UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE TECNOLOGIA E GEOCIÊNCIAS
DEPARTAMENTO DE ENGENHARIA MECÂNICA

CECILIA VIRGINIA SANTOS DA SILVA

IMPLEMENTAÇÃO DE UM CONTROLADOR SEGUIDOR DE ESPIRAL APLICADO À UM ROBÔ DIFERENCIAL PARA EVITAR OBSTÁCULOS ESTÁTICOS E DINÂMICOS

Recife

2022

CECILIA VIRGINIA SANTOS DA SILVA

IMPLEMENTAÇÃO DE UM CONTROLADOR SEGUIDOR DE ESPIRAL APLICADO À UM ROBÔ DIFERENCIAL PARA EVITAR OBSTÁCULOS ESTÁTICOS E DINÂMICOS

Trabalho de Conclusão de Curso submetido ao Departamento de Engenharia Mecânica da Universidade Federal de Pernambuco como requisito parcial para obtenção do título de Bacharel em Engenharia Mecânica.

Orientador: Prof. Dr. Adrien Durand-Petiteville

Recife

2022

Ficha de identificação da obra elaborada pelo autor,
através do programa de geração automática do SIB/UFPE

Silva, Cecilia Virginia Santos da.
Implementação de um controlador seguidor de espiral aplicado à um robô diferencial para evitar obstáculos estáticos e dinâmicos / Cecilia Virginia Santos da Silva. - Recife, 2022.
53p : il., tab.

Orientador(a): Adrien Durand-Petiteville
Trabalho de Conclusão de Curso (Graduação) - Universidade Federal de Pernambuco, Centro de Tecnologia e Geociências, Engenharia Mecânica - Bacharelado, 2022.
Inclui referências, apêndices.

1. Navegação autônoma. 2. Método reativo. 3. Controlador espiral. 4. ROS. 5. Robótica. I. Durand-Petiteville, Adrien. (Orientação). II. Título.

620 CDD (22.ed.)

CECILIA VIRGINIA SANTOS DA SILVA

IMPLEMENTAÇÃO DE UM CONTROLADOR SEGUIDOR DE ESPIRAL APLICADO À UM ROBÔ DIFERENCIAL PARA EVITAR OBSTÁCULOS ESTÁTICOS E DINÂMICOS

Trabalho de Conclusão de Curso apresentado à Coordenação do Curso de Graduação em Engenharia Mecânica da Universidade Federal de Pernambuco, como requisito parcial à obtenção do grau de Engenharia Mecânica.

Aprovado em: 17/05/2022

BANCA EXAMINADORA

_____

Prof. Adrien Durand-Petiteville
Universidade Federal de Pernambuco

_____

Prof. João Paulo Cerquinho Cajueiro
Universidade Federal de Pernambuco

_____

Prof. Hansenclever de França Bassani
Universidade Federal de Pernambuco

RESUMO

A robótica moderna traz inúmeros desafios relacionados as áreas de automação e de navegação. Um dos principais está na área de planejamento automático do movimento que possui duas grandes áreas de abordagem, a primeira baseada na construção de um mapa global e a segunda baseada no controle reativo ou sem mapa. Este trabalho foca nesta segunda abordagem, ou seja, na implementação de um controlador que se beneficia da informação local utilizando sensores para melhor conduzir a navegação. A partir da localização de um objeto, o controlador segue um caminho em espiral ao redor do objeto, corrigindo a própria angulação e mantendo uma distância segura do objeto, evitando colisões. Além disso, também é realizada uma análise sobre critério de classificação do objeto, se é estático ou dinâmico. A implementação é a aplicada à um robô diferencial utilizando framework ROS e o ambiente de simulação do Gazebo.

Palavras-chave: Navegação autônoma. Método reativo. Controlador espiral. ROS. Robótica.

ABSTRACT

Modern robotics brings numerous challenges related to the areas of automation and navigation. One of the main challenges is in the area of automatic motion planning that has two major areas of approach, the first based on the construction of a global map and the second based on reactive or map-less control. This work focuses on this second approach, that is, on implementing a controller that takes advantage of local information using sensors to better drive navigation. From the location of an object, the controller follows a spiral path around the object, correcting its own angulation and maintaining a safe distance from the object, avoiding collisions. Furthermore, an analysis is also performed on the object's identification criteria, whether it is static or dynamic. The implementation is applied to a differential robot using the ROS framework and the Gazebo simulation environment.

Keywords: Autonomous navigation. Reactive method. Spiral controller. ROS. Robotics.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

ROS	*Robot Operating System*
SCD	*Spiral Controller Design*
SCP	*Spiral Center Point*
LiDAR	*Light Detection And Ranging*
IMU	*Inertial Measurement Unit*
DoF	*Degrees of Freedom*
ODE	*Open Dynamics Engine*
DART	*Dynamic Animation and Robotics Toolki*
OGRE	*Object-Oriented Graphics Rendering Engine*

LISTA DE SÍMBOLOS

O_c	Ponto mais próximo detectado
O_b	Ponto que representa o baricentro
O_s	*Spiral Center Point*
α^*	Ângulo de referência
d^*	Distância mínima de referência
α_B	Ângulo de giro da espiral, para dentro ou para fora
$v(t)$	Velocidade linear
$e_B(t)$	Erro entre $\alpha(t)$ e α_B
$\omega(t)$	Velocidade angular
λ_B	Escalar positivo
t_c	momento em que $\alpha(t)$ converge para α_B
λ_S	Escalar positivo
H_{t_i,t_p}	Matriz transformação entre dois quadros
F_w	Quadro global
$F_r(t_i)$	Quadro no tempo atual
$F_r(t_p)$	Quadro no tempo passado
$\hat{d}(t_i, t_p)$	Distância prevista
$r(t_i, t_p)$	Resíduo
δ	Limiar adaptativo

SUMÁRIO

1 INTRODUÇÃO

A robótica moderna tem atingido grandes progressos desde o mundo da manufatura industrial. Nesse período viu-se a aplicação da robótica e automação em diversas areas, como em linhas de montagem e solda na indústria automobilística, no transporte e organização de estoques, limpeza doméstica e até missões de exploração em outro planeta (SIEGWART; NOURBAKHSH; SCARAMUZZA, 2011). Embora os robôs móveis tenham um vasto conjunto de aplicações e mercados, existe um fator que é verdadeiro para praticamente todos os robôs móveis: a sua concepção envolve a integração de muitos corpos de conhecimento diferentes.

Segundo Choset et al. (2005), alguns dos desafios mais significativos que a robótica autônoma móvel enfrenta encontram-se na área do planejamento automático do movimento. O objetivo é ser capaz de especificar uma tarefa em uma linguagem de alto nível e fazer com que o robô compile automaticamente esta especificação em um conjunto de movimentos primitivos de baixo nível, ou controladores com feedback, para realizar a tarefa. A tarefa típica é encontrar um caminho para um robô, seja ele um braço de robô ou um robô móvel, indo de um ponto a outro evitando obstáculos.

A literatura propõe uma grande variedade de abordagens que variam em um amplo espectro. De um dos lados do espectro, uma primeira classe de métodos, chamados abordagens baseadas em mapas, dependem de um mapa do ambiente para planejar uma trajetória que conduza ao objetivo ao mesmo tempo em que se lida com restrições tais como a presença de obstáculos ou a cinemática robótica (SIEGWART; NOURBAKHSH; SCARAMUZZA, 2011). Os desafios destes métodos residem principalmente na construção do mapa e na localização do robô. Primeiro, o mapa tem de ser suficientemente preciso para permitir uma estimativa precisa da posição do robô. Em segundo lugar, tem de conter toda a informação relevante para conduzir a uma trajetória, lidando com restrições, por exemplo, a presença de obstáculos. Assim, estes métodos parecem relevantes para realizar navegação de longo alcance na presença de restrições, mas parecem ser menos adequados para lidar com obstáculos inesperados, apesar das extensões existentes na literatura (ver Minguez, Lamiraux e Laumond (2016)).

Do outro lado do espectro, uma segunda classe de métodos, denominada abordagens reativas ou sem mapa, não utiliza um mapa do ambiente e beneficia da informação local. A maioria destes métodos baseia-se em abordagens baseadas em sensores, no sentido em que a posição atual e o objetivo são definidos em termos de medidas relevantes no espaço do sensor (SIEGWART; NOURBAKHSH; SCARAMUZZA, 2011). Diferentes sensores podem ser considerados, tais como sensores de visão ou sensores de raio laser, levando a diferentes espaços de expressão (BONIN-FONT; ORTIZ; OLIVER, 2008). A navegação é então realizada por um controlador de feedback de saída para fazer desaparecer o erro entre os dados sensorias atuais e os dados sensoriais de referência. Assim, não é necessário qualquer processo de localização

em relação a um quadro global ou a um mapa. O desafio nesta classe de métodos reside na concepção de controladores capazes de garantir a estabilidade em malha fechada, ao mesmo tempo que gerem as diferentes restrições necessárias para a execução da tarefa. Estes métodos parecem relevantes para navegar na presença de obstáculos desconhecidos, mas parecem ser menos adequados para lidar com numerosas restrições.

Este trabalho foca no método reativo, mais especificamente no cálculo de um caminho em espiral ao redor de um obstáculo, no qual não há a necessidade de um conhecimento prévio do mapa global. Através dos valores das distâncias até o objeto, é definido o melhor centro para essa espiral e um controlador ajuda a percorrer esse caminho evitando colisões. A performance do controlador será analisada tanto no ambiente estático como no dinâmico. Vale salientar que este controlador tem como propósito contornar o objeto e que, para uma boa performance de navegação local, ele deve trabalhar associado a outro controlador. O trabalho tem a seguinte divisão: no Capítulo 2 tem-se o Referencial teórico com uma apresentação do ROS e revisão da literatura; no Capítulo 3 é indicado o robô e software de simulação utilizado, bem como uma descrição do fluxo do código; no Capítulo 4 tem-se a descrição e validação dos testes e apresentação dos resultados; e finalizando no Capítulo 5 com a conclusão.

1.1 OBJETIVOS

1.1.1 Objetivo geral

O trabalho tem o objetivo de implementar e testar um controlador seguidor de espiral para o desvio de obstáculos. A abordagem deve ser implementado para um robô diferencial usando o middleware ROS e o simulador Gazebo.

1.1.2 Objetivos específicos

- Criar ambiente de simulação com diferentes tipos de obstáculos.
- Estudo e implementação do cálculo dos possíveis centros para espiral.
- Estudo e implementação de dois controladores reativos.
- Realizar testes com obstáculos estático e dinâmico.

2 REFERENCIAL TEÓRICO

Antes de iniciar o projeto, é importante a revisão teórica de todos os conceitos a serem abordados, de modo a expandir o conhecimento do realizador e facilitar a compreensão do leitor.

2.1 ROBOT OPERATING SYSTEM (ROS)

2.1.1 Apresentação

Nesse trabalho, os vários algoritmos processando os dados e controlando o robô foram implementados usando ROS (Robot Operating System), que é um framework de desenvolvimento de software open-source para aplicações de robótica (QUIGLEY; GERKEY; SMART, 2015). O ROS oferece uma plataforma de software padrão para desenvolvedores em todos os setores, que os transportará desde a pesquisa e prototipagem até a implantação e produção. A concepção do ROS foi feita baseada nos desafios atuais de desenvolvimento em robótica, na qual a grande variedade de hardware e diferentes tipos de robôs dificultava o intercâmbio, e até o reuso, de códigos em diferentes plataformas. Através da criação desse framework, é possível um trabalho integrativo em grande escala em diferentes situações, à medida que os sistemas robóticos se tornam cada vez mais complexos.

A filosofia do ROS é resumida em:

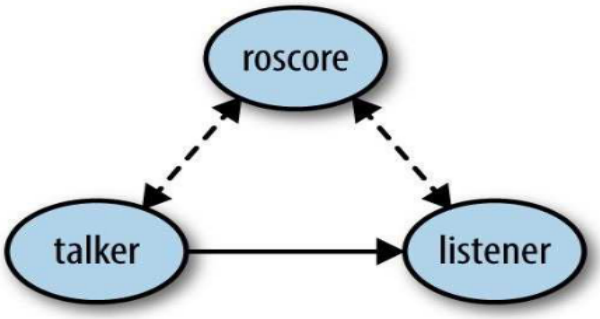
- **Peer-to-peer**: Os sistemas ROS consistem em vários pequenos programas de computador que se conectam e trocam mensagens continuamente. Essas mensagens viajam diretamente de um programa para outro, não há serviço de roteamento central.
- **Tools-based**: Sistemas de software complexos podem ser criados a partir de muitos programas pequenos e genéricos. Esses programas performam diferentes tarefas como navegar na árvore do código-fonte, visualizar as interconexões do sistema, plotar graficamente fluxos de dados, gerar documentação, registrar dados.
- **Multi-lingual**: Cada projeto em ROS pode ser implementado em qualquer linguagem de programação moderna. ROS foi completamente implementado para C++, Python e Lisp, além disso existem bibliotecas experimentais para Java e Lua.
- **Thin**: As convenções de ROS incentivam os colaboradores a criar bibliotecas independentes e, em seguida, agrupar essas bibliotecas para que possam enviar e receber mensagens para, e de, outros módulos do ROS. Essa camada extra destina-se a permitir a reutilização de software fora do ROS para outras aplicações.
- **Free and Open-source**: O núcleo do ROS é liberado sob a licença permissiva BSD (Berkeley Software Distribution), que permite o uso comercial e não comercial.

2.1.2 Nomenclatura

De acordo com Quigley, Gerkey e Smart (2015), as aplicações do ROS são fundamentadas em ambientes robóticos, que podem realizar atividades bastantes complexas, por exemplo, um robô doméstico que reconhece um ambiente usando câmeras e sensores e navega por meio de uma base móvel, buscando objetos com um braço manipulador, requer que todas essas funções funcione simultaneamente. Assim, uma forma bem simples e ilustrativa criada para mostrar as comunicações simultâneas é através do *ROS Graph*. É conveniente o uso de um grafo matemático para representar o fluxo de mensagens, indicando de quem se comunica e o destino da mensagem. O grafo também é intuitivo, visto que o próprio "nó"do grafo é o Nó do sistema ROS, que por sua vez é representa um módulo de software que envia ou recebe mensagens.

Roscore é um serviço que fornece informação de conexão aos nós para que eles possam transmitir mensagens uns aos outros. Cada nó liga-se a *roscore* no momento de inicialização do sistema para registar detalhes dos fluxos de mensagens que publica e dos fluxos a que deseja assinar. Quando surge um novo nó, *roscore* fornece-lhe a informação necessária para formar uma ligação direta *peer-to-peer* com outros nós, publicando e assinando nos mesmos tópicos de mensagens. Cada sistema ROS precisa de um *roscore* em funcionamento, uma vez que, sem ele, os nós não conseguem encontrar outros nós. Apesar da necessidade de funcionamento do *roscore* no sistema ROS, ele não é um mediador da comunicação entre os pares de nós, ele apenas indica quem são os pares de comunicação. Ou seja, *roscore* é como uma Master que sabe quais tipos de mensagens existem, quem publica e quem assina em todo o sistema, porém nenhum fluxo de mensagem passa através dele.

Figura 1 – Conexão momentânea do *Roscore* com os nós.



Fonte: (QUIGLEY; GERKEY; SMART, 2015)

Os nós, de certa forma, só se tornam úteis quando estão conectados e trocando informações com outros nós. A forma mais comum dessa comunicação é por meio de Tópicos. Os Tópicos implementam um mecanismo de comunicação de publicação/assinatura. Antes dos nós começarem a transmitir dados sobre os tópicos, devem primeiro anunciar, ou indicar ao *roscore*, tanto o nome do tópico como os tipos de mensagens que vão ser enviadas. Depois

podem começar a enviar, ou a publicar, os dados reais sobre o tópico. Os nós que desejam receber mensagens sobre um tópico podem subscrever-se a esse tópico, fazendo um pedido ao *roscore*. Após a assinatura, todas as mensagens sobre o tópico são entregues ao nó que fez o pedido.

Os serviços são outra forma de passar dados entre nós no ROS. Os serviços são apenas chamados de procedimento remoto síncrono, ou seja, permitem que um nó chame uma função que executa em outro nó. Definimos as entradas e saídas desta função de forma semelhante à forma como definimos novos tipos de mensagens. Serviços são requisitados geralmente para atividades que ocorrem ocasionalmente ou que levam um tempo limitado.

2.2 SPIRAL CONTROLLER DESIGN (SCD)

O objetivo deste projeto é implementar um método que permita a um robô evitar obstáculos com base em dados fornecidos por um laser. Para isso, propõe-se a utilização de um método de rastreamento em espiral. Nesta seção, apresentamos, portanto, as diferentes ferramentas utilizadas para realizar esta tarefa: o modelo de uma espiral, o controlador que permite o rastreamento de uma espiral, os métodos de cálculo dos diferentes parâmetros da espiral.

2.2.1 Apresentação da espiral

O controlador espiral se baseia em métodos de controladores reativos, ou seja, a movimentação do robô depende de informações locais do ambiente obtidas através de sensores embarcados.

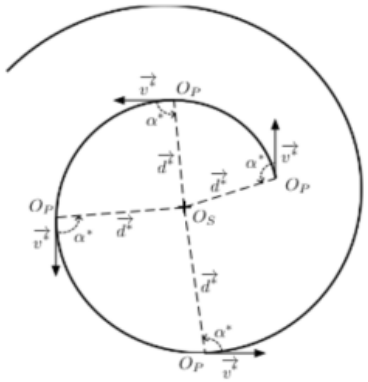
De acordo com Leca et al. (2019), primeiramente determina-se os parâmetros que definem a espiral equiangular, que é uma curva tal que a amplitude do ângulo formado pela tangente em qualquer dos seus pontos com a reta que liga este ponto ao centro, é constante. Como mostrado na Figura 2, a espiral é descrita pelo ponto O_p que se move no plano com relação ao ponto fixo O_s, que é considerado o centro da espiral. O vetor $\overrightarrow{v^*}$ é o vetor velocidade aplicado em O_p e sua norma é denotada por $v^*(t)$. Além disso, $\overrightarrow{d^*}$ é o vetor que conecta O_s a O_p, o qual sua norma é $d^*(t)$. E finalmente, $\alpha^*(t)$ é o ângulo de orientação entre $\overrightarrow{v^*}$ e $\overrightarrow{d^*}$.

Em Boyadzhiev (1999) é mostrado que se ambos $v^*(t)$ e $\alpha^*(t)$ são constantes, então O_p descreve uma espiral na qual O_s é o centro. Assim, eles são então denominados por v^* e α^*. Além disso, Boyadzhiev (1999) fornece uma equação importante relacionada ao $d^*(t)$:

$$\dot{d^*}(t) = -v^* \cos(\alpha^*) \tag{2.1}$$

Analisando a equação anterior percebe-se que a execução da espiral depende apenas do parâmetro α^*, e seu sinal permite definir o sentido de movimento (horário se negativo, anti-horário se positivo), então seu valor determina o tipo de espiral: para dentro se $0 \leq \alpha^* < \pi/2$ ou $0 \leq \alpha^* < -\pi/2$, para fora se $\pi/2 < \alpha^* \leq \pi$ ou $\pi/2 < \alpha^* \leq -\pi$, e um círculo se $\alpha^* = \pm\pi/2$. Por essa análise, o conceito pode ser facilmente adaptado para performar desvios de obstáculos.

Figura 2 – Modelo da espiral.



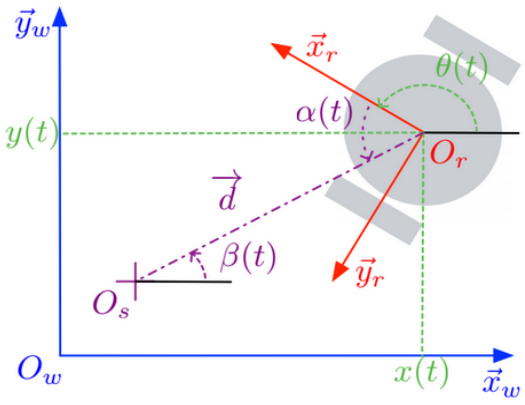
Fonte: Leca et al. (2019)

De fato, fixar o ponto central da espiral (spiral center point - SCP) na superfície do obstáculo e selecionar valores cabíveis para α^* e d^*, permitindo escolher o sentido de movimento do desvio ao redor do obstáculo e então controlar a distância desejada d^* e sua evolução.

2.2.2 Modelo do robô

O modelo de robô utilizado, é definido como ilustrado na Figura 3. Para essa abordagem foi considerado um robô diferencial equipado com um telêmetro a laser e com sensores clássico de localização (odometria).

Figura 3 – Modelo do robô.



Fonte: Leca et al. (2019). Imagem atualizada.

Como indicado na imagem, define-se $F_w = (O_w, \overrightarrow{x_w}, \overrightarrow{y_w}, \overrightarrow{z_w})$ e $F_r = (O_r, \overrightarrow{x_r}, \overrightarrow{y_r}, \overrightarrow{z_r})$ como os quadros atrelados ao mundo e ao robô, respetivamente. A configuração do robô é definida por $X = [x(t), y(t), \theta(t)]^T$, onde $x(t)$ e $y(t)$ são coordenadas de O_r em F_w, e $\theta(t)$ é o

ângulo entre $\overrightarrow{x_r}$ e $\overrightarrow{x_w}$. O_s é o centro da espiral a ser seguida, enquanto $\overrightarrow{d}(t)$ representa o vetor que conecta O_s e O_r, e cuja norma é $d(t)$. Finalmente, $\alpha(t)$ é o ângulo entre $\overrightarrow{x_r}$ e $\overrightarrow{d}(t)$, e $\beta(t)$ é o ângulo entre $\overrightarrow{x_w}$ e $\overrightarrow{d}(t)$. Além disso, as seguintes equações que serão usadas para a síntese do controlador são fornecidas em Boyadzhiev (1999):

$$\alpha(t) = \pi - \theta(t) + \beta(t) \tag{2.2}$$

$$\dot{\alpha}(t) = -\dot{\theta}(t) + \dot{\beta}(t) = -\omega(t) + \frac{v(t)}{d(t)}\sin(\alpha(t)) \tag{2.3}$$

2.2.3 Escolha de parâmetros da espiral para obstáculo estático

Segundo Leca et al. (2019), a escolha dos parâmetros α^*, d^* e SCP são determinantes para o tipo e performance da espiral. São eles que ditam, por exemplo, a distância segura, o sentido e direção de giro da espiral.

2.2.3.1 Escolha do α^*

A escolha do α^* permite definir o modo como o robô se moverá com relação ao obstáculo. Um ângulo menor que $\pi/2$ (espiral para dentro) faz o robô chegar cada vez mais perto do obstáculo, levando a colisão. Um ângulo maior que $\pi/2$ (espiral para fora) não é ideial ou porque faz o robô ir cada vez mais longe do obstáculo ou leva para uma trajetória mais longa. Por isso, é imposto $\alpha^* = \pm\pi/2$, onde o sinal indica o sentido do movimento (pela direita ou pela esquerda).

2.2.3.2 Escolha do d^*

Aqui o valor de d^* é fixado como constante para ser coerente com o valor de α^*. Essa escolha assegura que o robô não se moverá mais perto do que essa distância de qualquer obstáculo detectado.

2.2.3.3 Escolha do SCP

Ainda por Leca et al. (2019), propõe-se a definição de um SCP móvel em vez de um SCP estático como em Futterlieb (2017) e Durand-Petiteville et al. (2017a). O SCP deve ser escolhido de modo a garantir que: (i) a não colisão e a segurança do robô são asseguradas; (ii) obstáculos de várias formas (convexos ou côncavos, com ângulos afiados, etc.) e tamanhos (pequenos ou grandes) são tratados de forma igual; (iii) é garantida uma trajetória suave do robô em torno dos obstáculos com importantes variações de forma, o que significa, por sua vez, que as potenciais partes desconhecidas do obstáculo têm de ser tidas em conta; (iv) a entrada de controle obtida tem de ser adaptada à cinemática do robô. A evolução do SCP depende, assim, do movimento do robô e dos dados do LiDAR (*Light Detection And Ranging*, ou Sistema de Varredura a Laser).

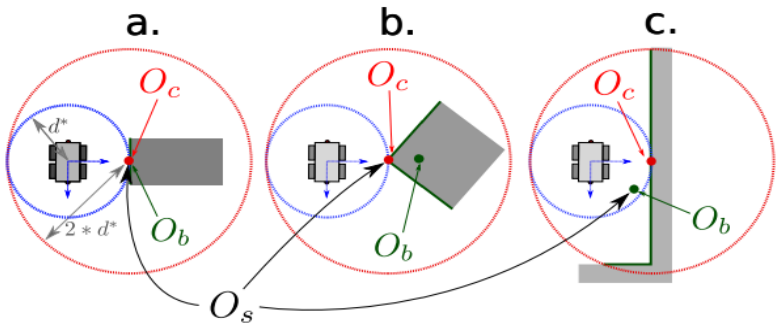
a) *Algoritmo*: O método consiste em computar o ponto central da espiral utilizando estes três passos :

- Passo 1 : Após uma aquisição de dados do LiDAR em torno do robô, o ponto mais próximo do robô O_c é calculado (ver figura 4).

- Passo 2 : Todos os pontos do LiDAR dentro de um raio de $2d^*$ metros, centrado em O_c, são utilizados para calcular o seu baricentro, O_b. Este $2d^*$ de raio é tomado para abranger a distância de segurança em torno do robô (círculos azuis na figura 4).

- Passo 3 : A distância d_b (respectivamente, d_c) entre O_b (respectivamente, O_c) e o robô é computada. O SCP O_s é dado pelo ponto mais próximo do robô, ou seja, o mais próximo do robô: O_b se $d_b < d_c$ e O_c se $d_b \geq d_c$.

Figura 4 – Computação do SCP para diferentes tipos de obstáculos: (a.) Obstáculo retilíneo. - (b.) Obstáculo convexo. - (c.) Obstáculo côncavo.



Fonte: (LECA et al., 2019)

b) *Cálculo do SCP para diferentes obstáculos*: A figura 4 mostra como o SCP é calculado para diferentes tipos de obstáculos durante a prevenção. Em verde são exibidos os pontos do sensor LiDAR, enquanto os círculos azul e vermelho, respectivamente, materializa a distância de segurança do robô e o círculo de raio $2d^*$ metros centrado em O_c que é utilizado para calcular o baricentro. Relativamente ao caso (a.), o algoritmo calcula primeiro O_c que é a projeção ortogonal do robô sobre a parede. Segundo, O_b é computado como o baricentro de todos os pontos que se encontram dentro do círculo vermelho. Neste caso, porque apenas um segmento é percebido, O_b coincide com O_c, o que leva a: $O_s = O_b = O_c$. No caso (b.), O_c está na quina do um obstáculo quadrangular (convexo). O baricentro O_b encontra-se então dentro da forma do obstáculo. Consequentemente, $O_s = O_c$. No caso (c.), de forma semelhante a (a.), O_c é obtido na parede. Os pontos verdes que se encontram no círculo vermelho são então considerados para deduzir o baricentro e são utilizados para calcular ponto O_b. Sendo este último mais próximo do robô do que O_c, segue-se que $O_s = O_b$. Assim, em caso de obstáculos retilíneo ou convexo, $O_s = O_c$ e O_s segue os limites do obstáculo. Quanto aos obstáculos côncavos, $O_s = O_b$ e O_s está no espaço livre. Esta propriedade antecipa o próximo formato côncavo do obstáculo, permitindo

evitar mínimos locais.

2.2.4 Controlador

Em Durand-Petiteville et al. (2017b), dois controladores são sintetizados para realizar o rastreamento de uma espiral. O primeiro permite o acompanhamento de uma espiral definida apenas por α^*, enquanto o segundo realiza o acompanhamento de uma espiral definida por α^* e d^*. Nesta seção apresentamos os dois controladores.

2.2.4.1 Primeiro controlador

A primeira solução consiste em conceber um controlador que faça convergir $\alpha(t)$ para um ângulo desejado chamado $\alpha_B = \alpha^*$. O α_B determina se a espiral é uma espiral para dentro, uma espiral para fora ou um círculo. Uma vez que o robô tenha convergido para a espiral, ou aumenta, ou diminui ou mantém constante $d(t)$. A fim de conceber um controlador, impõe-se uma velocidade linear constante, como $v(t) = v^*$. Assim define-se um erro $e_B(t)$ e é calculado $\dot{e}_B(t)$:

$$e_B(t) = \alpha(t) - \alpha_B \tag{2.4}$$

Usando 2.4 e 2.3, obtemos:

$$\dot{e}_B(t) = \dot{\alpha}(t) = -\omega(t) + \frac{v^*}{d(t)} \sin(\alpha(t)) \tag{2.5}$$

Como se pode ver na equação 2.5, $\dot{e}_B(t)$ depende tanto de v^* como de $\omega(t)$. Sendo a velocidade linear constante, é então necessário compensar o seu efeito enquanto se controla o robô usando apenas $\omega(t)$. Além disso, para fazer desaparecer o erro $e_B(t)$, foi proposto a definição de um controlador não-linear. Assim, segue-se isso:

$$\omega(t) = \lambda_B e_B(t) + \frac{v(t)}{d(t)} \sin(\alpha(t)) \tag{2.6}$$

onde λ_B é um escalar positivo.

O controlador na equação 2.6 permite controlar o comportamento do robô, ou seja, aumentar, diminuir ou manter constante a distância $d(t)$. No entanto, não permite seguir uma espiral específica. De fato, existe uma infinidade de espirais definidas por α_B e para definir uma espiral específica é obrigatório selecionar tanto o ângulo α_B como a distância inicial $d^*(0)$. Ao utilizar o controlador na equação 2.6, $d^*(0)$ não é escolhido e corresponde a $d(t_c)$, onde t_c é o momento em que $\alpha(t)$ converge para α_B.

2.2.4.2 Segundo controlador

Assim, a fim de seguir uma espiral específica, foi imposta mais uma vez uma velocidade linear constante, como $v(t) = v^*$. Então, a definição do erro é dada por:

$$e_S(t) = \alpha(t) - \alpha_S(t) = \alpha(t) - \alpha_B - \alpha_D \epsilon(t) \tag{2.7}$$

Com o intuito de fazer desaparecer o novo erro $e_S(t)$, a referência constante α_B foi substituída por uma referência não constante, $\alpha_S(t) = \alpha_B + \alpha_D\epsilon(t)$. O erro de desaparecimento na equação 2.7 deve levar a dois comportamentos robotizados sucessivos. Primeiro, enquanto $d^*(t) \neq d(t)$, o robô tem de navegar em direção à espiral a fim de fazer $d^*(t) = d(t)$. Para tal, o novo ângulo de referência $\alpha_S(t)$ é igual a α_B modificado pelo montante $\alpha_D\epsilon(t)$. α_D é o valor máximo que pode ser adicionado/subtraído a α_B para fazer o robô convergir para a espiral sem alterar o seu sentido de rotação em relação ao centro da espiral (sentido horário ou anti-horário).

Então $\epsilon(t)$ deve ter a sua norma entre 0 e 1 quando $||d(t) - d^*(t)|| \geq ||d(0) - d^*(0)||$, e igual a 0 quando $d(t) = d^*(t)$. Assim, quando o robô está longe da espiral, converge para ele utilizando o maior ângulo possível, ou seja $\alpha_S(t) = 0$ ou $\alpha_S(t) = \pm\pi$, dependendo das condições iniciais. Além disso, quando $d(t) = d^*(t)$, então $\alpha_S(t) = \alpha_B$. Assim, o robô segue a espiral, mantendo a distância desejada. Esta última parte corresponde ao segundo comportamento esperado ao controlar o robô através do desaparecimento do $e_S(t)$. Para obter os comportamentos anteriormente explicados, foi proposta a definição de $\epsilon(t)$ e α_D como:

$$\epsilon(t) = sign(d^*(t) - d(t))min\left(||\frac{d^*(t)-d(t)}{d^*(0)-d(0)}||, 1\right) \tag{2.8}$$

e

$$\alpha_D = \begin{cases} sign(\alpha_B)\pi - \alpha_B; & \text{se } d^*(0) > d(0) \\ \alpha_B; & \text{se } d^*(0) < d(0) \end{cases} \tag{2.9}$$

$\epsilon(t)$ é o erro normalizado entre $d^*(t)$ e $d(t)$. Foi saturado a ± 1. Assim, o seu valor pertence ao domínio $[0, 1]$ se $d^*(0) > d(0)$ ou $[-1, 0]$ se $d^*(0) < d(0)$. Assim, $\alpha_S \neq \alpha_B$ quando $d^*(t) \neq d(t)$, enquanto $\alpha_S = \alpha_B$ quando $d^*(t) = d(t)$. Quanto à equação 2.9, quando $\alpha_B \in [0, \pi]$ então é obrigatório que $\alpha_S \in [0, \pi]$. Caso contrário, quando $\alpha_B \in [0, -\pi]$, então $\alpha_S \in [0, -\pi]$. As equações 2.8 e 2.9 garantem que a direção de rotação do robô não é modificada por $\alpha_D\epsilon(t)$, permitindo ao mesmo tempo que o robô convirja para $d^*(t)$. A fim de identificar os termos envolvidos na sua dinâmica, calcula-se a primeira derivada em relação ao tempo desse novo erro.

$$\dot{e}_S(t) = \dot{\alpha}(t) - \alpha_D\dot{\epsilon}(t) = -\omega(t) + \frac{v(t)}{d(t)}\sin(\alpha(t)) - \alpha_D\dot{\epsilon}(t) \tag{2.10}$$

A fim de fazer desaparecer o $e_S(t)$, foi proposto a concepção de um controlador não linear a ele que também compense os outros termos envolvidos na equação 2.10, dado por:

$$\omega(t) = \lambda_S e_S(t) + \frac{v(t)}{d(t)}\sin(\alpha(t)) - \alpha_D\dot{\epsilon}(t) \tag{2.11}$$

onde λ_S é um escalar positivo. O controlador 2.11 torna possível que o robô convirja para qualquer espiral específica.

2.2.5 Mudança de parâmetros da espiral em situações de obstáculo dinâmico

Para situações dinâmicas deve-se novamente definir os parâmetros principais da espiral, o SCP, o ângulo α^* e a distância d^*. Assim como em casos estáticos, o SCP foi escolhido como o ponto mais próximo entre o robô e o obstáculo detectado pelos lasers, enquanto o ângulo α^* foi fixado a um valor constante igual a $\pi/2$. A principal diferença surge com a definição de d^*. É necessário considerar um d^* variável. Assim, (RENAK et al., 2019) propõe calcular dinamicamente o valor de d^* graças ao valor residual entre as distâncias previstas e medidas entre o robô e o SCP. A partir deste residual, é calculado um limiar adaptativo (SHI et al., 2005), permitindo discriminar os obstáculos estáticos e dinâmicos e atualizar adequadamente d^*, dependendo do caso.

2.2.5.1 Computação do valor residual

Para calcular o resíduo, o primeiro passo é prever a distância entre o robô e o obstáculo no instante atual, utilizando informações anteriores do laser. Para afirmar o problema, considera-se duas varreduras do laser. A primeira é adquirida no momento atual t_i e é assinalada por $P(t_i, F_r(t_i))$, dados medidos no instante t_i e expressos no quadro do robô no instante t_i. Permite determinar o ponto central da espiral $O_c(t_i)$ (escolhido como o ponto mais próximo detectado na superfície do obstáculo) e para extrair as medidas atuais da distância $d(t_i)$ entre O_r e $O_c(t_i)$ e o ângulo $\alpha(t_i)$. A segunda varredura do laser denotada por $P(t_p, F_r(t_p))$ é obtida em um tempo anterior, $t_p < t_i$. O objetivo é utilizar os dados do laser no instante t_p para prever o valor da distância entre o robô e o SCP no instante t_i e compará-lo com o seu valor $d(t_i)$ medido acima. Para este fim, calcula-se primeiro $P(t_p, F_r(t_i))$ que denota o conjunto de pontos que seriam percebidos no tempo t_i se apenas o robô tivesse estado em movimento durante o intervalo de tempo $[t_p, t_i]$. Para tal, a matriz de transformação homogênea entre dois quadros H_{t_i,t_p} é calculada utilizando os dados fornecidos pelos sensores propriocetivos (i.e. odometria, IMU).

$$H_{t_i,t_p} = H_{i,w} H_{w,p} \tag{2.12}$$

onde $H_{i,w}$ e $H_{w,p}$ são respectivamente as matrizes homogêneas de transformação entre F_w e $F_r(t_i)$, e entre $F_r(t_p)$ e F_w, ou seja,

$$H_{t_i,t_p} = \begin{bmatrix} \cos(\theta(t_i)) & -\sin(\theta(t_i)) & 0 & x(t_i) \\ \sin(\theta(t_i)) & \cos(\theta(t_i)) & 0 & y(t_i) \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}^{-1} \begin{bmatrix} \cos(\theta(t_p)) & -\sin(\theta(t_p)) & 0 & x(t_p) \\ \sin(\theta(t_p)) & \cos(\theta(t_p)) & 0 & y(t_p) \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.13}$$

onde, x e y são as coodrenadanas do robô em F_w e θ sua orientação.

Assim, as coordenadas previstas dos pontos do laser da aquisição anterior no quadro do robô atual podem ser calculados da seguinte forma:

$$P(t_p, F_r(t_i)) = H_{t_i,t_p} P(t_p, F_r(t_p)) \tag{2.14}$$

A partir de $P(t_p, F_r(t_i))$, é possível calcular o ponto mais próximo previsto $\hat{O}_c(t_i, t_p)$, e deduzir a distância prevista $\hat{d}(t_i, t_p)$. A partir deste último, propomos definir o residual $r(t_i, t_p)$ como segue-se:

$$r(t_i, t_p) = \frac{d(t_i) - \hat{d}(t_i, t_p)}{t_i - t_p} \tag{2.15}$$

Este valor residual ajudará a decidir se o obstáculo é estático ou não. De fato, no primeiro caso, $r(t_i, t_p)$ será próximo de 0, enquanto que no segundo caso é esperado um valor maior. No entanto, poderá não ser suficiente para obter uma discriminação robusta, por causa de vários fenômenos como o ruído de medição, um movimento demasiado pequeno do obstáculo móvel, erros de modelização, etc. Neste contexto, a definição de um limiar adaptativo pode ser interessante para classificar claramente os objetos dinâmicos e estáticos.

2.2.5.2 Definição do limiar adaptativo e da distância adaptativa d^*

Procedimentos de limiar adaptativo resumidos em Emami-Naeini, Akhter e Rock (1988) ou Ding e Frank (1992) são geralmente utilizados para detectar falhas do sistema e dependem de um estudo estatístico do sinal residual. Ver, por exemplo Wang, Kruger e Lennox (2003) e Shi et al. (2005). Neste caso, o objetivo é decidir se o obstáculo encontrado é estático ou dinâmico, apesar dos fenômenos acima mencionados. Em Shi et al. (2005), um limiar adaptativo é calculado a partir da média e variância do sinal residual para detectar falhas em uma instalação electro-hidráulica não linear. Seguindo um raciocínio semelhante, foi calculada a média e a variância do resíduo, considerando a sua variação nos tempos q da amostra:

$$\eta(t_i) = \frac{1}{q} \sum_{j=1}^{q} r(t_i, t_{i-j}) \tag{2.16}$$

$$\sigma^2(t_i) = \frac{1}{q-1} \sum_{j=1}^{q} (r(t_i, t_{i-j}) - \eta(t_i))^2 \tag{2.17}$$

A partir das equações acima, o limiar adaptativo δ no instante t_i é calculado da seguinte forma (SHI et al., 2005):

$$\delta(t_i) = \eta(t_i) \pm 2.17\sigma(t_i) \tag{2.18}$$

em que o coeficiente 2.17 corresponde a um nível de confiança de 97% . Assim, se $r(t_i)$ pertence ao intervalo $[\delta^+(t_i), \delta^-(t_i)]$, onde $\delta^+(t_i) = \eta(t_i) + 2.17\sigma(t_i)$ e $\delta^-(t_i) = \eta(t_i) - 2.17\sigma(t_i)$, então o obstáculo é considerado estático. Caso contrário, é esperado um obstáculo dinâmico.

Além disso, o limite superior do limiar adaptativo pode ser utilizado para ajustar a distância de referência d^* como se segue:

$$d^*(t_i) = d_{nominal} + \delta(t_i) \tag{2.19}$$

onde $d_{nominal}$ é uma distância adequada que deseja-se manter em um ambiente com obstáculo estático. Isto leva a um comportamento robótico mais seguro. Se um obstáculo em movimento for em direção ao robô, a cada iteração, a distância real $d(t_i)$ será menor do que a prevista $\hat{d}(t_i, t_p)$. Assim, o residual $r(t_i, t_p)$ vai começar a aumentar, assim como o $\delta(t_i)$. Consequentemente, a distância desejada $d^*(t_i)$ aumentará, e o robô evitará o obstáculo a uma distância maior do que se o obstáculo tivesse sido estático. Nota-se que, nesta utilização particular do limiar adaptativo, pode ocorrer que $d^*(t_i)$ caia e atinja valores que podem ser perigosos para o robô. Para evitar tais problemas, foi acrescentado um limite mínimo correspondente à distância mais baixa aceitável.

3 METODOLOGIA

Para a metodologia, serão abordados os meios utilizados para a ralização do trabalho, como o tipo de robô utilizado e suas características, o simulador escolhido com suas vantagens e desvantagens, e da descriçaõ da implementação com ROS.

3.1 ROBÔ TIAGO PAL ROBOTICS

TIAGo é um robô de serviço produzido para operar em ambientes internos. Ele herda toda a tecnologia e robustez resultantes de anos de desenvolvimento e uso extensivo de outros robôs da PAL Robotics. A empresa tem integrado todos os seus softwares em ROS, além de fornecer documentação e suporte abrangentes para ajudar os pesquisadores a começar a trabalhar com os robôs e obter resultados em um curto espaço de tempo (ROBOTICS, 2021).

Figura 5 – Descrição do Robô TIAGo.

Fonte: (ROBOTICS, 2021)

TIAGo combina mobilidade, percepção, manipulação e capacidades de interação humano-robô. Como pode-se ver na Figura 5, o TIAGo é uma plataforma robótica que simula alguns movimentos humanos, pesa 70 Kg, diâmetro da basa de 54 cm, altura de 110 cm e um total de 12 DoF (Degrees of Fredom), onde 2 estão na base, 7 no braço, 2 na cabeça e 1 no torso. Sua base diferencial alcança velocidade máxima de 1 m/s e opera em ambientes internos. O torso possui deslocamento vertical podendo chegar a uma altura de até 145 cm. Além disso, tem-se também a mobilidade da cabeça e um braço (podendo ter dedos ou garra) para realização de manipulação.

Já os sensores, o robô possui na base um laser com alcance angular de -1.92 rad até 1.92 rad (-110° até 110°) em um raio de até 25 metros, uma IMU com 6 DoF que permite obter feedback de movimento, posição e orientação do robô com relação à um ponto global de referência. Também tem-se uma câmera RGB-D localizada na cabeça, um alto-falante e microfone estéreo para emissão e captara de áudio. Além disso, ele possui autonomia energética de 4-5 horas com uma bateria e 8-10 com duas baterias, e a carga é fornecida em tempo real por meio de um display LCD (ROBOTICS, 2021).

3.2 SIMULADOR GAZEBO

Como apresentado por Koenig e Howard (2004), o Gazebo foi elaborado para reproduzir com precisão os ambientes dinâmicos que um robô pode encontrar. Todos os objetos simulados têm massa, velocidade, fricção, e numerosos outros atributos que lhes permitem comportar-se realisticamente quando empurrados, puxados, derrubados, ou transportados.

Os próprios robôs são estruturas dinâmicas compostas por corpos rígidos ligados através de juntas. Forças, tanto angulares como lineares, podem ser aplicadas às superfícies e juntas para gerar locomoção e interação com um ambiente. O mundo em si pode ser descrito por paisagens, edifícios extrudidos e outros objetos criados pelo utilizador. Quase todos os aspectos da simulação são controláveis, desde condições de iluminação à coeficientes de fricção.

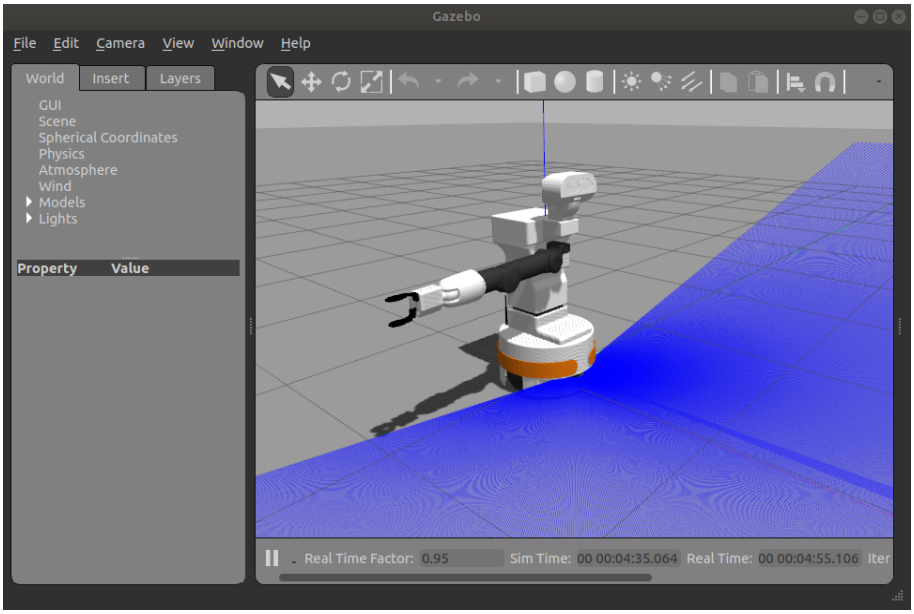
O Gazebo é completamente open-source e encontra-se disponível gratuitamente, assim, ele possui uma base ativa de contribuidores que rapidamente evoluem o simulador de acordo com suas necessidades.

No processo de simulação dinâmica, o Gazebo pode ter acesso aos motores de física de alto desempenho como o Open Dynamics Engine (ODE) (SMITH, 2022), Bullet (SIMULATION, 2022), Simbody (SIMBODY, 2022) e Dynamic Animation and Robotics Toolkit (DART) (LAB; LAB, 2022) que são utilizados para a simulação física rígida do corpo. Object-Oriented Graphics Rendering Engine (OGRE) (OGRE3D, 2022) fornece a renderização gráfica 3D dos ambientes no Gazebo.

Sistemas robóticos complexos que envolvem interação, levantamento e agarramento de objetos e outras tarefas que requerem localização e mapeamento simultâneos podem ser simulados com os poderosos motores de física do Gazebo, usando cenários muito mais realistas e maior grau de confiabilidade. De acordo com a revista IEEE Spectrum (ACKERMAN, 2016), o Gazebo é reconhecido por muitos especialistas como o melhor simulador robótico devido (i) à capacidade de simulação precisa, (ii) flexibilidade extrema, (iii) ter quatro engenhos físicos diferentes, (iv) grande integração com ROS e (v) uma grande e ativa comunidade de colaboradores.

Diante da impossibilidade na realização de testes com o robô real, o Gazebo se torna uma ótima escolha para validação da proposta. A sua capacidade de simulação precisa faz com que a presença de sensores no simulador seja bastante similar ao real, simulando até os ruídos

Figura 6 – Robô TIAGo em ambiente simulado no Gazebo.



Fonte: Autor

das medições dos sensores. Logo, o trabalho elaborado para o ambiente de simulação poderia ser utilizado no ambiente real sem precisar de grandes alterações, apenas alguns ajustes.

3.3 APRESENTAÇÃO DO GRAFO ROS

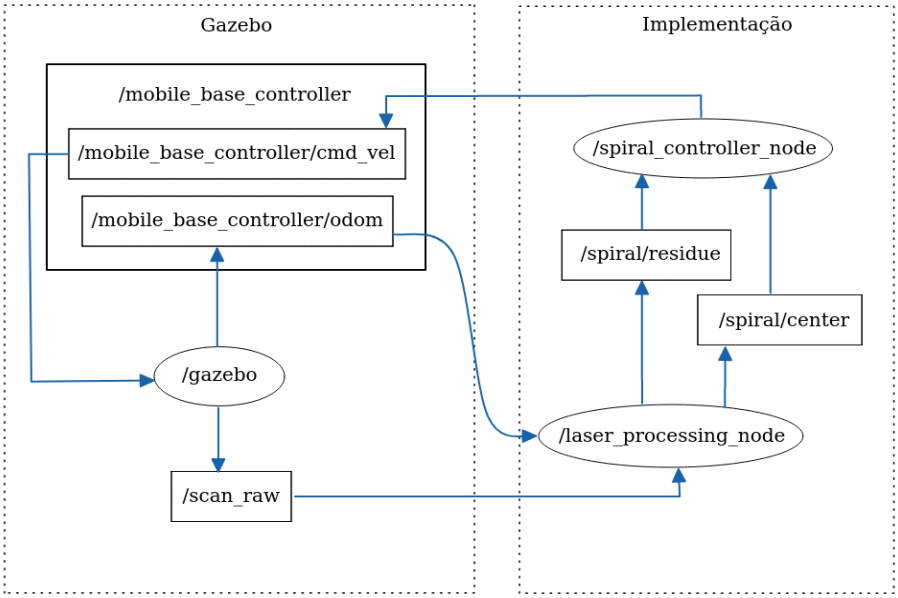
Um sistema construído em ROS consiste de um conjunto de processos que podem ser executados em diferentes ambientes. A comunicação desses processos se dá através de publicações e assinaturas entre os nós, tópicos e o *roscore* para envio e recebimento de mensagens ou solicitação de serviços, por exemplo. A representação dessa rede de comunicação é mostrada por meio de grafos, neles são indicados quais os tipos de relação entre cada nó e cada tópico, quem está recebendo e quem está enviando mensagens.

Para o grafo, os processos são representados da seguinte forma: os tópicos são ilustrados por retângulos, os nós por círculos e as setas indicam o sentido de fluxo da mensagem. É possível obter o grafo através do nó *rqt_graph*.

A Figura 7 ilustra o fluxo de comunicação do projeto, onde tem-se apenas alguns dos tópicos e nós, já existentes na simulação, que interagem diretamente com os tópicos e nós criados para a implementação. Os demais pontos da rede do grafo são aqueles gerados para pleno funcionamento do robô TIAGo.

Primeiramente, nesta figura, temos o nó chamado */gazebo* que representa o ambiente de simulação onde a implementação foi realizada. Este nó faz publicações em basicamente todos os tópicos dos sensores, neste caso o */scan_raw* onde são publicados os dados do laser

Figura 7 – Grafo de comunicação ROS no Gazebo.



Fonte: Autor

e o */mobile_base_controller/odom* onde são publicados os dados de odometria. A partir disso, foi criado um nó chamado */laser_processing_node* em que são calculados o centro O_s da espiral, o estado do obstáculo e os resíduos da espiral, a partir de dados da odometria e do laser, vindos da assinatura nos respectivos tópicos. Essas informações do nó */laser_processing_node* são publicadas nos tópico */spiral/center* e */spiral/residue*. Em seguida foi criado o nó */spiral_controller_node* que assina os tópicos */spiral/center* e */spiral/residue* e utiliza as informações dessas mensagens para calcular a velocidade linear e angular do robô que será publicada no tópico */mobile_base_controller/cmd_vel*. E finalmente, o nó */gazebo* assina o tópico */mobile_base_controller/cmd_vel* para rodar no ambiente de simulação.

3.4 DESCRIÇÃO DO FLUXO DE CÓDIGO

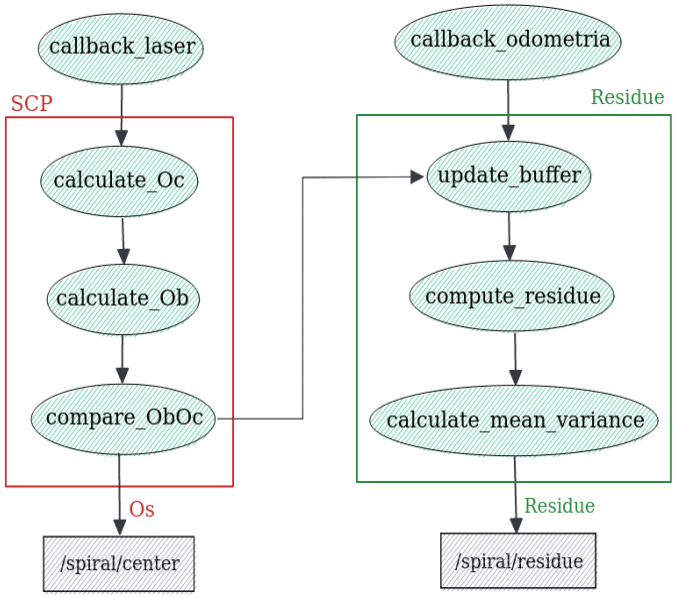
A implementação do algoritmo de desvio de obstáculos foi realizada através da criação de dois nós principais, */laser_processing_node* e */spiral_controller_node*, onde na primeira são realizadas as leituras dos dados do laser e da odometria, a partir deles são calculados o centro da espiral, bem como o estado do obstáculo e os resíduos; e no segundo nó tem-se o controle e a correção da velocidade angular baseados nas mensagens lidas dos tópicos */spiral/center* e */spiral/residue*.

3.4.1 Nó */laser_processing_node*

Na inicialização, tem-se a definição de algumas variáveis para coordenadas, *callbacks*, matrizes para os buffers e, principalmente, são ditos quais tópicos serão publicados ou assinados.

São assinados os tópicos */scan_raw* e */mobile_base_controler/odom*, para leitura das mensagens vindas do laser e da odometria respectivamente, e são publicadas mensagens nos tópicos */spiral/center* e */spiral/residue* criados.

Figura 8 – Fluxograma do Nó */laser_processing_node*



Fonte: Autor

Na função *callback* do laser, chamada cada vez que o nó recebe dados via tópico, são armazenadas os dados de todos os feixes do laser, no caso a distância até um obstáculo em cada ponto do laser. Também é feita a filtragem dos dados, visto que o simulador também gera ruídos assim como no ambiente real.

Com os dados do frame atual filtrados é calculado o centro da espiral, escolhido entre o O_c e o O_b. A função *calculate_Ob* calcula a distância e a ângulação do baricentro entre todos os pontos do objeto lidos pelo laser. Já a função *calculate_Oc* calcula a distância e a ângulação do ponto do obstáculo que esteja mais próximo ao robô. Assim, a função *compare_OcOb* compara entre as distâncias de O_c e O_b qual é a menor e a define como O_s, como explicado na seção 2.2.3.3. Assim, o O_s (distância e ângulo) é publicado no tópico */spiral/center*.

No *callback* da odometria, as orientações do robô obtidas estão na forma de quartenios e são transformadas para ângulos de Euler para obter a orientação θ do eixo vertical do robô. Essas orientações são adicionadas em um buffer, de tamanho N, juntamente com a leitura do laser do mesmo instante e também o valor do próprio instante de tempo.

Em *compute_residue* tem-se o cálculo dos resíduos de distância e ângulo do O_s em N instantes passados, onde N é o tamanho do buffer de armazenamento dos últimos dados do laser. Primeiro, é calculada a matriz homogênea de transformação do instante passado para o instante atual a partir das matrizes homogêneas de transformação entre o quadro do instante atual e

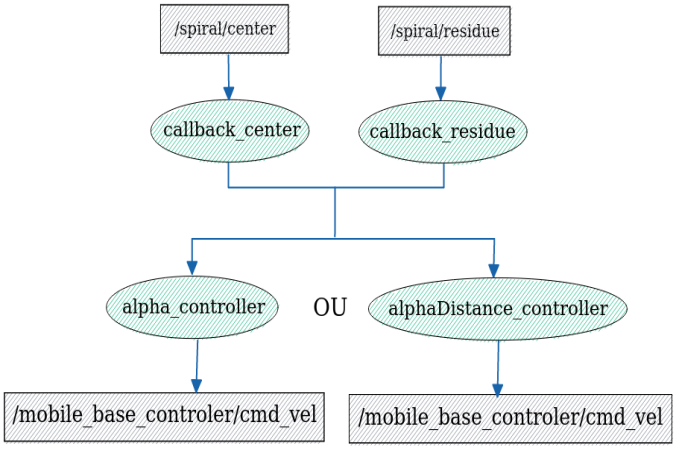
quadro global, e o quadro do instante passado e quadro global, através de dados passado e atual da odometria (ver Equação 2.12 e Equação 2.13). Com a matriz homogênea de transformação e os dados do laser do instante passado, faz-se a projeção desses dados para o instante atual (Equação 2.14. Para cada projeção é calculado o centro O_s e o resíduo (Equação 2.15).

A partir dos resíduos calculados, *calculate_mean_variance* determina a média e a variância dos resíduos da distância e do ângulo. Por fim, calcula-se o limiar adaptativo δ para os resíduos (Equação 2.18) e também determina o status do obstáculo (se é estático ou dinâmico) a partir dos valores da média dos resíduos da distância e do ângulo, publicando no tópico */spiral/residue*.

3.4.2 Nó */spiral_controller_node*

Para este nó, tem-se a definição do tópico que será publicado, */mobile_base_controller/cmd_vel* onde são enviados os valores de velocidade, e os tópicos que serão assinados, */spiral/center* e */spiral/residue* onde são lidos os valores do centro O_s do obstáculo e dos resíduos.

Figura 9 – Fluxograma do Nó */spiral_controller_node*



Fonte: Autor

No *callback_center* do centro da espiral, serão recebidas mensagens enviadas do tópico */spiral/center*, que são os valores de distância e do ângulo do centro O_s atual escolhido. Já no *callback_residue* são recebidas mensagens sobre o status do obstáculo, se é estático ou dinâmico, e os valores do limiar adaptativo δ para os resíduos.

A função *alpha_controller* corresponde a implementação do primeiro controlador, onde é corrigido apenas o valor da velocidade angular. O cálculo depende da distância e do ângulo de O_s, da velocidade linear, do erro e_b e das constantes α_b e λ_b (Equação 2.6).

Já a função *alphaDistance_controller* corresponde a implementação do segundo controlador, que corrige o valor da velocidade angular e também da distância do robô ao obstáculo,

essa distância deve ser maior que o limite mínimo definido. O cálculo depende da proximidade do robô e o obstáculo, o tipo de obstáculo (estático ou dinâmico), o erro e_s de correção do ângulo e do limiar adaptativo δ (Equação 2.11 e 2.19).

4 RESULTADOS

Os testes da abordagem de desvios de obstáculo foram realizados de forma virtual através do simulador Gazebo. Foram criados vários ambientes (mundos) com obstáculos de dimensões e formas diferentes no simulador para poder validar a implementação.

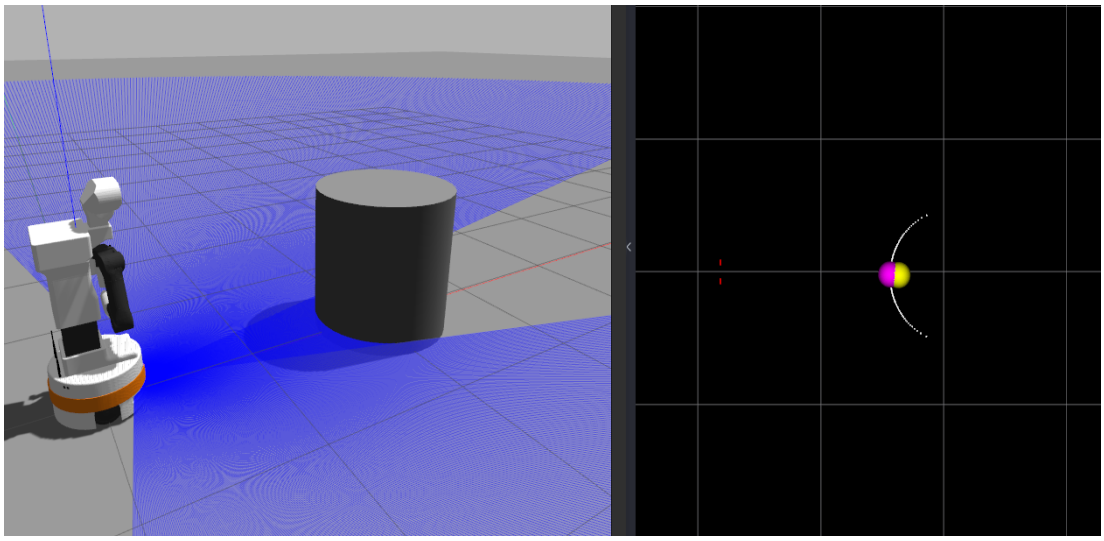
4.1 CÁLCULO DOS CENTROS O_b E O_c

Os primeiros testes realizados foram os de cálculo dos centros, observando se os pontos obtidos estavam como esperado. Como explicado anteriormente (seção 2.2.3.3), o O_b representa o baricentro dos pontos do objeto captados pelo laser e o O_c é o ponto mais próximo dentre os pontos captados pelo laser.

As figuras a seguir mostram os testes para diferentes objetos, onde na grade a direita da figura tem-se o contorno do objeto, detectado pelo laser, e os pontos amarelo que indica o O_b e o rosa que indica o O_c. E a esquerda o ambiente de simulação com o robô e o objeto.

Na Figura 10 tem-se um objeto convexo, no caso um cilindro, posicionado centralizado em frente ao robô, ilustrando um caso em que o centro O_c é mais próximo que o centro O_b que está dentro do objeto.

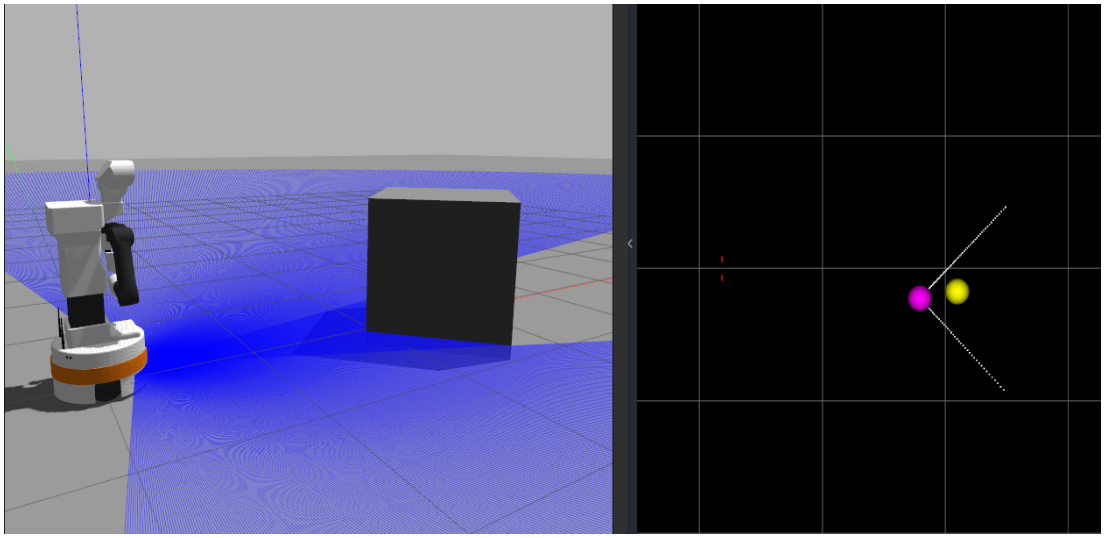
Figura 10 – Cálculo dos pontos O_b (ponto amarelo) e O_c (ponto rosa) com um cilindro.



Fonte: Autor

Na Figura 11 tem-se um objeto convexo não curvo, no caso um cubo, posicionado centralizado em frente ao robô, este melhor ilustra um caso em que o centro O_c é mais próximo que o centro O_b que está dentro do objeto.

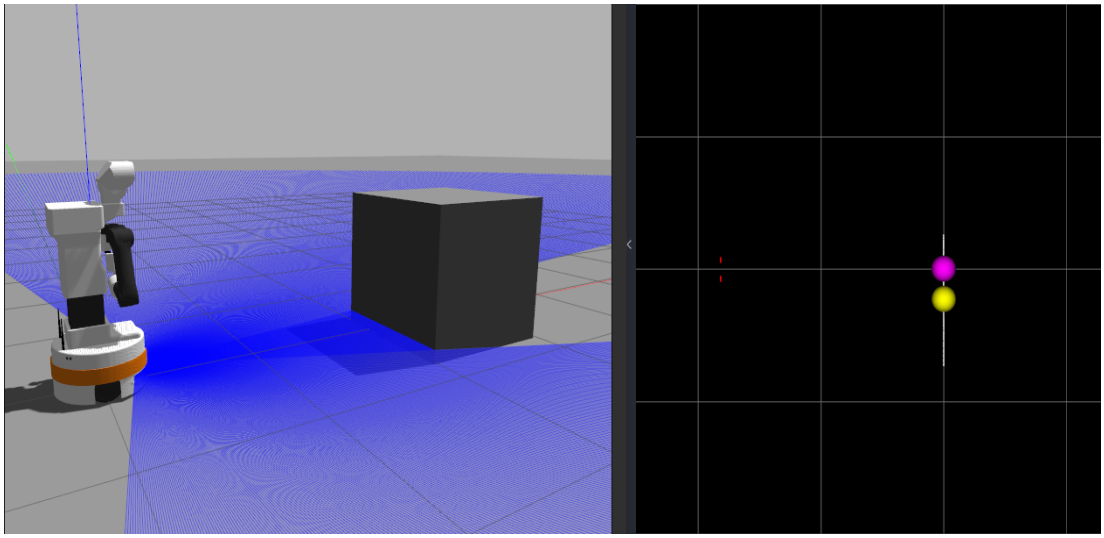
Figura 11 – Cálculo dos pontos O_b (ponto amarelo) e O_c (ponto rosa) com uma aresta.



Fonte: Autor

Na Figura 12 tem-se uma superfície plana, como uma parede, posicionada a direita em frente ao robô, que ilustra um caso em que os centros O_c e O_b estão na superfície do objeto. Neste caso, dependendo da posição do robô, os dois centros podem coincidir.

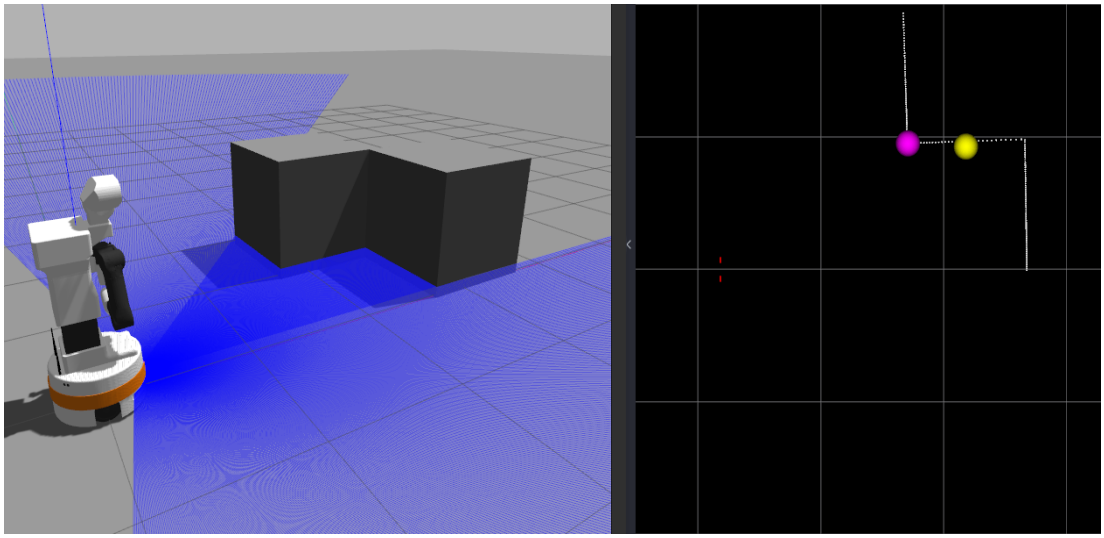
Figura 12 – Cálculo dos pontos O_b (ponto amarelo) e O_c (ponto rosa) com uma parede.



Fonte: Autor

Na Figura 13 tem-se um objeto côncavo não curvo, como um canto interno, posicionados a esquerda em frente ao robô, ilustrando também um caso em que o centro O_c é mais próximo que o centro O_b.

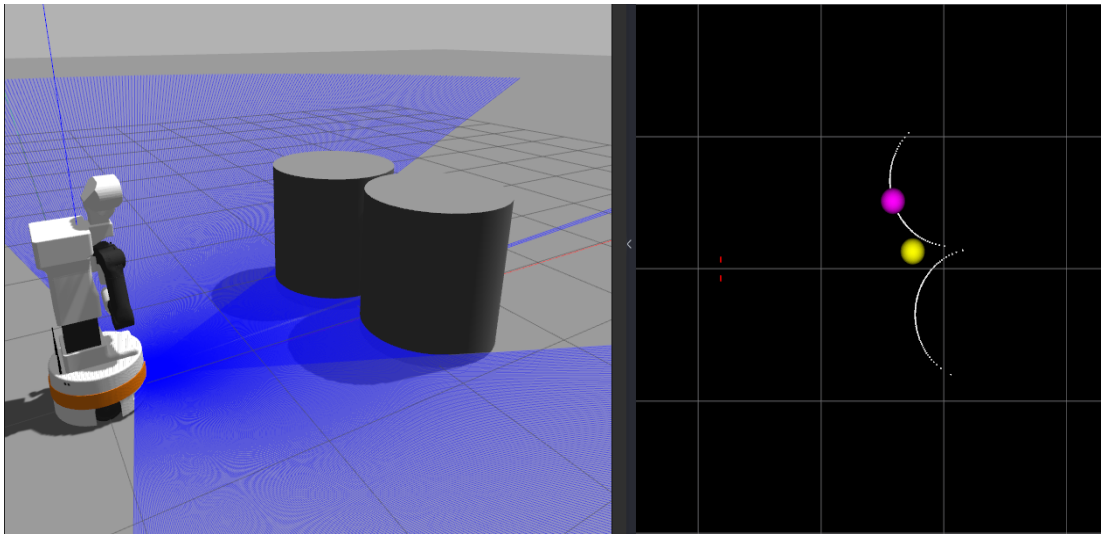
Figura 13 – Cálculo dos pontos O_b (ponto amarelo) e O_c (ponto rosa) com duas paredes.



Fonte: Autor

Já na Figura 14 tem-se dois objetos convexos, no caso dois cilindros, posicionados em frente ao robô, este ilustra um caso em que o centro O_b é um ponto fora dos objetos, mas que o centro O_c ainda é o mais próximo.

Figura 14 – Cálculo dos pontos O_b (ponto amarelo) e O_c (ponto rosa) com dois cilindros.



Fonte: Autor

4.2 PLOT DOS CAMINHOS

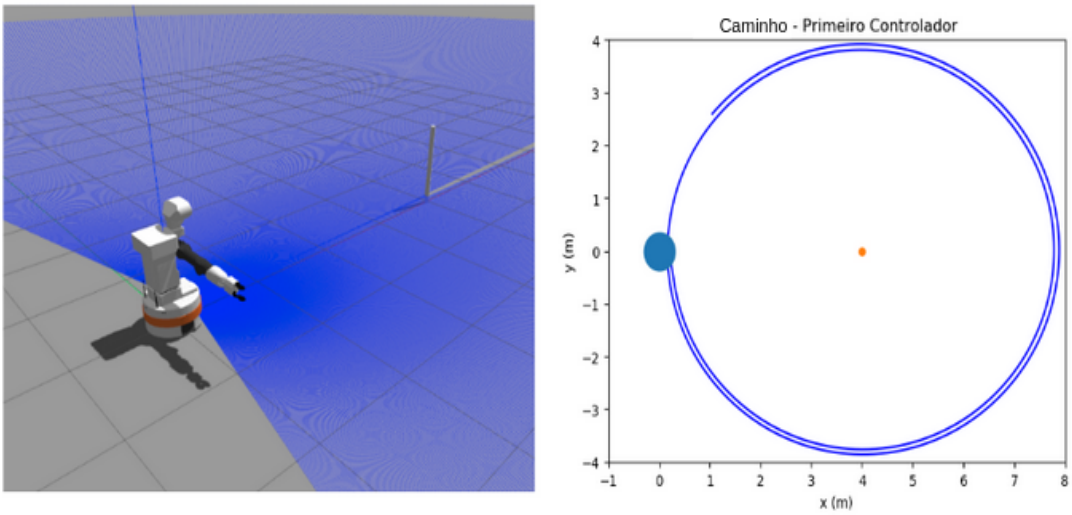
Aqui serão mostrados os testes de performance de ambos os controladores em um ambiente com obstáculo estático ou dinâmico. Foram utilizados dois tipos de obstáculos, um cilindro e um cubo, posicionados a 4m de distância do robô.

4.2.1 Ambiente Estático

O primeiro controlador tem o foco apenas em corrigir o ângulo do robô para que ele siga o caminho da espiral criada, independentemente da sua distância ao obstáculo. As Figuras 15 e 16 ilustram bem a trajetória realizada para $\alpha_B = \pi/2$.

Na Figura 15 tem-se à esquerda a imagem do ambiente simulado com o robô e um cilindro muito fino e à direita a trajetória ao redor dele formando uma espiral com raio quase constante, mas que aumenta com o tempo.

Figura 15 – Caminho espiral do primeiro controlador ao redor de um cilindro.

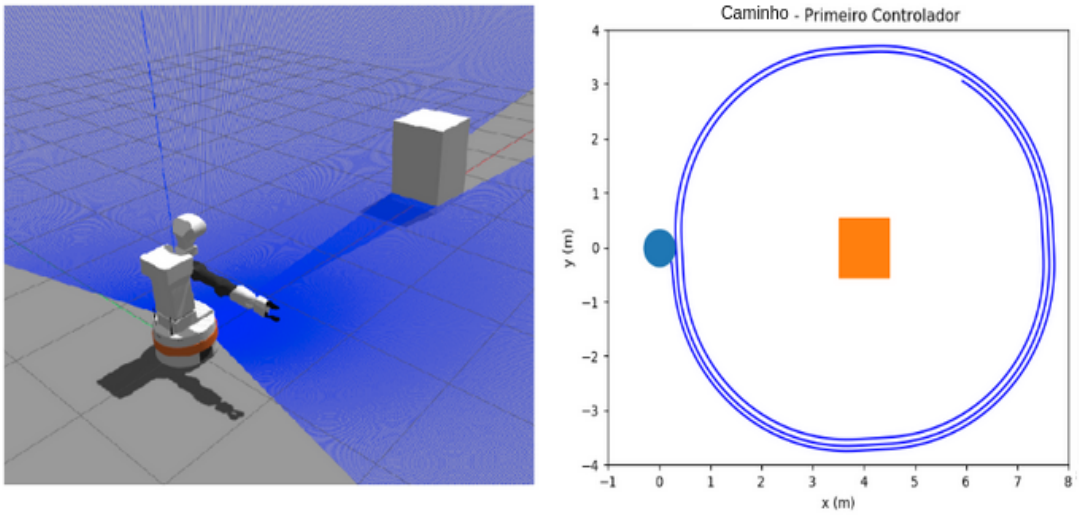


Fonte: Autor

Na Figura 16 tem-se à esquerda a imagem do ambiente simulado com o robô e um cubo com dimensões maiores que o próprio robô. E à direita a trajetória ao redor dele formando uma espiral com raio quase constante, mas que diminui com o tempo, podendo eventualmente colidir com o objeto.

Em ambos os casos, o robô não ajusta a distância ao obstáculo e passa a seguir a espiral a partir de sua posição inicial. Além disso, para ambos os casos, a distância ao obstáculo não é constante apesar do uso de uma referência angular $\alpha_B = \pi/2$. Isso ocorre porque o obstáculo não é um único ponto e o centro da espiral calculado é ligeiramente diferente a cada iteração.

Figura 16 – Caminho espiral do primeiro controlador ao redor de um cubo.

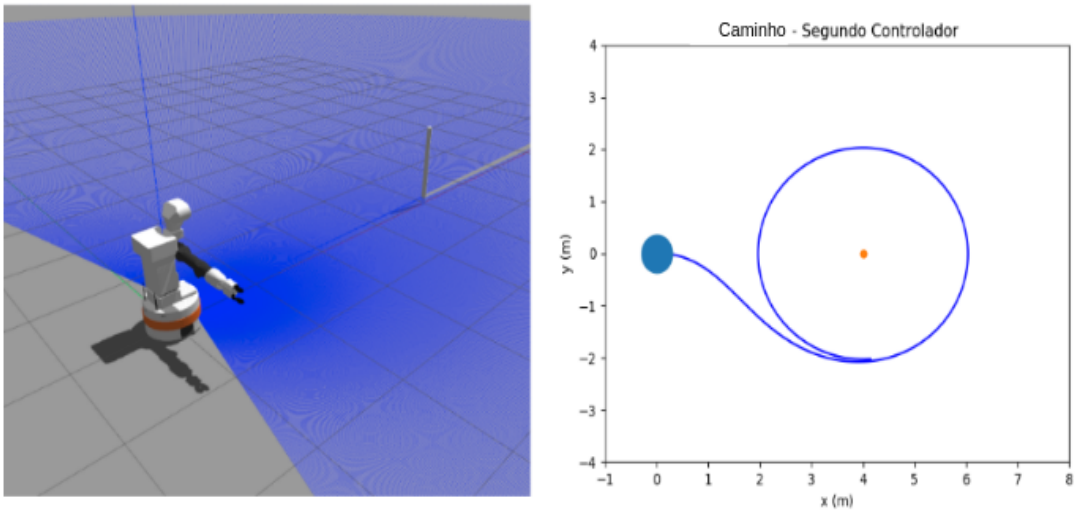


Fonte: Autor

O segundo controlador tem o foco em corrigir o ângulo e a aproximação do robô ao obstáculo para que ele siga o caminho da espiral criada, respeitando a distância mínima estabelecida. As Figuras 17 e 18 ilustram bem a trajetória realizada para $\alpha_B = \pi/2$ e $d^* = 2$m.

Na Figura 17 tem-se à esquerda a imagem do ambiente simulado com o robô e um cilindro muito fino e à direita a trajetória ao redor dele formando uma espiral. Nesse controlador ocorre a aproximação do robô até a distância mínima estabelecida (no caso 2 m de distância) antes de iniciar o contorno ao objeto.

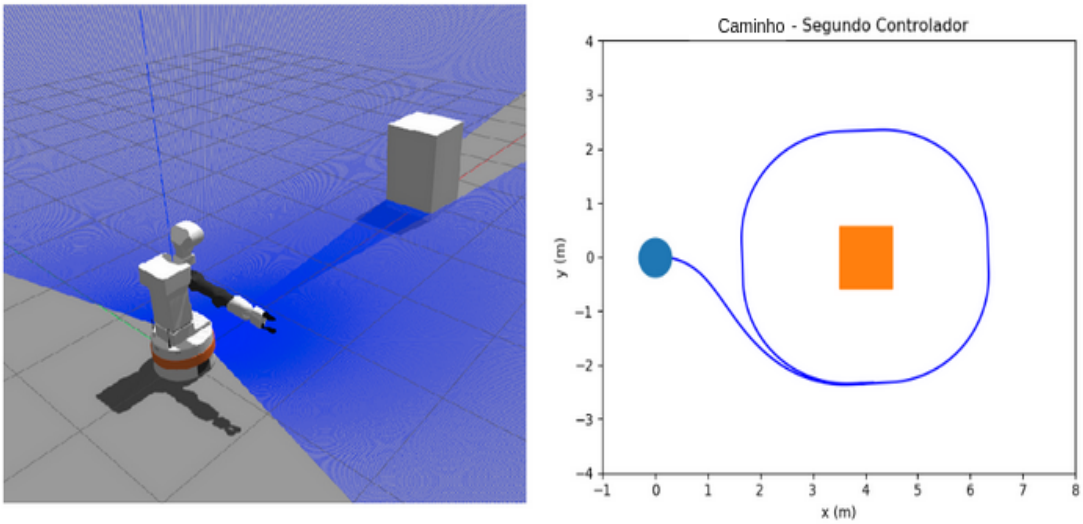
Figura 17 – Caminho espiral do segundo controlador ao redor de um cilindro.



Fonte: Autor

Na Figura 18 tem-se à esquerda a imagem do ambiente simulado com o robô e um cubo e à direita a trajetória ao redor dele formando uma espiral. Aqui também tem-se a aproximação do robô até a distância mínima estabelecida (no caso 2 m de distância) antes de iniciar o contorno ao objeto.

Figura 18 – Caminho espiral do segundo controlador ao redor de um cubo.



Fonte: Autor

Em ambos os casos, o robô ajusta a distância ao obstáculo e começa a seguir a espiral assim que a distância desejada é alcançada. Além disso, em ambos os casos, a distância ao obstáculo é constante. Ao contrário do primeiro, este controlador ajusta a distância e lida com o fato de que o centro da espiral se move levemente a cada iteração.

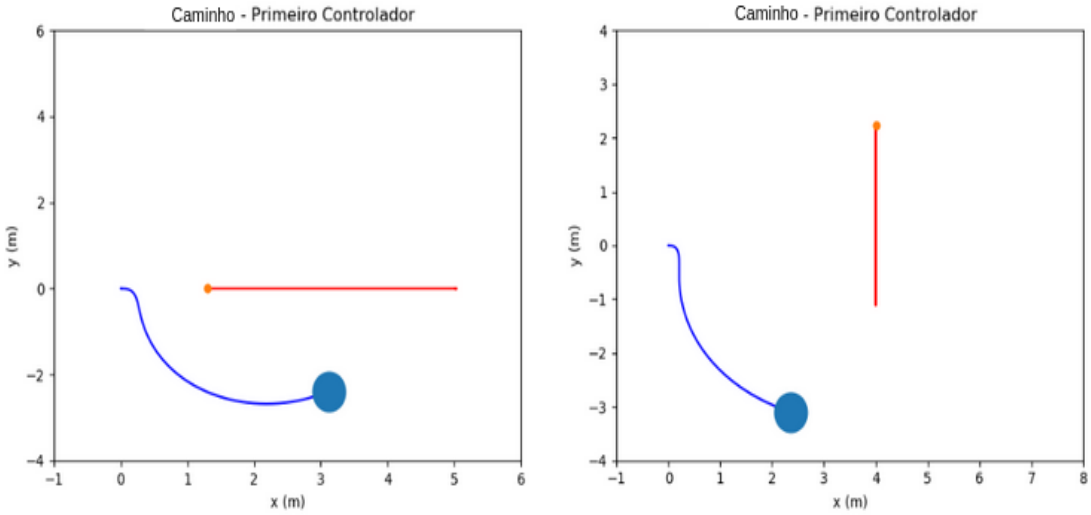
4.2.2 Ambiente Dinâmico

Nesses testes são apresentadas dois tipos de situações, a primeira que é quando o objeto vai na direção do robô e a segunda quando o objeto passa em frente ao robô de um lado a outro. Essas duas situação são aplicadas a dois tipo de objeto, ambos com velocidades de 0.03m/s e o robô à 0.1m/s.

Nas Figuras 19 e 20 foi utilizado o primeiro controlador com $\alpha_B = \pi/2$ em um ambiente dinâmico. Apesar do controlador realizar o propósito de evitar a colisão, o robô, como esperado, inicia a trajetória de desvio bem antecipadamente pois ele apenas corrige o ângulo.

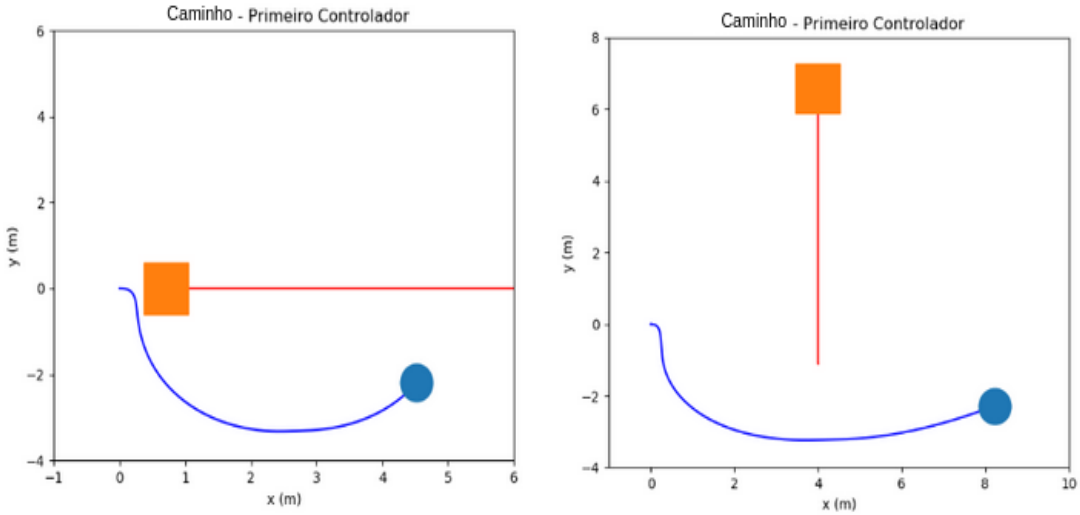
Já as Figuras 21 e 22 mostram a trajetória do segundo controlador com $\alpha_B = \pi/2$ e $d^* = 2$ m em um ambiente dinâmico. Ambas as imagens ilustram situações com um cubo em movimento, onde à direita está a trajetória com a distancia desejada d^* constante, e à esquerda com a distancia desejada d^* modificada pelo valor δ (ver Equação 2.19).

Figura 19 – Caminho espiral do primeiro controlador desviando de um cilindro em movimento.



Fonte: Autor

Figura 20 – Caminho espiral do primeiro controlador desviando de um cubo em movimento.

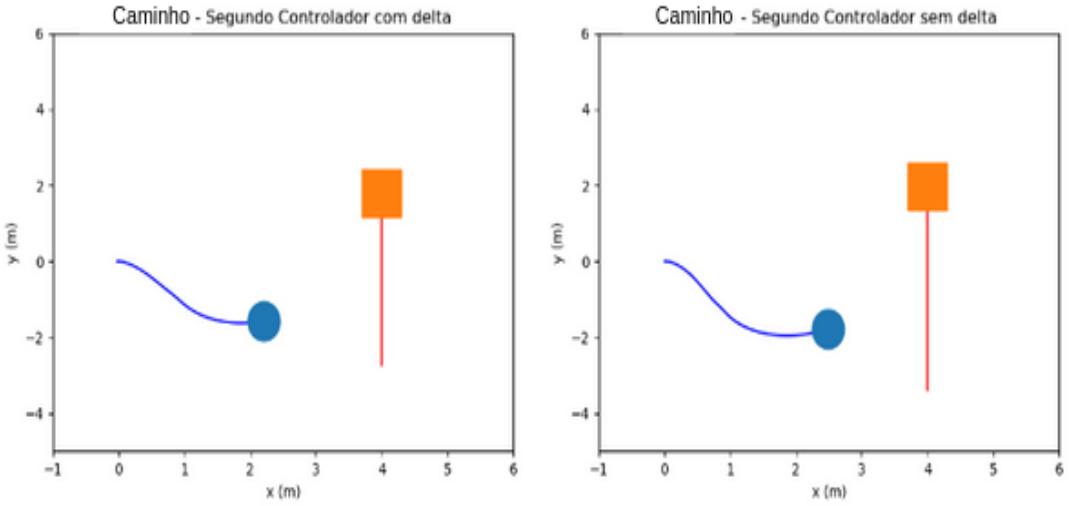


Fonte: Autor

Na Figura 21, como o valor do δ tende a ser negativo, nota-se que há uma ligeira correção do caminho, diminuindo a curva de desvio. Ou seja, em uma situação considerada de "baixo risco"ou "não perigosa", o δ atua para diminuir a distância entre o robô e o objeto.

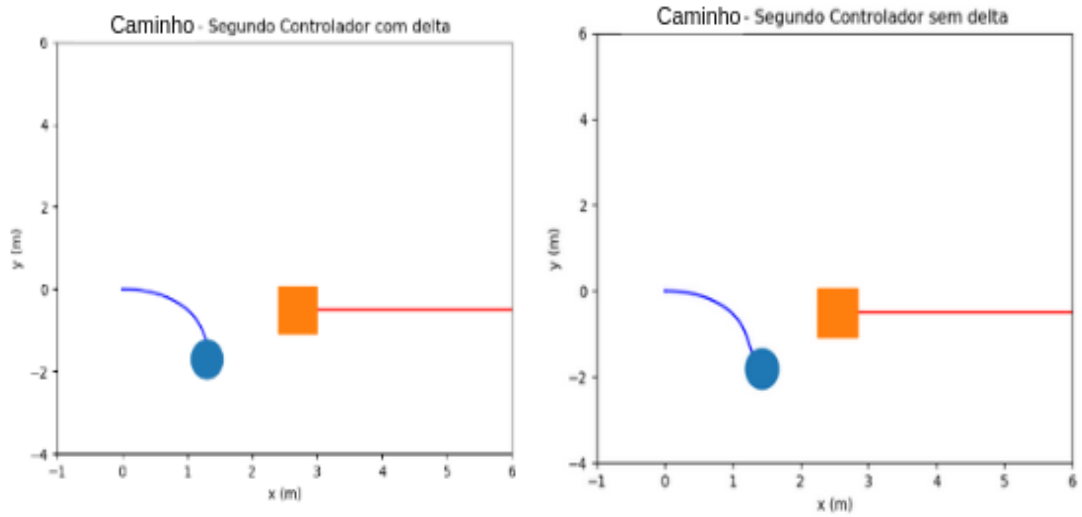
Na Figura 22, como o valor do δ tende a ser positivo, nota-se que também há uma ligeira correção do caminho, antecipando a curva de desvio (comparando as trajetórias). Ou seja, em uma situação considerada de "alto risco"ou "perigosa", com o objeto indo ao encontro do robô, o δ atua para aumentar a distância entre o robô e o objeto.

Figura 21 – Caminho espiral do segundo controlador desviando de um cubo em movimento.



Fonte: Autor

Figura 22 – Caminho espiral do segundo controlador desviando de um cubo em movimento.



Fonte: Autor

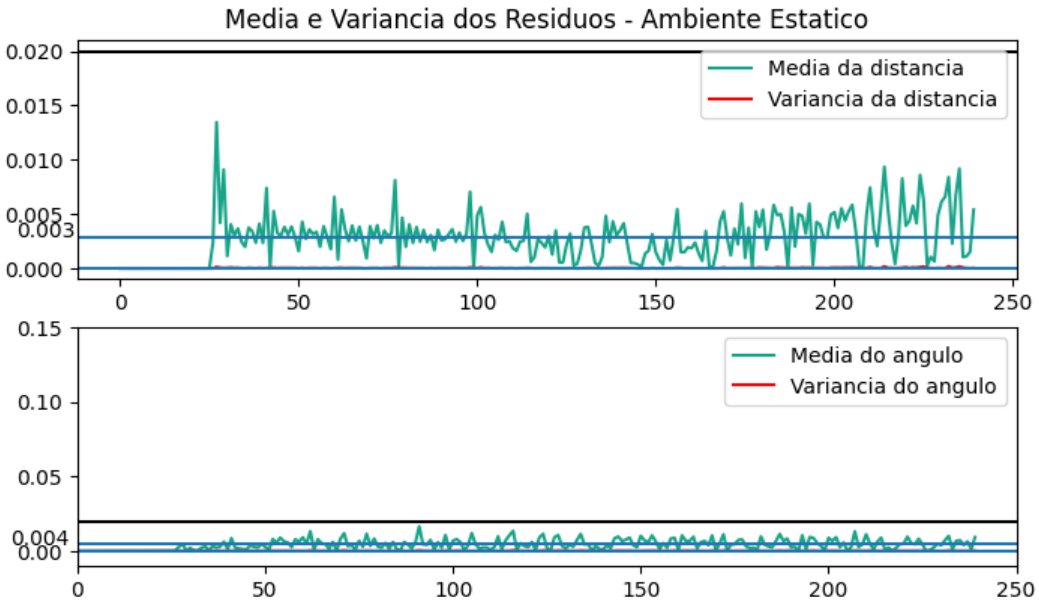
Por realizar um deslocamento mais otimizado, percebe-se que a trajetória é bem mais próxima do obstáculo, o que aumenta a possibilidade de colisão.

4.3 ANÁLISE DO CRITÉRIO DO SEGUNDO CONTROLADOR

As figuras a seguir mostram os gráficos da média e da variância dos resíduos em três casos diferentes. Aqui o buffer de armazenamento do resíduos tem tamanho 12, ou seja, a média e variância são calculadas baseadas no valor mais atual e os últimos 11 valores passados. A

Figura 23 é caso de um objeto estático e com o robô seguido a trajetória da espiral. Na Figura 24 tem-se um caso de objeto dinâmico se movendo da direita para a esquerda e robô estático. Já a Figura 25 é um caso de ambiente totalmente dinâmico, com o robô e o objeto em movimento. Para essas imagens considera-se que o plot superior são dos resíduos da distância e o inferior são dos resíduos do ângulo e a linha azul representa a média desses dados.

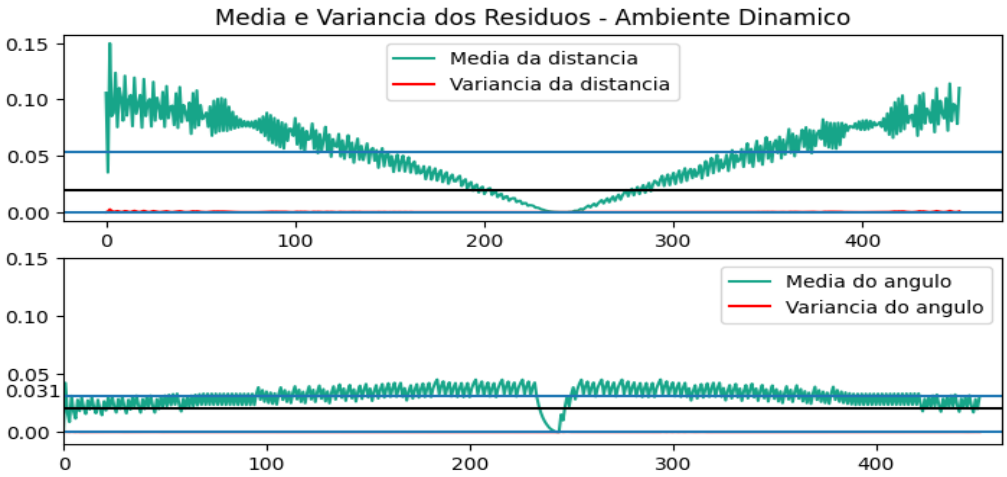
Figura 23 – Gráfico das médias e dos sigmas dos resíduos em um ambiente com obstáculo estático.



Fonte: Autor

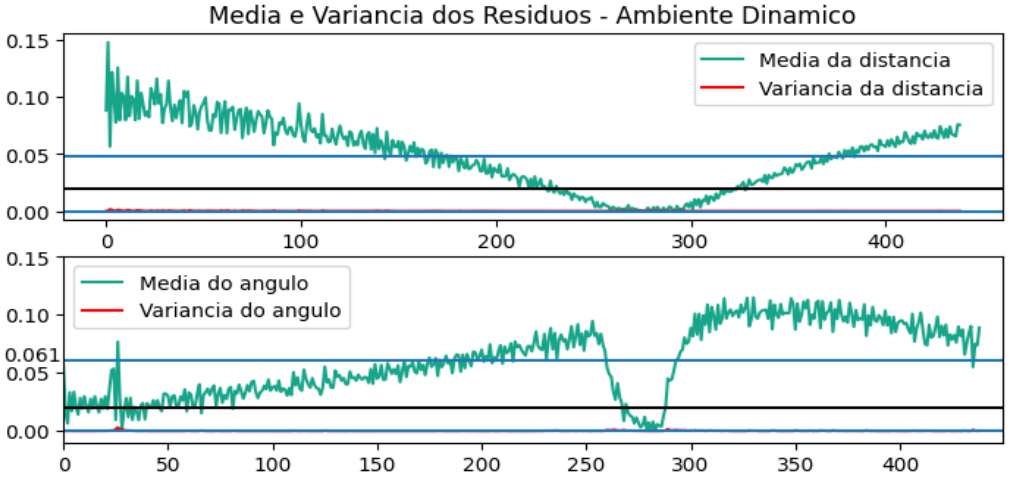
A Tabela 1 apresentada a seguir mostra o desempenho do critério presentado na Equação 2.18, chamado critério do intervalo, usado para identificar objetos dinâmicos. Como se vê na tabela, este critério não parece ser altamente relevante para o problema. De fato, sua taxa média de identificação é de cerca de 54%, com valores que não ultrapassam de 75%. Então, foi proposto definir um novo critério para identificar obstáculos dinâmicos. Para isso, conta-se com a análise dos dados apresentados nas figuras. A maioria dos valores das médias para um objeto estático não ultrapassam de 0.01. Já com o objeto em movimento, o valor da média tem um mínimo em aproximadamente 0.01 e o máximo não passa de 0.15. Assim, o novo limiar é definido pelo valor da média do resíduo da distância que será de 0.02 (linha preta nos gráficos). Logo, se a média dos resíduos da distância e a média dos resíduos do ângulo mais recentes for menor que 0.02, o objeto é considerado como estático, caso contrário será dinâmico. Pode ser visto na Tabela 1 que o novo critério, chamado critério da média, tem um percentual médio de 97%, com valores maiores que 80%.

Figura 24 – Gráfico das médias e dos sigmas dos resíduos em um ambiente com obstáculo dinâmico.



Fonte: Autor

Figura 25 – Gráfico das médias e dos sigmas dos resíduos em um ambiente com obstáculo dinâmico.



Fonte: Autor

Tabela 1 – Tabela do percentual de acertos para identificar um objeto dinâmico

	Critério do intervalo	Critério da média
Tentativa 1	0.29	0.96
Tentativa 2	0.24	0.98
Tentativa 3	0.54	1.00
Tentativa 4	0.72	0.97
Tentativa 5	0.56	0.97
Tentativa 6	0.54	0.98
Tentativa 7	0.24	0.99
Tentativa 8	0.29	0.84
Tentativa 9	0.62	0.81
Tentativa 10	0.60	0.86

5 CONCLUSÕES

Como visto, este trabalho propôs a implementação de um controlador espiral para evitar colisões com objetos, aplicado à um robô diferencial. Também foi analisada a capacidade do robô de reconhecer se o obstáculo é estático ou dinâmico e de desviá-lo. Esses tipos de controladores reativos se mostram bem eficientes em ambientes não previsíveis.

No capítulo anterior, foi visto os resultados da implementação dos controladores e, de fato, o primeiro controlador funcionou bem no sentido de evitar colisões, em situações estáticas e dinâmicas, mas como ele corrige apenas o ângulo, a distância ao objeto (ou raio da trajetória) acaba sendo uma consequência do ângulo, fazendo ele seguir uma trajetória demasiadamente longa (Figuras 16 e 19) ou contornar muito próximo ao objeto. Porém, é importante ressaltar que esse controlador foi projetado para uma situação estática e que, apesar disso, teve um bom desempenho em um ambiente dinâmico, sem ter que modificar/adaptar nada.

Já o segundo controlador permite a correção do ângulo e da distância, ou seja, ele se aproxima até uma distância mínima de segurança definida enquanto faz a correção do ângulo, como bem ilustrado na Figura 18. Porém a situação muda quando o controlador interage com objetos em movimento, pois como o centro da espiral está em constante movimento e o robô segue nessa direção antes de iniciar o desvio, é possível que o tempo de resposta não seja tão rápido quanto a velocidade do objeto.

O critério utilizado para identificação do objeto se baseia no limiar adaptativo δ calculado a partir da média e da variância dos resíduos. Mas, a Tabela 1 mostra que esse critério tem uma precisão média de 54%, o que é bem baixa visto que o controlador lida com objetos dinâmicos. Então, foi proposto um novo critério que se baseia no valor das médias dos resíduos e os resultados indicam uma ótima precisão, aproximadamente 97%. Isso garante que mais vezes ele corrigirá a distância e, consequentemente, sua trajetória.

Para trabalhos futuros, como aqui foram analisadas situações com apenas um objeto e o robô em cena, seria interessante avaliar seu comportamento em um ambiente com diversos obstáculos, em movimento ou não, pois o controlador contorna um objeto mas não é capaz de decidir quais dos objetos no ambiente é o mais perigoso. Também é válida a análise de obstáculos com diferentes velocidades, bem como sugerir uma distância de segurança ótima em função das características físicas e habilidades do robô e como isso influencia na performance do controlador.

REFERÊNCIAS

ACKERMAN, E. Latest version of gazebo simulator makes it easier than ever to not build a robot. **IEEE Spectrum**, 2016. Citado na página 24.

BONIN-FONT, F.; ORTIZ, A.; OLIVER, G. Visual navigation for mobile robots: A survey. **Journal of intelligent and robotic systems**, Springer, v. 53, n. 3, p. 263–296, 2008. Citado na página 10.

BOYADZHIEV, K. N. Spirals and conchospirals in the flight of insects. **The college mathematics Journal**, Taylor & Francis, v. 30, n. 1, p. 23–31, 1999. Citado 2 vezes nas páginas 14 e 16.

CHOSET, H. et al. **Principles of robot motion: theory, algorithms, and implementations**. [S.l.]: MIT press, 2005. Citado na página 10.

DING, X.; FRANK, P. Frequency domain approach and threshold selector for robust model-based fault detection and isolation. In: **Fault Detection, Supervision and Safety for Technical Processes 1991**. [S.l.]: Elsevier, 1992. p. 271–276. Citado na página 21.

DURAND-PETITEVILLE, A. et al. Design of a sensor-based controller performing u-turn to navigate in orchards. In: **International Conference on Informatics in Control, Automation and Robotics**. [S.l.: s.n.], 2017. v. 2, p. 172–181. Citado na página 16.

DURAND-PETITEVILLE, A. et al. Design of a sensor-based controller performing u-turn to navigate in orchards. In: **International Conference on Informatics in Control, Automation and Robotics**. [S.l.: s.n.], 2017. v. 2, p. 172–181. Citado na página 18.

EMAMI-NAEINI, A.; AKHTER, M. M.; ROCK, S. M. Effect of model uncertainty on failure detection: the threshold selector. **IEEE Transactions on Automatic Control**, IEEE, v. 33, n. 12, p. 1106–1115, 1988. Citado na página 21.

FUTTERLIEB, M. **Vision based navigation in a dynamic environment**. Tese (Doutorado) — Université Paul Sabatier-Toulouse III, 2017. Citado na página 16.

KOENIG, N.; HOWARD, A. Design and use paradigms for gazebo, an open-source multi-robot simulator. In: IEEE. **2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)(IEEE Cat. No. 04CH37566)**. [S.l.], 2004. v. 3, p. 2149–2154. Citado na página 24.

LAB, G. T. G.; LAB, H. R. **DART (Dynamic Animation and Robotics Toolkit)**. 2022. Disponível em: http://dartsim.github.io/. Citado na página 24.

LECA, D. et al. Sensor-based obstacles avoidance using spiral controllers for an aircraft maintenance inspection robot. In: IEEE. **2019 18th European Control Conference (ECC)**. [S.l.], 2019. p. 2083–2089. Citado 4 vezes nas páginas 14, 15, 16 e 17.

MINGUEZ, J.; LAMIRAUX, F.; LAUMOND, J.-P. Motion planning and obstacle avoidance. In: **Springer handbook of robotics**. [S.l.]: Springer, 2016. p. 1177–1202. Citado na página 10.

OGRE3D. **Object-Oriented Graphics Rendering Engine**. 2022. Disponível em: http://www.ogre3d.org/. Citado na página 24.

QUIGLEY, M.; GERKEY, B.; SMART, W. D. **Programming Robots with ROS: a practical introduction to the Robot Operating System**. [S.l.]: "O'Reilly Media, Inc.", 2015. Citado 2 vezes nas páginas 12 e 13.

RENAK, A. et al. Design of an intelligent navigation strategy to deal with unexpected dynamic obstacles. In: **Brazilian Symposium on Intelligent Automation**. [S.l.: s.n.], 2019. Citado na página 20.

ROBOTICS, P. **TIAGo - PAL Robotics: Leading Service Robotics**. 2021. Disponível em: https://pal-robotics.com/robots/tiago/. Citado 2 vezes nas páginas 23 e 24.

SHI, Z. et al. The development of an adaptive threshold for model-based fault detection of a nonlinear electro-hydraulic system. **Control Engineering Practice**, Elsevier, v. 13, n. 11, p. 1357–1367, 2005. Citado 2 vezes nas páginas 20 e 21.

SIEGWART, R.; NOURBAKHSH, I. R.; SCARAMUZZA, D. **Introduction to autonomous mobile robots**. [S.l.]: MIT press, 2011. Citado na página 10.

SIMBODY. **Simbody: Multibody Physics API**. 2022. Disponível em: https://simtk.org/home/simbody/. Citado na página 24.

SIMULATION, R.-T. P. **BULLET Physics Library**. 2022. Disponível em: https://pybullet.org/wordpress/. Citado na página 24.

SMITH, R. L. **Open Dynamics Engine**. 2022. Disponível em: http://www.ode.org/. Citado na página 24.

WANG, X.; KRUGER, U.; LENNOX, B. Recursive partial least squares algorithms for monitoring complex industrial processes. **Control Engineering Practice**, Elsevier, v. 11, n. 6, p. 613–632, 2003. Citado na página 21.

6 LASERPROCESSING.PY

```
#! /usr/bin/python

import rospy
import math
import time
import numpy as np

#import for message
from scp.msg import spiral_center
from scp.msg import spiral_residue

#import for laser
from sensor_msgs.msg import LaserScan
from geometry_msgs.msg import Twist

#import for odometry
from nav_msgs.msg import Odometry
from tf.transformations import euler_from_quaternion


class myRobot():

    def __init__(self):
        self.Nstatic = 0
        self.Ndynamic = 0

        # Init the callback variables
        self.position_list = [0, 0, 0]
        self.orientation = 0

        # Buffer variables
        self.bufferQuantity = 0

        # Coordinates of the center of the obstacle Ob
        x_Ob = y_Ob = theta_Ob = dist_Ob = 0

        # Coordinates of the closest point Oc
        x_Oc = y_Oc = theta_Oc = dist_Oc = 0

        self.staticObs = False
        self.saveLaserData = True
        # Publisher message
        self.pub_message1 = rospy.Publisher('/spiral/center', spiral_center, queue_size=1)
        self.msgCenter = spiral_center()
        self.pub_message2 = rospy.Publisher('/spiral/residue', spiral_residue, queue_size=1)
        self.msgResidue = spiral_residue()
        # Subscriber laser
        if (self.staticObs):
            self.sub_laser = rospy.Subscriber('/scan_raw', LaserScan, self.callback_laser_obsstatic, \
                queue_size=1)
        else:
            self.sub_laser = rospy.Subscriber('/scan_raw', LaserScan, self.callback_laser_obsdinamic, \
                queue_size=1)
        self.obj_point_coord = []
```

```
    # Subscriber odometry
    self.sub_odo = rospy.Subscriber('/mobile_base_controller/odom', Odometry, \
        self.callback_odometria, queue_size=1)

    self.buffer_laser_size = 12
    self.buffer_laser = np.zeros((self.buffer_laser_size, 666, 4))
    self.buffer_odom = np.zeros((self.buffer_laser_size, 3)) # [x, y, orientation]
    self.buffer_residue = np.zeros((self.buffer_laser_size, 3)) # [residue, time]
    self.buffer_laser_index = -1    # to store the data in the most current index possible


def callback_laser_obsstatic(self, msg):
    # store laser data
    self.increment = msg.angle_increment
    self.size = len(msg.ranges)
    self.range_max = msg.range_max

    if self.saveLaserData:
        self.saveLaserData = False

        self.obj_point_coord = []
        self.past_iterations = []
        self.values_OcList = []
        self.values_Oc = []
        for i in range(self.size):
            if msg.ranges[i] < 25 and msg.ranges[i] > 0.15:      #laser noise filter
                self.obj_point_coordList = []
                theta = self.increment * (i - self.size/2)       #theta in rad
                Xpoint = msg.ranges[i] * math.cos(theta)
                Ypoint = msg.ranges[i] * math.sin(theta)
                self.obj_point_coordList.append(Xpoint)
                self.obj_point_coordList.append(Ypoint)
                self.obj_point_coord.append(self.obj_point_coordList)

                self.values_OcList.append(msg.ranges[i])
                self.values_OcList.append(theta)
                self.values_Oc.append(self.values_OcList)
    (a, b) = tiago.calculate_Ob(self.obj_point_coord)
    (c, d) = tiago.calculate_Oc(self.obj_point_coord)
    (e, f) = tiago.compare_OcOb(a, b, c, d)

def callback_laser_obsdinamic(self, msg):
    # Log current time
    time_value = (msg.header.stamp.nsecs)*10**(-9) + msg.header.stamp.secs #time in seconds
    self.time = float(time_value)

    # Log current pose
    xRobot = self.position_list[0]
    yRobot = self.position_list[1]
    thetaRobot = self.orientation

    # store laser data
    self.increment = msg.angle_increment
    self.size = len(msg.ranges)
    self.range_max = msg.range_max


    self.obj_point_coord = []
    self.past_iterations = []
    self.values_OcList = []
```

```
        self.values_Oc = []

        for i in range(self.size):
            laser_value = msg.ranges[i]
            if msg.ranges[i] > 25 or msg.ranges[i] < 0.15:  #laser noise filter
                laser_value = float("NaN")
            self.obj_point_coordList = []
            theta = self.increment * (i - self.size/2)      #theta in rad
            Xpoint = laser_value * math.cos(theta) + 0.202 # To express in the robot frame
            Ypoint = laser_value * math.sin(theta)          # and not in the laser one
            self.obj_point_coordList.append(Xpoint)
            self.obj_point_coordList.append(Ypoint)
            self.obj_point_coordList.append(0)
            self.obj_point_coordList.append(1)
            self.obj_point_coord.append(self.obj_point_coordList)

        # Update the index value of the buffer
        if self.buffer_laser_index == self.buffer_laser_size -1:
            self.buffer_laser_index = 0
        else:
            self.buffer_laser_index += 1

        # Update the quantity of data in the buffer
        if(self.bufferQuantity < self.buffer_laser_size):
            self.bufferQuantity += 1

        # Store in the laser buffer
        self.buffer_laser[self.buffer_laser_index,:,:] = self.obj_point_coord
        # Store in the odometry buffer  ## [x, y, orientation z]
        self.buffer_odom[self.buffer_laser_index,:] = [xRobot,yRobot,thetaRobot]
        # Store the time in the residue buffer    ## [distance_residue, theta_residue time]
        self.buffer_residue[self.buffer_laser_index,:] = [0, 0, self.time]

        tiago.compute_residue()
        (medD, sigD, medT, sigT) = tiago.calculate_mean_variance()
        tiago.delta_interval(medD, sigD, medT, sigT)

        self.saveLaserData = True

    def callback_odometria(self, msg):
        # store odometry data
        (x, y, z) = euler_from_quaternion ([msg.pose.pose.orientation.x, \
            msg.pose.pose.orientation.y, msg.pose.pose.orientation.z, msg.pose.pose.orientation.w])
        self.orientation = z
        self.position_list = [msg.pose.pose.position.x, msg.pose.pose.position.y, \
            msg.pose.pose.position.z]

    def trans_matrix(self, odom_data_past, odom_data_current):

        # Homogeneous transformation matrix from current frame to world frame
        dX = odom_data_current[0]
        dY = odom_data_current[1]
        dz = odom_data_current[2]
        H_w_c = np.array([[ math.cos(dz), -math.sin(dz), 0, dX],
                          [ math.sin(dz), math.cos(dz), 0, dY],
                          [ 0           , 0    , 1 , 0],
                          [ 0           , 0    , 0 , 1]])

        # Homogeneous transformation matrix from past frame to world frame
        dX = odom_data_past[0]
```

```
        dY = odom_data_past[1]
        dz = odom_data_past[2]
        H_w_p = np.array([[ math.cos(dz), -math.sin(dz), 0, dX],
                          [ math.sin(dz), math.cos(dz), 0, dY],
                          [ 0           , 0    , 1 , 0],
                          [ 0           , 0    , 0 , 1]])

        # Homogeneous transformation matrix from past frame to current frame
        H_c_w = np.linalg.inv(H_w_c)
        H_c_p = np.dot(H_c_w, H_w_p)


        return H_c_p
        # return np.linalg.inv(H_w_p)

    def frame_projection(self, mat_t, buffer_laser_data):
        # product of matrix T by the laser data matrix
        transposta = np.transpose(buffer_laser_data)
        frame_prjt = []
        frame_prjt = np.dot(mat_t, transposta)

        return frame_prjt

    def compute_residue(self):
        init = self.buffer_laser_index
        projection = []
        mat_T = []
        #filtering out laser data that is different from 'NaN'
        dataToProcess = []
        for data in self.buffer_laser[init]:
            if not math.isnan(data[0]):
                dataToProcess.append(data)

        ## calculating the current Os distance to compare with the previous data in the buffer
        (d_Ob_ti, t_Ob_ti) = tiago.calculate_Ob(dataToProcess)
        (d_Oc_ti, t_Oc_ti) = tiago.calculate_Oc(dataToProcess)
        (d_Os_ti, t_Os_ti) = tiago.compare_OcOb(d_Ob_ti, t_Ob_ti, d_Oc_ti, t_Oc_ti)
        time_ti = self.buffer_residue[init][2]
        self.dOs_current = d_Os_ti
        self.tOs_current = t_Os_ti

        # Initiaize the buffer reading from the current line
        bufferIndex = init

        for i in range(self.bufferQuantity-1):
            # init bufferQuantity = 0, bufferQuantity+=1 in each
            # laserCallback until bufferQuantity == bufferSize
            # update the buffer index
            bufferIndex -= 1
            if bufferIndex < 0:
                bufferIndex = self.buffer_laser_size -1

            # extract the laser data to process and filter the NaN
            dataToProcess = []
            for data in self.buffer_laser[bufferIndex]:
                if not math.isnan(data[0]):
                    dataToProcess.append(data)

            ## compute transformation matrix
            mat_T = tiago.trans_matrix(self.buffer_odom[bufferIndex], self.buffer_odom[init])
```

```
        ## project laser data into current frame (matrix * dados)
        projection = tiago.frame_projection(mat_T, dataToProcess)

        ## compute predicted spiral center      # time in nanoseconds
        (d_Ob_pred, t_Ob_pred) = tiago.calculate_Ob(np.transpose(projection))
        (d_Oc_pred, t_Oc_pred) = tiago.calculate_Oc(np.transpose(projection))
        (d_Os_pred, t_Os_pred) = tiago.compare_OcOb(d_Ob_pred, t_Ob_pred, d_Oc_pred, t_Oc_pred)
        residue = (d_Os_pred - d_Os_ti)/((self.buffer_residue[bufferIndex][2] - time_ti))
        residue2 = (t_Os_pred - t_Os_ti)/((self.buffer_residue[bufferIndex][2] - time_ti))

        ## save results in an array
        self.buffer_residue[bufferIndex][0] = residue
        self.buffer_residue[bufferIndex][1] = residue2


def calculate_mean_variance(self):
    sum = 0
    sum2 = 0
    # computing the mean of the distance residuals
    bufferIndex = self.buffer_laser_index
    for i in range(self.bufferQuantity-1):
        # init bufferQuantity = 0, bufferQuantity+=1 in each
        # laserCallback until bufferQuantity == bufferSize
        bufferIndex -= 1
        if bufferIndex < 0:
            bufferIndex = self.buffer_laser_size -1

        sum += self.buffer_residue[bufferIndex][0]
    mean_d = sum/(self.bufferQuantity-1)

    # computing the variance of the distance residuals
    bufferIndex = self.buffer_laser_index
    for i in range(self.bufferQuantity-1):
        # init bufferQuantity = 0, bufferQuantity+=1 in each
        # laserCallback until bufferQuantity == bufferSize
        bufferIndex -= 1
        if bufferIndex < 0:
            bufferIndex = self.buffer_laser_size -1
        sum2 += (self.buffer_residue[bufferIndex][0] - mean_d)**2
    sigma2 = sum2/(self.bufferQuantity-1)
    sigma_d = math.sqrt(sigma2)

    sum = 0
    sum2 = 0
    sigma2 = 0
    ### computing the mean of the theta residuals
    bufferIndex = self.buffer_laser_index
    for i in range(self.bufferQuantity-1):
        # init bufferQuantity = 0, bufferQuantity+=1 in each
        # laserCallback until bufferQuantity == bufferSize
        bufferIndex -= 1
        if bufferIndex < 0:
            bufferIndex = self.buffer_laser_size -1

        sum += self.buffer_residue[bufferIndex][1]
    mean_t = sum/(self.bufferQuantity-1)

    ### computing the variance of the theta residuals
    bufferIndex = self.buffer_laser_index
    for i in range(self.bufferQuantity-1):
```

```
            # init bufferQuantity = 0, bufferQuantity+=1 in each
            # laserCallback until bufferQuantity == bufferSize
            bufferIndex -= 1
            if bufferIndex < 0:
                bufferIndex = self.buffer_laser_size -1
            sum2 += (self.buffer_residue[bufferIndex][1] - mean_t)**2
        sigma2 = sum2/(self.bufferQuantity-1)
        sigma_t = math.sqrt(sigma2)
        # print('{} {} {} {}'.format(mean_d, sigma_d, mean_t, sigma_t))

        return mean_d, sigma_d, mean_t, sigma_t

    def delta_interval(self, mean_d, sigma_d, mean_t, sigma_t):
        # computing the delta interval for the residues
        delta_min_d = mean_d - 2.17*sigma_d
        delta_max_d = mean_d + 2.17*sigma_d
        recent_residue_d = self.buffer_residue[self.buffer_laser_index-1][0]
        delta_min_t = mean_t - 2.17*sigma_t
        delta_max_t = mean_t + 2.17*sigma_t
        recent_residue_t = self.buffer_residue[self.buffer_laser_index-1][1]

        # if the most recent residue value is within the
        # delta range, then the obstacle is static
        state = ''

        # range criterion
        if recent_residue_d >= delta_min_d and recent_residue_d <= delta_max_d:
        # new criterion to identify an obstacle
        # if abs(mean_d) < 0.02 and abs(mean_t) < 0.02:

            self.Nstatic+= 1
            state = 'STATIC'
        else:

            self.Ndynamic+= 1
            state = 'DYNAMIC'

        print(self.Nstatic, self.Ndynamic)
        self.msgCenter.distance_Os = self.dOs_current
        self.msgCenter.thetaf_Os = self.tOs_current
        self.msgResidue.obs_state = state
        self.msgResidue.deltaD = delta_max_d
        self.msgResidue.deltaT = delta_max_t
        self.pub_message1.publish(self.msgCenter)
        self.pub_message2.publish(self.msgResidue)

    def calculate_Ob(self, laser_projected_data):
        #calculate the Ob center i.e baricenter
        x = y = 0
        xSum = ySum = 0

        for i in range(len(laser_projected_data)):
            x = laser_projected_data[i][0]
            y = laser_projected_data[i][1]
            xSum = xSum + x
            ySum = ySum + y
        if len(laser_projected_data) > 0:
            self.x_Ob = xSum/len(laser_projected_data)
            self.y_Ob = ySum/len(laser_projected_data)
            self.theta_Ob = np.arctan2(self.y_Ob,self.x_Ob)
```

```
            self.dist_Ob = math.sqrt(self.x_Ob**2 + self.y_Ob**2)

        self.saveLaserData = True

        return self.dist_Ob, self.theta_Ob

    def calculate_Oc(self, laser_projected_data):
        #calculate the Oc i.e closest point
        distOc_min = self.range_max
        thetaOc_min = 0
        list_mat_laser_projected_data = []

        for i in range(len(laser_projected_data)):
            d = math.sqrt(laser_projected_data[i][0]**2 + laser_projected_data[i][1]**2)
            th = np.arctan2(laser_projected_data[i][1], laser_projected_data[i][0])
            list_mat_laser_projected_data.append([d,th])

        for n in range(len(list_mat_laser_projected_data)):
            if list_mat_laser_projected_data[n][0] < distOc_min:
                distOc_min = list_mat_laser_projected_data[n][0]
                thetaOc_min = list_mat_laser_projected_data[n][1]
        self.dist_Oc = distOc_min
        self.theta_Oc = thetaOc_min

        return distOc_min, thetaOc_min

    def compare_OcOb(self, dOb, tOb, dOc, tOc):
        #choose between Ob and Oc which one will be Os
        self.dist_Os = 0
        self.theta_Os = 0
        if dOc > dOb:
            self.dist_Os = dOb
            self.theta_Os = tOb
        if dOb > dOc:
            self.dist_Os = dOc
            self.theta_Os = tOc

        dOs = self.dist_Os
        tOs = self.theta_Os

        return dOs, tOs

if __name__ == "__main__":
    # Start the node
    rospy.init_node("/laser_processing_node")
    tiago = myRobot()
    time.sleep(1)


    rospy.spin()
```

7 SPIRALCONTROLLER.PY

```
#! /usr/bin/python
import rospy
import math
import time
import numpy

#import for reading the center
from scp.msg import spiral_center
#import for reading the residue
from scp.msg import spiral_residue
#import for base movements
from geometry_msgs.msg import Twist
#import for gazebo odometry
from gazebo_msgs.msg import ModelStates
from tf.transformations import euler_from_quaternion

class myRobotmove():

    def __init__(self):
        # Subscriber laser
        self.sub_centermsg = rospy.Subscriber('/spiral/center', spiral_center, \
            self.callback_center, queue_size=1)
        self.sub_residuemsg = rospy.Subscriber('/spiral/residue', spiral_residue, \
            self.callback_residue, queue_size=1)

        # Publisher base
        self.pub_base = rospy.Publisher('/mobile_base_controller/cmd_vel', Twist, queue_size=1)
        self.cmd = Twist()
        self.approach = 0

        # Subsriber gazebo odometry
        self.sub_odo_gaz = rospy.Subscriber('/gazebo/model_states', ModelStates, \
            self.callback_odometria_gazebo, queue_size=1)

    def callback_center(self, msg):
        self.center_dist = msg.distance_Os
        self.center_theta = msg.thetaf_Os

    def callback_residue(self, msg):
        self.obstacle = msg.obs_state
        self.delta_max_d = msg.deltaD
        self.delta_max_t = msg.deltaT

    def callback_odometria_gazebo(self, msg):
        (x, y, z) = euler_from_quaternion ([msg.pose[2].orientation.x, msg.pose[2].orientation.y, \
            msg.pose[2].orientation.z, msg.pose[2].orientation.w])
        self.giro = z
        self.pose_list = [msg.pose[2].position.x, msg.pose[2].position.y, z]

    def alpha_controller(self):      # first_approach
        alpha_b = math.pi/2
        lambda_b = 1
        Vlinear = 0.1

        erro_b = self.center_theta - alpha_b
```

```
        Vangular = lambda_b*erro_b + (Vlinear/self.center_dist)*math.sin(self.center_theta)

        self.cmd.linear.x = Vlinear
        self.cmd.angular.z = Vangular
        self.pub_base.publish(self.cmd)


    def initAlphaDistanceController(self, alpha, Lambda, v, d):
        self.alpha_b = alpha
        self.lambda_s = Lambda
        self.Vlinear = v
        self.d_mindef = d

        self.d_inicial = self.center_dist

    def alphaDistance_controller(self):    # second_approach
        deltad = 0
        # correcting d* with delta(+)
        deltad = self.delta_max_d
        self.d_min = self.d_mindef + deltad

        epsilon = math.copysign(1.0, self.d_min - self.center_dist) * min(abs((self.d_min \
            - self.center_dist)/abs(self.d_min - self.d_inicial)), 1)
        if self.d_min > self.d_inicial:
            alpha_d = math.copysign(1.0, self.alpha_b) * math.pi - self.alpha_b
        elif self.d_min < self.d_inicial:
            alpha_d = self.alpha_b

        erro_s = self.center_theta - self.alpha_b - alpha_d * epsilon

        Vangular = self.lambda_s * erro_s + (self.Vlinear/self.center_dist) \
            * math.sin(self.center_theta) #- alpha_d * epsilon

        self.cmd.linear.x = self.Vlinear
        self.cmd.angular.z = Vangular
        self.pub_base.publish(self.cmd)


if __name__ == "__main__":
    # Start the node
    rospy.init_node("/spiral_controller_node")
    tiago = myRobotmove()
    time.sleep(1)

    tiago.initAlphaDistanceController(math.pi/2, 1, 0.1, 2)
    r = rospy.Rate(5)

    while not rospy.is_shutdown():
        # tiago.alpha_controller()
        tiago.alphaDistance_controller()

        r.sleep()
    rospy.spin()
```