

Guilherme Cano Lopes

**Mapeamento e localização de robô móvel em  
ambiente interno utilizando *landmarks* e visão  
computacional**

**Sorocaba-SP, Brasil**

**2016**

Guilherme Cano Lopes

## **Mapeamento e localização de robô móvel em ambiente interno utilizando *landmarks* e visão computacional**

Trabalho de Graduação apresentado ao Instituto de Ciência e Tecnologia Campus de Sorocaba, Universidade Estadual Paulista (UNESP), como parte dos requisitos para obtenção do grau de Bacharel em Engenharia de Controle e Automação.

UNESP - Instituto de Ciência e Tecnologia de Sorocaba

Engenharia de Controle e Automação

Orientador: Prof. Dr. Alexandre da Silva Simões

Sorocaba-SP, Brasil

2016

Ficha catalográfica elaborada pela Biblioteca da Unesp  
Instituto de Ciência e Tecnologia – Câmpus de Sorocaba

Lopes, Guilherme Cano.

Mapeamento e localização de um robô móvel em ambiente interno utilizando landmarks e visão computacional / Guilherme Cano Lopes, 2016.

73 f.: il.

Orientador: Alexandre da Silva Simões.

Trabalho de Conclusão de Curso (Graduação) – Universidade Estadual Paulista "Júlio de Mesquita Filho". Instituto de Ciência e Tecnologia (Câmpus de Sorocaba), 2016.

1. Robótica móvel. 2. Localização. 3. Mapeamento. 4. Visão computacional I. Universidade Estadual Paulista "Júlio de Mesquita Filho". Instituto de Ciência e Tecnologia (Câmpus de Sorocaba). II. Título.

Guilherme Cano Lopes

## **Mapeamento e localização de robô móvel em ambiente interno utilizando *landmarks* e visão computacional**

Trabalho de Graduação apresentado ao Instituto de Ciência e Tecnologia Campus de Sorocaba, Universidade Estadual Paulista (UNESP), como parte dos requisitos para obtenção do grau de Bacharel em Engenharia de Controle e Automação.

Trabalho aprovado. Sorocaba-SP, Brasil, 24 de Agosto de 2016:

---

**Prof. Dr. Alexandre da Silva Simões**  
Orientador

---

**Prof. Dra. Ester Luna Colombini**  
Avaliador 1

---

**Prof. Dra. Marilza Antunes de Lemos**  
Avaliador 2

Sorocaba-SP, Brasil  
2016

*Este trabalho é dedicado a meu avô Cirilo Cano.*

# Agradecimentos

Agradeço primeiramente aos meus pais Elaine Cano e João Lopes, por todo apoio que me deram desde sempre. Assim como minha irmã Aline, minhas avós Vilma e Olga, meu padrasto e amigo Alexandre, e toda minha família.

Agradeço à minha namorada Renata Cardetas por se fazer importante na minha vida e estar sempre presente, apesar da distância.

Agradeço aos amigos engenheiros de controle e automação Renato Formigari, Mateus Mussi, Thiago Martins, Rodrigo Ferreira, João Victor, José Miguel e os demais colegas de curso.

Agradeço às minhas amigas engenheiras ambientais Thiemi Igarashi, Isadora Bertini, Vivian Souza, Isabela Moreira.

Agradeço a meus irmãos de consideração Kauê Aftimus Rosa, Yan Aftimus Rosa e Henrique Rodrigues, com quem eu sempre pude contar.

Por fim, agradeço aos professores Alexandre Simões, Marilza de Lemos e Átila Bueno, que me orientaram em meus principais projetos durante a graduação e me ensinaram a complexidade e a beleza da engenharia.

*“Ontem passado. Amanhã futuro. Hoje agora.  
Ontem foi. Amanhã será. Hoje é.  
Ontem experiência adquirida. Amanhã lutas novas.  
Hoje, porém, é a nossa hora de fazer e de construir.  
(Chico Xavier)*

# Resumo

O presente trabalho tem por objetivo implementar e avaliar uma metodologia para estimação de localização de um robô móvel, por meio de visão computacional e marcos artificiais, em um ambiente interno. Adicionalmente, duas outras metodologias foram implementadas para fins de comparação com metodologias consolidadas. São elas: dead reckoning (navegação por sensores internos ou odometria) e *SLAM - Simultaneous Localization And Mapping*, mais especificamente o algoritmo *gmapping*. O trabalho foi desenvolvido com auxílio do software de simulação *V-REP*, da plataforma *ROS - Robot Operating System* e algoritmos de reconstrução 2D-3D presentes na biblioteca *OpenCV*.

**Palavras-chave:** robótica móvel, localização, mapeamento, visão computacional.

# Abstract

The present work has as its main goal to implement and to evaluate a methodology for pose estimation of a mobile robot by means of computer vision and artificial landmarks in an indoor environment. Additionally, two other methods were implemented for purposes of comparison. They are: dead reckoning (navigation by internal sensors and odometry ) and SLAM - Simultaneous Localization And Mapping, more specifically the algorithm gmapping. The work was developed with the help of V-REP simulation software, ROS - Robot Operating System platform and 2D-3D reconstruction algorithms present in the *OpenCV* library.

**Keywords:** mobile robotics, location, mapping, computer vision.

# Listas de ilustrações

Figura 1 – Formas de programação no ambiente <i>V-REP</i> . <i>Script</i> embarcado, <i>plugins</i> , nós do <i>ROS</i> , APIs remotas, soluções customizadas. . . . .	16
Figura 2 – Grafo com representação do uso de tópicos e serviços entre nós do <i>ROS</i> . . . . .	19
Figura 3 – Processo de estimativa de posição 2D-3D. . . . .	22
Figura 4 – Efeitos da distorção na imagem. (Esquerda) Efeito de distorção conhecido por ”almofada de alfinetes“. (Direita) Distorção do tipo barril. . . . .	24
Figura 5 – <i>Frames</i> de referência global e local. . . . .	25
Figura 6 – Representação de um <i>encoder</i> incremental(esquerda) e absoluto (direita). . . . .	26
Figura 7 – Exemplo de ambiente discretizado em um mapa métrico de grades de ocupação. . . . .	28
Figura 8 – Representação topológica de um ambiente. . . . .	29
Figura 9 – Ambiente de operação do robô móvel, elaborado no simulador <i>V-REP</i> . . . . .	32
Figura 10 – Ambiente de operação do robô móvel, elaborado no simulador <i>V-REP</i> . . . . .	32
Figura 11 – Aplicativo <i>Ogre</i> , que utiliza realidade aumentada. . . . .	33
Figura 12 – Marcadores para realidade aumentada. . . . .	33
Figura 13 – Ambiente de operação com marcadores de realidade aumentada posicionados com distância de um metro entre eles. . . . .	34
Figura 14 – Base do robô <i>dr20</i> . . . . .	34
Figura 15 – (Esquerda)Sensor <i>laser rangefinder</i> real. (Direita) Sensor equivalente no ambiente <i>V-REP</i> . . . . .	35
Figura 16 – Representação de veículo com acionamento diferencial. . . . .	36
Figura 17 – Representação do sistema de odometria por meio de grafos do <i>ROS</i> . . . . .	39
Figura 18 – Determinação de localização pelo módulo <i>cv_node</i> . . . . .	43
Figura 19 – Grafo do experimento de localização por visão computacional e <i>landmarks</i> . . . . .	44
Figura 20 – Árvore de transformações padrão para odometria. Cada “seta” representa um matriz de transformação homogênea entre dois <i>frames</i> . . . . .	45
Figura 21 – Árvore de transformações para o pacote <i>gmapping</i> . . . . .	46
Figura 22 – Grafo da metodologia de localização por <i>SLAM</i> . . . . .	46
Figura 23 – Simulação das metodologias no <i>V-REP</i> . . . . .	47
Figura 24 – Localização por <i>dead reckoning</i> . . . . .	48
Figura 25 – Localização por visão computacional e <i>landmarks</i> . . . . .	48
Figura 26 – Relação entre a orientação calculada e medida, obtida experimentalmente. . . . .	49
Figura 27 – Localização por visão computacional e <i>landmarks</i> com correção da orientação. . . . .	50
Figura 28 – Comparação da crença de localização para o caso de mudança súbita da posição do robô (indicada pela seta). . . . .	50

Figura 29 – Marcadores de realidade aumentada posicionados no chão em ambiente real. . . . .	51
Figura 30 – Processo de construção de um mapa por meio do algoritmo para <i>SLAM</i> implementado. . . . .	51
Figura 31 – Localização publicada pelo nó <i>slam_gmapping</i> . . . . .	52
Figura 32 – Mapa construído após SLAM. . . . .	52

# **Lista de abreviaturas e siglas**

ROS	Robot Operating System
V-REP	Virtual Experimentation Platform
SLAM	Simultaneous Localization and Mapping
MCD	Modelo Cinemático Direto
OpenCV	Open-source Computer Vision

# Listas de símbolos

$\xi$	Vetor de localização
<b>A</b>	Matriz de parâmetros intrínsecos
<b>rVec</b>	Vetor de parâmetros extrínsecos de rotação.
<b>tVec</b>	Vetor de parâmetros extrínsecos
$fx$	Distância focal no eixo horizontal
$fy$	Distância focal no eixo vertical
$cx$	Ponto principal da imagem (eixo horizontal)
$cy$	Ponto principal da imagem (eixo vertical)
$R$	Matriz de rotação
<b>J</b>	Matriz jacobiana
<b>q</b>	Vetor de posição angular das juntas do robô
$\lambda$	Vetor de características do robô

# Sumário

<b>1</b>	<b>INTRODUÇÃO</b>	<b>14</b>
<b>2</b>	<b>REVISÃO BIBLIOGRÁFICA</b>	<b>16</b>
<b>2.1</b>	<b><i>V-REP</i></b>	<b>16</b>
<b>2.2</b>	<b><i>Robot Operating System (ROS)</i></b>	<b>16</b>
2.2.1	Principais conceitos	17
<b>2.3</b>	<b>Visão Computacional</b>	<b>20</b>
2.3.1	<i>OpenCV</i>	20
2.3.2	Estimação de posição tridimensional	21
<b>2.4</b>	<b>Localização</b>	<b>24</b>
2.4.1	<i>Localização por dead reckoning</i>	25
2.4.2	<i>Landmarks</i>	26
<b>2.5</b>	<b>Mapeamento</b>	<b>27</b>
2.5.1	Mapas métricos	27
2.5.2	Mapas topológicos	28
<b>2.6</b>	<b><i>SLAM</i></b>	<b>28</b>
<b>3</b>	<b>METODOLOGIA</b>	<b>31</b>
<b>3.1</b>	<b>Elaboração do ambiente de simulação</b>	<b>31</b>
3.1.1	<i>Landmarks</i> artificiais	31
<b>3.2</b>	<b>Robô móvel utilizado</b>	<b>34</b>
3.2.1	Modelo cinemático direto	35
<b>3.3</b>	<b>Experimento desenvolvido</b>	<b>37</b>
3.3.1	<i>Odometria pura</i> ( <i>dead reckoning</i> )	38
3.3.2	Metodologia por visão computacional	39
3.3.3	<i>SLAM</i>	44
<b>4</b>	<b>RESULTADOS E DISCUSSÕES</b>	<b>47</b>
<b>5</b>	<b>CONCLUSÃO</b>	<b>54</b>
	<b>Referências</b>	<b>56</b>
	<b>ANEXOS</b>	<b>58</b>

# 1 Introdução

Os robôs de base fixa, muito presentes na indústria para realização de tarefas repetitivas e com elevada precisão nas linhas de montagem, consistem uma tecnologia bem consolidada e extremamente útil nos processos produtivos. Porém, a fim de eliminar suas limitações de espaço, passaram a ser pesquisadas formas de um robô se movimentar sem supervisão humana, ou seja, de forma autônoma (SIEGWART; NOURBAKHS, 2004). Um robô móvel é dotado de mecanismos de locomoção, o que traz novas frentes de atuação aos mecanismos que, há algumas décadas, eram utilizados essencialmente em processos fabris. Veículos autônomos, *drones* e robôs para a exploração de outros planetas são alguns exemplos das possibilidades promovidas pelo avanço da robótica móvel. Espera-se que, com o contínuo aprimoramento de *hardware* e *software*, os robôs estejam cada vez mais populares. Podendo vir a ser parte da rotina das pessoas e realizar as mais diversificadas tarefas. Da mesma forma que os computadores pessoais, a *web* e os *smartphones* tornaram-se tecnologias populares e acessíveis, os robôs tendem a sofrer um processo similar nos próximos anos.

Segundo Romero et al. (2014) a capacidade de realizar tarefas a partir da interação com o ambiente e a tomada de decisões corretas constitui o maior desafio do campo de robótica móvel. Isto porque o ambiente pode ser desconhecido, irregular, ou pode alterar-se rapidamente com a adição ou retirada de obstáculos, ou mesmo devido à circulação de pessoas no local. Adicionalmente, os sensores e atuadores que compõem os robôs não fornecem dados ideais, estando sujeitos a um determinado grau de incerteza. Verifica-se, portanto, que a *autolocalização* é um problema fundamental da robótica móvel, sendo muito importante para a elaboração de tarefas de maior grau de complexidade.

Dentre as técnicas para a estimativa relativa da posição do robô, a odometria é uma das mais simples de se implementar, e consiste em uma estimativa da posição e orientação do robô móvel com base na rotação de suas rodas e modelo cinemático. Quando a odometria é responsável por todo sistema de localização, é dito que o robô se localiza por reconhecimento passivo, ou *dead reckoning*.

No caso de ambientes internos em que a planta do local de operação do robô móvel é previamente conhecida, é possível "auxiliar" o robô no processo de localização a partir de *landmarks* naturais ou artificiais. *Landmarks* são características discrimináveis no ambiente, e que podem ser identificadas por um sensor. A partir delas, um robô pode obter informação sobre sua localização e sobre o ambiente.

*Landmarks* podem ser constituídas por etiquetas *RFID* - (*Radio Frequency Identification*), códigos de barras, pontos magnéticos no solo, características do próprio ambiente

que sejam identificáveis (*landmarks naturais*), marcadores de realidade aumentada, entre outros marcos do ambiente.

Neste trabalho, implementou-se uma metodologia para obtenção da localização de um robô móvel dotado de rodas em ambiente conhecido a partir de algoritmos de visão computacional, *landmarks* e odometria. Adicionalmente, implementou-se a localização por *dead reckoning* (odometria pura) e uma metodologia para mapeamento e localização simultâneos (*SLAM*). A fim de comparar os resultados obtidos na metodologia proposta com metodologias consolidadas. Os experimentos foram realizados em ambiente simulado (*V-REP*) e a estrutura dos softwares desenvolvidos foram baseadas na interface *ROS*.

Este trabalho encontra-se dividido da seguinte forma: O primeiro capítulo consiste na revisão dos conceitos e ferramentas relacionados ao trabalho desenvolvido. O segundo capítulo apresenta a metodologia para o desenvolvimento do trabalho, com detalhes sobre o ambiente de simulação, modelo do robô utilizado e de cada uma das técnicas aplicadas. O terceiro capítulo consiste na apresentação e discussão das estimativas de localização obtidas. Por fim, são apresentadas as conclusões e considerações finais quanto às metodologias implementadas.

## 2 Revisão Bibliográfica

### 2.1 *V-REP*

O software *V-REP* consiste em um simulador para aplicações em robótica, que define um ambiente de desenvolvimento integrado, baseado na arquitetura de controle distribuída ([COPPELIA, 2016](#)). Ou seja, cada objeto ou modelo pode ser controlado individualmente, de seis possíveis formas (figura 1).

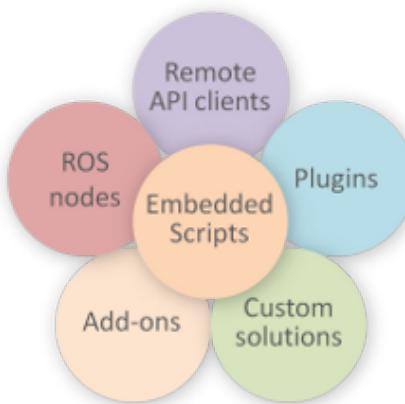


Figura 1 – Formas de programação no ambiente *V-REP*. *Script* embarcado, *plugins*, nós do *ROS*, APIs remotas, soluções customizadas.

Fonte: ([COPPELIA, 2016](#))

No *V-REP*, é possível combinar funcionalidades de alto e baixo nível para desenvolver aplicações como simulações de sistemas de automação, monitoramento remoto, controle de *hardware*, prototipagem, entre outras. Ou seja, consiste em um ambiente simulado muito versátil, o qual permite o desenvolvimento de controladores em linguagens como C/C++, Python, Java, LUA, Matlab, Octave ou Urbi. O software também auxilia a coleta de dados em trabalhos que envolvem robótica móvel. Pois no V-REP é possível computar e armazenar variáveis que são difíceis de mensurar com robôs reais em movimento. Como por exemplo, a trajetória de um robô móvel ou sua orientação em um determinado instante.

### 2.2 *Robot Operating System (ROS)*

O ROS, da sigla em inglês *Robotic Operating System* é definido como um metasistema operacional ou *middleware* destinado à aplicações de robótica. O *ROS* fornece as principais funções que são esperadas de um sistema operacional, tal como permitir

controle de dispositivos de baixo nível, apresentar uma abstração de hardware, possuir uma interface de troca de mensagens entre processos, gerenciamento de pacotes, entre outras. Além disso, o *ROS* tem a vantagem de ser *open source*, conta com pacotes e bibliotecas que tem por objetivo auxiliar no desenvolvimento de softwares para o controle de robôs, além de ferramentas de síntese, simulação e análise de dados.(THOMAS, 2014) De acordo com O’Kane (2013), as principais vantagens do uso do *ROS* são:

- Agilidade na execução de testes: A realização de experimentos e testes envolvendo robôs é uma tarefa complexa. Muitas vezes devido à indisponibilidade de um robô físico ou mesmo de algum componente de alto custo. Mas mesmo quando estes estão disponíveis, os testes tendem a ser trabalhosos e nem sempre acurados. O *ROS* facilita a realização de testes pois a modularização promovida pela plataforma facilita que as aplicações de alto nível sejam elaboradas em separado das de baixo nível. Assim sendo, o baixo nível pode ser facilmente substituído por simulações ou outros equipamentos. Além do mais, o *ROS* engloba um ferramenta de gravação de fluxo de dados, que podem ser medições de sensores, dados de localização, etc. De modo que uma mesma sequência de dados pode ser processada de diferentes maneiras e quantas vezes for preciso.
- Reuso de *software* e computação distribuída: As bibliotecas padrões do *ROS* fornecem uma variedade de algoritmos robustos para aplicações em robótica. Além disso, a interface de troca de mensagens do *ROS* tem se tornado um padrão de interoperabilidade para softwares de robótica, o que incentiva o reuso de técnicas e algoritmos já consolidados e facilita o desenvolvimento de novos experimentos. Essa estrutura modular que o *ROS* fornece também traz grandes vantagens quando há a necessidade de processar informações oriundas de diferentes plataformas. Desta forma, programas que utilizam a interface do *ROS* podem ser executados em computadores ou robôs interconectados. O que pode ser útil, por exemplo, em aplicações de robótica cooperativa.

O *Robot Operating System* é uma ferramenta que contribui para o avanço da pesquisa e desenvolvimento de software para robôs de diversas formas. Sua utilização requer o conhecimento dos conceitos fundamentais que constituem a interface.

### 2.2.1 Principais conceitos

Os conceitos que constituem a interface *ROS* podem ser subdivididos em três níveis distintos. São eles:

1. Sistema de arquivo

## 2. Grafos computacionais

### 3. Comunidade *ROS*

O primeiro nível consiste dos arquivos disponíveis em disco, tais como pacotes, arquivos de descrição de mensagens e serviços, entre outros. O segundo nível representa a rede *peer-to-peer* dos processos que ocorrem simultaneamente no *ROS*. Pertencem a este nível conceitos como ”nós“, ”*ROS* Mestre“, ”mensagens“, ”serviços“ e ”tópicos“. O terceiro e último nível diz respeito à distribuição e publicação de software livre e de conhecimento a partir da comunidade *ROS*. ([ROMERO, 2014](#))

Visando a melhor compreensão do uso do ambiente *ROS* neste trabalho, os conceitos mais relevantes serão apresentados a seguir:

- Pacotes (*packages*):

Os pacotes do *ROS* são conjuntos de arquivos relacionados, nos quais geralmente inclui os arquivos de execução (*nodes*) e demais componentes de um programa. A organização em pacotes visa condensar programas relacionados em um mesmo local. Um arquivo denominado ”manifesto“ define cada pacote. O manifesto é um arquivo que descreve os detalhes do pacote, tais como o nome do desenvolvedor, versão e suas dependências. ([O'KANE, 2013](#))

- Nós (*nodes*):

Os nós são instâncias executáveis que realizam processamento. De forma que cada nó pode ser dedicado a um determinado tipo de tarefa. Por exemplo, um nó pode realizar o processamento de dados de sensores do tipo *encoder* enquanto outro pode realizar uma computação de alto nível, como um planejamento de trajetórias. Diversas tarefas, complementares entre si ou não, podem ser executadas ao mesmo tempo por meio dos nós. A execução dos nós se dá por meio do comando *rosrun*.

- Mestre (*Master*):

A instância mestre do *ROS* é responsável pelo registro de nomes e por permitir a intercomunicação entre os nós. O *ROS Master* deve sempre estar ativo para utilizar o *ROS* e pode ser executado por meio do comando *roscore* em um terminal do linux.

- Mensagens (*messages*):

Mensagem é a forma adotada pelo *middleware* para realizar a comunicação de diferentes tipos de dados entre os nós. As mensagens são definidas por meio de arquivos de extensão *.msg*, que consistem em estruturas de dados simples. Algumas mensagens podem conter exclusivamente um determinado tipo primitivo como um numero inteiro, ponto flutuante, variável booleana, etc. No entanto, muitas das

mensagens contidas no *ROS* foram designadas para realizar a comunicação de dados comuns em equipamentos utilizados em robótica. Como por exemplo, as categorias *sensor\_msgs* e *nav\_msgs*, que consistem em um compilado de mensagens adequadas para transmissão de dados provenientes de sensores comuns e dados úteis para navegação, respectivamente.

- Tópicos (*topics*):

As mensagens são organizadas em tópicos nomeados. Logo, a comunicação acontece da seguinte maneira: Os nós que transmitem informação devem publicar mensagens em um determinado tópico. Enquanto um nó que necessita receber tal informação deverá subscrever ao tópico em questão. Constituindo a comunicação P2P entre nós.

- Serviços (*services*):

Apesar da arquitetura *publisher/subscriber* ser eficiente para grande parte das aplicações, existem casos em que a comunicação um para um é desejada. Nestes casos, a utilização de serviços é recomendada. Os serviços constituem uma forma de comunicação por meio de pedidos e respostas. Um nó pode oferecer serviços, identificados a partir de um nome, definidos por um par de estruturas de mensagem: uma para o pedido e outra para a resposta. (ROMERO, 2014) Os serviços se distinguem dos tópicos pois são acionados eventualmente por um nó ou pelo usuário e constituem uma alternativa de arquitetura de comunicação promovida pela interface *ROS*.

A figura 2 apresenta um exemplo de estrutura de software com base nos conceitos anteriormente explicados, por meio de um grafo.

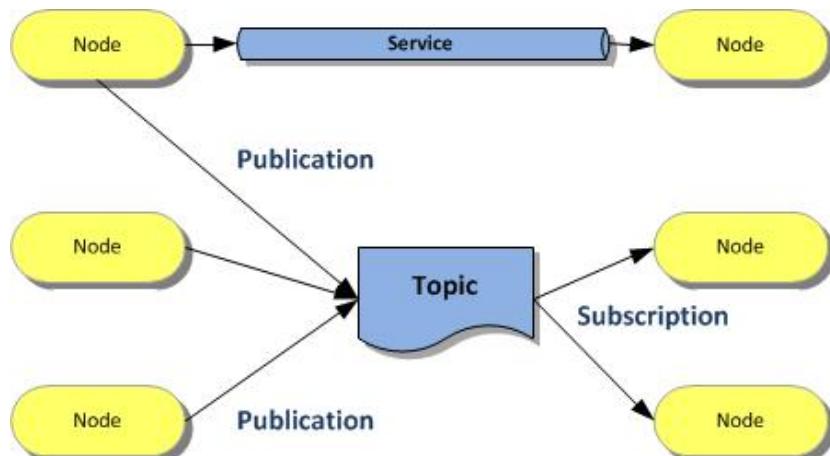


Figura 2 – Grafo com representação do uso de tópicos e serviços entre nós do *ROS*.

Fonte: (MAZZARI, 2016)

## 2.3 Visão Computacional

A visão computacional, segundo [Bradski e Kaehler \(2008\)](#), é definida da seguinte maneira:

Visão computacional é a transformação de dados de uma imagem estática ou de uma câmera de vídeo em uma decisão ou nova representação. De forma que estas transformações são realizadas com o intuito de atingir um determinado objetivo. Os dados de entrada podem incluir informações contextuais, tais como "a câmera está montada sobre um veículo" ou "o sensor indica um objeto há um metro de distância". E a decisão pode ser "há uma pessoa nesta imagem" ou "existem 14 células de tumor nesta imagem". Uma nova representação pode significar mudar a escala de cores em uma imagem para níveis de cinza ou remover o movimento da câmera em uma sequência de imagens. ([BRADSKI; KAEHLER, 2008](#), p. 2)

Em outras palavras, a visão computacional busca obter resultados e informações úteis a partir da captura, processamento e análise de imagens. Buscando, de certa forma, simular a visão humana. Na indústria, sistemas de visão computacional estão presentes em uma vasta gama de aplicações. A identificação de peças defeituosas, resíduos ou a mesma a estimativa de posição de um objeto em relação a um robô manipulador são exemplos comuns do uso deste tipo de equipamento. Como o campo de visão computacional é extenso e existem algoritmos consolidados para realizar as mais variadas operações, recomenda-se o reuso de software a partir de bibliotecas de visão computacional. Desta forma, evita-se a necessidade de reescrever códigos já existentes e otimizados para executar manipulações em imagens, tendo em vista a disponibilidade de ferramentas que aceleram o desenvolvimento de softwares para visão.

### 2.3.1 *OpenCV*

Uma das principais bibliotecas de visão computacional é a *OpenCV* ([ITSEEZ, 2015](#)) (*Open Source Computer Vision*), que inclui centenas de algoritmos e é constantemente atualizada. Além de possuir licença *BSD 3-Clause*, uma das licenças de código aberto mais permissivas, e que o uso da maioria das classes contidas no *OpenCV* até mesmo em aplicativos comerciais.

A API do *OpenCV*, a partir de sua versão 2.0, é principalmente escrita em C++. A biblioteca é modular, designada para eficiência computacional e com foco em aplicações em tempo real.

Os principais módulos disponibilizados pelo *OpenCV* são:

- *Core*: Módulo que define as estruturas básicas de dados, tal como o tipo de matriz multidimensional *cv::Mat*. Uma das principais estruturas para armazenar informações de uma imagem ou *frame* de um vídeo.

- *imgproc*: Consiste no módulo de tratativa e processamento de imagem. Oferece funções para aplicação de filtros lineares e não-lineares, transformações de imagem, conversão de níveis de cores, entre outras.
- *video*: Seção que inclui algoritmos de estimativa de movimento, subtração de fundo e rastreamento de objetos.
- *calib3d*: É principalmente composto por algoritmos de calibração para câmeras simples e estéreo, funções para estimativa de posição e elementos de reconstrução 3D.
- *highgui*: Módulo com interfaces gráficas simples e *codecs* de video.
- *features2D*: Fornece algoritmos para reconhecimento de características salientes, como quinas e bordas, além de descritores e detectores para comparação de características entre imagens ([ITSEEZ, 2014](#)).

Além dos módulos supracitados, o *OpenCV* ainda apresenta a possibilidade de integração com outras bibliotecas de visão computacional, e conta com um módulo de algoritmos experimentais.

Visando a aplicação de visão computacional para a localização de robôs móveis, este trabalho faz uso de algoritmos provenientes do módulo *calib3d*. Na seção [2.3.2](#) serão explicados os fundamentos teóricos para a realização de reconstrução 2D-3D, visando identificar a localização de objetos e *landmarks*.

### 2.3.2 Estimação de posição tridimensional

A reconstrução 3D para estimativa de posição apresenta grande importância em diversas aplicações de visão computacional, tais como navegação de robôs, aplicativos de realidade aumentada, detecção de translação e rotação de objetos, entre outras. Este processo se baseia em encontrar correspondências entre pontos no ambiente real e em sua projeção na imagem. Em outras palavras, a estimativa de posição visa associar dados de um sensor 2D (imagem de um objeto) com um modelo 3D (objeto real) ([ROSENHAHN CHRISTIAN PERWASS, 2004](#)). Uma representação desta técnica é apresentada na figura 3.

Adotando-se o modelo de câmera estenopeica <sup>1</sup>, ou câmera *pinhole*, a cena capturada é formada a partir da projeção dos pontos 3D no plano da imagem.

Representada pela transformação de perspectiva:

$$s \mathbf{m}' = \mathbf{A}[\mathbf{R}|\mathbf{t}]\mathbf{M}' \quad (2.1)$$

---

<sup>1</sup> Dispositivo de captura de imagem simples que apresenta forma de uma caixa fechada ou câmara. Em um de seus lados há um pequeno orifício o qual, pela propagação de luz retilínea, gera uma imagem do espaço exterior, no lado oposto da caixa.

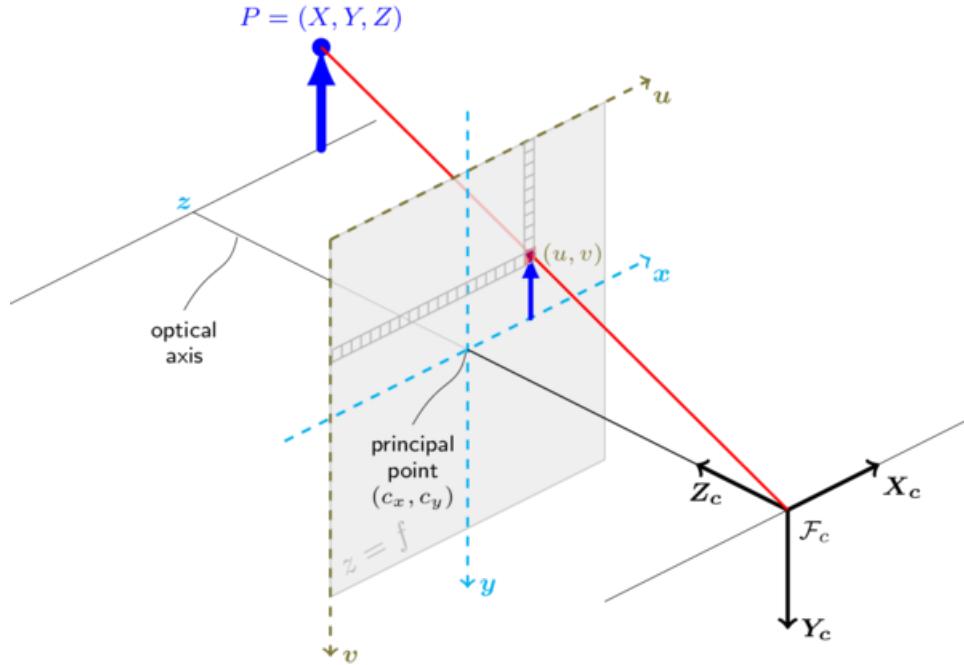


Figura 3 – Processo de estimativa de posição 2D-3D.

Fonte: (ITSEEZ, 2014)

que pode ser escrita como:

$$s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \quad (2.2)$$

sendo:

- $(X, Y, Z)$  a localização do ponto 3D nas coordenadas do ambiente.
- O vetor  $\mathbf{m}'$  ou  $(u, v, 1)^T$  representa as coordenadas dos pontos chave na imagem em pixels.
- $\mathbf{A}$  a matriz dos parâmetros intrínsecos da câmera.
- $(f_x, f_y)$  as distâncias focais da câmera, também expressas em pixel.
- $(c_x, c_y)$  é o ponto principal da imagem, geralmente localizado em seu centro.
- $s$  o fator de escala da imagem.
- $[\mathbf{R}|\mathbf{t}]$  é a matriz de transformação homogênea ou matriz dos parâmetros extrínsecos.

A matriz de parâmetros intrínsecos  $\mathbf{A}$  é característica do sensor de visão utilizado e, portanto, a cena capturada não influencia em seus valores. Portanto, uma vez que os valores de  $f_x$ ,  $f_y$ ,  $c_x$  e  $c_y$  são estimados e fixos, a matriz  $\mathbf{A}$  é conhecida e constante (ITSEEZ, 2014). Assim sendo, para descrever o movimento de um objeto ao redor de uma câmera parada ou, de forma similar, descrever o movimento da câmera em relação a um objeto estático, é necessário calcular a matriz de parâmetros extrínsecos  $[\mathbf{R}|\mathbf{t}]$  (matriz de rotação  $\mathbf{R}$  e vetor de translação  $\mathbf{t}$ ).

A transformação 2.2 é equivalente à equação a seguir (para  $z \neq 0$ ):

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \mathbf{R} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} + \mathbf{t} \quad (2.3)$$

Adotando-se:

$$x' = x/z$$

$$y' = y/z$$

São obtidas as seguintes equações de transformação:

$$u = f_x x' + c_x \quad (2.4)$$

$$v = f_y y' + c_y \quad (2.5)$$

Levando-se em consideração que câmeras convencionais apresentam distorções na imagem capturada, comumente representadas pelo modelo de Brown–Conrad (C.BROWN, 1966) para distorções radiais e tangenciais, torna-se preciso estender o modelo acima, com o objetivo de compensar os efeitos destes dois tipos de distorções. A distorção causa um efeito que dá um aspecto curvado às linhas que deveriam ser retas em uma projeção. A figura 4 mostra efeitos comuns dos coeficientes de distorção na imagem.

Acrescentando-se os coeficientes de distorção radial  $k_1, k_2, k_3, k_4, k_5$  e  $k_6$ , e também os coeficientes de distorção tangencial  $p_1$  e  $p_2$ , tem-se:

$$x'' = x' \frac{1 + k_1 r^2 + k_2 r^4 + k_3 r^6}{1 + k_4 r^2 + k_5 r^4 + k_6 r^6} + 2p_1 x' y' + p_2 (r^2 + 2x'^2) \quad (2.6)$$

$$y'' = y' \frac{1 + k_1 r^2 + k_2 r^4 + k_3 r^6}{1 + k_4 r^2 + k_5 r^4 + k_6 r^6} + p_1 (r^2 + 2y'^2) + 2p_2 x' y' \quad (2.7)$$

sendo  $r^2 = x'^2 + y'^2$

Substituindo  $x'$  e  $y'$  por  $x''$  e  $y''$  nas equações 2.4 e 2.5, respectivamente, são obtidas a relação entre coordenadas projetadas e reais:

$$u = f_x x'' + c_x \quad (2.8)$$

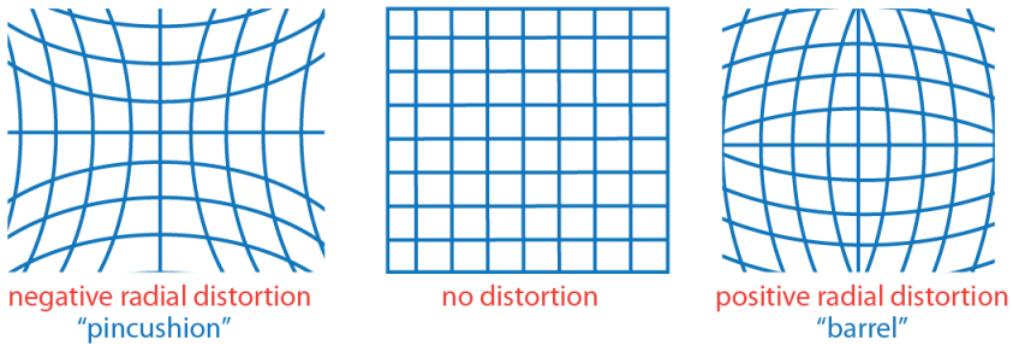


Figura 4 – Efeitos da distorção na imagem. (Esquerda) Efeito de distorção conhecido por “almofada de alfinetes”. (Direita) Distorção do tipo barril.

Fonte: ([MATHWORKS, 2016](#))

$$v = f_y \ y'' + c_y \quad (2.9)$$

A partir das funções do módulo *calib3d* do *OpenCV* é possível determinar os parâmetros intrínsecos da câmera por meio do processo de calibração. Com isso, a biblioteca fornece funções para projetar pontos 3D no plano da imagem, calcular os vetores de rotação e translação com base em pontos 3D e suas projeções e também funções específicas para o uso em câmeras estéreo (com mais de uma lente).

## 2.4 Localização

O sistema de navegação de um robô móvel pode ou não depender do conhecimento de sua localização no ambiente. Podendo, em alguns casos, o robô navegar por meio de comportamentos reativos. Isto é, a navegação do robô por meio de ações predefinidas ativadas de acordo com informações sensoriais locais. Sistemáticas de comportamento reativo, como a arquitetura de subsunção ([BROOKS, 1986](#)) ou arquitetura baseada em campos potenciais ([BALAKRISHNAN; NEVZOROV, 2004](#)) não requerem de fato um sistema de localização com representação do ambiente. No entanto, em aplicações onde o robô opera em ambientes mais complexos ou extensos, geralmente são utilizados representações do ambiente, ou mapas ([ROMERO et al., 2014](#)).

O desafio de determinar a localização de um robô móvel sobre um plano consiste em estabelecer, com uma certa precisão, onde o robô móvel está e para qual direção ele está orientado. Levando-se em consideração uma representação bidimensional, a localização do robô no ambiente é especificada por meio de uma relação entre o *frame* de referência global do ambiente  $\{X_I, Y_I\}$  e o *frame* local do robô  $\{X_R, Y_R\}$ . Ambos evidenciados na figura 5.

O primeiro define a base inercial sobre o plano onde o robô se movimenta, e tem

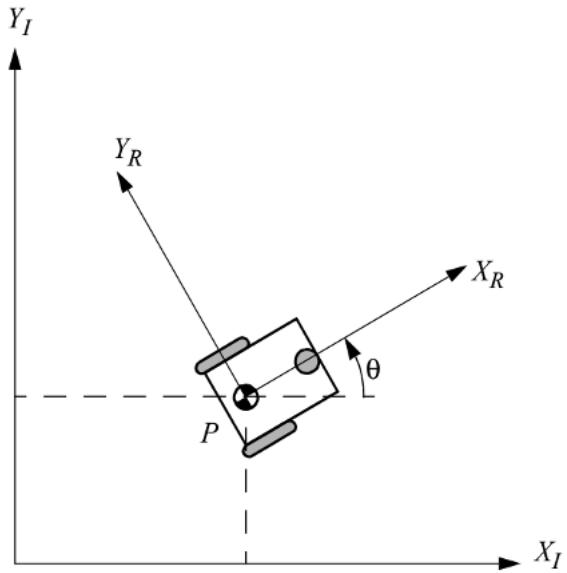


Figura 5 – *Frames* de referência global e local.

Fonte: ([SIEGWART; NOURBAKHSH, 2004](#)).

uma origem fixa  $O : \{X_I, Y_I\}$ . Já a base  $\{X_R, Y_R\}$  define dois eixos fixos a um ponto  $P$  arbitrário, alocado no chassi do robô. Deste modo, a posição (x,y) do robô no ambiente se dá pela posição do ponto P em relação aos eixos de referência globais. Já sua orientação  $\theta$  é calculada pela diferença angular entre os *frames* de referência global e local ([SIEGWART; NOURBAKHSH, 2004](#)). Assim, tem-se o vetor que representa a localização do robô no ambiente:

$$\xi_I = \begin{bmatrix} X \\ Y \\ \theta \end{bmatrix} \quad (2.10)$$

Existe uma variedade de técnicas para a determinação da localização de um robô móvel. Para determinar uma sistemática eficaz, é necessário conhecer o tipo de ambiente e complexidade do robô. Certas técnicas podem ser adequada para um ambiente desconhecido, enquanto outras requerem alguma conhecimento prévio sobre local para que o robô possa navegar com sucesso. Um método para a obtenção de uma estimativa de posição relativa é a odometria, também denominada reconhecimento passivo ou *dead reckoning* quando o robô utiliza unicamente este método para determinar onde está.

#### 2.4.1 Localização por *dead reckoning*

*Dead reckoning*, em robótica móvel, é o processo de determinar a posição atual do robô com base na projeção de seu curso e velocidade, a partir de posições anteriores

conhecidas. É uma técnica simples, onde o deslocamento do robô móvel é obtido por meio de sensores internos (proprioceptivos), tais como os *encoders* incrementais ou absolutos (figura 6). Por fim, relaciona-se a informação de deslocamento dos sensores com um modelo cinemático do robô para obter as coordenadas do robô no ambiente e sua orientação.

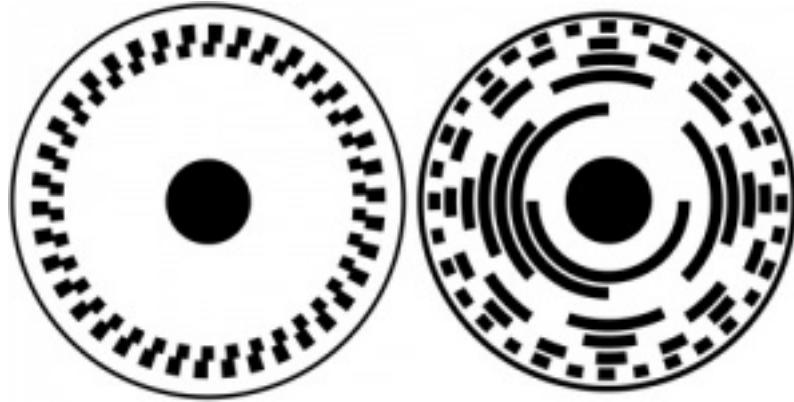


Figura 6 – Representação de um *encoder* incremental(esquerda) e absoluto (direita).

Fonte: ([PHIDGETS, 2014](#))

O reconhecimento passivo é uma técnica eficaz para a estimativa de posição para curtas distâncias. No entanto este método está sempre sujeito a erros devido ao deslizamento das rodas, limite de resolução dos *encoders*, irregularidade do piso, entre outros fatores. Isto significa que o *dead reckoning* é uma técnica de estimativa de posição relativa suscetível a erros incrementais. Por outro lado, com o uso de sensores externos (ou exteroceptivos) é possível adquirir dados de localização absolutos como sensores de visão, sensor *laser rangefinder*, entre outros. Pode-se, portanto combinar o *dead reckoning*, que é uma estimativa relativa, com técnicas baseadas em sensores externos, que fornecem uma estimativa de localização absoluta. Desse modo as vantagens dos dois tipos de sensores são utilizadas para criar uma crença de posição e orientação robusta ([SEKIMOR; MIYAZAKI, 2007](#)).

#### 2.4.2 *Landmarks*

*Landmarks* são características presentes no ambiente que podem ser utilizadas por um robô móvel para a obtenção de informações sobre a sua localização. Estes pontos de referência podem ser marcos naturais (características próprias do ambiente a serem extraídas e reconhecidas) ou artificiais (marcos estrategicamente adicionados ao ambiente). De acordo com [Riisgaard e Blas \(2004\)](#), as *landmarks* que serão reconhecidas pelo robô móvel devem seguir as seguintes recomendações para serem efetivas:

- *Landmarks* devem ser facilmente reobserváveis.

- Cada marco individual deve ser distingível dos os outros.
- O ambiente deve possuir uma grande quantidade de *landmarks*, de forma que o robô não passe muito tempo embasando-se somente em sua odometria.
- *Landmarks* devem ser sempre estacionárias.

A seleção de *landmarks* para o uso de sistemas de navegação de robôs móveis é importante para a construção de uma representação do ambiente. No entanto, *landmarks* podem se utilizadas para descrever uma determinada rota ([HALLAM; FLOREANO; MEYER, 2002](#)) ou para ajustar o sistema de localização de um robô móvel.

## 2.5 Mapeamento

Um mapa é uma representação de um ambiente, que contém informações sobre elementos presentes em um local, com algum nível de abstração. Um robô pode navegar e se localizar com auxílio de um mapa para aprimorar a precisão de sua localização, pois as características detectadas por sensores externos podem ser relacionadas a elementos presentes na representação do ambiente e, por fim, esta informação pode ser utilizada como uma realimentação do estado do robô. Segundo [Romero et al. \(2014\)](#), diferentes tipos de mapas foram propostos na literatura. Podendo ser classificados em dois principais tipos, de acordo com o grau de abstração e finalidade. São eles os mapas métricos e mapas topológicos.

### 2.5.1 Mapas métricos

Os mapas métricos possuem menor nível de abstração, consistem em representações claramente próximas às características um ambiente. Uma técnica muito difundida de mapeamento métrico são as grades de ocupação (*grid maps*). Este tipo de mapa divide o espaço em células menores e regulares. Cada célula possui um estado de ocupação (vazio, ocupado ou inexplorado), que limita a região de navegação do robô. Sendo assim, os espaços contínuos do ambiente são discretizados, permitindo que toda a planta seja representada por uma matriz multidimensional. A figura 7 mostra um exemplo de grade de ocupação.

Em resumo, os mapas métricos apresentam as vantagens de serem facilmente construídos, representados e mantidos (mesmo em ambientes extensos). Além disso, promovem uma facilidade de reconhecimento de locais diferentes, baseando-se na posição geométrica do robô em relação a um sistema de coordenadas global, e facilitam a computação de caminhos curtos.

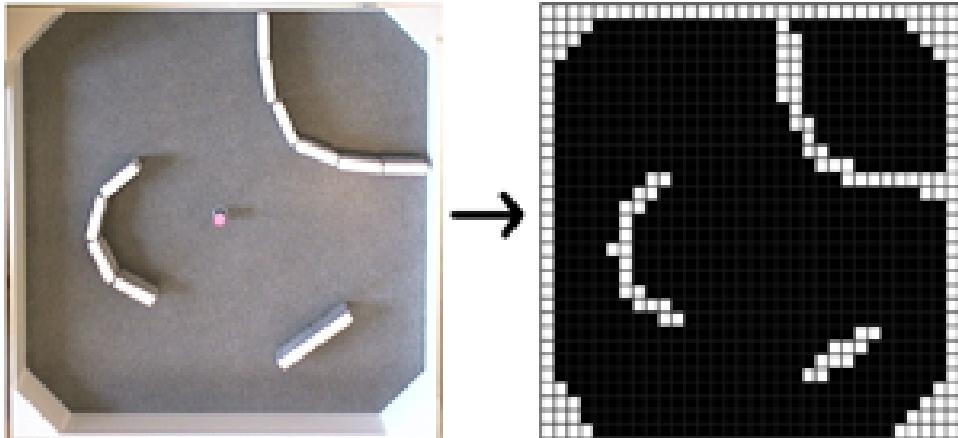


Figura 7 – Exemplo de ambiente discretizado em um mapa métrico de grades de ocupação.

Fonte: ([BAATH, 2008](#)).

Por outro lado, os mapas métricos requerem um alto custo computacional e, consequentemente, alta capacidade de processamento e memória do robô. Visto que sua representação do ambiente geralmente contém mais detalhes do que na abordagem topológica. Adicionalmente, durante o processo de construção deste tipo de mapa, a informação da localização do robô precisa ser exata.

### 2.5.2 Mapas topológicos

Mapas topológicos, por sua vez, são representados a partir de grafos que contém nós interconectados por arestas. Assim como mostra a figura 8.

Geralmente, os grafos representam os espaços livres para a circulação do robô, os nós definem as regiões nas quais existem informações sensoriais homogêneas e os arcos definem a interconexão entre os nós. A exata localização do robô não é representada neste tipo de mapa. Em vez disso, é possível obter uma informação abstrata sobre a região em que o robô se encontra. Em compensação, a estrutura de grafo reflete uma metodologia de mapeamento com menor custo computacional, e por isso pode ser utilizada em robôs móveis com menor capacidade de processamento. Além disso, os mapas topológicos são uma representação mais adequada para a resolução de problemas de alto nível.

## 2.6 SLAM

O conceito de mapeamento e localização simultâneo foi desenvolvido por Hugh Durrant-Whyte e John J.Leonard, e tem por objetivo dotar um robô móvel da capacidade de construir mapas de ambientes desconhecidos enquanto navega por este mesmo ambiente. *SLAM - Simultaneous Localization and Mapping* consiste em uma frente de pesquisa de robótica móvel abrangente, que engloba técnicas de localização referenciadas a um mapa

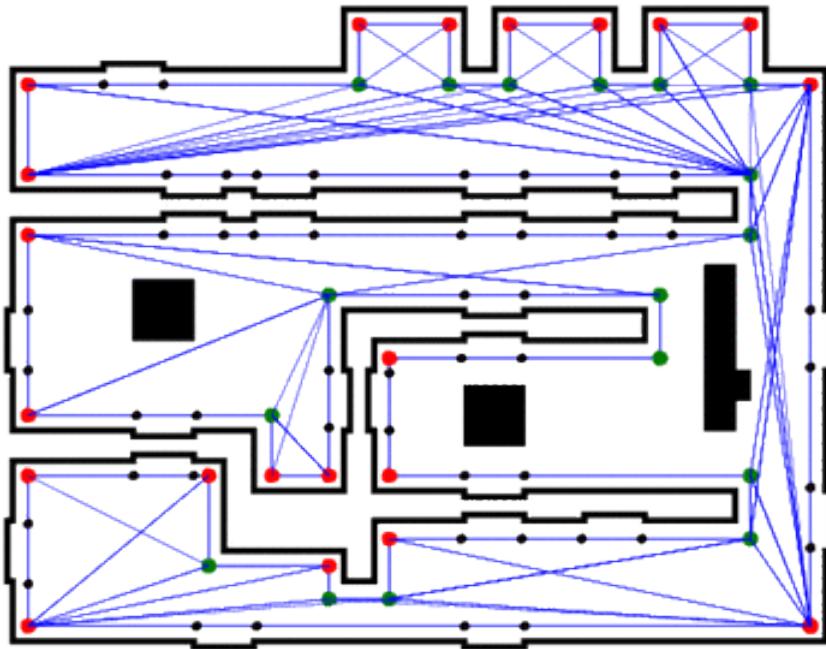


Figura 8 – Representação topológica de um ambiente.

Fonte: ([BEEVERS, 2004](#)).

do ambiente. O *SLAM* apresenta o chamado “problema do ovo e da galinha”, tendo em vista que a fim de se obter informações sobre a localização do robô em um determinado instante, é necessário que o módulo de localização se baseie em um mapa. Da mesma forma, para construir um mapa (2D ou 3D) confiável, é necessário que as informações de localização sejam adequadas. Logo, ambos os processos ocorrem simultaneamente e apresentam elevada interdependência ([ROMERO et al., 2014](#)).

Segundo Riisgaard e Blas ([2004](#)), a implementação do *SLAM* é composta por algumas etapas. São elas:

1. Extração de *landmarks*.
2. Associação de dados.
3. Estimação de estados.
4. Atualização de estados.
5. Atualização de *landmarks*.

O objetivo do *SLAM* é fazer uso do ambiente para corrigir a posição do robô. A partir de sensores externos do tipo *laser rangefinder*, sonar, ou mesmo sensores de visão (câmeras) estéreo, é possível extrair características de um determinado local e verificar a

movimentação do robô móvel. Geralmente utilizam-se filtros probabilísticos para se obter uma crença de localização mais acurada e monitorar a incerteza das medições.

A metodologia de *SLAM* 2D apresentada em Grisetti, Stachniss e Burgard (2007) opta pelo uso de filtros de partículas de Rao-Blackwell. A ideia principal deste filtro é estimar a probabilidade posterior  $p(x_{1:t}, m|z_{1:t}, u_{1:t-1})$ , sobre o mapa  $m$  e a trajetória  $x_{1:t}$ . Esta estimativa é feita com base nas observações  $z_{1:t}$ , que corresponde às observações (medidas de sensores externos) e as medições de odometria  $u_{1:t-1}$ . Tal função de probabilidade posterior é dada pela equação 2.11.

$$p(x_{1:t}, m|z_{1:t}, u_{1:t-1}) = p(m|x_{1:t}, z_{1:t}) \cdot p(x_{1:t}|z_{1:t}, u_{1:t-1}) \quad (2.11)$$

Esta fatoração permite estimar primeiramente apenas a trajetória do robô. E com isso, construir o mapa.

O procedimento completo desta metodologia foi condensado em um pacote do *ROS* chamado *gmapping* (SCHWEIGERT, 2015) e sua implementação é detalhada na seção 3.3.

# 3 Metodologia

A metodologia definida neste capítulo tem por objetivo a implementação e análise comparativa de três técnicas de localização de um robô móvel em ambiente interno. São elas:

1. *Dead reckoning*/Reconhecimento passivo.
2. Metodologia por visão computacional com auxílio de *landmarks* artificiais.
3. *SLAM* - Mapeamento e localização simultâneos a partir de sensor *laser rangefinder*.

Neste capítulo será explorado o modelo de robô móvel utilizado, tal como as especificidades do ambiente de operação do mesmo. Além disso, o desenvolvimento de cada técnica aplicada será apresentado. Por fim, serão especificados os experimentos realizados para a obtenção dos resultados.

## 3.1 Elaboração do ambiente de simulação

O projeto foi desenvolvido com auxílio do software de simulação para robótica *V-REP* da *Coppelia Robots*. O ambiente criado para a realização das simulações é cercado e sem obstáculos móveis. O qual tem por objetivo representar um ambiente interno convencional, em que seja possível coletar dados com pouca ou nenhuma interferência arbitrária. O local consiste em uma planta de dimensões 5x10m, subdividida em dois aposentos. O ambiente, portanto, apresenta um formato de “L”, tal como mostram as figuras 9 e 10. O *frame* de referência global do ambiente foi alocado em seu centro ( $X_0 = 0, Y_0 = 0$ ).

### 3.1.1 *Landmarks* artificiais

Para a aplicação da metodologia de localização por visão computacional, utilizou-se marcadores fiduciais<sup>1</sup>, gerados a partir da biblioteca *ArUco* (GARRIDO-JURADO et al., 2014), que atualmente é parte do módulo de algoritmos experimentais do *OpenCV*. É comum o uso de marcadores destas etiquetas em aplicações como jogos eletrônicos ou animações que utilizam realidade aumentada. Tendo em vista que são recursos de baixo custo e que podem ser usados para identificar a posição relativa de um ponto no ambiente em relação à câmera, detectar a orientação de um aparelho celular ou definir um plano de

---

<sup>1</sup> Marcos alocados no campo de visão da câmera a fim de serem utilizados como ponto de referência ou de medição.

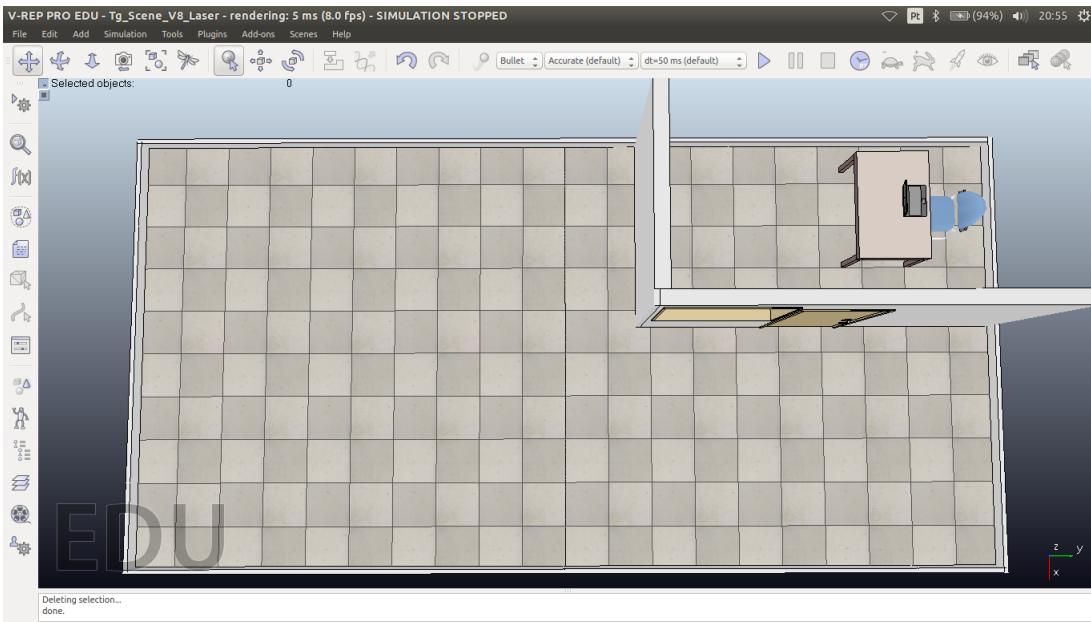


Figura 9 – Ambiente de operação do robô móvel, elaborado no simulador V-REP.

Fonte: Produzido pelo autor.

referência para personagens e objetos virtuais. A figura 11 apresenta um exemplo de sua utilização.

Este tipo de marcador é bastante parecido com etiquetas de código de barras 2D ou *QR-Codes*. Entretanto, a única informação que os marcadores de realidade aumentada

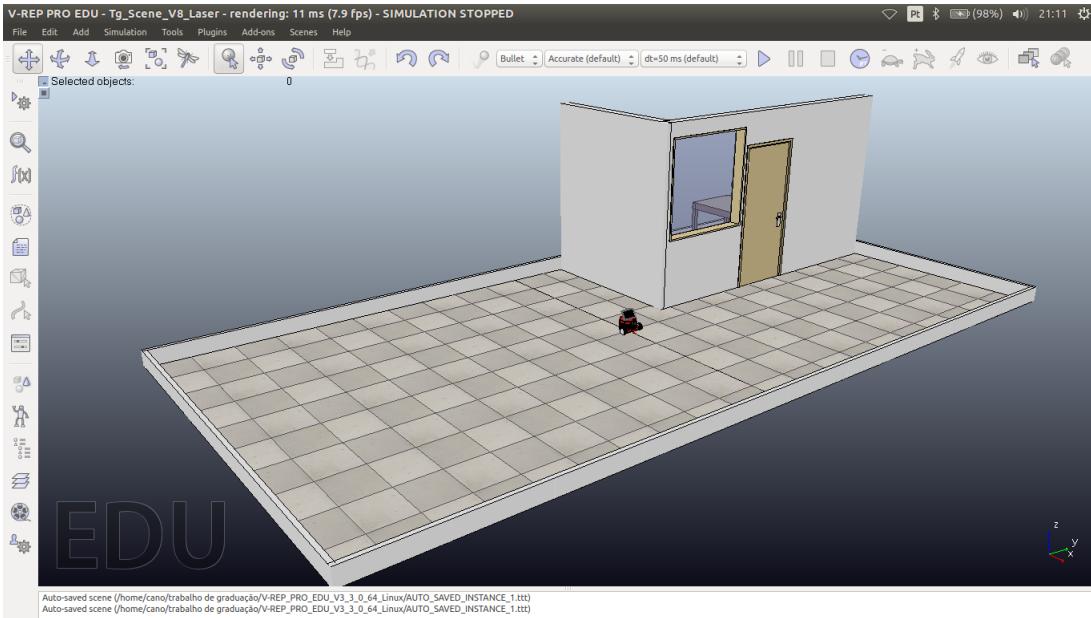


Figura 10 – Ambiente de operação do robô móvel, elaborado no simulador V-REP.

Fonte: Produzido pelo autor.

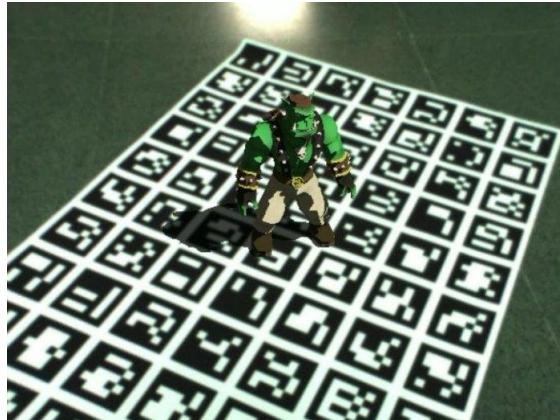


Figura 11 – Aplicativo *Ogre*, que utiliza realidade aumentada.

Fonte: ([GARRIDO-JURADO et al., 2014](#)).

carregam é um índice numérico, codificado em uma imagem. O índice permite que um marcador seja distingível dos demais. A biblioteca contém funções para a extração da informação sobre o índice de cada marcador, que consistem em detectar, subdividir a imagem em um *grid* de informações binárias (figura 12) e, em seguida, identificar a qual número decimal a combinação no marcador corresponde.

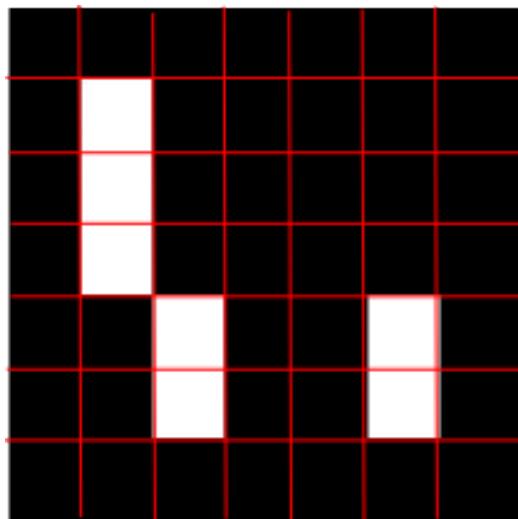


Figura 12 – Marcadores para realidade aumentada.

Fonte: ([GARRIDO-JURADO et al., 2014](#)).

Foram gerados 42 marcadores de realidade aumentada distintos, que foram distribuídos uniformemente no local. A princípio, adotou-se uma distância de um metro entre cada marcador, como mostra a figura 13. Com isso, o ambiente virtual de operação do robô contém uma “matriz” de *landmarks*, artificiais no piso. Tornando o local adequado para a aplicação dos algoritmos de visão computacional para a localização. Apresentados na seção de experimentos (3.3).

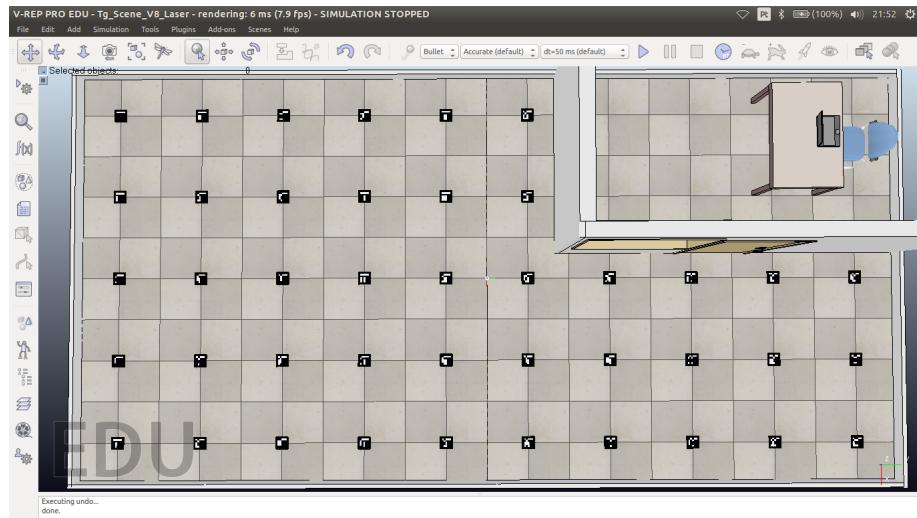


Figura 13 – Ambiente de operação com marcadores de realidade aumentada posicionados com distância de um metro entre eles.

Fonte: Produzido pelo autor.

### 3.2 Robô móvel utilizado

O robô móvel utilizado, *dr20* presente na biblioteca de modelos do *V-REP*, consiste em um pequeno veículo dotado de rodas. O arranjo das rodas é composto por duas rodas alinhadas que exercem tração e um terceiro ponto de apoio (figura 14).

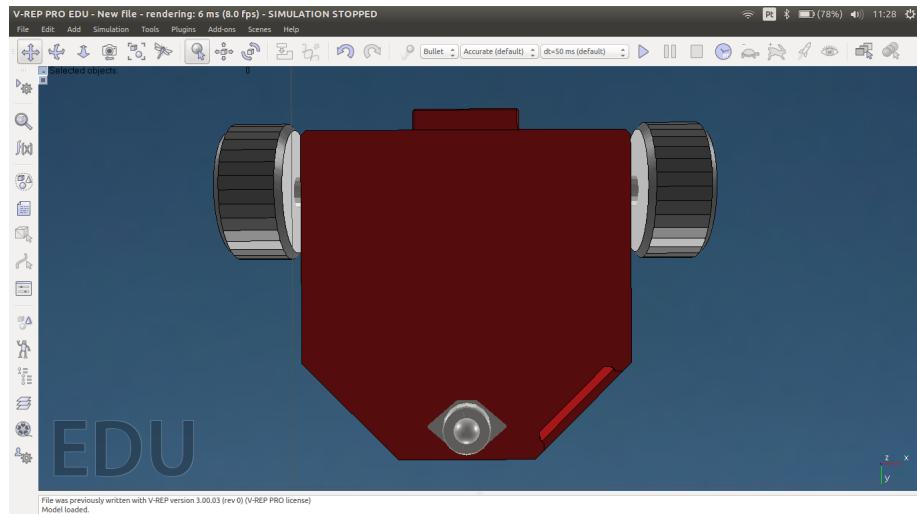


Figura 14 – Base do robô *dr20*.

Fonte: Produzido pelo autor.

Algumas modificações foram feitas ao modelo do *dr20*. O sensor de visão do robô, com resolução de 256x256 px foi alocado sobre o robô e inclinado em 40° para baixo (*pitch*), de modo que este capture a imagem das *landmarks* no piso. Adicionalmente um sensor do

tipo *laser rangefinder* do modelo Hokuyo URG-04LX-UG01 (figura 15) foi adicionado à parte frontal do robô.

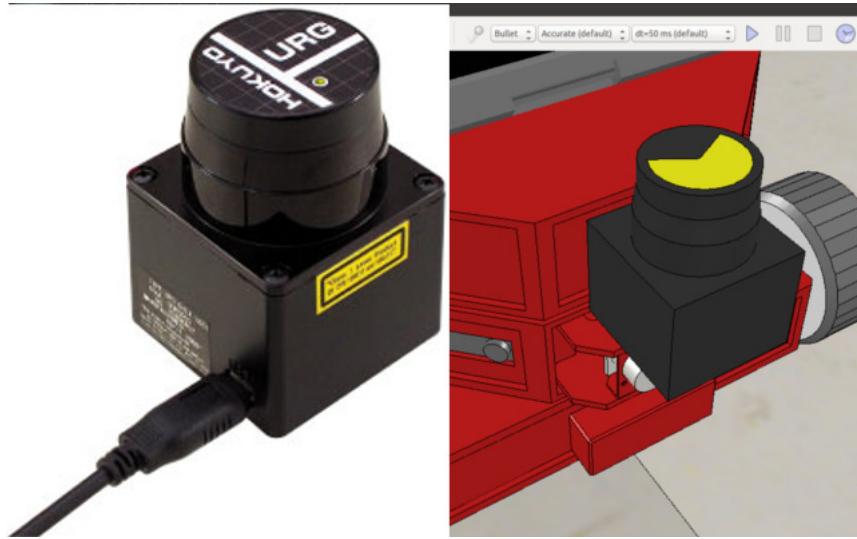


Figura 15 – (Esquerda) Sensor *laser rangefinder* real. (Direita) Sensor equivalente no ambiente *V-REP*.

Fonte: (Esquerda) ([HOKUYO, 2009](#)). (Direita) Produzido pelo autor.

### 3.2.1 Modelo cinemático direto

O *dr20* é um robô móvel de acionamento diferencial (figura 16). Esta configuração representa um veículo que executa curvas a partir da diferença de velocidades entre suas duas rodas, que podem exercer tração em dois sentidos de rotação. Considerou-se, portanto, um modelo de robô diferencial supondo velocidades angulares aplicadas diretamente às rodas.

Considerou-se o chassi do robô como um corpo rígido, com simetria em relação a seu eixo longitudinal e centro de giro sobre este mesmo eixo. O modelo cinemático direto deste robô é definido por:

$$\dot{\xi} = \mathbf{J}(\mathbf{q}, \lambda, \theta) \cdot \dot{\mathbf{q}} \quad (3.1)$$

Nesta equação 3.1,  $\xi$  representa o vetor de *pose* do robô em relação ao *frame* de referência global,  $\mathbf{J}$  a matriz jacobiana do modelo cinemático direto,  $\mathbf{q}$  o vetor de posição angular das juntas e  $\lambda$  o vetor de parâmetros geométricos das rodas e chassi do robô, definido por:

$$\lambda = \begin{pmatrix} r \\ b \end{pmatrix} \quad (3.2)$$

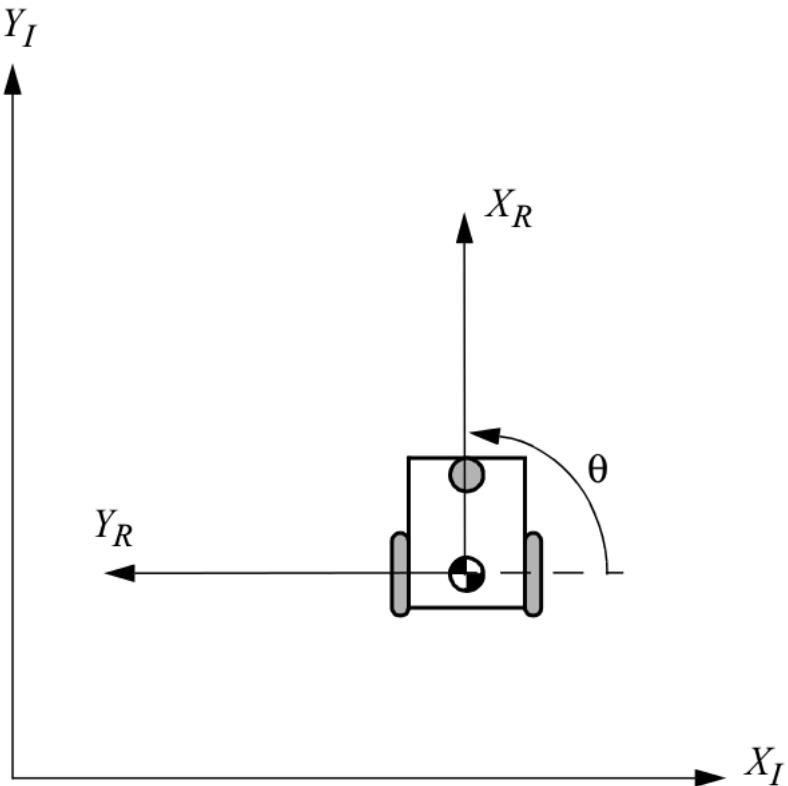


Figura 16 – Representação de veículo com acionamento diferencial.

Fonte: ([SIEGWART; NOURBAKHSH, 2004](#)).

Em que  $r$  corresponde ao valor do raio das rodas, e  $b$  à distância de separação entre as rodas do robô (assumindo que a origem *frame* de referência local  $O : \{X_R, Y_R\}$  encontra-se localizado no centro do chassi e entre as duas rodas do robô). Deste modo, a matriz jacobiana para esta configuração de acionamento é dada por:

$$\mathbf{J} = \begin{pmatrix} \frac{r}{2} \cos \theta & \frac{r}{2} \cos \theta \\ \frac{r}{2} \sin \theta & \frac{r}{2} \sin \theta \\ \frac{r}{b} & -\frac{r}{b} \end{pmatrix} \quad (3.3)$$

E, por fim, o vetor de deslocamento angular das juntas é descrito da seguinte forma:

$$\mathbf{q} = \begin{pmatrix} \phi_d \\ \phi_e \end{pmatrix} \quad (3.4)$$

Assumindo que as variáveis  $\phi_d$  e  $\phi_e$  são mensuráveis a cada instante, e sabendo os valores dos parâmetros geométricos  $reb$ , é possível obter as três equações para localização do robô por odometria a partir da integração deste modelo cinemático:

$$x(t) = x_0 + \frac{b(\dot{\phi}_d + \dot{\phi}_e)}{2(\dot{\phi}_d - \dot{\phi}_e)} \left[ \sin\left(\frac{r(\dot{\phi}_d - \dot{\phi}_e)t}{b} + \theta_0\right) - \sin\theta_0 \right] \quad (3.5)$$

$$y(t) = y_0 + \frac{b(\dot{\phi}_d + \dot{\phi}_e)}{2(\dot{\phi}_d - \dot{\phi}_e)} \left[ \cos\left(\frac{r(\dot{\phi}_d - \dot{\phi}_e)t}{b} + \theta_0\right) - \sin\theta_0 \right] \quad (3.6)$$

$$\theta(t) = \frac{rt(\phi_d - \phi_e)}{b} + \theta_0 \quad (3.7)$$

Portanto, a estimativa de localização relativa por meio de sensores internos (*dead reckoning*) pode ser obtida por meio das equações 3.5, 3.6 e 3.7. Alternativamente, uma simplificação deste cálculo pode ser aplicado para a determinação da odometria, como exposto em [Borenstein, Everett e Feng \(1996\)](#) e [Lucas \(2000\)](#):

$$x = x_0 + \bar{s} \cos(\theta) \quad (3.8)$$

$$y = y_0 + \bar{s} \sin(\theta) \quad (3.9)$$

$$\theta = \theta_0 + \frac{r(\phi_d - \phi_e)}{2b} \quad (3.10)$$

Sendo:

$$\bar{s} = \frac{r(\phi_d + \phi_e)}{2}$$

Esta abordagem visa reduzir o custo computacional do algoritmo e o erro cumulativo de localização, simplesmente dividindo por dois o valor de cada incremento de  $\theta$  ao longo do tempo. As equações 3.8, 3.9 e 3.10 foram utilizadas nas três técnicas de localização abordadas neste trabalho.

### 3.3 Experimento desenvolvido

Nesta seção será detalhada a metodologia aplicada para a aquisição de resultados em cada uma das três metodologias propostas. Todas as metodologias foram programadas essencialmente em C++, com auxílio de *scripts* LUA para a transmissão e subscrição de mensagens referentes aos sensores do robô e posições calculadas dentro dos *ROS nodes*. Adotou-se a frequência de 20 Hz (padrão de simulação do *V-REP*) para transmissão de mensagens em todos os nós do ROS.

O experimento realizado neste trabalho consiste na obtenção de dados sobre a eficácia de cada uma das três metodologias. Para isso, as três metodologias foram

desenvolvidas externamente ao *V-REP* (em pacotes e nós do ROS). Então, os vetores de localização  $\xi = (X, Y, \theta)$  obtidos em cada caso foram publicados e comparados em uma mesma trajetória, executada novamente dentro do ambiente do *V-REP*.

### 3.3.1 *Odometria pura* (dead reckoning)

Para calcular a odometria do *dr20* foi preciso obter a leitura das velocidades angulares de suas juntas rotacionais. Para isso, foi criado um *child script (non threaded)*, associado ao *dr20*, para transmitir o estado das juntas. Dentro deste *script*, utilizou-se a função *JointPublisher* de modo que o *V-REP* possa publicar a matriz de estados de todas as juntas existentes na simulação:

```
1 JointPublisher = simExtROS_enablePublisher("/JointState", 1,
    simros_strmcmd_get_joint_state, simHandle_all, 0, "")
```

Em seguida, criou-se um nó denominado “odometria\_simples\_node”. Este programa é responsável por subscrever ao tópico “/JointState”, aquisitar as velocidades angulares das juntas, calcular a odometria a partir da posição inicial do robô e, por fim, publicar o vetor de localização. Utilizou-se uma função de *callback*, para realizar o processo de subscrição e publicação das mensagens. Uma seção do código deste programa é apresentado a seguir:

```
1 void Odom::jointCallback(const sensor_msgs::JointState::ConstPtr&
    msg)
2 {
3     /*Calculo da odometria*/
4     double lVel, rVel, s, b = 0.25408, sl, sr;
5         lVel = msg->velocity[2];
6         rVel = msg->velocity[3];
7
8         sr = 0.0425*rVel*0.05;
9         sl = 0.0425*lVel*0.05;
10        s = ((sr + sl))/2;
11        v = s/0.05;
12        theta = ((sr - sl))/b+ theta;
13        x = x + s*cos(theta);
14        y = y + s*sin(theta);
15
16        ...
17
18        PosePub.publish(pstamp); //Publica posicao
19 }
```

A função “Odom::jointCallback“ transforma as velocidades angulares das juntas do robô em uma estimativa do vetor de  $\text{pose } \xi$ , que é novamente enviada ao *V-REP* por meio de uma mensagem do tipo *geometry\_msgs::PoseStamped*. Para que, posteriormente, seja feita a comparação das crenças de localização do robô com base nas três sistemáticas. Logo, a primeira metodologia, *dead reckoning*, foi implementada. Este processo pode ser representado por meio do grafo:



Figura 17 – Representação do sistema de odometria por meio de grafos do *ROS*.

Fonte: Produzido pelo autor

### 3.3.2 Metodologia por visão computacional

Esta segunda metodologia consiste na complementação da odometria por meio de algoritmos de reconstrução 2D-3D e marcos *ArUco*. A ideia fundamental deste método é corrigir o sistema de odometria de forma arbitrária. Cada vez que uma etiqueta *ArUco* for detectada pela câmera do robô, um novo cálculo da localização absoluta do robô é realizado. E as variáveis  $x$ ,  $y$  e  $\theta$  das equações de odometria são atualizadas diretamente com esta nova medição. Desta forma, busca-se obter um processo simples e de baixo custo para o sistema de localização do robô móvel.

Para criar este sistema, o primeiro passo consiste em ter acesso à imagem da câmera do robô. No ambiente *V-REP*, um *script LUA* foi desenvolvido para publicar a imagem capturada pelo sensor de visão do *dr20* para os programas externos. A função de publicação desta informação é mostrada abaixo:

```

1 CameraHandler=simExtROS_enablePublisher('visionSensorData',1,
                                         simros_strmcmd_get_vision_sensor_image,visionSensorHandle,0,'')

```

Com isto, a imagem amostrada pelo robô se torna acessível para a leitura em um programa em c++ externo, permitindo que a biblioteca do *OpenCV* seja utilizada para processá-la. O nó do *ROS* utilizado para o processamento de imagens é denominado *cv\_node*. Este nó é responsável por realizar as seguintes tarefas:

1. Receber a mensagem publicada no tópico *vision sensor data*.

2. Analisar cada quadro de imagem do sensor de visão do *dr20* para verificar se há marcadores de realidade aumentada na imagem capturada.
3. Calcular os vetores de rotação e translação, denominados *rVec* e *tVec*, respectivamente. Que fornecem informação sobre a posição e orientação de um marcador em relação ao *frame* de referência da câmera.
4. Calcular a localização da câmera em relação ao *frame* do marcador detectado, a partir de uma transformação linear.
5. Calcular a localização global do robô, associando as coordenadas globais do marcador por meio de uma look-up-table.
6. Publicar a posição global e para correção da odometria, calculada em outro nó.

O recebimento de mensagens por meio do *ROS* é dado por uma função de *callback*. Logo, foi desenvolvida a função *cvProcess::imageCallback* para o recebimento da imagem. Como resumido no código a seguir:

```

1 void cvProcess::imageCallback(const sensor_msgs::ImageConstPtr&
2   msg)
3 {
4   ...
5   cv_bridge::CvImagePtr cv_ptr;
6   cv_ptr = cv_bridge::toCvCopy(msg);
7   /*cv_ptr->image agora contém a imagem a ser analisada no
8    formato cv::Mat, matriz multidimensional padrão do opencv
9    para armazenamento de imagens*/
10  ...
11 }
```

O segundo passo foi analisar a imagem para detectar a presença de *landmarks*. Para isso, foram utilizados os parâmetros intrínsecos do sensor de visão do *V-REP*, funções da biblioteca *ArUco*, contidas na classe *aruco::MarkerDetector* e a classe herdeira *aruco::Marker*. Logo, todos os marcadores são armazenados em um vetor de alocação dinâmica. E por fim, foi elaborada uma função para o cálculo dos parâmetros extrínsecos *rVec* e *tVec*:

```

1 ...
2 /*Parâmetros intrínsecos da camera */
3 const Mat cameraMatrix = (Mat_<double>)(3,3) <<
4   221.70249590873925, 0.0, 128.0, 0.0, 221.70249590873925, 128.0,
5   0.0, 0.0, 1.0 );
6 const Mat distCoeffs = (Mat_<double>)(1,5) << 0.0, 0.0, 0.0, 0.0,
7   0.0);
```

```

5
6 aruco::MarkerDetector MDetector;
7 std::vector< aruco::Marker > Markers;
8 try
9 {
10     cv_ptr = cv_bridge::toCvCopy(msg);
11     MDetector.detect(cv_ptr->image, Markers); //Detecta landmarks
12
13     for (unsigned int i=0;i<Markers.size();i++) {
14         if (Markers.size() == 1) //garante a detecção de uma
15             landmark por vez.
16     {
17         Markers[i].draw(cv_ptr->image, Scalar(0,0,255),2);
18         calculateExtrinsics(Markers, 0.15, cameraMatrix,
19             distCoeffs, rVec, tVec);
20         //Obtem-se rVec e tVec.
21         ...
22     }
23 }
```

No trecho do programa acima, a localização relativa do marcador em relação à câmera foi computada (definida pelos vetores  $rVec$  e  $tVec$ ). O vetor  $rVec$  é representado por meio da representação de ângulos de Euler. Logo:

$$\mathbf{rVec} = \begin{bmatrix} \theta_x \\ \theta_y \\ \theta_z \end{bmatrix}$$

e

$$\mathbf{tVec} = \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix}$$

Sendo assim, para a explicação do cálculo da posição relativa entre câmera e marcador, algumas definições são necessárias.

O *frame*  $M_{(k)}$  de uma *landmark* arbitrária é definido como  $\{X_{m(k)}, Y_{m(k)}, \theta_{m(k)}\}$ , sendo  $k$  o índice do marcador em questão. Sua posição em relação ao *frame* de referência  $C$ , referente à câmera do robô, pode ser escrita como  ${}^C\xi_{M(k)} = ({}^CX_{M(k)}, {}^CY_{M(k)}, {}^C\theta_{M(k)})$ . A orientação da etiqueta em relação à câmera ( ${}^C\theta_{M(k)}$ ) é equivalente ao ângulo de Euler ao redor do eixo Z do marcador. Os seja, o valor de  ${}^C\theta_{M(k)}$  está contido em  $rVec$ .

O vetor  $rVec$  foi convertido em uma matriz de rotação  $\mathbf{R}$  (3x3), que é uma representação mais adequada para a realização dos cálculos desta etapa. Portanto, a posição relativa do centro da etiqueta até a origem do *frame*  $C$  é representado na equação 3.11.

$$\begin{bmatrix} {}^C X_{M(k)} \\ {}^C Y_{M(k)} \\ {}^C Z_{M(k)} \end{bmatrix} = \mathbf{R} \begin{bmatrix} {}^{M(k)} X_C \\ {}^{M(k)} Y_C \\ {}^{M(k)} Z_C \end{bmatrix} + \mathbf{tVec} \quad (3.11)$$

Que pode ser escrito como:

$$\begin{bmatrix} {}^{M(k)} X_C \\ {}^{M(k)} Y_C \\ {}^{M(k)} Z_C \end{bmatrix} = \mathbf{R}^{-1} \left( \begin{bmatrix} {}^C X_{M(k)} \\ {}^C Y_{M(k)} \\ {}^C Z_{M(k)} \end{bmatrix} - \mathbf{tVec} \right) \quad (3.12)$$

Em 3.12 tem-se o cálculo do vetor de posição 3D que representa a posição da câmera em relação ao marcador observado. Logo, como os vetores de posições  $({}^C X_{M(k)}, {}^C Y_{M(k)}, {}^C Z_{M(k)})^T$  (tal que  $k = 0, 1, 2, \dots, 41$ ) são conhecidos, é possível estimar a posição 3D da câmera em relação ao *frame* de referência global. Identificando-se o índice do marcador detectado e comparando-o em uma *look-up-table*. Como mostrado no trecho de código a seguir:

```

1 cv::Matx<double, 3, 1> getMarkersPoses(int id)
2 {
3     cv::Matx<double, 3, 1> poses;
4     switch (id)
5     {
6         case 0: poses = { -2.0000, -4.5000, 0 };
7             break;
8         case 1: poses = { -1.0000, -4.5000, 0 };
9             break;
10        case 2: poses = { 0, -4.5000, 0 };
11            break;
12        ...
13
14        case 40: poses = { 1.0000, 4.5000, 0 };
15            break;
16        case 41: poses = { 2.0000, 4.5000, 0 };
17            break;
18        default: break;
19    }
20    return poses;
21 }
```

No contexto deste trabalho, o valor de  ${}^C Z_{M(k)}$  é dispensável, visto que o robô se locomove sobre um plano e a localização desejada é bidimensional. Portanto, o vetor de localização

do sensor de visão (câmera) é dado por:

$$\xi_C = \begin{bmatrix} {}^C X_{M(k)} \\ {}^C X_{M(k)} \\ -\theta_z \end{bmatrix} \quad (3.13)$$

Como a origem do *frame* da câmera está localizada exatamente sobre o centro de giro do robô, a posição estimada em 3.13 é equivalente à posição/orientação do robô móvel no ambiente. A figura 18 mostra o nó *cv\_node* em funcionamento.

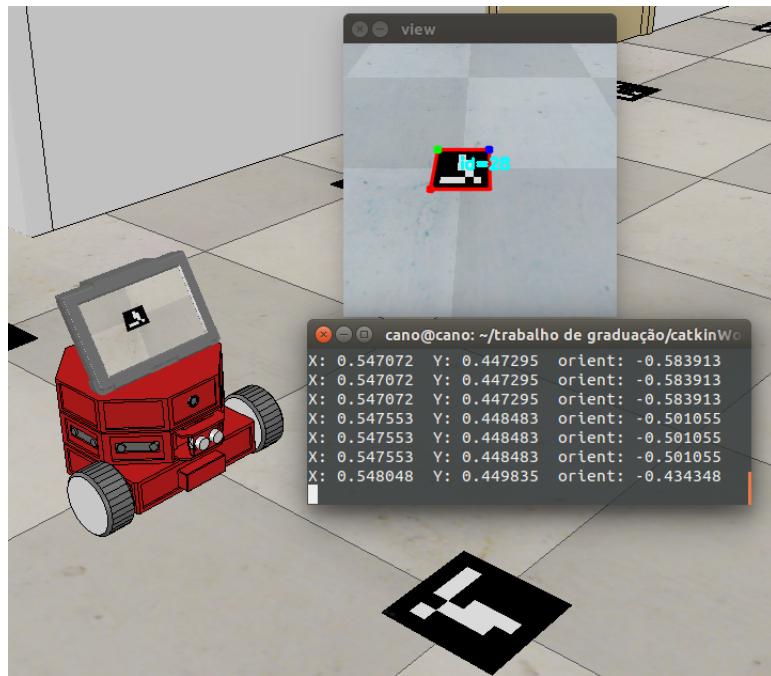


Figura 18 – Determinação de localização pelo módulo *cv\_node*.

Fonte: Produzido pelo autor.

Finalmente, publicou-se este vetor de posição em uma mensagem do tipo *float64* no tópico "CorrecaoPose" do *ROS*. Um outro nó para o cálculo da odometria (*odometria\_node*) foi desenvolvido. Este programa é similar ao desenvolvido anteriormente para o *dead reckoning*, contudo este nó recebe as atualizações da localização do robô, emitidas pelo programa *cv\_node*. Logo, foi obtido um cálculo de odometria que é corrigido arbitrariamente a cada detecção de *landmarks*.

Este processo pode ser visualizado por meio do grafo da figura 19:

Por fim, os dados de posição/orientação de odometria com correção foram publicados por meio do tópico *OdomToVrep* para análise no simulador.

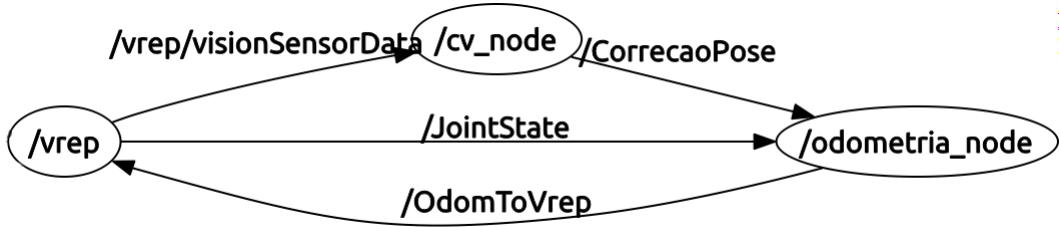


Figura 19 – Grafo do experimento de localização por visão computacional e *landmarks*.

Fonte: Produzido pelo autor.

### 3.3.3 SLAM

Na terceira e última metodologia foi utilizado o pacote *gmapping* do *ROS* para realizar o processo de mapeamento e localização simultâneos, a partir do sensor *laser rangefinder* acoplado no robô móvel e da reutilização do pacote de *dead reckoning* (primeiro método de localização deste trabalho).

Para aplicar esta técnica foi necessário reutilizar o nó *odometria\_simple\_node* e, além disso, publicar dois dados necessários para o *SLAM*. As medições do sensor laser e a árvore de transformações do *ROS*, dada pelo pacote *tf* (SAITO, 2015). A árvore de transformações corresponde a uma estrutura de dados que armazena as relações entre componentes do robô. Por meio do nó *tf*, as transformações de base relevantes para uma determinada aplicação são definidas. Por padrão, um sistema de odometria no *ROS* é representado pela árvore de transformações da figura 20.

Em que *odom* é o sistema de eixos de referência global do ambiente, que é fixo, *base\_link* é o *frame* fixo ao robô móvel e o *Hokuyo\_URG\_04LX\_UG01\_ROS* consiste no *frame* fixo no sensor *laser rangefinder* simulado.

Com isto, executou-se o nó *slam\_gmapping* durante a realização do percurso do robô móvel no *V-REP*. De modo que mais um *frame* foi adicionado à árvore de transformações, como mostra a figura 21.

A transformação *map -> base\_link*, publicada pelo *slam\_gmapping*, contém as informações sobre a localização do robô móvel obtidas por meio do algoritmo para *SLAM*, com base em mapeamento de Rao-Blackwell. Esta informação não pode ser enviada para o *V-REP* na forma em que foi publicada por este nó. Portanto, a informação da árvore de transformação (*map -> base\_link*) foi convertida para uma mensagem de posição geométrica e orientação em um outro módulo, denominado *pose\_publisher*.

Com isso, a localização por meio do *SLAM* foi obtida e, assim como nos casos anteriores, publicada para ser lida pelo nó *V-REP* e analisada. Adicionalmente, um mapa métrico do ambiente foi gerado e visualizado por meio do programa *rviz* (COLEMAN, 2015). O grafo que resume este terceiro experimento é apresentado na figura 22.

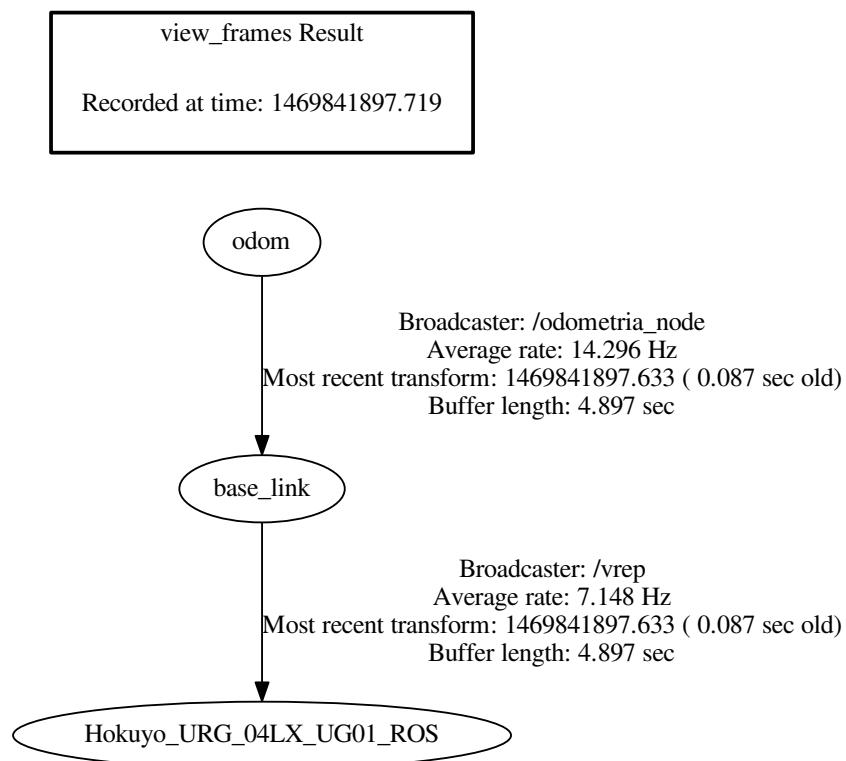


Figura 20 – Árvore de transformações padrão para odometria. Cada “seta” representa um matriz de transformação homogênea entre dois *frames*.

Fonte: Produzido pelo autor.

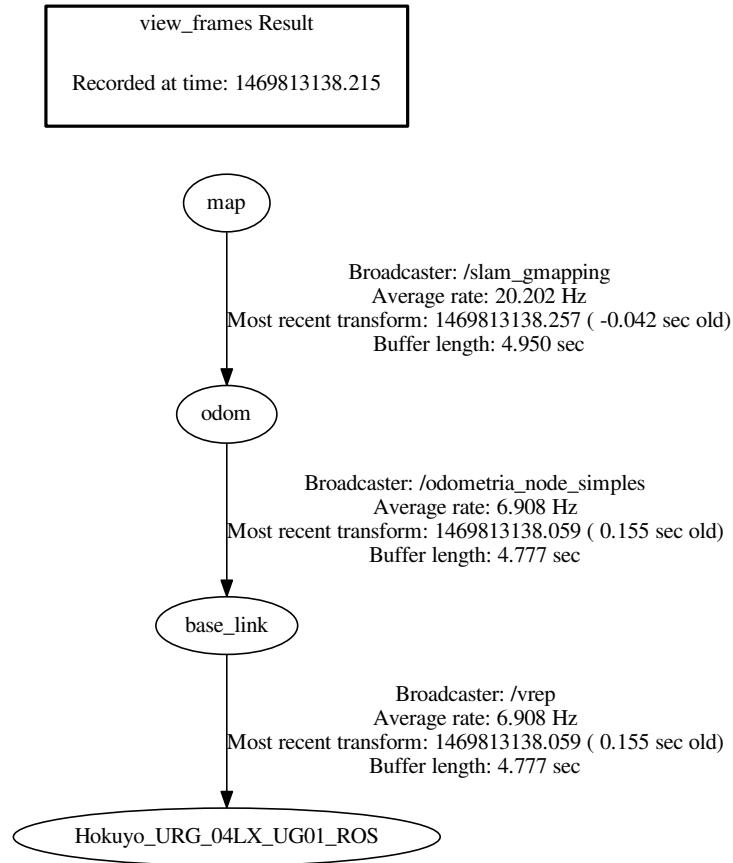


Figura 21 – Árvore de transformações para o pacote *gmapping*.

Fonte: Produzido pelo autor.

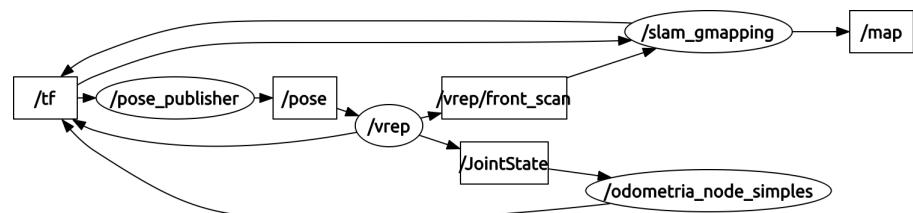


Figura 22 – Grafo da metodologia de localização por *SLAM*.

Fonte: Produzido pelo autor.

## 4 Resultados e discussões

Neste capítulo serão apresentados os resultados obtidos a partir da aplicação das três metodologias desenvolvidas neste trabalho. Em todos os casos, a análise e apresentação dos resultados da crença de localização do robô foram realizado no software *V-REP*. De modo a disponibilizar os gráficos de crença de localização do *dr20* no próprio ambiente. Para isto, cada metodologia foi comparada com a localização real em um trajeto comum. Tal como mostra a figura 23.

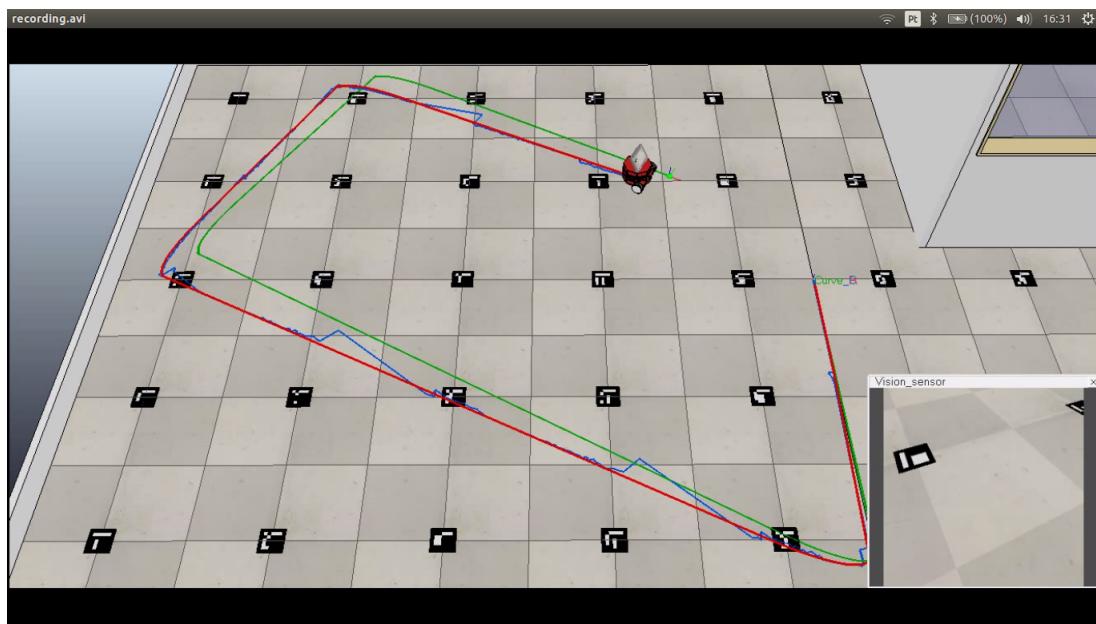


Figura 23 – Simulação das metodologias no V-REP.

Fonte: Produzido pelo autor.

Aplicando-se a metodologia de *dead reckoning*, o perfil de trajetória obtido é apresentado na figura 24.

Nota-se que, conforme a movimentação do robô, a localização por odometria pura tende a se distanciar da localização real. No entanto, devido às modificações que resultaram na equação simplificada 3.10, observa-se que o erro converge para o interior da curva, o que reduziu o erro de posição final. Ainda assim, este método opera completamente em “malha aberta”, confiando apenas em seus sensores proprioceptivos, e qualquer alteração forçada na posição do robô pode fazer com que este sistema de localização seja inutilizado. Logo, esta metodologia é pouco acurada, pois com o robô fica sujeito a erros sistemáticos (desalinhamento das rodas, limitação da resolução dos sensores do tipo *encoder*, imperfeições mecânicas) e não sistemáticos (fatores externos, derrapagem, piso irregular).

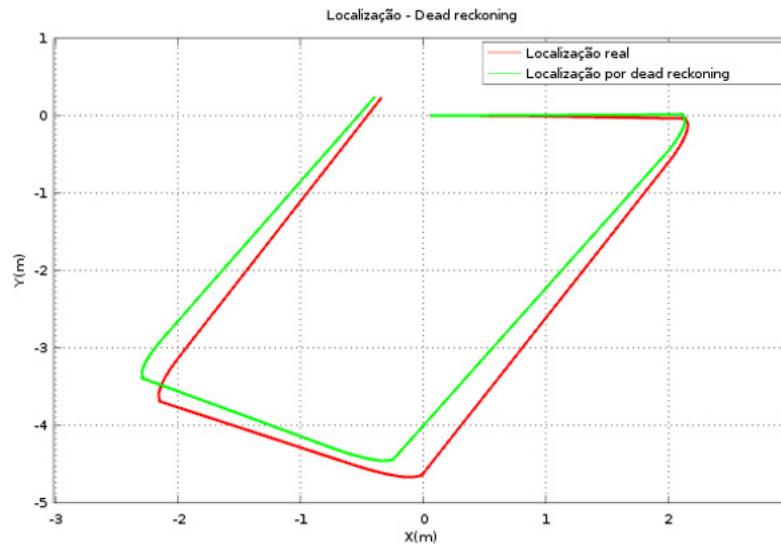


Figura 24 – Localização por *dead reckoning*.

Fonte: Produzido pelo autor.

Em seguida, aplicou-se a metodologia de localização por meio de marcadores fiduciais de realidade aumentada, aqui utilizados como landmarks, visão computacional e odometria. Para o mesmo caminho percorrido, a crença de localização obtida por meio deste método é apresentado na figura 25.

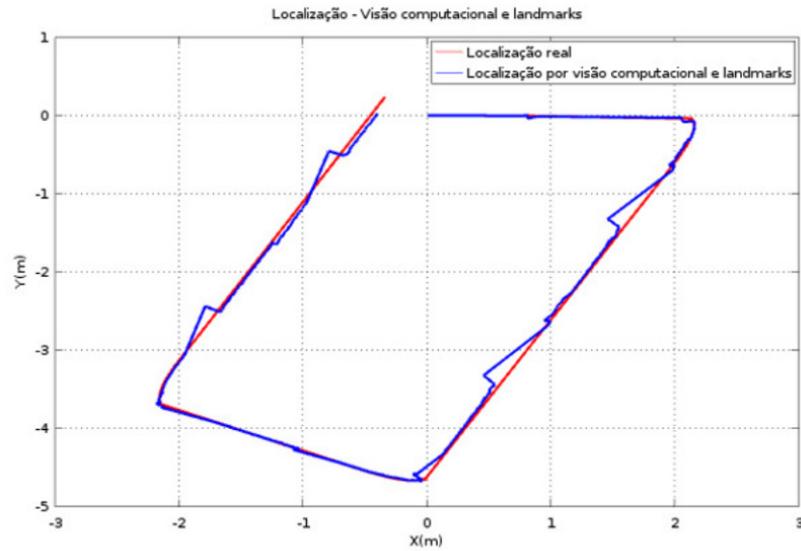


Figura 25 – Localização por visão computacional e *landmarks*.

Fonte: Produzido pelo autor.

Neste caso, foi possível perceber que a localização estimada possui certas descontinuidades. Isto ocorre porque nos instantes em que as *landmarks* são detectadas a odometria é corrigida bruscamente. Apesar da correção da posição ser satisfatória, observou-se um

erro considerável na orientação estimada do robô. Ou seja, a posição do robô é corrigida, mas sua orientação é ajustada para um valor com um erro previsível.

Visando reduzir o erro sobre a orientação estimada, foi executado um procedimento de correção com o auxílio do *V-REP*, realizado da seguinte maneira: Posicionou-se o robô *dr20* em frente um marcador, sendo que a orientação  $\theta$  real (informada pelo *V-REP*) fosse equivalente à  $0^\circ$ , e o valor da orientação estimada foi registrado. Após isso, o robô móvel foi novamente reposicionado, de modo que sua orientação real fosse acrescida em  $20^\circ$ . O processo foi repetido até o robô ter completado uma volta completa ( $360^\circ$ ) em torno de um marcador. A figura 26 apresenta a relação entre a orientação determinada pelo algoritmo de estimativa de posição e a orientação real do *dr20*.

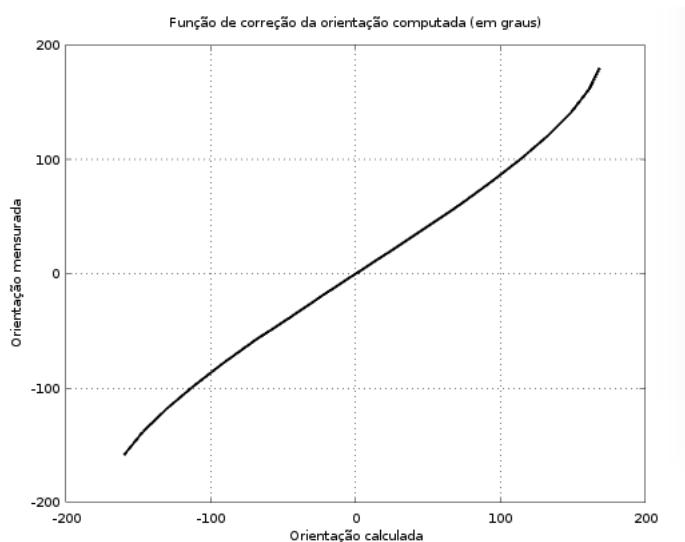


Figura 26 – Relação entre a orientação calculada e medida, obtida experimentalmente.

Fonte: Produzido pelo autor.

Por fim, calculou-se uma função polinomial para relacionar a orientação estimada pelos algoritmos de estimativa de posição com a orientação real do robô. Logo, aplicou-se uma interpolação por *spline* cúbica a fim de se obter uma função polinomial (4.1) de correção.

$$\theta_{corrigido} = \theta^3 \cdot 0.0283944 + \theta^2 \cdot 0.0054546 + \theta \cdot 0.7742107 - 0.0111378 \quad (4.1)$$

Aplicou-se esta modificação ao algoritmo, aprimorando a orientação estimada do robô. O experimento foi realizado novamente, e o resultado é apresentado na figura 27.

Observou-se uma melhoria no sistema de localização, uma vez que as descontinuidades foram atenuadas por conta da correção de  $\theta$ . Deste modo, observa-se que esta metodologia possui maior robustez do que o sistema de *dead reckoning*. Pois a posição é

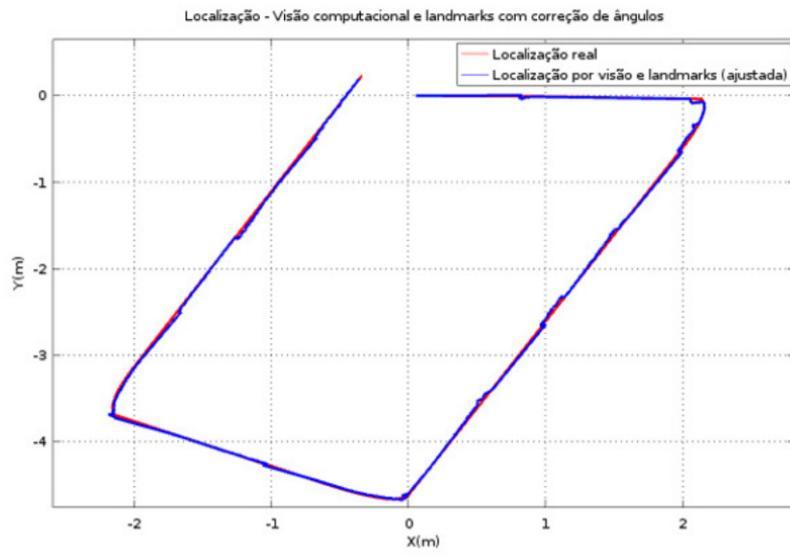


Figura 27 – Localização por visão computacional e *landmarks* com correção da orientação.

Fonte: Produzido pelo autor.

ajustada de frequentemente, sendo que no instante em que um marco é capturado pela câmera, o robô atualiza sua posição e orientação. Assim sendo, este sistema é eficaz até mesmo em casos de intervenção externa na posição do robô. Desta forma, caso o robô seja subitamente movido de local durante a navegação, o sistema de localização será rapidamente ajustado. A figura 28 mostra o resultado desta intervenção.

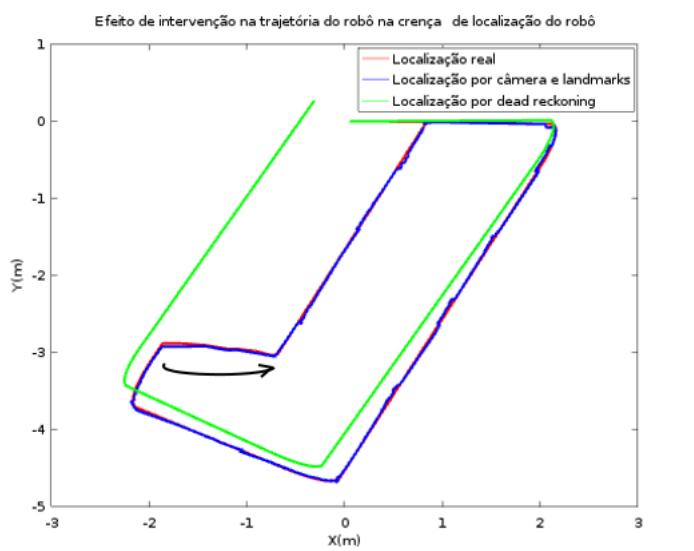


Figura 28 – Comparação da crença de localização para o caso de mudança súbita da posição do robô (indicada pela seta).

Fonte: Produzido pelo autor.

Esta segunda metodologia foi aplicada em um outro projeto, para resolver o

problema de autolocalização de um veículo semi-autônomo industrial, no qual verificou-se a funcionalidade da metodologia fora do ambiente de simulação. Como apresentado na figura 29.



Figura 29 – Marcadores de realidade aumentada posicionados no chão em ambiente real.

Fonte: Produzido pelo autor.

Notou-se um potencial problema desta metodologia, quando aplicada em ambiente real, que é a necessidade de boas condições de iluminação sobre as *landmarks*. Visto que a câmera pode não detectar os marcos corretamente quando há uma forte incidência de luz nas etiquetas, ou a ausência dela no ambiente.

Por fim, verificou-se a eficácia do algoritmo *slam\_gmapping* para a obtenção da localização. A partir das informações oriundas da odometria e do sensor *laser rangefinder*, um mapa foi criado durante o percurso do robô. Este mapa é apresentado na figura 30.

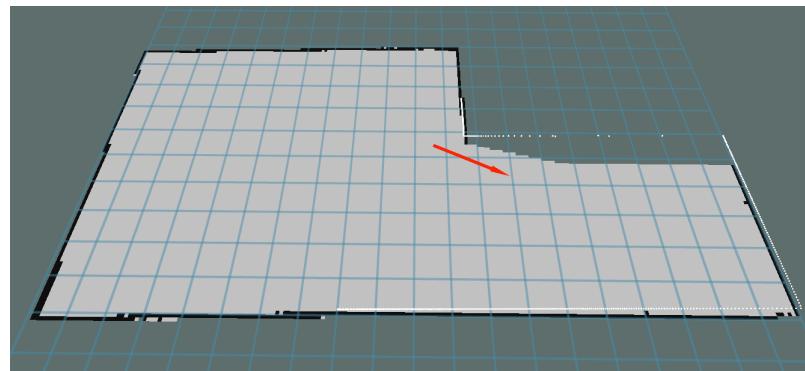


Figura 30 – Processo de construção de um mapa por meio do algoritmo para *SLAM* implementado.

Fonte: Produzido pelo autor.

Os resultados obtidos em relação à localização por *SLAM* é mostrado na figura 31.

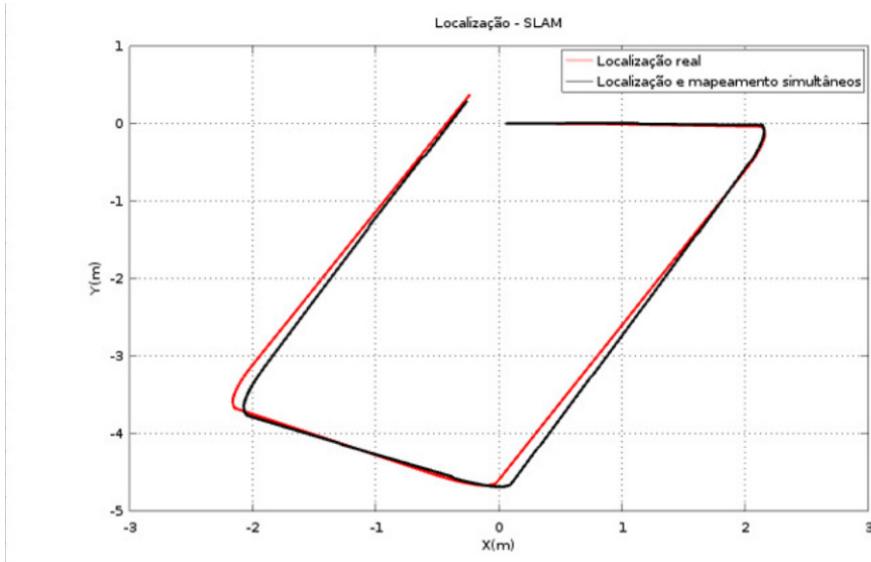


Figura 31 – Localização publicada pelo nó *slam\_gmapping*.

Fonte: Produzido pelo autor.

Neste caso, nota-se uma certa similaridade do perfil da crença de localização por *SLAM* com o demonstrado no método de reconhecimento passivo (primeiro caso). Isto ocorre porque o *slam\_gmapping* depende da odometria durante o processo de construção do mapa. No entanto, o sensor externo do robô fornece um *feedback* independente da odometria, assim como na metodologia por visão computacional.

Caso o robô continue percorrendo o ambiente, o mapa é progressivamente aprimorado. Após um tempo com o robô em operação, obteve-se o mapa completo do ambiente. Apresentado na figura 32.

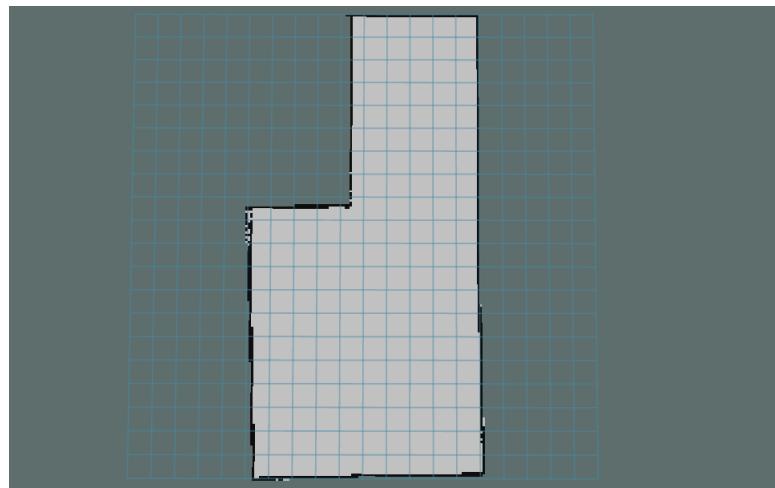


Figura 32 – Mapa construído apóis SLAM.

Fonte: Produzido pelo autor

Este mapa indica as regiões livres de obstáculos e serve como base para correção da localização do robô. Sendo atualizado periodicamente durante a execução do algoritmo. Isto é vantajoso para ambientes dinâmicos, nos quais os obstáculos podem ser adicionados ou retirados. O *ROS* permite fácil acesso ao tópico que publica os dados da matriz multidimensional que define o mapa, e estas informações podem ser utilizadas por sistemas de navegação ou de planejamento de tarefas de alto nível.

## 5 Conclusão

A implementação da metodologia de localização por visão computacional e marcadores fiduciais foi bem sucedida. Assim como a odometria pura e o *SLAM*. Conclui-se, portanto, que o sistema de localização por reconhecimento passivo se mostrou uma metodologia rudimentar para o cálculo da crença de localização. No entanto, a utilidade desta sistemática está na sinergia da odometria com outras metodologias. O cálculo de odometria pode ser utilizado como base para algoritmos mais elaborados, e que se baseiam na colaboração entre sensores externos e internos. Tal como mostrado nos outros dois sistemas de localização deste trabalho.

Com relação à metodologia por visão computacional, foi possível concluir que, após a correção experimental da orientação do robô, foi obtido um sistema de localização consistente para ambiente interno e conhecido. Promovendo a crença de localização mais próxima da real entre as três técnicas, considerando-se o ambiente simulado. No entanto, para ser aplicada na prática é preciso garantir uma preparação prévia do ambiente. Pois é necessário cadastrar em uma *look-up-table* as posições de cada *landmark* corretamente, garantir boas condições de iluminação no ambiente interno e pouca ou nenhuma reflexão sobre os marcadores de realidade aumentada, já que reflexos na etiqueta podem atrapalhar a identificação do índice dos marcadores e comprometer o sistema de localização. Outra desvantagem se dá no fato da crença de localização fornecida por esta metodologia apresentar algumas descontinuidades, o que pode prejudicar seu uso em um sistema de navegação ou de construção de mapas.

Adicionalmente, é uma sistemática que envolve um custo menor do que o *SLAM* via laser. Visto que para ser implementada, basta utilizar etiquetas de realidade aumentada, que podem ser fixas no piso, paredes ou objetos estáticos, além de um sensor de visão. Que pode até mesmo ser uma simples *webcam* com resolução da ordem de 640x480 px.

Notou-se que a metodologia por mapeamento e simulação simultâneos apresentou a vantagem de não requerer conhecimento nem preparo prévio do ambiente, como ocorre com a segunda metodologia com a distribuição de *landmarks* artificiais. Além disso, os resultados sobre a localização do robô a partir desta técnica indicam uma boa eficácia do *slam\_gmapping*, que também utiliza a odometria, mas realimenta a crença de localização por meio do sensor externo *laser rangefinder*. Esta metodologia apresenta maior complexidade e custo, visto que sensores laser de curto alcance (cerca de 5 metros, como o usado neste trabalho) apresentam valores em torno de USD \$1,115.00. Enquanto sensores de maior alcance, como o Hokuyo UTM-30LX Scanning Laser Rangefinder (30m), custam em torno de USD \$4,825.00.

Em resumo, conclui-se que um sistema de localização deve ser robusto e adequado ao ambiente, custo, grau de autonomia e capacidade de processamento do robô. O *emphdead reckoning* se mostrou pouco confiável e sensível a erros sistemáticos e não sistemáticos. Sendo útil para pequenas distâncias e integração com medidas externas. A localização por visão computacional e *landmarks* pode ser utilizada em robôs autônomos ou semi-autônomos, operando em local conhecido e com boa iluminação. No qual haja a possibilidade de alocar marcadores de realidade aumentada em pontos estratégicos. Já o *SLAM* pode operar em ambientes mais dinâmicos, pois registra constantemente uma representação do ambiente em um mapa métrico. Além de também promover uma crença de localização robusta e praticamente sem descontinuidades.

Verificou-se também a importância de bibliotecas de software livre como o *OpenCV*, softwares de simulação e a plataforma *ROS*. Que vem trazendo colaboratividade entre pesquisadores e aprimorando o desenvolvimento de tecnologia em robótica.

# Referências

- BAATH, R. A. *Map-making Robots: A Review of the Occupancy Grid Map Algorithm.* 2008. Disponível em: <<http://www.ikaros-project.org/articles/2008/gridmaps/>>. Acesso em: 30 jul 2016. Citado na página 28.
- BALAKRISHNAN, N.; NEVZOROV, V. B. *A primer on statistical distributions.* [S.l.]: John Wiley & Sons, 2004. Citado na página 24.
- BEEVERS, K. *Topological mapping with sensing-limited robots.* 2004. Disponível em: <<http://cs.krisbeever.com/topological.html>>. Acesso em: 30 jul 2016. Citado na página 29.
- BORENSTEIN, J.; EVERETT, H. R.; FENG, L. "Where am I? - Systems and Methods for Mobile Robot Positioning. [S.l.: s.n.], 1996. Citado na página 37.
- BRADSKI, G.; KAEHLER, A. *Learning OpenCV: Computer vision with the OpenCV library.* [S.l.]: "O'Reilly Media, Inc.", 2008. Citado na página 20.
- BROOKS, R. A robust layered control system for a mobile robot. *IEEE Journal on Robotics and Automation*, v. 2, n. 1, p. 14–23, 1986. ISSN 0882-4967. Acesso em: 04 ago 2016. Citado na página 24.
- C.BROWN, D. Decentering distortion of lenses. *Photogrammetric Engineering*, 1966. Citado na página 23.
- COLEMAN, D. *A Tutorial and Elementary Trajectory Model for the Differential Steering System of Robot Wheel Actuators.* 2015. Disponível em: <<http://wiki.ros.org/rviz>>. Acesso em: 05 ago 2016. Citado na página 44.
- COPPELIA. *Virtual Robot Experimentation Platform user manual.* 2016. Citado na página 16.
- GARRIDO-JURADO, S. et al. Automatic generation and detection of highly reliable fiducial markers under occlusion. *Pattern Recognition*, v. 47, n. 6, p. 2280 – 2292, 2014. ISSN 0031-3203. Disponível em: <<http://www.sciencedirect.com/science/article/pii/S0031320314000235>>. Citado 2 vezes nas páginas 31 e 33.
- GRISSETTI, G.; STACHNISS, C.; BURGARD, W. Improved techniques for grid mapping with rao-blackwellized particle filters. *IEEE Transactions on Robotics*, v. 23, n. 1, p. 34–46, Feb 2007. ISSN 1552-3098. Citado na página 30.
- HALLAM, B.; FLOREANO, D.; MEYER, J.-A. Learning to autonomously select landmarks for navigation and communication. MIT Press, p. 151–160, 2002. Disponível em: <<http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6278220>>. Citado na página 27.
- HOKUYO. *Scanning range finder (SOKUIKI sensor).* 2009. Disponível em: <[https://www.hokuyo-aut.jp/02sensor/07scanner/urg\\_04lx\\_ug01.html](https://www.hokuyo-aut.jp/02sensor/07scanner/urg_04lx_ug01.html)>. Acesso em: 02 ago 2016. Citado na página 35.

- ITSEEZ. *The OpenCV Reference Manual*. 2.4.9.0. ed. [S.l.], 2014. Acesso em: 26 jul 2016. Citado 3 vezes nas páginas 21, 22 e 23.
- ITSEEZ. *Open Source Computer Vision Library*. 2015. <<https://github.com/itseez/opencv>>. Acesso em: 26 jul 2016. Citado na página 20.
- LUCAS, G. *A Tutorial and Elementary Trajectory Model for the Differential Steering System of Robot Wheel Actuators*. 2000. Disponível em: <<http://rossum.sourceforge.net/papers/DiffSteer/#d6>>. Acesso em: 01 ago 2016. Citado na página 37.
- MATHWORKS. *CameraParameters class*. 2016. Disponível em: <<http://www.mathworks.com/help/vision/ref/cameraparameters-class.html>>. Acesso em: 26 jul 2016. Citado na página 24.
- MAZZARI, V. *What is ROS?* 2016. Génération Robots. Disponível em: <<http://www.generationrobots.com/blog/en/2016/03/ros-robot-operating-system-2/>>. Acesso em: 26 jul 2016. Citado na página 19.
- O'KANE, J. M. *A Gentle Introduction to ROS*. CreateSpace Independent Publishing Platform, 2013. ISBN 9781492143239. Disponível em: <<http://www.cse.sc.edu/~jokane/agitr/>>. Citado 2 vezes nas páginas 17 e 18.
- PHIDGETS. *Encoder Primer*. 2014. Disponível em: <[http://www.phidgets.com/docs/Encoder\\_Primer](http://www.phidgets.com/docs/Encoder_Primer)>. Acesso em: 15 jul 2016. Citado na página 26.
- RIISGAARD, S.; BLAS, M. R. *SLAM for Dummies - A Tutorial Approach to Simultaneous Localization and Mapping*. [S.l.: s.n.], 2004. Citado 2 vezes nas páginas 26 e 29.
- ROMERO, A. M. *Wiki: ROS/Concepts*. 2014. Wiki do ROS. Disponível em: <<http://wiki.ros.org/ROS/Concepts>>. Acesso em: 16 jul 2016. Citado 2 vezes nas páginas 18 e 19.
- ROMERO, R. A. F. et al. *Robótica móvel*. 1. ed. [S.l.: s.n.], 2014. Citado 4 vezes nas páginas 14, 24, 27 e 29.
- ROSENHAHN CHRISTIAN PERWASS, G. S. B. *Foundations about 2D-3D Pose Estimation*. [S.l.]: CVonline: On-Line Compendium of Computer Vision - R. Fisher (Ed), 2004. Citado na página 21.
- SAITO, I. *Wiki: tf*. 2015. Wiki do ROS. Disponível em: <<http://wiki.ros.org/tf>>. Acesso em: 5 ago 2016. Citado na página 44.
- SCHWEIGERT, S. *Wiki: gmapping*. 2015. Wiki do ROS. Disponível em: <<http://wiki.ros.org/tf>>. Acesso em: 07 ago 2016. Citado na página 30.
- SEKIMOR, D.; MIYAZAKI, F. Precise dead-reckoning for mobile robots using multiple optical mouse sensors. *Informatics in Control, Automation and Robotics II*, 2007. Citado na página 26.
- SIEGWART, R.; NOURBAKHSH, I. R. *Introduction to Autonomous Mobile Robots*. Scituate, MA, USA: Bradford Company, 2004. ISBN 026219502X. Citado 3 vezes nas páginas 14, 25 e 36.
- THOMAS, D. *Wiki: ROS/Introduction*. 2014. Wiki do ROS. Disponível em: <<http://wiki.ros.org/ROS/Introduction>>. Acesso em: 15 jul 2016. Citado na página 17.

## Anexos

Nó *odometria\_node* para calculo e correção de odometria.

```
1 #include <iostream>
2 #include <ros/ros.h>
3 #include <sensor_msgs/JointState.h>
4 #include <std_msgs/Float64MultiArray.h>
5 #include <nav_msgs/Odometry.h>
6 #include <tf/transform_broadcaster.h>
7
8 using namespace std;
9
10 class OdomCorrigido
11 {
12 private:
13     ros::NodeHandle nh;
14     ros::Subscriber jntSub;
15     ros::Subscriber cvSub;
16     ros::Publisher odomPub;
17     ros::Publisher sPosePub; // Publisher de mensagem do tipo
18     // pose stamped.
19     tf::TransformBroadcaster odom_broadcaster; //
20     double x, y, theta;
21     int countCorrecao, countJointState;
22 public:
23     OdomCorrigido();
24     void jointCallback(const sensor_msgs::JointState::
25         ConstPtr& msg);
26     void cvCallback(const std_msgs::Float64MultiArray::
27         ConstPtr& corr);
28 };
29
30 OdomCorrigido::OdomCorrigido()
31 {
32     // inicializa o das variáveis privadas:
33     x = 5.5119e-02;
34     y = -4.8883e-04;
35     theta = 1.0000e-02;
36     countCorrecao = 0;
37     countJointState = 1;
38
39     //Callbacks
40     cvSub = nh.subscribe("/CorrecaoPose", 1,&OdomCorrigido::
41         cvCallback, this);
```

```
38     jntSub = nh.subscribe("/JointState", 1,&OdomCorrigido::  
39         jointCallback, this);  
40     odomPub = nh.advertise<nav_msgs::Odometry>("Odometria",  
41         10);  
42     sPosePub = nh.advertise<geometry_msgs::PoseStamped>("OdomToVrep", 10);  
43 }  
44  
45 // Correcao de odometria:  
46 void OdomCorrigido::cvCallback(const std_msgs::Float64MultiArray  
47     ::ConstPtr& corr)  
48 {  
49     if(corr->data.size() >= 1)  
50     {  
51         x = corr->data[0];  
52         y = corr->data[1];  
53         theta = corr->data[2];  
54         countCorrecao++;  
55         //cout<< "\nPosicoes corrigidas\n";  
56     }  
57 }  
58  
59 // C lculo da odometria:  
60 void OdomCorrigido::jointCallback(const sensor_msgs::JointState::  
61     ConstPtr& msg)  
62 {  
63     double lPos, rPos, s, b = 0.25408, sl, sr, v, vx, vy;  
64     if (countCorrecao != countJointState)  
65     {  
66         lPos = msg->velocity[2];  
67         rPos = msg->velocity[3];  
68  
69         sr = 0.0425*rPos*0.05;  
70         sl = 0.0425*lPos*0.05;  
71         s = ((sr + sl))/2;  
72         v = s/0.05;  
73  
74         theta = ((sr - sl))/2*b+ theta;  
75         x = x + s*cos(theta);  
76         y = y + s*sin(theta);  
77 }
```

```
75             vx = v*cos(theta);
76             vy = v*sin(theta);
77
78         }
79     else
80     {
81         cout<< "\n\n Hulll\n";
82         countJointState++;
83     }
84
85     geometry_msgs::Quaternion odom_quat = tf::
86         createQuaternionMsgFromYaw(theta);
87
88 //tf
89     geometry_msgs::TransformStamped odom_trans;
90     odom_trans.header.stamp = msg->header.stamp;
91     odom_trans.header.frame_id = "odom";
92     odom_trans.child_frame_id = "base_link";
93     odom_trans.transform.translation.x = x;
94     odom_trans.transform.translation.y = y;
95     odom_trans.transform.translation.z = 0.0;
96     odom_trans.transform.rotation = odom_quat;
97
98     odom_broadcaster.sendTransform(odom_trans);
99
100 //Pose Stamped para o Vrep:
101     geometry_msgs::PoseStamped pstamp;
102     pstamp.header.stamp = msg->header.stamp;
103     pstamp.header.frame_id = "base_link";
104     pstamp.pose.position.x = x;
105     pstamp.pose.position.y = y;
106     pstamp.pose.position.z = 0.0;
107     pstamp.pose.orientation = odom_quat;
108
109
110 //Odom
111     nav_msgs::Odometry odom;
112     odom.header.stamp = msg->header.stamp;
113     odom.header.frame_id = "odom";
114
115     odom.pose.pose.position.x = 0;
```

```

116     odom.pose.pose.position.y = 0;
117     odom.pose.pose.position.z = 0.0;
118     odom.pose.pose.orientation = odom_quat;
119
120     odom.child_frame_id = "base_link";
121     odom.twist.twist.linear.x = 0;
122     odom.twist.twist.linear.y = 0;
123     odom.twist.twist.angular.z = 0.0;
124
125     sPosePub.publish(pstamp);
126     odomPub.publish(odom);
127
128     cout<<"X: "<<x<<" Y: "<<y<<" Theta: "<<theta<<endl;
129 }
130
131
132 int main(int argc, char **argv)
133 {
134
135     // Inicializa o do Node
136     ros::init(argc, argv, "odometria_node");
137     OdomCorrigido vaique;
138     ros::spin();
139 }
```

Nó *odometria\_simples* (sem correção):

```

1 #include <iostream>
2 #include <ros/ros.h>
3 #include <sensor_msgs/JointState.h>
4 #include <std_msgs/Float64MultiArray.h>
5 #include <nav_msgs/Odometry.h>
6 #include <tf/transform_broadcaster.h>
7 using namespace std;
8
9 class Odom
10 {
11 private:
12     ros::NodeHandle nh;
13     ros::Subscriber jntSub;
14     ros::Publisher odomPub;
15     ros::Publisher sPosePub;
16     tf::TransformBroadcaster odom_broadcaster;
```

```
17         double x, y, theta;
18
19 public:
20     Odom();
21     void jointCallback(const sensor_msgs::JointState::
22                         ConstPtr& msg);
23 };
24
25 Odom::Odom()
26 {
27     // inicializa o das variáveis privadas:
28     x = 5.5119e-02;
29     y = -4.8883e-04;
30     theta = 1.0000e-02;
31
32     //Callbacks
33     jntSub = nh.subscribe("/JointState", 1,&OdomCorrigido::
34                           jointCallback, this);
35     odomPub = nh.advertise<nav_msgs::Odometry>("Odometria_simples", 10);
36     sPosePub = nh.advertise<geometry_msgs::PoseStamped>("OdomToVrep_simples",10);
37 }
38
39 void Odom::jointCallback(const sensor_msgs::JointState::ConstPtr&
40                         msg)
41 {
42     double lPos, rPos, s, b = 0.25408, sl, sr, v, vx, vy;
43     lPos = msg->velocity[2];
44     rPos = msg->velocity[3];
45
46     sr = 0.0425*rPos*0.05;
47     sl = 0.0425*lPos*0.05;
48     s = ((sr + sl))/2;
49     v = s/0.05;
50
51     theta = ((sr - sl))/2*b+ theta;
52     x = x + s*cos(theta);
53     y = y + s*sin(theta);
```

```
54
55         vx = v*cos(theta);
56         vy = v*sin(theta);
57
58     geometry_msgs::Quaternion odom_quat = tf::
59             createQuaternionMsgFromYaw(theta);
60
61         //tf
62
63     geometry_msgs::TransformStamped odom_trans;
64     odom_trans.header.stamp = msg->header.stamp;
65     odom_trans.header.frame_id = "odom";
66     odom_trans.child_frame_id = "base_link";
67     odom_trans.transform.translation.x = x;
68     odom_trans.transform.translation.y = y;
69     odom_trans.transform.translation.z = 0.0;
70     odom_trans.transform.rotation = odom_quat;
71
72
73     odom_broadcaster.sendTransform(odom_trans);
74
75     //Pose Stamped para o Vrep:
76
77     geometry_msgs::PoseStamped pstamp;
78     pstamp.header.stamp = msg->header.stamp;
79     pstamp.header.frame_id = "base_link";
80     pstamp.pose.position.x = x;
81     pstamp.pose.position.y = y;
82     pstamp.pose.position.z = 0.0;
83     pstamp.pose.orientation = odom_quat;
84
85
86     //Odom
87
88     nav_msgs::Odometry odom;
89     odom.header.stamp = msg->header.stamp;
90     odom.header.frame_id = "odom";
91
92     odom.pose.pose.position.x = 0;
93     odom.pose.pose.position.y = 0;
94     odom.pose.pose.position.z = 0.0;
```

```

95     odom.twist.twist.linear.y = 0;
96     odom.twist.twist.angular.z = 0.0;
97
98     PosePub.publish(pstamp);
99     odomPub.publish(odom);
100
101    cout<<"X: "<<x<<" Y: "<<y<<" Theta: "<<theta<<endl;
102 }
103
104
105 int main(int argc, char **argv)
106 {
107
108     // Inicializa o do Node
109     ros::init(argc, argv, "odometria_node_simples");
110     OdomCorrigido vaique;
111     ros::spin();
112 }
```

Nó *cv\_node* para estimação de posição.

```

1 //Programa em C++ para localiza o da c mera com base na
   detec o de marcadores.
2
3 #include "../include/tg/funcoes.h"
4 #include <ros/ros.h>
5 #include <image_transport/image_transport.h>
6 #include <opencv2/opencv.hpp>
7 #include "../../cv_bridge/include/cv_bridge/cv_bridge.h"
8 #include <aruco/aruco.h>
9 #include <iostream>
10 #include <cstdio>
11 #include <std_msgs/Float64MultiArray.h>
12 #include <geometry_msgs/PoseStamped.h>
13 #include <tf/transform_broadcaster.h>
14
15 using namespace std;
16 using namespace cv;
17
18 const Mat cameraMatrix = (Mat_<double>(3,3) <<
   221.70249590873925, 0.0, 128.0, 0.0, 221.70249590873925, 128.0,
   0.0, 0.0, 1.0 );
19 const Mat distCoeffs = (Mat_<double>(1,5) << 0.0, 0.0, 0.0, 0.0,
```

```
    0.0);
20 const Matx<double, 3, 1> initPose = (0,0,0);
21
22 class cvProcess
23 {
24     private:
25         ros::NodeHandle nh;
26         ros::Publisher pub, testPub;
27         image_transport::Subscriber sub;
28     public:
29         cvProcess();
30         void imageCallback(const sensor_msgs::ImageConstPtr& msg)
31             ;
32 };
33
34 cvProcess::cvProcess()
35 {
36     pub = nh.advertise<std_msgs::Float64MultiArray>(
37         "CorrecaoPose", 10);
38     testPub = nh.advertise<geometry_msgs::PoseStamped>(
39         "Testando", 10);
40     image_transport::ImageTransport it(nh);
41     sub = it.subscribe("/vrep/visionSensorData", 20,
42         &cvProcess::imageCallback, this);
43     cv::namedWindow("view");
44     cv::startWindowThread();
45 }
46
47 cv::Matx<double, 3, 1> getMarkersPoses(int id)
48 {
49     cv::Matx<double, 3, 1> poses;
50     switch (id)
51     {
52         case 0: poses = { -2.0000, -4.5000, 0 };
53                 break;
54         case 1: poses = { -1.0000, -4.5000, 0 };
55                 break;
56         case 2: poses = { -8.3447e-07, -4.5000, 0 };
57                 break;
58         case 3: poses = { 1.0000, -4.5000, 0 };
59                 break;
```

```
57         case 4: poses = { 2.0000, -4.5000, 0 };
58             break;
59         case 5: poses = { -2.0000,-3.5000, 0 };
60             break;
61         case 6: poses = { -1.0000, -3.5000, 0 };
62             break;
63         case 7: poses = { -7.1526e-07, -3.5000, 0 };
64             break;
65         case 8: poses = { 1.0000, -3.5000, 0 };
66             break;
67         case 9: poses = { 2.0000, -3.5000, 0 };
68             break;
69         case 10: poses = { -2.0000, -2.5000, 0 };
70             break;
71         case 11: poses = { -1.0000, -2.5000, 0 };
72             break;
73         case 12: poses = { -7.4506e-07, -2.5000, 0 };
74             break;
75         case 13: poses = { 1.0000,-2.5000, 0 };
76             break;
77         case 14: poses = { -2.0000, -1.5000, 0 };
78             break;
79         case 15: poses = { 2.0000, -2.5000, 0 };
80             break;
81         case 16: poses = { -1.0000, -1.5000, 0 };
82             break;
83         case 17: poses = { -7.1526e-07, -1.5000, 0 };
84             break;
85         case 18: poses = { 1.0000, -1.5000, 0 };
86             break;
87         case 19: poses = { 2.0000, -1.5000, 0 };
88             break;
89         case 20: poses = { -2.0000e+00,-5.0000e-01, 0 };
90             break;
91         case 21: poses = { -1.0000e+00, -5.0000e-01, 0 };
92             break;
93         case 22: poses = { -9.0897e-07, -5.0000e-01, 0 };
94             break;
95         case 23: poses = { +1.0000e+00, -5.0000e-01, 0 };
96             break;
97         case 24: poses = { +2.0000e+00,-5.0000e-01, 0 };
98             break;
```

```
99         case 25: poses = {-2.0000, 4.5000, 0 };
100            break;
101        case 26: poses = {-1.0000, 5.0000e-01, 0 };
102            break;
103        case 27: poses = {-1.7062e-06, 5.0000e-01, 0 };
104            break;
105        case 28: poses = {1.0000, 5.0000e-01, 0 };
106            break;
107        case 29: poses = {2.0000, 5.0000e-01, 0 };
108            break;
109        case 30: poses = {-1.7062e-06, 1.5000, 0 };
110            break;
111        case 31: poses = {1.0000, 1.5000, 0 };
112            break;
113        case 32: poses = {2.0000, 1.5000, 0 };
114            break;
115        case 33: poses = {-1.7062e-06, 2.5000, 0 };
116            break;
117        case 34: poses = {1.0000, 2.5000, 0 };
118            break;
119        case 35: poses = {2.0000, 2.5000, 0 };
120            break;
121        case 36: poses = {-1.7062e-06, 3.5000, 0 };
122            break;
123        case 37: poses = {1.0000, 3.5000, 0 };
124            break;
125        case 38: poses = {2.0000, 3.5000, 0 };
126            break;
127        case 39: poses = {-1.7062e-06, 4.5000, 0 };
128            break;
129        case 40: poses = {1.0000, 4.5000, 0 };
130            break;
131        case 41: poses = {2.0000, 4.5000, 0 };
132            break;
133        default: std::cout << "\nId do marcador n o
134 condiz com as posi es setadas em
135 getMarkersPoses.\nVerificar se o n mero de
136 marcadores est correto ou se h
interfer ncia\n\n";
137            break;
138        }
139
140        return poses;
141    }
```

```
137 }
138
139
140 void calculateExtrinsics(vector<aruco::Marker> mVec, float
141   markerSizeMeters, cv::Mat camMatrix, cv::Mat distCoeff, Mat &
142   raux, Mat &taux) throw(cv::Exception)
143 {
144
145   double halfSize = markerSizeMeters / 2.;
146   cv::Mat ObjPoints(4, 3, CV_32FC1);
147   ObjPoints.at< float >(1, 0) = -halfSize;
148   ObjPoints.at< float >(1, 1) = halfSize;
149   ObjPoints.at< float >(1, 2) = 0;
150   ObjPoints.at< float >(2, 0) = halfSize;
151   ObjPoints.at< float >(2, 1) = halfSize;
152   ObjPoints.at< float >(2, 2) = 0;
153   ObjPoints.at< float >(3, 0) = halfSize;
154   ObjPoints.at< float >(3, 1) = -halfSize;
155   ObjPoints.at< float >(3, 2) = 0;
156   ObjPoints.at< float >(0, 0) = -halfSize;
157   ObjPoints.at< float >(0, 1) = -halfSize;
158   ObjPoints.at< float >(0, 2) = 0;
159
160   cv::Mat ImagePoints(4, 2, CV_32FC1);
161
162   // Set image points from the marker
163   for (int c = 0; c < 4; c++) {
164     ImagePoints.at< float >(c, 0) = (mVec[0][c].x);
165     ImagePoints.at< float >(c, 1) = (mVec[0][c].y);
166   }
167
168   //cv::Mat rv, tv;
169   cv::solvePnP(ObjPoints, ImagePoints, camMatrix, distCoeff,
170     raux, taux );
171
172 void cvProcess::imageCallback(const sensor_msgs::ImageConstPtr&
173   msg)
174 {
175
176   cv_bridge::CvImagePtr cv_ptr;
```

```
175     Mat rVecs, tVecs;
176     Matx<double, 3, 1> cameraPose, orientacao, auxT, auxR;
177     Matx<double, 3, 3> R, inv;
178     double auxY, auxX;
179     aruco::MarkerDetector MDetector;
180     vector< aruco::Marker > Markers;
181     try
182     {
183         cv_ptr = cv_bridge::toCvCopy(msg);
184         MDetector.detect(cv_ptr->image, Markers);
185
186         for (unsigned int i=0;i<Markers.size();i++) {
187             if (Markers.size() == 1)
188             {
189                 Markers[i].draw(cv_ptr->image, Scalar(0,0,255),2);
190                 calculateExtrinsics(Markers, 0.15, cameraMatrix,
191                                     distCoeffs, rVecs, tVecs);
192
193             //Vetores auxiliares
194
195             auxR(0) = rVecs.at<double>(0);
196             auxR(1) = rVecs.at<double>(1);
197             auxR(2) = rVecs.at<double>(2);
198
199             auxT(0) = tVecs.at<double>(0);
200             auxT(1) = tVecs.at<double>(1);
201             auxT(2) = tVecs.at<double>(2);
202             //cout<<"\nT0: "<< auxT(0) << "    T1: " <<
203             auxT(1) << "    T2: "<< auxT(2);
204
205             Rodrigues(auxR, R); //Converte de rVec (Vetor de
206             //angulos de rota o 1x3 para matriz de rota o
207             //3x3)
208             inv = R.t();
209             orientacao = rot2euler(inv);
210
211             cameraPose = inv*initPose - inv*auxT; //Equa o matricial de transforma o
212             auxY = cameraPose(0);
213             cameraPose(0) = cameraPose(1);
214             cameraPose(1) = -auxY;
```

```
212         cameraPose = cameraPose + getMarkersPoses
213             (Markers[0].id);
214         auxR(1) = pow(auxR(1),3)*0.0283944 + pow(
215             auxR(1),2)*0.0054546 + auxR(1)
216             *0.7742107 -0.0111378;
217         cout << "X: " << cameraPose(0) << " Y: " <<
218             cameraPose(1) << " orient: " << auxR
219             (1)*(180/CV_PI) << endl;
220 //cout << "Y: " << auxR(0)*(180/3.14159) <<
221             " Z: " << auxR(1)*(180/3.14159) << "
222                 orient: " << auxR(2)*(180/3.14159) <<
223                     endl;
224 //define vetor a ser transmitido
225 std::vector<double> vec;
226 vec.push_back(cameraPose(0));
227 vec.push_back(cameraPose(1));
228 vec.push_back(auxR(1));
229 //passa par o blob de dados de sa da
230 std_msgs::Float64MultiArray output;
231 output.layout.dim.push_back(std_msgs::
232             MultiArrayDimension());
233 output.layout.dim[0].size = vec.size();
234 output.layout.dim[0].stride = 1;
235 output.layout.dim[0].label = "pose";
236 vector<double>::const_iterator itr, end(
237             vec.end());
238 for(itr = vec.begin(); itr!= end; ++itr)
239 {
240     output.data.push_back(*itr);
241 }
242
243 //teste
244 geometry_msgs::Quaternion odom_quat = tf::
245             createQuaternionMsgFromYaw(auxR(1));
246
247 geometry_msgs::PoseStamped pstamp;
248
249 pstamp.pose.position.x = cameraPose(0);
250 pstamp.pose.position.y = cameraPose(1);
251 pstamp.pose.position.z = 0.0;
252 pstamp.pose.orientation = odom_quat;
```

```
243             testPub.publish(pstamp);
244             // fim do teste
245
246             pub.publish(output);
247             vec.clear();
248         }
249     }
250
251     cv::imshow("view", cv_ptr->image);
252 }
253 catch (cv_bridge::Exception& e)
254 {
255     ROS_ERROR("cv_bridge exception: %s", e.what());
256     return;
257 }
258 }
259
260 int main(int argc, char **argv)
261 {
262     // Inicializa o do Node
263     ros::init(argc, argv, "cv_node");
264     cvProcess corretor;
265     ros::spin();
266
267 }
```

Parâmetros utilizados no slam-gmapping (*default*):

- inverted\_laser (string, default: "false")
- throttle\_scans (int, default: 1)
- base\_frame (string, default: "base\_link")
- map\_frame (string, default: "map")
- odom\_frame (string, default: "odom")
- map\_update\_interval (float, default: 5.0)
- maxUrange (float, default: 80.0)
- sigma (float, default: 0.05)
- kernelSize (int, default: 1)

- lstep (float, default: 0.05)
- astep (float, default: 0.05)
- iterations (int, default: 5)
- lsigma (float, default: 0.075)
- ogain (float, default: 3.0)
- lskip (int, default: 0)
- minimumScore (float, default: 0.0)
- srr (float, default: 0.1)
- srt (float, default: 0.2)
- str (float, default: 0.1)
- stt (float, default: 0.2)
- linearUpdate (float, default: 1.0)
- angularUpdate (float, default: 0.5)
- temporalUpdate (float, default: -1.0)
- resampleThreshold (float, default: 0.5)
- particles (int, default: 30)
- xmin (float, default: -100.0)
- ymin (float, default: -100.0)
- xmax (float, default: 100.0)
- ymax (float, default: 100.0)
- delta (float, default: 0.05)
- ll samplerange (float, default: 0.01)
- ll samplestep (float, default: 0.01)
- la samplerange (float, default: 0.005)
- la samplestep (float, default: 0.005)
- transform\_publish\_period (float, default: 0.05)

- occ\_thresh (float, default: 0.25)
- maxRange (float)