Universidad Zaragoza
1542

Trabajo Fin de Grado

Navegación autónoma de robot manipulador móvil con cámara y láser en el entorno ROS

Autonomous navigation of mobile manipulator robot with camera and laser in the ROS environment

Autor

David Barrera Gracia

Directores

Rosario Aragüés Muñoz

Gonzalo López Nicolás

ESCUELA DE INGENIERÍA Y ARQUITECTURA
2020

AGRADECIMIENTOS

A mis directores, Rosario y Gonzalo:
Por haberme ofrecido la oportunidad de realizar este proyecto.

A Rafael Herguedas:
Por hacer posible el uso del robot Campero en los experimentos.

Navegación autónoma de robot manipulador móvil con cámara y láser en el entorno ROS

RESUMEN

La robótica es un campo cada vez más multidisciplinar y cada vez se requiere de mayor especialización de los equipos de trabajo. Esto ha llevado a lo que hoy en día se conoce como desarrollo de software robótico colaborativo. Dentro de este contexto nos encontramos con el entorno ROS (Robot Operating System). Cuenta con una comunidad internacional muy activa y tiene una gran proyección de futuro.

Es por ello que en este trabajo hemos usado el entorno ROS y sus herramientas. El objetivo era simular e implantar la navegación autónoma de un robot manipulador móvil, haciendo uso tanto de sensores láser, como de visión. Además de desarrollar código para hacer posible la navegación en el entorno ROS, hemos creado entornos de simulación 3D para Gazebo, modelos 3D de marcas visuales, hemos realizado el mapeado de entornos reales y hemos simulado e implantado distintos tipos de navegación en un robot real. Por lo que hemos cumplido con nuestros objetivos.

Para ello hemos empleado un robot manipulador móvil conocido como robot Campero. Este robot es un prototipo del robot comercial RB-EKEN, creado por la empresa Robotnik. Este proyecto ha sido pionero en la implantación de un sistema de navegación autónoma en el robot Campero disponible en la Universidad de Zaragoza. Además, también hemos sido pioneros en utilizar las cámaras de este robot. Consiguiendo un sistema de navegación basado en la detección de marcas visuales.

Por otra parte, este trabajo sigue la filosofía de ROS, y parte del tiempo invertido en este proyecto puede ser aprovechado para futuros proyectos. De manera que esta memoria cuenta con tutoriales detallados para que cualquier persona sea capaz de replicar los experimentos que hemos realizado. Se incluye además información de utilidad sobre el robot Campero, con el objetivo de hacer más sencillas las primeras tomas de contacto con el hardware.

Índice

Capítulo 1

Introducción

1.1. Motivación

La robótica es una rama de conocimiento que cobra cada vez más importancia. Cada día son más procesos los que necesitan ser robotizados, ya sea por su peligrosidad para el ser humano y por su repetitividad. Cada vez se intentar robotizar procesos o actividades más complejas por lo que es necesario que un robot sea capaz incluso de tomar decisiones por si mismo cuando surge algún imprevisto. De ahí surge el interés por los robots autónomos, robots que mediante sus sensores son capaces de percibir su entorno y, conforme a la información que poseen, son capaces de tomar la decisión que consideran óptima.
Dentro de los robots autónomos podemos encontrar tanto robot manipuladores como móviles. Los primeros, fueron creados con la intención de ayudar al ser humano en operaciones que requerían gran esfuerzo, por lo que estaban presentes mayormente en entornos industriales. Debido a su fuerza normalmente trabajaban en entornos distintos al de los demás operarios para evitar accidentes. Conforme la robótica ha ido avanzando y con la llegada de los robots autónomos, podemos encontrar robots manipuladores que incorporan varios mecanismos de seguridad, y que comparten su espacio de trabajo con los humanos. Llegando así a lo que conocemos hoy en día como robótica colaborativa.
Los segundos son un grupo muy variado, ya que son muchos sus campos de aplicación. Podemos encontrar robots terrestres, marinos o aéreos y pueden ser usados para trabajar en entornos remotos o peligrosos, para realizar tareas de vigilancia o rescate, para mantenimiento y para muchos otros campos.

En el estudio de este trabajo se va a utilizar un manipulador móvil autónomo y colaborativo conocido como robot Campero, que es la fase previa al modelo comercial del "Manipulador móvil RB-EKEN"[1]. Este robot está dotado con distintos

sensores que le harán capaz de situarse en el entorno y mediante las directrices que nosotros le indiquemos actuará en consecuencia. Aunque tenga características de robot manipulador, en este trabajo lo emplearemos solamente como robot móvil.

Como ya hemos comentado los robots móviles pueden cumplir distintas tareas, pero la navegación va a ser una parte común a todas ellas, ya que han de ser capaces de desplazarse por un entorno por ellos mismos. Es por esto que este trabajo se va a centrar en cómo un robot, mediante sus distintos sensores, es capaz de desenvolverse por el entorno y alcanzar las posiciones necesarias para posteriormente desempeñar su función.

Para el desarrollo y estudio de la navegación del robot vamos a usar el entorno ROS [2]. Dado que es un entorno colaborativo y de código abierto, implementaremos algún paquete desarrollado por la comunidad. Y, por supuesto, desarrollaremos nuestros propios programas y entornos de simulación.

1.2. Contexto

Este trabajo forma parte de las actividades del proyecto COMMANDIA. COMMANDIA es un proyecto cofinanciado por el Programa Interreg Sudoe y por el Fondo Europeo de Desarrollo Regional (FEDER). El nombre del proyecto COMMANDIA significa Robótica móvil colaborativa de objetos deformables en aplicaciones industriales [3]. Este trabajo toma el relevo a otro trabajo de fin de grado que formó parte de este proyecto: 'Navegación de robots manipuladores en el entorno de ROS' [4]

Dado que el robot Campero es un prototipo no comercial, vamos a hablar de las especificaciones de su modelo comercial, el robot RB-EKEN 10 de la empresa Robotnik.

Figura 1.1: Foto del robot Campero

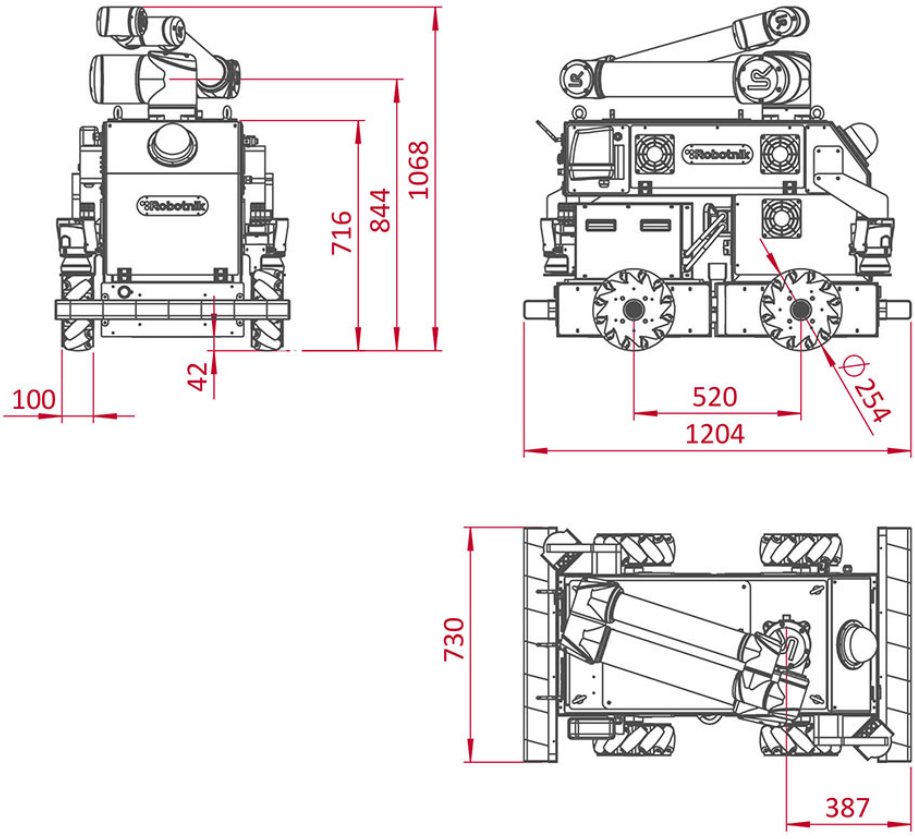

Figura 1.2: Dimensiones RB-EKEN 10 [1]

El robot cuenta con un peso de 270 kg. Cuenta dos dos tipos de ruedas, unas de goma recomendadas para exterior y otras mecanum recomendadas para interior. Sus dimensiones con las ruedas mecanum son de aproximadamente: 1 metro de alto, 73 centímetros de ancho y 1'2 metros de largo. Y puede alcanzar una velocidad de 2 metros por segundo. La base es una plataforma omnidireccional con 4 ruedas motorizadas que puede transportar cargas de hasta 300kg. Cuenta con dos sensores láser SICK, otros dos Intel Real Sense y una cámara PTZ.

El Campero está capacitado para realizar una navegación mediante SLAM (Simultaneous Localization And Mapping). Con los láseres SICK el robot construye un mapa del entorno. Esta navegación fue simulada en el proyecto "Navegación de robots manipuladores en el entorno de ROS"[4]. Además utiliza el método de localización AMCL apoyándose en el mapa y los láser. Nosotros utilizaremos los mapas que el robot genera para la navegación, y además, añadiremos la información que nos aportan las marcas visuales, en concreto, las marcas ArUco.

ArUco [5][6]

ArUco es una librería de código abierto que detecta una serie de marcas cuadradas en imágenes. Además, puede estimar la posición de la cámara respecto a las marcas, siempre y cuando la cámara esté calibrada. Hay varios tipos de marcas, cada una de ellas pertenece a un diccionario. El diccionario que nosotros usaremos es el original ArUco, aunque la librería es capaz de detectar marcas de otros diccionarios (ArToolKit+, Chilitags y AprilTags). La librería ha sido creada por el grupo de investigación 'Aplicaciones de la Visión Artificial' de la Universidad de Córdoba [7].

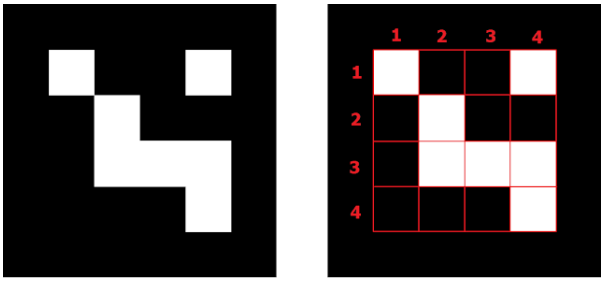

Figura 1.3: Marca ArUco

Las marcas están formadas por un borde negro y una región interior que representa un patrón binario codificado (el patrón de la figura 1.3 se representa en la tabla 1.1).

Cada patrón binario es único y sirve para identificar cada marca. Dependiendo del diccionario las marcas tienen más o menos bits (en nuestro caso utilizaremos marcas 5x5). Cuantos más bits, más marcas habrá en el diccionario, pero se necesitará una

1	0	0	1
0	1	0	0
0	1	1	1
0	0	0	1

Tabla 1.1: Patrón binario

mayor resolución para una detección correcta. Nosotros no necesitaremos un diccionario grande, así que nos limitaremos a usar el original de ArUco.

Para la estimación de la posición de las marcas utilizaremos marcas individuales, aunque también es posible utilizar mapas de marcas (figura 1.4). Para la estimación de la posición se utilizan sólo cuatro puntos coplanares y esto puede generar lo que se conoce como problema de ambigüedad. Es decir, en caso de que queramos estimar la posición de la cámara, la cámara tiene dos posibles localizaciones. O en caso de que estemos estimando la posición de la marca, la marca tiene dos posibles orientaciones. Este problema solo debería ser preocupante en caso de que la marca sea pequeña, ya sea por su tamaño o por la distancia de la cámara a la marca. En nuestro caso, hemos observado que alguna vez el eje Z de la marca se ha invertido, pero filtrando las medidas podemos corregir ese error de manera sencilla. Eso sí, siempre y cuando la cámara tenga una imagen de la marca lo suficientemente grande.

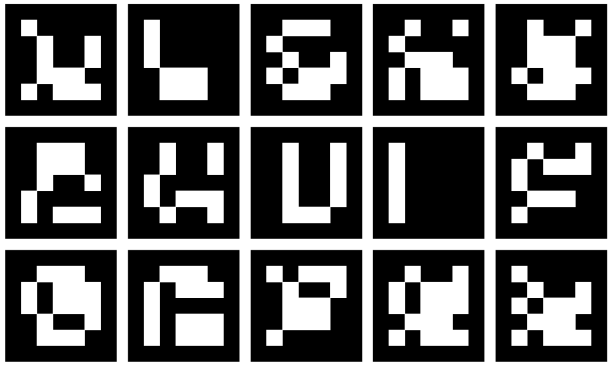

Figura 1.4: Mapa ArUco

1.3. Objetivos

Para alcanzar nuestra meta vamos a tener que ir cumpliendo distintos objetivos hasta llegar al último, que será realizar la navegación con el robot Campero.

1. Utilizaremos la distribución de Linux 'ubuntu 16.04 LTS', así que el primer paso será instalarla y descargar los paquetes de ROS.

2. Antes de trabajar con el Robot Campero en ROS, debemos aprender a usar el

entorno y sus distintas herramientas, siguiendo la documentación oficial.

3. Hecha ya la primera toma de contacto y con un conocimiento básico del entorno, ya podremos descargar los paquetes del Campero. Además, habrá que reproducir los programas que se han usado hasta la fecha en otros proyectos. [4]

4. Llegados a este punto ya seremos capaces de desarrollar nuestros propios programas para la navegación. Estudiaremos distintos tipos de navegación:

   a) Navegación mediante mapa y láser

      – Aprenderemos a programar las acciones en ROS
      – Simularemos el movimiento del robot indicándole el punto objetivo del mapa

   b) Navegación mediante cámara

      – Aprenderemos a utilizar las marcas ArUco
      – Implementaremos las marcas ArUco en el entorno ROS
      – Simularemos la navegación del robot usando la marca

   c) Navegación mediante mapa, láser y cámara

      – A partir de la marca, obtendremos el punto de destino del robot
      – Simularemos la navegación del robot usando el mapa y la marca

5. A continuación, deberemos validar experimentalmente los métodos de navegación desarrollados y simulados, usando para ello el robot real.

6. Cumplido el objetivo principal, solo quedará sacar las conclusiones oportunas y desarrollar la memoria del proyecto.

1.4. Organización del documento

Este documento se organizará de la siguiente manera:

– Capítulo 1: El capítulo en el que nos encontramos. Donde introducimos y ponemos en contexto este proyecto.

– Capítulo 2: Servirá como introducción para quién desconozca el entorno ROS. También hablaremos sobre las herramientas que hemos usado para trabajar en el entorno.

- Capítulo 3: Hablaremos sobre el Robot campero en entorno ROS. Resaltando los principales archivos y elementos que debemos conocer del robot para trabajar en ROS. Así como una breve explicación de cada uno.
- Capítulo 4: Expondremos varios métodos de navegación y analizaremos cada uno por separado. Para ver en qué situaciones nos conviene usar cada uno de ellos.
- Capítulo 5: Este capítulo recoge varios experimentos que hemos realizado en simulación, así como los resultados.
- Capítulo 6: Llegamos al hardware. Aquí se explicarán las características del robot Campero, como ponerlo en marcha y los distintos experimentos que hemos realizado.
- Capítulo 7: Desarrollamos las conclusiones y hablamos sobre las posibilidades de los proyectos que sucedan a este.
- Bibliografía.
- Anexo A: Aquí se recogen los principales programas que hemos elaborado, así como una breve explicación de cada uno.
- Anexo B: Cuenta con tutoriales paso a paso sobre como crear marcas visuales tridimensionales con Blender y como exportarlas a Gazebo. Además, hablamos de los entornos 3D que hemos creado en Gazebo.
- Anexo C: Explicamos de manera detallada como ejecutar en ROS los distintos tipos de navegación que explicamos en el capítulo 4.

Capítulo 2

Entorno de trabajo ROS

En este capítulo se introducen los fundamentos de ROS, así como su terminología y los conceptos básicos que debemos entender antes de usar este entorno. Para ahondar más, recomendamos la lectura de 'ROS Robot Programming' [8]. Además, hablaremos de las herramientas que hemos utilizado, tanto para la simulación, como para la implementación en el robot real.

2.1. Iniciación a ROS

ROS (Robot Operating System) se define como un "meta-sistema operativo", es decir, no es como un sistema operativo convencional como Windows o Linux, sino que corre sobre un sistema operativo. Al instalar ROS dotamos al sistema operativo de funciones y librerías esenciales para la programación de aplicaciones robóticas. La idea principal de ROS es hacer un desarrollo colaborativo, de manera que si un equipo de trabajo está especializado en un campo concreto se puedan centrar en él, y posteriormente lo compartan para que otros equipos puedan aprovecharlo. Y que a su vez, también puedan aprovechar el trabajo de otros equipos especializados en campos distintos al suyo.

2.1.1. Conceptos principales

Para empezar a trabajar con ROS primero hay que asimilar una serie de conceptos y terminología. En esta sección explicaremos los conceptos más básicos.

Nodos

Los nodos son la unidad mínima de procesamiento en ROS, algo así como un programa ejecutable. La idea es que cada nodo sirva para un propósito concreto pero que esté diseñado de manera modular, para que se pueda reutilizar. Los nodos se compilan individualmente unos de otros y los ejecuta y gestiona un nodo principal

(ROS Master). Pero los nodos no pueden comunicarse entre ellos sino que tienen que hacerlo a través de un tópico.

Tópicos

Los tópicos hacen posible la comunicación entre nodos. Un elemento característico de cada tópico es el tipo de mensaje que se puede publicar en él. Cualquier nodo puede publicar en un tópico siempre y cuando publique el tipo de mensaje característico del tópico. Además, otros nodos pueden suscribirse para leer los mensaje que se publican en él. De manera que si un nodo quiere enviarle un mensaje a otro deberá hacerlo a través de un tópico. El nodo que quiere enviar el mensaje lo publica en el tópico y el nodo que lo quiere recibir se suscribe al tópico. La comunicación en los tópicos es siempre unidireccional y asíncrona.

Master

El *master* permite la conexión entre nodos y la comunicación mediantes mensajes. Registra el nombre de cada nodo y puede recibir información de ellos cuando es necesario. Sin el master no es posible ningún tipo de comunicación.

Aparte de los tres conceptos principales que acabamos de explicar, queremos resaltar los dos medios de comunicación que existen aparte de los tópicos. Porque como ya hemos dicho los tópicos solo permiten comunicación unidireccional. Entonces, si queremos que exista una realimentación debemos recurrir a otros medios:

Servicio

El servicio es una comunicación bidireccional síncrona entre el cliente que solicita el servicio, y el servidor que responde la solicitud. El cliente solicita el servicio y el servidor cuando lo ha completado responde al cliente. El servicio no mantiene la conexión, es decir, mientras el servidor está realizando el servicio no se comunica con el cliente. Cuando el servicio se ha realizado, el servidor comunica al cliente que el servicio ha sido realizado y los dos nodos finalizan la conexión.

Acción

La acción es parecida al servicio, también hay un cliente que solicita la acción a un servidor, y el servidor cuando completa dicha acción, se lo transmite al cliente. Pero el servidor, además de avisar cuando ha acabado la acción, puede dar feedback al cliente. Esto es muy útil cuando mandamos al robot que vaya a una posición determinada, ya

que no sólo sabremos si ha llegado o no, si no que sabremos el camino que ha seguido y, si no llega, podemos saber hasta donde ha llegado. Este feedback transmite de manera asíncrona un mensaje bidireccional entre el cliente y el servidor. De esta manera, el cliente además de recibir información acerca del progreso de la acción, puede mandar una orden de cancelación al servidor para acabar con la acción que se está ejecutando.

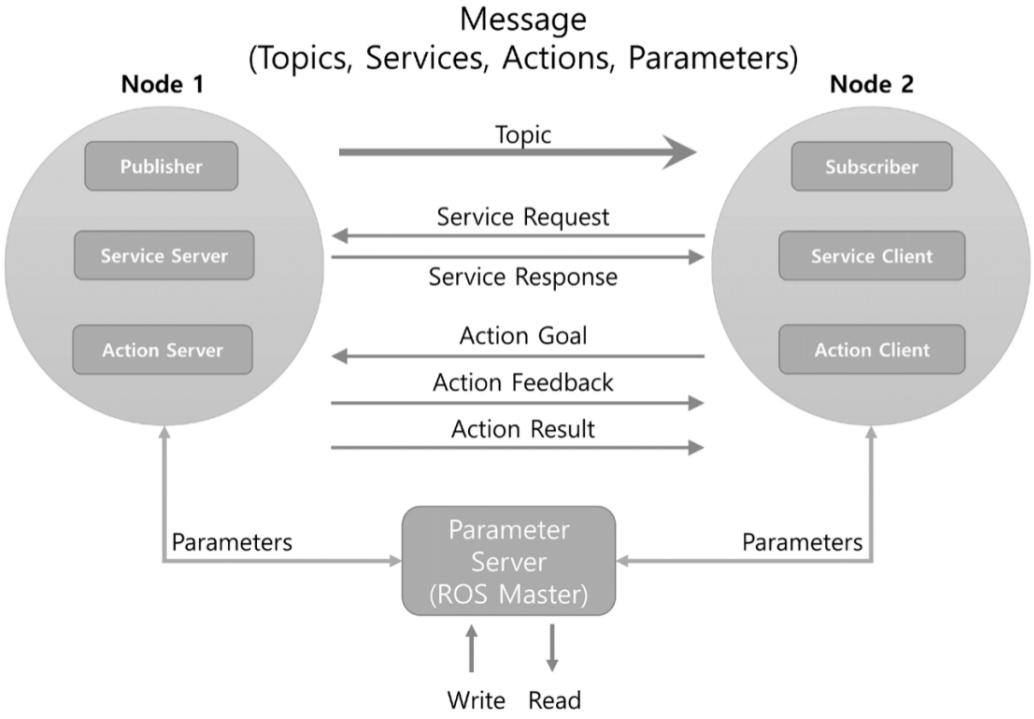

Figura 2.1: Comunicación en ROS [8]

2.1.2. Mensajes

Un mensaje es un paquete de datos que van desde los tipos básicos como *integer*, *string*, *floating point*, *boolean*, etc. Hasta mensajes que contienen otros mensajes, por ejemplo 'geometry_msgs/Pose' que contiene a 'geometry_msgs/Point Message' con tres mensajes 'float64' (uno para cada coordenada x, y, z) y a 'geometry_msgs/Quaternion Message' con cuatros mensajes 'float64' (una para cada valor del cuaternión). Como hemos dicho antes, en cada tópico solo se puede publicar un tipo de mensaje. Si no sabemos qué tipo de mensaje se publica en un tópico, podemos averiguarlo abriendo la línea de comandos e introduciendo:

$ rostopic info /nombre_del_topic

```
david@david-System-Product-Name:~/campero_ws$ rostopic info /campero/move_base_simple/goal
Type: geometry_msgs/PoseStamped

Publishers:
 * /rviz (http://192.168.1.37:37905/)

Subscribers:
 * /campero/move_base (http://192.168.1.37:44383/)
```

Figura 2.2: Comando: 'rostopic info'

En la figura 2.2 vemos un ejemplo, el comando nos devuelve el tipo de mensaje, así como los *publishers* y los *subscribers*. Aunque sepamos el nombre del mensaje, aún no sabemos que mensajes lo componen, para ello existe el comando:

$ rosmsg show nombre_del_dato

```
david@david-System-Product-Name:~/campero_ws$ rosmsg show geometry_msgs/PoseStamped
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
geometry_msgs/Pose pose
  geometry_msgs/Point position
    float64 x
    float64 y
    float64 z
  geometry_msgs/Quaternion orientation
    float64 x
    float64 y
    float64 z
    float64 w
```

Figura 2.3: Comando 'rosmsg show'

En la figura 2.3 vemos un ejemplo con 'geometry_msgs/PoseStamped'. Vemos que este mensaje contiene al mensaje 'geometry_msgs/Pose', que es el que nombrábamos al principio de esta sección. Este ejemplo nos sirve para ilustrar la modularidad de los mensajes y cómo los mensajes complejos están formados por mensajes más simples hasta llegar a los básicos.

2.2. Desarrollo del entorno para la simulación

A continuación hablaremos de las distintas herramientas de trabajo que hemos utilizado a lo largo de este proyecto en el entorno ROS.

2.2.1. Blender

Figura 2.4: Logo de Blender

Blender es un programa de creación de gráficos en 3D. Este programa no está directamente relacionado ni con la robótica, ni con ROS. Ante la necesidad de tener que utilizar marcas visuales en Gazebo, decidimos crearlas nosotros mismos. Hemos decidido usar Blender porque es la herramienta que se recomienda en los tutoriales oficiales de Gazebo para exportar archivos COLLADA. Las marcas visuales que vamos a utilizar pertenecen a la librería ArUco (ver sección 1.2).

Para crear el modelos 3D nosotros seguimos las instrucciones de este video [9]. De todas formas en el anexo B.1 explicamos nosotros mismos como crear, paso a paso, una marca visual con Blender v 2.82a.

2.2.2. Gazebo

Figura 2.5: Logo de Gazebo

Gazebo es un simulador 3D pensado para la robótica. Nos permite simular multitud de robots, sensores, entornos y cuenta con su propio motor de física. Además, es una pieza fundamental en nuestro proyecto ya que nos permitirá experimentar con la navegación antes de implementarla en un entorno real. Para la simulación utilizaremos un entorno de interior en el que situaremos diversas marcas visuales que crearemos nosotros mismos (ver anexo B.1). En el anexo B.2 explicamos paso a paso como implementar las marcas visuales en Gazebo.

2.2.3. RViz

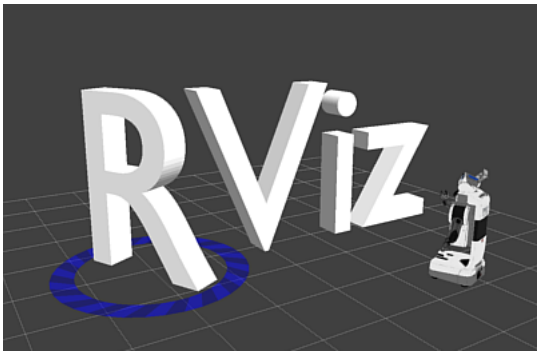

Figura 2.6: Logo de RViz

RViz es una herramienta de visualización 3D para ROS. Esta herramienta es muy útil ya que nos muestra información de los sensores de manera visual. Además también se puede usar para publicar en tópicos.

En nuestro caso nos servirá principalmente para visualizar la detección de obstáculos de los láseres, la captación de imágenes de la cámara y para pilotar el robot con el ratón en el caso de la simulación.

Capítulo 3

Robot Campero en el entorno ROS

Los archivos del robot para el entorno ROS nos han sido proporcionados por la empresa Robotnik [10]. Como hemos comentado antes, el robot estaba en una fase preliminar, por lo que no cuenta con una documentación detallada sobre los archivos que utiliza. Así que en esta sección comentaremos la información que creemos que es más relevante para trabajar en simulación con el robot Campero.

3.1. Archivos y directorios principales

Una de las principales dificultades que nos hemos encontrado en este proyecto, ha sido encontrar el archivo que debemos modificar para cambiar alguna característica, o bien del robot, o bien de la ejecución. Así como saber en qué directorios se encuentran. El directorio principal del robot Campero cuenta con 395 directorios y 1536 archivos. Por ello en esta sección vamos a hablar de los archivos que hemos necesitado modificar para la correcta ejecución de nuestros programas y de los directorios en los que se encuentran. Vamos a ir desde los más generales, los que tendremos que acceder de manera habitual, hasta los más precisos, que sólo deberemos modificar en circunstancias concretas.

3.1.1. Archivo principal de ejecución: *campero_nav.launch*

Este archivo se encuentra en:
src/campero/campero_common/campero_navigation/launch

Se encarga de lanzar Gazebo, RViz y permite seleccionar algunas características del robot. De manera que en él podemos cambiar: El mapa de Gazebo, el mapa de RViz (es importante que el mapa de RViz corresponda al de Gazebo y viceversa), el tipo de ruedas del robot (omnidireccionales o diferenciales), el valor de *gmapping* (si esta activo escaneará la habitación para hacer un mapa) o la posición inicial del robot. Hay

más parámetros que no nombramos porque no nos han hecho falta, por ejemplo, el tipo de brazo que vamos a usar.

3.1.2. Mapas y entornos

Tenemos que distinguir entre los entornos 3D que usa Gazebo (ver 3.1) y los mapas en 2D que usa RViz (ver 3.2). Empecemos con los primeros.

Entornos

Se guardan en el siguiente directorio:
src/campero/campero_sim/campero_gazebo/worlds

Por defecto, tendremos un mapa de interior y otro de exterior (más información en el anexo B.3). El mapa de interior y sus variaciones, son los que usamos en este proyecto, pero el de exterior es más amplio y nos puede ser de interés si queremos poner a prueba la navegación. Se aconseja que si creamos nuevos entornos se guarden aquí.

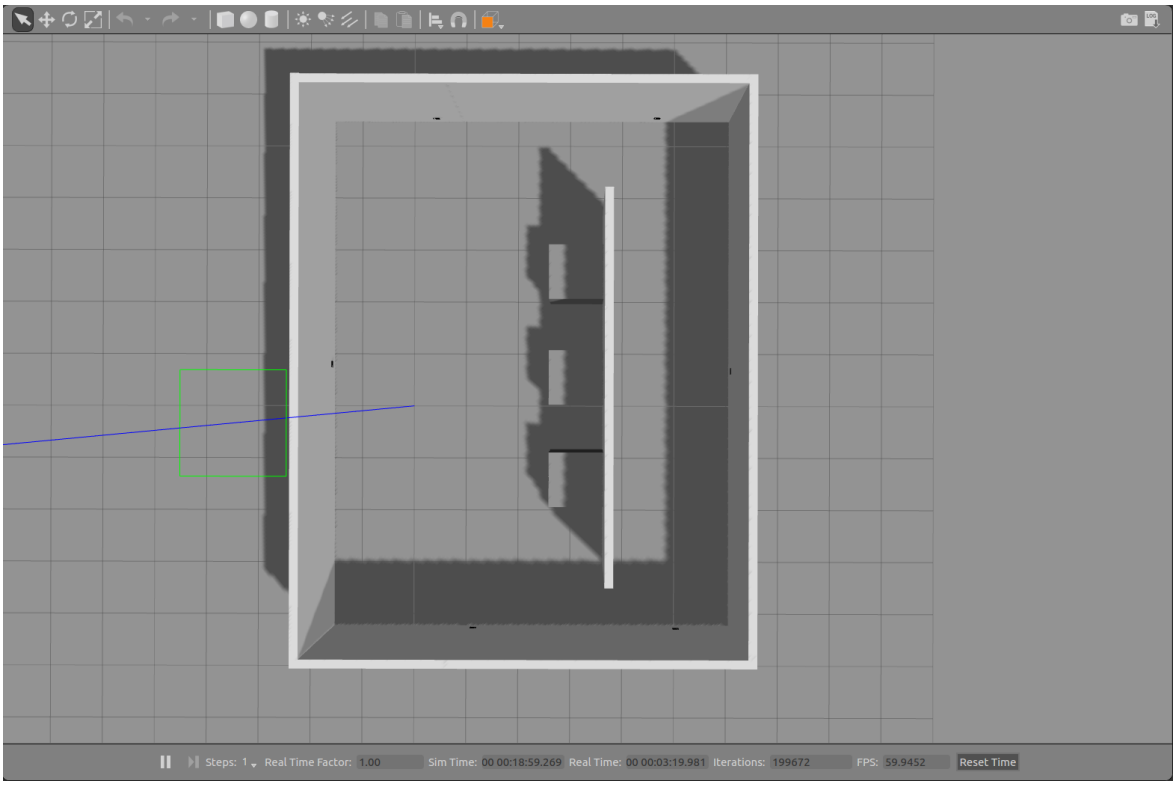

Figura 3.1: Entorno 3D de Gazebo (mapa interior con marcas ArUco)

Mapas

Se guardan en el siguiente directorio:

src/campero/campero_common/campero_localization/maps

Cada mapa se corresponde a un entorno que ha sido explorado y escaneado, por ello es importante que si creamos un nuevo entorno, lo escaneemos y guardemos aquí el archivo del mapeado (podemos encontrar como hacerlo en la sección C.1). Por defecto contaremos también con un mapa del entorno de exterior y del de interior.

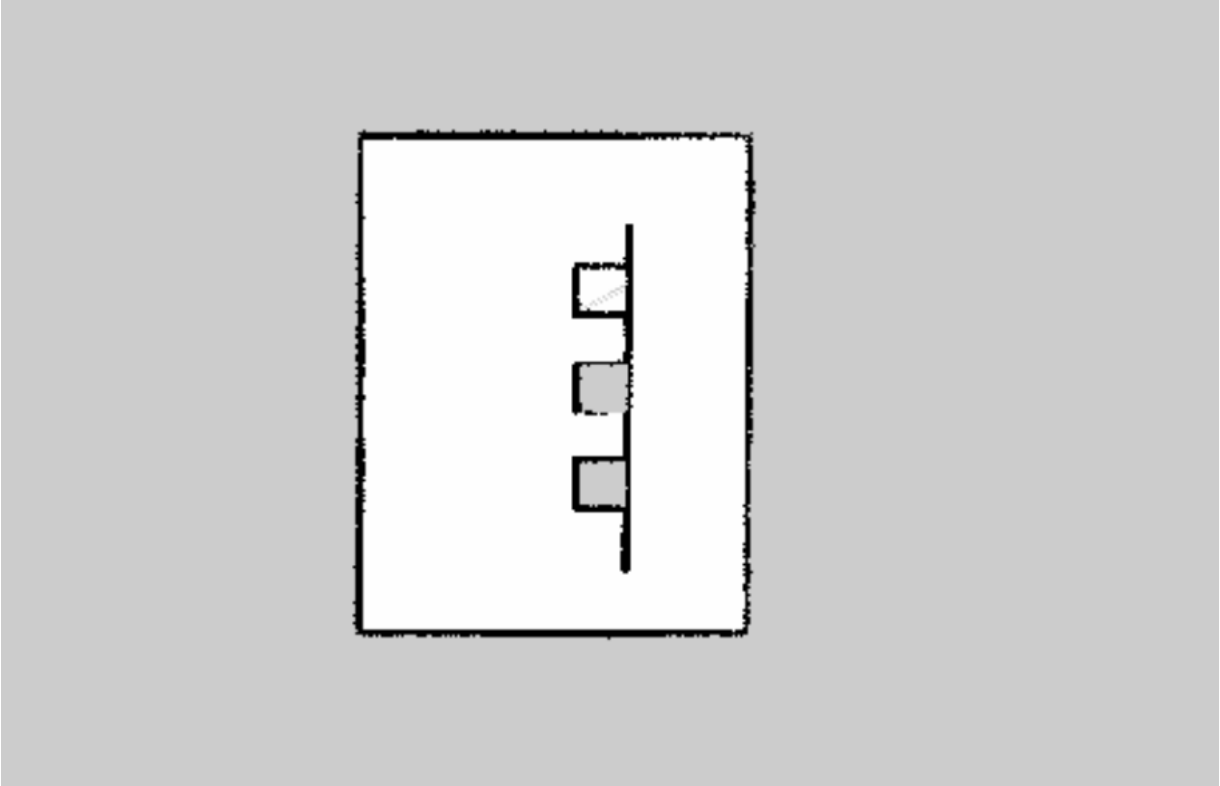

Figura 3.2: Mapa interior de RViz

3.1.3. Marcas ArUco

Para la detección de las marcas ArUco hemos usado el paquete 'aruco_ros' de PAL-Robotics [11] (para más información ir al anexo C.3). Si queremos estimar la posición de una marca usaremos el archivo 'single.launch' y si queremos estimar la de dos 'double.launch'. En nuestro caso solo hemos trabajado con una marca, así que usamos 'single.launch'. Ambos archivos se encuentran en el directorio:

src/campero/aruco_ros/aruco_ros/launch

En estos archivos podemos elegir el identificador de la marca que vamos a detectar, el tamaño de esta y cual va a ser la transformación de referencia sobre la que se hará la de la marca (ver figura C.6)).

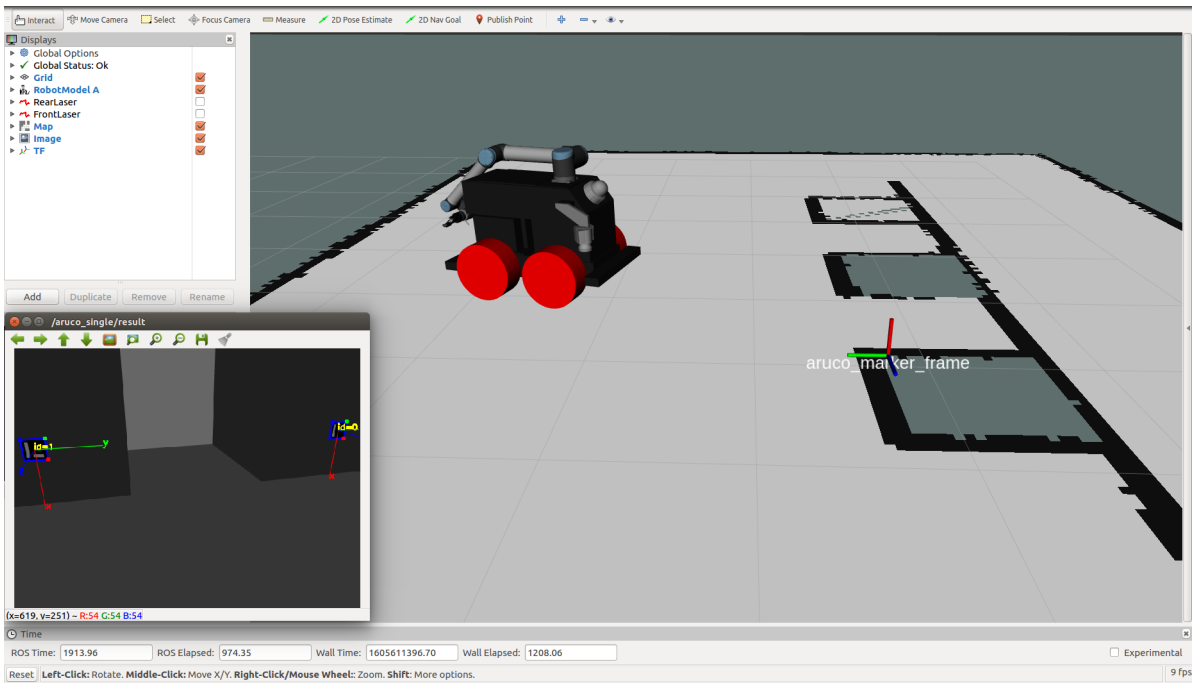

Figura 3.3: Estimación de la posición de una marca con 'single.launch'

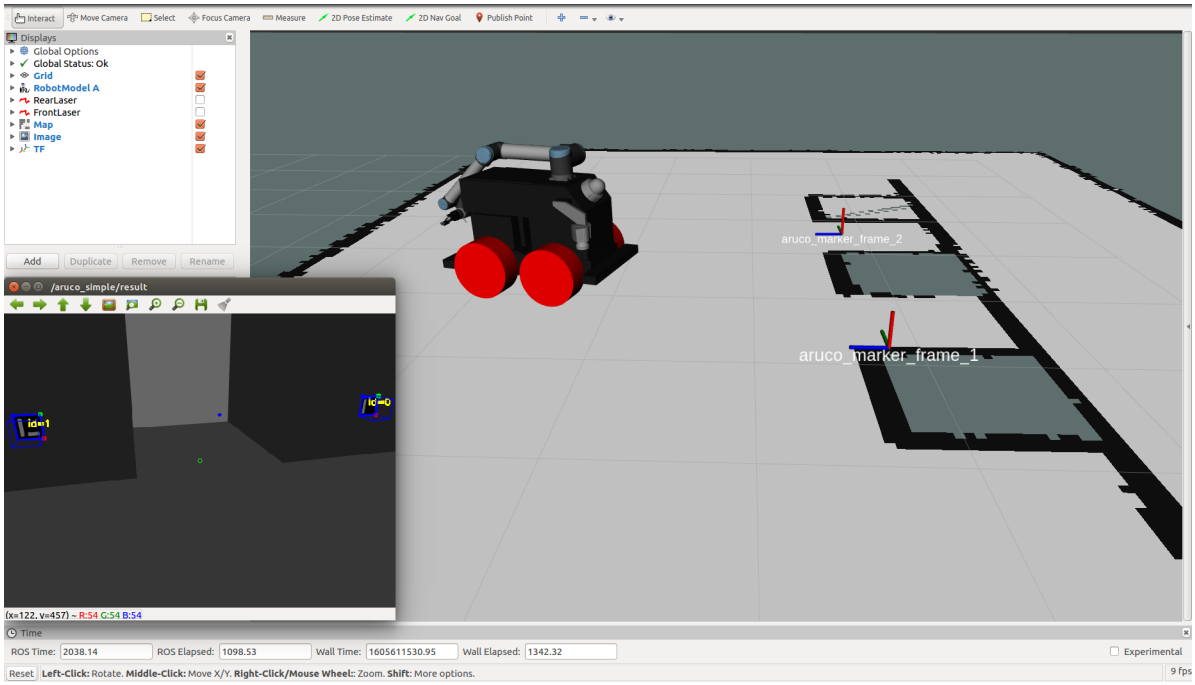

Figura 3.4: Estimación de la posición de dos marcas con 'double.launch'

En las imágenes anteriores vemos como ya sea con 'single.launch' o con 'double.launch' se detectan varias marcas. Pero lo que nos interesa no es la detección sino la estimación de la posición. Que como vemos con 'single.launch' se estima la posición de una marca, y con 'double.launch' de dos. Esta no es la única diferencia que tienen estos dos archivos, pero como no hemos trabajado con 'double.launch' no

entraremos en detalle.

Si queremos entender mejor como se realiza la detección de las marcas podemos acudir a los archivos originales de detección de marcas ArUco [5][6] , 'simple_single.cpp' o 'simple_double.cpp'. Se encuentran en:

src/campero/aruco_ros/aruco_ros/src

3.1.4. Cámara PTZ

Una cámara PTZ cuenta con tres parámetros como su nombre indica, y son: 'Pan' (ángulo en el plano horizontal), 'Tilt' (ángulo en el plano vertical) y 'Zoom'. En nuestro caso la cámara va a estar centrada en el plano horizontal (ángulo 'Pan'), pero puede ser que queramos variar su ángulo en el plano vertical ('Tilt'), ya que dependiendo de dónde situemos las marcas es posible que la cámara no llegue a verlas. Podemos modificar los ángulos en el archivo 'axis_m5525.urdf.xacro', que se encuentra en el directorio:

src/campero/robotnik_sensors/urdf

La cámara ha sido utilizada por primera vez en este proyecto. En la primera simulación observamos que la cámara estaba apuntando al techo. Cuando cambiamos el valor de la cámara a su límite inferior del ángulo tilt, vimos que no llegaba a ver el suelo. Después lo cambiamos a su límite superior y observamos que en la imagen aparecía el modelo 3D del robot (es decir, apuntaba "hacia atrás"). Obviamente estos valores límite no correspondían con los del robot real, así que procedimos a cambiarlos.

```
71  <joint name="${prefix}_tilt_joint" type="revolute">
72    <axis xyz="0 -1 0"/>
73    <origin xyz="0.0 0.0 0.0" rpy="0 -2 0"/>
74    <parent link="${prefix}_pan_link"/>
75    <child link="${prefix}_tilt_link"/>
76    <limit effort="${ptz_joint_effort_limit}" velocity="${ptz_joint_velocity_limit}" lower="-3.1416" upper="3.1416"/>
77    <joint_properties damping="${ptz_joint_damping}" friction="{ptz_joint_friction}"/>
78  </joint>
```

Figura 3.5: Archivo *axis_m5525.urdf.xacro*

Los límites se cambian en la línea 76 del archivo, y el valor en la línea 73. En la imagen 3.5 se muestra que valores hemos usado.

3.1.5. Planificador de trayectorias

En este trabajo realizaremos la navegación del robot mediante el paquete 'move_base' de ROS [12]. Nosotros usaremos el tópico adecuado para decirle al robot la posición a la que debe dirigirse y el planificador calculará la trayectoria óptima, en caso de aparecer algún obstáculo, cambiará de trayectoria. Para el Campero,

los fabricantes nos recomendaron usar el paquete 'eband_local_planner' [13]. Para modificar los valores del planificador existen dos archivos, dependiendo del tipo de rueda que estemos usando. En nuestro caso como son las omnidireccionales, emplearemos el archivo 'eband_local_planner_omni_params.yaml' (se encuentra en el directorio: *campero_ws/src/campero/campero_common/campero_navigation/config*).
Los parámetros más relevantes son:

La precisión con la que el robot llega al punto objetivo, para esto hay dos parámetros, uno para la distancia en X e Y y otro para el ángulo (líneas 6 y 7 de la imagen 3.6). Las simulaciones se han realizado para una tolerancia de 20 centímetros en los ejes X e Y y 11 º para la orientación.

```
1  base_local_planner: eband_local_planner/EBandPlannerROS
2
3  EBandPlannerROS:
4
5    # GoalTolerance
6    xy_goal_tolerance: 0.2                    # Distance tolerance for reaching goal pose (default: 0.1)
7    yaw_goal_tolerance: 0.1                   # Orientation tolerance for reaching the desired goal pose (default: 0.05)
8    rot_stopped_vel: 0.01                     # Angular velocity lower bound that determines if the robot should stop to avoid
9                                              # limit-cycles or locks (default: 0.01)
10   trans_stopped_vel: 0.01                   # Linear velocity lower bound that determines if the robot should stop to avoid
11                                             # limit-cycles or locks (default: 0.01)
12
```

Figura 3.6: Tolerancias del planificador

La velocidad linear y angular máxima que alcanza el robot (líneas 36 y 37 de la imagen 3.7). Estos parámetros tienen especial importancia en la implementación, por motivos de seguridad hay que asegurarse que los valores no son elevados.

```
35   # TRAJECTORY CONTROLLER PARAMETERS
36   max_vel_lin: 0.75                          # Maximum linear velocity (default: 0.75)
37   max_vel_th: 1.0                           # Maximum angular velocity (default: 1.0)
38   min_vel_lin: 0.0                           # Minimum linear velocity (default: 0.1)
39   min_vel_th: 0.0                            # Minimum angular velocity (default: 0.0)
40   min_in_place_vel_th: 0.0                   # Minimum in-place angular velocity (default: 0.0)
41   in_place_trans_vel: 0.0                    # Minimum in place linear velocity (default: 0.0)
42   k_prop: 3.0                                # Proportional gain of the PID controller (default: 4.0)
43   k_damp: 2.0                                # Damping gain of the PID controller (default: 3.5)
44   Ctrl_Rate: 12                              # Control rate (default: 10.0)
45   max_acceleration: 0.2                      # Maximum allowable acceleration (default: 0.5)
46   virtual_mass: 0.75                         # Virtual mass (default: 0.75)
47   max_translational_acceleration: 0.2        # Maximum linear acceleration (default: 0.5)
48   max_rotational_acceleration: 0.2           # Maximum angular acceleration (default: 1.5)
49   rotation_correction_threshold: 1.5        # Rotation correction threshold (default: 0.5)
50   differential_drive: False                  # Denotes whether to use the differential drive mode (default: True)
51   bubble_velocity_multiplier: 2.0            # Multiplier of bubble radius (default: 2.0)
52   rotation_threshold_multiplier: 1         # Multiplier of rotation threshold (default: 1.0)
53   disallow_hysteresis: False                 # Determines whether to try getting closer to the goal, in case of
54                                              # going past the tolerance (default: False)
```

Figura 3.7: Velocidades máximas del planificador

3.2. Tópicos y mensajes

Si queremos trabajar con el robot al igual que tenemos que saber qué archivos nos son de interés, debemos saber qué tópicos nos van a ser de utilidad a la hora de elaborar nuestros programas. En esta sección vamos a hablar de los tópicos más relevantes que hemos usado, o bien para publicar mensajes, o bien para leerlos, así como el tipo de mensaje que utiliza cada uno.

3.2.1. Navegación

Para enviar al robot a una posición objetivo tenemos dos opciones. Podemos publicar la posición con un mensaje 'geometry_msgs/PoseStamped' en el tópico 'campero/move_base_simple/goal', o si queremos usar una acción, publicamos un mensaje 'move_base_msgs/MoveBaseGoal' en el tópico '/campero/move_base'.
El segundo mensaje contiene al primero así que son prácticamente iguales, sólo que el segundo se usa para las acciones. En la figura 3.8 se muestra el contenido de 'move_base_msgs/MoveBaseGoal'.

```
geometry_msgs/PoseStamped target_pose
  Header header
    uint32 seq
    time stamp
    string frame_id
  geometry_msgs/Pose pose
    geometry_msgs/Point position
      float64 x
      float64 y
      float64 z
    geometry_msgs/Quaternion orientation
      float64 x
      float64 y
      float64 z
      float64 w
```

Figura 3.8: move_base_msgs/MoveBaseGoal Message [14]

3.2.2. Marcas ArUco

Cuando lanzamos 'single.launch' del paquete 'aruco_ros', podemos acudir al tópico '/aruco_single/result' para obtener información acerca de la detección. El mensaje que se publica en este tópico es 'imagen_sensor_msgs/Image', para ver este tipo de mensajes necesitamos el siguiente comando:

$ rosrun image_view image_view image:=/nombre_del_topic

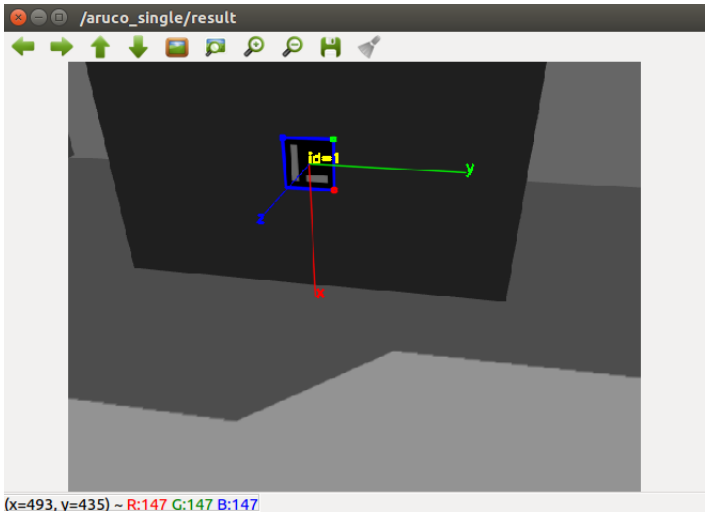

Figura 3.9: Visualización de mensaje imagen_sensor_msgs/Image

3.2.3. Velocidad del robot

Para controlar la velocidad del robot directamente, como hace por ejemplo el Teleoperador de RViz (en el cual podemos manejar las velocidades del robot en la propia interfaz de RViz), existe el tópico '/campero/cmd_vel'. En este tópico se publican mensajes del tipo 'geometry_msgs/Twist' (ver figura 3.10), que están compuestos por la velocidad lineal y por la velocidad angular en cada uno de los tres ejes.

Vector3 linear
  float64 x
  float64 y
  float64 z
Vector3 angular
  float64 x
  float64 y
  float64 z

Figura 3.10: geometry_msgs/Twist Message [15]

3.3. Transformaciones del Campero

Para localizar tanto el robot en sí, como sus distintas partes, tenemos que ver cuáles van a ser las transformaciones más útiles. En la siguiente imagen vemos todas las transformaciones que aparecen en la simulación.

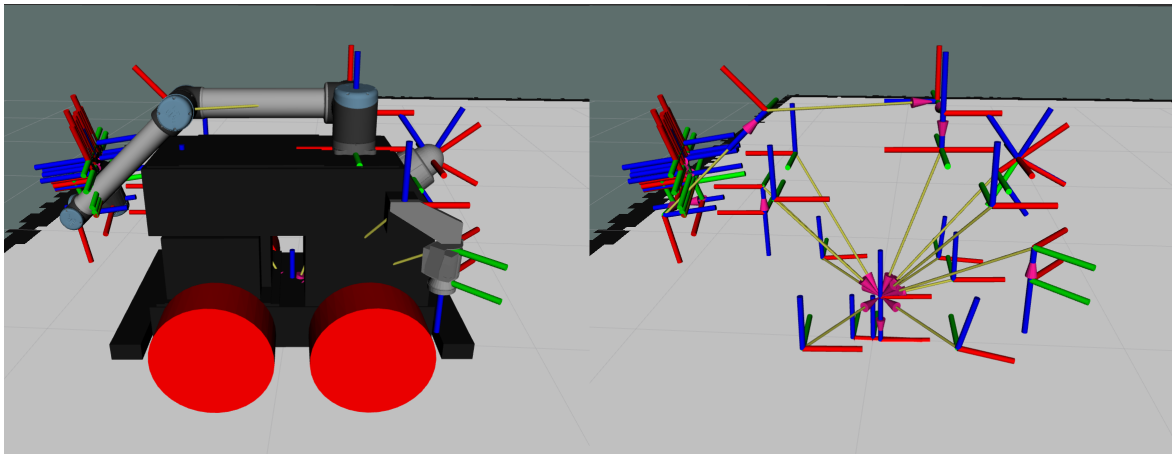

Figura 3.11: Transformaciones del Campero en simulación (izquierda con el modelo 3D del robot, derecha sin el modelo)

En la navegación vamos a necesitar principalmente tres transformaciones: Una para el mundo (transformación de referencia), una para localizar al robot y otra para la cámara.

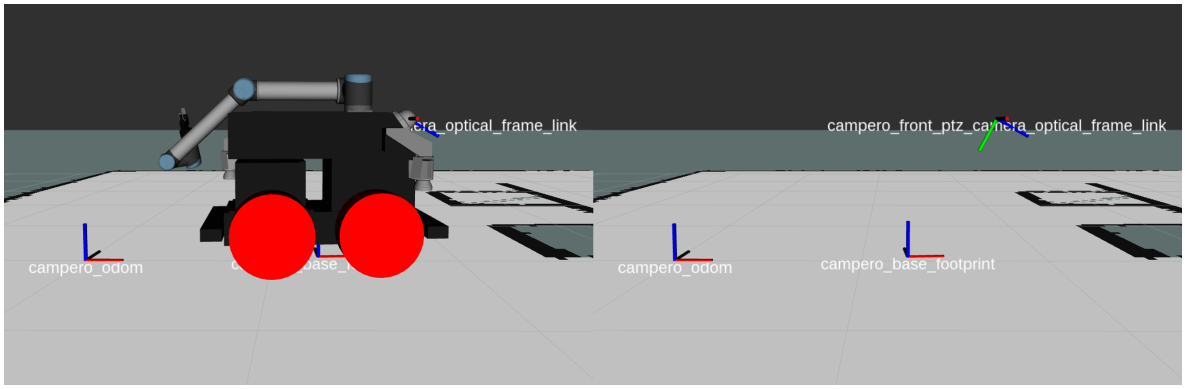

Figura 3.12: Transformaciones del Campero para la navegación en simulación

A la hora de elegir la transformación de referencia tenemos dos opciones a tener en cuenta. Una es la transformación que se crea cuando iniciamos el robot ('campero_odom'), y la otra es la que se sitúa en el origen del mapa ('campero_map'). En simulación, como hacemos que el robot se inicie en el punto de origen del mapa, las dos transformaciones son iguales. Sin embargo, con el robot real esto no es así, y la transformación que nos interesa es la que nos sitúa en el punto de origen del mapa. Por lo tanto, en simulación usaremos 'campero_odom' o 'campero_map', pero con el

robot real usaremos 'campero_map'.
Para el robot utilizaremos la transformación que se sitúa en el centro de la base 'campero_base_footprint'.
En cuanto a las cámaras utilizaremos las transformaciones cuyo eje z representa la dirección en la apunta la cámara, estas transformaciones tienen un nombre que acaba con '[...]_optical_frame'. Para la cámara PTZ 'campero_front_ptz_camera_optical_frame_link' en simulación, 'campero_front_ptz_camera_optical_frame' en el robot real. Para la cámara trasera, que no está en los paquetes de simulación, solo en el robot real, la transformación es 'campero_rear_rgbd_camera_color_optical_frame'. Estas transformaciones serán clave a la hora de localizar las marcas ArUco.

Además de las transformaciones del robot, usaremos las transformaciones de las marcas ArUco. Por defecto al detectar una marca se crea la transformación 'aruco_marker_frame'. Para mayor comodidad crearemos dos transformaciones a partir de esta: 'aruco_tf' y 'marker_tf'.
La primera de ellas está pensada para la navegación usando el mapa. Representa la posición que debe alcanzar la base del robot para situarse frente a la marca. Por lo que, cuando usemos 'aruco_tf' consideraremos que el robot está frente a la marca cuando 'campero_base_footprint' se sitúe sobre 'aruco_tf'. En la siguiente imagen mostramos la transformación que localiza al robot ('campero_base_footprint'), la transformación que localiza a la marca ('aruco_marker_frame') y la transformación que debe alcanzar el robot ('aruco_tf').

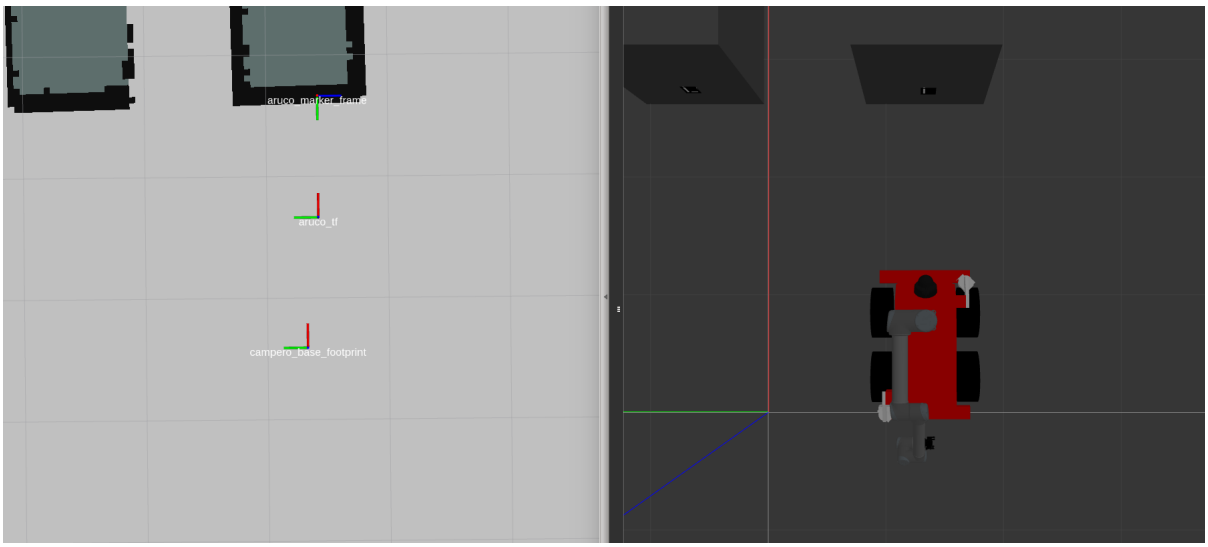

Figura 3.13: Izquierda: Transformaciones en RViz. Derecha: Simulación en Gazebo

Mientras que 'aruco_tf' es muy útil para la navegación con mapa, ya que representa el punto del mapa al que debe dirigirse el robot. Si queremos usar solo la cámara, preferimos que la transformación se sitúe sobre la marca, como ya hace 'aruco_marker_frame'. El problema de esta transformación es que no concuerda con nuestros ejes de referencia, por ejemplo, la dirección del eje X en nuestro robot (color rojo) representa el desplazamiento frontal, mientras que en la marca el eje X representa la dirección vertical. Así que esta transformación simplemente hace que las direcciones de los ejes de la marca concuerden con los de el robot (figura 3.14) .

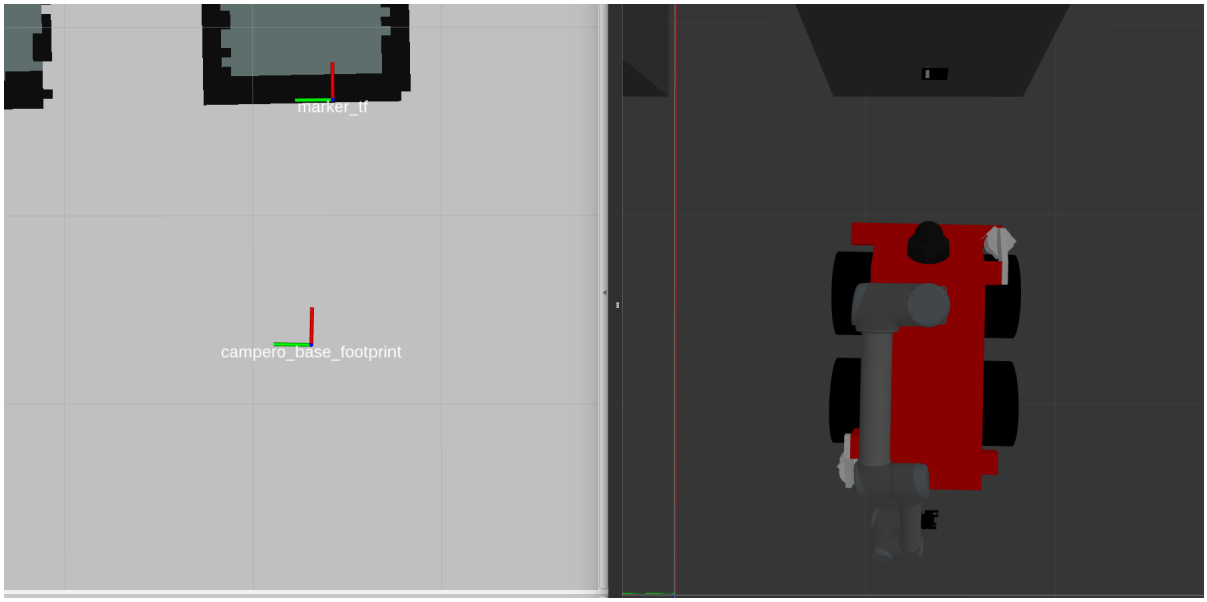

Figura 3.14: Izquierda: Transformaciones en RViz. Derecha: Simulación en Gazebo

Capítulo 4

Navegación

En este capítulo vamos a estudiar distintos tipos de navegación que podemos implementar con los medios que poseemos. El robot cuenta con dos sensores láser SICK, con una cámara PTZ y con dos cámaras Intel Real Sense . Los sensores láser permiten al robot crear un mapa de la habitación en la que se encuentra. Además, si surge un obstáculo que no se encuentra en el mapa el sensor lo detectará. Y la cámara la usaremos para indicar las posiciones a las que debe dirigirse el robot mediante marcas visuales.

Hasta ahora el robot sólo es capaz de navegar en simulación mediante la opción '2D Nav Goal' o el teleoperador de RViz. Nosotros vamos a programar tres tipos de navegación en función de las herramientas que el robot emplea para orientarse. Queremos analizar que ventajas e inconvenientes tiene usar: O los láseres, o la cámara o los dos juntos. De manera que al final tengamos un sistema de navegación que reúna todas las ventajas posibles. Así que, primero realizaremos una navegación con ayuda de los láseres en un entorno mapeado. Después, añadiremos la cámara a esta navegación. Por último, planteamos una navegación en un entorno que no haya sido escaneado, es decir, sin mapa, en la que el robot debe aproximarse a una marca visual empleando únicamente la cámara.

4.1. Navegación mediante percepción láser

En esta sección realizaremos un navegación en un entorno conocido, por lo que el robot tendrá un mapa del entorno en el que se encuentra. El mapa del robot está constituido únicamente por los puntos que puede alcanzar. Así que si le marcamos una posición que no se encuentre dentro de estos puntos el robot no se moverá. Para esta navegación, el robot tiene que escanear el entorno en el que se encuentra y crear un mapa con las posiciones que son alcanzables para él. Cuando el robot posee el mapa ya podemos indicarle a qué posición queremos que se dirija. El planificador de trayectorias

calcula la trayectoria óptima y los láseres son capaces de detectar obstáculos que no se encuentran en el mapa. En el anexo C.1 explicamos detalladamente como hemos obtenido el mapa y en el anexo C.2 como hemos realizado esta navegación en ROS.

4.2. Navegación mediante percepción láser y cámara

En esta sección añadiremos la cámara en la navegación, y como antes, el entorno es conocido. El objetivo de esta navegación es que el robot vaya al punto de destino sin que nosotros se lo tengamos que indicar como hacíamos antes. Para que el robot sepa cuál es su destino, utilizaremos marcas visuales. El punto de destino será situarse a un metro frente a la marca. Hay distintas librerías de marcas visuales pero nosotros vamos a utilizar las marcas ArUco. Para esta navegación la cámara del robot tiene que estar detectando la marca, dado que creará una referencia a partir de ella que será su posición objetivo. El software del robot se encargará de esquivar los obstáculos (que serán detectados por los láseres) que haya en el camino. En el anexo C.3 explicamos como usar esta navegación en ROS.

4.3. Navegación con cámara

En las secciones anteriores hemos trabajado en entornos que eran conocidos por el robot, pero creemos que es interesante que el robot pueda navegar sin un escaneado previo, utilizando solo las marcas y la cámara. En esta sección vamos a hacer que el robot navegue con la única ayuda de marcas visuales. Como en la navegación anterior, el robot tiene que estar detectando la marca para poder iniciar la navegación. En el anexo C.4 explicamos como hemos programado la navegación y como implementarla en ROS. Además, en la sección siguiente, compararemos los tres tipos de navegación que hemos expuesto a lo largo del capítulo.

La navegación va a constar de tres fases: En la primera el robot girará sobre su propio eje hasta tener la marca en el centro de la imagen y avanzará en línea recta (puede hacer algún pequeño giro si se descentra la marca) hasta situarse a 1,3 metros de ella (esta distancia no está escogida al azar, si no que se ha estudiado la precisión para distintas distancias, ver sección 5.3). En la segunda fase el robot hará un movimiento circular para situarse justo enfrente de la marca (hasta que la orientación de la marca y el robot coincidan) a 1,3 metros de distancia. En la última fase el robot realizará un movimiento lineal hasta alcanzar la posición final. La siguiente imagen puede ayudar

a la comprensión.

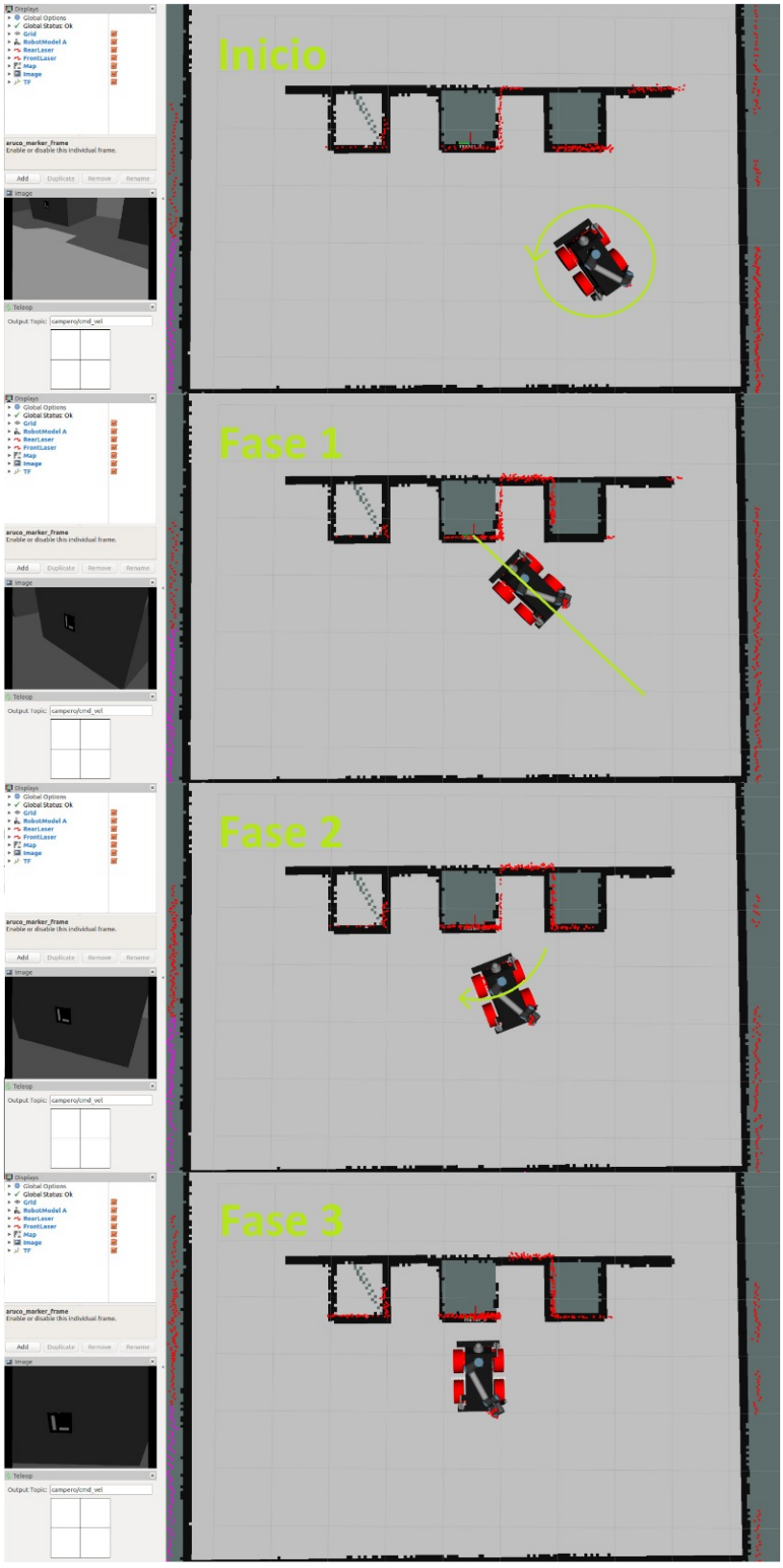

Figura 4.1: Fases de la navegación con cámara

4.4. Análisis

En este capítulo hemos visto tres tipos de navegación distinta (En el anexo C se detalla como implementarlas en ROS). Una vez ya hemos simulado todas y hemos realizado los experimentos pertinentes (ver sección 5), vamos a proceder a analizar cada tipo de navegación para ver cuando es conveniente usar cada una.

Navegación con percepción láser

- Ventajas:
  - Navegación dinámica
  - Sirve para cualquier tipo de distancia
  - Se puede lanzar desde cualquier lugar dentro del mapa
- Desventajas:
  - Entorno escaneado previamente
  - Error del planificador de trayectorias (tolerancia ajustable)

La principal ventaja es la flexibilidad: El robot tiene una posición objetivo y el sólo busca la manera de alcanzarla. En caso de encontrar un obstáculo que no ha detectado en el mapeado es capaz de reaccionar y cambiar la trayectoria. Por otro lado, cuando el robot alcanza la posición objetivo siempre hay un pequeño error (eso sí es ajustable en el planificador de trayectorias). Además el mapa tiene que haber sido escaneado previamente y cuando queramos llevarle a una posición tenemos que saber a qué coordenadas de su mapa corresponden. Aunque para entornos estáticos, trabajar con las coordenadas del mapa no debería ser un problema.

Navegación con percepción láser y cámara

- Ventajas:
  - Navegación dinámica
  - No es necesario conocer las coordenadas del mapa del robot
- Desventajas:
  - Entorno escaneado previamente
  - Error del planificador de trayectorias (tolerancia ajustable)

    - Error de la estimación de la marca (menor cuanto menor es la distancia del robot a la marca)
    - El robot tiene que estar situado de manera que vea la marca.
    - Solo válido para distancias de unos 3 metros

Esta navegación es parecida a la anterior. La diferencia es que el robot no tiene una posición determinada a la que ir. Si no que cuando lanzamos el programa, en la marca la cámara está viendo (tiene que ser la que hayamos configurado en 'single.launch' y la cámara tiene que estar viéndola al inicio del programa), se crea una posición objetivo a la que se dirige el robot. A la hora de trabajar en un entorno realista, es mucho más sencillo poner una marca en la posición a la que queremos que vaya al robot. En vez de medir esa posición con exactitud en el mapa para luego indicársela al robot. Pero al error del planificador de trayectorias hay que sumarle el error de estimación de la posición de la marca. Por lo que es la navegación menos precisa. Además, a partir de cierta distancia el robot no detecta la marca (ver experimento 5.3).

Navegación con cámara

- Ventajas:
    - El error de estimación de la marca es minúsculo
    - No hace falta tener mapa del entorno
- Desventajas:
    - No esquiva obstáculos (solo se detiene por seguridad)
    - El robot tiene que estar situado de manera que vea la marca.
    - Solo válido para distancias de unos 3 metros

Lo que hace muy interesante a esta navegación es la precisión que se llega a alcanzar con las marcas ArUco. En la navegación anterior la posición de la marca se estimaba al principio y el planificador de trayectorias dirigía al robot a esa posición. Ahora, la estimación de la posición de la marca se realiza constantemente. De manera que cuanto más se acerca el robot, más precisa es esa estimación (ver experimento 5.3). Por otro lado, aunque no haga falta conocer el entorno de trabajo, el no poder detectar obstáculos puede ocasionar que el robot no llegue a alcanzar la posición objetivo.

Conclusiones

Ha sido interesante analizar distintos tipos de navegación en función de los sensores que estemos usando. Pero como nosotros disponemos de ambos, lo ideal va a ser mezclar los tres tipos de navegación de manera que tengamos todas las ventajas. Así que, en un primer momento nos acercaremos a una posición dónde la marca sea visible con la navegación mediante láseres. Una vez en la posición objetivo, se lanzará la navegación mediante cámara y láseres para hacer una aproximación a la marca. Cuando estemos ya muy cerca de la marca, se usará la navegación mediante cámara para corregir el error del planificador de trayectorias y de la estimación inicial de la marca. De esta manera tendremos una navegación dinámica, precisa y segura. Aunque esto solamente nos servirá para entornos previamente escaneados. Si por alguna razón no tenemos el mapa del entorno, tendremos que limitarnos a una navegación con el uso exclusivo de la cámara.

Capítulo 5

Experimentos en simulación

5.1. Precisión de una posición

Uno de los problemas que nos preocupa es que exista error en la odometría y se pueda acumular, esto es algo habitual, al menos en robots reales. Vamos a comprobar si esto sucede en la simulación, para ello vamos a medir el error de posición que tiene el robot al mandarle a 10 posiciones distintas una detrás de otra.

	Error absoluto		
réplica	x (cm)	y (cm)	θ (grados)
1	13	10	4
2	1	18	5
3	13	11	3
4	0.7	17	6
5	2	19	2.5
6	1	1	7
7	1	6	6
8	2	16	5.5
9	0.8	2	5
10	1	5	6

Tabla 5.1: Precisión del robot para alcanzar posiciones

Con el número de pruebas que hemos hecho vemos que no podemos afirmar que haya error en la odometría, ya que el error no tiende a aumentar conforme aumentan las réplicas. Este resultado no nos sorprende ya que estamos en simulación, y el error de odometría se suele producir en entornos reales. Además el robot se apoya en el mapa y en los láseres para localizarse, por lo que si existiese error en la odometría se debería corregir. Este experimento también nos ha servido para corroborar que se cumple la tolarencia de error que se introduce en el planificador de rutas. Para estos experimentos la tolerancia estaba en 20 cm en posición y hasta 9 grados de orientación.

5.2. Precisión de una trayectoria

Ya sabemos que la tolerancia del error del planificador de trayectorias se cumple. Ahora queremos comprobar si el error de posición que se comete varía en función de la trayectoria que haga el robot. Para ello vamos a hacer una trayectoria en la que el robot irá en línea recta sin girar a lo largo de cinco posiciones, y una segunda trayectoria que no será en línea recta y además el robot tendrá que rotar.

En la primera trayectoria, el robot se moverá en línea recta pasando por 5 posiciones. Estas posiciones son:
(0.0,-0.5,0),(0.0,-1.0,0),(0.0,-1.5,0),(0.0,-2.0,0),(0.0,-2.5,0)

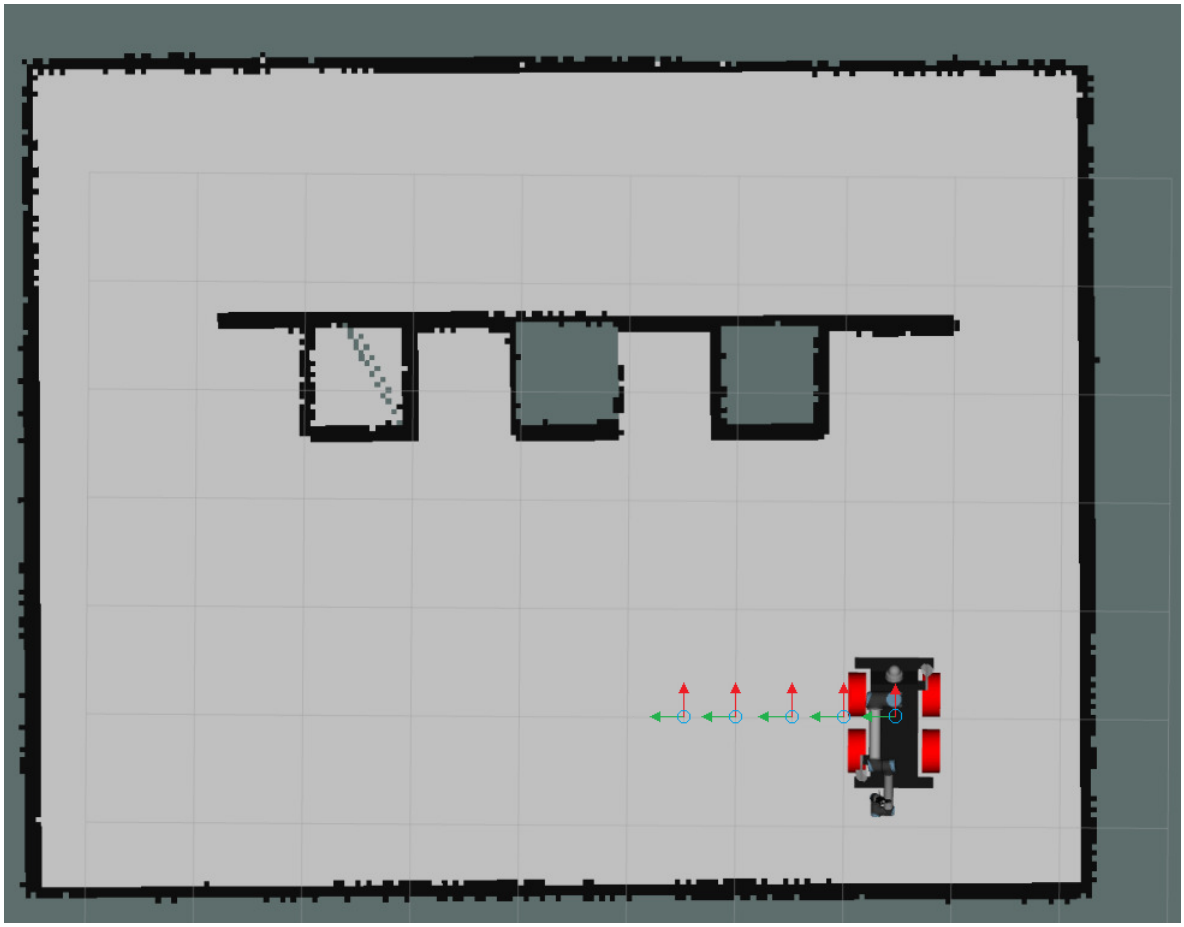

Figura 5.1: Posiciones de la trayectoria rectilínea

	Mediciones			Error absoluto		
réplica	x (m)	y (m)	θ (°)	x (cm)	y (cm)	θ (°)
1	-0.061	-2.326	0.458	6.1	17.4	0.458
2	-0.019	-2.367	0.364	6.1	13.3	0.364
3	-0.025	-2.354	0.355	2.5	14.6	0.355
4	-0.027	-2.321	-0.153	2.7	17.9	0.153
5	-0.025	-2.361	0.235	2.5	13.9	0.235
6	-0.016	-2.346	0.318	1.6	15.4	0.318
7	-0.009	-2.367	0.164	0.9	13.3	0.164
8	-0.007	-2.334	0.037	0.7	16.6	0.037
9	-0.008	-2.343	0.001	0.8	15.7	0.001
10	-0.010	-2.352	-0.121	1.0	14.8	0.121

Tabla 5.2: Tabla de medidas, trayectoria rectilínea

En la segunda trayectoria el robot recorrerá las siguientes posiciones: (0.0,1.0,90),(1.0,1.0,0),(1.0,-1.0,-90),(0.0,-1.0,180),(0.0,0,0)

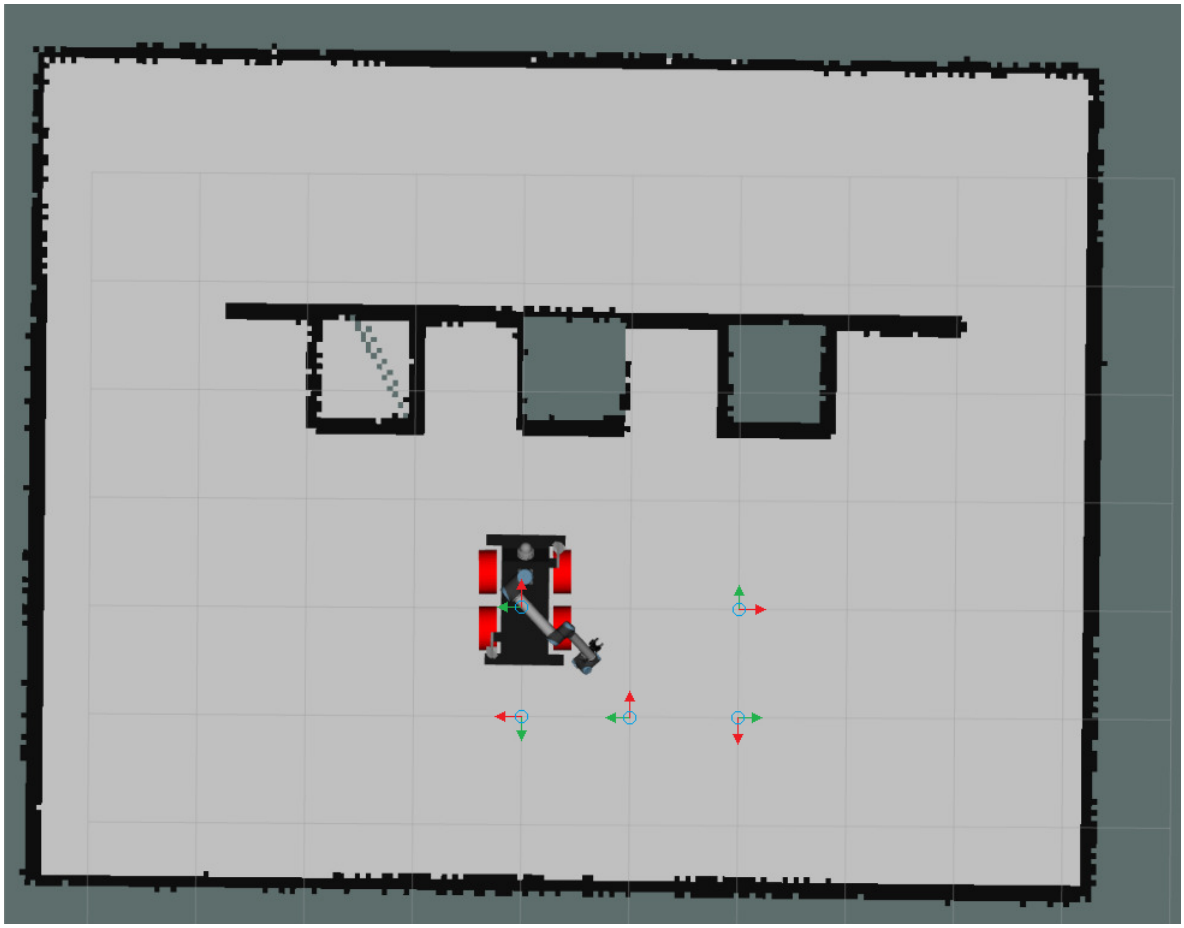

Figura 5.2: Posiciones de la trayectoria rectangular

	Mediciones			Error absoluto		
réplica	x (m)	y (m)	θ (º)	x (cm)	y (cm)	θ (º)
1	0.012	-0.011	-4.331	1.2	1.1	4.331
2	0.025	-0.004	-4.878	2.5	0.4	4.878
3	0.030	-0.007	-5.260	3.0	0.7	5.260
4	0.019	-0.022	-5.186	1.9	2.2	5.186
5	0.024	-0.001	-5.149	2.4	0.1	5.149
6	0.016	0.004	-4.637	1.6	0.4	4.637
7	0.001	-0.011	-3.329	0.1	1.1	3.329
8	0.021	0.013	-4.692	2.1	1.3	4.692
9	0.015	-0.014	-4.078	1.5	1.4	4.078
10	0.013	0.007	-5.514	1.3	0.7	5.514

Tabla 5.3: Tabla de medidas, trayectoria rectangular

Como en el experimento anterior, vemos que el error no aumenta conforme aumenta el número de repeticiones. Además, el orden de magnitud del error máximo concuerda con la tolerancia del planificador. En la trayectoria rectilínea observamos como el error en la orientación es pequeño y como el error es más grande en el eje del desplazamiento, tal y como era de esperar. En la segunda trayectoria nos sorprende el error tan pequeño que se comete en la posición, de apenas unos pocos centímetros.

5.3. Estimación de la marca

Los experimentos anteriores han estado enfocados al planificador de trayectorias. Ahora es momento de comprobar la precisión que existe en la estimación de las marcas ArUco. Porque cuando usemos la navegación con cámara y láser, el error de la estimación de la marca se acumulará con la tolerancia del error de posición del planificador.

Para ello, vamos a analizar a qué distancias podemos asumir que la estimación de la marca es precisa. Colocaremos al robot frente a la marca a distintas distancias, para cada una de ellas usaremos el programa 'transform_listener.py', programa que hemos creado para ver la transformación entre el robot y la marca. Este programa nos mostrará por pantalla constantemente la posición de la marca respecto al robot.
En la siguiente tabla vamos a ver cuanto varía la estimación de la posición de la marca en función de la distancia del robot a la marca.

	Variación		
distancia (m)	x (cm)	y (cm)	θ (grados)
3,3	8	1	20
2,8	1	0,2	9
2,3	0,8	0,1	8
1,8	0,4	0,04	4
1,3	0,15	0,03	0,5

Tabla 5.4: Precisión de la detección

Estos datos nos son muy útiles, por una parte vemos que aunque el robot esté a 2,8 metros la estimación de la distancia del robot a la marca es bastante precisa (dado que solo varía un centímetro). Por el contrario, la orientación no lo es tanto. Sabiendo esto, si queremos que el robot esté bien orientado respecto a la marca, tendremos primero que acercarlo a ella (como hemos hecho en la navegación mediante cámara). Para posiciones próximas a la marca como a 1,3 metros de distancia (hay que tener en cuenta que las distancias en el eje x se miden desde el centro del robot a la marca, no desde la cámara a la marca) la estimación es muy precisa. Haciendo el error prácticamente despreciable respecto a la tolerancia del planificador (20 cm de posición y 9 grados en la orientación).

Por lo tanto, en este experimento hemos podido comprobar que los errores en la estimación de la posición de la marca (y por lo tanto de la posición del robot respecto a ella) son muy precisos a distancias cortas. Por lo que la navegación mediante sólo cámara es mucho más precisa que la navegación que usa el mapa. Hay que tener en cuenta, que en la navegación mediante cámara y láser se sumará el error de la estimación de la posición de la marca al error de posición del planificador de trayectorias, por lo que es la navegación menos precisa.

5.4. Navegación multimarca

Hasta ahora sólo hemos trabajado utilizando una marca. Aunque podamos detectar un único identificador, podemos usar varias marcas siempre y cuando utilicen el mismo identificador. En este experimento vamos a usar cuatro marcas. El robot se acercará a ellas y luego girará 90º para visualizar la siguiente marca a la que debe dirigirse (ver código en el anexo A.5). Este programa se ejecutará en bucle, de manera que podremos comprobar como de robusta es la detección y si hay peligro de que en algún momento falle.

Para este experimento lo primero que hemos tenido que hacer es un entorno 3D nuevo en Gazebo (ver imágenes en el anexo B.3 figuras B.18 y B.19). El entorno está constituido por cuatro paredes y cuatro marcas ArUco suspendidas en el aire.

El experimento duró alrededor de unos 40 minutos hasta que lo detuvimos manualmente. Por lo que podemos afirmar que el programa es robusto. A lo largo del experimento no hubo que intervenir en ningún momento, así que el experimento fue un éxito. También nos sirve para afirmar que, al menos en simulación, para una distancia de unos 2,5 metros, la detección de las marcas es correcta.

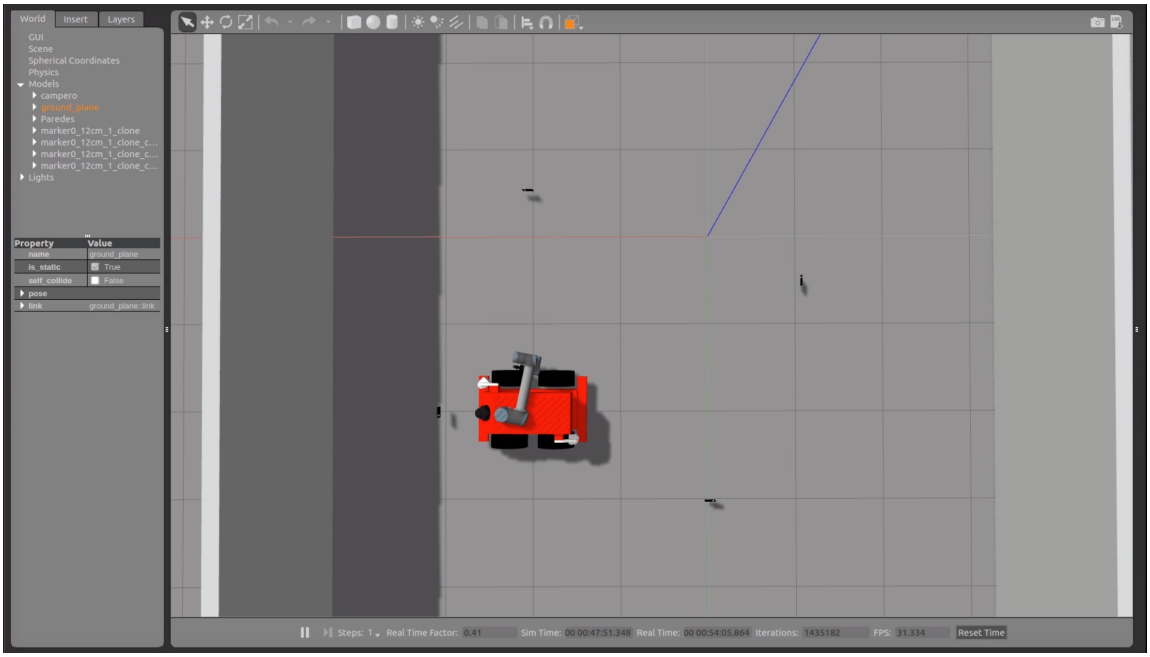

Figura 5.3: Llegada a la primera marca con 'multimarca_loop.py'

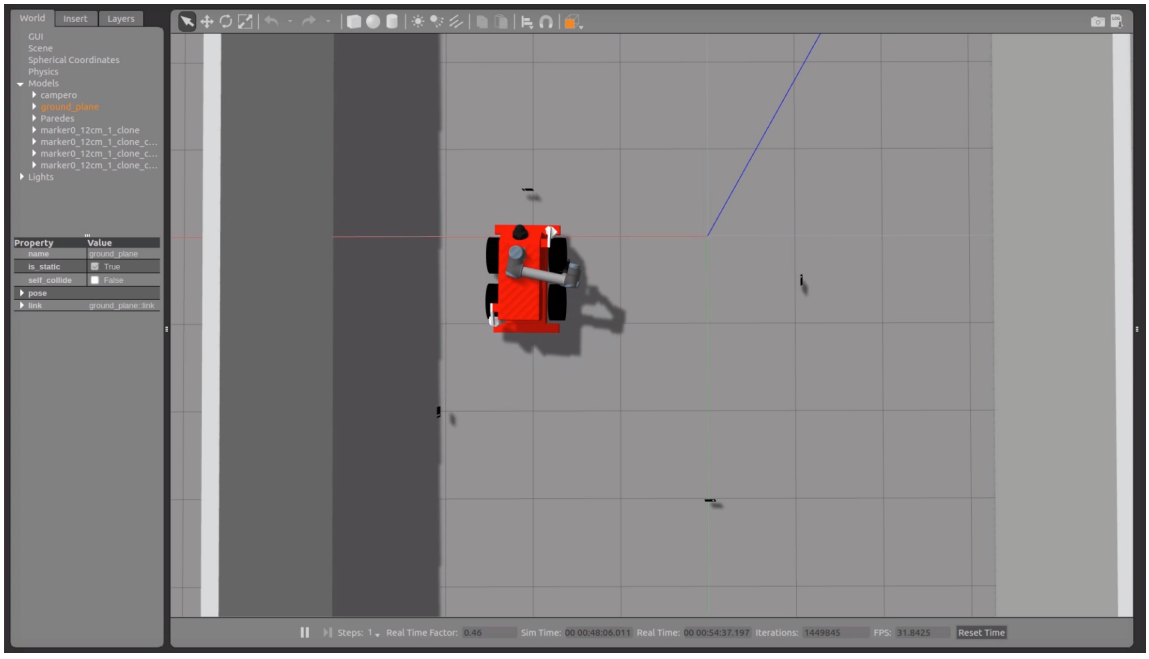

Figura 5.4: Llegada a la segunda marca con 'multimarca_loop.py'

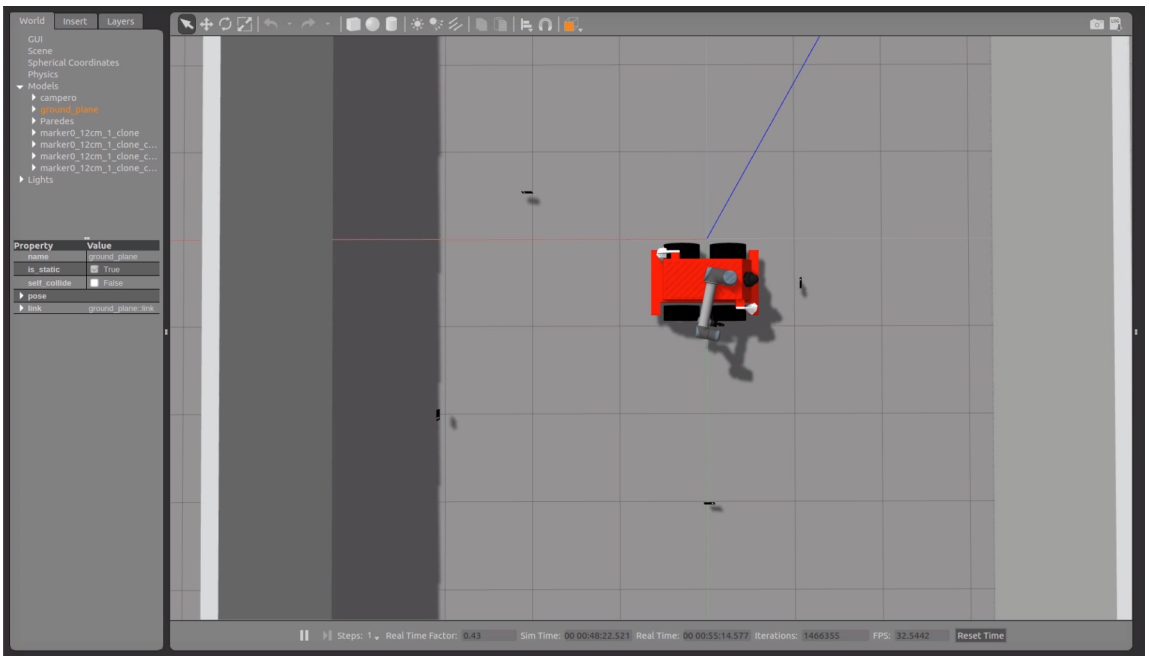

Figura 5.5: Llegada a la tercera marca con 'multimarca_loop.py'

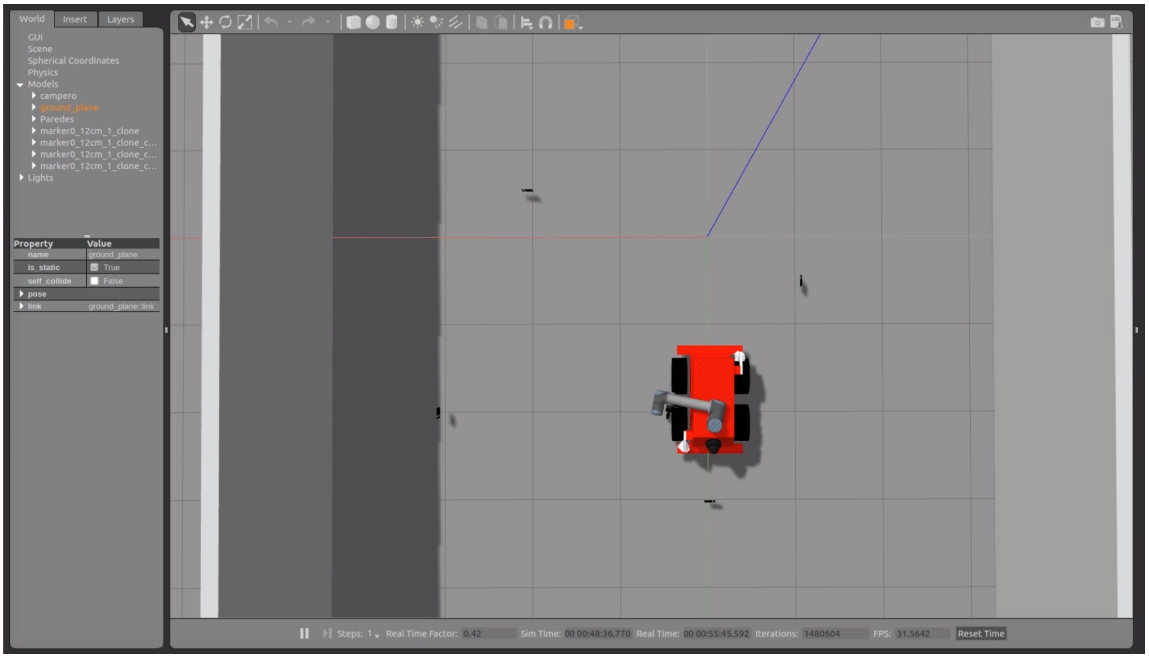

Figura 5.6: Llegada a la cuarta marca con 'multimarca_loop.py'

Capítulo 6

Implantación

En este capítulo vamos a hablar sobre como implementar los distintos tipos de navegación que hemos comentado anteriormente en este trabajo. Además damos acceso a un enlace [16] dónde ver distintos vídeos sobre los experimentos que se abordan en este capítulo.

6.1. Seguridad

Antes de hablar sobre como implementar los programas que hemos creado en simulación, vamos a hablar sobre las distintas medidas de seguridad que posee el robot. En simulación, como no había ningún riesgo, prácticamente no se ha hablado de medidas de seguridad. Pero en el robot real, existen varias medidas que debemos tener en cuenta. Para empezar, si queremos mover el robot tendremos que rearmarlo con el botón verde del mando de seguridad (ver figura 6.1). Cuando el robot esté bloqueado parpadeará la luz naranja central del panel trasero del robot (figura 6.3). Para bloquearlo en caso de emergencia tenemos la seta tanto del mando como del panel trasero. Aparte de usando las setas de emergencia, el robot también se detendrá si detecta un obstáculo cerca, siempre y cuando tengamos activada la seguridad de los láseres. Para activar o desactivar este sistema necesitamos introducir una llave en el panel trasero, depende en que posición la dejemos girada se activará o desactivará este sistema. Hay que tener en cuenta que la distancia de detección varia de la velocidad a la que se mueva el robot. Como último recurso, si los sistemas de seguridad anteriores fallasen el robot tiene bumpers tanto en su parte delantera como en su parte trasera (figura 6.4).

Además de los sistemas de seguridad ya comentados, el propio pad con el que moveremos el robot también cuenta con medidas de seguridad. Para realizar cualquier acción con el mando hace falta tener pulsado un botón, en este caso el botón R1 del mando 'DualShock 4' (ver figura 6.2). Por ejemplo, para mover el robot hace falta usar

los *joysticks* mientras mantenemos pulsado el botón de seguridad. El mando también cuenta con cinco niveles de velocidad, con el más bajo el robot no se moverá aunque movamos el joystick. En general con los dos primeros niveles de velocidad es suficiente. Recordemos que a mayor velocidad, mayor es la distancia con la que el robot se detiene si la seguridad de los láseres está activada. Por lo que usar velocidades altas no nos ahorrará tiempo si no probablemente todo lo contrario. Solo recomendamos usar niveles de velocidad superiores al segundo para movimientos puramente rotacionales.

Figura 6.1: Mando de seguridad

Figura 6.2: Pad para controlar el Campero

Figura 6.3: Panel trasero del Campero

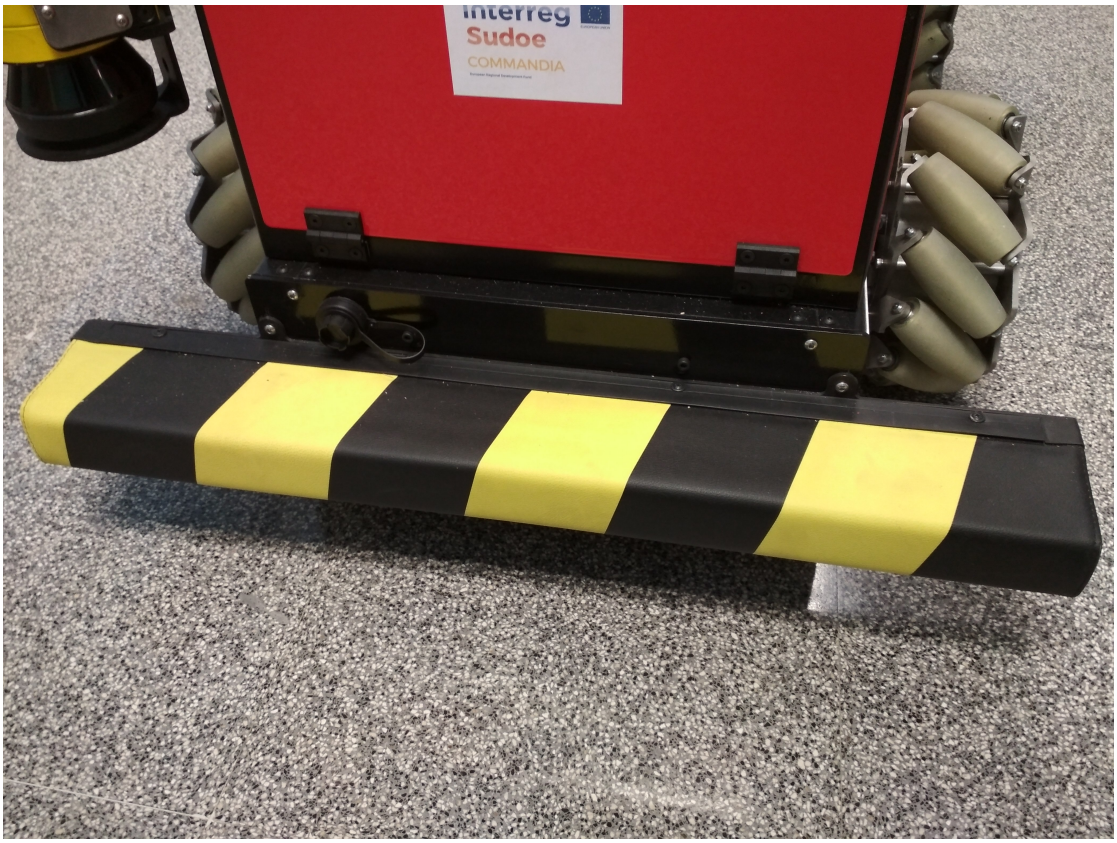

Figura 6.4: Bumper delantero del Campero

6.2. Puesta en marcha

El Campero se compone de una plataforma robótica y de un brazo robótico, cada uno con una CPU independiente. Como nosotros solo usaremos la plataforma, para la implementación solo necesitaremos encender dicha GPU. Para llevar al robot a la zona en la que queramos trabajar emplearemos un pad Bluetooth 'DualShock 4' (figura 6.2), este pad publican en el tópico 'campero/pad_teleop/cmd_vel'. Por motivos de seguridad, siempre que el robot este en movimiento alguien deberá portar el mando de seguridad y estar listo para pulsar la seta de emergencia en cualquier momento. La función de seguridad de los láseres debe estar siempre activada excepto cuando sea estrictamente necesario desactivarlos, por ejemplo cuando tenga que pasar por una puerta.

Problemas con la cámara PTZ

Figura 6.5: Cámara PTZ

A la hora de implementar la detección de marcas ArUco nos encontramos con que la estimación de la posición de lar marcas no se realizaba correctamente. Esto es debido a que la localización de la cámara del robot no era la correcta. Nos dimos cuenta de que la transformación '[...]/optical_frame' no estaba orientada igual que en simulación. Además, la posición en la que se encontraba la transformación era extraña, ya que la transformación estaba fuera de la cámara. Tras contactar con la empresa Robotnik nos proporcionaron un archivo nuevo. Al cambiarlo por el antiguo había diversos errores, de manera que solo cambiamos los valores de las transformaciones de la cámara en el archivo antiguo por la que había en el nuevo. Con estos cambios la transformación ya se localizaba dentro de la cámara pero la orientación seguía siendo distinta a la de la

simulación, y distinta al resto de las demás cámaras que tiene el robot. En las demás cámaras el eje Z se sitúa en la dirección en la que apunta la cámara, en la cámara PTZ en simulación también pero en el robot real no. Volvimos a contactar con la empresa y se pusieron a investigarlo. Hasta que nos ayudasen a solucionar el problema decidimos usar la cámara trasera del robot ('Intel Real Sense D435' [17]). Para poder usarla hay que mover el brazo como se muestra en la figura 6.6.

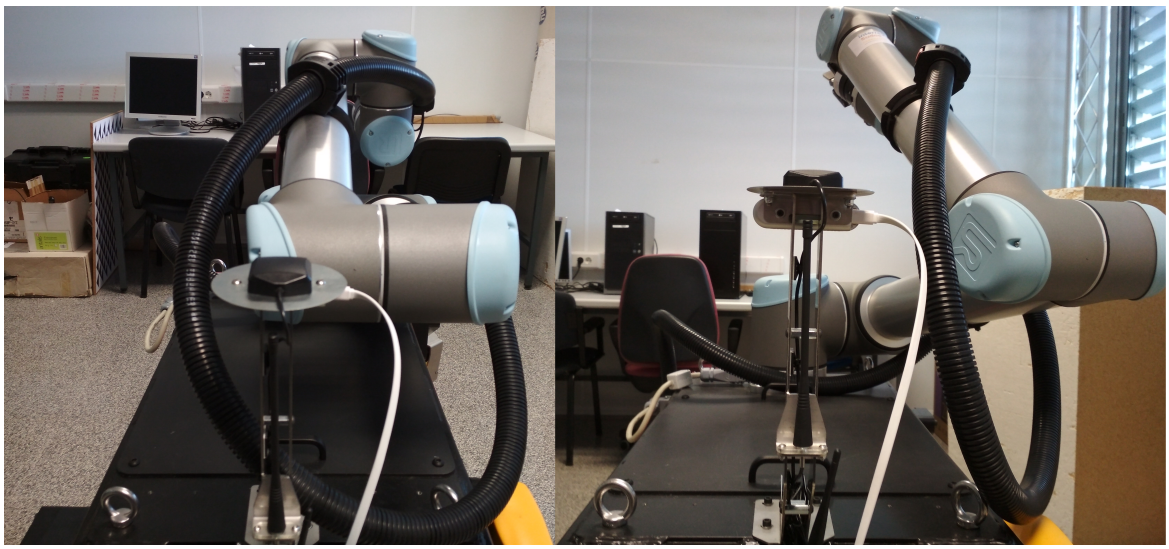

Figura 6.6: Izquierda: Brazo recogido. Derecha: Posición del brazo para el uso de la cámara

6.3. Experimentación

Una vez tengamos el robot en la zona de trabajo, tendremos que acceder a la interfaz web de Robotnik a través del navegador. Si ya tenemos el mapa del entorno guardado, escribimos el nombre del archivo y desde la interfaz lanzamos los procesos 'Map Server', 'Localization' y 'Navigation'. Si no queremos usar mapa, el robot cuenta con uno vacío llamado 'map_empty.yaml'.

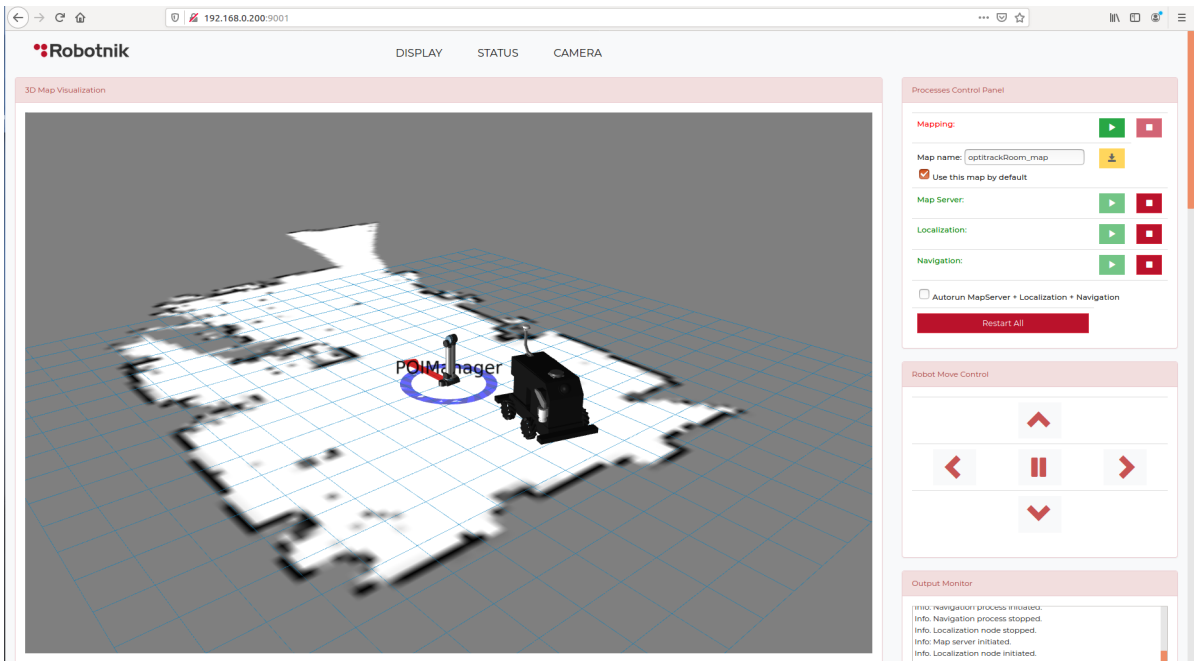

Figura 6.7: Interfaz web de Robotnik

Si queremos crearlo, con todos los procesos detenidos y el mapa vacío, lanzamos el proceso 'Mapping'. Veremos como se empieza a dibujar el mapa en la pantalla. Vamos moviendo el robot hasta que consideremos que el mapa ya está acabado. Para guardarlo, escribimos el nombre que queremos darle al mapa en 'Map name', pulsamos el botón amarillo con el símbolo de descargar, comprobamos que se haya guardado (se guardan en el directorio: 'src/campero/campero_common/campero_localization/maps') y ya podemos detener el proceso de 'Mapping'.

Para cargar nuestros programas, una vez tenemos los procesos en marcha, nos conectaremos a la GPU del robot desde un escritorio remoto. Ahora, simplemente abrimos un terminal y ya podemos probar nuestros programas. Lo primero que lanzaremos será RViz para visualizar toda la información que el robot posee del entorno. Cuando ya esté en marcha, ya podemos proceder a probar los programas.

6.3.1. Navegación mediante percepción láser

Antes de lanzar los programas de navegación, tenemos que asegurarnos de que el planificador de trayectorias tiene los valores límite de la velocidad adecuados. Los valores que nosotros hemos usado han sido 0.2 m/s para la velocidad lineal y 0.5 rad/s para la angular. Para elegir la posición objetivo del robot, lo mejor es seleccionar la opción 'Publish Point' de la barra superior de RViz. Una vez seleccionada ponemos el cursor sobre el punto queremos que el robot se dirija y en la esquina inferior izquierda de RViz vemos a que coordenada corresponde. Ahora ya podemos abrir los programas

'actionGoal.py' o 'multipleGoals.py' y poner las coordenadas X e Y del 'goal' al que queremos que se dirija el robot, así como la orientación (más información sobre estos programas en la sección C.2 o en los anexos A.1 y A.2). Si la esta navegación se ha simulado correctamente, no debería haber ningún problema en la ejecución con el robot real. En el robot real, en RViz aparecerá una línea verde que indica la trayectoria que seguirá el robot (ver figura 6.8).

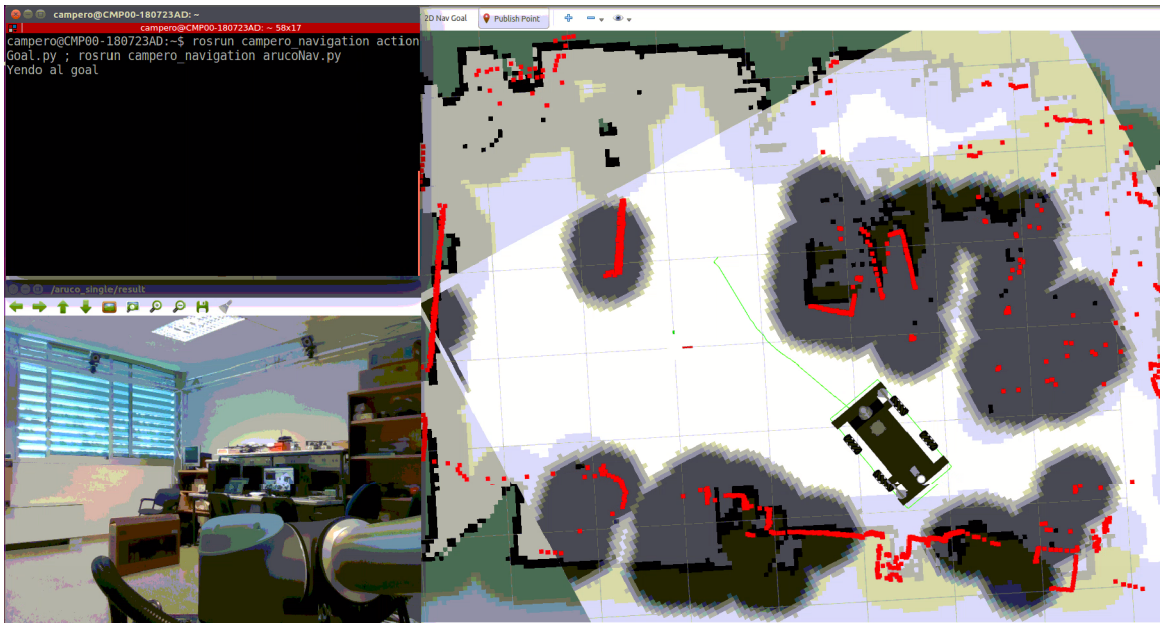

Figura 6.8: Ejecución de 'actionGoal.py'

6.3.2. Navegación mediante percepción láser y cámara

Para esta navegación, antes de lanzar los programas, tendremos que tener en cuenta los valores de las velocidades máximas como en la sección anterior. Pero además, tendremos que acudir al archivo 'single.launch' y asegurarnos de que los nombre de los tópicos del robot están bien (para más información sobre como modificar este archivo volver a la sección C.3 y para ver los nombres de las distintas transformaciones ir a la sección 3.3). Dado que los nombres de algunas transformaciones no son iguales a los de la simulación en el robot real, pero siguen siendo perfectamente reconocibles. Una vez configurados todos los archivos, ya podemos lanzar los programas tal y como hicimos en simulación (sección C.3).

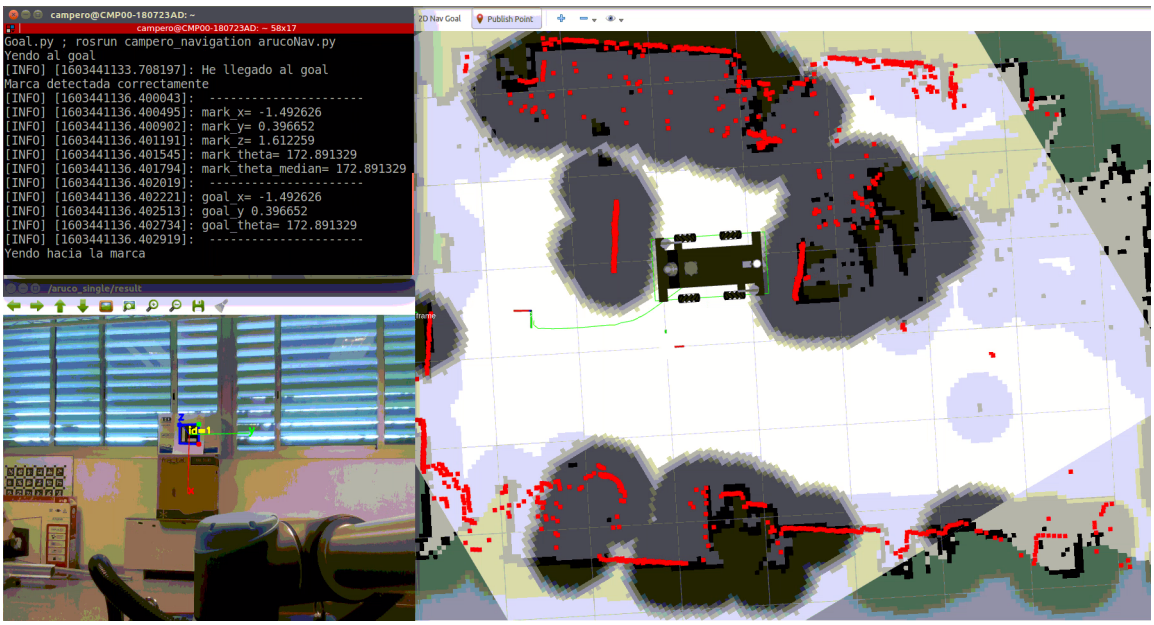

Figura 6.9: Ejecución de 'arucoNav.py'

Debido al trabajo invertido en simulación, la implementación ha funcionado correctamente.

6.3.3. Navegación mediante cámara

La única diferencia con la simulación para esta navegación, ha sido la reducción de las velocidades en cada fase. Como ya comentamos en su momento, esta navegación no es dinámica, si aparece un obstáculo el robot no lo esquivará. Pero, si tenemos el sistema de seguridad de los láseres activados, el robot se detendrá si detecta un obstáculo con el que pueda colisionar. Los programas se tienen que lanzar en el mismo orden y bajo las mismas condiciones que en simulación, si es necesario revisar la sección C.4. Tras varias pruebas, hemos comprobado que el programa funciona correctamente.

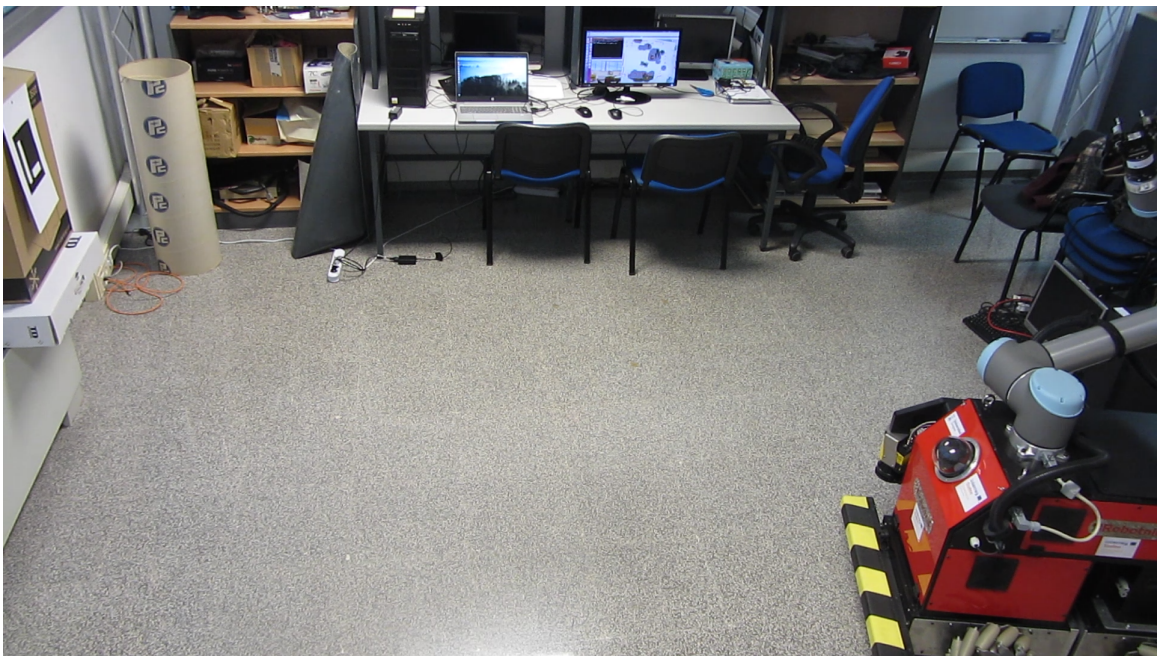

Figura 6.10: Posición inicial

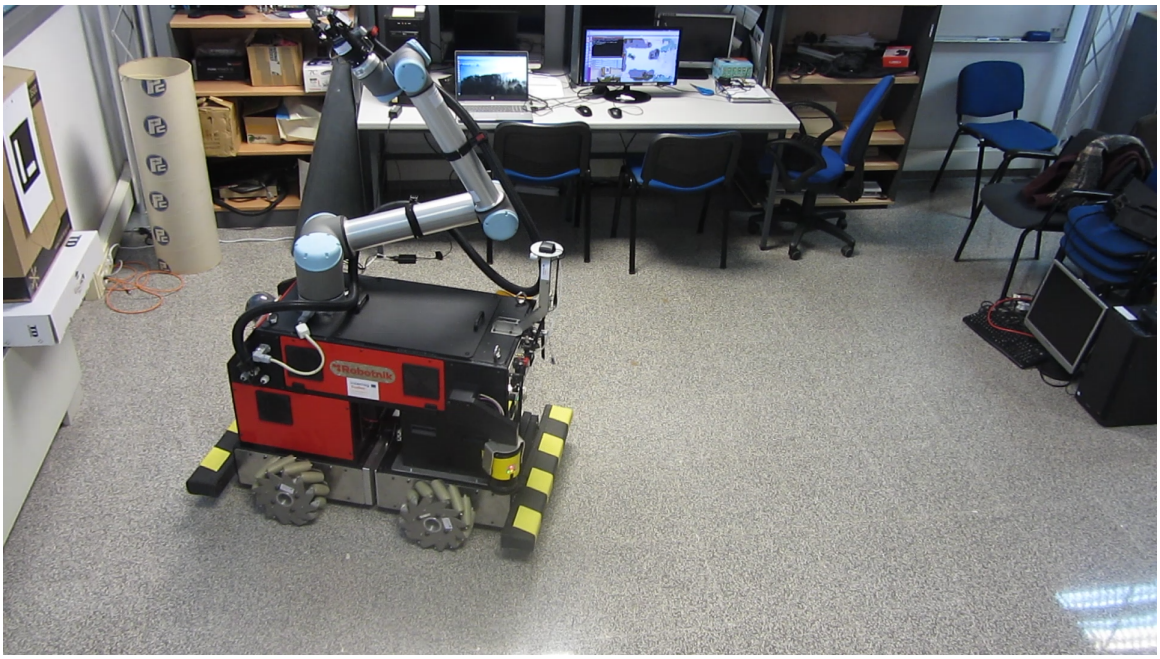

Figura 6.11: Fase 1 de la navegación (acercamiento a la marca)

Figura 6.12: Fase 2 de la navegación (movimiento circular)

Figura 6.13: Fase 3 de la navegación (posición final)

6.3.4. Combinando los distintos métodos de navegación

Como ya dijimos en la sección 4.4 lo ideal es mezclar los distintos tipos de navegación para aprovechar las ventajas de cada una. Y esto es lo que hemos hecho en los experimentos. Antes de lanzar los métodos de navegación que emplean la cámara, enviamos al robot a una posición en la que sabemos que ve la marca. Para mezclar los distintos tipos de navegación no ha hecho falta crear ningún método nuevo. Simplemente cuando lanzamos los programas desde la terminal, los lanzamos de manera consecutiva. Para ello tenemos que escribir ';' en la terminal, entre los programas que se van a lanzar. Por ejemplo:

$ rosrun campero navigation actionGoal.py ; rosrun campero navigation arucoCmd.py

Con este comando el robot se aproxima a una posición desde la que ve la marca. A continuación, se inicia la navegación con cámara. En las siguientes imágenes mostramos el proceso:

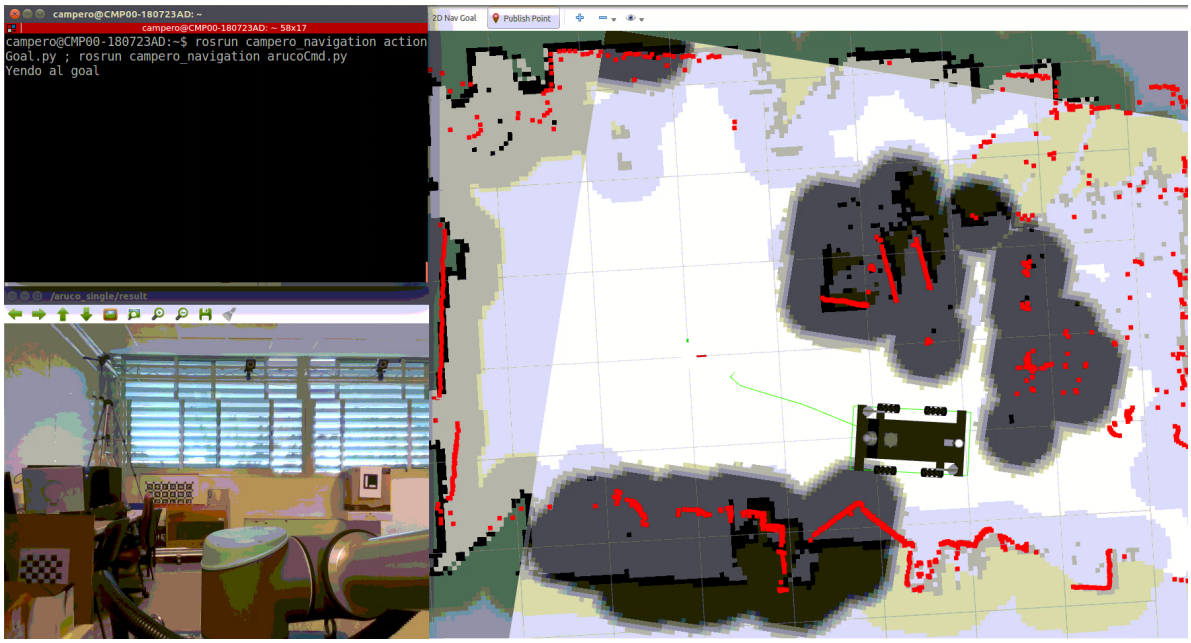

Figura 6.14: Inicio del primer programa

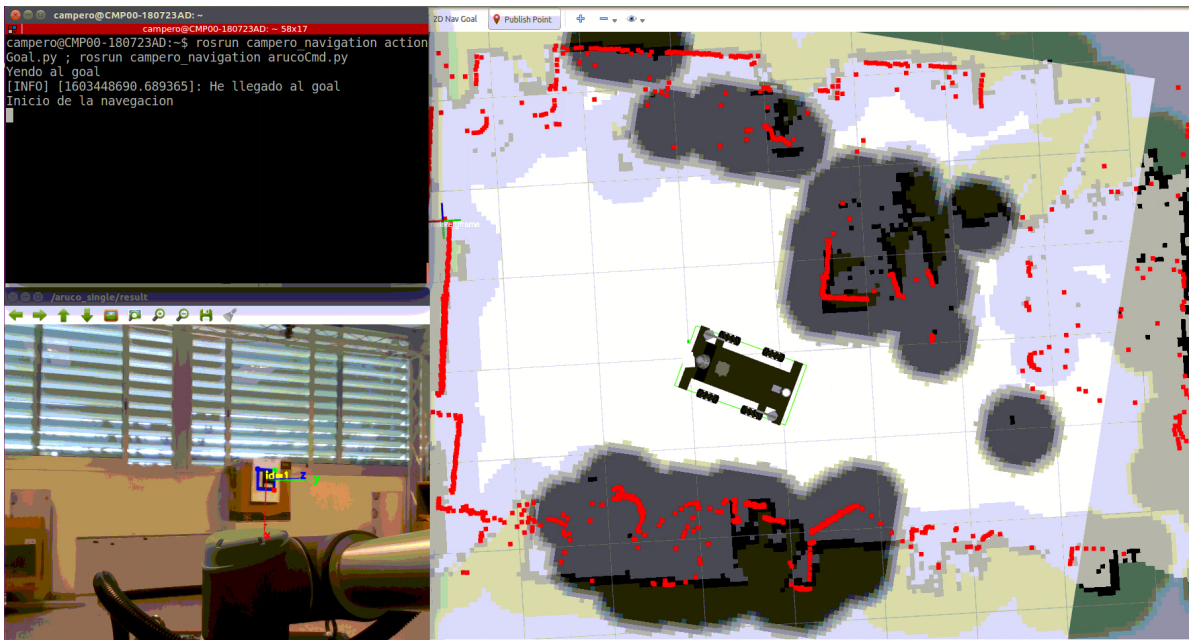

Figura 6.15: Fin del primer programa e inicio del segundo

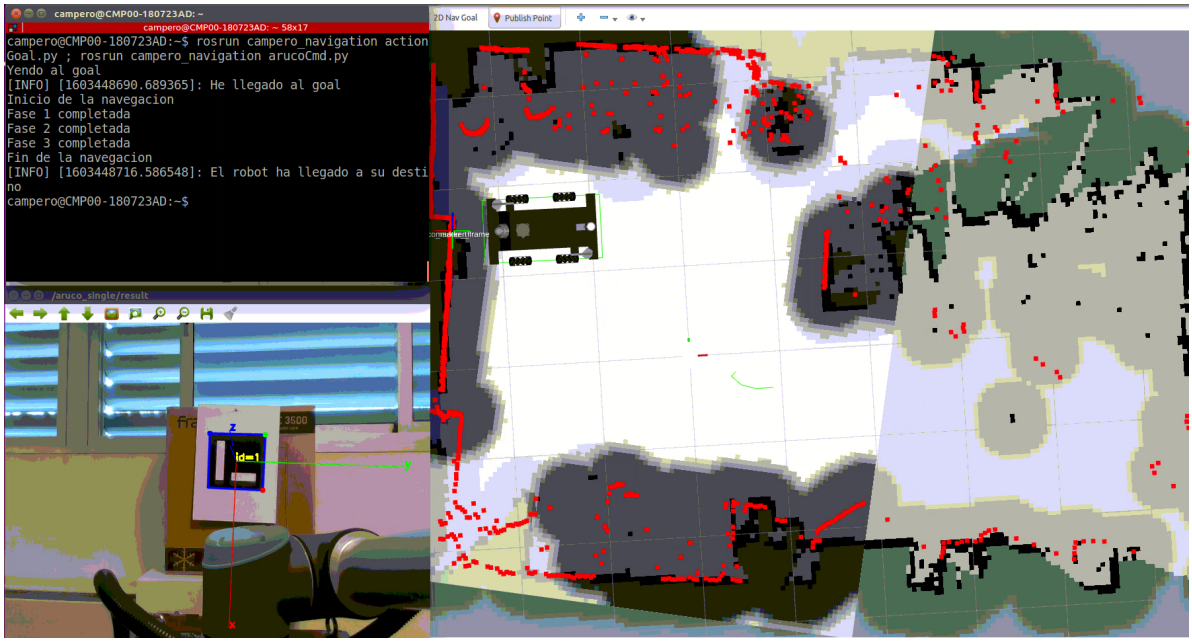

Figura 6.16: Fin del segundo programa

En este ejemplo solamente hemos usado dos tipos de navegación, pero lo ideal sería usar los tres. De manera que haya un aproximación inicial hasta ver la marca, luego una segunda aproximación utilizando los láseres y un ajuste final del error de posición con la navegación con uso exclusivo de la cámara. Para llevarlo acabo bastaría con escribir los comandos correspondientes separados por ';'. Igual que en el experimento que acabamos de mostrar.

Capítulo 7

Conclusiones y trabajo futuro

7.1. Conclusiones

Tras horas de trabajo, podemos afirmar que hemos cumplido ampliamente con nuestros objetivos. Hemos logrado utilizar varios métodos de navegación autónoma en el robot Campero. Para llegar hasta aquí hemos tenido que: Familiarizarnos con la distribución de Linux ubuntu 16.04, familiarizarnos con el entorno ROS y sus herramientas, aprender a crear marcas visuales con software de creación de gráficos tridimensionales, crear entornos de simulación tridimensionales en Gazebo, desarrollar código en Python para la navegación, trabajar con lenguaje XML para modificar diversos archivos, implementar paquetes de uso libre en el proyecto, simular sistemas de navegación autónoma en ROS, mapear entornos tanto simulados como reales e implementar en el robot Campero los sistemas de navegación que habíamos simulado. Llegados a este punto, podemos afirmar que la filosofía ROS funciona. Existe una comunidad activa, que ayuda tanto con el desarrollo de paquetes como resolviendo dudas en foros. Además, si se hace un buen trabajo en simulación, la mayor dificultad para implementarlo en un sistema real es que hay que familiarizarse con el hardware. Una vez hecho esto, la metodología de trabajo es prácticamente idéntica a la que se usa en simulación. Esto lleva a que lo aprendido en este trabajo pueda usarse en otros robots a parte del Campero, en un tiempo razonable.

7.2. Trabajo futuro

En este proyecto hemos aprendido mucho sobre el robot Campero y el entorno ROS. Por eso en este documento se ha plasmado todo lo aprendido. De manera que en futuros proyectos se pueda ahorrar tiempo de búsqueda, gracias a toda la información recogida en este documento. Además de información, también hay tutoriales detallados en los anexos para replicar diversas situaciones que se han planteado en este proyecto.

Una vez sabemos lo principal sobre el robot y el entorno ROS, se abren muchas opciones para futuros proyectos tanto para la simulación como para la implementación. El más obvio es el uso del brazo robótico: Ahora que el robot tiene mecanismos básicos de navegación sería muy interesante usar el brazo. Además, se puede aprovechar lo aprendido con las marcas visuales. En cuanto a la cámara, sería interesante plantear el uso de mapas ArUco en vez de usar marcas individuales, incluso hacer un estudio sobre qué marca es mejor usar dependiendo las circunstancias. También se debería intentar, en una misma ejecución poder estimar la posición de más de una marca. Respecto a la navegación, sería interesante estudiar las posibilidades que ofrecen los planificadores de trayectorias. Ya que en el campero, hay más de un planificador incorporado.

Capítulo 8

Bibliografía

[1] Robotnik. RB-EKEN. https://bit.ly/3pDj34j, 2020.

[2] ROS. https://www.ros.org/, 2020.

[3] COMMANDIA. http://commandia.unizar.es/lo-basico-de-commandia/, 2020.

[4] Jorge Playán Garai. Navegación de robots manipuladores en el entorno de ROS. Trabajo fin de grado, Universidad de Zaragoza, 2019.

[5] Francisco Romero-Ramirez, Rafael Muñoz-Salinas, and Rafael Medina-Carnicer. Speeded up detection of squared fiducial markers. *Image and Vision Computing*, 76, 06 2018.

[6] Sergio Garrido-Jurado, Rafael Muñoz-Salinas, Francisco Madrid-Cuevas, and Rafael Medina-Carnicer. Generation of fiducial marker dictionaries using mixed integer linear programming. *Pattern Recognition*, 51, 10 2015.

[7] Sergio Garrido-Jurado, Rafael Muñoz-Salinas, Francisco Madrid-Cuevas, and Rafael Medina-Carnicer. Aruco. http://www.uco.es/investiga/grupos/ava/node/26, 12-8-2020.

[8] Pyo YoonSeok, Cho HanCheol, Jung RyuWoon, and Lim TaeHoon. *ROS Robot Programming*. ROBOTIS Co.,Ltd., 2017.

[9] Cómo agregar una textura a un objeto en Blender 2.8. https://www.youtube.com/watch?v=r5YNJghc81U, 2020.

[10] Robotnik. https://robotnik.eu/es/, 2020.

[11] PAL Robotics. Paquete aruco_ros. https://bit.ly/2Hbpr1a, 2020.

[12] ROS. Paquete move_base. https://bit.ly/38S1RSG, 2020.

[13] ROS. Paquete eband_local_planner. https://bit.ly/38XbGij, 2020.

[14] ROS. Documentación mensaje MoveBaseGoal. https://bit.ly/2Kl2xFV, 2020.

[15] ROS. Documentación mensaje Twist. https://bit.ly/32Roi6r, 2020.

[16] David Barrera. Vídeos robot Campero. https://bit.ly/3930CAh, 2020.

[17] Intel Real Sense. D435. https://www.intelrealsense.com/depth-camera-d435/, 2020.

Lista de Figuras

Lista de Tablas

Anexos

Anexos A

Código utilizado para la navegación

En este anexo vamos a incluir el código de los programas más importantes que hemos creado en simulación. En el encabezado de cada programa hay un enlace al código en el que nos hemos basado, si no lo hay es que lo hemos creado a partir de nuestros propios programas. Los programas en los que solo ha habido que realizar unas pequeñas modificaciones no aparecerán aquí, dado que no son de interés. Hemos decidido colocar los programas en el anexo porque no es tan importante el código de los mismos, si no que sean funcionales y nos permitan alcanzar los objetivos propuesto en este proyecto. Todos los programas que aparecen en este anexo han sido usados para la navegación y se encuentran en el directorio: *src/campero/campero_common/campero_navigation/scripts*

Para la ejecución de estos programas hace falta lanzar primero el archivo *campero_nav.launch*.

A.1. actionGoal.py

Este programa se encarga de usar una acción para que el robot se dirija a la posición que le indicamos en el propio programa. Cuando se inicia el programa el robot muestra un mensaje de que está yendo a la posición objetivo. Cuando se cumple la acción el programa muestra por pantalla que ya ha llegado a la posición y se cierra el programa. Si abortamos el programa el robot se detiene y se muestra por pantalla que el programa se ha cancelado.

A continuación se muestra el código completo:

```
1 #!/usr/bin/env python
2 """ actionGoal.py - Version 1.0 10-8-2020
3     Autor: David Barrera
4     Codigo modificado a partir de: https://hotblackrobotics.github.io/en/blog/2018/01/29/action-client-py/
```

```
5  """
6
7  import rospy
8  import math
9  import time
10 import numpy
11 # Brings in the SimpleActionClient
12 import actionlib
13
14 from geometry_msgs.msg import Quaternion
15 from tf.transformations import quaternion_from_euler
16 # Brings in the .action file and messages used by the move base action
17 from move_base_msgs.msg import MoveBaseAction, MoveBaseGoal
18
19
20 goal = MoveBaseGoal()
21
22 # Coordenadas a las que se dirigira el robot
23 x_goal=0.0
24 y_goal=0.0
25 theta_goal=0
26
27 def goal2MoveBaseGoal(x=0.0, y=0.0, theta=0): #Funcion que transforma una posicion en formato MoveBaseGoal
28
29     goal.target_pose.header.frame_id = "campero_map"
30     goal.target_pose.header.stamp = rospy.Time.now()
31
32     #Definimos la posicion
33     goal.target_pose.pose.position.x = x
34     goal.target_pose.pose.position.y = y
35     goal.target_pose.pose.position.z = 0.0
36
37     #Definimos la orientacion
38     q=quaternion_from_euler(0.0, 0.0, numpy.deg2rad(theta))
39     goal.target_pose.pose.orientation=Quaternion(*q)
40
41
42 def movebase_client():
43
44    # Create an action client called "move_base" with action definition file "MoveBaseAction"
45     client = actionlib.SimpleActionClient('/campero/move_base',MoveBaseAction)
46
47    # Waits until the action server has started up and started listening for goals.
48     client.wait_for_server()
49
50     goal2MoveBaseGoal(x_goal,y_goal,theta_goal)   #Creamos el goal en el formato MoveBaseGoal
51
52    # Sends the goal to the action server.
53     client.send_goal(goal)
54     print("Yendo al goal")
55
56    # Waits for the server to finish performing the action.
57     wait = client.wait_for_result()
58
59    # If the result doesn't arrive, assume the Server is not available
60     if not wait:
61         rospy.logerr("Action server not available!")
62         rospy.signal_shutdown("Action server not available!")
63     else:
64     # Result of executing the action
65         return client.get_result()
66
67
68     # If the python node is executed as main process (sourced directly)
69 if __name__ == '__main__':
70     try:
71         # Initializes a rospy node to let the SimpleActionClient publish and subscribe
72         rospy.init_node('movebase_client_py')
73         result = movebase_client()
74         if rospy.is_shutdown():
75             rospy.loginfo("Se ha cancelado la ruta")
76         elif result:
77             rospy.loginfo("He llegado al goal")
78
79     except rospy.ROSInterruptException:
80         rospy.loginfo("Navigation test finished.")
```

A.2. multipleGoals.py

Esta programa tiene la misma función que actionGoal.py solo que aquí el robot recorre una secuencia de posiciones. Las posiciones a recorrer se indican en el propio programa y también se usa una acción cada vez que el robot se dirige a una posición. Cada vez que el robot va de camino a una posición de la secuencia, por pantalla se indica a cual esta yendo y cuando ha llegado. Si se cancela el programa también aparece un mensaje en pantalla.

El código completo se muestra a continuación:

```
#!/usr/bin/env python
""" multipleGoals.py - Version 1.0 10-8-2020
    Autor: David Barrera
    Codigo modificado a partir de:
    https://hotblackrobotics.github.io/en/blog/2018/01/29/action-client-py/
"""

import rospy
import math
import time
import numpy
# Brings in the SimpleActionClient
import actionlib

from geometry_msgs.msg import Quaternion
from tf.transformations import quaternion_from_euler
# Brings in the .action file and messages used by the move base action
from move_base_msgs.msg import MoveBaseAction, MoveBaseGoal

class goal:
  def __init__(self, x, y, theta):
    self.x=x
    self.y=y
    self.theta=theta

    self.createMoveBaseGoal() #El atributo que usaremos para la navegacion sera del tipo MoveBaseGoal


  def createMoveBaseGoal(self): #Damos valor al atributo MoveBaseGoal en funcion de x, y, theta
    self.MoveBaseGoal=MoveBaseGoal()

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

    #Definimos la posicion
    self.MoveBaseGoal.target_pose.pose.position.x = self.x
    self.MoveBaseGoal.target_pose.pose.position.y = self.y
    self.MoveBaseGoal.target_pose.pose.position.z = 0.0

    #Definimos la orientacion
    q=quaternion_from_euler(0.0, 0.0, numpy.deg2rad(self.theta))
    self.MoveBaseGoal.target_pose.pose.orientation=Quaternion(*q)


def movebase_client():

   # Create an action client called "move_base" with action definition file "MoveBaseAction"
  client = actionlib.SimpleActionClient('/campero/move_base',MoveBaseAction)

   # Waits until the action server has started up and started listening for goals.
  client.wait_for_server()

  goalSequence=[]     #Creamos el array que contiene la secuencia de goals
  goal1=goal(0.0,-0.5,0)   #Creamos el goal en el formato MoveBaseGoal
  goal2=goal(0.0,-1.0,0)
  goal3=goal(0.0,-1.5,0)
```

```
57   goal4=goal(0.0,-2.0,0)
58   goal5=goal(0.0,-2.5,0)
59
60   goalSequence=[goal1, goal2, goal3, goal4, goal5]
61   n=1
62
63   while not len(goalSequence)==0 and not rospy.is_shutdown():
64   # Sends the goal to the action server.
65     client.send_goal(goalSequence[0].MoveBaseGoal)
66     print("Yendo al goal"+str(n))
67
68   # Waits for the server to finish performing the action.
69     wait = client.wait_for_result()
70
71   # If the result doesn't arrive, assume the Server is not available
72     if not wait:
73       rospy.logerr("Action server not available!")
74       rospy.signal_shutdown("Action server not available!")
75     else:
76       # Result of executing the action
77       print("He llegado al goal"+str(n))
78       goalSequence.pop(0)   #Eliminamos el punto al que el robot acaba de llegar
79       n=n+1
80
81   return client.get_result()
82
83
84
85
86 # If the python node is executed as main process (sourced directly)
87 if __name__ == '__main__':
88     try:
89     # Initializes a rospy node to let the SimpleActionClient publish and subscribe
90     rospy.init_node('movebase_client_py')
91     result = movebase_client()
92     if rospy.is_shutdown():
93       rospy.loginfo("Se ha cancelado la ruta")
94     elif result:
95       rospy.loginfo("Se ha completado la ruta")
96
97     except rospy.ROSInterruptException:
98         rospy.loginfo("Navigation test finished.")
```

A.3. arucoNav.py

Este programa envía al robot a la posición de la transformación 'aruco_tf', es decir, de frente a la marca a un metro de distancia. El programa necesita que estén en funcionamiento 'aruco_single.launch', el comando image_view y que se use a la vez el programa 'transform_listener.py' para saber en qué posición se encuentra 'aruco_tf' (que se crea lanzando el programa 'aruco_tf.py'). Una vez el robot sabe donde dirigirse usa una acción como hacen los programas anteriores. Por pantalla se indica primero si el robot ha visto la marca, si no lo ha hecho hecho se informa por pantalla. Si el robot ve la marca muestra la posición en la que la ha visto y la posición a la que se dirige el robot. Como 'actionGoal.py' informa de cuando empieza a moverse hacia la marca y cuando llega a ella. Una vez llega frente a la marca el programa se termina, si el programa se cancela por el usuario también se informa por pantalla.

A continuación se encuentra el código completo:

```
#!/usr/bin/env python
"""
    arucoNav.py - Version 1.0 10-8-2020
    Autor: David Barrera
    Codigo modificado a partir de: https://hotblackrobotics.github.io/en/blog/2018/01/29/action-client-py/
"""

import rospy
import math
import time
import numpy
import tf
# Brings in the SimpleActionClient
import actionlib

from geometry_msgs.msg import Quaternion, PoseWithCovarianceStamped, PoseStamped
from tf.transformations import quaternion_from_euler
# Brings in the .action file and messages used by the move base action
from move_base_msgs.msg import MoveBaseAction, MoveBaseGoal


class arucoNavigation():
    def __init__(self):
        # Para el action
        self.goal = MoveBaseGoal()
        self.goal_x=0.0
        self.goal_y=0.0
        self.goal_theta=0.0

        # Para la posicion de la marca
        self.mark_x=0.0
        self.mark_y=0.0
        self.mark_z=0.0
        self.mark_alpha=0.0
        self.mark_beta=0.0
        self.mark_theta=0.0

        self.mark_theta_list=[]
        self.mark_theta_median=0.0



# Funciones para la navegacion
def goal2MoveBaseGoal(x=0.0, y=0.0, theta=0):
#Funcion que transforma una posicion dada por x, y e angulo en z (en grados), en formato MoveBaseGoal

```

```
47      arucoNav.goal.target_pose.header.frame_id = "campero_map"
48      arucoNav.goal.target_pose.header.stamp = rospy.Time.now()
49
50      #Definimos la posicion
51      arucoNav.goal.target_pose.pose.position.x = x
52      arucoNav.goal.target_pose.pose.position.y = y
53      arucoNav.goal.target_pose.pose.position.z = 0.0
54
55      #Definimos la orientacion
56      q=quaternion_from_euler(0.0, 0.0, numpy.deg2rad(theta))
57      arucoNav.goal.target_pose.pose.orientation=Quaternion(*q)
58
59
60
61  def movebase_client():
62
63     # Create an action client called "move_base" with action definition file "MoveBaseAction"
64      client = actionlib.SimpleActionClient('/campero/move_base',MoveBaseAction)
65
66     # Waits until the action server has started up and started listening for goals.
67      client.wait_for_server()
68
69      #Si los valores no se han actualizado es que la camara no ve la marca
70      if arucoNav.mark_x==0.0 and arucoNav.mark_y==0.0 and arucoNav.mark_theta==0.0:
71          marca_detectada=0
72          print('No detecto a la marca, cancela el programa y comprueba que transform_listener.py esta funcionando')
73      else:
74          marca_detectada=1
75          print('Marca detectada correctamente')
76
77      if marca_detectada:
78          arucoNav.goal_x = arucoNav.mark_x
79          arucoNav.goal_y = arucoNav.mark_y
80          arucoNav.goal_theta = arucoNav.mark_theta_median
81
82          # Mostramos la informacion por pantalla
83          rospy.loginfo(' ---------------------- ')
84
85          rospy.loginfo('mark_x= %f ', arucoNav.mark_x)
86          rospy.loginfo('mark_y= %f ', arucoNav.mark_y)
87          rospy.loginfo('mark_z= %f ', arucoNav.mark_z)
88          rospy.loginfo('mark_theta= %f ', arucoNav.mark_theta)
89          rospy.loginfo('mark_theta_median= %f ', arucoNav.mark_theta_median)
90
91          rospy.loginfo(' ---------------------- ')
92
93          rospy.loginfo('goal_x= %f ', arucoNav.goal_x)
94          rospy.loginfo('goal_y %f ', arucoNav.goal_y)
95          rospy.loginfo('goal_theta= %f ', arucoNav.goal_theta)
96
97          rospy.loginfo(' ---------------------- ')
98
99          #Creamos el goal en el formato MoveBaseGoal
100         goal2MoveBaseGoal(arucoNav.goal_x,arucoNav.goal_y,arucoNav.goal_theta)
101
102     # Sends the goal to the action server.
103         client.send_goal(arucoNav.goal)
104         print("Yendo hacia la marca")
105
106     # Waits for the server to finish performing the action.
107         wait = client.wait_for_result()
108
109     # If the result doesn't arrive, assume the Server is not available
110         if not wait:
111             rospy.logerr("Action server not available!")
112             rospy.signal_shutdown("Action server not available!")
113         else:
114         # Result of executing the action
115             print("Estoy frente a la marca")
116             return client.get_result()
117
118
119
120 # Averiguamos la posicion del robot
121 def arucoCallback(aruco_pose_message):
122     arucoNav.mark_x=aruco_pose_message.pose.position.x
123     arucoNav.mark_y=aruco_pose_message.pose.position.y
124     arucoNav.mark_z=aruco_pose_message.pose.position.z
125     rotation=aruco_pose_message.pose.orientation
```

```
126     angles=tf.transformations.euler_from_quaternion(quaternion=(rotation.x, rotation.y, rotation.z, rotation.w))
127     arucoNav.mark_alpha=numpy.rad2deg(angles[0])
128     arucoNav.mark_beta=numpy.rad2deg(angles[1])
129     arucoNav.mark_theta=numpy.rad2deg(angles[2])
130 
131     # Realizamos un filtrado del angulo theta, para ello guardamos 10 valores y sacamos la mediana
132     if len(arucoNav.mark_theta_list)>9:
133         arucoNav.mark_theta_list.remove(arucoNav.mark_theta_list[0])
134     if not len(arucoNav.mark_theta_list)==0:
135         arucoNav.mark_theta_median=numpy.median(arucoNav.mark_theta_list)
136     else:
137         arucoNav.mark_theta_median=arucoNav.mark_theta
138 
139 
140 
141 def listener():
142     rospy.Subscriber('aruco_pose', PoseStamped, arucoCallback)
143 
144 
145     # If the python node is executed as main process (sourced directly)
146 if __name__ == '__main__':
147     try:
148         rospy.init_node('aruco_navigation')
149         arucoNav=arucoNavigation()
150         listener()
151         result = movebase_client()
152         if not result:
153             rospy.spin()
154         if rospy.is_shutdown():
155             rospy.loginfo("Se ha cancelado el programa")
156         elif result:
157             rospy.loginfo("Programa finalizado")
158     except rospy.ROSInterruptException:
159         rospy.loginfo("Navigation test finished.")
```

A.4. arucoCmd.py

Este programa se encarga de colocar al robot frente a la marca en la posición que le indiquemos. En nuestro caso esta configurado para que se sitúe con ángulo entre -2 y 2 grados, a una distancia X de 1 metro y con una distancia respecto a Y entre -0.05 y 0.05 metros. Para la primera fase de la navegación hacemos que el robot se acerque a 1,3 metros, de manera que no este muy cerca como para colisionar con la pared (si se diese el caso de tener estar en un ángulo grande respecto a la marca). Pero a la vez tener una muy buena estimación de la posición de la marca como vimos en el experimento de la sección 5.3. En la fase 2 para el movimiento circular elegimos 0,5 rad/s como velocidad angular y como velocidad lineal el radio de la circunferencia (1,3 m) multiplicado por la velocidad angular y un por un factor de compensación (tras varias pruebas se le ha asignado el valor de 0,5275) que hemos añadido para asegurarnos que el movimiento realmente es circular. Por pantalla el programa muestra cuando se inicia la navegación, cuando se cumple cada una de las fases, si el programa es cancelado y cuando se ha llegado a la posición deseada.
Para lanzar este programa tenemos que haber lanzado previamente: 'single.launch', 'image_view', 'marker_tf' y 'transform_listener'.

A continuación está el código del programa:

```
#!/usr/bin/env python
"""
    arucoCmd.py - Version 1.0 5-9-2020
    Autor: David Barrera
"""

import rospy
import math
import time
import numpy
import tf


from geometry_msgs.msg import Quaternion, PoseWithCovarianceStamped, PoseStamped
from tf.transformations import quaternion_from_euler

from geometry_msgs.msg import Twist


class arucoNavigation():
    def __init__(self):
    # Para la velocidad del robot
    self.vel = Twist()
    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

    # Para la posicion de la marca
    self.mark_x=0.0
    self.mark_y=0.0
    self.mark_z=0.0
    self.mark_alpha=0.0
    self.mark_beta=0.0
```

```
38      self.mark_theta=0.0
39
40      self.mark_theta_list=[]
41      self.mark_theta_median=0.0
42
43      # Para saber si esta bien posicionado
44      self.fase1=0
45      self.fase2=0
46      self.fase3=0
47
48      # Para saber si el robot ha llegado a la posicion objetivo
49      self.fin=0
50
51
52  def navigation():
53
54    # Fase 1: El robot gira hasta estar centrado a la marca y avanza hacia ella
55
56    #Orientacion
57    if (arucoNav.fase1==0):
58      if (arucoNav.mark_y<-0.05):
59        arucoNav.vel.linear.x=0.0
60        arucoNav.vel.angular.z=-0.3
61      elif (arucoNav.mark_y>0.05):
62        arucoNav.vel.linear.x=0.0
63        arucoNav.vel.angular.z=0.3
64      #Acercamiento
65      elif (arucoNav.mark_x>1.3):
66        arucoNav.vel.linear.x=0.2
67        arucoNav.vel.angular.z=0.0
68      elif (arucoNav.mark_x<=1.3):
69        arucoNav.vel.linear.x=0.0
70        arucoNav.fase1=1
71        print("Fase 1 completada")
72
73
74
75    # Fase 2: El robot hace un movimiento circular hasta estar perpendicular a la marca
76    if (arucoNav.fase1==1 and arucoNav.fase2==0):
77      if (arucoNav.mark_theta_median<-2):
78        arucoNav.vel.angular.z = -0.5
79        arucoNav.vel.linear.x = 0.0
80        arucoNav.vel.linear.y = -1.3*arucoNav.vel.angular.z*0.5275
81      elif (arucoNav.mark_theta_median>2):
82        arucoNav.vel.angular.z = 0.5
83        arucoNav.vel.linear.x = 0.0
84        arucoNav.vel.linear.y = -1.3*arucoNav.vel.angular.z*0.5275
85      elif (arucoNav.mark_theta_median>-2 and arucoNav.mark_theta_median<2):
86        arucoNav.vel.angular.z = 0.0
87        arucoNav.vel.linear.x = 0.0
88        arucoNav.vel.linear.y = 0.0
89        arucoNav.fase2=1
90        print("Fase 2 completada")
91
92    # Fase 3: El robot se acerca a la marca hasta estar justo enfrente de ella
93    if (arucoNav.fase1==1 and arucoNav.fase2==1):
94      if (arucoNav.mark_x>1.0):       #Si esta lejos avanza en x
95        arucoNav.vel.linear.x=0.15
96        arucoNav.vel.linear.y=0.0
97        if (arucoNav.mark_y>0.05):      #Si no esta centrado se desplaza lateralmente
98          arucoNav.vel.linear.y=0.1
99        elif (arucoNav.mark_y<-0.05):
100         arucoNav.vel.linear.y=-0.1
101     elif (arucoNav.mark_x<1.0):       #Si esta cerca se detiene en x
102       arucoNav.vel.linear.x=0.0
103       arucoNav.vel.linear.y=0.0
104       if (arucoNav.mark_y>0.05):      #Si no esta centrado se desplaza lateralmente
105         arucoNav.vel.linear.y=0.1
106       elif (arucoNav.mark_y<-0.05):
107         arucoNav.vel.linear.y=-0.1
108       else:               #Si esta centrado y cerca, esta en la posicion deseada
109         arucoNav.fase3=1
110         print("Fase 3 completada")
111
112
113
114   # Si el robot esta bien posicionado en x e y y ademas esta orientado en z, ha cumplido el objetivo
115   if (arucoNav.fase1 and arucoNav.fase2 and arucoNav.fase3):
116     arucoNav.vel.angular.z = 0.0
```

```
117     arucoNav.vel.linear.x = 0.0
118     arucoNav.vel.linear.y = 0.0
119     arucoNav.fin=1
120     print("Fin de la navegacion")
121   pub.publish(arucoNav.vel)
122 
123 
124 # Averiguamos la posicion del robot respecto a la marca
125 def arucoCallback(aruco_pose_message):
126     arucoNav.mark_x=aruco_pose_message.pose.position.x
127     arucoNav.mark_y=aruco_pose_message.pose.position.y
128     arucoNav.mark_z=aruco_pose_message.pose.position.z
129     rotation=aruco_pose_message.pose.orientation
130     angles=tf.transformations.euler_from_quaternion(quaternion=(rotation.x, rotation.y, rotation.z, rotation.w))
131     arucoNav.mark_alpha=numpy.rad2deg(angles[0])
132     arucoNav.mark_beta=numpy.rad2deg(angles[1])
133     arucoNav.mark_theta=numpy.rad2deg(angles[2])
134 
135     # Realizamos un filtrado del angulo theta, para ello guardamos 10 valores y sacamos la mediana
136     if len(arucoNav.mark_theta_list)>9:
137         arucoNav.mark_theta_list.remove(arucoNav.mark_theta_list[0])
138     if not len(arucoNav.mark_theta_list)==0:
139         arucoNav.mark_theta_median=numpy.median(arucoNav.mark_theta_list)
140     else:
141         arucoNav.mark_theta_median=arucoNav.mark_theta
142 
143 
144 def listener():
145     rospy.Subscriber('aruco_pose', PoseStamped, arucoCallback)
146 
147 
148     # If the python node is executed as main process (sourced directly)
149 if __name__ == '__main__':
150   try:
151     rospy.init_node('aruco_navigation')
152     pub = rospy.Publisher('/campero/cmd_vel',Twist,queue_size=10)
153     arucoNav=arucoNavigation()
154     listener()
155     rospy.sleep(1)
156     print("Inicio de la navegacion")
157     while not arucoNav.fin and not rospy.is_shutdown():
158       navigation()
159     #if rospy.is_shutdown():
160     if rospy.is_shutdown():
161       rospy.loginfo("Se ha cancelado el programa")
162     else:
163       rospy.loginfo("El robot ha llegado a su destino")
164   except rospy.ROSInterruptException:
165     rospy.loginfo("Navigation test finished.")
```

A.5. multimarca_loop.py

Este es el único programa que se reproduce en bucle, está diseñado para usarse en el entorno de Gazebo 'multimarca' (ver figuras B.18 y B.19). El programa arranca con el robot visualizando una marca. El robot se acerca a ella, una vez está a la distancia que hemos configurado gira 90 grados y se dirige hacia la siguiente marca. Los programas que tienen que ser lanzados previamente son los mismos que se lanzan en' arucoCmd.py'.

```
1 #!/usr/bin/env python
2 """ arucoNav.py - Version 1.0 17-9-2020
3     Autor: David Barrera
4     Esta version realiza el giro con rospy.Timer()
5 """
6 
7 import rospy
8 import math
9 import time
10 import numpy
```

```
11 import tf
12
13
14 from geometry_msgs.msg import Quaternion, PoseWithCovarianceStamped, PoseStamped
15 from tf.transformations import quaternion_from_euler
16
17 from geometry_msgs.msg import Twist
18
19
20 class arucoNavigation():
21     def __init__(self):
22     # Para la velocidad del robot
23     self.vel = Twist()
24     self.vel.linear.x=0.0
25     self.vel.linear.y=0.0
26     self.vel.linear.z=0.0
27
28     self.vel.angular.x=0.0
29     self.vel.angular.y=0.0
30     self.vel.angular.z=0.0
31
32     # Para la posicion de la marca
33     self.mark_x=0.0
34     self.mark_y=0.0
35     self.mark_z=0.0
36     self.mark_alpha=0.0
37     self.mark_beta=0.0
38     self.mark_theta=0.0
39
40     self.mark_theta_list=[]
41     self.mark_theta_median=0.0
42
43     # Para saber si esta bien posicionado
44     self.orientado=0
45     self.cerca=0
46     self.centrado=0
47
48     # Para saber si el robot ha llegado a la posicion objetivo
49     self.fin=0
50     self.girocompleto=0
51
52
53 def navigation():
54
55   if not rospy.is_shutdown():
56     # Avance del robot
57     if (arucoNav.mark_x<0.01):  # Si ya esta lo suficientemente cerca no avanza
58       arucoNav.vel.linear.x=0.0
59       arucoNav.cerca=1
60     #Fase de avance rapido
61     elif (arucoNav.mark_x>0.3):
62       arucoNav.vel.linear.x=0.3
63       arucoNav.vel.angular.z=0.0
64     #Fase de avance lento
65     elif (arucoNav.mark_x>=0.1):
66       arucoNav.vel.linear.x=0.15
67       arucoNav.vel.angular.z=0.0
68
69
70     # Desplazamiento lateral del robot
71
72     if (arucoNav.mark_y<0.1) and (arucoNav.mark_y>-0.1): # Si esta centrado en la marca no se desplaza
73       arucoNav.vel.linear.y=0.0
74       arucoNav.centrado=1
75
76     # Fase de esplazamiento rapido
77     elif (arucoNav.mark_y>0.3):
78       arucoNav.vel.linear.y=0.3
79       arucoNav.vel.angular.z=0.0
80     elif (arucoNav.mark_y<-0.3):
81       arucoNav.vel.linear.y=-0.3
82       arucoNav.vel.angular.z=0.0
83
84     #Fase de desplazamiento lento
85     elif (arucoNav.mark_y>0.05):
86       arucoNav.vel.linear.y=0.1
87       arucoNav.vel.angular.z=0.0
88     elif (arucoNav.mark_y<-0.05):
89       arucoNav.vel.linear.y=-0.1
```

```
90        arucoNav.vel.angular.z=0.0
91
92
93      # Rotacion del robot
94
95      if((arucoNav.mark_theta_median<2.0)and(arucoNav.mark_theta_median>-2.0)): # Si el robot esta bien orientado no
96        arucoNav.vel.angular.z=0.0
97        arucoNav.orientado=1
98      #Fase de giro rapido
99      elif (arucoNav.mark_theta_median>10.0):
100       arucoNav.vel.linear.x=0.0
101       arucoNav.vel.linear.y=0.0
102       arucoNav.vel.angular.z=0.3
103     elif(arucoNav.mark_theta_median<-10.0):
104       arucoNav.vel.linear.x=0.0
105       arucoNav.vel.linear.y=0.0
106       arucoNav.vel.angular.z=-0.3
107     # Fase de giro lento
108     elif (arucoNav.mark_theta_median>2.0) and arucoNav.mark_x<0.5:
109       arucoNav.vel.linear.x=0.0
110       arucoNav.vel.linear.y=0.0
111       arucoNav.vel.angular.z=0.15
112     elif(arucoNav.mark_theta_median<-2.0) and arucoNav.mark_x<0.5:
113       arucoNav.vel.linear.x=0.0
114       arucoNav.vel.linear.y=0.0
115       arucoNav.vel.angular.z=-0.15
116
117     # Si el robot esta bien posicionado en x e y y ademas esta orientado en z, ha cumplido el objetivo
118     if (arucoNav.orientado and arucoNav.centrado and arucoNav.cerca):
119       arucoNav.fin=1
120     pub.publish(arucoNav.vel)
121
122
123
124 def giro():
125   print("Empiezo a girar durante 7.5 segundos")
126
127   rospy.Timer(rospy.Duration(7.5),my_callback,oneshot=True)
128
129   while (not arucoNav.girocompleto==1) and not rospy.is_shutdown():
130     arucoNav.vel.angular.z=-0.5
131     pub.publish(arucoNav.vel)
132
133
134   print("Giro terminado")
135   arucoNav.girocompleto=0
136   arucoNav.vel.angular.z=0.0
137   pub.publish(arucoNav.vel)
138
139
140 def my_callback(event):
141   print("Han pasado 7.5 segundos")
142   arucoNav.girocompleto=1
143
144
145 # Averiguamos la posicion del robot respecto a la marca
146 def arucoCallback(aruco_pose_message):
147     arucoNav.mark_x=aruco_pose_message.pose.position.x
148     arucoNav.mark_y=aruco_pose_message.pose.position.y
149     arucoNav.mark_z=aruco_pose_message.pose.position.z
150     rotation=aruco_pose_message.pose.orientation
151     angles=tf.transformations.euler_from_quaternion(quaternion=(rotation.x, rotation.y, rotation.z, rotation.w))
152     arucoNav.mark_alpha=numpy.rad2deg(angles[0])
153     arucoNav.mark_beta=numpy.rad2deg(angles[1])
154     arucoNav.mark_theta=numpy.rad2deg(angles[2])
155
156     # Realizamos un filtrado del angulo theta, para ello guardamos 10 valores y sacamos la mediana
157     if len(arucoNav.mark_theta_list)>9:
158         arucoNav.mark_theta_list.remove(arucoNav.mark_theta_list[0])
159     if not len(arucoNav.mark_theta_list)==0:
160         arucoNav.mark_theta_median=numpy.median(arucoNav.mark_theta_list)
161     else:
162         arucoNav.mark_theta_median=arucoNav.mark_theta
163
164     # Si queremos ver la localizacion de la marca en todo momento descomentar esta parte
165     #rospy.loginfo('La marca se encuentra la posicion: x= %f , y= %f ,z= %f con la orientacion: alpha= %f , beta= %
166     #rospy.loginfo('x= %f ', arucoNav.mark_x)
167     #rospy.loginfo('y= %f ', arucoNav.mark_y)
168     #rospy.loginfo('z= %f ', arucoNav.mark_z)
```

```
169     #rospy.loginfo('alpha= %f ', arucoNav.mark_alpha)
170     #rospy.loginfo('beta= %f ',arucoNav. mark_beta)
171     #rospy.loginfo('theta= %f ', arucoNav.mark_theta)
172     #rospy.loginfo('theta_median= %f ', arucoNav.mark_theta_median)
173
174
175 def listener():
176     rospy.Subscriber('aruco_pose', PoseStamped, arucoCallback)
177
178
179     # If the python node is executed as main process (sourced directly)
180 if __name__ == '__main__':
181   try:
182     rospy.init_node('aruco_navigation')
183     pub = rospy.Publisher('/campero/cmd_vel',Twist,queue_size=10)
184     arucoNav=arucoNavigation()
185     listener()
186     rospy.sleep(1)
187     while not rospy.is_shutdown():
188       print("Voy hacia la marca")
189
190       arucoNav.orientado=0
191       arucoNav.cerca=0
192       arucoNav.centrado=0
193       arucoNav.fin=0
194       while not arucoNav.fin and not rospy.is_shutdown():
195         navigation()
196       print("Acercamiento a la marca completado")
197       giro()
198
199     #if rospy.is_shutdown():
200     if rospy.is_shutdown():
201       rospy.loginfo("Se ha cancelado el programa")
202   except rospy.ROSInterruptException:
203     rospy.loginfo("Navigation test finished.")
```

Anexos B

Cómo crear marcas ArUco para utilizarlas en el entorno Gazebo

En este anexo vamos a explicar de manera muy detallada como crear el modelo 3D de marcas visuales utilizando el programa Blender v2.82a y como usarlas en el entorno Gazebo. En nuestro caso serán marcas ArUco, pero se puede utilizar cualquier otro tipo de marca.

B.1. Creación de marcas visuales con Blender v2.82a en formato COLLADA

Primero abrimos Blender y creamos un nuevo archivo, nos aparecerá una escena con un cubo en el centro, este cubo lo vamos a transformar en nuestra marca. Por defecto el cubo debería medir 2x2x2 m, de todas maneras vamos a comprobarlo con la herramienta 'Measure' (1). Una vez sabemos la medida el cubo, en 'Object Properties' (2), cambiamos los valores de 'Scale' (3) en cada eje hasta tener un objeto de dimensiones 0,01x0,12x0,12 m, como en la figura B.2.

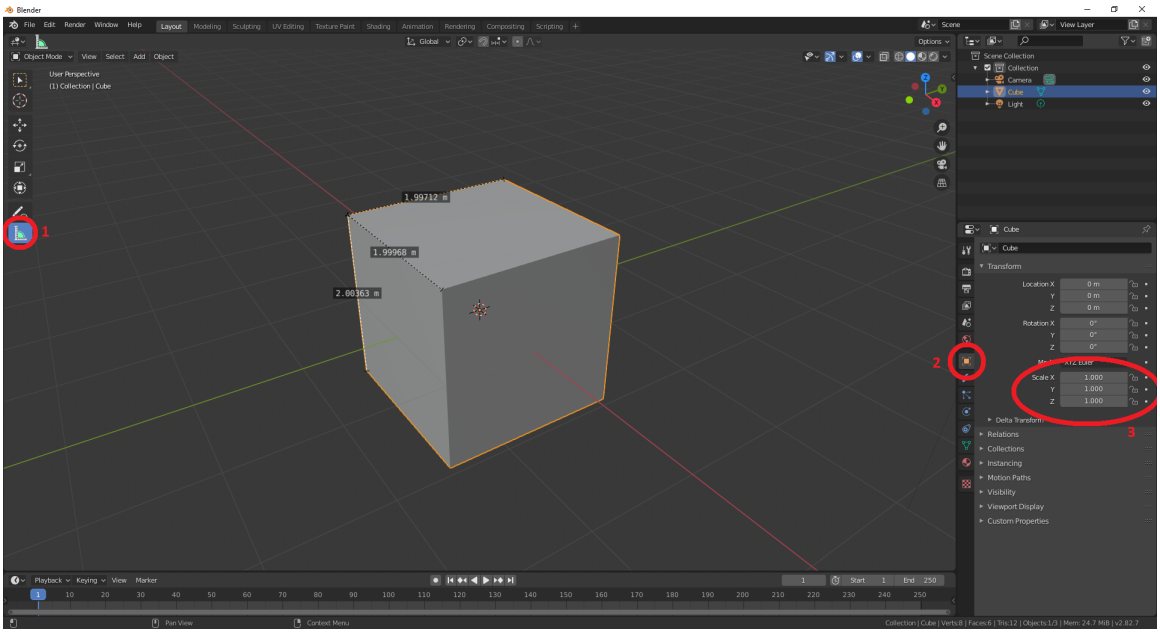

Figura B.1: Medidas del cubo que aparece por defecto al crear un archivo

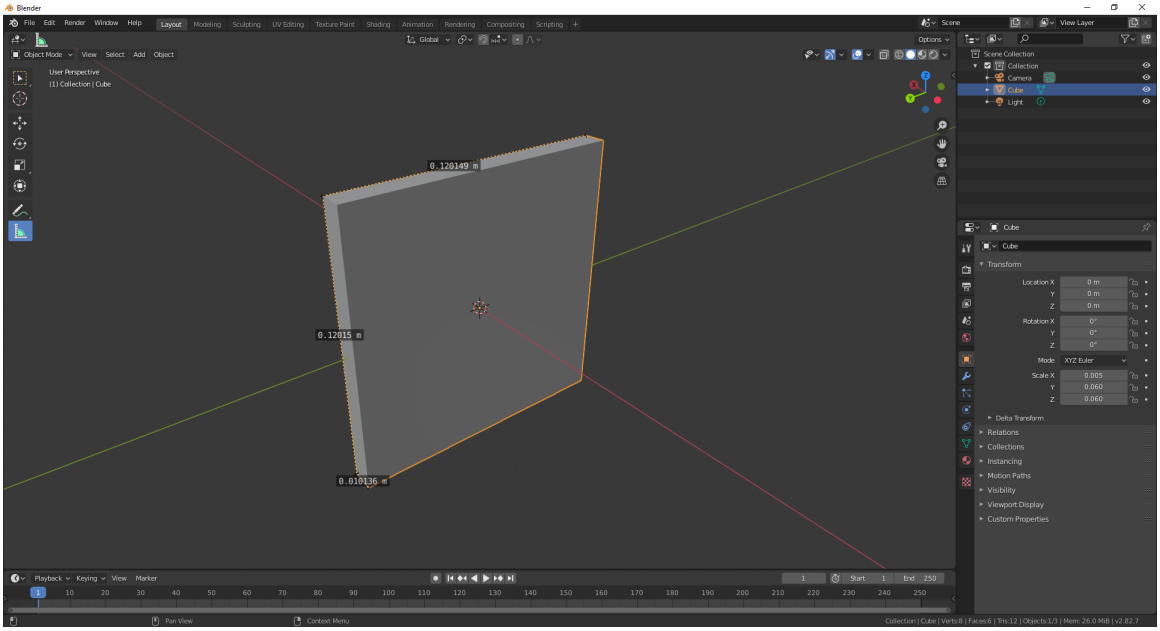

Figura B.2: Cambio en las dimensiones del cubo

Una vez tenemos la marca dimensionada tenemos que agregar como textura la imagen de la marca. Para ello vamos a 'Material Properties' (4), seleccionamos el icono circular a la derecha de 'Base Color' (5), se abrirá una pestaña en la que seleccionamos 'Image Texture', justo debajo seleccionamos el archivo en 'Open' (6), aquí simplemente elegimos la imagen de la marca que queremos crear, una vez elegida vemos que nada ha cambiado. Tenemos que ir a la parte de arriba y pulsar el icono (7) para que se muestre el material del objeto.

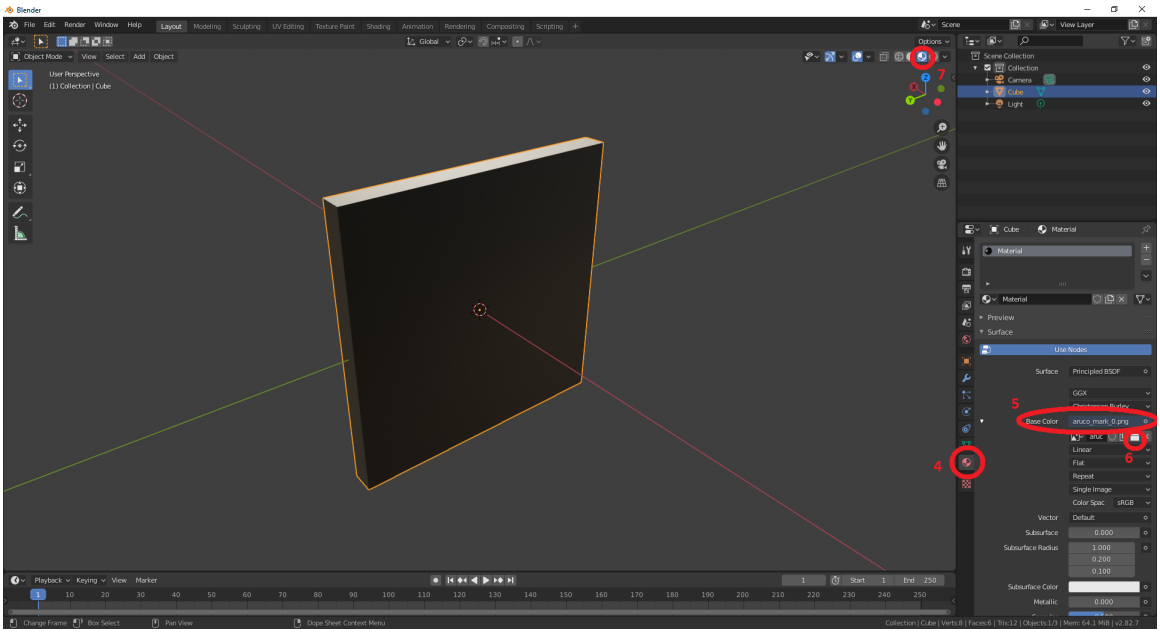

Figura B.3: Agregando una textura a la marca

Ya tenemos el modelo 3D y la textura agregada, pero la textura no se visualiza correctamente. Así que, seleccionamos el modelo 3D de la marca con un solo click, pulsamos la tecla 'Tab', seguido de la tecla 'U', a continuación seleccionamos 'Unwrap' (figura B.4), colocamos el cursor en la esquina superior izquierda de la pantalla al lado izquierdo del botón 'Editor Type' (8), mantenemos pulsado el botón izquierdo del ratón y desplazamos hacia la derecha, de manera que veremos como se va abriendo una nueva ventana. En esta nueva ventana pulsamos en 'Editor Type' (8) y seleccionamos 'UV Editor', en la barra de 'UV Editor' pulsamos en el icono que representa la imagen (9) y elegimos nuestra imagen. Ahora veremos la imagen en pantalla, para hacer *zoom* usamos la ruleta del ratón, alejamos un poco la imagen y pulsamos en 'UV Sync Selection' (10).

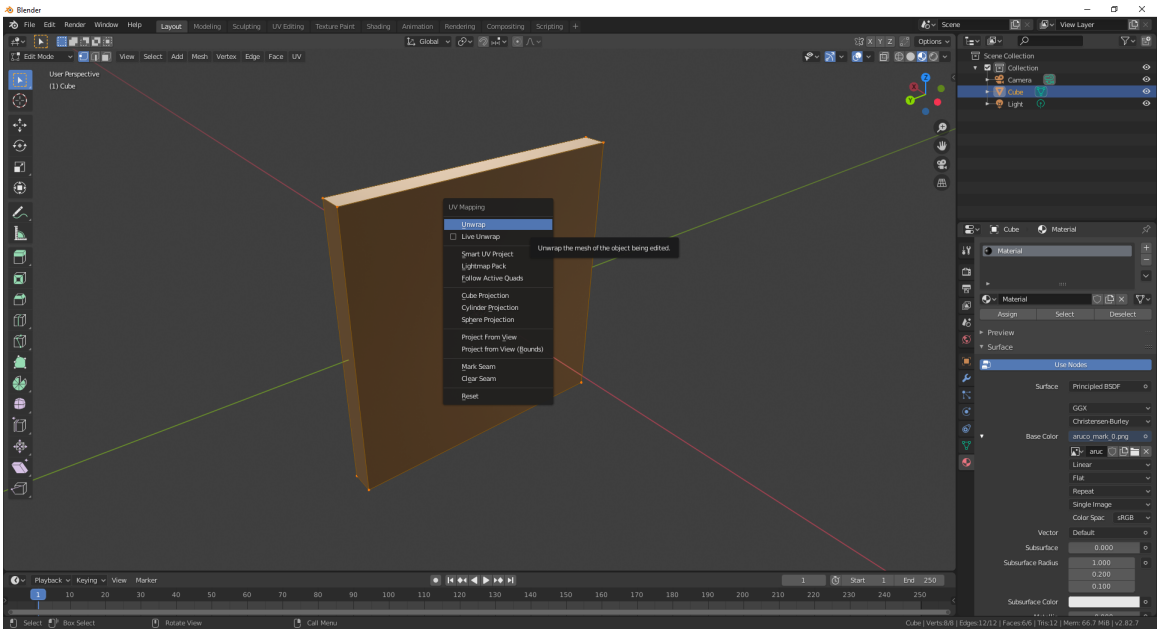

Figura B.4: Seleccionando el botón 'Unwrap'

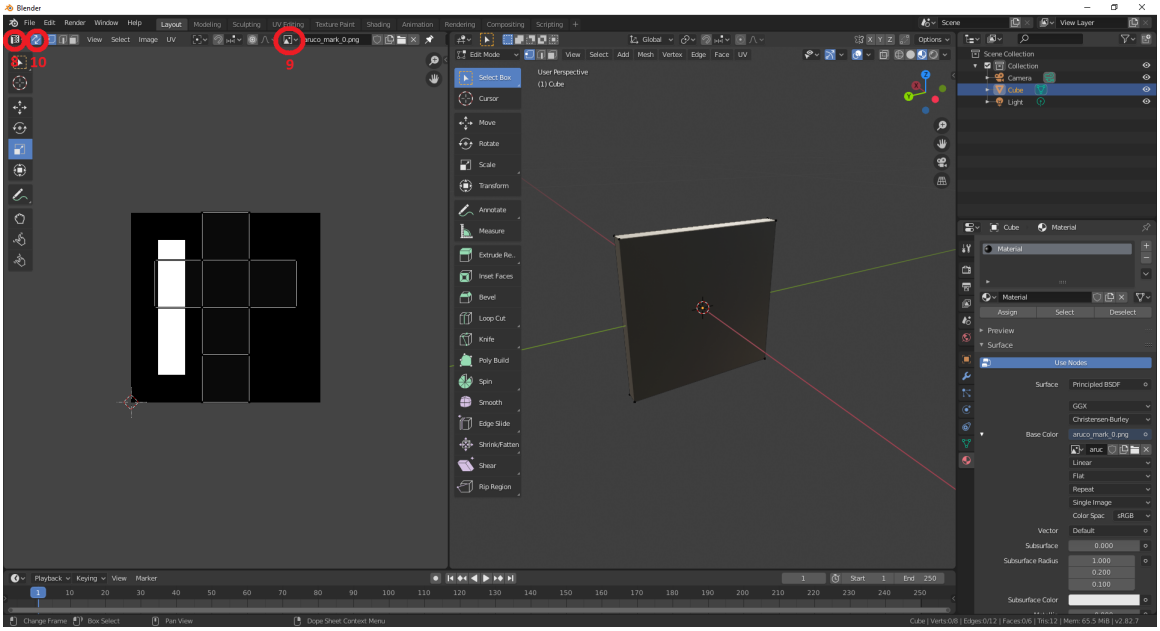

Figura B.5: Abriendo el 'UV Editor' y cargando la imagen

Observamos como sobre nuestra imagen tenemos seis cuadrados, cada cuadrado representa una cara de nuestro objeto. Lo que haremos será que la imagen de la marca se muestre en la cara frontal, para ello, seleccionamos 'Face select' (11), a la izquierda (12) vemos que tenemos varias herramientas, las que nos interesan son: 'Select Box' para elegir las caras que vamos a manipular, 'Move' para mover las caras seleccionadas, 'Rotate' para rotar la imagen en las caras de manera que la marca esté con la orientación correcta y 'Scale' para aumentar o disminuir la imagen dentro de las caras. Esta parte es

más sencilla e intuitiva, simplemente con 'Face select' buscamos la cara frontal (cuando seleccionamos una cara, en la ventana de la derecha, se ilumina la cara que estamos seleccionando), separamos esta cara de las demás (una vez está seleccionada, usamos la herramienta 'Move'). Una vez separadas seleccionamos las caras que no nos interesan y las movemos fuera de la imagen de la marca (como antes seleccionamos las caras con 'Face select' y luego las desplazamos con 'Move'). Ahora se trata de cuadrar la cara frontal con la imagen de la marca. Para ello seleccionamos la cara y la desplazamos al centro de la imagen, una vez centrada usamos 'Scale' para que la cara abarque la imagen completa, si no cuadra del todo podemos usar 'Move' de nuevo hasta que cuadren perfectamente. Si vemos que en la imagen de la derecha la marca no está orientada correctamente, simplemente seleccionamos la cara y usamos la herramienta 'Rotate' hasta que la marca tenga la orientación deseada. Con el resto de las caras para asegurarnos que se quedan de color negro vamos a seleccionarlas, y con 'Scale' reduciremos su tamaño y con 'Move' las colocamos en el borde negro de la marca (como en la figura B.6).

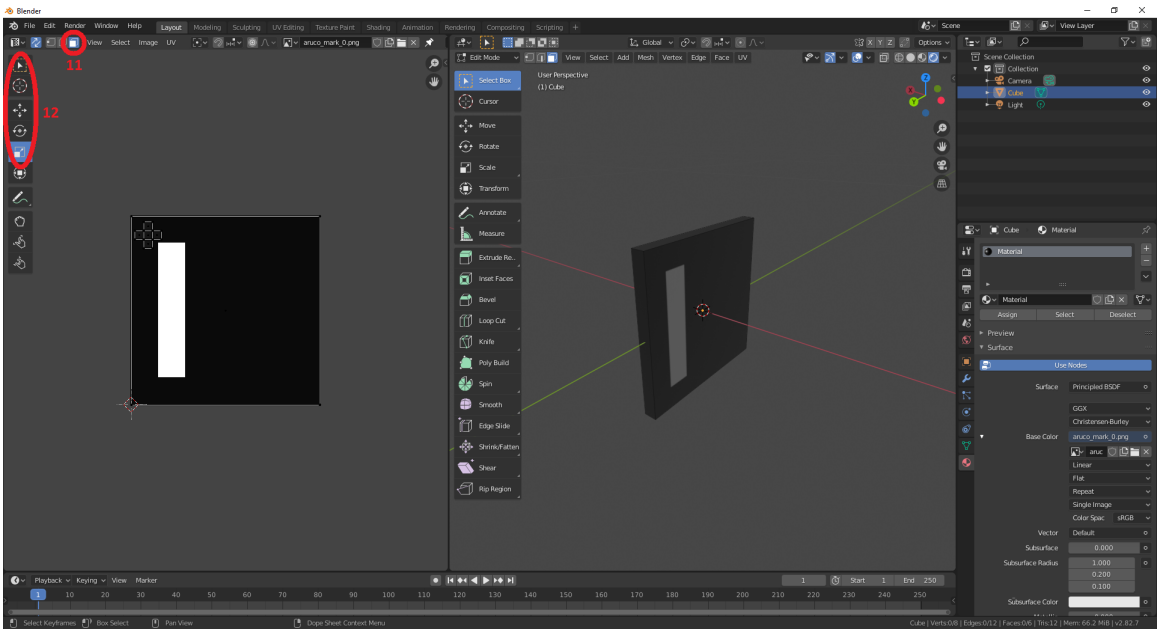

Figura B.6: Aspecto final de la marca

Ahora que ya tenemos nuestra marca ArUco, seleccionamos todas las caras y vamos a la barra superior y seleccionamos 'File', 'Export', 'Collada (Default) (.dae)' y en opciones vamos al desplegable en el que aparece 'Operator Presets' y seleccionamos 'SI+Open Sim Static', nombramos el archivo, elegimos donde guardarlo y lo exportamos.

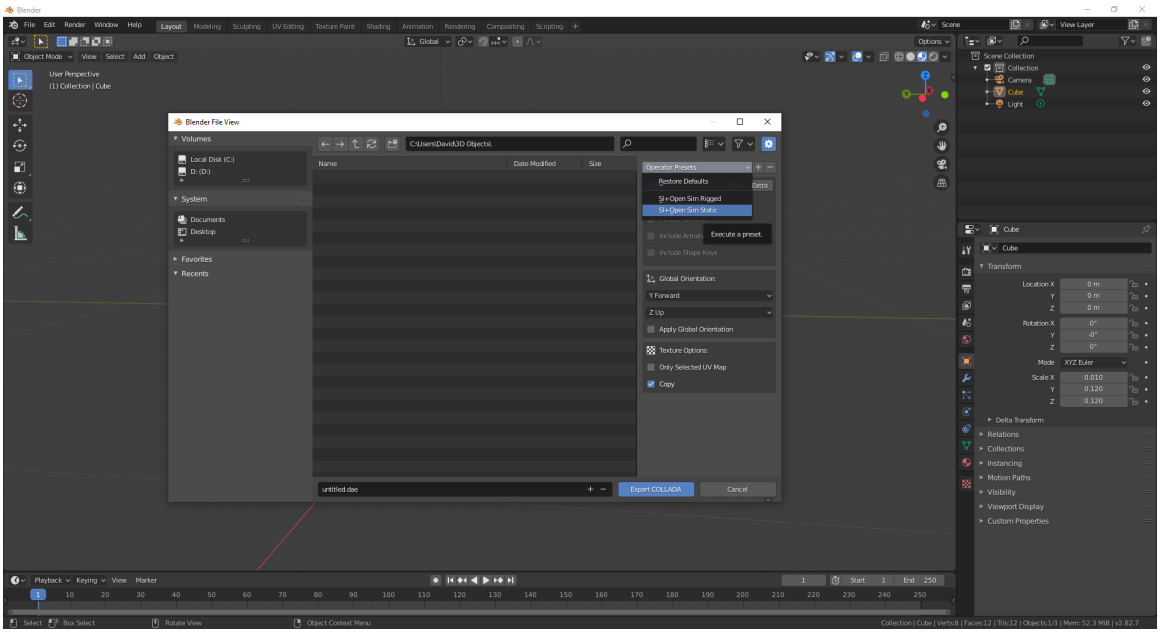

Figura B.7: Exportando nuestra marca

B.2. Cómo importar nuestras marcas visuales en formato COLLADA a Gazebo

En la sección anterior hemos explicado como crear nuestras propias marcas visuales. Una vez que ya las tenemos en formato de archivo COLLADA, estamos listos para usarlas en Gazebo.

Para crear una marca necesitamos tres archivos: El modelo 3D de la marca que hemos creado en Blender, la imagen de la marca y el material de la marca que es el único que aún no tenemos.

Crear el material es muy sencillo, solamente tenemos que abrir nuestro editor de texto y copiar el siguiente código:

```
1  material mi_material
2  {
3      technique
4      {
5          pass
6          {
7              texture_unit
8              {
9                  texture mi_marca.png
10             }
11         }
12     }
13 }
```

Figura B.8: Archivo mi_material.material

En la línea 1 ponemos 'material' seguido del nombre que le queremos dar al material,

en este caso 'mi_material' y en la línea 9 'texture' seguido del nombre de la imagen que vamos a usar como textura, que será la misma que usamos para crear la marca en Blender. Guardamos este archivo con extensión '.material' y listo.

También tendremos que modificar una línea del archivo COLLADA, así que lo abrimos con nuestro editor de texto y buscamos la línea donde aparece la imagen que usamos para crear la marca, debería estar en la línea 44. Veremos que simplemente aparece el nombre del archivo, lo que tenemos que hacer el poner el directorio donde se encuentra, como podemos ver en la figura B.9.

```
34                  <index_of_refraction>
35                    <float sid="ior">1.45</float>
36                  </index_of_refraction>
37                </lambert>
38              </technique>
39            </profile_COMMON>
40          </effect>
41        </library_effects>
42        <library_images>
43          <image id="aruco_mark_6_png" name="aruco_mark_6_png">
44            <init_from>/home/david/campero_ws/src/campero/mi_marca/mi_marca.png</init_from>
45          </image>
46        </library_images>
47        <library_materials>
48          <material id="Material-material" name="Material">
49            <instance_effect url="#Material-effect"/>
50          </material>
51        </library_materials>
52        <library_geometries>
53          <geometry id="Cube-mesh" name="Cube">
```

Figura B.9: Archivo mi_marca.dae

Una vez ya tenemos los archivos listos ya podemos iniciar Gazebo. Lo primero que hay que hacer es abrir el entorno que queremos editar, para ello introducimos el siguiente comando:

$ gazebo mi_directorio/nombre_entorno.world

En caso de querer hacerlo en un entorno vacío en vez del comando anterior escribiremos:

$ gazebo

Ahora tenemos que crear o usar un objeto ya existente que hará de marca. En nuestro caso lo creamos, usaremos el cubo que tiene Gazebo por defecto. Una vez creado, lo seleccionamos con el click izquierdo, una vez seleccionado pulsamos el botón derecho del ratón y en el desplegable que aparece seleccionamos 'Edit model'. En modo de edición hacemos lo mismo que antes, seleccionamos con click izquierdo, pulsamos click derecho y ahora en el desplegable elegimos 'Open Link Inspector'. Se nos abrirá una nueva ventana con tres pestañas, vamos a la pestaña 'Visual' y dentro de esta pestaña al apartado 'Geometry', en el desplegable de 'Geometry' seleccionamos 'mesh', aparecerá una nueva opción para cargar nuestro archivo, así que eso haremos, pulsamos en los tres puntos y cargamos nuestro archivo con extensión '.dae' (ver figura B.10). En la pestaña 'Collision' hacemos exactamente lo mismo. Ahora pulsamos en 'OK', vamos

arriba a 'File', guardamos el nuevo objeto que hemos creado y en 'File' le damos a salir del editor. Bien ahora veremos que tenemos la forma de la marca pero no el material. Vamos a guardar el entorno que hemos creado y cerramos Gazebo.

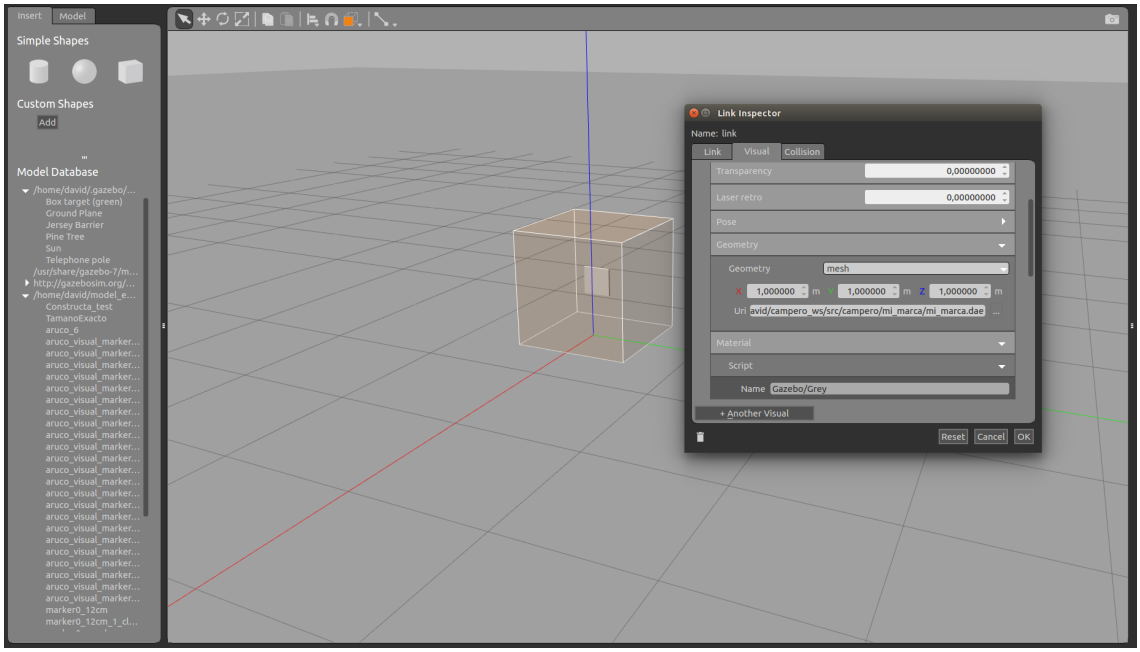

Figura B.10: Seleccionar archivo COLLADA en Gazebo

Con el editor de texto abrimos el entorno de Gazebo que acabamos de guardar, ahora buscamos donde se ha guardado nuestra marca, vamos al apartado '<visual>', dentro de este al apartado '<material>' y dentro de este a '<script>', tenemos que cambiar las dos líneas. En la primera (línea 139 de la figura B.11) tenemos que poner el nombre que tiene nuestro material, no el nombre del archivo, si no el nombre que pusimos en la primera línea del archivo (ver B.8), en nuestro caso 'mi_material'. En la segunda (línea 140 de la figura B.11) tenemos que poner el directorio donde se encuentra el archivo del material que creamos previamente. Una vez hecho esto guardamos los cambios y volvemos a Gazebo.

```
<self_collide>0</self_collide>
<kinematic>0</kinematic>
<gravity>1</gravity>
<visual name='visual'>
  <geometry>
    <mesh>
      <uri>/home/david/campero_ws/src/campero/mi_marca/mi_marca.dae</uri>
      <scale>1 1 1</scale>
    </mesh>
  </geometry>
  <material>
    <script>
      <name>mi_material</name>
      <uri>/home/david/campero_ws/src/campero/mi_marca/mi_material.material</uri>
    </script>
    <ambient>0.3 0.3 0.3 1</ambient>
    <diffuse>0.7 0.7 0.7 1</diffuse>
    <specular>0.01 0.01 0.01 1</specular>
    <emissive>0 0 0 1</emissive>
    <shader type='vertex'>
      <normal_map>__default__</normal_map>
    </shader>
  </material>
  <pose frame=''>0 0 0 0 -0 0</pose>
  <cast_shadows>1</cast_shadows>
```

Figura B.11: Archivo mi_world.world

Cargamos el entorno que hemos creado previamente mediante el comando:

$ gazebo mi_directorio/nombre_entorno.world

Si la marca aún no se ve correctamente, abrimos el modo edición y a continuación el 'Link Inpector' como hicimos antes. En la pestaña 'Visual' vamos al apartado 'Material', ponemos el nombre del material en 'Script' que en nuestro caso es 'mi_material' (ver figura B.12). Con esto ya tenemos nuestra marca, así que la guardamos, salimos del modo edición y comprobamos como se ve correctamente.

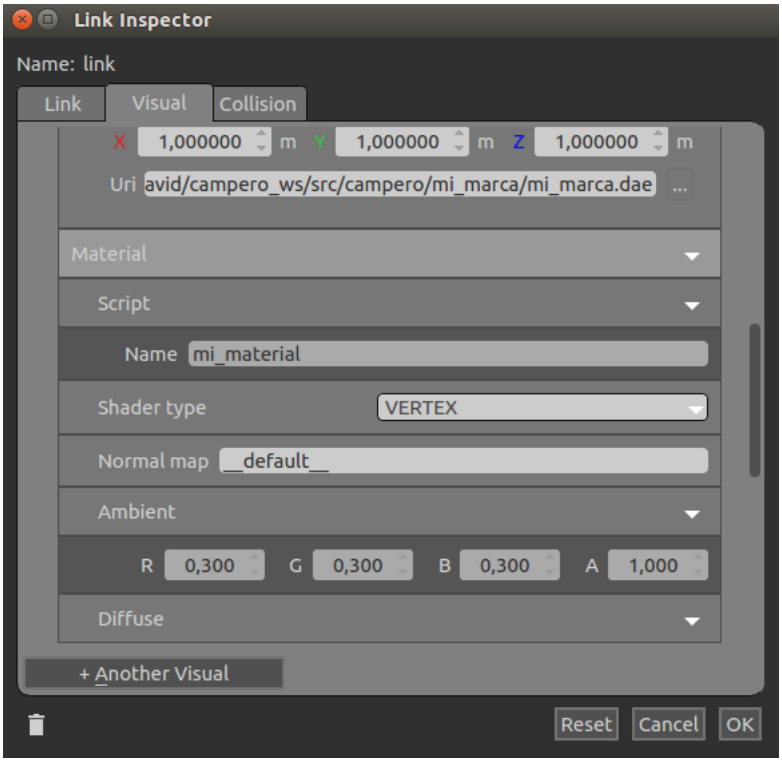

Figura B.12: Apartado 'Material' del 'Link Inspector'

B.3. Entornos creados en Gazebo

Cuando comenzamos este proyecto contábamos con dos entornos de Gazebo ya creados. Un entorno de exterior ('campero_outside.world') y otro de interior ('campero_inside.world'). Dado que los experimentos con el Campero se iban a realizar en interiores, el entorno de exterior casi no lo hemos empleado. Sin embargo, el de interior sí.

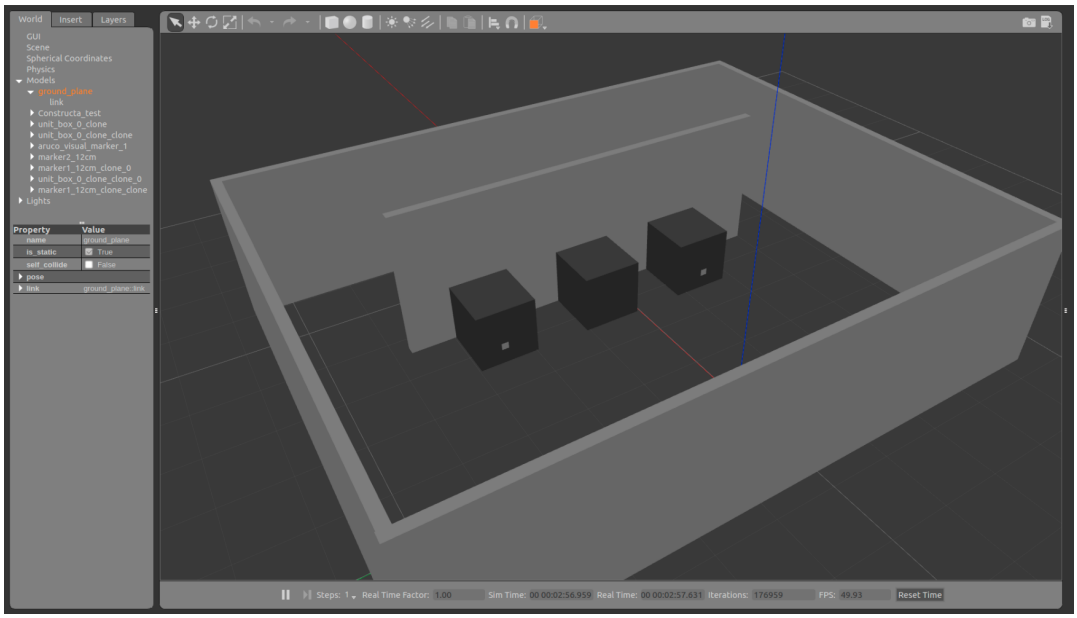

Figura B.13: Entorno 'campero_inside.world'

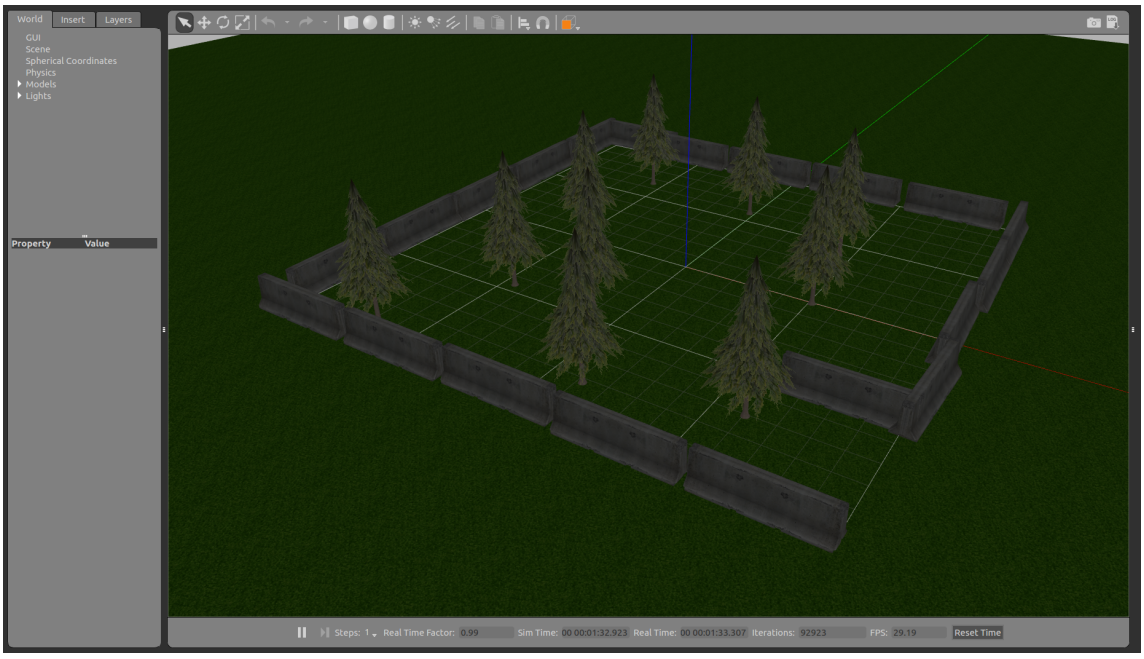

Figura B.14: Entorno 'campero_outside.world'

Conforme avanzaba el proyecto, necesitábamos entornos que se adaptasen a situaciones que queríamos simular. El mapa de interior cuenta con tres marcas ArUco pero todas se encuentran con la misma orientación. Así que decidimos modificarlo para tener marcas con distintas orientaciones. El objetivo era comprobar la robustez de nuestros programas. De esta necesidad surgió el entorno 'aruco':

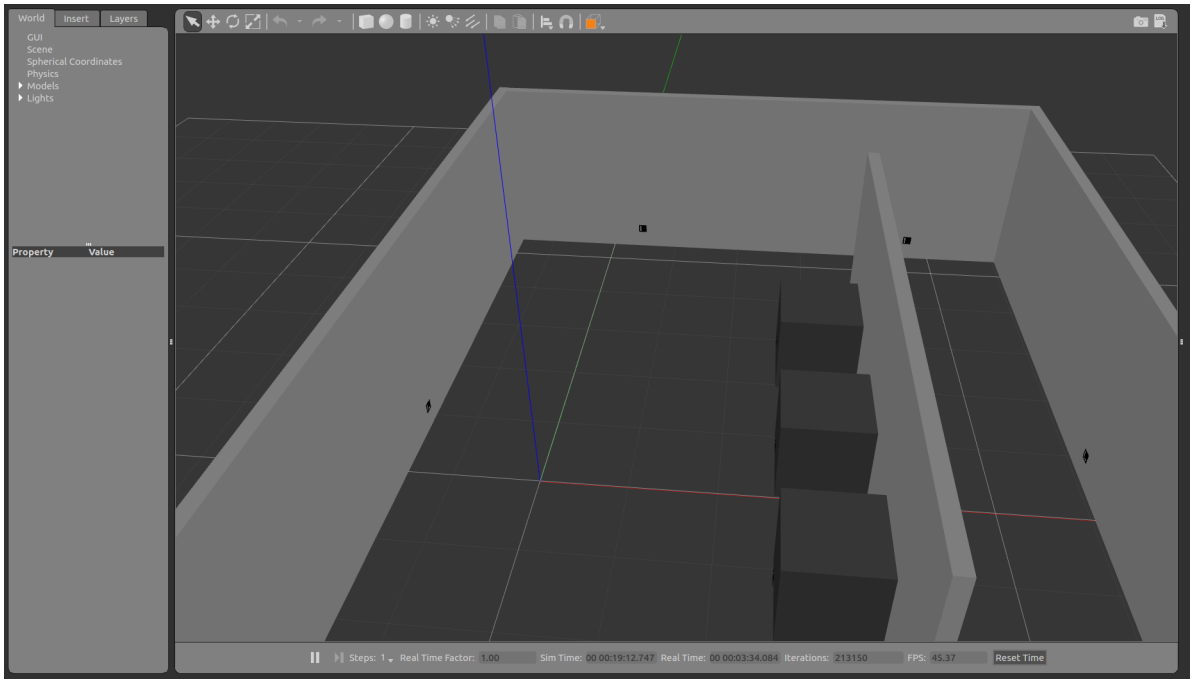

Figura B.15: Entorno 'aruco' primera vista

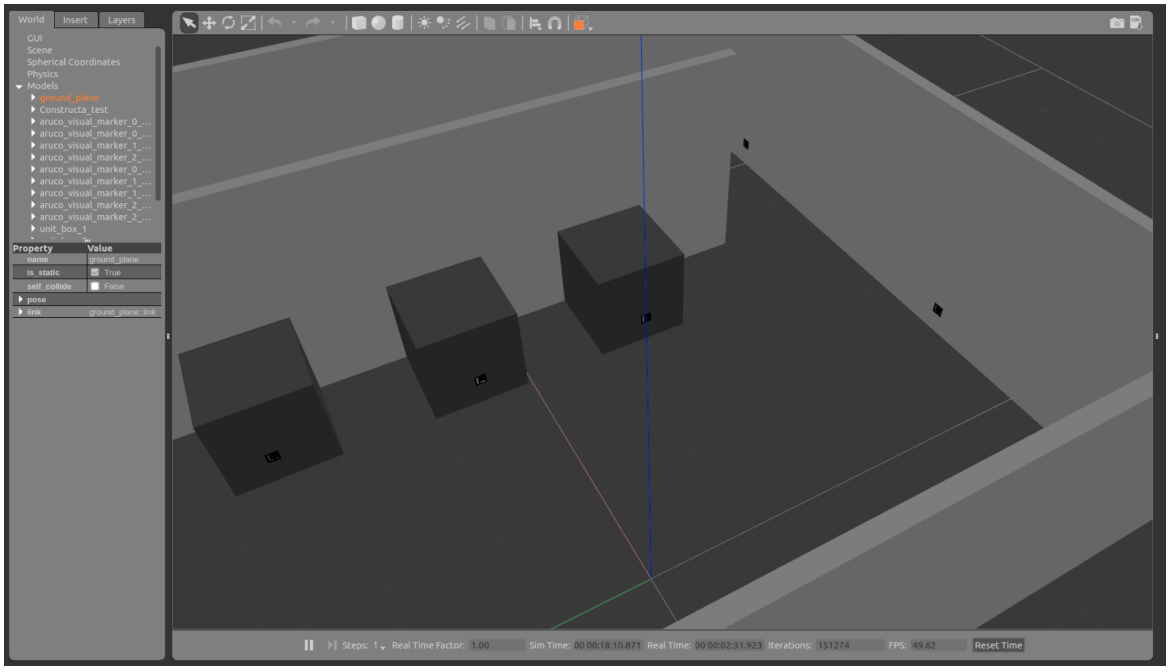

Figura B.16: Entorno 'aruco' segunda vista

Más adelante en el proyecto quisimos realizar una navegación usando varias marcas. Los mapas de interior que teníamos hasta el momento contaban con elementos prescindibles, como los cubos o la pared interior. Así que decidimos crear dos mapas más sencillos sin estos elementos. De ahí surgieron los entornos 'bimarca' y 'multimarca' (usados en el experimento 5.4).

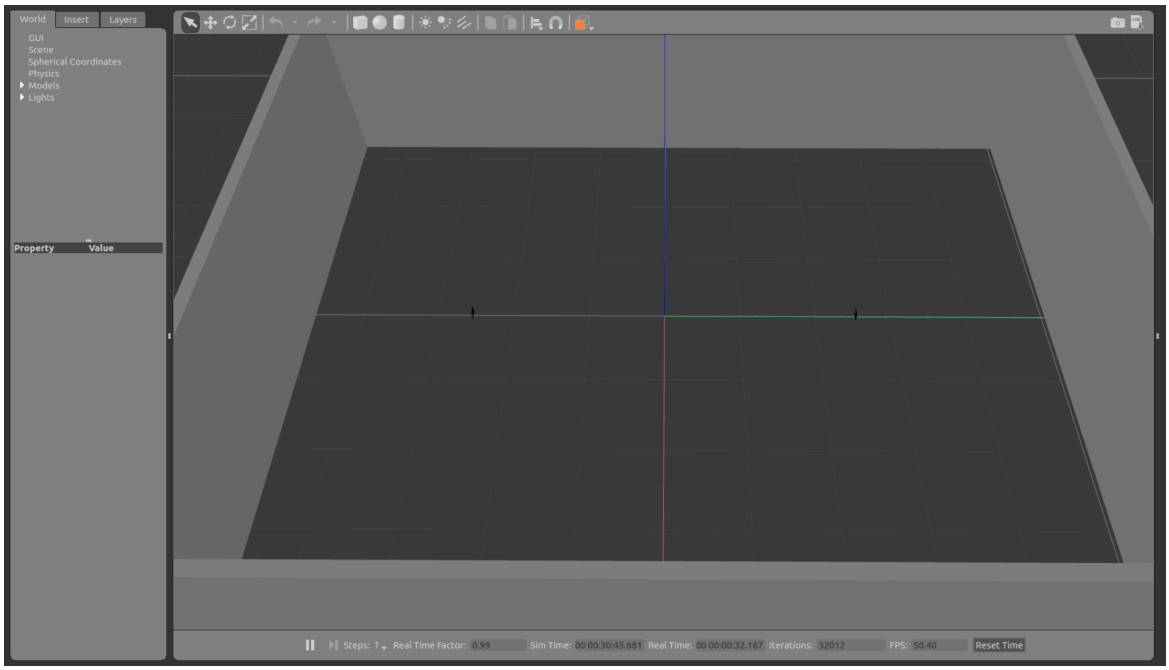

Figura B.17: Entorno 'bimarca'

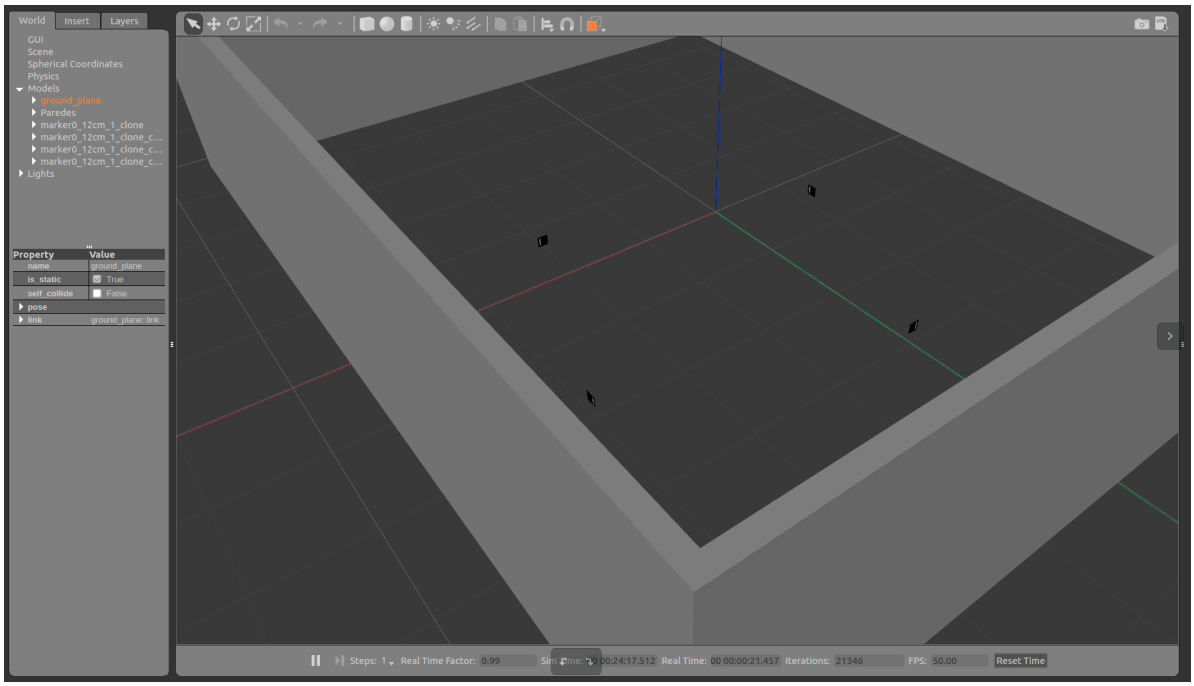

Figura B.18: Entorno 'multimarca' vista general

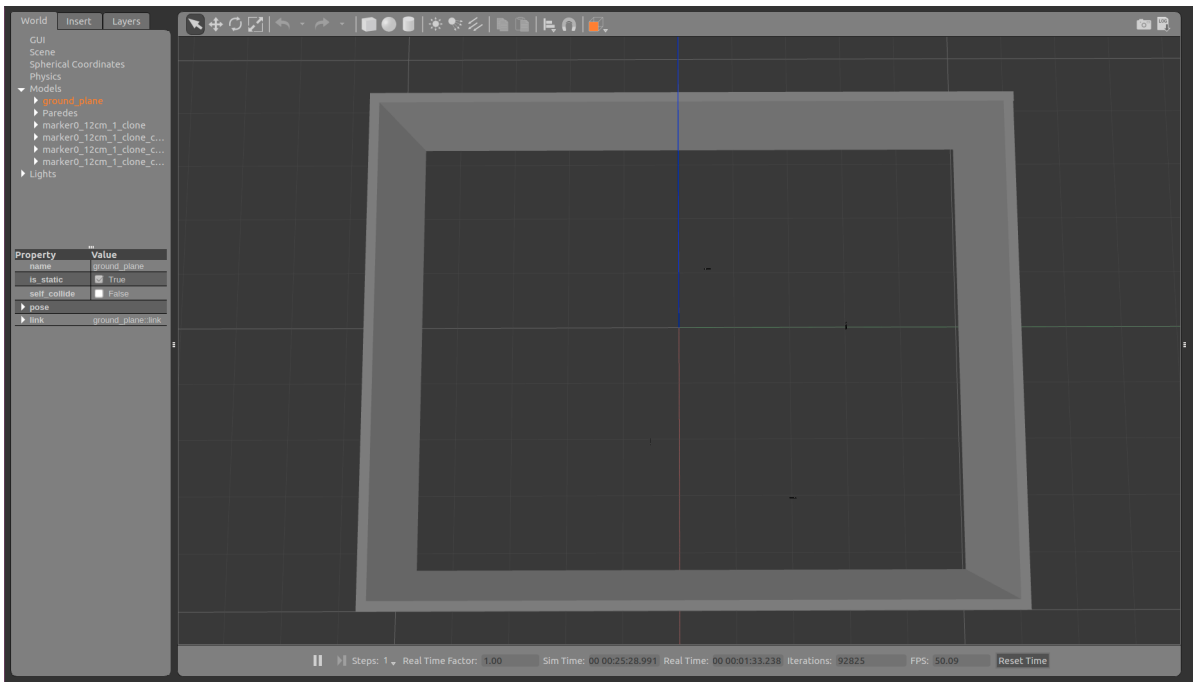

Figura B.19: Entorno 'multimarca' vista superior

Anexos C

Ejecución en ROS de los distintos tipos de navegación

C.1. Mapeado

Antes de empezar con la navegación, necesitamos reconocer el entorno y tener un mapa sobre el que podamos indicar a qué coordenada queremos que se dirija el robot. El método para mapear un entorno de Gazebo en RViz, está detallado en 'Navegación de robots manipuladores en el entorno de ROS' [4]. Pero, con el fin de que este documento sea autocontenido, a continuación explicamos como hacer el mapeado. Lo primero que tenemos que hacer es asegurarnos de que el mapa que se mostrará en RViz es un mapa vacío. Nuestro mapa vacío se encuentra en el directorio:
src/campero/campero_common/campero_localization/maps/empty
En nuestro caso el archivo tiene como nombre:
map_empty.yaml

Una vez hemos comprobado que el mapa vacío se encuentra en la localización adecuada y sabemos que nombre tiene, tenemos que indicar al programa que este es el archivo que debe abrir. Para ello abrimos el archivo *campero_nav.launch* (figura C.1) y en el apartado del mapa de RViz (línea 22) introducimos el nombre de nuestro mapa vacío.

```
1  <?xml version="1.0"?>
2  <launch>
3
4  <!-- Argumentos de entrada -->
5      <!-- Nombre del robot -->
6      <arg name="id_robot" default="campero"/>
7
8      <!-- Posición inicial -->
9      <arg name="x_init_pose" default="0"/>
10     <arg name="y_init_pose" default="0"/>
11     <arg name="z_init_pose" default="0"/>
12
13     <!-- Ejecución de Gmapping, idealmente deberíamos ejecutarlo al principio
14     y manejar manualmente el robot por el entorno para realizar un mapa y,
15     posteriormente, añadirlo -->
16     <arg name="gmapping" default="false"/>
17
18     <!-- Xacro utilizada -->
19     <arg name="xacro_robot" default="campero_rubber.urdf.xacro"/>
20
21     <!-- Mapa de Rviz -->
22     <arg name="map_file" default="map_empty.yaml"/>                    <!-- Aquí elegimos el mapa que muestra Rviz -->
23
24     <!-- Tipo de movimiento -->
25     <arg name="robot_localization_mode" default="odom"/>
26     <arg name="ros_planar_move_plugin" default="true"/>
27
28     <!-- Tipo de brazo -->
29     <arg name="3_finger_gripper" default="false"/>
30
31     <!-- Prefijo para encontrar topics -->
32     <arg name="prefix" value="$(arg id_robot)_"/>
33
```

Figura C.1: *campero_nav.launch* selección del mapa de RViz

Además también tenemos que elegir el entorno de Gazebo que queremos escanear. Para ello vamos al apartado final del mismo archivo y donde pone 'Mundo de Gazebo' seleccionamos el entorno que vamos a simular (figura C.2, línea 118).

```
110
111     <!-- Ejecutar Gazebo y Rviz-->
112     <include file="$(find campero_gazebo)/launch/gazebo_rviz.launch">
113
114         <!-- Ejecutar Rviz -->
115         <arg name="launch_rviz" value="true"/>
116
117         <!-- Mundo de Gazebo-->
118         <arg name="world" value="$(find campero_gazebo)/worlds/aruco"/>        <!-- Aquí cambiamos el mapa de Gazebo -->
119
120         <!-- Setting debug-->
121         <arg name="debug" value="false"/>
122     </include>
123 </launch>
124
```

Figura C.2: *campero_nav.launch* selección del mapa de Gazebo

Cuando tengamos todo configurado vamos a la línea de comandos y escribimos:

$ roslaunch campero_navigation campero_nav.launch gmapping:=true

Apoyándonos en RViz vamos moviendo el robot y observando como se va creando el mapa. Para guardarlo, en un nuevo terminal escribimos el comando:

$ rosrun map_server map_saver map:=/campero/map -f nombre_del_mapa

El mapa se habrá guardado en nuestro directorio al nivel de las carpetas *devel*, *source* y *src*. Tendremos un archivo en formato *png* que podremos utilizar para comprobar que el mapa se ha dibujado correctamente y otro en formato

yaml que es el que usaremos en RViz. Lo mejor será llevarlos al directorio *src/campero/campero_common/campero_localization/maps* .

Ahora que ya tenemos el mapa simplemente tenemos que cambiar el mapa vacío por nuestro mapa en el archivo 'campero_nav.launch' (línea 22, figura C.1).

C.2. Ejecución en ROS de la navegación con láseres

Para la navegación necesitaremos un mapa del entorno en el que vamos a trabajar. Si no lo tenemos, en la sección anterior se explica cómo crearlo.
Abrimos el archivo 'campero_nav.launch' con un editor de texto y ponemos el mapa que queremos abrir tanto en RViz como en Gazebo. Una vez seleccionados escribimos en la línea de comandos:

$ roslaunch campero_navigation campero_nav.launch

Una vez haya arrancado RViz si queremos que el robot vaya a una posición indicada usaremos el programa 'actionGoal.py', para indicar las coordenadas del mapa a las que queremos que se dirija el robot abrimos el archivo y indicamos los valores de x, y, *theta* (líneas 18,19 y 20 respectivamente, figura C.3). Ahora simplemente lanzamos el programa con el comando:

$ rosrun campero_navigation actionGoal.py

Cuando llegue a la posición indicada lo indicará por pantalla y el programa finalizará. Si hay algún problema también se mostrará por pantalla.

```
Set as interpreter
1  #!/usr/bin/env python
2  import rospy
3  import math
4  import time
5  import numpy
6  # Brings in the SimpleActionClient
7  import actionlib
8
9  from geometry_msgs.msg import Quaternion
10 from tf.transformations import quaternion_from_euler
11 # Brings in the .action file and messages used by the move base action
12 from move_base_msgs.msg import MoveBaseAction, MoveBaseGoal
13
14
15 goal = MoveBaseGoal() #Este dato es equivalente al PoseStamped: 3 coordenadas de posicion (x,y,z) y un cuaternion para la orientacion
16
17 # Coordenadas a las que se dirigira el robot
18 x_goal=1.0
19 y_goal=0.5
20 theta_goal=-5.0
21
22 def goal2MoveBaseGoal(x=0.0, y=0.0, theta=0): #Funcion que transforma una posicion dada por x, y e angulo en z (en grados), en formato MoveBaseGoal
23
24     goal.target_pose.header.frame_id = "campero_map"
25     goal.target_pose.header.stamp = rospy.Time.now()
26
27     #Definimos la posicion
28     goal.target_pose.pose.position.x = x
29     goal.target_pose.pose.position.y = y
30     goal.target_pose.pose.position.z = 0.0
```

Figura C.3: *actionGoal.py*

Para que el robot recorra una secuencia de coordenadas usaremos el programa 'multipleGoals.py' (programa completo en el anexo A.2). Antes de iniciar el programa,

abrimos el archivo y elegimos las coordenadas de los 'goals' (líneas de la 48 hasta la 52, figura C.4). Si queremos quitar o añadir elementos de la secuencia basta con borrar o añadir un 'goal', darle valores x, y, *theta* y añadirlo en la línea 54 (figura C.4). Para iniciar el programa introducimos el comando:

$ rosrun campero_navigation multipleGoals.py

```
38
39  def movebase_client():
40
41      # Create an action client called "move_base" with action definition file "MoveBaseAction"
42      client = actionlib.SimpleActionClient('/campero/move_base',MoveBaseAction)
43
44      # Waits until the action server has started up and started listening for goals.
45      client.wait_for_server()
46
47      goalSequence=[]         #Creamos el array que contiene la secuencia de goals
48      goal1=goal(0.0,-0.5,0)   #Creamos el goal en el formato MoveBaseGoal
49      goal2=goal(0.0,-1.0,0)
50      goal3=goal(0.0,-1.5,0)
51      goal4=goal(0.0,-2.0,0)
52      goal5=goal(0.0,-2.5,0)
53
54      goalSequence=[goal1, goal2, goal3, goal4, goal5]
```

Figura C.4: *multipleGoals.py*

Cada vez que se llegue a un goal se indicará por pantalla. Cuando se complete la secuencia también se indicará y el programa finalizará. Si hubiese algún error también se nos informaría.

C.3. Ejecución en ROS de la navegación con láseres y cámara

C.3.1. Desarrollo

Para realizar esta navegación hemos usado el paquete aruco_ros de PAL-Robotics [11]. Pero por supuesto hay que adaptarlo para el robot campero así que en el archivo 'single.launch' (figura C.5) hemos tenido que editar varias líneas de código.

```
1  <launch>
2
3      <arg name="markerId"        default="582"/>
4      <arg name="markerSize"      default="0.034"/>    <!-- in m -->
5      <arg name="eye"             default="left"/>
6      <arg name="marker_frame"    default="aruco_marker_frame"/>
7      <arg name="ref_frame"       default=""/>  <!-- leave empty and the pose will be published wrt param parent_name -->
8      <arg name="corner_refinement" default="LINES" /> <!-- NONE, HARRIS, LINES, SUBPIX -->
9
10
11     <node pkg="aruco_ros" type="single" name="aruco_single">
12         <remap from="/camera_info" to="/stereo/$(arg eye)/camera_info" />
13         <remap from="/image" to="/stereo/$(arg eye)/image_rect_color" />
14         <param name="image_is_rectified" value="True"/>
15         <param name="marker_size"        value="$(arg markerSize)"/>
16         <param name="marker_id"          value="$(arg markerId)"/>
17         <param name="reference_frame"    value="$(arg ref_frame)"/>   <!-- frame in which the marker pose will be refered -->
18         <param name="camera_frame"       value="stereo_gazebo_$(arg eye)_camera_optical_frame"/>
19         <param name="marker_frame"       value="$(arg marker_frame)" />
20         <param name="corner_refinement"  value="$(arg corner_refinement)" />
21     </node>
22
23 </launch>
24
```

Figura C.5: Archivo single.launch original

En la línea 3, pondremos el número de la marca que queremos identificar donde esta el valor ”582”, así que antes de lanzar aruco_single tendremos que asegurarnos de que está puesta la marca que vamos a usar en la navegación. En la línea 4 ponemos el tamaño en metros de nuestras marcas; en nuestro caso son 12 cm, osea que donde pone ”0.034” escribimos ”0.12”. La línea 5 la podemos eliminar ya que en nuestro caso solo vamos a utilizar una cámara. El frame que utilizaremos de referencia será el mismo que el que usamos para la odometría del robot *campero_odom*. En las líneas 10, 11 y 16 tenemos que cambiar los datos para adaptarlos a nuestro robot tal y como se muestra en la figura C.6. Por último y de manera opcional, podemos añadir en la línea 9 una opción para que se nos muestre por pantalla los valores que está usando el archivo cada vez que lo arranquemos. Para ello simplemente escribimos output=”screen”.

```
1  <launch>
2
3      <arg name="markerId"        default="1"/>
4      <arg name="markerSize"      default="0.12"/>    <!-- in m -->   <!-- default="0.034" -->
5      <arg name="marker_frame"    default="aruco_marker_frame"/>
6      <arg name="ref_frame"       default="campero_odom"/>  <!-- leave empty and the pose will be published wrt param parent_name -->
7      <arg name="corner_refinement" default="LINES" /> <!-- NONE, HARRIS, LINES, SUBPIX -->
8
9      <node pkg="aruco_ros" type="single" name="aruco_single" output="screen">
10         <remap from="/camera_info" to="/campero/campero_front_ptz_camera/camera_info" />
11         <remap from="/image" to="/campero/campero_front_ptz_camera/image_raw" />
12         <param name="image_is_rectified" value="True"/>
13         <param name="marker_size"        value="$(arg markerSize)"/>
14         <param name="marker_id"          value="$(arg markerId)"/>
15         <param name="reference_frame"    value="$(arg ref_frame)"/>   <!-- frame in which the marker pose will be refered -->
16         <param name="camera_frame"       value="campero_front_ptz_camera_optical_frame_link"/>
17         <param name="marker_frame"       value="$(arg marker_frame)" />
18         <param name="corner_refinement"  value="$(arg corner_refinement)" />
19
20     </node>
21
22 </launch>
23
```

Figura C.6: Archivo single.launch modificado

Una vez hemos adaptado el código ya podemos comprobar su funcionamiento.

Configuramos el archivo 'campero_nav.launch' para abrir el mapa que deseemos tanto en Gazebo como en RViz. Abrimos un terminal y escribimos el comando:

$ roslaunch campero_navigation campero_nav.launch

Una vez haya arrancado RViz y Gazebo abrimos otro terminal e introducimos:

$ roslaunch aruco_ros single.launch

Si hemos puesto la opción de output="screen" se nos mostrará por pantalla las características de la marca que va a detectar. Para la detección de la marca tenemos que ejecutar otro comando en una nueva terminal:

$ rosrun image_view image_view image:=/aruco_single/result

Se nos abrirá una ventana donde podemos ver la cámara del robot campero tal y como se muestra también en RViz, solo que en esta ventana cuando aparezca una marca se nos mostrará su identificador y su transformación. En esta ventana se detectan todas las marcas, independientemente de que identificador hayamos puesto en 'single.launch'. Tenemos que tener en cuenta que la transformación que se muestran en esta ventana no es la que se muestra en RViz, que es la que emplearemos para la navegación. Por defecto no se muestran las transformaciones así que tendremos que añadirlas manualmente. Para ello en el apartado "Displays" de RViz (donde aparecen: "Grid", "RobotModel A", "RearLaser", "FrontLaser", etc.), pulsamos el botón "Add", buscamos donde está la opción "TF", la elegimos y pulsamos "OK". Para hacer más fácil la tarea de acercamiento del robot a la marca, hemos creado una transformación para que la orientación y la posición se correspondan con la que el robot tendrá cuando se sitúe frente a la marca. Ahora debemos crearla, así que abrimos un nuevo terminal y escribimos:

$ rosrun aruco_ros aruco_tf.py

Esta es la última transformación que necesitábamos, así que para visualizar correctamente las que nos interesan, vamos al apartado "Frames" de "TF" y activamos los siguientes: "aruco_marker_frame"(la transformación de la marca), "campero_base_footprint"(la transformación de la base del robot), "campero_odom"(la transformación del mundo) y "aruco_tf" (la transformación que hemos creado). Todas estas transformaciones se muestran en la figura C.7 (para poder ver "campero_base_footprint" hemos desactivado la visualización del modelo 3D del robot).

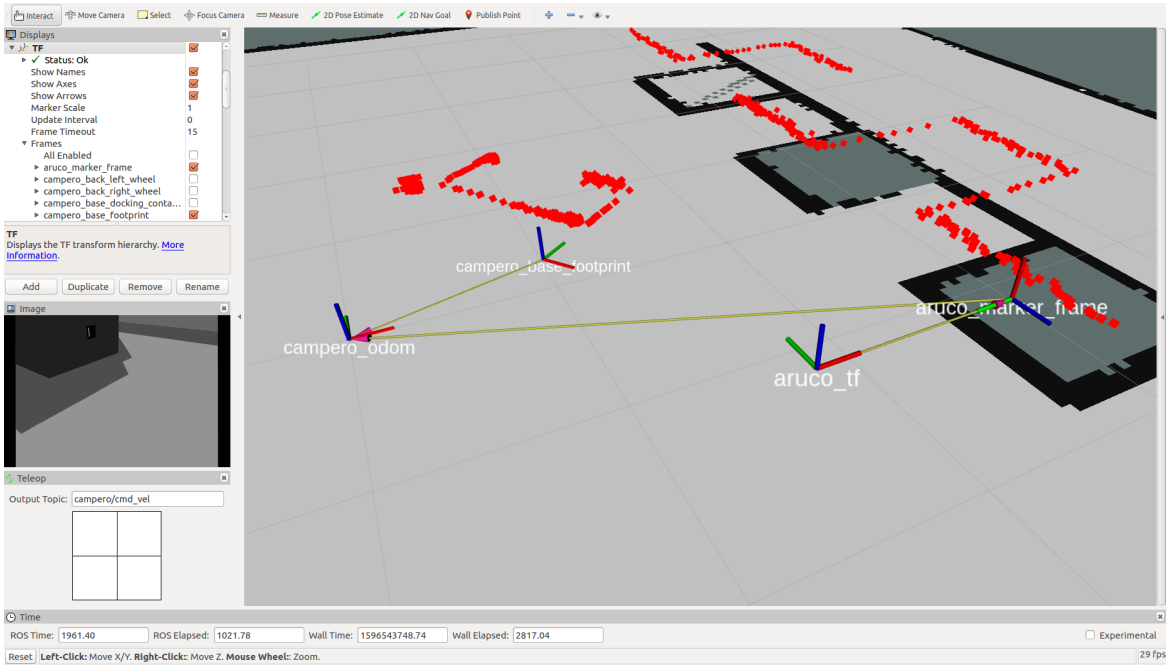

Figura C.7: Transformaciones en RViz

Antes de iniciar la navegación solo nos queda lanzar un programa que se encargue de publicar en un tópico la posición y orientación de la transformación "aruco_tf" para que el programa que se encarga de la navegación mueva el robot a esa posición. Para esta navegación tenemos que configurar el archivo transform_listener.py para obtener la tranformación entre "aruco_tf" y la transformación del mapa ("campero_odom"), es decir, la referencia global. Para ello en la línea 31 y 36 tenemos que poner "campero_odom" para la primera transformación y "aruco_tf" para la segunda, tal y como se muestra en la figura C.8.

```
27  if __name__ == '__main__':
28      rospy.init_node('aruco_listener',anonymous=True)
29      pub = rospy.Publisher('aruco_pose', PoseStamped, queue_size=10)
30      tf_listener=tf.TransformListener()
31      tf_listener.waitForTransform('campero_odom','aruco_tf',rospy.Time(), rospy.Duration(0.1))          #Cambiar aqui las transformaciones y abajo tambien
32      rate = rospy.Rate(12)
33
34      while not rospy.is_shutdown():
35          try:
36              (trans,rot)=tf_listener.lookupTransform('campero_odom','aruco_tf',rospy.Time(0))          #Cambiar las transformaciones
37          except (tf.LookupException, tf.ConnectivityException, tf.ExtrapolationException):
38              continue
39
40          x=trans[0]
41          y=trans[1]
42          z=trans[2]
```

Figura C.8: Configuración archivo transform_listener.py para navegación con cámara y láser

Ahora que ya lo hemos configurado abrimos un nuevo terminal e introducimos el siguiente comando:

$ rosrun aruco_ros transform_listener.py

Para terminar solo tenemos que lanzar el programa de navegación 'arucoNav.py' (para ver el código completo acudir al anexo A.3) cuyo funcionamiento es prácticamente el

mismo que para la navegación sin cámara. Solo que ahora la posición objetivo en vez de dársela nosotros, se la da la transformación que hemos creado. El comando es el siguiente:

$ rosrun campero_navigation arucoNav.py

C.3.2. Resumen

Como hay que abrir varios terminales y ejecutar diversos comandos vamos a resumir brevemente todas las ejecuciones que hay que hacer para llevar a cabo la navegación mediante cámara. Vamos a necesitar un total de seis terminales, en cada uno introduciremos uno de los comandos siguientes y deberán ser ejecutados en este orden:

$ roslaunch campero_navigation campero_nav.launch

$ roslaunch aruco_ros single.launch

$ rosrun image_view image_view image:=/aruco_single/result

$ rosrun aruco_ros aruco_tf.py

$ rosrun aruco_ros transform_listener.py

$ rosrun campero_navigation arucoNav.py

Cuidado si se utiliza la opción de copiar el texto para luego pegarlo en la línea de comandos porque el carácter "_" puede ser eliminado. Así que al pegar tenemos que asegurarnos que todo el texto se ha copiado correctamente.

C.4. Ejecución en ROS de la navegación con cámara

Para esta navegación tenemos que configurar el archivo single.launch tal y como hicimos en la sección anterior. Esta vez utilizaremos otra transformación distinta a "aruco_tf", utilizaremos una con la misma orientación pero se situará justo donde está la marca, a esta transformación la hemos llamado "marker_tf". Tenemos que editar el archivo transform_listener.py. Para ello en la línea 31 y 36 tenemos que poner "campero_base_footprint" para la primera transformación y "marker_tf" para la segunda, tal y como se muestra en la figura C.9.

```
if __name__ == '__main__':
    rospy.init_node('aruco_listener',anonymous=True)
    pub = rospy.Publisher('aruco_pose', PoseStamped, queue_size=10)
    tf_listener=tf.TransformListener()
    tf_listener.waitForTransform('campero_base_footprint','marker_tf',rospy.Time(), rospy.Duration(0.1))           #Cambiar aqui las transformaciones y abajo tambien
    rate = rospy.Rate(12)

    while not rospy.is_shutdown():
        try:
            (trans,rot)=tf_listener.lookupTransform('campero_base_footprint','marker_tf',rospy.Time(0))         #Cambiar las transformaciones
        except (tf.LookupException, tf.ConnectivityException, tf.ExtrapolationException):
            continue

        x=trans[0]
        y=trans[1]
        z=trans[2]
```

Figura C.9: Configuración archivo transform_listener.py para navegación con cámara

Una vez hecho esto, en el archivo campero_nav.launch elegimos el mapa que queremos abrir tanto en Gazebo como en RViz y escribimos el comando:

$ roslaunch campero_navigation campero_nav.launch

Una vez haya arrancado RViz y Gazebo abrimos otro terminal e introducimos:

$ roslaunch aruco_ros single.launch

Para la detección de la marca tenemos que ejecutar otro comando en una nueva terminal:

$ rosrun image_view image_view image:=/aruco_single/result

Ahora crearemos la transformación marker_tf, abrimos un terminal y escribimos

$ rosrun aruco_ros marker_tf.py

Antes de iniciar la navegación tenemos que lanzar transform_listener.py, por lo que introducimos en otro terminal:

$ rosrun aruco_ros transform_listener.py

Ahora ya está todo listo para iniciar la navegación, toca lanzar el programa de navegación *arucoCmd.py* (código completo en el anexo A.4) abrimos un terminal e introducimos el siguiente comando:

$ rosrun campero_navigation arucoCmd.py

Para el correcto funcionamiento del programa, tenemos que asegurarnos de que la cámara está detectando la marca antes de iniciar la navegación. Conforme se vayan completando cada una de las fases de la navegación, aparecerá un mensaje por pantalla, cuando llegue a la posición final también se nos informará y el programa finalizará.