Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Implementação de Simulação da Navegação de Robôs de Acionamento Diferencial no Simulador HMR Sim Usando o Controlador PID

Danilo Inácio dos Santos Silva

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia da Computação

Orientador
Prof. Dr. Genaina Nunes Rodrigues

Brasília
2023

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Implementação de Simulação da Navegação de Robôs de Acionamento Diferencial no Simulador HMR Sim Usando o Controlador PID

Danilo Inácio dos Santos Silva

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia da Computação

Prof. Dr. Genaina Nunes Rodrigues (Orientador)
CIC/UnB

MSc. Ricardo Caldas
Chalmers University of Technology

Prof. Dr. Rodrigo Bonifácio
Universidade de Brasília

Prof. Dr. João Luiz Azevedo De Carvalho
Coordenador do Curso de Engenharia da Computação

Brasília, 27 de julho de 2023

Dedicatória

Dedico esta monografia à minha mãe, Dona Ana Lúcia, uma mulher forte e guerreira que sempre me apoiou e me incentivou a estudar.

Agradecimentos

Gostaria de agradecer ao Gabriel que me recebeu de braços abertos no projeto e me guiou através dessa jornada afim de aprimorar esta promissora ferramenta de simulação.

Também a minha orientadora Genaina pela paciência e compreensão quando eu passei por momentos de dificuldade.

Por fim, um agradecimento especial ao meu amigo Kálley que foi quem me apresentou o projeto e que, apesar de estar trabalhando em uma parte diferente dele, compartilhou comigo as dificuldades e conquistas durante o seu desenvolvimento.

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES), por meio do Acesso ao Portal de Periódicos.

Resumo

O uso de robôs para executar tarefas repetitivas e tediosas, tem se tornado nos últimos anos uma tendência não só na área industrial, mas também em ambientes hospitalares, educacionais e residenciais. Essa demanda crescente pelo uso de robôs em ambientes dinâmicos e alocados para tarefas mais complexas, fez crescer o interesse em simuladores de Sistemas Multi-Robôs (SMR).

Os testes e prototipação de sistemas robóticos em ambientes reais podem ser muito dispendiosos, exigindo muito tempo e recursos para preparar o ambiente. Mas essas dificuldades podem ser superadas pelo uso de simuladores que permitem a criação de ambientes seguros de forma mais barata e rápida. Existem hoje vários simuladores com diferentes propostas como Gazebo, MORSE, entre outros. Contudo, todos eles possuem limitações como número baixo de agentes robóticos ou baixa variedade de robôs e componentes que podem ser simulados.

Nesse contexto foi desenvolvido o HMR Sim, um simulador com baixo nível de detalhamento físico, que permite simulações com um maior número de agentes robóticos por um menor custo computacional, e tem por objetivo complementar outros simuladores presentes na literatura. Ele também conta com uma integração com o *middleware* ROS, amplamente utilizado em aplicações robóticas, e que permite com que softwares de controle externos ao simulador, possam comunicar-se com ele utilizando as mesmas interfaces ROS que usariam para o controle de robôs reais.

No entanto, apesar do HMR Sim cumprir com a sua proposta inicial, ele só é capaz de simular o modelo de navegação omni-direcional. Logo, ao simular robôs com modelos de navegação diferentes, a movimentação do robô simulado será muito distante da real, podendo levar ao desenvolvimento de algoritmos incorretos para coordenação de SMRs, visto que o robô poderá percorrer trajetórias diferentes das disponíveis para ele. Além disso, a lógica que determina os caminhos que devem ser percorridos pelo agente robótico, é implementada internamente no simulador, retirando do software de coordenação de robôs a autonomia sobre o planejamento de rotas.

Este projeto tem como objetivo desenvolver um novo mecanismo de navegação que simule robôs de acionamento diferencial e desacopla lógicas de planejamento de rotas que

devem ser transparentes ao simulador. A implementação deste novo mecanismos é feita utilizando o modelo cinemático diferencial e um controlador PID, que é um algoritmo de controle versátil e de fácil implementação.

O novo mecanismo de navegação implementado, agora permite que o HMR Sim simule um modelo de navegação mais comum e com maior autonomia do software externo de controle de robôs, sobre o planejamento de rotas.

Palavras-chave: simulador, ROS, PID, cinemática de acionamento diferencial, behaviour trees

Abstract

The use of robots to perform repetitive and tedious tasks has become a trend in recent years, not only in the industrial field but also in hospital, educational, and residential environments. This increasing demand for the use of robots in dynamic and allocated environments for more complex tasks has sparked interest in Multi-Robot Systems (MRS) simulators.

Testing and prototyping robotic systems in real environments can be costly, requiring a lot of time and resources to set up the environment. However, these difficulties can be overcome by using simulators that allow for the creation of safe environments in a cheaper and faster way. There are currently various simulators with different proposals, such as Gazebo, MORSE, among others. However, they all have limitations, such as a low number of robotic agents or a limited variety of robots and components that can be simulated.

In this context, the HMR Sim was developed as a simulator with a low level of physical detail, allowing simulations with a higher number of robotic agents at a lower computational cost. Its objective is to complement other simulators available in the literature. It also integrates with the ROS middleware, widely used in robotic applications, which enables external control software to communicate with it using the same ROS interfaces they would use to control real robots.

However, despite fulfilling its initial purpose, the HMR Sim can only simulate the omnidirectional navigation model. Therefore, when simulating robots with different navigation models, the movement of the simulated robot will be significantly different from reality. This can lead to the development of incorrect algorithms for coordinating MRSs, as the robot may traverse different paths than those available to it. Additionally, the logic determining the paths to be taken by the robotic agent is implemented internally in the simulator, depriving the robot coordination software of autonomy in route planning.

This project aims to develop a new navigation mechanism that simulates differential drive robots and decouples route planning logics that should be transparent to the simulator. The implementation of this new mechanism utilizes the differential kinematic model and a PID controller, which is a versatile and easy-to-implement control algorithm.

With the newly implemented navigation mechanism, the HMR Sim now allows for simulating a more common navigation model with greater autonomy for external robot control software in route planning.

Keywords: simulator, ROS, PID, differential drive kinematics, behaviour trees

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

DC Direct Current.

DES Discrete Event Simulation.

ECS Entity-Component-System.

HMR Sim Heterogeneous Multi-Robot Simulator.

MRS Multi-Robot System.

PID Proporcional Integral Derivativo.

RAM Random-access memory.

ROS Robot Operating System.

SMR Sistema Multi-Robôs.

Capítulo 1

Introdução

O interesse no uso de sistemas robóticos para executar tarefas repetitivas em ambientes industriais, hospitalares, residenciais e educacionais tem crescido nos últimos anos. Esta demanda fez aumentar as pesquisas e o desenvolvimento de sistemas multi-robôs (sistemas com mais de um agente robótico).

O desenvolvimento de sistemas multi-robôs (SMR) exige testes e prototipação com alto custo de tempo e recursos em ambientes reais, e para mitigar isso os desenvolvedores frequentemente utilizam simuladores de ambientes robóticos. Os simuladores permitem a criação de ambientes seguros de forma mais barata e rápida e dessa forma, os desenvolvedores são capazes de identificar falhas durante a simulação sem que o sistema tenha sido implementado com robôs reais.

Existem na literatura vários simuladores com diferentes propostas como Gazebo [2], MORSE [3] entre outros. No entanto, todos eles possuem limitações como número baixo de agentes robóticos ou baixa variedade de robôs e componentes que podem ser simulados. Além disso, devido ao alto nível de detalhamento físico, as simulações de SMRs têm um alto custo computacional ou nem são suportadas por estes simuladores.

O HMR Sim [1] é uma proposta de simulador que busca complementar os simuladores comumente utilizados em aplicações robóticas, abstraindo das simulações detalhes físicos para permitir uma quantidade maior de robôs a um custo computacional menor. Portanto, ele é útil para testar o controle a nível de missão, ficando a cargo de outro simulador com maior detalhamento físico, os testes a nível de sensores e atuadores. Além disso, ele também conta com uma camada de comunicação com o *middleware* ROS que é amplamente utilizado em sistemas robóticos. Esta integração permite com que um software separado da simulação possa controlar o robô utilizando as mesmas interfaces ROS utilizadas por robôs reais. Dessa forma, os módulos de controle construídos utilizando o simulador podem ser facilmente transportados para agentes robóticos reais.

No entanto, o HMR Sim ainda possui algumas limitações no que tange à navegação dos

robôs simulados. Ele suporta apenas o modelo de navegação omni-direcional e dessa forma a simulação de robôs que usam outros modelos mais comuns, como o diferencial, acaba ficando distante do comportamento real. Além disso, no contexto de coordenação de times de robôs, esta imprecisão pode resultar no desenvolvimento de algoritmos incorretos, uma vez que a trajetória percorrida pelo agente robótico pode ser diferente da disponível para ele.

Outra limitação é que a lógica que determina o caminho que será percorrido pelo agente robótico é implementada internamente no simulador, quando deveria ser de responsabilidade do software externo de controle do robô. Portanto o simulador acaba tirando do sistema simulado a autonomia sobre o planejamento e tomada de decisões de navegação, uma vez que o comportamento já é definido pelo próprio HMR Sim.

A cinemática de acionamento diferencial é um modelo cinemático popular para a movimento de robôs de duas rodas. Neste tipo de modelo a posição do robô é determinada pelas coordenadas x, y e pela orientação θ da sua parte frontal. E para o controle do deslocamento e rotação do agente robótico de uma determinada posição inicial para uma posição final, uma solução simples é o uso de um controlador Proporcional-Integral-Derivativo (PID). Este tipo de controlador é amplamente utilizado pela sua facilidade de implementação e ampla gama de aplicações. Na simulação do movimento de um robô de acionamento diferencial, ele pode ser utilizado na forma de dois controladores operando simultaneamente para controlar a velocidade de angular e a velocidade linear, para movimentar o robô até a sua posição de destino.

A substituição do atual mecanismo de navegação do HMR Sim, pelo modelo de cinemática diferencial aplicado com controladores PID, permitirá uma simulação de navegação mais próximas à de robôs reais. Um exemplo de agente robótico real que poderia ser melhor simulado é o turtlebot, que é uma plataforma robótica de acionamento diferencial muito popular para educação e pesquisa.

A limitação quanto ao acoplamento das lógicas de navegação ao simulador, poderia ser mitigada removendo essas lógicas do simulador e implementando um controlador externo com autonomia sobre o planejamento e decisões de navegação. Este sistema de controle poderia utilizar as interfaces ROS já existentes no simulador e ser desenvolvido usando *Behaviour Trees*, que são uma forma de definir o comportamento de entidades utilizando conceitos de árvores.

1.1 Objetivos Gerais

O objetivo deste trabalho é desenvolver um novo mecanismo de navegação no HMR Sim que simule a navegação de robôs de acionamento diferencial. Também faz parte

deste objetivo o desacoplamento de lógicas de planejamento de rota que não compelem ao simulador.

Para alcançar este objetivo, pretende-se desenvolver um novo mecanismo de navegação que aplica um modelo de cinemática diferencial utilizando um controlador PID. A implementação no HMR Sim será feita desenvolvendo novos sistemas e componentes, que substituirão alguns dos sistemas atualmente responsáveis pela navegação, mas mantendo comunicação com outros sistemas e com a camada de integração com o ROS.

O novo mecanismo de navegação permitirá a simulação da navegação de robôs de acionamento diferencial como o turtlebot, tornando o simulador mais eficiente em simular aplicações robóticas que utilizam este tipo de robô. Além disso, os software de controle executados no simulador terão maior autonomia no planejamento e nas decisões de navegação. Logo, isto complementará a integração com o ROS, no sentido de tornar menos onerosa a migração de uma aplicação robótica do ambiente simulado para um ambiente real e vice-e-versa.

Contribuições

As principais contribuições alcançadas neste projeto foram as seguintes:

- Implementação de uma simulação leve da navegação diferencial utilizando um controlador PID
- Desacoplamento do planejamento de rota do simulador, dando mais autonomia no controle da navegação por software externos ao simulador

1.2 Organização do Trabalho

O Capítulo 1 trará uma contextualização do desenvolvimento do simulador HMR Sim e apresentará algumas de suas limitações atuais. Ao final do capítulo serão apresentados os principais objetivos deste trabalho. Em seguida, o Capítulo 2 apresentará um referencial teórico, em que será detalhado o funcionamento atual do simulador HMR Sim, além de conceitos importantes que serão utilizados nos capítulos posteriores. No Capítulo 3 será descrito em detalhes o processo de implementação dos objetivos apresentados no Capítulo 1. O Capítulo 4 mostrará dois cenários de simulação desenvolvidos como prova de conceito dos sistemas implementados. Por fim, o Capítulo 5 trará as conclusões do trabalho e sugestões para trabalhos futuros.

Capítulo 2

Referencial Teórico

Este capítulo tratará dos conceitos mais relevantes para um completo entendimento das implementações realizadas no projeto. A Seção 2.1 descreve de forma sucinta o funcionamento atual do simulador HMR Sim. Depois, na Seção 2.2 são apresentados os principais conceitos do *middleware* ROS. Na Seção 2.3, são mostrados alguns fundamentos de *Behaviour Trees*. A Seção 2.4, demonstra a cinemática de acionamento diferencial. Por fim, na Seção 2.5, é explicado o funcionamento de um controlador PID.

2.1 HMR Sim

A ferramenta HMRSim [1] é um simulador com foco em auxiliar o desenvolvimento de algoritmos de coordenação de SMRs. Ela abstrai detalhes de implementação dos agentes do sistema e simula apenas os efeitos das capacidades dos mesmos, permitindo um maior desempenho na simulação de times grandes de robôs sem prejudicar sua finalidade principal que é testar algoritmos de coordenação desses times.

Na Figura 2.1 podemos ver a interface de visualização da simulação em execução no navegador. Esta interface possui alguns botões de controle da visualização, um indicador dos passos que estão sendo executados, além do título da simulação. No mapa simulado as paredes são representadas na cor azul, o objeto ao qual o robô poderá interagir aparece na cor vermelha, e o próprio agente robótico é apresentado na cor amarela.

A proposta do simulador HMR Sim é a de complementar outros simuladores bem estabelecidos na literatura, atuando em uma etapa separada com foco na coordenação de múltiplos agentes robóticos. Nesta etapa a execução da simulação é à nível de missão, visando validar a execução de algoritmos de coordenação. Em uma etapa seguinte, poderá então se utilizar um outro simulador com maior detalhamento físico para realizar uma simulação à nível de sensores e atuadores.

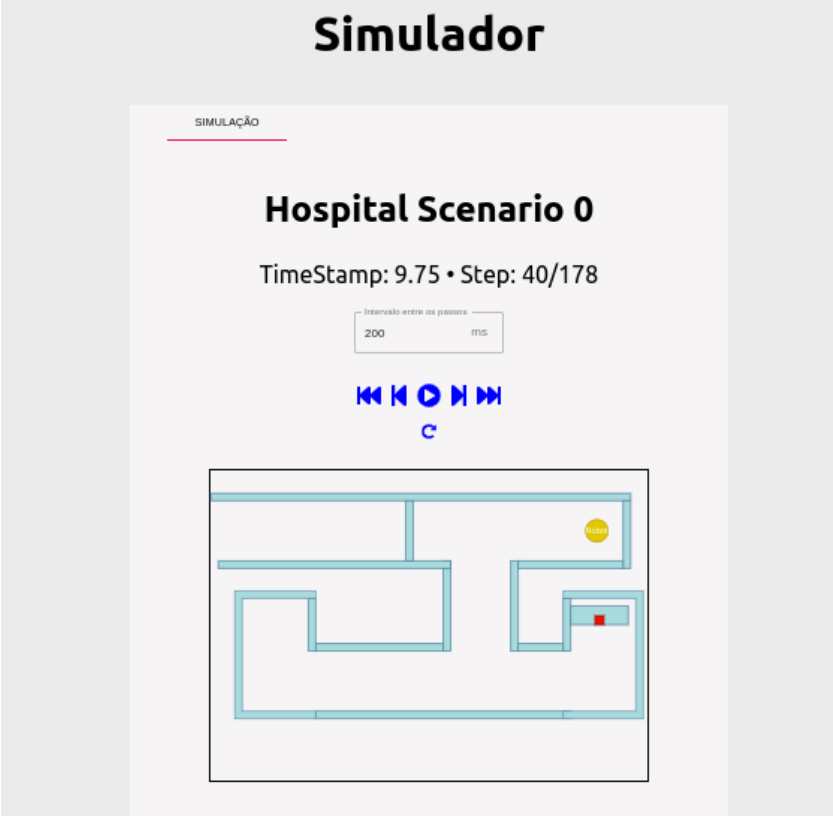

Figura 2.1: Exemplo de simulação executada no HMR Sim. Fonte: De autoria própria.

2.1.1 Arquitetura

O HMR Sim foi construído em Python, uma linguagem de programação versátil e muito popular atualmente. Sua arquitetura é baseada no *design pattern Entity-Component-System (ECS)*, muito utilizada em engines de jogos como *Unity* e *Unreal*, e foi implementada com o esper [4], uma biblioteca de ECS leve com foco em performance e escrita na linguagem Python.

O ECS é um padrão baseado em composição que permite a criação de tipos complexos combinando componentes de vários tipos, em vez de herdar de uma classe pai, mas diferentemente do padrão *Object-Oriented Composition*, ele também dá suporte a separação dos dados e da lógica. Dentro desse padrão as entidades são identificadores de coleções de componentes que, por sua vez, armazenam dados, mas a lógica que age sobre esses dados é implementada nos sistemas. Portanto, um sistema que atua sobre um determinado tipo de componente irá atuar sobre todas as entidades que possuem esse tipo. Por exemplo, uma entidade que possui a capacidade de movimentar-se, poderia possuir componentes de posição e velocidade que seriam manipulados por um sistema de movimentação. Tal sistema utilizaria das informações do componente de velocidade para aplicar uma lógica de atualização das coordenadas da entidade no componente de posição para realizar o

movimento.

Utilizando esse padrão de arquitetura, o HMR Sim torna-se modular e facilmente extensível, pois os dados e a lógica são separados em componentes e sistemas que podem ser facilmente implementados para adicionar novos comportamentos às entidades da simulação. Além disso, o uso de componentes, possibilita que as capacidades dos objetos da simulação possam ser facilmente adicionadas e removidas inclusive de forma dinâmica em tempo de execução.

As simulações no HMR Sim podem ser controladas por técnicas distintas a depender do tipo de sistema simulado.

Simulação por tempo discreto com intervalo de incremento fixo: é uma técnica de simulação bem estabelecida em que o incremento de tempo da simulação entre t_i e $t_i + 1$ é pré-definido e fixo. Cada variável de estado do sistema é uma função de variáveis de estado até o momento anterior. Essa técnica é ideal para sistemas que mudam constantemente, como por exemplo a temperatura de um ambiente ao longo do tempo. Simulações desse tipo são suportadas pela biblioteca *esper* [4] e agem sobre os sistemas compatíveis com ela.

Simulação por eventos discretos (DES, *Discrete Event Simulation*): nessa técnica um evento e possui um tempo t e uma função f que altera o estado da simulação e também pode criar outros eventos, que serão adicionados a uma fila de eventos. Cada estado $s_i + 1$ é o resultado de processar o primeiro evento da fila sobre o estado s_i. Essa técnica é indicada para simular sistemas que mudam de forma esporádica ao longo do tempo. A biblioteca que a implementa no HMRSim é a *simpy*, um *framework* que gerencia os eventos e sua execução em sistemas compatíveis com DES.

O HMR Sim conta com um conjunto de sistemas considerados críticos já implementados, dos quais abordaremos apenas os de navegação que serão os mais relevantes para este projeto.

2.1.2 Sistemas de Navegação do HMR Sim

Nesta seção serão trados cinco sistemas responsáveis pelo mecanismo de navegação presente no HMR Sim atualmente. O conhecimento destes sistemas será importante para o entendimento das limitações atuais de navegação do simulador e a compreensão das novas implementações que os substituirão.

Sistema de Comandos de Navegação (*GoToDESProcessor*)

Este sistema é responsável por tratar comandos de navegação para entidades da simulação. Uma vez que ele receba um comando para que uma determinada entidade navegue

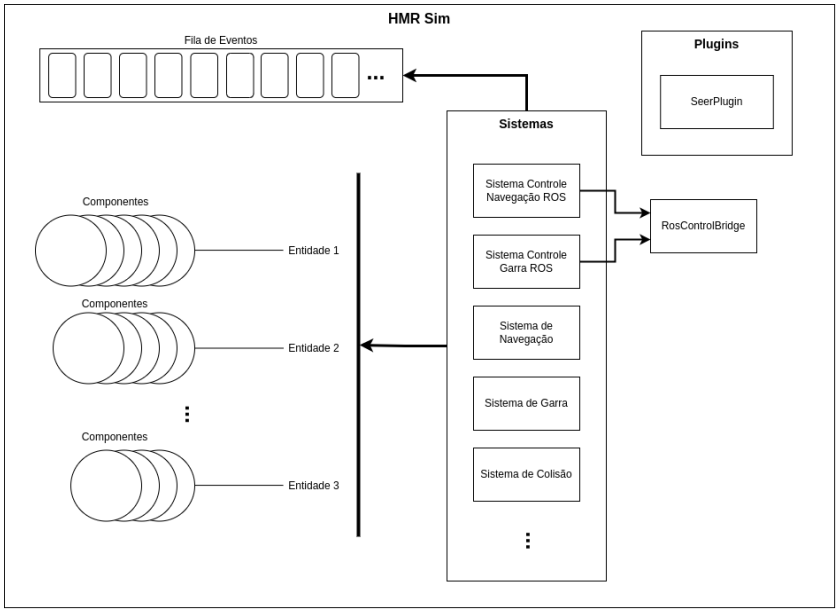

Figura 2.2: Diagrama representando a arquitetura do HMR Sim. Fonte: De autoria própria baseado em [1], página 35.

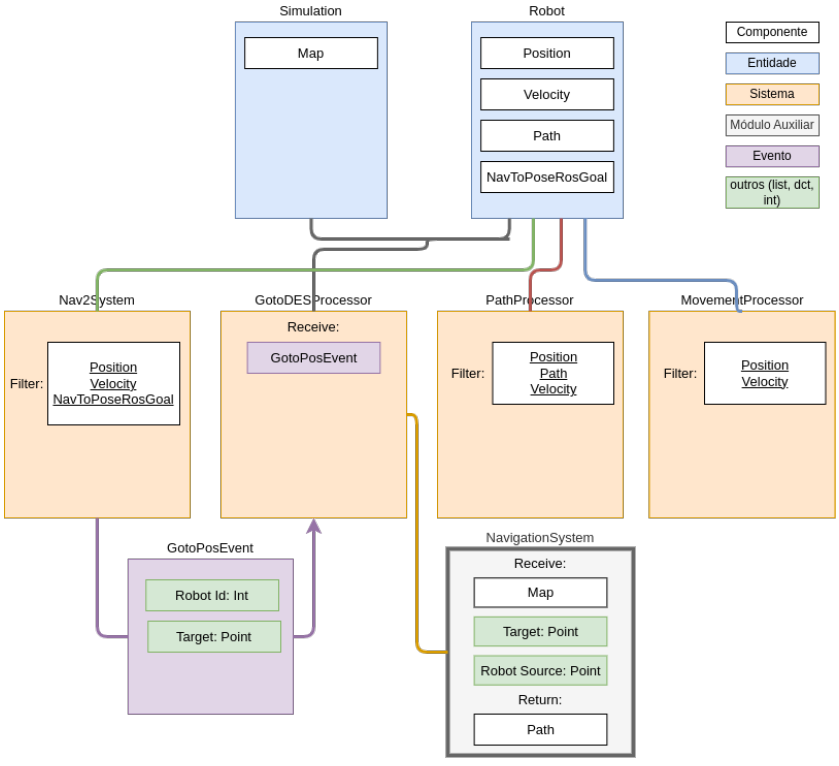

Figura 2.3: Diagrama ECS representando o mecanismo de navegação do HMR Sim. Fonte: De autoria própria.

de um ponto A para um ponto B, ele obtém a partir do sistema de navegação o caminho para que a entidade chegue a este objetivo. O caminho é um componente contendo os

pontos que devem ser percorridos e que é adicionado à entidade que recebeu o comando.

Neste projeto o tratamento de comandos de navegação será feito por um novo sistema, que apenas adicionará um componente contendo um único ponto de destino à entidade alvo do comando. Desta forma, o simulador será responsável por realizar a movimentação até um ponto de cada vez sem qualquer controle sobre quais pontos serão percorridos e qual será o destino final.

Sistema de Caminhos (*PathProcessor*)

No sistema de caminhos são calculados os valores do componente de velocidade para que entidades que possuem um componente de caminho (*Path*) possam se movimentar até os pontos que compõem está rota. Ao alcançar o último ponto da rota, o componente de *Path* é removido da entidade e um evento indicando que o caminho foi finalizado é colocado na fila de eventos para comunicar os outros sistemas do simulador.

Neste sistema, apenas a velocidade linear é calculada, pois ele não altera a orientação do robô. Os componentes x e y do vetor de velocidade linear são determinados com base no módulo da velocidade do robô e nas diferenças dx e dy da sua posição atual e a posição de destino. Dessa forma, são aplicadas velocidades de módulo constante e com direção e sentido determinados por dx e dy, resultando em movimentos em linha reta na vertical, horizontal e diagonal.

Este sistema será substituído por um novo sistema que implementará o modelo de cinemática diferencial para calcular os valores do componente de velocidade.

Módulo de Navegação (*NavigationSystem*)

É um módulo auxiliar que determina a rota que uma entidade deve percorrer para chegar a um ponto de destino. Ele obtém a rota a partir do componente de Mapa que faz parte da entidade simulação. Este componente armazena pontos de interesse no mapa (POIs), que são coordenadas identificadas por uma tag identificadora e inseridas no mapa da simulação para serem utilizadas nos comandos de navegação. Além dos POIs, o componente *Map* também guarda um grafo dos pontos que podem ser percorridos, obtidos a partir de *map-path*s que são setas inseridas no mapa para indicar os caminhos seguros. A rota calculada a partir destes dados é disponibilizada na forma de um componente de caminho (Path) contendo os pontos que devem ser percorridos pela entidade para chegar ao destino indicado pelo comando de navegação.

Este módulo deixará de ser utilizado para que o software de coordenação de robôs tenha autonomia sobre o planejamento e decisões de navegação.

Sistema de Movimentação (*MovemmentProcessor*)

É neste sistema em que a movimentação propriamente ocorre. Ele atua sobre uma entidade aplicando um deslocamento em seu componente de posição de acordo com os dados do componente de velocidade da mesma.

Não será necessário realizar praticamente nenhuma alteração neste sistema para atender aos objetivos do projeto.

Sistema de Interface de Navegação do ROS (*Nav2System*)

É o sistema que implementa uma interface de comunicação com o ROS para o controle da navegação do robô. Ele recebe a partir do ROS os comandos de navegação do software de coordenação do agente robótico e os repassa na forma de eventos para o sistema de comandos de navegação do simulador.

Alguns pequenos ajustes serão necessários para que este sistema se comunique corretamente com o novo sistema de comandos de navegação que será implementado neste projeto.

2.1.3 Integração do HMR Sim com o *middleware* ROS

A integração entre HMR Sim e ROS é feita através de alguns sistemas que implementam interfaces de comunicação com o ROS para permitir o controle da navegação e do componente de garra dos agentes robóticos simulados. Estas interfaces são definidas no Nav2 [5] e no Movit [6], dois importantes projetos que estendem as funcionalidades do ROS. Como eles são projetos bastante utilizados, a implementação de suas interfaces permite com que mais aplicações robóticas possam se comunicar com o HMR Sim. Dessa forma, o software de coordenação dos robôs simulados pode ser mais facilmente transportado para uma aplicação robótica real, visto que as interfaces utilizadas para controlar o agente robótico na simulação são as mesmas utilizadas em robôs reais.

Além dos sistemas que implementam estas interfaces, também faz parte da integração o sistema de controle ROS, que gerencia as comunicações com o simulador através do *middleware*. Este sistema sincroniza a execução dos sistemas de comunicação com o loop principal da simulação e abstrai dos mesmos lógicas genéricas de instanciamento e finalização da interação com o ROS.

2.2 *Robot Operating System* (ROS)

O *Robot Operating System* (ROS) é um conjunto de bibliotecas e ferramentas *open-source* muito utilizado no desenvolvimento de aplicações robóticas [7]. Ele funciona como

um sistema operacional distribuído provendo abstração de hardware, troca de mensagens entre processos, gerenciamento de pacotes, entre outras funcionalidades comumente usadas. Por ter uma característica distribuída, um dos conceitos mais importantes no ROS é o de Nodes (ou Nó).

2.2.1 Node

Um node é um processo independente responsável por uma tarefa específica e que se comunica com outros nodes para realizar operações mais complexas em um sistema robótico. Ele pode ser responsável por diferentes funcionalidades dentro de uma aplicação, como processar dados de sensores, controlar motores, realizar um planejamento de rota e etc.

A abordagem distribuída baseada em nodes torna o ROS modular e flexível, o que permite o desenvolvimento de sistemas complexos de forma mais simples, dividindo-os em funcionalidades que podem ser implementadas em nodes independentes. Esta modularidade facilita a reutilização de código, a substituição de partes específicas do sistema, além de isolar falhas em apenas um node.

2.2.2 Comunicação entre Nodes

O ROS possui diferentes tipos de comunicação entre nodes. Aqui trataremos dos Tópicos (*topics*), Serviços (*services*) e *Actions*.

Tópicos

Os tópicos são um mecanismo de comunicação unidirecional em que os produtores de dados são chamados de *publishers* e os consumidores são chamados de *subscribers*. Um node não tem conhecimento dos outros nodes com quem está se comunicando através do tópico, ele apenas publica informações ou se inscreve em um tópico de interesse para consumi-las.

Serviços

Os serviços são um mecanismo de comunicação baseado no modelo cliente e servidor, em que a comunicação é feita por duas mensagens, uma de requisição e outra de resposta. Ao contrário dos tópicos que permitem que um node inscrito receba continuamente dados publicados, os serviços só retornam um dado para o cliente quando requisitado.

Um node ROS servidor disponibiliza um serviço e um node cliente o solicita através de uma requisição. Quando ela chega ao servidor ele retorna uma mensagem de resposta através do serviço para o cliente.

Actions

As *actions* são um outro tipo de comunicação entre nodes ROS, com funcionamento semelhante aos serviços, mas destinada a tarefas de longa duração. Elas são composições dos mecanismos de comunicação apresentados anteriormente, consistindo em três partes: um serviço de objetivo, um serviço de resposta e um tópico de *feedback*.

A *actions* do cliente envia uma requisição através do serviço de objetivo e a *actions* do servidor retorna uma resposta aceitando ou não o objetivo. Caso o objetivo tenha sido aceito, a *actions* do cliente envia uma nova requisição, agora através do serviço de resposta, e fica recebendo mensagens pelo tópico de *feedback* enquanto aguarda um retorno. Ao final da execução do objetivo a publicação de *feedback* é encerrada e a *actions* do servidor retorna uma resposta.

2.3 *Behaviour Trees*

As *Behaviour Trees* são estruturas de tomada de decisão muito utilizadas para o controle de comportamento em sistemas robóticos, inteligência artificial e video games [8]. Seus nós internos são composições de comportamentos e são chamados de nós de controle, enquanto que seus nós folhas representam atuações ou detecções e são chamadas de nós de execução. A execução de uma *Behaviour Trees* começa a partir do seu nó raiz que envia *ticks* com uma dada frequência para os seus nós filhos. Os *ticks* são etapas de atualização discreta que são propagadas pela árvore ativando os nós, que então retornam para o seus nós pai um status que pode ser *Running*, *Success* ou *Failure*. Segue abaixo os principais nós presentes em uma *Behaviour Trees* e os seus respectivos comportamentos:

2.3.1 Nós de Controle

São nós internos responsáveis pela forma como a árvore é percorrida de acordo com os status dos seus nós filhos. Todos os tipos de nós internos podem ter qualquer quantidade de filhos, exceto pelo *Decorator*, que pode ter apenas um. Os nós filhos podem ser nós de controle ou de ações. Os quatro tipos de nós de controle são:

- Sequence: é representado por um quadrado com uma seta para a direita. Este tipo de nó executa os nós filhos em ordem e retorna Success quando todos os filhos retornam Sucess, caso contrário ele retorna o status do filho, Running ou Failure e para a execução.
- *Fallback*: é representado por uma interrogação e possui um comportamento oposto ao do *Sequence*. Ele executa os nós filhos em ordem e retorna *Failure* se todos os

nós filhos retornam *Failure*, caso contrário ele retorna o status do filho, *Running* ou *Sucess* e para a execução.

- *Parallel*: é representado por um quadrado com duas setas em paralelo para a direita. Este nó executa todos os seus nós filhos e retorna *Success* se um número M de filhos retornarem *Success*, retorna *Failure* se mais que $N - M$ filhos retornarem *Failure* e retorna *Running* em qualquer outro caso.
- *Decorator*: possui apenas um nó filho e seu comportamento pode ser determinado pelo desenvolvedor da *Behaviour Tree* para adicionar semântica ou mudar o status de retorno de um nó. Um exemplo clássico seria o Inverter que inverte o status do nó filho.

2.3.2 Nós de Execução

São os nós folha da árvore e são divididos em dois tipos:

- *Action*: é representado por um retângulo. Ele pode executar várias operações retornando *Running* até chegar a um estado final, Success ou *Failure*.
- *Condition*: este tipo de nó é representado por uma elipse. Ele checa uma condição e retornam *Failure* ou *Success*.

2.3.3 Bibliotecas Utilizadas

Este projeto fará uso de *Behaviour Trees* para implementar um software de coordenação de robôs em uma trajetória de lemniscata. Para tal, a biblioteca que será utilizada é a *Py Trees* [9], uma biblioteca do Python que implementa *Behaviour Trees* com foco no controle de robôs. Além dela, também será utilizada a *Py Trees ROS* [10], que estende a biblioteca *Py Trees* adicionando funcionalidades específicas para interagir com o ROS.

2.4 Modelo Cinemático Diferencial

O modelo cinemático de um robô móvel é um modelo matemático que descreve o seu movimento em termos de velocidades e posição, desconsiderando as causas [11]. Utilizando modelos cinemáticos é possível projetar robôs mais apropriados para determinadas tarefas e desenvolver algoritmos de controle corretos. Os modelos cinemáticos podem ser classificados em diretos, em que o modelo descreve o movimento do agente robótico obtido como resposta a uma entrada de controle, e indiretos, em que, dado um movimento desejado, obtemos através do modelo as entradas de controle para alcançá-lo. Neste projeto

o foco é na simulação da movimentação de robôs, portanto sempre que o termo 'modelo cinemático' for usado, estará referindo-se ao modelo cinemático direto.

Existem robôs com diferentes tipos de navegação como omni-direcional, *Ackerman* ou diferencial. Dentre estes os que mais se destacam na categoria de robôs de serviço, são os de acionamento diferencial devido a sua simplicidade e facilidade de controle.

Este tipo de robô é caracterizado por ter duas rodas tracionadas independentes em um eixo, o que permite com que cada roda seja controlada separadamente. Dessa forma, para que o agente robótico siga em linha reta, basta acionar as duas rodas com a mesma velocidade, e para mudar a direção, basta ajustar a velocidade de uma roda em relação a outra.

A Figura 2.4 mostra uma representação no plano cartesiano do comportamento de um robô de acionamento diferencial. Nela, o robô de duas rodas de raio r com velocidades lineares V_L e V_R e separadas por um eixo de comprimento L, movimenta-se por uma trajetória curvilínea de raio R em torno do ponto ICR.

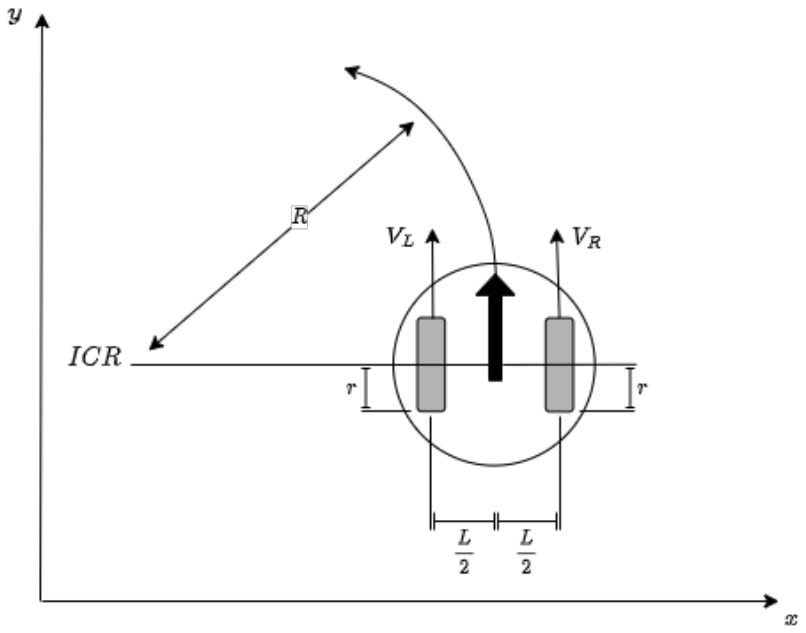

Figura 2.4: Cinemática de um robô de acionamento diferencial. Fonte: De autoria própria.

A velocidade angular ω da trajetória de ambas as rodas do robô em torno do ponto ICR, é a mesma. Portanto, dado que a velocidade linear das rodas é igual a velocidade angular das mesmas multiplicado pelas suas respectivas distâncias até o eixo de rotação , V_L e V_R podem ser expressas por (2.1) e (2.2), respectivamente. Dessa forma, ω pode ser obtidos em termos de V_L na equação (2.3) ou de V_R na equação (2.4).

$$V_L = \omega \left(R - \frac{L}{2} \right) \tag{2.1}$$

$$V_R = \omega \left(R + \frac{L}{2} \right) \tag{2.2}$$

$$\omega = \frac{V_L}{R - L/2} = \frac{r\omega_L}{R - L/2} \tag{2.3}$$

$$\omega = \frac{V_R}{R + L/2} = \frac{r\omega_R}{R + L/2} \tag{2.4}$$

Substituindo (2.3) em (2.4) e isolando ω, obtêm-se (2.5), em que a velocidade angular do robô é expressa em função das velocidades de suas duas rodas.

$$\omega = \frac{V_R - V_L}{L} = \frac{r(\omega_R - \omega_L)}{R - L/2} \tag{2.5}$$

Da mesma forma, também pode-se obter a velocidade linear do agente robótico, pois dado que $V = \omega\, R$, igualando (2.3) e (2.4) e isolando R, obtemos (2.6), que multiplicado por (2.5), resulta em (2.7).

$$R = \frac{L(\omega_L + \omega_R)}{2(\omega_R - \omega_L)} \tag{2.6}$$

$$v = \frac{V_R + V_L}{2} = \frac{r(\omega_R + \omega_L)}{2} \tag{2.7}$$

A partir dos valores de v e ω , pode-se determinar a posição e a orientação do robô no plano, aplicando uma rotação θ em torno do eixo z. Esta expressão é representada de forma matricial na equação (2.8).

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = R_z(\theta) \begin{bmatrix} v \\ 0 \\ \omega \end{bmatrix} = \begin{bmatrix} cos\theta & sen\theta & 0 \\ sen\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} v \\ 0 \\ \omega \end{bmatrix} \tag{2.8}$$

Substituindo (2.5) e (2.7) em (2.8), obtêm-se o modelo cinemático diferencial em (2.9).

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} rcos\theta/2 & rcos\theta/2 \\ rsen\theta/2 & rsen\theta/2 \\ r/L & -r/L \end{bmatrix} \begin{bmatrix} \omega_R \\ \omega_L \end{bmatrix} \tag{2.9}$$

2.5 Controlador PID

O controlador *Proporcional-Integral-Derivativo* (PID) é um algoritmo de controle versátil e de fácil implementação muito utilizado na industria e na literatura. Ele faz parte da família de controladores de malha fechada que caracterizam-se por monitorarem continuamente o erro do sistema para ajustar o sinal de controle, afim de aproximar o erro à zero [12]. No controlador PID este ajuste é realizado através da soma das ações Proporcional (P), Derivativa (D) e Integral (I), aplicadas sobre o erro, mas podendo também ser utilizado com subconjuntos destas, como P, PI, e PD.

A Figura 2.5 ilustra o diagrama de blocos de um controlador PID. Nela a referência de entrada do sistema, *setpoint* (SP), é subtraída pela variável do processo medida pelo sensor, *process variable* (PV), a fim de obter-se o erro. Em seguida, são aplicadas as ações proporcional, diferencial e integral sobre o erro, e seus resultados são somadas para obter-se o sinal de controle ou variável manipulada, *manipulated variable* (MV). O MV é enviado ao processo, que então gera um PV que será novamente medido pelo sensor, realimentando o loop. Dessa forma, o controlador continua atuando sobre o processo enquanto o erro for diferente de zero, e.g., PV for diferente de SP.

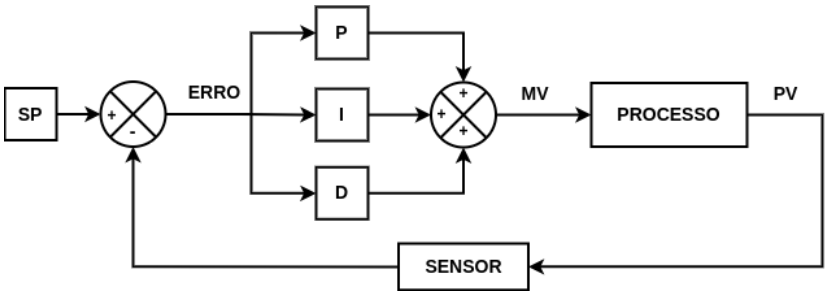

Figura 2.5: Diagrama de Blocos de um Controlador PID. Fonte: De autoria própria.

Cada uma das ações de controle do PID são o produto de um ganho por uma expressão do erro e atuam de forma diferente no ajuste do sinal de controle. Os ganhos dos termos do controlador vão depender do processo controlado e podem ser obtidos de forma empírica ou utilizando métodos de sintonia como o de oscilação de Ziegler-Nichols [12].

Ação Proporcional (P)

Fornece uma contribuição proporcional ao erro. Aumentar o seu ganho K_P reduzirá o tempo de resposta do controlador, mas o PV poderá ultrapassar ou ficar abaixo do SP em regime permanente. A ação proporcional é dada pela expressão (2.10).

$$K_P e(t) \tag{2.10}$$

Ação Derivativa (D)

Está ação é proporcional a taxa de variação do erro. Também é chamada de modo preditivo, pois atua sobre a tendência do erro e dessa forma antecipa a reação do controlador. O aumento da sua contribuição K_D torna a resposta mais rápida e menos oscilatória, mas em regime permanente a sua ação torna-se nula, pois o erro torna-se constante. A ação derivativa é calculada pela expressão (2.11), com o calculo da derivada do erro podendo ser aproximado pela equação (2.12).

$$K_D \frac{de(t)}{dt} \tag{2.11}$$

$$\frac{de(t)}{dt} \approx \frac{e(t) - e(t - \Delta\, t)}{\Delta\, t} \tag{2.12}$$

Ação Integral (I)

A ação integral age sobre o erro acumulado, o que implica em uma reação de controle lenta. Mesmo para um erro pequeno a ação integral acumula-se ao longo do tempo tendendo o erro a zero no estado estacionário. Contudo, se aplicada de forma isolada pode resultar em um fenômeno chamado de *windup*, em que a ação integral satura o controlador sem que ele consiga aproximar o erro à zero. O aumento da sua contribuição K_I resulta em uma redução grande do erro em estado estacionário, mas aumenta o tempo de estabilização e pode fazer com que PV ultrapasse ou fique abaixo de SP em regime permanente. Ela é dada pela expressão (2.13), podendo ter o seu fator integral aproximado pela equação (2.14)

$$K_I \int_o^t e(\tau)\, d\tau \tag{2.13}$$

$$\int_o^t e(\tau)\, d\tau \approx \sum_{n=0}^{t/\Delta\, t} e(i\Delta\, t)\Delta\, t \tag{2.14}$$

O controlador PID completo pode ser expresso pela equação (2.15), em que o sinal de controle $u(t)$ é resultado da soma das contribuições das três ações de controle.

$$u(t) = K_P e(t) + K_I \int_o^t e(\tau)\, d\tau + K_D \frac{de(t)}{dt} \tag{2.15}$$

Capítulo 3

Implementação de Simulação da Navegação de Robôs de Acionamento Diferencial no HMR Sim

Neste capítulo serão apresentadas as novas implementações realizadas no HMR Sim para alcançar os objetivos deste projeto. Ele está estruturado da seguinte forma: A Seção 3.1 discorrerá sobre o novo mecanismo de navegação implementado no simulador. A Seção 3.2 tratará do desacoplamento do planejamento de rota da simulação de movimentação. A Seção 3.3 apresentará um novo sistema que simula a movimentação baseado no modelo cinemático diferencial. Por fim, a Seção 3.4 trará o desenvolvimento de um controlador PID que complementa o sistema apresentado na seção anterior, ajustando as velocidades do robô para alcançar um determinado destino.

Vale ressaltar que o termo mecanismo refere-se a um conjunto de sistemas, componentes e eventos do HMR Sim, que juntos implementam uma determinada funcionalidade do simulador, como por exemplo a simulação de navegação.

3.1 Novo Mecanismo de Navegação

Foram adicionados novos sistemas e reaproveitados alguns já existentes para implementar um novo mecanismo de navegação no HMR Sim. A Figura 3.1 ilustra este mecanismo com destaque em vermelho para as novas implementações realizadas neste projeto.

Os sistemas desenvolvidos foram os responsáveis por receber os comandos de movimentação e por controlar as velocidades que serão aplicadas à entidade para que o movimento ocorra. Este último foi complementado com um módulo de controle para auxiliar na simulação da movimentação de robôs diferenciais. Também foram adicionadas novos

componentes, para que os novos sistemas atuem sobre a entidade, e novos eventos para realizar a comunicação entre os sistemas.

O novo mecanismo de navegação implementado por estes sistemas, permitirá a simulação em alto nível da navegação de robôs de acionamento diferencial, a um baixo custo computacional. Além disso, ele tornará possível o planejamento e a tomada de decisões de navegação, por parte do software externo de controle do robô, pois as novas implementações desacoplam do simulador estas lógicas. Dessa forma, as aplicações robóticas simuladas no HMR Sim poderão ser mais facilmente transportadas para robôs reais.

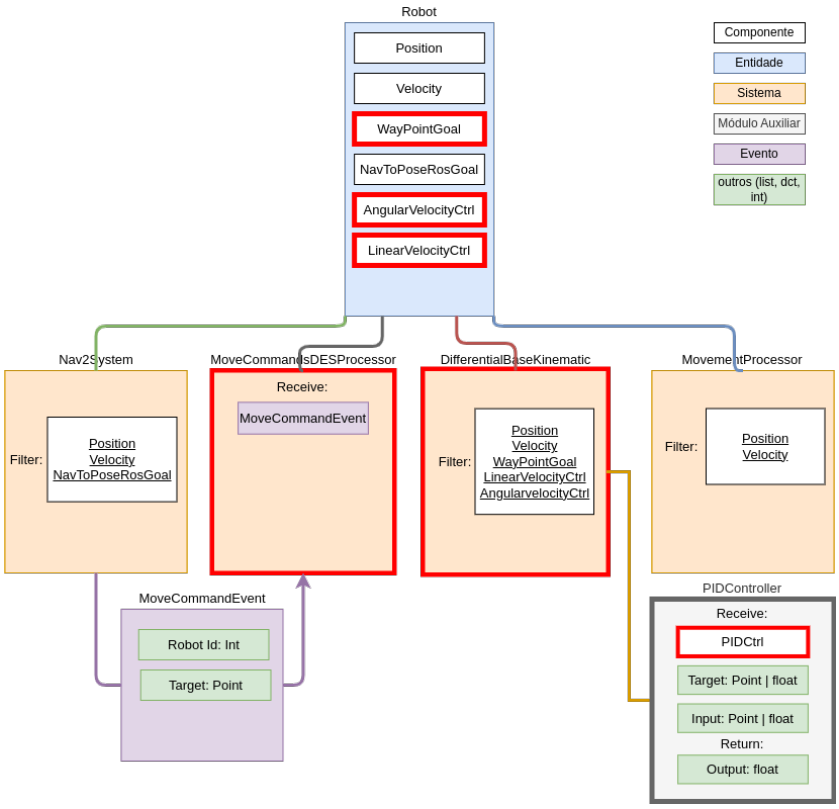

Figura 3.1: Diagrama ECS do novo mecanismo de navegação. Fonte: De autoria própria.

As principais melhorias trazidas pelo mecanismo de navegação implementado neste projeto em relação ao usado anteriormente pelo HMR Sim estão na movimentação do agente robótico simulado e na dinâmica de controle da sua navegação. Anteriormente a movimentação dos robôs eram omni-direcional, e os sistemas que a implementavam eram acoplados à lógicas de planejamento de rota que existiam no simulador. Dessa forma, tornava-se inviável a implementação de modelos cinemáticos de robôs mais comuns, como o diferencial ou o *Ackerman*. Além disso, também impossibilitava o software externo de controle do robô ou de coordenação de múltiplos robôs, de planejar a rota seguida pelo agente robótico, pois esta era determinada internamente por um algoritmo do simulador. Com o mecanismo de navegação desenvolvido neste projeto, a movimentação simulada é

a de robôs de acionamento diferencial, muito comuns em aplicações robóticas. Ademais, os software externos que operam sobre o agente robótico agora tem autonomia total sobre a rota percorrida pelo mesmo, sendo o simulador responsável apenas por deslocar o robô diretamente de um ponto A para um ponto B.

3.2 Desacoplando o Planejamento de Rota da Simulação de Movimentação do Robô

Como descrito na Subseção 2.1.2, os sistemas de comandos de navegação (*GotoDESProcessor*) e de caminhos (*PathProcessor*) eram acoplados a uma lógica de planejamento de rotas do simulador. Ambos os sistemas atuavam sobre um componente de rota (*Path*), determinado pelo sistema de navegação (*NavigationSystem*), contendo os pontos que deveriam ser percorridos . Este sistema de navegação era responsável pelo planejamento da rota a partir das informações de posição e destino do agente robótico comandado. Além disso, ele também utilizava informações de caminhos seguros e POIs obtidas a partir do componente de *Map* da entidade simulação, para determinar a rota a ser percorrida.

O controle de rotas realizado internamente pelo HMR Sim impossibilitava que o software externo de controle do robô planejasse e tomasse decisões sobre o percurso percorrido pelo agente robótico até o destino. Esta limitação era especialmente importante no que tange a coordenação de times de robôs, em que é necessário um maior controle das rotas percorridas por cada um dos agentes simulados. Além disso, devido a este acoplamento, a aplicação robótica simulada precisaria ser adaptada para ser transportada do ambiente simulado para o ambiente real ou vice-e-versa. Portanto, optou-se pela remoção do planejamento de rotas do simulador, para que o mesmo ocupe-se apenas de simular o ambiente e o agente robótico que será controlado.

Neste projeto, os três sistemas acoplados ao planejamento de rotas foram removidos do mecanismo de navegação dando lugar a novos sistemas que implementam a movimentação do robô na simulação, sem quaisquer considerações sobre caminhos que possam ser percorridos ou pontos de interesse no mapa. Torna-se de inteira responsabilidade do software externo que opera sobre os agentes robóticos simulados, o planejamento e decisões de navegação.

3.2.1 Sistema de Comandos de Navegação

Um novo sistema de comandos de navegação (*MoveCommandsDESProcessor*) foi desenvolvido para substituir o anterior *GotoDESProcessor* . Este sistema recebe a partir do sistema de controle de navegação ROS (*Nav2System*) descrito na Subseção 2.1.2, eventos

de comandos de movimentação (*MoveCommandEvent*) que fornecem uma identificação da entidade comandada e a posição de destino. A partir destes dados o sistema cria um novo componente de ponto de caminho (*WayPointGoal*) contendo as coordenadas de destino, e o adiciona à entidade para que o sistema responsável por calcular as velocidades necessárias para executar o movimento, possa atuar sobre ela.

Esta nova implementação do sistema de comandos de navegação não possui nenhuma lógica de planejamento de rotas e não interage com informações de caminhos seguros ou POIs extraídas do cenário de simulação. Ao contrário do sistema utilizado anteriormente que podia receber o destino como coordenadas no mapa ou tags identificadoras de POIs, este novo sistema apenas recebe coordenadas. Além disso, diferentemente do componente *Path* que era adicionado à entidade na implementação anterior, o componente *WayPointGoal* adicionado pelo o novo sistema, contém apenas um ponto de destino. Dessa forma, o sistema de comandos de navegação implementado neste projeto, apenas adiciona à entidade um ponto de destino para uma movimentação simples e direta. O planejamento dos pontos que devem ser percorridos para executar um percurso mais complexo, fica a cargo do software externo de controle do agente robótico.

3.3 Simulação da Movimentação de Robôs de Acionamento Diferencial

Os robôs de acionamento diferencial apresentados na Seção 2.4 são muito utilizados em aplicações robóticas no geral, devido a sua simplicidade e facilidade de controle. No entanto, a simulação deste tipo de robô no HMR Sim, levava a resultados muito distantes da realidade, pois o simulador suportava apenas o modelo omni-direcional e dessa forma a movimentação realizada pelo robô simulado pouco se assemelhava a de um robô de acionamento diferencial real. Além disso, do ponto de vista da coordenação de robôs, essa discrepância poderia levar ao desenvolvimento de algoritmos incorretos, visto que o agente robótico poderia percorrer trajetórias diferentes das disponíveis para ele.

Para mitigar estas limitações e permitir que o HMR Sim possa ser usado em uma gama maior de aplicações robóticas, foi desenvolvido neste projeto um sistema de controle de movimentação baseado no modelo cinemático diferencial (*DifferentialBaseKinematicsProcessor*). Este sistema tem uma função equivalente a do *PathProcessor* apresentado na Subseção 2.1.2, mas ao contrário do último, ele não contém nenhum acoplamento com a lógica de planejamento de rotas que existia no HMR Sim. Ele obtém a informação do ponto de destino a partir do componente *WayPointGoal* , inserido na entidade pelo novo sistema de comandos de navegação, e para determinar as velocidades que serão aplicadas no

robô para que ocorra o movimento, ele baseia-se nas equações matemáticas que definem o modelo cinemático diferencial.

O modelo cinemático diferencial, como apresentado na Seção 2.4, pode ser expresso em função das velocidades das rodas do robô, pela equação matricial (2.9). Esta representação é muito utilizada, por exemplo, no controle das velocidades de rotação de motores DC que movimentam as rodas do agente robótico. Entretanto, no HMR Sim o controle da movimentação é feito diretamente pelas velocidades linear e angular da entidade, pois como descrito na Seção 2.1, ele simula com um baixo nível de detalhamento físico para garantir simulações mais leves. Portanto, componentes como rodas e motores não são simulados diretamente, apenas as capacidades que eles provem ao agente robótico. Dessa forma, para preservar a proposta inicial do HMR Sim, optou-se por utilizar neste sistema, a representação do modelo cinemático diferencial mostrado em (2.9), em que a posição e orientação do robô são dadas em termos das velocidades linear (v) e angular (ω). O controle por estes parâmetros é mais intuitivo e também é comumente aceito pelos drivers de robôs de acionamento diferencial. A Figura 3.2 ilustra a movimentação de um robô diferencial no plano em termos de v e ω.

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} cos\theta & 0 \\ sen\theta & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} v \\ \omega \end{bmatrix} \tag{3.1}$$

As equações (3.2), (3.3) e (3.4) deduzidas a partir de (3.1) mostram como a posição e orientação do robô no tempo, podem ser obtidas a partir das velocidades fornecidas pelo sistema de controle de movimentação desenvolvido.

$$\dot{x}(t) = v(t)cos\theta(t) \tag{3.2}$$

$$\dot{y}(t) = v(t)sen\theta(t) \tag{3.3}$$

$$\dot{\theta}(t) = \omega(t) \tag{3.4}$$

Vale ressaltar, que a alteração da posição e orientação do robô, continua a cargo do sistema de movimentação (*MovemmentProcessor*) apresentado na Subseção 2.1.2. O novo sistema de controle de movimentação, controla apenas as velocidades que atuam sobre o movimento, atualizando-as no componente correspondente como era feito anteriormente pelo *PathProcessor*.

Além de desacoplar lógicas de rota e aplicar o modelo cinemático diferencial, o sistema desenvolvido também adotou uma nova estratégia para o controle das velocidades

da entidade. Como apresentado na Subseção 2.1.2, o *PathProcessor* obtinha a velocidade linear baseada apenas nas posições de origem e destino do robô, sem qualquer dependência da orientação ou controle sobre a velocidade angular do mesmo. Dessa forma, o agente robótico não mudava a sua orientação e realizava apenas movimentos em linha reta na vertical, horizontal e diagonal. No entanto, o novo sistema determina as velocidades linear e angular, não só com base nas posições do robô simulado, mas também em sua orientação. Portanto, para concluir o desenvolvimento deste sistema, identificou-se a necessidade de utilizar uma nova estratégia de controle das velocidades e dentre as disponíveis na literatura, optou-se pelo o desenvolvimento de um controlador PID.

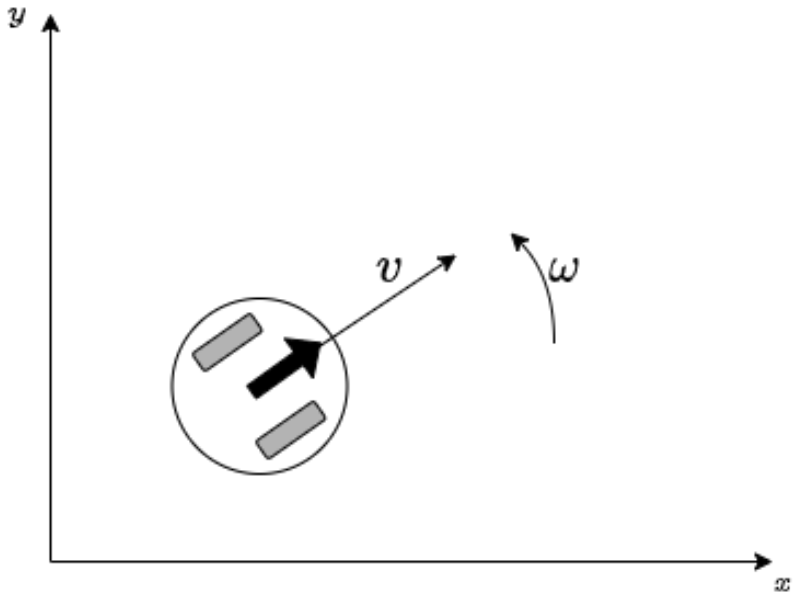

Figura 3.2: Movimentação de um robô diferencial em termos de v e ω. Fonte: De autoria própria.

3.4 Desenvolvimento de um Controlador PID

O controlador PID apresentado na Seção 2.5 é um algoritmo de controle muito presente na literatura e caracterizado pela sua versatilidade e facilidade de implementação. Ele é frequentemente utilizado no controle das velocidades de rotação de motores DC em robôs diferenciais, a fim de manter as velocidades linear e angular do robô estáveis ao longo do tempo [13]. Mas também pode ser utilizado como controlador para seguimento de caminho (*path following*), ajustando as velocidades para reduzir o erro entre a pose atual e a pose de destino ao longo do caminho. Este uso do PID não é muito comum na literatura, visto que existem outros algoritmos de controle mais eficientes para está tarefa como o de

Linearização por Realimentação Dinâmica (DFL – *Dynamic Feedback Linearization*) [14] e o de Controle por Coordenadas Polares (POL – *Polar coordinate control*) [15]. Contudo, foi decidido pela sua utilização, pois ele atende bem ao problema de controle de movimentação apresentado na seção anterior, e possui um grande potencial de uso futuro em outros sistemas que venham a ser desenvolvidos no HMR Sim.

3.4.1 Controle de Robôs de Acionamento Diferencial

Para aplicar o controlador PID na movimentação diferencial, primeiro verificou-se que o algoritmo deveria controlar as velocidades linear e angular separadamente. Segundo, determinou-se que a variável de referência do controlador deveria ser a pose de destino da entidade, e a variável de processo, que fornece o *feedback* ao loop, deveria ser a pose atual do agente robótico. Portanto, para o controle da velocidade angular, as entradas do controlador seriam a orientação atual e a de destino. Enquanto que para o controle da velocidade linear, seriam a posição atual e a posição de destino. Na Figura 3.3 são ilustradas as posições e orientações, atual e de destino de um robô diferencial. Nela x_r, y_r e θ_r são a posição e orientação do robô e x_g, y_g e θ_g são as desejadas ao final do movimento.

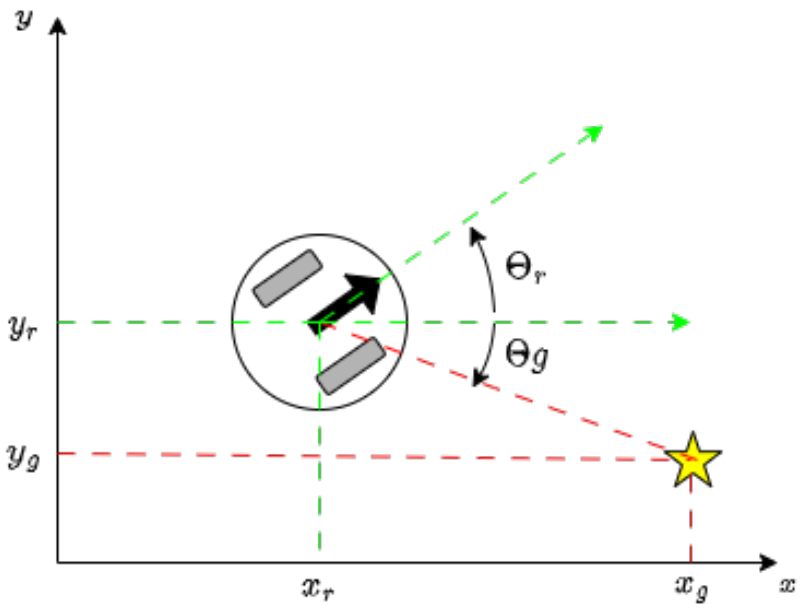

Figura 3.3: Posições e orientações de um robô diferencial e seu objetivo no plano. Fonte: De autoria própria.

Além disso, também foi importante determinar como seria calculado o erro utilizado pelo controlador para ajustar as velocidades, visto que os termos proporcional, integral e derivativo são calculados a partir dele, como apresentado na Seção 2.5.

No controle da velocidade angular utilizou-se a seguinte expressão para determinar o erro entre θ_g e θ_r:

$$erro = \Theta_g \ominus \Theta_r \tag{3.5}$$

Em que o simbolo $\ominus$ indica a menor diferença entre os dois ângulos, ou seja, o menor deslocamento angular, no sentido horário ou no anti-horário, para sair de θ_r e chegar a θ_g.

Para o controle da velocidade linear, utilizou-se a distância euclidiana entre dois pontos, a posição atual do robô e a do seu objetivo:

$$erro = \sqrt{(y_g - y_r)^2 + (x_g - x_r)^2} \tag{3.6}$$

Dessa forma, aplicando o controlador PID como descrito nesta seção e baseando-se no modelo cinemático diferencial implementado pelo sistema de controle de movimento, descrito na Seção 3.3, o simulador poderá executar movimentações como as ilustradas na Figura 3.4.

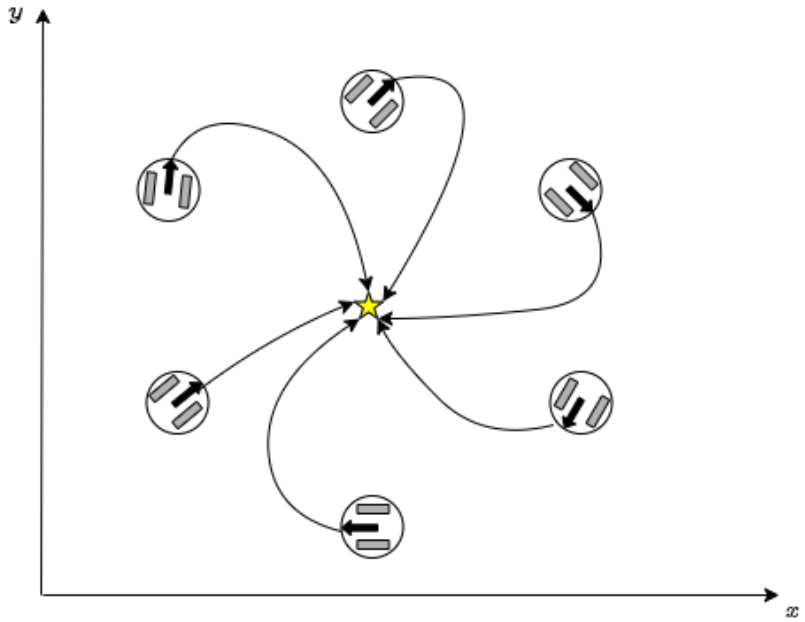

Figura 3.4: Exemplos de trajetórias que podem ser percorridas por um robô diferencial no plano. Fonte: De autoria própria.

3.4.2 Implementação do Controlador PID na Arquitetura ECS

Como descrito na Subseção 2.1.1, o HMR Sim é implementado na arquitetura ECS, em que toda a estrutura do simulador é modelada em termos de entidades, componentes e sistemas, garantindo-lhe modularidade e extensibilidade. Portanto, considerou-se importante que o controlador PID fosse implementado de forma que respeitasse esta estrutura. Além disso, como descrito anteriormente, observou-se um grande potencial de uso deste tipo de controlador em outras partes do simulador. Logo, deu-se uma maior importância na sua generalidade para que ele possa ser mais facilmente reutilizado.

O primeiro grande desafio na sua implementação, foi o de armazenar os dados utilizados em suas iterações, como erro anterior e erro acumulado, em cada entidade controlada. A solução natural na arquitetura ECS é o uso de componentes, mas este componente precisaria permitir que o mesmo algoritmo de controle pudesse operar com tipos diferentes de parâmetros como por exemplo, orientações e coordenadas de posição no plano. Ademas, uma mesma entidade poderia ter diferentes instancias deste componente, cada uma para uma variável de controle diferente. Por exemplo, um robô diferencial simulado teria suas velocidades angular e linear controladas separadamente, portanto necessitaria de dois componentes que armazenassem os dados de iteração do controlador.

Outro desafio, foi o de como modularizar o controlador PID de forma que ele respeitasse a arquitetura ECS e pudesse ser usado em conjunto com outros sistemas. Simplesmente implementá-lo como um novo sistema, poderia prejudicar a sua performance e a sua modularidade, pois ele interagiria com outros sistemas por eventos ou precisaria ser acoplado à lógica dos mesmos.

3.4.3 Componentes de Controle PID

A solução para o primeiro desafio, foi a utilização de herança de classes e o desenvolvimento de três classes de componentes. A primeira classe desenvolvida, foi a *PIDControl* que implementa a interface *Component*, utilizada na arquitetura do simulador, e possui todos os atributos necessários para o uso do controlador PID. Apenas com este componente, já seria possível aplicar o PID em uma ampla variedade de problemas de controle. Contudo, para a aplicação dele neste projeto, foi necessário o desenvolvimento de mais dois componentes que herdam de *PIDControl*. O uso de herança neste caso, permite com que estes componentes possam ser utilizados na mesma implementação do controlador que recebe a classe pai como parâmetro, e que possam coexistir na mesma entidade.

A segunda classe de componente desenvolvida foi o *AngularVelocityCtrl* , que herda de *PIDControl* e armazena os dados de controle necessários para a atuação do PID na velocidade angular da entidade. O desenvolvimento deste componente foi necessário,

não apenas para separar as informações de controle das velocidades angular e linear, mas também para definir a função de cálculo de erro que será utilizada pelo controlador. Como visto na Subseção 3.4.1, o erro utilizado no controle da velocidade angular é obtido pela equação (3.5). Portanto, este novo componente sobrescreve o atributo que armazena uma referência para a função que realiza o cálculo do erro, atribuindo-lhe uma nova referência para uma função que implementa a equação (3.5).

Por fim, foi desenvolvido uma terceira classe de componente, o *LinearVelocityCtrl*, que também herda de *PIDControl*, e armazena os dados de controle da velocidade linear da entidade. Assim como a *AngularVelocityCtrl*, está classe também precisou ser implementada para distinguir os dados de controle das velocidades angular e linear, e sobrescrever o atributo de referência para a função de cálculo do erro. Como o controle da velocidade linear é realizado com base em posições no plano cartesiano, o cálculo do erro utiliza a distância Euclidiana como mostrado na equação (3.6), portanto uma função que realiza este cálculo foi atribuída ao componente.

Vale ressaltar, que a presença de uma referência para uma função dentro de um componente, fere o princípio da arquitetura ECS de separar a lógica dos dados. No entanto, optou-se por realizar está pequena violação, pois ela foi fundamental para que a mesma implementação do controlador PID, pudesse ser usada com diferentes parâmetros de entrada, como ângulos e coordenadas cartesianas.

A Figura 3.5 mostra um diagrama de classes dos componentes implementados. Nela é ilustrada a relação de herança entre eles e os atributos herdados e sobrescritos.

Abaixo seguem uma descrição de cada um dos atributos presentes nos componentes e de seus respectivos papéis no controlador PID implementado.

Kp* ,*Kd* e *Ki Armazenam os ganhos aplicados em cada termo do controlador.

output_limits São os limites superior e inferior que são utilizados para evitar a saturação da saída do controlador. No controle da velocidade linear, seria a velocidade máxima que o robô pode alcançar.

diff Guarda uma referência para uma função que será utilizada pelo controlador para calcular o erro.

integrator É um acumulador de erros utilizado para o cálculo aproximado do termo integral do controlador.

integral*, *proporcional* e *derivative Armazenam os valores calculados de cada um dos termos do controlador.

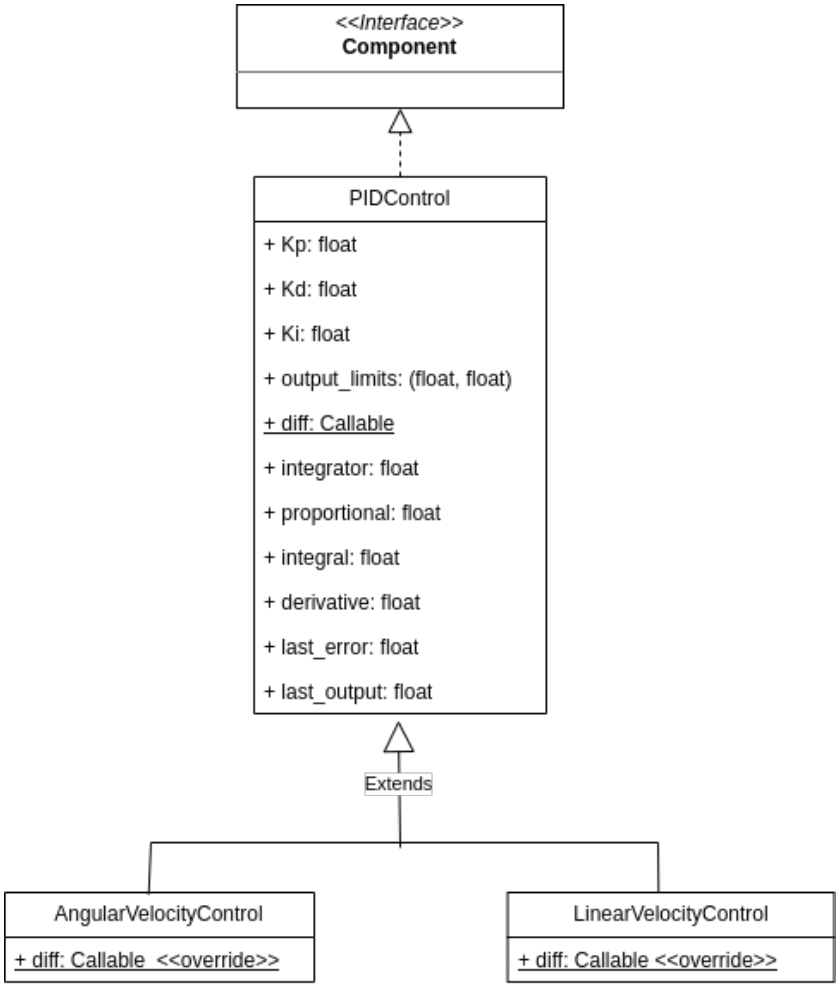

Figura 3.5: Diagrama de classes dos componentes de controle PID. Fonte: De autoria própria.

last_error* e *last_output Guardam o erro e a saída de controle da última iteração do controlador. Este erro é utilizado para o cálculo aproximado do termo derivativo na próxima iteração do controlador.

Os parâmetros de ganho e os limites de saída do controlador são passados na instanciação dos componentes de controle das velocidades. Assim como os demais componentes do simulador, eles podem ser inseridos nas entidades definidas no arquivo do mapa da simulação ou no arquivo de configuração da simulação [1].

3.4.4 Módulo de Controle PID

O desafio de modularização do controlador PID na arquitetura ECS foi solucionando implementando-o como um módulo a parte que pode compor os sistemas implementados no HMR Sim. Dessa forma, ele pode ser facilmente reaproveitado entre os sistemas e a sua comunicação com eles pode ser feita através da chamada do seu método principal. Além

disso, ele respeita a arquitetura do simulador, pois, de forma semelhante aos sistemas, ele apenas contém a lógica de controle, deixando os dados manipulados armazenados nos componentes apresentado na seção anterior.

Utilizando esta estratégia o controlador desenvolvido pode ser incorporado ao sistema *DifferentialBaseKinematics* para enfim concluir a implementação da simulação de movimentação de robôs de acionamento diferencial, descrita na seção 3.3. O diagrama de classes da Figura 3.6 ilustra a relação de composição entre o sistema de controle de movimentação e o controlador PID.

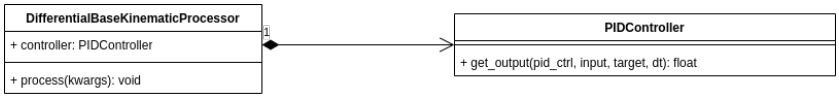

Figura 3.6: Diagrama de classes do sistema de controle de movimentação e o controlador PID. Fonte: De autoria própria.

A Listagem 3.1 mostra o código da classe *PIDController* que implementa o controlador PID . Nela observa-se que o controlador foi implementado no formato de uma classe contendo um único método público chamado *get_output*, correspondente a uma iteração do loop de controle. Este método recebe como parâmetros um componente da classe *PIDControl* (*ctrl*), a entrada de referência do controlador (*target*), a variável de processo (*input*) e o tempo decorrido entre as iterações (*dt*).

```
def _clamp(value, limits):
    lower, upper = limits
    if value is None:
        return None
    elif (upper is not None) and (value > upper):
        return upper
    elif (lower is not None) and (value < lower):
        return lower
    return value

class PIDController():
    """An simple adaptation of the PID controller for the ECS architecture"""
    def __init__(self) -> None:
        self.logger = logging.getLogger(__name__)

    def get_output(self, ctrl: PIDControl, input_: Union[float, Point], target: Union[
    float, Point], dt: float) -> float:

        # Compute error terms using component specif diff function
        error = ctrl.diff(input_, target)
        d_error = error - (ctrl.last_error if (ctrl.last_error is not None) else error)
        ctrl.integrator = ctrl.integrator + error * dt

        # Compute the proportional term
        ctrl.proportional = ctrl.Kp * error

        # Compute the integral term
```

```
27          ctrl.integral = ctrl.Ki * ctrl.integrator
28          ctrl.integral = _clamp(ctrl.integral, ctrl.output_limits)
29
30          # Compute the derivative term
31          ctrl.derivative = ctrl.Kd * d_error / dt
32
33          output = ctrl.proportional + ctrl.integral + ctrl.derivative
34          output = _clamp(output, ctrl.output_limits)
35
36          # Keep track of state on ctrl component
37          ctrl.last_error = error
38          ctrl.last_output = output
39
40          return output
```

Listagem 3.1: Módulo de Controle PID. Fonte: De autoria própria.

O método *get_output* começa sua execução pelo cálculo do erro *error* a partir das entradas *target* e *input* e utiliza a função fornecida pelo componente de controle *ctrl*. Ele também calcula a diferença entre os erros atual e o da última iteração *d_error* e acumula o *error* no atributo *integrator* do componente de controle *ctrl*.

Em seguida, são executados os cálculos dos termos do controlador começando pelo proporcional expresso pela equação (2.10), que é executado simplesmente pelo produto do ganho *Kp* fornecido pelo componente *ctrl* , pelo o *error* calculado anteriormente. O termo integral que pode ser aproximado por (2.14) , é obtido pelo produto do ganho *Ki* do *ctrl* pelo *integrator* e passa por uma função *clamp* que aplica os *output_limits* também fornecidos pelo *ctrl*, para limitar o seu valor à uma faixa aceitável pelo processo controlado.O termo derivativo é calculado de forma aproximada de acordo com (2.12), realizando o produto do ganho Ki dado pelo *ctrl*, pela divisão do *d_error*, calculado anteriormente, pelo *dt*.

Por fim, a saída de controle *output* é obtida a partir da soma dos termos calculados e é também passada por uma função de *clamp* para evitar valores inválidos para o processo controlado. O erro e a saída calculados são guardados em *ctrl* nos atributos *last_error* e *last_output*.

Capítulo 4

Experimentação

Neste capítulo o novo mecanismo de navegação será exercitado em dois cenários de simulação no HMR Sim. O objetivo é validar os novos sistemas implementados e o controlador PID. Tal validação será feita através da execução de alguns percursos por um agente robótico simulado, em que será verificada a correta execução dos comandos de navegação e a simulação de navegação diferencial. Por fim, será verificado o desempenho do simulador com as novas implementações para avaliar o impacto das mesmas.

No primeiro cenário, apresentado na Seção 4.1, será mostrado um robô percorrendo uma trajetória simples até um ponto de destino, através de alguns comandos de navegação utilizando o ROS. A trajetória executada pelo agente robótico será comparada com a obtida atualmente no HMR Sim no mesmo cenário de simulação . Também será discutido as novas dinâmicas de navegação para que o robô execute este percurso.

O segundo cenário, apresentado na Seção 4.2, simulará a execução de um percurso mais complexo para exercitar o comportamento diferencial da navegação. Um software utilizando o conceito de *Behaviour Tree* apresentado na Seção 2.3, será utilizado para controlar a navegação do robô através de uma trajetória de lemniscata.

Em ambos os cenários de simulação o robô estará configurado com uma velocidade linear máxima de 10 m/s e uma velocidade angular máxima de $\pi/8\ rad/s$. Os ganhos do controlador PID utilizados serão de $K_P = 1, K_D = 0,\ e\ K_I = 0$, para o controle da velocidade linear, e $K_P = 0.5, K_D = 0,\ e\ K_I = 0$, para o controle da velocidade angular.

Os parâmetros do PID foram determinados de forma empírica para alcançar os melhores resultados nos cenários propostos. Primeiramente foi determinado experimentalmente o valor do termo proporcional e em seguida foi averiguado a necessidade dos termos integral e derivativo. Verificou-se que apenas com o termo proporcional do controlador, o simulador já obtém resultados satisfatórios na simulação de navegação diferencial. Isto ocorre pois o controlador atua diretamente sobre o valor das velocidades, portanto se a velocidade aplicada for um pouco menor ou a entidade passar um pouco da posição de

destino os resultados não serão prejudicados. Contudo, em simulações que exijam uma maior precisão, o uso dos outros termos do controlador poderá ser necessário.

Os ganhos do controlador também podem ser obtidos através de testes em campo utilizando o mesmo algoritmo de controle, mas aplicado em um robô real. Dessa forma, pode-se obter parâmetros do controlador PID que reduzam o gap entre a simulação e o agente robótico real.

As simulações apresentadas neste capítulo foram executadas em uma ambiente com as seguintes configurações: sistema operacional *Linux Ubuntu 20.04*, processador Intel i5-8250U 3.400GHz e 8GB de memória RAM. A versão do Python utilizada foi a 3.8.10 e a versão do ROS foi a foxy.

4.1 Cenário 1 - Navegação Diferencial com Comandos mais Simples

O objetivo neste cenário é verificar se o agente robótico simulado executa corretamente os comandos de navegação enviados sem a lógica de rotas existente anteriormente no simulador.

O mapa utilizado nesta simulação pode ser visto na Figura 4.1. Nela o robô de nome *R2D2* é ilustrado por um quadrado azul no canto superior esquerdo. Os pontos coloridos indicados na imagem são apenas referências visuais das coordenadas utilizadas na navegação. O destino final da trajetória do robô é indicado pelo ponto em preto e os pontos em vermelho definem a rota que será percorrida para alcançar este objetivo. Neste mapa não existem mais os *map-path* para definir os caminhos seguros que o agente robótico poderá percorrer. Como discutido na Seção 3.2, os *map-path* faziam parte da lógica de planejamento de rotas que existia no HMR Sim e que foi removida no novo mecanismo de navegação implementado.

A Figura 4.2 mostra o mesmo mapa de simulação, mas com a presença dos *map-paths*. Nela, observa-se que para a execução de uma simulação semelhante a desta seção, era necessário a definição dos caminhos seguros no mapa da simulação. A partir deles o simulador escolhia a rota até o destino e dessa forma, apenas um comando de navegação era suficiente, pois os demais pontos percorridos na trajetória eram definidos pelo próprio simulador.

As Figuras Listagem 4.1, Listagem 4.2, Listagem 4.3 e Listagem 4.4 mostram a sequencia de comandos de navegação enviados. Neles são indicados o nome do robô a ser comandado e as coordenadas de destino.

Figura 4.1: Mapa utilizado no primeiro cenário de simulação. Fonte: De autoria própria.

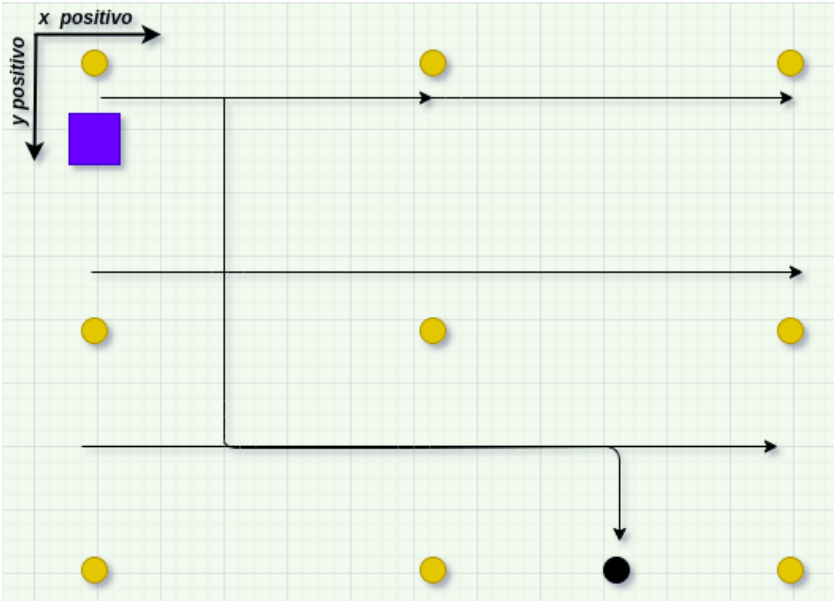

Figura 4.2: Mapa utilizado anteriormente pelo HMR Sim. Fonte: De autoria própria.

```
ros2 action send_goal navigate_to_pose/R2D2 nav2_msgs/action/NavigateToPose "{pose: {
    pose: { position: {x: 140, y: 85, z: 1} } } }" --feedback
```

Listagem 4.1: Primeiro comando de navegação enviado ao robô.Fonte: De autoria própria.

```
ros2 action send_goal navigate_to_pose/R2D2 nav2_msgs/action/NavigateToPose "{pose: {
    pose: { position: {x: 175, y: 235, z: 1} } } }" --feedback
```

Listagem 4.2: Primeiro comando de navegação enviado ao robô.Fonte: De autoria própria.

```
ros2 action send_goal navigate_to_pose/R2D2 nav2_msgs/action/NavigateToPose "{pose: {
    pose: { position: {x: 365, y: 270, z: 1} } } }" --feedback
```

Listagem 4.3: Primeiro comando de navegação enviado ao robô.Fonte: De autoria própria.

```
ros2 action send_goal navigate_to_pose/R2D2 nav2_msgs/action/NavigateToPose "{pose: {
    pose: { position: {x: 400, y: 350, z: 1} } } }" --feedback
```

Listagem 4.4: Primeiro comando de navegação enviado ao robô.Fonte: De autoria própria.

Na Figura 4.3 é mostrado o terminal com os logs do HMR Sim após a execução da simulação. Observe que os quatro comandos de navegação enviados foram recebidos corretamente pelo sistema *MoveCommandsDESProcessor*, apresentado na seção 3.2.1, e o robô alcançou os pontos desejados.

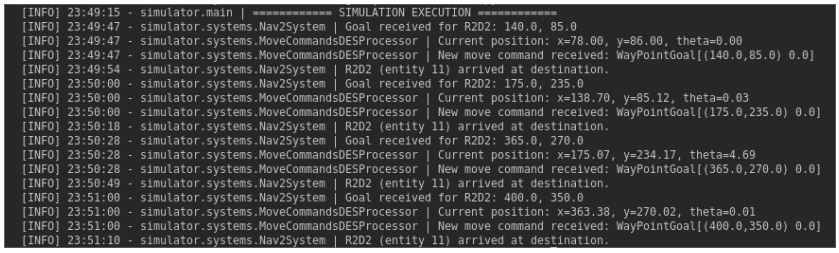

Figura 4.3: Terminal com os logs do simulador após executar o cenário 1. Fonte: De autoria própria.

A Figura 4.4 mostra a trajetória percorrida pelo agente robótico após a execução da simulação. Nela verifica-se que o robô executa corretamente os comandos de navegação e alcança o destino final. Também verifica-se a movimentação diferencial do agente robótico, que corrige sua orientação executando uma trajetória curvilinear através da rota percorrida.

Para efeitos de comparação, a Figura 4.5 ilustra a trajetória executada por um agente robótico no mesmo mapa simulado no atual mecanismo de navegação do HMR Sim. O deslocamento do robô ocorre sem alterar a sua orientação e executa apenas movimentos retos na diagonal, vertical e horizontal.

A partir dos resultados apresentados, conclui-se que o HMR Sim agora é capaz de simular a navegação diferencial através de comandos de navegação que executam trajetórias simples, deixando a cargo do software de controle do robô o planejamento da rota completa que será percorrida.

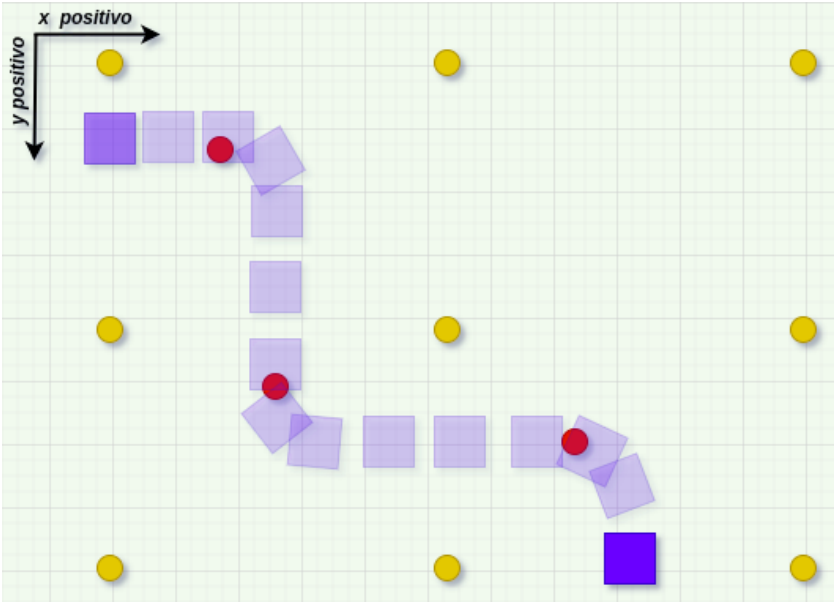

Figura 4.4: Trajetória percorrida pelo robô ao final da simulação do cenário 1. Fonte: De autoria própria.

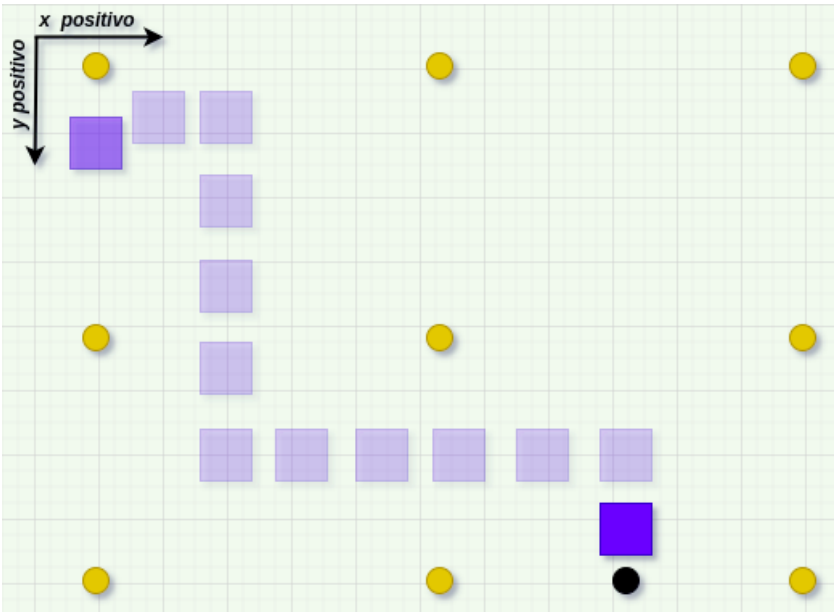

Figura 4.5: Trajetória percorrida pelo robô com o atual mecanismo de navegação. Fonte: De autoria própria.

4.2 Cenário 2 - Navegação Diferencial com *Behaviour Tree*

Neste cenário será verificado com mais ênfase a simulação de navegação diferencial. Para isso, uma *Behaviour Tree* será utilizada para controlar a navegação do robô simulado,

através de uma trajetória de lemniscata. Dessa forma, será possível exercitar melhor as capacidades do novo mecanismo de navegação de executar trajetórias curvilíneas de forma semelhante a de robôs de acionamento diferencial reais.

A Figura 4.8 mostra o mapa utilizado neste cenário de simulação. O robô aparece novamente ilustrado por um quadrado de cor roxa e os pontos em vermelho são representações visuais das coordenadas que serão utilizadas para que o agente robótico execute a trajetória desejada. O final do percurso é representado por um ponto na cor preta.

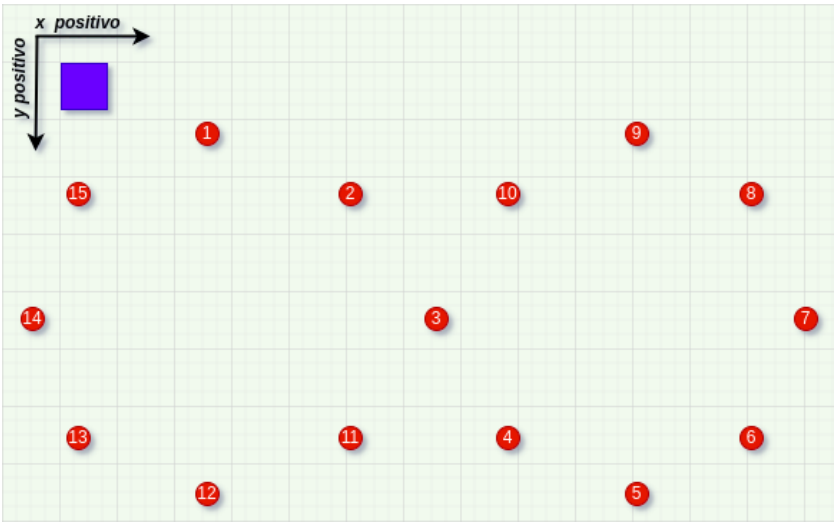

Figura 4.6: Mapa utilizado no segundo cenário de simulação. Fonte: De autoria própria.

O objetivo da simulação é bastante simples. O robô deve partir da coordenada $x = 62, y = 70$ e alcançar o seu objetivo final na coordenada $x = 45, y = 124$, seguindo os pontos que compõem a rota em forma de lemniscata.

O controle da navegação será feito pela *Behaviour Tree* mostrada na Figura 4.7. Ela é composta por um *decorator* de repetição (*repeat*) como nó raiz (*root*), que repetirá a execução de sua sub-arvore filha enquanto não receber n retornos de sucesso ou uma falha. O número n representa a quantidade de vezes que será executado o comando de navegação e corresponde ao tamanho da fila de pontos de caminho ($WayPoints$) que compõem a rota. O *decorator* tem como nó filho uma *sequence* que executa em sequencia os nós folhas da árvore até que ambos retornem sucesso ou um deles retorne falha. O primeiro destes nós é o *GetWayPoint* que pega um dos *WayPoints* da fila e escreve na forma de um comando de navegação na *blackboard*. O segundo é o *GoToWayPoint* que lê o comando de navegação da *blackboard* e o envia para o simulador através de uma *Action* ROS. O código da função que constrói está *Behaviour Tree* é mostrada na Listagem 4.5.

```
1 def create_root() -> py_trees.behaviour.Behaviour:
2
3     way_points = [(135.0, 82.0), (235.0, 124.0), (295.0, 211.0), (345.0, 294.0),
4                   (435.0, 333.0), (515.0, 294.0), (553.0, 211.0), (515.0, 124.0),
```

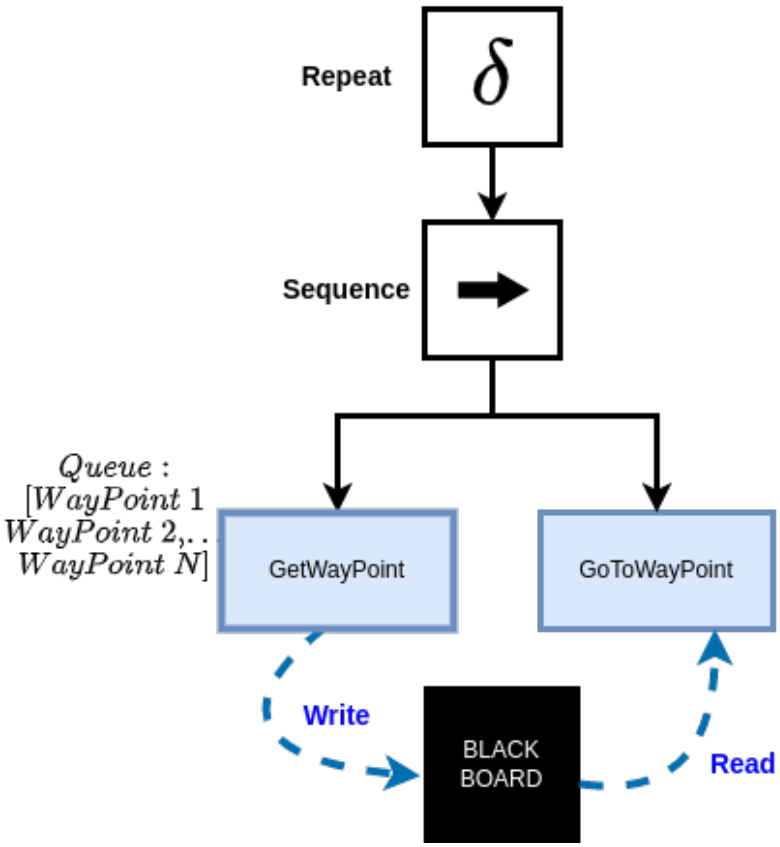

Figura 4.7: *Behaviour Tree* utilizada no controle do robô no cenário 2. Fonte: De autoria própria.

```
                (435.0, 82.0), (345.0, 124.0), (235.0, 294.0), (135.0, 333.0),
                (45.0, 294.0), (13.0, 211.0), (45.0, 124.0)
            ]
    sequence = py_trees.composites.Sequence("GoToWayPointSequence")
    get_waypoint = GetWayPoint(way_points=way_points, name="GetWayPoint")
    robot_name = "R2D2"
    action_client = py_trees_ros.action_clients.FromBlackboard(
        action_type=NavigateToPose,
        action_name="navigate_to_pose/" + robot_name,
        name="GoToWayPoint",
        key="waypoint",
        generate_feedback_message=lambda msg: "remaining: {0}".format(msg.feedback.
    distance_remaining)
    )
    sequence.add_children([get_waypoint, action_client])
    root = Repeat(
        name="GoToWayPointRepeatDecorator",
        child=sequence,
        num_success=len(way_points)
    )
```

```
24  return root
```

Listagem 4.5: Método de criação da *behaviour tree* a ser utilizada neste cenário. Fonte: De autoria própria.

Optou-se por utilizar *Behaviour Trees* nesta simulação, pois a trajetória que será percorrida pelo agente robótico é um pouco mais complexa. Além disso, o controle da rota não é mais feito internamente pelo simulador, logo será necessário que o software de controle de robô consiga enviar um conjunto maior de comandos de navegação de forma eficiente.

Na Figura 4.8 são mostrados os logs de execução da *Behaviour Tree*. Nela as quatro primeiras imagens corresponde a execução da *sequence* e seus nós folha que se repete para os quinze *WayPoints* que compõem a trajetória. Na quinta imagem é mostrado o retorno do nó *root*, ao final das repetições.

Figura 4.8: Terminal com os logs da execução da *Behaviour Tree* após a simulação. Fonte: De autoria própria.

Por fim, na Figura 4.9 é mostrado a trajetória executada pelo robô ao final da simulação. Ele percorre todos os pontos ajustando corretamente a sua orientação entre cada um deles.

4.3 Avaliação de Desempenho

Nesta seção será avaliado o impacto das novas implementações no desempenho do HMR Sim. Para tal, serão medidos os tempos de execução de iterações do loop principal de simulação, para o cenário apresentado na Seção 4.1, executado na versão atual do

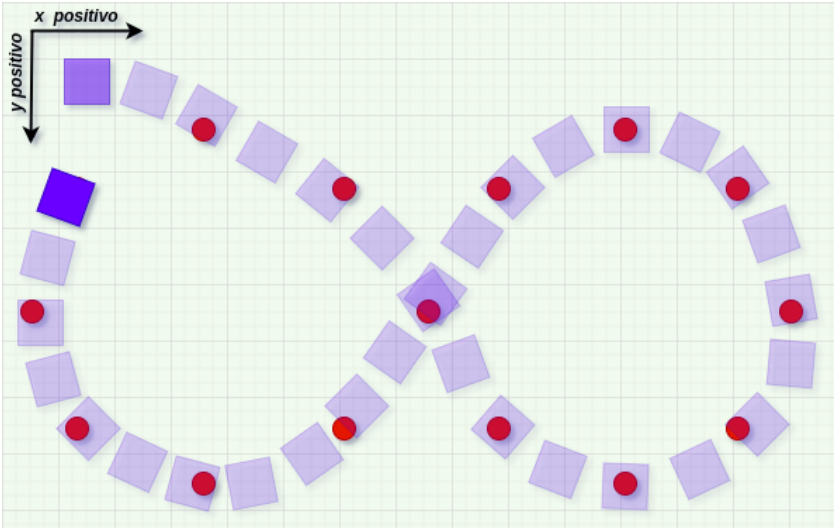

Figura 4.9: Trajetória percorrida pelo robô ao final da execução do cenário 2. Fonte: De autoria própria.

simulador e na versão com o novo mecanismo de navegação. O mecanismo de visualização e os demais sistemas do HMR Sim que não fazem parte da navegação serão desabilitados.

A Tabela 4.1 mostra uma amostragem dos tempos medidos para iterações do simulador atual e a Tabela 4.2 mostra os obtidos para o simulador utilizando o novo mecanismo de navegação.

O tempo médio de uma iteração do simulador utilizando o novo mecanismo de navegação é de $\approx 1\ ms$. Enquanto que no estado atual do simulador a execução de uma iteração leva em média $\approx 0,6\ ms$. Portanto, as novas implementações causaram um aumento de 66% no tempo de execução da navegação no simulador. Este impacto se deve principalmente pelo módulo de controle PID, visto que os demais sistemas desenvolvidos substituem outros utilizados anteriormente, removendo lógicas de planejamento de rota e, portanto, tornado a execução mais leve. Contudo, em valores absolutos este tempo de execução ainda é muito baixo e mantém o HMR Sim com uma boa performance.

Tempo inicial (s)	**Tempo final (s)**	**Tempo decorrido (s)**
1689483937,837579	1689483937,837983	0,000404
1689483937,838323	1689483937,838913	0,000590
1689483937,839529	1689483937,840345	0,00082
1689483937,840828	1689483937,84121	0,000382
1689483937,619165	1689483937,619625	0,000460
1689483937,620255	1689483937,621163	0,000908
1689483937,724964	1689483937,725516	0,000552
Média (s)		$0,000588 \pm 0,000189 (\pm 32.15\%)$

Tabela 4.1: Execuções realizadas com o mecanismo de navegação anterior. Margem de erro baseada em desvio padrão utilizando nível de confiança de um desvio padrão.

Tempo inicial (s)	**Tempo final (s)**	**Tempo decorrido (s)**
1689483835,906795	1689483835,907381	0,000586
1689483835,90789	1689483835,908708	0,000818
1689483836,012447	1689483836,013403	0,000956
1689483836,579623	1689483836,580597	0,000974
1689483836,581367	1689483836,582561	0,001194
1689483837,141438	1689483837,142521	0,001083
1689483837,3621	1689483837,363328	0,001228
Média (s)		$0,000977 \pm 0,000207 (\pm 21.18\%)$

Tabela 4.2: Execuções realizadas utilizando o novo mecanismo de navegação. Margem de erro baseada em desvio padrão utilizando nível de confiança de um desvio padrão.

Capítulo 5

Conclusão e Trabalhos Futuros

O principal objetivo deste trabalho é implementar no HMR Sim uma simulação leve da navegação de robôs de acionamento diferencial, para que dessa forma o simulador possa ser utilizado na simulação de um maior número de aplicações robóticas e de forma mais eficiente.

Este objetivo é alcançado através do desenvolvimento de um novo mecanismo de navegação que substitui alguns dos sistemas utilizados anteriormente no simulador. Os novos sistemas implementados simulam a movimentação de robôs diferenciais e desacoplam do HMR Sim lógicas de navegação que fogem de suas competências como simulador.

A simulação leve de robôs de acionamento diferencial é alcançada principalmente pelo sistema *DifferentialBaseKinematics*, apresentado na Seção 3.3. Ele aplica o modelo cinemático diferencial nos robôs simulados, utilizando um controlador PID implementado pela classe *PIDController*. Este sistema consegue simular de forma satisfatória a movimentação deste tipo de agente robótico e sem comprometer significativamente o desempenho do simulador. Além disso, a classe *PIDController* desenvolvida neste projeto poderá ser facilmente reutilizada no desenvolvimento de outros sistemas que exijam algum tipo de controle à nível de simulador.

O sistema *MoveCommandsDESProcessor* descrito na Subseção 3.2.1, substitui no HMR Sim outros sistemas que implementam lógicas de determinação de rotas que tiravam da aplicação robótica simulada, a autonomia sobre o planejamento e a tomada de decisões de navegação. Este novo sistema recebe apenas comandos simples de movimentação, deixando o controle da trajetória completa para o software externo de controle do agente robótico.

Além das implementações descritas, também foram feitas melhorias no mecanismo de visualização das simulações no HMR Sim que permitiram a sua execução em ambiente local e em container docker.

Para trabalhos futuros, algumas possíveis implementações para o HMR Sim são:

- Implementar outros modelos cinemáticos de movimentação de robôs como o *Ackerman*
- Testar a navegação em cenários de simulação mais complexos, com múltiplos agentes robóticos.
- Implementar sistema na interface ROS para publicação de dados de navegação que possam ser utilizados pelo software de controle do robô para planejamento de rotas
- Implementar um novo mecanismo de visualização da simulação, mais fácil de executar e com mais opções de controle da simulação através da interface.
- Realizar estudo comparativo da execução de aplicações robóticas no HMR Sim e em robôs reais.

Alguns dos objetivos listados serão mais facilmente alcançados com as implementações deste projeto. Agora o HMR Sim é capaz de simular a navegação de robôs de acionamento diferencial e dar mais autonomia sobre o planejamento de rota para softwares externos de controle dos robôs simulados.

Referências

[1] Guidini, Giovanni e Cristiane Cardoso: *Desenvolvimento e teste da ferramenta hmr sim: Um simulador de ambientes multi-robôs*. Universidade de Brasília, 1(1), 2021. (acessado em 14/02/2023). xi, 1, 4, 7, 27

[2] Open Robotics: *Gazebo*, 2023. `http://gazebosim.org/`, (acessado em 27/01/2022). 1

[3] LAAS-CNRS: *The morse simulator documentation*, 2016. `https://www.openrobots.org/morse/doc/stable/morse.html`, (acessado em 24/02/2023). 1

[4] Benjamin Moran: *Esper is a lightweight entity system module for python, with a focus on performance*, 2022. `https://github.com/benmoran56/esper`, (acessado em 29/01/2023). 5, 6

[5] ROS Planning: *Nav2*, 2020. `https://navigation.ros.org/`, (acessado em 17/05/2022). 9

[6] PickNik Robotics: *Moveit 2 tutorials*, 2023. `https://moveit.picknik.ai/foxy/index.html`, (acessado em 27/01/2023). 9

[7] Open Robotics: *Ros - robot operating system*, 2021. `https://ros.org/`, (acessado em 15/02/2023). 9

[8] Araújo, Gabriel: *Simulation and execution of dynamic behavior trees in the service robots context*. Universidade de Brasília, 1(1), 2021. 11

[9] Splintered Reality: *Py trees*, 2023. `https://py-trees.readthedocs.io/en/release-2.2.x/`, (acessado em 05/02/2023). 12

[10] Splintered Reality: *Py trees for ros*, 2021. `https://py-trees-ros.readthedocs.io/en/release-2.1.x/`, (acessado em 05/02/2023). 12

[11] Siegwart, Roland, Illah R. Nourbakhsh e Davide Scaramuzza: *Introduction to Autonomous Mobile Robots*. The MIT Press, 2nd edição, 2011, ISBN 0262015358. 12

[12] Goodwin, Graham C., Stefan F. Graebe e Mario E. Salgado: *Control System Design*. Prentice Hall PTR, USA, 1st edição, 2000, ISBN 0139586539. 15

[13] Hirpo, Boru Diriba e Wang Zhongmin: *Design and control for differential drive mobile robot*. International Journal of Engineering Research & Technology (IJERT), 6(10):327–334, 2017. 22

[14] Novel, B. d'Andréa, G. Campion e G. Bastin: *Control of nonholonomic wheeled mobile robots by state feedback linearization.* The International Journal of Robotics Research, 14(6):543–559, 1995. https://doi.org/10.1177/027836499501400602. 23

[15] Casalino, G., M. Aicardi, A. Bicchi e A. Balestrino: *Closed-loop steering for unicycle-like vehicles: A simple lyapunov like approach.* IFAC Proceedings Volumes, 27(14):335–342, 1994, ISSN 1474-6670. https://www.sciencedirect.com/science/article/pii/S1474667017473356, Fourth IFAC Symposium on Robot Control, Capri, Italy, September 19-21, 1994. 23