MÁSTER UNIVERSITARIO EN INGENIERÍA INDUSTRIAL

TRABAJO DE FIN DE MÁSTER

Desarrollo de un módulo de navegación en entornos dinámicos para una plataforma robótica con cinemática diferencial

Autor: Lorenzo Rodríguez Pérez

Directores:

Jaime Boal Martín-Larrauri

Juan Luis Zamora Macho

Madrid

Junio 2024

Declaro, bajo mi responsabilidad, que el Proyecto presentado con el título DESARROLLO DE UN MÓDULO DE NAVEGACIÓN EN ENTORNOS DINÁMICOS PARA UNA PLATAFORMA CON CINEMÁTICA DIFERENCIAL en la ETS de Ingeniería - ICAI de la Universidad Pontificia Comillas en el curso académico 2023/24 es de mi autoría, original e inédito y no ha sido presentado con anterioridad a otros efectos. El Proyecto no es plagio de otro, ni total ni parcialmente y la información que ha sido tomada de otros documentos está debidamente referenciada.

Fdo.: Lorenzo Rodríguez Pérez Fecha: 27/ 06/ 2024

Autorizada la entrega del proyecto

EL DIRECTOR DEL PROYECTO

Firmado digitalmente por BOAL MARTIN LARRAURI JAIME - 05304600H
Fecha: 2024.06.26 17:10:49 +02'00'

Fdo.: Jaime Boal Martín-Larrauri Fecha: ……/ ……/ ……

Fdo.: Juan Luis Zamora Macho Fecha: ……/ ……/ ……

MÁSTER UNIVERSITARIO EN INGENIERÍA INDUSTRIAL

TRABAJO DE FIN DE MÁSTER

Desarrollo de un módulo de navegación en entornos dinámicos para una plataforma robótica con cinemática diferencial

Autor: Lorenzo Rodríguez Pérez

Directores:

Jaime Boal Martín-Larrauri

Juan Luis Zamora Macho

Madrid

Junio 2024

Agradecimientos

Quiero agradecer en primer lugar a mis directores, Jaime Boal y Juan Luis Zamora, por su constante disposición para ayudarme y por todo lo que me han enseñado durante este proyecto.

A mis padres, por brindarme la oportunidad de estudiar en esta universidad, por confiar siempre en mí y por inculcarme la pasión por las matemáticas y la ingeniería.

Finalmente, quiero agradecer a la Universidad Pontificia Comillas por la oportunidad de trabajar en este proyecto y por la formación que me ha proporcionado.

Desarrollo de un módulo de navegación en entornos dinámicos para una plataforma robótica con cinemática diferencial

Autor: Rodríguez Pérez, Lorenzo.

Directores: Boal Martín-Larrauri, Jaime; Zamora Macho, Juan Luis.

Entidad Colaboradora: ICAI – Universidad Pontificia Comillas.

Resumen del proyecto

1. Introducción

Este proyecto participa en la competición UNIJES SocialTech Challenge, organizada por cuatro universidades. El objetivo es construir una silla de ruedas autónoma que permita a personas con movilidad reducida visitar espacios públicos. El trabajo se enfoca en integrar algoritmos de navegación y control de trayectorias en una plataforma robótica con cinemática diferencial.

2. Arquitectura

La arquitectura del sistema puede observarse en la Figura 1. El proceso comienza con un LiDAR que envía datos de localización a una Nvidia Jetson AGX Orin. En esta, se ejecuta el módulo de planificación usando Nav2. Las rutas se envían a una Raspberry Pi 4, que ejecuta el controlador para generar comandos de velocidad. Estos comandos se transmiten al sistema de control del vehículo.

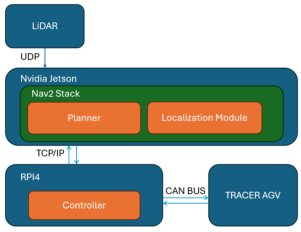

Figura 1: Arquitectura del sistema

2.1 Hardware y comunicaciones

El sistema utiliza dos componentes principales: la Nvidia Jetson AGX Orin y la Raspberry Pi 4B. En la primera se ejecutan los módulos de planificación y localización, mientras que en la segunda se ejecutan los algoritmos de control. La comunicación entre estos componentes se realiza a través de TCP/IP, con servidores y clientes programados en Simulink Coder y Python. La Raspberry Pi accede y maneja datos del vehículo mediante un adaptador CAN a USB.

2.2 ROS2

Se ha utilizado ROS2 [1] junto con Nav2 para la integración de los módulos de navegación y localización. El comportamiento del sistema se resume en la interacción de diversos nodos y *topics*. En la Figura 2 puede observarse el grafo resultante durante la ejecución del sistema. Los rectángulos representan *topics* y las elipses nodos. Los hexágonos representan conjuntos de nodos y *topics*.

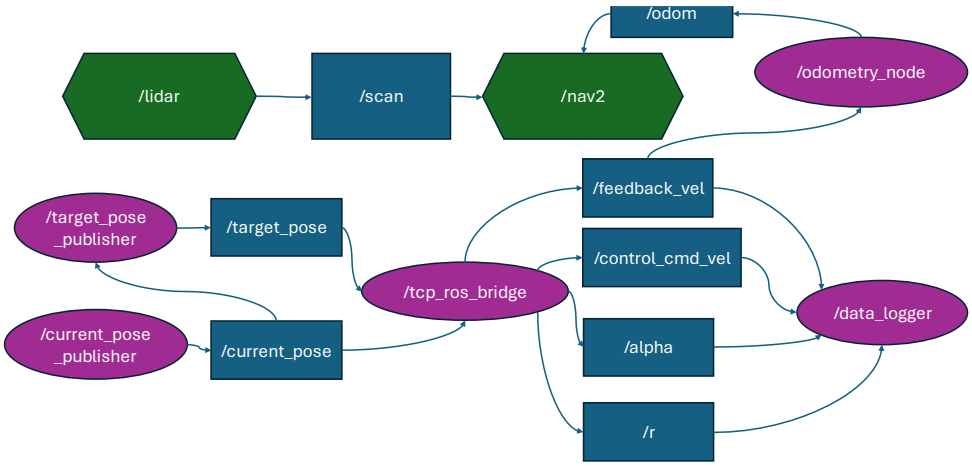

Figura 2: Grafo resultante durante la ejecución

3. Seguimiento de rutas

Se han probado tres alternativas para el algoritmo de control de seguimiento de rutas: LQR (*linear quadratic regulator*), MPC (*model predictive control*) y *pure pursuit*. Los reguladores LQR y MPC fueron optimizados mediante algoritmos genéticos, mientras que el *pure pursuit* fue optimizado mediante prueba y error porque su mayor simplicidad lo permite. Para los tres, se muestran a continuación simulaciones de un movimiento a un punto situado a 1m detrás del robot.

3.1 LQR

Para el LQR no fue necesaria una ejecución larga del algoritmo genético, puesto que las matrices Q y R propuestas como solución inicial ya otorgaron una solución bastante buena. En la Figura 3 puede observarse la respuesta. La desviación máxima es de 0.072 m en el eje Y y la posición objetivo se alcanza en menos de 10s.

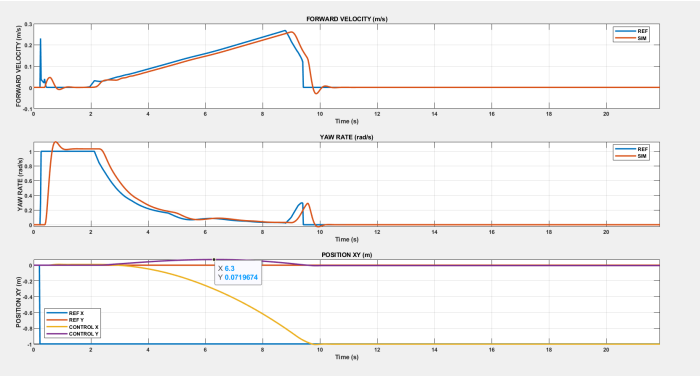

Figura 3: Simulación del controlador LQR

3.2 MPC

El MPC, por contar con una carga computacional mucho más grande, necesitó periodos de optimización mucho más largos que el LQR. En la Figura 4 puede observarse una simulación con el controlador MPC.

Puede observarse que la respuesta cuenta con desviaciones mayores que las del controlador LQR. Además, el tiempo de alcance es mayor. Teóricamente, este controlador podría dar una respuesta mejor al LQR, pero se abandonó el proceso de optimización porque la Raspberri Pi no cuenta con potencia computacional suficiente para ejecutarlo.

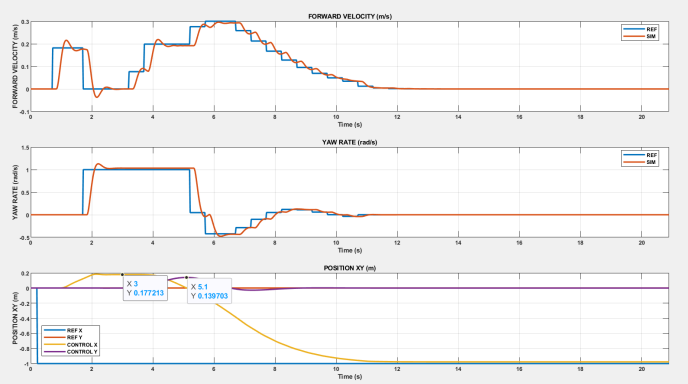

Figura 4: Simulación del controlador MPC

3.3 Pure Pursuit

Para el controlador *pure pursuit* se obtuvo una respuesta con un sobrepaso máximo en el eje Y de 0.063 m y un tiempo de alcance menor a los 8 segundos. Además de ser más rápido y contar con un sobrepaso ligeramente inferior al LQR, este cuenta con una ventaja adicional: la suavidad de los mandos. Puede observarse la respuesta en la Figura 5.

En el diseño final del prototipo se mantuvo el controlador *pure pursuit* ya que se consiguió un conjunto de parámetros que daba una respuesta suficientemente buena de forma rápida, pero debe destacarse que el LQR y el MPC tienen el potencial de dar soluciones mejores por ser controladores de carácter dinámico.

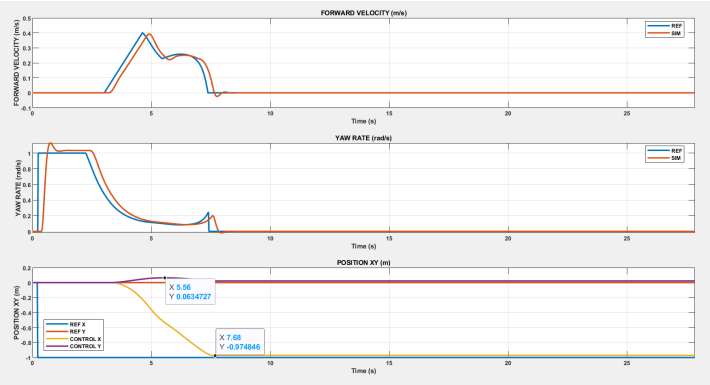

Figura 5: Simulación del controlador Pure Pursuit

4. Planificación de rutas

Para el módulo de planificación de rutas, se han explorado dos alternativas: La implementación de un RRT* mediante un *plugin* de Nav2 y usar el planificador estándar de Nav2, el NavFnPlanner.

4.1 RRT*

El *plugin* ha sido programado en C++ en el *stack* de Nav2. El algoritmo RRT* se ha implementado junto a 2 etapas adicionales, un procedimiento de simplificación mediante el algoritmo de Douglas-Peucker [1], y la segunda, un proceso de suavización que utiliza técnicas de descenso de gradiente. Sin embargo, este planificador resultó optar por maniobras de giro redundantes en algunas ocasiones.

4.2 NavFnPlanner

Este planificador es versátil en su funcionamiento, pudiendo recurrir al algoritmo de Dijkstra o al algoritmo A* en función de las preferencias del usuario. La elección ha recaído en el algoritmo A*, ya que este garantiza un menor número de iteraciones [2]. Durante las evaluaciones comparativas, el NavFnPlanner ha demostrado una eficiencia superior en la generación de rutas, y por esto este ha sido el empleado en el sistema final. Puede observarse un ejemplo de planificación en la Figura 6.

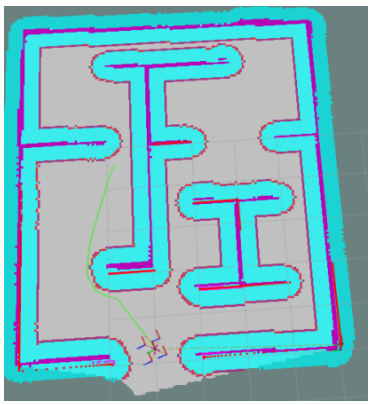

Figura 6: Ejemplo de planificación del NavFnPlanner

5. Integración

5.1 Integración Hardware-Software

Para facilitar el manejo del sistema, se han integrado un conmutador, un botón y un joystick. Estos permiten al usuario cambiar entre modos de funcionamiento, permitiendo un movimiento manual con el joystick.

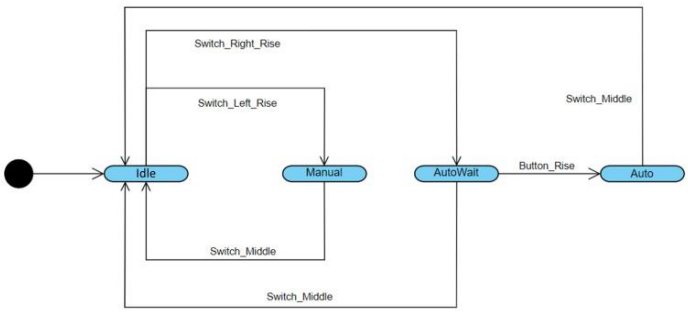

Figura 7: Transiciones en la máquina de estados

Una máquina de estados gestiona los distintos modos de funcionamiento del sistema. Esta máquina de estados activa o desactiva la navegación autónoma según el modo seleccionado. La transición entre los estados se ilustra en la Figura 7.

6. Resultados

El prototipo final mostró un buen rendimiento durante el evento final del UNIJES SocialTech Challenge, siendo el único en superar con éxito las dos pruebas realizadas: navegación en el entorno y navegación en el entorno con obstáculos estáticos no presentes en el mapa. Este desempeño le permitió ganar los premios de "Mejor Comportamiento en Pruebas" y "Ganador Absoluto", obteniendo así dos de los tres galardones otorgados en la competición.

7. Conclusiones y futuros desarrollos

7.1 Eliminación de la Raspberry Pi de la arquitectura

La Raspberry Pi, inicialmente seleccionada por su facilidad de integración, resulta redundante en el diseño final. Su eliminación traería varios beneficios, como la simplificación de las comunicaciones, la posibilidad de usar controladores más complejos y la integración de todo el software en ROS2.

7.2 Planificación en base a los cambios en el mapa de costes

El sistema actual, aunque funcional, presenta oscilaciones debido a la alta frecuencia de planificación. Para solucionar esto, se propone un *plugin* en Nav2 que planifique rutas basadas en cambios significativos en el mapa de costes. Esto reduciría las oscilaciones sin comprometer la capacidad de adaptación a cambios en el entorno.

Referencias

[1] ROS2 Community, «ROS2 Documentation,» [En línea]. Available: https://docs.ros.org/en/humble/index.html. [Último acceso: 10 June 2024].

[2] D. Douglas y T. Peucker, «Algorithms for the reduction of the number of points required to represent a digitized line or its caricature,» *Cartographica: The International Journal for Geographic Information and Geovisualization ,* vol. X, nº 2, pp. 112-122, 1973.

[3] R. Siegwart, I. R. Nourbakhsh y D. Scaramuzza, Introduction to Autonomous Mobile Robots, Cambridge: The MIT Press, 2011.

Development of a Navigation Module in Dynamic Environments for a Robotic Platform with Differential Kinematics

Author: Rodríguez Pérez, Lorenzo.

Supervisors: Boal Martín-Larrauri, Jaime; Zamora Macho, Juan Luis.

Collaborating Entity: ICAI – Universidad Pontificia Comillas.

Project Summary

1. Introduction

This project is part of the UNIJES SocialTech Challenge competition, organized by four universities. The objective is to build an autonomous wheelchair that allows people with reduced mobility to visit public spaces. The work focuses on integrating navigation algorithms and trajectory control in a robotic platform with differential kinematics.

2. Architecture

The system architecture can be seen in Figure 1. The process begins with a LiDAR that sends localization data to a Nvidia Jetson AGX Orin. On this platform, the planning module runs using Nav2. The routes are sent to a Raspberry Pi 4, which executes the controller to generate velocity commands. These commands are then transmitted to the vehicle's control system.

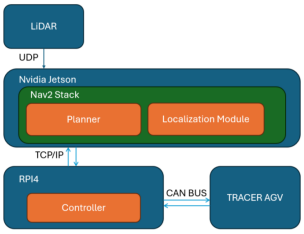

Figure 1: System architecture

2.1 Hardware and Communications

The system uses two main components: the Nvidia Jetson
AGX Orin and the Raspberry Pi 4B. The planning and localization modules run on the former, while the control algorithms run on the latter. Communication between these components is carried out via TCP/IP, with servers and clients programmed in Simulink Coder and Python. The Raspberry Pi accesses and manages vehicle data through a CAN to USB adapter.

2.2 ROS2

ROS2 [1] has been used along with Nav2 for the integration of the navigation and localization modules. The system's behavior is summarized through the interaction of various nodes and topics. Figure 2 shows the resulting graph during the system's execution. The rectangles represent topics, the ellipses represent nodes, and the hexagons represent sets of nodes and topics.

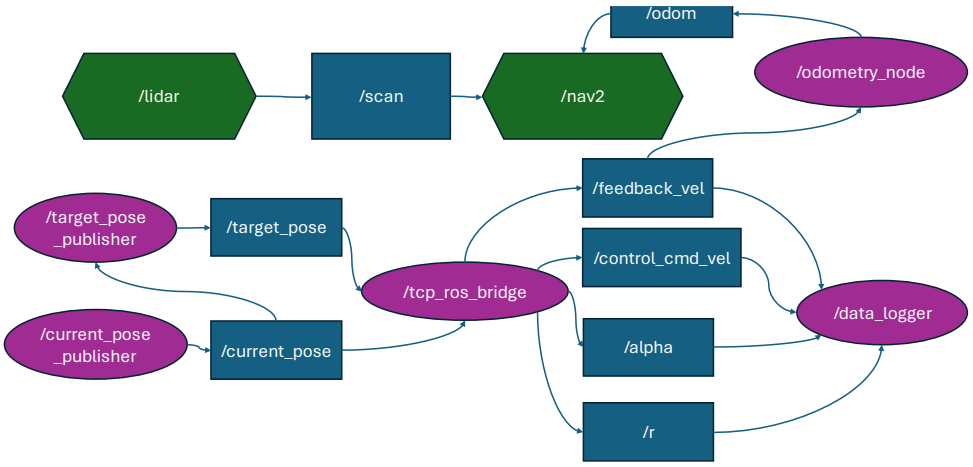

Figure 2: ROS2 Graph

3. Path Tracking

Three alternatives have been tested for the path-following control algorithm: LQR (linear quadratic regulator), MPC (model predictive control), and pure pursuit. The LQR and MPC controllers were optimized using genetic algorithms, while the pure pursuit was optimized through trial and error due to its greater simplicity. Below, simulations of a movement to a point located 1 meter behind the robot are shown for all three.

3.1 LQR

For the LQR, a lengthy execution of the genetic algorithm was not necessary, as the Q and R matrices proposed as the initial solution already provided a fairly good result. The response can be seen in Figure 3. The maximum deviation is 0.072 m on the Y-axis, and the target position is reached in less than 10 seconds.

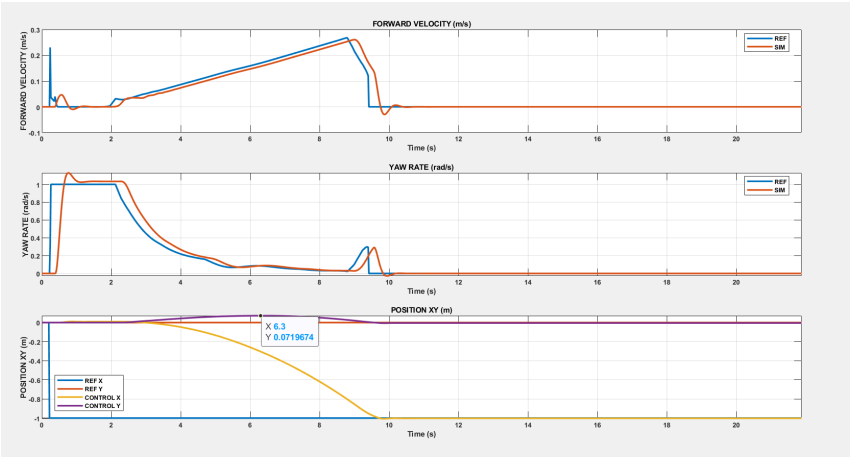

Figure 3: LQR simulation

3.2 MPC

The MPC, due to its much larger computational load, required much longer optimization periods than the LQR. A simulation with the MPC controller can be seen in Figure 4.

It can be observed that the response has greater deviations than the LQR controller. Additionally, the reach time is longer. Theoretically, this controller could provide a better response than the LQR, but the optimization process was abandoned because the Raspberry Pi does not have sufficient computational power to run it.

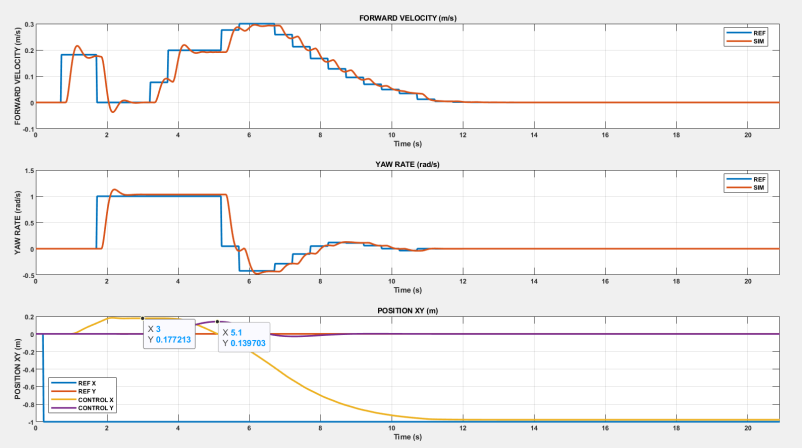

Figure 4: MPC simulation

3.3 Pure Pursuit

For the pure pursuit controller, a response with a maximum overshoot of 0.063 m on the Y-axis and a reach time of less than 8 seconds was obtained. Besides being faster and having a slightly lower overshoot compared to the LQR, it has an additional advantage: the smoothness of the commands. The response can be seen in Figure 5.

In the final design of the prototype, the pure pursuit controller was retained as a set of parameters was found that provided a sufficiently good response quickly. However, it should be noted that the LQR and MPC have the potential to offer better solutions due to their dynamic control nature.

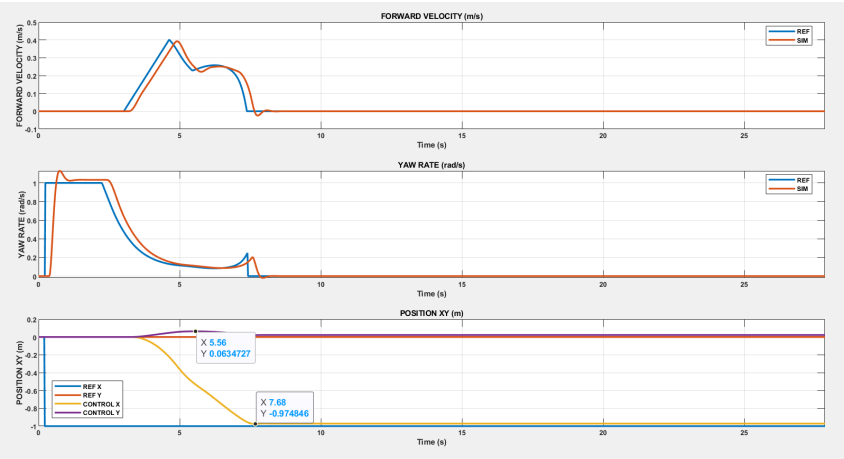

Figure 5: Pure Pursuit simulation

4. Path Planning

For the path planning module, two alternatives have been explored: implementing an RRT* through a Nav2 plugin and using the standard Nav2 planner, the NavFnPlanner.

4.1 RRT*

The plugin has been programmed in C++ within the Nav2 stack. The RRT* algorithm has been implemented along with two additional stages: a simplification procedure using the Douglas-Peucker algorithm [1], and a smoothing process that employs gradient descent techniques. However, this planner occasionally opted for redundant turning maneuvers.

4.2 NavFnPlanner

This planner is versatile in its operation, capable of using either the Dijkstra algorithm or the A* algorithm based on user preferences. The choice fell on the A* algorithm, as it ensures fewer iterations [2]. During comparative evaluations, the NavFnPlanner demonstrated superior efficiency in route generation, and thus it was used in the final system. An example of the planning can be seen in Figure 6.

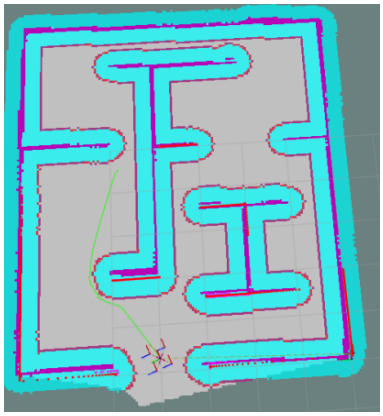

Figure 6: Path generated by NavFnPlanner

5. Integration

5.1 Hardware-Software Integration

To facilitate system operation, a switch, a button, and a joystick have been integrated. These allow the user to switch between operating modes, enabling manual movement with the joystick. A state machine manages the different operating modes of the system. This state machine activates or deactivates

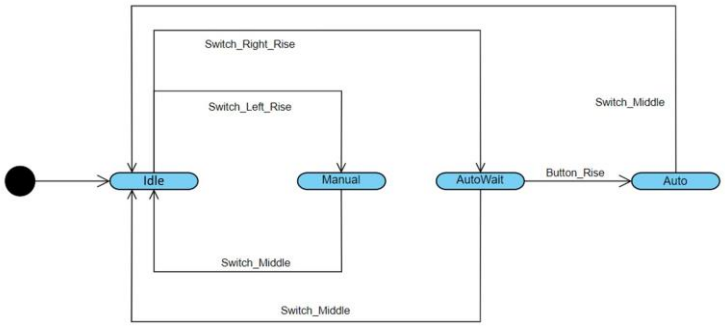

Figure 7: State Machine

autonomous navigation based on the selected mode. The transition between states is illustrated in Figure 7.

6. Results

The final prototype showed good performance during the final event of the UNIJES SocialTech Challenge, being the only one to successfully pass both tests: navigation in the environment and navigation in the environment with static obstacles not present in the map. This performance earned it the awards for "Best Performance in Tests" and "Overall Winner," thus securing two of the three prizes awarded in the competition.

7. Conclusions and Future Developments

7.1 Elimination of the Raspberry Pi from the Architecture

The Raspberry Pi, initially selected for its ease of integration, proves redundant in the final design. Its elimination would bring several benefits, such as simplifying communications, enabling the use of more complex controllers, and integrating all software into ROS2.

7.2 Planning Based on Changes in the Cost Map

The current system, while functional, presents oscillations due to the high frequency of planning. To address this, a plugin in Nav2 is proposed that plans routes based on significant changes in the cost map. This would reduce oscillations without compromising the ability to adapt to changes in the environment.

References

[1] ROS2 Community, «ROS2 Documentation,» [En línea]. Available: https://docs.ros.org/en/humble/index.html. [Accessed: 10 June 2024].

[2] D. Douglas y T. Peucker, «Algorithms for the reduction of the number of points required to represent a digitized line or its caricature,» *Cartographica: The International Journal for Geographic Information and Geovisualization ,* vol. X, nº 2, pp. 112-122, 1973.

[3] R. Siegwart, I. R. Nourbakhsh y D. Scaramuzza, Introduction to Autonomous Mobile Robots, Cambridge: The MIT Press, 2011.

Contenidos

1. INTRODUCCIÓN

Este proyecto se enmarca en la competición de robótica UNIJES SocialTech Challenge, una iniciativa conjunta de cuatro universidades: Universidad de Deusto, Universitat Ramon Llull, Universidad Pontificia Comillas y Universidad Loyola, todas ellas pertenecientes a UNIJES, la red de universidades vinculadas a la Compañía de Jesús en España. El objetivo final de la competición es demostrar el impacto social de la tecnología, fomentando la innovación y la creatividad entre los participantes. En la edición de este año, cada universidad participante construirá una silla de ruedas autónoma que permitirá a personas con movilidad reducida realizar visitas a espacios públicos, como museos. El propósito de este trabajo fin de máster es integrar algoritmos de navegación a largo y corto plazo para evitar obstáculos, así como implementar un control de seguimiento de trayectorias en una plataforma robótica con cinemática diferencial sobre la que se fijará la silla.

2. ESTADO DE LA CUESTIÓN

2.1 PLATAFORMAS ROBÓTICAS

Se utilizará el robot comercial TRACER AGV de AGILEX como base de soporte para la silla y todo el *hardware* requerido. Existen varias plataformas robóticas comerciales diseñadas para interiores que cabe mencionar:

- Ridgeback Omnidirectional Platform: Ridgeback es una plataforma de robot de tamaño mediano diseñada para uso en interiores que utiliza una unidad de manejo omnidireccional para mover manipuladores y cargas pesadas [3]. La base omnidireccional proporciona posicionamiento de precisión en entornos con espacio limitado y viene completamente integrada con una unida de procesamiento a bordo, escáneres láser frontal y una IMU (Unidad de Medición Inercial). Ridgeback ofrece una integración nativa con ROS y Gazebo, y es compatible con los accesorios para robots de Clearpath. Entre sus características destacan las siguientes:
    - Omnidireccional: Ridgeback está diseñado con ruedas Mecanum omnidireccionales emparejadas con un sistema de accionamiento de precisión y *encoders* de alta resolución (35840 pulsos por revolución) para permitir un control de posición exacto con un margen de maniobrabilidad de hasta 3 grados.
    - Desarrollo sin límites: La versatilidad de Ridgeback permite la planificación de rutas y algoritmos sin restricciones. Incluso puede simular una plataforma más limitada al no utilizar sus capacidades de desplazamiento lateral.
    - Campo de visión (FOV) de 360 grados: El LIDAR integrado y la placa superior están diseñados para proporcionar un campo de visión LIDAR de 360 grados sin obstrucciones.
    - Pequeñas dimensiones: Ridgeback tiene solamente 30 cm de altura y puede pasar por una puerta estándar de 80 cm.

- Personalizable: Ridgeback es compatible con conexión y uso de los accesorios para robots de Clearpath Robotics.
- Especificaciones técnicas: Dimensiones de 960x793x311 mm, peso de 135 kg, carga máxima de 100 kg y velocidad máxima de 1.1 m/s.

En la Figura puede observarse la plataforma Rigdeback transportando dos robots antropomórficos.

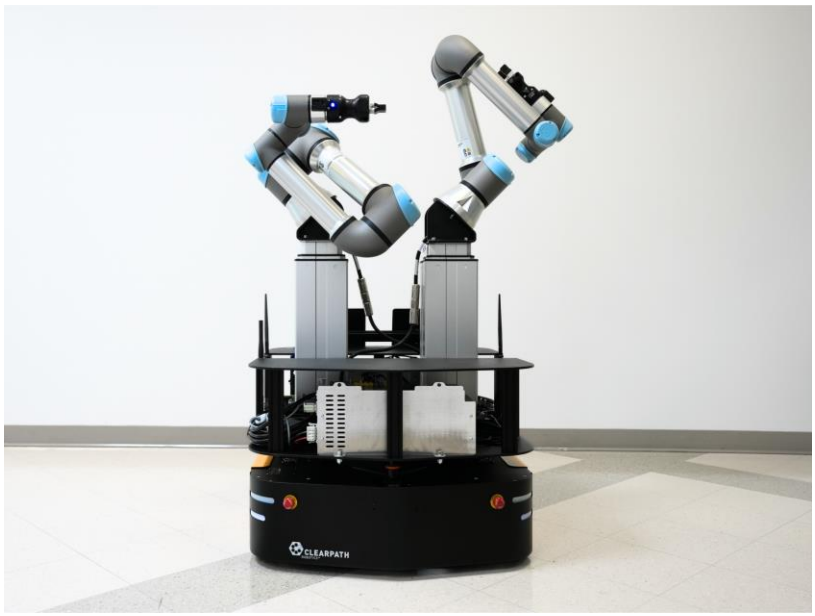

Figura 1: Plataforma Ridgeback [3]

- Boxer indoor mobile robot: Boxer es una plataforma robótica grande diseñada para el desarrollo de aplicaciones industriales en interiores [4]. Adaptado del robot móvil autónomo (AMR) OTTO Motors OTTO 100, Boxer es un robot móvil de grado industrial que es programable y extensible con *hardware* adicional. Entre sus características destacan las siguientes:
  - Completamente extensible: Boxer es compatible con conexión y uso de sensores y componentes de terceros, y puede personalizarse para adaptarse a una amplia variedad de aplicaciones.

- Listo para ROS: La programación de aplicaciones complejas es sencilla gracias al Sistema Operativo de Robots de código abierto (ROS) y al soporte para el simulador de físicas Gazebo, URDF, RViz y el planificador de movimientos MoveIT.
- Equipado con sensores: Los sensores integrados incluyen una cámara estéreo orientada hacia adelante y LIDAR, sensores ultrasónicos traseros, IMU y encoders de las ruedas, todos los cuales están disponibles a través de ROS.
- Especificaciones técnicas: Dimensiones de 750x550x304 mm, peso de 127 kg, carga máxima de 100 kg y velocidad máxima de 2 m/s.

En la Figura 2 puede observarse una imagen de la plataforma Boxer.

Figura 2: Plataforma Boxer [4]

- TRACER AGV: Tracer está diseñado como un Vehículo Terrestre no Tripulado (UGV, por sus siglas en inglés) multipropósito [5]. La combinación de un chasis de dos ruedas diferenciales y motores de cubo le permite moverse de manera flexible en interiores. Componentes adicionales como una cámara estéreo, radar láser, GPS, IMU (Unidad de Medición Inercial) y un manipulador robótico se pueden instalar opcionalmente en TRACER para aplicaciones avanzadas de navegación y visión artificial. TRACER se utiliza con frecuencia en la educación e investigación de

AMRs, patrullaje de seguridad en interiores y exteriores, y transporte, entre otros. Entre sus características destacan las siguientes:

- Desarrollo rápido: Los usuarios pueden comunicarse con el control principal a través del protocolo CAN bus, y también se proporciona un SDK de código abierto y ROS_PACKAGE para el desarrollo.
- Especificaciones técnicas: Dimensiones de 685x570x155 mm, peso de 30 kg, carga máxima de 100 kg y velocidad máxima de 1.5 m/s.

En la Figura 3 puede observarse una imagen de un Tracer transportando paquetes.

Figura 3: Plataforma Tracer [6]

Todas estas plataformas podrían ser utilizadas para el proyecto por contar con suficientes dimensiones y una carga máxima igual o superior a 100 kg, pero se ha seleccionado el TRACER AGV por motivos económicos, ya que este tiene un coste de 7500€, bastante menor a los 46368€ del Ridgeback y los 52800€ del Boxer.

2.2 ALGORITMOS DE SEGUIMIENTO DE RUTAS

El proyecto necesita tanto un planificador de rutas como un algoritmo de seguimiento de estas. Es común encontrar los siguientes algoritmos de seguimiento de trayectorias en aplicaciones industriales similares a esta:

- Sigue la zanahoria o *follow-the-carrot*: El algoritmo *follow-the-carrot* se basa en una idea muy sencilla. En cada instante, el algoritmo identifica el punto en el camino frente al vehículo a una distancia $R0$ del centro de su eje delantero. Luego, ajusta el valor del ángulo de dirección $\delta(t)$ haciendo uso de un control proporcional que tiene en cuenta el ángulo de desviación y el error de trayectoria [7]. En la Figura 4 puede observarse un diagrama de apoyo para explicar el algoritmo. Los puntos W_i y W_{i+1} son el comienzo y el final del segmento que se quiere seguir en este instante de tiempo (nótese que una trayectoria puede expresarse como un conjunto de puntos unidos por segmentos, por lo que no hay pérdida de generalidad). El punto S es el punto del segmento que se encuentra a una distancia $R0$ del centro del eje delantero del vehículo. En azul se marca la dirección actual del vehículo, de esta manera la dirección actual viene determinada por el ángulo ψ y la dirección deseada por el ángulo ψ_d.

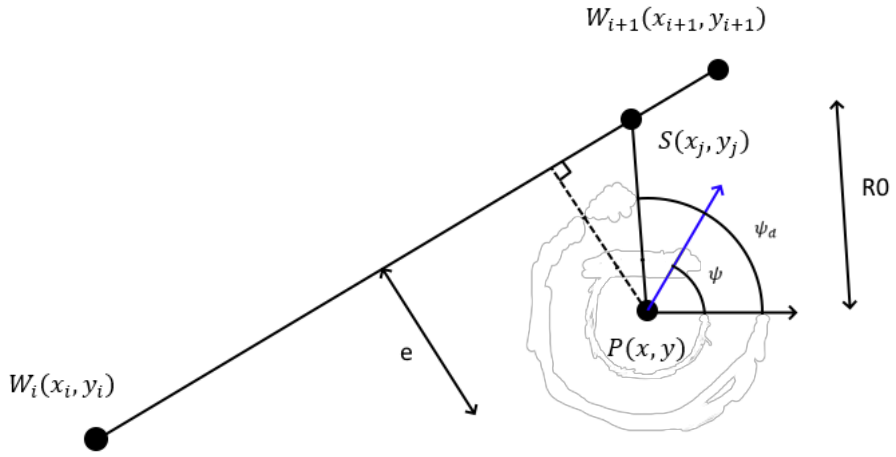

Figura 4: Esquema geométrico del algoritmo follow-the-carrot

El algoritmo *follow-the-carrot* sigue los siguientes pasos en cada iteración [8]:

1. Se calcula la posición del punto $S(x_j, y_j)$.
2. Se calcula el ángulo deseado $\psi_d = \text{atan}\left(\frac{y_j - y}{x_j - x}\right)$.
3. Se calcula el ángulo actual $\psi = \text{atan}\left(\frac{y - y_{-1}}{x - x_{-1}}\right)$ donde $P_{-1}(x_{-1}, y_{-1})$ es la posición del vehículo en la iteración anterior.
4. Se calcula el error de trayectoria e definido como la distancia entre el punto actual del robot y el segmento que une W_i y W_{i+1}.
5. Se actualiza el mando que gobierna el ángulo de dirección $u = K_1 * (\psi_d - \psi) + K_2 * e$, donde K_1 y K_2 son dos constantes a determinar.

Es importante destacar que es necesario tener en cuenta el cuadrante de los ángulos ψ y ψ_d al calcular la arcotangente.

Este algoritmo resulta fácil de implementar, pero cuenta con algunas limitaciones puesto que no considera la influencia de factores externos ya que funciona sin hacer uso de un modelo de la planta.

- Persecución pura o *pure pursuit*: Este algoritmo determina la dirección necesaria para que el robot se desplace desde su posición actual hacia un punto de referencia adelante del robot. El algoritmo ajusta continuamente la posición del punto de referencia a lo largo de la trayectoria según la posición actual del robot, hasta llegar al último punto de la trayectoria. Para ello, en cada iteración calcula un recorrido circular entre el punto en el que se encuentra el robot y el situado en la trayectoria a una distancia concreta denominada distancia de anticipación o *lookahead distance*, en inglés. En la Figura 5 se muestra un esquema del algoritmo donde α es el ángulo entre la dirección actual del vehículo y el segmento l_d, y l_d es el *lookahead distance* [9].

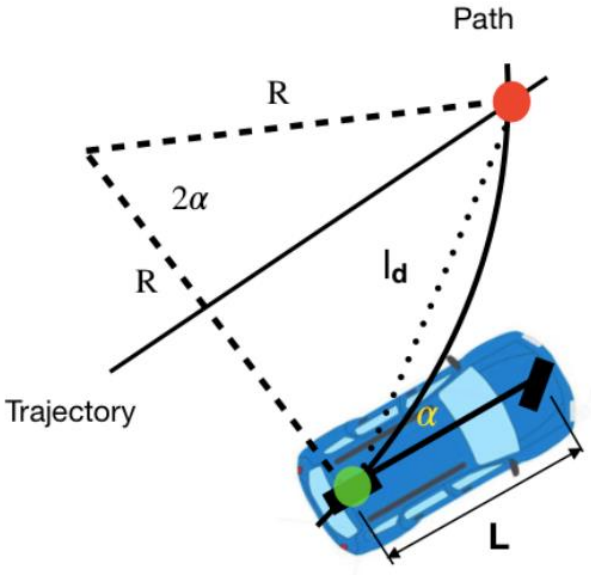

Figura 5: Esquema del algoritmo Pure Pursuit [10]

Para calcular el ángulo de dirección en *pure pursuit* es necesario recurrir al modelo de bicicleta 2D [11]. El modelo de bicicleta 2D es un modelo simplificado en el que el vehículo está compuesto por una rueda delantera, una rueda trasera y un segmento que las une. En la Figura 6 puede observarse un esquema gráfico de este.

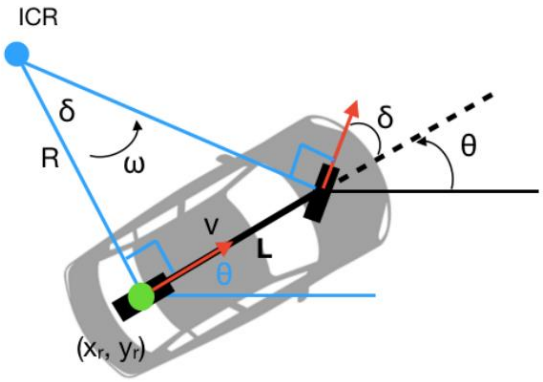

Figura 6: Esquema del modelo de bicicleta 2D [11]

De la Figura 5, aplicando el teorema del seno, puede deducirse la siguiente expresión:

$$\frac{l_d}{\sin(2\alpha)} = \frac{R}{\sin\left(\frac{\pi}{2} - \alpha\right)}$$

$$\frac{l_d}{2 * \sin(\alpha)\cos(\alpha)} = \frac{R}{\cos(\alpha)}$$

$$R = \frac{l_d}{2 * \sin(\alpha)}$$

Por el otro lado, de la Figura 6 puede deducirse esta otra expresión:

$$\tan(\delta) = \frac{L}{R}$$

Combinando ambas expresiones y despejando:

$$\delta = \text{atan}\left(\frac{2 * L * \sin(\alpha)}{l_d}\right)$$

De esta manera, en cada iteración el algoritmo *pure pursuit* actualiza el ángulo de dirección δ según esta última expresión. Existe una versión en la que se sustituye l_d por $K_{dd} * V_f$ donde K_{dd} es una constante a determinar y V_f la velocidad del vehículo.

Esto último se debe a que el control se puede volver excesivamente agresivo si se combinan una velocidad alta con un *lookahead distance* suficientemente bajo.

Existe también una implementación diseñada para vehículos con cinemática diferencial. En la Figura 7 puede observarse un esquema descriptivo.

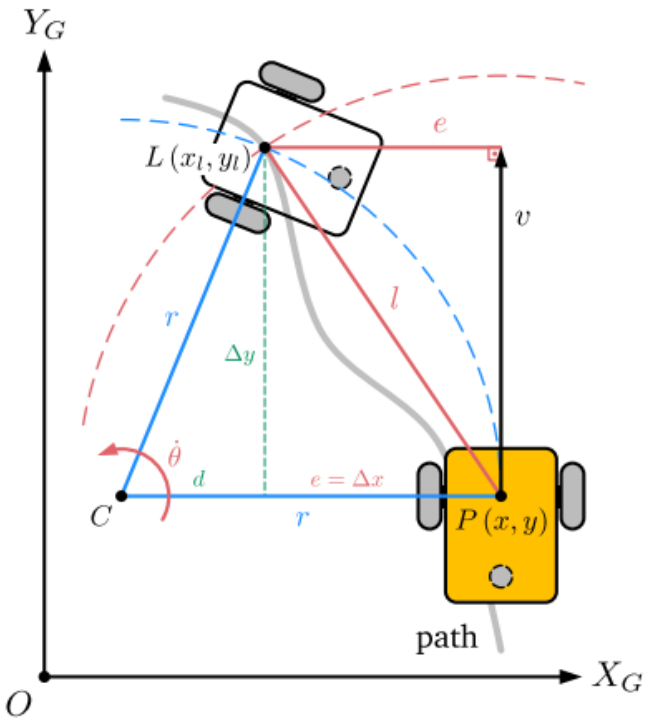

Figura 7: Esquema descriptivo del algoritmo pure pursuit en un vehículo con cinemática diferencial [12]

De la Figura 7, puede deducirse el siguiente desarrollo matemático (ignórese el signo de la velocidad angular de momento):

$$v = \omega r = \dot{\theta} r$$

$$r = d + e = d + \Delta x$$

$$r^2 = d^2 + \Delta y^2$$

$$r^2 = (r - \Delta x)^2 + \Delta y^2$$

$$r^2 = r^2 + \Delta x^2 - 2r\Delta x + \Delta y^2$$

$$2r\Delta x = \Delta x^2 + \Delta y^2$$

$$l^2 = \Delta x^2 + \Delta y^2$$

$$r = \frac{l^2}{2\Delta x}$$

$$|\dot{\theta}| = |\omega| = \frac{2v\Delta x}{l^2} = \frac{2ve}{l^2}$$

Para calcular el signo de la velocidad angular de forma correcta, se debe calcular el error transversal futuro e (nótese que este error transversal futuro e no es igual al error de trayectoria e mencionado en el algoritmo *follow-the-carrot*) mediante el ángulo de desviación (obsérvese la Figura 8).

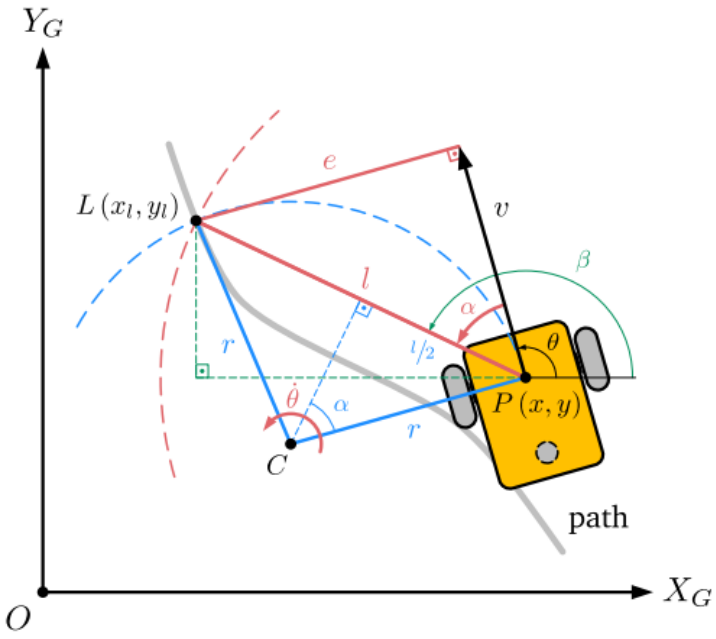

Figura 8: Esquema descriptivo del algoritmo pure pursuit con el signo correcto de la velocidad angular [12]

De la Figura 8 y el desarrollo anterior pueden deducirse las siguientes expresiones:

$$e = lsin(\alpha)$$

$$\dot{\theta} = \omega = \frac{2vsin(\alpha)}{l}$$

- *Stanley Controller*: El *Stanley Controller* es el algoritmo de seguimiento de trayectorias utilizado por el equipo de Stanford en el DARPA Grand Challenge celebrado en el 2005 [13]. Este algoritmo calcula el ángulo de dirección mediante la suma de dos términos que penalizan el error en la dirección y la distancia con respecto a la trayectoria. De esta manera, el ángulo de dirección se calcula mediante la siguiente expresión [14]:

$$\delta(t) = \varphi(t) + \text{atan}\left(\frac{K * e(t)}{K_s + v_f(t)}\right) si \left|\varphi(t) + \text{atan}\left(\frac{K * e(t)}{K_s + v_f(t)}\right)\right| < \delta_{máx}$$

$$\delta(t) = \delta_{máx}\, si\, \varphi(t) + \text{atan}\left(\frac{K*e(t)}{K_s + v_f(t)}\right) \geq \delta_{máx}$$

$$\delta(t) = -\delta_{máx}\, si\, \varphi(t) + \text{atan}\left(\frac{K*e(t)}{K_s + v_f(t)}\right) \leq -\delta_{máx}$$

Donde δ es el ángulo de dirección, $\delta_{máx}$ el máximo giro que es capaz de realizar el vehículo, φ el ángulo de desviación, e la distancia con respecto a la trayectoria, v_f la velocidad y K y K_s dos constantes a determinar. El primer término penaliza el error en la dirección, de manera que si este es grande el vehículo tenderá a girar hacia la trayectoria, y el segundo penaliza la distancia con respecto a la trayectoria, haciendo así que el vehículo gire hacia la trayectoria si esta distancia incrementa. En la Figura 9 puede verse un esquema gráfico de las variables.

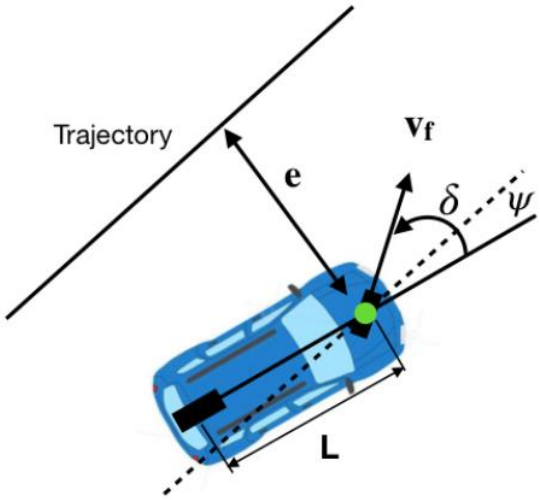

Figura 9: Esquema gráfico de variables del Stanley Controller [10]

- LQR: Aunque los controles mostrados anteriormente pueden ser diseñados sin considerar las dinámicas del vehículo, estos no garantizan seguridad en la conducción, especialmente en malas condiciones [15], por esto mismo se han dedicado muchos estudios a controladores que tienen en cuenta las dinámicas del vehículo, como el LQR o el MPC. Un regulador lineal cuadrático (LQR) es un controlador por realimentación de estado en el que los valores de las matrices de ganancias son calculados mediante la optimización de una función de costes que penaliza la desviación con respecto a los valores deseados de las variables de estado y los esfuerzos en los actuadores. Es una estrategia de control válida para sistemas MIMO (*multiple input multiple output*), por lo que puede utilizarse para seguimiento de trayectorias. Formalmente, el problema asociado al diseño de un controlador LQR puede definirse de la siguiente manera [16]:

  Se trata de un sistema lineal con múltiples entradas descrito por la ecuación $\frac{dx}{dt} = AX + BU$, donde $X \in R^n$ y $U \in R^p$. El objetivo es minimizar la función de coste cuadrático $J = \int_0^\infty (X^t Q_x X + U^t Q_u U) dt$, donde $Q_x \geq 0$ y $Q_u > 0$ son matrices simétricas, positivas de dimensiones apropiadas. Todas las variables de estado definidas en la representación de estado se suponen como incrementales con respecto

al punto de operación sobre el que se está trabajando. De esta manera, la matriz Q_x permite regular la desviación de las distintas variables de estado con respecto a su valor en el punto de operación, y la matriz Q_u el esfuerzo en los distintos mandos.

- MPC: El control predictivo, también conocido como MPC por sus siglas en inglés (*Model Predictive Control*), es una técnica de control avanzada utilizada en ingeniería y automatización. Desde su creación en la década de 1970, el Control Predictivo basado en Modelo (MPC) ha sido aplicado con éxito a procesos industriales complejos. Ha demostrado un gran potencial para abordar problemas de control de optimización con restricciones complicadas [17]. Su principio fundamental es predecir el comportamiento futuro de un sistema dinámico y calcular de manera iterativa las acciones de control óptimas para minimizar una función de costes definida. En el contexto del seguimiento de trayectorias, el MPC utiliza un modelo matemático del sistema y las trayectorias deseadas para predecir cómo se comportará el sistema en el futuro. Luego, ajusta las entradas de control de manera continua para minimizar el error entre la trayectoria deseada y la real. Esto permite que el sistema se adapte a cambios en las condiciones y siga una trayectoria deseada de manera precisa. En la Figura 10 puede observarse la estructura general del MPC.

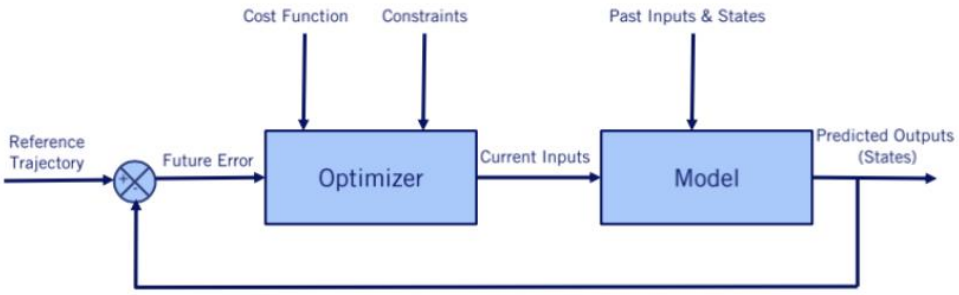

Figura 10: Estructura de un controlador MPC [10]

La estrategia de MPC se caracteriza por los siguientes pasos [18]:

- Predicción de las salidas futuras para un horizonte determinado (N) en cada instante (t), utilizando el modelo del proceso. Estas predicciones dependen de los valores conocidos hasta el instante *t* (entradas y salidas) y de las señales de control futuras que deben ser calculadas y enviadas al sistema.
- Cálculo de la secuencia de señales de control futuras, minimizando un criterio para mantener el proceso lo más cerca posible de la trayectoria de referencia. Este criterio suele ser una función cuadrática del error entre la salida predicha y la trayectoria de referencias futuras, incluyendo también el esfuerzo de control.
- Aplicación de la señal de control calculada u(t | t) al proceso. Las demás señales calculadas no se consideran, ya que en el siguiente instante de muestreo $y(t + 1)$ es ya conocida y se repiten los pasos anteriores con esta nueva información.

2.3 ALGORITMOS DE PLANIFICACIÓN DE RUTAS

Los algoritmos de planificación de trayectorias pueden ser divididos en deterministas y estocásticos. En los deterministas, para cualquiera que sea la representación del mapa elegida, el objetivo de la planificación de rutas es encontrar la mejor ruta en el grafo de conectividad del mapa entre el punto de inicio y el punto de destino, donde "mejor" hace referencia a los criterios de optimización seleccionados (por ejemplo, la ruta más corta) [2]. Sin embargo, cuando se enfrentan a problemas complejos de planificación de rutas de alta dimensionalidad, se vuelve inviable resolverlos exhaustivamente en un tiempo razonable.. En tales situaciones, la los algoritmos estocásticos se vuelven útiles, ya que sacrifican la optimalidad de la solución a cambio de un cálculo de solución más rápido [2].

2.3.1 Algoritmos deterministas

Entre los algoritmos deterministas, cabe destacar los siguientes:

- Grafos de visibilidad: Se construye un grafo en el que los vértices del grafo son los puntos iniciales y de destino, y los vértices de los obstáculos en el espacio de

configuración. Todos los vértices que son visibles entre sí son conectados. Puede observarse un ejemplo en la Figura 11.

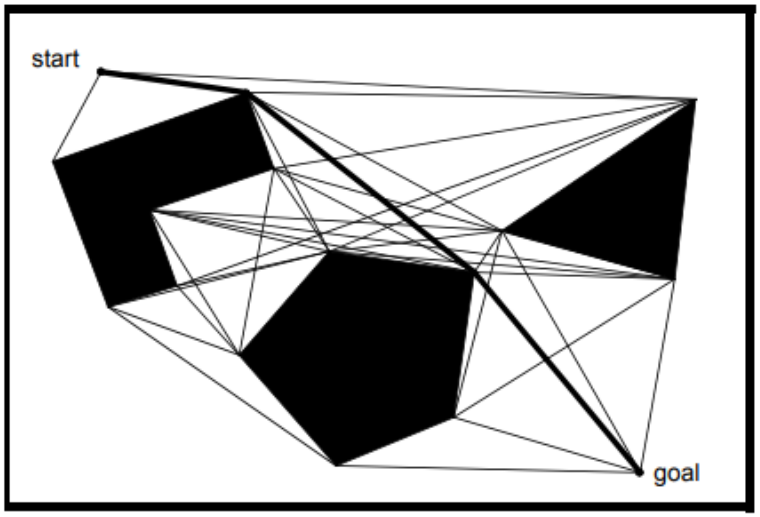

Figura 11: Ejemplo de grafo de visibilidad [2]

Nótese que este grafo considera el AMR un punto, por lo que al seguir el camino este colisionaría con los obstáculos. Esto obliga a aumentar el tamaño virtual de los obstáculos antes de aplicar el algoritmo. Una vez se ha construido el grafo, puede ser utilizado el algoritmo de Dijkstra o similar para encontrar el camino más corto.

- Diagramas Generalizados de Voronoi: A diferencia del enfoque del grafo de visibilidad, el diagrama de Voronoi es un método que busca maximizar la distancia entre el robot y los obstáculos en el mapa. El diagrama de Voronoi está compuesto por líneas que se construyen a partir de puntos que están equidistantes de dos o más obstáculos. Los puntos en el diagrama de Voronoi indican la transición desde segmentos de línea recta (donde la distancia entre dos líneas es la mínima) a segmentos parabólicos (donde la distancia mínima es entre una línea y un punto). Puede observarse un ejemplo en la Figura 12. Al igual que en los grafos de visibilidad, una vez construido el grafo puede ser utilizado el algoritmo de Dijkstra o similar para encontrar el camino más corto.

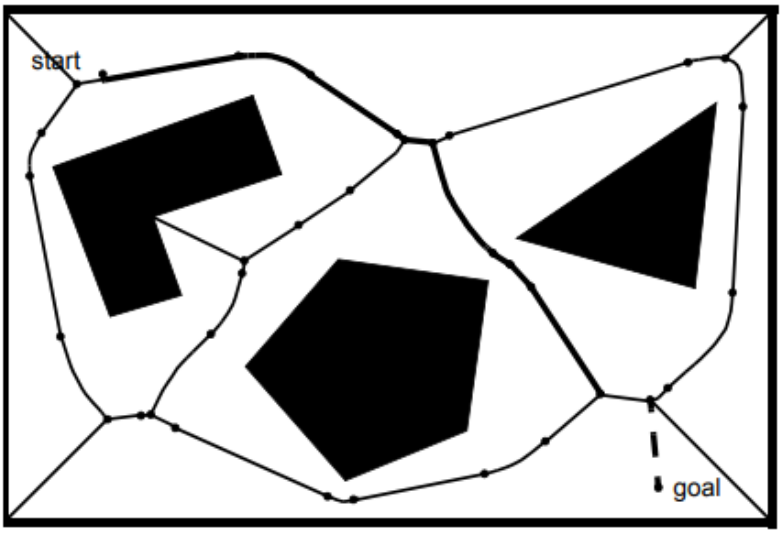

Figura 12: Ejemplo de diagrama de Voronoi [2]

- A*: El algoritmo A* es una variante del algoritmo de Dijkstra. Para comprender su funcionamiento, resulta útil abordar ciertos conceptos matemáticos. Se define $f(n)$ como el coste total (la distancia del camino recorrido) entre el punto de inicio y el destino al pasar por el vértice n, $g(n)$ como el coste acumulado entre el punto de inicio y el vértice n, y $h(n)$ como el coste estimado entre el vértice n y el destino. Así, el coste $f(n)$ se puede expresar de la siguiente manera:

  $$f(n) = g(n) + h(n)$$

  En contraste con el algoritmo de Dijkstra, que no tiene en cuenta la información $h(n)$, el algoritmo A* incorpora $h(n)$ como la distancia entre el nodo n y el destino, asumiendo la inexistencia de obstáculos en la ruta. Esto implica que, por lo general, el algoritmo A* necesita calcular la distancia a un número considerablemente menor de vértices en comparación con el algoritmo de Dijkstra [2].

  Es importante destacar que el algoritmo A* opera en un espacio discreto, dividiendo el entorno en celdas que representan los vértices del grafo que contiene la ruta a seguir.

- D*: El algoritmo D* es una versión mejorada del algoritmo A*. El algoritmo utiliza el esfuerzo de búsqueda previo en iteraciones de búsqueda posteriores. Si el AMR

encuentra cambios en el espacio a recorrer, en lugar de generar una nueva solución desde cero (como haría A*), solo se recalculan los estados afectados por las celdas de obstáculos añadidas o eliminadas. De esta manera, gran parte de la solución anterior sigue siendo válida para el nuevo cálculo. En comparación con A*, el tiempo de búsqueda puede disminuir en un factor de uno a dos órdenes de magnitud [2].

2.3.2 Algoritmos estocásticos

Estos algoritmos de carácter estocástico presentan distintas maneras de construir un grafo en el espacio a explorar. Una vez construido el grafo, es necesario aplicar algoritmos como Dijskstra o A* para encontrar un camino.

- *Probabilistic Roadmap*: El algoritmo *Probabilistic Roadmap* genera un grafo añadiendo vértices de forma aleatoria en el espacio. Cada vez que se añade un vértice este se conecta a sus vecinos más cercanos, es decir, se conecta a los *k* vecinos más próximos, siendo *k* un parámetro configurable por el usuario. En la Figura 13 puede observarse un ejemplo.

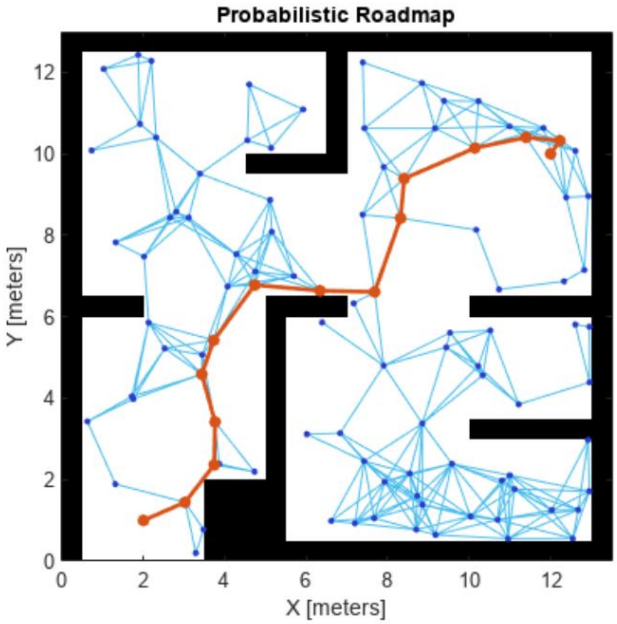

Figura 13: Ejemplo de Probabilistic Roadmap [19]

- RRT: El algoritmo RRT (*Rapidly Exploring Random Trees*) también añade vértices al espacio de forma aleatoria. Su diferencia fundamental con el *Probabilistic Roadmap* es que los vértices no son añadidos al espacio de forma completamente aleatoria, sino que en caso de encontrarse a una distancia mayor a un máximo predefinido estos se sitúan a esa distancia máxima, pero manteniendo la dirección a la que se encontraban de su vecino más cercano. Además, el nuevo vértice solamente se conecta al que era su vecino más cercano antes de ser movido a la distancia máxima. Esto hace que el grafo generado sea un árbol (el grafo generado no contiene ciclos). En la Figura 14 puede observarse un ejemplo.

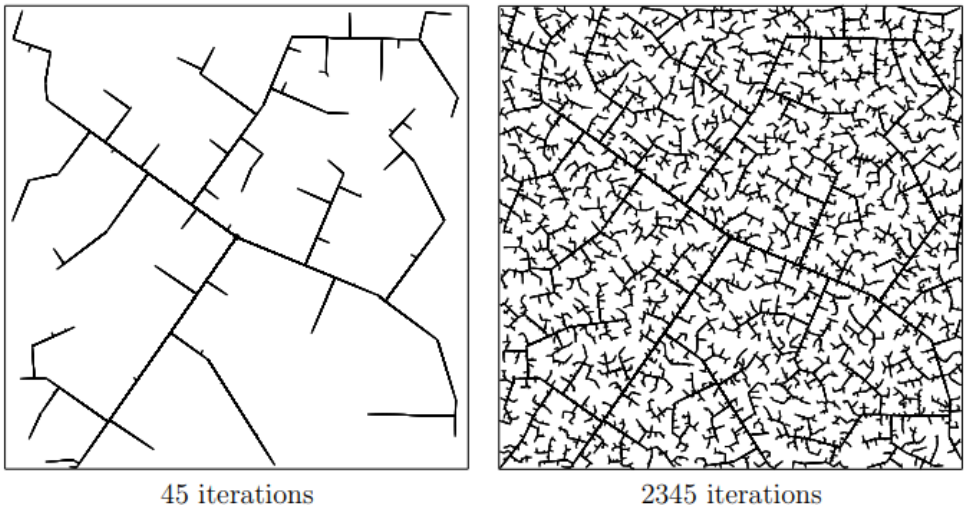

Figura 14: Ejemplo de RRT [20]

- RRT*: RRT* es una mejora del algoritmo RRT que implementa dos cambios fundamentales: Las operaciones de búsqueda de vecinos y las operaciones de reconexión [21].
    - Las operaciones de búsqueda de vecinos cercanos encuentran el mejor vértice padre para el nuevo vértice antes de su inserción en el árbol, es decir, el nuevo vértice es conectado a aquel vértice que minimiza su coste (distancia al origen).
    - Las operaciones de reconexión eliminan y crean nuevas conexiones si estas minimizan el coste de los vértices, es decir, después de añadir y conectar un

nuevo vértice se comprueba si conectar este a otros vértices reduce el coste de estos últimos. Si es así, estas conexiones se llevan a cabo.

2.3.3 Comparación de los distintos algoritmos

A continuación, se muestra una tabla comparativa de los distintos algoritmos como resumen.

Tabla 1: Comparación de los algoritmos

Algoritmo	Determinismo	Facilidad de implementación	Librerías compatibles con ROS
Grafos de visibilidad	Determinista	Moderada	Sí. VGRAPH.
Diagramas de Voronoi	Determinista	Moderada	Sí. voronoi_planner.
A*	Determinista	Fácil	Sí. path_planner.
D*	Determinista	Fácil	No (no existen librerías ampliamente testeadas y mantenidas con regularidad)
Probabilistic Roadmap	Estocástico	Fácil	Sí. OMPL [22].
RRT	Estocástico	Fácil	Sí. OMPL.
RRT*	Estocástico	Moderada	Sí. OMPL.

3. ARQUITECTURA

La Figura 15 ilustra el diseño de la arquitectura integral del sistema. El proceso inicia con el LiDAR, que transfiere los datos cruciales para la localización a través del protocolo UDP hacia una Nvidia Jetson AGX Orin. En este dispositivo, se lleva a cabo la ejecución del módulo de planificación, que tiene como función principal el cálculo de una ruta óptima utilizando los datos proporcionados por el LiDAR. Este proceso se gestiona mediante el uso de Nav2, una librería integrada en el entorno de ROS2. La ruta calculada se comunica posteriormente mediante el protocolo TCP/IP a una Raspberry Pi 4, que se encarga de ejecutar el controlador. Este último es el responsable de generar los mandos de velocidad lineal y angular. Finalmente, estas instrucciones se transmiten al sistema de control interno del vehículo a través del bus CAN.

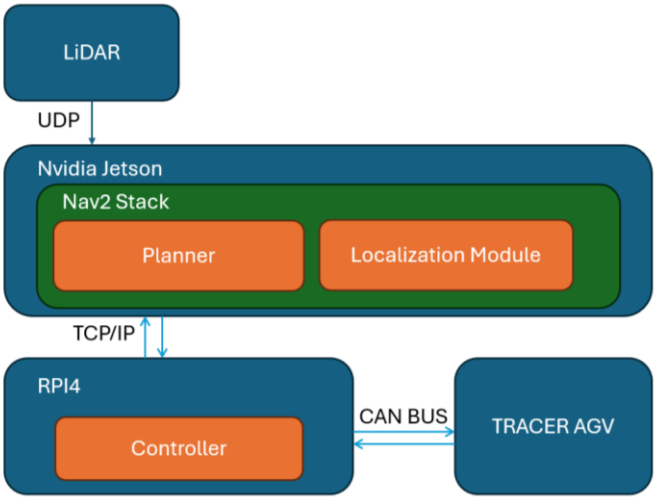

Figura 15:Arquitectura del sistema

3.1 PLATAFORMA ROBÓTICA

Como se ha detallado anteriormente en la revisión del estado de la cuestión, el proyecto ha seleccionado la plataforma robótica TRACER AGV como eje central de su arquitectura de

navegación. Este vehículo autónomo se distingue por incorporar un bus CAN integrado, facilitando la comunicación y el envío de comandos, como los mandos de velocidad lineal y angular. Este aspecto es fundamental, dado que el TRACER AGV ya incluye sistemas de control para ambas velocidades, permitiendo una gestión precisa del movimiento y la trayectoria del vehículo en tiempo real.

El TRACER AGV, seleccionado por su versatilidad y eficiencia económica comparativa, se presenta como una solución robusta y adaptable para el desarrollo de aplicaciones avanzadas en navegación y visión artificial. Equipado con capacidades que van desde la movilidad flexible en espacios interiores gracias a su diseño con cinemática diferencial, hasta la posibilidad de expansión con componentes adicionales como cámaras estéreo, radares láser, y unidades de medición inercial (IMU).

3.2 HARDWARE Y COMUNICACIONES

La configuración hardware del sistema, como se ilustra en la Figura 15, se centra en dos componentes principales: la Nvidia Jetson AGX Orin y la Raspberry Pi 4B. Estas plataformas desempeñan roles específicos dentro del esquema de navegación y control del proyecto:

- **Nvidia Jetson AGX Orin**: Este potente módulo computacional hospeda el software esencial para la ejecución de los módulos de planificación y localización, aprovechando las capacidades avanzadas del *stack* Nav2 integrado en ROS2. La Jetson AGX Orin cuenta con una GPU basada en la arquitectura NVIDIA Ampere con 2048 núcleos CUDA y 64 núcleos Tensor, lo que le permite alcanzar hasta 275 TOPS (trillones de operaciones por segundo) de rendimiento en AI. Está equipada con una CPU de 12 núcleos ARM Cortex-A78AE, y ofrece 64GB de RAM LPDDR5, lo que facilita el procesamiento de grandes volúmenes de datos en tiempo real, crucial para una navegación precisa y adaptativa en entornos dinámicos. El sistema operativo utilizado es Ubuntu 22.04 con el entorno de desarrollo NVIDIA JetPack

SDK, que incluye bibliotecas y herramientas necesarias para el desarrollo de aplicaciones de inteligencia artificial y robótica avanzada [23] [24].

- **Raspberry Pi 4B**: Esta plataforma compacta y versátil se emplea para implementar los algoritmos de control desarrollados en Simulink, utilizando Simulink Coder para la compilación de estos diseños a código ejecutable en C/C++. La Raspberry Pi 4B cuenta con un procesador de cuatro núcleos ARM Cortex-A72 a 1.5 GHz, 4GB de RAM LPDDR4, y es capaz de ejecutar sistemas operativos basados en Linux, como Raspbian. Destaca por su facilidad de integración gracias a su amplia conectividad, incluyendo puertos USB 3.0, GPIO y soporte para comunicación Ethernet y Wi-Fi. Esta capacidad de conectividad permite una comunicación efectiva entre los diferentes módulos y componentes del sistema [25] [26].

La interacción entre la Raspberry Pi y la Nvidia Jetson se establece a través de comunicaciones TCP/IP. Dos servidores TCP-IP han sido programados haciendo uso de las herramientas Simulink Coder y Python en la Raspberry Pi. Una serie de clientes TCP/IP presentes en la Nvidia Jetson AGX Orin ejecutados bajo el entorno de ROS2 solicitan la información necesaria para la correcta ejecución de los módulos de navegación y planificación y envían información sobre la posición actual y objetivo del AGV a la Raspberry Pi.

Adicionalmente, la interacción con el bus CAN del TRACER AGV se logra mediante el uso de un adaptador CAN a USB, permitiendo que la Raspberry Pi acceda y maneje los datos relativos a la velocidad lineal y angular del vehículo. Este intercambio de información se realiza a través de una API en Python y una conexión TCP/IP en *loopback*. La API de Python se encarga de comunicar los mandos de velocidad lineal y angular a través del bus CAN, mientras que la conexión TCP/IP hace posible la comunicación entre los mandos generados por el controlador programado en Matlab/Simulink y el servidor que se ejecuta en Python. Un esquema descriptivo puede observarse en Figura 16.

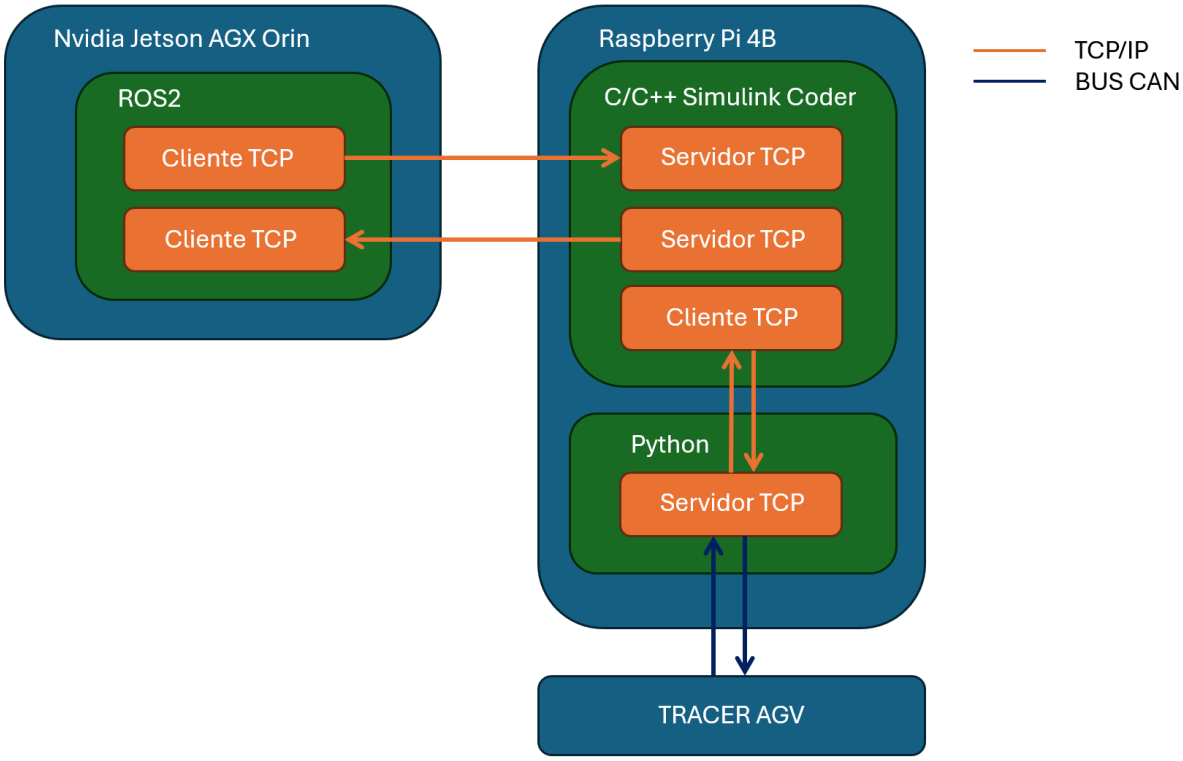

Figura 16: Esquema descriptivo de las comunicaciones

Los clientes TCP, operando bajo el entorno de ROS2, son responsables de transmitir los datos correspondientes a la pose actual y la pose objetivo, así como de recibir los datos de velocidad y velocidad angular. Como se ilustra en la Figura 16, estos interactúan con dos servidores TCP configurados mediante Simulink Coder. Adicionalmente, un cliente TCP, también configurado a través de Simulink Coder, establece comunicación con un servidor TCP implementado en Python para enviar los mandos de velocidad y velocidad angular, así como para recibir realimentación de estas variables.

3.3 ORGANIZACIÓN DEL MIDDLEWARE (ROS2)

3.3.1 Introducción a ROS2

El Robot Operating System 2 (ROS2) es un conjunto de bibliotecas y herramientas de código abierto que se utilizan para construir aplicaciones de robots. ROS2 es la evolución del sistema original ROS, diseñado para superar las limitaciones de su predecesor, ofreciendo

una mayor flexibilidad, seguridad y soporte para sistemas distribuidos en tiempo real. Este middleware facilita la programación de robots proporcionando abstracciones de hardware, controladores de dispositivos, bibliotecas y herramientas de visualización, entre otros [27].

3.3.1.1 Conceptos clave de ROS2

- **Nodos**: En ROS2, un nodo es una unidad básica de procesamiento que ejecuta un único proceso. Los nodos son responsables de realizar tareas específicas, como el control del movimiento, la recopilación de datos de sensores o la ejecución de algoritmos de planificación. Cada nodo puede comunicarse con otros nodos a través de mensajes.
- **Mensajes**: Los nodos intercambian información mediante mensajes. Un mensaje en ROS2 es una estructura de datos predefinida que se utiliza para enviar información entre nodos. Los mensajes pueden contener diversos tipos de datos, como números enteros, flotantes, cadenas de texto, y más.
- ***Topics***: Los mensajes se envían y reciben a través de *topics* (temas). Un *topic* es un canal de comunicación al que los nodos pueden suscribirse o publicar. Por ejemplo, un nodo que recopila datos de un sensor de distancia puede publicar estos datos en un *topic*, y otros nodos interesados en esta información pueden suscribirse a dicho *topic* para recibir los datos.
- **Archivos de lanzamiento (*launch files*)**: Los archivos de lanzamiento son *scripts* que permiten iniciar múltiples nodos y configurar sus parámetros de manera sencilla. Estos archivos son especialmente útiles para configurar y ejecutar aplicaciones complejas que involucran múltiples nodos y recursos.
- ***Workspaces***: Un *workspace* en ROS2 es un entorno de desarrollo donde se almacenan, compilan y ejecutan paquetes de ROS2. Los *workspaces* permiten organizar los diferentes componentes de un proyecto de robótica, facilitando la gestión del código y las dependencias. Un *workspace* típico contiene directorios como src (para el código fuente), install (para las instalaciones de los paquetes) y build (para los archivos compilados).

- **Paquetes**: Un paquete en ROS2 es la unidad básica de organización del software. Cada paquete contiene nodos, bibliotecas, archivos de configuración y otros recursos necesarios para realizar una tarea específica. Los paquetes se pueden compartir y reutilizar entre diferentes proyectos, promoviendo la modularidad y la colaboración en la comunidad de desarrollo de ROS2.

3.3.2 Workspace programado

Como se detalló en la Sección 3, se ha implementado el Middleware ROS2 en conjunto con el *stack* de Nav2 para la integración de los módulos de navegación y localización.

Se configuró un *workspace* denominado ros2_tracer_ws, que incluye los siguientes paquetes:

- lidar_bringup: Este paquete alberga archivos de lanzamiento necesarios para la operación del LiDAR. Incluye `lidar_display.launch.xml` para verificar el funcionamiento adecuado del LiDAR y `lidar.launch.xml`, que puede ser invocado por otros archivos de lanzamiento para activar el LiDAR.
- livox_ros_driver2: Proporcionado por el fabricante del LiDAR, este paquete contiene el controlador del dispositivo.
- p2l_remapper: Utilizado para asegurar la compatibilidad en la calidad de servicio entre diferentes nodos del sistema. La calidad de servicio en ROS2 se refiere a las políticas que determinan cómo se entregan los mensajes entre los nodos. Esto incluye parámetros como la fiabilidad (si los mensajes deben ser confirmados), la durabilidad (si los mensajes deben ser guardados y reenviados a los nuevos suscriptores), y el historial (cuántos mensajes deben ser guardados). El uso de este paquete se explica con mayor detalle en la sección 3.3.3.
- tracer_bringup: Empleado para iniciar todo el sistema. Contiene dos archivos de lanzamiento: `tracer_real.launch.xml` para operar todo el sistema en modo de navegación autónoma, y `tracer_real_scan.launch.xml` para realizar el escaneo de nuevos entornos.

- tracer_description: Incluye todo lo relacionado con la visualización y descripción física del robot, como archivos .stl y .urdf, que son esenciales para que otros paquetes comprendan la estructura física del robot.
- tracer_odometry: Responsable de generar la odometría del robot.
- tracer_tcp_ros_bridge: Encargado de crear y gestionar los clientes TCP/IP para la comunicación con la Raspberry Pi.
- waypoint_finder: Extrae y publica la pose actual y la pose objetivo en *topics*, permitiendo su envío a la Raspberry Pi a través del protocolo TCP/IP.
- waypoint_commander: Permite al usuario enviar una serie de poses por las que el robot pasará, respetando el orden especificado.

En la Figura 17 puede observarse el grafo resultante (conjunto de nodos y *topics*) durante la ejecución del sistema.

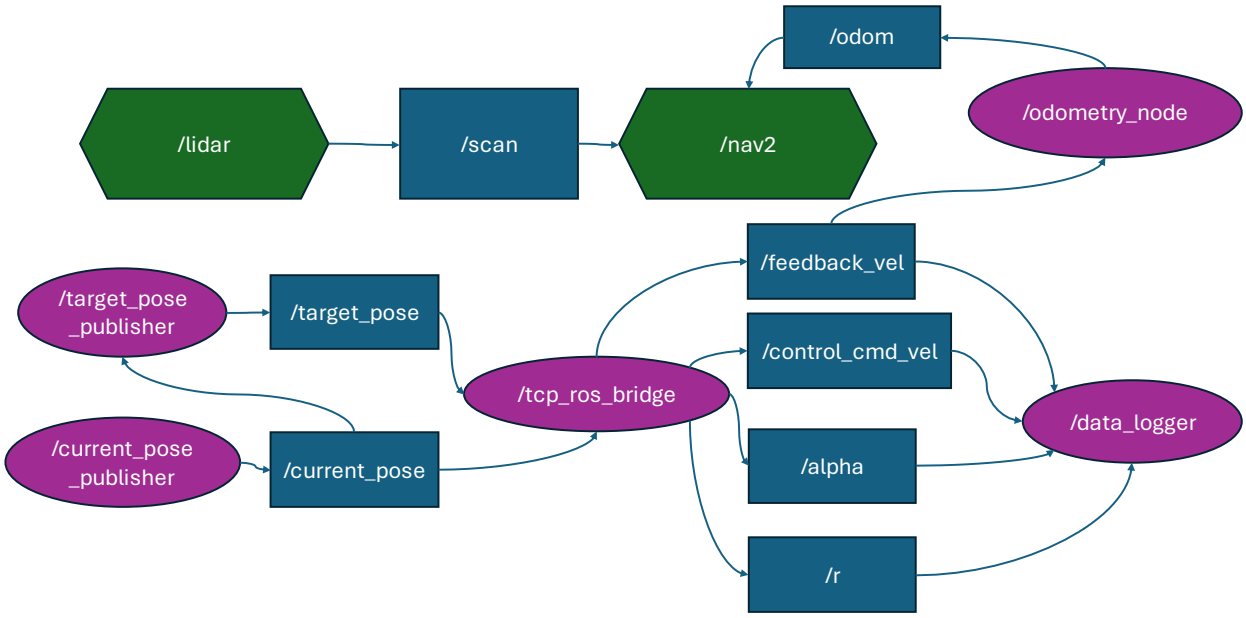

Figura 17: Grafo resultante durante la ejecución del sistema

Siguiendo la notación utilizada por el comando `rqt_graph` en ROS2, los rectángulos representan *topics* y las elipses nodos. Los hexágonos representan conjuntos de nodos y *topics* (por ejemplo, el conjunto /lidar contiene el grafo mostrado en la Figura 18).

De esta manera, el comportamiento del sistema puede resumirse de la siguiente manera:

- El nodo tcp_ros_bridge establece comunicaciones TCP/IP con la Raspberry Pi para enviar la pose actual del robot y la siguiente pose objetivo en la ruta. Además este recibe las velocidades realimentadas, los mandos de velocidad calculados por el control y la distancia y error angular hasta el siguiente objetivo. Todos estos son publicados en sus correspondientes *topics*.
- Los nodos target_pose_publisher y current_pose_publisher publican la siguiente pose objetivo en la ruta y la pose actual y lo publican en *topics*. De esta manera, el tcp_ros_bridge puede acceder a estas poses y enviarlas a la Raspberry Pi.
- El nodo odometry_node calcula una estimación de la pose actual del robot integrando la velocidad y velocidad angular. Esta estimación se publica en el *topic* /odom para que pueda ser recibida por Nav2.
- El conjunto de nodos y *topics* asociados a la correcta ejecución del LiDAR (/lidar en la Figura 17) publican los datos asociados al escaneo del entorno en el *topic* /scan. De esta manera, estos datos pueden ser recibidos por Nav2.
- El nodo /data_logger guarda toda la información publicada en los *topics* /feedback_vel, /control_cmd_vel, /alpha y /r en un archivo `.csv` para que esta pueda ser analizada, facilitando así las tareas de *debugging*.

3.3.3 Gestión de la calidad de servicio de los topics asociados al LiDAR

El LiDAR genera un *topic* con mensajes del tipo PointCloud2. Sin embargo, para este proyecto, el *stack* de Nav2 ha sido configurado para recibir mensajes del tipo LaserScan. Por esta razón, se ha hecho necesario utilizar el paquete pointcloud_to_laserscan para convertir la nube de puntos 3D del PointCloud2 en un LaserScan 2D. Durante este proceso, se identificaron problemas relacionados con la calidad de servicio de los mensajes de salida del

paquete pointcloud_to_laserscan, los cuales resultaron incompatibles con la calidad de servicio esperada por Nav2 en el topic /scan. Para resolver este inconveniente, se implementó un nodo de reasignación en el paquete p2l_remapper, tal como se detalla en la sección 3. Este nodo ajusta la calidad de servicio a las especificaciones requeridas.

La estructura resultante se ilustra en la Figura 18. En esta configuración, el *topic* /livox/lidar, generado por el controlador del LiDAR y que publica mensajes de tipo PointCloud2, es procesado por el pointcloud_to_laserscan_node para transformarlo a LaserScan y publicarlo en el *topic* /scanout. Posteriormente, el nodo de reasignación (remap) se suscribe a /scanout y publica la información en el *topic* /scan con la calidad de servicio ajustada conforme a las expectativas de Nav2.

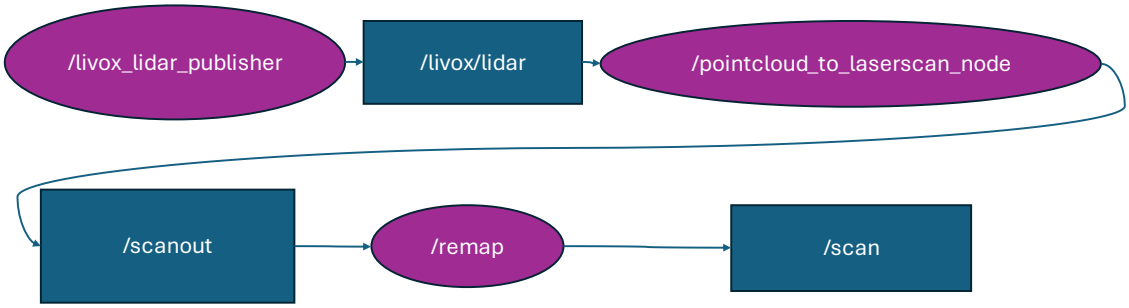

Figura 18: Esuqema resultante de la gestión de la calidad de servicio del mensaje del LiDAR

Además, en el nodo /remap se edita el mensaje generado por el LiDAR para ignorar las patas de la silla en la localización, ya que estas pueden ser identificadas como obstáculos de forma errónea.

4. SEGUIMIENTO DE RUTAS

La Figura 19 despliega la estructura detallada del sistema de control para el seguimiento de rutas. Este sistema de control procesa las entradas compuestas por las velocidades lineal y angular (z_v y z_ω), que son proporcionadas en tiempo real por el TRACER AGV, junto con las variables de estado r y α. Las mencionadas variables de estado, que representan la distancia hasta el objetivo y el ángulo de desviación con respecto a este último, respectivamente, son cruciales para el cálculo de los mandos de velocidad y velocidad angular. Estas variables requieren de una elaboración preliminar basada en la información sobre la ubicación actual del robot, su orientación, y el punto objetivo proporcionados por el módulo de localización; aunque este proceso de cálculo previo no se especifica en la visualización ofrecida por la Figura 14.

En base a estos datos, el sistema de control calcula y emite los comandos de velocidad lineal y angular (u_v y u_ω respectivamente). Estos comandos se transmiten al mecanismo robótico a través del bus CAN.

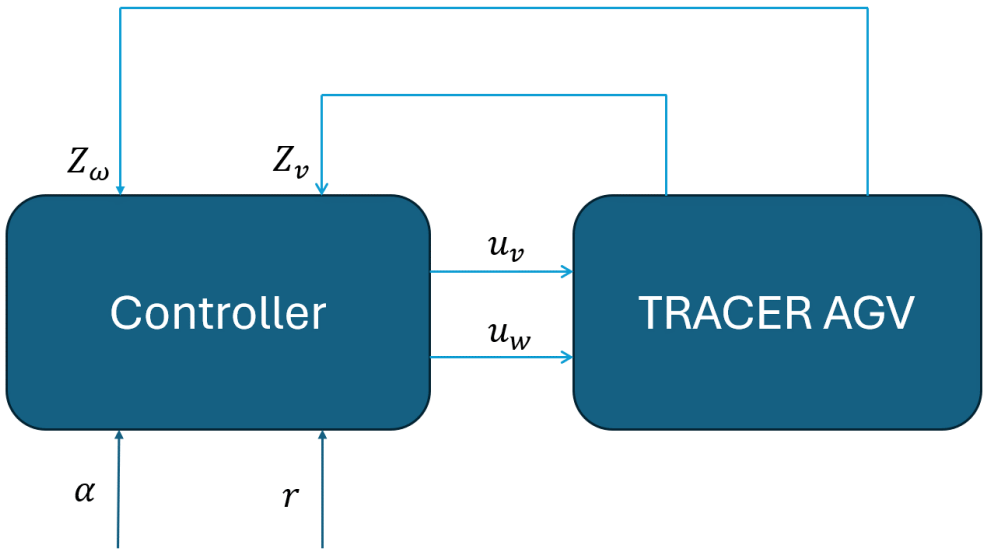

Figura 19: Arquitectura del algoritmo de seguimiento de rutas

En el desarrollo del algoritmo de seguimiento de rutas, ilustrado como *Controller* en la Figura 15, se ha procedido a la evaluación de tres metodologías: el Regulador Lineal Cuadrático (LQR), el Control Predictivo basado en Modelo (MPC) y el *pure pursuit*. La elección del LQR y el MPC se debe a su naturaleza de controladores dinámicos, los cuales, en contraste con las técnicas basadas en geometría, ofrecen una promesa significativa de mejora en el rendimiento gracias a su capacidad para integrar y procesar un espectro más amplio de información relativa al sistema [15]. Por el otro lado, el *pure pursuit* fue seleccionado por su fácil implementación.

La implementación del LQR y el MPC se fundamenta en un modelo unificado de espacio de estados que describe el comportamiento y la interacción del sistema con su entorno. Dentro de este modelo, se define α como el ángulo que indica la desviación respecto a la orientación deseada hacia el objetivo, y r como la distancia que separa al robot de dicho punto de destino. La Figura 20 ofrece una representación gráfica de este modelo, donde los ejes X e Y delinean el plano de referencia absoluto y los ejes X' e Y' se mueven con el robot, siendo X' un indicador de la dirección actual del robot.

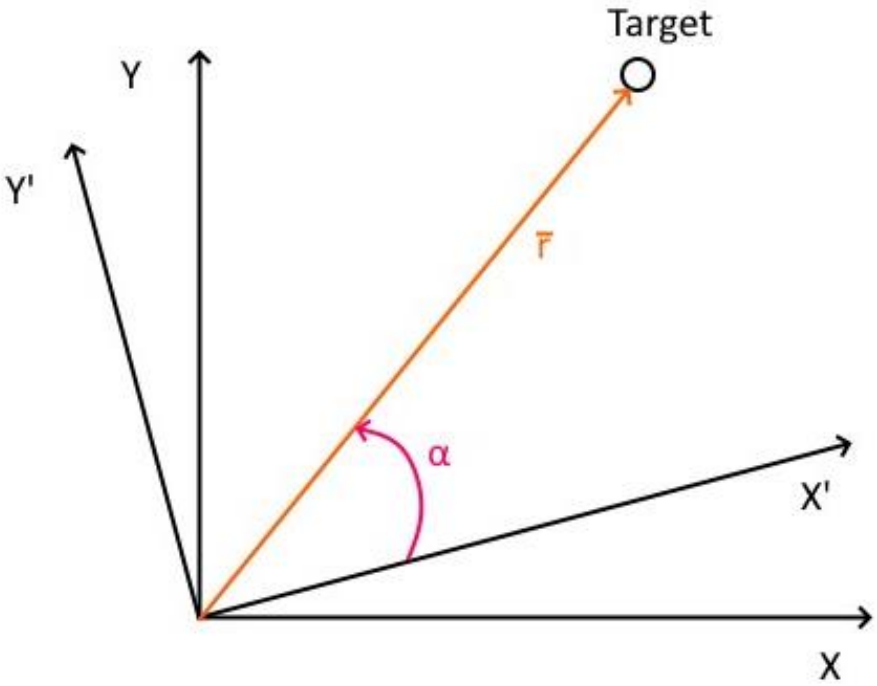

Figura 20: Esquema visual de las variables del modelo de espacio de estados

De la Figura 20, expresando los vectores como fasores, se puede llegar a las siguientes expresiones (el desarrollo completo puede observarse en el Anexo C):

$$\frac{d\alpha}{dt} = \frac{v}{r}\sin(\alpha) - \omega$$

$$\frac{dr}{dt} = -v\cos(\alpha)$$

Donde v es la velocidad lineal del robot y ω es su velocidad angular. Para completar un modelo de espacio de estados, es necesario modelar las variables v y ω, pero las matrices A, B, C y D de los modelos de espacio de estados de estas variables pueden obtenerse fácilmente de los ensayos de identificación del lazo cerrado de velocidad lineal y velocidad angular

(recuérdese que el Tracer AGV ya cuenta con controladores integrados para velocidad lineal y angular). De esta manera:

$$\frac{dX_v}{dt} = A_v * X_v + B_v * u_v$$

$$\frac{dv}{dt} = C_v * X_v + D_v * u_v$$

$$\frac{dX_\omega}{dt} = A_\omega * X_\omega + B_\omega * u_\omega$$

$$\frac{d\omega}{dt} = C_\omega * X_\omega + D_\omega * u_\omega$$

Donde las variables de estado son las utilizadas por defecto por el comando `ss` de Matlab y las matrices A, B, C y D se han obtenido de los ensayos de identificación. En la Figura 21 y la Figura 22 pueden observarse los ensayos de identificación de los controles de velocidad lineal y angular respectivamente. Además, se detallan los valores de las matrices a continuación.

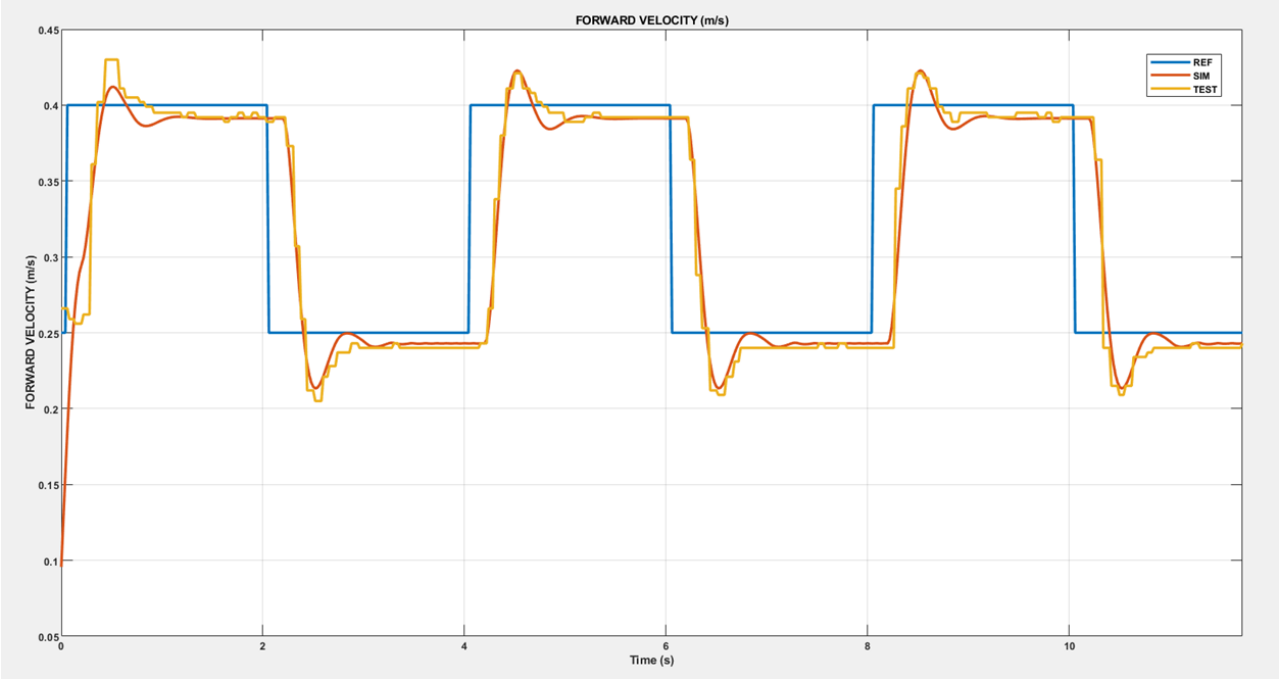

Figura 21: Resultado del ensayo de identificación del control de velocidad lineal

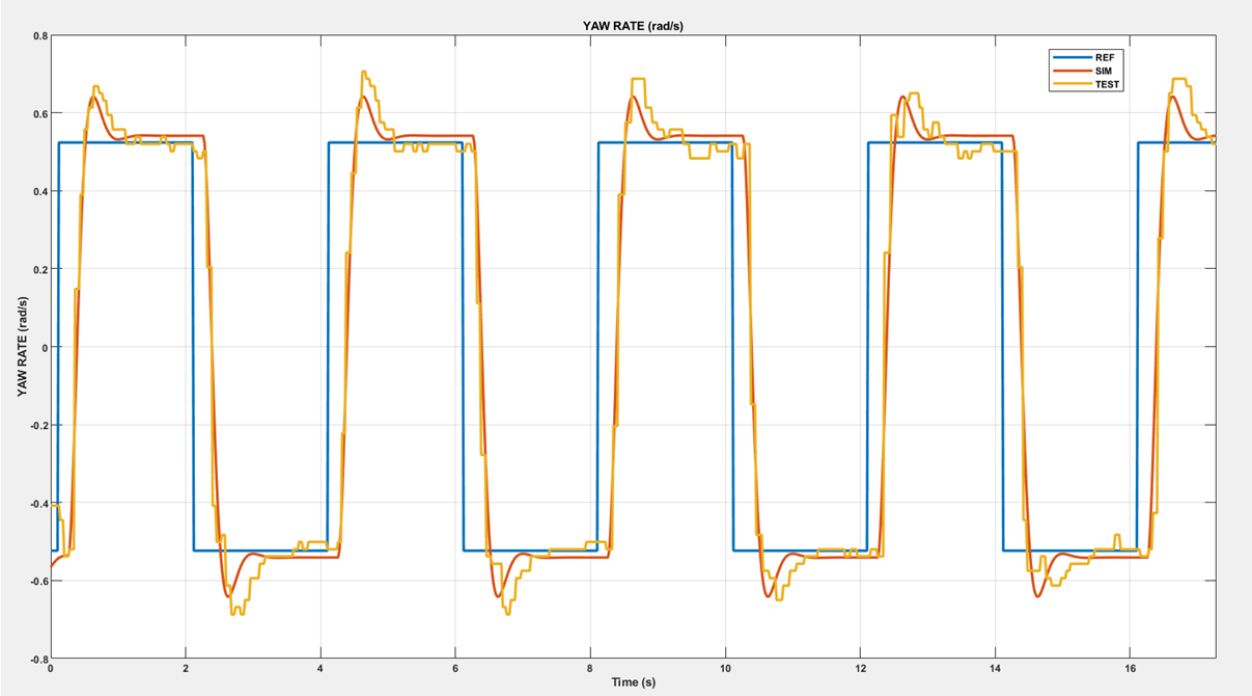

Figura 22: Resultado del ensayo de identificación del control de velocidad angular

Los valores de las matrices A, B, C y D de estos sistemas son los siguientes (tomando como variables de estado las utilizadas por defecto por el comando `ss` de Matlab):

$$A_v = \begin{bmatrix} -9.685 & -14.699 \\ 8 & 0 \end{bmatrix}$$

$$B_v = \begin{bmatrix} 4 \\ 0 \end{bmatrix}$$

$$C_v = [0 \quad 3.5945]$$

$$D_v = [0]$$

$$A_\omega = \begin{bmatrix} -12.893 & -14.148 \\ 8 & 0 \end{bmatrix}$$

$$B_\omega = \begin{bmatrix} 4 \\ 0 \end{bmatrix}$$

$$C_\omega = [0 \quad 3.6542]$$

$$D_\omega = [0]$$

4.1 LQR

Tal como se expuso en la sección 2.2, el LQR se basa en un controlador de realimentación de estado, donde la selección óptima de las matrices de ganancia se logra mediante la minimización de una función de costes cuidadosamente definida. Esta función de costes está diseñada para sancionar no solo las discrepancias entre los valores actuales y los objetivos de las variables de estado, sino también el nivel de esfuerzo exigido a los actuadores para alcanzar dichos objetivos. Este mecanismo asegura un balance entre la precisión en el seguimiento de la trayectoria deseada y la eficiencia en el uso de la energía y la preservación de la integridad mecánica del sistema.

La estructuración y los parámetros específicos de esta función de coste, que son cruciales para el ajuste fino del controlador LQR y su rendimiento general, se detallan con mayor profundidad en la sección 2.2.

Matlab/Simulink proporciona una plataforma versátil para modelar, simular y analizar algoritmos de control complejos, por lo que se ha utilizado esta herramienta para diseñar este controlador y el MPC y después se ha trasladado a la Raspberry Pi mediante el Simulink Coder (herramienta que permite compilar modelos de Simulink a código en C/C++ ejecutable en la Raspberry Pi).

4.1.1 Implementación en Matlab/Simulink

La implementación del controlador LQR en el entorno de Matlab/Simulink se inspira directamente en la metodología aplicada a los robots utilizados en el laboratorio correspondiente a la asignatura de Control Avanzado del máster. Este enfoque contempla la utilización de 500 puntos de operación distintos, estableciendo un diseño de control LQR específico para cada uno mediante el uso del comando `dlqr` de Matlab. Dicho comando se encarga de calcular la matriz de ganancias óptimas K, con el propósito de minimizar la función de costes J delineada previamente en la sección 2.2, abordando el cálculo bajo la perspectiva de un horizonte de tiempo infinito. Este procedimiento da lugar a un esquema de control adaptativo que ajusta las matrices de ganancia en función del punto de operación actual, lo cual constituye una estrategia de programación de ganancia.

4.2 MPC

Como se ha comentado en la revisión del estado de la cuestión, el controlador MPC es una estrategia de control cuyo principio fundamental reside en predecir el comportamiento futuro de un sistema dinámico y calcular de manera iterativa las acciones de control óptimas para minimizar un objetivo definido. El MPC hace uso de una función de costes cuadrática como criterio para encontrar los mandos óptimos. La función de costes general de un controlador MPC tiene la siguiente forma:

$$J = \sum_{i=1}^{N} w_{x_i} (r_i - x_i)^2 + \sum_{i=1}^{N} w_{u_i} \Delta u_i{}^2$$

Donde x_i son las variables controladas (en este caso r y α), r_i las referencias (en este caso 0 tanto para r como para α), w_{xi} parámetros ajustados para penalizar la desviación de las variables controladas, Δu_i la variación entre muestreos de las variables de control (en este caso v y ω) y w_{ui} parámetros ajustados para penalizar la variación agresiva de las variables de control. De esta manera, para implementar el controlador MPC de seguimiento de rutas, es necesario determinar 5 parámetros: $w_{r,}, w_\alpha, w_v, w_\omega$ y el tiempo de muestreo (T_s). También es necesario definir los horizontes de control y predicción, pero estos se han fijado a un valor constante por su gran correlación con el periodo de muestreo, como se detalla en la sección 4.4.1.2.

4.2.1 Implementación en Matlab/Simulink

Para la implementación en Simulink del controlador MPC, se ha usado el bloque de controlador MPC no lineal (NLMPC) proporcionado por Simulink. Nótese que el controlador ha de ser no lineal porque el modelo de variables de estado contiene ecuaciones no lineales (en concreto las derivadas de r y α). Puede observarse en la Figura 23 la implementación en Simulink.

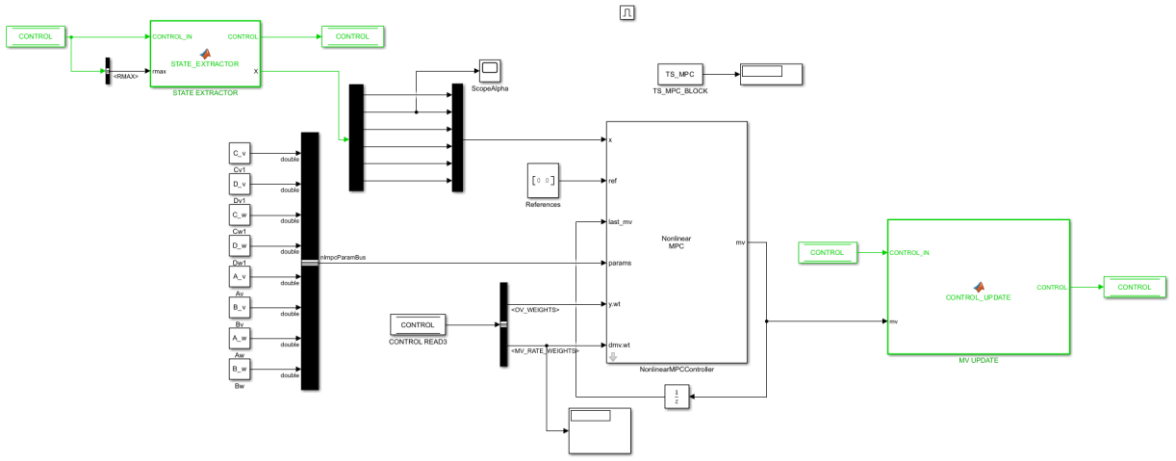

Figura 23: Implementación del MPC en Simulink

Las variables de estado se extraen del bus CONTROL utilizando el bloque 'STATE EXTRACTOR'. Posteriormente, estas variables se introducen en el bloque del controlador MPC no lineal. La variable α, que desempeña un papel fundamental en la optimización de parámetros del controlador que será expuesta más adelante, se aísla en esta etapa. En conjunción, se suministran las matrices representativas del modelo de estado para la velocidad lineal y angular, complementadas por los pesos asignados a las desviaciones de las variables controladas (w_r y w_α) y aquellos aplicados a mitigar las variaciones agresivas en las acciones de control (w_v y w_ω).

El núcleo del sistema, el bloque del controlador MPC no lineal, ejerce su función calculando y emitiendo las órdenes de control. Estas órdenes son procesadas por el bloque 'MV UPDATE', que se encarga de actualizar los valores de las variables de control para su ejecución. Es imperativo que el bloque del controlador tenga acceso a un objeto 'nlmpc', cuya estructura encapsula el número de variables de estado, variables controladas y salidas, entre otros aspectos esenciales para la operatividad del sistema de control. Dicho objeto debe ser previamente inicializado en Matlab, garantizando así la preparación adecuada para su integración y funcionamiento dentro de Simulink, tal como se ilustra en la Figura 24.

```
%%-----------------------------------------------
%% NON-LINEAR MPC DEFINITION
%%-----------------------------------------------
nx = 6;
ny = 2;
nu = 2;
nlmpc_obj = nlmpc(nx, ny, nu);
nlmpc_obj.PredictionHorizon = 20; % 15
nlmpc_obj.ControlHorizon = 5;
nlmpc_obj.Model.IsContinuousTime = true;
nlmpc_obj.Weights.ManipulatedVariables = [0, 0];
maxMVs = [0.5, 1];
minMVs = [0, -1];
nlmpc_obj.Model.NumberOfParameters = 8;
nlmpc_obj.Model.StateFcn = "StateFunction";
nlmpc_obj.Model.OutputFcn = "OutputFunction";
nlmpc_obj.Jacobian.StateFcn = "StateJacobian";
nlmpc_obj.Jacobian.OutputFcn = "OutputJacobian";

for j = 1:nu
    nlmpc_obj.MV(j).Min = minMVs(j);
    nlmpc_obj.MV(j).Max = maxMVs(j);
end

%Save MPC Controller
NLMPC = nlmpc_obj;
```

Figura 24: Inicialización del objeto nlmpc

4.2.2 Optimizador del MPC

Como se ha mencionado en la sección 4.2.1, el controlador MPC efectúa una optimización en tiempo real para calcular los mandos. El eje de esta optimización es el algoritmo seleccionado, que en este caso es el optimizador no lineal estándar de Matlab, conocido como fmincon, que implementa el método de Programación Cuadrática Secuencial (SQP).

El algoritmo de Programación Cuadrática Secuencial (SQP) es un método efectivo para la optimización de problemas no lineales con restricciones, abordando tanto restricciones de igualdad como de desigualdad. Este procedimiento se fundamenta en la solución de un subproblema de programación cuadrática en cada iteración, el cual modela el problema original en el punto actual mediante una aproximación cuadrática de la función objetivo y una linealización de las restricciones. Esta estrategia iterativa se ajusta progresivamente

hacia la solución óptima del problema no lineal original, manejando las restricciones de manera eficiente a través de las aproximaciones sucesivas [28].

4.3 PURE PURSUIT

Como se detalló en la sección 2.2, el algoritmo *pure pursuit* genera una trayectoria circular que conecta la posición actual del robot con un punto específico de la ruta que se encuentra a una cierta distancia hacia adelante, conocida como distancia de anticipación o *lookahead distance*. Se trata de un controlador geométrico con una fácil implementación (se ha implementado en Simulink haciendo uso de un código bastante parecido al pseudocódigo mostrado a continuación).

Para este proyecto se ha programado una versión del algoritmo *pure pursuit* que hace uso de una velocidad variable. La velocidad se calcula en función de la distancia al siguiente punto objetivo r y del error de desviación α (obsérvese la Figura 20). Puede observarse un pseudocódigo del algoritmo a continuación:

```
def purepursuit(r, alpha, k1, k2, k3, sat_v, sat_w, max_alpha_move):
    """
    Pure pursuit controller.

    :param r: Distance to next waypoint
    :param alpha: Angle to next waypoint
    :param k1: Gain for v
    :param k2: Gain for v to adjust speed in curves
    :param k3: Gain for w
    :param sat_v: Maximum speed
    :param sat_w: Maximum angular velocity
    :param max_alpha_move: Maximum angle at which the robot can move
    :return: v and w commands
    """

    v = k1 * r - k2 * abs(alpha)
    w = k3 * alpha

    if v < 0:
        v = 0
    elif v > sat_v:
        v = sat_v

    if w < -sat_w:
        w = -sat_w
    elif w > sat_w:
```

```
        w = sat_w

    if alpha > max_alpha_move:
        v = 0

    return v, w
```

Como puede observarse en el pesudocódigo, el algoritmo actualiza los mandos de velocidad y velocidad angular en función de los errores de ángulo y de posición. Primero se calcula el mando de velocidad lineal en función de la distancia al punto objetivo y del error angular (mayor velocidad a mayor distancia y menor velocidad a mayor error angular, para así garantizar un correcto trazado de las curvas). Después se calcula el mando de velocidad angular en función del ángulo de desviación (mayor velocidad angular a mayor error). A ambas variables saturan en un valor máximo para garantizar que estas no toman valores excesivamente grandes, y el mando de velocidad se corrige a 0 si el error angular es demasiado grande, garantizando así un seguimiento correcto de la ruta en las curvas. Además, el mando de velocidad también satura en un mínimo de 0, garantizando así que el vehículo no se mueva marcha atrás.

4.3.1 Implementación en Matlab/Simulink

El controlador *pure pursuit* también ha sido implementado en Matlab/Simulink con un código parecido al pseudocódigo mostrado en la sección anterior. Sin embargo, este se ha implementado con dos conjuntos de parámetros posibles aplicando una estrategia de programación de ganancia. Existe un conjunto de parámetros agresivo que es el que el controlador utiliza por defecto, y cuando el robot está a menos de una distancia predefinida del objetivo final, el controlador cambia a un conjunto de parámetros pasivo. De esta manera, se puede garantizar que el robot no se acerque a una velocidad excesivamente grande al objetivo final, impidiendo la parada en este punto por no contar con una distancia de frenada suficiente.

Además, si el conjunto de parámetros utilizado es el pasivo, la variable r satura en un mínimo, garantizando así que el acercamiento al punto final es suficientemente rápido. Si no

se aplica esta saturación, el vehículo reduce su velocidad de forma significativa cuando se está acercando al punto objetivo, llevándole bastante tiempo alcanzarlo.

4.4 RESULTADOS PRELIMINARES

4.4.1 Optimización mediante Algoritmos Genéticos

Los parámetros asociados a las funciones de costes de los controladores LQR y MPC han sido optimizados mediante algoritmos genéticos. Los algoritmos genéticos son herramientas poderosas en el campo de la optimización y la búsqueda de soluciones a problemas complejos en diversas disciplinas. Estos algoritmos se han utilizado eficazmente tanto en aplicaciones de ciencia como ingeniería para resolver problemas prácticos, como en modelos computacionales de sistemas evolutivos naturales. Su utilidad se extiende más allá de los límites estrictos de la computación para incluir teoría de sistemas dinámicos, teoría de juegos, biología molecular, ecología, genética poblacional y más [29].

El funcionamiento de los algoritmos genéticos se basa en la selección, combinación y mutación de individuos dentro de una población de posibles soluciones a un problema dado, con el objetivo de encontrar la solución óptima o satisfactoria a lo largo de generaciones sucesivas.

En la optimización tanto del controlador LQR como del MPC, la función de coste se ha definido como la suma acumulativa de los errores en las coordenadas X e Y a lo largo de una trayectoria predefinida. Esta trayectoria, que comprende un desplazamiento desde el origen (0, 0) hasta el punto (-1, 0), ha sido seleccionada por su relevancia práctica, ya que representa un escenario complejo que implica un giro completo de 180 grados, una maniobra complicada en entornos de navegación autónoma.

La función de coste, *J*, se expresa matemáticamente como:

$$J = \int_0^T |x - x_{ref}| + |y - y_{ref}| \, dt$$

donde x_{ref} y y_{ref} corresponden a las coordenadas de referencia en el punto objetivo (para este ensayo -1 y 0 respectivamente). Es importante resaltar que en la práctica, la evaluación de *J* no requiere de la integración continua; en su lugar, se emplea una suma discreta donde los errores se calculan y se acumulan en intervalos regulares correspondientes a cada tiempo de muestreo.

4.4.1.1 Optimización del LQR

Para la optimización del LQR se ha aplicado el algoritmo genético a conjuntos de 10 parámetros de manera que cada potencial solución generada por el algoritmo comprende un conjunto de estos parámetros, que incluye las 8 componentes de la diagonal de la matriz Q y los 2 elementos de la diagonal de la matriz R.

Los valores dentro de la matriz Q se distribuyen de tal manera que asignan ponderaciones específicas a diferentes variables de estado del sistema: dos corresponden a las variables de estado de la velocidad lineal, otros dos a la velocidad angular, dos a las variables de estado r y α y otros dos a las desviaciones de estas últimas. Estos errores de desviación representan la diferencia entre el valor actual y el valor de referencia deseado, y son cruciales para dirigir el comportamiento del controlador hacia el cumplimiento de la trayectoria objetivo.

Por su parte, la matriz R juega un papel estratégico al regular la intensidad de los comandos de control, asignando pesos a la agresividad de las entradas de velocidad lineal y angular. Pueden observarse las matrices Q y R a continuación:

$$Q = \begin{pmatrix} w_{v1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & w_{v2} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & w_{\omega 1} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & w_{\omega 2} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & w_r & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & w_\alpha & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & w_{ri} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & w_{\omega i} \end{pmatrix}$$

$$R = \begin{pmatrix} w_v & 0 \\ 0 & w_\omega \end{pmatrix}$$

4.4.1.2 Optimización del MPC

En la optimización del controlador MPC mediante algoritmos genéticos, se ha focalizado la atención en la modulación de cinco parámetros esenciales: los pesos asignados a las variables controladas de distancia r y ángulo de desviación α, los pesos correspondientes a los actuadores de velocidad v y de giro ω, y el intervalo de muestreo Ts. Estos parámetros son de vital importancia pues dictaminan la efectividad con la que el MPC responde a las dinámicas del sistema y alcanza los objetivos de control.

El horizonte de control y el horizonte de predicción, elementos intrínsecos en la configuración del MPC, se han establecido en 5 y 20 muestras respectivamente. Esta elección estratégica permite una optimización más ágil y eficaz, al reducir la complejidad que conllevaría la calibración simultánea del periodo de muestreo junto con la longitud de los horizontes (son parámetros correlacionados).

4.4.2 Simulaciones

4.4.2.1 Simulación del LQR

Para el LQR no ha sido necesaria una ejecución larga del algoritmo genético, puesto que las matrices Q y R propuestas como solución inicial ya otorgaron una solución bastante buena (obsérvese la Figura 25). La desviación máxima es de 0.072 m en el eje Y y la posición objetivo se alcanza en menos de 10s.

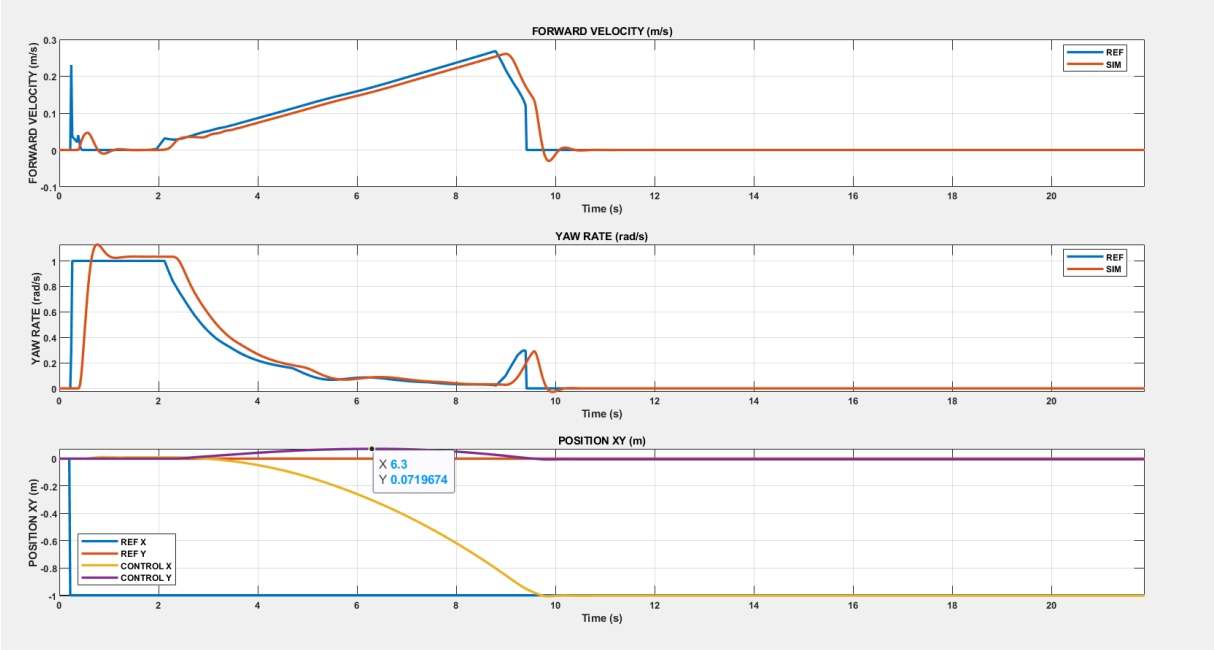

Figura 25: Simulación con el controlador LQR

4.4.2.2 Simulación del MPC

El controlador MPC ha necesitado periodos de optimización bastante más largos que los del LQR para alcanzar una respuesta suficientemente buena. Esto se debe a que el MPC trabaja con tiempos de muestreos mucho más grandes (el MPC necesita realizar una optimización en tiempo real en cada iteración) lo que hace que en sistemas rápidos como este sea necesario editar bastante sus parámetros. Además, las simulaciones son considerablemente más lentas. En la Figura 26 puede observarse una simulación con el controlador MPC.

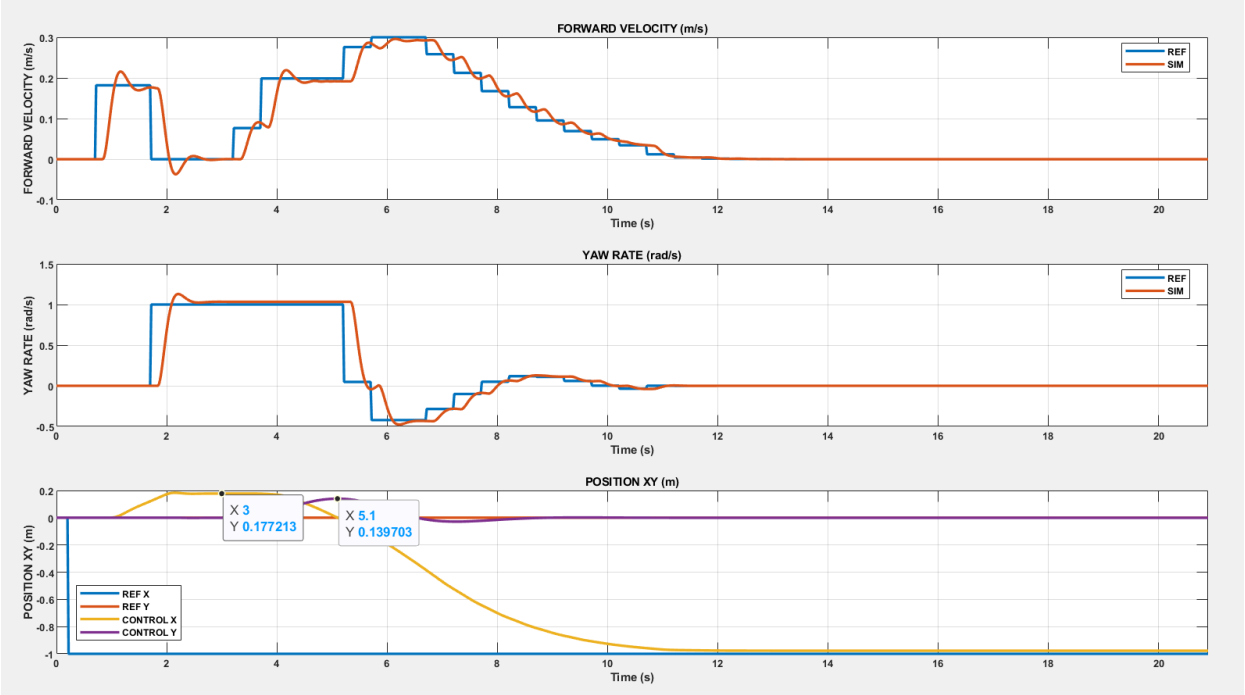

Figura 26: Simulación con el controlador MPC

Puede observarse que la respuesta cuenta con desviaciones mayores que las del controlador LQR. Además, el tiempo de alcance es mayor. Teóricamente, este controlador podría dar una respuesta mejor al LQR, pero se abandonó el proceso de optimización porque la Raspberri Pi no cuenta con potencia computacional suficiente como para ejecutarlo en tiempo real.

4.4.2.3 Simulación del *pure pursuit*

El controlador basado en el algoritmo *pure pursuit* ha sido optimizado sin usar algoritmos genéticos, ya que su simplicidad y su poca cantidad de parámetros lo permiten. Se ha obtenido una respuesta con un sobrepaso máximo en el eje Y de 0.063 m y un tiempo de alcance menor a los 8 segundos. Además de ser más rápido y contar con un sobrepaso ligeramente inferior al LQR, este cuenta con una ventaja adicional: la suavidad de los mandos. Si se comparan los mandos de la Figura 25 con los de la Figura 27 se puede observar que el LQR tiende a hacer un uso de los mandos mucho más enérgico que el *pure pursuit*.

Esto ha llevado a que el comportamiento del *pure pursuit* sea bastante mejor que el del LQR en el robot real, por lo que para la solución final se ha optado por el *pure pursuit*.

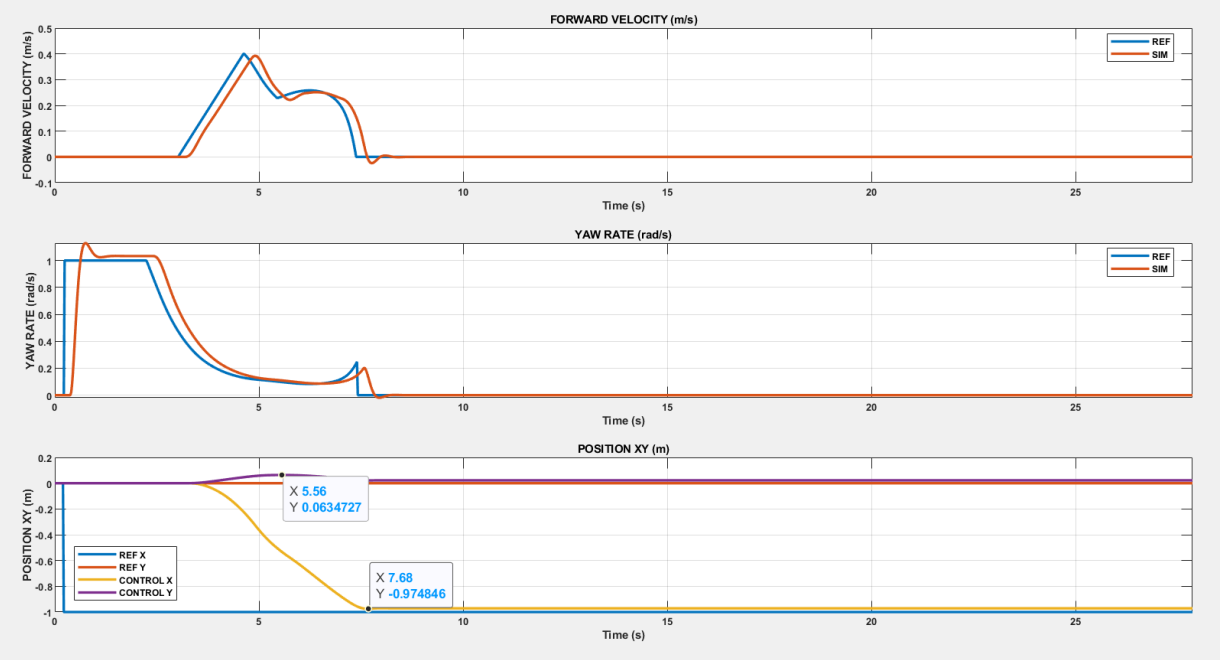

Figura 27: Simulación del controlador Pure Pursuit

5. PLANIFICACIÓN DE RUTAS

La Figura 28 presenta la configuración detallada del módulo de planificación. Dicho módulo está diseñado para procesar tres tipos de información esencial: la ubicación actual, el destino final deseado y el mapa del entorno. Integrando estas entradas, el planificador tiene la tarea de generar una trayectoria factible y segura, definida como una ruta que el robot puede seguir sin incurrir en colisiones con obstáculos existentes en su entorno.

Para lograr este objetivo, se han evaluado dos enfoques distintos para el planificador. El primero utiliza el algoritmo RRT*, conocido por su eficacia en la exploración rápida del espacio de búsqueda y su capacidad para encontrar caminos en entornos complejos. El segundo enfoque se basa en el planificador predeterminado incluido en el *stack* Nav2, que tiene la flexibilidad de emplear tanto el algoritmo de Dijkstra como el A* para la generación de rutas. La selección entre estos algoritmos depende de la configuración específica de los parámetros, lo que permite una adaptabilidad a diferentes requisitos de planificación y complejidades del entorno.

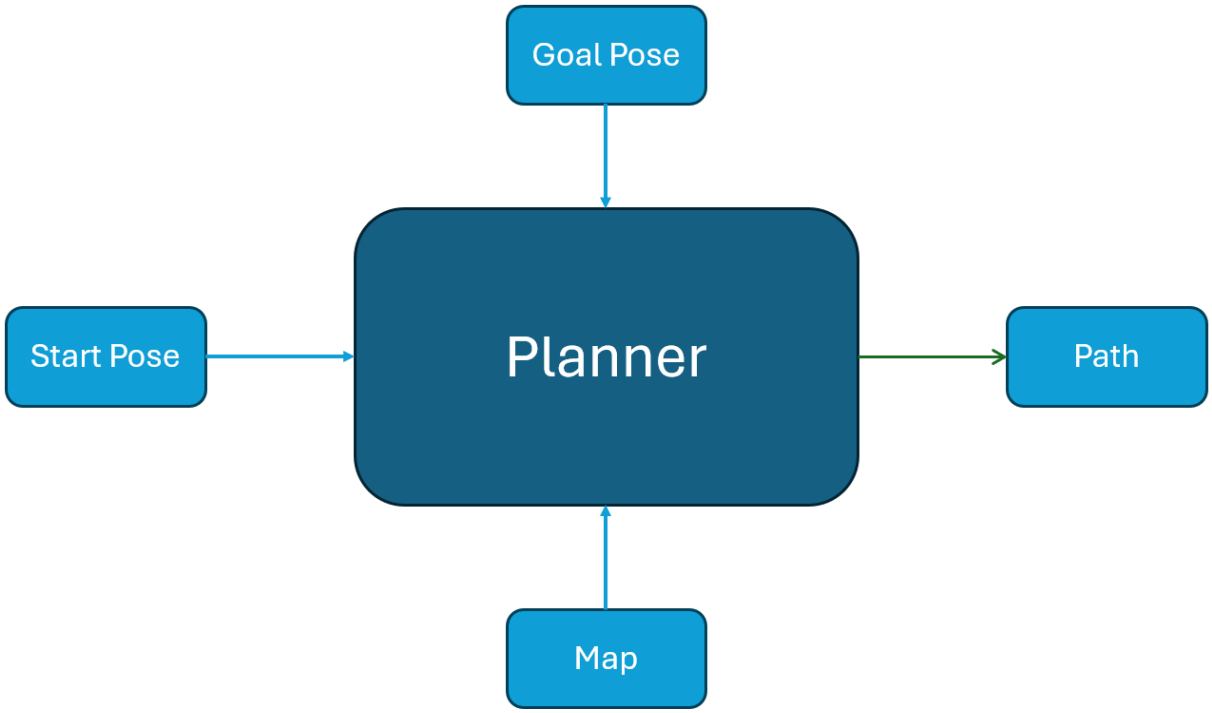

Figura 28: Arquitectura del planificador

5.1 RRT*

Tal y como se analiza en la sección 2.3.2, el algoritmo RRT* mejora la metodología RRT original incorporando dos innovaciones sustanciales: las operaciones de búsqueda de vecinos cercanos y las de reconexión de nodos. Estas innovaciones están orientadas a optimizar el camino generado, resultando en trayectorias que no solo son más directas sino también más fluidas.

Inicialmente, las rutas producidas por la primera versión del RRT* mostraban una tendencia hacia la complejidad y falta de fluidez, con una densidad de puntos que podía complicar su implementación práctica, tal como se evidencia en la Figura 29. Ante este desafío, se integraron dos etapas adicionales de refinamiento de trayectoria: la primera, un procedimiento de simplificación mediante el algoritmo de Douglas-Peucker [1], y la segunda, un proceso de suavización que utiliza técnicas de descenso de gradiente.

El algoritmo de Douglas-Peucker actúa como un filtro, reduciendo el número de puntos mientras preserva la esencia geométrica del camino. Por otro lado, el método de suavización por descenso de gradiente perfecciona la ruta, eliminando irregularidades y facilitando el movimiento del robot. La configuración mejorada del planificador, que combina estos procesos, se detalla visualmente en la Figura 30.

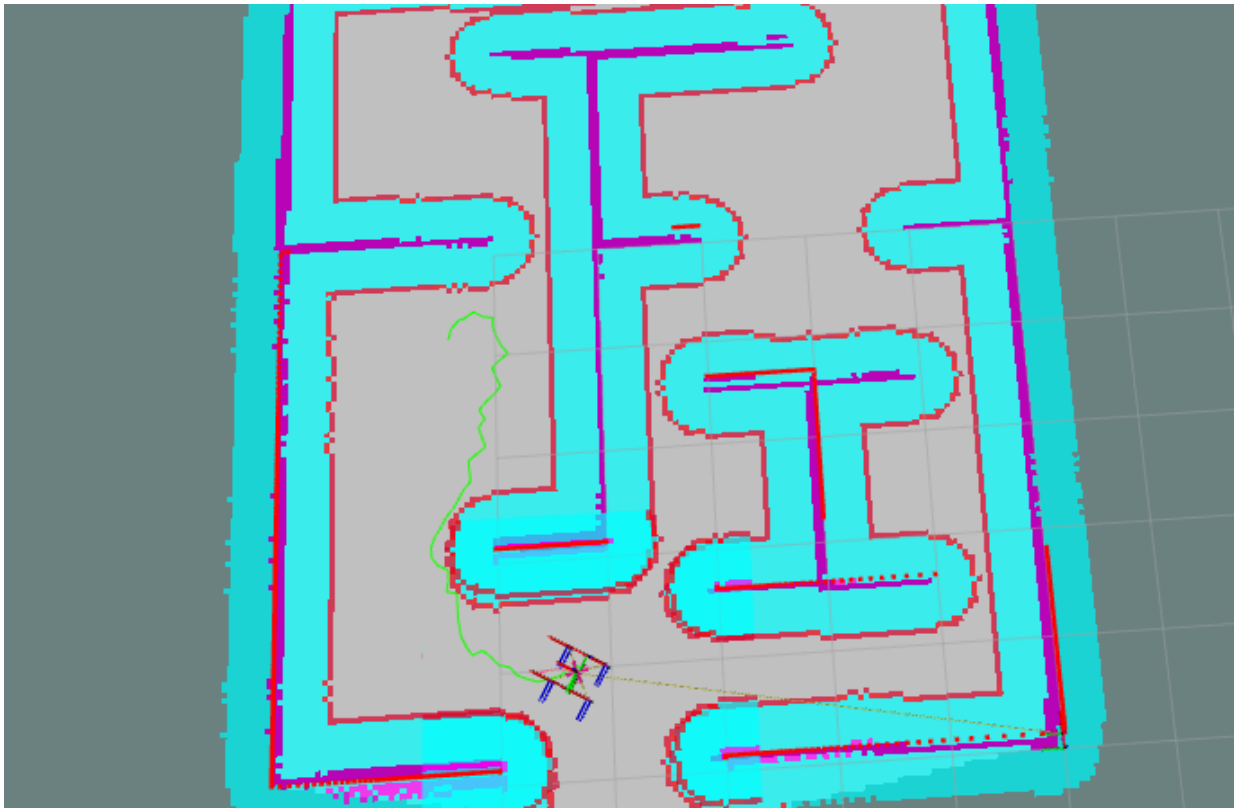

Figura 29: Ruta resultante en la implementación inicial

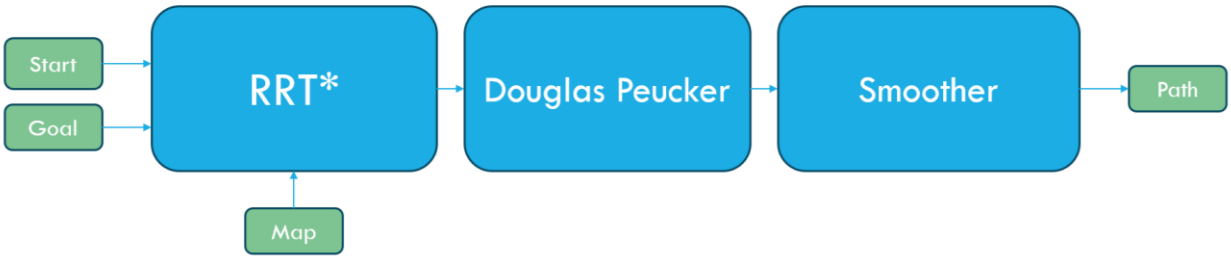

Figura 30: Diagrama del planificador con la simplificación y suavización

El bloque inicial del planificador integra el algoritmo RRT* tal como se detalla en la sección 2.3.2, desplegando su capacidad para trazar una ruta preliminar. Esta ruta se transfiere al siguiente módulo, encargado de refinarla aplicando el método de Douglas-Peucker.

El algoritmo de Douglas-Peucker divide la ruta de manera recursiva. Al principio, se le asignan todos los puntos existentes entre el primer y último punto, marcando automáticamente el primero y el último como puntos a conservar. Luego identifica el punto que se encuentra a mayor distancia del segmento definido por los puntos extremos. Si este punto se halla a una distancia menor que ε del segmento, entonces todos los puntos no marcados para conservar son descartados. Por el otro lado, si el punto más alejado del segmento de línea se encuentra a una distancia mayor que ε de la aproximación, entonces ese punto debe ser conservado. El algoritmo se llama a sí mismo de forma recursiva, primero con el punto inicial y el más lejano, y después con el punto más lejano y el final, marcando al punto más alejado como conservado. Una vez completada la recursión, se puede generar una nueva curva de salida que consista únicamente en aquellos puntos que han sido marcados para conservarse [30]. Puede observarse parte de la implementación en C++ en el Anexo B.

Finalmente, la ruta ya simplificada entra en la etapa final de refinamiento en el bloque denominado 'smoother', donde se aplica un método de descenso de gradiente para pulir y suavizar la trayectoria. Este proceso iterativo, centrado en cada punto de la ruta s_i, ajusta las coordenadas de manera independiente utilizando la siguiente regla de actualización:

$$s_i = s_i + \alpha(p_i - s_i) + \beta(s_{i+1} + s_{i-1} + -2s_i)$$

Este proceso se repite sucesivamente, ajustando cada punto hasta que la suma de los cambios absolutos en ambas coordenadas, x e y, no supera un límite de tolerancia previamente establecido. Dentro de esta fórmula, el término α representa el *data weight*, que influye en la proximidad de la ruta final a la trayectoria inicial; un valor más alto de α tiende a preservar la forma original del camino. Por otro lado, β actúa como el *smoothing weight*, con un mayor valor promoviendo una mayor suavidad en la trayectoria resultante.

La efectividad de esta metodología se puede apreciar visualmente en la Figura 31, donde se exhibe la trayectoria procesada por los bloques de Douglas-Peucker y el bloque ‘smoother'.

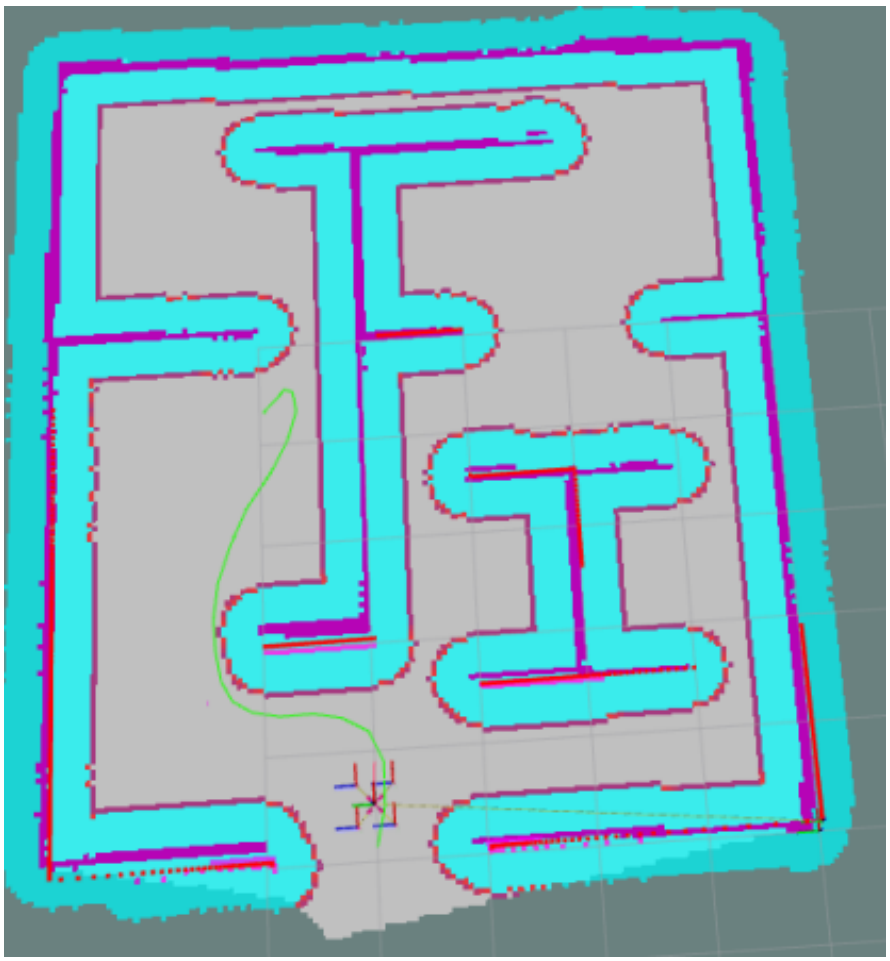

Figura 31: Ruta resultante en la implementación final

*5.1.1 Implementación del RRT**

El planificador basado en el algoritmo RRT* se ha implementado sobre el *stack* de Nav2 mediante un *plugin*. Obsérvese la arquitectura de Nav2 en la Figura 32. La flexibilidad de Nav2 es tal que admite la personalización de su arquitectura mediante la incorporación de *plugins* especializados, lo cual facilita la implementación de planificadores alternativos que pueden ser intercambiados con facilidad en el núcleo del sistema de planificación, concretamente en la sección denominada 'planner server'.

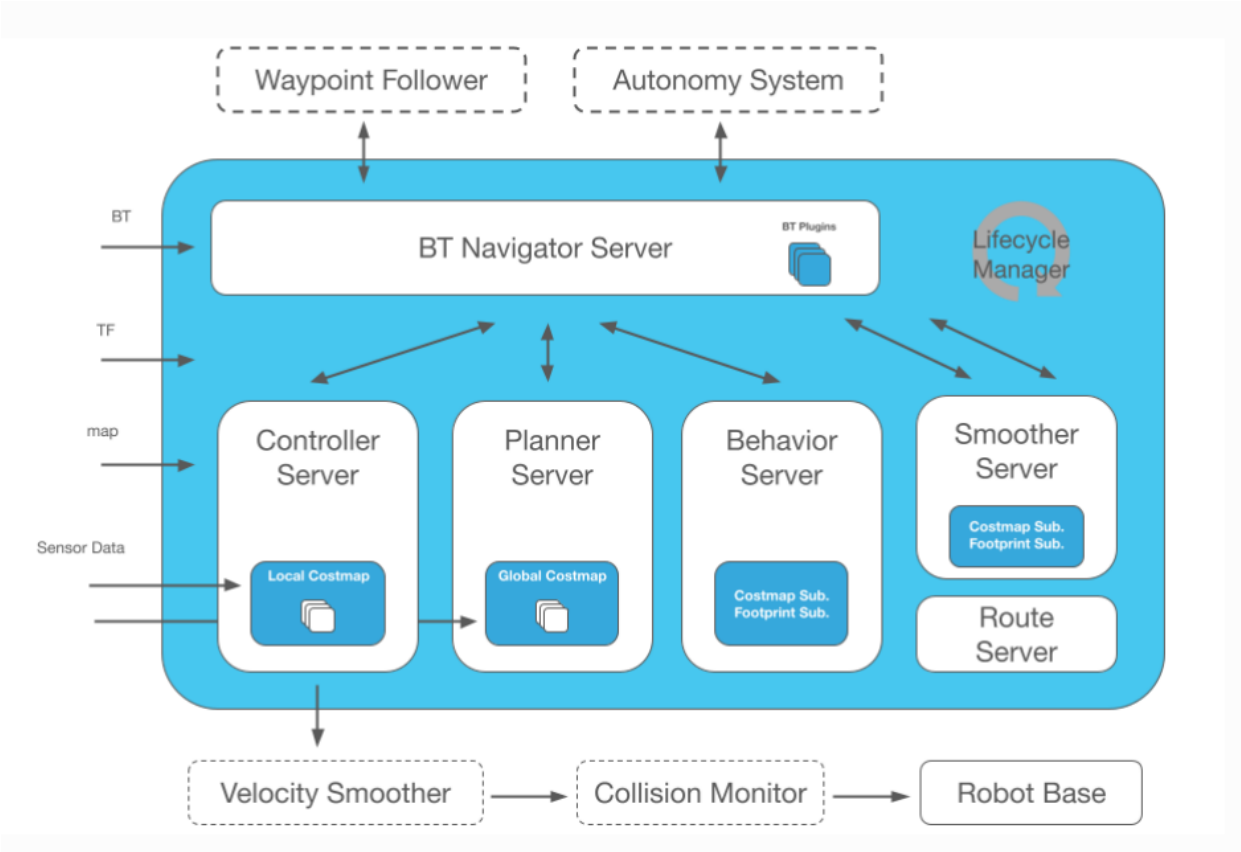

Figura 32: Arquitectura de Nav2

El plugin ha sido programado en C++ creando un hijo de la clase GlobalPlanner presente en el *stack* de Nav2. Para ello se han creado la clase *vertex* (que representa un punto en el espacio) y la clase *tree* (que representa un árbol formado por puntos en el espacio). A estas clases se le han añadido los métodos necesarios para implementar el algoritmo. Se pueden observar extractos del código en el Anexo B.

5.2 NAV2 NAVFNPLANNER

Como se ha mencionado en el comienzo de esta sección, también se ha explorado usar el planificador que usa Nav2 por defecto, el NavFnPlanner. Este planificador es versátil en su funcionamiento, pudiendo recurrir al algoritmo de Dijkstra o al algoritmo A* en función de las preferencias del usuario. La elección ha recaído en el algoritmo A*, ya que este tiende a ser más eficiente en términos del número de iteraciones necesarias. Por lo general, el

algoritmo A* necesita evaluar un volumen menor de vértices, lo cual es debido a la información proporcionada en la función heurística (véase la sección 2.3.1), resultando en una reducción del espacio de búsqueda y, por ende, en un proceso más rápido y directo para hallar el camino más corto [16].

5.2.1 Implementación del NavFnPlanner

La implementación del NavFnPlanner, que es el planificador estándar de Nav2, se caracteriza por su simplicidad operativa. Para su puesta en marcha, únicamente se requiere la especificación adecuada de los parámetros del planificador dentro del archivo de configuración de Nav2. Los detalles de la configuración pueden ser consultados en la Figura 33, la cual proporciona una representación visual clara de los parámetros necesarios para la correcta operación del planificador dentro del ecosistema de Nav2.

```yaml
planner_server:
  ros__parameters:
    expected_planner_frequency: 20.0
    use_sim_time: True
    planner_plugins: ["GridBased"]
    GridBased:
      plugin: "nav2_navfn_planner/NavfnPlanner"
      tolerance: 0.5
      use_astar: true
      allow_unknown: true
```

Figura 33: Parámetros del NavFnPlanner

En la Figura 34 puede observarse una ruta planificada por el NavFnPlanner basado en el A*.

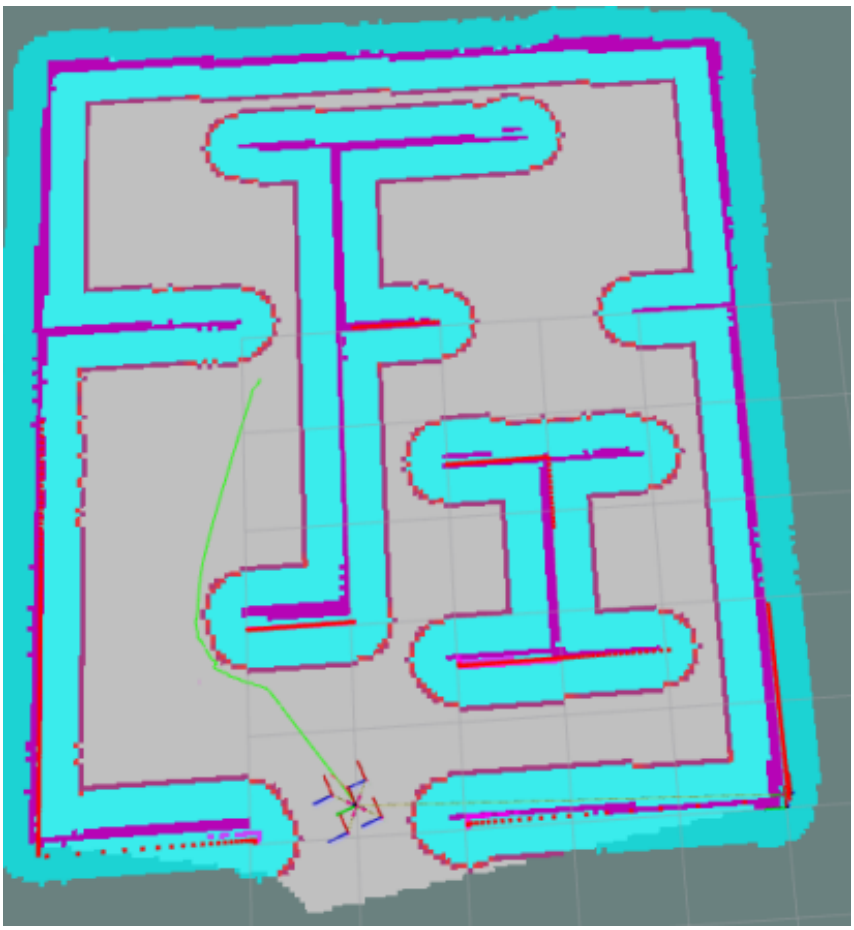

Figura 34: Ruta planificada por el NavFnPlanner

5.3 RESULTADOS PRELIMINARES

Durante las evaluaciones comparativas, el NavFnPlanner ha demostrado una eficiencia superior en la generación de trayectorias, distinguiéndose por su capacidad para identificar caminos válidos con rapidez. Además, el planificador basado en el algoritmo RRT* a veces opta por ejecutar maniobras de giro redundantes, en lugar de elegir una ruta directa hacia el destino, como puede observarse en la Figura 31. Estas desviaciones ocasionales han inclinado la balanza a favor del NavFnPlanner para su adopción en la versión definitiva del sistema.

Para asegurar la robustez y fiabilidad del NavFnPlanner, se han efectuado pruebas en el entorno mapeado original y en un escenario alterado que presenta obstáculos adicionales, ausentes en la cartografía estándar. Estos entornos experimentales pueden ser observados en la Figura 35 y la Figura 36.

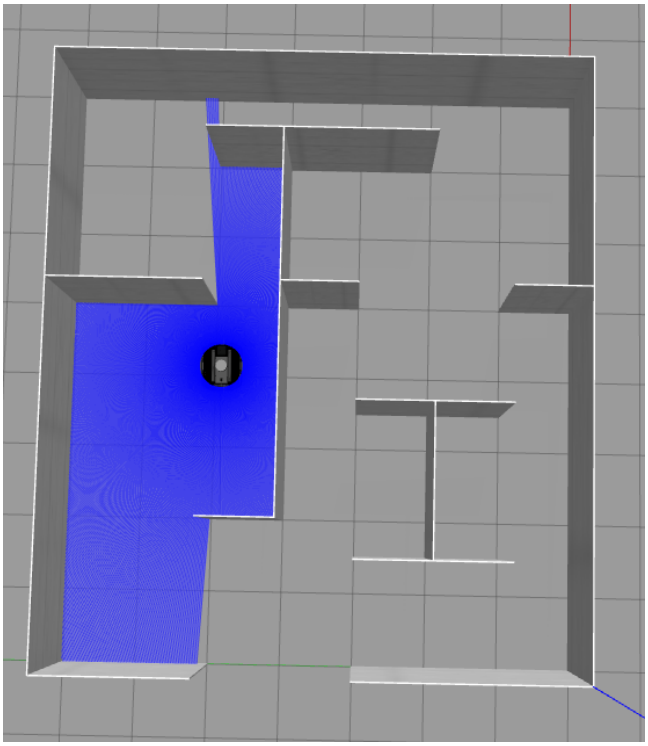

Figura 35: Entorno de simulación sin objetos adicionales

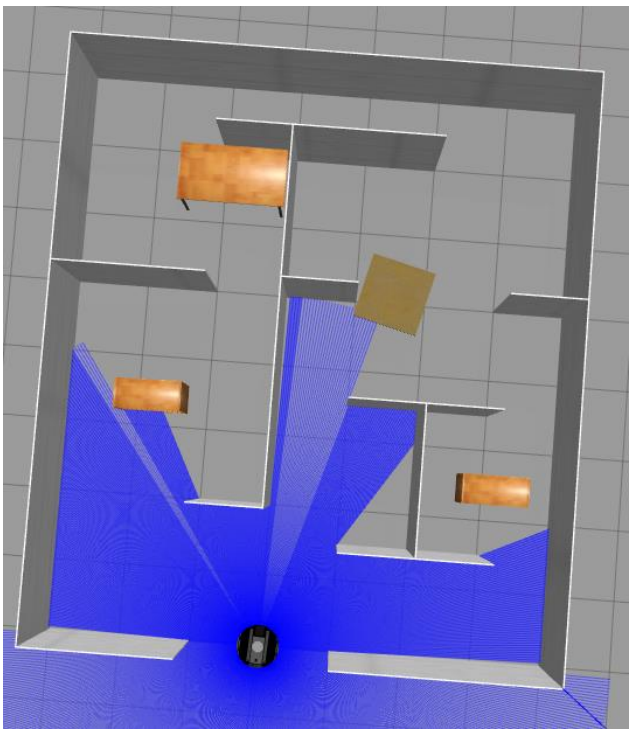

Figura 36: Entorno de simulación con objetos no presentes en el mapa

5.3.1 Simulación en el mapa sin objetos adicionales

Las simulaciones en el mapa sin objetos adicionales han dado buenos resultados. Las rutas planificadas son válidas y de longitud adecuada, como puede observarse en la Figura 34. Además, el planificador no muestra problemas con las zonas más estrechas del mapa, proporcionando siempre una ruta válida, como puede observarse en la Figura 37.

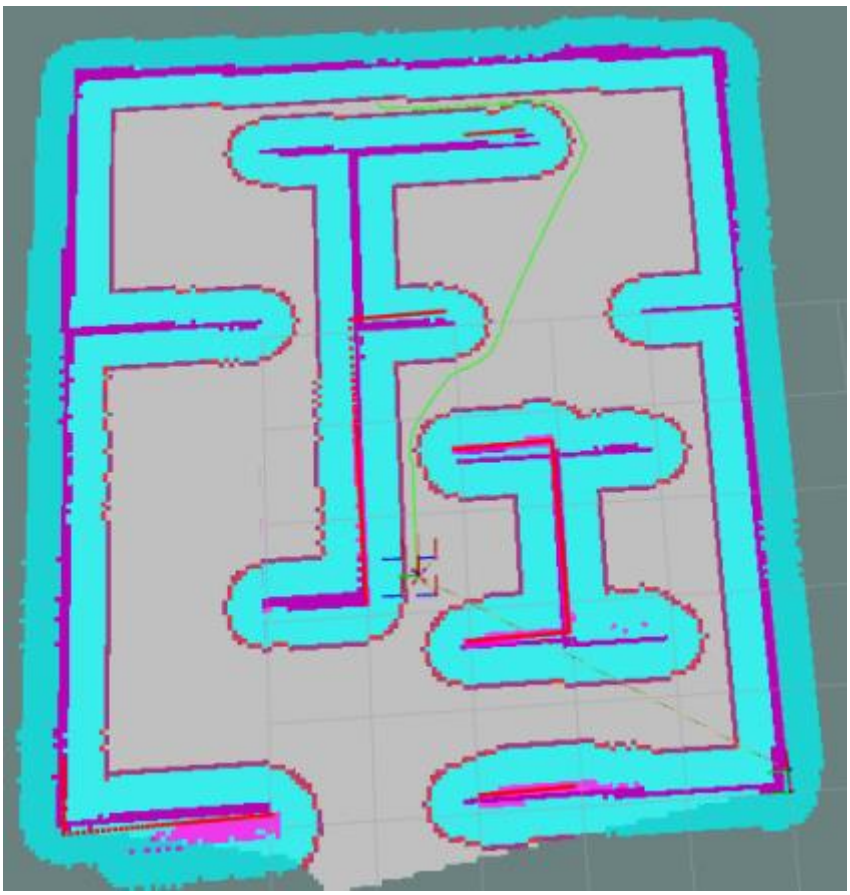

Figura 37: Planificación por lugares estrechos

5.3.2 Simulación en el mapa con objetos adicionales

Las simulaciones con objetos no presentes en el mapa también han resultado exitosas. Como puede observarse en la Figura 38, el planificador proporciona una ruta no válida en un principio, pero una vez se acerca suficiente al objeto es capaz de localizarlo y calcular una nueva ruta compatible con la localización de este (obsérvese la Figura 39).

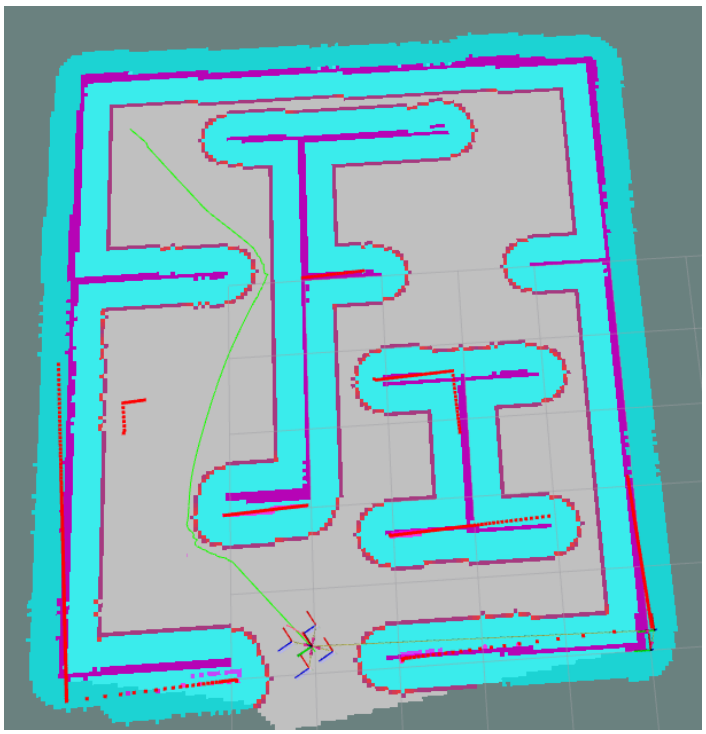

Figura 38: Planificación inicial con objetos no proporcionados en el mapa

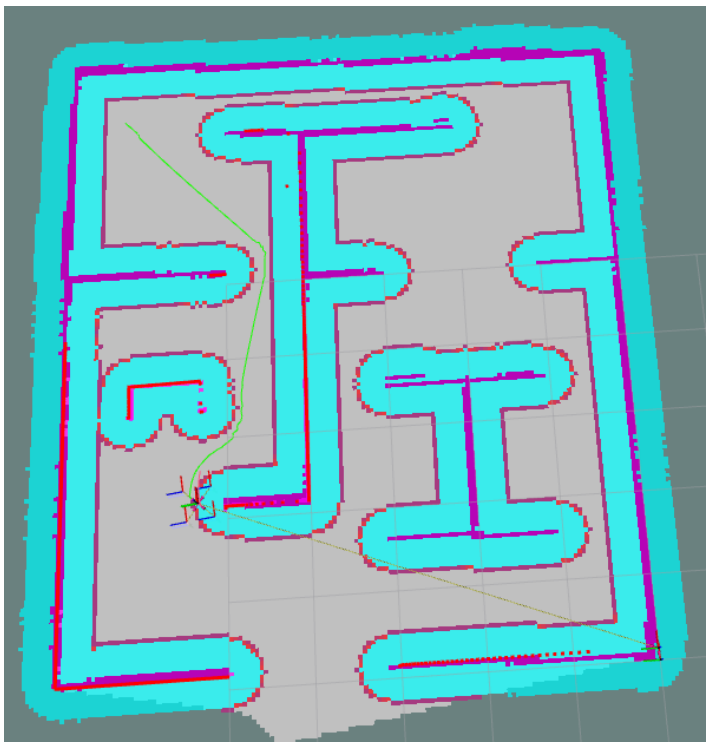

Figura 39: Planificación tras detectar el objeto nuevo en el mapa

Durante las simulaciones, se identificó una incidencia recurrente: el robot, en ocasiones, quedaba inmóvil después de ejecutar una acción de replanificación. Un análisis reveló que la causa subyacente residía en el componente 'local planner' de Nav2. No obstante, se tomó la decisión de no intervenir en este aspecto debido a que, en la configuración final del sistema integrado, tal como se describe en la sección de Arquitectura y se ilustra en la Figura 15, la tarea asignada al 'local planner' de Nav2 es asumida directamente por el controlador desarrollado en Matlab/Simulink. Se anticipó que el controlador personalizado no manifestaría la misma problemática observada en las simulaciones, ya que en el entorno de pruebas, el controlador no estaba integrado y su funcionalidad estaba siendo emulada por el 'local planner' de Nav2.

6. INTEGRACIÓN

6.1 INTEGRACIÓN DEL MÓDULO DE PLANIFICACIÓN DE RUTAS Y EL MÓDULO DE CONTROL

Como se describió en la sección 3, la integración de estos dos módulos se facilita mediante una comunicación TCP/IP. A través de esta interfaz, el módulo de planificación de rutas, operado en la Nvidia Jetson AGX Orin, transmite la pose actual y la pose objetivo del AGV al módulo de control, alojado en la Raspberry Pi. Es fundamental recordar que el módulo de control requiere conocer la pose actual y la pose objetivo para calcular las variables r y α. Estas funciones se realizan mediante los paquetes waypoint_finder y tracer_tcp_ros_bridge, detallados previamente en la sección 3.3.

Adicionalmente, el módulo de localización, también ejecutado en la Nvidia Jetson AGX Orin, depende de los datos de odometría para su operación adecuada. Nav2 utiliza estos datos, junto con la información de la nube de puntos generada por el LiDAR, para estimar la posición del robot. En este contexto, un servidor TCP/IP en la Raspberry Pi recopila y envía la información de velocidad y velocidad angular, obtenida a través del bus CAN, a la Nvidia Jetson AGX Orin. La Figura 40 muestra el subsistema de Simulink diseñado para gestionar estas comunicaciones.

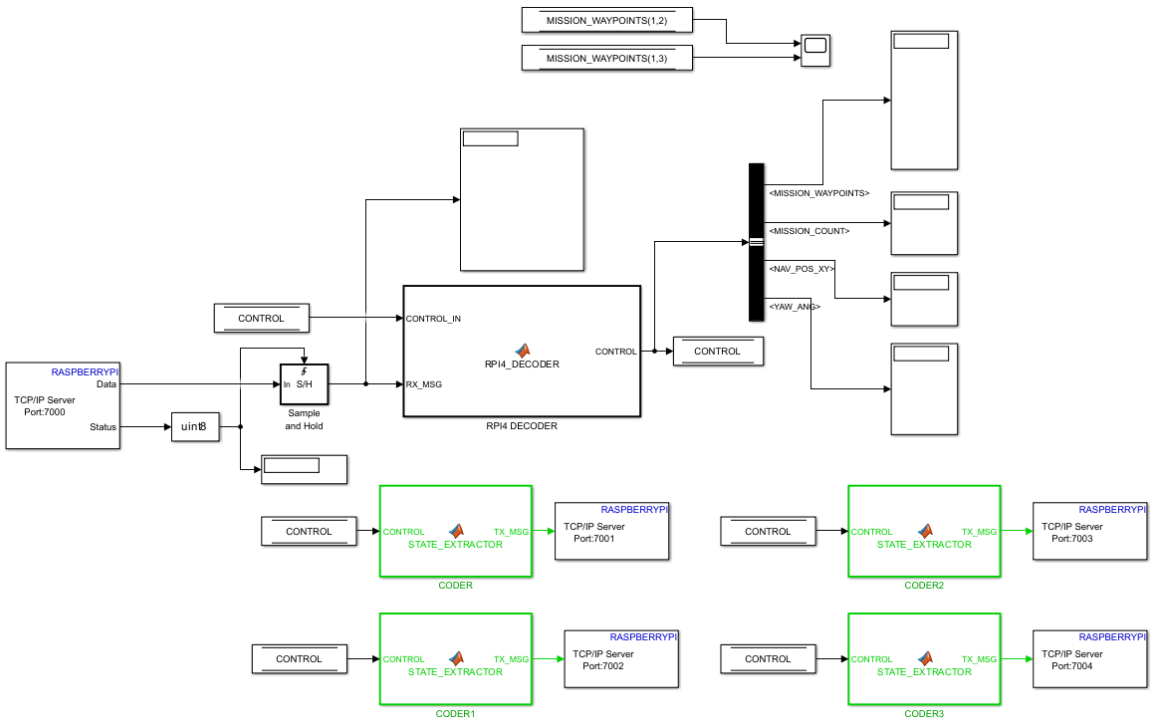

Figura 40: Subsistema de Simulink diseñado para manejar las comunicaciones

6.2 INTEGRACIÓN HARDWARE-SOFTWARE

Para facilitar el manejo por parte del usuario se ha integrado un conmutador, un botón y un joystick en el sistema. El conmutador y el botón permiten al usuario cambiar de modo de funcionamiento para que el robot pueda moverse de forma autónoma o ser controlado de forma manual a través del joystick.

Se ha programado una máquina de estados en Python para gestionar los distintos modos de funcionamiento del sistema. Esta máquina de estados se ejecuta en la Raspberry Pi, y activa y desactiva la navegación autónoma en función del modo de funcionamiento. Puede observarse la transición entre los distintos estados en la Figura 41.

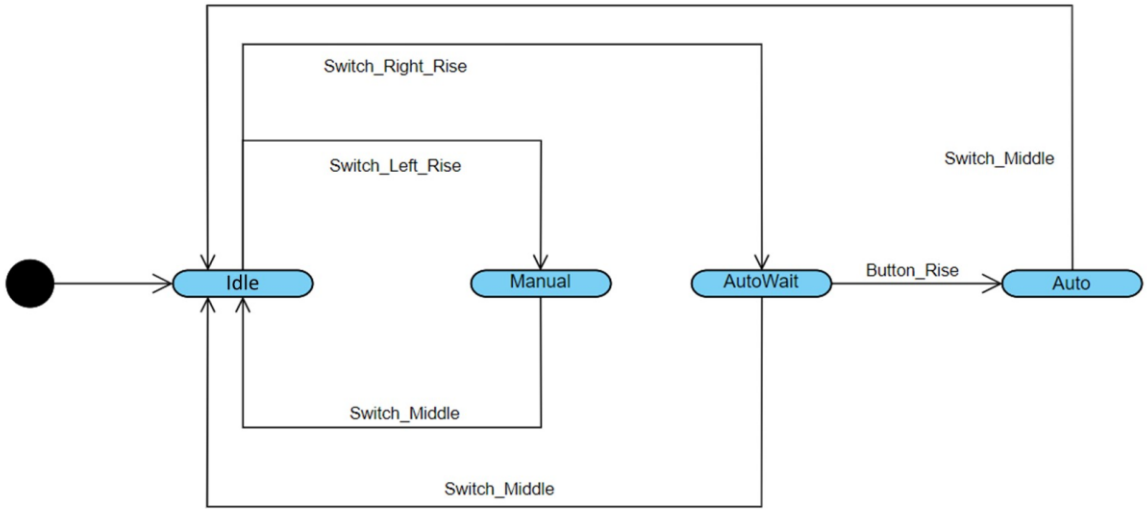

Figura 41: Transiciones de la máquina de estados

Como puede observarse en la Figura 41 la máquina de estados cuenta con 4 estados que cambian en función de los flancos y el estado de las distintas entradas (tres entradas asociadas al conmutador de 3 posiciones y una entrada asociada al botón). En los distintos estados, el comportamiento es el siguiente:

- **Idle**: El sistema permanece en reposo pase lo que pase. No entra en movimiento si se usa el joystick o si se establece un punto objetivo para la navegación autónoma.
- **Manual**: El sistema puede ser controlado a través del joystick.
- **AutoWait**: El sistema permanece en reposo pase lo que pase esperando a que se indique el inicio de la navegación autónoma pulsando el botón. Si se devuelve el conmutador a la posición media, el sistema retorna al estado Idle.
- **Auto**: El sistema entra en navegación autónoma, dirigiéndose al punto objetivo si este se ha proporcionado.

7. RESULTADOS

El prototipo final mostró un buen rendimiento durante el evento final del UNIJES SocialTech Challenge, siendo el único en superar con éxito las dos pruebas realizadas: navegación en el entorno y navegación en el entorno con obstáculos estáticos no presentes en el mapa. Este desempeño le permitió ganar los premios de "Mejor Comportamiento en Pruebas" y "Ganador Absoluto", obteniendo así dos de los tres galardones otorgados en la competición. El diseño cuenta con un planificador capaz de adaptarse a entornos dinámicos y un controlador capaz de seguir la ruta de forma precisa, como se han mostrado en los resultados parciales de las secciones 4 y 5.

8.CONCLUSIONES Y FUTUROS DESARROLLOS

8.1 ELIMINACIÓN DE LA RASPBERRY PI DE LA ARQUITECTURA

Como se indica en la sección 3.2, la Raspberry Pi fue seleccionada en un principio por su facilidad de integración y para poder diseñar controles e implementarlos de forma rápida. Sin embargo, en el diseño final de este prototipo la Raspberry Pi es redundante, ya que el algoritmo de control podría también ser ejecutado en la Nvidia Jetson AGX Orin. Eliminar la Raspberry Pi de la arquitectura del sistema traería una serie de beneficios:

- **Eliminación de comunicaciones TCP/IP**: La eliminación de todas las comunicaciones TCP/IP programadas entre la Raspberry Pi y la Nvidia Jetson llevaría a una arquitectura más limpia y fácil de entender. Además facilitaría las tareas de *debugging*, ya que los posibles fallos en el software estarían siempre localizados en el mismo dispositivo.

- **Implementación de sistemas de control más complejos**: Como se comentó en la sección 4.4.2.2, la optimización e implementación del controlador MPC fue abandonada porque la Raspberry Pi 4B no cuenta con potencia computacional suficiente como para ejecutar este controlador. Eliminando la Raspberry Pi e implementando el controlador en la Nvidia Jetson se podrían probar controladores con alta carga computacional como el MPC. Esto podría llevar a un mejor comportamiento del sistema ya que estos controladores tienen en cuenta dinámicas del sistema que los controladores geométricos como el *pure pursuit* ignoran [15].
- **Integración de toda la arquitectura de software en ROS2**: Actualmente la máquina de estados que gobierna los modos de funcionamiento, el sistema de control de rutas y el servidor que se comunica con el BUS CAN se ejecutan fuera del entorno de ROS2. Eliminando la Raspberry Pi se podrían llevar todos estos elementos del software a ROS2, creando así una arquitectura más limpia. En la Figura 42 puede observarse el grafo propuesto para esta nueva arquitectura.

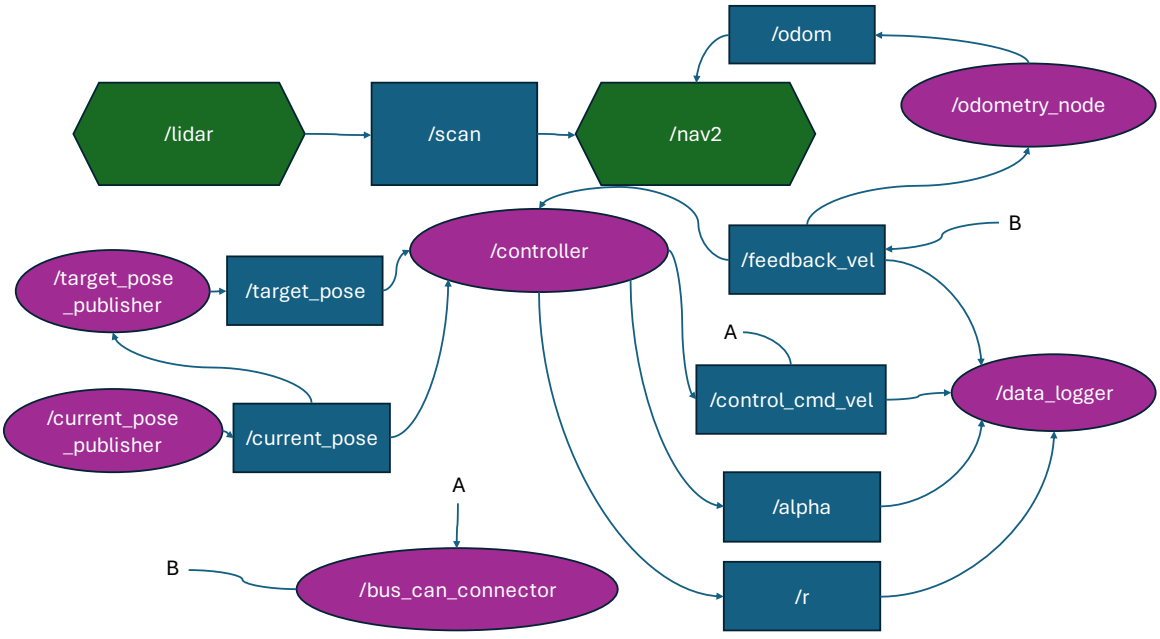

Figura 42: Grafo de ROS2 propuesto para una arquitectura sin la Raspberry Pi

El grafo presentado es bastante similar al de la Figura 17, con la diferencia de que desaparece el nodo tcp_ros_bridge y aparecen los nodos controller y bus_can_connector, cuyas funcionalidades son las siguientes:

- **controller**: Este nodo ejecuta la funcionalidad del controlador presentado en la sección 4. Calcula las variables r y α a partir de los datos de los *topics* /target_pose y /current_pose y, a partir de estas, calcula los mandos de velocidad lineal y angular. Además, este nodo está suscrito al *topic* /feedback_vel porque la realimentación de velocidad lineal y angular es necesaria en un controlador dinámico. Por último, este nodo publica las variables r y α en *topics* para que estas puedan ser guardadas por el nodo data_logger.
- **bus_can_connector**: Este nodo gestiona la conexión con el bus can de manera similar al servidor TCP/IP de la Raspberry Pi presentado en la Figura 16. Extrae los datos de realimentación de velocidad lineal y angular y los publica en el *topic* /feedback_vel. Además, envía los comandos de velocidad lineal y angular publicados por el nodo controller en el *topic* /control_cmd_vel.

8.2 PLANIFICACIÓN EN BASE A LOS CAMBIOS EN EL MAPA DE COSTES

Aunque el funcionamiento correcto del sistema ha sido ampliamente testeado y se ha llegado a la conclusión de que el módulo de navegación diseñado es funcional, este cuenta con un principal problema: Es demasiado oscilante. Actualmente el planificador está diseñado para generar rutas válidas con una frecuencia de 1 Hz y cada vez que el robot ha recorrido 0.2 m. Aunque estas planificaciones continuas hacen que el vehículo sea más resiliente frente a la presencia de objetos estáticos no presentes en el mapa y objetos dinámicos, también generan más oscilaciones, ya que el controlador debe adaptarse continuamente a la nueva ruta planificada. Esto ha sido comprobado en pruebas experimentales en las que no se ha planificado con una frecuencia alta, resultando estas en movimientos del robot mucho más suaves y, por lo tanto, más cómodos para el usuario.

La solución propuesta para este problema es crear un *plugin* en Nav2 que permita planificar cada vez que se encuentra un cambio significativo en el mapa de costes. La ejecución del algoritmo, descrita en un alto nivel, estaría compuesta por los siguientes pasos:

1. **Planificación inicial**: El módulo de navegación planifica siempre que se publica una nueva pose objetivo.
2. **Chequeo del mapa de costes**: De forma continua, el algoritmo suma los costes asociados a todos los puntos del mapa de costes, y calcula la diferencia de estos con los calculados la última vez que se realizó una planificación. Si esta diferencia es mayor a cierto valor, se vuelve a planificar una ruta.

De esta manera solamente se planificaría cuando aparece un objeto no esperado, manteniendo así movimientos suaves la mayoría del tiempo sin existir un compromiso con la resiliencia frente a objetos no presentes en el mapa.

8.2.1 Implementación del algoritmo de planificación basado en los cambios en el mapa de costes

Como se ha mencionado en la sección anterior, este algoritmo se podría implementar mediante un *plugin* en Nav2. Este *plugin* sería un decorador para los *behavior trees* de Nav2 que mandase *ticks* cuando existe el cambio en el mapa de costes detallado en la sección anterior.

Un *behavior tree* es un modelo de comportamiento utilizado principalmente en inteligencia artificial para controlar el comportamiento de agentes autónomos, como personajes en videojuegos o robots [31]. Es una estructura jerárquica que organiza las tareas de un agente en forma de árbol, donde cada nodo representa una tarea específica o una decisión. Los *behavior trees* son apreciados por su flexibilidad y modularidad, lo que permite diseñar comportamientos complejos de manera más sencilla y mantenible en comparación con otros enfoques, como las máquinas de estados finitos.

Los *behavior trees* funcionan mediante un proceso cíclico conocido como *ticks*. Un *tick* es una señal que se envía desde la raíz del árbol a sus nodos hijos para evaluar y ejecutar sus tareas. Cada nodo en el árbol recibe *ticks* periódicamente, y su comportamiento depende de su tipo y de su estado actual. Los nodos pueden devolver tres tipos de estados en respuesta a un *tick*:

1. **Éxito (*Success*)**: La tarea asociada al nodo se ha completado correctamente.
2. **Fallo (*Failure*)**: La tarea asociada al nodo ha fallado.
3. **En ejecución (*Running*)**: La tarea asociada al nodo todavía está en progreso.

Los *behavior trees* cuentan con los siguientes elementos básicos o nodos (nótese que este concepto es distinto al de nodo de ROS2, un nodo en un *behavior tree* es un componente lógico dentro del árbol unido a otros nodos):

1. **Nodos de Control**:
    - **Selector (Selector)**: Ejecuta a sus hijos de izquierda a derecha y retorna éxito tan pronto como uno de sus hijos lo haga. Si todos fallan, el selector retorna fracaso.
    - **Secuencia (Sequence)**: Ejecuta a sus hijos de izquierda a derecha y retorna fracaso tan pronto como uno de sus hijos lo haga. Si todos tienen éxito, la secuencia retorna éxito.
    - **Paralelo (Parallel)**: Ejecuta todos sus hijos simultáneamente. Puede estar configurado para esperar que un número mínimo de hijos tengan éxito antes de retornar éxito o fracaso. Los nodos paralelos son útiles para tareas que pueden ejecutarse al mismo tiempo.

2. **Nodos de Acción**: Son las tareas específicas que el agente debe realizar, como moverse a una ubicación, recoger un objeto, o interactuar con el entorno. Estos nodos ejecutan acciones concretas y retornan éxito, fracaso o en ejecución.

3. **Nodos de Condición**: Verifican una condición específica en el entorno o el estado interno del agente. Por ejemplo, verificar si hay un enemigo a la vista o si el agente tiene suficiente energía. Estos nodos no realizan acciones, solo verifican condiciones y retornan éxito o fracaso.

4. **Decoradores**: Son nodos que modifican el comportamiento de sus nodos hijos. Pueden controlar cuándo y cuántas veces se ejecuta un nodo, alterar sus resultados, o agregar condiciones adicionales. Ejemplos de decoradores incluyen repetidores (que repiten la ejecución de un nodo hasta que se cumpla una condición), inversores (que cambian éxito por fracaso y viceversa), y limitadores de tiempo (que imponen un tiempo máximo de ejecución a un nodo).

En el contexto de Nav2, los *behavior trees* son utilizados para controlar el comportamiento y la ejecución de las acciones de los robots móviles. En la Figura 43 puede observarse el *behavior tree* utilizado en el módulo de navegación diseñado.

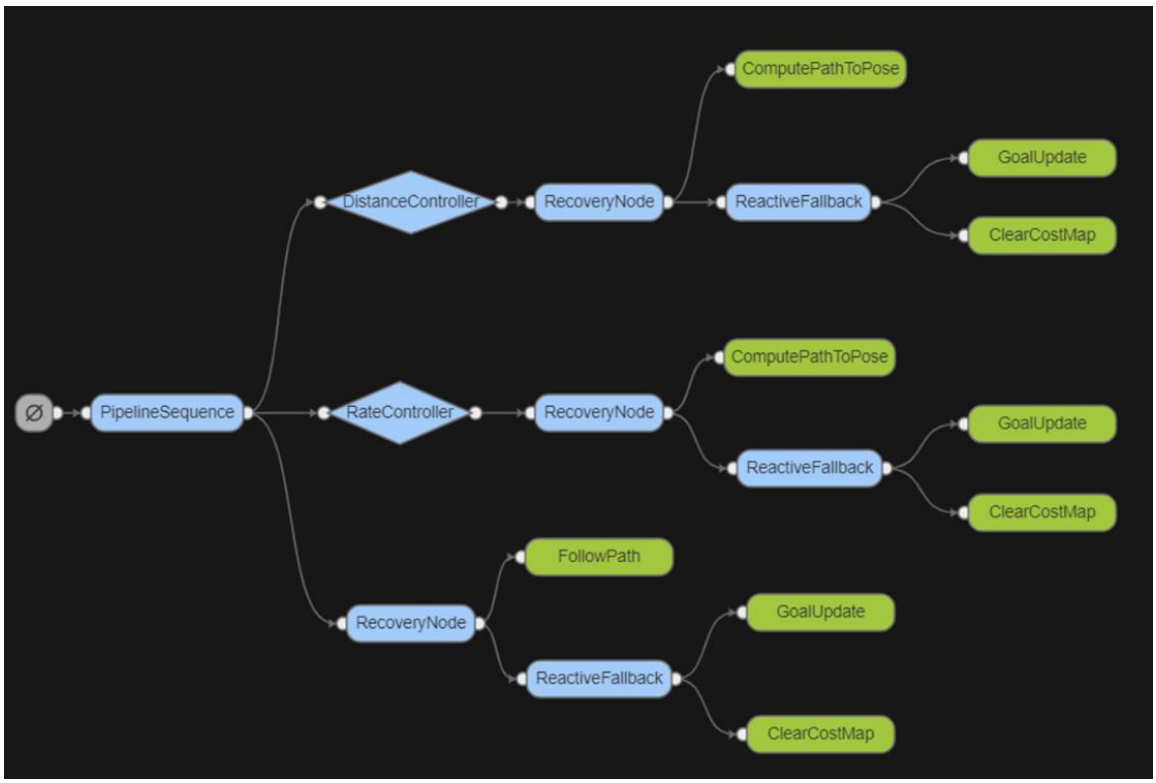

Figura 43: Behavior Tree diseñado

Para implementar el comportamiento deseado, se programaría un *plugin* que implementase un decorador que ejecutase a su hijo cada vez que el mapa de costes cambiase de forma significativa. Este decorador sustituiría al decorador DistanceController en la Figura 43, y después se disminuiría la frecuencia del RateController de forma significativa. Por último, se editaría el *behavior tree* para también replanificar cada vez que se especifica una nueva pose objetivo.

9. BIBLIOGRAFÍA

[1] D. Douglas y T. Peucker, «Algorithms for the reduction of the number of points required to represent a digitized line or its caricature,» *Cartographica: The International Journal for Geographic Information and Geovisualization* , vol. X, nº 2, pp. 112-122, 1973.

[2] R. Siegwart, I. R. Nourbakhsh y D. Scaramuzza, Introduction to Autonomous Mobile Robots, Cambridge: The MIT Press, 2011.

[3] Clearpath Robotics, [En línea]. Available: https://clearpathrobotics.com/ridgeback-indoor-robot-platform/. [Último acceso: 20 9 2023].

[4] Clearpath Robotics, [En línea]. Available: https://clearpathrobotics.com/boxer/. [Último acceso: 20 9 2022].

[5] Agilex Robotics, [En línea]. Available: https://global.agilex.ai/products/tracer. [Último acceso: 19 9 2023].

[6] AgileX Robotics, [En línea]. Available: https://global.agilex.ai/chassis/1. [Último acceso: 14 11 2023].

[7] E. Cocconi, «Enhanced Pure Pursuit Algorithm & Autonomous Driving,» UNSW Sidney, Sidney, 2019.

[8] R. Bhadani, «Path Planning of Unmanned System using Carrot-chasing,» The University of Arizona, Tucson, 2020.

[9] Y. Cao, J. Zhao, H. Zhu, Z. Yang y J. Fan, «PP-ST: An Indoor Mobile Robot Path Tracking Algorithm,» *IEEE Access,* vol. 11, pp. 116356-116365, 2023.

[10] Y. Ding, «shuffleai,» 16 Noviembre 2021. [En línea]. Available: https://www.shuffleai.blog/blog/Three_Methods_of_Vehicle_Lateral_Control.html. [Último acceso: 25 Septiembre 2023].

[11] D. Yan, «Simple Understanding of Kinematic Bicycle Model,» [En línea]. Available: https://www.shuffleai.blog/blog/Simple_Understanding_of_Kinematic_Bicycle_Model.html. [Último acceso: 16 November 2023].

[12] J. Boal Martín-Larrauri, «Tema 7 Seguimiento de rutas,» Madrid, 2024.

[13] G. Hoffmann, C. Tomlin, M. Montemerlo y S. Thrun, «Autonomous Automobile Trajectory Tracking for Off-Road Driving,» Stanford University, Stanford, 2005.

[14] A. AbdElmoniem, A. Osama, M. Abdelaziz y S. A. Maged, «A path-tracking algorithm using,» *International Journal of Advanced Robotic Systems,* pp. 1-11, 2020.

[15] Z. Wang, K. Sun, S. Ma, L. Sun, W. Gao y Z. Dong, «Improved Linear Quadratic Regulator Lateral Path Tracking Approach Based on a Real-Time Updated Algorithm with Fuzzy Control and Cosine Similarity for Autonomous Vehicles,» 2022. [En línea]. Available: https://www.mdpi.com/2079-9292/11/22/3703.

[16] K. J. Aström y R. M. Murray, Feedback Systems: An Introduction for Scientists and Engineers, Princeton: Princeton University Press, 2008.

[17] J. Xu, J. Lai, R. Guo, X. Lu y L. Xu, «Efficiency-Oriented MPC Algorithm for Path Tracking in Autonomous Agricultural Machinery,» *MPDI,* vol. 1662, p. 12, 2022.

[18] E. F. Camacho y C. Bordons Alba, «Control predictivo: pasado, presente y futuro,» *Revista Iberoamericana de Automática e Informática Industrial,* vol. 1 (3), pp. 5-28, 2004.

[19] «https://matlab.mathworks.com/,» MathWorks, [En línea]. Available: https://es.mathworks.com/help/robotics/ug/probabilistic-roadmaps-prm.html. [Último acceso: 12 10 2023].

[20] S. M. LaValle, Planning Algorithms, Cambridge: Cambridge University Press, 2006.

[21] N. Iram, A. Khan y Z. Habib, «A Comparison of RRT, RRT* and RRT*-Smart Path Planning Algorithms,» COMSATS Institute of Information Technology, Lahore, 2016.

[22] Ş. Ioan A., M. Mark y K. Lydia E., «The Open Motion Planning Library,» *IEEE Robotics & Automation Magazine,* vol. 19, nº 4, pp. 72-82, 2012.

[23] Nvidia, «Jetson AGX Orin for Next-Gen Robotics,» [En línea]. Available: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-orin/. [Último acceso: 10 June 2024].

[24] Nvidia, «Jetson Developer Kits | Nvidia Developer,» [En línea]. Available: https://developer.nvidia.com/embedded/jetson-developer-kits. [Último acceso: 10 June 2024].

[25] Raspberry Pi, «Raspberry Pi Model 4B specifications,» [En línea]. Available: https://www.raspberrypi.com/products/raspberry-pi-4-model-b/specifications/. [Último acceso: 10 June 2024].

[26] Raspberry Pi, «raspberry pi 4B datasheet,» [En línea]. Available: https://datasheets.raspberrypi.com/rpi4/raspberry-pi-4-datasheet.pdf#:~:text=URL%3A%20https%3A%2F%2Fdatasheets.raspberrypi.com%2Frpi4%2Fraspberry. [Último acceso: 10 June 2024].

[27] ROS2 Community, «ROS2 Documentation,» [En línea]. Available: https://docs.ros.org/en/humble/index.html. [Último acceso: 10 June 2024].

[28] J. Nocedal y S. J. Wright, Numerical Optimization, Springer, 2006.

[29] M. Melanie, An Introduction to Genetic Algorithms, MIT Press, 1998.

[30] «Wikipedia,» 2018 June 19. [En línea]. Available: https://en.wikipedia.org/wiki/Ramer%E2%80%93Douglas%E2%80%93Peucker_algorithm#:~:text=The%20Ramer%E2%80%93Douglas%E2%80%93Peucker%20algorithm,algorithms%20developed%20for%20cartographic%20generalization.. [Último acceso: 2024 March 2024].

[31] M. Colledanchise y P. Ögren, Behavior Trees in Robotics and AI, Cornell University, 2022.

[32] NVIDIA, «NVIDIA Jetson AGX Orin Developer Kit User Guide,» [En línea]. Available: https://developer.nvidia.com/embedded/learn/jetson-agx-orin-devkit-user-guide/index.html. [Último acceso: 23 November 2023].

[33] J. B. Martín-Larrauri, «Tema 7 Seguimiento de rutas,» Madrid, 2024.

ANEXO A: ALINEACIÓN CON LOS OBJETIVOS DE DESARROLLO SOSTENIBLE (ODS)

Varios de los Objetivos de Desarrollo Sostenible (ODS) de las Naciones Unidas pueden ser respaldados mediante el desarrollo de una silla de ruedas autónoma. Algunas posibles alineaciones son las siguientes:

- ODS 3: Salud y Bienestar. Las sillas de ruedas autónomas pueden aumentar la movilidad e independencia de las personas con discapacidades, mejorando su bienestar general y calidad de vida.

- ODS 9: Infraestructura, Industria e Innovación. La creación de sillas de ruedas autónomas requiere innovación en robótica y tecnología de asistencia, lo que impulsa tanto la tecnología como la infraestructura.

- ODS 10: Reducción de las Desigualdades. La brecha de movilidad entre personas con y sin discapacidades puede cerrarse mediante sillas de ruedas autónomas, fomentando la inclusión social y reduciendo las desigualdades.

ANEXO B: EXTRACTOS DEL CÓDIGO DESARROLLADO

IMPLEMENTACIÓN DEL ALGORITMO DE DOUGLAS-PEUCKER EN C++.

```
std::vector<std::shared_ptr<Vertex>> douglasPeucker(const
std::vector<std::shared_ptr<Vertex>>& path, double epsilon)
  {
    std::vector<std::shared_ptr<Vertex>> simplified_path;
    double dmax = 0;
    int index = 0;
    int end = path.size() - 1;
    for (int i = 1; i < end; ++i)
    {
      double d = perpDistance(path[i], path[0], path[end]);
      if (d > dmax)
      {
        index = i;
        dmax = d;
      }
    }
    if (dmax > epsilon)
    {
      std::vector<std::shared_ptr<Vertex>> rec_results1 =
douglasPeucker(std::vector<std::shared_ptr<Vertex>>(path.begin(),
path.begin() + index), epsilon);
      std::vector<std::shared_ptr<Vertex>> rec_results2 =
douglasPeucker(std::vector<std::shared_ptr<Vertex>>(path.begin() +
index, path.end()), epsilon);
      simplified_path.insert(simplified_path.end(),
rec_results1.begin(), rec_results1.end() - 1);
      simplified_path.insert(simplified_path.end(),
rec_results2.begin(), rec_results2.end());
    }
    else
    {
      simplified_path.push_back(path[0]);
      simplified_path.push_back(path[end]);
    }
    return simplified_path;
  }
```

DECLARACIÓN CLASE VERTEX

```
class Vertex
```

```
{
  private:
    // x and y coordinates of the vertex
    double x_;
    double y_;
    // distance to the parent vertex
    double distance_to_parent_;
    // smart pointer to the parent vertex
    std::shared_ptr<Vertex> parent_;
    // vector of smart pointers to the children vertices
    std::vector<std::shared_ptr<Vertex>> children_;

  public:
    // Constructors
    Vertex();
    Vertex(double x, double y);
    void setX(double x);
    void setY(double y);
    void setDistanceToParent(double distance);
    void setParent(std::shared_ptr<Vertex> parent);
    void addChild(std::shared_ptr<Vertex> child);
    double getX() const;
    double getY() const;
    double getDistanceToParent() const;
    std::shared_ptr<Vertex> getParent() const;
};
```

DECLARACIÓN DE LA CLASE TREE

```
class Tree
  {
    private:
      // Vector of vertices that make up the tree
      std::vector<std::shared_ptr<Vertex>> vertices_;
      // Max distance between vertices
      double max_distance_;
      // Max and min x and y values
      double max_x_;
      double max_y_;
      double min_x_;
      double min_y_;
      // Interpolation distance for the costmap
      double interpolation_dist_;
      // Tolerance for checking if a vertex is close to the goal
      double tolerance_;
      // Radius for searching for nearby vertices (RRT* algorithm)
      double search_radius_;
      // Costmap
      nav2_costmap_2d::Costmap2D* costmap_;
      // Random number generator
      std::random_device rd;
      std::mt19937 gen;
```

```
        std::uniform_real_distribution<> dis_x;
        std::uniform_real_distribution<> dis_y;

    public:
      Tree(double max_distance, double max_x, double max_y, double
min_x,
           double min_y, double interpolation_dist, double tolerance,
           double search_radius,nav2_costmap_2d::Costmap2D* costmap);
      void addVertex(std::shared_ptr<Vertex> vertex);
      // Add a random vertex to the tree according to RRT* algorithm
      void addRandomVertex();
      // gets nearest Vertex in the tree and sets it as parent of the
input vertex
      std::shared_ptr<Vertex>
getNearestVertex(std::shared_ptr<Vertex>& vertex);
      double getMaxDistance() const;
      double getMaxX() const;
      double getMaxY() const;
      double getMinX() const;
      double getMinY() const;
      double getInterpolationDist() const;
      double getTolerance() const;
      // returns random x according to max and min x
      double getRandomX();
      // returns random y according to max and min y
      double getRandomY();
      // Check if the line between two vertices is obstacle free
      bool isObsFree(const std::shared_ptr<Vertex>& vertex1, const
std::shared_ptr<Vertex>& vertex2) const;
      std::vector<std::shared_ptr<Vertex>> getVertices() const;
      // Sets the parent of the input vertex to the vertex with
lowest cost (the one that makes it closer to the origin)
      std::shared_ptr<Vertex>
setLowestCostParent(std::shared_ptr<Vertex>& vertex);
      double getDistanceToOrigin(std::shared_ptr<Vertex>& vertex)
const;
      void reconnectNearVertices(std::shared_ptr<Vertex>& vertex);
  };
```

ANEXO C: MODELO MATEMÁTICO DE NAVEGACIÓN

La implementación del controlador MPC y del LQR se fundamentan en un modelo unificado de espacio de estados que describe el comportamiento y la interacción del sistema con su entorno, como se menciona en la sección 4. Dentro de este modelo, se define α como el ángulo que indica la desviación respecto a la orientación deseada hacia el objetivo, y r como la distancia que separa al robot de dicho punto de destino. En la Figura 44 puede observarse una representación gráfica de este modelo.

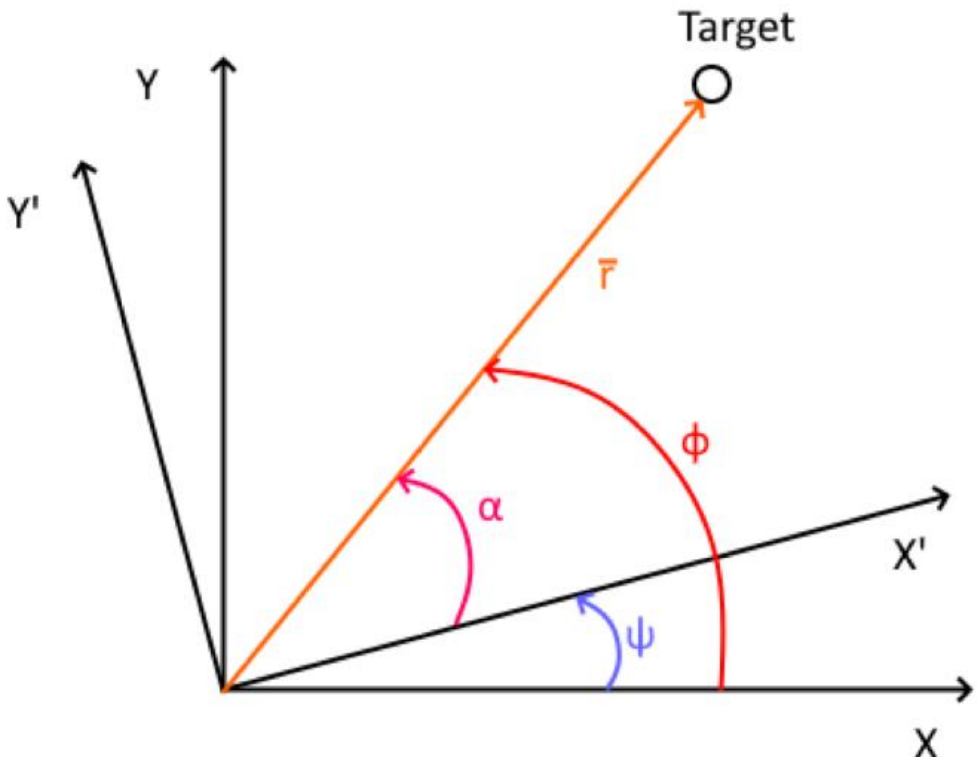

Figura 44: Representación gráfica del modelo de navegación

Teniendo en cuenta que en la figura 36 el origen de coordenadas representa la posición actual del robot, si se toma el punto "Target" como referencia y se define $\overline{v}$ como el vector velocidad del robot:

$$-\overline{v} = -ve^{j\psi} = \frac{d\overline{r}}{dt} = \frac{d}{dt}(re^{j\varphi}) \quad 1$$

Usando, por supuesto, notación de fasores. Si se continúa desarrollando esta expresión se llega al siguiente resultado:

$$\frac{d}{dt}\left(re^{j\varphi}\right) = \frac{dr}{dt}e^{j\varphi} + rj\frac{d\varphi}{dt}e^{j\varphi} = \left(\frac{dr}{dt} + jr\frac{d\varphi}{dt}\right)e^{j\varphi}$$

Expresando función exponencial del último término en coordenadas polares se puede llegar a la siguiente expresión:

$$\left(\frac{dr}{dt} + jr\frac{d\varphi}{dt}\right)e^{j\varphi} = \frac{dr}{dt}\cos(\varphi) + j\frac{dr}{dt}\sin(\varphi) + jr\frac{d\varphi}{dt}\cos(\varphi) - r\frac{d\varphi}{dt}\sin(\varphi) =$$
$$= \frac{dr}{dt}\cos(\varphi) - \frac{d\varphi}{dt}\sin(\varphi) + j(\frac{dr}{dt}\sin(\varphi) + r\frac{d\varphi}{dt}\cos(\varphi))$$

De la expresión $-\overline{v} = -ve^{j\psi} = \frac{d\overline{r}}{dt} = \frac{d}{dt}(re^{j\varphi})$ 1 se puede igualar la expresión anterior a $-\overline{v}$.

$$-\overline{v} = \frac{dr}{dt}\cos(\varphi) - \frac{d\varphi}{dt}\sin(\varphi) + j(\frac{dr}{dt}\sin(\varphi) + r\frac{d\varphi}{dt}\cos(\varphi))$$

$$-v\left(\cos(\psi) + jsin(\psi)\right) = \frac{dr}{dt}\cos(\varphi) - \frac{d\varphi}{dt}\sin(\varphi) + j(\frac{dr}{dt}\sin(\varphi) + r\frac{d\varphi}{dt}\cos(\varphi))$$

Igualando las partes reales e imaginaria resultan las siguientes ecuaciones:

$$-vcos(\psi) = \frac{dr}{dt}\cos(\varphi) - \frac{d\varphi}{dt}\sin(\varphi)$$

$$-vsin(\psi) = \frac{dr}{dt}\sin(\varphi) + r\frac{d\varphi}{dt}\cos(\varphi)$$

Resolviendo el sistema de ecuaciones tomando como variables $\frac{dr}{dt}$ y $\frac{d\varphi}{dt}$ resultan las siguientes expresiones:

$$\frac{dr}{dt} = -v(\cos(\varphi)\cos(\psi) + \sin(\varphi)\sin(\psi)) = -vcos(\varphi - \psi) = -vcos(\alpha)$$

$$\frac{d\varphi}{dt} = \frac{v}{r}\sin(\psi - \varphi) = \frac{v}{r}\sin(\alpha)$$

Y, por lo tanto, puede expresarse la derivada de α mediante la siguiente fórmula:

$$\frac{d\alpha}{dt} = \frac{d\varphi}{dt} - \frac{d\psi}{dt} = \frac{v}{r}\sin(\alpha) - \omega$$

Finalmente, se llega a las expresiones que relacionan las derivadas de las variables de estado con la velocidad y velocidad angular, resultando un modelo de variable de estado no lineal. Debe tenerse en cuenta que los modelos de espacio de estado de v y ω ya han sido obtenidos mediante los ensayos de identificación descritos en la sección 4.

$$\frac{dr}{dt} = -vcos(\alpha)$$

$$\frac{d\alpha}{dt} = \frac{v}{r}\sin(\alpha) - \omega$$

De esta manera, resulta un modelo de espacio de estados con variables de estado X, entradas U y salidas Y definidas de la siguiente forma:

$$X = \begin{bmatrix} r \\ \alpha \\ x_{v1} \\ x_{v2} \\ x_{w1} \\ x_{w2} \end{bmatrix} \qquad U = \begin{bmatrix} v_{ref} \\ \omega_{ref} \end{bmatrix} \qquad Y = \begin{bmatrix} r \\ \alpha \end{bmatrix}$$

Donde x_{v1}, x_{v1}, x_{w1} y x_{w2} son las variables de estado de los modelos de espacio de estado de velocidad y velocidad angular identificados en la sección 4. Es importante destacar que también ha sido necesario calcular una serie de jacobianos para la correcta ejecución del controlador MPC. Sin estos cálculos, el controlador hace una estimación numérica en tiempo real que lleva a una ejecución significativamente más lenta. Los jacobianos necesarios son el jacobiano de la función de estado con respecto a las variables de estado (denotado como A en Matlab), el jacobiano de la función de estado con respecto a los mandos (denotado como B en Matlab) y el jacobiano de la función de salida con respecto a las variables de estado (denotado como C en Matlab). Las expresiones simbólicas de estos jacobianos fueron calculadas y programadas en Matlab.

ANEXO D: MANUAL DE USUARIO

1. PUESTA EN MARCHA DE LA NAVEGACIÓN AUTÓNOMA

Para iniciar la navegación autónoma, es necesario ejecutar los siguientes pasos en el orden detallado. Se detallan los comandos necesarios juntos a cada paso.

1. Abra una terminal en la Nvidia Jetson y acceda al *workspace* del proyecto (ros2_tracer_ws).
   ```
   cd ros2_tracer_ws
   ```
2. Acceda a la carpeta install y haga source del *workspace*.
   ```
   cd install
   source setup.sh
   ```
3. Vuelva al directorio ros2_tracer_workspace y configure la ip del LiDAR. El LiDAR debe estar conectado y encendido. Debe haberse esperado a que este haya comenzado a funcionar (se puede saber cuándo está funcionando porque el ruido que hace aumenta considerablemente cuando lo está haciendo).
   ```
   sudo ifconfig eth0 192.168.1.50
   ```
4. Arranque el sistema mediante el archivo de lanzamiento programado en el paquete tracer_bringup. Tras unos instantes, debería de iniciarse Rviz. Debería de ver una imagen parecida a la mostrada en la Figura 45.
   ```
   ros2 launch tracer_bringup tracer_real.launch.xml
   ```

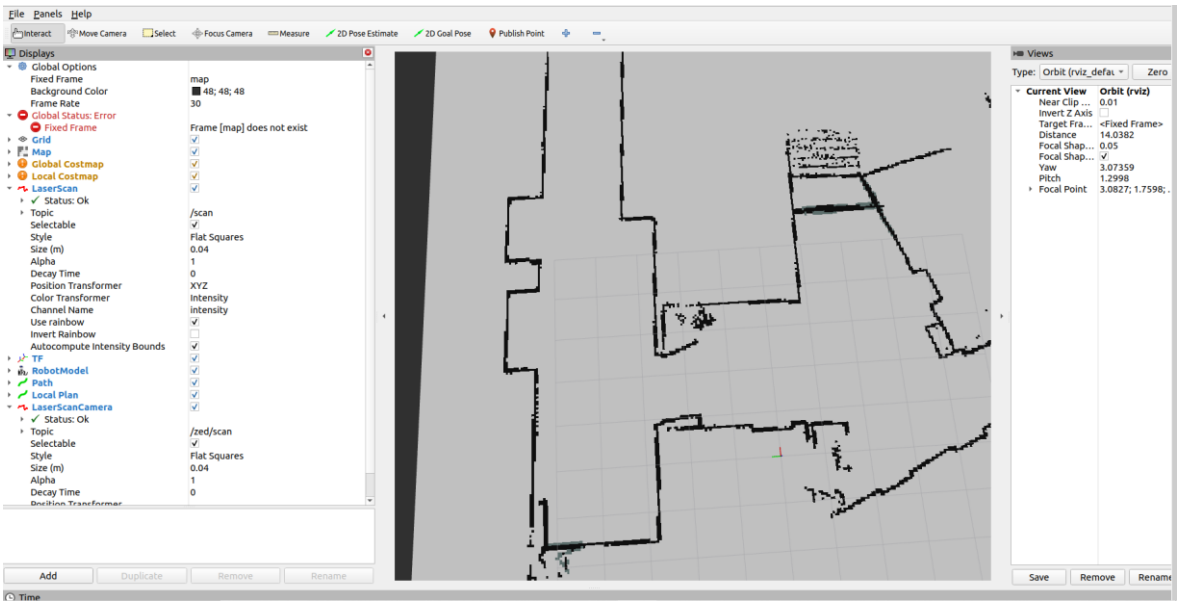

Figura 45: Inicialización de Rviz

5. Para iniciar Nav2 correctamente debe dar una estimación de la pose actual del robot. Para ello seleccione `2D Pose Estimate` en la parte superior junto a `2D Goal Pose` y seleccione la pose en el mapa haciendo click y mostrando la dirección actual del robot. Obsérvese la Figura 46 y la Figura 47. Tras realizar esta última acción, el robot debería de aparecer en el mapa. Debe de tenerse en cuenta que la orientación del robot debe de estar alineada con los ejes proporcionados por Nav2 para que este funcione de forma correcta, ya que no se han tenido en cuenta las rotaciones en los cambios de ejes realizados entre el sistema de referencia de Nav2 y el de la Raspberry Pi. El sistema de referencias de Nav2 puede observarse en Rviz en la parte inferior derecha, siendo el eje rojo el X, el verde el Y y el azul el Z.

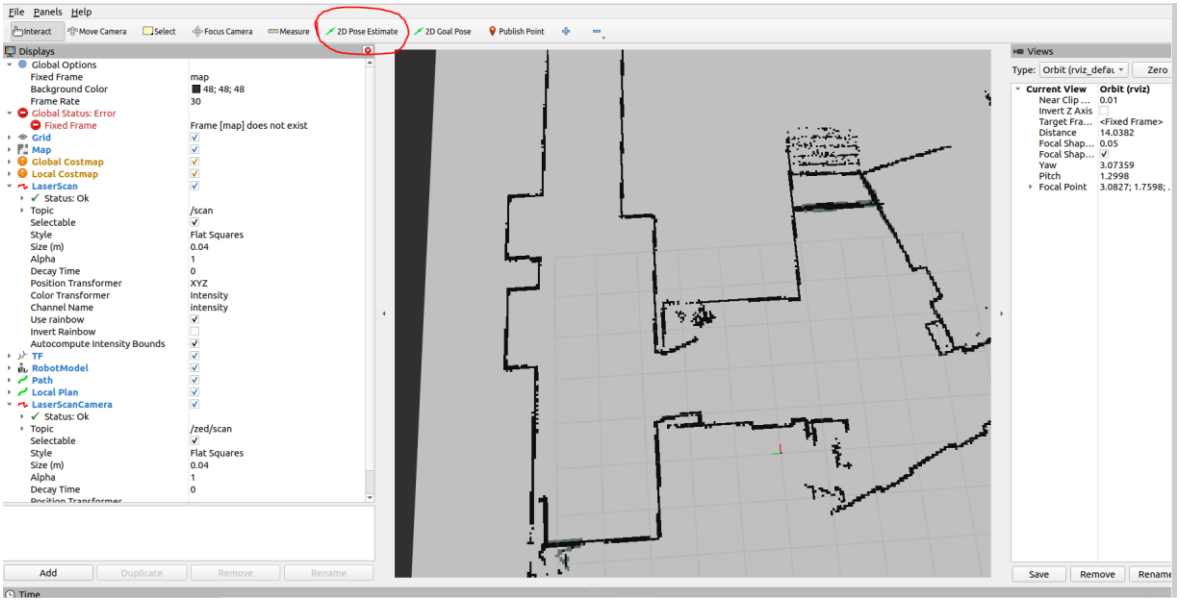

Figura 46: Situación de la opción 2D Pose Estimate

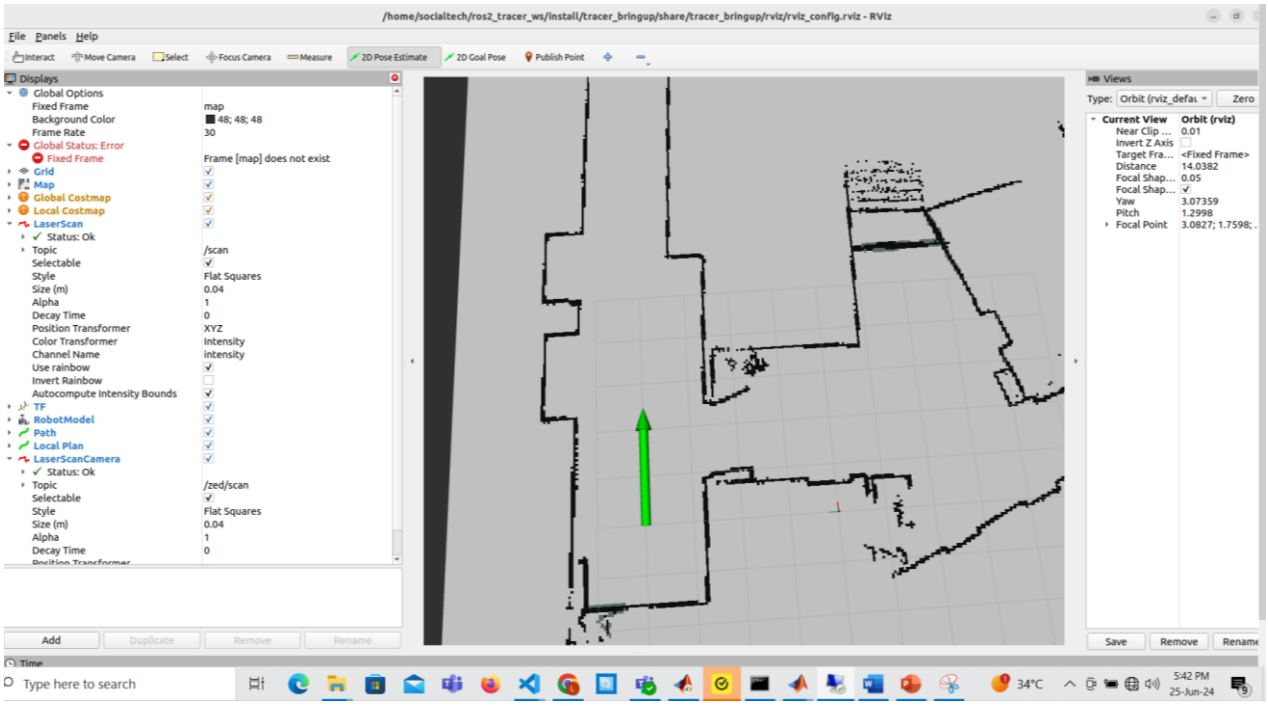

Figura 47: Selección de la pose inicial

6. Por último, debe de seleccionar una pose objetivo a la que moverse seleccionando `2D Goal Pose` y haciendo click con la orientación deseada en el mapa. Obsérvese la Figura 48.

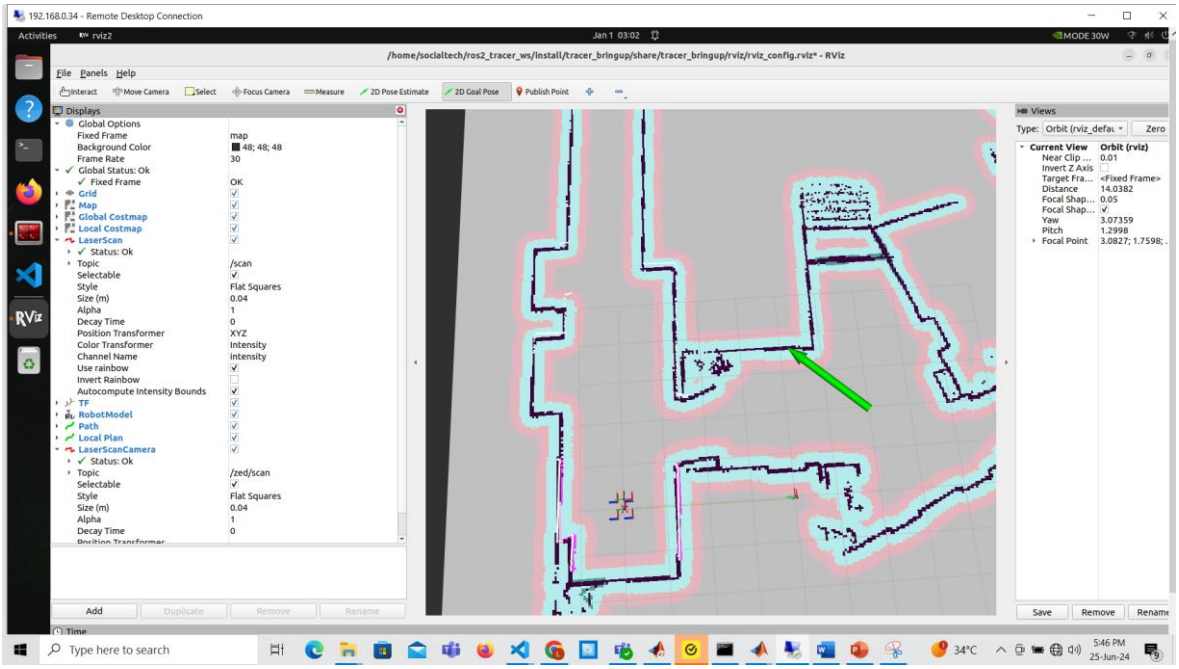

Figura 48: Selección de la pose objetivo

2. GENERACIÓN DE UN MAPA

Para generar un mapa, deberá de seguir los pasos 1, 2 y 3 de la sección anterior. Después, en la misma terminal, podrá comenzar la generación del mapa mediante el archivo de lanzamiento programado. Para ello ejecute el siguiente comando:

```
ros2 launch tracer_bringup tracer_real_scan.launch.xml
```

Debería de ver una imagen parecida a la de la Figura 49. Moviendo el robot con la emisora debería de poder escanear el entorno en el que esté trabajando con facilidad. Cuando haya terminado de generar el mapa, abra otra terminal (sin cerrar la anterior) y ejecute el siguiente comando:

```
ros2 run nav2_map_server map_saver_cli -f [path_to_directory/name]
```

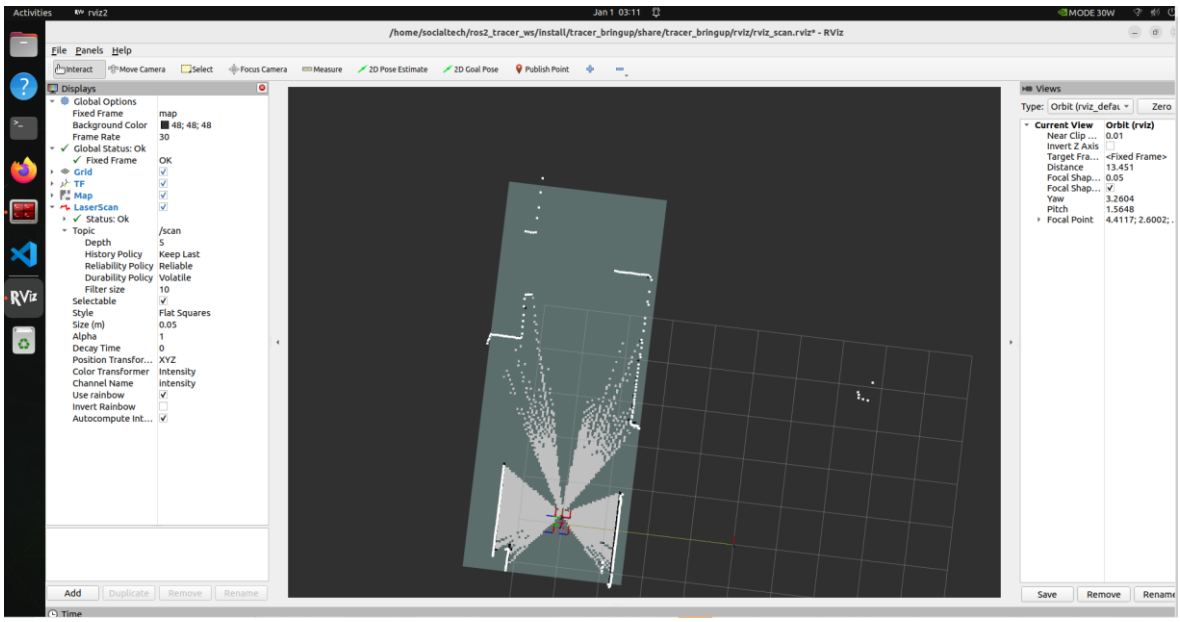

Figura 49: Generación de un nuevo mapa

Por supuesto deberá de completar con la ruta al directorio en el que desee guardar el mapa y el nombre que quiera que el archivo tenga. Para ejecutar la navegación autónoma con este mapa deberá de cambiar el parámetro en el archivo de lanzamiento asociado. En el directorio tracer_bringup/launch acceda al archivo `tracer_real.launch.xml`. Dentro de este, en la incialización de Nav2 cambie el parámetro map incluyendo el mapa deseado. El mapa deseado debe estar guardado en tracer_bringup/maps. Obsérvese la Figura 50, donde se detalla el parámetro exacto. Tras guardar el mapa, es necesario volver a construir el paquete tracer_bringup. Para ello, en una terminal, en el directorio ros2_tracer_ws ejecute el siguiente comando:

```
colcon build –packages-select tracer_bringup
```

```
<!-- Start the nav2 bringup launch file to start the navigation -->
<include file="$(find-pkg-share nav2_bringup)/launch/bringup_launch.py" >
    <arg name="map" value="$(find-pkg-share tracer_bringup)/maps/ultimate_corridor.yaml" />
    <arg name="params_file" value="$(find-pkg-share tracer_bringup)/params/nav2_params_real.yaml" />
    <arg name="use_sim_time" value="False" />
</include>
```

Figura 50: Parámetro usado para seleccionar el mapa de navegación

3. SEGUIMIENTO DE UNA SERIE DE *WAYPOINTS*

Para seguir un conjunto de *waypoints* especificados previamente, primero deberá de ejecutar los pasos 1, 2, 3 y 4 de la sección 1 de este anexo. Después, deberá de ejecutar el nodo waypoint_commander del paquete con el mismo nombre. Para ello, en otra terminal, vuelva a ejecutar los pasos 1 y 2 de la sección 1 de este anexo. Después, ejecute el nodo mediante este comando:

```
ros2 run waypoint_commander waypoint_commander
```

El robot debería de comenzar el recorrido por los *waypoints* especificados en el nodo. Para cambiar estos puntos, debe de acceder al archivo `waypoint_commander.py` dentro del paquete waypoint_commander. Debe de especificar la posición inicial del robot (parámetro self.initial_nav2_pose) y el conjunto de *waypoints* con el formato [posición x, posición y, ángulo en radianes]. A continuación se puede ver un extracto del código en el que se especifica la posición inicial del robot y 4 *waypoints* por los que pasará.

```
# Initial pose in the nav2 frames
self.initial_nav2_pose = self.create_pose_stamped(0.0, 4.0, 0.0)
# Send initial pose to nav2
self.nav.setInitialPose(self.initial_nav2_pose)
# Wait until nav2 is active
self.nav.waitUntilNav2Active()
# Generate the vector of poses to iterate over
self.poses = [np.array([2.0, 0.0, 1.57]),
              np.array([5.0, 0.10, 1.57]),
              np.array([30, -1.0, 3.14]),
              np.array([0.0, 0.0, 0.0])]
self.poses = [self.transform_vector_to_nav2_base(p) for p in
self.poses]
self.poses_stamped = [self.create_pose_stamped(p[0], p[1], p[2]) for
p in self.poses]
```

4. PROBLEMAS COMUNES

4.1 Funcionamiento del LiDAR

En ciertas ocasiones puede ocurrir que tras indicar la pose inicial en Rviz el robot no aparezca. Esto puede ocurrir por múltiples motivos, pero en la mayoría de los casos es algún error en la configuración del LiDAR. Para comprobar si el LiDAR está funcionando

correctamente, ejecute los pasos 1, 2 y 3 de la sección 1 de este anexo y compruebe el funcionamiento del LiDAR ejecutando el siguiente comando:

```
ros2 launch lidar_display.launch.xml
```

Este comando ejecuta un archivo de lanzamiento diseñado para mostrar la nube de puntos generada por el LiDAR en Rviz. Si esto no ocurre el problema se encuentra en el LiDAR (compruebe que ha especificado la ip correctamente). Un ejemplo de funcionamiento correcto puede observarse en la Figura 51.

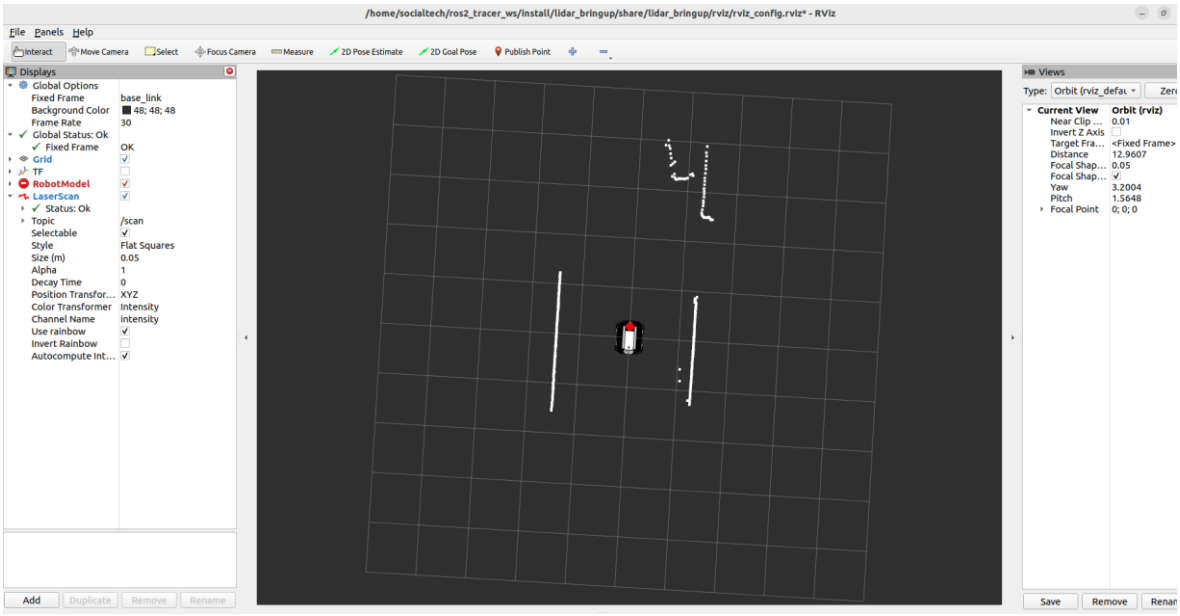

Figura 51: Funcionamiento correcto del LiDAR

4.2 Inicialización de la Raspberry Pi

Si el robot no se mueve cuando se le envía una pose objetivo, es posible que haya habido un error en la inicialización de la Raspberry Pi. Para comprobar esto, conéctese a la Raspberry Pi mediante ssh y ejecute el comando `htop`. Deberían de estar ejecutándose el servidor de comunicaciones que se conecta con el Tracer (server_coms.py), la máquina de estados (StateMachine.py) y lo programado en Matlab/Simulink. Si ha fallado el servidor de comunicaciones, se recomienda reiniciar el sistema completo (apagar el Tracer, esperar unos

10 segundos y volver a encenderlo). En caso de que haya fallado la máquina de estados, esta puede iniciarse de forma manual accediendo a home/pi/state_machine y ejecutando el script de Python `StateMachine.py`. Por último, si los programas asociados a Matlab/Simulink no se están ejecutando debe de haber ocurrido un error en la instalación del firmware asociado a estos programas, por lo que se recomienda reinstalarlo. Si se sabe que estos ya estaban instalados y funcionando, se recomienda reiniciar el sistema completo.

4.3 Error en el envío inicial de la pose

Si se revisa el código, se verá que el target_pose_publisher debe recibir la pose inicial del robot para luego enviar a la Raspberry Pi el resto de poses objetivos con respecto a esta, ya que el código ejecutado en la Raspberry Pi considera la pose inicial como el origen de coordenadas. En ciertas ocasiones, si el target_pose_publisher no se inicia suficientemente rápido, este puede recibir una pose inicial errónea, lo que hace que el robot salga disparado sin seguir la trayectoria. Para evitar este error, comente la sección del código donde se lanza el nodo target_pose_publisher en el archivo de lanzamiento `tracer_real.launch.xml` y, cuando inicie la navegación autónoma, ejecute el nodo target_pose_publisher de forma manual antes de ejecutar el archivo de lanzamiento `tracer_real.launch.xml`. En la terminal donde ha ejecutado el target_pose_publisher debería de poder ver la distancia a la siguiente pose objetivo cuando se haya planificado una ruta, si esta distancia es coherente con el *lookahead distance* especificado, entonces el error no ha ocurrido.

4.4 Bloqueo en el control de ángulo

Si se especifica una próxima pose objetivo cuando el robot todavía está tomando la orientación correcta en la pose anterior, este puede quedarse bloqueado en el control de ángulo, por lo que no se moverá de la pose. Para sacar al robot de este estado, dele como pose objetivo la pose actual y espere unos segundos.