Universidad de Alcalá
Escuela Politécnica Superior

GRADO EN INGENIERÍA ELECTRÓNICA Y AUTOMÁTICA INDUSTRIAL

Trabajo Fin de Grado

ESTUDIO DEL INTERFAZ MATLAB-ROS PARA EL DESARROLLO DE APLICACIONES ROBÓTICAS

Autor: Jose Enrique Escobar Abansés

Tutor/es: Elena López Guillén

2016

UNIVERSIDAD DE ALCALÁ

Escuela Politécnica Superior

GRADO EN INGENIERÍA ELECTRÓNICA Y AUTOMÁTICA INDUSTRIAL

Trabajo Fin de Grado

"Estudio del interfaz Matlab-ROS para el desarrollo de aplicaciones robóticas"

Jose Enrique Escobar Abansés

2016

UNIVERSIDAD DE ALCALÁ

Escuela Politécnica Superior

GRADO EN INGENERÍA EN ELECTRÓNICA Y AUTOMÁTICA INDUSTRIAL

Trabajo Fin de Grado

"Estudio de toolbox en entorno Matlab para su unión con ROS y desarrollo de aplicaciones"

Autor:	Jose Enrique Escobar Abansés
Universidad:	Universidad de Alcalá
País:	España
Profesor Tutor:	María Elena López Guillén

Presidente: Julio Pastor Mendoza

Vocal 1: Santiago Cóbreces Álvarez

Vocal 2: Elena López Guillén

CALIFICACIÓN:.....

FECHA:.....

Agradecimientos

Gracias a mi familia, por apoyarme en todo momento, y en especial a mis padres, que han hecho este sueño posible. Gracias también a mis amigos, por estar siempre ahí, acompañándome en esta aventura, y en especial a mi novia, María, por aguantarme en esta etapa final, la más dura. Gracias.

Tabla de contenido

Tabla de ilustraciones

Resumen

En este proyecto se van a desarrollar una serie de aplicaciones de control de movimiento de un robot móvil mediante la comunicación de Matlab con el framework para la escritura de software de robots "Robot Operating System" (ROS), para aprovechar las ventajas de cada uno. El fin es manejar y leer los datos del robot desde Matlab simulándolo en ROS en dos de sus simuladores, uno bidimensional (STDR) y otro más complejo en 3D y con motor de físicas (Gazebo).

Previamente al desarrollo de las aplicaciones, se estudiará la toolbox de Matlab que permite dicha comunicación, Robotics System Toolbox, y se adquirirán conocimientos básicos de ROS y sus características principales para su comunicación con Matlab.

Palabras clave: Robot móvil, Matlab, ROS, STDR, Gazebo, Robotics System Toolbox.

Abstract

In this project, several motion control applications of a mobile robot are going to be developed through the communication of Matlab with the framework for the writing of the robot's software "Robot Operating System" (ROS), to take advantage of each one. The purpose is to manage and read the robot data from Matlab simulating them in ROS, in two simulators; one will be two-dimensional (STDR) and the other one, more complex, in 3D and with a physics engine (Gazebo).

Before the development of the applications, the Matlab toolbox will be studied, (which allows the communication previously mentioned), Robotics System Toolbox. Basic knowledges of ROS and its main features will be acquired, for its communication with Matlab.

Keywords: Mobile Robot, Matlab, ROS, STDR, Gazebo, Robotics System Toolbox.

1 Introducción

1.1 El auge de la robótica móvil

Desde la antigüedad el ser humano ha intentado construir máquinas automáticas que facilitaran su trabajo, imitaran partes del cuerpo humano o por el simple afán de aprender e investigar. Ya los antiguos egipcios acoplaban partes del cuerpo mecánicas a las estatuas o la antigua Grecia creaba los primeros "autómatas", ingenios mecánicos o hidráulicos que imitaban la figura y los movimientos de un ser animado y que solían ir más dirigidos al entretenimiento que a una labor productiva [1].

Estos "autómatas" son el origen de la robótica móvil, pero sufrieron un largo proceso de desarrollo para llegar a lo que se conoce hoy en día. Durante el Renacimiento, grandes figuras de la época se atrevieron con estas complejas obras, entre ellos el hombre por excelencia del Renacimiento, Leonardo Da Vinci, que creó entre otros un león que andaba solo y abría la boca dejando asombrado a aquél que lo contemplaba. Con la entrada del siglo XVIII se produce un gran avance con la mejora y perfeccionamiento de los autómatas. Se obsesionan con intentar reproducir lo más fielmente posible los movimientos y comportamientos de los seres vivos, dando lugar a creaciones tales como el pato con aparato digestivo de Jacques de Vaucanson o El escritor de Pierre Jaquet-Droz que era capaz de escribir pequeños textos con pluma imitando detalles como mojar la tinta y escurrir el sobrante.

Ya en la década de los 50's se desarrollan los primeros robots gracias al avance en la investigación en inteligencia artificial, como la creación de las primeras computadoras electrónicas. En 1954, Devol diseña el primer robot programable y a lo largo de las

siguientes décadas se producirá un intenso desarrollo de la robótica y en concreto de la robótica industrial.

Durante los 60's y 70's se crean robots que utilizan sonares para moverse o que son capaces de seguir líneas, utilizando cámaras para ver. La Unión Soviética explora la Luna con su vehículo lunar Lunojod en 1973 y poco más tarde, la NASA envía dos vehículos espaciales no tripulados a Marte por vez primera.

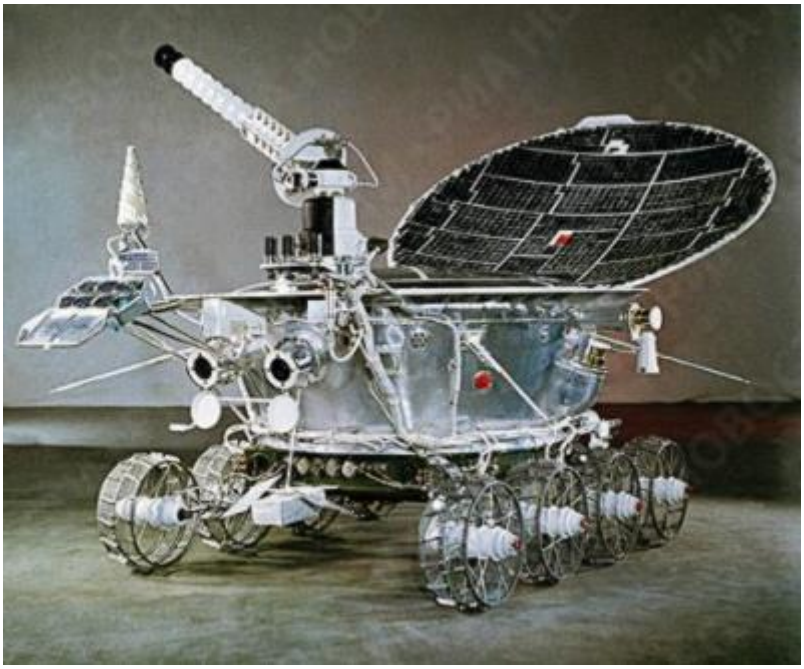

Ilustración 1 Vehículo lunar Lunojod

El aumento del interés del público tiene como resultado el avance en robots para uso doméstico, sirviendo de entretenimiento o educativos como el RB5X, que todavía existe. En 1987 los laboratorios Hughes Research muestran la primera operación autónoma de un vehículo robótico basada en sensores para cruzar un país. A principios de los 90 se desarrollan proyectos de gran transcendencia, como robots móviles para transportar medicamentos o instrumental en hospitales, el MDARS-I, desarrollado por el Departamento de Defensa americano para mejorar la seguridad o DANTE para la exploración de volcanes activos.

A mediados de los 90 se produce un punto de inflexión al aparecer en el mercado el robot móvil programable Pioneer a un precio asequible. Esto permite un aumento generalizado de la robótica de investigación y de su estudio en universidades, durante la siguiente década se convertiría en una parte habitual dentro del plan de estudios de cualquier universidad.

Ya en el siglo XXI la industria ha pasado de tener un uso intensivo de mano de obra a una industria muy automatizada con el objeto de cumplir con los máximos niveles de exigencia en calidad, productividad y eficiencia, y en estos procesos de automatización la robótica móvil tiene una presencia determinante. La integración de diferentes niveles de automatización en la cadena de producción asegura la competitividad de forma duradera y una serie de ventajas decisivas que hacen que el potencial de crecimiento sea inmenso.

Los campos de aplicación en la industria y en la sociedad crecen día a día, el transporte y logística donde está más implantada, su incorporación a muchas otras fases del proceso de producción donde los robots empiezan a ser más habituales, la ayuda que aportan a hospitales, cuerpos de bomberos o servicios en general, sumado al avance constante de la tecnología hacen que la robótica móvil esté en auge, y no hay expectativa de que esto vaya a cambiar.

Ilustración 2 Robot móvil de uso industrial

1.2 Elementos principales de un robot móvil

La estructura de un robot es el conjunto de elementos mecánicos que le dan forma y soportan los demás elementos que lo componen [2] [3]. Dependiendo de la aplicación, serán necesarios unos u otros componentes, pero todos ellos están formados fundamentalmente por:

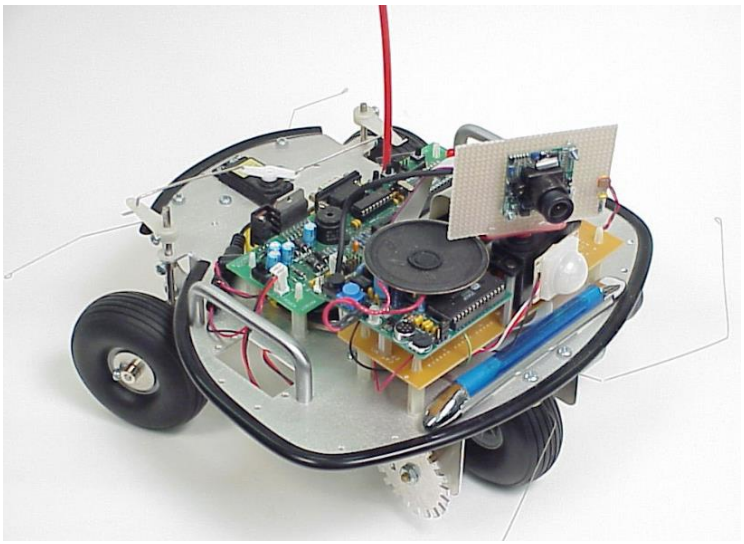

Ilustración 3 Vista interna de un robot móvil

- **Sensores**: Los sensores son los encargados de recoger la información del entorno y enviarla al sistema de control para su procesamiento. Los sensores se pueden clasificar en internos o externos dependiendo de la función que realicen:
    - Sensores propioceptivos: Sirven para detectar variables internas al propio robot, como la fuerza ejercida, la velocidad de los motores, posición de elementos móviles, etc.
    - Sensores exteroceptivos: Son los que sirven para tomar datos del entorno del robot, como detectar objetos, distancias, niveles de iluminación, temperatura, etc.
- **Actuadores**: Son los encargados de generar el movimiento en los elementos móviles del robot. Dependiendo de la forma en la que envíen la energía para realizar el movimiento se pueden clasificar como:
    - Actuadores eléctricos: Son los motores eléctricos que se emplean para robots de tamaño mediano, que no requieren de tanta velocidad ni potencia como los robots diseñados para funcionar con impulsión hidráulica.

  - Actuadores hidráulicos y neumáticos: Funcionan sobre las bases de fluidos a presión y de aire a presión respetivamente, los actuadores neumáticos pueden ejercer una menor fuerza y precisión, aunque es suficiente en muchos casos. Sus funciones son similares y son utilizados para generar más potencia que la que podría dar un actuador eléctrico (en la actualidad el avance tecnológico empieza a equiparar los actuadores eléctricos a los hidráulicos, relegando a estos a un segundo plano).

- **Sistemas de control**: Los sistemas de control son los encargados de analizar la información que les mandan los sensores, tomar decisiones y dar órdenes para que las ejecuten los actuadores. Estos sistemas de control pueden realizarse de dos formas. Mediante un circuito electrónico programable, para pequeños robots de poca complejidad, o mediante ordenador con el uso de los diversos entornos de desarrollo de aplicaciones robóticas.

1.3 Entornos de desarrollo para aplicaciones robóticas

A la hora de programar robots se deben cumplir ciertos requisitos más exigentes que los requeridos para la programación en entornos más habituales como los ordenadores personales. Estos programas deben dominar gran cantidad de dispositivos sensoriales y de actuación y deben ser ágiles para cumplir con los requisitos de tiempo real y, al tener que atender a muchas operaciones simultáneamente, necesitan un sistema operativo que soporte la multitarea y la comunicación interprocesos. Resulta muy útil disponer de una interfaz gráfica para la depuración de los programas, pero el código de la aplicación debe ser el que actualice la interfaz y esté pendiente de la interacción con el usuario, lo que le confiere una mayor complejidad.

El mayor problema es que las aplicaciones para robots no cuentan con un marco estable, pues existe una heterogeneidad debido en parte a la inmadurez del mercado. Por suerte se está avanzando hacia la estandarización gracias a los entornos de desarrollo.

Los Entornos o Plataformas de Desarrollo se sitúan entre la aplicación del robot y el Sistema Operativo para facilitar el acceso a los drivers mediante abstracciones de hardware y la estructuración del software. En la ilustración 4 se puede ver un esquema de su localización dentro del software del robot.

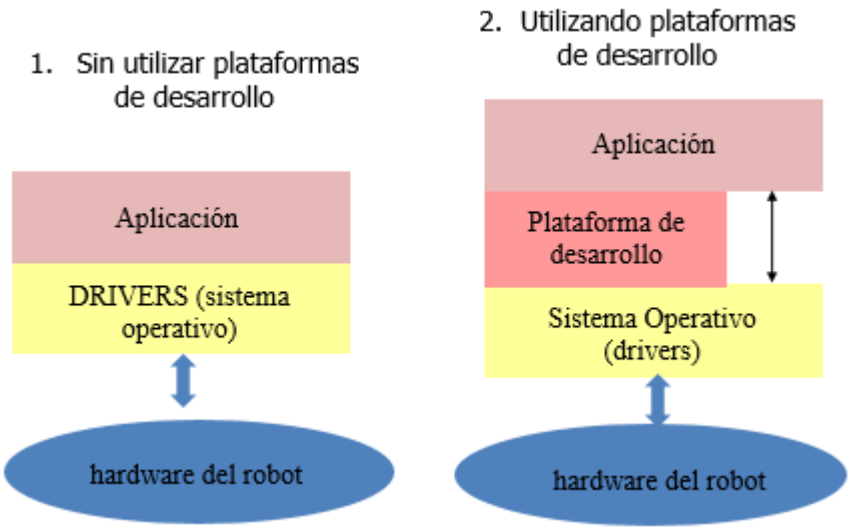

Ilustración 4 Esquema sin y con entorno de desarrollo

Al programar sin entornos de desarrollo [4], los datos de los dispositivos robóticos son adquiridos directamente por el programa de usuario, que contiene también el sistema de control, por lo que cada robot tiene su programa específico. Sin embargo, al utilizar entornos de desarrollo son éstos los encargados de adquirir los datos y depurarlos y el programa sólo se encarga del sistema de control por lo que se facilita el acceso a los drivers mediante abstracciones de hardware y además los entornos de desarrollo incluyen un modelo de programación que facilita la estructuración del software e interfaces gráficas o simuladores.

Los entornos de desarrollo necesitan un Sistema Operativo que ofrezca acceso rápido al hardware del robot para la lectura de los sensores y envío de comandos a través de drivers, soporte para utilizar el hardware de comunicaciones y para el desarrollo de interfaces gráficas, multitarea y los medios para cargar programas y ejecutarlos en los procesadores del propio robot.

Estos SO se dividen en dos categorías, los SO Dedicados, utilizados fundamentalmente en robots pequeños porque aportan ventajas al estar muy ligados al procesador, sensores y actuadores concretos; y los SO Generalistas, entre éstos el que más aceptación ha ganado es GNU/LINUX por ser software libre, la comunidad muy activa y amplia de desarrolladores, las continuas actualizaciones, el abanico de herramientas de desarrollo, su potencia y robustez, la multitarea y sus interfaces gráficas. Uno de sus defectos es que

no es un sistema de tiempo real puro, pero suele ser suficiente para aplicaciones no críticas, aunque si fuera necesario, existe la variante RT-LINUX.

A la hora de programar las aplicaciones robóticas se utilizan lenguajes genéricos siendo los más extendidos C/C++ y Python. Las plataformas de desarrollo suelen determinar el lenguaje en que se escribe la aplicación para poder utilizar la funcionalidad ya resuelta en la plataforma.

Dentro de los entornos de desarrollo se pueden encontrar los que pertenecen a fabricantes avanzados y los que pertenecen a grupos de investigación.

Fabricantes avanzados:

- ARIA: Es un entorno de desarrollo perteneciente a MobileRobots que permite controlar dinámicamente la velocidad del robot, el rumbo, rumbo relativo, y otros parámetros de movimiento, ya sea a través de simples comandos de bajo nivel o a través de su infraestructura de acciones de alto nivel. ARIA también recibe estimaciones de posición, lecturas de sonda y todos los demás datos de funcionamiento. Soporta robots de Pioneer, es de software libre y su SSOO son Linux y MS-Windows.
- ERSP: Este entorno de desarrollo ha sido creado por Evolution Robotics y en su última versión es capaz de reconocer imágenes y objetos de forma fiable en entornos reales y permite al robot o dispositivo moverse de forma autónoma y segura. Soporta los robots de la marca, los ER1, en este caso no son de software libre y si sus Sistemas Operativos son Linux y MS-Windows.
- OPEN-R: La arquitectura OPEN-R es específica para robots de entretenimiento (AIBO). La arquitectura implica el uso de componentes modulares de hardware, como apéndices que se pueden quitar fácilmente y reemplazarse para cambiar la forma y función de los robots, y componentes de software modulares que se pueden intercambiar para cambiar sus patrones de comportamiento y movimiento. Funciona sobre un SO propio de Sony, Aperios, pero a principios de los 2000 deja de utilizarse.

- MRDS (Microsoft Robotics Developer Studio): Es un entorno basado en Windows para el control de robots y la simulación. Está dirigido a desarrolladores académicos, aficionados y comerciales; fue muy utilizado hasta finales de la década pasada y aunque no es de software libre funciona sobre MS-Windows, dando facilidad de uso a usuarios menos especializados.

Grupos de investigación:

- CARMEN: Es una colección de código abierto de software para el control de robots móviles. Carmen es un software modular diseñado para proporcionar la navegación básica incluyendo: base y control del sensor, logging, evitación de obstáculos, localización, planificación de trayectorias, y mapeado.

- Player/Stage: Player es un servidor de red para el control de robots. Proporciona una interfaz limpia y sencilla de los sensores y actuadores del robot a través de la red IP y es compatible con diferentes robots; la plataforma original es la familia de robots Pioneer, pero soporta muchos otros, así como una gran cantidad de sensores comunes. La arquitectura modular de Player hace fácil añadir soporte para nuevo hardware, y la activa comunidad de usuarios y desarrolladores contribuye a aportar los nuevos drivers.

- ROS: Es un proyecto de software libre desarrollado por el Laboratorio de Inteligencia Artificial de Stanford para dar soporte originalmente a un robot propio Desde 2008 continúa en desarrollo primordialmente en Willow Garage, un instituto de investigación robótico con más de veinte instituciones colaborando en él. ROS provee los servicios estándar de un sistema operativo tales como abstracción de hardware, control de dispositivos de bajo nivel o implementación de funcionalidad de uso común. Está basado en una arquitectura de grafos donde el procesamiento tiene lugar en los nodos que pueden recibir, mandar y multiplexar mensajes de sensores, control, estados, planificaciones y actuadores, entre otros. Está orientado para un sistema UNIX (Ubuntu) aunque se ha adaptado a otros sistemas operativos.

1.4 Motivación y objetivos del proyecto

La elección de tema viene dada por la reciente aparición de la toolbox de Matlab "Robotics System Toolbox", herramienta que genera gran interés y que tiene mucho potencial al unir dos mundos muy diferentes, uno ya asentado desde hace años en el mundo matemático como es Matlab y otro que quiere cambiar por completo el ámbito de la robótica, y parece que con éxito, como es ROS.

Matlab ha estado en la primera línea de las herramientas matemáticas durante la última década al ser muy potente a nivel de análisis matemático y utilizado en gran cantidad de sectores, sobre todo el académico, que le aporta mucha popularidad, pero tiene fallos a la hora de simular situaciones complejas por lo que su utilización en la robótica es meramente didáctica.

Por otro lado, ROS es un entorno de desarrollo robótico que apareció hace unos años con la intención de homogeneizar la programación y el control de robots; parte de su éxito se debe al software libre y gratuito y a una comunidad muy activa que permite que el número de robots que soporta aumente cada día. Está basado en Linux y al no tener una interfaz propia puede hacerse complicado para usuarios menos avanzados.

Como se ve la unión de Matlab y ROS permite hacer uso de una plataforma de desarrollo relativamente nueva y que trabaja bajo un sistema operativo menos utilizado, como es Linux, desde un programa con más recorrido y una programación más conocida, como es Matlab. Es su primera aparición en Matlab (2015) y se da por seguro que, si funciona bien, en futuras versiones tendrá el apoyo y recursos suficientes para desarrollarse y aumentar sus posibilidades.

El presente proyecto tiene como objetivo la investigación y estudio de la toolbox "Robotics System Toolbox" de Matlab para su unión con ROS y la creación posterior de varias aplicaciones, el control del movimiento de un robot básico con el simulador STDR en 2D de ROS y el control de un dron desde Matlab, simulado en Gazebo (entorno 3D de simulación de ROS) y con el análisis de sus sensores y periféricos desde Matlab. Estas simulaciones servirían de punto de partida para el control de robots reales pues se necesitan pocos cambios en el código para implementarlo.

El campo de aplicación de este Trabajo de Fin de Grado en la robótica es el control de cualquier robot, desde pequeños robots de poca complejidad hasta drones y robots industriales.

1.5 Estructura de la memoria

En la primera sección, en la que se encuentra esta clasificación, se presenta la introducción del proyecto, empezando con un breve repaso a la historia de la robótica, la descripción de un robot móvil, los diferentes entornos de desarrollo y la motivación y objetivo del proyecto.

En la segunda sección se desarrolla la descripción de las herramientas utilizadas, la máquina virtual, Matlab y ROS y en más profundidad de los simuladores y herramientas utilizadas en este proyecto.

El tercer apartado muestra las aplicaciones desarrolladas, divididas en función del simulador sobre el que trabajen, el STDR de simulación bidimensional y Gazebo, de simulación tridimensional. Dentro de cada uno de ellos se subdivide en aplicaciones desarrolladas sobre código Matlab o sobre Simulink.

En el cuarto capítulo se expresan las conclusiones y los trabajos futuros, especificando las ventajas y las posibles mejoras que se podrían hacer en la toolbox.

La quinta sección está dedicada a los planos, y al ser este un proyecto de software, en ella se muestran los códigos de las diferentes aplicaciones antes expuestas.

En el sexto apartado se encuentra el pliego de condiciones, donde se detallan los medios utilizados para la realización del proyecto y las características de cada uno.

El séptimo capítulo muestra el presupuesto que sería requerido, de forma teórica, para poder llevar a cabo el proyecto, contando con el coste material y el personal.

El octavo y penúltimo capítulo es el manual de usuario, donde se desarrolla punto por punto la manera de ejecutar cada aplicación y lo que hace cada botón de su interfaz.

El libro se cierra con la bibliografía, donde se pueden encontrar referencias a todo aquel material bibliográfico del que se ha hecho uso para su desarrollo.

2 Herramientas utilizadas

En este apartado se muestra una descripción de las herramientas empleadas para realizar el proyecto. Se usará una máquina virtual que cargue LINUX para hacer funcionar Matlab (Windows) y ROS (Ubuntu) en el mismo ordenador, pero se pueden utilizar ordenadores diferentes; En aplicaciones reales el robot llevaría cargado Ubuntu y se manejaría a distancia desde un ordenador.

2.1 Esquema general del sistema

A continuación, se puede ver un esquema de la unión entre MATLAB (sobre Windows) y ROS (sobre Máquina Virtual con Linux) y la creación de aplicaciones.

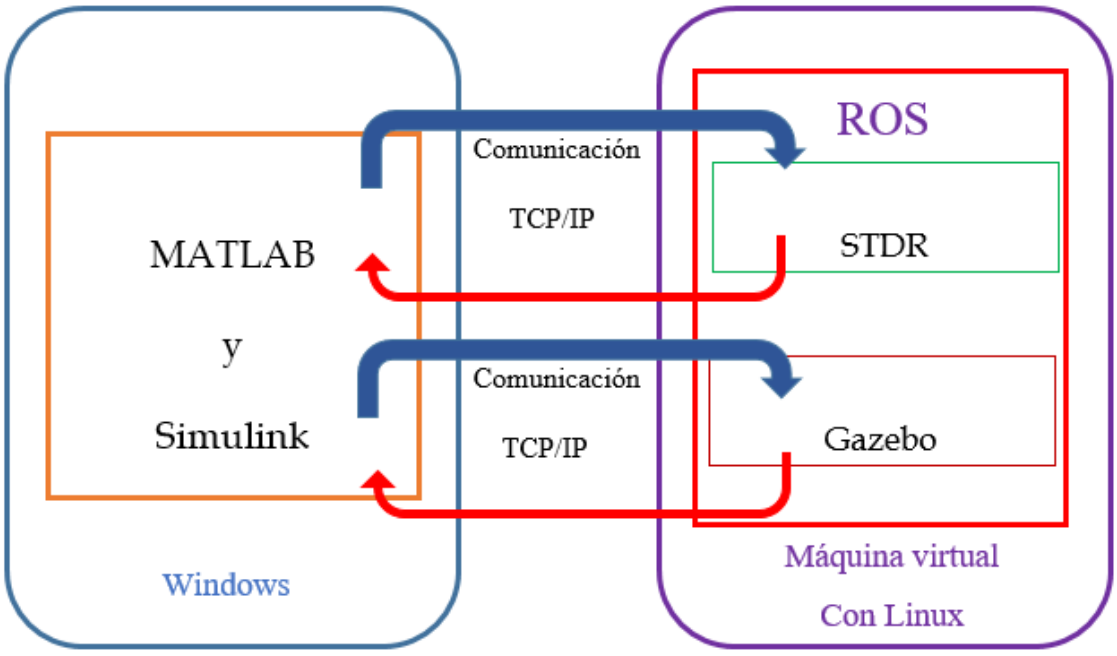

Ilustración 5 Esquema de la unión Matlab-ROS

2.2 Máquina virtual

Para el desarrollo de este proyecto se necesita una máquina virtual debido a que ROS debe correr en LINUX. La gran ventaja de las máquinas virtuales es que permiten ejecutar sistemas operativos diferentes al actual sin la necesidad de realizar particiones de disco duro y de una forma sencilla. Se ha decidido utilizar VMware Workstation 12 player por permitir ejecutar UBUNTU 14.04, versión que se va a utilizar, y ser una opción perfecta para las necesidades del proyecto en cuanto a rendimiento y potencia.

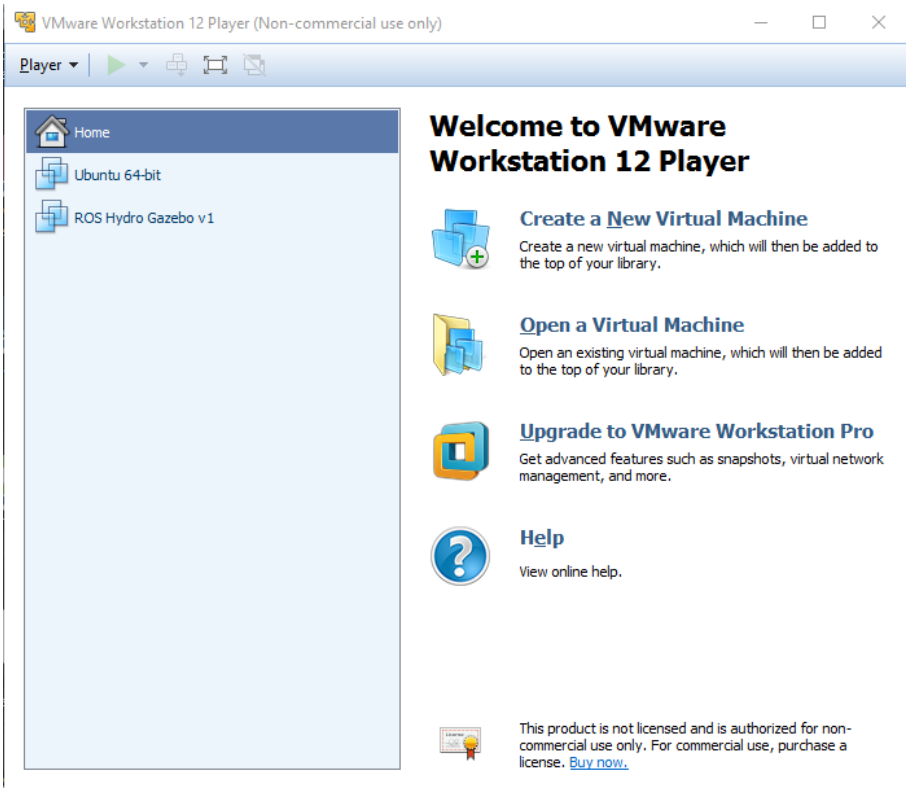

Ilustración 6 Interfaz máquina virtual VMware

Esta versión de VMWare Workstation 12 es gratuita y compatible con Windows 10 y DirectX 10, versiones utilizadas en el presente proyecto. Esta máquina virtual se instala como una aplicación más y permite ejecutar los sistemas operativos que se carguen en ella. Se ha utilizado una versión de Ubuntu 14.04 personalizada en la que ya se ha instalado ROS Índigo y su simulador Gazebo.

2.3 ROS

En este apartado se va a describir la herramienta que ha cambiado el panorama de la robótica. ROS es un framework para la escritura de software de robots. Contiene herramientas y bibliotecas que tienen como objetivo simplificar la tarea de crear aplicaciones complejas y robustas de comportamiento del robot [5].

Nació ante la dificultad de hacer frente a los problemas que se generaban al intentar aunar entornos y herramientas para el desarrollo de aplicaciones para robots. Era necesaria una estructura apoyada por la comunidad y de fácil acceso que permitiera hacer posible esto. ROS aporta todas estas facilidades y los usuarios lo han sabido ver. Ha sido desarrollado con la colaboración de varias instituciones y universidades de reconocido prestigio en robótica. Actualmente soporta muchos tipos de robots y gran variedad de hardware, con una gran popularidad y una comunidad muy activa.

Ilustración 7 Descripción general de ROS

ROS está diseñado para ser lo más distribuido y modular posible, de modo que los usuarios pueden usar tanto o tan poco de ROS como deseen. A nivel bajo está basado en comunicación interproceso (nodos) mediante el paso de mensajes. Al implementar una nueva aplicación robótica, el sistema de comunicación a menudo es una de las primeras necesidades que surgen. Con el sistema de mensajes para la comunicación entre nodos se ahorra tiempo, se mejora la encapsulación y se fomenta la reutilización de código. Las interacciones entre los procesos de ROS se realizan de forma síncrona mediante llamadas a servicios.

Además, ROS proporciona librerías comunes y herramientas que facilitan el desarrollo de las aplicaciones para los diferentes robots. Entre ellas destacan librerías de geometría de los robots, de diagnóstico de llamadas a procesos remotos, transformación de sistemas de coordenadas, de estimación de la posición, de localización, mapeado o de navegación autónoma y herramientas de simulación, como el simulador STDR o Gazebo, que más adelante se desarrollarán. Una de las grandes virtudes es la posibilidad de simular el comportamiento del robot ante nuestra aplicación, mejorar y perfeccionar el código e implementarlo en el robot real con pocos o ningún cambio en el código.

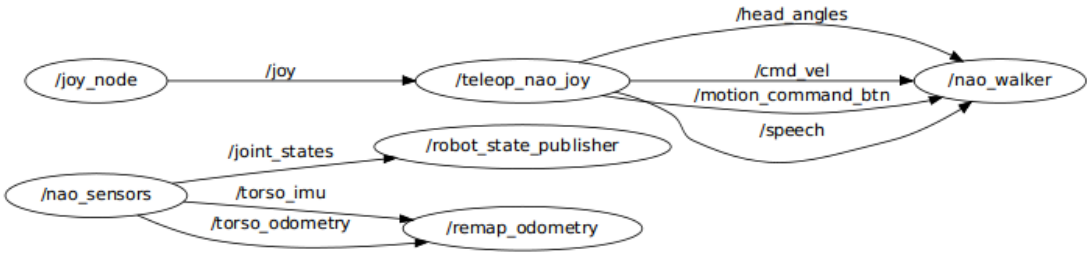

Ilustración 8 Grafo de nodos

ROS se basa en un sistema de grafos *(ver ilustración 8)* en el que hay que tener claros los siguientes conceptos:

- Nodos: Son los programas ejecutables. Facilitan el diseño modular pues cada nodo se compila, ejecuta y maneja individualmente. Se escriben utilizando una librería cliente de ROS en C++ o en Phyton. Los nodos pueden subscribirse o publicar en *topics* y pueden proporcionar o utilizar *servicios.* Con los comandos desde consola asociados a la herramienta *rosnode* se puede obtener información y administrar los nodos del sistema.
- Topics: Son canales de comunicación utilizados por los nodos para comunicarse entre sí. Cada topic transporta un único tipo de mensaje. Con los comandos desde consola asociados a la herramienta *rostopic* se obtiene información y se administran los topics presentes en el sistema.
- Mensajes: Tipo estructurado de datos para la comunicación entre nodos. En cada topic solo se puede transmitir un tipo de mensaje. Se pueden definir nuevos mensajes, aunque existen algunos ya definidos para los tipos de datos más utilizados como lecturas láser, odometría, mapas de ocupación, etc.

- Servicios: Es un modelo de comunicación síncrona entre nodos que se utiliza habitualmente para eventos especiales o que necesiten confirmación. Con la herramienta *rosservice* se obtiene información y se administran los servicios presentes en el sistema.
- ROS Master: Permite que los nodos se localicen entre sí, registra los nombres de topics, servicios, etc. Puede considerarse como el gestor de ROS y ha de estar siempre presente en el sistema. Se ejecuta con la instrucción *roscore.*
- Rosbag: Es un conjunto de herramientas que permiten almacenar y reproducir datos de una ejecución del sistema.
- TF: Es un topic y mensaje especial de ROS que sirve para relacionar los distintos marcos de coordenadas, por regla general siempre está presente en el sistema.
- Rosrun: Permite ejecutar un único nodo.
- Roslaunch: Herramienta que permite lanzar un conjunto de nodos y ficheros de configuración.

En resumen, cada programa en ejecución dentro de ROS se considera un nodo. ROS necesita un nodo maestro en ejecución continuamente que se encargue de coordinar y sincronizar todo el sistema, y este es conocido como *Roscore*. Los nodos se comunican entre sí mediante “topics” o “servicios” siendo la diferencia entre éstos si la comunicación entre nodos es continua (topics) o del tipo llamada-respuesta (servicios). Existen un topic especial que se debe conocer, el topic /tf (transformadas) que está siempre presente y sirve para relacionar los distintos marcos de coordenadas que existen en el sistema.

ROS se organiza en distribuciones anuales ligadas a versiones de Ubuntu. Las distribuciones liberadas en años pares son LTS y tienen soporte de 5 años y las liberadas en años impares tienen un soporte de 2 años. Actualmente la versión más reciente es Kinetic Kame de mayo de 2016 pero para el propósito de éste proyecto se va a utilizar la versión Indigo asociada a Ubuntu 14.04 LTS, disponible desde julio de 2014.

2.3.1 Simulador STDR

El simulador STDR (Simple Two Dimensional Robot) destaca por su simpleza, su objetivo no es ser el simulador más realista si no reducir al mínimo las acciones necesarias para empezar a experimentar. Tiene una interfaz simple y permite modelar y simular diferentes robots y sistemas sensoriales dentro de un entorno de movimiento bidimensional.

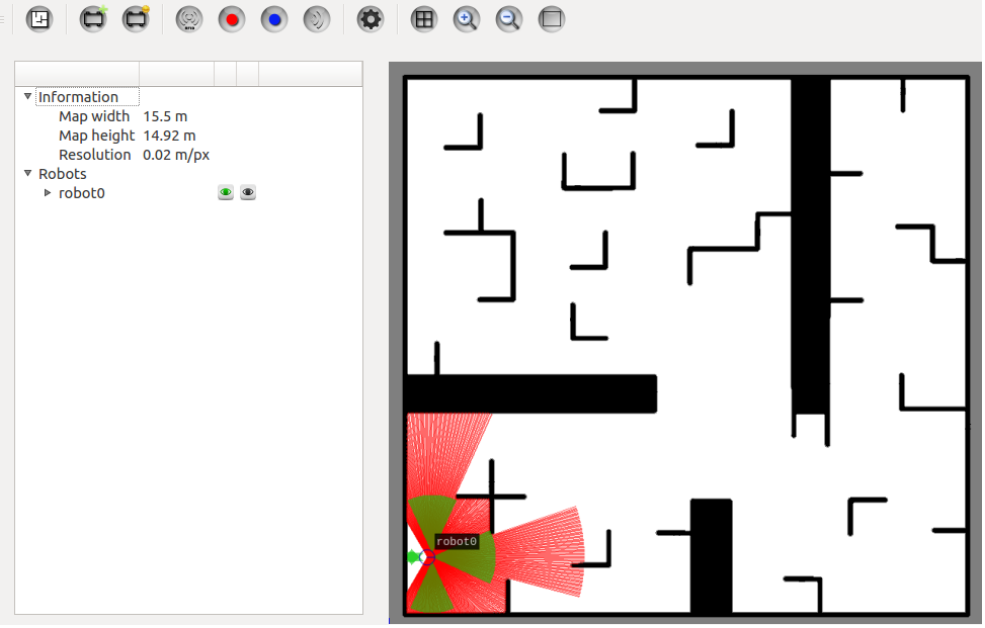

Ilustración 9 Interfaz del simulador STDR

Está creado específicamente para ROS, cada robot y sensor emite una transformada y todas las medidas se publican en los topics de ROS. De esta manera STDR utiliza todas las ventajas de ROS aportando una interfaz gráfica 2D y la posibilidad de cargar mapas de creación propia siempre que sean monocromáticos.

Tiene algunas limitaciones, como que es estrictamente bidimensional, el modelado de los robots es cinemático pero no dinámico y la falta de colisiones entre robots.

2.3.2 Gazebo

Gazebo es un simulador 3D multi-robot para interiores y exteriores, con un motor de físicas y cinemáticas muy potente. La integración entre ROS y Gazebo es proporcionada por un conjunto de plugins que soportan una gran variedad de robots y sensores. Como estos plugins presentan la misma interfaz de mensajes que el resto del ecosistema de ROS, se pueden escribir nodos en ROS perfectamente compatibles con la simulación y el hardware. Se puede desarrollar una aplicación en la simulación y luego implementarla en el robot real sin apenas cambios en el código.

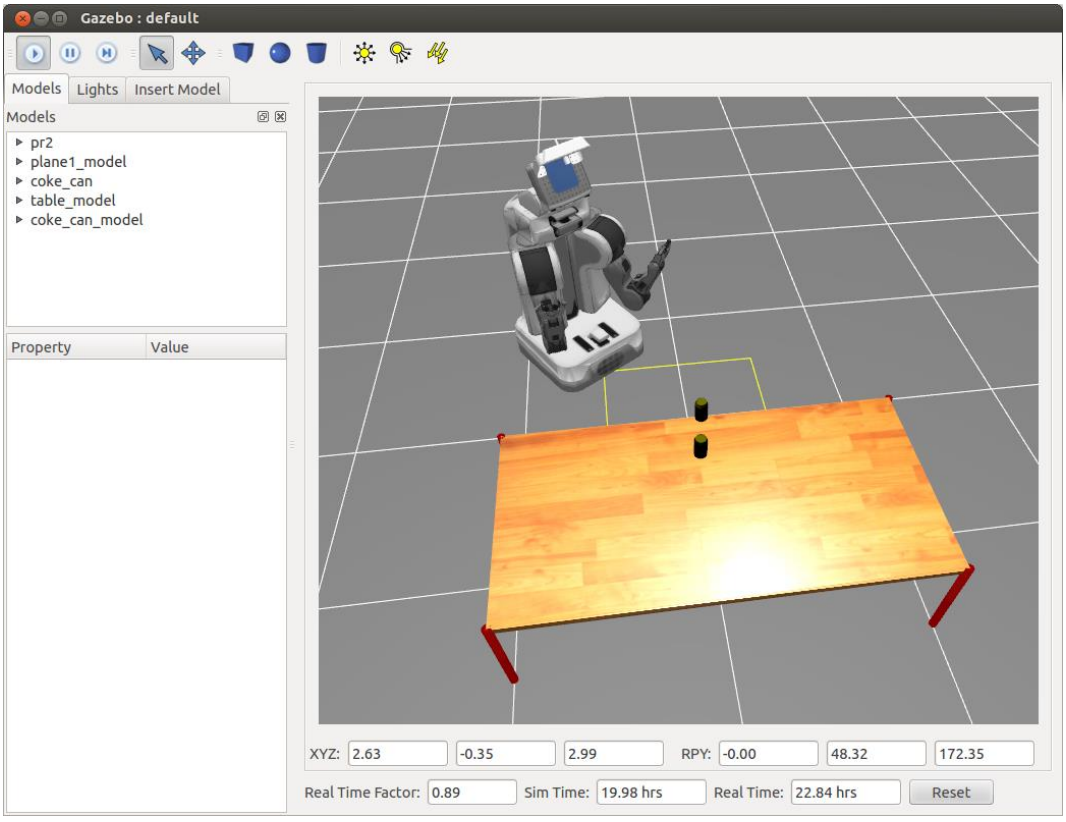

Ilustración 10 Interfaz del simulador Gazebo

2.4 Robotics System Toolbox

En este apartado se va a desarrollar la toolbox que une Matlab y ROS [9 Bibliografía6], sus características, cómo comunicarse con ROS y los algoritmos más importantes que presenta.

2.4.1 Generalidades

Robot System Toolbox es una toolbox de Matlab 2015 que proporciona algoritmos y conectividad con hardware para el desarrollo de aplicaciones autónomas de robótica móvil. Permite representar mapas, planificar trayectorias y seguirlas de forma eficiente con robots autónomos utilizando Matlab o Simulink e integrarlos con los algoritmos de Robotics System Toolbox.

Esta toolbox proporciona una interfaz entre Matlab/Simulink y ROS que permite probar y verificar las aplicaciones de los robots incluidos en ROS y utilizar sus simuladores. Es compatible con la generación de código C++, lo que permite generar un nodo ROS a partir de Matlab e introducirlo en la red de ROS, trabajar con los ROS messages, publicar y subscribir en topics, acceder a los servidores de parámetros de ROS o al árbol de transformadas. En definitiva, poder manejar ROS desde Matlab.

2.4.2 Comunicación con ROS

En este apartado se va a explicar la comunicación entre Matlab y ROS utilizando:

- Un equipo con SO Windows 10 con Matlab R2015a instalado.
- Una máquina virtual (VMWare) con Ubuntu 14.04 y ROS Índigo.
- Comunicación entre los equipos vía Wifi o Ethernet.

Matlab y la Máquina Virtual (MV) pueden ejecutarse en el mismo equipo, o bien en equipos distintos, ya que se van a comunicar utilizando una red local en cualquiera de los dos casos.

En primer lugar, se debe conocer la dirección IP de la máquina virtual; para ello hay que abrir un terminal en Linux y ejecutar el comando "ifconfig".

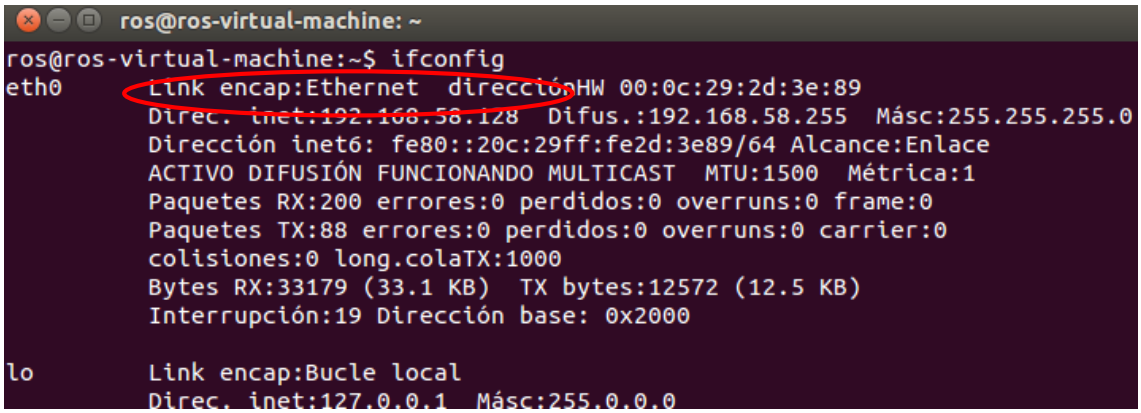

Ilustración 11 Localización de la IP del equipo

ROS utiliza una serie de variables de entorno para establecer la comunicación entre los diferentes nodos que se encuentran conectados a una red. Estas variables tienen información de la dirección IP del equipo, así como su identificador (nombre).

Estas variables se establecerán utilizando el terminal y contendrán la IP obtenida anteriormente.

export ROS_MASTER_URI=http://192.168.58.128:11311

export ROS_HOSTNAME=192.168.58.128

export ROS_IP=192.168.58.128

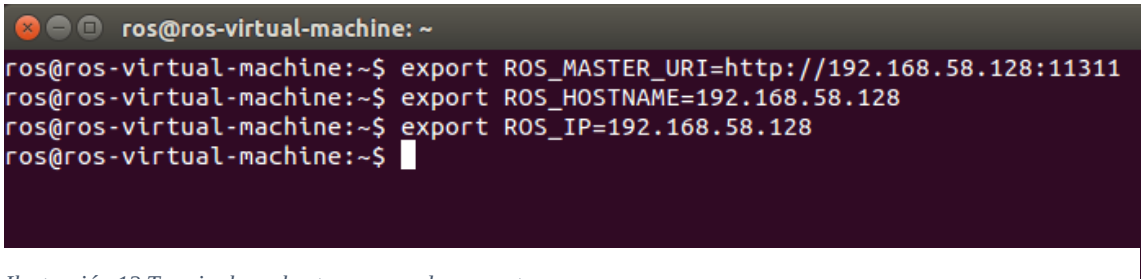

Ilustración 12 Terminal con los tres comandos export

Para no tener que realizar éste proceso cada vez que sea abierto un terminal, se pueden incluir estas tres variables en el archivo *.bashrc* de Linux. Este archivo es ejecutado cuando se abre el terminal, por lo que se ahorra un tiempo importante. Tan solo se debe estar atento por si la IP cambia.

Una vez hecho esto se puede iniciar el proceso Máster de ROS, que será el encargado de comunicar y sincronizar las diferentes aplicaciones del ecosistema de ROS. Para ello se debe ejecutar el comando *roscore* en el terminal.

```
roscore
```

A partir de ahora en este terminal se ejecutará el Máster de ROS, para poder ejecutar otras aplicaciones se debe abrir un nuevo terminal sin cerrar este y volver a poner las variables de entorno, a no ser que se hayan incluido en el archivo *.bashrc*.

ROS ya está en funcionamiento por lo que solo queda unirlo a Windows, para ello Matlab tiene el comando *rosinit* el cual se debe ejecutar con la IP de la máquina virtual de la siguiente manera:

```
rosinit ('192.169.58.128')
```

Tras ejecutar esta línea ya debería estar conectado ROS con Matlab. Para comprobarlo se le puede pedir a Matlab que muestre la lista de topics con el comando “rostopic list”. Si aparecen los topics que existen en el sistema Linux, todo estará correcto.

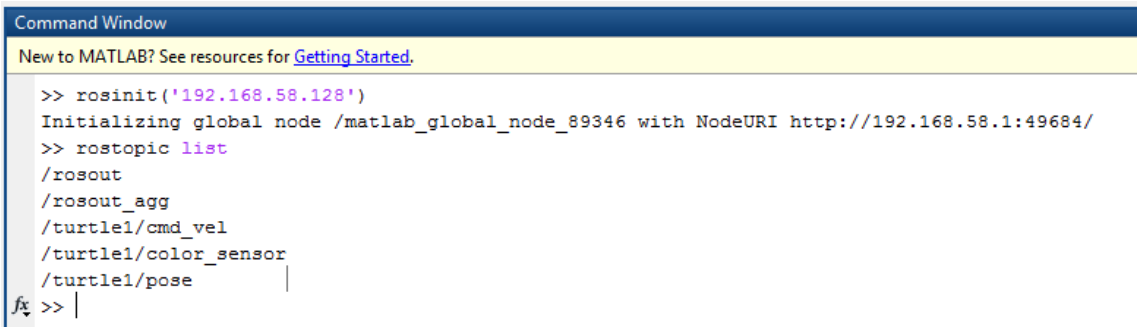

Ilustración 13 Ventana de comandos de Matlab con rosinit ejecutado

Las funciones incluidas en esta toolbox permiten subscribirse a los topics, publicar en ellos, crear y leer mensajes o conocer los topics y nodos existentes en la red. A continuación se detallan los más importantes:

- rossubscriber: Permite subscribirse a los mensajes de un topic, es decir, leerlos.
- rospublisher: Permite publicar mensajes en un topic.
- rosmessage: Crea mensajes de ROS.
- receive: Recibe nuevos mensajes de ROS.
- rosnode: Muestra información sobre los nodos de la red de ROS.
- rosservice: Muestra información sobre los servicios de la red de ROS.
- rostopic: Muestra información sobre los topics de la red de ROS.
- rosinit: Conecta el nodo de Matlab con la red de ROS.
- rosshutdown: Cierra el nodo de Matlab en la red de ROS.

Con estas funciones principales y otras de uso menos frecuente se puede interactuar con el resto de nodos y topics igual que si se estuviera en ROS.

2.4.3 Algoritmos de navegación

Los algoritmos de la toolbox Robotics System en su versión para Matlab 2015a están centrados en ayudar en el desarrollo de aplicaciones de robótica móvil. Se pueden crear mapas de entornos usando rejillas de ocupación binaria (Binary Occupancy Grid), desarrollar la planificación de una ruta para el robot en un entorno determinado o controlar el paso por unos puntos de referencia. En las siguientes líneas se desarrollan los algoritmos más importantes.

a) Binary Occupancy Grid (Rejillas de ocupación)

Las “rejillas de ocupación” [7] se utilizan para representar el espacio de trabajo de un robot en forma de cuadrícula discreta. La información del entorno puede ser recogida por los sensores del robot en tiempo real o puede ser cargada si es conocida previamente. Se

utilizan comúnmente para encontrar obstáculos en aplicaciones de robótica, telémetros láser, cámaras o sensores de profundidad.

Las rejillas de ocupación se usan en los algoritmos robóticos como planificación de rutas. Se usan en las aplicaciones de mapas para integrar la información de los sensores en un mapa discreto, en la planificación de rutas para encontrar rutas sin colisiones, y para localizar robots en un entorno conocido. Se pueden crear mapas de diferentes tamaños y resoluciones para ajustarse a una aplicación específica.

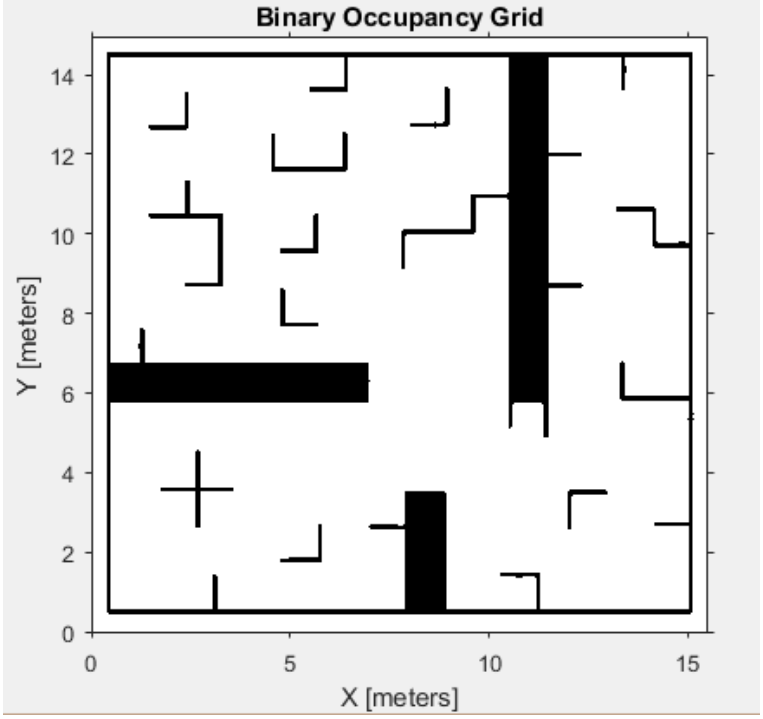

Ilustración 14 Mapa formado por una rejilla de ocupación binaria

Las rejillas de ocupación en 2D tienen dos representaciones:

- Rejillas de ocupación binaria
- Rejillas de ocupación probabilísticas

La rejilla de ocupación binaria usa valores verdaderos para representar el espacio de trabajo ocupado (obstáculos) y valores falsos para representar el espacio de trabajo libre. La rejilla muestra dónde están los obstáculos y si un robot es capaz de moverse a través de esos espacios. Si el tamaño de la memoria es un factor en la aplicación, se debería usar una rejilla de ocupación binaria.

Una rejilla de ocupación probabilística usa valores de probabilidad para crear un mapa de representación más detallado. Esta representación es el método preferido para usar rejillas de ocupación. Cada celda en la rejilla de ocupación tiene un valor representando la probabilidad de ocupación de esa celda. Los valores próximos a 1 suponen una alta probabilidad de que esa celda contenga un obstáculo, mientras que los valores próximos a 0 suponen que no hay ocupación, y por tanto que no hay obstáculo. Los valores de probabilidad pueden aportar una mayor certeza sobre los objetos y mejorar la actuación de ciertos algoritmos.

Las rejillas de ocupación binaria y probabilísticas comparten varias propiedades y detalles algorítmicos. Las coordenadas Grid y World son aplicables a ambos tipos de rejillas de ocupación. El método de inflado es también aplicable a ambas rejillas, pero cada rejilla lo utiliza de manera diferente.

Coordenadas World y Grid

Cuando se trabaja con las rejillas de ocupación en Matlab se pueden usar tanto las coordenadas World como las Grid.

El marco de referencia absoluta en el cual el robot opera se denomina marco World en la rejilla de ocupación. La mayoría de las operaciones de Robotics System Toolbox se llevan a cabo en el marco World, y es la selección por defecto cuando se usan las funciones de Matlab en esta toolbox. Las coordenadas World se usan como un marco de coordenadas absolutas con un origen fijo, y a los puntos se les puede fijar cualquier resolución. Sin embargo, todas las localizaciones se convierten en coordenadas Grid debido al almacenaje de datos y a los límites de resolución del propio mapa.

Las coordenadas Grid definen la resolución real de la rejilla de ocupación y las localizaciones de los obstáculos. El origen de las coordenadas Grid está localizado en la esquina superior izquierda de la rejilla, teniendo la primera localización el índice (1,1). Sin embargo, la propiedad de la rejilla de ocupación *GridLocationInWorld* en Matlab define la esquina inferior izquierda de la rejilla en coordenadas World. Cuando se crea un objeto de la rejilla de ocupación, propiedades como *XWorldLimits* y *YWorldLimits* están definidas por las entradas *width, height,* y *resolution*. Estas figuras muestran una representación visual de dichas propiedades, así como la relación entre las coordenadas World y Grid.

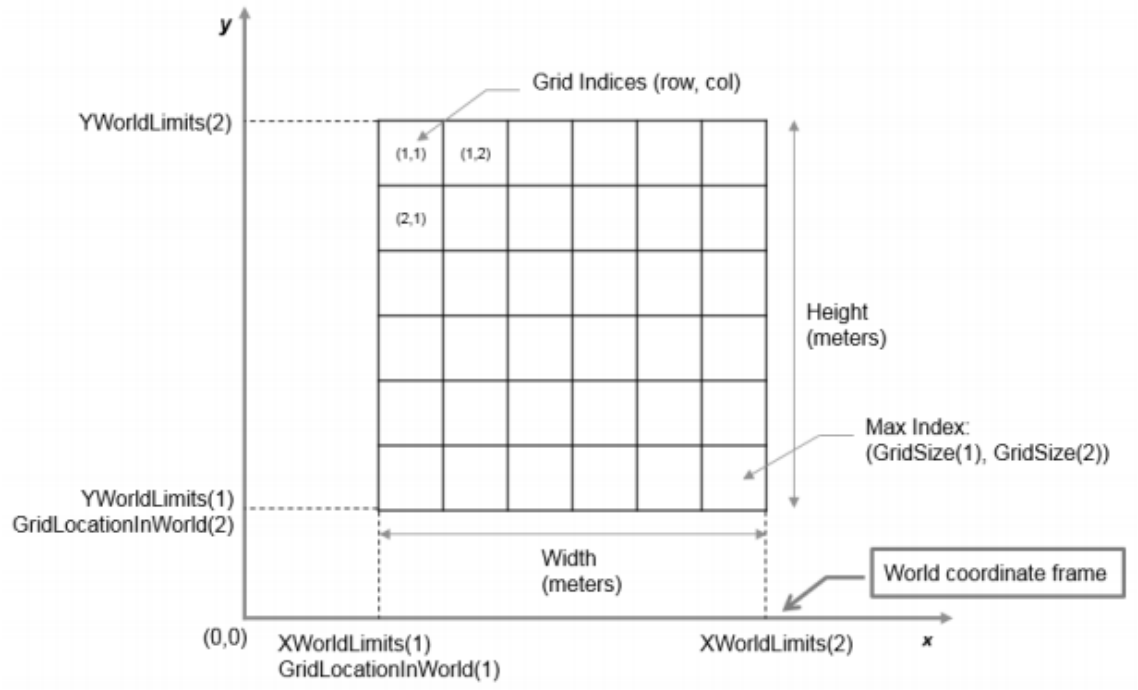

Ilustración 15 Coordenadas World y Grid

Inflado de las coordenadas

Tanto la rejilla de ocupación binaria como la normal tienen una opción de inflado de objetos. Esto se usa para añadir un factor de seguridad entre los obstáculos y el robot. El método *inflate* de un objeto de la rejilla de ocupación convierte el radio especificado al número de celdas rodeadas por la resolución multiplicada por el radio. Cada algoritmo usa este valor de celda de forma separada para modificar los valores sobre los obstáculos.

La función *robotics.BinaryOccupancyGrid* permite crear la rejilla de ocupación con valores binarios. Se puede cambiar el ancho, largo o la resolución de la rejilla. Una vez creado el mapa existen diversas funciones para interactuar con el:

- *getOccupancy:* Obtiene el valor de ocupación para una o más posiciones.
- *setOccupancy:* Establece el valor de ocupación para una o más posiciones.
- *grid2world:* Convierte los índices de la rejilla en coordenadas World.
- *world2grid:* Convierte coordenadas World en los índices de la rejilla.
- *inflate:* Infla cada celda ocupada de la rejilla.
- *show:* Muestra el mapa con sus valores ocupados.
- *readBinaryOccupancyGrid:* Este algoritmo permite leer un mapa mediante la lectura de los datos incluidos en un ROS message del tipo */nav_msgs OccupancyGrid.* Todos los datos del mensaje con valores mayores o iguales a un

valor umbral, el cual puede ser configurado, se consideran 1, zona ocupada, en el mapa. Todos los demás valores, incluidos los desconocidos se consideran zona desocupada, 0, en el mapa.

- *writeBinaryOccupancyGrid:* Escribe los valores de ocupación y otra información para el *ROS message* del mapa.

b) Probabilistic Roadmaps (PRM)

Un PRM [8] es un gráfico de red de las posibles rutas en un mapa determinado basado en los espacios libres y ocupados. La clase *robotics.PRM* genera aleatoriamente nodos y crea conexiones entre ellos basada en los parámetros algorítmicos PRM. Los nodos están conectados en base a la localización específica del obstáculo en el mapa, y en la variable *ConnectionDistance*. Puedes personalizar el número de nodos, *NumNodes*, para ajustar la dificultad del mapa a la necesidad de encontrar la ruta más eficiente. El algoritmo PRM usa la conexión entre nodos para encontrar una ruta sin obstáculos desde un origen hasta un destino.

Cuando se crea o se actualiza la clase *robotics.PRM*, las localizaciones de los nodos son generadas aleatoriamente, lo cual puede afectar a la ruta final entre múltiples iteraciones. Esta selección de nodos se da cuando se especifica un mapa inicial, se cambian los parámetros o se realiza una actualización.

Selección del número de nodos

Se usa la propiedad *NumNodes* para especificar el número de puntos, o nodos, en el mapa. Se emplea la propiedad *ConnectionDistance* como un límite de la distancia entre nodos. El algoritmo conecta todos los puntos que no tienen obstáculos bloqueando la ruta directa entre ellos.

Aumentando el número de nodos se puede aumentar la eficiencia de la ruta, generando más rutas factibles. Sin embargo, la dificultad incrementada aumenta consecuentemente el tiempo de computación. Para obtener una buena cobertura del mapa se pueden necesitar un alto número de nodos. Debido a la localización aleatoria de los nodos, algunas áreas del mapa pueden no tener los nodos suficientes para conectarse con el resto del mapa.

Selección de la distancia de conexión

La propiedad *ConnectionDistance* supone un límite máximo para los puntos que están conectados en la ruta del mapa. Cada nodo está conectado a todos los nodos dentro de esta distancia máxima de conexión y sin obstáculos entre ellos. Reduciendo la distancia de conexión se puede limitar el tiempo de computación y simplificar el mapa. Sin embargo, una distancia reducida limita el número de rutas disponibles entre las que encontrar una ruta complete sin obstáculos. Cuando se trabaja con mapas simples se pueden usar distancias mayores en las conexiones y con un menor número de nodos para incrementar la eficiencia. En mapas complejos, con muchos obstáculos, un mayor número de nodos con una menor distancia entre los nodos incrementa la probabilidad de encontrar una solución favorable.

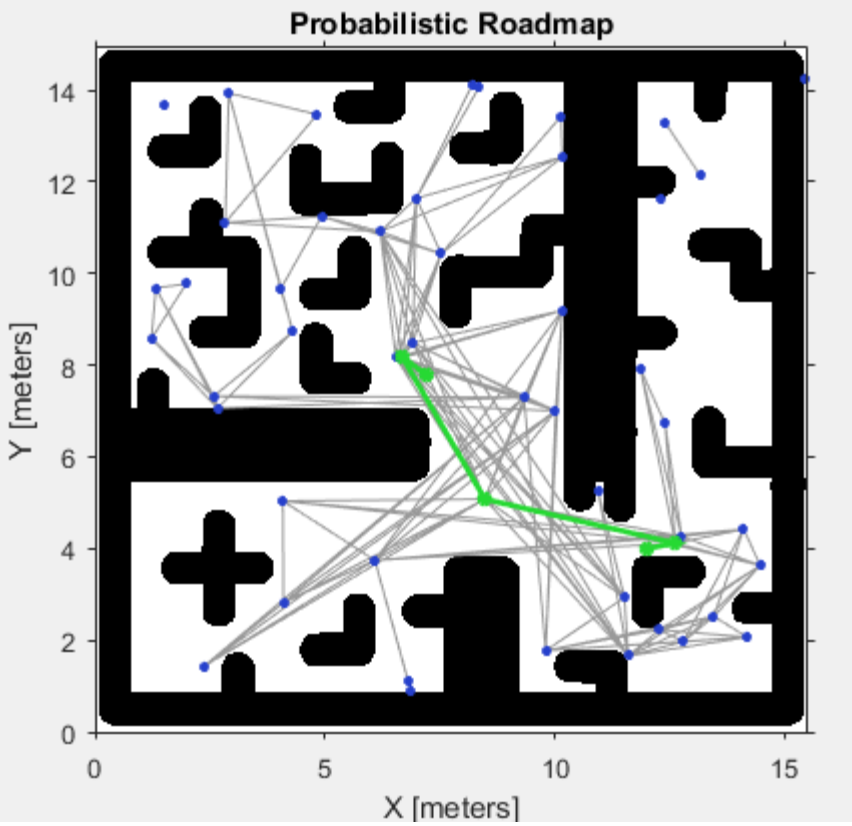

Ilustración 16 Mapa con la ruta elegida

Tras definir la ubicación de inicio y la final, para encontrar un camino libre de obstáculos usando la red de conexiones utiliza la función *FindPath*. Esta función devuelve una matriz con los puntos de la hoja de ruta, pero si no encuentra el camino devuelve una matriz vacía.

PRM tiene diferentes funciones que se utilizan relacionadas con el:

- *findpath:* Busca el camino entre un punto inicial y uno final en la hoja de ruta.
- *show:* Permite mostrar el mapa, la hoja de ruta y el camino entre puntos.
- *update:* Crea o actualiza la hoja de ruta.

c) Controlador de trayectoria "Pure Pursuit"

PurePursuit es un algoritmo de seguimiento de trayectoria [9]. Se calcula la velocidad angular que mueve el robot desde su posición actual hasta llegar a un cierto punto de preanálisis frente al robot. La velocidad lineal se asume constante, por lo tanto, se puede cambiar la velocidad lineal en cualquier punto. El algoritmo entonces mueve el punto de preanálisis de la ruta basándose en la posición actual hasta llegar al último punto de la trayectoria. Se puede pensar esto como el robot persiguiendo constantemente un punto delante de él. La propiedad *LookAheadDistance* decide cuán lejos se coloca el punto de preanálisis. Esta propiedad es explicada con más detalle en una sección posterior.

La clase *PurePursuit* (robotics.PurePursuit) no es un controlador tradicional, pero actúa como un algoritmo de seguimiento de rutas. En la Robotics System Toolbox, se puede crear un controlador *PurePursuit* y definir una lista de puntos de paso. La velocidad lineal deseada y la máxima velocidad angular pueden ser especificadas. Estas propiedades se determinan basándose en las especificaciones del robot. Dada la posición y la orientación del robot como entrada, el objeto *PurePursuit* los utiliza para calcular los comandos de velocidad angular y lineal del robot. Cómo el robot utiliza estos parámetros depende del sistema que se esté utilizando, por lo que se debe considerar como responderá el robot ante ellos.

Sistema de coordenadas de referencia

Es importante entender el marco de coordenadas de referencia usado por el algoritmo *PurePursuit* para las entradas y las salidas. La ilustración inferior muestra el sistema de coordenadas de referencia. Los puntos de paso son coordenadas [x y], que se utilizan para calcular los comandos de velocidad del robot. La posición y orientación (theta) del robot se muestra como [x y theta]. Las direcciones *x* e *y* positivas son hacia la derecha y hacia

arriba respectivamente. La variable *theta* es la orientación angular del robot, medida en sentido anti horario en radianes desde el eje *x*.

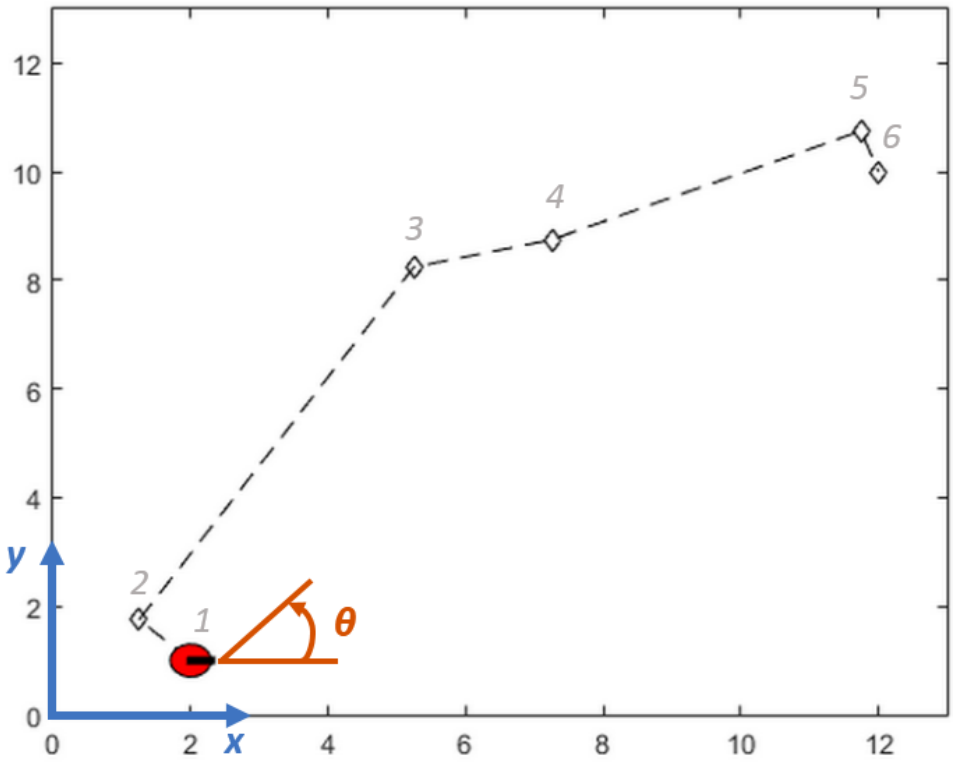

Ilustración 18 Sistema de coordenadas

Distancia de preanálisis

La propiedad *LookAheadDistance* es la propiedad de configuración principal para el controlador *PurePursuit*. La distancia de preanálisis (look ahead distance) es hasta qué punto a lo largo de la trayectoria debe verse desde la ubicación actual para calcular los comandos de velocidad angular. La ilustración 19 muestra el robot y el punto de preanálisis. Como se muestra en esta imagen, hay que tener en cuenta que el camino real no coincide con la línea directa entre los puntos de la trayectoria.

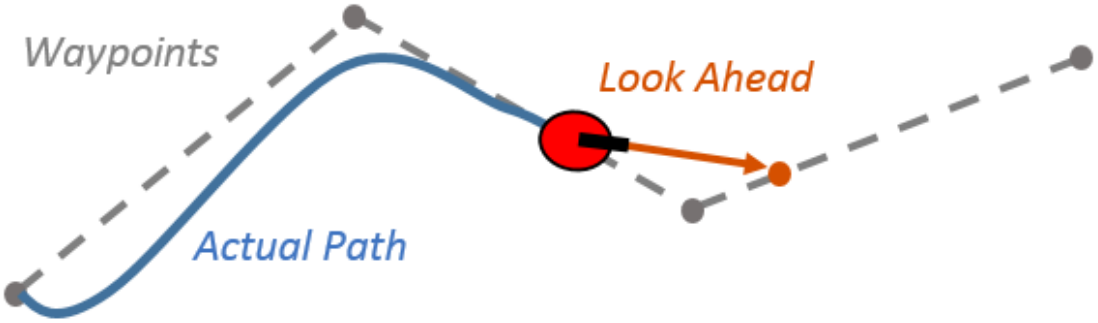

Ilustración 19 Representación del recorrido usando PurePursuit

Cambiar el parámetro *LookaheadDistance* tiene un impacto significativo en el rendimiento del algoritmo. Hay dos objetivos principales, recuperar el camino y mantenerse en él. Cuanto mayor sea la distancia de preanálisis en una trayectoria más suave será para el robot, pero puede hacer que el robot "corte" las esquinas durante el trayecto con grandes curvas.

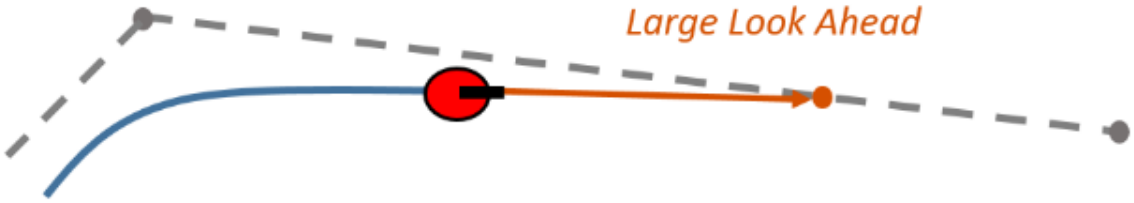

Ilustración 20 Recorrido si el punto a seguir es lejano

Si la distancia es demasiado baja, el robot se mueve rápidamente hacia al camino, lo sobrepasa y termina oscilando a lo largo de la trayectoria deseada por lo que su comportamiento se vuelve inestable, como se ve en la ilustración inferior.

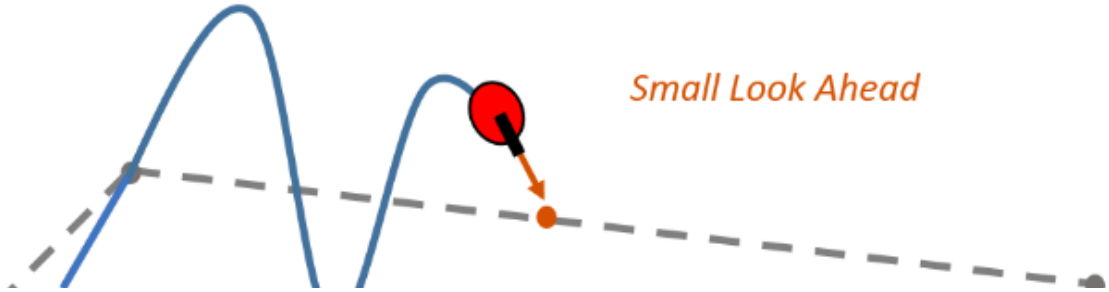

Ilustración 21 Recorrido si el punto a seguir está muy cerca

La propiedad *LookAheadDistance* debe ser configurada para las aplicaciones que hagan uso del controlador *PurePursuit*, diferentes velocidades lineales y angulares afectarán a la respuesta del robot y deben ser consideradas para el seguimiento de la trayectoria.

Este algoritmo presenta algunas limitaciones a tener en cuenta:

- Como se puede observar, el controlador no puede seguir exactamente los caminos entre puntos. Los parámetros deben ajustarse para optimizar el rendimiento y que siga la trayectoria de la forma más razonable.
- Este algoritmo no estabiliza el robot al llegar a cada punto, por lo que para una aplicación se debe implementar un umbral de distancia que detenga el robot cerca de la meta deseada.

Existen diferentes funciones que pueden ser utilizadas con PurePursuit para complementar su uso:

- *clone:* Crea un objeto PurePursuit con las mismas propiedades.
- *info:* Información característica del objeto PurePursuit.
- *step:* Calcula los comandos de control de las velocidades lineal y angular

3 Aplicaciones desarrolladas

Este apartado está dedicado a las aplicaciones que han sido creadas gracias a la toolbox Robotics System de Matlab. Se realizan aplicaciones que muestran las virtudes de los algoritmos de la toolbox sobre el simulador bidimensional STDR de ROS, tanto desde código cómo desde Simulink. También se muestra una aplicación para ver la respuesta y posibilidades de la toolbox ante un simulador tridimensional como es Gazebo.

3.1 Control de robots móviles

En esta sección se van a describir las aplicaciones creadas en Matlab y comunicadas con el simulador bidimensional STDR de ROS gracias a la toolbox Robotics System. Se explica primeramente cómo instalar éste simulador y cómo configurarlo correctamente.

3.1.1 Configuración del simulador STDR

El simulador STDR puede instalarse desde los repositorios de ROS para Ubuntu con la siguiente instrucción:

```
sudo apt-get install ros-indigo-stdr-simulator
```

Una vez instalado se ejecuta con el siguiente comando:

```
roslaunch stdr_launchers server_with_map_and_gui_plus_robot.launch
```

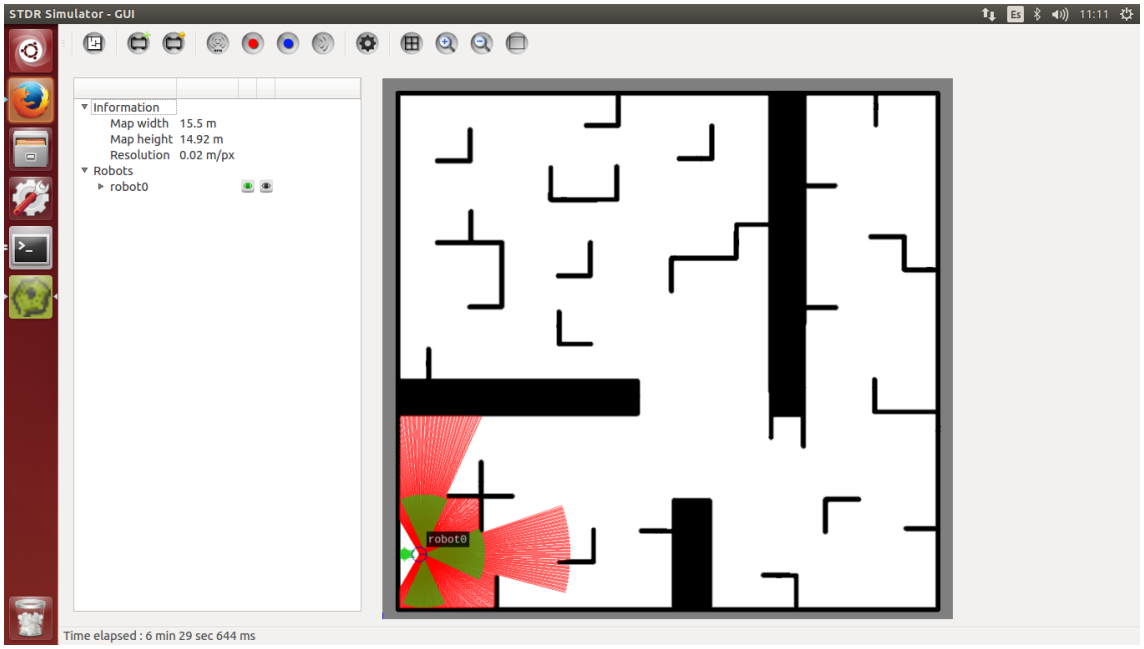

Ilustración 22 Interfaz del simulador STDR

Deberá aparecer la interfaz del simulador STDR con un mapa y un robot cargados como en la figura 22.

Con el simulador ya en marcha se van a dar unas simples directrices para poder poner un mapa o robot de creación propia. Además, existe la posibilidad de cargar algunos ya predefinidos.

Para crear un robot en primer lugar se debe pinchar sobre el botón “Create robot”, señalado en la ilustración 23. Por defecto los robots creados tienen forma circular, pero se pueden definir formas poligonales utilizando la opción “Footprint” para definir los vértices.

Además de la forma del robot se pueden añadir láseres en la sección “láseres” o sonares en la sección “sonares”. Dentro de cada uno se pueden definir varios parámetros como la distancia máxima, el número de rayos para los láseres o la orientación para los sonares. Cuando el robot haya sido creado, se debe guardar dándole el nombre que se prefiera.

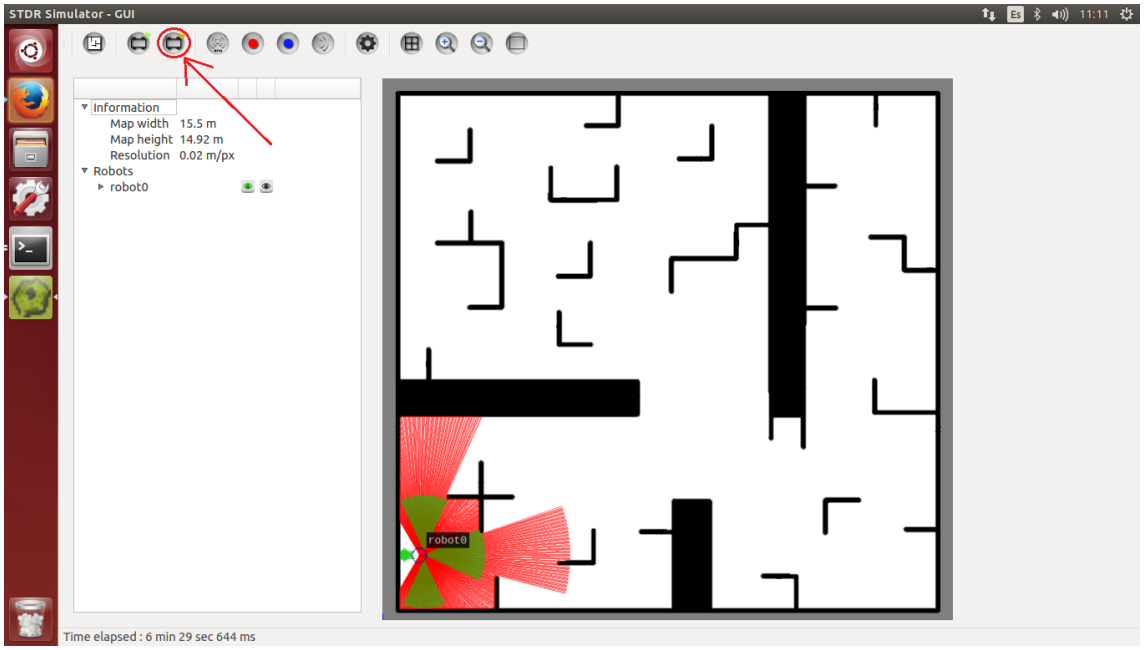

Ilustración 24 Botón para crear un robot en STDR

Para cargar un robot de creación propia o uno predefinido se debe pulsar el botón "Load robot", señalado en la ilustración 24. Una vez cargado el robot puede ser modificado de forma sencilla cambiando el archivo .xml.

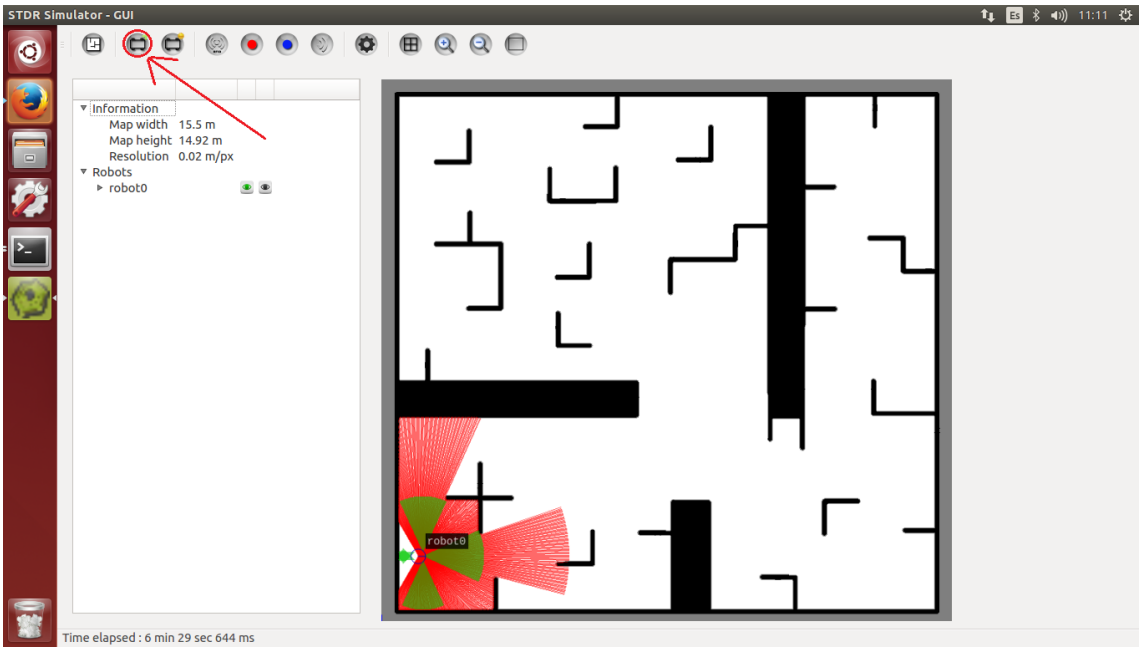

Ilustración 23 Botón para cargar un robot en STDR

Los mapas en el simulador STDR vienen definidos por un archivo de imagen (png) y un archivo de descripción .yaml. El concepto es simple, se define un nivel dentro de la escala de grises y los píxeles por encima serán considerados ocupados (obstáculo) y los píxeles

por debajo serán considerados vacíos (espacio libre). Para cargarlos se debe pulsar sobre el primer botón de la barra de herramientas del simulador STDR.

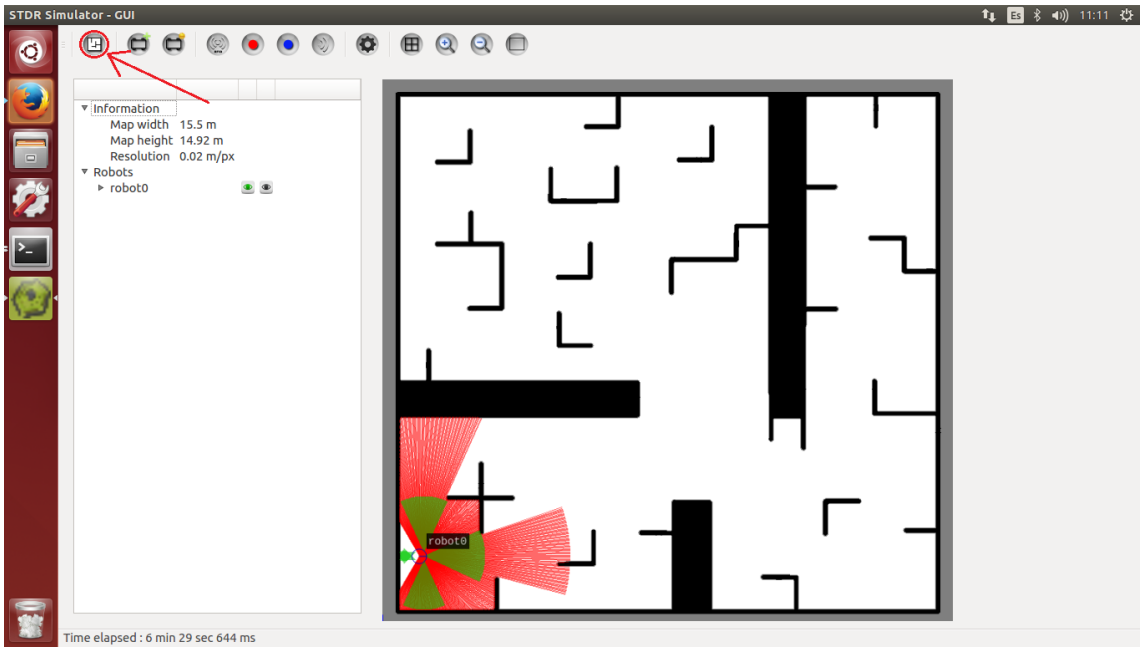

Ilustración 25 Botón para cargar un mapa en STDR

Entre los topics más importantes del simulador STDR se encuentran el topic */map*, que permite conocer las dimensiones y bits de ocupación del mapa que haya sido cargado, y los topics del robot, como */cmd_vel*, topic que al que se envían datos para modificar la velocidad del robot, o */odom* para conocer la posición y orientación del robot. Además, existen topics asignados a cada láser y a cada sonar.

3.1.2 Control desde ficheros .M de Matlab

Las aplicaciones que se desarrollan a continuación utilizan los algoritmos de la toolbox Robotics System de Matlab, clasificadas desde aplicaciones más simples, tales cómo mover un robot una casilla o leer datos del sonar y el láser a planificaciones globales de ruta conociendo el mapa o sin conocerlo. Para todas ellas se va a utilizar la herramienta *Guide* de Matlab, que aporta una pequeña interfaz gráfica a las aplicaciones, y aporta controles tales como menús, barras de herramientas, botones o controles deslizantes, pero cambia ligeramente la escritura de la misma. En cada apartado siguiente se explican los cambios producidos.

Control de movimiento usando la odometría

En esta primera aplicación el objetivo es mover el robot una determinada distancia por el mapa, indicando en la interfaz si se quiere avanzar, girar a la izquierda o a la derecha. La distancia a avanzar puede ser modificada, como se ve en la imagen inferior.

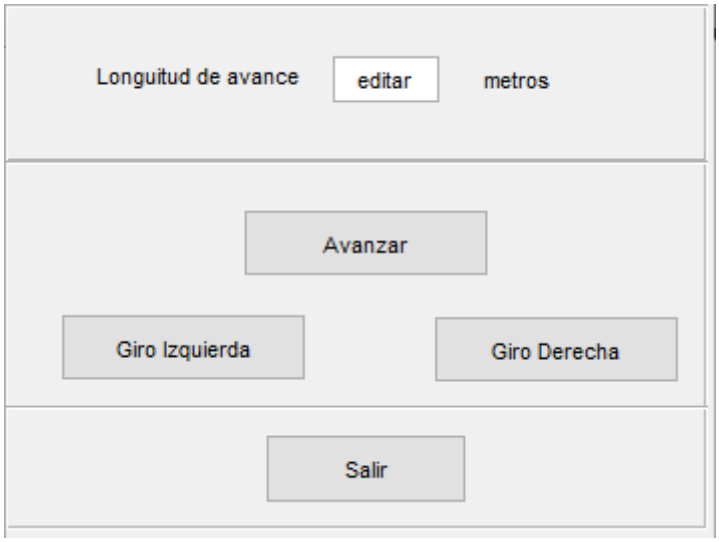

Ilustración 26 Interfaz de la aplicación Control de movimiento

Para hacer que el robot avance o gire hay que hacerse *rospublisher* del topic */robot0/cmd_vel* y crear el mensaje que permite enviarle datos. Además, es necesario conocer la odometría del robot, por lo que hacer *rossubsriber* del topic */robot0/odom* es necesario. En las siguientes líneas de código se muestra cómo se ha hecho.

```
robot = rospublisher('/robot0/cmd_vel');
velmsg = rosmessage(robot);
posicion = rossubscriber('/robot0/odom');
```

Estas variables son incluidas en la función *OpeningFcn* de *Guide*, por lo que se ejecutan al abrir la aplicación. En esta función también se incluye la definición de las variables globales que se utilizan en el resto del programa, así como la longitud inicial de avance, que se ha estipulado en un metro. Esta longitud puede ser cambiada escribiendo el avance deseado en la casilla señalada en la ilustración 27.

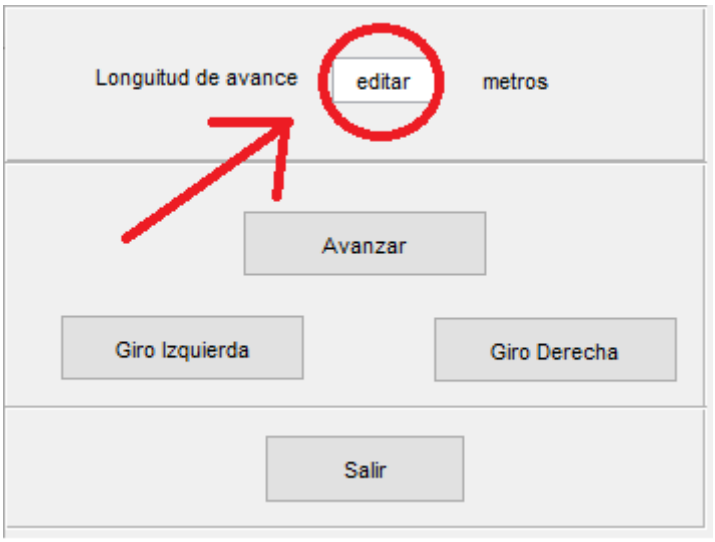

Ilustración 27 Casilla para editar el avance

Al editar la longitud de avance se produce una llamada a la función *LongitudAvance_Callback* de *Guide* que guarda los nuevos datos. Estos datos son guardados como *cadena* por lo que se deben cambiar a *double* para poder ser utilizados. Para poder usar esta distancia en otras funciones de *Guide* se guarda dentro de la variable *handles.LongitudAvance*.

```
function LongitudAvance_Callback(hObject, eventdata, handles)

LdeCasilla=get(hObject,'String');    %Almacenar valor ingresado
LCasilla=str2double(LdeCasilla);     %Transformar de cadena a número
handles.LongitudAvance=LCasilla;     %Almacenar el indentificador
guidata(hObject,handles);           %Salvar datos de la función
```

Para avanzar la distancia establecida lo primero que se hace es conocer la odometría del robot, para ello se utiliza la función *receive* que devuelve la posición en coordenadas X e Y y la orientación en un cuaternio. Este cuaternio debe pasarse a un ángulo en radianes para poder ser utilizado por la aplicación. Pasando el cuaternio por la función de Matlab *quat2angle* se consigue el valor *jaw* que se buscaba.

```
%%%%%%%% Se halla la posicion del robot y su orientacion %%%%%%
posdata = receive(posicion,4);
q =[posdata.Pose.Pose.Orientation.W posdata.Pose.Pose.Orientation.X
posdata.Pose.Pose.Orientation.Y posdata.Pose.Pose.Orientation.Z];
[yaw, pitch, roll] = quat2angle(q);

x = posdata.Pose.Pose.Position.X;
y = posdata.Pose.Pose.Position.Y;

PosicionRobot = [x y];
```

Se localiza en el mapa el punto de destino, se envían los datos de velocidad al robot y se entra en un bucle en el que se actualiza constantemente la distancia al objetivo para saber cuándo parar. Una vez alcanzado el punto de destino, al cual se le da un margen de error de 0.1 metros, debido a los tiempos de ejecución, se envían los datos necesarios al robot para pararlo.

```
%%%%%%%%%% Localizamos en el mapa el punto de destino %%%%%%%%%%%%
x1=x + handles.LongitudAvance*cos(yaw);
y1=y + handles.LongitudAvance*sin(yaw);

robotGoal = [x1 y1];

%distanceToGoal debe ser igual a la distancia de avance %%%%%%%%
distanceToGoal = norm(robotGoal - PosicionRobot);

 %%%%%% Se envía la velocidad al robot para que se mueva %%%%%%%
velmsg.Linear.X = 0.2;
send(robot,velmsg);

%%% Bucle que realimenta con la odomentria hasta llegar al objetivo
    while( distanceToGoal > goalRadius )

        posdata = receive(posicion,4);
         PosicionRobot = [posdata.Pose.Pose.Position.X
         posdata.Pose.Pose.Position.Y];

        %%% Actualizamos la distancia al objetivo %%%
        distanceToGoal = norm(robotGoal - PosicionRobot);

    end

%%%%%% Paramos el robot %%%%%
velmsg.Linear.X= 0;
send(robot,velmsg);
```

Para girar a izquierda o derecha el código es muy similar, variando únicamente el sentido de giro. Lo primero que se debe obtener es la odometría del robot, de igual manera que se explicó anteriormente para avanzar.

Con la odometría ya conocida, se localiza un punto de destino a 90 grados hacienda uso de la función *angdiff* de Matlab, que será el utilizado para girar el robot. Se envían los datos necesarios para hacer girar el robot en el sentido oportuno y se entra en un bucle que actualiza constantemente el ángulo del robot y se compara con el ángulo objetivo. Una vez alcanzado se sale del bucle y se envían los datos para parar el robot. En las líneas de código siguientes se muestra como se ha hecho para realizar el giro a la izquierda.

```
%%%%%%%%%% Localizamos en el mapa el punto de destino %%%%%%%%%%%%
AngFinal=angdiff(0,yaw+(pi/2));
x1=x + 1*cos(AngFinal);
y1=y + 1*sin(AngFinal);

robotGoal = [x1 y1];

velmsg.Angular.Z= 0.1;
send(robot,velmsg);

 %%%%%%% bucle para que gire hasta la orientacion correcta %%%%%%%
      while (yaw <= (atan2(robotGoal(2)-PosicionRobot(2),
      robotGoal(1)-PosicionRobot(1))-0.015) || (yaw >=
      (atan2(robotGoal(2)-PosicionRobot(2),robotGoal(1)-
      PosicionRobot(1))+0.015)))

      posdata = receive(posicion,4);
      q =[posdata.Pose.Pose.Orientation.W
      posdata.Pose.Pose.Orientation.X
      posdata.Pose.Pose.Orientation.Y
      posdata.Pose.Pose.Orientation.Z];

      [yaw, pitch, roll] = quat2angle(q);
      end

velmsg.Angular.Z= 0;
send(robot,velmsg);
```

Al igual que en el bucle de avance, se aporta un margen de error de 0.015 radianes debido a que los tiempos de conmutación hacen que el bucle no se actualice en tiempo real. El código para girar a la derecha es similar, cambiando el punto objetivo, que estará en *yaw-(pi/2)* y dando una velocidad negativa al robot, por lo que girará en sentido horario.

Se ha añadido también un botón para salir de la aplicación, el cual abre un cuadro de diálogo adicional como se ve en la imagen inferior.

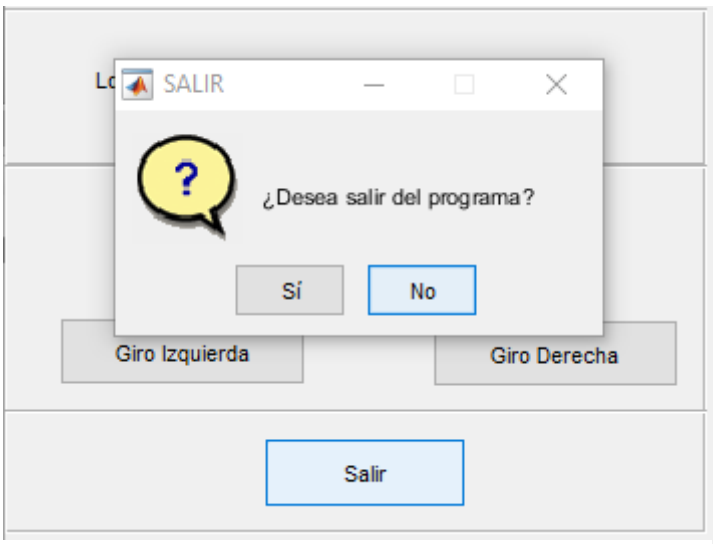

Ilustración 28 Botón y cuadro adicional para salir de la aplicación

Lectura del láser y el sonar

En esta aplicación se va a mostrar cómo leer los sensores del robot haciendo uso de la toolbox de Matlab y de la función *Guide*. Para hacerlo de una forma más visual el robot se moverá hacia delante a una velocidad baja y se representarán los sensores en el mapa a medida que éste avanza.

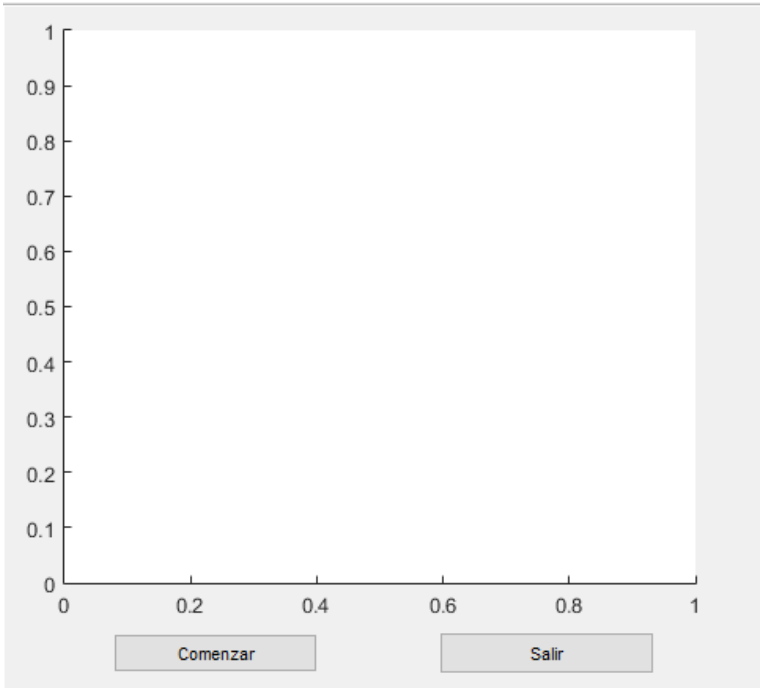

Ilustración 29 Interfaz de la aplicación Lectura de láser y sonar

La interfaz de esta aplicación es sencilla, con un botón para iniciar la lectura de sensores y otro para parar y salir del programa. El cuadro situado encima de los botones mostrará la lectura del láser y el sonar, así como el eje de coordenadas.

Al pulsar el botón *Comenzar*, el código establece la posición de los sonares del robot y se hace *rossubscriber* de los topics de los sensores para poder recibir los datos de cada uno de ellos. Además, se hace *rospublisher* del topic */robot0/cmd_vel* del robot para poder enviarle datos de velocidad. Hay que tener en cuenta que la posición de los sonares varían para cada robot, por lo que se debe configurar de nuevo si este cambia. En el código siguiente se muestra cómo se ha hecho.

```
%Posiciones y orientaciones de los sonares respecto al centro del
robot
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sonares_x=[0.076 0.125 0.15 0.15 0.125 0.076 -0.14 -0.14];
sonares_y=[0.1 0.075 0.03 -0.03 -0.075 -0.1 -0.058 -0.058];
sonares_th=[1.5708 0.715585 0.261799 -0.261799 -0.715585 -1.5708 -
2.53073 2.53073];

%CREAMOS EL SUBSCRIBER DE LOS SENSORES
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%LASER
laser_sub = rossubscriber('/robot0/laser_1',
rostype.sensor_msgs_LaserScan);
laser_msg = rosmessage(laser_sub);

%SONARES
sonar_0_sub = rossubscriber('/robot0/sonar_0',
rostype.sensor_msgs_Range);
sonar_0_msg= rosmessage(sonar_0_sub);

sonar_1_sub = rossubscriber('/robot0/sonar_1',
rostype.sensor_msgs_Range);
sonar_1_msg= rosmessage(sonar_1_sub);

sonar_2_sub = rossubscriber(. . . . .
. . . . . .

%CREAMOS EL PUBLISHER PARA ENVIAR DATOS DE VELOCIDAD
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
vel_pub = rospublisher('/robot0/cmd_vel');
vel_msg = rosmessage(vel_pub);
```

El siguiente paso es darle velocidad al robot para que las lecturas de los sensores vayan cambiando y sea más visual. Para ello se envía un mensaje, *vel_msg*, con la velocidad al topic */robot0/cmd_vel* del que ya se es *rospublisher*.

```
%ENVÍO DE MOVIMIENTOS CON LECTURA DE SENSORES
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

vel_msg.Angular.Z=0;
vel_msg.Linear.X = 0.1;
send(vel_pub,vel_msg);
```

Después se entra en el bucle en el cual se van analizando los datos del láser y de los sonares. Para ello se utiliza la función *receive* con cada sensor y posteriormente se representan. Los datos del láser se pueden dibujar directamente, pero para los datos de los sonares se deben hacer unos cálculos matemáticos, pues se conocen por un lado la

orientación y posición de cada uno de ellos, y por otro la distancia a la que chocan con un obstáculo. En el bucle se dibuja también una circunferencia de radio el del robot para representarlo sobre la figura.

```
while (a==0)

    %Lectura de sensores
    sonar_0_msg = receive (sonar_0_sub);
    sonar_1_msg = receive (sonar_1_sub);
    sonar_2_msg = receive (sonar_2_sub);
    sonar_3_msg = receive (sonar_3_sub);
    sonar_4_msg = receive (sonar_4_sub);
    sonar_5_msg = receive (sonar_5_sub);
    sonar_6_msg = receive (sonar_6_sub);
    sonar_7_msg = receive (sonar_7_sub);
    laser_msg = receive(laser_sub);

    %Representacion gráfica de los datos del láser.
    hold off;
    plot(laser_msg,'MaximumRange',6)

    %Dibujo los datos de los sonares sobre la figura
    x_son(1)=sonares_x(1)+sonar_0_msg.Range_*cos(sonares_th(1));
    y_son(1)=sonares_y(1)+sonar_0_msg.Range_*sin(sonares_th(1));
    x_son(2)=sonares_x(2)+sonar_1_msg.Range_*cos(sonares_th(2));
    y_son(2)=sonares_y(2)+sonar_1_msg.Range_*sin(sonares_th(2));
    x_son(3)=. . . . . .

    hold on;
    plot(x_son, y_son, 'ro')

    %%%%% Se dibuja el robot en el mapa %%%%%
    ang=0:0.01:2*pi;
    xp=0.15 * cos(ang);
    yp=0.15 * sin(ang);
    plot(xp,yp,'g');

end
```

En la Ilustración 30 se puede ver cómo quedan representados los sensores una vez ha sido puesta en marcha la aplicación. Los puntos de choque del láser aparecen en azul formando barreras que ilustran los obstáculos. Esto es debido a las características propias del láser, éste contiene 672 rayos cuyos puntos de choque al ser dibujados, se mezclan con los más cercanos, dando la sensación de dibujar una línea. El punto en el que cada sonar choca

con un obstáculo está representado como una circunferencia de color rojo y la posición del robot, como una circunferencia de color verde y radio el propio del robot.

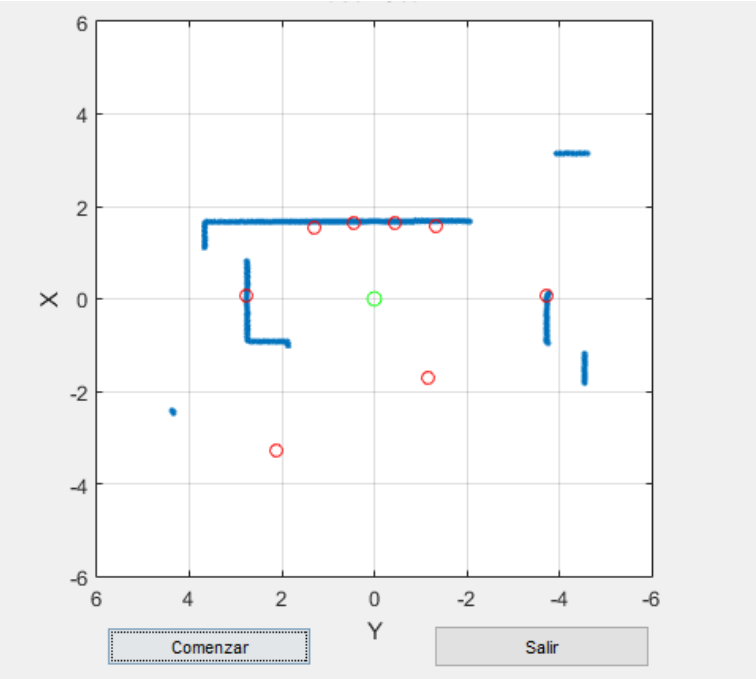

Ilustración 30 Lectura de sensores en funcionamiento

Para poder parar la aplicación se ha puesto un botón de salir, que al igual que en la aplicación anterior abre un cuadro de diálogo como el de la Ilustración 31. En el código de esta función además de cerrar todo se debe enviar de nuevo datos al topic */robot0/cmd_vel* para parar el robot y que no siga avanzando.

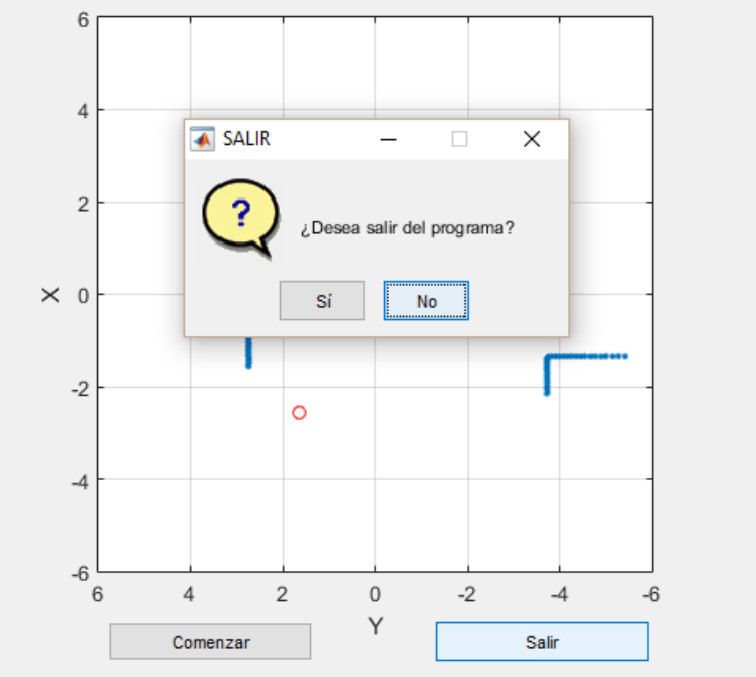

Ilustración 31 Cuadro de diálogo para salir de la aplicación

Control de movimiento con parada de emergencia

Esta aplicación une las dos anteriores, mover el robot en la dirección que se elija por pantalla y lectura de los sensores, utilizada para parar el robot si se fuera a chocar y avisar de ello. En la interfaz de la aplicación además de elegir la dirección a tomar se puede cambiar la distancia de seguridad a la que se avisa de un posible choque.

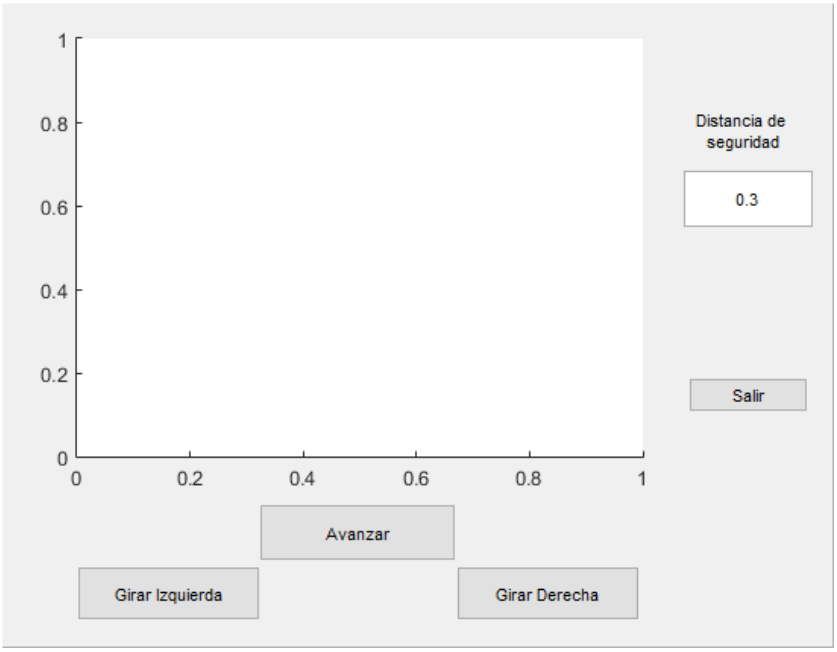

Ilustración 32 Interfaz de la aplicación Control de Movimiento con Parada de Emergencia

En esta ocasión hay que hacerse *rossubscriber* de los topics de los sensores y de la odometría, así como hacerse *rospublisher* de la velocidad para poder mover el robot. Todo ello se hace en la función de *Guide* llamada *Movimiento_Con_Evitacion _Guide_openingFcn* por lo que se ejecutarán al abrir la aplicación. En esta función también se incluyen la posición y orientación de los sónares.

La distancia de seguridad será por defecto de 0,3 metros, pero se puede cambiar en el cuadro mostrado en la Ilustración 33. Al escribir un nuevo dato, se llamará a la función de *Guide DistSecurity_callback* que cambia el formato para poder usar el dato como *double*.

Ilustración 33 Cuadro para cambiar la distancia de seguridad

Los botones de girar a la izquierda o a la derecha ejecutan el mismo código que en la aplicación *Control de Movimiento usando la Odometría*, pero el botón avanzar incluye un nuevo bucle, que compara los datos de los sensores delanteros para saber si el robot mantiene la distancia de seguridad establecida, parándolo en caso de superarse y avisando de ello.

```
if ((sonar_2_msg.Range_ <= handles.DistSecurity) || (sonar_3_msg.
Range_ <= handles.DistSecurity))

        distanceToGoal = 0;
        errordlg('Cuidado, el robot se chocará','Cuidado');
end
```

Este bucle está incluido dentro del bucle que actualiza la posición y analiza si la distancia al objetivo se cumple. En caso de no respetar la distancia de seguridad aparecerá un cuadro de aviso como el de la Ilustración 34.

Ilustración 34 Cuadro de aviso de choque

En esta aplicación en lugar de dibujar el radio del robot, se dibuja el radio de seguridad. Para ello se usa como radio la variable *handles.DistSecurity* de *Guide*. Esta forma de nombrar la variable es la que utiliza *Guide* para utilizar variables globales procedentes de otras funciones, en este caso de la función *DistSecurity_Callback*.

```
ang=0:0.01:2*pi;

xp=handles.DistSecurity * cos(ang);
yp=handles.DistSecurity * sin(ang);

plot(xp,yp,'g');
```

Al igual que en las anteriores aplicaciones existe el botón *Salir*, que mediante un cuadro de diálogo adicional para evitar pulsarlo por error, permite cerrar el programa y parar el robot.

Lectura de mapa de ocupación

Esta sencilla aplicación muestra cómo se dibuja el mapa de ocupación. Para ello hay que hacerse *rossubscriber* del topic */map*. Este topic contiene los datos binarios que componen el propio mapa. Para sacar estos datos en lugar de utilizar la función *receive* se utiliza la función *readBinaryOccupancyGrid* sobre el último mensaje del topic. En el código siguiente se puede ver la función que dibuja el mapa *DibujarMapa_callback*.

```
function DibujarMapa_Callback(hObject, eventdata, handles)

mapa = rossubscriber('/map')
MAP = readBinaryOccupancyGrid(mapa.LatestMessage)
show(MAP)
```

La interfaz de esta aplicación es simple, con un botón para dibujar el mapa, que contiene la función superior y un botón para salir de la aplicación.

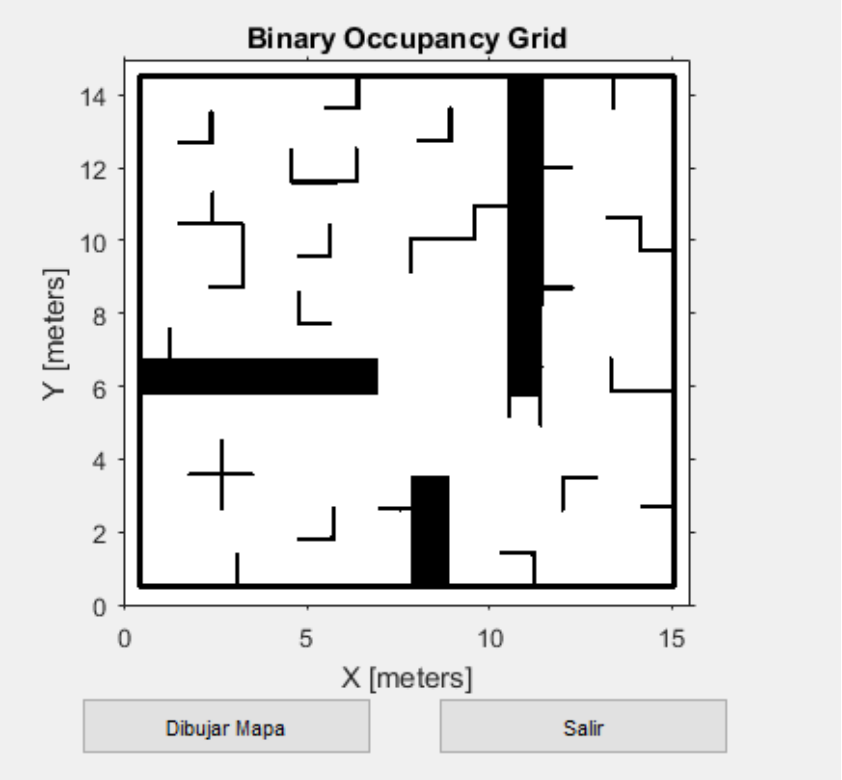

Ilustración 35 Interfaz aplicación Creación de mapa de Ocupación

Planificación global con PRM y planificador local básico

Esta aplicación hace uso del algoritmo *Probabilistic Road Map* explicado en el apartado 2.4.3. Este algoritmo permite, conociendo el mapa y eligiendo por pantalla el punto de destino, dibujar nodos de forma aleatoria y unirlos en el mapa evitando los obstáculos, y utilizando posteriormente la función *findpath*, se puede sacar la matriz de puntos que permite ir desde la posición del robot a un punto de destino elegido.

Para poder utilizar el *PRM* hay que subscribirse al topic */map* para poder conocer el mapa, como se explica en la aplicación anterior, y al topic */robot0/odom* para saber la posición y orientación del robot. También es necesario hacerse *rospublisher* del topic */robot0/cdm_vel* para poder enviar los datos de velocidad necesarios para mover el robot. Esto se hace en la función de *Guide PRM_Basico_Guide_OpeningFcn* y se guardan como variables globales para utilizarlos en el resto de funciones.

```
function PRM_Basico_Guide_OpeningFcn(hObject, eventdata, handles,
varargin)
global mapa robot posicion
mapa = rossubscriber('/map')
robot = rospublisher('/robot0/cmd_vel');
posicion = rossubscriber('/robot0/odom');

handles.output = hObject;

% Update handles structure
guidata(hObject, handles);
```

Una vez conocido el mapa, hay que utilizar la función *inflate* para inflar los obstáculos del mapa, debido a que al utilizar el algoritmo *PRM*, los trayectos que genera podrían pasar demasiado cerca de algún obstáculo con el que el robot puede chocarse. Por ello se infla el mapa con un radio igual al del robot que permita evitar estos choques. El siguiente código muestra cómo se hace.

```
MAP = readBinaryOccupancyGrid(mapa.LatestMessage)

robotRadius = 0.3;
mapInflated = copy(MAP);
inflate(mapInflated,robotRadius)
```

En la ilustración inferior se ve la comparación entre el mapa original, y el mapa inflado, que es el usado para generar las trayectorias con el algoritmo *PRM*.

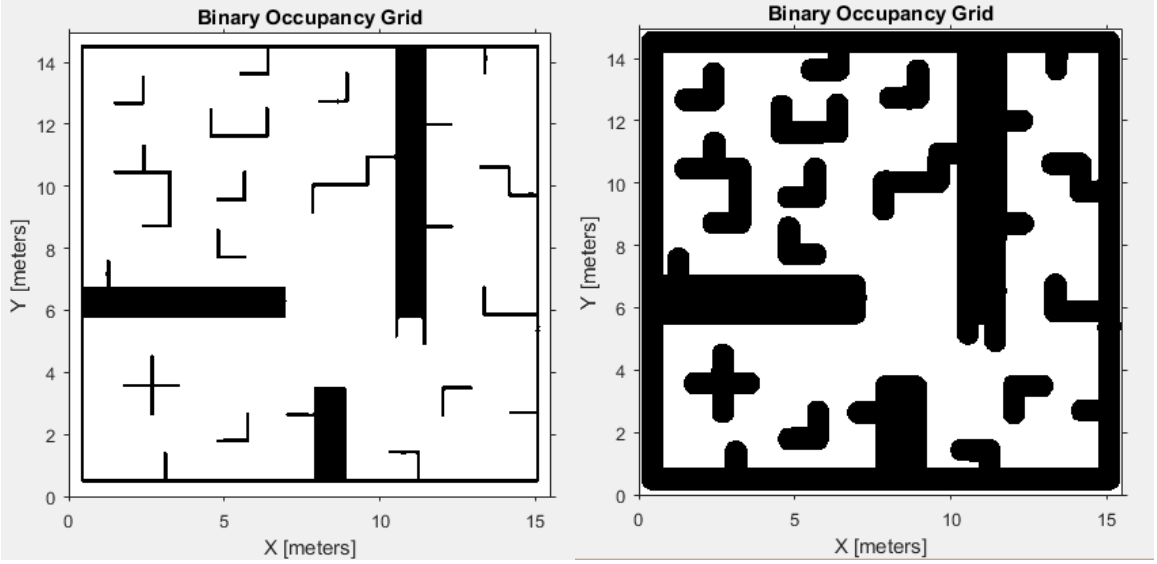

Ilustración 36 A la izquierda, el mapa sin inflar, y a la derecha, el mapa inflado

Con este mapa inflado ya se puede ejecutar el *PRM*. En este caso, al ser un mapa poco complicado se ha optado por configurar el número de nodos a 50 y la distancia máxima entre nodos a 5 metros. En mapas de laberinto o mapas más complejos debe incrementarse el número de nodos y bajar la distancia entre ellos.

```
prm = robotics.PRM(mapInflated);
prm.NumNodes = 50;
prm.ConnectionDistance = 5;
```

La localización inicial se saca de la posición del robot, y se marca en el mapa con una X roja. Para la posición final, la de destino, aparecerá en la interfaz un cuadro de diálogo, cómo el de la ilustración 37, indicando que hay que pinchar en el mapa dicho punto.

Para conseguir esto se hace uso de dos funciones, *questdlg* de *Guide*, para el cuadro informativo, y *ginput* para poder elegir un punto pinchando en un mapa de coordenadas.

```
startLocation = [posdata.Pose.Pose.Position.X
posdata.Pose.Pose.Position.Y]
plot(startLocation(1,1),startLocation(1,2),'+r');

opc=questdlg({'Debe pinchar el punto de destino','¿De
acuerdo?'},'Informacion','Sí','Salir', 'Salir');
if strcmp(opc,'Salir')
   clear,clc,close all;
end

robotGoal=ginput(1);
```

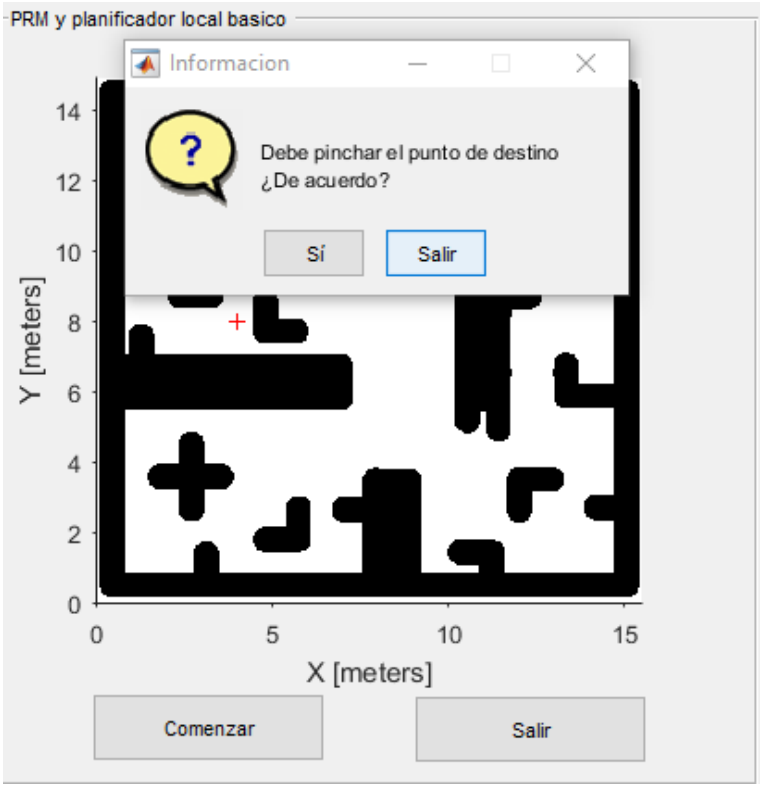

Ilustración 37 Cuadro informativo para elegir punto de destino

Ahora que ya se tienen los puntos de origen y destino, y los nodos unidos, se utiliza la función *findpath* que encuentra el camino más fácil hasta el destino y guarda los puntos en una matriz.

```
path = findpath(prm, startLocation, robotGoal)
```

Si la función *findpath* no consigue encontrar un camino, la matriz saldrá vacía, por lo que se ha incluido un bucle *while* que aumenta el número de nodos y actualiza el *PRM* haciendo uso de la función *update*.

```
%Por si no se ha encontrado el camino y se necesitan mas nodos

while isempty(path)

    prm.NumNodes = prm.NumNodes + 10;

    update(prm);

    path = findpath(prm, startLocation, endLocation);
end
```

Una vez que se tiene la matriz con los puntos a seguir, hay que hacer que el robot los siga de una forma más manual. Para ello se entra en un bucle *for* que recorra los diferentes puntos uno a uno. Primero se orienta el robot en la dirección adecuada, sacando la orientación final respecto a la del robot, girando en el sentido más rápido con la función *angdiff* y actualizando la orientación del robot para pararlo cuando llegue a la orientación correcta.

```
robotGoal = path(a,:);
    angFinal = atan2(robotGoal(2)-robotCurrentPose(2),robotGoal(1)-
robotCurrentPose(1));

    %%%%%%%%%% decidimos en que sentido girar %%%%%%%%%%
      angulo = angdiff(angFinal, yaw);
        if angulo >= 0
         velmsg = rosmessage(robot);
         velmsg.Angular.Z= -0.3;
         send(robot,velmsg);
        end
        if angulo <= 0
         velmsg = rosmessage(robot);
         velmsg.Angular.Z= 0.3;
         send(robot,velmsg);
        end

     %%%%% bucle para que gire hasta la orientacion correcta %%%%
    while (yaw <= (atan2(robotGoal(2)-
robotCurrentPose(2),robotGoal(1)-robotCurrentPose(1))-0.015) || (yaw
>= (atan2(robotGoal(2)-robotCurrentPose(2),robotGoal(1)-
robotCurrentPose(1))+0.015)))

        posdata = receive(posicion,4);
        q =[posdata.Pose.Pose.Orientation.W
posdata.Pose.Pose.Orientation.X posdata.Pose.Pose.Orientation.Y
posdata.Pose.Pose.Orientation.Z];
        [yaw, pitch, roll] = quat2angle(q);
    end

     velmsg.Angular.Z= 0;
     send(robot,velmsg);
```

Con la orientación correcta lo siguiente es avanzar hasta la posición objetivo, entrando en un bucle que actualice la distancia al objetivo para saber cuándo parar. En la ilustración 38 se ve un ejemplo de la interfaz con los nodos y el camino ya trazado.

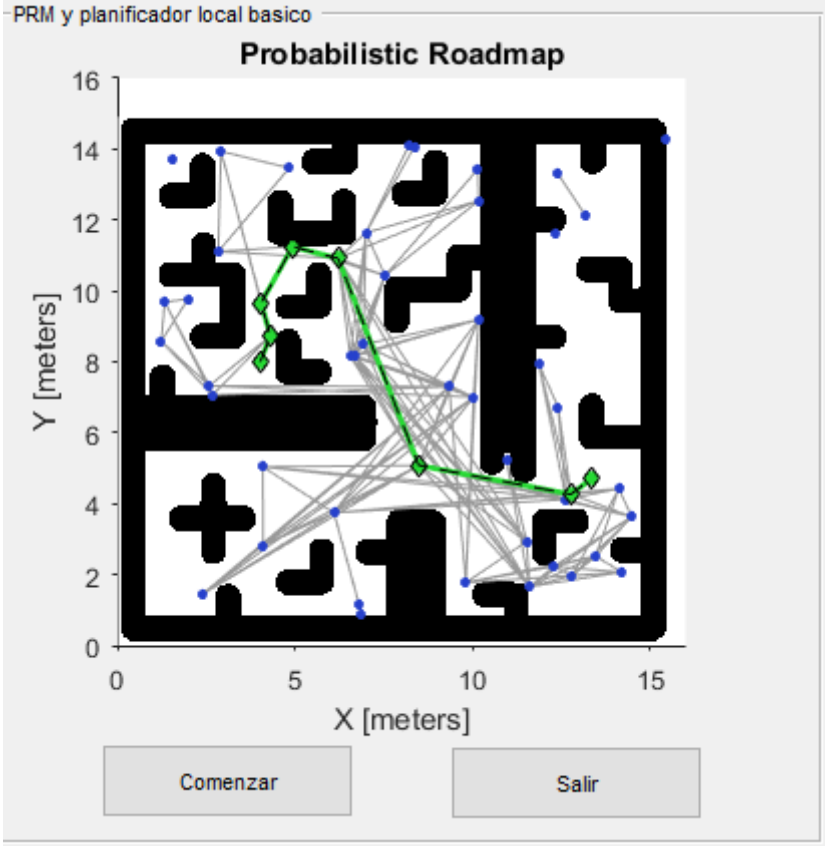

Ilustración 38 Interfaz con PRM y trayecto trazado

Se cuenta con un botón de *Salir* como en anteriores aplicaciones para poder cerrar el programa.

La forma de seguir los puntos es sencilla, girar a la orientación adecuada y seguir en línea recta hasta el siguiente punto. En la próxima aplicación se ve una forma más compleja de hacerlo, utilizando la función *PurePursuit*.

Planificación global con PRM y local con Pure Pursuit

Esta aplicación es parecida a la anterior, pero en lugar de seguir los puntos de forma manual, se siguen haciendo uso del algoritmo *Pure Pursuit*, explicado en el apartado 2.4.3. Este algoritmo controla las velocidades lineal y angular, y modifica esta última según la posición de un punto cercano de la trayectoria, así se consigue que el movimiento sea más realista que avanzar recto a un punto, parar, girar y volver a avanzar, pero también puede producir más errores.

La interfaz de esta aplicación es igual que la anterior, con dos botones principales, uno para dar comienzo al programa, el botón *Comenzar*, y otro para cerrarlo, el botón *Salir*.

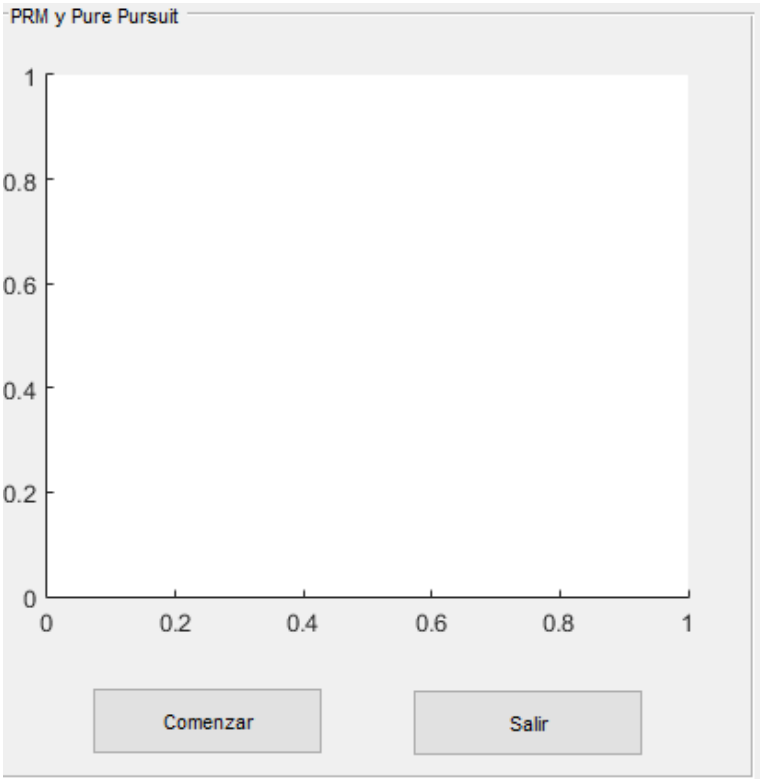

Ilustración 39 Interfaz aplicación PRM y Pure Pursuit

Al avanzar siguiendo un punto cercano, puede que el robot acorte en una esquina del trayecto, provocando que choque con algún obstáculo imprevisto. O puede ocurrir que el punto esté tan cerca que el algoritmo se desestabilice y oscile sobre la trayectoria. Para impedir que ocurran errores, el algoritmo *Pure Pursuit* permite configurar las diferentes variables que lo componen.

La variable más importante es la matriz de puntos que debe seguir el robot, ésta se saca de la misma forma que en la anterior aplicación, hacienda uso primero del algoritmo *PRM* y posteriormente de la función *FindPath*. El nombre de esta variable, y donde hay que meter la matriz de puntos, es *robotics.PurePursuit.Waypoints*.

Otra variable importante es *robotics.PurePursuit.LookaheadDistance*, donde se configura la distancia del punto de la trayectoria a seguir. Es una variable que hay que ajustar con precisión para cada caso, pues una mala configuración puede producir muchos errores.

Las velocidades se configuran en las variables *robotics.PurePursuit.MaxAngularVelocity* y *robotics.PurePursuit.DesiredLinearVelocity*, en la primera se establece la velocidad angular máxima, con un valor bajo de ésta el robot necesitará más espacio para realizar una curva, y con un valor alto el giro será más brusco. La segunda variable determina la velocidad lineal elegida, la cual no varía durante el trayecto.

```
%%% Configuracion de parametros de Pure Pursuit %%%%
PP=robotics.PurePursuit;
PP.Waypoints=path;
PP.MaxAngularVelocity=0.6;
PP.LookaheadDistance=0.2;
PP.DesiredLinearVelocity=0.2;
```

El problema que tiene este algoritmo, es que no envía la velocidad al robot ni sabe cuándo parar, por lo que hay que añadir un bucle *while* que actúe mientras la distancia al objetivo sea mayor a una distancia dada. Dentro del bucle se sacan las velocidades con la función *step* de Matlab y se envían al robot. Una vez que el objetivo está dentro de la distancia elegida, se sale del bucle y se para el robot.

```
while (distanceToGoal>0.5)
    [v,w]=step(PP,robotCurrentLocation);
    velmsg.Linear.X = v;
    velmsg.Angular.Z= w;
    send(robot,velmsg);

    posdata = receive(posicion,4);
    current_orient=quat2eul([posdata.Pose.Pose.Orientation.W
posdata.Pose.Pose.Orientation.X posdata.Pose.Pose.Orientation.Y
posdata.Pose.Pose.Orientation.Z]);
    robotCurrentLocation = [posdata.Pose.Pose.Position.X
posdata.Pose.Pose.Position.Y current_orient(1)];
    distanceToGoal = norm(robotCurrentLocation(1:2) - robotGoal);
end

velmsg.Linear.X = 0;
velmsg.Angular.Z= 0;
send(robot,velmsg);
```

Planificación global con PRM y mapa desconocido

Esta aplicación hace uso de todas las anteriores y es, por tanto, la más compleja. El objetivo es seguir una trayectoria hacia un punto determinado por pantalla, pero sin tener conocimiento del mapa. Para ello se hace uso del *PRM* para conocer la trayectoria, del *PurePursuit* para seguirla, de los sónares para evitar que choque, y del láser para dibujar el mapa cuando el robot encuentre un obstáculo.

Cuándo los sónares detectan un obstáculo demasiado cercano, el robot para y se dibuja el mapa a su alrededor. Una vez dibujado se vuelve a ejecutar el PRM y se traza una nueva trayectoria evitando los nuevos obstáculos.

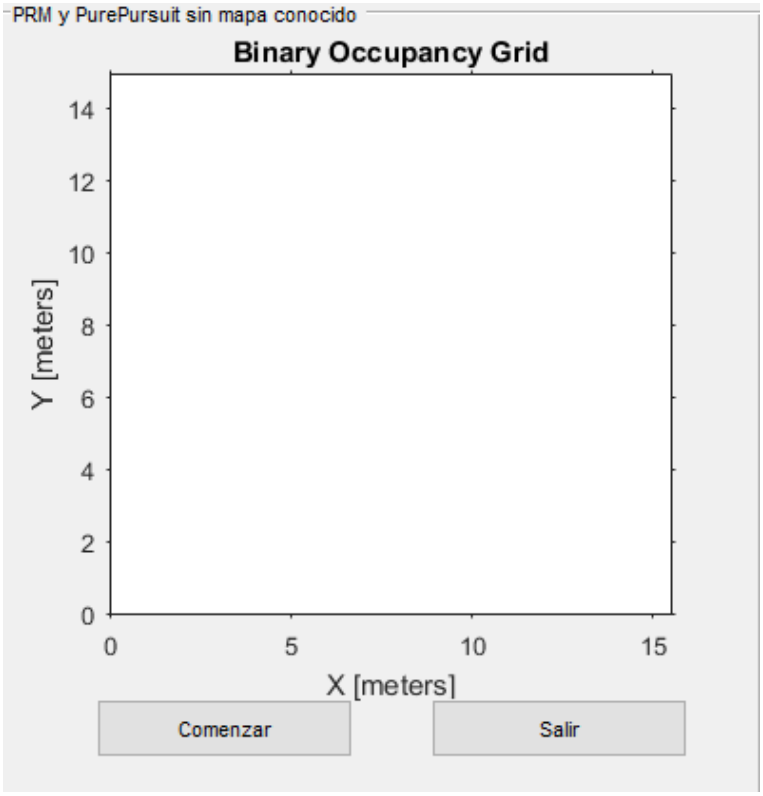

Ilustración 40 Interfaz aplicación PRM y PurePursuit sin mapa conocido

La interfaz de la aplicación es igual a las anteriores, como se ve en la Ilustración 40. Lo diferente empieza cuando se pulsa el botón *Comenzar*. Tras informar de que hay que pinchar en el destino, se ve que el mapa tan solo tiene dibujados los obstáculos que el láser del robot detecta.

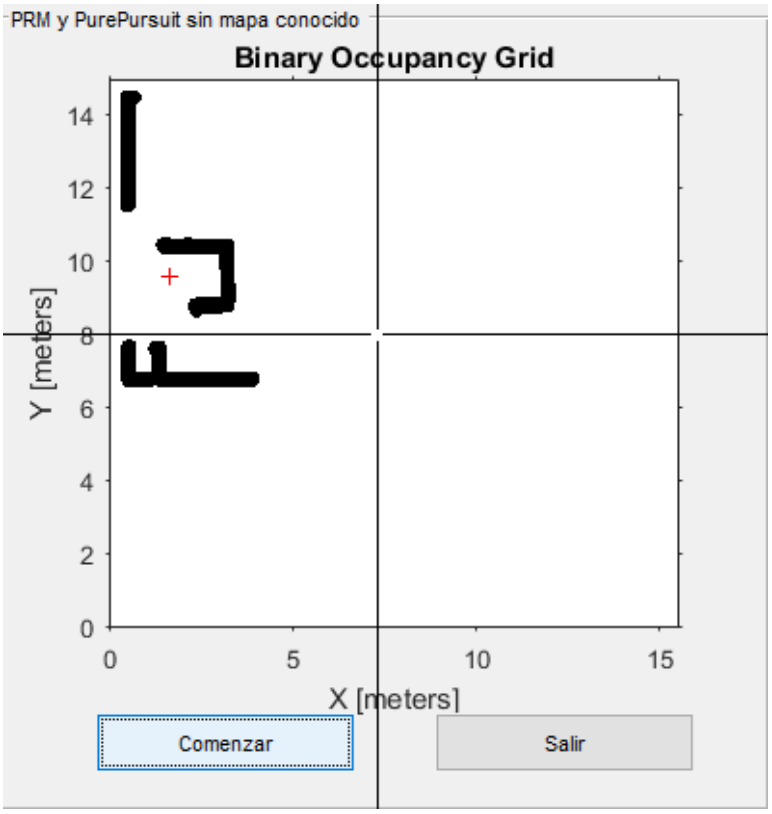

Ilustración 41 Mapa parcial y selección de punto de destino

Para esta aplicación es necesario hacerse *rossubscriber* de la odometría, del láser y de los sónares, y hacerse *rospublisher* del topic *robot0/cmd_vel* que permite dar velocidad al robot. Esto se hace en la función *OpeningFcn* de *Guide*, por lo que se ejecuta cuando se abre la aplicación. En esta función además se define la posición del sónar y se dibuja un mapa en blanco.

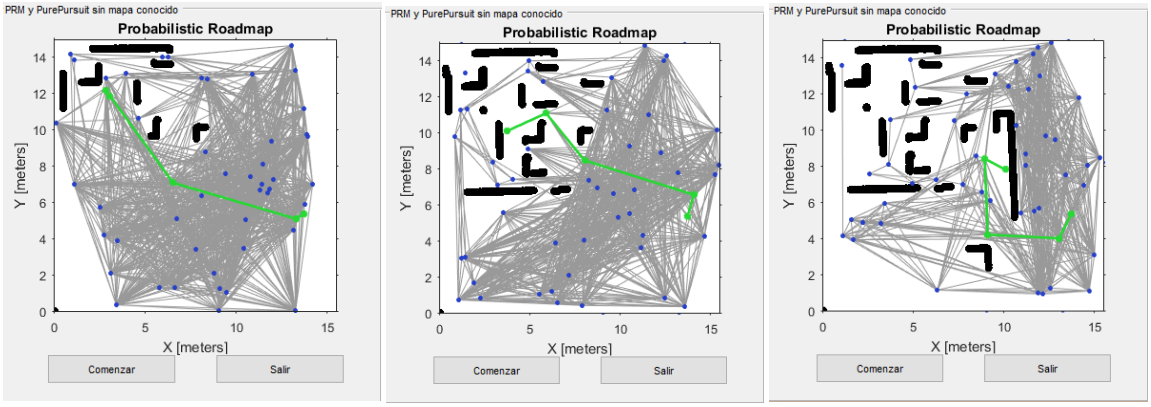

Ilustración 42 Diferentes PRMs durante el avance hacia el destino

Una vez que se pulsa *Comenzar* se dibuja el mapa con un bucle que recorre cada rayo del láser dibujando el punto en el mapa en el que choca, y se pide por pantalla que se pinche el punto de destino haciendo uso de la función *ginput* de Matlab. Después de esto se entra en el bucle *while* que contiene el resto del código donde se configura el PRM y se traza y sigue la trayectoria.

Para avanzar se utiliza el algoritmo *PurePursuit* por lo que se configura antes de que el robot comience a andar, de igual manera que se hizo en la aplicación anterior. Durante el avance se analizan los dos sónares delanteros para evitar que el robot choque. Cuando el robot para debido a la presencia delante suya de un obstáculo, el robot gira 180º para no chocar con él, pues el algoritmo *PurePursuit* funciona de tal manera que mantiene una velocidad lineal y va variando la velocidad angular conforme a un punto cercano de la trayectoria. Si se tiene el obstáculo enfrente, al girar podría chocar con él y este giro lo evita.

```
%%%%%%%%%% Hacemos que de media vuelta para no chocar %%%%%

                PosX = posdata.Pose.Pose.Position.X;
                PosY = posdata.Pose.Pose.Position.Y;
                PosicionRobot = [PosX PosY];

                AngFinal=angdiff(0,yaw-(pi));
                PosX1= PosX + 1*cos(AngFinal);
                PosY1= PosY + 1*sin(AngFinal);

                DestinoGiro = [PosX1 PosY1];

                velmsg.Angular.Z= 0.3;
                send(robot,velmsg);

%%%%%%% bucle para que gire hasta la orientacion correcta %%%%%%%
                 while (yaw <= (atan2(DestinoGiro(2)-
PosicionRobot(2),DestinoGiro(1)-PosicionRobot(1))-0.15) || (yaw >=
(atan2(DestinoGiro(2)-PosicionRobot(2),DestinoGiro(1)-
PosicionRobot(1))+0.15)))

                 posdata = receive(posicion,4);
                 q =[posdata.Pose.Pose.Orientation.W
posdata.Pose.Pose.Orientation.X posdata.Pose.Pose.Orientation.Y
posdata.Pose.Pose.Orientation.Z];
                 [yaw, pitch, roll] = quat2angle(q);
                 end

                 velmsg.Angular.Z= 0;
                 send(robot,velmsg);
```

Una vez realizado el giro el bucle *while* vuelve a empezar, hallando una nueva trayectoria hacia el punto de destino y siguiéndola de nuevo.

La interfaz contiene el habitual botón *Salir* para cerrar la aplicación y parar el robot.

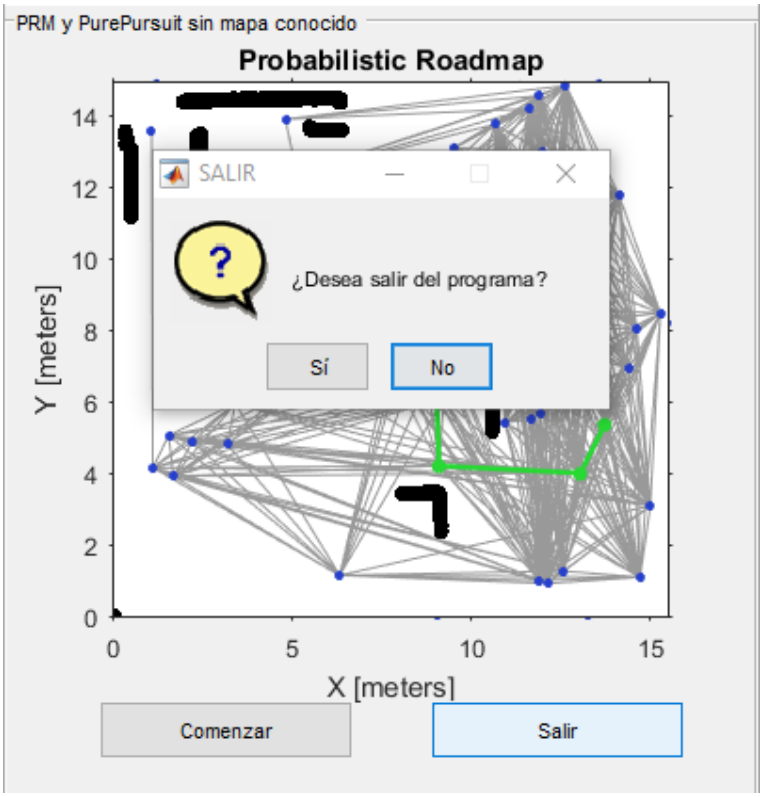

Ilustración 43 Botón y cuadro para salir de la aplicación

3.1.3 Control desde modelos de Simulink

Para realizar el control desde Simulink se deben hacer una serie de configuraciones previas. Como simulador se va a seguir utilizando el STDR y el mismo robot.

La primera configuración será para conectar Simulink a ROS, para ello se accede al menú *Tools* > *Robot Operating System* > *Configure Network Addresses.*

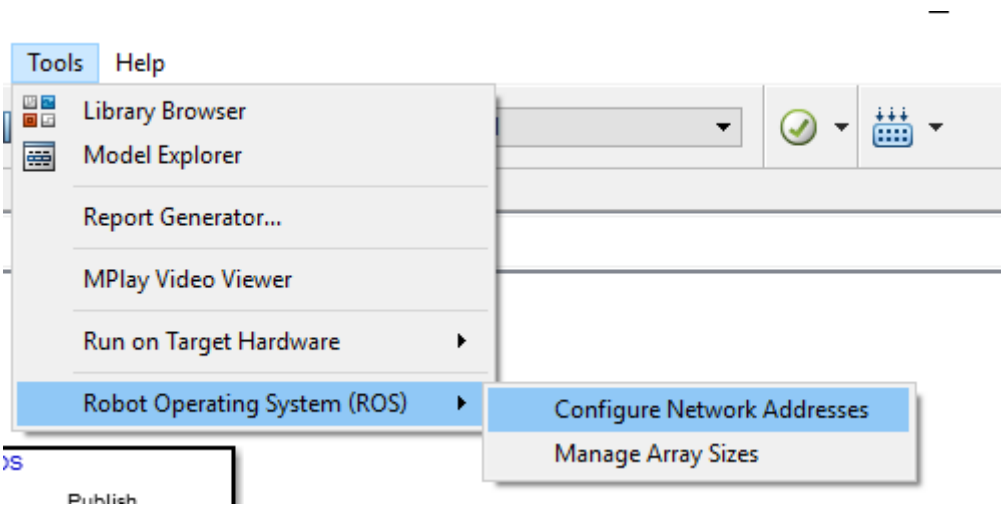

Ilustración 44 Acceso a la ventana de configuración de red

En la sección *ROS master* se selecciona *Custom* dentro de la pestaña *Network Address* y se pone la IP de nuestra red en *Hostname/IP Address* además del puerto, comúnmente 11311 dentro de la pestaña *Port.*

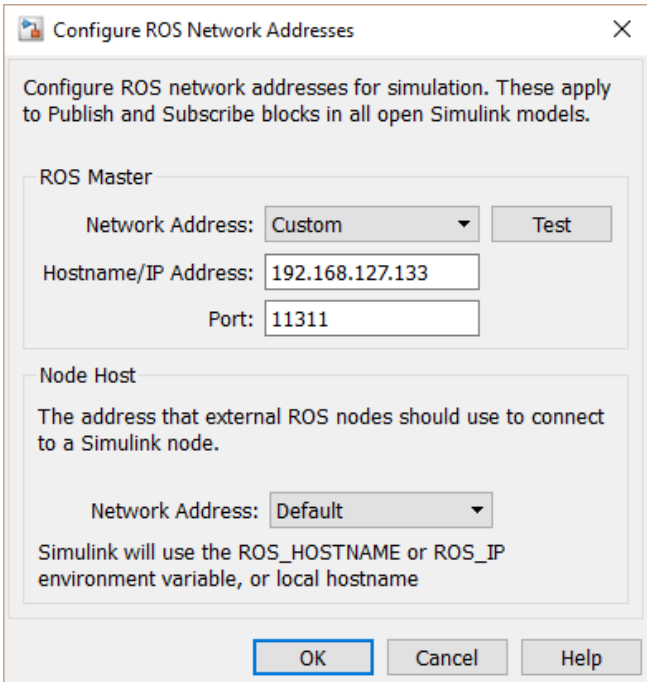

Ilustración 45 Ventana de configuración de la red

Mover el robot de forma manual

Para esta primera aplicación se debe tener abierto el simulador STDR con un robot, igual que en las aplicaciones controladas desde fichero .M. Al abrir dentro de *Library Browser* la toolbox *Robotics System* se ve que sólo aparecen tres bloques, uno para enviar mensajes a los topics, otro para publicar en ellos y otro para subscribirse. Primero se explica cómo configurar cada uno de ellos antes de entrar en materia con la aplicación.

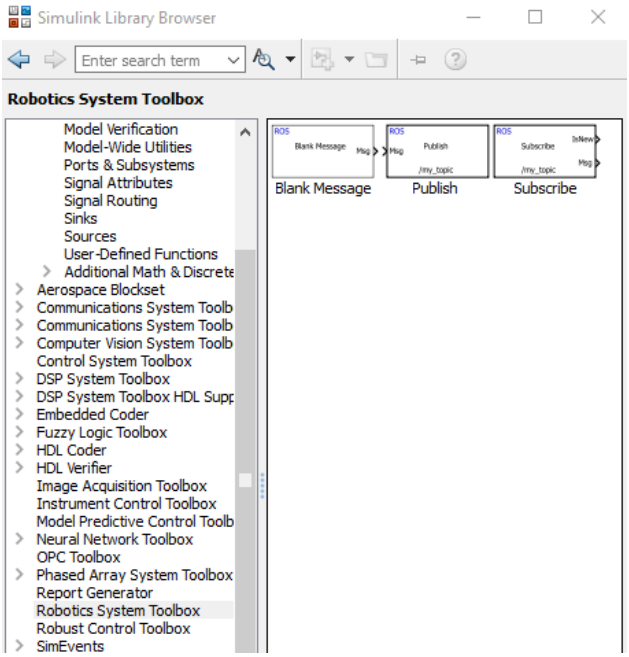

Ilustración 46 Librería Robotics System Toolbox de Simulink

El primero de ellos, *Blank Message*, crea un bus correspondiente al tipo de mensaje de ROS para el que haya sido configurado. En cada "pulso" emite una señal en blanco del tipo de mensaje asignado por lo que en cada aplicación se debe "llenar" ese pulso.

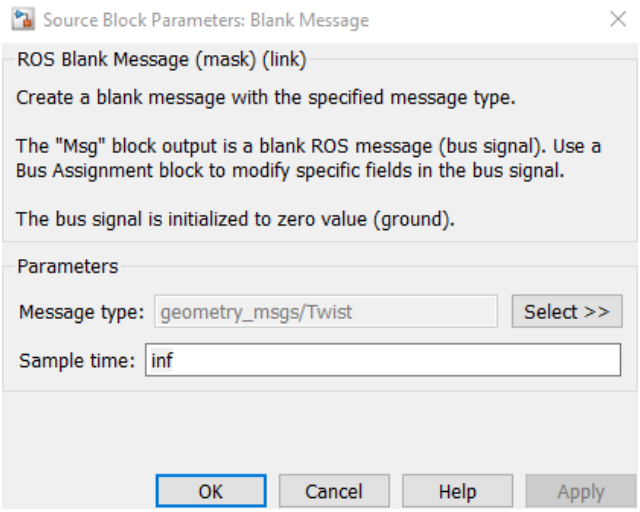

Ilustración 47 Configuración del bloque ROS Blank Message

El siguiente bloque, *Publish*, permite publicar en el topic que se elija, por lo que se debe tener ya hecha la conexión con ROS cuando se esté configurando el bloque.

Una vez seleccionado el topic, debajo de él aparece el tipo de mensaje asociado, por lo que la mejor idea es configurar el bloque *Blank Message* una vez conocido el tipo de mensaje que se necesita.

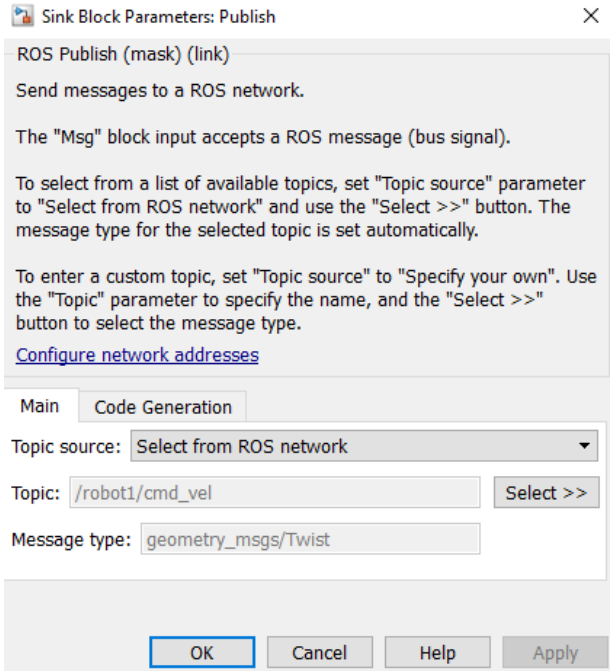

Ilustración 48 Configuración del bloque ROS Publish

Para poder modificar la velocidad del robot, se necesita acceder dentro del topic */robot0/cmd_vel* a la velocidad lineal X y a la velocidad angular Z. Con el bloque *Bus Assignment* se pueden seleccionar y darle a la señal del tipo de mensaje el valor que se necesita.

Para que aparezcan las señales del topic primero hay que unir los bloques *Bus Assignment* y *Publish* teniendo éste ya configurado. Dentro del bloque *Bus Assignment* hay que quitar la señal inicial *signal1* seleccionándola y dándole al botón *Remove* que aparece en la Ilustración 49. Ahora, para añadir las señales necesarias, se seleccionan y pulsando el botón *Select>>* se añadirán.

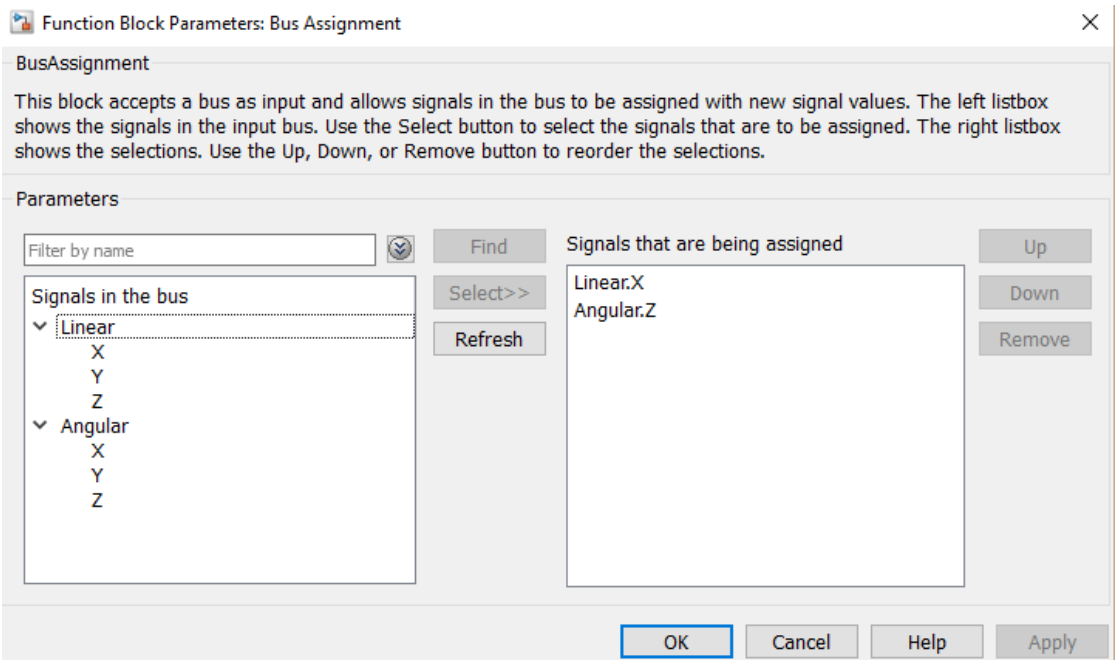

Ilustración 49 Configuración del bloque Bus Assignment

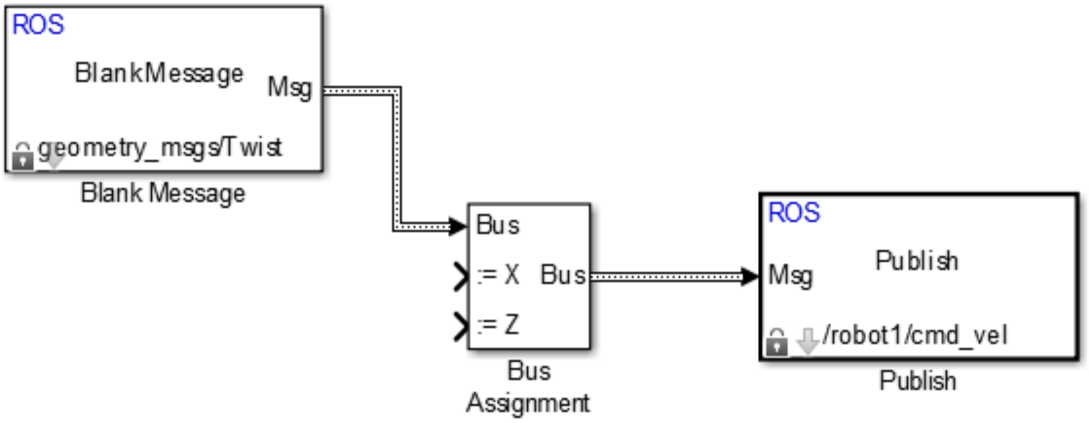

Ilustración 50 Bloques Blank Message y Publish sobre Simulink

Para leer por Matlab los datos del robot se utiliza el tercer bloque de la Toolbox *Robotics System*, el bloque *Subscribe*. La configuración de este bloque es sencilla, únicamente se debe seleccionar el topic al que se desea subscribirse para leer sus datos.

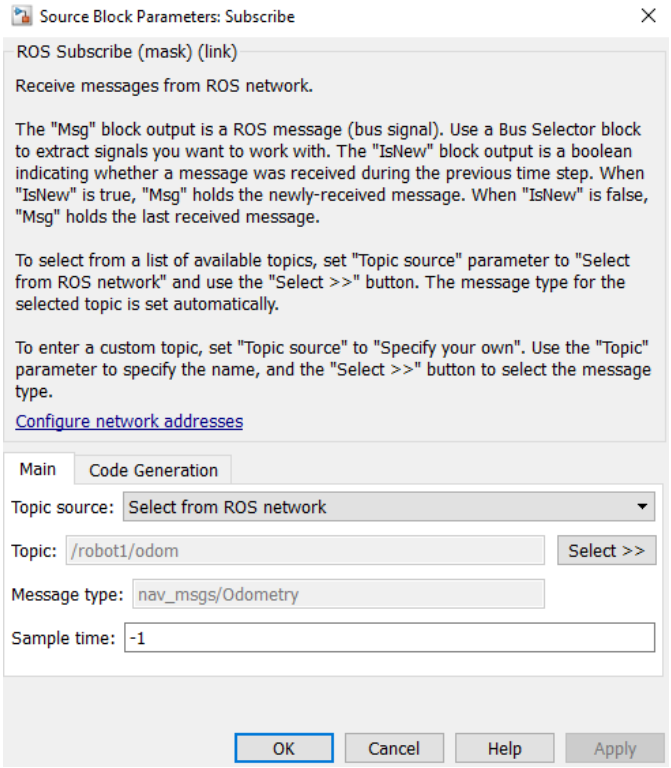

Ilustración 51 Configuración del bloque Subscribe

Como reza la explicación del bloque que se ve en la ilustración superior, para poder trabajar con las señales que se desea, se debe unir la salida *Msg* a un Bus selector, donde se eligen, en la presente aplicación, las coordenadas X e Y del robot de la misma forma que se hizo con el bloque *Bus Assignment*.

La salida *IsNew* indica si se ha recibido el mensaje durante el paso de tiempo anterior. En este caso no será necesaria por lo que se conecta esta salida a un bloque *Terminator* para que no quede al aire.

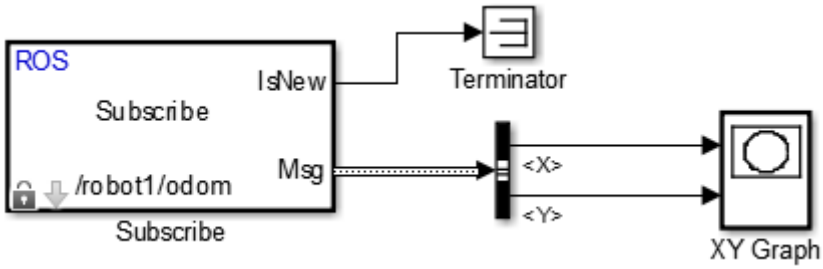

Ilustración 52 Bloque Subscribe conectado a XY Graph

Además, se ha añadido un *XY Graph* para poder visualizar la posición del robot mientras se mueve. Se ha configurado con las medidas del mapa del simulador STDR de ROS.

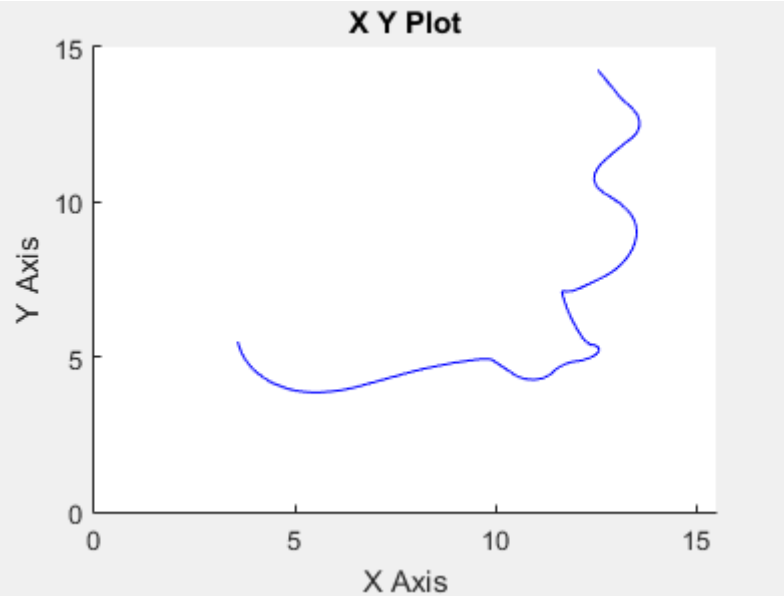

Ilustración 53 Ejemplo de recorrido del robot

Con esto se tiene la aplicación casi acabada. Solo falta darle valor al bus que permite que se publique en el topic de la velocidad del robot.

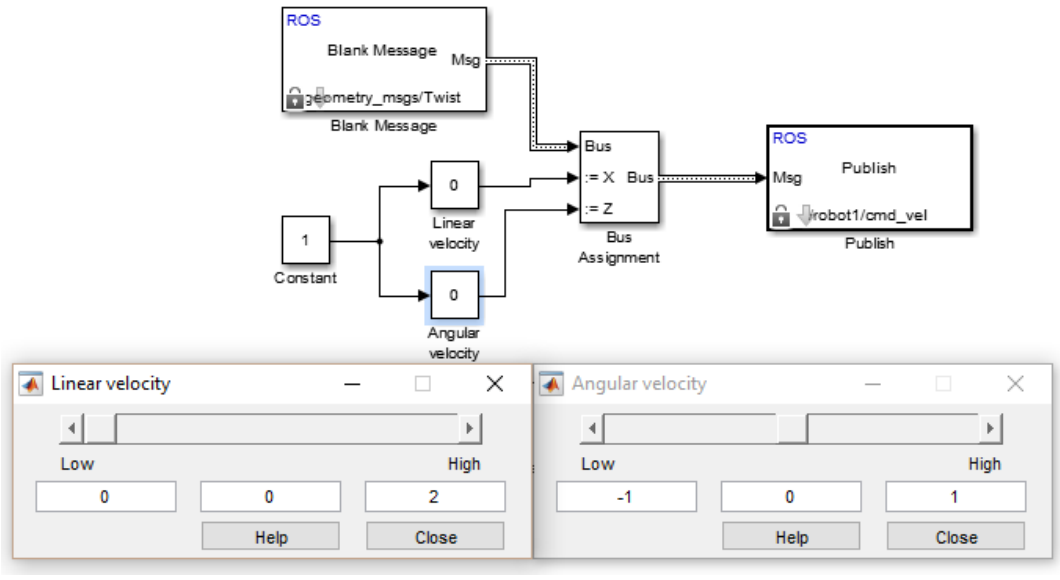

Ilustración 54 bloques con los sliders finales

Se ha incluido una constante 1 unida a dos bloques *Slider Gain* para poder modificar las velocidades lineales y angulares, como se puede ver en la Ilustración 54. La velocidad lineal puede variar entre 0 m/s y 2 m/s y la velocidad angular puede girar hasta a 1 rad/seg, en sentido horario si es negativo o en sentido anti horario si es positivo.

Con esta aplicación se puede mover el robot de forma manual por el mapa cargado en el simulador STDR de ROS.

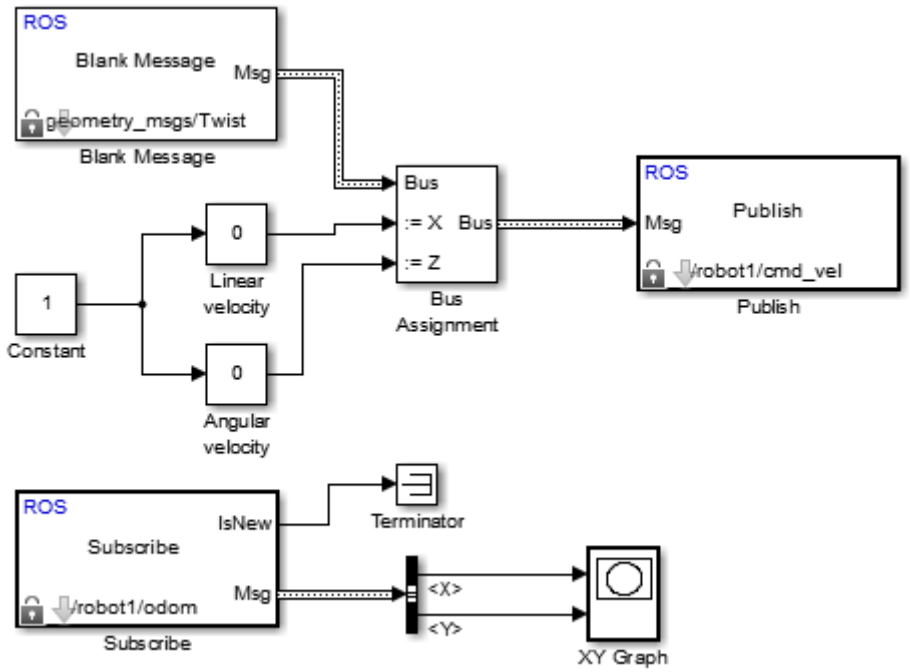

Ilustración 55 Conjunto final de la aplicación Mover Robot

Controlador de movimiento con realimentación

Esta aplicación mueve al robot hasta un punto objetivo, realimentando con la odometría del robot para saber en qué dirección avanzar y cuándo parar. Hace uso de los bloques propios de la toolbox *Robotics System* para subscribirse y publicar en los topics */robot0/odom* y */robot0/cdm_vel* respectivamente.

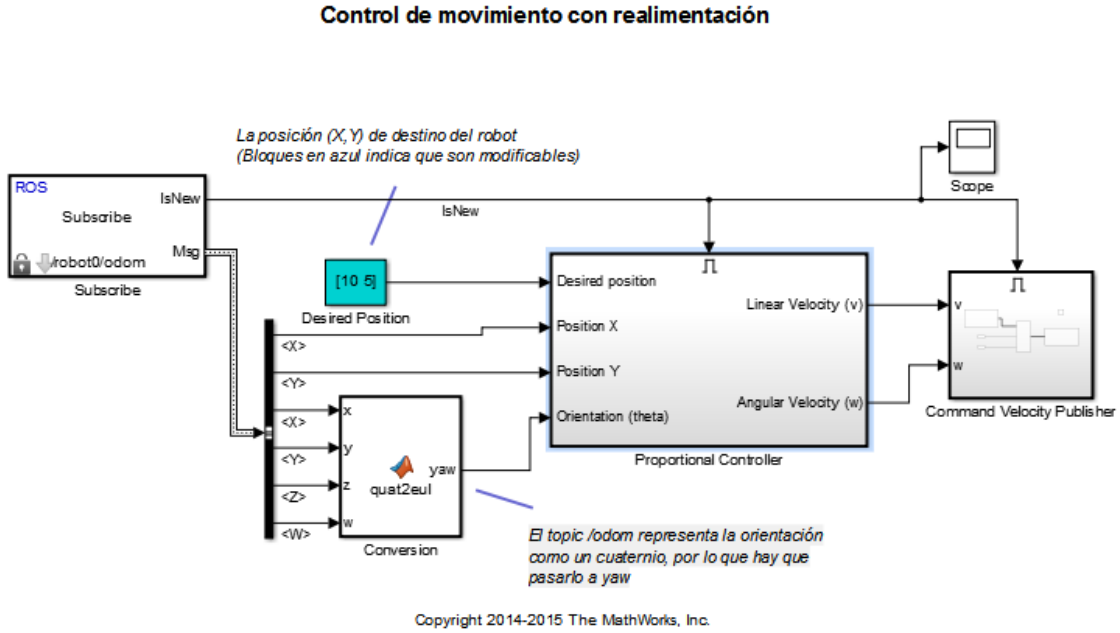

Ilustración 56 Esquema de bloques de la aplicación

Como la odometría es representada por el topic */robot0/odom* como un cuaternio, la única forma de poder pasarlo a ángulos de Euler es a través de código. En la ilustración 56 se ve el bloque *Conversion* que realiza esta función, al que le llegan las variables *X*, *Y*, *Z* y *W* como entrada, y saca la variable que va a ser utilizada de los ángulos de Euler, la variable *Yaw*. El código que se esconde tras éste bloque es el siguiente:

```
function yaw  = quat2eul(x,y,z,w)
%#codegen

% Conversión del cuaternio a angulo yaw (angulo zyx euler)
eul = quat2eul([w,x,y,z]);
yaw = eul(1);

end
```

Con el ángulo yaw, el punto de destino y las posiciones X e Y se entra en el bloque *Proportional Controller*, que analiza la distancia al objetivo y la compara con la distancia límite. En caso de ser la distancia al objetivo inferior a la distancia límite, el robot se para. Esto se consigue con el interruptor *Switch* utilizado como comparador, si la distancia no alcanza el límite, se mantiene en la posición F (false), unida a la velocidad lineal, pero si se supera el límite, el *Switch* pasa a la posición T (true), donde una constant en 0 hace que el robot pare.

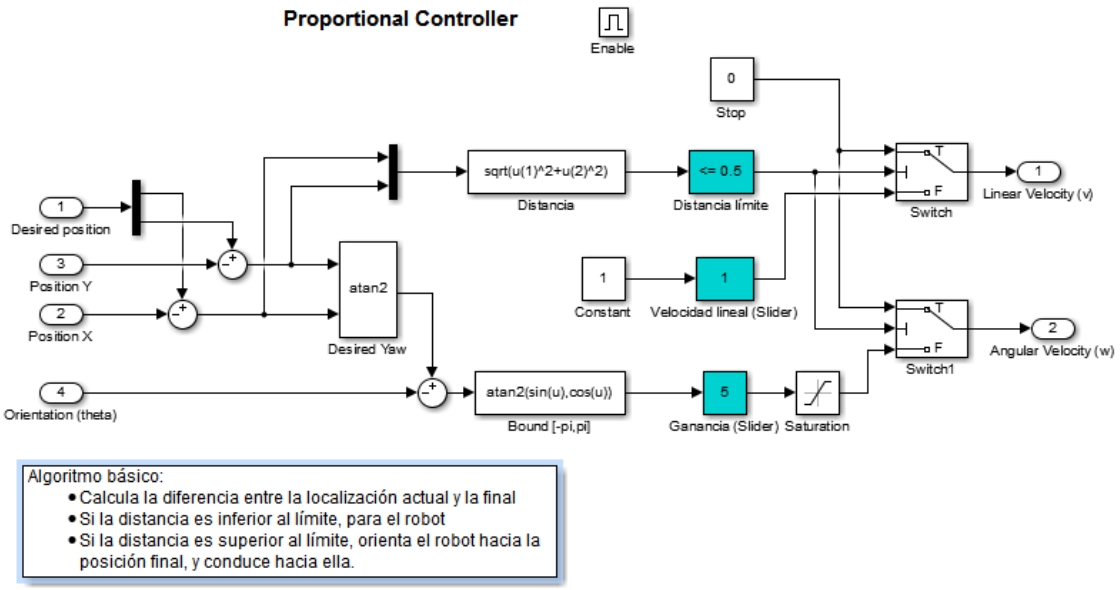

Ilustración 57 Bloque Proportional Controller

Por otro lado, compara la orientación del robot y la posición final para colocar el robot en la dirección adecuada. El bloque se actualiza en cada pulso del puerto *IsNew*, explicado en la aplicación anterior, que está unido al *Enable* del bloque, como se ve en la Ilustración 58.

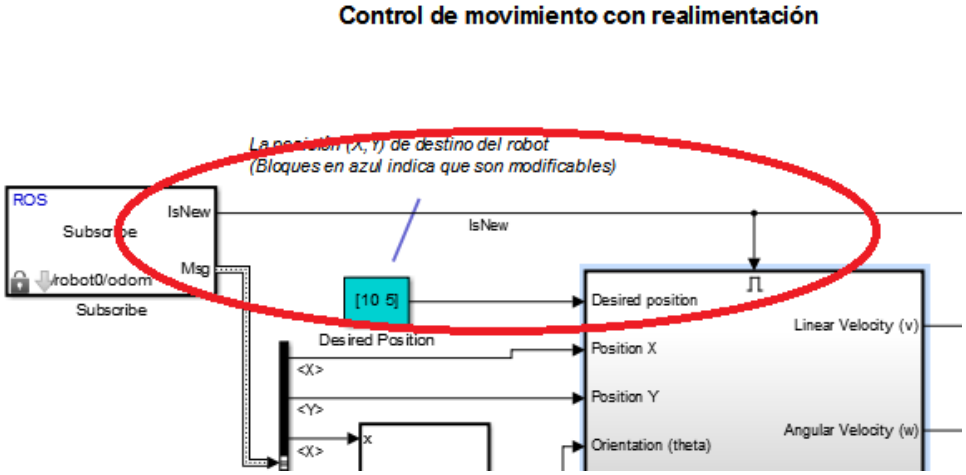

Ilustración 58 Conexión del puerto IsNew

Cuando las posiciones y orientaciones estás comparadas, ya se puede mover el robot. En el último bloque, *Command Velocity Publisher*, se hace lo mismo que en la aplicación anterior para enviar la velocidad al robot, se utilizan los bloques *BlankMessage* y *Publish* para poder mandar los mensajes al topic */robot0/cmd_vel* y se utiliza el bloque *Bus Assignment* para asignar las señales correspondientes.

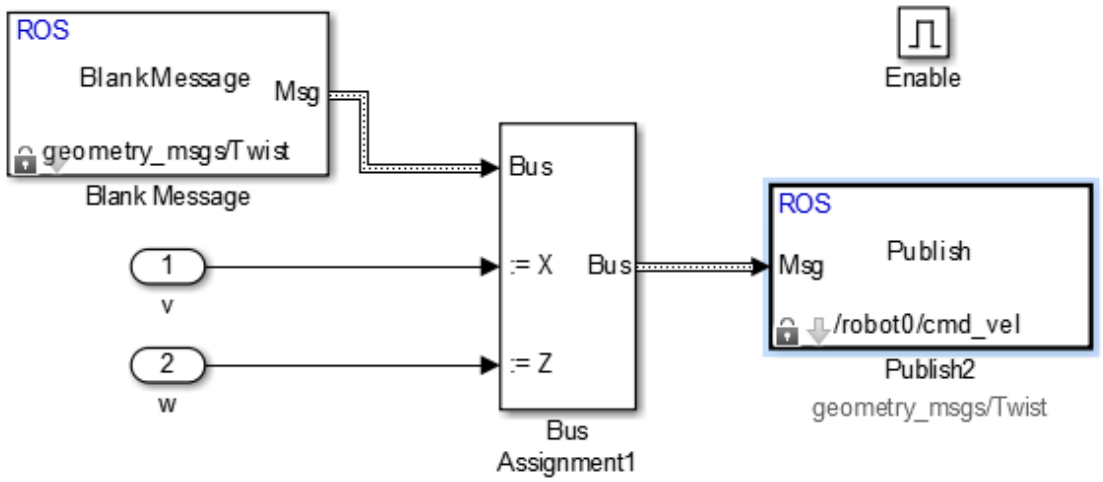

Ilustración 59 Bloque Command Velocity Publisher

Esta aplicación muestra una forma de controlar las velocidades de un robot haciendo uso de su odometría, permitiéndole desplazarse de un punto a otro de forma autónoma. Puede implementarse en aplicaciones más complejas, que permitan conocer, además de la odometría, la posición de los obstáculos gracias a los sensores, permitiendo esquivarlos para alcanzar el objetivo.

3.2 Control de cuadricópteros

En este apartado se simula un dron cuadricóptero en un entorno de simulación más complejo y en 3D, como es Gazebo. Después de configurarlo se hace un ejemplo sencillo de cómo controlar este simulador desde Matlab.

3.2.1 Configuración del simulador Gazebo

Se ha creado un paquete de ROS que permite simular en Gazebo un mundo (entorno de movimiento) y los diferentes sensores embarcados en un robot aéreo. Para arrancar dicho simulador, debe escribirse en un terminal el siguiente comando:

```
roslaunch hector_quadrotor_demo eq_simulation.launch
```

La ventana del simulador presentará el siguiente aspecto:

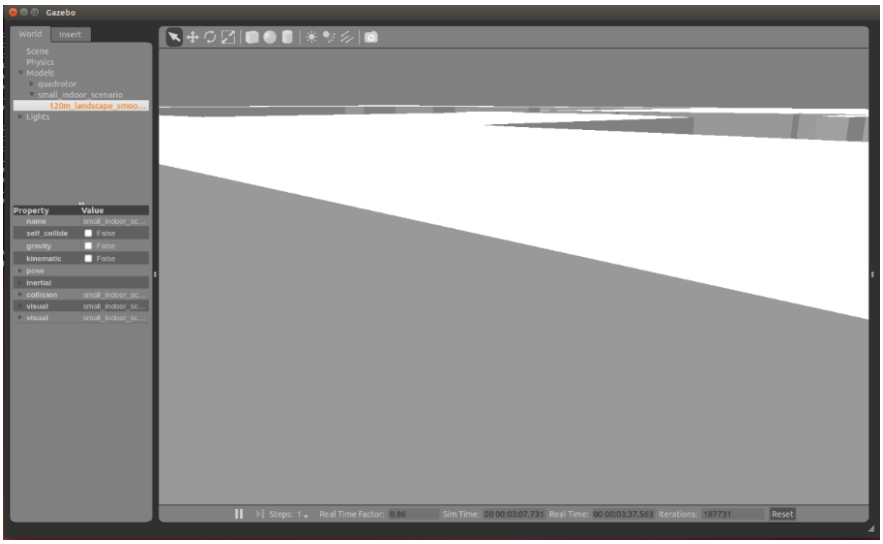

Ilustración 60 Interfaz de Gazebo

Mediante el teclado y el ratón se puede modificar la vista del entorno:

Roll del ratón -> zoom

Shift + ratón -> punto de vista

Ctrl + ratón -> desplazamiento

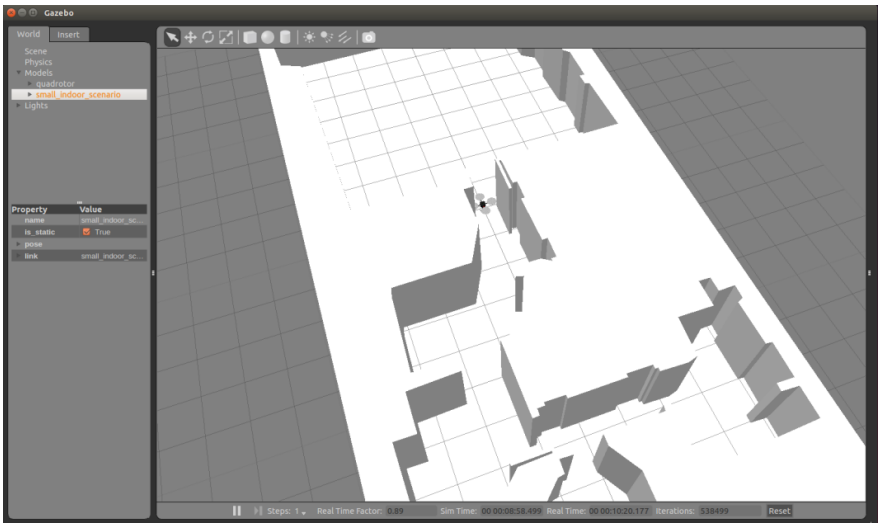

Ilustración 61 Entorno simulado en Gazebo

Para cerrar correctamente el simulador, además de cerrar su ventana con el icono **X** de la misma, debe activarse el terminal desde el cual se lanzó y escribir en él **Ctrl+C**.

A continuación, se explican algunos cambios básicos que pueden realizarse sobre la configuración del simulador.

a) Elección del mundo (world)

Es posible seleccionar entre varios entornos predefinidos en Gazebo, editando para ello el fichero lanzador (launch) de la simulación, del siguiente modo:

```
cd

gedit catkin_ws/src/hector_quadrotor-indigo-
evel/hector_quadrotor_demo /launch/ eq_simulation.launch
```

En la parte inicial del fichero hay un mundo descomentado, y el resto comentados. Para comentar el mundo actual, añadir **<!--** al principio de la línea, y --> al final. Del mismo modo, para descomentar el mundo seleccionado, eliminar **<!--** al principio de la línea, y --> al final.

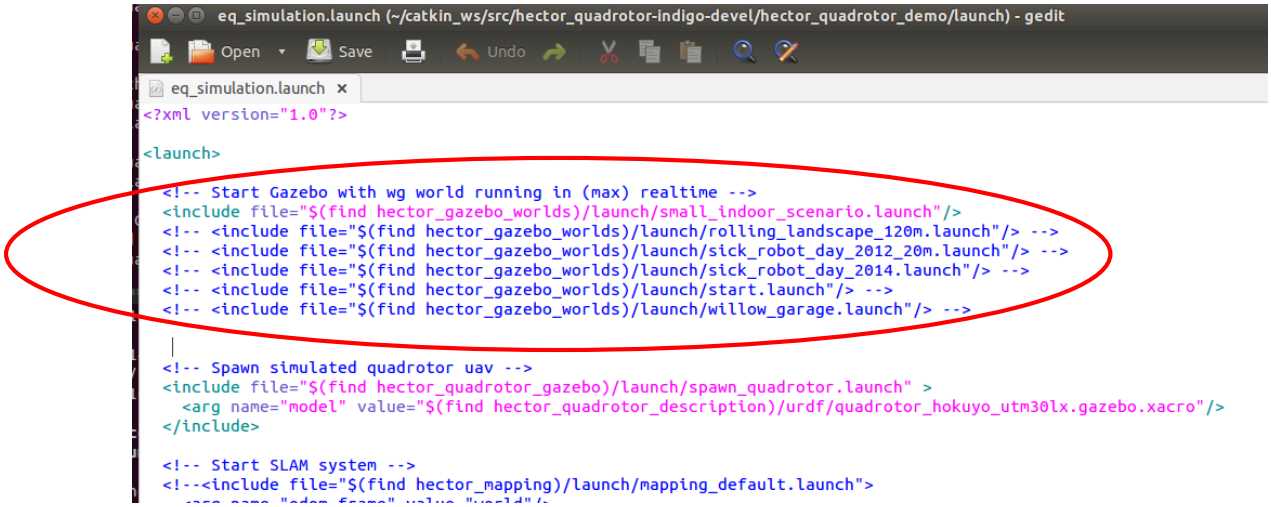

Ilustración 62 Código comentado del fichero lanzador

NOTA: Editando el fichero *launch* escogido (todos ellos se encuentran en el directorio */home/alumno/catkin_ws/src/hector_gazebo indigodevel/ hector_ gazebo _worlds/launch*), se puede ver con qué fichero *world* enlazan (suele coincidir con el nombre del fichero *launch*). Los ficheros *world* se encuentran en el directorio */home/alumno/catkin_ws/src/hector_gazebo-indigo_devel/hector_ gazebo_worlds/ worlds* y contienen, en modo texto, las configuraciones básicas de ese mundo.

Por otro lado, existen en internet repositorios con más modelos de entorno, o bien pueden crearse manualmente desde cero (consultar los manuales online de Gazebo).

b) Posición inicial del robot

Para modificar la posición inicial del robot dentro del entorno, debe editarse el fichero correspondiente:

```
cd

gedit catkin_ws/src/hector_quadrotor-indigo-devel/hector_quadrotor_
gazebo/launch/ spawn_quadrotor.launch
```

Y configurar correctamente los parámetros correspondientes a la posición inicial:

```
<?xml version="1.0"?>

<launch>
  <arg name="name" default="quadrotor"/>
  <arg name="model" default="$(find hector_quadrotor_description)/urdf/quadrotor.gazebo.xacro"/>
  <arg name="tf_prefix" default="$(optenv ROS_NAMESPACE)"/>
  <arg name="controller_definition" default="$(find hector_quadrotor_controller)/launch/controller.launch"/>

  <arg name="x" default="0.0"/>
  <arg name="y" default="0.0"/>
  <arg name="z" default="0.3"/>
  <arg name="roll" default="0.0"/>
  <arg name="pitch" default="0.0"/>
  <arg name="yaw" default="0.0"/>

  <arg name="use_ground_truth_for_tf" default="true" />
  <arg name="use_ground_truth_for_control" default="true" />
  <arg name="use_pose_estimation" if="$(arg use_ground_truth_for_control)" default="false"/>
```

Ilustración 63 Parámetros de la posición incial

Puede observarse que la coordenada *Z* del dron debe ponerse a 0.3 si se desea que el robot esté apoyado en el suelo, ya que esta es la altura añadida por las patas (necesarias para añadir el sensor láser en la parte inferior del robot).

c) Configuración de los sensores

El robot simulado es un cuadricóptero equipado con los siguientes sensores: IMU, láser, dos sónares enfocados al suelo y el techo, y cámara. Las configuraciones de estos sensores pueden modificarse editando los ficheros *.xacro* en los que están definidos, que son los siguientes:

- *Configuración de los sonares superior e inferior*

  Editar el siguiente fichero:

```
cd
gedit catkin_ws/src/hector_quadrotor-indigo-devel/hector_quadrotor_
description/urdf/quadrotor_base.urdf.xacro
```

```
    <!-- Sonar height sensor1 (roof)-->
    <xacro:sonar_sensor name="sonar1" parent="base_link" ros_topic="sonar_height1" update_rate="10"
min_range="0.03" max_range="3.0" field_of_view="${40*pi/180}" ray_count="3">
      <origin xyz="0 0 0.04" rpy="0 -1.57 0"/>
    </xacro:sonar_sensor>

    <!-- Sonar height sensor2 (floor) -->
    <xacro:sonar_sensor name="sonar2" parent="base_link" ros_topic="sonar_height2" update_rate="10"
min_range="0.03" max_range="3.0" field_of_view="${40*pi/180}" ray_count="3">
      <origin xyz="-0.16 0.0 -0.012" rpy="0 ${90*pi/180} 0"/>
    </xacro:sonar_sensor>
```

Ilustración 64 Configuración de los sensores

- Configuración de la IMU. Editar el siguiente fichero:

```
cd

gedit catkin_ws/src/hector_quadrotor-indigo-devel/hector_quadrotor_
gazebo/urdf/quadrotor_sensors.gazebo.xacro
```

- Configuración del láser y de la cámara. Editar el siguiente fichero:

```
Cd

gedit catkin_ws/src/hector_quadrotor-indigo-devel/hector_quadrotor_
description/urdf/quadrotor_hokuyo_utm30lx.urdf.xacro
```

NOTA IMPORTANTE: el láser está colocado boca abajo en la parte inferior del dron. Por eso se le han colocado unas patas cuya altura es de 0.3 metros (esta será la coordenada Z mínima que se podrá enviar al dron). Por otro lado, las lecturas del láser deberán ser invertidas en orden en Matlab para "darles la vuelta" y pasarlas al sistema de referencia del dron (esto ya se ha realizado en los ficheros de ejemplo).

Uso del visualizador RViz

El visualizador RViz permite visualizar los diferentes datos presentes en los topics de ROS. En la máquina virtual se ha realizado la configuración necesaria para que, al arrancar RViz, se visualicen directamente todos los datos de los sensores de interés.

Para ejecutar RViz, escribir en un nuevo terminal:

```
rosrun rviz rviz
```

El aspecto del visualizador al arrancar será el siguiente:

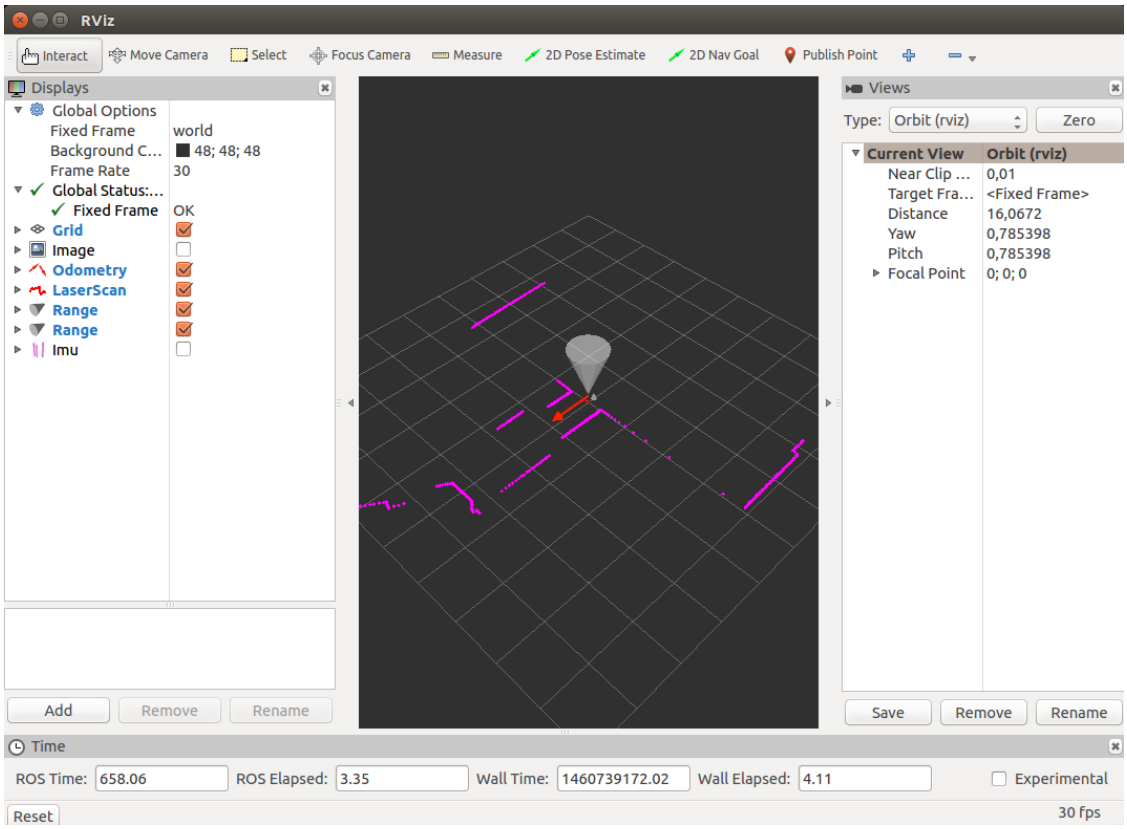

Ilustración 65 Visualizador RViz

Las variables visualizadas pueden activarse o desactivarse en el panel de *Displays* de la parte izquierda de la ventana. Por defecto se encuentran activadas la rejilla (Grid), la odometría del simulador (Odometry – mostrada mediante una flecha roja), las lecturas del láser (LaserScan – mostradas con puntos morados) y los sonares del techo y suelo (Range – conos grises). También pueden activarse, con el tick correspondiente, las imágenes de la cámara y la medida de los acelerómetros de la IMU.

Para salir de RViz, pulsar en el icono **X** de la ventana. Al cerrar, preguntará si se desea guardar la última configuración. Si no se quiere modificar la configuración inicial realizada, seleccionar la opción *Quit without saving*.

3.2.2 Control desde fichero .m de Matlab

Se proporciona el fichero *Sim_Drone.m* como ejemplo de envío de posiciones al dron simulado en la máquina virtual, y de lectura de los datos del láser, sónares e IMU. El programa ejecuta varios bucles que son similares entre sí, únicamente cambiando el movimiento realizado (avance, retroceso, ascenso, descenso y giro).

En primer lugar, se debe configurar Gazebo y crear la comunicación Matlab-ROS como se ha explicado anteriormente. A continuación, se analiza el código utilizado para el ejemplo.

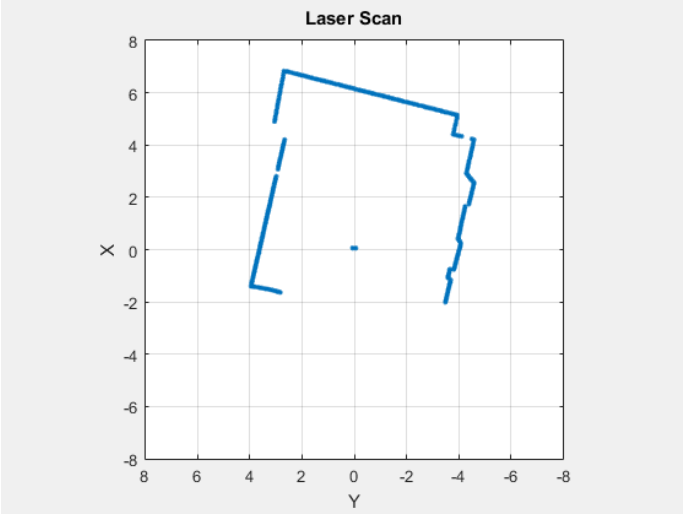

Ilustración 66 Entorno dibujado por el láser

Hay que subscribirse a los topics que se quiere leer, para cada sensor también se crea un mensaje en el que poder realizar las lecturas posteriormente.

```
%LASER
laser_sub = rossubscriber('/scan', rostype.sensor_msgs_LaserScan);
laser_msg = rosmessage(laser_sub);

%IMU
imu_sub = rossubscriber('/raw_imu', rostype.sensor_msgs_Imu);
imu_msg = rosmessage(imu_sub);

%SONAR TECHO
sonar_techo_sub = rossubscriber ('/sonar_height1',
rostype.sensor_msgs_Range);
sonar_techo_msg= rosmessage(sonar_techo_sub);

%SONAR SUELO
sonar_suelo_sub = rossubscriber ('/sonar_height2',
rostype.sensor_msgs_Range);

sonar_suelo_msg = rosmessage(sonar_suelo_sub);
```

Se crea el *Publisher* para enviar los datos de posición, el topic en concreto es el */gazebo/set_model_state* de Gazebo.

```
pos_pub = rospublisher ('/gazebo/set_model_state',
'gazebo_msgs/ModelState');
pos_msg = rosmessage(pos_pub);
```

Después se rellenan los datos del mensaje de estado del robot, se fija el robot sobre el cual se quiere modificar su estado (posición), el sistema de referencia sobre el que se aplica la traslación y rotación, y la posición inicial por defecto, su translación y orientación en cuaternio. Estos campos del mensaje serán los que se tendrán que modificar para enviar nuevas posiciones al quadrotor.

```
pos_msg.ModelName='quadrotor';
pos_msg.ReferenceFrame='world';
pos_msg.Pose.Position.X=0;
pos_msg.Pose.Position.Y=0;
pos_msg.Pose.Position.Z=0.3;  %0.3 es la Z minima (altura de las patas)
pos_msg.Pose.Orientation.X=0;
pos_msg.Pose.Orientation.Y=0;
pos_msg.Pose.Orientation.Z=0;
pos_msg.Pose.Orientation.W=1;
```

En esta aplicación se va a desactivar la física de la gravedad en el simulador, para poder simular la posición del robot desde Matlab sin tener en cuenta las físicas del robot que carga ROS. Para ello se crea el cliente para llamar a servicios de *Gazebo* de la siguiente forma:

```
srv_client = rossvcclient('/gazebo/set_physics_properties');
svc_req=rosmessage(srv_client);
svc_req.Gravity.X=0;
svc_req.Gravity.Y=0;
svc_req.Gravity.Z=0;
svc_req.TimeStep=0.001;
```

Primero se crea la petición (mensaje) para enviar al servicio y se rellena la petición de servicio. Es importante mantener los parámetros por defecto. Se fija también el *timestep* de la simulación, en este caso a 0.001 segundos.

El resto de parámetros se dejan por defecto, en el siguiente código se pueden ver sus valores. Con todos los parámetros configurados, se realiza la llamada al servicio para desactivar la gravedad y fijar el *timestep*.

```
%Resto de parametros del solver por defecto
svc_req.OdeConfig.AutoDisableBodies=0;
svc_req.OdeConfig.SorPgsPreconIters=0;
svc_req.OdeConfig.SorPgsIters=10;
svc_req.OdeConfig.SorPgsW=1.3;
svc_req.OdeConfig.SorPgsRmsErrorTol=0;
svc_req.OdeConfig.ContactSurfaceLayer=0.001;
svc_req.OdeConfig.ContactMaxCorrectingVel=10.0;
svc_req.OdeConfig.Cfm=0.0;
svc_req.OdeConfig.Erp=0.2;
svc_req.OdeConfig.MaxContacts=20;
%Llamada al servicio. Se deshabilita la gravedad en la simulacion y
se fija el timestep
call(srv_client,svc_req,'Timeout',3);
```

Una vez la gravedad está desactivada, se puede realizar el envío de movimientos al dron. Lo primero que hará el dron será avanzar a lo largo del eje X durante 20 iteraciones, a una altura de 0.5 metros. Dentro del bucle *for* lo primero es realizar la lectura de los sensores (imu, sonar techo, sonar suelo y láser).

```
for i=0:20
    %Lectura de sensores
    imu_msg = receive(imu_sub);
    sonar_techo_msg = receive (sonar_techo_sub);
    sonar_suelo_msg = receive (sonar_suelo_sub);
    laser_msg = receive(laser_sub);  %Lectura directa del láser, que
está hacia abajo
```

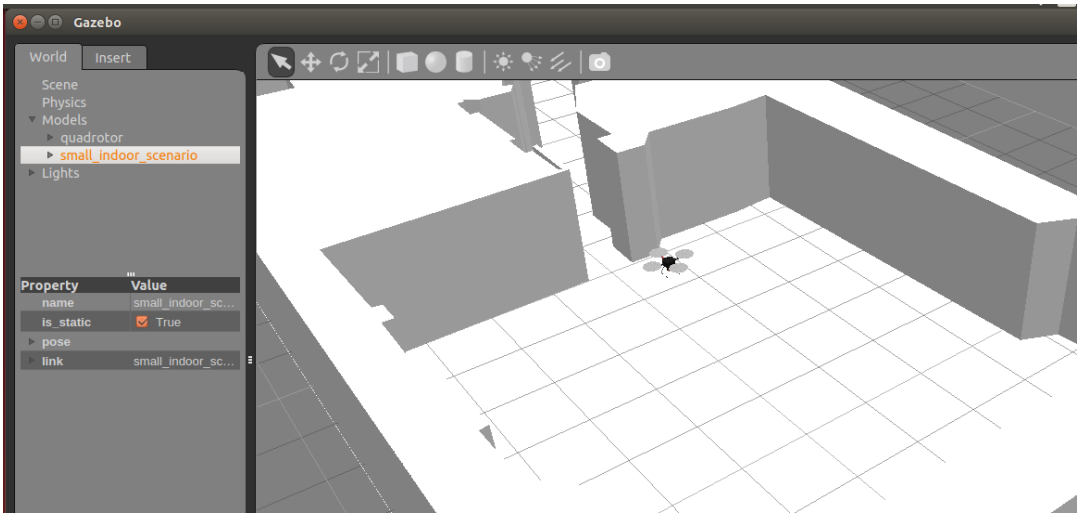

Ilustración 67 Drone en el entorno

El siguiente paso es incrementar la coordenada X de posición y enviar los datos al dron.

```
pos_msg.Pose.Position.X=pos_msg.Pose.Position.X+0.1;
pos_msg.Pose.Position.Z=0.5;
send(pos_pub,pos_msg);
```

Para representar gráficamente los datos del láser, es importante tener cuidado pues éste está hacia abajo, por lo que hay que "darle la vuelta" a los datos recibidos, invirtiendo el orden de las distancias leídas.

```
distancias(length(laser_msg.Ranges):-1:1)=laser_msg.Ranges;
laser_msg.Ranges=distancias;
plot(laser_msg,'MaximumRange',5);
```

Por último, para cerrar el bucle, se muestran por pantalla los datos de la imu y de los sónares.

```
        %Se muestran por pantalla los datos de la imu
    wx=imu_msg.AngularVelocity.X;
    wy=imu_msg.AngularVelocity.Y;
    wz=imu_msg.AngularVelocity.Z;
    ax=imu_msg.LinearAcceleration.X;
    ay=imu_msg.LinearAcceleration.Y;
    az=imu_msg.LinearAcceleration.Z;
    cadena=sprintf('IMU:: Velocidad angular=[%f %f %f]  Aceleración
lineal=[%f %f %f]',wx,wy,wz,ax,ay,az);
    disp(cadena);
    %Se muestran por pantalla los datos de los sonares
    cadena=sprintf('SONAR SUELO:: Rango=%f   SONAR TECHO::
Rango=%f',sonar_suelo_msg.Range_,sonar_techo_msg.Range_);
    disp(cadena);
end
```

El resto de movimientos del dron se realizan en bucles parecidos. El siguiente tiene como objetivo hacer retroceder el dron a lo largo del eje X durante 40 iteraciones, por lo que el código del bucle es muy similar, a excepción del punto en el que se envían los datos de posición al robot, la cual se decrementa en lugar de incrementar.

```
    %Decremento de la coordenada X y envío de los datos
    pos_msg.Pose.Position.X=pos_msg.Pose.Position.X-0.1;
    pos_msg.Pose.Position.Z=0.5;
    send(pos_pub,pos_msg);
```

Para los dos siguientes movimientos, ascender y descender a lo largo del eje Z durante 40 iteraciones, hay que modificar la posición de este eje Z.

```
%Incremento de la coordenada Z y envío de los datos
    pos_msg.Pose.Position.Z=pos_msg.Pose.Position.Z+0.1;
    send(pos_pub,pos_msg);
```

Si se quiere girar sobre alguno de los ejes, como en el siguiente y último movimiento en el que se pretende girar sobre el eje Z, se deben hacer varios cambios de unidades, de grados a radianes y de ángulos de Euler a cuaternio.

```
%Incremento del yaw.
giro=giro+deg2rad(5);   %Aumentamos 5 grados el yaw
eulerZYX=[giro 0 0];
quat=eul2quat(eulerZYX);
pos_msg.Pose.Orientation.X=quat(2);
pos_msg.Pose.Orientation.Y=quat(3);
pos_msg.Pose.Orientation.Z=quat(4);
pos_msg.Pose.Orientation.W=quat(1);


send(pos_pub,pos_msg);
```

Con estos simples movimientos se muestra cómo se maneja el simulador Gazebo de ROS desde la toolbox *Robotics System* de Matlab.

3.2.3 Control desde Simulink

Se proporciona el modelo *Sim_Drone_Simulink.slx* que contiene los bloques necesarios para enviar posiciones del dron al simulador, y para recibir las medidas de los diferentes sensores. Antes de utilizar el modelo, debe ejecutarse el script *ini_simulink.m* que realiza las siguientes acciones:

1. Definición del valor del periodo de muestreo para el envío de datos al dron
2. Establecimiento de la conexión con ROS
3. Envío a Gazebo de la solicitud de desactivación de la gravedad. Si esta parte no se ejecuta, el dron "caerá" al suelo en cuanto dejen de enviársele comandos de posición.

Por lo tanto, en primer lugar debe ejecutarse:

```
ini_simulink
```

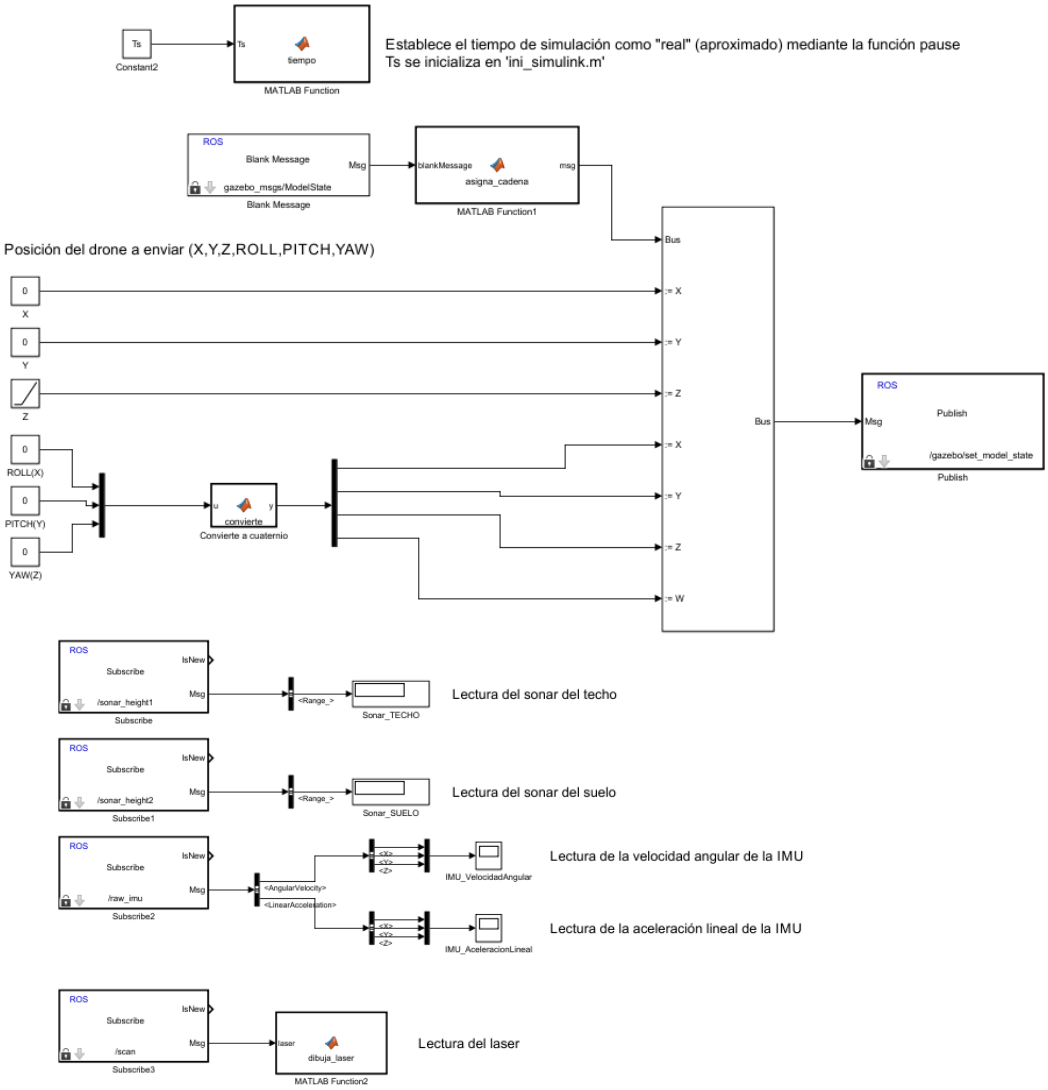

Ilustración 68 Vista de los bloques de la aplicación para Gazebo

A continuación, se muestra, por partes, el modelo de Sim_Drone_Simulink.slx:

- Temporización del envío de datos: Puesto que Simulink no trabaja en tiempo real, los datos se enviarán al simulador de la máquina virtual a la velocidad máxima que permita la simulación. Para evitar esto, y controlar la frecuencia de envío, se ha añadido una *Matlab Function* que ejecuta una función *pause*(Ts). Ts deberá ajustarse de manera que la frecuencia de envío sea la deseada.

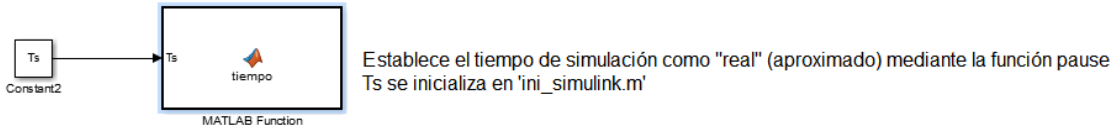

Ilustración 69 Temporización del envío de datos

- Envío de posiciones al dron: El bloque *Blank Message* crea un mensaje de tipo *gazebo_msgs/ModelState*, que a continuación hay que rellenar para enviar la posición al dron. La función *asigna_cadena* rellena los campos que son de tipo texto (nombre del modelo – 'quadrotor'- y sistema de referencia – 'world'), mientras que el bloque *Bus Assignment* rellena el resto de campos numéricos, que son los que contienen la posición del dron (traslación y rotación en cuaternio). Finalmente, el bloque *Publish* publica el mensaje en ROS.

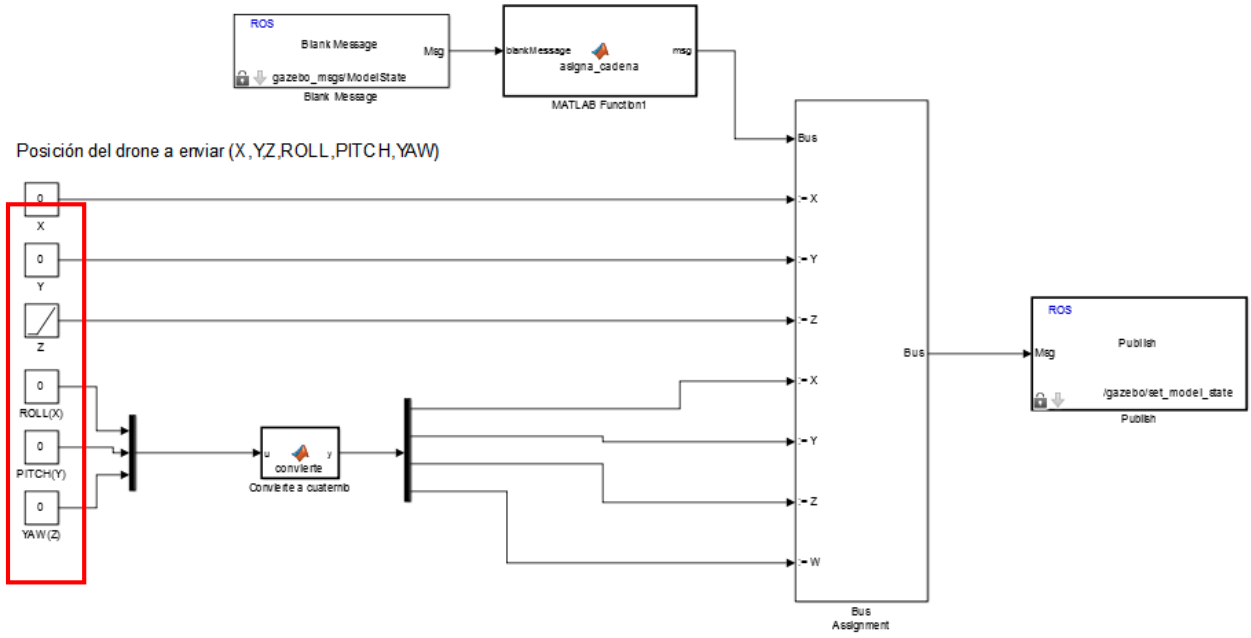

Ilustración 70 Posiciones del drone a enviar desde Simulink a Gazebo

Por lo tanto, las variables en las que hay que introducir la posición del dron son las remarcadas en el recuadro rojo, y corresponden a la traslación (X,Y,Z) y la

rotación en ángulos de Euler (ROLL, PITCH, YAW). En el ejemplo que se muestra, el dron asciende (rampa en Z). *NOTA: Es importante recordar que el valor mínimo de Z es de 0.3, por la altura de las patas).*

- Lectura de datos de los sensores: Los bloques *Subscribe* se subscriben a los topics de ROS necesarios para leer los diferentes sensores. Para el caso de los sonares y la IMU, a continuación se seleccionan los campos de interés del mensaje (los que contienen las medidas) mediante bloques *Bus Selector* y se muestran en displays. Para los datos del láser, se ha utilizado un bloque *Matlab function* que permite dibujarlos (y que previamente les da la vuelta al orden de las medidas ya que el láser está boca abajo, para pasarlas al sistema de referencia del dron).

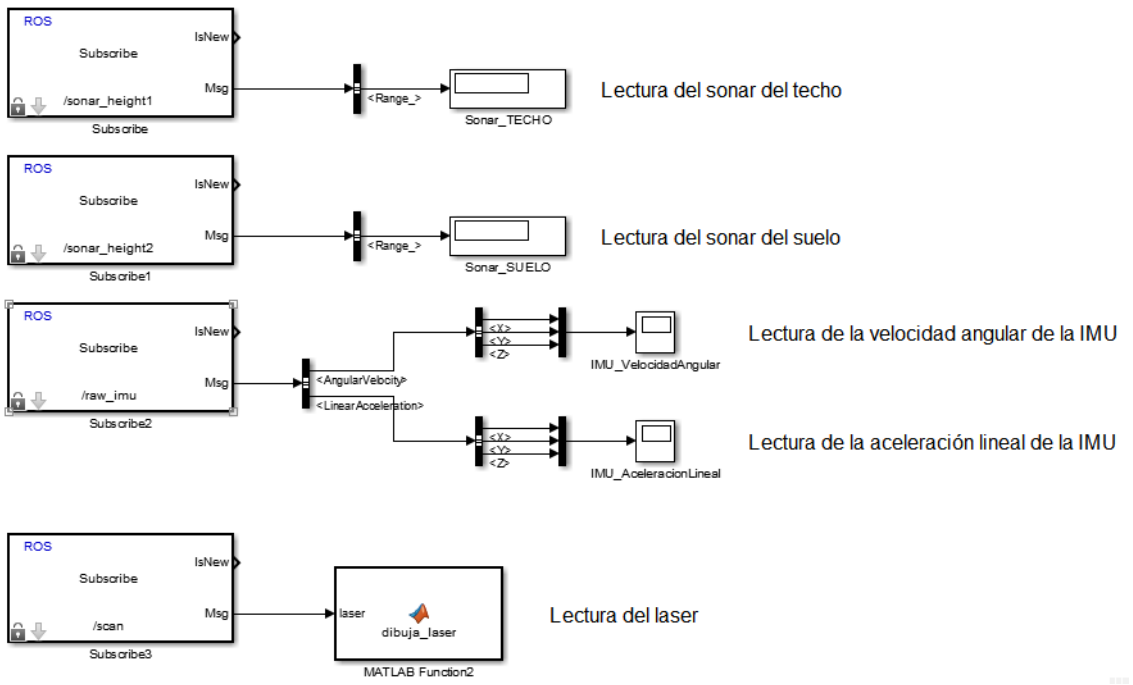

Ilustración 71 Lectura de sensores del dron

Con esta aplicación se muestra una de las posibilidades de manejo de Gazebo en ROS con Simulink de Matlab. Las posibilidades son mucho mayores, pero en este proyecto se acerca al lector a ejemplos fáciles de entender. Sin embargo, algunas funciones deben hacerse por código y es algo a mejorar en la toolbox *Robotics System* en sus funciones para Simulink.

4 Conclusiones y trabajos futuros

En el presente proyecto se ha investigado y desarrollado la toolbox *Robotics System* que permite unirse a la red de ROS y manejar un robot con ROS desde Matlab, y se han realizado una serie de aplicaciones ejemplo para mostrar su funcionamiento.

Las aplicaciones se centran más en el simulador STDR, en 2D, porque ofrece una respuesta del robot simple y por tanto una programación más sencilla para probar la toolbox. Además, también se han hecho aplicaciones que prueban el mismo simulador desde Matlab Simulink, aunque éste está más limitado en cuanto a opciones de la toolbox, pues sólo tiene dos bloques, uno para subscribirse a los topics y otros para publicar en ellos.

Las pruebas con el simulador Gazebo, bastante más complejo que el simulador STDR al ser en 3D e incluir físicas, también se han incluido en este proyecto, pero tan sólo con una aplicación de código y otra de Simulink. Estas aplicaciones son de movimientos simples de un dron, pero ofrecen un primer paso para profundizar más en el simulador Gazebo, que aporta más realismo.

Estas aplicaciones pueden ser llevadas a la práctica, sustituyendo el simulador STDR por robots pequeños y de poca potencia, en los que las aceleraciones y giros no supongan un problema de físicas. Para robots más grandes habría que realizar la simulación en Gazebo y luego llevarla a la práctica, pero en este proyecto no ha sido ese el objetivo, se ha querido demostrar la potencia de la toolbox de Matlab y su utilidad.

Durante la realización del proyecto se han logrado todos los objetivos, apoyados en una toolbox fuerte y que permite, sobre todo por código, realizar casi todas las acciones que

se pueden realizar en ROS. Desde lectura y escritura de topics y nodos hasta utilización de algoritmos de gran utilidad para la robótica como PRM o Pure Pursuit.

Con todo ello, la toolbox tiene cosas que mejorar, a pesar de ser bastante potente. Una de las cosas más importantes es el sistema de coordenadas. Si el origen fuera el propio robot, un origen local, las aplicaciones ganarían en realismo y podrían aplicarse mejor en robots físicos. Actualmente existen algoritmos de localización basados en el método de Monte Carlo que permiten establecer una probabilidad de la posición del robot.

5 Planos

En este capítulo se encuentran los códigos de las aplicaciones antes explicadas. Cada código está precedido de una breve explicación de la aplicación.

5.1 Control de robots móviles con STDR

En las siguientes secciones se verán los códigos de las aplicaciones destinadas a controlar un robot en el simulador STDR de ROS.

Control de movimiento usando la odometría

En esta primera aplicación se mueve un robot, eligiendo si se quiere girar a la izquierda, a la derecha o avanzar. La distancia de avance se puede modificar.

```
function varargout = Mover robot guide(varargin)

gui_Singleton = 1;
gui_State = struct('gui_Name',       mfilename, ...
                   'gui_Singleton',  gui_Singleton, ...
                   'gui_OpeningFcn', @Mover robot guide OpeningFcn, ...
                   'gui_OutputFcn',  @Mover robot guide OutputFcn, ...
                   'gui_LayoutFcn',  [] , ...
                   'gui_Callback',   []);
if nargin && ischar(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end

if nargout
    [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
    gui_mainfcn(gui_State, varargin{:});
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%% Esta funcion se ejecuta al abrir la Guide %%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function Mover robot guide OpeningFcn(hObject, eventdata, handles, varargin)

%%%%%%%% Se hacen los publisher y subscriber necesarios %%%%%%%%%%

global robot velmsg posicion
```

```
robot = rospublisher('/robot0/cmd_vel');  %Revisar por si cambia el nombre.
velmsg = rosmessage(robot);

posicion = rossubscriber('/robot0/odom'); %Leemos lo que pone en la odometría

%%%%%%%%%%%%%%%%%%%% Se definen las variables %%%%%%%%%%
global x x1 y y1 goalRadius
x=0; x1=0;
y=0; y1=0;
goalRadius=0.1;

handles.LongitudAvance=double(1);           %Almacenar el indentificador

handles.output = hObject;
guidata(hObject,handles);                   %Salvar datos de la aplicacion

function varargout = Mover_robot_guide_OutputFcn(hObject, eventdata, handles)

varargout{1} = handles.output;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%Función que permite al robot avanzar una casilla%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function avanzar Callback(hObject, eventdata, handles)

%%%%%%%%%% Se definen las variables globales %%%%%%%%%%%%%%%%%%%%%%
global robot velmsg posicion
global x x1 y y1 goalRadius

%%%%%%%%%% Se halla la posicion del robot y su orientacion %%%%%%%%%%
posdata = receive(posicion,4);
q =[posdata.Pose.Pose.Orientation.W posdata.Pose.Pose.Orientation.X
posdata.Pose.Pose.Orientation.Y posdata.Pose.Pose.Orientation.Z];
[yaw, pitch, roll] = quat2angle(q);

x = posdata.Pose.Pose.Position.X;
y = posdata.Pose.Pose.Position.Y;

PosicionRobot = [x y];

%%%%%%%%%% Localizamos en el mapa el punto de destino %%%%%%%%%%%%%
x1=x + handles.LongitudAvance*cos(yaw);
y1=y + handles.LongitudAvance*sin(yaw);

robotGoal = [x1 y1];

%distanceToGoal debe ser igual a la distancia de avance
distanceToGoal = norm(robotGoal - PosicionRobot);

 %%%%%%%%%% Se envía la velocidad al robot para que se mueva%%%%%%%%%%%%
velmsg.Linear.X = 0.2;
send(robot,velmsg);

 %%%%% Bucle que realimenta con la odomentria hasta llegar al objetivo
    while( distanceToGoal > goalRadius )

        posdata = receive(posicion,4);

        PosicionRobot = [posdata.Pose.Pose.Position.X posdata.Pose.Pose.Position.Y];

        %%% Actualizamos la distancia al objetivo %%%
        distanceToGoal = norm(robotGoal - PosicionRobot);

    end

%%%%%% Paramos el robot %%%%%
velmsg.Linear.X= 0;
send(robot,velmsg);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%%%%%%%%%%%%%%%%%%%% Funcion para girar a la izquierda %%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function GiroIzquierda Callback(hObject, eventdata, handles)

global robot velmsg posicion
global x x1 y y1

posdata = receive(posicion,4);

q =[posdata.Pose.Pose.Orientation.W posdata.Pose.Pose.Orientation.X
posdata.Pose.Pose.Orientation.Y posdata.Pose.Pose.Orientation.Z];

[yaw, pitch, roll] = quat2angle(q);

x = posdata.Pose.Pose.Position.X;
y = posdata.Pose.Pose.Position.Y;
PosicionRobot = [x y];

%%Sacamos la posicion  del objetivo

AngFinal=angdiff(0,yaw+(pi/2));
x1=x + 1*cos(AngFinal);
y1=y + 1*sin(AngFinal);

robotGoal = [x1 y1];

velmsg.Angular.Z= 0.1;
send(robot,velmsg);

 %%%%%%% bucle para que gire hasta la orientacion correcta %%%%%%%
    while (yaw <= (atan2(robotGoal(2)-PosicionRobot(2),robotGoal(1)-PosicionRobot(1))-
0.015) || (yaw >= (atan2(robotGoal(2)-PosicionRobot(2),robotGoal(1)-
PosicionRobot(1))+0.015)))

        posdata = receive(posicion,4);
        q =[posdata.Pose.Pose.Orientation.W posdata.Pose.Pose.Orientation.X
posdata.Pose.Pose.Orientation.Y posdata.Pose.Pose.Orientation.Z];
        [yaw, pitch, roll] = quat2angle(q);
    end
velmsg.Angular.Z= 0;
send(robot,velmsg);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%% Funcion para girar a la derecha %%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function GiroDerecha_Callback(hObject, eventdata, handles)

global robot velmsg posicion

global x x1 y y1

posdata = receive(posicion,4);

q =[posdata.Pose.Pose.Orientation.W posdata.Pose.Pose.Orientation.X
posdata.Pose.Pose.Orientation.Y posdata.Pose.Pose.Orientation.Z];

[yaw, pitch, roll] = quat2angle(q);

x = posdata.Pose.Pose.Position.X;
y = posdata.Pose.Pose.Position.Y;

PosicionRobot = [x y];

AngFinal=angdiff(0,yaw-(pi/2));

x1=x + 1*cos(AngFinal);
y1=y + 1*sin(AngFinal);

robotGoal = [x1 y1];
```

```
velmsg = rosmessage(robot);
velmsg.Angular.Z= -0.1;
     send(robot,velmsg);


 %%%%%%% bucle para que gire hasta la orientacion correcta %%%%%%%
    while (yaw <= (atan2(robotGoal(2)-PosicionRobot(2),robotGoal(1)-PosicionRobot(1))-
0.015) || (yaw >= (atan2(robotGoal(2)-PosicionRobot(2),robotGoal(1)-
PosicionRobot(1))+0.015)))

        posdata = receive(posicion,4);
        q =[posdata.Pose.Pose.Orientation.W posdata.Pose.Pose.Orientation.X
posdata.Pose.Pose.Orientation.Y posdata.Pose.Pose.Orientation.Z];
        [yaw, pitch, roll] = quat2angle(q);
    end
velmsg.Angular.Z= 0;
send(robot,velmsg);

function LongitudAvance Callback(hObject, eventdata, handles)

LdeCasilla=get(hObject,'String');   %Almacenar valor ingresado
LCasilla=str2double(LdeCasilla);    %Transformar de cadena a numero
handles.LongitudAvance=LCasilla;    %Almacenar el indentificador
guidata(hObject,handles);          %Salvar datos de la aplicacion

function LongitudAvance_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end

function Salir_Callback(hObject, eventdata, handles)

opc=questdlg('¿Desea salir del programa?','SALIR','Sí','No','No');
if strcmp(opc,'No')
    return;
end
clear,clc,close all
```

Lectura del láser y el sonar

En esta aplicación se muestra cómo hacer la lectura del láser y el sonar, y sus datos se representan en un mapa.

```
function varargout = Lectura De Sensores Guide(varargin)

% Begin initialization code - DO NOT EDIT
gui_Singleton = 1;
gui_State = struct('gui_Name',       mfilename, ...
                   'gui_Singleton',  gui_Singleton, ...
                   'gui_OpeningFcn', @Lectura De Sensores Guide OpeningFcn, ...
                   'gui_OutputFcn',  @Lectura De Sensores Guide OutputFcn, ...
                   'gui_LayoutFcn',  [] , ...
                   'gui_Callback',   []);
if nargin && ischar(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end

if nargout
    [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
    gui mainfcn(gui State, varargin{:});
end
% End initialization code - DO NOT EDIT
```

```
% --- Executes just before Lectura_De_Sensores_Guide is made visible.
function Lectura De Sensores Guide OpeningFcn(hObject, eventdata, handles, varargin)

global a;
a=0;

handles.output = hObject;

guidata(hObject, handles);

function varargout = Lectura_De_Sensores_Guide_OutputFcn(hObject, eventdata, handles)

varargout{1} = handles.output;

% --- Executes on button press in Salir.
function Salir_Callback(hObject, eventdata, handles)

opc=questdlg('¿Desea salir del programa?','SALIR','Sí','No','No');
if strcmp(opc,'No')
    return;
end

global a;
a=1;

vel_pub = rospublisher('/robot0/cmd_vel');  %Revisar por si cambia el nombre.
vel_msg = rosmessage(vel_pub);

vel msg.Linear.X = 0;
send(vel pub,vel msg);

clear,clc,close all

% --- Executes on button press in Lectura.
function Lectura_Callback(hObject, eventdata, handles)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Posiciones y orientaciones de los sonares respecto al centro del robot
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sonares x=[0.076 0.125 0.15 0.15 0.125 0.076 -0.14 -0.14];
sonares_y=[0.1 0.075 0.03 -0.03 -0.075 -0.1 -0.058 -0.058];
sonares_th=[1.5708 0.715585 0.261799 -0.261799 -0.715585 -1.5708 -2.53073 2.53073];

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%INICIALIZACION COMUNICACION ROS-MATLAB
%Es preferible ejecutar estas funciones en linea de comandos
%cuando se desee establecer la comunicación con ROS o cerrarla
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%rosshutdown;                %%Cierra conexiones previas con ROS
%rosinit('192.168.58.135');  %%Inicializacion comunicacion con ROS
                             %%Indicar la dirección IP de la máquina
                             %%virtual

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%SUSCRIPCIONES A LOS TOPICS QUE QUEREMOS LEER
%Para cada sensor a leer, nos suscribimos al topic correspondiente de ROS
%y creamos un mensaje en el que poder realizar las lecturas posteriormente
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%LASER
laser_sub = rossubscriber('/robot0/laser_1', rostype.sensor_msgs_LaserScan);
laser_msg = rosmessage(laser_sub);

%SONARES
sonar 0 sub = rossubscriber('/robot0/sonar 0', rostype.sensor msgs Range);
sonar_0_msg= rosmessage(sonar_0_sub);

sonar 1 sub = rossubscriber('/robot0/sonar 1', rostype.sensor msgs Range);
sonar 1 msg= rosmessage(sonar 1 sub);

sonar_2_sub = rossubscriber('/robot0/sonar_2', rostype.sensor_msgs_Range);
```

```
sonar_2_msg= rosmessage(sonar_2_sub);

sonar_3_sub = rossubscriber('/robot0/sonar_3', rostype.sensor_msgs_Range);
sonar_3_msg= rosmessage(sonar_3_sub);

sonar_4_sub = rossubscriber('/robot0/sonar_4', rostype.sensor_msgs_Range);
sonar_4_msg= rosmessage(sonar_4_sub);

sonar_5_sub = rossubscriber('/robot0/sonar_5', rostype.sensor_msgs_Range);
sonar_5_msg= rosmessage(sonar_5_sub);

sonar_6_sub = rossubscriber('/robot0/sonar_6', rostype.sensor_msgs_Range);
sonar_6_msg= rosmessage(sonar_6_sub);

sonar_7_sub = rossubscriber('/robot0/sonar_7', rostype.sensor_msgs_Range);
sonar_7_msg= rosmessage(sonar_7_sub);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%CREAMOS EL PUBLISHER PARA ENVIAR DATOS DE VELOCIDAD
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
vel_pub = rospublisher('/robot0/cmd_vel');  %Revisar por si cambia el nombre.
vel_msg = rosmessage(vel_pub);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%ENVÍO DE MOVIMIENTOS CON LECTURA DE SENSORES
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
vel_msg.Angular.Z=0;
vel_msg.Linear.X = 0.1;
send(vel_pub,vel_msg);
global a;

while (a==0)

    %Lectura de sensores
    sonar_0_msg = receive (sonar_0_sub);
    sonar_1_msg = receive (sonar_1_sub);
    sonar_2_msg = receive (sonar_2_sub);
    sonar_3_msg = receive (sonar_3_sub);
    sonar_4_msg = receive (sonar_4_sub);
    sonar_5_msg = receive (sonar_5_sub);
    sonar_6_msg = receive (sonar_6_sub);
    sonar_7_msg = receive (sonar_7_sub);
    laser_msg = receive(laser_sub);

    %Representacion gráfica de los datos del láser.
    hold off;
    plot(laser_msg,'MaximumRange',6)

    %Muestro por pantalla los datos de los sonares
    cadena=sprintf('SONARES:  [%f %f %f %f %f %f %f
%f]',sonar_0_msg.Range_,sonar_1_msg.Range_,sonar_2_msg.Range_,sonar_3_msg.Range_,sonar_4
_msg.Range_,sonar_5_msg.Range_,sonar_6_msg.Range_,sonar_7_msg.Range_)
    disp(cadena);

    %Los dibujo sobre la figura
    x_son(1)=sonares_x(1)+sonar_0_msg.Range_*cos(sonares_th(1));
    y_son(1)=sonares_y(1)+sonar_0_msg.Range_*sin(sonares_th(1));
    x_son(2)=sonares_x(2)+sonar_1_msg.Range_*cos(sonares_th(2));
    y_son(2)=sonares_y(2)+sonar_1_msg.Range_*sin(sonares_th(2));
    x_son(3)=sonares_x(3)+sonar_2_msg.Range_*cos(sonares_th(3));
    y_son(3)=sonares_y(3)+sonar_2_msg.Range_*sin(sonares_th(3));
    x_son(4)=sonares_x(4)+sonar_3_msg.Range_*cos(sonares_th(4));
    y_son(4)=sonares_y(4)+sonar_3_msg.Range_*sin(sonares_th(4));
    x_son(5)=sonares_x(5)+sonar_4_msg.Range_*cos(sonares_th(5));
    y_son(5)=sonares_y(5)+sonar_4_msg.Range_*sin(sonares_th(5));
    x_son(6)=sonares_x(6)+sonar_5_msg.Range_*cos(sonares_th(6));
    y_son(6)=sonares_y(6)+sonar_5_msg.Range_*sin(sonares_th(6));
    x_son(7)=sonares_x(7)+sonar_6_msg.Range_*cos(sonares_th(7));
    y_son(7)=sonares_y(7)+sonar_6_msg.Range_*sin(sonares_th(7));
    x_son(8)=sonares_x(8)+sonar_7_msg.Range_*cos(sonares_th(8));
    y_son(8)=sonares_y(8)+sonar_7_msg.Range_*sin(sonares_th(8));
```

```
    hold on;
    plot(x son, y son, 'ro')

    %%%%% Se dibuja el robot en el mapa %%%%%
    ang=0:0.01:2*pi;
    xp=0.15 * cos(ang);
    yp=0.15 * sin(ang);
    plot(xp,yp,'g');

end
```

Control de movimiento con parada de emergencia

Esta aplicación engloba las dos anteriores, permite controlar el movimiento del robot y utiliza los sensores para saber si el robot va a chocar, y en ese caso parar el robot y avisar por pantalla de ello.

```
function varargout = movimiento_Con_evitacion_Guide(varargin)

% Begin initialization code - DO NOT EDIT
gui Singleton = 1;
gui_State = struct('gui_Name',       mfilename, ...
                   'gui_Singleton',  gui_Singleton, ...
                   'gui_OpeningFcn', @movimiento Con evitacion Guide OpeningFcn, ...
                   'gui_OutputFcn',  @movimiento Con evitacion Guide OutputFcn, ...
                   'gui_LayoutFcn',  [] , ...
                   'gui_Callback',   []);
if nargin && ischar(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end

if nargout
    [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
    gui_mainfcn(gui_State, varargin{:});
end
% End initialization code - DO NOT EDIT

% --- Executes just before movimiento_Con_evitacion_Guide is made visible.
function movimiento_Con_evitacion_Guide_OpeningFcn(hObject, eventdata, handles,
varargin)

global sonares_x sonares_y sonares_th
sonares_x=[0.076 0.125 0.15 0.15 0.125 0.076 -0.14 -0.14];
sonares_y=[0.1 0.075 0.03 -0.03 -0.075 -0.1 -0.058 -0.058];
sonares th=[1.5708 0.715585 0.261799 -0.261799 -0.715585 -1.5708 -2.53073 2.53073];

global robot velmsg
robot = rospublisher('/robot0/cmd_vel');  %Revisar por si cambia el nombre.
velmsg = rosmessage(robot);

global posicion
posicion = rossubscriber('/robot0/odom'); %Leemos lo que pone en la odometría

%LASER
global laser sub laser msg
laser sub = rossubscriber('/robot0/laser 1', rostype.sensor msgs LaserScan);
laser msg = rosmessage(laser sub);

%SONARES
global sonar_0_sub sonar_0_msg
sonar 0 sub = rossubscriber('/robot0/sonar 0', rostype.sensor msgs Range);
sonar 0 msg= rosmessage(sonar 0 sub);
```

```
global sonar_1_sub sonar_1_msg
sonar_1_sub = rossubscriber('/robot0/sonar_1', rostype.sensor_msgs_Range);
sonar_1_msg= rosmessage(sonar_1_sub);

global sonar_2_sub sonar_2_msg
sonar_2_sub = rossubscriber('/robot0/sonar_2', rostype.sensor_msgs_Range);
sonar_2_msg= rosmessage(sonar_2_sub);

global sonar_3_sub sonar_3_msg
sonar_3_sub = rossubscriber('/robot0/sonar_3', rostype.sensor_msgs_Range);
sonar_3_msg= rosmessage(sonar_3_sub);

global sonar_4_sub sonar_4_msg
sonar_4_sub = rossubscriber('/robot0/sonar_4', rostype.sensor_msgs_Range);
sonar_4_msg= rosmessage(sonar_4_sub);

global sonar_5_sub sonar_5_msg
sonar_5_sub = rossubscriber('/robot0/sonar_5', rostype.sensor_msgs_Range);
sonar_5_msg= rosmessage(sonar_5_sub);

global sonar_6_sub sonar_6_msg
sonar_6_sub = rossubscriber('/robot0/sonar_6', rostype.sensor_msgs_Range);
sonar_6_msg= rosmessage(sonar_6_sub);

global sonar_7_sub sonar_7_msg
sonar_7_sub = rossubscriber('/robot0/sonar_7', rostype.sensor_msgs_Range);
sonar_7_msg= rosmessage(sonar_7_sub);

handles.DistSecurity = 0.3;

handles.output = hObject;

guidata(hObject, handles);

function varargout = movimiento_Con_evitacion_Guide_OutputFcn(hObject, eventdata,
handles)

varargout{1} = handles.output;

% --- Executes on button press in Avanzar.
function Avanzar_Callback(hObject, eventdata, handles)

x=0; x1=0;
y=0; y1=0;
goalRadius=0.1;

global posicion
posdata = receive(posicion,4);

q =[posdata.Pose.Pose.Orientation.W posdata.Pose.Pose.Orientation.X
posdata.Pose.Pose.Orientation.Y posdata.Pose.Pose.Orientation.Z];

[yaw, pitch, roll] = quat2angle(q);

x = posdata.Pose.Pose.Position.X;
y = posdata.Pose.Pose.Position.Y;

PosicionRobot = [x y];

x1=x + 1*cos(yaw);
y1=y + 1*sin(yaw);

robotGoal = [x1 y1];
distanceToGoal = norm(robotGoal - PosicionRobot); %distanceToGoal debe ser igual a L

global velmsg robot
velmsg.Linear.X = 0.2;
send(robot,velmsg);

     %%% Bucle que realimenta con la odomentria hasta llegar al objetivo
    while( distanceToGoal > goalRadius )
```

```
    %Lectura de sensores
   global sonar_0_msg sonar_1_msg sonar_2_msg sonar_3_msg sonar_4_msg
   global sonar_5_msg sonar_6_msg sonar_7_msg laser_msg

   global sonar_0_sub sonar_1_sub sonar_2_sub sonar_3_sub sonar_4_sub
   global sonar_5_sub sonar_6_sub sonar_7_sub laser_sub

    sonar_0_msg = receive (sonar_0_sub);
    sonar_1_msg = receive (sonar_1_sub);
    sonar_2_msg = receive (sonar_2_sub);
    sonar_3_msg = receive (sonar_3_sub);
    sonar_4_msg = receive (sonar_4_sub);
    sonar_5_msg = receive (sonar_5_sub);
    sonar_6_msg = receive (sonar_6_sub);
    sonar_7_msg = receive (sonar_7_sub);
    laser_msg = receive(laser_sub);

    %Representacion gráfica de los datos del láser.
    hold off;
    plot(laser_msg,'MaximumRange',6)

    %Muestro por pantalla los datos de los sonares
    cadena=sprintf('SONARES:  [%f %f %f %f %f %f %f
%f]',sonar_0_msg.Range ,sonar_1_msg.Range ,sonar_2_msg.Range ,sonar_3_msg.Range ,sonar_4
 msg.Range ,sonar_5_msg.Range ,sonar_6_msg.Range ,sonar_7_msg.Range );
    disp(cadena);

    %Los dibujo sobre la figura
    global sonares_x sonares_y sonares_th
    x_son(1)=sonares_x(1)+sonar_0_msg.Range_*cos(sonares_th(1));
    y_son(1)=sonares_y(1)+sonar_0_msg.Range_*sin(sonares_th(1));
    x_son(2)=sonares_x(2)+sonar_1_msg.Range_*cos(sonares_th(2));
    y_son(2)=sonares_y(2)+sonar_1_msg.Range_*sin(sonares_th(2));
    x_son(3)=sonares_x(3)+sonar_2_msg.Range_*cos(sonares_th(3));
    y_son(3)=sonares_y(3)+sonar_2_msg.Range_*sin(sonares_th(3));
    x_son(4)=sonares_x(4)+sonar_3_msg.Range_*cos(sonares_th(4));
    y_son(4)=sonares_y(4)+sonar_3_msg.Range_*sin(sonares_th(4));
    x_son(5)=sonares_x(5)+sonar_4_msg.Range_*cos(sonares_th(5));
    y_son(5)=sonares_y(5)+sonar_4_msg.Range_*sin(sonares_th(5));
    x_son(6)=sonares_x(6)+sonar_5_msg.Range_*cos(sonares_th(6));
    y_son(6)=sonares_y(6)+sonar_5_msg.Range_*sin(sonares_th(6));
    x_son(7)=sonares_x(7)+sonar_6_msg.Range_*cos(sonares_th(7));
    y_son(7)=sonares_y(7)+sonar_6_msg.Range_*sin(sonares_th(7));
    x_son(8)=sonares_x(8)+sonar_7_msg.Range_*cos(sonares_th(8));
    y_son(8)=sonares_y(8)+sonar_7_msg.Range_*sin(sonares_th(8));

    hold on;
    plot(x_son, y_son, 'ro')

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%% Dibujamos el circulo de seguridad %%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

ang=0:0.01:2*pi;
xp=handles.DistSecurity * cos(ang);
yp=handles.DistSecurity * sin(ang);

plot(xp,yp,'g');

        posdata = receive(posicion,4);
        q =[posdata.Pose.Pose.Orientation.W posdata.Pose.Pose.Orientation.X
posdata.Pose.Pose.Orientation.Y posdata.Pose.Pose.Orientation.Z];
        [yaw, pitch, roll] = quat2angle(q);

        PosicionRobot = [posdata.Pose.Pose.Position.X posdata.Pose.Pose.Position.Y];

        %%% Actualizamos la distancia al objetivo %%%
        distanceToGoal = norm(robotGoal - PosicionRobot);

         if ((sonar_2_msg.Range_ <= handles.DistSecurity) || (sonar_3_msg.Range_ <=
handles.DistSecurity))
                distanceToGoal = 0;
```

```
                errordlg('Cuidado, el robot se chocará','Cuidado');
        end
    end

velmsg.Linear.X= 0;
send(robot,velmsg);

% --- Executes on button press in GirarIzquierda.
function GirarIzquierda Callback(hObject, eventdata, handles)
 direccion = 0;
global robot posicion velmsg

x=0; x1=0;
y=0; y1=0;
goalRadius=0.1;
posdata = receive(posicion,4);

q =[posdata.Pose.Pose.Orientation.W posdata.Pose.Pose.Orientation.X
posdata.Pose.Pose.Orientation.Y posdata.Pose.Pose.Orientation.Z];

[yaw, pitch, roll] = quat2angle(q);

x = posdata.Pose.Pose.Position.X;
y = posdata.Pose.Pose.Position.Y;
 %%Sacamos la posicion del robot y del objetivo
    PosicionRobot = [x y];

    AngFinal=angdiff(0,yaw+(pi/2));

    x1=x + 1*cos(AngFinal);
    y1=y + 1*sin(AngFinal);

    robotGoal = [x1 y1];
    distanceToGoal = norm(robotGoal - PosicionRobot); %distanceToGoal debe ser igual a L

     velmsg.Angular.Z= 0.1;
     send(robot,velmsg);

      %%%%%%% bucle para que gire hasta la orientacion correcta %%%%%%%
    while (yaw <= (atan2(robotGoal(2)-PosicionRobot(2),robotGoal(1)-PosicionRobot(1))-
0.02) || (yaw >= (atan2(robotGoal(2)-PosicionRobot(2),robotGoal(1)-
PosicionRobot(1))+0.02)))

        %%%%%%%%%%%% Dibujo el mapa mientras gira %%%%%%%%%%%%
          %Lectura de sensores
   global sonar 0 msg sonar 1 msg sonar 2 msg sonar 3 msg sonar 4 msg
   global sonar_5_msg sonar_6_msg sonar_7_msg laser_msg

   global sonar 0 sub sonar 1 sub sonar 2 sub sonar 3 sub sonar 4 sub
   global sonar 5 sub sonar 6 sub sonar 7 sub laser sub

    sonar_0_msg = receive (sonar_0_sub);
    sonar_1_msg = receive (sonar_1_sub);
    sonar_2_msg = receive (sonar_2_sub);
    sonar 3 msg = receive (sonar 3 sub);
    sonar 4 msg = receive (sonar 4 sub);
    sonar 5 msg = receive (sonar 5 sub);
    sonar_6_msg = receive (sonar_6_sub);
    sonar_7_msg = receive (sonar_7_sub);
    laser_msg = receive(laser_sub);

    %Representacion gráfica de los datos del láser.
    hold off;
    plot(laser_msg,'MaximumRange',6)

    %Muestro por pantalla los datos de los sonares
    cadena=sprintf('SONARES:  [%f %f %f %f %f %f %f
%f]',sonar_0_msg.Range_,sonar_1_msg.Range_,sonar_2_msg.Range_,sonar_3_msg.Range_,sonar_4
_msg.Range_,sonar_5_msg.Range_,sonar_6_msg.Range_,sonar_7_msg.Range_);
    disp(cadena);

    %Los dibujo sobre la figura
    global sonares_x sonares_y sonares_th
```

```
    x_son(1)=sonares_x(1)+sonar_0_msg.Range_*cos(sonares_th(1));
    y_son(1)=sonares_y(1)+sonar_0_msg.Range_*sin(sonares_th(1));
    x_son(2)=sonares_x(2)+sonar_1_msg.Range_*cos(sonares_th(2));
    y_son(2)=sonares_y(2)+sonar_1_msg.Range_*sin(sonares_th(2));
    x_son(3)=sonares_x(3)+sonar_2_msg.Range_*cos(sonares_th(3));
    y_son(3)=sonares_y(3)+sonar_2_msg.Range_*sin(sonares_th(3));
    x_son(4)=sonares_x(4)+sonar_3_msg.Range_*cos(sonares_th(4));
    y_son(4)=sonares_y(4)+sonar_3_msg.Range_*sin(sonares_th(4));
    x_son(5)=sonares_x(5)+sonar_4_msg.Range_*cos(sonares_th(5));
    y_son(5)=sonares_y(5)+sonar_4_msg.Range_*sin(sonares_th(5));
    x_son(6)=sonares_x(6)+sonar_5_msg.Range_*cos(sonares_th(6));
    y_son(6)=sonares_y(6)+sonar_5_msg.Range_*sin(sonares_th(6));
    x_son(7)=sonares_x(7)+sonar_6_msg.Range_*cos(sonares_th(7));
    y_son(7)=sonares_y(7)+sonar_6_msg.Range_*sin(sonares_th(7));
    x_son(8)=sonares_x(8)+sonar_7_msg.Range_*cos(sonares_th(8));
    y_son(8)=sonares_y(8)+sonar_7_msg.Range_*sin(sonares_th(8));

    hold on;
    plot(x_son, y_son, 'ro')

ang=0:0.01:2*pi;
xp=handles.DistSecurity * cos(ang);
yp=handles.DistSecurity * sin(ang);

plot(xp,yp,'g');
 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

        posdata = receive(posicion,4);
        q =[posdata.Pose.Pose.Orientation.W posdata.Pose.Pose.Orientation.X
posdata.Pose.Pose.Orientation.Y posdata.Pose.Pose.Orientation.Z];
        [yaw, pitch, roll] = quat2angle(q);
    end
     velmsg.Angular.Z= 0;
     send(robot,velmsg);

% --- Executes on button press in GirarDerecha.
function GirarDerecha_Callback(hObject, eventdata, handles)
 direccion = 0;
robot = rospublisher('/robot0/cmd_vel');  %Revisar por si cambia el nombre.
posicion = rossubscriber('/robot0/odom'); %Leemos lo que pone en la odometría

x=0; x1=0;
y=0; y1=0;
goalRadius=0.1;
posdata = receive(posicion,4);

q =[posdata.Pose.Pose.Orientation.W posdata.Pose.Pose.Orientation.X
posdata.Pose.Pose.Orientation.Y posdata.Pose.Pose.Orientation.Z];

[yaw, pitch, roll] = quat2angle(q);

x = posdata.Pose.Pose.Position.X;
y = posdata.Pose.Pose.Position.Y;
% hObject    handle to GiroDerecha (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
PosicionRobot = [x y];
    %%% tenemos que pasarlo a grados para poder manejar todos los
    %%% cuadrantes y poder sacar la posicion final %%%%
  AngFinal=angdiff(0,yaw-(pi/2));

    x1=x + 1*cos(AngFinal);
    y1=y + 1*sin(AngFinal);

    robotGoal = [x1 y1];
    distanceToGoal = norm(robotGoal - PosicionRobot); %distanceToGoal debe ser igual a L

     velmsg = rosmessage(robot)
     velmsg.Angular.Z= -0.1;
     send(robot,velmsg);

      %%%%%%% bucle para que gire hasta la orientacion correcta %%%%%%%
```

```
    while (yaw <= (atan2(robotGoal(2)-PosicionRobot(2),robotGoal(1)-PosicionRobot(1))-
0.02) || (yaw >= (atan2(robotGoal(2)-PosicionRobot(2),robotGoal(1)-
PosicionRobot(1))+0.02)))

         %%%%%%%%%%%% Dibujo el mapa mientras gira %%%%%%%%%%%%
          %Lectura de sensores
   global sonar_0_msg sonar_1_msg sonar_2_msg sonar_3_msg sonar_4_msg
   global sonar 5 msg sonar 6 msg sonar 7 msg laser msg

   global sonar 0 sub sonar 1 sub sonar 2 sub sonar 3 sub sonar 4 sub
   global sonar_5_sub sonar_6_sub sonar_7_sub laser_sub

    sonar_0_msg = receive (sonar_0_sub);
    sonar 1 msg = receive (sonar 1 sub);
    sonar 2 msg = receive (sonar 2 sub);
    sonar_3_msg = receive (sonar_3_sub);
    sonar_4_msg = receive (sonar_4_sub);
    sonar_5_msg = receive (sonar_5_sub);
    sonar 6 msg = receive (sonar 6 sub);
    sonar 7 msg = receive (sonar 7 sub);
    laser_msg = receive(laser_sub);

    %Representacion gráfica de los datos del láser.
    hold off;
    plot(laser msg,'MaximumRange',6)

    %Muestro por pantalla los datos de los sonares
    cadena=sprintf('SONARES:  [%f %f %f %f %f %f %f
%f]',sonar_0_msg.Range_,sonar_1_msg.Range_,sonar_2_msg.Range_,sonar_3_msg.Range_,sonar_4
 msg.Range ,sonar 5 msg.Range ,sonar 6 msg.Range ,sonar 7 msg.Range );
    disp(cadena);

    %Los dibujo sobre la figura
    global sonares x sonares y sonares th
    x son(1)=sonares x(1)+sonar 0 msg.Range *cos(sonares th(1));
    y son(1)=sonares y(1)+sonar 0 msg.Range *sin(sonares th(1));
    x son(2)=sonares x(2)+sonar 1 msg.Range *cos(sonares th(2));
    y_son(2)=sonares_y(2)+sonar_1_msg.Range_*sin(sonares_th(2));
    x_son(3)=sonares_x(3)+sonar_2_msg.Range_*cos(sonares_th(3));
    y son(3)=sonares y(3)+sonar 2 msg.Range *sin(sonares th(3));
    x son(4)=sonares x(4)+sonar 3 msg.Range *cos(sonares th(4));
    y son(4)=sonares y(4)+sonar 3 msg.Range *sin(sonares th(4));
    x_son(5)=sonares_x(5)+sonar_4_msg.Range_*cos(sonares_th(5));
    y_son(5)=sonares_y(5)+sonar_4_msg.Range_*sin(sonares_th(5));
    x son(6)=sonares x(6)+sonar 5 msg.Range *cos(sonares th(6));
    y son(6)=sonares y(6)+sonar 5 msg.Range *sin(sonares th(6));
    x son(7)=sonares x(7)+sonar 6 msg.Range *cos(sonares th(7));
    y_son(7)=sonares_y(7)+sonar_6_msg.Range_*sin(sonares_th(7));
    x_son(8)=sonares_x(8)+sonar_7_msg.Range_*cos(sonares_th(8));
    y_son(8)=sonares_y(8)+sonar_7_msg.Range_*sin(sonares_th(8));

    hold on;
    plot(x_son, y_son, 'ro')

ang=0:0.01:2*pi;
xp=handles.DistSecurity * cos(ang);
yp=handles.DistSecurity * sin(ang);

plot(xp,yp,'g');
 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

        posdata = receive(posicion,4);
        q =[posdata.Pose.Pose.Orientation.W posdata.Pose.Pose.Orientation.X
posdata.Pose.Pose.Orientation.Y posdata.Pose.Pose.Orientation.Z];
        [yaw, pitch, roll] = quat2angle(q);
    end
     velmsg.Angular.Z= 0;
     send(robot,velmsg);

function DistSecurity_Callback(hObject, eventdata, handles)

Distancia=get(hObject,'String');     %Almacenar valor ingresado
DistanciaS=str2double(Distancia);    %Transformar de cadena a numero
handles.DistSecurity=DistanciaS;     %Almacenar el indentificador
```

```
guidata(hObject,handles);

% --- Executes during object creation, after setting all properties.
function DistSecurity CreateFcn(hObject, eventdata, handles)

%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end

% --- Executes on button press in Salir.
function Salir_Callback(hObject, eventdata, handles)

opc=questdlg('¿Desea salir del programa?','SALIR','Sí','No','No');
if strcmp(opc,'No')
    return;
end
clear,clc,close all
```

Lectura de mapa de ocupación

Esta aplicación muestra cómo dibujar el mapa de ocupación.

```
function varargout = Crear_mapa_de_ocupacion_guide(varargin)

gui_Singleton = 1;
gui State = struct('gui Name',       mfilename, ...
                   'gui Singleton',  gui Singleton, ...
                   'gui OpeningFcn', @Crear mapa de ocupacion guide OpeningFcn, ...
                   'gui_OutputFcn',  @Crear_mapa_de_ocupacion_guide_OutputFcn, ...
                   'gui_LayoutFcn',  [] , ...
                   'gui Callback',   []);
if nargin && ischar(varargin{1})
    gui State.gui Callback = str2func(varargin{1});
end

if nargout
    [varargout{1:nargout}] = gui mainfcn(gui State, varargin{:});
else
    gui mainfcn(gui State, varargin{:});
end
% End initialization code - DO NOT EDIT

% --- Executes just before Crear mapa de ocupacion guide is made visible.
function Crear mapa de ocupacion guide OpeningFcn(hObject, eventdata, handles, varargin)

% Choose default command line output for Crear_mapa_de_ocupacion_guide
handles.output = hObject;

% Update handles structure
guidata(hObject, handles);

% --- Outputs from this function are returned to the command line.
function varargout = Crear mapa de ocupacion guide OutputFcn(hObject, eventdata,
handles)

varargout{1} = handles.output;

% --- Executes on button press in DibujarMapa.
function DibujarMapa Callback(hObject, eventdata, handles)

mapa = rossubscriber('/map')
MAP = readBinaryOccupancyGrid(mapa.LatestMessage)
```

```
show(MAP)

% --- Executes on button press in Salir.
function Salir Callback(hObject, eventdata, handles)

opc=questdlg('¿Desea salir del programa?','SALIR','Sí','No','No');
if strcmp(opc,'No')
    return;
end
clear,clc,close all
```

Planificación global con PRM y planificador local básico

En esta aplicación se hace uso del algoritmo PRM para localizar la mejor ruta desde la posición inicial del robot hasta un punto de destino evitando los obstáculos, y se conduce el robot hasta este punto con movimientos básicos (recta y giro).

```
function varargout = PRM_Basico_Guide(varargin)

% Begin initialization code - DO NOT EDIT
gui Singleton = 1;
gui_State = struct('gui_Name',       mfilename, ...
                   'gui_Singleton',  gui_Singleton, ...
                   'gui_OpeningFcn', @PRM_Basico_Guide_OpeningFcn, ...
                   'gui OutputFcn',  @PRM Basico Guide OutputFcn, ...
                   'gui LayoutFcn',  [] , ...
                   'gui Callback',   []);
if nargin && ischar(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end

if nargout
    [varargout{1:nargout}] = gui mainfcn(gui State, varargin{:});
else
    gui_mainfcn(gui_State, varargin{:});
end
% End initialization code - DO NOT EDIT

% --- Executes just before PRM_Basico_Guide is made visible.
function PRM_Basico_Guide_OpeningFcn(hObject, eventdata, handles, varargin)
global mapa robot posicion
mapa = rossubscriber('/map')
robot = rospublisher('/robot0/cmd vel');
posicion = rossubscriber('/robot0/odom');

handles.output = hObject;

% Update handles structure
guidata(hObject, handles);

% --- Outputs from this function are returned to the command line.
function varargout = PRM Basico Guide OutputFcn(hObject, eventdata, handles)

varargout{1} = handles.output;

% --- Executes on button press in Comenzar.
function Comenzar_Callback(hObject, eventdata, handles)

global mapa robot posicion
posdata = receive(posicion,4);
```

```
q =[posdata.Pose.Pose.Orientation.W posdata.Pose.Pose.Orientation.X
posdata.Pose.Pose.Orientation.Y posdata.Pose.Pose.Orientation.Z];
[yaw, pitch, roll] = quat2angle(q);

MAP = readBinaryOccupancyGrid(mapa.LatestMessage)

robotRadius = 0.3;
mapInflated = copy(MAP);
inflate(mapInflated,robotRadius)

prm = robotics.PRM(mapInflated);
prm.NumNodes = 50;
prm.ConnectionDistance = 5;
hold on

show(mapInflated)
startLocation = [posdata.Pose.Pose.Position.X posdata.Pose.Pose.Position.Y]
plot(startLocation(1,1),startLocation(1,2),'+r');

opc=questdlg({'Debe pinchar el punto de destino','¿De
acuerdo?'},'Informacion','Sí','Salir', 'Salir');
if strcmp(opc,'Salir')
   clear,clc,close all;
end

robotGoal=ginput(1);
path = findpath(prm, startLocation, robotGoal)

%Por si el circuito es complicado y necesita mas nodos
while isempty(path)

    prm.NumNodes = prm.NumNodes + 10;

    update(prm);

    path = findpath(prm, startLocation, endLocation);
end
show (prm)

%%%%   Vamos a hacer que se mueva siguiendo la ruta establecida %%%%
 robotCurrentLocation = path(1,:);
 a=2;
 robotGoal = path(a,:);
 initialOrientation = yaw;
 robotCurrentPose = [robotCurrentLocation initialOrientation];

 goalRadius = 0.1;
 distanceToGoal = norm(robotCurrentLocation - robotGoal);

 plot(path(:,1), path(:,2),'k--d')
 xlim([0 16])
 ylim([0 16])
velmsg.Linear.X = 0;

  L=length(path);
 for a=2:L
     robotGoal = path(a,:);
    angFinal = atan2(robotGoal(2)-robotCurrentPose(2),robotGoal(1)-robotCurrentPose(1));

    %%%%%%%%%% decidimos en que sentido girar %%%%%%%%%%

      angulo = angdiff(angFinal, yaw);
        if angulo >= 0
         velmsg = rosmessage(robot);
         velmsg.Angular.Z= -0.3;
         send(robot,velmsg);
        end
        if angulo <= 0
         velmsg = rosmessage(robot);
         velmsg.Angular.Z= 0.3;
         send(robot,velmsg);
         end
```

```
     %%%%%%% bucle para que gire hasta la orientacion correcta %%%%%%%
    while (yaw <= (atan2(robotGoal(2)-robotCurrentPose(2),robotGoal(1)-
robotCurrentPose(1))-0.015) || (yaw >= (atan2(robotGoal(2)-
robotCurrentPose(2),robotGoal(1)-robotCurrentPose(1))+0.015)))

        posdata = receive(posicion,4);
        q =[posdata.Pose.Pose.Orientation.W posdata.Pose.Pose.Orientation.X
posdata.Pose.Pose.Orientation.Y posdata.Pose.Pose.Orientation.Z];
        [yaw, pitch, roll] = quat2angle(q);
    end
     velmsg.Angular.Z= 0;
     send(robot,velmsg);

 distanceToGoal = norm(robotGoal - robotCurrentPose(1:2));
%%%%%% Bucle para avanzar hasta la posicion de destino %%%%%%%

while( distanceToGoal > goalRadius )

     velmsg.Linear.X = 0.3;
     send(robot,velmsg);

    posdata = receive(posicion,4);
    q =[posdata.Pose.Pose.Orientation.W posdata.Pose.Pose.Orientation.X
posdata.Pose.Pose.Orientation.Y posdata.Pose.Pose.Orientation.Z];
    [yaw, pitch, roll] = quat2angle(q);

PosicionRobot = [posdata.Pose.Pose.Position.X posdata.Pose.Pose.Position.Y yaw];
    robotCurrentPose = (PosicionRobot);

    %%% Actualizamos la distancia al objetivo %%
    distanceToGoal = norm(robotGoal - robotCurrentPose(1:2));

end       %% Final del bucle While

velmsg.Linear.X = 0;
     send(robot,velmsg);

 end  %% Final del bucle for y de la funcion

% --- Executes on button press in Salir.
function Salir Callback(hObject, eventdata, handles)

opc=questdlg('¿Desea salir del programa?','SALIR','Sí','No','No');
if strcmp(opc,'No')
    return;
end
clear,clc,close all
```

Planificación global con PRM y local con Pure Pursuit

Se vuelve a hacer uso del PRM en esta aplicación igual que en la anterior, pero para conducir el robot hasta el punto de destino se utiliza otro algoritmo, el Pure Pursuit, que varía la velocidad angular del robot para dirigirlo hacia un punto cercano de la trayectoria.

```
function varargout = PRM_y_Pure_Pursuit_Guide(varargin)

% Begin initialization code - DO NOT EDIT
gui Singleton = 1;
gui_State = struct('gui_Name',       mfilename, ...
                   'gui_Singleton',  gui_Singleton, ...
                   'gui_OpeningFcn', @PRM_y_Pure_Pursuit_Guide_OpeningFcn, ...
                   'gui_OutputFcn',  @PRM_y_Pure_Pursuit_Guide_OutputFcn, ...
                   'gui_LayoutFcn',  [] , ...
                   'gui_Callback',   []);
```

```
if nargin && ischar(varargin{1})
    gui State.gui Callback = str2func(varargin{1});
end

if nargout
    [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
    gui mainfcn(gui State, varargin{:});
end
% End initialization code - DO NOT EDIT

% --- Executes just before PRM_y_Pure_Pursuit_Guide is made visible.
function PRM_y_Pure_Pursuit_Guide_OpeningFcn(hObject, eventdata, handles, varargin)

handles.output = hObject;

guidata(hObject, handles);

function varargout = PRM y Pure Pursuit Guide OutputFcn(hObject, eventdata, handles)

varargout{1} = handles.output;

% --- Executes on button press in Comenzar.
function Comenzar Callback(hObject, eventdata, handles)

mapa = rossubscriber('/map')
robot = rospublisher('/robot0/cmd_vel');
velmsg = rosmessage(robot);
posicion = rossubscriber('/robot0/odom');
posdata = receive(posicion,4);
q =[posdata.Pose.Pose.Orientation.W posdata.Pose.Pose.Orientation.X
posdata.Pose.Pose.Orientation.Y posdata.Pose.Pose.Orientation.Z];
[yaw, pitch, roll] = quat2angle(q);

MAP = readBinaryOccupancyGrid(mapa.LatestMessage)

robotRadius = 0.3;
mapInflated = copy(MAP);
inflate(mapInflated,robotRadius)

prm = robotics.PRM(mapInflated);
prm.NumNodes = 200;
prm.ConnectionDistance = 5;
hold on
show(mapInflated)
startLocation = [posdata.Pose.Pose.Position.X posdata.Pose.Pose.Position.Y]
plot(startLocation(1,1),startLocation(1,2),'+r');

opc=questdlg({'Debe pinchar el punto de destino','¿De
acuerdo?'},'Información','Sí','Salir', 'Salir');
if strcmp(opc,'Salir')
   clear,clc,close all;
end

robotGoal=ginput(1);
path = findpath(prm, startLocation, robotGoal);

%Por si el circuito es complicado y necesita mas nodos
while isempty(path)

    prm.NumNodes = prm.NumNodes + 10;

    update(prm);

    path = findpath(prm, startLocation, robotGoal);
end
show (prm)

%%%%   Vamos a intentar que se mueva siguiendo la ruta establecida
%%%%%   utilizando Pure Pursuit   %%%%

%%% COnfiguracion de parametros de Pure Pursuit %%%%
```

```
PP=robotics.PurePursuit;
PP.Waypoints=path;
PP.MaxAngularVelocity=0.5;
PP.LookaheadDistance=0.2;
PP.DesiredLinearVelocity=0.2;

posdata = receive(posicion,4);
current orient=quat2eul([posdata.Pose.Pose.Orientation.W posdata.Pose.Pose.Orientation.X
posdata.Pose.Pose.Orientation.Y posdata.Pose.Pose.Orientation.Z]);
robotCurrentLocation = [posdata.Pose.Pose.Position.X posdata.Pose.Pose.Position.Y
current_orient(1)];

distanceToGoal = norm(robotCurrentLocation(1:2) - robotGoal);

while (distanceToGoal>0.5)
    [v,w]=step(PP,robotCurrentLocation);
    velmsg.Linear.X = v;
    velmsg.Angular.Z= w;
    send(robot,velmsg);

    posdata = receive(posicion,4);
    current_orient=quat2eul([posdata.Pose.Pose.Orientation.W
posdata.Pose.Pose.Orientation.X posdata.Pose.Pose.Orientation.Y
posdata.Pose.Pose.Orientation.Z]);
    robotCurrentLocation = [posdata.Pose.Pose.Position.X posdata.Pose.Pose.Position.Y
current orient(1)];

    distanceToGoal = norm(robotCurrentLocation(1:2) - robotGoal);

end

velmsg.Linear.X = 0;
velmsg.Angular.Z= 0;
send(robot,velmsg);

% --- Executes on button press in Salir.
function Salir_Callback(hObject, eventdata, handles)

opc=questdlg('¿Desea salir del programa?','SALIR','Sí','No','No');
if strcmp(opc,'No')
    return;
end
clear,clc,close all
```

Planificación global con PRM y mapa desconocido

En esta aplicación se desconoce el mapa, por lo que se utiliza un bucle para ir actualizando la ruta trazada con PRM cada vez que el robot encuentra un obtáculo, el cual detecta haciendo uso de los sensores. La nueva ruta se traza con el obstáculo dibujado. Para el seguimiento de la trayectoria se ha utilizado el algoritmo Pure Pursuit.

```
function varargout = Aplicacion final PurePursuit Guide(varargin)

% Begin initialization code - DO NOT EDIT
gui_Singleton = 1;
gui_State = struct('gui_Name',       mfilename, ...
                   'gui_Singleton',  gui_Singleton, ...
                   'gui_OpeningFcn', @Aplicacion final PurePursuit Guide OpeningFcn, ...
                   'gui_OutputFcn',  @Aplicacion final PurePursuit Guide OutputFcn, ...
                   'gui_LayoutFcn',  [] , ...
                   'gui_Callback',   []);
if nargin && ischar(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end
```

```
if nargout
    [varargout{1:nargout}] = gui mainfcn(gui State, varargin{:});
else
    gui_mainfcn(gui_State, varargin{:});
end

function Aplicacion final PurePursuit Guide OpeningFcn(hObject, eventdata, handles,
varargin)


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Posiciones y orientaciones de los sonares respecto al centro del robot
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

global sonares_x sonares_y sonares_th
sonares_x=[0.076 0.125 0.15 0.15 0.125 0.076 -0.14 -0.14];
sonares_y=[0.1 0.075 0.03 -0.03 -0.075 -0.1 -0.058 -0.058];
sonares_th=[1.5708 0.715585 0.261799 -0.261799 -0.715585 -1.5708 -2.53073 2.53073];

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%INICIALIZACION COMUNICACION ROS-MATLAB
%Es preferible ejecutar estas funciones en linea de comandos
%cuando se desee establecer la comunicación con ROS o cerrarla
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%rosshutdown;                %%Cierra conexiones previas con ROS
%rosinit('192.168.58.135');  %%Inicializacion comunicacion con ROS
                             %%Indicar la dirección IP de la máquina
                             %%virtual

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%SUSCRIPCIONES A LOS TOPICS QUE QUEREMOS LEER
%Para cada sensor a leer, nos suscribimos al topic correspondiente de ROS
%y creamos un mensaje en el que poder realizar las lecturas posteriormente
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

global posicion
posicion = rossubscriber('/robot0/odom'); %Leemos lo que pone en la odometría

%LASER
global laser_sub laser_msg
laser_sub = rossubscriber('/robot0/laser_1', rostype.sensor_msgs_LaserScan);
laser msg = rosmessage(laser sub);

global sonar 0 sub sonar 0 msg
sonar_0_sub = rossubscriber('/robot0/sonar_0', rostype.sensor_msgs_Range);
sonar_0_msg= rosmessage(sonar_0_sub);

global sonar 1 sub sonar 1 msg
sonar 1 sub = rossubscriber('/robot0/sonar 1', rostype.sensor msgs Range);
sonar 1 msg= rosmessage(sonar 1 sub);

global sonar_2_sub sonar_2_msg
sonar 2 sub = rossubscriber('/robot0/sonar 2', rostype.sensor msgs Range);
sonar 2 msg= rosmessage(sonar 2 sub);

global sonar_3_sub sonar_3_msg
sonar_3_sub = rossubscriber('/robot0/sonar_3', rostype.sensor_msgs_Range);
sonar_3_msg= rosmessage(sonar_3_sub);

global sonar 4 sub sonar 4 msg
sonar_4_sub = rossubscriber('/robot0/sonar_4', rostype.sensor_msgs_Range);
sonar_4_msg= rosmessage(sonar_4_sub);

global sonar 5 sub sonar 5 msg
sonar 5 sub = rossubscriber('/robot0/sonar 5', rostype.sensor msgs Range);
sonar_5_msg= rosmessage(sonar_5_sub);

global sonar_6_sub sonar_6_msg
sonar 6 sub = rossubscriber('/robot0/sonar 6', rostype.sensor msgs Range);
sonar_6_msg= rosmessage(sonar_6_sub);
```

```
global sonar_7_sub sonar_7_msg
sonar_7_sub = rossubscriber('/robot0/sonar_7', rostype.sensor_msgs_Range);
sonar_7_msg= rosmessage(sonar_7_sub);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%CREAMOS EL PUBLISHER PARA ENVIAR DATOS DE VELOCIDAD
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

global robot velmsg
robot = rospublisher('/robot0/cmd_vel');  %Revisar por si cambia el nombre.
velmsg = rosmessage(robot);

global MAP
MAP = robotics.BinaryOccupancyGrid(15.5,14.92,50); %creamos el mapa
show(MAP)

global volver PrimeraVez
volver = 0; PrimeraVez = 1;

global PosX PosY PosX1 PosY1 X Y X1 Y1 XY
PosX=0; PosY=0; X=0; Y=0;
PosX1=0; PosY1=0; X1=0; Y1=0; XY=0;
handles.output = hObject;

% Update handles structure
guidata(hObject, handles);

function varargout = Aplicacion_final_PurePursuit_Guide_OutputFcn(hObject, eventdata,
handles)

varargout{1} = handles.output;

% --- Executes on button press in Comenzar.
function Comenzar_Callback(hObject, eventdata, handles)

%%%% Comezar %%%%%
 global sonar_2_msg sonar_3_msg
 global sonar_2_sub sonar_3_sub
 global laser_sub velmsg
 global volver posicion robot
 global MAP
 global PosX PosY PosX1 PosY1 X Y X1 Y1 XY

posdata = receive(posicion,4);
q =[posdata.Pose.Pose.Orientation.W posdata.Pose.Pose.Orientation.X
posdata.Pose.Pose.Orientation.Y posdata.Pose.Pose.Orientation.Z];
[yaw, pitch, roll] = quat2angle(q);

startLocation = [posdata.Pose.Pose.Position.X posdata.Pose.Pose.Position.Y];
plot(startLocation(1,1),startLocation(1,2),'+r');
laserdata = receive(laser_sub,4);

     for j=1:length(laserdata.Ranges)

     orientacionLaser = laserdata.AngleMin + (j * laserdata.AngleIncrement);

     X=cos(yaw+orientacionLaser)*laserdata.Ranges(j);
     Y=sin(yaw+orientacionLaser)*laserdata.Ranges(j);

     X1 = posdata.Pose.Pose.Position.X + X;
     Y1 = posdata.Pose.Pose.Position.Y + Y;
         if X1 == inf || X1 == -inf
          X1 = 0;
          Y1 = 0;
         end
    X1= double(X1);
    Y1= double(Y1);
    XY=[X1 Y1];

    setOccupancy(MAP, XY, 1);
    mapInflated = copy(MAP);
```

```
    inflate(mapInflated,0.2);
    show (mapInflated)
    hold on
    plot(startLocation(1,1),startLocation(1,2),'+r');
    end

        opc=questdlg({'Debe pinchar el punto de destino','¿De
acuerdo?'},'Información','Sí','Salir', 'Salir');
        if strcmp(opc,'Salir')
         clear,clc,close all;

        end

        robotGoal=ginput(1);

while volver == 0;
      volver = 1;

posdata = receive(posicion,4);
q =[posdata.Pose.Pose.Orientation.W posdata.Pose.Pose.Orientation.X
posdata.Pose.Pose.Orientation.Y posdata.Pose.Pose.Orientation.Z];
[yaw, pitch, roll] = quat2angle(q);

startLocation = [posdata.Pose.Pose.Position.X posdata.Pose.Pose.Position.Y];
plot(startLocation(1,1),startLocation(1,2),'+r');

mapInflated = copy(MAP);
inflate(mapInflated,0.2)
prm = robotics.PRM(mapInflated);
prm.NumNodes = 50;
prm.ConnectionDistance = 5;
hold on

show(mapInflated)
path = findpath(prm, startLocation, robotGoal);

%Por si el circuito es complicado y necesita mas nodos
while isempty(path)

    prm.NumNodes = prm.NumNodes + 10;

    update(prm);

    path = findpath(prm, startLocation, robotGoal);
end
show (prm)
%%%%   Vamos a intentar que se mueva siguiendo la ruta establecida
%%%%%   utilizando Pure Pursuit   %%%%

%%% COnfiguracion de parametros de Pure Pursuit %%%%
PP=robotics.PurePursuit;
PP.Waypoints=path;
PP.MaxAngularVelocity=0.3;
PP.LookaheadDistance=0.4;
PP.DesiredLinearVelocity=0.15;

posdata = receive(posicion,4);
current_orient=quat2eul([posdata.Pose.Pose.Orientation.W posdata.Pose.Pose.Orientation.X
posdata.Pose.Pose.Orientation.Y posdata.Pose.Pose.Orientation.Z]);
robotCurrentLocation = [posdata.Pose.Pose.Position.X posdata.Pose.Pose.Position.Y
current orient(1)];

distanceToGoal = norm(robotCurrentLocation(1:2) - robotGoal);

while (distanceToGoal>0.5)
    [v,w]=step(PP,robotCurrentLocation);
    velmsg.Linear.X = v;
    velmsg.Angular.Z= w;
    send(robot,velmsg);

    posdata = receive(posicion,4);
```

```
    current_orient=quat2eul([posdata.Pose.Pose.Orientation.W
posdata.Pose.Pose.Orientation.X posdata.Pose.Pose.Orientation.Y
posdata.Pose.Pose.Orientation.Z]);
    robotCurrentLocation = [posdata.Pose.Pose.Position.X posdata.Pose.Pose.Position.Y
current orient(1)];

  %%% Lectura de sonares delanteros %%%

    sonar 2 msg = receive (sonar 2 sub);
    sonar 3 msg = receive (sonar 3 sub);

    posdata = receive(posicion,4);
    q =[posdata.Pose.Pose.Orientation.W posdata.Pose.Pose.Orientation.X
posdata.Pose.Pose.Orientation.Y posdata.Pose.Pose.Orientation.Z];
    [yaw, pitch, roll] = quat2angle(q);

    PosicionRobot = [posdata.Pose.Pose.Position.X posdata.Pose.Pose.Position.Y yaw];
    robotCurrentPose = (PosicionRobot);

    %%% Actualizamos la distancia al objetivo %%
    distanceToGoal = norm(robotGoal - robotCurrentPose(1:2));

  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  %%%%%%% Se comprueba si va a chocar %%%%%%%%%%%%%

        if ((sonar_2_msg.Range_ <= 0.3 ) || (sonar_3_msg.Range_ <= 0.3))
               distanceToGoal = 0;
               volver=0;
               %%% Se para el robot si choca %%%
               velmsg.Angular.Z = 0;
               velmsg.Linear.X = 0;
               send(robot,velmsg);

        end
end       %% Final del bucle While

velmsg.Angular.Z = 0;
velmsg.Linear.X = 0;
send(robot,velmsg);

 %%%%%%%%%%%%%%%  Dibujamos el mapa %%%%%%%%%%%%

 laserdata = receive(laser_sub,4);
  posdata = receive(posicion,4);
  q =[posdata.Pose.Pose.Orientation.W posdata.Pose.Pose.Orientation.X
posdata.Pose.Pose.Orientation.Y posdata.Pose.Pose.Orientation.Z];
  [yaw, pitch, roll] = quat2angle(q);

     for j=1:length(laserdata.Ranges)

     orientacionLaser = laserdata.AngleMin + (j * laserdata.AngleIncrement);

   X=cos(yaw+orientacionLaser)*laserdata.Ranges(j);
   Y=sin(yaw+orientacionLaser)*laserdata.Ranges(j);

   X1 = posdata.Pose.Pose.Position.X + X;
   Y1 = posdata.Pose.Pose.Position.Y + Y;
    if X1 == inf || X1 == -inf
       X1 = 0;
       Y1 = 0;
   end
   X1= double(X1);
   Y1= double(Y1);
   XY=[X1 Y1];

    setOccupancy(MAP, XY, 1);
   end
     %%%%%%%%%% Hacemos que de media vuelta para no chocar %%%%%

               PosX = posdata.Pose.Pose.Position.X;
               PosY = posdata.Pose.Pose.Position.Y;
               PosicionRobot = [PosX PosY];
```

```
                AngFinal=angdiff(0,yaw-(pi));
                PosX1= PosX + 1*cos(AngFinal);
                PosY1= PosY + 1*sin(AngFinal);

                DestinoGiro = [PosX1 PosY1];

                velmsg.Angular.Z= 0.3;
                send(robot,velmsg);

                 %%%%%%% bucle para que gire hasta la orientacion correcta %%%%%%%
                 while (yaw <= (atan2(DestinoGiro(2)-PosicionRobot(2),DestinoGiro(1)-
PosicionRobot(1))-0.15) || (yaw >= (atan2(DestinoGiro(2)-
PosicionRobot(2),DestinoGiro(1)-PosicionRobot(1))+0.15)))

                 posdata = receive(posicion,4);
                 q =[posdata.Pose.Pose.Orientation.W posdata.Pose.Pose.Orientation.X
posdata.Pose.Pose.Orientation.Y posdata.Pose.Pose.Orientation.Z];
                 [yaw, pitch, roll] = quat2angle(q);
                 end

                 velmsg.Angular.Z= 0;
                 send(robot,velmsg);

 end %% Final del bucle while

function Salir_Callback(hObject, eventdata, handles)

opc=questdlg('¿Desea salir del programa?','SALIR','Sí','No','No');
if strcmp(opc,'No')
    return;
end
global robot velmsg
 velmsg.Linear.X= 0;
 velmsg.Angular.Z= 0;
 send(robot,velmsg);

clear, clc,close all
```

5.1 Control de un dron en Gazebo

Para el control de un dron en Gazebo se ha implementado un código que muestra la ejecución de movimientos en un laberinto 3D. Para la realización de dichos movimientos se hace uso de la odometría y se muestran por pantalla la lectura de los sensores. Antes de ejecutarse se desactiva la gravedad para que el dron no caiga al suelo.

```
close all

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%INICIALIZACION COMUNICACION ROS-MATLAB
%Es preferible ejecutar estas funciones en linea de comandos
%cuando se desee establecer la comunicación con ROS o cerrarla
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%rosshutdown;                %%Cierra conexiones previas con ROS
%rosinit('192.168.58.133');  %%Inicializacion comunicacion con ROS
                             %%Indicar la dirección IP de la máquina
                             %%virtual



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%SUSCRIPCIONES A LOS TOPICS QUE QUEREMOS LEER
%Para cada sensor a leer, nos suscribimos al topic correspondiente de ROS
```

```
%y creamos un mensaje en el que poder realizar las lecturas posteriormente
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%LASER
laser_sub = rossubscriber('/scan', rostype.sensor_msgs_LaserScan);
laser_msg = rosmessage(laser_sub);

%IMU
imu sub = rossubscriber('/raw imu', rostype.sensor msgs Imu);
imu msg = rosmessage(imu sub);

%SONAR TECHO
sonar_techo_sub = rossubscriber('/sonar_height1', rostype.sensor_msgs_Range);
sonar techo msg= rosmessage(sonar techo sub);

%SONAR SUELO
sonar_suelo_sub = rossubscriber('/sonar_height2', rostype.sensor_msgs_Range);
sonar suelo msg = rosmessage(sonar suelo sub);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%CREAMOS EL PUBLISHER PARA ENVIAR DATOS DE POSICIÓN
%Los datos de posición se deben escribir en el topic
%'/gazebo/set_model_state' de Gazebo
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
pos pub = rospublisher('/gazebo/set model state','gazebo msgs/ModelState');
pos msg = rosmessage(pos pub);
%Se rellena el mensaje de estado del robot:
%1.Se fija el robot sobre el cual se quiere modificar su estado (posición)
pos_msg.ModelName='quadrotor';
%2.Se fija el sistema de referencia sobre el que se aplica la traslacion/rotacion
pos msg.ReferenceFrame='world';
%4.Se fija la posición inicial por defecto (traslación, y orientación en
%cuaternio. Estos campos del mensaje son los que tenemos que modificar para
%enviar nuevas posiciones al quadrotor
pos msg.Pose.Position.X=0;
pos msg.Pose.Position.Y=0;
pos_msg.Pose.Position.Z=0.3;  %0.3 es la Z minima (altura de las patas)
pos_msg.Pose.Orientation.X=0;
pos_msg.Pose.Orientation.Y=0;
pos_msg.Pose.Orientation.Z=0;
pos msg.Pose.Orientation.W=1;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%CREAMOS EL CLIENTE PARA LLAMAR A SERVICIOS DE GAZEBO
%El objetivo de esta parte es desactivar la física de la gravedad en el
%simulador
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
srv_client = rossvcclient('/gazebo/set_physics_properties');
%Crea la peticion ("mensaje") para enviar al servicio (rellenar los campos)
svc_req=rosmessage(srv_client);
%Se rellena la peticion de servicio para anular la gravedad y poder simular la posicion
del robot desde Matlab sin tener en cuenta las fisicas del robtot que carga ROS
%Es importante mantener los parametros por defecto
%Se anula la gravedad
svc_req.Gravity.X=0;
svc_req.Gravity.Y=0;
svc req.Gravity.Z=0;
%Se fija el timestep de la simulacion a 0.001s
svc req.TimeStep=0.001;
%Resto de parametros del solver por defecto
svc_req.OdeConfig.AutoDisableBodies=0;
svc_req.OdeConfig.SorPgsPreconIters=0;
svc req.OdeConfig.SorPgsIters=10;
svc req.OdeConfig.SorPgsW=1.3;
svc_req.OdeConfig.SorPgsRmsErrorTol=0;
svc_req.OdeConfig.ContactSurfaceLayer=0.001;
svc_req.OdeConfig.ContactMaxCorrectingVel=10.0;
svc req.OdeConfig.Cfm=0.0;
svc req.OdeConfig.Erp=0.2;
svc_req.OdeConfig.MaxContacts=20;
%Llamada al servicio. Se deshabilita la gravedad en la simulacion y se fija el timestep
call(srv_client,svc_req,'Timeout',3);



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%ENVÍO DE MOVIMIENTOS AL DRONE
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%Avance del drone a lo largo del eje X durante 20 iteraciones, a una altura
%de 0.5 metros
for i=0:20
    %Lectura de sensores
    imu msg = receive(imu sub);
    sonar techo msg = receive (sonar techo sub);
    sonar_suelo_msg = receive (sonar_suelo_sub);
    laser_msg = receive(laser_sub);  %Lectura directa del láser, que está hacia abajo

    %Incremento de la coordenada X de posición y envío de los datos
    pos msg.Pose.Position.X=pos msg.Pose.Position.X+0.1;
    pos msg.Pose.Position.Z=0.5;
    send(pos_pub,pos_msg);

    %Representacion gráfica de los datos del láser. Como el laser está
    %hacia abajo, para "darle la vuelta" hay que invertir el orden de las
    %distancias leídas
    distancias(length(laser_msg.Ranges):-1:1)=laser_msg.Ranges;
    laser_msg.Ranges=distancias;
    plot(laser msg,'MaximumRange',8);

    %Muestro por pantalla los datos de la imu
    wx=imu msg.AngularVelocity.X;
    wy=imu_msg.AngularVelocity.Y;
    wz=imu_msg.AngularVelocity.Z;
    ax=imu msg.LinearAcceleration.X;
    ay=imu msg.LinearAcceleration.Y;
    az=imu msg.LinearAcceleration.Z;
    cadena=sprintf('IMU:: Velocidad angular=[%f %f %f]  Aceleración lineal=[%f %f
%f]',wx,wy,wz,ax,ay,az);
    disp(cadena);

    %Muestro por pantalla los datos de los sonares
    cadena=sprintf('SONAR SUELO:: Rango=%f   SONAR TECHO::
Rango=%f',sonar_suelo_msg.Range_,sonar_techo_msg.Range_);
    disp(cadena);

end


%Retroceso del dron a lo largo del eje X durante 40 iteraciones, a 0.5
%metros de altura
for i=21:60
    %Lectura de sensores
    imu_msg = receive(imu_sub);
    sonar_techo_msg = receive (sonar_techo_sub);
    sonar_suelo_msg = receive (sonar_suelo_sub);
    laser msg = receive(laser sub); %lecturas directas del láser, que está hacia abajo

    %Decremento de la coordenada X y envío de los datos
    pos_msg.Pose.Position.X=pos_msg.Pose.Position.X-0.1;
    pos_msg.Pose.Position.Z=0.5;
    send(pos pub,pos msg);

    %Representacion gráfica de los datos del láser. Como el laser está
    %hacia abajo, para "darle la vuelta" hay que invertir el orden de las
    %distancias leídas
    distancias(length(laser_msg.Ranges):-1:1)=laser_msg.Ranges;
    laser msg.Ranges=distancias;
    plot(laser msg,'MaximumRange',8);

    %Muestro por pantalla los datos de la imu
    wx=imu_msg.AngularVelocity.X;
    wy=imu_msg.AngularVelocity.Y;
    wz=imu msg.AngularVelocity.Z;
    ax=imu_msg.LinearAcceleration.X;
    ay=imu_msg.LinearAcceleration.Y;
    az=imu msg.LinearAcceleration.Z;
    cadena=sprintf('IMU:: Velocidad angular=[%f %f %f]  Aceleración lineal=[%f %f
%f]',wx,wy,wz,ax,ay,az);
    disp(cadena);
```

```
    %Muestro por pantalla los datos de los sonares
    cadena=sprintf('SONAR SUELO:: Rango=%f   SONAR TECHO::
Rango=%f',sonar_suelo_msg.Range_,sonar_techo_msg.Range_);
    disp(cadena);

end

%Descenso del dron a lo largo del eje Z durante 40 iteraciones
for i=101:140
    %Lectura de sensores
    imu_msg = receive(imu_sub);
    sonar_techo_msg = receive (sonar_techo_sub);
    sonar_suelo_msg = receive (sonar_suelo_sub);
    laser_msg = receive(laser_sub); %Lectura directa del láser, que está hacia abajo

    %Decremento de la coordenada Z y envío de los datos
    pos_msg.Pose.Position.Z=pos_msg.Pose.Position.Z-0.1;
    send(pos_pub,pos_msg);

    %Representacion gráfica de los datos del láser. Como el laser está
    %hacia abajo, para "darle la vuelta" hay que invertir el orden de las
    %distancias leídas
    distancias(length(laser_msg.Ranges):-1:1)=laser_msg.Ranges;
    laser_msg.Ranges=distancias;
    plot(laser_msg,'MaximumRange',8);

    %Muestro por pantalla los datos de la imu
    wx=imu_msg.AngularVelocity.X;
    wy=imu_msg.AngularVelocity.Y;
    wz=imu_msg.AngularVelocity.Z;
    ax=imu_msg.LinearAcceleration.X;
    ay=imu_msg.LinearAcceleration.Y;
    az=imu_msg.LinearAcceleration.Z;
    cadena=sprintf('IMU:: Velocidad angular=[%f %f %f]  Aceleración lineal=[%f %f
%f]',wx,wy,wz,ax,ay,az);
    disp(cadena);

    %Muestro por pantalla los datos de los sonares
    cadena=sprintf('SONAR SUELO:: Rango=%f   SONAR TECHO::
Rango=%f',sonar_suelo_msg.Range_,sonar_techo_msg.Range_);
    disp(cadena);

end
```

```
    disp(cadena);

    %Muestro por pantalla los datos de los sonares
    cadena=sprintf('SONAR SUELO:: Rango=%f   SONAR TECHO::
Rango=%f',sonar_suelo_msg.Range_,sonar_techo_msg.Range_);
    disp(cadena);

end

%Giro positivo del dron entorno al eje Z durante 40 iteraciones
giro=0;
for i=141:180
    %Lectura de sensores
    imu msg = receive(imu sub);
    sonar techo msg = receive (sonar techo sub);
    sonar_suelo_msg = receive (sonar_suelo_sub);
    laser_msg = receive(laser_sub); %Lectura directa de los datos del láser, que está
hacia abajo

    %Incremento del yaw.
    giro=giro+deg2rad(5);   %Aumentamos 5 grados el yaw
    eulerZYX=[giro 0 0];
    quat=eul2quat(eulerZYX);
    pos msg.Pose.Orientation.X=quat(2);
    pos msg.Pose.Orientation.Y=quat(3);
    pos msg.Pose.Orientation.Z=quat(4);
    pos_msg.Pose.Orientation.W=quat(1);

    send(pos_pub,pos_msg);

    %Representacion gráfica de los datos del láser. Como el laser está
    %hacia abajo, para "darle la vuelta" hay que invertir el orden de las
    %distancias leídas
    distancias(length(laser msg.Ranges):-1:1)=laser msg.Ranges;
    laser msg.Ranges=distancias;
    plot(laser msg,'MaximumRange',8);

    %Muestro por pantalla los datos de la imu
    wx=imu_msg.AngularVelocity.X;
    wy=imu msg.AngularVelocity.Y;
    wz=imu msg.AngularVelocity.Z;
    ax=imu msg.LinearAcceleration.X;
    ay=imu_msg.LinearAcceleration.Y;
    az=imu_msg.LinearAcceleration.Z;
    cadena=sprintf('IMU:: Velocidad angular=[%f %f %f]  Aceleración lineal=[%f %f
%f]',wx,wy,wz,ax,ay,az);
    disp(cadena);

    %Muestro por pantalla los datos de los sonares
    cadena=sprintf('SONAR SUELO:: Rango=%f   SONAR TECHO::
Rango=%f',sonar suelo msg.Range ,sonar techo msg.Range );
    disp(cadena);

end
```

6 Pliego de condiciones

En este apartado se incluyen los medios necesarios para la realización del proyecto y sus características.

6.1 Hardware

- **Portátil ACER Aspire E 15**
  - Procesador: *Intel® Core™ i5-4210 1,7 GHz*
  - Memoria RAM: *8 Gb DDR3*
  - Tarjeta gráfica: *NVIDIA® 840M con 2 Gb dedicados*

6.2 Software

- Windows 10 © Microsoft Corporation
- MATLAB 2015a (con toolbox *Robotics System*)
- Máquina virtual VMware Workstation 12 player con Ubuntu 14.04
- Robot Operating System (ROS) corriendo en Ubuntu 14.04

7 Presupuesto

En esta sección se desarrollan los costes teóricos del proyecto, desglosados en costes materiales y en costes de personal.

7.1 Costes materiales

Los costes materiales incluyen los gastos en hardware y en software necesarios para la realización del proyecto.

Objeto		Cantidad	Precio Unidad (€)	Precio Total (€)
Hardware	Portátil ACER	1	569,00.-	569,00.-
Software	Microsoft Windows 10	1	0,00.-[1]	0,00.-
	Matlab 2015a	1	0,00.-[2]	0,00.-
	VMware workstation 12 Player	1	134,95.-	134,95.-
	ROS	1	0,00.-[3]	0,00.-
		Total:		**703,95.-**

Tabla 1 Costes materiales

[1] El precio del Sistema Operativo Windows 10 está incluido en el precio del ordenador portátil.

[2] La versión de Matlab 2015a es una versión de estudiante gratuita aportada por la universidad.

[3] Robot Operating System es software libre.

7.2 Costes de personal

Los costes de personal incluyen los costes de realización del proyecto correspondientes al tiempo dedicado a su desarrollo.

Objeto	Tiempo (meses)	Coste unidad (€/mes)	Coste total (€)
Investigación y desarrollo	5	1.400,00.-	7.000,00.-
Escritura del proyecto	2	700,00.-	1.400,00.-
	Total:		**8.400,00.-**

Tabla 2 Costes de personal

7.3 Costes totales

Los costes totales del proyecto se obtienen aplicando el IVA profesional a la suma de los costes materiales y de personal.

Objeto	Coste total (€)
Coste material	703,95.-
Coste de personal	8.400,00.-
Subtotal	9.103,95.-
IVA (21%)	1.911,83.-
Total:	**11.015,78.-**

Tabla 3 Costes totales

8 Manual de usuario

En esta sección se detalla cómo ejecutar y utilizar las diferentes aplicaciones creadas; además se incluye una guía de botones para un mejor entendimiento del funcionamiento.

8.1 Arranque del sistema

Para poder ejecutar las aplicaciones primero debe estar unido Matlab con ROS. Para ello se ejecuta la máquina virtual, en este caso la VMware Workstation 12 y se pincha una vez en el botón *Open a Virtual Machine* para abrir un explorador donde buscar la máquina virtual con Ubuntu y ROS. Se selecciona la máquina virtual y se ejecuta.

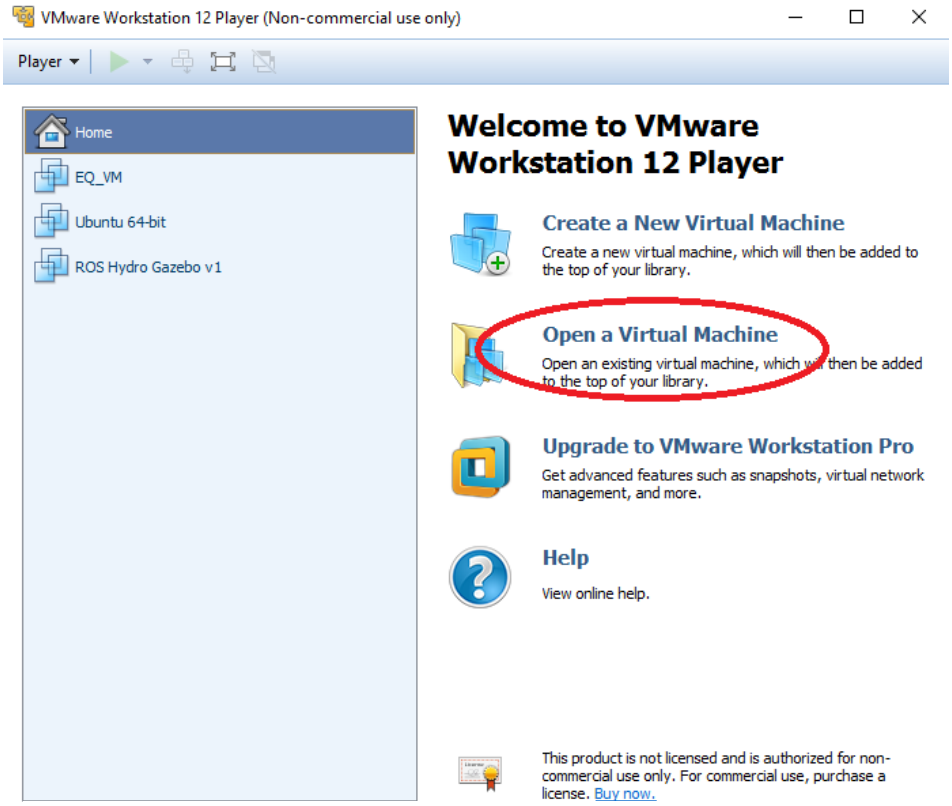

Ilustración 72 Botón para abrir una máquina virtual en VMware

Una vez lanzado Linux hay que seguir unos pasos previos explicados en el apartado *2.4.2 Comunicación con ROS* antes de poder ejecutar ROS. Después en el terminal se introduce el comando siguiente para ejecutarlo:

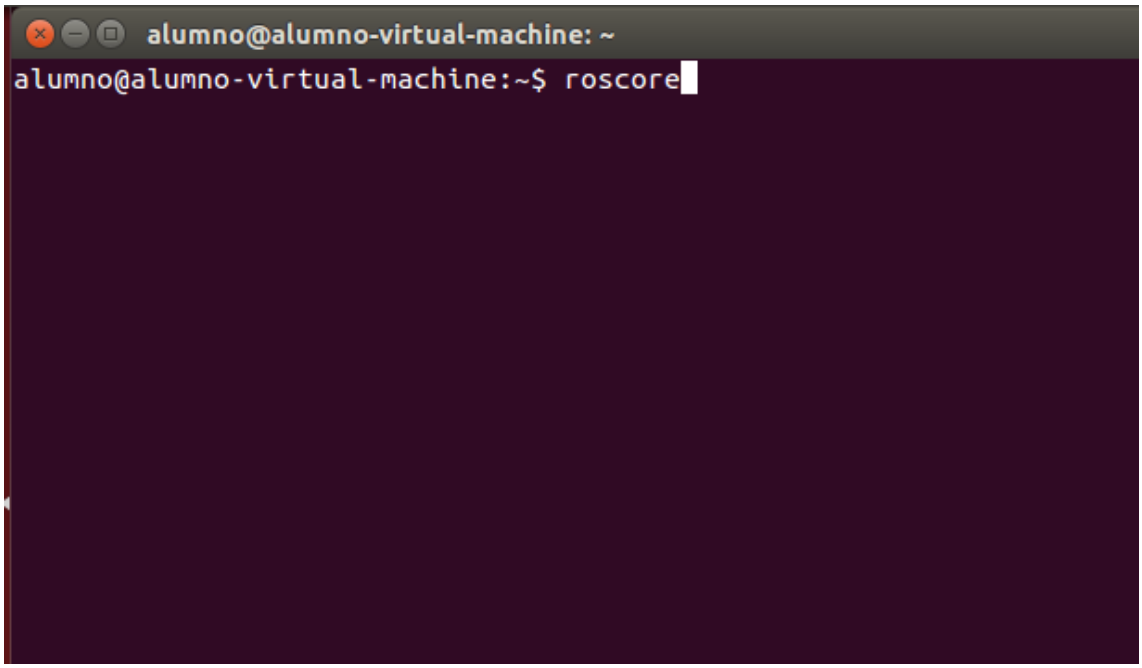

Ilustración 73 ejecutar comando roscore en el terminal de Linux

Para configurar el simulador STDR se debe seguir el apartado *3.1.1 Configuración del simulador STDR* donde se explica cómo hacerlo. Escribiendo en el terminal *roslaunch practica_1 amigobot.launch* como en la ilustración 74 se abre el simulador STDR con el robot sobre el que se han basado las aplicaciones, con el láser y los sónares en posiciones concretas.

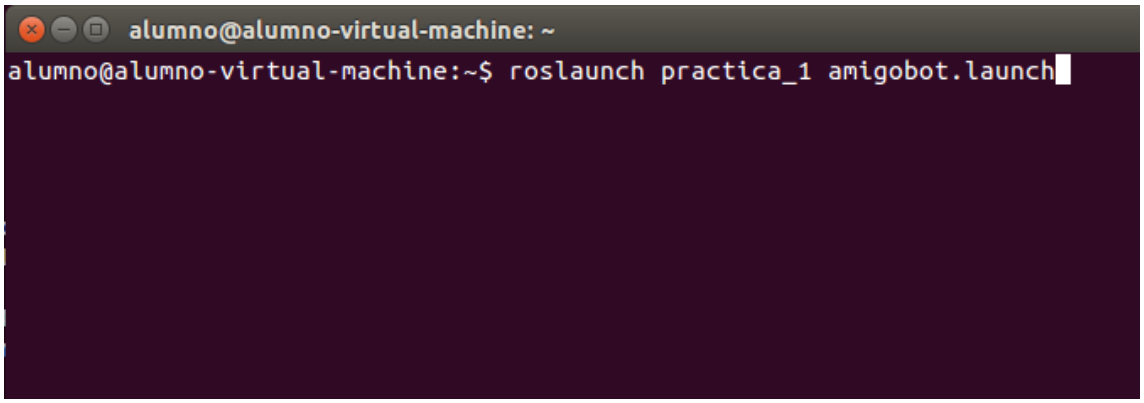

Ilustración 74 Iniciar el simulador STDR con el robot personalizado Amigobot

Gazebo se abre de una forma similar. Para ejecutarlo con el mapa y el dron de nuestra aplicación, si se han seguido las configuraciones del apartado *3.2.1 Configuración de Gazebo,* en el terminal se debe poner el nombre del *launch* correspondiente cómo en la ilustración inferior.

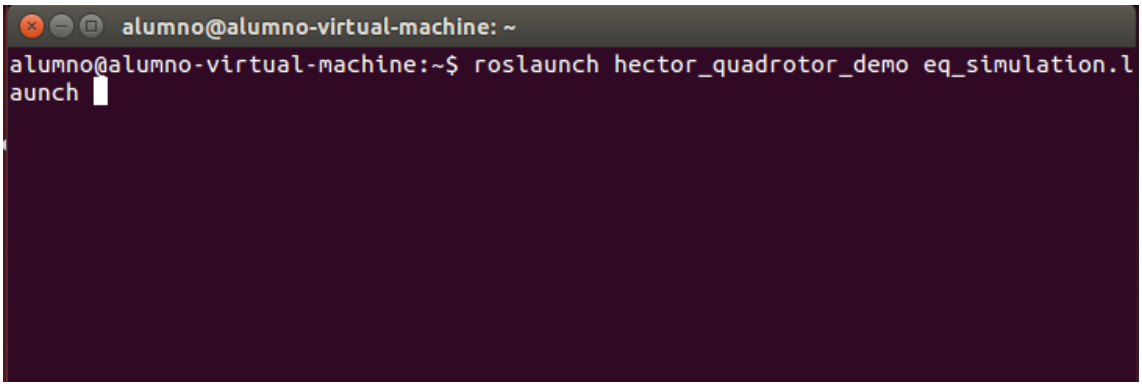

Ilustración 75 Terminal con el comando para ejecutar Gazebo

Ahora que ya está ROS iniciado hay que unirlo a Matlab. Para iniciarlo se ejecuta la aplicación y se escribe en línea de comandos *Rosinit* y la IP, que se debe conocer si se han seguido los pasos anteriores, de la siguiente manera:

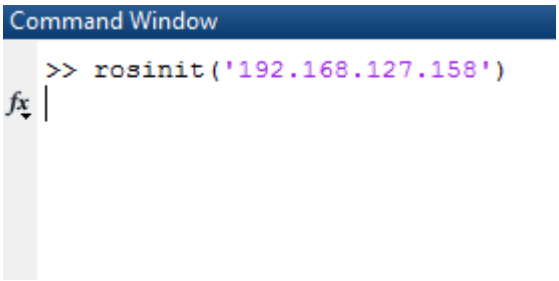

Ilustración 76 iniciar ROS en Matlab

Si se quiere comprobar si la unión está bien hecha, se puede utilizar en Matlab cualquiera de los comandos que permiten ver los topics o los nodos, como *rostopic list* o *rosnode list*. En la siguiente ilustración se ve que el comando *rostopic list* da una lista de los topics existentes en la red de ROS, entre ellos los topics del simulador STDR y el robot, por lo que la unión está bien hecha.

```
>> rosinit('192.168.127.158')
Initializing global node /matlab_global_node_23374 with NodeURI http://192.168.127.1:63152/
>> rostopic list
/map
/map_metadata
/robot0/cmd_vel
/robot0/laser_1
/robot0/odom
/robot0/sonar_0
/robot0/sonar_1
/robot0/sonar_2
/robot0/sonar_3
/robot0/sonar_4
/robot0/sonar_5
/robot0/sonar_6
/robot0/sonar_7
/rosout
/rosout_agg
/stdr_server/active_robots
/stdr_server/co2_sources_list
/stdr_server/delete_robot/cancel
```

Ilustración 77 Rosinit iniciado y lista de topics

Además de los topics básicos también aparecen los de cada sensor del robot, así como del mapa, la velocidad o la odometría.

8.2 Ejecución de aplicaciones

Para iniciar las aplicaciones una vez realizado el arranque del sistema, hay diversas maneras. Una de las formas es buscar en el explorador de Windows las aplicaciones y hacer doble click sobre ellas, pero se va a usar una diferente y que permite hacer todo desde Matlab. En la imagen inferior está señalado el explorador donde buscar las aplicaciones.

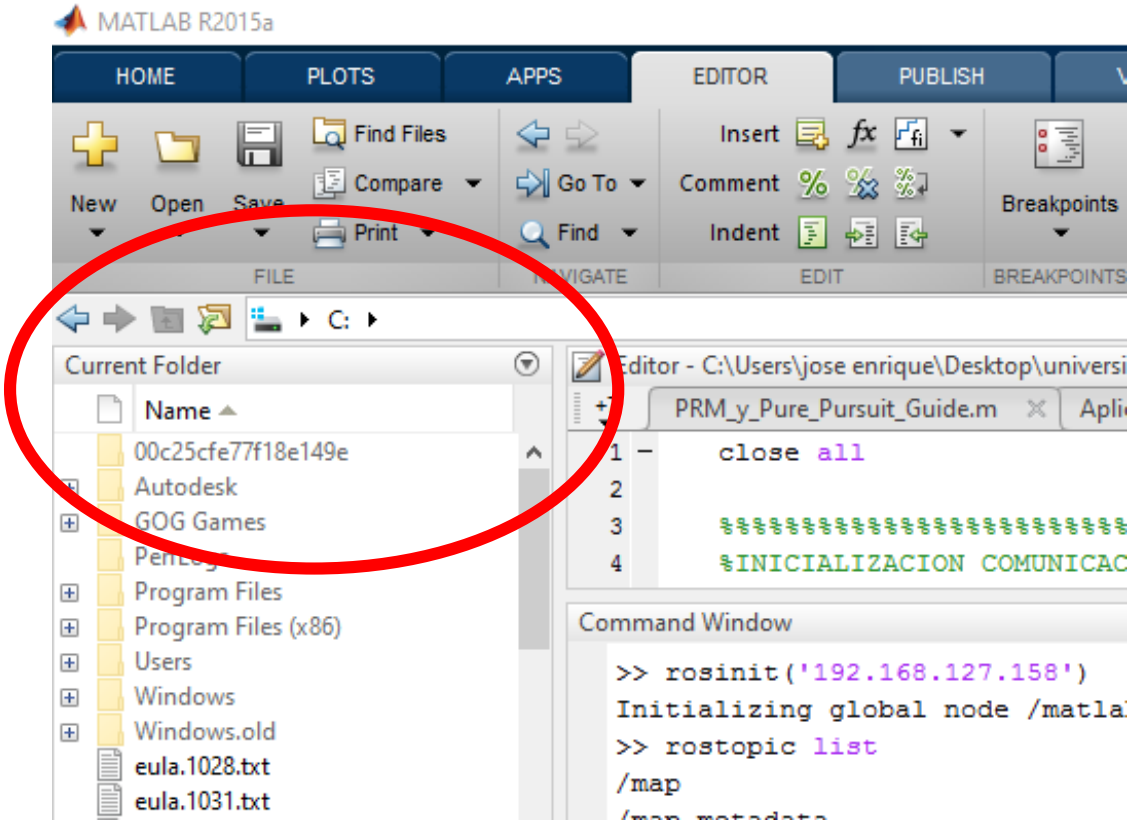

Ilustración 78 Explorador donde buscar las aplicaciones en Matlab

Una vez se tenga la carpeta con las aplicaciones ya abierta, hay dos formas de ejecutarlas, dando doble click en el .m de la aplicación y pulsando el botón *Run* que aparece en las opciones de la barra superior o escribiendo su nombre en la ventana de comandos.

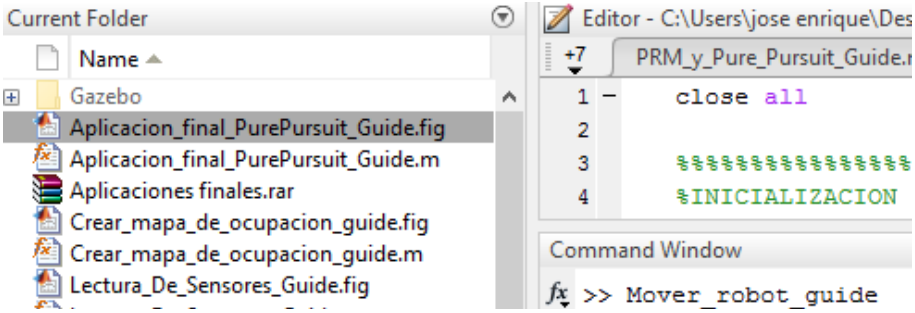

Ilustración 79 Ejecución de las aplicaciones

8.2.1 Aplicación Mover Robot (STDR)

Cuando esta aplicación se inicia una interfaz con cuatro botones y un cuadro editable. Este cuadro, señalado en la imagen inferior, permite configurar la longitud de avance del robot, la cual de forma inicial será 1 metro.

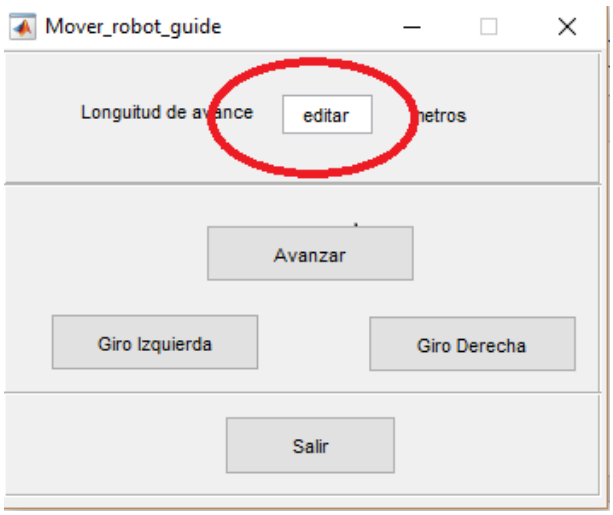

Ilustración 80 Cuadro de configuración de longitud de avance

El robot puede realizar tres acciones, avanzar hacia delante la longitud configurada, girar a la derecha 90 grados, o girar a la izquierda otros 90º. Estos botones están indicados en las ilustraciones inferiores.

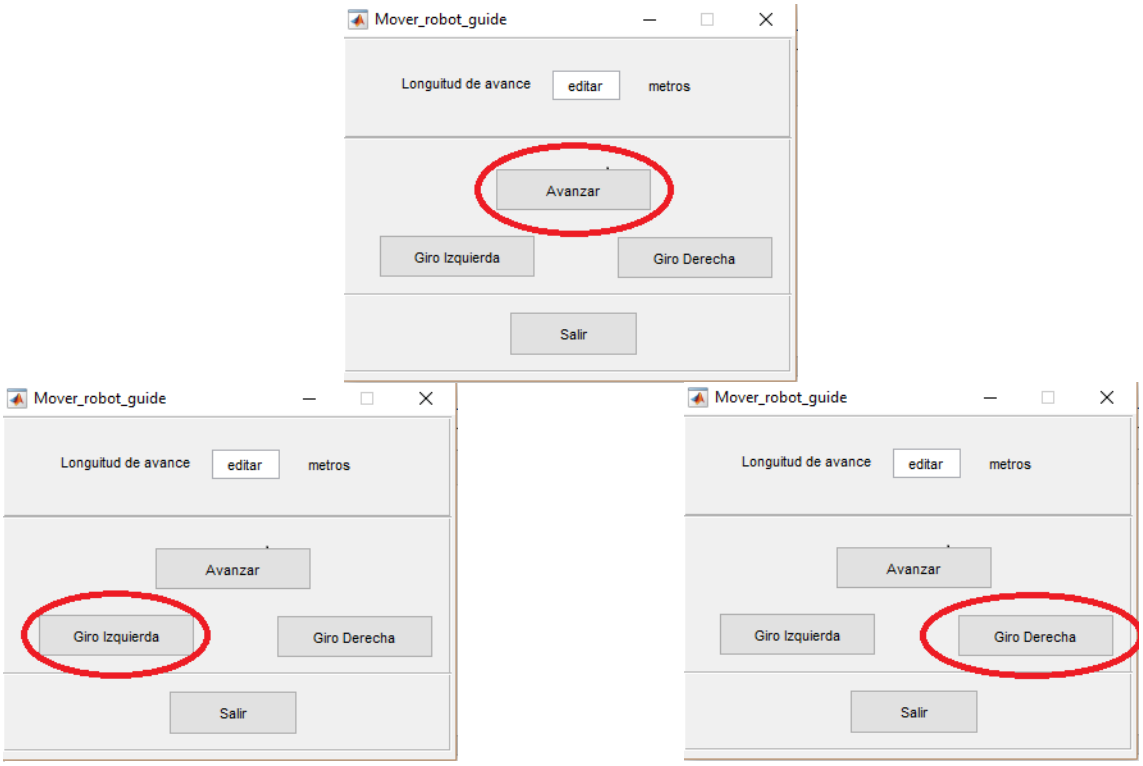

Ilustración 81 Arriba, botón Avanzar, abajo a la izda, botón Giro Izquierda, y abajo a la dcha, botón Giro Derecha

Por último, esta aplicación, al igual que las posteriores, contiene un botón de *Salir*, que permite, con una ventana adicional, cerrar el programa y salir de la aplicación.

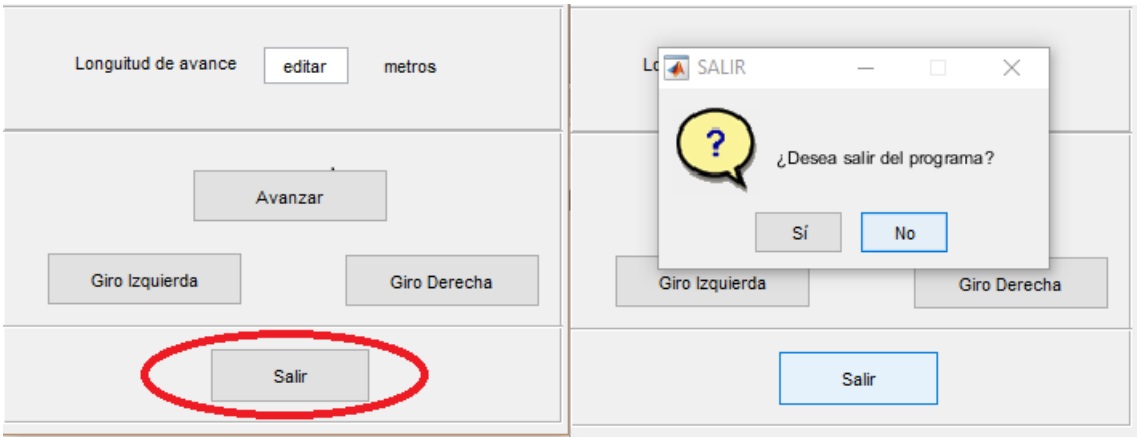

Ilustración 82 Botón Salir y ventana adicional para ello

8.2.2 Aplicación Lectura de Sensores (STDR)

La interfaz de esta aplicación tan sólo tiene dos botones y un eje de coordenadas, el botón *Comenzar* dibuja en el eje de coordenadas las lecturas de los sensores, es decir, del láser y de los sónares.

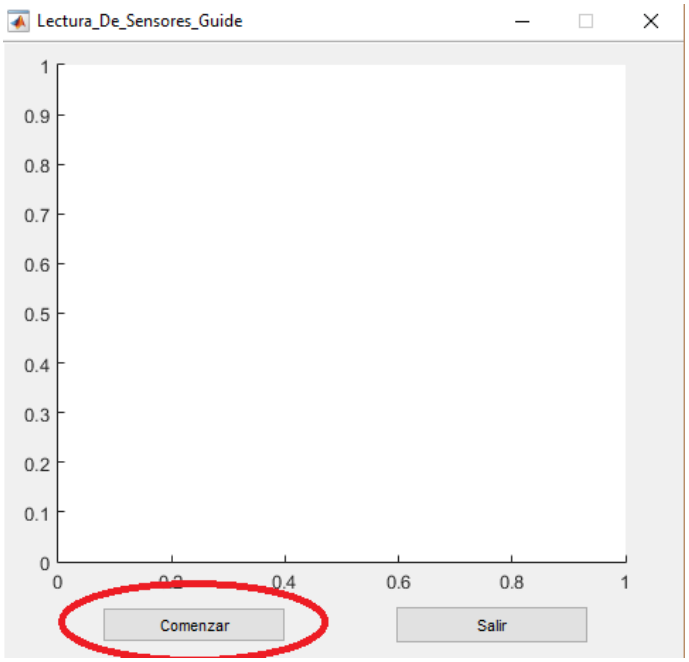

Ilustración 83 Interfaz de la aplicación Lectura de sensores

El botón *Salir* permite, a través de una ventana emergente, cerrar la aplicación y salir de ella.

8.2.3 Aplicación Control de movimiento con parada de emergencia (STDR)

Esta aplicación, de nombre *movimiento_Con_evitacion_Guide.m*, permite mover el robot y además evita que choque haciendo uso de los sensores. La interfaz contiene los botones para el movimiento del robot, de la misma forma que la aplicación *Mover Robot*, un nuevo cuadro configurable y añade un eje de coordenadas donde se dibuja la lectura de los sensores.

Ilustración 84 Interfaz movimiento con evitacion

En el cuadro configurable se puede modificar la distancia de seguridad, es decir, la distancia mínima a la que puede estar el robot de un obstáculo. Superado este límite el robot se para y salta una ventana emergente que avisa de ello.

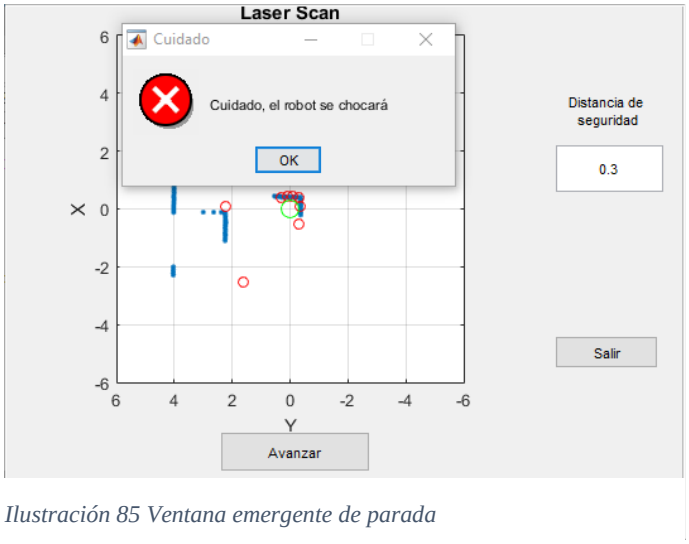

Ilustración 85 Ventana emergente de parada

8.2.4 Aplicación Lectura de mapa de ocupación (STDR)

La interfaz de esta aplicación es bastante simple, tan sólo tiene dos botones, el botón *Salir* común a todas las aplicaciones y el botón *Dibujar Mapa* que leyendo el topic del mapa del simulador STDR, lo dibuja en el eje de coordenadas superior.

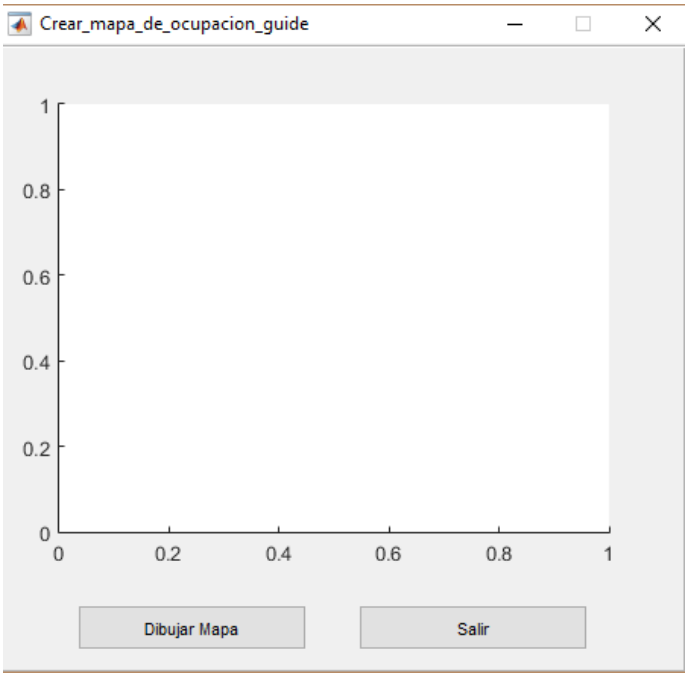

Ilustración 86 Interfaz Aplicación Crear mapa de ocupación

8.2.4 Aplicaciones de Planificación global con PRM (STDR)

Esta sección engloba tres aplicaciones, *Planificación global con PRM y local básico*, *Planificación global con PRM y local con Pure Pursuit* y *Planificación global con PRM y mapa desconocido*, pues las tres tienen la misma interfaz, cambiando únicamente la forma de realizar su tarea.

Su función es dirigirse a un punto de destino marcado por el usuario en el eje de coordenadas. Estas aplicaciones tienen una interfaz con dos botones y el eje de coordenadas propiamente dicho. El botón *Comenzar* da inicio a la aplicación y el botón *Salir* permite salir de ella de la misma forma que en las anteriores.

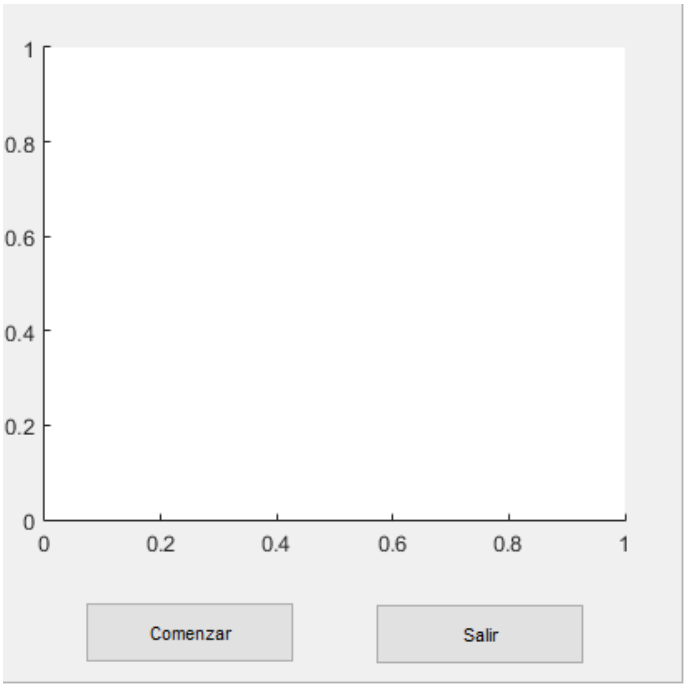

Ilustración 88 Interfaz de las aplicaciones con PRM

Una vez que se pulsa el botón comenzar, aparece una ventana emergente que pide que se pinche en el punto de destino. Al pulsar en *Sí* se puede ver el mapa y una cruz roja en la posición del robot. Hay que pinchar el punto de destino elegido para que la aplicación pueda seguir ejecutándose.

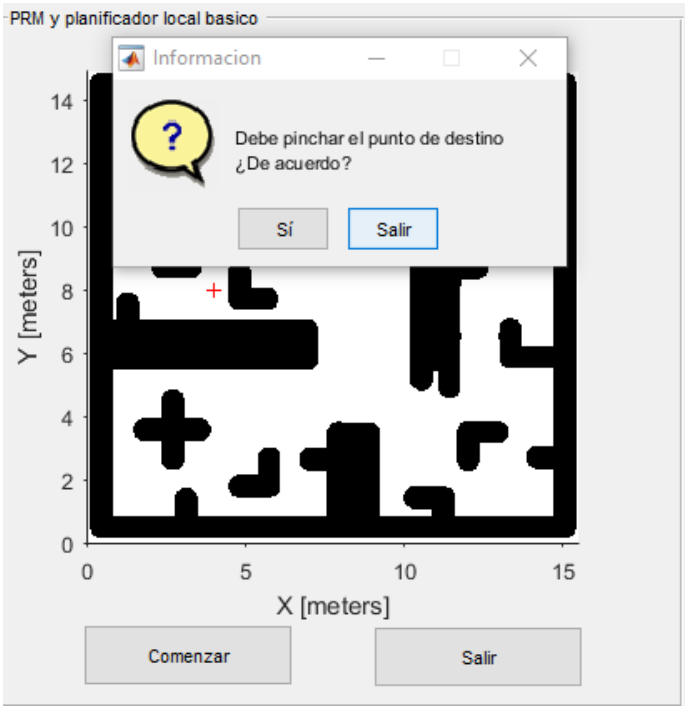

Ilustración 87 Ventana emergente de inicio de la aplicación

8.2.5 Aplicación Mover el robot de forma manual (STDR con Simulink)

Para iniciar aplicaciones en Simulink, hay que dar doble click en la aplicación correspondiente, de extensión .mdl. Esto abre un dibujo de bloques que corresponde al de la aplicación iniciada. Para ejecutarla, hay que pinchar en el símbolo de *Play* destacado en la figura inferior.

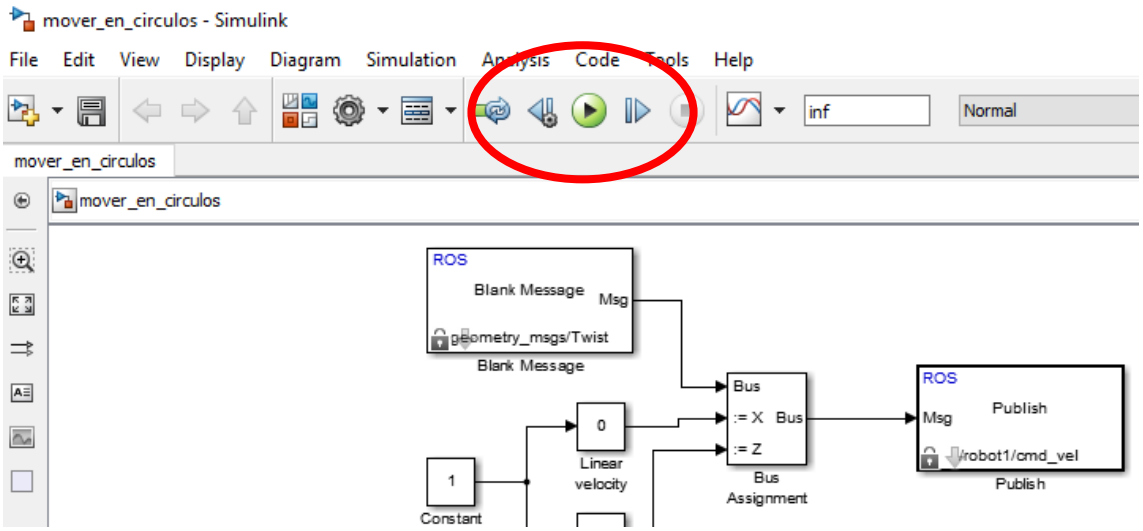

Ilustración 89 Botón play en Simulink

En cuanto a la aplicación, esta permite gracias a dos *sliders* controlar la velocidad lineal y angular del robot, una por cada *slider*. El *slider* para la velocidad lineal tiene un rango de 0 m/seg hasta 2 m/seg y el *slider* para la velocidad angular permite variar esta entre 1 rad/seg y -1 rad/seg, para poder girar hacia los dos lados. En la ilustración 90 se pueden ver estos dos *sliders.*

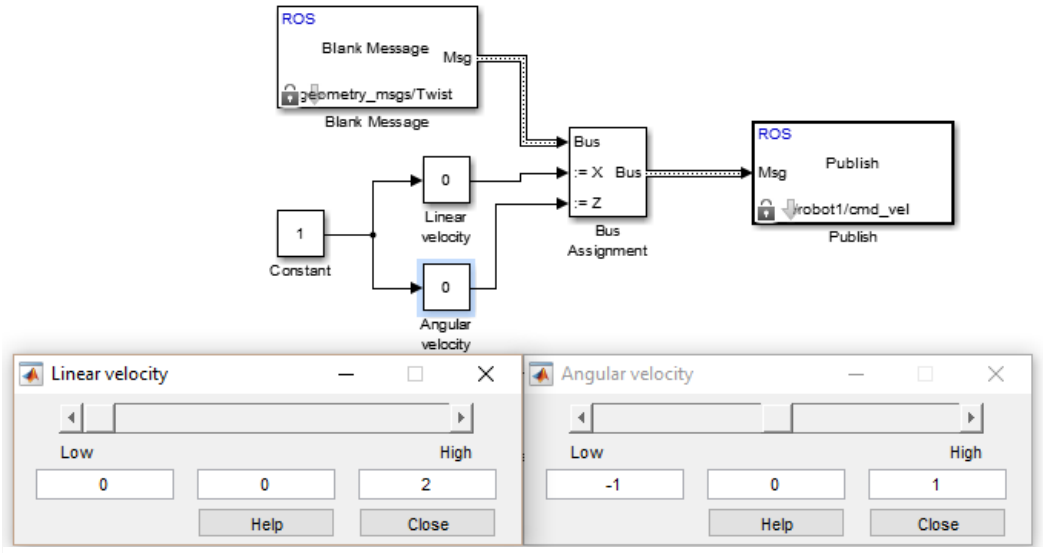

Ilustración 90 Sliders de la aplicación Mover el robot de forma manual

8.2.5 Controlador de movimiento con realimentación (STDR con Simulink)

Esta aplicación se abre y ejecuta de la misma forma que la anterior, pero este caso, existen bloques que pueden ser modificables. El primero de ellos se encuentra en la capa superior, y marca el punto de destino para el robot. En él hay que configurar el punto al que se quiere que el robot se dirija.

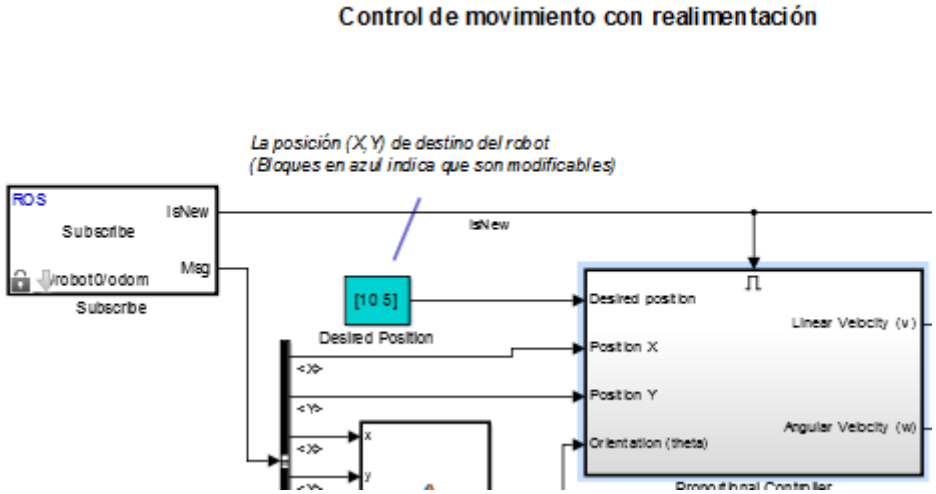

Ilustración 91 Bloque de modificación del punto de destino

Dentro del bloque *Proportional Controler* hay varios bloques configurables; por un lado está la distancia límite, que será la distancia mínima que los sensores del robot pueden detectar un objeto sin parar el robot por peligro próximo. Por otro lado están las velocidades, la velocidad lineal, cuyo bloque comparte nombre, y la velocidad angular, cuyo bloque se denomina *Ganancia*. Estos dos últimos bloques se modifican a través de un *slider*. En la figura inferior se puede ver la posición de estos.

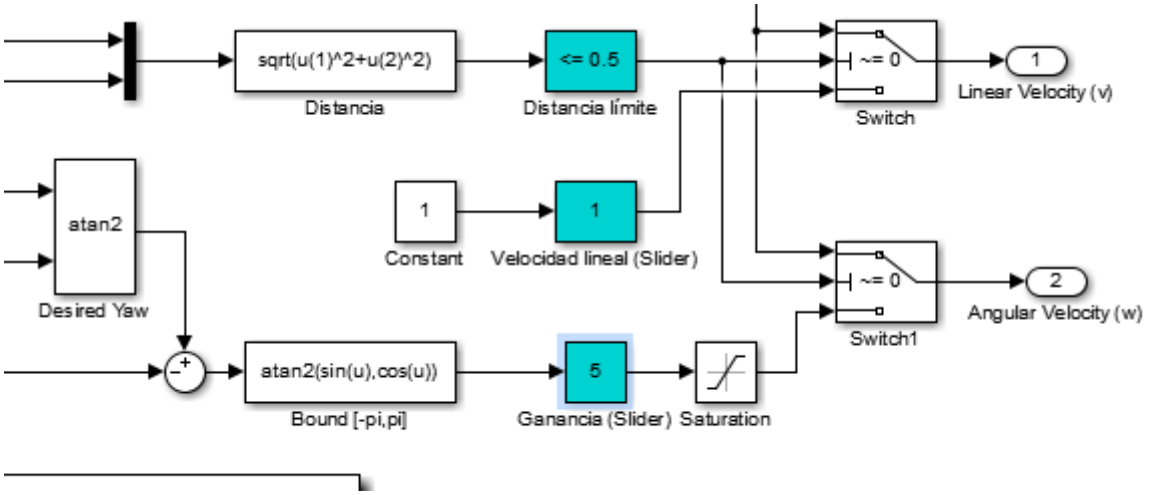

Ilustración 92 Bloques configurables del Control Proporcional

8.2.5 Aplicación Simulación Dron (Gazebo)

Esta aplicación no dispone de interfaz, por lo que se ejecuta su código .m directamente de la manera antes expuesta, escribiendo su nombre, *Sim_Drone,* en el terminal o haciendo doble click en ella y dando al botón *RUN*. Una vez se ejecuta la aplicación el dron simulado en Gazebo realiza usa serie de movimientos y la lectura de sus sensores se dibuja y ve por pantalla.

8.2.5 Aplicación Simulación Dron (Gazebo con Simulink)

Realiza la misma función que la aplicación anterior, pero ha sido desarrollada en Simulink. Antes de poder ejecutarse se debe iniciar el script *ini_simulink.m* que realiza las siguientes acciones:

1. Definición del valor del periodo de muestreo para el envío de datos al dron
2. Establecimiento de la conexión con ROS
3. Envío a Gazebo de la solicitud de desactivación de la gravedad. Si esta parte no se ejecuta, el dron "caerá" al suelo en cuanto dejen de enviársele comandos de posición.

Una vez ejecutado el archivo de inicialización, se puede iniciar la aplicación cómo se muestra en las anteriores aplicaciones sobre Simulink.

9 Bibliografía

[1] http://robotiica.blogspot.com.es/2007/10/historia-de-la-robtica.html

[2] Emilio Casamayor, "Robótica.Robocup" Sistemas inteligentes. 5º curso. Universidad Politécnica de Madrid.

[3] Elena López Guillén, Manuel Ocaña, "Introducción al guiado autónomo de vehículos" Máster Universitario en Sistemas Electrónicos Avanzados. Universidad de Alcalá.

[4] Elena López Guillén, Manuel Ocaña, "Entorno de desarrollo para aplicaciones robóticas" Máster Universitario en Sistemas Electrónicos Avanzados. Universidad de Alcalá.

[5] Página web oficial de ROS (online). Disponible: http://www.ros.org/

[6] Manual de usuario de la toolbox *Robotics System* de Matlab (online). Disponible:

http://es.mathworks.com/help/robotics/index.html

[7] Alberto Elfes, "Using Occupancy Grids for Mobile Robot Perception and Navigation" Carnegie Mellon University. Disponible: http://www.sci.brooklyn.cuny.edu/~parsons/courses/3415-fall-2011/papers/elfes.pdf

[8] Sebastian Thrun, "Probabilistics Robotics" (online). Disponible:

https://docs.ufpr.br/~danielsantos/ProbabilisticRobotics.pdf

[9] R. Craig Coulter, "Implementation of the Pure Pursuit Path Tracking Algorithm" The robotic Institute, Pittsburgh, Pennsylvania.

Universidad de Alcalá
Escuela Politécnica Superior

ESCUELA POLITECNICA
SUPERIOR

Universidad
de Alcalá