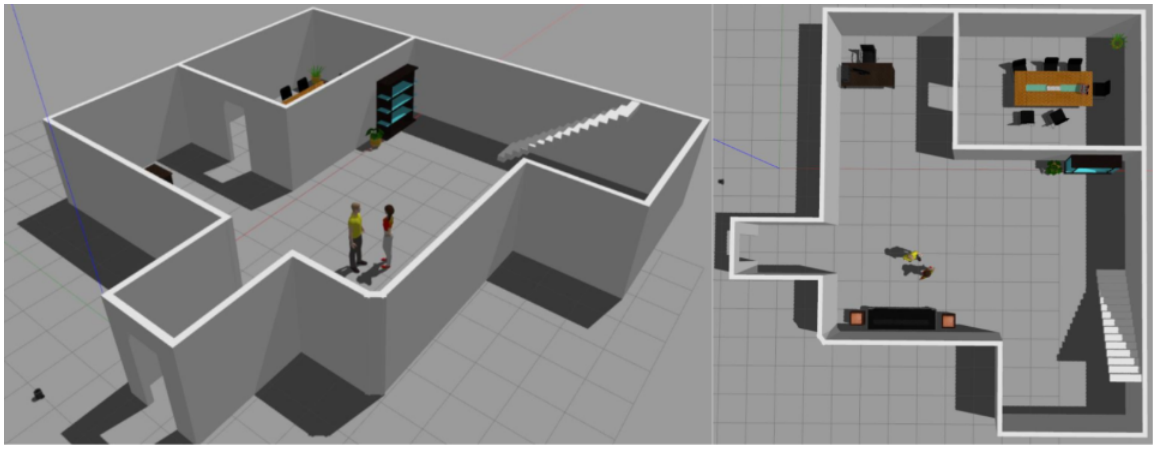
Para o ambiente externo foi utilizado um mapa disponibilizado gratuitamente online, o objetivo desta escolha foi tentar emular as situações encontradas nas sequências provenientes dos testes em situações do mundo real, com um espaço com presença de objetos dinâmicos (carros e pedestres), bem como uma densidade maior de informações que possam ser detectadas como *features*. O ambiente externo é representado na figura 24.

Figura 24 – Ambiente de simulação externo (cidade) para gravação das novas sequências.

Fonte: Autor

O ambiente interno foi desenvolvido através de um modelo de uma residência com obstáculos, móveis e pessoas. Neste caso manteve-se um padrão com menor resolução e densidade para analisar como os algoritmos se comportariam em uma situação mais desafiadora. A figura 25 representa duas perspectivas deste ambiente interno.

Figura 25 – Ambiente de simulação interno (residência) para gravação das novas sequências.



Fonte: Autor

4 RESULTADOS

Esta seção discutirá os resultados obtidos por meio de experimentação ao longo da realização do trabalho. De início foram testadas as sequências provindas dos conjuntos de dados escolhidos, de forma a tentar reproduzir trajetórias condizentes com os dados de *ground truth*.

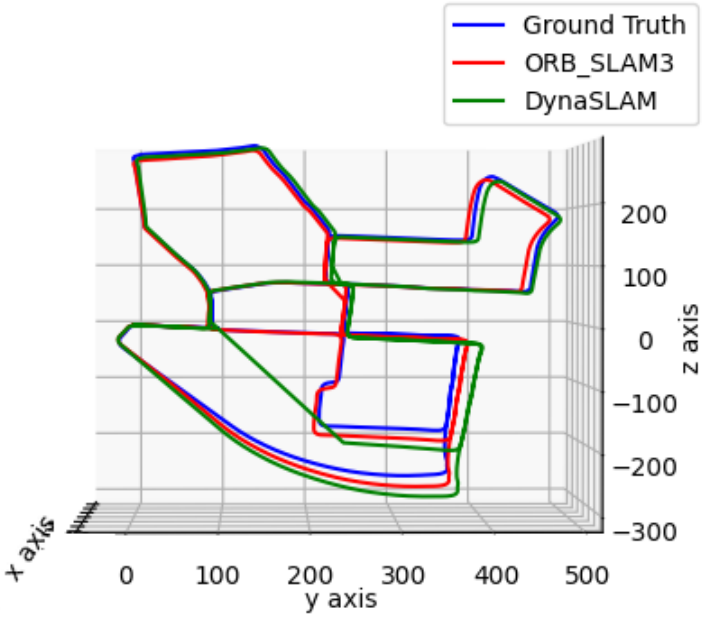
Durante os experimentos alguns parâmetros foram ajustados visando a otimização das saídas das redes como o número de *ORBs* que poderiam ser extraídas a cada *frame* e a sensibilidade para o qual o algoritmo de detecção de cantos teria. A gama de cores das imagens provindas dos *datasets* foram transformadas para *grayscale* afim de tentar não enviesar as redes em razão dos canais de cores.

4.1 KITTI

Na figura 26, tanto o *ORB-SLAM3* quanto o *DynaSLAM* foram testados na sequência 00 do *KITTI*, que representa um carro navegando no ambiente da figura 24 com diversos objetos presentes na cena. Na maior parte do tempo, ambos os algoritmos obtiveram resultados semelhantes ao *ground truth*. No entanto, é importante notar que houve um ponto específico na sequência em que o carro encontrou uma área com menos objetos, resultando em um número reduzido de recursos *ORB* disponíveis para detecção.

Durante a sequência 00, ambos os métodos perderam temporariamente a noção de sua posição espacial no mapa. No entanto, o *DynaSLAM* demonstrou desempenho superior ao se relocalizar mais cedo em comparação com o *ORB-SLAM3*. Isso indica que o *DynaSLAM* exibiu melhor desempenho geral durante esse cenário específico.

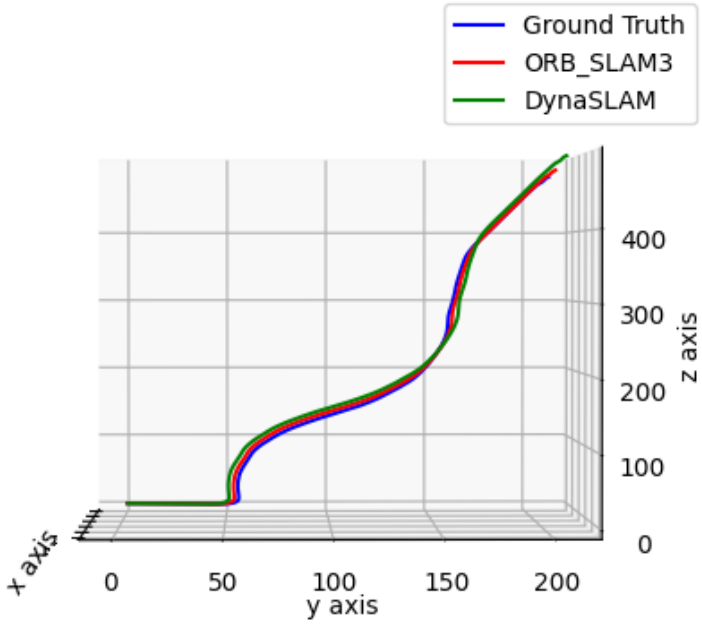
Figura 26 – Ground truth e trajetórias estimadas por DynaSLAM e ORB-SLAM3 no KITTI sequência 00.



Fonte: Autor.

Na figura 27 observamos a sequência 03 do *KITTI*, que apresenta um percurso mais linear com menos curvas fechadas e desvios. É evidente a partir dos resultados que ambos os sistemas demonstraram desempenho semelhante, com um nível de precisão que os manteve próximos do *ground truth*.

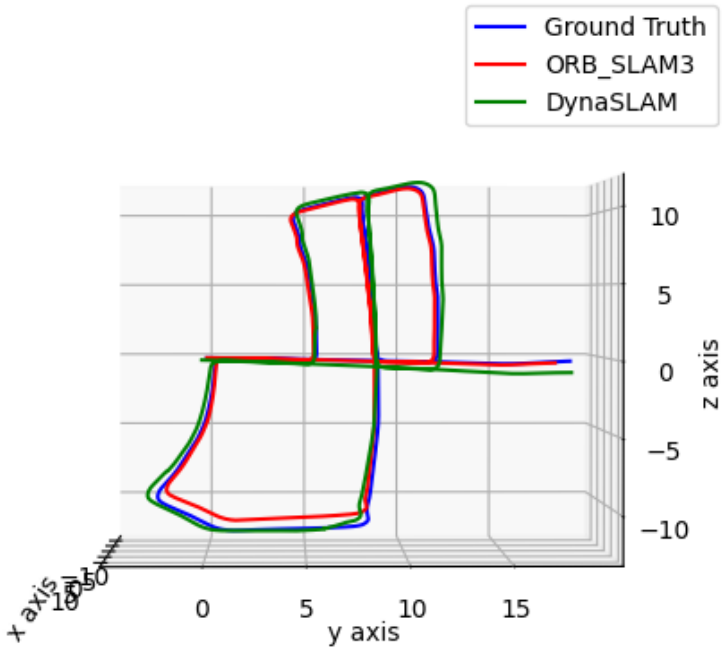
Figura 27 – Ground truth e trajetórias estimadas por DynaSLAM e ORB-SLAM3 no KITTI sequência 03.



Fonte: Autor.

Na figura 28, que representa os resultados obtidos na sequência número 05, percebe-se uma situação semelhante a sequência 00, no sentido de que *DynaSLAM* performou marginalmente melhor que o *ORB-SLAM3*. Esta é uma sequência que possui laços onde a plataforma percorre o mesmo caminho mais de uma vez, então atesta que ambos os algoritmos conseguem se localizar de maneira satisfatória em ambientes que já são conhecidos e interligar os dois pontos do mapa nos quais o laço é convertido. É importante também frisar o papel que o *BA* desempenha nestas situações para corrigir erros acumulados de trajetória e angulação para que o fechamento de laço seja desempenhado com sucesso.

Figura 28 – Ground truth e trajetórias estimadas por DynaSLAM e ORB-SLAM3 no KITTI sequência 05.



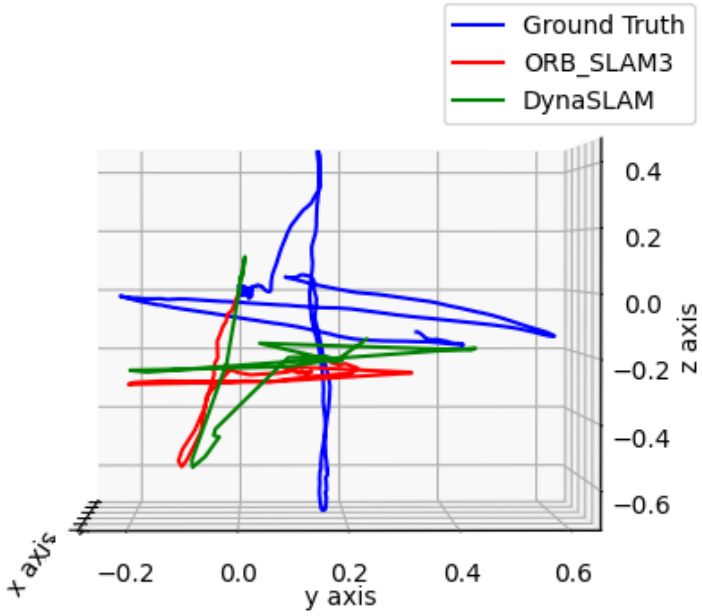
Fonte: Autor.

4.2 TUM RGB-D

Na figura 29, examinamos os resultados obtidos do conjunto de dados *"fr3-walking-xyz"* do *TUM RGB-D*, que apresenta um cenário consideravelmente mais desafiador para ambas as abordagens. Este conjunto de dados apresenta um ambiente menos texturizado em comparação com os exemplos anteriores. Nesse cenário desafiador, ambos os métodos encontraram dificuldades em rastrear com precisão o movimento e aderir ao verdadeiro caminho. Notavelmente, ambos os algoritmos perderam o controle de sua posição no meio da sequência.

A figura 30 mostra os resultados da sequência *"freiburg3-long-office-household"* do TUM, onde ambos os algoritmos exibem desempenho comparável. Notavelmente, nesta sequência, ambos os métodos demonstraram melhores resultados, com taxas de erro mantidas em um nível menor que as anteriores.

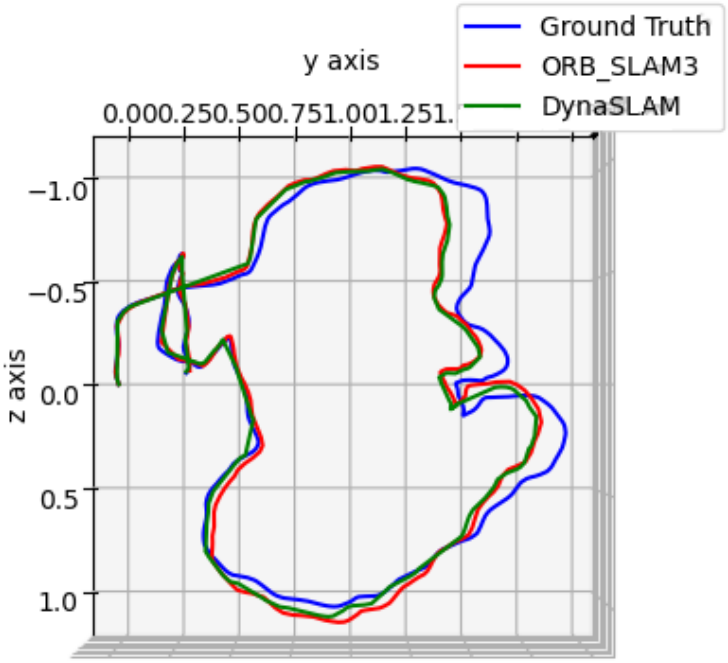
Figura 29 – Ground truth e trajetórias estimadas por DynaSLAM e ORB-SLAM3 no TUM sequência fr3-walking-xyz.



Fonte: Autor.

A comparação da distância percorrida pelas câmeras monoculares usando ambas as técnicas, conforme mostrado na Tabela 1, pode ser utilizada para comparar a capacidade de correção de erros de translação dos algoritmos, destacando que o DynaSLAM segue de perto os valores reais na maioria das sequências. Ele exibe uma taxa média de erro de aproximadamente 24%, indicando um desvio relativamente pequeno da distância correta. Por outro lado, o ORB SLAM3 desvia em média 28% da distância correta, implicando um nível de erro ligeiramente maior em comparação com o DynaSLAM.

Figura 30 – Ground truth e trajetórias estimadas por DynaSLAM e ORB-SLAM3 no TUM sequência fr3-long-office-household.



Fonte: Autor.

Tabela 1 – Distância percorrida pela câmera monocular (m)

Sequência	Ground Truth	ORB-SLAM3	DynaSLAM
KITTI 00	3662.13	3083.56	3290.67
KITTI 03	697.39	523.48	543.41
KITTI 05	112.95	85.23	90.97
fr3 walking xyz	4.88	2.77	2.98
fr3 long office	8.33	7.97	7.74

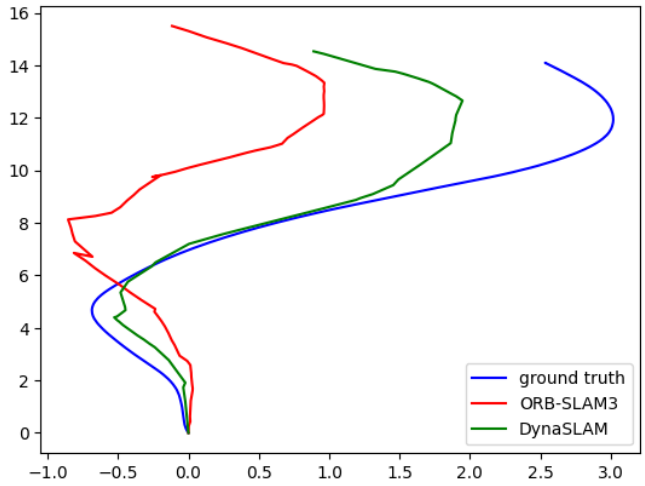
4.3 SIMULAÇÕES EM AMBIENTE EXTERNO

Como esperado os resultados das simulações ficaram aquém das trajetórias reais e isto se deve a diversos fatores mas principalmente a falta de textura, ou densidade material, presente nas imagens gravadas.

A figura 31 apresenta o caminho percorrido em uma sequência gravada em simulação em ambiente externo e nele foi descrita uma rota em formato de “S”. Ao analisar as rotas percebe-se que o ORB-SLAM3 teve problema em acompanhar o movimento correto e estimar a distância entre uma curva e outra, enquanto o DynaSLAM mesmo que tendo

seguido mais próximo ao correto ainda assim presumiu que a segunda curva aconteceu antes do que o realizado.

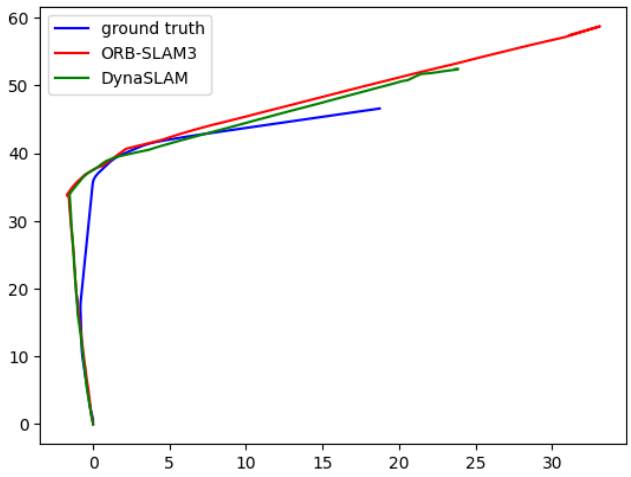
Figura 31 – Ground truth e trajetórias estimadas por DynaSLAM e ORB-SLAM3 no ambiente simulado externo.



Fonte: Autor.

Já a figura 32 denota que o real trajeto percorrido foi antevisto com uma acurácia maior. Com um circuito mais simples, em formato de “L” foi possível obter melhores resultados do que na sequência anterior.

Figura 32 – Ground truth e trajetórias estimadas por DynaSLAM e ORB-SLAM3 no ambiente simulado externo.



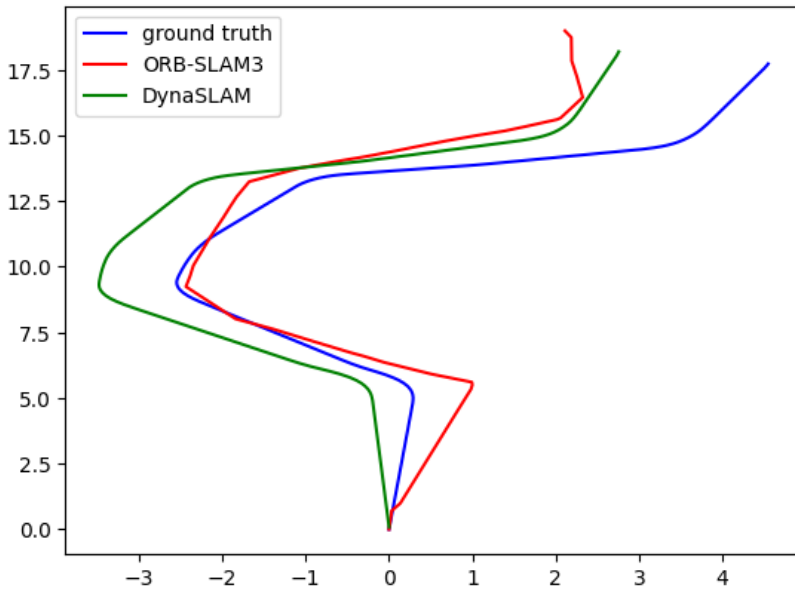
Fonte: Autor.

4.4 SIMULAÇÕES EM AMBIENTE INTERNO

Partindo para as simulações em ambientes internos, seus resultados foram menos satisfatórios. Os algoritmos tiveram dificuldades para seguir as trajetórias principalmente por diferenças de taxas de rotação, advindas em problemas com as fases de *Bundle Adjustment*.

Na figura 33 foi realizada uma trajetória em formato de S semelhante à anterior, e é visto que os dois algoritmos performam de maneira semelhante, mesmo com a mudança de local.

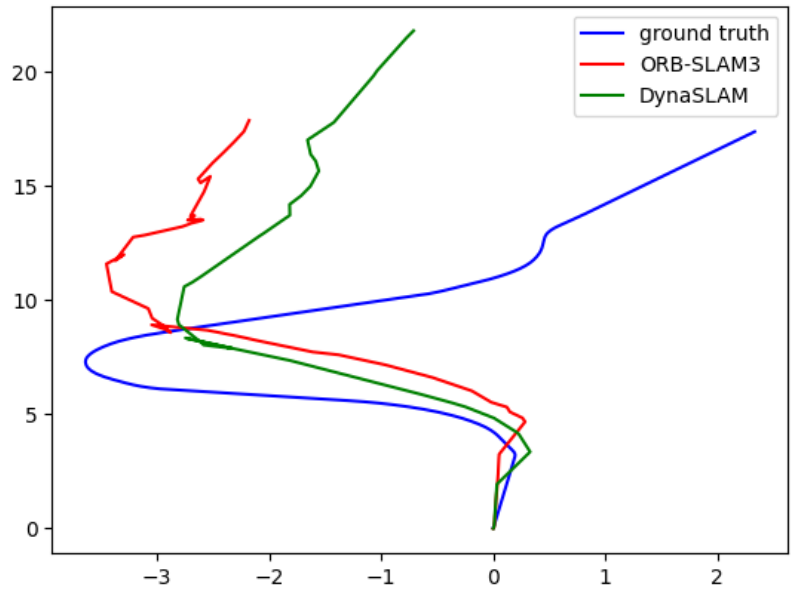
Figura 33 – Ground truth e trajetórias estimadas por DynaSLAM e ORB-SLAM3 no ambiente simulado interno.



Fonte: Autor.

Durante a sequência da figura 34 houve uma perda de localização durante a primeira curva para ambos os algoritmos que logo foi relocalizada, porém não antes sem acumular um erro significativo nos dois casos onde causou as acurácias a decaírem muito, ficando distantes do caminho ideal em um nível muito maior que em testes anteriores.

Figura 34 – Ground truth e trajetórias estimadas por DynaSLAM e ORB-SLAM3 no ambiente simulado interno.



Fonte: Autor.

No que diz respeito as distâncias percorridas em simulação a tendência que se formou com os testes anteriores foi repetida com o DynaSLAM chegando marginalmente mais próximo que o ORB-SLAM3 nas triagens virtuais.

Tabela 2 – Distância percorrida pela câmera monocular simulação (m)

Sequência	**Ground Truth**	**ORB-SLAM3**	**DynaSLAM**
HUSKY 01 EXT.	53.54	56.81	55.67
HUSKY 02 EXT.	50.86	71.31	59.05
TBOT 01 INT.	20.04	23.64	20.75
TBOT 02 INT.	20.46	21.18	20.53

Sequências mais longas com laços presentes na trajetória foram gravadas porém não atingiram um resultado satisfatório, portanto não foram expostas neste trabalho.

4.5 TEMPO DE PROCESSAMENTO

Algumas considerações que foram observadas durante as experimentações e possuem relevância serão expostas a seguir.

Nos dias de hoje a complexidade computacional é de suma importância para qualquer sistema que contenha *Machine Learning*, pois algoritmos que façam usos de redes neurais geralmente requerem um poderio de processamento muito acima da média. Durante os testes foi identificada uma diferença significativa no tempo que tomava para um terminar determinada sequência em comparação com o outro.

Por esta razão foram medidos os tempos médios para a obtenção e processamento de *frames* de cada um dos algoritmos comparados no trabalho (Tabela 3) e se identificou que o DynaSLAM levava em média próximo de 15 vezes mais tempo para tal. Esse fenômeno acontece pela natureza da rede de segmentação presente no algoritmo mais lento, que toma a maior parte do tempo para detectar e extrair a máscara de objetos presentes nas imagens.

Tabela 3 – Tempo médio para o processamento de um frame entre todos os testes realizados (s)

Algoritmo	**Mediana**	**Média**
ORB-SLAM3	0.0195	0.021
DynaSLAM	0.348	0.353

5 CONCLUSÃO

Foram alcançados os objetivos estipulados para este trabalho. Ambientes de teste foram desenvolvidos para rodar os algoritmos e ambientes de simulação com os materiais e métodos definidos, bem como as comparações que foram propostas e aprofundamento teórico na área de *Visual SLAM*.

Ao analisar os resultados podemos concluir que ambos os métodos apresentam um desempenho satisfatório, de modo a capturar a maior parte das movimentações realizadas pelo sujeito, quando apresentados a cenários que não possuem nenhuma de suas limitações. Enquanto eles tem dificuldades para manter o rastreamento em ambientes de baixa textura, seus recursos em situações semelhantes às sequências de conjuntos de dados *KITTI* se mostram muito mais próximos do *ground truth*. A associação de dados de longo prazo parece ser o maior empecilho destas abordagens e é o primeiro aspecto que parece desmoronar quando a situação inicia a se tornar mais desafiadora.

A implementação do *DynaSLAM* de uma máscara de objeto dinâmico e filtro em cima do *ORB-SLAM2* com *Mask R-CNN* melhora seu desempenho por alguma margem. Os resultados coletados apontam que quando os dois algoritmos executaram o mesmo cenário, embora que ambos ainda perdessem a geolocalização ocasionalmente, o *DynaSLAM* se mostrou capaz de realizar o fechamento de laço no mapa antes do *ORB-SLAM3*. Também conseguiu ficar mais próximo da verdadeira distância percorrida na maioria das sequências de teste. Percebe-se que os algoritmos tem melhor desempenho quando apresentados com situações reais, onde há maior densidade de informações e consequentemente mais possibilidades de propriedades a serem extraídas e monitoradas das imagens.

O tempo para a obtenção de *frames* e o valor deste sendo uma ordem de magnitude maior que o *ORB-SLAM3* inviabiliza a utilização do *DynaSLAM* em uma série de aplicações que requerem *SLAM* em tempo real mas não tem disponível tanto poder de processamento embarcado. Este é um problema atualmente mas em um futuro próximo com avanço na área de computação pode se tornar irrisório. Além disto, o *DynaSLAM* tende a perder sua geolocalização quando um objeto dinâmico ocupa a maior parte da corrente imagem, como um carro ou pessoa.

Visual SLAM é uma área em constante evolução e que tem tomado os holofotes recentemente, então avanços vêm sendo feitos e o papel de trabalhos como este é tentar direcionar pesquisas futuras a caminhos que gerem frutos mais relevantes no caminho. Por essa razão, pesquisas futuras sobre o tópico podem incluir, mas não estão limitadas a, adicionar os objetos dinâmicos no mapa gerado, utilizando os dados comprovados dos recursos *ORB* e a segmentação de objetos já presentes no *DynaSLAM* para fazer isso e substituir o *Mask R-CNN* por um algoritmo de segmentação de objetos mais moderno para melhorar o tempo de processamento por *frame*.

REFERÊNCIAS

ARAUJO, A. **Uma Arquitetura utilizando Algoritmo Genético Interativo e Aprendizado de Máquina aplicado ao Problema do Próximo Release**. 03 2015. Tese (Doutorado), 03 2015.

BEGHDADI, A.; MALLEM, M. A. comprehensive overview of dynamic visual slam and deep learning: concepts, methods and challenges. **Machine Vision and Applications**, v. 33, p. 54, 2022.

BESCOS, B.; FACIL, J. M. **DynaSLAM**. 2019. https://github.com/BertaBescos/DynaSLAM.

BESCOS, B. et al. Dynaslam: Tracking, mapping, and inpainting in dynamic scenes. **IEEE Robotics and Automation Letters**, v. 3, n. 4, p. 4076–4083, 2018.

BOBBE, M. et al. An automated rapid mapping solution based on orb slam 2 and agisoft photoscan api. **IMAV 2017**, 09 2017.

BOUAZZAOUI, I. E.; FLOREZ, S.; OUARDI, A. E. Enhancing rgb-d slam performances considering sensor specifications for indoor localization. **IEEE Sensors Journal**, v. 22, p. 4970–4977, 2022.

CALONDER, M. et al. Brief: Binary robust independent elementary features. **Eur. Conf. Comput. Vis.**, v. 6314, p. 778–792, 09 2010.

CAMPOS, C.; ELVIRA, R.; RODRIGUEZ, J. J. G. Orb-slam3: An accurate open-source library for visual, visual–inertial, and multimap slam. **IEEE Transactions on Robotics**, v. 37, n. 6, p. 1874–1890, 2021.

CAMPOS, C. et al. **ORB-SLAM3**. 2021. https://github.com/UZ-SLAMLab/ORB_SLAM3.

CLEARPATH. **HUSKY UNMANNED GROUND VEHICLE**. 2023. Disponível em: <https://clearpathrobotics.com/husky-unmanned-ground-vehicle-robot/>.

CRAMARIUC, A. et al. maplab 2.0 – a modular and multi-modal mapping framework. **IEEE Robotics And Automation Letters**, v. 8, p. 520–527, 2 2023.

DEEP Learning. 2022. Disponível em: <https://www.deeplearningbook.com.br/>.

DRAELOS, R. **Segmentation: U-Net, mask R-CNN, and medical applications**. 2020. Disponível em: <https://glassboxmedicine.com/2020/01/21/segmentation-u-net-mask-r-cnn-and-medical-applications/>.

GALVEZ-LóPEZ, D.; TARDOS, J. D. Bags of binary words for fast place recognition in image sequences. **IEEE Transactions on Robotics**, v. 28, n. 5, p. 1188–1197, 2012.

GEIGER, A. et al. Vision meets robotics: The kitti dataset. **The International Journal of Robotics Research**, v. 32, n. 11, p. 1231–1237, 2013.

GETTING Started with Mask R-CNN for Instance Segmentation. 2020. Disponível em: <https://www.mathworks.com/help/vision/ug/getting-started-with-mask-r-cnn-for-instance-segmentation.html>.

GIRSHICK, R. Fast r-cnn. **2015 IEEE International Conference on Computer Vision (ICCV)**, p. 1440–1448, 2015.

GIRSHICK, R. et al. Rich feature hierarchies for accurate object detection and semantic segmentation. **2014 IEEE Conference on Computer Vision and Pattern Recognition**, p. 580–587, 2014.

HE, K. et al. Mask r-cnn. **2017 IEEE International Conference on Computer Vision (ICCV)**, p. 2980–2988, 2017.

JESUS, K. et al. Comparison of visual slam algorithms orb-slam rtab-map and sptam in internal and external environments with ros. **Latin American Robotics Symposium (LARS), 2021 Brazilian Symposium On Robotics (SBR), And 2021 Workshop On Robotics In Education (WRE)**, v. 2, p. 216–221, 2021.

KLEIN, G.; MURRAY, D. Parallel tracking and mapping for small ar workspaces. **6th IEEE ACM Int. Symp. Mixed mented Reality**, p. 225–234, 2007.

KRAMER, A. **Orb-Slam-loop-closure**. 2019. Disponível em: <http://andrewjkramer.net-/beyond-ekf-slam/orb-slam-loop-closure/>.

KUMAR, A. et al. Comparison of visual slam and imu in tracking head movement outdoors. **Behav Res**, 2022.

KüMMERLE, R. et al. G2o: A general framework for graph optimization. **2011 IEEE International Conference on Robotics and Automation**, p. 3607–3613, 2011.

LAVIN, A.; GRAY, S. **Fast Algorithms for Convolutional Neural Networks**. 2015.

LEE, J. et al. Improved real-time monocular slam using semantic segmentation on selective frames. **IEEE Transactions on Intelligent Transportation Systems**, v. 24, n. 3, p. 2800–2813, 2023.

LEMAIRE, T. et al. Vision-based slam: Stereo and monocular approaches. **International Journal of Computer Vision**, v. 74, p. 343–364, 2007.

LIN, T.-Y. et al. Microsoft coco: Common objects in context. **Computer Vision – ECCV 2014**, Cham, p. 740–755, 2014.

MISHRA, M. **Convolutional Neural Networks, explained**. 2020. Disponível em: <https:/-/towardsdatascience.com/convolutional-neural-networks-explained-9cc5188c4939>.

MORÉ, J. J. The levenberg-marquardt algorithm: Implementation and theory. **Numerical Analysis**, Springer Berlin Heidelberg, Berlin, Heidelberg, p. 105–116, 1978.

MUR-ARTAL, R.; MONTIEL, J.; TARDOS, J. O.-s. A. Versatile and accurate monocular slam system. **IEEE Transactions On Robotics**, v. 31, p. 1147–1163, 10 2015.

MUR-ARTAL, R.; TARDOS, J. D. Orb-slam2: An open-source slam system for monocular, stereo, and rgb-d cameras. **IEEE Transactions on Robotics**, v. 33, n. 5, p. 1255–1262, 2017.

NEWCOMBE, R. A.; LOVEGROVE, S. J.; DAVISON, A. J. Dtam: Dense tracking and mapping in real-time. **Proc. IEEE Int. Conf. Comput. Vision**, p. 2320–2327, 2011.

REN, S. et al. **Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks**. 2016.

ROBOTIS. **TURTLEBOT 3**. Robotis, 2023. Disponível em: <https://www.robotis.us-/turtlebot-3/>.

ROS Wiki. 2023. Disponível em: <http://wiki.ros.org/>.

RUBLEE, E. et al. Orb: an efficient alternative to sift or surf. **Proceedings of the IEEE International Conference on Computer Vision**, p. 2564–2571, 11 2011.

STURM, J. et al. A benchmark for the evaluation of rgb-d slam systems. **2012 IEEE/RSJ International Conference on Intelligent Robots and Systems**, p. 573–580, 2012.

TOURANI, A. et al. Visual slam: What are the current trends and what to expect? **Sensors**, v. 22, n. 23, p. 9297, 2022.

ZITNICK, C. L.; DOLLÁR, P. Edge boxes: Locating object proposals from edges. **Computer Vision – ECCV 2014**, Cham, p. 391–405, 2014.

ANEXO A – CÓDIGO GRAVAÇÃO DE SEQUÊNCIAS

```
# Importando bibliotecas necessarias
import rospy # Python para ROS
from sensor_msgs.msg import Image # Image e o tipo de mensagem
from nav_msgs.msg import Odometry
from cv_bridge import CvBridge # Para conversao de imagens no formato
    ↪ OpenCV
import cv2 # OpenCV
import message_filters

# Funcao para ler dados do topico
def ler_dados():

  data = []
  x = float
  y = float
  z = float
  count = 0
  frame = 0
  # Nó vai se inscrever no tópico image_raw e odom
  camera = message_filters.Subscriber('/realsense/color/image_raw',
      ↪ Image)
  odom = message_filters.Subscriber('/husky_velocity_controller/odom',
      ↪ Odometry)

  ts = message_filters.ApproximateTimeSynchronizer([camera, odom],
      ↪ queue_size=10, slop=0.5)
  ts.registerCallback(callback)
  rospy.spin()
# Funcao de callback
def callback(camera, odom):
  # Usado para converter imagens do ROS para OpenCV
  br = CvBridge()
  # Converte imagem ROS para OpenCV
  frame = br.imgmsg_to_cv2(camera)
  x = odom.pose.pose.position.x
  y = odom.pose.pose.position.y
```

```
33    z = odom.pose.pose.position.z
34    salvar()
35  # Funcao para salvar imagem e dados
36  def salvar():
37    if frame is not None and x is not None and y is not None and z is not
      ↪  None:
38
39      print('Processando frame {}...'.format(count))
40
41      yo = str(count).zfill(6)
42
        ↪  cv2.imwrite("/home/mateus/Documents/data/images/image_0/{}.png".format(yo),
        ↪  frame)
43      # Mostra imagem
44      cv2.imshow("camera", frame)
45      count += 1
46      data.append([0,x,y,z])
47      cv2.waitKey(100)
48      frame = None
49      x = None
50      y = None
51      z = None
52
53  # Funcao principal
54  if __name__ == '__main__':
55
56    # Define o nome do nó
57    rospy.init_node('listener')
58    try:
59      ler_dados()
60    except rospy.ROSInterruptException:
61      pass
62
63    # Fechar tudo quando terminar
64    with open("/home/mateus/Documents/data/images/times.txt", "w") as f:
65      for i in data:
66        f.write(str(i[0]) + " " + str(i[1]) + " " + str(i[2]) + " " +
          ↪  str(i[3]) + "\n")
67    cv2.destroyAllWindows()
```

ANEXO B – CÓDIGO PLOTAGEM DE TRAJETÓRIAS

```
!pip install -U -q PyDrive
from pydrive.auth import GoogleAuth
from pydrive.drive import GoogleDrive
from google.colab import auth
from oauth2client.client import GoogleCredentials

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation

auth.authenticate_user()
gauth = GoogleAuth()
gauth.credentials = GoogleCredentials.get_application_default()
drive = GoogleDrive(gauth)

def rotate_vector(data, angle):
    # make rotation matrix
    theta = np.radians(angle)
    co = np.cos(theta)
    si = np.sin(theta)
    rotation_matrix = np.array(((co, -si), (si, co)))
    # rotate data vector
    rotated_vector = data.dot(rotation_matrix)
    # return index of elbow
    return rotated_vector

downloaded = drive.CreateFile({'id':'1Wl-vapAopt6ylEhIosFaEOn9W7VLJf2s'})
downloaded.GetContentFile('husky_gt.txt')
ground_truth_00 = pd.read_csv('husky_gt.txt')

downloaded = drive.CreateFile({'id':'1TxjLJbAboeOPevcH_Ezbg_Uu5LCzP6k3'})
downloaded.GetContentFile('husky2_dyna.txt')
DYNASLAM_KITTI_00 = pd.read_csv('husky_dyna.txt')

downloaded = drive.CreateFile({'id':'1lNPfPp6VmN_WdBtrmyvS78O4uAJZUwxX'})
```

```
downloaded.GetContentFile('husky_orb.txt')
ORBSLAM3_KITTI_00 = pd.read_csv('husky_orb.txt')

ORBSLAM3_KITTI_00_x = []
ORBSLAM3_KITTI_00_y = []
DYNASLAM_KITTI_00_x = []
DYNASLAM_KITTI_00_y = []
ground_truth_00_x = []
ground_truth_00_y = []

for i in range(0, len(ORBSLAM3_KITTI_00['id x z y']), 1):
  ORBSLAM3_KITTI_00_x.append(ORBSLAM3_KITTI_00['id x z y'][i].split('
  ↪ ')[1])
  ORBSLAM3_KITTI_00_y.append(ORBSLAM3_KITTI_00['id x z y'][i].split('
  ↪ ')[3])

for i in range(0, len(DYNASLAM_KITTI_00['id x z y']), 1):
  DYNASLAM_KITTI_00_x.append(DYNASLAM_KITTI_00['id x z y'][i].split('
  ↪ ')[1])
  DYNASLAM_KITTI_00_y.append(DYNASLAM_KITTI_00['id x z y'][i].split('
  ↪ ')[3])

for i in range(0, len(ground_truth_00['id x z y']), 1):
  ground_truth_00_x.append(ground_truth_00['id x z y'][i].split(' ')[1])
  ground_truth_00_y.append(ground_truth_00['id x z y'][i].split(' ')[2])

for i in range(len(ORBSLAM3_KITTI_00_x)):
  ORBSLAM3_KITTI_00_x[i] = float(ORBSLAM3_KITTI_00_x[i])
  ORBSLAM3_KITTI_00_y[i] = float(ORBSLAM3_KITTI_00_y[i])

zeros = [DYNASLAM_KITTI_00_x[0], DYNASLAM_KITTI_00_y[0]]
for i in range(len(DYNASLAM_KITTI_00_x)):
  DYNASLAM_KITTI_00_x[i] =
  ↪ (float(DYNASLAM_KITTI_00_x[i])-float(zeros[0]))
  DYNASLAM_KITTI_00_y[i] =
  ↪ (float(DYNASLAM_KITTI_00_y[i])-float(zeros[1]))

zeros = [ground_truth_00_x[0], ground_truth_00_y[0]]
for i in range(len(ground_truth_00_x)):
```

```
70    gtx = ground_truth_00_x[i]
71    gty = ground_truth_00_y[i]
72    ground_truth_00_x[i] = (float(gtx) - float(zeros[0])) * np.cos(71.3) +
      ↪  (float(gty)- float(zeros[1])) * np.sin(71.3)
73    ground_truth_00_y[i] = (float(gty)- float(zeros[1])) * np.cos(71.3) -
      ↪  (float(gtx) - float(zeros[0])) * np.sin(71.3)
74
75  plt.plot(ground_truth_00_x[0:len(ground_truth_00_x):1] ,
    ↪  ground_truth_00_y[0:len(ground_truth_00_y):1], label='ground truth',
    ↪  color='b')
76  plt.plot(ORBSLAM3_KITTI_00_x[0:len(ORBSLAM3_KITTI_00_x):1] ,
    ↪  ORBSLAM3_KITTI_00_y[0:len(ORBSLAM3_KITTI_00_y):1], label='ORB-SLAM3',
    ↪  color='r')
77  plt.plot(DYNASLAM_KITTI_00_x[0:len(DYNASLAM_KITTI_00_x):1] ,
    ↪  DYNASLAM_KITTI_00_y[0:len(DYNASLAM_KITTI_00_y):1], label='DynaSLAM',
    ↪  color='g')
78
79  plt.legend()
80  plt.show()
81
82  #Transformando em array
83  df_ground_truth_x = np.array(ground_truth_00_x)
84  df_ground_truth_y = np.array(ground_truth_00_y)
85
86  df_orb_slam3_Mono_x = np.array(ORBSLAM3_KITTI_00_x)
87  df_orb_slam3_Mono_y = np.array(ORBSLAM3_KITTI_00_y)
88
89  df_dyna_Mono_x = np.array(DYNASLAM_KITTI_00_x)
90  df_dyna_Mono_y = np.array(DYNASLAM_KITTI_00_y)
91
92  distance_df_ground_truth_x = []
93  distance_df_ground_truth_y = []
94  distance_df_ground_truth_z = []
95
96  distance_df_orb_slam3_Mono_x = []
97  distance_df_orb_slam3_Mono_y = []
98  distance_df_orb_slam3_Mono_z = []
99
100 distance_df_dyna_Mono_x = []
```

```
101 distance_df_dyna_Mono_y = []
102 distance_df_dyna_Mono_z = []
103
104 #Math of trajectory
105 for i in range(len(df_ground_truth_x) - 1):
106   distance_df_ground_truth_x.append(abs(df_ground_truth_x[i] -
      ↪  df_ground_truth_x[i+1]))
107   distance_df_ground_truth_y.append(abs(df_ground_truth_y[i] -
      ↪  df_ground_truth_y[i+1]))
108
109 #Math of trajectory
110 for i in range(len(df_orb_slam3_Mono_x) - 1):
111   distance_df_orb_slam3_Mono_x.append(abs(df_orb_slam3_Mono_x[i] -
      ↪  df_orb_slam3_Mono_x[i+1]))
112   distance_df_orb_slam3_Mono_y.append(abs(df_orb_slam3_Mono_y[i] -
      ↪  df_orb_slam3_Mono_y[i+1]))
113
114 #Math of trajectory
115 for i in range(len(df_dyna_Mono_x) - 1):
116   distance_df_dyna_Mono_x.append(abs(df_dyna_Mono_x[i] -
      ↪  df_dyna_Mono_x[i+1]))
117   distance_df_dyna_Mono_y.append(abs(df_dyna_Mono_y[i] -
      ↪  df_dyna_Mono_y[i+1]))
118
119 #Pitagoras
120 print('Trajetória ground Truth: ',  np.sqrt(
    ↪  sum(distance_df_ground_truth_x)**2 +
    ↪  sum(distance_df_ground_truth_y)**2 ))
121 print('Trajetória orb_slam3_Mono: ',  np.sqrt(
    ↪  sum(distance_df_orb_slam3_Mono_x)**2 +
    ↪  sum(distance_df_orb_slam3_Mono_y)**2 ))
122 print('Trajetória dyna_Mono: ',  np.sqrt( sum(distance_df_dyna_Mono_x)**2
    ↪  + sum(distance_df_dyna_Mono_y)**2 ))
123
124 #Distance
125 ground_truth_trajectory =     np.sqrt( sum(distance_df_ground_truth_x)**2
    ↪  + sum(distance_df_ground_truth_y)**2 )
```

```
126 orb_slam3_Mono_trajectory =   np.sqrt(
    ↪  sum(distance_df_orb_slam3_Mono_x)**2 +
    ↪  sum(distance_df_orb_slam3_Mono_y)**2 )
127 orb_dyna_trajectory =   np.sqrt( sum(distance_df_dyna_Mono_x)**2 +
    ↪  sum(distance_df_dyna_Mono_y)**2 )
```

ANEXO C – CÓDIGO MODELO DYNASLAM - KITTI

```cpp
#include<iostream>
#include<algorithm>
#include<fstream>
#include<chrono>
#include<iomanip>
#include <unistd.h>
#include<opencv2/core/core.hpp>

#include "Geometry.h"
#include "MaskNet.h"
#include"System.h"

using namespace std;

void LoadImages(const string &strSequence, vector<string>
    &vstrImageFilenames,
                vector<double> &vTimestamps);

int main(int argc, char **argv)
{
    if(argc != 4 && argc != 5)
    {
        cerr << endl << "Usage: ./mono_kitti path_to_vocabulary
            path_to_settings path_to_sequence (path_to_masks)" << endl;
        return 1;
    }

    // Caminhos das imagens
    vector<string> vstrImageFilenames;
    vector<double> vTimestamps;
    LoadImages(string(argv[3]), vstrImageFilenames, vTimestamps);

    int nImages = vstrImageFilenames.size();

    // Inicializar Mask R-CNN
    DynaSLAM::SegmentDynObject *MaskNet;
```

```
35      if (argc==5)
36      {
37          cout << "Loading Mask R-CNN. This could take a while..." << endl;
38          MaskNet = new DynaSLAM::SegmentDynObject();
39          cout << "Mask R-CNN loaded!" << endl;
40      }
41
42      // Criar sistema de SLAM. Inicializa o sistema e o deixa pronto para
        ↪  receber frames.
43      ORB_SLAM2::System
        ↪  SLAM(argv[1],argv[2],ORB_SLAM2::System::MONOCULAR,true);
44
45      // Vetor para rastrear o tempo
46      vector<float> vTimesTrack;
47      vTimesTrack.resize(nImages);
48
49      cout << endl << "-------" << endl;
50      cout << "Start processing sequence ..." << endl;
51      cout << "Images in the sequence: " << nImages << endl << endl;
52
53      // Laco principal
54      cv::Mat im;
55
56      // Configuracao de dilatacao
57      int dilation_size = 15;
58      cv::Mat kernel = getStructuringElement(cv::MORPH_ELLIPSE,
59                                             cv::Size( 2*dilation_size + 1,
                                               ↪  2*dilation_size+1 ),
60                                             cv::Point( dilation_size,
                                               ↪  dilation_size ) );
61
62      for(int ni=0; ni<nImages; ni++)
63      {
64          // Ler arquivo de imagem
65          im = cv::imread(vstrImageFilenames[ni],CV_LOAD_IMAGE_UNCHANGED);
66          double tframe = vTimestamps[ni];
67
68          if(im.empty())
69          {
```

```
70              cerr << endl << "Failed to load image at: " <<
                ↪  vstrImageFilenames[ni] << endl;
71              return 1;
72          }
73
74  #ifdef COMPILEDWITHC11
75          std::chrono::steady_clock::time_point t1 =
            ↪  std::chrono::steady_clock::now();
76  #else
77          std::chrono::monotonic_clock::time_point t1 =
            ↪  std::chrono::monotonic_clock::now();
78  #endif
79
80          // Segmentar as imagens
81          int h = im.rows;
82          int w = im.cols;
83          cv::Mat mask = cv::Mat::ones(h,w,CV_8U);
84          if(argc == 5)
85          {
86              cv::Mat maskRCNN;
87              maskRCNN =
                ↪  MaskNet->GetSegmentation(im,string(argv[4]),vstrImageFilenames[ni].
88                          vstrImageFilenames[ni].begin(),
                            ↪  vstrImageFilenames[ni].end()-10,""));
89              cv::Mat maskRCNNdil = maskRCNN.clone();
90              cv::dilate(maskRCNN, maskRCNNdil, kernel);
91              mask = mask - maskRCNN;
92          }
93
94          // Passar a imagem para o sistema SLAM
95          SLAM.TrackMonocular(im, mask, tframe);
96
97  #ifdef COMPILEDWITHC11
98          std::chrono::steady_clock::time_point t2 =
            ↪  std::chrono::steady_clock::now();
99  #else
100         std::chrono::monotonic_clock::time_point t2 =
            ↪  std::chrono::monotonic_clock::now();
101 #endif
```

```
102
103         double ttrack=
            ↪  std::chrono::duration_cast<std::chrono::duration<double> >(t2
            ↪  - t1).count();
104
105         vTimesTrack[ni]=ttrack;
106
107         // Wait to load the next frame
108         double T=0;
109         if(ni<nImages-1)
110             T = vTimestamps[ni+1]-tframe;
111         else if(ni>0)
112             T = tframe-vTimestamps[ni-1];
113
114         if(ttrack<T)
115             usleep((T-ttrack)*1e6);
116     }
117
118     // Parar tudo
119     SLAM.Shutdown();
120
121     // Estatisticas de tempo
122     sort(vTimesTrack.begin(),vTimesTrack.end());
123     float totaltime = 0;
124     for(int ni=0; ni<nImages; ni++)
125     {
126         totaltime+=vTimesTrack[ni];
127     }
128     cout << "-------" << endl << endl;
129     cout << "median tracking time: " << vTimesTrack[nImages/2] << endl;
130     cout << "mean tracking time: " << totaltime/nImages << endl;
131
132     // Salvar trajetoria
133     SLAM.SaveKeyFrameTrajectoryTUM("KeyFrameTrajectory.txt");
134
135     return 0;
136 }
137
```

```
138 void LoadImages(const string &strPathToSequence, vector<string>
    ↪ &vstrImageFilenames, vector<double> &vTimestamps)
139 {
140     ifstream fTimes;
141     string strPathTimeFile = strPathToSequence + "/times.txt";
142     fTimes.open(strPathTimeFile.c_str());
143     while(!fTimes.eof())
144     {
145         string s;
146         getline(fTimes,s);
147         if(!s.empty())
148         {
149             stringstream ss;
150             ss << s;
151             double t;
152             ss >> t;
153             vTimestamps.push_back(t);
154         }
155     }
156 
157     string strPrefixLeft = strPathToSequence + "/image_0/";
158 
159     const int nTimes = vTimestamps.size();
160     vstrImageFilenames.resize(nTimes);
161 
162     for(int i=0; i<nTimes; i++)
163     {
164         stringstream ss;
165         ss << setfill('0') << setw(6) << i;
166         vstrImageFilenames[i] = strPrefixLeft + ss.str() + ".png";
167     }
168 }
```

ANEXO D – CÓDIGO MODELO ORB-SLAM3 - KITTI

```cpp
#include<iostream>
#include<algorithm>
#include<fstream>
#include<chrono>
#include<iomanip>

#include<opencv2/core/core.hpp>

#include"System.h"

using namespace std;

void LoadImages(const string &strSequence, vector<string>
 ↪  &vstrImageFilenames,
                vector<double> &vTimestamps);

int main(int argc, char **argv)
{
    if(argc != 4)
    {
        cerr << endl << "Usage: ./mono_kitti path_to_vocabulary
         ↪  path_to_settings path_to_sequence" << endl;
        return 1;
    }

    // Caminho das imagens
    vector<string> vstrImageFilenames;
    vector<double> vTimestamps;
    LoadImages(string(argv[3]), vstrImageFilenames, vTimestamps);

    int nImages = vstrImageFilenames.size();

    // Criar sistema de SLAM. Inicializa o sistema e o deixa pronto para
     ↪  receber frames.
    ORB_SLAM3::System
     ↪  SLAM(argv[1],argv[2],ORB_SLAM3::System::MONOCULAR,true);
```

```
33      float imageScale = SLAM.GetImageScale();
34
35      // Vetor para rastrear o tempo
36      vector<float> vTimesTrack;
37      vTimesTrack.resize(nImages);
38
39      cout << endl << "-------" << endl;
40      cout << "Start processing sequence ..." << endl;
41      cout << "Images in the sequence: " << nImages << endl << endl;
42
43      // Laco principal
44      double t_resize = 0.f;
45      double t_track = 0.f;
46
47      cv::Mat im;
48      for(int ni=0; ni<nImages; ni++)
49      {
50          // Ler imagem
51          im = cv::imread(vstrImageFilenames[ni],cv::IMREAD_UNCHANGED);
            ↪  //,cv::IMREAD_UNCHANGED);
52          double tframe = vTimestamps[ni];
53
54          if(im.empty())
55          {
56              cerr << endl << "Failed to load image at: " <<
                ↪  vstrImageFilenames[ni] << endl;
57              return 1;
58          }
59
60          if(imageScale != 1.f)
61          {
62  #ifdef REGISTER_TIMES
63      #ifdef COMPILEDWITHC11
64              std::chrono::steady_clock::time_point t_Start_Resize =
                ↪  std::chrono::steady_clock::now();
65      #else
66              std::chrono::monotonic_clock::time_point t_Start_Resize =
                ↪  std::chrono::monotonic_clock::now();
67      #endif
```

```
68  #endif
69              int width = im.cols * imageScale;
70              int height = im.rows * imageScale;
71              cv::resize(im, im, cv::Size(width, height));
72  #ifdef REGISTER_TIMES
73      #ifdef COMPILEDWITHC11
74              std::chrono::steady_clock::time_point t_End_Resize =
                ↪  std::chrono::steady_clock::now();
75      #else
76              std::chrono::monotonic_clock::time_point t_End_Resize =
                ↪  std::chrono::monotonic_clock::now();
77      #endif
78              t_resize =
                ↪  std::chrono::duration_cast<std::chrono::duration<double,std::milli>
                ↪  >(t_End_Resize - t_Start_Resize).count();
79              SLAM.InsertResizeTime(t_resize);
80  #endif
81          }
82
83  #ifdef COMPILEDWITHC11
84          std::chrono::steady_clock::time_point t1 =
            ↪  std::chrono::steady_clock::now();
85  #else
86          std::chrono::monotonic_clock::time_point t1 =
            ↪  std::chrono::monotonic_clock::now();
87  #endif
88
89          // Passar a imagem para o sistema de SLAM
90          SLAM.TrackMonocular(im,tframe,vector<ORB_SLAM3::IMU::Point>(),
            ↪  vstrImageFilenames[ni]);
91
92  #ifdef COMPILEDWITHC11
93          std::chrono::steady_clock::time_point t2 =
            ↪  std::chrono::steady_clock::now();
94  #else
95          std::chrono::monotonic_clock::time_point t2 =
            ↪  std::chrono::monotonic_clock::now();
96  #endif
97
```

```
98  #ifdef REGISTER_TIMES
99              t_track = t_resize +
                ↪  std::chrono::duration_cast<std::chrono::duration<double,std::milli>
                ↪  >(t2 - t1).count();
100             SLAM.InsertTrackTime(t_track);
101 #endif
102
103         double ttrack=
            ↪  std::chrono::duration_cast<std::chrono::duration<double> >(t2
            ↪  - t1).count();
104
105         vTimesTrack[ni]=ttrack;
106
107         // Esperar o proximo frame
108         double T=0;
109         if(ni<nImages-1)
110             T = vTimestamps[ni+1]-tframe;
111         else if(ni>0)
112             T = tframe-vTimestamps[ni-1];
113
114         if(ttrack<T)
115             usleep((T-ttrack)*1e6);
116     }
117
118     // Parar
119     SLAM.Shutdown();
120
121     // Estatisticas de tempo
122     sort(vTimesTrack.begin(),vTimesTrack.end());
123     float totaltime = 0;
124     for(int ni=0; ni<nImages; ni++)
125     {
126         totaltime+=vTimesTrack[ni];
127     }
128     cout << "-------" << endl << endl;
129     cout << "median tracking time: " << vTimesTrack[nImages/2] << endl;
130     cout << "mean tracking time: " << totaltime/nImages << endl;
131
132     // Salvar trajetoria
```

```
133     SLAM.SaveKeyFrameTrajectoryTUM("KeyFrameTrajectory.txt");
134
135     return 0;
136 }
137
138 void LoadImages(const string &strPathToSequence, vector<string>
    ↪ &vstrImageFilenames, vector<double> &vTimestamps)
139 {
140     ifstream fTimes;
141     string strPathTimeFile = strPathToSequence + "/times.txt";
142     fTimes.open(strPathTimeFile.c_str());
143     while(!fTimes.eof())
144     {
145         string s;
146         getline(fTimes,s);
147         if(!s.empty())
148         {
149             stringstream ss;
150             ss << s;
151             double t;
152             ss >> t;
153             vTimestamps.push_back(t);
154         }
155     }
156
157     string strPrefixLeft = strPathToSequence + "/image_0/";
158
159     const int nTimes = vTimestamps.size();
160     vstrImageFilenames.resize(nTimes);
161
162     for(int i=0; i<nTimes; i++)
163     {
164         stringstream ss;
165         ss << setfill('0') << setw(6) << i;
166         vstrImageFilenames[i] = strPrefixLeft + ss.str() + ".png";
167     }
168 }
```