Universidade de Brasília

Faculdade de Tecnologia

Planejamento de Rotas e Seguimento de Trajetória para a Plataforma NAO

Lívia Gomes Costa Fonseca

TRABALHO DE CONCLUSÃO DE CURSO

ENGENHARIA DE CONTROLE E AUTOMAÇÃO

Brasília

2025

Universidade de Brasília
Faculdade de Tecnologia

Planejamento de Rotas e Seguimento de Trajetória para a Plataforma NAO

Lívia Gomes Costa Fonseca

Trabalho de Conclusão de Curso submetido como requisito parcial para obtenção do grau de Engenheiro de Controle e Automação.

Orientador: Prof. Dr. Roberto de Souza Baptista

Brasília
2025

FICHA CATALOGRÁFICA

Fonseca, Lívia Gomes Costa.

Planejamento de Rotas e Seguimento de Trajetória para a Plataforma NAO / Lívia Gomes Costa Fonseca; orientador Roberto de Souza Baptista. –– Brasília, 2025.

57 p.

Trabalho de Conclusão de Curso (Engenharia de Controle e Automação) –– Universidade de Brasília, 2025.

1. Planjamento de trajetória. 2. NAO. 3. Robótica móvel. 4. RoboCup. I. Baptista, Roberto de Souza, orient. II. Título.

Universidade de Brasília
Faculdade de Tecnologia

Planejamento de Rotas e Seguimento de Trajetória para a Plataforma NAO

Lívia Gomes Costa Fonseca

Trabalho de Conclusão de Curso submetido como requisito parcial para obtenção do grau de Engenheiro de Controle e Automação.

Trabalho aprovado. Brasília, 26 de fevereiro de 2025:

Prof. Dr. Roberto de Souza Baptista, UnB/FGA
Orientador

Prof. Dr. Geovany Araujo Borges,
UnB/FT/ENE
Examinador interno

Prof. Dr. Walter Britto Vidal Filho,
UnB/FT/ENM
Examinador interno

Agradecimentos

Primeiramete, gostaria de agradecer ao meu orientador, Roberto Baptista por toda a paciência ao longo da confeção desse trabalho, não foi um processo fácil. Gostaria de agradecer, também, à *UnBeatables* por todos os aprendizados, amizades e aventuras em competições que tive no tempo em que fui da equipe. Obrigada a todos os amigos que fiz durante esses anos de faculdade. Pessoas que fizeram parte da minha rotina, dos dias divertidos, dos dias estressantes, vou levar essas amizades comigo para o resto da vida. Finalmente, gostaria de agadecer à minha família que me deu a oportunidade de poder estudar e me apoiou em todo esse processo.

"If you find that you're spending almost all your time on theory,
start turning some attention to practical things;
it will improve your theories.
If you find that you're spending almost all your time on practice,
start turning some attention to theoretical things;
it will improve your practice."
(Donald Knuth)

Resumo

Na robótica móvel, a fim de completar tarefas, é necessário que o robô seja capaz de ir de um ponto a outro sem colidir com obstáculos que existam no caminho. No contexto de futebol de robôs, ele deve se locomover até o destino, a bola, por exemplo, sem colidir com os outros jogadores. Esse trabalho tem como objetivo realizar o planejamento de rotas, que consiste em encontrar e parametrizar um caminho entre o robô e seu destino final evitando obstáculos, e fazer o robô seguir essa trajetória. Para isso, serão usados planejadores para definir essa trajetória, além de técnicas de controle para realizar o seguimento da trajetória em um contexto de futebol de robôs. Todos esses processos serão realizados para a plataforma humanoide NAO, robô que a Universidade de Brasília (UnB) possui no Laboratório de Automação e Robótica (LARA).

Palavras-chave: Planjamento de trajetória; NAO; robótica móvel; RoboCup.

Abstract

In order to complete tasks, a mobile robot needs to be able to move from one point to another without colliding with obstacles along the way. In the context of robot soccer, the robot must navigate to its destination, such as the ball, without colliding with other players. The goal of this work is to perform path planning, which involves finding a path between the robot and its final destination, avoiding obstacles, and ensuring that the robot follows this path. To achieve this goal, path planners will be used to define the trajectory, along with control techniques to ensure trajectory tracking in the context of robot soccer. All these processes will be carried out on the NAO humanoid platform, a robot owned by the University of Brasília (UnB) in the Automation and Robotics Laboratory (LARA).

Keywords: Path planning; NAO; mobile robotics; RoboCup.

Lista de figuras

Lista de tabelas

Sumário

1 Introdução

Ao longo da história, a humanidade sempre buscou criar ferramentas e, tempos depois, máquinas, para facilitar e eventualmente substituir o trabalho humano em certas áreas. Tal busca culminou na robótica moderna. Nessa área, um robô é definido como um sistema autônomo capaz de perceber o ambiente e agir para atingir um objetivo estabelecido (Mataric, 2011).

Quanto à capacidade de movimentação, os robôs podem ser divididos em duas categorias: robô manipulador e robô móvel. A estrutura mecânica de um robô manipulador consiste em partes rígidas unidas por articulações (juntas) (Siciliano *et al.*, 2008). Os robôs manipuladores possuem sua plataforma fixa durante toda a sua execução de movimento, e sua área de trabalho é determinada por sua estrutura física. Os robôs móveis, por sua vez, são caracterizados por uma base móvel, o que os permite mover-se livremente pelo ambiente (Siciliano *et al.*, 2008). O sistema de locomoção de um robô móvel pode consistir em rodas ou pernas. Robôs com rodas tipicamente consistem em um corpo rígido e um sistema de rodas, que os permite se movimentar. Os robôs dotados de pernas tipicamente possuem uma série de partes rígidas interconectadas por juntas, permitindo, assim, sua movimentação. Entre os robôs móveis com pernas, destacam-se os humanoides.

Robôs humanoides são capazes de imitar certos comportamentos humanos e, por isso, são de grande interesse em pesquisas. Para que os robôs humanoides possam realizar diversas tarefas, a navegação nos ambientes é fundamental.

O objetivo deste trabalho é realizar o planejamento de rotas, que consiste em encontrar e parametrizar um caminho entre o robô e seu destino final, evitando obstáculos ao longo do trajeto, e fazer com que o robô siga essa trajetória. Para isso, serão utilizados planejadores de trajetória que definirão o melhor caminho, além de técnicas de controle que permitirão ao robô seguir a trajetória estabelecida de forma precisa. Todo esse processo será aplicado no contexto de futebol de robôs, em que o robô deverá se movimentar de maneira autônoma para realizar tarefas específicas no campo. A plataforma utilizada será o robô NAO, um modelo humanoide que a Universidade de Brasília (UnB) possui em seu Laboratório de Automação e Robótica (LARA).

1.1 Contextualização

Durante o *Workshop on Grand Challenges in Artificial Intelligence*, realizado em Tóquio no ano de 1992, o principal tópico de discussão envolvia os possíveis projetos que apresentassem grandes desafios técnicos (Kitano *et al.*, 1993). Essa conferência levou a discussões sobre se utilizar o futebol como forma de promover o desenvolvimento tecnológico com

impacto social, econômico e científico relevantes. Como fruto dessa conferência, surgiu a *RoboCup*. O objetivo final dessa iniciativa é desenvolver um time de robôs que seja capaz de ganhar da seleção humana vencedora da Copa do Mundo da Federação Internacional de Futebol (FIFA)[1] no ano de 2050 (Federation,).

A *RoboCup* possui cinco ligas de futebol de robôs, cada uma com suas características e desafios particulares. Entre as ligas está a *Standard Platform League* (SPL)[2]. Na SPL, todos os times utilizam a mesma plataforma, o robô NAO produzido pela *SoftBank Robotics*, e o hardware não pode ser alterado. Os robôs jogam de maneira completamente autônoma e devem tomar decisões sozinhos, mas jogam como um time utilizando comunicação entre os jogadores. As partidas acontecem em um campo verde com linhas brancas e marcas do gol e a bola é a tradicional branca e preta. E, a cada ano, são introduzidos novos desafios (Federation,).Para que o robô consiga jogar futebol, é necessário que ele seja capaz de planejar e seguir uma trajetória no campo, desviando de possíveis obstáculos, como outros jogadores.

1.2 A plataforma NAO

O NAO (Figura 1.1) é um robô humanoide produzido pela *Softbank Robotics* e é utilizado atualmente como plataforma da *RoboCup*. A versão 6 do robô possui os seguintes componentes: LEDs; botões táteis e de contato na cabeça, peito, mãos e pés; alto-falantes e microfones na cabeça; duas câmeras também em sua cabeça; sensores de posição em seus motores; sensores resistivos localizados nos pés do robô; dois pares de sonar localizados em seu peito; e uma unidade inercial. As especificações dos dispositivos estão apresentadas na tabela 1.1. O modelo cinemático do robô é descrito por Gouaillier et al. (Gouaillier *et al.*, 2009).

Tabela 1.1 – Especificações dos disposivos dos robôs NAOs versão v4 e v6

Dispositivo	**Parâmetro**	**V4**	**V6**
Placa-mãe	Processador	ATOM Z530	ATOM E3845
	CPU	uni core	quad core
	Frequência de relógio	1,6 GHz	1,91 GHz
	RAM	1 GB	4 GB DDR3

[1] Em francês, Fédération Internationale de Football Association, FIFA.

[2] Em português, Liga de Plataforma Padrão.

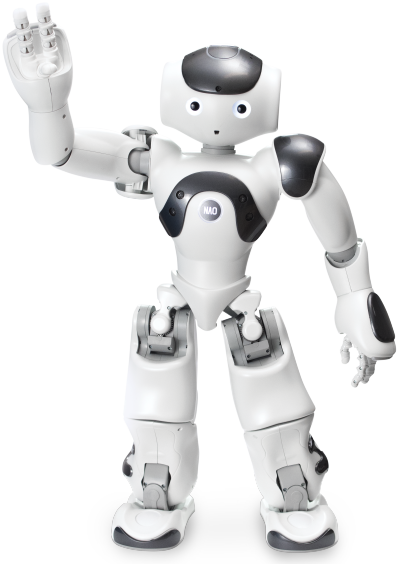


Figura 1.1 – Robô NAO. Fonte: RobotLAB

1.3 NAOqi

O software principal que roda no NAO é chamado de NAOqi. Esse software é uma distribuição GNU/Linux baseada em Gentoo e adaptada para as necessidades do robô. Através do *framework* NAOqi o usuário consegue utilizar softwares para controlar todo o comportamento do robô. Esse *framework* contém bibliotecas, compiladores e Interface de Programação de Aplicação (API). Através da API, é possível carregar módulos para acessar os sensores e atuadores do NAO, além de executar rotinas já pré-definidas. O funcionamento dessa chamada de métodos está ilustrado na Figura 1.2.

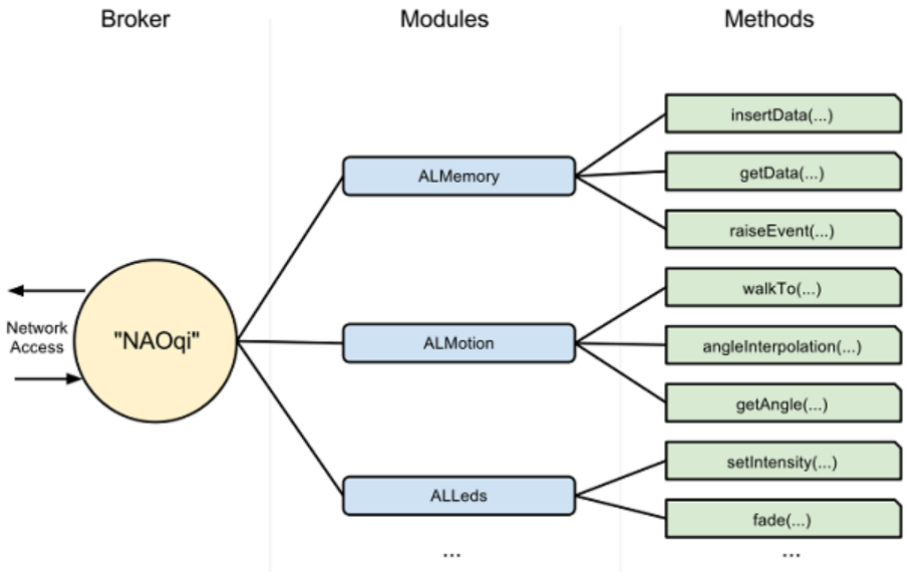


Figura 1.2 – Funcionamento do carregamento de módulos da NAOqi. Fonte: *SoftBank Robotics*

1.4 Campo da SPL

A cada ano, as regras da *RoboCup* se alteram com o intuito de tornar as condições dos jogos mais parecidas com as de um jogo de futebol humano. O livro de regras da *RoboCup* define, além das regras do jogo em si, as especificações do campo como material e dimensões. O livro de regras publicado em 2019 (Committee,) definiu um campo com grama sintética verde e marcação em branco para linhas de meio de campo, círculo central, laterais, fundo, pequena área, cruz do pênalti e marca de meio de campo, conforme a Figura 1.3, cujas respectivas dimensões estão na Tabela 1.2. As especificações do gol podem ser observadas nas Figuras 1.4 e 1.5.

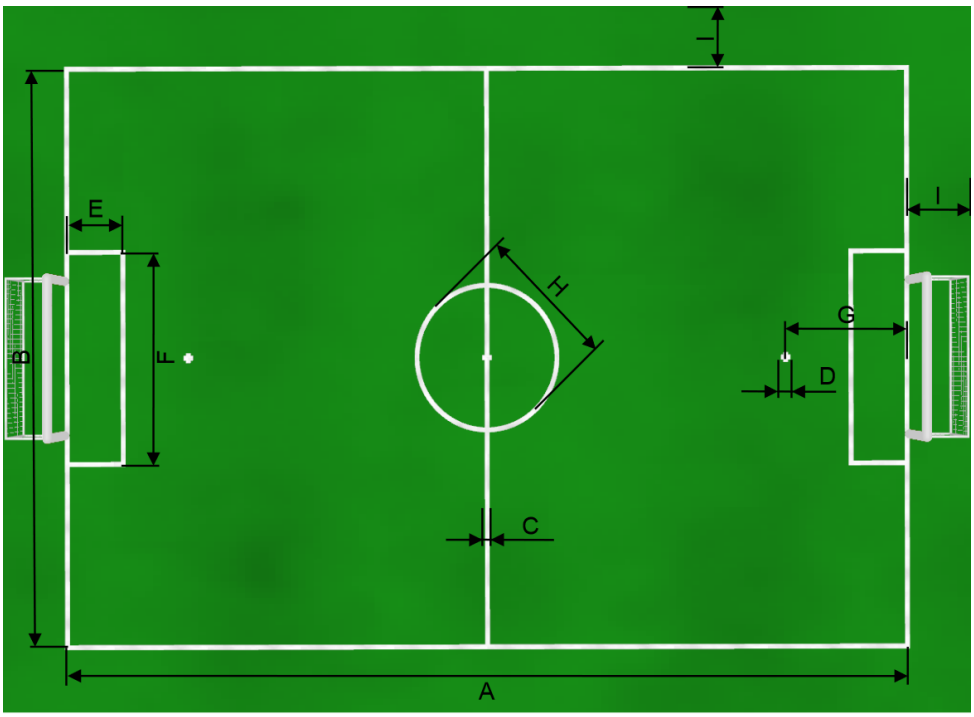


Figura 1.3 – Diagrama com as marcações do campo de futebol da SPL. Fonte: *RoboCup Technical Committee*

Tabela 1.2 – Medidas do campo de futebol da SPL

Símbolo	Descrição	Medida (mm)
A	Comprimento do campo	9000
B	Largura do campo	6000
C	Largura da linha	50
D	Marca do pênalti	100
E	Comprimento da área do pênalti	600
F	Largura da área do pênalti	2200
G	Distância da marca do pênalti	1300
H	Diâmetro do círculo de meio de campo	1500
I	Distância entre linha de campo e fim do gramado	700

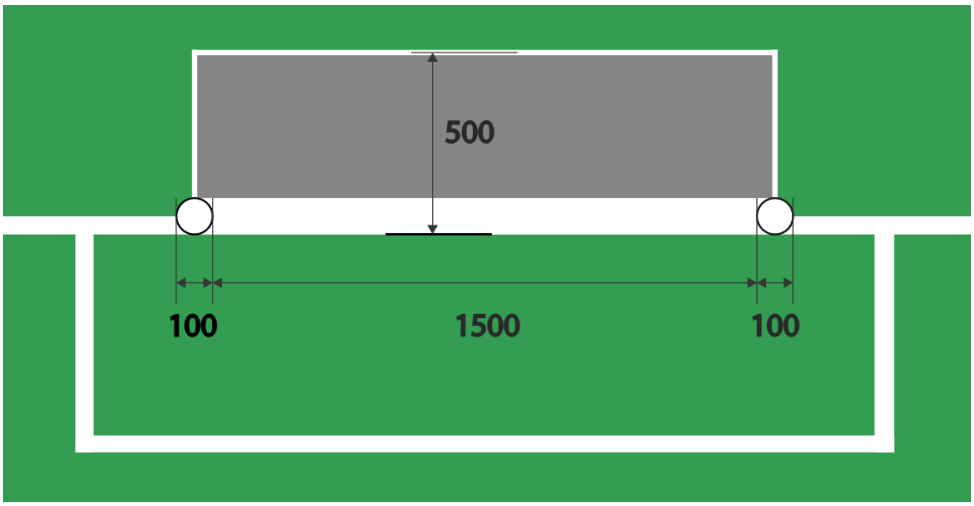


Figura 1.4 – Diagrama com as marcações do gol da SPL. Fonte: *RoboCup Technical Committee*

1.5 Ambiente de simulação

Para realizar as simulações de planejamento e seguimento de rotas, será usado o modelo do robô NAO existente no simulador *CoppeliSim* (Rohmer; Singh; Freese, 2013). A cena construída pode ser observada na Figura 1.6. O simulador fornece a posição e orientação do NAO a cada instante de tempo, e o controle e planejamento de trajetória podem ser feitos em um script externo.

1.5.1 Possíveis cenários

Diferentes cenários são possíveis durante um jogo de futebol. É possível que o caminho à frente esteja livre e o NAO necessite apenas seguir uma linha reta até seu objetivo, porém também é possível que haja vários robôs à frente bloqueando a passagem. Além disso, apesar de as dimensões do campo serem fixas, o cenário é dinâmico, pois os outros robôs (obstáculos) estão em movimento e a posição final do NAO, que geralmente é à frente da bola, também

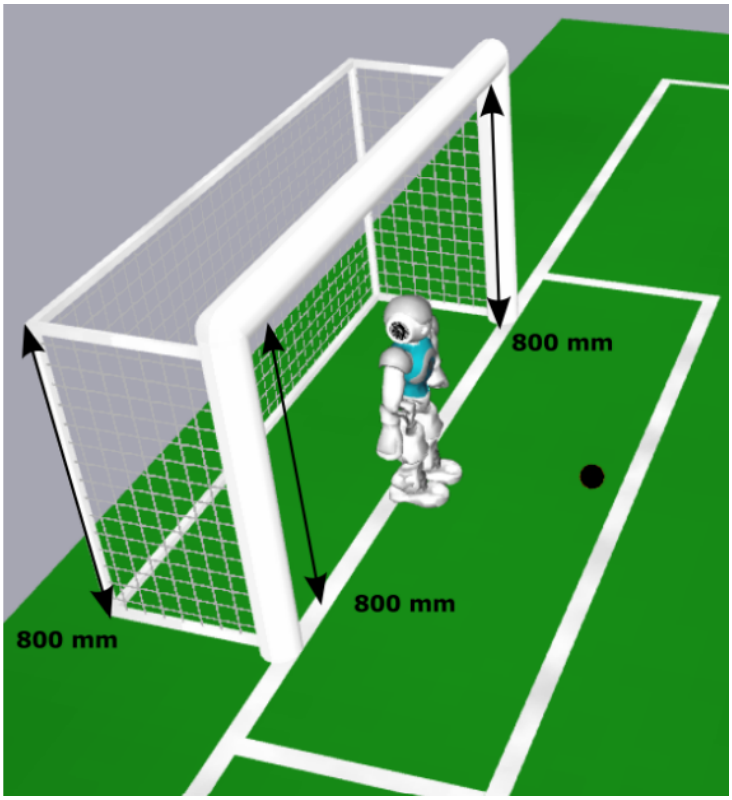


Figura 1.5 – Diagrama com as medidas do gol da SPL. Fonte: *RoboCup Technical Committee*

Figura 1.6 – Ambiente de simulação no CoppeliaSim. Fonte: Autora

está. As Figuras 1.7 e 1.8 ilustram dois possíveis cenários observados durante um jogo. O NAO que é objeto de estudo está marcado por uma caixa branca à sua volta.

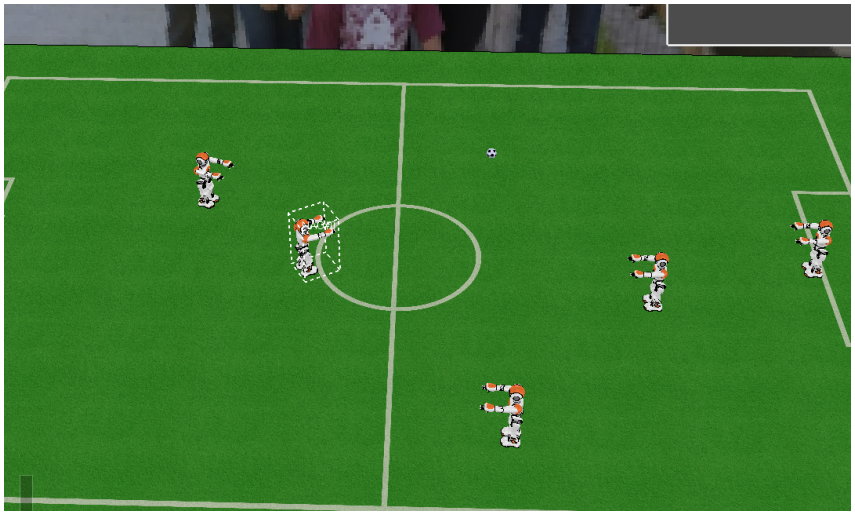


Figura 1.7 – Caminho até a bola livre de obstáculos. Fonte: Autora

Figura 1.8 – Caminho até a bola obstruído por outros jogadores. Fonte: Autora

1.6 Revisão Bibliográfica

Diversas abordagens de planejamento de trajetória são encontradas na literatura e são aplicadas em diferentes categorias de robôs, como robôs com rodas e robôs humanoides. Esse trabalho propõe um planejamento de trajetória para o robô humanoide NAO.

Ottoni (Ottoni; Fetter, 2003) descreve um sistema de planejamento de trajetória para o robô móvel Twil em um ambiente desconhecido. Apenas as posições final e inicial são conhecidas e suas respectivas orientações. Para reconhecer o ambiente, o robô utiliza um sonar. O sistema de planejamento é baseado no método da decomposição em células aproximada, que consiste em decompor o espaço em pequenas células, de forma que seja trivial ir de uma célula para a seguinte.

Adorno et al. (Adorno; Rocha; Borges, 2005) apresentam um planejamento de trajetória para um robô omnidirecional em um ambiente conhecido. A proposta baseia-se no Mapa de Rotas Probabilístico (Kavraki *et al.*, 1996), com suavização de trajetórias baseada em um interpolador por curvas de Bezier e apresenta a implementação de uma estratégia de amostragem gaussiana para as configurações aleatórias.

Em seu artigo, Hugel et al. (Hugel; Jouandeau, 2012) apresentam um algoritmo de locomoção e planejamento de trajetória para o robô NAO. Os padrões de caminhada foram feitos para possibilitar que o robô seja capaz de jogar futebol no campo simulado da RoboCup. A caminhada é baseada no modelo 3D-LIP e o artigo propõe detalhar como conectar as marchas primitivas, especialmente no começo e no final da caminhada.

A capacidade de resolver labirintos tem sido um tópico frequente em diversas competições de robótica e, entre diversas abordagens utilizando diferentes sensores, aquelas que utilizam imagens de câmeras têm ganhado cada vez mais espaço. Nesse contexto, Rodriguez-Tirado et al. (Rodriguez-Tirado *et al.*, 2020) propõem um pipeline que permite que o robô NAO resolva, de maneira autônoma, um labirinto, utilizando imagens obtidas pela câmera. Além disso, também permite que o robô seja capaz de transferir a solução para outros robôs. Para a navegação no labirinto, foi utilizado o algoritmo Tremaux, que se mostrou rápido e eficiente.

Considerando um ambiente familiar e os vários fatores de diferentes algoritmos de planejamento de trajetória, o algoritmo de campo potencial artificial é selecionado como o algoritmo de planejamento da trajetória. Além de derivar o algoritmo de campo potencial artificial tradicional, DongLi et al. (Li *et al.*, 2018) também introduziram soluções, como a forma de resolver os vários defeitos do algoritmo de campo potencial artificial tradicional, a simulação do algoritmo de campo potencial artificial tradicional e verificar a viabilidade de modificação do algoritmo. A integridade e superioridade do processo são as marcas deste artigo.

No contexto em que o ambiente em que o robô está inserido é desconhecido, Zhou et al. (Zhai; Egerstedt; Zhou, 2019) introduzem um método baseado em grafo e guiado por potencial para planejamento de trajetória em que os obstáculos só são conhecidos a partir do momento em que o robô se aproxima. O algoritmo proposto gera um grafo que conecta as configurações inicial e final e, a cada nova informação que o robô obtém sobre o ambiente, o grafo é atualizado.

Para esse mesmo contexto Zookar et al. (Fakoor; Kosari; Jafarzadeh, 2016) propõe a utilização do processo de decisão *fuzzy Markov* (FMDP)[3], que pode ser considerado como uma extensão das cadeias de Markov, sendo definido como um processo estocástico em tempo discreto, associado posteriormente com uma lógica *fuzzy*. No fluxo de controle, o

[3] do inglês, fuzzy Markov decision process

ambiente é percebido utilizando estratégias de visão computacional e a decisão do caminho a ser percorrido utiliza FMDP.

Khaksar et al. (Khaksar *et al.*, 2015) faz uma revisão de diversos métodos para fazer o planejamento de trajetória em ambientes desconhecidos. Os métodos são separados em duas categorias, *Optimized Classic Approaches*, que inclui campos potenciais, busca heurística, entre outros, e *Evolutionary and Hybrid Approaches*, que inclui inteligência artificial, algoritmos genéticos, entre outros. A comparação entre as duas categorias foi feita e suas vantagens e desvantagens listadas. Em termos de tempo computacional, as abordagens da segunda categoria obtiveram um desempenho melhor, tendo em vista que as abordagens clássicas são matematicamente mais complexas, o que demanda alto recurso computacional. Com relação à eficiência em desviar de obstáculos, as abordagens do grupo *Evolutionary and Hybrid Approaches* mostraram uma ligeira vantagem. E, por fim, com relação ao tamanho do caminho percorrido para se chegar à posição final, essa métrica foi mais dependente do ambiente do que da abordagem de planejamento de trajetória em si.

2 Fundamentos e Modelagem Matemática

Nesse capítulo são apresentados os fundamentos teóricos utilizados para o planejamento e seguimento de trajetória. Primeiramente, apresentamos como a plataforma NAO é representada no espaço. Em seguida, temos a fundamentação teórica dos modelos usados para planejamento de trajetória. Por fim, apresentamos os controladores utilizados para o seguimento da trajetória.

2.1 Representação espacial

Para o problema de planejamento e seguimento de uma trajetória, é imprescindível termos uma representação do robô no ambiente. No espaço tridimensional, um ponto é facilmente representado por um vetor de posição em relação a um sistema de coordenadas de referência (Corke, 2013). Esse vetor indica a translação em relação à origem desse sistema. Um objeto é um conjunto de pontos. Considerando que o objeto é rígido, ou seja, seus pontos mantêm uma posição relativa fixa em relação ao sistema de coordenadas do objeto, em vez de descrever os pontos separadamente, descrevemos a posição e a orientação do objeto através da posição e orientação do seu sistema de coordenadas (Siciliano *et al.*, 2008). O sistema de coordenadas do robô é comumente definido em seu centro de massa, ou em algum outro ponto de interesse.

A plataforma NAO é um robô humanoide capaz de se movimentar de forma autônoma. Para esse trabalho, vamos considerá-lo um objeto rígido que se movimenta em um plano horizontal. Definimos uma base inercial ortonormal arbitrária $\{0, \overrightarrow{I}_1, \overrightarrow{I}_2\}$, e um ponto de referência P, no robô, de base arbitrária $\{\overrightarrow{x}_1, \overrightarrow{x}_2\}$. Dessa forma, podemos especificar, de forma completa, a pose do robô usando apenas as variáveis x, y e θ. Assim, x e y correspondem às coordenadas de P, e θ a orientação de $\{\overrightarrow{x}_1, \overrightarrow{x}_2\}$ com relação à base inercial $\{\overrightarrow{I}_1, \overrightarrow{I}_2\}$ (Campion; Bastin; Dandrea-Novel, 1996). Isso pode ser observado na figura 2.1.

Podemos, então, definir a pose do robô NAO através do vetor ξ (equação 2.1).

$$\xi \triangleq \begin{bmatrix} x \\ y \\ \theta \end{bmatrix} \tag{2.1}$$

E também temos a matriz de rotação R, que transfere o sistema de coordenadas do global para o referente ao robô.

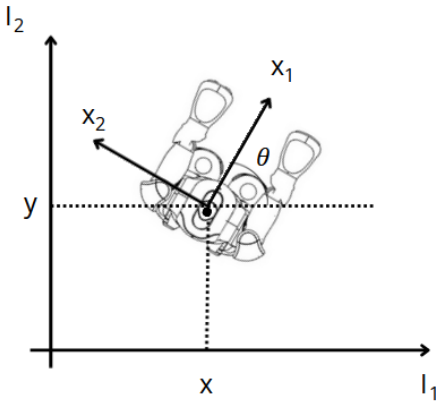


Figura 2.1 – Representação espacial do robô NAO. Fonte: Autora

$$R \triangleq \begin{bmatrix} cos(\theta) & sen(\theta) & 0 \\ -sen(\theta) & cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.2}$$

2.2 Planejamento de trajetória

O planejamento de trajetória para robôs móveis consiste em encontrar e parametrizar, considerando posição e orientação, um caminho entre o robô e seu destino final de forma a evitar obstáculos (Siciliano *et al.*, 2008).

Diversos algoritmos podem ser usados para fazer o planejamento da trajetória, um deles é a Árvore Aleatória de Exploração Rápida (*RRT* do inglês *Rapidly-exploring random tree* (Lavalle, 1998)). O *RRT* constrói uma árvore a partir da configuração inicial usando amostras aleatórias do espaço de busca (Figura 2.2). Para cada amostra, é feita uma tentativa de conexão com o nó mais próximo da árvore. Se a conexão for viável, ou seja, não cruzar com nenhum obstáculo e obedecer a qualquer outra restrição imposta, o novo nó é adicionado à árvore. Quando a configuração de objetivo é atingida, o algoritmo se encerra.

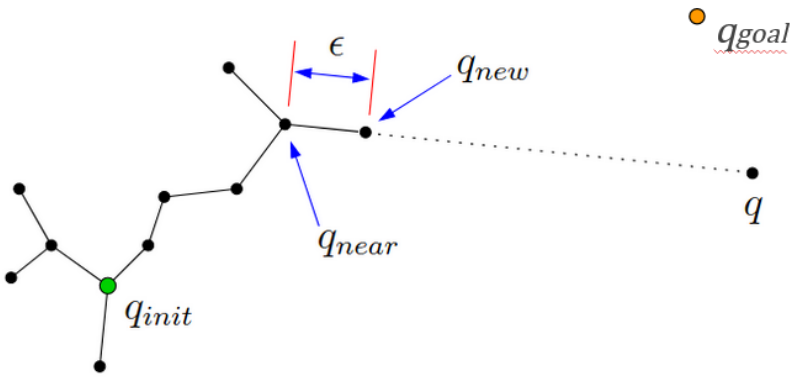


Figura 2.2 – Algoritmo *RRT*. Fonte: (Kuffner; LaValle, 2000)

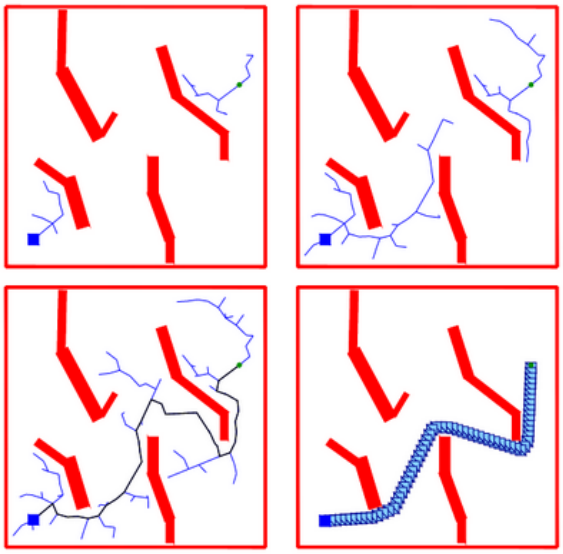


Figura 2.3 – Algoritmo *RRTconnect*. Fonte: (Kuffner; LaValle, 2000)

Uma variação do *RRT* é o *RRTconnect* (Kuffner; LaValle, 2000). Nesse algoritmo, são construídas duas árvores, uma com origem no ponto inicial, onde está o robô, e a outra com origem no ponto final, que é a configuração de objetivo. Cada árvore explora o seu entorno e avança em direção uma à outra, como mostrado na figura 2.3. Quando as duas árvores se encontram, o caminho é obtido.

Outro método que apresenta um bom desempenho para planejamento de trajetória para robôs móveis (Adorno; Rocha; Borges, 2005) é o Mapa de Rotas Probabilístico *PRM*, do inglês *Probabilistic Roadmap*. O mapa de rotas é um grafo unidirecional que contém os nós, que são as configurações do robô, e as bordas, que são os caminhos que conectam os nós. Esse algoritmo consiste em duas fases: fase de aprendizado e fase de questionamento. Na fase de aprendizado, o algoritmo gera nós aleatórios pelo espaço e os conecta de forma que não haja colisão. Os caminhos são calculados por um planejador local, que é extremamente rápido. Em seguida, armazena essas informações em um grafo (Figura 2.4); caso o planejador das bordas seja determinístico, os caminhos não são diretamente armazenados no grafo, o que gera uma economia de espaço de armazenamento (Kavraki *et al.*, 1996).

Na fase de questionamento, dados os pontos de origem e destino do robô, o algoritmo tenta encontrar um caminho, no grafo, que conecte esses dois pontos. Dada uma configuração de início e uma configuração de objetivo, tentamos conectar esses dois pontos a algum dos nós armazenados no grafo, com caminhos viáveis. Se isso falhar, a consulta falha. Caso contrário, é computado um caminho para os nós subsequentes. Um caminho viável da origem para o destino é eventualmente construído, e todas as bordas são recalculadas (Kavraki *et al.*, 1996).

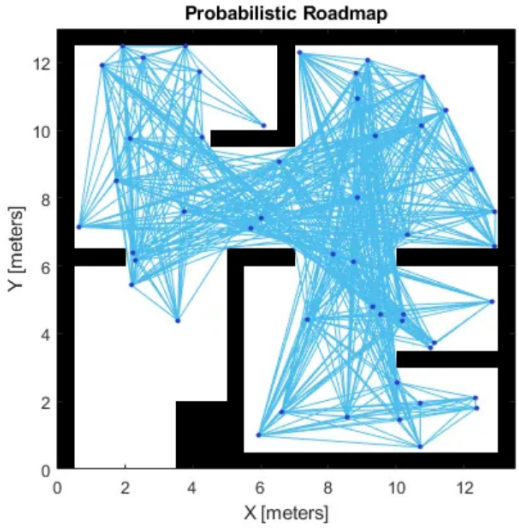


Figura 2.4 – Algoritmo *PRM*. Fonte: (Khokhar, 2025)

2.3 Seguimento de trajetória

O seguimento de trajetória consiste em programar o robô para seguir uma trajetória no espaço ao longo do tempo. Pode ser uma trajetória em 3D, comum para robôs manipuladores e braços robóticos, ou 2D, comum para robôs com rodas e humanoides que desejam seguir um caminho, como é o caso da plataforma NAO.

2.3.1 Seguimento da trajetória

Para a realização do controle de trajetória, existem algumas abordagens diferentes, como seguimento de linha, de ponto. Para esse trabalho, foi usada uma abordagem em que o robô segue um ponto em movimento que acompanha uma trajetória pré-estabelecida.

Considerando $\xi^*(t)$ a pose desejada, $\xi(t)$ a pose atual e $\epsilon(t)$ o erro de controle (equação 2.3).

$$\epsilon(t) = \xi^*(t) - \xi(t) \tag{2.3}$$

Temos que:

$$\dot{\epsilon}(t) = \dot{\xi}^*(t) - \dot{\xi}(t) \tag{2.4}$$

Seja a matriz A uma matriz quadrada mxm de autovalores de parte real negativa 2.6, temos o objetivo de encontrar uma lei de controle que satisfaça a equação 2.5, pois para qualquer $\varepsilon(0)$, o módulo de $\varepsilon(t)$ tende a zero quando t tende a infinito. Isso ocorre porque o modelo corresponde a um sistema autônomo no espaço de estados, e é estável pois os autovalores da matriz A têm parte real negativa (Khalil, 2002).

$$\dot{\epsilon}(t) = A \cdot \epsilon(t) \tag{2.5}$$

$$A = \begin{bmatrix} \lambda_1 & 0 & 0 \\ 0 & \lambda_2 & 0 \\ 0 & 0 & \lambda_3 \end{bmatrix}, \lambda_i < 0, i = 1,2,3 \tag{2.6}$$

Dessa forma, temos que:

$$\dot{\epsilon}(t) = A \cdot \epsilon(t) = \dot{\xi}^*(t) - \dot{\xi}(t) \tag{2.7}$$

$$\Rightarrow \dot{\xi}(t) = \dot{\xi}^*(t) - A \cdot \epsilon(t) \tag{2.8}$$

Definindo $\dot{q}(t)$ o vetor de velocidades lineares e angular relativo ao eixo de coordenadas do robô NAO 2.10, temos que:

$$\dot{q}(t) = R \cdot \dot{\xi}(t) = R \cdot (\dot{\xi}^*(t) - A \cdot \epsilon(t)) \tag{2.9}$$

$$\dot{q}(t) = \begin{bmatrix} \dot{x}_r(t) \\ \dot{y}_r(t) \\ \dot{\theta}_r(t) \end{bmatrix} \tag{2.10}$$

O controlador pode ser observado na figura 2.5.

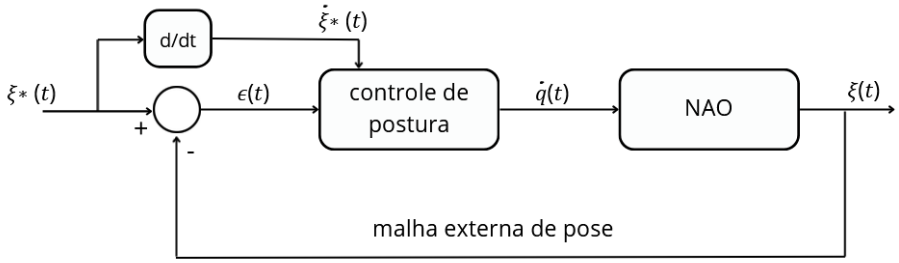


Figura 2.5 – Esquema da malha de controle. Fonte: Autora

2.3.2 Movimentando para uma pose fixa

O primeiro passo para que o robô consiga seguir uma trajetória é fazê-lo ir a um ponto específico. O controle é feito com o objetivo de encontrar $\dot{q}(t)$ que é a matriz de velocidades lineares e angular (equação 2.10), sendo $\dot{x}(t)$ a velocidade linear no eixo x, $\dot{y}(t)$ a velocidade linear em y e $\dot{\theta}(t)$, a velocidade angular em z, com relação aos eixos de coordenadas do robô.

Podemos usar a mesma lei de controle apresentada em 2.9 e, como o ponto é fixo, $\dot{\xi}^*(t) = 0$. Assim, obtemos a equação 2.11.

$$\dot{q}(t) = R \cdot \dot{\xi}(t) = -R \cdot A \cdot \epsilon(t) \tag{2.11}$$

Uma outra abordagem seria restringir a movimentação do robô em y, relativo ao próprio eixo de coordenadas. Como sugerido por Corke et al. (Corke, 2013), para atingir uma pose específica, controlaremos a velocidade proporcionalmente à distância do ponto objetivo . Dessa forma, encontraremos $\dot{q}(t)$. Assim, calculamos a velocidade em x de acordo com a equação 2.12.

$$v_x = K_v \cdot \sqrt{(x^* - x)^2 + (y^* - y)^2}, K_v > 0 \tag{2.12}$$

$$\theta^* = tan^{-1} \frac{y^* - y}{x^* - x} \tag{2.13}$$

$$\dot{\theta} = K_h(\theta^* \ominus \theta), K_h > 0 \tag{2.14}$$

Assim, obtemos:

$$\dot{\xi}(t) = \begin{bmatrix} v_x(t) \\ 0 \\ \dot{\theta}(t) \end{bmatrix} \tag{2.15}$$

$$\Rightarrow \dot{q}(t) = R \cdot \dot{\xi}(t) = R \cdot \begin{bmatrix} v_x(t) \\ 0 \\ \dot{\theta}(t) \end{bmatrix} \tag{2.16}$$

3 Desenvolvimento

Nesse capítulo são apresentadas as bibliotecas utilizadas e toda a preparação do ambiente de simulação. Além disso, é apresentada toda a implementação dos sistemas.

3.1 Bibliotecas e frameworks utilizados

3.1.1 NAOqi API e NAOqi SDK

Para realizar a locomoção do NAO, é utilizada sua biblioteca padrão de movimentação. Através da API da NAOqi, é possível acionar os sensores e atuadores do robô, além disso, existem rotinas pré-definidas de movimentação que permitem definir uma marcha para o NAO. Através das funções específicas de movimentação, incluídas na classe *ALMotionProxy*, é possível definir a velocidade da marcha, tamanho do passo, tempo de duração da caminhada, direção e angulação, o que permite que o robô faça curvas. As principais funções dessa biblioteca a serem usadas são:

- ALMotionProxy::move
  Faz com que o robô se mova a uma dada velocidade.
- ALMotionProxy::setFootSteps
  Define o tamanho do passo do robô.
- ALMotionProxy::moveInit
  Inicializa o processo de movimento.
- ALMotionProxy::stopMove
  Interrompe o processo de movimento.

A NAOqi SDK cria um robô virtual e, a partir dos comandos da NAOqi API, obtém a posição das juntas do robô. Essa informação é passada para o *CoppeliaSim* para fazer o NAO da simulação se movimentar.

3.1.2 Coppelia Remote API

A Coppelia Remote API é uma interface de programação de aplicativos (API) que permite interagir remotamente com o software *CoppeliaSim* (Rohmer; Singh; Freese, 2013). A CoppeliaSim permite simular diversos tipos de sistemas, incluindo robôs móveis, manipuladores, sistemas multirrobôs, entre outros. A API remota possibilita o controle remoto de tais simulações, permitindo a comunicação entre o *CoppeliaSim* e outros programas, normalmente em outras linguagens de programação. Essa API foi usada para obter as coordenadas

dos componentes da simulação: robô, bola, caminho da trajetória, no código Python que foi responsável pelo controle.

3.1.3 OMPL

A biblioteca *Open Motion Planning Library* (OMPL) é uma biblioteca que consiste na implementação de diversos algoritmos do estado da arte para planejamento de trajetória (Şucan; Moll; Kavraki, 2012).

O planejamento de trajetória foi executado por meio de um *plugin* da biblioteca OMPL. Esta biblioteca fornece como parâmetros de configuração:

- O ponto inicial e o ponto final da trajetória
- A área de busca em torno do ponto inicial
  É importante configurar corretamente este valor e garantir que o ponto final desejado esteja dentro da área de busca.
- O objeto "colididor"
  O objeto que vai se mover no ambiente e poderá colidir com objetos da cena.
- Os objetos "colidíveis"
  Objetos da cena que são obstáculos.
- Algoritmo utilizado como planejador.

3.1.4 Python Numpy

A biblioteca Numpy fornece uma sintaxe que permite acessar e manipular vetores unidimensionais, matrizes e vetores de dimensões superiores em linguagem Python (Harris *et al.*, 2020). Nesse trabalho, essa biblioteca foi utilizada para realizar os cálculos dos vetores e matrizes utilizados nos controladores.

3.2 Ambiente de simulação

No simulador *CoppeliaSim* (Rohmer; Singh; Freese, 2013) foi criada uma cena representando uma situação de um jogo de futebol de robôs, mostrada na figura 3.2. Essa cena contém o campo, os gols, a bola, o jogador que será controlado e os demais jogadores. No *CoppeliaSim*, também encontramos o *script* com o planejamento de trajetória. O controlador, no entanto, está em um programa escrito em Python, que executa externamente. Esse programa obtém as informações da cena utilizando *Coppelia Remote API* (seção 3.1.2) e socket, e aciona a NAOqi SDK através da NAOqi API com os valores de velocidades lineares em x e y, e angular em z. Por fim, utilizando um módulo escrito em C++ que comunica com o

CoppeliaSim através da API remota, a NAOqi envia a posição das juntas do robô NAO. Esse fluxo está apresentado na Figura 3.1, e o guia de como conectar o sistema está no *github* [1].

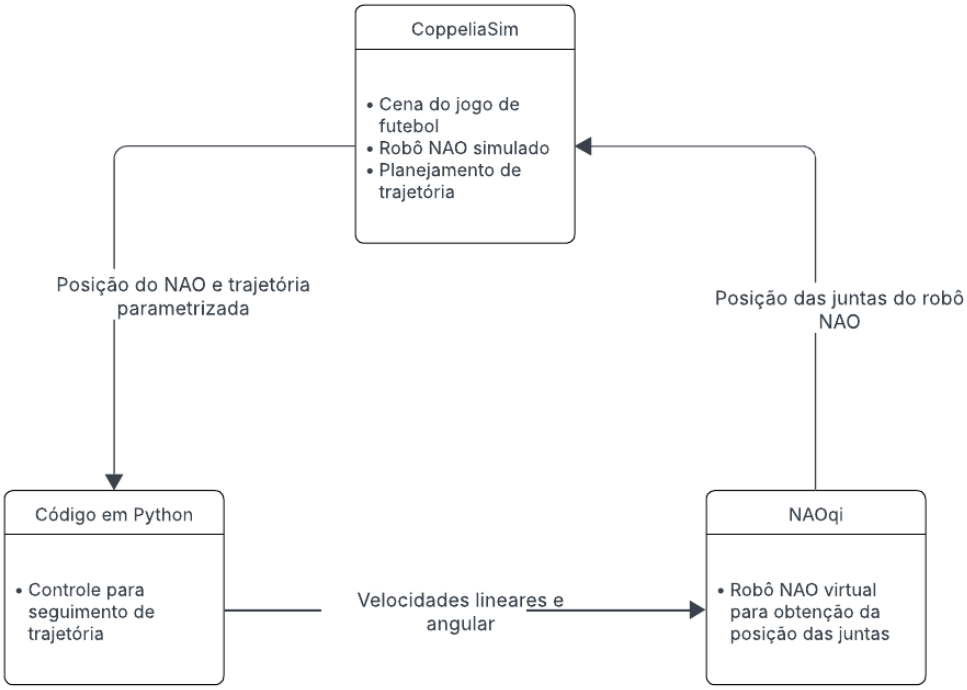


Figura 3.1 – Diagrama do sistema. Fonte: Autora

Figura 3.2 – Ambiente de simulação no *CoppeliaSim*.Fonte: Autora

[1] https://github.com/Liviagcf/nao_path_planning

3.3 Planejamento de trajetória

Para planejar a trajetória, utilizamos a biblioteca OMPL em um script interno no *CoppeliaSim*. Para configurá-la, os seguintes parâmetros foram definidos:

- O ponto inicial da trajetória
  O ponto incial da trajetória é a posição incial do robô NAO;
- O ponto final da trajetória
  O ponto final da trajetória foi definido como a posição da bola, que é onde o robô quer chegar em uma possível situação de jogo;
- A área de busca em torno do ponto inicial
  A área de busca corresponde ao campo de futebol;
- O objeto "colididor"
  O objeto colididor é o robô NAO, contudo, para uma margem de segurança, foi definido um cuboide cujas dimensões são ligeiramente maiores que o NAO;
- Os objetos "colidíveis"
  Os objetos colidíveis consistem nos outros jogadores e nos gols;
- Algoritmo
  O algoritmo que será utilizado para planejar a trajetória.

Em seguida, foi criada a trajetória, utilizando os algoritmos *RRT*, *RRTconnect* e *PRM* e essa trajetória foi enviada para o código em linguagem *Python* responsável pelo controle. Ao executar cada algoritmo de planejamento de trajetória, foi criado um caminho que liga o ponto inicial ao ponto final; este caminho é otimizado, porém ainda é probabilístico. Logo, a cada execução, é formado um caminho diferente.

3.4 Seguimento de trajetória

Para o controlador que faz o robô seguir a trajetória, desenvolvemos códigos em linguagem Python. Para o movimento a um ponto específico, primeiramente obtemos a posição da bola e a definimos como posição de destino. Em seguida, conectamos à NAOqi e obtemos o serviço de movimento ALMotion. Para o caso do controlador no espaço de estados, definimos a matriz diagonal de autovalores A. Obtemos a posição e a orientação atuais do robô no simulador. Calculamos a matriz de rotação R com base na orientação do robô. Calculamos o erro entre a posição desejada e a posição medida do NAO. Calculamos o comando de movimento q com base nesse erro. Enviamos os comandos de movimento ao robô utilizando ALMotion. E, finalmente, quando o robô atinge a posição desejada, paramos o movimento. Já para o caso do controlador com limitação de movimento lateral, o procedimento é parecido; contudo, em vez da matriz de autovalores, definimos os valores das constantes do controlador proporcional. Além disso, utilizamos outra lei de controle, como apresentado na seção 2.3.2

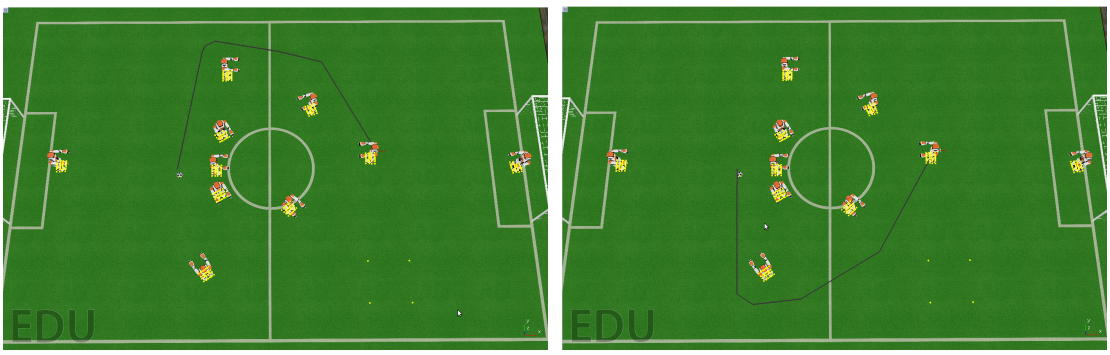
Para o seguimento da trajetória definida, obtemos o caminho calculado via *socket*. Em seguida, conectamos à NAOqi e obtemos o serviço de movimento ALMotion. Definimos a matriz diagonal de autovalores A. Definimos a posição inicial desejada como o primeiro ponto da trajetória. Para cada ponto da trajetória, atualizamos a posição desejada com base no passo atual. Calculamos a taxa de variação do caminho desejado. Para o cálculo dessa taxa, discretizamos a derivada com base na Série de Taylor. Obtemos a posição e a orientação atuais do robô no simulador. Calculamos a matriz de rotação R com base na orientação do robô. Calculamos o erro entre a pose desejada e a pose medida do robô. Calculamos o comando de movimento q com base no erro e na variação do caminho. Enviamos os comandos de movimento ao robô utilizando ALMotion. E, finalmente, após terminar o caminho, paramos o movimento do robô. Os códigos em Python podem ser vistos no apêndice A.

4 Resultados

A execução dos experimentos foi feita em um computador com sistema operacional Linux Ubuntu 24.04 LTS, com processador Intel Core i7, 32 GB de memória RAM e placa de vídeo NVIDIA GeForce RTX 2060.

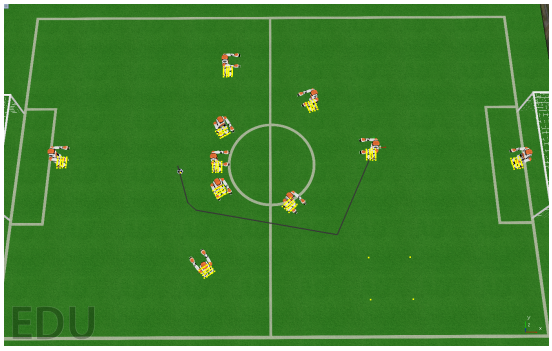
4.1 Planejamento de trajetória

Utilizando a biblioteca OMPL, foram executados três algoritmos de planejamento de trajetória: RRT (árvore aleatória de exploração rápida), RRTConnect e PRM (mapa de rotas probabilístico). O mesmo mapa foi usado para os três planejadores e, para a medição de tempo, foram feitas 50 execuções. O algoritmo PRM obteve o caminho mais curto até a bola e um tempo médio de 0,008 segundos com desvio padrão de 0,002 s (Figura 4.2). Já o algoritmo RRT obteve um caminho maior e com ângulos mais fechados, contudo, seu tempo foi de 0,005 segundos, com desvio padrão igual a 0,002 s. Finalmente, o algoritmo RRTConnect obteve o caminho de tamanho intermediário e o tempo de execução foi de 0,006 segundos, com desvio padrão igual a 0,002 s (Figura 4.1). Para os três casos, os tempos de execução foram similares e todos obtiveram sucesso em encontrar o caminho em todas as execuções. O algoritmo PRM obteve a rota ótima, a mais curta. Dessa forma, o escolhemos como planejador de trajetória.



(a) RRT (b) RRTConect

Figura 4.1 – Trajetória planejada no ambiente de simulação. Fonte: Autora



(a) PRM

Figura 4.2 – Trajetória planejada no ambiente de simulação. Fonte: Autora

4.2 Seguimento de trajetória

4.2.1 Ponto fixo

Primeiramente, fizemos o controlador para que o robô NAO se movimentasse para um ponto específico, desconsiderando a orientação. Para isso, seguimos duas abordagens: a primeira, realizando o controle no espaço de estados (Figura 4.3), e a segunda, restringindo a movimentação no eixo y, referente ao sistema de coordenadas do robô (Figura 4.4).

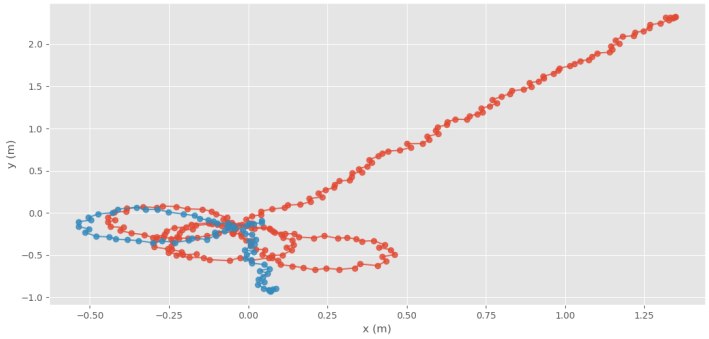


Figura 4.3 – Resultado do controlador com restrição de movimento lateral para diferentes posições iniciais. Fonte: Autora

Podemos observar que o controlador por espaço de estados obteve sucesso em fazer o robô NAO atingir a posição desejada. O controlador que restringe a movimentação lateral, no entanto, não atingiu o regime permanente, em tempo hábil, com erro próximo a zero. Apesar disso, podemos mitigar esse comportamento definindo uma condição de parada. Nesse caso, definimos que, se o erro tivesse valor absoluto menor que 0,03 m, a movimentação estaria concluída. Podemos observar esse comportamento nas figuras 4.5 e 4.6.

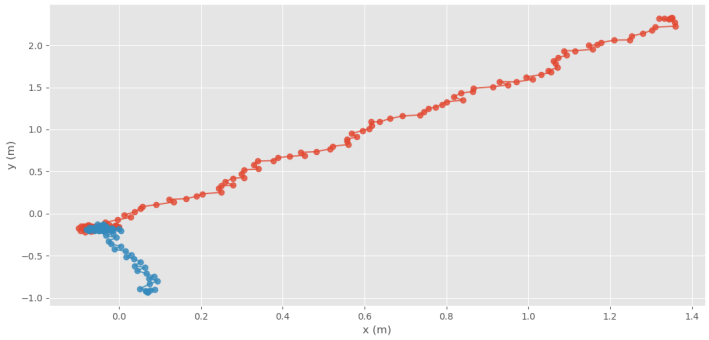


Figura 4.4 – Resultado do controlador no espaço de estados para diferentes posições iniciais. Fonte: Autora

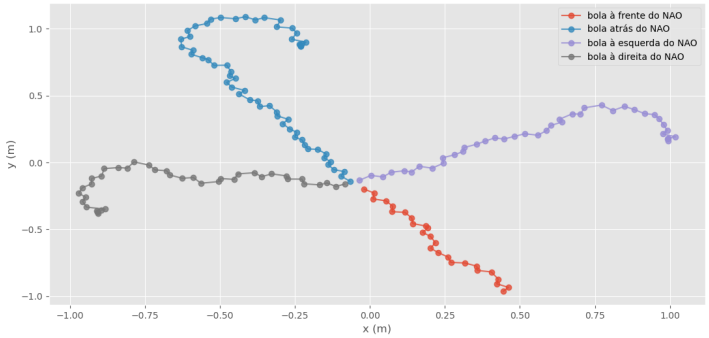


Figura 4.5 – Resultado do controle com restrição de movimento lateral para diferentes posições iniciais com condição de parada. Fonte: Autora

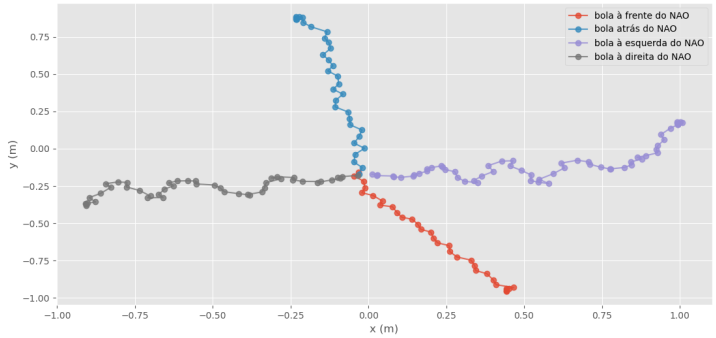


Figura 4.6 – Resultado do controle no espaço de estados para diferentes posições iniciais com condição de parada. Fonte: Autora

Nessa situação, os dois modelos atingiram a posição de destino. Contudo, o modelo de espaço de estados gerou um caminho percorrido com menos desvios, o que, em uma situação de jogo de futebol, é mais interessante, pois grandes desvios de uma trajetória podem gerar colisões.

4.2.2 Trajetória

Para uma mesma trajetória estabelecida, definimos diferentes autovalores a fim de verificar a resposta do sistema com um controlador mais rápido ou mais lento.

- $\lambda = -0{,}5$

  O robô NAO foi capaz de seguir a trajetória (Figura 4.7), contudo, com um pequeno atraso e um desvio de posição em Y, mostrado no gráfico no segundo 35. Esse desvio não é desejado, pois pode gerar colisões.

- $\lambda = -1$

  Para autovalores iguais a -1, o caminho percorrido pelo robô foi mais próximo da trajetória estabelecida (Figura 4.8).

- $\lambda = -5$

  Para autovalor igual a -5, a resposta foi ainda melhor, fazendo com que o NAO fosse capaz de acompanhar a trajetória de forma satisfatória (Figura 4.9). Assim, é possível perceber que, aumentando o módulo do autovalor, o sistema apresenta uma resposta mais rápida e o NAO percorre um caminho mais próximo da trajetóia estabelecida.

- $\lambda = -10$

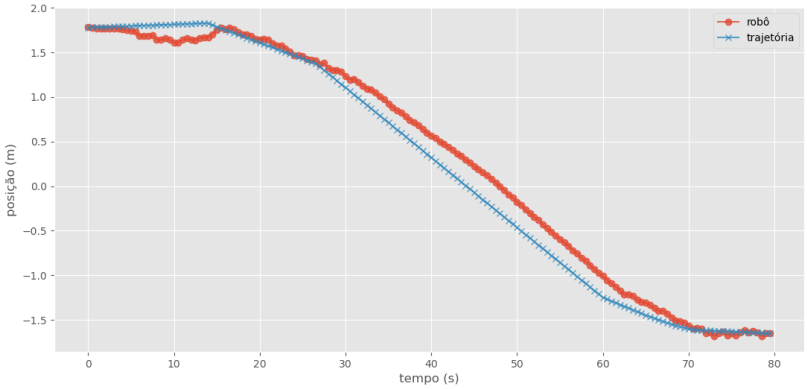
  Para autovalores iguais a -10, a trajetória percorrida pelo NAO apresentou muito pouco desvio em relação à trajetória planjada, inclusive na mudança de direção que ocorre próximo ao segundo 25 (Figura 4.10). Contudo, o robô passou a oscilar mais ao caminhar.

- $\lambda = -30$

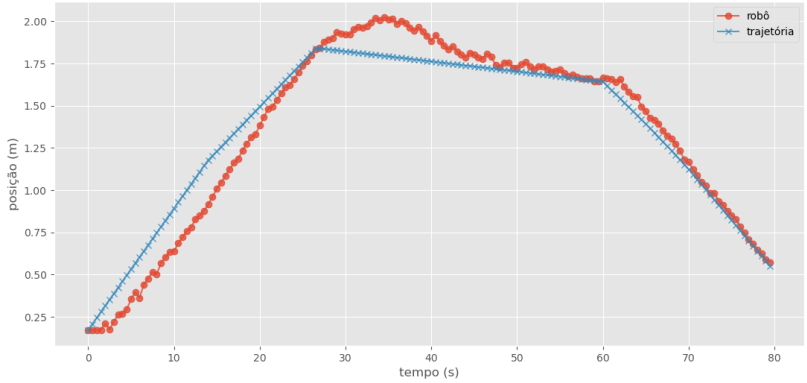
  Para autovales ainda menores, o centro de massa do robô passa a oscilar (Figura 4.11). Uma oscilação acentuada pode gerar uma instabilidade na marcha do robô, podendo fazer com que ele caia durante o movimento e isso, em uma situação de jogo,é indesejado.

  A partir de certo valor de autovalor, o comportamento não altera significativamente, apesar de ainda gerar um pouco mais de oscilação na movimentação. Isso se deve a própria biblioteca NAOqi API que limita a velocidade para garantir a integridade da plataforma NAO. Dessa forma, não consguimos atingir o caso de que a malha externa de controle é mais rápida que a malha interna, que controla os motores responsáveis pela marcha.

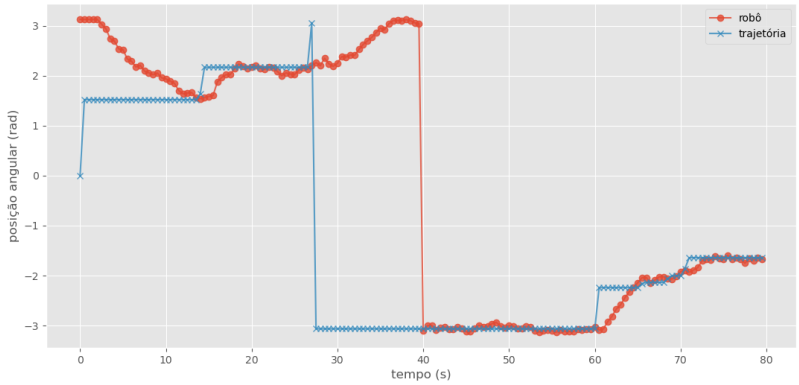
Assim, podemos observar que autovalores próximos a zero geram uma resposta de controle lenta, que pode ocasionar o não seguimento da rota e risco de colisão com os obstáculos. Já autovalores de módulos altos geram instabilidade e são restringidos pela limitação imposta pela biblioteca de marcha para a segurança da integridade dos atuadores do NAO. Dessa forma, observamos que os valores ótimos estão em torno de -5, como observado na figura 4.9.



(a) Posição em x

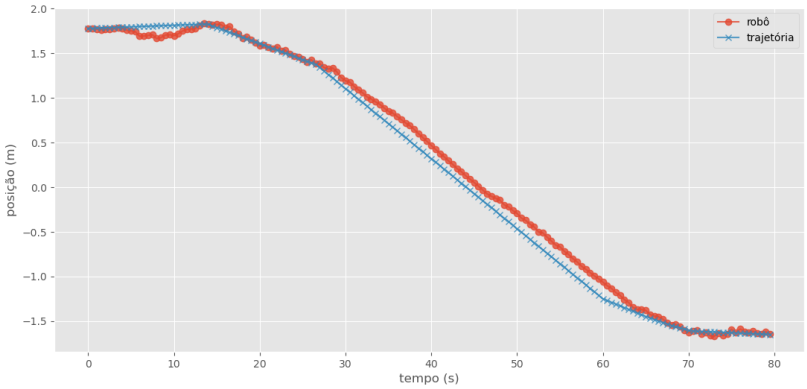


(b) Posição em y

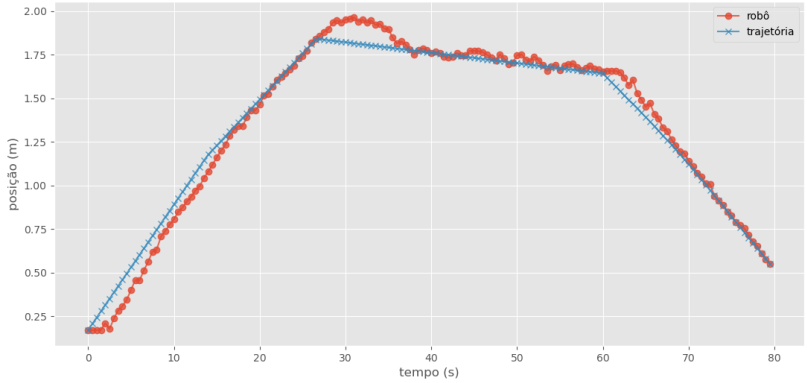


(c) Orientação θ

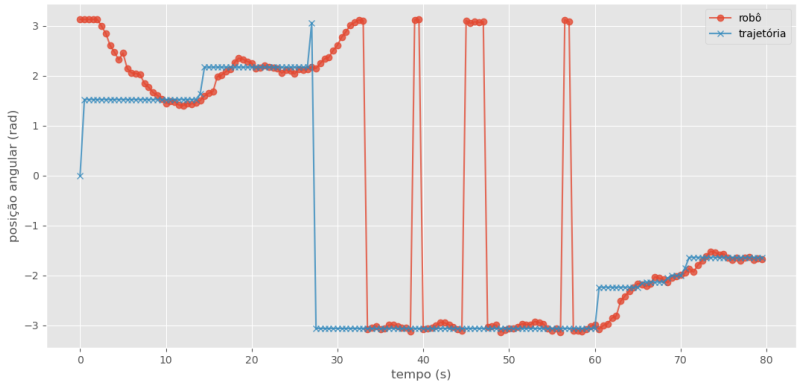
Figura 4.7 – Resposta do sistema para autovalores iguais a λ = -0,5. Fonte:Autora



(a) Posição em x

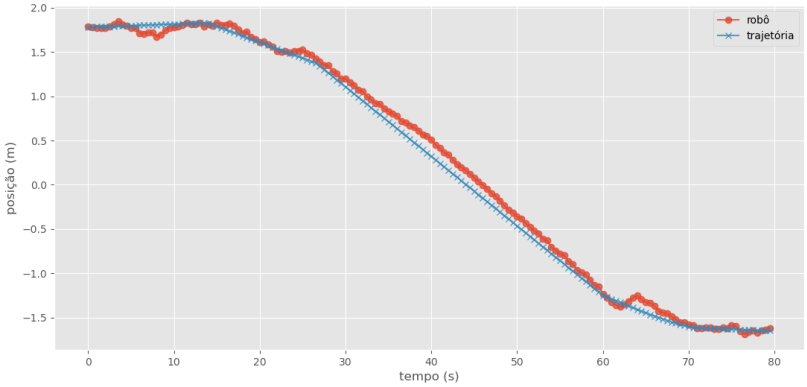


(b) Posição em y

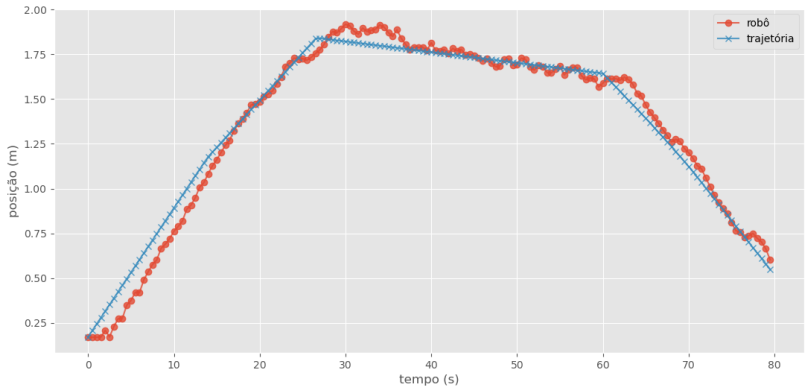


(c) Orientação θ

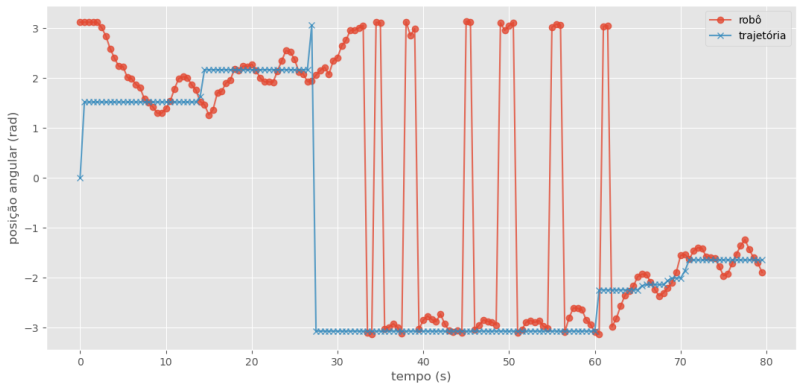
Figura 4.8 – Resposta do sistema para autovalores iguais a λ = -1. Fonte:Autora



(a) Posição em x

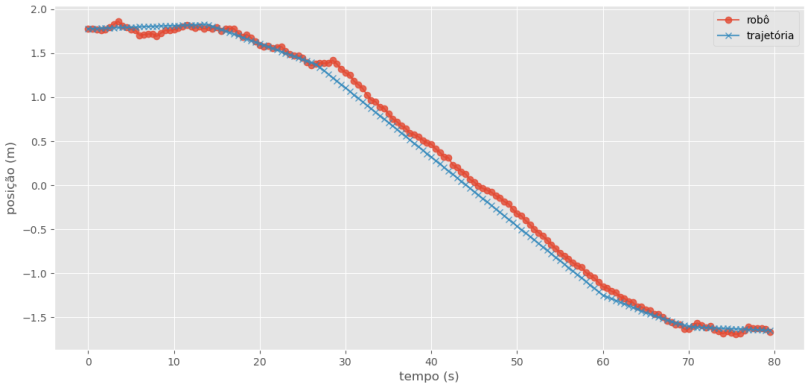


(b) Posição em y

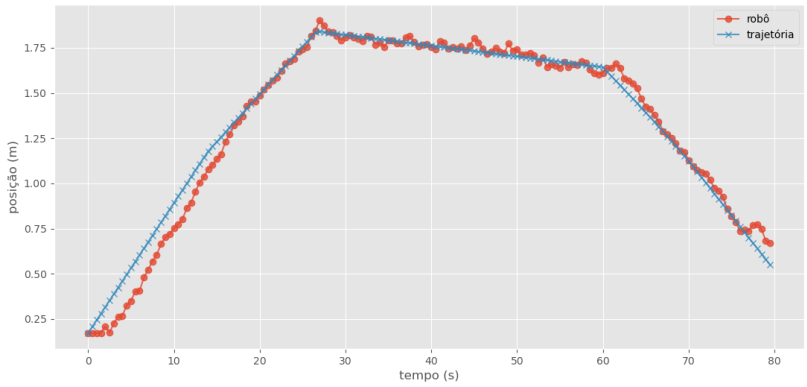


(c) Orientação θ

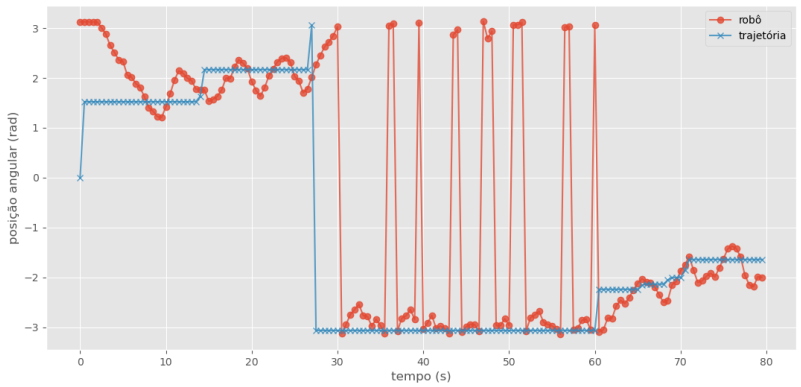
Figura 4.9 – Resposta do sistema para autovalores iguais a λ = -5. Fonte:Autora



(a) Posição em x

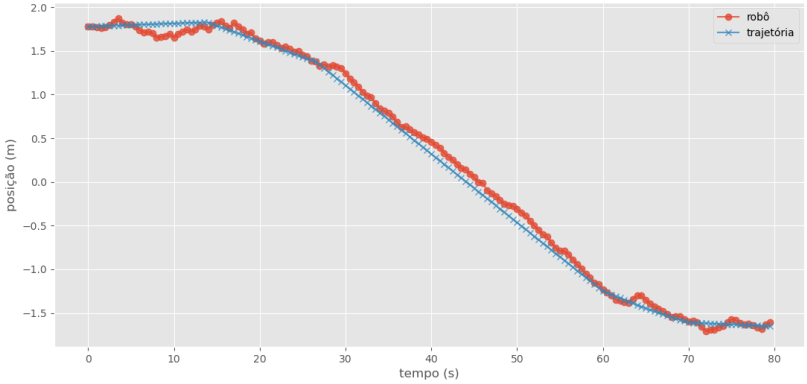


(b) Posição em y

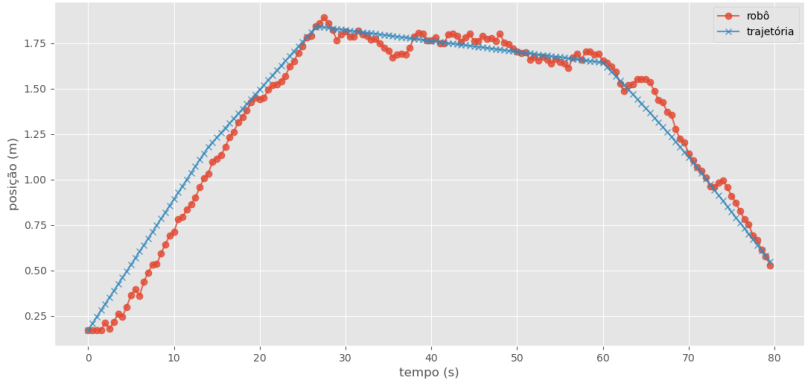


(c) Orientação θ

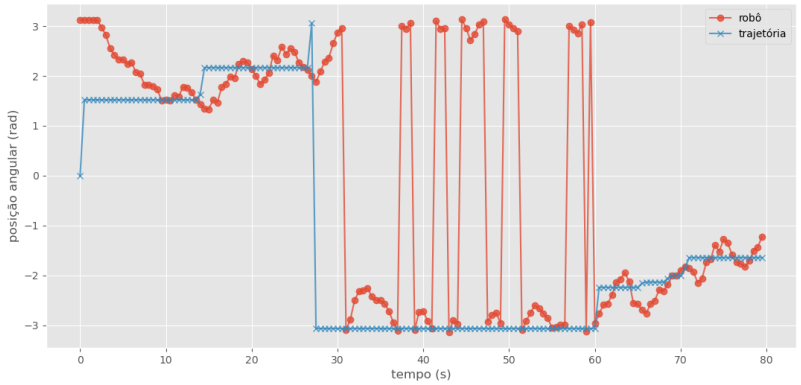
Figura 4.10 – Resposta do sistema para autovalores iguais a λ = -10. Fonte:Autora



(a) Posição em x



(b) Posição em y



(c) Orientação θ

Figura 4.11 – Resposta do sistema para autovalores iguais a λ = -30. Fonte:Autora

5 Conclusões

Esse trabalho consistiu na pesquisa e implementação de planejamento e controle para seguimento de trajetória para a plataforma NAO no contexto de um jogo de futebol da *RObоCup* SPL. Para a etapa de planejamento de trajetória, foram avaliados três planejadores, *RRT*, *RRTConnect* e *PRM*. Era necessário que o planejador escolhido apresentasse robustez e completude em um cenário com vários espaços vazios e tempo de processamento razoável. Todos os três obtiveram sucesso no quesito completude e robustez e obtiveram um tempo de processamento razoável para o cenário. O planejador *PRM*, contudo, foi superior ao planejar uma rota mais curta do que os outros, sendo, então, o escolhido.

Para o seguimento da trajetória, inicialmente foi realizado um controlador para que o NAO fosse para um ponto fixo no plano. Duas abordagens foram apresentadas. A primeira restringindo a movimentação lateral do robô, tendo em vista que a marcha se comporta melhor quando o robô se move para a frente, e a segunda, implementando um controlador no espaço de estados com linearização de Lyapunov. Apesar da possível mitigação do comportamento, o controlador não atingiu um regime permanente com erro de posição próximo a zero. Sendo assim, o controlador no espaço de estados apresentou um resultado mais robusto.

Em seguida, unimos as duas etapas e fizemos o NAO seguir a trajetória planejada, utilizando a abordagem de seguir um ponto em movimento que percorre tal trajetória. Avaliamos o comportamento do sistema para diferentes autovalores e observamos que autovalores próximos a zero geraram uma resposta de controle lenta, o que pode ocasionar o não seguimento da rota e risco de colisão com os obstáculos. Já autovalores de módulos muito altos geraram instabilidade, gerando o risco de queda, e foram restringidos pela limitação imposta pela biblioteca de marcha para a segurança da integridade dos atuadores do NAO. Dessa forma, escolhemos autovalores intermediários, que proporcionaram uma melhor resposta do sistema, conforme apresentado nos experimentos.

5.1 Perspectivas Futuras

Para a evolução desse trabalho, podemos considerar a situação dos outros jogadores em movimento, inclusive com o planejamento de trajetória a partir da posição prevista para os obstáculos e não a atual. Outra possível evolução seria a identificação dos obstáculos a partir dos sensores do NAO, como as câmeras e o sonar. Por fim, seria interessante combinar com um algoritmo de localização e utilizar o que foi apresentado no robô físico em uma situação real de jogo.

Referências

ADORNO, B.; ROCHA, C.; BORGES, G. Planejamento de trajetória para o robô omni utilizando o algoritmo mapa de rotas probabilístico. *In*: . [*S.l.*: *s.n.*], 2005. Citado nas pp. 19 e 23.

CAMPION, G.; BASTIN, G.; DANDREA-NOVEL, B. Structural properties and classification of kinematic and dynamic models of wheeled mobile robots. **IEEE Transactions on Robotics and Automation**, v. 12, n. 1, p. 47–62, 1996. DOI 10.1109/70.481750. Citado na p. 21.

COMMITTEE, R. T. **RoboCup Standard Platform League (NAO) Rule Book**. [*s.d.*]. Disponível em: http://spl.robocup.org/wp-content/uploads/downloads/Rules2019.pdf. Citado na p. 15.

CORKE, P. **Robotics, Vision and Control: Fundamental Algorithms in MATLAB**. 1st. ed. [*S.l.*]: Springer Publishing Company, Incorporated, 2013. ISBN 3642201431. Citado nas pp. 21 e 26.

FAKOOR, M.; KOSARI, A.; JAFARZADEH, M. Humanoid robot path planning with fuzzy markov decision processes. **Journal of Applied Research and Technology**, v. 14, p. 300–310, 10 2016. DOI 10.1016/j.jart.2016.06.006. Citado na p. 19.

FEDERATION, R. **Objective**. [*s.d.*]. Disponível em: http://www.robocup.org/objective. Citado na p. 13.

GOUAILLIER, D.; HUGEL, V.; BLAZEVIC, P.; KILNER, C.; MONCEAUX, J.; LAFOURCADE, P.; MARNIER, B.; SERRE, J.; MAISONNIER, B. Mechatronic design of nao humanoid. *In*: . [*S.l.*: *s.n.*], 2009. p. 769 – 774. DOI 10.1109/ROBOT.2009.5152516. Citado na p. 13.

HARRIS, C. R.; MILLMAN, K. J.; WALT, S. J. van der; GOMMERS, R.; VIRTANEN, P.; COURNAPEAU, D.; WIESER, E.; TAYLOR, J.; BERG, S.; SMITH, N. J.; KERN, R.; PICUS, M.; HOYER, S.; KERKWIJK, M. H. van; BRETT, M.; HALDANE, A.; RÍO, J. F. del; WIEBE, M.; PETERSON, P.; GÉRARD-MARCHANT, P.; SHEPPARD, K.; REDDY, T.; WECKESSER, W.; ABBASI, H.; GOHLKE, C.; OLIPHANT, T. E. Array programming with NumPy. **Nature**, Springer Science and Business Media LLC, v. 585, n. 7825, p. 357–362, set. 2020. DOI 10.1038/s41586-020-2649-2. Disponível em: https://doi.org/10.1038/s41586-020-2649-2. Citado na p. 28.

HUGEL, V.; JOUANDEAU, N. Walking patterns for real time path planning simulation of humanoids. *In*: **2012 IEEE RO-MAN: The 21st IEEE International Symposium on**

Robot and Human Interactive Communication. [*S.l.*: *s.n.*], 2012. p. 424–430. DOI 10.1109/ROMAN.2012.6343789. Citado na p. 19.

KAVRAKI, L.; SVESTKA, P.; LATOMBE, J.-C.; OVERMARS, M. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. **IEEE Transactions on Robotics and Automation**, v. 12, n. 4, p. 566–580, 1996. DOI 10.1109/70.508439. Citado nas pp. 19 e 23.

KHAKSAR, W.; VIVEKANANTHEN, S.; SAHARIA, K. S. M.; YOUSEFI, M.; ISMAIL, F. B. A review on mobile robots motion path planning in unknown environments. *In*: **2015 IEEE International Symposium on Robotics and Intelligent Sensors (IRIS)**. [*S.l.*: *s.n.*], 2015. p. 295–300. DOI 10.1109/IRIS.2015.7451628. Citado na p. 20.

KHALIL, H. K. **Nonlinear systems**. Upper Saddle River, N.J.: Prentice Hall, 2002. ISBN 0130673897 9780130673893 0131227408 9780131227408. Citado na p. 24.

KHOKHAR, A. Probabilistic roadmap (prm) for path planning in robotics. **Medium**, 2025. Acessado em: 16 fev. 2025. Disponível em: https://medium.com/acm-juit/probabilistic-roadmap-prm-for-path-planning-in-robotics-d4f4b69475ea. Citado nas pp. 8 e 24.

KITANO, H.; HAHN, W.; HUNTER, L.; OKA, R.; WAH, B.; YOKOI, T. Grand challenge ai applications. *In*: **IJCAI**. [*S.l.*: *s.n.*], 1993. p. 1677–1683. Citado na p. 12.

KUFFNER, J.; LAVALLE, S. Rrt-connect: An efficient approach to single-query path planning. v. 2, p. 995–1001 vol.2, 2000. DOI 10.1109/ROBOT.2000.844730. Citado nas pp. 8, 22 e 23.

LAVALLE, S. M. Rapidly-exploring random trees: A new tool for path planning. 1998. Citado na p. 22.

LI, D.; XING, J.; SHI, Z.; LIU, Y. Study on path planning of patrol obstacle avoidance based on robot nao. *In*: **2018 2nd IEEE Advanced Information Management,Communicates,Electronic and Automation Control Conference (IMCEC)**. [*S.l.*: *s.n.*], 2018. p. 2190–2193. DOI 10.1109/IMCEC.2018.8469601. Citado na p. 19.

MATARIC, M. J. **The Robotics Primer**. [*S.l.*: *s.n.*], 2011. Citado na p. 12.

OTTONI, G.; FETTER, L. Navegação de robôs móveis em ambientes desconhecidos utilizando sonares de ultra-som. **Sba: Controle Automação Sociedade Brasileira de Automatica**, v. 14, 12 2003. DOI 10.1590/S0103-17592003000400008. Citado na p. 18.

RODRIGUEZ-TIRADO, A.; MAGALLAN-RAMIREZ, D.; MARTINEZ-AGUILAR, J. D.; MORENO-GARCIA, C. F.; BALDERAS, D.; LOPEZ-CAUDANA, E. A pipeline framework for robot maze navigation using computer vision, path planning and communication protocols. *In*: **2020 13th International Conference on Deve-**

lopments in eSystems Engineering (DeSE). [*S.l.*: *s.n.*], 2020. p. 152–157. DOI 10.1109/DeSE51703.2020.9450731. Citado na p. 19.

ROHMER, E.; SINGH, S. P. N.; FREESE, M. Coppeliasim (formerly v-rep): a versatile and scalable robot simulation framework. 2013. www.coppeliarobotics.com. Citado nas pp. 16, 27 e 28.

SICILIANO, B.; SCIAVICCO, L.; VILLANI, L.; ORIOLO, G. **Robotics: Modelling, Planning and Control**. 1st. ed. [*S.l.*]: Springer Publishing Company, Incorporated, 2008. ISBN 1846286417. Citado nas pp. 12, 21 e 22.

ŞUCAN, I. A.; MOLL, M.; KAVRAKI, L. E. The Open Motion Planning Library. **IEEE Robotics & Automation Magazine**, v. 19, n. 4, p. 72–82, December 2012. https://ompl.kavrakilab.org. DOI 10.1109/MRA.2012.2205651. Citado na p. 28.

ZHAI, H.; EGERSTEDT, M.; ZHOU, H. Path planning in unknown environments using optimal transport theory. 09 2019. Citado na p. 19.

Apêndices

Apêndice A – Códigos de programação

A.1 Controlador para movimentação a um ponto fixo

Código A.1 – Código de Python

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-

import math
import signal
import socket
import time

import qi
import b0RemoteApi
import numpy as np

import curses
stdscr = curses.initscr()


keep_running = True
Frequency = 0.5

qsi_time = 0.5

def calculate_angle(dot2, dot1):
    angle = math.atan2((dot2[1][0] - dot1[1][0]),
                       (dot2[0][0] - dot1[0][0]))
    return angle


# Controle para para movimentacao a ponto fixo no espaco de estados
def robot_control():
    session = qi.Session()
    robot_id = 0

    session.connect("localhost:" + str(9600 + robot_id))
    motion_service = session.service("ALMotion")

    robot_position = []

    with b0RemoteApi.RemoteApiClient('b0RemoteApi_NAO_Controller',
        'b0RemoteApiNAO') as sim:
        r, targetHandle = sim.simxGetObjectHandle('Bola',
            sim.simxServiceCall())
        if not r:
            return
```

```
42          r, robotHandle = sim.simxGetObjectHandle('NAO#0',
                sim.simxServiceCall())
43          if not r:
44              return
45
46          autovalue = -1
47          A = np.array([[autovalue, 0, 0],
48                        [0, autovalue, 0],
49                        [0, 0, autovalue]])
50
51          r, target_pos = sim.simxGetObjectPosition(targetHandle, -1,
                sim.simxServiceCall())
52          r, target_ori = sim.simxGetObjectOrientation(targetHandle, -1,
                sim.simxServiceCall())
53
54          r, robotPosition = sim.simxGetObjectPosition(robotHandle, -1,
                sim.simxServiceCall())
55          r, robotOrientation = sim.simxGetObjectOrientation(robotHandle,
                -1, sim.simxServiceCall())
56          qsi_measured = np.array([[robotPosition[0]], [robotPosition[1]],
                [robotOrientation[2]]])
57
58          target_pose = np.array([[target_pos[0]], [target_pos[1]],
                [target_ori[2]]])
59          print('Bola: ', target_pose)
60
61          while True:
62              r, robotPosition = sim.simxGetObjectPosition(robotHandle, -1,
                    sim.simxServiceCall())
63              r, robotOrientation =
                    sim.simxGetObjectOrientation(robotHandle, -1,
                    sim.simxServiceCall())
64
65              R = np.array([[np.cos(robotOrientation[2]),
                    np.sin(robotOrientation[2]), 0],
66                            [-np.sin(robotOrientation[2]),
                                np.cos(robotOrientation[2]), 0],
67                            [0, 0, 1]])
68
69              qsi_measured = np.array([[robotPosition[0]],
                    [robotPosition[1]], [robotOrientation[2]]])
70
71              error = target_pose - qsi_measured
72              error[2][0] = math.atan2(math.sin(error[2][0]),
                    math.cos(error[2][0]))
73
74              q = -A.dot(error)
75              q = R.dot(q)
76
77              motion_service = session.service("ALMotion")
78
79              # Envia as velocidades para a NAOqi
80              try:
```

```
81                  motion_service.move(float(q[0][0]),float(q[1][0]),float(q[2][0]))
82              except:
83                  exit()
84
85              time.sleep(0.5)
86              if not keep_running:
87                  break
88
89              print(qsi_measured)
90
91              # Salva as posicoes em um array para depois colocar no arquivo
92              robot_position.append(qsi_measured.transpose()[0])
93
94              if np.mean(abs(error)) < 0.03 :
95                  break
96
97              # Le uma tecla apertada a qualquer momento
98              x = stdscr.getkey()
99              if x=='a':
100                 break
101             print('\x1b[?25h')
102
103         # Salva a posicao da bola e as posicoes do robo em arquivo
104         np.savetxt('robot.csv', robot_position)
105         np.savetxt('ball.csv', target_pose)
106
107         print('\x1b[?25h')
108
109         try:
110             motion_service.stopMove()
111         except:
112             pass
113
114
115 # Controle para movimentacao a ponto fixo com restricao de movimento
        lateral
116 def robot_control_corke():
117     session = qi.Session()
118     robot_id = 0
119
120     session.connect("localhost:" + str(9600 + robot_id))
121     motion_service = session.service("ALMotion")
122
123     robot_position = []
124
125     with b0RemoteApi.RemoteApiClient('b0RemoteApi_NAO_Controller',
            'b0RemoteApiNAO') as sim:
126         r, targetHandle = sim.simxGetObjectHandle('Bola',
                sim.simxServiceCall())
127         if not r:
128             return
129         r, robotHandle = sim.simxGetObjectHandle('NAO#0',
                sim.simxServiceCall())
```

```
130         if not r:
131             return
132 
133         Kv = 2
134         Kh = 2
135 
136         r, target_pos = sim.simxGetObjectPosition(targetHandle, -1,
                sim.simxServiceCall())
137         r, target_ori = sim.simxGetObjectOrientation(targetHandle, -1,
                sim.simxServiceCall())
138 
139         target_pose = np.array([[target_pos[0]], [target_pos[1]],
                [target_ori[2]]])
140         print('Bola: ', target_pose)
141 
142         while True:
143             r, robotPosition = sim.simxGetObjectPosition(robotHandle, -1,
                    sim.simxServiceCall())
144             r, robotOrientation =
                    sim.simxGetObjectOrientation(robotHandle, -1,
                    sim.simxServiceCall())
145 
146             R = np.array([[np.cos(robotOrientation[2]),
                    np.sin(robotOrientation[2]), 0],
147                           [-np.sin(robotOrientation[2]),
                              np.cos(robotOrientation[2]), 0],
148                           [0, 0, 1]])
149 
150             qsi_measured = np.array([[robotPosition[0]],
                    [robotPosition[1]], [robotOrientation[2]]])
151 
152             vx = Kv*math.sqrt((target_pose[0][0]-qsi_measured[0][0])**2 +
                    (target_pose[1][0]-qsi_measured[1][0])**2)
153             target_theta =
                    math.atan2((target_pose[1][0]-qsi_measured[1][0]),
                    (target_pose[0][0]-qsi_measured[0][0]))
154 
155             error = target_pose - qsi_measured
156             error[2][0] = target_theta - qsi_measured[2][0]
157             error[2][0] = math.atan2(math.sin(error[2][0]),
                    math.cos(error[2][0]))
158             vtheta = Kh*error[2][0]
159 
160             q = np.array([[vx], [0], [vtheta]])
161 
162             motion_service = session.service("ALMotion")
163 
164             # Envia as velocidades para a NAOqi
165             try:
166                 motion_service.move(float(q[0][0]),float(q[1][0]),float(q[2][0]))
167             except:
168                 exit()
169 
```

```
170             time.sleep(0.5)
171             if not keep_running:
172                 break
173 
174             print(qsi_measured)
175 
176             # Salva as posicoes em um array para depois colocar no arquivo
177             robot_position.append(qsi_measured.transpose()[0])
178 
179             if np.mean(abs(error[:2])) < 0.03 :
180                 break
181 
182             # Le uma tecla apertada a qualquer momento
183             x = stdscr.getkey()
184             if x=='a':
185                 break
186             print('\x1b[?25h')
187 
188         # salva a posicao da bola e as posicoes do robo em arquivo
189         np.savetxt('robot.csv', robot_position)
190         np.savetxt('ball.csv', target_pose)
191 
192         # faz com que o que foi digitado volte a aparecer no terminal
193         print('\x1b[?25h')
194 
195         try:
196             motion_service.stopMove()
197         except:
198             pass
199 
200 
201     return
202 
203 
204 def signal_handler(sig, frame):
205     global keep_running
206     keep_running = False
207     return
208 
209 if __name__ == '__main__':
210     robot_control()
211     exit(0)
```

A.2 Controlador para seguimento de trajetória

Código A.2 – Código de Python

```python
#!/usr/bin/env python2
# -*- coding: utf-8 -*-

import math
import signal
import socket
import time

import qi
import b0RemoteApi
import numpy as np


keep_running = True
Frequency = 0.5

def calculate_angle(dot2, dot1):
    angle = math.atan2((dot2[1][0] - dot1[1][0]),
                       (dot2[0][0] - dot1[0][0]))
    return angle


def robot_control(path_array):

    session = qi.Session()
    robot_id = 0

    session.connect("localhost:" + str(9600 + robot_id))
    motion_service = session.service("ALMotion")

    it = iter(path_array)
    path = np.array([[[xp], [yp], [zp]] for xp, yp, zp in zip(it, it, it)])

    step = 5
    time_arr = []
    robot_position = []
    desired_path = []
    qsi_desired = path[0]

    with b0RemoteApi.RemoteApiClient('b0RemoteApi_NAO_Controller',
        'b0RemoteApiNAO') as sim:
        r, targetHandle = sim.simxGetObjectHandle('Bola',
            sim.simxServiceCall())
        if not r:
            return
        r, robotHandle = sim.simxGetObjectHandle('NAO#0',
            sim.simxServiceCall())
        if not r:
            return
```

```
47          r, dummyHandle = sim.simxGetObjectHandle('Start',
                sim.simxServiceCall())
48          if not r:
49              return
50          r, sim_time = sim.simxGetSimulationTime(sim.simxServiceCall())
51          time_arr.append(sim_time)
52
53          autovalue = -1
54          A = np.array([[autovalue, 0, 0],
55                        [0, autovalue, 0],
56                        [0, 0, autovalue]])
57
58          r, target_pos = sim.simxGetObjectPosition(targetHandle, -1,
                sim.simxServiceCall())
59          r, target_ori = sim.simxGetObjectOrientation(targetHandle, -1,
                sim.simxServiceCall())
60
61          target_pose = np.array([[target_pos[0]], [target_pos[1]],
                [target_ori[2]]])
62          print('Bola: ', target_pose)
63
64          for i in range(len(path)//step):
65
66              r, sim_time = sim.simxGetSimulationTime(sim.simxServiceCall())
67              time_arr.append(sim_time)
68              print(sim_time)
69
70              old_qsi_desired = qsi_desired
71              qsi_desired = path[step*i]
72              qsi_desired[2][0] = calculate_angle(qsi_desired,
                    old_qsi_desired)
73              qsi_desired_d = (qsi_desired - old_qsi_desired) /
                    (time_arr[-1] - time_arr[-2])
74
75              pos = [qsi_desired[0][0], qsi_desired[1][0], 0.4]
76              ori = [0,0,qsi_desired[2][0]]
77              sim.simxSetObjectPosition(dummyHandle,-1, pos,
                    sim.simxServiceCall())
78              sim.simxSetObjectOrientation(dummyHandle,-1, ori,
                    sim.simxServiceCall())
79
80              r, robotPosition = sim.simxGetObjectPosition(robotHandle, -1,
                    sim.simxServiceCall())
81              r, robotOrientation =
                    sim.simxGetObjectOrientation(robotHandle, -1,
                    sim.simxServiceCall())
82
83              R = np.array([[np.cos(robotOrientation[2]),
                    np.sin(robotOrientation[2]), 0],
84                            [-np.sin(robotOrientation[2]),
                                np.cos(robotOrientation[2]), 0],
85                            [0, 0, 1]])
86
```

```
87              qsi_measured = np.array([[robotPosition[0]],
                    [robotPosition[1]], [robotOrientation[2]]])
88
89              error = qsi_desired - qsi_measured
90              error[2][0] = math.atan2(math.sin(error[2][0]),
                    math.cos(error[2][0]))
91
92              q = qsi_desired_d - A.dot(error)
93              q = R.dot(q)
94
95              motion_service = session.service("ALMotion")
96              try:
97                  motion_service.move(float(q[0][0]),float(q[1][0]),float(q[2][0]))
98              except:
99                  exit()
100
101             time.sleep(0.5)
102             if not keep_running:
103                 break
104
105             # Salva as posicoes em um array para depois colocar no arquivo
106             robot_position.append(qsi_measured.transpose()[0])
107             desired_path.append(qsi_desired.transpose()[0])
108
109         # salva a posicao da bola e as posicoes do robo em arquivo
110         np.savetxt('robot.csv', robot_position)
111         np.savetxt('ball.csv', target_pose)
112         np.savetxt('tempo.csv', time_arr)
113         np.savetxt('path.csv', desired_path)
114
115         try:
116             motion_service.stopMove()
117         except:
118             pass
119
120
121 def signal_handler(sig, frame):
122     global keep_running
123     keep_running = False
124     return
125
126
127 def receive_data(client_socket):
128     # Recebe o comprimento da string JSON (9 bytes)
129     data_str = client_socket.recv(9)
130     if not data_str:
131         print('Nao chegou nada')
132         return None
133     data_length = int(data_str)
134
135     # Recebe a string JSON
136     client_socket.settimeout(10)
137     data_list = list()
```

```
138     while True:
139         data_str = client_socket.recv(4096).decode('utf-8')
140         if not data_str:
141             break
142         else:
143             data_list.append(data_str)
144 
145     data = ''.join(data_list)
146     if len(data) != data_length:
147         print(len(data), '<>', data_length)
148     return ''.join(data)
149 
150 
151 def main():
152     server_address = (socket.gethostname(), 32000)
153     server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
154     server_socket.bind(server_address)
155     signal.signal(signal.SIGINT | signal.SIGABRT, signal_handler)
156 
157     server_socket.listen(1)
158     print('Aguardando conexao...')
159     client_socket, client_address = server_socket.accept()
160     print('Conexao recebida de', client_address)
161 
162     try:
163         while True:
164             data_str = receive_data(client_socket)
165             if data_str is not None:
166                 path_points = eval(data_str)
167                 break
168     finally:
169         client_socket.close()
170         server_socket.close()
171 
172     # print(path_points)
173     robot_control(path_points)
174     return
175 
176 if __name__ == '__main__':
177     main()
178     exit(0)
```

A.3 Planejador de trajetória

Código A.3 – Código de simulador

```
visualizePath=function(path)
    if not _lineContainer then
        _lineContainer=sim.addDrawingObject(sim.drawing_lines,3,0,-1,99999,{0.2,0.2,0
    end
    sim.addDrawingObjectItem(_lineContainer,nil)
    if path then
        local pc=#path/3
        for i=1,pc-1,1 do
            lineDat={path[(i-1)*3+1],path[(i-1)*3+2],initPos[3],path[i*3+1],path[i*3+2
            sim.addDrawingObjectItem(_lineContainer,lineDat)
        end
    end
end


function sysCall_threadmain()
    dummyHandle=sim.getObjectHandle('Start')
    targetHandle=sim.getObjectHandle('Bola')
    robotHandle=sim.getObjectHandle('NAO#0')
    bigCubeHandle=sim.getObjectHandle('Cuboid3')

    simSock = require('socket')

    robotPos = sim.getObjectPosition(robotHandle,-1)
    sim.setObjectPosition(dummyHandle,-1,robotPos)

    initPos=sim.getObjectPosition(dummyHandle,-1)
    initOrient=sim.getObjectOrientation(dummyHandle,-1)


    task = simOMPL.createTask('task')
    state_space={simOMPL.createStateSpace('3d',simOMPL.StateSpaceType.pose2d,bigCubeHa
    simOMPL.setStateSpace(task, state_space)
    simOMPL.setAlgorithm(task,simOMPL.Algorithm.RRTConnect)
    simOMPL.setCollisionPairs(task,{bigCubeHandle,sim.getCollectionHandle('Obstacles')

    startpos=sim.getObjectPosition(dummyHandle,-1)
    startorient=sim.getObjectOrientation(dummyHandle,-1)
    startpose={startpos[1],startpos[2],startorient[3]}
    simOMPL.setStartState(task,startpose)

    goalpos=sim.getObjectPosition(targetHandle,-1)
    goalorient=sim.getObjectOrientation(targetHandle,-1)
    goalpose={goalpos[1],goalpos[2],goalorient[3]}
    simOMPL.setGoalState(task,goalpose)

    r, path=simOMPL.compute(task,4,-1,800)
    visualizePath(path)

```

```
    json = require('cjson')
    data = json.encode(path)
    size = string.format("%09d", #data)

    repeat
        conexao = simSock.connect('localhost', 32000)
        if conexao then
            conexao.send(conexao, size)
            conexao.send(conexao, data)
            break
        end
    until sim.getSimulationState() == sim.simulation_advancing_abouttostop

end

function sysCall_cleanup()
end
```

Anexos


UnB