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

CAIO CÉSAR MARINHO DE LIMA

# DETECÇÃO, RASTREAMENTO E ESTIMATIVA DE VELOCIDADE DE OBSTÁCULOS MÓVEIS PARA NAVEGAÇÃO AUTÔNOMA UTILIZANDO LIDAR BIDIMENSIONAL

Recife

2023

CAIO CÉSAR MARINHO DE LIMA

# DETECÇÃO, RASTREAMENTO E ESTIMATIVA DE VELOCIDADE DE OBSTÁCULOS MÓVEIS PARA NAVEGAÇÃO AUTÔNOMA UTILIZANDO LIDAR BIDIMENSIONAL

Monografia submetida ao Departamento de Engenharia Mecânica, da Universidade Federal de Pernambuco - UFPE, para conclusão do curso de Graduação em Engenharia Mecânica

Orientador(a): ADRIEN DURAND-PETITEVILLE

Recife

2023

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

Lima, Caio César Marinho de.
Detecção, rastreamento e estimativa de velocidade de obstáculos móveis para navegação autônoma utilizando lidar bidimensional / Caio César Marinho de Lima. - Recife, 2023.
88 p. : il., tab.

Orientador(a): Adrien Durand-Petiteville
(Graduação) - Universidade Federal de Pernambuco, Centro de Tecnologia e Geociências, , 2023.
Inclui referências, apêndices.

1. Robôs Móveis. 2. Navegação autônoma. 3. Detecção de obstáculos. 4. Rastreamento de obstáculos dinâmicos. I. Durand-Petiteville, Adrien. (Orientação). II. Título.

620 CDD (22.ed.)

**Universidade Federal de Pernambuco**
**Departamento de Engenharia Mecânica Centro de Tecnologia e Geociências- CTG/EEP**

DEMEC

## ATA DE SESSÃO DE DEFESA DE TRABALHO DE CONCLUSÃO DE CURSO – TCC2

Ao 29.° dia do mês de setembro do ano de dois mil e vinte e três, às 10:00 horas, de forma virtual através da plataforma google meet, reuniu-se a banca examinadora para a sessão pública de defesa do Trabalho de Conclusão de Curso em Engenharia Mecânica da Universidade Federal de Pernambuco, intitulado **Detecção, rastreamento e estimativa de velocidade de obstáculos móveis para navegação autônoma utilizando lidar bidimensional**, elaborado pelo aluno **Caio Cesar Marinho de Lima**, matrícula 20160004866, composta pelos avaliadores Prof. **Adrien Durand-Petiteville** (orientador), Prof. **Pedro Manuel Gonzalez Del Foyo** (avaliador) e Prof. **Jose Rodrigues de Oliveira Neto** (avaliador). Após a exposição oral do trabalho, o candidato foi arguido pelos componentes da banca que em seguida reuniram-se e deliberaram pela sua aprovação, atribuindo-lhe a média 8,5, julgando-o apto( ) / inapto( ) à conclusão do curso de Engenharia Mecânica. Para constar, redigi a presente ata aprovada por todos os presentes, que vai assinada pelos membros da banca.

| | | |
|---|---|---|
| Orientador: | Prof. Adrien Durand-Petiteville | Nota: 8,5 |
| Assinatura | _____ | |
| Avaliador Interno: | Prof. Pedro Manuel Gonzalez Del Foyo | Nota: 8,5 |
| Assinatura | _____ | |
| Avaliador Interno: | Prof. Jose Rodrigues de Oliveira Neto | Nota: 8,5 |
| Assinatura | _____ | |

Recife, 29 de setembro de 2023.

Prof. Marcus Costa de Araújo
Coordenador de Trabalho de Conclusão de curso - TCC
Curso de Graduação em Engenharia Mecânica – CTG/EEP-UFPE

*Emitido em 02/10/2023*

**ATA DE APROVACAO Nº 489/2023 - DEMEC (11.65.55)**

**(Nº do Protocolo: NÃO PROTOCOLADO)**

***(Assinado digitalmente em 02/10/2023 17:16 )***
ADRIEN JOAN SYLVAIN DURAND PETITEVILLE
*PROFESSOR DO MAGISTERIO SUPERIOR*
*DEMEC (11.65.55)*
*Matrícula: ###333#9*

***(Assinado digitalmente em 03/10/2023 06:53 )***
JOSE RODRIGUES DE OLIVEIRA NETO
*PROFESSOR DO MAGISTERIO SUPERIOR*
*DEMEC (11.65.55)*
*Matrícula: ###012#3*

***(Assinado digitalmente em 02/10/2023 17:21 )***
PEDRO MANUEL GONZALEZ DEL FOYO
*PROFESSOR DO MAGISTERIO SUPERIOR*
*DEMEC (11.65.55)*
*Matrícula: ###212#8*

Visualize o documento original em http://sipac.ufpe.br/documentos/ informando seu número: **489**, ano: **2023**, tipo: **ATA DE APROVACAO**, data de emissão: **02/10/2023** e o código de verificação: **be91c8819a**

# AGRADECIMENTOS

Aos meus pais, Flávio e Mônica, pela compreensão e pelo suporte necessário para a realização deste trabalho. Ao meu irmão, Carlos, pela companhia e risadas proporcionadas nos momentos de tensão durante essa jornada. Aos meus amigos e colegas que estiveram presentes comigo em noites insones, presencialmente ou pelo discord. Em especial, ao meu amigo Henrique, por ter me apresentado ao material que reacendeu minha paixão pela programação. Ao professor Adrien, pela orientação dedicada e paciente.

*“Ao homem que conhece essa Vontade,*
*não há ‘por quê’ ou ‘por quê não’,*
*nem ‘pode’ ou ‘não pode’; ele é!.”*
*(Jack Parsons)*

RESUMO

Este trabalho busca propor uma abordagem para a detecção e rastreamento de obstáculos dinâmicos no contexto da navegação autônoma. Inicialmente, é apresentada uma contextualização dos conceitos pertinentes ao assunto abordado, assim como uma discussão sobre trabalhos existentes na área. Para o desenvolvimento da solução, se faz necessária a adoção de um modelo para o processamento dos obstáculos no domínio sensorial, assim como a análise de dados para estimativa de velocidade, direção e trajetória dos obstáculos dinâmicos. Os testes são realizados em ambiente controlado, visando a análise qualitativa das técnicas utilizadas e uma avaliação da viabilidade do método escolhido. As técnicas desenvolvidas neste trabalho fornecem, na forma de um nó ROS, os meios necessários para a identificação, rastreamento estimativa de velocidade obstáculos móveis.

**Palavras-chaves:** Robôs Móveis, Navegação autônoma, Detecção de obstáculos, Rastreamento de obstáculos dinâmicos.

ABSTRACT

This work aims to propose an approach for the detection and tracking of dynamic obstacles in the context of autonomous navigation. Initially, a contextualization of concepts related to the subject of autonomous navigation is presented, as well as a discussion about existing works in the area. To develop the solution, it is necessary to adopt a model for processing obstacles in the sensorial space, as well as data analysis to estimate the velocity and trajectory of dynamic obstacles. The tests are carried out in a controlled environment, aiming at the qualitative analysis of the techniques used and an evaluation of the viability of the chosen method. The proposed techniques provide, in the form of a ROS node, the capacity to identify, track and estimate the speed of moving obstacles.

**Key-words**: Mobile Robots, Autonomous Navigation, Obstacle Detection, Dynamic Obstacle Tracking.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

# 1 INTRODUÇÃO

Com o avançar do desenvolvimento tecnológico, robôs estão paulatinamente ganhando espaço em certos setores da economia, se tornando cada vez mais capazes de performar tarefas que antes estavam reservadas a humanos. Para terem a devida flexibilidade de operar em cenários variados, é necessário que essas máquinas sejam capazes de navegar e interagir pelo ambiente para qual foram destinadas.

A navegação autônoma trata do processo de direcionar um sistema robótico de um local para outro independente de interferência humana, levando em consideração os fatores ambientais, as capacidades sensoriais, as características motoras e o comportamento desejado do robô. A garantia de que robôs se movam e naveguem em ambientes não estruturados sem intervenção humana permite que a tecnologia robótica se torne cada vez mais atrativa para uma ampla gama de aplicações, como por exemplo: busca e resgate, atendimento a desastres, exploração, segurança etc. Porém, a flexibilidade oferecida pela navegação autônoma não está restrita à substituição do trabalho humano em situações perigosas, sendo também capaz de tornar robôs aptos a auxiliar idosos e pessoas com deficiências no contexto doméstico.

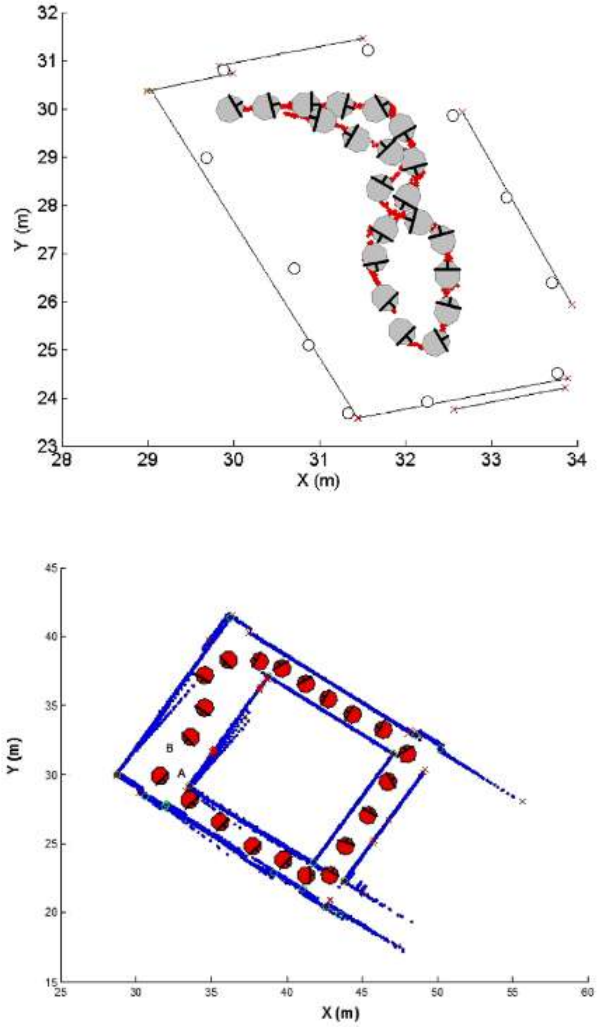
Um dos principais desafios no desenvolvimento da navegação robótica autônoma é garantir que o robô seja capaz de navegar com segurança e eficácia pelo ambiente sem colidir com obstáculos e sem se perder. A escolha da abordagem adequada para alcançar o resultado desejado depende de vários fatores, incluindo as características do ambiente em que o robô irá operar, as restrições de hardware e a complexidade da tarefa a ser realizada.

## 1.1 Classificação de técnicas de navegação

O paradigma clássico na navegação autônoma trata da classificação das abordagens com base na adoção, ou ausência, de um mapa para a localização do robô, dividindo, dessa forma, as técnicas em duas grandes categorias. A primeira faz uso de um mapa do ambiente para ajudar inferir a localização e planejar caminhos pertinentes ao objetivo. Essa abordagem é conhecida como navegação baseada em mapa (*map-based*) e é capaz de garantir uma boa performance para grandes deslocamentos. Todavia, a navegação baseada em mapa é bastante ineficiente para lidar com ambientes de configuração dinâmica, devido a sua inerente baixa flexibilidade.

Dentre as abordagens *map-based* temos algumas subcategorias, como a SLAM (*Simultaneous Localization and Mapping*, ou, Localização e Mapeamento Simultâneos), que envolve a criação simultânea de um mapa do ambiente e a localização do robô nesse mapa. O robô utiliza sensores, como câmeras, LIDAR ou SONAR, para adquirir informações do ambiente e construir um mapa à medida que se move, como é apresentado em Cheein *et al.* (2010). O SLAM é particularmente eficaz em ambientes desconhecidos, permitindo que o robô crie um mapa e determine sua posição relativa em relação a esse mapa, como visto na Figura 1.

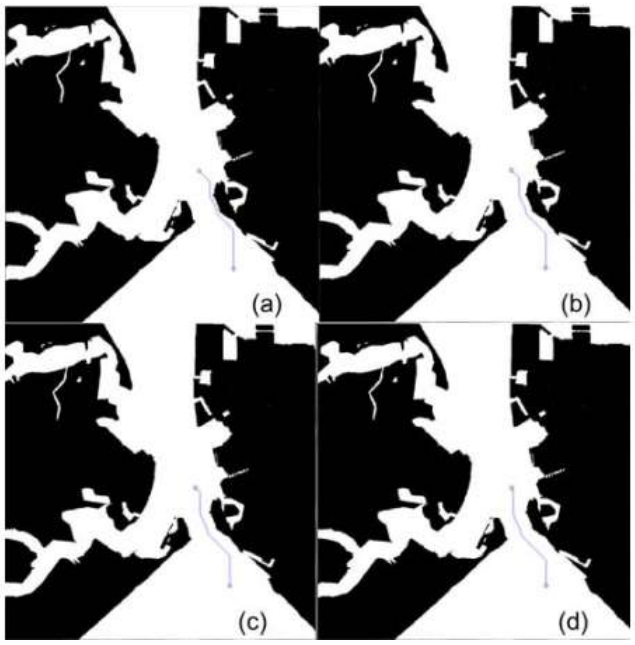
Figura 1 – Trajetória de sistema robótico e mapas construídos por meio de implementação de técnica *SLAM*.




Fonte: Adaptado de Cheein *et al.* (2010).

Quando há um mapa do ambiente disponível, os algoritmos de planejamento de trajetória são frequentemente usados para calcular trajetórias seguras e eficientes com base no mapa do ambiente. Algoritmos como o A* (A-estrela) e Dijkstra (SINGH *et al.*, 2018) são comuns nesse contexto onde são consideradas informações do mapa, como obstáculos e áreas livres, para determinar o caminho ótimo do robô em direção ao seu destino. Um exemplo de navegação autônoma utilizando esse tipo de abordagem é explorada em Singh *et al.* (2018) no contexto marítimo, com trajetórias obtidas mostradas na Figura 2, que retrata a representação gráfica do ambiente obtida por meio de sensores.

Figura 2 – Trajetória de sistema robótico e mapas construídos por meio de implementação de técnica *SLAM*.




Fonte: Singh *et al.* (2018).

Uma abordagem alternativa, conhecida como navegação sem mapa (*mapless*) ou reativa, não requer um mapa global. Em vez disso, sensores são empregados para varredura dos arredores e para determinação de parâmetros que descrevem o estado do robô a cada ponto da navegação. O movimento é então coordenado por controladores de *feedback*, que têm como objetivo a minimização do erro entre os dados sensoriais atuais e a referência. Essa abordagem é mais adequada quando a aplicação envolve ambientes que estão em constante mudança e que não requer um grande deslocamento como objetivo. Por isso, a navegação sem mapa é uma abordagem promissora para navegação autônoma, especialmente em ambientes dinâmicos ou difíceis de mapear.

## 1.2 Métodos de detecção e desvio de obstáculos

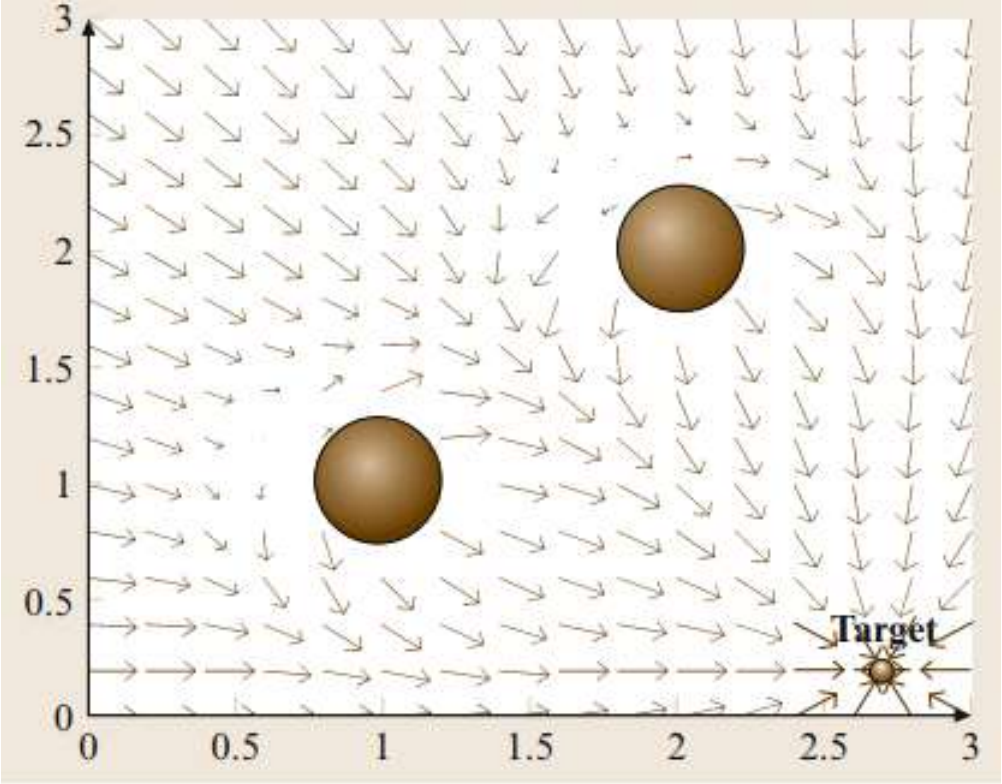
Para a locomoção efetiva e segura de unidades robóticas autônomas, se faz necessária a identificação e caracterização de obstáculos, estáticos ou dinâmicos, para que colisões sejam evitadas, e na busca desse objetivo estão envolvidos o desenvolvimento de algoritmos sofisticados, estratégias de controle e tecnologia de sensores.

A navegação autônoma reativa é o agrupamento de métodos e técnicas que conferem aos robôs móveis a capacidade de navegar por ambientes e realizar decisões em tempo real baseadas em informações obtidas por meios de sensores, evitando obstáculos e chegando a um destino desejado sem depender de mapas ou planejamento extensivo. Em casos de uso, é esperado desses mesmos métodos reativos que eles sejam capazes de lidar com obstáculos dinâmicos, ou seja, aqueles que não permanecem estacionários e podem se mover imprevisivelmente. Para garantir uma navegação segura, é crucial calcular não apenas a posição atual dos obstáculos, mas também suas velocidades. Isso permite que os sistemas de controle reajam adequadamente para evitar colisões ou interferências.

Para os casos com alto grau de imprevisibilidade, como ambientes urbanos, se faz necessário reconhecimento, rastreio e o desvio de obstáculos em movimento, que são muito mais difíceis de evitar. Dentre os desafios impostos pelos obstáculos em movimento está o fato de que eles podem aparecer de repente no caminho do robô, impondo a necessidade de uma reação num curto intervalo de tempo para evitar colisões que podem danificar o robô ou o obstáculo. Nesses casos, os métodos reativos de navegação são mais apropriados, visto que as informações a respeito do ambiente são obtidas e consideradas em tempo real para a tomada de decisão na navegação do robô, permitindo maior flexibilidade ao lidar com obstáculos dinâmicos de difícil modelagem cinemática, por exemplo.

Dentro das técnicas reativas, há métodos que calculam a movimentação em um única etapa e métodos que calculam a movimentação em mais de uma etapa. Os métodos de única etapa derivam a movimentação diretamente da informação obtida por meio de sensores, como métodos heurísticos ou métodos de analogia física, que buscam associar o problema do desvio de obstáculos a algum problema de modelagem física bem descrita, como é o caso do método de campos potenciais, onde o robô é considerado como uma partícula se movendo sob a influência de um campo de forças, com o destino exercendo forças atrativas sobre o robô enquanto os obstáculos o repelem. Um exemplo de visualização do método de campos potenciais pode ser observado na Figura 3.

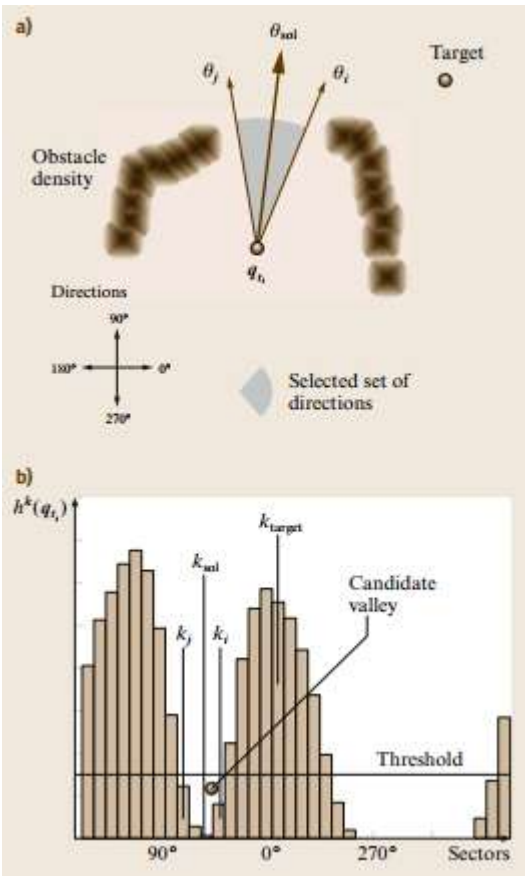
Figura 3 – Visualização das direções de movimento computadas para cada ponto no espaço no método de campos potenciais.




Fonte: Adaptado de Siciliano e Khatib (2008).

Já no caso dos métodos de várias etapas, há um intermédio de informação que separa os dados obtidos pelos sensores da movimentação da unidade robótica. Um exemplo de métodos de múltiplas etapas está nas técnicas que apresentam subconjuntos de controles, como é o caso do histograma do campo de vetores (*Vector Field Histogram*, ou VFH, representado na Figura 4), onde primeiro é obtido um histograma polar do espaço que circunda o robô, representando a densidade de obstáculos distribuídos no ambiente, para que as direções que apresentem valores de densidade menor que um limite sejam adicionadas a um subconjunto de direções possíveis. Em posse do conjunto, o movimento é computado tomando uma das direções dentre as obtidas nos passos anteriores.

Figura 4 – Demonstração do histograma do campo de vetores. Em a) é apresentada a distribuição de obstáculos e em b) o histograma correspondente.




Fonte: Siciliano e Khatib (2008).

## 1.3 Sensores

Todas as abordagens de navegação autônoma reativa precisam de uma interface com o ambiente em que o robô está inserido, ou seja, sensores capazes de captar informações relevantes para a tomada de decisões que regem o seu movimento, o que torna a detecção de obstáculos e, consequentemente a estimativa de suas velocidades fundamentais para uma navegação segura.

Buscando enfrentar os desafios apresentados pelos obstáculos em movimento, os robôs autônomos frequentemente empregam uma variedade de sensores que desempenham um papel crucial na detecção e reação a esses objetos em constante mudança. Sensores como câmeras, radares, lidares e ultrassônicos são comumente utilizados nesse contexto.

As câmeras são amplamente empregadas devido à sua capacidade de fornecer informações visuais detalhadas sobre o ambiente circundante. Elas podem identificar objetos em movimento, estimar suas trajetórias e, assim, contribuir para a detecção

e rastreamento de obstáculos em tempo real. Os radares, por sua vez, são valiosos para a detecção de objetos em movimento, especialmente em condições climáticas adversas, onde a visibilidade pode ser limitada. Lidares, que utilizam lasers para medir a distância até objetos, são excelentes para criar mapas tridimensionais do ambiente, permitindo uma detecção precisa de obstáculos estáticos e móveis. Finalmente, sensores ultrassônicos são úteis para detecção de obstáculos próximos, como paredes ou objetos em movimento muito próximos, e são eficazes em evitar colisões imediatas.

A motivação para o uso desses sensores reside na necessidade de coletar informações detalhadas e em tempo real sobre o ambiente, especialmente quando obstáculos em movimento estão envolvidos. O reconhecimento, rastreamento e desvio eficaz desses obstáculos são cruciais para garantir a segurança do robô e prevenir danos a ele ou a objetos ao seu redor. Portanto, a combinação adequada de sensores é fundamental para possibilitar uma resposta rápida e precisa às ameaças em constante evolução em ambientes complexos, como os urbanos mencionados.

## 1.4 Objetivos

É nesse contexto que o presente trabalho busca, como objetivo geral, a implementação de uma técnica de detecção e estimativa de velocidade de obstáculos móveis por meio de um sensor LIDAR de captura bidimensional.

### 1.4.1 Objetivos específicos

Visando o objetivo geral, será necessário alcançar os seguintes objetivos específicos: aquisição de dados do sensor; segmentação dos dados obtidos em objetos; determinação de pontos de interesse; determinação do deslocamento de obstáculos; estimativa da velocidade dos obstáculos.

## 1.5 Estrutura

O presente trabalho está estruturado da seguinte forma:

- No Capítulo 2 é apresentada uma revisão da literatura com relação à navegação autônoma e rastreamento de obstáculos;
- No Capítulo 3 é apresentada uma fundamentação teórica a respeito da classificação das diferentes técnicas de navegação autônoma,assim como as vantagens e desvantagens relacionadas a cada uma delas, e uma apresentação das ferramentas utilizadas;

- O Capítulo 4 exibe a proposta de metodologia adotada para a execução deste trabalho.
- No Capítulo 5 são exibidos os resultados obtidos com a abordagem tratadas no Capítulo 4, onde são avaliados a extração de pontos de interesse para rastreamento dos obstáculos, a implementação de técnica para a estimativa de velocidade e trajetória e a viabilidade da implementação da técnica proposta;
- Finalmente, na Capítulo 6 são apresentadas as conclusões extraídas do presente trabalho e apontadas sugestões para trabalhos futuros.

# 2 REVISÃO BIBLIOGRÁFICA

A navegação em espaços desordenados é avaliada por Durand-Petiteville *et al.* (2015), onde os autores indicam que o formalismo clássico é limitante e não permite a flexibilidade necessária para lidar com os problemas de navegação autônoma compatíveis com cenários reais. Como alternativa às diretrizes clássicas, é proposta uma abordagem baseada em seis princípios: percepção, modelagem, planejamento, localização, ação e decisão. Os autores argumentam que essa estrutura fornece diretrizes suficientes para auxiliar a elaboração de uma solução a partir desses pontos fundamentais. Utilizando essa abordagem, é proposto um sistema que usa controladores baseados em sensores para navegar por um ambiente desordenado com o auxílio de marcações artificiais para construir um mapa topológico, sendo capaz de evitar obstáculos inesperados ao mesmo tempo que tolera oclusões parciais. Porém, a fim de melhorar a navegabilidade em ambientes humanos, os autores sugerem que ainda se faz necessária a substituição das marcações artificiais por marcos naturais e a implementação de estratégias aptas para lidar com ambientes dinâmicos.

O trabalho apresentado em Dekan *et al.* (2018) propõe uma abordagem para o processamento de dados provenientes de sensores laser medidores de distância para identificação de objetos estáticos e móveis, por meio da implementação em quatro etapas, que são descritas como: segmentação, onde os dados do laser são agrupados em segmentos, de acordo com critérios lógicos, que correspondem aos objetos no ambiente; simplificação, onde a representação de um objeto é reduzida aos seus pontos julgados como significativos; correspondência entre medições sucessivas, que busca relacionar segmentos entre iterações e verificar se o objeto é estático ou móvel; classificação, pela qual os objetos são classificados de acordo com sua variação espacial por meio da avaliação dos pontos de interesse.

Sob o contexto de buscar uma solução para o problema de movimentação de cargas em portos, onde há o uso de veículos de movimentação autônoma em um ambiente dinâmico e de difícil mapeamento, o trabalho desenvolvido por Vaquero *et al.* (2019) propõe um algoritmo de detecção e rastreamento de obstáculos móveis, a fim de garantir uma maior autonomia para esses veículos. A técnica empregada, que consiste na adoção de uma forma primitiva para a representação de obstáculos, seguida pela classificação entre estático e dinâmico para que possa ser realizada a propagação da geometria dos objetos no espaço sensorial do robô, baseando-se na trajetória prevista para a velocidade estimada.

O trabalho desenvolvido em Durand-Petiteville e Cadenat (2021) considera uma unidade robótica de movimentação diferencial equipado com uma câmera como sensor principal, com auxílio de lasers para identificação de obstáculos próximos, para navegação e propõe um modelo preditivo com uma série de restrições para evitar colisões ao longo da trajetória. Os autores validam o modelo por meio de simulações em ambiente virtual e empregam métodos de relaxamento para as restrições de entrada, a fim de garantir estabilidade e diminuição de custo computacional para trajetórias maiores, e para refinamento de mínimos locais encontrados na avaliação da função de custo usada para encontrar a trajetória.

Já em Brito (2022), o autor faz a implementação de um método de navegação multi-controlador, onde o sistema, baseado em critérios estabelecidos no decorrer do trabalho, altera entre um controlador de servo baseado em imagem (*Image Based Visual Servo*, IBVS), que faz uso de dados visuais, e um controlador de seguidor de espiral, que é responsável pelo desvio de obstáculos. A obtenção dos resultados se deu por meio de testes realizados pela integração do ROS, *Robot Operating System*, com um ambiente de simulação. O trabalho ainda sugere que, para estudos futuros, sejam feitos testes em cenários que envolvam obstáculos mais complexos ou mais dinâmicos para avaliar o desempenho da solução proposta.

# 3 FUNDAMENTAÇÃO TEÓRICA

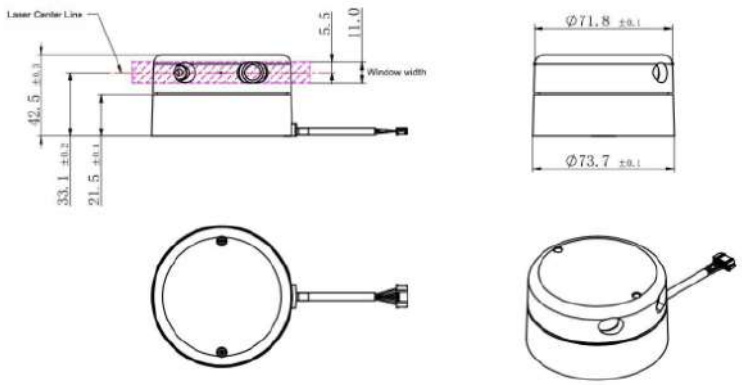
Para o pleno acompanhamento da solução proposta, se faz importante a familiarização prévia com as técnicas e ferramentas que são empregadas no presente trabalho, como o tipo de sensor utilizado, a linguagem de programação empregada, assim como os algoritmos e arquitetura tomadas como base para a implementação.

## 3.1 Lidar

O princípio de funcionamento de um sensor LIDAR (*Light Detection And Ranging*) baseia-se na emissão de pulsos de laser e na consequente medição do tempo que leva para esses pulsos de luz retornarem ao detector após serem refletidos por objetos no ambiente, calculando assim a distância até o objeto. Ao combinar múltiplas medições de distância de diferentes ângulos, um sensor LIDAR pode criar uma representação do ambiente, permitindo a detecção e mapeamento de objetos e terrenos.

Para o desenvolvimento e obtenção de resultados para a técnica de detecção de obstáculos dinâmicos e estimativa de velocidade, optou-se pela utilização de um sensor LIDARmodelo G2 (Figura 5 e Figura 6) da fabricante YDLIDAR, com as especificações expostas na Tabela 1:

Figura 5 – Desenho técnico do sensor YDLIDAR G2.




Fonte:Shenzhen EAI Technology Co.,Ltd. (2019).

Figura 6 – Sensor YDLIDAR G2.

Fonte:Shenzhen EAI Technology Co.,Ltd. (2019).

Tabela 1 – Especificações disponíveis para o sensor YDLIDAR G2.

| **Item** | **Mínimo** | **Típico** | **Máximo** |
|---|---|---|---|
| Frequência do Motor | 5Hz | 7Hz | 12Hz |
| Distância de detecção | 0.12m | - | 16m |
| Campo de visão | - | 0° - 360° | - |
| Resolução de ângulo | 0.36° | 0.504° | 0.864° |

Fonte: Adaptado de Shenzhen EAI Technology Co.,Ltd. (2019).

O preço de sensores lidar no mercado pode variar bastante de acordo com as especificações de distância de detecção, campo de visão e resolução das leituras. Para modelos similares ao escolhido para este projeto, com campo de visão de $360^o$, podem ser encontrados produtos a partir de 50 dólares, para sensores mais simples, e até 30 mil dólares para modelos mais sofisticados.

O sensor utilizado neste trabalho se encontra na faixa dos 150 dólares e foi escolhido por apresentar um preço acessível combinado com um desempenho suficientemente satisfatório para projetos estudantis. Trabalhos futuros buscarão investigar a viabilidade da montagem desse modelo de sensor em um robô móvel de fins educacionais.

## 3.2 ROS

ROS, ou *Robot Operating System*, é uma potente coletânea de *frameworks* de código aberto destinado ao desenvolvimento de aplicações robóticas, amplamente utilizado no contexto de pesquisa e de criação de novas aplicações e sistemas autônomos. O ROS é responsável por realizar a comunicação entre diferentes componentes que podem estar envolvidos em uma aplicação específica, realizando a integração de

sistemas de câmera, laser, GPS etc.

Uma representação possível da arquiterura ROS é a de grafo, onde os processos ocorrem em nós com capacidade de receber e enviar dados pertinentes ao sistema. A rede composta por esses nós pode estar distribuídas sobre diversas máquinas que realizam a comunicação por protocolo TCP/IP. Na Figura 7 está demonstrado um exemplo de uma possível configuração de nós ROS, representados por elipses.

Figura 7 – Exemplo de comunicação de nós ROS.

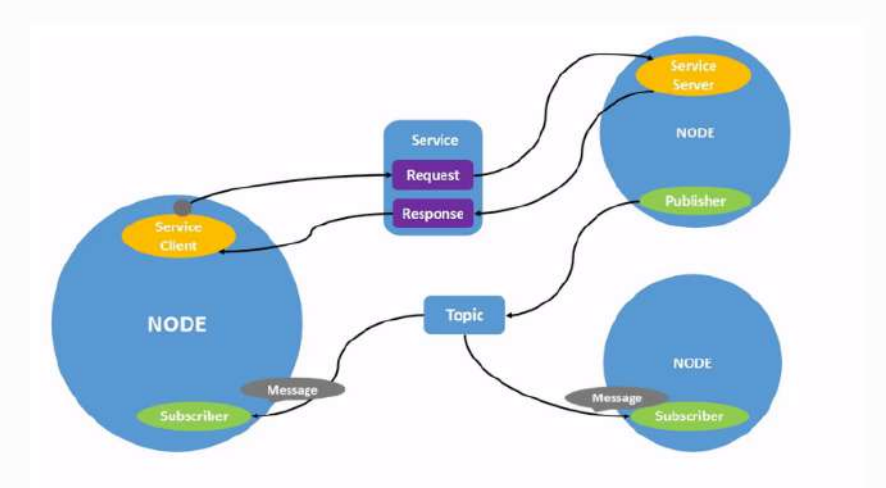

Fonte: Embarcados (2022)

.

Para o estabelecimento de comunicação entre os nós, é necessária a existência de estrutura de dados referidas como **mensagens**, que trazem em si informações sobre o tipo de dados que serão trafegados sob as diferentes mensagens. Por sua vez, os tópicos, no contexto do ROS, são a maneira pela qual um fluxo de mensagens é mantido entre os nós, num esquema de *Publisher/Subscriber*, onde o *Publisher* é aquele responsável pela emissão de dados em um tópico e os *Subscribers* são aqueles nós responsáveis pela recepção dessas mensagens emitidas. Por exemplo, os dados obtidos por um nó ROS responsável por estabelecer comunicação com uma câmera, podem ser enviados para um nó responsável pelo processamento por meio do uso de um tópico chamado **imageTopic**, que envia mensagens do tipo **Image**.

Outra maneira de transmitir dados entre diferentes nós está na criação de **serviços**, que funcionam por chamadas síncronas, ou *requests*, enviadas por um nó cliente a serviço que se encarrega de iniciar uma tarefa no nó servidor. O cliente realiza a solicitação ao serviço e aguarda a resposta antes de dar continuidade à execução. A principal diferença entre a comunicação por tópicos e a comunicação por serviços está no aspecto assíncrono do primeiro modo, onde mensagens estão sendo publicadas periodicamente em um tópico, independente de ter *subscribers* fazendo uso da informação, enquanto a comunicação por serviços segue a lógica síncrona, visto que o cliente precisa aguardar a resposta do serviço. A Figura 8 exemplifica os possíveis tipos de comunicações entre os nós ROS, representados pelos círculos azuis. O tópico, representado pelo retângulo azul inferior, demonstra o fluxo de mensagens (balões cinza) partindo do nó *Publisher* para os nós *Subscribers*. Já o retângulo superior representa o serviço que liga, por meio de solicitações e respostas, o nó cliente (círculo azul à esquerda) ao nó servidor (círculo na direita superior).

Figura 8 – Diagrama da comunicação de nós ROS.




Fonte: ROS 2 Documentation: Foxy (2023).

Por meio dessas conexões, é possível fazer uso da modularidade para contornar as limitações de poder computacional disponível em sistemas embarcados, por exemplo, destinando o processamento de atividades mais custosas para máquinas mais poderosas.

No contexto do projeto, o ROS surge como a ferramenta que possibilita a criação de um nó destinado ao processamento de dados obtidos pelo Lidar. A vantagem dessa abordagem está na modularidade e no potencial para integrar os resultados aqui obtidos em futuros projetos de maior escopo.

## 3.3 Linguagem de programação

É utilizado no projeto para implementação de rotinas, principalmente, Python, que é uma linguagem de programação de alto nível, interpretada, dinâmica e multiplataforma. Possui um grande conjunto de bibliotecas, que a tornam extremamente poderosa e versátil. É uma linguagem eficiente, expressiva e de fácil compreensão, o que a torna ideal para aplicações em diversas áreas, desde scripts simples até aplicações complexas (PYTHON, 2023).

## 3.4 ICP

O algoritmo de ponto mais próximo iterativo (*Iterative Closest Point*, ou ICP) é uma técnica fundamental no campo da visão computacional e da robótica que pode vir a desempenhar um papel crucial em aplicações como navegação autônoma, mapeamento 3D e reconhecimento de objetos.

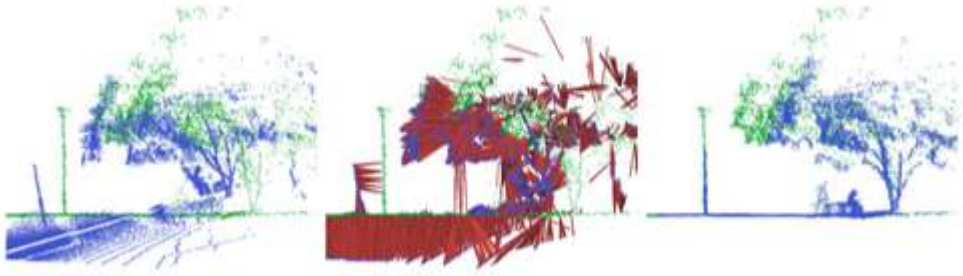
O ICP é usado principalmente para alinhar duas ou mais nuvens de pontos ou conjuntos de pontos no espaço. No contexto da navegação autônoma, o ICP é normalmente empregado para alinhar uma nuvem de pontos ao vivo ou observada de um sensor (como câmera LiDAR ou RGB-D) com uma nuvem de pontos de referência, muitas vezes um mapa pré-construído do ambiente.

Para a implementação proposta neste trabalho, é importante que nuvens de pontos consecutivas estejam alinhadas a fim de avaliar deslocamentos dos obstáculos detectados ao mesmo tempo que minimiza os efeitos de movimento aparente causados pela movimentação do sensor.

O algoritmo funciona de forma iterativa e busca minimizar a diferença entre os pontos correspondentes nas duas nuvens de pontos (Figura 9), estimando a transformação (translação e rotação) necessária para alinhar a nuvem de pontos observada com a nuvem de pontos de referência. As etapas básicas do algoritmo ICP são as seguintes:

1. Identificação de pares de pontos entre as nuvens de pontos observados e de referência que provavelmente correspondem ao mesmo ponto físico no ambiente.
2. Estimativa da transformação que minimize a diferença entre os pontos correspondentes.
3. Aplicação da transformação estimada à nuvem de pontos observada.
4. Repetição das etapas acima até que a convergência ou um número predefinido de iterações seja alcançado.

Figura 9 – Exemplos de registro geométrico entre uma nuvem de pontos de referência (pontos em verde claro) e uma nuvem de pontos de leitura (pontos em azul escuro). Esquerda: Posição inicial das duas nuvens de pontos. Meio: Erro de alinhamento (linhas em vermelho escuro). Direita: Alinhamento final das duas nuvens de pontos.



Fonte: Pomerleau *et al.* (2015).

Na navegação autônoma, o ICP pode ter aplicações que incluem, por exemplo, o mapeamento e localização da unidade robótica (MENDES *et al.*, 2016); a calibração de sensores, ao realizar um alinhamento dos dados sensoriais com dados de referência (FUKUSHIMA, 2018). O presente trabalho busca fazer uso do algoritmo na prevenção de obstáculos, por meio da análise de sucessivas nuvens de pontos, servindo como ferramenta para a detecção de obstáculos em movimento.

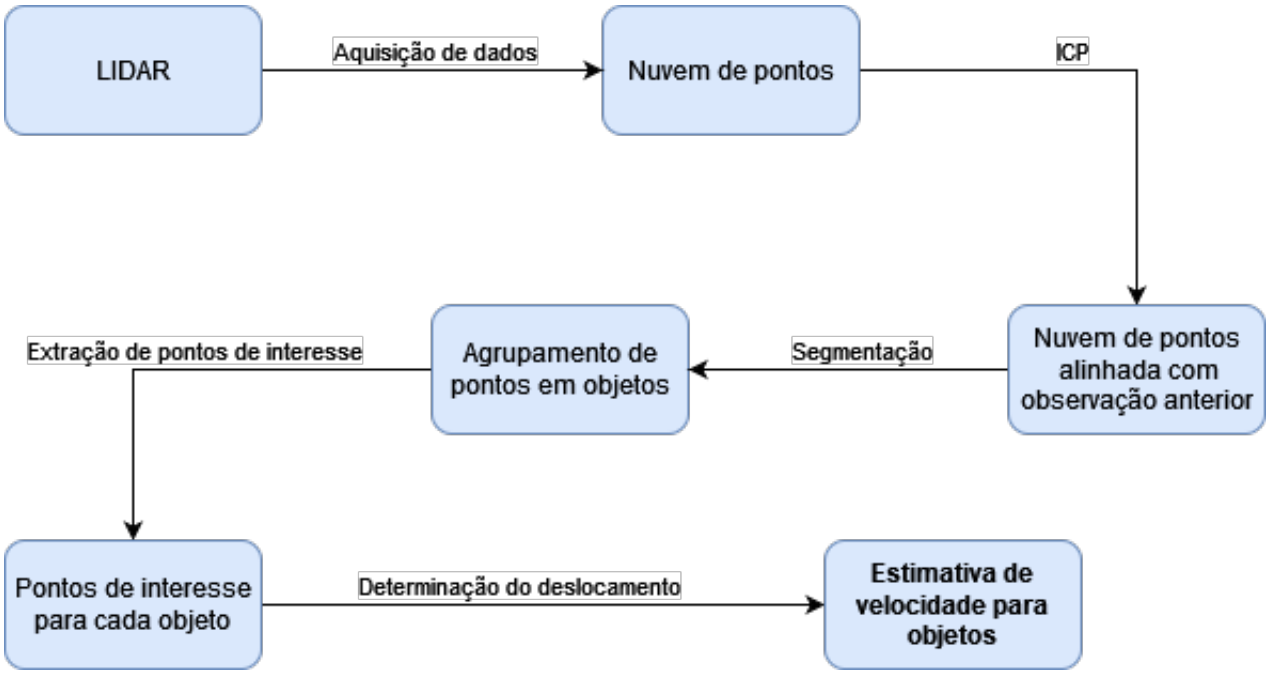
# 4 METODOLOGIA

O trabalho se propõe a utilizar técnicas anteriormente desenvolvidas para processamento de dados sensoriais na identificação de obstáculos móveis e na estimativa de velocidade desses objetos, a fim de dar continuidade ao trabalho desenvolvido por Silva (2022) e Brito (2022), enquanto estabelece uma base para futuros trabalhos que busquem empregar sensores Lidar na navegação autônoma de unidades robóticas.

## 4.1 Proposta

Visando a completude dos objetivos definidos em 1.4.1, o presente trabalho busca oferecer uma abordagem para lidar com obstáculos móveis por meio da identificação, rastreio e consequentes estimativas de velocidade e trajetória, fazendo uso do *framework* ROS (*Robot Operating System*), por meio do qual serão implementados métodos de processamento de dados que visam a extração de pontos de interesse dos possíveis obstáculos presentes em um dado cenário. Em posse dos pontos de interesse, é possível realizar estimativas de velocidade e trajetória, fazendo o rastreamento do obstáculo nos arredores do robô. A Figura 10 demonstra o fluxo aqui proposto.

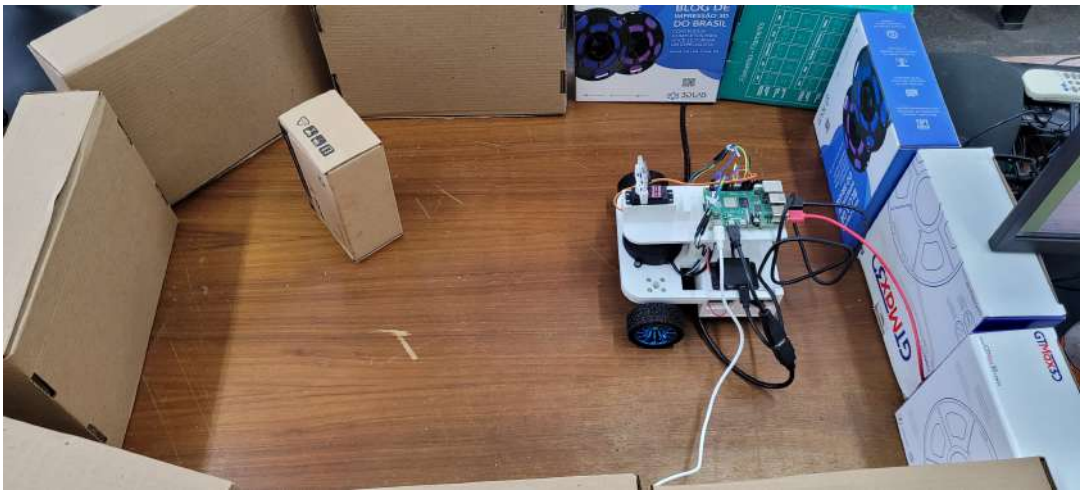
Figura 10 – Fluxo proposto neste trabalho para aquisição da estimativa de velocidade para obstáculos dinâmicos.




Fonte: Próprio autor.

A fim de ilustrar as diferentes etapas de processamento, é estabelecido o cenário 1, apresentado na Figura 11, que apresenta um LIDAR do modelo YDLIDAR G2 montado sobre uma unidade robótica estática, rodeados por caixas de papelão no papel de obstáculos estáticos.

Figura 11 – Cenário 1, proposto para ilustrar as etapas de processamento.



Fonte: Próprio autor.

## 4.2 Aquisição de dados

A fim de alcançar uma solução mais robusta, optou-se por fazer uso de dados provenientes de um LiDAR real durante o desenvolvimento e a testagem dos algoritmos propostos. Essa abordagem possibilita identificar as particularidades inerentes à natureza dos sensores digitais e, consequentemente, tomar decisões de projeto apropriadas para mitigar os efeitos indesejados no funcionamento normal da aplicação.

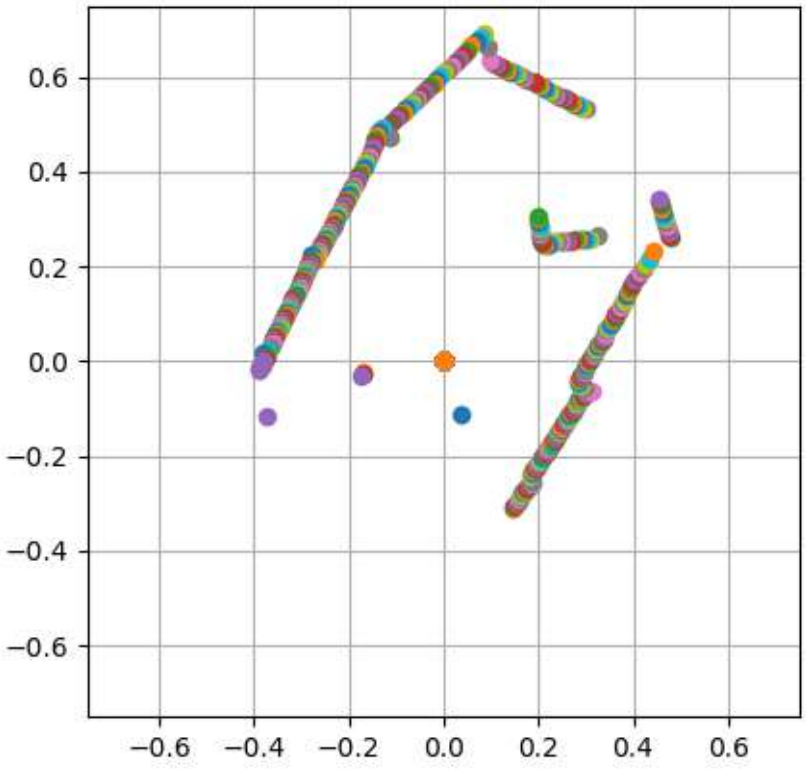
O fabricante do sensor utilizado disponibiliza um kit de desenvolvimento de software e um *workspace* ROS que conta com um nó de publicação responsável por enviar informações do sensor para qualquer módulo que esteja inscrito neste nó.

Os dados obtidos para o cenário 1 estão dispostos na Figura 12, com os pontos ilustrados por círculos coloridos, a posição do sensor centrada na origem da imagem e com a escala dos eixos em metros.

## 4.3 Processamento de dados

A etapa de processamento de dados é uma das etapas mais importantes para o desenvolvimento de uma solução que funcione em tempo real. Após a aquisição de dados por meio de sensores, como o LiDAR, é necessário realizar uma série de operações para que esses dados sejam representativos e úteis. Essas operações visam

Figura 12 – Visualização dos dados adquiridos para o cenário 1.

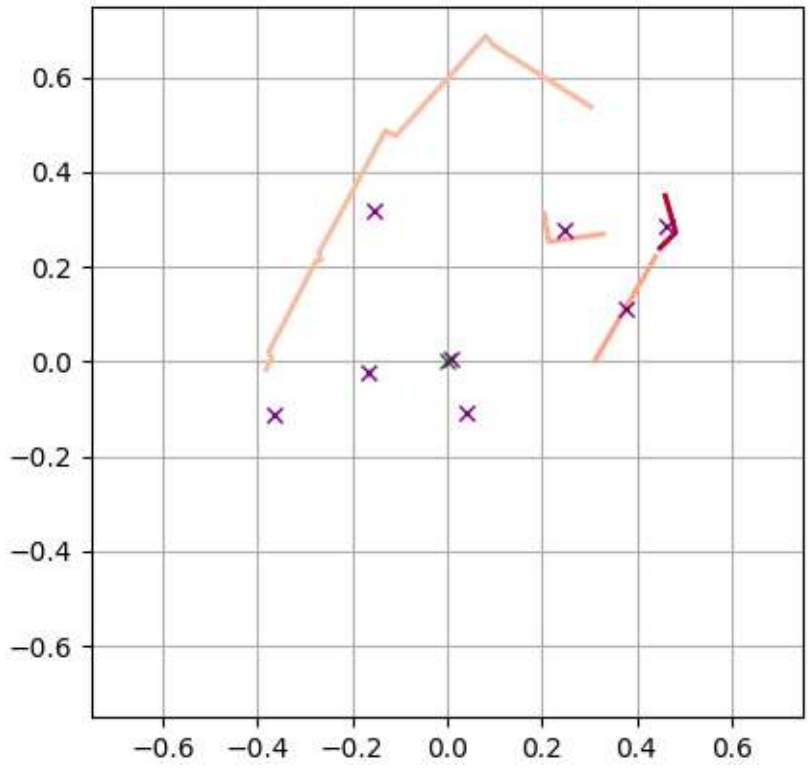



Fonte: Próprio autor.

otimizar a quantidade de dados trafegada e analisada para que algoritmos responsáveis pela navegação possam ter um tempo de resposta adequado.

A primeira etapa do processamento consiste na filtragem e segmentação dos dados provenientes do sensor. Baseando-se nos critérios de decisão descritos em Dekan *et al.* (2018), foi possível implementar operações que reduzem significativamente a quantidade de pontos necessários para a representação de um objeto, estabelecendo critérios de agrupamento baseados em distância e ângulo entre pontos. Um exemplo da segmentação para o cenário 1 é mostrado na Figura 13, onde pontos pertencentes aos mesmos objetos estão ligados por linhas contínuas e o centro geométrico da distribuição de pontos de cada objeto está marcado com X da cor roxa na imagem.

Figura 13 – Segmentação aplicada ao cenário 1.



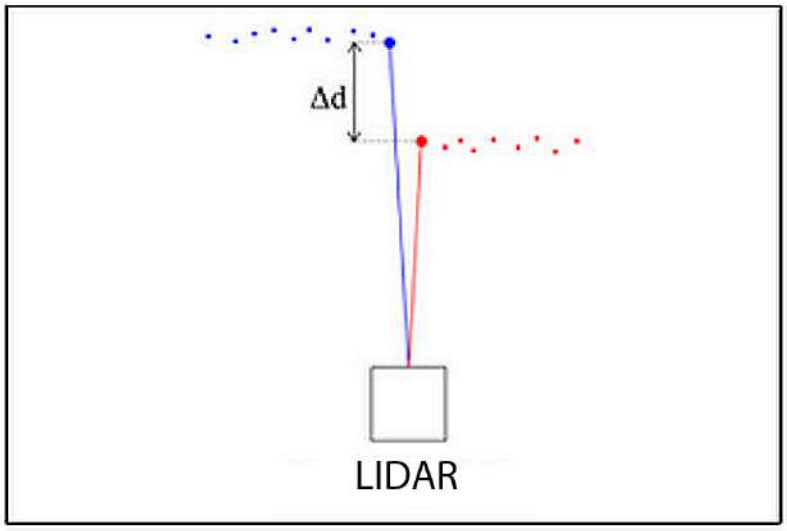

Fonte: Próprio autor.

O princípio fundamental da segmentação está apresentado na Figura 14 e trata da distância de pontos subsequentes que, se maior que um limite parametrizado, é indicativa de um novo segmento. Outro critério utilizado trata do ângulo entre pontos obtidos na leitura e busca excluir pontos preservando a fidelidade da representação do objeto real. O ângulo entre pontos utilizado como critério de exclusão é determinado pela Equação 1, onde $\alpha_{j-i}$ é o ângulo entre pontos consecutivos (Figura 15), $l$ e representa a distância do sensor aos pontos de índice $i$ e $j$.

$$\beta_{i,j} = a \cdot cos\left(\frac{l_j - l_i cos(\alpha_{j-i})}{\sqrt{l_i^2 + l_j^2 - 2 l_i l_j cos(\alpha_{j-i})}}\right) \tag{1}$$

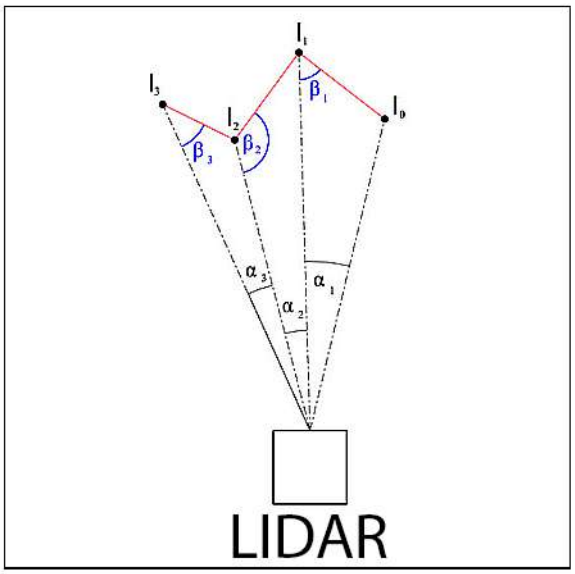
Em posse de $\beta_{i,j}$, é estabelecido um limite para ângulo $p$ que é comparado com o módulo da diferença entre o $\beta_{i-1-k,i}$ (relacionado ao último ângulo considerado como parte do objeto) e $\beta_{i,i+1}$ (relacionado ao ponto $l_i$, ao qual se deseja determinar se fará ou não parte do objeto). A representação do critério de exclusão pode ser definida de acordo com a Equação 2:

Figura 14 – Demonstração do princípio de limite de distância. A figura mostra dois segmentos, indicados pelas cores vermelho e azul.




Fonte: Adaptado de Dekan *et al.* (2018).

Figura 15 – Demonstração dos ângulos utilizados no critério de exclusão de pontos.

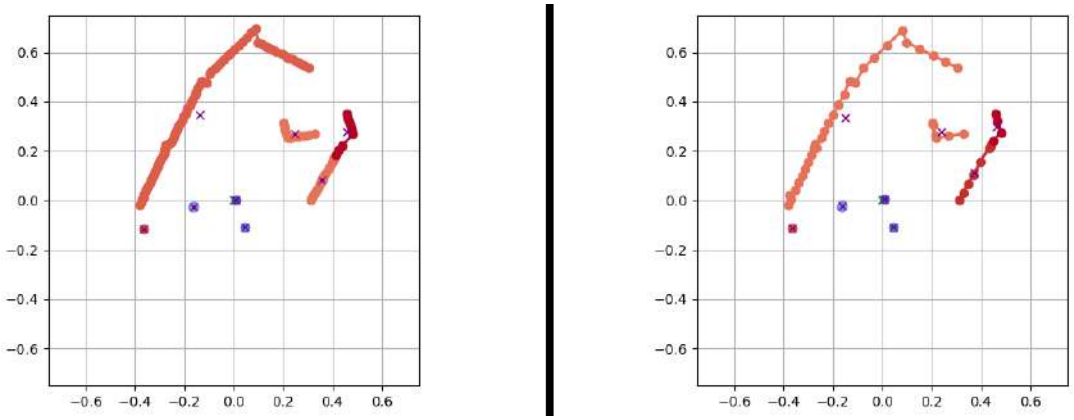



Fonte: Adaptado de Dekan *et al.* (2018).

$$F_{(l_i)} = (|\beta_{i-1-k,i} - \beta_{i,i+1}|) < p \tag{2}$$

Estabelecido o critério de exclusão, foi então definido o número máximo de pontos excluídos consecutivamente, uma vez que a exclusão baseada somente no critério de ângulo poderia resultar em estimativas pouco representativas da geometria para objetos de superfície com curvatura suave. Na Figura 16 é possível observar o efeito da exclusão de pontos baseada no critério de ângulo. Os círculos na imagem representam os pontos e os objetos são separados por cores e pelas linhas que ligam os pontos constituintes.

Figura 16 – À esquerda, segmentação do cenário 1 sem critério de exclusão de ângulos; à direita, aplicado o critério de exclusão de ângulos com o máximo de 5 pontos consecutivos excluídos.




Fonte: Próprio autor.

Outro passo importante no processamento de dados ocorre após a segmentação inicial, que é a união de objetos aparentemente separados. Do ponto de vista do sensor, um objeto contínuo pode representar uma descontinuidade em sua representação caso um outro objeto se interponha entre ele o sensor.

### 4.3.1 Pontos significativos

A caracterização dos obstáculos ocorre por meio da identificação dos chamados pontos significativos, que seriam equivalentes a locais de interesse por apresentar características distintas e únicas, sendo muito importantes para o rastreamento do obstáculo ao longo das diferentes medições.

Os primeiros pontos significativos e mais facilmente identificáveis para cada objeto são as extremidades na nuvem de pontos que representa cada obstáculo após a segmentação. Em seguida, como proposto em Dekan *et al.* (2018), é realizada a busca por pontos significativos internos que representem uma mudança drástica na geometria observada com critérios parametrizados de ângulo entre os pontos que descrevem o objeto. A Equação 3 define os critérios de decisão para determinação dos pontos significativos internos, sendo um ponto $l_i$ considerado significativo quando $F_t(l_i, k) = 1$.

$$F_t(l_i,k) = \begin{cases} 0, k = 0 & \|l_i - l_{i-1}\| < mLen \\ 0, k = 0 & \|l_i - l_{i-1}\| > mLen \wedge \|l_i - l_j\| > mLen \wedge \gamma_{i,j} \notin \langle \delta_m, \delta_M \rangle \\ F_t(l_i, k = k+1) & \|l_i - l_{i-1}\| > mLen \wedge \|l_i - l_j\| < mLen \\ 1, k = 0 & \|l_i - l_{i-1}\| > mLen \wedge \|l_i - l_j\| > mLen \wedge \gamma_{i,j} \in \langle \delta_m, \delta_M \rangle \end{cases} \tag{3}$$

Sendo:

$$j = i + 1 + k \tag{4}$$

$$\gamma_{i,j} = 180^\circ - (\beta_{i-1,j} + \alpha_i) + \beta_{i,j} \tag{5}$$

Os parâmetros $mLen$, $\delta_m$ e $\delta_M$ são, respectivamente, a distância mínima entre pontos significativos, o limite inferior e superior do ângulo $\gamma$, sendo $\gamma_{i,j}$ o ângulo entre dois segmentos de reta.

A Equação 3 pode ser entendida como uma função que retorna 0 (ou seja, falso, para o caso onde o ponto avaliado $l_i$ não é um ponto significativo) ou 1 (ou seja, verdadeiro, para o caso onde o ponto avaliado $l_i$ é, de fato um ponto significativo).

Há duas condições onde $F_t(l_i, k)$ tem retorno falso:

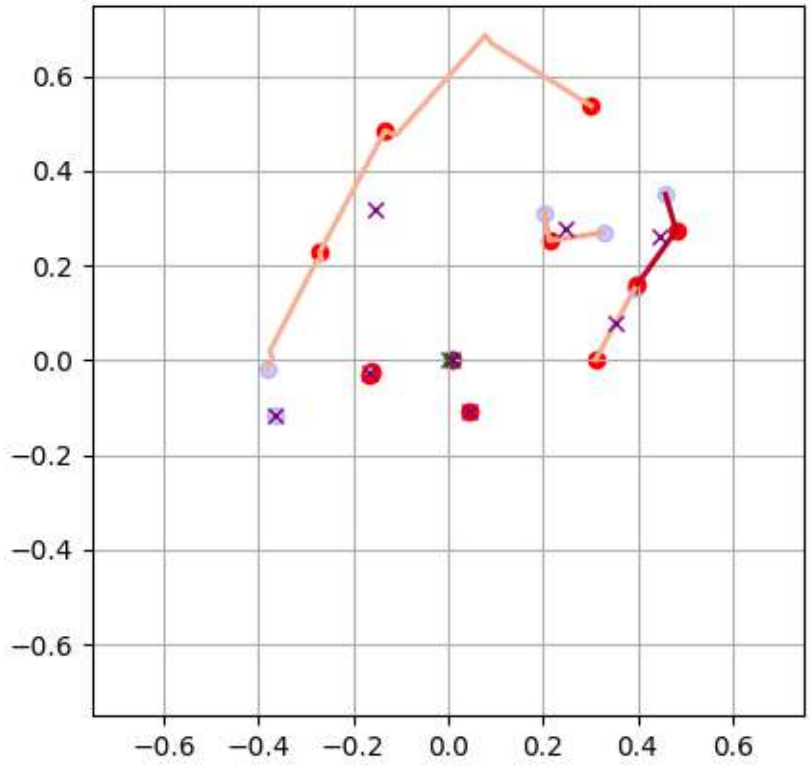
- Caso o módulo da distância entre o ponto avaliado, $l_i$, e o ponto anterior pertencente ao objeto, $l_{i-1}$ seja **menor** que $mLen$;
- Caso o módulo da distância entre $l_i$ e $l_{i-1}$ seja **maior** que $mLen$ **e** o módulo da distância entre $l_i$ e $l_j$ seja **maior** que $mLen$ **mas** o ângulo entre o segmento de reta descrito entre $l_i$ e $l_{i-1}$ e o segmento de reta descrito entre $l_i$ e $l_j$ esteja fora dos limites parametrizados para o ângulo.

Enquanto que, para que $F_t(l_i, k)$ seja avaliada como verdadeira, a única condição necessária é a de que o módulo da distância entre $l_i$ e $l_{i-1}$ seja **maior** que $mLen$ **E** o módulo da distância entre $l_i$ e $l_j$ seja **maior** que $mLen$ **e** o ângulo entre o segmento de reta descrito entre $l_i$ e $l_{i-1}$ e o segmento de reta descrito entre $l_i$ e $l_j$ esteja dentro dos limites parametrizados para o ângulo.

Ainda na Equação 3, $k$ representa o aspecto iterativo da função, uma vez que a condição apresentada na terceira linha esteja satisfeita (o que ocorre quando o módulo da distância entre $l_i l_{i-1}$ é **maior** que $mLen$ **mas** o módulo da distância entre $l_i$ e $l_j$ é **menor** que $mLen$), a resposta da função $F_t$ é a própria $F_t$ com o parâmetro $k$ incrementado de 1 unidade. Assim sendo, $k$ pode ser entendido como o incremento ao índice do ponto avaliado, ou seja, sendo o ponto $l_i$ o ponto passado à função $F_t$, quando $k = 0$, teríamos $j = i+1+0 = i+1$, tornando $l_j$ o ponto seguinte ao $l_i$ ($l_j = l_i+2$ para $k = 1, l_j = l_i + 3$ para $k = 2$...).

A Figura 17 demonstra a representação de pontos significativos obtidos com o algoritmo descrito.

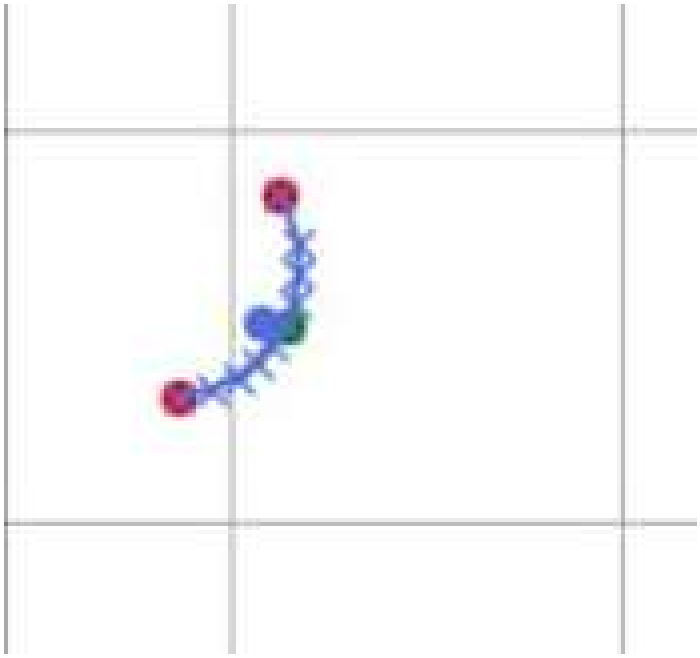
Figura 17 – Extração de pontos significativos para o cenário 1. Os pontos significativos estão marcados na imagem como círculos.




Fonte: Próprio autor.

Por fim, caso a etapa anterior não identifique pontos significativos internos, é ainda realizada mais uma avaliação ao conjunto de pontos que busca identificar um ponto de interesse por um critério de excentricidade, para o caso de superfícies curvas. A decisão é realizada encontrando o ponto do objeto mais distante das suas extremidades $l_i$. Em seguida, é avaliada a distância desse ponto $l_i$ ao segmento de reta descrito pelas extremidades, de comprimento $L$. Caso a distância de $l_i$ ao segmento de reta seja maior que uma fração de $L$ (limite parametrizado), o ponto é marcado como ponto significativo interno. Segundo Dekan *et al.* (2018) um limite apropriado seria o de $1/5L$. Um exemplo de objeto curvo com ponto significativo interno é mostrado na Figura 18:

Figura 18 – Exemplo de objeto de superfície curva com ponto significativo interno obtido pelo critério de excentricidade. Os círculos vermelhos representam as extremidades e o ponto significativo interno está marcado em verde.



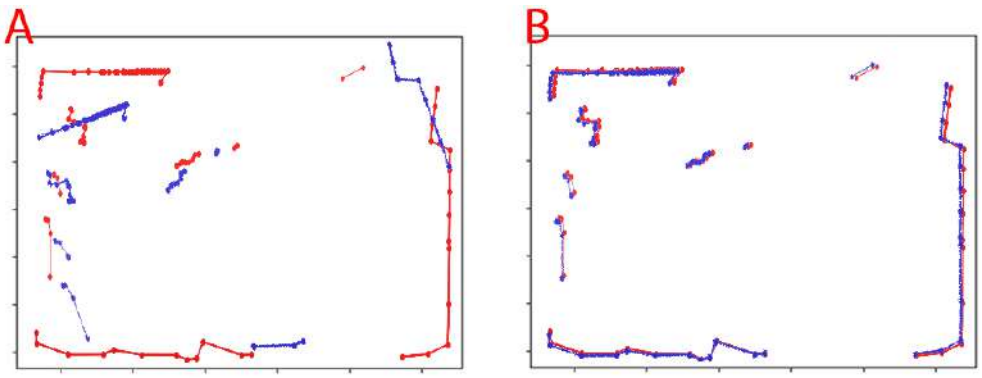
Fonte: Próprio autor.

## 4.4 Determinação de deslocamento

Em posse dos pontos significativos que caracterizam os obstáculos previamente segmentados, é feita uma análise que compara a observação mais recente com a última observação realizada. Nesta análise, é realizada uma correspondência entre os pontos significativos das duas observações, a fim de identificar a correspondência de pontos entre as observações. A fim de realizar comparação e pareamento que não sejam afetados pelo movimento relativo sensor-obstáculos, as observações inicialmente passam por um processamento para alinhar a leitura dos pontos.

Na Figura 19, há, em vermelho, a nuvem de pontos da última observação processada e, em azul, nova nuvem de pontos a ser processada, tendo A-B apresenta a transição de estados antes e depois do alinhamento. A utilização dessa técnica assume a hipótese que, para as observações utilizadas na análise do cenário proposto, haverá, majoritariamente, pontos que correspondam a obstáculos estáticos.

Figura 19 – Ilustração do processo de alinhamento que ocorre com a nuvem de pontos pela aplicação da técnica de ICP.




Fonte: Próprio autor.

A correspondência ocorre entre os pontos significativos que apresentam menor distância entre uma observação e outra. O deslocamento do obstáculo então é definido como a média dos deslocamentos dos seus pontos significativos. Em posse do deslocamento do obstáculo, podemos inferir a velocidade instantânea do objeto de acordo com a Equação 6:

$$v = \frac{\Delta x_{ij}}{\Delta t_{ij}} = \Delta x_{ij} \cdot f_{lidar} \tag{6}$$

Sendo $v$ a velocidade no instante avaliado, $\Delta x_{ij}$ e $\Delta t_{ij}$ o deslocamento e o tempo entre as observações, respectivamente, e $f_{lidar}$ a frequência de varredura do sensor Lidar.

# 5 RESULTADOS E DISCUSSÕES

O código responsável pela obtenção e processamento de dados foi executado em um computador com um processador Intel i5-8265U da oitava geração, com 8GB de memória RAM e placa de vídeo dedicada GeForce MX110.

O tempo de processamento de cada ciclo levou, em média, 90ms, incluindo a geração dos quadros para a representação gráfica em tempo real, que é computacionalmente custosa. O tempo médio de processamento ficou abaixo da frequência do laser, caso contrário, poderia ocorrer de ter varreduras que ficariam de fora da análise, prejudicando significativamente os resultados.

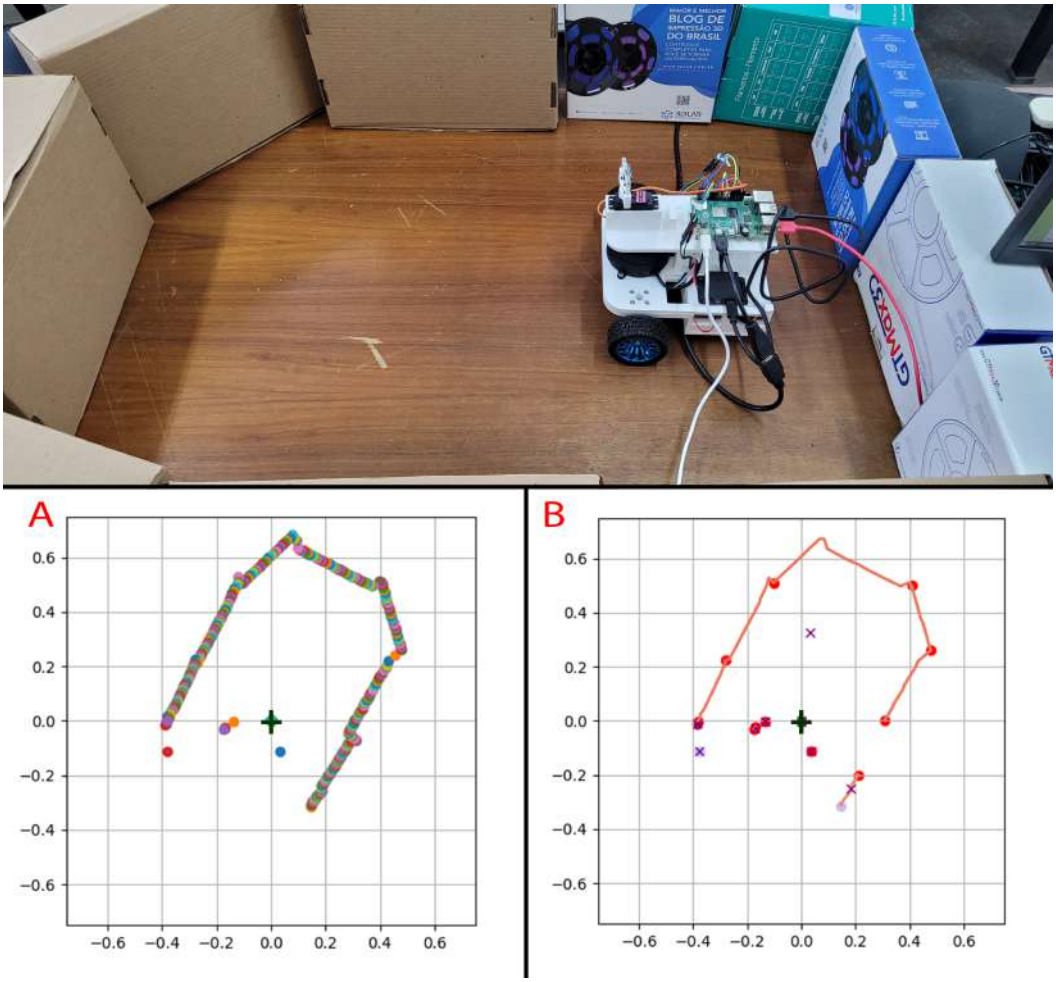
Durante os testes, o YDLIDAR G2 demonstrou certa instabilidade na leitura dos pontos, demonstrando oscilações significativas mesmo para pontos derivados de obstáculos estáticos. A resolução do sensor também pode se tornar um fator limitante para a aplicação proposta à medida que a distância de detecção aumenta, uma vez que o número de pontos que representam o obstáculo pode se tornar pouco representativo para a análise desejada e, por isso, foi decidido que os cenários de testes se limitariam a cenários em um raio de até 1 metro.

## 5.1 Aquisição e processamento preliminar dos dados

A fim de avaliar o processamento da nuvem de pontos obtida pelo sensor, são tomadas cinco variações do cenário 1, apresentado na seção 4.1. Para as figuras que ilustram os diferentes cenários, está determinada a apresentação da fotografia do ambiente, assim como uma captura da nuvem de pontos antes (à esquerda) e depois (à direita) do processamento ser realizado.

O cenário 2, mostrado na Figura 20, é caracterizado pela ausência de obstáculos entre o robô e as paredes que o cercam. A segmentação sugere a descrição de 6 obstáculos estáticos em cena.

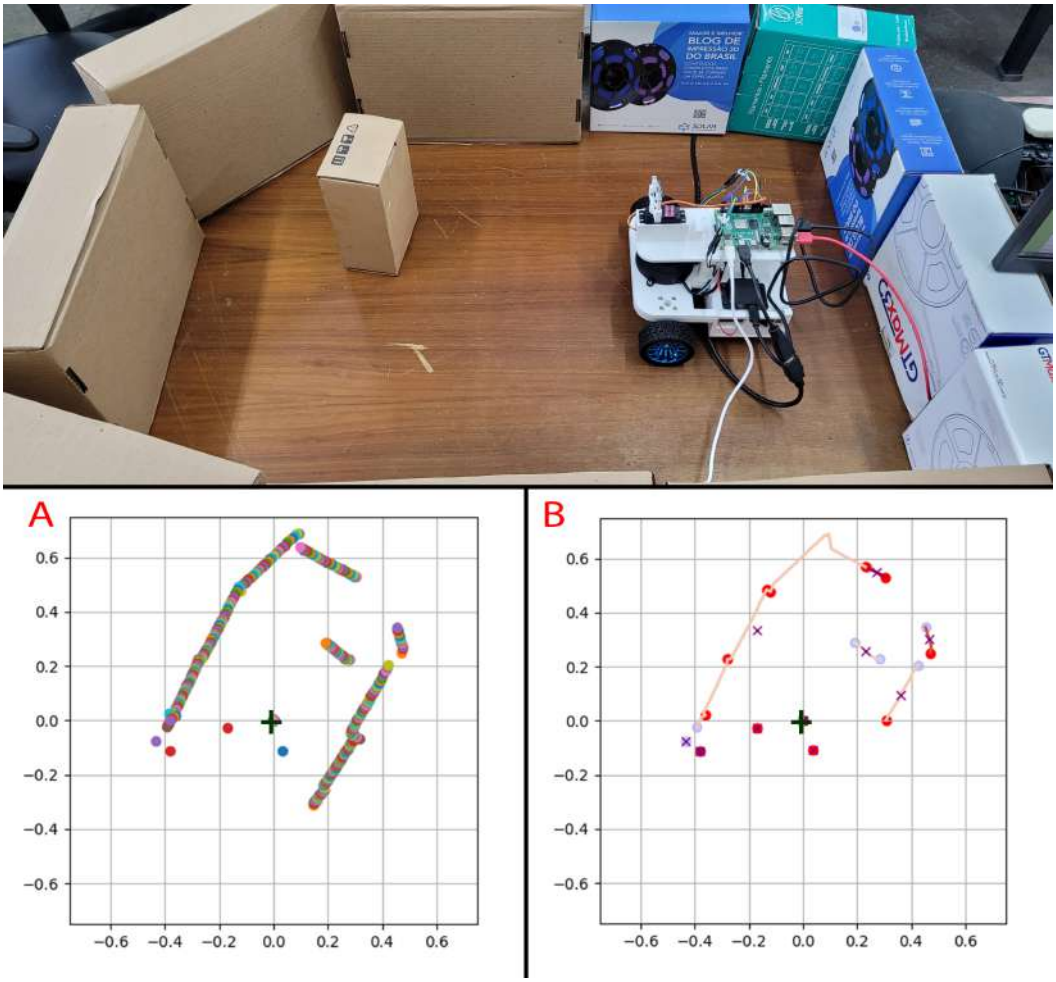
Figura 20 – Cenário 2.




Fonte: Próprio autor.

O cenário 3, mostrado na Figura 21, é semelhante ao primeiro cenário apresentado, exceto pela rotação da caixa que interpõe o sensor e as paredes. Este obstáculo estático causa uma obstrução na linha de visão do sensor, o que confere uma maior segmentação dos objetos em cena, apontando 9 obstáculos estáticos distintos.

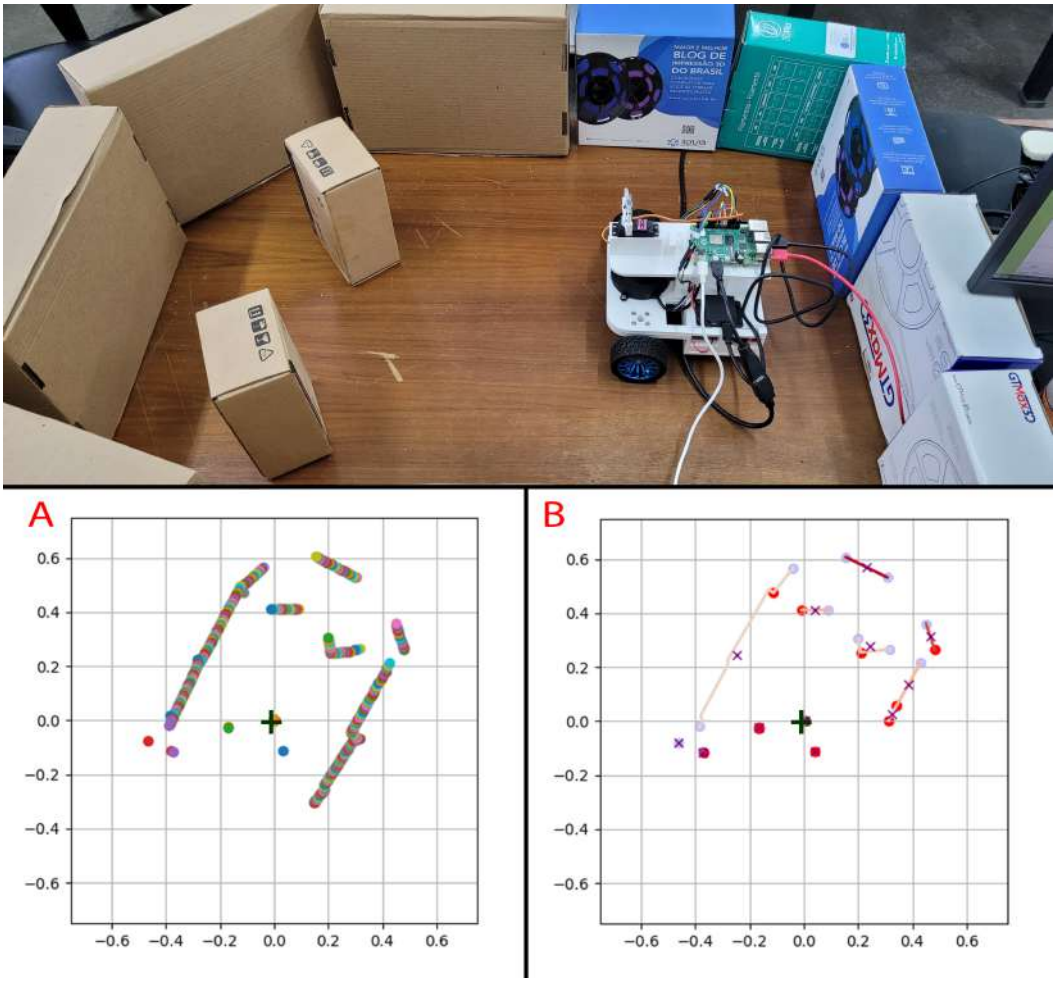
Figura 21 – Cenário 3.




Fonte: Próprio autor.

O cenário 4, mostrado na Figura 22, apresenta duas caixas dispostas diante do robô que se mostram suficientemente separadas para serem tomadas como entidades distintas no processo de segmentação proposto. Novamente, há uma obstrução na linha de visão do robô que causa a segmentação do obstáculo parcialmente ocultado e resulta na representação de 11 obstáculos estáticos.

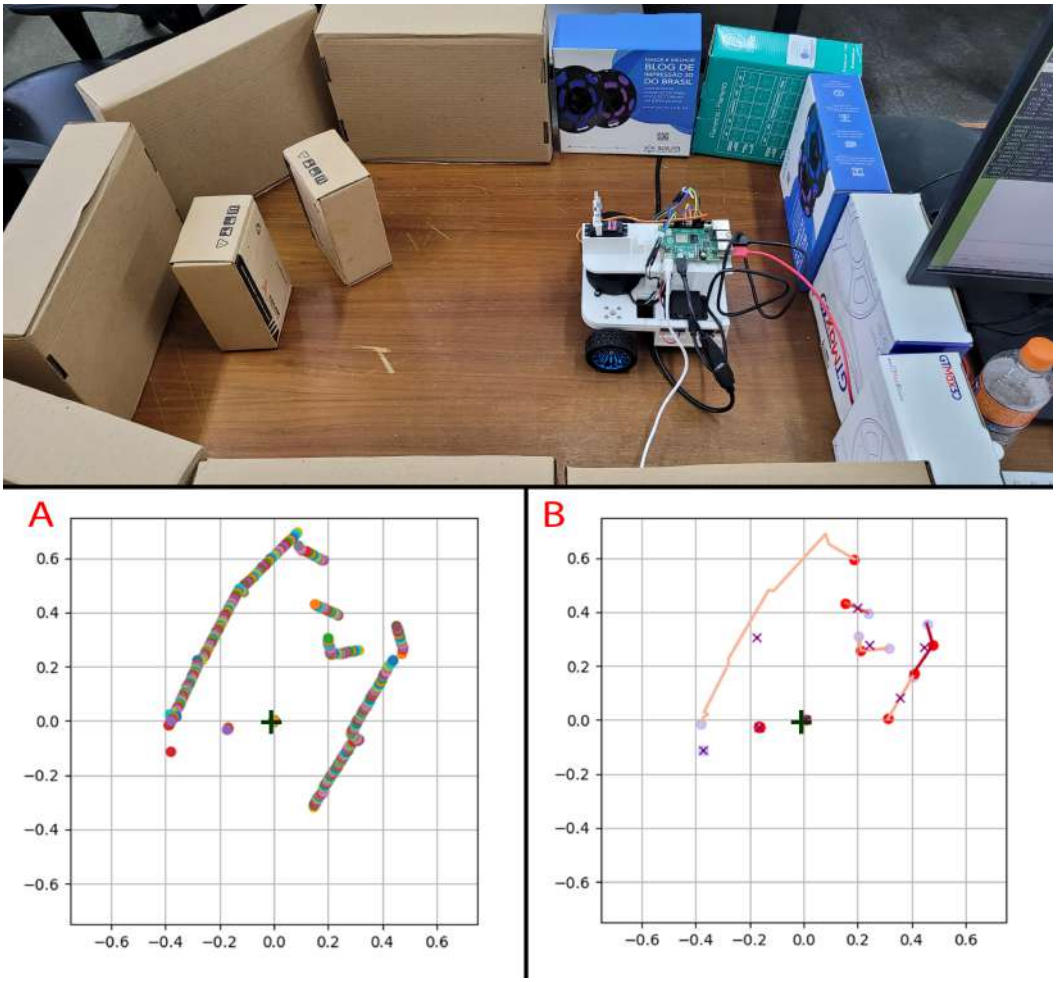
Figura 22 – Cenário 4.




Fonte: Próprio autor.

O cenário 5, mostrado na Figura 23, mantém as duas caixas do cenário anterior, com uma breve aproximação. A distância entre as caixas ainda não é curta o suficiente para que elas sejam consideradas o mesmo objeto após o processamento. Para este cenário, há 7 obstáculos estáticos identificados.

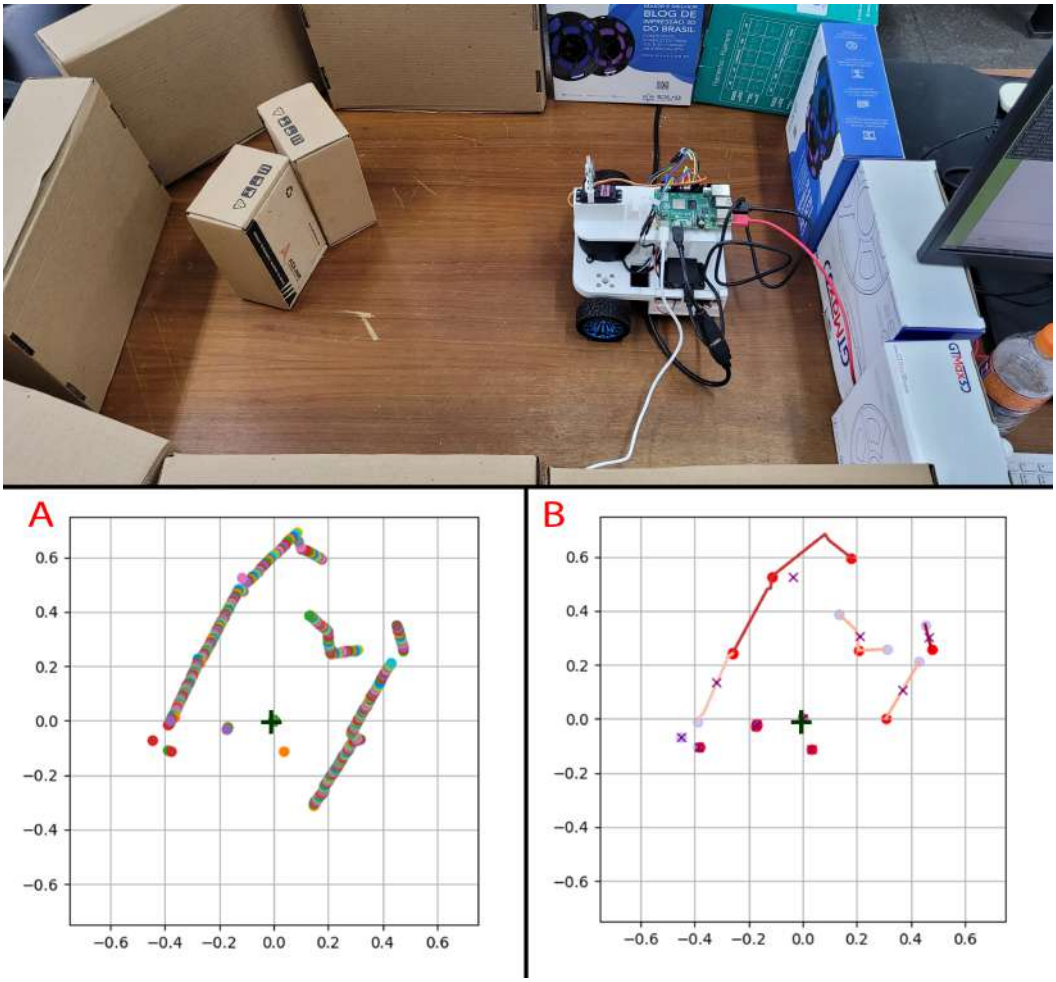
Figura 23 – Cenário 5.




Fonte: Próprio autor.

Por último, o cenário 6, mostrado na Figura 24, se diferencia do cenário 5 pela aproximação das caixas que se encontram diante do robô, de maneira que elas se encontram juntas o suficiente para que o processamento as classifique como um único objeto e atribua um ponto significativo à parte sobressalente da forma resultante. Para este cenário, há a determinação de 9 obstáculos estáticos diferentes.

Figura 24 – Cenário 6.




Fonte: Próprio autor.

Comum a todos os cenários avaliados está a obstrução da linha de visão do LIDAR pela montagem do sensor no robô, que impossibilita a aquisição de pontos provenientes da região traseira da unidade robótica. A solução proposta também apresenta, consistentemente, falhas na caracterização de obstáculos que se encontram à direita do robô.

## 5.2 Identificação de obstáculos dinâmicos

Para a avaliação do caráter dinâmico da proposta, são avaliados cenários de teste no ambiente apresentado na Figura 25, sendo o objeto utilizado como obstáculo dinâmico uma caixa de papelão de dimensões 14x10x9 cm, que foi manualmente movimentada diante do sensor para avaliar a detecção e a estimativa de velocidade.

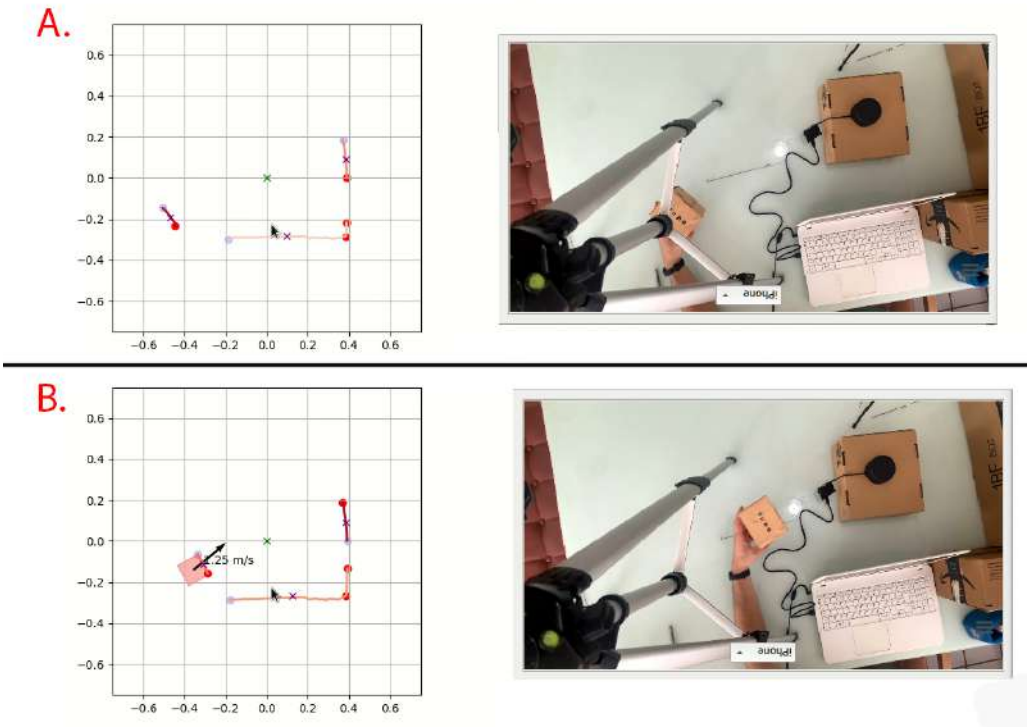
Os obstáculos têm da sua geometria com base na informação dos pontos constituintes, que permite ao programa sugerir um centro geométrico que é incluído na mensagem como parte das informações que caracterizam o obstáculo dinâmico.

Figura 25 – Ambiente de testes. A - Sensor Lidar, B - obstáculo móvel.


Fonte: Próprio autor.

As Figuras 26 e 27 demonstram visualmente a detecção e estimativa de velocidade, dispondo à direita, a representação gráfica da segmentação da nuvem de pontos, com a unidade de medida dos eixos em metros e, à esquerda, o registro de câmera do cenário de testes. O obstáculo móvel está demarcado na representação gráfica pelo retângulo vermelho e a seta indica a direção do movimento.

Figura 26 – Sequência A-B que demonstra movimento de aproximação do obstáculo móvel ao sensor.

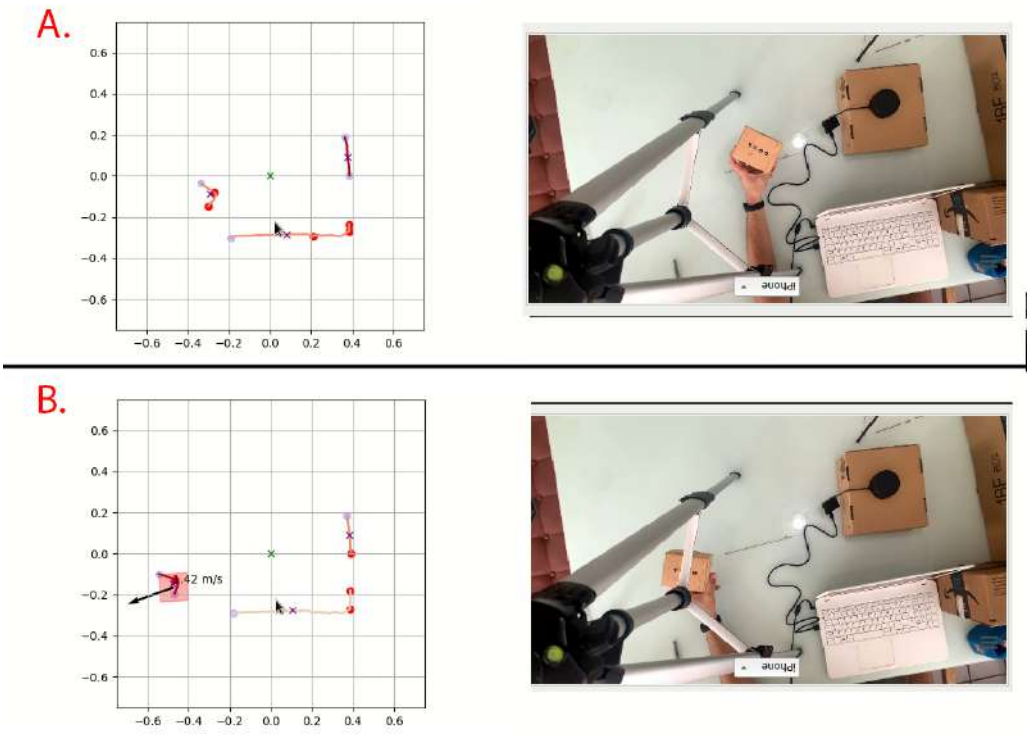



Fonte: Próprio autor.

O algoritmo proposto foi capaz de identificar obstáculos móveis para casos de deslocamento de baixa velocidade e sem sobreposição de obstáculos na linha de visão do sensor. Devido à ruído na aquisição de dados, há ainda uma instabilidade nos pontos adquiridos a cada varredura, que ocasionalmente apontam para um falso positivo na identificação de obstáculos móveis.

A ocorrência de falsos positivos pôde ser atenuada pela modificação de parâmetros do programa com o consequente comprometimento da sensibilidade, ou seja, quando mais robusto à ruidosidade dos dados, menos sensível na identificação do movimento de obstáculos, e vice-versa.

Figura 27 – Sequência A-B que demonstra movimento de afastamento do obstáculo móvel ao sensor.




Fonte: Próprio autor.

# 6 CONCLUSÕES

Neste trabalho foi apresentada uma proposta para a detecção e estimativa de velocidade para obstáculos dinâmicos em tempo real no contexto de navegação autônoma. O uso de um sensor Lidar no lugar de simulações computacionais aproximou o desenvolvimento da solução proposta do cenário real de uso, evidenciando limitações das técnicas utilizadas.

Como resultado, foi possível obter um programa modular capaz de identificar obstáculos dinâmicos e oferecer uma estimativa de direção de movimento e magnitude da velocidade para cenários controlados. A implementação em ROS permite que as informações geradas pelo programa sejam utilizadas em controladores de movimento como parte do processo decisório para a movimentação robótica.

Durante os testes, a instabilidade das medições, causada por fatores variados (vibração do sensor, iluminação, superfícies inconsistentes), fez necessária adoção de medidas que minimizassem o impacto sobre as medições sem ferir demasiadamente a sensibilidade da detecção. A abordagem escolhida ainda se demonstrou sensível a movimentação brusca dos obstáculos e do sensor.

Outro aspecto do trabalho que exigiu flexibilidade diz respeito ao sensor escolhido. Para sensores com resoluções de ângulo mais finas, a caracterização dos obstáculos se tornaria mais direta, assim como a aplicação dos algoritmos de ICP e de pareamento dos pontos de interesse.

# REFERÊNCIAS BIBLIOGRÁFICAS


BRITO, P. F. M. P. *Implementação em ROS de uma estratégia de navegação reativa para um robô móvel em um ambiente com obstáculos estáticos.* 2022. Trabalho de conclusão de curso (Monografia) - Universidade Federal de Pernambuco, Recife, 2022.

CHEEIN, F. A. A.; LOPEZ, N.; NATALIA; SORIA, C. M.; SCIASCIO, F. A. di; PEREIRA, F. L.; CARELLI, R. Slam algorithm applied to robotics assistance for navigation in unknown environments. *Journal of NeuroEngineering and Rehabilitation*, v. 7, 2010. Disponível em: <https://doi.org/10.1186/1743-0003-7-10>.

DEKAN, M.; FRANTIšEK, D.; ANDREJ, B.; JOZEF, R.; DáVID, R.; JOSIP, M. Moving obstacles detection based on laser range finder measurements. *International Journal of Advanced Robotic Systems*, v. 15, n. 1, p. 1729881417748132, 2018. Disponível em: <https://doi.org/10.1177/1729881417748132>.

DURAND-PETITEVILLE, A.; CADENAT, V. Advanced visual predictive control scheme for the navigation problem. In: . [S.l.: s.n.], 2021.

DURAND-PETITEVILLE, A.; CADENAT, V.; OUADAH, N. A complete sensor-based system to navigate through a cluttered environment. In: *2015 12th International Conference on Informatics in Control, Automation and Robotics (ICINCO)*. [S.l.: s.n.], 2015. v. 02, p. 166–173.

Embarcados. *Uma introdução ao Robot Operating System (ROS)*. 2022. Disponível em: <https://www.embarcados.com.br/uma-introducao-ao-robot-operating-system-ros/> . Acesso em: 16 maio 2022.

FUKUSHIMA, N. Icp with depth compensation for calibration of multiple tof sensors. In: *2018 - 3DTV-Conference: The True Vision - Capture, Transmission and Display of 3D Video (3DTV-CON)*. [S.l.: s.n.], 2018. p. 1–4.

MARINHOCESAR. *marinhocesar/moving_obstacles_detection: Moving obstacle detection*. Zenodo, 2023. Disponível em: <https://doi.org/10.5281/zenodo.8274870>.

MENDES, E.; KOCH, P.; LACROIX, S. Icp-based pose-graph slam. In: *2016 IEEE International Symposium on Safety, Security, and Rescue Robotics (SSRR)*. [S.l.: s.n.], 2016. p. 195–200.

POMERLEAU, F.; COLAS, F.; SIEGWART, R. *A Review of Point Cloud Registration Algorithms for Mobile Robotics*. [S.l.: s.n.], 2015.

PYTHON. *About Python*. Python Software Foundation, 2023. Disponível em: <https://www.python.org/about/>.

ROS 2 Documentation: Foxy. *Understanding nodes*. 2023. Disponível em: <https://docs.ros.org/en/foxy/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Nodes/Understanding-ROS2-Nodes.html> . Acesso em: 18 setembro 2023.

Shenzhen EAI Technology Co.,Ltd. *G2 DATA SHEET*. [S.l.], 2019.

SICILIANO, B.; KHATIB, O. (Ed.). *Springer Handbook of Robotics*. Berlin, Heidelberg: Springer, 2008. ISBN 978-3-540-23957-4. Disponível em: <http://dx.doi.org/10.1007/978-3-540-30301-5>.

SILVA, C. V. S. da. *Implementação de um controlador seguidor de espiral aplicado à um robô diferencial para evitar obstáculos estáticos e dinâmicos.* 2022. Trabalho de conclusão de curso (Monografia) - Universidade Federal de Pernambuco, Recife, 2022.

SINGH, Y.; SHARMA, S.; SUTTON, R.; HATTON, D.; KHAN, A. Feasibility study of a constrained dijkstra approach for optimal path planning of an unmanned surface vehicle in a dynamic maritime environment. In: *2018 IEEE International Conference on Autonomous Robot Systems and Competitions (ICARSC)*. [S.l.: s.n.], 2018. p. 117–122.

VAQUERO, V.; REPISO, E.; SANFELIU, A. Robust and real-time detection and tracking of moving objects with minimum 2d lidar information to advance autonomous cargo handling in ports. *Sensors*, v. 19, n. 1, 2019. ISSN 1424-8220. Disponível em: <https://www.mdpi.com/1424-8220/19/1/107>.

# Apêndices

# APÊNDICE A – CÓDIGO

A implementação está disponível no repositório remoto hospedado na plataforma GitHub, no perfil marinhocesar (2023).

```python
# src/plot_split_and_merge.py

import time
from typing import List
from matplotlib.axes import Axes
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cm as cm
import rospy
from sensor_msgs.msg import LaserScan
from moving_obstacles_detection.msg import MovingObstacles
from aux import aux_functions as aux
from aux.icp import icp


from aux.iterative_closest_point import (
    get_avg_displacement_for_significant_points,
    get_displacement_for_significant_points,
)
from components.buffer import SignificantBuffer
from components.point import Point
from components.object import Object
from aux.global_config import GlobalConfig
from aux import split_and_merge, point_extractor
import warnings

warnings.simplefilter("ignore")


class Publisher:
    def __init__(self) -> None:
        self.pub = rospy.Publisher("/obstacles", MovingObstacles, queue_size=1)


class SimpleSubscriber:
    def __init__(self):
        self.sub = rospy.Subscriber("/scan", LaserScan, self.callback, queue_size=1)
        self.points = list()

```

```
40      def callback(self, msg: LaserScan):
41          ranges = msg.ranges
42          if len(ranges) == 0:
43              return
44          self.points = [
45              Point(range=ranges[i], angle=msg.angle_min + msg.
    angle_increment * i)
46              for i in range(0, len(ranges))
47          ]
48
49      def printMsg(self):
50          print(self.points)
51
52      def get_points(self) -> List[Point]:
53          return self.points
54
55
56  def animate(
57      scan: List[Point],
58      significant_buffer: SignificantBuffer,
59      lidar: Axes,
60      publisher: Publisher,
61  ):
62      if not scan:
63          return
64
65      lidar.clear()
66      lidar.autoscale(False)
67      plt.grid(True)
68      ax = plt.gca()
69      ax.set_xlim([-GlobalConfig.MAX_RANGE, GlobalConfig.MAX_RANGE])
70      ax.set_ylim([-GlobalConfig.MAX_RANGE, GlobalConfig.MAX_RANGE])
71      plt.gca().set_aspect("equal")
72      cmap = cm.get_cmap(
73          "coolwarm"
74      )  # to see how many points in each segment, ordered by distance to
    sensor:
75
76      # configure data for visualization
77      structured_data = [Point(range=point.range, angle=point.angle) for
    point in scan]
78
79      # Performing ICP before segmentation
80
81      if significant_buffer.last_record > 1:
82          reference_points = aux.get_numpy_array_from_points(
83              points=significant_buffer.get_lidar()
```

```
84          )
85      else:
86          reference_points = aux.get_numpy_array_from_points(points=
    structured_data)
87
88      points = aux.get_numpy_array_from_points(points=structured_data)
89      history, structured_data = icp(reference_points=reference_points,
    points=points)
90      structured_data = aux.get_points_from_numpy_array(numpy_array=
    structured_data)
91
92      segments = split_and_merge.segment_lidar_data(scan_points=
    structured_data)
93      joined_segments = split_and_merge.
    join_segments_that_have_close_boundaries(
94          segments=segments
95      )
96
97      if GlobalConfig.DEBUG:
98          zero_angle_line = [Point(range=a / 100, angle=0) for a in range
    (1, 100)]
99          forty_five_line = [Point(range=a / 100, angle=3.14 / 4) for a in
     range(1, 100)]
100         line = Object(points=zero_angle_line + forty_five_line)
101         segments.append(line)
102
103     distances = [seg.distance_to_center for seg in joined_segments]
104     max_d = GlobalConfig.MAX_RANGE
105     if distances:
106         max_d = max(distances)
107
108     point_extractor.get_real_significant_points(
109         segments=joined_segments, all_points=structured_data
110     )
111
112     # plots the position of the sensor
113     lidar.plot(0, 0, "x", color="green")
114
115     all_real_significant: List[Point] = list()
116     all_apparent_significant: List[Point] = list()
117
118     for segment in joined_segments:
119         # plot the basic points that form each obstacle
120         lidar.plot(
121             segment.x_pos,
122             segment.y_pos,
123             "-",
```

```
124             linewidth=2,
125             color=cmap(segment.distance_to_center / max_d),
126         )
127 
128         if segment.y_line and GlobalConfig.SHOULD_RENDER_FIT_CURVE:
129             # renders the curve that best fits the point distribution
    for a particular obstacle
130             lidar.plot(
131                 segment.x_pos,
132                 segment.y_line,
133                 "-",
134                 linewidth=1,
135                 color="red",
136             )
137 
138         if segment.real_significant_points:
139             real_significant = Object(points=segment.
    real_significant_points)
140             all_real_significant += real_significant.points
141             lidar.scatter(real_significant.x_pos, real_significant.y_pos
    , color="red")  # type: ignore
142 
143         if segment.apparent_significant_points:
144             apparent_significant = Object(points=segment.
    apparent_significant_points)
145             all_apparent_significant += apparent_significant.points
146             lidar.scatter(
147                 apparent_significant.x_pos,
148                 apparent_significant.y_pos,
149                 color="blue",
150                 alpha=0.2,
151             )
152 
153         # Plot center of object
154         center_x, center_y = segment.center.get_rectangular()
155         lidar.plot(center_x, center_y, "x", color="purple")
156 
157     significant_buffer.register_significant(
158         real=all_real_significant,
159         apparent=all_apparent_significant,
160         lidar=structured_data,
161     )
162 
163     # get displacements
164     (
165         displacement_real_significant,
166         displacement_apparent_significant,
```

```
    ) = get_displacement_for_significant_points(point_cloud_history=
significant_buffer)
    # register displacements to buffer
    significant_buffer.register_displacement(
        real=displacement_real_significant,
        apparent=displacement_apparent_significant,
    )
    # get average displacements per buffer history
    (
        avg_displ_real,
        avg_displ_apparent,
    ) = get_avg_displacement_for_significant_points(
        point_cloud_history=significant_buffer
    )

    # Registering of displacements per Object
    for segment in joined_segments:
        real_sig = segment.real_significant_points
        apparent_sig = segment.apparent_significant_points
        real_displacements = dict()
        apparent_displacements = dict()

        for point in real_sig:
            displacement: Point = avg_displ_real.get(point, Point(0, 0))
            real_displacements[point] = displacement

        for point in apparent_sig:
            displacement: Point = avg_displ_apparent.get(point, Point(0,
    0))
            apparent_displacements[point] = displacement

        segment.register_displacement_real_significant_points(
real_displacements)
        segment.register_displacement_apparent_significant_points(
            apparent_displacements
        )
        segment.set_average_displacement()

    center_x = list()
    center_y = list()
    velocity_x = list()
    velocity_y = list()
    speed = list()
    radius = list()
    # display of vector
    time_step = significant_buffer.get_time_between()
    for segment in joined_segments:
```

```
211         shape = segment.get_shape()
212         if segment.avg_displacement is None:
213             continue
214         if segment.avg_displacement == Point(0, 0):
215             continue
216         disp = segment.avg_displacement
217 
218         center_speed = segment.avg_displacement.range / time_step
219         if center_speed < GlobalConfig.SPEED_THRESHOLD:
220             continue
221 
222         shape_patch = shape.get_patch(segment=segment)
223 
224         if shape_patch:
225             ax.add_patch(shape_patch)
226 
227         shape_center = aux.get_numpy_array_from_points([segment.center])
228         if shape_patch is not None:
229             shape_center = shape_patch.get_center()
230 
231         lidar.quiver(
232             *np.array([shape_center[0], shape_center[1]]),
233             np.array(disp.x),
234             np.array(disp.y),
235             color="black",
236             scale=0.9 / GlobalConfig.VECTOR_SCALING,
237         )
238 
239         lidar.annotate(
240             str(round(center_speed, 2)) + " m/s",
241             xy=(segment.center.x, segment.center.y),
242         )
243 
244         center_x.append(shape_center[0])
245         center_y.append(shape_center[1])
246         velocity_x.append(disp.x / time_step)
247         velocity_y.append(disp.y / time_step)
248         speed.append(center_speed)
249         radius.append(segment.get_length() / 2)
250 
251     msg = MovingObstacles()
252     msg.number_of_obstacles = len(center_x)
253     msg.center_x = center_x
254     msg.center_y = center_y
255     msg.velocity_x = velocity_x
256     msg.velocity_y = velocity_y
257     msg.speed = speed
```

```
    msg.radius = radius

    time_step = (
        GlobalConfig.DIFFERENCE_FOR_DISPLACEMENT
        * GlobalConfig.ANIMATION_INTERVAL
        / 1000
    )
    publisher.pub.publish(msg)
    et = time.time()
    frame_time = et - st
    if GlobalConfig.DEBUG is True:
        print("total", frame_time, "s")


# Main
if __name__ == "__main__":
    # configure plot
    fig = plt.figure()
    lidar = plt.subplot()
    ax = plt.gca()

    rospy.init_node("simpleSubOOP")
    subObj = SimpleSubscriber()
    publisher = Publisher()

    significant_buffer = SignificantBuffer(size=GlobalConfig.BUFFER_SIZE
    )

    while not rospy.is_shutdown():
        st = time.time()
        animate(
            scan=subObj.points,
            significant_buffer=significant_buffer,
            lidar=lidar,
            publisher=publisher,
        )
        plt.pause(GlobalConfig.ANIMATION_INTERVAL / 1000)
```

```
# src/components/point.py

import math

from typing import Tuple


class Point:
    def __init__(self, range: float, angle: float):
```

```
        self.range = range
        # self.angle = angle
        self.angle = angle if angle >= 0 else (2 * math.pi + angle)
        self.x, self.y = self.get_rectangular()

    def get_rectangular(self) -> Tuple[float, float]:
        x = self.range * math.cos(self.angle)
        y = self.range * math.sin(self.angle)
        return x, y

    def normalize(self) -> "Point":
        mag = self.range
        if mag == 0:
            return self

        return Point.init_from_rectangular(x=self.x / mag, y=self.y /
mag)

    def scale(self, factor: float) -> "Point":
        if self.range == 0:
            return self
        return Point.init_from_rectangular(x=self.x * abs(factor), y=
self.y * abs(factor))

    def get_projection(self):
        return self.range*math.cos(self.angle)

    @classmethod
    def init_from_rectangular(cls, x: float, y: float):
        radius = math.sqrt(x**2 + y**2)
        angle = math.atan2(y, x)
        return cls(range=radius, angle=angle)

    def __eq__(self, __value: "Point") -> bool:
        return self.range == __value.range and self.angle == __value.
angle

    def __hash__(self) -> int:
        return hash(("range", self.range, "angle", self.angle))

    def __repr__(self) -> str:
        x, y = self.get_rectangular()
        return f"(\n\trange:{self.range},\n\tangle:{self.angle*180/math.
pi},\n\tx:{x},\n\ty:{y}\n)"
```

```
# src/components/object.py

```

```python
import math
from scipy import optimize
from typing import List, Optional

from components.shape import Shape
from .point import Point
from aux import aux_functions as aux
from aux.global_config import GlobalConfig


class Object:
    def __init__(self, points: List[Point]):
        self.points = points
        self.n: int = len(points)
        self.x_pos: List[float] = [
            point.range * math.cos(point.angle) for point in points
        ]
        self.y_pos: List[float] = [
            point.range * math.sin(point.angle) for point in points
        ]
        self.avg_x: float = sum(self.x_pos) / self.n
        self.avg_y: float = sum(self.y_pos) / self.n
        self.std_dx: float = math.sqrt(
            sum([(xi - self.avg_x) ** 2 for xi in self.x_pos]) / self.n
        )
        self.std_dy: float = math.sqrt(
            sum([(yi - self.avg_y) ** 2 for yi in self.y_pos]) / self.n
        )
        self.a = 1.5 * self.std_dx
        self.b = 1.5 * self.std_dy
        self.distance_to_center = math.sqrt(self.avg_x**2 + self.avg_y
    **2)
        self.left_boundary: Point = self.get_boundary(left=True)
        self.right_boundary: Point = self.get_boundary()
        self.real_significant_points: List[Point] = list()
        self.apparent_significant_points: List[Point] = list()
        self.distances: List[float] = self.get_distances()
        self.min_length: float = 0 if len(self.distances) == 0 else min(
    self.distances)
        self.significant_inside: List[Point] = self.
    get_inside_significant()
        self.center: Point = Point(
            range=self.distance_to_center, angle=math.atan2(self.avg_y,
    self.avg_x)
        )
        self.a_coeff, self.b_coeff, self.c_coeff = None, None, None
        self.y_line = list()
```

```
46          if self.n > 1:
47              params, extra = optimize.curve_fit(
48                  f=aux.linear_function,
49                  xdata=self.x_pos,
50                  ydata=self.y_pos,
51              )
52              self.a_coeff, self.c_coeff = params
53              self.b_coeff = -1
54              self.y_line = [
55                  aux.linear_function(x=x, a=self.a_coeff, b=self.c_coeff)
56                  for x in self.x_pos
57              ]
58
59          self.distance_to_fitting = [
60              self.get_distance_to_fitting_line(point=point) for point in
    self.points
61          ]
62          self.real_significant_displacements: "dict[Point, Point]" = dict
    ()
63          self.apparent_significant_displacements: "dict[Point, Point]" =
    dict()
64          self.avg_displacement: Optional[Point] = None
65          self.velocity_x: List[float] = list()
66          self.velocity_y: List[float] = list()
67          self.is_eccentric = False
68
69      def register_displacement_real_significant_points(
70          self, real_displacements: "dict[Point, Point]"
71      ) -> None:
72          self.real_significant_displacements = real_displacements
73
74      def register_displacement_apparent_significant_points(
75          self, apparent_displacements: "dict[Point, Point]"
76      ) -> None:
77          self.apparent_significant_displacements = apparent_displacements
78
79      def set_velocity_vector_for_display(self) -> None:
80          if self.avg_displacement is None:
81              print("No average displacement has been set.")
82              return
83          if self.avg_displacement.range == 0:
84              self.velocity_x = []
85              self.velocity_y = []
86              return
87
88          normalized_displacement = self.avg_displacement.normalize()
89          scaled_displacement = normalized_displacement.scale(
```

```
90              0.05 * GlobalConfig.VECTOR_SCALING
91          )
92
93          velocity_vector = Point.init_from_rectangular(
94              x=self.center.x + scaled_displacement.x,
95              y=self.center.y + scaled_displacement.y,
96          )
97
98          x = [self.center.x, velocity_vector.x]
99          y = [self.center.y, velocity_vector.y]
100         params, extra = optimize.curve_fit(f=aux.linear_function, xdata=
    x, ydata=y)
101
102         a_coeff, b_coeff = params
103
104         self.velocity_x = x
105         self.velocity_y = [
106             aux.linear_function(x=x, a=a_coeff, b=b_coeff) for x in self
    .velocity_x
107         ]
108
109     def set_average_displacement(self) -> None:
110         real_displacements = self.real_significant_displacements
111         apparent_displacements = self.apparent_significant_displacements
112         n = len(real_displacements) + len(apparent_displacements)
113
114         if n == 0:
115             return
116
117         x_coord = (
118             sum(
119                 [
120                     real_displacements[displacement].x
121                     for displacement in real_displacements
122                     if real_displacements[displacement]
123                 ]
124                 + [
125                     apparent_displacements[displacement].x
126                     for displacement in apparent_displacements
127                 ]
128             )
129             / n
130         )
131         y_coord = (
132             sum(
133                 [
134                     real_displacements[displacement].y
```

```
                    for displacement in real_displacements
                ]
                + [
                    apparent_displacements[displacement].y
                    for displacement in apparent_displacements
                ]
            )
            / n
        )

        self.avg_displacement = Point.init_from_rectangular(x=x_coord, y=y_coord)

    def get_distance_to_fitting_line(self, point: Point) -> float:
        if self.a_coeff is None or self.b_coeff is None or self.c_coeff is None:
            return 0

        x = point.range * math.cos(point.angle)
        y = point.range * math.sin(point.angle)
        d = abs(self.a_coeff * x + self.b_coeff * y + self.c_coeff) / math.sqrt(
            self.a_coeff**2 + self.b_coeff**2
        )
        return d

    def get_distance_to_predicted_point(self, point) -> float:
        if self.a_coeff is None or self.b_coeff is None or self.c_coeff is None:
            return 0

        theta = point.angle
        predicted_point_x = -(self.c_coeff * math.cos(theta)) / (
            self.a_coeff * math.cos(theta) + self.b_coeff * math.sin(theta)
        )
        predicted_point_y = -(self.c_coeff * math.sin(theta)) / (
            self.a_coeff * math.cos(theta) + self.b_coeff * math.sin(theta)
        )

        predicted_point = Point(
            range=math.sqrt(predicted_point_x**2 + predicted_point_y**2),
            angle=math.atan2(predicted_point_y, predicted_point_x),
        )

```

```
175         return aux.get_distance_between_data_points(point,
    predicted_point)
176
177     def get_avg_significant(self):
178         n = len(self.significant_inside)
179         if n < 2:
180             return self.significant_inside
181
182         significant_cartesian = [
183             (point.range * math.cos(point.angle), point.range * math.sin
    (point.angle))
184             for point in self.significant_inside
185         ]
186
187         avg_x = sum([point[0] for point in significant_cartesian]) / n
188         avg_y = sum([point[1] for point in significant_cartesian]) / n
189
190         avg_r = math.sqrt(avg_x**2 + avg_y**2)
191         avg_angle = math.atan2(avg_y, avg_x)
192
193         return [Point(range=avg_r, angle=avg_angle)]
194
195     def register_real_significant_points(
196         self, real_significant_points: List[Point]
197     ) -> None:
198         points_dict = dict.fromkeys(real_significant_points, "")
199         for point in points_dict.keys():
200             self.real_significant_points.append(point)
201
202     def register_apparent_significant_points(
203         self, apparent_significant_points: List[Point]
204     ) -> None:
205         points_dict = dict.fromkeys(apparent_significant_points, "")
206         for point in points_dict.keys():
207             self.apparent_significant_points.append(point)
208
209     def get_boundary(self, left=False):
210         if self.points[0].angle > self.points[-1].angle:
211             if left:
212                 return self.points[0]
213             return self.points[-1]
214         else:
215             if left:
216                 return self.points[-1]
217             return self.points[0]
218
219     def get_inside_significant(self):
```

```
220         significant = []
221         for idx, point in enumerate(self.points):
222             if idx == 0 or idx == self.n - 1:
223                 continue
224             if self.is_internal_significant(i=idx, k=0):
225                 significant.append(point)
226 
227         if len(significant) == 0:
228             significant = self.get_eccentric_significant()
229 
230         return significant
231 
232     def get_furthermost_from_border(self, points):
233         curve_length = 0
234         for i in range(len(points) - 1):
235             curve_length += aux.get_distance_between_data_points(
236                 points[i], points[i + 1]
237             )
238 
239         if curve_length == 0:
240             return
241 
242         distances = []
243         for point in points:
244             start_distance = aux.get_distance_between_data_points(points
    [0], point)
245             end_distance = aux.get_distance_between_data_points(points
    [-1], point)
246 
247             border_distance = min(start_distance, end_distance)
248 
249             normalized_distance = border_distance / curve_length
250 
251             distances.append(normalized_distance)
252 
253         furthest_index = distances.index(max(distances))
254 
255         return points[furthest_index]
256 
257     def get_eccentric_significant(self):
258         significant_points = []
259         if len(self.points) == 1:
260             return significant_points
261 
262         point = self.get_furthermost_from_border(self.points)
263         if point is None:
264             return significant_points
```

```
265
266         left_b = self.get_boundary(left=True)
267         right_b = self.get_boundary(left=False)
268
269         length = aux.get_distance_between_data_points(left_b, right_b)
270
271         distance_to_line = aux.get_distance_to_line(segment=self, point=
    point)
272         threshold = GlobalConfig.ECCENTRICITY_THRESHOLD_COEFFICIENT *
    length
273
274         if distance_to_line > threshold:
275             significant_points.append(point)
276             self.is_eccentric = True
277
278         return significant_points
279
280     def is_boundary_close(self, other_segment):
281         # checks is the boundaries of a segment are next to the
    boundaries of other segment
282
283         xb = [other_segment.x_pos[0], other_segment.x_pos[-1]]
284         yb = [other_segment.y_pos[0], other_segment.y_pos[-1]]
285
286         if self.a == 0 or self.b == 0:
287             return False
288
289         for i in range(2):
290             first_b = ((xb[i] - self.x_pos[0]) / self.a) ** 2 + (
291                 (yb[i] - self.y_pos[0]) / self.b
292             ) ** 2 < 1
293             second_b = ((xb[i] - self.x_pos[-1]) / self.a) ** 2 + (
294                 (yb[i] - self.y_pos[-1]) / self.b
295             ) ** 2 < 1
296             if first_b or second_b:
297                 return True
298
299         return False
300
301     def is_internal_significant(self, i, k):
302         j = i + 1 + k
303         points = self.points
304
305         if j > self.n - 1:
306             return False
307
308         point_i = points[i]
```

```
309         point_i_minus_one = points[i - 1]
310         point_j = points[j]
311 
312         angle = self.get_angles_between_line_segments(i=i, j=j)
313         angle = angle * (180 / math.pi)
314 
315         if abs(point_i.range - point_i_minus_one.range) < self.
    min_length:
316             return False
317         if (
318             abs(point_i.range - point_i_minus_one.range) > self.
    min_length
319             and abs(point_i.range - point_j.range) > self.min_length
320             and (angle < GlobalConfig.MIN_ANGLE or angle > GlobalConfig.
    MAX_ANGLE)
321         ):
322             return False
323         if (
324             abs(point_i.range - point_i_minus_one.range) > self.
    min_length
325             and abs(point_i.range - point_j.range) < self.min_length
326         ):
327             return self.is_internal_significant(i=i, k=k + 1)
328         if (
329             abs(point_i.range - point_i_minus_one.range) > self.
    min_length
330             and abs(point_i.range - point_j.range) > self.min_length
331             and (angle >= GlobalConfig.MIN_ANGLE and angle <=
    GlobalConfig.MAX_ANGLE)
332         ):
333             return True
334 
335         return False
336 
337     def get_angles_between_line_segments(self, i, j):
338         point: Point = self.points[i]
339         prev: Point = self.points[i - 1]
340         after: Point = self.points[j]
341 
342         point_x, point_y = point.get_rectangular()
343         prev_x, prev_y = prev.get_rectangular()
344         after_x, after_y = after.get_rectangular()
345 
346         m1, m2 = 0, 0
347         if point_x - prev_x != 0:
348             m1 = (point_y - prev_y) / (point_x - prev_x)
349         if after_x - point_x != 0:
```

```
            m2 = (after_y - point_y) / (after_x - point_x)

        tan = (m2 - m1) / (1 + m1 * m2)
        angle_between = math.atan(tan)

        return angle_between

    def get_distances(self):
        distances = []
        for idx, point in enumerate(self.points):
            if idx == 0:
                continue

            prev_point = self.points[idx - 1]
            d = aux.get_distance_between_data_points(prev_point, point)
            distances.append(abs(d))

        return distances

    def get_length(self):
        return aux.get_distance_between_data_points(
            self.left_boundary, self.right_boundary
        )

    def get_shape(self):
        if len(self.significant_inside) == 0:
            point_angle = self.right_boundary
            angle = aux.get_angles_between_three_points(
                [Point(0, 0), self.center, point_angle]
            )
            if self.right_boundary.range < self.left_boundary.range:
                angle -= math.pi / 2
            else:
                angle += math.pi / 2

            return Shape(
                shape_type="rectangle",
                center=point_angle,
                significant_length=self.get_length(),
                angle=angle,
            )

        if self.is_eccentric is False and len(self.significant_inside)
    == 1:


            point_angle = self.right_boundary
```

```
396             angle = math.pi * (7 / 10) + aux.
    get_angles_between_three_points(
397                 [Point(0, 0), self.center, point_angle]
398             )
399 
400             return Shape(
401                 shape_type="rectangle",
402                 center=point_angle,
403                 significant_length=self.get_length(),
404                 angle=angle,
405             )
406 
407         if self.is_eccentric is True and len(self.significant_inside) ==
     1:
408             center_offset = (self.get_length() / 2) * (
409                 1 + GlobalConfig.BOUNDARY_SAFETY_COEFF
410             )
411             return Shape(
412                 shape_type="circle",
413                 center=Point(
414                     range=self.center.range + center_offset, angle=self.
    center.angle
415                 ),
416                 significant_length=self.get_length(),
417             )
418 
419         return Shape(
420             shape_type="circle",
421             center=self.center,
422             significant_length=self.get_length(),
423         )
```

```
1  # src/components/shape.py
2  from __future__ import annotations
3  import math
4  from typing import Optional, TYPE_CHECKING
5 
6  from matplotlib.patches import Circle, Ellipse, Rectangle
7  from aux.global_config import GlobalConfig
8  from components.point import Point
9 
10 if TYPE_CHECKING:
11     from components.object import Object
12 
13 
14 class Shape:
15     def __init__(
```

```
        self,
        shape_type: str,
        significant_length: float,
        center: Point,
        angle: Optional[float] = None,
        secondary_length: Optional[float] = None,
    ):
        self.shape_type = shape_type
        self.significant_length = (
            1 + GlobalConfig.BOUNDARY_SAFETY_COEFF
        ) * significant_length
        self.angle = angle
        self.center = center
        self.secondary_length = secondary_length

    def get_patch(self, segment: Object, color: str = "r", alpha: float = 0.3):
        if self.shape_type == "circle":
            return Circle(
                (self.center.x, self.center.y),
                self.significant_length / 2,
                color=color,
                alpha=alpha,
            )
        if self.shape_type == "rectangle":
            return Rectangle(
                xy=(self.center.x, self.center.y),
                width=self.significant_length,
                height=self.significant_length,
                angle=180 * (segment.center.angle + self.angle) / math.pi,  # type: ignore
                color=color,
                alpha=alpha,
            )
        if self.shape_type == "ellipse":
            if self.secondary_length is None:
                raise Exception("missing secondary length")
            return Ellipse(
                xy=(self.center.x, self.center.y),
                width=self.significant_length,
                height=self.secondary_length,
                angle=self.angle,  # type: ignore
                color=color,
                alpha=alpha,
            )
```

```
# src/components/buffer.py
```

```
2
3  from typing import List
4  from aux.global_config import GlobalConfig
5
6  from components.point import Point
7
8
9  class SignificantBuffer:
10     def __init__(self, size: int):
11         self.size: int = size
12         self.real_significant: List[List[Point]] = [list()] * self.size
13         self.apparent_significant: List[List[Point]] = [list()] * self.
       size
14         self.lidar: List[List[Point]] = [list()] * self.size
15         self.displacement_real: List[dict[Point, Point]] = [dict()] *
       self.size
16         self.displacement_apparent: List[dict[Point, Point]] = [dict()]
       * self.size
17         self.time: List[float] = [0] * self.size
18         self.last_record: int = -1
19
20     def register_displacement(
21         self, real: "dict[Point, Point]", apparent: "dict[Point, Point]"
22     ) -> None:
23         self.__register_real_displacement(real)
24         self.__register_apparent_displacement(apparent)
25
26     def __register_real_displacement(self, real: "dict[Point, Point]")
       -> None:
27         index = self.last_record % self.size
28         self.displacement_real[index] = real
29
30     def __register_apparent_displacement(self, apparent: "dict[Point,
       Point]") -> None:
31         index = self.last_record % self.size
32         self.displacement_apparent[index] = apparent
33
34     def register_significant(
35         self, real: List[Point], apparent: List[Point], lidar: List[
       Point]
36     ) -> None:
37         self.last_record += 1
38         index = self.last_record % self.size
39
40         self.time[index] = self.time[index-1] + 1 / GlobalConfig.
       LIDAR_FREQUENCY
41
```

```
42          self.__register_real_significant(points_to_save=real,
    record_position=index)
43          self.__register_apparent_significant(
44              points_to_save=apparent, record_position=index
45          )
46          self.__register_lidar(
47              points_to_save=lidar, record_position=index
48          )
49
50      def __remove_duplicate_points(self, points: List[Point]):
51          points_dict = {point for point in points}
52          return list(points_dict)
53
54      def __register_lidar(
55          self, points_to_save: List[Point], record_position: int
56      ) -> None:
57          points = self.__remove_duplicate_points(points_to_save)
58          self.lidar[record_position] = points
59
60      def __register_real_significant(
61          self, points_to_save: List[Point], record_position: int
62      ) -> None:
63          points = self.__remove_duplicate_points(points_to_save)
64          self.real_significant[record_position] = points
65
66      def __register_apparent_significant(
67          self, points_to_save: List[Point], record_position: int
68      ) -> None:
69          points = self.__remove_duplicate_points(points_to_save)
70          self.apparent_significant[record_position] = points
71
72      def get_time_between(self):
73          last_time_idx = self.last_record % self.size
74          last_time = self.time[last_time_idx]
75          if self.last_record < GlobalConfig.DIFFERENCE_FOR_DISPLACEMENT:
76              previous_idx = 0
77          else:
78              previous_idx = (self.last_record - GlobalConfig.
    DIFFERENCE_FOR_DISPLACEMENT + 1) % self.size
79          previous_time = self.time[previous_idx]
80
81          return last_time - previous_time
82
83      def get_lidar(self):
84          last_time_idx = self.last_record % self.size
85          lidar = self.lidar[last_time_idx]
86          return lidar
```

```python
# src/aux/global_config.py

class GlobalConfig:
    DEBUG = False
    SEGMENTATION_THRESHOLD = 0.1
    EXCLUSION_THRESHOLD = 25  # degrees
    MAX_SKIPPED = -1  # maximum skipped points
    MIN_ANGLE = 40
    MAX_ANGLE = 140
    ECCENTRICITY_THRESHOLD_COEFFICIENT = 1 / 5
    MIN_RANGE = -1
    MAX_RANGE = 0.75
    SHOULD_JOIN = True
    MAX_ALLOWED_MEASURE_DIFF = 0.1
    SHOULD_RENDER_FIT_CURVE = False

    # SEED_SEGMENT DETECTION
    P_MIN = 10  # minimum number of laser points contained in an extracted line segment
    S_NUM = 6  # number of points in a seed-segment
    DISTANCE_THRESHOLD_POINT_LINE = 0.03  # meters
    DISTANCE_THRESHOLD_POINT_POINT = 0.1  # meters

    # rounding factor used when mapping lidar data to a python dict using the truncated angle as a key0
    ROUNDING_FACTOR = 10**10

    # number of adjacent points to be checked while wanting to determine if a point is apparent or real significant
    NEIGHBORHOOD_CHECK_FOR_APPARENT_SIGNIFICANT = 10

    DIFFERENCE_FOR_DISPLACEMENT = 2
    BUFFER_SIZE = 10
    DISPLACEMENT_THRESHOLD = 0.05
    VARIANCE_THRESHOLD = 0.01
    ANIMATION_INTERVAL = 1  # Delay between frames in milliseconds.

    VECTOR_SCALING = 1

    MAX_MATCHING_DISTANCE = 0.25
    SPEED_THRESHOLD = 0  # m/s

    BOUNDARY_SAFETY_COEFF = 0.1
    LIDAR_FREQUENCY = 10  # Hz
```

```python
# src/aux/aux_functions.py

```

```
3  from decimal import Decimal
4  import math
5  from typing import List, Optional
6  import numpy as np
7
8  from components.object import Object
9  from components.point import Point
10
11
12 def linear_function(x, a, b):
13     y = a * x + b
14     return y
15
16
17 def get_distance_between_data_points(first_data_point: Point,
       second_data_point: Point):
18     return math.sqrt(
19         (second_data_point.x - first_data_point.x) ** 2
20         + (second_data_point.y - first_data_point.y) ** 2
21     )
22
23
24 def get_angle_between_data_points(first_data_point, second_data_point):
25     first_angle = first_data_point.angle
26     if first_angle < 0:
27         first_angle = 2 * math.pi + first_angle
28     second_angle = second_data_point.angle
29     if second_angle < 0:
30         second_angle = 2 * math.pi + second_angle
31     return abs(Decimal(first_angle) - Decimal(second_angle))
32
33
34 def get_distance_to_line(segment: Object, point: Point):
35     left = segment.get_boundary(left=True)
36     right = segment.get_boundary()
37
38     x2 = left.range * math.cos(left.angle)
39     y2 = left.range * math.sin(left.angle)
40     x1 = right.range * math.cos(right.angle)
41     y1 = right.range * math.sin(right.angle)
42     x0 = point.range * math.cos(point.angle)
43     y0 = point.range * math.sin(point.angle)
44
45     length = math.sqrt((x2 - x1) ** 2 + (y2 - y1) ** 2)
46
47     if length == 0:
48         return 0
```

```
49
50      d = abs((x2 - x1) * (y1 - y0) - (x1 - x0) * (y2 - y1)) / length
51
52      return d
53
54
55  def are_same_point(point1: Point, point2: Optional[Point]):
56      if isinstance(point1, Point) and isinstance(point2, Point):
57          return point1 == point2
58      return False
59
60
61  def get_numpy_array_from_points(points: List[Point]):
62      array = [(point.x, point.y) for point in points]
63      return np.array(array)
64
65
66  def get_points_from_numpy_array(numpy_array):
67      points = [
68          Point.init_from_rectangular(x=point[0], y=point[1]) for point in
        numpy_array
69      ]
70      points.sort(key=lambda x: x.angle)
71      return points
72
73
74  def get_angles_between_three_points(points: List[Point]):
75      assert len(points) == 3
76
77      point: Point = points[1]
78      prev: Point = points[0]
79      after: Point = points[2]
80
81      point_x, point_y = point.get_rectangular()
82      prev_x, prev_y = prev.get_rectangular()
83      after_x, after_y = after.get_rectangular()
84
85      m1, m2 = 0, 0
86      if point_x - prev_x != 0:
87          m1 = (point_y - prev_y) / (point_x - prev_x)
88      if after_x - point_x != 0:
89          m2 = (after_y - point_y) / (after_x - point_x)
90
91      tan = (m2 - m1) / (1 + m1 * m2)
92      angle_between = math.atan(tan)
93
94      return angle_between
```

```python
# src/aux/iterative_closest_point.py

import math
from typing import List, Tuple

# from aux.icp import icp

from aux.global_config import GlobalConfig
from components.buffer import SignificantBuffer
from aux import aux_functions as aux
from components.point import Point


def get_displacement_for_significant_points(
    point_cloud_history: SignificantBuffer,
) -> "Tuple[dict[Point, Point], dict[Point, Point]]":
    buffer_size = point_cloud_history.size
    current_record_n = point_cloud_history.last_record
    current_record_idx = current_record_n % buffer_size
    all_real_significant = point_cloud_history.real_significant
    all_apparent_significant = point_cloud_history.apparent_significant
    last_real_significant = all_real_significant[current_record_idx]
    last_apparent_significant = all_apparent_significant[
   current_record_idx]
    interval_size = GlobalConfig.DIFFERENCE_FOR_DISPLACEMENT

    if current_record_n < interval_size:
        n_records = current_record_n
    else:
        n_records = interval_size

    displacement_real_significant = {
        point: Point(0, 0) for point in last_real_significant
    }
    displacement_apparent_significant = {
        point: Point(0, 0) for point in last_apparent_significant
    }

    if current_record_n == 0:
        no_displacement_real = {
            point: Point(range=0, angle=0) for point in
   last_real_significant
        }
        no_displacement_apparent = {
            point: Point(range=0, angle=0) for point in
   last_apparent_significant
        }
```

```
45          return no_displacement_real, no_displacement_apparent
46
47      idx = (current_record_idx - n_records + 1) % buffer_size
48      compare_real_sig = all_real_significant[idx]
49      compare_apparent_sig = all_apparent_significant[idx]
50
51      if len(compare_real_sig) == 0 and len(compare_apparent_sig) == 0:
52          return displacement_real_significant,
    displacement_apparent_significant
53
54      # evaluating real significant points
55      for point in last_real_significant:
56          distances_real_sig = [
57              aux.get_distance_between_data_points(point, other_point)
58              for other_point in compare_real_sig
59          ]
60          if len(distances_real_sig) == 0:
61              distances_real_sig = [0.0]
62              compare_real_sig = [point]
63
64          distances_apparent_sig = [
65              aux.get_distance_between_data_points(point, other_point)
66              for other_point in compare_apparent_sig
67          ]
68          if len(distances_apparent_sig) == 0:
69              distances_apparent_sig = [0.0]
70              compare_apparent_sig = [point]
71
72          closest_real_point = compare_real_sig[
73              distances_real_sig.index(min(distances_real_sig))
74          ]
75          closest_apparent_point = compare_apparent_sig[
76              distances_apparent_sig.index(min(distances_apparent_sig))
77          ]
78
79          if min(distances_real_sig) < min(distances_apparent_sig):
80              displacement = get_displacement(
81                  first_point=point,
82                  second_point=closest_real_point,
83              )
84          else:
85              displacement = get_displacement(
86                  first_point=point,
87                  second_point=closest_apparent_point,
88              )
89
90          displacement_real_significant[point] = displacement
```

```
91
92      # evaluating apparent significant points
93      for point in last_apparent_significant:
94          distances_real_sig = [
95              aux.get_distance_between_data_points(point, other_point)
96              for other_point in compare_real_sig
97          ]
98          if len(distances_real_sig) == 0:
99              distances_real_sig = [0.0]
100             compare_real_sig = [point]
101
102         distances_apparent_sig = [
103             aux.get_distance_between_data_points(point, other_point)
104             for other_point in compare_apparent_sig
105         ]
106         if len(distances_apparent_sig) == 0:
107             distances_apparent_sig = [0.0]
108             compare_apparent_sig = [point]
109
110         closest_real_point = compare_real_sig[
111             distances_real_sig.index(min(distances_real_sig))
112         ]
113         closest_apparent_point = compare_apparent_sig[
114             distances_apparent_sig.index(min(distances_apparent_sig))
115         ]
116
117         if min(distances_real_sig) < min(distances_apparent_sig):
118             displacement = get_displacement(
119                 first_point=point,
120                 second_point=closest_real_point,
121             )
122         else:
123             displacement = get_displacement(
124                 first_point=point,
125                 second_point=closest_real_point,
126             )
127
128         displacement_apparent_significant[point] = displacement
129
130     return displacement_real_significant,
    displacement_apparent_significant
131
132
133 def get_avg_displacement_for_significant_points(
134     point_cloud_history: SignificantBuffer,
135 ) -> "Tuple[dict[Point, Point], dict[Point, Point]]":
136     buffer_size = point_cloud_history.size
```

```
137     current_record_n = point_cloud_history.last_record
138     current_record_idx = current_record_n % buffer_size
139     all_real_displacement = point_cloud_history.displacement_real
140     all_apparent_displacement = point_cloud_history.
    displacement_apparent
141     last_real_significant = all_real_displacement[current_record_idx]
142     last_apparent_significant = all_apparent_displacement[
    current_record_idx]
143     if current_record_n < buffer_size:
144         n_records = current_record_n
145     else:
146         n_records = buffer_size
147 
148     displ_real: dict[Point, List[Point]] = {
149         point: list() for point in last_real_significant
150     }
151     displ_apparent: dict[Point, List[Point]] = {
152         point: list() for point in last_apparent_significant
153     }
154 
155     if current_record_n < 1:
156         no_displacement_real = {
157             point: Point(range=0, angle=0) for point in
    last_real_significant
158         }
159         no_displacement_apparent = {
160             point: Point(range=0, angle=0) for point in
    last_apparent_significant
161         }
162         return no_displacement_real, no_displacement_apparent
163 
164     for i in range(n_records + 1):
165         idx = (current_record_idx - i) % buffer_size
166         compare_real_displ = all_real_displacement[idx]
167         compare_apparent_displ = all_apparent_displacement[idx]
168         compare = {**compare_real_displ, **compare_apparent_displ}
169 
170         if len(compare) == 0:
171             empty_displ = dict()
172             return empty_displ, empty_displ
173 
174         for point in displ_real.keys():
175             closest_point = get_closest_point(
176                 point=point,
177                 points_to_compare=list(compare),
178             )
179 
```

```
180             displacement = get_displacement(point, closest_point)
181
182             displ_real[point].append(displacement)
183
184         for point in displ_apparent.keys():
185             closest_point = get_closest_point(
186                 point=point,
187                 points_to_compare=list(compare),
188             )
189             displacement = get_displacement(point, closest_point)
190
191             displ_apparent[point].append(displacement)
192
193     avg_displ_real = {point: Point(0, 0) for point in displ_real}
194     avg_displ_apparent = {point: Point(0, 0) for point in displ_apparent
    }
195
196     for point in displ_real.keys():
197         avg_displ_real[point] = get_avg_displacement(displacements=
    displ_real[point])
198     for point in displ_apparent.keys():
199         avg_displ_apparent[point] = get_avg_displacement(
200             displacements=displ_apparent[point]
201         )
202
203     return avg_displ_real, avg_displ_apparent
204
205
206 def get_displacement(
207     first_point: Point,
208     second_point: Point,
209 ) -> Point:
210     d = aux.get_distance_between_data_points(
211         first_data_point=first_point, second_data_point=second_point
212     )
213     if d > GlobalConfig.MAX_MATCHING_DISTANCE:
214         return Point(0, 0)
215
216     x_coord = first_point.x - second_point.x
217     y_coord = first_point.y - second_point.y
218     return Point.init_from_rectangular(x=x_coord, y=y_coord)
219
220
221 def get_avg_displacement(displacements: List[Point]) -> Point:
222     no_displacement = Point(0, 0)
223     real_displacements = [
224         disp
```

```
225         for disp in displacements
226         if disp.range > GlobalConfig.DISPLACEMENT_THRESHOLD
227     ]
228     # real_displacements = [disp for disp in displacements]
229     n = len(real_displacements)
230     if n < 1:
231         return no_displacement
232 
233     x_pos = [point.x for point in real_displacements]
234     y_pos = [point.y for point in real_displacements]
235     avg_x = sum(x_pos) / n
236     avg_y = sum(y_pos) / n
237     v_x = math.sqrt(sum([(xi - avg_x) ** 2 for xi in x_pos]) / n)
238     v_y = math.sqrt(sum([(yi - avg_y) ** 2 for yi in y_pos]) / n)
239 
240     if v_x < GlobalConfig.VARIANCE_THRESHOLD and v_y < GlobalConfig.
    VARIANCE_THRESHOLD:
241         return no_displacement
242 
243     avg_disp = Point.init_from_rectangular(x=avg_x, y=avg_y)
244 
245     if avg_disp.range < GlobalConfig.DISPLACEMENT_THRESHOLD:
246         return no_displacement
247 
248     return avg_disp
249 
250 
251 def get_closest_point(point: Point, points_to_compare: List[Point]) ->
    Point:
252     distances_to_point = [
253         aux.get_distance_between_data_points(point, other_point)
254         for other_point in points_to_compare
255     ]
256     closest_point_index = distances_to_point.index(min(
    distances_to_point))
257     closest = points_to_compare[closest_point_index]
258     return closest
259 
260 
261 def get_second_closest_point(
262     point: Point, closest_index: int, points_to_compare: List[Point]
263 ) -> Point:
264     new_points = points_to_compare.copy()
265     new_points.pop(closest_index)
266 
267     distances_to_point = [
268         aux.get_distance_between_data_points(point, other_point)
```

```
269         for other_point in new_points
270     ]
271     second_closest_point_index = distances_to_point.index(min(
    distances_to_point))
272     second_closest = new_points[second_closest_point_index]
273     return second_closest
```

```
1  # src/aux/point_extractor.py
2
3  from typing import List
4
5  from components.object import Object
6  from components.point import Point
7  from aux.global_config import GlobalConfig
8
9
10 def get_real_significant_points(segments: List[Object], all_points: List
    [Point]):
11
12     max_range = GlobalConfig.NEIGHBORHOOD_CHECK_FOR_APPARENT_SIGNIFICANT
13
14     for segment in segments:
15         real_significant_points = []
16         apparent_significant_points = []
17         # check points to the left of the boundaries
18         left_outline_point = segment.get_boundary(left=True)
19
20         # find boundary in all_points
21         l_boundary_idx = 0
22         for idx, point in enumerate(all_points):
23             if point.angle == left_outline_point.angle:
24                 l_boundary_idx = idx
25                 break
26
27         h_left = [
28             all_points[(l_boundary_idx - j) % len(all_points)] for j in
    range(max_range)
29         ]
30
31         is_real_significant = False
32         for point in h_left:
33             if point.range > left_outline_point.range:
34                 is_real_significant = True
35                 break
36
37         if is_real_significant is True:
38             # if left_outline_point in real_significant_points:
```

```
            #     continue
            real_significant_points.append(left_outline_point)
        else:
            # if left_outline_point in apparent_significant_points:
            #     continue
            apparent_significant_points.append(left_outline_point)

        # check points to the right of the boundaries
        right_outline_point = segment.get_boundary(left=False)

        # find boundary in all_points
        r_boundary_idx = 0
        for idx, point in enumerate(all_points):
            if point.angle == right_outline_point.angle:
                r_boundary_idx = idx
                break

        h_right = [
            all_points[(r_boundary_idx + j) % len(all_points)] for j in
    range(max_range)
        ]

        is_real_significant = False
        for point in h_right:
            if point.range > right_outline_point.range:
                is_real_significant = True
                real_significant_points.append(right_outline_point)
                break

        if is_real_significant is True:
            # if right_outline_point in real_significant_points:
            #     continue
            real_significant_points.append(right_outline_point)
        else:
            # if right_outline_point in apparent_significant_points:
            #     continue
            apparent_significant_points.append(right_outline_point)

        for point in segment.significant_inside:
            if point not in real_significant_points:
                real_significant_points.append(point)

        segment.register_real_significant_points(
            real_significant_points=real_significant_points
        )
        segment.register_apparent_significant_points(
            apparent_significant_points=apparent_significant_points
```

```
        )
```

```python
# src/aux/split_and_merge.py

import math
from typing import List, Optional

from aux import aux_functions as aux
from aux.global_config import GlobalConfig
from components.object import Object
from components.point import Point


def get_angle_for_exclusion(first_data_point, second_data_point):
    l_1 = first_data_point.range
    l_2 = second_data_point.range
    angle_between = aux.get_angle_between_data_points(
        first_data_point=first_data_point, second_data_point=
    second_data_point
    )

    if angle_between == 0:
        #  limit when angle -> 0
        return (
            ((l_2 - l_1) / math.sqrt(l_1**2 + l_2**2 - l_1 * l_2)) * 180
    / math.pi
        )

    acos_argument = (l_2 - l_1 * math.cos(angle_between)) / (
        math.sqrt(l_1**2 + l_2**2 - 2 * l_1 * l_2 * math.cos(
    angle_between))
    )

    return math.acos(acos_argument) * 180 / math.pi  # returns in
    degrees


def should_be_excluded(current, last_in_object, last_evaluated):
    if last_in_object is None:
        return False

    beta_1 = get_angle_for_exclusion(
        first_data_point=last_in_object,
        second_data_point=last_evaluated,
    )
    beta_2 = get_angle_for_exclusion(
        first_data_point=last_evaluated,
```

```
42          second_data_point=current,
43      )
44
45      return abs(beta_1 - beta_2) < GlobalConfig.EXCLUSION_THRESHOLD
46
47
48  def join_segments_that_have_close_boundaries(segments: List[Object]) ->
        List[Object]:
49      if GlobalConfig.SHOULD_JOIN is False:
50          return segments
51
52      pairs = dict()
53      for idx1, segment1 in enumerate(segments):
54          for idx2, segment2 in enumerate(segments):
55              if idx1 == idx2:
56                  continue
57
58              if idx2 in pairs.values():
59                  continue
60
61              if segment1.is_boundary_close(segment2):
62                  pairs[idx1] = idx2
63
64      joined = list()
65      for key in pairs:
66          first = segments[key]
67          second = segments[pairs[key]]
68          first_points = first.points
69          second_points = second.points
70
71          if first.center.angle - second.center.angle > 0:
72              first_and_second = first_points + second_points
73          else:
74              first_and_second = second_points + first_points
75
76          new_segment = Object(points=first_and_second)
77          joined.append(new_segment)
78
79      for idx, segment in enumerate(segments):
80          if idx not in pairs.keys() and idx not in pairs.values():
81              joined.append(segment)
82
83      return joined
84
85
86  def segment_lidar_data(scan_points: List[Point]):
87      segments = []
```

```
88      segment = []
89      last_evaluated_point = scan_points[0]
90      last_in_object: Optional[Point] = None
91      skipped_points = 0
92      for current_point in scan_points:
93          distance_between_last_and_current_point = aux.
    get_distance_between_data_points(
94              last_evaluated_point, current_point
95          )
96
97          if current_point.range == 0 or current_point.range == -0:
98              # ignore points that have range equal to zero (exactly)
99              continue
100
101         if current_point.range < GlobalConfig.MIN_RANGE:
102             # ignore points that have range close to zero
103             continue
104
105         # compares to the last range measurement for the same angle
106         # current_point = compare_to_previous_measurement(
    previous_points=previous_points, last_evaluated_point=current_point)
107
108         if current_point.range >= GlobalConfig.MAX_RANGE:
109             # current_point = Point(range=float("Inf"), angle=
    current_point.angle)
110             continue
111
112         if (
113             distance_between_last_and_current_point
114             > GlobalConfig.SEGMENTATION_THRESHOLD
115         ):
116             """
117             uses the distance threshold criteria to determine if two
    points
118             are part of the same segment.
119             if the distance is greater than the threshold, the current
    segment
120             is closed and a new segment is created.
121             """
122
123             if bool(segment) is not False:
124                 if not aux.are_same_point(last_evaluated_point,
    last_in_object):
125                     # checks if the last evaluated was included in the
    previous
126                     # segment before closing it and creating the next
    one
```

```
127
128                     segment.append(last_evaluated_point)  # INCLUDES
129
130                 # CLOSES SEGMENT
131                 closed_segment = Object(points=segment)
132                 segments.append(closed_segment)
133
134             # OPENS NEW SEGMENT
135
136             segment = [current_point]  # INCLUDES
137             last_in_object = current_point
138             skipped_points = 0
139
140         else:
141             if aux.are_same_point(last_evaluated_point, last_in_object):
142                 last_evaluated_point = current_point
143                 continue
144
145             if skipped_points < GlobalConfig.MAX_SKIPPED and
    should_be_excluded(
146                 current=current_point,
147                 last_evaluated=last_evaluated_point,
148                 last_in_object=last_in_object,
149             ):
150                 skipped_points += 1
151                 pass
152             else:
153                 segment.append(current_point)  # INCLUDES
154                 last_in_object = current_point
155                 skipped_points = 0
156
157         last_evaluated_point = current_point
158
159     return segments
```