Trabajo Fin de Máster
Ingeniería Industrial

Sistema de Navegación de Robot Terrestre para Agricultura

Autor: Julio Román Fernández
Tutor: Ángel Rodríguez Castaño

Dpto. de Ingeniería de Sistemas y Automática.
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2023

Trabajo Fin de Máster
Ingeniería Industrial

Sistema de navegación de robot terrestre para agricultura

Autor:
Julio Román Fernández

Tutor:
Ángel Rodríguez Castaño
Profesor Contratado Doctor

Dpto. de Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2023

Trabajo Fin de Máster: Sistema de navegación de robot terrestre para agricultura

Autor: Julio Román Fernández

Tutor: Ángel Rodríguez Castaño

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2023

El Secretario del Tribunal

A mi familia, por su apoyo incondicional.

A Pablo G-C., por su ayuda desinteresada.

Resumen

Este proyecto detalla el proceso de desarrollo de un entorno de simulación y la implementación de funciones de guiado para un robot terrestre agrícola, permitiéndole navegar autónomamente por un campo de brócolis sin causar daños. Para llevar a cabo este trabajo, se utilizó ROS en su versión Melodic y Gazebo para la simulación del entorno. Se implementó el algoritmo de navegación Pure Pursuit y, a través de una serie de experimentos, se analizó el impacto de dos de sus parámetros clave: la velocidad lineal y el valor de Look-Ahead. Los resultados de estos experimentos demuestran que el algoritmo produce resultados satisfactorios cuando se utilizan los valores adecuados para dichos parámetros.

Como una extensión del alcance del proyecto, se implementa un sistema de navegación más sofisticado conocido como Move Base Flex (MBF). Sorprendentemente los resultados obtenidos son menos satisfactorios en comparación con Pure Pursuit. No obstante, se cree que aún no se han alcanzado los parámetros óptimos. Existe confianza en que una vez se logre ajustar adecuadamente estos parámetros, el comportamiento del robot mejorará. En cualquier caso, MBF aporta nuevas funcionalidades interesantes que Pure Pursuit no, como por ejemplo la evasión de obstáculos.

Abstract

This project details the process of developing a simulation environment and implementing guidance functions for an agricultural ground robot, allowing it to navigate autonomously through a broccoli field without causing damage. To carry out this work, ROS was used in its Melodic version and Gazebo was used to simulate the environment. The Pure Pursuit navigation algorithm was implemented and, through a series of experiments, the impact of two of its key parameters were analysed: the linear velocity and the Look-Ahead value. The results of these experiments demonstrate that the algorithm produces satisfactory results when appropriate values for these parameters are used.

As an extension of the project scope, a more sophisticated navigation system known as Move Base Flex (MBF) is implemented. Surprisingly, the results obtained are less satisfactory compared to Pure Pursuit. However, it is believed that the optimal parameters have not yet been reached. There is confidence that once these parameters are properly tuned, the robot's behavior will improve. In any case, MBF brings new interesting functionalities that Pure Pursuit does not, such as obstacle avoidance.

Índice

Lista de Figuras

REFERENCIAS

Ref.	Título	Año	Autor (es)
[1]	Agribot https://github.com/PRBonn/agribot/	2022	Ahmadi, Alireza
[2]	Agribot_TFM-MII https://github.com/julioRF5/Agribot_TFM-MII	2023	Román Fernández, Julio
[3]	Modelo cinemático y simulación de un robot móvil diferencial	2021	www.roboticoss.com
[4]	Definición y Requisitos de movimientos. Proyecto ELISA.	2021	Martínez López, Alejandro
[5]	Robot Operating System (ROS)	-/-	R. Suárez , J. Rosell, M. Vinagre, F. Cortes, A. Ansuategui, I. Maurtua, D. Martin, A. Guash, J. Azpiazu, D. Serrano, N. García.
[6]	http://models.gazebosim.org/	-	-
[7]	https://github.com/leofansq/	2022	Siqi Fan
[8]	ACCURATE PATH TRACKING BY ADJUSTING LOOK-AHEAD POINT IN PURE PURSUIT METHOD	2020	Joonwoo Ahn , Seho Shin, Minsung Kim and Jaeheung Park
[9]	Implementation of Pure Pursuit Algorithm for Nonholonomic Mobile Robot using Robot Operating System	2021	Omur Aydogmus and Gullu Boztas
[10]	Move Base Flex A Highly Flexible Navigation Framework for Mobile Robots	2018	Sebastian Putz, Jorge Santos Simon and Joachim Hertzberg
[11]	https://github.com/magazino/move_base_flex	2022	Magazino
[12]	http://wiki.ros.org/teb_local_planner	2020	Christoph Roesmann

1. INTRODUCCIÓN

1.1 Marco

La agricultura de precisión ha experimentado avances significativos en los últimos años gracias a la aplicación de tecnologías robóticas y de automatización. El uso de sistemas autónomos en la agricultura permite dar solución a desafíos que van desde la escasez de mano de obra hasta las condiciones climáticas impredecibles. Estos desafíos crean la necesidad de obtener soluciones más confiables, menos costosas y más efectivas. En este contexto, el desarrollo de un sistema de navegación móvil para recorrer de forma autónoma una plantación de brócolis representa un enfoque innovador para el cultivo de este vegetal.

El brócoli es una hortaliza ampliamente cultivada debido a su valor nutricional y demanda en el mercado. Sin embargo, el manejo de una plantación de brócolis presenta desafíos en términos de mantenimiento, monitoreo y cosecha. La detección temprana de enfermedades, el control de malezas y la supervisión del crecimiento son aspectos críticos para garantizar una producción exitosa. El desarrollo de un sistema de navegación móvil puede proporcionar una solución automatizada y precisa para abordar estos desafíos.

Los sistemas de navegación móvil se basan en la integración de tecnologías de percepción y control para permitir que un robot se desplace de manera autónoma en un entorno específico. Estos sistemas utilizan sensores como cámaras, lídar y ultrasonidos para obtener información del entorno circundante, y algoritmos de planificación de ruta para determinar la trayectoria óptima a seguir. Además, los sistemas de navegación móvil pueden utilizar actuadores para evitar obstáculos y garantizar un movimiento suave y seguro.

Los sistemas de percepción pueden utilizar técnicas de visión por computadora para detectar y reconocer plantas, malezas, enfermedades u otros elementos relevantes en la plantación. Además, los sensores de proximidad y obstáculos, como lídar o ultrasonidos, son fundamentales para evitar colisiones y garantizar la seguridad del robot y las plantas. Por otro lado, la planificación de la ruta también es esencial para guiar al robot de manera eficiente a través de la plantación de brócolis. Los algoritmos de planificación de ruta deben considerar la distribución de las plantas, la topografía del terreno y otros obstáculos presentes en el entorno. Además, es importante incorporar estrategias de control que permitan ajustar la velocidad, la dirección y la respuesta del robot en tiempo real para adaptarse a cambios en el entorno o en las condiciones de cultivo.

El proyecto aquí expuesto utiliza el trabajo desarrollado por Alireza Ahmadi, de la Universidad de Bonn, para contribuir al proyecto liderado por el Grupo de Robótica, Visión y Control (GRVC) de la Universidad de Sevilla, conocido como ELISA, el cuál pretende desarrollar un sistema de navegación para recorrer con un vehículo terrestre una plantación de brócolis de manera autónoma. Para más información sobre el repositorio de GitHub de la Universidad de Bonn del que parte este proyecto vea la ref. [1]. Este repositorio de partida ha sido ampliado y modificado con nuevos desarrollos que dan respuesta a los objetivos del proyecto en cuestión. El nuevo repositorio resultante con la solución al trabajo expuesto en este documento puede encontrarse en la ref. [2].

La motivación del autor para llevar a cabo este trabajo fue el deseo de aprender más sobre robótica móvil, así como intentar entender si es un sector en el que quiere desarrollarse profesionalmente. Además, tal y como se explica anteriormente, aporta conocimiento a un problema de actualidad sufrido por el sector agrícola.

1.2 Alcance

Aunque el repositorio en el que se basa este proyecto (Ref. [1]) es extenso y desarrolla diversos paquetes que aportan funcionalidades variadas al robot, en este proyecto solo se explicarán los que sean relevantes para el trabajo aquí expuesto. Particularmente, este trabajo se ha enfocado únicamente

en la parte de navegación y de simular el entorno por donde se mueve el vehículo.

Los objetivos pueden dividirse en dos: objetivos técnicos y objetivos personales.

Los objetivos técnicos son aquellos que se han acordado con el tutor del proyecto y cuyo propósito es darle una funcionalidad concreta al robot. Son los siguientes:

- Crear un modelo en Gazebo de una plantación de brócolis. La plantación debe ser de 10 metros de largo por 8 metros de ancho. Tendrá 6 hileras, a su vez con dos filas de brócolis en cada hilera. La separación entre hileras es de 1.6 metros. La separación entre brócolis de una misma hilera debe ser equidistante.
- Diseño y desarrollo de un sistema de navegación autónoma del vehículo basado en un sistema de localización externo absoluto, como un sistema GNSS. El objetivo del sistema de navegación es circular por las calles de la plantación sin dañar los brócolis. Las coordenadas que definen cada calle se suponen conocidas a priori.

Los objetivos personales son la razón por la que se ha realizado este proyecto. Son todos aquellos que se ha propuesto el autor por motivación propia. Son los siguientes:

- Tener una visión general y aproximada de qué implica trabajar como ingeniero en un departamento de Desarrollo e Innovación de una empresa de robótica móvil. Para ello se propone una metodología de trabajo (ver Sección 1.3) que sea la más cercana a la realidad posible.
- Mejorar los conocimientos de programación, especialmente en los lenguajes más usados en robótica móvil, como por ejemplo C++ y Python.
- Aprender software específico de robótica usado en las empresas del sector hoy día, como por ejemplo ROS.
- Aprender aplicaciones empleadas en empresas para el desarrollo de código en equipo y control de versiones, como por ejemplo Git y GitHub.

1.3 Metodología

En este apartado se presenta de forma descriptiva la metodología seguida para la ejecución del presente proyecto, indicando las actividades llevadas a cabo en las distintas fases del proyecto.

En las etapas iniciales del proyecto, se definieron tanto el alcance como los objetivos del mismo. Posteriormente, se llevó a cabo una investigación sobre diversas tecnologías para su implementación, considerando la posibilidad de usar ROS2. Sin embargo, esta opción se descartó finalmente porque todo el trabajo previo del que parte este proyecto está desarrollado en ROS1, lo que podría generar problemas de compatibilidad.

Para gestionar las versiones del código y permitir la revisión por parte del tutor del trabajo fin de máster, se optó por utilizar Git y GitHub. Además, para facilitar la colaboración y el trabajo conjunto, se eligió Google Drive Docs para la redacción de la memoria, permitiendo que varias personas pudieran editar el documento simultáneamente.

Dado que ROS, Gazebo, Git y GitHub eran tecnologías nuevas para el autor de este trabajo, se realizaron tutoriales previos al inicio del proyecto. Una vez se adquirieron los conocimientos necesarios se comenzó con el trabajo. En primer lugar, se diseñó y desarrolló el entorno de simulación para poder probar visualmente todas las implementaciones que se llevarían a cabo a posteriori. Luego, se investigaron diferentes algoritmos de navegación y se seleccionaron los más adecuados para la implementación. Para ello, se buscaron alternativas en GitHub desarrolladas por otros usuarios, asegurándose de que fueran compatibles con la versión de ROS y Ubuntu utilizada, y estuvieran bien documentadas y estructuradas.

El ajuste de los parámetros del algoritmo al robot utilizado y a la aplicación en cuestión se realizó de forma experimental mediante ensayo y error. Este proceso implicó varias iteraciones para observar el comportamiento del robot en la simulación y comprender cómo afectan los valores de los parámetros al desempeño del autómata.

2. DESCRIPCIÓN DEL SISTEMA

En esta sección se muestra el robot utilizado, se explica brevemente su modelo cinemático, los componentes que forman el sistema de accionamiento del vehículo y el sistema de dirección y navegación.

2.1 Modelo cinemático del vehículo

Para la recolección de brócolis se usa un robot móvil terrestre diferencial con orugas (**Figura 1**). Este tipo de vehículos tienen el mismo movimiento/configuración que un robot diferencial con ruedas en cuanto a modelo cinemático. Es por eso que, tal y como se menciona en la sección 4, puede utilizarse en la simulación el modelo de un robot diferencial con ruedas.

Figura 1 - Robot seleccionado

Un robot con tracción diferencial es un sistema que utiliza un sistema de transmisión de dos ruedas independientes, cada una de ellas unida a su propio motor. Por lo tanto, la locomoción del vehículo se basa en la diferencia de velocidades de cada una de sus ruedas instaladas en un único eje. Normalmente, además de las ruedas activas, se coloca una rueda libre que gira pasivamente y sirve para dar estabilidad al robot. En el caso del vehículo con orugas utilizado el funcionamiento es equivalente, salvo que no necesita ruedas adicionales para la estabilidad.

El principio de funcionamiento de un robot diferencial es el siguiente:

- Movimiento en línea recta: Las dos ruedas activas giran en el mismo sentido y velocidad. Dependiendo del sentido de giro el robot se desplazará hacia adelante o hacia atrás (**Figura 2a**)
- Movimiento de giro: Las dos ruedas activas giran en el mismo sentido, pero con velocidades diferentes. La rueda con mayor velocidad hará girar el vehículo sobre la rueda con menor velocidad. Es decir, si la rueda izquierda tiene mayor velocidad que la derecha el vehículo girará hacia la derecha (**Figura 2b**).
- Movimiento de giro sobre el propio eje del robot: Ambas ruedas activas giran a la misma velocidad, pero en sentidos opuestos (**Figura 2c**).

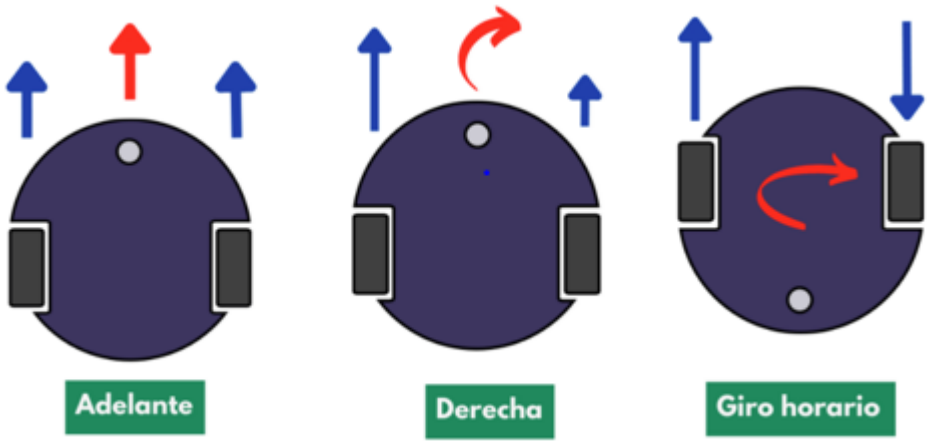

Figura 2 - Movimientos de un robot diferencial 2a) Recto. 2b) Giro. 2c) Giro sobre su propio eje

Puede encontrar más información sobre robots diferenciales en la ref. [3]

Aunque el diseño del vehículo a nivel mecánico está fuera del alcance de este proyecto, se describe brevemente su accionamiento y funcionamiento para la recogida de brócolis. La siguiente información ha sido directamente extraída del documento que recoge la definición y requisitos de movimientos del robot para el proyecto ELISA (Ref. [4]).

2.2 Accionamientos en la cosechadora

- Oruga:

  Cada oruga dispone de un motor hidráulico en circuito cerrado con una bomba de caudal variable controlada por una válvula proporcional con una señal de intensidad de 4 a 20 mA. Estas bombas están accionadas por un motor eléctrico asíncrono de 15 kW.

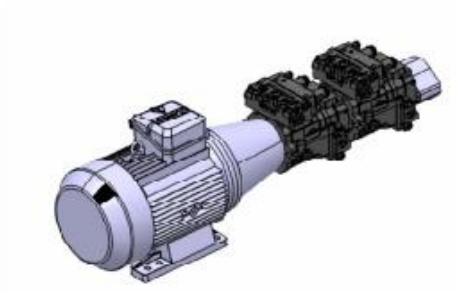

Figura 3 – Conjunto motobomba

- Robot cartesiano:

  Compuesto por 3 servomotores para los movimientos X,Y y Z más un motor paso a paso para un movimiento en Z de 200 mm.

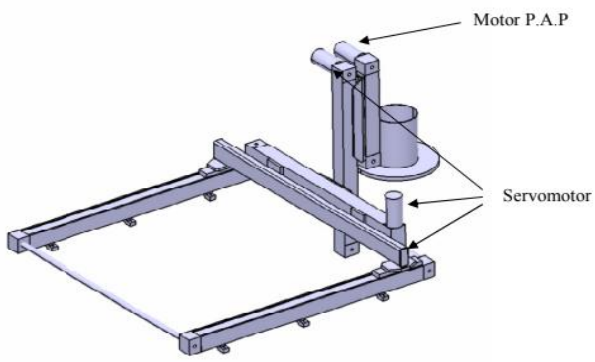

Figura 4 - Estructura del robot cartesiano

- Cabezal de corte:

  A definir.

- Cinta transportadora:

  Dispone de un motor eléctrico asíncrono (potencia estimada de 1,5 KW).

Figura 5 - Cinta transportadora

- Tolva:

  Dispone de dos accionamientos, un motor eléctrico para el movimiento transversal y dos cilindros hidráulicos controlados por una electroválvula con dos bobinas de 24V DC para la apertura y cierre de ésta. Un encoder proporciona la lectura de la posición transversal y dos sensores de posición indican si está abierta o cerrada.

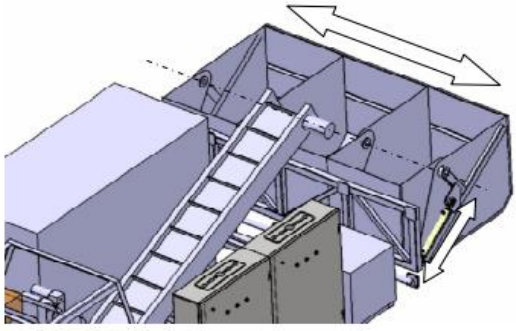

Figura 6 – Tolva de descarga

2.3 Funcionamiento en recogida

Robot cartesiano:

Se define como origen (homing) donde se deposita el brócoli en la cinta transportadora coordenadas (x_0, y_0, z_0):

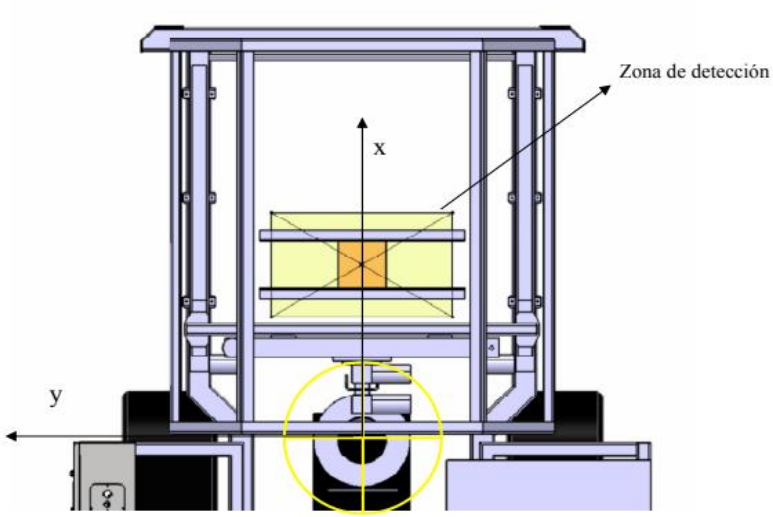

Figura 7 - Funcionamiento en recogida del robot cartesiano

Una vez recibidas las coordenadas de recogida (x_r, y_r, z_r), los movimientos secuenciales serían los siguientes:

- Desplazamiento a las coordenadas (x_{01}, y_0, z_0). Donde x_{01}es un desplazamiento en x para librar la tolva donde se deposita el brócoli (a definir según tamaño del cabezal de corte).
- Desplazamiento a las coordenadas $(x_r, y_r, -200)$, incluye un movimiento en z del primer actuador de carrera 200 mm.
- Desplazamiento a las coordenadas (x_r, y_r, z_r). Movimiento en z con el segundo actuador.

Una vez recogido el brócoli, para depositarlo en la tolva, el camino de vuelta sería el inverso al de recogida.

Cinta transportadora:

La cinta transportadora es fija en el chasis y traslada el brócoli desde la tolva de recepción a la tolva de descarga.

Tolva de descarga:

La tolva de descarga está formada por tres compartimientos, los cuales se irán llenando durante la recolección del brócoli. Si uno se completa, la tolva realiza un desplazamiento para situar otro compartimento debajo de la cinta transportadora. Para la detección de llenado se utiliza una célula de carga. También se utiliza el desplazamiento lateral para alinear la tolva con los palots a la hora de la descarga de brócolis. Una vez alineados se produce la apertura de la tolva gracias a los cilindros hidráulicos.

2.4 Sistema de dirección y navegación – Orugas

- El sistema de dirección viene implementado de forma intrínseca en el mecanismo de tracción, que son las orugas.
- Con las válvulas proporcionales (señal de 4 a 20 mA) se controla el caudal que recibe cada motor instalado en la oruga. Con un caudal diferencial en las orugas se consigue el girado del vehículo. En función de dicha diferencia se consigue un mayor o menor radio de giro.
- Los motores instalados en las orugas son de doble velocidad, se tiene una marcha corta y otra larga. Se controla por medio de una bobina de 24 V DC.
- El radio de giro del vehículo puede llegar a 0 m (girar sobre su propio eje) si los motores de las orugas giran en sentido opuesto recibiendo cada uno el mismo caudal.
- Para la navegación, se va a instalar un encoder en cada oruga para que sirva de odómetro. Esta medida es necesaria para hacer una estimación de la posición del robot por la deriva.

3. DESARROLLO DEL ENTORNO DE SIMULACIÓN

Para poder probar los algoritmos, la aplicación que se ha desarrollado y el comportamiento del robot, se simula un entorno lo más realista posible. Para ello, se recurre a un complemento de ROS llamado Gazebo. Este simulador ofrece la posibilidad de simular con precisión una gran variedad de robots, objetos y sensores. Gazebo es capaz de generar tanto la realimentación realista de sensores, como las interacciones entre objetos físicos, incluyendo el modelado de la física del cuerpo rígido.

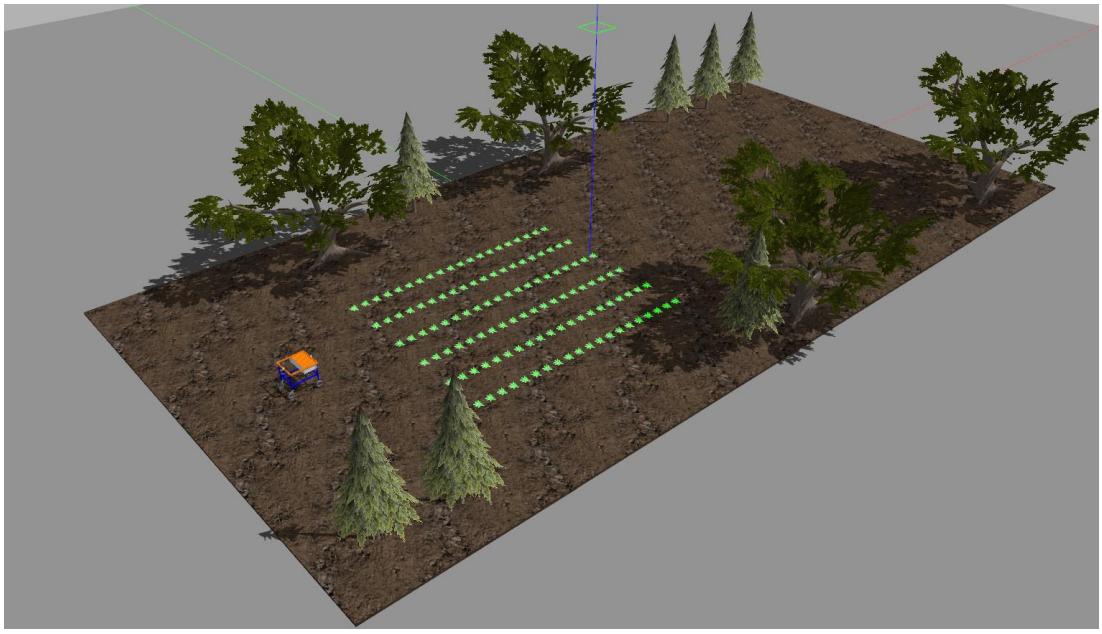

Figura 8 - Simulación en Gazebo

Para ejecutar la simulación se crea un *launch* llamado ***agribot_farm.launch***. Este es el encargado de lanzar el mundo Gazebo que está definido en ***FarmWithCropRow.world***, así como el modelo del robot colocándolo en una posición predefinida.

3.1 Entorno de desarrollo utilizado

El sector comercial de la robótica prioriza la dependencia del proveedor sobre la innovación. Cada robot dispone de su propio sistema operativo y lenguajes de programación, lo que implica que al programar un robot se escribe código que no es reutilizable en otros tipos de robots. Mantener la tecnología en sistemas cerrados obliga a los programadores a dedicar mucho esfuerzo en la puesta en marcha del hardware que compone su banco de ensayos, dejando poco tiempo para la innovación. Para hacer frente al tiempo improductivo surge ROS (Robot Operating System), un framework de código abierto que permite la reutilización de código dando soporte a distintos tipos de robots y de aplicaciones.

ROS ha llegado para quedarse. Es uno de los frameworks para desarrollo de robots más utilizado en la comunidad científica, y que poco a poco también intenta hacerse cabida en el ámbito industrial con la adopción de ROS-I.

El proyecto aquí expuesto se ha desarrollado íntegramente en ROS, en su versión Melodic. Respecto al Sistema Operativo de la máquina se ha optado por Ubuntu 18.04.6 LTS por su compatibilidad con ROS Melodic. Por tanto, el correcto funcionamiento del código desarrollado en este proyecto puede garantizarse para el Sistema Operativo y la versión de ROS anteriormente nombrada.

Es sabido que ROS sigue evolucionando, entre otras cosas pensando en su adopción por el ámbito industrial, el cuál ha introducido nuevos requerimientos. Allá por el año 2014, para poder implementar más adecuadamente estos requerimientos y dar un mejor soporte al mundo industrial, comienza la implementación de ROS2. Si uno de los objetivos personales del autor de este proyecto era obtener conocimiento en las herramientas utilizadas en el desarrollo de robots móviles, quizás, hubiera tenido más sentido ganar experiencia en ROS2. Sin embargo, hay un motivo fundamental por el que se descarta la idea de usar ROS2, y es que desafortunadamente ROS2 no mantiene la compatibilidad hacia atrás, lo que significa que en general el código desarrollado en ROS no funcionará en ROS2. El repositorio que se utiliza como referencia para este proyecto usa ROS, por tanto es motivo suficiente para descartar el uso de ROS2. La parte positiva es que ROS2 se basa en ROS, por lo que no es un sistema completamente nuevo. Muchos de los conceptos y aplicaciones en los que se apoya ROS siguen presentes en ROS2, por lo que no debe ser difícil aprender ROS2 en el futuro.

Para una información más detallada acerca de los orígenes de ROS, sus características y prestaciones y casos de uso, entre otros, consulte ref. [5].

3.2 Mundo Gazebo

Los archivos con extensión *.world* usan SDF (Formato de descripción de simulación). SDF es un formato XML para describir los objetos y entornos contenidos en la simulación.

El mundo Gazebo viene definido en ***FarmWithCropRow.world*** mediante código XML. Para la creación de este código se ha desarrollado un archivo en Python llamado ***GazeboCropRowGenerator.py***. Este archivo de Python automatiza la generación del código necesario para simular el entorno, así como guardar la posición en la que se encuentran los brócolis en un fichero de texto llamado ***waypoints.txt***, que será usado para la navegación del robot.

GazeboCropRowGenerator.py está parametrizado permitiendo modificar el entorno simulado de forma rápida y sencilla. Recibe, entre otras cosas, el número de hileras de brócolis que se desea, la longitud de las mismas y el número máximo de brócolis por hilera. En base a estos parámetros el código se encarga de distribuir los brócolis a lo largo de cada hilera de forma equidistante. Como resultado el fichero en cuestión genera el código XML que describe los brócolis, su posición, así como otros elementos presentes en la simulación que serán descritos a continuación (terreno y distintos tipos de árboles). Todo este código XML se guarda en ***FarmWithCropRow.world***.

Gazebo es un simulador de código abierto que cuenta con una gran comunidad que lo soportan. Gracias a esto hay una gran cantidad de modelos de objetos ya creados que pueden encontrarse en un repositorio (Ref. [6]). Tanto para los árboles como para el terreno se ha recurrido a este repositorio. La forma de usar estos modelos ya creados es sencilla. Una de las opciones que hay es simplemente descargar la carpeta en cuestión de la Ref. [6] y guardarlo en la carpeta local */.gazebo/models* que se crea una vez instalado Gazebo. Luego, en el código en el que se describe el mundo (archivo .world) habrá que poner la ruta a ese directorio. Cabe destacar que este repositorio ofrece junto con los modelos el código SDF que describe al elemento. Habrá que copiar este código y pegarlo en nuestro archivo *.world,* siendo necesarios algunos cambios para ajustarlo a nuestras necesidades. En el caso de este proyecto solo fue necesario modificar la posición en el que colocar cada objeto.

- Terreno:

  Tal y como se mencionó antes se ha descargado del repositorio (Ref. [6]). Puede encontrarse bajo el nombre de */mud_box.* La textura puede verse en la **Figura 9**. Respecto al uso que se le ha dado en este proyecto, se ha utilizado como el terreno de la plantación de brócolis. La dimensión de cada trozo de terreno es de 8x10 metros. Se han usado un total de 10 trozos consiguiendo un terreno de 40x20 metros.

Figura 9 - Textura del terreno

- Brócolis:

  En el caso de los brócolis, su modelo y código SDF se obtuvo del repositorio de GitHub del que parte este proyecto (Ref. [1]). En la siguiente figura se muestra el aspecto del modelo usado para una planta de brócoli.

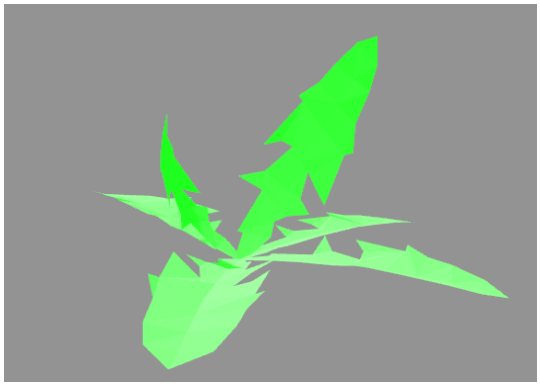

Figura 10 - modelo de la planta de brócoli

La plantación de brócolis cumple con uno de los objetivos del proyecto (Sección 1.2). Es de 10 metros de largo por 8 metros de ancho. Tiene 6 hileras, a su vez con dos filas de brócolis en cada hilera. La separación entre hileras es de 1.6 metros. Por último, la separación entre brócolis de una misma hilera es equidistante.

- Árboles:

  Se han usado dos tipos de árboles que, tal y como se mencionó antes, se han descargado del repositorio (Ref. [6]). El único propósito de estos árboles es decorar el entorno, aunque pueden llegar a dificultar la navegación del robot debido a que es posible una colisión con ellos.

  El primero de los árboles utilizado es un pino y puede encontrarse en el repositorio bajo el nombre de */pine_tree*. El aspecto del modelo utilizado se muestra en la **Figura 11**. Las texturas del tronco y hojas en la **Figura 12**.

Figura 11 - Modelo del pino

Figura 12 - Texturas tronco y hojas del pino

El siguiente árbol utilizado es el Quercus y puede encontrarse en el repositorio (Ref. [6]) bajo el nombre de */oak_tree*. El aspecto del modelo utilizado se muestra en la **Figura 13**. Las texturas del tronco y de las hojas en la **Figura 14**.

Figura 13 - Modelo del árbol Quercus

Figura 14 - Texturas tronco y hojas del Quercus

3.3 Robot en Gazebo

Como se explicó en la sección 2, el vehículo utilizado es un vehículo tipo oruga. Sin embargo, debido a que el movimiento/configuración es el mismo que un robot diferencial con ruedas en cuanto a modelo cinemático, se propone usar un robot de este tipo. De esta forma, no solo se simplifica el modelado del robot, sino que se puede hacer uso del trabajo ya realizado en el repositorio que usamos como referencia (Ref. [1]), evitando así tener que modelar el vehículo y dedicar el tiempo en el objetivo real de este proyecto; la navegación autónoma por el campo de brócolis. Por lo tanto, debido a que el objetivo principal de este proyecto no era modelar el vehículo, se hace uso de una simulación ya existente.

Aunque este documento no dará información detallada sobre el procedimiento de modelado, a continuación se muestra el vehículo a utilizar en Gazebo, así como datos generales sobre el robot.

Los bocetos y modelos primarios se han realizado en el software de simulación Solidworks y después se han aplicado desarrollos en Autodesk Fusion 360. El modelo final es el mostrado en la imagen siguiente.

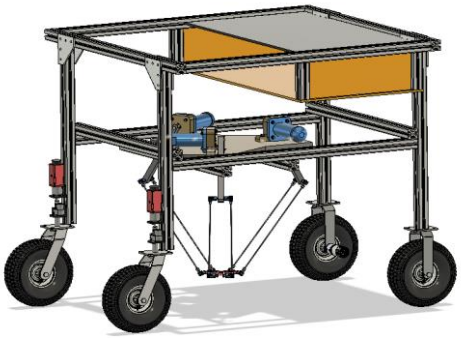

Figura 15 - Modelo CAD del robot diferencial

Tiene una sección superior de forma cuadrada y cuatro patas de 1052 mm de altura, cada una de las cuales sostienen una rueda. Las ruedas que proporcionan el movimiento son las traseras, siendo las delanteras las pasivas.

En cuanto a las dimensiones, las ruedas cuentan con un radio de 150 mm, la altura del vehículo es de 1052 mm con un ancho de 1000 mm y profundidad de 1080 mm. La distancia de centro a centro de las ruedas delanteras y traseras es de 1170 mm. El peso se calcula en unos 35 Kg. En las siguientes imágenes se muestra un esquema con las dimensiones.

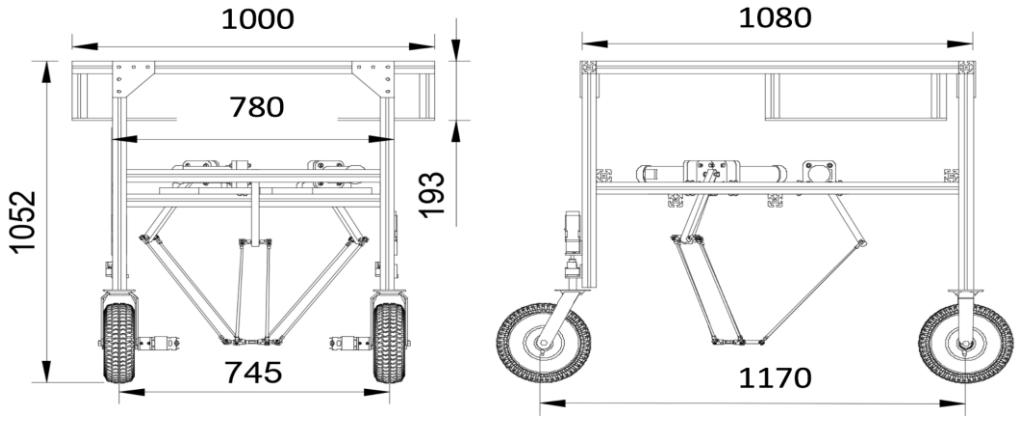

Figura 16 - Dimensiones del robot diferencial

Finalmente, el aspecto del robot que se utiliza en la simulación de Gazebo se muestra en la **Figura 17**. Puede verse como el sistema de recogida de brócolis, al no ser de interés para este proyecto, no se incluye en el modelo.

Figura 17 - Modelo del robot usado en Gazebo

4. DESARROLLO DE FUNCIONES DE GUIADO DEL VEHÍCULO

En el presente capítulo se explica la implementación de un conocido algoritmo de navegación, el algoritmo de persecución pura o como es más conocido por su nombre en inglés Pure Pursuit. Se probará con distintos parámetros y se comentarán sus resultados.

Antes de aclarar todo lo relacionado con el algoritmo de navegación se dedican unas líneas a explicar el testeo previo que se hizo del modelo del robot, así como la herramienta que se usó también a posteriori para comprobar que el robot recorre el camino correcto y con qué precisión.

4.1 Pruebas iniciales al modelo del robot

Antes de empezar a implementar cualquier algoritmo de navegación y realizar tareas más complejas conviene testear el comportamiento del robot. Se recuerda que el modelo del robot ha sido reutilizado en su totalidad por lo que se desconoce cómo está hecho y si funciona correctamente.

En primer lugar, una vez lanzada la simulación, es de interés conocer todos los topics disponibles. Esto puede saberse fácilmente escribiendo por terminal el comando **rostopic list.** El resultado es el mostrado en la figura siguiente.

```
juliorf@juliorf-pc:~/catkin_ws/src/julio_tfm/agribot-master$ rostopic list
/agribot/back_camera/camera_info
/agribot/back_camera/image_raw
/agribot/back_camera/image_raw/compressed
/agribot/back_camera/image_raw/compressed/parameter_descriptions
/agribot/back_camera/image_raw/compressed/parameter_updates
/agribot/back_camera/image_raw/compressedDepth
/agribot/back_camera/image_raw/compressedDepth/parameter_descriptions
/agribot/back_camera/image_raw/compressedDepth/parameter_updates
/agribot/back_camera/image_raw/theora
/agribot/back_camera/image_raw/theora/parameter_descriptions
/agribot/back_camera/image_raw/theora/parameter_updates
/agribot/back_camera/parameter_descriptions
/agribot/back_camera/parameter_updates
/agribot/front_camera/camera_info
/agribot/front_camera/image_raw
/agribot/front_camera/image_raw/compressed
/agribot/front_camera/image_raw/compressed/parameter_descriptions
/agribot/front_camera/image_raw/compressed/parameter_updates
/agribot/front_camera/image_raw/compressedDepth
/agribot/front_camera/image_raw/compressedDepth/parameter_descriptions
/agribot/front_camera/image_raw/compressedDepth/parameter_updates
/agribot/front_camera/image_raw/theora
/agribot/front_camera/image_raw/theora/parameter_descriptions
/agribot/front_camera/image_raw/theora/parameter_updates
/agribot/front_camera/parameter_descriptions
/agribot/front_camera/parameter_updates
/clock
/cmd_vel
/gazebo/link_states
/gazebo/model_states
/gazebo/parameter_descriptions
/gazebo/parameter_updates
/gazebo/set_link_state
/gazebo/set_model_state
/imu/data
/joint_states
/odom
/pose
/rosout
/rosout_agg
/tf
```

Figura 18 - Lista de topics iniciales disponibles al lanzar la simulación

Se aprecian varios topics sobre la cámara frontal y trasera, pero los obviamos porque no se utilizará la cámara para nada en esta aplicación. Otros topics están relacionados con Gazebo y no con el robot, por lo que en principio tampoco son de interés. Hay otros topics que permiten conocer los datos de la IMU, la posición actual del robot y su odometría. Estos tres pueden ayudar a entender si el robot se mueve, a qué velocidad y qué orientación tiene. Por último mediante el topic */cmd_vel* se puede publicar la velocidad lineal y angular. Será este último topic, junto con el de odometría */odom,* los que se usan para probar el modelo.

Para probar el correcto funcionamiento del robot se introducen distintas velocidades por terminal mediante el topic */cmd_vel.* La forma de hacer esto es publicando en dicho topic con el comando **rostopic pub /cmd_vel geometry_msgs/Twist**, siendo *Twist* el tipo de mensaje que usa este topic para enviar los datos. Este comando arroja la siguiente información que hay que rellenar:

```
^Cjuliorf@juliorf-pc:~/catkin_ws/src/julio_tfm/agribot-master$ rostopic pub /cmd_vel geometry_msgs/Twist "linear:
  x: 0.0
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 0.0"
```

Figura 19 - Publicar en el topic /cmd_vel

Se realizaron las siguientes tres pruebas:

- Velocidad lineal en X: Este es el único eje en el que puede avanzar el robot, ya que sus ruedas activas no permiten el giro. Al ser diferencial, la forma de girar es mediante la diferencia de velocidades de ambas ruedas activas.
- Velocidad en X y giro horario: El giro horario se consigue introduciendo en la velocidad angular del eje Z un valor negativo. Se observa que el robot al hacer el giro con una velocidad de unos 3 rad/s o más no mantiene bien el control, provocando incluso el deslizamiento de las ruedas. Esto indica que cuando se implemente el algoritmo de navegación habrá que limitar de alguna forma la velocidad angular máxima con la que puede tomar las curvas.
- Velocidad en X y giro antihorario: El giro antihorario se consigue introduciendo en la velocidad angular del eje Z un valor positivo. Al igual que en el giro horario, el robot no es capaz de mantener un buen control en el giro para velocidades angulares mayores de unos 3 rad/s.

Una forma de comprobar que lo que se publica en */cmd_vel* le llega al robot, es viendo lo que recibe el topic */odom*. Esto puede hacerse mediante el comando **rostopic echo /odom.** Se observa que sí recibe la velocidad, sin embargo no exactamente la que se le envía, siempre entre un 10% y un 15% menos. No es algo que extrañe demasiado ya que se conoce que el cálculo de velocidades mediante la odometría no es la solución más precisa.

4.2 Herramienta de depuración: Rviz

Además de la opción ya explicada para hacer el testeo inicial mediante terminal, también se pueden publicar velocidades de una forma más sencilla y amena a través de una herramienta que ofrece ROS llamada **Rviz.** Esta herramienta no solo se usó para el testeo inicial sino para probar y depurar la implementación de los algoritmos de navegación. En esta subsección se hace una introducción a qué es Rviz y cómo se usó para realizar los mismos test que se han explicado anteriormente usando la terminal. En la subsección siguiente se volverá a mencionar esta herramienta para explicar cómo se ha verificado el algoritmo de navegación implementado.

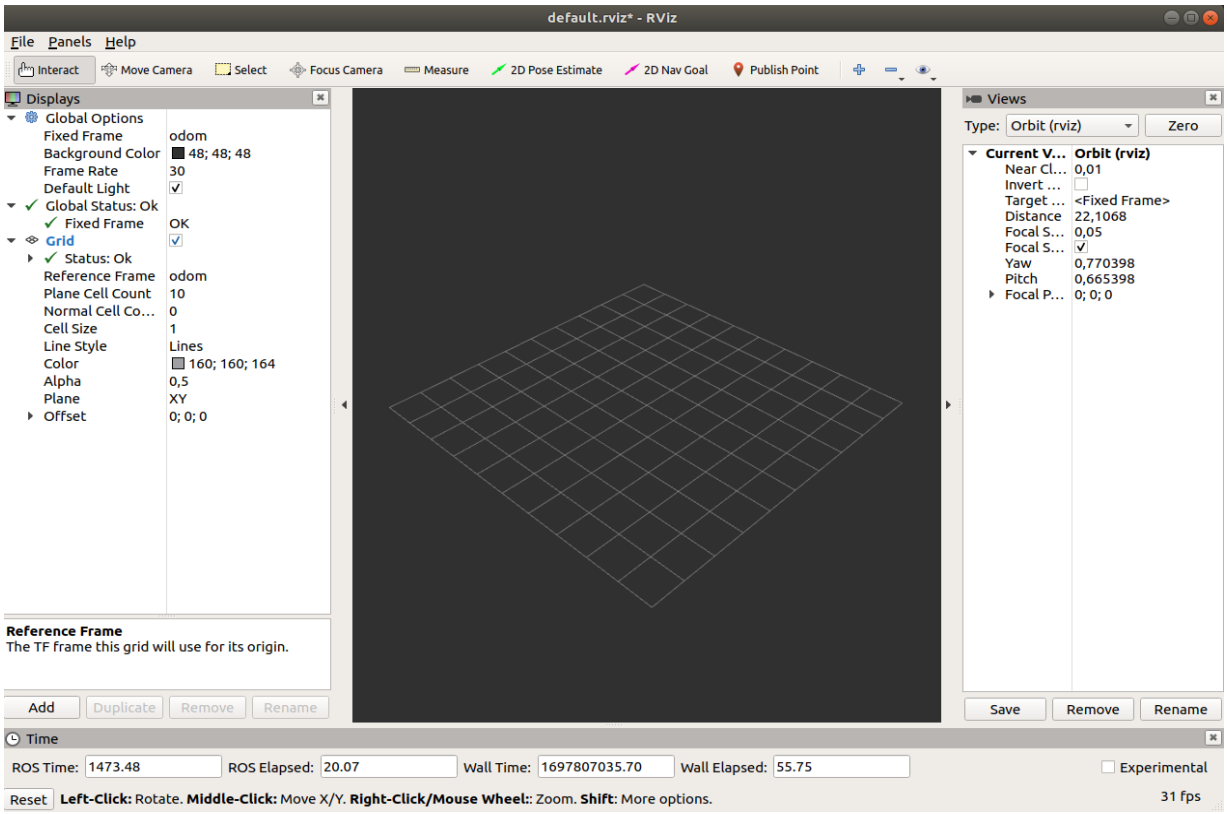

Figura 20 - Aspecto de Rviz

ROS Rviz, abreviatura de “ROS Visualization”, es una potente herramienta de visualización 3D interactiva diseñada específicamente para aplicaciones robóticas que te permite comprender el entorno del robot. Tiene cinco principales características que son:

1. **Interactividad**: Los usuarios pueden interactuar con los modelos 3D, ajustar la visualización y explorar el entorno robótico desde diferentes ángulos y perspectivas.
2. **Personalización**: Rviz es altamente personalizable, lo que permite a los desarrolladores adaptar la visualización según sus necesidades específicas, mostrando únicamente los datos relevantes para la tarea en cuestión.
3. **Integración con Sensores**: Puede mostrar datos de una amplia gama de sensores, como cámaras, lidar, IMU (Unidad de Medición Inercial) y muchos más, proporcionando una representación visual en tiempo real de los datos capturados.
4. **Herramienta de Depuración**: Rviz es una herramienta invaluable para depurar algoritmos de percepción, planificación de movimiento y otros componentes robóticos, permitiendo a los ingenieros visualizar y comprender el comportamiento del sistema en tiempo real.
5. **Simulación y Evaluación**: Además de la visualización en tiempo real, Rviz se utiliza en simulaciones robóticas para evaluar el rendimiento del algoritmo y refinar el diseño del robot antes de implementarlo en el mundo real.

Particularmente en esta primera fase inicial se ha utilizado un plugin the rviz llamado ***Teleop***, que como su nombre indica sirve para realizar la teleoperación del robot. Una vez instalado dicho plugin, lo ejecutamos abriendo un nuevo Panel y posteriormente seleccionando el plugin de Teleop. Se verá algo como lo que se muestra en la imagen siguiente.

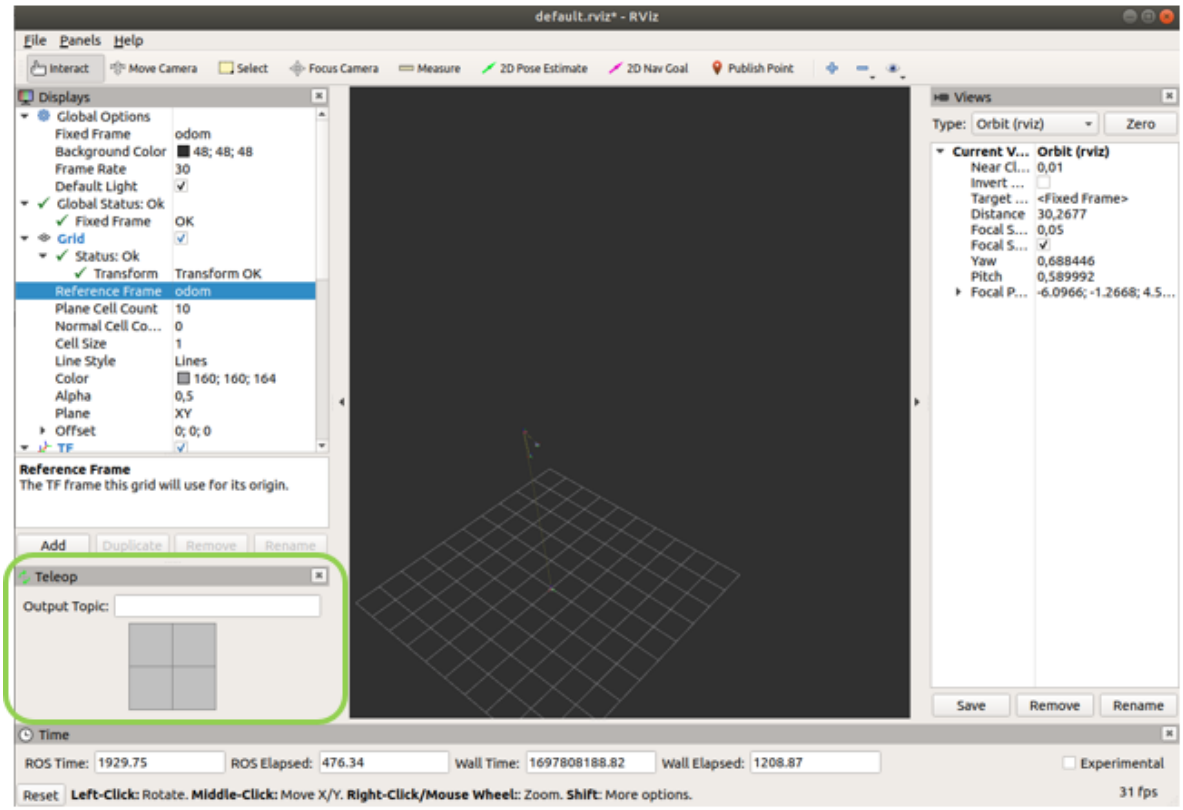

Figura 21 - Rviz con el plugin Teleop

Ahora tan solo faltará añadir el Topic en el que vamos a publicar las posiciones, en nuestro caso /cmd_vel. Una vez hecho eso se habilitará el cuadrado cambiando su color de gris a blanco. Al clicar con el ratón sobre el cuadrado, aparecerán unas líneas verdes que marcan la dirección y velocidad a la que se quiere llevar el robot. Estas líneas verdes se pueden apreciar en la siguiente imagen.

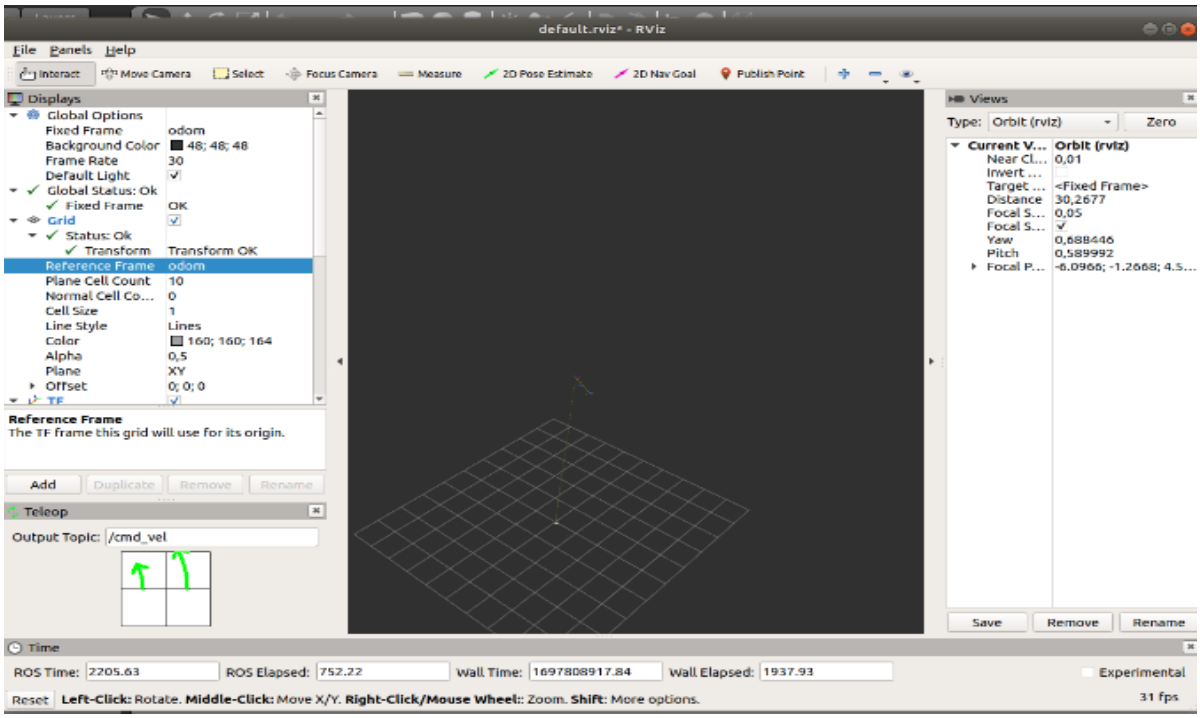

Figura 22 - Rviz con el plugin Teleop enviando una velocidad al robot

Esta es otra de las muchas posibilidades que existen para probar el robot. Al igual que se hizo anteriormente por terminal, mediante este plugin se pueden mandar velocidades al robot y observar su comportamiento. Eso sí, a diferencia de publicar la velocidad en el Topic mediante la terminal, a través de esta herramienta no tenemos control de la velocidad y dirección exacta que se envía. Sin embargo, para el tipo de pruebas que se querían hacer esta forma es más que suficiente.

4.3 Algoritmo de Navegación: Pure Pursuit

El algoritmo Pure Pursuit, o Persecución Pura en español, es un algoritmo de seguimiento muy popular en robótica móvil por su simplicidad conceptual y su capacidad para guiar al robot a lo largo de una trayectoria predefinida. Este algoritmo recibe la posición actual del robot y el conjunto de waypoints, y calcula la velocidad angular (w) del robot a partir de la velocidad lineal (v) en ese instante. La velocidad lineal normalmente viene fijada por un sistema de nivel superior, aunque también puede establecerse dentro del sistema de navegación.

Existe otro parámetro conocido como Look-ahead (L_d), o distancia de anticipación, que determina qué tan lejos en la trayectoria se busca el punto objetivo (TP). Evidentemente, el valor de este parámetro tiene implicaciones en el desempeño del algoritmo que se verán más adelante.

Sin entrar en mucho detalle, el algoritmo implementado consiste en calcular el punto objetivo (TP) en función de la posición y orientación actual del robot. Para ello se traza una circunferencia de radio L_d con centro en la posición actual. Allá donde corte la circunferencia con la trayectoria a seguir será el nuevo punto objetivo (TP). A continuación, el algoritmo busca alcanzar el punto calculado (TP) mediante un arco de circunferencia, y en base a eso se obtiene el radio de giro que se requiere trazar (R en la **Figura 23**). Este proceso se repite de manera continuada mientras el robot se mueve permitiendo al robot ajustar su trayectoria a medida que avanza. En la siguiente figura se representa gráficamente la explicación anterior.

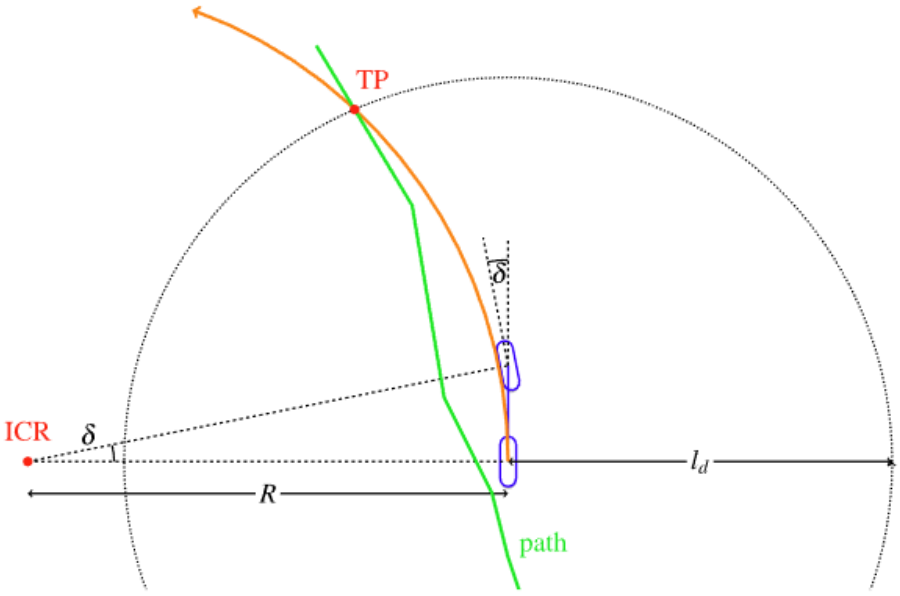

Figura 23 - Representación gráfica del cálculo del punto objetivo (TP)

Como se mencionó anteriormente, la elección de la velocidad (v) y Look-Ahead (L_d) repercuten en la navegación del robot. Ambos parámetros son directamente proporcionales y se relacionan según $L_d = \mathrm{K_{pp}} * \mathrm{v}$, siendo $\mathrm{K_{pp}}$ la ganancia del sistema. El impacto de modificar el Look-Ahead y la velocidad es el siguiente:

- **Look-Ahead**: Si se toma un valor pequeño, puede mejorar el seguimiento de la trayectoria del robot ya que el robot tomará decisiones más rápidas y reaccionará más pronto a las curvas. Además, el robot alcanzará los waypoints. Sin embargo, si no está bien afinado y es demasiado bajo puede provocar inestabilidad en el sistema, lo que se traduce en oscilaciones.

  Con valores más altos se obtiene lo contrario, el seguimiento del camino es más suave pero el tiempo de respuesta es mayor ante cambios repentinos en la trayectoria. Un problema típico en valores altos de Look-Ahead es el de "las esquinas cortadas". Esto es un fenómeno en el cuál el robot se mueve dentro de la esquina en vez de alrededor de ella.

  En la siguiente figura pueden verse las limitaciones del Pure Pursuit explicadas anteriormente en relación al Look-Ahead. La imagen superior izquierda son las oscilaciones provocadas a causa de un Look-Ahead demasiado pequeño. La imagen inferior izquierda es el comportamiento suave que experimenta el robot ante valores altos de Look-Ahead. Por último, la imagen de la derecha ejemplifica el problema de "las esquinas cortadas".

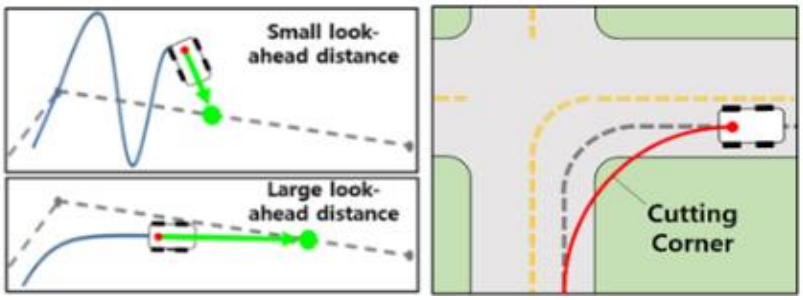

Figura 24 - Limitaciones del Pure Pursuit en relación a distintos valores de Look-Ahead

La siguiente imagen extraída de una publicación en una revista de ingeniería eléctrica y computacional (Ref. [9]) muestra los efectos de diferentes valores de Look-Ahead para el seguimiento de una trayectoria. También aquí pueden verse los fenómenos explicados anteriormente, es decir, oscilaciones para Look-Ahead bajos y el problema de la esquina cortada para valores altos.

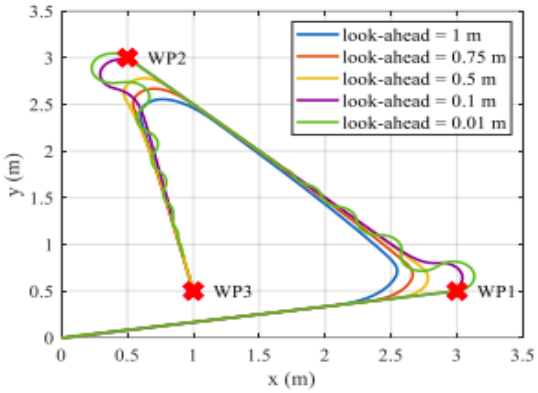

Figura 25 - Efectos de diferentes valores de Look-Ahead en el seguimiento de trayectorias.

- **Velocidad**: Si la velocidad del robot es alta puede tener efectos negativos, como por ejemplo que su inercia será mayor. Esto significa que necesita más tiempo y espacio para detenerse o cambiar de dirección. Otro inconveniente es que tiene un menor tiempo de reacción, lo que puede provocar los mismos efectos que antes; falta de tiempo para cambiar la dirección. En conclusión, velocidades altas pueden provocar errores de seguimiento, especialmente en curvas pronunciadas o situaciones de obstáculos inesperados.

  También tiene efectos positivos; por ejemplo, tiene una mayor eficiencia en tiempo pudiendo recorrer distancias más largas en menos tiempo, lo que es esencial para algunas aplicaciones en las que prima la velocidad. En situaciones en los que el recorrido es sencillo, sin obstáculos y principalmente recto, puede tener sentido usar velocidades más altas.

  Cuando se usan valores bajos de velocidad la precisión aumenta, sin embargo se puede pecar de demasiada cautela y perder eficiencia.

En definitiva, ajustar correctamente la velocidad y el Look-Ahead es esencial para un buen comportamiento del robot. Ya sean velocidades o Look-ahead altos o bajos se pueden encontrar ventajas y desventajas, por eso es importante ajustar ambos valores en conjunto, pués hay una relación entre sí. A continuación, se explica el comportamiento esperado para cada una de las posibilidades. Aunque no se mencione, en todos los casos siguen presentes las ventajas e inconvenientes arriba descritos.

- Look-Ahead alto y velocidad alta: A causa de la velocidad alta el tiempo de reacción es menor, sin embargo se compensa con el valor alto de Look-Ahead que permite planificar su trayectoria asegurando un comportamiento suave, a costa de una menor precisión especialmente en curvas.
- Look-Ahead alto y velocidad baja: Esta configuración permite una navegación precisa y estable, consiguiendo movimientos suaves. Sin embargo, este es el caso extremo de prudencia lo que significa que se puede pecar de exceso de prudencia.
- Look-Ahead bajo y velocidad alta: Si un Look-Ahead bajo puede conllevar problemas de inestabilidad, más aún a velocidades altas que el tiempo para corregir errores es menor. En particular para la aplicación aquí estudiada no es una buena configuración, ya que no se beneficia de ninguna de las ventajas que tienen valores altos de velocidad y bajo de Look-Ahead.
- Look-Ahead bajo y velocidad baja: A causa de un Look-Ahead bajo se pueden producir inestabilidades a causa de una navegación reactiva, sin embargo se compensa de alguna forma disminuyendo la velocidad. Esta configuración es buena para la navegación en espacios reducidos donde hacen falta reacciones rápidas y ajustes precisos. Este no es el caso de la aplicación aquí estudiada.

En conclusión, es importante encontrar un equilibrio entre ambos parámetros. Parece razonable que para velocidades altas se debe optar por un Look-Ahead moderadamente alto para proporcionar una anticipación adecuada sin sacrificar demasiado tiempo de reacción y reducir las posibilidades de inestabilidades. Si por el contrario se escoge una velocidad baja, la mejor opción parece ser un valor de Look-Ahead moderadamente bajo, especialmente en entornos confinados.

4.3.1 Implementación

Anteriormente se ha explicado de forma genérica en qué consiste a grandes rasgos el algoritmo Pure Pursuit, sin embargo se pueden encontrar varias versiones algo diferentes unas de otras. En el caso de este proyecto se ha optado por la solución propuesta en el repositorio **ROS_Pure_Pursuit** del usuario *leofansq* (Ref. [7]), ya que es compatible con la versión de ROS Melodic que se usa en este proyecto.

Dentro de este repositorio se encuentran principalmente dos archivos importantes, **pure_pursuit.cpp**, que es la implementación del algoritmo en c++, y **pure_pursuit_node.launch**, que es el fichero tipo *launch* que permite lanzar el nodo y parametrizar el código sin necesidad de modificar el código del algoritmo. Aún así el código de c++ ha requerido cierta modificación para añadir los *markers* de la posición del robot en cada instante. Estos *markes* pueden representarse en herramientas como rviz para saber la trayectoria que ha seguido realmente el robot. En cualquier caso, como se explicará en la siguiente sección 4.3.2, al final no se usaron los markers aquí programados sino que se representó directamente la odometría del robot usando rviz.

Además, el código **pure_pursuit.cpp** ha requerido otra modificación muy importante debido a que no era correcto del todo. En la parte del código en la que se calcula la velocidad angular faltaba considerar los valores negativos para así limitar la velocidad de giro en las curvas con sentido horario. Tal y como estaba el código antes de la modificación (vea la **Figura 26**) se tomaba el menor valor de dos posibles velocidades angulares. En el caso en el que el giro sea horario, la velocidad angular es negativa y siempre será menor que w_max_ que siempre es positiva, provocando que se cojan valores mayores que w_max_ en valor absoluto. La consecuencia de esto era enviar al robot velocidades angulares negativas muy altas haciendo que el robot se descontrole y gire sobre sí mismo. Para solucionarlo se añade la condición de no sobrepasar una velocidad angular mayor que -w_max_. El código resultante puede verse en la **Figura 27**.

```
cmd_vel_.angular.z = std::min(2*v_/lookahead_distance_*lookahead_distance_*lateral_offset, w_max_);
```

Figura 26 - Código antes de la corrección.

```
cmd_vel_.angular.z = std::min(2*v_/lookahead_distance_*lookahead_distance_*lateral_offset, w_max_);

if (cmd_vel_.angular.z < -1)
{
  cmd_vel_.angular.z = -w_max_;     // Condición para que tampoco supere w_max negativa (giro horario)
}
```

Figura 27 - Código después de la corrección.

Respecto al archivo **pure_pursuit_node.launch**, se ha tenido que parametrizar de acuerdo a los *frames* definidos en este proyecto, como son *"base_link"* para el frame del robot, y *"odom"* para el frame del Look-Ahead. Además de eso, se ha utilizado para probar diferentes valores de velocidad lineal y angular máxima, del Look-Ahead y de tolerancias.

Como se ha mencionado anteriormente, una de las entradas que necesita este algoritmo son los waypoints que definen el camino a seguir por el robot. Todos los waypoints son conocidos de antemano y se definen en el archivo **GazeboCropRowGenerator.py**, el cuál entre otras cosas da como resultado un fichero de texto llamado **waypoints.txt** donde se guardan todos los waypoints. Para la plantación de brócolis propuesta se obtienen un total de 188 waypoints, ya que cada uno de los brócolis se define como un waypoint. Además de los brócolis, también se guardan como waypoints puntos de seguridad situados a 5 metros de distancia antes de la primera planta de la hilera y otro punto a la misma distancia medido a partir de la última planta de la hilera. De este modo se consigue un margen de seguridad en ambos extremos de cada una de las hileras para dificultar que el robot pise los brócolis en caso de no seguir la trayectoria correctamente. Asimismo, se programa la función ***waypoints2semicircle***, que a partir de la posición del último brócoli de una hilera y la posición del primer brócoli de la siguiente hilera define un arco de circunferencia con 10 waypoints. De esta forma se establece la trayectoria que debe seguir el robot para pasar de una fila de brócolis a la siguiente.

Una vez se tiene el camino a seguir definido mediante los waypoints en el fichero de texto, hay que

buscar la forma de enviárselos al Pure Pursuit. Para eso se recurre al repositorio **ROS_Waypoints_Processor** creado de nuevo por el usuario *leofansq* (Ref. [7]). De este repositorio el único archivo relevante es **load_waypoints.py** que ha sido renombrado para este proyecto con el nombre de **load_waypoints_purePursuit.py**. En este nuevo archivo se han hecho las modificaciones oportunas para leer el fichero de texto donde están guardados los waypoints guardados, así como publicar los waypoints en el frame correcto; en este caso */odom*.

4.3.2 Resultados

Hasta aquí se ha explicado lo que cabe esperar en general según diferentes valores de Look-Ahead y velocidad. Ahora se pasa a ver como se comporta el Pure Pursuit para esta aplicación en concreto considerando diferentes valores de Look-Ahead y velocidad. Se mostrarán gráficas que representan el camino a seguir frente al camino realmente seguido por el robot. Además las imágenes se complementarán con una explicación del comportamiento del vehículo que se ha observado durante la simulación, que en algunas ocasiones no es apreciable únicamente en las gráficas.

Para encontrar los mejores valores de Look-Ahead y velocidad se ha hecho de forma experimental observando el comportamiento del robot y modificando los parámetros en consecuencia. Para poder juzgar el comportamiento del vehículo de una forma más objetiva se ha representado la odometría frente a la trayectoria. Para estas representaciones se ha usado **rviz**, con la configuración que puede verse en la **Figura 28**.

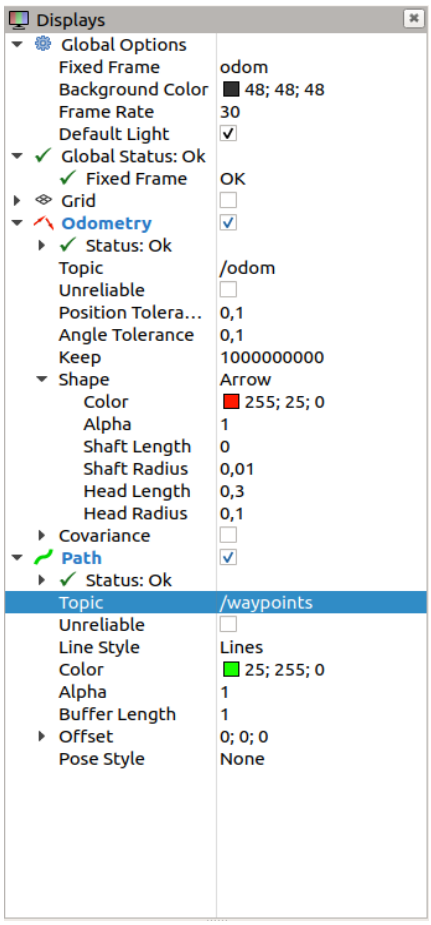

Figura 28 - Configuración de rviz para mostrar la odometría y los waypoints

Con una línea roja se representa la odometría del robot y en verde el conjunto de todos los waypoints que forman la trayectoria que debe seguir el robot. Cabe destacar que la trayectoria a seguir no muestra de manera clara la dimensión de las hileras de brócolis haciendo parecer que son más largas de lo que en realidad son, ya que también incluye las posiciones de seguridad a 5 metros de cada extremo de cada hilera. De forma general en todas las imágenes se verá como, especialmente al principio, el robot pasa por encima de la línea verde pareciendo que está pisando los brócolis o saltándose algunos de ellos, cuando en realidad no tiene por qué ser así porque aún no hay brócolis en esa zona. Lo que se pretende mostrar con las imágenes siguientes es cómo afectan los valores de Look-Ahead y velocidad a la navegación, “obviando” si se pisan o se saltan los brócolis. Se explicará en cada caso particular si el robot ha pisado o no la plantación de brócolis o si se ha saltado alguno de ellos. En todos los casos aquí estudiados el robot es capaz de completar el recorrido, sin embargo se analizará de qué manera lo hace.

A continuación se muestran los resultados para diferentes valores de Look-Ahead (L) a dos velocidades diferentes, manteniendo la velocidad angular constante en todos los casos y con valor 1 rad/s. Se recuerda que la línea roja representa la odometría del robot y la verde el conjunto de todos los waypoints que forman la trayectoria que debe seguir el robot. La hilera por la que el robot empieza recorriendo la plantación es la que se ve en las imágenes la primera comenzando por la izquierda.

Velocidad lineal = 1 m/s

- Look-Ahead = 4 m

  Según se vio antes de forma teórica, este parece ser un caso de Look-Ahead demasiado alto. Se aprecia el fenómeno de las esquinas cortadas, pues no llega el robot a hacer el giro en el lugar que debería hacerlo. También se puede observar cómo se aproxima a cada hilera después de girar de manera suave y no pasa por encima de todos los waypoints, en particular por los puntos de seguridad definidos. Todo esto ha provocado que el robot pise los primeros brócolis de cada hilera.

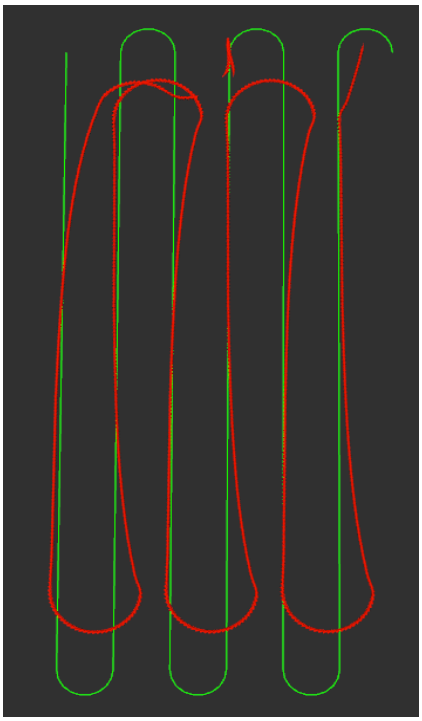

Figura 29 - Comportamiento del robot para v = 1 m/s y L = 4 m

- Look- Ahead = 2.5 m

  Este es el mejor valor de Look-Ahead que se ha encontrado para una velocidad de 1 m/s. Puede apreciarse cómo el robot se ha ajustado mucho mejor a la trayectoria sin perder suavidad en la navegación. Sigue existiendo el fenómeno de “la esquina cortada” pero en menor medida. De todos modos, para esta aplicación no es necesario que el robot siga la trayectoria durante el giro a la perfección, porque debido al margen de seguridad que se ha fijado el robot tiene espacio de sobra para maniobrar en el cambio de hilera consiguiendo no pisar ninguno de los brócolis. Puede observarse también que cuando el robot se incorpora a la primera hilera lo hace con mucha más antelación, al contrario que en el caso de L=4m. Por último, se aprecia que incluso cuando los parámetros están mejor ajustados, el robot no responde bien a curvas tan cerradas. En este caso, tiende a realizar la curva de forma más amplia y abierta.

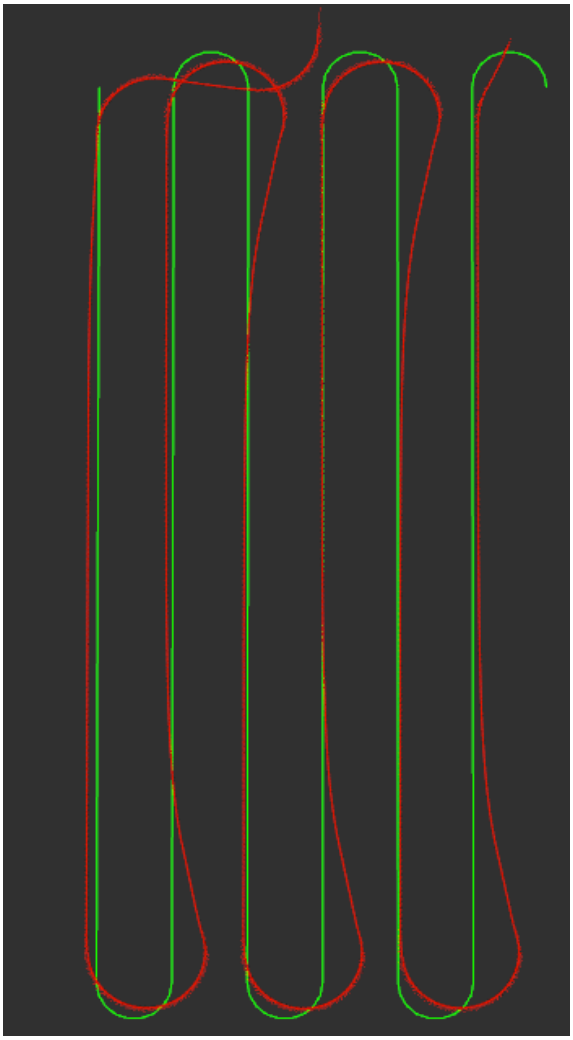

Figura 30- Comportamiento del robot para v = 1 m/s y L = 2.5 m

- Look-Ahead = 1.5 m

  Con valores de Look-Ahead más pequeños se producen oscilaciones, como la que se puede ver en la primera fila de brócolis. Además, el robot intenta alcanzar todos los waypoints y ya no se produce el efecto de "la esquina cortada", al contrario, en el giro el robot sobrepasa el camino a seguir. Por el hecho de querer alcanzar todos los waypoints, el robot realiza al principio una trayectoria poco óptima para conseguir aproximarse al primer punto de seguridad y al primer brócoli (que es el segundo waypoint definido). Esta forma de avanzar hacia la primera hilera provoca que el autómata pase por encima de los primeros brócolis y luego oscile antes de estabilizarse en la fila.

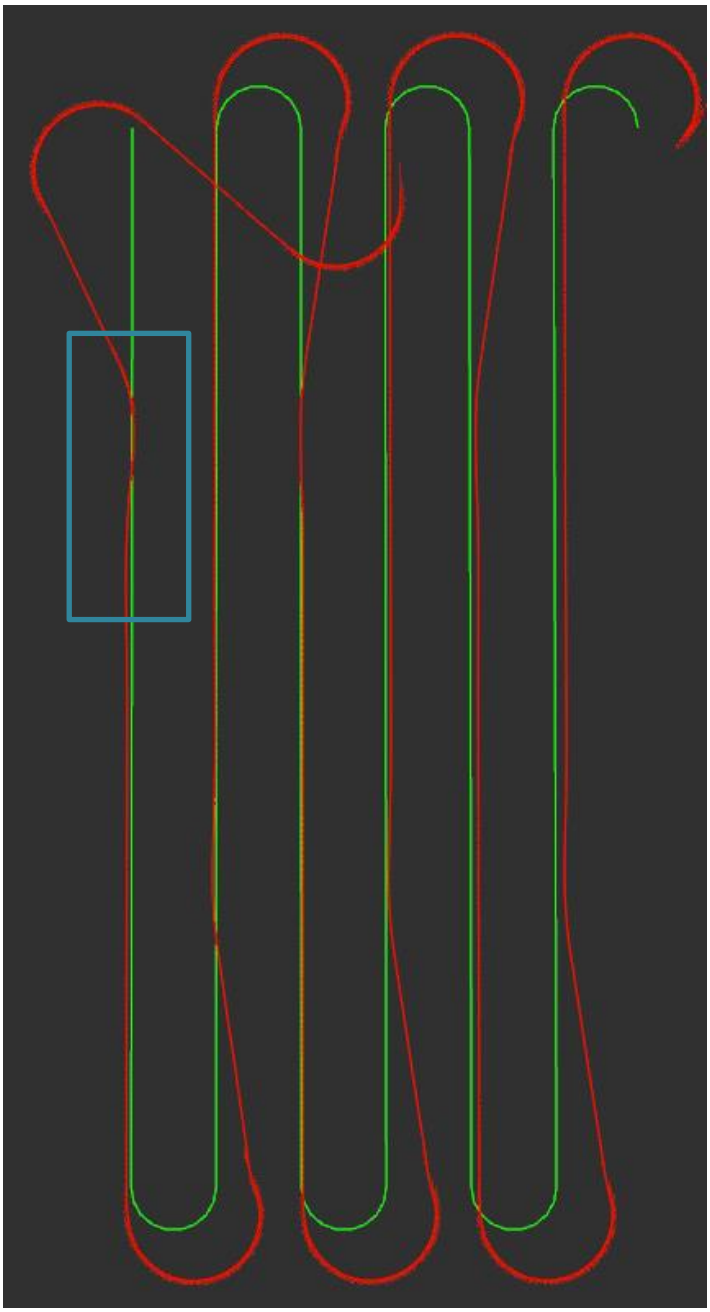

Figura 31 - Comportamiento del robot para v = 1 m/s y L = 1.5 m

- Look-Ahead = 1.3 m

  Este caso es similar al anterior, pero llevado al extremo. Las oscilaciones son aún más pronunciadas, llegando al punto en el que el robot pierde por completo el control y comienza a dar vueltas sobre sí mismo al entrar en la primera curva.

Figura 32 - Comportamiento del robot para v = 1 m/s y L = 1.3 m

Velocidad lineal = 0.75 m/s

Al igual que antes, se analiza el comportamiento del robot con diferentes valores de Look-Ahead, pero esta vez a una velocidad menor.

- Look-Ahead = 3 m

  Para valores altos de Look-Ahead se siguen observando cambios de dirección suaves, necesitando cierto tiempo para incorporarse a la primera hilera. También se aprecia el fenómeno de “las esquinas cortadas”.

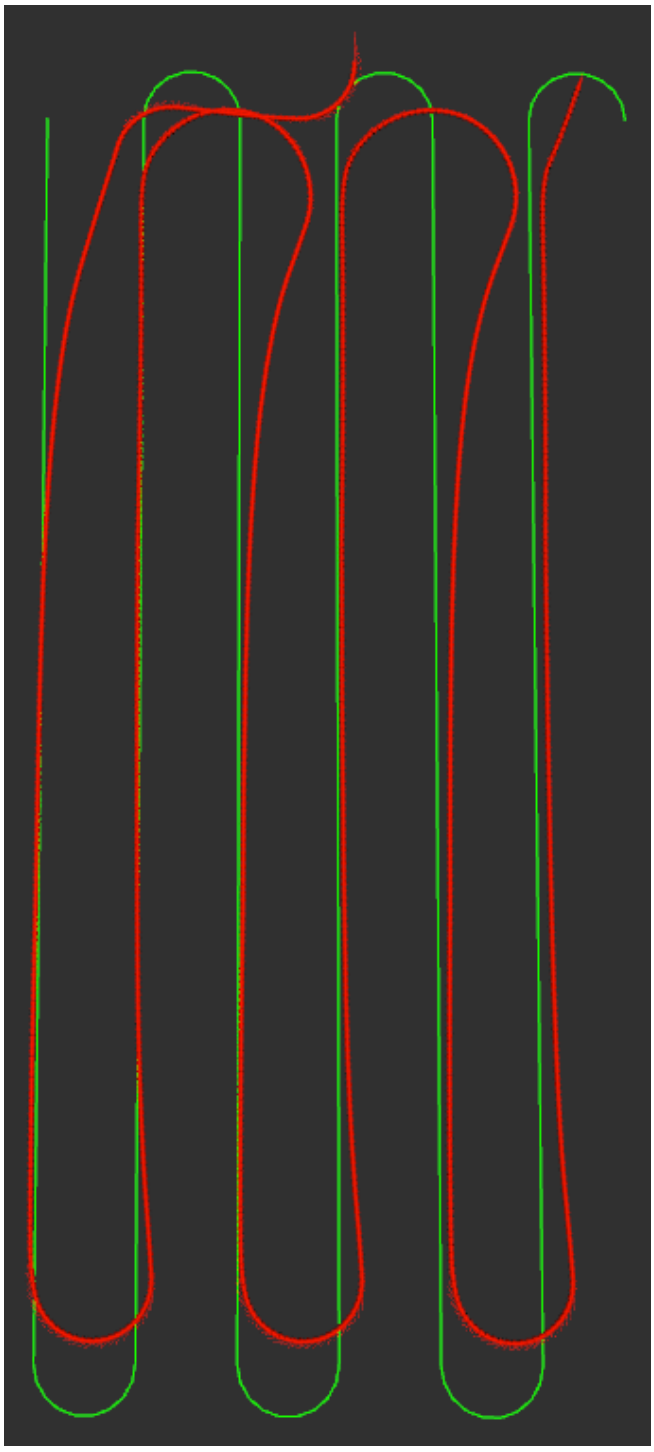

Figura 33 - Comportamiento del robot para v = 0.75 m/s y L = 3 m

- Look-Ahead = 1.8 m

  Este es el mejor valor de Look-Ahead que se ha encontrado para una velocidad de 0.75 m/s. En comparación con el mejor caso analizado a una velocidad de 1 m/s, cuando el robot va más lento tiene mucha más precisión, especialmente en los giros que los hace mucho más controlados. En cualquier caso, parece que las curvas son demasiado cerradas para seguirlas con total exactitud.

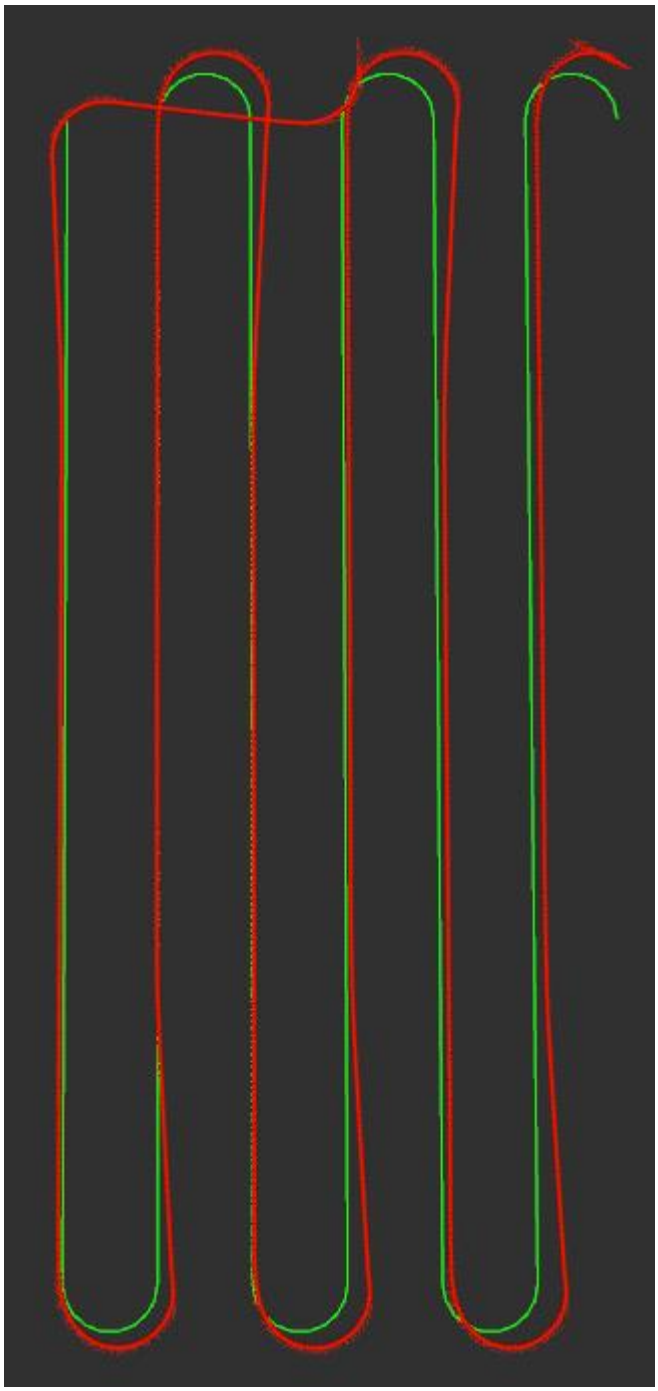

Figura 34 - Comportamiento del robot para v = 0.75 m/s y L = 1.8 m

- Look-Ahead = 1.36 m

  Para valores pequeños de Look-Ahead empiezan a aparecer oscilaciones, como la que se puede observar en la primera hilera. En comparación con el escenario equivalente en el que el robot se desplaza a una velocidad de 1 m/s, la oscilación generada a una velocidad de 0.75 m/s es menor. Esto es algo esperado, ya que al tener menor velocidad el robot tiene más control. Como se evidenciará en el próximo caso analizado, no es viable reducir significativamente el valor de Look-Ahead para provocar un aumento de las oscilaciones, ya que, al disminuir tan solo 16 cm, el robot pierde por completo el control.

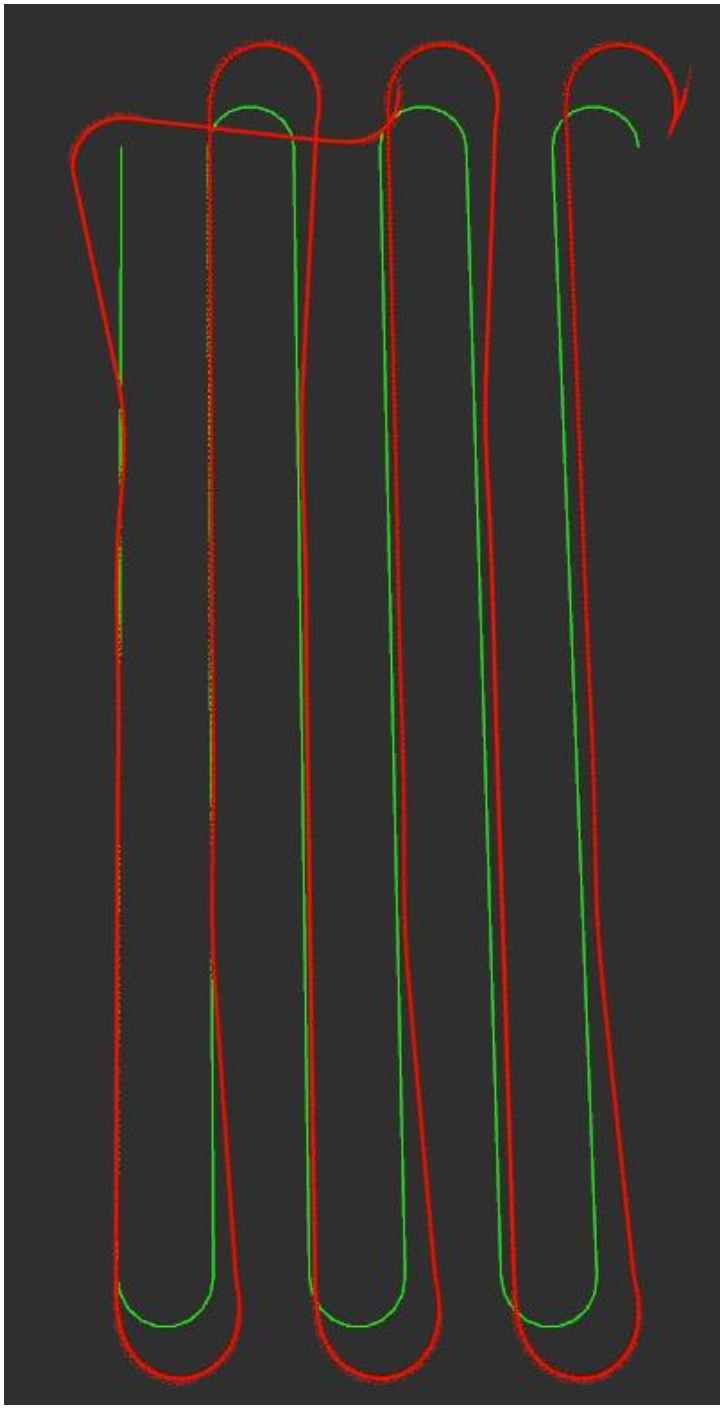

Figura 35 - Comportamiento del robot para v = 0.75 m/s y L = 1.36 m

- Look-Ahead = 1.2

  A medida que se reduce el valor de Look-Ahead, la navegación se vuelve más reactiva, lo que resulta en oscilaciones más acentuadas. Este proceso llega a un punto crítico en el que el robot pierde por completo el control y empieza a dar vueltas sobre sí mismo al enfrentar la primera curva.

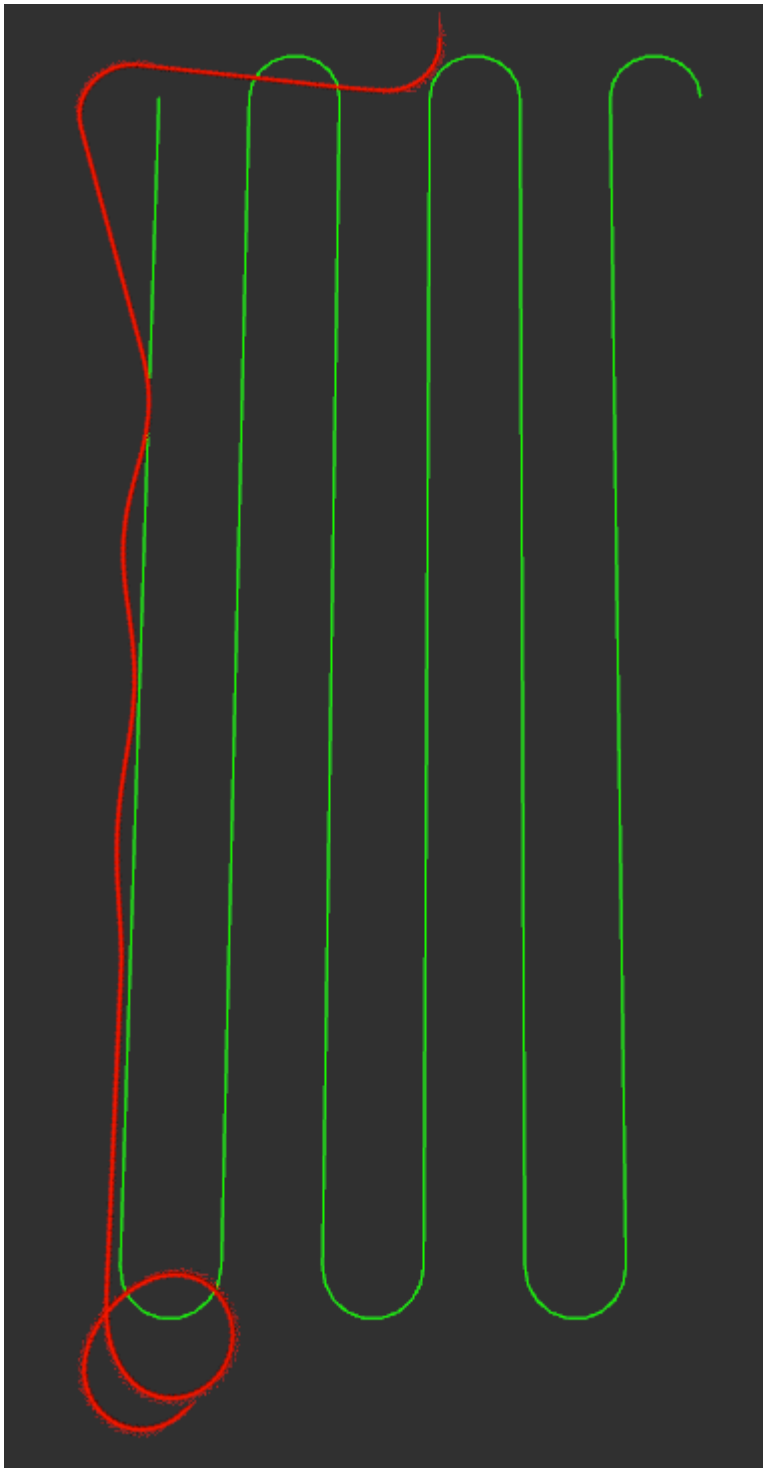

Figura 36 - Comportamiento del robot para v = 0.75 m/s y L = 1.2 m

Mediante todos estos experimentos en los que se ha analizado el comportamiento del robot para diferentes valores de Look-Ahead a dos velocidades diferentes, se ha evidenciado el comportamiento que se cabía esperar y que fue explicado en la sección 4.3 . Independientemente de la velocidad, a valores altos de Look-Ahead la reacción del robot es más lenta y se produce el efecto de la esquina cortada. A valores más bajos, el comportamiento es más reactivo pudiendo provocar oscilaciones, y en el caso extremo la pérdida de control del robot. En comparación entre el comportamiento del robot a 1 m/s y 0.75 m/s, se puede afirmar que a menor velocidad, el autómata exhibe una mayor precisión, especialmente en las curvas. Además, se ha visto cómo para 1 m/s el mejor comportamiento se consigue con un Look-Ahead de 2.5 m, mientras que para 0.75 m/s se logra con un valor de 1.8 m. Esto demuestra que existe una relación entre el valor del Look-Ahead y la velocidad del robot: es necesario aumentar el valor del Look-Ahead al incrementar la velocidad y, por el contrario, reducirlo al disminuir la velocidad para mantener un control adecuado del sistema.

Aunque no se hayan mostrado aquí, también se realizaron pruebas a 1.5 m/s no consiguiendo buenos resultados. Por tanto, parece claro que este robot se comporta mejor a velocidades bajas. Debido a la aplicación para la que está ideado este autómata, no se requieren altas velocidades pero sí más precisión. Por ende, en caso de usar este algoritmo, se recomienda una velocidad de 0.75 m/s y un Look-Ahead de 1.8 metros para conseguir un buen compromiso entre eficiencia y precisión.

4.4 Move Base Flex

En este capítulo se presenta una alternativa al Pure Pursuit, Move Base Flex (MBF). De acuerdo con la publicación “Move Base Flex, a Highly Flexible Navigation Framework for Mobile Robots“ (Ref. [10]), MBF es un sistema de navegación altamente flexible, modular, independiente de mapas y de código abierto para ROS. MBF proporciona acciones modulares para ejecutar plugins de planificación de trayectorias, control de movimientos y modos de recuperación. Estas acciones definen interfaces para ejecutivos externos para permitir estrategias de navegación altamente flexibles, que pueden entrelazarse con otras tareas del robot. MBF es un reemplazo compatible con versiones anteriores de Move Base, ya que puede usar complementos existentes para Move Base y proporciona una versión mejorada de las interfaces ROS del planificador, el controlador y el plugin de recuperación. MBF permite más flexibilidad, personalización, modularidad, es capaz de dar información detallada del estado actual y además es más eficiente que Move Base.

En definitiva, MBF es un paquete que facilita la configuración, ejecución e interacción entre el planificador local y global, los modos de recuperación y el control de movimientos. Estos son elementos fundamentales para la navegación flexible e inteligente de robots, y especialmente para evitar obstáculos. La planificación se divide entre un escenario local y otro global, utilizando un mapa de costes global para representar el mapa estático y un mapa de costes local para representar el entorno local, almacenando datos dinámicos de obstáculos detectados por los sensores. Un planificador global vinculado a un mapa de costes global se utiliza como guía de alto nivel para la planificación de rutas desde una pose inicial (normalmente la pose del robot) hasta la posición objetivo. El mapa de costes global suele representar el entorno ya mapeado. Además, los comportamientos de recuperación deben hacer que el sistema sea robusto teniendo en cuenta el mapa de costes local y global.

La siguiente figura describe la arquitectura de MBF. MBF está dividido en diferentes niveles de abstracción e implementación. El Nivel de Abstracción del Move Base Flex se representa mediante el paquete "mbf abstract nav", que proporciona una abstracción general del sistema. Luego, está el Nivel de Implementación de Mapa de Costes, que maneja la creación y gestión de los mapas de costes globales y locales. La implementación del nivel de mapas de costes se encuentra en el paquete "mbf costmap nav", encargado de gestionar los mapas de costes y de inicializar los plugins extendidos.

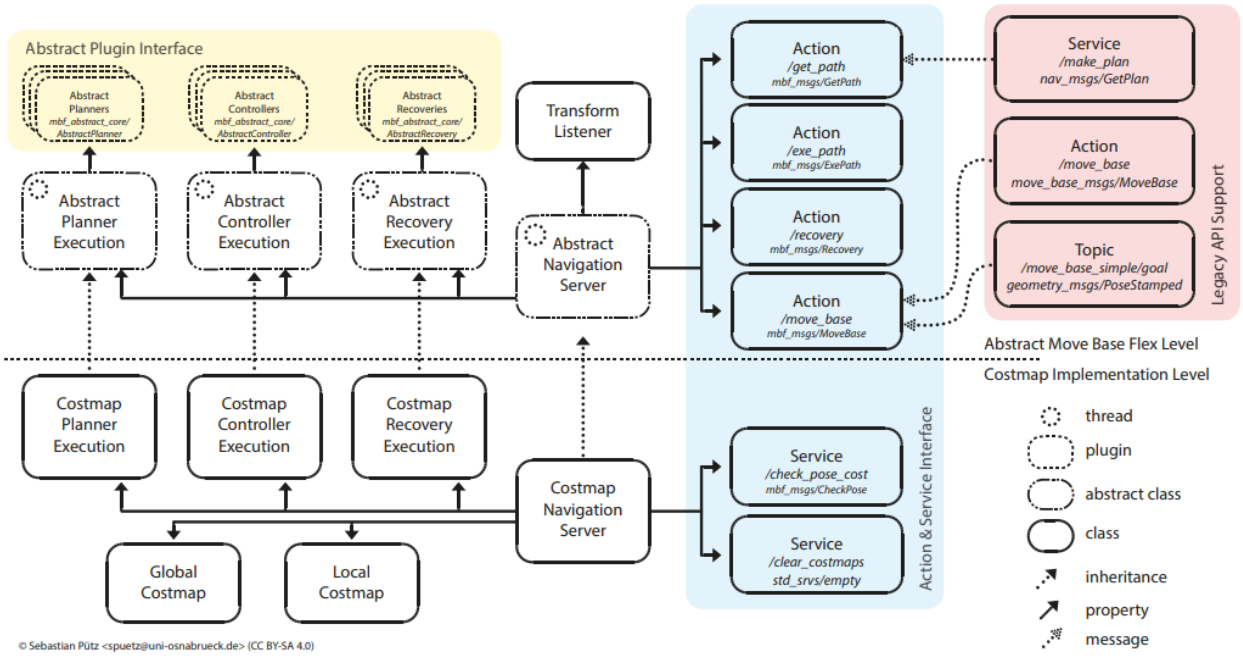

Figura 37 - Arquitectura de Move Base Flex

4.4.1 Implementación

En primer lugar, como se ha visto antes, este paquete de navegación permite evitar obstáculos, sin embargo no se ha probado esta funcionalidad en este proyecto. Para ello, lo primero que habría que hacer es integrar sensores al robot para detectar los obstáculos y alimentar los mapas de costes.

El paquete de MBF se ha descargado del repositorio de GitHub del usuario Magazino (Ref. [11]). A continuación, se especifican los archivos más relevantes del repositorio para la implementación. Una modificación general que se ha tenido que hacer en todos los archivos ha sido el uso de los frames adecuados.

- **controllers.yaml:** Se ha modificado para elegir el planificador local. En este caso se ha elegido el Teb_Local_Planer.
- **local_costmap_params.yaml:** Define el mapa de costes local. Se ha modificado el código para fijar el tamaño del mapa a 10x10 m.

```
local_costmap:
  global_frame: odom
  update_frequency: 3.0
  publish_frequency: 3.0
  static_map: False
  rolling_window: true
  width: 10
  height: 5
  plugins:
    - {name: static,       type: "costmap_2d::StaticLayer"}
```

Figura 38 - local_costmap_params.yaml modificado

- **global_costmap_params.yaml:** Define el mapa de costes global. Se ha modificado el código para fijar el tamaño del mapa a 50x50 m.

- **Teb_local_planner.yaml:** Es el controlador usado para la trayectoria. Este es el principal archivo que hay que modificar. Aquí están definidos todos los parámetros que considera el planificador local para hacer sus cálculos. Existen variables para la trayectoria, el robot (por ejemplo: velocidades y aceleraciones máximas o el mínimo radio de giro), tolerancias, obstáculos y para la optimización del controlador. Son muchos parámetros los que hay que ajustar hasta conseguir el comportamiento deseado y es la parte más tediosa del proceso de implementación del move base flex. El proceso de tuneado del controlador se ha hecho a prueba y error, aunque conociendo el significado de cada parámetro. La definición de cada parámetro, así como el paquete del planificar local, puede encontrarse en la ref. [12].

  Debido a que son muchos los parámetros que hay que configurar, se ha recurrido a una herramienta que permite modificarlos en tiempo real, llamada **rqt_reconfigure**. Es importante destacar que esta herramienta no guarda el valor que se le ha dado a cada parámetro, tendrá que ser el desarrollador el que lo haga manualmente una vez encontrados los valores que funcionan. Para lanzar esta herramienta hay que ejecutar el siguiente comando: **rosrun rqt_reconfigure rqt_reconfigure**. El aspecto puede verse en la **Figura 39**.

  Los parámetros que se han configurado han sido los de las pestañas: “Optimization”, “Robot”, “GoalTolerance” y “Trajectory”.

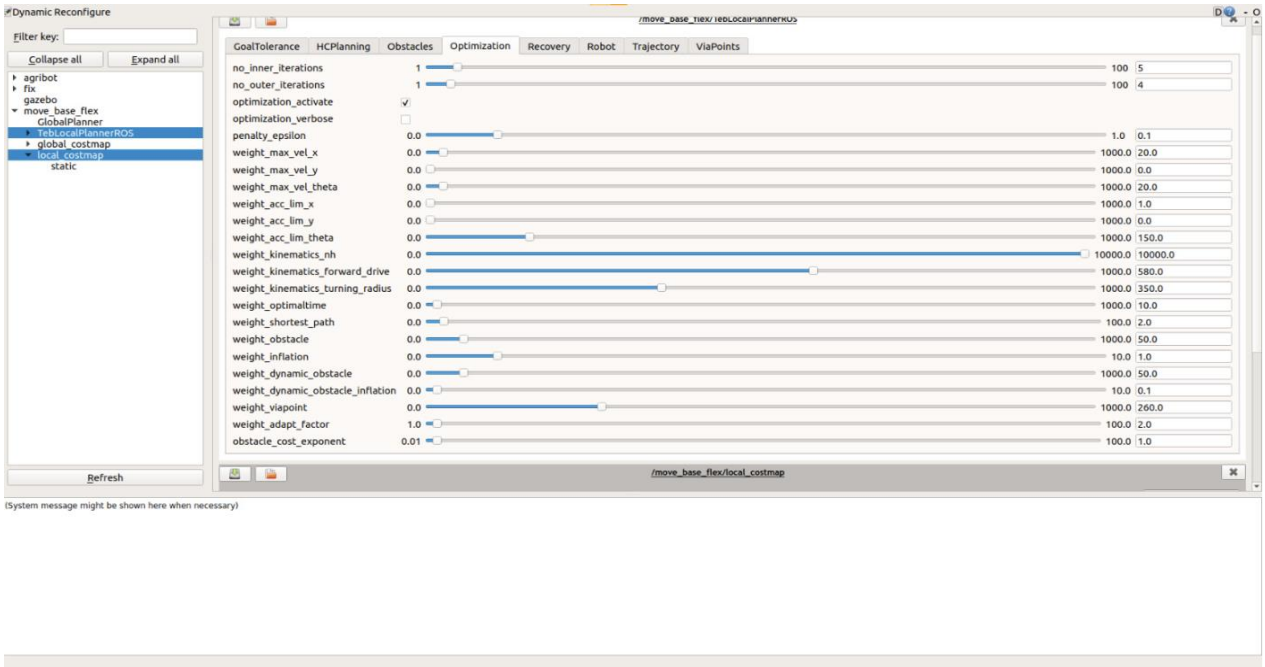

Figura 39 - Aspecto de la herramienta rqt_reconfigure

Para terminar de implementar MBF es necesario enviarle el camino que tiene que recorrer el robot. La procedencia de los waypoints es la misma que la explicada para el Pure Pursuit, todos están guardados en el fichero de texto *waypoints.txt,* la diferencia está en cómo se envían. Una particularidad de MBF es que hay que enviarle la ruta completa, y no punto a punto. La forma de enviarle el camino es por medio de *acciones* en vez de *topics*, por lo que es necesario crear un código nuevo para enviar la trayectoria, **load_waypoints_moveBase.py**. Este código es igual que el usado para el Pure Pursuit (**load_waypoints_purePursuit.py**), con la diferencia de que se usan acciones para enviar la ruta. La acción usada para enviar el camino es *exe_path*, que, dada una ruta y un controlador como entrada,

transfiere la ruta al controlador seleccionando. Pone en marcha el controlador y lo ejecuta con una frecuencia predefinida, mientras calcula los comandos de velocidad que se publican directamente para ordenar al robot que se mueva. Durante su ejecución devuelve la posición actual del robot, la velocidad y la distancia y el ángulo a la posición objetivo. Como resultado, devuelve la posición final y la distancia y el ángulo a la posición objetivo.

La visualización en rviz de MBF una vez implementado puede verse en la **Figura 40**, siendo la línea roja la representación de la odometría del robot, la línea verde el planificador global, la línea fina marrón el planificador local y el cuadrado gris el mapa de costes local.

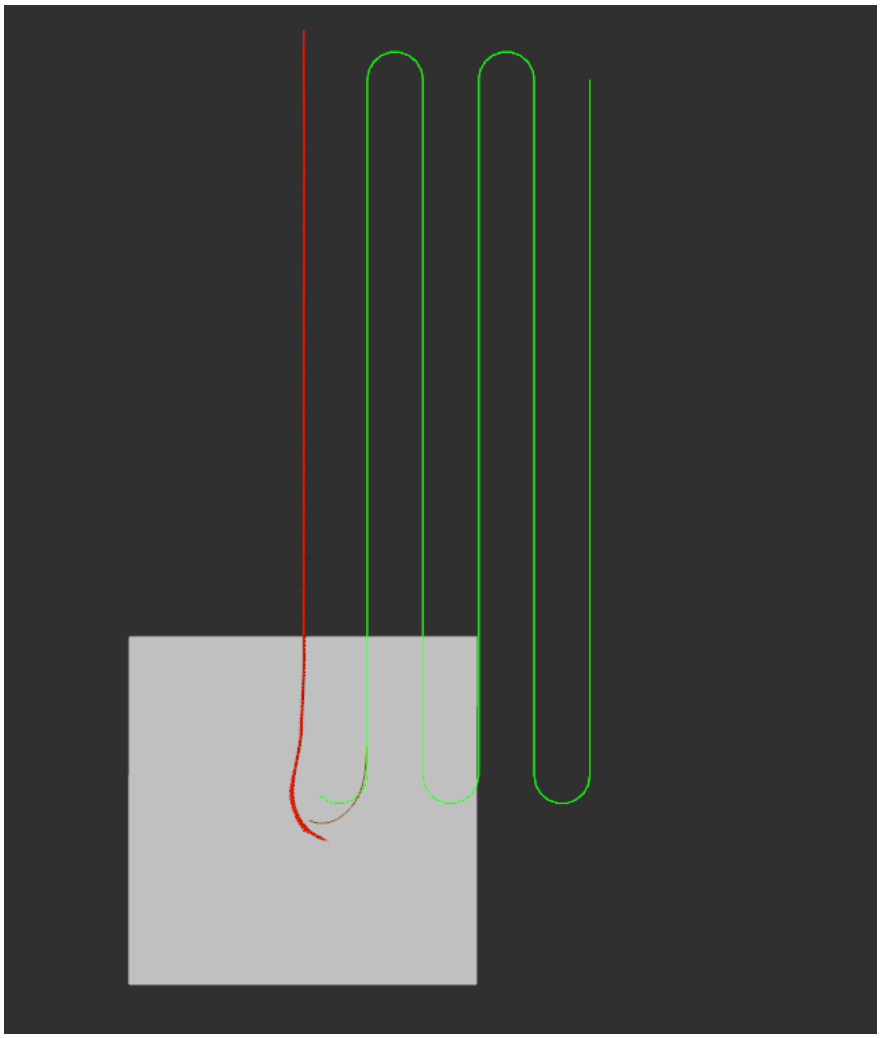

Figura 40 - MBF implementado

4.4.2 Resultados

A continuación, se pasan a explicar los problemas encontrados y las soluciones propuestas hasta conseguir el resultado final.

El primer problema encontrado fue que, aunque el robot recibía la ruta completa, no empezaba a recorrerla desde el principio. Se descubrió que algunos algoritmos de navegación que buscan la optimización de su ruta, como es el caso de Move Base Flex, comienzan a recorrer la trayectoria por el punto más cercano. Como para esta aplicación es necesario que se navegue por todo el campo de brócolis, se solucionó el problema posicionando al robot cerca del principio de la ruta.

Otra particularidad de este algoritmo es que cuando el robot no detecta más puntos a los que seguir, deja de perseguir la ruta preestablecida y directamente se dirige al punto final del trayecto por el camino más corto. Esto ocurría al pasar por los puntos de seguridad, pues están a cinco metros de distancia de la hilera y no hay ningún waypoint definido entre medio. La solución de esto fue interpolar los puntos, consiguiendo que en la ruta no hubiera ningún trozo sin waypoints a los que seguir, que era lo que provocaba que el robot se perdiera.

Después de resolver los problemas anteriores, se probó con varios valores de sus parámetros hasta conseguir que el robot terminara el recorrido. El resultado puede verse en la **Figura 41**.

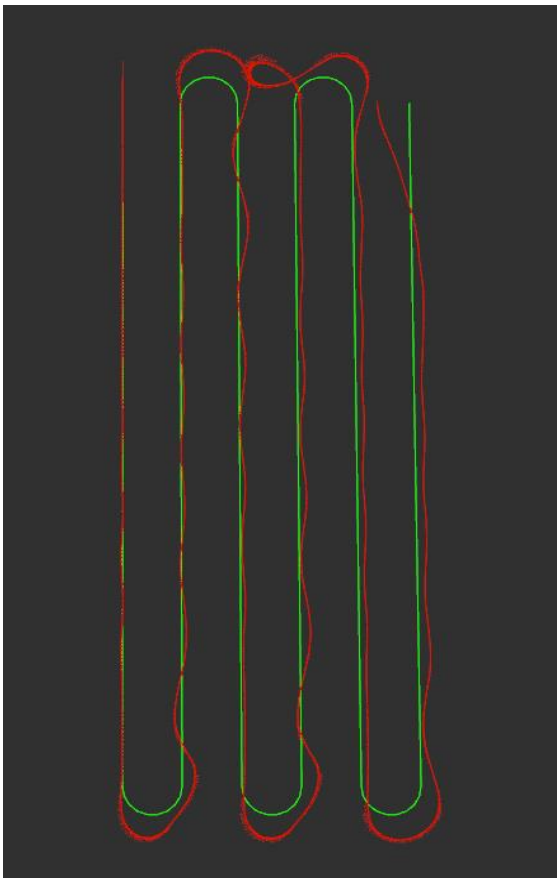

Figura 41 - Resultado con MBF en la primera iteración

Se observa que el comportamiento no es el deseado, principalmente se aprecian oscilaciones al recorrer las hileras, así como problemas al tomar las curvas. La trayectoria no es correcta, por lo que se asume que el controlador de la trayectoria, el TEB Local Planner, no está bien configurado. Las razones pueden ser varias:

- El parámetro que controla la velocidad angular máxima del robot, si tiene un valor muy alto puede hacer que el robot gire bruscamente, contribuyendo al patrón de Zigzag.
- Los parámetros que limitan la aceleración lineal y angular máxima. Si la aceleración es demasiado alta, el robot puede tener dificultades para cambiar de dirección suavemente.
- weight_viapoint: Un peso muy alto en este parámetro da rigidez a la navegación, ya que sirve para minimizar la distancia a los waypoints. Disminuyendo este valor se relaja la condición de pasar exactamente por cada punto.
- Aumentar el Look-Ahead: Como se evidenció en las pruebas realizadas con el algoritmo Pure Pursuit, valores muy bajos de "Look-Ahead" pueden generar oscilaciones debido a una

respuesta reactiva del robot. Por esta razón, se decidió aumentar este valor. No obstante, es crucial ser cauteloso al incrementarlo, ya que se experimentaron problemas cuando este valor era excesivamente alto. Cuando el robot se aproxima a la penúltima curva, se acerca lo suficiente al punto final para que sea visible con el valor de "Look-Ahead" configurado. Esto lleva a que el robot se detenga, creyendo que ha alcanzado su destino.

Luego de realizar ajustes en los parámetros mencionados anteriormente, se lograron significativas mejoras, evidenciadas en la **Figura 42**. En esta ocasión, las oscilaciones han sido completamente eliminadas, y los giros han sido mejorados. No obstante, persisten dificultades en el cuarto giro que aún requieren solución.

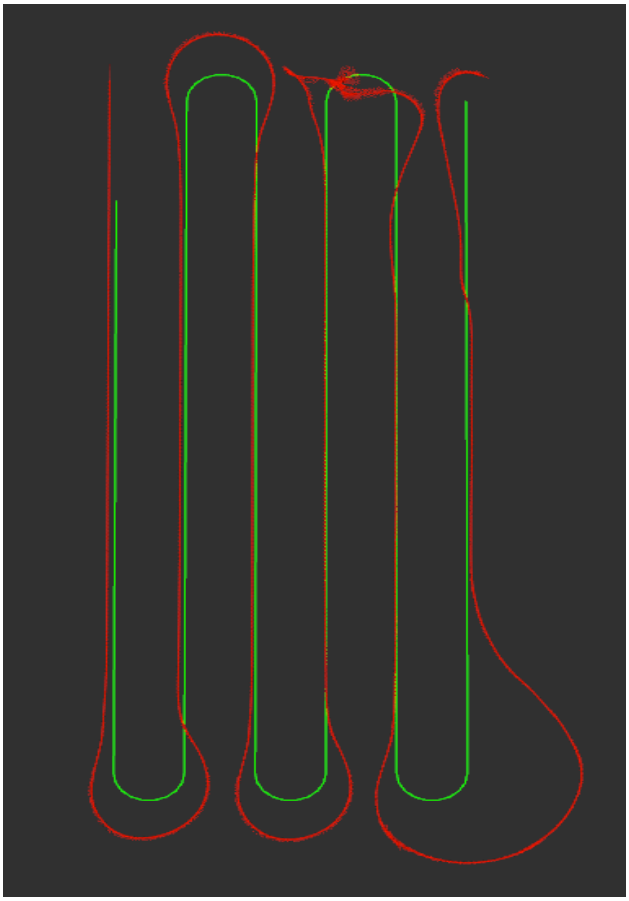

Figura 42 - Resultado con MBF en la segunda iteración

En esta última iteración se intentan mejorar los giros. Para ello se hacen los siguientes ajustes:

- Se vuelven a ajustar los valores límites de velocidad y aceleración angular, así como al peso que se le da a cumplir con dicho valor. Ambos se aumentaron, pensando que la velocidad angular y su aceleración eran demasiado bajos imposibilitando realizar curvas más cerradas.
- Se aumenta el valor del parámetro ***weight_kinematics_turning_radius*** para forzar a usar el mínimo radio de giro.
- Se detectó que la “huella” del robot (***footprint_model***), definido tanto en el Teb Local Planner como en el mapa de costes global, estaba mal fijado. Este parámetro es importante para la planificación de trayectorias, ya que define el tamaño del robot.

El reajuste de estos parámetros dio como resultado el siguiente comportamiento mostrado en la **Figura 43**.

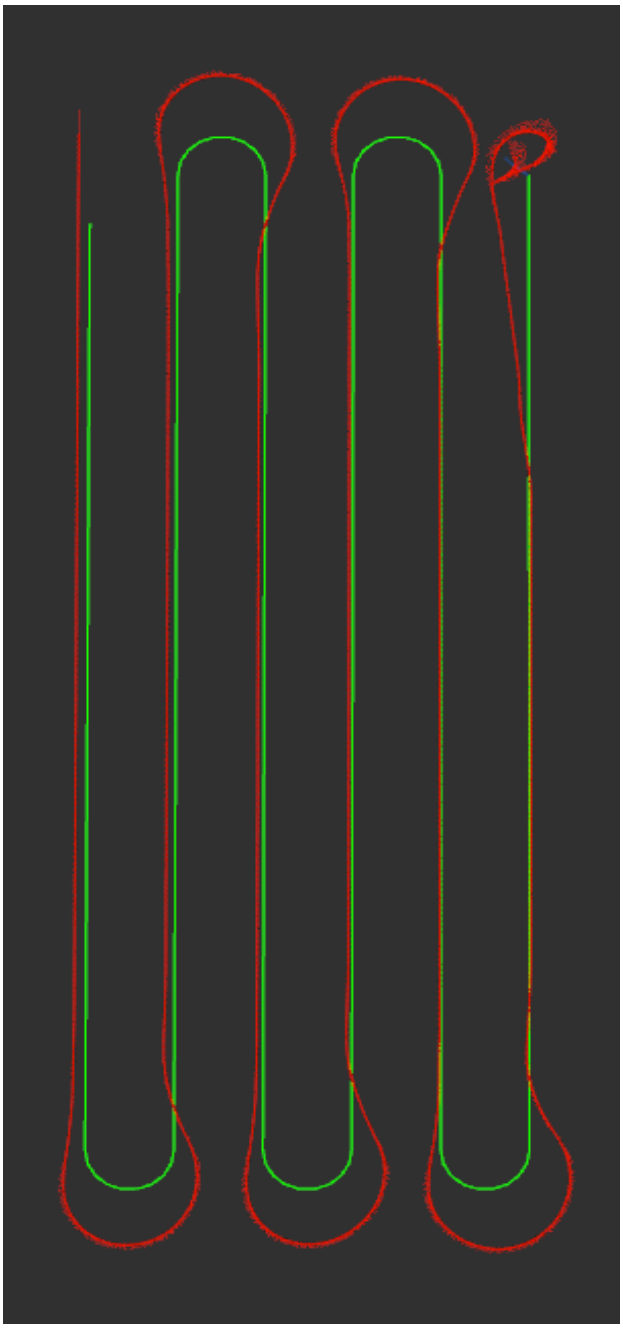

Figura 43 - Resultado final usando MBF

A pesar de las mejoras en la navegación, en comparación con el mejor rendimiento alcanzado mediante Pure Pursuit (con un "Look-Ahead" de 1.8 m y una velocidad lineal de 0.75 m/s), los resultados obtenidos con MBF son menos satisfactorios.

En favor de MBF frente a Pure Pursuit, es notable que el robot logra incorporarse a las hileras más rápidamente después de las curvas. Además, el planificador local ajusta continuamente la velocidad a lo largo del recorrido, aumentándola o disminuyéndola según convenga. Por otro lado, con MBF, a diferencia de con Pure Pursuit, el robot se ha conseguido recuperar en los momentos en los que ha perdido el control. Se cree que aún no se han alcanzado los parámetros óptimos, pero se confía en que una vez conseguido el comportamiento del robot mejorará. En cualquier caso, la implementación del MBF no ha sido en vano, ya que abre la puerta a nuevas funcionalidades muy interesantes para futuros desarrollos, como es la evasión de obstáculos.

5. CONCLUSIONES

Este proyecto detalla el proceso de desarrollo de un entorno de simulación y la implementación de funciones de guiado para un robot terrestre agrícola, permitiéndole navegar autónomamente por un campo de brócolis sin causar daños. Para llevar a cabo este trabajo, se utilizó ROS en su versión Melodic y Gazebo para la simulación del entorno, además de otras herramientas como Git y GitHub para el control de versiones.

Se ha demostrado que mediante un algoritmo Pure Pursuit los resultados son satisfactorios. Para ello se han analizado diferentes escenarios en los que se usaban distintas velocidades lineales y valores de Look-Ahead evidenciando los efectos de alterar ambos parámetros, así como la relación que guardan entre ellos. Se ha comprobado cómo con valores altos de Look-Ahead la reacción del robot es más suave y se produce el efecto de la esquina cortada. Mientras que con valores más bajos, el comportamiento es más reactivo pudiendo provocar oscilaciones, y en el caso extremo la pérdida de control del robot. Respecto a la relación que guardan la velocidad y el Look-Ahead, se confirma que a mayor velocidad el valor del Look-Ahead debe crecer para darle más tiempo de reacción al robot.

De todos los experimentos hechos, con el que se han obtenido mejores resultados ha sido con una velocidad lineal de 0.75 m/s y un Look-Ahead de 1.8 m. De esta forma se consigue un buen compromiso entre eficiencia y, sobre todo, precisión, qué es lo que más requiere esta aplicación. Cabe mencionar que la precisión en los giros es mejorable, en todos los casos el robot ha tomado la curva de forma más abierta y necesitando cierto tiempo para incorporarse de nuevo a la hilera. Aunque la precisión en los giros no es imprescindible, debido a que se realizan a una distancia de seguridad de los brócolis para dificultar que sean dañados, mejorarlos aumentaría la eficiencia. Esto permitiría reducir el margen de seguridad sin preocupación por dañar la plantación.

Como extensión del alcance del proyecto se implementa Move Base Flex (MBF) con el fin de probar un sistema de navegación más potente que Pure Pursuit. A diferencia de este, MBF cuenta con un planificador local y global, así como modos de recuperación. También permite evitar obstáculos y modificar los valores de velocidad según convenga. Por tanto, se esperan resultados mejores que los obtenidos con el Pure Pursuit. Sin embargo, en comparación con el mejor rendimiento alcanzado mediante Pure Pursuit (con un "Look-Ahead" de 1.8 m y una velocidad lineal de 0.75 m/s), los resultados obtenidos con MBF son menos satisfactorios. En favor de MBF frente a Pure Pursuit, el robot se ha conseguido recuperar en los momentos en los que ha perdido el control. Además, aporta otras funcionalidades que Pure Pursuit no. Se cree que aún no se han alcanzado los parámetros óptimos, pero se confía en que una vez conseguido el comportamiento del robot mejorará.

En conclusión, este trabajo fin de máster que ha contribuido al proyecto ELISA, cumple con los objetivos técnicos y personales acordados por el tutor del proyecto y el alumno. Sin embargo, siempre hay posibilidad de mejora, por lo que se proponen a continuación formas de continuar este proyecto.

Como trabajo futuro se propone:

- Mejorar la precisión del robot en los giros. Se cree que el motivo de la falta de precisión en las curvas es la inercia que lleva el autómata debido a la velocidad que trae de recorrer la hilera. Además de eso, en una curva tan cerrada, sería conveniente tener un Look-Ahead menor para que el robot no se salte ningún punto y sea más reactivo. Por tanto, se propone modificar el algoritmo Pure Pursuit para que comande una velocidad lineal y un Look-Ahead variable dependiendo de si el robot está recorriendo la fila de brócolis o está girando. Esto es, para navegar por la hilera la velocidad y el Look-Ahead debe ser grande y al llegar a la curva se deben reducir ambos.
- Probar con otros algoritmos de navegación más sofisticados que permitan, por ejemplo, la detección de obstáculos y tengan métodos de recuperación. Por ejemplo, continuar con el trabajo aquí empezado con Move Base Flex.

- Hacer uso de cámaras, lídar u otros sensores para reconocer el entorno y evitar obstáculos o deformidades en el terreno. Así como aumentar la precisión del posicionamiento del robot, que hasta ahora solo se hace mediante la odometría.
- Implementar otras funcionalidades diferentes de la navegación, como pueden ser el reconocimiento de las plantas mediante técnicas de percepción, la recogida del brócoli o la detección y remedio de enfermedades del cultivo.

6. ANEXOS

A continuación, se detallan los comandos necesarios para ejecutar el proyecto contenido en el repositorio (Ref. [2]):

1. Se lanza la simulación en Gazebo mediante el comando:

   roslaunch agribot_gazebo agribot_farm.launch

2. Se especifica la transformada para superponer el fram *map* y *odom* mediante el comando

   rosrun tf static_transform_publisher 0 0 0 0 0 0 map odom 100

Si se quiere lanzar el Pure Pursuit:

3. Lanzar el Pure Pursuit. Estando en la carpeta */ROS_Pure_Pursuit/launch*:

   roslaunch pure_pursuit_node.launch

4. Lanzar el nodo que envía los waypoints. Estando en la carpeta */ROS_Waypoints_Processor*:

   python load_waypoints_purePursuit.py

Si se quiere lanzar Move Base Flex:

3. Lanzar move base flex.

   roslaunch agribot_navigation agribot_navigation.launch

4. Lanzar el nodo que envía los waypoints. Estando en la carpeta */ROS_Waypoints_Processor:*

   python load_waypoints_moveBase.py

Si se quiere modificar el entorno en Gazebo, esto es: el número de hileras, la longitud de cada una de ellas, así como el número máximo de brócolis por hilera, los cambios hay que hacerlos en **GazeboCropRowGenerator.py**. En la siguiente figura se muestran los parámetros a modificar.

```
Row_Num = 6             # Numero de hileras
hilera = 2              # Numero de plantas en paralelo por cada hilera
Max_BPR = 20            # Numero máximo de plantas grandes en cada hilera
Row_lenght = 10         # Longitud de las hileras en metros
```

Figura 44 - Parámetros a modificar para un nuevo escenario en Gazebo

Cuando se hayan hecho los cambios, se ejecuta el código y se generará un archivo llamado **FarmWithCropRow.world** que debe ser guardado en la carpeta */agribot_gazebo/worlds.*