Trabajo Fin de Grado
Grado de Ingeniería en Tecnologías Industriales

Gestión de una flota de robots móviles terrestres

Autor: Fernando Martínez Zapata
Tutor: Carlos Bordons Alba
Tutor externo: Ramón Andrés García Rodríguez

Dpto. de Ingeniería de sistemas y automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2020

Trabajo Fin de Grado
Grado en Ingeniería de Tecnologías Industriales

Gestión de una flota de robots móviles terrestres

Autor:
Fernando Martínez Zapata

Tutor:
Carlos Bordons Alba
Profesor catedrático

Tutor externo:
Ramón Andrés García Rodríguez

Dpto. de Ingeniería de sistemas y automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2020

Trabajo Fin de Grado: Gestión de una flota de robots móviles terrestres

Autor: Fernando Martínez Zapata

Tutor: Carlos Bordons Alba

Tutor externo: Ramón Andrés García Rodíguez

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

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2020

El Secretario del Tribunal

A mi familia y amigos por apoyarme en los momentos duros y celebrar los buenos.

A mis profesores por haberme enseñado lo necesario para desarrollarme como estudiante.

AGRADECIMIENTOS

Me resulta raro dar las gracias por escrito en un documento técnico, sin embargo considero también que es una parte importante puesto que de no haber conocido algunas personas no habría llegado a donde estoy ahora.

En primer lugar me gustaría dar las gracias a mis padres por estar ahí en todo momento, incluso en los momentos más complicados. Además de darme fuerza y apoyo para seguir adelante. Y en especial a mi hermano que siempre ha tenido tiempo para animarme en los momentos más duros y me ha estado ayudando tanto en la carrera como en este trabajo.

A mis amigos que siempre han estado ahí para alegrarme el día a pesar de la lejanía y para dar vueltas caminando hasta las tanta de la noche charlando sobre cualquier tema.

A mis compañeros de la carrera que sin ellos la carrera hubiera resultado aún más complicada y me han estado recordando que dentro del estudio siempre hay tiempo para salir y descansar.

Y por último, a todo el profesorado por todos los conocimientos aprendidos y por la exigencia que me ha enseñado hasta donde soy capaz de llegar.

Fernando Martínez Zapata

Sevilla, 2020

RESUMEN

El objetivo principal de este trabajo fin de grado consiste en la programación de varios robots autónomos móviles terrestres que miden la radiación del sol en diferentes puntos de una planta solar mediante un algoritmo genético que planifica las posiciones a las que tiene que ir cada uno. Para ello se ha utilizado la plataforma ROS (Robot Operating System).

Este proyecto se ha relizado en conjunto con un equipo de estudiantes y doctorandos en la Escuela Técnica Superior de Ingeniería, cuyos resultados obtenidos en simulaciones se habrían aplicado en robots reales para dotar a este proyecto más realismo y aplicación en cuanto a resultados se refiere. Sin embargo, debido a la aparición del Covid-19 se redujo solamente a las simulaciones por ordenador.

ABSTRACT

The main goal of this dissertation consists on programming several autonomous land mobile robots that measure the radiation of the sun at different points of a solar plant using a genetic algorithm that plans the positions that each one has to go to. This has been done with the ROS (Robot Operating System) platform.

This project has been carried out alongside with a team of students and doctorate students at the Higher Technical School of Engineering, whose results obtained in simulations should have been applied in real robots to make this project more realistic and applicable in terms of results. However, due to the appearance of Covid-19, it has been forced to reduce it only to computer simulations.

ÍNDICE

ÍNDICE DE FIGURAS

GUÍA DE ECUACIONES

1 INTRODUCCIÓN

1.1 Los robots

Los robots son máquinas automáticas programables capaces de realizar determinadas operaciones de manera autónoma y sustituir a los seres humanos en algunas tareas, en especial las pesadas, repetitivas o peligrosas. Además puede estar dotada de sensores que le permiten adaptarse a nuevas situaciones, como se cuenta en [1].

Se podría decir que el primer robot de la historia fue en siglo IV antes de Cristo, cuando el matemático griego Arquitas de Tarento construyó un ave mecánica que funcionaba con vapor y a la que llamó *La paloma*. Aunque las máquinas totalmente autónomas no aparecieron hasta el siglo XX con el primer robot programable y dirigido de forma digital llamado Unimate [2]. Este robot fue instalado en 1961 para levantar piezas calientes de metal de una máquina de moldeo y colocarlas.

A dia de hoy, se puede observar que los robots son cada vez más independientes del ser humano gracias al avance de la tecnología hasta el punto de llegar a ser completamente autónomos. Tal y como se indica en [3], para que un robot llegue a ser completamente autónomo tiene que tener la capacidad de:

- Obtener información de su entorno.
- Trabajar durante un tiempo prolongado sin intervención humana.
- Moverse total o parcial de si mismo a través del entorno sin que intervenga ningún ser humano.
- Evitar situaciones perjudiciales para las personas, bienes o sí mismo.
- Autoaprendizaje para el cumplimiento de sus tareas o para adaptarse a posibles cambios de su entorno.

Sin embargo, todavía requieren de un mantenimiento regular. En la Figura 1.1, se ve un ejemplo de robot que cumple gran parte de los requisitos de autónomo.

Figura 1.1 Aspirador IROBOT Roomba. Obtenida de [22].

Las ventajas de estos robots y por ello la existencia de un alto interés son por el aumento de la eficiencia, seguridad y comodidad para los seres humanos. Aunque también tienen sus desventajas como la pérdida de puestos de trabajos y la dependencia a los robots ya que forman parte de nuestras vidas, como se explica en [2].

Una vez abordada una breve introducción sobre los robots, es momento de comentar sobre el proyecto en el que se basa este trabajo.

1.2 OCONTSOLAR

El trabajo que abarca esta memoria forma parte del proyecto OCONTSOLAR (control óptimo de sistemas de energía solar térmica, Figura 1.2) que tiene como objetivo desarrollar nuevos métodos de control para utilizar sensores móviles montados en drones y vehículos terrestres no tripulados (UGV) como parte integral de los sistemas de control. Es por ello que se ha utilizado las plantas de energía solar como estudio de caso con el objetivo de optimizar su funcionamiento mediante estimaciones y predicciones de irradianza espacial. Los principales objetivos se podrían resumir en:

1. Métodos para controlar flotas de sensores móviles e integrarlas como parte esencial de los sistemas de control generales.
2. Métodos de estimación de la irradiancia solar distribuidos espacialmente mediante una flota variable de sensores montados en drones y UGV.
3. Algoritmos de control predictivo de nuevo modelo (MPC) que utilizan estimaciones y predicciones de sensores solares móviles para producir una operación más segura y eficiente de las plantas, lo que permite la integración efectiva de la energía solar en los sistemas que suministran energía a las redes u otros sistemas mientras se cumplen los compromisos de producción.

Además OCONTSOLAR ya incluye pruebas de conceptos por implementación en la Plataforma Solar de Almería. Este proyecto es de alto riesgo y alta ganancia con objetivos extramadamente ambiciosos e innovadores que aprovechan las nuevas oportunidades que presentan las últimas tecnologías de detección móvil y se enfrentan a los exigentes problemas de control que surgen debido a la necesidad de integrar sensores móviles en sistemas de control, produciendo estimaciones de irradiancia espacial y pronosticando y manejando sistemas interconectados a gran escala con topologías cambiantes. Además de investigar los aspectos más teóricos, el equipo de OCONTSOLAR también espera encontrar soluciones de control innovadoras para el control de plantas de energía solar y otros sistemas complejos a gran escala, como se cuenta en [4].

Figura 1.2 Proyecto OCONTSOLAR. Obtenida de [4].

1.3 Objetivo del proyecto

A lo largo de esta memoria se explicará el estudio y el proceso que se ha llevado a cabo para programar varios robots completamente autónomos para que recorran un campo solar, el cual puede observarse en la Figura 1.3, midiendo la radiación del sol, de forma que optimice la captación de energía solar de las placas solares.

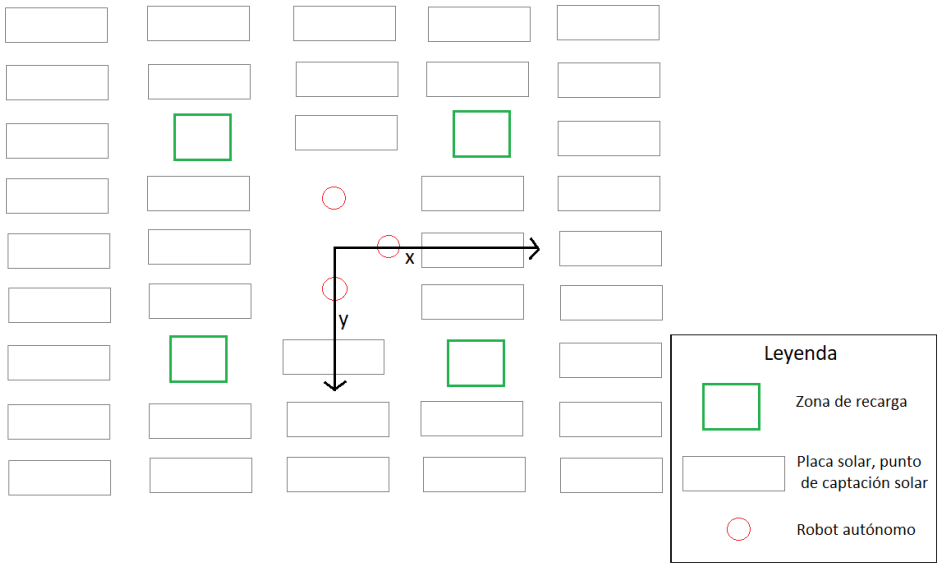

Figura 1.3 Plano del mapa del parque solar visualizado en Gazebo.

A priori el objetivo de este proyecto era utilizar robots reales que se moviesen por un campo de futbol simulando como si fuese un campo solar, sin embargo, la pandemia del Covid-19 ha imposibilitado trabajar con los robots físicos. Por lo tanto se optó por realizar dicho trabajo mediante un simulador, llamado Gazebo.

A continuación se expondrá los siguientes capítulos que ayudarán al seguimiento y entendimiento del trabajo:

- Capitulo 2: Sistema Operativo Robótico (ROS)
- Capitulo 3: Herramientas del proyecto
- Capitulo 4: Navegación con “move_base”
- Capítulo 5: Programación del robot
- Capítulo 6: Resultados
- Capítulo 7: Posibles ampliaciones
- Capítulo 8: Conclusiones

2 SISTEMA OPERATIVO ROBÓTICO (ROS)

ROS es un sistema operativo que permite el desarrollo del software de los robots. Al igual que posibilita el control de dispositivos de bajo nivel, implementación de funcionalidad de uso común, paso de mensajes entre procesos y mantenimiento de paquetes. Como se explica en [5], está basado en una arquitectura de grafos donde el procesamiento toma lugar en los nodos que pueden recibir, mandar y multiplexar mensajes de sensores, control, estados, planificaciones y actuadores, entre otros. La librería está orientada para un sistema UNIX (Ubuntu, Linux).

Figura 2.1 Robot Operating System – ROS. Obtenida de [23].

2.1 Conceptos

A continuación para entender mejor ROS se explicará los conceptos más importantes que más se utlizan, como se detalla en [6]:

a) Nodos

Son archivos ejecutables dentro de paquetes de ROS, los cuales utilizan librerías para comunicarse con otros nodos. Además en caso de querer enviar o recibir algún tipo de mensaje necesita publicar o suscribirse a un tema. En este proyecto se ha utilizado C++ como lenguaje de programación para ejecutar estos archivos.

b) Temas

Son buses a través de los cuales los nodos pueden intercambiar mensajes. Los temas son unidireccionales para la comunicación en tiempo real. En caso de necesitar realizar llamadas a procedimientos remotos, es decir, recibir una respuesta a una solicitud, deben utilizar servicios en su lugar.

c) Mensajes

Los nodos se comunicán entre sí mediante la publicación de mensajes sobre temas. Un mensaje es una estructura de datos simple, que comprende campos escritos. Se admiten los tipos primitivos estándar (entero, punto flotante, booleano, etc.), al igual que las matrices de tipos primitivos. Los mensajes pueden incluir estructuras y matrices anidadas arbitrariamente. Existen diferentes tipos de mensajes dependiendo de que se quiere enviar.

d) Maestro

Proporciona servicios de nomenclatura y registro al resto de los nodos del sistema ROS. Realiza un seguimiento de los editores y suscriptores de temas y servicios. La función del maestro es permitir que los nodos ROS individuales se ubiquen entre sí. Una vez que estos nodos se han ubicado entre sí, se comunican entre sí de igual a igual.

e) Espacio de trabajo

Es una carpeta en la que se modifica, crea e instala paquetes catkin. En esta carpeta será donde guardarán todos los archivos ejecutables del proyecto.

2.2 Herramientas de líneas de comando

En el presente apartado se explicarán las líneas de comando más utilizadas para el desorrollo de este trabajo:

a) Roscore

Es el comando fundamental por excelencia en ROS, el cual es imprescindible para que los nodos funcionen. Al ejecutarlo ponemos en marcha el maestro, el servidor de parámetros y rosout que es el log donde se puede mostrar datos por depuración entre otros, como bien se explica en [7].

b) Rosrun

Es el comando que permite ejecutar el nodo deseado, cuya estructura, que se indica en [8], es la siguiente:

rosrun [nombre de paquete] [nombre del nodo]

c) Roslaunch

Es el comando que permite ejecutar el maestro y varios nodos simultáneamente sin necesidad de ejecutar roscore previamente. Su estructura, que se indica en [9], es la siguiente:

roslaunch [nombre del paquete] [nombre del archivo .launch]

d) Rostopic

Es capaz de mostrar una lista de temas activos, la tasa de publicación de un tema, el ancho de banda de un tema, los mensajes publicados en un tema, los editores y suscriptores de un tema específico. Esta herramienta es muy útil puesto que se puede saber fácilmente los temas que hay, a que nodos están conectados y el tipo de mensaje que transmite, como se detalla en [10].

e) Rosnode

Es capaz de mostrar información de depuración sobre los nodos de ROS, incluidas publicaciones, suscripciones y conexiones. También contiene una biblioteca experimental para recuperar información de los nodos. Esta herramienta es muy útil puesto que se puede saber fácilmente los nodos que hay y a que temas está publicando y/o suscrito, como se explica en [11].

f) Catkin_make

Es una herramienta muy práctica que permite crear código en un espacio de trabajo. Siempre se le debe llamar en la raíz de su espacio de trabajo, detallado en [12].

3 HERRAMIENTAS DEL PROYECTO

Como todo proyecto, han sido imprescindibles algunas herramientas para la realización de este trabajo. Estos integrantes, en general, se pueden agrupar de la siguiente forma:

3.1 Gazebo

Es el simulador que se utiliza para mostrar cómo se mueven todos los robots por todo el campo solar, el cual puede observarse en la Figura 3.1, comprobando que efectivamente se desplazan a las posiciones destinadas de cada uno.

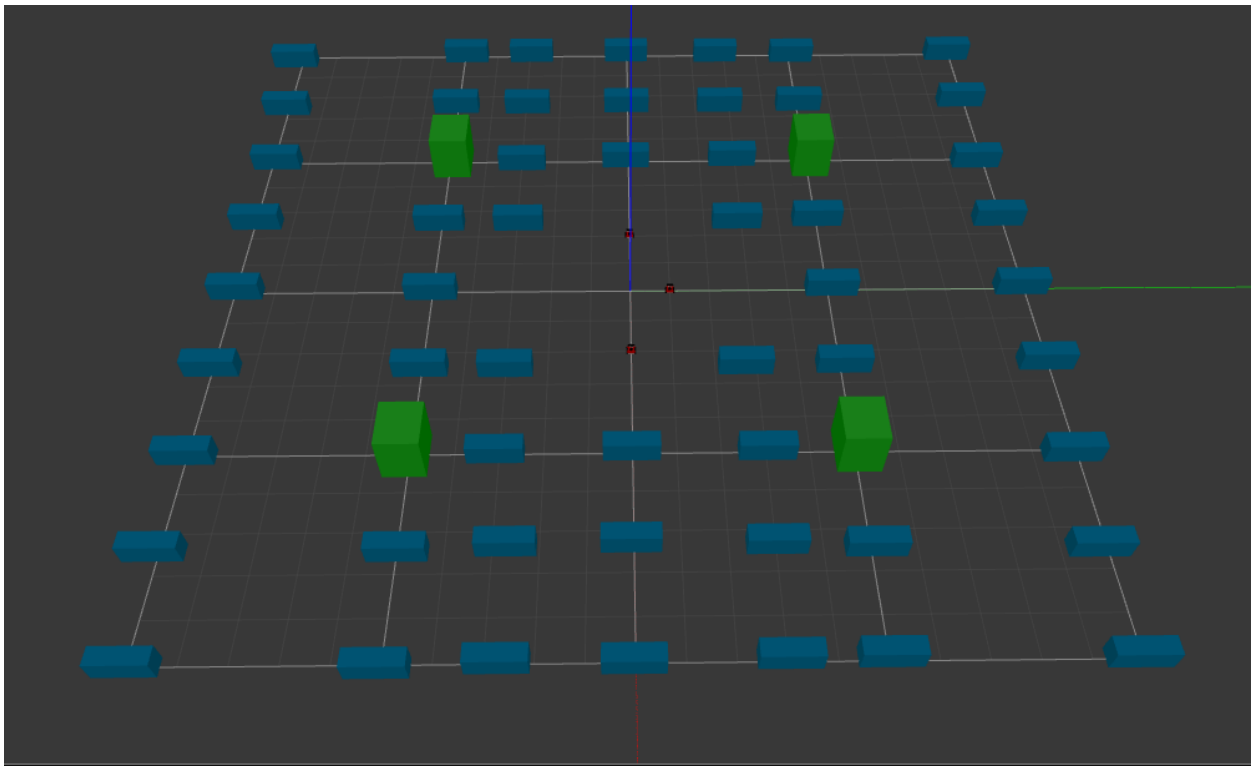

Figura 3.1 Parque solar visualizado en Gazebo donde se puede ver en color rojo los robots, en azul los paneles solares y en verde los puntos de recarga.

3.2 Matlab

Es un programa matemático muy complejo y completo que te permite realizar operaciones muy complejas que un ser humano no sería capaz de realizar a papel, tal y como se detalla en [13].

En este proyecto se usará para que planifique la distribución de puntos objetivos que tiene que ir cada robot mediante un algoritmo genético, el cual se encuentra dentro del propio Matlab. Para ello, se ha utilizado un artículo sobre la asignación de tareas basada en eventos MILP para redes de sensores robóticos heterogéneos para plantas termosolares, el cual propone un algoritmo basado en eventos MILP para resolver el problema de asignación de tareas en una red de robóticos (RSN), como se profundiza en [14].

3.3 Rosbot 2.0

Rosbot 2.0, el cual es fabricado por Husarion DOC [15], integra varios componentes:

- CPU: Rockchip RK3288, procesador ARM Cortex-A17 de cuatro núcleos de 32 bits, 1,8 GHz
- Procesador gráfico: ARM Mali-T764 MP2
- RAM: 2 GB LPDDR3
- Plataforma móvil de 4 ruedas que contiene motores de CC con codificadores y marco de aluminio.
- Cámara Orbbec Astra RGBD
- Escáner láser RPLIDAR A2
- Sensor inercial MPU 9250 (acelerómetro + giroscopio)
- Sensor de distancia de tiempo de vuelo VL53L0X
- Panel trasero que proporciona interfaces para módulos adicionales
- Chasis de aluminio robusto

Además dispone cuatro ruedas que le permite girar sobre si mismo, lo que facilita bastante su desplazamiento hacia el punto de destino. En la Figura 3.2 se puede ver el robot en cuestión.

Figura 3.2 Robot autónomo - Rosbot 2.0. Obtenida de [15].

4 NAVEGACIÓN CON EL PAQUETE "MOVE_BASE"

El paquete "move_base" proporciona una implementación de una acción que, dado un objetivo en el mundo, intentará alcanzarlo con una base móvil. El nodo "move_base" vincula un planificador global y local para realizar su tarea de navegación global y también mantiene dos mapas de costos, uno para el planificador global y otro para un planificador local que se utilizan para realizar tareas de navegación.

Además este nodo proporciona una interfaz ROS para configurar, ejecutar e interactuar con la pila de navegación en un robot. Se puede observar en la Figura 4.1 el alto nivel del nodo "move_base" y su interacción con otros componentes, como se detalla en [16].

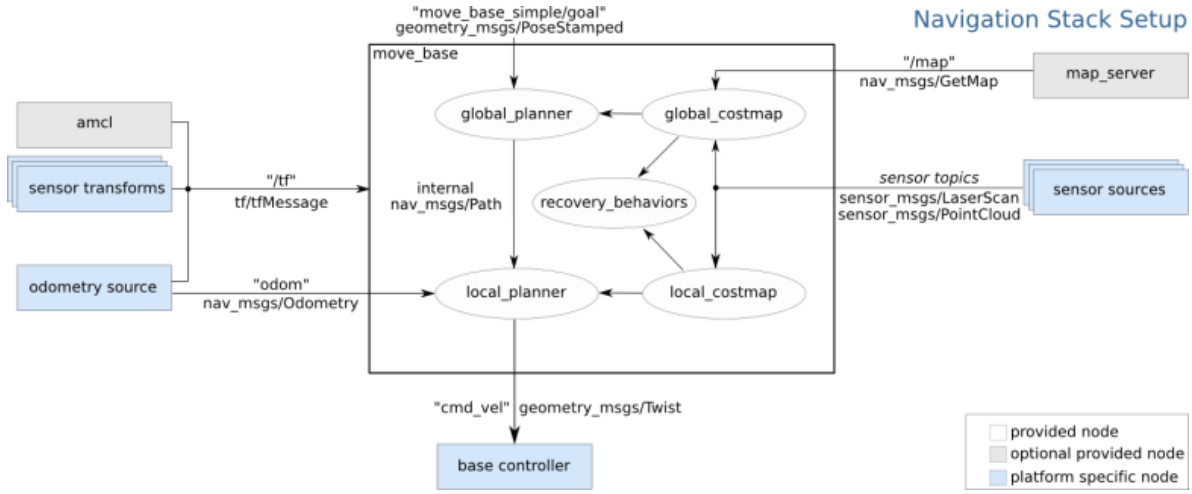

Figura 4.1 Esquema del paquete move_base. Obtenida de [16].

Los módulos azules varían según la plataforma del robot mientras que los módulos grises son opcionales pero se proporcionan para todos los sistemas, y los nodos blancos son obligatorios pero también se proporcionan para todos los sistemas.

La verdadera utilidad de ejecutar el nodo "move_base" en un robot es cumplir la misión de mover un robot de un punto a otro esquivando los obtáculos que se vaya encontrando durante el camino. Además tiene una cierta tolerancia, especificada por el usuario, para considerar que la base del robot ha llegado a su destino.

Al final se optó por utilizar "move_base" para mover el robot por su sencillez y por obtener resultados aceptables aunque de vez en cuando daba errores dificultando que el robot llegase a su destino. Sin embargo, al principio se optó por realizarlo de otra manera, la cual se explicará a continuación.

4.1 Navegación mediante un controlador PID

El código que se realizó consistía en recibir el punto objetivo leyendo un documento txt y guardándolo en una matriz llamada *matrix1* que hace función de pila donde guarda todas las misiones. Una vez que el robot obtuviese el objetivo, sabiendo la posición del robot mediante coordenadas polares calcularía θ. Luego giraría el robot hasta alcanzar ese ángulo. Y por último, el robot se movería hacia adelante hasta alcanzar el punto objetivo.

A continuación en la Figura 4.2 se muestra la trayectoria de un robot hacia un punto objetivo, donde R es la

posición del robot, *O* la posición del punto objetivo, *r* la longitud del vector entre *R* y *O* , *a* la proyección de *r* en coordenadas X, *b* la proyección de *r* en coordenadas Y, y θ el ángulo que forma el vector *r* con la coordenada X.

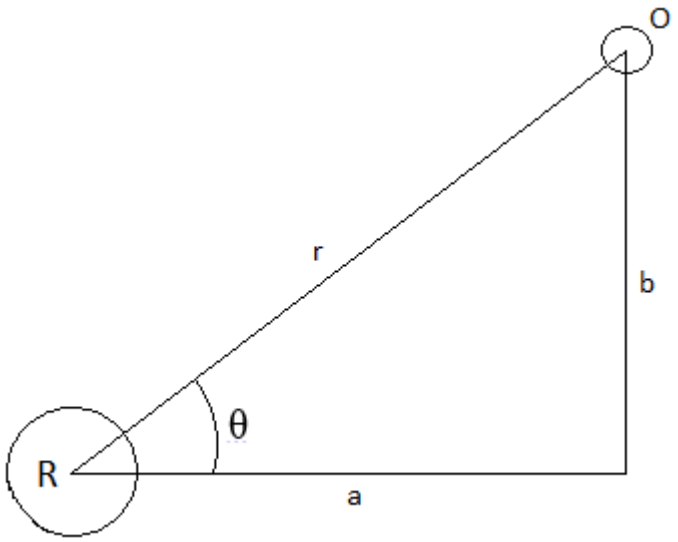

Figura 4.2 Desplazamiento del robot mediante controlador PID.

Conociendo las posiciones del robot y del punto objetivo se puede calcular θ con la siguiente fórmula.

$$a = x_o - x_R$$
$$b = y_o - y_R \qquad 4.1$$
$$\theta = \text{atan}(b/a)$$

siendo x_o, y_o las coordenadas del punto objetivo y x_R, y_R las coordenadas del robot.

Una vez calculado el valor de θ se procede a obtener las variables del controlador PID para asignarle una velocidad de giro al robot.

```
error1=theta-msg->pose.pose.orientation.z;

cumerror1=cumerror1+error1;

set_vel1.angular.z=kp*error1+ki*cumerror1+kd*(error1-lasterror1);

lasterror1=error1;
```

4.2

siendo *theta* el valor de θ, *mgs->poso.pose.orientation.z* el angulo que ha girado el robot, *cumerror1* el sumatorio de todos *error1* que va calculando, *kp*, *ki* y *kd* son constantes propia del controlador (obtenidos mediante “prueba y error”), *lasterror1* es el error anterior y *set_vel1.angular.z* es la velocidad de giro.

Luego se le asigna al robot *set_vel1.angular.z* para que gire hasta alcanzar el ángulo θ. Dado que esta velocidad de giro se calcula continuamente, su valor variará comenzando con un valor alto y terminando siendo bajo para no girar de más.

a) Explicación del código:

Para ello, se creó un nodo llamado *robot1*, situado en el Anexo A, que permite controlar el robot de la siguiente manera, el cual se compone de tres funciones.

- Main:

  Esta es la función principal que será la primera que se ejecute y la que llamará a las otras funciones. En primer lugar inicializa el nodo y luego notifica a ROS sobre que temas publicar y suscribir, de los cuales son: *action_pub1* (mueve el robot), *sub1* (obtiene las coordenadas del robot) *y sub2* (lee los nuevos objetivos). La variable *sub1* será la que llame a la función *counterCallback1* cada vez que reciba un mensaje del tema suscrito, mientras que *sub2* realiza lo mismo pero con otro tema llamando *counterCallback2*.

  Los dos temas a los que se suscribe este nodo son: */Robot1/odom* y */array*, los cuales pertenecen a las funciones *counterCallback1* y *counterCallback2* respectivamente.

- CounterCallback1:

  Esta función se encarga de controlar el robot. Para ello, en primer lugar pregunta si hay un nuevo objetivo; en caso de haberlo, guarda las coordenadas en las variables x e y. Luego pregunta en que cuadrante se encuentra el punto objetivo dado que será importante a la hora de saber el sentido y ángulo de giro. A continuación, calcula el valor de θ con la ecuación 4.1. En caso de ser 90º o -90º se introduce directamente el valor sin necesidad de usar la ecuación debido a que en esos casos *atan* devuelve infinito. En caso de ser 180º también se introduce su valor directamente al ser el extremo del rango de giro que tiene el robot. Para el resto de ángulos se calcula θ mediante la ecuación 4.1, la cual hay que pasarlo primero a radianes y luego realizar una conversión a un valor entre -1 y 1 dado que será lo que realmente reciba el robot para que lo entienda. Esta transformación se obtuvo mediante el método "prueba y error" hasta conseguir una relación lo suficientemente adecuada.

  Luego el robot gira y pregunta si ha alcanzado el ángulo deseado. De no ser así, se aplica de nuevo la ecuación 4.2 para obtener la velocidad de giro de forma que el robot siga girando, mientras que si lo fuera, deja de girar y asigna un valor a la velocidad lineal para que se mueva el robot hacia adelante. En el momento de que el robot alcance el punto objetivo, se para y lee el siguiente objetivo en *matrix1* en caso de que lo hubiera.

- CounterCallback2:

  Esta función se encarga de recibir las nuevas misiones a través del tema */array*. Así que cada vez que reciba un nuevo mensaje lo guarda en *matrix1* para que el robot las vaya cumpliendo en el mismo orden que las va recibiendo.

b) La desventaja de este tipo de navegación:

El problema que conlleva este tipo de navegación es el hecho de que a pesar de llegar adecuadamente al primer punto objetivo, cuando se dirige al segundo no lo alcanza debido a los errores que va acumulando a medida de que se mueve.

4.2 Navegación mediante el paquete "move_base"

Este paquete se puede ejecutar tanto desde la misma terminal, tal y como se muestra en la Figura 4.3, como por código escrito en un nodo. En caso de que se usara en la terminal solo habría que publicar, con el comando *rostopic pub*, la posición y orientación que quiere que tenga el robot en el tema *move_base_simple* del robot en cuestión, teniendo claramente abierto el mundo gazebo.

```
root@fernando-HP-Laptop-15-da0xxx:~# rostopic pub /Robot1/move_base_simple/goal
geometry_msgs/PoseStamped "header:
 seq: 0
 stamp:
  secs: 0
  nsecs: 0
frame_id: "/Robot1/map"
pose:
 position:
  x: 4
  y: 0
  z: 0
 orientation:
  x: 0
  y: 0
  z: 0
  w: 0"
```

Figura 4.3 Ejecución del comando *rostopic pub* para mover el robot 1.

En caso de querer ejecutarlo en un nodo, lo primero es declarar la variable donde se va a publicar el mensaje y la variable donde se guardará el mensaje en cuestión. Luego se le asigna el tema al que va a publicar y el tipo de mensaje que va a enviar. Después se le asigna al mensaje: el nombre del mapa, la posición y orientación que quiera que esté el robot. Y por último, se envía el mensaje en el tema que se mencionó antes. Se puede encontrar en el Anexo B.

En las Figuras 4.4 a 4.10 se puede observar la trayectoria de un robot utilizando “move_base”.

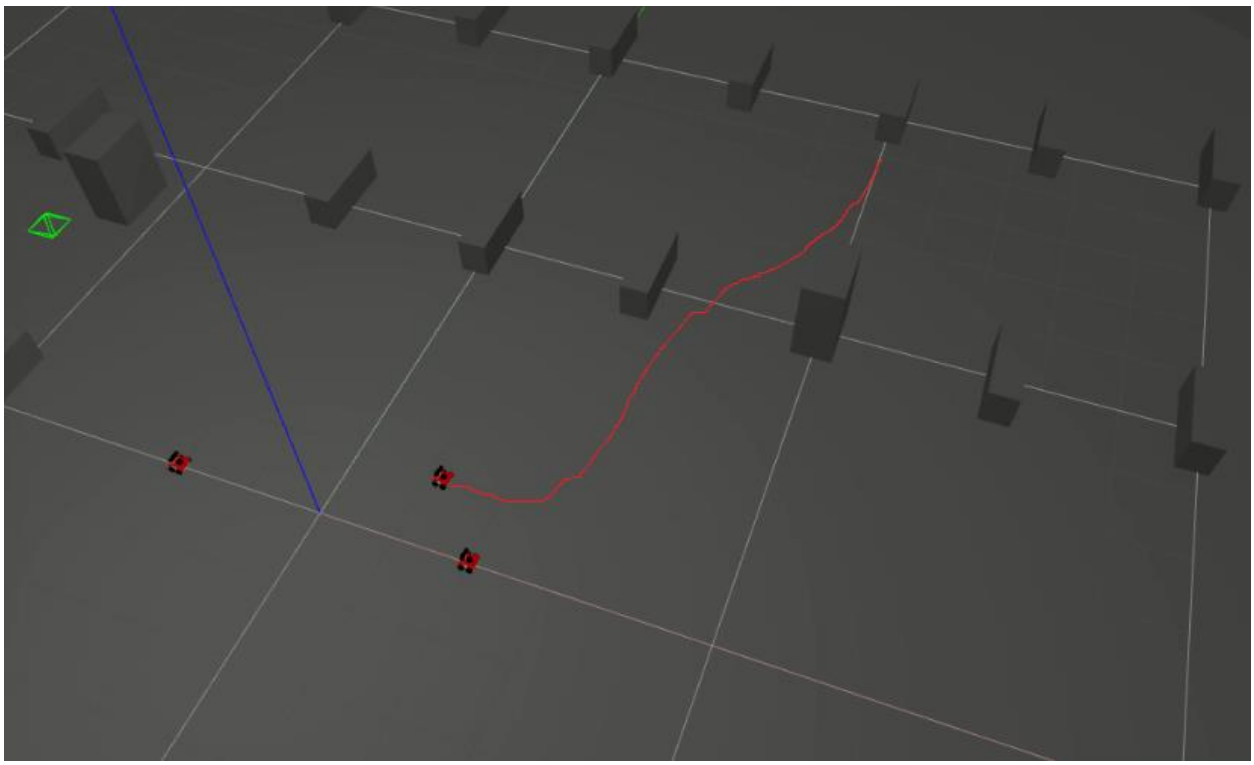

Figura 4.4 Desplazamiento del robot visualizado en Gazebo (1).

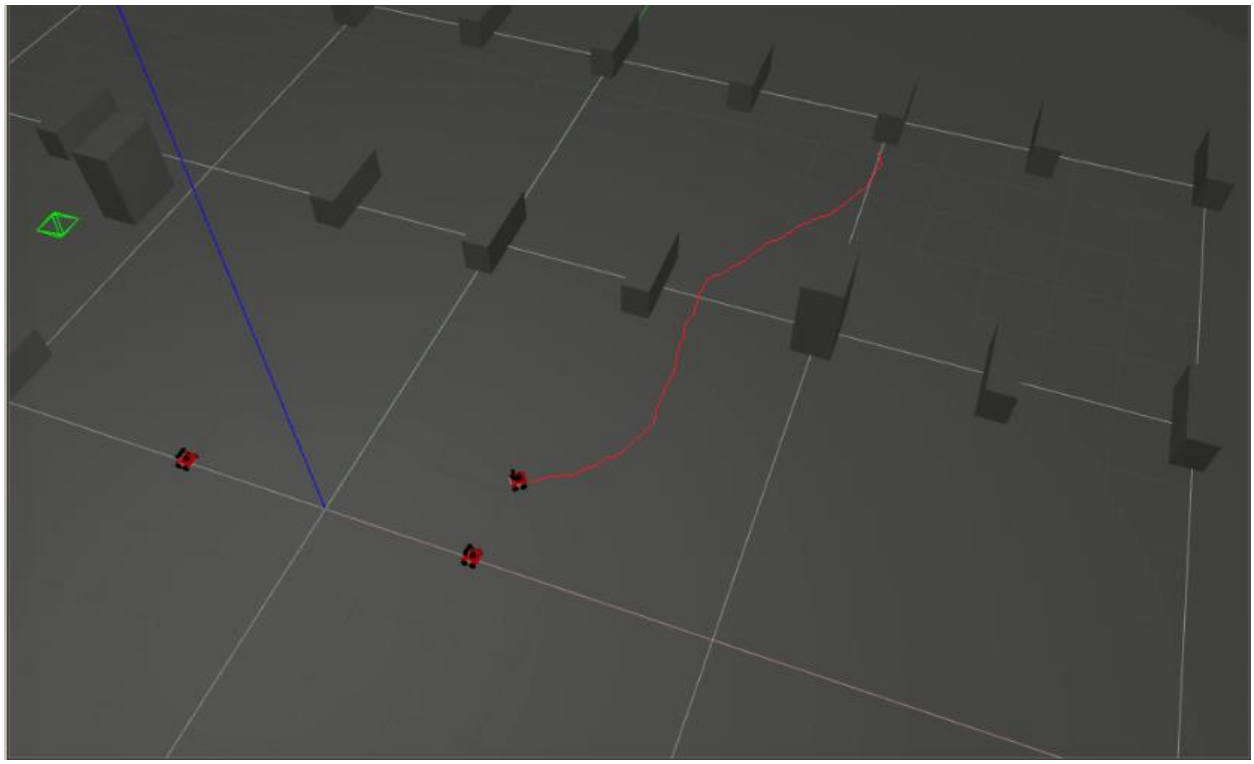

Figura 4.5 Desplazamiento del robot visualizado en Gazebo (2).

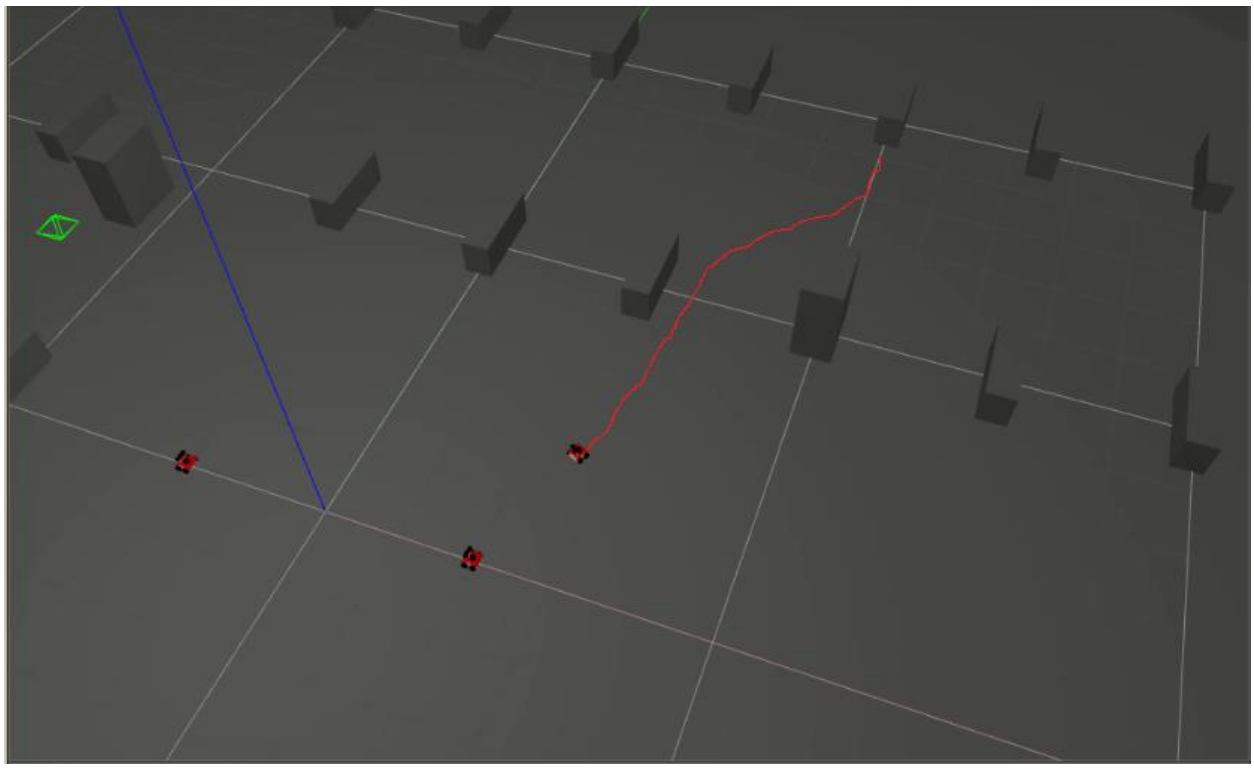

Figura 4.6 Desplazamiento del robot visualizado en Gazebo (3).

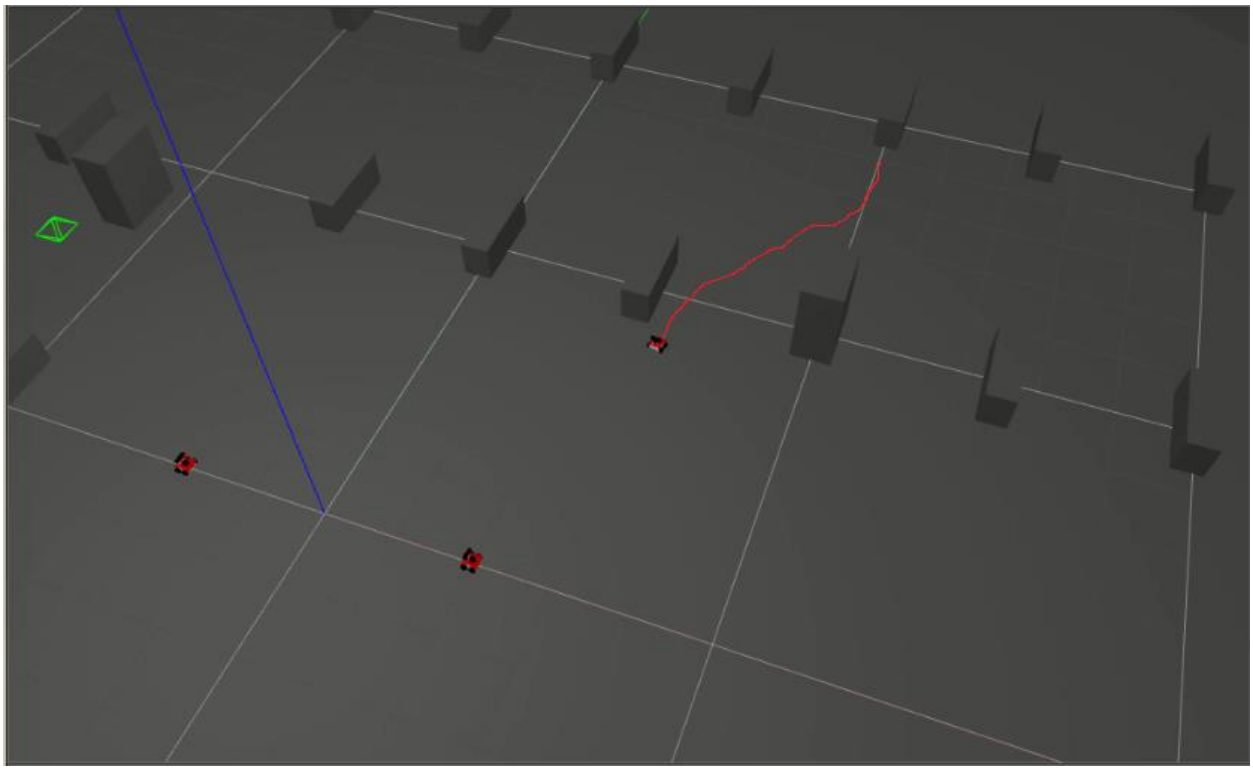

Figura 4.7 Desplazamiento del robot visualizado en Gazebo (4).

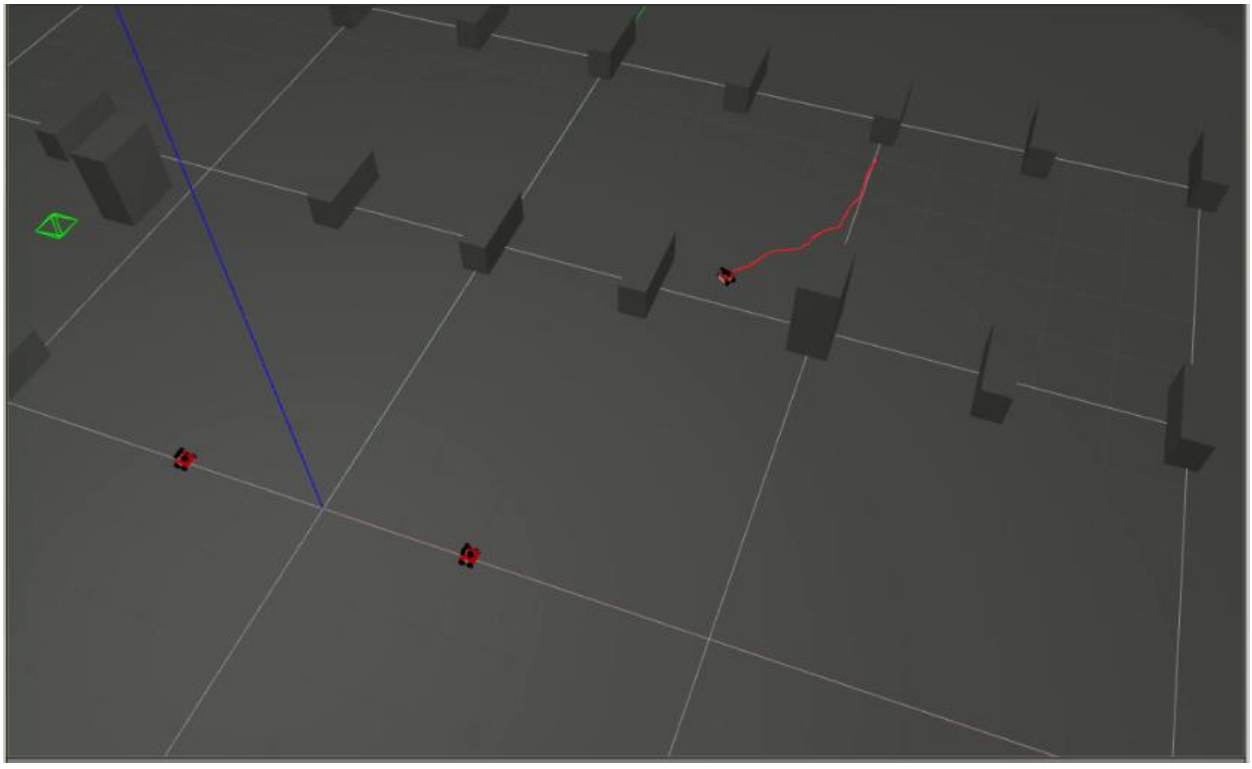

Figura 4.8 Desplazamiento del robot visualizado en Gazebo (5).

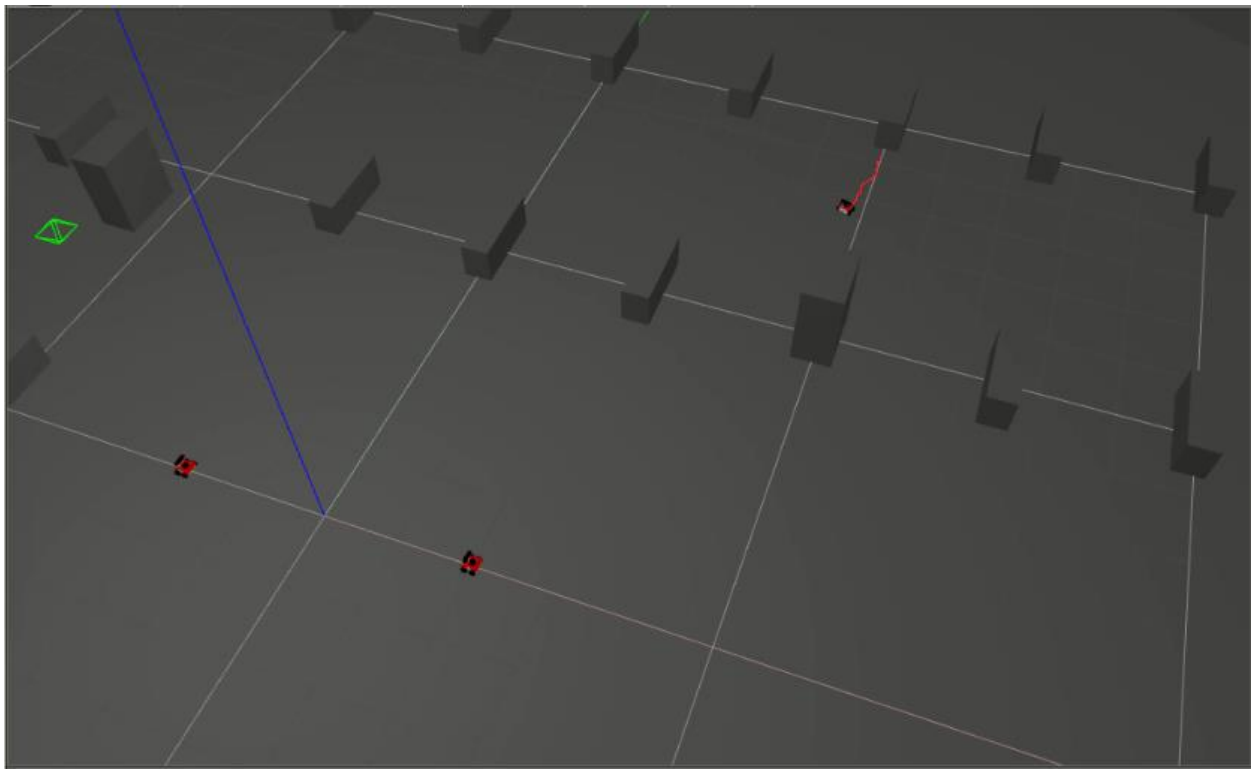

Figura 4.9 Desplazamiento del robot visualizado en Gazebo (6).

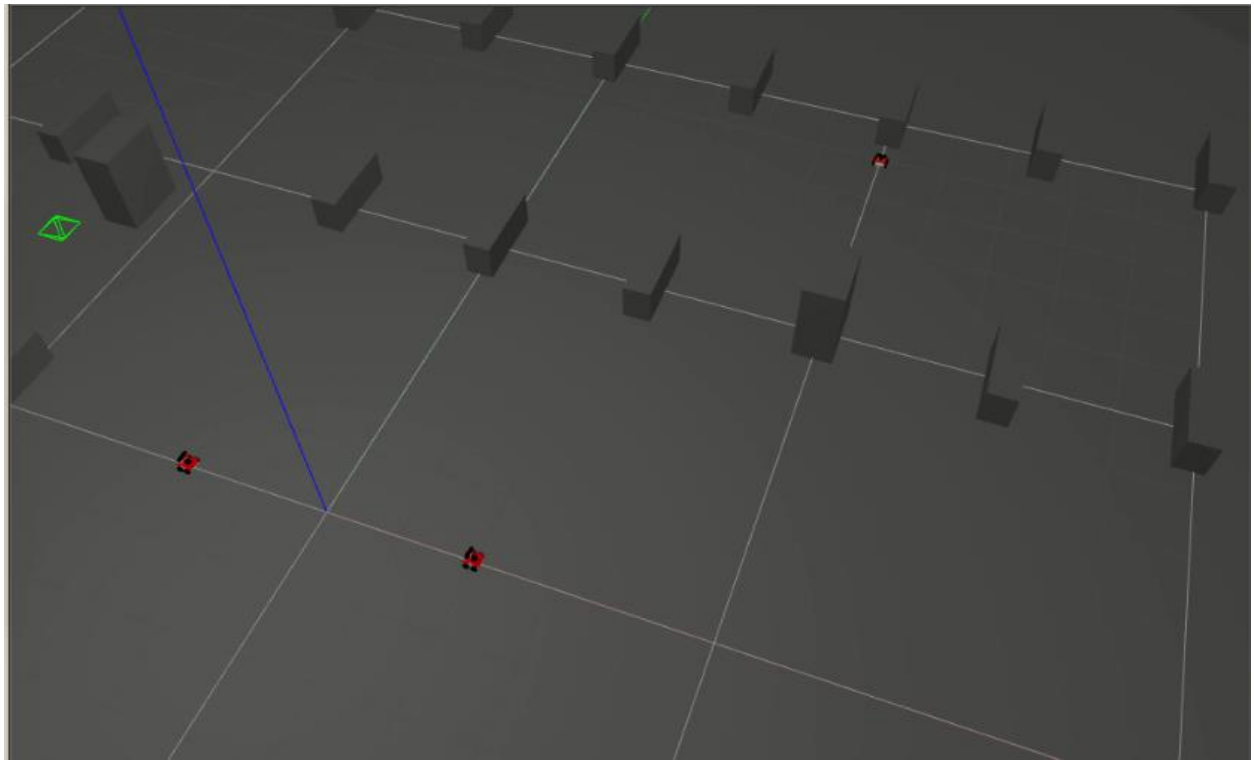

Figura 4.10 Desplazamiento del robot visualizado en Gazebo (7)

.

5 PROGRAMACIÓN

Este capítulo es el más importante de todos puesto que se tratará de explicar paso a paso lo que se ha ido realizando para llevar a cabo este trabajo. En primer lugar se ha tenido que instalar el sistema operativo Ubuntu, como se explica en [17], debido a que es necesario para que funcione correctamente Gazebo. Luego se ha tenido que descargar con la terminal los paquetes de ROS, Gazebo y opencv (para poder visualizar la cámara), como se detalla en [18] y [19]. Una vez hecho esto, en la misma terminal se crea un espacio de trabajo donde guardaremos todos los archivos ejecutables necesarios para el trabajo. Para ello se ha utilizado estos comandos:

- mkdir -p ~/catkin_ws/src
- cd ~/catkin_ws/src
- catkin_init_workspace
- cd ~/catkin_ws
- catkin_make
- source devel/setup.sh

Como se podrá observar se ha creado un espacio de trabajo llamado *catkin_ws*, en el cual habrá tres carpetas llamadas *src*, *devel* y *build*.

- Src:

  Es el espacio que contiene el código fuente de los paquetes catkin. Aquí es donde puede extraer, verificar y clonar el código fuente de los paquetes que desea compilar. Cada carpeta dentro del espacio de origen contiene uno o más paquetes catkin. Este espacio debe permanecer sin cambios al configurar, construir o instalar. CMake invoca este archivo durante la configuración de los proyectos catkin en el espacio de trabajo. Principalmente será la carpeta en la que guardaremos los archivos que se irán creando.

- Devel:

  Es donde se colocan los objetivos construidos antes de ser instalados. La forma en que se organizan los objetivos en el espacio de desarrollo es la misma que su disposición cuando se instalan. Esto proporciona un entorno de prueba y desarrollo útil que no requiere invocar el paso de instalación.

- Build:

  Es el espacio de compilación donde se invoca a CMake para compilar los paquetes catkin en el espacio de origen. CMake y catkin guardan aquí su información de caché y otros archivos intermedios.

5.1 Programación de los launch

Luego se ha creado tres archivos launch para ejecutar el mundo en Gazebo y los robots que se encuentran en él:

- *Main.launch*, situado en el Anexo C:

  En este archivo se ejecuta el mundo que aparecerá en Gazebo y también el archivo *robots.launch*.

- *Robots.launch*, situdo en el Anexo D:

  En este archivo se crea tanto robots como se haya puesto, imponiendo su nombre y posición en el mundo. Además se ejecuta *one_robot.launch* para cada robot.

- *One_robot.launch*, situado en el Anexo E:

  En este archivo se ejecuta el nodo *gmapping* para cambiar el nombre de los temas del robot y así diferenciarlo de los otros robots. También se ejecuta el nodo *move_base* para configurar las líneas de comandos y obtener parámetros, así como cargar y volcar el estado del servidor de parámetros al archivo.

  Este procedimiento estructurado se puede observar en la Figura 5.1.

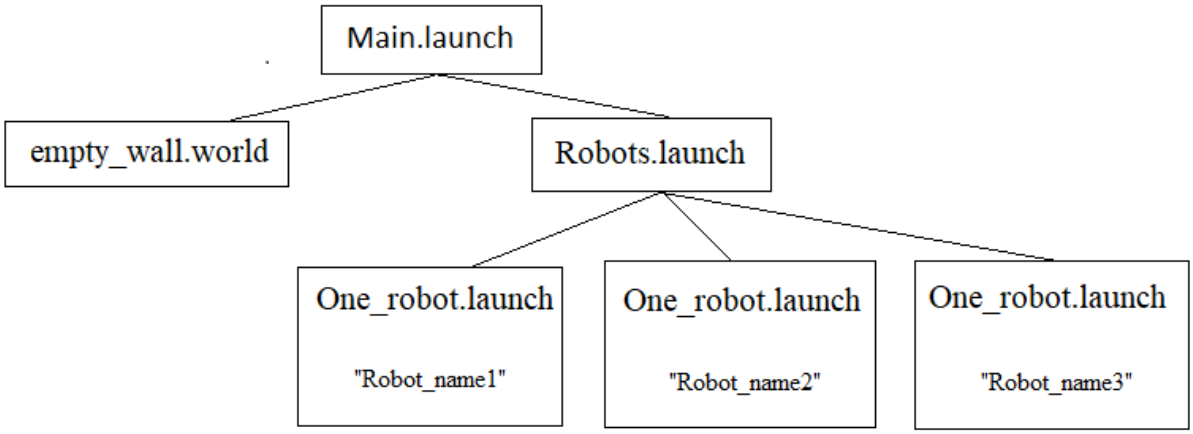

Figura 5.1 Estructuta de la ejecución de los launch.

A continuación se escribió en archivos (tipo cpp) líneas de códigos, los cuales acturán como nodos que servirán para mover los robots, mandar mensajes, visualizar las cámaras, etc...

Para programar los robots se podría utilizar tanto Python como C++ como lenguaje de programación. Para este trabajo se optó por C++ ya que había más información en este lenguje.

5.2 Programación de los robots

En este apartado se va a explicar los códigos necesarios para controlar los robots y leer los nuevos objetivos que se ha asignado a cada robot.

Dado que hay tres robots, hay tres nodos para cada uno de ellos llamados *move_base_robot1*, *move_base_robot2* y *move_base_robot3*, que se encargan de controlar los robots. A continuación para hacer referencia a estos nodos de forma genérica se le llamará *move_base_robot*, el cual puede observarse en el Anexo H. Por otra parte, hay otro nodo llamado *nuevas_misiones*, que se encuantra en el Anexo I, el cual se encarga de leer las nuevas misiones aportadas por Matlab y mandárselas a cada robot. Además, también se ha creado tres nodos de los cuales cada uno muestra la camera de un robot llamados *image_converter_1*, *image_converter_2* y *image_converter_3*, de los cuales se puede observar uno de ellos en el Anexo G puesto que el resto tienen el mismo código variando solo el tema de la cámara del robot que se suscribe.

Tanto *move_base_robot*, *nuevas_misiones*, *image_converter_1*, *image_converter_2* y *image_converter_3* son ejecutadas en un mismo launch llamado *nav_robot*, el cual se encuentra en el Anexo F.

Por una parte, el nodo *move_base_robot* irá sobreescribiendo información en *posicion_robot_1* o *posicion_robot_2* o *posicion_robot_3*, dependiendo del robot que esté controlando. Esa información necesaria para la planificación contendrá la posición y batería del robot, distancia al punto que se dirige y distancia al punto de carga más cercano del objetivo.

Por otra parte, el nodo *nuevas_misiones* se encargará de leer el documento *misiones.txt* cada vez que se sobreescriba en él. La información extraida por el nodo serán los nuevos objetivos de cada robot, los cuales serán enviados mediante temas al robot correspondiente.

Ese mismo tema que envió *nuevas_misiones* lo recibirá *move_base_robot*, el cual lo guardará en una matriz que actua como una pila. Esta pila es la que irá acumulando las nuevas misiones y cada vez que el robot termine de cumplir un objetivo, leerá la siguiente misión de la pila a la que tendrá que ir.

En caso de que el robot no tenga suficiente batería para realizar la misión, se dirigirá al punto de carga más cercano para recargar dejando en espera las misiones que tiene asignadas. Una vez que se haya cargado por completo, seguirá realizando su recorrido.

Debido a que los robots son iguales, el código para controlarlos también lo son, excepto por el nombre del nodo y a los temas que pública y suscribe. A continuación se explicará los nodos antes mencionados:

a) Move_base_robot:

Este nodo se compone de tres funciones: main, counterCallback1, counterCallback2.

- Main:

  Esta es la función principal que será la primera que se ejecute y la que llamará a las otras funciones. En primer lugar inicializa el nodo y luego notifica a ROS sobre que temas publicar y suscribir, de los cuales son: *action_pub1* (mueve el robot), *sub1* (obtiene las coordenadas del robot) *y sub2* (lee los nuevos objetivos). La variable *sub1* será la que llame a la función *counterCallback1* cada vez que reciba un mensaje del tema suscrito, mientras que *sub2* realiza lo mismo pero con otro tema llamando *counterCallback2*. A continuación, abre un documento txt llamado *posición_robot1* o *posición_robot2* o *posición_robot3* (dependiendo del robot que esté controlando) donde sobreescribirá las posiciones del robot, porcentaje de su batería y la variable *recarga*, la cual informa si el robot está recargando. Por último, entra en un bucle while donde quedará atrapado hasta que el usuario decida detenerlo.

  Los dos temas a los que se suscribe este nodo son: */Robot1/odom* y */array*, los cuales pertenecen a las funciones *counterCallback1* y *counterCallback2* respectivamente.

- CounterCallback1:

  Esta función se encarga principalmente de controlar el robot en cuestión. Para ello, tiene una variable llamada *time* que se actualiza constantemente para poder controlar los tiempos de parada y recarga. De forma que cada vez que el robot llega al punto objetivo o punto de recarga actualizará las variables *parada* o *cargar* respectivamente, y así no pueda recibir ninguna publicación hasta que no haya transcurrido el tiempo de parada o recarga.

  Luego, pregunta si el robot tiene un nuevo objetivo y no está parado cumpliendo una misión ni recargando; si es así, estima el consumo de la batería calculando la distancia d_{AB} y d_{BC}, siendo A la posición del robot, B la coordenada del punto objetivo y C la posición del punto de recarga más cercano al objetivo, tal y como se muestra en la Figura 5.2. En caso de tener suficiente energía, guarda la coordenada de la nueva misión, de lo contrario guarda la coordenada del punto de recarga más cercano al robot y coloca 1 a la variable *recarga* para indicar que el robot va a recargar.

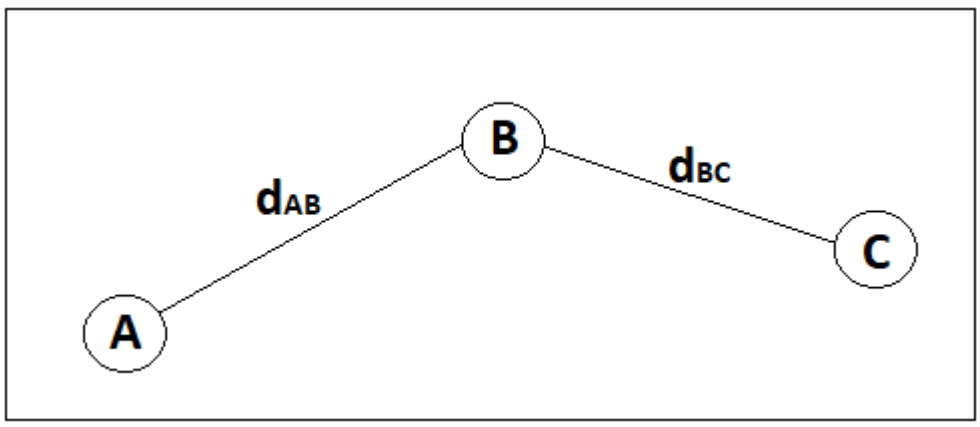

Figura 5.2 Distancias entre el robot al punto objetivo y el punto objetivo al punto de recarga.

Inicialmente para mover el robot se publica la coordenada del objetivo en *action_pub1* dos veces para asegurar que lo reciba correctamente. Una vez que se mueve el robot, cada 5 segundos actualiza *posición_robot*, reduce la batería consumida en ese periodo y en caso de que haya pasado 10 segundos parado a mitad del camino vuelve a publicar en *action_pub1* el objetivo para mover el robot de nuevo.

En caso de llegar al punto objetivo, publica en *action_pub2* para detener el robot, actualiza la variable *parada*, reduzco la batería que conllevaría estar parado captando la radiación solar e incrementa la variable *fila* para leer el siguiente objetivo.

Si por el contrario el robot llega al punto de recarga, publica en *action_pub2* para detener el robot, actualiza la variable *cargar*, modifica *batería_inicial* simulando que la batería vuelve a estar 100% y coloca 0 a la variable *recarga* para indicar que el robot está disponible para cumplir nuevos objetivos.

- CounterCallback2:

  Esta función se encarga de almacenar los nuevos objetivos en la pila *matrix1* que le manda *nuevas_misiones*, de forma que cada vez que el robot cumpla una misión, vaya leyendo esta matriz en el mismo orden que va llegando los objetivos.

b) Nuevas_misiones:

Este nodo se compone de tres funciones: main, lectura y escritura:

- Main:

  Esta función es la principal y primera en ejecutarse. En primer lugar notifica a ROS los temas que se va a publicar, de los cuales son: *pub*, *pub2* y *pub3*. Cada uno es un tema por donde se mandará los objetivos al robot correspondiente.

  Luego entra en un bucle while donde llama la función *lectura*. Seguidamente comprueba si los vectores *m1* o *m2* o *m3*, los cuales almacena las coordenadas de los objetivos leídos en la función lectura del robot correspondiente, han sido modificados. En caso de ser así, se guarda la coordenada en una variable tipo mensaje, se publica en el tema correspondiente antes mencionado y se coloca 1 a la variable *bandera1* o *bandera2* o *bandera3* indicando que ya se ha enviado el mensaje del robot en cuestión.

- Lectura:

  Esta función se encarga como bien indica su nombre de leer un documento llamado *misiones.txt*, donde Matlab irá sobreescribiendo los nuevos objetivos de cada robot. En primer lugar, se abre el documento en cuestión y luego se inicializa los vectores *m1*, *m2* y *m3*. Seguidamente se entra en un bucle while donde se guarda todo el contenido del documento en una cadena llamado *texto*. Dado que el código de este bucle se ejecuta dos veces, en caso de ser la primera vez que se entra y *texto* no esté vacio, se procede a recoger toda la información.

  Para ello, se entra en un bucle for del cual no se sale hasta que se haya obtenido todos los objetivos incrementando el índice *a* para leer carácter a caracter. Una vez dentro, se guarda la primera coordenada en una cadena hasta que se encuentre un espacio. Luego se transforma a float y se guarda en *m1*. A continuación se hace lo mismo para la siguiente coordenada del primer robot. Una vez que se haya completado el vector *m1*, se incrementa la variable *a* 3 veces para comenzar a extraer la primera coordenada de *m2*. El proceso sería el mismo para el resto de vectores. En la Figura 5.3 se muestra un ejemplo del documento *misiones.txt*:

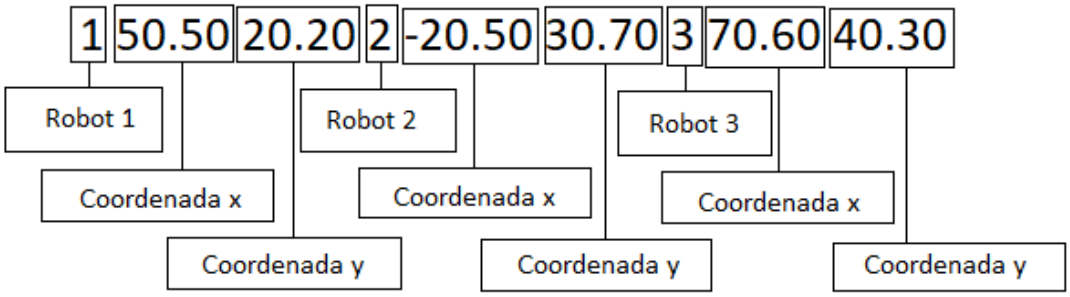

Figura 5.3 Ejemplo del documento *misiones.txt*.

Luego se llama a la función *escritura* para vaciar el documento y se inicializa las variables *bandera1, bandera2* y *bandera3*. Una vez realizado todo este proceso y fuera del bucle while, se procede a limpiar todas las variables y cerrar el documento.

- Escritura:

  Esta función solamente se encarga de vaciar *misiones.txt*. Para ello, primero se abre el documento y luego se sobreescribe un espacio vacio borrando todo lo anterior. Por último, se cierra el archivo.

A continuación en la Figura 5.4 se muestra un diagrama de flujo del código de *move_base_robot* para que resulte más fácil y visible la lógica del código.

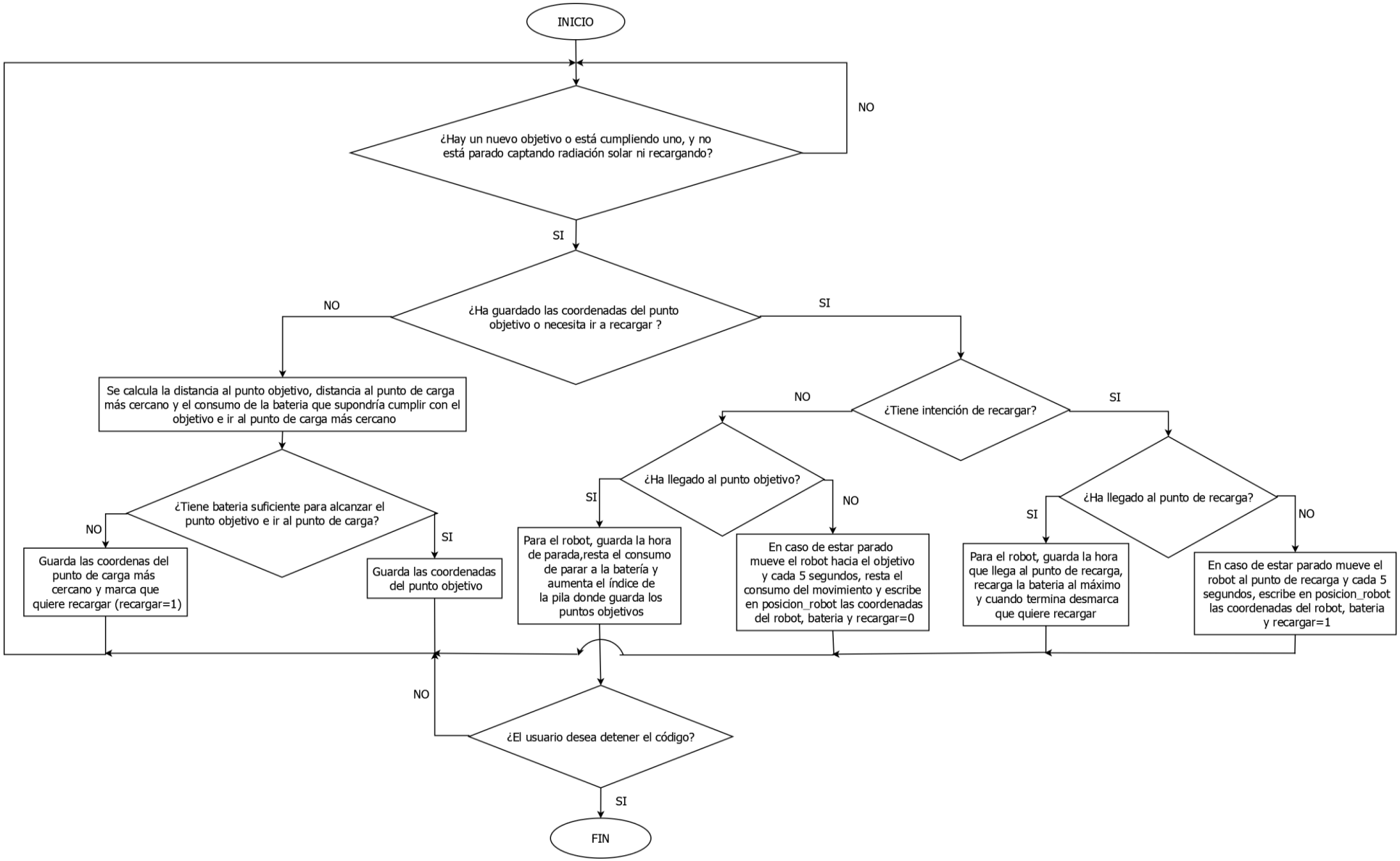

Figura 5.4 Diagrama de flujo del código de *move_base_robot*.

5.3 Programación con Matlab

Por otra parte también se ha programado en Matlab debido a que contiene funciones muy útiles para planificar la gestión de la flota de robots. A continuación, se va a explicar con detalle la programación realizada en este programa.

Cuando el usuario escriba los diferentes puntos que quiera que vayan los robots, el código de Matlab lee los documentos *posición_robot_1, posición_robot_2* y *posición_robot_3* antes mencionados, de los cuales se muestra un ejemplo en la Figura 5.5, para guardar la información en variables, las cuales se introducirán en la función que realizará la planificación. Esta función devolverá dos matrices, de las cuales una asigna una misión a cada robot y la otra guarda las posiciones que no se han asignado a ningún robot.

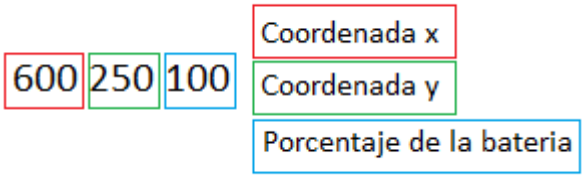

Figura 5.5 Ejemplo de posición_robot1.

Las misiones sin asignar se guardarán para la siguiente ocasión que el usuario escriba nuevos objetivos, mientras que los que han sido asignados se sobreescribe en *misiones.txt*, el cual será leído por el nodo *nuevas_misiones*.

Se ha creado tres scripts de Matlab para realizar esta parte:

a) Inicializa:

Este script solo se ejecuta al comienzo. Se encarga de crear una matriz que contiene las posiciones del punto de recarga, un vector vacio de las posiciones no asignadas y activar el cronómetro, el cual permitirá controlar el tiempo que lleva un objetivo sin asignar para darle prioridad. Se encuentra en el Anexo J.

b) Nuevos_objetivos:

Es la función que ejecuta el usuario cuando quiere introducir nuevas misiones. Esta función recibe los nuevos objetivos introducidos por el ususario, los cuales son guardados en la matriz *objetivos_nuevos* como se puede observar en la Figura 5.6, la matriz de las posiciones de los puntos de recarga y el vector de las posiciones no asignadas. Se encarga de leer el documento txt que contiene información de los robots, luego ejecuta *funcion_de_coste*, la cual devolverá dos matrices: una con un objetivo asignado a cada robot y la otra los objetivos no asignados. Y por último, escribe en *misiones.txt* la misión asignada a cada robot. Se encuentra en el Anexo K.

Figura 5.6 Ejemplo de objetivos_nuevos.

c) Función_de_coste:

Esta función recibe las baterías y posiciones en las que se encuentran cada robot. También los puntos de carga y las posiciones no asignadas. Una vez que se haya agrupado toda la información se ejecuta el comando *intlinprog*, el cual es un programador lineal de enteros mixtos que planifica las misiones de cada robot. Se encuentra en el Anexo L.
Después los datos que *intlinprog* devuelve, se guardan en dos matrices, las cuales son:

- **New missions**: contiene las misiones que se le asigna a cada robot, como se puede observar en la Figura 5.7.

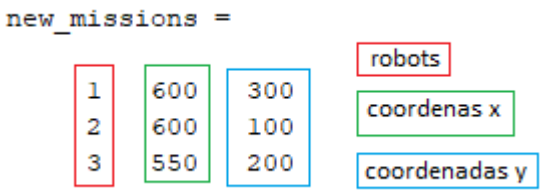

Figura 5.7 Ejemplo de new_missions.

- **Objetive_spots_not_assigned**: contiene las misiones que no se han asignado a ningún robot y el tiempo que llevan sin ser asignadas, como se puede observar en la Figura 5.8.

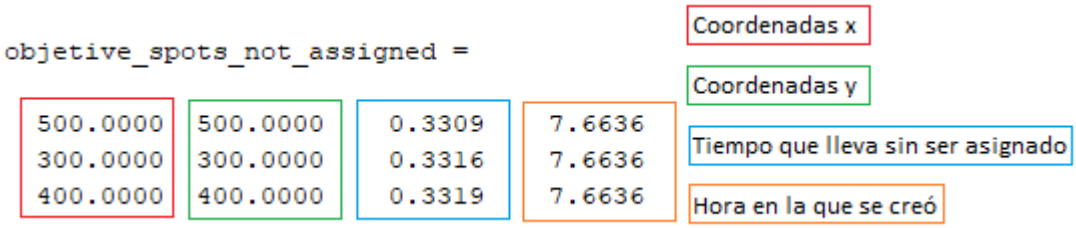

Figura 5.8 Ejemplo de objetive_spots_not_assigned.

Luego estas dos matrices será lo que devolverá la *funcion_de_costes*.

5.4 Estimación de la batería del robot

Dado que se está utilizando un simulador, el cual no publica un tema que te muestre la batería del robot, hay que estimarla. Para ello, se ha buscado características del robot real como por ejemplo: el tiempo que dura la batería estando parado o en movimiento el robot y la velocidad máxima que alcanza.

El robot con el que se trabajaría físicamente sería Rosbot 2.0, el cual se alimenta de un paquete de baterías de iones de litio recargables internos que contiene 3 celdas conectadas en serie. Este tipo de conexión se llama "3S". Dado que la capacidad es de 3500 mAh y el voltaje nominal de cada batería es de 3.7V, se podría calcular la energía máxima con la siguiente ecuación:

$$W_b = 3 * 3.7 * 3500 = 38.85\ Wh \qquad 5.1$$

siendo W_b la energía máxima del paquete de batería que tiene el robot.

En vista de que según el manual del robot la batería dura entre 1.5-5 horas, se ha realizado las siguientes consideraciones:

- El robot es capaz de moverse sin parar durante 1.5 horas.
- El robot es capaz de estar parado estando encendido durante 5 horas.

En consecuencia de estas suposiciones se puede estimar el consumo de la batería cuando está en movimiento y cuando está parado captando radiación solar.

En caso de estar parado, su consumo sería:

$$t_p = 5\ horas * \frac{60\ minutos}{1\ hora} = 300\ minutos$$

5.2

$$W_{cp} = \frac{W_b}{t_p} = \frac{38.85\ Wh}{300\ minutos} = 0.13\ {}^{Wh}/_{min}$$

siendo $t_p$el tiempo máximo que puede estar parado el robot y W_{cp} la energía que consume el robot por minuto.

En caso de estar en movimiento suponiendo que se mueve con su velocidad máxima, la cual sería 1.25 m/s, su consumo sería:

$$d_m = 1.5\ horas * \frac{3600\ segundos}{1\ hora} * \frac{1.25\ metros}{1\ segundo} = 6750\ metros = 6.75\ km$$

5.3

$$W_{cm} = \frac{W_b}{d_m} = \frac{38.85\ Wh}{6.75\ km} = 5.756\ {}^{Wh}/_{km}$$

siendo d_m la distancia máxima que puede recorrer el robot a velocidad máxima y W_{cm} la energía que consume el robot por kilometro recorrido.

Una vez obtenido todos estos parámetros se puede proceder a obtener la fórmula que permite estimar el consumo total que conllevaría controlar el robot.

$$W_{CT} = W_{cp} * t + W_{cm} * d = 0.13 * t + 5.756 * d \qquad 5.4$$

siendo W_{CT} el consumo total del robot, *t* el tiempo que está parado el robot y *d* la distancia que recorre el robot.

En caso de querer saber si tiene suficiente batería para cumplir un nuevo objetivo, se calcularía las distancias d_{AB} y d_{BC} (véase la Figura 5.2) y sabiendo el tiempo que está el robot parado captando la radiación solar, se podría perfectamente estimar el consumo que le conllevaría realizar todo ese recorrido.

Además se podría calcular el porcentaje de batería que le queda al robot con la siguiente fórmula:

$$\eta_b = \frac{W_b - W_{CT}}{W_b} * 100 = \frac{38.85 - W_{CT}}{38.85} * 100 \qquad 5.5$$

siendo η_b el rendimiento del porcentaje que le queda a la batería.

Para ensayar en el simulador y comprobar que funciona todo correctamente, se ha reducido los valores de las constantes W_{cp} y W_{cm} reduciendo así el tiempo que dura la batería. Al igual que el tiempo de carga, la cual dura 3 horas, se ha reducido a 10 segundos y el tiempo que está captando radiación solar a 10 segundos.

6 RESULTADOS

Una vez que se ha hecho una estimación de la batería y programado todos los códigos necesarios, como pueden ser: los tres launchs para ejecutar el mundo en Gazebo, los tres nodos encargados de controlar los robots capaces de moverse evitando cualquier obstáculos, otro nodo llamado *nuevas_misiones* para enviar los objetivos a los robots y por último, los tres códigos de Matlab para planificar los objetivos especificados por el usuario.

De esta manera, habría un algoritmo que se encarga de planificar todos los objetivos requeridos por el usuario mandando un objetivo a cada robot y dejando en espera el resto de misiones. A continuación, los robots se desplazan para alcanzar cualquier punto del campo solar evitando cualquier obstáculo que pudieran encontrarse. Además en caso de contar con poca batería, ellos mismos son capaces de dirigirse al punto de recarga más cercano dejando en espera los objetivos que les quedan por cumplir. Después de recargar continuarían cumpliendo sus misiones por todo el campo solar.

A continuación, se procede a mostrar los resultados obtenidos. Para ello, se han realizado varias fotografías con el fin de mostrar las trayectorias de los tres robots.

A continuación se va hacer una breve explicación de lo que muestra en las Figuras 6.2 a 6.16:

- Se escribe por Matlab 4 nuevos objetivos, de los cuales tres son asignandos y uno no.
- Cada robot se dirige a su punto objetivo.
- Cuando dos de los robots llegan a su destino, se vuelve a escribir en Matlab dos nuevos objetivos.
- Dado que anteriormente un objetivo no fue asignado, junto con los dos nuevos objetivos se le asigna uno a cada robot.
- Dos de ellos se dirigen a su punto objetivo mientras que otro (Robot 1) necesita ir al punto de recarga.
- Una vez que ha llegado y recargado, se dirige a su objetivo que se le asignó y no lo cumplió.

A continuación en la Figura 6.1 se muestra un plano alzado del campo solar.

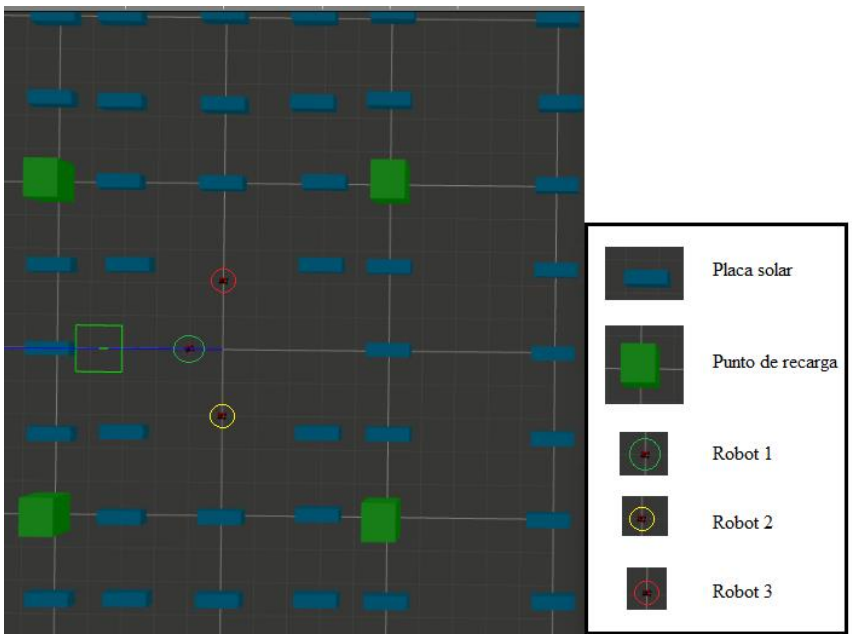

Figura 6.1 Vista alzada del parque solar viasualizado en Gazebo.

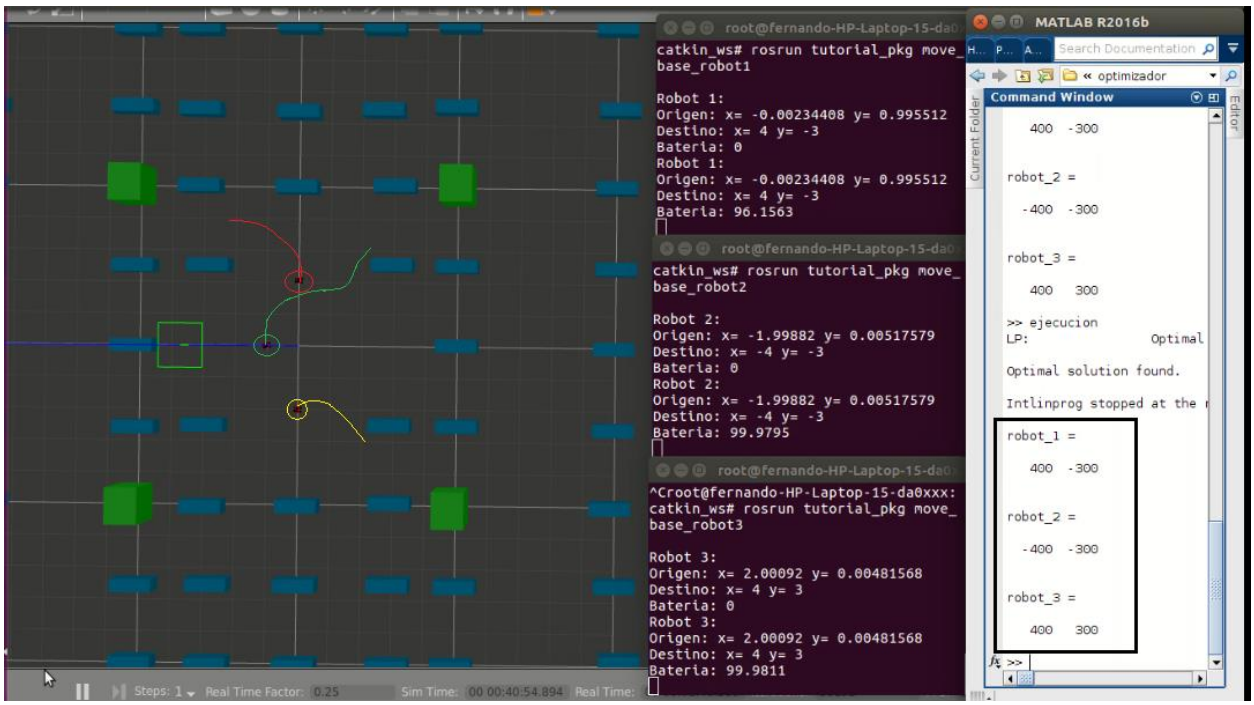

Figura 6.2 Trazado de trayectorias de tres robots. Se ordena 4 nuevos objetivos e inicia el movimiento.

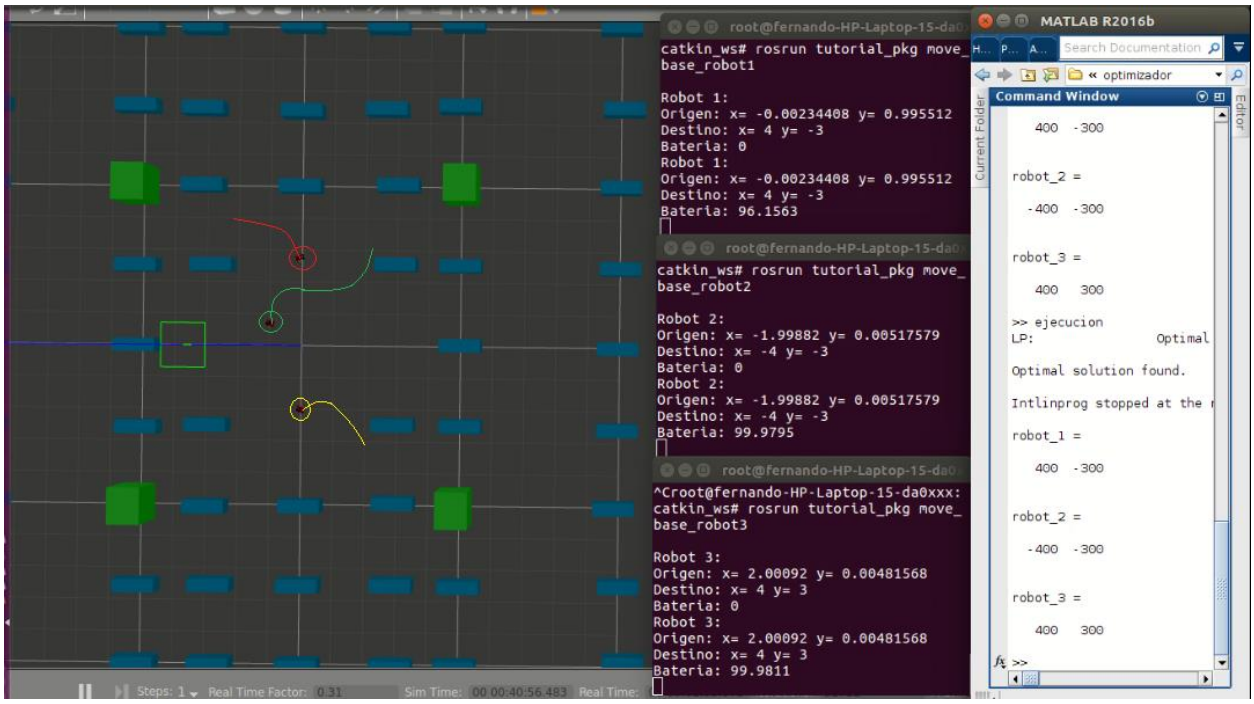

Figura 6.3 Trazado de trayectorias de tres robots. Pequeño desplazamiento (1).

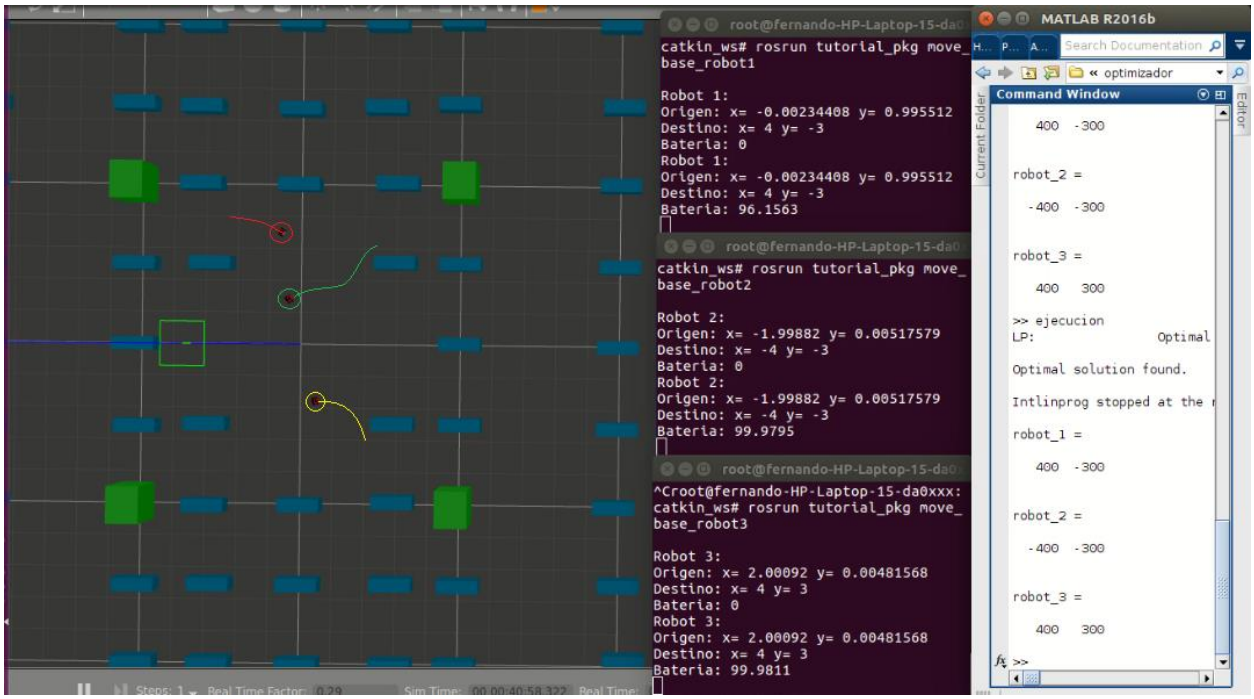

Figura 6.4 Trazado de trayectorias de tres robots. Pequeño desplazamiento (2).

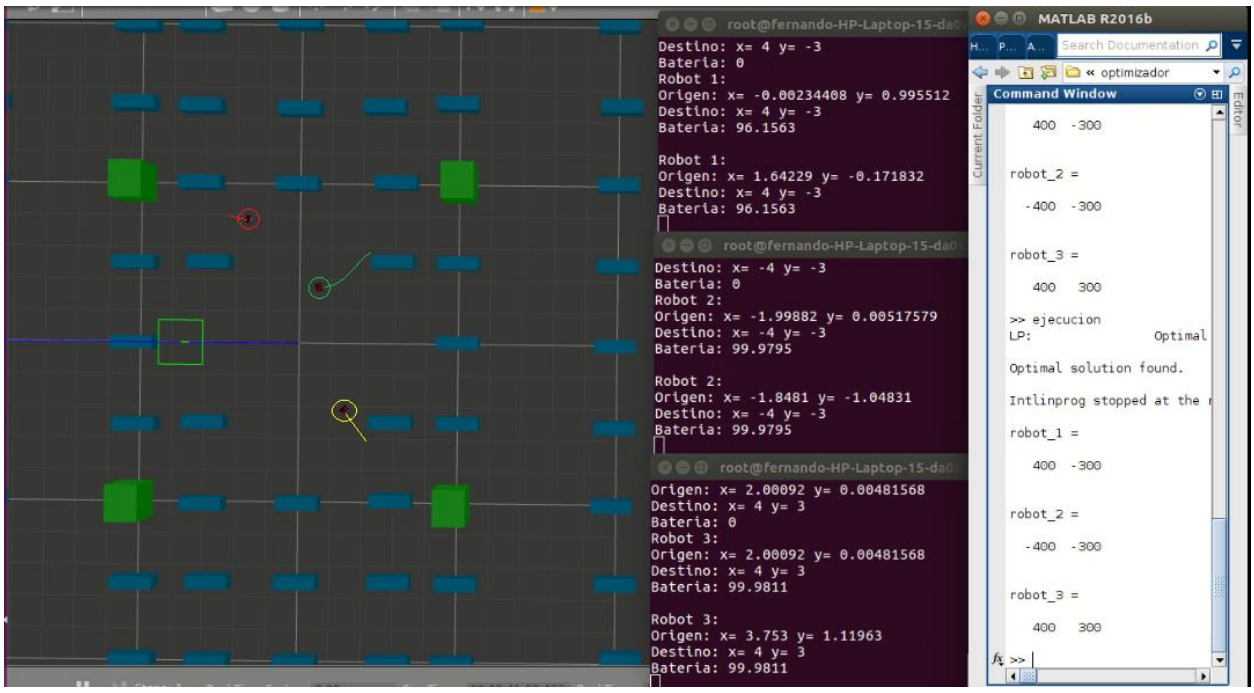

Figura 6.5 Trazado de trayectorias de tres robots. Pequeño desplazamiento (3).

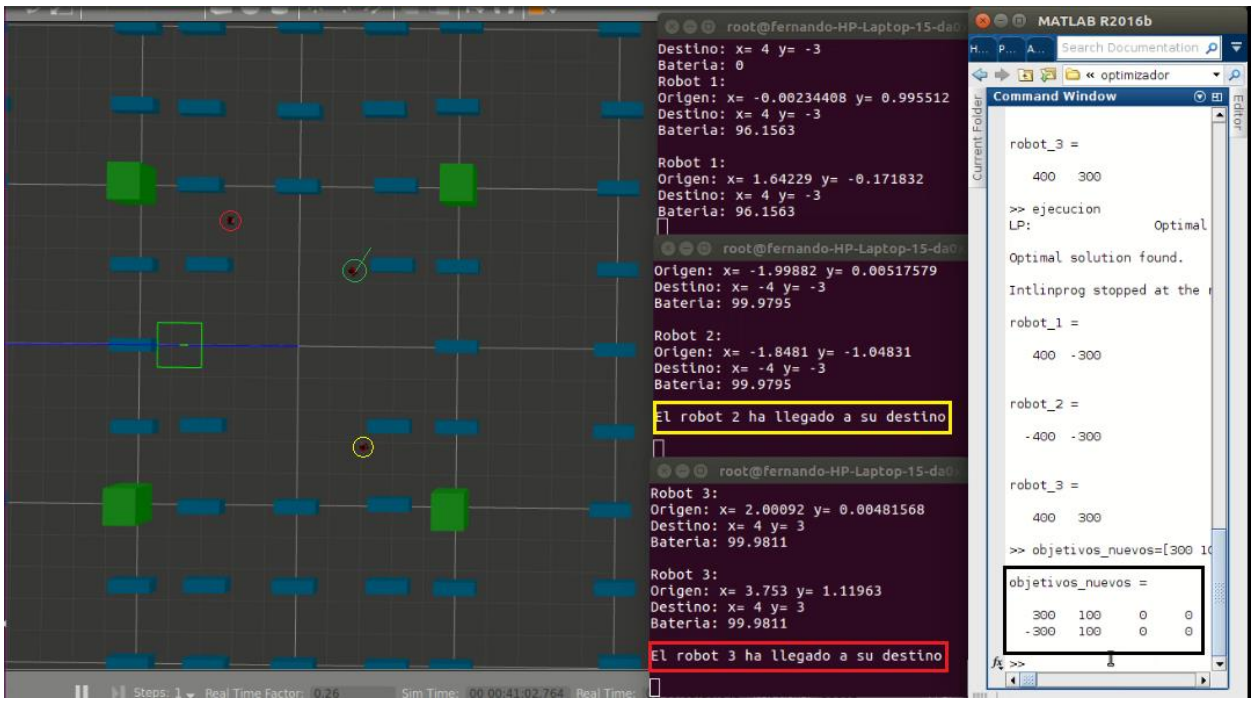

Figura 6.6 Trazado de trayectorias de tres robots. Los robots 2 y 3 han llegado a su destino, y además se ordena 2 nuevos objetivos.

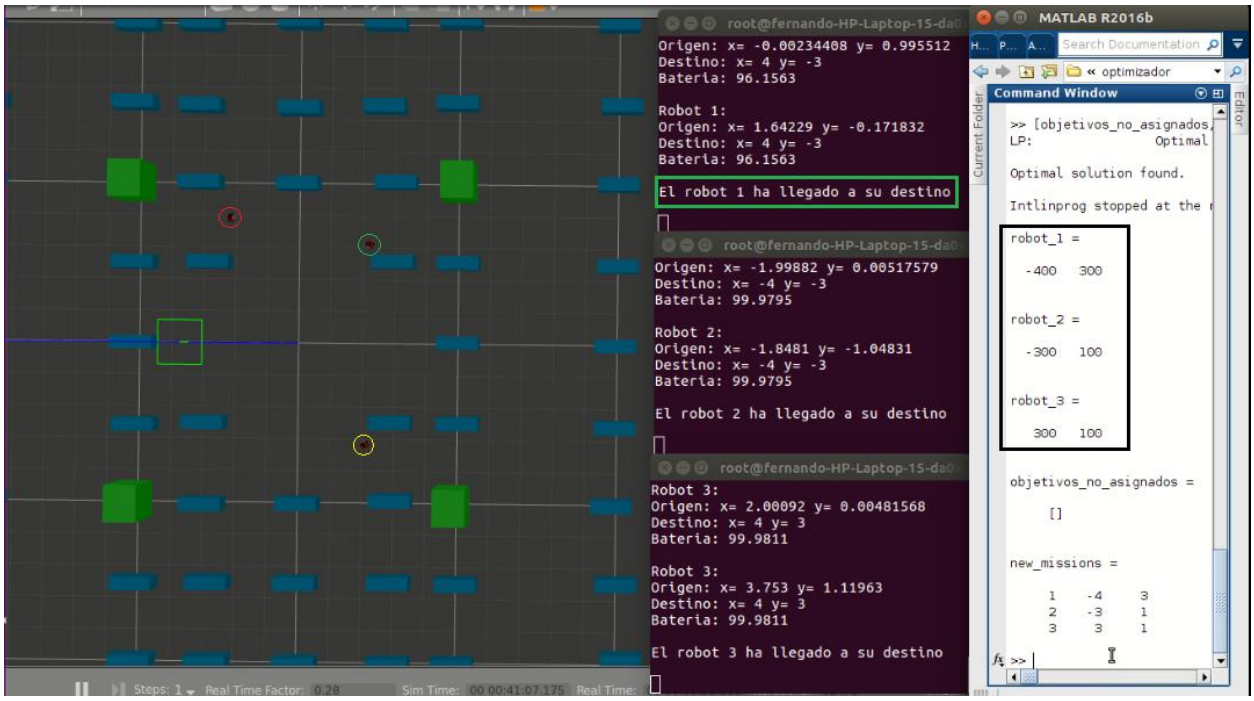

Figura 6.7 Trazado de trayectorias de tres robots. El robot 1 ha llegado a su destino.

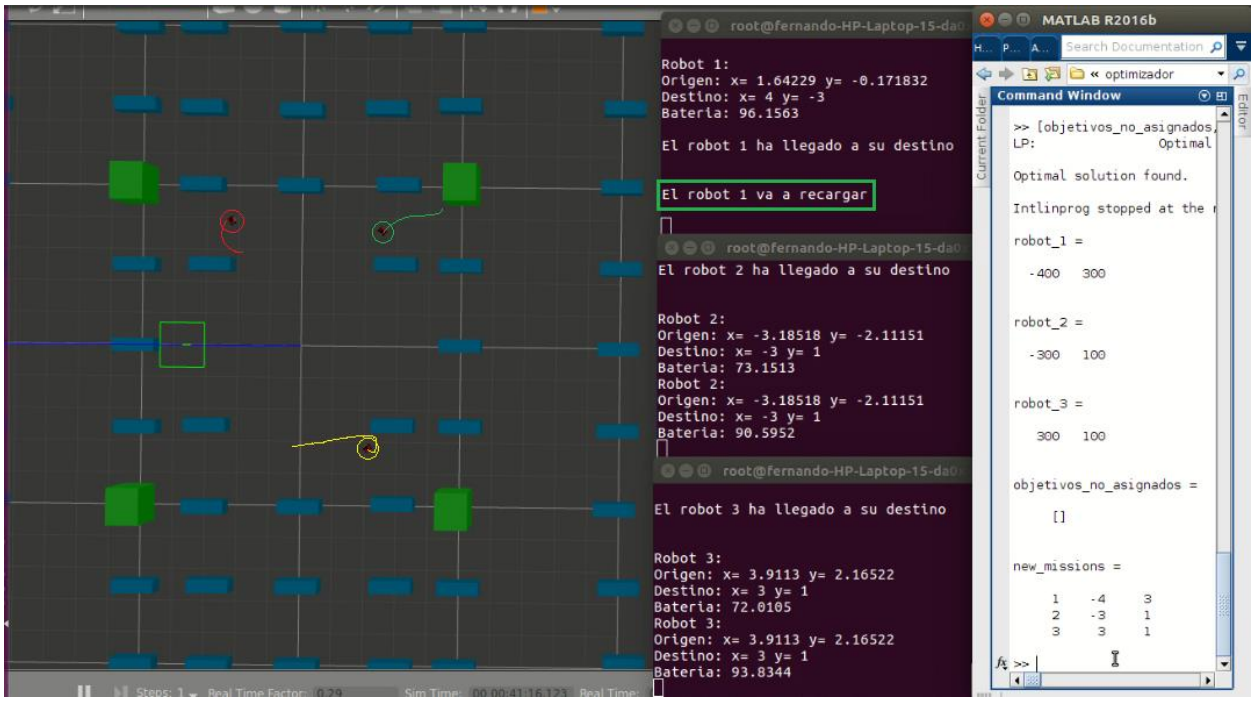

Figura 6.8 Trazado de trayectorias de tres robots. Inicio del movimiento donde los robots 2 y 3 se dirigen a su objetivo mientras que el robot 1 va al punto de recarga.

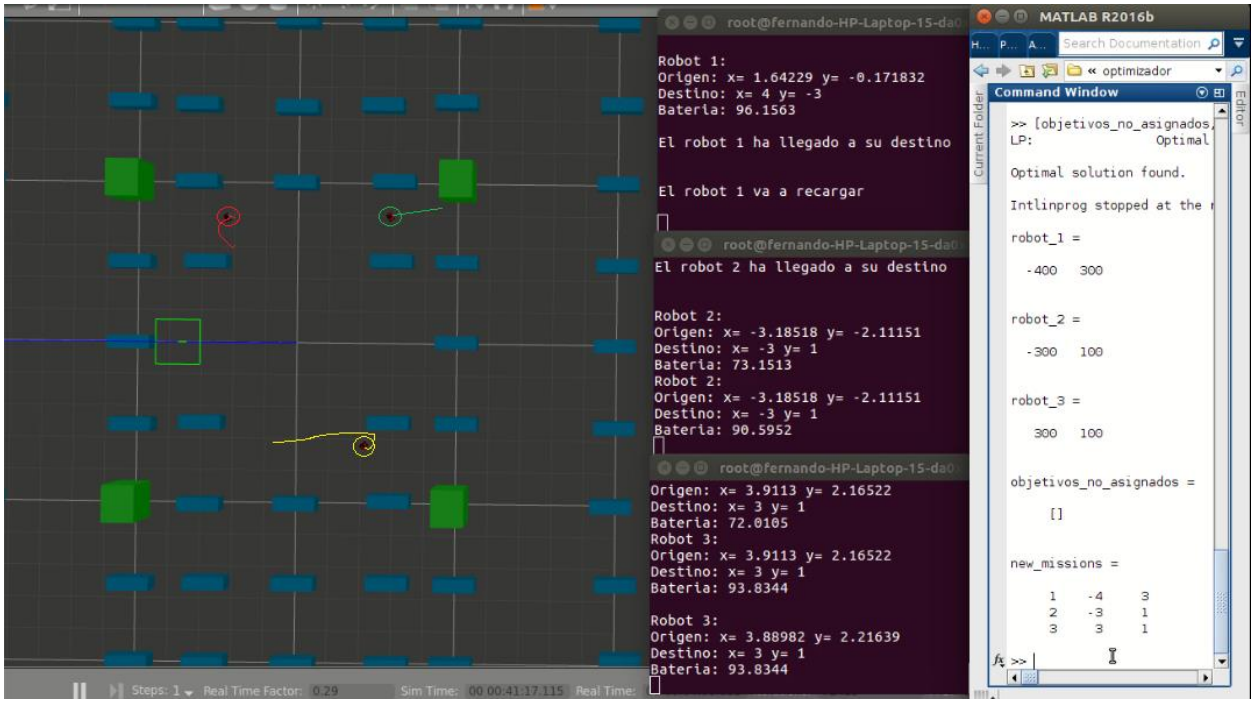

Figura 6.9 Trazado de trayectorias de tres robots. Pequeño desplazamiento.

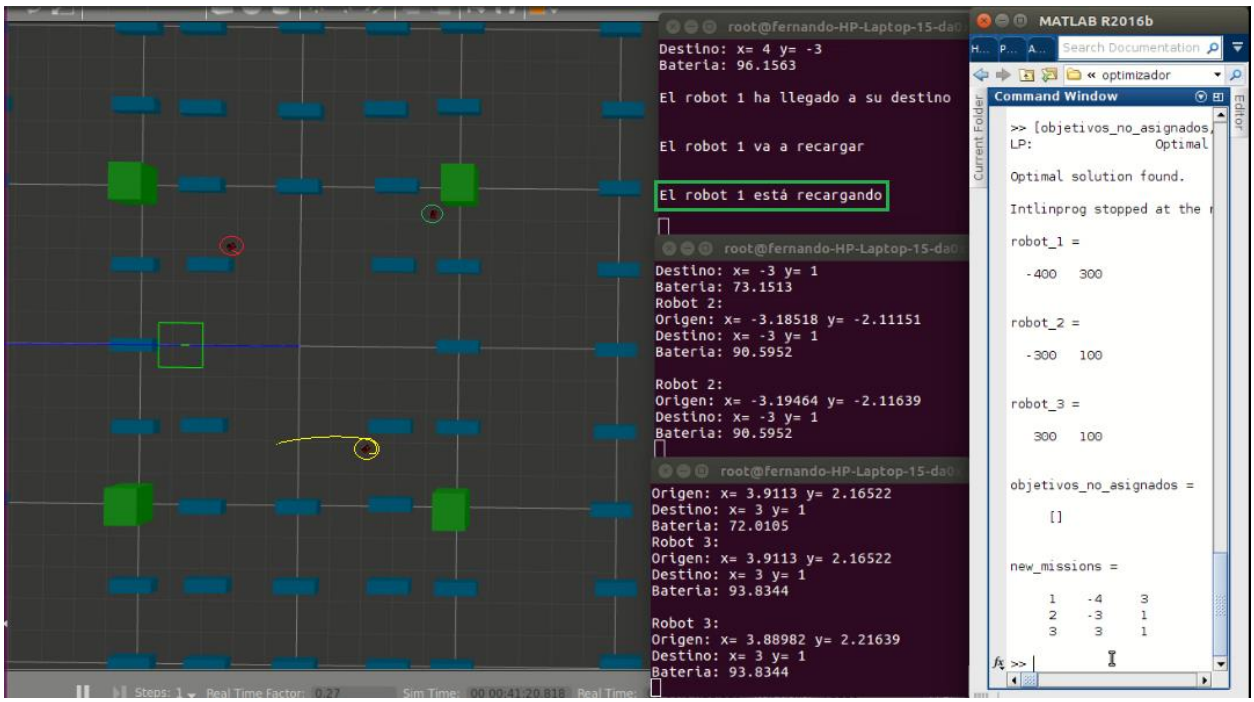

Figura 6.10 Trazado de trayectorias de tres robots. El robot 1 ha llegado al punto de recarga.

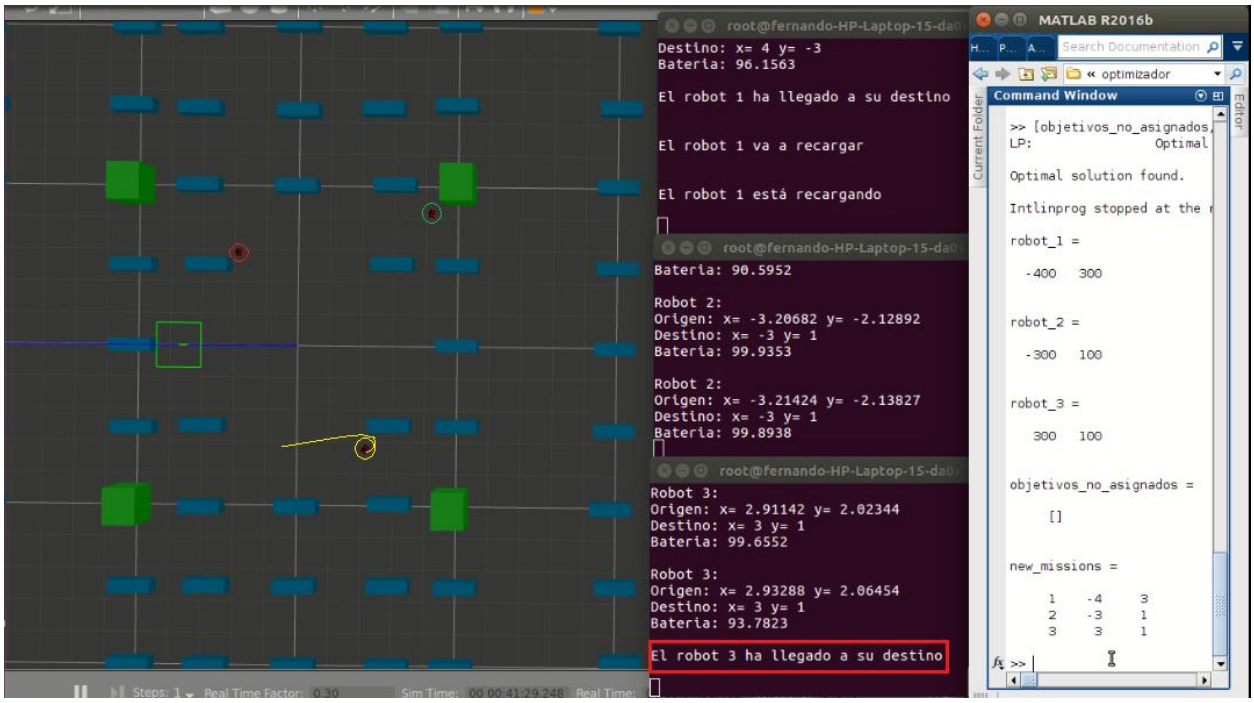

Figura 6.11 Trazado de trayectorias de tres robots. El robot 3 ha llegado a su destino.

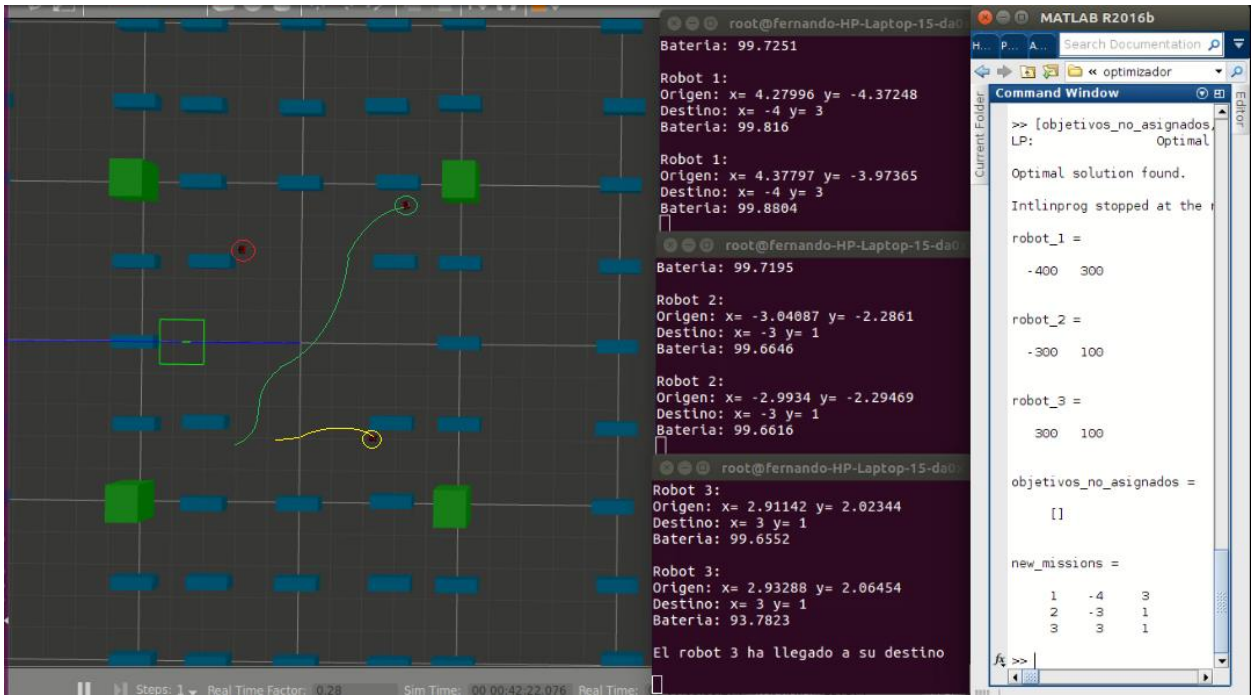

Figura 6.12 Trazado de trayectorias de tres robots. El robot 1 se dirige a su objetivo mientras que el robot 2 está en camino.

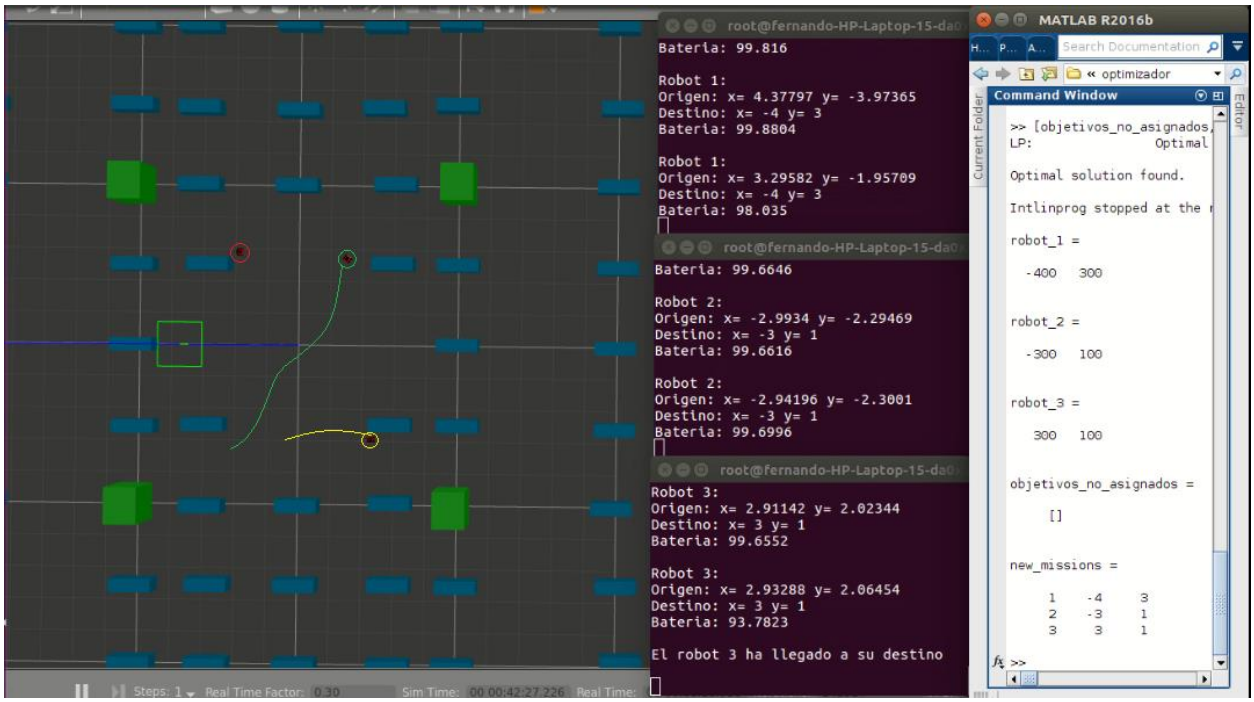

Figura 6.13 Trazado de trayectorias de tres robots. Pequeño desplazamiento (1).

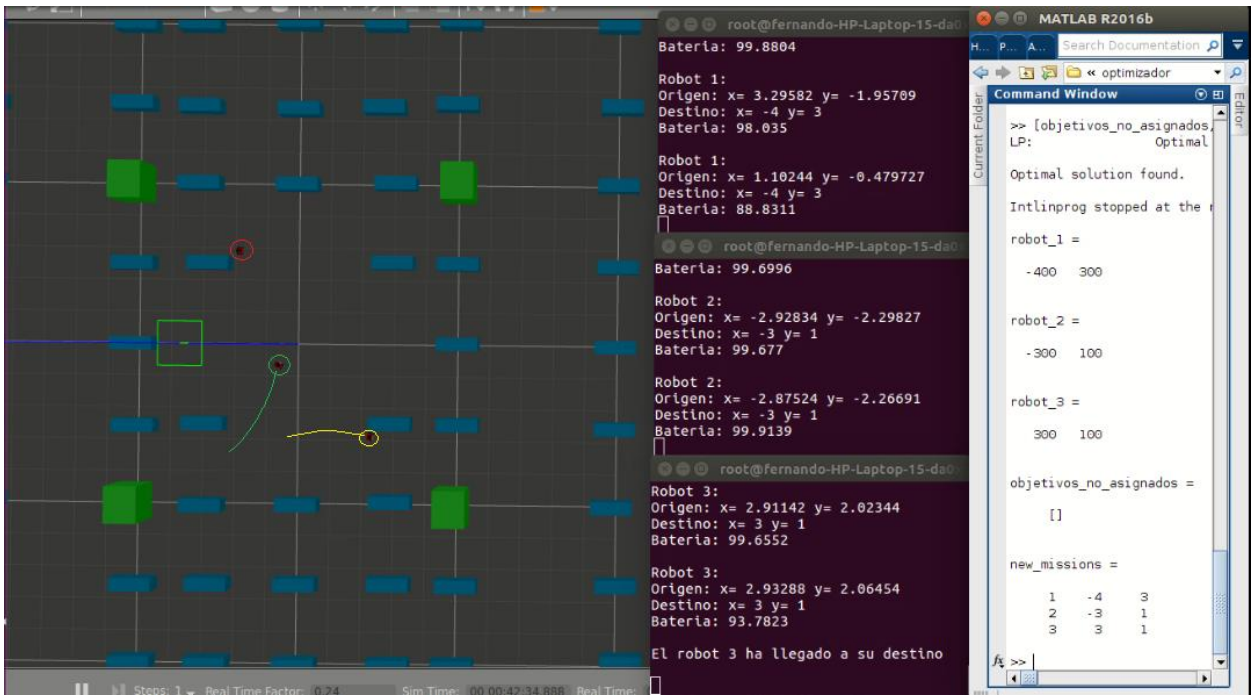

Figura 6.14 Trazado de trayectorias de tres robots. Pequeño desplazamiento (2).

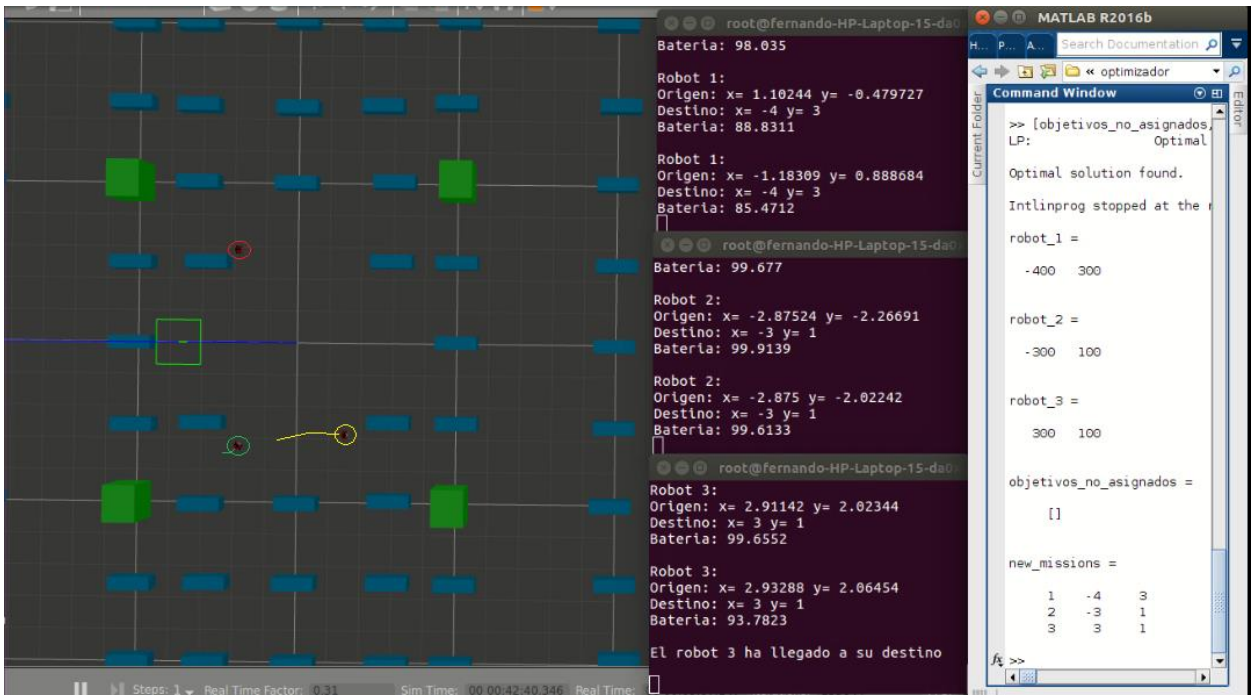

Figura 6.15 Trazado de trayectorias de tres robots. Pequeño desplazamiento (3).

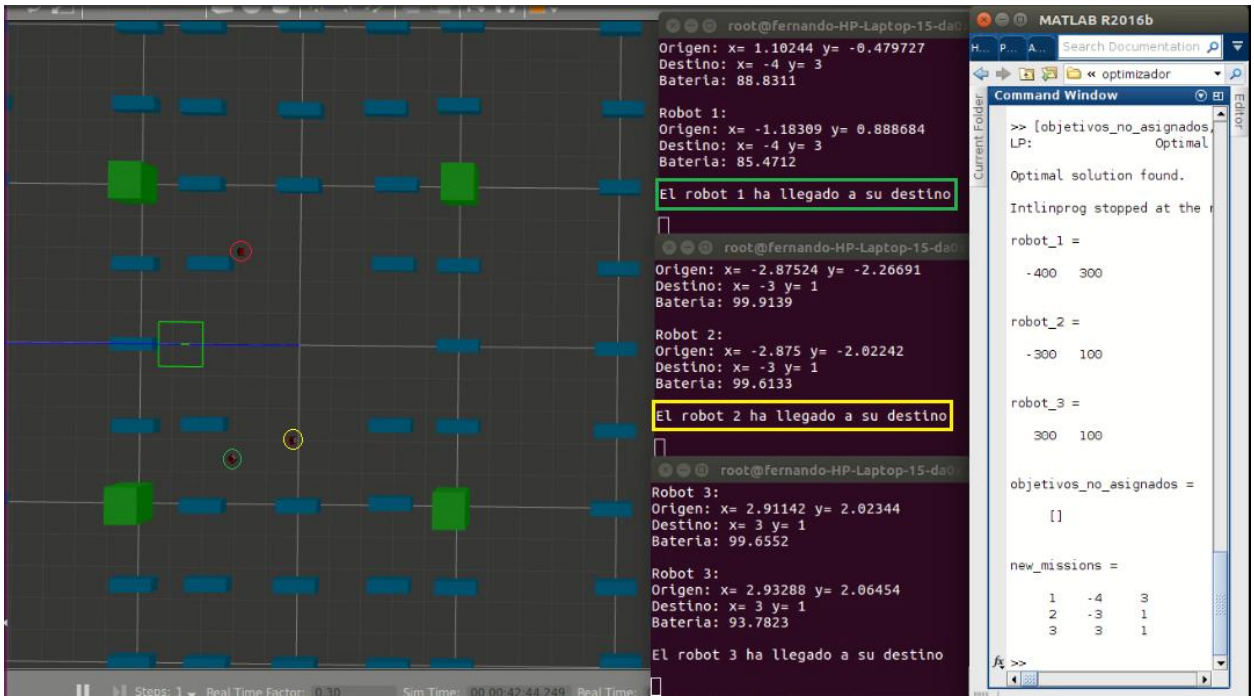

Figura 6.16 Trazado de trayectorias de tres robots. Los robots 1 y 2 han llegado a su destino.

7 POSIBLES AMPLIACIONES

En este proyecto podría haberse realizado con robots físicamente reales en vez de utilizar un simulador como Gazebo. La razón por la que no se ha podido implementar ha sido por la aparición de la pandemia del Covid-19, la cual me ha imposibilitado poder trabajar con los robots.

A continuación se explica cómo se habría hecho:

Cada robot tiene un microprocesador el cual se le puede conectar una pantalla y un teclado y funcionaría como un ordenador. Una vez encendido el robot, aparecería en la pantalla el escritorio del sistema operativo Ubuntu donde se podría crear un espacio de trabajo al igual que se ha hecho para el simulador. Luego se crearían varios archivos que actuarían como nodos para controlar el robot. Además habría que modificar el nombre de los temas puesto que no serían los mismos que del simulador. También no haría falta estimar la batería del robot ya que el propio robot tiene un tema que te lo puede indicar. Esto se haría con cada robot individualmente.

A continuación, se adjudicaría como maestro a un ordenador, donde permitiría controlar todos los robots por via wifi. Además se escribiría todas las misiones que desearía el usuario.

Por último, se utilizaría el campo de futbol de la ETSI como zona de prueba de la gestión de la flota de robots la cual simularía un campo solar.

8 CONCLUSIÓN

En este proyecto se ha realizado la programación para la planificación de la gestión de una flota de robots autónomos capaces de recorrer un campo solar sin colisionarse entre ellos ni con otros obstáculos. A la vez que se reparten los puntos que tienen que medir la radiación solar mediante un optimizador y en caso de quedarse sin baterías, ellos mismos se dirigen al punto de carga más cercano para recargar.

Con todo lo desarrollado en este trabajo y de haber alcanzado al objetivo final, se ha tenido que lidiar con varias adversidades, como puede ser el problema de la estimación de la batería de los robots de forma que se pareciera lo máximo posible al gasto de las baterías. Así mismo, la comunicación entre ROS y Matlab, el cual ha resultado una tarea compleja para conseguir la coordinación entre los dos programas. Otro contratiempo que se tuvo fue conseguir que los robots llegasen a la posición objetivo, el cual resultó una tarea más compleja de lo esperado puesto que el propio Gazebo daba problemas a la hora de actualizar el mapa durante la ejecución.

Este trabajo fin de grado me ha resultado un reto personal debido a que en numerosas ocasiones he tenido situaciones desfavorables, las cuales he tenido que resolver por mi cuenta, aunque siempre con la inestimable ayuda de mi tutor y otros colaboradores. Además, considero que, a pesar de todos los conocimientos adquiridos en el grado, ha sido fundamental también la organización del tiempo y la disciplina para llevar a cabo el trabajo junto con otras asignaturas del curso, lo cual incrementaba aún más la dificultad. También el hecho de haber realizado el trabajo durante el estado de alarma por la pandemia, me ha ayudado a percatarme que en todo proyecto siempre aparecerán situaciones adversas impredecibles y fuera de nuestro control.

Por último, pongo a disposición todos los archivos, que han sido utilizados para la realización de este trabajo fin de grado, al departamento de Ingeniería de sistemas y automática de la ETSI de la Universidad de Sevilla para su utilización en futuros proyectos.

REFERENCIAS

[1] Lexico, «robot ,» 2020 [En línea]. Available: https://www.lexico.com/es/definicion/robot [Último acceso: Agosto 2020]

[2] Wikipedia, «Robot ,» 2020 [En línea]. Available: https://es.wikipedia.org/wiki/Robot [Último acceso: Agosto 2020]

[3] Wikipedia, «Robot autónomo ,» 2020 [En línea]. Available: https://es.wikipedia.org/wiki/Robot_aut%C3%B3nomo [Último acceso: Agosto 2020]

[4] European dissemination media agency, «Proyecto OCONTSOLAR- Control óptimo de sistemas de energía solar térmica ,» 19 de julio de 2018 [En línea]. Available: https://www.europeandissemination.eu/ocontsolar-project/2563 [Último acceso: Octubre 2020]

[5] Wikipedia, «ROS (Robots Operating System),» 2020 [En línea]. Available: https://es.wikipedia.org/wiki/Sistema_Operativo_Rob%C3%B3tico [Último acceso: Octubre 2020]

[6] ROS.org, «ROS/concepts ,» 2020 [En línea]. Available: http://wiki.ros.org/ROS/Concepts [Último acceso: Octubre 2020]

[7] ROS.org, «roscore» 2020 [En línea]. Available: http://wiki.ros.org/roscore [Último acceso: Agosto 2020]

[8] ROS.org, «rosbash» 2020 [En línea]. Available: http://wiki.ros.org/rosbash#rosrun [Último acceso: Agosto 2020]

[9] ROS.org, «roslaunch» 2020 [En línea]. Available: http://wiki.ros.org/roslaunch [Último acceso: Agosto 2020]

[10] ROS.org, «rostopic» 2020 [En línea]. Available: http://wiki.ros.org/rostopic [Último acceso: Agosto 2020]

[11] ROS.org, «rosnode» 2020 [En línea]. Available: http://wiki.ros.org/rosnode [Último acceso: Agosto 2020]

[12] ROS.org, «catkin_make» 2020 [En línea]. Available: http://wiki.ros.org/catkin/commands/catkin_make [Último acceso: Agosto 2020]

[13] Wikipedia, « MATLAB ,» 2020 [En línea]. Available: https://es.wikipedia.org/wiki/MATLAB [Último acceso: Octubre 2020]

[14] J. G. Martin, R.A. García, E.F. Camacho (2020, marzo), Event-MILP-Based Task Allocation for Heterogeneous Robotic Sensor Network for Thermosolar Plants

[15] Husarion Docs, «ROSbot 2.0 ,» Noviembre 2017 [En línea]. Available: https://husarion.com/manuals/rosbot-manual/#rosbot-manual-overview [Último acceso: Octubre 2020]

[16] ROS.org, «move_base ,» 2020 [En línea]. Available: http://wiki.ros.org/move_base [Último acceso: Octubre 2020]

[17] Walter Rosero, «Youtube Como Descargar e Instalar Ubuntu Junto a Windows ,» 3 de septiembre de 2018 [En línea]. Available: https://www.youtube.com/watch?v=h9cPABYSJSI&list=PLhS2yXI-fzCrivAty4FnUZJoKU9oVPO-H&index=5&t=1215s&ab_channel=WalterRosero [Último acceso: Marzo 2020]

[18] JFET991, «Youtube Instalar ROS-KINETIC/GAZEBO/TURTLEBOT ,» 10 de marzo de 2018 [En línea]. Available: https://www.youtube.com/watch?v=q-I-3x7mPgs&list=PLhS2yXI-

fzCrivAty4FnUZJoKU9oVPO-H&index=4&t=562s&ab_channel=JFET991 [Último acceso: Marzo 2020]

[19] ProgrammingKnowledge2, «Install OpenCV in Ubuntu 16.04 / Ubuntu 18.04 LTS (Linux) / Ubuntu 20.04 LTS,» 11 de Agosto 2017 [En línea]. Available: https://www.youtube.com/watch?v=0vjC2UHptU4&list=PLhS2yXI-fzCrivAty4FnUZJoKU9oVPO-H&index=17&ab_channel=ProgrammingKnowledge2 [Último acceso: Marzo 2020]

[20] The Construct, «Youtube [ROS Q&A] 130 - How to lanch multiple robots in Gazebo simulator?,» 18 de junio de 2018 [En línea]. Available: https://www.youtube.com/watch?v=mFTkN5v4Jzc&list=PLhS2yXI-fzCrivAty4FnUZJoKU9oVPO-H&index=15&ab_channel=TheConstruct [Último acceso: Marzo 2020]

[21] Programación ATS, «Youtube 123. Programación en C++ || Archivos || Escribir en un archivo de texto ,» 28 de diciembre de 2016 [En línea]. Available: https://www.youtube.com/watch?v=GaqgqQL3wnQ&list=PLhS2yXI-fzCrivAty4FnUZJoKU9oVPO-H&index=21&t=485s&ab_channel=Programaci%C3%B3nATS [Último acceso: Abril 2020]

[22] «Euronics,» 2020 [En línea]. Available: https://www.euronics.es/irobot-roomba-605-robot-aspirador-negro-y-blanco.html [Último acceso: Octubre 2020]

[23] «ROS logo.svg,» 2020 [En línea]. Available: https://es.wikipedia.org/wiki/Archivo:Ros_logo.svg [Último acceso: Octubre 2020]

ANEXOS

Anexo A. Código “robot1.cpp”

```
# include <ros/ros.h>
# include <math.h>
# include <std_msgs/Float32MultiArray.h>
# include <geometry_msgs/Twist.h>
# include <sensor_msgs/Image.h>
# include <std_msgs/UInt8.h>
# include <std_srvs/Empty.h>
#include <nav_msgs/Odometry.h>
#include <image_transport/image_transport.h>
#include <cv_bridge/cv_bridge.h>
#include <sensor_msgs/image_encodings.h>
#include <opencv2/imgproc/imgproc.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <iostream>
#include<stdlib.h>
#include<string.h>
#include<stdio.h>
#include<sstream>
#include<fstream>
#include "std_msgs/Float64MultiArray.h"
#include <array>
ros::Publisher action_pub1;
geometry_msgs::Twist set_vel1;
int flag1=0;
int flag3=0;
int flag_robot1=0;
float p=0;
float r1=0;
float a=0;
float x;
float y;
float xuno=0;
float yuno=0;
```

```
int kp=5;
int kd=11;
float ki=0;
float r;
float theta;
double pi=acos(-1.0);
float error1=0;
float lasterror1=0;
float cumerror1=0;
float matrix1[2][10/2];
int d1=0;
int rec1=0;
int tam1=0;
float posx1=1.0;
float posy1=1.0;

//si theta=0.7 gira 90º, si theta=0.99 gira 180º, si theta=0.35 gira 45º
void counterCallback1(const nav_msgs::Odometry::ConstPtr &msg) {
        if(flag_robot1==1 && rec1<tam1){
                x=matrix1[0][rec1];
                y=matrix1[1][rec1];
            xuno=x-posx1;
            yuno=y-posy1;
            if(xuno<0 && yuno<0){
                  flag3=-1;
             }
             if(yuno>=0){ //primer y segundo cuadrante
                   if(xuno>0 && yuno>=0){ //giro primer cuadrante
                         theta=atan(yuno/xuno)*(180/pi)*(0.35/45);
                    }
                    if(xuno==0 && yuno>0){ //gira 90º
                    theta=0.7;

             }
             if(xuno<0 && yuno>0){ //giro segundo cuadrante
                    theta=180+atan(yuno/xuno)*(180/pi);
                    if(theta>90 && theta<=116.565){
                           theta=0.7+(0.15/26.565)*(theta-90);
```

```
            }
            else if(theta>116.565 && theta<=135){
                    theta=0.85+(0.08/18.435)*(theta-116.565);
            }
            else if(theta>135 && theta<=153.435){
                   theta=0.93+(0.05/18.435)*(theta-135);
            }
            else if(theta>153.435 && theta<180){
                    theta=0.98+(0.02/26.565)*(theta-153.435);
            }
     }
     if(xuno<0 && yuno==0){ //gira 180º
            theta=0.999999;
     }
     error1=theta-msg->pose.pose.orientation.z;
     cumerror1=cumerror1+error1;
     if(msg->pose.pose.orientation.z!=theta){
            set_vel1.angular.z = kp*error1+ki*cumerror1+kd*(error1-lasterror1); //controlador PID
        }
        if(fabs(error1)<0.001 && flag1==0){
            set_vel1.linear.x = 0.5 ;
        }
}
else{ //tercer y cuarto cuadrante
        if(xuno>0){ //giro cuarto cuadrante
             theta=atan(yuno/xuno)*(180/pi)*(0.35/45);
        }
        if(xuno==0){ //gira -90º
             theta=-0.7;

        }
        if(xuno<0){ //giro tercer cuadrante
             theta=180+atan(yuno/xuno)*(180/pi);
             if(theta>180 && theta<=206.565){
                    theta=-0.99+(0.01/26.565)*(theta-180);
             }
             else if(theta>206.565 && theta<=225){
                    theta=-0.98+(0.05/18.435)*(theta-206.565);
```

```
                }
                else if(theta>225 && theta<=243.435){
                        theta=-0.93+(0.08/18.435)*(theta-225);
                }
                else if(theta>243.435 && theta<=270){
                        theta=-0.85+(0.15/26.565)*(theta-243.435);
                }
            }
            error1=theta-msg->pose.pose.orientation.z;
            cumerror1=cumerror1+error1;
            if(msg->pose.pose.orientation.z!=theta){
                    set_vel1.angular.z = kp*error1+ki*cumerror1+kd*(error1-lasterror1); //controlador PID
            }
            if(-error1<0.01 && flag1==0){
                    set_vel1.linear.x = 0.5 ;
            }
      }
      if(fabs(msg->pose.pose.position.x-x)<0.2 && fabs(msg->pose.pose.position.y-y)<0.2){
             set_vel1.linear.x = 0 ;
             flag1=1;
             rec1=rec1+1;
    }
    if(msg->pose.pose.position.x-x<0.1 && msg->pose.pose.position.y-y<0.1 && flag3==-1){
            set_vel1.linear.x = 0 ;
            flag1=1;
            rec1=rec1+1;
   }
   lasterror1=error1;
   action_pub1.publish(set_vel1);
    if(flag1==1){
           posx1=msg->pose.pose.position.x;
           posy1=msg->pose.pose.position.y;
           lasterror1=0;
           cumerror1=0;
           sleep(5);
           flag1=0;
    }
}
```

```
void counterCallback2(const std_msgs::Float64MultiArray::ConstPtr &msg) {
int tam=10;
float Arr[10];
int i = 0;
int j = 0;
int k=1;
for(auto it = msg->data.begin(); it != msg->data.end(); ++it){
    Arr[i] = *it;
    i++;
}
for(i = 0; i < 2; i++){
    for(j = 0; j < Arr[0]; j++){
        matrix1[i][j]=Arr[k];
        k++;
    }
}
tam1=Arr[0];
    flag_robot1=1;
 }
int main ( int argc, char **argv) {
    ros::init(argc, argv, "robot1" );
    ros:: NodeHandle n ( "~" ) ;
    ros:: Rate loop_rate ( 50 ) ;
    action_pub1 = n.advertise<geometry_msgs::Twist>( "/robot1/cmd_vel" , 1 );
    ros::Subscriber sub1 = n.subscribe("/robot1/odom", 1000, counterCallback1);
    ros::Subscriber sub2 = n.subscribe("/array", 1000, counterCallback2);
    set_vel1.linear.x = 0 ;
    set_vel1.linear.y = 0 ;
    set_vel1.linear.z = 0 ;
    set_vel1.angular.x = 0 ;
    set_vel1.angular.y = 0 ;
    set_vel1.angular.z = 0 ;
    while (ros::ok()) {
        ros::spinOnce();
        loop_rate.sleep();
    }
}
```

Anexo B. Código “move_base.cpp”

```
//Lo primero es declarar la variable donde se va a publicar el mensaje y la variable donde se guardará el mensaje en cuestión
ros::Publisher action_pub1;
geometry_msgs::PoseStamped pos;
//Luego se le asigna el tema al que va a publicar y el tipo de mensaje que va a enviar.
action_pub1 = n.advertise<geometry_msgs::PoseStamped>( "/Robot1/move_base_simple/goal" , 1 );
//Después se le asigna al mensaje: el nombre del mapa, la posición y orientación que quiera que esté el robot.
pos.header.frame_id="Robot1/map";
pos.pose.position.x=400;
pos.pose.position.y=200;
pos.pose.position.z=0;
pos.pose.orientation.x=0;
pos.pose.orientation.y=0;
pos.pose.orientation.z=0;
pos.pose.orientation.w=1;
//Y por último, se envía el mensaje en el tema que se introdujo antes.
action_pub1.publish(pos);
```

Anexo C. Código “main.launch”

```
<launch>
  <param name="/use_sim_time" value="true" />
  <!-- start world -->
  <node name="gazebo" pkg="gazebo_ros" type="gazebo"
args="$(find turtlebot_gazebo)/worlds/empty_wall.world" respawn="false" output="screen" />
  <!-- include our robots -->
  <include file="$(find tutorial_pkg)/launch/robots.launch"/>
</launch>
```

Anexo D. Código “robots.launch”

```
<launch>
 <arg name = "frame1" default = "Robot1"/>
 <arg name = "frame2" default = "Robot2"/>
 <arg name = "frame3" default = "Robot3"/>
 <param name="robot_description"
  command="$(find xacro)/xacro.py $(find rosbot_description)/urdf/rosbot.xacro" />

 <!-- BEGIN ROBOT 1-->
 <group ns="$(arg frame1)">
  <param name="tf_prefix" value="$(arg frame1)" />
  <include file="$(find tutorial_pkg)/launch/one_robot.launch" >
   <arg name="init_pose" value="-x 0 -y 0 -z 0" />
   <arg name="robot_name"  value="$(arg frame1)"/>
   <arg name="frame" value= "$(arg frame1)"/>
  </include>
 </group>

 <!-- BEGIN ROBOT 2-->
 <group ns="$(arg frame2)">
  <param name="tf_prefix" value="$(arg frame2)" />
  <include file="$(find tutorial_pkg)/launch/one_robot.launch" >
   <arg name="init_pose" value="-x -2 -y 0 -z 0" />
   <arg name="robot_name"  value="$(arg frame2)"/>
   <arg name="frame" value= "$(arg frame2)"/>
  </include>
 </group>

<!-- BEGIN ROBOT 3-->
 <group ns="$(arg frame3)">
  <param name="tf_prefix" value="$(arg frame3)" />
  <include file="$(find tutorial_pkg)/launch/one_robot.launch" >
   <arg name="init_pose" value="-x 2 -y 0 -z 0" />
   <arg name="robot_name"  value="$(arg frame3)"/>
   <arg name="frame" value= "$(arg frame3)"/>
  </include>
 </group>
```

```
</launch>
```

Anexo E. Código “one_robot.launch”

```
<?xml version="1.0" encoding="UTF-8"?>
<launch>
  <arg name="robot_name"/>
  <arg name="init_pose"/>
  <arg name="frame"/>
  <node name="spawn_minibot_model" pkg="gazebo_ros" type="spawn_model"
   args="$(arg init_pose) -urdf -param /robot_description -model $(arg robot_name) "
   respawn="false" output="screen" />

  <node pkg="robot_state_publisher" type="robot_state_publisher"
       name="robot_state_publisher" output="screen"/>

 <node pkg="gmapping" type="slam_gmapping" name="gmapping">
    <param name="base_frame" value="$(arg frame)/base_link"/>
    <param name="odom_frame" value="$(arg frame)/odom" />
    <param name="map_frame" value="$(arg frame)/map" />
    <param name="delta" value="0.1" />
  </node>

  <!-- The odometry estimator, throttling, fake laser etc. go here -->
   <node pkg="move_base" type="move_base" name="move_base" output="screen">
    <param name="controller_frequency" value="10.0"/>
    <rosparam file="$(find rosbot_navigation)/config/costmap_common_params.yaml" command="load"
ns="global_costmap" />
    <rosparam file="$(find rosbot_navigation)/config/costmap_common_params.yaml" command="load"
ns="local_costmap" />
    <rosparam file="$(find rosbot_navigation)/config/local_costmap_params.yaml" command="load" />
    <rosparam file="$(find rosbot_navigation)/config/global_costmap_params.yaml" command="load" />
    <rosparam file="$(find rosbot_navigation)/config/trajectory_planner.yaml" command="load" />
  </node>
  <!-- All the stuff as from usual robot launch file -->
</launch>
```

Anexo F. Código “nav_robot.launch”

```
<launch>
  <node pkg="tutorial_pkg" type="move_base_robot1" name="move_base_robot1"/>
  <node pkg="tutorial_pkg" type="move_base_robot2" name="move_base_robot2"/>
  <node pkg="tutorial_pkg" type="move_base_robot3" name="move_base_robot3"/>
  <node pkg="tutorial_pkg" type="nuevas_misiones" name="nuevas_misiones" />
  <node pkg="tutorial_pkg" type="image_converter_1" name="image_converter_1"/>
  <node pkg="tutorial_pkg" type="image_converter_2" name="image_converter_2"/>
  <node pkg="tutorial_pkg" type="image_converter_3" name="image_converter_3"/>
</launch>
```

Anexo G. Código “image_converter_1.cpp”

```
#include <ros/ros.h>
#include <image_transport/image_transport.h>
#include <cv_bridge/cv_bridge.h>
#include <sensor_msgs/image_encodings.h>
#include <opencv2/imgproc/imgproc.hpp>
#include <opencv2/highgui/highgui.hpp>
static const std::string OPENCV_WINDOW = "Camara robot1";
class ImageConverter{
  ros::NodeHandle nh;
  image_transport::ImageTransport it_;
  image_transport::Subscriber image_sub_;
  image_transport::Publisher image_pub_;
public:
  ImageConverter()
    : it_(nh){
   image_sub_ = it_.subscribe("/Robot1/camera/rgb/image_raw", 1, &ImageConverter::imageCb, this);
   image_pub_ = it_.advertise("/image_converter_1/output_video", 1);
   cv::namedWindow(OPENCV_WINDOW);
  }
  ~ImageConverter(){
   cv::destroyWindow(OPENCV_WINDOW);
  }
  void imageCb(const sensor_msgs::ImageConstPtr& msg){
   cv_bridge::CvImagePtr cv_ptr;
```

```
    try{
      cv_ptr = cv_bridge::toCvCopy(msg, sensor_msgs::image_encodings::BGR8);
    }
    catch (cv_bridge::Exception& e){
      ROS_ERROR("cv_bridge exception: %s", e.what());
      return;
    }
    cv::imshow(OPENCV_WINDOW, cv_ptr->image);
    cv::waitKey(3);
    // Publica el tema para mostrar la cámara del robot
    image_pub_.publish(cv_ptr->toImageMsg());
  }
};
int main(int argc, char** argv){
  ros::init(argc, argv, "image_converter_1");
  ImageConverter ic;
  ros::spin();
  return 0;
}
```

Anexo H. Código “move_base_robot1.cpp”

```
# include <ros/ros.h>
# include <std_msgs/Float32MultiArray.h>
# include <geometry_msgs/Twist.h>
# include <sensor_msgs/Image.h>
# include <std_msgs/UInt8.h>
# include <std_srvs/Empty.h>
#include <nav_msgs/Odometry.h>
#include <image_transport/image_transport.h>
#include <cv_bridge/cv_bridge.h>
#include <sensor_msgs/image_encodings.h>
#include <opencv2/imgproc/imgproc.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <iostream>
#include<stdlib.h>
#include<string.h>
#include<stdio.h>
```

```
#include<sstream>
#include<fstream>
#include "std_msgs/Float64MultiArray.h"
#include <array>
#include <geometry_msgs/PoseStamped.h>
#include <actionlib_msgs/GoalID.h>
#include <rosgraph_msgs/Clock.h>
using namespace std;
//Se declaran las variables de los temas que publica
ros::Publisher action_pub1;
ros::Publisher action_pub2;
ros::Publisher action_pub3;
std_msgs::Float64MultiArray mision;
geometry_msgs::PoseStamped pos;
actionlib_msgs::GoalID cancel;
std_msgs::Float64MultiArray bateria1;

double antes_time; //hora que se actualiza cada 5 segundos
double parada=-10; // hora en la que el robot para
double cargar=-30; //hora en la que el robot recarga
double reenviar; //hora que se actualiza si el robot se ha detenido en medio del camino
int flag=0;
int flag2=0;
int flag_robot=1;
float a=0;
float x=0;
float y=0;
float inix=0;
float iniy=0;
int recarga=0;
int pc[4][4]={{5, 5},{-5, 5},{5, -5},{-5, -5}}; //posiciones de los puntos de recarga
int pcx=0;
int pcy=0;
int k=0;
float distpc_menor=0;
float distanciax=0;
float distanciay=0;
float distancia=0;
```

```
float distpcx=0;
float distpcy=0;
float distpc=0;
float cons=0;
int tp=10;  //tiempo que el robot está parado (segundos)
int tr=30;  //tiempo que el robot está recargando (segundos)
float matrix1[20][2]; //pila donde guarda las misiones
int i=0;
int fila=0;
float t=0.5; //tiempo que está captando la radiación solar (minutos)
float d=0; //distancia que ha recorrido el robot cada 5 segundos (km)
float dx;
float dy;
float consumo;
float rendimiento; //rendimiento de la batería
float bateria_inicial=38.85;
float m=1000;
ofstream archivo;

//Función Callback1
void counterCallback1(const nav_msgs::Odometry::ConstPtr &msg) {
   double time =ros::Time::now().toSec();  //actualiza time
   if(flag_robot==1 && i>fila && time>tp+parada && time>tr+cargar){
      if((x!=matrix1[fila][0] || y!=matrix1[fila][1]) && recarga==0){
            distanciax=msg->pose.pose.position.x-matrix1[fila][0];
            distanciay=msg->pose.pose.position.y-matrix1[fila][1];
            distancia=sqrt(distanciax*distanciax+distanciay*distanciay)/1000; //distancia del punto objetivo
            distpcx=matrix1[fila][0]-pc[0][0];
            distpcy=matrix1[fila][1]-pc[0][1];
            distpc_menor=sqrt(distpcx*distpcx+distpcy*distpcy)/1000;
            for(k=1;k<4;k++){
                  distpcx=matrix1[fila][0]-pc[k][0];
                  distpcy=matrix1[fila][1]-pc[k][1];
                  distpc=sqrt(distpcx*distpcx+distpcy*distpcy)/1000;
                  if(distpc<distpc_menor){
                        distpc_menor=distpc; //distancia del punto de recarga más cercano
                  }
            }
```

```
        cons=m*(distancia+distpc_menor)+0.1295*t; //estimo el consumo de la batería
        if(cons>bateria_inicial){
            distpcx=msg->pose.pose.position.x-pc[0][0];
            distpcy=msg->pose.pose.position.y-pc[0][1];
            distpc_menor=sqrt(distpcx*distpcx+distpcy*distpcy)/1000;
            pcx=5;
            pcy=5;
            for(k=1;k<4;k++){
                distpcx=msg->pose.pose.position.x-pc[k][0];
                distpcy=msg->pose.pose.position.y-pc[k][1];
                distpc=sqrt(distpcx*distpcx+distpcy*distpcy)/1000;
                  if(distpc<distpc_menor){
                    distpc_menor=distpc;
                    pcx=pc[k][0];
                    pcy=pc[k][1];
                }
            }
            x=pcx;
            y=pcy;
            recarga=1;
        }
        else{
            x=matrix1[fila][0];
            y=matrix1[fila][1];
        }
    }
    else{
        if(recarga==0){
            x=matrix1[fila][0];
            y=matrix1[fila][1];
            if(fabs(msg->pose.pose.position.x-x)<1 && fabs(msg->pose.pose.position.y-y)<1){
                mision.data.clear();
                mision.data.push_back(x);
                mision.data.push_back(y);
                mision.data.push_back(rendimiento);
                action_pub3.publish(mision);
                cancel.id="";
                action_pub2.publish(cancel);
```

```
        parada =ros::Time::now().toSec();
        consumo=0.1295*t;
        rendimiento=100*(bateria_inicial-consumo)/bateria_inicial;
        bateria_inicial=bateria_inicial-consumo;
        flag=0;
        fila=fila+1;
    }
    else{
        if(time-antes_time>5){
            antes_time =ros::Time::now().toSec();
            dx=fabs(msg->pose.pose.position.x-inix);
        dy=fabs(msg->pose.pose.position.y-iniy);
        inix=msg->pose.pose.position.x;
        iniy=msg->pose.pose.position.y;
        d=sqrt(dx*dx+dy*dy)/1000;
        consumo=m*d;
        rendimiento=100*(bateria_inicial-consumo)/bateria_inicial;
        bateria_inicial=bateria_inicial-consumo;
        archivo.open("posicion_robot1.txt",ios::out);
        if(archivo.fail()){
            exit(1);
        }
        archivo<<msg->pose.pose.position.x;
        archivo<<" ";
        archivo<<msg->pose.pose.position.y;
        archivo<<" ";
        archivo<<rendimiento;
        archivo<<" ";
        archivo<<recarga;
        archivo<<" ";
        archivo.close();
        if(dx<0.1 && dy<0.1 && fabs(inix-0)>0.1 && fabs(iniy-0)>0.1 && time-reenviar>10){
            reenviar =ros::Time::now().toSec();
            pos.pose.position.x=x;
            pos.pose.position.y=y;
            pos.pose.position.z=0;
            pos.pose.orientation.x=0;
            pos.pose.orientation.y=0;
```

```
                pos.pose.orientation.z=0;
                pos.pose.orientation.w=1;
                action_pub1.publish(pos);
            }
        }
        if(flag<1){ //lo envia otra vez para asegurar que el robot ha recibido la nueva mision
            pos.header.frame_id="Robot1/map";
            pos.pose.position.x=x;
            pos.pose.position.y=y;
            pos.pose.position.z=0;
            pos.pose.orientation.x=0;
                pos.pose.orientation.y=0;
                pos.pose.orientation.z=0;
                pos.pose.orientation.w=1;
                action_pub1.publish(pos);
                flag=flag+1;
            }
        }
    }
    else{
        if(fabs(msg->pose.pose.position.x-x)<0.2 && fabs(msg->pose.pose.position.y-y)<0.2){
            mision.data.clear();
            mision.data.push_back(x);
            mision.data.push_back(y);
            mision.data.push_back(rendimiento);
            action_pub3.publish(mision);
            cancel.id="";
            action_pub2.publish(cancel);
            cargar =ros::Time::now().toSec();
            bateria_inicial=38.85;
            flag=0;
            recarga=0;
        }
        else{
            if(time-antes_time>5){
                antes_time =ros::Time::now().toSec();
                dx=fabs(msg->pose.pose.position.x-inix);
                dy=fabs(msg->pose.pose.position.y-iniy);
```

```
                    archivo.open("posicion_robot1.txt",ios::out);
                    if(archivo.fail()){
                        cout<<"No se pudo abrir el archivo";
                        exit(1);
                    }
                    archivo<<msg->pose.pose.position.x;
                    archivo<<" ";
                    archivo<<msg->pose.pose.position.y;
                    archivo<<" ";
                    archivo<<rendimiento;
                    archivo<<" ";
                archivo<<recarga;
                archivo<<" ";
                archivo.close();
                if(dx<0.1 && dy<0.1){
                    pos.pose.position.x=x;
                    pos.pose.position.y=y;
                    pos.pose.position.z=0;
                    action_pub1.publish(pos);
                }
            }
            if(flag<1){ //lo envia varias veces para asegurar que el robot ha recibido la nueva mision
                pos.header.frame_id="Robot1/map";
                pos.pose.position.x=x;
                pos.pose.position.y=y;
                pos.pose.position.z=0;
                pos.pose.orientation.x=0;
                pos.pose.orientation.y=0;
                pos.pose.orientation.z=0;
                pos.pose.orientation.w=1;
                action_pub1.publish(pos);
                flag=flag+1;
            }
        }
    }
  }
 }
}
```

```
//Función Callback2
void counterCallback2(const std_msgs::Float64MultiArray::ConstPtr &msg) {
	float Arr[2];
	int j = 0;
	for(auto it = msg->data.begin(); it != msg->data.end(); ++it){
		Arr[j] = *it;
		j++;
	}
	for(j = 0; j < 2; j++){
		matrix1[i][j]=Arr[j];
	}
	i=i+1;
	flag_robot=1;
}

int main ( int argc, char **argv) {
	ros::init(argc, argv, "move_base_robot1" );
	ros:: NodeHandle n ( "~" ) ;
	ros:: Rate loop_rate ( 50 ) ;
	action_pub1 = n.advertise<geometry_msgs::PoseStamped>( "/Robot1/move_base_simple/goal" , 1 );
	action_pub2 = n.advertise<actionlib_msgs::GoalID>( "/Robot1/move_base/cancel" , 1 );
	action_pub3 = n.advertise<std_msgs::Float64MultiArray>("/Robot1/mision", 1);
	ros::Subscriber sub1 = n.subscribe("/Robot1/odom", 1000, counterCallback1);
	ros::Subscriber sub2 = n.subscribe("/array", 1000, counterCallback2);
	archivo.open("posicion_robot1.txt",ios::out);  //abro el documento txt para sobreescribir
	if(archivo.fail()){
		cout<<"No se pudo abrir el archivo";
		exit(1);
	}
	archivo<<inix;
	archivo<<" ";
	archivo<<iniy;
	archivo<<" ";
	archivo<<100;
	archivo<<" ";
	archivo<<0;
	archivo<<" ";
```

```
        archivo.close();
        antes_time =ros::Time::now().toSec();
        reenviar =ros::Time::now().toSec();
        while (ros::ok()) {
                ros::spinOnce();
                ros::Duration d(5);
                loop_rate.sleep();
        }
}
```

Anexo I. Código “nuevas_misiones.cpp”

```
#include <iostream>
#include<stdlib.h>
#include<string.h>
#include<stdio.h>
#include<sstream>
#include<fstream>
#include "ros/ros.h"
#include "std_msgs/Float64MultiArray.h"

using namespace std;
int tc=2;
int nh=75;
float m1[2];
float m2[2];
float m3[2];
int bandera1=0;
int bandera2=0;
int bandera3=0;

void lectura();
void escritura();

int main(int argc, char **argv){
        ros::init(argc, argv, "nuevas_misiones");
        ros::NodeHandle n;
```

```
ros:: Rate loop_rate ( 50 ) ;
ros::Publisher pub = n.advertise<std_msgs::Float64MultiArray>("/array", 100);
 ros::Publisher pub2 = n.advertise<std_msgs::Float64MultiArray>("/array2", 100);
 ros::Publisher pub3 = n.advertise<std_msgs::Float64MultiArray>("/array3", 100);
std_msgs::Float64MultiArray array;
std_msgs::Float64MultiArray array2;
std_msgs::Float64MultiArray array3;
while (ros::ok()) {
        ros::spinOnce();
        loop_rate.sleep();
        lectura(); //llama a la función lectura


          if(((m1[0]!=0 && m1[1]!=0) || (m1[0]==0 && m1[1]!=0) || (m1[0]!=0 && m1[1]==0)) &&
bandera1==0){
                    array.data.clear();
                     for(int j=0;j<tc;j++){
                           array.data.push_back(m1[j]);
                     }
                     pub.publish(array);
                     bandera1=1;
        }
        if(((m2[0]!=0 && m2[1]!=0) || (m2[0]==0 && m2[1]!=0) || (m2[0]!=0 && m2[1]==0)) &&
bandera2==0){
                   array2.data.clear();
                    for(int j=0;j<tc;j++){
                           array2.data.push_back(m2[j]);
                    }
                    pub2.publish(array2);
                    bandera2=1;
        }
        if(((m3[0]!=0 && m3[1]!=0) || (m3[0]==0 && m3[1]!=0) || (m3[0]!=0 && m3[1]==0)) &&
bandera3==0){
                    array3.data.clear();
                    for(int j=0;j<tc;j++){
                           array3.data.push_back(m3[j]);
                    }
                    pub3.publish(array3);
                    bandera3=1;
        }
```

```
    }
    return 0;
}
void lectura(){
    ifstream archivo;
    string texto;
    char texto2[20];
    int a=0;
    int b=0;
    int c=0;
    int f=0;
    int flag=0;
    int tam=0;
    archivo.open("misiones.txt",ios::in); //tiene que ejecutar el nodo en el directorio que esta el txt
    if(archivo.fail()){
        cout<<"No se pudo abrir el archivo";
        exit(1);
    }
    for(int i=0;i<tc;i++){
        m1[i]=0;
        m2[i]=0;
        m3[i]=0;
    }
    while(!archivo.eof()){ //hace el bucle while dos veces por eso esta el flag para que haga el for una vez
        getline(archivo,texto);
        if(flag==0 && texto!=""){ //pregunto si es la primera vez que entra y si hay algo escrito en el archivo
            cout<<"entra";
            for(a=2;a<nh && f==0;a++){
                if(c<2){
                    if(texto[a]!=' '){ //guarda el numero en formato string
                        texto2[b]=texto[a];
                        b=b+1;
                    }
                    else{
                        istringstream(texto2) >> m1[c]; //lo pasa a float y lo guarda en un vector
                        strcpy(texto2, "");
                        c=c+1;
                        b=0;
```

```
            }
            if(c==2){
                a=a+3;
            }
        }
        if(c>=2 && c<4){
            if(texto[a]!=' '){ //guarda el numero en formato string
                texto2[b]=texto[a];
                b=b+1;
            }
            else{
                istringstream(texto2) >> m2[c-2]; //lo pasa a float y lo guarda en un vector
                    strcpy(texto2, "");
                    c=c+1;
                    b=0;
                }
                if(c==4){
                    a=a+3;
                }
            }
            if(c>=4 && c<6){
                if(texto[a]!=' '){ //guarda el numero en formato string
                    texto2[b]=texto[a];
                    b=b+1;
                }
                else{
                    istringstream(texto2) >> m3[c-4]; //lo pasa a float y lo guarda en un vector
                    strcpy(texto2, "");
                    c=c+1;
                    b=0;
                }
            }
        }
        escritura(); //para borrar lo escrito
        bandera1=0;
        bandera2=0;
        bandera3=0;
    }
```

```
            flag=flag+1;
        }
         fflush(stdin); //limpia las variables
        archivo.close();
}

void escritura(){
    ofstream archivo;
    archivo.open("misiones.txt",ios::out);
    if(archivo.fail()){
        cout<<"No se pudo abrir el archivo";
        exit(1);
  }
  archivo<<"";
  archivo.close();
}
```

Anexo J. Código “inicializa.m”

```
charge_point=[500 500 -500 -500;500 -500 500 -500]'; // matriz de las posiciones de los puestos de recarga
objetivos_no_asignados=[]; //vector de los objetivos que no se asignen
tic //inicia el contador de Matlab
```

Anexo K. Código “nuevos_objetivos.m”

```
function [objetivos_no_asignados,new_missions] = nuevos_objetivos(objetivos_nuevos,charge_point,
objetivos_no_asignados)
robots=zeros(3,3);
```

//Lee el documento *posicion_robot1.txt* para obtener las posiciones y la bateria del primer robot

```
fileID = fopen('/root/catkin_ws/src/tutorial_pkg/src/posicion_robot1.txt');
matrix=fscanf(fileID,'%f');
fclose(fileID);
matrix=matrix';
robots(1,1:3)=matrix(1:3);
```

//Lee el documento *posicion_robot2.txt* para obtener las posiciones y la bateria del segundo robot

```
fileID = fopen('/root/catkin_ws/src/tutorial_pkg/src/posicion_robot2.txt');
matrix=fscanf(fileID,'%f');
```

```
fclose(fileID);
matrix=matrix';
robots(2,1:3)=matrix(1:3);
```
//Lee el documento *posicion_robot3.txt* para obtener las posiciones y la bateria del tercer robot
```
fileID = fopen('/root/catkin_ws/src/tutorial_pkg/src/posicion_robot3.txt');
matrix=fscanf(fileID,'%f');
fclose(fileID);
matrix=matrix';
robots(3,1:3)=matrix(1:3);
```
//Ejecuta la función *funcion_de_costes*
```
[objetive_spots_not_assigned, new_missions] = funcion_de_costes (robots,objetivos_nuevos,charge_point,
objetivos_no_asignados);
objetivos_no_asignados=objetive_spots_not_assigned;
[M N]=size(new_missions);
```
//Guarda los nuevos objetivos en la matriz *nueva_mision*
```
nueva_mision=[1 0 0 2 0 0 3 0 0];
for i=1:M
    if(new_missions(i,1)==1)
      nueva_mision(2:3)=new_missions(i,2:3);
    end
    if(new_missions(i,1)==2)
      nueva_mision(5:6)=new_missions(i,2:3);
    end
    if(new_missions(i,1)==3)
      nueva_mision(8:9)=new_missions(i,2:3);
    end
end
```
//Sobreescribe las misiones en el documento *misiones.txt*
```
    fileID = fopen('/root/catkin_ws/src/tutorial_pkg/src/misiones.txt','w');
    fprintf(fileID,'%d ',nueva_mision(1));
    fprintf(fileID,'%2.2f ',nueva_mision(2:3));
    fprintf(fileID,'%d ',nueva_mision(4));
    fprintf(fileID,'%2.2f ',nueva_mision(5:6));
    fprintf(fileID,'%d ',nueva_mision(7));
    fprintf(fileID,'%2.2f ',nueva_mision(8:9));
    fclose(fileID);
    nueva_mision
end
```

Anexo L. Código "función_de_costes.m"

```
function [objetive_spots_not_assigned, new_missions] = funcion_de_costes (robots, objetivos_nuevos,
charge_point, objetivos_no_asignados)
//Asigna la hora en la que se creo los objetivos nuevos
objetivos_nuevos(:,4)=toc;
Q1 = 1;
Q4 = 0.075;
Q3 = 0.5;
objetives=[objetivos_nuevos;objetivos_no_asignados];
N_robots =size(robots,1);
N_objetives = size(objetives,1);

  if  N_objetives ~= 0
     Aeq = [];
     for i=1:N_robots
        Aeq = [Aeq, eye(N_objetives)];
     end
     Aeq = [Aeq, eye(N_objetives)];
     Beq = ones(N_objetives,1);
     A = [];
     A_aux_1 = zeros(N_robots, N_objetives);
     for i = 1:N_robots
        A_aux_2 = A_aux_1;
        A_aux_2(i,:) = ones(1,N_objetives);
        A = [A, A_aux_2];
     end
     A = [A, A_aux_1];
     B = ones(N_robots,1);
     f = [];
//calcula la distancia del próximo objetivo y del punto objetivo al punto de recarga más cercano
     for i = 1:N_robots
        for j = 1:N_objetives
           distancia_objetivo=sqrt((robots(i,1)-objetives(j,1))^2+(robots(i,2)-objetives(j,2))^2);
           for l = 1:length(charge_point)
              distancia_charge_point(l)=sqrt((objetives(j,1)-charge_point(l,1))^2+(objetives(j,2) -
```

```
            charge_point (l,2))^2);
        end
        distance_from_objetive_2_charge = min(distancia_charge_point);
//calcula la matriz Q2 y landa necesarias para asignar las misiones
        Q2 = 1.1 - 0.005*robots(i,3);
        landa(i,j) = (Q1 * distancia_objetivo + Q2 * distance_from_objetive_2_charge)/1000;
        f = [f, landa(i,j)];
      end
    end
    for k = 1:length(f)
      if f(k) == Inf
        f(k) = 0;
        Aeq(:,k) = zeros(size(Aeq,1),1);
      end
    end
//calcula c necesario para asignar las misiones
    for j = 1:N_objetives
      t_na = objetives(j,3);
      c(j)=Q3 + Q4 * t_na;
      f = [f, c(j)];
    end
    lb = zeros(1,size(A,2));
    ub = ones(1,size(A,2));
    for i=1:size(f,2)
      intcon(i) = i;
    end
//Con la función de Matlab intlinprog distribuye los objetivos de cada robot
    x = intlinprog(f,intcon,A,B,Aeq,Beq,lb,ub);
//Guarda los objetivos asignados en la matriz Uij y los no asignados en Vj
    Uij = x(1:end - N_objetives);
    Vj = x(end - N_objetives + 1:end);
  end
  if N_objetives == 0
    Uij = [];
    Vj = [];
  end
  number_of_objetives = size(Vj,1);
  number_of_robots = size(Uij,1) / number_of_objetives;
```

```
    objetive_spots_not_assigned = [];
    new_missions = [];
    if number_of_objetives ~= 0 && number_of_robots ~= 0
        new_missions = [];
        k = 1;
```

//Asigna un objetivo a cada robot guardándolo en la matriz *new_missions*

```
        for i = 1: number_of_robots
            for j = 1: number_of_objetives
                if  Uij(k) == 1
                    robot_index = i;
                    mission_type = 2;
                    destination_spot = [objetives(j,1) objetives(j,2)];
                    new_missions = [new_missions; robot_index destination_spot];
                end
                k = k + 1;
               end
            end
            objetive_spots_not_assigned = [];
```

//Guarda los objetivos no asignados en el vector *objetive_spots_not_assigned*

```
            for j = 1:number_of_objetives
                  if Vj(j) == 1
                        not_assigned_spot = [objetives(j,1) objetives(j,2)];
                        objetive_appareance = objetives(j,4);
                        not_assigned_time = toc-objetives(j,4);
                        objetive_spots_not_assigned = [objetive_spots_not_assigned; not_assigned_spot
                        not_assigned_time objetive_appareance];
                  end
            end
        end
end
```