



Trabajo de Fin de Grado

**Implementación de localización geométrica para
robots móviles bajo ROS.**

Autor:

Marina Galli

Tutor:

Ramón Barber Castaño

Cotutor:

Luis Santiago Garrido Bullón



Índice general

1	Introducción	1
1.1	Objetivos	2
1.2	Estructura del trabajo	2
2	Conceptos básicos	5
2.1	Robot móvil	5
2.1.1	Locomoción	6
2.1.2	Matlab	10
2.2	ROS	11
2.2.1	Introducción a entorno <i>ROS</i>	12
2.2.2	Stage	14
2.2.3	Gazebo	14
2.2.4	URDF	15
2.2.5	Rviz	16
3	Conexión Matlab ROS	17
3.1	Configuración de la conexión	17
3.2	Intercambio de información	19
3.2.1	Recibir mensajes	19
3.2.2	Enviar mensajes	24
3.3	Procesado de datos	25
3.4	Bucle de control	26
4	Integración de algoritmos	27
4.1	Teleoperación	27
4.2	Wanderer	28
4.3	Localización	29
4.3.1	Filtro de partículas	30
4.4	Mapping	33
4.5	SLAM	35
4.6	Planificador de trayectorias	35
4.6.1	PRM	37
4.6.2	Fast Marching Square	39
4.7	Seguidor de trayectorias	40
4.7.1	PurePursuit	41
4.8	Programa autónomo	43
5	Implementación	45
5.1	Mapping	45
5.1.1	BinaryOccupancyGrid	46
5.2	Teleoperación	46
5.3	Wanderer	49

5.4	Localización	50
5.5	Path planning	54
5.5.1	PRM	54
5.5.2	FM2	55
5.6	PurePursuit	55
5.7	Programa autónomo	57
6	Resultados experimentales	59
6.1	Plataforma robótica : Turtlebot	59
6.2	Generación de Mapas	60
6.3	Visualización de datos	62
6.4	Programa Autónomo	63
6.4.1	PRM	63
6.4.2	Fast Marching Square	72
7	Impacto socio económico y presupuesto	75
7.1	Impacto socio económico	75
7.2	Presupuesto	75
8	Conclusiones y trabajos futuros	79
8.1	Conclusiones	79
8.2	Trabajos futuros	80
	Bibliografía	82
	Anexos	85
A	Algoritmos	I
A.1	Teleoperación	I
A.2	Teclas	V
A.3	Programa autónomo	VII
A.4	FM2	XIII

Índice de figuras

2.1	Robot cuadrípedo, Boston Dynamics	7
2.2	Robot bípedo Aismo de Honda	7
2.3	Dron, robot móvil aéreo	8
2.4	Ruedas estándar, de Castor y Suecas	8
2.5	Esquema de configuración Ackerman	9
2.6	Esquema de configuración diferencial	9
2.7	Esquema de configuración síncrona	10
2.8	Esquema de configuración omnidireccional	10
2.9	<i>ROS</i> Jade Turtle	12
2.10	<i>ROS</i> Indigo Igloo	12
2.11	Ejemplo de publicación y suscripción a nodos	13
2.12	Esquema de comunicación	14
2.13	Simulacion en Gazebo de Turtlebot	15
2.14	Esquema de comunicación	16
3.1	Conexión <i>ROS-Matlab</i>	18
3.2	Esquema de la estructura del mensaje nav_msgs/Odometry	20
3.3	Visualización de los datos del láser	22
3.4	Imagen de la cámara	22
3.5	Esquema de la estructura de la transformada	23
3.6	Esquema de la estructura del mensaje geometry_msgs/Twist	24
3.7	Información de rosbag	25
3.8	<i>Topics</i> de rosbag	26
4.1	Diagrama de teleoperación	28
4.2	Diagrama de Wanderer	28
4.3	Diagrama de Wanderer	30
4.4	Funcionamiento de la multiplicación sobre partículas	31
4.5	Esquema de etapas de un filtro de partículas	32
4.6	Comparación mapa continuo y simplificación	33
4.7	Descomposición en celdas	34
4.8	Mapa de entorno visualizado con BinaryOccupancyGrid	34
4.9	Mapa insuflado	36
4.10	Esquema algoritmo PRM	38
4.11	Aplicación de PRM	38
4.12	Aplicación de PRM	41
4.13	Curvatura con PurePursuit	42
4.14	Cuerda corta con PurePursuit	42

4.15 Cuerda corta con PurePursuit	42
4.16 Esquema de programa autónomo	43
5.1 Ventana de teleoperación	47
6.1 Turtlebot 2	60
6.2 Comparación de generación de mapas a distintas frecuencias	61
6.3 Visualización de cámara y láser	63
6.4 Visualización de cámara y láser	64
6.5 PRM variación de nodos	65
6.6 trayectorias reales PurePUrsuit	66
6.7 Trayectoria real con filtro de partículas	67
6.8 Convergencia de filtro de partículas (1)	67
6.9 Convergencia de filtro de partículas (2)	68
6.10 Convergencia de filtro de partículas (3)	68
6.11 Inicio de la trayectoria equivocado	69
6.12 Inicio de la trayectoria equivocado	70
6.13 Trayectoria con nuevo obstáculo	71
6.14 Cambio de trayectoria	71
6.15 Trayectoria FM2 (1)	72
6.16 Tiempo de ejecución FM2 (1)	72
6.17 Trayectoria FM2 (2)	73
6.18 Tiempo de ejecución FM2 (2)	73
6.19 Trayectoria FM2 (3)	74
7.1 Diagrama de Gantt	76

Índice de Tablas

6.1 Tiempo de mapeo	62
6.2 Tiempo de ejecución	62
6.3 Tiempo de calculo de PRM	64
7.1 Fases de trabajo	76
7.2 Presupuesto de mercancía	77
7.3 Presupuesto de total	77



Resumen

En este trabajo se exploran las posibilidades de que aporta Matlab al diseño, ejecución y visualización de programas de navegación autónomos en robots móviles tanto en ambiente simulados como sobre plataformas reales, a través de su nueva *toolbox* de robótica. *Robotics System Toolbox* se consolida como una herramienta de integración sólida entre Matlab y robots que operan bajo entorno *ROS*.

Se explorarán todas las herramientas disponibles en la *toolbox* que permiten la consecución de la conexión entre las dos plataformas además de la generación de algoritmos de localización, planificación, mapeo y navegación autónoma.



Abstract

In this paper the possibilities that Matlab provides to design, implementation and monitoring of programs of autonomous navigation robot mobile robots on both simulated and real platforms through its new *toolbox* robotic environment will be explored. *Robotics System Toolbox* has established itself as a solid tool of integration between Matlab and robot operating under *ROS* environment.

It will be studied the tools available in this toolbox that allow achieving the connection between the two platforms, in addition to the generation of algorithms of location, path planning, mapping and autonomous navigation.



Capítulo 1

Introducción

Los robots móviles nacen de la necesidad de sistemas robóticos más flexibles, que no estén anclados a una posición, sino que dispongan de la capacidad de moverse por el entorno. Esto ha provocado que sean considerados una plataforma ideal de investigación en procesos de control y percepción.

El principal objetivo de estos sistemas es su desplazamiento sin intermediación de medios externos, utilizando exclusivamente la información proporcionada por los sensores, precisando mecanismos avanzados de percepción y razonamiento.

La consecución de tareas de inspección de áreas peligrosas, recogida de muestras y transporte de materiales, son las típicamente realizada por esos sistemas. La dificultad de realización de estas va a depender directamente de las características del entorno y del propio robot.

Uno de los principales problemas que se debe de superar para la realización de un sistema robótico autónomo es la monitorización continua del robot. Esta engloba la determinación de la posición y orientación durante el movimiento en el entorno de trabajo.

Las diferentes soluciones propuestas al problema de la localización de robots móviles se clasifican en sistemas de posicionamiento incrementales y absolutos.

Los primeros utilizan sistemas como los odométricos, no precisan información externa del movimiento del vehículo. En cambio los segundos se basan en procesos de localización mediante reconocimiento de marcas artificiales. La mayoría de los robots móviles combinan ambos métodos para conocer su posición respecto al entorno.

1.1. Objetivos

Hasta el momento la mejor y casi única forma de desarrollar código en ámbito de robótica es a través de *ROS*, sistema operativo diseñado para trabajar en Linux. Esto impone y limita el trabajo en esta plataforma, convirtiéndose para una persona que nunca haya trabajado con él, una dificultad añadida.

Con este trabajo se pretende probar una opción que permitiría trabajar en Windows, sistema operativo más difundido en mundo.

Se pretende realizar las mismas acciones que se realizan en un ambiente *ROS* a través de la *toolbox* de Matlab *Robotics System Toolbox*, que permite aprovechar la estructura de mensajes de *ROS* en un entorno de Windows sin ser necesario tener conocimientos de Linux.

El objetivo principal es estudiar la viabilidad y alcance de esta herramienta, además de su utilidad. Se utilizarán diversos algoritmos sencillos de teleoperación o navegación ya utilizado con *ROS* para comprobar la eficiencia de comunicación aprovechando las herramientas de procesamiento de datos que brinda Matlab.

Por lo tanto los objetivos concretos de este trabajo son:

1. Estudio las características y posibilidades de ROS, potente entorno de desarrollo de aplicaciones robóticas.
2. Estudio de Matlab como herramienta de desarrollo de algoritmos de navegación a través de la toolbox de robótica.
3. Comprobar viabilidad de la toolbox de integración de Matlab con ROS.
4. Prueba de algoritmos a través de la toolbox en el espacio de simulación Gazebo.
5. Estudiar la integración Matlab ROS probando algoritmos de navegación de robots en una plataforma real.

1.2. Estructura del trabajo

El contenido de este trabajo se encuentra distribuido en ocho capítulos. En el Capítulo 2 se inicia con una introducción de conceptos de robótica móvil y entorno de programación ROS.

El Capítulo 3 continente toda las información de la toolbox de Matlab utilizada en el trabajo, *Robotics System Toolbox*. Explica las estructuras de comunicación con el robot que serán posteriormente utilizadas por los algoritmos.

Es en el Capítulo 4, donde se realiza de forma teórica la explicación y caracterización de los diversos algoritmos que se utilizarán. Para que en el siguiente

capítulo se desarrolle la implementación de estos en Matlab.

Los datos experimentales recogidos durante la realización del trabajo se plasman en Capítulo 6. Posteriormente en el Capítulo 7 se realiza un pequeño análisis socio económico del trabajo. Para finalizar en el último capítulo se encuentras las conclusiones y los trabajos futuros.

Además el trabajo dispone de un Anexo A que recoge el código de los distintos algoritmos utilizados.



Capítulo 2

Conceptos básicos

En este capítulos se tratará el estado del arte de la robótica móvil y las distintas plataforma de programación más convencionales.

2.1. Robot móvil

Un robot móvil, tal como indica su nombre, es capaz de moverse por su entorno, en contraposición con otros robots, como los industriales, que generalmente consisten en un brazo articulado y una herramienta fija.

Esta capacidad les confiera la posibilidad de explorar ambientes inaccesibles al ser humano debido a la lejanía o peligrosidad y realizar tareas desagradables o laboriosas [15]. La robótica móvil es un campo en el cual intervienen distintas ramas de la ingeniería desde mecánica, eléctrica o mecatrónica.

De acuerdo con el grado de autonomía, los robots se clasifican en teleoperados, con funcionamiento repetitivo y autónomos o inteligentes.

Los **robots teleoperados** agrupan a todos aquellos en los que las tareas de percepción del entorno, planificación y manipulación son realizadas por un ser humano. El operador realiza el control de alto nivel a tiempo real. Las mayores limitaciones radican en la capacidad del operador de procesamiento y precisión, además del retraso transmisión de información de la pareja robot-operador.

Los **robots de funcionamiento repetitivo** constituyen la mayor parte de los que se emplean en cadenas de producción industrial. Se caracterizan por una alta repetibilidad y precisión. Realizan trabajos invariantes y poseen una limitada percepción del entorno.

Por último, los **robots autónomos**, son los más evolucionados desde el punto de vista de procesamiento de información. Así, la robótica móvil autónoma es la parte de la robótica que se centra en el desarrollo de robots con capacidad de movimientos sin necesidad de la intervención de la mano humana, basándose en hacer robots que sean capaces de tomar sus propias decisiones para la realización de una o varias tareas. Dentro de este último grupo se encontraría la plataforma

robótica utilizada en este trabajo [7].

Estos disponen de los elementos sensoriales y actuadores necesarios para modelar el entorno en el que se encuentra, permitiendo con diversos algoritmos planificar y actuar para alcanzar un objetivo, llevando a la consecución del movimiento autónomo por el medio. De esta forma pueden trabajar en entornos poco estructurados y dinámicos [1].

En este desplazamiento autónomo se tiene en cuenta cinco bloques principales:

Percepción Involucra a los distintos sensores del robot, con ellos el robot adquiere conocimiento del entorno. En esta etapa se interpretan las medidas de los sensores y se extrae información que será utilizada en el resto de las etapas.

Localización Es uno de los campos que más dificultades acarrea, el conocimiento de la posición exacta en la que el robot se encuentra dentro del entorno en cada momento. La forma más sencilla de realizar esta tarea es mediante la odometría, pero conlleva muchos errores de inexactitud.

Mapeo Consiste en la representación del medio en el que se encuentra el robot. No siempre se dispondrá del mapa del arquitecto o mapa a priori del entorno, por lo que el robot se construirá uno mediante la información de los sensores. Este tipo de tareas se denominan SLAM (Simultaneous Localization and Mapping), que consisten como su nombre indica en la localización del robot y mapeo del entorno simultaneo.

Planificador de trayectoria Algoritmo que selecciona una ruta libre de obstáculos hacia un punto deseado, partiendo de la posición en cada momento y la información del entorno.

Navegación Fase que agrupa a todas las anteriores y que se centra en alcanzar un punto final con la información anterior, determinando el estado de los actuadores.

Estos cinco bloques se comentarán de forma más extendida en los próximos capítulos.

2.1.1. Locomoción

Finalmente se debe de tener en cuenta otra clasificación de robot móvil, esta vez no de implementación como la anterior, si no atendiendo a sus capacidades de movimiento o tipo de locomoción.

Existen muchos tipos de robots móviles y su configuración depende del tipo del medio en que se muevan así como las condiciones del entorno. Los campos de aplicación abarcan la prospección, vigilancia, rescate y mantenimiento de estructuras, servicios hasta tareas tan diversas como la exploración espacial.

Se clasifican en cuatro grandes grupos dependiendo del tipo de locomoción: aéreos, acuáticos, con ruedas o con patas. Cada uno adaptado a un entorno específico y con distintas configuraciones de estabilidad, dinámica y maniobrabilidad [1][22].

Robots móviles con patas

La locomoción esta caracterizada por una serie de puntos de contacto discretos entre el robot y el terreno, confiriéndole buenas propiedades para moverse en entornos con obstáculos. Este tipo de configuración posibilita la omnidireccionalidad, pero aumenta el consumo de energía y la complejidad tanto de mecanismos como de algoritmos.

Entre las configuraciones más comunes se encuentran robots de seis, cuatro y dos patas, inspirados respectivamente en insectos, reptiles y humanos. En la Figura 2.1 y 2.2 podemos ver ejemplos de estos robot.



Figura 2.1: Robot cuadrípedo, Boston Dynamics



Figura 2.2: Robot bípedo Aismo de Honda

Robots móviles acuáticos y aéreos

Poseen la capacidad de desenvolverse en medios acuáticos o aéreos. Estos robots están muy enfocados a la exploración y a la toma de datos o mantenimiento de instalaciones en entornos naturales los cuales resulta muy difícil el acceso al ser humano.

Dentro de esta categoría cabe destacar un tipo de robot aéreo que esta teniendo un auge importante, el dron, Figura 2.3



Figura 2.3: Dron, robot móvil aéreo

Robots móviles con ruedas

Su locomoción se posibilita mediante el uso de ruedas, son capaces de alcanzar una gran eficiencia en terrenos duros con una implementación mecánica simple.

A la hora de clasificar los robots móviles con ruedas existe un gran número de variantes fruto de las posibles combinaciones de tipos de ruedas y su disposición en el robot.

Distinguimos cuatro tipos de ruedas, como se observa en la Figura 2.4.

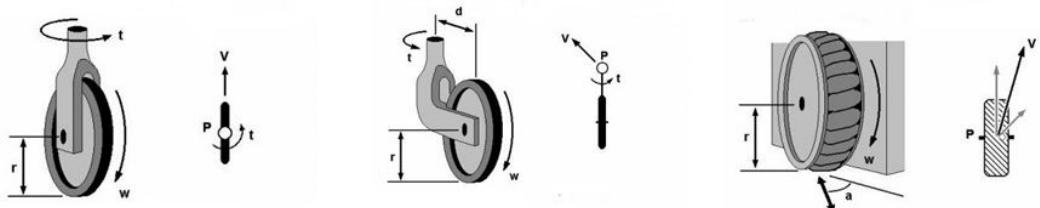


Figura 2.4: Ruedas estándar, de Castor y Suecas

Las dos primeras ruedas en la Figura 2.4 se corresponden a una rueda convencional y rueda de Castor. Se caracterizan por una rodadura pura, la velocidad en el punto de contacto con la superficie es cero. La principal diferencia entre estas dos es que las ruedas de castor posee un offset con respecto al eje de giro.

Por ultimo la rueda Sueca posee menos restricciones de movimiento, la única velocidad que es cero pertenece al punto de contacto con el suelo en la dirección al movimiento, no el resto, lo que posibilita una mayor facilidad de movimiento en el resto de direcciones [1][22].

Dentro de las configuraciones más utilizadas, caben destacar la locomoción síncrona, diferencial, Ackerman, omnidireccional y deslizamiento.

- **Ackerman:** Inspirado en los vehículos convencionales, consta de cuatro ruedas estándar, siendo las dos delanteras las que proporcionan el giro, mientras que las traseras poseen la capacidad motora como observamos en la Figura 2.5. Además se aprecia como el ángulo de giro de la rueda interior es menor que la exterior, esto se realiza para evitar el deslizamiento. Su principal desventaja reside en la limitación de la maniobrabilidad [8].

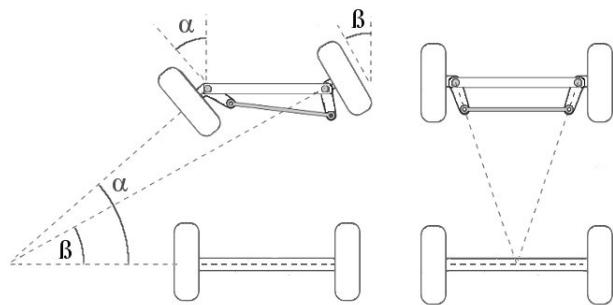


Figura 2.5: Esquema de configuración Ackerman

- **Locomoción diferencial:** Formado por dos ruedas motoras y normalmente una tercera pasiva. En esta configuración el direccionamiento viene dado por la diferencia de velocidades de las ruedas laterales. Como se observa en la Figura 2.6 el giro se consigue como $v_R - v_L$ siendo las velocidades de la rueda derecha e izquierda respectivamente [2].

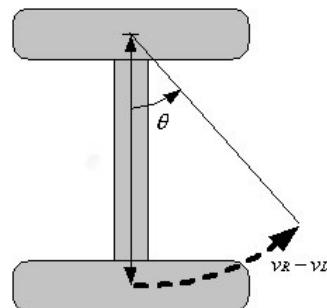


Figura 2.6: Esquema de configuración diferencial

Siendo esta configuración diferencial la que se tomara como referencia en los siguientes capítulos, por ser la que posee el robot del caso de estudio.

- **Locomoción síncrona:** La actuación de las ruedas tiene lugar de forma simultánea. Dispone de dos motores uno de translación y de rotación que transmiten el movimiento a las ruedas mediante coronas de engranajes, Figura 2.7 [1].

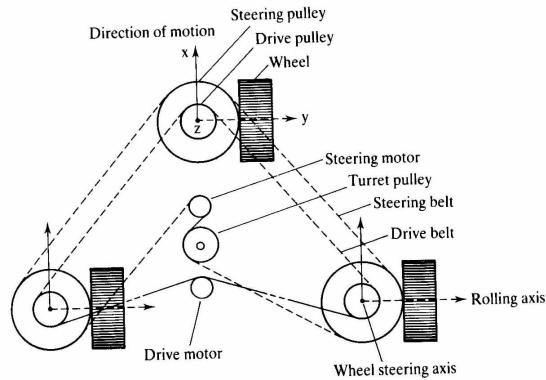


Figura 2.7: Esquema de configuración síncrona

- **Locomoción omnidireccional:** Esta configuración (Figura 2.8) precisa de tres ruedas accionadas por tres motores distintos. En concreto las ruedas utilizadas son las ruedas Suecas por lo que opción proporciona una excelente maniobrabilidad [1].

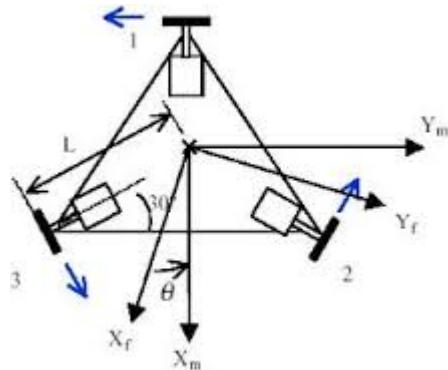


Figura 2.8: Esquema de configuración omnidireccional

- **Locomoción por deslizamiento:** Son vehículos tipo oruga, en ellos la fuerza motora y el direccionamiento de transmite mediante pistas de deslizamiento.

2.1.2. Matlab

MathWork es una de las empresas líderes de desarrollo de software para computo matemático. Dentro de los productos que comercializa podemos hacer una diferencia entre Matlab y Simulink.

Matlab es una gran herramienta para resolver problemas de ingeniería y científicos, posee un lenguaje basado en matrices. Constituye un entorno para el desarrollo de código, análisis de datos, visualización y computación numérica.

Simulink, en cambio, es un entorno gráfico de simulación y desarrollo de modelos. Basado en diagramas de bloques y librerías personalizables.

2.2. ROS

El desarrollo de software para robots se está convirtiendo en un proceso cada vez conlleva un mayor grado de dificultad, debido principalmente al incremento de la complejidad de las tareas que se desea que estos desempeñen.

Cada vez se especializan más los robots para realizar tareas muy concretas, generando una gran variedad de hardware distinto, lo que produce software y programación específica, que dificulta enormemente la reutilización de código. A esto hay que agregar que a menudo se debe que programar desde los drivers de los actuadores, hasta la percepción, o el razonamiento de alto nivel del robot.

De esta forma, para superar estos problemas, nacen distintos entornos de trabajo para facilitar el desarrollo de robots. *The Robot Operating System(ROS)* es el principal entorno de desarrollo en robótica, y ofrece herramientas y librerías para el desarrollo de sistemas robóticos.

Es lo que se conoce como un *framework*, una plataforma de desarrollo de software específico para robótica. Fue creado en 2007, por el instituto de investigación *Willow Garage* bajo licencia BSD (Berkeley Software Distribution) y todos sus programas son de código abierto, por lo que posibilita su uso gratuito tanto para la investigación como en ámbitos comerciales.

Fue ideado bajo la filosofía de que todos los programas sean de uso libre , reutilizables, implementables e integrables con todo software de robótica actual y futuro.

Aporta las soluciones necesarias como la abstracción de hardware, algoritmos de robótica desde control o cinemática a planificación de tareas, además de herramientas de desarrollo y monitorización de sistemas robóticos y administración de paquetes, tanto para robots simulados como reales.

ROS tiene dos partes básicas: la parte del sistema operativo, ros y ros_pkg, y una colección de paquetes aportados por la comunidad de usuarios (organizados en conjuntos llamados stacks) que implementan funcionalidades tales como SLAM, planificación, percepción, simulación, etc [23][16][20].

Debido a que es una plataforma en continuo desarrollo, existen diversas distribuciones o versiones:[28]

- **ROS Box Turtle**, Marzo 2010
- **ROS C Turtle**, Agosto 2010
- **ROS Diamondback**, Marzo 2011
- **ROS Electric Emys**, Agosto 2011
- **ROS Fuerte Turtle**, Abril 2012

- **ROS Groovy Galapagos**, Diciembre 2012
- **ROS Hydro Medusa**, Septiembre 2013
- **ROS Indigo Igloo**, Julio 2014 (ver Figura 2.9)
- **ROS Jade Turtle**, Mayo 2015 (ver Figura 2.10)

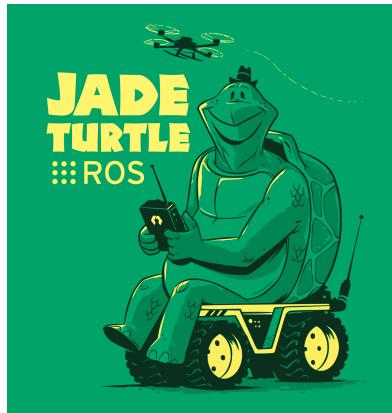


Figura 2.9: *ROS Jade Turtle*



Figura 2.10: *ROS Indigo Igloo*

2.2.1. Introducción a entorno *ROS*

Debido a que la toolbox se comunica con el robot, que trabaja bajo *ROS*, se debe tener conocimiento de la estructuras datos y la forma de enviarlos.

Se describe la estructura de entorno como arquitectura de grafos donde el procesamiento toma lugar en los nodos que pueden recibir, mandar y multiplexar mensajes de sensores, control, estados, planificaciones y actuadores [16] [20].

Esta estructura se modula en los siguientes elementos:

- **Nodos:** puede ser un módulo o un proceso individual en donde se realizan los cálculos o cómputos.

Un sistema robótico normalmente engloba un gran número de nodos pasando por los que controla los sensores y actuadores, a específicos de cinemática o planificadores.

Los nodos se comunican entre sí utilizando *topics*, *services* y *parameter server*. Estos pueden ser escritos en *C++* o *python* utilizando su respectiva librería *roscpp* y *rospy* [23].

- **Topics:** son los buses por los cuales los nodos intercambian mensajes de forma asíncrona. Los nodos que quieren conocer unos datos se suscriben al *topics* concreto, y los que generan datos publican en el *topic* específico.

Se pueden encontrar múltiples publicaciones para un solo *topic* y así mismo un nodo puede publicar o suscribirse a diversos *topics* a la vez. En la Figura 2.11 se observa un esquema genérico de relación *nodo-topic*.

Los *topics* son anónimos, es decir los nodos publican o se suscriben a ellos, pero no conocen el nodo concreto que realiza la acción, lo que permite el desacople de la producción del consumo de la información [20].

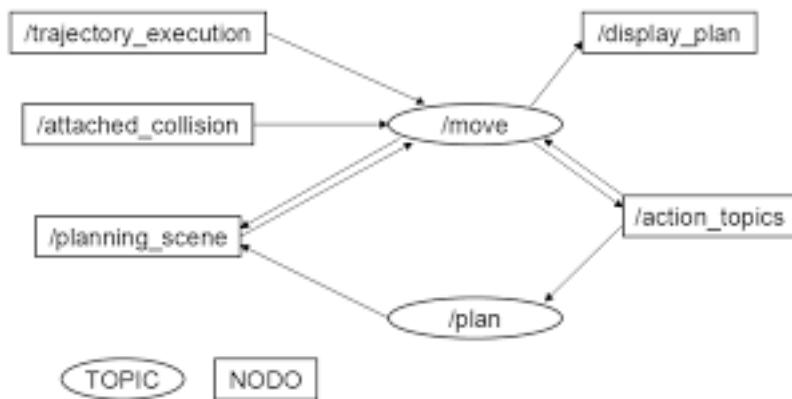


Figura 2.11: Ejemplo de publicación y suscripción a nodos

- **Services:** la forma anteriormente mencionada de publicar-suscribir nodos permite una gran flexibilidad para transmitir información, pero cuando se requiere un intercambio de mensajes síncrono se utilizan los *services*.

Trabajan con parejas de nodo, unos que envía un mensaje de pregunta y otros que recibe la respuesta cuando esta tiene lugar.

Un nodo puede proveer de un servicio, donde el resto de nodos clientes pueden utilizarlos mediante el envío de un mensaje de petición y la posterior espera de la información [23].

- **Mensajes:** los nodos se comunican entre sí mediante la publicación de mensajes que transmiten a través *topics* o *services*. Un mensaje es una estructura de diferentes tipos de datos (enteros, decimales, booleanos, etc), equivalente a una estructura en *C++*. También soportan vectores y estructuras de datos.

Los *topics* están fuertemente restringidos por el tipo de mensaje que se utiliza al publicar, debido a que los nodos solo pueden recibir mensajes con una estructura idéntica a la suya, así no se establecerá un subscriptor a no ser que el tipo de mensaje concuerde con el nodo [23].

- **Master:** es un caso especial de nodo, también llamado *roscore*, que funciona de núcleo del sistema . Este provee el nombre y registro del resto de nodos del sistema.

Actúa de servidor central de la arquitectura y permite que los nodos se localicen entre ellos para que puedan comunicarse mediante *topics* y *services*.

Roscore proporciona tres funciones fundamentales:

- El *Master*, como plataforma de comunicación para nodos.
- *Parameter server* o servidor de parámetros.

- Registro de salida al que los nodos envían información denominado *rosout*

Cada *nodo* se dirige primero al *Master* para crear y buscar el *topic* al que quieren publicar o suscribirse, Figura 2.12.

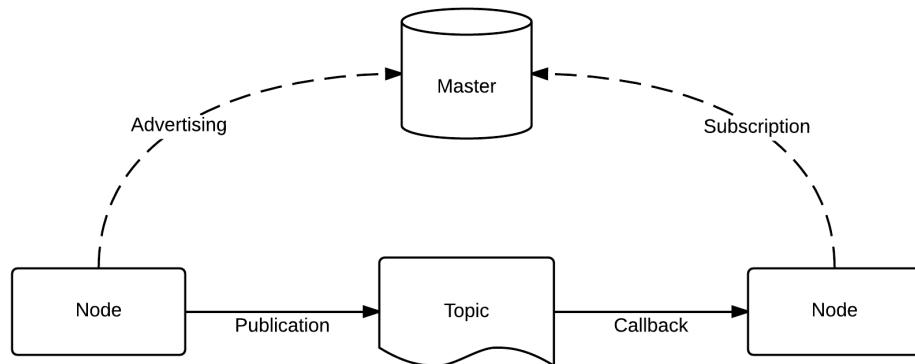


Figura 2.12: Esquema de comunicación

- **Parameter server:** Los nodos pueden utilizar este servidor para guardar u obtener parámetros, permitiendo a la vez al usuario ver y modificar estos parámetros por terminal, ambas operaciones en tiempo de ejecución.

2.2.2. Stage

Stage es un simulador 2-D capaz de manejar múltiples robots y sensores como láseres, codificadores o sonar. Es capaz de simular una gran número de robots moviéndose y recibiendo información del entorno a un coste computacional bajo, ya que utiliza modelos simples de robots [23].

2.2.3. Gazebo

Gazebo es un simulador 3D cinemático, dinámico y multirobot, más sofisticado que Stage, que permite realizar simulaciones de robots articulados en entornos complejos, interiores o exteriores, realistas y tridimensionales.

Desarrollado, en parte por *Willow Garage*, pertenece al grupo de *OpenSource* (software libre), puede ser reconfigurado, ampliado y modificado para crear escenarios de simulación deseados.

Los robots pueden interactuar con el mundo y viceversa, les afecta la gravedad y puede colisionar con obstáculos del entorno, consiguiendo así una simulación realista.

Además, permite desarrollar y simular modelos de robots propios (URDF) e incluso cargarlos en tiempo de ejecución. También tiene la posibilidad de crear

escenarios de simulación.

Contiene diversos plugins para añadir sensores al modelo del robot y simularlos, como sensores de odometría, de fuerza, de contacto, láser y cámaras.

Es totalmente compatible con *ROS*, pudiendo ejecutar Gazebo para enviar y recibir información de los robots simulados. Por lo tanto permite testear aplicaciones en el robot sin depender del robot físico, ahorrando así tiempo y dinero [6][28].

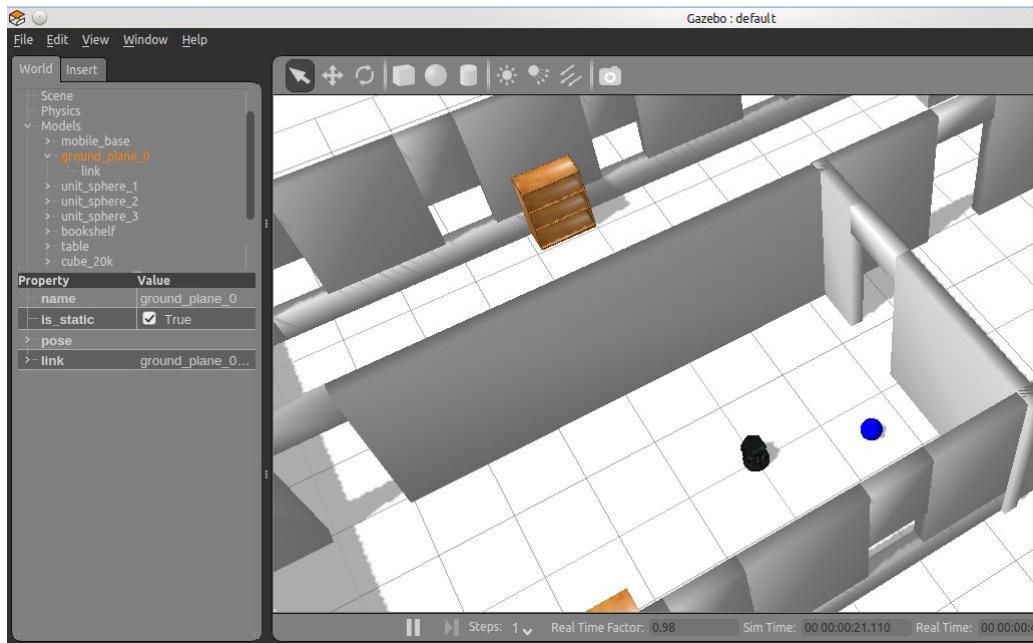


Figura 2.13: Simulacion en Gazebo de Turtlebot

2.2.4. URDF

URDF es el paquete central de *robot_model*, un conjunto de paquetes para el modelado cinemático y dinámico de robots, además de herramientas para la validación y conversión en distintos formatos.

Permite especificar el modelo del robot en formato XML a través de un intérprete Unified Robot Description Format, URDF. Es compatible tanto con el visualizador rviz y Gazebo.

URDF usa un formato de descripción que consta de una serie de etiquetas que permiten especificar los parámetros de cada eslabón articulación del robot [23].

2.2.5. Rviz

Es un programa de visualización 3D de datos provenientes de sensores o estados de *ROS*. Permite combinar en una sola vista modelos de robots, datos de sensores como puedan ser la cámara y láser, Figura 2.14.

Con el se consigue que la plataforma robótica pueda ser representada en 3D, respondiendo a lo que ocurre en el mundo real en tiempo de ejecución [23].

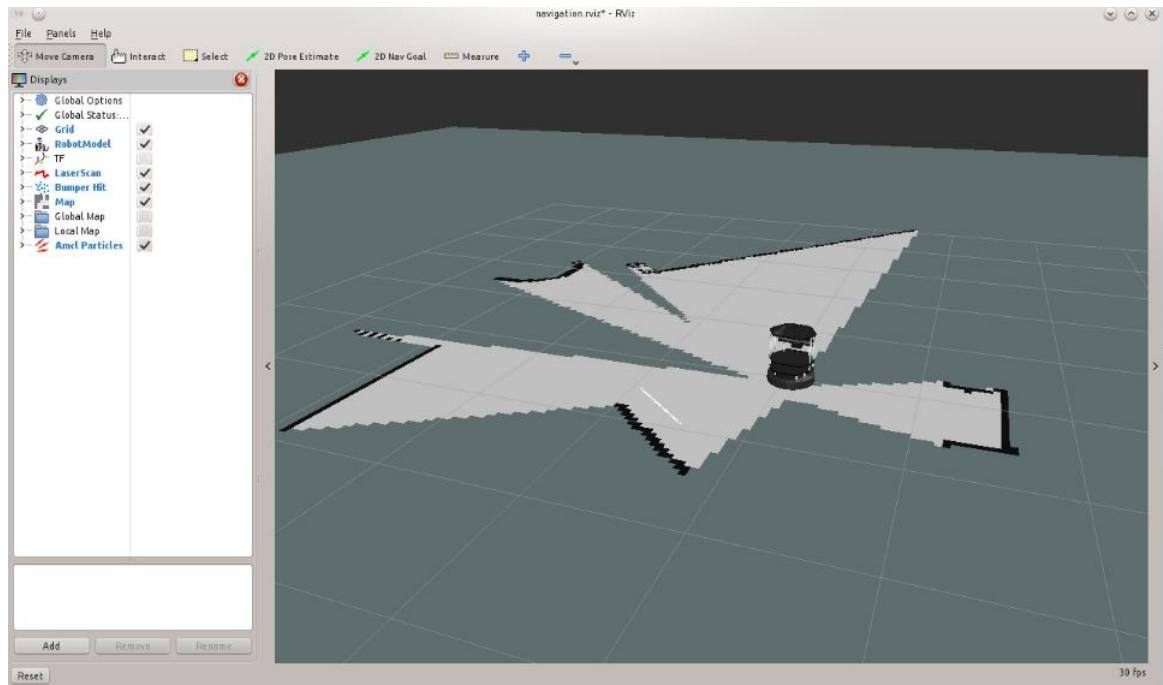


Figura 2.14: Esquema de comunicación

Capítulo 3

Conexión Matlab ROS

Robotics System Toolbox es una *toolbox* que se incorpora en Matlab a partir de la versión R2015A.

Matlab es una herramienta muy importante para diseñar prototipo y simular el control de sistemas robóticos, pero hasta el momento la única forma de conectar un robot bajo ROS con otras plataformas de control y análisis era mediante puentes, en el caso de Matlab se utilizan puente basado en Java [3].

La *toolbox* posibilita la integración directa entre Matlab, Simulink y *ROS* que permite escribir, compilar y ejecutar código en robots que operen bajo *ROS* y a través de simuladores como Gazebo [17].

3.1. Configuración de la conexión

El primer paso para trabajar con la toolbox es establecer una conexión con el robot. *ROS* permite la comunicación entre distintos dispositivos, todos trabajan bajo un mismo *Master*.

En este caso Matlab actúa como un nodo más dependiente del *Master*, que se lanzaría en el ordenador del robot. Para que esto sea posible todos los dispositivos deben de estar conectados a la misma red.

Por lo tanto para realizar la conexión se debe conocer la dirección IP tanto del robot como del ordenador donde se esta ejecutando Matlab, Figura 3.1 [14].

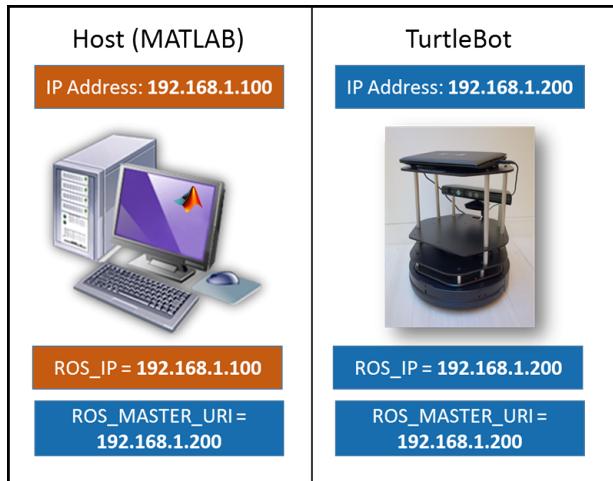


Figura 3.1: Conexión *ROS-Matlab*

Posteriormente se registrar esta información en Matlab de la siguiente forma:

```

1 % Configuracion de la conexion
2
3 setenv('ROS_MASTER_URI','http://192.168.1.200:11311');
4 setenv('ROS_IP','192.168.1.100');
5 setenv('ROS_HOSTNAME','192.168.1.100');
```

Así se esta enviado al entorno de Matlab las tres direcciones que posibilitan la conexión.

ROS_MASTER_URI La dirección del nodo principal, el *Master*.

ROS_IP y **ROS_HOSTNAME** Especifican la dirección del nodo global de Matlab, donde publica y subscribe.

A continuación se lanzaría la siguiente orden para inicializar el entorno de trabajo.

```
1 rosinit;
```

Para comprobar que la conexión se ha hecho de forma satisfactoria se puede ejecutar el comando:

```
1 rostopic list;
```

Este muestra todos los *topics* bajo el *Master*. El motivo de realizar esta comprobación es esencialmente que en ocasiones algunos *topics* como los relacionados con la cámara tienen problemas para conectarse adecuadamente.

3.2. Intercambio de información

Una vez establecida la conexión se puede recibir y enviar datos. Las siguientes estructuras se repetirán en la mayoría de algoritmos que sean probado durante este trabajo.

3.2.1. Recibir mensajes

Esencialmente se tratarán tres *topics* /odom , /scan y /camera/rgb/image_color/compressed que se corresponden a la odometría, el láser y la cámara respectivamente[18].

Mensaje odometría

```

1 %Establecer subscripcion a /odom
2
3 odom = rossubscriber('/odom');
4 odomdata = receive(odom,3);
5 pose = odomdata.Pose.Pose;
6 x = pose.Position.X;
7 y = pose.Position.Y;
8 z = pose.Position.Z;
9 quat = pose.Orientation;
10 angles = quat2eul([quat.W quat.X quat.Y quat.Z]);
11 theta = rad2deg(angles(1))

```

Tal como se muestra en las líneas de código, en primer lugar se subscribe al *topic* /odom mediante *rossubscriber*, y posteriormente se establece la variable donde se va a almacenar de forma temporal la información recibida.

Con el comando *receive* se actualizan los datos de odometría cada vez que se recibe nueva información al *topic*.

Para saber como acceder a la información concreta del *topic*, es preciso conocer el tipo de mensaje con el que se publican datos. Esto lo realizamos a través de:

```
1 rostopic info /odom;
```

Comando que devuelve la información de dicho *topic*

```

1 Type: nav_msgs/Odometry
2
3 Publishers:
4 * /gazebo (http://192.168.1.100:54690/)
5
6 Subscribers:
7 * /matlab_global_node_16200 (http://192.168.1.200:55004/)

```

En la respuesta mostrada se especifica el tipo de mensaje del *topic* y cual es el nodo que publica además del subscriptor a este.

A continuación ejecutaría el comando siguiente para conocer la estructura del mensaje .

```
1 rosmsg show nav_msgs/Odometry;
```

Obteniendo,

```
1 % This represents an estimate of a position and velocity in
  free space.
2 % The Pose in this message should be specified in the
  coordinate frame given by Header.frame_id.
3 % The Twist in this message should be specified in the
  coordinate frame given by the ChildFrameId
4 std_msgs/Header Header
5 char ChildFrameId
6 geometry_msgs/Pose
7 geometry_msgs/Twist
```

Realizando el paso anterior con *geometry_msgs/Pose* y así sucesivamente, se comprueba que realmente se trata de una estructura anidada con la siguiente disposición, Figura 3.2:

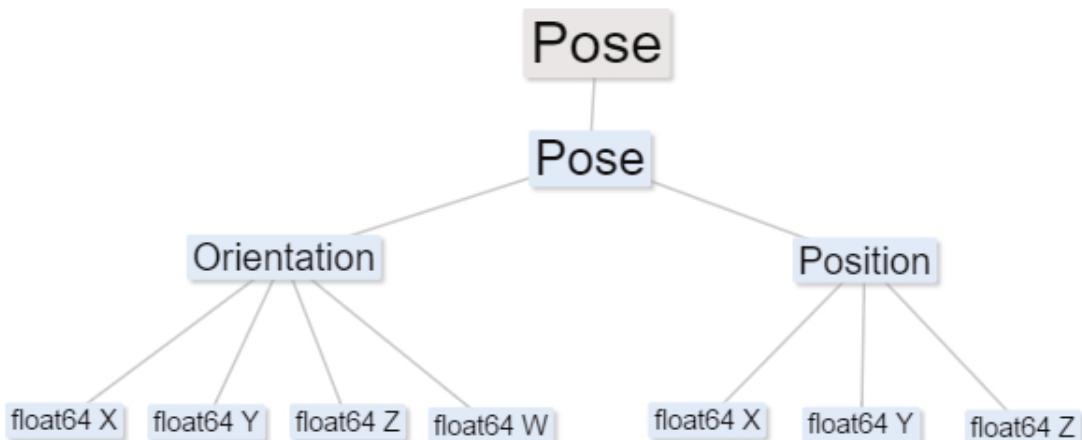


Figura 3.2: Esquema de la estructura del mensaje nav_msgs/Odometry

De esta forma se generan el resto de variables relacionados con la odometría, como la posición o el ángulo de giro.

Debido a que la información de la odometría son datos que se utilizan en repetidas ocasiones dentro de un mismo algoritmo, en las posteriores secciones se utilizará la siguiente función:

```

1 function [pose]=odompose(odomdata)
2
3 pose_odom = odomdata.Pose.Pose;
4 x_odom = pose_odom.Position.X;
5 y_odom = pose_odom.Position.Y;
6 quat = pose_odom.Orientation;
7 angles = quat2eul([quat.W quat.X quat.Y quat.Z]);
8 theta = angles(1);
9 pose=[x_odom y_odom theta];
10 end

```

Esta grua las variable anteriores en una estructura de tipo fila de tres componente: posición en x , posición en y y angulo en radianes.

Mensaje láser

En el caso del láser se realizaría la suscripción al *topic* `/scan`, y siguiendo un procedimiento homólogo al anterior, consiguiendo saber el tipo de mensaje y su estructura.

Concretamente `sensor_msgs/LaserScan` que posee una disposición lógica mucho más simple, ya que se utiliza toda la información recibida.

```

1 %Establecer subscripcion a /scan
2
3 laser = rossubscriber('/scan')
4 scan = receive(laser,10)
5 data = readCartesian(scan)
6 plot(scan,'MaximumRange',7)

```

En la variable `laser`, se crea la suscripción al `/scan`, y en la variable con el mismo nombre se almacena la información recibida que se actualizará cada vez que haya nuevos datos de la cámara.

En la línea 5 de código se realiza una conversión de esos datos provenientes del láser a un formato de coordenadas cartesianas y posteriormente se visualizan en un `plot`, Figura 3.3.

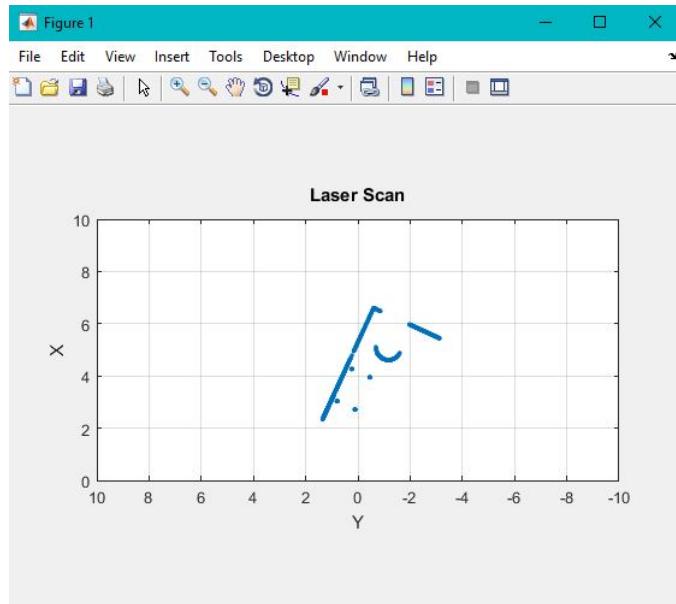


Figura 3.3: Visualización de los datos del láser

Mensaje cámara

Por último la información de la cámara, siendo el nombre del mensaje sensor_msgs/CompressedImage.

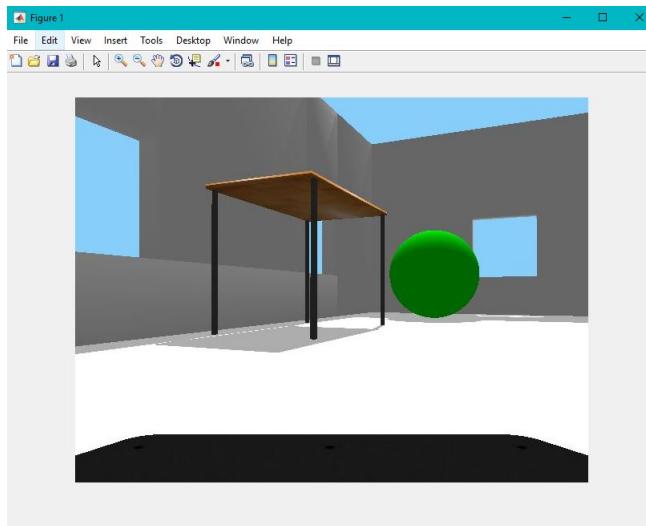


Figura 3.4: Imagen de la cámara

Se realiza suscripción al mensaje y posterior almacenamiento de información en la variable *img*, esta a través del comando `readImage` puede ser mostrada, Figura 3.4.

```

1 %Establecer subscripcion a /camera/rgb/image_raw
2
3 imsub = rossubscriber('/camera/rgb/image_raw');
```

```

4 img = receive(imsub);
5 figure
6 imshow(readImage(img));

```

Mensaje *tf* (transforms frames)

tf es un paquete de *ROS* que permite realizar un seguimiento de múltiples sistemas de coordenadas a lo largo del tiempo. Mantiene la relación de los diversos sistemas en un estructura en forma de árbol, además permite al usuario transformar puntos vectores o estructuras entre sistemas de coordenadas distintos. En esta sección se hace un mención a esta estructura debido a su importancia en diversos algoritmos.

```

1 tftree = rostf;
2 tftree.AvailableFrames;

```

Como se observa en la imagen los distintos sistemas que están disponibles son:

- /base_link
- /camera_depth_frame

La función *waitForTransform* espera la llegada de la transformada entre la '/*base_link*' y '/*camera_depth_frame*', que corresponden a la base del robot y la cámara. Esta llamada se bloquea hasta que la transformación que toma los datos es válida y disponible en el árbol de transformación, para posteriormente almacenar la información en la variable *cameraTransform*.

```

1 waitForTransform(tftree,'/base_link','/camera_depth_frame');
2 cameraTransform = getTransform(tftree,'/base_link','/
3   camera_depth_frame');
4 cameraRotation=cameraTransform.Transform.Rotation;
5 cameraTranslation=cameraTransform.Transform.Translation;

```

Este *topic* dispone de la siguiente estructura:

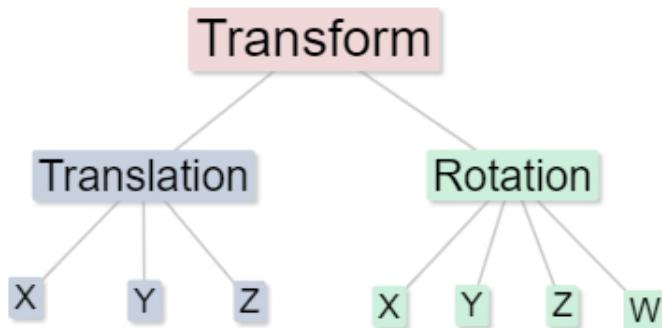


Figura 3.5: Esquema de la estructura de la transformada

Así, como indica la Figura 3.5, para el caso de sistemas de coordenadas de la cámara y láser del robot se podría obtener información de la distancia entre uno y otro accediendo a *Translation* tanto en *x*, *y* o *z*, especificando en la línea 3 del código que dirección queremos estudiar. Además de conocer la rotación relativa entre ellos mediante *Rotation* con el mismo procedimiento.

3.2.2. Enviar mensajes

A la hora de enviar comandos al robot la única parte móvil que dispone son las ruedas, por lo que se impondrá la velocidad angular y lineal deseada. Mediante el establecimiento de una publicación de parte del nodo Matlab al *topic /mobile_base/commands/velocity* [19].

Mensaje velocidad

Por lo tanto el primer paso del código es crear la publicación, esto se realiza con *rospublisher* especificando el nombre del *topic*, además se crea el mensaje acorde con el *topic* a través de *rosmessage*.

```

1 %Establecer la publicacion a /mobile_base/commands/velocity
2
3 vel=rospublisher('/mobile_base/commands/velocity');
4 velmsg = rosmessage(vel);
5 velmsg.Linear.X = 0.5;
6 velmsg.Angular.Z = 0.5;
7 send(vel,velmsg);

```

Igual que el *topic /scan* posee una estructura específica, Figura 3.6:

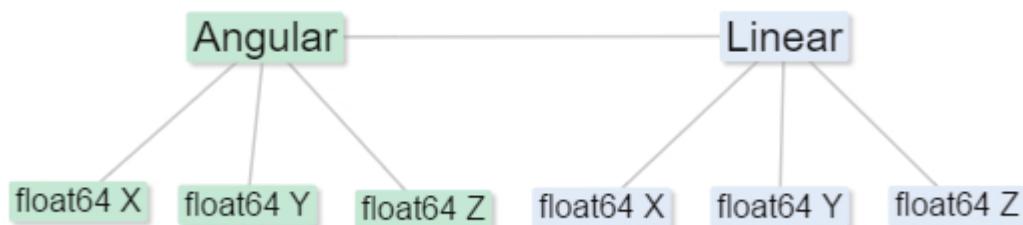


Figura 3.6: Esquema de la estructura del mensaje *geometry_msgs/Twist*

Por último con la orden *send* se envía el mensaje al *topic* con los valores deseados de velocidades.

3.3. Procesado de datos

En esta sección se mencionará una herramienta que dispone la *toolbox Robotics System Toolbox* para el procesado de datos de sistemas robóticos.

Una de las formas de procesado de información, es la anteriormente mencionada generar suscripciones a los *topics* deseados y crear estructuras de almacenaje en Matlab. La *toolbox* incorpora una segunda forma para realizar la misma tarea mediante la utilización de archivos *rosbag*. Son un formato para guardar y reproducir mensajes de *ROS* de forma offline.

Rosbag es un mecanismo importante para el almacenamiento de datos, como lecturas de sensores, que pueden ser difícilmente adquiridas, pero que son necesarias para el desarrollo y comprobación de algoritmos.

Con el siguiente comando se lee y trascibe todas la información de la *rosbag* a Matlab.

```
1 %Procesado de rosbag
2 bag = rosbag(filePath);
```

A continuación se obtiene información de ella, como puede ser el tiempo de grabado, el número de mensajes o *topics*, Figura 3.7

```
1 bagsel = select(bag)
2 bagsel.AvailableTopics
```



```
bagsel =
BagSelection with properties:
```

```

    FilePath: 'C:\Users\Marina\Desktop\rosbag\odom_scan.bag'
    StartTime: 353.3300
    EndTime: 389.4800
    NumMessages: 3848
    AvailableTopics: [2x3 table]
    MessageList: [3848x4 table]
```

Figura 3.7: Información de rosbag

En el caso concreto de la *rosbag* que se ha utilizado como ejemplo, con el segundo comando comprobamos que el archivo contiene grabados de los *topics* /odom /scan , Figura 3.8

```
ans =
```

	NumMessages	MessageType	MessageDefinition
/odom	3611	nav_msgs/Odometry	[1x2918 char]
/scan	237	sensor_msgs/LaserScan	[1x2122 char]

Figura 3.8: Topics de rosbag

3.4. Bucle de control

Una de la herramientas importantes para cualquier control de sistemas robóticos es la frecuencia con la que se cierra el bucle de control.

Utilizando la *Robotics System Toolbox* el bucle de control se cierra en Matlab, por lo que si no se especifica una frecuencia fija de trabajo, el código se ejecutara a la máxima capacidad de Matlab.

La *toolbox* dispone de una función para controlar este factor.

```
1 desireRate=20;
2 rate=robotics.Rate(desireRate);
3 rate.OverrunAction='drop';
```

El uso de *robotics.Rate* lo que permite es ejecutar bucles a frecuencia fija. En la variable *desireRate* se especifica la frecuencia en Hz, posterior mente creamos el objeto *robotics.Rate*.

Este dispone de dos modos de ejecución 'drop' y 'slip', el primero espera a que se lea todo el código aunque el tiempo sea mayor que el estipulado por la frecuencia, mientras el segundo reinicia el bucle rigiéndose únicamente por la frecuencia [14].

```
1 reset(rate)
2
3 while rate.TotalElapsedTime < 10
4 send(vel,velmsg)
5 waitfor(rate);
6 end
```

Lo que se realiza para inicializar el bucle es un reset del objeto *robotics.Rate*, controlado por *rate.TotalElapsedTime* se ejecuta el código en este caso un simple envío de velocidades , se espera a cumplir con la frecuencia deseada para reiniciar el proceso.

Capítulo 4

Integración de algoritmos

A lo largo este capítulo se describirán los algoritmos con los que se ha trabajado y probado, además de definir conceptos clave para el entendimiento de estos.

4.1. Teleoperación

Existen un gran número de aplicaciones robóticas donde la intervención del ser humano es imprescindible y necesaria. Estos son casos que se caracterizan por entornos no estructurados y dinámicos, en los que la percepción y la planificación automática son muy complejos. En estas situaciones se utiliza la teleoperación.

Estos algoritmos presentan el menor grado de autonomía, el bucle de control está controlado por el operador, es decir, las tareas de percepción del entorno y planificación están realizadas total o parcialmente por este.

El operador puede intervenir ya sea de forma directa, en los actuadores del robot, o de forma de toma de decisiones a la hora de realizar tareas que si están automatizadas.

Debido a que en muchos casos en los que se utilizan este tipos de algoritmos el operador se encuentra físicamente separado del robot, este debe tener opción de visualizar los datos de los sensores del láser y cámara, que le servirán como referencia. Así se puede definir la teleoperación como la extensión de capacidades de percepción y destreza del ser humano en una localización remota [1].

Como se observa en la Figura 4.1 el algoritmo de teleoperación que se ha integrado con la *toolbox*, se restringe a movimientos simples de avance y giros.

Realiza una teleoperación directa, donde mediante teclado se especifican las velocidades linear y angular que son enviadas al robot. De este se utilizan los datos del láser y la cámara que serán procesados y visualizados como referencia para el operador, cerrándose así el bucle de control.

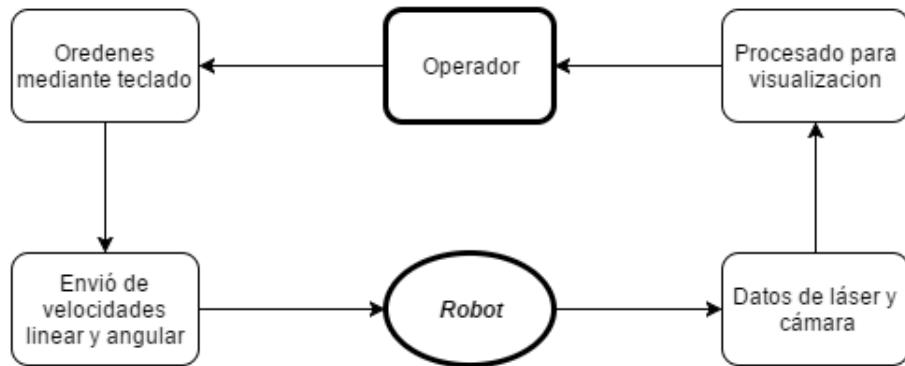


Figura 4.1: Diagrama de teleoperación

4.2. Wanderer

Este algoritmo posee un nivel superior de autonomía al mencionado en la sección 4.1. En el robot es capaz de esquivar objetos u obstáculos.

Durante la ejecución el robot se mueve por el entorno sin tener un objetivo fijo, modificando su trayectoria si los sensores detectan un objeto a una distancia menor a de una longitud límite como se muestra en la Figura 4.2.

Es esta longitud límite es un parámetro fundamental para el funcionamiento. Esta se debe de ajustar de forma que se tenga en cuenta las dimensiones del robot, para que permita el movimiento alrededor de obstáculos. Pero también teniendo en cuenta el entorno en el que se encuentra, ya que si los obstáculos se encuentran a una distancia menor del límite establecido el robot no se moverá.

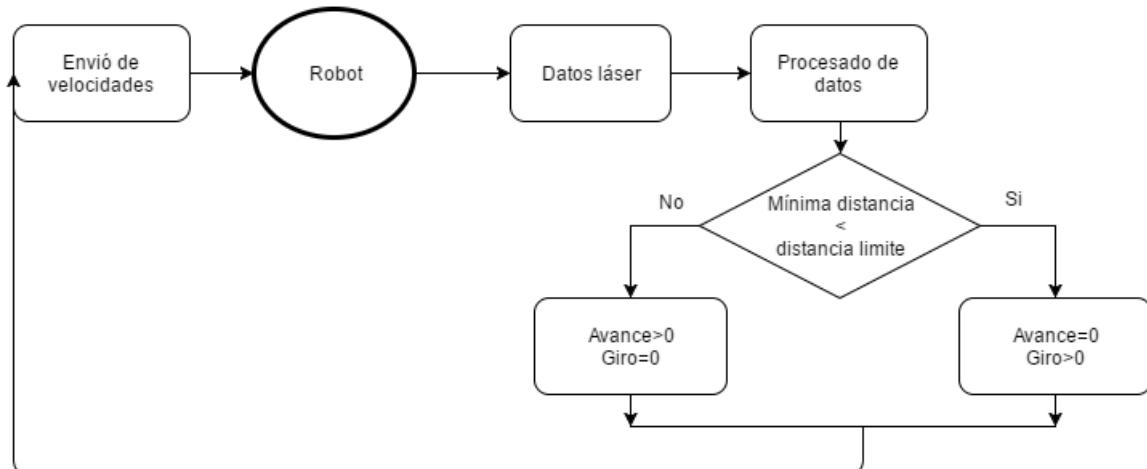


Figura 4.2: Diagrama de Wanderer

Presentan un lógica muy simple, tal como indica su nombre *Wanderer* que se podría traducir como 'aquel que vaga', estos algoritmos se restringen a 'vagar' por el entorno. Su principal utilidad reside en el reconocimiento del entorno y fabricación de mapas.

4.3. Localización

La estimación de la localización del robot forma parte de unos de los bloques esenciales dentro de la navegación, debido a que permite conocer la posición y orientación de este en cada instante.

La odometría es uno de los métodos más utilizados para estimar la posición de un robot. Se basa en ecuaciones sencillas de implementar, las cuales hacen uso del valor de los encoders de las ruedas del robot para traducir las revoluciones de estas a un desplazamiento lineal relativo al suelo a lo largo del tiempo.

Es un método de conocer la localización del robot barato y que permite altas tasas de muestreo, pero que aporta acumulación de errores, los cuales van aumentando proporcionalmente con la distancia recorrida pudiendo llegar a un punto en el que la información proporcionada por ellos no sea acorde con la realidad.

Así se define incertidumbre geométrica a la imprecisión en el conocimiento de la localización del robot respecto a un sistema de referencia dado.

Las principales causas de error debidas a la odometría son las siguientes:.

- Errores sistemáticos

- Distintos diámetros en las ruedas.
- Diferencias entre el diámetro nominal de la rueda y su valor real.
- Desalineamiento de las ruedas.
- Resolución discreta del encoder.

- Errores no sistemáticos

- Desplazamiento en suelos desnivelados .
- Desplazamientos sobre objetos inesperados que se encuentran en el suelo.
- Deslizamientos con la superficie de contacto.
- Fuerzas externas e internas aplicadas sobre las ruedas.
- Pérdida de contacto de la rueda con el suelo.

Cada posición del robot obtenida mediante odometría tiene asociada una elipse de error debido a la incertidumbre, que indica la posible localización del robot, como se muestra en la Figura 4.3. Estas elipses aumentan de tamaño con el tiempo al realizar una trayectoria, siempre y cuando no se disponga de un sistema de estimación absoluta que reduzca el crecimiento de la incertidumbre a cero.

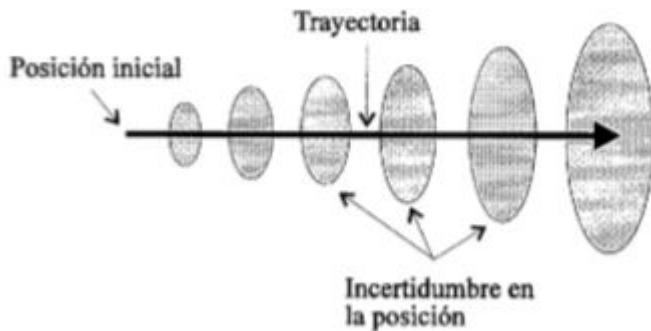


Figura 4.3: Diagrama de Wanderer

Por lo tanto para reducir este error, se debe conocer el modelo de representación de la incertidumbre y un método de integración que permita manejar dicha representación. Para ello, existen diversos tipos de filtros, en la próxima sección se mencionaran los filtro de partículas.

4.3.1. Filtro de partículas

Debido a que la información proveniente de los sensores están afectadas por una incertidumbre sistemática, la única forma de representarla es mediante una probabilidad.

La introducción de técnicas probabilísticas en ámbitos de robótica, sobre todo en localización, posibilitó el desarrollo de programas más fieles a la realidad de percepción. Estas permiten integrar modelos y sensores imperfectos mediante leyes probabilistas [22].

El filtro de partículas , también conocido Método de Monte Carlo, es un método empleado para estimar el estado de un sistema que cambia a lo largo del tiempo.

Consiste en generar una población de partículas que se distribuyen uniformemente en todo el entorno y se trata de que esta nube converja a la posición correcta del robot.

El filtro esta formado por un conjunto de muestras en el espacio que tienen asociadas un peso normalizado entre $[0,1]$. Normalmente se trabaja con un vector que engloba $[x, y, \theta]$, movimiento en la dirección x e y , además de la orientación en el eje z .

El problema de estimación de la localización se restringe a un filtro Bayesiano cuyo objetivo es encontrar la probabilidad de posición del robot. La función de densidad de probabilidad en una aproximación bayesiana engloba toda la información que tenemos sobre un estado x_k , y de ella se estima la posición real [2][5].

Por lo tanto el algoritmo se centra en calcular:

$$p(x_k | Z^k) \quad (4.1)$$

Donde Z^k es la información que obtenemos de los sensores.

El algoritmo consta de cuatro etapas inicialización, actualización, selección y predicción:

Inicialización Se genera una distribución $p(x_0)$, aleatoria. Si no se conoce ninguna estimación de la posición inicial, las partículas estarán distribuidas por todo el espacio de búsqueda.

Actualización Se define una función que sea proporcional a la posición de las partículas que se encuentran sobre el objeto que se quiere localizar.

Sobre el conjunto de partículas se realiza la multiplicación por esta función proporcional. Con esta operación se consigue que los pesos de las partículas sean proporcionales a la función por la que se multiplica, manteniendo la posición de cada una.

Esta transformación se puede entender de forma gráfica en la Figura 4.4

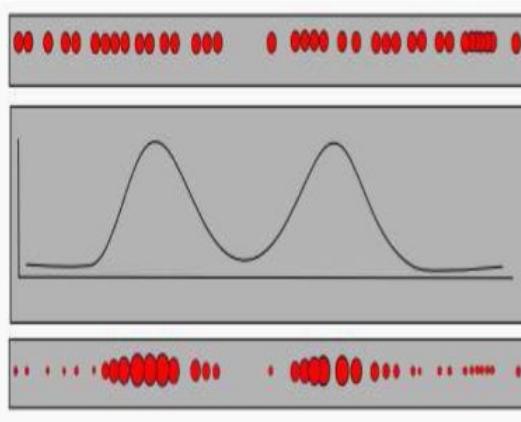


Figura 4.4: Funcionamiento de la multiplicación sobre partículas

Lo que se realiza es una variación de los pesos de cada partícula incorporando la información obtenida de los dispositivos de medida.

Selección En esta etapa las partículas con pesos alto (que aumentan su probabilidad) se replicarán un mayor número de veces comparadas a las partículas con pesos inferiores, las cuales dependiendo de la magnitud del peso podrán ser eliminadas.

Así partiendo del conjunto de partículas obtenido en la etapa de *actualización* se crea uno nuevo con el mismo número de partículas.

Predicción En esta última fase se realiza una predicción el modelo de estados.

Este modelo busca representar el conocimiento que se posee acerca del movimiento. Si se conoce la posición en el instante $t-1$ poseemos información acerca de la posible posición en el instante t

$$x_t^i = f(x_{t-1}^i) + v^i \quad (4.2)$$

Siendo x_t^i cada una de las partículas de la predicción, la función que propaga los estado en el tiempo y v_k^i corresponde a ruido aditivo.

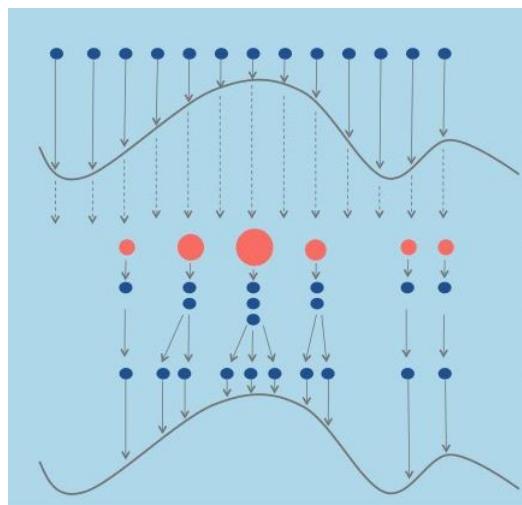


Figura 4.5: Esquema de etapas de un filtro de partículas

En la figura 4.5 se puede ver el esquema que seguiría un filtro de partículas a través de sus cuatro fases, proceso que se repetiría recursivamente exceptuando la inicialización hasta conseguir que la nube converja [24].

4.4. Mapping

Los mapas son las única forma de conocer el entorno a priori, además de ayudar a localizar al robot en el entorno.

El problema de representar el entorno influye directamente en la representación del las posiciones de robot y a las decisiones que se pueden tomar respecto a ellas, así la fidelidad de la posición del robot en el mapa depende intrínsecamente de la fidelidad el mapa.

Debido a esto se debe tener en cuenta tres parámetros para seleccionar un tipo de representación:

- La precisión del mapa debe ser acorde con la precisión con la que el robot debe alcanzar un punto determinado.
- La complejidad del mapa conlleva un mayor esfuerzo computacional.
- La precisión del mapa y el tipo de objetos representados en el deben de coincidir con la precisión y el tipo de datos que aportan los sensores del robot.

Los mapas pueden ser creados por el propio robot, tema que se discutirá en la próxima sección 4.5, o pueden ser los denominados mapa del arquitecto.

La representación continua es un tipo de mapa donde se plasma el entorno de forma exacta. Debido a la gran cantidad de información que estos poseen solo son implementables en representaciones 2D, ya que el esfuerzo computacional para más dimensiones sería inmenso. La mayor ventaja que tienen estos tipos de mapa es la gran precisión a cambio de un mayor coste computacional [22].

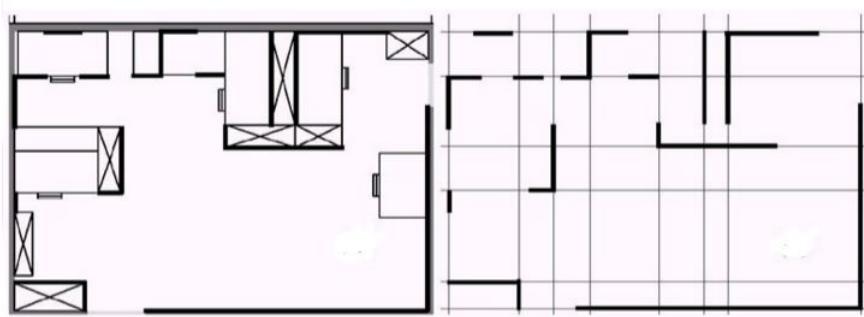


Figura 4.6: Comparación mapa continuo y simplificación

A partir de ellos se puede extraer la representación simplificada de los obstáculos del entorno eliminando los datos de diferenciación de objetos, como se muestra en la Figura 4.6.

Otro tipo de simplificación de mapa es mediante descomposición y abstracción del entorno. Una de estas técnicas consiste en dividir el mapa en casillas pasando de una representación continua a una discreta, Figura 4.7. El principal problema de esta simplificación reside en el tamaño de estas divisiones donde se podría perder información de objetos pequeños.

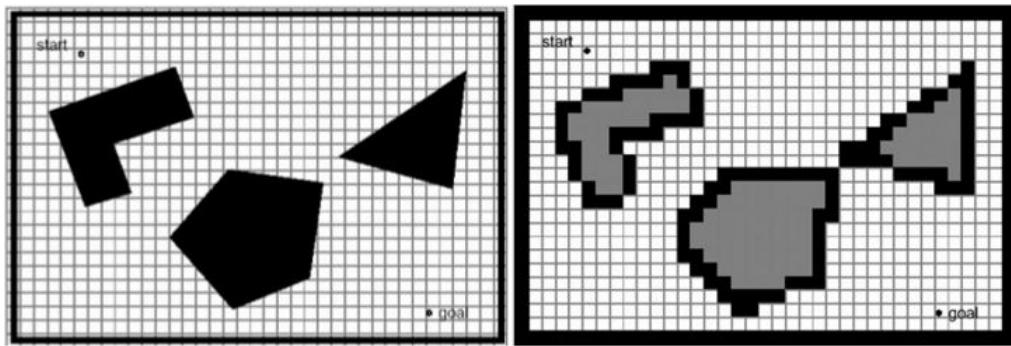


Figura 4.7: Descomposición en celdas

La descomposición en celdas fijas también es conocida como *occupancy grid*. El entorno es dividido en una grilla discreta, donde una celda puede estar ocupada si representa a una parte de un obstáculo o vacía si corresponde a espacio libre. Este tipo de metodología es muy útil cuando el robot construye el mapa del entorno a partir de los datos del láser y posición relativa.

La Figura 4.8 corresponde a un mapa obtenido mediante este método del entorno de simulación Gazebo.

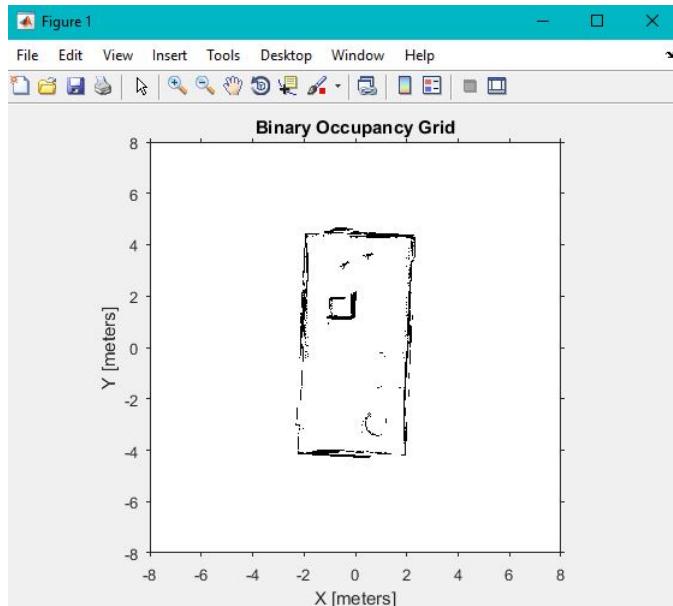


Figura 4.8: Mapa de entorno visualizado con BinaryOccupancyGrid

La principal desventaja *occupancy grid* reside que a la vez que crece el entorno que se desea representar, crece la memoria necesaria para almacenar esa información.

Ademas hay que tener en cuenta otra limitación, que es que se debe considerar a priori el tamaño aproximado del entorno para crear la grilla [22][4].

4.5. SLAM

Debido a que una de las aplicaciones de los robots móviles es la exploración de entornos desconocidos o cambiantes, no siempre se dispondrá de un mapa de arquitecto, en estos casos es el propio robot el que cree el mapa.

SLAM (*Simultaneous Localization And Mapping*), localización y mapeo simultáneos. Es una aplicación que realiza esta tarea y en la que se une tanto el problema de la localización en el entorno y la creación de un mapa que lo represente es mediante [25].

Se parte de la suposición de que la posición inicial es conocida, y partiendo de ese punto se crea el mapa con la ayuda de los sensores, almacenando los datos en un *occupancy grid* [9].

La mayor dificultad realizando SLAM es que tanto la posición como los datos de los objetos del entorno están afectados por ruido y errores [4].

4.6. Planificador de trayectorias

Un planificador de trayectorias se encarga de encontrar una ruta libre de obstáculos desde un punto de origen a una posición objetivo.

Juegan un papel fundamental en aplicaciones con robots móviles. A partir de datos sobre el entorno deben calcular de forma eficiente como llevar a cabo las misiones de forma segura.

Las técnicas para planificar trayectorias se basan en que el robot se encuentra en *configuration space*, este engloba a todo lo que se encuentra en entorno por el que el robot se va mover. De él se identifica el *configuration space obstacle*, que englobaría los obstáculos que puede encontrarse el robot. Partiendo de esta dos premisas se obtiene el espacio libre por el cual el robot se desplazará, como la resta de estos dos espacios [22].

La gran mayoría de los planificadores trabajan bajo dos premisas:

- El robot se puede representar mediante la terna (x, y, θ) , correspondiéndose al movimiento en dirección x e y en el plano y a la velocidad angular sobre si mismo.
- Asumir al robot como un punto en el espacio simplificando la representación del *configuration space* a una en dos dimensiones en los ejes x e y .

De esta forma se simplifica de gran manera la complejidad de un planificador si bien no todos los robots móviles pueden ajustarse a la primera premisa la gran mayoría si, como pueden ser los robots diferenciales.

Además hay que tener una consideración importante a raíz de la segunda simplificación. Al reducir el robot a un punto se realiza un agrandamiento de los obstáculos igual al radio del robot para compensar el cambio, asegurando que ninguna trayectoria calculada por el planificador produzca el choque del robot con objetos del entorno.

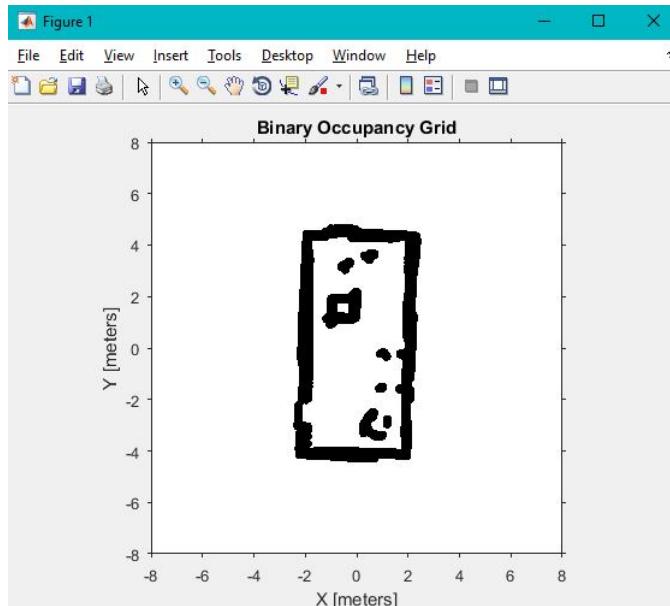


Figura 4.9: Mapa insuflado

La Figura 4.9 muestra el mapa insuflado que corresponde a la Figura 4.8 considerando un radio del robot igual a 0.15 metros.

La representación del entorno como ya se mencionó en la sección 4.4 puede presentarse en diversas formas, el primer paso en cualquier planificador es discretizar y adaptar el mapa dependiendo del algoritmo utilizado.

Atendiendo a este criterio se clasifican los algoritmos planificadores en tres grandes grupos:

Road map En español mapa de rutas identifica las posibles rutas dentro del espacio libre.

Descomposición de celdas Diferencia entre celdas ocupadas de las libres.

Campos potenciales Impone una función matemática sobre el espacio que genera potenciales que atrae el robot hacia el objetivo y lo repele de los obstáculos.

4.6.1. PRM

PRM (*Probabilistic RoadMap*) es un algoritmo planificador de trayectorias que busca la conectividad de distintos puntos por el espacio libre, desde un punto inicial al objetivo final evitando colisiones con los obstáculos del entorno, utilizando métodos aleatorios [12].

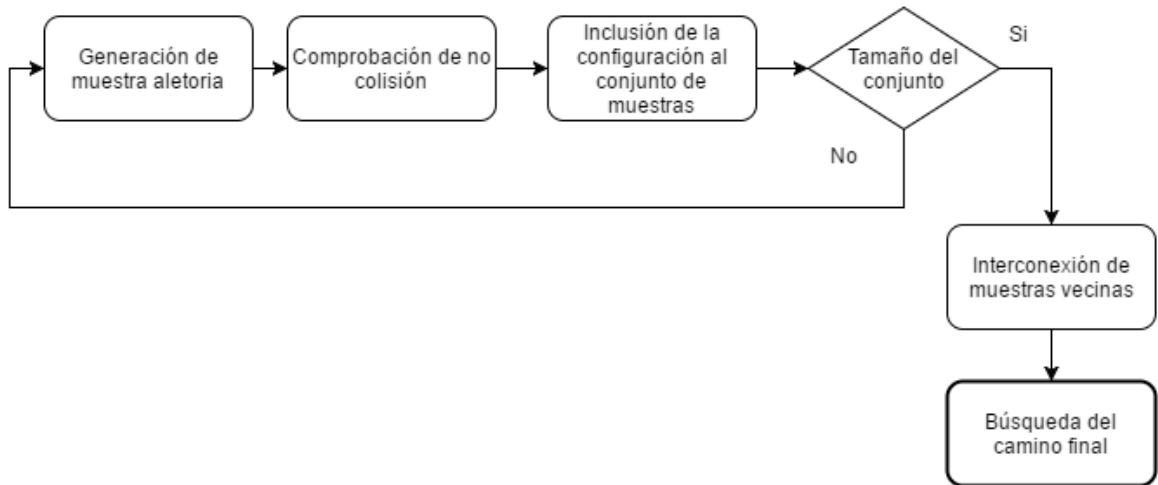
El algoritmo consta de dos etapas:

Learning phase o fase de aprendizaje. En esta fase se genera unas muestras de nodos dentro del *configuration space* usando métodos aleatorios, asignando valores al azar a las coordenadas de la configuración.

Posteriormente se comprueba si el robot colisiona o no en cada una de las configuraciones. Este es un proceso que se repite hasta obtener un conjunto de muestras aptas de la magnitud deseada. Para poder encontrar una solución se debe agregar siempre al conjunto de muestras las posiciones iniciales y objetivo.

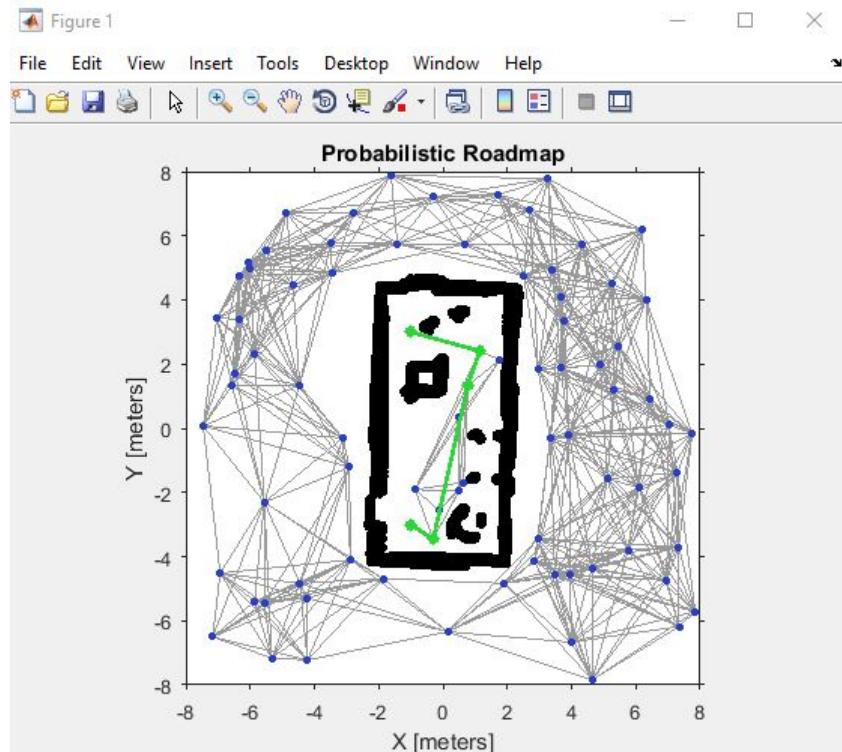
Con el fin de conectar las muestras es necesario encontrar cada una de estas y cual es su vecina mas cercana.

Query phase Conocida como fase de búsqueda, en ella se intenta encontrar la conectividad entre el punto final y el inicial a través de la muestras de la fase anterior. El algoritmo devuelve los nodos por los que tiene que pasar el robot.


Figura 4.10: Esquema algoritmo PRM

Como se observa en la Figura 4.10 el algoritmo si divide en un bucle para encontrar todas las muestras en el espacio libre para, posteriormente conectarlas entre ellas, y buscar el camino final.

Si el entorno en el que se encuentra el robot es muy complejo este tipo de algoritmo permite encontrar una solución sin un gran requerimiento computacional, por lo que se utiliza en espacio con gran número de dimensiones.


Figura 4.11: Aplicación de PRM

Uno de los principales problemas de PRM es que las soluciones que encuentra no tiene por que ser el camino óptimo. Además debido a que la forma de generar los nodos es completamente aleatoria, produce una distribución no homogénea de las muestras en el espacio. Estas dos desventajas se observan en la Figura 4.11 [13].

4.6.2. Fast Marching Square

Fast Marching Square (FM^2) es un algoritmo de path planning, que a diferencia de PRM busca la trayectoria optima entre dos puntos.

FM^2 se podría considerar que está basado en la creación de campos potenciales, pero se diferencia de este tipo de algoritmos debido a que tiene en cuenta el concepto de tiempo, considerando al robot como una partícula que se mueven bajo la influencia del tiempo. Eligiendo el camino que emplea menor tiempo posible para alcanzar un objetivo.

Este usa como base de calculo el método Fast Marching Method (FMM). Se trata de un algoritmo de seguimiento y modelado de una interfaz física de onda.

Fast Marchin Method

Fast Marching Method utiliza una función con comportamiento similar al de propagación de una onda en un fluido. La forma que dicha onda se propaga, siguiendo la ecuación Eikonal, de un punto inicial hasta alcanzar un objetivo es el camino más eficiente en términos de tiempo para alcanzarlo [10].

Esta onda se denomina interfaz, el algoritmo de FMM calcula el tiempo (T) que la interfaz emplea para alcanzar cada punto del espacio partiendo de un punto inicial, pudiendo haber más de uno de estos.

FMM parte de la discretización del entorno en una rejilla de celdas sobre la cual se resolverá de manera iterativa la ecuación de Eikonal.

Cada una de estas celdas puede estar definida de la siguiente forma :

Unknown Celda por la que la onda no ha pasado, por lo que no se conoce el valor de T .

Narrow Band Conjunto de celdas por la que la onda se expandirá en la próxima iteración. Estas tienen asignado un valor de T que puede cambiar en las sucesivas iteraciones.

Estas representan la interfaz de la onda.

Frozen Aquellas celdas por las que la onda ya ha pasado y que por consiguiente mantienen un valor de T fijo.

El algoritmo consta de tres pasos:

- **Inicialización** Se establece el valor de $T = 0$ en las celdas donde se origina la onda y son etiquetadas como *frozen*. Se etiquetan las celdas contiguas como *narrow band* y se calcula el tiempo de llegada de T .
- **Bucle principal** Se resolverá de forma secuencial la ecuación Eikonal para las celdas etiquetadas como *narrow band* pasando a ser *frozen* cuando almacenen el menor valor de tiempo de llegada T .
- **Finalización** Cuando todas las celdas pertenecen al conjunto denominado *frozen* el algoritmo concluye.

FM² en path planning

Las trayectorias obtenidas mediante FMM presentan dos inconvenientes : gran brusquedad de giros y la obligación de que las trayectorias pasen muy próximas a los obstáculos. Haciendo imposible el uso del algoritmo como planificador de trayectorias en robótica real.

La principal modificación en FM^2 consiste en solucionar estos problemas mediante la generación de un mapa de velocidades que modifique la expansión de la onda teniendo en cuenta la cercanía a los obstáculos [11].

4.7. Seguidor de trayectorias

En las secciones anteriores se describen las distintas fases para la consecución de un robot móvil autónomo, desde el reconocimiento del entorno y localización hasta los planificadores de trayectorias.

En esta sección se mencionará los algoritmos de control que permiten al robot la consecución estas trayectorias de forma autónoma. Los algoritmos que se mencionarán trabajan con un vector de coordenadas a seguir que puede ser la solución de PRM.

El controlador más sencillo consiste en tres pasos repetitivos. Iniciando con el cálculo del ángulo que forman las coordenadas actual y el punto consecutivo de la trayectoria, para que el robot se oriente. Una vez completado este procedimiento se procede a un avance en línea recta hasta la posición deseada.

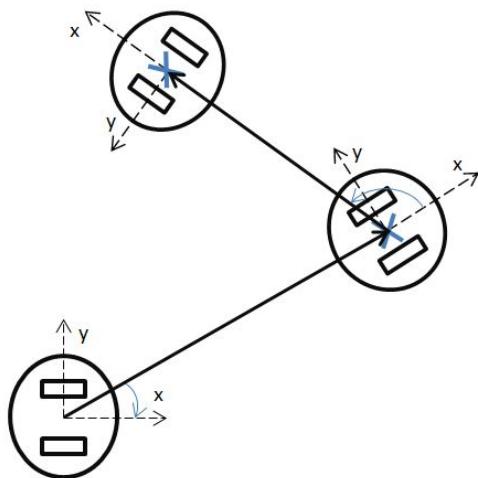


Figura 4.12: Aplicación de PRM

Como se observa en la Figura 4.12 es un proceso cíclico hasta alcanzar la posición final del set de coordenadas.

El movimiento durante la trayectoria de un controlador de este tipo no es ni fluido ni óptimo, una forma de mejorar este aspecto sería realizar movimientos simultáneos de giro y avance en las zonas cercanas al cambio de ángulo. Un algoritmo que cumple con estos requisitos de movimiento es PurePursuit [22].

4.7.1. PurePursuit

PurePursuit es un algoritmo de seguimiento de trayectorias que calcula la curvatura con la que el robot debe moverse durante su recorrido para alcanzar la posición final [21].

El algoritmo funciona utilizando un punto de la trayectoria que se encuentra una distancia determinada por delante de la posición en cada momento del robot, el cual el robot debe alcanzar (*pursuit*).

El controlador devuelve la velocidad que debe tener en cada momento el robot para seguir la trayectoria. Lo que realiza es el cálculo de la velocidad angular, manteniendo la velocidad linear constante, desde la posición en la que se encuentra al punto por delante en la trayectoria.

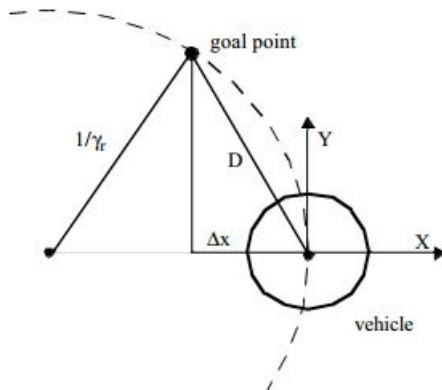


Figura 4.13: Curvatura con PurePursuit

Un arco une estos dos puntos, el cual debe recorrer el robot. La cuerda de este arco será la distancia hasta ese punto que se encuentra delante que persigue, Figura 4.13.

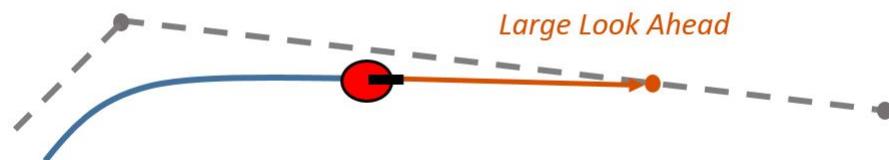


Figura 4.14: Cuerda corta con PurePursuit

La magnitud de la cuerda va influir de manera notable en el comportamiento del movimiento. Cuerdas de un gran tamaño generarán movimiento más fluidos pero que en algunas ocasiones pueden producir que el robot no siga a la perfección el camino exigido, Figura 4.14.

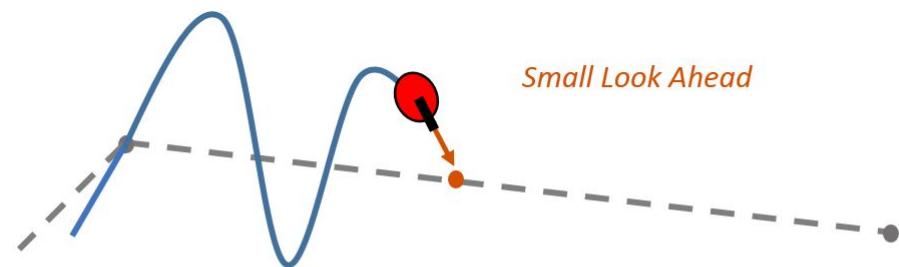


Figura 4.15: Cuerda corta con PurePursuit

En cambio cuerdas pequeñas producen giros bruscos e incluso oscilaciones, Figura 4.15 [27].

4.8. Programa autónomo

Un programa autónomo completo sería el cual engloba a todos los procesos anteriores, siguiendo el esquema que se muestra a continuación, Figura 4.16:

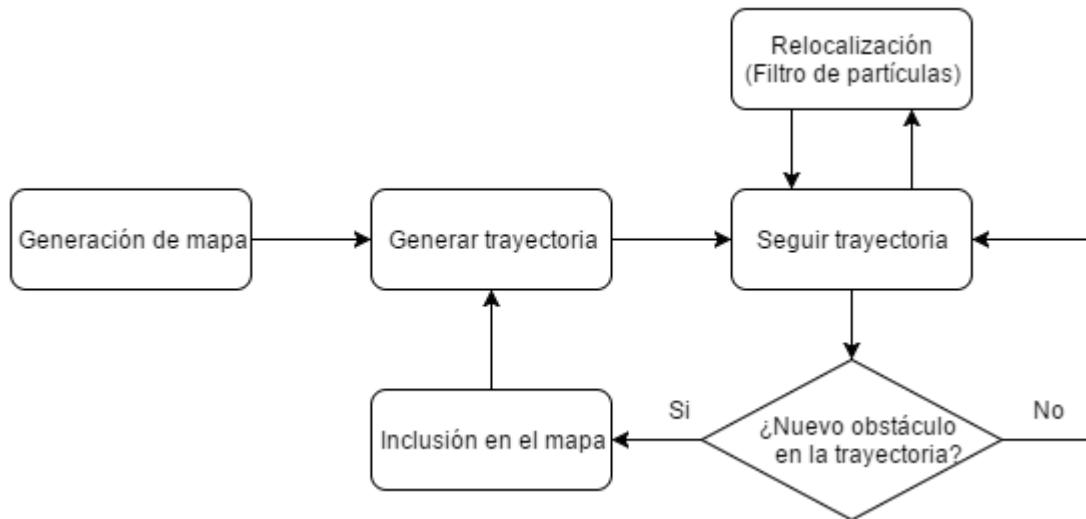


Figura 4.16: Esquema de programa autónomo

Así el programa se iniciaría con un mapa del entorno por el que se mueve el robot, para posteriormente generar la trayectoria a seguir para alcanzar un objetivo.

Con PurePursuit el robot seguiría el conjunto de posiciones que componen la trayectoria. Si durante el recorrido de esta el robot se encuentra con un objeto que no estaba señalizado en el mapa, se añadiría este obstáculo y se recalcularía la trayectoria desde el punto en el que se encuentra.

Como se observa en el esquema, Figura 4.16, durante el movimiento del robot se realiza la relocalización del robot mediante el filtro de partículas para eliminar lo máximo posible los errores de posición.



Capítulo 5

Implementación

En este capítulo se explicará como se han implementado los algoritmos mencionados en el capítulo anterior, Capítulo 4.

5.1. Mapping

La función *mapa* utiliza los datos del láser y la odometría para crear el mapa del entorno, siendo estos tres elementos lo que se deben pasar.

En la línea 5 se guardan los datos del láser en forma cartesiana. A continuación se extraen todos los datos de la odometría y, en la línea 20, se realiza la transformación para combinar los datos del láser con los de la odometría.

```
1 function mapa(laser,odom,map)
2
3
4 scan = receive(laser,3);
5 data = readCartesian(scan) * [0 1; -1 0];
6
7
8 odomdata = receive(odom,3);
9 pos = odomdata.Pose.Pose;
10 xpos = pos.Position.X;
11 ypos = pos.Position.Y;
12 quat = pos.Orientation;
13 angles = quat2eul([quat.W quat.X quat.Y quat.Z]);
14 theta = angles(1);
15 pose = [xpos, ypos, theta];
16 th = pose(3)-pi/2;
17
18
19 dataWorld = data*[cos(th) sin(th);-sin(th) cos(th)] + ...
20 repmat(pose(1:2),[numel(data(:,1)),1]);
```

```

21
22 setOccupancy(map,dataWorld, 1);
23
24 end

```

Por último esta información almacenada en la variable *dataWorld*, se utiliza para graficar el mapa por donde se desplaza el robot mediante *BinaryOccupancyGrid* utilizando el comando *setOccupancy*

5.1.1. BinaryOccupancyGrid

BinaryOccupancyGrid es una potente herramienta para representar y visualizar el espacio de trabajo del robot. Utiliza los datos del sensor láser y la posición para estimar la localización aproximada de los objetos que rodean al robot.

Con la primera línea de código mostrado a continuación se crea un mapa con unas dimensiones y resolución deseadas. Posteriormente se establecen las celdas de este que están ocupadas por objetos, esto se realiza asignando un 1 a las ocupadas y 0 a las libres. Mediante el comando *show*, línea 3, se muestra de forma gráfica el mapa.

```

1 map = robotics.BinaryOccupancyGrid(width,height,resolution)
2 setOccupancy(map,xy,1)
3 show(map)

```

Además dispone de otras acciones como *inflate* o *copy* que se utilizarán más adelante, que tienen un gran interés para la creación de mapas para el posterior uso de planificadores de trayectorias.

5.2. Teleoperación

El algoritmo de Teleoperación usa una ventana independiente para recoger la información de si se ha presionado una tecla, y esto se traduce en acciones en movimiento del robot.

Como se observa en la Figura 5.1 dentro de las opciones disponibles en el algoritmo se encuentran:

- Movimiento de avance y retroceso con las teclas *w* y *s*
- Movimiento de giro tanto horario y antihorario con las teclas *a* y *d* respectivamente
- Aumento o decremento tanto de velocidad linear y angular con las parejas de teclas *u-j* y *i-k*

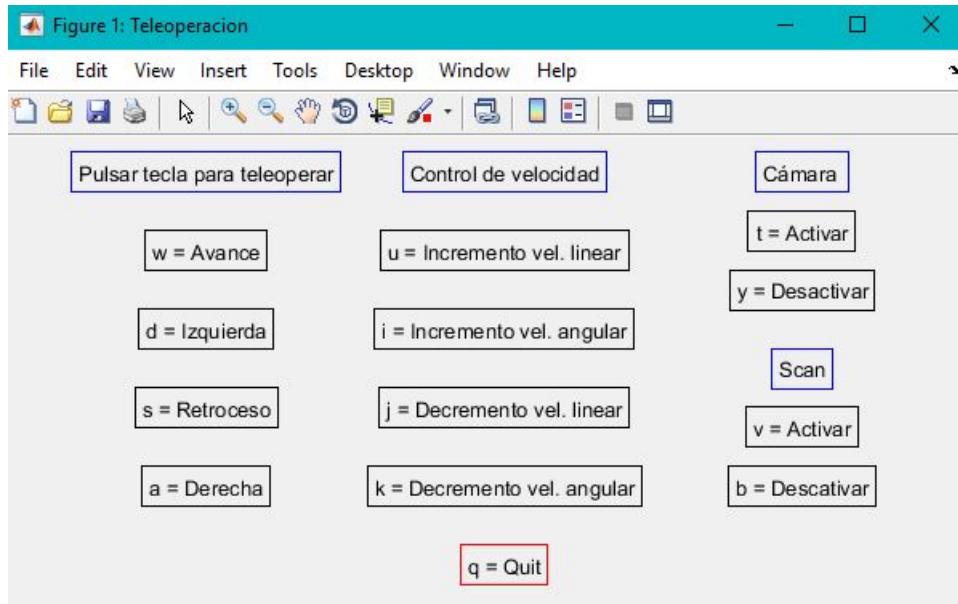


Figura 5.1: Ventana de teleoperación

- Activar o desactivar la visualización de los datos de la cámara y láser. Como se ha mencionado en la sección 4.1, en los casos en el que el robot se encuentra en una posición remota son muy necesarias estas imágenes.
- Cierre del bucle mediante la tecla *q*

La ventana alrededor la cual se articula el programa se crea mediante la función *teclas()*, Anexo A.2. Una sencilla función que crea una figura con la información que queremos mostrar y que posee el código para captar si se ha oprimido una tecla.

La implementación de la teleoperación se realiza mediante una función con el mismo nombre(Anexo A.1), a la cual se le pasan la suscripción al los *topics* /mobile_base/commands/velocity, /scan, /odom y /camera/rgb/image_color/compressed además del mensaje de velocidad y la variable *map*.

Esta última se corresponde a un mapa creado por *BinaryOccupancyGrid* ya que se ha incluido el reconocimiento del medio en el algoritmo.

Se comienza con la inicialización de variables. En la línea 5 y 6 se encuentran los comandos de inicialización de la ventana de teleoperación y la variable donde se almacena el valor asociado a la tecla oprimida.

A continuación, se inicializan las variables asociadas con la velocidad siendo *velavance* y *velgiro* las máximas velocidades en *m/s* y *rad/s*, y *avance* y *giro* las variables de velocidad instantánea del robot. La variable *camara* maneja si la visualización de datos esta activada o no.

Por último se realiza la inicialización del *robotics.Rate* para el control de ejecución del bucle con una frecuencia de 10 Hz, lineas 13 al 15.

El bucle de control se ejecuta hasta que se presiona la tecla *q*, así se realiza un *while* siempre y cuando la variable *teclado* no equivalga a este valor. Al final de cada bucle se realiza un reset a *robotics.Rate* para comenzar la ejecución del código.

En la línea 25 se pasan los datos del robot para realizar el mapa, sección 5.1, y a continuación se encuentra el comando de envío de velocidades instantáneas.

La parte más importante del bucle de control se encuentra en el *switch reply* en el cual dependiendo de la variable *teclado* se realizarán distintas opciones. Esta variable se actualiza mediante *getKeystroke(teclado)*, función de Matlab que reconoce la tecla presionada en cada momento.

Así los primeros cuatro casos se limitan a variar la velocidad instantánea tanto lineal o angular. Para que el cambio de la velocidad tenga lugar de una forma más paulatina se utilizan las funciones *acelerarm* y *decelerarm* (se presentaran al final de esta sección) que con la información de la velocidad máxima, la velocidad instantánea y tasa de variación incrementan o decrementan la velocidad.

El algoritmo no permite el movimiento lineal y angular simultaneo, por lo tanto en los casos de variación de velocidad linear se impone la velocidad de giro cero y viceversa.

Los cuatro siguientes casos controlan las velocidades máximas, al presionar cualquiera de las teclas para aumentar o disminuir la velocidad máxima permitida se realiza el cambio con una variación de 0.05 y se imprime por terminal la velocidad final.

Posteriormente, se encuentran los casos relacionados con visualización de datos (línea 79), esta parte del código sigue una estructura muy simple dependiendo de si se desea visualizar sólo los datos de la cámara (*camara=1*) o sólo el láser (*camara=2*) o ambos (*camara=3*) se varía el valor de la variable (*camara*), para ser posteriormente comparado en otro *switch camara*, línea 108. El cual controla la forma de visualización de las imágenes en la figura.

Finalmente, para cerrar el bucle se encuentra la igualación de las variables de velocidad que serán enviadas al robot y el *wait(for)* correspondiente al *rosrate*.

Si durante la ejecución del bucle se presione *q* este finalizaría, se igualarían las variables de velocidad a cero y se cerraría la ventana de teleoperación.

A continuación se presenta la estructura de la función *acelerarm*. Realiza un cambio gradual de la velocidad inicial a la deseada incrementando de forma lineal su valor.

```

1 function [speed]=acelerarm(speedmax,speedactual,paso)
2
3 incremento=abs((speedmax-0.1)/paso);
4 speed=speedactual+incremento;
5
6 if speed>=speedmax
7 speed=speedmax;
8 end
9 end
10

```

La función *decelerarm* es homóloga a la anterior.

5.3. Wanderer

Al algoritmo de Wanderer al igual que el de teleoperación se le ha incluido la adquisición de datos del entorno en forma de mapa.

En la inicialización se encuentran las velocidades máximas e instantáneas, además la variable *distanciaThreshold* que determina la mínima distancia a la que el robot va a acercarse de los objetos que le rodean, líneas 4 al 9.

Si se analiza el bucle de control, se inicia con el envío de los datos para la generación del mapa, en la línea 19 se envía los valores de velocidad del robot.

Posteriormente, línea 23, se extraen los datos del láser, y de ellos se individualiza la menor distancia para posteriormente ser comparada con *distanciaThreshold* en el caso de ser mayor que esta se continua con un movimiento rectilíneo.

Si por el contrario el valor de la distancia mínima es menor que *distanciaThreshold* el robot debe realizar un giro. La dirección del giro va a depender de el criterio donde se encuentre esa distancia mínima con respecto al conjunto de datos del láser.

Estas acciones se repiten hasta cumplir con los requisitos de tiempos establecidos.

```

1 % Inicializar variables
2
3 velgiro = 0.45;
4 velavance = 0.15;
5 distanciaThreshold = 0.8;
6 giro=0;
7 avance=0;
8 i=0;
9

```

```
10
11 %Bucle de control
12
13 while i <1500
14
15     SLAMmapa(laser,odom,map3);
16
17     send(vel,velmsg);
18     pause(0.5);
19
20     scan = receive(laser);
21     data = readCartesian(scan);
22     x = data(:,1);
23     y = data(:,2);
24     dist = sqrt(x.^2 + y.^2);
25     minDist = min(dist);
26     distsize = size(dist);
27     criterio = find(minDist==dist)/distsize(1);
28
29     if minDist <= distanciaThreshold
30
31         if criterio < 0.5
32             giro=acelerarm(velgiro,giro,10);
33             avance = 0;
34         else
35             giro = decelerarm(-velgiro,giro,10);
36             avance = 0;
37         end
38
39     else
40         avance= acelerarm(velavance,avance,15);
41         giro = 0;
42     end
43
44 velmsgAngular.Z= giro;
45 velmsgLinear.X = avance;
46
47
48 i=i+1;
49
50
51 end
```

5.4. Localización

Para ejecutar el filtro de partículas, el algoritmo necesita de un modelo de la odometría del robot, en este caso se le proporciona el modelo de un robot diferencial mediante el comando de la línea 3. A continuación, se suministra un vector

con los valores de la distribución gaussiana del ruido de los sensores.

```

1 %inicializar filtro de particulas. Algoritmo de localizacion
2 Monte Carlo
3
4 odometryModel = robotics.OdometryMotionModel;
odometryModel.Noise = [0.2 0.2 0.2 0.2];

```

Lo que realiza el filtro de partículas es comprar las mediciones del láser provenientes de la Kinect, con los datos de una simulación de láser que reconoce del mapa del entorno que se le proporciona.

Así las características del sensor simulado se deben proporcionar a la estructura *robotics.LikelihoodFieldSensorModel*. A ella se le especifica el rango de visión del láser que debe coincidir con el del robot real y el mapa generado mediante *BinaryOccupancyGrid* por el que se mueve en robot.

```

1
2 rangeFinderModel = robotics.
3     LikelihoodFieldSensorModel;
4 rangeFinderModel.SensorLimits = [0.45 8];
rangeFinderModel.Map = map;

```

Además a la estructura anteriormente mencionada precisa conocer las transformada de las coordenadas de la cámara con respecto a la base del robot, que serán utilizadas para poder pasar del sistema de coordenadas del robot al de la cámara y viceversa.

Mediante la transformada *tf* descrita en la sección 3.2.1 se obtiene el el sistema de coordenadas de la base que se almacena como otra componente más de la estructura *robotics.LikelihoodFieldSensorModel*.

```

1
2 % Query the Transformation Tree (tf tree) in ROS.
3 tftree = rostf;
4 waitForTransform(tftree,'/base_link','/
5     camera_depth_frame');
sensorTransform = getTransform(tftree,'/base_link',
6     '/camera_depth_frame');

7 % Get the euler rotation angles.
8 laserQuat = [sensorTransform.Transform.Rotation.W
9             sensorTransform.Transform.Rotation.X ...
10            sensorTransform.Transform.Rotation.Y sensorTransform
11            .Transform.Rotation.Z];
laserRotation = quat2eul(laserQuat, 'ZYX');

```

```

11
12     rangeFinderModel.SensorPose = ...
13     [sensorTransform.Transform.Translation.X
      sensorTransform.Transform.Translation.Y
      laserRotation(1)];

```

Tras especificar las características del láser simulado se inicializa el algoritmo con el comando *robotics.MonteCarloLocalization*

```

1 %Inicializacion del algoritmo
2         amcl = robotics.MonteCarloLocalization;
3

```

A este se le pasa tanto el modelo de la odometría como el del láser simulado, además se especifica el intervalo entre cada iteración y el mínimo cambio de odometría que el algoritmo reconoce como no ruido. Esto se realiza para que no se recalcule la posición debido a errores en la odometría que se podrían interpretar como un cambio de posición del robots.

```

1 % Asignacion de modelos
2         amcl.MotionModel = odometryModel;
3         amcl.SensorModel = rangeFinderModel;
4
5 % Minimo desplazamiento reconocido
6         amcl.UpdateThresholds = [0.2,0.2,0.2];
7
8 % Asignacion de intervalo entre iteraciones
9         amcl.ResamplingInterval = 1;

```

A continuación, se debe de delimitar el límite inferior y superior de partículas que utilizará el filtro. Cuanto mayor sea el límite superior mayor es la probabilidad de encontrar la posición verdadera del robot, con el consiguiente coste computacional.

Debido a que el algoritmo trabaja con posiciones iniciales aproximadas, se le proporciona el valor estimado indicando *false* en *amcl.GlobalLocalization*. Si por el contrario, del robot no se tuviese ninguna certeza de la posición, se le asignaría el valor *true*.

En este último caso el algoritmo genera una nube de partículas uniforme por todo el mapa, por consiguiente requiriendo un mayor número de partículas y de iteraciones para resolver el problema de la posición .

En la línea 7 se ejecuta un visualizador del mapa con la posición del robot y la distribución de partículas del filtro.

```

1 % Inicializacion del numero de particulas
2
3     amcl.ParticleLimits = [500 5000];
4     amcl.GlobalLocalization = false;
5     amcl.InitialPose = robotCurrentPose;
6     amcl.InitialCovariance = eye(3)*0.5;
7     visualizationHelper = ExampleHelperAMCLVisualization
        (map);

```

Se comienza el bucle de control inicializando las variables de máximo número de iteraciones y la variable que lleva la cuenta de estas.

El primer paso es actualizar los datos de los sensores de odometría y láser, de este último tanto los valores de distancia con los objetos como los ángulos que estos forman. Estos serán utilizados para actualizar el algoritmo de localización en la función *step*.

Tras esto se aumenta en uno el número de iteraciones realiza la visualización de los datos hasta llegar al máximo de iteraciones.

```

1 % Bucle de control
2 numUpdates = 100;
3 i = 0;
4 while i < numUpdates
5
6     odomdata = receive(odom,3);
7     pose=odompose(odomdata);
8     theta= rad2deg(pose(3));
9
10    ranges = double(scan.Ranges);
11    angles = double(scan.readScanAngles);
12
13    [isUpdated,estimatedPose, estimatedCovariance] = step(
14        amcl, pose, ranges, angles);
15    if isUpdated
16        i = i + 1;
17        plotStep(visualizationHelper, amcl, estimatedPose,
18            ranges, angles, i)
19    end
20 end

```

5.5. Path planning

5.5.1. PRM

El algoritmo PRM utiliza un mapa para individualizar el camino que deberá seguir el robot desde un punto inicial a un objetivo deseado, este mapa se debe de proporcionar en forma de *BinaryOccupancyGrid*.

Para prevenir las colisiones con los objetos del entorno durante el seguimiento del path el algoritmo infla el entorno con el valor del radio. Así en la línea 3 se especifica el radio de robot para posteriormente realizar un copia del mapa e inflarlo. Estas dos ultimas acciones se realizan con comandos específicos de *BinaryOccupancyGrid*, como son *copy()* e *inflate()*.

```
1  robotRadius = 0.15;
2  mapInflated = copy(map);
3  inflate(mapInflated,robotRadius);
4  show(mapInflated)
```

Una vez realizadas estas acciones se procede a la llamada del algoritmo mediante el comando *robotics.PRM*. A esta estructura se le debe asignar:

- Mapa inflado del entorno.
- Número de nodo.
- Distancia de conexión entre nodos.
- Posición inicial y final del robot.

```
1  % Inicializacion de PRM
2  prm = robotics.PRM
3
4  prm.Map = mapInflated;
5  prm.NumNodes = 80;
6  prm.ConnectionDistance = 5;
7  startLocation = [-1 -3];
8  endLocation = [-1 3];
```

Cuando el algoritmo tiene todos los datos necesarios se ejecuta con la llamada a la función *findpath()*. En el caso, que con la combinación de número de nodos y distancia entre ellos, no de lugar a una solución se incrementará el número de nodos hasta conseguir una solución.

```

1 path = findpath(prm, startLocation, endLocation)
2 show(prm)
3 while isempty(path)
4 prm.NumNodes = prm.NumNodes + 5;
5 update(prm);
6 path = findpath(prm, startLocation, endLocation);
7
8 end

```

El algoritmo devuelve el camino a seguir por el robot como un conjunto de puntos x e y por lo que debe pasar.

5.5.2. FM2

El algoritmo de Fast Marching Square (Anexo A.4), fue realizado por Luis Santiago Garrido Bullón, [11]. Y consiste en un algoritmo de simulación de trayectorias, el cual se modificó para comprobar su viabilidad durante la ejecución real con un robot.

Se inicia aplicando FM^2 sobre el mapa de arquitecto proporcionado para calcular una primera trayectoria. Posteriormente se ejecuta el bucle de control el cual se limita realizar la trayectoria inicial hasta que se encuentra un objeto inesperado.

En este caso se incorpora a un mapa temporal y se realiza FM^2 local, únicamente en las zonas cercanas al robot para reducir al máximo el esfuerzo computacional.

Mencionar que este algoritmo no trabaja con mapas *BinaryOccupancyGrid* si no en **.bmp*. Como controlador de trayectorias se implementó PurePursuit.

5.6. PurePursuit

PurePursuit es un algoritmo cuyo cometido es el de calcular la velocidad, tanto linear como angular, que debe suministrarse los motores al robot para seguir los puntos de una trayectoria.

De este modo la posición inicial como la final de la trayectoria son dos elementos que deben de suministrarse para inicializar el algoritmo, además de la posición actual del robot en el entorno. Para facilitar los cálculos se reinician los encoder de la odometría por lo que la posición inicial del robot sería el primer conjunto de coordenadas de la trayectoria.

```

1 robotCurrentLocation = path(1,:);
2 robotGoal = path(end,:);
3 initialOrientation = 0;
4 robotCurrentPose = [path(1,:); 0];
5

```

El siguiente paso consiste en generar la estructura de PurePursuit con el comando *robotics.PurePursuit*, que precisa para su correcta ejecución el set de coordenadas a seguir, la velocidad lineal y angular máxima y la magnitud de distancia que corresponde con cuanto delante esta el punto que persigue con respecto a la posición actual.

```

1 controller = robotics.PurePursuit
2 controller.Waypoints = path;
3 controller.DesiredLinearVelocity = 0.15;
4 controller.MaxAngularVelocity = 0.3;
5 controller.LookaheadDistance = 0.3;
6
7 goalRadius = 0.15;
8 distanceToGoal = norm(path1(1,:) - robotGoal);
9

```

En la línea 8 y 9 se encuentran dos variables importantes para la ejecución del bucle de control del algoritmo. La primera *goalRadius* especifica con cuanta exactitud el robot debe alcanzar el objetivo final. Mientras que la segunda especifica la distancia en linea recta entre la posición inicial y final de la trayectoria.

```

1 while( distanceToGoal > goalRadius )
2
3
4 odomdata = receive(odom,3);
5 poseodom=odompose(odomdata);
6 pose=pose+[path(1,1) path1(1,2) 0];
7
8 [v, omega] = step(controller, pose );
9 velmsg.Angular.Z= omega;
10 velmsg.Linear.X =v;
11
12 send(vel,velmsg);
13
14
15 distanceToGoal = norm([pose(1) pose(2)] - robotGoal);
16
17 end

```

Finalmente se realiza el bucle de control hasta que *distanceToGoal > goalRadius*. En cada ejecución del *while* se realizan las siguientes acciones :

- Actualización de la posición del robot con la odometría
- Cálculo de la velocidad lineal y angular del robot, linea 8.
- Envío de velocidad

- Recálculo de la distancia para alcanzar el objetivo, *distanceToGoal*

Como se observa en la línea 6 la posición del robot en cada momento consiste en la posición inicial de la trayectoria más el incremento de los encoders que contabilizan la distancia con respecto al origen.

5.7. Programa autónomo

Utilizando las estructuras anteriormente mencionadas se obtiene el código para la consecución de un movimiento autónomo, Anexo A.3.

Con los mismos procedimientos explicados en los apartados anteriores se realiza la inicialización del filtro de partículas (5.4), del controlador PurePursuit (5.6). Ademas se parte de un mapa ya generado del entorno mediante *BinaryOccupancyGrid* y una trayectoria inicial generada por PRM (5.5.1).

Las etapas que sigue el bucle de control son las siguientes;

- Inicialización de variables
- Actualización de sensores, odometría y kinect.
- Ejecución de filtro de partículas para el calculo de la posición estimada.
- Alineación con el primer punto de la trayectoria.
- Detección de obstáculos.
- Cálculo de velocidades de PurePursuit.

De estas etapas cabe mencionar dos de ellas en detalle.

La alineación (a partir de la linea 51) con el primer punto de la trayectoria se realiza debido a que por sencillez y para no incorporar mayor incertidumbre a la posición inicial, siempre se coloca el robot en su primer punto con una orientación alineada con los ejes del sistema de coordenadas. Esto provoca que en determinadas trayectorias el robot no esté alineado con el camino que debe seguir.

El controlador de trayectorias PurePursuit envía siempre tanto velocidad angular como lineal, en los casos en que el cambio de orientación inicial sea muy brusco se realiza un giro previo del robot sobre si mismo, para que el movimiento no modifique la trayectoria delimitada. Se ha establecido que si la diferencia entre la orientación del robot y el ángulo de la trayectoria se sea mayor de 15 grados se ejecute esta reorientación.

Esta alineación consiste en un simple cálculo del ángulo de la trayectoria, para posteriormente enviar un mensaje de velocidad con una valor de giro determinado hasta alcanzar la orientación deseada. Una vez realizada la acción se actualizan



las variables de posición y orientación que utiliza el algoritmo de PurePursuit

La otra etapa importante es la detección de obstáculos (línea 110), esta solo se ejecutada si el láser detecta un nuevo objeto a una distancia menor de 0,6 metros. En este caso se incorpora el objeto en el mapa local para recalcular la trayectoria, posteriormente se procedería a la alineación con el primer punto de la nueva trayectoria si fuese necesario.

Capítulo 6

Resultados experimentales

En esta sección se comentarán los datos experimentales obtenidos durante la realización del trabajo.

Debido a que los programas utilizados son básicamente de navegación los conceptos bajo estudio serán:

- Consecución de posición objetivo
- Tiempos de ejecución
- Trayectoria real frente teórica
- Fluidez de movimiento

Todos los datos han sido tomados en el laboratorio del Departamento de Ingeniería de Sistemas y Automática.

6.1. Plataforma robótica : Turtlebot

Turtlebot es un robot móvil low-cost basado en la filosofía software y arquitectura hardware libre, esto significa que puede utilizar elementos fabricados por distintas compañías, ya sea la base robótica el ordenador o la cámara .

Es un robot especialmente indicado para la investigación y prueba de algoritmos debido a que permite el acceso información de más bajo nivel [28][20].

En concreto se utilizará la segunda versión de este robot, Turtlebot 2, Figura 6.1.



Figura 6.1: Turtlebot 2

Para este caso de estudio el robot dispone de los siguientes componentes:

Mobile Base: Kobuki, componente fundamental del robot. Consiste en un robot de cinemática diferencial el cual permite la locomoción, contiene los sensores para la odometría, además de la fuente de alimentación en forma de batería.

Sensor 3D: ASUS Xtion PRO, dispone de un sensor de infrarrojos, detección adaptativa de profundidad y cámara con imagen a color.

Portatil: Procesador del conjunto , trabaja bajo *ROS* con la distribución de linux Ubuntu 14.04 LTS (Trusty Tahr). La versión de *ROS* bajo la que trabaja el robot es *ROS Indigo Igloo* [26].

6.2. Generación de Mapas

La generación de mapas como se ha visto a lo largo de este trabajo es uno de los pilares principales en un algoritmo autónomo.

Experimentalmente se ha comprobado que el algoritmo, 5.1, presenta errores debido al procesamiento de información dependiendo a la velocidad con la que reciba los datos de la odometría y kinect.

Esto se debe a un envío asíncrono de la información de ambos *topics* y a la limitación de capacidad computacional de Matlab ya que puede que se pierda la recepción de datos debido a que el código se está ejecutando.

Este problema se soluciona utilizando *robotics.Rate* para controlar el flujo de información para que ninguno de los dos casos mencionados ocurran o acotarlos lo máximo posible.

Así se ha realizado una comparativa de mapas generados a diferente frecuencia de ejecución generados con el algoritmo Wanderer (5.3):

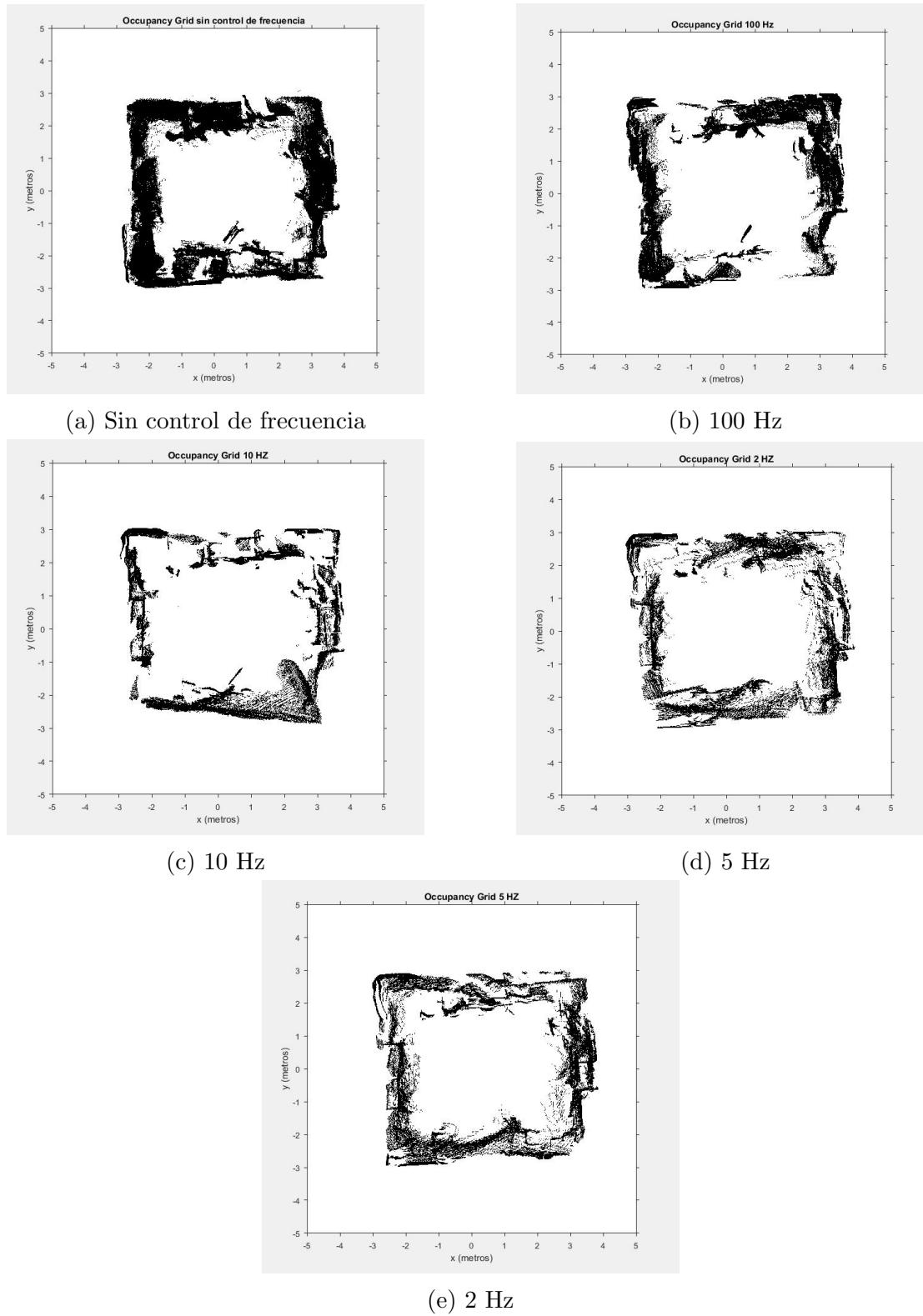


Figura 6.2: Comparación de generación de mapas a distintas frecuencias

Como se observa en la Figura 6.2 existe una gran diferencia en realizar el mapeo de un entorno sin control de frecuencia (Figura 6.2a) durante la ejecución a si realizarlo. Además cabe mencionar que a menor frecuencia mayor nitidez, pero a su vez conlleva un mayor retraso en el envío de información.

El mapa se genera a la vez que el robot se mueve por el entorno, por lo tanto si aumenta el tiempo entre envío de velocidades esto perjudicará la fluidez de movimiento, produciendo que el robot se pare de forma intermitente generando vibraciones en él, situación en la que se encuentra a partir de frecuencias inferiores a 2 Hz. Esto puede producir un aumento en la incertidumbre de la odometría que perjudique la obtención de mapas.

La frecuencia también afectará al tiempo total de mapeo:

Tiempos	Sin control	100 Hz	10 Hz	5 Hz	2 Hz
Segundos	84.752	96.993	131.977	241.636	263.94

Tabla 6.1: Tiempo de mapeo

Como se observa en la tabla 6.1 para mapear un mismo entorno el tiempo requerido para la tarea aumenta al diminuir la frecuencia de control.

6.3. Visualización de datos

Otro de los aspectos temporales que se comprueba es el tiempo necesario para la ejecución de código con la visualización de la cámara y el láser.

Para ello se realizó un movimiento en línea recta con el algoritmo teleoperación, 5.2, que presenta menor requerimiento computacional que el resto de programas para que los tiempos mostradas consistan principalmente en procesamiento de imagen.

Tiempos	Sin visualización	Visualización de cámara	Visualización de láser	Visualización de ambos
Segundos	0.1017	0.2834	0.2072	0.3155

Tabla 6.2: Tiempo de ejecución

La tabla 6.2 muestra el tiempo medio entre ejecución de cada bucle de control. Como era de esperar el menor tiempo corresponde al caso sin visualización y el mayor al momento en que se visualizan las dos opciones cámara y láser, como se puede apreciar en la Figura 6.3.

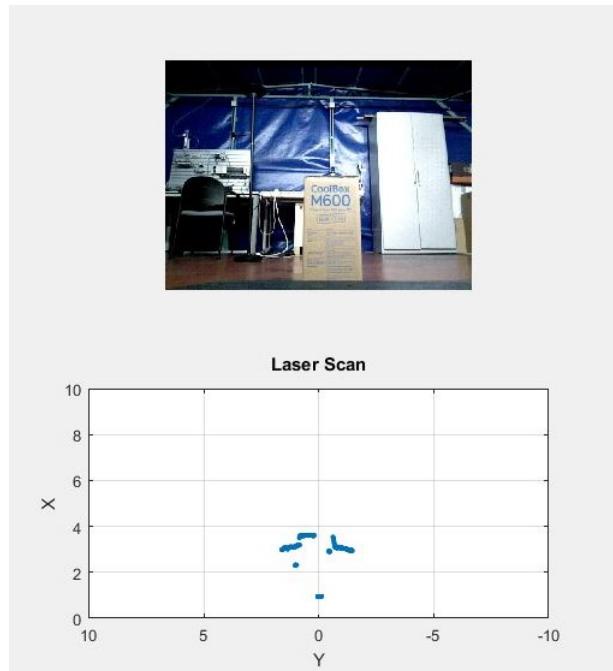


Figura 6.3: Visualización de cámara y láser

Cabe destacar que en el caso de la teleoperación este retraso debido al procesamiento no influye al funcionamiento del algoritmo ya que es aproximadamente a partir de los 0.6 segundo que el robot presenta un comportamiento intermitente.

Además mencionar que las imágenes presentan un delay debido a la conexión WiFi y a la forma intrínseca de procesamiento de Matlab.

6.4. Programa Autónomo

6.4.1. PRM

PRM es un algoritmo planificador de trayectoria cuyo resultado está influenciado por dos parámetros:

- Número de nodos.
- Distancia de conexión entre nodos en metros.

Estos no solo tendrán una gran importancia en la trayectoria final generada, como en el tiempo para obtenerla. Así, con un mismo mapa e iguales posiciones inicial y objetivo, se observa en la Figura 6.4, como existe una clara tendencia a aumentar el tiempo de cálculo al incrementarse el número de nodos, debido a que la posibilidades de combinatoria entre ellos crece.

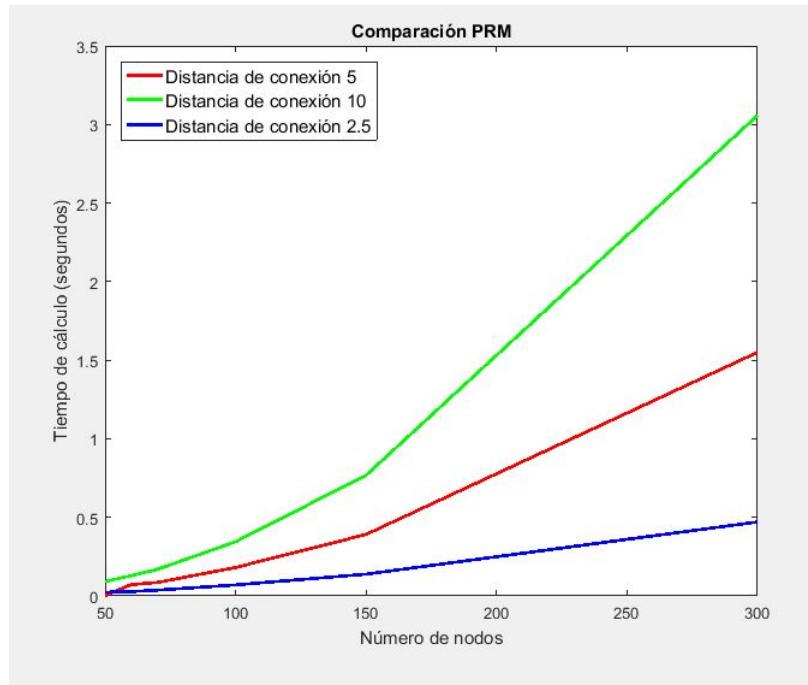


Figura 6.4: Visualización de cámara y láser

Pero además se comprueba que a medida que se impone un mayor número de nodos la diferencia computacional crece variando la distancia de conexión, esto esta altamente relacionado con el entorno por donde se mueve el robot, como se observa en la Figura 6.5. El mapa representa un área de trabajo de unos 25 m^2 , por lo que si se impone distancias de conexión grandes las posibilidades de encontrar una solución decrecen.

Todos estos datos se encuentran desglosados en la tabla 6.3 contigua, donde se observa que los tiempos de ejecución son relativamente bajos.

Nº nodos	50	60	70	100	150	300
Distancia de conexión 2.5	0.0234 s	0.0274 s	0.0366 s	0.0699 s	0.1385 s	0.4708 s
Distancia de conexión 5	0.017 s	0.0723 s	0.0855 s	0.1809 s	0.3913 s	1.5501 s
Distancia de conexión 10	0.0918 s	0.1287 s	0.1690 s	0.3448 s	0.7679 s	3.0579 s

Tabla 6.3: Tiempo de calculo de PRM

A continuación se muestran las trayectorias obtenidas variando el número de nodos manteniendo la distancia de conexión de 2,5 metros.

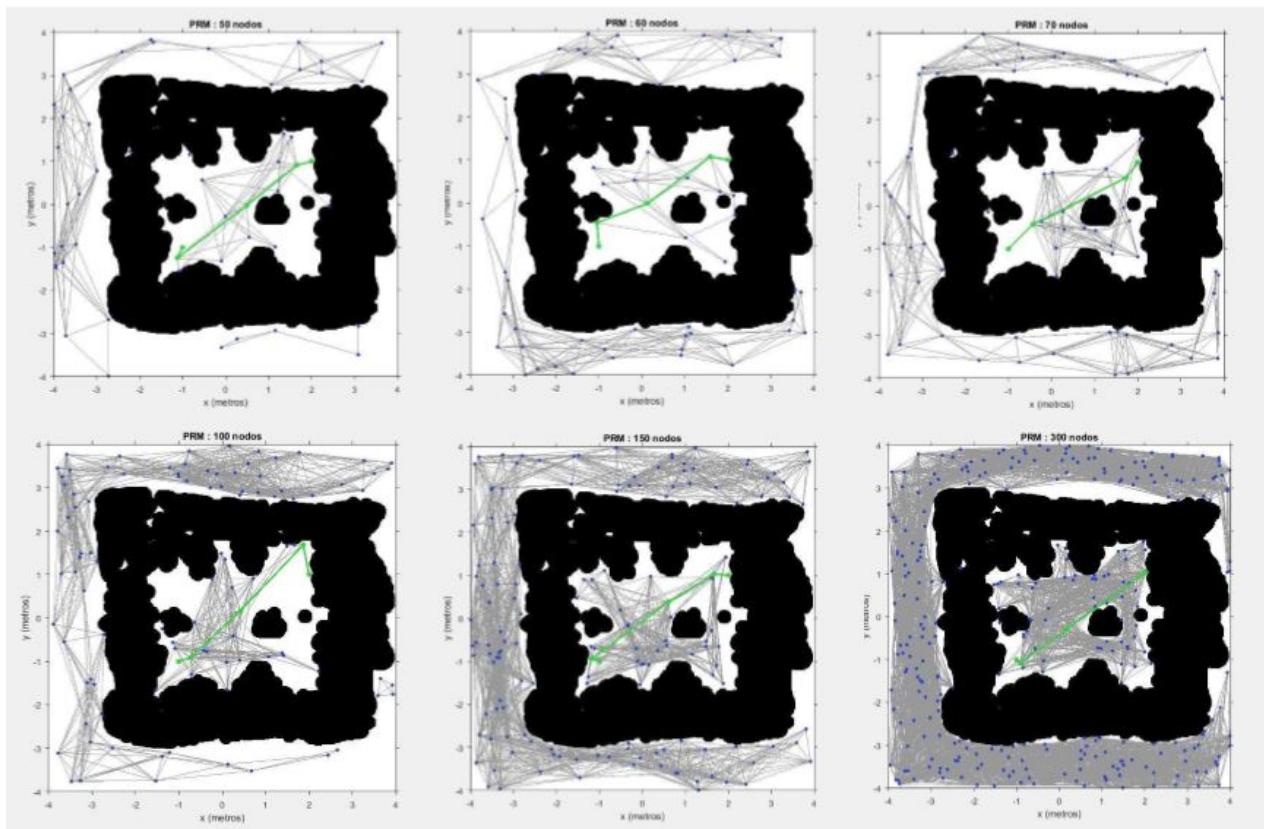


Figura 6.5: PRM variación de nodos

Al ser PRM un algoritmo aleatorio no siempre asegura que al aumentar el número de nodos mejore la trayectoria final, concepto que se puede observar con facilidad en la Figura 6.5.

Se comprueba que hay trayectorias como puede ser la obtenidas con 60, 70 y 300 nodos que pasan cerca de obstáculos, si bien estos han sido inflados para evitar colisiones, en la vida real trayectorias cercanas a obstáculos no son seguras debido a los distintos errores que puede acumular un robot, como puede ser inexactitud en la localización.

Otro de los factores que no pueden ser controlados modificando los parámetros de ejecución es la optimalidad de la trayectoria con respecto a la longitud de la trayectoria y giros bruscos que no podrán ser realizados por las restricciones cinemáticas y dinámicas del robot.

Trayectorias reales

Una vez obtenida la trayectoria que debe realizar el robot el siguiente paso es la consecución de la misma. Las pruebas realizadas en esta sección fueron usando el algoritmo PurePursuit con una velocidad angular máxima de 0.45 rad/s y 0.15 m/s de velocidad linear, sobre una trayectoria generada por PRM con 50 nodos y una distancia de conexión igual a 5 metros.

Se estudió las variaciones en la trayectoria real al variar el valor Lookahead-Distance, que se corresponde a la distancia a la que se encuentra el punto que el robot persigue durante el movimiento.

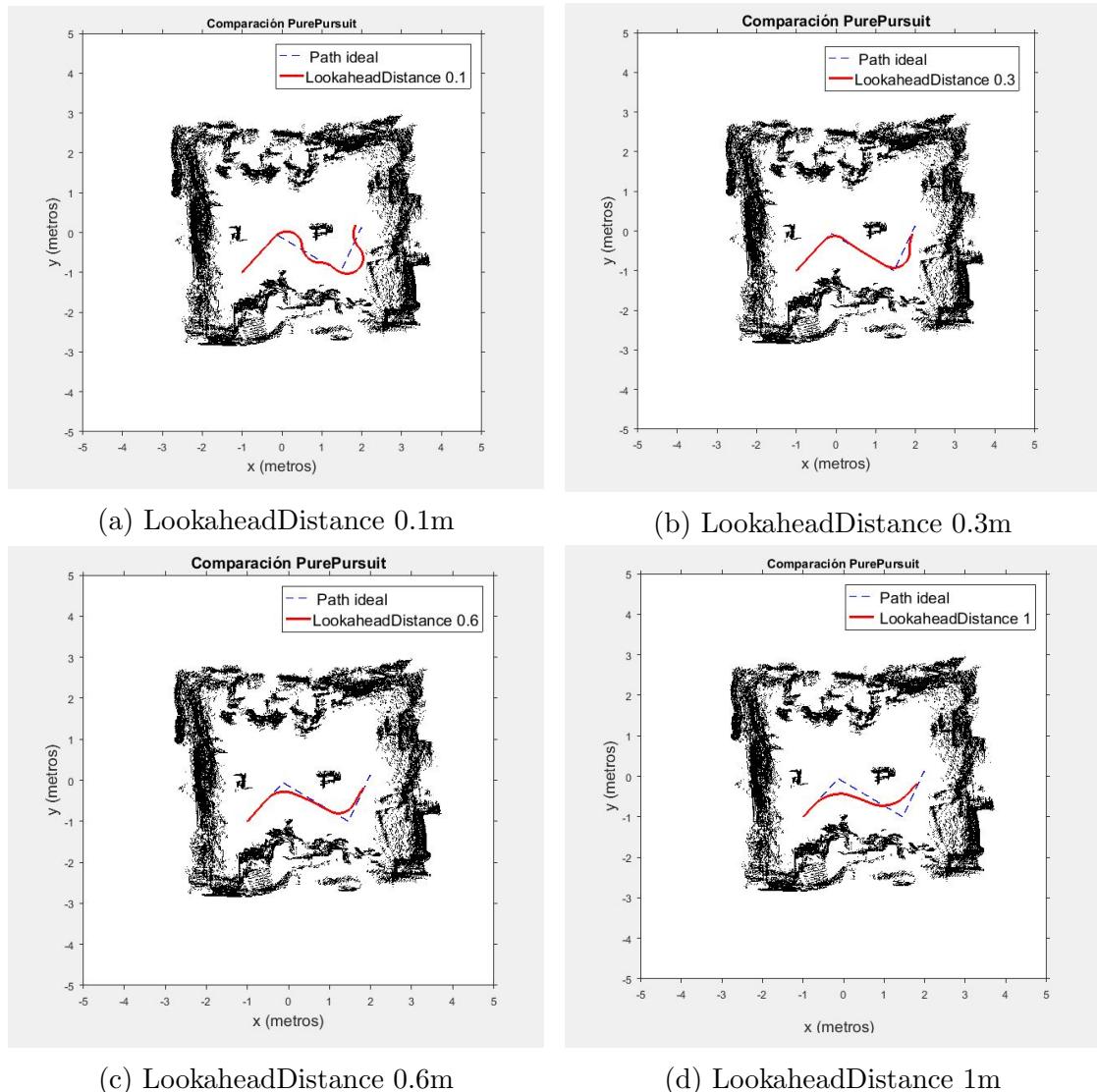


Figura 6.6: trayectorias reales PurePUsuit

De forma experimental se observa (Figura 6.6) que con valores bajos de LookaheadDistance el robot sigue un trayectoria algo alejada a la deseada debido a los giros bruscos que se ve forzados a realizar, mientras que valores a partir del metro el controlador suaviza tanto la trayectoria llegando al extremo que podría producirse el choque con objetos.

Al igual que ocurre con los parámetro de PRM el valor de LookaheadDistace va a depender en gran medida del entorno y la longitud y curvatura de la trayectoria siendo en este caso de estudio concreto el intervalo de valores optimos entre 0.3-0.6 metros.

Filtro de partículas

Con un valor de LookaheadDistance de 0.6 y mismos valores velocidades que la sección anterior se ejecutó el algoritmo PurePursuit añadiendo un filtro de partículas.

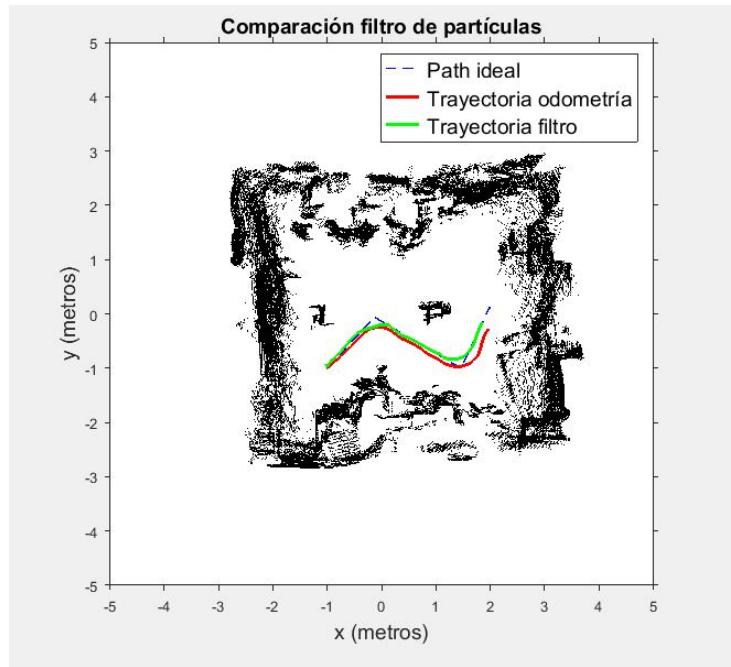


Figura 6.7: Trayectoria real con filtro de partículas

La Figura 6.7 muestra que el robot sigue con mayor precisión la trayectoria impuesta. Se observan dos etapas bien diferenciadas, una primera donde la trayectoria real y la estimada por el filtro son casi idénticas, y una segunda donde difieren. Esto se debe a la convergencia del filtro.

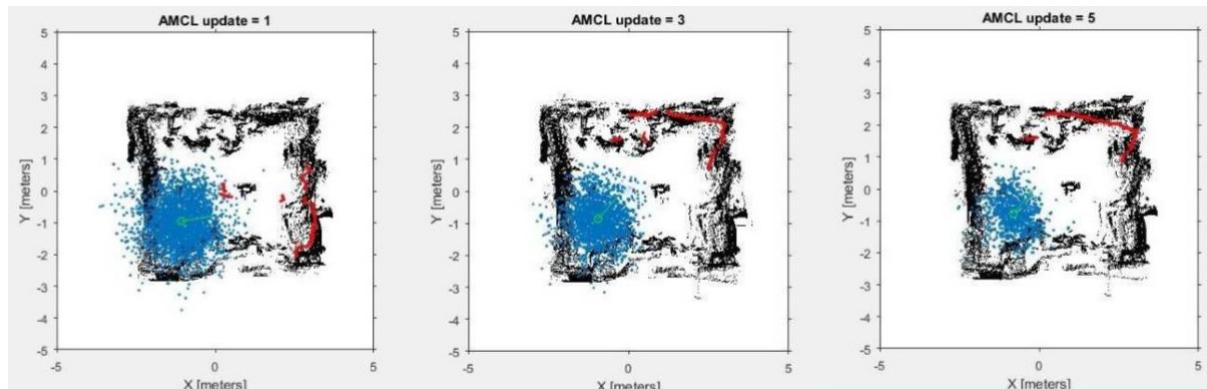


Figura 6.8: Convergencia de filtro de partículas (1)

Para entender estos cambio se debe observar la secuencia de convergencia del filtro. En la Figura 6.8 se observa que la nube de partículas se inicia con el tamaño máximo de componentes y se reduce paulatinamente, cabe destacar que en estas etapas aunque la convergencia no es completa si que concuerda las imágenes del láser simulado con el real (lineas rojas) por lo que el robot si que sigue la trayectoria esperada, con un error mínimo.

Ya en la etapa 11 (Figura 6.9 , imagen de la derecha) aunque el número de partículas ha disminuido considerablemente se observa que siguen una distribución poco uniforme y que los datos del láser no concuerdan con los del mapa, es en este momento en el cual los errores de odometría son lo suficientemente grandes como para que se note la corrección de trayectoria del filtro.

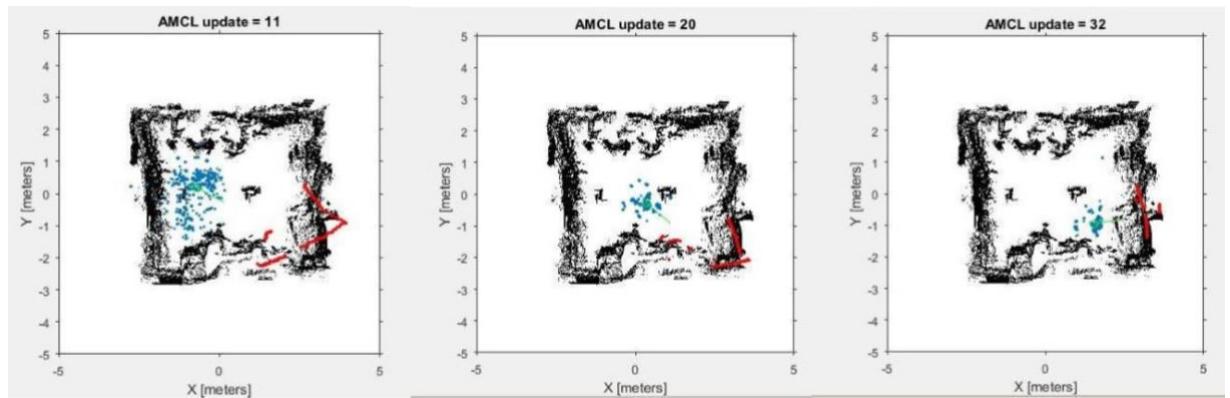


Figura 6.9: Convergencia de filtro de partículas (2)

Finalmente, se observa en la Figura 6.10 que en los puntos finales de la trayectoria se que se produce la convergencia del filtro.

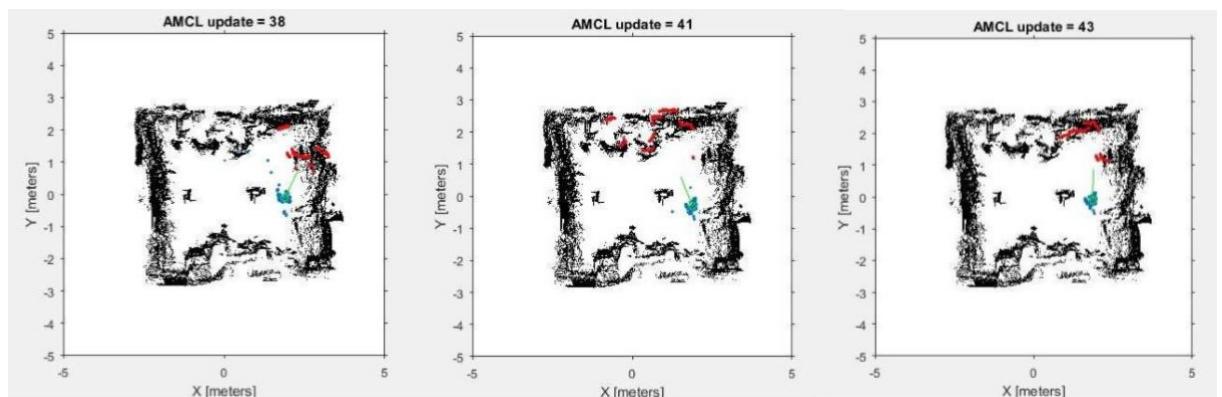


Figura 6.10: Convergencia de filtro de partículas (3)

Destacar que la diferencia entre no usar filtro de partículas o si reside únicamente en que el punto objetivo se alcanza con mayor exactitud en el segundo caso, pero que al ser un entorno reducido la tardanza en converger del filtro puede producir comportamientos inesperados y colisiones.

Posición errónea en el filtro de partículas

El filtro de partículas no solo tiene importancia en casos de corregir la incertidumbre proveniente de los sensores, sino la de reconocer posiciones erróneas de robot.

En este caso se le impone al robot realizar la misma trayectoria que en los casos anteriores pero se cambia la posición inicial.

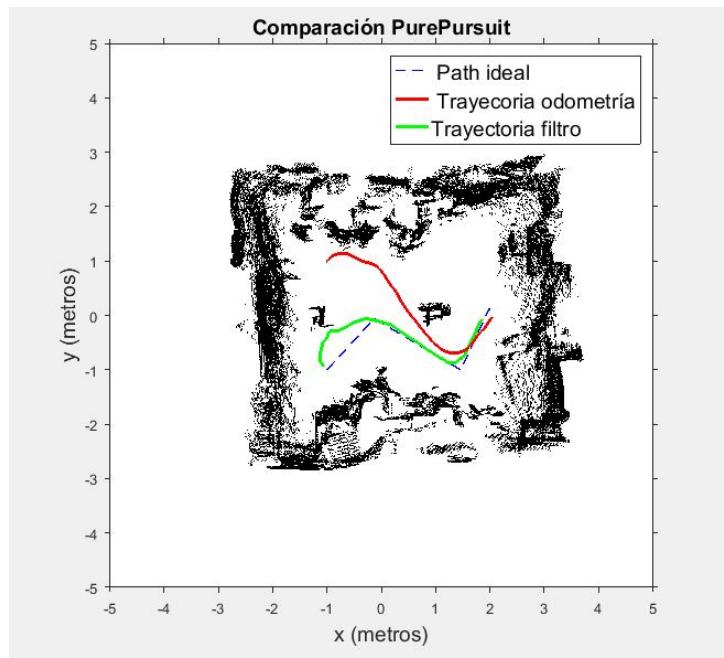


Figura 6.11: Inicio de la trayectoria equivocado

En la Figura 6.11 se observa en gran medida esta corrección en línea roja se encuentra los datos de la odometría y en verde la posición que el filtro estima que tiene el robot.

Hay que tener en cuenta que el algoritmo PurePursuit trabaja con el Path ideal, por lo tanto lo que permite que el robot realmente alcance el punto deseado es el cambio de la posición instantánea del robot debido al filtro de partículas.

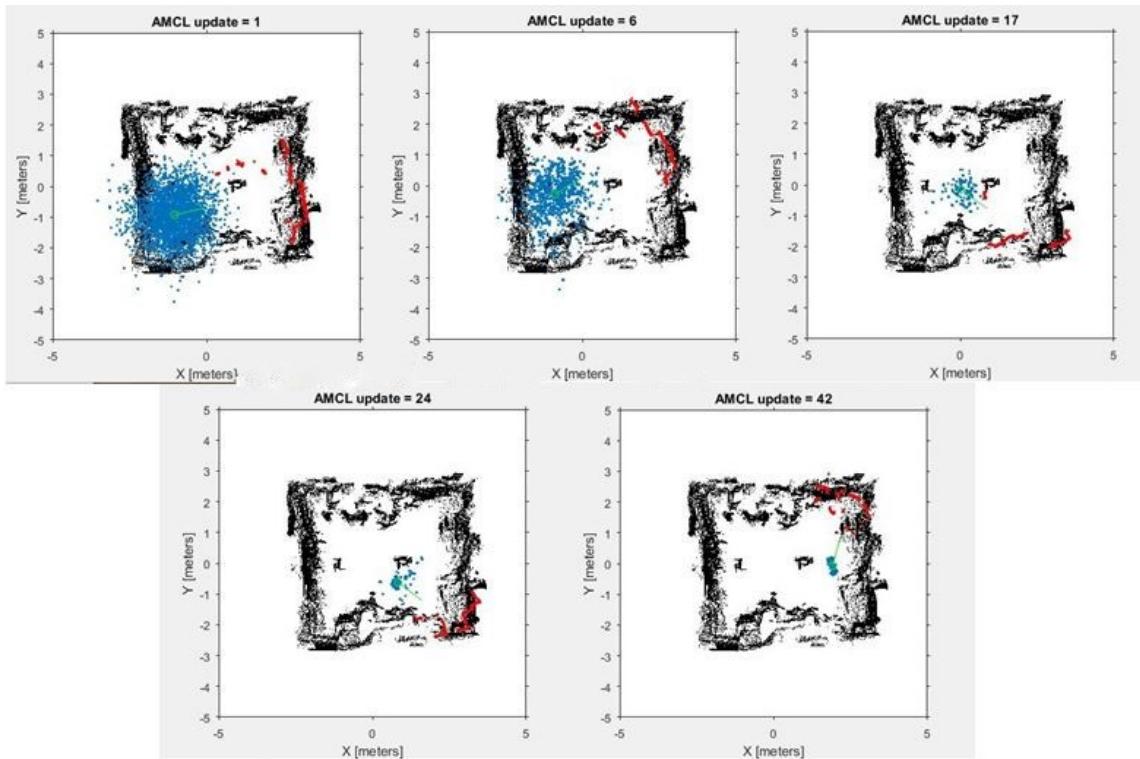


Figura 6.12: Inicio de la trayectoria equivocado

Se adjunta la convergencia del filtro, Figura 6.12, durante la trayectoria. En el se aprecia claramente como la posición inicial no es la real del robot y como a lo largo del tiempo corrige esta información.

Obstáculo en la trayectoria

Otros de los casos estudiados es el cual el robot se encuentra con un objeto no incluido en el mapa inicial. Como se observa (Figura 6.13a), el robot inicia el movimiento como en los casos anteriores hasta que a mitad del mapa se encuentra con un nuevo obstáculo. Lo que realiza es una parada, cálculo de la nueva trayectoria libre de colisiones y posterior alineación con esta.

Realizar estas tres acciones conlleva un tiempo considerable, si la ejecución normal del código equivale a 0.048 segundos de media entre pasada del bucle de control, desde que se reconoce el nuevo obstáculo hasta que se reinicia el bucle trascurren aproximadamente 16 segundo, Figura 6.13b.

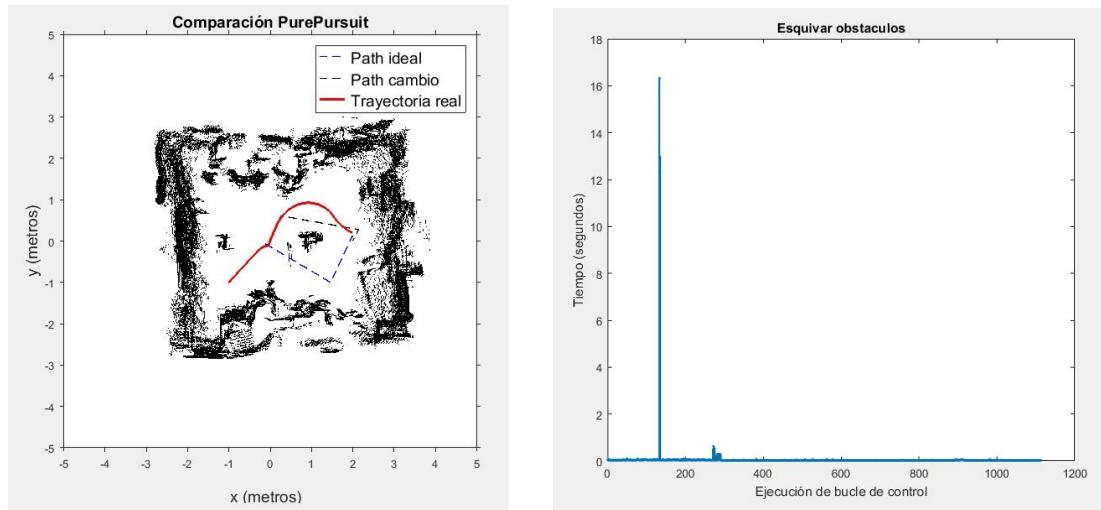


Figura 6.13: Trayectoria con nuevo obstáculo

El recálculo de la trayectoria debido a la detección de un objeto no solo debe a obstáculos no considerados en el mapa, sino a fallos de localización sumados a valores poco restrictivos de la variable a partir de la cual la Kinect considere obstáculo a los objetos del entorno (`distanciaThreshold`, mencionado en 5.3)

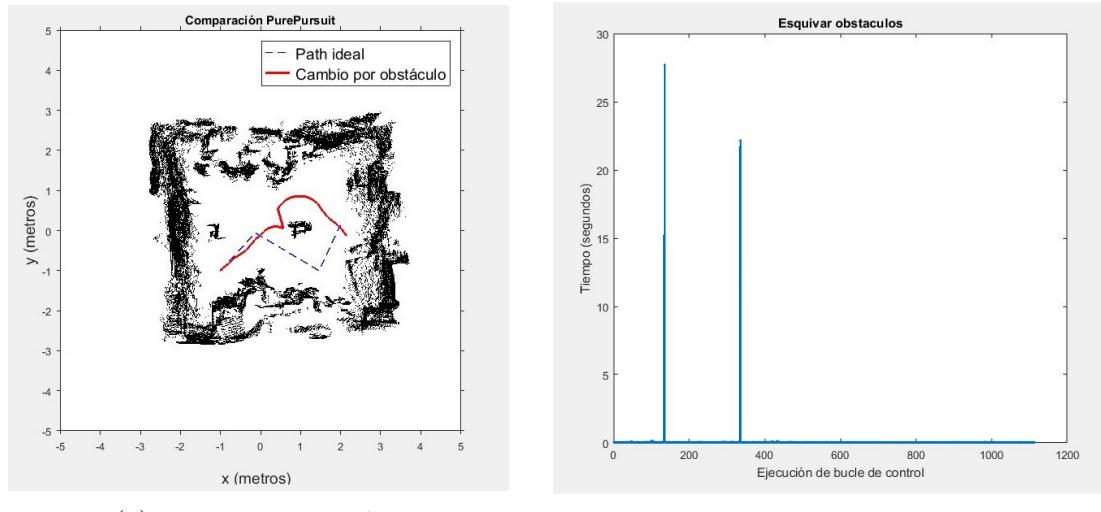


Figura 6.14: Cambio de trayectoria

Como en el caso anterior se vuelve a calcular una nueva trayectoria aunque no haya un nuevo obstáculo si no porque el robot, por errores de localización se ha acercado de forma excesiva a un objeto, Figura 6.14a. Esto sucede dos veces como se aprecia en la Figura 6.13b, debido a que existen dos picos de tiempo.

6.4.2. Fast Marching Square

El último conjunto de experimentos que se realizó fueron con el algoritmo de FM^2 , desarrollado en la UC3M.

En ellos se partía de un mapa en blanco, la principal función que se desea estudiar es localización de obstáculos en la trayectoria. Al contrario que el algoritmo PRM, la ejecución se realiza de forma continua sin interrupciones al detectar objetos y recalcular trayectorias.

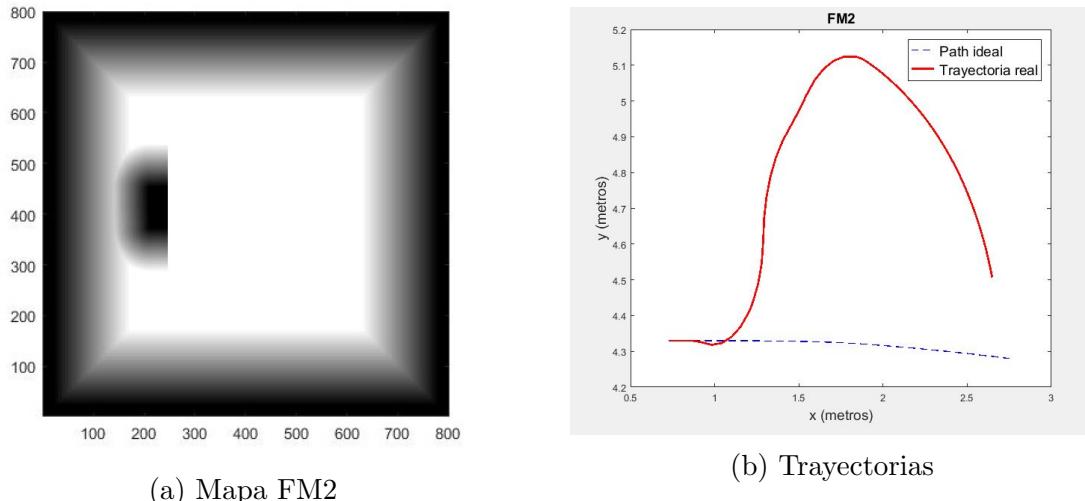


Figura 6.15: Trayectoria FM2 (1)

Se inicia con el reconocimiento de un solo objeto en una trayectoria recta. Se observa, Figura 6.15b, como el algoritmo genera una trayectoria óptima para esquivarlo. Siendo el tiempo de ejecución medio de 0.051 , con un máximo local algo mayor de 0.6 segundo en el momento que reconoce el objeto, Figura 6.16.

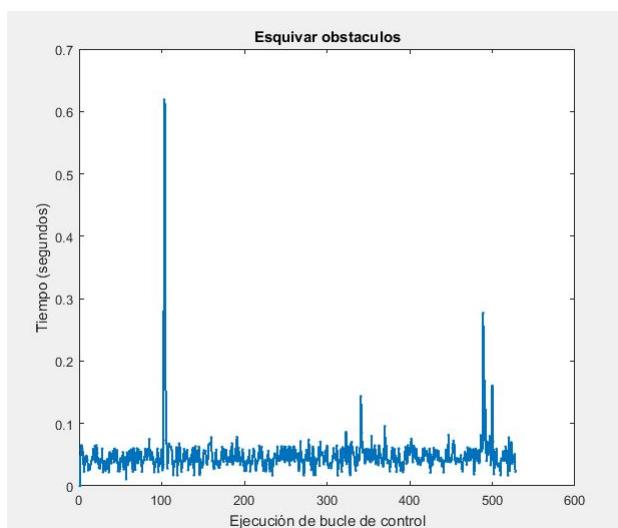
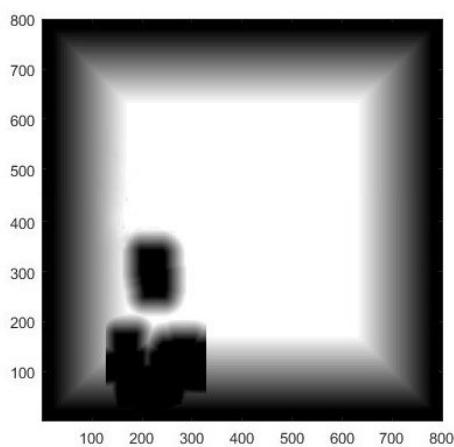
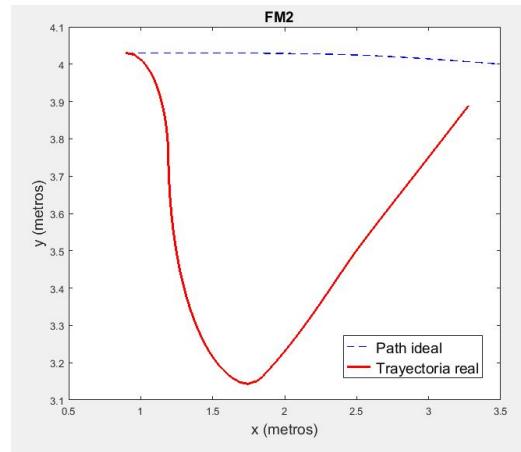


Figura 6.16: Tiempo de ejecución FM2 (1)

Aunque los tiempos de detección son inferiores a los obtenidos con PRM el superar los 0.6 segundo produce como ya se ha mencionado con anterioridad un movimiento intermitente debido a la lentitud al enviar los mensajes de velocidad.



(a) Mapa FM2



(b) Trayectorias

Figura 6.17: Trayectoria FM2 (2)

Situación que se repite en el momento que dos objetos , teniendo dos momentos intermitentes Figura 6.18 con tiempos de ejecución superiores a 0.6 segundos. En la Figura 6.17a se aprecian los distintos objetos que los sensores reconocen durante la trayectoria.

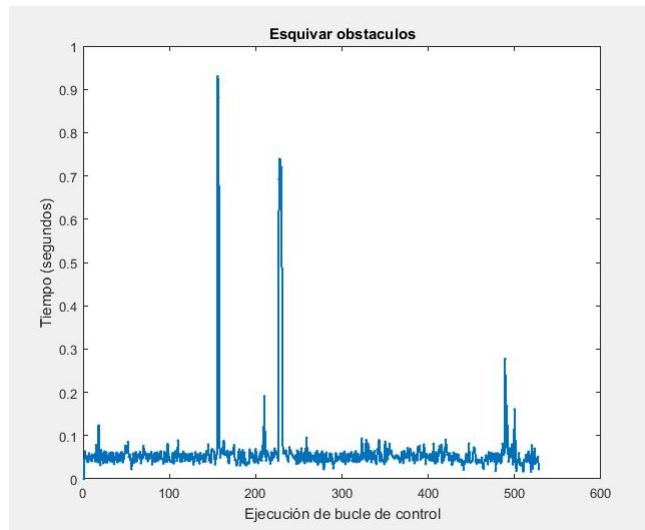


Figura 6.18: Tiempo de ejecución FM2 (2)

En el momento en que el algoritmo, Figura 6.17a, debe detectar más de dos objetos es cuando el movimiento no es fluido casi en ningún momento, debido a que requiere de un mayor esfuerzo computacional.

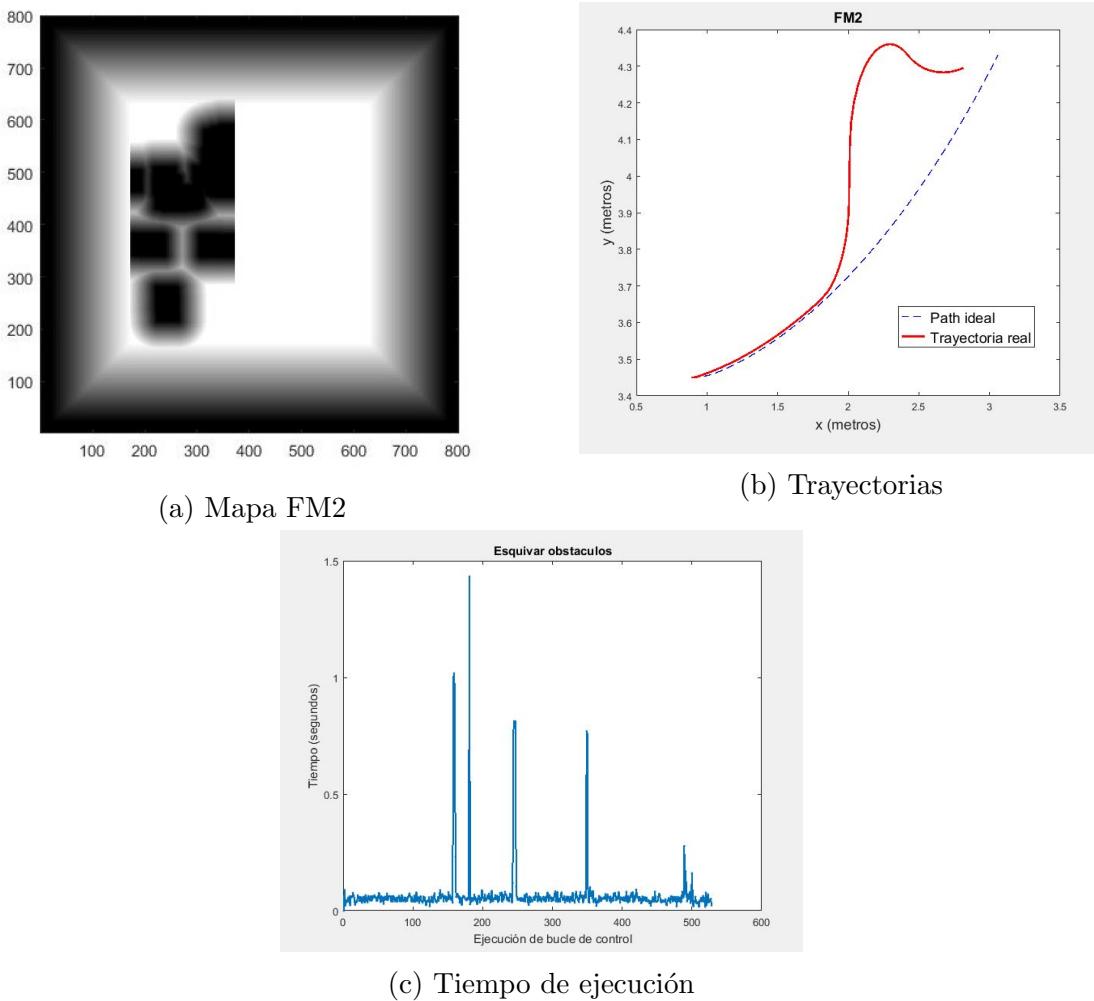


Figura 6.19: Trayectoria FM2 (3)

Capítulo 7

Impacto socio económico y presupuesto

En el capítulo se desarrollará un pequeño análisis del trabajo desde un punto socio económico.

7.1. Impacto socio económico

El sector industrial ha sido durante años la principal área de utilización de los robots. Un cambio en las necesidades en el sector servicios y determinadas actividades industriales a propiciado la aparición de nuevas plataformas robóticas, dentro de las cuales se encuentran los robots móviles.

En el área doméstico, se destaca la utilización de robot móviles en tarea de limpieza y ayuda a personas con reducida movilidad.

En investigación la robótica móvil tiene una gran utilidad en acción de exploración de entornos remotos y cuya alta peligrosidad impiden el acceso del ser humano, como puede ser áreas contaminadas.

Estos son dos ejemplos de la importancia e utilidad de los robots móviles en la sociedad y de aquí radica los grandes esfuerzos en investigación que se están realizando en este área durante los últimos años.

7.2. Presupuesto

A continuación se presentará un breve estudio de coste de realización del trabajo.

La Tabla 7.1 presenta las distintas fases que llevaron a la consecución del trabajo. Iniciando un una breve recopilación de información del tema para comenzar con un estudio de las principales funciones y herramientas tanto de la *toolbox*

como de *ROS*. Tras esto se comenzó con la investigación y posterior implementación de los algoritmos desarrollados. Finalizando con la redacción de esta memoria.

Fase 1	Recopilación de información	32 horas
Fase 2	Familiarización entorno ROS y toolbox	100 horas
Fase 3	Investigación	112 horas
Fase 4	Implementación de algoritmos	140 horas
Fase 5	Memoria	80 horas

Tabla 7.1: Fases de trabajo

Los tiempos empleados para cada fase además de encontrarse en la tabla 7.1, se han representados en forma de diagrama de Gantt, Figura 7.1.

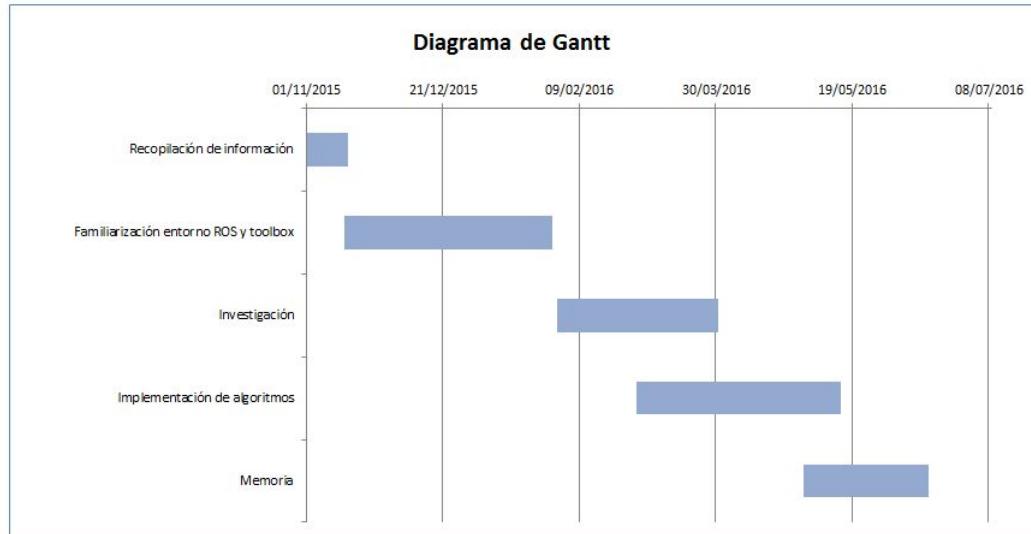


Figura 7.1: Diagrama de Gantt

Los costes de material se reducen a los que engloban al robot ya sea su base, Kinect u ordenador, además de un segundo portátil en el cual se ejecuta Matlab y material de oficina diverso, tabla 7.2.

Concepto	Importe (euros)	Amortización (euros)
Turtlebot	800	80
Kinect	200	20
Portatil 1	1200	120
Portatil 2	800	80
Router	50	5
Material de oficina	20	20
Total	3070	325

Tabla 7.2: Presupuesto de mercancía

Los presupuestos totales se deducen añadiendo el coste de horas trabajadas más los gastos indirectos. Se considera una amortización de materiales de un 10 % y que los gastos indirectos suponen un 15 % del total.

Concepto	Importe (euros)
Gastos materiales	325
Coste por horas trabajadas	9360
Gastos indirectos	1450
Total	11138

Tabla 7.3: Presupuesto de total

Siendo el importe total necesario para la realización de este trabajo aproximadamente 10.847 euros, Tabla 7.3.



Capítulo 8

Conclusiones y trabajos futuros

Para finalizar se resumirán las conclusiones que se desprenden de los datos experimentales, y plantearlos posibles trabajos futuros.

8.1. Conclusiones

Durante la realización de este trabajo se han estudiado las principales funcionalidades de la integración de Matlab con el entorno *ROS*, llegando a la conclusión que esta supone un puente estable de conexión que permite el envío información y simplificando el desarrollo y análisis de algoritmos.

Siendo este unos de los puntos fuertes de la *toolbox*, un único entorno de generación y visualización de datos eliminando el tiempo de conversión de *ROS* a Matlab.

Con la *toolbox* se ha experimentado como mover el robot desde esta plataforma comenzando acciones muy sencillas como puede ser la teleoperación a algoritmos complejos y con mayor autonomía.

Así se realizaron distintos experimentos en el laboratorio en los cuales se ponían a prueba los distintos algoritmos de planificación, localización, mapeo y navegación que son propios de un sistema robótico autónomo. Ejecutándose estas tareas de forma satisfactoria sobre un robot real.

Además de probar programas que hasta el momento se restringían a simulación, que gracias a esta herramienta pueden pasar a un nuevo nivel donde sean ejecutados sobre una plataforma robótica real, como se ha comprobado con el algoritmo de FM2 (sección 5.5.2).

Los problemas que se han presentado durante el trabajo se resumen en :

- Problemas con tiempos de ejecución
- Guardar estructuras de *Robotics System Toolbox*
- Desconexión de *topics /scan*

Haciendo mención al primer problema, ya se ha explicado en el Capítulo 6 que el tiempo de ejecución tiene una influencia importante en diversos aspectos.

La fluidez de movimiento va a depender de la frecuencia con la que se envían los mensajes de velocidad al robot. Así se ha comprobado que esta falta de fluidez se puede dar tanto si se impone una frecuencia de ejecución baja mediante *robotics.Rate*, como retrasos de ejecución debido a algoritmos que requieren un gran esfuerzo computacional pudiendo retrasar el envío de información.

La falta de reacción a eventos externos como puede ser la percepción de un objeto también está influenciada por el tiempo de ejecución. El tiempo que tarda el sistema robótico en reconocer los objetos con los sensores más el tiempo que el algoritmo tarde de procesar esa información son conceptos claves en casos de navegación automática. Siendo así que en los casos los algoritmos tienen un tiempo de ejecución mayor se ha debido de aumentar la distancia mínima con la que el robot debe percibir objetos para aumentar el tiempo de reacción.

Robotics System Toolbox no permite guardar ciertas estructuras de datos propias, por lo que para poder acceder a esta información para el posterior análisis se deben almacenar los datos en matriz que se actualizan dentro del bucle de control.

El último de los problemas hace referencia a la conexión a la Kinect, se ha comprobado que a la hora de inicializar la conexión ROS-Matlab el *topic /scan* presenta más dificultades que el resto.

A través de este trabajo se ha comprobado que *Robotics System Toolbox* es una opción viable para la implementación de programas autónomos para robots móviles. La cual es mucho más cercana a la gran mayoría de estudiantes e ingenieros que no tienen conocimiento en Linux ni ROS, pero que además aporta herramientas interesante para la investigación.

8.2. Trabajos futuros

Las posibles ramas de investigación para futuros trabajos se centrarían en eliminar los retrasos en tiempo de ejecución e introducir posibles mejoras.

Además de la implementación de nuevos programas, ya sea utilizados en otras plataformas o desarrollados íntegramente en la toolbox, y prueba de ellos en entornos más grandes. Los datos experimentales que se han sido tratados fueron únicamente tomados en el ambiente del laboratorio. Sería necesario comprobar la validez de ellos en ambientes reales donde la información del entorno es mucho mayor y más compleja.

Un punto importante de la toolbox del cual no se ha mencionado en este trabajo es la creación de nuevos nodos a partir de Simulink, área que queda por investigar y que puede tener una gran importancia en la creación de nuevos algoritmos. Los

nuevos nodos se crearían en Simulink donde se les atribuiría todas sus características y dependencias para ser posteriormente incluidos en el entorno ROS del robot.

Además Matlab posee potentes herramientas de procesamiento y reconocimiento que podrían ser muy útiles en algoritmos de percepción y reconocimiento de objetos y personas.



Bibliografía

- [1] Anibal OlleroBaturone. *Robótica: manipuladores y robots móviles*. Barcelona, 2005.
- [2] Bruno Siciliano. *Handbook of Robotics*. Springer, 2008
- [3] Christopher Crick, Graylin Jay, Sarah Osentosiki, Benjamin Pitzer y Odest Chadwicke Jenkins. *Rosbridge: ROS for Non-ROS User*. 2011
- [4] CyrillStachniss *Robotic Mapping and Exploration* Springer Tracts in Advance Robotics , Volume 55. Springer, 2009.
- [5] Frank Dellaert, Dieter Fox, Wolfram Burgardz y Sebastian Thruny *Monte Carlo Localization for Mobile Robots* ELsevier , 2000.
- [6] Gazebo, <http://gazebosim.org/>
- [7] Gómez, C., Hernández, A. C., J. Crespo and Barber, R. *Integration of Multiple Events in a Topological Autonomous Navigation System*. IEEE, International Conference on Autonomous Robot Systems and Competitions (ICARSC), 16th edition. 2016.
- [8] Guy Campion, Georges Bastin, and Brigitte D' AndrCa-Novel *Structural Properties and Classification of Kinematic and Dynamic Models of Wheeled Mobile Robots*. IEEE Transactions on robotics and automation, Vol.12, No 1, Febrero 1999.
- [9] Hernández, A. C., Gómez, C., Crespo, J. and Barber, R. *Object Classification in Natural Environments for Mobile Robot Navigation*. IEEE, International Conference on Autonomous Robot Systems and Competitions (ICARSC), 16th edition. 2016.
- [10] Jose Pardeiro Blanco. Trabajo de Fin de Máster: *Algoritmos de planificación de trayectoria basado en Fast Marching Square*. Máster en Robótica y Automatización.2014.
- [11] J.V. Gomez, S. Garrido, L. Moreno. *The Path to Efficiency: Fast Marching Method for Safer, More Efficient Mobile Robot Trajectories*. IEEE Robotics and Automation Magazine. No. 4, vol. 20, paginas: 111 - 120, 2013.
- [12] Li Gang, Jingfang Wang. *PRM path planning optimization algorithm research* 2016

- [13] Lydia E. Kavraki, Mihail N. Kolountzakis, and Jean-Claude Latombe *Analysis of Probabilistic Roadmap for Path Palnning* IEEE Transactions on robotics and automation, Vol.14, No 1, Febrero 1998.
- [14] Matlab: *Robotics System Toolbox*, <http://mathworks.com/help/robotics/index.html>
- [15] M. A. Rodriguez.Conejo, J. Melendez, R. Barber, S. Garrido. *Design of an Infrared Imaging System for Robotic Inspection of Gas Leaks in Industrial Environments*. International Journal of Advanced Robotic Systems. No. 23, Vol. 12, 2015.
- [16] Morgan Quigley, Brian Gerkey, Ken Conley, Josh Faust, Tully Foote, Jeremy Leibs, Eric Berger, Rob Wheeler y Andrew Ng. *ROS: an open-source Robot Operating System*. ICRA Workshop on Open Source Software, 2009.
- [17] Peter Corke. *Integrating ROS and MATLAB*. IEEE Robotics & Automation Magazine. Junio, 2015.
- [18] *Robotics System Toolbox. Reference*. MathWorks, 2015.
- [19] *Robotics System Toolbox. User's Guide*. MathWorks, 2015.
- [20] R.Pratrick Goebel *ROS By Examble. A do-it-yourself guide to the Robot Operation System*, volume 1. 2015
- [21] R. Craig Conlter.*Implementation of the Pure Pursuit Path Tracking Algorithm*.1992
- [22] Roland Siegwart, Illah R. Nourbakhsh, and Davide Scaramuzza. *Introduction to Autonomous Mobile Robots*, second edition. The Mit Press, 1993.
- [23] ROS, <http://www.ros.org/>
- [24] Sebastian Thrun *Particle Filters in Robotics*.2002
- [25] Sebastian Thrun. *Robotic Mapping : A Survey*. Pittsburgh , febrero 2002.
- [26] Turtlebot: Turtlebot2, <http://www.turtlebot.com/>.
- [27] Vicent Girbés, Leopoldo Armesto, Josep Tornero, and J. Ernesto Solanes. *Smooth Kinematic Controller vs. Pure-Pursuit for Non-holonomic Vehicles*
- [28] Willow Garage <https://www.willowgarage.com/>.

Anexos

Anexos A

Algoritmos

A.1. Teleoperación

```
1 function teleoperacion (velmsg,vel,laser,odom,imsub,map)
2
3 %%%%%%% Inicializacion de varilables %%%%%%
4
5 ventanateclado = teclas();
6 teclado = 0;
7 velavance= 0.15;
8 velgiro= 0.4;
9 avance = 0;
10 giro = 0;
11 camara= 0;
12
13 desireRate=10;
14 rate=robotics.Rate(desireRate);
15 rate.OverrunAction='drop';
16
17 %%%%%% Bucle de control %%%%%%
18
19 while teclado ~= 'q'
20
21 reset(rate)
22 while rate.TotalElapsedTime < 1
23
24 SLAMmapa(laser,odom,map);
25 send(vel,velmsg);
26
27 teclado = getKeystroke(ventanateclado);
28
29 switch reply
30
31 case 'w'
```



```
34     if avance<0
35         avance = 0;
36     end
37     avance = acelerarm(velavance,avance,5);
38     giro = 0;
39
40     case 's'
41
42         if avance>0
43             avance = 0;
44         end
45         avance = decelerarm(-velavance,avance,5);
46         giro = 0;
47
48     case 'a'
49
50         if giro<0
51             giro = 0;
52         end
53         giro = acelerarm(velgiro,giro,5);
54         avance = 0;
55
56     case 'd'
57
58         if giro>0
59             giro = 0;
60         end
61         giro = decelerarm(-velgiro,giro,5);
62         avance = 0;
63
64     case 'u'
65
66         velavance = velavance+0.05;
67         disp('Velocidad linear =');
68         disp(velavance);
69
70     case 'j'
71
72         velavance= velavance-0.05;
73         disp('Velocidad linear =');
74         disp(velavance);
75
76     case 'i'
77
78         velgiro= velgiro+0.1;
79         disp('Velocidad angular =');
80         disp(velgiro);
81
82     case 'k'
```



```
84     velgiro= velgiro-0.1;
85     disp('Velocidad angular =');
86     disp(velgiro);
87
88     case 't'
89
90         if camara==2
91             camara = 3;
92         else
93             camara = 1;
94         end
95
96     case 'v'
97
98         if camara==1
99             camara = 3;
100        else
101            camara = 2;
102        end
103
104    case 'b'
105
106        if camara==3
107            camara = 1;
108        else
109            camara = 0;
110        end
111
112    case 'y'
113
114        if camara==3
115            camara = 2;
116        else
117            camara = 0;
118        end
119
120    end
121
122    switch camara
123
124        case 0
125
126            close(figure(2))
127
128        case 1
129
130            figure(2);
131            img = receive(imsub);
132            subplot(1,1,1)
133            imshow(readImage(img));
```

```
134
135     case 2
136
137         figure(2);
138         scan = receive(laser,3);
139         subplot(1,1,1)
140         plot(scan);
141
142     case 3
143
144         figure(2);
145         scan = receive(laser,3);
146         subplot(2,1,2)
147         plot(scan);
148         img = receive(imsub);
149         subplot(2,1,1)
150         imshow(readImage(img));
151
152 end
153
154 velmsg.Linear.X = avance;
155 velmsg.Angular.Z = giro;
156 waitfor(rate);
157
158 end
159
160 end
161
162 velmsg.Linear.X=0;
163 velmsg.Angular.Z=0;
164 closeFigure(ventanateclado);
165
166 end
```

A.2. Teclas

```

1 classdef teclas < handle
2 properties
3   Figure = [];
4   Axes = [];
5 end
6
7 methods
8
9   function obj = teclas()
10
11   callstr = 'set(gcbf, ''Userdata'', double(get(gcbf, ''
12     Currentcharacter''))); uiresume';
13
14   obj.Figure = figure...
15     'Name', 'Teleoperacion', ...
16     'KeyPressFcn', callstr, ...
17     'Position', [500 500 600 300], ...
18     'UserData', 'Timeout');
19
20   obj.Axes = axes('Color', 'k', 'Visible', 'Off', 'XLim'
21     ,[0,100], 'YLim', [0,100]);
22
23   text(10,100, 'Pulsar tecla para teleoperar',...
24     'HorizontalAlignment', 'center', 'EdgeColor', 'b');
25
26   text(10,80, 'w = Avance', 'HorizontalAlignment', 'center',...
27     'EdgeColor', 'k');
28
29   text(10,40, 's = Retroceso', 'HorizontalAlignment', 'center',...
30     'EdgeColor', 'k');
31
32   text(10,60, 'd = Izquierda', 'HorizontalAlignment', 'center',...
33     'EdgeColor', 'k');
34
35   text(10,20, 'a = Derecha', 'HorizontalAlignment', 'center',...
36     'EdgeColor', 'k');
37
38   text(50,100, 'Control de velocidad', 'HorizontalAlignment',...
39     'center', 'EdgeColor', 'b');
40
41   text(50,80, 'u = Incremento vel. linear',...
42     'HorizontalAlignment', 'center', 'EdgeColor', 'k');
43
44   text(50,40, 'j = Decremento vel. linear',...
45     'HorizontalAlignment', 'center', 'EdgeColor', 'k');
46
47   text(50,60, 'i = Incremento vel. angular',...
48     'HorizontalAlignment', 'center', 'EdgeColor', 'k');

```

```
37
38     text(50,20,'k = Decremento vel. angular','
39           HorizontalAlignment','center','EdgeColor','k');
40
41     text(50,0,'q =      Quit','HorizontalAlignment','center'
42           ,,'EdgeColor','r');
43
44     text(90,100,'Camara ','HorizontalAlignment','center',
45           'EdgeColor','b');
46
47     text(90,85,'t = Activar','HorizontalAlignment','center',
48           'EdgeColor','k');
49
50     text(90,70,'y = Desactivar','HorizontalAlignment','center'
51           ,,'EdgeColor','k');
52
53     text(90,50,'Scan','HorizontalAlignment','center',
54           'EdgeColor','b');
55
56     text(90,35,'v = Activar','HorizontalAlignment','center',
57           'EdgeColor','k');
58
59     text(90,20,'b = Descativar','HorizontalAlignment','center'
60           ,,'EdgeColor','k');
61
62 end
63
64 function keyout = getKeystroke(obj)
65 try
66 figure(obj.Figure);
67 uiwait;
68 keyout = get(obj.Figure,'Userdata') ;
69 catch
70     keyout = 'q';
71 end
72
73 end
74
75 function closeFigure(obj)
76 try
77 figure(obj.Figure);
78 close(obj.Figure);
79 catch
80 end
81
82 end
83
84 end
85
86 end
87
88 end
```

A.3. Programa autónomo

```

1
2
3 while( distanceToGoal > goalRadius )
4
5 %%%%%%%% Inicializacion de varilables %%%%%%%%
6
7   goalRadius = 0.20;
8   distanciaThreshold = 0.6;
9   alineado=1;
10  i = 0;
11  hecho=0;
12  mapaglobal=map8;
13  maplocal=map8;
14
15
16
17 %%%%%%%% Actualizacion de sensores %%%%%%%
18
19   odomdata = receive(odom,3);
20   pose=odompose(odomdata);
21   pose=pose+[path1(1,1) path1(1,2) 0];
22   theta= rad2deg(pose(3));
23
24   scan = receive(laser);
25   data = readCartesian(scan);
26   x = data(:,1);
27   y = data(:,2);
28   dist = sqrt(x.^2 + y.^2);
29   minDist = min(dist);
30
31
32 %%%%%%%% Filtro de particulas %%%%%%%
33
34   ranges = double(scan.Ranges);
35   angles = double(scan.readScanAngles);
36
37   [isUpdated,estimatedPose, estimatedCovariance] = step(amcl,
38             pose, ranges, angles);
39   if isUpdated
40     i = i + 1;
41     plotStep(visualizationHelper, amcl, estimatedPose, ranges
42               , angles, i)
43   end
44
45 %%%%%%%% Alineacion con el primer punto %%%%%%

```

```
46
47 if path1(2,2)-pose(2)>0 && path1(2,1)-pose(1)<=0
48   signo=180;
49   angulo=signo-abs(rad2deg(atan((path1(2,2)-pose(2))/(path1
50     (2,1)-pose(1))))) ;
51 elseif path1(2,1)-pose(1)<0 && path1(2,2)-pose(2)<0
52   signo=-180;
53   angulo=signo+abs(rad2deg(atan((path1(2,2)-pose(2))/(path1
54     (2,1)-pose(1))))) ;
55 elseif path1(2,1)-pose(1)>0 && path1(2,2)-pose(2)<=0
56   signo=0;  angulo=signo+rad2deg(atan((path1(2,2)-pose(2))/(path1
57     (2,1)-pose(1))))) ;
58 elseif path1(2,1)-pose(1)>=0 && path1(2,2)-pose(2)>-0
59   signo=0;
60   angulo=signo+rad2deg(atan((path1(2,2)-pose(2))/(path1(2,1)
61     -pose(1))))) ;
62 end
63
64
65 if abs(angulo-theta)>15 && hecho==0
66
67   alineado=0;
68
69 while alineado==0
70
71   odomdata = receive(odom,3);
72   pose=odompose(odomdata);
73   pose=pose+[path1(1,1) path1(1,2) 0];
74
75   theta= rad2deg(pose(3));
76
77   if (angulo-theta)<0 && abs(angulo-theta)<=180
78
79     if abs(angulo-theta)>0.5
80       giro= -0.2;
81     else
82       alineado=1;
83       giro =0;
84     end
85   else
86     if abs(angulo-theta)>0.5
87       giro =0.2;
88     else
89       alineado=1;
90       giro =0;
91     end
92   end
93
94   velmsgAngular.Z= giro;
95   velmsgLinear.X =0;
```

```

92     send(vel,velmsg);
93
94 end
95
96 initialOrientation = theta;
97 robotCurrentPose = pose;
98 hecho=1;
99
100 else
101     hecho=1;
102 end
103
104
105 %%%%%% Deteccion de obstaculos %%%%%%
106
107
108 if minDist <= distanciaThreshold
109     hecho=0;
110     velmsg.Angular.Z= 0;
111     velmsg.Linear.X =0;
112     send(vel,velmsg);
113
114 %%%%%% Incorporacion al mapa temporal %%%%%%
115
116 data = data * [0 1; -1 0];
117
118 th = pose(3)-pi/2;
119 dataWorld = data*[cos(th) sin(th);-sin(th) cos(th)] ...
120 + repmat(pose(1:2),[numel(data(:,1)),1]);
121
122 setOccupancy(maplocal,dataWorld, 1);
123 pause(1);
124
125 %%%%%% Actulizacion PRM %%%%%%
126
127 robotRadius = 0.2;
128 mapInflated = copy(maplocal);
129 inflate(mapInflated,robotRadius);
130 prmlocal = robotics.PRM;
131 prmlocal.Map = mapInflated;
132 prmlocal.NumNodes = 100;
133 prmlocal.ConnectionDistance = 4;
134 startLocation = [pose(1) pose(2)];
135 endLocation = [path(length(path),1) path(length(path),2)];
136 path2 = findpath(prmlocal, startLocation, endLocation);
137
138 while isempty(path2)
139     prmlocal.NumNodes = prmlocal.NumNodes + 5;
140     update(prmlocal);
141     path2 = findpath(prmlocal, startLocation, endLocation);

```

```
142     end
143
144     robotCurrentLocation = path2(1,:);
145     robotGoal = path2(end,:);
146     initialOrientation = theta;
147     robotCurrentPose = pose;
148     controller = robotics.PurePursuit;
149     controller.Waypoints = path2;
150     controller.DesiredLinearVelocity = 0.2;
151     controller.MaxAngularVelocity = 0.33;
152     controller.LookaheadDistance = 0.27;
153     alineado=0;
154
155 %%%%%%% Alineacion con el primer punto %%%%%%
156
157 if path2(2,2)-pose(2)>0 && path2(2,1)-pose(1)<=0
158     signo=180;
159     angulo=signo-abs(rad2deg(atan((path2(2,2)-pose(2))/(path2
160         (2,1)-pose(1)))));
160 elseif path2(2,1)-pose(1)<0 && path2(2,2)-pose(2)<0
161     signo=-180; angulo=signo+abs(rad2deg(atan((path2(2,2)-
162         pose(2))/(path2(2,1)-pose(1)))));
163 elseif path2(2,1)-pose(1)>0 && path2(2,2)-pose(2)<=0
164     signo=0;
165     angulo=signo+rad2deg(atan((path2(2,2)-pose(2))/(path2(2,1)
166         -pose(1))));
167 elseif path2(2,1)-pose(1)>=0 && path2(2,2)-pose(2)>-0
168     signo=0;
169     angulo=signo+rad2deg(atan((path2(2,2)-pose(2))/(path2(2,1)
170         -pose(1)));
171 end
172
173 if abs(angulo-theta)>15 && hecho==0
174     alineado=0;
175
176 while alineado==0
177
178     odomdata = receive(odom,3);
179     pose=odompose(odomdata);
180     pose=pose+[path1(1,1) path1(1,2) 0];
181
182     theta= rad2deg(pose(3));
183
184     if (angulo-theta)<0 && abs(angulo-theta)<=180
185         if abs(angulo-theta)>0.5
186             giro= -0.2;
187         else
188             alineado=1;
```



```
188         giro =0;
189     end
190 else
191     if abs(angulo-theta)>0.5
192         giro =0.2;
193     else
194         alineado=1;
195         giro =0;
196     end
197 end
198
199     velmsg.Angular.Z= giro;
200     velmsg.Linear.X =0;
201     send(vel,velmsg);
202
203 end
204
205 initialOrientation = theta;
206 robotCurrentPose = pose;
207 hecho=1;
208
209 else
210     hecho=1;
211 end
212 end
213
214 %%%%%%%% Velocidades PurePursuit %%%%%%%%
215
216 [v, omega] = step(controller, pose );
217 velmsg.Angular.Z= omega;
218 velmsg.Linear.X =v;
219
220 send(vel,velmsg);
221
222 distanceToGoal = norm([estimatedPose(1) estimatedPose(2)]
223 - robotGoal);
224 end
```



A.4. FM2

```

1    %%%%%% Inicializacion de varilables %%%%%%
2
3    path1=zeros(50,2);
4    distanciaThreshold = 0.7;
5    p=1;
6
7    l=1;
8    n = 200;
9
10   map_name = 'arquitecto.bmp';
11
12   [Wo, cm] = imread(map_name);
13
14   bw=imread('arquitecto.bmp');
15
16   SE=strel('square',40);
17   bw3 = imdilate(~bw,SE);
18   bw4=~bw3;
19
20   disp('FM2 1st step');
21   tic;
22
23
24   W=FMdist(bw4);
25   toc;
26   Wini=W;
27
28
29   W1= rescale( double(Wo) );
30
31   [start_points,end_points] = pick_start_end_point(Wo);
32
33   options.nb_iter_max = Inf;
34   options.end_points = end_points;
35   options.Tmax = sum(size(W));
36   disp('Performing front propagation.');
37   tic
38   [D,S] = perform_fast_marching_2d(W, start_points, options);
39   toc
40
41   disp('Extracting path.');
42   tic;
43   path = extract_path_2d(D,end_points, options);
44   toc;
45
46   figure(1);
47

```

```

48 plot_path_2d(W1,S,path,start_points,end_points);
49 colormap gray(256);
50
51 mapa=imread('rectangulo','bmp');
52
53 [mapmin,mapmax]=lim_mapa(mapa,0);
54
55 posicion=zeros(1,3);
56 posicion(1:2)=start_points;
57
58 [fil,col]=size(path);
59 path_real=zeros(fil,5);
60 t_real=0;
61 speed=0;
62 final_path=0;
63
64 %%%%%%%% Controlador PurePursuit %%%%%%%%
65
66 controller = robotics.PurePursuit
67 release(controller)
68 controller.DesiredLinearVelocity = 0.13;
69 controller.MaxAngularVelocity = 0.4;
70 controller.LookaheadDistance = 0.3;
71
72 dimension=size(path);
73 path2=flip(path/100);
74 intervalo=(path2(dimension(1),1)-path2(1,1))/50;
75
76 k=1;
77 q=1;
78
79 while q<50
80 criterio=intervalo*q;
81 if path2(k,1)>=(criterio+path2(1,1))
82 path1(q,1)=path2(k,1);
83 path1(q,2)=path2(k,2);
84 q=q+1;
85 end
86 k=k+1;
87 end
88
89 path1(q,1)=path2(dimension(1),1);
90 path1(q,2)=path2(dimension(1),2);
91
92 posicion_inicial=path1(1,:);
93 odomdata = receive(odom,3);
94 poseodom=odompose(odomdata);
95 posicion=[((posicion_inicial*100)) 0]+[(poseodom(1)*100) (
poseodom(2)*100) radtodeg(poseodom(3))];

```

```

96 pose=[posicion_inicial 0]+[poseodom(1) poseodom(2) (poseodom
97     (3))];
98
99
100 controller.Waypoints = path1;
101 robotCurrentLocation = path1(1,:);
102 robotGoal = path1(end,:);
103
104 robotCurrentPose = [path1(1,:) radtodeg(poseodom(3))];
105
106 goalRadius = 0.25;
107 distanceToGoal = norm(path1(1,:)-robotGoal);
108
109
110 %%%%%%%% Bucle de control %%%%%%%
111
112
113 while( distanceToGoal > goalRadius )
114
115
116     th0=posicion(3);
117
118
119     scan = receive(laser,3);
120     data = readCartesian(scan) ;
121     x = data(:,1);
122     y = data(:,2);
123     dist = sqrt(x.^2 + y.^2);
124     minDist = min(dist);
125
126     odomdata = receive(odom,3);
127     poseodom=odompose(odomdata);
128     posicion=[((posicion_inicial*100)) 0]+[(poseodom(1)*100) (
129         poseodom(2)*100) radtodeg(poseodom(3))];
130     pose=[posicion_inicial 0]+[poseodom(1) poseodom(2) (
131         poseodom(3))];
132     th = poseodom(3)-pi/2;
133
134     if minDist <= distanciaThreshold
135         data=data*[0 1; -1 0];
136         dataWorld = data*[cos(th) sin(th); -sin(th) cos(th)] ...
137         + repmat(pose(1:2), [numel(data(:,1)), 1]);
138         puntos_laser=dataWorld*100;
139
140         Wt=bw;
141         num_sensores=size(dataWorld);
142
143         for index=1:num_sensores(1)
144             if puntos_laser(index,1)<400

```

```

143     ni=round(puntos_laser(index,1));
144     nj=round(puntos_laser(index,2));
145     Wt(nj,ni)=0;
146   end
147 end;
148
149 %%%%%%% Fast marching de zona local %%%%%%
150
151 disp('fast marching de zona local')
152
153 x_c=round(posicion(1));
154 y_c=round(posicion(2));
155 [fils,cols]=size(Wt);
156 if x_c<100, x_c_i=1;x_c_f=200;
157 elseif x_c>cols-100, x_c_i=cols-200; x_c_f=cols;
158 else x_c_i=x_c-100;x_c_f=x_c+100;
159 end
160
161 Wloc=Wt(1:fils,x_c_i:x_c_f);
162
163 SE=strel('square',40);
164 wl3 =imdilate(~Wloc,SE);
165 wl4=~wl3;
166 Wlocal=FMdist(wl4');
167
168 Wtot=Wini;
169 [ifil,icol]=size(Wlocal);
170 ily=0;
171
172 for iy=x_c_i:x_c_f,
173   ily=ily+1;
174   ilx=0;
175   for ix=1:fils,
176     ilx=ilx+1;
177     Wtot(iy,ix)=min(Wtot(iy,ix),Wlocal(ily,ilx));
178   end
179 end
180
181
182 for ix=1:fils,
183   for iy=1:cols,
184     if Wtot(iy,ix)>0.4, Wtot(iy,ix)=0.4;
185     end
186   end
187 end
188
189 figure(8);
190 dibuja_mapanew(Wtot');
191 W=Wtot;
192 bw=Wt;

```



```
193
194
195     W1= rescale( double(Wt') );
196
197
198     start_points=posicion(1:2)';
199
200     options.nb_iter_max = Inf;
201     options.end_points = end_points;
202     options.Tmax = sum(size(W));
203     [D,S] = perform_fast_marching_2d(W, start_points, options)
204     ;
205
206
207     path = extract_path_2d(D,end_points, options);
208
209
210     figure(1);
211     plot_path_2d(W1,S,path,start_points,end_points);
212     colormap gray(256);
213
214
215     [fil,col]=size(path);
216
217     dibuja_robot(posicion);
218
219
220     %%%%%%% Controlador PurePursuit %%%%%%
221
222     dimension=size(path);
223     path2=flip(path/100);
224     intervalo=(path2(dimension(1),1)-path2(1,1))/50;
225
226     k=1;
227     q=1;
228     while q<50
229         criterio=intervalo*q;
230         if path2(k,1)>=(criterio+path2(1,1))
231             path1(q,1)=path2(k,1);
232             path1(q,2)=path2(k,2);
233             q=q+1;
234         end
235         k=k+1;
236     end
237
238     path1(q,1)=path2(dimension(1),1);
239     path1(q,2)=path2(dimension(1),2);
240
241     release(controller)
242     controller.Waypoints = path1;
243     robotCurrentLocation = path1(1,:);
```

```
242     robotGoal = path1(end,:);
243     distanceToGoal = norm(path1(1,:) - robotGoal);
244     robotCurrentPose = [path1(1,:) radtodeg(poseodom(3))];
245
246     pathcambio7.p=path;
247     p=p+1;
248
249 end
250
251 [v, omega] = step(controller, pose);
252 velmsg.Angular.Z= omega;
253 velmsg.Linear.X =v;
254 send(vel,velmsg);
255 distanceToGoal = norm([pose(1) pose(2)] - robotGoal);
256
257 dibuja_robot(posicion);
258
259 end
```