Universidad Zaragoza
1542

Trabajo Fin de Grado

Conducción autónoma de robot manipulador móvil mediante detección y seguimiento de carriles

Self driving of robotic mobile manipulator based on road lane detection and tracking

Autor

Oscar Alejandro Aina

Directores

Gonzalo López Nicolás
Rosario Aragüés Muñoz

ESCUELA DE INGENIERÍA Y ARQUITECTURA

2021

Conducción autónoma de robot manipulador móvil mediante detección y seguimiento de carriles

Resumen:

Los avances en la navegación robótica son cada vez mayores, no solo en entornos industriales sino también en ámbitos públicos como la conducción autónoma de vehículos. Es por ello por lo que el desarrollo de estrategias de navegación se está extendiendo y profundizando cada día más para obtener mejores resultados.

El objetivo de este trabajo es el desarrollo de un sistema de navegación autónomo que mediante la información recibida por una cámara sea capaz de detectar marcas, que en nuestro caso son líneas marcadas en el suelo, y navegar siguiendo dichas líneas, pudiendo enfrentarse a situaciones distintas. Una de ellas podría ser la pérdida de la línea y su posterior búsqueda, o incluso la posibilidad de tener obstáculos en medio de su recorrido pudiendo evadirlos y proseguir su trayecto sin problemas, haciendo uso para ello del sensor láser. En definitiva, el objetivo principal es la realización de un sistema de navegación autónomo mediante marcas, a través de una cámara y un sensor láser, que permita realizar una navegación independiente.

En este trabajo se ha utilizado el entorno ROS y sus herramientas, ya que está específicamente diseñado para trabajar con robots y también porque cuenta con una comunidad detrás muy activa. Aparte del desarrollo de las estrategias que nos permiten hacer la navegación, en este proyecto se han diseñado entornos en Gazebo para poder efectuar simulaciones. Para dichos entornos se han creado también los correspondientes mapas del entorno.

Se han cumplido todos los objetivos: se han desarrollado mecanismos de percepción, estrategias de comportamiento, implementación de sistemas de navegación, análisis y evaluación de resultados, que han sido positivos, empleando para ello el robot prototipo desarrollado en el marco del proyecto europeo COMMANDIA. Además, en este trabajo se ha aportado información relevante derivada del cambio de versión de ROS, de Kinetic a Melodic. Lo mencionado anteriormente es debido a que el conjunto de paquetes original del robot usado está pensado para ROS Kinetic una versión que se está quedando desactualizada. Por tanto, en este trabajo se ha procedido a la migración de los mencionados paquetes a la versión ROS Melodic.

Índice

1. Introducción

1.1. Contexto y Motivación

Los avances tanto en la robótica como en la navegación autónoma son cada vez mayores, dentro no solo de la logística industrial sino también dentro de los sectores servicios. Uno de los objetivos es facilitar el trabajo a los trabajadores, poniendo su seguridad en menor riesgo en tareas que presenten cierta peligrosidad. De igual modo se utilizan para facilitar la ejecución de tareas que son tediosas o requieran alta precisión y/o rapidez en la realización. Por estas razones, cada vez se desarrollan más proyectos que se enfocan en la robotización de distintos procesos. Estos pueden ser tanto industriales como del sector servicios, dentro de los cuales se pueden encontrar a los robots manipuladores móviles o simplemente a robots autónomos, como los de la *Figura 1*.

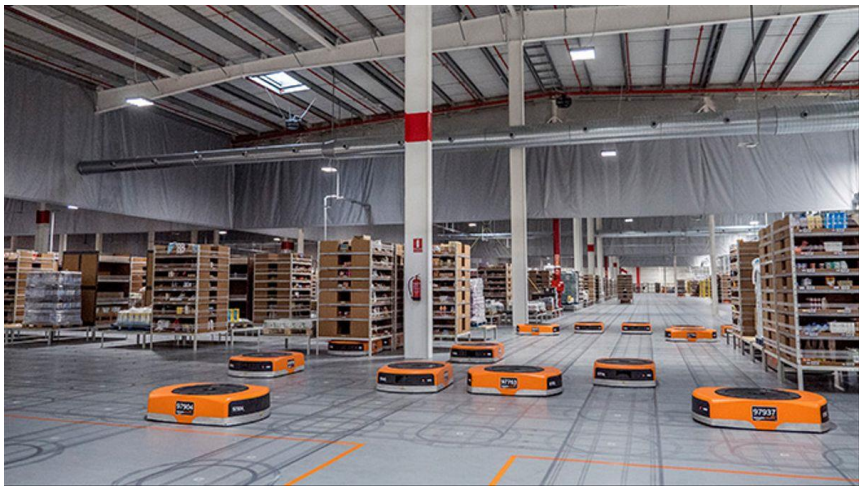


Figura 1: Robots autónomos utilizados en los almacenes de Amazon. / www.elperiodico.com, RICARD FADRIQUE

Los robots manipuladores móviles son capaces de realizar, por una parte, tareas de fabricación, como los ya presentes robots industriales los cuales prestan ayuda a los operarios o directamente se encargan de tareas dentro de las cadenas de montaje. Por otra parte, tareas como el seguimiento de objetivos y navegación en cualquier tipo de entorno, posibilitando el traslado de las tareas de fabricación de una zona a otra o simplemente la búsqueda de distintos objetivos. Además de ejecutar tareas de logística como el traslado de recursos desde distintos puntos dentro de un recorrido conocido.

Aunque esto último, todavía sigue en progreso y cada día se avanza más hacia robots que posibiliten la realización de diferentes tareas dentro de una misma producción, mediante el desplazamiento de sus herramientas de manipulación o del traslado de los recursos a aquellos lugares donde son necesarios.

Algo común en estos robots manipuladores móviles será la navegación autónoma en ambientes varios, siendo esta navegación autónoma otro de los puntos que más se están desarrollando y donde más avances se pueden encontrar hoy en día.
En este TFG por lo tanto nos centraremos en esta navegación autónoma por parte del robot utilizado, que se tratará del prototipo, robot denominado Campero, desarrollado en el marco del proyecto europeo COMMANDIA [3], por medio tanto de la cámara delantera que posee como de los sensores láser. A partir del prototipo comentado se ha desarrollado el modelo comercial RB-EKEN que se puede ver en la *Figura 2*.

Figura 2: Modelo comercial de Campero.

Además, se tendrá la certeza de que se trabaja en un entorno seguro con este robot debido a los sistemas de seguridad que posee, pese a las dimensiones que presenta. Por otro lado, Campero trabajará fácilmente en interiores gracias a que posee dos tipos de ruedas unas de goma y otras mecanum, estos dos tipos de rueda nos posibilitan poder trabajar con dos tipos de movimiento distintos, diferencial y omnidireccional. Mientras que las ruedas de goma solo permiten movimientos diferenciales, las mecanum además de estos movimientos diferenciales permiten también realizar movimientos omnidireccionales. Y gracias a que permiten realizar ambos tipos de movimientos es por lo que se ha decidido utilizar este tipo de ruedas en el trabajo, ya que estas otorgan la posibilidad de que la navegación que se desarrolle se pueda adaptar a uno u otro movimiento según convenga.

Para el desarrollo de la navegación autónoma, la cual se ha comentado que será el centro de este proyecto, será necesario la creación de estrategias de navegación basadas en la percepción de carriles a través de las imágenes obtenidas por la cámara que posee Campero. Por otra parte, dentro de estas estrategias también se deberá dar solución a

problemas que puedan producirse como: la perdida de la línea por parte del robot durante la realización de su recorrido o la aparición de obstáculos, los cuales mediante el sensor delantero que posee sea capaz de evadir. Es por todo lo dicho anteriormente que en este trabajo se desarrolla una navegación capaz de ejecutar una navegación autosuficiente. Tanto para el desarrollo de lo que se acaba de comentar como para su estudio se utilizará el entorno ROS (Robot Operating System).

Este trabajo entra dentro de las actividades realizadas por el proyecto COMMANDIA, el cual es un proyecto cofinanciado por el Programa Interreg Sudoe y por el Fondo Europeo de Desarrollo Regional (FEDER). Cuyo nombre COMMANDIA es el acrónimo en inglés de Robótica móvil colaborativa de objetos deformables en aplicaciones industriales.

1.2. Objetivos

El objetivo principal de este trabajo es el desarrollo de una navegación autónoma basada en el seguimiento de marcas, que son carriles que podrán variar de color y grosor, situadas en su entorno. Mientras el robot las recorre, éste es capaz de esquivar los posibles obstáculos que pudiera encontrarse, además de volver a la línea si la perdiese de vista. Algo que podría asemejarse a la *Figura 3*.

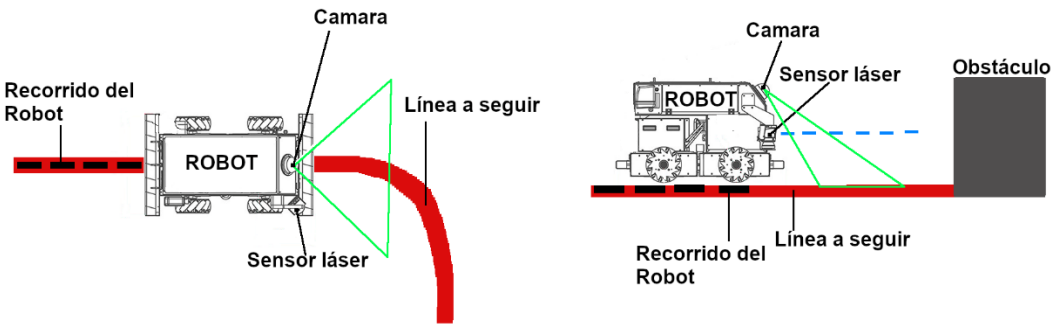


Figura 3: Imagen ilustrativa del objetivo del trabajo.

Para conseguir el objetivo global se completan unas ciertas metas u objetivos más específicos:

1. Instalar tanto la versión Ubuntu con la que se va a trabajar (Ubuntu 18.04 Bionic), como su correspondiente versión de ROS compatible, ROS Melodic Morenia, y todos los paquetes que sean necesarios.

2. Familiarizarse con el entorno ROS y sus funcionalidades a través de su página oficial.

3. Descargar las carpetas y paquetes del simulador del robot Campero.

4. Migración de los paquetes del robot Campero a la versión ROS Melodic.

5. Familiarizarse con el Campero mediante los ficheros existentes y entenderlos.

6. Tras aprender lo básico sobre ROS y haberse familiarizado al Campero, se desarrollarán estrategias propias de navegación a través de Python[17], versión 2.7.17. Por un lado, se realizará el estudio y desarrollo de un sistema de percepción de carriles que nos otorgue la información deseada, además de la creación de un sistema de detección y evasión de obstáculos. Por otra parte, también se generará un sistema de búsqueda de línea ante posibles pérdidas, al igual que se desarrollarán varios sistemas de navegación, los cuales se estudiarán y de los cuales se analizará su viabilidad.

7. Validar en simulación los métodos de navegación desarrollados y realizar validaciones experimentales mediante implementación en el Campero real.

8. Finalmente analizar los resultados y desarrollar la memoria del proyecto.

1.3. Organización del documento

Este documento se organizará de la siguiente forma:

-En la sección 1, en la cual nos encontramos, se realiza una introducción y se pone en contexto este trabajo.

-En la sección 2, se hace una introducción a ROS y también se habla sobre las herramientas utilizadas y los archivos modificados debidos al cambio de versión de Ubuntu.

-En la sección 3, encontramos una explicación general de los métodos de navegación desarrollados en este trabajo. Al igual que se incluyen el análisis y comparación de estos métodos de navegación.

-En la sección 4, tenemos una explicación más detallada del funcionamiento interno de las estrategias de navegación y de sus componentes, como el sistema de percepción de línea, la evasión de obstáculos, la búsqueda de línea tras perdida o los métodos de navegación. Además de incluir la relación entre los distintos scripts.

-En la sección 5, se recogen varios experimentos que hemos realizado en simulación y su análisis.

-En la sección 6, se realiza una guía con los pasos necesarios para realizar la implementación con el robot real.

-En la sección 7, desarrollamos las conclusiones y hablamos de los posibles trabajos o mejoras futuras que se pueden realizar tras este proyecto.

-Bibliografía.

-Anexo A, se recogen los programas desarrollados junto con una breve explicación de estos al principio de cada uno.

-Anexo B, encontramos un manual de usuario del orden en el que se deben ejecutar los programas desarrollados.

2. Entorno ROS

2.1. Introducción a ROS

Figura 4: Logotipo de ROS

ROS se corresponde con el acrónimo inglés *Robot Operating System* [1], del cual se puede intuir que estamos ante una especie de sistema operativo. Siendo ROS un meta-sistema operativo especialmente diseñado para usarse en robots. Además de ser este un sistema operativo específico para robots, también se encuentra que es de código abierto y proporciona los servicios estándares de un sistema operativo común. Como pueden ser la abstracción del hardware, el control de dispositivos de bajo nivel, la implementación de funcionalidad de uso común, el paso de mensajes entre procesos y el mantenimiento de paquetes.

Las dos partes básicas de ROS son, por un lado, la parte del sistema operativo y por otro lado ros-pkg, la cual consiste en una suite de paquetes aportados por la contribución de usuarios que implementan distintas funcionalidades, como pueden ser la localización, la planificación o el mapeo entre otras muchas más. Todo ello gracias a como se ha comentado ser ROS de código abierto, además de estar diseñado para que distintos paquetes o códigos, pertenecientes a proyectos distintos, sean capaces de funcionar en implementaciones diferentes dentro de otros proyectos.

2.2. Conceptos básicos y Comunicación

Este meta-sistema está basado en una arquitectura de grafos donde el procesamiento toma lugar en los **nodos** que pueden recibir, mandar y multiplexar mensajes de sensores, control, planificaciones, entre otros. Todo ello a través de **topics** y **mensajes**, mientras son controlados y registrados todos ellos por el **Master**.

Algunas definiciones/conceptos relevantes de ROS son los siguientes:

- **Master:** Es el nodo principal y el encargado de dar nombre y registrar los servicios de los nodos, así como monitorizar el tráfico y encargarse de habilitar y mantener localizados a estos nodos. Sin el Master los nodos no podrían encontrarse, ni comunicarse.

- **Nodos:** Son los procesos o los programas que se ejecutan y que realizan funciones concretas, como el mapeado, la planificación de rutas o el acceso a motores. Cada nodo es compilado por separado y todos son gestionados y registrados por un nodo principal que es el Master. La comunicación entre estos nodos se produce a través de los topics, donde mandan o leen información por medio de mensajes o servicios.

- **Topics:** Son el nombre que utilizamos para identificar el contenido de la información que llevan los mensajes, que proporcionan o escuchan los nodos. Debido a que estos mensajes están dirigidos a través de un sistema de transporte de información, que trabaja a través de la publicación y la suscripción, los nodos pueden publicar información en un topic determinado y a la vez subscribirse para recabar contenido que les sea de interés en otro distinto. Pudiendo coexistir varios publicistas o subscriptores en el mismo topic.

- **Mensajes:** Los mensajes son el medio que tienen los nodos para comunicarse entre sí, comprenden campos escritos y admiten los tipos básicos como enteros, floats, booleanos, etc. Además de estructuras como arrays de estos tipos de datos y también estructuras que se puedan encontrar en C.

La comunicación que se da en ROS es una comunicación basada en la subscripción y publicación de información en los topics por parte de los nodos. La información que se publica en estos topics debe ser únicamente de un tipo de mensaje, por lo que cada nodo interesado en publicar o subscribirse a dicho topic tendrá que ser capaz de generar o recibir dicho tipo de mensaje de antemano.

Este tipo de comunicación permite que varios nodos sean capaces de obtener información de un mismo topic, lo que facilita que diferentes nodos encargados de tareas completamente distintas puedan desempeñar sus respectivas funciones a través de la información que reciben, sin molestar o interferir en el resto de los nodos.

Figura 5: Ejemplo comunicación nodos. Rqt-graph comando ROS

La *Figura 5* obtenida por el comando de ROS: *rosrun rqt_graph rqt_graph*, muestra un ejemplo de la comunicación entre dos nodos, en azul (move_base) y verde (twist_mux), a través del topic en rojo de la cual hablábamos anteriormente. El nodo azul es un proceso que genera un movimiento en función de una posición dada, este nodo publica en el topic cmd_vel la información de la velocidad tanto lineal como angular mediante un mensaje. La información que se publica por el nodo azul en el topic será visualizable y alcanzable por todos los nodos que estén interesados en esta información, en el ejemplo únicamente el nodo verde se suscribirá a este topic para obtener esta información, con la que calculará la velocidad de cada motor.

2.3. Entornos de simulación ROS

En este punto se explicarán los programas utilizados en el trabajo, todos ellos son las herramientas usualmente más utilizadas junto a ROS debido a que potencian su funcionalidad. Estas herramientas permiten visualizar la información, además de posibilitar su guardado y por otra parte permiten la ejecución de scripts, que en el caso de este trabajo estarán sobre todo orientados a la navegación y a la recepción y manipulación de imágenes.

2.3.1. Gazebo

Gazebo [4] es un simulador de robótica 3D de código abierto, que da la posibilidad de evaluar el comportamiento de un robot en un mundo virtual. Gracias a que permite diseñar o importar como en nuestro caso modelos de robots, además Gazebo permite sincronizarse con ROS para que lo robots que se emulen publiquen la información de sus sensores, como se haría en un entorno real. Con dicha información después se podrá implementar una lógica y un control para manejar el robot según sean nuestras tareas e intereses.

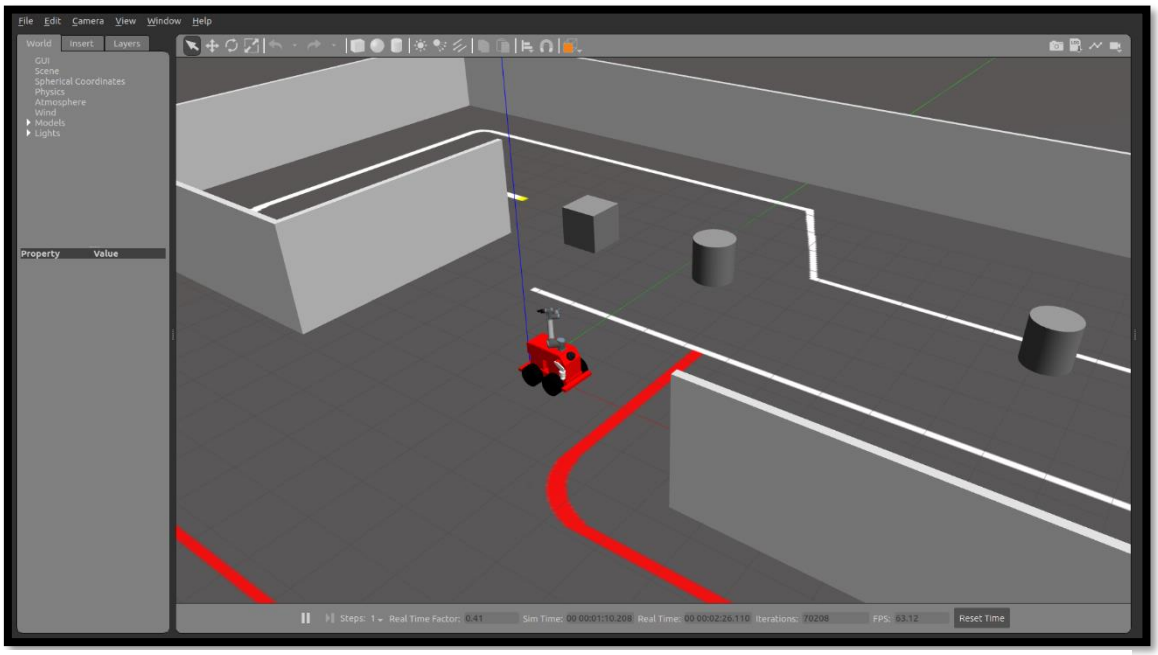


Figura 6: Ejemplo de mundo virtual en Gazebo desarrollado durante la realización de este trabajo.

En la *Figura 6* se puede observar un entorno creado directamente en Gazebo, donde se tienen tanto el modelo del robot como distintos obstáculos, cubos y cilindros, y marcas, líneas de colores. Los obstáculos que vemos son creados directamente a partir de los modelos que proporciona Gazebo, aunque se pueden crear los que uno quiera con programas externos y cargarlos en Gazebo, y que permiten crear el mundo virtual donde se situará posteriormente el robot, las líneas sin embargo son la adición de una textura al suelo del mundo de Gazebo. El robot por su parte es un modelo procedente del simulador de Campero que se importará al mundo de Gazebo generado con anterioridad.

Debido a que en Gazebo tenemos tanto obstáculos como robots con físicas propias y vemos los detalles del entorno, es el motivo por el que los sensores y cámaras pueden devolver la información que hay alrededor del robot, y por lo que Gazebo es tan importante a la hora de simular ya que es esta información la que como se ha comentado se utilizara para desarrollar los programas que creemos. En nuestro caso lo más importante serán los datos que nos devuelva Gazebo tanto de la cámara como de los sensores de proximidad.

Tanto el mundo de Gazebo como la importación del robot se hará mediante un archivo *roslaunch* (.launch), el cual permite generar el mundo e importar en él los robots que se vayan a utilizar a la vez.

Si únicamente se requiere lanzar un mundo de Gazebo para su posible modificación se puede mediante las instrucciones:
Roscore, comando fundamental e imprescindible para que los nodos funcionen, ya que este pone en ejecución el Master.

Rosrun gazebo gazebo /path/to/ .world file, este comando ejecuta Gazebo inicializando en el mundo que se haya indicado en la dirección y en el nombre.world.

2.3.2. RViz

Rviz [5] es una herramienta de visualización en 3D para aplicaciones ROS, proporcionando una vista del modelo del robot, permitiendo además la captura de la información de los sensores y la reproducción de los datos capturados. También permite interactuar con el robot a través de distintas formas que se explicarán posteriormente.
Gracias a Rviz se puede representar los datos de los sensores, así como qué está capturando la cámara en el mundo virtual.

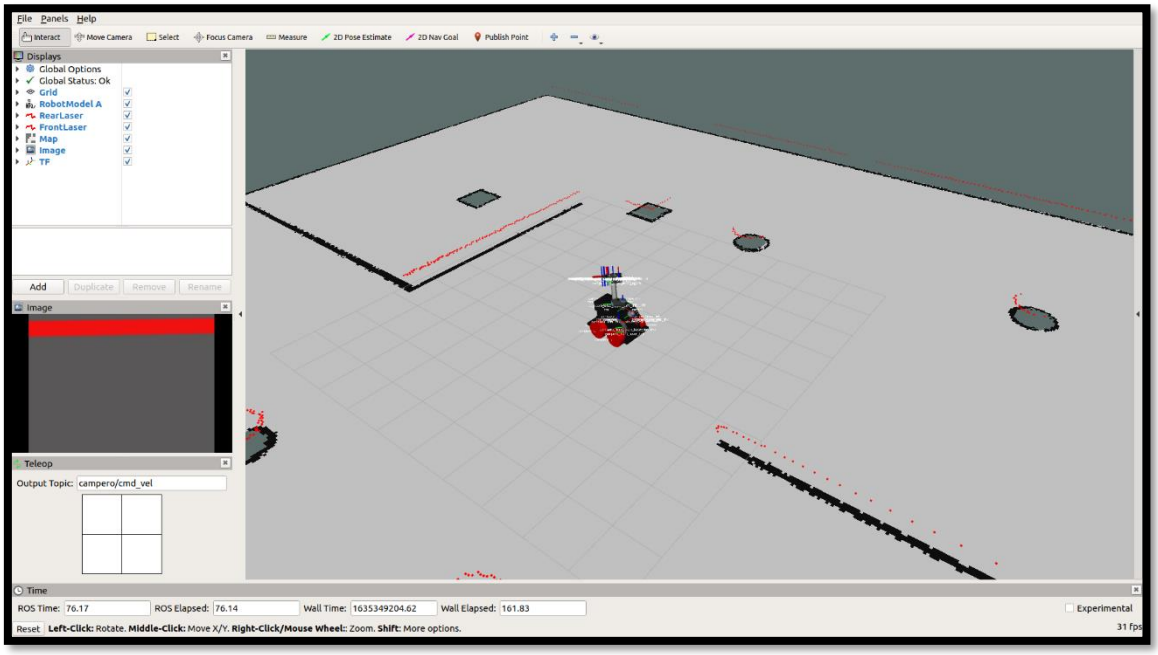


Figura 7: Ejemplo de simulación en Rviz desarrollado durante este trabajo.

En la *Figura 7* se puede ver la representación del mundo que teníamos en Gazebo en la *Figura 6* pero en Rviz, esta imagen representa el mapa 3D de la figura 6 pero en 2D por medio de su mapeado, donde se puede observar con puntos o trazos negros la información de los obstáculos que en el mundo 3D haya, además también se observa la información que detectan tanto los sensores de proximidad del robot, con los puntos rojos que hay en la pantalla, como la cámara, en la parte izquierda en el recuadro que pone *image*.
Gracias a esta información, Rviz se puede utilizar para tomar distancias ya que además tiene herramientas de medición y con las cuales se pueden comprobar y medir para realizar distintos ajustes en los programas.
Rviz a partir del *teleoperation panel* que se encuentra en el recuadro inferior izquierda nos permite realizar la tele operación con lo que se puede manejar el robot por el mundo

con suma facilidad, además de este modo que es externo a Rviz y que es un plugin que se debe instalar, Rviz proporciona la opción de posicionar al robot de forma aproximada y colocar una posición a la que navegue, esas opciones las se tienen en la parte superior (2D *Pose Estimate* y 2D *Nav Goal*).

En Rviz también se pueden visualizar si así se desean los ejes tanto del robot como del origen del mapa o incluso de las marcas a las cuales y sobre las cuales se realizan las transformaciones para ir hasta ellas. Es por todo lo dicho anteriormente que Rviz es una de las herramientas fundamentales, junto a Gazebo, a la hora de trabajar con ROS puesto que como se ha visto otorga mucha información y además muy valiosa.

2.4. Adaptación de archivos de versión Kinetic a Melodic

Los archivos que se utilizan como punto de partida para trabajar con el robot Campero en ROS son los archivos modificados por el TFG [2], y en los que ya había cambios con respecto a los originales que entregaba Robotnik, empresa fabricante del robot. Sin embargo, en este trabajo también se han realizado algunos cambios en los archivos, en este apartado se habla de los cambios más importantes que se realizaron en estos archivos base. Estos cambios se efectuaron debido al cambio de versión de Ubuntu, de la 16.04 a 18.04, ya que la 18.04 es una versión más actual y los paquetes del Campero son para la versión 16.04 que se está quedando desactualizada. Por ello en este trabajo ha sido necesario modificar varios archivos para lograr su adaptación a la versión empleada la 18.04.

-En primer lugar, nótese que hubo que añadir ficheros durante la instalación de los archivos de Campero, estos paquetes era necesarios añadirlos porque en la ejecución del Campero se solicitaban. Sin embargo, debido al cambio de versión, de Kinetic a Melodic, los siguientes paquetes se añadieron descargando directamente sus carpetas, para su posterior compilación desde archivos fuente, ya que no existía instalación directa desde el repositorio oficial de ROS.

-Costmap_prohibition_layer [12]
-Openrave [13]
-Robot_localization [14]
-Teleop_panel [15]
-Universal_robot [16]

-En segundo lugar, se ha de explicar uno de los errores que aparecían a la hora de lanzar el Campero. Este era producido debido a que era necesario cambiar la declaración de algunas de las macros que había en los archivos, porque por el cambio de versión habían actualizado la manera de hacerlo y producía fallos.

El mensaje que se obtiene por parte de ROS es:

Deprecated:xacro tag 'box_inertia' w/o 'xacro:' xml namespace prefix (will be forbidden in Noetic) when processing file: /home/… /name_file.urdf.xacro

ROS facilita un comando para corregir las etiquetas que contenían estos fallos, el cual es:

find . -iname "*.xacro" | xargs sed -i
's#<\/([/]\?\)\(if\|unless\|include\|arg\|property\|macro\|insert_block\)#<\1xacro:\2#g'

Dicho script solucionaba solo los fallos que estuvieran contenidos en las etiquetas recogidas en él, sin embargo, si la etiqueta era un nombre creado por los fabricantes los errores no se solucionaban automáticamente con este script. Por lo que este error se vuelve muy tedioso, ya que hay que buscar una por una cada etiqueta que te marca el error entre los distintos archivos.
Para facilitar el trabajo a futuras personas se indicarán los archivos específicos y las etiquetas modificadas, a las que había que añadir 'xacro:':

-campero_base.urdf.xacro, xacro:box_inertia

-rubber_wheel.urdf.xacro y mecanum_wheel.urdf.xacro, xacro:cylinder_inertia

-sick_s300.urdf.xacro, xacro:sensor_sick_s300_gazebo

-axis_m5525.urdf.xacro, xacro:sensor_axis_m5525_gazebo

-robotiq_fts300.urdf.xacro, xacro:insert_block

-Una vez solucionados estos problemas, aparecieron otros referidos a la navegación y más concretamente al *gmapping* del mapa, en los archivos:

-costmap_commo_params.yaml

-local_costmap_params.yaml

-global_costmap_params.yaml

En dichos archivos hubo que eliminar las declaraciones static_map puesto que con el cambio de versión habían quedado anticuadas, además en el primero de ellos se debían añadir *plugins* por demanda de ROS, cuya declaración es:

plugins:

- {name: static_layer, type: "costmap_2d::StaticLayer"}

- {name: obstacles, type: "costmap_2d::VoxelLayer"}

Esta declaración se obtuvo directamente de la página oficial de ROS[1] para costmap_2d.

Además de todo lo anterior, también hubo que añadir instancias al archivo campero_one_robot_nav.launch del *gmapping* las cuales si estaban en el campero_one_robot.launch.

<arg name="launch_gmapping" default="false"/>

<!-- gmapping -->

<include if="$(arg launch_gmapping)" file="$(find campero_localization)/launch/slam_gmapping.launch">

<arg name="prefix" value="$(arg prefix)"/>

</include>

Y también hubo que añadir un argumento a campero_nav.launch, aunque si no se hubiera querido cambiar campero_one_robot_nav.launch, se podría haber sustituido directamente campero_one_robot.launch por campero_one_robot_nav.launch en dicho archivo.

El argumento añadido:

<!-- Gmapping -->

<arg name="launch_gmapping" value="$(arg gmapping)"/>

3. Estrategias de navegación

En esta sección se efectúa una explicación general de los distintos tipos de estrategias de navegación que se han desarrollado en el trabajo, para crear una referencia de lo que se desarrollará en el apartado siguiente. Además de analizar la evolución de estas estrategias de navegación y realizar una comparación entre los puntos fuertes y débiles de ellas, destacando las ventajas y desventajas de emplear unas u otras. Algo que se debe destacar a priori es que todas las estrategias tendrán algunas cosas en común, como puede que usen la cámara PTZ que posee Campero para encontrar las marcas que debe seguir. Y, por otro lado, que utilizan todas ellas el sensor láser delantero, el cual proporciona información sobre si existe algún obstáculo que pudiese interrumpir el recorrido de Campero.

3.1. Navegación basada en centroides y a velocidad lineal no constante

Para empezar, se explicará el primer tipo de navegación que se desarrolló y el que sentó las bases de los dos siguientes. Esta navegación consistía en realizar recorridos entre puntos que se posicionaban en el mapa y que eran obtenidos a través de las imágenes que captaba la cámara como se explicará más adelante.

Básicamente, esta navegación efectuaba un control tanto en la velocidad lineal como en la angular, basándose en la diferencia que había entre la posición y orientación del Campero en un instante y el punto objetivo en ese mismo instante. Tras lo anterior se evaluaba si se había llegado a una cierta distancia del objetivo, en caso afirmativo se paraba el robot y se procedía a realizar una rectificación en la orientación del robot. De manera que el robot se orientase en el mismo sentido en el que estuviera el punto al que se quería llegar, de esta forma se procedía una y otra vez hasta que alcanzarse un final, en el caso en que lo hubiera.

Una vez se alcanzaba un punto con orientación y posición válidas en esta navegación se daba un aviso para qué se generará un nuevo punto al que trasladarse.

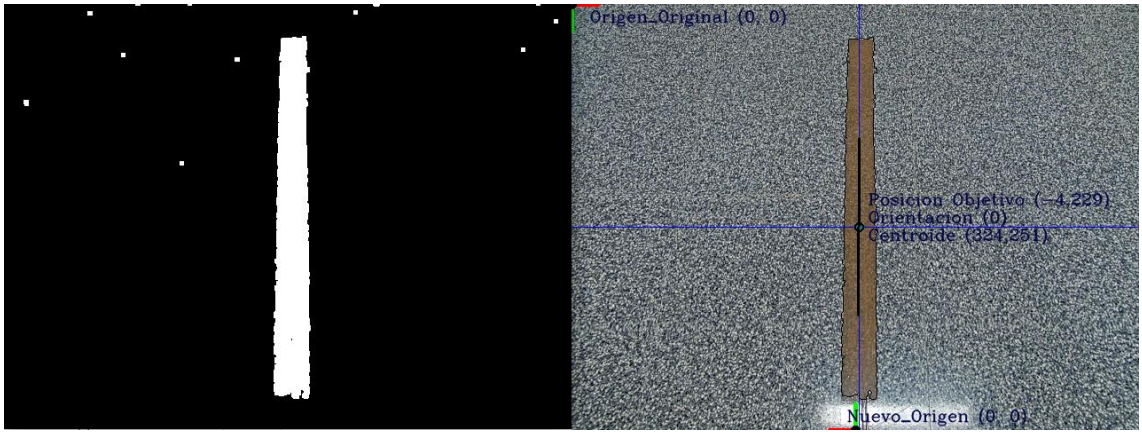


Figura 8: Representación de la mascara y de la información recopilada, como el centroide, posición objetivo y orientación, de una imagen captada por Campero en un experimento real.

Como indica el nombre de esta navegación se basaba en trabajar con centroides del contorno del carril que hubiera en la imagen recibida. Debido a que estos eran los que proporcionaban la posición en x e y, además de la orientación del punto objetivo al que se trasladaría el robot, como se observa en la *Figura 8*. De igual modo como indica el nombre era a velocidad lineal no constante, porque no se mantenía una velocidad constante durante toda la navegación, sino que era dependiente de la distancia a la que esté estuviera de cada objetivo o de si estaba esperando un nuevo objetivo, donde la velocidad era nula.

En esta navegación cuando se perdía la línea se realizaba una búsqueda de esta, de forma que el robot se orientase a izquierdas o a derechas de su posición progresivamente y en cada una de las situaciones se evaluaba si había línea, pero esto evaluación solamente se efectuaba cuando se estaba 90º a uno u otro lado exactamente. En caso de que no hubiera línea el robot volvía tras sus pasos y recorría en sentido inverso la línea, debido que se asumía que no había línea en el recorrido que se seguía.

Con respecto a la evasión de obstáculos en esta navegación se realizaba como en el resto de los tipos de navegación, cuya explicación se realizará más en profundidad en la siguiente sección.
Finalmente se ha de decir que esta navegación se desechó, por motivos que se explicarán en el apartado de análisis y comparación, y es por eso por lo que no se hablará de ella en las siguientes secciones.

3.2. Navegación basada en centroides y a velocidad lineal constante

A continuación, se encuentra el segundo tipo de navegación que se generó, el cual está basado en la navegación anterior solo que en esta se eliminaron las pausas para esperar nuevos objetivos y se estableció una velocidad lineal *x* constante e invariable, a excepción de cuando se pierda la línea y se proceda a su búsqueda o cuando se tiene que evadir un obstáculo.

Esta navegación se centra sobre todo en un movimiento del robot más fluido en el cual solamente se modifica y se controla la velocidad angular, la cual se regulará mediante un control proporcional para la orientación entre el robot y el punto objetivo, algo que se explica más en profundidad en la siguiente sección. Con lo que la rectificación que se hacía en la anterior navegación ya no se realiza, porque el robot con esta nueva navegación es capaz de rectificar en movimiento la orientación y posicionarse siempre encima de la línea o marca que se esté siguiendo, manteniendo una orientación adecuada sin necesidad de pararse.

Esta navegación sigue trabajando con los centroides, que observamos en la *Figura 8*, que obtenemos de igual modo que antes de los contornos del carril de la imagen, pues son los que generan los puntos objetivos sobre los que se desplaza el robot. Sin embargo, estos puntos ya no son reclamados por la navegación si no que se generan cada segundo con la información que se capta por la cámara, con esto permitimos que el robot no cambie su velocidad lineal y que avance siempre que tenga línea delante suyo.

También se ha modificado la búsqueda de línea, de modo que en esta navegación cuando se pierde la línea ya no es necesario llegar a 90º a izquierdas o derechas de la posición del robot, ya que si se encuentra la línea en el proceso de girar va directamente a esa posición de la línea detectada.

Por otro lado, la evasión de obstáculos es la misma que se realizaba en la primera navegación, dicha evasión se explica en el apartado 4.3.

3.3. Navegación con orientación a la línea prioritaria

Finalmente, se procede a explicar el último tipo de navegación que se ha desarrollado cuya estructura básica es la misma que las anteriores, pero con la diferencia de que se quiso y se consiguió eliminar la dependencia con los puntos que se generaban y directamente se trabaja con la información de la cámara.
En esta navegación se trabaja con velocidad lineal tanto en *x* como en *y*, gracias a las ruedas omnidireccionales que posee Campero, lo que da la ventaja de poder hacer desplazamientos no solo en avance sino también lateralmente, además de mantenerse la velocidad angular para realizar los giros.

El control sobre las velocidades esta vez será sobre la lineal en *y*, y la angular, donde el control de la velocidad lineal en *y* es un control proporcional de la distancia del centroide al punto de medio de la imagen captada por la cámara en el eje *x*. En la *Figura 9* tenemos el sistema de referencia del robot y el sistema de referencia global con el que se trabaja, con lo cual se puede intuir que los desplazamientos en avance se realizan sobre el eje *x* y los laterales sobre el eje *y* distinguiendo en este eje velocidades tanto positivas como negativas.
Por su parte el control de la velocidad angular será un control proporcional también de la orientación entre el punto del objetivo y el robot, pero esta vez con un offset que será la resta del valor que tenga la velocidad lineal en *y*, debido a su participación en los desplazamientos laterales.

La novedad de esta navegación es trabajar con una velocidad lineal en *y*, lo que hace que el robot siempre se sitúe de forma que el centroide este lo más centrado posible en la imagen captada, reduciendo a su vez el ángulo del robot con respecto a la línea debido a que siempre intenta posicionarse en el mismo sentido que esta. Por lo que los giros que se realizaran en este tipo de navegación son menos bruscos y por lo tanto las velocidades angulares son menores.

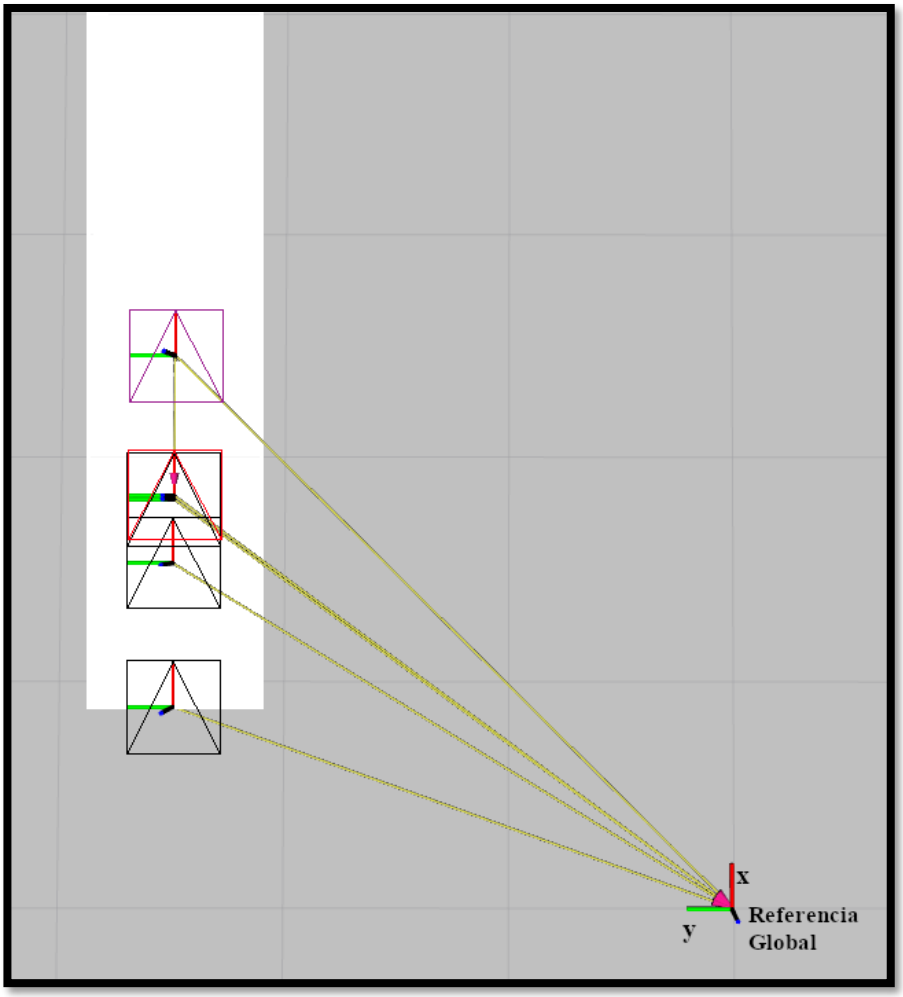


Figura 9: Sistema de referencias, donde se encuentra la referencia global, así como la posición actual(Rojo), pasadas(Negro) y futura(Violeta) del Campero y también la línea, en blanco, que sigue.

Por lo dicho anteriormente es por lo que esta navegación se llama Navegación con orientación a la línea prioritaria, ya que en ella se da prioridad a estar en una posición lo más centrada posible con respecto a la línea, algo que nos lo indica el centroide de la *Figura 8*, ya que éste es un indicador de hacia dónde va a ir a línea, porque si el centroide está situado a izquierda o derecha de la imagen y no centrado quiere decir que hay tramos de línea cercanos que producirán giros.

3.4. Comparativa y análisis

En esta sección se han explicado los tres tipos de navegación que se han generado y usado en este trabajo, por lo que ahora y tras realizar las simulaciones y haber comprobado los puntos fuertes y débiles de cada una, se procederá a compararlas entre ellas y a sacar las ventajas y desventajas a la hora de poder utilizar una u otra navegación.

	Navegación basada en centroides y a velocidad lineal no constante	**Navegación basada en centroides y a velocidad lineal constante**	**Navegación con orientación de línea prioritaria**
Fortalezas	• Siempre orientado respecto a objetivo • Buena evasión de obstáculos	• Tiempo de simulación medio • Siempre situado en la línea • Velocidades lineales medias • Facilidad evasión en todo tipo de situaciones	• Tiempos de simulación bajos • Velocidades lineales altas • Velocidades angulares bajas • Seguimiento correcto de la línea • Facilidad con giro >70º
Debilidades	• Tiempos de simulación largos • Pérdida de línea recurrente • Velocidades muy lentas cerca del objetivo • Muchas esperas de puntos	• Posibilidad de velocidades angulares altas • Dificultad en giros >70º • Pérdidas ocasionales de la línea	• Dificultad de evasión de obstáculos cercanos a giros

Tabla 1: Fortalezas y debilidades de las navegaciones.

A través de la información de *Tabla 1* podemos comparar cada una de las navegaciones y sacar las siguientes conclusiones basándose en las ventajas y desventajas que pueden ofrecer cada una.

Estas conclusiones son, que la Navegación con orientación prioritaria es la navegación más viable debido a que presenta la ventaja con respecto al resto de que puede trabajar

con menos tiempo. Gracias a que las velocidades lineales pueden ser altas y, por lo tanto, favorecen esta reducción de tiempos, algo que no se da en ninguna de las otras navegaciones, pese a que en la Navegación basada en centroides y a velocidad lineal constante se tienen velocidades lineales medias.
Además, en esta navegación tenemos velocidades angulares bajas con lo que se tiene la ventaja con respecto a la Navegación basada en centroides y a velocidad lineal constante de que no se tendrán posibles inestabilidades en los giros o dificultades en ángulos radicales, puesto que al trabajar con velocidades bajas se reduce la posibilidad de que el robot deslice o gire más de la cuenta pudiendo perder el control.
Sin embargo, y como se ha comprobado en simulación, esta navegación puede presentar dificultades en la evasión de obstáculos dependiendo de dónde estén situados dichos obstáculos, algo que no ocurre en ninguna de las anteriores.

La segunda navegación que más viabilidad tendría y que para casos en los que no se tenga la posibilidad de trabajar con velocidades lineales en y, sería la Navegación basada en centroides y a velocidad lineal constante, puesto que a pesar de las desventajas que presenta con la anterior, proporciona tiempos en navegación no excesivamente altos, además esta navegación presenta la ventaja de poder esquivar cualquier tipo de obstáculo sin importar donde se encuentre este con mejor resultado que la navegación prioritaria. La mayor desventaja que puede presentar es que puede llegar a perder la línea si se encuentra ante ángulos radicales (>70º) para esta navegación y también que se pueden producir velocidades angulares altas que generen deslizamientos en el robot.

La opción que no sería recomendable utilizar sería la primera navegación que se generó, Navegación basada en centroides y a velocidad lineal no constante, pues no presenta grandes ventajas con respecto al resto, únicamente que presenta una buena evasión de obstáculos por estar siempre posicionado bien en los objetivos. Sin embargo, con respecto al resto de características únicamente presenta desventajas, sobre todo en cuanto al tiempo que tarda en realizar las trayectorias con respecto a las otras dos, ya que este es muy elevado y conlleva una pérdida de tiempo muy grande para los resultados que proporciona porque suele perder la línea, lo que representa otra desventaja en comparación a los otros dos que se mantienen siempre en la línea. Es por ello que se descartó esta navegación y por lo que no se volverá a nombrar.

4. Desarrollo del sistema de navegación

En esta sección se explican cambios necesarios para el desarrollo del sistema de navegación del trabajo en archivos base de Campero. Además de explicarse tanto el desarrollo de la percepción de la línea como de las estrategias de navegación más en profundidad.
Por otra parte, al final de la sección se incluye la relación entre los archivos ejecutables creados que dan lugar a las estrategias de navegación.

4.1. Sistema de percepción de línea

En este apartado se lleva a cabo la explicación de todos los pasos realizados para tratar la imagen que se obtiene de la cámara, aparte de explicar cómo se obtiene toda la información relevante y fundamental de estas imágenes. Por otro lado, se aporta una explicación de los cambios que hubo que efectuar en la cámara para conseguir una correcta visualización de la línea.

-Lo primero que se comentará es la modificación que se realizó en la cámara, dicha modificación se realizó para mejorar la visión del carril y, por lo tanto, obtener mejor información. Para ello se realiza un cambio en la orientación del eje de recepción de los *frames* de la Cámara.

La modificación que se realizó fue la sustitución de estas líneas de código del archivo: axis_m5525.urdf.xacro:

```
<joint name="${prefix}_pan_joint" type="revolute">
    <axis xyz="1 0 0"/>
    <origin xyz="0.0 0.0 0.087" rpy="0 0 0"/>

<joint name="${prefix}_tilt_joint" type="revolute">
    <axis xyz="0 -1 0"/>
    <origin xyz="0.0 0.0 0.0" rpy="0 ${-PI/2} 0"/>
```

Por estas otras:

```
<joint name="${prefix}_pan_joint" type="revolute">
    <axis xyz="1 0 0"/>
```

```
    <origin xyz="0.0 0.0 0.087" rpy="${PI/30} ${-PI/9} 0"/>


<joint name="${prefix}_tilt_joint" type="revolute">
    <axis xyz="0 -1 0"/>
    <origin xyz="0.0 0.0 0.0" rpy="0 ${-PI/4} 0"/>
```

Obteniendo una imagen de la cámara mucho más cercana al suelo con lo que proporciona una información más precisa de la línea que se tiene cerca, reduciendo el error generado en el seguimiento por no ver la línea que está cerca del robot. Además de esta forma se consigue tener más puntos con los que navegar, debido a que las distancias de posicionamiento de estos en la realidad son más cercanas a nuestro robot, con lo que en una misma línea se obtienen más puntos. Debido a que se verá, como se explicará a continuación, mucho más tiempo la línea de lo que la se vería con la cámara por defecto donde se pierde antes de vista.

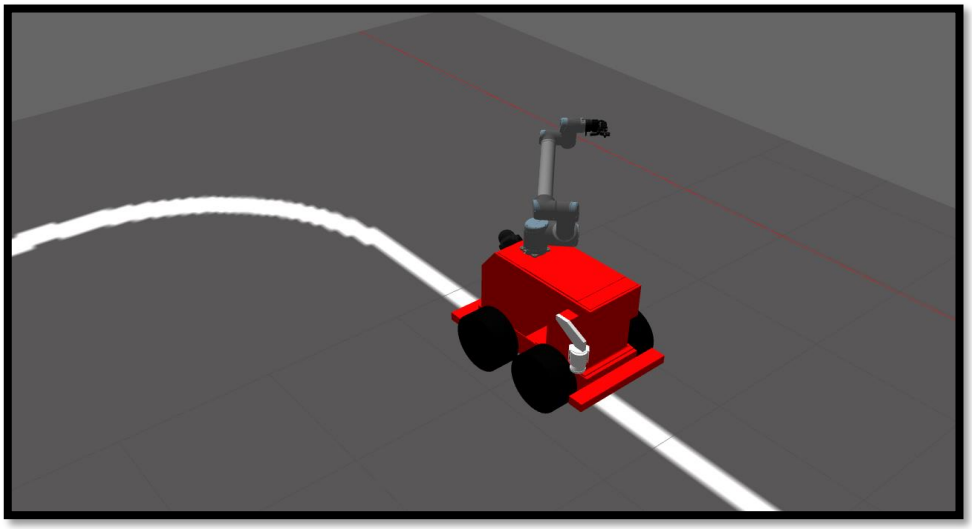


Figura 10: Ejemplo Gazebo, Campero frente a línea.

En la *Figura 10* se puede observar un ejemplo de cuando el robot esta frente a la línea y tiene en un futuro cercano que realizar un giro, en la *Figura 11* lo que se encuentra es la imagen que obtiene la cámara para esa posición del robot, pero en las dos configuraciones, una con la cámara por defecto y otra con la cámara bajada.

Con estas figuras se puede explicar la decisión de moverla hacia abajo, ya que al hacer esto se permite que el robot no pierda la información de la línea que está más próxima a él. Aparte y aunque parezca que se pierde información, ya que con la cámara normal

se ve más línea esto no es así, puesto que cuanto más nos aproximamos al giro más información se pierde si la cámara se mantiene por defecto y menos si se baja la cámara. Además de observarse la línea más cercana al robot en vez de visualizar información futura.

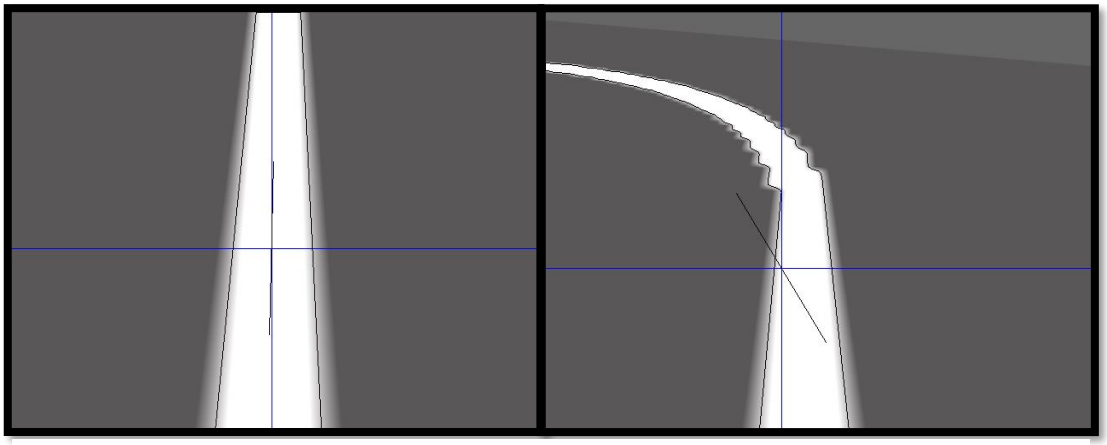


Figura 11: Imágenes captadas con la cámara normal (Derecha) y con la cámara hacia abajo (Izquierda)

Una vez se ha explicado el cambio realizado en la cámara, se procede a explicar el ejecutable **Line_tracker.py**, el cual trabaja con ella. Pero antes de comenzar la explicación se tiene que comentar que los códigos desarrollados en este trabajo han sido creados de cero y únicamente se han introducido ciertas líneas, que se mencionarán, pertenecientes a códigos ajenos. Como en el ejecutable que se va a explicar a continuación donde se utilizaron las siguientes líneas:

```
[vx,vy,x,y] = cv2.fitLine(c,cv2.DIST_L2,0,0.01,0.01)
angle = math.atan2(vy, vx)
if angle < 0:
   angle = angle + math.pi / 2
else:
   angle = angle - math.pi / 2
angle *= -1
```

Dichas líneas pertenecen a dudasdavid y se encuentran en GitHub en el enlace [8], las cuales se usaron porque facilitaban el cálculo de la orientación de la línea.

De la misma forma se ha de mencionar que al comienzo del proyecto se revisó el código de khaledgabr77 que se encuentra en GitHub en el enlace [10], para obtener una perspectiva de como comenzar a trabajar con OpenCV(1)[11], no obstante, de su código no se empleó nada salvo la organización que él empleaba para el trabajo con las imágenes,

(1) *Open Computer Vision*, biblioteca de código abierto que contiene implementaciones que abarcan más de 2500 algoritmos y está especializada en el sistema de visión artificial y machine learning.

aunque conforme se avanzó en el trabajo esta organización cambio totalmente hasta llegar a la actual.

En este ejecutable lo que principalmente se hace es generar información, como es la posición y orientación de puntos de la línea que detecte, para posteriormente situarlos en el entorno real, gracias a la información que obtiene de las imágenes captadas por la cámara. Además de informar al ejecutable Line_follower.py de la posición del centroide, en la imagen de la línea detectada para que pueda trabajar con ello.
Todo esto se realiza de la siguiente manera:

-En primer lugar, se obtiene de la imagen que proporciona el topic ('/campero/campero_front_ptz_camera/image_raw'), las dimensiones y si estas no son las adecuadas se redimensiona la imagen por interpolación de áreas de pixeles.
Tras efectuar esto se prepara la imagen para su utilización, para esto se realiza un desenfoque gaussiano, que permite eliminar el ruido de alta frecuencia (pixeles cambiando muy rápido) en las imágenes que se reciben, mediante el método de OpenCV cv2.GaussianBlur.
Seguidamente, se aplica una máscara con los umbrales máximos y mínimos para los colores que se utilicen, tras obtener la máscara se hace una erosión y dilatación de la imagen para eliminar ruido de la máscara, además de aislar elementos individuales y unir elementos dispares de la imagen.
Todo esto se realiza mediante las siguientes líneas:

threshImg = cv2.inRange(blurImg, threshold_value_White[0], threshold_value_White[1])
mask = cv2.erode(threshImg, None, iterations=2)
mask = cv2.dilate(mask, None, iterations=2)

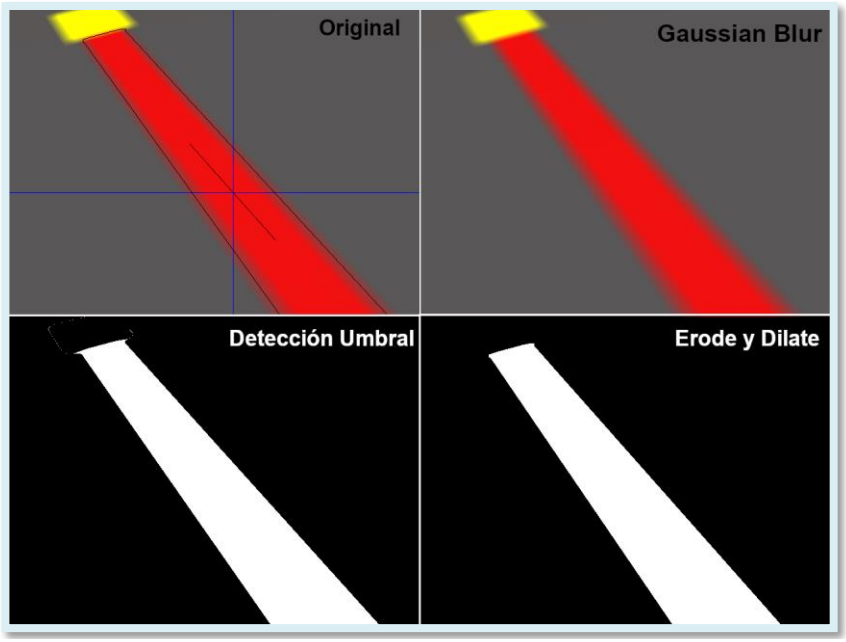


Figura 12: Pasos para la obtención de la máscara, de izquierda a derecha y de arriba abajo.

En la *Figura 12*, se tienen los pasos que se han explicado anteriormente, representados con las imágenes que se les corresponden.

Una vez se tiene una máscara aceptable, se procede a trabajar con ella y lo primero que se hace es encontrar los contornos en la máscara que estén dentro del umbral, mediante la siguiente línea de código.

```
_, contours, hierarchy = cv2.findContours(mask.copy(), 1, cv2.CHAIN_APPROX_NONE)
```

Con cv2.CHAIN_APPROX_NONE se almacenan todos los puntos que se detectan de los contornos.

Tras detectar al menos un contorno en la máscara de la imagen lo que se hará será obtener el mayor de los contornos si hubiera varios y de éste obtener su área, todo ello mediante:

```
c = max(contours, key=cv2.contourArea)
area = cv2.contourArea(c)
```

Gracias al área se podrá decidir si lo que se ve es la línea que se quiere o simplemente es una marca no deseada. Si el área es suficiente, algo que se ha puesto en como mínimo un valor de 6000 pixeles se procederá a marcar que la línea ha sido detectada, con lo que se avisa al resto de programas, a la vez que se calcularán los momentos del contorno, dichos momentos son un promedio

ponderado particular de las intensidades de píxeles de la imagen, con los cuales se podrán encontrar ciertas propiedades específicas de una imagen como la que se utilizará que es el centroide.

Los momentos y el centroide se obtienen a través de estas líneas:

```
M = cv2.moments(c)
cx = int(M['m10'] / M['m00'])
cy = int(M['m01'] / M['m00'])
```

Además del centroide también se calcula el ángulo con el que se orienta la línea en la máscara en función de un punto normalizado, que se obtiene al realizar el ajuste de mínimos cuadrados de todos los puntos del contorno.

Este punto normalizado se obtiene a partir de la función,

```
[vx,vy,x,y] = cv2.fitLine(c,cv2.DIST_L2,0,0.01,0.01)
```

El algoritmo se basa en la técnica *M-estimator* que ajusta iterativamente la línea utilizando el algoritmo de mínimos cuadrados ponderados. Después de cada iteración, los pesos w_i se ajustan para que sean inversamente proporcionales a $\rho(r_i)$, que es una función de la distancia que en nuestro caso es Cv2.DIST_L2 siendo el método más rápido y simple de ajuste de mínimos cuadrados, y que se corresponde con $\rho(r) = r^2/2$.

Una vez obtenido el punto normalizado se procede a obtener el ángulo por medio del arco tangente de vy y vx.

Una vez se ha obtenido de la imagen la información fundamental, de qué punto se quiere situar y con qué orientación como se observa en la *Figura 13*, se procederá a crear un *goal* que se publicará posteriormente y con el que trabajará en Line_marker.py. Este punto objetivo, referenciado siempre desde el robot, tendrá como posición en x la altura de la imagen menos la posición y del centroide, y como posición en y se han diferenciado los valores positivos y negativos, debido a que el sistema de referencia está centrado, por medio de la resta de la mitad de la anchura de la imagen menos la posición x del centroide.

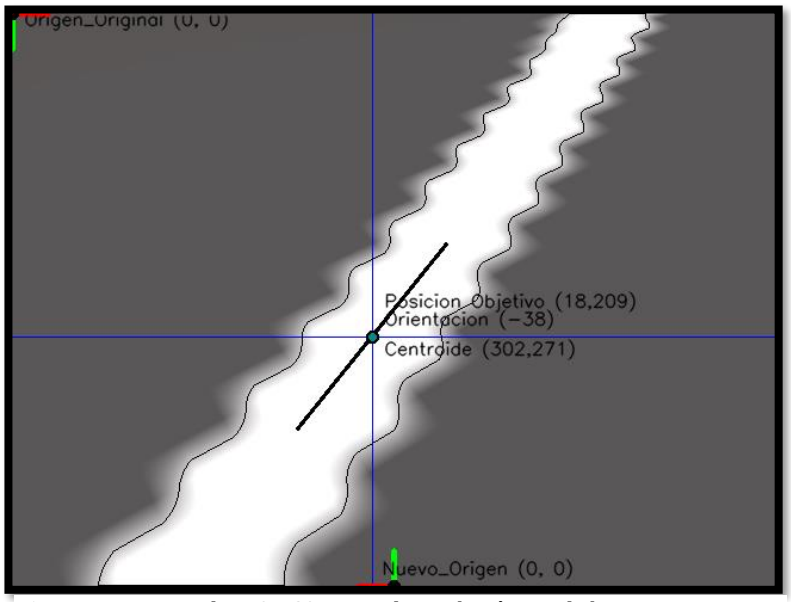


Figura 13: Imagen de 640x480p captada por la cámara de la marca con su información.

Siendo w y h, la anchura y altura de la imagen respectivamente y cx y cy la posición del centroide, la posición del punto objetivo o *goal*, representado en la *Figura 13,* se calcula así:

position.x = h - cy

position.y = (w / 2) – cx

Si se obtuviera que el área del contorno de la línea no es suficiente como para asegurar que es la línea objetivo, se procede a comprobar si, en el caso de que se hubiera detectado un punto final de la misma forma que se hacía con la línea, este cumple igual que antes los requisitos de área que se han marcado en este caso el mismo valor que previamente. En el caso de que se confirmase que hay un punto final que cumpliera con el tamaño de área, se generará un punto objetivo o *goal* del mismo modo que antes, pero con la diferencia de que no habrá variación en la orientación manteniendo la que tuviese el robot.

Tras todos estos procesos se publicarán los diferentes datos en sus respectivos topics.

4.2. Estrategias de navegación

Después de haber explicado cómo se consigue la información de la cámara, la cual es fundamental para el desarrollo de nuestra estrategia de navegación. Puesto que gracias a esta información se han desarrollado los dos tipos de navegación que se explicarán a continuación.

Antes de proceder a la explicación de las estrategias de navegación se debe hacer un inciso para explicar otro de los ejecutables, puesto que es a partir de éste con el que se trabaja en uno de los tipos de navegación. Además de ayudar en Rviz a visualizar la línea detectada.

El ejecutable del cual se habla es **Line_marker.py**, en este ejecutable lo que se hace es generar las referencias o marcas que vemos en Rviz de los puntos objetivos, como en la *Figura 14*, mediante las cuales el código Line_follower_Marks.py, del cual se habla posteriormente, puede realizar su navegación.

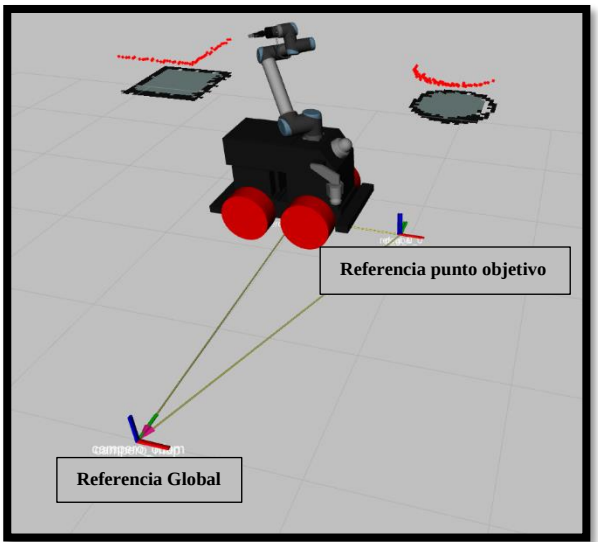


Figura 14: Representación de las referencias creadas por Line_marker.py, en Rviz

La forma de proceder en este código es generar una localización relativa con respecto a la base del robot, de los puntos que proporciona Line_tracker.py, debido a que los puntos que proporciona la cámara están situados siempre con respecto a esta localización. Posteriormente, se realiza una transformación para obtener la localización en función del sistema de referencia absoluto, que en nuestro caso es el *campero_odom* y poder trabajar tanto las marcas como el robot con localizaciones con referencia absoluta.

Para realizar todo esto lo que se hace es a través de la información recibida de Line_tracker.py, se crea una primera transformada referenciada al robot, ya sea para la línea o para el final, mediante el código:

```
MP.transformStamped.header.stamp = rospy.Time.now()
MP.transformStamped.header.frame_id = MP.frame_array[0]
MP.transformStamped.child_frame_id = MP.frame_array[MP.id]

MP.transformStamped.transform.translation.x = msg.position.x / MP.rescale_x
MP.transformStamped.transform.translation.y = msg.position.y / MP.rescale_y
MP.transformStamped.transform.translation.z = msg.position.z
MP.transformStamped.transform.rotation.x = msg.orientation.x
MP.transformStamped.transform.rotation.y = msg.orientation.y
MP.transformStamped.transform.rotation.z = msg.orientation.z
MP.transformStamped.transform.rotation.w = msg.orientation.w
```

Donde se hace uso del tipo de dato transformStamped para crear una transformación desde un eje de coordenadas(header), en este caso se trata de campero_base_footprint, a otro eje(child_frame) que en nuestro caso será el que se genere y que se llamará goal_i, siendo i el número del eje que corresponda. La translación que se realiza es la posición que calculábamos antes en el otro ejecutable, lo único que se hace es transformar esta posición para tenerla en metros y no en píxeles, esto se hace dividiendo en x e y por un valor calculado e igual a: rescale_x = 333.0 / 1.0 y rescale_y = 320.0 / 0.5, siendo 333p el tamaño en el eje y de la imagen que corresponde con 1 metro en la realidad, algo que se obtuvo mediante calibración. Al igual que 320p se corresponde con la mitad del tamaño de la imagen en x, que es 640p, y tras la calibración se obtuvo que se puede visualizar aproximadamente 0.5 metros reales, tanto positiva y negativamente desde la nueva referencia que vemos en la *Figura 13*. Estos valores deberán ser ajustados si se utiliza una cámara distinta o si se utiliza otra resolución. Además, la orientación es la misma que se obtenía.

Tras generar esta transformación se publicará y, por lo tanto, se creará el eje mediante la línea:

```
br2.sendTransform(MP.transformStamped)
```

En este trabajo, dicha función es la perteneciente a tf2_ros.StaticTransformBroadcaster() que es la clase que proporciona la manera de publicar estas nuevas transformaciones, estas son estáticas y no necesitan ser refrescadas continuamente para mantenerse, lo que facilita su uso. Existe la clase tf2_ros.TransformBroadcaster() cuyas publicaciones sí que necesitan ser refrescadas continuamente y que no se utilizó en este trabajo.

Una vez se consigue generar la primera transformación lo que se hace es esperar a que esta esté publicada y obtener la transformación al eje que se genera(goal_i) desde el sistema de referencia absoluto(campero_odom), obteniendo tanto la translación como la orientación, todo esto mediante:

```
MP.listn.waitForTransform('/campero_odom', MP.frame_array[MP.id], rospy.Time(),
rospy.Duration(10))
(T, R) = MP.listn.lookupTransform('/campero_odom', MP.frame_array[MP.id], rospy.Time())
```

Una vez se obtienen la nueva posición y orientación se crea nuevamente una transformación como antes, pero esta vez desde el sistema de referencia absoluto, además de publicar de nuevo la transformación, de la siguiente forma:

```
MP.transformStamped_ref.header.stamp = rospy.Time.now()
MP.transformStamped_ref.header.frame_id = MP.frame_ref_array[0]
MP.transformStamped_ref.child_frame_id = MP.frame_ref_array[MP.id]

MP.transformStamped_ref.transform.translation.x = T[0]
MP.transformStamped_ref.transform.translation.y = T[1]
MP.transformStamped_ref.transform.translation.z = T[2]
MP.transformStamped_ref.transform.rotation.x = R[0]
MP.transformStamped_ref.transform.rotation.y = R[1]
MP.transformStamped_ref.transform.rotation.z = R[2]
MP.transformStamped_ref.transform.rotation.w = R[3]

br2.sendTransform(MP.transformStamped_ref)
```

A continuación, se habla de los ejecutables **Line_follower.py** y **Line_follower_Marks.py**, los cuales representan las dos estrategias navegación de las cuales se han hablado en el apartado 3 y que han resultado ser las más viables.

Los ejecutables Line_follower.py y Line_follower_Marks.py son los utilizados para generar el movimiento del Campero basándose en la información que les proporcionan Line_tracker.py y Line_Marker.py. Aunque ambos contienen la misma estructura la cual se explica a continuación, se puede encontrar una diferencia en la manera en la que se produce la navegación, pues cada una trabaja con un tipo de información distinta para realizarla.

Antes de proceder con la explicación de la estructura básica de los ejecutables se explicará que la forma en que se realiza la navegación, o sea el movimiento por la línea, pertenece a dudasdavid y se encuentra en GitHub en el enlace [9], las líneas que se utilizan para desarrollar esta navegación son las siguientes:

```
vel_msg.angular.z = P_angle * (angle) + P_linear * -(xval - xmax/2)
vel_msg.linear.y = P_linear * -(xval - xmax / 2)
```

De estas líneas de código que proporcionan el movimiento se adaptaron las constantes de los controles proporcionales para obtener los resultados que se requerían.

Una vez comentado esto se puede explicar la estructura básica de los ejecutables desarrollados para las estrategias de navegación es la siguiente:

1º. Se comienza con una navegación de comienzo, la cual sirve para posicionarse encima de la línea. Una vez se tiene la certeza de que se está en la línea, algo que se sabe gracias a la comunicación entre este ejecutable y Line_tracker.py por medio del topic, ***'/campero/in_line'***.

   A través de las líneas:

```
if follower.Start:
  Start_program()
  if follower.in_line == 1 and i < 1:
    goal_arr_pub.publish(True)
    i += 1
# rospy.loginfo("Comienzo")
```

   Donde la condición que contiene nos permite iniciar la creación de marcas en la simulación a partir de Line_marker.py.

   El código de la función Start_program(), se basa en una comunicación continua con Line_tracker.py debido a que este indica en qué situación se encuentra campero con respecto al posicionamiento sobre la línea. Ya que se indican los pasos a seguir a través del topic mencionado anteriormente, siendo tres los pasos que se deberán seguir.

   - El primero de ellos asociado al número 0, le indica a este código que debe realizar un giro a derechas para encontrar la línea.

```
if follower.in_line == 0:
    rospy.loginfo("0")
    follower.vel.angular.z = -0.5
```

   - Una vez Line_tracker.py confirma que se ha encontrado la línea publica un 1 en el topic, con lo que este código comienza el segundo paso que es realizar una navegación hacia una marca, por medio del código de navegación por marcas, que se explicará más adelante. Una vez la navegación termina

correctamente esta devuelve que ha llegado al objetivo con lo que se publica en el topic ***'/campero/positioned'***, que se está en posición y se puede pasar al siguiente paso.

```
elif follower.in_line == 1:
    rospy.loginfo("1")
    follower.vel.linear.x = 0.0
    follower.vel.angular.z = 0.0
    if follower.nav:
        navigation()
    elif follower.step_end:
        follower.vel.linear.x = 0.0
        positioned_in_line_pub.publish(True)
```

- El último paso que se realiza es iniciar el resto del programa a través de la habilitación de nuevo de la creación de marcas, y gracias la respuesta de Line_tracker.py de que el programa se ha iniciado y, por lo tanto, se hace falso que se continúe con el código de comienzo.

```
elif follower.in_line == 2:
    # rospy.loginfo("2")
    follower.vel.angular.z = 0.0
    goal_arr_pub.publish(True)
    cmd_vel_pub.publish(follower.vel)

def Has_started_callback(self,msg):
    self.Start = False
```

2º. Se evalúa constantemente si se produce alguna de las siguientes condiciones:

- Si se ha detectado un obstáculo y éste no ha sido mientras se buscaba la línea.
- Si se ha encontrado un punto final y hay que navegar a él.
- Si hemos perdido la línea y se ha de buscar.
- Si no se produce ninguna de las anteriores quiere decir que se puede continuar navegando.

Estas condiciones las encontramos en las siguientes líneas, donde únicamente diferenciaríamos el tipo de navegación:

```
 if follower.obstacle and not follower.search:
   rospy.loginfo("Obstaculo detectado")
   obstacle_avoidance()
elif follower.end_nav:
   # rospy.loginfo("Navegando al punto final")
   end_nav()
elif follower.search:
   rospy.loginfo("Buscando la linea")
   search_line()
elif not follower.obstacle and not follower.end_nav and not follower.search:
   navigation ()
   goal_arr_pub.publish(True)
   searching_line_pub.publish(False)
```

Y para cada una de las condiciones se ha generado un código, el cual se explica en los siguientes apartados.

4.3. Evitación de obstáculos

En este punto se realiza la explicación de como las estrategias de navegación son capaces de realizar la evasión de obstáculos. Aunque como sucede en el punto de la percepción de la línea, se debe explicar previamente una modificación en los archivos para poder efectuar correctamente la evasión y así alcanzar los objetivos del trabajo.

-La modificación que se realizó fue modificar la distancia mínima al obstáculo desde el sensor en el archivo teb_local_planner_omni_params.yaml, que utiliza move_base al ejecutarse, para evitar que se chocase con facilidad a obstáculos con distintas formas.

La línea de código modificada en este archivo es:

min_obstacle_dist: 0.435 # minimum distance to obstacle: it depends on the footprint_model

Por defecto el valor que nos daban era de 0.435 metros y la evasión de obstáculos con forma cilíndrica era correcta la mayor parte del tiempo, pero en obstáculos con forma cúbica se producían choques o desplazamientos de estos obstáculos siempre, algo que era imperioso evitar. Algo que se puede ver en la *Figura 15*, donde se puede comprobar que, si el obstáculo es cilíndrico, con radio 0.5 metros, es capaz de esquivarlo, pero pasando muy cerca de él, y cuando el obstáculo es con esquinas no redondeadas, de 1 metro de lado, se choca con él.

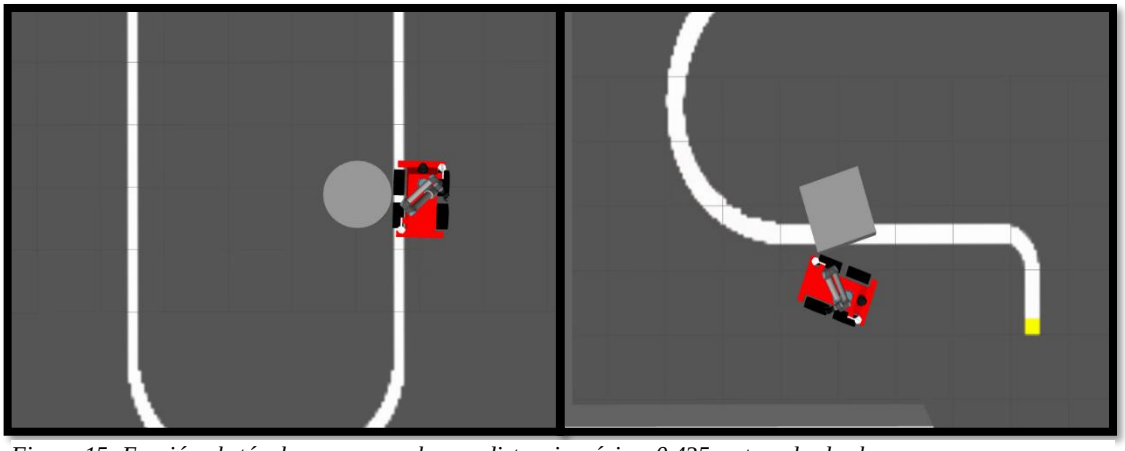


Figura 15: Evasión obstáculos con move_base y distancia mínima 0.435 metros desde el sensor.

Por ello se sustituyó este valor por 0.7 metros y se comprobó visualmente que con dicho valor se incurría en una trayectoria menos problemática y permitía una evasión de obstáculos más segura como vemos en la *Figura 16*.

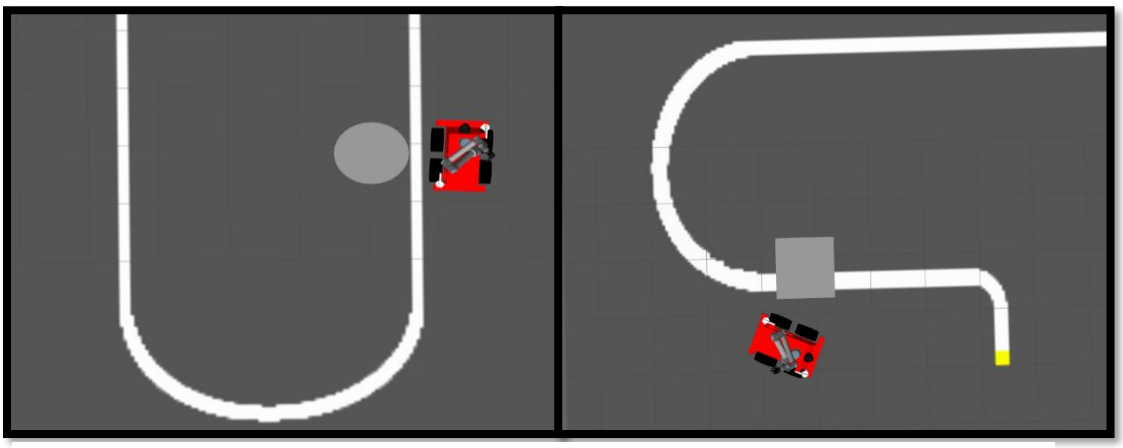


Figura 16: Evasión obstáculos con move_base y distancia mínima 0.7 metros desde el sensor.

Como se comprueba al aumentar la distancia mínima el robot pasa por posiciones más alejadas de los obstáculos, evitando el choque contra éstos, algo que es fundamental en nuestra aplicación.

Una vez se ha aclarado este cambio se puede explicar cómo se trabaja en las estrategias de navegación la evasión de obstáculos, en ambas estrategias se utiliza el mismo código para la evasión.

Lo primero que se ha de explicar es que para la detección de obstáculos se evalúan los datos que se reciben del sensor láser frontal, diferenciando en este, tres zonas puesto que dicho sensor no está situado en la parte central delantera del robot, sino que está situado en la parte derecha delantera, con lo que tuvimos que tomar tres distancias mínimas de detección una para cada área.

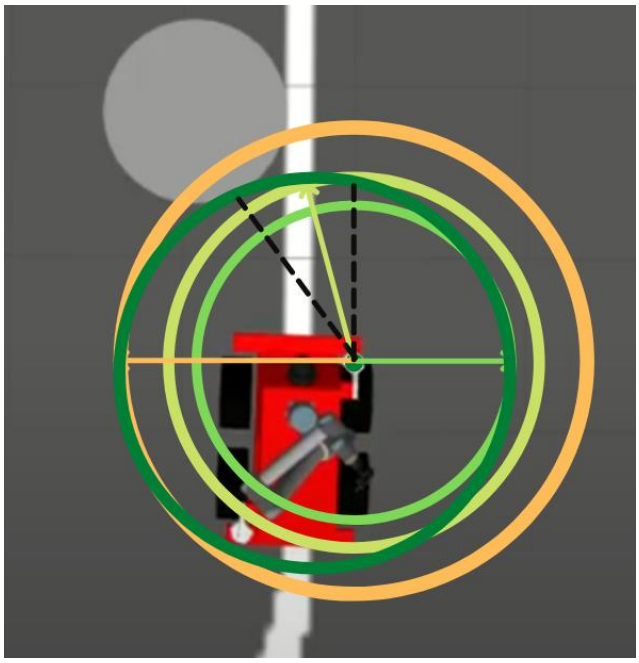


Figura 17: Área de la evasión de obstáculos.

Como se observa en la *Figura 17* lo que se hizo es diferenciar 3 radios distintos del sensor, verde claro, lima y naranja, cuyas distancias son 0.7 metros, 0.9 metros y 1.1 metros, y a su vez determinar cuánto rango se tendría en cuenta de estas áreas barridas algo que se muestra con las líneas a puntos de color negro. Quedándose un área barrida aproximada, que es la que se ve con color verde oscuro y que se encuentra más centrada en el robot lo que facilita encontrar obstáculos delante de él a la misma distancia en todos los ángulos.

Tras esta explicación se puede explicar qué se realiza cuando se detecta un obstáculo, y que se divide en tres fases:

-La primera fase es la detención del Campero para evitar la colisión con el obstáculo.

```
if follower.fase == 0:
        # rospy.loginfo("Obstacle detected")
        follower.vel.linear.x = 0.0
        follower.vel.linear.y = 0.0
        follower.vel.linear.z = 0.0
        follower.vel.angular.x = 0.0
        follower.vel.angular.y = 0.0
```

```
follower.vel.angular.z = 0.0
cmd_vel_pub.publish(follower.vel)
follower.fase = 1
```

-La segunda fase es reorientar al campero a la última orientación del punto objetivo, por si no se hubiera producido, para estar lo más recto posible en la línea, y poder afrontar el obstáculo recto frente a él. Por lo cual se rota hasta estar en la orientación adecuada.

```
elif follower.fase == 1:
        follower.listn.waitForTransform('/ref_goal_%s' % follower.id,
        '/campero_base_footprint', rospy.Time(), rospy.Duration(25))
        (trans, rot) = follower.listn.lookupTransform('/campero_base_footprint',
        '/ref_goal_%s' % follower.id, rospy.Time(0))
        euler = follower.quat_to_euler(rot)
        if abs(euler[2]) > 0.01:
                # rospy.loginfo("Estoy rotando")
                if euler[2] > 0:
                        follower.vel.angular.z = 0.5
                else:
                        follower.vel.angular.z = -0.5
                follower.vel.linear.x = 0.0
                cmd_vel_pub.publish(follower.vel)
        else:
                follower.vel.linear.x = 0.0
                follower.vel.angular.z = 0.0
                cmd_vel_pub.publish(follower.vel)
                follower.fase = 2
                follower.i = 0
```

-La tercera fase es llamar a la función ***move_base_client()*** y que esta realice un movimiento hasta una posición que se encuentre 3.5 metros delante de la posición actual del Campero, se eligió esta distancia porque representaba aproximadamente tres veces la longitud del Campero, lo que proporciona cierta holgura con respecto a los obstáculos con los que se pueda encontrar y no pasarnos la línea que este posteriormente, esta función generará un movimiento autónomo gracias al paquete ***move_base***, el cual permite la evasión del obstáculo automáticamente. Tras la realización de esta navegación el propio paquete devuelve una confirmación de que se ha podido realizar y que se ha finalizado el movimiento con éxito tras esto lo que se hace es continuar con la navegación normal.

```
follower.aux_position = [follower.position.x, follower.position.y, follower.position.z]
follower.aux_orientation = [follower.orientation.x, follower.orientation.y,
                        follower.orientation.z, follower.orientation.w]
elif follower.fase == 2:
        result = movebase_client()
```

```
        if result:
                # rospy.loginfo("Goal execution done!")
                follower.fase = 0
                follower.obstacle = False
                goal_arr_pub.publish(True)


def movebase_client():
        client = actionlib.SimpleActionClient('/campero/move_base',MoveBaseAction)
        client.wait_for_server()
        goal = MoveBaseGoal()
        goal.target_pose.header.frame_id = "campero_map"
        goal.target_pose.header.stamp = rospy.Time.now()
        ori_euler = follower.quat_to_euler(follower.aux_orientation)
        goal.target_pose.pose.position.x = 3.5 * math.cos(ori_euler[2]) + follower.position.x
        goal.target_pose.pose.position.y = 3.5 * math.sin(ori_euler[2]) + follower.position.y
        goal.target_pose.pose.orientation.x = follower.aux_orientation[0]
        goal.target_pose.pose.orientation.y = follower.aux_orientation[1]
        goal.target_pose.pose.orientation.z = follower.aux_orientation[2]
        goal.target_pose.pose.orientation.w = follower.aux_orientation[3]
        client.send_goal(goal)
        wait = client.wait_for_result()
        if not wait:
                rospy.logerr("Action server not available!")
                rospy.signal_shutdown("Action server not available!")
        else:
                return client.get_result()
```

4.4. Estrategia de búsqueda de línea

Ahora se procederá a explicar la estrategia para la búsqueda de la línea, cuando el robot la pierde de vista y necesita volver a ella o encontrarla simplemente.

El código utilizado para la búsqueda de la línea es el que se encuentra en la función *search_line()* presente en ambas estrategias de navegación, lo que se realiza es un giro 90º a izquierdas de la posición actual del robot la cual en el momento que Line_tracker.py indica que no la encuentra se guarda, si cuando el robot está girando la detecta, se establece como línea encontrada y prosigue la navegación. En caso de que la línea no sea encontrada tras realizar este giro, se gira 180º a derechas para mirar si la línea estuviera a la derecha de la primera posición, y se realiza lo mismo que en el caso anterior.

Por último, si no se hubiera encontrado la línea todavía se continúa girando 90º más, puesto que querría decir que no hay línea en el camino que se estaba siguiendo y se debe volver por sus mismos pasos.

En la *Figura 18,* está el diagrama de flujo de la explicación anterior sobre cómo funciona la función search_line(), para facilitar la explicación.

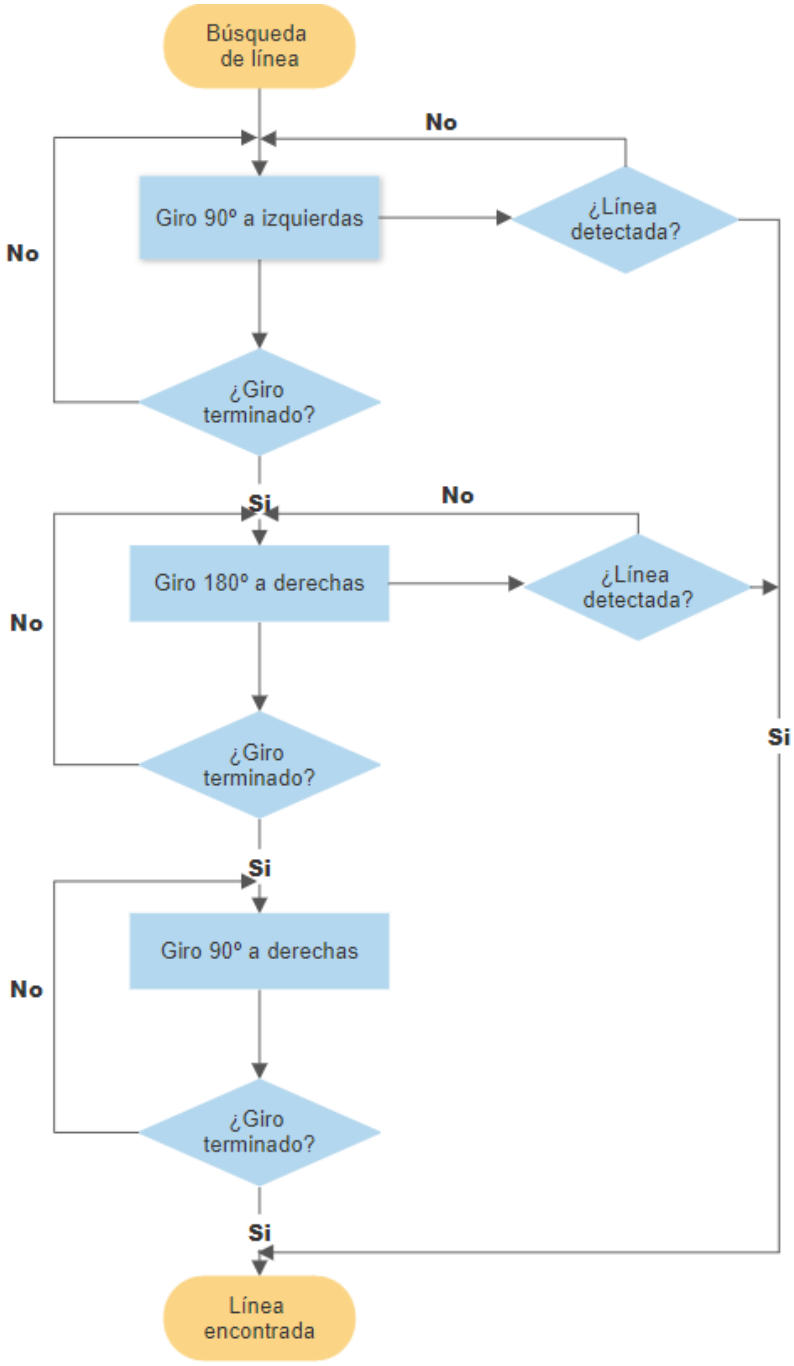


Figura 18: Diagrama de flujo de la búsqueda de línea.

4.5. Seguimiento de línea

En este punto se van a comentar los tipos de navegación comentados en la sección anterior, a excepción del primero el cual como ya se ha explicado fue descartado, con lo cual en este apartado se analiza su código, además explicarse que sucede si se encuentra con una marca final y como se navega a esta marca.

Por lo tanto, se comenzará explicando el código de navegación para una marca asociada a un final, esta navegación es por marcas, que recordemos que son las referencias que se crean de la línea que se sigue. Sin embargo, en este caso la marca no será la línea sino un punto de un color distinto al que se le asociará un punto objetivo y se navegará hasta él, hasta estar a una cierta distancia aceptable.
Tras conseguir llegar a este punto se publica que se ha llegado al final del recorrido para que finalicen el resto de los programas ejecutados.

```
def end_nav():
    rospy.loginfo("Final de la navegacion")
    follower.listn.waitForTransform('/end_goal_0', '/campero_base_footprint',
                                    rospy.Time(),rospy.Duration(10))
    (trans, rot) = follower.listn.lookupTransform('/campero_base_footprint', '/end_goal_0', rospy.Time(0))

    distancia = follower.euclidean_distance(trans)

  if distancia > 0.15:

     linear = 0.5
  else:
     # rospy.loginfo("Estoy en posicion")

     linear = 0.0
     follower.fin = True
     final_complete_pub.publish(follower.fin)

  follower.vel.linear.x = linear
  follower.vel.angular.z = 0.0
  cmd_vel_pub.publish(follower.vel)
```

Ahora se procederá a explicar los dos tipos de navegación, la asociada al seguimiento de marcas, similar a la navegación al punto final, y la asociada a la prioritaria a la línea.

-La primera navegación se basa en el cálculo de cuál es la distancia al punto objetivo y a partir de una velocidad lineal constante e igual a 0.35 m/s, elegido este valor para obtener arcos de circunferencia bajos con los cuales no se generen inestabilidades en los giros, aunque para ello se haya tenido que sacrificar la velocidad en línea recta. Se controla únicamente la velocidad angular por medio

de un control proporcional sobre el arco tangente de la componente x e y de la traslación a realizar, dicho arco tangente nos devuelve el ángulo necesario para estar orientado al punto objetivo. A partir del control proporcional se transforma dicho ángulo en la velocidad angular necesaria, para generar el radio de la circunferencia que junto a la velocidad lineal x nos lleve al punto objetivo.

Se evalúa si se está a una cierta distancia del objetivo y si es así se cambia de objetivo para que el Campero no pare en cada marca.

```
def navigation():
    # rospy.loginfo("No navegacion final")
    # rospy.loginfo("Estoy navegando")
    rospy.loginfo(follower.id)

    follower.listn.waitForTransform('/ref_goal_%s' % follower.id,
'/campero_base_footprint', rospy.Time(),
                                rospy.Duration(25))
    (trans, rot) = follower.listn.lookupTransform('/campero_base_footprint',
'/ref_goal_%s' % follower.id,
                                          rospy.Time(0))

    distancia = follower.euclidean_distance(trans)
    euler = follower.quat_to_euler(rot)

    linear = 0.35
    angular = 0.65 * math.atan2(trans[1], trans[0])

    if distancia > 0.05 and follower.Start:
        rospy.loginfo("Estoy en posicion de comienzo")
        linear = 0.0
        angular = 0.0
        follower.step_end = True
        follower.nav = False

    elif distancia > 0.2 and  not follower.Start:
        rospy.loginfo("Estoy en posicion de comienzo")
        follower.id = follower.id_aux

    follower.vel.linear.x = linear
    follower.vel.angular.z = angular

    cmd_vel_pub.publish(follower.vel)
```

-La segunda navegación, contiene un código más simple pues únicamente es dependiente de la información de la posición del centroide, la cual es mandada por Line_tracker.py.

En esta navegación se mantiene una velocidad lineal en x constante a 0.5 m/s, valor superior a la navegación anterior, puesto que en esta navegación los arcos son menores gracias a una cierta velocidad y utilizada reduce la velocidad angular.

Después se realiza un control proporcional sobre la velocidad lineal en y la cual permite posicionarse mejor en la línea, esta velocidad trabaja de forma que en función de si el centroide está a la derecha o izquierda del centro de la imagen, se da una velocidad a Campero para que corrija dicha posición y centre el centroide en la imagen.

Por último, se realiza nuevamente un control proporcional de la velocidad angular, a la que además se le resta un offset que es la velocidad lineal y, ya que esta también interviene en el movimiento angular reduciendo tanto el ángulo obtenido de la línea como, por lo tanto, la velocidad angular.

```
def navigation_Vcte():
    if (follower.angle and follower.xval) == 0.0:
        follower.vel.angular.z = 0.0
        follower.vel.linear.y = 0.0
    else:
        follower.vel.angular.z = P_angle * (follower.angle) + P_linear * -(follower.xval
- follower.xmax/2)
        follower.vel.linear.y = P_linear * -(follower.xval - follower.xmax / 2)
    follower.vel.linear.x = 0.5

    cmd_vel_pub.publish(follower.vel)
```

4.6. Relación de ejecutables mediante tópicos

La relación entre los ejecutables es una comunicación entre nodos a través de topics la cual se explicó con anterioridad y fue mostrada en un ejemplo en la Figura 5, en este caso la relación la vemos a través de la Figura 19, donde se tienen los tres archivos como nodos, y los distintos topics.

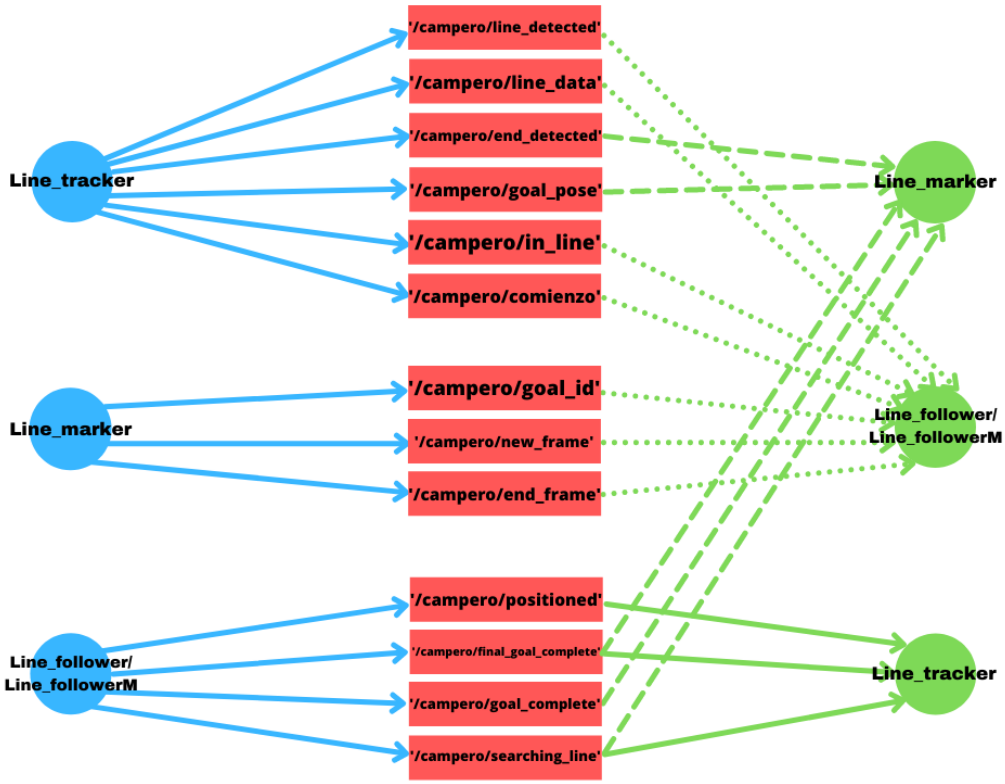


Figura 19: Relación entre los tres scripts, Line_tracker.py, Line_marker.py y Line_follower/Line_follower_Marks.py, por medio de topics

Como se ve en la imagen los nodos publicadores están representados en azul y los nodos suscriptores en verde, a su vez los topics están representados en rojo.

Las dos líneas de código que se utilizan recursivamente para realizar la comunicación de la que se ha hablado son las siguientes:

1ª rospy.Subscriber(A, B, C)

A: Dirección del topic

B: Formato del dato

C: Función en la que se depositará el dato

2ª Ejemplo_pub = rospy.Publisher(A, B, C)
Ejemplo_pub.publish(D)

A: Dirección del topic

B: Formato del dato

C: Tamaño de la cola de datos

D: Dato que se publica, con el formato requerido

Donde la primera línea de código sirve para suscribirnos a los topics deseados de los cuales se recibe información, y la otra para publicar en los topics la información.

5. Experimentos en simulación

Tras los análisis visuales y la detección de fortalezas y debilidades de las navegaciones, así como, de cuáles eran los más viables, se decidió que el primer tipo de navegación basado en el seguimiento de marcas a velocidad no constante se descartaría y se trabajaría con los otros dos.

En este punto se analizarán los resultados numéricos obtenidos en los distintos entornos desarrollados, los cuales son tres, para cada una de las dos navegaciones viables. Los datos que se analizaran serán las posiciones del Campero en cada instante, comparándolas con las de las marcas que se crean de los puntos obtenidos por la Cámara. También se analizarán las velocidades lineales y angulares.

5.1. Entorno sin obstáculos y trayectoria circular

El entorno con el que se trabajará aquí es el mostrado en la *Figura 20*, en este entorno se trabaja sin obstáculos y en una trayectoria que será completamente circular.

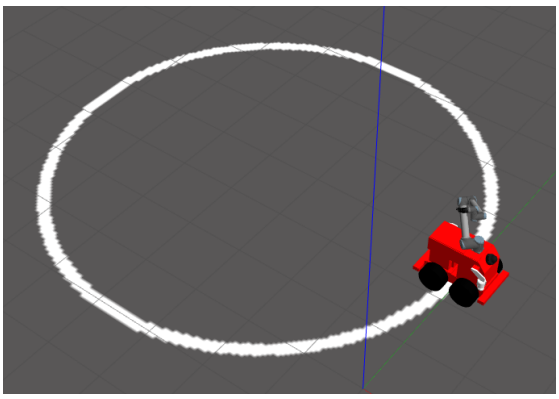
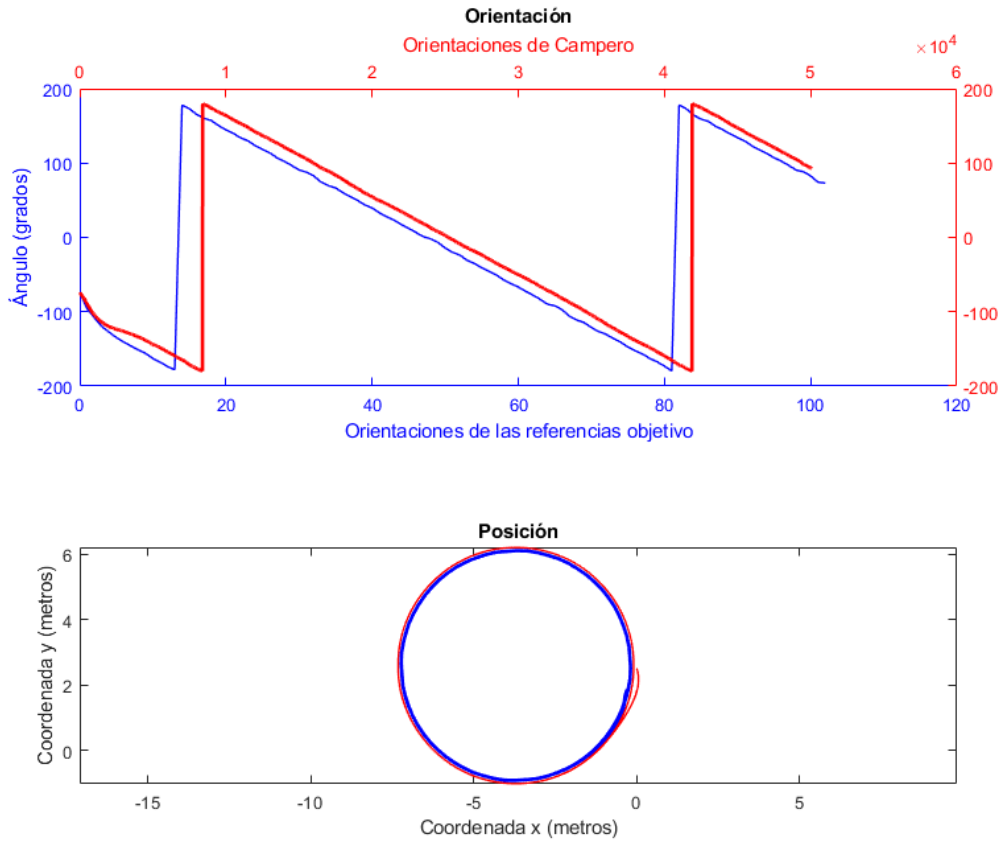


Figura 20: Entorno circular usado para simulación.

Este es un entorno básico, pero que a su vez es capaz de proporcionar información valiosa de la robustez de nuestra navegación, en cuanto a la realización de trayectorias no rectas sino completamente curvas. Lo que puede provocar en navegaciones poco sólidas la pérdida constante de la línea o la no realización de trayectorias curvas sino poligonales.

- Navegación basada en centroides y a velocidad lineal constante:



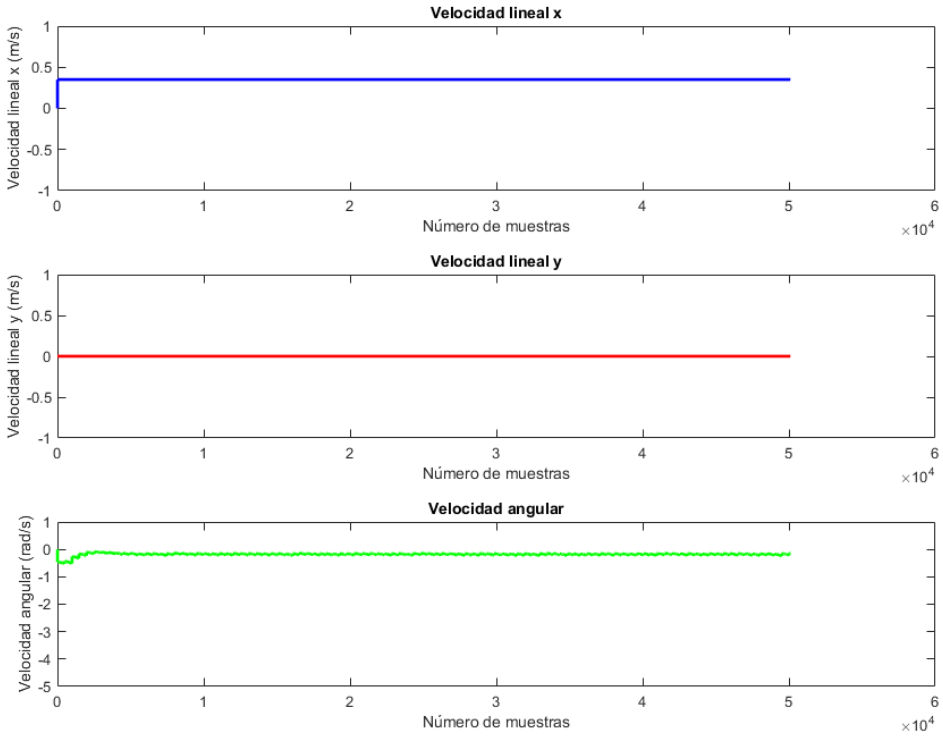
Gráfica 1: Posición y Orientación de la navegación basada en marcas en trayectoria circular.

Los resultados obtenidos en el entorno circular por parte de esta navegación son los que se encuentran en la *Gráfica* 1, donde lo que se puede ver es la posición y la orientación de Campero y a su vez la posición y orientación de los puntos objetivos a los que se debía desplazar. Todos estos datos fueron recogidos en el mismo intervalo de tiempo, pero a distintas frecuencias de muestreo, es por ello por lo que tenemos más cantidad de datos para los puntos de Campero que para los puntos objetivos, pues la frecuencia de recogida de datos en Campero es mayor.

Como se puede observar en la gráfica de posición, el Campero representado por la línea roja realiza una trayectoria prácticamente igual que la objetivo en azul, pero situándose siempre a una cierta distancia de él, derivado del error de posicionamiento de los puntos obtenidos de la cámara. Puesto que estos contienen un cierto error en la colocación en el eje y con respecto al robot, ya que el ángulo de visión de la cámara proporciona más información verticalmente que horizontalmente.

También se puede observar en la *Gráfica* 1, que la orientación del campero y de los puntos objetivos son la misma en todo momento, lo que indica que el Campero es capaz de mantener la orientación continuamente.

Además, en la *Gráfica 2* se pueden visualizar las velocidades lineales y angulares de Campero, demostrando que campero permanece con velocidad lineal x constante durante todo el tiempo, siendo esta velocidad lineal una especificación de este tipo de navegación, permaneciendo como se observa la velocidad lineal y cero, puesto que no se utiliza. A la vez que se ve que la velocidad angular de campero tiende a ser -0.18 rad/s, siendo estable en el permanente únicamente controlando esta velocidad mediante con un control proporcional.



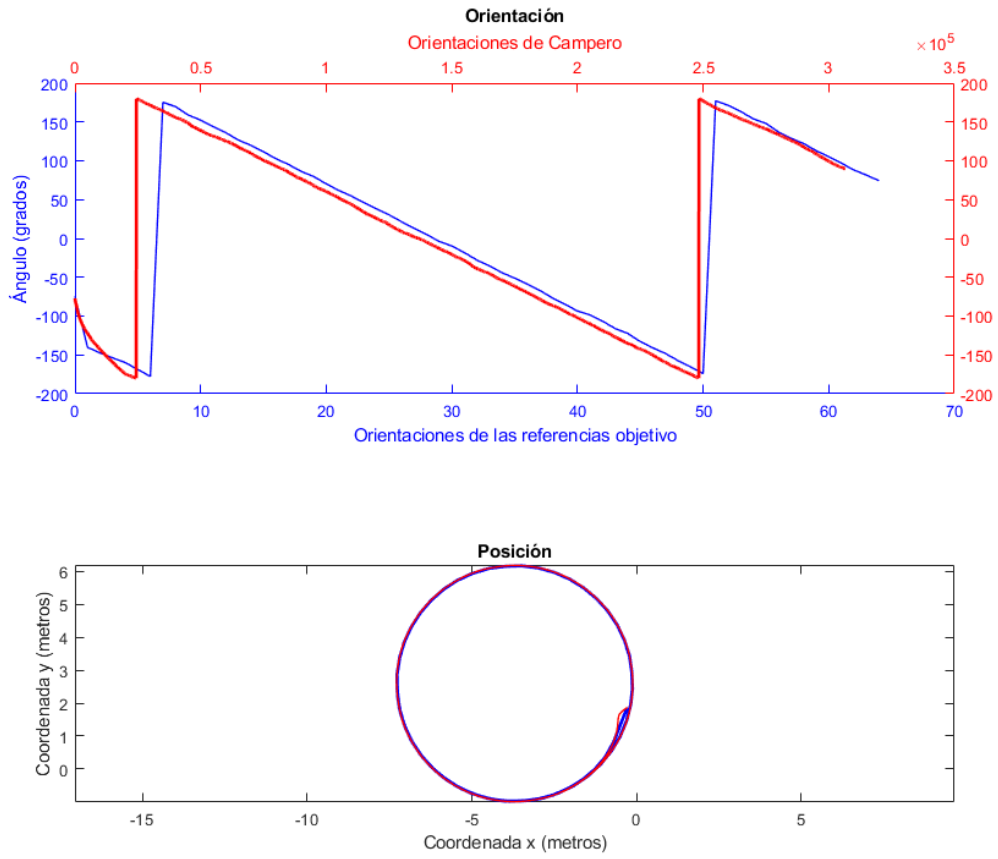
Gráfica 2: Velocidades lineales y angulares de la navegación basada en marcas en trayectoria circular.

Gracias a que esta velocidad angular es estable en el permanente y tiende a un valor, el Campero es capaz de seguir la línea circular como lo hace y con una orientación tan precisa como la encontrada en la *Gráfica* 1.

Con la información que se ha analizado se puede decir que esta navegación es muy robusta en trayectorias circulares y que el seguimiento de la línea es lo suficientemente buena como para poder obtener una buena posición y orientación mediante los puntos que se obtienen por cámara.

-Enlace al experimento:
https://drive.google.com/file/d/1CJZ3EyPUNe9iGwGGLImk_9I7lUFtIp0O/view?usp=sharing

- Navegación con orientación de línea prioritaria:



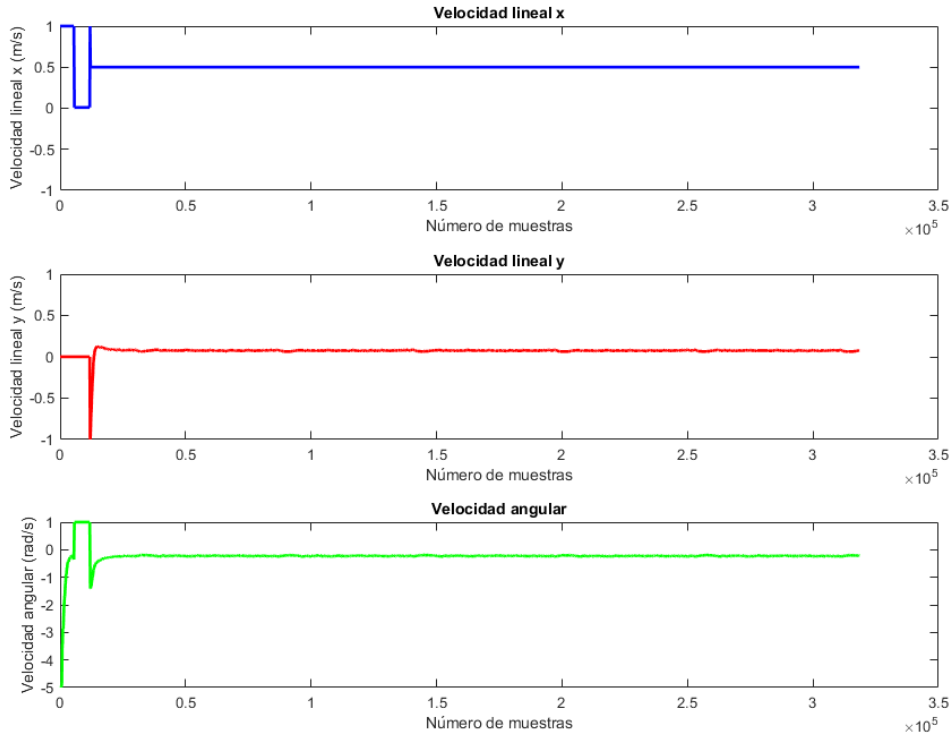
Gráfica 3: Posición y Orientación de la navegación con orientación de línea prioritaria en trayectoria circular.

Los resultados obtenidos en la trayectoria circular, pero utilizando la navegación con orientación de línea prioritaria son los que se tienen en la *Gráfica 3* y *Gráfica 4*, donde nuevamente se tendrán la posición y orientación tanto de Campero como de las marcas en la primera gráfica y las velocidades en la segunda gráfica.

Como se puede observar en la *Gráfica 3*, con esta navegación el seguimiento de la línea es mucho más precisa que con la anterior navegación, al que se observa claramente en la gráfica de la posición donde ambas líneas están sobrepuestas con lo que se puede asegurar que mantienen posiciones idénticas. Esto se debe en gran parte a que en esta navegación debemos recordar que no se seguían las marcas creadas, sino que se realizaba la navegación directamente con la información que se obtenía del centroide de la imagen de

la cámara y se realizaban correcciones de posicionamiento de este mediante la velocidad lineal en y.

Como también se puede observar en la gráfica de la orientación, las líneas son iguales recorriendo los mismos ángulos, pero, en cambio, mucho más suaves en estos que con la navegación anterior.



Gráfica 4: Velocidades lineales y angulares de la navegación con orientación de línea prioritaria en trayectoria circular.

Como se puede ver en la *Gráfica 4* las velocidades sufren cambios en los primeros instantes producidos porque como se comentaba en puntos anteriores, en esta navegación el comienzo para situarse en la línea se realiza por una navegación por marcas. Es por ello por lo que se contemplan ciertas variaciones al inicio, tras esto la navegación como se observa se basa en una navegación en la que se trabaja con dos velocidades lineales y una angular. La velocidad lineal en x se mantiene constante e igual que en la anterior navegación, pero esta vez a 0,5 m/s y únicamente se controla la velocidad lineal en y con un proporcional, con el cual se observa que es suficiente y no genera inestabilidad, tendiendo a una velocidad estable y en torno a 0,08 m/s. Dicha velocidad como se comentaba en otro apartado es el offset que se le agregaba a la velocidad angular, permitiendo el trabajo con valores menores en esta velocidad. A la velocidad angular aparte de este offset también se le realiza un control proporcional obteniendo como se muestra en la gráfica un resultado estable en el permanente y tendiente a -0,22 rad/s.

Por lo que se puede decir que esta navegación al igual que la anterior presenta una gran robustez en trayectorias circulares, y no se podría llegar a decidir cuál usar en base únicamente a este experimento. Por lo que realizaran otros que se explicaran a continuación.

-Enlace al experimento:

https://drive.google.com/file/d/1h5jn7fKmDEJy1EGCpDksgAZVTjyli7X5/view?usp=sharing

5.2. Entorno con obstáculos y trayectoria con esquinas en 90 grados

El siguiente entorno de trabajo que se utilizó en simulación es el que podemos observar en la *Figura 21*, fue el primer entorno que se utilizó. El cual consta de un recorrido representado por una línea blanca que deberá seguir el robot, y dos obstáculos de diferente forma, uno cilíndrico y otro cúbico.

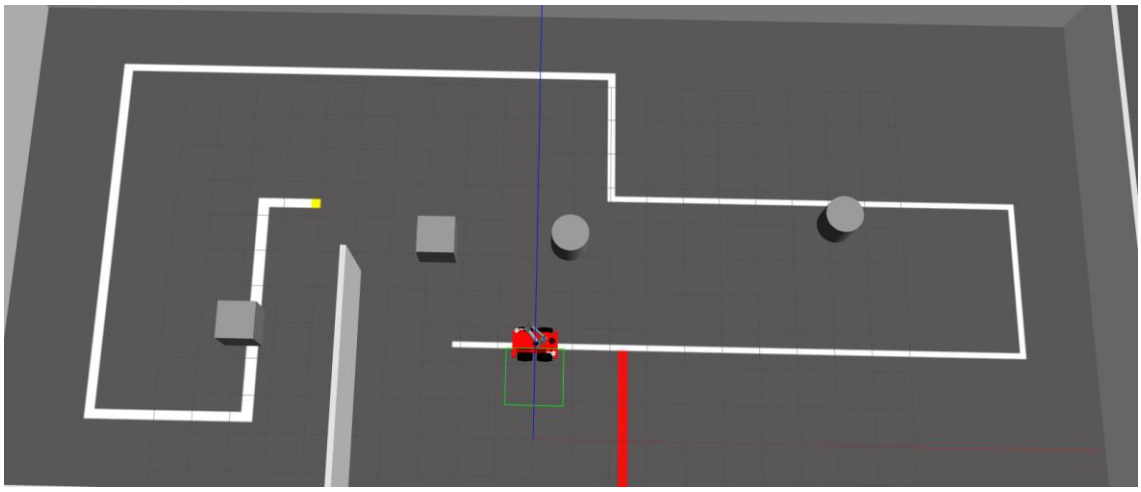
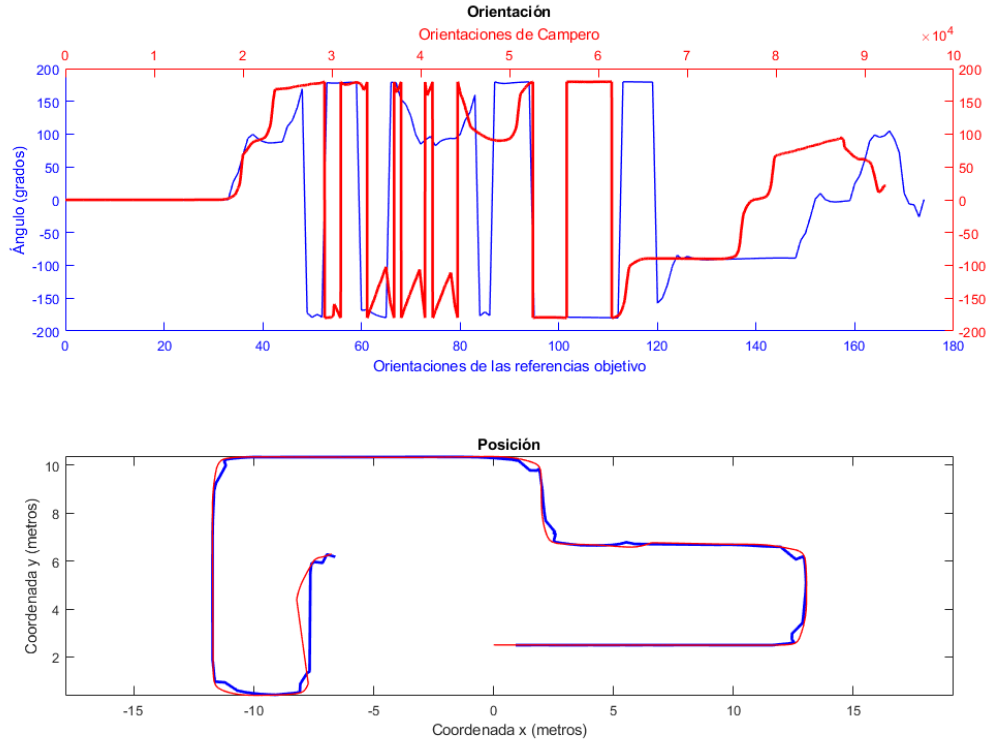


Figura 21: Entorno de simulación con obstáculos y esquinas con giros de 90º.

En este entorno como se ha comentado se trabaja con dos obstáculos, lo que nos permite conocer la robustez que presenta nuestra navegación a la evasión de estos. Además, como se puede observar los giros que hay presentan ángulos de 90º con lo que se pondrá también a prueba si la navegación es capaz de realizar estos giros sin llegar a perder la línea, a la vez que se comprobará cuál es la eficacia de seguimiento de línea de Campero en estas condiciones.

- Navegación basada en centroides y a velocidad lineal constante:



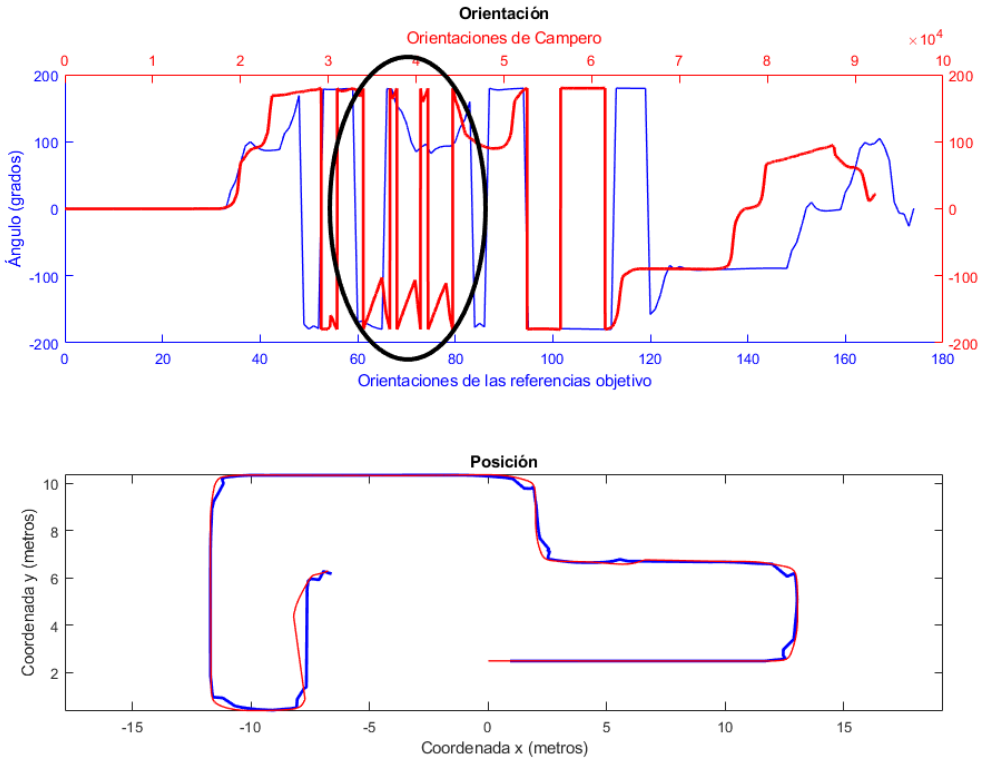
Gráfica 5: Posición y Orientación de la navegación basada en marcas en trayectoria con esquinas.

Los resultados que se obtuvieron para la navegación basada en marcas en este entorno, en el cual ya se trabajaba con trayectorias no cerradas y en las cuales no había giros suaves sino ángulos rectos, son los que se encuentran en la *Gráfica 5* y *Gráfica 7.*

En la primera gráfica de nuevo se encuentran las posiciones de tanto Campero como de los puntos objetivos, donde se puede ver que el robot sigue la línea muy bien, siendo la línea roja de la gráfica de posición la trayectoria seguida por el robot, a pesar de que el posicionamiento de los puntos en el mapa, línea azul en la gráfica, no era del todo correcto debido a las esquinas en 90º que desplazaban el centroide y, por lo tanto, dichos puntos.

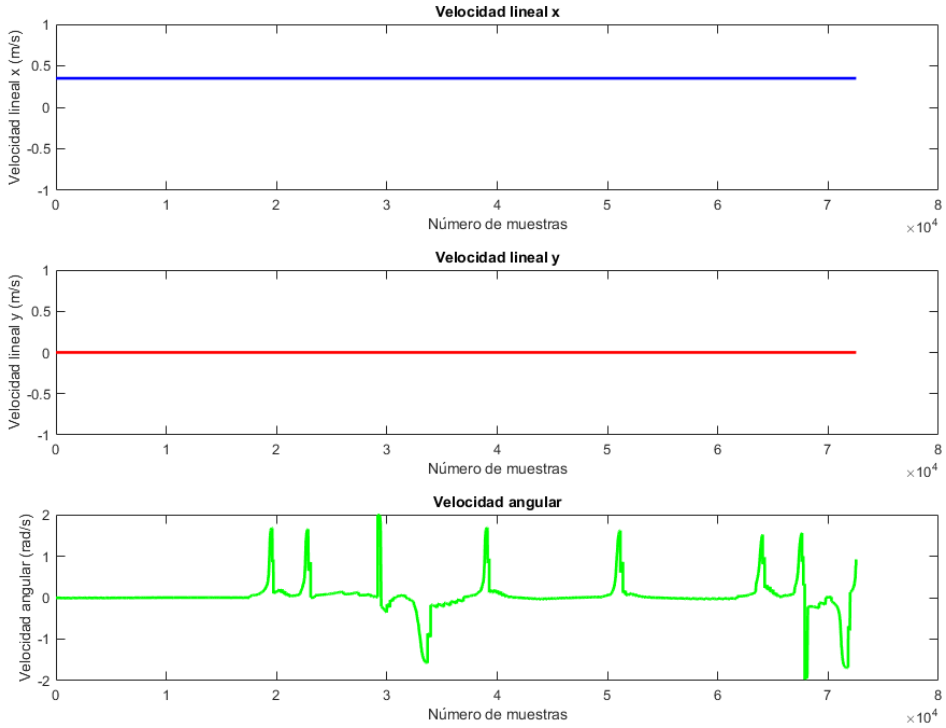
A pesar de todo, el Campero como se comentaba realiza un buen posicionamiento en la línea y además con la otra gráfica, en la cual se puede ver cuándo se ha producido un giro por los cambios angulares, se puede confirmar el buen posicionamiento pues pese a que el posicionamiento de los puntos no era bueno, la orientación sí que era más acertada y aunque no fuera totalmente correcta se realiza una buena orientación con dicha navegación.

Además, se observa que existe un giro en el que Campero sufre dificultades con esta navegación, estas se dan en el tercer giro, puesto que como se observa en la *Gráfica 6* donde se ha resaltado la zona en la orientación campero pierde la línea e intenta reorientarse tres veces, consiguiéndolo finalmente y prosiguiendo la navegación sin más dificultades.



Gráfica 6: Fallo de la navegación en el tercer giro.

El final de esta trayectoria se encuentra en una zona crítica, puesto que el punto final está cercano a un giro y este a su vez a una evasión de un obstáculo, es por ello por lo que se visualiza tanto un alejamiento y un acercamiento de nuevo a la línea azul en la gráfica de posición. También es el motivo por el que la orientación realiza esos cambios en la gráfica hasta llegar a 0º desde los 90º del giro.



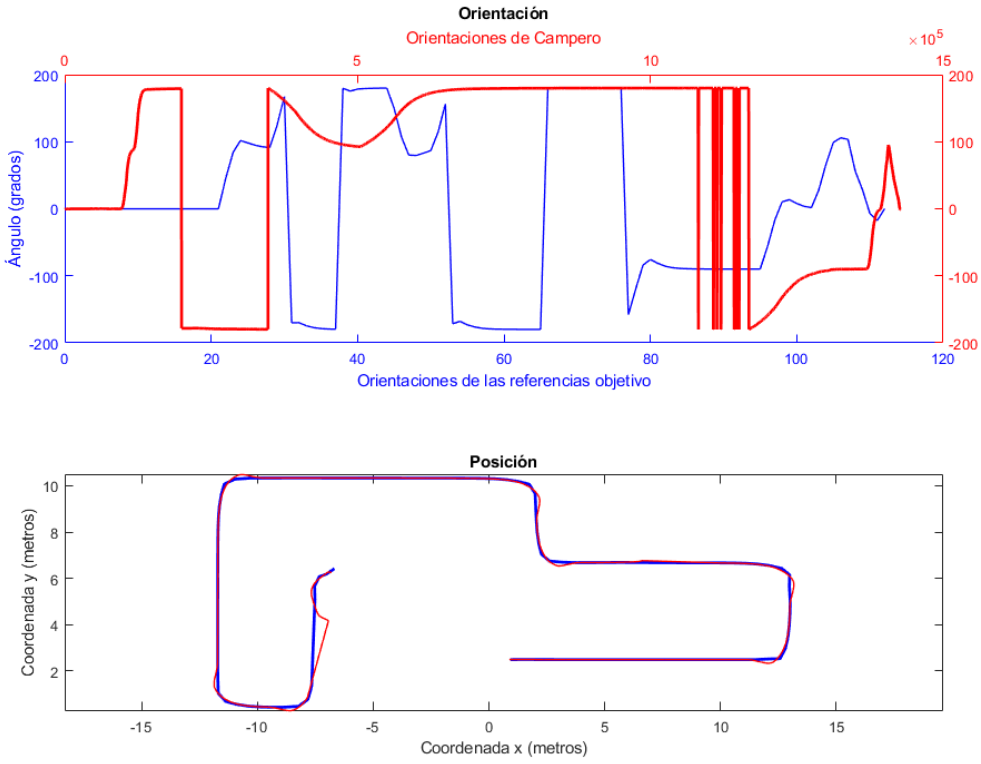
Gráfica 7: Velocidades lineales y angulares de la navegación basada en marcas en trayectoria con esquinas.

En la gráfica de velocidades, *Gráfica 7*, lo que más relevancia tiene aparte de que se sigue manteniendo velocidad lineal constante sin problemas, es que como se observa en las velocidades angulares el Campero sufre variaciones bruscas en cada uno de los giros por ser estos de 90º, lo que le hacían realizar arcos de circunferencia pequeños y a una gran velocidad angular, como se ha comprobado anteriormente el proporcional utilizado era suficiente en trayectorias circulares y como demuestra tanto la orientación como la posición del campero, el control de la velocidad angular con el proporcional es suficiente. Puesto que a pesar de los cambios bruscos en estas velocidades el robot no pierde de vista la línea salvo en la excepción anterior.

-Enlace al experimento:

https://drive.google.com/file/d/1l63jyxbWY4npoxWOo-IyBqLIGqhfw6bM/view?usp=sharing

- Navegación con orientación de línea prioritaria:

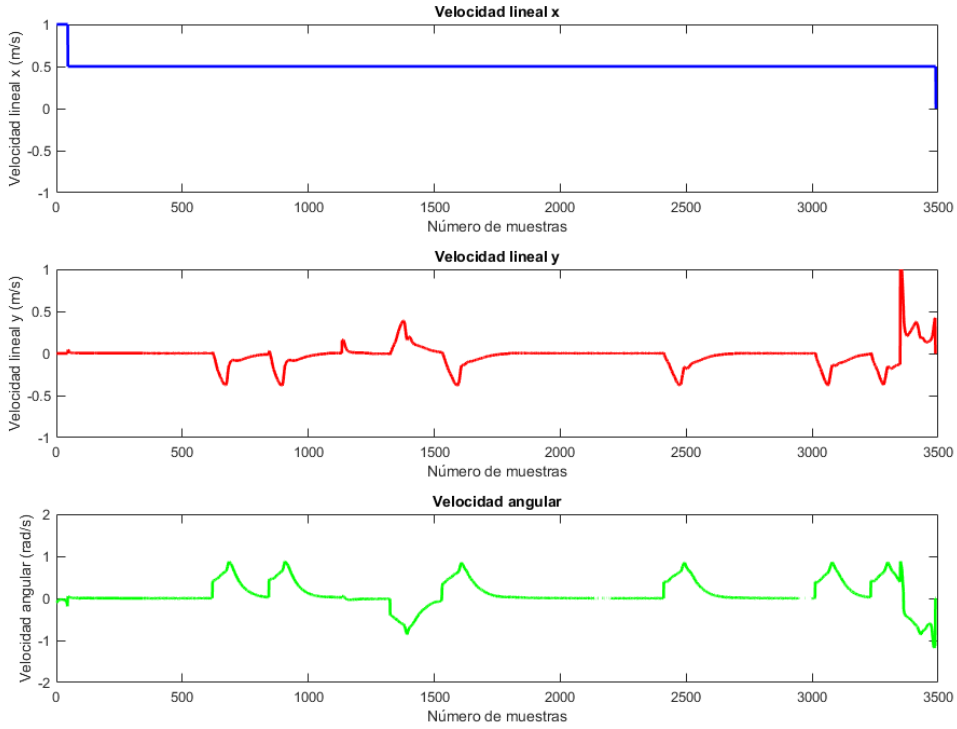


Gráfica 8: Posición y Orientación de la navegación con orientación de línea prioritaria en trayectoria con esquinas.

Los resultados que se encuentran en este punto son los que se obtuvieron de realizar en este entorno la navegación por orientación de línea prioritaria. Estos resultados se pueden encontrar en la *Gráfica 8* y *Gráfica 9*.

Lo que se comentará de estos resultados es que como se observa en la *Gráfica 8*, el posicionamiento de Campero por medio de esta navegación no solo daba resultados más limpios en cuanto al seguimiento de la línea en comparación a la navegación anterior, sino que también permitía que las marcas que se creasen de la línea fuesen colocadas mejor de lo que se hacía en la anterior. Como también se puede ver la orientación que presentaba campero era mucho más suave y eso es debido en gran parte a trabajar con la ya comentada velocidad lineal en y, que permitía cuando se llegaba a un giro de 90º posicionarse mejor con respecto a la línea inmediatamente posterior a este, como se puede observar en la gráfica de posición.
Permitiendo de esta forma no perder la línea en ningún momento y situarse tanto posicional como orientativamente mejor que en la navegación anterior.

El único problema con el que se encontraba esta navegación es en el obstáculo cúbico que se encuentra al final, que se puede ver en *Figura 21*, en el cual como se comprueba en la *Gráfica 8* el campero necesita volver a encontrar la línea y es por ello por lo que en la gráfica de posición aparece desplazado de la línea azul.



Gráfica 9: Velocidades lineales y angulares de la navegación con orientación de línea prioritaria en trayectoria con esquinas.

En cuanto a las velocidades, se puede observar que gracias a la velocidad lineal y la velocidad angular es menor con respecto a la obtenida con la navegación anterior.
Además, se puede ver que en el único problema que teníamos en este entorno la velocidad lineal y es capaz de reconducir al Campero a la línea con gran facilidad, aunque para ello el valor deba ser bastante alto con respecto al resto de velocidades lineales, aunque dicho valor no presenta un problema pues no es tan grande tratándose de Campero el cual es un robot bastante pesado.

Como conclusión a este experimento en un entorno con giros en ángulos rectos se puede afirmar que ambas navegaciones presentan buenos resultados, pero que a la hora del seguimiento de líneas sería mejor la utilización de esta última navegación pues presenta velocidades menores, orientaciones más suaves y mejor posicionamiento.

-Enlace al experimento:

https://drive.google.com/file/d/1vtfiRVPDUDwwc0hhWuDd7FnklVO18ni6/view?usp=sharing

5.3. Entorno con obstáculos y trayectoria sin esquinas

En este punto el entorno utilizado es el representado en la *Figura 22*, el cual es la modificación del que se encuentra en la *Figura 21* debido a que se eliminan las esquinas con giros de 90º por trayectorias circulares o por diagonalizaciones de esquinas muy próximas, como la que se puede observar. Al realizar estos cambios la intención era comprobar si el robot podía seguir la trayectoria con mayor facilidad al anterior entorno, para determinar qué tipo de trazado es el más conveniente.

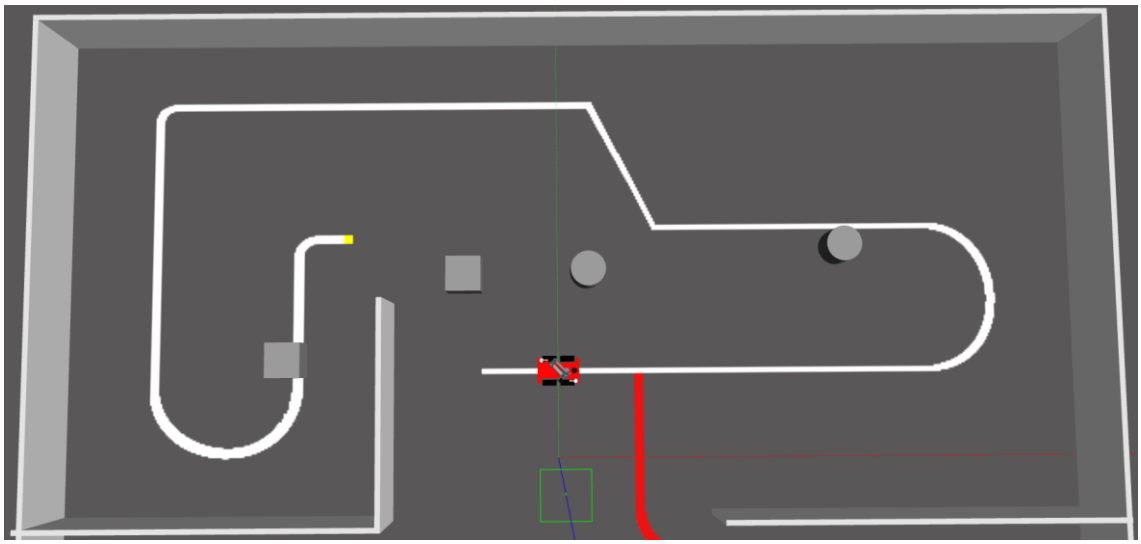
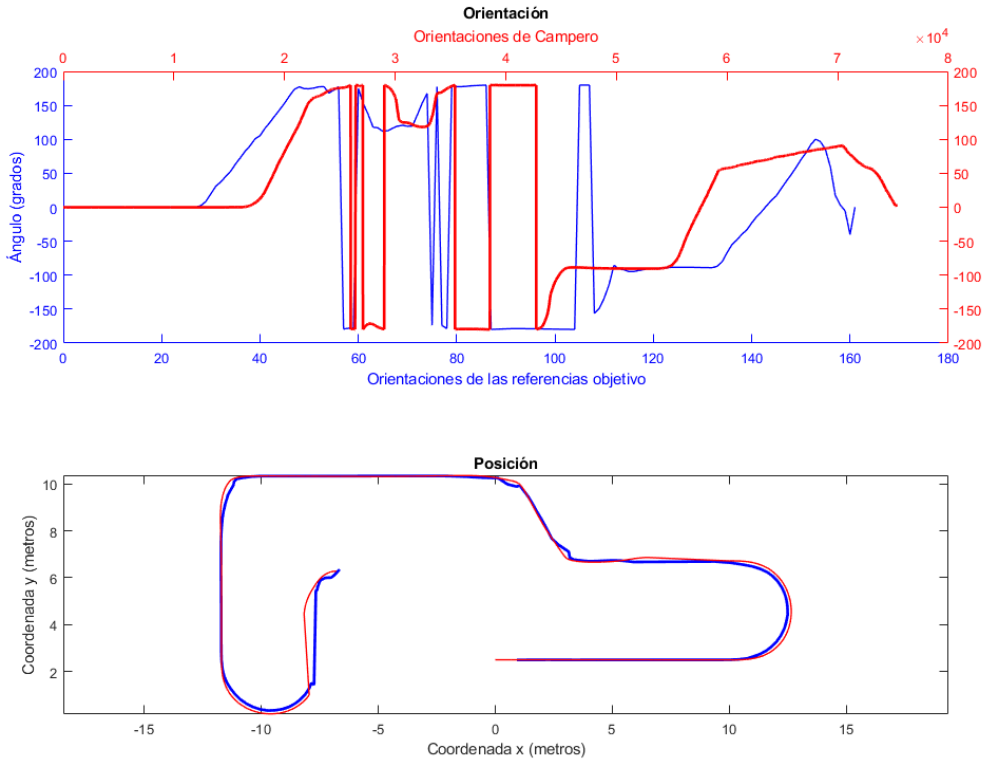


Figura 22: Entorno de simulación con obstáculos y esquinas con ángulos suavizados o sustituidos.

También se realizan estos cambios para comprobar cómo reacciona el robot, con las dos navegaciones, ante obstáculos que se puedan encontrarse inmediatamente posteriores a la realización de una trayectoria circular, como el que se puede observar en la *Figura 22* en la parte inferior izquierda. Donde se ve un obstáculo cúbico que se encuentra muy próximo a la finalización de una trayectoria curva.

A su vez se comprobará si el Campero es capaz de situarse en una posición previa que le permita realizar la evasión de la manera más sencilla posible, y en el caso en el que esto no fuera posible comprobar que medidas toma para regresar de forma casi inmediata a la línea.

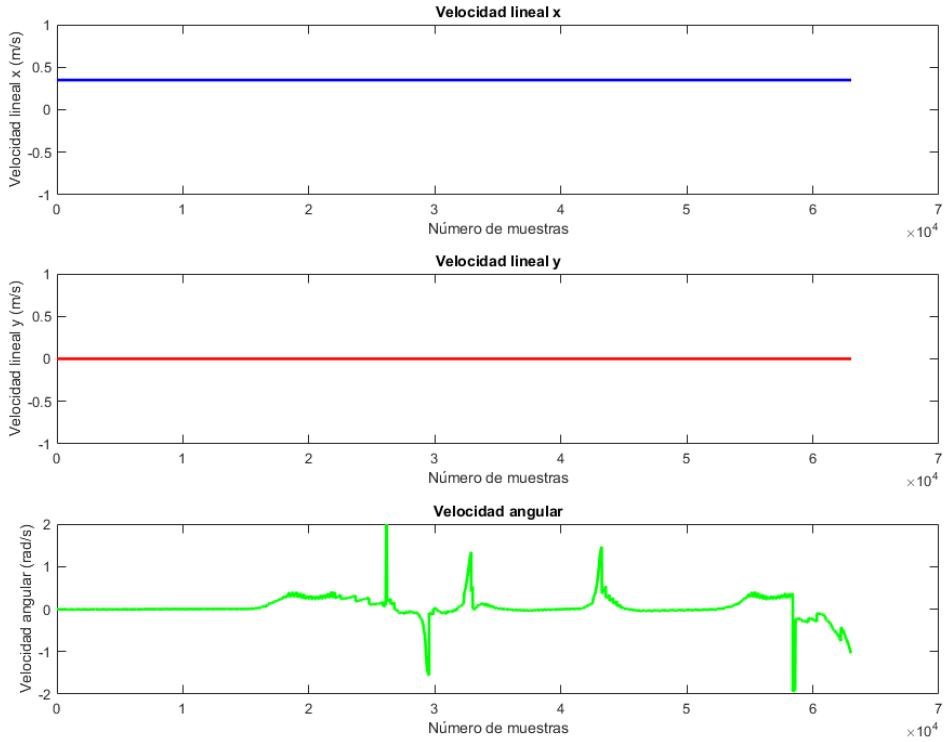
- Navegación basada en centroides y a velocidad lineal constante:



Gráfica 10: Posición y Orientación de la navegación basada en marcas en trayectoria sin esquinas.

Los resultados obtenidos en este entorno para navegación basada en marcas son los que tenemos en la *Gráfica 10* y *Gráfica 11*, donde se puede observar que esta navegación trabaja mucho mejor cuando no ha de enfrentarse a ángulos rectos, puesto que se posiciona y orienta mucho mejor. Además de realizar un seguimiento de la línea más limpio posibilitando que las marcas con las que trabaja sean más correctas. Como se ve los únicos momentos en los que la navegación hace que se generen malos puntos es cuando se tienen trayectorias curvas, pero muy cerradas lo que sería asimilable a los giros de 90º y cuando en el posicionamiento se ve la diagonal.

En cuanto a la orientación se han encontrado mejoras pues no se observan fallos como el que se encontraba en el experimento anterior, y además el Campero presenta la misma orientación que los objetivos a los que sigue, por lo que se puede afirmar que trabaja mejor en este experimento.



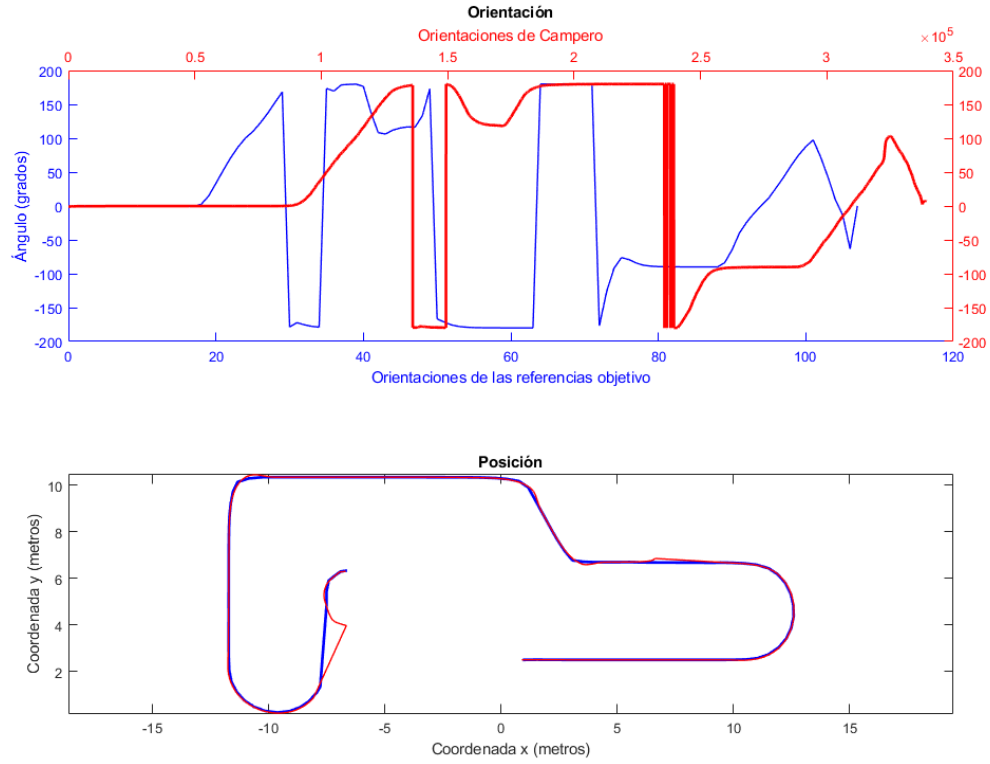
Gráfica 11: Velocidades lineales y angulares de la navegación basada en marcas en trayectoria con esquinas.

Con respecto a las velocidades que podemos observar en *Gráfica 11*, lo que se puede observar es que como se preveía en el giro de reposicionamiento, en los giros curvos cerrados y en la trayectoria diagonal es en los únicos momentos en los que la navegación presenta velocidades angulares elevadas, pero aun en esos momentos no pierde la línea. Y en las curvas más amplias Campero es capaz de mantener una velocidad angular aceptablemente baja y seguir la línea con esta velocidad muy bien cómo se comprobaba en el primer experimento.

-Enlace al experimento:

https://drive.google.com/file/d/1qvZmZ_1cv3dnqBIsHqitR3K_OwPKorYM/view?usp=sharing

- Navegación con orientación de línea prioritaria:

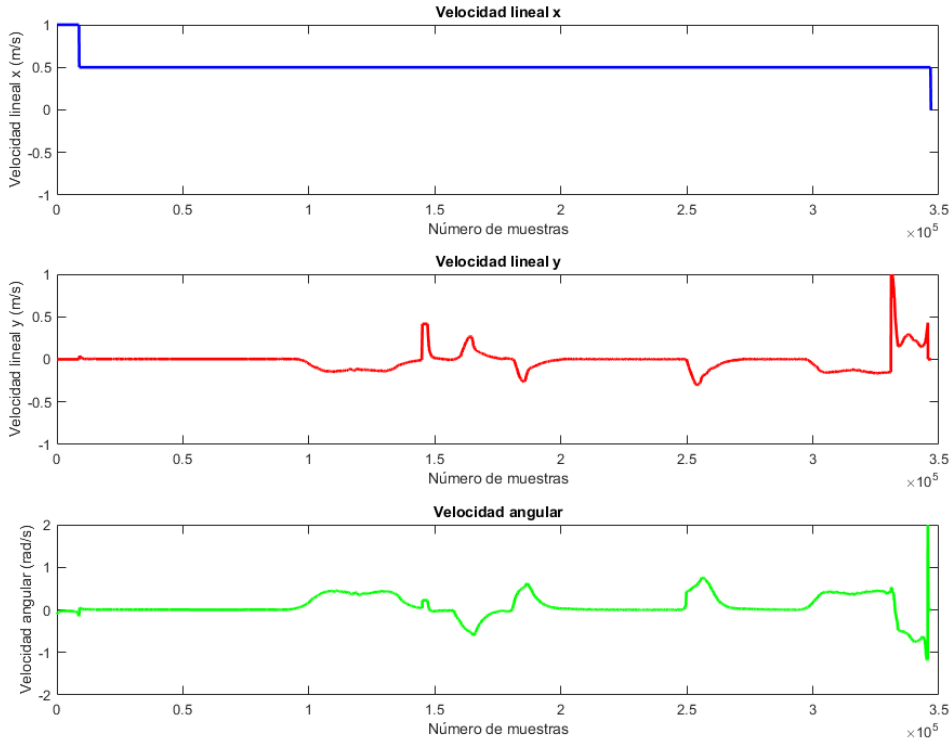


Gráfica 12: Posición y Orientación de la navegación con orientación de línea prioritaria en trayectoria sin esquinas.

Con esta navegación en este experimento los resultados obtenidos son los que se tienen en la *Gráfica 12* y *Gráfica 13*.

En la *Gráfica 12* lo que más llama la atención es que con esta navegación la posición del Campero es mucho mejor que con la navegación anterior, y también permite como sucedía en el resto de los experimentos que las marcas generadas fuesen mejores.

Además, la orientación que se obtiene con esta navegación como sucedía anteriormente es mucho más suave y, por lo tanto, facilita el seguimiento de la línea. Como sucedía en el experimento anterior, pero esta vez únicamente en el trayecto diagonal el campero se aleja al realizar los giros, para poder estar orientado a la línea todo el rato, solamente realiza esto en la diagonal, puesto que es tramo más radical para esta navegación.



Gráfica 13: Velocidades lineales y angulares de la navegación con orientación de línea prioritaria en trayectoria sin esquinas.

Como ya sucediese en el experimento anterior en la *Gráfica 13*, correspondiente a las velocidades, se puede observar que en los giros la velocidad lineal y reduce las velocidades angulares, provocando no solo esta reducción sino también como ya se ha comentado la obtención de una orientación más suave y coherente.

Además, como pasaba en el entorno explicado anteriormente para esta misma navegación en el último obstáculo vuelve a generar una velocidad y mayor, permitiendo que el Campero se vuelva a posicionar en la línea mucho más rápido.

-Enlace al experimento:

https://drive.google.com/file/d/1dM9W2yEftPrbKKII8ny_NcNQrk2_KMqc/view?usp=sharing

Como consecuencia de los experimentos realizados se puede concluir que ambas navegaciones son muy buenas y que, si hubiera que elegir una única de ellas, se debería elegir la mejor se adaptase a nuestro entorno.
Porque si nuestro entorno contiene esquinas con curvas o serpenteos, o incluso se pueden llevar a encontrar obstáculos podríamos elegir la navegación basada en marcas.
Pero ante cualquier otra situación la que nos otorgaría resultados más fiables sería la navegación con orientación de línea prioritaria, la cual como se ha visto presenta buen posicionamiento y orientación.

Únicamente se tomarán estas decisiones de elección si el robot con el que se trabaje tiene la posibilidad de trabajar omnidireccionalmente, si no se deberá utilizar la navegación basada en marcas, la cual también presenta como se ha visto buenos resultados.

Además, se realizaron los dos últimos experimentos para un trayecto distinto con la línea roja, obteniendo similares resultados que en los experimentos anteriores. Los entornos utilizados son los que encontramos en la *Figura 23* y *Figura 24*.

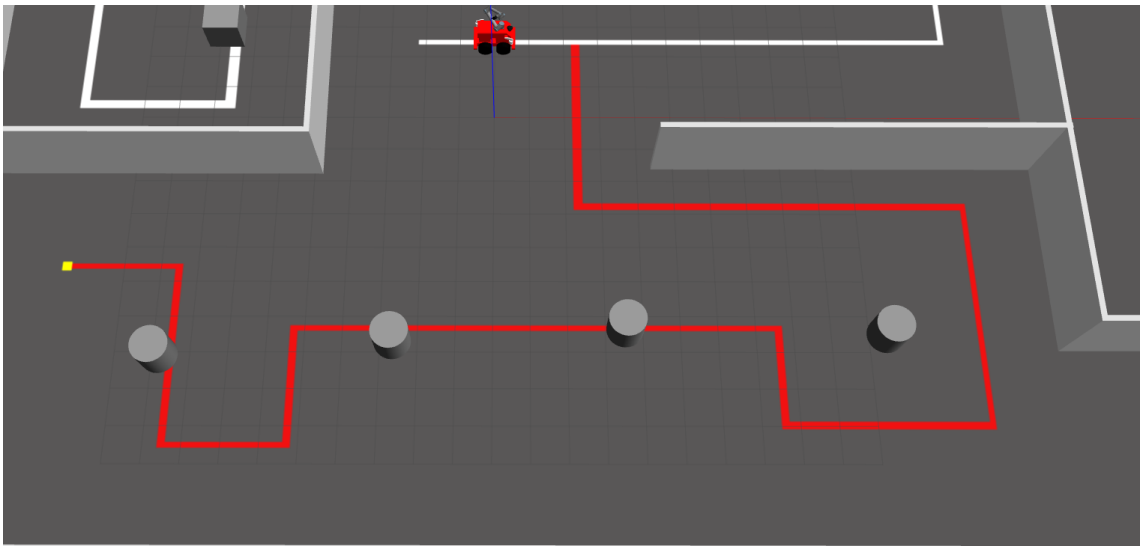


Figura 23: Entorno de simulación con obstáculos y giros en 90º, con línea roja.

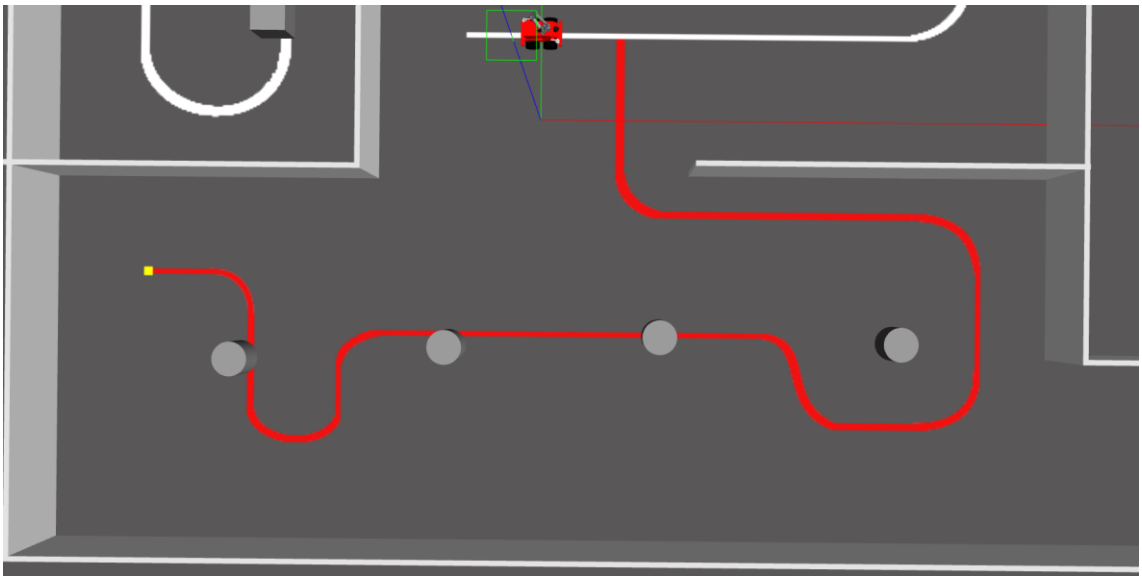


Figura 24: Entorno de simulación con obstáculos y esquinas con ángulos suavizados o sustituidos, con línea roja.

Como ya se ha comentado en estos experimentos se obtuvieron similares resultados que los obtenidos con la línea blanca, se realizaron para probar sobre todo el comportamiento de la navegación ante un nuevo tipo de color de línea. Aparte de probar como funcionaría ante más obstáculos y los resultados fueron buenos.

6. Experimentos en entorno real

En este apartado se realizará una guía de cuáles son los primeros pasos que se deben dar para realizar una posible implementación.

-En primer lugar, lo que se debe realizar es la configuración de los topics, que no hayan sido creados por nosotros, puesto que los utilizados en simulación pueden tener otro nombre con respecto a los que tenga el robot real. Si no se realizase este cambio podrían saltar errores que no dejasen posiblemente realizar la implementación.

-En segundo lugar, se tiene que hacer es calibrar la cámara con el robot real, posicionándola de tal forma que las imágenes obtenidas sean lo más parecidas a las obtenidas en simulación. O dicho de otra manera posicionando la cámara con una visión lo más abajo posible que se pueda.

Tras obtenerse una buena posición de la cámara lo que se tiene que hacer es conseguir los umbrales de la línea que se quiera utilizar, para conseguir las máscaras que se deben utilizar.

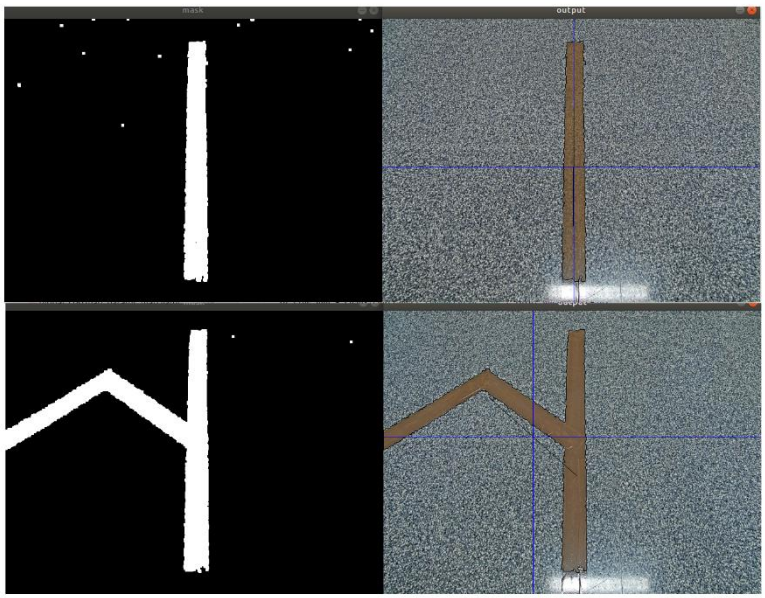


Figura 25: Imágenes captadas por la cámara de campero en una posición favorable y la mascaras obtenidas de las líneas que hay en estas imágenes.

-En tercer lugar, se debe hacer es modificar las velocidades y adaptarlas a la realidad, debido a que en simulación se trabajaban con entornos con dimensiones muy grandes y esto da pie a utilizarse velocidades más altas.

-En cuarto lugar, sería crear los entornos de implementación y realizar las implementaciones.
https://drive.google.com/drive/folders/1Am1QEeBLy4SrzVSeKpD72lNf2ncmGiO0

7. Conclusiones y trabajo futuro

7.1. Conclusiones

Tras mucho trabajo podemos decir que se han cumplido los objetivos propuestos, pues se ha conseguido generar códigos de navegación desde cero que trabajasen de forma autónoma y de forma robusta. Además, se ha conseguido que el robot realice las navegaciones basadas en visión y seguimiento de carriles, con evasión de los obstáculos que pueda haber, y el reposicionamiento en la línea tras perdida, sin muchos problemas y obteniendo los resultados esperados, alcanzando los objetivos que se marcaban paso a paso.

En primer lugar, efectuamos el cambio de versión de Ubuntu 16.04 a Ubuntu 18.04, con las necesarias adaptaciones que este cambio nos generaba, también nos familiarizamos con ROS y con sus herramientas, pues se tuvo que aprender ya no solo a trabajar con las distintas funciones que ROS proporciona en código. Si no también con las herramientas de visualización, Gazebo y Rviz, para las cuales se tuvo que crear entornos de simulación y sus mapas.

Además del trabajo con ROS y sus herramientas en este trabajo también se han adquirido conocimientos en el lenguaje XML, debido a que se han tenido que realizar algunas modificaciones en los códigos internos de Campero, y en algún otro. Y, por otra parte, se ha profundizado en los conocimientos básicos, pero existentes de la programación en Python, pues en este proyecto han sido necesarios para poder realizar los códigos de navegación, debido a que se trabajaba con scripts.

Con todo lo anterior se pudo desarrollar el sistema de percepción de carril utilizado en la navegación, a partir de las imágenes obtenidas por la cámara, con el cual se desarrollaron métodos de seguimiento de línea, así como el sistema de evasión de obstáculos y de reposicionamiento en línea. La unión de los anteriores métodos dio lugar a las estrategias de navegación con las que obtuvimos una navegación robusta, demostrando esto a través de los experimentos realizados tanto en simulación como por medio de la implementación con Campero. Donde los resultados obtenidos por medio de gráficas de posicionamiento y de velocidad fueron positivos.

7.2. Trabajo futuro

En este trabajo se ha partido de cero en cuanto a dar funcionalidad de seguimiento continuo de marcas a Campero, a la vez que se ha aprendido sobre éste y sobre ROS.

A partir de este trabajo se pueden desarrollar no solo mejoras con respecto a lo presentado en él, sino también se podrían incluir opciones como el uso conjunto de brazo y seguimiento. Pues podría ser viable el realizar tareas en distintos puntos señalizados con marcas, mientras el robot utilizaría estos u otras navegaciones para llegar a través de líneas hasta dichos puntos, mezclando varias cosas a la vez y haciendo a Campero un robot manipulador móvil versátil.
También sería de consideración dotar a Campero de códigos propios que le permitiesen evadir obstáculos, sin la necesidad de disminuir completa o parcialmente su velocidad, como ocurre en la navegación que se ha desarrollado en la cual la evasión se realiza por medio de uno de los paquetes de ROS, para hacer la navegación completamente dependiente de una única velocidad lineal.

Además, se podría desarrollar estrategias de navegación que permitiesen trabajar con intersecciones o bifurcaciones de la línea permitiendo realizar navegaciones sin problemas al encontrarse con estas.

Bibliografía

[1] ROS http://wiki.ros.org/, 2021.

[2] David Barrera Gracia. Navegación autónoma de robot manipulador móvil con cámara y laser en el entorno ROS. Trabajo fin de grado, Universidad de Zaragoza, 2020-2021 https://zaguan.unizar.es/record/101417/files/TAZ-TFG-2020-4852.pdf/

[3] COMMANDIA. http://commandia.unizar.es/es/lo-basico-de-commandia/ ,2021.

[4] Gazebo, http://gazebosim.org/, 2021.

[5] Rviz, http://wiki.ros.org/rviz, 2021.

[6] Artículo sobre robots Amazon, 2017 https://www.elperiodico.com/es/economia/20170426/amazon-implantara-sus-robots-en-las-plantas-de-el-prat-y-castellbisbal-5998232

[7] Carpeta con todos los experimentos en simulación realizados: https://drive.google.com/drive/folders/1YaWkC_9oJoWginiw3LKRy1AD1oduFH32?usp=sharing

[8] dudasdavid. Código detector, Github, 2020. https://github.com/dudasdavid/line_follower/blob/master/nodes/line_detector.py

[9] dudasdavid. Código navegación, Github, 2020. https://github.com/dudasdavid/line_follower/blob/master/nodes/line_controller.py

[10] khaledgabr77. Código detector, Github, 2019. https://github.com/khaledgabr77/line_follwer_ros/blob/master/follower_ros.py

[11] OpenCV https://opencv.org, 2021.

[12] Costmap_prohibition_layer, https://wiki.ros.org/costmap_prohibition_layer, 2021.

[13] Openrave, http://wiki.ros.org/openrave, 2021.

[14] Robot_localization, http://wiki.ros.org/robot_localization, 2021.

[15] Teleop_panel, http://docs.ros.org/en/melodic/api/rviz_plugin_tutorials/html/panel_plugin_tutorial.html , 2021.

[16] Universal_robot, https://github.com/ros-industrial/universal_robot/tree/melodic-devel, 2019.

[17] Python, https://www.python.org/, 2021.

Anexos

Anexo A. Códigos creados

El siguiente código es el encargado de trabajar con las imágenes captadas por la cámara, obteniendo de ellas la línea y generando los puntos, con posición y orientación que se utilizaran en el resto.

- Line_tracker.py

```
#!/usr/bin/env python

"""line_tracker.py V. 1.0 07/07/2021
    Autor: Oscar Alejandro
    Codigo modificado a partir de:
https://github.com/khaledgabr77/line_follwer_ros/blob/master/follower_ros.py
"""
import rospy, cv2, cv_bridge, numpy, math, tf
from tf.transformations import quaternion_from_euler
from sensor_msgs.msg import Image, CameraInfo
from geometry_msgs.msg import Twist,Pose
from std_msgs.msg import Bool, Int8, Float32MultiArray
from nav_msgs.msg import Odometry


threshold_value_White = numpy.array([[195,195,195],[255,255,255]]) #White
#threshold_value = numpy.array([[0,0,0],[38,38,38]]) #Black
threshold_value_Yellow = numpy.array([[0,130,150],[150,255,255]]) #Yellow
threshold_value_Red = numpy.array([[0,0,150],[150,150,255]]) #Red
#threshold_value = numpy.array([[27,100,27],[73,255,193]]) #Green

class Tracker():
  def __init__(self):
    self.bridge = cv_bridge.CvBridge()
    self.goal = Pose()
    self.goal.orientation.w = 1.0
    self.new_goal = False
    self.line_detected = False
    self.array_to_send = Float32MultiArray()
    self.end_detected = False
    self.in_line = False
    self.positioned = False
    self.oriented = False
    self.orientation = None
    self.aux_orientation = None
    self.i = 0
    self.searching_wait = False
    # self.after_searching = False
    self.end = False


  # def new_goal_callback(self, msg):
  #     self.new_goal = msg
  def actual_ori_callback(self,msg):
      self.orientation = msg.pose.pose.orientation
```

```
  def quat_to_euler(self, quat):
      quaternion = tuple([quat[0], quat[1], quat[2], quat[3]])
      euler = tf.transformations.euler_from_quaternion(quaternion)
      return euler
  def positioned_in_line_callback(self,msg):
      self.positioned = msg
  def oriented_in_line_callback(self,msg):
      self.oriented = msg
  def searching_line_callback(self, msg):
      self.searching_wait = msg.data
      # if msg.data:
      #     self.after_searching = True
  def final_callback(self,msg):
      self.end = msg


def image_callback(msg):
      image = tracker.bridge.imgmsg_to_cv2(msg, desired_encoding='bgr8')
      h, w = image.shape[:2]
      cx, cy = 0,0
      if h != 480 or w != 640:
          dim = (640, 480)
          image = cv2.resize(image, dim, interpolation=cv2.INTER_AREA)
      #PREPARE THE IMAGE FOR UTILIZATION

      blurImg = cv2.GaussianBlur(image, (5,5), 0) #Gaussian Blur Smoothing

      threshImg = cv2.inRange(blurImg, threshold_value_White[0], threshold_value_White[1])
      # threshImg = cv2.inRange(blurImg, threshold_value_Red[0], threshold_value_Red[1])
      threshImg_end = cv2.inRange(blurImg, threshold_value_Yellow[0], threshold_value_Yellow[1])

      #Erode and dilate to remove accidental line detections
      mask = cv2.erode(threshImg, None, iterations=2)
      mask = cv2.dilate(mask, None, iterations=2)

      mask_end = cv2.erode(threshImg_end, None, iterations=2)
      mask_end = cv2.dilate(mask_end, None, iterations=2)

      #FIND TH CONTOURS OF THE IMAGE
      _, contours, hierarchy = cv2.findContours(mask.copy(), 1, cv2.CHAIN_APPROX_NONE)
      _, contours_end, hierarchy_end = cv2.findContours(mask_end.copy(), 1,
cv2.CHAIN_APPROX_NONE)

      h, w, d = image.shape
      #cx = None
      c = None
      area = None
      c_end = None
      area_end = None

      if len(contours)>0:
          c = max(contours, key=cv2.contourArea)
          area = cv2.contourArea(c)
          rospy.loginfo("AREA Linea")
          rospy.loginfo(area)

      if len(contours_end) > 0 and tracker.in_line:
          c_end = max(contours_end, key=cv2.contourArea)
          area_end = cv2.contourArea(c_end)
          rospy.loginfo("AREA FINAL")
```

```
        rospy.loginfo(area_end)

    if area > 6000:
        tracker.line_detected = True
        rospy.loginfo("Linea detectada")

        # c = max(contours, key=cv2.contourArea)

        M = cv2.moments(c)
        cx = int(M['m10'] / M['m00'])
        cy = int(M['m01'] / M['m00'])

        cv2.line(image, (cx, 0), (cx, h), (255, 0, 0), 1)
        cv2.line(image, (0, cy), (w, cy), (255, 0, 0), 1)
        cv2.drawContours(image, c, -1, (0, 0, 0), 1)

        [vx,vy,x,y] = cv2.fitLine(c,cv2.DIST_L2,0,0.01,0.01)

        angle = math.atan2(vy, vx)
        if angle < 0:
            angle = angle + math.pi / 2
        else:
            angle = angle - math.pi / 2

        angle *= -1
        rospy.loginfo(vx)
        rospy.loginfo(x)
        rospy.loginfo(vy)
        rospy.loginfo(y)
        cv2.line(image,
                 (int(cx - math.cos(angle - math.pi / 2) * 100), int(cy + math.sin(angle - math.pi / 2) * 100)),
                 (int(cx + math.cos(angle - math.pi / 2) * 100), int(cy - math.sin(angle - math.pi / 2) * 100)),
                 (0, 0, 0), 1)

        tracker.array_to_send.data = [w, cx, angle]
        ##########CREATE_THE_GOAL_POINT###########
        if tracker.in_line  and not tracker.searching_wait: #and tracker.new_goal

            tracker.goal.position.x = h - cy
            tracker.goal.position.y = (w / 2) - cx
            tracker.goal.position.z = 0.0

            ori = tf.transformations.quaternion_from_euler(0.0, 0.0, angle, 'sxyz')

            tracker.goal.orientation.x = ori[0]
            tracker.goal.orientation.y = ori[1]
            tracker.goal.orientation.z = ori[2]
            tracker.goal.orientation.w = ori[3]

            goal_pose_pub.publish(tracker.goal)
            tracker.new_goal = False
            # goal_pose_pub.publish(tracker.goal)
    else:

        tracker.line_detected = False
        rospy.loginfo("Linea no detectada")

        if area_end > 6000:

            tracker.line_detected = True
```

```
            tracker.end_detected = True

            rospy.loginfo("Punto final detectado")

            M_end = cv2.moments(c_end)
            cx = int(M_end['m10'] / M_end['m00'])
            cy = int(M_end['m01'] / M_end['m00'])

            cv2.drawContours(image, c_end, -1, (0, 0, 0), 1)

            [vx, vy, x, y] = cv2.fitLine(c_end, cv2.DIST_L2, 0, 0.01, 0.01)

            angle = math.atan2(vy, vx)
            if angle < 0:
               angle = angle + math.pi / 2
            else:
               angle = angle - math.pi / 2

            angle *= -1
            cv2.line(image,
                  (int(cx - math.cos(angle - math.pi / 2) * 100), int(cy + math.sin(angle - math.pi / 2) * 100)),
                  (int(cx + math.cos(angle - math.pi / 2) * 100), int(cy - math.sin(angle - math.pi / 2) * 100)),
                  (0, 0, 0), 1)

            tracker.goal.position.x = h - cy
            tracker.goal.position.y = (w / 2) - cx
            tracker.goal.position.z = 0.0

            ori = tf.transformations.quaternion_from_euler(0.0, 0.0, angle, 'sxyz')

            tracker.goal.orientation.x = 0.0
            tracker.goal.orientation.y = 0.0
            tracker.goal.orientation.z = 0.0
            tracker.goal.orientation.w = 1.0

            end_detected_pub.publish(tracker.end_detected)
            line_detected_pub.publish(tracker.line_detected)

            goal_pose_pub.publish(tracker.goal)
         line_detected_pub.publish(tracker.line_detected)
         tracker.array_to_send.data = [w, 0.0, 0.0]

      line_detected_pub.publish(tracker.line_detected)
      line_data_pub.publish(tracker.array_to_send)
      if not tracker.in_line:
         # rospy.loginfo(cx)
         # rospy.loginfo(tracker.i)
         if  tracker.line_detected and tracker.i == 0: #317 < cx < 323
               tracker.i = 1
               tracker.goal.position.x = h - cy + 100
               tracker.goal.position.y = (w / 2) - cx
               tracker.goal.position.z = 0.0

               tracker.goal.orientation.x = 0.0
               tracker.goal.orientation.y = 0.0
               tracker.goal.orientation.z = 0.0
               tracker.goal.orientation.w = 1.0

               # goal_pose_pub.publish(tracker.goal)
               # tracker.new_goal = False
```

```
        elif tracker.positioned and tracker.i == 1:

            tracker.i = 2
            response_pub.publish(True)
            tracker.in_line = True
        in_line_pub.publish(tracker.i)
        goal_pose_pub.publish(tracker.goal)
        tracker.new_goal = False


    cv2.imshow("mask", mask)
    cv2.imshow("mask_end", mask_end)
    cv2.imshow("output", image)
    cv2.waitKey(3)

def listener():
    rospy.Subscriber('/campero/campero_front_ptz_camera/image_raw',Image, image_callback)
    # rospy.Subscriber('/campero/goal_complete', Bool, tracker.new_goal_callback)
    rospy.Subscriber('/campero/robotnik_base_control/odom', Odometry, tracker.actual_ori_callback)
    rospy.Subscriber('/campero/positioned', Bool, tracker.positioned_in_line_callback)
    rospy.Subscriber('/campero/oriented', Bool, tracker.oriented_in_line_callback)
    rospy.Subscriber('/campero/searching_line', Bool, tracker.searching_line_callback)
    rospy.Subscriber('/campero/final_goal_complete', Bool, tracker.final_callback)

if __name__ == '__main__':
    try:
        rospy.init_node('line_tracker')
        line_detected_pub = rospy.Publisher('/campero/line_detected',Bool, queue_size = 10)
        line_data_pub = rospy.Publisher('/campero/line_data',Float32MultiArray, queue_size = 1)
        end_detected_pub = rospy.Publisher('/campero/end_detected',Bool, queue_size = 10)
        goal_pose_pub = rospy.Publisher('/campero/goal_pose',Pose, queue_size = 10)
        in_line_pub = rospy.Publisher('/campero/in_line',Int8, queue_size = 10)
        response_pub = rospy.Publisher('/campero/comienzo', Bool, queue_size=10)
        tracker = Tracker()
        listener()
        while not tracker.end and not rospy.is_shutdown():

            rospy.sleep(0.05)

    except rospy.ROSInterruptException:
        rospy.loginfo("Detection test finished.")
```

El siguiente código es el encargado de generar las localizaciones, por medio de transformaciones de un sistema de referencia relativo a un sistema de referencia absoluto, con la información de la posición y orientación de los puntos obtenidos en la imagen.

- Line_marker.py

```
#!/usr/bin/env python
# coding=utf-8
"""line_marker.py V. 1.0 07/07/2021
   Autor: Oscar Alejandro
"""
import rospy, cv2, cv_bridge, numpy, math, tf, tf2_ros, tf2_msgs.msg
from sensor_msgs.msg import Image, CameraInfo
from geometry_msgs.msg import Twist,PoseStamped,Pose,TransformStamped
from std_msgs.msg import Bool, Int16
from tf.transformations import quaternion_from_euler


class Marker_point():
   def __init__(self):
      self.transformStamped = TransformStamped()
      self.transformStamped_ref = TransformStamped()
      self.id = 1
      self.rescale_x = 333.0 / 1.0
      self.rescale_y = 320.0 / 0.5
      self.new_frame = False
      self.end_frame = False
      self.end = False
      self.frame_array = ['/campero_base_footprint']
      self.frame_ref_array = ['/campero_odom']
      self.localization_array = []
      self.localization_ref_array = []
      self.listn = tf.TransformListener()
      self.br = tf.TransformBroadcaster()
      self.pub_tf = rospy.Publisher("/tf", tf2_msgs.msg.TFMessage, queue_size=1)
      self.searching_wait = False
      self.Pose = Pose()
      self.last_frame = False

   def new_frame_callback(self,msg):
      self.new_frame = msg
   def end_frame_callback(self,msg):
      if not self.end_frame:
         self.end_frame = msg
   def final_callback(self,msg):
      self.end = msg
   def searching_line_callback(self,msg):
      self.searching_wait = msg.data

   def Marker_callback(self,msg):
      self.Pose = msg

def Marker_creation(msg):
   if not MP.searching_wait:
      rospy.loginfo("<nuevo frame")
      if not MP.end:
         if MP.end_frame:
```

```
            # base_marker(Pose)
            ####################################
            child_frame_id = 'goal_%s' % (MP.id - 1)
            MP.frame_array.append(child_frame_id)

            MP.transformStamped.header.stamp = rospy.Time.now()
            MP.transformStamped.header.frame_id = MP.frame_array[0]
            MP.transformStamped.child_frame_id = MP.frame_array[MP.id]

            MP.transformStamped.transform.translation.x = (msg.position.x / MP.rescale_x)
            MP.transformStamped.transform.translation.y = msg.position.y / MP.rescale_y
            MP.transformStamped.transform.translation.z = msg.position.z
            MP.transformStamped.transform.rotation.x = msg.orientation.x
            MP.transformStamped.transform.rotation.y = msg.orientation.y
            MP.transformStamped.transform.rotation.z = msg.orientation.z
            MP.transformStamped.transform.rotation.w = msg.orientation.w
            MP.localization_array.append(MP.transformStamped)

            br2.sendTransform(MP.transformStamped)
            ########################################
            MP.listn.waitForTransform('/campero_odom', MP.frame_array[MP.id], rospy.Time(),
rospy.Duration(10))
            (T, R) = MP.listn.lookupTransform('/campero_odom', MP.frame_array[MP.id], rospy.Time())

            child_frame_id = 'end_goal_0'
            MP.frame_ref_array.append(child_frame_id)

            MP.transformStamped_ref.header.stamp = rospy.Time.now()
            MP.transformStamped_ref.header.frame_id = MP.frame_ref_array[0]
            MP.transformStamped_ref.child_frame_id = MP.frame_ref_array[MP.id]

            MP.transformStamped_ref.transform.translation.x = T[0]
            MP.transformStamped_ref.transform.translation.y = T[1]
            MP.transformStamped_ref.transform.translation.z = T[2]

            MP.transformStamped_ref.transform.rotation.x = R[0]
            MP.transformStamped_ref.transform.rotation.y = R[1]
            MP.transformStamped_ref.transform.rotation.z = R[2]
            MP.transformStamped_ref.transform.rotation.w = R[3]

            MP.localization_ref_array.append(MP.transformStamped_ref)
            br2.sendTransform(MP.transformStamped_ref)
            ################################################
            euler = quat_to_euler(R)
            with open('/home/oscar/Documentos/Documentos_TFG/Datos/Posicion_a_alcanzar_Campero',
'a') as f:
                f.write('       ' + str(T[0]) + '       ' + str(T[1]) + '       ' + str(euler[2]) + '       Posición final\n')
            ################################################
            ########################################
            # end_marker()
            MP.end_frame = True
            end_frame_pub.publish(MP.end_frame)
            MP.last_frame = True
        else:
            MP.end_frame = False
            if MP.new_frame:

                # base_marker(Pose)
                ####################################
                child_frame_id = 'goal_%s' % (MP.id - 1)
```

```
            MP.frame_array.append(child_frame_id)

            MP.transformStamped.header.stamp = rospy.Time.now()
            MP.transformStamped.header.frame_id = MP.frame_array[0]
            MP.transformStamped.child_frame_id = MP.frame_array[MP.id]


            MP.transformStamped.transform.translation.x = (msg.position.x / MP.rescale_x)
            MP.transformStamped.transform.translation.y = msg.position.y / MP.rescale_y
            MP.transformStamped.transform.translation.z = msg.position.z
            MP.transformStamped.transform.rotation.x = msg.orientation.x
            MP.transformStamped.transform.rotation.y = msg.orientation.y
            MP.transformStamped.transform.rotation.z = msg.orientation.z
            MP.transformStamped.transform.rotation.w = msg.orientation.w
            MP.localization_array.append(MP.transformStamped)

            br2.sendTransform(MP.transformStamped)
            ######################################
            # rospy.loginfo("transformStamped: %s", MP.localization_array)
            # rospy.loginfo("frame_array: %s", MP.frame_array)

            # ref_marker()
            ######################################
            MP.listn.waitForTransform('/campero_odom', MP.frame_array[MP.id], rospy.Time(),
rospy.Duration(10))
            (T, R) = MP.listn.lookupTransform('/campero_odom', MP.frame_array[MP.id],
rospy.Time())

            child_frame_id = 'ref_goal_%s' % (MP.id - 1)
            MP.frame_ref_array.append(child_frame_id)

            MP.transformStamped_ref.header.stamp = rospy.Time.now()
            MP.transformStamped_ref.header.frame_id = MP.frame_ref_array[0]
            MP.transformStamped_ref.child_frame_id = MP.frame_ref_array[MP.id]

            MP.transformStamped_ref.transform.translation.x = T[0]
            MP.transformStamped_ref.transform.translation.y = T[1]
            MP.transformStamped_ref.transform.translation.z = T[2]

            MP.transformStamped_ref.transform.rotation.x = R[0]
            MP.transformStamped_ref.transform.rotation.y = R[1]
            MP.transformStamped_ref.transform.rotation.z = R[2]
            MP.transformStamped_ref.transform.rotation.w = R[3]

            MP.localization_ref_array.append(MP.transformStamped_ref)
            br2.sendTransform(MP.transformStamped_ref)
            # ######################################
            id_pub.publish(MP.id - 1)
            MP.id += 1
            ##################################################
            euler = quat_to_euler(R)
            with
open('/home/oscar/Documentos/Documentos_TFG/Datos/Posicion_a_alcanzar_Campero', 'a') as f:
                f.write('       ' + str(T[0]) + '       ' + str(T[1]) + '       ' + str(math.degrees(euler[2])) + '
\n')
            ##################################################
            new_frame_pub.publish(MP.new_frame)
            # response_pub.publish(MP.new_frame)
            MP.new_frame = False
            # rospy.loginfo("Pose_x: %d", MP.transformStamped.transform.translation.x)
```

```
                # rospy.loginfo("Pose_y: %d", MP.transformStamped.transform.translation.y)
                # rospy.loginfo("Pose_z: %d", MP.transformStamped.transform.translation.z)
                # rospy.loginfo("transformStamped: %s", MP.localization_ref_array)
                # rospy.loginfo("frame_array: %s", MP.frame_ref_array)
                # rospy.loginfo("Initial goal published! Goal ID is: %d", MP.id)
            else:
                MP.new_frame = False
        else:
            MP.new_frame = False
            MP.end_frame = False

def quat_to_euler(quat):
    quaternion = tuple([quat[0], quat[1], quat[2], quat[3]])
    euler = tf.transformations.euler_from_quaternion(quaternion)
    return euler

def listener():
    rospy.Subscriber('/campero/goal_pose', Pose, MP.Marker_callback)
    rospy.Subscriber('/campero/goal_complete', Bool, MP.new_frame_callback)
    rospy.Subscriber('/campero/end_detected', Bool, MP.end_frame_callback)
    rospy.Subscriber('/campero/final_goal_complete', Bool, MP.final_callback)
    rospy.Subscriber('/campero/searching_line', Bool, MP.searching_line_callback)


if __name__ == '__main__':

    try:
        rospy.init_node('line_marker')
        # listn = tf.TransformListener()
        br2 = tf2_ros.StaticTransformBroadcaster()
        id_pub = rospy.Publisher('/campero/goal_id', Int16 , queue_size=1)
        new_frame_pub = rospy.Publisher('/campero/new_frame', Bool , queue_size=10)
        # response_pub = rospy.Publisher('/campero/comienzo', Bool , queue_size=10)
        end_frame_pub = rospy.Publisher('/campero/end_frame', Bool , queue_size=10)
        MP = Marker_point()
        listener()
        ##################################################
        with open('/home/oscar/Documentos/Documentos_TFG/Datos/Posicion_a_alcanzar_Campero', 'w')
as f:
            f.write('Posición y orientación \n')
            f.write('      X                Y                EULER      \n')
        ##################################################
        while not MP.end and not rospy.is_shutdown():
            if not MP.last_frame:
                Marker_creation(MP.Pose)
            rospy.sleep(1.0)

            if MP.end_frame:
                rospy.loginfo("FINAL")

            # br2.sendTransform(MP.transformStamped)
            # if MP.id > 1:
            #     br2.sendTransform(MP.transformStamped_ref)




    except rospy.ROSInterruptException:
        rospy.loginfo("Detection test finished.")
```

El código que tenemos a continuación es el código de navegación de seguimiento de línea con orientación prioritaria de línea. En el cual se trabajan las distintas situaciones que pueden encontrarse, como son la perdida de línea y su búsqueda, la navegación a un punto final, la evasión de obstáculos o la navegación por la línea normal.

- Line_follower.py

```
#!/usr/bin/env python
# coding=utf-8
"""line_follower.py V. 1.0 25/06/2021
   Autor: Oscar Alejandro
"""

import rospy, cv2, numpy, tf, math
import actionlib
from geometry_msgs.msg import Twist, PoseWithCovarianceStamped
from move_base_msgs.msg import MoveBaseAction, MoveBaseGoal
from tf.transformations import quaternion_from_euler
from visualization_msgs.msg import Marker
from std_msgs.msg import Bool, Int16, Int8, Float32MultiArray
from sensor_msgs.msg import LaserScan
from nav_msgs.msg import Odometry

P_angle = 1.0 #1.3
P_linear = 0.004

class Follower():
   def __init__(self):
      # Para la velocidad del robot
      self.vel = Twist()
      self.pose = PoseWithCovarianceStamped()
      self.listn = tf.TransformListener()
      self.vel.linear.x = 0.0
      self.vel.linear.y = 0.0
      self.vel.linear.z = 0.0

      self.vel.angular.x = 0.0
      self.vel.angular.y = 0.0
      self.vel.angular.z = 0.0

      self.xmax = 0.0
      self.xval = 0.0
      self.angle = 0.0

      self.line_detected = False
      self.goal_complete = False
      self.nav = False
      self.end_nav = False
      self.end_goal = False
      self.id = 0
      self.fin = False
      self.Start = True
      self.step_end = False
      ########################
      self.obstacle = False
```

```
        self.fase = 0
        self.i = 0
        self.in_line = 0
        self.orientation = None
        self.position = None
        self.aux_orientation = None
        self.aux_position = None
        self.reori = 0
        self.search = False
        self.correction = False

    def nav_callback(self,msg):
        self.nav = msg.data
        self.step_end = False
    def end_nav_callback(self,msg):
        self.end_nav = msg.data
    def estimated_pose_callback(self,msg):
        self.pose = msg
    def id_callback(self,msg):
        self.id = msg.data
    def Has_started_callback(self,msg):
        self.Start = False
    def obst_dist_callback(self,msg):
        aux=min(msg.ranges)

        if ((aux < 1.1 and 0 <= msg.ranges.index(aux) < 150) or (aux < 0.9 and 150 <=
msg.ranges.index(aux) < 199) or (aux < 0.7 and 199 <= msg.ranges.index(aux) < 349))and not
(self.obstacle) and not (self.search):

            self.obstacle = True
            follower.aux_position = [follower.position.x, follower.position.y, follower.position.z]

            follower.aux_orientation = [follower.orientation.x, follower.orientation.y, follower.orientation.z,
                    follower.orientation.w]

    def actual_pose_callback(self,msg):
        self.orientation = msg.pose.pose.orientation
        self.position = msg.pose.pose.position
    def euclidean_distance(self, dist):
        return math.sqrt(pow(dist[0], 2) +
                pow(dist[1], 2))
    def quat_to_euler(self, quat):
        quaternion = tuple([quat[0], quat[1], quat[2], quat[3]])
        euler = tf.transformations.euler_from_quaternion(quaternion)
        return euler
    def robot_in_line_callback(self,msg):
        self.in_line = msg.data
        # ori_list = [follower.orientation.x, follower.orientation.y, follower.orientation.z,
        #             follower.orientation.w]
        # follower.aux_orientation = follower.quat_to_euler(ori_list)
        # rospy.loginfo(self.in_line)
    def line_data_control_callback(self,msg):
        self.xmax = msg.data[0]
        self.xval = msg.data[1]
        self.angle = msg.data[2]

def Goal_callback(msg):
    follower.line_detected = msg.data
    if  not follower.Start:
        if msg.data == False and follower.reori == 0 and not follower.end_nav:
```

```
            rospy.loginfo('DEBO BUSCAR LA LINEA')
            ori_list = [follower.orientation.x, follower.orientation.y, follower.orientation.z,
                    follower.orientation.w]
            follower.aux_orientation = follower.quat_to_euler(ori_list)
            searching_line_pub.publish(True)
            follower.search = True
            follower.reori = 1

def Start_program():

    if follower.in_line == 0:
        # rospy.loginfo("0")
        follower.vel.angular.z = -0.5
    elif follower.in_line == 1:
        # rospy.loginfo("1")
        follower.vel.linear.x = 0.0
        follower.vel.angular.z = 0.0
        if follower.nav:
            navigation_ByMarks()
        elif follower.step_end:
            follower.vel.linear.x = 0.0
            positioned_in_line_pub.publish(True)
    elif follower.in_line == 2:
        # rospy.loginfo("2")
        follower.vel.angular.z = 0.0
        goal_arr_pub.publish(True)
    cmd_vel_pub.publish(follower.vel)

def end_nav():


    rospy.loginfo("Final de la navegacion")

    follower.listn.waitForTransform('/end_goal_0', '/campero_base_footprint', rospy.Time(),
                            rospy.Duration(10))
    (trans, rot) = follower.listn.lookupTransform('/campero_base_footprint', '/end_goal_0', rospy.Time(0))
    rospy.loginfo(rot)
    distancia = follower.euclidean_distance(trans)

    if distancia > 0.15:

        linear = 0.5
    else:
        # rospy.loginfo("Estoy en posicion")

        linear = 0.0
        follower.fin = True
        final_complete_pub.publish(follower.fin)
    # linear = 0.35
    # angular = 0.35 * euler
    #
    # if distancia < 0.2:
    #     # rospy.loginfo("Estoy en posicion")
    #     angular = 0.0
    #     linear = 0.0
    #     follower.fin = True
    #     final_complete_pub.publish(follower.fin)

    follower.vel.linear.x = linear
    follower.vel.linear.y = 0.0
```

```
    follower.vel.angular.z = 0.0

##########################################################################################
###
    with open('/home/oscar/Documentos/Documentos_TFG/Datos/Velocidad_Campero', 'a') as f:
        f.write('       ' + str(follower.vel.linear.x) + '       ' + str(follower.vel.linear.y) + '       ' + str(
            follower.vel.angular.z) + '       Velocidad final\n')

##########################################################################################
###
    cmd_vel_pub.publish(follower.vel)

def navigation_ByMarks():
    # rospy.loginfo("No navegacion final")
    # rospy.loginfo("Estoy navegando")

    follower.listn.waitForTransform('/ref_goal_%s' % follower.id, '/campero_base_footprint',
rospy.Time(),
                                    rospy.Duration(25))
    (trans, rot) = follower.listn.lookupTransform('/campero_base_footprint', '/ref_goal_%s' % follower.id,
                                          rospy.Time(0))

    distancia = follower.euclidean_distance(trans)
    euler = follower.quat_to_euler(rot)

    if distancia > 0.05:
        angular = 15.0 * math.atan2(trans[1], trans[0])
        # linear = 0.5 * distancia
        linear = 1.0


    elif abs(euler[2]) > 0.05:
        if euler[2] > 0:
            # angular = 0.6
            angular = 1.0
        else:
            # angular = -0.6
            angular = -1.0
        linear = 0.01
    else:
        # rospy.loginfo("Estoy en posicion")
        angular = 0.0
        linear = 1.0
        follower.step_end = True
        follower.nav = False

    follower.vel.linear.x = linear
    follower.vel.angular.z = angular

##########################################################################################
###
    with open('/home/oscar/Documentos/Documentos_TFG/Datos/Velocidad_Campero', 'a') as f:
        f.write('       ' + str(follower.vel.linear.x) + '       ' + str(follower.vel.linear.y) + '       ' + str(
            follower.vel.angular.z) + '       \n')

##########################################################################################
###
    cmd_vel_pub.publish(follower.vel)

def navigation_Vcte():
```

```
    if (follower.angle and follower.xval) == 0.0:
        follower.vel.angular.z = 0.0
        follower.vel.linear.y = 0.0
    else:
        follower.vel.angular.z = P_angle * (follower.angle) + P_linear * -(follower.xval - follower.xmax/2)
        follower.vel.linear.y = P_linear * -(follower.xval - follower.xmax / 2)
    follower.vel.linear.x = 0.5


##########################################################################################
###
    with open('/home/oscar/Documentos/Documentos_TFG/Datos/Velocidad_Campero', 'a') as f:
        f.write('       '+str(follower.vel.linear.x)+'       '+str(follower.vel.linear.y)+'
'+str(follower.vel.angular.z)+'       \n')

##########################################################################################
###
    cmd_vel_pub.publish(follower.vel)

def obstacle_avoidance():

    if follower.fase == 0:
        # rospy.loginfo("Obstacle detected")
        # rospy.loginfo(follower.obstacle)
        follower.vel.linear.x = 0.0
        follower.vel.linear.y = 0.0
        follower.vel.linear.z = 0.0
        follower.vel.angular.x = 0.0
        follower.vel.angular.y = 0.0
        follower.vel.angular.z = 0.0

        cmd_vel_pub.publish(follower.vel)

        follower.fase = 1

    elif follower.fase == 1:


        follower.vel.linear.x = -0.4
        cmd_vel_pub.publish(follower.vel)
        rospy.loginfo(follower.position.x)
        rospy.loginfo(follower.aux_position[0] + 0.5)
        if follower.position.x <= follower.aux_position[0] + 0.5:
            follower.vel.linear.x = 0.0
            cmd_vel_pub.publish(follower.vel)
            follower.fase = 2
            follower.i = 0

    elif follower.fase == 2:
        # rospy.loginfo("Moving 2 m forward")
        result = movebase_client()
        if result:
            # rospy.loginfo("Goal execution done!")
            follower.fase = 0
            follower.obstacle = False
            goal_arr_pub.publish(True)

def movebase_client():

    client = actionlib.SimpleActionClient('/campero/move_base',MoveBaseAction)
```

```
    client.wait_for_server()

    goal = MoveBaseGoal()
    goal.target_pose.header.frame_id = "campero_map"
    goal.target_pose.header.stamp = rospy.Time.now()

    ori_euler = follower.quat_to_euler(follower.aux_orientation)

    goal.target_pose.pose.position.x = 3.5 * math.cos(ori_euler[2]) + follower.position.x
    goal.target_pose.pose.position.y = 3.5 * math.sin(ori_euler[2]) + follower.position.y

    goal.target_pose.pose.orientation.x = follower.aux_orientation[0]
    goal.target_pose.pose.orientation.y = follower.aux_orientation[1]
    goal.target_pose.pose.orientation.z = follower.aux_orientation[2]
    goal.target_pose.pose.orientation.w = follower.aux_orientation[3]

    client.send_goal(goal)
    wait = client.wait_for_result()
    if not wait:
        rospy.logerr("Action server not available!")
        rospy.signal_shutdown("Action server not available!")
    else:
        return client.get_result()

def search_line():
    follower.vel.linear.x = 0.0
    follower.vel.linear.y = 0.0
    # rospy.loginfo("Girando para detectar linea")
    if follower.reori == 1:
        # rospy.loginfo("Reorientacion 1")
        ori_list = [follower.orientation.x, follower.orientation.y, follower.orientation.z,
                    follower.orientation.w]
        ori_euler = follower.quat_to_euler(ori_list)

        angle_start = follower.aux_orientation[2]

        angle_final = normalization(follower.aux_orientation[2] + ((89 * numpy.pi) / 180))

        # follower.vel.angular.z = 0.3
        follower.vel.angular.z = 0.7
        if follower.line_detected:
            follower.vel.angular.z = 0.0
            follower.reori = 0
            follower.search = False

        elif (angle_start >= 0.0 and angle_final >= 0.0 and ori_euler[2] > angle_final) or (angle_start <= 0.0
and angle_final <= 0.0 and ori_euler[2] > angle_final) or (angle_start <= 0.0 and angle_final >= 0.0 and
ori_euler[2] > angle_final):
            follower.vel.angular.z = 0.0
            if follower.line_detected:
                follower.reori = 0
                follower.search = False
            else:
                ori_list = [follower.orientation.x, follower.orientation.y, follower.orientation.z,
                            follower.orientation.w]
                follower.aux_orientation = follower.quat_to_euler(ori_list)
                follower.reori = 2
        elif  (angle_start >=0.0 and angle_final <=0.0 ):
            if (ori_euler[2] >= 0.0):
                angle_now = - ori_euler[2]
```

```
            if (angle_now > abs(angle_final)):
                follower.vel.angular.z = 0.0
                if follower.line_detected:
                    follower.reori = 0
                    follower.search = False
                else:
                    ori_list = [follower.orientation.x, follower.orientation.y, follower.orientation.z,
                            follower.orientation.w]
                    follower.aux_orientation = follower.quat_to_euler(ori_list)
                    follower.reori = 2
        else:
            angle_now = abs(ori_euler[2])
            if (angle_now < abs(angle_final)):
                follower.vel.angular.z = 0.0
                if follower.line_detected:
                    follower.reori = 0
                    follower.search = False
                else:
                    ori_list = [follower.orientation.x, follower.orientation.y, follower.orientation.z,
                            follower.orientation.w]
                    follower.aux_orientation = follower.quat_to_euler(ori_list)
                    follower.reori = 2

    else:
        follower.vel.angular.z = 0.7

elif follower.reori == 2:
    rospy.loginfo("Reorientacion 2")
    ori_list = [follower.orientation.x, follower.orientation.y, follower.orientation.z,
            follower.orientation.w]
    ori_euler = follower.quat_to_euler(ori_list)

    rospy.loginfo(ori_euler[2])
    rospy.loginfo(follower.aux_orientation[2])

    angle_start = follower.aux_orientation[2]
    angle_now = 0.0
    angle_final = normalization(follower.aux_orientation[2] - (numpy.pi))# ((179 * numpy.pi) / 180))

    rospy.loginfo(angle_final)

    follower.vel.angular.z = -0.7
    if follower.line_detected:
        follower.vel.angular.z = 0.0
        follower.reori = 0
        follower.search = False

    elif (angle_start >=0.0 and angle_final<=0.0 and ori_euler[2] < angle_final):
        follower.vel.angular.z = 0.0
        if follower.line_detected:
            follower.reori = 0
            follower.search = False
        else:
            ori_list = [follower.orientation.x, follower.orientation.y, follower.orientation.z,
                    follower.orientation.w]
            follower.aux_orientation = follower.quat_to_euler(ori_list)
            follower.reori = 3

    elif (angle_start <=0.0 and angle_final >=0.0):
        if (ori_euler[2] >= 0.0):
```

```
            angle_now = ori_euler[2] - numpy.pi
        else:
            angle_now = ori_euler[2] + numpy.pi
        if angle_now < (angle_final - numpy.pi):
            follower.vel.angular.z = 0.0
            if follower.line_detected:
                follower.reori = 0
                follower.search = False
            else:
                ori_list = [follower.orientation.x, follower.orientation.y, follower.orientation.z,
                        follower.orientation.w]
                follower.aux_orientation = follower.quat_to_euler(ori_list)
                follower.reori = 3
    else:
        follower.vel.angular.z = -0.7

elif follower.reori == 3:
    # rospy.loginfo("Reorientacion 3")
    ori_list = [follower.orientation.x, follower.orientation.y, follower.orientation.z,
            follower.orientation.w]
    ori_euler = follower.quat_to_euler(ori_list)

    angle_start = follower.aux_orientation[2]
    angle_now = 0.0
    angle_final = normalization(follower.aux_orientation[2] - ((89 * numpy.pi) / 180))

    follower.vel.angular.z = -0.7
    if follower.line_detected:
        follower.vel.angular.z = 0.0
        follower.reori = 0
        follower.search = False

    elif (ori_euler[2] < angle_final) and ((angle_start >= 0.0 and angle_final >= 0.0) or (angle_start >=
0.0 and angle_final <= 0.0 ) or (angle_start <= 0.0 and angle_final <= 0.0 )) :
        follower.vel.angular.z = 0.0
        follower.reori = 0
        follower.search = False
    elif (angle_start <= 0.0 and angle_final >= 0.0):
        if (ori_euler[2] >= 0.0):
            angle_now = ori_euler[2] - numpy.pi
        else:
            angle_now = ori_euler[2] + numpy.pi
        if angle_now < (angle_final - numpy.pi):
            follower.vel.angular.z = 0.0
            follower.reori = 0
            follower.search = False
    else:
        follower.vel.angular.z = -0.7

cmd_vel_pub.publish(follower.vel)

# def waiting():
#     rospy.loginfo("No navego")
#     follower.vel.linear.x = 0.0
#     # follower.vel.linear.x = 1.0
#     follower.vel.linear.y = 0.0
#     follower.vel.linear.z = 0.0
#     follower.vel.angular.x = 0.0
#     follower.vel.angular.y = 0.0
#     follower.vel.angular.z = 0.0
```

```
#     cmd_vel_pub.publish(follower.vel)

def normalization(angle):
    if angle > numpy.pi:
        angle = angle - (2 * numpy.pi)
    elif angle <= -numpy.pi:
        angle = angle + (2 * numpy.pi)
    return angle

def listener():
    rospy.Subscriber('/campero/line_detected',Bool, Goal_callback)
    rospy.Subscriber('/campero/goal_id', Int16, follower.id_callback)
    rospy.Subscriber('/campero/new_frame', Bool, follower.nav_callback)
    rospy.Subscriber('/campero/end_frame', Bool, follower.end_nav_callback)
    rospy.Subscriber('/campero/amcl_pose', PoseWithCovarianceStamped,
follower.estimated_pose_callback)
    rospy.Subscriber('/campero/comienzo', Bool, follower.Has_started_callback)
    rospy.Subscriber('/campero/front_laser/scan', LaserScan, follower.obst_dist_callback)
    rospy.Subscriber('/campero/robotnik_base_control/odom', Odometry, follower.actual_pose_callback)
    rospy.Subscriber('/campero/in_line', Int8, follower.robot_in_line_callback)
    rospy.Subscriber('/campero/line_data', Float32MultiArray, follower.line_data_control_callback)

if __name__ == '__main__':
    try:
        rospy.init_node('line_follower')
        # listn = tf.TransformListener()
        cmd_vel_pub = rospy.Publisher('/campero/cmd_vel', Twist, queue_size=1)
        searching_line_pub = rospy.Publisher('/campero/searching_line', Bool, queue_size=10)
        goal_arr_pub = rospy.Publisher('/campero/goal_complete', Bool, queue_size=10)
        final_complete_pub = rospy.Publisher('/campero/final_goal_complete', Bool, queue_size=10)
        positioned_in_line_pub = rospy.Publisher('/campero/positioned', Bool, queue_size=10)
        oriented_in_line_pub = rospy.Publisher('/campero/oriented', Bool, queue_size=10)
        follower = Follower()
        listener()
        i = 0
        ##################################################
        with open('/home/oscar/Documentos/Documentos_TFG/Datos/Posicion_Campero', 'w') as f:
            f.write('Posición y orientación del Campero\n')
            f.write('       X                Y                EULER      \n')
        with open('/home/oscar/Documentos/Documentos_TFG/Datos/Velocidad_Campero', 'w') as f:
            f.write('Velocidad en navegacion \n')
            f.write('       v_X       v_Y         w      \n')
        ##################################################
        while not follower.fin and not rospy.is_shutdown():

            if follower.Start:
                Start_program()
                if follower.in_line == 1 and i < 1:
                    goal_arr_pub.publish(True)
                    i += 1
                # rospy.loginfo("Comienzo")
            else:
                ##################################################
                ori_list = [follower.orientation.x, follower.orientation.y, follower.orientation.z,
                            follower.orientation.w]
                euler = follower.quat_to_euler(ori_list)
                with open('/home/oscar/Documentos/Documentos_TFG/Datos/Posicion_Campero', 'a') as f:
                    f.write(
                        '       ' + str(follower.position.x) + '       ' + str(follower.position.y) + '       ' + str(
                            math.degrees(euler[2])) + '       \n')
```

```
        ##################################################
        if not follower.fin:
            if follower.obstacle and not follower.search:
                rospy.loginfo("Obstaculo detectado")
                obstacle_avoidance()
            elif follower.end_nav:
                # rospy.loginfo("Navegando al punto final")
                end_nav()
            elif follower.search:
                rospy.loginfo("Buscando la linea")
                search_line()
            elif not follower.obstacle and not follower.end_nav and not follower.search:
                navigation_Vcte()
                goal_arr_pub.publish(True)
                searching_line_pub.publish(False)
                # waiting()
                # rospy.sleep(0.25)
            else:
                rospy.loginfo("Esperando")
                goal_arr_pub.publish(True)
                searching_line_pub.publish(False)

            # rospy.loginfo(follower.id)

except rospy.ROSInterruptException:
    rospy.loginfo("Detection test finished.")
```

El código que tenemos a continuación es el código de navegación de seguimiento de línea por marcas a velocidad constante. En el cual se trabajan las distintas situaciones que pueden encontrarse, como son la perdida de línea y su búsqueda, la navegación a un punto final, la evasión de obstáculos o la navegación por la línea normal.

- Line_follower_Marks.py

```
#!/usr/bin/env python
# coding=utf-8
"""line_follower.py V. 1.0 25/06/2021
   Autor: Oscar Alejandro
"""

import rospy, cv2, numpy, tf, math
import actionlib
from geometry_msgs.msg import Twist, PoseWithCovarianceStamped
from move_base_msgs.msg import MoveBaseAction, MoveBaseGoal
from tf.transformations import quaternion_from_euler
from visualization_msgs.msg import Marker
from std_msgs.msg import Bool, Int16, Int8
from sensor_msgs.msg import LaserScan
from nav_msgs.msg import Odometry
class Follower():
   def __init__(self):
      # Para la velocidad del robot
      self.vel = Twist()
      self.pose = PoseWithCovarianceStamped()
      self.listn = tf.TransformListener()
      self.vel.linear.x = 0.0
      self.vel.linear.y = 0.0
      self.vel.linear.z = 0.0

      self.vel.angular.x = 0.0
      self.vel.angular.y = 0.0
      self.vel.angular.z = 0.0


      self.line_detected = False
      self.goal_complete = False
      self.nav = False
      self.end_nav = False
      self.end_goal = False
      self.id = 0
      self.id_aux = 0
      self.fin = False
      self.Start = True
      self.step_end = False
      ##########################
      self.obstacle = False
      self.fase = 0
      self.i = 0
      self.in_line = 0
      self.orientation = None
      self.position = None
      self.aux_orientation = None
```

```
        self.aux_position = None
        self.reori = 0
        self.search = False
        self.correction = False

    def nav_callback(self,msg):
        self.nav = msg.data
        self.step_end = False
    def end_nav_callback(self,msg):
        self.end_nav = msg.data
    def estimated_pose_callback(self,msg):
        self.pose = msg
    def id_callback(self,msg):
        if self.Start:
            self.id = msg.data
        else:
            self.id_aux = msg.data
    def Has_started_callback(self,msg):
        self.Start = False
    def obst_dist_callback(self,msg):
        aux=min(msg.ranges)

        if ((aux < 1.1 and 0 <= msg.ranges.index(aux) < 150) or (aux < 0.9 and 150 <=
msg.ranges.index(aux) < 199) or (aux < 0.7 and 199 <= msg.ranges.index(aux) < 349))and not
(self.obstacle) and not (self.search):

            self.obstacle = True

    def actual_pose_callback(self,msg):
        self.orientation = msg.pose.pose.orientation
        self.position = msg.pose.pose.position
    def euclidean_distance(self, dist):
        return math.sqrt(pow(dist[0], 2) +
                    pow(dist[1], 2))
    def quat_to_euler(self, quat):
        quaternion = tuple([quat[0], quat[1], quat[2], quat[3]])
        euler = tf.transformations.euler_from_quaternion(quaternion)
        return euler
    def robot_in_line(self,msg):
        self.in_line = msg.data
        # ori_list = [follower.orientation.x, follower.orientation.y, follower.orientation.z,
        #             follower.orientation.w]
        # follower.aux_orientation = follower.quat_to_euler(ori_list)
        # rospy.loginfo(self.in_line)
def Goal_callback(msg):
    follower.line_detected = msg.data
    if  not follower.Start:
        if msg.data == False and follower.reori == 0 and not follower.obstacle:
            ori_list = [follower.orientation.x, follower.orientation.y, follower.orientation.z,
                        follower.orientation.w]
            follower.aux_orientation = follower.quat_to_euler(ori_list)
            searching_line_pub.publish(True)
            follower.search = True
            follower.reori = 1

def Start_program():
    if follower.in_line == 0:
        rospy.loginfo("0")
        follower.vel.angular.z = -0.5
    elif follower.in_line == 1:
```

```
        rospy.loginfo("1")
        follower.vel.linear.x = 0.0
        follower.vel.angular.z = 0.0
        if follower.nav:
            navigation()
        elif follower.step_end:
            follower.vel.linear.x = 0.0
            positioned_in_line_pub.publish(True)
    elif follower.in_line == 2:
        rospy.loginfo("2")
        follower.vel.angular.z = 0.0
        goal_arr_pub.publish(True)
    cmd_vel_pub.publish(follower.vel)

def end_nav():
    rospy.loginfo("Final de la navegacion")
    follower.listn.waitForTransform('/end_goal_0', '/campero_base_footprint', rospy.Time(),
                                    rospy.Duration(10))
    (trans, rot) = follower.listn.lookupTransform('/campero_base_footprint', '/end_goal_0', rospy.Time(0))

    distancia = follower.euclidean_distance(trans)

    if distancia > 0.15:

        linear = 0.5
    else:
        # rospy.loginfo("Estoy en posicion")

        linear = 0.0
        follower.fin = True
        final_complete_pub.publish(follower.fin)

    follower.vel.linear.x = linear
    follower.vel.angular.z = 0.0
    cmd_vel_pub.publish(follower.vel)

def navigation():
    # rospy.loginfo("No navegacion final")
    # rospy.loginfo("Estoy navegando")
    rospy.loginfo(follower.id)

    follower.listn.waitForTransform('/ref_goal_%s' % follower.id, '/campero_base_footprint',
rospy.Time(),
                                    rospy.Duration(25))
    (trans, rot) = follower.listn.lookupTransform('/campero_base_footprint', '/ref_goal_%s' % follower.id,
                                          rospy.Time(0))

    distancia = follower.euclidean_distance(trans)
    euler = follower.quat_to_euler(rot)

    linear = 0.35
    angular = 0.65 * math.atan2(trans[1], trans[0])

    if distancia > 0.05 and follower.Start:
        rospy.loginfo("Estoy en posicion de comienzo")
        linear = 0.0
        angular = 0.0
        follower.step_end = True
        follower.nav = False
```

```
    elif distancia > 0.2 and  not follower.Start:
        rospy.loginfo("Estoy en posicion de comienzo")
        follower.id = follower.id_aux

    follower.vel.linear.x = linear
    follower.vel.angular.z = angular

##########################################################################################
###
    with open('/home/oscar/Documentos/Documentos_TFG/Datos/Velocidad_Campero', 'a') as f:
        f.write('       ' + str(follower.vel.linear.x) + '       ' + str(follower.vel.linear.y) + '       ' + str(
            follower.vel.angular.z) + '       \n')

##########################################################################################
###
    cmd_vel_pub.publish(follower.vel)

def obstacle_avoidance():

    if follower.fase == 0:
        # rospy.loginfo("Obstacle detected")
        # rospy.loginfo(follower.obstacle)
        follower.vel.linear.x = 0.0
        follower.vel.linear.y = 0.0
        follower.vel.linear.z = 0.0
        follower.vel.angular.x = 0.0
        follower.vel.angular.y = 0.0
        follower.vel.angular.z = 0.0

        cmd_vel_pub.publish(follower.vel)

        follower.fase = 1

    elif follower.fase == 1:

        # follower.vel.linear.x = -0.4
        # cmd_vel_pub.publish(follower.vel)
        # rospy.loginfo(follower.position.x)
        # rospy.loginfo(follower.aux_position[0] + 0.5)
        follower.listn.waitForTransform('/ref_goal_%s' % follower.id, '/campero_base_footprint',
rospy.Time(),
                                        rospy.Duration(25))
        (trans, rot) = follower.listn.lookupTransform('/campero_base_footprint', '/ref_goal_%s' %
follower.id,
                                                      rospy.Time(0))
        euler = follower.quat_to_euler(rot)
        if abs(euler[2]) > 0.01:
            # rospy.loginfo("Estoy rotando")
            if euler[2] > 0:
                follower.vel.angular.z = 0.5
            else:
                follower.vel.angular.z = -0.5
            follower.vel.linear.x = 0.0
            cmd_vel_pub.publish(follower.vel)
        else:
            follower.vel.linear.x = 0.0
            follower.vel.angular.z = 0.0
            cmd_vel_pub.publish(follower.vel)
            follower.fase = 2
            follower.i = 0
```

```
            follower.aux_position = [follower.position.x, follower.position.y, follower.position.z]

            follower.aux_orientation = [follower.orientation.x, follower.orientation.y, follower.orientation.z,
                                        follower.orientation.w]

    elif follower.fase == 2:
        # rospy.loginfo("Moving 2 m forward")
        result = movebase_client()
        if result:
            # rospy.loginfo("Goal execution done!")
            follower.fase = 0
            follower.obstacle = False
            goal_arr_pub.publish(True)

def movebase_client():

    client = actionlib.SimpleActionClient('/campero/move_base',MoveBaseAction)
    client.wait_for_server()

    goal = MoveBaseGoal()
    goal.target_pose.header.frame_id = "campero_map"
    goal.target_pose.header.stamp = rospy.Time.now()

    ori_euler = follower.quat_to_euler(follower.aux_orientation)

    goal.target_pose.pose.position.x = 3.5 * math.cos(ori_euler[2]) + follower.position.x
    goal.target_pose.pose.position.y = 3.5 * math.sin(ori_euler[2]) + follower.position.y

    goal.target_pose.pose.orientation.x = follower.aux_orientation[0]
    goal.target_pose.pose.orientation.y = follower.aux_orientation[1]
    goal.target_pose.pose.orientation.z = follower.aux_orientation[2]
    goal.target_pose.pose.orientation.w = follower.aux_orientation[3]

    client.send_goal(goal)
    wait = client.wait_for_result()
    if not wait:
        rospy.logerr("Action server not available!")
        rospy.signal_shutdown("Action server not available!")
    else:
        return client.get_result()

def search_line():
    follower.vel.linear.x = 0.0
    follower.vel.linear.y = 0.0
    # rospy.loginfo("Girando para detectar linea")
    if follower.reori == 1:
        # rospy.loginfo("Reorientacion 1")
        ori_list = [follower.orientation.x, follower.orientation.y, follower.orientation.z,
                    follower.orientation.w]
        ori_euler = follower.quat_to_euler(ori_list)

        angle_start = follower.aux_orientation[2]

        angle_final = normalization(follower.aux_orientation[2] + ((89 * numpy.pi) / 180))

        # follower.vel.angular.z = 0.3
        follower.vel.angular.z = 0.7
        if follower.line_detected:
            follower.vel.angular.z = 0.0
            follower.reori = 0
```

```
            follower.search = False

        elif (angle_start >= 0.0 and angle_final >= 0.0 and ori_euler[2] > angle_final) or (angle_start <= 0.0
and angle_final <= 0.0 and ori_euler[2] > angle_final) or (angle_start <= 0.0 and angle_final >= 0.0 and
ori_euler[2] > angle_final):
            follower.vel.angular.z = 0.0
            if follower.line_detected:
                follower.reori = 0
                follower.search = False
            else:
                ori_list = [follower.orientation.x, follower.orientation.y, follower.orientation.z,
                        follower.orientation.w]
                follower.aux_orientation = follower.quat_to_euler(ori_list)
                follower.reori = 2
        elif  (angle_start >=0.0 and angle_final <=0.0 ):
            if (ori_euler[2] >= 0.0):
                angle_now = - ori_euler[2]
                if (angle_now > abs(angle_final)):
                    follower.vel.angular.z = 0.0
                    if follower.line_detected:
                        follower.reori = 0
                        follower.search = False
                    else:
                        ori_list = [follower.orientation.x, follower.orientation.y, follower.orientation.z,
                                follower.orientation.w]
                        follower.aux_orientation = follower.quat_to_euler(ori_list)
                        follower.reori = 2
            else:
                angle_now = abs(ori_euler[2])
                if (angle_now < abs(angle_final)):
                    follower.vel.angular.z = 0.0
                    if follower.line_detected:
                        follower.reori = 0
                        follower.search = False
                    else:
                        ori_list = [follower.orientation.x, follower.orientation.y, follower.orientation.z,
                                follower.orientation.w]
                        follower.aux_orientation = follower.quat_to_euler(ori_list)
                        follower.reori = 2

        else:
            follower.vel.angular.z = 0.7

    elif follower.reori == 2:
        rospy.loginfo("Reorientacion 2")
        ori_list = [follower.orientation.x, follower.orientation.y, follower.orientation.z,
                follower.orientation.w]
        ori_euler = follower.quat_to_euler(ori_list)

        rospy.loginfo(ori_euler[2])
        rospy.loginfo(follower.aux_orientation[2])

        angle_start = follower.aux_orientation[2]
        angle_now = 0.0
        angle_final = normalization(follower.aux_orientation[2] - (numpy.pi))# ((179 * numpy.pi) / 180))

        rospy.loginfo(angle_final)

        follower.vel.angular.z = -0.7
        if follower.line_detected:
```

```
            follower.vel.angular.z = 0.0
            follower.reori = 0
            follower.search = False

        elif (angle_start >=0.0 and angle_final<=0.0 and ori_euler[2] < angle_final):
            follower.vel.angular.z = 0.0
            if follower.line_detected:
                follower.reori = 0
                follower.search = False
            else:
                ori_list = [follower.orientation.x, follower.orientation.y, follower.orientation.z,
                        follower.orientation.w]
                follower.aux_orientation = follower.quat_to_euler(ori_list)
                follower.reori = 3

        elif (angle_start <=0.0 and angle_final >=0.0):
            if (ori_euler[2] >= 0.0):
                angle_now = ori_euler[2] - numpy.pi
            else:
                angle_now = ori_euler[2] + numpy.pi
            if angle_now < (angle_final - numpy.pi):
                follower.vel.angular.z = 0.0
                if follower.line_detected:
                    follower.reori = 0
                    follower.search = False
                else:
                    ori_list = [follower.orientation.x, follower.orientation.y, follower.orientation.z,
                            follower.orientation.w]
                    follower.aux_orientation = follower.quat_to_euler(ori_list)
                    follower.reori = 3
        else:
            follower.vel.angular.z = -0.7

    elif follower.reori == 3:
        # rospy.loginfo("Reorientacion 3")
        ori_list = [follower.orientation.x, follower.orientation.y, follower.orientation.z,
                follower.orientation.w]
        ori_euler = follower.quat_to_euler(ori_list)

        angle_start = follower.aux_orientation[2]
        angle_now = 0.0
        angle_final = normalization(follower.aux_orientation[2] - ((89 * numpy.pi) / 180))

        follower.vel.angular.z = -0.7
        if follower.line_detected:
            follower.vel.angular.z = 0.0
            follower.reori = 0
            follower.search = False

        elif (ori_euler[2] < angle_final) and ((angle_start >= 0.0 and angle_final >= 0.0) or (angle_start >=
0.0 and angle_final <= 0.0 ) or (angle_start <= 0.0 and angle_final <= 0.0 )) :
            follower.vel.angular.z = 0.0
            follower.reori = 0
            follower.search = False
        elif (angle_start <= 0.0 and angle_final >= 0.0):
            if (ori_euler[2] >= 0.0):
                angle_now = ori_euler[2] - numpy.pi
            else:
                angle_now = ori_euler[2] + numpy.pi
            if angle_now < (angle_final - numpy.pi):
```

```
            follower.vel.angular.z = 0.0
            follower.reori = 0
            follower.search = False
        else:
            follower.vel.angular.z = -0.7

    cmd_vel_pub.publish(follower.vel)

def waiting():
    # rospy.loginfo("No navego")
    follower.vel.linear.x = 0.0
    # follower.vel.linear.x = 1.0
    follower.vel.linear.y = 0.0
    follower.vel.linear.z = 0.0
    follower.vel.angular.x = 0.0
    follower.vel.angular.y = 0.0
    follower.vel.angular.z = 0.0
    cmd_vel_pub.publish(follower.vel)

def normalization(angle):
    if angle > numpy.pi:
        angle = angle - (2 * numpy.pi)
    elif angle <= -numpy.pi:
        angle = angle + (2 * numpy.pi)
    return angle

def listener():
    rospy.Subscriber('/campero/line_detected',Bool, Goal_callback)
    rospy.Subscriber('/campero/goal_id', Int16, follower.id_callback)
    rospy.Subscriber('/campero/new_frame', Bool, follower.nav_callback)
    rospy.Subscriber('/campero/end_frame', Bool, follower.end_nav_callback)
    rospy.Subscriber('/campero/amcl_pose', PoseWithCovarianceStamped,
follower.estimated_pose_callback)
    rospy.Subscriber('/campero/comienzo', Bool, follower.Has_started_callback)
    rospy.Subscriber('/campero/front_laser/scan', LaserScan, follower.obst_dist_callback)
    rospy.Subscriber('/campero/robotnik_base_control/odom', Odometry, follower.actual_pose_callback)
    rospy.Subscriber('/campero/in_line', Int8, follower.robot_in_line)

if __name__ == '__main__':
    try:
        rospy.init_node('line_follower')
        # listn = tf.TransformListener()
        cmd_vel_pub = rospy.Publisher('/campero/cmd_vel', Twist, queue_size=1)
        searching_line_pub = rospy.Publisher('/campero/searching_line', Bool, queue_size=10)
        goal_arr_pub = rospy.Publisher('/campero/goal_complete', Bool, queue_size=10)
        final_complete_pub = rospy.Publisher('/campero/final_goal_complete', Bool, queue_size=10)
        positioned_in_line_pub = rospy.Publisher('/campero/positioned', Bool, queue_size=10)
        oriented_in_line_pub = rospy.Publisher('/campero/oriented', Bool, queue_size=10)
        follower = Follower()
        listener()
        i = 0
        ##################################################
        with open('/home/oscar/Documentos/Documentos_TFG/Datos/Posicion_Campero', 'w') as f:
            f.write('Posición y orientación del Campero\n')
            f.write('      X                Y                EULER      \n')
        with open('/home/oscar/Documentos/Documentos_TFG/Datos/Velocidad_Campero', 'w') as f:
            f.write('Velocidad en navegacion \n')
            f.write('      v_X       v_Y        w      \n')
        ##################################################
        while not follower.fin and not rospy.is_shutdown():
```

```
        if follower.Start:
            Start_program()
            if follower.in_line == 1 and i < 1:
                goal_arr_pub.publish(True)
                i += 1
            # rospy.loginfo("Comienzo")
        else:
            ####################################################
            ori_list = [follower.orientation.x, follower.orientation.y, follower.orientation.z,
                        follower.orientation.w]
            euler = follower.quat_to_euler(ori_list)
            with open('/home/oscar/Documentos/Documentos_TFG/Datos/Posicion_Campero', 'a') as f:
                f.write('       ' + str(follower.position.x) + '       ' + str(follower.position.y) + '       ' + str(
                    euler[2]) + '       \n')
            ####################################################
            if not follower.fin:
                if follower.obstacle and not follower.search:
                    # rospy.loginfo("Obstaculo detectado")
                    obstacle_avoidance()
                elif follower.end_nav:
                    # rospy.loginfo("Navegando al punto final")
                    follower.vel.angular.z = 0.0
                    cmd_vel_pub.publish(follower.vel)
                    follower.nav = False
                    end_nav()

                elif follower.search:
                    rospy.loginfo("Buscando la linea")
                    search_line()
                elif not follower.obstacle and not follower.end_nav and not follower.search:
                    navigation()
                    goal_arr_pub.publish(True)
                    searching_line_pub.publish(False)
                    # waiting()
                    # rospy.sleep(0.25)

                # rospy.loginfo(follower.id)

    except rospy.ROSInterruptException:
        rospy.loginfo("Detection test finished.")
```

El siguiente código no es necesario para realizar la navegación, su funcionalidad es únicamente obtener los umbrales de la máscara para la marca que queramos obtener. Lo anterior se realiza por medio del cambio de los valores mínimos y máximos RGB, con lo que podemos adaptar el valor del color que visualizamos para obtener la máscara deseada.

- prueba.py

```
#!/usr/bin/env python

"""line_tracker.py V. 1.0 07/07/2021
   Autor: Oscar Alejandro
"""
import rospy, cv2, cv_bridge, numpy, math
from geometry_msgs.msg import Pose
from std_msgs.msg import Bool, Int8, Float32MultiArray

# threshold_value_White = numpy.array([[195,195,195],[255,255,255]]) #White
# threshold_value = numpy.array([[0,0,0],[38,38,38]]) #Black
threshold_value_Yellow = numpy.array([[0, 130, 150], [150, 255, 255]])  # Yellow
threshold_value_Red = numpy.array([[0, 0, 150], [150, 150, 255]])  # Red
# threshold_value = numpy.array([[27,100,27],[73,255,193]]) #Green

class Tracker():
   def __init__(self):
      self.bridge = cv_bridge.CvBridge()
      self.i = 0
      #
_____
_____
      self.rl = 95
      self.gl = 40
      self.bl = 40
      self.rh = 190
      self.gh = 150
      self.bh = 120
      self.win=cv2.namedWindow("detect")
      cv2.createTrackbar("high_blue", "detect", 0, 255, self.change_bh)
      cv2.createTrackbar("high_green", "detect", 0, 255, self.change_gh)
      cv2.createTrackbar("high_red", "detect", 0, 255, self.change_rh)

      cv2.createTrackbar("low_blue", "detect", 0, 255, self.change_bl)
      cv2.createTrackbar("low_green", "detect", 0, 255, self.change_gl)
      cv2.createTrackbar("low_red", "detect", 0, 255, self.change_rl)

   def change_bh(self, val):
      self.bh = val
   def change_gh(self, val):
      self.gh = val
   def change_rh(self, val):
      self.rh = val
```

```
    def change_bl(self, val):
        self.bl = val
    def change_gl(self, val):
        self.gl = val
    def change_rl(self, val):
        self.rl = val

def image_callback(msg):
    # image = tracker.bridge.imgmsg_to_cv2(msg, desired_encoding='bgr8')
    image = cv2.imread(msg)
    h, w = image.shape[:2]
    cx, cy = 0, 0
    if h != 480 or w != 640:
        dim = (640, 480)
        image = cv2.resize(image, dim, interpolation=cv2.INTER_AREA)

    # PREPARE THE IMAGE FOR UTILIZATION
    blurImg = cv2.GaussianBlur(image, (5, 5), 0)  # Gaussian Blur Smoothing

    threshImg = cv2.inRange(blurImg, numpy.array([tracker.bl,tracker.gl,tracker.rl],numpy.uint8),
numpy.array([tracker.bh,tracker.gh,tracker.rh],numpy.uint8))
    # threshImg = cv2.inRange(blurImg, threshold_value_Red[0], threshold_value_Red[1])
    threshImg_end = cv2.inRange(blurImg, threshold_value_Yellow[0], threshold_value_Yellow[1])

    # Erode and dilate to remove accidental line detections
    mask = cv2.erode(threshImg, None, iterations=2)
    mask = cv2.dilate(mask, None, iterations=2)

    mask_end = cv2.erode(threshImg_end, None, iterations=2)
    mask_end = cv2.dilate(mask_end, None, iterations=2)

    # FIND TH CONTOURS OF THE IMAGE
    _, contours, hierarchy = cv2.findContours(mask.copy(), 1, cv2.CHAIN_APPROX_NONE)
    _, contours_end, hierarchy_end = cv2.findContours(mask_end.copy(), 1,
cv2.CHAIN_APPROX_NONE)

    h, w, d = image.shape
    # cx = None
    c = None
    area = None
    c_end = None
    area_end = None

    if len(contours) > 0:
        c = max(contours, key=cv2.contourArea)
        area = cv2.contourArea(c)
        rospy.loginfo("AREA Linea")
        rospy.loginfo(area)

    if len(contours_end) > 0 and tracker.in_line:
        c_end = max(contours_end, key=cv2.contourArea)
        area_end = cv2.contourArea(c_end)
        rospy.loginfo("AREA FINAL")
        rospy.loginfo(area_end)

    if area > 6000:
        tracker.line_detected = True
        rospy.loginfo("Linea detectada")

        # c = max(contours, key=cv2.contourArea)
```

```
    M = cv2.moments(c)
    cx = int(M['m10'] / M['m00'])
    cy = int(M['m01'] / M['m00'])

    cv2.line(image, (cx, 0), (cx, h), (255, 0, 0), 1)
    cv2.line(image, (0, cy), (w, cy), (255, 0, 0), 1)
    cv2.drawContours(image, c, -1, (0, 0, 0), 1)

    [vx, vy, x, y] = cv2.fitLine(c, cv2.DIST_L2, 0, 0.01, 0.01)

    angle = math.atan2(vy, vx)
    if angle < 0:
       angle = angle + math.pi / 2
    else:
       angle = angle - math.pi / 2

    angle *= -1
    rospy.loginfo(vx)
    rospy.loginfo(x)
    rospy.loginfo(vy)
    rospy.loginfo(y)
    cv2.line(image,
           (int(cx - math.cos(angle - math.pi / 2) * 100), int(cy + math.sin(angle - math.pi / 2) * 100)),
           (int(cx + math.cos(angle - math.pi / 2) * 100), int(cy - math.sin(angle - math.pi / 2) * 100)),
           (0, 0, 0), 1)

else:

    tracker.line_detected = False
    rospy.loginfo("Linea no detectada")

    if area_end > 6000:

       tracker.line_detected = True
       tracker.end_detected = True

       rospy.loginfo("Punto final detectado")

       M_end = cv2.moments(c_end)
       cx = int(M_end['m10'] / M_end['m00'])
       cy = int(M_end['m01'] / M_end['m00'])

       cv2.drawContours(image, c_end, -1, (0, 0, 0), 1)

       [vx, vy, x, y] = cv2.fitLine(c_end, cv2.DIST_L2, 0, 0.01, 0.01)

       angle = math.atan2(vy, vx)
       if angle < 0:
          angle = angle + math.pi / 2
       else:
          angle = angle - math.pi / 2

       angle *= -1
       cv2.line(image,
              (int(cx - math.cos(angle - math.pi / 2) * 100), int(cy + math.sin(angle - math.pi / 2) * 100)),
              (int(cx + math.cos(angle - math.pi / 2) * 100), int(cy - math.sin(angle - math.pi / 2) * 100)),
              (0, 0, 0), 1)

cv2.imshow("mask", mask)
```

```
    cv2.imshow("mask_end", mask_end)
    cv2.imshow("output", image)
    cv2.waitKey(3)


if __name__ == '__main__':
    try:
        rospy.init_node('line_tracker')
        line_detected_pub = rospy.Publisher('/campero/line_detected', Bool, queue_size=10)
        line_data_pub = rospy.Publisher('/campero/line_data', Float32MultiArray, queue_size=1)
        end_detected_pub = rospy.Publisher('/campero/end_detected', Bool, queue_size=10)
        goal_pose_pub = rospy.Publisher('/campero/goal_pose', Pose, queue_size=10)
        in_line_pub = rospy.Publisher('/campero/in_line', Int8, queue_size=10)
        response_pub = rospy.Publisher('/campero/comienzo', Bool, queue_size=10)
        tracker = Tracker()
        while not rospy.is_shutdown():
            image_callback("/home/oscar/Documentos/Documentos_TFG/Imagenes Campero
real/Capturas_campero/prueba_cinta.jpg")
            rospy.sleep(0.05)

    except rospy.ROSInterruptException:
        rospy.loginfo("Detection test finished.")
```

Anexo B. Manual de usuario

En este anexo se puede encontrar una pequeña guia para poner en marcha los programas que se han utilizado en el trabajo y el orden de ejecución de estos mismos para un correcto funcionamiento.

-Lanzamiento de la simulación, seguimiento de línea:

1º

source /opt/ros/melodic/setup.bash
cd campero_ws/
source devel/setup.bash
roslaunch campero_navigation campero_nav.launch

2º

source /opt/ros/melodic/setup.bash
cd campero_ws/
source devel/setup.bash
rosrun campero_navigation line_tracker.py

3º

source /opt/ros/melodic/setup.bash
cd campero_ws/
source devel/setup.bash
rosrun campero_navigation line_marker.py

4º

source /opt/ros/melodic/setup.bash
cd campero_ws/
source devel/setup.bash
rosrun campero_navigation line_follower.py

5º Este último es solo para conocer los umbrales de la línea, no es necesario su ejecución para un correcto funcionamiento.

source /opt/ros/melodic/setup.bash
cd campero_ws/
source devel/setup.bash
rosrun campero_navigation prueba.py

Para hacer ejecutables los scripts:

chmod +x /home/'username'/path/file_name.py