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

PAULO FELIPE MAIA PASSOS BRITO

Implementação em ROS de uma estratégia de navegação reativa para um robô móvel em um ambiente com obstáculos estáticos.

Recife
2022

PAULO FELIPE MAIA PASSOS BRITO

Implementação em ROS de uma estratégia de navegação reativa para um robô móvel em um ambiente com obstáculos estáticos.

Trabalho de Conclusão de Curso apresentado ao Departamento de Engenharia Mecânica da Universidade Federal de Pernambuco, como requisito para obtenção do grau 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

Maia Passos Brito, Paulo Felipe.
Implementação em ROS de uma estratégia de navegação reativa para um robô móvel em um ambiente com obstáculos estáticos / Paulo Felipe Maia Passos Brito. - Recife, 2022.
76 p. : il.

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. robótica móvel. 2. navegação reativa. 3. image-based visual servoing. 4. seguimento de espiral. 5. desvio de obstáculos estáticos. I. Durand-Petiteville, Adrien. (Orientação). II. Título.

620 CDD (22.ed.)

RESUMO

Este trabalho tem como foco o problema de navegação de robôs móveis em um ambiente com obstáculos. Propõe-se a implementação de uma solução reativa composta por dois controladores para realizar a navegação. O primeiro, controlador Image-Based Visual Servoing, depende de imagens para conduzir o robô em direção ao objetivo. O segundo, controlador de seguimento de espiral, usa dados de um LiDAR para evitar colisões com obstáculos. Além disso, é apresentado uma máquina de estados finitos que permite selecionar o controlador apropriado dependendo da situação. Assim, neste trabalho, é apresentado os algoritmos para processar os dados, os dois controladores e a máquina de estados finitos. A seguir, é detalhado como esses métodos foram implementados dentro do framework ROS usando a linguagem Python. Por fim, é apresentado resultados obtidos em simulações utilizando o *software* Gazebo. Esses resultados dizem respeito aos algoritmos de processamento de dados, ambos controladores, e à estratégia geral de navegação para diferentes tipos de obstáculos.

Palavras-chave: robótica móvel; navegação reativa; image-based visual servoing; seguimento de espiral; desvio de obstáculos estáticos; ROS; Gazebo; Python.

ABSTRACT

This paper focuses on the problem of navigation of mobile robots in an environment with obstacles. It is proposed the implementation of a reactive solution composed of two controllers to perform the navigation. The first Image-Based Visual Servoing controller, relies on images to guide the robot towards the objective. The second, spiral-following controller, uses data from a LiDAR to avoid collisions with obstacles. In addition, a finite state machine is presented that allows selecting the appropriate controller depending on the situation. Thus, in this paper, the algorithms for processing the data, the two controllers and the finite state machine are presented. Next, it is detailed how these methods were implemented within the ROS framework using the Python language. Finally, results obtained from simulations using the Gazebo *software* are presented. These results concern the data processing algorithms, both controllers, and the overall navigation strategy for different types of obstacles.

Keywords: mobile robotics; reactive navigation; image-based visual servoing; spiral tracking; static obstacles avoidance; ROS; Gazebo; Python.

LISTA DE FIGURAS

SUMÁRIO

1 INTRODUÇÃO

A partir do fim do século XX, os robôs vêm chamando a atenção dos seres humanos por serem utilizados para substituir ou auxiliar humanos em atividades repetitivas, perigosas, de alta precisão ou custosas. Alguns exemplos dessa transformação são robôs de linhas de montagem industriais, robôs de resgate, robôs de atividades domésticas, robôs cirurgiões ou robôs para inspeção, sendo eles autônomos ou remotamente controlados (MELO et al., 2012). Para realizar suas atividades de forma segura, sem ocorrer colisões e causar acidentes indesejados, principalmente os que operam com objetivo de busca, resgate e inspeção, é de extrema importância conhecer o ambiente ao seu redor para poder executar uma navegação segura, detectando e analisando possíveis obstáculos, tendo um controle e implementação eficiente, sendo esses os principais desafios da robótica atualmente.

Muitas aplicações exigem que o robô seja capaz de se mover de forma autônoma em um ambiente pouco conhecido ou desconhecido. A literatura propõe uma grande variedade de abordagens que abrangem um amplo espectro. De um lado do espectro, uma primeira classe de métodos, chamada de abordagens baseadas em mapas, baseia-se em um mapa do ambiente para planejar uma trajetória que leva ao objetivo enquanto lida com restrições como a presença de obstáculos (SIEGWART; NOURBAKHSH; SCARAMUZZA, 2011). Os desafios desses métodos residem principalmente na construção do mapa e na localização do robô. Primeiro, o mapa deve ser preciso o suficiente para permitir uma estimativa precisa da pose do robô. Em segundo lugar, deve conter todas as informações relevantes para conduzir a uma trajetória que lide com as restrições, por exemplo a presença de obstáculos. Assim, esses 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 (por exemplo (MINGUEZ; LAMIRAUX; LAUMOND, 2016)).

Do outro lado do espectro, uma segunda classe de métodos, chamada de abordagens reativas ou sem mapa, não usa um mapa do ambiente e se beneficia de informações locais. A maioria desses métodos se baseia em abordagens baseadas em sensores, no sentido em que a pose atual e o objetivo são definidos em termos de medidas relevantes no espaço do sensor (SIEGWART; NOURBAKHSH; SCARAMUZZA, 2011). Diferentes sensores podem ser considerados como telêmetros de visão ou 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 sensoriais atuais e de referência. Assim, nenhum processo de localização em

relação a um quadro global ou a um mapa é necessário. O desafio nesta classe de métodos está em projetar controladores capazes de garantir a estabilidade em malha fechada enquanto gerenciam as diferentes restrições necessárias para a execução da tarefa (por exemplo, evitar obstáculos, saturação do atuador, etc.). Esses métodos parecem relevantes para navegar na presença de obstáculos desconhecidos, mas parecem ser menos adequados para lidar com inúmeros restrições.

Neste trabalho, o foco se dará pela abordagem reativa para um robô móvel em um ambiante com obstáculos, onde serão apresentados dois controladores: controlador seguidor de espiral e o controlador IBVS (*image-based visual servoing*). O primeiro controlador permite seguir uma espiral com o objetivo de evitar colisões baseado no dados obtidos pelo laser, enquanto o segundo é um controlador baseado em imagem e tem como objetivo dirigir o robô até o alvo. Para ser possível definir a espiral, será apresentado um processamento de dados do laser a fim de obter o centro da espiral para obstáculos côncavos e convexos. O projeto dos controladores se baseiam no trabalho apresentado em Boyadzhiev (1999), que descreve como os insetos voam em torno de um ponto de interesse executando uma espiral, e na pesquisa apresentada por Durand-Petiteville et al. (2017). Em outras palavras, mostra como seguir um caminho, baseado em uma espiral, conhecendo apenas as coordenadas de um ponto de interesse no objeto em movimento. A partir dos controladores implementados, será apresentado uma máquina de estados finitos para alternar entre os dois controladores quando necessário a fim de se obter uma navegação eficiente e segura.

1.1 Objetivos

O objetivo deste trabalho é implementar usando o *framework* ROS uma estratégia de navegação reativa para um robô móvel evoluindo em um ambiente com obstáculos estáticos. O sistema de navegação consiste em um controlador IBVS usando informações visuais, um controlador seguidor de espiral baseado em dados de laser e uma máquina de estados finitos selecionando o controlador adaptado à situação atual. Os diferentes módulos implementados, bem como o sistema de navegação, devem ser testados usando ferramentas de simulação como Gazebo e Rviz.

1.1.1 Objetivos específicos

Os objetivos específicos para realização deste trabalho são:

- Desenvolvimento de um ambiente de simulação utilizando ROS e Gazebo.

- Implementação de um método de navegação multi-controlador.
- Implementação de um algoritmo de processamento de nuvem de pontos para detecção de pontos de interesse.
- Implementação de um algoritmo de seleção de controlador.

2 REFERENCIAL TEÓRICO

Nesse capítulo serão abordados conceitos a respeito do robô TIAGo, ambiente de navegação Gazebo com *framework* ROS, laser LiDAR, processamento de imagem AprilTag, modelagem do robô, controlador de navegação IBVS e o controlador de espiral.

2.1 Robô TIAGo

O robô utilizado nesse trabalho é o TIAGo, mostrado na Figura 1, pois possui um conjunto de sensores no qual permite que ele execute diversas funções de captação de dados, manipulação e navegação. O TIAGo é um robô móvel projetado para trabalhar em ambientes internos e possui um tronco extensível e um braço manipulador para conseguir manusear objetos com cargas de até 3kg, quando totalmente estendido (ROBOTICS, 2022). O robô possui 12 graus de liberdade, sendo 7 graus de liberdade somente para o braço. Essa combinação entre o dorso e o braço permite que o robô alcance objetos tanto em altura próximas ao chão quanto em alturas mais elevadas próximas ao tamanho de uma mesa (ROBOTICS, 2022). Ele é comercializado pela empresa PAL Robotics e promete ser uma plataforma ideal para pesquisa, pois combina capacidades de mobilidade, percepção, manipulação, facilmente configurável e atualizável, e interação homem-robô para um objetivo específico: ser capaz de auxiliar em pesquisas (ROBOTICS, 2022).

Segundo o manual do usuário do TIAGo, sua base móvel possui um mecanismo de acionamento diferencial e contém um computador de bordo, baterias, conector de alimentação, três sonares traseiros, um painel de usuário, um painel de serviço e dois sensores de Wi-fi para garantir a conectividade sem fio. O laser presente no TIAGo é do modelo TIM561-2050101, tem um range de 0,05 a 10 metros, uma frequência de 15 Hz, um campo de visão de 180° e um ângulo de passo de 33°. A cabeça do robô conta com a presença de uma câmera RGB-D do modelo Xtion Pro Live, com campo de visão 58° H, 45° V, 70° D, com modos de fluxo de cores com QVGA 320x240 60 fps, VGA 640x480 30 fps, XGA 1280x720 15 fps e com um range de profundidade do sensor de 0,4 a 8 metros.

Atualmente, o TIAGo é utilizado como fonte de diversas pesquisas acadêmicas com temas a respeito de percepção, manipulação e navegação, juntamente com planejamento motriz, inteligência artificial e interação humano-robô, com foco nas áreas de aplicação em indústrias e inteligência das coisas (IoT) (ROBOTICS, 2022). Além dos benefícios citados para a escolha desse robô, ele possui uma plataforma de código aberto e compatível com ROS e sua simulação é compatível com os sistemas operacionais Linux, Mac e Windows.

Figura 1 – Robô *TIAGo*.

Fonte – ROBOTICS (2022).

2.2 Sistema ROS

Segundo Kerr e Nickels (2012) em sua pesquisa a respeito dos principais sistemas de operação para robôs, exitem vários sistemas para desenvolver o controle de robôs, porém o ROS (*Robot Operating System*) (Figura 2) é o que mais se destaca. Além disso, o autor menciona que o sistema ROS é a ferramenta mais utilizada entre os desenvolvedores atualmente. O ROS é um *framework* de robótica, ou seja, uma estrutura de *software* que une códigos comuns entre vários projetos de *software* provendo uma funcionalidade genérica contendo códigos base que permitem que um programador desenvolva aplicações mais complexas sem a necessidade de desenvolver estruturas simples e repetitivas para uma nova aplicação. Assim, um *framework* é uma camada de abstração de *software* que é executado sobre um sistema operacional completo. Ao contrário das bibliotecas, é o *framework* quem dita o fluxo de controle da aplicação, chamado de Inversão de Controle (WIKIPEDIA, 2021).

O ROS é um conjunto de bibliotecas de *software* de código aberto e ferramen-

Figura 2 – Logo do ROS.

Fonte – ROS (2022a).

tas que ajudam na construção de um projeto de robótica, no qual define componentes, interfaces e ferramentas para construir robôs, em que em sua maioria são constituídos de atuadores, sensores e sistemas de controles (cérebro do robô) (ROS, 2022b). Esse sistema de desenvolvimento de robô é uma estrutura distribuída de processos composto por uma quantidade de Nós (também conhecido como *Nodes*, em inglês), que são uma instância em execução de um programa e ao juntar todos os nós é formado um grafo, no qual permite que os executáveis sejam projetados individualmente e acoplados livremente em tempo de execução, e que se comunicam entre si pelo envio e recebimento de mensagens através de tópicos e RPC's (chamadas remotas à procedimentos) (ROBOTICS, 2014), como mostrado na Figura 3.

Figura 3 – Esquema de comunicação do ROS.

Fonte – Elaborado pelo autor (2022).

Com ROS é possível criar ambientes de simulação que pode ser facilmente utilizados por diferentes tipos de robôs, sendo possível realizar testes e simulações de forma mais simples e efetivas do que com robôs reais. A plataforma suporta interfaces de hardware compatíveis com diversos componentes, como câmeras, LiDARs, sensores, controladores de motor, entre outros, e fornece serviços de abstração de hardware, controle de dispositivo de baixo nível, implementação de funcionalidades comumente usadas, passagem de mensagens entre processos e gerenciamento de pacotes (ROS, 2022b).

O motivo para a seleção dessa plataforma de *software* se deu por ROS ter como objetivo a reutilização de código na pesquisa e desenvolvimento de robótica. O *software* possui uma independência de linguagem de programação e suas bibliotecas já implementas foram escritas em Python, C++ e Lisp, tendo algumas bibliotecas experimentais em Java e Lua (DATTALO, 2018). A linguagem escolhida para implementar o algoritmo de controle e processamento de imagem foi Python por ser uma linguagem popular e de utilização simplificada. Para ser possível utilizar o sistema do ROS, é necessário de plataformas baseadas em Unix, como sistemas Mac OS X e Ubuntu, tendo os demais sistemas ainda não explorados o suficiente para ser amplamente utilizados.

2.2.1 Rosgraph

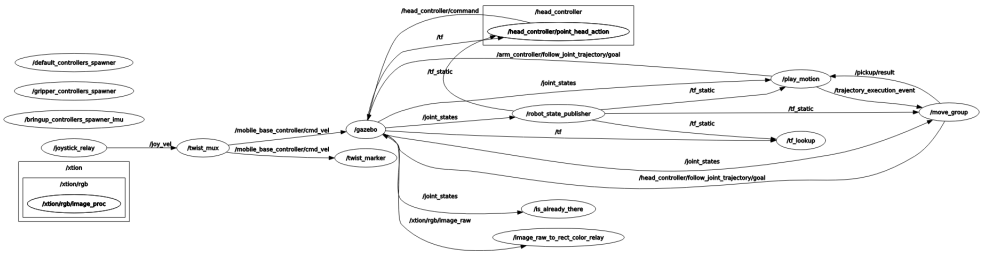
Para visualizar os nós e tópicos usados para controlar o robô é utilizado o Rosgraph, que é uma ferramenta do ROS que imprime informações sobre o *ROS Computation Graph* e fornece uma biblioteca interna que pode ser usada por ferramentas gráficas. Com Rosgraph, é possível visualizar o grafo que está sendo utilizado pelo robô, como mostrado na Figura 4, criado em ROS utilizado o comando rqt graph (COLEMAN, 2018). No grafo, a simbologia das elipses representam os nós criados e as setas, os tópicos, onde haverá a transferência de dados. No APÊNDICE **??**, está disponível a Figura 4 no sentido vertical para que possa ver em detalhes os nós e tópicos utilizados para a simulação.

Apesar de alguns dos nós e tópicos representados no grafo da Figura 4 serem disponibilizados pelo ROS para o controle do robô, nem todos eles são utilizados. Neste trabalho é realizado a inscrição no tópico /xtion/rgb/Image_raw para o processamento de imagem, /scan_raw para o processamento dos dados do laser e /joint_states para os dados de estado da cabeça do robô e também é realizado a publicação no tópico /mobile_base_controller/cmd_vel para o controle da base móvel do robô e, para o controle da cabeça do robô, no tópico head_controller/command .

2.2.2 Gazebo

Para ser possível realizar testes sem um robô real, é necessário usar uma ferramenta que simule o ambiente computacionalmente. O ambiente de simulação utilizado foi o Gazebo, que é simulador projetado que permite testar rapidamente algoritmos, projetar robôs, realizar testes de regressão e treinar sistemas de inteligência artificial usando cenários realistas (GAZEBO, 2014). O site oficial do Gazebo expõe que o ambiente oferece a capacidade de simular com precisão e eficiência populações de robôs em ambientes

Figura 4 – Nós do grafo utilizado para controlar o robô *TIAGo*.



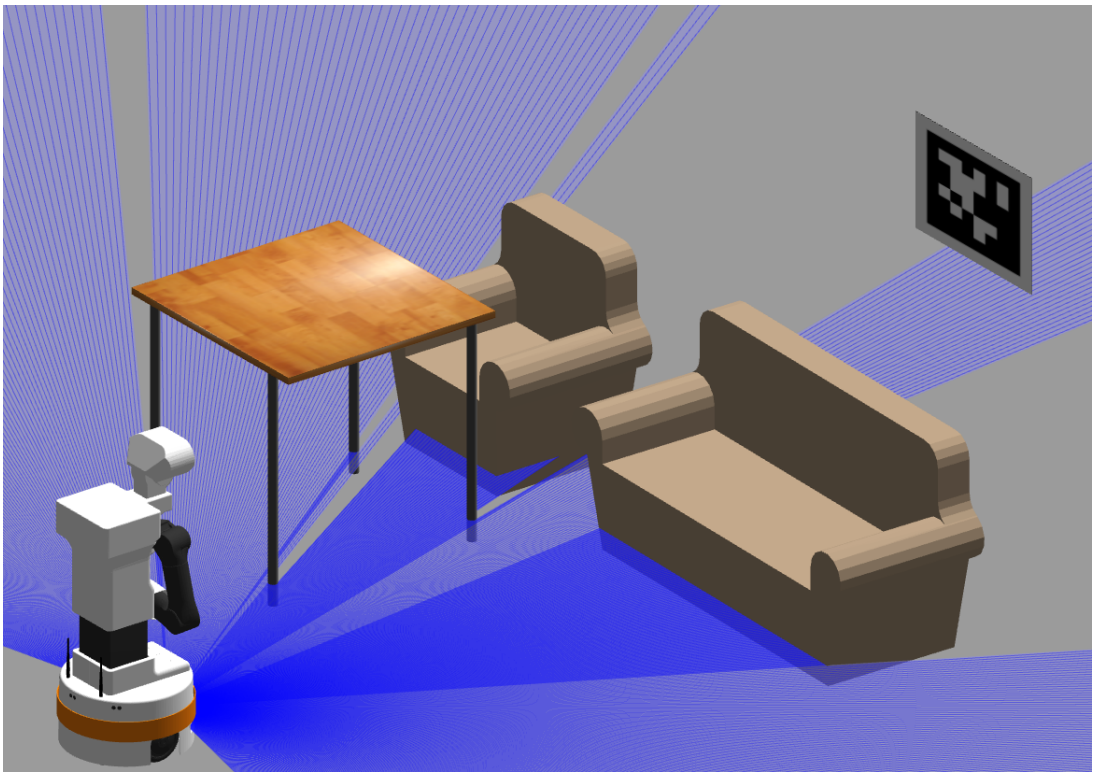
Fonte – Elaborado pelo autor (2022).

internos e externos complexos e possui um mecanismo de física robusto, gráficos de alta qualidade e interfaces gráficas e programáticas convenientes. Ainda segundo o site oficial Gazebo (2014), o desenvolvimento do ambiente de simulação de código aberto teve seu início em 2002 na Universidade do Sul da Califórnia com os criadores: Dr. Andrew Howard e seu aluno Nate Koenig, com objetivo de construir um simulador de alta fidelidade para simular robôs em ambientes externos sob diversas condições. A Figura 5 mostra a representação do robô TIAGo no ambiente de navegação do Gazebo, onde foram realizados todos os testes e análises dos resultados obtidos.

2.2.3 RVIZ

Para visualizar os resultados do tratamento dos dados do laser, se faz necessário a utilização de uma ferramenta de visualização 3D para aplicações ROS, sendo utilizado neste trabalho o RVIZ, que fornece uma visão do modelo do seu robô, captura informações dos sensores do robô e reproduz os dados capturados. Ele pode exibir dados da câmera, lasers, de dispositivos 3D e 2D, incluindo imagens e nuvens de pontos (RVIZ, 2022). Essa ferramenta de visualização funciona como um *subscriber*, onde irá "ler" as informações de uma mensagem inserida por um *publisher*. O RVIZ foi utilizado no trabalho para validar o resultado do controlador da espiral proveniente do processamento do laser. A Figura 6 mostra um teste de uma aplicação ROS realizado utilizando a ferramenta de visualização 3D RVIZ, onde o sistema de coordenadas em vermelho (eixo x), verde (eixo y) e azul (eixo z) representa o sistema de coordenadas do robô da simulação e a linha vermelha representa a captação dos dados do obstáculo pelo sensor.

Figura 5 – Robô TIAGo no ambiente de navegação Gazebo.

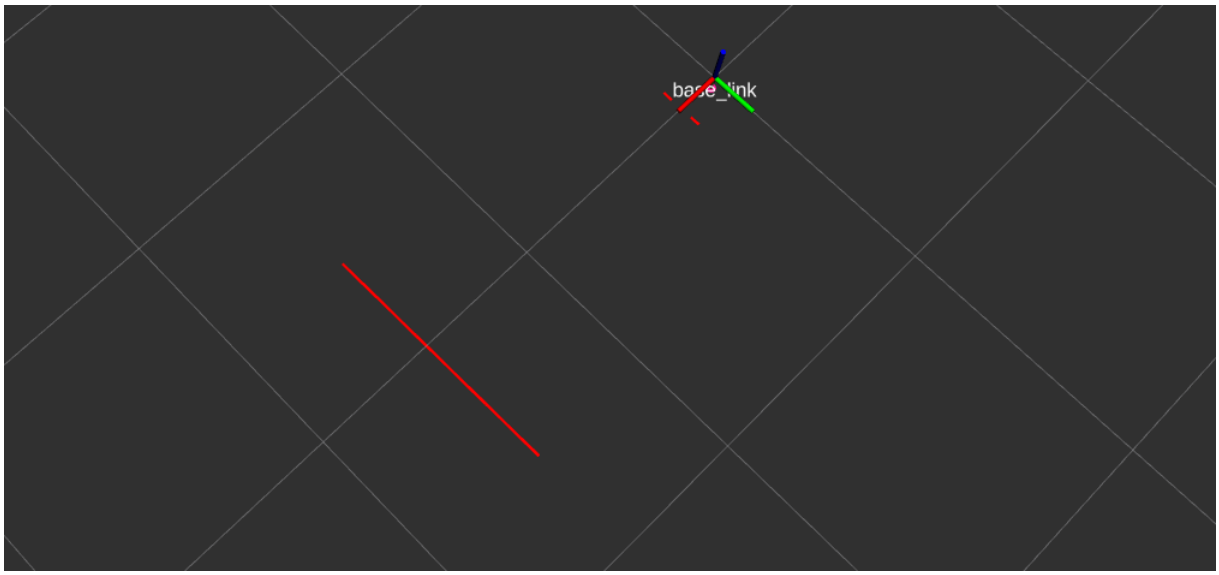


Fonte – Elaborado pelo autor (2022).

2.3 Sensor LiDAR

Para ser possível que a ferramenta de visualização 3D RVIZ consiga representar a posição dos obstáculos próximos ao robô, é necessário um sensor capaz de medir a distância até um objeto, sendo utilizado para este trabalho um laser com a tecnologia LiDAR. Segundo Areann et al. (2001), a tecnologia LiDAR *(Light Detection And Ranging)* é aplicada para medir distâncias até um objeto ou superfície os iluminando com raios laser (luz infravermelha invisível) e processando a onda de resposta refletida. O sensor LiDAR é utilizado para diversas aplicações, entre elas o levantamento topográfico de regiões em nuvens de pontos (para sensores de alto alcance), automação de processos industriais de armazenamento e aplicações balísticas militares (AREANN et al., 2001). Sensores medidores de distâncias, como o LiDAR, funciona utilizando o método de emissão de raios laser invisíveis em um objeto ou superfície, e realiza o processamento da onda de resposta refletida pelo objeto alvo, assim sendo possível calcular sua distância ao objeto.

Figura 6 – Representação de um obstáculo no RVIZ.



Fonte – Elaborado pelo autor (2022).

2.4 AprilTag

Para realizar o controle a partir do processamento dos dados da visão da câmera do robô, é necessário a utilização de um alvo. Neste trabalho é usado o sistema fiducial visual AprilTag, útil para uma ampla variedade de tarefas, incluindo realidade aumentada, robótica e calibração de câmeras. Os alvos podem ser criados a partir de uma impressora comum, e o *software* de detecção AprilTag calcula a posição 3D precisa, orientação e identidade das tags em relação à câmera (OLSON, 2010). A biblioteca AprilTag é implementada em C sem dependências externas e foi projetado para ser facilmente incluído em outros aplicativos, além de ser portátil para dispositivos embarcados, tendo seu desempenho em tempo real podendo ser alcançado mesmo em processadores de grau de telefone celular (OLSON, 2010). As tags fornecem um meio de identificação e posicionamento 3D, mesmo em condições de baixa visibilidade e agem como códigos de barras, armazenando uma pequena quantidade de informações (identificação da etiqueta), ao mesmo tempo em que permitem uma estimativa simples e precisa de poses 6D (x, y, z, ângulo chamado de rolagem *Roll*, ângulo chamado de arfagem *Pitch*, ângulo de guinada *yaw*) da tag (KNOWLEDGEBASE, 2018).

AprilTags são objetos de referência que são colocados no campo de visão da câmera quando uma imagem ou quadro de vídeo é capturado. A borda preta ao redor do

marcador torna mais fácil para algoritmos de visão computacional e processamento de imagem detectarem as AprilTags em uma variedade de cenários, incluindo variações de rotação, escala, condições de iluminação, entre outros. Esse marcador pode ser comparado a um *QR Code*, um padrão binário 2D que pode ser detectado usando algoritmos de visão computacional. No entanto, uma AprilTag contém apenas 4-12 bits de dados, várias ordens de magnitude menores que um código QR (um código QR típico pode conter até 3 KB de dados) (KNOWLEDGEBASE, 2018).

Na documentação oficial da biblioteca da AprilTags, como suas cargas úteis são tão pequenas, elas podem ser detectadas mais facilmente, identificadas de forma mais robusta e menos difíceis de detectar em intervalos mais longos. Segundo KNOWLEDGEBASE (2018), marcadores fiduciais, como AprilTags, são parte integrante de muitos sistemas de visão computacional, incluindo, entre outros: calibração de câmeras, estimativa do tamanho do objeto, medição de distância entre câmera e um objeto, posicionamento 3D, orientação de objeto e navegação robótica de forma autônoma para um objetivo marcado.

Os marcadores da biblioteca da AprilTags possui propriedades como:

- Possui o fundo preto.
- O primeiro plano é um padrão gerado na cor branca.
- Há uma borda preta ao redor do padrão.
- Os quadrados possuem valores binários.
- Podem ser criados em diversos tamanhos.

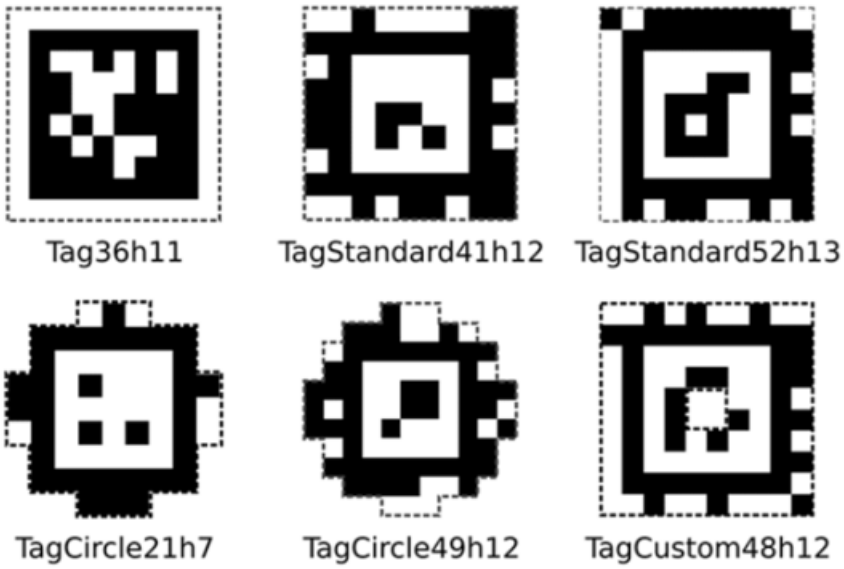
Alguns problemas podem ser citados na utilização da biblioteca dos AprilTags, tais como: o movimento e a distância até a tag afetam a precisão da estimativa de posição. Uma alternativa para AprilTags para localização/estimativa de posição de um objeto é a tag AR, que foi criada para realidade aumentada, mas pode fornecer estimativa de posição semelhante.

O benefício de usar AprilTags é a precisão aprimorada, especialmente em condições de iluminação menos ideais. Um dos problemas com a execução do nó AprilTag é que a saída do quadro depende da orientação do movimento do corpo. Logo, se faz necessário o controle da cabeça do robô para que acompanhe o movimento do corpo.

Para serem gerados é necessário apenas um *software* básico e uma impressora, pois basta gerar a AprilTag em seu sistema, imprimi-la e incluí-la em seu pipeline de processamento de imagem. Uma família em AprilTags define o conjunto de tags que o detector AprilTag assumirá na imagem de entrada. A família AprilTag padrão é “Tag36h11”; no

entanto, há um total de seis famílias em AprilTags (ROSEBROCK, 2020), como mostrado na Figura 7 . Para esse trabalho, foi utilizado a família padrão “Tag36h11” da biblioteca do AprilTag, utilizando Python para a captação e processamento de imagem.

Figura 7 – Famílias de AprilTags.

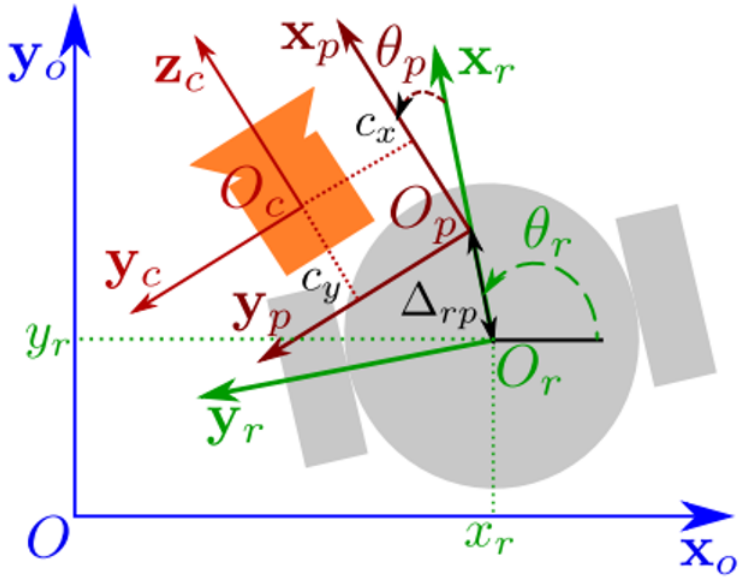


Fonte – Rosebrock (2020).

2.5 Modelagem do robô

Neste trabalho é utilizado a modelagem do robô representado pela Figura 8. O robô é representado pela cor cinza, tendo seu sistema de coordenadas representado pela cor verde, a cabeça do robô representada pelo sistema de coordenadas vermelho, em laranja é representado sua câmera e em azul o sistema de coordenadas global . O vetor posição do robô é definido como $F_r = (O_r, X_r, Y_r)$, onde X_r e Y_r são as coordenadas do robô no sistema de coordenadas O_r e sua orientação da sua base móvel é dada por θ_r e a orientação da cabeça é dada por θ_p. É definido v como velocidade linear da base móvel, ω_r como velocidade angular da base móvel e ω_p como velocidade angular da plataforma da cabeça.

Figura 8 – Modelagem do robô.



Fonte – Elaborado pelo autor (2022).

2.6 Controlador de navegação IBVS

O servomotor visual é uma técnica de controle de robôs usando visão em malhas de controle de realimentação. Comumente é classificado em servo visual baseado em posição *(PBVS)* e servo visual baseado em imagem IBVS (GAO; PIAO; TONG, 2012).

O PBVS se baseia na extração de características da imagem para calcular a posição e orientação do objeto em relação a câmera. Para esse tipo de controle, é utilizado a estimativa dos parâmetros empregados na definição do erro, definido como a diferença entre a posição e orientação atual e a desejada, e exige o conhecimento prévio da geometria do alvo e das características internas da câmera (MUÑOZ, 2011).

No IBVS, é possível controlar o movimento do robô a partir de uma informação visual bidimensional. A estratégia de controle visa reduzir as discrepâncias entre a imagem atual e a imagem desejada. A utilização de um servo visual baseado em imagem se faz necessário principalmente pela sua característica de baixa sensibilidade a erro de calibração de câmera e erro de medição de imagem, para garantir uma boa precisão de

posicionamento e rastreamento do robô (CHAUMETTE; HUTCHINSON, 2007).

Para construir o cálculo do sinal de controle por IBVS, são necessários algumas considerações feitas pelo autor Chaumette e Hutchinson (2007). Definimos r como a pose da câmera (posição e orientação) e

$$\dot{r} = (T_x, T_y, T_z, W_x, W_y, W_z)^T, \tag{1}$$

como o torsor cinemático da câmera, onde T_x, T_y e T_z são as velocidades lineares ao longo dos eixos x, y e z da câmera, e W_x, W_y e W_z são as velocidades angulares em torno dos eixos x, y e z da câmera.

As coordenadas no plano da imagem de um ponto P_i são definidas como:

$$p_i = (u_i, v_i)^T, \tag{2}$$

onde u_i e v_i representam a abscissa e a ordenada no plano da imagem. Assim, obtemos

$$\dot{p}_i = (\dot{u}_i, \dot{v}_i)^T, \tag{3}$$

como a velocidade correspondente. A velocidade de um ponto no plano da imagem e o torsor cinemático da câmera são ligados pela matriz jacobiana da imagem J_i como:

$$\dot{p}_i = J_i(p_i, r)\dot{r}, \tag{4}$$

onde

$$J_i = \begin{bmatrix} \frac{\lambda}{z_i} & 0 & \frac{-u_i}{z_i} & \frac{-u_i v_i}{\lambda} & \frac{\lambda^2+u_i^2}{\lambda} & -v_i \\ 0 & \frac{\lambda}{z_i} & \frac{-v_i}{z_i} & \frac{-\lambda^2-v_i^2}{\lambda} & \frac{u_i v_i}{\lambda} & u_i \end{bmatrix}, \tag{5}$$

em que λ é a distância focal da câmera e z_i a distancia do ponto P_i no quadro da câmera. Para um número de pontos n, definimos o vetor de informações visuais como:

$$s = [p_1^T, p_2^T, ..., p_n^T]^T, \tag{6}$$

e

$$\dot{s} = [\dot{p}_1^T, \dot{p}_2^T, ..., \dot{p}_n^T]^T, \tag{7}$$

A abordagem utilizada no trabalho para o controlador IBVS se baseia em usar como lei do controle:

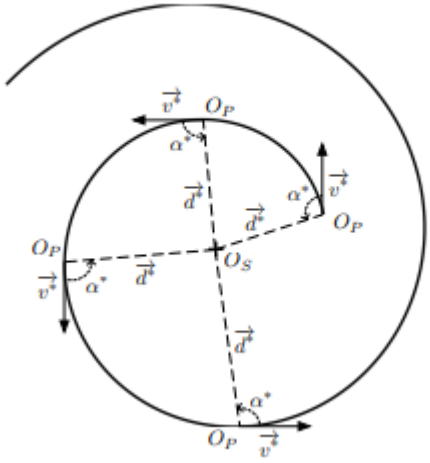
$$\dot{r} = kJ^+(s - s^*), \tag{8}$$

onde s^* são as coordenadas de referência, k é uma matriz de ganho, $J = [J_1^T, J_2^T, ..., J_n^T]^T$ e J^+ a pseudo-inversa de J.

2.7 Controlador de espiral

De acordo com MATHEMATICS (2011), as espirais 2D são curvas planas que giram em torno de um ou mais pontos, movendo-se em direção ou longe dele(s). Esta interpretação do termo não é uma definição estrita. Distingue-se dois tipos: espirais algébricas e pseudo-espirais. Segundo Boyadzhiev (1999), uma espiral pode ser descrita pelo ponto O_p movendo-se em um plano em relação a um ponto fixo O_s, como mostrado na Figura 9.

Figura 9 – Modelo de espiral.



Fonte – Renak et al. (2019).

$\alpha^*(t)$ é a orientação do ângulo entre $\vec{v}^*$ e $\vec{d}^*$ (RENAK et al., 2019).
De acordo com Renak et al. (2019), $\vec{d}^*$ pode ser definido como:

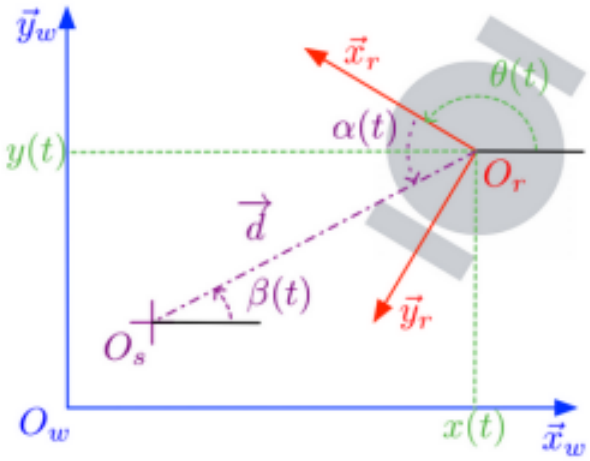
$$\vec{d}^* = -v^* \cos(\alpha^*). \tag{9}$$

A partir da Equação 9, nota-se que o valor da distância entre O_s e O_p só depende do valor de α^*.

- $0 \leq \alpha^* < \frac{\pi}{2}$ ou $0 \leq \alpha^* < -\frac{\pi}{2}$ (A espiral tem distância crescente).
- $\frac{\pi}{2} < \alpha^* \leq \pi$ ou $\frac{-\pi}{2} < \alpha^* \leq -\pi$ (A espiral tem distância decrescente).
- $\alpha^* = \pm\frac{\pi}{2}$ (A espiral vira um círculo).

Logo, é possível desenvolver um método de controle de navegação para um robô a partir do cálculo de centro da espiral *(Spiral Center Position SCP)* a fim de evitar colisão em obstáculos, sendo necessário, apenas, a seleção do ângulo α^* e calculando a distância ao centro da espiral (RENAK et al., 2019). Nesse trabalho, é utilizado o controlador proposto em Durand-Petiteville et al. (2017), que prova ser possível fazer o veículo seguir uma espiral dedicada para o modelo de robô da Figura 10. Considerando α e d obtidos por dados de laser, a espiral é projetada para garantir a prevenção de obstáculos.

Figura 10 – Modelo de robô.



Fonte – Renak et al. (2019).

Para adaptar o modelo do robô ao modelo da espiral, Renak et al. (2019) define $\vec{d} = \vec{O_s O_r}$ e $\alpha(t)$ como ângulo entre $\vec{X}_r$ e $\vec{d}$. Fazendo com que α e d convirjam para α^* e d^*, é possível fazer o robô seguir uma espiral dedicada. Para isso, Renak et al. (2019) projeta uma lei de controle que permite fazer (α, d) convergir para (α^*,d^*).

Para chegar na lei de controle, Renak et al. (2019) define:

$$\dot{d}(t) = -v(t)\cos(\alpha(t)) \tag{10}$$

e

$$\dot{\alpha}(t) = -\dot{\theta}(t) + \dot{\beta}(t) = -w(t) + \frac{v(t)}{d(t)}\sin(\alpha(t)). \tag{11}$$

O controlador para o robô seguir a espiral, segundo Renak et al. (2019), é dado por:

$$w(t) = \lambda_s E_s(t) + \frac{v(t)}{d(t)} \sin(\alpha(t)) - \alpha_d \dot{\varepsilon}(t), \tag{12}$$

onde v é a velocidade linear fixa e diferente de zero e λ_s é um vetor escalar positivo, $E_s(t)$ é a parcela de erro. O erro é definido pela parcela de α e de d, como:

$$E_s(t) = \alpha(t) - \alpha^* - \alpha_d \varepsilon(t), \tag{13}$$

onde

$$\varepsilon(t) = sign(d^*(t) - d(t)) \frac{\min(\ ||d^*(t) - d(t)\ ||, n)}{n} \tag{14}$$

representa o erro normalizado entre $d^*(t)$ e d(t) e varia entre ± 1 e α_d é definido como:

$$\alpha_d = \begin{cases} sign(\alpha^*) * \pi - \alpha^*, & se \ \ d^*(0) > d(0) \\ \alpha^*, & se \ \ d^*(0) > d(0) \end{cases}$$

3 METODOLOGIA

Este capítulo descreve a ideia principal do trabalho, quais foram as métodos utilizados, os critérios de seleção para agrupar obstáculos, seleção dos pontos importantes e alternância do controlador e como ocorre sua utilização. Para isso é mostrado a descrição dos tópicos e nós utilizados e todo o fluxo de dados, tendo a linguagem Python como a linguagem utilizada nesse trabalho, por possuir uma ênfase em facilitar a leitura do código e possuir uma licença *open-source*. Outra vantagem em relação à outras linguagens suportadas pelo ROS é que, utilizando Python é possível escrever o mesmo código utilizando menos linhas, além de permitir programas limpos tanto em pequena quanto em grandes escalas (ROSSUM et al., 2007).

3.1 Processamento dos dados do laser

Para auxiliar no processamento dos dados do laser, foi utilizado o tópico */scan_raw* disponibilizado pelo próprio *ROS*. Ao realizar a assinatura no tópico do /scan_raw, o ROS disponibiliza os valores detectados pelo laser em coordenadas polares, d e θ. Primeiro, temos que agrupar os pontos pertencentes a um mesmo obstáculo. Para isso, é necessário decidir o critério de seleção para identificação de um obstáculo de acordo com a distancia entre dois pontos captados pelo laser. Nesse critério de seleção é definido um limite usado para agrupar nuvens de pontos de 15cm, ou seja, se $||d(k)-d(k+1)||>15$cm significa que possui mais de um objeto. Tendo o critério de seleção dos obstáculos definido, se faz necessário definir o SCP (sigla em inglês para posição do centro da espiral). Segundo Leca et al. (2019), a evolução do SCP depende do movimento do robô e de dados do LiDAR e a escolha do SCP garante:

- A não colisão e a segurança do robô.
- Obstáculos de várias formas (convexas ou côncavas, com ângulos, etc.) e tamanhos (pequenos ou grandes) são tratados em uma maneira igual.
- Uma trajetória suave do robô em torno do obstáculos com importantes variações de forma, o que significa, por sua vez, que as potenciais partes desconhecidas do obstáculo deve ser levado em consideração.
- O controle de entrada obtido deve ser adaptado à cinemática do robô.

O método utilizado nesse trabalho para seleção de O_s, centro da espiral, foi o método proposto por Leca (2021), que consiste no 4 passos a seguir:

- Passo 1: Após adquirir os pontos captados pelo LiDAR em volta do robô, o ponto mais próximo a ele na norma cartesiana é calculado e chamado de O_c.

- Passo 2: Para se obter uma distância de segurança maior ao redor do robô, também é calculado o baricentro, chamado O_b, calculado após captar todos os pontos do LiDAR pertencentes a um mesmo obstáculo.

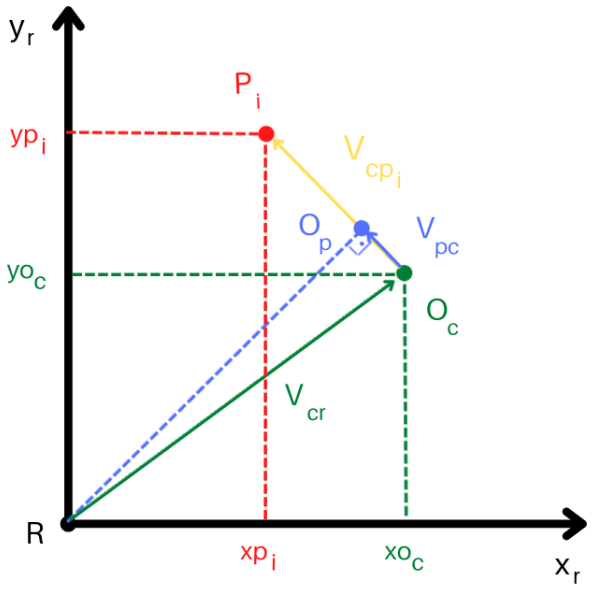
- Passo 3: Para cada ponto P_i pertencente a um mesmo obstáculo é calculado a projeção ortogonal da posição do robô O_{p_i} sobre o vetor $\vec{O_cP_i}$. Caso o O_{p_i} pertencer ao vetor, é selecionada, entre todas projeções ortogonais pertencentes ao vetor, a que mais se aproxima do robô na norma cartesiana, e denominada como O_p. Se o O_{p_i} projetado estiver fora do vetor, $O_p = O_c$.

  Para o cálculo de O_p, foi utilizado o sistema de coordenadas centrado no robô, como é mostrado na Figura 11, onde estão representadas as coordenadas do ponto O_c e do ponto P_i, e o vetores $\vec{V_{cp_i}}$, $\vec{V_{cr}}$ e $\vec{V_{pc}}$, sendo eles o vetor com os pontos O_c e P_i, o vetor com os pontos O_r (centro do robô, considerado na origem) e O_c e o vetor perpendicular a reta $\vec{O_cP_i}$, respectivamente. A partir dos vetores $\vec{V_{cp_i}}$ e $\vec{V_{cr}}$ calculados, é possível encontrar o vetor $\vec{V_{pc}}$ a partir do produtor vetorial entre $\vec{V_{cp_i}}$ e $\vec{V_{cr}}$. Portanto, tendo a posição de O_p calculada, é realizada a verificação para checar se o ponto encontrado pertence à reta $\vec{O_cP_i}$. Caso O_p não pertença a reta, a posição considerada para ele será igual à posição de O_c.

- Passo 4: A seleção do centro da espiral O_s é dada pelo ponto mais próximo do robô entre O_c, O_p e O_b.

Para ser possível comprovar os passos descritos, Leca (2021) propôs o exemplo representado na Figura 12 onde mostra as etapas para a seleção do centro da espiral a partir de O_c e O_p para um caso de obstáculo côncavo (polígono ABCDEFG) com uma largura de concavidade menor que $2d^*$. Para o passo 1, após ser calculado os pontos retornado pelo LiDAR, obtêm um resultado que o ponto mais próximo ao robô é o ponto B (mostrado em vermelho), logo O_c = B. Leca (2021) utiliza o passo 3 para determinar O_p, sendo que o ponto projetado mais próximo ao robô está no segmento que liga O_c ao ponto na outra extremidade da concavidade, próximo ao ponto E. Logo, quando o robô evita o obstáculo, o ponto O_c mais próximo no intervalo de tempo t permanece quase coincidente com o ponto B. Com os pontos O_c e O_p calculados, é possível selecionar o centro da espiral, segundo o passo 4, que se dá pelo ponto mais próximo entre eles, que no exemplo descrito é sempre escolhido como o projetado sobre este segmento $\vec{BE}$. Segundo Leca (2021), do ponto de vista do método de rastreio em espiral, que apenas

Figura 11 – Sistema de coordenadas para cálculo de O_p.

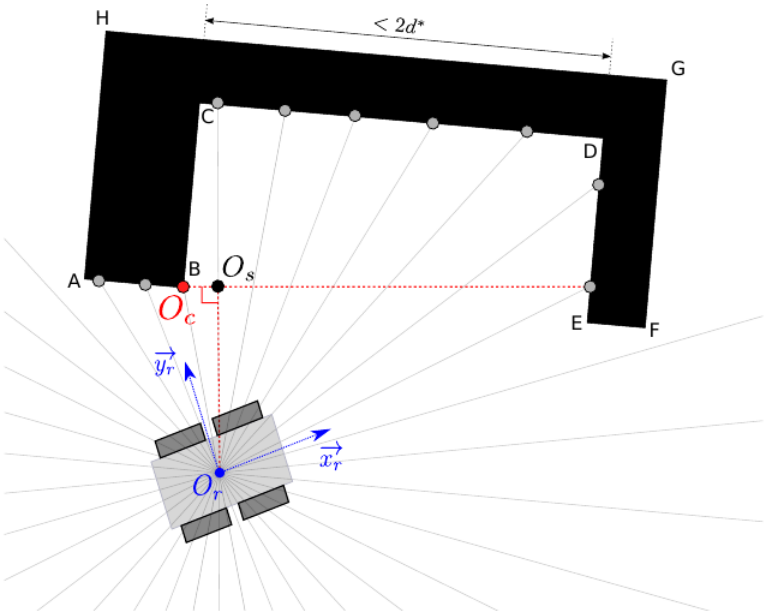


Fonte – Elaborado pelo autor (2022).

requer a informação O_s, o comportamento para este obstáculo côncavo como para um obstáculo convexo semelhante ao polígono AFGH é semelhante.

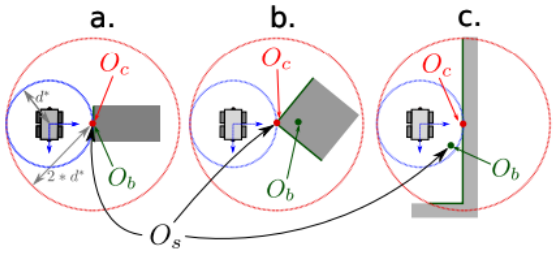
Leca et al. (2019) também propôs um exemplo para comprovar a metodologia para obtenção de O_s a partir do cálculo do baricentro, representado pela Figura 13, onde mostra alguns exemplos de casos destacando o mesmo método em casos côncavos e convexos. Em verde são exibidos os pontos do laser LiDAR, enquanto os círculos azul representa o círculo de raio d^* centrado no centro do robô e o círculo vermelho representa o círculo com raio $2d^*$ centrado em O_c, ambos utilizados para o cálculo de O_b. No caso (a.). é calculado a projeção ortogonal do robô no obstáculo convexo, O_c. Logo após, O_b é calculado como o baricentro de todos os pontos que estão dentro do círculo vermelho. Neste caso, como só existe uma possibilidade para O_b, o que coincide com Oc, logo O_s = O_b = O_c. Já para o caso (b.), o ponto mais próximo ao obstáculo (O_c) seria a quina do polígono, e o cálculo do baricentro resultaria num ponto dentro dele, consequentemente o ponto mais próximo ao robô entre O_c e O_b seria O_c. Logo, para o caso (b.), O_s = O_c. Em relação ao caso (c.), O_c seria na parede do obstáculo e os pontos verdes situados nos círculos vermelhos são então considerados para deduzir o baricentro e são usados para calcular ponto O_b. Como O_b calculado está mais próximo ao robô do que o ponto O_c, o centro de espiral O_s para esse caso seria O_s = O_b.

Figura 12 – Exemplo de cálculo de O_s para um obstáculo côncavo.



Fonte – Leca (2021).

Figura 13 – Exemplos de cálculo de O_s para diferentes tipos de obstáculos.



(a.) Obstáculo reto. (b.) Obstáculo convexo. (c.) Obstáculo côncavo.

Fonte – Leca et al. (2019).

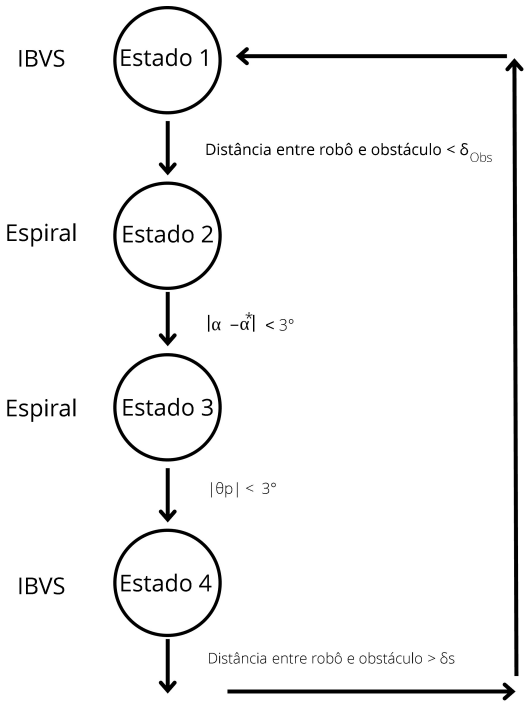
3.2 Critério de mudança de controladores

Os controladores do robô para seguir a espiral e o IBVS são utilizados nesse trabalho para realizar o movimento do robô até o alvo final selecionado. Para isso, é necessário um nó com critério pré definido que alterne entre o controlador de seguir a espiral e o IBVS quando necessário. Esse nó foi chamado de *MixerContoller* e determina um critério de

entrada e saída com distância fixa e outro critério de entrada e saída onde é baseado na previsão da distância, ambos em um ciclo de estado finito.

3.2.1 Critério de seleção com distância fixa

Figura 14 – Máquina de estados para o critério de seleção com distância fixa.



Fonte – Elaborado pelo autor (2022).

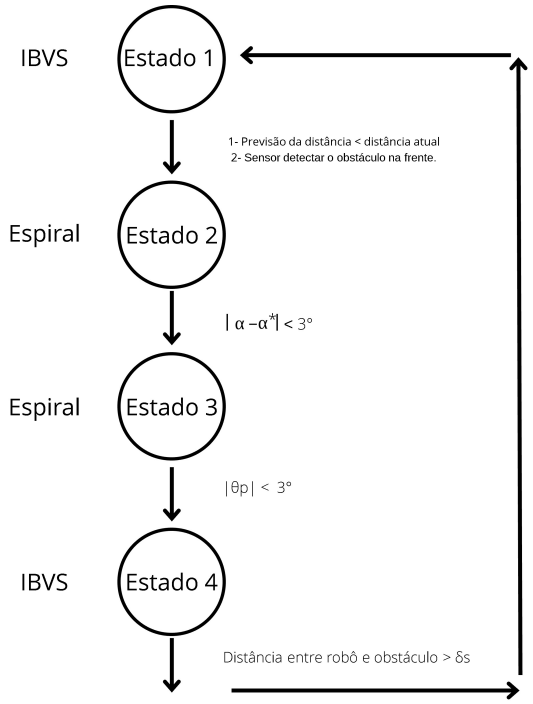
O primeiro critério de seleção utilizado para o trabalho, tendo sua máquina de estados representada na Figura 14, se baseia em avaliar a distância do centro da espiral ao robô, onde tem como estado inicial do ciclo o controlador IBVS. Caso a distância entre o obstáculo e o robô seja menor que $\delta_{Obs} = 0,8$ metros, é alterado para o controlador seguidor de espiral onde insere o robô na espiral, sendo definido como segundo estado. O estado dois é mantido até que o robô esteja presente na trajetória da espiral, evitando, assim, oscilação entre os controladores. Após a identificação do robô na espiral, que ocorre quando $|\alpha - \alpha^*| < 3°$, o estado três é acionado mantendo o controlador seguidor de espiral até que ocorra o fenômeno de alinhamento da cabeça do robô com seu corpo, que ocorre quando $|\theta_p| < 3°$, já que durante toda a navegação a cabeça do robô aponta para a AprilTag, e então é selecionado o quarto estado com o controlador IBVS. Para fechar o

ciclo, o critério para voltar para o estado inicial é estabelecido um limiar de distância maior que δ_s para impedir que entre em uma nova espiral.

O controlador seguidor de espiral permite apenas realizar o controle da base móvel do robô. Eventualmente quando o obstáculo não for mais considerado como perigoso e realizar a troca para o controlador IBVS, é estritamente necessário que o alvo se mantenha visível pela câmera em todo o movimento. Para isso, durante a execução do seguidor de espiral é utilizado uma versão simplificada do IBVS apenas para realizar o controle da cabeça do robô e impedir que perca o obstáculo do campo de visão. O design do controlador simplificado do IBVS para o controle da cabeça foi baseado no trabalho de Petiteville et al. (2011).

3.2.2 Critério de seleção baseada na previsão da distância

Figura 15 – Máquina de estados para o critério de seleção baseada na previsão da distância.

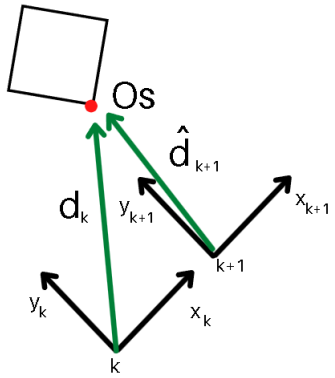


Fonte – Elaborado pelo autor (2022).

Já o segundo critério, utiliza de um método de previsão da próxima distância captada pelo sensor após a utilização do controlador, tendo sua máquina de estados representada na Figura 15. Para esse critério foi utilizada a matriz de transformação homogênea para calcular a distância $\hat{d}(k+1)$ para o próximo ponto do obstáculo, representada

na Figura 16. Os critérios de entrada e saída de ambos os controladores são similares, exceto pela condição de mudança do primeiro estado para o segundo. Para esse critério, são utilizados duas condições de mudança entre o estado 1 e 2: a previsão da distância seja menor que a distância atual e o sensor detectar o obstáculo na frente.

Figura 16 – Representação dos sistema de coordenadas do robô antes e depois do controlador.



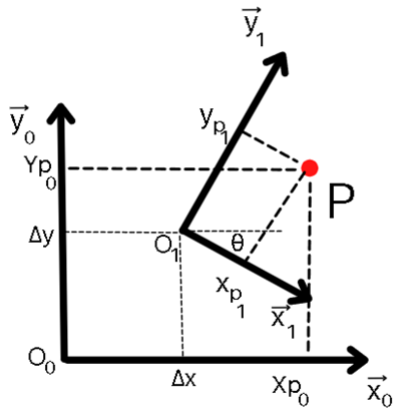
Fonte – Elaborado pelo autor (2022).

Para obter a distância do centro da espiral para sistema ”virtual”($X_1\,Y_1$) é necessário a utilização da matriz de transformação homogênea. A matriz de transformação é utilizada para realizar transformações lineares arbitrárias com objetivo de serem exibidas em um formato consistente, apropriado para cálculos. Segundo Lamas W.; Groh (2002), a matriz de transformação homogênea é uma matriz a qual traça um vetor de posição expresso em coordenadas homogêneas a partir de um sistema de coordenadas para outro sistema de coordenadas. É representado na Figura 17 o sistema de coordenadas para utilização da matriz de transformação homogênea, onde é necessário definir:

$$T_{01} = \begin{bmatrix} \cos\theta & -\sin\theta & \Delta X \\ \sin\theta & \cos\theta & \Delta Y \\ 0 & 0 & 1 \end{bmatrix}, \tag{15}$$

$$P_0 = \begin{bmatrix} x_{p_0} \\ y_{p_0} \\ 1 \end{bmatrix}, \tag{16}$$

Figura 17 – Transformação de sistema de coordenadas.



Fonte – Elaborado pelo autor (2022).

$$P_1 = \begin{bmatrix} x_{p_1} \\ y_{p_1} \\ 1 \end{bmatrix}, \tag{17}$$

$$P_0 = T_{01} P_1, \tag{18}$$

$$P_1 = T_{01}^{-1} P_0, \tag{19}$$

$$\Delta X = \frac{v}{w} \sin(w_r t_s), \tag{20}$$

$$\Delta Y = -\frac{v}{w} \cos(w_r t_s - 1), \tag{21}$$

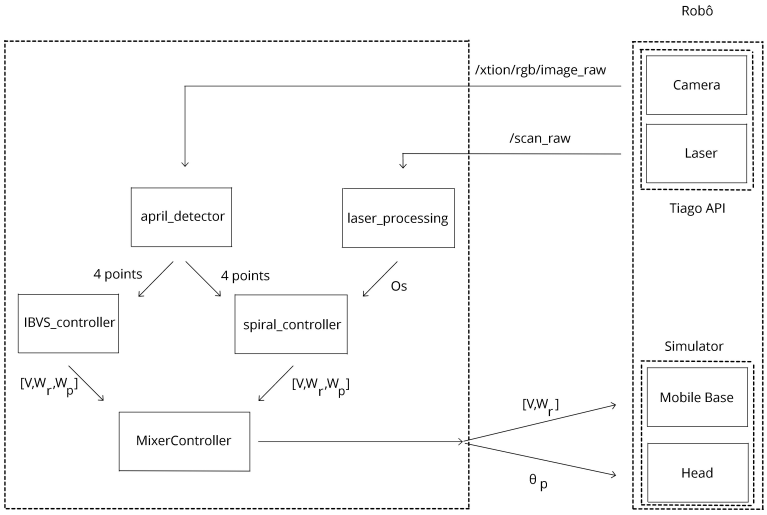
onde T_{01} é a matriz de transformação homogênea, P_0 é o ponto selecionado no sistema de coordenadas $X_0 Y_0$, P_1 é o ponto selecionado no sistema de coordenadas $X_1 Y_1$, θ é o ângulo de rotação do sistema de coordenada, v é velocidade linear da base móvel, w

é velocidade angular da base móvel, t_s é período de amostragem, ΔX e ΔY são a variação das coordenadas x e y, respectivamente (DURAND-PETITEVILLE et al., 2017). Assim, após obter os pontos da previsão da posição do robô com a transformação homogênea, é possível calcular a distância entre a posição virtual do robô e o centro da espiral. O critério de seleção utilizado compara a distância d_k atual e a distância $\hat{d}(k+1)$ virtual e caso $d_k \leq \hat{d}(k+1)$ o controlador a ser utilizado será o IVBS, caso contrário o controlador utilizado será o controlador do seguir da espiral.

3.3 Implementação no ROS

O grafo de fluxo de controle para a implementação da simulação do robô TIAGo no Ros é mostrada na Figura 18. Para alcançar o objetivo de controlar o robô a partir do método do controle da espiral, a problemática foi dividida nos seguintes passos:

Figura 18 – Esquema de organização do trabalho.



Fonte – Elaborado pelo autor (2022).

- Processamento do laser: Os dados obtidos pelo laser são disponibilizados no tópico /scan_raw do ROS e são utilizado pelo *laser_processing* (nó utilizado para o processamento dos dados do laser) por meio de inscrição. A Figura 19 mostra o diagrama de fluxo de controle para o processamento de laser, tendo como entrada os dados do laser inscritos no tópico $/scan\_raw$. Após obtenção dos dados, é chamada a função *callback_laser* para salvar em memória os dados de distância e ângulo do obstáculo. Com a posse dos dados das coordenadas polares, é estabelecido um critério de

agrupamento de obstáculo na função *mergeObstacles*, onde define que caso dois obstáculos estejam a uma distância menor que δ_{2o}, o algoritmo deverá considerar apenas 1 obstáculo. Tendo as distinções dos obstáculos, é necessário identificar qual deles está mais próximo ao robô para que possa ser calculado O_c, O_b, O_p e O_s, e esse processo é realizado na função *nearestObstacle*. Após ter o obstáculo mais próximo ao robô, é hora de realizar o cálculo dos centros, começando pelo O_c onde a função $calculate\_O_c$ calcula o ponto mais próximo ao robô, logo após é selecionado O_b na função $calculate\_O_b$, calculando o baricentro do obstáculo, e então calcula a posição de O_p em $calculate\_O_p$, considerando todos as possíveis retas contendo O_c e os pontos do obstáculo, como representado na Figura 12, calculando, assim, todas as distâncias perpendiculares a essas retas e pegando a distância mais próxima ao robô como O_p. Após calculados O_c, O_b e O_p, a função $calculate\_Os$ verifica qual é a distância entre o centro do robô e os 3 pontos e seleciona O_s como a menor distância. Logo, após a função $calculate\_O_s$ já é alcançado o objetivo do processamento do laser, só restando realizar a publicação do valor de O_s, através da função *pubMarker*.

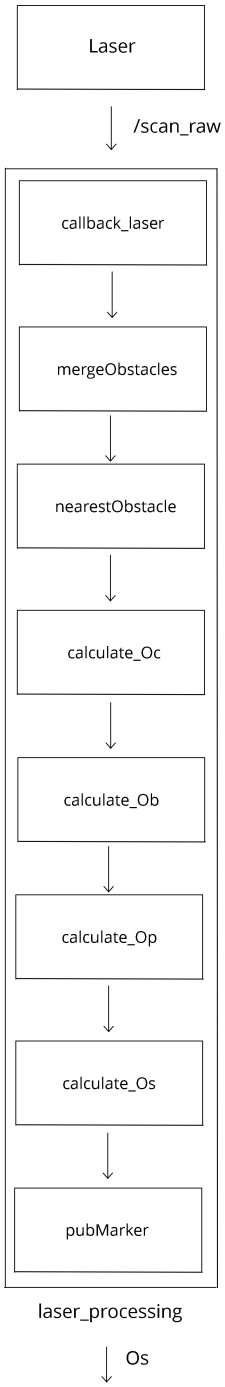
- Processamento de dados da câmera: Para processar os dados obtidos pela câmera do robô da simulação, é possível se inscrever no tópico $/xtion/rgb/image\_raw$, disponibilizado pelo ROS. Essa inscrição é realizada pelo nó do *april_detector*, no qual foi disponibilizado para prosseguimento do trabalho. O nó do *april_detector* publica 4 pontos, onde cada ponto representa a coordenada de um vértice da tag utilizada, onde serão utilizados nos controladores de espiral e no IVBS.

- Controlador de espiral: Para realizar o controle do robô segundo o método do centro da espiral (equação 12), é utilizado o nó $spiral\_controller$, disponibilizado para aplicar no trabalho. O $spiral\_controller$ se inscreve nos tópicos publicados pelo $april\_detector$ e $laser\_processing$ para ter acesso aos valores das coordenadas da tag e do centro da espiral O_s. Após o processamento dos dados, esse nó calcula e publica os valores de v, ωr e ωp.

- Controladores IVBS: Para realizar o controle do robô seguindo o método do IVBS (equação 8), é utilizado o nó $IBVS\_controller$, onde se inscreve no tópico que contêm os 4 pontos das coordenadas da *tag* disponibilizados pelo *april_detector* e também retorna a publicação de valores para v, ωr e ωp. O nó do $IBVS\_controller$ também foi disponibilizado para aplicar no trabalho.

- Seleção do controlador: Para realizar a seleção de qual controlador se encaixa me-

Figura 19 – Diagrama de controle de fluxo do laser_processing.

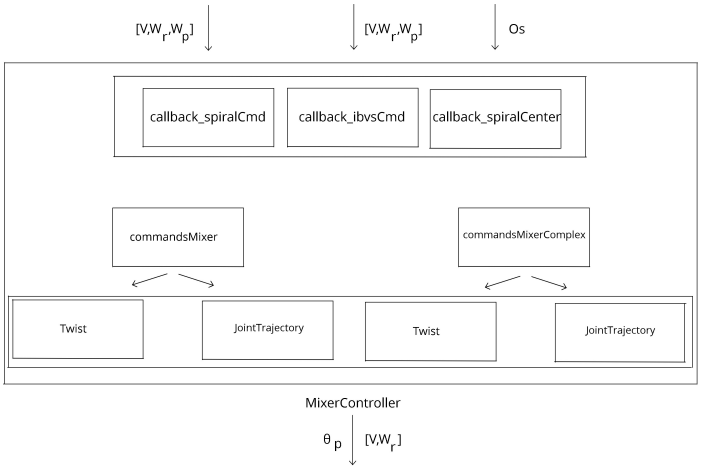


Fonte – Elaborado pelo autor (2022).

lhor para o robô alcançar o destino sem ter uma rota perigosa, são utilizados dois critérios como entrada e saída dos controladores, descritos na seção 3.2. A Fi-

gura 20 mostra o diagrama de controle de fluxo do *MixerController*, no qual a função *callback_spiralCmd* salva na memória os valores de v, ωr e ωp provenientes do *IBVS_controller* e a função *callback_ivbsCmd* salva os dados de v, ωr e ωp provenientes do *spiral_controller*. O valor de O_s também é utilizado por esse nó, sendo salvo na memória na função *callback_spiralCenter*, após se inscrever no tópico publicado pelo nó do *laser_processing*. Após ter todos os dados necessários na memória, é possível escolher entre os dois critérios implementados, o *CommandMixer* e o *commandMixerComplex*. Ambos os critérios retornam um *Twist* e um *JoinTrajecory*, ambos disponibilizados pelo ROS para realizar o controle do robô, sendo o *Twist* para controlar a base do robô e o *JoinTrajecory* para controlar a cabeça.

Figura 20 – Diagrama de controle de fluxo do MixerController.



Fonte – Elaborado pelo autor (2022).

4 RESULTADOS

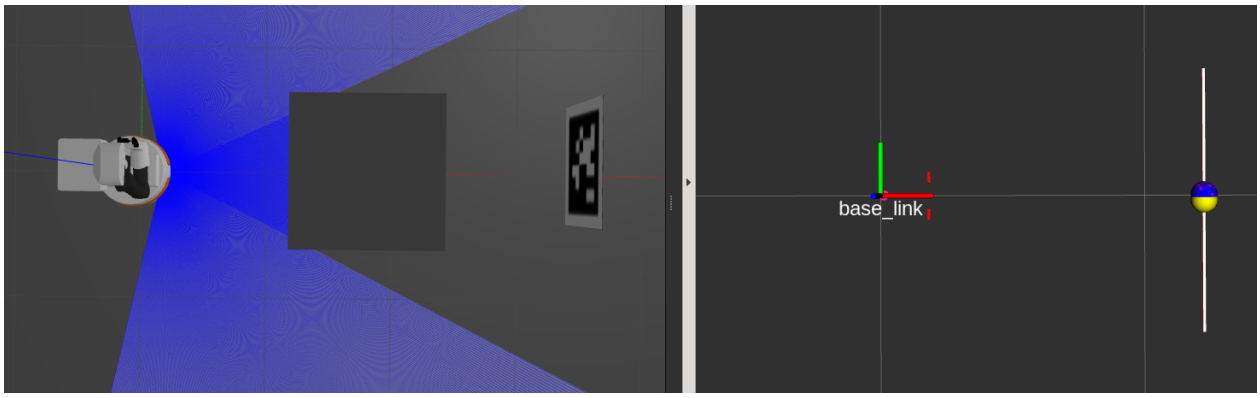
Nesse capítulo são descritos todos os testes realizados, suas configurações, bem como os resultados dos mesmos. A fim de facilitar a compreensão dos resultados obtidos, foi utilizado a ferramenta RVIZ para mostrar os pontos calculados e o ambiente de simulação do Gazebo para visualizar o robô realizando a trajetória evitando obstáculos. A primeira etapa consiste em testes experimentais da localização do laser e cálculo do centro da espiral. Já na segunda etapa, é dividido entre a escolha dos critérios de alternância entre o controlador de espiral e o controlado IBVS.

4.1 Cálculo do centro da espiral

Para visualizar o resultado do cálculo realizado para o centro da espiral O_s, a partir do algoritmo de processamento de dados do laser, é utilizado a ferramenta de visualização 3D RVIZ. Para validação do cálculo foi utilizado o modelo apresentado por Leca et al. (2019) na Figura 13 como testes iniciais. Na parte esquerda da Figura 21 é representado a simulação em ROS do modelo (a.) presente na Figura 13, onde está presente o robô TIAGo, um cubo e uma tag da família do AprilTag no qual serve de alvo para a posição final do robô. O resultado do cálculo de O_s pode ser visualizado na parte direita da Figura 21, onde a linha pontilhada branca representa o obstáculo selecionado após aplicar o critério de seleção, a esfera amarela representa o ponto O_b, a esfera azul escuro representa o ponto O_s e a esfera rosa representa o ponto O_c. Nesse resultado, onde todos os pontos estão sobrepostos, podemos validar o cálculo de acordo com os resultados obtidos por Leca et al. (2019), em que é calculado a projeção ortogonal do robô no obstáculo convexo e chega a conclusão que só existe uma possibilidade para O_p, o que coincide com Oc, logo $O_s = O_c = O_b = O_p$. Para validar o cálculo segundo o modelo (.b) da Figura 13, foi simulado no ROS como mostrado no lado esquerdo da Figura 22. Como resultado dessa simulação, mostrado na Figura 22 na parte direita, em que é calculado a projeção ortogonal do robô no obstáculo convexo disposto a um ângulo de 45°, e obtido o ponto mais próximo ao obstáculo (O_c) posicionado na quina do cubo e O_b posicionado dentro do cubo. Para esse teste o cálculo de O_p resulta na mesma posição que O_c, consequentemente o ponto mais próximo ao robô entre $O_c = O_p$ e O_b é O_c e O_p. Logo, a esfera azul escura, azul clara (representação de O_p) e rosa estão sobrepostas, representando que $O_c = O_p = O_s$. Em relação ao caso (c.) da Figura 13, a simulação realizada para teste pode ser visualizada na parte esquerda da Figura 23, onde possui a junção de dois obstáculos convexos para representar um obstáculo côncavo. Para esse teste, assim como no resultado obtido por Leca et al. (2019), o ponto O_c está posicionado na parede do obstáculo e o baricentro

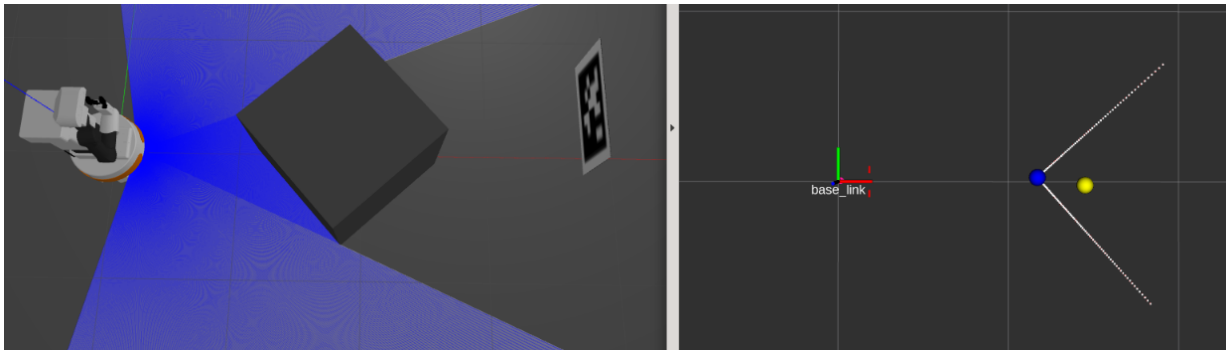
(O_b) está localizado mais próximo ao robô, como mostrado na parte direita da Figura 23, porém como o autor não considera o cálculo de O_p, o resultado seria $O_s = O_b$. Levando em consideração o cálculo de O_p, o valor de O_s é igual a O_p, pois é o ponto mais próximo ao robô entre O_c, O_p e O_p.

Figura 21 – Obstáculo convexo entre o robô e o alvo.



Fonte – Elaborado pelo autor (2022).

Figura 22 – Representação de um obstáculo convexo rotacionado entre o robô e o alvo.

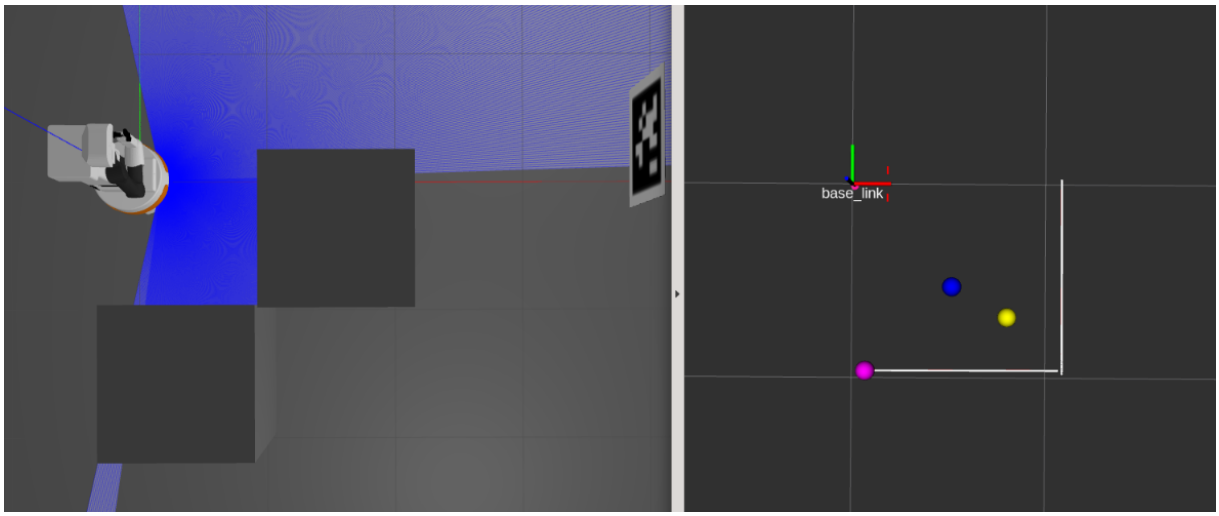


Fonte – Elaborado pelo autor (2022).

A fim de obter a posição do centro da espiral para mais tipos de obstáculos e possibilidades, são realizados diversos testes alternando a posição do robô, inserindo mais obstáculos e alternando a posição dos obstáculos, como mostrado abaixo:

- Obstáculo esférico entre o robô e o alvo: A parte esquerda da Figura 24 representa um obstáculo esférico entre o robô e o alvo no Gazebo e na parte direita o resultado

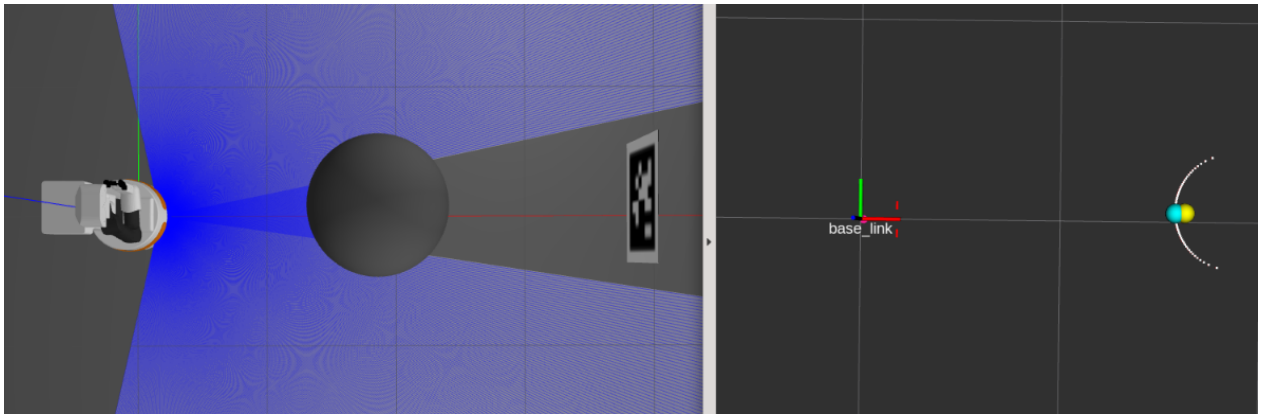
Figura 23 – Representação de um obstáculo côncavo entre o robô e o alvo.



Fonte – Elaborado pelo autor (2022).

do cálculo do centro da espiral no RVIZ. No caso de um esfera, o ponto O_c e O_p estão sobrepostos e mais próximos ao robô do que o baricentro O_b, logo pode-se afirmar que $O_p = O_c = O_s$.

Figura 24 – Representação de um obstáculo esférico entre o robô e o alvo.

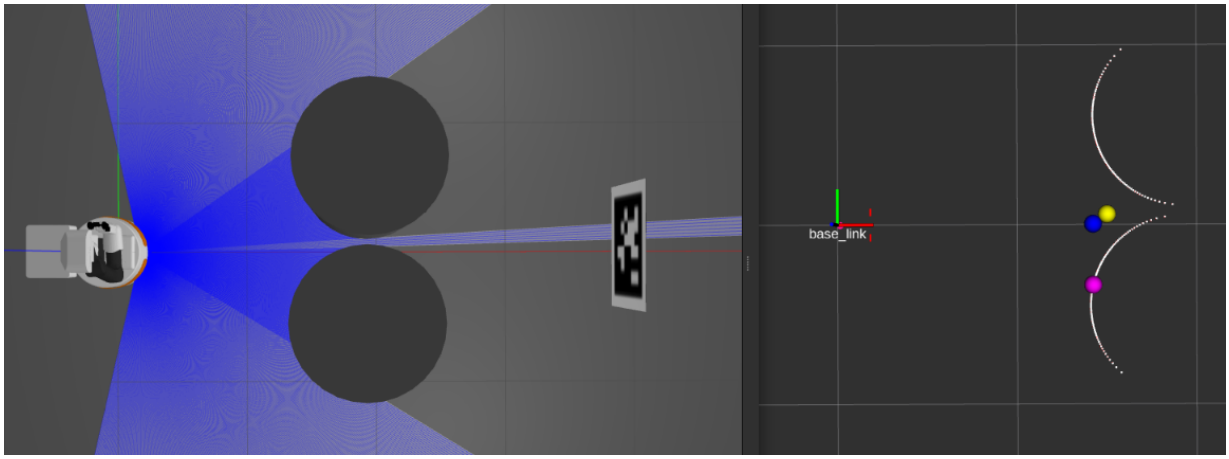


Fonte – Elaborado pelo autor (2022).

- Dois obstáculos cilíndricos entre o robô e o alvo: A parte esquerda da Figura 25 representa dois obstáculos cilíndricos entre o robô e o alvo no Gazebo e na parte direita o resultado do cálculo do centro da espiral no RVIZ. Os dois obstáculos estão

a uma distância menor que 15cm, logo após utilizar o critério de agrupamento das nuvens de pontos para identificação de um obstáculo, o algoritmo assume que os dois cilindros pertencem a um só objeto, como mostrado na linha pontilhada branca da parte direita da Figura 25. Para esse teste, nota-se que os pontos O_c, O_p e O_b são distintos e pode-se afirmar que ponto O_p é o ponto mais próximo ao robô, portanto $O_p = O_s$.

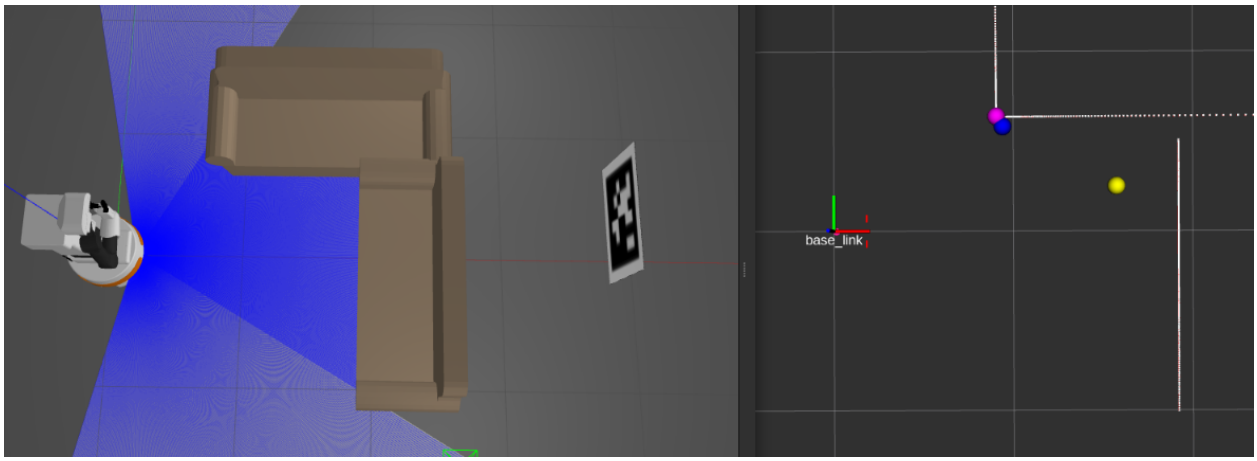
Figura 25 – Representação de dois obstáculos cilíndricos entre o robô e o alvo.



Fonte – Elaborado pelo autor (2022).

- Dois sofás em L entre o robô e o alvo: A parte esquerda Figura 26 representa dois sofás em L entre o robô e o alvo no Gazebo e na parte direita o resultado do cálculo do centro da espiral no RVIZ. Os dois obstáculos estão a uma distância menor que 15cm, logo são considerados como um só obstáculo, tendo o ponto O_p como ponto mais próximo ao robô, portanto $O_p = O_s$.
- Uma mesa entre o robô e o alvo: A parte esquerda Figura 27 representa uma mesa entre o robô e o alvo no Gazebo e na parte direita o resultado do cálculo do centro da espiral no RVIZ. Para esse teste, a altura que está posicionado o laser do robô para detecção da nuvem de pontos está abaixo do tampo da mesa, fazendo com que só seja identificado no RVIZ os pés da mesa. O ponto O_c, que representa o ponto mais próximo ao robô está localizado em um dos pés da mesa, o ponto O_b é resultado do cálculo do baricentro que está localizado para dentro da mesa e, por fim, o ponto O_p, que é a menor distância perpendicular das possíveis retas com o ponto O_c e os demais pontos do obstáculo, está localizado entre os dois pés da

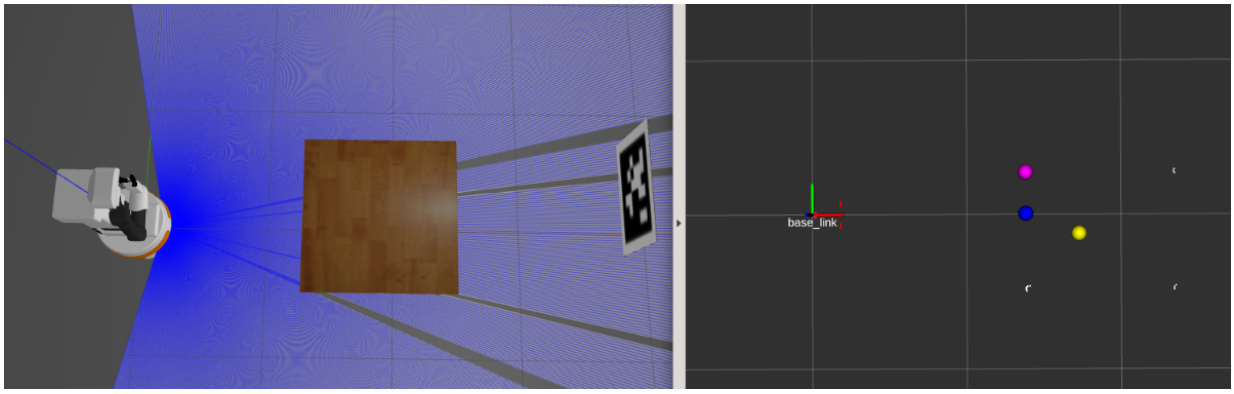
Figura 26 – Representação de dois sofás em L entre o robô e o alvo.



Fonte – Elaborado pelo autor (2022).

mesa mais próximos ao robô. Após a detecção de todos os três pontos, pode-se afirmar que o centro da espiral considerado é o ponto O_p, logo $O_p = O_s$.

Figura 27 – Representação de uma mesa entre o robô e o alvo.

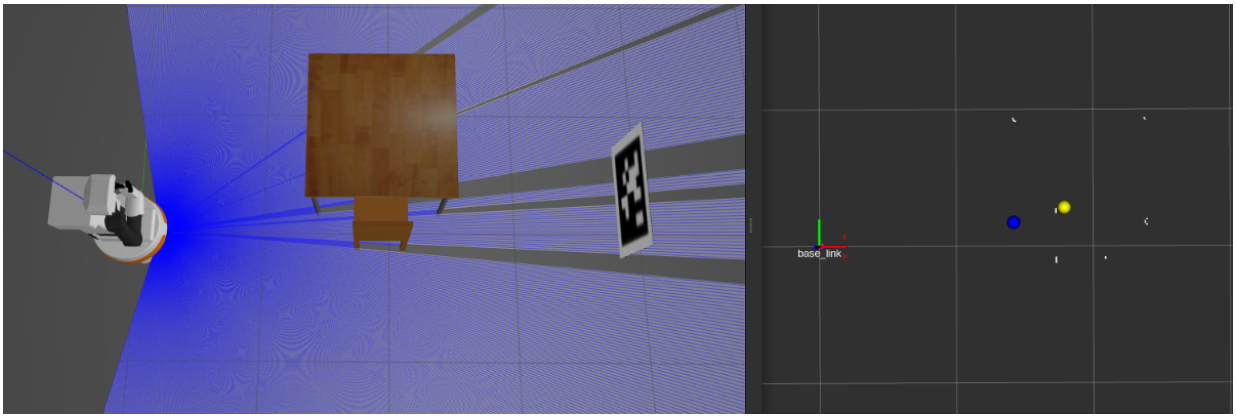


Fonte – Elaborado pelo autor (2022).

- Uma mesa com cadeira entre o robô e o alvo: A parte esquerda da Figura 28 representa uma mesa com uma cadeira entre o robô e o alvo no Gazebo, onde a cadeira está na passagem do robô e a mesa mais afastada e na parte direita o resultado do cálculo do centro da espiral no RVIZ. Com a inserção da cadeira no teste, o obstáculo considerado pelo algoritmo é a combinação da mesa com os pés da cadeira. Novamente é aplicado o cálculo dos pontos e nota-se que o ponto mais próximo

ao robô é um dos pés da mesa para ser considerado como O_c e o baricentro O_b está localizado entre a mesa e a cadeira. Já para o cálculo do ponto O_p, a distância perpendicular mais próxima do robô com as possíveis retas não pertence à reta, resultando com que O_p seja igual que O_c. Portanto, o centro da espiral considerado é o ponto O_c = O_p = O_s.

Figura 28 – Representação de uma mesa com cadeira entre o robô e o alvo.

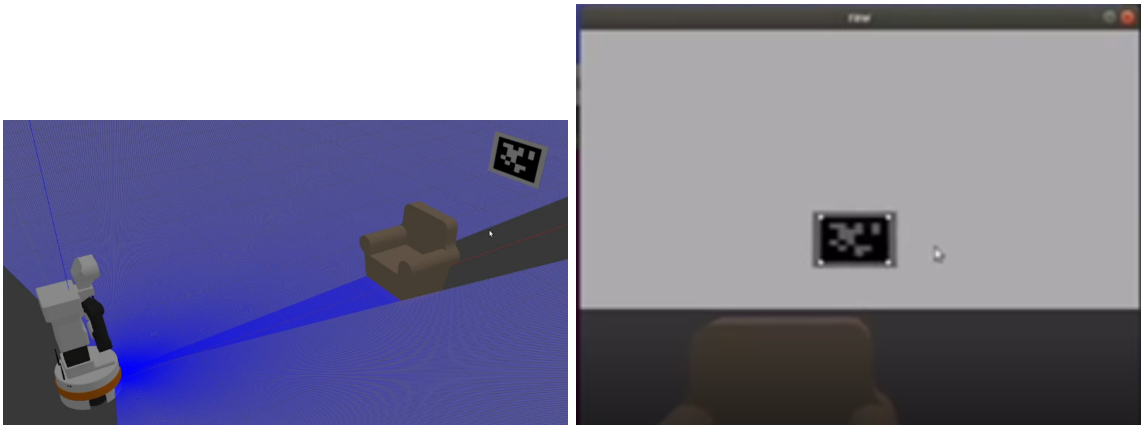


Fonte – Elaborado pelo autor (2022).

4.2 Critério de seleção com distância fixa

Para visualizar a trajetória do robô a partir dos controladores implementados, foi utilizado o critério de seleção de controladores com a distância fixa de 0,8 metros para o teste de uma poltrona como obstáculo entre o robô e o AprilTag, como mostrado na Figura 29a. A Figura 29b mostra a visão da câmera no tempo $t = 0s$, tendo sua posição inicial representada pela Figura 30a e inicialmente com o controlador IBVS. No tempo $t = 10s$, o robô identifica o robô a menos de 0,8 metros e realiza a troca de controlador para seguir a espiral com o critério de seleção com distância fixa, como mostrado na Figura 30b. As Figuras 30b a 30h representam o robô seguindo a espiral para realizar uma navegação segura e evitar de colidir em obstáculos com um intervalo de 5 segundos entre cada figura. A Figura 30i representa a volta para o controlador de IBVS no tempo $t = 40s$, onde ocorre o alinhamento da base e da cabeça do robô e a câmera alinha com os pontos do AprilTag finalizando a navegação. Após analisar o resultado, é possível notar que ocorre uma limitação na utilização do critério de mudança de controladores com base na distância fixa, onde o robô inicia o seguidor da espiral distante do obstáculo, sendo possível chegar mais próximo do obstáculo para começar a rotacionar, logo o uso do

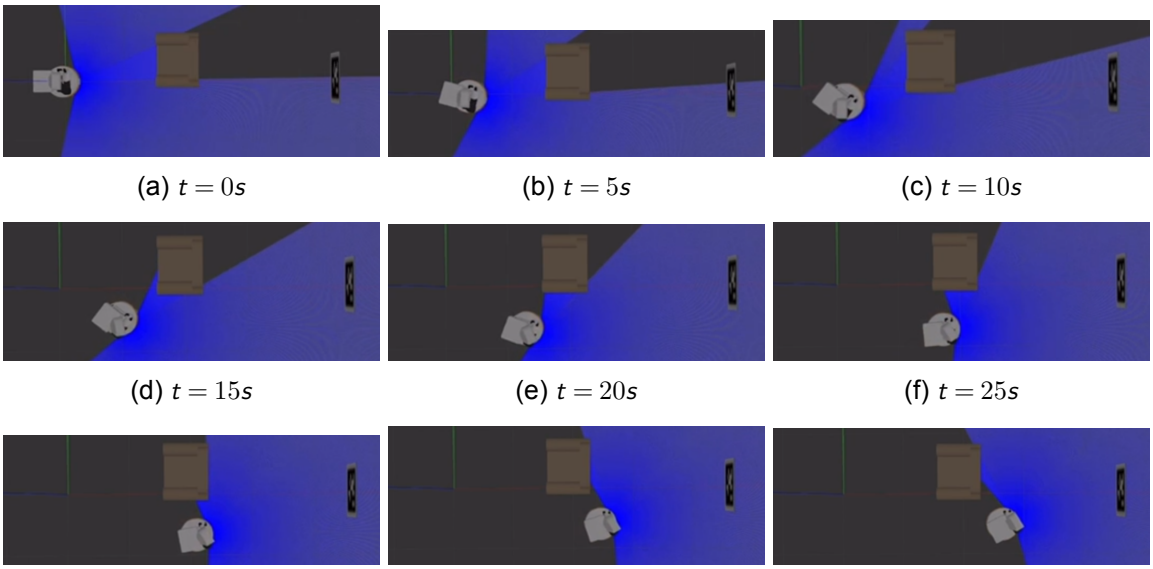
Figura 29 – Primeiro cenário.



(a) Ambiente de navegação com uma poltrona. (b) Visão da câmera na posição inicial.

Fonte – Elaborado pelo autor (2022).

Figura 30 – Evolução do robô com o critério de seleção com distância fixa.



(a) $t = 0s$ (b) $t = 5s$ (c) $t = 10s$

(d) $t = 15s$ (e) $t = 20s$ (f) $t = 25s$

(g) $t = 30s$ (h) $t = 35s$ (i) $t = 40s$

Fonte – Elaborado pelo autor (2022).

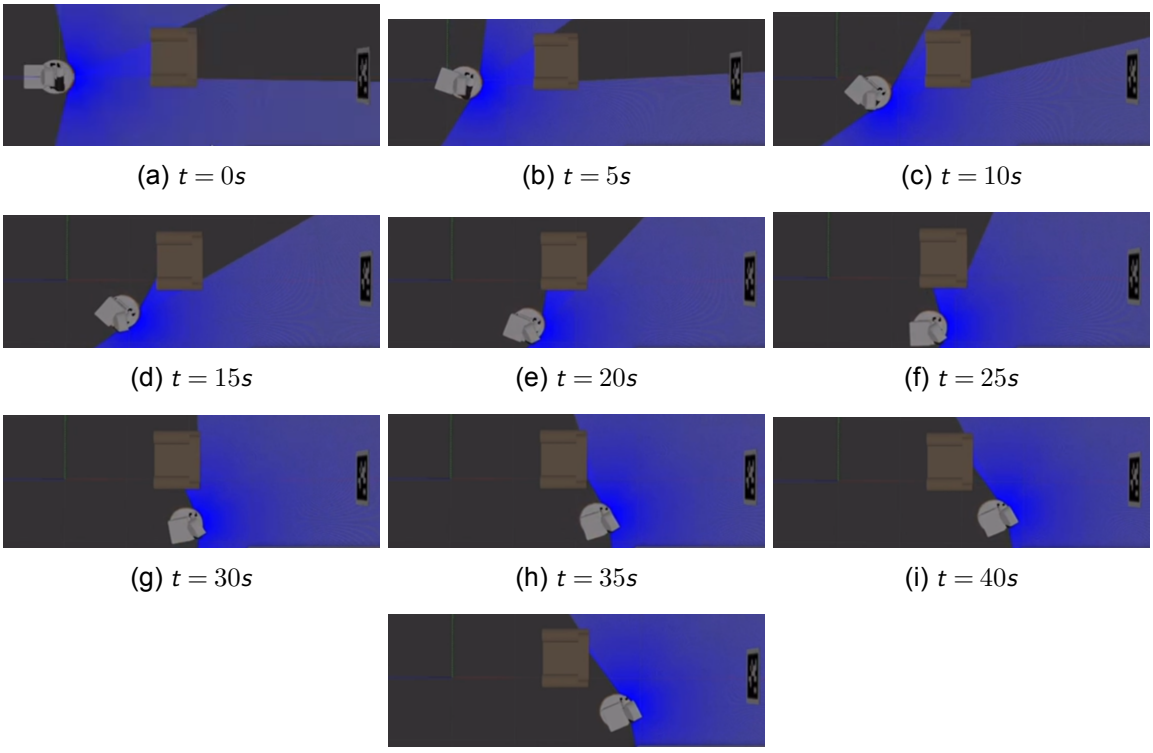
seguidor de espiral se torna desnecessário pois não ocorre uma antecipação de posição e é selecionado a depender do critério de segurança empregado, além de não se adaptar bem a novos obstáculos. A navegação completa do teste realizado pode ser encontrada em <https://youtu.be/bFiDskFjKSs>.

4.3 Critério de seleção baseada na previsão da distância

Para visualizar a diferença entre os dois critérios utilizados, é realizado o teste com a mesma configuração do anterior, onde utiliza a poltrona como obstáculo representado pela Figura 29a. O critério de seleção baseada na previsão da distância que se baseia em prever a distância virtual em $k+1$, com objetivo de antecipar o movimento e realizar uma navegação segura tendo o início do seguidor da espiral mais próximo ao obstáculo tornando, assim, uma trajetória com utilização de espaço otimizado. O robô inicia na posição $t = 0s$ representada pela Figura 31a e no momento $t = 5s$, representado pela Figura 31b, é realizado a iteração e ocorre a condição em que a previsão da próxima posição $\hat{d}(k+1)$ é menor que a distância atual e o obstáculo está posicionado na frente do robô, sendo necessário trocar para o controlador de seguidor de espiral. O robô permanece nesse controlador durante o tempo $t = 5s$ até o tempo $t = 35s$, representados pelas Figuras 31b a 31h. Para finalizar a trajetória, no tempo $t = 45s$ (Figura 31j) o algoritmo implementado reconhece não ter mais risco de obstrução com obstáculo, tendo a cabeça e a base do robô alinhados e a câmera alinhada com a AprilTag. Logo, para esse teste é mostrado que o critério de seleção baseada na previsão da distância é um método otimizado para a simulação da navegação, tendo seu movimento de rotação iniciado mais próximo ao obstáculo, mantendo a segurança no trajeto. A navegação completa do teste realizado pode ser encontrada em <https://youtu.be/bKBhqHlcF-g>.

Para validar o critério utilizado, é realizado outro teste com dois obstáculos com uma disposição mais complexa, representado por uma mesa e uma cadeira entre o robô e o AprilTag, mostrada na Figura 32a e a visão do robô na Figura 32b. A Figura 33a mostra a posição inicial do robô no ambiente de simulação do Gazebo. Como o obstáculo está próximo ao robô, no tempo $t = 5s$ (Figura 33b) é iniciado o controlador seguidor de espiral, onde é mantido nesse controlador até o tempo $t = 60s$, sendo representado da Figura 33b até a Figura 33m. No tempo $t = 65s$ (Figura 33n), ocorre a finalização do movimento pois a câmera do robô encontra os pontos da AprilTag e a cabeça se alinha com a base do robô. Portanto, é validado a utilização do critério de seleção, onde em todos os testes apresentados é alcançado o objetivo de finalizar a trajetória com segurança sem haver colisões com obstáculos. A navegação completa do teste realizado pode ser encontrada em <https://youtu.be/10y93o_l0z0>.

Figura 31 – Evolução do robô com o critério de seleção baseada na previsão da distância.



(a) $t = 0s$ (b) $t = 5s$ (c) $t = 10s$

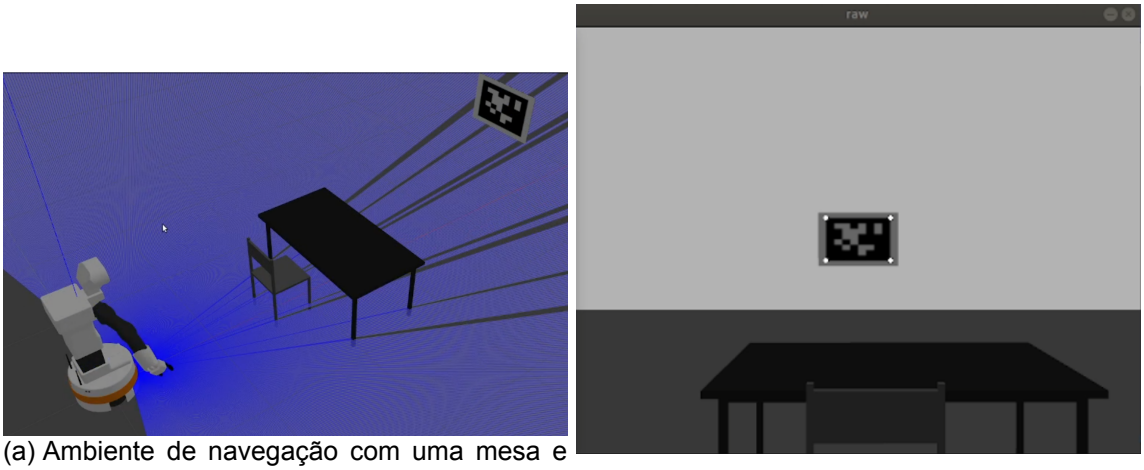
(d) $t = 15s$ (e) $t = 20s$ (f) $t = 25s$

(g) $t = 30s$ (h) $t = 35s$ (i) $t = 40s$

(j) $t = 45s$

Fonte – Elaborado pelo autor (2022).

Figura 32 – Segundo cenário.

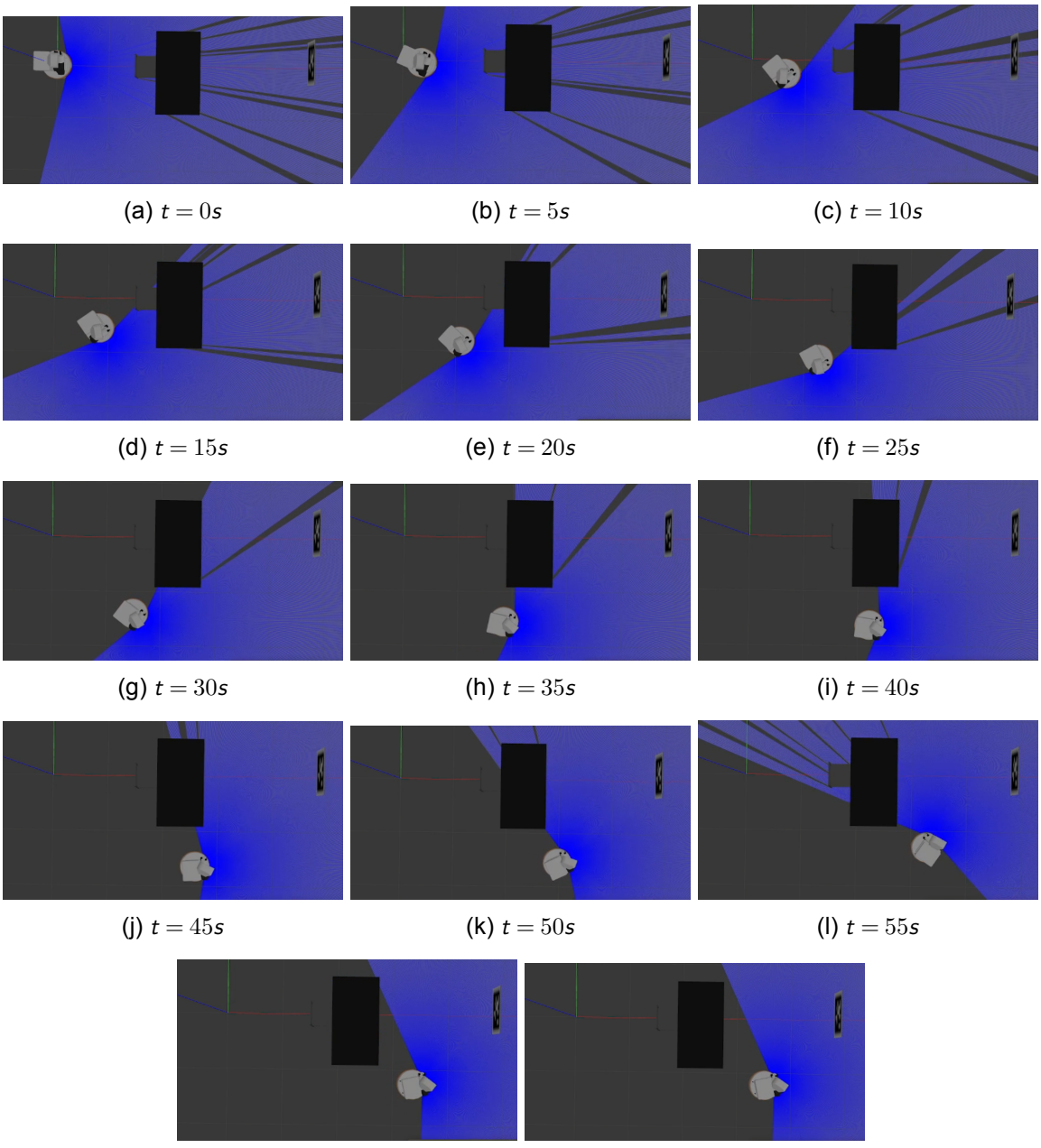


(a) Ambiente de navegação com uma mesa e uma cadeira.

(b) Visão da câmera na posição inicial.

Fonte – Elaborado pelo autor (2022).

Figura 33 – Evolução do robô com o critério de previsão da distância com mesa e cadeira.



(a) $t = 0s$ (b) $t = 5s$ (c) $t = 10s$

(d) $t = 15s$ (e) $t = 20s$ (f) $t = 25s$

(g) $t = 30s$ (h) $t = 35s$ (i) $t = 40s$

(j) $t = 45s$ (k) $t = 50s$ (l) $t = 55s$

(m) $t = 60s$ (n) $t = 65s$

Fonte – Elaborado pelo autor (2022).

5 CONCLUSÃO

Inicialmente foram apresentadas duas opções de navegação que permitem que robô seja capaz de se mover de forma autônoma em um ambiente pouco conhecido ou desconhecido, onde a primeira se baseia em um mapa do ambiente para planejar uma trajetória em meio a obstáculos para alcançar o objetivo final e a segunda que não se baseia na utilização de um mapa do ambiente, conhecida como navegação reativa, e se beneficia de informações locais geralmente baseadas em sensores, sendo a segunda opção escolhida para prosseguimento do trabalho. Para realizar a navegação reativa, foi apresentado uma máquina de estados finitos que permite selecionar o controlador apropriado dependendo da situação, onde seleciona entre o controlador seguidor de espiral e o controlador IBVS utilizando critérios com distância fixa ou baseado na previsão da distância. O controlador IBVS se baseia na visão da câmera do robô para dirigi-lo ao alvo, enquanto o controlador seguidor de espiral se baseia no dados de um laser para evitar colisões. Com isso, foi apresentado o processamento dos dados do laser para calcular o centro da espiral para obstáculos convexos e côncavos. Para ser possível validar a navegação segura do robô com os métodos propostos, foi necessário a utilização do ambiente de simulação Gazebo, onde foi inserido o robô TIAGo e um AprilTag, utilizando algoritmos de controle de seguidor de espiral e IBVS, implementados em Python no *framework* ROS, ambos de trabalhos anteriores disponíveis na literatura. Os resultados do processamento de dados obtidos são validados quando comparados com estudos apresentados na literatura e a navegação obtida com Gazebo comprova que é possível realizar uma navegação segura iniciando o movimento de contorno do obstáculo próximo ao mesmo, sendo desnecessário ter uma distância pré estabelecida.

O trabalho apresentado possui duas limitações, sendo a primeira limitação o fato de não ser possível aplicar o método apresentado em casos onde o alvo está dentro de um obstáculo côncavo, pois não possui um sistema de decisão implementado para saber se fecha virtualmente o obstáculo; e a segunda limitação é que o robô precisa sempre visualizar o alvo para que consiga realizar o controle corretamente. Para futuros trabalhos a respeito da análise apresentada, é sugerido adicionar diversos obstáculos complexos para obter mais validações para o método proposto, podendo ser inseridor em um ambiente real, como por exemplo em um ambiente residencial; implementar um critério de decisão para verificar se haverá ou não o fechamento virtual do obstáculo de acordo com o posicionamento do alvo; criar uma estratégia para achar o alvo caso o robô o perca de vista; e inserir obstáculos dinâmicos para verificar se ainda é possível manter uma navegação segura.

REFERÊNCIAS

AREANN, M.-C. et al. Laser ranging: a critical review of usual techniques for distance measurement [j]. **Opt. Eng**, v. 40, n. 1, 2001. Citado na página 16.

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 8.

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 9 e 22.

CHAUMETTE, F.; HUTCHINSON, S. Visual servo control. ii. advanced approaches [tutorial]. **IEEE Robotics & Automation Magazine**, IEEE, v. 14, n. 1, p. 109–118, 2007. Citado na página 21.

COLEMAN, D. **ROS, W. rqt graph.** 2018. Disponível em: <https://wiki.ros.org/rqt_graph>. Citado na página 14.

DATTALO, A. **ROS Introduction**. 2018. Disponível em: <https://wiki.ros.org/ROS/Introduction>. Citado na página 14.

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 3 vezes nas páginas 9, 23 e 33.

GAO, C.; PIAO, X.; TONG, W. Optimal motion control for ibvs of robot. In: IEEE. **Proceedings of the 10th World Congress on Intelligent Control and Automation**. [S.l.], 2012. p. 4608–4611. Citado na página 20.

GAZEBO. **Robot simulation made easy**. 2014. Disponível em: <https://gazebosim.org/>. Citado 2 vezes nas páginas 14 e 15.

KERR, J.; NICKELS, K. Robot operating systems: Bridging the gap between human and robot. In: IEEE. **Proceedings of the 2012 44th Southeastern Symposium on System Theory (SSST)**. [S.l.], 2012. p. 99–104. Citado na página 12.

KNOWLEDGEBASE, R. **APRILTAG, Poses**. 2018. Disponível em: <https://roboticsknowledgebase.com/wiki/sensing/apriltags/>. Citado 2 vezes nas páginas 17 e 18.

LAMAS W.; GROH, F. G. F. **CÁLCULO E VALIDAÇÃO DA CINEMÁTICA DIRETA E DA CINEMÁTICA INVERSA PARA USO NA TRAJETÓRIA DE UM ROBÔ CILÍNDRICO**. 2002. II Congress of Mechanical Engineering, August 2002, João Pessoa, Brazill. Citado na página 31.

LECA, D. **Navigation autonome d'un robot agricole**. Tese (Doutorado) — Université Toulouse 3 Paul Sabatier (UT3 Paul Sabatier), 2021. Citado 3 vezes nas páginas 25, 26 e 28.

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 25, 27, 28 e 37.

MATHEMATICS, E. O. **Spirals**. 2011. Disponível em: <https://encyclopediaofmath.org/wiki/Spirals>. Citado na página 22.

MELO, L. F. de et al. Mobile robots and wheelchairs control navigation design using virtual simulator tools. In: IEEE. **2012 IEEE International Symposium on Industrial Electronics**. [S.l.], 2012. p. 1280–1285. Citado na página 8.

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 8.

MUÑOZ, G. L. L. Análise comparativa das técnicas de controle servo-visual de manipuladores robóticos baseadas em posição e em imagem. 2011. Citado na página 20.

OLSON, E. **APRILTAG**. 2010. Disponível em: <https://april.eecs.umich.edu/software/apriltag>. Citado na página 17.

PETITEVILLE, A. D. et al. 2d visual servoing for a long range navigation in a cluttered environment. In: IEEE. **2011 50th IEEE Conference on Decision and Control and European Control Conference**. [S.l.], 2011. p. 5677–5682. Citado na página 30.

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 3 vezes nas páginas 22, 23 e 24.

ROBOTICS, C. **ROS 101: Intro to the Robot Operating System**. 2014. Disponível em: <https://robohub.org/ros-101-intro-to-the-robot-operating-system/>. Citado na página 13.

ROBOTICS, P. **Technical specifications**. 2022. Disponível em: <https://pal-robotics.com/robots/tiago/>. Citado 2 vezes nas páginas 11 e 12.

ROS. **ROS**. 2022. Disponível em: <https://www.embarcados.com.br/uma-introducao-ao-robot-operating-system-ros/>. Citado na página 13.

ROS. **ROS**. 2022. Disponível em: <https://www.ros.org>. Citado na página 13.

ROSEBROCK, A. **AprilTag with Python**. 2020. Disponível em: <https://pyimagesearch.com/2020/11/02/apriltag-with-python/>. Citado na página 19.

ROSSUM, G. V. et al. Python programming language. In: **USENIX annual technical conference**. [S.l.: s.n.], 2007. v. 41, n. 1, p. 1–36. Citado na página 25.

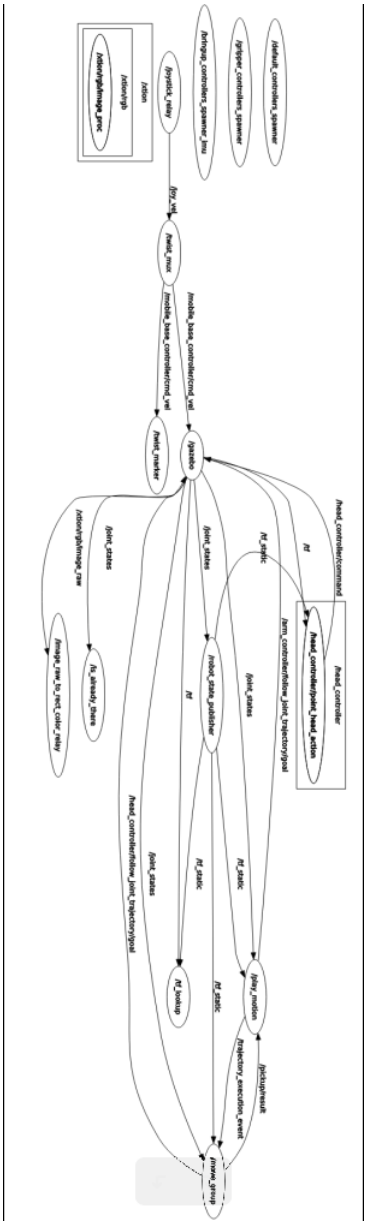
RVIZ. **RVIZ**. 2022. Disponível em: <https://docs.aws.amazon.com/pt_br/robomaker/latest/dg/simulation-tools-rviz.html>. Citado na página 15.

SIEGWART, R.; NOURBAKHSH, I. R.; SCARAMUZZA, D. **Introduction to autonomous mobile robots**. [S.l.]: MIT press, 2011. Citado na página 8.

WIKIPEDIA. **Framework**. 2021. Disponível em: <https://pt.wikipedia.org/wiki/Framework>. Citado na página 12.

APÊNDICE A – GRAFO ROS DO SISTEMA

Figura 34 – Nós do grafo utilizado para controlar o robô *TIAGo*.



Fonte – Elaborado pelo autor (2022).

APÊNDICE B – LASER_PROCESSING

```
#! /usr/bin/python

import rospy
import math
import time
import numpy

import std_msgs.msg
from std_msgs.msg import Float32MultiArray
from std_msgs.msg import Int32
from sensor_msgs.msg import PointCloud
from geometry_msgs.msg import Point32
from sensor_msgs.msg import LaserScan
from geometry_msgs.msg import Twist

from visualization_msgs.msg import Marker



class staticObst_class():

    def __init__(self):

        # Subscrib to the laser
        self.sub_laser = rospy.Subscriber('/scan_raw', LaserScan, self.callback_laser, qSize=1)

        # Publish spiral center
        self.pub_message = rospy.Publisher('/spiral/center', Float32MultiArray, qSize=1)
        self.spiralCenter = Float32MultiArray()
        self.spiralCenter.data = [0.0]*2

        # Publish number of obstacles
        self.pub_nbObst = rospy.Publisher('/spiral/nbObst', Int32, qSize=1)

        # Publish point cloud of the nearest obstacle
        self.pub_obstacle = rospy.Publisher('/rviz/obstacle', PointCloud, qSize=1)
        self.LaserPoints = PointCloud()

        # Pointcloud header
        header = std_msgs.msg.Header()
        header.stamp = rospy.Time.now()
        header.frame_id = 'base_link'
        self.LaserPoints.header = header

        # Publisher of the computed points
        self.pub_Oc = rospy.Publisher('/rviz/oc', Marker, qSize=1)
        self.pub_Ob = rospy.Publisher('/rviz/ob', Marker, qSize=1)
        self.pub_Op = rospy.Publisher('/rviz/op', Marker, qSize=1)
        self.pub_Os = rospy.Publisher('/rviz/os', Marker, qSize=1)
```

```
        # Threshold used to cluster point clouds
        self.pcThreshold = 0.15

        # Threshold to merge the obstacles
        self.mergeThreshold = 1.0

        # Points Oc, Ob, Op, Os [x,y,distance,theta]
        self.oc = [0.0, 0.0, float('inf'), 0.0]
        self.ob = [0.0, 0.0, float('inf'), 0.0]
        self.op = [float('inf'), float('inf'), float('inf'), 0.0]
        self.os = [0.0]*4

    #-----------------------------
    def callback_laser(self, msg):

        print("Laser")

        # Update laser parameters
        self.increment = msg.angle_increment
        self.size = len(msg.ranges)
        self.range_max = msg.range_max
        self.angle_min = msg.angle_min

        # Clear the lists
        self.cartesian_points = []
        self.polar_points = []
        self.pointsIdentifiers = [] # number to idenitfy the obstacle attached to the point
        self.numberOfObstacle = 0

        Xlist = []
        Ylist = []
        Dlist = []
        Tlist = []

        # For each point of the laser scan
        for i in range(self.size):

            # To not take into account the robot points and infinity
            if msg.ranges[i] > 0.15 and msg.ranges[i] < 25:

                # Count the number of obstacles based on a distance criteria
                if( i != 0):
                    if (abs(msg.ranges[i-1] - msg.ranges[i]) >= self.pcThreshold):
                        self.numberOfObstacle += 1

                # Allocate the obstacle number to the point
                self.pointsIdentifiers.append(self.numberOfObstacle)

                # Polar angle in the laser frame
                theta_laser = self.increment * i + self.angle_min

                # Cartesian coordinates in the robot frame
                # X + 0.202 to exress the coordiantes in the robot frame
                Xpoint = msg.ranges[i] * math.cos(theta_laser) + 0.202
                Ypoint = msg.ranges[i] * math.sin(theta_laser)
```

```
            # Polar coordinates in the robot frame
            distance_robot = math.sqrt((Xpoint **2) +  (Ypoint**2))
            theta_robot = numpy.arctan2(Ypoint,Xpoint)

            # Save the coordinates
            Xlist.append(Xpoint)
            Ylist.append(Ypoint)
            Dlist.append(distance_robot)
            Tlist.append(theta_robot)

    # [0][]:x, [1][]:y, [][i]
    self.cartesian_points.append(Xlist)
    self.cartesian_points.append(Ylist)
    self.polar_points.append(Dlist)
    self.polar_points.append(Tlist)

    # If there is an obstacle in the field of view of the robot
    if (len(self.cartesian_points[0]) > 0):

        # Merge close obstacles
        self.mergeObstacles()

        # Select the nearest obstacle
        self.nearestObstacle()

        # Compute the closet point
        self.calculate_Oc()

        # Compute the barycenter
        self.calculate_Ob()

        # Compute the closest virtual point
        self.calculate_Op()

        # Compute the spiral center
        self.calculate_Os()

        # Update the spiral center message
        self.spiralCenter.data[0] = self.os[2]
        self.spiralCenter.data[1] = self.os[3]

    else:

        # Update the spiral center message
        self.spiralCenter.data = [float("NaN")]*2

        # Update the points coordinates
        self.oc = [float("NaN")]*4
        self.ob = [float("NaN")]*4
        self.op = [float("NaN")]*4
        self.os = [float("NaN")]*4

    # Publish the spiral center coordinates
    self.pub_message.publish(self.spiralCenter)
```

```
        # Publish the computed points for visualization in rviz
        self.pubMarker()

        # Publish the number of obstacle
        self.pub_nbObst.publish(self.numberOfObstacle)

    #--------------------------
    def callback_cmd(self, msg):
        self.v = msg.linear.x
        self.w = msg.angular.z

    #-----------------------------
    def mergeObstacles(self):

        # Lists of point list organized by identifier
        Xid = []
        Yid = []

        # Point list for one identifier
        X = []
        Y = []

        # Current identifier
        currentId = 1

        # Split the points based on their obstacle identifier
        for i in range(len(self.pointsIdentifiers)):

            # Check if other obstacle
            if (self.pointsIdentifiers[i] != currentId):

                # Add a new list
                Xid.append(X)
                Yid.append(Y)

                # Update the current identifier
                currentId = self.pointsIdentifiers[i]

                # Clear list
                X = []
                Y = []

            # Include the point in the list
            X.append(self.cartesian_points[0][i])
            Y.append(self.cartesian_points[1][i])

        # Add the last point list
        if (len(X)):
            Xid.append(X)
            Yid.append(Y)

        # List of couple to merge
        groupListObstacle = []
```

```
    # Compute the minimal distance between the clusters
    for c1 in range(self.numberOfObstacle - 1):
        for c2 in range(c1, self.numberOfObstacle):

            # Number of points in each cluster
            nbElmt1 = len(Xid[c1])
            nbElmt2 = len(Xid[c2])

            # Point identifier in the cluster
            p1 = 0
            p2 = 0

            computing = True
            while(computing):

                # Compute the distance
                dist = math.sqrt((Xid[c1][p1]-Xid[c2][p2])**2+(Yid[c1][p1]-Yid[c2][p2])**2)

                # Check merging condition
                if (dist < self.mergeThreshold):
                    # Stop for these two clusters
                    computing = False

                    # Save the couple id to be merged
                    groupListObstacle.append([c1+1,c2+1])

                else:
                    # Consider next points
                    p2 += 1

                    if (p2 == nbElmt2):
                        p1 += 1
                        p2 = 0

                        if (p1 ==  nbElmt1):
                            # Stop for these two clusters
                            computing = False

    # Update the identifier list
    for couple in reversed(groupListObstacle):
        for n in range(len(self.pointsIdentifiers)):
            if (self.pointsIdentifiers[n] == couple[1]):
                self.pointsIdentifiers[n] = couple[0]

def nearestObstacle(self):

    # Look for the shortest distance
    minDist = min(self.polar_points[0])
    # Look for the index of the shortest distance
    idxMinValue = 0
    for index in range(len(self.polar_points[0])):
        if(self.polar_points[0][index] == minDist):
            idxMinValue = index
    # Closest obstacle identifer
    idxNearestObstacle = self.pointsIdentifiers[idxMinValue]
```

```
    # Compute the startig position of the nearest obstacle and its number of points
    initObs = -1
    nbOfPoints = 0
    for i in range(len(self.pointsIdentifiers)):
        if(self.pointsIdentifiers[i] == idxNearestObstacle):
            # Detect the starting point
            if initObs < 0:
                initObs = i
            # Count the number of points
            nbOfPoints += 1

    # Compute the ending position of the nearest obstacle
    endObs = initObs + nbOfPoints

    # Remove the points belonging to the other obstacles
    if (endObs != len(self.pointsIdentifiers)):
        del self.cartesian_points[0][endObs:]
        del self.cartesian_points[1][endObs:]
        del self.polar_points[0][endObs:]
        del self.polar_points[1][endObs:]
    del self.cartesian_points[0][:initObs]
    del self.cartesian_points[1][:initObs]
    del self.polar_points[0][:initObs]
    del self.polar_points[1][:initObs]

    # Fill the point cloud
    self.LaserPoints.points = []
    for index in range(len(self.cartesian_points[0])):
         self.LaserPoints.points.append(Point32(self.cartesian_points[0][index],
         self.cartesian_points[1][index], 0))

    # Publish the point cloud
    self.pub_obstacle.publish(self.LaserPoints)

#----------------------
def calculate_Oc(self):

    # Look for the shortest distance
    minDist = self.range_max
    index = 0
    for i in range(len(self.polar_points[0])):
        if self.polar_points[0][i] < minDist:
            minDist = self.polar_points[0][i]
            index = i

    # Save the coordinates of Oc
    self.oc[0] = self.cartesian_points[0][index]
    self.oc[1] = self.cartesian_points[1][index]
    self.oc[2] = self.polar_points[0][index]
    self.oc[3] = self.polar_points[1][index]

#----------------------
def calculate_Ob(self):
```

```
    # Sum the coordinates of the obstacle points
    xSum = ySum = 0
    nbPoints = len(self.cartesian_points[0])
    for i in range(nbPoints):
        xSum += self.cartesian_points[0][i]
        ySum += self.cartesian_points[1][i]

    # Save the coordinates of Ob
    self.ob[0] = xSum/nbPoints
    self.ob[1] = ySum/nbPoints
    self.ob[2] = math.sqrt(self.ob[0]**2 + self.ob[1]**2)
    self.ob[3] = numpy.arctan2(self.ob[1], self.ob[0])

#----------------------
def calculate_Op(self):

    # Coordinates of point Oc
    x_Oc = self.oc[0]
    y_Oc = self.oc[1]

    # Robot coordinates in the robot frame
    X_Robot = 0.0
    Y_Robot = 0.0

    # List to store the Op candidates
    candidates = []
    # Include the Oc point
    candidates.append(self.oc)

    # Compute Op for each point of the obstacle
    nbPoints = len(self.cartesian_points[0])
    for (x_Obstacle, y_Obstacle) in zip(self.cartesian_points[0], self.cartesian_points[1]):

        # Projection of the robot center on the Oc-obstacle line
        if(x_Obstacle != x_Oc):

            #OcOo Vector
            Vco = [x_Obstacle - x_Oc, y_Obstacle - y_Oc]
            #OcOr vector
            Vcr = [X_Robot - x_Oc, Y_Robot - y_Oc]

            #Project Vcr onto values_Oc
            num = Vcr[0]*Vco[0] + Vcr[1]*Vco[1]
            den = Vco[0]*Vco[0] + Vco[1]*Vco[1]
            res = num/den
            v = [Vco[0]*res, Vco[1]*res]

            #Point projected onto the line
            P = [x_Oc + v[0], y_Oc + v[1]]

            # Polar coordinates of the candidates
            distance = math.sqrt(P[0]**2 + P[1]**2)
            theta = numpy.arctan2(P[1],P[0])

            # Check if the point belongs to the segment
```

```
            belongs = False
            if(x_Obstacle >  x_Oc):
                if(y_Obstacle >  y_Oc):
                    if(P[0]<=x_Obstacle and P[0]>=x_Oc and P[1]<=y_Obstacle and P[1]>=y_Oc):
                        candidates.append([P[0], P[1], distance, theta])
                        belongs = True
                else:
                    if(P[0]<=x_Obstacle and P[0]>=x_Oc and P[1]<=y_Oc and P[1]>=y_Obstacle):
                        candidates.append([P[0], P[1], distance, theta])
                        belongs = True
            else:
                if(y_Obstacle >  y_Oc):
                    if(P[0]<=x_Oc and P[0]>=x_Obstacle and P[1]<=y_Obstacle and P[1]>=y_Oc):
                        candidates.append([P[0], P[1], distance, theta])
                        belongs = True
                else:
                    if(P[0]<=x_Oc and P[0]>=x_Obstacle and P[1]<=y_Oc and P[1]>=y_Obstacle):
                        candidates.append([P[0], P[1], distance, theta])
                        belongs = True

    # Look for the shortest distance
    minDist = float('inf')
    index = 0
    for i in range(len(candidates)):
        if candidates[i][2] < minDist:
            minDist = candidates[i][2]
            index = i

    # Save the coordinates of Op
    self.op[0] = candidates[index][0]
    self.op[1] = candidates[index][1]
    self.op[2] = candidates[index][2]
    self.op[3] = candidates[index][3]

#---------------------
def calculate_Os(self):

    # Look for the shortest distance
    minDist = float('inf')
    index = 0
    candidates = [self.oc, self.ob, self.op]

    for i in range(3):
        if candidates[i][2] < minDist:
            minDist = candidates[i][2]
            index = i

    # Save the coordinates of Os
    self.os[0] = candidates[index][0]
    self.os[1] = candidates[index][1]
    self.os[2] = candidates[index][2]
    self.os[3] = candidates[index][3]

#-------------------
def pubMarker(self):
```

```
        marker = Marker()
        marker.header.frame_id = "base_link"
        marker.type = marker.SPHERE
        marker.action = marker.ADD
        marker.scale.x = 0.1
        marker.scale.y = 0.1
        marker.scale.z = 0.1
        marker.color.a = 1.0
        marker.color.r = 1.0
        marker.color.g = 0.0
        marker.color.b = 1.0
        marker.pose.orientation.w = 1.0
        marker.pose.position.x = self.oc[0]
        marker.pose.position.y = self.oc[1]
        marker.pose.position.z = 0.0
        self.pub_Oc.publish(marker)

        marker.color.r = 1.0
        marker.color.g = 1.0
        marker.color.b = 0.0
        marker.pose.position.x = self.ob[0]
        marker.pose.position.y = self.ob[1]
        self.pub_Ob.publish(marker)

        marker.color.r = 0.0
        marker.color.g = 1.0
        marker.color.b = 1.0
        marker.pose.position.x = self.op[0]
        marker.pose.position.y = self.op[1]
        self.pub_Op.publish(marker)

        marker.color.r = 0.0
        marker.color.g = 0.0
        marker.color.b = 1.0
        marker.pose.position.x = self.os[0]
        marker.pose.position.y = self.os[1]
        self.pub_Os.publish(marker)


#-------------------------
if __name__ == "__main__":

    rospy.init_node("laser_processing_node")
    processsing = staticObst_class()
    time.sleep(1)
    rospy.spin()
```

APÊNDICE C – APRIL_DETECTOR

```
#! /usr/bin/python

import rospy
import time
import cv2

import apriltag
from cv_bridge import CvBridge, CvBridgeError
from sensor_msgs.msg import Image
from std_msgs.msg import Float32MultiArray

import sys

class AprilDetector_class():

    def __init__(self):
        print('AprilTag detector node initialization')

        # Number of visual features
        self.nbVF = 8

        # Bridge to convert ROS message to openCV
        self.bridge = CvBridge()

        # Subscriber to the camera image
        self.image_sub=rospy.Subscriber("/xtion/rgb/image_raw",Image,self.callback_SubCamera)

        # Publisher: apriltag 4 points coordinates
        self.vf_pub = rospy.Publisher("/ibvs/vf", Float32MultiArray, queue_size=1)
        self.vf = Float32MultiArray()
        self.vf.data = [0.0]*self.nbVF

        # Apriltag Detector ----------------------------------------------------
        options = apriltag.DetectorOptions(families='tag36h11')
        self.detector = apriltag.Detector(options)

        # Camera geometry
        self.f = 1#0.0095 # camera focal
        self.ps = 19e-5 # pixel size
        self.camx = 320.5
        self.camy = 240.5
        self.camf = 522.1910329546544

    def callback_SubCamera(self, msg):
        print('Camera callback')
        try:
            self.cv_image = self.bridge.imgmsg_to_cv2(msg, "bgr8")
        except CvBridgeError as e:
            print(e)
```

```
        self.cv_image_gray = cv2.cvtColor(self.cv_image, cv2.COLOR_BGR2GRAY)

        # Apriltag detection
        result = self.detector.detect(self.cv_image_gray)

        # Convert from pixel to metric
        for i in range(self.nbVF/2):
            self.vf.data[2*i] = (result[0].corners[i][1]-self.camy)/self.camf # Row
            self.vf.data[2*i+1] = -(result[0].corners[i][0]-self.camx)/self.camf # Col

        # Publish the visual features
        self.vf_pub.publish(self.vf)

        x1 = int(result[0].corners[0][1])
        y1 = int(result[0].corners[0][0])
        x2 = int(result[0].corners[1][1])
        y2 = int(result[0].corners[1][0])
        x3 = int(result[0].corners[2][1])
        y3 = int(result[0].corners[2][0])
        x4 = int(result[0].corners[3][1])
        y4 = int(result[0].corners[3][0])

        cv2.rectangle(self.cv_image, (y1-1, x1+1), (y1+1, x1-1), (255, 255, 255), 2)
        cv2.rectangle(self.cv_image, (y2-1, x2+1), (y2+1, x2-1), (255, 255, 255), 2)
        cv2.rectangle(self.cv_image, (y3-1, x3+1), (y3+1, x3-1), (255, 255, 255), 2)
        cv2.rectangle(self.cv_image, (y4-1, x4+1), (y4+1, x4-1), (255, 255, 255), 2)
        cv2.imshow("raw", self.cv_image)
        cv2.waitKey(3)
        cv2.imwrite("target.png", self.cv_image)

# MAIN ----------------------------------------------------------------------

rospy.init_node('aprilDetector_node')
detector = AprilDetector_class()
# time.sleep(1)
rospy.spin()
```

APÊNDICE D – SPIRAL_CONTROLLER

```
#! /usr/bin/python


import rospy
import math
import time
import numpy as np

from std_msgs.msg import Float32MultiArray
from std_msgs.msg import Int32
from sensor_msgs.msg import JointState

class Spiral_class():

    def __init__(self):

        # Subscriber to the spiral center topic
        self.subCtrMsg=rospy.Subscriber("/spiral/center",Float32MultiArray,self.cb_center,qSize=1)

        # Subscriber to the visual features topic
        self.image_sub = rospy.Subscriber("/ibvs/vf", Float32MultiArray, self.cb_SubscribeVF)

        # Subscriber to the head state
        self.headSate_sub = rospy.Subscriber("/joint_states", JointState, self.cb_SubHeadState)

        # Subscriber to control mode
        self.sub_ControlMode = rospy.Subscriber("mix/mode", Int32, self.callback_controlMode)

        # Publisher for the commands
        self.commands_pub = rospy.Publisher("/spiral/cmd", Float32MultiArray, queue_size=1)
        self.cmdToPub = Float32MultiArray()
        self.cmdToPub.data = [0.0]*3

        # Platform orientation
        self.thetaP = 0

        # Y center of the visual features
        self.Y0 = 0

        # Interaction Matrix of the center point (Y only)
        self.Ly = np.zeros((1,3))

        # Mobile Base Jacobian Matrix
        self.Jb = np.zeros((3,2))

        # Head Jacobian Matrix
        self.Jh = np.zeros((3,1))

        # Mobile base command vector
        self.qb = np.zeros((2,1))
```

```
        # Head controller gain
        self.lambdaHead = 0.3

        # Commnands
        self.v = 0.2
        self.w = 0.0
        self.wh = 0.0

        # Spiral center coordinates
        self.deltaBH = 0.12 # Distance between the base center and the head
        self.current_d = 0.0
        self.current_theta = 0.0

        # Camera parameters
        self.cx = 0#0.096
        self.cy = 0#0.022

        # Control mode
        self.controlMode = 0

    def cb_center(self, msg):

        # Update the spiral center values
        self.current_d = msg.data[0]
        self.current_theta = msg.data[1]

        # In the presence of an obstacle
        if (not(math.isnan(self.current_d))):

            # Call the spiral following controller
            self.alphaDistance_controller()

            # Call the visual head controller
            self.head_controller()

            # Update the commands
            self.cmdToPub.data[0] = self.v
            self.cmdToPub.data[1] = self.w
            self.cmdToPub.data[2] = self.wh

        # Abscence of obstacle
        else:

            # Update the commands
            self.cmdToPub.data[0] = 0.0
            self.cmdToPub.data[1] = 0.0
            self.cmdToPub.data[2] = 0.0

        # Publish the commands
        self.commands_pub.publish(self.cmdToPub)

    def cb_SubHeadState(self, msg):
        # Update the head angle
        self.thetaP = msg.position[9]
```

```
def cb_SubscribeVF(self, msg):

    # Compute the abscisse mean of the visual features
    self.Y0 = 0;
    for i in range(4):
        self.Y0 += msg.data[2*i+1]
    self.Y0 /= 4

def callback_controlMode(self, msg):
    # Updade the control mode
    self.controlMode = msg.data

def head_controller(self):

    # Fixed visual features depth
    z = 1

    # Update the Interaction Matrix
    self.Ly[0][0] = -1/z
    self.Ly[0][1] = self.Y0/z
    self.Ly[0][2] = 1 + (self.Y0*self.Y0)

    # Update the Robot Jacobian
    self.Jb[0][0] = -math.sin(self.thetaP)
    self.Jb[1][0] = math.cos(self.thetaP)
    self.Jb[2][0] = 0

    self.Jb[0][1] = self.cx + self.deltaBH*math.cos(self.thetaP)
    self.Jb[1][1] = -self.cy + self.deltaBH*math.sin(self.thetaP)
    self.Jb[2][1] = -1

    self.Jh[0][0] = self.cx
    self.Jh[1][0] = -self.cy
    self.Jh[2][0] = -1

    # Mobile base command vector
    self.qb[0] = self.v
    self.qb[1] = self.w

    # Compute the IBVS controller
    det = self.Ly.dot(self.Jh)
    tmp = self.Ly.dot(self.Jb)
    tmp = tmp.dot(self.qb)
    self.wh = -1/det*(self.lambdaHead*self.Y0 + tmp)

def init_alphaDistance_controller(self, alpha, d, v, Lambda):

    # Update only in IBVS mode
    if (self.controlMode == 0):

        print("Init controller")

        self.alpha_b = alpha
        self.lambda_s = Lambda
```

```
            self.v= v
            self.d_ref = d

            self.d_inicial = self.current_d

    def alphaDistance_controller(self):

        # Init the controller parameters
        self.init_alphaDistance_controller(math.pi/2, 0.75, 0.2, 1)

        # Computation epsilon
        a = math.copysign(1.0,self.d_ref-self.current_d)
        b = min(abs((self.d_ref-self.current_d)/abs(self.d_ref-self.d_inicial)),1)
        epsilon= a * b

        # Computation of alpha_d
        if self.d_ref > self.d_inicial:
            alpha_d = math.copysign(1.0, self.alpha_b) * math.pi - self.alpha_b
        elif self.d_ref < self.d_inicial:
            alpha_d = self.alpha_b

        # Computation of the task error
        error_s = self.current_theta - self.alpha_b - alpha_d * epsilon

        # Computation of the angular valocity for a given linear velocity
        self.w = self.lambda_s * error_s + (self.v/self.current_d)*math.sin(self.current_theta)


if __name__ == "__main__":

    # Start the node
    rospy.init_node("SpiralController_node")

    time.sleep(1)

    spiral = Spiral_class()

    rospy.spin()
```

APÊNDICE E – IBVS_CONTROLLER

```
#! /usr/bin/python


import rospy
import cv2
import math
import numpy as np

from std_msgs.msg import Float32MultiArray
from sensor_msgs.msg import JointState

class IBVS_class():

    def __init__(self):
        print('IBVS controller node initialization')

        # Subscriber to the visual features topic
        self.image_sub=rospy.Subscriber("/ibvs/vf",Float32MultiArray,self.cb_SubscribeVF)

        # Subscriber to the head state
        self.headSate_sub=rospy.Subscriber("/joint_states",JointState,self.cb_SubscribeHeadState)

        # Publisher: IBVS commands
        self.commands_pub = rospy.Publisher("/ibvs/cmd", Float32MultiArray, queue_size=1)
        self.cmdToPub = Float32MultiArray()
        self.cmdToPub.data = [0.0]*3

        # Robot and camera geometry
        self.f = 1#0.0095 # camera focal
        self.ps = 19e-5 # pixel size
        self.camx = 320.5
        self.camy = 240.5
        self.camf = 522.1910329546544

        self.deltaBH = 0.12 # Distance between the base center and the head
        self.cx = 0#0.096
        self.cy = 0#0.022

        # Robot control vector -----------------------------------------------
        self.cmd = np.zeros((3,1))
        self.v_saturation = 0.5
        self.w_saturation = 0.25
        self.wp_saturation = 0.25


        # Control gain
        lambda_v = 0.1
        lambda_w_wp = 0.1
        self.Lambda = np.zeros((3,3))
        self.Lambda[0][0] = lambda_v
```

```
        self.Lambda[1][1] = lambda_w_wp
        self.Lambda[2][2] = lambda_w_wp

        # Platform orientation
        self.thetaP = 0

        # Visual features vector
        self.vf = np.zeros((8,1))

        # reference visual features vector
        self.vfRef = np.ones((8,1))
        self.vfRef[0] = (157-self.camy)/self.camf # Row
        self.vfRef[1] = -(206-self.camx)/self.camf # Col
        self.vfRef[2] = (157-self.camy)/self.camf
        self.vfRef[3] = -(450-self.camx)/self.camf
        self.vfRef[4] = (320-self.camy)/self.camf
        self.vfRef[5] = -(450-self.camx)/self.camf
        self.vfRef[6] = (320-self.camy)/self.camf
        self.vfRef[7] = -(206-self.camx)/self.camf

        for i in range(len(self.vfRef)):
            self.vfRef[i] = round(self.vfRef[i],3)

        # Interaction Matrix
        self.L = np.zeros((8,3))

        # Robot Jacobian Matrix
        self.J = np.zeros((3,3))

    def cb_SubscribeHeadState(self, msg):
        # Update the head angular position
        print('Joint callback')
        self.thetaP = msg.position[9]

    def cb_SubscribeVF(self, msg):
        # Update the visual features
        print("VF Callback")
        for i in range(8):
            self.vf[i] = msg.data[i]

        # Update the Interaction Matrix
        z = 1 # Fixed visual features depth

        self.L[0][0] = 0
        self.L[0][1] = self.vf[0]/z
        self.L[0][2] = (self.vf[0]*self.vf[1])
        self.L[1][0] = -1/z
        self.L[1][1] = self.vf[1]/z
        self.L[1][2] = 1 + (self.vf[1]*self.vf[1])

        self.L[2][0] = 0
        self.L[2][1] = self.vf[2]/z
        self.L[2][2] = (self.vf[2]*self.vf[3])
        self.L[3][0] = -1/z
        self.L[3][1] = self.vf[3]/z
```

```
        self.L[3][2] = 1 + (self.vf[3]*self.vf[3])

        self.L[4][0] = 0
        self.L[4][1] = self.vf[4]/z
        self.L[4][2] = (self.vf[4]*self.vf[5])
        self.L[5][0] = -1/z
        self.L[5][1] = self.vf[5]/z
        self.L[5][2] = 1 + (self.vf[5]*self.vf[5])

        self.L[6][0] = 0
        self.L[6][1] = self.vf[6]/z
        self.L[6][2] = (self.vf[6]*self.vf[7])
        self.L[7][0] = -1/z
        self.L[7][1] = self.vf[7]/z
        self.L[7][2] = 1 + (self.vf[7]*self.vf[7])

        # Update the Robot Jacobian
        self.J[0][0] = -math.sin(self.thetaP)
        self.J[0][1] = self.cx + self.deltaBH*math.cos(self.thetaP)
        self.J[0][2] = self.cx
        self.J[1][0] = math.cos(self.thetaP)
        self.J[1][1] = -self.cy + self.deltaBH*math.sin(self.thetaP)
        self.J[1][2] = -self.cy
        self.J[2][0] = 0
        self.J[2][1] = -1
        self.J[2][2] = -1

        # Compute the IBVS controller
        tmp = np.linalg.pinv(self.L.dot(self.J))

        # Compute the image error
        e = np.zeros((8,1))
        for i in range(8):
            e[i] = self.vf[i] - self.vfRef[i]

        cmd = -self.Lambda.dot(tmp.dot(e))

        # Saturating the command
        print("Command")

        # Convert to ROS message and publish
        self.cmdToPub.data[0] = cmd[0]
        self.cmdToPub.data[1] = cmd[1]
        self.cmdToPub.data[2] = cmd[2]

        self.commands_pub.publish(self.cmdToPub)



# MAIN ----------------------------------------------------------------------

rospy.init_node('IBVSController_node')
ibvs = IBVS_class()

rospy.spin()
```

APÊNDICE F – MIXER_CONTROLLER

```
#! /usr/bin/python

import rospy as rp

import math
import numpy as np

from geometry_msgs.msg import Twist
from trajectory_msgs.msg import JointTrajectory, JointTrajectoryPoint
from std_msgs.msg import Float32MultiArray as f32MultArray
from std_msgs.msg import Int32
from sensor_msgs.msg import JointState

from visualization_msgs.msg import Marker

class controller_class:

    def __init__(self):

        # Subcrisber to the IBVS commands
        self.sub_ibvsCmd = rp.Subscriber("/ibvs/cmd", f32MultArray, self.callback_ibvsCmd)

        # Subscriber to the spiral following commands
        self.sub_spiralCmd = rp.Subscriber("/spiral/cmd", f32MultArray, self.cb_spiralCmd)

        # Subscriber to the head state
        self.sub_headAngle = rp.Subscriber("/joint_states", JointState, self.cb_headAngle)

        # Subscriber to the spiral center topic
        self.subSpiralCtr=rp.Subscriber("/spiral/center",f32MultArray,self.cbSpiralCtr,qSize=1)

        # Subscriber to the number of obstacles
        self.sub_nbObst = rp.Subscriber("/spiral/nbObst", Int32, self.callback_nbObst, qSize=1)

        # Publisher: control of the mobile base
        self.pub_base = rp.Publisher("/mobile_base_controller/cmd_vel", Twist, qSize=1)

        # Publisher: control of the head
        self.pub_head = rp.Publisher("head_controller/command", JointTrajectory, qSize=1)

        # Publisher: controller mode
        self.pub_ControlMode = rp.Publisher("mix/mode", Int32, qSize=1)

        # Store the ibvs commands
        self.ibvsCmd = [0.0]*3

        # Store the spiral commands
        self.spiralCmd = [0.0]*3

        # Twist message to control the mobile base
```

```
        self.twist = Twist()
        self.twist.linear.x = 0
        self.twist.linear.y = 0
        self.twist.linear.z = 0
        self.twist.angular.x = 0
        self.twist.angular.y = 0
        self.twist.angular.z = 0

        # Trajectory message to control the head
        self.jt = JointTrajectory()
        self.jt.joint_names.append("head_1_joint")
        self.jt.joint_names.append("head_2_joint")

        jtPoint = JointTrajectoryPoint()
        jtPoint.positions = [0]*2
        jtPoint.time_from_start = rp.Duration(0.01)
        self.jt.points.append(jtPoint)

        # Save the head angular velocity
        self.wh = 0.0

        # Head orientation
        self.thetaP = 0.0

        # Spiral center polar coordinates
        self.spiralCenter = [0.0]*2 # [norm, angle]

        # Number of obstacle
        self.nbObst = 0

        # Mode for the simple mixer
        self.mode = 0

        # Control mode (0: ibvs, 1: spiral)
        self.controlMode = 0

    def callback_ibvsCmd(self, msg):
        # Update the ibvs commands
        self.ibvsCmd[0] = msg.data[0]
        self.ibvsCmd[1] = msg.data[1]
        self.ibvsCmd[2] = msg.data[2]

    def cb_spiralCmd(self, msg):
        # Update the spiral commands
        self.spiralCmd[0] = msg.data[0]
        self.spiralCmd[1] = msg.data[1]
        self.spiralCmd[2] = msg.data[2]

    def cb_headAngle(self, msg):
        # Update the head angular position
        self.thetaP = msg.position[9]

    def cbSpiralCtr(self, msg):
        # Update the spiral center values
        self.spiralCenter[0] = msg.data[0]
```

```
        self.spiralCenter[1] = msg.data[1]

    def callback_nbObst(self, msg):
        # Update the number of obstacle
        self.nbObst = msg.data

    def noMixer(self):
        # self.twist.linear.x = 0.2
        # self.twist.angular.z = 0.05
        # self.wh = 0

        self.twist.linear.x = self.spiralCmd[0]
        self.twist.angular.z = self.spiralCmd[1]
        self.wh = self.spiralCmd[2]
        self.controlMode = 1
        # Publish the control mode
        self.pub_ControlMode.publish(self.controlMode)

    # Simplest version based on the distance
    def commandsMixer(self):
        if (self.mode == 0):
            print("ibvs controller")
            self.twist.linear.x = self.ibvsCmd[0]
            self.twist.angular.z = self.ibvsCmd[1]
            self.wh = self.ibvsCmd[2]

            if (self.spiralCenter[0] < 0.8):
                print(self.spiralCenter[0])
                self.mode = 1

            self.controlMode = 0


        elif(self.mode == 1):
            print("spiral controller")
            self.twist.linear.x = self.spiralCmd[0]
            self.twist.angular.z = self.spiralCmd[1]
            self.wh = self.spiralCmd[2]

            if (abs(abs(self.spiralCenter[1]) - math.pi/2) < 3.0*3.14156/180.0):
                self.mode = 2

            self.controlMode = 1

        elif(self.mode == 2):
            print("spiral controller")
            self.twist.linear.x = self.spiralCmd[0]
            self.twist.angular.z = self.spiralCmd[1]
            self.wh = self.spiralCmd[2]

            if (abs(self.thetaP) < 3.0*3.14156/180.0):
                self.mode = 3

            self.controlMode = 1
```

```
        else:
            print("ibvs controller")
            self.twist.linear.x = self.ibvsCmd[0]
            self.twist.angular.z = self.ibvsCmd[1]
            self.wh = self.ibvsCmd[2]

            if (self.spiralCenter[0] > 1.35):
                self.mode = 0

            self.controlMode = 0

        # Publish the control mode
        self.pub_ControlMode.publish(self.controlMode)

    # Second version
    def commandsMixerComplex(self):

        # In the presence of an obstacle
        if (self.nbObst):

            # Current position of the obstacle in the robot frame
            Xo = self.spiralCenter[0] * math.cos(self.spiralCenter[1])
            Yo = self.spiralCenter[0] * math.sin(self.spiralCenter[1])
            # Virtual sampling time
            ts = 1

            # Predict the robot displacement when using ibvs command
            theta_next = self.ibvsCmd[1] * ts

            if(abs(self.ibvsCmd[1]) < 0.0001):
                dX = self.ibvsCmd[0] * ts
                dy = 0
            else:
                dX = (self.ibvsCmd[0] / self.ibvsCmd[1]) * math.sin(theta_next)
                dy = -(self.ibvsCmd[0] / self.ibvsCmd[1]) * (math.cos(theta_next) - 1)

            # Compute the position of the obstacle in the predicted robot frame
            a = math.cos(theta_next)*Xo + math.sin(theta_next)*Yo
            b = math.cos(theta_next)*dX - math.sin(theta_next)*dy
            c = -math.sin(theta_next)*Xo + math.cos(theta_next)*Yo
            d = math.sin(theta_next)*dX - math.cos(theta_next)*dy
            X_next = a - b
            Y_next = c + d

            # Predicted distance
            D_next = math.sqrt(X_next**2 + Y_next**2)

            # Select the control law based on the current and predicted distance
            if(self.spiralCenter[0] <= D_next):
                print("ibvs controller")
                self.twist.linear.x = self.ibvsCmd[0]
                self.twist.angular.z = self.ibvsCmd[1]
                self.wh = self.ibvsCmd[2]
                self.controlMode = 0
            else:
```

```
            print("spiral controller")
            self.twist.linear.x = self.spiralCmd[0]
            self.twist.angular.z = self.spiralCmd[1]
            self.wh = self.spiralCmd[2]
            self.controlMode = 1

    else:
        print("ibvs controller - no obstacle")
        self.twist.linear.x = self.ibvsCmd[0]
        self.twist.angular.z = self.ibvsCmd[1]
        self.wh = self.ibvsCmd[2]
        self.controlMode = 0

    # Publish the control mode
    self.pub_ControlMode.publish(self.controlMode)

# Second version
def commandsMixerComplex2(self):

    # In the presence of an obstacle
    if (self.nbObst):

        # Current position of the obstacle in the robot frame
        Xo = self.spiralCenter[0] * math.cos(self.spiralCenter[1])
        Yo = self.spiralCenter[0] * math.sin(self.spiralCenter[1])
        # Virtual sampling time
        ts = 1

        # Predict the robot displacement when using ibvs command
        theta_next = self.ibvsCmd[1] * ts

        if(abs(self.ibvsCmd[1]) < 0.0001):
            dX = self.ibvsCmd[0] * ts
            dy = 0
        else:
            dX = (self.ibvsCmd[0] / self.ibvsCmd[1]) * math.sin(theta_next)
            dy = -(self.ibvsCmd[0] / self.ibvsCmd[1]) * (math.cos(theta_next) - 1)

        # Compute the position of the obstacle in the predicted robot frame
        a = math.cos(theta_next)*Xo + math.sin(theta_next)*Yo
        b = math.cos(theta_next)*dX - math.sin(theta_next)*dy
        c = -math.sin(theta_next)*Xo + math.cos(theta_next)*Yo
        d = math.sin(theta_next)*dX - math.cos(theta_next)*dy
        X_next = a - b
        Y_next = c + d

        # Predicted distance
        D_next = math.sqrt(X_next**2 + Y_next**2)

        # Compute the obstacle angle
        obstAngle = np.arctan2(Yo,Xo)


        if (self.mode == 0):
            print("ibvs controller - 0")
```

```
            self.twist.linear.x = self.ibvsCmd[0]
            self.twist.angular.z = self.ibvsCmd[1]
            self.wh = self.ibvsCmd[2]

            if ((self.spiralCenter[0] > D_next) and abs(obstAngle) < 45.0*3.14156/180.0):
                print(self.spiralCenter[0])
                self.mode = 1

            self.controlMode = 0

        elif(self.mode == 1):
            print("spiral controller - 1")
            self.twist.linear.x = self.spiralCmd[0]
            self.twist.angular.z = self.spiralCmd[1]
            self.wh = self.spiralCmd[2]

            if (abs(abs(self.spiralCenter[1]) - math.pi/2) < 5.0*3.14156/180.0):
                self.mode = 2

            self.controlMode = 1

        elif(self.mode == 2):
            print("spiral controller - 2")
            self.twist.linear.x = self.spiralCmd[0]
            self.twist.angular.z = self.spiralCmd[1]
            self.wh = self.spiralCmd[2]

            if (abs(self.thetaP) < 5.0*3.14156/180.0):
                self.mode = 3

            self.controlMode = 1

        else:
            print("ibvs controller - 3")
            self.twist.linear.x = self.ibvsCmd[0]
            self.twist.angular.z = self.ibvsCmd[1]
            self.wh = self.ibvsCmd[2]

            if (self.spiralCenter[0] > 1):
                self.mode = 0

            self.controlMode = 0

    else:
        print("ibvs controller - no obstacle")
        self.twist.linear.x = self.ibvsCmd[0]
        self.twist.angular.z = self.ibvsCmd[1]
        self.wh = self.ibvsCmd[2]
        self.mode = 0
        self.controlMode = 0


    # Publish the control mode
    self.pub_ControlMode.publish(self.controlMode)
```

```
rp.init_node('controller_node')
ctr = controller_class()

hz = 100
rate = rp.Rate(hz)

while not rp.is_shutdown():

    # Mix the commands
    ctr.commandsMixer()
    #ctr.commandsMixerComplex2()
    # ctr.noMixer()

    # Publish mobile base commands
    ctr.pub_base.publish(ctr.twist)

    # Publish head command
    ctr.thetaP += ctr.wh*(1.0/hz)
    ctr.jt.points[0].positions[0] =  ctr.thetaP
    ctr.pub_head.publish(ctr.jt)

    rate.sleep()
```