TRABALHO DE GRADUAÇÃO

UMA ARQUITETURA DE NAVEGAÇÃO PARA ROBÔS MÓVEIS

Rodrigo Werberich da Silva Moreira de Oliveira

Brasília, julho de 2017

ENGENHARIA MECATRÔNICA

UNIVERSIDADE DE BRASÍLIA

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

TRABALHO DE GRADUAÇÃO

UMA ARQUITETURA DE NAVEGAÇÃO PARA ROBÔS MÓVEIS

Rodrigo Werberich da Silva Moreira de Oliveira

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

Banca Examinadora

Professor Geovany Araújo Borges, ENE/UnB
Orientador

Professor Guilherme Novaes Ramos, CIC/UnB
Coorientador

Professora Carla M. C.e C. Koike, CIC/UnB
Examinador interno

Professor Alexandre Ricardo Soares Romariz, ENE/UnB
Examinador interno

Brasília, julho de 2017

FICHA CATALOGRÁFICA

DE OLIVEIRA, RODRIGO WERBERICH DA SILVA MOREIRA
Uma Arquitetura de Navegação para Robôs Móveis

[Distrito Federal] 2017.

xiii, 70p., 297 mm (FT/UnB, Engenheiro, Controle e Automação, 2017).
Trabalho de Graduação – Universidade de Brasília. Faculdade de Tecnologia.

1. Arcabouço de Navegação 2.Planejamento de Rotas
3. Robótica

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

REFERÊNCIA BIBLIOGRÁFICA

DE OLIVEIRA, RODRIGO WERBERICH DA SILVA MOREIRA, (2017). Uma Arquitetura de Navegação para Robôs Móveis. Trabalho de Graduação em Engenharia de Controle e Automação, Publicação FT. TG-$n°$09/2017, Faculdade de Tecnologia, Universidade de Brasília, Brasília, DF, 70p.

CESSÃO DE DIREITOS

AUTOR: Rodrigo Werberich da Silva Moreira de Oliveira

TÍTULO DO TRABALHO DE GRADUAÇÃO: Uma Arquitetura de Navegação para Robôs Móveis.

GRAU: Engenheiro ANO: 2017

Rodrigo Werberich da Silva Moreira de Oliveira

Campus Darcy Ribeiro, SG-11, Universidade de Brasília.

70919-970 Brasília – DF – Brasil.

Dedicatória

Aos meus pai, Rui e Eliana e a equipe DROID.

Rodrigo Werberich da Silva Moreira de Oliveira

Agradecimentos

Primeiramente gostaria de agradecer aos meus orientadores, Prof. Geovany Borges e Prof. Guilherme N. Ramos, que foram sempre muito prestativos, tirando minhas dúvidas, me dando conselhos e me guiando para o desenvolvimento de um trabalho melhor, disponibilizando seu tempo em suas agendas muito apertadas.

Gostaria de agradecer muito ao meus companheiros de curso, e especialmente a DROID e todas as pessoas maravilhosas que a compõe. Essa equipe foi o lugar que me fez apaixonar completamente por essa profissão e nunca duvidar do caminho que eu escolhi. Esse trabalho é minha tentativa de devolver tudo que eu aprendi com vocês. Obrigado, Abdullah, Bruno, Schiavini, Gabriel, Letícia Ploeg, Bamidele, Daniel, Natalia, Pedro, Sara, Teo, Bira, Mah, Aninha, Felipe, entre tantos outros que vivenciaram essa equipe comigo. E um agradecimento especial para João Bosco, que entrou comigo no primeiro semestre e se tornou um dos meus melhores amigos.

Sou muito grato ao meus queridos amigos e mentores da 4Flyers, Giordano, Artur, Rodrigo e Alexandre, por sempre me incentivarem e ajudarem, principalmente entendendo a minha situação e me dando o tempo necessário para que eu pudesse fazer esse trabalho. Muito obrigado pela oportunidade de não só trabalhar com vocês, mas de ser seu amigo.

Um agradecimento muito especial a todos que ajudaram no desenvolvimento do Bruce, aguentando todos as suas birras e malcriações. Muito obrigado Thiago e Luan. E mais ainda para as três pessoas que não só criaram o Bruce conosco, mas também fizeram parte deste trabalho: Letícia Helena, Camila Brito e Ricardo Bauchspiess. Obrigado também a DROID, por fornecer todo o apoio para a construção robô e proporcionar as condições para que participássemos da competição. Nada desse trabalho teria sido feito sem vocês.

Camila, muito obrigado pela oportunidade de trabalhar com você e ser seu amigo. Todas as experiências que vivenciamos juntos me fez crescer muito, tanto profissionalmente, quanto pessoalmente. Continue sendo sempre a pessoa que você é.

Letícia Helena, você sempre me acompanhou desde o início dessa jornada, sempre me motivando e me animando, nunca me deixando desistir ou mostrar nenhum sinal de fraqueza. Estivemos juntos como amigos e namorados durante quase toda essa experiência, vivenciando inclusive um intercâmbio que com certeza não teria sido tão bom sem você. Muito obrigado por tudo que você proporcionou na minha vida, com certeza não teria chegado aqui sem você. Te amo do fundo do meu coração.

Por fim, não poderia deixar de agradecer minha família. Muito obrigado por sempre acreditar em mim, e estar sempre lá quando eu preciso. Obrigado aos meus pais Rui Moreira e Eliana Werberich, e aos meus irmãos e irmãs Carla, Carol, Karen e Rhon por tudo que eles já fizeram pra mim. Amo muito todos vocês.

Rodrigo Werberich da Silva Moreira de Oliveira

RESUMO

Este trabalho apresenta o desenvolvimento de um arcabouço de navegação capaz de controlar os comportamentos de um robô móvel e sua posterior implementação em um robô desenvolvido de forma paralela para superar o desafio proposto pela competição *Robomagellan*. Esse arcabouço proposto é dividido em três níveis estruturais, cada um responsável por uma determinada função. O nível organizacional é o mais abstrato dos três, e é responsável por coordenar todos os comportamentos do sistema. O nível funcional é a camada intermediária, na qual os comportamentos controlados pelo nível acima são de fato implementados de maneira modular. O nível executivo é responsável por lidar com o baixo nível, traduzindo as leituras da plataforma física para um nível abstrato, assim como o caminho contrário. A implementação no robô exigiu o desenvolvimento de vários módulos diferentes, sendo a elaboração de um planejador de rotas também incluída neste trabalho. É utilizado o algoritmo RRT de planejamento de rotas e apresentado uma maneira simples de construir o espaço de configurações necessário para utilizá-lo. Os resultados obtidos demonstram que o planejador funciona como desejado, e que o arcabouço é capaz de criar arquiteturas de controle de navegação com as características desejadas.

Palavras Chave: robótica móvel, arquitetura de controle, arcabouço de navegação, competições de robótica, planejamento de rotas, RRT

ABSTRACT

This work describes the development of a navigation framework capable of controlling the behaviours of a mobile robot and its implementation in a robot developed simultaneously in order to complete the challenge from the Robomagellan competition. The proposed framework is divided in three different levels, each one responsible for an specific function. The organizational level is the most abstract of them and it is responsible for the coordination of all the system's behaviours. The functional level is the middle layer, in which the behaviours controlled by the superior levels are really implemented as modules. The executive level is responsible for dealing with low level, translating the reading from the physical platform to an abstract level, and also the other way around. The implementation on the robot also required the development of different modules. The creation of the route planning module is also part of this work. We used the RRT algorithm and a simple way was presented to create its configuration space. The results obtained show that this planner works as desired, and that the framework is capable of creating navigation control architectures with the desired traits.

Keywords: mobile robotics, control architecture, navigation framework, robotics competition, route planning, RRT

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SÍMBOLOS

Símbolos Latinos

x	Estado
X	Espaço de estados
u	Ação
$U(x)$	Espaço de ações do estado x
U	Conjunto das ações dos possíveis estados x
$\mathbb{Z}$	Conjunto dos números inteiros
$\mathbb{N}$	Conjunto dos números naturais
Q	Uma lista de elementos
b	Nível de ramificação de uma árvore de busca
m	Profundidade de uma árvore de busca
l	Largura de um mapa de custo
c	Comprimento de um mapa de custo
$\mathbb{R}$	Conjunto dos números reais
$\mathbb{S}$	Espaço rotacional
$\mathcal{C}$	Espaço de configurações
q	Configuração de um corpo rígido
$\mathcal{A}$	Definição de um corpo rígido
$\mathcal{O}$	Conjunto que define a posição de obstáculos
$\mathcal{G}$	Grafo topológico
V	Vértices de um grafo topológico
E	Arestas de um grafo topológico
$\mathcal{S}$	Conjunto de todos os pontos alcançados pelo grafo topológico
$\mathcal{T}$	Uma árvore que faz parte de $\mathcal{G}$
i, j, k	Contador

Símbolos Gregos

α	Sequência densa e infinita	
ρ	Distância entre duas configurações	
ω	Velocidade angular	[rad/s]

Subscritos

ref	referência
fer	ferramenta
sis	sistema
des	desejado
G	objetivo
I	inicial
obs	obstáculo
$free$	livre

Sobrescritos

$\cdot$	Variação temporal
$'$	Novo estado após ação
$-$	Vetor

Siglas

LARC	Competição Latino-Americana de Robótica - *Latin American Robotics Competition*
UnB	Universidade de Brasília
DROID	Divisão de Robótica Inteligente
GPS	Sistema de Posicionamento Global - *Global Positioning System*
SLAM	*Simultaneous Localization and Mapping*
GPIO	Entradas e Saídas de Propósito Geral - *General Purpose Input/Output*
ROS	*Robot Operating System*
RF	Rádio Frequência
IMU	Unidade de medida inercial - *Inertial Measurement Unit*
RDEs	Ambientes de Desenvolvimento Robótico - *Robotic Development Environments*
ODE	*Open Dynamics Engine*
IDA*	*Iterative Deepening A**
SE	Espaços Euclidianos Especiais - *Special Euclidean spaces*
RDT	*Rapidly Exploring Dense Trees*
RRT	*Rapidly Exploring Random Tree*
FSM	Máquina de Estados Finitos - *Finite State Machine*

Capítulo 1

Introdução

1.1 Contextualização

A robótica é um tema que, cada vez mais, está presente em notícias e capas de jornais. Os robôs já estão há muito tempo presentes no chão de fábrica, construindo produtos como carros, entretanto, as últimas pesquisas provenientes da indústria e laboratórios acadêmicos têm capturado a imaginação do público como nunca antes. Estão surgindo robôs capazes de navegar de maneira independente pela rua, realizar cirurgias, e até mesmo funcionar como astronautas, explorando a superfície de Marte [1].

A maioria desses tópicos que instigam a curiosidade das pessoas tem, geralmente, pouco a ver com os braços robóticos presentes nas fábricas, conforme ilustrado pela Figura 1.1, e mais a ver com robôs móveis, como , por exemplo, o robô astronauta *Curiosity*, da Figura 1.2. Dessa

Figura 1.1: Linha de montagem com braços robóticos. - Fonte: Framepool [1]

[1] http://footage.framepool.com/en/shot/408614787-robot-arm-artificial-intelligence-assembly-montage-assembling

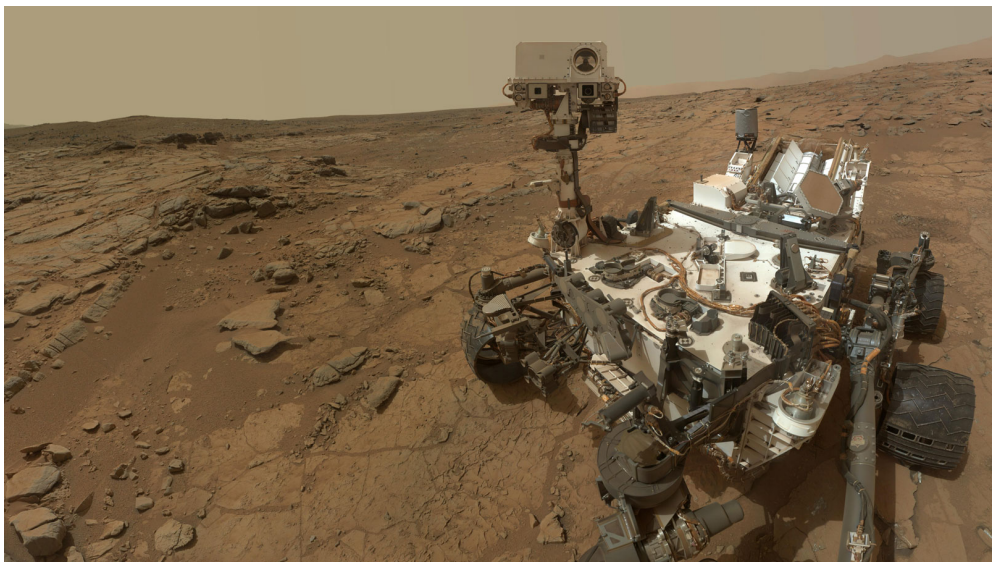


Figura 1.2: *Curiosity* o robô astronauta da NASA, tirando uma foto de si mesmo na superfície de Marte. - Fonte: Nasa[2]

maneira, o campo da robótica móvel tem crescido muito nos últimos anos, com inclusive um recente surgimento de robôs domésticos aspiradores de pó no mercado. No entanto, a robótica móvel ainda é jovem, e sua principal aplicação ainda está muito limitada a laboratórios de pesquisa, que usam plataformas robóticas padrão para o desenvolvimento de campos como a cognição, a localização e a navegação [2].

Toda essa recente exposição gerada pelas notícias e por filmes de ficção científica acerca do tema de robôs tem contribuído com o surgimento de entusiastas do assunto, tanto dentro, quanto fora do meio acadêmico. Essas pessoas, com o intuito de desenvolver esse campo, têm criado cada vez mais desafios e competições robóticas. Esses tipos de evento reúnem pesquisadores e desenvolvedores de vários países, que trazem novos métodos e técnicas para solucionar os problemas apresentados. Esses desafios geralmente englobam temas relevantes e que não apresentam soluções fechadas sobre a robótica móvel, apresentando sempre a oportunidade dela crescer.

Esses eventos têm se tornado cada vez mais presentes. Com competições como a LARC (*Latin American Robotics Competition*) - Competição Latino Americana de robótica [3], a *Winter Challenge* [4], uma competição organizada pela *RoboCore*, uma loja de componentes eletrônicos, e a *RoboGames*[5], as olimpíadas da robótica, que acontecem anualmente, permitindo que um número maior de pessoas participem desse campo do conhecimento e, por consequência, o melhorem.

Isso tem gerado o surgimento de grupos de estudos focados no desenvolvimento de sistemas robóticos que resolvam desafios inteligentes. Esses grupos são, geralmente, conhecidos como equipes de competição, tendo a própria Universidade de Brasília (UnB) algumas dessas equipes. A Divisão de Robótica Inteligente (DROID) é uma delas. Ela existe desde 2009, participando anualmente de

[2] `https://www.jpl.nasa.gov/news/news.php?feature=4516`
[3] `http://www.cbrobotica.org/?lang=pt`
[4] `https://www.robocore.net/eventos`
[5] `http://http://robogames.net/index.php`

várias competições e aproximando os alunos da UnB com o desenvolvimento da robótica móvel e autônoma.

A navegação faz parte da maioria desses desafios, sendo em muitos casos o foco do mesmo. A *RoboMagellan* é um exemplo disso. O objetivo dessa categoria, que faz parte da *RoboGames*, é que robôs autônomos naveguem por um ambiente externo, desviando de obstáculos estáticos, como árvores, e dinâmicos, como pessoas ou carros, para alcançar um marco previamente definido. Esse marco é indicado de duas maneiras: uma coordenada no sistema de posicionamento global (GPS) e um cone laranja sobre a posição desejada. Além disso, existem marcos intermediários que fornecem uma pontuação extra para quem os alcance, incentivando um certo planejamento das atividades . A tentativa de completar o desafio é terminada caso o robô toque e pare no marco final, ou então acabe o tempo máximo da rodada[6].

Para realizar a navegação exigida para essa competição, é possível perceber que uma série de tarefas diferentes deve ser realizada. Um exemplo delas é a localização, o problema de definir a posição e a orientação do robô em relação ao mundo [3]. Intimamente ligado a ele se tem o problema de mapeamento, que consiste em criar uma representação do ambiente no qual o robô se encontra. Em muitos casos ambos são tratados ao mesmo tempo, criando um dos problemas mais famosos da robótica o SLAM, do inglês *Simultaneous Localization and Mapping*. Esse tema é muito pesquisado, e Durrant-Whyte e Bailey apresentam uma boa discussão do assunto, incluindo a explicação de várias dessas soluções em seu trabalho divido em duas partes [4] e [5].

Outra tarefa envolvida é o planejamento de rotas, que deve decidir por quais pontos um robô deve passar considerando o seu conhecimento do mundo, obtido pelo mapeamento. Sabendo por onde ele deve passar, o planejador de trajetória deve calcular quais as velocidades que o corpo deve ter para alcançar os pontos planejados, dada a localização estimada. Essa velocidade deve ser convertida para a atuação correta no sistema de locomoção que pode ser de diversos tipos, como por exemplo um robô aranha multi articulado, ou então um sistema com quatro motores rotacionais como um carro, até mesmo sistemas que sejam aéreos ou aquáticos.

Além disso, ele deve conseguir ler e interpretar sensores, de forma que ele possa reagir ao ambiente caso algo mude, entre muitas outras tarefas que ele precisa realizar para garantir o seu funcionamento. Dessa forma, é possível perceber que é necessária a existência de algo que organize e estruture todo o comportamento do robô. Isso geralmente é feito por uma arquitetura de controle.

As arquiteturas de controle são o que provê o cérebro para o robô de forma que ele possa ser autônomo e alcançar os seus objetivos [6]. São elas que coordenam os comportamentos do robô para que ele possa alcançar metas complexas. Este trabalho visa criar um arcabouço de navegação que possa implementar essas arquiteturas de controle com um objetivo de navegar um robô que seja capaz de resolver a competição apresentada.

[6] `http://robogames.net/rules/magellan.php`

1.2 Definição do problema

Neste trabalho é abordado o problema de organizar e estruturar os diferentes componentes que envolvem a navegação de um robô que seja capaz de completar o desafio *Robomagellan*. Além disso, também se busca desenvolver, em conjunto com outros trabalhos, todas as funcionalidades necessárias. Sendo as funcionalidades que esse trabalho tentará sanar a leitura correta de sensores de distância ultrassônicos, a obtenção de um mapa que represente o ambiente que o robô se encontrará e o planejamento de rotas.

A plataforma robótica utilizada foi desenvolvida em paralelo à elaboração deste trabalho com o auxílio de uma equipe formada por membros da equipe DROID, e sua arquitetura e funcionamento serão brevemente descritos no tópico a seguir.

1.2.1 Arquitetura do robô

O robô construído para a competição se chama Bruce, e pode ser observado na Figura 1.3. Ele foi montado a partir de um chassi *Wild Thumper 6WD*. Esse chassi é não holonômico, o que significa que existem restrições físicas que impedem a modelagem do sistema em relação às coordenadas globais a partir de equações diferencias [7], significando, em termos mais simples, que ele não consegue se mover livremente em qualquer direção. Ele possui seis rodas acopladas par a par com um sistema de amortecimento baseado em molas, de forma que pequenos obstáculos como pedras e desníveis do solo possam ser facilmente superados.

A Figura 1.4 exibe um diagrama que apresenta o funcionamento do Bruce, além de mostrar

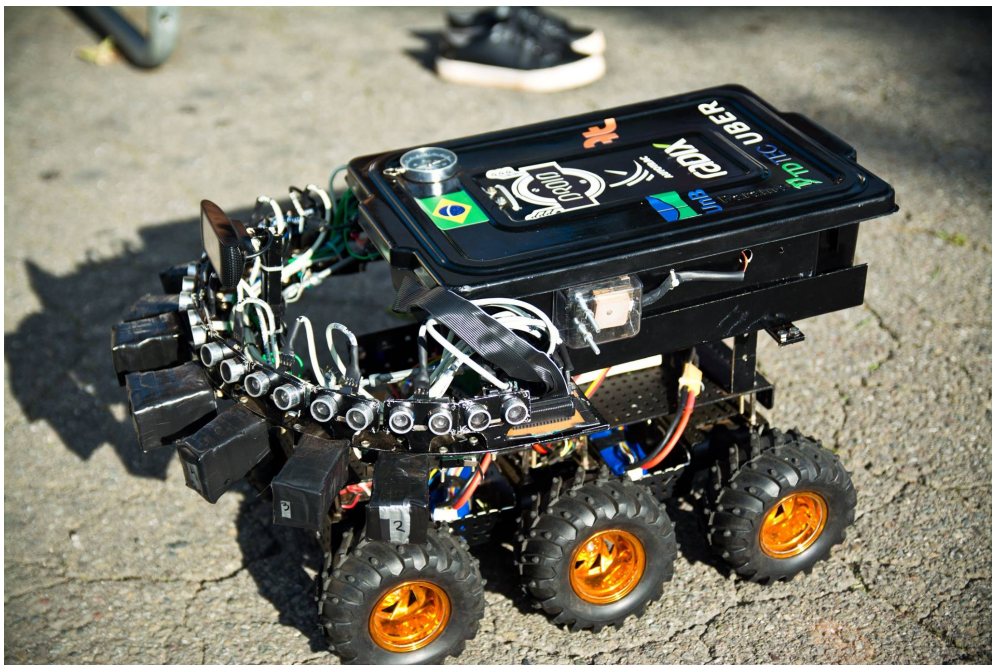


Figura 1.3: Bruce, o robô desenvolvido para a competição *Robomagellan*. - Fonte[7]

[7]Fotografia tirada por João Bosco Gouvêa Ramos

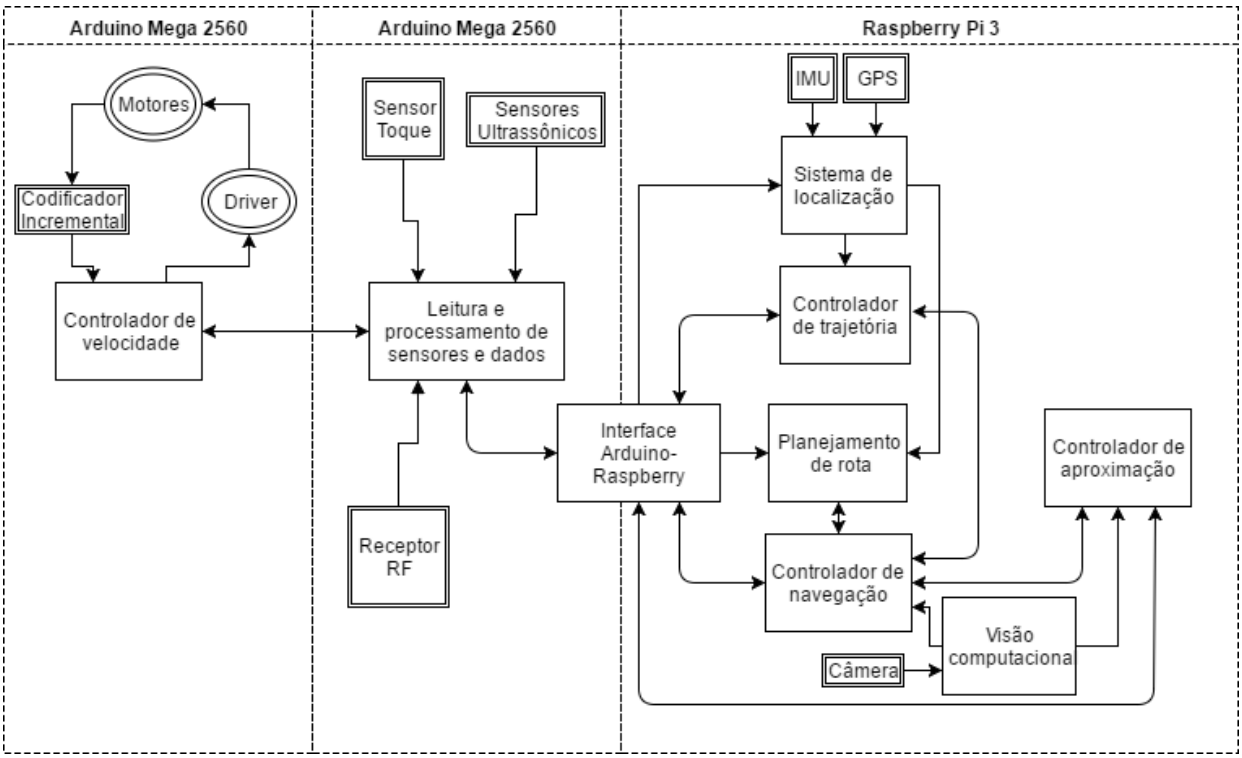


Figura 1.4: Diagrama de funcionamento do robô.

quais os componentes foram utilizados para a sua montagem. Nela, os blocos retangulares de borda simples apresentam os módulos de processamento. Já os de borda dupla representam os sensores utilizados, enquanto que as elipses de borda dupla representam os atuadores utilizados.

O processamento mais pesado, que exige mais poder computacional e memória é feito em um Raspberry Pi 3[8], um computador embarcado que tem um processador de 64 bits de 4 núcleos com uma frequência de relógiol de 1,2GHz. Ele possui uma memória RAM de 1GB e 32GB para armazenar dados. Ele roda uma versão do Linux chamada de Raspbian[9] que provê uma facilidade de utilização dos seus recursos, que incluem 40 entradas e saída de uso geral (GPIO), quatro interfaces USB, adaptadores de Wi-Fi e Bluetooth, entre outros.

O Raspberry é responsável por rodar os algoritmos mais complexos e que exigem maior capacidade de processamento. Ele também cuida de sensores que exigem maior gerenciamento de recursos como a câmera. Os módulos, que estão contidos nele, desenvolvem em sua maioria algoritmos mais abstratos como planejar a rota, ou controlar a trajetória. Eles são coordenados e se comunicam por meio do *Robot Operational System* (ROS), que será explicado posteriormente na Subseção 2.2.1. Essa plataforma foi escolhida, além de todo suporte que ela fornece, pois é compatível com o Raspbian e com o Ubuntu, sistema operacional Linux presente em muitos computadores. Dessa forma, o desenvolvimento pode ser realizado fora do robô, e depois facilmente integrado. Muitos desses módulos foram desenvolvidos em outros trabalhos paralelos, sendo o foco desse trabalho o planejamento de rotas, o controlador de navegação, a interface Arduino-Raspberry e a estrutura

[8]https://www.raspberrypi.org/products/raspberry-pi-3-model-b/
[9]https://www.raspbian.org/

contida no ROS que gerencia tudo.

Para ler a maioria dos sensores foram utilizados dois Arduino Mega 2560[10], um microcontrolador que possui 54 portas de entrada e saída, mas que opera a uma taxa de 16MHz, contando com apenas 8KB de memória RAM. Um deles tem o propósito de cuidar da locomoção do robô. Ele lê os codificadores incrementais e implementa o controlador de velocidade, desenvolvido em outro trabalho.

Essa informação é passada para o outro Arduino, que é responsável por fazer a leitura de todos os sensores e passá-las para o Raspberry por meio da interface desenvolvida. Ele também recebe comandos por meio dessa interface, e do receptor de Rádio Frequência (RF), que devem ser repassados para o Arduino que controla os motores. O desenvolvimento da estrutura que gerencia essa comunicação entre os microcontroladores e os sensores também faz parte do escopo do problema deste trabalho.

Os sensores que fazem parte do sistema possuem todos um objetivo claro. Os sensores ultrassônicos de distância servem para detecção de obstáculos desconhecidos e consequente desvio dos mesmos. O sensor de toque serve para detectar se o robô tocou de fato no objetivo. A Unidade de Medida Inercial (IMU, sigla em inglês), o receptor GPS e a codificador incremental servem para auxiliar a localização, sendo que esse último também é utilizado para ter um maior controle sobre a velocidade na qual os motores estão girando. A câmera é utilizada para detectar o alvo, por meio de algoritmos de reconhecimento de padrão desenvolvidos em outros trabalho. Por fim, o receptor RF serve para transporte do robô e medidas de segurança exigidas pela competição, como por exemplo, parar imediatamente caso seja necessário.

Os únicos atuadores presentes na plataforma são os seis motores, que são controlados com o auxílio de um *driver* de alta potência e seu objetivo é deslocar o robô no seu ambiente.

1.3 Objetivos do projeto

Os objetivos deste trabalho consistem no desenvolvimento de um arcabouço de navegação e sua subsequente implementação em algumas configurações, incluindo um contexto de competição apresentado na *RoboGames*. Portanto, é necessário desenvolver uma estrutura que seja capaz de implementar arquiteturas de navegação, que gerenciam os comportamentos e ações que um robô deve ter para alcançar o objetivo. Isso é feito por meio de um arcabouço dividido em três diferentes níveis. O primeiro nível apresenta uma maneira de descrever o comportamento do robô e é responsável por controlar quando ele vai executar cada coisa. O segundo é onde os comportamentos descrito e controlados na camada anterior são de fato implementados de maneira modular. O último é responsável por lidar com os sensores e atuadores do sistema, e transformar essa interface de baixo nível em um nível de abstração maior, de tal forma que os dois níveis anteriores possam ser o mais independente possíveis da plataforma física do robô.

Para poder implementar esse arcabouço é necessário desenvolver também os módulos que ele

[10]https://www.arduino.cc/

controla, dessa maneira, outros objetivos são a elaboração dos seguintes módulos: o planejador de rota e o tratamento dos valores lidos pelos sensores. O restante deles será construído em outros trabalhos paralelos a este.

1.4 Resultados obtidos

Com o trabalho desenvolvido, foi possível criar e testar a estrutura de um arcabouço de navegação. Foi possível verificar que ele seria capaz de implementar a estratégia para a resolução da competição *Robomagellan*, porém a solução completa não foi realizada devido a problemas com a elaboração de alguns dos módulos necessários.

No entanto, os testes realizados mostraram a versatilidade do arcabouço, apresentando sua expansibilidade, sua reprogramabilidade e a capacidade de ser rapidamente adaptado a outras plataformas físicas, de forma que ele atendeu todas as características básicas desejadas neste trabalho.

Também foi possível mostrar o funcionamento desejado do planejador de rotas desenvolvido, incluindo sua interação com outros módulos desenvolvidos por outros trabalhos, e com o controlador de navegação incorporado na arcabouço.

1.5 Apresentação do manuscrito

Este trabalho está organizado em cinco capítulos. O Capítulo 2 apresenta a fundamentação teórica para a compreensão dos conceitos básicos deste trabalho. Ele é dividido em quatro seções, a primeira explica o conceito de sistemas operacionais para robótica, e qual são seus propósitos. A segunda discorre sobre simuladores, apresenta algumas opções e descreve o simulador escolhido para realização dos testes. A terceira seção conceitua arcabouços de navegação, explicando o que eles são, apresentando os seus métodos de classificação e alguns exemplos de aplicação da literatura. A última seção apresenta a teoria básica de planejamento de rotas, e explica o algoritmo RRT utilizado. O capítulo 3 apresenta a estrutura geral do arcabouço desenvolvido, apresenta o funcionamento de cada uma das suas partes e as regras que devem ser seguidas durante a sua criação. Em seguida, apresenta-se uma implementação desse arcabouço, com o propósito de exemplificar a sua implementação. No Capítulo 4 estão contidos os resultados dos testes realizados para comprovar as capacidades do sistema de navegação proposto. E por fim, o Capítulo 5 contém as conclusões deste trabalho, assim como algumas propostas de trabalhos futuros a partir do arcabouço aqui desenvolvido.

Capítulo 2

Fundamentação

2.1 Introdução

Este capítulo visa apresentar os conceitos e bases teóricas necessários para que o leitor não necessite de um amplo conhecimento sobre os elementos essenciais para o desenvolvimento de um arcabouço de navegação. Ele está dividido em quatro grandes partes, cada uma focando em algum conceito ou método fundamental.

A primeira parte apresenta alguns sistemas operacionais para a robótica, explica o que esse termo significa, apresenta alguns exemplos e por fim explicita um caso específico, que será utilizado neste trabalho, o ROS.

A segunda seção apresenta os simuladores para robótica, o porquê de eles serem necessários e suas funções básica. Apresenta também alguns simuladores disponíveis e detalha um pouco sobre o Gazebo e o motivo dele ter sido escolhido.

Depois é apresentado o conceito de que, de fato, é um arcabouço, do inglês *Framework*, de navegação, de onde esse termo surgiu. O paralelo que ele apresenta com arquiteturas de controle. Além de exemplificar tipos de arcabouços que existem e o que deseja-se alcançar com o arcabouço que será desenvolvido.

Por fim, serão explicados os conceitos e técnicas elementares para um dos módulos de navegação desenvolvidos neste trabalho: O planejador de rotas. Será explicado o que engloba planejar neste contexto, e serão apresentadas todas as ferramentas necessárias para uma compreensão básica do planejador desenvolvido no próximo capítulo.

2.2 Sistemas operacionais para robótica

O termo "Sistema Operacional" para a robótica possui um significado um pouco diferente daquele comumente atribuído à sistemas operacionais. Ele não representa necessariamente um organizador de tarefas e de execução, ou a representação da máquina através de um sistema de arquivos. No entanto, olhando-se as duas definições de uma maneira mais ampla pode-se observar

as similaridades.

Tanenbaum [8, p. 16-17] define sistemas operacionais como duas coisas ao mesmo tempo: Uma máquina estendida, ou seja, algo que faça com que a parte do Hardware fique transparente para o Software. Dessa forma, o programador não precisa se preocupar com as minúcias de como a parte física funciona para poder programar. Ou como um gerenciador de recursos, coordenando quais programas podem acessar o que em qual intervalo de tempo.

Já na robótica esse termo representa, na verdade, uma classe de arcabouços que tem como tarefas principais a organização e facilitação da complexidade de sistemas e um aumento da velocidade na prototipagem de programas, principalmente, para a pesquisa e desenvolvimento de sistemas robóticos [9].

Esses arcabouços são comumente conhecidos como Ambientes de Desenvolvimento Robótico, do inglês *Robotic Development Enviroments* (RDEs) [10]. Eles são desenvolvidos com o intuito de facilitar o desenvolvimento de agentes robóticos, facilitando uma abstração relacionada a implementação física do sistema, permitindo que uma série de programas desenvolvidos possam ser reaproveitados em diferentes plataformas.

Vendo esses objetivos é possível criar um ponte entre os RDEs e os sistemas operacionais, já que ambos querem organizar e facilitar uma maneira mais universal de se criar programas para seus sistemas da maneira mais independente da estrutura física possível.

Existe um grande número de RDEs, sendo vários desses gratuitos ou pagos, alguns com propósitos mais gerais outros com propósitos bem específicos, além de muitas outras características que fazem eles se diferenciarem. Em [10], o autor faz uma comparação extensiva de vários desses RDEs, como por exemplo o TeamBots desenvolvido pela Universidade Carnegie-Mellon e o Player/Stage desenvolvido pela Universidade do Sul da Califórnia.

Em [9], o autor estende essa análise incluindo o ROS - *The Robot Operating System*. Esse foi o RDE escolhido para o desenvolvimento da plataforma apresentada neste trabalho, devido a sua grande comunidade, boa documentação e facilidade de acesso a informações e soluções já prontas. A Subseção 2.2.1 irá explicar em maiores detalhes o que é o ROS e seu funcionamento.

2.2.1 *The Robot Operating System* (ROS)

Esse arcabouço foi desenvolvido em conjunto pela Universidade de Stanford e a Universidade do Estado do Oregon para auxiliar no desenvolvimento de pesquisas na área de robótica ocorrendo em ambos os lugares. Tendo sua primeira versão pública lançada em 2010 como ROS Box Turtle.

Desde então é uma plataforma aberta que recebe a contribuição de pessoas do mundo inteiro e cresce cada dia mais. Estando no momento da escrita deste trabalho em sua 11^a versão, ROS Lunar Loggerhead.

A versão escolhida para desenvolvimento foi a ROS Indigo Igloo, lançada em Julho de 2014, pois é a versão considerada pelos desenvolvedores como mais estável e a que eles disponibilizam suporte por um maior período de tempo, até Maio de 2019. Além de ser a versão com maior

integração com o sistema operacional *Raspbian* do *Raspberry Pi*.

O ROS tem como objetivos filosóficos ser uma plataforma ponto a ponto, baseada em ferramentas, multilíngue, magra e aberta [9]. Esses conceitos representam, respectivamente que ela é capaz de:

a. Funcionar em rede, com um administrador chamado de *roscore* coordenando as interações;

b. Possuir várias ferramentas de suporte, permitindo navegar pelos módulos, visualizar tarefas, criar registros, entre outros;

c. Ser programada em diversas linguagens de programação, incluindo Python, Octave, C++ e outros, sendo que independente da linguagem, todas as partes conversam com facilidade;

d. Ser independente da estrutura física, deixando as partes bem modulares, de tal forma que algoritmos possam ser reaproveitados em outros sistemas;

e. Ser editado por qualquer um, e assim crescer com a comunidade, sem impor custo àqueles que a utilizam.

Para atingir esses objetivos, o ROS possui uma estrutura composta por pacotes, arquivos de lançamento, nós, tópicos e mensagens que seguem duas filosofias principais: servidor e cliente, e propagador e assinante.

Esses elementos e sua funcionalidades serão melhores explicados a seguir e sua descrições foram retiradas de [9].

2.2.1.1 Pacotes

Pacotes, ou *Packages* no ambiente do ROS, são a maneira criada para facilitar o compartilhamento de conhecimento e de informações. Eles possuem uma definição bem relaxada, são basicamente compostos por um diretório que contém um arquivo XML descrevendo o pacote e suas dependências.

Esses pacotes contêm todas as partes realmente funcionais do ROS como por exemplo, definições de mensagens, arquivos de criação dos nós, arquivos de lançamento, entre outros. Eles são compilados e processados individualmente, sendo que o único requerimento deles é que suas dependências estejam presentes no sistema.

O ROS mantém um índice sobre todos os pacotes instalados, facilitando não somente um pacote achar o outro, mas também permite que se chegue diretamente no local de seu computador em que ele se encontra, através apenas de seu nome, com o auxílio da ferramenta *roscd*.

2.2.1.2 Nós

Os nós, ou *nodes*, são as unidades que executam alguma computação. São responsáveis então pela definição do funcionamento do sistema robótico, sendo o padrão do ROS que essa computação

ocorra de maneira granular, ou seja, vários nós separados, cada um com uma função específica.

Esse arcabouço é projetado de maneira que cada nó seja executado independente de outros, ou seja, eles tentam ocorrer de maneira paralela, dependo da máquina em que o sistema esteja rodando para que esse processamento seja realmente paralelo ou não (número de processadores, sistema operacional, entre outros fatores determinantes).

Para que haja coordenação entre os nós são utilizadas os dois paradigmas citados acima, mostrados na Figura 2.1:

a. Cliente e Servidor
   Nessa modalidade, um nó é o servidor que disponibiliza um serviço, o qual um outro nó cliente pode requisitar a qualquer momento recebendo assim uma resposta do servidor;

b. *Publisher* e *Subscriber* - Propagador e Assinante
   Já nessa modalidade, existe um nó propagador que contém uma informação. Toda vez que ele deseja passar essa informação ou atualizá-la, ele pega essa informação, coloca-a em uma mensagem e em seguida publica-a em um tópico. Outro nó que deseje aquela informação deve assinar aquele tópico, recebendo assim um aviso toda vez que uma atualização chega a ele.

O segundo paradigma foi o que surgiu junto com o ROS, tendo sido implementada a ideia Cliente-Servidor posteriormente, e ainda é uma das maneiras mais utilizadas para coordenar o trabalho de vários nós.

2.2.1.3 Mensagens e Tópicos

As mensagens e os tópicos são elementos que compõem a estrutura principal de comunicação do ROS, são o que permite que todos os nós se integrem independente se foram escritos na mesma linguagem ou não, ou até mesmo se não estiverem no mesmo computador.

As mensagens são uma estrutura de dados estritamente e formalmente descrita. Podem conter dados dos tipos primitivos (inteiros, ponto flutuante, booleanos, entre outros), além de vetores e outras mensagens.

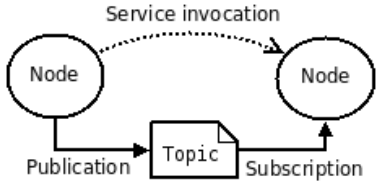


Figura 2.1: Representação dos paradigmas cliente e servidor (linha pontilhada) e Propagador e Assinante (linhas cheias). - Fonte: Wiki ROS[1]

[1] `http://wiki.ros.org/ROS/Concepts`

Os tópicos são estruturas que guardam as mensagens publicadas. São descritos por um nome e o tipo de mensagem que eles guardam. O mesmo tópico pode ter vários nós como propagadores e vários nós como assinantes, assim como um mesmo nó pode publicar e assinar vários tópicos. O interessante desse tipo de troca é que não importa para o propagador quem é seu assinante, e vice versa, o tópico que coordena essa troca.

2.2.1.4 Outras estruturas e ferramentas

Além disso, existem várias outras estruturas e ferramentas essenciais para o funcionamento do ROS, assim como algumas outras mais auxiliares, não necessárias para o funcionamento, mas que ajudam muito os desenvolvedores.

Dentre as essenciais pode-se citar principalmente o *roscore* e o *roslaunch*. O primeiro é quem é responsável por coordenar toda a comunicação entre os nós, ele que administra os tópicos, lida com máquinas externas, dita a execução dos nós, cria registros, entre outras funções necessárias para que tudo funcione corretamente e de maneira transparente. O segundo é um enorme facilitador, permitindo a criação de arquivos de lançamento, dessa forma é possível deixar tudo pré configurado e iniciar vários nós e outros programas necessários para que o seu sistema funcione, tudo isso através de apenas um comando.

Das ferramentas auxiliares é importante citar aqui principalmente a *rosbag* e o *Rviz*. As *rosbag*'s permitem que o usuário grave todas as informações e eventos do sistema a medida que eles aconteçam. Dessa forma, é possível simular exatamente a situação que o sistema vivenciou apenas utilizando uma reprodução da *bag*, um tipo de registro, que o *rosbag* gera.

O *Rviz* é uma poderosa ferramenta de visualização, que permite transformar as mensagens publicadas nos tópicos do ROS em um mapa visual das percepções do robô, onde ele se encontra no seu mundo, os valores lidos em seus sensores, ter uma visão gráfica do estado interno do robô. É muito importante frisar que ele é uma ferramente de visualização e não de simulação.

A Figura 2.2 mostra a visualização da percepção do ambiente de um robô móvel. Onde ele acha que está, o que os seus sensores de proximidade estão enxergando (linha vermelha fina) e o caminho que ele quer percorrer.

Com o uso de todas esses instrumentos, com todas os pacotes criados pela comunidade, contendo *drivers*, algoritmos e outras ferramentas, além da capacidade de integração do ROS com alguns simuladores. Ele tem se tornado um dos RDEs mais versáteis e próximo de ser o que ele almeja, um sistema operacional para a robótica.

2.3 Simuladores para robótica

A robótica é, geralmente, uma área muito cara, envolvendo muitos componentes sensíveis que podem ser danificados quando experimentos falham. Esses componentes fazem parte de sistemas complexos que exigem a coordenação de muitos componentes e muitos sistemas. Desta maneira,

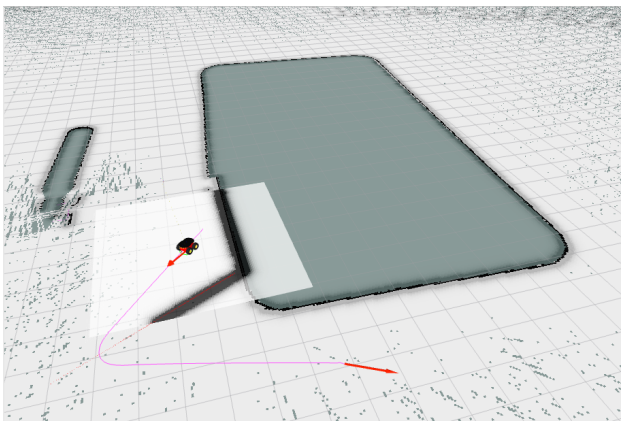


Figura 2.2: Imagem de tela capturada mostrando os dados dos sensores e representação do ambiente do robô no Rviz.

falhas são bem comuns, principalmente em estágios iniciais de desenvolvimento.

Portanto, foi necessária a criação de simuladores que conseguissem representar de uma maneira fidedigna o funcionamento de sistemas robóticos interagindo no mundo real. Surgiram então vários simuladores, como o Gazebo [11], o Darwin2K [12], o OpenSim[2], entre outros, cada um focando em alguma necessidade específica, ou então tentando ser o mais geral possível.

Em [13], foi realizada uma pesquisa buscando classificar e analisar esses simuladores para facilitar outros pesquisadores escolherem qual ferramenta utilizar para sua pesquisa. Essa pesquisa foi realizada com 119 pessoas da área da robótica, abrangendo participantes principalmente dos Estados Unidos, França, Alemanha e Itália. Foi determinado que os simuladores mais utilizados eram o Gazebo e o *Open Dynamics Engine*[3] (ODE), que na verdade é uma biblioteca grátis para a simulação da dinâmica de corpos rígidos articulados, que o próprio Gazebo utiliza.

O Gazebo foi o simulador escolhido para esse trabalho, devido principalmente à sua grande integração com o ROS e sua capacidade de simular as situações necessárias.

2.3.1 Gazebo

O Gazebo começou seu desenvolvimento em 2002 na universidade do Sul da Califórnia tendo em mente principalmente a integração com a plataforma *Player/Stage*, sendo em 2009 integrado ao ROS, se tornando uma das ferramentas mais utilizadas por essa comunidade[4].

Ele possui suporte para quatro diferentes motores de física, *softwares* capazes de simular a física. Eles são: o já citado ODE, o *Bullet*, o *Simbody* e o *DART*. O padrão é o ODE, sendo ele o primeiro utilizado durante o desenvolvimento dessa ferramenta.

A estrutura do Gazebo é bem simples, existe um mundo, que é simplesmente o conjunto de todos os modelos e fatores ambientais como gravidade e luminosidade. Os modelos representam

[2] `http://opensimulator.sourceforge.net/`
[3] `http://opende.sourceforge.net/`
[4] `http://gazebosim.org/`

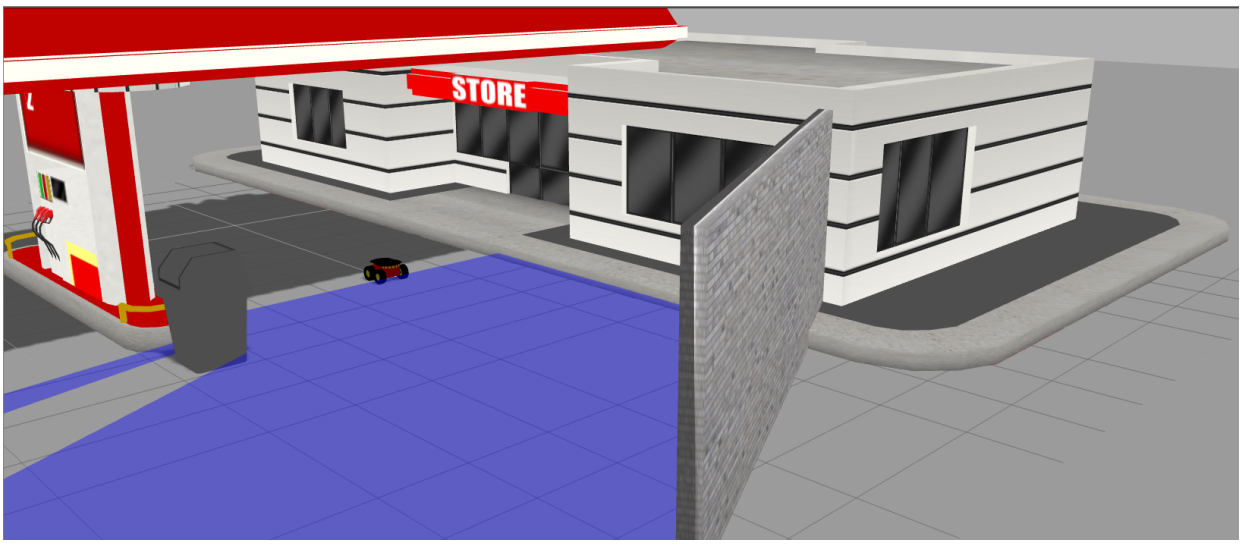


Figura 2.3: Imagem de tela capturada com o Gazebo em funcionamento.

os objetos e podem ser compostos por:

- Corpos, um corpo rígido representado por uma forma geométrica;
- Juntas, a conexão entre os corpos rígidos, podendo representar vários tipos de movimentos, como lineares, rotacionais, entre outros;
- Sensores, um objeto sem representação física capaz de ler dados sobre a simulação.

A representação gráfica gerada pelo Gazebo desse mundo, incluindo o modelo de uma parede, um posto de gasolina e um robô, pode ser vista na Figura 2.3.

Cada modelo é controlado a partir de uma interface que pode mandar comandos para as juntas, mandando elas se movimentarem, assim como ler os dados de sensores. Isso é feito de tal maneira que o Gazebo não fica preso a um cliente específico como o ROS ou o *Player*, podendo ter seu uso expandido[11].

Sua integração com o ROS é feita a partir de um conjunto de *plugins* ligados aos modelos, que transforma as entradas e saídas dessa interface em tópicos que podem, respectivamente, ser publicados ou assinados, fazendo seu uso ser simples e facilmente substituível pelo robô real.

2.4 Arcabouços de navegação

Arcabouços ou arquiteturas de navegação são estruturas organizacionais que tem como intuito coordenar todas as ações relacionadas ao movimento e deslocamento do robô. Eles abrangem desde o movimento controlado, por meio de telecomandos, até o movimento autônomo, envolvendo ou não planejamento, mapeamento e outras tarefas inerentes à navegação.

Essa classe de soluções na verdade é uma subclasse de arquiteturas de controle, que são arquiteturas que definem não apenas a movimentação, mas sim um conjunto de ações que o robô deve

tomar para cumprir um determinado objetivo, ou que simplesmente definem o comportamento em geral do sistema quando inserido em seu ambiente.

No entanto, como em muitos robôs móveis, o único tipo de ação que se pode tomar são aquelas de navegação e tarefas a ela associadas, grande parte da literatura sobre arquiteturas de controle pode ser aproveitada, assim como grande parte dos requisitos e características. Desta forma, permitindo que se faça uma análise extrapolada dela para a de navegação.

Iniciando pela consideração que o sistema seja autônomo, [14] afirma que a estrutura organizacional de um robô deve ser capaz de antecipar e se adequar às situações de maneira apropriada. Ele deve ser capaz de ter resposta em tempo real aos eventos e ser capaz de tomar decisões e definir suas ações em tempo de execução. Para isso os autores enumeram um conjunto de propriedades, listadas a seguir, que o sistema deve ter para atingir esses objetivos.

i. Programável
   Um robô não deve ser criado para um único e específico ambiente e objetivo, ele deve ser capaz de realizar várias tarefas descritas em um nível abstrato. Suas funções devem ser facilmente combináveis para realizar a tarefa a ser executada;

ii. Autonomia e adaptabilidade
   O sistema deve ser capaz de mudar a tarefa sendo executada de acordo com a situação, de tal forma que ela fique mais de acordo com o objetivo e contexto atual;

iii. Reativo
   O robô deve ser capaz de responder aos eventos em tempo real, ou seja, ter uma resposta rápida o suficiente para lidar de maneira apropriada com o acontecimento, levando em conta principalmente seus objetivos que incluem sua própria segurança;

iv. Comportamentos consistentes
   As ações tomadas pela arquitetura devem ser consistentes com o atual objetivo do sistema;

v. Robustez
   O arcabouço deve ser capaz de perceber e sobrepor falhas que ocorram durante a execução;

vi. Extensível
   A adição de novas funções, comportamentos e ações deve ser fácil.

É interessante considerar essas propriedades no desenvolvimento de arcabouços de navegação e arquiteturas de controle em geral. Entretanto, a estrutura desenvolvida em [14] é apenas uma possível arquitetura. Vários autores têm diferentes estratégias de desenvolvimento e execução de arcabouços de navegação.

Em [15], é apresentada uma estrutura de laços de controle aninhados, sendo cada nível de controle responsável por responder a determinados tipos de eventos. Já [16], apresenta uma abordagem usando lógica *fuzzy*, ele pega as leituras do sensores, as transforma na lógica *fuzzy* e as mapeia em uma biblioteca de comportamentos. Outras arquiteturas podem ser observadas em [17, 18, 19, 20].

Mesmo com todas essas diferentes arquiteturas, elas podem, ainda assim, ser classificadas de acordo com a maneira à qual elas reagem ao ambiente. Mataric [6] apresenta quatro diferentes tipos de classificação: reativa, deliberativa, híbrida e baseada em comportamento. Seus conceitos e explicações serão dados a seguir baseados em sua obra.

- **Arquitetura reativa**

  A arquitetura reativa é um dos métodos mais utilizados para o controle de robôs. Ela não requer uma representação interna do ambiente e realiza apenas um mapeamento entre os sensores e os atuadores, como representado na Figura 2.4.

  Nesses sistemas é criado um conjunto de regras simples que não exijam muito tempo de processamento, de tal forma que a resposta ocorra em tempo real. Para isso é necessário que para cada combinação entre os possíveis valores dos sensores exista somente uma possível regra, evitando assim conflitos na hora de decidir como atuar. Em resumo a regra é: sentir e agir.

  Para criar sistemas reativos mais complexos é possível adicionar estados, que representem a evolução do ambiente, e incluí-los simulando uma entrada sensorial no conjunto de estímulos relacionados às regras.

  O grande problema dessa arquitetura é que ela é rígida, as regras devem ser simples e rápidas, fazendo com que a criação de um sistema, que reaja a diferentes situações e consiga trabalhar com novos contextos, seja um desafio muito grande e na maioria dos casos intratáveis devido a explosão combinatorial de entradas que o sistema acaba tendo.

- **Arquitetura deliberativa**

  A arquitetura deliberativa surgiu a partir da inteligência artificial clássica. A deliberação é relacionada com o ato de pensar e pesar as ações antes de realizá-las. Isso implica que as entradas sensoriais passam por um planejador que decide o que o robô deve fazer, ou como e para onde ele deve se mover, e somente após isso atua, como representado na Figura 2.5.

  A deliberação exige planejamento e isso geralmente demanda poder computacional e tempo, como será explicado na Seção 2.5. E é nesse ponto que se encontra um dos maiores problemas dessa arquitetura.

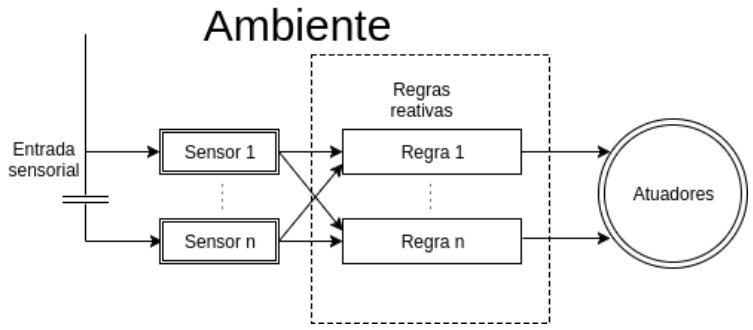


Figura 2.4: Diagrama de Arquitetura Reativa.

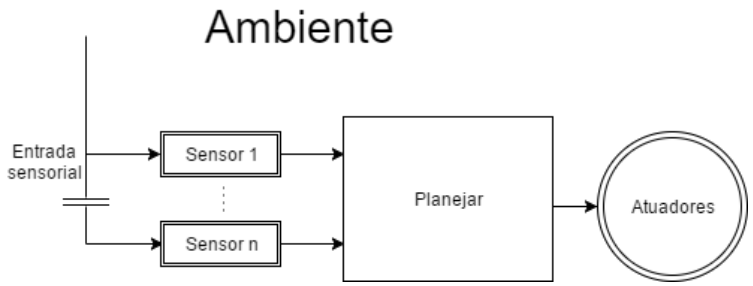


Figura 2.5: Diagrama de Arquitetura Deliberativa.

O planejamento, em geral, exige que o robô tenha uma representação interna do seu ambiente. Para isso, ele deve ser capaz de colher as informações necessárias para gerar essa representação. Isso já pode ser um grande problema, pois geralmente existem ruídos e imprecisões associadas aos sensores que atrapalham sua geração.

Se essa representação estiver ruim, o plano gerado não será de muita valia tornando todo o processo inútil. Além disso, o planejamento pode consumir muita memória vasculhando os possíveis cenários para a criação de um plano, tornando necessário um computador bem potente para processá-lo.

Mas o maior problema de todos com essa arquitetura é que mesmo se tudo ocorrer bem, a representação estiver boa o suficiente, exista memória para o processamento do plano e os sensores conseguiram captar a informação necessária. Se o ambiente não se manter estático durante a execução do plano, o robô tem que parar e replanejar suas ações, tornando-o inadequado para lidar com situações que envolvam tempo real.

- **Arquitetura híbrida**

  A arquitetura híbrida surgiu para tentar incorporar as qualidades e suprir os problemas das duas anteriores por meio da combinação de ambas. Dessa forma, ela tem a velocidade da arquitetura reativa combinada com a capacidade de planejar da deliberativa.

  Esse novo paradigma cria no entanto um novo problema. Como combinar essas duas de forma correta e eficiente? Uma das formas de se fazer isso é através da adição de uma camada intermediária que consiga gerenciar e coordenar as ações reativas imediatas com as ações a longo prazo decididas pelo planejador na camada deliberativa. A Figura 2.6 representa uma das possíveis maneiras na qual esse paradigma pode se organizar.

  Os problemas dessa arquitetura estão justamente na nova camada, a camada intermediária, e nos desafios que ela deve superar. Ela tem que ser capaz de compensar a limitação das outras duas camadas, conciliar as diferentes escalas de tomada de decisões, lidar com as diferentes representações e definir o que fazer quando ambas outras camadas mandam comandos contraditórios para o robô. Resumidamente, ela deve saber a hora de correta de usar cada um dos níveis.

  No entanto, isso também é o seu ponto forte. Agora o sistema consegue planejar e aprender, mas não precisa ficar parado fazendo isso. Ele consegue também lidar com mudanças no

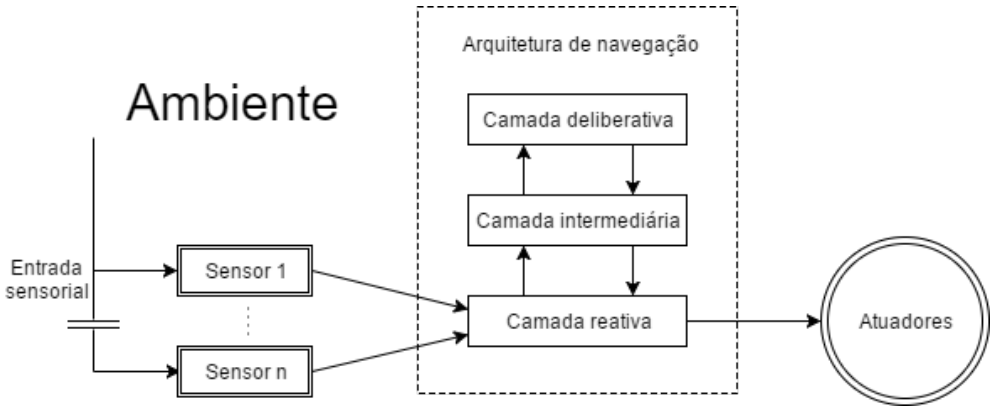


Figura 2.6: Diagrama de Arquitetura Híbrida.

ambiente e reagir de uma maneira apropriada enquanto ele espera um novo plano ser formado. Além de não ficar preso aos limites encontrados na arquitetura puramente reativa.

- **Arquitetura baseada em comportamento**

  A arquitetura baseada em comportamento tenta conciliar as funcionalidades de ambas as arquiteturas reativa e deliberativa, mas tentando ser menos complexa do que a híbrida. Esse arcabouço é derivado do reativo e é inspirado em sistemas biológicos, buscando dividir o controle em vários módulos simples que cuidam de diferentes comportamentos.

  Assim, a arquitetura baseada em comportamento se resume à implementação de vários comportamentos e a combinação deles, como mostrado na Figura 2.7. Portanto, para entender essa arquitetura, é importante definir o que é um comportamento.

  Existem várias maneiras de codificar e implementar os comportamentos, sendo esse um dos pontos fortes desse arcabouço. No entanto, existem restrições e regras que devem ser seguidas na hora de criar um comportamento.

  Um comportamento deve ter objetivos para completar ou manter. Ele deve se estender ao longo do tempo, não sendo instantâneo. Comportamentos devem ter a capacidade de receber entradas de sensores e de outros comportamentos, da mesma forma que suas saídas podem

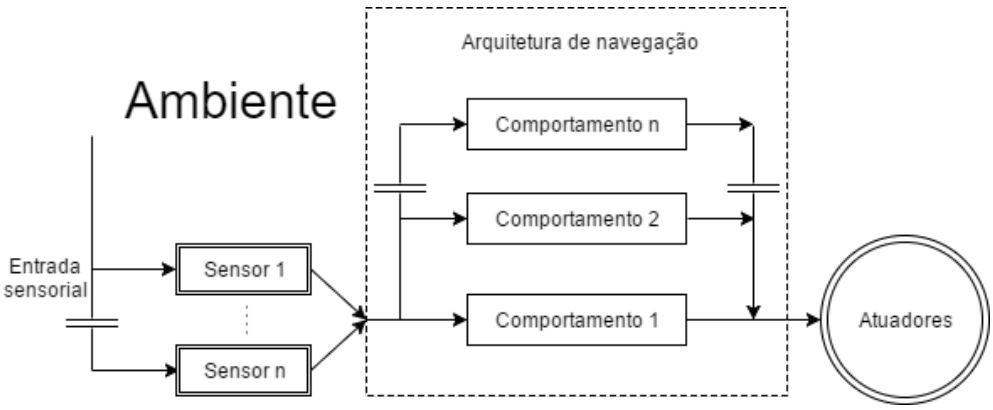


Figura 2.7: Diagrama de Arquitetura baseada em comportamento.

ser para atuadores ou para outros comportamentos. Dessa forma, eles podem se comunicar entre si. E finalmente, comportamentos devem ser mais complexos do que ações.

Como essa arquitetura não é um dos focos desse trabalho não serão dados mais detalhes sobre ela. Caso o leitor se interesse informações mais detalhadas podem ser encontradas no capítulo 16 de [6].

Após analisar essas diferentes classes de arquiteturas, procurou-se desenvolver uma estrutura que pudesse incorporar as propriedades apresentadas por [14] e que permitisse a implementação das diferentes classes de arquitetura apresentada. Dessa forma, foi criado uma arcabouço de navegação que foca principalmente em ser programável, autônomo, adaptável e extensível. A robustez, reatividade e consistência do comportamento depende de como ele é utilizado.

2.5 Planejamento de rotas

Planejamento é um termo com significados muito diferente para diferentes grupos de pessoas [21]. Neste trabalho, será adotado o significado de planejamento de rotas como um algoritmo que seja capaz de determinar as ações que um agente deve tomar para sair de um ponto inicial e chegar a um ponto final.

Escolher o caminho a percorrer é uma parte muito importante de um arcabouço de navegação e está muita associada à parte deliberativa do processo. Como deliberar representa computar algo para máquinas, é necessário que exista uma descrição formal do plano, do ambiente e do que o sistema pode fazer. LaValle[21], chama esses componentes como os ingredientes básicos do planejamento. Sendo eles:

- o estado, algo que representa a situação do ambiente em um determinado momento. Todas as possíveis representações do ambiente formam juntas o espaço de estados, que podem ser tanto discretas quanto contínuas;
- as ações, que representam o que o sistema pode fazer e como elas afetam o mundo ao seu redor e mudam o estado;
- o tempo, sempre está envolvido com o planejamento, seja de forma direta, na qual ele deve ser levado em conta na hora de decidir o que fazer, ou de uma maneira indireta, na qual as ações devem ter um certa ordem sequencial;
- os estados inicial e final, os problemas de planejamento, especialmente os de planejamento de rota, geralmente são relacionados com a partir de um estado inicial específico, procura-se encontrar um conjunto de ações que leve ao estado final;
- um critério que determine o objetivo do plano, ele pode ser viabilidade, verificar se existe um caminho que alcance o estado final, e otimização, achar dentre os caminhos que existe, o melhor deles segundo algum parâmetro definido para o problema.

É importante notar que descobrir um plano viável, principalmente em um curto espaço de tempo, já é um desafio em alguns casos. Portanto, não necessariamente se busca o plano ótimo. Ter um que seja próximo do ótimo é, muitas vezes, mais do que o suficiente. Isso se deve, normalmente, ao fato de que o peso computacional extra requerido com a determinação do caminho ótimo nem sempre vale o ganho de desempenho associado.

O espaço de estados associado ao problema pode ser de natureza contínua ou discreta. Isso é um dos fatores determinantes para saber se o planejamento é de uma dessas duas naturezas. O problema aqui apresentado é o de planejamento de rotas, ou seja, é um problema do mundo contínuo, porém existem maneiras discretas de resolvê-lo sem muita perda de informação.

Os textos a seguir apresentarão uma breve explicação e conceituação sobre os planejamentos discretos e contínuos, respectivamente, de modo que o leitor possa se familiarizar com os conceitos num ambiente mais simples antes de passar para conceitos mais avançados.

É importante ressaltar que todos os conceitos apresentados nessa seção, e suas respectivas subseções foram retiradas de [21] e [22].

2.5.1 Planejamento discreto

O planejamento discreto como descrito nesta subseção é muitas vezes chamado de solucionador de problemas no campo da inteligência artificial. Russel e Norvig [22] definem ambos como coisas diferentes, porém LaValle [21] aponta que essas diferenças não são tão grandes e opta por tratar os dois como a mesma coisa. Essa será a estratégia adotada neste trabalho, no entanto, é importante frisar que os solucionadores de problemas tem conceitos que abrangem uma classe de problemas maior do que aqueles necessários ao planejamento.

A formulação discreta de um modelo de planejamento pode ser feita utilizando o modelo de espaço de estados. Dessa forma, cada situação na qual o mundo se encontra pode ser chamada de um estado, representado por x. O conjunto de todos os possíveis estados é chamado de espaço de estados, X. O espaço de estados para o caso discreto pode ser infinito, mas deve ser contável.

A escolha da representação dos estados é de suma importância para resolução do problema. Informações desnecessárias devem ser deixadas de fora do modelo para evitar dificultar a busca de uma solução, e ao mesmo tempo informações relevantes devem ser inclusas para que uma melhor solução seja encontrada. Nem sempre a linha entre o que é relevante ou não é clara.

O mundo pode ser alterado por meio de ações, u, escolhidas pelo planejador. Cada uma delas quando aplicada ao estado atual x produz um novo estado x'. Essa transformação é representada por

$$x' = f(x, u). \tag{2.1}$$

Para cada estado, um conjunto de ações pode ser aplicado, esse conjunto é chamado de espaço de ações, representado por $U(x)$. O espaço de ações de dois estados x e x' não é necessariamente

disjunto, as mesmas ações podem ser aplicadas em vários estados. Portanto, é definido um conjunto U, tal que ele inclua todas as possíveis ações de todos os estados.

$$U = \bigcup_{x \in X} U(x) \tag{2.2}$$

Redefinindo a Equação 2.1, como f^*, de tal forma que sua entrada seja um estado x e um vetor $\bar{u} = [u_1\ u_2\ ...\ u_n]$, tal que, $u_i \in U$ com $i = 1, 2, .., n$. O seu estado resultante x' pode ser calculado de forma recursiva utilizando a função f em cada termo do vetor $\bar{u}$ sucessivamente.

$$x' = f^*(x, \bar{u}) \tag{2.3}$$
$$x_1 = f(x_0, u_1) \tag{2.4}$$
$$x_2 = f(x_1, u_2) \tag{2.5}$$
$$\vdots$$
$$x' = f(x_{n-1}, u_n) \tag{2.6}$$

Dessa forma, o planejamento se resume a uma busca por um vetor de ações $\bar{u}$ que ao passar pela equação de transição de estados expandida, f^*, a partir de um estado inicial x_i forneça um estado desejado x_G atendendo uma série de restrições impostas ao problema.

Esse problema pode, também, ser modelado como uma busca em um grafo, como representado na Figura 2.8, em que cada estado é um nó. A função de transição de estados fornece os possíveis arcos que conectam os nós, ou seja, as ações. E a função de transição expandida são os nós e arcos no caminho do estado inicial, em verde, até o estado final, em vermelho.

Este exemplo já apresenta um dos grandes problemas desse tipo de busca. Existe mais de um possível caminho que sai do estado x_1 e chega ao estado x_5. Isso significa que um planejador

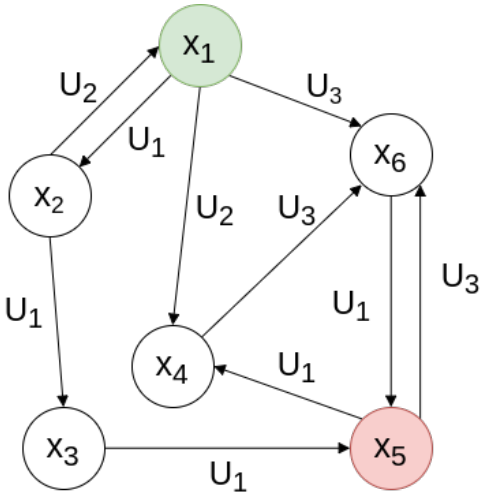


Figura 2.8: Grafo representando um espaço de estados, suas possíveis ações e transições.

poderia dar como resposta um plano composto por $\bar{u} = \{u_1, u_1, u_1\}$, ou então, $\bar{u} = \{u_2, u_3, u_1\}$. Como escolher qual deles utilizar? O que determina se um plano é melhor que o outro?

Além disso, o que fazer caso existam restrições associadas ao problema? Por exemplo, se o estado x_6 for um estado indesejado e não se possa passar por ele, ou então exista alguma penalidade associada a utilizar alguma ação em particular.

Para poder responder e entender melhor essas perguntas, serão analisados alguns métodos para encontrar as soluções desses problemas apresentados por Russel e Norvig [22], porém contextualizados para o planejamento de rotas.

2.5.1.1 Planejamento de rotas discreto

Para melhor entender os conceitos e problemas associados ao planejamento de rotas será apresentada uma contextualização do problema e a apresentação de alguns métodos de solução, incluindo a indicação de outros métodos que não serão analisados mais detalhadamente.

Uma maneira simples de modelar o problema de planejamento de rotas de maneira discreta é atribuir que cada estado é uma posição em um espaço bidimensional, $x = (p_x, p_y)$. E as possíveis ações seriam se mover uma unidade em um dos dois eixos. Considerando a função de transição de estados uma soma, existem no problema quatro possíveis ações $u_1 = (1, 0)$, $u_2 = (-1, 0)$, $u_3 = (0, 1)$ e $u_4 = (0, -1)$. Logo, o modelo matemático é:

$$X = \mathbb{Z}^2 \tag{2.7}$$

$$U = U(x) = \{(1, 0), (-1, 0), (0, 1), (0, -1)\} \tag{2.8}$$

$$f(x, u) = x + u \tag{2.9}$$

Portanto, nessa versão bem simplificada do problema, basta encontrar um vetor de possíveis ações $\bar{u} = {}_0u_{j_0}, {}_1u_{j_1}, \ldots, {}_nu_{j_n}$, com ${}_iu_{j_k} \in U$, que resolva a seguinte função de transição estendida f^*.

$$x_G = f^*(x_i, \bar{u}) = x + \sum_{i=0}^{n} u_i \tag{2.10}$$

Dessa forma, a solução é uma combinação linear das possíveis ações dessa modelagem que satisfaça a Equação 2.10, dada por

$$\bar{u} = au_1 + bu_2 + cu_3 + du_4,\ a, b, c, d \in \mathbb{N}. \tag{2.11}$$

No entanto, essa modelagem é muito simples. Ela não inclui a possibilidade da existência de obstáculos, ela não considera velocidade, tempo, e muitos outras características. Expandir essa abordagem para incluir todas essas outras variáveis pode chegar em modelos matemáticos muito complexos ou simplesmente não modeláveis.

Além disso, ela não responde qual dessas possíveis opções escolher, ou qual delas é a melhor. Para isso, seria necessário acrescentar alguma forma de otimização ao sistema, ou alguma estratégia de escolha.

Essa modelagem é equivalente ao grafo apresentado na Figura 2.9. Para resolver esse problema utilizando a busca em grafos, sem a utilização de informações extras, existem vários métodos, como por exemplo, a busca em profundidade, a busca em largura, a busca de custo uniforme, a busca bidirecional, a busca em profundidade limitada, entre outras.

No entanto, todas essas diferentes abordagens são variações do mesmo algoritmo, a busca geral (Algoritmo 1). Esse algoritmo começa inserindo o estado inicial em uma lista Q, e então percorre-a até não existirem mais estados nela. A cada passo ele verifica se o nó que ele pegou da lista é o objetivo. Se for, o objetivo foi alcançado e o resultado é o caminho até ali. Se não, ele explora todas as possíveis ações a partir do nó atual, verificando quais os estados provenientes das mesmas. Se esse estado não tiver sido visitado, ele é adicionado à lista Q. Caso contrário, algo deve ser feito para evitar que o algoritmo entre em um ciclo reavaliando estados mais de uma vez.

Todos os diferentes métodos apresentados se diferem, normalmente, em um mesmo quesito do geral: a estratégia de escolha do próximo estado a ser avaliado. Essa pequena mudança tem consequências enormes, influenciando se a busca consegue ser completa,ou seja , achar um resultado, se ela é ótima, quanto de memória é necessário utilizar, a velocidade da busca, entre outros aspectos.

A escolha da estratégia determina qual das possíveis soluções será a escolhida, inclusive permitindo que, no caso da busca uniforme aplicada ao planejamento de rotas, o caminho ótimo seja encontrado. Isso ocorre pois essa estratégia permite que sejam associados custos às ações, dessa forma, a estratégia de escolha sempre pega o caminho com o menor custo, conseguindo assim o caminho ótimo até o objetivo.

O grande problema de obter esse caminho ótimo é, como mencionado anteriormente, o elevado custo computacional associado. O custo temporal associado ao algoritmo de busca de custo

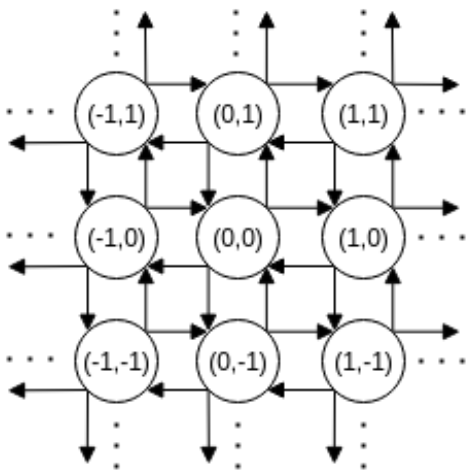


Figura 2.9: Grafo representando a modelagem de um caso simples de planejamento de rotas.

Algoritmo 1: Algoritmo de busca geral

```
1  Q.Inserir(x_i) e marcar x_i como visitado ;
2  enquanto Q não vazio faça
3      x ← Q.pegarProximo();
4      se x ∈ X_G então
5          retorna SUCESSO
6      para todo u ∈ U(x) faça
7          x' ← f(x, u) se x' não visitado então
8              Marcar x' como visitado;
9              Q.inserir(x');
10         senão
11             Resolver x' duplicado;
12 retorna FALHA
```

uniforme é da ordem de $O(b^m)$, sendo b o nível de ramificação da árvore e m a profundidade da mesma. Dessa forma, se no simples problema modelado pela Equação 2.7, deseja-se alcançar o ponto (100,0), a partir do ponto (0,0), o custo computacional seria da ordem de 10^{60}.

Entretanto, existem várias características e métodos para diminuir significantemente esse espaço de busca. Um dos passos essenciais para lidar com esse problema é tratar os estados já visitados. Um exemplo disso pode ser visualizado na Figura 2.10. Nesta figura está exemplificada a execução do algoritmo de busca uniforme, assumindo que todas as ações tem o mesmo peso. O ponto (0,0) é o estado inicial, e o ponto (2,0) o estado final. Os nós verdes são os nós explorados. Os laranjas, os nós que já haviam sido explorados e, portanto, não há necessidade de expandir. E os em azul o caminho que leva ao objetivo.

A escolha de não expandir nós repetidos diminui de 21 para 13 o número de estados analisados. Essa diferença se torna ainda mais notória quando se pensa no caso de chegar ao ponto (100,0). O custo computacional cai da ordem de 10^{60} para uma ordem de 10^4. Isso é especialmente verdade em casos de planejamento de rotas, que consideram apenas as posições como estados, pois o espaço de estados pode se tornar um mapa. No caso do planejamento ser bidimensional, o mapa que o representa também o será, como o representado na Figura 2.11.

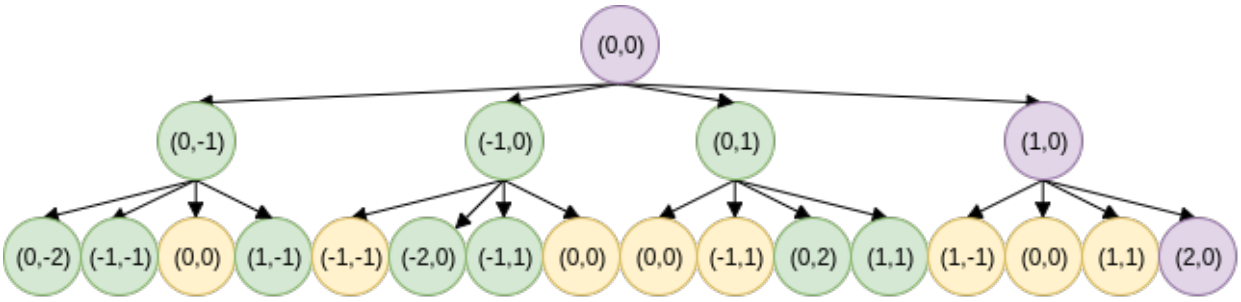


Figura 2.10: Execução da busca uniforme em um grafo representando um mapa.

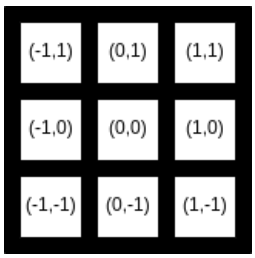


Figura 2.11: Grafo do espaço de estados representado como um mapa.

Essa representação de mapa permite ver que a complexidade temporal do algoritmo cai para uma $O(l \times c)$, sendo l e c as dimensões do mapa. Esse tipo de função cresce bem mais lentamente comparada a uma exponencial, como a $O(b^m)$. Essa representação traz também outros benefícios, como por exemplo, a possibilidade de associar custos diferentes à vários pontos do mapa. Permitindo que obstáculos sejam incluídos de maneira simples, incorporando à um nó um custo infinito ou proibitivo. Isso é chamado de mapa de custo e um exemplo pode ser observado na Figura 2.12.

Outra maneira de agilizar a busca, em boa parte dos casos, é por meio da utilização de heurísticas para a escolha de qual estado explorar. Heurísticas são funções que representam, de maneira numérica, uma informação adicional que indica a estimativa do custo aproximado de um ponto até o objetivo. Elas devem seguir uma série de regras que não serão explicadas aqui, no entanto, Russel e Norvig [22] discorrem bastante sobre o assunto.

Existem vários algoritmos que utilizam a heurística para agilizar a busca e, inclusive, alguns conseguem alcançar uma solução ótima. Alguns deles são, a busca gulosa, o algoritmo A*, o *Iterative deepening A** (IDA*) [23], o D* [24], dentre outros.

Apesar desses algoritmos serem bem rápidos, eles ainda apresentam um custo significativo, principalmente ao imaginar um cenário onde o ambiente de busca é muito grande. O que é uma verdade para o planejamento de rotas. Considere que deseja-se representar uma área de 300 por 300 metros e que cada estado no mapa represente uma área de 20 por 20 centímetros. Isso totaliza um total de 2.250.000 estados, uma quantidade significativa que mesmo esses algoritmos rápidos podem demorar alguns segundos para calcular.

No entanto, deve-se lembrar que os computadores embarcados possuem uma capacidade de processamento menor. Transformando esses alguns segundos para algumas dezenas de segundos. O que é um tempo inaceitável para lidar com o ambiente dinâmico em que o robô existe. Por isso,

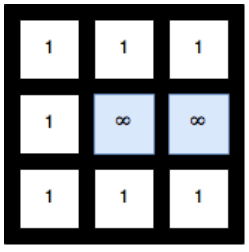


Figura 2.12: Mapa de custo representando um obstáculo como custo infinito.

serão analisado agora os planejadores contínuos para buscar alternativas.

2.5.2 Planejamento contínuo

O planejamento contínuo de rotas é a versão que lida com o espaço de estados que melhor representa o mundo em que vivemos, o contínuo. As técnicas apresentadas para o caso discreto não podem ser diretamente aplicadas aqui, pois esse espaço não é contável. Não existe um valor de uma unidade que possa representar o passo entre um estado e o outro. Independente de quão pequeno for esse passo, sempre existirá um infinidade de outros estados entre eles.

LaValle [21] afirma que existem duas principais filosofias para lidar com esse tipo de problema. O planejamento de movimento baseado em amostras e o planejamento de movimento combinatorial. No entanto, serão analisados apenas os planejadores baseados em amostras, pois são um tema amplamente pesquisado nos últimos anos e apresentam soluções com um relativo baixo custo computacional associado.

Porém, para entender esses planejadores é necessário explicar o que é o espaço de configuração. Ele é considerado como um conjunto de possíveis transformações que podem ser aplicadas ao robô, baseado na mecânica Lagrangiana e no trabalho de Lozano-Pérez [21]. Esse é o espaço de estados para o planejamento contínuo e é geralmente chamado de espaço $\mathcal{C}$.

O espaço $\mathcal{C}$ de corpos rígidos bidimensional é o SE(2). Ele é uma combinação do espaço translacional $\mathbb{R}^2$ com o espaço rotacional $\mathbb{S}^1$. Essa frase pode parecer estranha e não significar muito para alguém que não conheça do assunto. Mas isso significa que o espaço de estados é composto por uma combinação das duas possíveis transformações que um corpo bidimensional pode sofrer: a translação e a rotação. O espaço $\mathcal{C}$ de corpos tridimensionais é o SE(3), muito mais complexo que o SE(2), pois necessita levar em conta a rotação ao redor dos 3 eixos existentes.

Entretanto, esses espaços euclidianos especiais, do inglês *special euclideans spaces*(SE), não serão aprofundados aqui, porque na verdade será trabalhado o espaço euclidiano $\mathbb{R}^2$. Essa escolha foi feita por causa de dois principais fatores. Primeiro, o robô não consegue atuar no eixo perpendicular à superfície terrestre. As suas variações de altura não são significantes para o planejamento de rota. Além disso, desníveis mais significativos, que a estrutura não conseguiria sobrepor, podem ser representados como obstáculos. E segundo, a rotação para garantir que ele alcance o objetivo será tratada em uma outra parte do sistema de navegação.

A diminuição do espaço de configurações de SE(3) para $\mathbb{R}^2$ atenua bastante a complexidade do problema, reduzindo assim o custo computacional associado. Além disso, essa representação não apresenta perdas significativas de informação para o sistema. Dessa forma, a modelagem do sistema fica bem mais simples e se assemelha àquela apresentada pelo planejador de rotas discreto. Com a diferença que o espaço de estados é o $\mathbb{R}^2$ e as ações que o sistema pode escolher são as transformações translacionais que também estão contidas em $\mathbb{R}^2$.

Os obstáculos do planejamento de rotas contínuo são representados pela remoção de elementos de $\mathcal{C}$. Os elementos não removidos são aqueles livres que devem ser escolhidos para que um caminho seja considerado seguro. Isso divide o espaço $\mathcal{C}$ em dois. A região dos obstáculos, que é conhecida

como $\mathcal{C}_{obs} \subseteq \mathcal{C}$, é definida na Equação 2.12. Isso significa que a região $\mathcal{C}_{obs}$ é composta de todas as configurações q nas quais o corpo rígido do robô $\mathcal{A}$ não colide com o conjunto $\mathcal{O}$ que define a posição dos obstáculos.

$$\mathcal{C}_{obs} = \{q \in \mathcal{C} | \mathcal{A}(q) \cap \mathcal{O} \neq 0\} \tag{2.12}$$

A região livre, que é por onde o robô deve passar é conhecida como $\mathcal{C}_{free}$ e é definida como a região $\mathcal{C}$ sem $\mathcal{C}_{obs}$.

$$\mathcal{C}_{free} = \mathcal{C} \setminus \mathcal{C}_{obs} \tag{2.13}$$

No espaço $\mathbb{R}^2$, no qual será trabalhado, é possível ter um visualização gráfica que representa bem o espaço $\mathcal{C}$. Ela é conhecida como mapa de custo contínuo, ilustrado na Figura 2.13. Nesse tipo de mapa, os pontos brancos representam o $\mathcal{C}_{free}$ e os pontos pretos o $\mathcal{C}_{obs}$.

Os mapas de custo para o planejamento de rotas no espaço $\mathbb{R}^2$ são, geralmente, representados por meio de imagens que são correlacionadas com uma visão de topo do ambiente do robô. Esse mapa deve ser feito de forma que o robô consiga relacionar rapidamente um ponto dele com o correspondente estado real que ele representa. Isso é alcançado de maneira simples, por meio de uma representação proporcional do mundo nessa imagem.

Apesar de imagens serem discretas, é possível representar um mundo contínuo com o auxílio de uma alta resolução. Isso implica um custo espacial maior para o problema, porém uma diminuição de custo temporal, pois o sistema pode simplesmente olhar a imagem e determinar se aquele estado pertence a $\mathcal{C}_{obs}$ ou $\mathcal{C}_{free}$. Esse tipo de comportamento é desejado, pois o sistema embarcado em que se está trabalhando possui uma restrição de capacidade de computação mais significativa do que a restrição de armazenamento de dados, cerca de 32 gigabytes.

2.5.2.1 Planejamento contínuo baseado em amostras

Conhecendo agora o espaço $\mathcal{C}$ e os mapas de custo contínuo, fica muito mais fácil apresentar o planejamento baseado em amostras. A ideia principal desse tipo de planejador é evitar construir

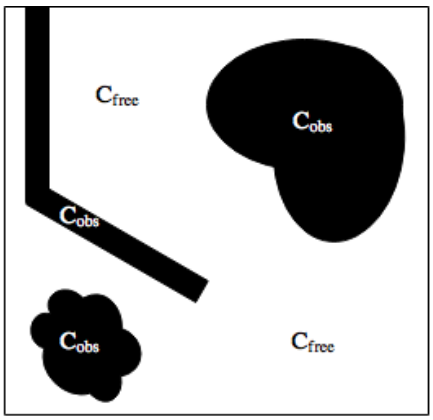


Figura 2.13: Espaço $\mathcal{C}$, representado como um mapa de custo contínuo.

explicitamente o $\mathcal{C}_{obs}$, e explorar o espaço $\mathcal{C}$ por meio de amostras. Essas são consideradas válidas, ou não, de acordo com um módulo de detecção de colisões [21]. Nesse trabalho, esse módulo será uma simples verificação do mapa de custo. Por fim, é realizada, uma busca discreta nessas amostras obtidas.

Esse modelo de planejamento é bom, pois dissocia, por meio da existência do módulo de detecção de colisão, os algoritmos da representação geométrica do problema, permitindo que a mesma solução seja aplicada em diversos cenários bem diferentes. No entanto, esse método apresenta um grande, porém solucionável problema.

O planejador baseado em amostras não é completo, pelo menos não pela definição formal desse conceito. Ou seja, dado um tempo finito, ele pode não retornar uma resposta em tempo finito. Isso se deve ao fato do espaço ser contínuo. Portanto, é possível ficar amostrando infinitamente e nunca obter as amostras necessárias para a solução.

Para isso não ser, de fato, um problema, duas coisas são muito importantes. A noção de densidade, que significa que as amostras se aproximam arbitrariamente de qualquer configuração quando o número de iterações tende ao infinito. E a estratégia de amostragem que deve seguir esse critério de densidade. Muitas delas, são baseadas em uma amostragem randômica, que é densa com uma probabilidade igual a um [21]. Algoritmos que utilizam essas estratégias são considerados probabilisticamente completos, o que significa que dado um número suficiente de pontos, a probabilidade que a solução seja encontrada converge para um.

Com esses conceitos fora do caminho, pode-se focar em uma das abordagens do planejamento baseado em amostras. O método de amostragem incremental e busca, mais especificamente as árvores densas de exploração rápidas, do inglês *Rapidly Exploring Dense Trees* - RDT. A ideia por trás desse algoritmo é construir, de maneira incremental, uma árvore de busca que gradualmente aumenta de resolução até preencher complementa o espaço $\mathcal{C}$. Essa árvore é construída com o auxílio de uma sequência. Caso ela seja aleatória, a árvore criada é conhecida pelo termo árvore aleatória de exploração rápida, *rapidly exploring random tree* - RRT.

A versão básica de construção dessas estruturas de busca é apresentada no Algoritmo 2. A árvore é inicializada com a primeira configuração desejada, q_0. Então, aumenta-se essa árvore quantas k vezes sejam desejadas. Em cada passo de crescimento da RDT, é escolhido um novo elemento da sequência densa e infinita α, que representa as amostras de $\mathcal{C}$. Caso essa sequência seja aleatória, a RDT encontra-se no caso particular das RRTs. Com esse ponto escolhido, procura-

Algoritmo 2: Algoritmo de construção de RDTs. Retirado de [21].

Entrada: q_0, o ponto inicial da árvore

```
1 G.Iniciar(q_0);
2 para i = 1 até k faça
3     G.adicionar_vertice(α(i));
4     q_n ← mais_proximo(S(G), α(i));
5     G.adicionar_aresta(q_n, α(i));
```

se qual elemento do grafo topológico $\mathcal{G}(V, E)$, um grafo que representa a RDT, constituído por vértices V e arestas E, pertence ao conjunto, $\mathcal{S} \in \mathcal{C}_{free}$, de todos os pontos alcançados por $\mathcal{G}$ e está mais próximo do atual ponto da sequência, $\alpha(i)$. Quando esse ponto, q_n, é determinado, inclui-se em $\mathcal{G}$ uma nova aresta entre q_n e $\alpha(i)$.

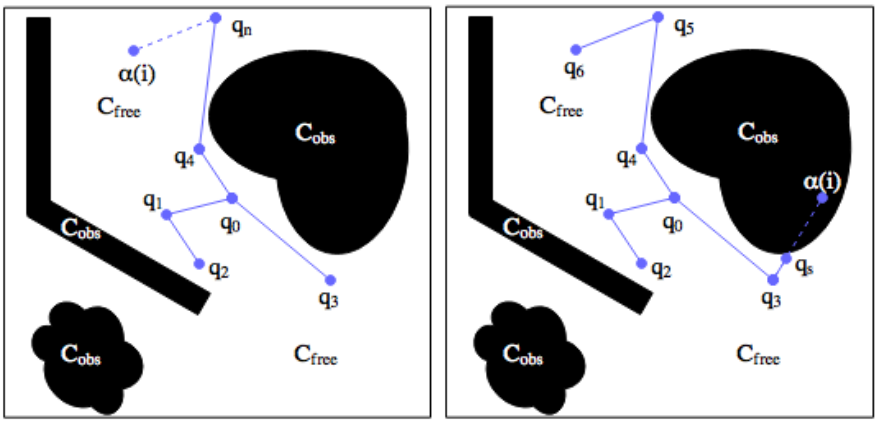
Esse algoritmo está representado na Figura 2.14a. Nela é possível observar uma etapa da construção de uma RDT, em que já existem vários pontos e arestas (em azul) que pertencem a $\mathcal{G}$, e agora deseja-se incluir a amostra $\alpha(i)$, cujo ponto mais próximo é q_n. A nova aresta sendo formada é representada por uma linha azul tracejada entre os dois.

É importante chamar a atenção para vários aspectos desse algoritmo apresentado. Primeiramente, ele necessita de alguma métrica. Na verdade, quase todos os algoritmos de planejamento baseado em amostras necessitam de alguma forma de medir a distância, ρ, entre duas amostras para que se possa decidir quando duas amostras estão próximas ou não. Essa métrica é bem simples no espaço $\mathbb{R}^2$ e é conhecida como distância euclidiana. Ela pode ser calculada com a fórmula da Equação 2.14. LaValle [21] apresenta muitas outras métricas para vários espaços $\mathcal{C}$ importantes para o planejamento de movimentos.

$$\rho(q_1, q_2) = \sqrt{({}_xq_1 - {}_xq_2)^2 + ({}_yq_1 - {}_yq_2)^2}. \tag{2.14}$$

Outro ponto importante é que esse algoritmo não está lidando com obstáculos. Isso é resolvido de uma maneira simples, e é um pequeno incremento ao Algoritmo 2. Ao invés de simplesmente pegar o ponto da amostra, $\alpha(i)$, e criar um vértice entre ele e o ponto mais próximo da árvore, q_n, verifica-se qual é a configuração q_s, no segmento que liga q_n a $\alpha(i)$, mais próxima do segundo, que não esteja em $\mathcal{C}_{obs}$ e nem seja q_n.

Por fim, destaca-se que esse algoritmo ainda não está buscando ainda pelo objetivo. Ele serve apenas para construir a RDT, ou nesse caso, a RRT. No entanto, é importante entender como o espaço de busca cresce por meio da expansão da árvore de exploração, para se ter uma compreensão do algoritmo de busca em RRTs que será apresentado.



(a) Etapa sem obstáculo (b) Etapa com obstáculo

Figura 2.14: Representação de etapas de construção de uma RDT.

Algoritmo 3: Algoritmo RRT de busca bidirecional. Retirado de [21].

Entrada: q_I, o ponto inicial da busca e q_G, o objetivo

$\mathcal{T}_a$.Iniciar(q_I); $\mathcal{T}_b$.Iniciar(q_G);
para $i = 1$ *até* k **faça**
    $q_n \leftarrow$ mais_proximo($\mathcal{S}_a, \alpha(i)$);
    $q_s \leftarrow$ configuracao_permitida($q_n, \alpha(i)$);
    se $q_s \neq q_n$ **então**
        $\mathcal{T}_a$.adicionar_vertice(q_s);
        $\mathcal{T}_a$.adicionar_aresta(q_n, q_s);
        $q'_n \leftarrow$ mais_proximo($\mathcal{S}_b, q_s$);
        $q'_s \leftarrow$ configuracao_permitida(q'_n, q'_s);
        se $q'_s \neq q'_n$ **então**
            $\mathcal{T}_b$.adicionar_vertice(q'_s);
            $\mathcal{T}_b$.adicionar_aresta(q'_n, q'_s);
        se $q'_s = q_s$ **então retorna** $SOLUÇÃO$;
    se $|\mathcal{T}_b| > |\mathcal{T}_a|$ **então** troca($\mathcal{T}_a$,$\mathcal{T}_b$);
retorna $FALHA$

O algoritmo de busca RRT consiste em, simplesmente, ir expandindo a árvore a partir do ponto inicial q_I seguindo a sequência α, verificando periodicamente se é possível conectar a árvore ao objetivo q_G. Isso é feito por meio de uma taxa, aqui denominada viés de objetivo, que varia de 0 a 100%.

O viés de objetivo indica a probabilidade do algoritmo buscar o objetivo ou um outro ponto aleatório. Esse número deve ser normalmente baixo, pois buscar sempre o objetivo leva a uma estratégia gulosa que pode ficar presa em certos ponto do espaço $\mathcal{C}$, e nunca encontrar a solução. Ele também não deve ser muito baixo, pois assim o algoritmo não possui nenhum incentivo para correr atrás do objetivo. Testes devem ser feitos para encontrar um valor que melhor contrabalanceie esses dois aspectos.

Para melhorar a performance do algoritmo, pode-se implementar duas árvores, uma expandindo a partir do ponto inicial q_I, e o outra a partir do objetivo q_G. O Algoritmo 3 apresenta uma versão básica desse algoritmo. O grafo topológico $\mathcal{G}(V, E)$ é dividido em duas partes $\mathcal{T}_a$ e $\mathcal{T}_b$, que inicialmente começam de q_I e q_G, respectivamente. $\mathcal{T}_a$ expande sempre seguindo a sequência α, enquanto $\mathcal{T}_b$ tenta sempre expandir na direção do ultimo vértice de $\mathcal{T}_a$, q_s. Se uma das duas árvores ficar maior que a outra em tamanho, elas trocam de função, fazendo com que sempre a menor árvore tente procurar de forma aleatória. Quando ambas as árvores se conectarem, o algoritmo para, pois a solução foi encontrada.

A Figura 2.15 mostra um exemplo desse algoritmo sendo aplicado. Nela é possível ver um dos maiores problema desse algoritmo. Ele não encontra um caminho ótimo. Ele apenas encontra um possível caminho. Além disso, como ele possui um caráter aleatório, o caminho encontrado será muito provavelmente diferente cada vez que ele rodar.

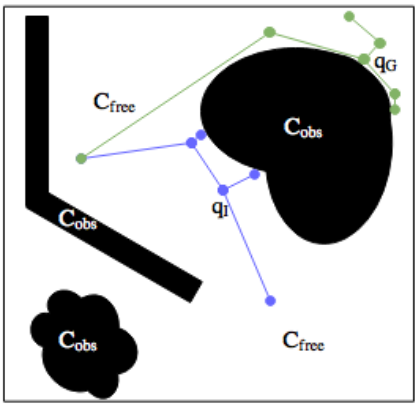


Figura 2.15: Espaço $\mathcal{C}$, representado como um mapa de custo contínuo.

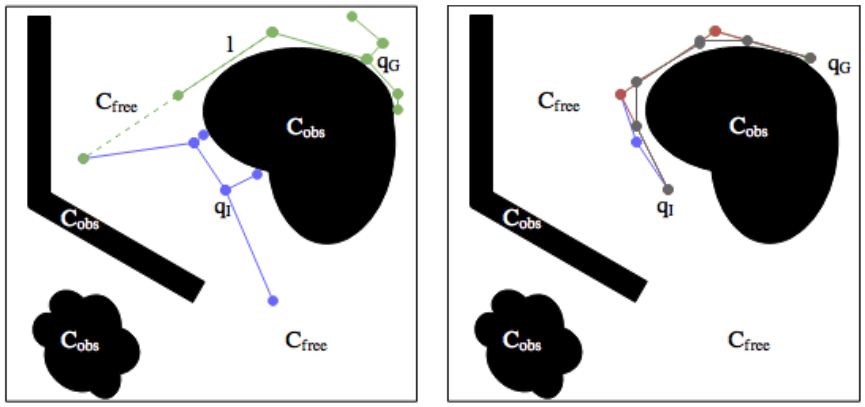
Esse algoritmo apresentado é a ideia básica por trás da busca em RRTs. Entretanto, existem várias mudanças e características diferentes que podem ser implementadas nessa busca, incrementando ela com várias ideias apresentadas no planejamento discreto. Em [25], são apresentadas algumas formas diferentes de implementar, e mudar um pouco o comportamento desses algoritmos, como por exemplo o RRT* e o RRG.

O algoritmo RRT implementado aqui possui algumas pequenas diferenças em relação ao apresentado. As árvores não ficam alternando de papel entre si, onde uma busca um ponto aleatório e a outra busca o último ponto adicionado na primeira. No caso aplicado nesse trabalho, ambas as árvores possuem o conceito de viés de objetivo e alternam entre os dois comportamentos.

Outra diferença é que existe uma limitação do tamanho do passo que cada árvore pode crescer de cada vez. Isso significa que dado um ponto da sequência de amostra $\alpha(i)$ e o vértice mais próximo dele q_n, a nova aresta criada não liga estas duas configurações, mas sim a uma configuração intermediária q_s que está sobre esse segmento de reta, porém com o comprimento máximo de passo estipulado. Isso pode acabar aumentando um pouco o custo computacional associado, mas diminui o comportamento mostrado na Figura 2.15, em que os vértices se conectaram de um jeito que produziu um desvio desnecessário no caminho. Esse comportamento está representado na Figura 2.16a.

A última diferença é que no final do algoritmo implementado nesse trabalho ocorre uma suavização da rota planejada, representada na Figura 2.16b. Isso é feito em duas etapas. A primeira, chamada de suavização de caminho, elimina pontos desnecessários do caminho original (roxo), isso retira laços, desvios e zigue-zagues, deixando o caminho o mais reto possível (vermelho). A segunda, é chamada de suavização de bordas, e tenta eliminar as quinas formadas na etapa anterior. Ela faz isso subdividindo as arestas em n partes, e tentando criar novas conexões que diminuam o caminho total. Gerando uma curva mais próxima de um resultado ótimo (cinza).

Essa última etapa aumenta o custo computacional do algoritmo RRT, porém não de uma forma que prejudique muito o seu desempenho. Ela traz um benefício de deixar a rota bem mais próxima da rota ótima, o que diminui muito o trajeto e a quantidade de ações que o robô deve fazer. Compensando a longo prazo o seu custo.



(a) Passo limitado ao comprimento l. (b) Suavização da rota.

Figura 2.16: Algoritmo RRT utilizado, com restrição de passo e suavização de caminho.

Com isso, foi apresentado todo o conteúdo básico necessário para compreender o módulo de planejamento de rotas apresentado posteriormente nesse trabalho. Deseja-se, também, enfatizar que o objetivo dessa seção era apenas dar uma pequena noção para o leitor sobre planejamento de rotas contínuo, especialmente aqueles baseados em amostras. Explicar de uma maneira simples, como um espaço contínuo pode ser amostrado de forma que possa-se trabalhar com ele utilizando os conceitos apresentados para os planejadores discretos, Subseção 2.5.1, sem perdas de informação.

Capítulo 3

Desenvolvimento

3.1 Introdução

Este capítulo visa apresentar o desenvolvimento da estrutura geral do arcabouço de navegação gerado nesse trabalho. Explicar seu funcionamento, como deve ser trabalhado e como implementá-lo em diversas situações. O capítulo é dividido em duas partes. Na primeira, são apresentados os três níveis que compõem o arcabouço. Quais são as características deles, suas funções e como eles interagem entre si.

A segunda parte mostra a implementação do arcabouço de navegação no robô Bruce, desenvolvido para participar da competição *Robomagellan*. Ela foca nos detalhes fundamentais para descrever a estratégia criada para ele no arcabouço. Além disso, mostra quais plataformas auxiliares foram necessárias, como foi o desenvolvimento do módulo planejador de rota e da camada de conversão.

3.2 Estrutura geral do arcabouço de navegação

Como dito anteriormente na Seção 2.4, deseja-se desenvolver um arcabouço de navegação que possua estas características: ser programável, autônomo, adaptável e extensível. Isso significa que a estrutura deva ser facilmente modificável, ou seja, deve ser simples mudar o comportamento do sistema em tempo de execução. Suas partes devem ser facilmente reaproveitáveis e reconfiguráveis, permitindo que o robô consiga ser reprogramado rapidamente e adquirir novos objetivos. E por fim, deve ser simples incorporar novos comportamentos e objetivos.

Para alcançar todos estes objetivos foi pensado em um arcabouço, representado na Figura 3.1, dividido em três diferente níveis: o Organizacional, o Funcional e o Executivo. Essa estrutura se assemelha um pouco com a apresentada em [14], porém, a nomenclatura, funcionamento e interação entre os vários níveis é diferente. Cada uma dessas camadas é responsável por uma função distinta, e todas elas trabalham em conjunto por meio de trocas de mensagem para navegar o robô corretamente.

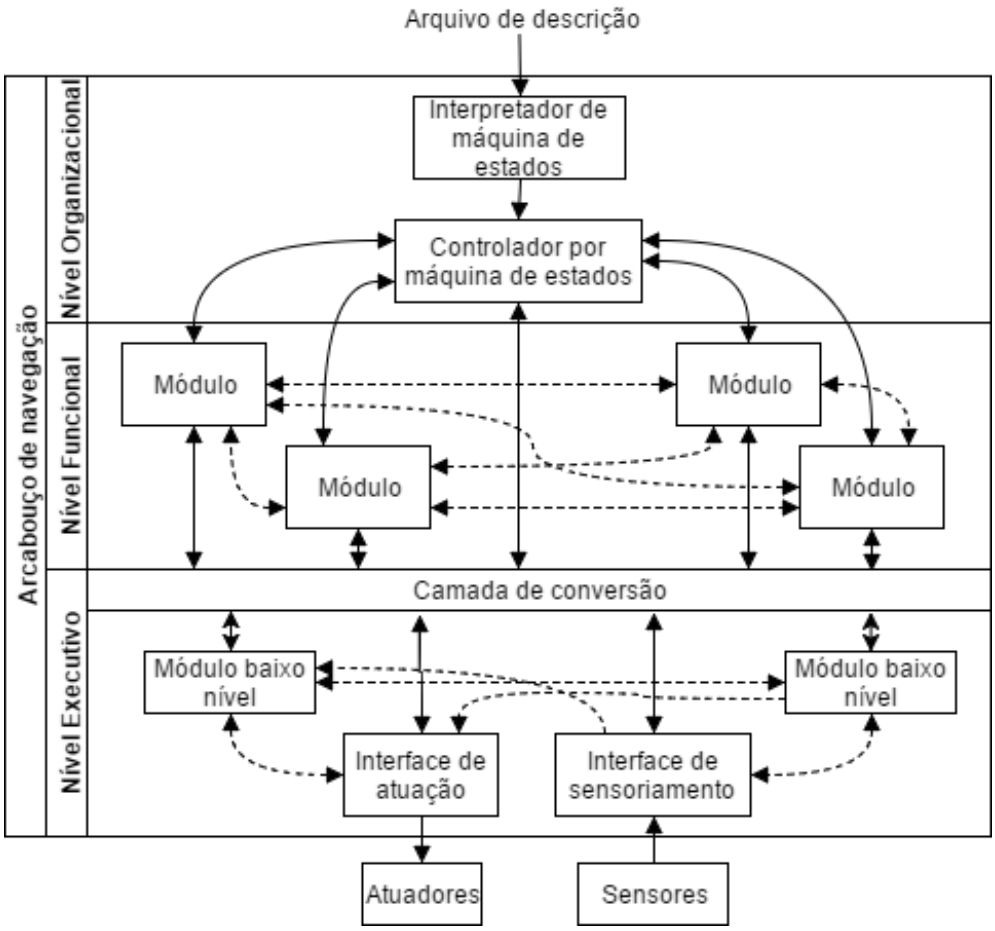


Figura 3.1: Estrutura geral do arcabouço de navegação.

Será analisado, agora, cada nível dessa arquitetura, para compreender o seu funcionamento e entender como ela cumpre os objetivos pretendidos. Começando a análise de cima para baixo, é importante entender que os níveis também representam o grau de abstração de cada camada. Quanto mais elevado, mais abstratos são os conceitos trabalhados. Portanto, o nível organizacional é o mais abstrato entre eles. Ele coordena todo o processo com o auxílio de uma máquina de estados, que é capaz de mudar o comportamento do robô de acordo a atual situação do ambiente.

O nível funcional é responsável por cuidar das ações e comportamentos do sistema. Ele é dividido em módulos que devem seguir uma série de regras que serão explicadas posteriormente. É nesse nível que se encontram os planejadores de rota, mapeamento, controladores, processamento de visão computacional, entre outros.

Por fim, o nível executivo é responsável por lidar diretamente com a parte física do processo. É importante ressaltar que o executivo nesse trabalho se referencia àquele que de fato executa, e não àquele que planeja a execução, como em muitos trabalhos no campo da robótica móvel. Esse é o nível que lida com a leitura dos sensores e controle dos atuadores, transmitindo para as camadas acima essa informação de um jeito mais compreensível para as mesmas. Ele também é responsável por comportamentos mais críticos, que necessitem ter um controle maior da parte física. No entanto, é importante frisar que esses módulos de baixo nível também podem se comunicar com

os módulos de alto nível e com o controlador.

Dividido dessa maneira, o arcabouço é capaz de ser facilmente reprogramado. Basta trocar a máquina de estados associada ao nível organizacional e todo o comportamento muda, sem necessidade de alterar mais nada. Os módulos do nível funcional permitem que seja criada uma biblioteca de módulos, facilitando o reaproveitamento de certos comportamentos em diversas situações. Essas bibliotecas podem ser facilmente expandidas com a incorporação de novos módulos, aumentando as capacidades do sistema. Além disso, a abstração fornecida pela camada de conversão do nível executivo permite dissociar esses comandos de alto nível da camada física, facilitando com que uma mesma estrutura funcione em diferentes robôs. Tendo, dessa maneira, um arcabouço com todas as características desejadas.

No entanto, para que tudo isso funcione da maneira desejada, é fundamental que se compreenda muito bem o papel de cada camada e como elas devem interagir entre si. Portanto, agora será apresentado o funcionamento detalhado de cada nível, e um conjunto de regras que deve ser seguido para o desenvolvimento de cada um deles.

3.2.1 Nível Organizacional: O controlador por máquina de estados

Começando pelo nível organizacional, como o próprio nome diz, ele é responsável por organizar o funcionamento do arcabouço. É essa camada que toma as decisões de o que o robô deve fazer e quando ele deve fazer isso. Para tanto, ela deve ser capaz de compreender o estado atual do ambiente e ter controle total sobre todas as ações do robô.

Esse controle é feito por meio de uma máquina de estados, como a representada na Figura 3.2. Nela, o comportamento de um robô é descrito por meio de estados que representam objetivos do sistema em um alto nível de abstração. Cada estado é, então, responsável por, decidir quais módulos ele deve habilitar ou desabilitar para alcançar esse objetivo, e quais comandos ele deve passar para os mesmos, de forma que ele consiga traduzir esse objetivo em alto nível em uma série de objetivos mais simples que cada módulo necessário possa cumprir.

Os estados devem, também, ser capazes de perceber quando seu objetivo foi alcançado. Essa informação pode vir tanto da camada de conversão, que mantém informações sobre os estados

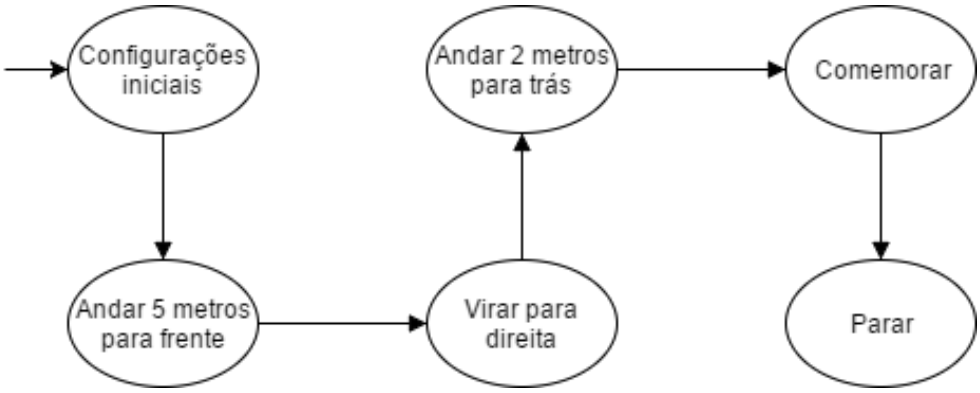


Figura 3.2: Uma máquina de estados genérica, que representa um comportamento de um robô.

do ambiente, quanto dos próprios módulos, que podem avisar que terminaram sua função, ou que produziram uma informação que o controlador necessite. Essa decisão pode vir inclusive do próprio estado, que percebeu que se passou tempo de mais, ou tentativas demais, e seu objetivo ainda não foi atingido.

Com o intuito de facilitar a implementação dessa máquina de estados, foi desenvolvida uma forma de descrevê-la por meio de um arquivo de descrição específico. Esse arquivo é lido na inicialização do sistema por um interpretador que consegue, em tempo de execução, montar a máquina de estados que será utilizada.

Dessa forma, para mudar o comportamento do sistema, basta dizer qual arquivo se deseja utilizar. Isso permite que vários comportamentos diferentes sejam preparados e rapidamente testados, sem a necessidade de recompilar todo o código. Esse comportamento é relevante, pois a compilação em ambientes embarcados pode ser bem demorada, e como uma das utilizações almejadas para esse arcabouço é a aplicação em competições de robótica, economizar tempo para mudar um comportamento e fazer testes pode ser a diferença entre ganhar e perder.

Esse arquivo de descrição é composto por quatro seções: ações, condições, estados e máquina de estados. As três primeiras são seções de declaração. Nelas são definidas, respectivamente, quais as ações, as condições e quais estados estarão presentes na máquina de estados. A última seção é a que realmente dita o comportamento dela, por meio das ações e condições declaradas. No entanto, falta agora entender o que é uma ação e o que é uma condição:

- **Ações**

  As ações são trechos mínimos de código pré-compilados que executam uma tarefa. Essas tarefas podem ser: trocar de estado, habilitar ou desabilitar um módulo, mudar variáveis para certos valores, definir as velocidades dos motores, criar, reiniciar e inicializar contadores e temporizadores, entre outras coisas. Basicamente qualquer coisa que se deseja que aconteça no sistema deve ser modelada como uma ação.

- **Condições**

  As condições também são trechos de código pré-compilados. A diferença é que, em vez de executar uma tarefa, elas verificam se algo aconteceu. Os trechos pré-compilados, que representam as condições, recebem sempre uma pergunta cuja resposta é verdadeiro ou falso e, caso ela seja verdadeira, uma ou mais ações são executadas. Elas podem ser: comparações entre variáveis, leituras de sensores, resultados de um módulo, verificação de um contador ou temporizador, entre outras coisas.

Ambas, ações e condições, devem ser simples e diretas. Elas não devem ter mais que uma função ao mesmo tempo. Seus objetivos e usos devem ser claros e não depender de outras partes. Dessa maneira, é possível criar bibliotecas de ações e condições que possam ser utilizadas em uma diversidade maior de contextos.

O interpretador deve ter acesso a essas bibliotecas para identificar quais são as ações e condições existentes no sistema. Dessa forma, ele pode buscar por suas definições e dependências durante as

suas respectivas declarações. A Figura 3.3 mostra um exemplo das três primeiras seções em que ocorrem essas declarações. A última seção, que descreve o comportamento da máquina de estados, é exibida na Figura 3.4. Essa máquina de estados, apresentada nesse arquivo de descrição dividido entre ambas as figuras, implementa a máquina representada na Figura 3.2.

As primeiras duas seções, *Actions* e *Conditions*, são onde ocorrem as declarações das ações e das condições, respectivamente. Cada seção é separada por meio de um "@", e nelas, as declarações são separadas por um sinal de "+". Em cada declaração dessas duas seções existem dois campos separados pelo símbolo ":". O primeiro campo é como aquela ação ou condição será conhecida nesse arquivo de configuração, e o segundo, o nome real da ação ou condição que o interpretador deve buscar em suas bibliotecas. Isso é feito para agilizar o processo de interpretação, restringindo a quantidade de vezes que ele deve buscar algum elemento em todas as bibliotecas existentes.

A seção *States* simplesmente declara quais estados irão existir, de forma a facilitar referências cruzadas. Suas descrições são apresentadas na seção FSM, que significa *Finite State Machine*, o termo em inglês para máquina de estados finita. A Figura 3.4 apresenta essa seção. Nela, os estados são separados por "+", enquanto as ações e condições são tabuladas e separadas por linhas. A ação resultante de uma condição se encontra na mesma linha dela, separada pelo símbolo ">>". Caso se deseje adicionar mais de uma condição necessária para ativar uma ação, utiliza-se o símbolo "&&". O mesmo vale para uma ou mais condições ativando mais de uma ação.

Uma outra característica importante das ações e das condições é que elas tem a capacidade de receber parâmetros que podem alterar o seu funcionamento. Esses parâmetros devem ser nomeados e são declarados junto com as definições das ações ou condições em suas bibliotecas. Eles podem ser colocados diretamente no arquivo de configuração, conforme o exemplo da Figura 3.4, ou serem indicados por meio de alguma variável de estado que a máquina possua.

O controlador por máquina de estados possui mais uma função que ainda não foi mencionada.

```
@Actions
+ mudar_estado : change_state
+ definir_odometria : set_odometry
+ definir_velocidade: set_velocity
+ dancar: dance

@Conditions
+ odometria_x_maior: odometry_x_bigger
+ odometria_angular_maior: odometry_theta_bigger
+ danca_completa: dance_complete

@States
+ configuracoes_iniciais
+ andar_5_metros_para_frente
+ virar_para_direita
+ comemorar
+ desligar
```

Figura 3.3: Exemplo das seções: Ações, Condições e Estados de um arquivo de configuração da máquina de estados.

```
@FSM
+ configuracoes_iniciais
        definir_odometria(x: 0, y: 0)
        mudar_estado(next: andar_5_metros_para_frente)
+ andar_5_metros_para_frente
        definir_velocidade(x: 1, y: 0, angular: 0)
        odometria_x_maior(value: 5) >> definir_velocidade(x: 0, y: 0, angular: 0) && mudar_estado(next:
virar_para_direita)
+ virar_para_direita
        definir_velocidade(x: 0, y: 0, angular: -1)
        odometria_angular_maior(value: 90) >> definir_velocidade(x: 0, y: 0, angular: 0) && mudar_estado(next:
comemorar)
+ comemorar
        dancar()
        danca_completa() >> mudar_estado(next: desligar)
+ desligar
        definir_velocidade(x: 0, y: 0, angular: 0)
```

Figura 3.4: Exemplo da seção que descreve o funcionamento da máquina de estados de um arquivo de configuração da máquina de estados.

Ele deve manter uma representação do ambiente, de forma que todos os estados possuam acesso a ela. Leituras de sensores, variáveis de controle, mapas, tudo isso deve ser acessível de todos os estados, e por consequência, deve existir uma maneira de referenciá-los a partir do arquivo de configuração. Na implementação apresentada essa referência é feita por nome, e é configurado no próprio interpretador quais são as variáveis que existem no sistema e como elas devem ser chamadas.

Outro detalhe específico dessa implementação é que uma vez que a máquina entre em um estado, ela fica rodando repetidamente as ações e condições ali especificadas até que ocorra uma mudança de estado. Por isso, achou-se necessário a criação de ações e condições relacionadas com temporizadores e contadores, de forma a garantir que algo aconteça somente um determinado número de vezes, ou após um determinado tempo, caso seja necessário.

As ações e condições são a base do funcionamento da máquina de estados. Elas servem várias funções diferentes no arcabouço de navegação. Elas são responsáveis por se comunicar com as outras camadas, por administrar o controle e por permitir que a máquina de estados seja representada de uma maneira mais abstrata. Elas possibilitam um crescimento contínuo da versatilidade dessa arquitetura, permitindo que rapidamente se combinem as ferramentas disponíveis e se crie um novo comportamento.

No entanto, é necessário lembrar que as ações e condições apenas coordenam as coisas: As coisas acontecem de fato nos próximos dois níveis do arcabouço.

3.2.2 Nível Funcional: Os módulos

No nível funcional são implementadas, realmente, as ações e comportamentos que o nível organizacional controla. Para alcançar esse objetivo, essa camada utiliza módulos que os representam. Quando a máquina de estados requisita uma ação, é enviada uma mensagem que solicita ao módulo associado àquela ação fazer o que for necessário.

Entretanto, em muitas ocasiões é necessário que o módulo já esteja rodando para poder fornecer

a resposta desejada pelo sistema. Um exemplo disso é a requisição de um mapa a um módulo de mapeamento. Se ele não estiver mapeando enquanto outras coisas acontecem, o mapa gerado não estará muito atualizado, e por consequência, não será muito útil. Assim sendo, concluiu-se que módulos devem rodar de maneira paralela e com um certo grau de independência do resto do sistema.

Diz-se um certo grau de independência, pois o sistema ainda precisa controlar o módulo. Isso significa que os módulos devem disponibilizar para o controlador alguma interface que permita o seu controle. Seja esse método uma troca de mensagens, ou alguma outra técnica, deve ser garantido que o módulo seja capaz de ser habilitado ou desabilitado quando necessário. Esses são os dois comandos mínimos que código deve ter para ser considerado um módulo.

Outras capacidades que um módulo pode ter são: ser interrompido e reiniciar de onde parou, aceitar parâmetros de entrada, enviar resultado a quem requisitar, se comunicar com outros módulos pedindo o que seja necessário deles, interagir com o nível executivo para obter valores de sensores ou atuar no ambiente, entre outras coisas necessárias para atingir o seu objetivo.

Assim, é possível entender o que um módulo pode fazer e como ele interage com o resto do arcabouço. Agora, basta entender o que eles realmente são. Módulos são trechos de códigos pré-compilados que buscam cumprir um objetivo, assim como as ações e condições do nível organizacional. No entanto, esses objetivos podem ser mais complexos do que os presentes nos outros dois. Na verdade, a maioria das ações e condições são apenas o meio que o controlador utiliza para se comunicar com o módulo que realmente implementa aquele comportamento.

Da mesma forma que as ações e condições devem ser simples e focadas, o objetivo do módulo deve ser único, bem definido e focado. Módulos que possuem muitos objetivos intermediários provavelmente poderiam ser reescritos como mais módulos com objetivos mais simples que se comunicam para alcançar um objetivo maior. Essa maior distribuição do sistema permite que também sejam criadas bibliotecas de módulos. Permitindo que módulos que originalmente ajudavam a cumprir um determinado objetivo, facilmente sejam integrados a outro propósito.

Um exemplo que ajuda a ilustrar a diferença entre módulos, ações e condições é mostrado na Figura 3.5. As ações estão em vermelho e as condições em azul. Os módulos são as caixas brancas de mapeamento, planejamento de rotas e controlador de trajetória. Os três possuem objetivos

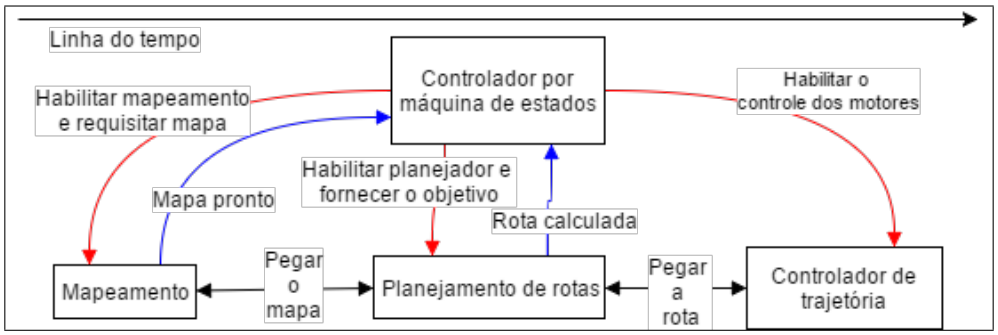


Figura 3.5: Exemplo de um controlador interagindo com os módulos por meio de ações (vermelho) e condições (azul).

específicos e executam o que for necessário para alcançá-los, inclusive interagir com os outros módulos para pegar informações que eles precisam. Os eventos apresentados na imagem ocorrem segundo uma linha do tempo, da esquerda para a direita, na qual o controlador vai emitindo ações e esperando condições se tornarem realidade para prosseguir, enquanto isso, os módulos abaixo continuam trabalhando paralelamente de acordo com os comandos recebidos.

Juntando essas informações é possível perceber que a biblioteca de ações e condições está intrinsecamente conectada à biblioteca de módulos. O módulo possui uma série de ações e condições associados a ele para que o controlador possa interagir com o mesmo. Portanto, o padrão correto de criação, desenvolvido nesse trabalho, foi: criar o módulo necessário para atingir a meta desejada, e desenvolver as ações e condições associadas àquele método. Fazendo assim um crescimento orgânico das duas bibliotecas.

A implementação teste desenvolvida neste trabalho utilizou as ferramentas disponíveis no ROS, Subseção 2.2.1, para elaborar os módulos e a comunicação entre os mesmos. Os módulos são representados por nós do ROS, e a comunicação utiliza o próprio sistema de mensagens já presente na plataforma para realizar a interação. Dessa forma, o ROS cuidou de toda a parte de gerenciamento de recursos e alocação de tarefas necessárias para que o sistema funcionasse e pudesse rodar de maneira paralela.

O nível funcional apenas faz toda a computação necessária para realizar o seu propósito, mas ainda falta a real atuação no ambiente físico. Ele ainda deve requisitar ao nível executivo que de fato mova o motor, ou ande a uma determinada velocidade.

3.2.3 Nível Executivo: O baixo nível

O nível executivo é aquele que tira tudo do mundo da computação e da abstração e transforma isso em atuações no ambiente em que o robô se encontra. Ele também é responsável por fazer o caminho inverso, pegar as informações desse mundo e trazer para um nível de abstração em que os módulos e o controlador consigam trabalhar.

Isso é feito por meio destes três principais componentes: a interface de sensoriamento, a interface de atuação e a camada de conversão. Os dois primeiros são os responsáveis por lidar diretamente com os sensores e atuadores, respectivamente. São eles que devem fazer todos os procedimentos necessários para que se consiga ler um ultrassom, um GPS, uma unidade de medida inercial, ou qualquer outro tipo de sensor. Ou então mover um motor, ativar uma bomba e acionar outros atuadores.

Ambas interfaces devem trocar informações com a camada de conversão, que tem como função abstrair esses dados para as camadas superiores. Avaliando, primeiramente, essa abstração pelo lado dos sensores, os dados obtidos pela interface de sensoriamento devem ser envelopados de uma maneira que se saiba que aquela informação veio de determinado tipo de sensor, mas não especificamente de um sensor de um modelo determinado. Por exemplo, a informação da velocidade das rodas, ou até mesmo a velocidade do robô, deve ser envelopada de um jeito que os níveis superiores saibam esses valores e o que eles significam, mas não precisem se importar com detalhes

de como ele foi obtido.

O mesmo deve acontecer com a interface de atuação. A camada de conversão deve ser capaz de receber pedidos abstratos como, por exemplo, andar a 5 m/s em direção ao eixo x do robô. E então ser capaz de transformar isso nos comandos corretos associados ao motores utilizados, independente da cinemática e dinâmica reais associadas àquele robô.

Esses envelopamentos de informação devem seguir uma convenção estabelecida previamente, de tal forma que os módulos e controlador consigam trabalhar com eles. A camada de conversão deve ser capaz de converter as informações contidas nesses envelopes em comandos de baixo nível, assim como fazer o caminho inverso. Dessa maneira, é possível construir algoritmos e comportamentos que consigam funcionar em uma variedade de estruturas robóticas diferentes.

É nesse ponto que os sistemas operacionais para robótica, explicados na Seção 2.2, apresentam grande potencial de ajuda. Alguns deles, como o ROS, já possuem diversas formas padrões de envelopar esses tipos de informação, permitindo, assim, integrar facilmente muitas ferramentas, já desenvolvidas, ao sistema, e assim expandir de maneira significativa as capacidades do mesmo.

Apesar de ser responsável por esse envelopamento, a camada de conversão deve disponibilizar também acesso direto às informações de baixo nível. Às vezes, é necessário ter um controle mais preciso do comportamento da estrutura e como ela realmente atua e percebe o mundo. Isso, no entanto, é extremamente contra os padrões do arcabouço aqui proposto, porque faz com que as técnicas utilizadas fiquem muito amarradas ao sistema no qual elas estão inseridas, dificultando, dessa forma, o reaproveitamento das mesmas em outros robôs.

Para suprir essa necessidade, foi incluída nas funcionalidades do nível executivo a existência dos módulos de baixo nível. Esses módulos têm as mesmas características que os do nível funcional. Eles ainda devem poder ser habilitados e desabilitados, receber parâmetros, retornar valores, entre outras coisas. Entretanto, é importante ressaltar que a interface entre o controlador de máquina de estados e eles é a camada de conversão, que possui agora mais essa responsabilidade.

A maior diferença entre os módulos do nível funcional e executivo são o seus propósitos. Os últimos têm como objetivos implementar comportamentos que necessitam uma maior proximidade com a interface física e implementar comportamentos emergenciais, que necessitem atuar de maneira muito rápida, sem gastar tempo na camada de envelopamento.

É nesses módulos que, geralmente, se encontram os controladores que asseguram o correto funcionamento de atuadores. Como por exemplo, o controlador de velocidade de um motor, que se assegura que dada uma velocidade rotacional de referência ω_{ref}, a velocidade rotacional real ω se aproxima dela com um mínimo erro.

Outro tipo de módulo que normalmente se encontra nesta camada são os que possuem um comportamento reativo. Esses módulos geralmente requerem um processamento rápido de entrada e saída, e portanto, são classificados como comportamentos emergenciais, que precisam estar próximos dos sensores e atuadores.

Com isso, obtém-se um entendimento geral do funcionamento e implementação desse arcabouço de navegação, e como ele alcança os objetivo que ele se propôs. Ele não é chamado de arquitetura

de maneira direta, pois é um pouco mais flexível que isso, e permite que o tipo de arquitetura varie de acordo com os módulos e o controlador presentes.

Para fazer um sistema com arquitetura pura reativa, basta não utilizar a camada funcional e desenvolver módulo somente na camada executiva que tenham esse perfil. O mesmo vale para a arquitetura deliberativa, no entanto, os módulos ficariam apenas na camada funcional. A arquitetura híbrida é alcançada unindo as duas alternativas anteriores, e utilizando o controlador para mediar a interação entre as duas. Até a arquitetura baseada em comportamento pode ser implementada. O único requerimento para isso é que os módulos possuam os perfis de comportamentos definidos por essa arquitetura.

3.3 Uma implementação para competição

Esse arcabouço foi implementado para estruturar o sistema do robô Bruce, apresentado na Subseção 1.2.1. Ele foi desenvolvido de forma paralela a este trabalho, e seu objetivo é cumprir o desafio proposto pela competição *RoboMagellan*, apresentada na Seção 1.1. De forma resumida, o robô deve alcançar certos marcos, cujas coordenadas GPS são conhecidas. Esses pontos são visualmente indicados por um cone laranja, e a tarefa é considerada concluída quando o robô o toca.

3.3.1 Estrutura proposta de solução

A estratégia pensada para cumprir essa tarefa pode ser dividida em duas partes. A primeira é uma aproximação em fase de cruzeiro, o robô utiliza a sua localização em relação ao mundo para tentar se aproximar da posição global do alvo. Essa localização, entretanto, não apresenta precisão suficiente para garantir que ele vá tocar no alvo. Por esse motivo, quando ele se aproximar o suficiente, ele muda para a segunda etapa, que é uma aproximação fina. Nela, ele utiliza a câmera para estimar a posição do alvo em relação a si para poder ter uma maior garantia que irá tocar no mesmo.

Essa estratégia representa de maneira resumida o funcionamento apresentado na Figura 1.4. Esses comportamentos podem ser facilmente modelados de acordo com o arcabouço de navegação desenvolvido na Seção 3.2. Um diagrama que demonstra esse funcionamento estruturado de acordo com o arcabouço pode ser visualizado na Figura 3.6. As cores azul, verde e roxo representam, respectivamente, em qual microcontrolador, Raspberry Pi, Arduino leitor de sensores, ou Arduino atuador aquela funcionalidade está implementada. As que possuem mais de uma cor estão implementadas de maneira distribuída.

A aproximação de cruzeiro exige que, primeiramente, o robô consiga se localizar em relação ao mundo. Com essa informação em mãos, ele precisa traçar um caminho que ele consiga seguir sem colidir com nenhum obstáculo previamente definido. E por fim, sabendo por onde ele deve passar, o robô precisa decidir como ele vai passar por esses lugares, ou seja, quais velocidades ele precisa imprimir em seus atuadores para que ele alcance os pontos desejados. Esses formam os três

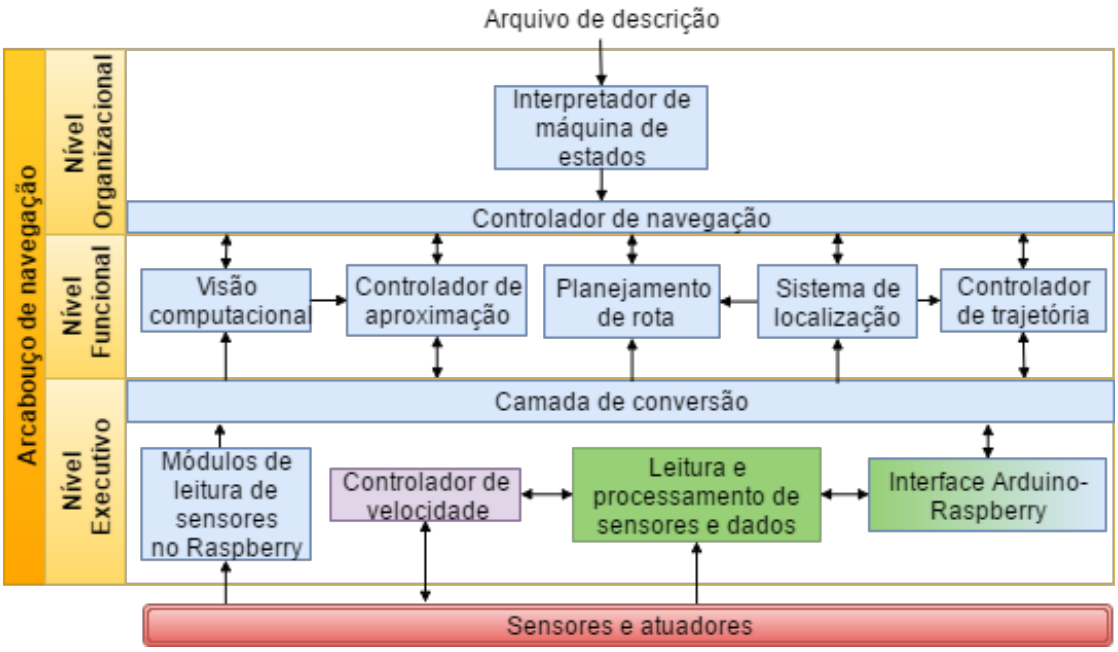


Figura 3.6: Diagrama que representa o funcionamento do Bruce estruturado no arcabouço de navegação.

primeiros módulo do nível funcional, respectivamente, o sistema de localização, o planejamento de rota e o controlador de trajetória. Dentre eles, o único que será desenvolvido neste trabalho será o segundo.

A aproximação fina depende apenas da câmera e dos atuadores. Ela necessita que seja estimada uma posição e ângulo do alvo relativos ao sistema de coordenadas local do robô a partir da imagem do cone, que o indica, obtida pela câmera. Com essas duas informações, o robô deve calcular as velocidades necessárias para que ele se aproxime do alvo. Esses comportamentos podem ser modelados nos outros dois módulos presentes no nível funcional, respectivamente, a visão computacional e o controlador de aproximação. Ambos estes módulos foram desenvolvidos em outros trabalhos, e serão somente utilizados aqui.

O ROS, Subseção 2.2.1, foi utilizado para implementar e gerenciar a comunicação entre os módulos. Cada um deles é um diferente nó, que utiliza o paradigma propagador-assinante apresentado para se comunicar. Isso significa que toda a informação relevante produzida por um módulo é publicada em um tópico, e um outro que a deseje pode simplesmente ir lá e pegar. Dessa maneira, a única dependência que os módulos têm entre si é o tipo de informação que eles trocam, deixando suas implementações específicas independentes, atendendo as exigências de reutilização e modularização do arcabouço.

O controlador por máquina de estados, aqui conhecido como controlador de navegação, também é um nó do ROS. Os módulos oferecem para ele uma interface de controle por meio de um tópico, o qual ele pode assinar e publicar os comandos que ele deseje. O restante do nível organizacional é implementado exatamente como explicado anteriormente na Subseção 3.2.1. A máquina de estados que contém o comportamento desejado, ilustrada na Figura 3.7, é descrita no arquivo de

configuração por meio de ações e condições, que são interpretadas em tempo de execução durante o início do ciclo de execução.

As Subseções 3.3.2, 3.3.3 e 3.3.4 a seguir entrarão em detalhes sobre o funcionamento do controlador de navegação, do planejador de rotas e do nível executivo, respectivamente. Dessa maneira, ficará mais elucidado como o arcabouço de navegação foi implementado, e como a interação entre seus diferentes níveis estruturais funciona.

3.3.2 Controlador de navegação

O controlador de navegação implementa a máquina de estados representada na Figura 3.7 com auxílio das 17 ações e 12 condições apresentadas nas Tabelas 3.1 e 3.2. Cada uma delas é associada a algum método, ou então estão ligadas ao controle da máquina de estados, como, por exemplo, a *change_ state*, que muda qual o estado atual. As condições podem, também, estar associadas à camada de conversão, como a *us_ reading_ smaller*, que verifica se o valor de um sensor ultrassônico é menor do que um outro valor. As definições de qual sensor, e qual valor comparar, são passadas como parâmetros para a condição.

Existem sete ações e três condições associadas ao controle da máquina de estados. Dentre as sete, uma já foi explicada, *change_ state*, e serve para mudar o estado atual. Um delas, a *print*, serve apenas para ajudar em testes e escreve coisas na tela. As restantes estão dividas em dois grupos que gerenciam os contadores e temporizadores. Cuidando da criação deles, *add_ counter* e *add_ timer*, da sua reinicialização, *reset_ counter* e *reset_ timer*, e no caso do contador, para aumentar o seu valor, *increment_ counter*. As três condições associadas ao controle também estão ligadas a estes dois grupos. Sendo uma para verificar se um temporizador já havia ultrapassado um determinado valor passado como argumento, *check_ timer*. E as outras duas para verificar se o contador era maior ou menor do que o parâmetro passado, *check_ counter_ bigger* e *check_ counter_ smaller*.

As dez ações restantes estão divididas como, cinco que interagem com o planejador de rota e cinco que comandam os outros módulos que não foram implementados neste trabalho. A

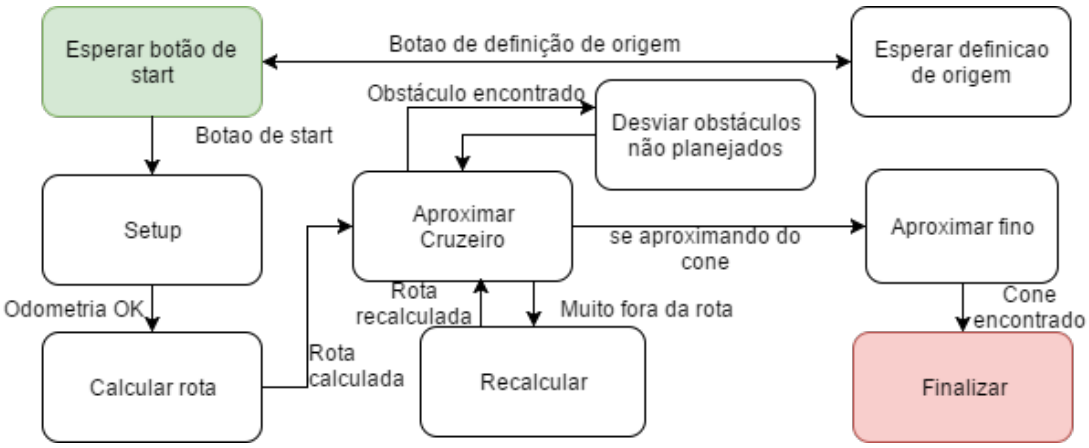


Figura 3.7: Máquina de estados que representa a estratégia implementada. Os estados verde e vermelho são, respectivamente o inicial e o final.

Tabela 3.1: Ações utilizadas na máquina de estados e os módulos associados a elas.

Ação	**Módulo associado**
change_state	Controle
set_map_path_goal_gps	Planejador
publish_map_bottom_left_corner	Planejador
add_timer	Controle
print	Controle
reset_timer	Controle
request_route_calculation	Planejador
insert_us_obstacles_in_map	Planejador
send_enable_path_follow	Controlador de trajetória
insert_camera_readings_in_map	Visão Computacional
set_calculating_route	Planejador
add_counter	Controle
increment_counter	Controle
set_velocity	Controlador de velocidade
reset_counter	Controle
send_enable_follow_camera	Controlador de aproximação
request_set_origin	Localização

send_enable_path_follow habilita o controlador de trajetória a seguir o caminho calculado pelo planejador de rota. A *request_set_origin* solicita ao sistema de localização que defina o ponto no qual o robô se encontra como a origem do sistema de coordenadas global dele. A *insert_camera_readings_in_map* coloca o cone encontrado pela câmera no mapa que representa o ambiente, para que seja verificado posteriormente se aquele cone está próximo de um dos pontos de interesse, ou se ele é apenas um cone qualquer. A *set_velocity* requisita ao controlador de velocidade que fique a alguma determinada velocidade linear e angular. Por fim, a *send_enable_follow_camera*, habilita o controlador de aproximação a usar a câmera como referência para se aproximar do alvo. As ações do planejador serão explicadas mais a frente, quando ele o for.

Já as nove condições restantes estão mais bem distribuídas entre os módulos e a camada de conversão. Sendo esta última a que mais as tem. Ela conta com quatro condições, cada uma responsável por retornar o estado atual de algum sensor. Elas são a *us_reading_smaller*, a *check_touch_sensor*, a *check_green_button* e a *check_black_button*, que, respectivamente, verifica se a leitura de distância do sensor ultrassônico é menor que um valor, se o sensor de toque acoplado ao para-choque está apertado, ou se os botões verde e preto do painel de controle estão pressionados. O sistema de localização tem três, a *odom_ok*, que diz se a posição e a orientação do robô estão calculadas com uma precisão aceitável, a *distant_from_route*, que utiliza essa posição estimada para analisar se ele está longe demais do caminho calculado, e se ele deveria recalcular a rota, e a *check_origin_received*, que diz se já foi definida a origem do sistema de coordenadas. As outras duas estão igualmente divididas para os seguintes módulos: o planejador de rotas, com a

Tabela 3.2: Condições utilizadas na máquina de estados e os módulos associados a elas.

Condição	**Módulo associado**
check_ timer	Controle
check_ touch_ sensor	Camada de conversão
odom_ ok	Localização
check_ counter_ bigger	Controle
route_ calculated	Planejador
check_ counter_ smaller	Controle
distant_ from_ route	Localização
check_ green_ button	Camada de conversão
cone_ found	Visão
check_ black_ buton	Camada de conversão
us_ reading_ smaller	Camada de conversão
check_ origin_ received	Localização

route_ calculated, que informa se ele já terminou de calcular a rota; e a visão computacional, com a *cone_ found*, que diz se a câmera está vendo algo que pode ser considerado como um cone.

Com uma melhor compreensão das ações e condições existentes, pode-se analisar cada estado implementado descrito em alto nível e ver como eles utilizam esses recursos para controlar o robô .

a) **Esperar botão de *start***

O robô espera um dos dois botões serem apertados. Caso o botão preto seja pressionado, ele muda de estado para definir o ponto atual como a origem do sistema de coordenadas. Caso seja o verde, ele começa a rotina e vai para a configuração inicial (*Setup*).

Ações utilizadas: *change_ state*.

Condições utilizadas: *check_ green_ button* e *check_ black_ button*.

b) **Esperar definição de origem**

Solicita-se que o sistema de localização defina o ponto no qual o robô se encontra como a origem do sistema de coordenadas. Quando isso estiver concluído, ele volta para o estado de esperar o botão de início.

Ações utilizadas: *change_ state* e *request_ set_ origin*.

Condições utilizadas: *check_ origin_ received*.

c) ***Setup***

Nesse estado, espera-se posição do robô convergir, assim que isso acontece, ele começa a calcular a rota. Também são realizadas algumas configurações iniciais para o planejador de rota, que recebe a coordenada GPS do canto inferior esquerdo do mapa.

Ações utilizadas: *change_ state* e *publish_ map_ bottom_ left_ corner*.

Condições utilizadas: *odom_ ok*.

d) **Calcular rota**

O robô analisa onde ele está, e tenta traçar uma rota até o seu destino. Quando a rota é calculada ele vai segui-la.

Ações utilizadas: *change_state*, *set_map_path_goal_gps*, *request_route_calculation* e *set_calculating_route*.

Condições utilizadas: *route_calculated*.

e) **Aproximar cruzeiro**

Neste estado, robô está se aproximando do objetivo seguindo uma rota. Ele fica monitorando a sua distância a essa rota e recalcula caso ele se distancie mais do que uma distância pré determinada da mesma. Além disso, ele verifica os sensores ultrassônicos, para procurar por obstáculos desconhecidos, tomando as ações necessárias para desviar deles quando necessário. Ele faz isso até ver o cone, quando ele muda para a etapa de aproximação fina.

Ações utilizadas: *change_state* e *send_enable_path_follow*.

Condições utilizadas: *distant_from_route*, *cone_found* e *us_reading_smaller*.

f) **Recalcular**

O robô recalcula o trajeto até o objetivo a partir do ponto atual, sem desativar os outros módulos já ativos. Ou seja, ele continua fazendo o que ele estivesse fazendo enquanto recalcula.

Ações utilizadas: *change_state*, *request_route_calculation* e *set_calculating_route*.

Condições utilizadas: *route_calculated*.

g) **Desviar obstáculos não planejados**

Pega as medidas dos sensores ultrassônicos de distância e transforma-as em obstáculos nas coordenadas do mapa do robô. Com essas coordenadas temporariamente adicionadas, ele recalcula a rota de forma a desviar dos obstáculos. Simulataneamente a isso, o controle de trajetória desvia dos obstáculos de maneira reativa, enquanto ele espera a nova rota.

Ações utilizadas: *change_state*, *request_route_calculation*, *set_calculating_route* e *insert_us_obstacles_in_map*.

Condições utilizadas: *route_calculated*.

h) **Aproximar fino**

Utiliza a câmera para determinar a posição do cone em relação ao robô e segue ele com o controlador de aproximação. Quando ele encostar no cone ele vai para o estado final.

Ações utilizadas: *change_state* e *send_enable_follow_camera*.

Condições utilizadas: *check_touch_sensor*.

i) **Finalizar**

Para o robô, pois a tarefa foi completada.

Ações utilizadas: *set_ velocity.*

Condições utilizadas: Nenhuma.

Dessa forma, é possível ver que o controlador foi projetado de acordo com o padrão de projeto estipulado pelo arcabouço. As ações e condições representam, com um alto nível de abstração, tarefas com objetivos focados e simples. Elas controlam módulos que implementam esses comportamentos de maneira paralela e desconexa, não dependendo do contexto para funcionar. Ambos podem ser reutilizados em diversas situações diferentes para gerar outros comportamentos, criando a estrutura reprogramável desejada.

3.3.3 Planejador de rotas

Como visto na Seção 2.5.2, o planejador necessita de uma representação do ambiente, de forma que ele possa construir o espaço $\mathcal{C}$ necessário para o planejamento. A competição acontece em um espaço externo de aproximadamente 300 por 300 metros, contendo prédios, árvores e outras estruturas grandes como obstáculos. Porém, sem a necessidade de lidar com desníveis significativos. Dessa forma, o ambiente pode ser simplificado como uma visualização de topo bidimensional.

Sendo assim, o ambiente pode ser representado de uma forma muita tradicional para os seres humanos: um mapa. Para que esse mapa faça sentido para o robô, ele precisa de no mínimo duas informações em cada ponto. A primeira é uma coordenada que indique rapidamente qual lugar no ambiente esse ponto é, que convenientemente é um informação facilmente obtida com o GPS. A segunda é uma informação de custo. Ela é um valor que varia de 0 a 100, dizendo o quão indesejável para o sistema é passar por aquele ponto, o 100 representando um lugar intransponível.

Para obter um mapa que tenha seus pontos em coordenadas conhecidas, utilizaram-se imagens georreferenciadas. O georreferenciamento é a ciência da relação exata de localizações com sistemas de referência espaciais, como o GPS [26]. Ele é feito em imagens, normalmente, geradas com satélites, que são processadas por equipes especializadas da área, e posteriormente distribuídas ao público por programas como o Google Earth ou o NASA World Wind.

Apesar desses programas possuírem imagens georreferenciadas, eles não apresentam uma forma nativa de exportá-las. Entre os dois, o Google Earth é o que possui a maneira mais simples de conseguir extrair essas informações. Ele permite salvar uma imagem, como a da Figura 3.8a, e obter o comprimento e altura da imagem em metros, assim como a coordenada GPS de cada um dos quatro cantos da imagem. Dessa forma, é possível determinar a dimensão de cada pixel e calcular a coordenada de um ponto intermediário da imagem relacionando essa distância entre ele e as bordas.

O planejador evita a necessidade de converter todos os pontos da imagem para as coordenadas GPS por meio de duas técnicas. Primeiramente, ele pega a origem do sistema de coordenadas do robô, e converte as quatro coordenadas dos cantos da imagem do sistema de localização global para o dele. Assim, a conversão entre um ponto intermediário da imagem para uma coordenada no sistema cartesiano do robô vira um simples problema de translação e rotação. A segunda coisa que

ele faz é trabalhar em seu próprio sistema de coordenadas. Ele recebe os pontos de interesse em relação as sua respectivas referências, converte-as para o seus sistema, calcula o plano, converte-o para sistema de coordenadas do robô e o devolve.

Um grande problema encontrado é que o robô não entende imagens como a da Figura 3.8a. Ele não consegue entender o que é ou não um obstáculo. Por esse motivo, é necessário transformar essa imagem em um mapa de custo, como o da Figura 3.8b. Esse mapa é um desenho que contém as mesmas informações da imagem georreferenciada, entretanto, em vez dos canais dela representarem as cores que formam a imagem, existe apenas um canal que indica o custo daquela posição. Esse custo varia de 0 a 255, associado a escala de 0 a 100 mencionada anteriormente, de forma que o valor mais baixo é o branco, variando na escala de cinza até o preto.

Essa representação é aceitável, pois o robô interage com ambientes externos, cujos obstáculos fixos são significantemente grandes, como árvores, cercas, prédios, entre outros. Sendo identificáveis nas imagens de satélites obtidas. A resolução desses mapas, cerca de $21,8$ centímetros por pixel nos mapas da Figura 3.8, é boa o suficiente para o objetivo pretendido, que é obter uma rota simples que evite esses objetos grandes.

Objetos menores e desconhecidos serão tratados com a combinação de dois comportamentos. O planejador de rota temporariamente adiciona o obstáculo detectado no mapa, e recalcula a rota. Enquanto isso acontece, o controlador de trajetória aplica métodos reativos de desvio de obstáculos, para que o robô não colida com eles enquanto recalcula o caminho.

Com o espaço $\mathcal{C}$ definido, e com o mapa de custo como ferramenta para construção do conjunto de pontos obstruídos $\mathcal{C}_{obs}$, o algoritmo RRT, apresentado na Subseção 2.5.2.1, foi implementado como um nó do ROS, que possui as ações e condições, para ser controlado pela máquina de estados, das Tabelas 3.1 e 3.2.

(a) Imagem georreferenciada obtida com o Google Earth[1].

(b) Mapa gerado com a identificação manual dos obstáculos.

Figura 3.8: Obtenção do mapa de custo do robô.

[1]https://www.google.com.br/intl/pt-PT/earth/

Ele recebe o objetivo como uma coordenada GPS por meio da ação *set_ map_ path_ goal_ gps*, e utiliza a posição do canto inferior esquerdo, que o controlador passou para ele (*publish_ map_ bottom _ left_ corner*), para convertê-la para o seu sistema de coordenadas. Ele é então solicitado a calcular a rota, *request_ route_ calculation*, a partir da posição atual fornecida pela sistema de localização. Em seguida, ele publica essa rota e avisa que terminou, com a condição *route_ calculated*. O planejador possui uma ação, *set_ calculating_ route*, auxiliar que evita que um pedido seja feito, enquanto outro está sendo computado.

Quando um obstáculo é identificado pelos sensores de distância ultrassônicos, eles são inseridos no mapa pelo controlador, *insert_ us_ obstacles_ in_ map*, que solicita um recálculo da rota. Ao término desse novo planejamento, esses obstáculos são removidos e espera-se um tempo curto, no qual não são aceitas novas solicitações de recálculo. Isso é feito para evitar que um processamento computacional extra seja gasto tentando desviar do mesmo obstáculo, deixando um tempo para que o controlador de trajetória possa seguir a nova rota.

3.3.4 Camada de conversão e o nível executivo

A camada de conversão pega as informações de baixo nível dos sensores e atuadores e as converte para um nível mais alto de abstração. Para atingir esse objetivo, foram utilizados os tópicos e mensagens disponíveis na plataforma ROS. Ela possui um grande número de mensagens pré-definidas pela comunidade. Essas mensagens apresentam funções abstratas, que atendem as demandas dessa camada, como, por exemplo, as mensagens de odometria, que indicam a posição, orientação e velocidade do robô em um determinado momento, ou as mensagens de sensores de distância, que representam medidas de sensores desse tipo, como os sensores ultrassônicos. Dessa maneira, pacotes desenvolvidos pela comunidade do ROS podem ser facilmente integrados no sistema como módulos do nível funcional.

Portanto, ela foi implementada como nós do ROS que leem os dados dos sensores e os envelopam dentro dessas mensagens, garantindo sempre que a informação mais recente esteja disponível, se não, avisando quando eles foram atualizados pela última vez. Assim como para os sensores, existem mensagens padrões para a atuação. A camada de conversão é responsável por pegar essas mensagens e converter nos comandos de baixo nível necessários.

Nesse caso específico, a parte do sensoriamento pode ser dividida em duas categorias: aquela que lê os sensores diretamente nas entradas GPIO do Raspberry, como por exemplo, o GPS, a IMU e a câmera; e aquela que se comunica com o Arduino por meio da interface serial implementada.

A primeira categoria é mais direta, e usada para ter um acesso mais constante às informações. Por isso, sensores vitais associados a localização do robô ficam aqui, de forma que o atraso entre a geração do dado e seu processamento seja mínimo. Sensores com fluxo muito grande de informação, como a câmera, também pertencem a essa categoria por dois motivos: os microcontroladores como o Arduino, não possuem uma interface adequada para tratar os seus dados; e a quantidade de dados ocuparia a maior parte da banda de comunicação serial entre eles. Esses nós acessam diretamente os dados dos sensores, fazendo os filtros e processamentos necessários e os publicam

nos seus respectivos tópicos.

A segunda categoria é mais lenta, e precisa da ajuda de módulos de baixo nível para conseguir cumprir suas funções. Por isso, sensores que não necessitam de uma taxa de atualização tão alta, cerca de algumas vezes por segundo apenas, ficam nessa categoria. Isso é feito, pois alguns microcontroladores como o Arduino são mais adequados para uma aplicação em tempo real do que um computador como o Raspberry. Tem-se um controle maior sobre o que realmente está sendo executado e quando essa execução ocorre, pois não existe um sistema operacional coordenando as tarefas. Eles possuem, também, uma grande biblioteca de suporte para a utilização de vários sensores e atuadores, facilitando a implementação do executivo.

Essa segunda categoria é dividida em duas partes devido às limitações de poder de processamento do Arduino. A primeira é a que lê os onze sensores de distância ultrassônicos, o sensor de toque, os botões e o receptor de rádio frequência. É essa parte que se comunica diretamente com o Raspberry, por meio do protocolo de comunicação serial. Ele envia os valores lidos nos sensores e recebe comandos de velocidade e de habilitação dos módulos de baixo nível.

Essa comunicação ocorre por um padrão de nove bytes por dado, o primeiro é somente um campo de identificação utilizado para que se saiba o que aquela informação é, os oito restantes são a informação em si. Por exemplo, o campo de identificação 0 indicaria que aquele dado representa a velocidade desejada na roda esquerda, e os 8 bytes seriam interpretados como o valor no qual se deseja que ela gire.

Ela também possui um módulo de baixo nível que recebe comandos do receptor RF. Esse receptor possui três canais que recebem sinais controlados por modulação de largura de pulso. Cada um desses canais representa o estado de um botão no controle remoto e está relacionado com uma determinada função. Um deles é um botão de dois estados, o comportamento dos outros dois botões é determinado pela sua posição. Isso significa que existem dois modos de funcionamento. O primeiro ignora completamente os comandos de velocidade provenientes da comunicação serial, deixando os outros dois canais funcionarem conforme um controle de carrinho remoto, facilitando o transporte do robô para os locais de teste. O segundo modo de funcionamento repassa as velocidades que a comunicação serial recebeu para o outro Arduino, somente se o botão de gatilho do controle estiver apertado. Esse comportamento é uma trava de segurança exigida pela competição.

A segunda parte dessa categoria é a parte de atuação que implementa o módulo de baixo nível de controle de velocidade. Ela teve de ser colocada em um microprocessador dedicado, devido à alta taxa de atualização dos sensores e a necessidade de se ter um controle maior sobre os intervalos de atuação. A necessidade de ler os onze sensores ultrassônicos atrasava um pouco o intervalo de amostragem do controlador, piorando o seu desempenho. Apesar da desvantagem de um maior atraso no recebimento do comando de velocidade, pois ele passa por duas comunicações seriais, o custo benefício de ter um intervalo de amostragem mais preciso foi satisfatório, principalmente devido ao fato de que esse atraso é quase insignificante para a atuação no motor.

Dessa forma, é possível perceber que o nível executivo faz aquilo que lhe é pedido. Os níveis superiores interagem com ele por meio de tópicos gerados pela camada de conversão, sendo a

implementação física real uma caixa preta. Eles somente fornecem entradas, comandos de atuação, e recebem saídas, valores de sensores. Assim, a mesma estrutura pode ser utilizada tanto para simulação, quanto para o robô real. A única diferença seria o nível executivo utilizado.

Capítulo 4

Resultados

4.1 Introdução

Esse capítulo apresenta o resultado de vários testes realizados com o arcabouço de navegação desenvolvido no Capítulo 3. O desejo inicial era que os testes fossem feitos durante a competição, com o robô realizando o desafio proposto. Infelizmente, devido a restrições financeiras, físicas e temporais, isso não foi possível.

Por esse motivo, todos os testes realizados envolvem tarefas necessárias para completar o desafio. Cada um deles visa averiguar alguma situação ou algum comportamento específico, sendo que dois deles foram realizados na plataforma robótica desenvolvida paralelamente a este trabalho, enquanto o outro foi realizado no ambiente de simulação Gazebo, apresentado na Subseção 2.3.1, justamente com o propósito de mostrar que o mesmo arcabouço pode ser facilmente adaptado para plataformas diferentes.

Esses testes foram pensados de maneira a mostrar que o arcabouço proposto seria capaz de implementar o comportamento desejado, descrito na Subseção 3.3.1. Mesmo que devido as restrições apresentadas, não tenha sido possível executar tudo que se planejava na plataforma física.

4.2 Teste de aproximação fina

O primeiro teste realizado assume que a etapa de aproximação cruzeiro conseguiu colocar o robô em uma posição na qual ele consegue ver o cone, executando somente a aproximação fina. Esse teste tem como objetivo mostrar a interação entre os módulos de visão computacional, o controlador de aproximação e o controlador de velocidade, mostrar a camada de conversão em ação no sistema real e mostrar o controlador de navegação percebendo mudanças no seu ambiente, por meio do sensor de toque do para-choque, para perceber que o robô alcançou o seu objetivo.

A Figura 4.1 ilustra três gráficos que representam o comportamento do robô durante o teste. O primeiro, de cima para baixo, é o gráfico da distância estimada pela câmera até o cone. O segundo é o gráfico do ângulo que o centro do cone faz com o centro do robô, ou seja, ele indica quão para

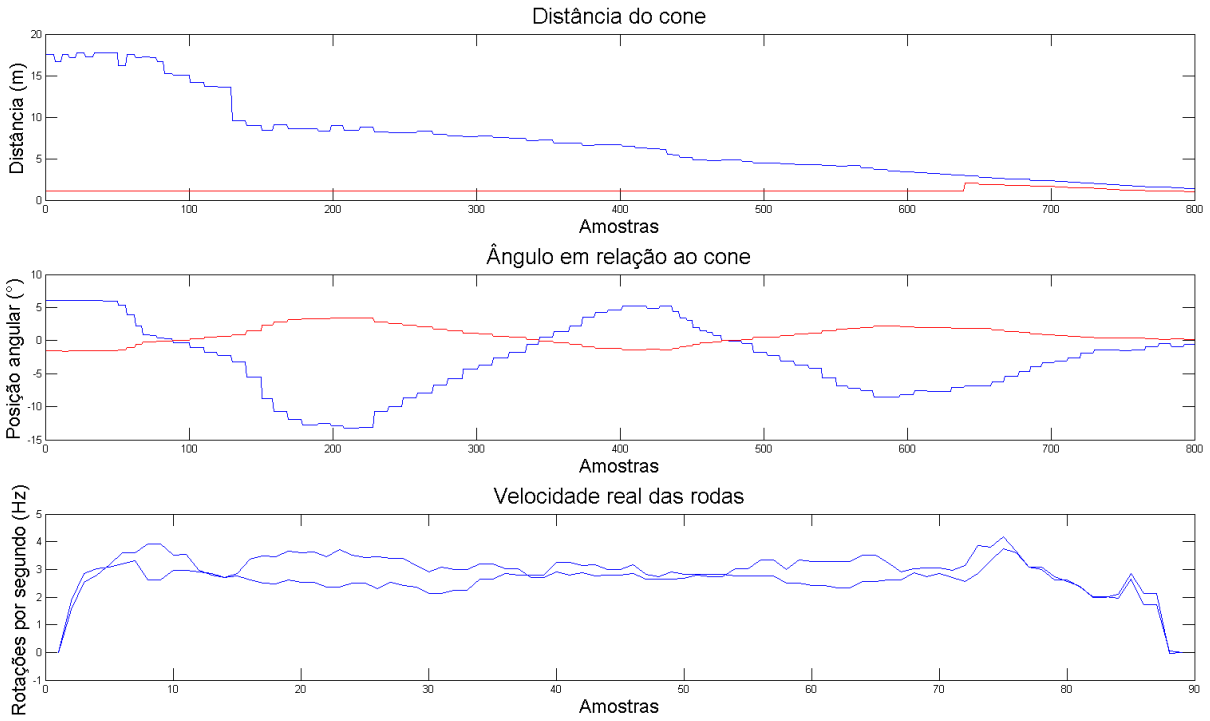


Figura 4.1: Gráficos representando a aproximação do cone.

direita ou para esquerda do centro do robô o alvo está. Por fim, o último indica as velocidades medidas nas rodas durante as etapas previamente apresentadas. As curvas vermelhas visíveis nos dois primeiros gráficos são, respectivamente, a velocidade linear e angular que o controlador de aproximação desejava em cada momento. É importante evidenciar que todos esses gráficos estão em função do tempo, mas não estão descritos em nenhuma unidade, com cada ponto nos gráficos representando apenas uma amostra.

O movimento feito pelo robô pode ser visualizado, também, por meio de coordenadas polares, como ilustrado na Figura 4.2. Nessa representação, é possível perceber o padrão de movimento que o comportamento produz, examinando-a temporalmente da direta para a esquerda. O robô se aproxima constantemente do alvo, sempre tentando deixar a imagem do cone centralizada por meio de movimento senoidais com amplitude decrescente.

A partir dos gráficos apresentados, é possível observar que o sistema teve exatamente o comportamento desejado, sempre diminuindo sua distância ao objetivo, e corrigindo sua orientação, de forma a chegar no alvo o mais alinhado possível. No final dos três primeiros gráficos é possível perceber uma discrepância entre os dados, apesar da câmera indicar que o cone ainda estava um pouco distante, algo por volta de um metro, e um pouco desalinhado, cerca de dois graus, a velocidade real medida foi para zero. Isso mostra justamente o arcabouço funcionando, pois o controlador por máquina de estados recebeu da camada de conversão o sinal que o sensor no para-choque havia sida acionado, e por isso, mudou o estado do sistema para finalizar a tarefa.

Portanto, mesmo que o módulo de visão computacional ainda mostrasse que o robô deveria

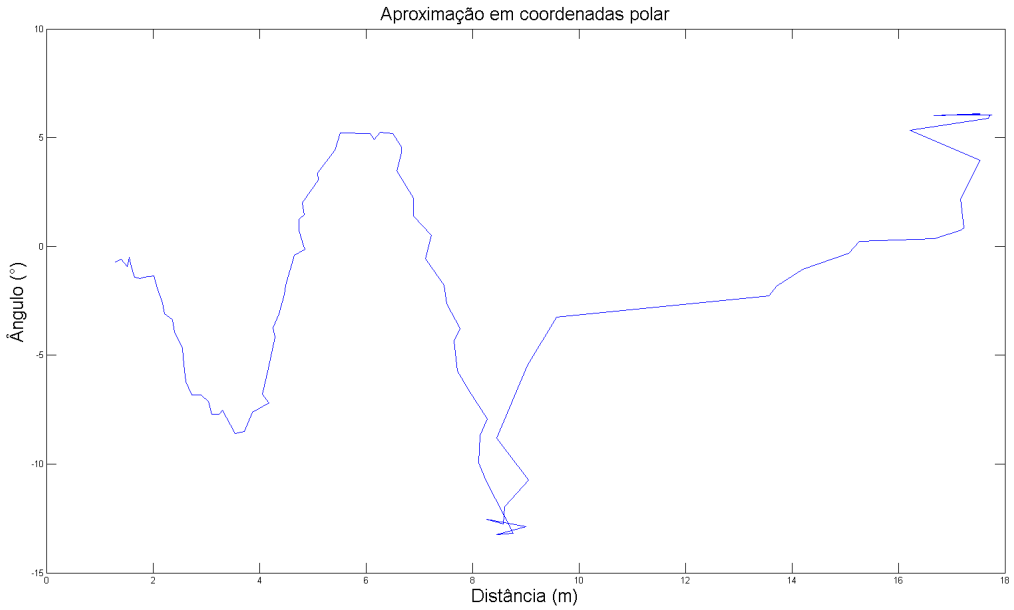


Figura 4.2: Aproximação do cone ocorrendo em coordenadas polares.

se mover, fazendo que o controlador de trajetória ainda quisesse mandar velocidades lineares e angulares para a camada de conversão, o controlador de navegação mudou o comportamento do sistema, definindo que esses módulos não estavam mais aptos a atribuir comandos de navegação ao robô, e que ele deveria ficar parado.

4.3 Teste com odometria

O segundo teste tem como objetivo testar o planejamento de rota e a etapa de aproximação de cruzeiro. Ele também foi realizado na plataforma física, porém ela estava suspensa, deixando apenas as rodas girarem, com a localização sendo calculada por uma odometria a partir da integração da velocidade das mesmas. Esse tipo de estimação acumula muitos erros e poderia levar a algum acidente que poderia danificar a estrutura do robô. Impedindo a união desse teste com o anterior. Isso teve que ser feito devido a problemas encontrados com os sensores associados à parte de navegação, que impediram maiores avanços nessa parte.

Para replicar condições parecidas com as da competição, foi escolhido realizar esse teste em "um ambiente externo". Como tudo foi feito utilizando apenas a integração da velocidade das todas, o robô não ficou realmente do lado de fora. O mapa foi construído baseado em uma quadra de Brasília, a partir de uma imagem de satélite georreferenciada.

O ponto inicial e o objetivo foram passados como coordenadas GPS para o módulo de planejamento, que gerou uma rota no sistema de coordenadas local do robô, mostrada na Figura 4.3. Essa figura mostra a visualização das percepções e representações do robô disponíveis com a ferramenta RVIZ. Nela, é possível ver o robô, ilustrado como um retângulo vermelho com um arco de leituras

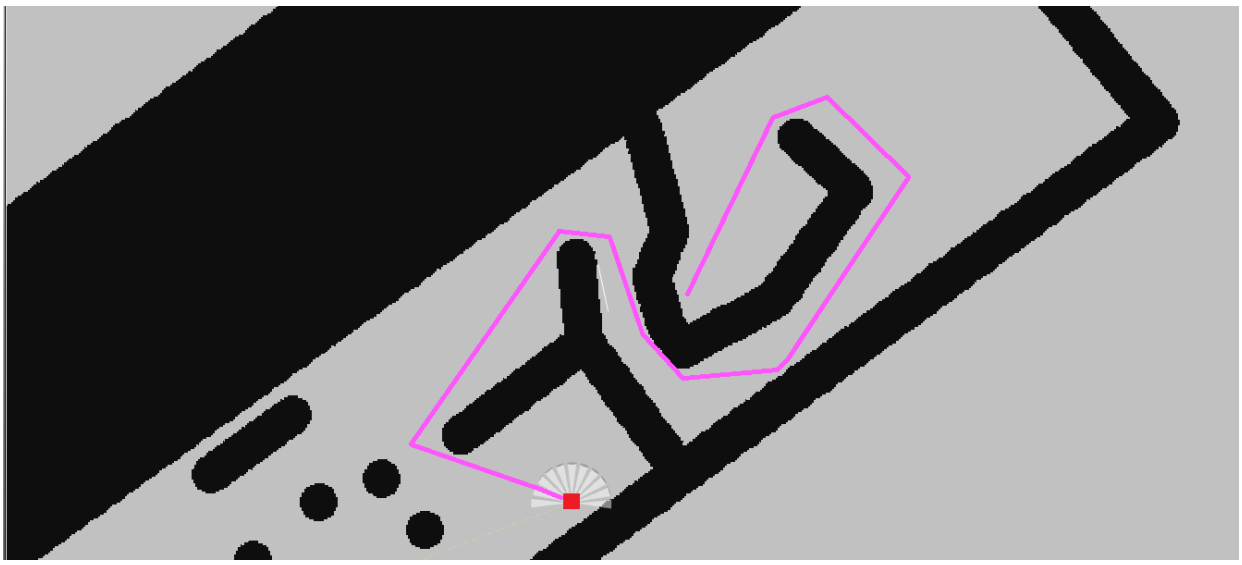


Figura 4.3: Rota inicial calculada visualizada no RVIZ.

de sensor ultrassônicos, no ponto onde o sistema de localização acha que ele está no mapa. As partes cinzas são os espaço livres e as pretas os obstáculos. As linhas roxas são a rota calculada pelo planejador.

O sistema foi programado para efetuar todos os passos associados à aproximação de cruzeiro, descritos na Subseção 3.3.1. Ele passou por todos os estados planejados, esperou o botão ser apertado, fez a configuração inicial, calculou a rota inicial e habilitou o controlador de trajetória para segui-la, sempre verificando se o robô não se afastava demais da mesma. O que acabou acontecendo, o controlador de trajetória acabou se perdendo e se distanciou mais do que a distância mínima da rota, o que foi identificado pelo controlador de navegação, que calculou uma nova rota e a enviou para o controlador de trajetória, que conseguiu seguir esse novo plano, representado na Figura 4.4.

Dessa maneira, o robô conseguiu chegar bem próximo do objetivo final, como pode-se ver na Figura 4.5, seguindo uma rota complexa calculada pelo planejador de rotas, sem colidir com obstáculos no caminho. Mostrando a robustez da estrutura de navegação proposta, pois mesmo com um erro do controlador de trajetória, ainda foi possível compensar isso, calculando um nova rota e chegando ao alvo. Infelizmente, como o robô estava na realidade parado, não foi possível testar a detecção reativa dos obstáculos com os sensores de distância ultrassônicos.

A Figura 4.6 mostra exatamente o caminho percorrido pelo robô durante o teste, traço azul. Ela ilustra muito bem o comportamento do sistema, as linhas verde e vermelha indicam a referência que o controlador de trajetória segue para fazer o robô acompanhar a rota calculada. Essa referência é dividida em diversos segmentos de reta que são trilhados um a um, sempre que um deles é superado, o controlador calcula um novo segmento a partir do ponto atual até o próximo ponto da rota, gerando, assim, a curva descontínua verde da imagem. A referência vermelha representa a mesma coisa, porém para a nova rota calculada. Deixando mais fácil identificar exatamente o momento no qual o controlador de trajetória se perdeu e o controlador de navegação calculou a

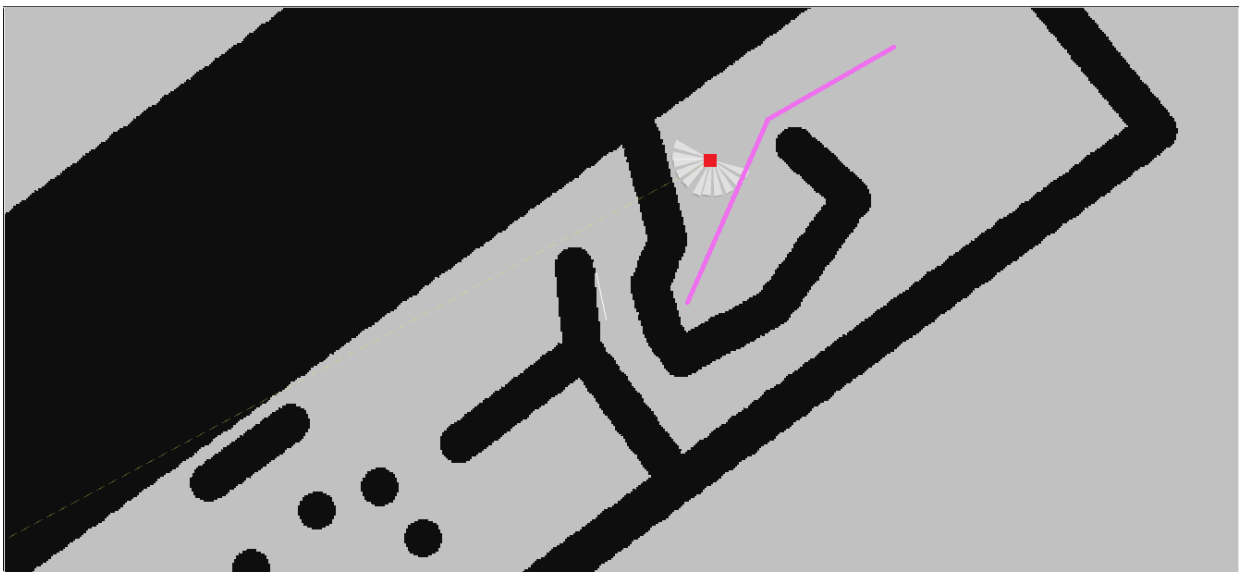


Figura 4.4: Nova rota, após recálculo.

nova rota.

Essas figuras evidenciam a necessidade da etapa de aproximação fina no sistema, pois apesar dele aparentar ter chegado exatamente aonde ele deveria pela Figura 4.6, é possível observar um pequeno erro na posição final na Figura 4.5, que seria facilmente corrigido pela etapa de aproximação com a câmera.

Esse teste também foi feito para mostrar a interação entre os três níveis do arcabouço de navegação nessa implementação que utiliza a plataforma ROS. A Figura 4.7 ilustra a comunicação entre os nós, retângulo com elipses dentro, e os tópicos, retângulos simples. O nível organizacional está todo contido no nó chamado de *bruce_ main_ 2*, como descrito anteriormente, ele utiliza tópicos

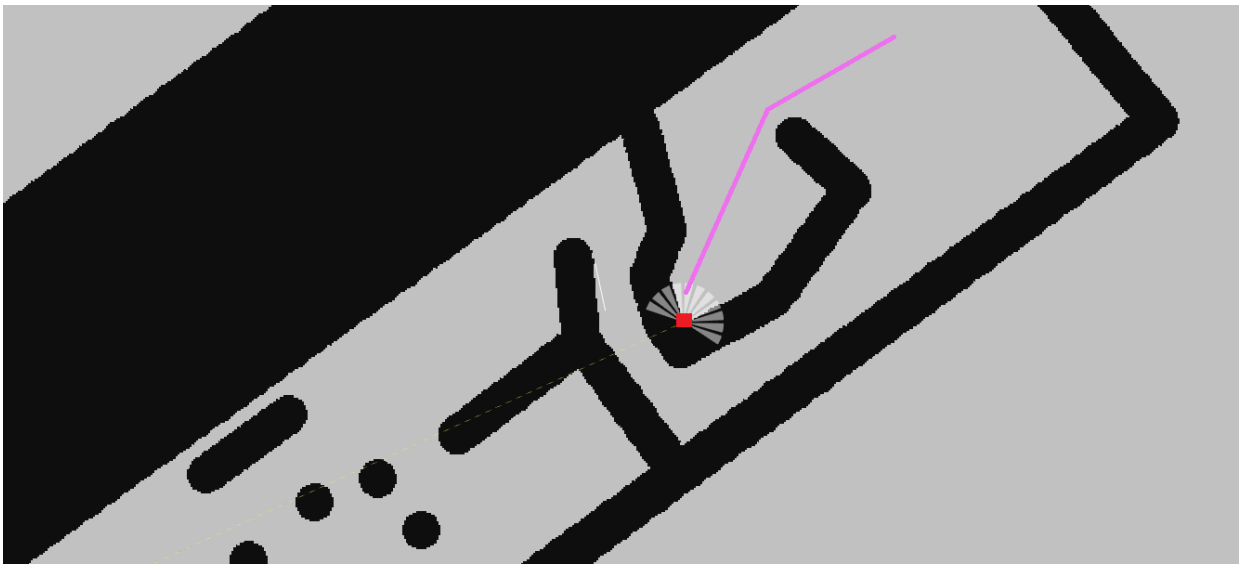


Figura 4.5: Robô no objetivo final, após a navegação.

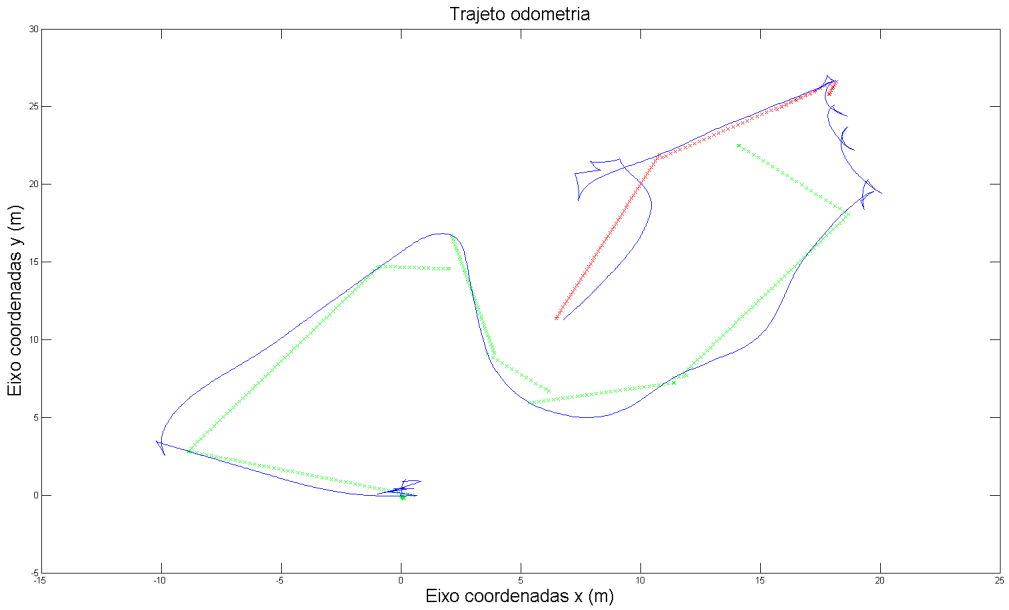


Figura 4.6: Comportamento do controlador de trajetória.

para comandar os módulos do nível funcional que estão divididos entre os cinco nós a seguir: *bruce_path_planner* é o planejador de rotas; *visao_usuario* é o módulo de visão computacional que determina a posição do cone em relação ao robô; *controle_camera* é o controlador de aproximação; *filtro* é o sistema de localização do robô; e *robo_virtual* é o controlador de trajetória.

O nível executivo e a camada de conversão são fáceis de identificar, essa última são os tópicos ligados aos nós *arduino_handler*, *serial_node* e *gpsUBX*, e o nível executivo são esses tópicos juntamente com os nós mencionados e a parte implementada no Arduino. O *serial_node* é responsável por toda a comunicação com os Arduinos, os tópicos ligados a ele são a maneira utilizada para passar informações entre eles, sendo que as mensagens que o Raspberry deseja enviar são gerenciadas pelos tópicos iniciados em *raspberry_* e os provenientes do Arduino pelos tópicos iniciados em *arduino_*.

Dessa maneira, o teste deixa claro a independência entre os módulos do nível funcional e o nível executivo. Dissociando os algoritmos e comportamentos desejados da plataforma física, fazendo com que os mesmos módulos possam ser utilizados não somente em outras situações, como em outras plataformas robóticas. Atingindo os objetivos propostos pelo arcabouço.

4.4 Teste na simulação

O terceiro teste foi utilizado para provar a versatilidade do arcabouço de navegação exibido, adaptando o segundo teste para um ambiente simulado pelo Gazebo, sem a necessidade de fazer quase nenhuma alteração nos níveis funcional e organizacional. Ele foi feito de maneira a emular o máximo possível as condições da competição, portanto, foi utilizado um mapa georreferenciado

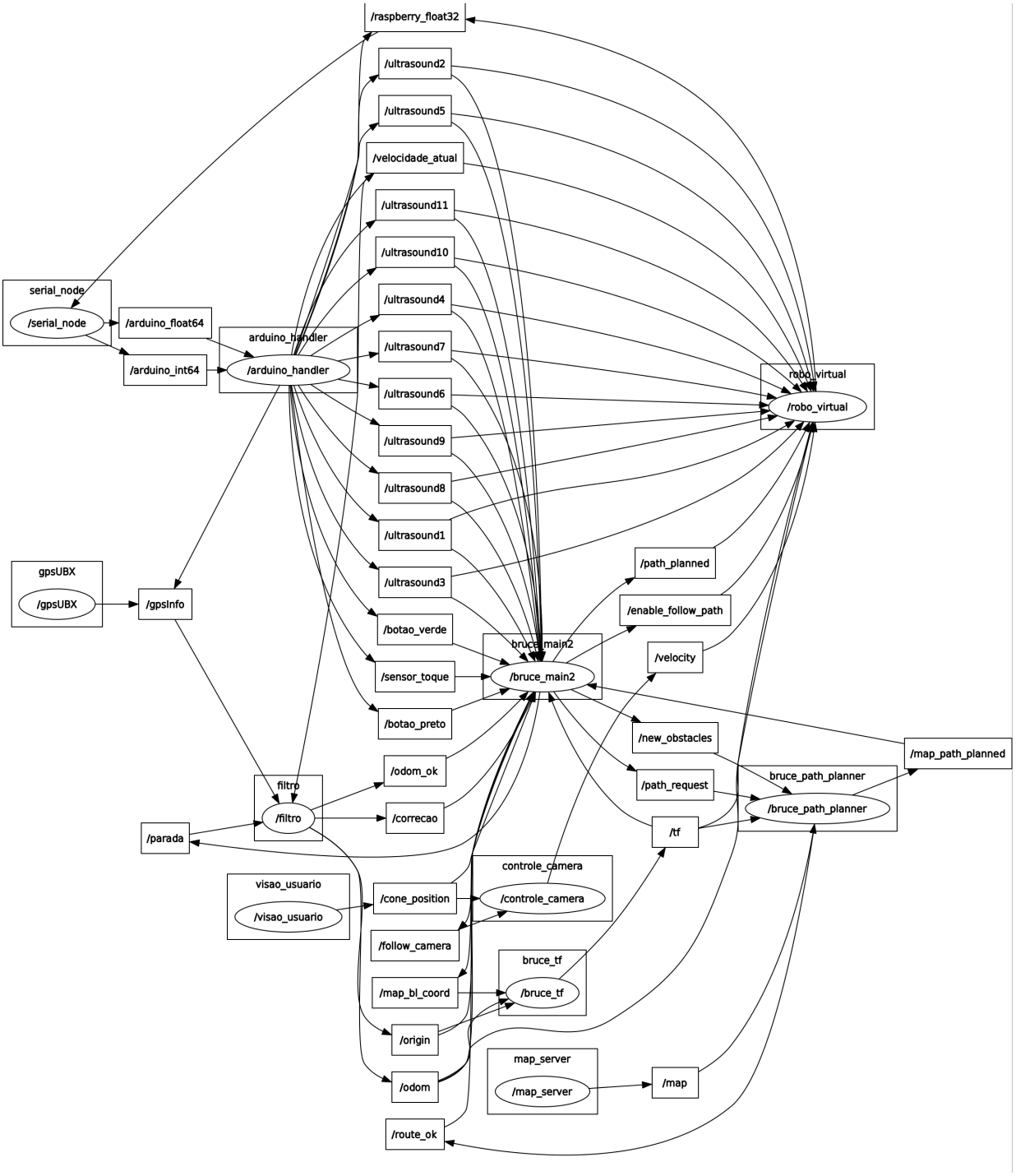


Figura 4.7: Grafo de interação dos nós e tópicos do ROS para o robô Bruce.

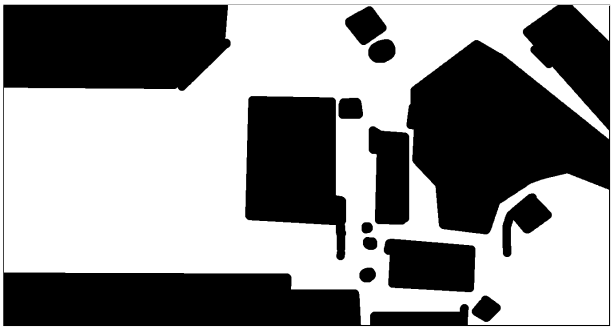
do local real da competição, ilustrado na Figura 4.8a. A partir dele foram construídos o mapa de custo do robô, exibido na Figura 4.8b, e o ambiente de simulação no Gazebo, que pode ser visualizado na Figura 4.8c.

Esse ambiente virtual criado possui as medidas corretas dos prédios e obstáculos em seus comprimento e largura. A altura de todas as estruturas é padrão e fixada em seis metros, o que não faz a menor diferença para o robô, que só analisa o mundo de maneira bidimensional e paralela ao chão.

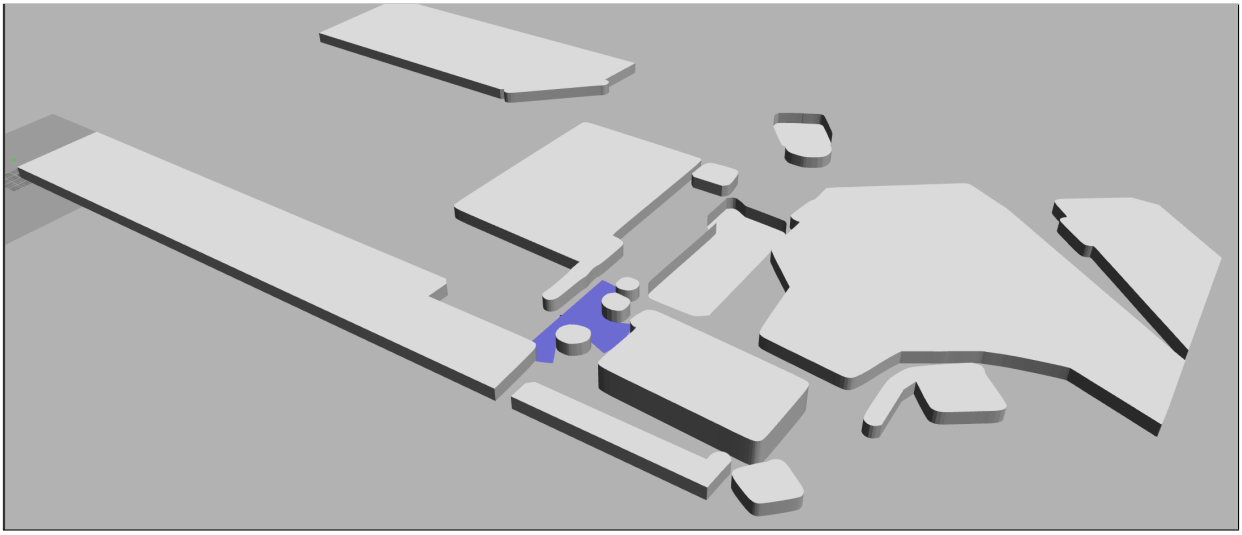
O robô utilizado na simulação foi o Pioneer 3-AT, ele possui quatro rodas diferenciais, fazendo o seu controle ser muito próximo ao do Bruce. Ele também é muito utilizado em laboratórios de pesquisa pelo mundo todo, por esse motivo, ele possui vários modelos para o Gazebo gratuitamente distribuídos pela comunidade que ajuda a o desenvolver. A Figura 4.9 mostra uma imagem mais próxima do robô no ambiente de simulação. O arco azul é a leitura do sensor laser incorporado no robô, que nesse teste será utilizado apenas para ajudar a visualizar a posição e orientação do robô no mundo.

Com o intuito de se aproximar mais da tarefa da competição, nesse teste o robô teve que ir atrás de mais de um objetivo, percorrendo grandes distâncias. A estratégia adotada foi a de lidar

(a) Imagem georreferenciada do local da competição.



(b) Mapa de custo do ambiente.



(c) Ambiente no gazebo que representa o local da competição *Robomagellan*.

Figura 4.8: As representações do ambiente para o teste de simulação.

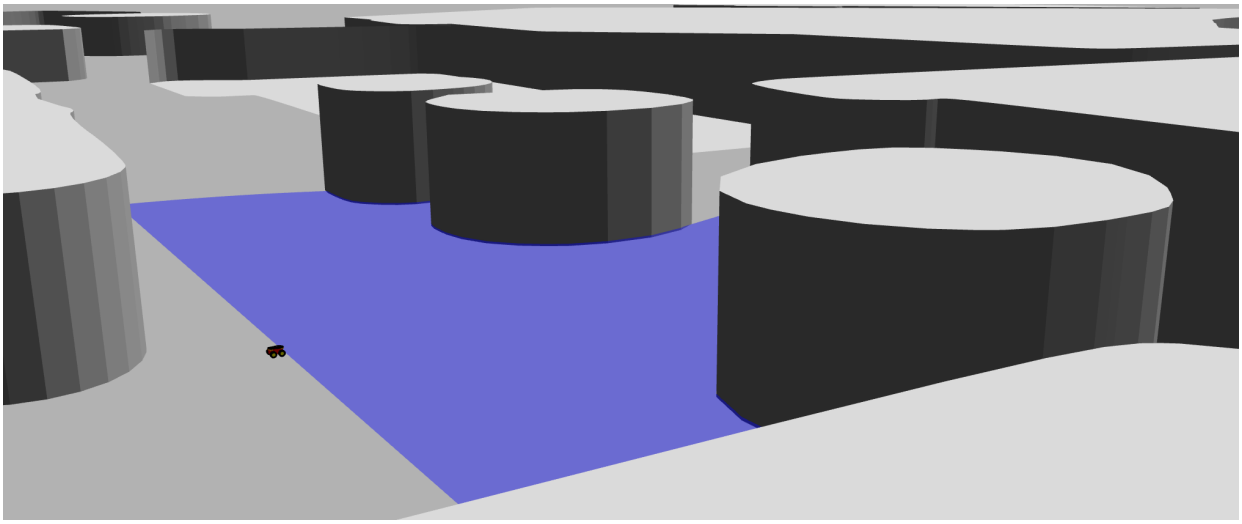
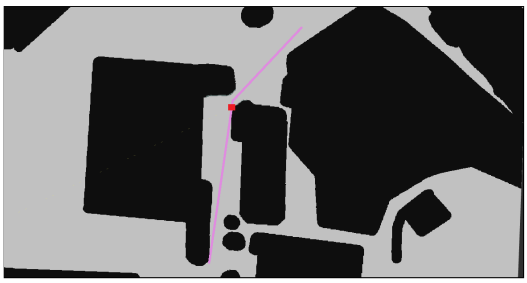


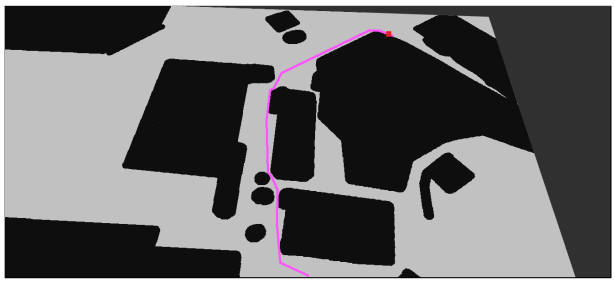
Figura 4.9: Robô Pioneer 3-AT no ambiente de simulação do Gazebo.

com um objetivo de cada vez, ou seja, foi utilizado o mesmo algoritmo do segundo teste, com a única mudança que após alcançar um objetivo, ao invés de parar, ele pegava a próxima meta e calculava a rota para ela. A Figura 4.10 apresenta as três rotas calculadas sendo mostradas por meio do RVIZ.

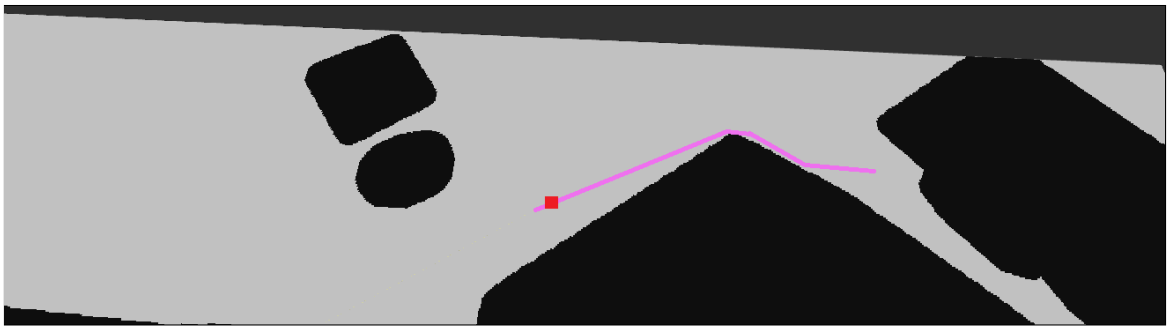
Após planejar e percorrer a primeira rota, apresentada na Figura 4.11, alcançando o primeiro objetivo, ele fez o mesmo para o segundo marco. No entanto, ele falhou no meio da terceira rota, batendo lateralmente em um dos obstáculos. Essa falha é muito importante, pois mostra o grande defeito de sistemas puramente deliberativos, como o robô não consegue seguir o plano



(a) Rota até o primeiro cone.



(b) Rota do segundo ao terceiro cone.



(c) Rota do primeiro ao segundo cone.

Figura 4.10: Rotas calculadas pelo planejador, visualizadas no RVIZ.

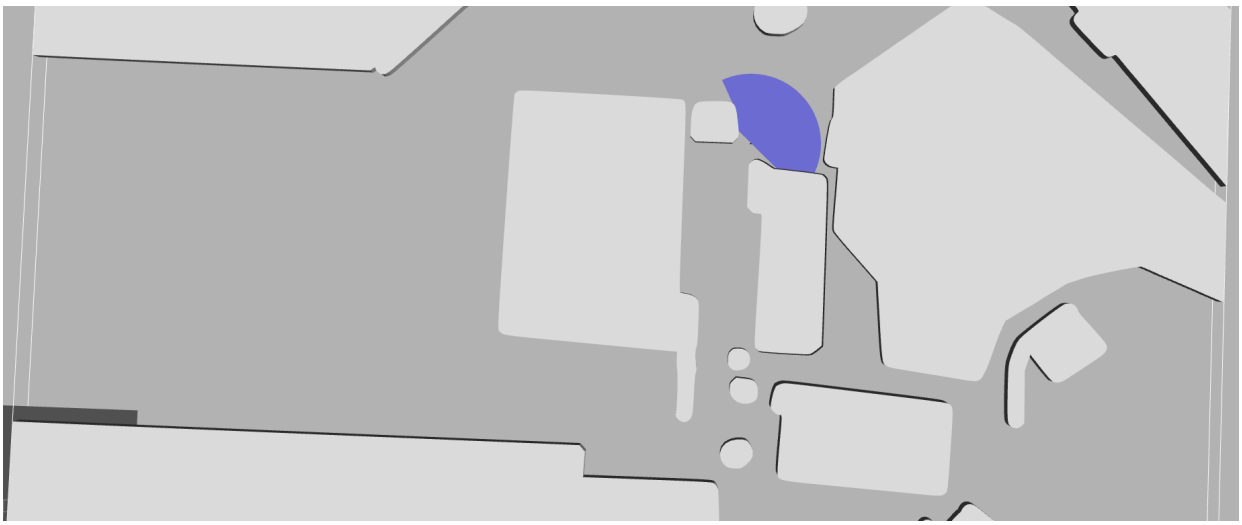


Figura 4.11: Robô Pioneer 3-AT percorrendo a primeira rota planejada no simulador Gazebo.

traçado de maneira perfeita, situações inesperadas ocorrem e acabam causando acidentes. Isso ocorre pois o planejador não leva em conta a tração diferencial do sistema, calculando uma rota que um robô omnidirecional conseguiria seguir perfeitamente, mas não um robô não holonômico. Se as medidas de desvio reativa estivessem funcionando, ele teria sido capaz de detectar que estava se aproximando do obstáculo e tomado as precauções necessárias para evitar a colisão e se manter seguindo a rota.

A Figura 4.12 mostra o caminho percorrido pelo robô durante a simulação, linha azul fina. As três rotas de referências estão representadas por meio de linhas mais grossas das cores verde, vermelho e azul claro, que equivalem, respectivamente, ao trajeto até o primeiro objetivo, até

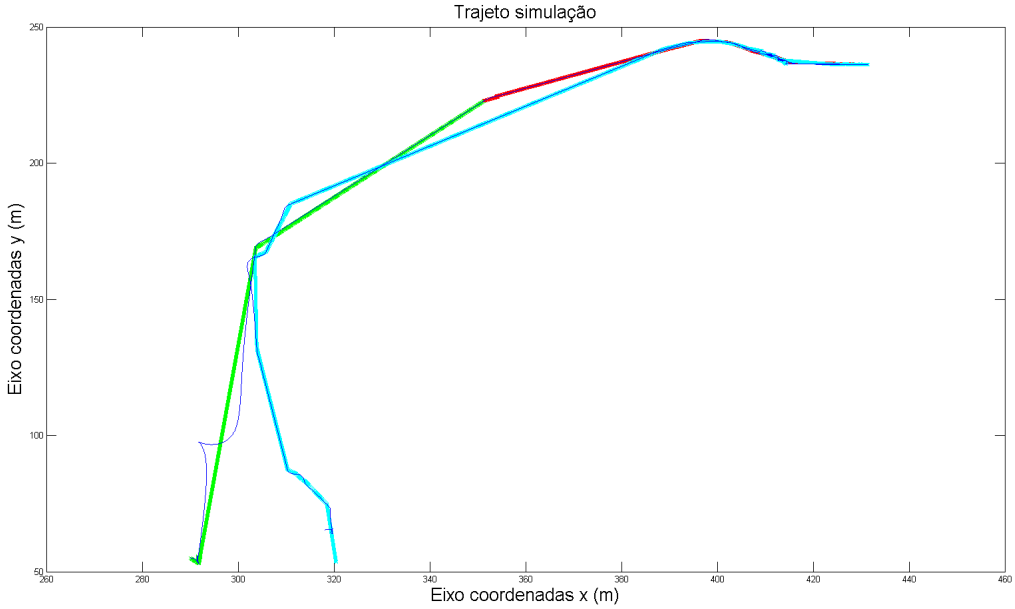


Figura 4.12: Trajeto percorrido pelo robô durante a simulação.

o segundo objetivo e até o terceiro objetivo. Esse gráfico também possibilita ver que a colisão ocorreu bem próxima do final do trajeto, e que o caminho feito no simulador foi muito mais suave do que aquele feito no robô real, mostrado na Figura 4.6. Isso ocorre, principalmente, porque no sistema real existe toda a dinâmica associada ao motor das rodas, deixando o sistema um pouco mais instável do que a simulação.

Apesar da diferença entre ambos sistemas, os robôs conseguiram apresentar um comportamento muito parecido, mostrando que os mesmos módulos do nível funcional e organizacional conseguiram ser utilizados em situações diferentes. A Figura 4.13 ilustra os nós e tópicos utilizados para esse teste, deixando claro que eles foram exatamente os mesmo daqueles apresentados para o teste da odometria. Eles se comunicam com a mesma camada de conversão, mostrando que não faz quase nenhuma diferença para o nível funcional qual é nível executivo atual. A única alteração que teve de ser feita foi mudar o tópico *raspberry_float32* utilizado no Bruce para definir a velocidade dos motores para o tópico *cmd_vel* utilizado pelo Gazebo, e isso só teve de ser feito devido a um erro de projeto que acabou não seguindo as convenções do ROS de comandos de velocidade, o que está nos planos para ser alterado nas próximas versões do sistema.

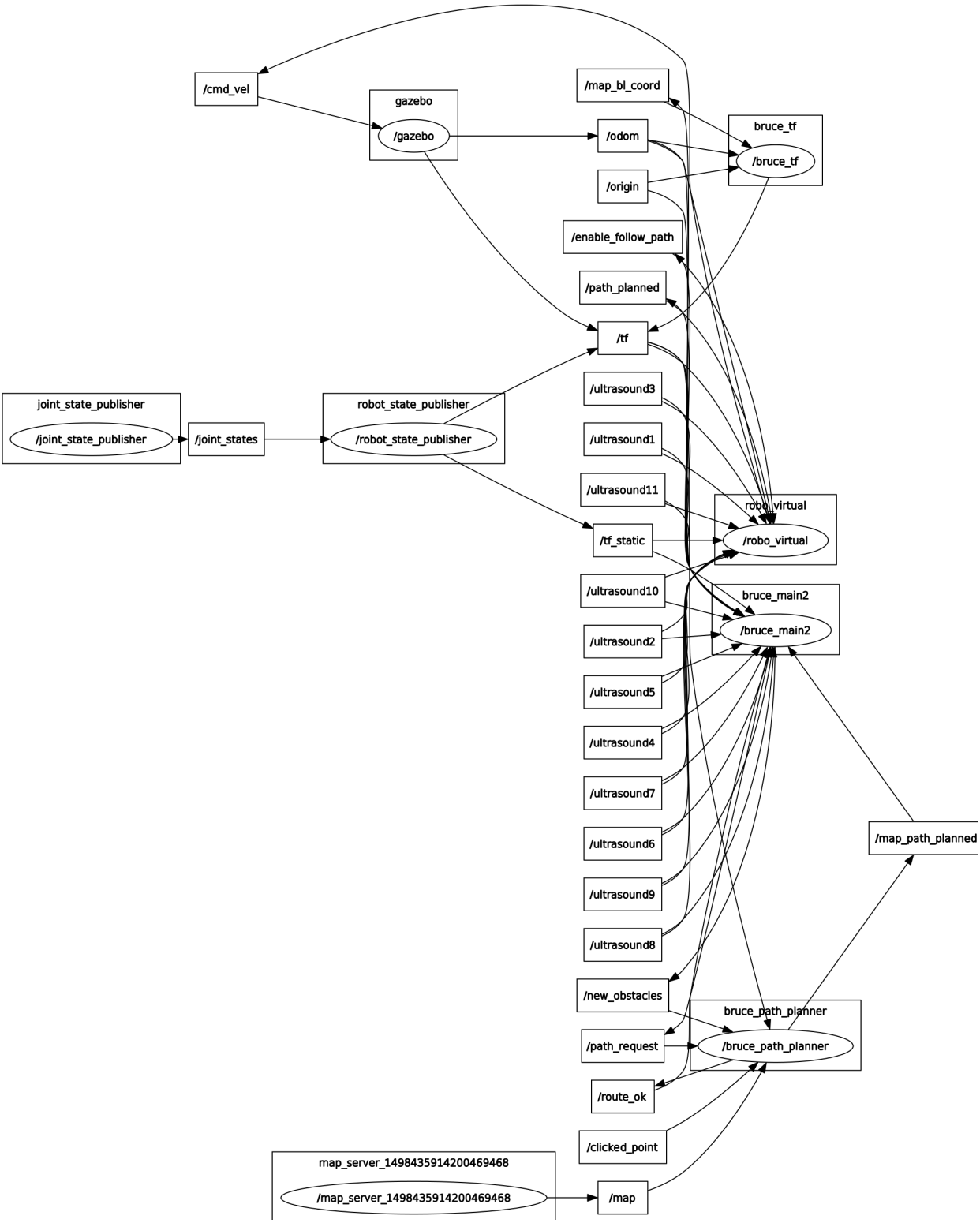


Figura 4.13: Grafo de interação dos nós e tópicos do ROS para o robô Pioneer 3-AT.

Capítulo 5

Conclusões

Esse trabalho apresentou o desenvolvimento de um arcabouço de navegação capaz de implementar diferentes arquiteturas de controle em robôs móveis. Essa estrutura foi embarcada em um robô desenvolvido em conjunto com outros trabalhos de graduação e com a equipe DROID da Universidade de Brasília para superar o desafio proposto na competição *Robomagellan* que faz parte do evento *Robogames*.

Inicialmente foram apresentadas fundamentações de conceitos básico para compreensão dos assuntos desenvolvidos. Primeiramente foram introduzidos os conceitos de sistemas operacionais e simuladores para robótica, para poder embasar a implementação proposta e explicar um pouco sobre um dos ambientes de teste utilizados. Em seguida foi explicado melhor o conceito de arcabouço de navegação, mostrando diversos trabalhos que propõem diferentes estruturas de navegação e apresentando as suas características desejadas. Por fim, foi esclarecido a maioria dos conceitos relacionados ao planejamento de rotas e ao algoritmo RRT, utilizado para implementar o módulo de navegação desenvolvido nesse trabalho.

Em seguida foi apresentado a estrutura do arcabouço de navegação, que é divido em três níveis: o nível organizacional; o nível funcional; e o nível executivo. Juntamente com os detalhes da implementação desse arcabouço no robô Bruce. Cada um desses níveis é responsável por uma função diferente, de forma que os comportamentos consigam ser modularizados ao máximo, criando assim uma maior versatilidade para o arcabouço que pode ser facilmente expandido, reprogramado e utilizado em diferentes plataformas físicas sem a necessidade de grandes mudanças nas suas camadas de processamento mais abstratas.

O nível organizacional comanda toda a estrutura de navegação, e é ele que permite que todo o comportamento do sistema seja reprogramado, mudando facilmente os objetivos e ordem de execução de tarefas do sistema. O nível funcional é responsável por conter todos os diferentes comportamentos, ele é dividido em módulos, cada um com um objetivo específico, que podem interagir entre si para criar comportamentos mais complexos. É ele que permite que o sistema seja expansível e adaptável, pois a mudança entre as interações entre os módulos, assim como a adição de novos módulos cria novos comportamentos sem a necessidade de começar tudo do zero. Por fim, o nível executivo é responsável por pegar esses comandos de alto nível e transformá-los

em baixo nível, deixando a estrutura física do robô como uma caixa preta para o resto do sistema. Dessa forma, os mesmo níveis organizacionais e funcionais podem funcionar em várias plataformas robóticas facilitando a reutilização de recursos, e ajudando o sistema a desenvolver algoritmos mais complexos, sem a necessidade de reimplementar tudo baseado em uma plataforma física específica.

Os testes realizados atingiram seus dois grandes objetivos: mostrar que o arcabouço de navegação proposto seria capaz de cumprir o desafio proposto pela competição *Robomagellan*. E também mostrar que essa estrutura é capaz de atingir todas as metas e características almejados durante a sua criação. Os dois primeiros testes mostraram a capacidade de facilmente mudar o comportamento do robô, apenas mudando a descrição dos módulos que se deseja utilizar. Enquanto o último mostrou que os mesmos níveis funcionais e organizacionais foram reutilizados sem grandes alterações para controlar uma outra plataforma robótica, assim como desejado. Infelizmente, não foi possível implementar todos os comportamentos desejados no robô devido a barreiras financeiras e temporais, porém foi montado uma base sólida para que se continue o desenvolvimento da plataforma de forma a atingir todas as metas desejadas.

Para trabalhos futuros, é proposto terminar a implementação desse sistema em um robô real, incluindo as partes reativas para desvios de obstáculos não planejados. Além disso, poderia expandir-se as capacidades dessa sistema incorporando um sistema de mapeamento, por meio de técnicas de SLAM. Para melhorar os mapas já utilizados nessa solução, poderiam ser desenvolvidos algoritmos que pegam de maneira automática as imagens de satélite durante o seu funcionamento e transformasse-as em mapas de custo, dessa maneira o sistema poderia ser mais geral, não ficando restrito a área previamente definida pelo seu mapa.

Outras ideias envolvem a integração de aprendizagem no arcabouço, para que ele possa gerar sozinho novos comportamentos, sem a necessidade de uma programação direta pelo usuário, dando uma maior autonomia para o sistema. Também propõe-se a incorporação de detectores de falha no sistema, para aumentar a sua robustez caso algo inesperado ocorra. Por fim, incetiva-se a implementação desse arcabouço em outras soluções, de forma que ele possa ser constantemente testado, desenvolvido e melhorado.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] ABNEY, K.; BEKEY, G. A.; LIN, P. *Robot Ethics : the ethical and social implications of robotics*. Cambridge, Massachussetts: The MIT Press, 2012. (Intelligent robotics and autonomous agents).

[2] SIEGWART, R.; NOURBAKHSH, I. R. *Introduction to Autonomous Mobile Robots*. Cambridge, Massachussetts: The MIT press, 2011.

[3] LEONARD, J. J.; DURRANT-WHYTE, H. F. Mobile robot localization by tracking geometric beacons. *IEEE transactions on robotics and automation*, IEEE, v. 7, n. 3, p. 376–382, 1991.

[4] DURRANT-WHYTE, H.; BAILEY, T. Simultaneous localization and mapping (SLAM): part I. *IEEE robotics & automation magazine*, IEEE, v. 13, n. 2, p. 99–110, 2006.

[5] BAILEY, T.; DURRANT-WHYTE, H. Simultaneous localization and mapping (SLAM): Part II. *IEEE Robotics & Automation Magazine*, IEEE, v. 13, n. 3, p. 108–117, 2006.

[6] MATARIĆ, M. J. *The Robotics Primer*. Cambridge, Massachussetts: The MIT Press, 2007.

[7] KOLMANOVSKY, I.; MCCLAMROCH, N. H. Developments in nonholonomic control problems. *IEEE control systems*, IEEE, v. 15, n. 6, p. 20–36, 1995.

[8] TANENBAUM, A. S.; WOODHULL, A. S. *Sistemas operacionais: projeto e implementação*. Porto Alegre: Bookman, 2000.

[9] QUIGLEY, M.; GERKEY, B.; CONLEY, K.; FAUST, J.; FOOTE, T.; LEIBS, J.; BERGER, E.; WHEELER, R.; NG, A. ROS: an open-source Robot Operating System. *ICRA workshop on open source software*, v. 3, p. 5, Maio 2009.

[10] KRAMER, J.; SCHEUTZ, M. Development environments for autonomous mobile robots: A survey. *Autonomous Robots*, v. 22, p. 101–132, 2007.

[11] KOENIG, N.; HOWARD, A. Design and use paradigms for gazebo, an open-source multi-robot simulator. In: IEEE. *Intelligent Robots and Systems, 2004.(IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on*. Sendai, Japão, 2004. v. 3, p. 2149–2154.

[12] LEGER, C. *Darwin2K: An evolutionary approach to automated design for robotics*. Nova Iorque: Springer Science & Business Media, 2012.

[13] IVALDI, S.; PADOIS, V.; NORI, F. Tools for dynamics simulation of robots: a survey based on user feedback. *arXiv preprint arXiv:1402.7050*, 2014.

[14] ALAMI, R.; CHATILA, R.; FLEURY, S.; GHALLAB, M.; INGRAND, F. An Architecture for Autonomy. *The International Journal of Robotics Research*, Sage Publications Sage CA: Thousand Oaks, CA, v. 17, n. 4, p. 315–337, 1998.

[15] SANTOS, V. M.; CASTRO, J. P.; RIBEIRO, M. I. A nested-loop architecture for mobile robot navigation. *The International Journal of Robotics Research*, SAGE Publications, v. 19, n. 12, p. 1218–1235, 2000.

[16] VALAVANIS, K. P.; DOITSIDIS, L.; LONG, M.; MURPHY, R. R. A case study of fuzzy-logic-based robot navigation. *IEEE robotics & automation magazine*, IEEE, v. 13, n. 3, p. 93–107, 2006.

[17] HUAN-CHENG, Z.; MIAO-LIANG, Z. Self-organized architecture for outdoor mobile robot navigation. *Journal of Zhejiang University-SCIENCE A*, Springer, v. 6, n. 6, p. 583–590, 2005.

[18] BAKLOUTI, E.; AMOR, N. B.; JALLOULI, M. Reactive control architecture for mobile robot autonomous navigation. *Robotics and Autonomous Systems*, Elsevier, v. 89, p. 9–14, 2017.

[19] MENG, W.; LIU, E.; HAN, S. A novel collaborative navigation architecture based on de-centralized and distributed ad-hoc networks. In: IEEE. *Communications (ICC), 2012 IEEE International Conference on*. Ottawa, ON, Canada, 2012. p. 606–610.

[20] FIACK, L.; CUPERLIER, N.; MIRAMOND, B. Embedded and real-time architecture for bio-inspired vision-based robot navigation. *Journal of Real-Time Image Processing*, Springer, v. 10, n. 4, p. 699–722, 2015.

[21] LAVALLE, S. M. *Planning algorithms*. Nova Iorque: Cambridge University Press, 2006.

[22] RUSSELL, S.; NORVIG, P. *Artificial intelligence: a modern approach*. Upper Saddle River, New Jersey: Prentice Hall, 1995.

[23] KORF, R. E. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, Elsevier, v. 27, n. 1, p. 97–109, 1985.

[24] STENTZ, A. Optimal and efficient path planning for partially-known environments. In: IEEE. *Robotics and Automation, 1994. Proceedings., 1994 IEEE International Conference on*. San Diego, CA, USA, 1994. p. 3310–3317.

[25] KARAMAN, S.; FRAZZOLI, E. Sampling-based algorithms for optimal motion planning. *The international journal of robotics research*, Sage Publications Sage UK: London, England, v. 30, n. 7, p. 846–894, 2011.

[26] HACKELOEER, A.; KLASING, K.; KRISP, J. M.; MENG, L. Georeferencing: a review of methods and applications. *Annals of GIS*, Taylor & Francis, v. 20, n. 1, p. 61–69, 2014.

ANEXOS

I. DESCRIÇÃO DO CONTEÚDO DO CD

O CD entregue juntamente com esse trabalho contém a versão digitalizada do mesmo, além de uma cópia do repositório digital disponível em `https://github.com/UnbDroid/robomagellan`. Esse repositório contém todo o código utilizado para a implementar o arcabouço desenvolvido e os seus módulos. O CD também disponibiliza um arquivo README que explica melhor a estrutura do código.