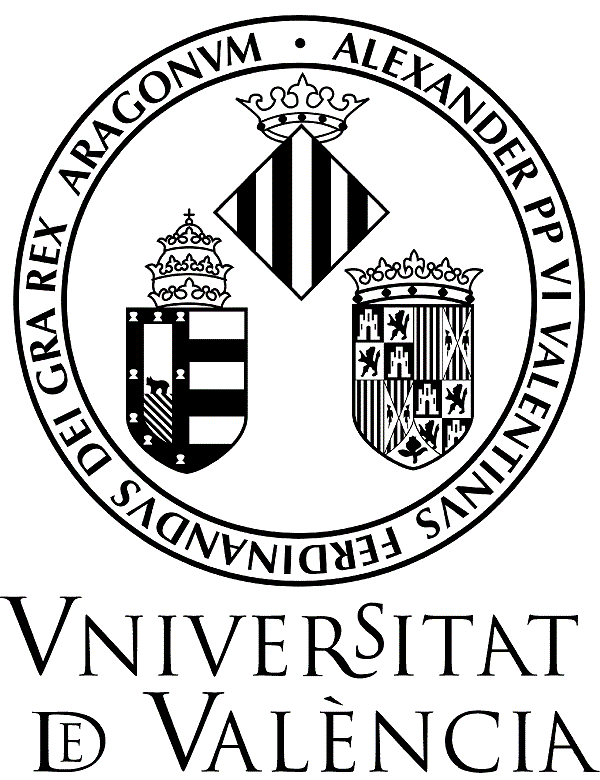
#### 

**Màster Universitari en Enginyeria Electrònica Industrial**

****

**Treball de Fi de Màster**

**SISTEMA D’ESTACIONAMENT PER A VEHICLES AUTOMATIZATS AMB TECNOLOGIA LIDAR**

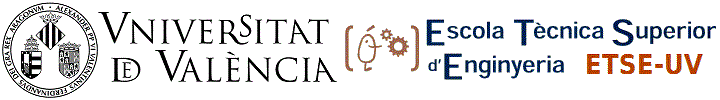
**autor/a:**

Jose Cuevas Mauricio

**Tutor:**

Vicent Girbes Juan

**Març 2025**

****

**Màster Universitari en Enginyeria Electrònica Industrial**

**Treball de Fi de Màster**

**SISTEMA D’ESTACIONAMENT PER A VEHICLES AUTOMATIZATS AMB TECNOLOGIA LIDAR**

**autor/a:**

Jose Cuevas Mauricio

**tutor:**

Vicent Girbes Juan

**Tribunal:**

|  |  |
| --- | --- |
| Presidente: | Vocal 1: |
| Vocal 2: | **Fecha de defensa:**  **Calificación:** |

**ÍNDEX**

[ÍNDEX DE FIGURES 4](#_Toc196254973)

[Resum 5](#_Toc196254974)

[Abstract 5](#_Toc196254975)

[1 Introducció 6](#_Toc196254976)

[1.1 Motivació 6](#_Toc196254977)

[1.2 Objectius 7](#_Toc196254978)

[2 Estat de l’art 7](#_Toc196254979)

[2.1 Avanços en robòtica mòbil 7](#_Toc196254980)

[2.2 Diferencies entre ROS1 i ROS2 9](#_Toc196254981)

[3 Metodologia 10](#_Toc196254982)

[3.1 Descripció de l’entorn 11](#_Toc196254983)

[3.2 Programa inicial per a una execució 17](#_Toc196254984)

[3.2.1 Càlcul de la posició inicial del robot 17](#_Toc196254985)

[3.2.2 Processament de dades del LiDAR i càlcul del punt de destinació 18](#_Toc196254986)

[3.2.3 Execució del moviment de docking 27](#_Toc196254987)

[3.2.4 Lectura de la posició final mitjançant odometria 29](#_Toc196254988)

[3.2.5 Avaluació del resultat i registre de mètriques 29](#_Toc196254989)

[3.3 Programa amb múltiples iteracions 30](#_Toc196254990)

[3.3.1 Reposicionament aleatori 30](#_Toc196254991)

[3.3.2 Navegació fins a l’origen 31](#_Toc196254992)

[3.3.3 Estructura del bucle principal 32](#_Toc196254993)

[4 Experimentació en l’entorn simulat amb Gazebo i ROS1 32](#_Toc196254994)

[4.1 Anàlisis algoritme de Docking per a una única iteració 32](#_Toc196254995)

[4.1.1 Configuració inicial 32](#_Toc196254996)

[4.1.2 Càlculs previs 33](#_Toc196254997)

[4.1.3 Representació en Geogebra 35](#_Toc196254998)

[4.1.4 Moviment del robot 36](#_Toc196254999)

[4.1.5 Registre de resultats 36](#_Toc196255000)

[4.2 Anàlisis algoritme de Docking per a 10 iteracions 37](#_Toc196255001)

[5 Transició i adaptacions a ROS2 38](#_Toc196255002)

[5.1 Introducció a la migració 38](#_Toc196255003)

[5.2 Adaptació del codi per a qualsevol LiDAR 38](#_Toc196255004)

[5.3 Implementació del moviment basat en odometria 41](#_Toc196255005)

[5.3.1 Explicació de la funció mover\_robot() 42](#_Toc196255006)

[5.3.2 Característiques del control proporcional 43](#_Toc196255007)

[5.4 Especificacions tècniques del robot físic en ROS2 43](#_Toc196255008)

[5.5 Descripció del codi complet en ROS2 44](#_Toc196255009)

[5.5.1 Estructura general del node 44](#_Toc196255010)

[5.5.2 Control proporcional 45](#_Toc196255011)

[5.5.3 Odometria: lectura i ús 45](#_Toc196255012)

[5.5.4 Diferències respecte a la versió ROS1 46](#_Toc196255013)

[5.6 Resultats obtinguts en ROS2 47](#_Toc196255014)

[5.6.1 Condicions inicials de la prova 47](#_Toc196255015)

[5.6.2 Generació de dades i anàlisi del LiDAR 48](#_Toc196255016)

[5.6.3 Resultat final del moviment 49](#_Toc196255017)

[5.6.4 Trajectòria real del robot 50](#_Toc196255018)

[6 Conclusions i Millores 52](#_Toc196255019)

[6.1 Propostes de millora futures 53](#_Toc196255020)

[7 Bibliografia 53](#_Toc196255021)

[8 Annexos 54](#_Toc196255022)

[8.1 Annex A. Repositori de GitHub 54](#_Toc196255023)

[8.2 Annex B. Codi del fitxer habitacion.word 54](#_Toc196255024)

[8.3 Annex C. Codi docking\_once\_inicial.cpp 58](#_Toc196255025)

[8.4 Annex D. Codi docking.cpp 71](#_Toc196255026)

[8.5 Annex E. Resultados.txt 89](#_Toc196255027)

[8.6 Annex F. Codi docking\_once\_lidar.cpp 91](#_Toc196255028)

[8.7 Annex G. Codi docking\_once\_odom.cpp 104](#_Toc196255029)

[8.8 Annex H. Codi docking\_once\_ROS2\_odom.cpp 118](#_Toc196255030)

# ÍNDEX DE FIGURES

[Figura 1: Entorn simulat amb Gazebo 12](#_Toc196255031)

[Figura 2: Descripció paret 1 Gazebo 13](#_Toc196255032)

[Figura 3: Habitació interior per al respawn 14](#_Toc196255033)

[Figura 4: Placa de referència 15](#_Toc196255034)

[Figura 5: Llançament .launch 15](#_Toc196255035)

[Figura 6: Fitxer launch 16](#_Toc196255036)

[Figura 7: Main Funcio lidarToFile() 18](#_Toc196255037)

[Figura 8: Càcul coordenades y dels extrems 20](#_Toc196255038)

[Figura 9: Reprentació en Geogebra de la placa 21](#_Toc196255039)

[Figura 10: Obtenció punt de destí (Pd) 22](#_Toc196255040)

[Figura 11: Punt de destí amb pendent negativa 23](#_Toc196255041)

[Figura 12: Distància entre Origen i punt Destí 24](#_Toc196255042)

[Figura 13: Giro\_Inicial en el primer quadrant 25](#_Toc196255043)

[Figura 14: Càlcul Giro\_Inicial al codi per al primer quadrant 25](#_Toc196255044)

[Figura 15: Punt destí en segon quadrant 26](#_Toc196255045)

[Figura 16: Gir inicial 28](#_Toc196255046)

[Figura 17: Avanç 28](#_Toc196255047)

[Figura 18: Gir final 29](#_Toc196255048)

[Figura 19: Posició inicial 33](#_Toc196255049)

[Figura 20: Resultats fitxer .txt 33](#_Toc196255050)

[Figura 21: Valors punts placa 34](#_Toc196255051)

[Figura 22: Representació gràfica punts cartesians 35](#_Toc196255052)

[Figura 23: Repesentació en geogebra 35](#_Toc196255053)

[Figura 24: Posició final 36](#_Toc196255054)

[Figura 25: Resultats odometria 36](#_Toc196255055)

[Figura 26: Posició inicial docking\_once\_lidar.cpp 40](#_Toc196255056)

[Figura 27: Gràfica Lidar 3240 41](#_Toc196255057)

[Figura 28: Rviz ROS2 Posició Inicial 47](#_Toc196255058)

[Figura 29: Foto posició inicial 48](#_Toc196255059)

[Figura 30: Representació gràfica posició inicial 49](#_Toc196255060)

[Figura 31: Rviz ROS2 Posició Final 49](#_Toc196255061)

[Figura 32: Foto posició final 50](#_Toc196255062)

[Figura 33: Inicial 51](#_Toc196255063)

[Figura 34: Posicio intermitja 1 51](#_Toc196255064)

[Figura 35: Posició intermitja 2 51](#_Toc196255065)

[Figura 36: Final 52](#_Toc196255066)

# Resum

Aquest treball de fi de màster presenta el desenvolupament i adaptació d’un sistema de *docking* autònom per a un robot mòbil mitjançant dades de LiDAR i odometria, integrat tant en ROS1 com en ROS2. Inicialment, el sistema es va implementar en ROS1 (Noetic) amb simulació en Gazebo, utilitzant un sensor LiDAR de 360 feixos. Posteriorment, es va migrar a ROS2 per tal de realitzar proves amb un robot físic proporcionat per la Universitat de València, que incorpora un sensor LiDAR d’alta resolució amb 3240 valors.

El sistema desenvolupat detecta una placa de referència mitjançant les dades del LiDAR, calcula la trajectòria necessària per arribar-hi (gir inicial, avanç i gir final) i executa els moviments utilitzant control proporcional basat en la lectura contínua de la odometria. El codi ha sigut redissenyat per ser escalable, permetent l’ús de qualsevol LiDAR independentment del nombre de feixos, i s’han adaptat els fitxers de simulació (.xacro i .world) per a representar fidelment el comportament del sensor real.

Tot i que les proves completes en ROS2 amb el robot físic es van veure limitades per problemes de robustesa, es va validar el sistema en simulació i es van obtenir resultats visuals del comportament del robot. A més, el treball inclou un repositori de codi completament funcional, amb l’estructura organitzada per facilitar la reutilització i l’ampliació en futurs projectes.

# Abstract

This master's thesis presents the development and adaptation of an autonomous docking system for a mobile robot using LiDAR data and odometry, integrated in both ROS1 and ROS2. The system was initially implemented in ROS1 (Noetic) and simulated in Gazebo using a LiDAR sensor with 360 beams. Later, the system was migrated to ROS2 in order to conduct tests with a physical robot provided by the University of Valencia, equipped with a high-resolution LiDAR sensor delivering 3240 values per scan.

The developed system detects a reference docking plate using LiDAR data, computes the required trajectory (initial rotation, forward movement, and final rotation), and executes the maneuvers using proportional control based on continuous odometry feedback. The code has been redesigned to be scalable, allowing compatibility with any LiDAR regardless of the number of beams. Simulation files (.xacro and .world) were also adapted to accurately reflect the real sensor’s behavior.

Although full tests in ROS2 with the physical robot were limited due to robustness issues, the system was validated in simulation, and visual evidence of the robot’s behavior was obtained. Furthermore, the project includes a fully functional and well-structured GitHub repository, facilitating reuse and further development in future applications.

# Introducció

En les últimes dècades, la robòtica ha experimentat un creixement exponencial, impulsada per la demanda de solucions automatitzades en sectors tan diversos com la indústria, la salut, l'agricultura i la logística. Malgrat aquests avenços, encara hi ha reptes significatius que dificulten l'adopció i l'aplicació universal de la tecnologia robòtica. Entre aquests reptes s'inclouen:

1. La navegació i localització precisa: En entorns dinàmics o amb obstacles, garantir que un robot es puga moure de manera eficient i precisa és fonamental.
2. La interacció segura amb humans i altres robots: Cal desenvolupar algoritmes i tecnologies que asseguren una col·laboració segura.
3. La sostenibilitat energètica: Molts sistemes robòtics tenen una autonomia limitada, la qual cosa redueix la seva eficàcia operativa.
4. La integració amb altres tecnologies: Per aconseguir sistemes més intel·ligents, cal superar les barreres de compatibilitat entre sensors, processadors i algoritmes.

Dins de la problemàtica general, el docking esdevé un repte clau per a robots mòbils autònoms. Aquest procés permet que el robot es dirigisca cap una ubicació específica —com ara una estació de càrrega— i s'alinee amb precisió per complir amb la seva funció, ja siga recarregar-se, descarregar càrrega o interactuar amb un dispositiu fix.

El docking és especialment crític en aplicacions on es requereix una operació contínua del robot sense intervenció humana. Tot i això, dissenyar un algoritme robust que assegure una alineació precisa i repetitiva és una tasca complexa.

En aquest cas en particular, el docking s'ha resolt mitjançant un algoritme dissenyat específicament per al sistema ROS i simulat en el robot Waffle3 amb un sensor LiDAR.

Aquest sistema permet una detecció precisa d’una placa (de 0.5 metres) que representa una estació de càrrega simulada. L'algoritme processa les dades del LiDAR per calcular el moviment necessari perquè el robot es posicione al centre de la placa.

Aquest desenvolupament s'ha provat en entorns simulats utilitzant Gazebo, garantint un entorn de prova controlat per perfeccionar l'algoritme abans de desplegar-lo en entorns reals.

## Motivació

Durant les pràctiques del màster, vaig tindre l'oportunitat de col·laborar amb l'empresa Proemisa, que es dedicava al desenvolupament de robots autònoms amb aplicacions específiques per a entorns sanitaris. Un dels seus projectes destacats consistia en la creació d'un robot capaç de moure's autònomament per les habitacions per dur a terme tasques de desinfecció, especialment rellevants durant la pandèmia de COVID-19.

Aquest robot disposava d'un sistema avançat per realitzar un mapeig previ de l'entorn, permetent-li conèixer la distribució de l'espai, així com un sistema de detecció i evitació de col·lisions que assegurava la seva operació segura en espais dinàmics. La meva tasca principal en aquest projecte va ser desenvolupar l'algoritme de docking que permetés al robot posicionar-se amb precisió en una estació designada per realitzar la recàrrega.

## Objectius

L'objectiu principal d'aquest treball és desenvolupar i adaptar un sistema d'acoplament autònom (docking) per a un robot mòbil, basat en la detecció d'una placa de referència mitjançant dades de LiDAR i l'ús d'odometria per guiar els moviments del robot. Aquest objectiu general es desglossa en els següents objectius específics:

1. **Implementar un sistema funcional en ROS1 (Noetic)** amb simulació en Gazebo, capaç de detectar la posició de la placa mitjançant un sensor LiDAR de 360 valors i executar el moviment fins a ella.
2. **Desenvolupar una versió escalable del codi**, capaç d’adaptar-se automàticament a qualsevol sensor LiDAR, independentment del nombre de feixos, gràcies a la detecció dinàmica de la mida del vector de rangs.
3. **Modificar els fitxers de descripció del robot (.xacro)** per simular correctament sensors de diferent resolució (360 i 3240 valors), assegurant una correspondència realista entre simulació i realitat.
4. **Migrar el sistema a ROS2 (Humble)** per poder executar-lo en un robot físic real proporcionat per la Universitat de València, mantenint la funcionalitat i adaptant el codi a l'arquitectura de ROS2.
5. **Implementar un sistema de moviment basat en odometria** amb control proporcional (PID simplificat) que permeta una navegació precisa fins al punt de destinació calculat a partir de la informació del LiDAR.
6. **Validar el comportament del sistema en simulació i en el robot físic**, documentant la posició inicial, final i la trajectòria, i registrant dades de rendiment (fitxers .txt, .csv i vídeos).
7. **Organitzar i documentar tot el projecte** en un repositori GitHub, incloent-hi el codi, fitxers de llançament, imatges, vídeos i resultats, per facilitar la reutilització i millora futura del sistema.

# Estat de l’art

## Avanços en robòtica mòbil

En les darreres dècades, la robòtica mòbil ha experimentat un creixement exponencial, impulsat per avanços en sensors, intel·ligència artificial i sistemes de control. Aquesta disciplina se centra en el desenvolupament de robots capaços de desplaçar-se de manera autònoma o semiautònoma en entorns dinàmics i no estructurats, oferint solucions en camps com el transport, la logística, l’exploració i el sector serveis.

Els sistemes actuals combinen tecnologies avançades de percepció, com càmeres RGB-D i sensors LiDAR, amb algoritmes de localització i mapatge simultani (SLAM), permetent als robots construir mapes precisos i localitzar-se amb alta precisió. Així mateix, les tècniques d’aprenentatge automàtic i planificació de trajectòries han millorat significativament la capacitat dels robots per prendre decisions òptimes en temps real. Aquest avanç tecnològic ha propiciat l’adopció de la robòtica mòbil en aplicacions automatitzades, des de vehicles autònoms fins a robots de lliurament i manipulació.

Un exemple destacat d’aquests avanços és el treball de Feng et al. (2024), que presenta el sistema STORM (*Self-configurable and Transformable Omni-Directional Robotic Modules*), un sistema modular auto-reconfigurable capaç de canviar de configuració i acoblar-se autònomament. Mitjançant un esquema dual de control —un controlador basat en la funció de Lyapunov per a alineacions precises i un algoritme d’optimització de trajectòries per a moviments més amplis—, es demostra la seva efectivitat en entorns dinàmics. Aquest sistema ofereix solucions versàtils per superar les limitacions de robots amb estructures fixes i millorar les capacitats en tasques complexes.

A més, Bostelman et al. (2016) aporten una revisió integral sobre el *docking* en robots mòbils i vehicles guiats automàticament (AGVs). Aquesta operació, fonamental per a tasques com la càrrega i descàrrega de materials, depèn d’un control precís i l’ús de sensors avançats com visió, làser i RFID. Els autors descriuen diverses tècniques de control, com clothoides, lògica difusa i xarxes de Petri, que han millorat significativament el rendiment del *docking*. També s’analitzen innovacions com l’ús de braços robòtics mòbils i es destaquen normes de seguretat i patents clau en aquest camp.

La combinació d’aquests estudis exemplifica l’abast i la complexitat de la recerca en robòtica mòbil i automatitzada, posant les bases per explorar més avanços. En futurs apartats, s’integraran altres treballs rellevants que complementen aquestes perspectives, analitzant aspectes com la interacció col·laborativa entre robots, l’ús d’intel·ligència artificial per a la presa de decisions en temps real i les aplicacions en entorns industrials connectats mitjançant IoT.

Aquest apartat serveix com una visió inicial, amb l’objectiu de construir una panoràmica completa de l’estat actual de la robòtica mòbil, tot destacant els sistemes de percepció, navegació i presa de decisions que impulsen aquesta disciplina.

En (Jia et al., 2023) es presenta un mètode innovador per a l'acoblament i recàrrega autònoms de robots mòbils en entorns de manufactura. Aquest mètode combina tècniques de visió per computadora i sensors LiDAR per a millorar la precisió i eficiència del procés d'acoblament i recàrrega. Els desafiaments en la recàrrega autònoma són crucials per a que els robots mòbils operen de manera contínua sense intervenció humana. Els mètodes existents, com els basats en sensors d'infrarojos, visió i làser, enfronten desafiaments significatius degut a les condicions d'il·luminació, el clima i la necessitat de càlculs complexos. El mètode proposat utilitza un model de detecció d'objectes basat en YOLOv7 i un sensor LiDAR per a reconèixer i localitzar estacions de càrrega en entorns de manufactura. La combinació de dades de visió i LiDAR permet una estimació precisa de la posició del robot i millora el procés d'acoblament. Es va utilitzar una càmera Hikvision per a recollir dades i construir un conjunt de dades per al carregador sense fils. El mètode va ser provat en escenaris del món real, aconseguint una precisió mitjana del 95% en el reconeixement de l'estació de càrrega. El mètode proposat millora la precisió i eficiència del reconeixement i acoblament d'estacions de càrrega en comparació amb mètodes anteriors. La fusió de dades de visió i LiDAR redueix els costos computacionals i augmenta la robustesa del sistema. Encara que el mètode és efectiu, presenta limitacions, com la necessitat de que l'estació de càrrega estiga en un espai tancat per a calcular la distància amb precisió. Futures investigacions se centraran en l'ús de càmeres estèreo i sistemes LiDAR per a millorar encara més l'estimació de la posició i la precisió de l'acoblament. Aquest enfocament té un gran potencial per a aplicacions en entorns de manufactura autònoms, on la precisió i l'eficiència són essencials per a l'operació contínua de robots mòbils.

L’article de Romanov & Tararin (2021) tracta un algorisme per a robots mòbils amb cinemàtica diferencial, que augmenta la precisió de posicionament i orientació per a l'acoblament a una estació de càrrega. La localització de l'estació de càrrega es realitza mitjançant la detecció de marcadors ArUco. A diferència d'altres mètodes de localització basats en sensors IR i LIDAR, aquest mètode és més robust i fàcil de configurar. El programari dissenyat basat en l'algorisme proposat i el Sistema Operatiu de Robots (ROS) proporciona una integració senzilla amb robots de diferents dissenys. L'algorisme inclou quatre etapes: cerca de l'estació, gir cap a l'estació, càlcul del punt de centrament i moviment cap a l'estació. Els experiments realitzats en el simulador Gazebo van demostrar l'efectivitat de l'algorisme, amb una precisió de +-5 cm i una distància mínima per a l'inici de l'acoblament de 58 cm. La cerca de l'estació es realitza girant el robot al voltant del seu eix a un cert angle i detectant el marcador ArUco. Si el marcador no es troba en 60 segons, l'algorisme es deté. Una vegada detectat el marcador, el robot es gira cap a l'estació i es calcula el punt de centrament, que és el punt més proper al robot situat en una línia recta que passa pel centre del marcador i perpendicular al seu pla. Després, el robot es mou cap a l'estació ajustant l'angle de rotació i la velocitat lineal en proporció a la distància a l'estació. Els resultats dels experiments mostren que tots els punts de centrament es troben dins del àrea de la qual el robot pot moure's i acoblar-se amb precisió a l'estació de càrrega, la qual cosa demostra que l'algorisme d'acoblament automàtic funciona correctament. La precisió de l'algorisme és superior a altres mètodes basats en la detecció de marcadors visuals o volumètrics.

## Diferencies entre ROS1 i ROS2

ROS1 (Robot Operating System 1) ha estat durant molt de temps una de les plataformes més utilitzades per al desenvolupament de sistemes robòtics, oferint una infraestructura robusta per a la creació i simulació de robots. Tanmateix, a mesura que la robòtica avançava, ROS1 va començar a mostrar algunes limitacions, especialment en termes de rendiment, escalabilitat i compatibilitat amb noves tecnologies. Això va conduir al desenvolupament de ROS2, una versió millorada que busca superar aquestes limitacions i proporcionar una plataforma més robusta per als robots del futur.

Una de les diferències més destacades entre ROS1 i ROS2 és el sistema de comunicació. ROS1 fa servir un sistema de comunicació basat en Publish/Subscribe mitjançant el middleware ROS Master, que gestiona la descoberta de nodes i el seu enllaç. Aquest enfocament presenta certs problemes en termes d’escalabilitat i robustesa en entorns distribuïts, ja que depèn d’un servidor centralitzat (ROS Master). En canvi, ROS2 utilitza DDS (Data Distribution Service) com a middleware de comunicació, el que permet una comunicació més descentralitzada i resilient, eliminant la dependència d'un servidor central i millorant el rendiment en entorns distribuïts i multi-robot. A més, DDS ofereix suport per a la qualitat del servei (QoS), la qual cosa permet un millor control sobre la fiabilitat, la seguretat i el rendiment de les comunicacions.

ROS2 ha estat dissenyat des del principi per a ser més escalable i per a suportar entorns de temps real, una característica que no estava plenament implementada en ROS1. Això és fonamental en aplicacions robòtiques on la resposta immediata és essencial, com en els robots autònoms que operen en entorns dinàmics. ROS2 aprofita les capacitats de DDS per garantir un comportament determinista i proporcionar una comunicació amb latència reduïda, mentre que en ROS1, la manca de suport natiu per a temps real feia que no fos adequat per a moltes aplicacions crítiques.

ROS2 inclou millores en seguretat, una característica que no estava present de manera robusta en ROS1. Els mecanismes de seguretat en ROS2 es basen en el sistema de seguretat de DDS, que ofereix encriptació, autenticació i control d’accés. Això fa que ROS2 sigui més adequat per a aplicacions que requereixen altes garanties de seguretat, com en la robòtica industrial o en aplicacions militars.

Una altra diferència important és la compatibilitat. ROS2 no és totalment compatible amb ROS1, cosa que significa que les aplicacions desenvolupades en ROS1 no es poden transferir directament a ROS2 sense algunes modificacions. Per això, ROS2 inclou eines com ros1\_bridge, que permet la interoperabilitat entre ROS1 i ROS2, facilitant la transició gradual de sistemes basats en ROS1 cap a ROS2.

ROS2 s’ha dissenyat per ser més flexible i extensible, permetent la integració amb noves tecnologies com la computació en el núvol, la robòtica col·laborativa i la intel·ligència artificial (IA). Aquesta flexibilitat és fonamental per als avenços futurs en robòtica, on es necessiten plataformes capaces de suportar entorns i tecnologies emergents. Per exemple, ROS2 és més adequat per a la integració amb plataformes com Azure Robotics i altres sistemes de computació distribuïda.

En resum, ROS2 supera moltes de les limitacions de ROS1, oferint una plataforma més robusta, escalable i segura per a les aplicacions robòtiques de propera generació.

# Metodologia

En aquest treball es presenta el desenvolupament i implementació d’un sistema de docking autònom per a un robot AGV simulat, basat en el model Waffle3. L’objectiu principal és que el robot detecte una placa específica mitjançant les dades proporcionades pel sensor LiDAR de 360° i es desplace cap al centre d’aquesta de manera precisa i autònoma.

La metodologia emprada es divideix en diverses fases interconnectades que abasten des de la configuració del simulador fins a la validació del sistema. Primer, es descriu la configuració de l’entorn simulat utilitzant Gazebo i la integració del robot amb el framework ROS Noetic. Com que ROS treballa de manera nativa en entorns Linux, s’ha utilitzat el subsistema Windows per a Linux (WSL2) per a habilitar un entorn Linux dins del sistema operatiu Windows. Aquesta configuració ha sigut essencial per al desenvolupament i l’execució de simulacions amb Gazebo i ROS.

A continuació, es detalla el disseny de l’algoritme de detecció i moviment, basat en l’anàlisi de dades del LiDAR i l’ús d’un conjunt de funcions personalitzades per controlar el moviment del robot cap al centre de la placa detectada.

Aquest enfocament permet validar la capacitat del robot per completar el procés de docking de manera eficient, assegurant la precisió en el posicionament final. La metodologia també inclou proves iteratives per ajustar els paràmetres del sistema, garantir-ne la robustesa i adaptar-lo a diverses situacions simulades.

Aquesta estructura metodològica garanteix que el sistema desenvolupat siga replicable, robust i aplicable en entorns reals similars.

Finalment, cal destacar que el sistema desenvolupat també s’ha adaptat per funcionar en ROS2, amb l’objectiu de garantir la compatibilitat amb diferents tipus de sensors LiDAR, independentment del nombre de feixos per volta. Per a aconseguir-ho, es va modificar el codi de manera que aquest puga llegir dinàmicament la longitud del vector de dades del LiDAR, permetent una escalabilitat més gran i una millor adaptabilitat a diversos entorns. Mentre que les proves inicials i la validació de l’algoritme es van dur a terme en un entorn simulat amb Gazebo i ROS1, la versió per a ROS2 es va verificar mitjançant l’ús d’un robot físic real, assegurant així la viabilitat del sistema tant en simulació com en condicions reals.

## Descripció de l’entorn

Com s’ha mencionat anteriorment, el simulador emprat per a dur a terme les proves ha sigut Gazebo. Per a això, s’ha dissenyat un fitxer .world que representa una habitació rectangular de 4 × 6 metres, amb parets d’una alçària de 20 cm, suficients perquè el sensor LiDAR les detecte correctament. A més, s’ha inclòs una placa verda d’una amplària de 50 cm i amb la mateixa alçària que les parets, situada amb el seu centre a les coordenades (-0.5, 0) en el pla XY.

L’últim element és un rectangle rosa, delimitat per quatre cantons interiors en les coordenades (2, 1), (2, -1), (3, 1) i (3, -1). Aquest rectangle defineix la zona en la qual el Waffle pot realitzar el respawn de manera aleatòria dins d’aquesta posició en cada iteració del programa. Els objectes que defineixen aquest rectangle no tenen component de col·lisió en el fitxer .world, ja que són purament visuals, permetent que el robot els travesse sense problemes.

Aquest arxiu es pot consultar a l’Annex B. Codi del fitxer habitacion.word.

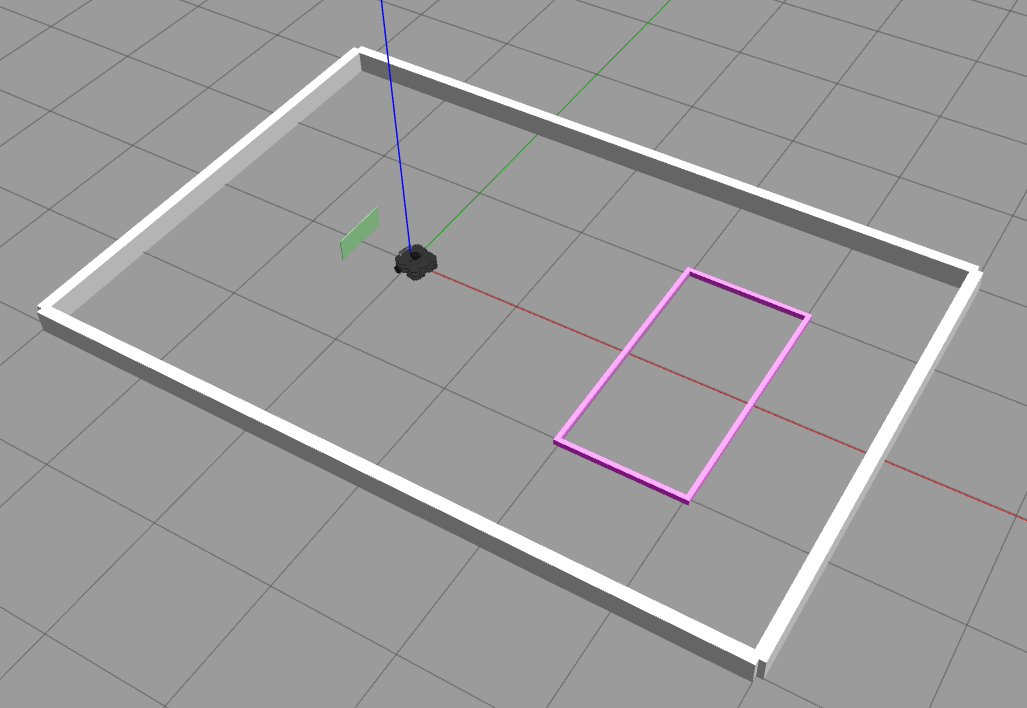


Figura 1: Entorn simulat amb Gazebo

La Figura 1 il·lustra l’entorn simulat descrit anteriorment, amb el robot Waffle3 situat en l’origen de coordenades (0, 0). Aquesta configuració es genera automàticament en executar el fitxer .launch que s’explicarà més endavant.

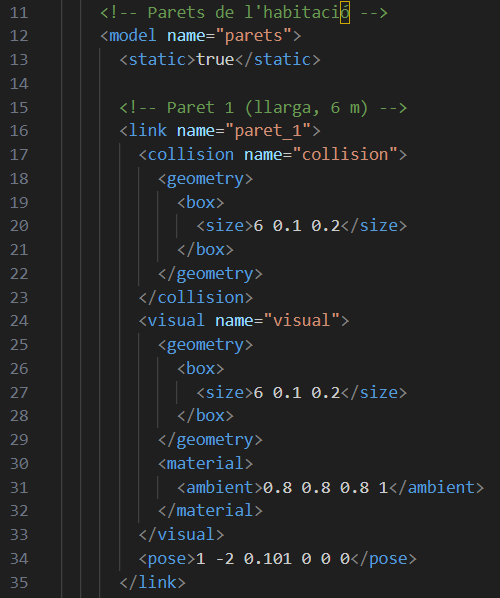


Figura 2: Descripció paret 1 Gazebo

En la Figura 2 es mostra una de les quatre parets que representen l’habitació simulada en Gazebo dins del fitxer habitacion.world.

Aquesta paret està definida amb els paràmetres següents:

<size>6 0.1 0.2</size>, on la primera coordenada (6) indica la llargària en l’eix X, la segona (0.1) representa l’amplària en l’eix Y i la tercera (0.2) defineix l’alçària en l’eix Z. Aquesta configuració està inclosa tant en els elements de “collision” com en els de “visual”, per la qual cosa la paret és visible en la simulació i actua com a barrera física per al robot. El paràmetre <ambient>0.8 0.8 0.8 1</ambient> especifica el color de la paret i la seua opacitat. Els tres primers valors (0.8 0.8 0.8) corresponen a les components RGB (roig, verd i blau), indicant una intensitat del 80% per a cadascun dels colors, cosa que dona lloc a un gris clar. L’últim valor (1) defineix l’opacitat, sent 1 completament opac i 0 completament transparent.

El paràmetre <pose>1 -2 0.101 0 0 0</pose> indica la posició i l’orientació del centre de la paret en l’espai. Les primeres tres coordenades (1 -2 0.101) especifiquen la posició del centre de l’objecte en els eixos X, Y i Z, respectivament. Les següents tres coordenades (0 0 0) defineixen l’orientació en radians respecte als eixos de rotació X, Y i Z, seguint el format roll-pitch-yaw. Roll representa la rotació al voltant de l’eix X, pitch representa la rotació al voltant de l’eix Y i yaw representa la rotació al voltant de l’eix Z. En aquest cas, totes les orientacions són 0, cosa que significa que la paret està alineada amb els eixos globals sense cap rotació. Les altres 3 parets estan representades de manera similar.

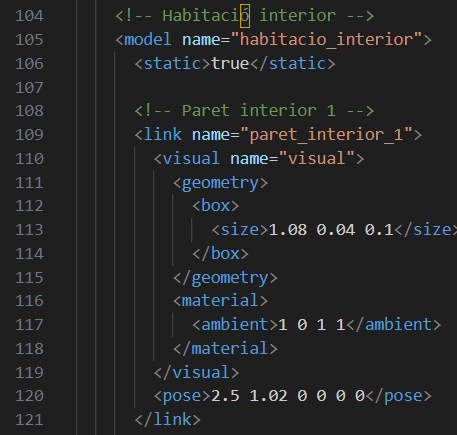


Figura 3: Habitació interior per al respawn

En la Figura 3 es pot observar que l'objecte no té component de col·lisió, atés que només es defineix com a element visual. Això vol dir que, tot i ser visible en la simulació, no interactua físicament amb el robot ni impedeix el seu moviment. A més, el color de l'objecte està especificat amb un ambient de valor 1 0 1 1, que representa el color rosa, amb 100% de roig i 100% de blau.



Figura 4: Placa de referència

Per acabar amb l’explicació del fitxer .world, la Figura 4 mostra la placa de referència a la qual el robot ha de posicionar-se davant. La combinació <ambient>0.2 0.7 0.2 1</ambient> dels valors RGB li atorga el color verd, amb 20% de roig, 70% de verd i 20% de blau, sent completament opaca.

Una volta creat el fitxer .world, es crea un fitxer .launch que permetrà llançar el món mencionat anteriorment juntament amb el robot Waffle per començar la simulació del programa. La instrucció a llançar en la terminal serà:

* roslaunch tfm\_simulation habitacion.launch

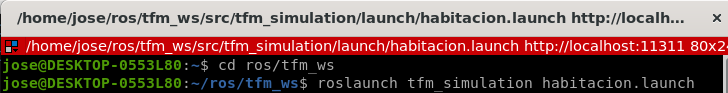


Figura 5: Llançament .launch

Com es pot observar en la Figura 5, aquest fitxer s’ha d’executar en la mateixa ruta de la nostra carpeta de treball.



Figura 6: Fitxer launch

A continuació, s'explica el contingut del fitxer habitacion.launch de la Figura 6:

1. Arguments per al model de robot i posició inicial:

Es defineixen els arguments del fitxer, com el model del robot que s'ha de carregar. Per defecte, el model és " default="waffle" ", però també es poden seleccionar altres models com "burger" o "waffle\_pi". A més es defineixen els arguments per a la posició inicial del robot en l'espai 3D, concretament les coordenades x, y i z, amb valors predeterminats a "0.0". El valor de "yaw" es defineix com 2.9 radians (166°), que indica l'orientació inicial del robot.

1. Incloure el món creat (habitacion.world):

Mitjançant l’etiqueta: <include file="$(find gazebo\_ros)/launch/empty\_world.launch">, s’incorpora el fitxer de llançament d'un món buit de Gazebo, empty\_world.launch, que es personalitza per tal de carregar el món de simulació desenvolupat previament, anomenat habitacion.world.

1. Carregar la descripció del robot:

Es carrega la descripció del robot (Waffle) "robot\_description" mitjançant el fitxer URDF generat amb xacro. Es fa referència a la descripció del robot mitjançant el paquet turtlebot3\_description, especificant el model seleccionat (per exemple, Waffle).

1. Llançar el model del robot en la posició especificada:

Finalment, s'utilitza el node "spawn\_model" del paquet "gazebo\_ros" per llançar el robot en la simulació, indicant el model que s'ha de crear (), així com la seva posició inicial (x, y, z) i orientació (yaw).

Amb aquest fitxer .launch, es llançarà el simulador Gazebo amb el món creat i el robot Waffle en la posició especificada, per poder iniciar la simulació del programa.

## Programa inicial per a una execució

Per tal d’assolir el docking de manera autònoma, s’ha desenvolupat un programa en C++ (Annex C. Codi docking\_once\_inicial.cpp) que integra diferents funcions per gestionar tant la percepció com el moviment i l’avaluació del robot. Aquest programa segueix una seqüència estructurada que permet calcular la trajectòria des del punt d’origen fins a una placa de referència utilitzant dades del sensor LiDAR i la informació d’odometria.

El procés complet inclou les següents etapes:

1. **Obtenció de la posició inicial del robot** mitjançant dades d’odometria.
2. **Processament de les dades del sensor LiDAR** per identificar la placa, calcular els seus extrems, determinar el punt objectiu i calcular els angles i la distància del moviment.
3. **Execució del moviment del robot**, que inclou un gir inicial, un desplaçament lineal i un gir final.
4. **Obtenció de la posició final del robot** per avaluar la precisió del moviment.
5. **Càlcul dels resultats**: distància a l’objectiu, desviació angular i temps emprat.

A continuació, s’expliquen detalladament les funcions que formen aquest programa.

### Càlcul de la posició inicial del robot

La primera funció executada és odom\_inicio(), i s’encarrega de capturar la posició inicial del robot abans d’iniciar el moviment de docking. Per fer-ho, el programa es subscriu al topic /odom, que publica les dades d’odometria.

Funcionament:

* Es crea un node ros::NodeHandle i un subscriptor al topic odom, amb un callback que només s’executa una vegada.
* Quan s’obté el primer missatge, es guarden tres valors fonamentals:
  + La posició del robot en X i Y.
  + L’orientació en l’eix Z (yaw), que es calcula a partir del quaternió amb la funció getYawFromQuaternion().
* Aquesta informació es guarda en variables globals (inicial\_x, inicial\_y, inicial\_w) i també s’escriu en un fitxer .txt per a la seua posterior avaluació.
* A més, s’emmagatzema l’hora exacta d’inici del procés, en format HH:MM:SS, que serà útil per calcular el temps total d’execució.

L’objectiu es establir un punt de partida fiable que permeta, al final del procés, comparar la posició assolida pel robot respecte a la posició ideal.

### Processament de dades del LiDAR i càlcul del punt de destinació

La funció void lidarToFile() és essencial en aquest treball, perquè s'encarrega de recollir les dades del LiDAR per identificar la posició de la placa de referència. Mitjançant una interpolació dels eixos de llum del LiDAR que detecten la placa, es pot calcular la seva ubicació amb precisió. Aquesta funció realitza diversos càlculs per identificar la posició i orientació de la placa de referència, i utilitza aquestes dades per calcular la trajectòria del robot. A continuació s'explica detalladament el que fa la funció:

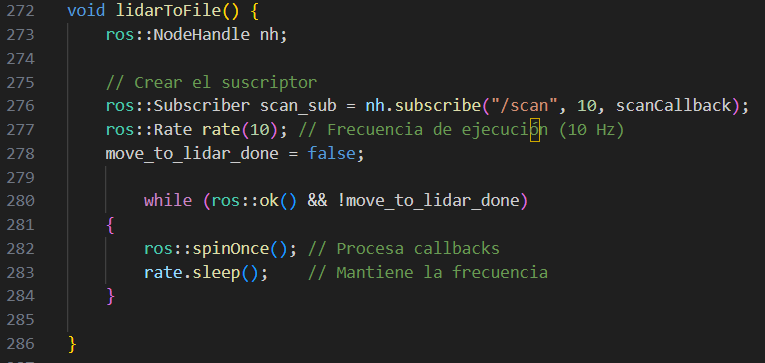


Figura 7: Main Funcio lidarToFile()

En la Figura 7, s’obserba que la funció té com a objectiu gestionar la subscripció al topic de ROS anomenat /scan, que proporciona dades del sensor LiDAR del robot. A grans trets, aquesta funció s'encarrega de processar la informació publicada en aquest topic mitjançant un callback.

Es crea un objecte ros::NodeHandle nh;, que serveix com a interfície principal per interactuar amb els recursos de ROS, com ara crear subscripcions, publicacions i serveis. Es declara un subscriptor utilitzant ros::Subscriber. Aquest subscriptor està configurat per escoltar el topic /scan amb una cua de 10 missatges. Quan hi haja noves dades en aquest topic, es cridarà automàticament la funció de callback scanCallback, que serà responsable de processar aquestes dades. La funció entra en un bucle while que continua executant-se mentre el sistema ROS estiga actiu (ros::ok()) i la variable move\_to\_lidar\_done siga false. Aquesta variable es modificarà a true al final del callback. Dins del bucle, s'executa ros::spinOnce();, una funció que processa les cridades als callbacks associats als topics o serveis que el node haja definit. En aquest cas, això assegura que scanCallback es cride quan hi haja dades noves en el topic /scan. La funció utilitza un objecte ros::Rate rate(10); per mantenir una freqüència de 10 Hz. Això implica que el bucle s'executa aproximadament cada 100 ms.

Es processen les dades del LiDAR dins de:



Primer, es divideixen les lectures del LiDAR en dues parts: valors per l’esquerra (de 0 a -179 graus que equivalen als valors de ranges[0] a ranges[179]) i valors per la dreta (de 1 a 180 graus que equivalen als valors de ranges[359] a ranges[180]).

S'analitzen els valors de distància per trobar el valor mínim de la distància de les lectures del LiDAR. Aquest valor és el més proper a l'objecte (la placa de referència).

**Filtrat de Lectures vàlides:**

Després, s'examinen les distàncies de les lectures dins de certs límits (msg->range\_min i msg->range\_max), i es filtra només aquelles lectures que són vàlides (distàncies dins del rang permès).

Aquestes lectures es guarden en dues llistes: valid\_ranges\_left i valid\_ranges\_right, que corresponen a les lectures de l'esquerra i de la dreta de la placa.

**Transformació de Coordinades de Polars a Cartesianes:**

Les dades del LiDAR es guarden en format polar (distància i angle). Per treballar millor amb elles en el sistema de coordenades cartesianes, es realitza la transformació utilitzant les següents fórmules:

**X = distància \* sin(angle)**

**Y = distància \* cos(angle)**  
Aquestes fórmules permeten obtenir les coordenades x i y dels punts mesurats pel LiDAR.

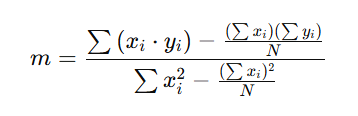
**Interpolació per Determinar les Coordenades dels Extrems de la Placa:**

Un cop obtingudes les coordenades cartesians de les lectures, es fa una interpolació per calcular les coordenades exactes dels extrems de la placa.

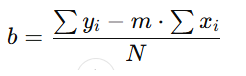
A través de la interpolació lineal, s'usen les lectures de les dues zones (esquerra i dreta) per estimar el punt mig de la placa i la seva orientació.

Es calcula la pendent (m) i la intersecció (b) de la recta que passa pels punts extrems de

la placa:



m = (sum\_xy - (sum\_x \* sum\_y) / n) / (sum\_x2 - (sum\_x \* sum\_x) / n)

****

b = (sum\_y - m \* sum\_x) / n

Aquesta fórmula estima la recta equivalent a la posició de la placa.

La coordenada x del valor més extrem a l’esquerra de la placa s’obte d’aquesta manera:

x\_extrem\_izq = x\_coords.back(); // Coordenada x del extrem esquerre de la placa

on x\_coords.back():

* x\_coords és un vector que emmagatzema les coordenades X calculades per als punts vàlids detectats pel LiDAR.
* El mètode .back() accedeix a l'últim element del vector x\_coords, és a dir, la coordenada X més recent afegida al vector.

La línia assigna el valor de l'últim punt X en el vector x\_coords a la variable x\_extrem\_izq.

El mateix es fa per a la coordenada de la dreta:

x\_extrem\_der = x\_coords.back(); // Coordenada x del extrem dret de la placa

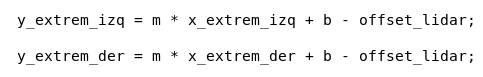


Figura 8: Càcul coordenades y dels extrems

Es fa servir l'equació de la recta: y=m⋅x+b

m: És la **pendent** de la recta, calculada prèviament en funció dels punts detectats.

b: És la **ordenada a l'origen**, és a dir, el punt on la recta talla l'eix Y.

Després de calcular les coordenades Y segons l'equació de la recta, es resta l'**offset\_lidar**. Aquest **offset** serveix per corregir el valor de Y, ajustant-lo per la posició relativa del LiDAR respecte el centre de gir. En aquest cas, fa que les coordenades Y dels extrems es desplacen cap avall respecte al sistema de referència del robot.

Amb les coordenades dels extrems de la placa, es calcula el punt mig, que és el punt central de la placa:

Pmx = (x\_extrem\_izq + x\_extrem\_der) / 2

Pmy = (y\_extrem\_izq + y\_extrem\_der) / 2

A partir d'aquest punt mig, es calcula una línia perpendicular a la recta de la placa, i es determina el punt de destinació a una distància de 0.15 metres (variable global: float dist\_to\_placa = 0.15) del centre de la placa. Aquest càlcul es fa a partir de la inversa de la pendent de la recta de la placa, utilitzant la fórmula:

m\_inv = -1/m (pendent perpendicular)

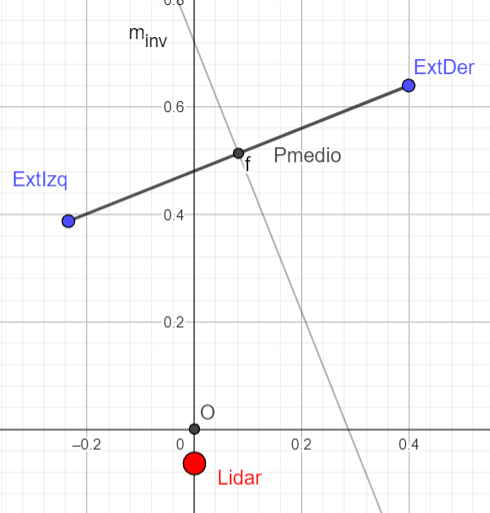


Figura 9: Reprentació en Geogebra de la placa

La Figura 9 descriu un possible exemple a partir de les dades obtingudes amb el LiDAR en la funció descrita anteriorment. En aquest cas els punts dels extrems tenen els següents valors:

* ExtIzq= (-0.23497,0.38728)
* ExtDer= (0.39913,0.63974)

Per tant el punts mitja serà: Pmedio=PuntoMedio(ExtIzq,ExtDer) =(0.08,0.51)

En aquesta situació ja s’ha aplicat el Offset causat per el LiDAR, que es troba a 0.064 metres per baix del centre de gir, per tant tots els càlculs següent es faran a partir del punt Origen (O).



Figura 10: Obtenció punt de destí (Pd)

Per calcular l’angle tecta emprem la seguent formula al codi:



On m\_inv és la pendent inversa de la Placa. Ens assegurem que l’angle obtingut mitjançant l’arctangent siga positiu sempre.

Amb el valor de tecta podem calcular les coordenades del punt de destí al que ha d’arribar el robot al acabar el Docking a 0.15 metres de la Placa (dist). La coordenada y (Pdy) sempre s’obtindrà de la mateixa manera:



Com el punt de destí sempre es trobarà per baix del punt mig, es restarà el sinus de tecta multiplicat per la distancia a la placa a la coordenada y del punt mig (Pmy).

Per a la component x dependrà de si la pendent de la placa es positiva o negativa. En el cas de la Figura 10, la pendent es positiva, per tant també s’ha de sumar a Pmx el cosinus de tecta multiplicat per la distància a la placa:



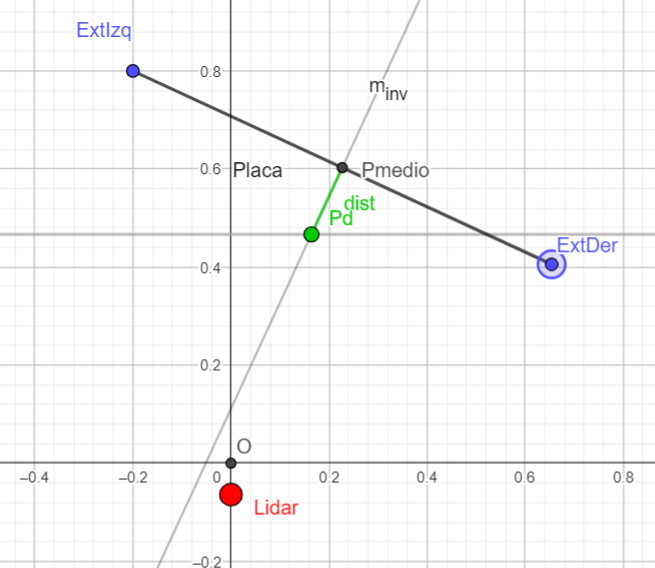


Figura 11: Punt de destí amb pendent negativa

Com es pot observar en la Figura 11, en aquest cas s’ha de restar a Pmx el cosinus de tecta multiplicat per la distància a la placa:

float Pdx = Pmx + (m / fabs(m)) \* dist\_to\_placa \* cos(tecta)

on el factor (m / fabs(m)) serà negatiu.

**Càlcul de la Distància i els Angles de Gir:**

A partir de la posició del punt de destinació, es calcula la distància entre el robot i aquest punt utilitzant la fórmula de distància euclidiana:

dist\_orig\_Pd = sqrt(Pdx*Pdx + Pdy*Pdy)

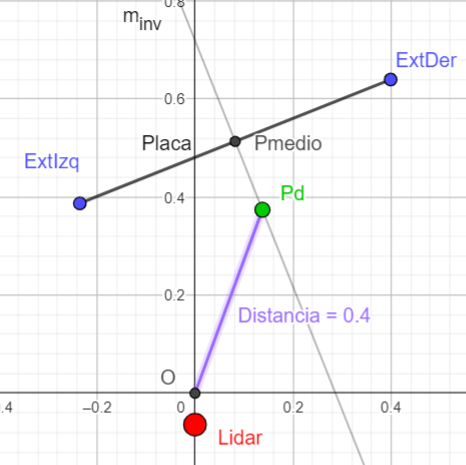


Figura 12: Distància entre Origen i punt Destí

Aquesta distància de la Figura 12 s'utilitza per determinar com de lluny està el robot del punt de referència de la placa.

Després, es calcula l'angle de gir necessari per a orientar-se cap al punt de destinació. Els possibles angles de gir depenen de la ubicació del punt de destinació. Per al primer gir, anomenat giro\_inicial s’ha de tindre en compte si el punt es troba en el primer o segon quadrant. Per tant, hi haurà dos casos possibles:

* Cas 1: Punt en el primer quadrant

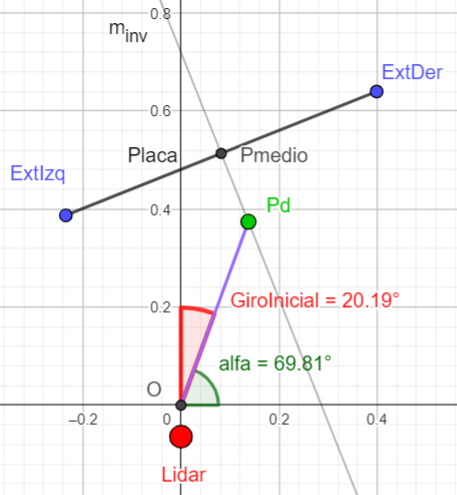


Figura 13: Giro\_Inicial en el primer quadrant

Primer es calcula el valor d’alfa:



A continuació es calcula el valor de Giro\_Inicial:

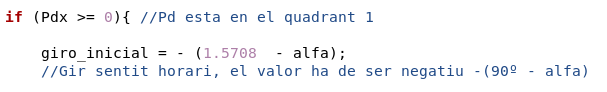


Figura 14: Càlcul Giro\_Inicial al codi per al primer quadrant

Tal com es pot observar en la Figura 13, el resultat final ha de ser negatiu, atés que el gir del robot s’ha d’efectuar en sentit horari. En l’entorn ROS, el sentit del gir es defineix mitjançant el comandament cmd\_vel, que controla la velocitat angular del robot.

En concret, el camp angular.z determina la rotació del robot al voltant del seu eix vertical. Un valor negatiu en aquest camp indica un gir en sentit horari (en el sentit de les agulles del rellotge), mentre que un valor positiu genera un gir en sentit antihorari. Per tant, per aconseguir que el robot gire correctament en sentit horari, el valor de angular.z ha de ser negatiu.

En el cas considerat en el codi, el valor 1.5708 correspon a π/2 radians, equivalent a 90 graus. Per calcular el gir inicial necessari, es resta el valor d'alfa, que ha estat calculat prèviament i representa l'angle entre l'eix de referència del robot i el punt objectiu Pd. La diferència (π/2 - alfa) proporciona l'angle requerit, mentre que el signe negatiu indica que el gir es fa en sentit horari.

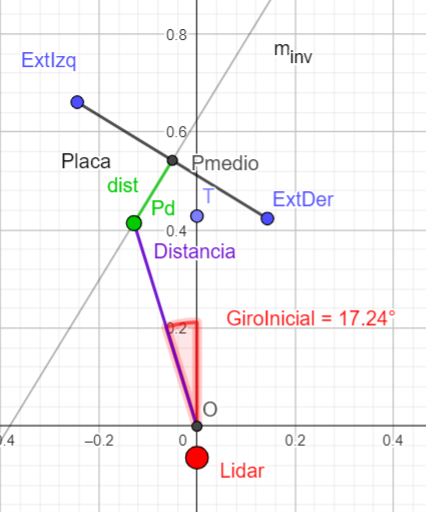
* Cas 2: Punt en el segon quadrant
* 

Figura 15: Punt destí en segon quadrant

En el cas en què el punt objectiu **Pd** es troba en el **segon quadrant** com en la Figura 15 (és a dir, quan la component Pdx és negativa), el robot haurà de girar en sentit **antihorari** per alinear-se correctament cap al punt desitjat. En l’entorn ROS, aquest tipus de gir el missatge **cmd\_vel**, haurà de tindre un valor positiu, per tant Giro\_Inicial també ho serà.

Una vegada el robot ha realitzat el giro\_inicial i l’avanç cal calcular l’últim moviment angular, conegut com giro\_final, que permet ajustar la posició del robot respecte a la inclinació de la placa de referència. Per calcular el valor de giro\_final, cal considerar tant la posició del punt de destinació Pd (en quin quadrant es troba) com la pendent de la placa (positiva o negativa). Segons aquestes dues variables, es poden distingir quatre situacions típiques:

* Situació 1: Punt destí en el primer quadrant i pendent de la placa positiva

Quan el punt Pd es troba en el primer quadrant (Pdx >= 0) i la pendent de la placa és positiva (m >= 0), el giro\_final es calcula com:  
giro\_final = π−alfa−tecta  
Aquest càlcul considera la diferència entre l’angle de 180 graus, el valor d’alfa i l’angle d’inclinació de la placa tecta.

* Situació 2: Punt destí en el primer quadrant i pendent de la placa negativa

Si el punt Pd està en el primer quadrant (Pdx >= 0), però la pendent de la placa és negativa (m < 0), el giro\_final es calcula com:  
giro\_final = tecta−alfa  
Ací es resta el valor d’alfa de l’angle d’inclinació de la placa, ja que el moviment es fa en sentit horari.

* Situació 3: Punt destí en el segon quadrant i pendent de la placa positiva

Quan el punt Pd es troba en el segon quadrant (Pdx < 0) i la pendent de la placa és positiva (m >= 0), el giro\_final es calcula com:  
giro\_final = alfa − tecta  
Aquest càlcul reflecteix que el moviment es fa en sentit antihorari per ajustar-se a la inclinació de la placa.

* Situació 4: Punt destí en el segon quadrant i pendent de la placa negativa

Si el punt Pd està en el segon quadrant (Pdx < 0) i la pendent de la placa és negativa (m < 0), el giro\_final es calcula com:  
giro\_fina l= −(π−alfa−tecta)  
El signe negatiu indica que el gir es fa en sentit horari, ajustant-se a l'orientació requerida per la pendent de la placa.

Aquest enfocament permet calcular de manera sistemàtica el giro\_final en funció de la ubicació del punt destí i la inclinació de la placa. Així, el robot pot completar l’alineació final necessària per interactuar correctament amb el punt de destinació.

**Emmagatzematge de les Dades**:

Totes les dades importants (com les coordenades dels extrems, el punt mig, la distància entre extrems, els càlculs trigonomètrics i els angles de gir) es guarden en dos fitxers: un fitxer TXT i un fitxer CSV. Aquestes dades es poden utilitzar per realitzar anàlisis posteriors, incloent l'estudi gràfic de la trajectòria i orientació del robot.

Un cop s'han realitzat tots els càlculs i s'han emmagatzemat les dades, la funció marca move\_to\_lidar\_done = true per indicar que el procés ha acabat.

Aquesta funció integra diversos passos de processament de dades del LiDAR per obtenir les coordenades precises de la placa de referència i calcular els moviments del robot per realitzar el docking.

### Execució del moviment de docking

La funció mover\_robot() s'encarrega de controlar el moviment del robot en tres passos bàsics per assolir el punt de destí, ajustant el comportament segons els valors de les variables globals giro\_inicial, avance i giro\_final. A continuació es descriu breument el funcionament:

**Gir inicial**:

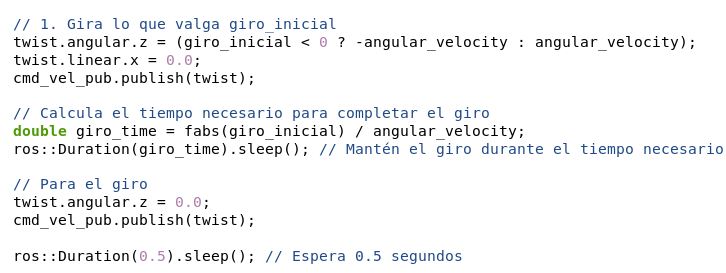


Figura 16: Gir inicial

El robot gira un angle determinat (giro\_inicial), ajustant la velocitat angular en funció del signe de l'angle (positiu o negatiu).

Es calcula el temps necessari per completar el gir basant-se en la velocitat angular fixada i es manté el moviment durant aquest temps.

Finalment, el gir s'atura amb una publicació de velocitat angular zero.

**Avançar en línia recta**:

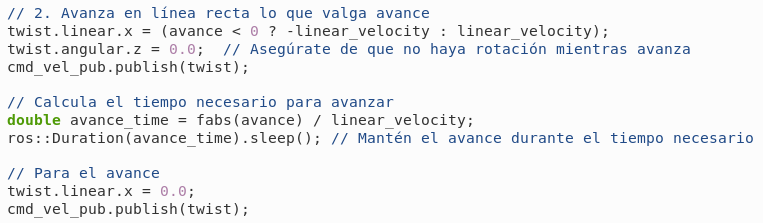


Figura 17: Avanç

El robot avança una distància específica (avance), ajustant la velocitat lineal en funció del signe de la distància (positiu per avançar, negatiu per retrocedir).

Es calcula el temps necessari per completar l'avanç basant-se en la velocitat lineal fixada i es manté el moviment durant aquest temps.

Finalment, l'avanç s'atura amb una publicació de velocitat lineal zero.

**Gir final**:

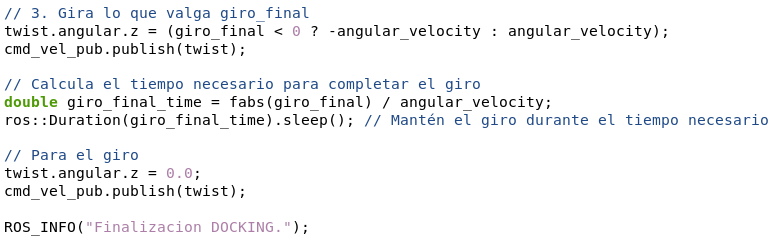


Figura 18: Gir final

El robot realitza un segon gir segons l'angle determinat per giro\_final, seguint el mateix procediment que en el primer gir.

S'atura després de completar aquest moviment.

La funció utilitza el mecanisme de publicació a través de ROS per enviar missatges al node /cmd\_vel, que controla les velocitats angulars i lineals del robot. A més, es fan pauses programades amb ros::Duration per assegurar-se que el robot realitza els moviments de forma precisa. Finalment, es registra al log de ROS la finalització del procés de docking

### Lectura de la posició final mitjançant odometria

Després de completar el moviment de docking, la funció odom\_final() s’utilitza per capturar la posició final del robot i comparar-la amb la posició inicial. Aquesta informació és essencial per avaluar la precisió del procés.

Els valors obtinguts (final\_x, final\_y, final\_w) es guarden en les mateixes variables globals que després s’utilitzaran per calcular la desviació respecte al punt de destinació. A més, igual que en la fase inicial, també s’emmagatzema l’hora exacta de finalització, que permet mesurar el temps total d’execució del procés de docking.

Finalment, totes aquestes dades es registren en el fitxer de resultats del projecte per a la seva posterior anàlisi.

### Avaluació del resultat i registre de mètriques

Un cop finalitzat el moviment de docking i obtinguda la posició final del robot, la funció calcularYEscribir() s'encarrega d'analitzar el rendiment del procés i registrar-ne les mètriques principals.

Aquesta funció compara la posició final del robot amb un punt de referència ideal situat a les coordenades (-0.35, 0) i amb orientació π radians. Amb aquesta comparació s’obtenen tres mètriques clau:

* **Distància euclidiana al punt de referència:**  
  S’estima la desviació en metres entre la posició final real i la posició objectiu.
* **Desviació angular:**  
  Es calcula la diferència en graus entre l’orientació final del robot (final\_w) i l’orientació ideal (π radians).
* **Temps total d’execució:**  
  Es calcula la diferència en segons entre l’hora d’inici (tiempo\_inicial) i l’hora de finalització (tiempo\_final), capturades en les funcions d’odometria.

Totes aquestes dades es guarden en el fitxer resultados.txt per tal de poder analitzar-les posteriorment. Aquest arxiu proporciona una base per estudiar l’eficàcia i la precisió del sistema de docking desenvolupat.

## Programa amb múltiples iteracions

Aquest nou programa, anomenat docking.cpp, amplia la funcionalitat del sistema anterior per tal d’avaluar la robustesa i la precisió del procés de docking al llarg de diverses execucions consecutives. El seu objectiu principal és realitzar **10 iteracions completes** del procés, començant cada vegada des d'una posició i orientació aleatòries dins de la zona específica de l'entorn simulat. D’aquesta manera, es poden recollir dades representatives de l’eficàcia del sistema en diferents condicions inicials. El codi complet d’aquest programa es pot consultar a l’Annex D. Codi docking.cpp

Les funcions lidarToFile(), mover\_robot(), odom\_inicio(), odom\_final() i calcularYEscribir() ja han estat explicades en l’apartat anterior, ja que comparteixen la mateixa estructura i funcionalitat que al programa docking\_once\_inicial.cpp. Tanmateix, en aquest nou context s'han introduït algunes modificacions per tal de registrar les dades de manera diferenciada per iteració. Per exemple, els arxius generats per cada execució s’identifiquen amb un número (ex.: lidar\_data\_iteracion3.txt), i s’inclouen etiquetes com “**Iteració X**” dins del fitxer de resultats.

A continuació es descriuen les funcions específiques del nou programa i la lògica del bucle principal

### Reposicionament aleatori

La primera funció, reset\_random(), té com a finalitat reposicionar el robot en una ubicació aleatòria dins del rectangle rosa esmentat prèviament a la simulació. Aquest reposicionament inclou tant una nova posició dins dels límits del rectangle com una orientació aleatòria en el pla XY. El canvi de posició s’efectua mitjançant una crida al servei /gazebo/set\_model\_state, al qual se li subministra un nou conjunt de coordenades i orientació calculades aleatòriament. Aquesta acció permet iniciar cada iteració des d’un escenari diferent, reproduint condicions diverses d’inici.

### Navegació fins a l’origen

La funció, void move\_origin(), s'encarrega de moure el robot fins a la posició (0,0) i orientar-lo cap a la placa de referència. L'objectiu d'aquesta funció és simular la navegació autònoma del robot utilitzant el sistema SLAM (Simultaneous Localization and Mapping), que permet a un robot construir un mapa de l'entorn mentre localitza la seva pròpia posició dins d'aquest mapa en temps real. El sistema SLAM és essencial per a la navegació autònoma en entorns desconeguts, ja que proporciona informació sobre la ubicació del robot i les seves trajectòries a partir de sensors, com els LiDAR o les càmeres, per evitar obstacles i arribar als punts desitjats. En aquest cas, el punt objectiu és la posició (0,0), on el robot ha de començar a realitzar el procés de docking, que consisteix a acoblar-se a una base de càrrega o una ubicació predeterminada per completar la tasca. El funcionament detallat de la funció és el següent:

**Inicialització de la publicació i subscripció**:

* Es crea un editor ROS per publicar els comandaments de moviment (cmd\_vel\_pub) i s'estableix la freqüència de publicació a 10 Hz.
* S'implementa un subscriptor per obtenir les dades d'odometria del robot a través de /odom.

**Fase de moviment cap a l'origen**:

* La funció comprova contínuament la posició del robot (coordenades x i y) i calcula l'angle necessari per orientar el robot cap al punt (0,0).
* Si el robot no està orientat correctament, ajusta l'orientació mitjançant un gir, fins que arribi a la posició correcta (aproximadament a x = 0 i y = 0).
* Un cop el robot es troba dins d'un radi de 0.1 metres de la posició objectiu, es marca que l'objectiu ha estat assolit i es fa una petita pausa.

**Fase de gir final**:

* Una vegada arribat a la posició (0,0), el robot ha de girar per orientar-se cap a la placa de referència. Es calcula la diferència d'angle entre la seva orientació actual i l'orientació desitjada.
* Si l'angle de diferència és superior a 0.2 radians, el robot continua girant fins que arribe a l'orientació correcta.

**Finalització del moviment**:

* Quan el gir final és completat, es publica un comandament de detenció per parar el robot i es marca que el procés ha acabat.
* La funció utilitza un bucle amb ros::spinOnce() per processar les dades d'odometria i mantenir la freqüència de control.

Aquesta funció simula la navegació autònoma del robot cap a un punt objectiu predefinit (l'origen), i la seva orientació per començar el procés de *docking* amb la placa de referència.

### Estructura del bucle principal

La funció main() del programa conté un bucle de **10 iteracions**, dins del qual es criden successivament totes les funcions necessàries per completar cada execució del docking. La seqüència de passos és:

1. Reposicionar el robot aleatòriament (reset\_random()).
2. Moure’l a l’origen i orientar-lo (move\_origin()).
3. Obtenir les dades inicials d’odometria (odom\_inicio()).
4. Analitzar les dades del LiDAR i calcular els moviments (lidarToFile()).
5. Executar els moviments (mover\_robot()).
6. Registrar la posició final (odom\_final()).
7. Calcular i guardar les mètriques d’error i temps (calcularYEscribir()).

Al final de cada iteració, el programa incrementa la variable global iteracion per controlar correctament el nom dels fitxers i la informació registrada.

# Experimentació en l’entorn simulat amb Gazebo i ROS1

Aquest capítol recull les proves realitzades en l'entorn de simulació utilitzant Gazebo i ROS1 per a validar el funcionament de l'algoritme de docking desenvolupat. Es presenten dos tipus d'experiments: una execució única del procés per a una anàlisi detallada, i una execució repetida durant 10 iteracions per a l’avaluació del rendiment global. Els codis font, així com els fitxers de resultats en format .txt i .csv, es poden consultar al repositori de GitHub, dins la carpeta corresponent a resultats, on també es troben representacions gràfiques del procés.

## Anàlisis algoritme de Docking per a una única iteració

En aquest apartat s’analitza exclusivament el comportament de la funció del docking. Per a això, s’ha creat un programa independent anomenat docking\_once\_inicial.cpp. L'objectiu és aïllar i examinar detalladament cada pas del procés, des de la configuració inicial fins al registre dels resultats finals, permetent una anàlisi precisa del rendiment de l'algoritme. Aquest programa segueix els passos següents:

### Configuració inicial

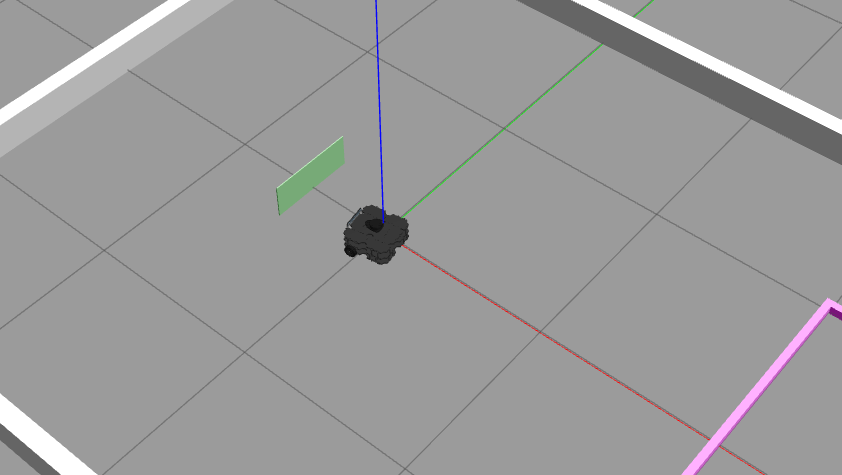


Figura 19: Posició inicial

El robot es col·loca en l'origen de l'entorn de simulació Gazebo, però amb una desviació en la seva orientació inicial. Aquesta configuració permet observar com el robot corregeix la seva trajectòria durant el procés de docking.

Es registren en el fitxer resultados.txt els valors odomètrics inicials. Els valors de la posició (coordenades *x* i *y*) i l'orientació (*yaw*) es guarden en un fitxer anomenat resultados.txt. Aquest registre és essencial per comparar les dades inicials amb les finals, determinant així la precisió del moviment.

### Càlculs previs

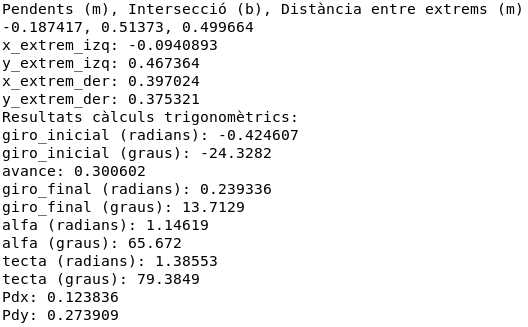


Figura 20: Resultats fitxer .txt

La funció lidartodata(), explicada a l'apartat anterior, s'utilitza per processar les dades del LiDAR del robot. Aquesta funció converteix les coordenades en format polar (distància i angle) en coordenades cartesianes (*x*, *y*). Això permet determinar amb precisió la posició dels punts d'interès en l'entorn.

|  |  |  |  |
| --- | --- | --- | --- |
| Figura 21: Valors punts placa |  |  |  |

En la Figura 21 es mostra els valors dels punts de la placa en coordenades polars i les seues coordenades corresponents en els eixos cartesians x i y.

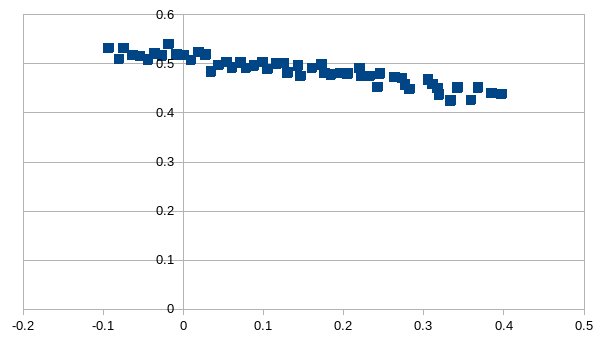


Figura 22: Representació gràfica punts cartesians

En laFigura 22, es mostra una representació gràfica dels punts en un sistema de coordenades cartesià, proporcionant una visualització clara de la posició relativa dels elements detectats.

### Representació en Geogebra

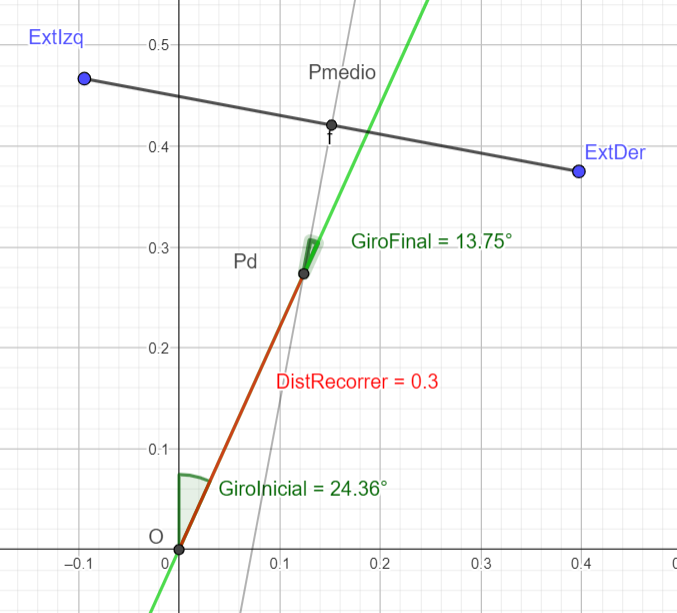


Figura 23: Repesentació en geogebra

Els punts cartesians obtinguts es representen gràficament en Geogebra (Figura 23) per comprovar la seva coherència amb l'entorn de simulació. Aquesta representació serveix com a eina visual per validar els càlculs i ajustar possibles desviacions en el procés.

### Moviment del robot

El robot executa els moviments calculats mitjançant la funció mover\_robot().

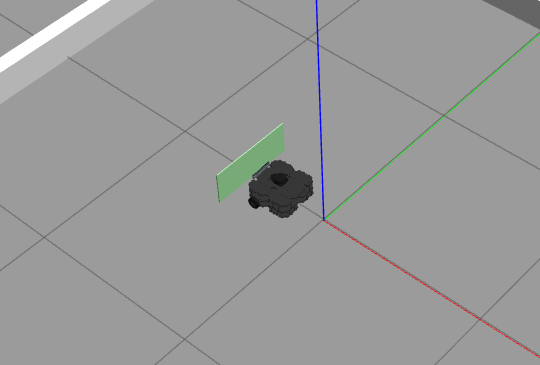


Figura 24: Posició final

Després de calcular les correccions necessàries amb lidartodata(), el robot executa els moviments corresponents mitjançant la funció mover\_robot().

Aquesta funció gestiona els moviments del robot en tres etapes:

* 1. Girar un angle inicial per alinear-se amb la direcció correcta.
  2. Avançar una distància determinada cap a l'objectiu.
  3. Realitzar un gir final per ajustar la seva orientació al punt de destí.

### Registre de resultats



Figura 25: Resultats odometria

Una vegada completats els moviments, es tornen a calcular els valors odomètrics (posició i orientació del robot).

Aquests valors es guarden de nou al fitxer resultados.txt (Figura 25), permetent comparar-los amb els valors inicials per avaluar a precisió en la correcció de la trajectòria i l'encert en la posició final respecte al punt objectiu.

Els resultats indiquen que l'algoritme de docking és altament precís en termes de posició (0.012 metres de desviació) i orientació (2.03° de diferència angular). A més, el temps emprat (11 segons) mostra una execució eficient per a aquesta tasca específica.

## Anàlisis algoritme de Docking per a 10 iteracions

Per complementar l’anàlisi detallada de l’apartat anterior, en aquest cas s’estudia el comportament global de l’algoritme de docking al llarg de 10 execucions consecutives. Per a això, s’ha implementat el programa docking.cpp, el qual incorpora totes les funcionalitats descrites prèviament al punt 3, com ara la recol·locació aleatòria del robot, la navegació cap a l’origen, la detecció de la placa mitjançant el LiDAR i l’execució dels moviments calculats.

Tots els fitxers generats durant les proves —tant els resultats numèrics (.txt) com les dades exportades (.csv) com el video— es troben disponibles al repositori de GitHub (Annex A. Repositori de GitHub), acompanyats dels codis font.

Durant cada iteració del bucle, s’executen les següents funcions en aquest ordre:

* **reset\_random()**: Reposiciona el robot de manera aleatòria dins del rectangle rosa definit a l’entorn de simulació.
* **move\_origin()**: Mou el robot cap a l’origen (0,0) i l’orienta correctament per iniciar el docking.
* **odom\_inicio()**: Captura i emmagatzema les coordenades i orientació inicials del robot.
* **lidarToFile()**: Processa les dades del sensor LiDAR per calcular la trajectòria de docking i les desa per a anàlisis posteriors.
* **mover\_robot()**: Executa els moviments necessaris (gir inicial, avanç, gir final) segons els càlculs anteriors.
* **odom\_final()**: Registra la posició i orientació finals del robot un cop acabat el moviment.
* **calcularYEscribir()**: Calcula la distància fins al punt de referència, l’error angular i el temps emprat, escrivint aquests valors en un fitxer de resultats.

L’objectiu d’aquesta bateria de proves és avaluar la robustesa i l’estabilitat del sistema en condicions inicials variables. L’anàlisi dels resultats obtinguts en les 10 iteracions, emmagatzemats al fitxer resultados.txt (Annex E. Resultados.txt), ha permés extraure les següents conclusions:

* **Iteració amb la distància més gran al punt (-0.35, 0)**: Iteració 5, amb una desviació de 0.037373 metres.
* **Iteració amb la diferència angular més gran respecte a π**: Iteració 7, amb un error de 11.4828 graus.
* **Iteració amb el temps d’execució més llarg**: Iteració 5, amb una duració de 13 segons.

Mitjanes dels valors registrats:

* Distància mitjana al punt (-0.35, 0): 0.0166 metres.
* Diferència angular mitjana respecte a π: 3.94 graus.
* Temps mitjà d’execució per iteració: 11 segons.

Aquests resultats demostren que el sistema és capaç de repetir el procés de docking amb un alt grau de precisió i estabilitat, malgrat les condicions inicials aleatòries. L’algoritme mostra un bon comportament general i constitueix una base sòlida per a futures millores o aplicacions en entorns reals.

# Transició i adaptacions a ROS2

## Introducció a la migració

Aquest projecte es va iniciar en ROS1 (Noetic) amb simulació en Gazebo, emprant un sensor LiDAR configurat amb 360 valors. Posteriorment, es va decidir adaptar el sistema a ROS2 per poder provar-lo en un robot físic proporcionat per la Universitat de València, que utilitza un sensor LiDAR d’alta resolució amb 3240 valors. Aquest canvi va implicar no només una migració del codi, sinó també una adaptació completa al nou entorn de hardware i al sistema operatiu ROS2.

A més, la migració va requerir modificar els scripts i dependències per adaptar-se a la nova arquitectura de ROS2, que presenta diferències significatives en la gestió dels nodes, la comunicació entre processos i la sintaxi del codi C++.

## Adaptació del codi per a qualsevol LiDAR

Amb el pas a ROS2 i la incorporació del robot físic de la Universitat de València, va ser necessari adaptar el sistema a un nou sensor LiDAR d’alta resolució amb 3240 valors per escaneig, en contrast amb els 360 valors del sensor emprat originalment en simulació amb ROS1. Aquesta diferència va obligar a redissenyar l’estructura del codi perquè fora totalment escalable i compatible amb qualsevol nombre de feixos del LiDAR.

El codi original assumia una mida fixa del vector ranges[]. Per resoldre aquesta limitació, es va implementar un nou enfocament en què el node detecta automàticament la mida del vector i divideix els valors en dos sectors (esquerra i dreta) segons la quantitat total de feixos (Annex F. Codi docking\_once\_lidar.cp):

int num\_beams = msg->ranges.size();

int half\_beams = num\_beams / 2;

Això permet adaptar el sistema a qualsevol sensor, sense necessitat de modificar manualment cap paràmetre. També es va ajustar el càlcul dels angles, garantint una conversió precisa en funció del nombre total de feixos.

A més, es van modificar els fitxers de descripció del robot per reflectir correctament el comportament dels diferents sensors LiDAR en l’entorn de simulació Gazebo. En concret, es van crear dues versions del fitxer turtlebot3\_waffle.gazebo.xacro:

* **turtlebot3\_waffle\_lidar360.gazebo.xacro**: la versió original, utilitzada en ROS1, amb 360 feixos. Inclou els paràmetres:

        <scan>

          <horizontal>

            <samples>360</samples>

            <resolution>1</resolution>

            <min\_angle>-3.14159</min\_angle>

            <max\_angle>3.14159</max\_angle>

          </horizontal>

        </scan>

* <samples>: Defineix el nombre total de feixos (360), és a dir, un per cada grau (resolució d'1º).
* <min\_angle> i <max\_angle>: Estableixen el rang angular del LiDAR entre -π i +π radians (−180º a +180º), centrant la mesura al front del robot.
* **turtlebot3\_waffle\_lidar3240.gazebo.xacro**: versió modificada per a un sensor d’alta resolució amb 3240 feixos. En aquest fitxer s’han actualitzat les línies:

        <scan>

    <horizontal>

      <samples>3240</samples>

      <angle\_increment>0.001937</angle\_increment>

      <min\_angle>0.0</min\_angle>

      <max\_angle>6.28319</max\_angle>

    </horizontal>

        </scan>

* <samples>: El sensor genera 3240 feixos per rotació, amb una resolució angular molt superior (aprox. 0.11º per feix).
* <angle\_increment>**:** Indica l’increment angular entre feixos en radians (0.001937 rad ≈ 0.111º).
* <min\_angle> i <max\_angle>: Defineixen el rang complet de 0 a 2π radians (0º a 360º), començant a mesurar des del darrere cap al front del robot, segons la configuració del sensor real.

Aquestes modificacions permeten una simulació coherent amb les característiques del sensor real que s’utilitza al robot físic en ROS2. Per garantir la compatibilitat, segons el tipus de LiDAR que es desitge simular, s’ha de copiar el fitxer corresponent amb el nom turtlebot3\_waffle.gazebo.xacro dins el directori del robot, sobreescrivint l’actual.

Un cop adaptat el fitxer .xacro, es va procedir a fer proves amb el nou codi docking\_once\_lidar.cpp, escalable a qualsevol LiDAR. A continuació es mostra una imatge de la posició inicial del robot (Figura 26) durant aquestes proves:

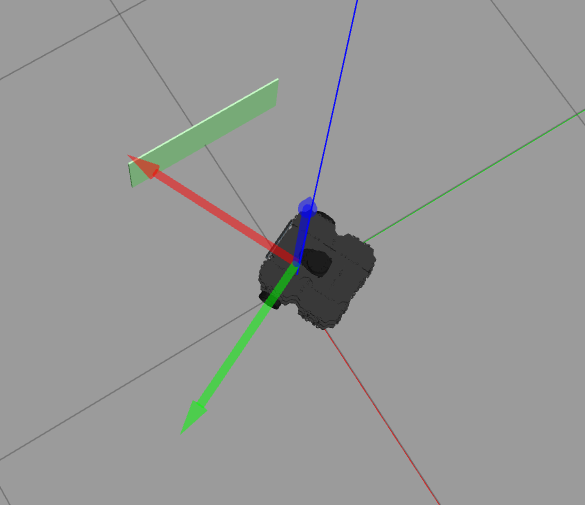


Figura 26: Posició inicial docking\_once\_lidar.cpp

També es va generar la representació gràfica dels punts detectats pel LiDAR en coordenades cartesianes, mostrant la placa de referència (Figura 27):

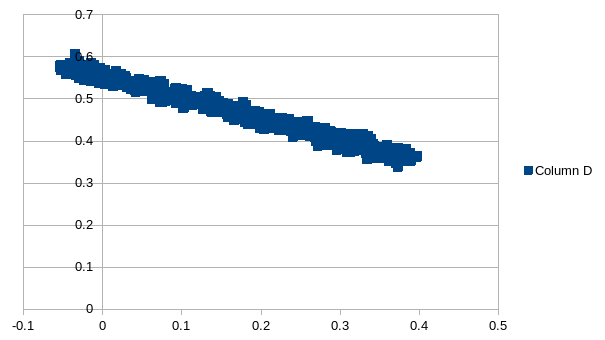


Figura 27: Gràfica Lidar 3240

Els fitxers de resultats generats automàticament per aquest nou codi (.txt i .csv) s’han organitzat dins una carpeta específica (“resultados\_once\_Lidar”) en el repositori de GitHub del projecte, juntament amb els fitxers launch necessaris per a la seva execució.

## Implementació del moviment basat en odometria

Degut als problemes detectats amb el sistema de moviment basat en el temps (sobretot quan es va intentar executar en el robot físic amb ROS2), es va decidir implementar un sistema de moviment més robust basat en **odometria**, primerament en l'entorn de simulació amb **ROS1 i Gazebo**.

Aquest nou enfocament es basa en:

* **Lectura de la posició inicial** del robot mitjançant el node /odom.
* **Avaluació contínua de la posició i orientació actuals** respecte al punt de destí.
* **Aplicació de control proporcional** per ajustar la velocitat lineal i angular segons la distància i orientació pendents.
* **Detenció automàtica** un cop s’ha assolit la posició o orientació desitjada amb prou precisió.

Aquest sistema permet una navegació més precisa i estable que la basada en temps, ja que es basa en la retroalimentació contínua de la posició del robot en lloc de suposar que es mourà una distància fixa en un temps determinat.

Aquest mètode es va implementar en el fitxer ros\_docking\_odom.cpp, el qual es pot consultar completament a Annex G. Codi docking\_once\_odom.cpp. Durant la simulació amb Gazebo, es va verificar que el robot aconsegueix realitzar el moviment de docking de manera precisa, corregint la seva posició i orientació de forma progressiva.

Tot i que en aquest apartat no s'inclouen imatges de la simulació, s’ha enregistrat un vídeo on es pot veure clarament el comportament del robot durant l'execució del procés de docking amb control per odometria. Aquest vídeo està disponible al **repositori de GitHub del project.**

### Explicació de la funció mover\_robot()

La funció mover\_robot() implementa el control del moviment del robot mitjançant odometria i control proporcional. Aquest mètode no es basa en temps fixes, sinó en l’error real entre la posició actual del robot i la posició o orientació desitjada. Aquesta estratègia és molt més precisa i estable, especialment en entorns reals on les condicions poden variar lleugerament.

La funció està dividida en tres fases, cadascuna amb control proporcional:

1. Gir inicial (giro\_inicial)

El robot realitza un gir per alinear-se en la direcció adequada.  
El moviment angular es calcula segons l’error entre la posició actual (yaw\_actual) i la posició objectiu (giro\_inicial):

        float yaw\_actual = yaw - yaw\_inicial;

        error = giro\_inicial - yaw\_actual;

        twist.angular.z = K \* error;

 Si l’error és positiu, el robot gira en sentit antihorari.

 Si és negatiu, gira en sentit horari.

 El moviment es manté fins que l’error angular siga inferior a 0.01 radians durant més de 10 iteracions consecutives.

2. Avanç lineal (avance)

Després del gir inicial, el robot avança cap al punt calculat.  
La distància actual es calcula mitjançant la distància euclidiana entre la posició inicial i l’actual:

        float distancia\_actual = sqrt(pow(x - x\_inicial, 2) + pow(y - y\_inicial, 2));

        float error\_dist = avance - distancia\_actual;

        twist.linear.x = K \* error\_dist;

 El robot avança mentre el desplaçament no ha superat la distància desitjada.

 Quan l’error és inferior a 0.01 metres de forma estable, el robot s’atur

3. Gir final (giro\_final)

Finalment, es realitza un segon gir per alinear el robot amb la placa.  
S’aplica el mateix control proporcional que en el primer gir:

        float yaw\_actual = yaw - yaw\_inicial;

        error = giro\_final - yaw\_actual;

        twist.angular.z = K \* error;

 El control proporcional permet una aproximació suau a l’orientació final.

 La fase acaba quan l’error angular es manté per sota del llindar establit.

### Característiques del control proporcional

 **K** és la constant de control que determina la rapidesa del moviment:  
Un valor més alt fa que el robot reaccione més ràpidament, però pot generar oscil·lacions.

 **Error** és la diferència entre el valor desitjat i el valor actual (distància o angle).

 L’ús de ros::spinOnce() en cada iteració permet obtenir dades d’odometria actualitzades contínuament.

Aquest enfocament fa que el moviment siga **progressiu i controlat**, evitant comportaments erràtics o desplaçaments incorrectes deguts a inèrcies, latències o variacions en la resposta dels motors.  
La funció mover\_robot() mostra clarament com la incorporació de dades d’odometria permet implementar una estratègia robusta per al docking, que posteriorment s’adaptarà al robot físic en ROS2.

## Especificacions tècniques del robot físic en ROS2

Per a les proves finals del projecte, es va utilitzar un robot físic proporcionat pel laboratori de la Universitat de València, equipat amb ROS2. Aquest robot permet posar a prova el sistema desenvolupat en un entorn real, amb condicions de hardware diferents respecte a la simulació.

A continuació es detallen les especificacions tècniques del robot, així com les funcions principals de cada component:

* **Xassís**: El robot compta amb un xassís metàl·lic i un sistema de tracció diferencial de quatre rodes. Aquest tipus d’estructura proporciona estabilitat i una bona maniobrabilitat en entorns interiors.
* **Computadora a borda**: El cervell principal del robot és una Raspberry Pi 4B amb 4 GB de memòria RAM, executant Ubuntu 20.04 i el sistema operatiu robòtic ROS2 Humble. Aquesta placa gestiona els nodes ROS, la captura i processament de dades dels sensors, i la comunicació amb els motors.
* **Microcontrolador STM32**: Aquest component s'encarrega de controlar els motors i llegir les dades dels encoders. Està connectat a la Raspberry Pi mitjançant USB i permet una comunicació eficaç i en temps real entre el control baix nivell i el sistema principal.
* **Sensor LiDAR**: El robot integra un sensor LiDAR 2D d'alta resolució, amb 3240 valors per volta, la qual cosa proporciona una detecció precisa de l'entorn. Aquesta resolució és molt superior a la utilitzada en la simulació amb ROS1, i va requerir adaptar el codi per a suportar aquest volum de dades.
* **Connexions**: La comunicació amb el robot es realitza principalment per SSH a través de WiFi, permetent un accés remot a la Raspberry Pi. El microcontrolador STM32 es connecta via USB, facilitant la transferència de dades i comandaments.
* **Altres sensors**: El sistema també disposa d'una IMU (Inertial Measurement Unit) per a detectar acceleracions i rotacions, i encoders per mesurar amb precisió la posició de les rodes. A més, el sistema admet la integració d’una càmera, encara que no ha estat necessària en aquest projecte.
* **Software utilitzat**: El robot fa ús de ROS2 Humble, amb eines com Gazebo per a simulació, RViz2 per visualitzar dades, rqt per monitoritzar nodes i gràfics, i scripts escrits en Python 3.10 per complementar el control i l'automatització de tasques.

## Descripció del codi complet en ROS2

L’última etapa del projecte consistí a integrar totes les funcionalitats prèvies dins d’un únic node ROS2, executable sobre el robot físic proporcionat per la Universitat de València. Aquest node es troba implementat en el fitxer docking\_once\_ROS2\_odom.cpp (vegeu Annex H. Codi docking\_once\_ROS2\_odom.cpp) i encapsula tot el procés de docking: des de la detecció de la placa mitjançant el sensor LiDAR, fins al moviment del robot utilitzant control proporcional basat en odometria.

### Estructura general del node

El node ROS2 ha estat dissenyat com una classe anomenada Docking, que hereta de rclcpp::Node. A diferència de ROS1, on normalment es treballa amb funcions globals i subscripcions directes, en ROS2 és habitual encapsular tota la funcionalitat dins d’una classe, gestionant les subscripcions, publicacions i altres recursos de manera més modular i escalable.

Aquest node executa la seqüència següent:

1. Espera a rebre el primer missatge de /odom i de /scan.
2. Guarda la posició inicial del robot mitjançant la funció odom\_inicio().
3. Processa les dades del LiDAR per localitzar la placa i calcular els punts extrems detectats.
4. Genera dos arxius (.txt i .csv) amb els resultats.
5. A partir de les dades geomètriques, calcula els tres moviments necessaris: giro\_inicial, avance i giro\_final.
6. Executa els moviments del robot amb un control proporcional sobre la odometria, a través de la funció mover\_robot().
7. Guarda la posició final amb odom\_final().
8. Calcula els errors finals respecte a un punt de referència (-0.35, 0.0) i l'angle de destinació (π radians), escrivint els resultats a resultados.txt.

### Control proporcional

Per executar els moviments amb precisió, s’ha implementat un sistema de control proporcional, tant per a les rotacions com per a l’avanç. A diferència del mètode anterior basat en càlculs de temps, aquest nou sistema mesura la posició actual mitjançant /odom i ajusta les velocitats en funció de l’error.

Les variables utilitzades per a aquest control són:

        float Kpv = 1.0, Kpw = 1.0; // Ganancia del control proporcional

        float Kiv = 0.0, Kiw = 0.0; // Ganancia del control integral (no)

Tot i que s’havia plantejat utilitzar un control PID (Proporcional-Integral-Derivatiu), només es va arribar a implementar la part proporcional. Les constants Kiv i Kiw corresponen al control integral, però van quedar desactivades (0.0) per manca de temps per a realitzar una posada a punt adequada. Amb el control proporcional es va obtindre un comportament suficientment estable i precís per als objectius del projecte.

Durant l'execució, es calcula l’error entre la posició real i l’objectiu, i aquest s’utilitza per ajustar la velocitat:

            float yaw\_actual = yaw - yaw\_inicial;

            error = giro\_inicial - yaw\_actual;

            int\_error = int\_error + error;

            w = Kpw \* error + Kiw \* int\_error;

S’han imposat límits màxims a aquestes velocitats per evitar moviments bruscos:

        float w\_max = 0.5, v\_max = 0.2;

### Odometria: lectura i ús

L’ús de la odometria és fonamental en aquesta versió. El node es subscriu al tema /robot02/odom i cada vegada que rep un missatge, extrau els valors de posició x, y i orientació yaw del robot:

  private:

    void odom\_callback(const nav\_msgs::msg::Odometry::SharedPtr msg) //const

    {

      if (!is\_processed) {

          is\_processed = true; // Marcar como procesado

      }

      odom\_ = \*msg;

      x = msg->pose.pose.position.x;

      y = msg->pose.pose.position.y;

      yaw = getYawFromQuaternion(msg->pose.pose.orientation);

      RCLCPP\_INFO(this->get\_logger(), "Odom values: position.x=%.2f, position.y=%.2f, yaw=%.4f", x, y, yaw);

    }

Per a la conversió del quaternion a yaw, s’empra la funció:

    // Función para convertir un cuaternión a yaw

    double getYawFromQuaternion(const geometry\_msgs::msg::Quaternion& q) {

        tf2::Quaternion quat(q.x, q.y, q.z, q.w);

        double roll, pitch, yaw;

        tf2::Matrix3x3(quat).getRPY(roll, pitch, yaw);  // Convierte cuaternión a ángulos de Euler

        return yaw;  // Devuelve el yaw (rotación alrededor del eje Z)

    }

El valor de yaw representa l’angle d’orientació del robot respecte a l’eix Z, i s’expressa en radians. Aquest valor pot oscil·lar entre −π i +π.

Els valors de posició inicial i final es guarden en arxius de text, juntament amb l’hora d’inici i de fi del moviment, per poder calcular posteriorment els errors i la duració del procés.

### Diferències respecte a la versió ROS1

Les principals diferències entre docking\_once\_odom.cpp (ROS1) i docking\_once\_ROS2\_odom.cpp (ROS2) són:

* **Estructura del node**: en ROS2 s’utilitza una classe (Docking) amb membres privats i mètodes, mentre que en ROS1 es feia ús de funcions globals i variables compartides.
* **Subscripcions i publicacions**: en ROS2 es creen amb create\_subscription() i create\_publisher(), i s’usa rclcpp::spin\_some() per processar les dades.
* **Temporització**: es fan servir funcions com rclcpp::sleep\_for() en lloc de ros::Duration().sleep().
* **Paràmetres i interfícies**: en ROS2 es controla l’accés al node amb get\_node\_base\_interface() per realitzar operacions dins dels bucles de control.
* **Gestió de missatges**: els missatges són tipus msg:: (per exemple, nav\_msgs::msg::Odometry), en comptes de les estructures pròpies de ROS1.
* **Arxius i directoris**: les rutes als fitxers de resultats han estat adaptades a l’entorn de ROS2 dins del robot físic.

## Resultats obtinguts en ROS2

Aquest apartat descriu els resultats experimentals obtinguts amb l'execució del node docking\_once\_ROS2\_odom.cpp sobre el robot físic de la Universitat de València, sota l'entorn ROS2 Humble.

### Condicions inicials de la prova

Abans d’iniciar el procés de docking, el robot es col·locà de forma manual a una distància aproximada de 50 cm respecte al punt de destí, amb una orientació desviada i una lleugera desalineació lateral respecte a la posició òptima davant de la placa de referència.

Aquesta situació inicial es va registrar de dues formes:

* Mitjançant **RViz**, visualitzant la posició inicial del robot dins de l’entorn simulat (Figura 28).

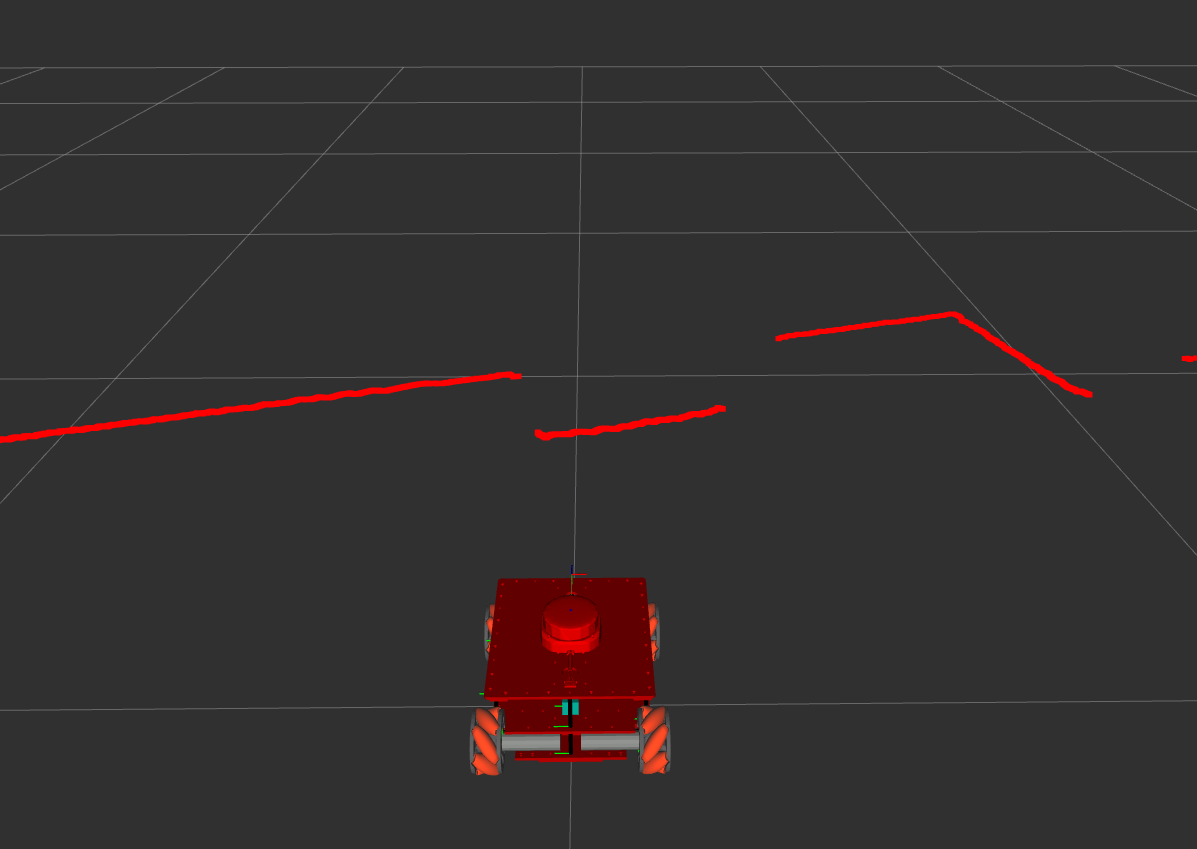


Figura 28: Rviz ROS2 Posició Inicial

* Mitjançant una fotografia real presa des de l’exterior (Figura 29).



Figura 29: Foto posició inicial

### Generació de dades i anàlisi del LiDAR

En aquesta posició inicial, es va executar el node ROS2 que processa els valors del sensor LiDAR amb 3240 feixos per volta. El node va generar automàticament dos fitxers:

* lidar\_data.txt, amb informació detallada de cada feix (angle, distància, coordenades).
* lidar\_data.csv, amb el mateix contingut estructurat per a facilitar-ne l’anàlisi posterior.

A partir d’aquestes dades, es va generar una representació gràfica en coordenades cartesianes dels punts detectats, mostrant clarament la posició i forma de la placa davant del robot (Figura 30).

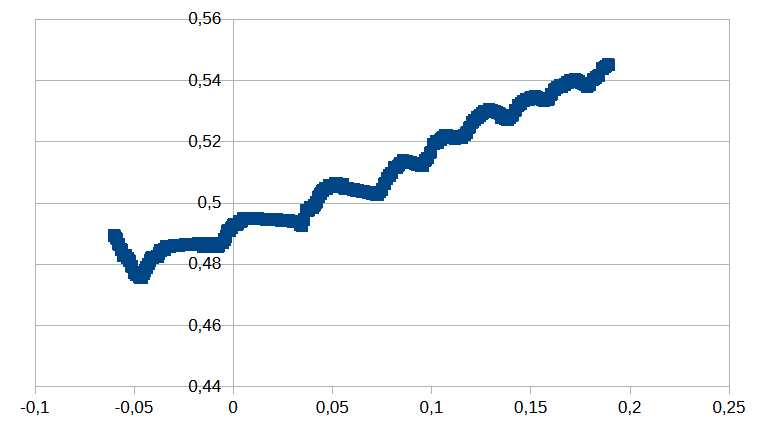


Figura 30: Representació gràfica posició inicial

### Resultat final del moviment

Un cop calculats els valors de gir inicial, desplaçament lineal (avance) i gir final, el robot va executar els moviments mitjançant control proporcional basat en odometria. El comportament va ser coherent, i el robot va aconseguir posicionar-se davant de la caixa que representa l’estació de docking.

Aquesta posició final es documenta mitjançant:

* Una captura de RViz de la posició final del robot (Figura 31).

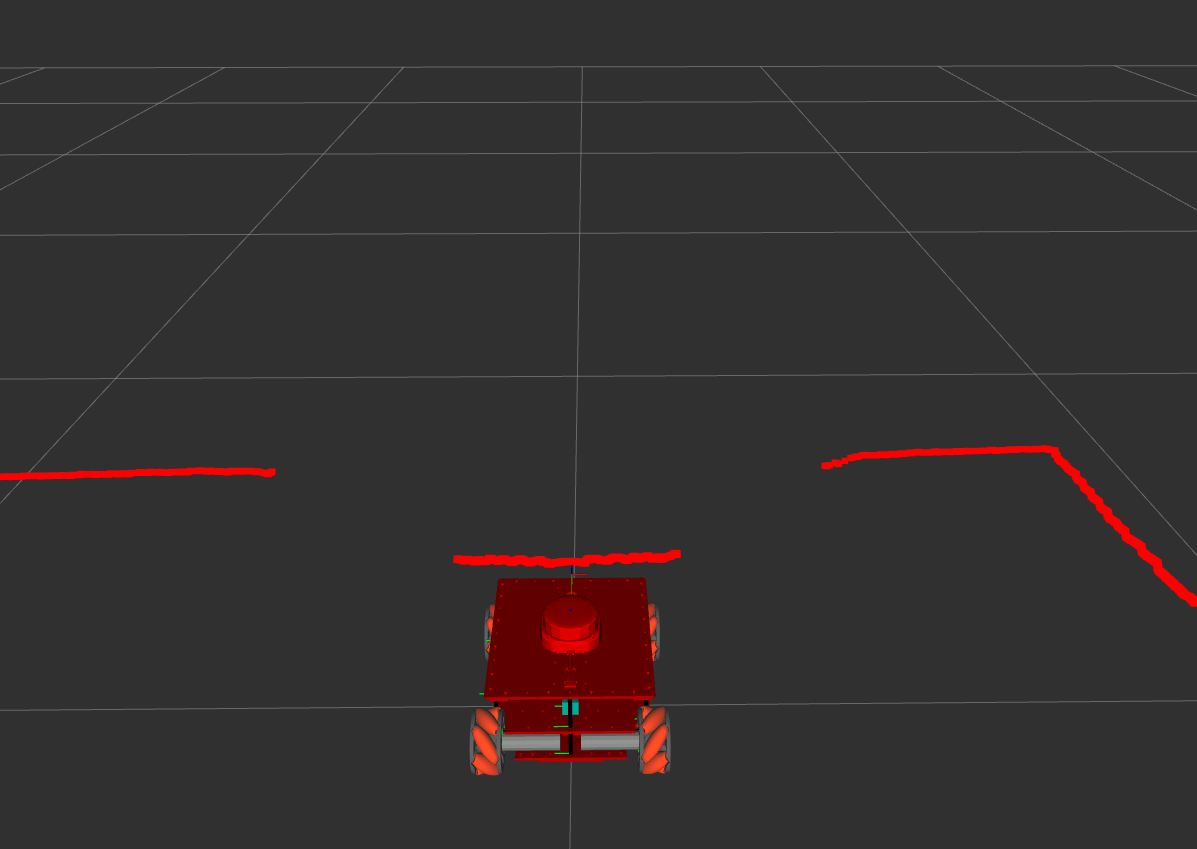


Figura 31: Rviz ROS2 Posició Final

* Una fotografia rea**l** presa al final de la prova (Figura 32).



Figura 32: Foto posició final

### Trajectòria real del robot

Tot i que no es va poder enregistrar un vídeo complet de la trajectòria (per problemes puntuals de robustesa en el codi o comportament del robot), sí que es van captar diverses fotografies intermitges al llarg de la ruta, mostrant el recorregut seguit pel robot fins al destí (Figura 33, Figura 34, Figura 35 i Figura 36).



Figura 33: Inicial



Figura 34: Posicio intermitja 1



Figura 35: Posició intermitja 2



Figura 36: Final

Aquestes imatges mostren un comportament progressivament convergent cap a la posició final, amb una correcció d’orientació i alineació realitzada automàticament durant l’execució dels tres moviments programats.

# Conclusions i Millores

Aquest treball ha permés dissenyar i implementar un sistema d'acoblament autònom (docking) per a un robot mòbil, basat en detecció làser i odometria, funcionant tant en simulació com en un robot físic amb ROS2. Els resultats mostren que, tot i les dificultats trobades, el sistema aconsegueix detectar la placa de referència i generar moviments coherents per apropar-se a ella de forma autònoma.

Entre les conclusions més destacades podem ressaltar:

* **El sistema és escalable**: el codi ha sigut adaptat per treballar amb qualsevol sensor LiDAR, llegint la mida del vector ranges[], la qual cosa el fa robust davant canvis en l'equipament hardware.
* **Migració exitosa a ROS2**: s'ha aconseguit portar tot el sistema a ROS2 Humble, incloent la subscripció i publicació de dades, càlculs geomètrics, control de moviment i registre de resultats, adaptant-se a les noves estructures com rclcpp, spin\_some o l'ús de SharedPtr.
* **Control proporcional funcional**: s'ha desenvolupat un sistema de control proporcional basat en odometria per a moviments angulars i lineals, millorant notablement la precisió en comparació amb l'enfocament inicial basat en temps.
* **Limitacions amb el robot físic**: tot i que la detecció i càlculs s'executaven correctament, **no es va poder completar una sèrie de proves amb 10 iteracions consecutives**, principalment per la manca de robustesa del sistema o inestabilitats del robot.
* **Desviació en les dades del LiDAR de ROS2**: s'ha detectat que els valors del sensor LiDAR en el robot físic (ROS2) presenten una **major desviació i menor linealitat** en comparació amb la simulació de ROS1, fet que pot dificultar la precisió del càlcul de posició i trajectòria.

## Propostes de millora futures

Per continuar millorant aquest sistema i augmentar-ne la fiabilitat, es proposen les següents línies de treball:

* **Afegir control integrador (PI)** per a reduir l’error acumulat, especialment en entorns amb soroll o fricció, cosa que millorarà l'estabilitat del moviment final.
* **Filtrat de les dades del LiDAR**, aplicant tècniques de suavitzat o regressió per reduir les desviacions detectades i obtenir una millor estimació de la posició de la placa.
* **Estudi del comportament del LiDAR físic**, comparant més a fons els valors amb els esperats segons les especificacions tècniques, i considerant alternatives de calibratge.
* **Millorar la robustesa del sistema en ROS2**, optimitzant la gestió d'errors, els paràmetres del control, i la integració amb el microcontrolador STM32 del robot.
* **Estendre el sistema per treballar en escenaris amb obstacles**, incorporant tècniques de planificació de trajectòria o navegació autònoma combinada amb detecció de la placa.

En conjunt, aquest projecte constitueix una base sòlida per al desenvolupament de sistemes autònoms d'acoblament en entorns reals, i obri les portes a una àmplia gamma d'aplicacions robòtiques futures.

# Bibliografia

Bostelman, R., Bostelman, R., & Hong, T. (2016). *Review of research for docking automatic guided vehicles and mobile robots*. US Department of Commerce, National Institute of Standards and Technology.

Feng, S., Liu, Y., Pressgrove, I., & Ben-Tzvi, P. (2024). Autonomous alignment and docking control for a self-reconfigurable modular mobile robotic system. *Robotics*, *13*(5), 81.

Jia, F., Afaq, M., Ripka, B., Huda, Q., & Ahmad, R. (2023). Vision-and lidar-based autonomous docking and recharging of a mobile robot for machine tending in autonomous manufacturing environments. *Applied Sciences*, *13*(19), 10675.

Romanov, A. M., & Tararin, A. A. (2021). An automatic docking system for wheeled mobile robots. *2021 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (ElConRus)*, 1040-1045.

# Annexos

## Annex A. Repositori de GitHub

<https://github.com/JoseCuevas93/TFM_ROS>

## Annex B. Codi del fitxer habitacion.word

<?xml version="1.0" ?>

<sdf version="1.6">

  <world name="habitacio">

    <include>

      <uri>model://sun</uri>

    </include>

    <include>

      <uri>model://ground\_plane</uri>

    </include>

    <!-- Parets de l'habitació -->

    <model name="parets">

      <static>true</static>

      <!-- Paret 1 (llarga, 6 m) -->

      <link name="paret\_1">

        <collision name="collision">

          <geometry>

            <box>

              <size>6 0.1 0.2</size>

            </box>

          </geometry>

        </collision>

        <visual name="visual">

          <geometry>

            <box>

              <size>6 0.1 0.2</size>

            </box>

          </geometry>

          <material>

            <ambient>0.8 0.8 0.8 1</ambient>

          </material>

        </visual>

        <pose>1 -2 0.101 0 0 0</pose>

      </link>

      <!-- Paret 2 (llarga, 6 m) -->

      <link name="paret\_2">

        <collision name="collision">

          <geometry>

            <box>

              <size>6 0.1 0.2</size>

            </box>

          </geometry>

        </collision>

        <visual name="visual">

          <geometry>

            <box>

              <size>6 0.1 0.2</size>

            </box>

          </geometry>

          <material>

            <ambient>0.8 0.8 0.8 1</ambient>

          </material>

        </visual>

        <pose>1 2 0.101 0 0 0</pose>

      </link>

      <!-- Paret 3 (curta, 4 m) -->

      <link name="paret\_3">

        <collision name="collision">

          <geometry>

            <box>

              <size>0.1 4 0.2</size>

            </box>

          </geometry>

        </collision>

        <visual name="visual">

          <geometry>

            <box>

              <size>0.1 4 0.2</size>

            </box>

          </geometry>

          <material>

            <ambient>0.8 0.8 0.8 1</ambient>

          </material>

        </visual>

        <pose>-2 0 0.101 0 0 0</pose>

      </link>

      <!-- Paret 4 (curta, 4 m) -->

      <link name="paret\_4">

        <collision name="collision">

          <geometry>

            <box>

              <size>0.1 4 0.2</size>

            </box>

          </geometry>

        </collision>

        <visual name="visual">

          <geometry>

            <box>

              <size>0.1 4 0.2</size>

            </box>

          </geometry>

          <material>

            <ambient>0.8 0.8 0.8 1</ambient>

          </material>

        </visual>

        <pose>4 0 0.101 0 0 0</pose>

      </link>

    </model>

    <!-- Habitació interior -->

    <model name="habitacio\_interior">

      <static>true</static>

      <!-- Paret interior 1 -->

      <link name="paret\_interior\_1">

        <visual name="visual">

          <geometry>

            <box>

              <size>1.08 0.04 0.1</size>

            </box>

          </geometry>

          <material>

            <ambient>1 0 1 1</ambient>

          </material>

        </visual>

        <pose>2.5 1.02 0 0 0 0</pose>

      </link>

      <!-- Paret interior 2 -->

      <link name="paret\_interior\_2">

        <visual name="visual">

          <geometry>

            <box>

              <size>1.08 0.04 0.1</size>

            </box>

          </geometry>

          <material>

            <ambient>1 0 1 1</ambient>

          </material>

        </visual>

        <pose>2.5 -1.02 0 0 0 0</pose>

      </link>

      <!-- Paret interior 3 -->

      <link name="paret\_interior\_3">

        <visual name="visual">

          <geometry>

            <box>

              <size>0.04 2.08 0.1</size>

            </box>

          </geometry>

          <material>

            <ambient>1 0 1 1</ambient>

          </material>

        </visual>

        <pose>1.98 0 0 0 0 0</pose>

      </link>

      <!-- Paret interior 4 -->

      <link name="paret\_interior\_4">

        <visual name="visual">

          <geometry>

            <box>

              <size>0.04 2.08 0.1</size>

            </box>

          </geometry>

          <material>

            <ambient>1 0 1 1</ambient>

          </material>

        </visual>

        <pose>3.02 0 0 0 0 0</pose>

      </link>

    </model>

    <!-- Nova placa -->

    <model name="placa">

      <static>true</static>

      <link name="placa">

        <collision name="collision">

          <geometry>

            <box>

              <size>0.005 0.5 0.2</size>

            </box>

          </geometry>

        </collision>

        <visual name="visual">

          <geometry>

            <box>

              <size>0.005 0.5 0.2</size>

            </box>

          </geometry>

          <material>

            <ambient>0.2 0.7 0.2 1</ambient>

          </material>

        </visual>

        <!-- Girar 90 graus al voltant de l'eix Z -->

        <pose>-0.5 0 0.101 0 0 0</pose>

      </link>

    </model>

  </world>

</sdf>

## Annex C. Codi docking\_once\_inicial.cpp

#include "ros/ros.h"

#include "sensor\_msgs/LaserScan.h"

#include <iostream>

#include <fstream>

#include <vector>

#include <limits>

#include <cmath> // Para cos() y sin()

#include "geometry\_msgs/Twist.h" // Para utilizar geometry\_msgs::Twist

#include <tf/tf.h>

#include <nav\_msgs/Odometry.h>

#include <gazebo\_msgs/SetModelState.h>

#include <gazebo\_msgs/ModelState.h>

#include <cstdlib>

#include <ctime>

#include <string>

// Variables globales

int angulos\_der = 0;

int angulos\_izq = 0;

double x\_extrem\_izq = 0;

double y\_extrem\_izq = 0;

double x\_extrem\_der = 0;

double y\_extrem\_der = 0;

float dist\_to\_placa = 0.15;

float giro\_inicial = 0;

float avance = 0;

float giro\_final = 0;

float offset\_lidar = 0.064; // 6.4 cm abajo del centro de giro

//bool move\_to\_origin\_done = false;

bool move\_to\_lidar\_done = false;

// Variables para almacenar los valores de odometría

float inicial\_x = 0.0, inicial\_y = 0.0, inicial\_w = 0.0;

float final\_x = 0.0, final\_y = 0.0, final\_w = 0.0;

// Variable global para almacenar el tiempo

std::string tiempo\_inicial = "";

std::string tiempo\_final = "";

// Función de callback global

void scanCallback(const sensor\_msgs::LaserScan::ConstPtr& msg) {

    //ROS\_INFO("entra lidar callback");

            if (move\_to\_lidar\_done)

        {

            return;

            }

    const int max\_angles\_left = 180;

    const int max\_angles\_right = 180;

    std::vector<std::pair<int, float>> valid\_ranges\_left;

    std::vector<std::pair<int, float>> valid\_ranges\_right;

    std::vector<float> x\_coords;

    std::vector<float> y\_coords;

    float min\_value = std::numeric\_limits<float>::max();

    // Encontrar el valor más pequeño entre los primeros 180 valores

    for (int i = 0; i < max\_angles\_left; ++i) {

        if (msg->ranges[i] < min\_value && msg->ranges[i] >= msg->range\_min && msg->ranges[i] <= msg->range\_max) {

            min\_value = msg->ranges[i];

        }

    }

    float last\_valid\_value\_left = min\_value;

    for (int i = 0; i < max\_angles\_left; ++i) {

        float current\_value = msg->ranges[i];

        if (current\_value >= msg->range\_min && current\_value <= msg->range\_max) {

            if (current\_value - last\_valid\_value\_left <= 0.1) {

                int custom\_angle = -i;

                valid\_ranges\_left.emplace\_back(custom\_angle, current\_value);

                last\_valid\_value\_left = current\_value;

            }

        }

    }

    float last\_valid\_value\_right = min\_value;

    for (int i = max\_angles\_left; i < max\_angles\_left + max\_angles\_right; ++i) {

        float current\_value = msg->ranges[i];

        if (current\_value >= msg->range\_min && current\_value <= msg->range\_max) {

            if (current\_value - last\_valid\_value\_right <= 0.1) {

                int custom\_angle = 360 - i;

                valid\_ranges\_right.emplace\_back(custom\_angle, current\_value);

                last\_valid\_value\_right = current\_value;

            }

        }

    }

    std::string output\_file\_path\_txt = "/home/jose/ros/tfm\_ws/src/tfm\_simulation/lidar\_data.txt";

    std::string output\_file\_path\_csv = "/home/jose/ros/tfm\_ws/src/tfm\_simulation/lidar\_data.csv";

    std::ofstream output\_file\_txt(output\_file\_path\_txt);

    std::ofstream output\_file\_csv(output\_file\_path\_csv);

    if (!output\_file\_txt.is\_open() || !output\_file\_csv.is\_open()) {

        ROS\_ERROR("No se ha podido abrir el archivo para escribir los datos.");

        return;  // Eliminas el ros::shutdown() para seguir con la ejecución

    }

    output\_file\_csv << "Angle (degrees), Distance (meters), X, Y\n";

    output\_file\_txt << "Valores izquierda (0 a -179 grados):\n";

    output\_file\_txt << "Angle  Modulo  Coord\_x Coord\_y:\n";

    for (const auto& range : valid\_ranges\_left) {

        float angle\_rad = range.first \* msg->angle\_increment + msg->angle\_min;

        float x = range.second \* sin(angle\_rad);

        float y = range.second \* cos(angle\_rad);

        x\_coords.push\_back(x);

        y\_coords.push\_back(y);

        output\_file\_txt << range.first << " " << range.second << " " << x << " " << y << "\n";

        output\_file\_csv << range.first << ", " << range.second << ", " << x << ", " << y << "\n";

    }

    ////

x\_extrem\_izq = x\_coords.back(); // Coordenada x del extrem esquerre de la placa

        // Escribimos los valores de derecha al archivo TXT y CSV (invertidos)

        output\_file\_txt << "\nValors dreta (1 a 180 graus):\n";

        output\_file\_txt << "Angle  Modul  Coord\_x Coord\_y:\n";

        for (auto it = valid\_ranges\_right.rbegin(); it != valid\_ranges\_right.rend(); ++it) {

            float angle\_rad = it->first \* msg->angle\_increment + msg->angle\_min;

            float x = it->second \* sin(angle\_rad); // Coordenada X en polars

            float y = it->second \* cos(angle\_rad); // Coordenada Y en polars

            x\_coords.push\_back(x);

            y\_coords.push\_back(y);

            // Escribir al archivo TXT

            output\_file\_txt << it->first << " " << it->second << " " << x << " " << y << "\n"; // Angle, distància i coordenades (x, y)

            // Escribir al archivo CSV

            output\_file\_csv << it->first << ", " << it->second << ", " << x << ", " << y << "\n"; // Angle, distància, coordenades (x, y)

        }

        x\_extrem\_der = x\_coords.back(); // Coordenada x del extrem dret de la placa

        // Guardar el primer valor de la derecha y el último valor de la izquierda en variables globales

        if (!valid\_ranges\_right.empty()) {

            angulos\_der = valid\_ranges\_right.front().first; // Primer valor de la derecha

        }

        if (!valid\_ranges\_left.empty()) {

            angulos\_izq = valid\_ranges\_left.back().first; // Último valor de la izquierda

        }

        // Calcular la pendiente (m) y la intersección (b) del segmento utilizando la fórmula

        int n = x\_coords.size();

        if (n > 1) {

            float sum\_x = 0, sum\_y = 0, sum\_x2 = 0, sum\_xy = 0;

            // Calcular las sumas necesarias

            for (int i = 0; i < n; ++i) {

                sum\_x += x\_coords[i];

                sum\_y += y\_coords[i];

                sum\_x2 += x\_coords[i] \* x\_coords[i];

                sum\_xy += x\_coords[i] \* y\_coords[i];

            }

            // Calcular la pendiente (m)

            float m = (sum\_xy - (sum\_x \* sum\_y) / n) / (sum\_x2 - (sum\_x \* sum\_x) / n);

            // Calcular la intersección (b)

            float b = (sum\_y - m \* sum\_x) / n;

            y\_extrem\_izq = m \* x\_extrem\_izq + b - offset\_lidar; // Calculo extremo con correccion offset

            y\_extrem\_der = m \* x\_extrem\_der + b - offset\_lidar; // Calculo extremo con correccion offset

            // Calcular distància entre extrems

            float distancia\_extrems = sqrt(pow(x\_extrem\_der - x\_extrem\_izq, 2) + pow(y\_extrem\_der - y\_extrem\_izq, 2));

            // Escriure resultats TXT

            output\_file\_txt << "Pendents (m), Intersecció (b), Distància entre extrems (m)\n";

            output\_file\_txt << m << ", " << b << ", " << distancia\_extrems << "\n";

            output\_file\_txt << "x\_extrem\_izq: " << x\_extrem\_izq << "\n";

            output\_file\_txt << "y\_extrem\_izq: " << y\_extrem\_izq << "\n";

            output\_file\_txt << "x\_extrem\_der: " << x\_extrem\_der << "\n";

            output\_file\_txt << "y\_extrem\_der: " << y\_extrem\_der << "\n";

        //Càlculs Trigonomètrics (dist\_to\_placa) fabs(numero);

        float Pmx = (x\_extrem\_izq + x\_extrem\_der) / 2;  //Coordenada x del punto medio entre los dos extremos placa

        float Pmy = (y\_extrem\_izq + y\_extrem\_der) / 2;  //Coordenada y del punto medio entre los dos extremos placa

        float m\_inv = -1/m; //Calcula la pendiente prependicula a la placa

        float tecta = fabs(atan(m\_inv));    //Angulo entre la pendiente inversa y eje x

        float Pdx = Pmx + (m/fabs(m)) \* dist\_to\_placa \* cos(tecta); //(m/fabs(m)) depen de la pendent de m se suma o se resta el coseno

        float Pdy = Pmy - dist\_to\_placa \* sin(fabs(tecta));     // Per a la coordenada y sempre se resta el sinus i

        float dist\_orig\_Pd = sqrt(Pdx\*Pdx + Pdy\*Pdy);       //Distancia Origen a punto Pd (distancia avance lineal)

        avance = dist\_orig\_Pd;

        float alfa = fabs(atan(Pdy/Pdx));               //Angle Pd amb eix x

        if (Pdx >= 0){ //Pd esta en el quadrant 1

            giro\_inicial = - (1.5708  - alfa); //Gir sentit horari, el valor ha de ser negatiu -(90º - alfa)

                if (m >= 0){ //Pendent placa positiva

                    giro\_final = M\_PI  - alfa - tecta;  //180º - alfa - tecta

                }

                if (m < 0){ //Pendent placa negativa

                    giro\_final = tecta - alfa;

                }

        }

        if (Pdx < 0){   //Pd esta en el quadrant 2

            giro\_inicial = M\_PI\_2  - alfa;  //Gir sentit antihorari (positiu (90º - alfa))

                if (m >= 0){ //Pendent placa positiva

                    //giro\_final = - fabs(tecta - alfa);

                    giro\_final = alfa - tecta;

                }

                if (m < 0){ //Pendent placa negativa

                    giro\_final = -(M\_PI  - alfa - tecta);   //180º - alfa - tecta

                }

        }

    // Escriure resultats de trigonometria en TXT

    output\_file\_txt << "Resultats càlculs trigonomètrics:\n";

    output\_file\_txt << "giro\_inicial (radians): " << giro\_inicial << "\n";

    output\_file\_txt << "giro\_inicial (graus): " << (giro\_inicial \* 180.0 / M\_PI) << "\n";

    output\_file\_txt << "avance: " << avance << "\n";

    output\_file\_txt << "giro\_final (radians): " << giro\_final << "\n";

    output\_file\_txt << "giro\_final (graus): " << (giro\_final \* 180.0 / M\_PI) << "\n";

    // Alfa

    output\_file\_txt << "alfa (radians): " << alfa << "\n";

    output\_file\_txt << "alfa (graus): " << (alfa \* 180.0 / M\_PI) << "\n";

    // Tecta

    output\_file\_txt << "tecta (radians): " << tecta << "\n";

    output\_file\_txt << "tecta (graus): " << (tecta \* 180.0 / M\_PI) << "\n";

    // Pdx

    output\_file\_txt << "Pdx: " << Pdx << "\n";

    // Pdy

    output\_file\_txt << "Pdy: " << Pdy << "\n";

    output\_file\_txt << "M\_PI\_2: " << M\_PI\_2 << "\n";

    output\_file\_txt << "M\_PI: " << M\_PI << "\n";

            // Escribir resultados CSV

            output\_file\_csv << "\nPendents (m), Intersecció (b), Distància entre extrems (m), x\_extrem\_izq, y\_extrem\_izq, x\_extrem\_der, y\_extrem\_der\n";

            output\_file\_csv << m << ", " << b << ", " << distancia\_extrems << ", " << x\_extrem\_izq << ", " << y\_extrem\_izq << ", " << x\_extrem\_der << ", " << y\_extrem\_der << "\n";

        ///////////

    }

    output\_file\_txt.close();

    output\_file\_csv.close();

    ROS\_INFO("Datos del LIDAR guardados en los archivos: %s y %s", output\_file\_path\_txt.c\_str(), output\_file\_path\_csv.c\_str());

    move\_to\_lidar\_done = true;

}

void lidarToFile() {

    ros::NodeHandle nh;

    // Crear el suscriptor

    ros::Subscriber scan\_sub = nh.subscribe("/scan", 10, scanCallback);

    ros::Rate rate(10); // Frecuencia de ejecución (10 Hz)

    move\_to\_lidar\_done = false;

        while (ros::ok() && !move\_to\_lidar\_done)

    {

        ros::spinOnce(); // Procesa callbacks

        rate.sleep();    // Mantiene la frecuencia

    }

}

void mover\_robot() {

    ros::NodeHandle nh;

    ros::Publisher cmd\_vel\_pub = nh.advertise<geometry\_msgs::Twist>("/cmd\_vel", 10);

    geometry\_msgs::Twist twist;

    // Constantes de velocidad (ajustables según el robot)

    float angular\_velocity = 0.2; // radianes/s

    float linear\_velocity = 0.1;  // metros/s

    ros::Duration(0.5).sleep(); // Espera 0.5 segundos antes de iniciar el movimiento

    // 1. Gira lo que valga giro\_inicial

    twist.angular.z = (giro\_inicial < 0 ? -angular\_velocity : angular\_velocity);

    twist.linear.x = 0.0;

    cmd\_vel\_pub.publish(twist);

    // Calcula el tiempo necesario para completar el giro

    double giro\_time = fabs(giro\_inicial) / angular\_velocity;

    ros::Duration(giro\_time).sleep(); // Mantén el giro durante el tiempo necesario

    // Para el giro

    twist.angular.z = 0.0;

    cmd\_vel\_pub.publish(twist);

    ros::Duration(0.5).sleep(); // Espera 0.5 segundos

    // 2. Avanza en línea recta lo que valga avance

    twist.linear.x = (avance < 0 ? -linear\_velocity : linear\_velocity);

    twist.angular.z = 0.0;  // Asegúrate de que no haya rotación mientras avanza

    cmd\_vel\_pub.publish(twist);

    // Calcula el tiempo necesario para avanzar

    double avance\_time = fabs(avance) / linear\_velocity;

    ros::Duration(avance\_time).sleep(); // Mantén el avance durante el tiempo necesario

    // Para el avance

    twist.linear.x = 0.0;

    cmd\_vel\_pub.publish(twist);

    ros::Duration(1).sleep(); // Espera 1 segundos

    // 3. Gira lo que valga giro\_final

    twist.angular.z = (giro\_final < 0 ? -angular\_velocity : angular\_velocity);

    cmd\_vel\_pub.publish(twist);

    // Calcula el tiempo necesario para completar el giro

    double giro\_final\_time = fabs(giro\_final) / angular\_velocity;

    ros::Duration(giro\_final\_time).sleep(); // Mantén el giro durante el tiempo necesario

    // Para el giro

    twist.angular.z = 0.0;

    cmd\_vel\_pub.publish(twist);

    ROS\_INFO("Finalizacion DOCKING.");

}

// Función para convertir un cuaternión a yaw

double getYawFromQuaternion(const geometry\_msgs::Quaternion& q) {

    tf::Quaternion quat(q.x, q.y, q.z, q.w);

    double roll, pitch, yaw;

    tf::Matrix3x3(quat).getRPY(roll, pitch, yaw);  // Convierte cuaternión a ángulos de Euler

    return yaw;  // Devuelve el yaw (rotación alrededor del eje Z)

}

// Función para calcular la odometría

void odom\_inicio(float& x, float& y, float& yaw) {

    ros::NodeHandle nh;

    bool is\_processed = false;

    // Callback que se ejecutará al recibir el mensaje

    auto odomCallback = [&x, &y, &yaw, &is\_processed](const nav\_msgs::Odometry::ConstPtr& msg) {

        if (!is\_processed) {

            x = msg->pose.pose.position.x;

            y = msg->pose.pose.position.y;

            // Calcular el yaw a partir del cuaternión de orientación

            yaw = getYawFromQuaternion(msg->pose.pose.orientation);

            ROS\_INFO("Odom values: position.x=%.2f, position.y=%.2f, yaw=%.4f", x, y, yaw);

            is\_processed = true; // Marcar como procesado

        }

    };

    // Crear el suscriptor al tópico de odometría

    ros::Subscriber odom\_sub = nh.subscribe<nav\_msgs::Odometry>("odom", 10, odomCallback);

    // Esperar hasta que se procese un mensaje

    ros::Rate rate(10); // Frecuencia en Hz

    while (ros::ok() && !is\_processed) {

        ros::spinOnce(); // Procesar mensajes

        rate.sleep();

    }

    // Ruta del archivo

    std::string file\_path = "/home/jose/ros/tfm\_ws/src/tfm\_simulation/resultados.txt";

    // Abrir el archivo en modo de añadir datos

    std::ofstream file(file\_path, std::ios::app);

    if (file.is\_open()) {

        file << ", los valores de odometría inicial son: "

             << "x=" << x << ", y=" << y << ", yaw=" << yaw << "\n";

        // Obtener el tiempo actual

        auto now = std::chrono::system\_clock::now();

        std::time\_t current\_time = std::chrono::system\_clock::to\_time\_t(now);

        char time\_str[100];

        std::strftime(time\_str, sizeof(time\_str), "%H:%M:%S", std::localtime(&current\_time));

        tiempo\_inicial = time\_str; // Guarda tiempo en varaible global

        //file << "Tiempo: " << time\_str << "\n";

        file.close();

        ROS\_INFO("Datos escritos en el archivo: %s", file\_path.c\_str());

    } else {

        ROS\_ERROR("No se pudo abrir el archivo: %s", file\_path.c\_str());

    }

}

// Función para calcular la odometría

void odom\_final(float& x, float& y, float& yaw) {

    ros::NodeHandle nh;

    bool is\_processed = false;

    // Callback que se ejecutará al recibir el mensaje

    auto odomCallback = [&x, &y, &yaw, &is\_processed](const nav\_msgs::Odometry::ConstPtr& msg) {

        if (!is\_processed) {

            x = msg->pose.pose.position.x;

            y = msg->pose.pose.position.y;

            // Calcular el yaw a partir del cuaternión de orientación

            yaw = getYawFromQuaternion(msg->pose.pose.orientation);

            ROS\_INFO("Odom values: position.x=%.2f, position.y=%.2f, yaw=%.4f", x, y, yaw);

            is\_processed = true; // Marcar como procesado

        }

    };

    // Crear el suscriptor al tópico de odometría

    ros::Subscriber odom\_sub = nh.subscribe<nav\_msgs::Odometry>("odom", 10, odomCallback);

    // Esperar hasta que se procese un mensaje

    ros::Rate rate(10); // Frecuencia en Hz

    while (ros::ok() && !is\_processed) {

        ros::spinOnce(); // Procesar mensajes

        rate.sleep();

    }

    // Ruta del archivo

    std::string file\_path = "/home/jose/ros/tfm\_ws/src/tfm\_simulation/resultados.txt";

    // Abrir el archivo en modo de añadir datos

    std::ofstream file(file\_path, std::ios::app);

    if (file.is\_open()) {

        file << ", los valores de odometría final son: "

             << "x=" << x << ", y=" << y << ", yaw=" << yaw << "\n";

        // Obtener el tiempo actual

        auto now = std::chrono::system\_clock::now();

        std::time\_t current\_time = std::chrono::system\_clock::to\_time\_t(now);

        char time\_str[100];

        std::strftime(time\_str, sizeof(time\_str), "%H:%M:%S", std::localtime(&current\_time));

        tiempo\_final = time\_str; // Guarda tiempo en varaible global

        //file << "Tiempo: " << time\_str << "\n";

        file.close();

        ROS\_INFO("Datos escritos en el archivo: %s", file\_path.c\_str());

    } else {

        ROS\_ERROR("No se pudo abrir el archivo: %s", file\_path.c\_str());

    }

}

// Función para convertir un string HH:MM:SS a segundos desde las 00:00:00

int convertirATiempoEnSegundos(const std::string& tiempo) {

    std::tm t = {};

    std::istringstream ss(tiempo);

    ss >> std::get\_time(&t, "%H:%M:%S");

    if (ss.fail()) {

        throw std::invalid\_argument("Formato de tiempo no válido: " + tiempo);

    }

    return t.tm\_hour \* 3600 + t.tm\_min \* 60 + t.tm\_sec;

}

// Función para convertir radianes a grados

float radiansToDegrees(float radians) {

    return radians \* (180.0 / M\_PI);

}

// Función que calcula y escribe los resultados

void calcularYEscribir() {

    // Definir la ruta del archivo como variable local

    std::string file\_path = "/home/jose/ros/tfm\_ws/src/tfm\_simulation/resultados.txt";

    // Punto de referencia

    float ref\_x = -0.35, ref\_y = 0.0;

    float ref\_w = M\_PI;

    // Calcular distancia euclidiana al punto de referencia

    float distancia = std::sqrt(std::pow(final\_x - ref\_x, 2) + std::pow(final\_y - ref\_y, 2));

    // Calcular la diferencia angular y convertir a grados

    float diferencia\_w = abs(radiansToDegrees(abs(final\_w) - ref\_w));

    // Calcular la diferencia de tiempo en segundos

    int diferencia\_tiempo = 0;

    try {

        if (!tiempo\_inicial.empty() && !tiempo\_final.empty()) {

            int tiempo\_inicial\_segundos = convertirATiempoEnSegundos(tiempo\_inicial);

            int tiempo\_final\_segundos = convertirATiempoEnSegundos(tiempo\_final);

            diferencia\_tiempo = tiempo\_final\_segundos - tiempo\_inicial\_segundos;

        }

    } catch (const std::invalid\_argument& e) {

        std::cerr << "Error al calcular la diferencia de tiempo: " << e.what() << std::endl;

    }

    // Escribir los resultados en el archivo

    std::ofstream file(file\_path, std::ios::app); // Abrir en modo de append

    if (file.is\_open()) {

        file << "Distancia al punto (-0.35, 0): " << distancia << " metros\n";

        file << "Diferencia angular con PI: " << diferencia\_w << " grados\n";

        file << "Diferencia de tiempo: " << diferencia\_tiempo << " segundos\n";

        file.close();

        std::cout << "Resultados guardados en " << file\_path << std::endl;

    } else {

        std::cerr << "No se pudo abrir el archivo " << file\_path << std::endl;

    }

}

int main(int argc, char \*\*argv) {

    ros::init(argc, argv, "docking\_loop");

    ros::start();

    ros::Rate rate(10); // Frecuencia del bucle

        // Calcular odometría inicial

        ROS\_INFO("Calcular odometría inicial");

        odom\_inicio(inicial\_x, inicial\_y, inicial\_w);

        ROS\_INFO("Pasando a datos lidar");

        ros::Duration(0.5).sleep();

        // Procesar datos del lidar y calcular movimientos Docking

        lidarToFile();

        ROS\_INFO("Pasando a mover robot");

        ros::Duration(0.5).sleep();

        // Mover el robot

        mover\_robot();

        ros::Duration(0.5).sleep();

        // Calcular odometría final

        ROS\_INFO("Calcular odometria final");

        odom\_final(final\_x, final\_y, final\_w);

        // Escribir resultados

        ROS\_INFO("Escribir resultados en archivo");

        calcularYEscribir();

    // Realiza shutdown del nodo

    ros::shutdown();

    return 0;

}

## Annex D. Codi docking.cpp

//Docking ROS1 10 iteraciones  con movimiento por tiempos

#include "ros/ros.h"

#include "sensor\_msgs/LaserScan.h"

#include <iostream>

#include <fstream>

#include <vector>

#include <limits>

#include <cmath> // Para cos() y sin()

#include "geometry\_msgs/Twist.h" // Para utilizar geometry\_msgs::Twist

#include <tf/tf.h>

#include <nav\_msgs/Odometry.h>

#include <gazebo\_msgs/SetModelState.h>

#include <gazebo\_msgs/ModelState.h>

#include <cstdlib>

#include <ctime>

#include <string>

// Variables globales

int angulos\_der = 0;

int angulos\_izq = 0;

double x\_extrem\_izq = 0;

double y\_extrem\_izq = 0;

double x\_extrem\_der = 0;

double y\_extrem\_der = 0;

float dist\_to\_placa = 0.15;

float giro\_inicial = 0;

float avance = 0;

float giro\_final = 0;

float offset\_lidar = 0.064; // 6.4 cm abajo del centro de giro

//bool move\_to\_origin\_done = false;

bool move\_to\_lidar\_done = false;

// Variables para almacenar los valores de odometría

float inicial\_x = 0.0, inicial\_y = 0.0, inicial\_w = 0.0;

float final\_x = 0.0, final\_y = 0.0, final\_w = 0.0;

// Variable global para almacenar el tiempo

std::string tiempo\_inicial = "";

std::string tiempo\_final = "";

int iteracion = 1;

// Declaración del publisher de cmd\_vel

//ros::Publisher cmd\_vel\_pub\_;

void reset\_random() {

    ros::NodeHandle nh;

    // Inicializar generador de números aleatorios

    std::srand(std::time(nullptr));

    // Generar valores aleatorios para la posición y orientación

    float random\_x = 2.0 + static\_cast<float>(std::rand()) / (static\_cast<float>(RAND\_MAX / (3.0 - 2.0)));

    float random\_y = -1.0 + static\_cast<float>(std::rand()) / (static\_cast<float>(RAND\_MAX / (1.0 - (-1.0))));

    float random\_orientation\_z = static\_cast<float>(std::rand()) / static\_cast<float>(RAND\_MAX);

    // Esperar fins que el servei estigui disponible

    ros::service::waitForService("/gazebo/set\_model\_state");

    // Crear un client per al servei

    ros::ServiceClient client = nh.serviceClient<gazebo\_msgs::SetModelState>("/gazebo/set\_model\_state");

    // Configurar l'estat del model

    gazebo\_msgs::ModelState model\_state;

    model\_state.model\_name = "turtlebot3\_waffle"; // Nom correcte del model

    model\_state.pose.position.x = random\_x;

    model\_state.pose.position.y = random\_y;

    model\_state.pose.position.z = 0.0;

    model\_state.pose.orientation.x = 0.0;

    model\_state.pose.orientation.y = 0.0;

    model\_state.pose.orientation.z = random\_orientation\_z; // Orientació aleatòria

    model\_state.pose.orientation.w = std::sqrt(1.0 - random\_orientation\_z \* random\_orientation\_z); // Normalització de quaternion

    model\_state.twist.linear.x = 0.0;

    model\_state.twist.linear.y = 0.0;

    model\_state.twist.linear.z = 0.0;

    model\_state.twist.angular.x = 0.0;

    model\_state.twist.angular.y = 0.0;

    model\_state.twist.angular.z = 0.0;

    model\_state.reference\_frame = "world";

    // Enviar la sol·licitud al servei

    gazebo\_msgs::SetModelState srv;

    srv.request.model\_state = model\_state;

    ros::Duration(1).sleep(); // Espera 1 segundo

    if (client.call(srv) && srv.response.success) {

        ROS\_INFO("Posicio restaurada a (%.2f, %.2f, 0) amb orientacio %.2f.", random\_x, random\_y, random\_orientation\_z);

    } else {

        ROS\_ERROR("No s'ha pogut restaurar la posició del TurtleBot3.");

    }

}

void move\_origin()

{

    ros::NodeHandle nh;

    ros::Publisher cmd\_vel\_pub = nh.advertise<geometry\_msgs::Twist>("/cmd\_vel", 10);

    ROS\_INFO("Comenzando movimiento a Origen");

    bool goal\_reached = false;

    bool final\_turn\_completed = false;

    bool move\_to\_origin\_done = false;

    // Función odomCallback definida dentro de move\_origin

   // auto odomCallback = [&cmd\_vel\_pub](const nav\_msgs::Odometry::ConstPtr& msg) {

        //static bool move\_to\_origin\_done = false; // Para indicar si la tarea se ha completado

        ros::Subscriber odom\_sub = nh.subscribe<nav\_msgs::Odometry>("/odom", 10, [&](const nav\_msgs::Odometry::ConstPtr& msg) {

        if (move\_to\_origin\_done)

        {

            return;

            }

        double x = msg->pose.pose.position.x;

        double y = msg->pose.pose.position.y;

        tf::Quaternion q(msg->pose.pose.orientation.x,

                         msg->pose.pose.orientation.y,

                         msg->pose.pose.orientation.z,

                         msg->pose.pose.orientation.w);

        tf::Matrix3x3 m(q);

        double roll, pitch, yaw;

        m.getRPY(roll, pitch, yaw);

        geometry\_msgs::Twist cmd\_vel;

        // Fase 1: Aproximarse al punto (0, 0)

        //static bool goal\_reached = false;

        //static bool final\_turn\_completed = false;

        //ROS\_INFO("Comenzando movimiento a Origen");

        if (!goal\_reached)

        {

            double target\_angle = atan2(-y, -x);

            double angle\_diff = target\_angle - yaw;

            if (angle\_diff > M\_PI)

                angle\_diff -= 2 \* M\_PI;

            if (angle\_diff < -M\_PI)

                angle\_diff += 2 \* M\_PI;

            if (fabs(x) > 0.1 || fabs(y) > 0.1)

            {

                if (fabs(angle\_diff) > 0.1)

                {

                    cmd\_vel.linear.x = 0.0;

                    cmd\_vel.angular.z = 0.2 \* angle\_diff;

                }

                else

                {

                    cmd\_vel.linear.x = 0.2;

                    cmd\_vel.angular.z = 0.0;

                }

            }

            else

            {

                goal\_reached = true;

                ROS\_INFO("Llegado al Origen (0, 0).");

                ros::Duration(0.5).sleep(); // Espera 0.5 segundos

            }

        }

        else if (!final\_turn\_completed)

        {

            double target\_yaw = M\_PI;

            double angle\_diff = target\_yaw - yaw;

            if (angle\_diff > M\_PI)

                angle\_diff -= 2 \* M\_PI;

            if (angle\_diff < -M\_PI)

                angle\_diff += 2 \* M\_PI;

            if (fabs(angle\_diff) > 0.2)

            {

                cmd\_vel.linear.x = 0.0;

                cmd\_vel.angular.z = 0.5 \* angle\_diff;

            }

            else

            {

                final\_turn\_completed = true;

                cmd\_vel.linear.x = 0.0;

                cmd\_vel.angular.z = 0.0;

                ROS\_INFO("Orientacion completada.");

                cmd\_vel\_pub.publish(cmd\_vel); // Detener el robot

                //ros::Duration(0.5).sleep(); // Espera 0.5 segundos

                move\_to\_origin\_done = true;

                //ros::shutdown(); // Detener el nodo si es necesario

            }

        }

        cmd\_vel\_pub.publish(cmd\_vel);

    });

    //ros::Subscriber odom\_sub = nh.subscribe<nav\_msgs::Odometry>("/odom", 10, odomCallback);

    // Usar un bucle con ros::spinOnce() para no bloquear el hilo principal

    ros::Rate rate(10); // Frecuencia de ejecución (10 Hz)

    while (ros::ok() && !move\_to\_origin\_done)

    {

        ros::spinOnce(); // Procesa callbacks

        rate.sleep();    // Mantiene la frecuencia

    }

    ROS\_INFO("Movimiento a origen completado.");

    return;

}

// Función de callback global

void scanCallback(const sensor\_msgs::LaserScan::ConstPtr& msg) {

    //ROS\_INFO("entra lidar callback");

            if (move\_to\_lidar\_done)

        {

            return;

            }

    const int max\_angles\_left = 180;

    const int max\_angles\_right = 180;

    std::vector<std::pair<int, float>> valid\_ranges\_left;

    std::vector<std::pair<int, float>> valid\_ranges\_right;

    std::vector<float> x\_coords;

    std::vector<float> y\_coords;

    float min\_value = std::numeric\_limits<float>::max();

    // Encontrar el valor más pequeño entre los primeros 180 valores

    for (int i = 0; i < max\_angles\_left; ++i) {

        if (msg->ranges[i] < min\_value && msg->ranges[i] >= msg->range\_min && msg->ranges[i] <= msg->range\_max) {

            min\_value = msg->ranges[i];

        }

    }

    float last\_valid\_value\_left = min\_value;

    for (int i = 0; i < max\_angles\_left; ++i) {

        float current\_value = msg->ranges[i];

        if (current\_value >= msg->range\_min && current\_value <= msg->range\_max) {

            if (current\_value - last\_valid\_value\_left <= 0.1) {

                int custom\_angle = -i;

                valid\_ranges\_left.emplace\_back(custom\_angle, current\_value);

                last\_valid\_value\_left = current\_value;

            }

        }

    }

    float last\_valid\_value\_right = min\_value;

    for (int i = max\_angles\_left; i < max\_angles\_left + max\_angles\_right; ++i) {

        float current\_value = msg->ranges[i];

        if (current\_value >= msg->range\_min && current\_value <= msg->range\_max) {

            if (current\_value - last\_valid\_value\_right <= 0.1) {

                int custom\_angle = 360 - i;

                valid\_ranges\_right.emplace\_back(custom\_angle, current\_value);

                last\_valid\_value\_right = current\_value;

            }

        }

    }

    //std::string output\_file\_path\_txt = "/home/jose/ros/tfm\_ws/src/tfm\_simulation/lidar\_data.txt";

    //std::string output\_file\_path\_csv = "/home/jose/ros/tfm\_ws/src/tfm\_simulation/lidar\_data.csv";

    std::string output\_file\_path\_txt = "/home/jose/ros/tfm\_ws/src/tfm\_simulation/lidar\_data\_interacion" + std::to\_string(iteracion) + ".txt";

    std::string output\_file\_path\_csv = "/home/jose/ros/tfm\_ws/src/tfm\_simulation/lidar\_data\_interacion" + std::to\_string(iteracion) + ".csv";

    std::ofstream output\_file\_txt(output\_file\_path\_txt);

    std::ofstream output\_file\_csv(output\_file\_path\_csv);

    if (!output\_file\_txt.is\_open() || !output\_file\_csv.is\_open()) {

        ROS\_ERROR("No se ha podido abrir el archivo para escribir los datos.");

        return;  // Eliminas el ros::shutdown() para seguir con la ejecución

    }

    output\_file\_csv << "Angle (degrees), Distance (meters), X, Y\n";

    output\_file\_txt << "Valores izquierda (0 a -179 grados):\n";

    output\_file\_txt << "Angle  Modulo  Coord\_x Coord\_y:\n";

    for (const auto& range : valid\_ranges\_left) {

        float angle\_rad = range.first \* msg->angle\_increment + msg->angle\_min;

        float x = range.second \* sin(angle\_rad);

        float y = range.second \* cos(angle\_rad);

        x\_coords.push\_back(x);

        y\_coords.push\_back(y);

        output\_file\_txt << range.first << " " << range.second << " " << x << " " << y << "\n";

        output\_file\_csv << range.first << ", " << range.second << ", " << x << ", " << y << "\n";

    }

    ////

x\_extrem\_izq = x\_coords.back(); // Coordenada x del extrem esquerre de la placa

        // Escribimos los valores de derecha al archivo TXT y CSV (invertidos)

        output\_file\_txt << "\nValors dreta (1 a 180 graus):\n";

        output\_file\_txt << "Angle  Modul  Coord\_x Coord\_y:\n";

        for (auto it = valid\_ranges\_right.rbegin(); it != valid\_ranges\_right.rend(); ++it) {

            float angle\_rad = it->first \* msg->angle\_increment + msg->angle\_min;

            float x = it->second \* sin(angle\_rad); // Coordenada X en polars

            float y = it->second \* cos(angle\_rad); // Coordenada Y en polars

            x\_coords.push\_back(x);

            y\_coords.push\_back(y);

            // Escribir al archivo TXT

            output\_file\_txt << it->first << " " << it->second << " " << x << " " << y << "\n"; // Angle, distància i coordenades (x, y)

            // Escribir al archivo CSV

            output\_file\_csv << it->first << ", " << it->second << ", " << x << ", " << y << "\n"; // Angle, distància, coordenades (x, y)

        }

        x\_extrem\_der = x\_coords.back(); // Coordenada x del extrem dret de la placa

        // Guardar el primer valor de la derecha y el último valor de la izquierda en variables globales

        if (!valid\_ranges\_right.empty()) {

            angulos\_der = valid\_ranges\_right.front().first; // Primer valor de la derecha

        }

        if (!valid\_ranges\_left.empty()) {

            angulos\_izq = valid\_ranges\_left.back().first; // Último valor de la izquierda

        }

        // Calcular la pendiente (m) y la intersección (b) del segmento utilizando la fórmula

        int n = x\_coords.size();

        if (n > 1) {

            float sum\_x = 0, sum\_y = 0, sum\_x2 = 0, sum\_xy = 0;

            // Calcular las sumas necesarias

            for (int i = 0; i < n; ++i) {

                sum\_x += x\_coords[i];

                sum\_y += y\_coords[i];

                sum\_x2 += x\_coords[i] \* x\_coords[i];

                sum\_xy += x\_coords[i] \* y\_coords[i];

            }

            // Calcular la pendiente (m)

            float m = (sum\_xy - (sum\_x \* sum\_y) / n) / (sum\_x2 - (sum\_x \* sum\_x) / n);

            // Calcular la intersección (b)

            float b = (sum\_y - m \* sum\_x) / n;

            y\_extrem\_izq = m \* x\_extrem\_izq + b - offset\_lidar; // Calculo extremo con correccion offset

            y\_extrem\_der = m \* x\_extrem\_der + b - offset\_lidar; // Calculo extremo con correccion offset

            // Calcular distància entre extrems

            float distancia\_extrems = sqrt(pow(x\_extrem\_der - x\_extrem\_izq, 2) + pow(y\_extrem\_der - y\_extrem\_izq, 2));

            // Escriure resultats TXT

            output\_file\_txt << "Pendents (m), Intersecció (b), Distància entre extrems (m)\n";

            output\_file\_txt << m << ", " << b << ", " << distancia\_extrems << "\n";

            output\_file\_txt << "x\_extrem\_izq: " << x\_extrem\_izq << "\n";

            output\_file\_txt << "y\_extrem\_izq: " << y\_extrem\_izq << "\n";

            output\_file\_txt << "x\_extrem\_der: " << x\_extrem\_der << "\n";

            output\_file\_txt << "y\_extrem\_der: " << y\_extrem\_der << "\n";

        //Càlculs Trigonomètrics (dist\_to\_placa) fabs(numero);

        float Pmx = (x\_extrem\_izq + x\_extrem\_der) / 2;  //Coordenada x del punto medio entre los dos extremos placa

        float Pmy = (y\_extrem\_izq + y\_extrem\_der) / 2;  //Coordenada y del punto medio entre los dos extremos placa

        float m\_inv = -1/m; //Calcula la pendiente prependicula a la placa

        float tecta = fabs(atan(m\_inv));    //Angulo entre la pendiente inversa y eje x

        float Pdx = Pmx + (m/fabs(m)) \* dist\_to\_placa \* cos(tecta); //(m/fabs(m)) depen de la pendent de m se suma o se resta el coseno

        float Pdy = Pmy - dist\_to\_placa \* sin(fabs(tecta));     // Per a la coordenada y sempre se resta el sinus i

        float dist\_orig\_Pd = sqrt(Pdx\*Pdx + Pdy\*Pdy);       //Distancia Origen a punto Pd (distancia avance lineal)

        avance = dist\_orig\_Pd;

        float alfa = fabs(atan(Pdy/Pdx));               //Angle Pd amb eix x

        if (Pdx >= 0){ //Pd esta en el quadrant 1

            giro\_inicial = - (1.5708  - alfa); //Gir sentit horari, el valor ha de ser negatiu -(90º - alfa)

                if (m >= 0){ //Pendent placa positiva

                    giro\_final = M\_PI  - alfa - tecta;  //180º - alfa - tecta

                }

                if (m < 0){ //Pendent placa negativa

                    giro\_final = tecta - alfa;

                }

        }

        if (Pdx < 0){   //Pd esta en el quadrant 2

            giro\_inicial = M\_PI\_2  - alfa;  //Gir sentit antihorari (positiu (90º - alfa))

                if (m >= 0){ //Pendent placa positiva

                    //giro\_final = - fabs(tecta - alfa);

                    giro\_final = alfa - tecta;

                }

                if (m < 0){ //Pendent placa negativa

                    giro\_final = -(M\_PI  - alfa - tecta);   //180º - alfa - tecta

                }

        }

    // Escriure resultats de trigonometria en TXT

    output\_file\_txt << "Resultats càlculs trigonomètrics:\n";

    output\_file\_txt << "giro\_inicial (radians): " << giro\_inicial << "\n";

    output\_file\_txt << "giro\_inicial (graus): " << (giro\_inicial \* 180.0 / M\_PI) << "\n";

    output\_file\_txt << "avance: " << avance << "\n";

    output\_file\_txt << "giro\_final (radians): " << giro\_final << "\n";

    output\_file\_txt << "giro\_final (graus): " << (giro\_final \* 180.0 / M\_PI) << "\n";

    // Alfa

    output\_file\_txt << "alfa (radians): " << alfa << "\n";

    output\_file\_txt << "alfa (graus): " << (alfa \* 180.0 / M\_PI) << "\n";

    // Tecta

    output\_file\_txt << "tecta (radians): " << tecta << "\n";

    output\_file\_txt << "tecta (graus): " << (tecta \* 180.0 / M\_PI) << "\n";

    // Pdx

    output\_file\_txt << "Pdx: " << Pdx << "\n";

    // Pdy

    output\_file\_txt << "Pdy: " << Pdy << "\n";

    output\_file\_txt << "M\_PI\_2: " << M\_PI\_2 << "\n";

    output\_file\_txt << "M\_PI: " << M\_PI << "\n";

            // Escribir resultados CSV

            output\_file\_csv << "\nPendents (m), Intersecció (b), Distància entre extrems (m), x\_extrem\_izq, y\_extrem\_izq, x\_extrem\_der, y\_extrem\_der\n";

            output\_file\_csv << m << ", " << b << ", " << distancia\_extrems << ", " << x\_extrem\_izq << ", " << y\_extrem\_izq << ", " << x\_extrem\_der << ", " << y\_extrem\_der << "\n";

        ///////////

    }

    output\_file\_txt.close();

    output\_file\_csv.close();

    ROS\_INFO("Datos del LIDAR guardados en los archivos: %s y %s", output\_file\_path\_txt.c\_str(), output\_file\_path\_csv.c\_str());

    move\_to\_lidar\_done = true;

}

void lidarToFile() {

    ros::NodeHandle nh;

    // Crear el suscriptor

    ros::Subscriber scan\_sub = nh.subscribe("/scan", 10, scanCallback);

    ros::Rate rate(10); // Frecuencia de ejecución (10 Hz)

    move\_to\_lidar\_done = false;

        while (ros::ok() && !move\_to\_lidar\_done)

    {

        ros::spinOnce(); // Procesa callbacks

        rate.sleep();    // Mantiene la frecuencia

    }

}

void mover\_robot() {

    ros::NodeHandle nh;

    ros::Publisher cmd\_vel\_pub = nh.advertise<geometry\_msgs::Twist>("/cmd\_vel", 10);

    geometry\_msgs::Twist twist;

    // Constantes de velocidad (ajustables según el robot)

    float angular\_velocity = 0.2; // radianes/s

    float linear\_velocity = 0.1;  // metros/s

    ros::Duration(0.5).sleep(); // Espera 0.5 segundos antes de iniciar el movimiento

    // 1. Gira lo que valga giro\_inicial

    twist.angular.z = (giro\_inicial < 0 ? -angular\_velocity : angular\_velocity);

    twist.linear.x = 0.0;

    cmd\_vel\_pub.publish(twist);

    // Calcula el tiempo necesario para completar el giro

    double giro\_time = fabs(giro\_inicial) / angular\_velocity;

    ros::Duration(giro\_time).sleep(); // Mantén el giro durante el tiempo necesario

    // Para el giro

    twist.angular.z = 0.0;

    cmd\_vel\_pub.publish(twist);

    ros::Duration(0.5).sleep(); // Espera 0.5 segundos

    // 2. Avanza en línea recta lo que valga avance

    twist.linear.x = (avance < 0 ? -linear\_velocity : linear\_velocity);

    twist.angular.z = 0.0;  // Asegúrate de que no haya rotación mientras avanza

    cmd\_vel\_pub.publish(twist);

    // Calcula el tiempo necesario para avanzar

    double avance\_time = fabs(avance) / linear\_velocity;

    ros::Duration(avance\_time).sleep(); // Mantén el avance durante el tiempo necesario

    // Para el avance

    twist.linear.x = 0.0;

    cmd\_vel\_pub.publish(twist);

    ros::Duration(1).sleep(); // Espera 1 segundos

    // 3. Gira lo que valga giro\_final

    twist.angular.z = (giro\_final < 0 ? -angular\_velocity : angular\_velocity);

    cmd\_vel\_pub.publish(twist);

    // Calcula el tiempo necesario para completar el giro

    double giro\_final\_time = fabs(giro\_final) / angular\_velocity;

    ros::Duration(giro\_final\_time).sleep(); // Mantén el giro durante el tiempo necesario

    // Para el giro

    twist.angular.z = 0.0;

    cmd\_vel\_pub.publish(twist);

    ROS\_INFO("Finalizacion DOCKING.");

}

// Función para convertir un cuaternión a yaw

double getYawFromQuaternion(const geometry\_msgs::Quaternion& q) {

    tf::Quaternion quat(q.x, q.y, q.z, q.w);

    double roll, pitch, yaw;

    tf::Matrix3x3(quat).getRPY(roll, pitch, yaw);  // Convierte cuaternión a ángulos de Euler

    return yaw;  // Devuelve el yaw (rotación alrededor del eje Z)

}

// Función para calcular la odometría

void odom\_inicio(float& x, float& y, float& yaw) {

    ros::NodeHandle nh;

    bool is\_processed = false;

    // Callback que se ejecutará al recibir el mensaje

    auto odomCallback = [&x, &y, &yaw, &is\_processed](const nav\_msgs::Odometry::ConstPtr& msg) {

        if (!is\_processed) {

            x = msg->pose.pose.position.x;

            y = msg->pose.pose.position.y;

            // Calcular el yaw a partir del cuaternión de orientación

            yaw = getYawFromQuaternion(msg->pose.pose.orientation);

            ROS\_INFO("Odom values: position.x=%.2f, position.y=%.2f, yaw=%.4f", x, y, yaw);

            is\_processed = true; // Marcar como procesado

        }

    };

    // Crear el suscriptor al tópico de odometría

    ros::Subscriber odom\_sub = nh.subscribe<nav\_msgs::Odometry>("odom", 10, odomCallback);

    // Esperar hasta que se procese un mensaje

    ros::Rate rate(10); // Frecuencia en Hz

    while (ros::ok() && !is\_processed) {

        ros::spinOnce(); // Procesar mensajes

        rate.sleep();

    }

    // Ruta del archivo

    std::string file\_path = "/home/jose/ros/tfm\_ws/src/tfm\_simulation/resultados.txt";

    // Abrir el archivo en modo de añadir datos

    std::ofstream file(file\_path, std::ios::app);

    if (file.is\_open()) {

        file << "Iteración " << iteracion << ", los valores de odometría inicial son: "

             << "x=" << x << ", y=" << y << ", yaw=" << yaw << "\n";

        // Obtener el tiempo actual

        auto now = std::chrono::system\_clock::now();

        std::time\_t current\_time = std::chrono::system\_clock::to\_time\_t(now);

        char time\_str[100];

        std::strftime(time\_str, sizeof(time\_str), "%H:%M:%S", std::localtime(&current\_time));

        tiempo\_inicial = time\_str; // Guarda tiempo en varaible global

        //file << "Tiempo: " << time\_str << "\n";

        file.close();

        ROS\_INFO("Datos escritos en el archivo: %s", file\_path.c\_str());

    } else {

        ROS\_ERROR("No se pudo abrir el archivo: %s", file\_path.c\_str());

    }

}

// Función para calcular la odometría

void odom\_final(float& x, float& y, float& yaw) {

    ros::NodeHandle nh;

    bool is\_processed = false;

    // Callback que se ejecutará al recibir el mensaje

    auto odomCallback = [&x, &y, &yaw, &is\_processed](const nav\_msgs::Odometry::ConstPtr& msg) {

        if (!is\_processed) {

            x = msg->pose.pose.position.x;

            y = msg->pose.pose.position.y;

            // Calcular el yaw a partir del cuaternión de orientación

            yaw = getYawFromQuaternion(msg->pose.pose.orientation);

            ROS\_INFO("Odom values: position.x=%.2f, position.y=%.2f, yaw=%.4f", x, y, yaw);

            is\_processed = true; // Marcar como procesado

        }

    };

    // Crear el suscriptor al tópico de odometría

    ros::Subscriber odom\_sub = nh.subscribe<nav\_msgs::Odometry>("odom", 10, odomCallback);

    // Esperar hasta que se procese un mensaje

    ros::Rate rate(10); // Frecuencia en Hz

    while (ros::ok() && !is\_processed) {

        ros::spinOnce(); // Procesar mensajes

        rate.sleep();

    }

    // Ruta del archivo

    std::string file\_path = "/home/jose/ros/tfm\_ws/src/tfm\_simulation/resultados.txt";

    // Abrir el archivo en modo de añadir datos

    std::ofstream file(file\_path, std::ios::app);

    if (file.is\_open()) {

        file << "Iteración " << iteracion << ", los valores de odometría final son: "

             << "x=" << x << ", y=" << y << ", yaw=" << yaw << "\n";

        // Obtener el tiempo actual

        auto now = std::chrono::system\_clock::now();

        std::time\_t current\_time = std::chrono::system\_clock::to\_time\_t(now);

        char time\_str[100];

        std::strftime(time\_str, sizeof(time\_str), "%H:%M:%S", std::localtime(&current\_time));

        tiempo\_final = time\_str; // Guarda tiempo en varaible global

        //file << "Tiempo: " << time\_str << "\n";

        file.close();

        ROS\_INFO("Datos escritos en el archivo: %s", file\_path.c\_str());

    } else {

        ROS\_ERROR("No se pudo abrir el archivo: %s", file\_path.c\_str());

    }

}

// Función para convertir un string HH:MM:SS a segundos desde las 00:00:00

int convertirATiempoEnSegundos(const std::string& tiempo) {

    std::tm t = {};

    std::istringstream ss(tiempo);

    ss >> std::get\_time(&t, "%H:%M:%S");

    if (ss.fail()) {

        throw std::invalid\_argument("Formato de tiempo no válido: " + tiempo);

    }

    return t.tm\_hour \* 3600 + t.tm\_min \* 60 + t.tm\_sec;

}

// Función para convertir radianes a grados

float radiansToDegrees(float radians) {

    return radians \* (180.0 / M\_PI);

}

// Función que calcula y escribe los resultados

void calcularYEscribir() {

    // Definir la ruta del archivo como variable local

    std::string file\_path = "/home/jose/ros/tfm\_ws/src/tfm\_simulation/resultados.txt";

    // Punto de referencia

    float ref\_x = -0.35, ref\_y = 0.0;

    float ref\_w = M\_PI;

    // Calcular distancia euclidiana al punto de referencia

    float distancia = std::sqrt(std::pow(final\_x - ref\_x, 2) + std::pow(final\_y - ref\_y, 2));

    // Calcular la diferencia angular y convertir a grados

    float diferencia\_w = abs(radiansToDegrees(abs(final\_w) - ref\_w));

    // Calcular la diferencia de tiempo en segundos

    int diferencia\_tiempo = 0;

    try {

        if (!tiempo\_inicial.empty() && !tiempo\_final.empty()) {

            int tiempo\_inicial\_segundos = convertirATiempoEnSegundos(tiempo\_inicial);

            int tiempo\_final\_segundos = convertirATiempoEnSegundos(tiempo\_final);

            diferencia\_tiempo = tiempo\_final\_segundos - tiempo\_inicial\_segundos;

        }

    } catch (const std::invalid\_argument& e) {

        std::cerr << "Error al calcular la diferencia de tiempo: " << e.what() << std::endl;

    }

    // Escribir los resultados en el archivo

    std::ofstream file(file\_path, std::ios::app); // Abrir en modo de append

    if (file.is\_open()) {

        file << "Distancia al punto (-0.35, 0): " << distancia << " metros\n";

        file << "Diferencia angular con PI: " << diferencia\_w << " grados\n";

        file << "Diferencia de tiempo: " << diferencia\_tiempo << " segundos\n";

        file.close();

        std::cout << "Resultados guardados en " << file\_path << std::endl;

    } else {

        std::cerr << "No se pudo abrir el archivo " << file\_path << std::endl;

    }

}

int main(int argc, char \*\*argv) {

    ros::init(argc, argv, "docking\_loop");

    ros::start();

    ros::Rate rate(10); // Frecuencia del bucle

    for (int i = 0; i < 10; ++i) {

        ROS\_INFO("Iteracion %d del bucle docking", i + 1);

        ros::Duration(0.5).sleep();

        // Resetear posición aleatoria

        ROS\_INFO("Resetear posicion aleatoria");

        reset\_random();

        ros::Duration(0.5).sleep();

        // Mover al origen

        ROS\_INFO("Mover al origen");

        move\_origin();

        // Calcular odometría inicial

        ROS\_INFO("Calcular odometría inicial");

        odom\_inicio(inicial\_x, inicial\_y, inicial\_w);

        ROS\_INFO("Pasando a datos lidar");

        ros::Duration(0.5).sleep();

        // Procesar datos del lidar y calcular movimientos Docking

        lidarToFile();

        ROS\_INFO("Pasando a mover robot");

        ros::Duration(0.5).sleep();

        // Mover el robot

        mover\_robot();

        ros::Duration(0.5).sleep();

        // Calcular odometría final

        ROS\_INFO("Calcular odometria final");

        odom\_final(final\_x, final\_y, final\_w);

        // Escribir resultados

        ROS\_INFO("Escribir resultados en archivo");

        calcularYEscribir();

        ROS\_INFO("Final de la iteracion %d", i + 1);

        iteracion++;

    }

    // Realiza shutdown del nodo

    ros::shutdown();

    return 0;

}

## Annex E. Resultados.txt

Iteración 1, los valores de odometría inicial son: x=-0.0463807, y=-0.0119563, yaw=-3.00944

Iteración 1, los valores de odometría final son: x=-0.340046, y=0.00954877, yaw=3.08057

Distancia al punto (-0.35, 0): 0.0137934 metros

Diferencia angular con PI: 3.49613 grados

Diferencia de tiempo: 10 segundos

Iteración 2, los valores de odometría inicial son: x=-0.0443807, y=0.00419941, yaw=2.98573

Iteración 2, los valores de odometría final son: x=-0.36253, y=0.013993, yaw=3.02366

Distancia al punto (-0.35, 0): 0.0187834 metros

Diferencia angular con PI: 6.75685 grados

Diferencia de tiempo: 11 segundos

Iteración 3, los valores de odometría inicial son: x=-0.0260128, y=-0.0020679, yaw=-2.95139

Iteración 3, los valores de odometría final son: x=-0.345007, y=0.00642066, yaw=3.11635

Distancia al punto (-0.35, 0): 0.00813356 metros

Diferencia angular con PI: 1.44654 grados

Diferencia de tiempo: 11 segundos

Iteración 4, los valores de odometría inicial son: x=-0.0328541, y=0.00116297, yaw=-2.98558

Iteración 4, los valores de odometría final son: x=-0.3418, y=-0.0202474, yaw=-3.10516

Distancia al punto (-0.35, 0): 0.0218447 metros

Diferencia angular con PI: 2.08729 grados

Diferencia de tiempo: 11 segundos

Iteración 5, los valores de odometría inicial son: x=0.0147662, y=0.0406801, yaw=3.05094

Iteración 5, los valores de odometría final son: x=-0.347777, y=0.0373068, yaw=3.04632

Distancia al punto (-0.35, 0): 0.037373 metros

Diferencia angular con PI: 5.45876 grados

Diferencia de tiempo: 13 segundos

Iteración 6, los valores de odometría inicial son: x=0.0775782, y=0.0206534, yaw=-3.08271

Iteración 6, los valores de odometría final son: x=-0.351136, y=-0.00374793, yaw=-3.10479

Distancia al punto (-0.35, 0): 0.00391618 metros

Diferencia angular con PI: 2.1089 grados

Diferencia de tiempo: 12 segundos

Iteración 7, los valores de odometría inicial son: x=-0.0441077, y=-0.0039013, yaw=3.13361

Iteración 7, los valores de odometría final son: x=-0.364799, y=0.00391538, yaw=2.94118

Distancia al punto (-0.35, 0): 0.0153083 metros

Diferencia angular con PI: 11.4828 grados

Diferencia de tiempo: 10 segundos

Iteración 8, los valores de odometría inicial son: x=-0.0463546, y=-0.00098869, yaw=3.14142

Iteración 8, los valores de odometría final son: x=-0.338401, y=0.0172704, yaw=3.06229

Distancia al punto (-0.35, 0): 0.0208042 metros

Diferencia angular con PI: 4.54378 grados

Diferencia de tiempo: 10 segundos

Iteración 9, los valores de odometría inicial son: x=-0.0403482, y=0.00739142, yaw=-3.11134

Iteración 9, los valores de odometría final son: x=-0.342854, y=0.00380241, yaw=3.13551

Distancia al punto (-0.35, 0): 0.00809422 metros

Diferencia angular con PI: 0.348695 grados

Diferencia de tiempo: 10 segundos

Iteración 10, los valores de odometría inicial son: x=0.070492, y=0.0477164, yaw=-2.95311

Iteración 10, los valores de odometría final son: x=-0.349903, y=0.0184313, yaw=-3.11203

Distancia al punto (-0.35, 0): 0.0184315 metros

Diferencia angular con PI: 1.69397 grados

Diferencia de tiempo: 12 segundos

## Annex F. Codi docking\_once\_lidar.cpp

#include "ros/ros.h"

#include "sensor\_msgs/LaserScan.h"

#include <iostream>

#include <fstream>

#include <vector>

#include <limits>

#include <cmath> // Para cos() y sin()

#include "geometry\_msgs/Twist.h" // Para utilizar geometry\_msgs::Twist

#include <tf/tf.h>

#include <nav\_msgs/Odometry.h>

#include <gazebo\_msgs/SetModelState.h>

#include <gazebo\_msgs/ModelState.h>

#include <cstdlib>

#include <ctime>

#include <string>

// Variables globales

int angulos\_der = 0;

int angulos\_izq = 0;

double x\_extrem\_izq = 0;

double y\_extrem\_izq = 0;

double x\_extrem\_der = 0;

double y\_extrem\_der = 0;

float dist\_to\_placa = 0.15;

float giro\_inicial = 0;

float avance = 0;

float giro\_final = 0;

float offset\_lidar = 0.064; // 6.4 cm abajo del centro de giro

//bool move\_to\_origin\_done = false;

bool move\_to\_lidar\_done = false;

// Variables para almacenar los valores de odometría

float inicial\_x = 0.0, inicial\_y = 0.0, inicial\_w = 0.0;

float final\_x = 0.0, final\_y = 0.0, final\_w = 0.0;

// Variable global para almacenar el tiempo

std::string tiempo\_inicial = "";

std::string tiempo\_final = "";

// Función de callback global

    void scanCallback(const sensor\_msgs::LaserScan::ConstPtr& msg) {

    if (move\_to\_lidar\_done) {

        return;

    }

    int num\_beams = msg->ranges.size();

    int half\_beams = num\_beams / 2;

    std::vector<std::pair<double, float>> valid\_ranges\_left;

    std::vector<std::pair<double, float>> valid\_ranges\_right;

    std::vector<float> x\_coords;

    std::vector<float> y\_coords;

    float min\_value = std::numeric\_limits<float>::max();

    // Encontrar el valor más pequeño en la primera mitad de los haces (izquierda)

    for (int i = 0; i < half\_beams; ++i) {

        if (msg->ranges[i] < min\_value && msg->ranges[i] >= msg->range\_min && msg->ranges[i] <= msg->range\_max) {

            min\_value = msg->ranges[i];

        }

    }

    float last\_valid\_value\_left = min\_value;

    for (int i = 0; i < half\_beams; ++i) {

        float current\_value = msg->ranges[i];

        if (current\_value >= msg->range\_min && current\_value <= msg->range\_max) {

            if (current\_value - last\_valid\_value\_left <= 0.1) {

                double custom\_angle = -i \* (360.0 / num\_beams);  // Mantener como double

                valid\_ranges\_left.emplace\_back(custom\_angle, current\_value);

                last\_valid\_value\_left = current\_value;

            }

        }

    }

    float last\_valid\_value\_right = min\_value;

    for (int i = half\_beams; i < num\_beams; ++i) {

        float current\_value = msg->ranges[i];

        if (current\_value >= msg->range\_min && current\_value <= msg->range\_max) {

            if (current\_value - last\_valid\_value\_right <= 0.1) {

                double custom\_angle = 360.0 - i \* (360.0 / num\_beams);  // Mantener como double

                valid\_ranges\_right.emplace\_back(custom\_angle, current\_value);

                last\_valid\_value\_right = current\_value;

            }

        }

    }

    std::string output\_file\_path\_txt = "/home/jose/ros/tfm\_ws/src/tfm\_simulation/lidar\_data.txt";

    std::string output\_file\_path\_csv = "/home/jose/ros/tfm\_ws/src/tfm\_simulation/lidar\_data.csv";

    std::ofstream output\_file\_txt(output\_file\_path\_txt);

    std::ofstream output\_file\_csv(output\_file\_path\_csv);

    if (!output\_file\_txt.is\_open() || !output\_file\_csv.is\_open()) {

        ROS\_ERROR("No se ha podido abrir el archivo para escribir los datos.");

        return;  // Eliminas el ros::shutdown() para seguir con la ejecución

    }

    output\_file\_csv << "Angle (degrees), Distance (meters), X, Y\n";

    output\_file\_txt << "Valores izquierda (0 a -179 grados):\n";

    output\_file\_txt << "Angle  Modulo  Coord\_x Coord\_y:\n";

    for (const auto& range : valid\_ranges\_left) {

        //float angle\_rad = range.first \* msg->angle\_increment + msg->angle\_min;

        float angle\_rad = range.first \* (M\_PI / 180.0); // Conversión corregida

        float x = range.second \* sin(angle\_rad);

        float y = range.second \* cos(angle\_rad);

        x\_coords.push\_back(x);

        y\_coords.push\_back(y);

        output\_file\_txt << range.first << " " << range.second << " " << x << " " << y << "\n";

        output\_file\_csv << range.first << ", " << range.second << ", " << x << ", " << y << "\n";

    }

    ////

x\_extrem\_izq = x\_coords.back(); // Coordenada x del extrem esquerre de la placa

        // Escribimos los valores de derecha al archivo TXT y CSV (invertidos)

        output\_file\_txt << "\nValors dreta (1 a 180 graus):\n";

        output\_file\_txt << "Angle  Modul  Coord\_x Coord\_y:\n";

        for (auto it = valid\_ranges\_right.rbegin(); it != valid\_ranges\_right.rend(); ++it) {

            //float angle\_rad = it->first \* msg->angle\_increment + msg->angle\_min;

            float angle\_rad = it->first \* (M\_PI / 180.0); // Conversión corregida

            float x = it->second \* sin(angle\_rad); // Coordenada X en polars

            float y = it->second \* cos(angle\_rad); // Coordenada Y en polars

            x\_coords.push\_back(x);

            y\_coords.push\_back(y);

            // Escribir al archivo TXT

            output\_file\_txt << it->first << " " << it->second << " " << x << " " << y << "\n"; // Angle, distància i coordenades (x, y)

            // Escribir al archivo CSV

            output\_file\_csv << it->first << ", " << it->second << ", " << x << ", " << y << "\n"; // Angle, distància, coordenades (x, y)

        }

        x\_extrem\_der = x\_coords.back(); // Coordenada x del extrem dret de la placa

        // Guardar el primer valor de la derecha y el último valor de la izquierda en variables globales

        if (!valid\_ranges\_right.empty()) {

            angulos\_der = valid\_ranges\_right.front().first; // Primer valor de la derecha

        }

        if (!valid\_ranges\_left.empty()) {

            angulos\_izq = valid\_ranges\_left.back().first; // Último valor de la izquierda

        }

        // Calcular la pendiente (m) y la intersección (b) del segmento utilizando la fórmula

        int n = x\_coords.size();

        if (n > 1) {

            float sum\_x = 0, sum\_y = 0, sum\_x2 = 0, sum\_xy = 0;

            // Calcular las sumas necesarias

            for (int i = 0; i < n; ++i) {

                sum\_x += x\_coords[i];

                sum\_y += y\_coords[i];

                sum\_x2 += x\_coords[i] \* x\_coords[i];

                sum\_xy += x\_coords[i] \* y\_coords[i];

            }

            // Calcular la pendiente (m)

            float m = (sum\_xy - (sum\_x \* sum\_y) / n) / (sum\_x2 - (sum\_x \* sum\_x) / n);

            // Calcular la intersección (b)

            float b = (sum\_y - m \* sum\_x) / n;

            y\_extrem\_izq = m \* x\_extrem\_izq + b - offset\_lidar; // Calculo extremo con correccion offset

            y\_extrem\_der = m \* x\_extrem\_der + b - offset\_lidar; // Calculo extremo con correccion offset

            // Calcular distància entre extrems

            float distancia\_extrems = sqrt(pow(x\_extrem\_der - x\_extrem\_izq, 2) + pow(y\_extrem\_der - y\_extrem\_izq, 2));

            // Escriure resultats TXT

            output\_file\_txt << "Pendents (m), Intersecció (b), Distància entre extrems (m)\n";

            output\_file\_txt << m << ", " << b << ", " << distancia\_extrems << "\n";

            output\_file\_txt << "x\_extrem\_izq: " << x\_extrem\_izq << "\n";

            output\_file\_txt << "y\_extrem\_izq: " << y\_extrem\_izq << "\n";

            output\_file\_txt << "x\_extrem\_der: " << x\_extrem\_der << "\n";

            output\_file\_txt << "y\_extrem\_der: " << y\_extrem\_der << "\n";

        //Càlculs Trigonomètrics (dist\_to\_placa) fabs(numero);

        float Pmx = (x\_extrem\_izq + x\_extrem\_der) / 2;  //Coordenada x del punto medio entre los dos extremos placa

        float Pmy = (y\_extrem\_izq + y\_extrem\_der) / 2;  //Coordenada y del punto medio entre los dos extremos placa

        float m\_inv = -1/m; //Calcula la pendiente prependicula a la placa

        float tecta = fabs(atan(m\_inv));    //Angulo entre la pendiente inversa y eje x

        float Pdx = Pmx + (m/fabs(m)) \* dist\_to\_placa \* cos(tecta); //(m/fabs(m)) depen de la pendent de m se suma o se resta el coseno

        float Pdy = Pmy - dist\_to\_placa \* sin(fabs(tecta));     // Per a la coordenada y sempre se resta el sinus i

        float dist\_orig\_Pd = sqrt(Pdx\*Pdx + Pdy\*Pdy);       //Distancia Origen a punto Pd (distancia avance lineal)

        avance = dist\_orig\_Pd;

        float alfa = fabs(atan(Pdy/Pdx));               //Angle Pd amb eix x

        if (Pdx >= 0){ //Pd esta en el quadrant 1

            giro\_inicial = - (1.5708  - alfa); //Gir sentit horari, el valor ha de ser negatiu -(90º - alfa)

                if (m >= 0){ //Pendent placa positiva

                    giro\_final = M\_PI  - alfa - tecta;  //180º - alfa - tecta

                }

                if (m < 0){ //Pendent placa negativa

                    giro\_final = tecta - alfa;

                }

        }

        if (Pdx < 0){   //Pd esta en el quadrant 2

            giro\_inicial = M\_PI\_2  - alfa;  //Gir sentit antihorari (positiu (90º - alfa))

                if (m >= 0){ //Pendent placa positiva

                    //giro\_final = - fabs(tecta - alfa);

                    giro\_final = alfa - tecta;

                }

                if (m < 0){ //Pendent placa negativa

                    giro\_final = -(M\_PI  - alfa - tecta);   //180º - alfa - tecta

                }

        }

    // Escriure resultats de trigonometria en TXT

    output\_file\_txt << "Resultats càlculs trigonomètrics:\n";

    output\_file\_txt << "giro\_inicial (radians): " << giro\_inicial << "\n";

    output\_file\_txt << "giro\_inicial (graus): " << (giro\_inicial \* 180.0 / M\_PI) << "\n";

    output\_file\_txt << "avance: " << avance << "\n";

    output\_file\_txt << "giro\_final (radians): " << giro\_final << "\n";

    output\_file\_txt << "giro\_final (graus): " << (giro\_final \* 180.0 / M\_PI) << "\n";

    // Alfa

    output\_file\_txt << "alfa (radians): " << alfa << "\n";

    output\_file\_txt << "alfa (graus): " << (alfa \* 180.0 / M\_PI) << "\n";

    // Tecta

    output\_file\_txt << "tecta (radians): " << tecta << "\n";

    output\_file\_txt << "tecta (graus): " << (tecta \* 180.0 / M\_PI) << "\n";

    // Pdx

    output\_file\_txt << "Pdx: " << Pdx << "\n";

    // Pdy

    output\_file\_txt << "Pdy: " << Pdy << "\n";

    output\_file\_txt << "M\_PI\_2: " << M\_PI\_2 << "\n";

    output\_file\_txt << "M\_PI: " << M\_PI << "\n";

            // Escribir resultados CSV

            output\_file\_csv << "\nPendents (m), Intersecció (b), Distància entre extrems (m), x\_extrem\_izq, y\_extrem\_izq, x\_extrem\_der, y\_extrem\_der\n";

            output\_file\_csv << m << ", " << b << ", " << distancia\_extrems << ", " << x\_extrem\_izq << ", " << y\_extrem\_izq << ", " << x\_extrem\_der << ", " << y\_extrem\_der << "\n";

        ///////////

    }

    output\_file\_txt.close();

    output\_file\_csv.close();

    ROS\_INFO("Datos del LIDAR guardados en los archivos: %s y %s", output\_file\_path\_txt.c\_str(), output\_file\_path\_csv.c\_str());

    move\_to\_lidar\_done = true;

}

void lidarToFile() {

    ros::NodeHandle nh;

    // Crear el suscriptor

    ros::Subscriber scan\_sub = nh.subscribe("/scan", 10, scanCallback);

    ros::Rate rate(10); // Frecuencia de ejecución (10 Hz)

    move\_to\_lidar\_done = false;

        while (ros::ok() && !move\_to\_lidar\_done)

    {

        ros::spinOnce(); // Procesa callbacks

        rate.sleep();    // Mantiene la frecuencia

    }

}

void mover\_robot() {

    ros::NodeHandle nh;

    ros::Publisher cmd\_vel\_pub = nh.advertise<geometry\_msgs::Twist>("/cmd\_vel", 10);

    geometry\_msgs::Twist twist;

    // Constantes de velocidad (ajustables según el robot)

    float angular\_velocity = 0.2; // radianes/s

    float linear\_velocity = 0.1;  // metros/s

    ros::Duration(0.5).sleep(); // Espera 0.5 segundos antes de iniciar el movimiento

    // 1. Gira lo que valga giro\_inicial

    twist.angular.z = (giro\_inicial < 0 ? -angular\_velocity : angular\_velocity);

    twist.linear.x = 0.0;

    cmd\_vel\_pub.publish(twist);

    // Calcula el tiempo necesario para completar el giro

    double giro\_time = fabs(giro\_inicial) / angular\_velocity;

    ros::Duration(giro\_time).sleep(); // Mantén el giro durante el tiempo necesario

    // Para el giro

    twist.angular.z = 0.0;

    cmd\_vel\_pub.publish(twist);

    ros::Duration(0.5).sleep(); // Espera 0.5 segundos

    // 2. Avanza en línea recta lo que valga avance

    twist.linear.x = (avance < 0 ? -linear\_velocity : linear\_velocity);

    twist.angular.z = 0.0;  // Asegúrate de que no haya rotación mientras avanza

    cmd\_vel\_pub.publish(twist);

    // Calcula el tiempo necesario para avanzar

    double avance\_time = fabs(avance) / linear\_velocity;

    ros::Duration(avance\_time).sleep(); // Mantén el avance durante el tiempo necesario

    // Para el avance

    twist.linear.x = 0.0;

    cmd\_vel\_pub.publish(twist);

    ros::Duration(1).sleep(); // Espera 1 segundos

    // 3. Gira lo que valga giro\_final

    twist.angular.z = (giro\_final < 0 ? -angular\_velocity : angular\_velocity);

    cmd\_vel\_pub.publish(twist);

    // Calcula el tiempo necesario para completar el giro

    double giro\_final\_time = fabs(giro\_final) / angular\_velocity;

    ros::Duration(giro\_final\_time).sleep(); // Mantén el giro durante el tiempo necesario

    // Para el giro

    twist.angular.z = 0.0;

    cmd\_vel\_pub.publish(twist);

    ROS\_INFO("Finalizacion DOCKING.");

}

// Función para convertir un cuaternión a yaw

double getYawFromQuaternion(const geometry\_msgs::Quaternion& q) {

    tf::Quaternion quat(q.x, q.y, q.z, q.w);

    double roll, pitch, yaw;

    tf::Matrix3x3(quat).getRPY(roll, pitch, yaw);  // Convierte cuaternión a ángulos de Euler

    return yaw;  // Devuelve el yaw (rotación alrededor del eje Z)

}

// Función para calcular la odometría

void odom\_inicio(float& x, float& y, float& yaw) {

    ros::NodeHandle nh;

    bool is\_processed = false;

    // Callback que se ejecutará al recibir el mensaje

    auto odomCallback = [&x, &y, &yaw, &is\_processed](const nav\_msgs::Odometry::ConstPtr& msg) {

        if (!is\_processed) {

            x = msg->pose.pose.position.x;

            y = msg->pose.pose.position.y;

            // Calcular el yaw a partir del cuaternión de orientación

            yaw = getYawFromQuaternion(msg->pose.pose.orientation);

            ROS\_INFO("Odom values: position.x=%.2f, position.y=%.2f, yaw=%.4f", x, y, yaw);

            is\_processed = true; // Marcar como procesado

        }

    };

    // Crear el suscriptor al tópico de odometría

    ros::Subscriber odom\_sub = nh.subscribe<nav\_msgs::Odometry>("odom", 10, odomCallback);

    // Esperar hasta que se procese un mensaje

    ros::Rate rate(10); // Frecuencia en Hz

    while (ros::ok() && !is\_processed) {

        ros::spinOnce(); // Procesar mensajes

        rate.sleep();

    }

    // Ruta del archivo

    std::string file\_path = "/home/jose/ros/tfm\_ws/src/tfm\_simulation/resultados.txt";

    // Abrir el archivo en modo de añadir datos

    std::ofstream file(file\_path, std::ios::app);

    if (file.is\_open()) {

        file << ", los valores de odometría inicial son: "

             << "x=" << x << ", y=" << y << ", yaw=" << yaw << "\n";

        // Obtener el tiempo actual

        auto now = std::chrono::system\_clock::now();

        std::time\_t current\_time = std::chrono::system\_clock::to\_time\_t(now);

        char time\_str[100];

        std::strftime(time\_str, sizeof(time\_str), "%H:%M:%S", std::localtime(&current\_time));

        tiempo\_inicial = time\_str; // Guarda tiempo en varaible global

        //file << "Tiempo: " << time\_str << "\n";

        file.close();

        ROS\_INFO("Datos escritos en el archivo: %s", file\_path.c\_str());

    } else {

        ROS\_ERROR("No se pudo abrir el archivo: %s", file\_path.c\_str());

    }

}

// Función para calcular la odometría

void odom\_final(float& x, float& y, float& yaw) {

    ros::NodeHandle nh;

    bool is\_processed = false;

    // Callback que se ejecutará al recibir el mensaje

    auto odomCallback = [&x, &y, &yaw, &is\_processed](const nav\_msgs::Odometry::ConstPtr& msg) {

        if (!is\_processed) {

            x = msg->pose.pose.position.x;

            y = msg->pose.pose.position.y;

            // Calcular el yaw a partir del cuaternión de orientación

            yaw = getYawFromQuaternion(msg->pose.pose.orientation);

            ROS\_INFO("Odom values: position.x=%.2f, position.y=%.2f, yaw=%.4f", x, y, yaw);

            is\_processed = true; // Marcar como procesado

        }

    };

    // Crear el suscriptor al tópico de odometría

    ros::Subscriber odom\_sub = nh.subscribe<nav\_msgs::Odometry>("odom", 10, odomCallback);

    // Esperar hasta que se procese un mensaje

    ros::Rate rate(10); // Frecuencia en Hz

    while (ros::ok() && !is\_processed) {

        ros::spinOnce(); // Procesar mensajes

        rate.sleep();

    }

    // Ruta del archivo

    std::string file\_path = "/home/jose/ros/tfm\_ws/src/tfm\_simulation/resultados.txt";

    // Abrir el archivo en modo de añadir datos

    std::ofstream file(file\_path, std::ios::app);

    if (file.is\_open()) {

        file << ", los valores de odometría final son: "

             << "x=" << x << ", y=" << y << ", yaw=" << yaw << "\n";

        // Obtener el tiempo actual

        auto now = std::chrono::system\_clock::now();

        std::time\_t current\_time = std::chrono::system\_clock::to\_time\_t(now);

        char time\_str[100];

        std::strftime(time\_str, sizeof(time\_str), "%H:%M:%S", std::localtime(&current\_time));

        tiempo\_final = time\_str; // Guarda tiempo en varaible global

        //file << "Tiempo: " << time\_str << "\n";

        file.close();

        ROS\_INFO("Datos escritos en el archivo: %s", file\_path.c\_str());

    } else {

        ROS\_ERROR("No se pudo abrir el archivo: %s", file\_path.c\_str());

    }

}

// Función para convertir un string HH:MM:SS a segundos desde las 00:00:00

int convertirATiempoEnSegundos(const std::string& tiempo) {

    std::tm t = {};

    std::istringstream ss(tiempo);

    ss >> std::get\_time(&t, "%H:%M:%S");

    if (ss.fail()) {

        throw std::invalid\_argument("Formato de tiempo no válido: " + tiempo);

    }

    return t.tm\_hour \* 3600 + t.tm\_min \* 60 + t.tm\_sec;

}

// Función para convertir radianes a grados

float radiansToDegrees(float radians) {

    return radians \* (180.0 / M\_PI);

}

// Función que calcula y escribe los resultados

void calcularYEscribir() {

    // Definir la ruta del archivo como variable local

    std::string file\_path = "/home/jose/ros/tfm\_ws/src/tfm\_simulation/resultados.txt";

    // Punto de referencia

    float ref\_x = -0.35, ref\_y = 0.0;

    float ref\_w = M\_PI;

    // Calcular distancia euclidiana al punto de referencia

    float distancia = std::sqrt(std::pow(final\_x - ref\_x, 2) + std::pow(final\_y - ref\_y, 2));

    // Calcular la diferencia angular y convertir a grados

    float diferencia\_w = abs(radiansToDegrees(abs(final\_w) - ref\_w));

    // Calcular la diferencia de tiempo en segundos

    int diferencia\_tiempo = 0;

    try {

        if (!tiempo\_inicial.empty() && !tiempo\_final.empty()) {

            int tiempo\_inicial\_segundos = convertirATiempoEnSegundos(tiempo\_inicial);

            int tiempo\_final\_segundos = convertirATiempoEnSegundos(tiempo\_final);

            diferencia\_tiempo = tiempo\_final\_segundos - tiempo\_inicial\_segundos;

        }

    } catch (const std::invalid\_argument& e) {

        std::cerr << "Error al calcular la diferencia de tiempo: " << e.what() << std::endl;

    }

    // Escribir los resultados en el archivo

    std::ofstream file(file\_path, std::ios::app); // Abrir en modo de append

    if (file.is\_open()) {

        file << "Distancia al punto (-0.35, 0): " << distancia << " metros\n";

        file << "Diferencia angular con PI: " << diferencia\_w << " grados\n";

        file << "Diferencia de tiempo: " << diferencia\_tiempo << " segundos\n";

        file.close();

        std::cout << "Resultados guardados en " << file\_path << std::endl;

    } else {

        std::cerr << "No se pudo abrir el archivo " << file\_path << std::endl;

    }

}

int main(int argc, char \*\*argv) {

    ros::init(argc, argv, "docking\_loop");

    ros::start();

    ros::Rate rate(10); // Frecuencia del bucle

        // Calcular odometría inicial

        ROS\_INFO("Calcular odometría inicial");

        odom\_inicio(inicial\_x, inicial\_y, inicial\_w);

        ROS\_INFO("Pasando a datos lidar");

        ros::Duration(0.5).sleep();

        // Procesar datos del lidar y calcular movimientos Docking

        lidarToFile();

        ROS\_INFO("Pasando a mover robot");

        ros::Duration(0.5).sleep();

        // Mover el robot

        mover\_robot();

        ros::Duration(0.5).sleep();

        // Calcular odometría final

        ROS\_INFO("Calcular odometria final");

        odom\_final(final\_x, final\_y, final\_w);

        // Escribir resultados

        ROS\_INFO("Escribir resultados en archivo");

        calcularYEscribir();

    // Realiza shutdown del nodo

    ros::shutdown();

    return 0;

}

## Annex G. Codi docking\_once\_odom.cpp

#include "ros/ros.h"

#include "sensor\_msgs/LaserScan.h"

#include <iostream>

#include <fstream>

#include <vector>

#include <limits>

#include <cmath> // Para cos() y sin()

#include "geometry\_msgs/Twist.h" // Para utilizar geometry\_msgs::Twist

#include <tf/tf.h>

#include <nav\_msgs/Odometry.h>

#include <gazebo\_msgs/SetModelState.h>

#include <gazebo\_msgs/ModelState.h>

#include <cstdlib>

#include <ctime>

#include <string>

// Variables globales

int angulos\_der = 0;

int angulos\_izq = 0;

double x\_extrem\_izq = 0;

double y\_extrem\_izq = 0;

double x\_extrem\_der = 0;

double y\_extrem\_der = 0;

float dist\_to\_placa = 0.15;

float giro\_inicial = 0;

float avance = 0;

float giro\_final = 0;

float offset\_lidar = 0.064; // 6.4 cm abajo del centro de giro

//bool move\_to\_origin\_done = false;

bool move\_to\_lidar\_done = false;

// Variables para almacenar los valores de odometría

float inicial\_x = 0.0, inicial\_y = 0.0, inicial\_w = 0.0;

float final\_x = 0.0, final\_y = 0.0, final\_w = 0.0;

// Variables globales para almacenar la odometría

float x = 0.0, y = 0.0, yaw = 0.0;

// Variable global para almacenar el tiempo

std::string tiempo\_inicial = "";

std::string tiempo\_final = "";

// Función de callback global

    void scanCallback(const sensor\_msgs::LaserScan::ConstPtr& msg) {

    if (move\_to\_lidar\_done) {

        return;

    }

    int num\_beams = msg->ranges.size();

    int half\_beams = num\_beams / 2;

    std::vector<std::pair<double, float>> valid\_ranges\_left;

    std::vector<std::pair<double, float>> valid\_ranges\_right;

    std::vector<float> x\_coords;

    std::vector<float> y\_coords;

    float min\_value = std::numeric\_limits<float>::max();

    // Encontrar el valor más pequeño en la primera mitad de los haces (izquierda)

    for (int i = 0; i < half\_beams; ++i) {

        if (msg->ranges[i] < min\_value && msg->ranges[i] >= msg->range\_min && msg->ranges[i] <= msg->range\_max) {

            min\_value = msg->ranges[i];

        }

    }

    float last\_valid\_value\_left = min\_value;

    for (int i = 0; i < half\_beams; ++i) {

        float current\_value = msg->ranges[i];

        if (current\_value >= msg->range\_min && current\_value <= msg->range\_max) {

            if (current\_value - last\_valid\_value\_left <= 0.1) {

                double custom\_angle = -i \* (360.0 / num\_beams);  // Mantener como double

                valid\_ranges\_left.emplace\_back(custom\_angle, current\_value);

                last\_valid\_value\_left = current\_value;

            }

        }

    }

    float last\_valid\_value\_right = min\_value;

    for (int i = half\_beams; i < num\_beams; ++i) {

        float current\_value = msg->ranges[i];

        if (current\_value >= msg->range\_min && current\_value <= msg->range\_max) {

            if (current\_value - last\_valid\_value\_right <= 0.1) {

                double custom\_angle = 360.0 - i \* (360.0 / num\_beams);  // Mantener como double

                valid\_ranges\_right.emplace\_back(custom\_angle, current\_value);

                last\_valid\_value\_right = current\_value;

            }

        }

    }

    std::string output\_file\_path\_txt = "/home/jose/ros/tfm\_ws/src/tfm\_simulation/lidar\_data.txt";

    std::string output\_file\_path\_csv = "/home/jose/ros/tfm\_ws/src/tfm\_simulation/lidar\_data.csv";

    std::ofstream output\_file\_txt(output\_file\_path\_txt);

    std::ofstream output\_file\_csv(output\_file\_path\_csv);

    if (!output\_file\_txt.is\_open() || !output\_file\_csv.is\_open()) {

        ROS\_ERROR("No se ha podido abrir el archivo para escribir los datos.");

        return;  // Eliminas el ros::shutdown() para seguir con la ejecución

    }

    output\_file\_csv << "Angle (degrees), Distance (meters), X, Y\n";

    output\_file\_txt << "Valores izquierda (0 a -179 grados):\n";

    output\_file\_txt << "Angle  Modulo  Coord\_x Coord\_y:\n";

    for (const auto& range : valid\_ranges\_left) {

        //float angle\_rad = range.first \* msg->angle\_increment + msg->angle\_min;

        float angle\_rad = range.first \* (M\_PI / 180.0); // Conversión corregida

        float x = range.second \* sin(angle\_rad);

        float y = range.second \* cos(angle\_rad);

        x\_coords.push\_back(x);

        y\_coords.push\_back(y);

        output\_file\_txt << range.first << " " << range.second << " " << x << " " << y << "\n";

        output\_file\_csv << range.first << ", " << range.second << ", " << x << ", " << y << "\n";

    }

    ////

x\_extrem\_izq = x\_coords.back(); // Coordenada x del extrem esquerre de la placa

        // Escribimos los valores de derecha al archivo TXT y CSV (invertidos)

        output\_file\_txt << "\nValors dreta (1 a 180 graus):\n";

        output\_file\_txt << "Angle  Modul  Coord\_x Coord\_y:\n";

        for (auto it = valid\_ranges\_right.rbegin(); it != valid\_ranges\_right.rend(); ++it) {

            //float angle\_rad = it->first \* msg->angle\_increment + msg->angle\_min;

            float angle\_rad = it->first \* (M\_PI / 180.0); // Conversión corregida

            float x = it->second \* sin(angle\_rad); // Coordenada X en polars

            float y = it->second \* cos(angle\_rad); // Coordenada Y en polars

            x\_coords.push\_back(x);

            y\_coords.push\_back(y);

            // Escribir al archivo TXT

            output\_file\_txt << it->first << " " << it->second << " " << x << " " << y << "\n"; // Angle, distància i coordenades (x, y)

            // Escribir al archivo CSV

            output\_file\_csv << it->first << ", " << it->second << ", " << x << ", " << y << "\n"; // Angle, distància, coordenades (x, y)

        }

        x\_extrem\_der = x\_coords.back(); // Coordenada x del extrem dret de la placa

        // Guardar el primer valor de la derecha y el último valor de la izquierda en variables globales

        if (!valid\_ranges\_right.empty()) {

            angulos\_der = valid\_ranges\_right.front().first; // Primer valor de la derecha

        }

        if (!valid\_ranges\_left.empty()) {

            angulos\_izq = valid\_ranges\_left.back().first; // Último valor de la izquierda

        }

        // Calcular la pendiente (m) y la intersección (b) del segmento utilizando la fórmula

        int n = x\_coords.size();

        if (n > 1) {

            float sum\_x = 0, sum\_y = 0, sum\_x2 = 0, sum\_xy = 0;

            // Calcular las sumas necesarias

            for (int i = 0; i < n; ++i) {

                sum\_x += x\_coords[i];

                sum\_y += y\_coords[i];

                sum\_x2 += x\_coords[i] \* x\_coords[i];

                sum\_xy += x\_coords[i] \* y\_coords[i];

            }

            // Calcular la pendiente (m)

            float m = (sum\_xy - (sum\_x \* sum\_y) / n) / (sum\_x2 - (sum\_x \* sum\_x) / n);

            // Calcular la intersección (b)

            float b = (sum\_y - m \* sum\_x) / n;

            y\_extrem\_izq = m \* x\_extrem\_izq + b - offset\_lidar; // Calculo extremo con correccion offset

            y\_extrem\_der = m \* x\_extrem\_der + b - offset\_lidar; // Calculo extremo con correccion offset

            // Calcular distància entre extrems

            float distancia\_extrems = sqrt(pow(x\_extrem\_der - x\_extrem\_izq, 2) + pow(y\_extrem\_der - y\_extrem\_izq, 2));

            // Escriure resultats TXT

            output\_file\_txt << "Pendents (m), Intersecció (b), Distància entre extrems (m)\n";

            output\_file\_txt << m << ", " << b << ", " << distancia\_extrems << "\n";

            output\_file\_txt << "x\_extrem\_izq: " << x\_extrem\_izq << "\n";

            output\_file\_txt << "y\_extrem\_izq: " << y\_extrem\_izq << "\n";

            output\_file\_txt << "x\_extrem\_der: " << x\_extrem\_der << "\n";

            output\_file\_txt << "y\_extrem\_der: " << y\_extrem\_der << "\n";

        //Càlculs Trigonomètrics (dist\_to\_placa) fabs(numero);

        float Pmx = (x\_extrem\_izq + x\_extrem\_der) / 2;  //Coordenada x del punto medio entre los dos extremos placa

        float Pmy = (y\_extrem\_izq + y\_extrem\_der) / 2;  //Coordenada y del punto medio entre los dos extremos placa

        float m\_inv = -1/m; //Calcula la pendiente prependicula a la placa

        float tecta = fabs(atan(m\_inv));    //Angulo entre la pendiente inversa y eje x

        float Pdx = Pmx + (m/fabs(m)) \* dist\_to\_placa \* cos(tecta); //(m/fabs(m)) depen de la pendent de m se suma o se resta el coseno

        float Pdy = Pmy - dist\_to\_placa \* sin(fabs(tecta));     // Per a la coordenada y sempre se resta el sinus i

        float dist\_orig\_Pd = sqrt(Pdx\*Pdx + Pdy\*Pdy);       //Distancia Origen a punto Pd (distancia avance lineal)

        avance = dist\_orig\_Pd;

        float alfa = fabs(atan(Pdy/Pdx));               //Angle Pd amb eix x

        if (Pdx >= 0){ //Pd esta en el quadrant 1

            giro\_inicial = - (1.5708  - alfa); //Gir sentit horari, el valor ha de ser negatiu -(90º - alfa)

                if (m >= 0){ //Pendent placa positiva

                    giro\_final = M\_PI  - alfa - tecta;  //180º - alfa - tecta

                }

                if (m < 0){ //Pendent placa negativa

                    giro\_final = tecta - alfa;

                }

        }

        if (Pdx < 0){   //Pd esta en el quadrant 2

            giro\_inicial = M\_PI\_2  - alfa;  //Gir sentit antihorari (positiu (90º - alfa))

                if (m >= 0){ //Pendent placa positiva

                    //giro\_final = - fabs(tecta - alfa);

                    giro\_final = alfa - tecta;

                }

                if (m < 0){ //Pendent placa negativa

                    giro\_final = -(M\_PI  - alfa - tecta);   //180º - alfa - tecta

                }

        }

    // Escriure resultats de trigonometria en TXT

    output\_file\_txt << "Resultats càlculs trigonomètrics:\n";

    output\_file\_txt << "giro\_inicial (radians): " << giro\_inicial << "\n";

    output\_file\_txt << "giro\_inicial (graus): " << (giro\_inicial \* 180.0 / M\_PI) << "\n";

    output\_file\_txt << "avance: " << avance << "\n";

    output\_file\_txt << "giro\_final (radians): " << giro\_final << "\n";

    output\_file\_txt << "giro\_final (graus): " << (giro\_final \* 180.0 / M\_PI) << "\n";

    // Alfa

    output\_file\_txt << "alfa (radians): " << alfa << "\n";

    output\_file\_txt << "alfa (graus): " << (alfa \* 180.0 / M\_PI) << "\n";

    // Tecta

    output\_file\_txt << "tecta (radians): " << tecta << "\n";

    output\_file\_txt << "tecta (graus): " << (tecta \* 180.0 / M\_PI) << "\n";

    // Pdx

    output\_file\_txt << "Pdx: " << Pdx << "\n";

    // Pdy

    output\_file\_txt << "Pdy: " << Pdy << "\n";

    output\_file\_txt << "M\_PI\_2: " << M\_PI\_2 << "\n";

    output\_file\_txt << "M\_PI: " << M\_PI << "\n";

            // Escribir resultados CSV

            output\_file\_csv << "\nPendents (m), Intersecció (b), Distància entre extrems (m), x\_extrem\_izq, y\_extrem\_izq, x\_extrem\_der, y\_extrem\_der\n";

            output\_file\_csv << m << ", " << b << ", " << distancia\_extrems << ", " << x\_extrem\_izq << ", " << y\_extrem\_izq << ", " << x\_extrem\_der << ", " << y\_extrem\_der << "\n";

        ///////////

    }

    output\_file\_txt.close();

    output\_file\_csv.close();

    ROS\_INFO("Datos del LIDAR guardados en los archivos: %s y %s", output\_file\_path\_txt.c\_str(), output\_file\_path\_csv.c\_str());

    move\_to\_lidar\_done = true;

}

void lidarToFile() {

    ros::NodeHandle nh;

    // Crear el suscriptor

    ros::Subscriber scan\_sub = nh.subscribe("/scan", 10, scanCallback);

    ros::Rate rate(10); // Frecuencia de ejecución (10 Hz)

    move\_to\_lidar\_done = false;

        while (ros::ok() && !move\_to\_lidar\_done)

    {

        ros::spinOnce(); // Procesa callbacks

        rate.sleep();    // Mantiene la frecuencia

    }

}

// Función para convertir un cuaternión a yaw

double getYawFromQuaternion(const geometry\_msgs::Quaternion& q) {

    tf::Quaternion quat(q.x, q.y, q.z, q.w);

    double roll, pitch, yaw;

    tf::Matrix3x3(quat).getRPY(roll, pitch, yaw);

    return yaw;

}

// Callback para la odometría (se actualiza cada vez que llega un mensaje de /odom)

void odom\_callback(const nav\_msgs::Odometry::ConstPtr& msg) {

    x = msg->pose.pose.position.x;

    y = msg->pose.pose.position.y;

    yaw = getYawFromQuaternion(msg->pose.pose.orientation);

    ROS\_INFO("Odom: x=%.2f, y=%.2f, yaw=%.4f", x, y, yaw);

}

// Función para registrar la odometría inicial

void odom\_inicio() {

    ros::NodeHandle nh;

    ros::Subscriber odom\_sub = nh.subscribe("/odom", 10, odom\_callback);

    ros::Rate rate(10); // 10 Hz

    bool is\_processed = false;

    // Esperar hasta que se reciba un mensaje de odometría

    while (ros::ok() && !is\_processed) {

        ros::spinOnce(); // Procesar mensaje de odometría

        if (x != 0.0 || y != 0.0 || yaw != 0.0) {

            is\_processed = true;

        }

        rate.sleep();

    }

    // Guardar los valores iniciales en un archivo

    std::string file\_path = "/home/jose/ros/tfm\_ws/src/tfm\_simulation/resultados.txt";

    std::ofstream file(file\_path, std::ios::app);

    if (file.is\_open()) {

        file << "Odometría inicial: x=" << x << ", y=" << y << ", yaw=" << yaw << "\n";

        // Obtener la hora actual

        auto now = std::chrono::system\_clock::now();

        std::time\_t current\_time = std::chrono::system\_clock::to\_time\_t(now);

        char time\_str[100];

        std::strftime(time\_str, sizeof(time\_str), "%H:%M:%S", std::localtime(&current\_time));

        tiempo\_inicial = time\_str;

        file << "Tiempo: " << time\_str << "\n";

        file.close();

        ROS\_INFO("Datos escritos en el archivo: %s", file\_path.c\_str());

    } else {

        ROS\_ERROR("No se pudo abrir el archivo: %s", file\_path.c\_str());

    }

}

void odom\_final() {

    ros::NodeHandle nh;

    ros::Subscriber odom\_sub = nh.subscribe("/odom", 10, odom\_callback);

    ros::Rate rate(10); // 10 Hz

    bool is\_processed = false;

    // Esperar hasta que se reciba un mensaje de odometría

    while (ros::ok() && !is\_processed) {

        ros::spinOnce(); // Procesar mensaje de odometría

        if (x != 0.0 || y != 0.0 || yaw != 0.0) {

            is\_processed = true;

        }

        rate.sleep();

    }

    // Ruta del archivo

    std::string file\_path = "/home/jose/ros/tfm\_ws/src/tfm\_simulation/resultados.txt";

    // Guardar los valores finales en un archivo

    std::ofstream file(file\_path, std::ios::app);

    if (file.is\_open()) {

        file << "Odometría final: x=" << x << ", y=" << y << ", yaw=" << yaw << "\n";

        // Obtener el tiempo actual

        auto now = std::chrono::system\_clock::now();

        std::time\_t current\_time = std::chrono::system\_clock::to\_time\_t(now);

        char time\_str[100];

        std::strftime(time\_str, sizeof(time\_str), "%H:%M:%S", std::localtime(&current\_time));

        tiempo\_final = time\_str;

        file << "Tiempo final: " << time\_str << "\n";

        file.close();

        ROS\_INFO("Datos de odometría final escritos en el archivo: %s", file\_path.c\_str());

    } else {

        ROS\_ERROR("No se pudo abrir el archivo: %s", file\_path.c\_str());

    }

}

void mover\_robot() {

    ros::NodeHandle nh;

    ros::Publisher cmd\_vel\_pub = nh.advertise<geometry\_msgs::Twist>("/cmd\_vel", 10);

    ros::Subscriber odom\_sub = nh.subscribe("/odom", 10, odom\_callback);

    geometry\_msgs::Twist twist;

    float K = 0.5; // Ganancia del control proporcional

    ROS\_INFO("Pulsa botón para enviar Twist 1");

    getchar();

    ros::Duration(0.5).sleep();

    // 1. Gira lo que valga giro\_inicial con control proporcional

    float yaw\_inicial = yaw;

    float error = 0.0;

    uint cnt = 0;

    bool converged = false;

    while (!converged && ros::ok()) {

        ros::spinOnce();

        float yaw\_actual = yaw - yaw\_inicial;

        error = giro\_inicial - yaw\_actual;

        twist.angular.z = K \* error;

        twist.linear.x = 0.0;

        cmd\_vel\_pub.publish(twist);

        ros::Duration(0.1).sleep();

        if (fabs(error) < 0.01) {

            cnt++;

            if (cnt > 10) converged = true;

        } else {

            cnt = 0;

        }

    }

    twist.angular.z = 0.0;

    cmd\_vel\_pub.publish(twist);

    ROS\_INFO("Pulsa botón para enviar Twist 2");

    getchar();

    ros::Duration(0.5).sleep();

    // 2. Avanza en línea recta con control proporcional usando distancia euclidiana

    float x\_inicial = x;

    float y\_inicial = y;

    cnt = 0;

    converged = false;

    while (!converged && ros::ok()) {

        ros::spinOnce();

        float distancia\_actual = sqrt(pow(x - x\_inicial, 2) + pow(y - y\_inicial, 2));

        float error\_dist = avance - distancia\_actual;

        twist.linear.x = K \* error\_dist;

        twist.angular.z = 0.0;

        cmd\_vel\_pub.publish(twist);

        ros::Duration(0.1).sleep();

        if (fabs(error\_dist) < 0.01) {

            cnt++;

            if (cnt > 10) converged = true;

        } else {

            cnt = 0;

        }

    }

    twist.linear.x = 0.0;

    cmd\_vel\_pub.publish(twist);

    ROS\_INFO("Pulsa botón para enviar Twist 3");

    getchar();

    ros::Duration(1).sleep();

    // 3. Gira lo que valga giro\_final con control proporcional

    yaw\_inicial = yaw;

    error = giro\_final;

    cnt = 0;

    converged = false;

    while (!converged && ros::ok()) {

        ros::spinOnce();

        float yaw\_actual = yaw - yaw\_inicial;

        error = giro\_final - yaw\_actual;

        twist.angular.z = K \* error;

        twist.linear.x = 0.0;

        cmd\_vel\_pub.publish(twist);

        ros::Duration(0.1).sleep();

        if (fabs(error) < 0.01) {

            cnt++;

            if (cnt > 10) converged = true;

        } else {

            cnt = 0;

        }

    }

    twist.angular.z = 0.0;

    cmd\_vel\_pub.publish(twist);

    ROS\_INFO("Finalización DOCKING.");

}

// Función para convertir un string HH:MM:SS a segundos desde las 00:00:00

int convertirATiempoEnSegundos(const std::string& tiempo) {

    std::tm t = {};

    std::istringstream ss(tiempo);

    ss >> std::get\_time(&t, "%H:%M:%S");

    if (ss.fail()) {

        throw std::invalid\_argument("Formato de tiempo no válido: " + tiempo);

    }

    return t.tm\_hour \* 3600 + t.tm\_min \* 60 + t.tm\_sec;

}

// Función para convertir radianes a grados

float radiansToDegrees(float radians) {

    return radians \* (180.0 / M\_PI);

}

// Función que calcula y escribe los resultados

void calcularYEscribir() {

    // Definir la ruta del archivo como variable local

    std::string file\_path = "/home/jose/ros/tfm\_ws/src/tfm\_simulation/resultados.txt";

    // Punto de referencia

    float ref\_x = -0.35, ref\_y = 0.0;

    float ref\_w = M\_PI;

    // Calcular distancia euclidiana al punto de referencia

    float distancia = std::sqrt(std::pow(final\_x - ref\_x, 2) + std::pow(final\_y - ref\_y, 2));

    // Calcular la diferencia angular y convertir a grados

    float diferencia\_w = abs(radiansToDegrees(abs(final\_w) - ref\_w));

    // Calcular la diferencia de tiempo en segundos

    int diferencia\_tiempo = 0;

    try {

        if (!tiempo\_inicial.empty() && !tiempo\_final.empty()) {

            int tiempo\_inicial\_segundos = convertirATiempoEnSegundos(tiempo\_inicial);

            int tiempo\_final\_segundos = convertirATiempoEnSegundos(tiempo\_final);

            diferencia\_tiempo = tiempo\_final\_segundos - tiempo\_inicial\_segundos;

        }

    } catch (const std::invalid\_argument& e) {

        std::cerr << "Error al calcular la diferencia de tiempo: " << e.what() << std::endl;

    }

    // Escribir los resultados en el archivo

    std::ofstream file(file\_path, std::ios::app); // Abrir en modo de append

    if (file.is\_open()) {

        file << "Distancia al punto (-0.35, 0): " << distancia << " metros\n";

        file << "Diferencia angular con PI: " << diferencia\_w << " grados\n";

        file << "Diferencia de tiempo: " << diferencia\_tiempo << " segundos\n";

        file.close();

        std::cout << "Resultados guardados en " << file\_path << std::endl;

    } else {

        std::cerr << "No se pudo abrir el archivo " << file\_path << std::endl;

    }

}

int main(int argc, char \*\*argv) {

    ros::init(argc, argv, "docking\_loop");  // Inicializa el nodo ROS

    ros::NodeHandle nh;

    ros::Rate rate(10); // Frecuencia de 10 Hz

    // Calcular odometría inicial

    ROS\_INFO("Calculando odometría inicial...");

    odom\_inicio();

    ros::Duration(0.5).sleep();

    // Procesar datos del LIDAR

    ROS\_INFO("Procesando datos del LIDAR...");

    lidarToFile();

    ros::Duration(0.5).sleep();

    // Mover el robot

    ROS\_INFO("Moviendo el robot...");

    mover\_robot();

    ros::Duration(0.5).sleep();

    // Calcular odometría final

    ROS\_INFO("Calculando odometría final...");

    odom\_final();

    ros::Duration(0.5).sleep();

    // Escribir resultados

    ROS\_INFO("Escribiendo resultados en archivo...");

    calcularYEscribir();

    // Finaliza el nodo ROS

    ros::shutdown();

    return 0;

}

## Annex H. Codi docking\_once\_ROS2\_odom.cpp

//#include "ros/ros.h"

//#include "sensor\_msgs/LaserScan.h"

//#include "geometry\_msgs/Twist.h" // Para utilizar geometry\_msgs::Twist

//#include <nav\_msgs/Odometry.h>

//#include <gazebo\_msgs/SetModelState.h>

//#include <gazebo\_msgs/ModelState.h>

//#include <tf/tf.h>

//#include <tf2/LinearMath/Quaternion.h>

#include <tf2\_geometry\_msgs/tf2\_geometry\_msgs.hpp>

#include <iostream>

#include <fstream>

#include <vector>

#include <limits>

#include <cmath> // Para cos() y sin()

#include <cstdlib>

#include <ctime>

#include <string>

#include <math.h>

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Necesario en ROS2

#include <chrono>

#include <functional>

#include <memory>

//#include <string>

#include "rclcpp/rclcpp.hpp"

#include "std\_msgs/msg/string.hpp"

#include "sensor\_msgs/msg/laser\_scan.hpp"

#include "geometry\_msgs/msg/twist.hpp"

#include "geometry\_msgs/msg/quaternion.hpp"

#include "nav\_msgs/msg/odometry.hpp"

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

using std::placeholders::\_1;

using namespace std::chrono\_literals;

/\* This example creates a subclass of Node and uses std::bind() to register a

\* member function as a callback from the timer. \*/

class Docking : public rclcpp::Node

{

  public:

    Docking()

    : Node("docking"), count\_(0)

    {

      is\_processed = false;

      //move\_to\_origin\_done = false;

      move\_to\_lidar\_done = false;

      sub\_odom\_ = this->create\_subscription<nav\_msgs::msg::Odometry>("/robot02/odom", 10, std::bind(&Docking::odom\_callback, this, \_1));

      sub\_laser\_ = this->create\_subscription<sensor\_msgs::msg::LaserScan>("/scan", 10, std::bind(&Docking::scan\_callback, this, \_1));

      pub\_cmd\_vel\_ = this->create\_publisher<geometry\_msgs::msg::Twist>("/cmd\_vel", 10);

      //publisher\_ = this->create\_publisher<std\_msgs::msg::String>("/data", 10);

      //timer\_ = this->create\_wall\_timer(500ms, std::bind(&Docking::timer\_callback, this));

      //this->docking();

    }

    void docking()

    {

        //rate = node.create\_rate(2);

      rclcpp::sleep\_for(std::chrono::milliseconds(500));

        // Calcular odometria inicial

        /\*while(!is\_processed){

            RCLCPP\_INFO(this->get\_logger(), ".");

            //ros::Duration(0.5).sleep();

            //rate.sleep()

            rclcpp::sleep\_for(std::chrono::milliseconds(500));

        }\*/

        RCLCPP\_INFO(this->get\_logger(), "Calcular odometría inicial: position.x=%.2f, position.y=%.2f, yaw=%.4f", x, y, yaw);

        odom\_inicio(x, y, yaw);

        // Espera de 0.5 segundos

        //ros::Duration(0.5).sleep();

        //rate.sleep()

        rclcpp::sleep\_for(std::chrono::milliseconds(500));

        // Procesar datos del lidar y calcular movimientos Docking

        RCLCPP\_INFO(this->get\_logger(), "Pasando a datos lidar");

        /\*while(!move\_to\_lidar\_done){

            //ros::Duration(0.5).sleep();

            //rate.sleep()

            rclcpp::sleep\_for(std::chrono::milliseconds(500));

        }\*/

        lidarToFile(scan\_);

        // Espera de 0.5 segundos

        //ros::Duration(0.5).sleep();

        //rate.sleep()

        rclcpp::sleep\_for(std::chrono::milliseconds(500));

        // Mover el robot

        RCLCPP\_INFO(this->get\_logger(), "Pasando a mover robot");

        mover\_robot();

        // Espera de 0.5 segundos

        //ros::Duration(0.5).sleep();

        //rate.sleep()

        rclcpp::sleep\_for(std::chrono::milliseconds(500));

        // Calcular odometria final

        RCLCPP\_INFO(this->get\_logger(), "Calcular odometría final: position.x=%.2f, position.y=%.2f, yaw=%.4f", x, y, yaw);

        odom\_final(x, y, yaw);

        // Escribir resultados

        RCLCPP\_INFO(this->get\_logger(), "Escribir resultados en archivo");

        calcularYEscribir();

    }

    bool is\_processed;

    //bool move\_to\_origin\_done;

    bool move\_to\_lidar\_done;

  private:

    void odom\_callback(const nav\_msgs::msg::Odometry::SharedPtr msg) //const

    {

      if (!is\_processed) {

          is\_processed = true; // Marcar como procesado

      }

      odom\_ = \*msg;

      x = msg->pose.pose.position.x;

      y = msg->pose.pose.position.y;

      yaw = getYawFromQuaternion(msg->pose.pose.orientation);

      RCLCPP\_INFO(this->get\_logger(), "Odom values: position.x=%.2f, position.y=%.2f, yaw=%.4f", x, y, yaw);

    }

    void scan\_callback(const sensor\_msgs::msg::LaserScan::SharedPtr msg) //const

    {

        if (!move\_to\_lidar\_done) {

            move\_to\_lidar\_done = true; // Marcar como procesado

        }

        scan\_ = \*msg;

        RCLCPP\_INFO(this->get\_logger(), "LaserScan values: '%f %d'", scan\_.ranges[0], scan\_.ranges.size());

    }

    //void timer\_callback()

    //{

    //  auto message = std\_msgs::msg::String();

    //  message.data = "Hello, world! " + std::to\_string(count\_++);

    //  RCLCPP\_INFO(this->get\_logger(), "Publishing: '%s'", message.data.c\_str());

    //  publisher\_->publish(message);

    //}

    void lidarToFile(sensor\_msgs::msg::LaserScan &msg) {

        //const int max\_angles\_left = 1620; //3240

        //const int max\_angles\_right = 1620;

        int num\_beams = msg.ranges.size();

        int half\_beams = num\_beams / 2;

        std::vector<std::pair<double, float>> valid\_ranges\_left;

        std::vector<std::pair<double, float>> valid\_ranges\_right;

        std::vector<float> x\_coords;

        std::vector<float> y\_coords;

        float min\_value = std::numeric\_limits<float>::max();

        // Encontrar el valor más pequeño entre los primeros 180 valores

        //for (int i = 0; i < max\_angles\_left; ++i) {

        for (int i = 0; i < half\_beams; ++i) {

            if (msg.ranges[i] < min\_value && msg.ranges[i] >= msg.range\_min && msg.ranges[i] <= msg.range\_max) {

                min\_value = msg.ranges[i];

            }

        }

        float last\_valid\_value\_left = min\_value;

        //for (int i = 0; i < max\_angles\_left; ++i) {

        for (int i = 0; i < half\_beams; ++i) {

            float current\_value = msg.ranges[i];

            if (current\_value >= msg.range\_min && current\_value <= msg.range\_max) {

                if (current\_value - last\_valid\_value\_left <= 0.1) {

                    //float custom\_angle = -float(i)\*180.0/max\_angles\_left;

                    double custom\_angle = -i \* (360.0 / num\_beams);

                    valid\_ranges\_left.emplace\_back(custom\_angle, current\_value);

                    last\_valid\_value\_left = current\_value;

                }

            }

        }

        float last\_valid\_value\_right = min\_value;

        //for (int i = max\_angles\_left; i < max\_angles\_left + max\_angles\_right; ++i) {

        for (int i = half\_beams; i < num\_beams; ++i) {

            float current\_value = msg.ranges[i];

            if (current\_value >= msg.range\_min && current\_value <= msg.range\_max) {

                if (current\_value - last\_valid\_value\_right <= 0.1) {

                    //float custom\_angle = 360.0 - float(i)\*180.0/max\_angles\_left;

                    double custom\_angle = 360.0 - i \* (360.0 / num\_beams);

                    valid\_ranges\_right.emplace\_back(custom\_angle, current\_value);

                    last\_valid\_value\_right = current\_value;

                }

            }

        }

        std::string output\_file\_path\_txt = "/home/robot/ros/docking\_ws/src/docking/lidar\_data.txt";

        std::string output\_file\_path\_csv = "/home/robot/ros/docking\_ws/src/docking/lidar\_data.csv";

        std::ofstream output\_file\_txt(output\_file\_path\_txt);

        std::ofstream output\_file\_csv(output\_file\_path\_csv);

        if (!output\_file\_txt.is\_open() || !output\_file\_csv.is\_open()) {

            RCLCPP\_ERROR(this->get\_logger(), "No se ha podido abrir el archivo para escribir los datos.");

            return;  // Eliminas el ros::shutdown() para seguir con la ejecución

        }

        output\_file\_csv << "Angle (degrees), Distance (meters), X, Y\n";

        output\_file\_txt << "Valores izquierda (0 a -179 grados):\n";

        output\_file\_txt << "Angle  Modulo  Coord\_x Coord\_y:\n";

        for (const auto& range : valid\_ranges\_left) {

            //float angle\_rad = range.first \* msg.angle\_increment + msg.angle\_min;

            float angle\_rad = degreesToRadians(range.first);// + msg.angle\_min;

            float x = range.second \* sin(angle\_rad);

            float y = range.second \* cos(angle\_rad);

            x\_coords.push\_back(x);

            y\_coords.push\_back(y);

            output\_file\_txt << range.first << " " << range.second << " " << x << " " << y << "\n";

            output\_file\_csv << range.first << ", " << range.second << ", " << x << ", " << y << "\n";

        }

        ////

        x\_extrem\_izq = x\_coords.back(); // Coordenada x del extrem esquerre de la placa

        // Escribimos los valores de derecha al archivo TXT y CSV (invertidos)

        output\_file\_txt << "\nValors dreta (1 a 180 graus):\n";

        output\_file\_txt << "Angle  Modul  Coord\_x Coord\_y:\n";

        for (auto it = valid\_ranges\_right.rbegin(); it != valid\_ranges\_right.rend(); ++it) {

            //float angle\_rad = it->first \* msg.angle\_increment + msg.angle\_min;

            float angle\_rad = degreesToRadians(it->first);// + msg.angle\_min;

            float x = it->second \* sin(angle\_rad); // Coordenada X en polars

            float y = it->second \* cos(angle\_rad); // Coordenada Y en polars

            x\_coords.push\_back(x);

            y\_coords.push\_back(y);

            // Escribir al archivo TXT

            output\_file\_txt << it->first << " " << it->second << " " << x << " " << y << "\n"; // Angle, distància i coordenades (x, y)

            // Escribir al archivo CSV

            output\_file\_csv << it->first << ", " << it->second << ", " << x << ", " << y << "\n"; // Angle, distància, coordenades (x, y)

        }

        x\_extrem\_der = x\_coords.back(); // Coordenada x del extrem dret de la placa

        // Guardar el primer valor de la derecha y el último valor de la izquierda en variables globales

        if (!valid\_ranges\_right.empty()) {

            angulos\_der = valid\_ranges\_right.front().first; // Primer valor de la derecha

        }

        if (!valid\_ranges\_left.empty()) {

            angulos\_izq = valid\_ranges\_left.back().first; // Último valor de la izquierda

        }

        // Calcular la pendiente (m) y la intersección (b) del segmento utilizando la fórmula

        int n = x\_coords.size();

        if (n > 1) {

            float sum\_x = 0, sum\_y = 0, sum\_x2 = 0, sum\_xy = 0;

            // Calcular las sumas necesarias

            for (int i = 0; i < n; ++i) {

                sum\_x += x\_coords[i];

                sum\_y += y\_coords[i];

                sum\_x2 += x\_coords[i] \* x\_coords[i];

                sum\_xy += x\_coords[i] \* y\_coords[i];

            }

            // Calcular la pendiente (m)

            float m = (sum\_xy - (sum\_x \* sum\_y) / n) / (sum\_x2 - (sum\_x \* sum\_x) / n);

            // Calcular la intersección (b)

            float b = (sum\_y - m \* sum\_x) / n;

            y\_extrem\_izq = m \* x\_extrem\_izq + b - offset\_lidar; // Calculo extremo con correccion offset

            y\_extrem\_der = m \* x\_extrem\_der + b - offset\_lidar; // Calculo extremo con correccion offset

            // Calcular distància entre extrems

            float distancia\_extrems = sqrt(pow(x\_extrem\_der - x\_extrem\_izq, 2) + pow(y\_extrem\_der - y\_extrem\_izq, 2));

            // Escriure resultats TXT

            output\_file\_txt << "Pendents (m), Intersecció (b), Distància entre extrems (m)\n";

            output\_file\_txt << m << ", " << b << ", " << distancia\_extrems << "\n";

            output\_file\_txt << "x\_extrem\_izq: " << x\_extrem\_izq << "\n";

            output\_file\_txt << "y\_extrem\_izq: " << y\_extrem\_izq << "\n";

            output\_file\_txt << "x\_extrem\_der: " << x\_extrem\_der << "\n";

            output\_file\_txt << "y\_extrem\_der: " << y\_extrem\_der << "\n";

            //Càlculs Trigonomètrics (dist\_to\_placa) fabs(numero);

            float Pmx = (x\_extrem\_izq + x\_extrem\_der) / 2;  //Coordenada x del punto medio entre los dos extremos placa

            float Pmy = (y\_extrem\_izq + y\_extrem\_der) / 2;  //Coordenada y del punto medio entre los dos extremos placa

            float m\_inv = -1/m; //Calcula la pendiente prependicula a la placa

            float tecta = fabs(atan(m\_inv));    //Angulo entre la pendiente inversa y eje x

            float Pdx = Pmx + (m/fabs(m)) \* dist\_to\_placa \* cos(tecta); //(m/fabs(m)) depen de la pendent de m se suma o se resta el coseno

            float Pdy = Pmy - dist\_to\_placa \* sin(fabs(tecta));     // Per a la coordenada y sempre se resta el sinus i

            float dist\_orig\_Pd = sqrt(Pdx\*Pdx + Pdy\*Pdy);       //Distancia Origen a punto Pd (distancia avance lineal)

            avance = dist\_orig\_Pd;

            float alfa = fabs(atan(Pdy/Pdx));               //Angle Pd amb eix x

            if (Pdx >= 0){ //Pd esta en el quadrant 1

                giro\_inicial = - (1.5708  - alfa); //Gir sentit horari, el valor ha de ser negatiu -(90º - alfa)

                    if (m >= 0){ //Pendent placa positiva

                        giro\_final = M\_PI  - alfa - tecta;  //180º - alfa - tecta

                    }

                    if (m < 0){ //Pendent placa negativa

                        giro\_final = tecta - alfa;

                    }

            }

            if (Pdx < 0){   //Pd esta en el quadrant 2

                giro\_inicial = M\_PI\_2  - alfa;  //Gir sentit antihorari (positiu (90º - alfa))

                    if (m >= 0){ //Pendent placa positiva

                        //giro\_final = - fabs(tecta - alfa);

                        giro\_final = alfa - tecta;

                    }

                    if (m < 0){ //Pendent placa negativa

                        giro\_final = -(M\_PI  - alfa - tecta);   //180º - alfa - tecta

                    }

            }

            // Escriure resultats de trigonometria en TXT

            output\_file\_txt << "Resultats càlculs trigonomètrics:\n";

            output\_file\_txt << "giro\_inicial (radians): " << giro\_inicial << "\n";

            output\_file\_txt << "giro\_inicial (graus): " << (giro\_inicial \* 180.0 / M\_PI) << "\n";

            output\_file\_txt << "avance: " << avance << "\n";

            output\_file\_txt << "giro\_final (radians): " << giro\_final << "\n";

            output\_file\_txt << "giro\_final (graus): " << (giro\_final \* 180.0 / M\_PI) << "\n";

            // Alfa

            output\_file\_txt << "alfa (radians): " << alfa << "\n";

            output\_file\_txt << "alfa (graus): " << (alfa \* 180.0 / M\_PI) << "\n";

            // Tecta

            output\_file\_txt << "tecta (radians): " << tecta << "\n";

            output\_file\_txt << "tecta (graus): " << (tecta \* 180.0 / M\_PI) << "\n";

            // Pdx

            output\_file\_txt << "Pdx: " << Pdx << "\n";

            // Pdy

            output\_file\_txt << "Pdy: " << Pdy << "\n";

            output\_file\_txt << "M\_PI\_2: " << M\_PI\_2 << "\n";

            output\_file\_txt << "M\_PI: " << M\_PI << "\n";

            // Escribir resultados CSV

            output\_file\_csv << "\nPendents (m), Intersecció (b), Distància entre extrems (m), x\_extrem\_izq, y\_extrem\_izq, x\_extrem\_der, y\_extrem\_der\n";

            output\_file\_csv << m << ", " << b << ", " << distancia\_extrems << ", " << x\_extrem\_izq << ", " << y\_extrem\_izq << ", " << x\_extrem\_der << ", " << y\_extrem\_der << "\n";

            ///////////

        }

        output\_file\_txt.close();

        output\_file\_csv.close();

        RCLCPP\_INFO(this->get\_logger(), "Datos del LIDAR guardados en los archivos: %s y %s", output\_file\_path\_txt.c\_str(), output\_file\_path\_csv.c\_str());

        move\_to\_lidar\_done = true;

    }

   /\* void mover\_robot() {

        // Constantes de velocidad (ajustables según el robot)

        float angular\_velocity = 0.1;  // radianes/s

        float linear\_velocity = 0.05;  // metros/s

        RCLCPP\_INFO(this->get\_logger(), "Pula boton para enviar Twist 1");

        getchar();

        //ros::Duration(0.5).sleep(); // Espera 0.5 segundos antes de iniciar el movimiento

        rclcpp::sleep\_for(std::chrono::milliseconds(500));

        // 1. Gira lo que valga giro\_inicial

        twist\_.angular.z = (giro\_inicial < 0 ? -angular\_velocity : angular\_velocity);

        twist\_.linear.x = 0.0;

        pub\_cmd\_vel\_->publish(twist\_);

        // Calcula el tiempo necesario para completar el giro

        double giro\_time = fabs(giro\_inicial) / angular\_velocity;

        //ros::Duration(giro\_time).sleep(); // Mantén el giro durante el tiempo necesario

        rclcpp::sleep\_for(std::chrono::milliseconds(int(giro\_time\*1000)));

        float yaw\_inicial = yaw;

        float yaw\_actual = 0.0;

        float error = 0.0;

        uint cnt = 0;

        bool converged = False;

        while(converged==False){

            yaw\_actual = yaw - yaw\_inicial;

            error = giro\_inicial - yaw\_actual;

            twist\_.angular.z = K \* error;

            twist\_.linear.x = 0.0;

            pub\_cmd\_vel\_->publish(twist\_);

            rclcpp::sleep\_for(std::chrono::milliseconds(1000));

            rclcpp::spin\_some(this);

            if(abs(error) < 0.01){

                cnt++;

                if(cnt>10)

                    converged = True;

            }else{

                cnt = 0;

            }

        }

        // Para el giro

        twist\_.angular.z = 0.0;

        pub\_cmd\_vel\_->publish(twist\_);

        RCLCPP\_INFO(this->get\_logger(), "Pula boton para enviar Twist 2");

        getchar();

        //ros::Duration(0.5).sleep(); // Espera 0.5 segundos

        rclcpp::sleep\_for(std::chrono::milliseconds(500));

        // 2. Avanza en línea recta lo que valga avance

        twist\_.linear.x = (avance < 0 ? -linear\_velocity : linear\_velocity);

        twist\_.angular.z = 0.0;  // Asegúrate de que no haya rotación mientras avanza

        pub\_cmd\_vel\_->publish(twist\_);

        // Calcula el tiempo necesario para avanzar

        double avance\_time = fabs(avance) / linear\_velocity;

        //ros::Duration(avance\_time).sleep(); // Mantén el avance durante el tiempo necesario

        rclcpp::sleep\_for(std::chrono::milliseconds(int(avance\_time\*1000)));

        // Para el avance

        twist\_.linear.x = 0.0;

        pub\_cmd\_vel\_->publish(twist\_);

        RCLCPP\_INFO(this->get\_logger(), "Pula boton para enviar Twist 3");

        getchar();

        //ros::Duration(1).sleep(); // Espera 1 segundos

        rclcpp::sleep\_for(std::chrono::milliseconds(1000));

        // 3. Gira lo que valga giro\_final

        twist\_.angular.z = (giro\_final < 0 ? -angular\_velocity : angular\_velocity);

        pub\_cmd\_vel\_->publish(twist\_);

        // Calcula el tiempo necesario para completar el giro

        double giro\_final\_time = fabs(giro\_final) / angular\_velocity;

        //ros::Duration(giro\_final\_time).sleep(); // Mantén el giro durante el tiempo necesario

        rclcpp::sleep\_for(std::chrono::milliseconds(int(giro\_final\_time\*1000)));

        // Para el giro

        twist\_.angular.z = 0.0;

        pub\_cmd\_vel\_->publish(twist\_);

        RCLCPP\_INFO(this->get\_logger(), "Finalizacion DOCKING.");

    }

    /\*

    void mover\_robot() {

    float K\_angular = 0.5;  // Constante proporcional para giro

    float K\_lineal = 0.5;   // Constante proporcional para avance

    float error, yaw\_inicial, x\_inicial, y\_inicial;

    uint cnt;

    RCLCPP\_INFO(this->get\_logger(), "Inicio de movimiento");

    // Obtener odometría inicial

    yaw\_inicial = yaw;

    x\_inicial = x;

    y\_inicial = y;

    // 1. Giro inicial con control proporcional

    cnt = 0;

    while (true) {

        float yaw\_actual = yaw - yaw\_inicial;

        error = giro\_inicial - yaw\_actual;

        twist\_.angular.z = K\_angular \* error;

        twist\_.linear.x = 0.0;

        pub\_cmd\_vel\_->publish(twist\_);

        rclcpp::sleep\_for(std::chrono::milliseconds(100));

        rclcpp::spin\_some(this);

        if (fabs(error) < 0.01) {

            if (++cnt > 10) break;

        } else {

            cnt = 0;

        }

    }

    twist\_.angular.z = 0.0;

    pub\_cmd\_vel\_->publish(twist\_);

    // 2. Avance en línea recta con control proporcional

    cnt = 0;

    while (true) {

        float distancia\_recorrida = sqrt(pow(x - x\_inicial, 2) + pow(y - y\_inicial, 2));

        error = avance - distancia\_recorrida;

        twist\_.linear.x = K\_lineal \* error;

        twist\_.angular.z = 0.0;

        pub\_cmd\_vel\_->publish(twist\_);

        rclcpp::sleep\_for(std::chrono::milliseconds(100));

        rclcpp::spin\_some(this);

        if (fabs(error) < 0.01) {

            if (++cnt > 10) break;

        } else {

            cnt = 0;

        }

    }

    twist\_.linear.x = 0.0;

    pub\_cmd\_vel\_->publish(twist\_);

    // 3. Giro final con control proporcional

    yaw\_inicial = yaw;

    cnt = 0;

    while (true) {

        float yaw\_actual = yaw - yaw\_inicial;

        error = giro\_final - yaw\_actual;

        twist\_.angular.z = K\_angular \* error;

        pub\_cmd\_vel\_->publish(twist\_);

        rclcpp::sleep\_for(std::chrono::milliseconds(100));

        rclcpp::spin\_some(this);

        if (fabs(error) < 0.01) {

            if (++cnt > 10) break;

        } else {

            cnt = 0;

        }

    }

    twist\_.angular.z = 0.0;

    pub\_cmd\_vel\_->publish(twist\_);

    RCLCPP\_INFO(this->get\_logger(), "Finalización de DOCKING.");

}

    \*/

    void mover\_robot() {

        // Constantes de velocidad

        float angular\_velocity = 0.1;  // rad/s

        float linear\_velocity = 0.05;  // m/s

        float Kpv = 1.0, Kpw = 1.0; // Ganancia del control proporcional

        float Kiv = 0.0, Kiw = 0.0; // Ganancia del control integral

        float w = 0, v = 0;

        float w\_max = 0.5, v\_max = 0.2;

        RCLCPP\_INFO(this->get\_logger(), "Pulsa botón para enviar Twist 1");

        getchar();

        rclcpp::sleep\_for(std::chrono::milliseconds(500));

        // 1. Gira lo que valga giro\_inicial con control proporcional

        float yaw\_inicial = yaw;

        float error = 0;

        float int\_error = 0;

        uint cnt = 0;

        bool converged = false;

        while (!converged && rclcpp::ok()) {

            rclcpp::spin\_some(this->get\_node\_base\_interface());

            float yaw\_actual = yaw - yaw\_inicial;

            error = giro\_inicial - yaw\_actual;

            int\_error = int\_error + error;

            w = Kpw \* error + Kiw \* int\_error;

            RCLCPP\_INFO(this->get\_logger(), "Error: %f, Int Error: %f, W: %f",error,int\_error,w);

            if(w>w\_max)

                w=w\_max;

            else if(w<-w\_max)

                w=-w\_max;

            twist\_.angular.z = w;

            twist\_.linear.x = 0.0;

            pub\_cmd\_vel\_->publish(twist\_);

            rclcpp::sleep\_for(std::chrono::milliseconds(100));

            if (fabs(error) < 0.01) {

                cnt++;

                if (cnt > 10) converged = true;

            } else {

                cnt = 0;

            }

        }

        twist\_.angular.z = 0.0;

        pub\_cmd\_vel\_->publish(twist\_);

        RCLCPP\_INFO(this->get\_logger(), "Pulsa botón para enviar Twist 2");

        getchar();

        rclcpp::sleep\_for(std::chrono::milliseconds(500));

        // 2. Avanza en línea recta con control proporcional

        float x\_inicial = x;

        float y\_inicial = y;

        error = 0;

        int\_error = 0;

        cnt = 0;

        converged = false;

        while (!converged && rclcpp::ok()) {

            rclcpp::spin\_some(this->get\_node\_base\_interface());

            float dist\_actual = sqrt(pow(x - x\_inicial, 2) + pow(y - y\_inicial, 2));

            error = avance - dist\_actual;

            int\_error = int\_error + error;

            v = Kpv \* error + Kiv \* int\_error;

            RCLCPP\_INFO(this->get\_logger(), "Error: %f, Int Error: %f, V: %f",error,int\_error,v);

            if(v>v\_max)

                v=v\_max;

            else if(v<-v\_max)

                v=-v\_max;

            twist\_.linear.x = v;

            twist\_.angular.z = 0.0;

            pub\_cmd\_vel\_->publish(twist\_);

            rclcpp::sleep\_for(std::chrono::milliseconds(100));

            if (fabs(error) < 0.01) {

                cnt++;

                if (cnt > 10) converged = true;

            } else {

                cnt = 0;

            }

        }

        twist\_.linear.x = 0.0;

        pub\_cmd\_vel\_->publish(twist\_);

        RCLCPP\_INFO(this->get\_logger(), "Pulsa botón para enviar Twist 3");

        getchar();

        rclcpp::sleep\_for(std::chrono::milliseconds(1000));

        // 3. Gira lo que valga giro\_final con control proporcional

        yaw\_inicial = yaw;

        error = 0;

        int\_error = 0;

        cnt = 0;

        converged = false;

        while (!converged && rclcpp::ok()) {

            rclcpp::spin\_some(this->get\_node\_base\_interface());

            float yaw\_actual = yaw - yaw\_inicial;

            error = giro\_final - yaw\_actual;

            int\_error = int\_error + error;

            w = Kpw \* error + Kiw \* int\_error;

            RCLCPP\_INFO(this->get\_logger(), "Error: %f, Int Error: %f, W: %f",error,int\_error,w);

            if(w>w\_max)

                w=w\_max;

            else if(w<-w\_max)

                w=-w\_max;

            twist\_.angular.z = w;

            twist\_.linear.x = 0.0;

            pub\_cmd\_vel\_->publish(twist\_);

            rclcpp::sleep\_for(std::chrono::milliseconds(100));

            if (fabs(error) < 0.01) {

                cnt++;

                if (cnt > 10) converged = true;

            } else {

                cnt = 0;

            }

        }

        twist\_.angular.z = 0.0;

        pub\_cmd\_vel\_->publish(twist\_);

        RCLCPP\_INFO(this->get\_logger(), "Finalización DOCKING.");

    }

    // Función para convertir un cuaternión a yaw

    double getYawFromQuaternion(const geometry\_msgs::msg::Quaternion& q) {

        tf2::Quaternion quat(q.x, q.y, q.z, q.w);

        double roll, pitch, yaw;

        tf2::Matrix3x3(quat).getRPY(roll, pitch, yaw);  // Convierte cuaternión a ángulos de Euler

        return yaw;  // Devuelve el yaw (rotación alrededor del eje Z)

    }

    // Función para calcular la odometría

    void odom\_inicio(float& x, float& y, float& yaw) {

        // Ruta del archivo

        std::string file\_path = "/home/robot/ros/docking\_ws/src/docking/resultados.txt";

        // Abrir el archivo en modo de añadir datos

        std::ofstream file(file\_path, std::ios::app);

        if (file.is\_open()) {

            file << ", los valores de odometría inicial son: "

                << "x=" << x << ", y=" << y << ", yaw=" << yaw << "\n";

            // Obtener el tiempo actual

            auto now = std::chrono::system\_clock::now();

            std::time\_t current\_time = std::chrono::system\_clock::to\_time\_t(now);

            char time\_str[100];

            std::strftime(time\_str, sizeof(time\_str), "%H:%M:%S", std::localtime(&current\_time));

            tiempo\_inicial = time\_str; // Guarda tiempo en varaible global

            //file << "Tiempo: " << time\_str << "\n";

            file.close();

            RCLCPP\_INFO(this->get\_logger(), "Datos escritos en el archivo: %s", file\_path.c\_str());

        } else {

            RCLCPP\_ERROR(this->get\_logger(), "No se pudo abrir el archivo: %s", file\_path.c\_str());

        }

    }

    // Función para calcular la odometría

    void odom\_final(float& x, float& y, float& yaw) {

        // Ruta del archivo

        std::string file\_path = "/home/robot/ros/docking\_ws/src/docking/resultados.txt";

        // Abrir el archivo en modo de añadir datos

        std::ofstream file(file\_path, std::ios::app);

        if (file.is\_open()) {

            file << ", los valores de odometría final son: "

                << "x=" << x << ", y=" << y << ", yaw=" << yaw << "\n";

            // Obtener el tiempo actual

            auto now = std::chrono::system\_clock::now();

            std::time\_t current\_time = std::chrono::system\_clock::to\_time\_t(now);

            char time\_str[100];

            std::strftime(time\_str, sizeof(time\_str), "%H:%M:%S", std::localtime(&current\_time));

            tiempo\_final = time\_str; // Guarda tiempo en varaible global

            //file << "Tiempo: " << time\_str << "\n";

            file.close();

            RCLCPP\_INFO(this->get\_logger(), "Datos escritos en el archivo: %s", file\_path.c\_str());

        } else {

            RCLCPP\_ERROR(this->get\_logger(), "No se pudo abrir el archivo: %s", file\_path.c\_str());

        }

    }

    // Función para convertir un string HH:MM:SS a segundos desde las 00:00:00

    int convertirATiempoEnSegundos(const std::string& tiempo) {

        std::tm t = {};

        std::istringstream ss(tiempo);

        ss >> std::get\_time(&t, "%H:%M:%S");

        if (ss.fail()) {

            throw std::invalid\_argument("Formato de tiempo no válido: " + tiempo);

        }

        return t.tm\_hour \* 3600 + t.tm\_min \* 60 + t.tm\_sec;

    }

    // Función para convertir radianes a grados

    float radiansToDegrees(float radians) {

        return radians \* (180.0 / M\_PI);

    }

    // Función para convertir radianes a grados

    float degreesToRadians(float degrees) {

        return degrees \* (M\_PI / 180.0);

    }

    // Función que calcula y escribe los resultados

    void calcularYEscribir() {

        // Definir la ruta del archivo como variable local

        std::string file\_path = "/home/robot/ros/docking\_ws/src/docking/resultados.txt";

        // Punto de referencia

        float ref\_x = -0.35, ref\_y = 0.0;

        float ref\_w = M\_PI;

        // Calcular distancia euclidiana al punto de referencia

        float distancia = std::sqrt(std::pow(final\_x - ref\_x, 2) + std::pow(final\_y - ref\_y, 2));

        // Calcular la diferencia angular y convertir a grados

        float diferencia\_w = abs(radiansToDegrees(abs(final\_w) - ref\_w));

        // Calcular la diferencia de tiempo en segundos

        int diferencia\_tiempo = 0;

        try {

            if (!tiempo\_inicial.empty() && !tiempo\_final.empty()) {

                int tiempo\_inicial\_segundos = convertirATiempoEnSegundos(tiempo\_inicial);

                int tiempo\_final\_segundos = convertirATiempoEnSegundos(tiempo\_final);

                diferencia\_tiempo = tiempo\_final\_segundos - tiempo\_inicial\_segundos;

            }

        } catch (const std::invalid\_argument& e) {

            std::cerr << "Error al calcular la diferencia de tiempo: " << e.what() << std::endl;

        }

        // Escribir los resultados en el archivo

        std::ofstream file(file\_path, std::ios::app); // Abrir en modo de append

        if (file.is\_open()) {

            file << "Distancia al punto (-0.35, 0): " << distancia << " metros\n";

            file << "Diferencia angular con PI: " << diferencia\_w << " grados\n";

            file << "Diferencia de tiempo: " << diferencia\_tiempo << " segundos\n";

            file.close();

            std::cout << "Resultados guardados en " << file\_path << std::endl;

        } else {

            std::cerr << "No se pudo abrir el archivo " << file\_path << std::endl;

        }

    }

    //rclcpp::TimerBase::SharedPtr timer\_;

    //rclcpp::Publisher<std\_msgs::msg::String>::SharedPtr publisher\_;

    rclcpp::Publisher<geometry\_msgs::msg::Twist>::SharedPtr pub\_cmd\_vel\_;

    rclcpp::Subscription<nav\_msgs::msg::Odometry>::SharedPtr sub\_odom\_;

    rclcpp::Subscription<sensor\_msgs::msg::LaserScan>::SharedPtr sub\_laser\_;

    size\_t count\_;

    // Variables globales

    int angulos\_der = 0;

    int angulos\_izq = 0;

    double x\_extrem\_izq = 0;

    double y\_extrem\_izq = 0;

    double x\_extrem\_der = 0;

    double y\_extrem\_der = 0;

    float dist\_to\_placa = 0.15;

    float giro\_inicial = 0;

    float avance = 0;

    float giro\_final = 0;

    float offset\_lidar = 0.064; // 6.4 cm abajo del centro de giro

    // Variables para almacenar los valores de odometría

    float inicial\_x = 0.0, inicial\_y = 0.0, inicial\_w = 0.0;

    float final\_x = 0.0, final\_y = 0.0, final\_w = 0.0;

    float x, y, yaw;

    // Variable global para almacenar el tiempo

    std::string tiempo\_inicial = "";

    std::string tiempo\_final = "";

    sensor\_msgs::msg::LaserScan scan\_;

    nav\_msgs::msg::Odometry odom\_;

    geometry\_msgs::msg::Twist twist\_;

  };

  int main(int argc, char \* argv[])

  {

    rclcpp::init(argc, argv);

    auto node = std::make\_shared<Docking>();

    //auto node = rclcpp::Node::make\_shared<Docking>();

    //while (rclcpp::ok())

    while (!node->is\_processed || !node->move\_to\_lidar\_done)

    {

        RCLCPP\_INFO(rclcpp::get\_logger("rclcpp"), "Waiting...");

        rclcpp::sleep\_for(std::chrono::milliseconds(1000));

        rclcpp::spin\_some(node);

    }

    node->docking();

    rclcpp::spin(node);

    rclcpp::shutdown();

    return 0;

  }