UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
ESCOLA DE ENGENHARIA
ENG. DE CONTROLE E AUTOMAÇÃO

GABRIEL RODRIGUES PETRY

NAVEGAÇÃO DE UM ROBÔ MÓVEL EM AMBIENTE SEMI-ESTRUTURADO

Porto Alegre
2019

GABRIEL RODRIGUES PETRY

NAVEGAÇÃO DE UM ROBÔ MÓVEL EM AMBIENTE SEMI-ESTRUTURADO

Trabalho de Conclusão de Curso (TCC-CCA) apresentado à COMGRAD-CCA da Universidade Federal do Rio Grande do Sul como parte dos requisitos para a obtenção do título de *Bacharel em Eng. de Controle e Automação* .

Orientador:

Prof. Dr. Walter Fetter Lages

Porto Alegre
2019

GABRIEL RODRIGUES PETRY

NAVEGAÇÃO DE UM ROBÔ MÓVEL EM AMBIENTE SEMI-ESTRUTURADO

Este Trabalho de Conclusão de Curso foi julgado adequado para a obtenção dos créditos da Disciplina de TCC do curso *Eng. de Controle e Automação* e aprovado em sua forma final pelo Orientador e pela Banca Examinadora.

Orientador: _____
Prof. Dr. Walter Fetter Lages, UFRGS
Doutor pelo Instituto Tecnológico de Aeronáutica – São José dos Campos, Brasil)

Banca Examinadora:

Prof. Dr. Walter Fetter Lages, UFRGS
Doutor pelo Instituto Tecnológico de Aeronáutica – São José dos Campos, Brasil)

Profª. Drª. Mariana Luderitz Kolberg, UFRGS
Doutora pela Pontifícia Universidade Católica do Rio Grande do Sul – Porto Alegre, Brasil

Prof. Dr. Alcy Rodolfo dos Santos Carrara, UFRGS
Doutor pela McMaster University – Hamilton, Canadá

_____
Prof. Dr. Marcelo Götz
Coordenador de Curso
Eng. de Controle e Automação

Porto Alegre, dezembro de 2019.

AGRADECIMENTOS

Agradeço aos meus pais e aos meus amigos por todo o apoio dado durante todo o período da minha graduação.

Ao meu orientador, Prof. Dr. Walter Fetter Lages, pelo auxílio e pelos conselhos dados ao longo da execução desse trabalho.

À Universidade Federal do Rio Grande do Sul pela oportunidade de realizar meus estudos de forma gratuita em uma instituição de excelência.

À Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) e à Escola de Engenharia ENSTA Paris pela oportunidade de realização do intercâmbio de duplo diploma na França, que foi uma experiência fundamental e transformadora tanto para minha vida acadêmica e profissional quanto para meu crescimento pessoal.

RESUMO

Este trabalho aborda a implementação de um sistema de navegação autônoma para um robô móvel a partir das ferramentas propostas pelo ROS. Primeiramente, o modelo do robô estudado foi obtido e adaptado para a tarefa proposta, incluindo a adição de sensores. Em seguida, trabalhou-se com a planta de um prédio existente nas dependências UFRGS para obter um mapa para a navegação. O sistema de navegação proposto no ROS foi estudado e configurado para ser então integrado ao projeto. Foram executadas simulações para avaliar o desempenho do sistema e ajustar os parâmetros necessários para garantir seu funcionamento. Ao final, obteve-se um sistema capaz de planejar e executar trajetórias para que o robô desloque-se de forma autônoma para posições arbitrárias dentro do ambiente utilizado.

Palavras-chave: Navegação autônoma, ROS, robôs móveis.

ABSTRACT

This project approaches the implementation of an autonomous navigation system for a mobile robot using tools proposed by ROS. Firsty, the model of the robot that was studied in this project was obtained and adapted for the proposed task, including the addition of sensors. Then, the floorplan of one of the building from UFRGS was used to obtain a map for the navigation to be based on. The navigation system proposed by ROS was studied and configured in order to be integrated to the project. Simulations were executed to evaluate the performance of the system and to adjust the parameters needed to ensure its correct operation. In the end, the designed system was capable of planning and executing trajectories allowing the robot to move autonomously to arbitrary positions in its environment.

Keywords: Autonomous navigation, ROS, mobile robots.

SUMÁRIO

LISTA DE ILUSTRAÇÕES

LISTA DE ABREVIATURAS

AGV	*Autonomous Guided Vehicle* (veículo guiado automaticamente)
AMCL	*Adaptive Monte Carlo Localization* (localização adaptativa de Monte Carlo)
LIDAR	*Light Detection and Ranging* (detecção de luz e alcance)
ODE	*Open Dynamics Engine* (mecanismo aberto de dinâmica)
OGRE	*Object-oriented Graphics Rendering Engine* (mecanismo de renderização gráfica orientada a objetos)
OSRF	*Open Source Robotics Foundation* (fundação de código aberto de robótica)
ROS	*Robot Operating System* (sistema operacional de robôs)
SLAM	*Simultaneous Localization and Mapping* (mapeamento e localização simultâneos)
UFRGS	Universidade Federal do Rio Grande do Sul

1 INTRODUÇÃO

1.1 Robôs móveis

Há décadas que o campo da robótica encontra sucesso no mundo da manufatura industrial, devido principalmente aos manipuladores robóticos, os quais executam uma grande variedade de tarefas, como pintura, solda e operações de montagem, com precisão e repetibilidade. No entanto, esse tipo de robô possui uma limitação importante: sua falta de mobilidade, que faz com que seu alcance dependa de como ele for fixado. Os robôs móveis, ao contrário, não possuem tal limitação, podendo se locomover para executar tarefas em diferentes pontos de uma fábrica (SIEGWART; NOURBAKHSH; SCARAMUZZA, 2011).

Um robô móvel pode ser definido como uma máquina com capacidades de percepção, decisão e ação, que permitem que ele aja de maneira autônoma em seu ambiente baseando-se na sua percepção do mesmo (FILLIAT, 2016). O desenvolvimento de robôs móveis envolve áreas do conhecimento diversas, como mecânica, eletrônica, automação, computação e inteligência artificial, cujos elementos devem ser integrados para garantir seu funcionamento.

Algumas aplicações de robôs móveis incluem a exploração de regiões inóspitas e inacessíveis ao ser humano, como a superfície de Marte e os oceanos; limpeza de ambientes, como no caso dos robôs aspiradores, já bastante popularizados; e transporte de peças e produtos entre estações de manufatura, tarefa para a qual as indústrias modernas já utilizam robôs do tipo AGV (*autonomous guided vehicle*) (SIEGWART; NOURBAKHSH; SCARAMUZZA, 2011).

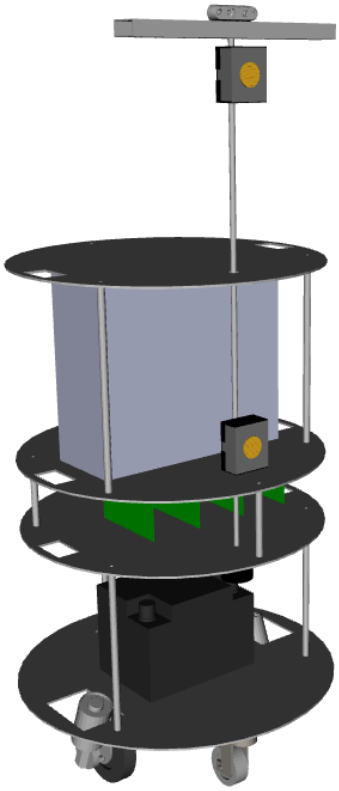
No contexto da Universidade Federal do Rio Grande do Sul (UFRGS), existem trabalhos tratando de diversos aspectos no campo da robótica móvel. A dissertação de mestrado de Barros (2014), por exemplo, aborda a arquitetura de software, simulação e sistema de controle de um robô móvel desenvolvido na universidade e chamado de Twil, que será o robô usado no presente projeto para o desenvolvimento de um sistema de navegação autônoma.

1.2 Objetivos

O modelo tridimensional do robô que será estudado nesse projeto é mostrado na Figura 1. Esse modelo foi concebido em trabalhos anteriores para simular um protótipo construído nas dependências da UFRGS, permitindo que novas funcionalidades sejam desenvolvidas e testadas por meio de simulações antes de serem integradas ao robô real. Partindo-se desse contexto, a motivação do presente projeto é ampliar o leque de atuação do robô Twil através do desenvolvimento de um sistema de navegação autônoma a partir do modelo existente.

Assim, os objetivos deste trabalho podem ser resumidos nos seguintes pontos: desenvolver um projeto com base em uma arquitetura de software existente projetada para o robô Twil; compreender os conceitos e as ferramentas necessárias para implementar um sistema de navegação autônoma no contexto estudado; obter um sistema capaz de navegar em um ambiente pré-definido na presença de obstáculos não previstos, ou seja, em um ambiente semi-estruturado; e, por último, utilizar o sistema desenvolvido para realizar a navegação dentro do Prédio Centenário da Escola de Engenharia da UFRGS, primeiramente por meio de simulação mas visando a realização de uma demonstração real futuramente.

Figura 1: Renderização em 3D do robô Twil.



Fonte: Autor

1.3 Organização do trabalho

Visando atingir os objetivos definidos acima, primeiramente os conceitos de navegação em robótica e as ferramentas utilizadas no trabalho serão descritos brevemente no capítulo de revisão da literatura. Em seguida, a metodologia aplicada será descrita em detalhe, com foco na arquitetura de software implementada e na integração de todos os componentes necessários para a navegação. Finalmente, no capítulo dedicado aos resultados, serão apresentados os experimentos feitos com o sistema desenvolvido e as performances obtidas, incluindo discussões sobre a influência dos parâmetros ajustados e as limitações dos resultados obtidos até esse ponto.

2 REVISÃO DA LITERATURA

2.1 Navegação de robôs

No seu sentido original, o termo navegação se aplica ao processo de levar um navio para o seu destino. Este processo consiste em três repetir três passos sucessivamente: primeiro o navegador determina a posição do navio em uma carta náutica da forma mais precisa possível; na carta, ele relaciona sua posição em relação ao destino, pontos de referência e possíveis obstáculos; finalmente, baseando-se nessas informações, ele define o curso da embarcação. Trazendo para o domínio da robótica, esta definição vinda da atividade náutica pode ser aplicada praticamente sem alterações (FRANZ; MALLOT, 2000).

Em Levitt (1990), navegação é definida como um processo que responde três questões fundamentais: (a) Onde estou? (b) Onde estão outros lugares em relação a mim? (c) Como posso chegar em outros lugares a partir daqui? Já para Trullier, Wiener et al. (1997), a palavra navegação se refere a todas as estratégias que possam ser empregadas por um robô com o propósito de mover-se em seus arredores. Estas estratégias vão desde o simples comportamento de se mover em direção a um objetivo visível até uma navegação complexa baseada em mapas, permitindo planejar-se movimentos até objetivos arbitrariamente distantes.

Esse segundo tipo de estratégia implica três sub-problemas: o aprendizado de um mapa, que se refere a construir um mapa que represente os arredores do robô; localização, que consiste em estimar a posição do robô no mapa; e planejamento, definida como a tarefa de conceber um plano para atingir um dado objetivo (FILLIAT; MEYER, 2002).

2.1.1 Tipos de navegação

As técnicas de navegação que permitem a um robô de se deslocar para atingir um objetivo são extremamente diversas, assim como as formas de classificá-las. Em Trullier e Meyer (1997), uma classificação inspirada nas estratégias de navegação usadas pelos animais é proposta, tendo a vantagem de distinguir as estratégias com e sem modelo interno (FILLIAT, 2016). Essa classificação comporta cinco categorias, que são, da mais simples até a mais complexa:

- Aproximação de um objeto: capacidade de se dirigir em direção a um objeto visível a partir da posição atual do robô. É uma estratégia local, ou seja, funciona unicamente na zona em que o objetivo é visível.

- Guiagem: capacidade de seguir certas referências presentes no ambiente a fim de atingir um objetivo. Também é uma estratégia local que não necessita um modelo interno do ambiente.

- Resposta associada a um local: é a primeira técnica que permite uma navegação global, i.e. que permite atingir-se um objetivo a partir de posições nas quais nem este objetivo nem referências ligadas a ele são visíveis. Requer uma representação interna do ambiente que consiste em definir zonas do espaço nas quais as percepções são similares umas às outras, associando-se ações a cada um destes lugares. A sequência dessas ações define uma rota que permite atingir-se um objetivo.

- Navegação topológica: extensão da técnica precedente que armazena no seu modelo interno as relações espaciais entre diferentes locais e a possibilidade de se deslocar entre eles, mas dessa vez sem que estejam associadas a um objetivo específico. Dessa forma, o modelo interno é um grafo que permite calcular diferentes caminhos para se deslocar entre dois locais arbitrários.

- Navegação métrica: extensão da estratégia anterior que torna possível para o robô planejar caminhos passando por zonas inexploradas do seu ambiente. Para isso, além da possibilidade de se deslocar entre diferentes locais, memoriza-se as suas posições métricas, as quais permitem o cálculo de um trajetória através de uma composição vetorial.

Em Franz e Mallot (2000), uma classificação similar é apresentada, porém com duas formas suplementares de navegação local: procura, que consiste em procurar um objetivo movendo-se aleatoriamente; e seguimento de direção, que consiste em manter uma direção específica de movimento que levará até um objetivo, como um navio seguindo um curso previamente definido.

2.1.2 Fontes de informação

Diversos tipos de sensores, tais como sonares, odômetros, lasers, unidades de medição inercial (IMU, em inglês), GPS (*global positioning system*) e câmeras são usados para tornar um robô capaz compreender os mais variados tipos de ambientes (ZAMAN; SLANY; STEINBAUER, 2011). Todos os sensores utilizados em robótica móvel, muitos dos quais são detalhados em Borenstein et al. (1997), fornecem informações de uma de duas categorias: informações proprioceptivas ou exteroceptivas (FILLIAT, 2016).

Proprioceptivas são informações internas ao robô que, no caso da navegação, se referem ao seu deslocamento no espaço. Podem se originar na medida da rotação de suas rotas (no caso de um odômetro) ou da sua aceleração por meio de uma IMU. Acumulando-se essas medidas ao longo do tempo é possível executar um processo de integração a fim de estimar o deslocamento efetuado pelo robô.

Já as informações exteroceptivas, também chamadas de percepções, são informações que o robô adquire de seu entorno, podendo ter naturezas extremamente distintas. Pode-se, por exemplo, medir a distância dos obstáculos através de sensores de infravermelho ou utilizando uma câmera.

2.2 ROS - Robot Operating System

O ROS é um *framework* para desenvolvimento de software para robôs. É constituído por uma coleção de ferramentas, bibliotecas e convenções que buscam simplificar a tarefa de criar comportamentos complexos e robustos para robôs em diversas plataformas robóticas (QUIGLEY et al., 2009). Caracteriza-se por ser um sistema modular no qual vários programas trabalham em conjunto se comunicando através de mensagens.

A unidade fundamental é o nodo, que é um programa executável que recupera dados dos sensores do robô e os transmite para outros nodos na forma de mensagens. Estas mensagens são transmitidas na forma de tópicos, que seguem modelos de formatação padronizados de acordo com o tipo de mensagem que eles encapsulam.

Um nodo que envia mensagens em um tópico é chamado de *publisher*, enquanto que o nodo que as recebe tem que se inscrever no tópico, recebendo por esse motivo o nome de *subscriber*. Nodos relacionados entre si são combinados em um pacote, que é representado por um arquivo XML, podendo ser facilmente compilado e transportado para outros computadores. Os pacotes são necessários para construir-se um sistema completo de controle baseado em ROS para um robô autônomo (ZAMAN; SLANY; STEINBAUER, 2011).

Sendo um projeto de código aberto em constante desenvolvimento, existem várias versões do ROS disponíveis, que são atualizadas frequentemente pelos seus desenvolvedores, sendo a mais recente versão estável e com suporte denominada Melodic, lançada em Maio de 2018. Ainda é possível encontrar projetos paralelos mantidos pela comunidade que agregam funcionalidades especificas ou pacotes para trabalhar-se com robôs, sensores e atuadores específicos. Destacam-se aqui as ferramentas de visualização, como `rviz` e `rqt`, que permitem gerar representações gráficas de diversos elementos do ROS, como dados de sensores, posição e movimentação do robô, gráficos dos nodos e tópicos em execução, etc., auxiliando assim o trabalho dos desenvolvedores e usuários do sistema.

2.2.1 Gazebo

O simulador Gazebo é uma ferramenta comumente utilizada em projetos com ROS, porém que funciona de forma independente do mesmo. Inicialmente desenvolvido por Koenig e Howard (2004), hoje ele é mantido pela *Open Source Robotics Foundation (OSRF)*, assim como o ROS.

Esse software permite a realização de simulações dinâmicas multi-robôs em ambientes 3D, baseando-se em sistemas de física como o ODE (*open dynamics engine*) para representar o funcionamento dinâmico e no sistema de renderização gráfica *open source* OGRE (*objected-oriented graphics rendering engine*) para renderizar os ambientes (OSRF, 2014) (KOENIG; HOWARD, 2004). Os modelos no Gazebo utilizam o formato SDF, um tipo de XML desenvolvido juntamente com ele e que descreve objetos e ambientes para simulação, visualização e controle de robôs (OSRF, 2019).

Dessa forma, utilizando-se conjuntamente ROS e Gazebo é possível desenvolver por exemplo sistemas de controle ou navegação para robôs autônomos e ao mesmo tempo testar seu funcionamento a partir de um modelo que simula seu comportamento dinâmico. O uso do simulador é fundamental durante o desenvolvimento pois consiste em uma forma rápida de se executar testes sem o envolvimento de robôs e ambientes reais.

2.2.2 ROS navigation stack

Diversos recursos presentes no ROS podem auxiliar um robô a se localizar, planejar e seguir uma trajetória, desviar de obstáculos e mesmo mapear um ambiente. Estes recursos estão agrupados em um conjunto conhecido como *navigation stack*. Alguns dos elementos desse conjunto são: o planejador de trajetória global, que traça a trajetória de menor custo entre a posição atual do robô e seu objetivo; o planejador local, que adapta essa trajetória aos obstáculos presentes nas proximidades; os mapas de custo (*costmap*) global e local, que representam as áreas seguras do ambiente pelas quais o robô pode passar sem colidir com outros objetos, e um sistema de localização (GUIMARÃES et al., 2016).

O planejador global utilizado é baseado na abordagem de janela dinâmica global proposta em Brock e Khatib (1999), que por sua vez é uma generalização da abordagem de janela dinâmica descrita em Fox, Burgard e Thrun (1997) e empregada no planejador local. Dado um mapa conhecido do ambiente, o sistema de localização utilizado é um pacote do ROS chamado de `amcl`, abreviação em inglês de Localização Adaptativa de Monte Carlo, que consiste em uma adaptação do algoritmo de Monte Carlo apresentado em Dellaert et al. (1999).

A fim de maximizar a performance do sistema de navegação do ROS, uma série de parâmetros podem ser ajustados em cada elemento do sistema. Uma descrição destes parâmetros e da sua influência é dada em Zheng (2016).

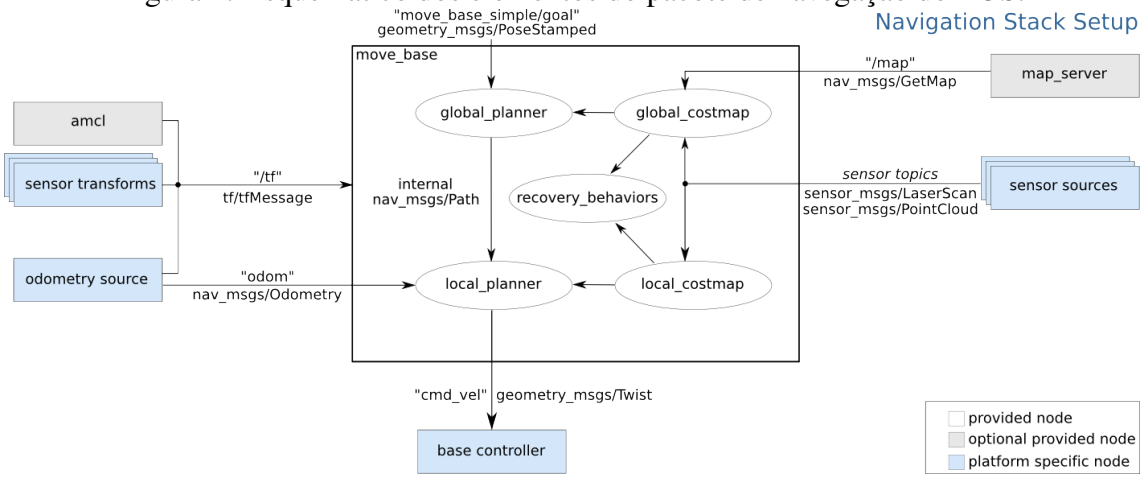
3 METODOLOGIA

Neste capítulo são apresentados os métodos utilizados e a descrição do trabalho executado. Para isso, um apanhado dos componentes necessários para implementar o sistema de navegação proposto é apresentado; após, cada um desses componentes é abordado individualmente; e, por fim, o procedimento de integração desses elementos ao redor do *stack* de navegação é discutido.

3.1 Panorama do trabalho

Uma forma de ter-se uma visão geral do *stack* de navegação do ROS é através da representação esquemática de seus componentes, tal como mostra a Figura 2. A partir dessa imagem, pode-se, notadamente, diferenciar as entradas, saídas e componentes internos do sistema.

Figura 2: Esquemático dos elementos do pacote de navegação do ROS.



Fonte: ROSwiki (2018)

Quanto às entradas, o tópico `/tf` representa a estrutura dimensional do robô na forma das transformações de coordenadas entre seus diferentes elos; `odom` consiste em um sinal de odometria, ou seja, uma medida do deslocamento do robô calculada a partir do giro de suas rodas, que, como visto na seção 2.1.2, trata-se de uma informação proprioceptiva; `/map` representa o mapa dentro do qual o robô está navegando; `sensor topics` são as informações exteroceptivas usadas na navegação, podendo ser do tipo `sensor_msgs/LaserScan` ou `sensor_msgs/PointCloud`; por último, a entrada `move_`

`base_simple/goal` é o objetivo, a referência de posição para a qual se deseja navegar de forma autônoma.

Os elementos centrais da Figura 2 são os fornecidos pelo pacote de navegação, os quais foram discutidos na seção 2.2.2. Vê-se aqui de que forma os planejadores de trajetória global e local atuam - baseando-se nos mapas de custo - de forma a produzir uma saída na forma de um comando de velocidade para o robô, representado pela saída `cmd_vel`.

O primeiro objetivo do projeto é integrar os pacotes existentes do Twil com esse sistema. Os tópicos `/tf` e `odom`, duas das entradas do sistema de navegação, representam informações internas do robô e já são publicados, restando apenas conectá-los. O mesmo é válido para o comando de velocidade resultante da navegação. O sistema de controle desenvolvido para o Twil faz com que o robô já possa receber uma entrada dessa natureza e executá-la, restando efetuar a conexão e testar o seu bom funcionamento.

Em relação ao mapa, como foi dito na introdução, o plano é realizar-se a navegação dentro do prédio Centenário da Escola de Engenharia da UFRGS. Para isso, obteve-se a planta desse prédio em formato.dwg, a qual serviu como ponto de partida para a concepção do mapa de acordo com os critérios do *stack* de navegação.

Já quanto aos dados dos sensores, como o *stack* de navegação necessita mensagens do tipo `sensor_msgs/LaserScan` ou `sensor_msgs/PointCloud`, que são geradas por um sensor LIDAR e uma câmera 3D, respectivamente, é preciso integrar um desses dois tipos de equipamento ao projeto.

Câmeras 3D são sensores comumente compostos por duas câmeras convencionais com uma certa distância entre elas permitindo calcular-se a profundidade dos objetos presentes efetuando-se uma triangulação entre as imagens das duas. Estas já são bastante comuns em ambientes acadêmicos e até em aplicações domésticas, como no caso do Microsoft Kinect. Já os sensores LIDAR são equipamentos mais sofisticados, os quais emitem diversos feixes de laser (comumente 32 ou 64) de forma a escanear o ambiente dentro de uma determinada faixa de alcance, obtendo assim seu modelo tridimensional.

Para o escopo desse projeto decidiu-se utilizar uma câmera 3D. Essa escolha se justifica pelo fato de que este é um sensor consideravelmente mais barato do que o LIDAR e já existem exemplares disponíveis no laboratório, facilitando assim a passagem da simulação para o sistema real. Assim, deve-se buscar um *plugin* para ROS que simule uma câmera 3D e adicioná-lo ao modelo do robô.

Após a integração dos pacotes do Twil, criação do mapa para a simulação e adição dos sensores, o passo seguinte é testar o planejamento de trajetórias e ajustar os parâmetros dos algoritmos a fim de avaliar os desempenhos resultantes.

3.2 Arquitetura de software do robô Twil

Conforme discutido na introdução, o robô móvel que será utilizado possui um modelo na forma de um *stack* do ROS, composto de pacotes representando diversos elementos do robô, como seu modelo no formato SDF utilizado no Gazebo, os controladores responsáveis pela sua movimentação de acordo com os comandos de velocidade fornecidos e a inicialização do sistema.

Notadamente, o pacote `twil_description` contém a descrição dos diversos componentes que compõem o robô Twil, como seu chassis, as rodas motoras e as rodas auxiliares. Ele contém, ainda, o arquivo de inicialização, também chamado de *launchfile*, responsável por inicializar o modelo do robô no Gazebo. Já o pacote `twil_bringup` pos-

sui os arquivos responsáveis pela inicialização do projeto, incluindo um *launchfile* global que é responsável por lançar os nodos referentes ao estado do robô e também os *launchfiles* específicos de descrição e de controle. Posteriormente, esse arquivo foi alterado para incluir também a publicação do tópico `/tf` e a execução da visualização no `rviz`.

A fim de reproduzir o funcionamento do modelo do robô com seu controle de velocidade, primeiramente foi preciso instalar os pacotes de base do ROS, especificamente da versão Melodic. Após, os arquivos do projeto foram obtidos a partir do diretório git do projeto, criando-se um novo *branch* para sincronizar as futuras modificações no servidor da UFRGS.

No escopo do ROS, a compilação do projeto é executada através do comando de terminal `catkin_make`. Após, deve-se efetuar o comando `source devel/setup.bash` a partir do diretório inicial do projeto a fim de configurar o ambiente do ROS para que ele aponte para o projeto corrente.

A partir desse ponto é possível executar o modelo, o que é feito com o comando `roslaunch [nome do pacote] [launchfile] [parâmetros]`, onde o pacote a ser indicado é o que contém o *launchfile* desejado. No caso deste projeto, os parâmetros definidos no *launchfile* global contido no pacote `twil_bringup` incluem o nome do controlador a ser usado e o caminho do seu arquivo de configuração, além de variáveis boolianas que configuram a execução da simulação. Caso estes parâmetros não sejam especificados no comando de inicialização, os valores definidos como *default* são utilizados.

Após inicializar o modelo, o robô se mantém estático até que um comando de velocidade seja recebido no tópico `/twist_mrac_linearizing_controller/command`, onde `twist_mrac_linearizing_controller` é o nome do controlador utilizado. Este controlador espera um comando do tipo `geometry_msgs/Twist`, que é idêntico ao comando gerado pelo sistema de navegação, mostrado na Figura 2. As mensagens desse tipo são formadas por dois vetores, representando, respectivamente, as velocidades linear e angular. Ademais, cada um desses vetores é composto por três valores (x, y e z), que por sua vez são números em ponto flutuante no formato float64.

No contexto do Twil, assim como em outros robôs móveis terrestres, somente a componente x da velocidade linear e a componente z da velocidade angular são utilizadas, logo as demais são ignoradas e podem ser omitidas ao executar-se um comando. Isso ocorre pois essas duas componentes já são suficientes para especificar a magnitude e direção de um movimento no plano xy, sendo utilizadas para calcular a velocidade a ser transmitida para cada uma das duas rodas motoras do robô.

Com isso, pode-se testar o funcionamento do robô antes da integração do sistema de navegação publicando-se mensagens de velocidade diretamente através do terminal do computador, o que é feito com o comando `rostopic pub [nome do tópico] [tipo de mensagem] [mensagem]`. Os resultados da execução do modelo e destes testes preliminares são apresentados no capítulo 4.

3.3 Integração da câmera de profundidade

Conforme explicado anteriormente, o *stack* de navegação necessita dados provenientes de um sensor para ser implementado, sendo que nesse projeto optou-se por utilizar uma câmera 3D. O modelo específico escolhido foi a câmera Intel RealSense D435 devido a sua disponibilidade em laboratório, o que facilita a futura integração do sistema desenvolvido no robô real.

No repositório Github dessa câmera (INTEL REALSENSE, 2019b) existem pacotes de-

senvolvidos para tornar possível sua utilização dentro do ROS, incluindo um que implementa a sua descrição no formato SDF. Após instalado, esse pacote permitiu integrar-se uma versão simulada da RealSense D435 ao modelo físico do robô Twil, o que foi feito incluindo à descrição da câmera nos componentes do robô e posicionando-a no topo, de forma a possibilitar uma visão desobstruída do ambiente.

Para a simulação, além de adicionar-se o modelo físico da câmera a ser incorporado ao robô, é preciso relacioná-lo a um componente do tipo sensor, que indica que o objeto em questão representa de fato uma câmera, ou seja, que ele é capaz de gerar mensagens como tal. Além disso, o sensor deve conter um *plugin* para ROS a fim de produzir efetivamente esse comportamento, gerando o mesmo tipo de mensagem que a câmera real produz para que a navegação possa ser simulada.

A documentação do Gazebo propõe um *plugin* desse tipo, o qual foi desenvolvido originalmente para o Microsoft Kinect mas que com o tempo tornou-se referência para implementar também outros modelos de câmeras de profundidade no escopo do ROS (OPEN SOURCE ROBOTICS FOUNDATION, 2014). Notadamente, esse *plugin* faz com que as mensagens do tipo `sensor_msgs/PointCloud`, que são as usadas na navegação, sejam produzidas durante a simulação.

Estas mensagens são compostas por pontos tridimensionais, representando as coordenadas x e y de cada ponto e também sua distância z em relação à câmera. O número de pontos é igual ao número de pixels que compõem as imagens geradas pela câmera. Dessa forma, as imagens do tipo `sensor_msgs/PointCloud` são usadas como uma medição da profundidade das imagens captadas pela câmera, sendo capazes de informar a distância da câmera em relação aos objetos presentes no seu campo de visão, o que justifica sua utilização na navegação no contexto da detecção de obstáculos.

O código usado para adicionar e configurar o sensor do tipo câmera de profundidade e o *plugin* correspondente possui diversos pontos personalizáveis, como os nomes dos tópicos que serão publicados e o nome do *frame* do modelo físico do robô que corresponde à câmera. Estes nomes foram conservados da forma original no código, pois eles correspondem ao padrão comumente utilizado em pacotes ROS, facilitando a troca entre a câmera simulada e o equipamento real.

Pode-se também modificar a taxa de atualização das mensagens geradas tanto no bloco do sensor quanto no interior da configuração do *plugin*. Contudo, esse segundo deve ser deixado com o valor original nulo para que seja usada a mesma taxa do sensor, a fim de não causar-se um estrangulamento adicional. Já os parâmetros de distorção do *plugin* devem ser idênticos aos presentes no sensor, tendo valor zero caso ele não possua estes valores. Os parâmetros `PointCloudCutoff` e `PointCloudCutoffMax` representam, respectivamente, as distâncias mínimas e máximas que podem ser representadas pelos pontos, o que deve ser ajustado considerando-se as limitações da câmera real, as quais estão descritas no seu *datasheet* (INTEL REALSENSE, 2019a).

Além disso, pode-se citar os parâmetros de taxa de atualização (`update_rate`), em imagens por segundo; campo de visão horizontal (`horizontal_fov`), em radianos; resolução da imagem (`width` e `height`); alcance mínimo e máximo em metros e ruído, os quais estão presentes no código do sensor e também podem ser ajustados de acordo com o *datasheet* a fim de melhor representar o componente real.

Após completar-se a integração da câmera de profundidade ao projeto, é necessário verificar seu bom funcionamento no escopo da simulação, o que pode ser feito inicializando a simulação no Gazebo como anteriormente e verificando se o modelo físico da câmera está de fato presente e se os dados de profundidade estão sendo gerados de forma

adequada. O resultado destes testes será apresentado no capítulo 4.

3.4 Construção do mapa

Em relação ao mapeamento, duas opções foram consideradas: partir-se de um ambiente 3D simulado no Gazebo e executar o mapeamento por SLAM como proposto pelo *stack* de navegação ou utilizar-se um mapa pronto no formato compatível com o *stack*. Foi decidido que primeiramente seria implementado o segundo método, pois a execução do SLAM, sendo uma etapa complementar, introduz mais complexidade à tarefa de navegação. Já utilizando-se um mapa gerado manualmente, garante-se que não ocorrerão erros na etapa de mapeamento, facilitando assim o teste do procedimento de navegação de forma isolada.

Para isso, o modelo em .dwg da planta do primeiro andar do prédio centenário foi primeiramente simplificado de forma a eliminar-se detalhes irrelevantes para a presente aplicação, como elementos externos ao prédio ou que estariam fora do alcance de um robô movimentando-se pelos corredores. A representação das portas e janelas também foi modificada em relação à planta original de forma a representar as portas como estando abertas e as janelas como estando fechadas. Isso foi feito dessa forma pois, como o robô não é capaz de abrir portas, elas devem estar previamente abertas para que o caminho esteja disponível. Já as janelas foram representadas como estando fechadas para desocupar o espaço correspondente às suas folhas. Em uma aplicação real, caso uma janela estiver aberta e sua folha estiver no caminho do robô, esta será interpretada como um obstáculo a ser evitado.

No ROS, os mapas são geridos por um nodo chamado `map_server`, que interpreta uma imagem de forma a gerar uma mensagem do tipo `nav_msgs/GetMap`, que é publicada no tópico `/map`. Para isso, é preciso fornecer uma imagem *bitmap*, além de uma série de parâmetros que são definidos em um arquivo de configuração. Esses parâmetros indicam a resolução do mapa em metros por pixel, a posição da sua origem em relação às coordenadas do sistema e os valores limite de probabilidade de ocupação dos pixels a partir dos quais eles serão considerados como livres ou ocupados.

Os valores de probabilidade de ocupação são definidos da seguinte forma: $p = (255 - x)/255.0$, sendo p probabilidade e x a cor do pixel em uma escala de 0 (preto) a 255 (branco). Dessa forma, um pixel preto corresponde a $p = 1$, enquanto que um branco é representado por $p = 0$. Definindo-se, por exemplo, o limite livre como $p = 0,3$ significa que qualquer pixel que tenha p abaixo desse valor será considerado como livre, e analogamente para o limite ocupado. Os pixels cujo valor de p esteja entre os dois limites são considerados como desconhecidos pelo `map_server`, sendo à princípio interpretados como livres no contexto da navegação.

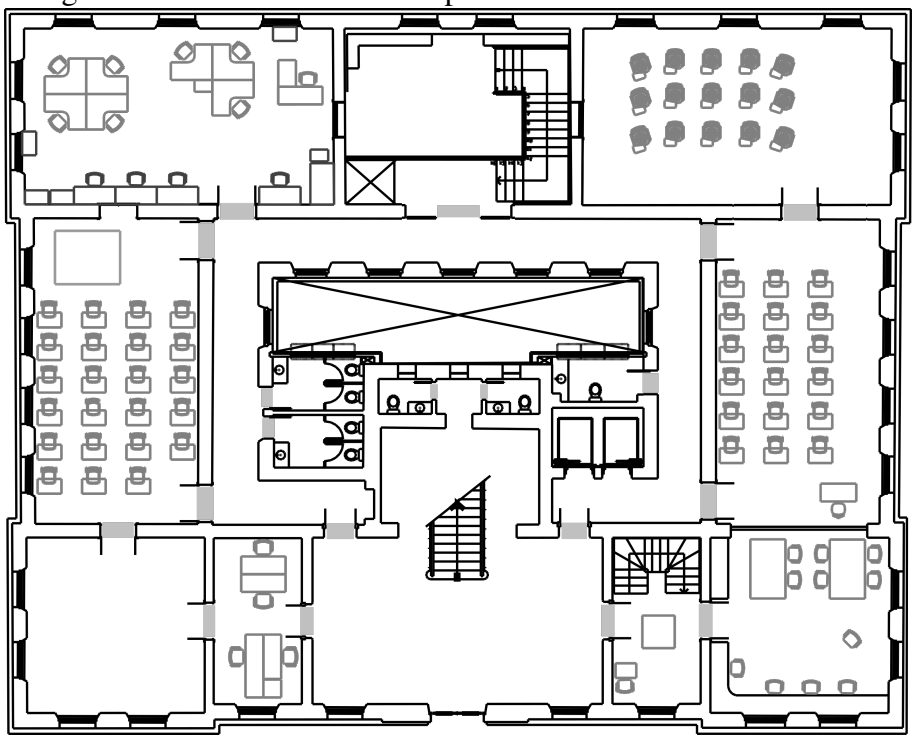
A possibilidade de ajustar estes limites e também de manipular a cor dos objetos na planta tornou possível criar-se um sistema de cores de forma a representar de forma diferente objetos cuja presença é dada como certa e outros que possuem diferentes níveis de incerteza. Dessa forma, código de cores concebido foi o seguinte:

- 0 (preto) - ocupado. Por exemplo: paredes, escadas e portas;
- 64 - provavelmente ocupado. Exemplo: mesas fixas junto às paredes;
- 128 - desconhecido. Exemplo: carteiras escolares, cuja posição no ambiente pode variar;

- 192 - provavelmente livre. Esse valor foi usado para as aberturas das portas, as quais dependendo da situação pode ser mais conveniente considerar como desconhecido ou como livre;
- 255 (branco) - livre. Representa os espaços vazios na planta.

A Figura 3 mostra a planta do 1º do prédio centenário da escola de engenharia após a execução das manipulações descritas e aplicação do código de cores desenvolvido. Em um primeiro momento, decidiu-se usar um limite livre de $p = 0,3$, o que significa que os pixels de valor 192 e 255 da planta são considerados como livres. Ou seja, assume-se que as portas se encontram abertas. Já o limite ocupado foi definido em $p = 0,7$, para que tanto os pixels de valor 0 quanto os de valor 64 sejam considerados como estando ocupados. Assim, os pontos de valor 128 se encontram entre os dois limites e são interpretados como desconhecidos.

Figura 3: Planta do 1º andar do prédio Centenário da EE-UFRGS.



Fonte: Autor

Até esse ponto, a planta estava sendo manipulada em um formato vetorial, podendo ser dimensionada livremente sem perder-se resolução. Contudo, como o `map_server` requer um arquivo *bitmap*, foi preciso exportá-la a partir do programa de manipulação vetorial utilizado, definindo assim uma resolução, a qual deve ser indicada no arquivo de configuração do mapa. Essa resolução deve ser suficiente para que um pixel não represente uma distância significativa em relação às dimensões do robô e dos obstáculos de navegação, de forma a não interferir no seu funcionamento. Por exemplo, uma abertura de porta tendo uma largura de aproximadamente 1 metro, enquanto que o robô tem um diâmetro de 0,6 metros, o que significa que restará somente 0,2 m de cada lado do robô no momento da passagem. Assim, uma resolução de 1 m/pixel estaria acima da ordem de grandeza necessária, enquanto que 0,1 m/pixel está na mesma ordem de grandeza mas é

demasiado próximo. Por isso, escolheu-se uma resolução de 0,01 m/pixel, ou seja, cada pixel na imagem da planta representando 1 centímetro do ambiente real, o que foi julgado como sendo suficiente para essa aplicação.

Tendo finalizado todos os procedimentos descritos, pode-se então usar o arquivo de configuração criado para executar o `map_server`, que está contido no pacote homônimo, o qual faz parte da instalação do ROS. Com isso, é possível verificar no `rviz` se a mensagem que contém o mapa está efetivamente sendo publicada no tópico adequado, o que é fundamental para realizar a integração dos elementos usados na navegação.

3.5 Integração dos pacotes de navegação

A integração do sistema de navegação proposto está descrita em ROSwiki (2018) e se dá ao redor do pacote `move_base`, que, como mostrado na Figura 2, recebe informações sobre o estado do robô (`/tf` e `odom`), dados dos sensores (nesse caso, da câmera 3D), além de um mapa e da pose que representa o objetivo da navegação.

O pacote `move_base` faz parte dos elementos presentes na instalação completa da versão Melodic do ROS, portanto não precisa ser adicionado aos pacotes do robô Twil. Sua utilização consiste, então, nos seguintes passos: criar os arquivos de configuração necessários e definir os valores dos parâmetros desses arquivos; garantir que os tópicos utilizados pelo `move_base` estão sendo publicados; por último, escrever um *launchfile* para inicializar a navegação com a configuração definida.

Ora, após concluído o trabalho descrito nas seções anteriores, já se obteve um sistema que atende aos critérios necessários para utilização do `move_base`. Assim, pode-se passar diretamente à definição dos arquivos de configuração para a navegação, os quais foram encapsulados em um novo pacote chamado `twil_2dnav`. Estes arquivos incluem a configuração do mapa usada pelo nodo `map_server`, que foi descrita na seção 3.4.

3.5.1 Parâmetros dos mapas de custo

Os mapas de custo (do inglês *costmap*) são estruturas em forma de grade nas quais cada ponto contém a probabilidade de que aquele espaço esteja ocupado. Eles fornecem informação sobre por onde o robô deve navegar, sendo construídos usando o mapa do ambiente fornecido pelo `map_server` e também dados dos sensores. Existe um mapa de custo global e um local, sendo o primeiro usado para planejar trajetórias a longo prazo sobre o ambiente inteiro, enquanto que o segundo é usado para planejamento local e para evitar obstáculos.

Existe uma série de parâmetros a serem definidos para os mapas de custo, os quais são agrupados em três arquivos de configuração: um comum aos dois mapas, um referente ao mapa global e outro referente ao mapa local. Os parâmetros comuns são mostrados na Listagem 1, e incluem o alcance em metros dentro do qual os obstáculos são detectados e adicionados ao mapa (`obstacle_range`) e a distância ao redor do robô na qual o algoritmo verifica se obstáculos previamente adicionados já não estão presentes, podendo ser eliminados (`raytrace_range`). Os valores mostrados desses parâmetros são os definidos como padrão, sendo necessário ajustá-los posteriormente através de testes com adição de obstáculos.

Ainda é preciso informar a posição das arestas do robô, ou seu raio no caso de um robô circular como o Twil, o qual possui um diâmetro de 60 centímetros. Dessa forma, o parâmetro `robot_radius` deve ser igual a 0.3. Além disso, o parâmetro `inflation_radius` representa a máxima distância de um obstáculo a partir da qual considera-se que não há

mais custo, ou seja, que o caminho encontra-se livre. A superfície entre o obstáculo, o qual possui custo máximo, e esse limite possui um custo decrescente, o que ajuda o robô a navegar a uma distância segura dos obstáculos. Esse valor tem influência significativa no desempenho do planejamento de trajetórias, e os experimentos executados para ajustá-lo são apresentados nos resultados.

Por último, o parâmetro `observation_sources` indica quais sensores estão sendo usados, o que nesse caso corresponde à nuvem de pontos gerada pela câmera 3D. Abaixo, indica-se em qual *frame* do robô este sensor está fixado, o tipo de mensagem produzido, o tópico correspondente, e, finalmente, as variáveis *marking* e *clearing* informam, respectivamente, se os dados do sensor serão usados para marcar obstáculos e para limpá-los do mapa de custo.

Listagem 1: Parâmetros comuns dos mapas de custo

```
obstacle_range: 2.5
raytrace_range: 3.0
robot_radius: 0.3
inflation_radius: 0.35

observation_sources: point_cloud_sensor

point_cloud_sensor: {sensor_frame: camera_link, data_type: PointCloud2,
            topic: /camera/depth/points, marking: true, clearing: true}
```

Já para configurar os mapas de custo global e local, primeiramente é preciso indicar os *frames* de referência global e da base do robô. Há também um parâmetro para indicar se um mapa estático será usado ou não. Como nesse caso efetivamente existe um mapa gerado previamente e que não muda ao longo da simulação, esse parâmetro deve ser verdadeiro.

A configuração do mapa de custo local é mostrada na Listagem 2. Como esse mapa de custo é o responsável por registrar obstáculos percebidos ao longo da navegação, ele deve ser continuamente atualizado. Assim, as frequências de atualização e de publicação das modificações para visualização devem ser indicadas. Essas são representadas na listagem 2 com seus valores padrão, que devem ser modificados conforme o desempenho observado nos testes.

O parâmetro `rolling_window`, por sua vez, indica se o mapa local deve permanecer centrado ao redor do robô conforme esse se move, o que deve ser verdadeiro para atingir-se os fins desejados. A altura e a largura do mapa de custo local também são configuráveis, ao contrário do global que tem o mesmo tamanho do mapa estático, que corresponde à planta mostrada na Figura 3. Já para a resolução, utiliza-se o mesmo valor escolhido para a resolução do mapa na seção anterior.

Listagem 2: Parâmetros do mapa de custo local

```
local_costmap:
    global_frame: odom
    robot_base_frame: twil_origin
    update_frequency: 5.0
    publish_frequency: 2.0
    static_map: true
```

```
rolling_window: true
width: 6.0
height: 6.0
resolution: 0.01
```

3.5.2 Parâmetros do planejador de trajetórias

O planejador de trajetórias é o componente responsável por gerar uma trajetória cinemática ligando a posição inicial do robô ao seu objetivo. Para isso, o algoritmo usa uma função de valor associada a cada posição possível ao redor do robô, que é usada para determinar os comandos de velocidade que serão enviados ao mesmo.

Para garantir que esses comandos de velocidade serão adequados às limitações do robô, é preciso definir uma série de parâmetros de velocidade e aceleração, os quais são mostrados na Listagem 3. As velocidades mínimas linear (`min_vel_x`) e angular (`min_in_place_vel_theta`), por exemplo, devem ser suficientemente elevadas para que o robô consiga superar o atrito estático e colocar-se em movimento, sendo possível descobrir seus valores através de experimentos de simulação em malha aberta, ou seja, enviando diretamente comandos de velocidade para o robô.

Já as velocidades e acelerações máximas são obtidas através dos limites presentes no modelo físico do robô, que indicam que cada uma das duas rodas móveis pode fornecer um torque $T = 100$ Nm e uma velocidade angular de $\omega = 2\pi$ rad. Como o raio dessas rodas é de $7,5$ cm, a velocidade linear máxima é simplesmente $v_{max} = \omega r \simeq 0,47$ m/s. A velocidade angular máxima do robô, por sua vez, ocorre quando as duas rodas estão girando na velocidade máxima e em sentidos contrários, fazendo com que a velocidade de rotação do eixo que as liga seja igual a:

$$\omega_{robô} = \frac{2v_{max}}{d} \tag{1}$$

Onde d corresponde à distância entre as rodas, que no Twil é igual a 32,2 cm. Assim, tem-se que $\omega_{robô} \simeq 2,92$ rad/s.

O parâmetro `escape_vel`, por sua vez, indica a velocidade que será usada pelo robô para andar de ré quando for necessário afastar-se de um obstáculo que esteja bloqueando sua trajetória, sendo assim um valor negativo. Para isso, escolheu-se um valor entre as velocidades mínima e máxima.

Em seguida, os parâmetros de aceleração foram calculados baseando-se no valor do torque máximo e na massa aproximada do robô (30 kg). A partir dessas informações, mais o raio da roda, pode-se calcular a aceleração linear máxima da seguinte forma:

$$a_{max,x} = \frac{T}{mr} \simeq 44,44\ m/s^2 \tag{2}$$

A aceleração máxima angular é calculada de maneira similar, utilizando-se o torque equivalente no eixo que liga as duas rodas quando o torque máximo T é aplicado às duas, em sentidos contrários. No lugar da massa, usa-se aqui o momento de inércia I_z do robô, que pode ser obtido aproximando o mesmo por um cilindro maciço de raio $R \simeq 0,3$ m e altura $h \simeq 1$ m . Dessa forma, tem-se:

$$a_{max,\theta} = \frac{\frac{T}{r}d}{I_z} = \frac{2\frac{T}{r}d}{mR^2} \simeq 79,5\ rad/s^2 \tag{3}$$

A Listagem 3 mostra ainda a existência de dois parâmetros de tolerância em relação ao objetivo da navegação, um em termos do ponto xy desejado (em metros) e outro da orientação (em radianos). Estes dois valores devem ser ajustados durante os testes a fim de otimizar o desempenho da navegação, já que uma tolerância demasiadamente baixa pode fazer com que o robô tenha dificuldade em atingir o objetivo, enquanto que uma tolerância alta faz com que ele possa parar mesmo estando distante da posição desejada.

Listagem 3: Parâmetros do planejador de trajetórias

```
TrajectoryPlannerROS:
  max_vel_x: 0.47
  min_vel_x: 0.2
  max_vel_theta: 2.92
  min_in_place_vel_theta: 1
  escape_vel: -0.3

  acc_lim_theta: 79.5
  acc_lim_x: 44.4

  yaw_goal_tolerance: 0.1
  xy_goal_tolerance: 0.2

controller_frequency: 4.0
```

Por último, o valor da frequência de execução do algoritmo de planejamento é ajustado de acordo com o desempenho do computador durante as simulações, sendo então dependente das condições de realização dos experimentos. Caso a frequência escolhida não esteja sendo atingida devido a limitações de processamento, uma mensagem de aviso será gerada pelo programa.

3.5.3 Execução da navegação

Após concluir a escrita dos arquivos de configuração, tendo ajustado os parâmetros de acordo com a configuração do robô, dos sensores e do mapa, deve-se escrever um *launchfile* a fim de lançar o nodo responsável pela navegação (`move_base`) e também o responsável pela interpretação do mapa (`map_server`) de acordo com as configurações definidas.

Para garantir a correta execução do conjunto, é necessário garantir que todas as entradas do (`move_base`) estejam presentes e que os tópicos correspondentes, mostrados no esquemático da Figura 2, possuam os nomes indicados, para que o nodo possa reconhecê-los. Analisando o sistema concebido até aqui, vê-se que o tópico do mapa (`/map`) e as transformações de coordenadas do robô (`/tf`) já estão corretos. Além disso, o tópico do sensor de profundidade utilizado é indicado no arquivo de configuração comum dos mapas de custo mostrado na Listagem 1, garantindo assim sua integração ao nodo de navegação.

O tópico de odometria gerado pela simulação do Twil, por outro lado, possui um nome diferente do padrão, sendo então necessário renomeá-lo para `/odom`, o que é feito dentro do código do *launchfile*. Analogamente, o comando de velocidade reconhecido pelo robô está em um tópico diferente do comando gerado pela navegação, o qual é publicado em

um tópico chamado `/cmd_vel`, sendo então necessário conectar os dois através do mesmo procedimento.

Tendo concluído todas essas etapas, pode-se iniciar os testes lançando em paralelo os pacotes do Twil da forma descrita na seção 3.2 e os pacotes de navegação conforme dito acima. Com isso, observa-se com auxílio do `rviz` o modelo do robô sobre o mapa, à espera de receber um objetivo para iniciar a navegação.

Esse objetivo pode ser definido diretamente na interface gráfica do `rviz`, a qual permite que se selecione diretamente o ponto e a orientação desejada no mapa, o que é então publicado no tópico `move_base_simple/goal`, que por sua vez é lido pelo planejador de trajetórias.

As ferramentas de visualização do `rviz` são fundamentais para análise do desempenho do sistema durante sua execução, já que, além do mapa do ambiente e do modelo do robô, pode-se também visualizar, por exemplo, os mapas de custo global e local, o objetivo de navegação e a evolução das trajetórias geradas. Estas ferramentas são utilizadas no próximo capítulo visando-se avaliar os resultados obtidos, além de observar a influência dos parâmetros de configuração discutidos no comportamento do robô.

4 RESULTADOS

Neste capítulo serão apresentados os resultados dos testes realizados em cada etapa do desenvolvimento do projeto. Os primeiros resultados obtidos referem-se à execução do modelo do Twil, incluindo o controle em malha aberta de sua velocidade e a integração da câmera de profundidade. Em seguida, o comportamento do sistema de navegação desenvolvido é verificado, ajustando-se as configurações a fim de avaliar seu desempenho resultante.

4.1 Resultados preliminares

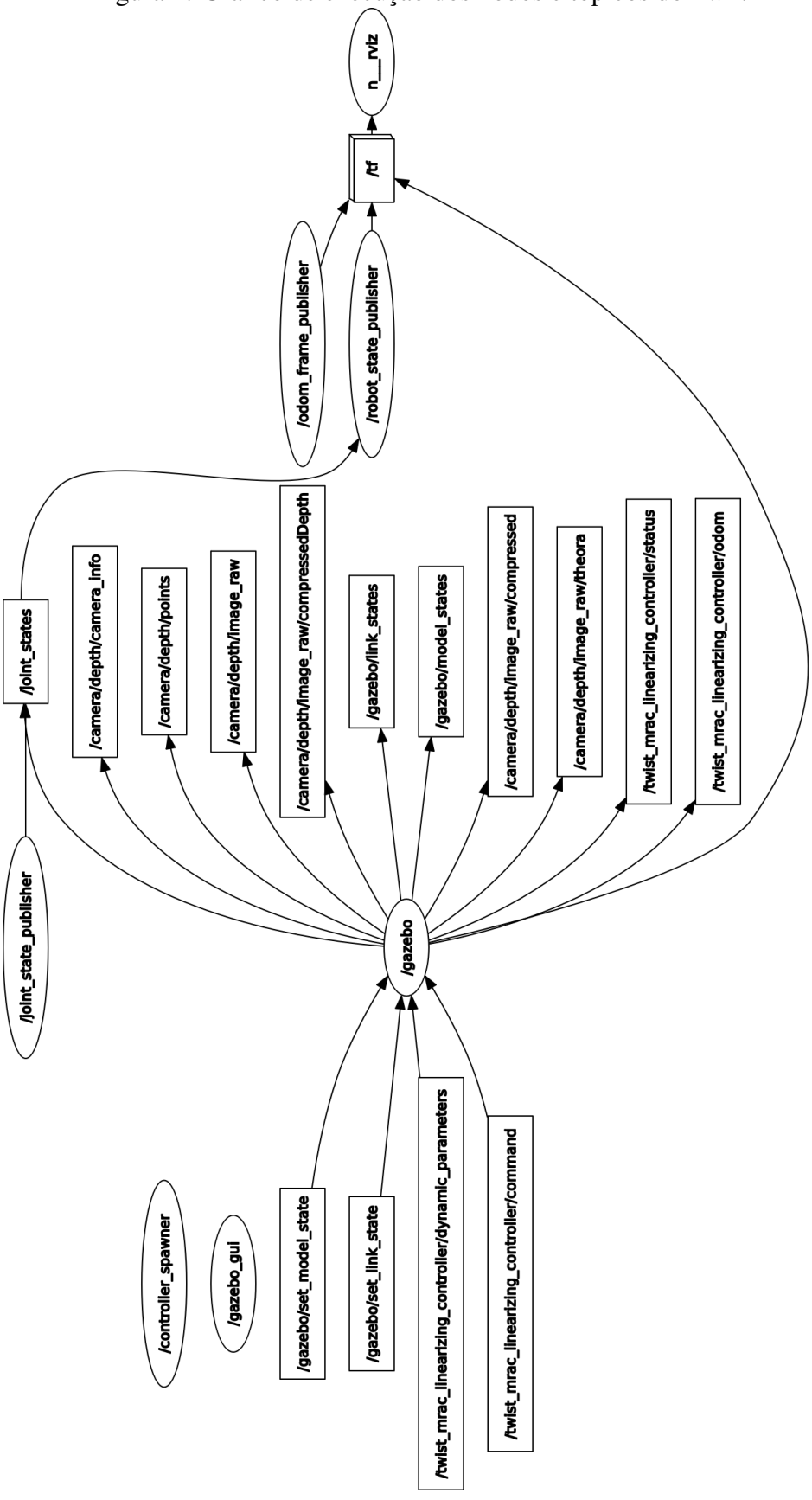
Conforme discutido no capítulo anterior, antes de implementar o sistema de navegação foi necessário obter uma simulação funcional do robô Twil no Gazebo, sendo capaz de receber comandos de velocidade, gerar dados de odometria e das transformações de coordenadas do robô e também simular o comportamento de uma câmera 3D. A Figura 4 mostra o gráfico de execução do sistema após a conclusão dessas etapas. Este gráfico é gerado pelo programa `rqt_graph`, e representa através de elipses os nodos do ROS em execução no instante em que ele foi produzido, enquanto que os tópicos são representados por retângulos.

Destaca-se nessa imagem o nodo `/gazebo`, que gere a execução da simulação. Notadamente, ele recebe os comandos do controlador de velocidade que são publicados no tópico `/twist_mrac_linearizing_controller/command` e gera os dados de odometria (`/twist_mrac_linearizing_controller/odom`) e também as transformações para o tópico `/tf`, que é lido pelo visualizador `rviz`. Vê-se também que diversos tópicos referentes à câmera são gerados pelo Gazebo, dos quais a nuvem de pontos presente no tópico `/camera/depth/points` será usada na navegação.

As Figuras 5 e 6 mostram, respectivamente, as interfaces do simulador Gazebo e do programa de visualização `rviz` após o início da execução do sistema. Na Figura 5 vê-se o modelo do robô dentro de um ambiente construído no próprio Gazebo para simular em três dimensões o mesmo andar do prédio da Escola de Engenharia da UFRGS cuja planta foi usada nesse trabalho. Nessa figura nota-se também a barra de ferramentas superior, a partir da qual pode-se adicionar objetos simples - cubos, esferas e cilindros - no ambiente de simulação e também mover e redimensionar os modelos mostrados.

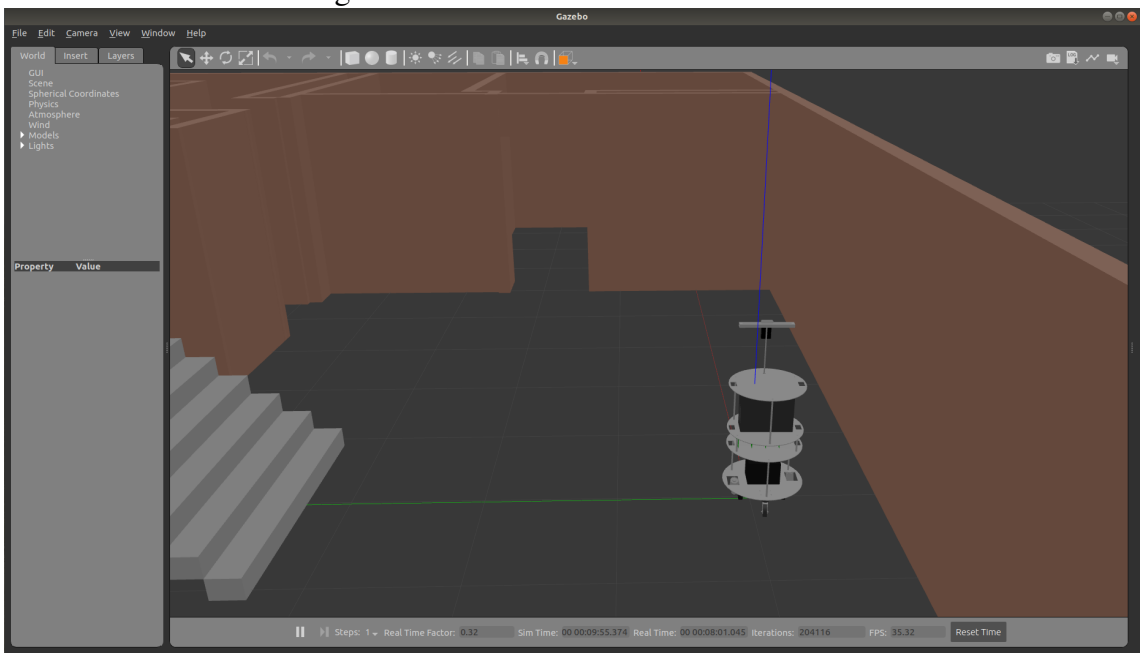
Já na interface do `rviz`, mostrada na Figura 6, vê-se à esquerda uma série de opções e caixas de seleção referentes aos dados que estão sendo visualizados na janela central. No caso dessa imagem, estão sendo mostrados o modelo do robô e as informações de odometria, representada pela seta vermelha no plano do solo cuja direção indica a orientação presente do robô. É possível configurar o número de amostras desse dado que são exibidas, o que permite ter-se um registro gráfico da trajetória executada pelo robô.

Figura 4: Gráfico de execução dos nodos e tópicos do Twil.



Fonte: Autor

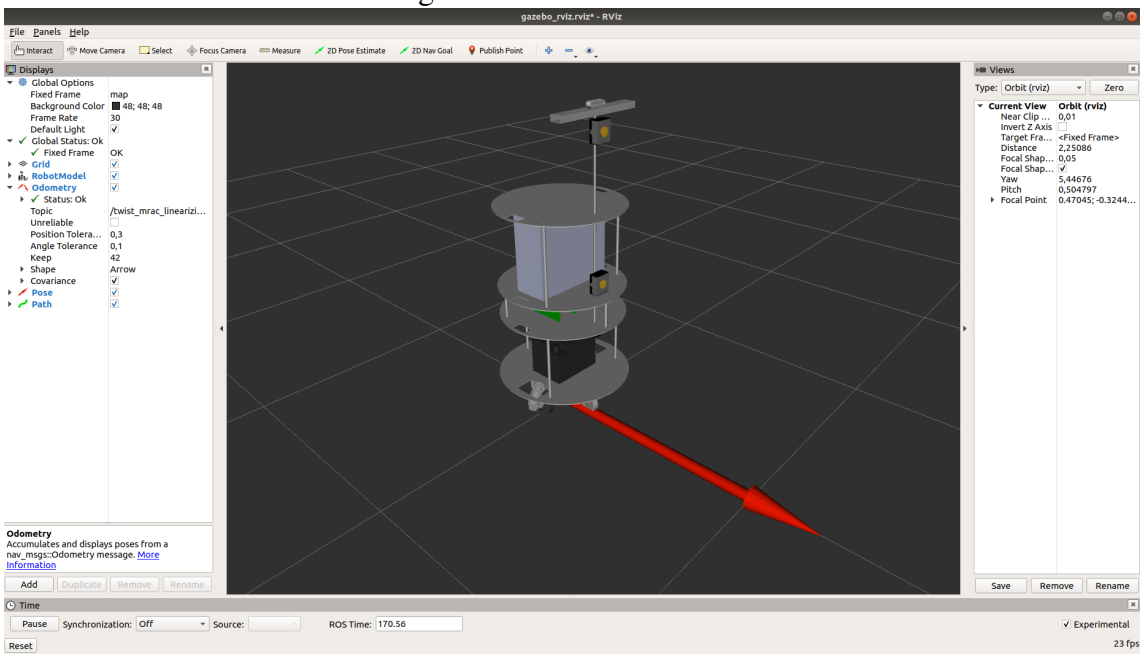
Figura 5: Interface do simulador Gazebo.



Fonte: Autor

O `rviz` também permite a visualização, por exemplo, da imagem da câmera e da nuvem de pontos produzida por ela, do mapa do ambiente e dos mapas de custo, da trajetória gerada pelo planejador e do objetivo atual da navegação, entre outras ferramentas que não serão abordadas aqui.

Figura 6: Interface do `rviz`.



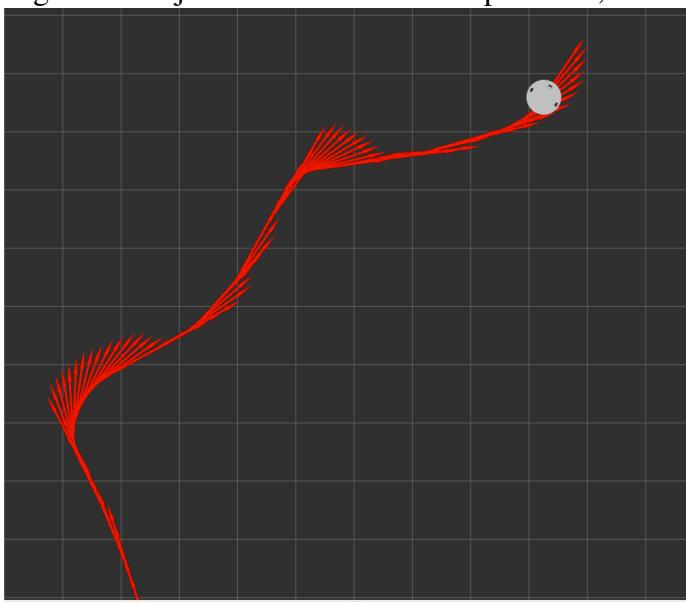
Fonte: Autor

4.1.1 Movimentação do robô

A fim de verificar-se o comportamento do modelo simulado do robô Twil frente a comandos de velocidade, foram realizados testes em malha aberta enviando comandos escritos manualmente ao tópico `/twist_mrac_linearizing_controller/command`. Conforme foi explicado na seção 3.2, o robô admite um valor de velocidade linear v e outro de velocidade angular ω, os quais foram testados primeiramente de forma isolada e após em conjunto.

Em relação à velocidade linear, alguns testes realizados com valores crescentes mostraram que era necessário um valor mínimo de aproximadamente 0.2 m/s para que o robô começasse a se movimentar, o que se deve ao atrito estático, que ocasiona este fenômeno de zona morta. Entretanto, ao contrário do esperado, este valor mínimo de velocidade faz com que o robô se mova em curva ao invés de em linha reta. Aumentando o valor v, vê-se que a trajetória se torna progressivamente mais retilínea, até estabilizar em aproximadamente 0,4 m/s. A Figura 7 representa o comportamento obtido com uma velocidade de referência de 0,3 m/s, ilustrando o movimento errático que ocorre nesse caso. Essa imagem foi obtida com auxílio do `rviz`, sendo que as setas representam consecutivas amostras do vetor gerado pela odometria, indicando assim a trajetória percorrida.

Figura 7: Trajetória em malha aberta para v=0,3 m/s.



Fonte: Autor

Se a referência de v continuar sendo aumentada, nota-se que o robô continua executando trajetórias retilíneas para valores até aproximadamente 0,7 m/s, quando as variações no sentido do movimento voltam a ocorrer. Nessa região ocorre também a saturação da velocidade resultante do robô, que estabiliza ao redor da velocidade máxima calculada de 0,47 m/s. É importante destacar que nos experimentos realizados o robô não seguiu as referências de velocidade fornecidas. Por exemplo, fornecendo uma referência de 0,4 m/s a velocidade resultante se mantém em torno de 0,3 m/s. Já a velocidade máxima é atingida para referências a partir de 0,6 m/s.

Já para a velocidade angular ω, os experimentos realizados mostraram que é preciso fornecer um valor de aproximadamente 1 rad/s para que o movimento seja iniciado par-

tindo do repouso. Entretanto, com essa referência o robô não é capaz de executar perfeitamente o movimento rotatório ao redor do seu próprio centro como esperado, realizando em vez disso um movimento em círculo com um centro externo a ele. Aumentando-se a referência de ω, o robô passa a girar em torno de si mesmo para valores de referência a partir de 1.7 rad/s

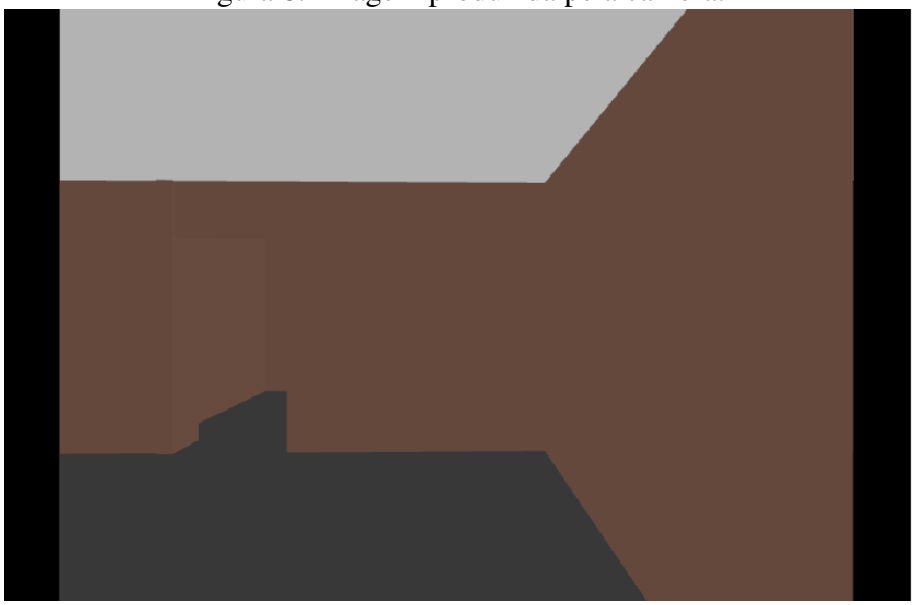
A observação dessas não-linearidades na resposta do robô em malha aberta são fundamentais para poder-se ajustar adequadamente os parâmetros de velocidade usados pelo planejador de trajetória afim de permanecer na faixa de valores que proporciona uma movimentação mais estável, facilitando assim o seguimento dos caminhos gerados pelo planejador.

4.1.2 Integração da câmera

Os últimos experimentos executados antes da integração do sistema de navegação consistiram em testar o funcionamento do *plugin* que havia sido adicionado ao modelo do robô para simular uma câmera Intel RealSense D435. Para isso, foi usada a configuração mostrada na Figura 5, onde à direita do robô há uma parede sólida e à sua frente uma parede com um portal de passagem. No `rviz`, pode-se observar a imagem produzida por esse sensor e publicada no tópico `camera/depth/image_raw` e a nuvem de pontos publicada em `camera/depth/points`, mostradas respectivamente nas Figuras 8 e 9. Como esperado, a imagem produzida pela câmera, ilustrada na Figura 8, mostra o ambiente simulado a partir do ponto de vista do robô.

Na Figura 9 a nuvem de pontos é mostrada de uma perspectiva mais elevada do que a da imagem mostrada na Figura 8, o que é possível já que se trata de uma representação tridimensional. Vê-se também a presença de um "buraco"na leitura do sensor após o portal, o que ocorre pois este pedaço do ambiente está fora do campo de visão da câmera, estando encoberto pela presença da parede. Assim, a figura mostra que efetivamente a nuvem de pontos permite a construção uma representação 3D do ambiente observado.

Figura 8: Imagem produzida pela câmera.

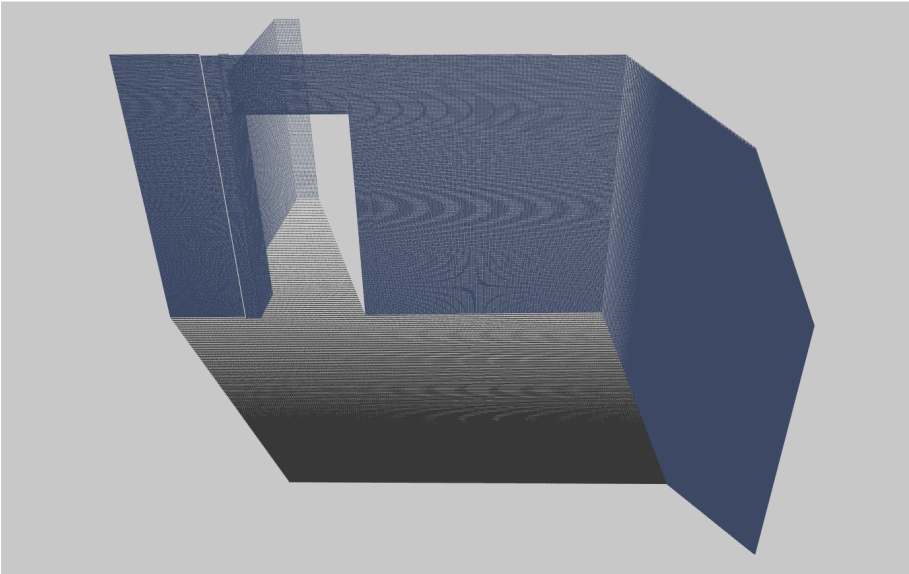


Fonte: Autor

Um problema observado durante esses testes foi que inicialmente a nuvem de pontos

não era projetada no mesmo sistema de coordenadas do robô. Em vez de aparecer em frente ao robô, como seria esperado, a representação em 3D do ambiente era projetada acima do mesmo. Descobriu-se que isso se devia à configuração do *plugin* utilizado, que considera o eixo z como sendo o que aponta para a frente da câmera, enquanto que o modelo usado nas simulações considera esse como sendo o eixo x. Para corrigir isso, foi criado um novo *frame* especificamente para o *plugin*, com uma rotação de eixos em relação ao *frame* da câmera simulada.

Figura 9: Representação da nuvem de pontos gerada pela câmera.



Fonte: Autor

4.2 Integração do sistema de navegação

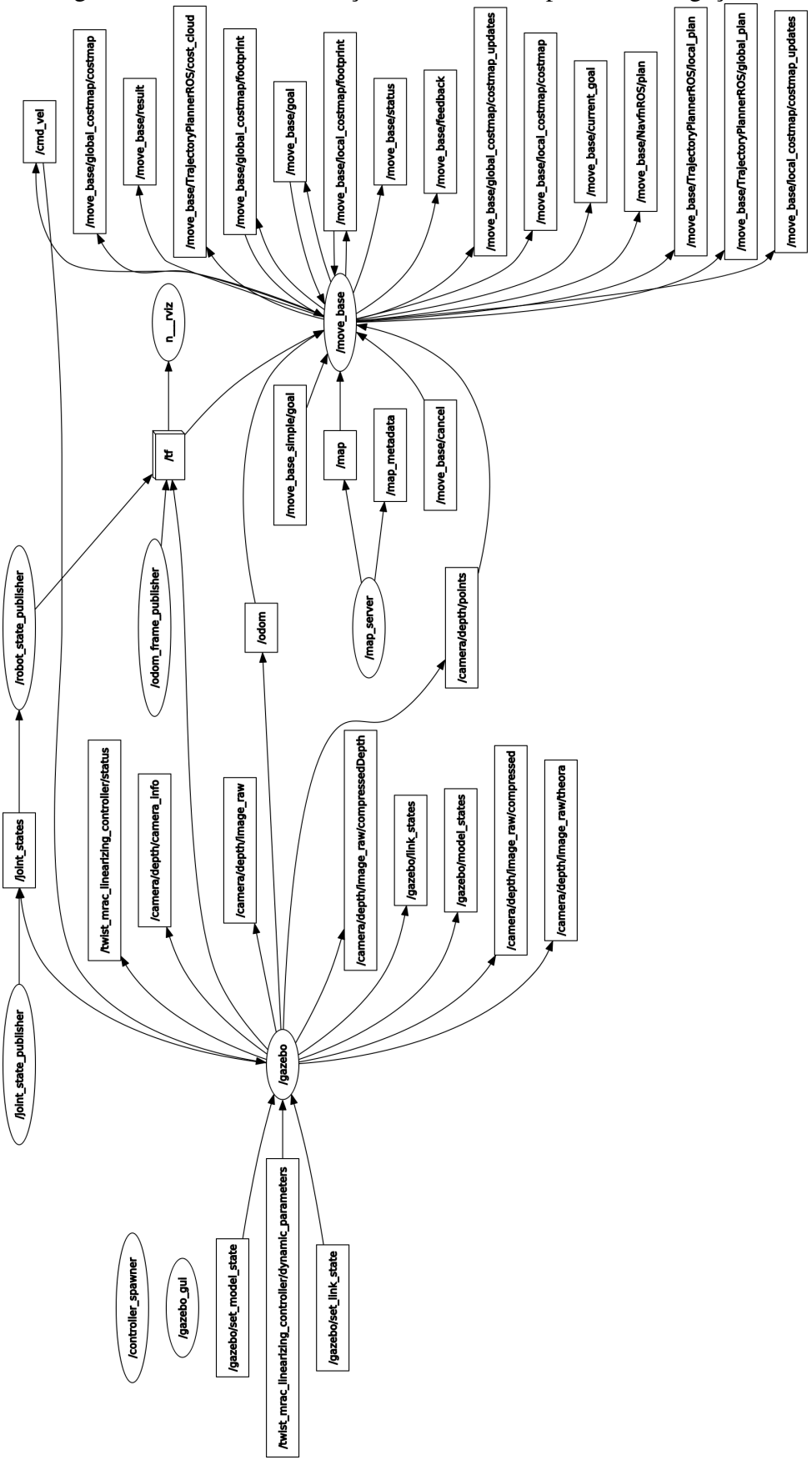
Após realizar-se os procedimentos de construção do mapa e configuração do pacote de navegação, pode-se testar o funcionamento do sistema completo, cujo gráfico de execução é mostrado na Figura 10. Assim como no gráfico mostrado na Figura 4, nota-se a presença do nodo do Gazebo, responsável por simular o comportamento do robô. Contudo, agora vê-se também o nodo `/move_base`, responsável pela geração dos mapas de custo e das trajetórias, e do `/map_server`, que fornece o mapa para a navegação.

Aqui, os tópicos de odometria e do comando de velocidade foram renomeados para `/odom` e `/cmd_vel`, respectivamente, de acordo com o padrão utilizado no *stack* de navegação. Vê-se que agora o comando de velocidade é produzido pelo `/move_base` e lido pelo Gazebo, enquanto que o contrário ocorre com a odometria. O `/move_base` recebe, ainda, as informações de `/tf`, a nuvem de pontos gerada pela câmera e um comando do usuário que corresponde ao objetivo da navegação. Dessa forma, verifica-se que todos os componentes necessário para realizar-se a navegação estão presentes e devidamente conectados, possibilitando o início dos testes de desempenho.

4.2.1 Leitura do mapa e mapas de custo

A fim de verificar se o `/map_server` interpretou corretamente a imagem da planta mostrada na Figura 3 para gerar um mapa, observa-se o tópico `/map` no `rviz`, tal qual

Figura 10: Gráfico de execução dos nodos e tópicos da navegação.

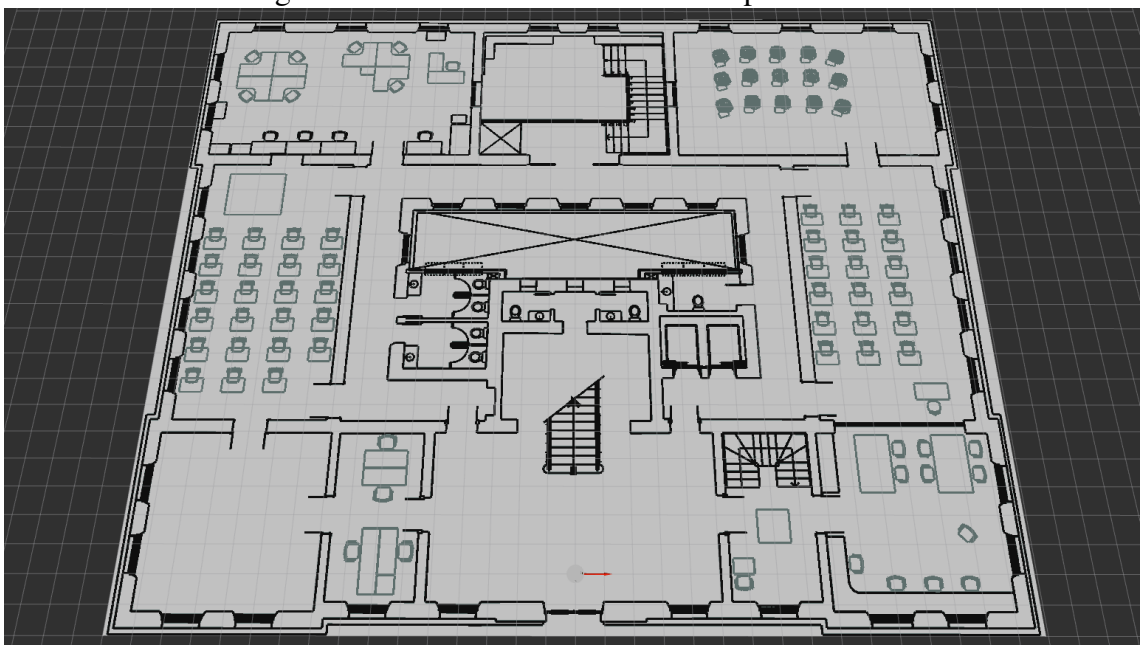


Fonte: Autor

ilustra a Figura 11. Nessa imagem é possível ver que a planta foi lida com sucesso, e os pixels foram devidamente interpretados de acordo com o código de cores e os valores limite descritos na seção 3.4.

Os elementos que haviam sido codificados como desconhecidos - como as carteiras escolares - aparecem em tonalidade cinzenta, enquanto que os considerados como ocupados são pretos. Nota-se também o posicionamento do robô, centralizado na parte inferior da imagem, que dessa forma inicia o programa localizado no saguão em frente à escadaria do prédio.

Figura 11: Modelo do robô sobre o mapa no `rviz`.



Fonte: Autor

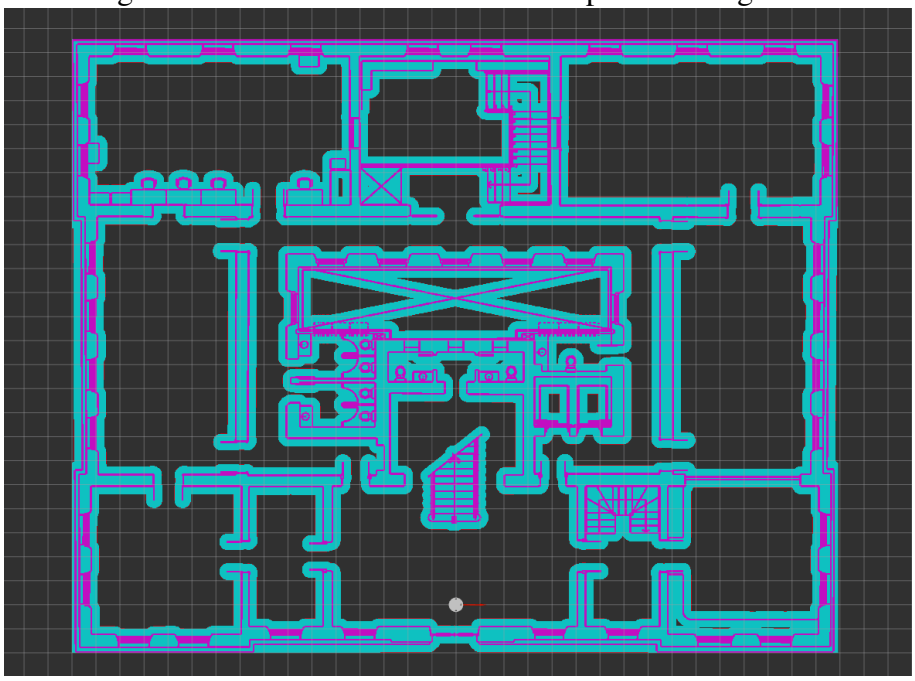
A Figura 12, por sua vez, mostra o mapa de custo global gerado pelo `/move_base` a partir do mapa ilustrado na Figura 11 e de acordo com as configurações discutidas na seção 3.5. Nota-se que aqui os elementos do mapa cuja presença é incerta são ignorados. Além disso, percebe também a presença de uma aura ao redor dos pixels ocupados, os quais possuem custo máximo. Essa área representa a zona inscrita no raio de inflação, que é um valor parametrizável que indica a máxima distância de um obstáculo a partir da qual o custo é nulo. Ao longo dos experimentos viu-se que o raio de inflação possui grande influência no planejador de trajetórias, já que este traça o caminho a ser percorrido pelo robô através da região de menor custo.

4.2.2 Desempenho da navegação

Como explicado anteriormente, a interface do `rviz` possibilita a publicação de comandos de pose para o navegador, os quais consistem no objetivo de posição e orientação a ser atingido pelo robô. Após gerada, essa pose pode ser visualizada no mesmo programa, assim como as trajetórias global e local geradas pelo `/move_base`.

A Figura 13 mostra um instante da execução desse sistema. Vê-se nela, assim como nos testes em malha aberta, o robô e as medidas sucessivas de sua odometria. Ademais, observa-se no canto superior direito da imagem o objetivo de navegação escolhido, e a

Figura 12: Modelo do robô sobre o mapa de custo global.



Fonte: Autor

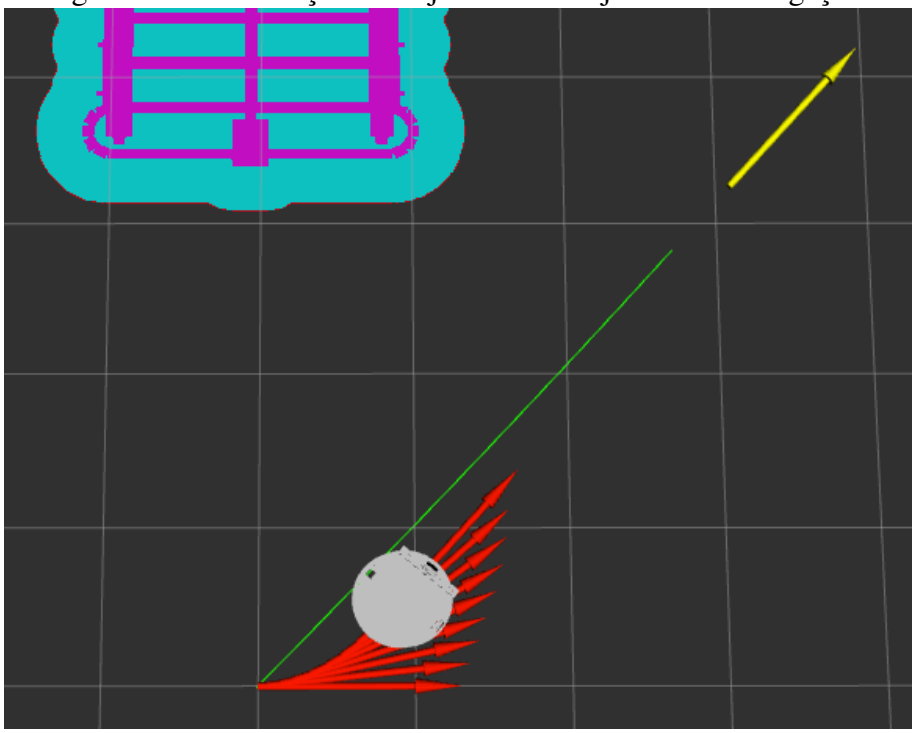
linha partindo da posição inicial do robô em direção a esse objetivo representa a trajetória global gerada pelo planejador. Essa trajetória age como uma referência, traçando o caminho mais simples até o objetivo, enquanto que o planejador local se encarrega de ajustar os comandos de velocidade transmitidos ao robô de forma a seguir, dentro do possível, o traçado global.

Nos primeiros testes realizados, notou-se que o robô tinha dificuldade em atingir o objetivo definido e parar de se mover. Em vez disso, ele permanecia indefinidamente girando ao seu redor, falhando em posicionar-se de forma precisa no local desejado. Assim, percebeu-se que era necessário aumentar a tolerância aceita pelo sistema para indicar que o objetivo foi atingido, já que a movimentação do robô mostrou não ser suficientemente precisa para trabalhar dentro da tolerância padrão de 0,1 metros em relação ao pontos xy de referência e 0,05 radianos em relação à orientação desejada.

Seria possível resolver esse problema definindo-se uma tolerância significativamente mais elevada do que os valores padrão. Entretanto, sendo a tolerância um parâmetro de qualidade do sistema de navegação, é desejável que ela seja tão pequena quanto possível sem causar o comportamento descrito acima. Dessa forma, realizaram-se diversos testes até escolher-se os valores de 0,25 m para a tolerância de posição e 0,1 rad para a de orientação, o que pode ser considerado adequado para a maioria das tarefas que poderiam ser efetuadas por um robô das dimensões do Twil.

É importante destacar que , mesmo após a realização desses ajustes, em algumas situações o robô ainda tem dificuldades em realizar a navegação de maneira correta. Um exemplo são os casos onde a orientação escolhida para o objetivo difere muito da orientação usada pelo robô para executar o trajeto direto entre os dois pontos. Nesse caso, o robô deveria traçar uma trajetória contornando o objetivo de forma a atingi-lo com uma orientação próxima da que se deseja, contudo o planejador não se mostrou capaz de conceber trajetórias dessa forma. Assim, uma maneira de contornar esse problema é definir

Figura 13: Visualização do objetivo e da trajetória de navegação.



Fonte: Autor

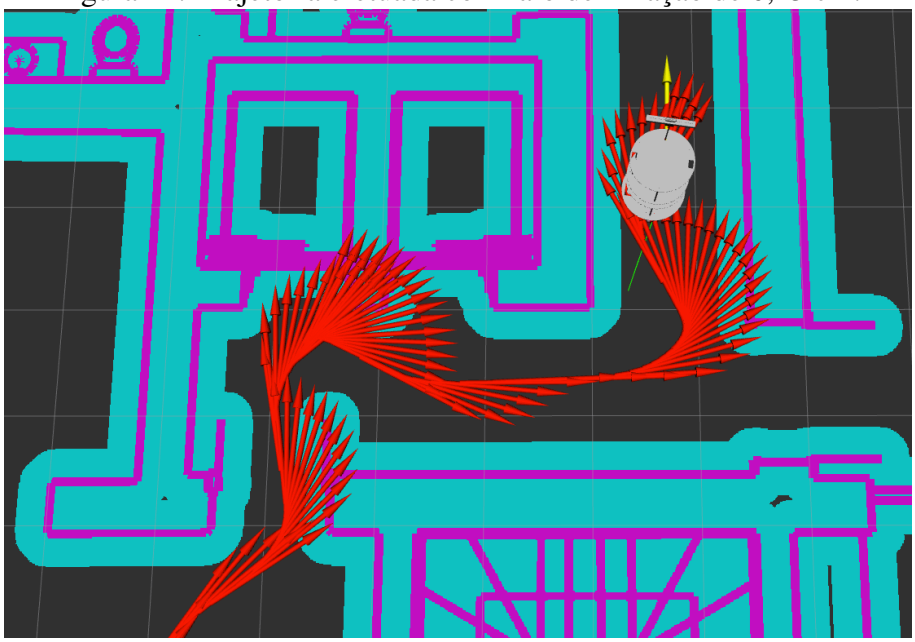
um objetivo intermediário que possa ajudar o robô a navegar até o destino final.

Além das tolerâncias, outro parâmetro sobre o qual foram realizados experimentos foi o raio de inflação dos obstáculos, que se mostrou fundamental para melhorar o desempenho do robô ao navegar através das portas e corredores do ambiente estudado. No mapa de custo mostrado na Figura 12, esse raio está configurado em 0,3 metros, valor idêntico ao raio do robô. Contudo, nos testes efetuados com esse valor viu-se que o robô tinha dificuldade nas situações onde era necessário navegar próximo aos obstáculos, por exemplo ao atravessar uma porta. O comportamento observado nesses casos foi que o robô tentava executar uma trajetória que ia de encontro ou acabaria tocando em um obstáculo, forçando-o a corrigi-la por meio de um giro ou um movimento em ré, o que diminui a fluidez da movimentação.

O caso em que o raio de inflação é igual ao raio do robô é o caso limite desse tipo de comportamento, que se mostrou predominante nos experimentos feitos com valores menores. A Figura 14 ilustra o caminho efetuado pelo robô com raio de inflação igual a 0,25 m. Nota-se que, como descrito acima, a trajetória original vai de encontro às paredes, forçando o robô a desviar. Isso ocorre pois o raio de inflação indica a partir de que distância do obstáculo é seguro navegar; logo, se esse valor for menor ou igual ao raio do próprio robô, o planejador pode tentar passar por pontos que na verdade causam o contato ou colisão com o obstáculo.

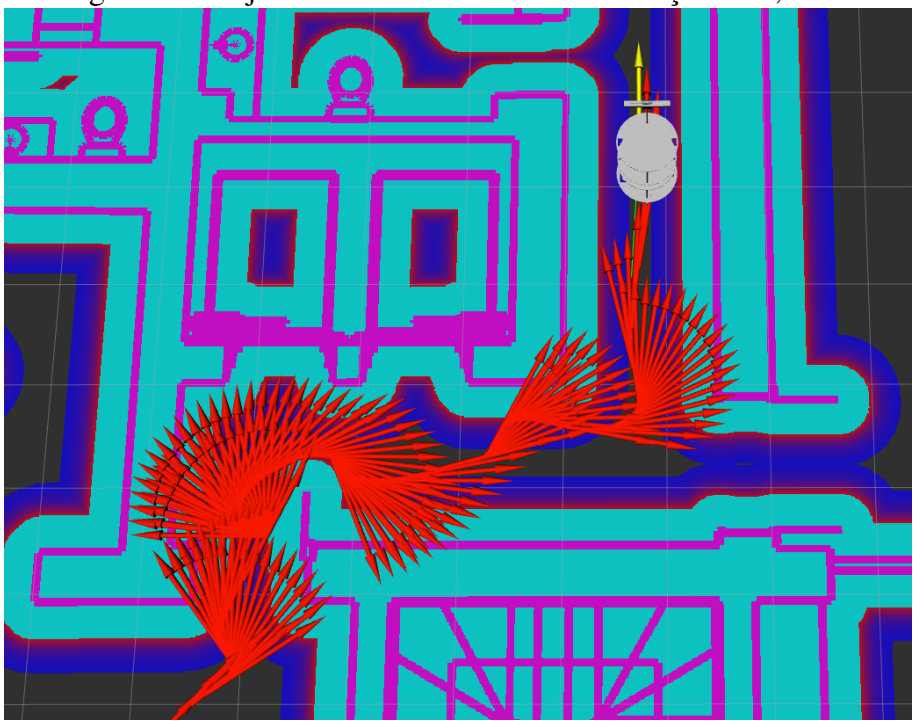
Já a escolha de um raio de inflação mais elevado pode ocasionar um problema diferente. Nesse caso, garante-se que as trajetórias planejadas são realmente seguras, já que a região de baixo custo estará a uma distância maior dos obstáculos do que o raio do robô. Contudo, isso pode fazer com que o espaço disponível para o robô seja muito reduzido, dificultando o seguimento das trajetórias definidas e causando oscilações. Isso pode ser ilustrado pelo resultado obtido na Figura 15, onde o raio de inflação é de 0,6 m.

Figura 14: Trajetória efetuada com raio de inflação de 0,25 cm.



Fonte: Autor

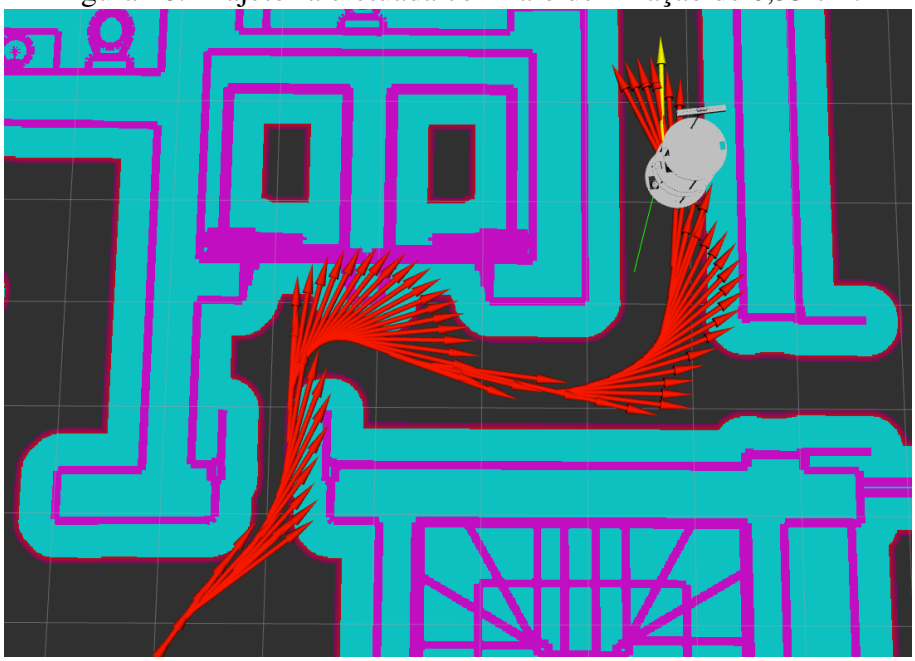
Figura 15: Trajetória efetuada com raio de inflação de 0,6 cm.



Fonte: Autor

Dessa forma, procura-se ajustar esse parâmetro para um valor maior do que o raio do robô mas não demasiadamente elevado. Após testar-se diversas possibilidades, o valor de 0,35 metros rendeu um desempenho considerado satisfatório, como mostra a Figura 16. Vê-se que nesse caso nenhum dos comportamentos descritos acima se repetiu, e o robô foi capaz de atingir o objetivo sem movimentos abruptos de troca de direção, os quais aumentam o tempo transcorrido no trajeto.

Figura 16: Trajetória efetuada com raio de inflação de 0,35 cm.



Fonte: Autor

4.2.3 Navegação em um ambiente simulado

Os testes cujos resultados foram apresentados acima, usados para o ajuste dos parâmetros da navegação, foram efetuados com um ambiente livre no simulador, baseando-se apenas na leitura do mapa estático fornecido. Dessa forma, os dados fornecidos pela câmera de profundidade não são usados diretamente, já que, como não havia nenhum objeto além do robô no ambiente do Gazebo, as leituras do sensor nunca detectavam nada.

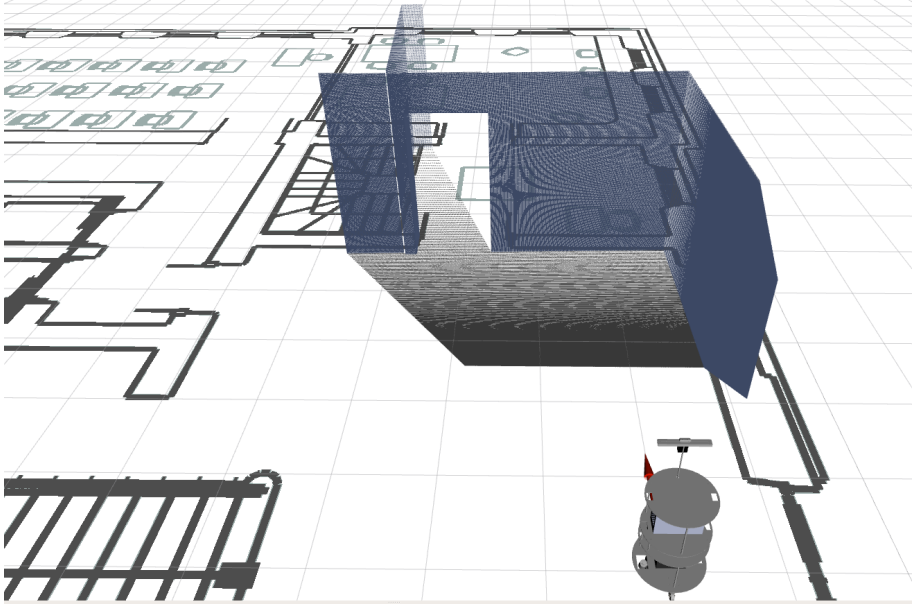
Posteriormente, o sistema completo foi executado no ambiente simulado que foi construído de para representar o mesmo local do mapa utilizado, tal qual é mostrado na Figura 5. Com isso, pode-se observar no `rviz` tanto o mapa gerado pelo `map_server` quanto a nuvem de pontos produzida pela câmera, obtendo-se o resultado mostrado na Figura 17. Nessa imagem vê-se que os objetos tridimensionais construídos com os dados do sensor combinam com o trecho do mapa correspondente a eles, o que pode ser percebido pela posição da porta e das paredes nas duas representações.

Entretanto, esse resultado só é obtido quando o robô se encontra em sua posição inicial, e à medida que ele começa a se mover, essas duas representações param de convergir. Isso ocorre pois a posição do robô mostrada no `rviz` é obtida a partir dos dados da odometria, enquanto que os elementos que se encontram no campo de visão da câmera correspondem à posição real do robô que está sendo simulado no Gazebo, e, como a odometria é uma forma de medição que possui uma deriva, a posição e orientação medidas

deixam de corresponder aos valores da simulação. Em um dado momento, por exemplo, pode haver uma porta no campo de visão do robô simulado, que será mostrada na nuvem de pontos; mas, por causa do erro de localização, é possível que no mapa essa porta não esteja em frente ao robô ou então que esteja a uma distância diferente da medida pela câmera.

Dessa forma, para que fosse possível usar os dados da câmera para evitar obstáculos na navegação, seria preciso primeiramente implementar um sistema de localização mais sofisticado, utilizando outros dados para complementar a odometria, a qual possui um erro que aumenta ao longo do deslocamento. Existem ferramentas no ROS para se realizar tal tarefa; contudo, o tema da localização não está no escopo do presente trabalho.

Figura 17: Superposição do mapa com os dados do sensor no `rviz`.



Fonte: Autor

5 CONCLUSÃO

Ao final, observando o conjunto do que foi desenvolvido e testado ao longo desse trabalho, pode-se dizer que o objetivo inicial proposto de estudar um robô já existente e ampliar suas capacidades de software por meio da integração de um sistema de navegação foi cumprido. O sistema obtido é capaz de planejar trajetórias através de um mapa conhecido e de executá-las de forma satisfatória. Além disso, a integração do sensoriamento por meio da câmera de profundidade permite a detecção de obstáculos presentes no ambiente, o que, após a adição de um sistema de localização, poderá ser usado para navegar-se na presença de obstáculos desconhecidos.

A metodologia aplicada nesse trabalho incluiu o uso de diversas ferramentas e sistemas desenvolvidos por diferentes autores, como o simulador Gazebo, o plugin da câmera de profundidade usado, o modelo do robô Twil e os pacotes de navegação. Isso demonstra como a modularidade proporcionada pelo ROS auxilia o desenvolvimento de projetos na área da robótica, já que seu *framework* comum facilita a integração de múltiplas funcionalidades sem que a questão da compatibilidade se torne um problema.

Além disso, o uso das bibliotecas presentes no ROS que implementam soluções frequentemente necessárias em sistemas robóticos, como o *stack* de navegação, ilustra a possibilidade de trabalhar com sujeitos complexos tais como a navegação autônoma através de ferramentas com alto nível de abstração, sendo assim acessíveis à diferentes tipos de desenvolvedores e usuários.

Os experimentos realizados a fim de avaliar e otimizar o desempenho do sistema de navegação evidenciam a diversidade de ferramentas de análise presentes no escopo do ROS e a importância que uma parametrização adequada possui nesse tipo de projeto. Apesar de aqui ter-se utilizado sobretudo ferramentas gráficas para analisar o planejamento e seguimento de trajetória, destaca-se que essa não é a única abordagem possível, sendo viável também, por exemplo, inspecionar-se diretamente as variáveis numéricas produzidas pelo sistema, como foi feito nos testes de velocidade em malha aberta.

A partir do que foi desenvolvimento nesse projeto, pode-se citar como possível trabalho futuro a integração de um sistema de localização para permitir a navegação em um ambiente na presença de obstáculos. Ademais, há a possibilidade de utilizar um algoritmo de SLAM para eliminar a necessidade de possuir-se um mapa do local sobre o qual se deseja navegar. Finalmente, o sistema de navegação estudado pode ser testado no robô real a fim de validar seu funcionamento e posteriormente utilizá-lo em aplicações práticas.

REFERÊNCIAS

BARROS, T. T. T. *Modelagem e implementação no ROS de um controlador para manipuladores móveis*. 2014. f. 159. Dissertação (Mestrado em engenharia elétrica) – Programa de Pós-Graduação em Engenharia Elétrica, Universidade Federal do Rio Grande do Sul, Porto Alegre.

BORENSTEIN, J. et al. Mobile robot positioning: Sensors and techniques. *Journal of robotic systems*, Wiley Online Library, v. 14, n. 4, p. 231–249, 1997.

BROCK, O.; KHATIB, O. High-speed navigation using the global dynamic window approach. In: PROCEEDINGS of the 1999 IEEE International Conference on Robotics and Automation. Detroit, MI, EUA: IEEE, 1999. v. 1, p. 341–346.

DELLAERT, F. et al. Monte carlo localization for mobile robots. In: PROCEEDINGS 1999 IEEE International Conference on Robotics and Automation. Detroit, MI, EUA: IEEE, 1999. v. 2, p. 1322–1328.

FILLIAT, D. *Robotique mobile*. Palaiseau, França, 2016. p. 175.

FILLIAT, D.; MEYER, J.-A. Global localization and topological map learning for robot navigation. In: SEVENTH International Conference on simulation of adaptive behavior: From Animals to Animats (SAB-2002). Edimburgo, Reino Unido: The MIT Press, 2002. p. 131–140.

FOX, D.; BURGARD, W.; THRUN, S. The dynamic window approach to collision avoidance. *IEEE Robotics & Automation Magazine*, IEEE, v. 4, n. 1, p. 23–33, 1997.

FRANZ, M. O.; MALLOT, H. A. Biomimetic robot navigation. *Robotics and autonomous Systems*, Elsevier, v. 30, n. 1-2, p. 133–153, 2000.

GUIMARÃES, R. L. et al. ROS navigation: Concepts and tutorial. In: ROBOT Operating System (ROS). Curitiba, Brasil: Springer, 2016. p. 121–160.

INTEL REALSENSE. *Intel RealSense D400 Series Product Family*. Santa Clara, CA, EUA, jan. 2019a. Revision 005, Document Number: 337029-005. Disponível em: <https://www.intel.com/content/dam/support/us/en/documents/emerging-technologies/intel-realsense-technology/Intel-RealSense-D400-Series-Datasheet.pdf>. Acesso em: 25 out. 2019.

INTEL REALSENSE. *ROS Wrapper for Intel® RealSense Devices*. 2019b. Disponível em: <https://github.com/IntelRealSense/realsense-ros>. Acesso em: 24 out. 2019.

KOENIG, N.; HOWARD, A. Design and use paradigms for gazebo, an open-source multi-robot simulator. In: 2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)(IEEE Cat. No. 04CH37566). Sendai, Japão: IEEE, 2004. v. 3, p. 2149–2154.

LEVITT, T. S. Qualitative navigation for mobile robots. *Int. J. Artificial Intelligence*, v. 44, p. 305–360, 1990.

OPEN SOURCE ROBOTICS FOUNDATION. *Tutorial: Using Gazebo plugins with ROS*. 2014. Disponível em: <http://gazebosim.org/tutorials?tut=ros_gzplugins&cat=connect_ros#OpenniKinect>. Acesso em: 24 mai. 2019.

OSRF. *Gazebo: robot simulation made easy*. 2014. Disponível em: <http://gazebosim.org>. Acesso em: 11 nov. 2019.

OSRF. *SDF*. 2019. Disponível em: <http://sdformat.org>. Acesso em: 11 nov. 2019.

QUIGLEY, M. et al. ROS: an open-source Robot Operating System. In: ICRA workshop on open source software. Kobe, Japan: IEEE, 2009. v. 3.

ROSWIKI. *Setup and configuration of the navigation stack*. 2018. Disponível em: <http://wiki.ros.org/navigation/Tutorials/RobotSetup>. Acesso em: 8 mai. 2019.

SIEGWART, R.; NOURBAKHSH, I. R.; SCARAMUZZA, D. *Introduction to autonomous mobile robots*. Cambridge, MA, EUA: MIT press, 2011.

TRULLIER, O.; MEYER, J.-A. Biomimetic Navigation Models and Strategies in Animats. *AI Communications*, v. 10, n. 2, p. 79–92, 1997.

TRULLIER, O.; WIENER, S. I. et al. Biologically based artificial navigation systems: Review and prospects. *Progress in neurobiology*, Elsevier, v. 51, n. 5, p. 483–544, 1997.

ZAMAN, S.; SLANY, W.; STEINBAUER, G. ROS-based mapping, localization and autonomous navigation using a Pioneer 3-DX robot and their relevant issues. In: 2011 Saudi International Electronics, Communications and Photonics Conference (SIECPC). Riyadh, Arábia Saudita: IEEE, 2011. p. 1–5.

ZHENG, K. *ROS Navigation Tuning Guide*. 2016. Disponível em: <http://kaiyuzheng.me/documents/papers/ros_navguide.pdf>. Acesso em: 5 mai. 2019.