UNIVERSIDAD DE VALLADOLID

ESCUELA DE INGENIERIAS INDUSTRIALES

Grado en Ingeniería Electrónica Industrial y Automática

Desarrollo de un sistema de navegación para un robot social.

Autor:

García Gómez, Miguel

Tutor(es):

Zalama Casanova, Eduardo
Gómez Ramos, Raúl
Departamento de Ingeniería de
Sistemas y Automática

Valladolid, Julio 2022

Agradecimientos

A mis tutores Eduardo y Raúl, por su preocupación y ayuda diaria en la elaboración del proyecto, dedicando su tiempo a resolverme dudas y transmitirme sus conocimientos sobre el tema.

A mis padres y mis abuelos, por formarme como persona y apoyarme en todo momento. Gracias a vosotros he podido llegar hasta aquí.

A mis amigos de la carrera, con los que he compartido durante estos cuatro años alegrías y agobios.

A mis amigos de siempre, mi segunda familia, por ser siempre una vía de escape ante cualquier mal momento.

A Celia, por ser uno de mis pilares fundamentales desde primero de carrera, con su apoyo y respaldo siempre que lo he necesitado.

Resumen

El presente trabajo consiste en la familiarización con la programación en ROS (“Robot Operating System”) y en el desarrollo de un sistema de navegación autónoma para el robot Turtlebot2 mediante la generación de mapas de entorno. Este robot podrá navegar a destinos establecidos de acuerdo a una agenda, ante la llamada del usuario o ante decisiones del sistema de control.

Para la familiarización con ROS, se han desarrollado diversos programas y se ha estudiado el funcionamiento de este entorno con la programación de distintos nodos y el manejo de mensajes y servicios. Además, se ha estudiado a fondo el “stack” de navegación de ROS. También se han estudiado los diferentes componentes del Hardware del robot y su comunicación con ROS.

Palabras clave: ROS, Turtlebot, Topic, Nodos, Odometría.

Abstract

This work consists of familiarisation with programming in ROS ("Robot Operating System") and the development of an autonomous navigation system for the Turtlebot2 robot through the generation of environment maps. This robot will be able to navigate to established destinations according to an agenda, upon the user's call or upon decisions of the control system.

In order to become familiar with ROS, various programmes have been developed and the operation of this environment has been studied with the programming of different nodes and the handling of messages and services. In addition, the ROS navigation stack was studied in depth. The different components of the robot hardware and its communication with ROS have also been studied.

Keywords: ROS, Turtlebot, Topic, Nodes, Odometry

Índice

1. Capítulo 1. Introducción. 1
1.1. Contexto 1
1.2. Motivación 2
1.3. Objetivos 3
1.4. Elementos del proyecto 3
1.5. Estructura de la memoria 5
2. Capítulo 2. Antecedentes 7
2.1. Robótica 7
2.2. Evolución de ROS (Robot Operating System) 12
2.3. La familia TurtleBot 14
3. Capítulo 3. Descripción del sistema 15
3.1. Robot Operating System (ROS) 15
3.1.1. Arquitectura y conceptos 16
3.1.1.1. El sistema de ficheros 16
3.1.1.2. Grafo de procesos 19
3.1.1.3. Comunidad de ROS 23
3.1.2. Herramientas y comandos 23
3.1.2.1. RVIZ 23
3.1.2.2. Simulador Gazebo 24
3.1.2.3. Comandos útiles 25
3.2. TurtleBot2 27
3.2.1. Base móvil 28
3.2.2. PC Controlador 31
3.2.3. Cámara 3D 32
3.2.4. Telémetro láser de escaneo 33
3.2.5. Brazo robótico 34
3.3. Pila de navegación de ROS 35
3.3.1. Odometría (odometry source) 36
3.3.2. Algoritmo de localización adaptativo de Monte Carlo (AMCL) 36
3.3.3. Transformadas de los sensores 37

3.3.4. Funcionamiento de la pila ... 37
3.3.4.1. Mapas de costo ... 37
3.3.4.2. Planificadores ... 38
3.3.4.3. Recovery Behaviors ... 38
3.3.4.4. Parámetros ... 39
3.4. Puesta en marcha del robot ... 39
3.4.1. Comunicación entre ordenadores ... 39
3.4.2. Conectar ROS Máster ... 40
3.4.3. Espacio de trabajo ... 41
3.4.4. Preparar Turtlebot ... 41
3.4.5. Teleoperación por teclado ... 42
3.4.6. Verificar el funcionamiento de la cámara ... 44
3.4.7. Construir un mapa (SLAM) ... 45
3.4.8. Navegación autónoma en un mapa conocido ... 49
3.4.9. Aparcamiento automático ... 49
4. Capítulo 4. Sistema de mapeo y navegación. ... 53
4.1. Mapeo automático. ... 53
4.1.1. Descripción del problema ... 53
4.1.2. Diagrama de flujo del programa ... 53
4.1.3. Explicación del código ... 55
4.1.4. Mejoras para el programa bumper.py ... 59
4.1.5. Paquete “explore_lite” ... 60
4.2. Sistema de navegación según agenda, llamada de usuario o decisiones del sistema de control ... 62
4.2.1. Descripción del problema ... 62
4.2.2. Método de escritura de los destinos en el fichero ... 62
4.2.2.1. Explicación del código ... 63
4.2.3. Librería “Schedule” ... 68
4.2.4. Sistema de navegación ... 68
4.2.4.1. Diagrama de flujo ... 69
4.2.4.2. Descripción de funciones ... 73
5. Capítulo 5. Puesta en marcha y resultados. ... 85
5.1. Exploración ... 85

5.1.1. Puesta en marcha bumper.py 85

5.1.2. Resultados bumper.py 86

5.1.3. Puesta en marcha “explore_lite” 86

5.1.4. Resultados “explore_lite” 87

5.2. Sistema de navegación por agenda 88

5.2.1. Puesta en marcha 88

5.2.2. Resultados 90

6. Capítulo 6. Gestión del trabajo. 91

6.1. Planificación del proyecto 91

6.2. Estudio económico 92

6.2.1. Recursos empleados 92

6.2.2. Costes directos 93

6.2.2.1. Coste del personal 93

6.2.2.2. Coste de amortización de equipos y programas 94

6.2.2.3. Costes del material utilizado 95

6.2.2.4. Costes directos totales 96

6.2.3. Costes indirectos 96

6.2.4. Costes totales 97

7. Capítulo 7. Conclusiones y líneas de trabajo futuras. 99

7.1. Conclusiones generales 99

7.2. Conclusiones sobre los objetivos planteados 99

7.3. Propuestas de trabajo futuro 101

8. Bibliografía 103

9. Anexos 107

9.1. Código 107

9.1.1. Scripts 107

9.1.1.1. bumper.py 107

9.1.1.2. clickedpoint.py 109

9.1.1.3. posinicial.py 111

9.1.1.4. sch_pos_with_battery_v4.py 112

9.1.2. Archivos launch 120

9.1.2.1. minimalv2.launch del paquete scheduleposition 120

9.1.2.2. minimal_with_hokuyo.launch 120

9.1.2.3. gmapping_demo_hokuyo.launch....121
9.1.2.4. amcl_demo_hokuyo.launch....122

Índice de ilustraciones

Ilustración 1 Turtlebot2 [3] 4
Ilustración 2 Gazebo [4] 5
Ilustración 3 Robot Elmer/Elsie. [5] 8
Ilustración 4 Robot Unimate. [6] 8
Ilustración 5 Robot IRB6. [7]....9
Ilustración 6 Robot Shakey. [8]....9
Ilustración 7 Clasificación de los robots en función de su entorno de trabajo (Elaboración propia)....10
Ilustración 8 Clasificación de los robots según su función (Elaboración propia)....10
Ilustración 9 Logotipo "Box Turtle"....12
Ilustración 10 Logotipo "ROS Noetic Ninjemys" [11]....13
Ilustración 11 Versiones TurtleBot [12] 14
Ilustración 12 Estructura del sistema de ficheros en ROS....16
Ilustración 13 Estructura de un paquete....17
Ilustración 14 Estructura de una pila 17
Ilustración 15 Ejemplo del manifiesto (package.xml) del paquete turtlebot_bringup....18
Ilustración 16 Conceptos del nivel de grafo de procesos [16]....20
Ilustración 17 Gráfico ROS Máster y nodos....21
Ilustración 18 Comunicación en ROS [17] 22
Ilustración 19 RVIZ 24
Ilustración 20 Simulación del robot Turtlebot en una habitación en Gazebo [18]....25
Ilustración 21 Grafo de comunicación rqt_graph 26
Ilustración 22 Robot TurtleBot2 utilizado 27
Ilustración 23 Base móvil "iClebo Kobuki" [19] 28
Ilustración 24 Parte inferior de la base 29
Ilustración 25 Intel NUC 7I5BNK [21] 31
Ilustración 26 Cámara 3D "Orbbec Astra" [22]....32
Ilustración 27 Telémetro láser de escaneo "Hokuyo URG-04LX" [23] 33
Ilustración 28 Brazo robótico "PhantomX Reactor" [25] 34
Ilustración 29 Flujo de la información en el paquete "navigation" [26] 35
Ilustración 30 Modelo de sistema de odometría de un robot [27] 36
Ilustración 31 Recovery Behaviors [28]....38
Ilustración 32 minimal_with_hokuyo.launch 42
Ilustración 33 Interfaz de teleoperación por teclado del robot 1 43
Ilustración 34 Interfaz de teleoperación por teclado del robot 2 43

Ilustración 35 Visualización de la imagen que manda la cámara en RViz 45
Ilustración 36 gmapping_demo_hokuyo.launch 46
Ilustración 37 Mapa realizado por la cámara 3D 47
Ilustración 38 Mapa realizado por el sensor LiDAR "Hokuyo".......................................47
Ilustración 39 Regiones rayos infrarrojos emitidas por la estación [29] 50
Ilustración 40 Recibidores infrarrojos [29] 50
Ilustración 41 Diagrama de flujo del programa "bumper.py" 54
Ilustración 42 kobuki_msgs/BumperEvent.msg 55
Ilustración 43 Constructor de objetos de la clase bumpernode() del programa bumper.py 55
Ilustración 44 Función callback() del programa bumper.py 56
Ilustración 45 Función navegar() del programa bumper.py 57
Ilustración 46 Función shutdown() del programa bumper.py..58
Ilustración 47 Programa bumper.py 58
Ilustración 48 Robot atascado en una esquina...59
Ilustración 49 Diagrama de flujo del programa "bumper.py" mejorado 60
Ilustración 50 Grafo del paquete "explore_lite" [30]..61
Ilustración 51 Fichero horario con destinos 63
Ilustración 52 Constructor de objetos de la clase Meta() en el programa de escritura en el horario 63
Ilustración 53 main() del programa de escritura en el horario.......................................64
Ilustración 54 Función que calcula el cuaternio que marca la orientación del robot en el punto final 64
Ilustración 55 Sistema de coordenadas TurtleBot 65
Ilustración 56 Sistema de coordenadas Turtlebot respecto sistema de coordenadas global..65
Ilustración 57 Producto escalar 66
Ilustración 58 Función encargada de escribir el horario en el fichero 67
Ilustración 59 Diagrama de flujo para actualizar el horario 70
Ilustración 60 Diagrama de flujo de los estados del Robot 71
Ilustración 61 pub_posini() del programa posinicial.py 73
Ilustración 62 Constructor de la clase SchedulePosition() del programa sch_pos_with_battery_v4 74
Ilustración 63 Función shutdown() del programa sch_pos_with_battery_v4..................75
Ilustración 64 Función LeerFichero() del programa sch_pos_with_battery_v4 76
Ilustración 65 Función ult_modificacion() del programa sch_pos_with_battery_v4 77
Ilustración 66 Función goto() del programa sch_pos_with_battery_v4 77
Ilustración 67 Función planificar() del programa sch_pos_with_battery_v4...................78
Ilustración 68 Función SensorStateCallback() del programa sch_pos_with_battery_v4.py 78
Ilustración 69 Función BackUp_Dockstation() del programa sch_pos_with_battery_v4 .79
Ilustración 70 Primera parte de la función GoToDockStation() del programa sch_pos_with_battery_v4 80

Ilustración 71 Segunda parte de la función GoToDockStation() del programa sch_pos_with_battery_v4 ... 81
Ilustración 72 Función NeedCharge() del programa sch_pos_with_battery_v4 ... 82
Ilustración 73 Función ButtonEventCallBack() del programa sch_pos_with_battery_v4 ... 82
Ilustración 74 Función BorrarCostmaps() del programa sch_pos_with_battery_v4 ... 82
Ilustración 75 Función ReproducirSonido() del programa sch_pos_with_battery_v4 ... 83
Ilustración 76 Cuerpo del programa sch_pos_with_battery_v4 ... 83
Ilustración 77 Terminal programa bumper.py ... 86
Ilustración 78 minimalv2.launch del paquete scheduleposition ... 89
Ilustración 79 Mapa utilizado en el proyecto ... 90
Ilustración 80 Planificación del proyecto ... 91

Índice de ecuaciones

(**1**) ... 66
(**2**) ... 66
(**3**) ... 67
(**4**) ... 67
(5) ... 79

Índice de tablas

Tabla 6.1. Coste anual del personal ... 93
Tabla 6.2. Días efectivos por año ... 93
Tabla 6.3. Amortización del material ... 94
Tabla 6.4. Costes del material utilizado (Hardware) ... 95
Tabla 6.5. Costes indirectos ... 96
Tabla 6.6. Costes totales ... 97

1. Capítulo 1. Introducción.

1.1. Contexto

Se define robot móvil como aquel dispositivo que puede desempeñar una serie de funciones de forma automática y que es capaz de lidiar con situaciones y entornos que cambian con el tiempo. [1]

Desde 1970, cuando el instituto Stanford lanzó a “*Shakey*”, el primer robot móvil capaz de tomar decisiones sobre su entorno, la creación de nuevos robots móviles ha crecido de forma exponencial. [2]

En estos últimos años gracias a las nuevas tecnologías y los avances en el campo de la robótica han provocado el auge de modelos robóticos móviles. Estos dispositivos son eficaces no solo cumpliendo una tarea encomendada sino proporcionando soluciones prácticas en entornos dinámicos y colaborativos. La creación de robots móviles ha sido de gran importancia para facilitar tareas a las personas y para realizar acciones que no habrían sido posibles sin la existencia de este tipo de robótica.

El objetivo final de cualquier robot es proporcionar una solución efectiva al mundo real en diferentes sectores. Tradicionalmente, los robots móviles han sido destinados a operaciones en entornos industriales, para transportar diferentes piezas o herramientas a través de las fábricas o como apoyo a la maquinaria. Sin embargo, a día de hoy los robots móviles también trabajan en ambientes donde hay un contacto con el público general. Por ejemplo, proporcionan información en tiendas, hoteles o incluso ayudan a los agentes de policía a patrullar la ciudad.

En el sector de la salud este tipo de robótica se ha convertido en un elemento esencial; la aplicación de estos robots en hospitales y centros de salud ha sido una gran ventaja a la hora de luchar contra el COVID-19 en estos últimos años. Los robots móviles aplicados al sector de la salud han resultado ser una herramienta excelente que mejora la calidad de vida de los pacientes y proporciona más autonomía a las personas dependientes.

Con la llegada del COVID-19 los centros de salud se han visto desbordados por la gran cantidad de personas ingresadas que necesitaban ser asistidas de forma continua y controlada. Igualmente, debido a la pandemia, muchas personas mayores o personas con movilidad reducida han tenido que permanecer aislados en sus hogares largos periodos de tiempo sin ningún tipo

de asistencia. La ausencia de este apoyo para desplazarse y realizar tareas cotidianas dentro de su casa ha podido empeorar su calidad de vida.

Ante estos problemas, surge la necesidad de diseñar dispositivos móviles que puedan moverse de forma autónoma por espacios diferentes, y así proporcionar servicios a las personas. Este proyecto se centra en el desarrollo de un robot móvil que sea capaz de desplazarse de forma dinámica en un entorno conocido.

Este robot podría convertirse en un apoyo para los auxiliares de enfermería, dando apoyo en hospitales o residencias trasladando medicamentos, comida y otros elementos necesarios a las habitaciones de cada paciente. Igualmente, otra aplicación que se le puede dar es la de asistente doméstico para personas dependientes; así este podría proporcionar los medicamentos necesarios a la persona asistida a unas horas determinadas, por ejemplo.

1.2. Motivación

El impulso de la robótica móvil en los últimos años conlleva a la necesidad de comprender el funcionamiento de estos robots. Los robots móviles, los cuales tienen un crecimiento anual de mercado muy alto, serán muy importantes en un futuro cercano. Siempre me había preguntado cómo funcionarían y quería aprender sobre ello. Viendo el robot aspirador "*Roomba*" en mi casa, no entendía cómo un robot sin cámaras podía reconocer el entorno y volver a su base de carga por sí solo.

Esto hizo que este proyecto me pareciera muy interesante, al tratarse del control de un robot móvil, ya que implicaba aprender sobre cómo funcionan estos robots. Las asignaturas de mi grado se centraban más en robots industriales, por lo que no había visto casi nada de robots móviles, haciendo de este trabajo un reto para mí.

A parte de la motivación que tenía gracias a que iba a aprender cosas nuevas, otro tema importante, y por lo que me decidí por este trabajo, era la ayuda social que iba a realizar.

Con la llegada de la pandemia, como se ha comentado antes, se vio lo importante que es invertir en médicos y enfermeros para cuidar a pacientes. Ante la escasez de trabajadores, el desarrollo de un proyecto como este podría ser de gran ayuda, ya que el robot puede realizar el trabajo que antes requería a un auxiliar hospitalario, como puede ser llevar medicamentos o comida al paciente a una hora determinada.

1.3. Objetivos

El objetivo principal del proyecto es el desarrollo de un sistema de navegación autónomo para el robot Turtlebot2 mediante la generación de mapas de entorno. El robot deberá navegar a destinos establecidos de acuerdo con una determinada agenda, ante la llamada del usuario o ante decisiones del sistema de control. Realizar esto requiere de un aprendizaje del entorno de desarrollo de software de aplicaciones robóticas "*Robot Operating System*" (ROS), con lo cual podemos dividir el proyecto en diversos objetivos intermedios:

- Familiarizarse con la programación en ROS, comprendiendo sus conceptos principales y llevándolos a la práctica.
- Comprender la aplicación de la navegación autónoma de robots de ROS.
- Aprender conceptos sobre el robot Turtlebot2, además de conocer el alcance de posibilidades de funcionamiento que puede otorgar.
- Conseguir que el robot sea capaz de mapear el entorno.
- Desarrollar programas para comprender los movimientos del robot y poder realizar el objetivo final de este proyecto.
- Desarrollar un sistema de planificación o agenda para que el robot pueda dirigirse a ciertas ubicaciones a una hora determinada.

1.4. Elementos del proyecto

A continuación, se describen algunos conceptos y elementos clave utilizados en el desarrollo de este trabajo.

- <u>UBUNTU</u>

  Sistema operativo que sigue la distribución Linux, donde se instalará ROS. Para cada versión de ROS se necesitará una versión de Ubuntu específica.

- <u>ROS</u>

  Entorno de desarrollo utilizado para el desarrollo de aplicaciones robóticas.

  Es de código abierto, lo que quiere decir que el modelo de desarrollo de software está basado en la colaboración abierta. "Software libre", cuyo

código fuente puede ser estudiado, modificado y utilizado libremente con cualquier finalidad.

Proporciona librerías y herramientas para el desarrollo de programas robóticos.

- **Turtlebot2**

Robot personal de código abierto capaz de navegar por el espacio.

Cuenta con una cámara 3D para ver el entorno. (Ilustración 1)

Ilustración 1 Turtlebot2 [3]

- **Base "Kobuki"**

Base móvil del robot Turtlebot2. Cuenta con sensores de choque y detectores de precipicios izquierdos, delanteros y derechos, además de sensores de caída de rueda.

- **Rviz**

Herramienta de visualización de ROS que puede mostrar una gran variedad de información, como puede ser el mapa de la sala, la trayectoria que va a realizar el robot o la imagen que transmite la cámara a tiempo real. También es capaz de mostrar una representación del robot utilizado.

- **Gazebo**

Entorno de simulación de ROS (Ilustración 2). Permite simular comportamientos del robot como si fuera real. Es de gran utilidad a la

hora de probar programas antes de meterlos en el robot real, evitando que pueda romperse debido a algún fallo de programa.

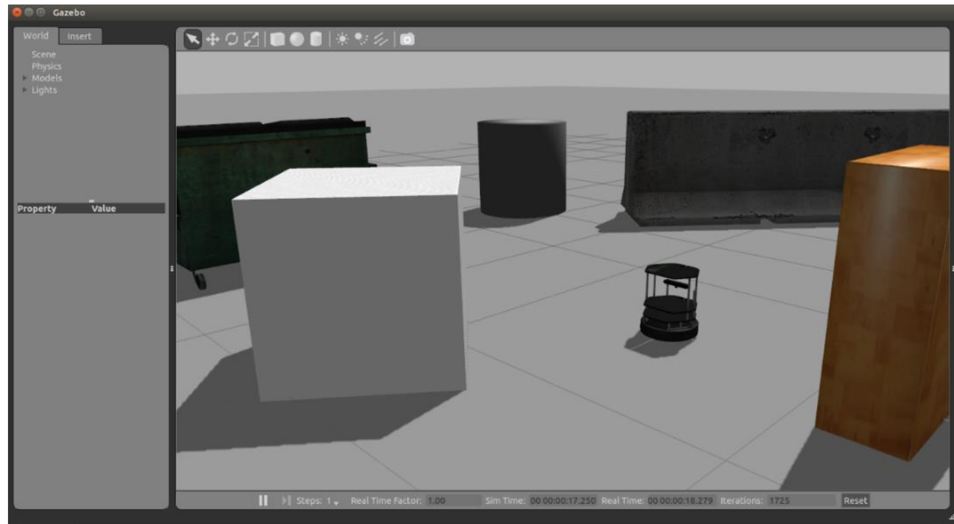

Ilustración 2 Gazebo [4]

1.5. Estructura de la memoria

La memoria se estructura en 8 capítulos:

- Capítulo 1. Introducción

  Es el capítulo actual, sirve para poner un contexto sobre el proyecto y una justificación del mismo, además de sus objetivos y la presentación de los elementos más importantes.

- Capítulo 2. Antecedentes.

  El objetivo de este capítulo es situar en un marco histórico las tecnologías que se van a utilizar en este proyecto.

- Capítulo 3. Descripción del sistema.

  En este capítulo se presenta el robot Turtlebot2 con sus componentes y el entorno de desarrollo “Robot Operating System”, explicando sus conceptos y teoría. Además, se expone la puesta en marcha del robot.

- Capítulo 4. Sistema de mapeo y navegación

  El fin de este capítulo es describir el problema a solucionar y explicar poco a poco el código en que se basa la solución a este asunto, para ayudar a su comprensión.

- Capítulo 5. Puesta en marcha y resultados

  Se expone la puesta en marcha de los programas realizados para el proyecto y muestra un análisis de los resultados obtenidos.

- Capítulo 6. Gestión del trabajo

  Informe de cómo se ha gestionado el tiempo a la hora de realizar el proyecto y el estudio económico de éste.

- Capítulo 7. Conclusiones

  Conclusiones sobre el proyecto en general, sobre los objetivos a realizar y sobre posibles líneas de trabajo futuras.

Para finalizar, se adjunta la bibliografía y anexos.

2. Capítulo 2. Antecedentes.

El objetivo de este capítulo es situar en un marco histórico las tecnologías que se van a utilizar en este trabajo.

2.1. Robótica

La robótica combina diversas disciplinas como la mecánica, electrónica, informática, matemáticas o física.

Antes de existir la palabra robot, se denominaban autómatas a los artefactos construidos con el objetivo de descargar de trabajos tediosos o peligrosos a los humanos.

El término ROBOT se utilizó por primera vez en una obra teatral denominada R.U.R. (Rossum's Universal Robots) del escritor checo Karel Capek en 1920. Robot proviene de la palabra checa “robota”, que significa “esclavo, trabajo forzoso, simplemente trabajar”. En ella, los robots eran máquinas que se parecían al hombre, las cuales trabajaban sin descanso, lo que conlleva a una revolución contra sus creadores.

El científico y escritor de origen ruso Isaac Asimov, conocido por sus obras de ciencia ficción, escribió las “las tres leyes de la robótica” en 1942 y establecen lo siguiente:

1. Un robot no hará daño a un humano ni, por inacción, permitirá que un humano sufra daño.
2. Un robot debe obedecer cualquier orden dada por los humanos, excepto aquellas que entren en conflicto con la primera ley.
3. Un robot debe proteger su propia existencia, siempre y cuando esta protección no entre en conflicto con la primera y la segunda ley.

En la última etapa de los años 40 aparecen los primeros robots móviles, Elmer y Elsie (Ilustración 3), ideados por el neurólogo americano William Grey Walter. Son consideradas como las primeras máquinas robóticas autónomas de la historia. Estas “tortugas robot” (denominados así por su parecido a una tortuga) podían moverse de manera autónoma en función de los focos de luz y esquivando obstáculos. Tenían dos motores, uno para avanzar y otro de giro.

Ilustración 3 Robot Elmer/Elsie. [5]

En la década de los 50 aparece el primer robot industrial comercial, conocido como "Unimate" (Ilustración 4). Un manipulador esférico controlado por interruptores de fin de carrera y levas. Utiliza motores hidráulicos.

Ilustración 4 Robot Unimate. [6]

En la década de los 70 aparece el primer robot del mundo con accionamiento totalmente eléctrico, el robot IRB6 (Ilustración 5) de la empresa sueca ASEA (empresa de la cual acaba surgiendo ABB)

Ilustración 5 Robot IRB6. [7]

En esta época, aparecía el primer robot inteligente de la historia, "Shakey" (Ilustración 6), capaz de percibir su entorno, generar sus rutas y moverse por su cuenta. Podía crear un mapa del entorno y decidir la trayectoria más rápida desde donde se encontraba hasta donde tenía que llegar. Fue el primer paso hacia una robótica con sistemas integrados de inteligencia artificial.

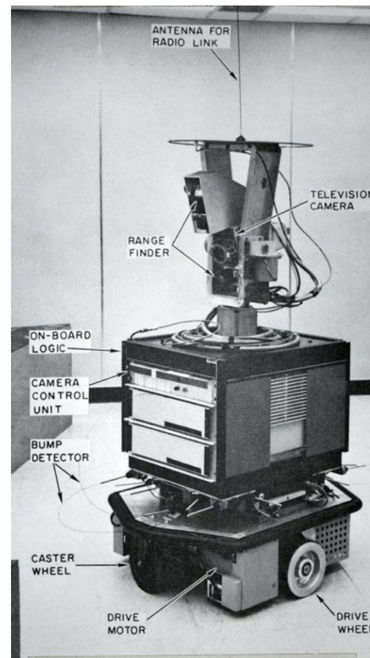

Ilustración 6 Robot Shakey. [8]

En la actualidad, el desarrollo de la robótica ha conseguido que los robots tengan un comportamiento cada vez más inteligente, siendo capaces de integrarse con el entorno y tomar decisiones por sí mismos en tiempo real más rápido de lo que lo haría cualquier humano.

Podemos definir robot como una máquina automática programable capaz de realizar determinadas operaciones de manera autónoma y sustituir a los seres humanos en algunas tareas, en especial las pesadas, repetitivas o peligrosas; puede estar dotada de sensores, que le permiten adaptarse a nuevas situaciones. [9]

Una manera de clasificar los robots es según su entorno de trabajo, donde se diferencia entre robots móviles y robots fijos. (Ilustración 7) Los robots fijos suelen ser industriales, ya que en su mayoría trabajan en una celda solos o cooperando con otros robots. En cambio, los robots móviles, como su propio nombre indica, pueden trasladarse de un lado a otro.

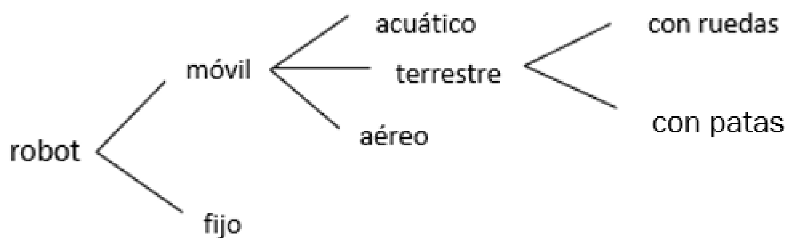

Ilustración 7 Clasificación de los robots en función de su entorno de trabajo (Elaboración propia)

Otra manera de clasificar los robots es según la función que van a realizar, en los que se distinguen dos grupos: industriales y de servicios. (Ilustración 8)

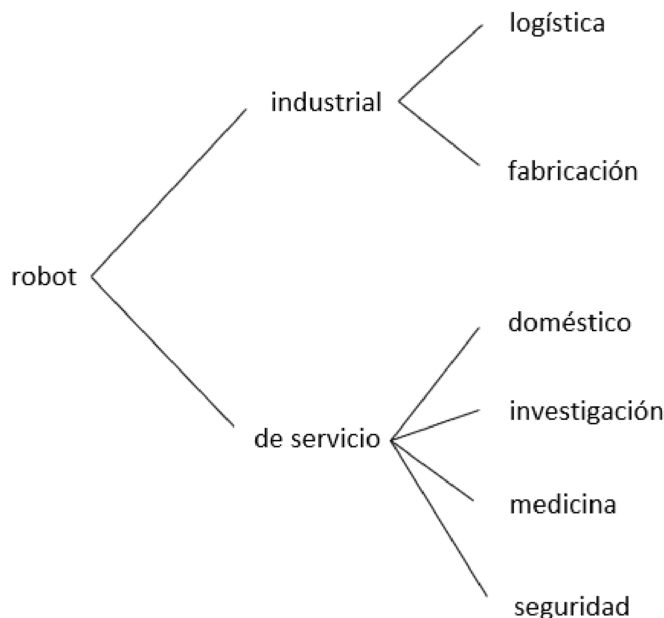

Ilustración 8 Clasificación de los robots según su función (Elaboración propia)

Los robots industriales son aquellos que están destinados a realizar trabajos relacionados con procesos industriales, suelen ser fijos.

Los robots de servicio suelen ser móviles y ayudan al ser humano a realizar trabajos que son repetitivos y tediosos, como pueden ser las tareas domésticas. La Federación Internacional de la Robótica define estos robots como aquel robot que realiza tareas útiles para humanos, excluyendo aplicaciones de automatización industrial. Estos pueden dividirse en diferentes tipos, como doméstico o para investigación, medicina, seguridad, etc. [10]

2.2. Evolución de ROS (Robot Operating System)

ROS (Robot Operating System) a pesar de su nombre, no es un sistema operativo, sino que es un marco de trabajo o "framework" utilizado para el desarrollo de aplicaciones robóticas. Suministra un conjunto de librerías y herramientas para el desarrollo de programas con los que podremos crear de una manera más fácil y rápida proyectos de robótica. Es un 'kit' para el desarrollo de software que ofrece bloques ya hechos necesarios para crear las aplicaciones robóticas.

Es de código abierto, lo que quiere decir que el modelo de desarrollo de software está basado en la colaboración abierta. "Software libre", cuyo código fuente puede ser estudiado, modificado y utilizado libremente con cualquier finalidad. Esta es otra de las ventajas de utilizar ROS en los proyectos, ya que ROS lleva existiendo más de 10 años y está siendo utilizado por millones de desarrolladores y usuarios, los cuales contribuyen a la mejora de este con sus proyectos. Esto hace que se pueda encontrar muchos trabajos de otros desarrolladores en Internet, pudiendo coincidir con lo que se requiera que haga tu robot, agilizando el proceso de pensar y escribir el programa.

Permite desarrollar computación distribuida, es decir, los programas pueden utilizar recursos en varios nodos de cálculo distintos para conseguir un objetivo compartido común.

Su historia comienza en 2007, cuando el Laboratorio de Inteligencia Artificial de Stanford desarrolla ROS con el nombre de 'switchyard' en un inicio, para dar soporte a los proyectos 'STAIR' (STanford Artificial Intelligence Robot) y 'Personal Robotic Program'. A partir de aquí, se sigue desarrollando principalmente en Willow Garage, un laboratorio de investigación robótica dedicado a la creación de software de código abierto para robots personales. Además, diferentes investigadores ajenos a Willow Garage contribuyeron en este desarrollo.

El 2 de marzo de 2010 se lanza la primera versión de ROS con el nombre Box Turtle (Ilustración 9), distribuyendo paquetes para uso público.

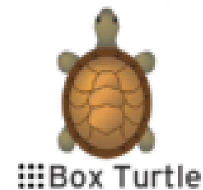

Ilustración 9 Logotipo "Box Turtle"

Meses después saldría la segunda versión ROS 'C Turtle', como una gran actualización de las librerías ya lanzadas en 'Box Turtle'.

En marzo de 2011 salió ROS 'Diamondback', que contendría nuevos paquetes, incluyendo soporte para la cámara Kinect.

Con la siguiente versión sacada en agosto de ese mismo año (ROS 'Electric EMys') aumentaban el soporte hacía nuevas plataformas como Android y Arduino. Además, fue el año del lanzamiento de ROS Answers, foro de preguntas y respuestas de usuarios de ROS.

Tras esto, cada año se iría sacando una nueva versión hasta 2018, incrementando considerablemente el número de usuarios que utilizan ROS. Los nombres de las versiones van en orden alfabético en función del orden de salida. Actualmente la última versión es ROS Noetic Ninjemys (Ilustración 10), siendo la versión final de los que se conoce como ROS1, con fecha de salida el 23 de Mayo de 2020 y con soporte hasta 2025.

Ilustración 10 Logotipo "ROS Noetic Ninjemys" [11]

En el año 2017 surge ROS2 como una mejora de ROS1, con el nombre ROS2 'Ardent Apalone', ofreciendo, por ejemplo, el salto a Windows 10. A partir de aquí, año a año han ido saliendo nuevas versiones, siguiendo la norma del orden alfabético de ROS1. Actualmente la última distribución es ROS2 'Humble Hawksbill' con fecha de salida el 23 de mayo de 2022 y con soporte hasta 2027.

2.3. La familia TurtleBot

Turtlebot es un robot personal de bajo precio y código abierto. Es muy utilizado para la educación y la investigación. Fue desarrollado en Willow Garage, al igual que ROS. Es capaz de desplazarse y de ver el entorno en 3D. En la actualidad han salido a la venta tres versiones diferentes (Ilustración 11)

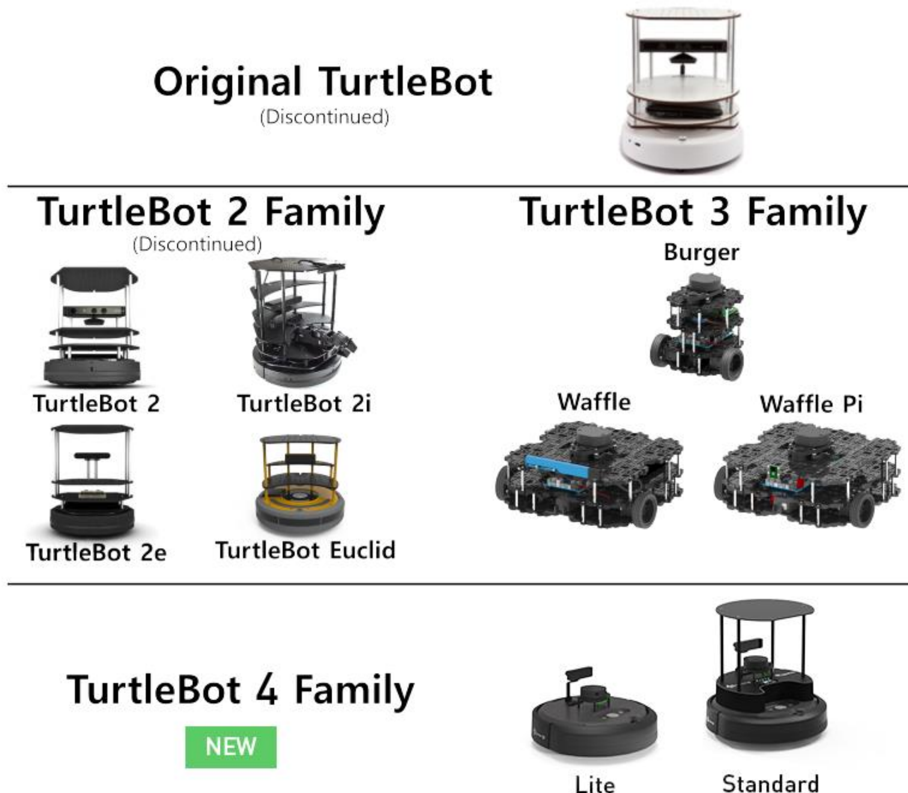

Ilustración 11 Versiones TurtleBot [12]

El TurtleBot1 salió a la venta en noviembre de 2010, con una base móvil denominada "iRobot Create".

Turtlebot2 salió en 2012 y cambió su base a una "Yujin Kobuki".

El lanzamiento del robot Turtlebot3 fue en 2017, hecho por placas modulares, las cuales el usuario puede modificar su forma.

En Julio de 2022 se pondrá a la venta el robot Turtlebot4.

3. Capítulo 3. Descripción del sistema

En este capítulo se introduce, en un marco teórico, las tecnologías utilizadas. Para ello se describirán conceptos sobre el software y el hardware utilizado. Es importante comprender bien cada concepto para poder entender el proyecto realizado. Además, se expone la puesta en marcha del robot.

3.1. Robot Operating System (ROS)

Como se ha mencionado en el capítulo anterior, ROS es un entorno de desarrollo para aplicaciones robóticas de código abierto, el cual suministra librerías y herramientas para la programación de robots.

La descripción de ROS según su página web oficial es: “un metasistema operativo de código abierto para su robot. Proporciona los servicios que esperaría de un sistema operativo, incluida la abstracción de hardware, el control de dispositivos de bajo nivel, la implementación de funciones de uso común, el paso de mensajes entre procesos y la gestión de paquetes. También proporciona herramientas y bibliotecas para obtener, construir, escribir y ejecutar código en varias computadoras”. [13]

Necesita de un sistema operativo base donde instalarlo, ya que no es un sistema operativo propiamente dicho. Actualmente ROS solo corre en plataformas basadas en UNIX. Se instalará sobre Ubuntu, el cual es una distribución de Linux. La versión de Ubuntu requerida dependerá de la versión que se necesite instalar de ROS.

Una de las características de ROS es que permite desarrollar una computación distribuida. Esto quiere decir que los programas pueden utilizar recursos en varios nodos de cálculo distintos para conseguir un objetivo compartido común. Por ejemplo, para la sincronización de dos robots esto es una ventaja, ya que cada robot, con sus nodos de cálculos distintos, puede acceder a la información del otro robot y hacer decisiones con respecto a esa información.

Cada robot puede tener muchos nodos de cálculos distintos (programas), desde el que se encarga de hacer que ande hasta el que detecta si hay obstáculos en los alrededores. A estos programas en la nomenclatura de ROS se les conoce como “Nodos”, de los cuales se hablará más tarde.

3.1.1. Arquitectura y conceptos

Al hablar de los conceptos de ROS, debemos distinguir entre tres apartados o niveles de conceptos. [14]

- El nivel de sistema de ficheros
- El nivel de grafo de procesos
- El nivel de la comunidad de ROS

3.1.1.1. El sistema de ficheros

Los conceptos de este nivel indican la estructura de las carpetas en las que está dividido un programa de ROS con sus archivos necesarios para el funcionamiento del programa. En la ilustración 12 se puede observar un esquema de su organización.

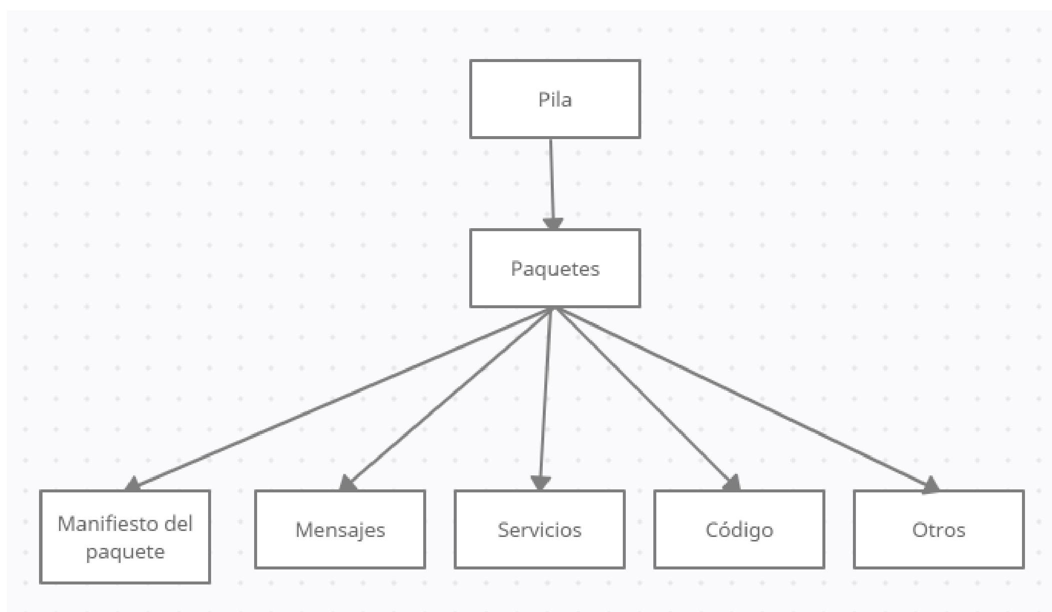

Ilustración 12 Estructura del sistema de ficheros en ROS

3.1.1.1.1. Paquetes (Packages)

Los paquetes son la principal unidad de organización de ROS, contiene la estructura mínima y el contenido para crear un programa en ROS. Todo el software de ROS está organizado en paquetes.

Un paquete es una colección coherente de ficheros (ejecutables y ficheros de soporte) que sirven para un propósito específico.

Estos paquetes son definidos por un manifiesto (fichero package.xml) que da información sobre el paquete, incluyendo su nombre, versión, descripción, información de licencia, dependencias…

El directorio que contiene el paquete se llama "*package directory*" y no puede haber más de un paquete por directorio.

El paquete contiene los procesos de ejecución (nodos), librerías, scripts, archivos de configuración (*Makefiles*), etc. (Ilustración 13)

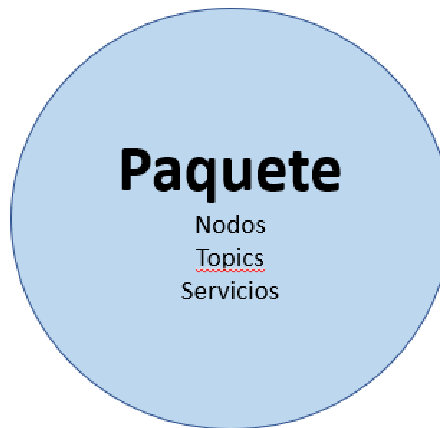

Ilustración 13 Estructura de un paquete

3.1.1.1.2. Pilas (Stack)

Las pilas son una colección de paquetes con funcionalidad relacionada (Ilustración 14). Es parecido a un metapaquete. Una pila muy utilizada en ROS es la "pila de navegación", la cual se presenta más adelante.

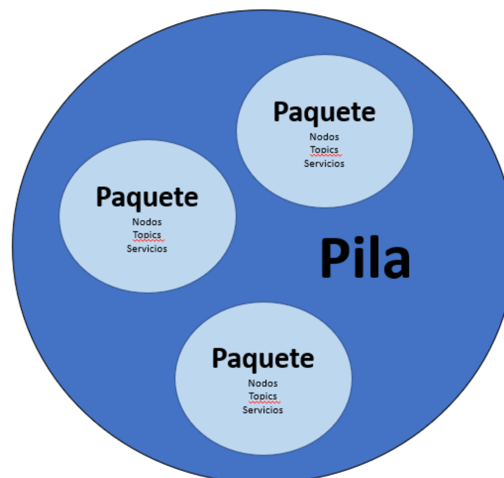

Ilustración 14 Estructura de una pila

3.1.1.1.3. Manifiesto del paquete

El manifiesto (*package.xml*) proporciona información sobre el paquete incluyendo su nombre, versión, descripción, información de licencia, dependencias, paquetes exportados… (Ilustración 15)

```
1 <package>
2   <name>turtlebot_bringup</name>
3   <version>2.4.2</version>
4   <description>
5      turtlebot_bringup provides roslaunch scripts for starting the TurtleBot base functionality.
6   </description>
7   <author>Tully Foote</author>
8   <maintainer email="d.stonier@gmail.com">Daniel Stonier</maintainer>
9   <license>BSD</license>
10  <url type="website">http://ros.org/wiki/turtlebot_bringup</url>
11  <url type="repository">https://github.com/turtlebot/turtlebot</url>
12  <url type="bugtracker">https://github.com/turtlebot/turtlebot/issues</url>
13
14  <buildtool_depend>catkin</buildtool_depend>
15
16  <run_depend>realsense_camera</run_depend>
17  <run_depend>astra_launch</run_depend>
18  <run_depend>zeroconf_avahi</run_depend>
19  <run_depend>yocs_cmd_vel_mux</run_depend>
20  <run_depend>kobuki_capabilities</run_depend>
21  <run_depend>kobuki_node</run_depend>
22  <run_depend>kobuki_bumper2pc</run_depend>
23  <run_depend>kobuki_safety_controller</run_depend>
24  <run_depend>create_node</run_depend>
25  <run_depend>turtlebot_capabilities</run_depend>
26  <run_depend>turtlebot_description</run_depend>
27  <run_depend>robot_state_publisher</run_depend>
28  <run_depend>robot_pose_ekf</run_depend>
29  <run_depend>diagnostic_aggregator</run_depend>
30  <run_depend>openni_launch</run_depend>
31  <run_depend>laptop_battery_monitor</run_depend>
32  <run_depend>rocon_app_manager</run_depend>
33  <run_depend>rocon_bubble_icons</run_depend>
34  <run_depend>depthimage_to_laserscan</run_depend>
35 </package>
```

Ilustración 15 Ejemplo del manifiesto (package.xml) del paquete turtlebot_bringup

3.1.1.1.4. Tipos de mensajes

Los mensajes son datos enviados de un proceso (nodo) a otro.

El directorio msg contiene información con las descripciones de los mensajes en un archivo "*.msg*". Este define las estructuras de datos para los mensajes que publican los diversos procesos. En ese archivo hay dos partes: campos y constantes. Los campos son los datos que se envían dentro del mensaje. Las constantes definen valores útiles que se pueden usar para interpretar esos campos. [15]

Por ejemplo, para la descripción de los campos:

```
int32 x
int32 y
```

Donde se define el tipo de variable que es utilizado.

También se le puede añadir un valor a esta descripción (constantes):

```
int32 X=123
int32 Y=-123
```

3.1.1.1.5. Tipos de servicios

El directorio srv contiene la información en un archivo ".*srv*" sobre la estructura del mensaje que se envían diferentes procesos utilizando la comunicación Petición-Respuesta. Describe los tipos de servicios. Sobre estos servicios se hablará en el apartado "3.1.1.2.4. Servicios".

La descripción del servicio consiste de dos partes separadas por guiones. En la parte de arriba se escribe el tipo de mensaje de la petición y en la de abajo el tipo de mensaje de la respuesta. Son dos archivos *".msg"* separados por guiones.

Se adjunta un ejemplo en el que se recibe un *String* y se devuelve otro *String*:

```
string str
---
string str
```

3.1.1.1.6. Código

Son directorios que contienen el código fuente de los programas que utilizará el paquete. Esta carpeta suele llamarse *src* o *scripts* (si el programa puede ser ejecutado por el usuario).

3.1.1.1.7. Launch

El directorio launch contiene ficheros Launch (*".launch"*), los cuales normalmente sirven para ejecutar varios nodos a la vez.

3.1.1.2. Grafo de procesos

ROS se basa en una red Peer-to-peer (P2P, red entre iguales) entre todos sus procesos, es decir, cada proceso está interconectado con otro y son iguales entre sí. Este sistema es el que permite desarrollar una computación distribuida, siendo de gran ventaja a la hora de detectar errores, ya que podemos aislar cada proceso de los demás, o a la hora de ejecutar procesos en diferentes ordenadores.

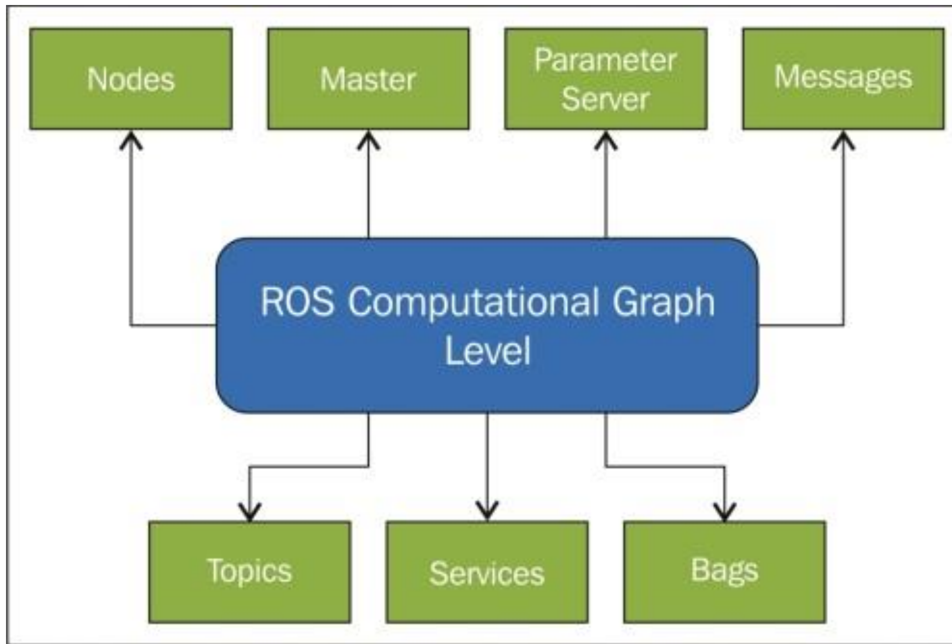

Ilustración 16 Conceptos del nivel de grafo de procesos [16]

Los conceptos más importantes de este nivel son los nodos, mensajes, topics, servicios, el nodo máster y los bags (Ilustración 16)

3.1.1.2.1. Nodos

Son la unidad de procesamiento, cada nodo es un proceso que realiza una tarea. Lo que se nombraba anteriormente como procesos, en nomenclatura de ROS se denominan nodos. Es un ejecutable dentro de un paquete ROS, escrito con la biblioteca de C++ (*roscpp*) o de Python (*rospy*).

Normalmente se utilizan varios nodos, estos se combinan dentro de un grafo compartiendo información (Grafo de procesos).

Por lo general, cada nodo tiene una función específica dentro del objetivo común. Esto es mejor que tener un único nodo con muchas funciones, ya que puede permitir detectar errores más fácilmente y una menor carga computacional. Por ejemplo, a la hora de mover un robot, un nodo puede encargarse de planificar una ruta mientras otro se encarga de visualizar el entorno.

3.1.1.2.2. Máster

El ROS Master permite la comunicación entre los nodos (Ilustración 17). Proporciona un registro de nombres que permite hacer una búsqueda dentro del grafo de computación. Sin el ROS Máster, los diferentes nodos del grafo no

se podrían encontrar unos a otros, y, como consecuencia, no podrían intercambiar mensajes ni invocar servicios.

Al ejecutar el ROS Máster, también se ejecuta el servidor de parámetros (*Parameter Server*), el cual utilizan los nodos para almacenar y recuperar parámetros en tiempo de ejecución. Además, se ejecuta el nodo "*rosout*", el cual se comporta como la salida estándar. Estos tres nodos siempre van de la mano.

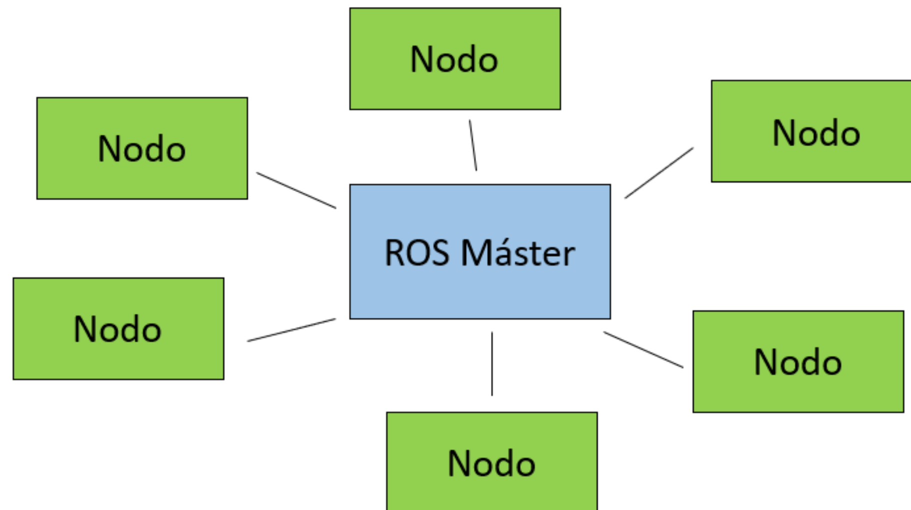

Ilustración 17 Gráfico ROS Máster y nodos

3.1.1.2.3. Mensajes

Los nodos se pueden comunicar entre ellos a través del uso de los mensajes. Estos mensajes son estructuras de datos con la información que se quiere comunicar. Admite los tipos estándar (*integer*, *floating point*, *boolean*, etc.) pero también se puede crear tipos personalizados.

3.1.1.2.4. Topics

Los nodos se pueden comunicar a través de un patrón de interacción publicador-suscriptor mediante los mensajes (Ilustración 18). Para poder comunicarse, se crean los topics, que son canales de información entre nodos. Luego un nodo puede comunicarse con otro publicando mensajes en un topic al cual el otro nodo tiene que estar suscrito (suscriptor).

Los nodos que quieran compartir una información en un determinado tipo de datos publicarán en un topic.

Los nodos que estén interesados en un determinado tipo de datos se suscribirán al topic correspondiente.

Un topic puede tener diversos publicadores y suscriptores concurrentes y un nodo puede publicar y/o suscribirse a varios topics.

Los publicadores y los suscriptores no son conscientes de la existencia de los demás.

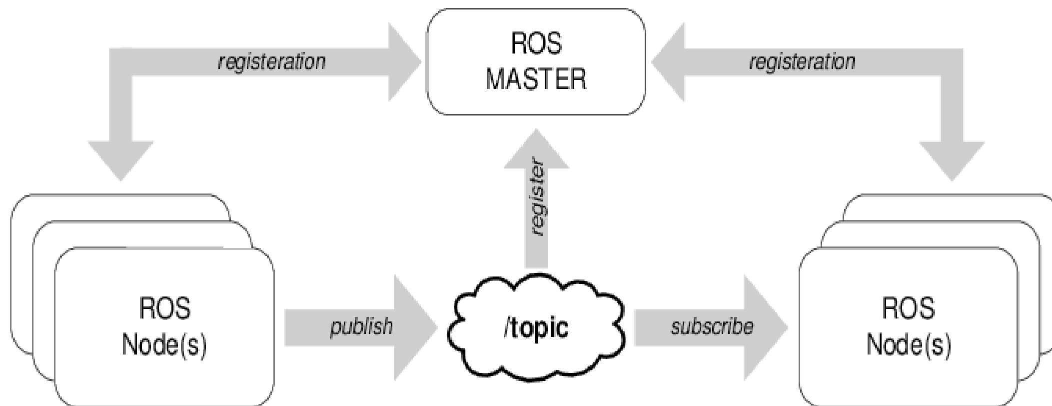

Ilustración 18 Comunicación en ROS [17]

3.1.1.2.5. Servicios

Los nodos también se pueden comunicar a través de un patrón de interacción entre nodos cliente-servidor, apropiado para las interacciones de petición-respuesta.

Utilizan dos mensajes, uno para petición y otro para respuesta.

Un nodo ofrece un servicio y otro nodo lo utiliza enviando una petición y esperando una respuesta.

3.1.1.2.6. Bolsas

Las bolsas (Bags) son un formato de almacenamiento de datos de ROS que permiten guardar datos enviados a través de mensajes, suscribiéndose al topic que se requiera y guardando los mensajes que se publiquen en un fichero. Estos ficheros también se pueden volver a reproducir en el mismo topic en el que fueron grabados. Son muy utilizadas para estudiar mensajes que publican los láseres, ya que a tiempo real publican mensajes

3.1.1.3. Comunidad de ROS

Una de las mayores ventajas de utilizar ROS es que es de código abierto, lo que quiere decir que el modelo de desarrollo de software está basado en la colaboración abierta. El nivel de comunidad de ROS se compone de recursos que permiten que diversos desarrolladores de cualquier parte del mundo puedan compartir sus proyectos y su conocimiento. Estos recursos incluyen:

- **Distribuciones**: Colecciones de distintas versiones de pilas que se pueden descargar
- **Repositorios**: ROS se basa en una red de repositorios de códigos, donde diferentes instituciones pueden desarrollar y lanzar sus propios componentes de software para robots. [14]
- **La enciclopedia de ROS**: Principal foro (wiki) para la documentación de información sobre ROS.
- **Sistema de fallos de tickets**: Sistema para indicar fallos en ROS.
- **Listas de correo**: Es el principal canal de comunicación sobre las nuevas actualizaciones de ROS.
- **ROS Answers**: Un foro de preguntas y respuestas donde los distintos usuarios de ROS exponen dudas y responden otras.

3.1.2. Herramientas y comandos

A continuación, se van a exponer las herramientas y comandos con los que cuenta ROS y van a ser importantes en la realización de este proyecto.

3.1.2.1. RVIZ

RVIZ proviene de la abreviatura de ROS Visualization, y es una herramienta gráfica que puede mostrar una gran variedad de información del sistema (Ilustración 19).

Con esta herramienta se puede visualizar la imagen que transmite la cámara 3D, ver la lectura de los sensores como el láser, el mapa 2D que obtiene el robot, la trayectoria que va a seguir dentro de este mapa, entre otras cosas. Además, es capaz de mostrar una representación del robot utilizado, pudiendo modelar cualquier robot en formato URDF.

Además de permitir visualizar gran información del sistema, también permite publicar mensajes sobre diversos topics.

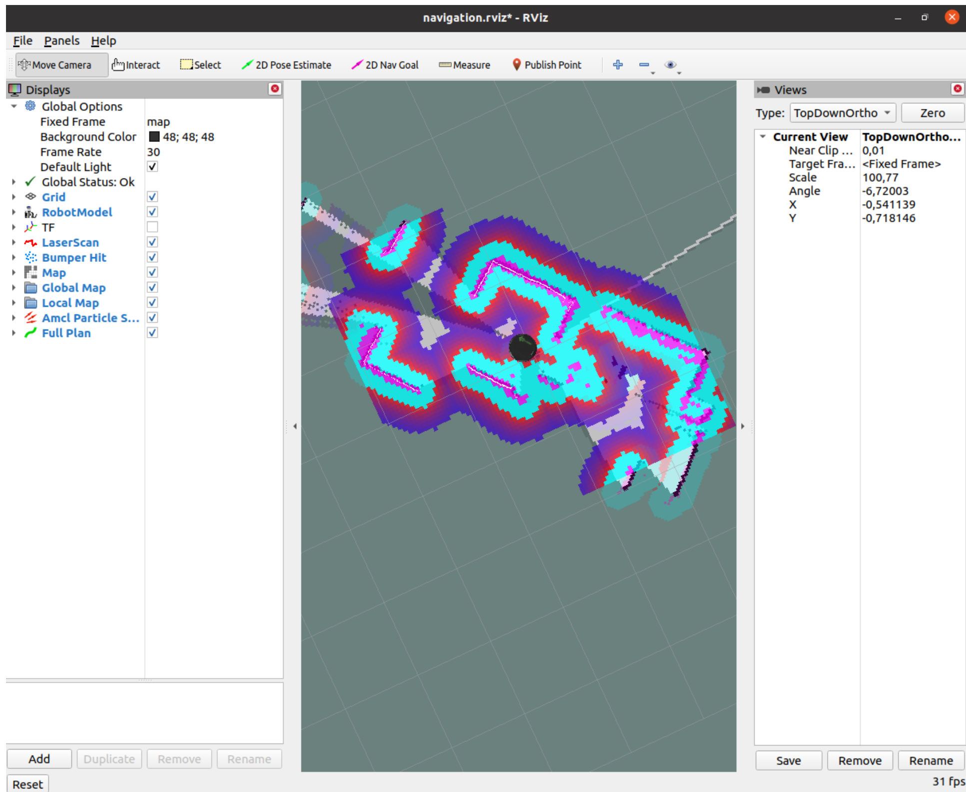

Ilustración 19 RVIZ

3.1.2.2. Simulador Gazebo

Gazebo es una herramienta de simulación que tiene un motor de físicas muy robusto, gráficos de gran calidad y una buena interfaz gráfica (ilustración 20). Permite la simulación de programas de control en un entorno 3D. Para esto lo principal es tener un fichero mundo ".world", que contenga los elementos de la simulación (robots, sensores, luces, objetos). Estos mundos pueden ser creados en la propia herramienta, con gran variedad de objetos ya creados. Es una aplicación sencilla, en pocos minutos se es capaz de realizar un mundo que imita una habitación. El otro concepto importante son los ficheros de modelo de los objetos y robots de la simulación. Con el robot Turtlebot2 no se va a tener ningún problema a la hora de implantar el modelo del robot, ya que es un robot muy común y la comunidad de ROS provee de este modelo bien realizado.

Gazebo se sincroniza con ROS de forma que el robot simulado puede publicar y recibir información de los diferentes topics al igual que si fuera un robot real. Por lo tanto, es una buena herramienta a la hora de realizar pruebas de programas de manera realista, ya que se evita utilizar el robot real, el cual puede romperse.

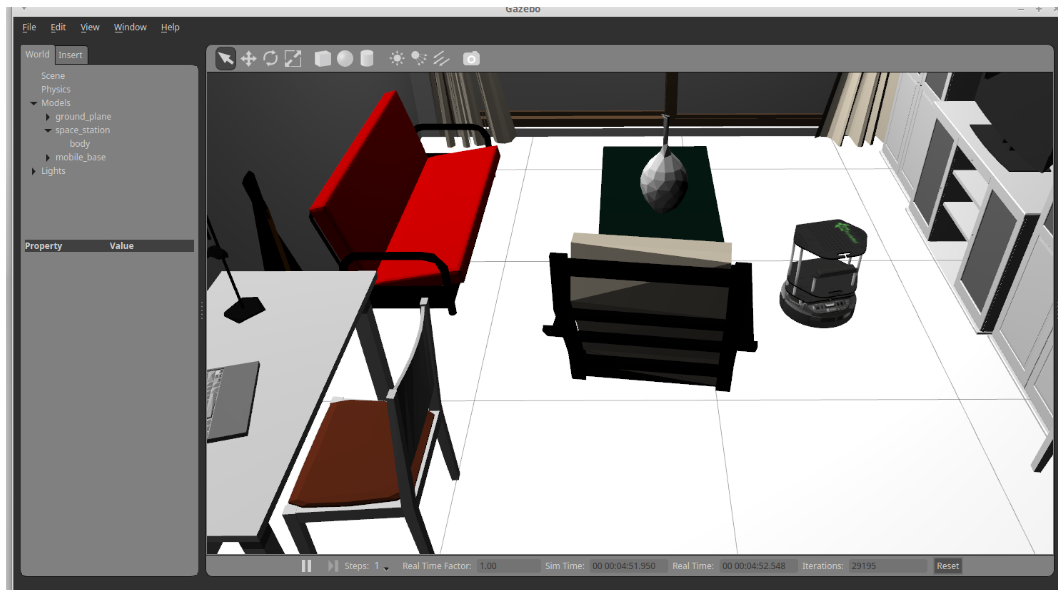

Ilustración 20 Simulación del robot Turtlebot en una habitación en Gazebo [18]

3.1.2.3. Comandos útiles

En este apartado se exponen diversos comandos de ROS que han sido útiles para la realización de este proyecto:

- **roscore**: inicia el máster, parameter server y rosout.
- **rospack list**: muestra paquetes instalados en ROS
- **rospack find nombre-paquete**: busca la ruta de un paquete.
- **rosrun nombre-paquete nombre-ejecutable**: ejecuta nodo de un paquete. Se puede pasar el valor de algún parámetro
- **rosnode list**: muestra nodos ejecutándose en ese instante
- **rosnode info nombre-nodo**: muestra información sobre el nodo (publicaciones, suscripciones, servicios…)
- **rqt_graph**: muestra el grafo de comunicación (Ilustración 21). Aquí los óvalos son los nodos y las etiquetas de las flechas son los topics. Las flechas representan las relaciones de publicación-suscripción.

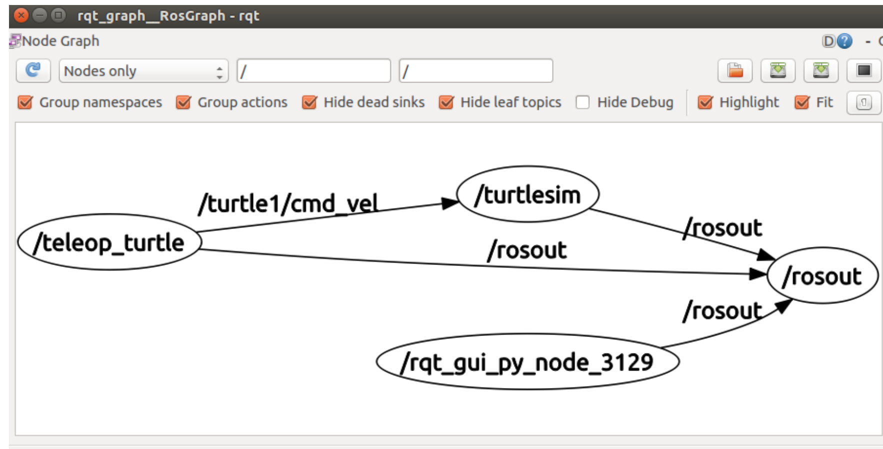

Ilustración 21 Grafo de comunicación rqt_graph

- **rostopic list**: muestra topics activos.
- **rostopic echo nombre-topic**: muestra los mensajes que se están publicando.
- **rostopic hz nombre-topic**: velocidad a la que se publican los mensajes.
- **rostopic bw nombre-topic**: ancho de banda consumido.
- **rostopic info nombre-topic**: información del topic (tipo de mensaje, publicadores, suscriptores)
- **rosmsg show nombre-tipo-mensaje**: detalles del tipo del mensaje.
- **rostopic pub -r rate-in-hz nombre-topic tipo-mensaje contenido-mensaje**: publica mensaje en el topic determinado a una velocidad determinada.
- **roslaunch nombre-paquete fichero.launch:** Lanza el fichero.launch determinado. Si se escribe este comando sin haber utilizado roscore antes, inicia el máster, parameter server y rosout.

3.2. TurtleBot2

Tras la explicación del software, se va a describir el hardware utilizado.

El robot que se ha empleado es la versión 2 del robot TurtleBot (Ilustración 22), al cual se le han añadido componentes para ampliar su rango de posibles actividades y mejorarle, como son un brazo robótico “PhantomX Reactor” y un telémetro láser (sensor LiDAR[1]).

Como se ha comentado en el apartado “2.3. La familia TurtleBot”, el robot Turtlebot es un robot de bajo precio y código abierto, lo cual hace que sea muy utilizado para la educación y la investigación.

Ilustración 22 Robot TurtleBot2 utilizado

Este robot se compone de una base móvil “Kobuki”, un miniordenador de controlador “Intel NUC 7I5BNK” y una cámara 3D “Orbbec Astra”, el cual se ha complementado con un telémetro láser de escaneo “Hokuyo URG-04LX” y un brazo robótico “PhantomX Reactor”.

[1] Light Imagine Detection and Ranging

3.2.1. Base móvil

La base móvil del robot es denominada “iClebo Kobuki”. Esta base de la empresa “Yujin Robot” (Ilustración 23) es de bajo precio y diseñada para la educación e investigación.

Ilustración 23 Base móvil "iClebo Kobuki" [19]

“iClebo Kobuki” cuenta con cuatro ruedas, dos de ellas en configuración diferencial (pueden girar a diferentes velocidades para girar), situadas en paralelo a la izquierda y derecha de la base, y otras dos como puntos de apoyo (Ilustración 24). Además, tiene diferentes sensores para evitar accidentes, tales como 3 sensores de presión colocados en la parte derecha, parte izquierda y parte frontal de la base; 2 sensores de detección de caída de rueda izquierda y derecha; y 3 sensores de detección de precipicio colocados también a la izquierda, derecha y parte frontal.

Para conseguir un sistema de navegación preciso, cuenta con sensores para la odometría (encoders en las ruedas muy precisos, se hablará sobre ella en el apartado “3.3. Pila de Navegación de ROS”) y un giroscopio calibrado de fábrica, lo que permite reconocer fácilmente dónde se encuentra el robot según sus movimientos.

La base está diseñada para que corra en ROS, con lo que todos estos sensores publican mensajes en determinados topics con sus respectivos estados.

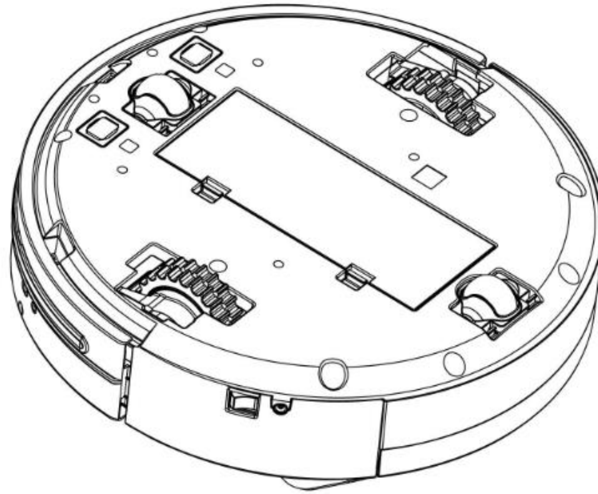

Ilustración 24 Parte inferior de la base

Además, provee de tres fuentes de alimentación (una de 5V y dos de 12V), luces LED programables, tres botones y audio.

Depende siempre de un ordenador de control, sin este no podría moverse, así que cuenta con una fuente para darle alimentación de 19V y un USB para su conexión.

Posee una estación de carga a la cual puede llegar automáticamente si se encuentra cerca gracias a sensores infrarrojos

En la guía de usuario de su página oficial, se pueden encontrar las especificaciones funcionales y del hardware. [20]

Especificaciones funcionales

- Velocidad lineal maxima de 0.7 m/s
- Velocidad angular máxima de 180 grados/s
- Capacidad de carga de 5kg en suelo duro, y de 4 Kg en alfombra.
- Detección de precipicios, no andará si hay una caída mayor de 5cm.
- Capaz de superar desniveles de 12 mm máximo (por ejemplo, subirse a una alfombra).
- Tiempo de operación de hasta 3/7 horas (batería pequeña/grande).
- Tiempo de carga de 1.5/2.6 horas (batería pequeña/grande).
- Capacidad de acoplarse a la base de carga por sí solo en un área de 2mx5m frente a la base.

Especificaciones del Hardware

- Conexión PC: USB o vía pines RX/TX
- Detección de sobrecarga del motor: corta la alimentación si detecta una corriente alta (3A).
- Odometría (52 ticks/enc rev, 2578.33 ticks/vuelta rueda o 11.7 ticks/mm).
- Giroscopio de 1 eje calibrado de fábrica (110 grados/s).
- Tres bumpers parachoques (frontal, izquierda y derecha).
- Tres sensores de precipicio (frontal, izquierda y derecha).
- Sensores de caída de rueda (rueda izquierda y derecha).
- Conectores de alimentación: 5V/1A, 12V/1.5A y 12V/5A.
- Conector de carga: 19V/2.1A.
- Pines de expansión: 3.3V/1A, 5V/1A, 4 entradas analógicas, 4 entradas digitales y 4 salidas digitales.
- Audio con diferentes secuencias programables usando pitidos.
- Tres botones táctiles.
- Dos LEDs programables de colores.
- Un LED de estado del robot:
    - Intermitente: Cargando batería.
    - Verde: Batería nivel alto.
    - Naranja: Batería nivel bajo.
- Batería de Lition-Ion de 2200 mAh (pqueña) y 4400 mAh (grande).
- Frecuencia de datos de los sensores: 50 Hz.
- Recarga de baterías mediante estación de carga.
- Firmware actualizable vía USB.

- Receptores IR de acoplamiento a la estación de carga (frontal, izquierda y derecha).

3.2.2. PC Controlador

El ordenador de control se encarga de ejecutar ROS y conectar entre sí todos los dispositivos del robot. Este es imprescindible para que el robot pueda moverse. Puede usarse cualquier ordenador que tenga una distribución de Linux que sea compatible con ROS.

Para este proyecto se ha elegido un Intel NUC 7I5BNK (Ilustración 25), un mini ordenador que ocupa poco espacio y tiene bajo consumo, además de tener una gran capacidad de procesamiento. Tiene un puerto HMI y cuatro puertos USB.

Este encaja perfectamente en la estructura del robot Turtlebot2.

Ilustración 25 Intel NUC 7I5BNK [21]

3.2.3. Cámara 3D

La cámara 3D es una cámara capaz realizar fotos en tres dimensiones, esto permite al robot tener una visión del entorno muy realista, pudiendo detectar el relieve de las cosas a las que enfoca.

Tener una cámara permite poder realizar aplicaciones con visión artificial, tales como aproximar el robot a un objeto que se requiera, leer un código QR, avistar señales,etc. Gracias a esto, se podría realizar un programa con el que se prohiba al robot entrar en una habitación poniendo una señal de prohibido.

Además, permite mapear el entorno gracias a sus medidas de profundidad.

El robot de este proyecto posee una cámara 3D conocida como “Orbbec Astra” (Ilustración 26), la cual tiene un campo de visión de 60º en horizontal y 49,5º en vertical y un alcance de 0,4 a 8 metros. Posee dos paquetes oficiales de ROS, gracias a los cuales la cámara se puede conectar con ROS y publicar sus datos en diversos topics, los cuales permiten utilizar la cámara para diversas funciones.

Por ejemplo, para poder mapear una habitación, la cámara publica mensajes del tipo LaserScan en el topic */scan* (es decir, se puede usar como telémetro láser), mientras que para poder visualizar las imágenes que está captando, escribe en el topic */camera/rgb/image_raw*.

Ilustración 26 Cámara 3D "Orbbec Astra" [22]

El problema de esta cámara es que, para mapear una habitación, su campo de visión es reducido, ya que con 60º de amplitud el robot solo es capaz de ver lo que tiene de frente, lo que hace que esta tarea sea tediosa de realizar. Este problema se va a solventar añadiendo al robot un telémetro láser de escaneo, el cual su única función es realizar medidas, con lo cual estas serán más

exactas que las de la cámara, además de conseguir abarcar un ángulo mayor del entorno estando en un mismo sitio.

3.2.4. Telémetro láser de escaneo

A pesar de que la cámara es capaz de mapear el entorno, se ha implementado un telémetro láser de escaneo "Hokuyo URG-04LX" (Ilustración 27), escáner láser 2D, pequeño, preciso y con alta velocidad de detección de obstáculos.

Este láser es un sensor LiDAR, permite determinar la distancia que le separa de los objetos que tiene a su alrededor por medio de un haz láser pulsado. Mide la distancia calculando el tiempo que tarda entre que emite el pulso hasta que recibe la señal rebotada. Este proceso es casi instantáneo, debido a la gran velocidad de la luz.

Ilustración 27 Telémetro láser de escaneo "Hokuyo URG-04LX" [23]

Uno de los aspectos destacados de este láser es el rango de escaneo de área de 240º, lo cual mejora considerablemente el campo visual que tiene la cámara (60º). Su rango de detección máximo son 4 metros (con precisión, sin precisión puede llegar a los 5,6m). Además, consigue unas medidas más precisas que la cámara.

3.2.5. Brazo robótico

Por último, el robot posee un brazo robótico "PhantomX Reactor" del fabricante "Trossen Robotics" (Ilustración 28) en su parte superior. Es un robot de 5 grados de libertad y compatible con ROS.

Se compone de 8 servomotores "AX-12A Dynamixel", de los cuales uno de ellos es utilizado para la apertura y cierre de la pinza y en dos articulaciones se utilizan un par de servomotores que trabajan de forma paralela, proporcionando mayor fuerza a esas articulaciones. Por esto que solo se consigan 5 grados de libertad con 8 servomotores. También posee un microcontrolador "Arbotix-M".

Para este robot, con este controlador, existe el paquete "*phantomx_reactor_arm*" realizado por la comunidad para poder controlar todas las articulaciones desde ROS. [24]

Debido al uso de servomotores como articulaciones, este robot puede llegar a ser algo impreciso.

Ilustración 28 Brazo robótico "PhantomX Reactor" [25]

En este proyecto no se ha dado uso a este brazo robótico.

3.3. Pila de navegación de ROS

La pila de navegación de ROS (navigation [26]) va a ser la base de la generación de trayectorias y del sistema de navegación autónomo del robot. Esta pila, la cual ha sido realizada por la comunidad, toma información de la odometría y de los sensores que recopilan datos sobre el entorno. Con esto, es capaz de estimar la posición del robot dentro de un mapa previamente generado, mediante el algoritmo de localización adaptativo de Monte Carlo (AMCL [2]) y generar trayectorias y modificarlas en función de los objetos que se encuentre por el camino y que no estén en el mapa (por ejemplo, una persona andando).

Para poder utilizar este paquete es necesario que el robot a utilizar tenga ruedas con control diferencial, para la odometría, y un láser 2D paralelo al suelo, para la construcción y localización de mapas. En la ilustración 29 se puede observar el esquema de flujo de información de esta pila, siendo los rectángulos azules nodos específicos que deben ser creados para cada robot, los grises son nodos opcionales que están proporcionados por el stack y los blancos son nodos requeridos que están ya implementados por el stack.

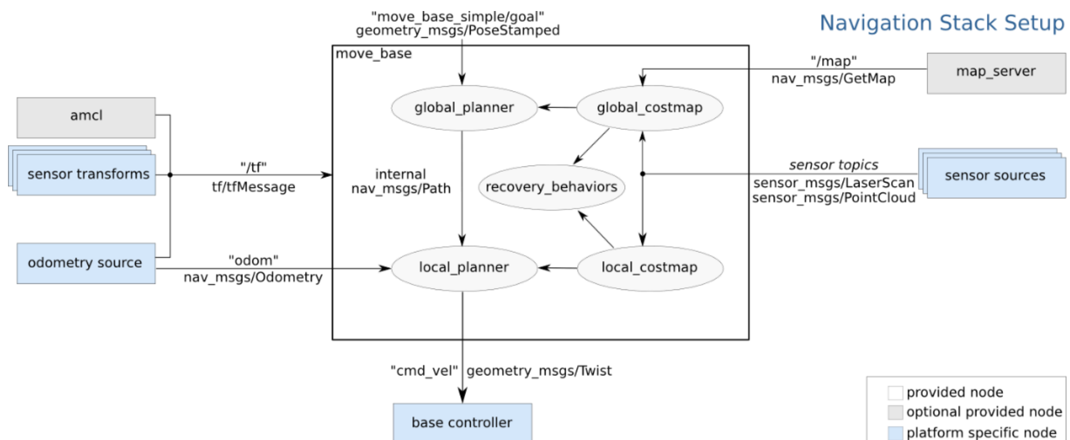

Ilustración 29 Flujo de la información en el paquete "navigation" [26]

Para la localización del robot dentro del mapa se van a utilizar tres elementos:

- Odometría.
- Transformadas de los diferentes sensores.
- Algoritmo de localización de Monte Carlo.

[2] Adaptative Monte Carlo Localization

3.3.1. Odometría (odometry source)

La odometría es el estudio de la estimación de la posición de vehículos con ruedas durante la navegación. Esta estimación se basa en la rotación de las ruedas para calcular los cambios de posición a lo largo del tiempo (ilustración 30).

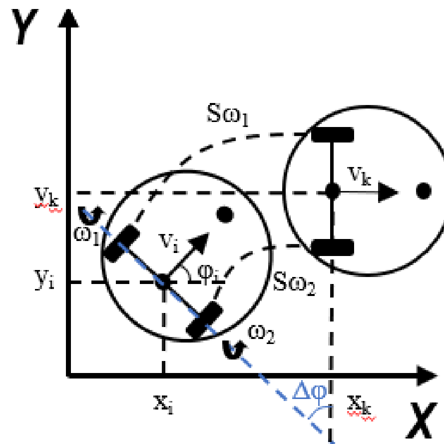

Ilustración 30 Modelo de sistema de odometría de un robot [27]

Para la generación de la odometría, el robot Turtlebot2 posee dos ruedas en configuración diferencial, donde sus respectivos encoders proporcionan la información con la posición de cada rueda.

El problema de la odometría es que no siempre se cumple ya que, por ejemplo, no tiene en cuenta el deslizamiento en las ruedas. Por esto, la estimación de la localización del robot se complementa con el algoritmo de localización adaptativo de Monte Carlo.

3.3.2. Algoritmo de localización adaptativo de Monte Carlo (AMCL)

El algoritmo de localización de Monte Carlo se basa en un filtro de partículas para que los robots puedan localizarse dentro de un entorno dado. Este filtro de partículas representa la distribución de estados posibles, es decir, cada partícula representa una hipótesis de dónde se encuentra el robot. Cada vez que el robot se mueve o detecta algo, las partículas se desplazan intentando predecir su nuevo estado, viendo si es coherente el estado que han previsto con el real obtenido por los sensores. Al final, las partículas deberían converger hacia la posición real del robot. Para esto, trabaja en paralelo con la odometría.

Tras el movimiento del robot, el algoritmo coge la información de los sensores y compara con qué posición del robot dentro del mapa se ajusta más, dando así una mejora en la localización del robot y una mayor precisión a lo largo del

tiempo, evitando los errores que existirían si la única fuente de la localización del robot fuera la odometría.

3.3.3. Transformadas de los sensores

Debido a la utilización de la odometría y de la información recibida de los sensores, es necesario que el sistema sepa en que posición en el cuerpo del robot se encuentran las ruedas o los láseres.

Para ello se utilizan diferentes sistemas de referencia para cada elemento. Se establece un sistema de referencia “padre” (suele ser el centro de la base), y a partir de él se establecen los demás. Para definir la relación entre el sistema de referencia “padre” con cualquier otro se utilizan las transformadas.

3.3.4. Funcionamiento de la pila

Tras haber explicado las bases para la localización de un robot dentro de un mapa, podemos explicar la generación de trayectorias que ofrece la pila explicando varios conceptos importantes:

- *Local_costmap*
- *Global_costmap*
- *Local_planner*
- *Global_planner*
- *Recovery_behaviors*

3.3.4.1. Mapas de costo

Para la generación de las trayectorias, la pila utiliza dos mapas.

Con el mapa que se le suministra al paquete (fichero *.yaml*), el sistema crea un mapa de costo denominado “mapa global” (*global_costmap*), utilizado para la posición general. Además, crea otro mapa de costo llamado “mapa local” (*local_costmap*) que supone una reconstrucción del mapa a corta distancia, generado por la lectura de los sensores, posicionando obstáculos si los hay.

A medida que se encuentran nuevos obstáculos, el mapa de costo global (*global_costmap*) es actualizado por el sistema con estos obstáculos, pero no reescribe el fichero *.yaml* de dónde se obtiene este mapa. Esto hace que obstáculos imprevistos puedan ser borrados sin necesidad de cambiar el mapa. Por ejemplo, si el mapa está guardado con las puertas abiertas de una sala y de repente se encuentra el obstáculo de que una puerta está cerrada,

esto haría que el sistema actualizase el mapa de costo global añadiendo un obstáculo ahí, pero no modificaría el mapa. Además, al abrir la puerta este obstáculo se borraría por sí solo.

3.3.4.2. *Planificadores*

El sistema de navegación va a estar pendiente todo el rato al topic "*move_base_simple/goal*", donde se publicarán las coordenadas del punto final al que se quiere llegar dentro del mapa. Cuando se recibe un mensaje en este topic, el sistema crea una primera trayectoria global a seguir a largo plazo con los datos recibidos del mapa de costo global (*global_planner*).

Para poder sortear obstáculos que no están contemplados en el mapa de costo global, se toman los datos del mapa de costo local y, con ellos, el sistema intenta encontrar una trayectoria a seguir de manera inmediata (trayectoria local) para sortear el obstáculo (*local_planner*, encargado de publicar los comandos de velocidad). Para que el robot se mueva, es necesario tener ambas trayectorias.

3.3.4.3. *Recovery Behaviors*

El sistema cuenta con mecanismos de recuperación si el robot no consigue llegar a la meta establecida.

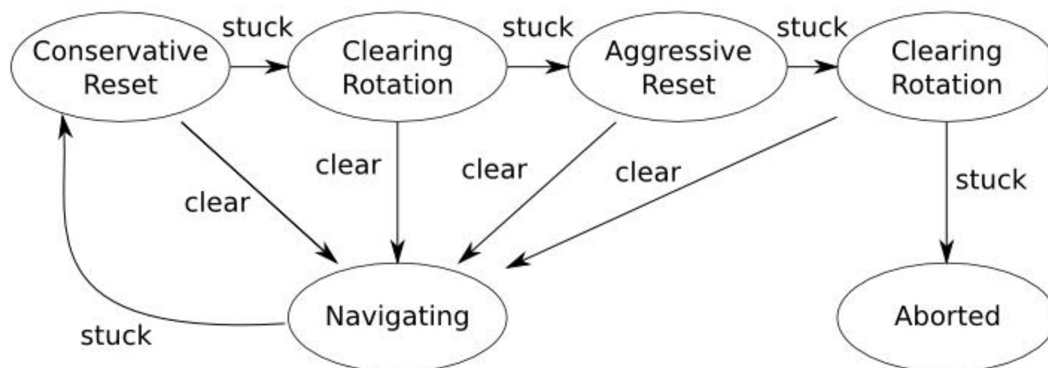

Ilustración 31 Recovery Behaviors [28]

Si no se encuentra trayectoria posible debido a que hay obstáculos de por medio, el robot se para en una posición. En esa posición, el robot resetea los mapas de costo a partir de un radio dado desde el centro del robot (3 metros por defecto, pero se puede cambiar en el parámetro "*conservative_reset*"), borrando los obstáculos que pueda haber fuera de ese radio. Si sigue atrapado,

da una vuelta de 360º actualizando los mapas de costo con lo que vea, comprobando si tiene salida por algún otro lado.

Si no la encuentra, reinicia los mapas de costo de manera más agresiva, borrando todos los obstáculos en un radio menor que el anterior (1,84 metros si no se ha cambiado el parámetro "*aggressive_reset*"), y, si con esto sigue igual, da otra vuelta de 360º. Tras esta, si no encuentra ninguna salida, el robot aborta el plan debido a que no se puede llegar a la meta. En la ilustración 31 se puede observar un gráfico con los diferentes estados del mecanismo de recuperación.

3.3.4.4. Parámetros

Para optimizar la navegación autónoma, los diferentes mapas y planificadores tienen diversos parámetros que se pueden editar en función de lo que se requiera.

Como parámetros importantes se pueden encontrar la distancia mínima que puede haber entre el robot y los diferentes obstáculos (*inflation_radius*), la dimensión del mapa local o los límites de velocidad y aceleración del robot.

3.4. Puesta en marcha del robot

En este apartado se describen los pasos a realizar para la puesta en marcha del robot Turtlebot. Cómo establecer la comunicación entre el PC controlador del robot con otro ordenador y cómo hacer que el robot empiece a funcionar. Además, se describirán paquetes básicos que se van a utilizar.

3.4.1. Comunicación entre ordenadores

Al no poder tener una pantalla conectada al mini ordenador controlador del robot mientras está en movimiento, se necesita que éste se pueda manejar desde otro ordenador. Para ello se utiliza el protocolo SSH (Secure Shell), el cual utiliza una arquitectura cliente/servidor y permite al usuario conectarse a un host remotamente por medio de un canal seguro en el que toda la información está cifrada, siempre y cuando esté

Para ello, primeramente, hay que tener instalado el servidor SSH en el ordenador de la Turlebot. Tras esto, ya se puede acceder vía SSH al ordenador.

Para ello en la terminal se utilizará el siguiente comando:

```
> ssh turtle@<TURTLEBOTP_IP>
```

Donde “*turtle*” es el nombre de usuario al que se quiere entrar y “<TURTLEBOTP_IP>” es la dirección IP de la Turtlebot. Tras ponerlo, pedirá la contraseña y se abrirá la terminal del PC del robot. No se tiene acceso a la visualización del escritorio, sino solo a su consola. Por cada terminal que se requiera, se necesitará ejecutar el comando SSH en otra terminal del ordenador fijo.

3.4.2. Conectar ROS Máster

Tras conseguir utilizar el ordenador de la Turtlebot de manera remota, ahora lo importante es conseguir que los nodos lanzados desde un ordenador u otro se puedan comunicar entre ellos. Para esto se necesita que ambos ordenadores puedan encontrar el ROS Máster y que sea el mismo. Para ello, hay que utilizar los siguientes comandos.

En el ordenador de la Turtlebot se exportan las variables ROS_MASTER_URI Y ROS_HOSTNAME con los siguientes valores:

```
export ROS_MASTER_URI=http://localhost:11311

export ROS_HOSTNAME=IP_OF_TURTLEBOT
```

Donde localhost e IP_OF_TURTLEBOT son la dirección IP del ordenador de la Turtlebot.

Mientras que en el ordenador fijo hay que exportar estas variables con los siguientes valores:

```
export ROS_MASTER_URI=http://IP_OF_TURTLEBOT:11311

export ROS_HOSTNAME=IP_OF_PC
```

Donde IP_OF_TURTLEBOT es la dirección IP del ordenador de la Turtlebot e IP_OF_PC es la dirección IP del ordenador fijo.

Estos comandos habría que escribirlos cada vez que se abra una nueva terminal. Para evitar esto, se escribirá en el */.bashrc* de cada ordenador, sus respectivos comandos. Esto se debe a que el archivo */.bashrc* es un script que se ejecuta cada vez que se inicia una nueva terminal, ejecutando todos los comandos que tenga en su interior. Luego si se le añaden estos comandos, se ejecutarán nada más abrir la terminal, evitando que se requiera ponerlos cada vez que necesitamos una nueva terminal.

Tras esto, para comprobar que ambos ordenadores están conectados al mismo ROS Máster, se lanza *roscore* y se mira la lista de topics de ambos ordenadores, donde deberían aparecer los mismos topics en los dos.

3.4.3. Espacio de trabajo

Lo más cómodo a la hora de trabajar con ROS es crear un espacio de trabajo, es decir, un directorio donde vayan a estar guardados todos los paquetes que se hayan realizado para el proyecto. ROS guarda por defecto los paquetes instalados en la ruta */opt/ros/kinetic/*. Se podrían guardar aquí los paquetes realizados, pero para modificar documentos de este directorio son necesarios privilegios de súper usuario, además de que queda tener directorio propio para los paquetes del proyecto hace que esté mejor organizado.

ROS posee una herramienta denominada *catkin*, la cual se utiliza para construir paquetes. Para poder utilizarlo es recomendado tener un espacio de trabajo. Esta herramienta permite crear paquetes de manera más cómoda que haciendo todo a mano, creando los ficheros necesarios con las dependencias que requiera el paquete.

En el proyecto se ha creado un espacio de trabajo llamado "*catkin_ws*", en él se crea una carpeta denominada "*src*" donde se guardan los paquetes creados y/o copiados de la comunidad de ROS para el proyecto.

Siempre que se vayan a utilizar nodos de algún directorio, hay que compilar el archivo de configuración del entorno. Para ello, cada vez que se utilice una nueva terminal, se escribirán los siguientes comandos:

```
$ source /opt/ros/%YOUR_ROS_DISTRO%/setup.bash
$ source /root/catkin_ws/devel/setup.bash
```

Siendo %YOUR_ROS_DISTRO% el nombre de la distribución instalada de ROS (por ejemplo, *kinetic*), ya que en esta ruta se guardan todos los nodos que han sido instalados.

El segundo comando compila los archivos de configuración del espacio de trabajo.

Para no tener que escribir estos comandos cada vez que se inicie una nueva terminal, se pueden escribir estos comandos en el archivo */.bashrc*, como en el caso anterior.

3.4.4. Preparar Turtlebot

Una vez instalados todos los paquetes necesarios, en especial de la pila "Turtlebot"; habiendo realizado la comunicación entre ambos ordenadores y compilado los archivos de configuración del entorno, es posible comenzar a manejar el robot.

Para el inicio del Robot, existe un paquete denominado “Turtlebot_bringup”. Este paquete contiene un archivo Launch que se llama “minimal.launch” y se encarga de ejecutar los nodos básicos. Uno de estos nodos básicos es el “kobuki_node”, encargado de realizar la comunicación entre la base móvil y ROS.

Este archivo Launch se ha modificado de cara al proyecto, creando uno nuevo, para que también ejecute el nodo que permita utilizar el sensor LiDAR “Hokuyo” en lugar de la cámara para el escaneo del entorno, debido a sus ventajas con respecto a esta (“Apartado 3.2.3. Cámara 3D”). El sensor comienza a publicar la información que obtiene del entorno en el topic /scan. Además, se modifican los valores de los parámetros que ajustan el campo de visión debido a que el sensor LiDAR tiene un mayor campo de visión que la cámara (Ilustración 32)

```
<launch>
  <!-- Turtlebot -->
  <arg name="base"              default="$(env TURTLEBOT_BASE)"         doc="mobile base type [create, roomba]"/>
  <arg name="battery"           default="$(env TURTLEBOT_BATTERY)"      doc="kernel provided locatio for battery info, use /proc/acp
i/bat$ name="stacks"            default="$(env TURTLEBOT_STACKS)"       doc="stack type displayed in visualisation/simulation [circl
es, h$ name="3d_sensor"         default="$(env TURTLEBOT_3D_SENSOR)"    doc="3d sensor types [kinect, asux_xtion_pro]"/>
  <arg name="simulation"        default="$(env TURTLEBOT_SIMULATION)"   doc="set flags to indicate this turtle is run in simulation
mode.$ name="serialport"        default="$(env TURTLEBOT_SERIAL_PORT)"  doc="used by create to configure the port it is connected on
 [/de$
  <param name="/use_sim_time" value="$(arg simulation)"/>

  <include file="$(find turtlebot_bringup)/launch/includes/robot.launch.xml">
    <arg name="base" value="$(arg base)" />
    <arg name="stacks" value="$(arg stacks)" />
    <arg name="3d_sensor" value="$(arg 3d_sensor)" />
  </include>
  <include file="$(find turtlebot_bringup)/launch/includes/mobile_base.launch.xml">
    <arg name="base" value="$(arg base)" />
    <arg name="serialport" value="$(arg serialport)" />
  </include>
  <include unless="$(eval arg('battery') == 'None')" file="$(find turtlebot_bringup)/launch/includes/netbook.launch.xml">
    <arg name="battery" value="$(arg battery)" />
  </include>

  <node name="hokuyo" pkg="urg_node" type="urg_node" respawn="false" output="screen">
    <param name="calibrate_time" type="bool" value="true"/>
    <param name="port" type="string" value="/dev/ttyACM0"/>
    <param name="intensity" type="bool" value="false"/>
    <param name="min_ang" value="-2.35619449615"/>
    <param name="max_ang" value="+2.09234976768"/>
    <param name="cluster" value="1"/>
    <param name="frame_id" value="hokuyo_laser_frame"/>
  </node>

</launch>
```

Ilustración 32 minimal_with_hokuyo.launch

Para poder ejecutar este Launch se escribirá el siguiente comando:

```
$ roslaunch turtlebot_bringup minimal_with_hokuyo.launch
```

Siendo *“minimal_with_hokuyo.launch”* el archivo Launch creado para el uso del sensor LiDAR.

3.4.5. Teleoperación por teclado

El movimiento del robot puede ser controlado por teclado, ya sea vía SSH o desde el PC fijo (si la comunicación se ha configurado bien), siempre y cuando se haya arrancado la Turtlebot como se ha explicado en el apartado anterior.

Para ello, la pila "Turtlebot" contiene un paquete denominado *"turtlebot_teleop"* con el script *"turtlebot_teleop_key"*, que puede ser ejecutado como nodo con el siguiente comando:

```
$ roslaunch turtlebot_teleop keyboard_teleop.launch
```

Siendo este archivo Launch el encargado de ejecutar el nodo.

La interfaz de usuario se vería como en la ilustración 33.

```
Control Your Turtlebot!
---------------------------
Moving around:
   u    i    o
   j    k    l
   m    ,    .

q/z : increase/decrease max speeds by 10%
w/x : increase/decrease only linear speed by 10%
e/c : increase/decrease only angular speed by 10%
space key, k : force stop
anything else : stop smoothly

CTRL-C to quit

currently:      speed 0.2       turn 1
```

Ilustración 33 Interfaz de teleoperación por teclado del robot 1

El paquete *"kobuki_keyop"* contiene otro script encargado de la teleoperación por teclado. En este los movimientos en función de las teclas son diferentes. Se ejecuta con el comando:

```
$ roslaunch kobuki_keyop keyop.launch
```

En este nodo, el robot se controla por incrementos de velocidad, incrementando la velocidad lineal o angular en función de la entrada del teclado (ilustración 34).

```
Reading from keyboard
---------------------------
Forward/back arrows : linear velocity incr/decr.
Right/left arrows : angular velocity incr/decr.
Spacebar : reset linear/angular velocities.
d : disable motors.
e : enable motors.
q : quit.
```

Ilustración 34 Interfaz de teleoperación por teclado del robot 2

El usuario puede elegir cualquiera de estos para mover el robot.

3.4.6. Verificar el funcionamiento de la cámara

Para utilizar la cámara si se requiere ver a través de ella, el paquete "*turtlebot_bringup*" tiene un archivo Launch llamado "3dsensor.launch" que permite a la cámara publicar en topics destinados a la visualización de la imagen de esta. Para ello, hay que pasarle el tipo de cámara que se está utilizando, en el caso de este proyecto es una astra:

```
$ roslaunch turtlebot_bringup 3dsensor.launch 3dsensor:=astra
```

El paquete *"image_view"* contiene un nodo con el mismo nombre que permite visualizar lo que está viendo la cámara traduciendo los datos que recibe en el topic indicado. El topic */camera/rgb/image_raw* recibe la imagen de la cámara a color, luego para poder visualizarlo se tiene que escribir el siguiente comando:

```
$ rosrun image_view image_view image:=/camera/rgb/image_raw
```

Este comando hay que escribirlo en el ordenador fijo, ya que la comunicación SSH solo permite visualizar la terminal y no sería capaz de abrir un display con la imagen.

Otro método para visualizar la imagen que recibe de la cámara es utilizar RViz. Para ello se ejecutaría RViz en el ordenador fijo.

Tras esto, se cambiaría la opción global Fixed Frame a *camera_rgb_optical_frame*, ya que solo se va a utilizar la cámara, no importa lo demás. Después, se añade un display tipo "*camera*" y en "*Image topic*" se añade el topic que se requiere visualizar, en este caso */camera/rgb/image_raw*. El resultado sería una ventana donde se puede visualizar la imagen que manda la cámara a tiempo real (ilustración 35).

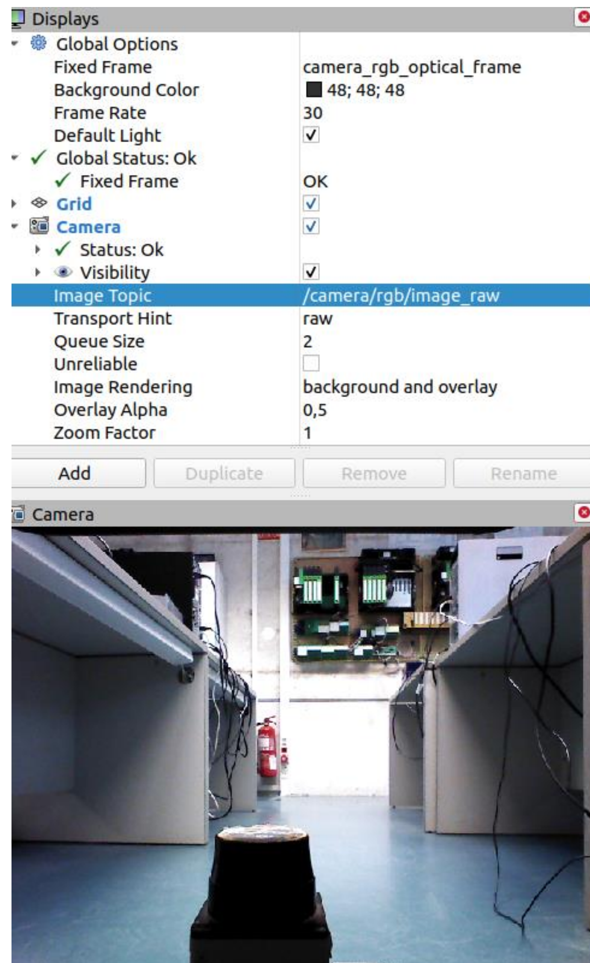

Ilustración 35 Visualización de la imagen que manda la cámara en RViz

Para poder realizar todo esto se tiene que haber arrancado la Turtlebot antes (Apartado "3.4.4. Preparar Turtlebot").

3.4.7. Construir un mapa (SLAM)

El paquete "*turtlebot_navigation*" proporciona las herramientas necesarias para crear un mapa a partir de los datos recibidos del entorno por medio de los sensores y la odometría.

Un algoritmo SLAM (del inglés, "*Simultaneous Localization And Mapping*") es aquel que consigue que el robot construya un mapa del entorno y que se localice mientras navega en él de manera simultánea. Esto se consigue gracias al archivo Launch del paquete "*turtlebot_navigation*" llamado "*gmapping_demo.launch*", el cual ejecuta un algoritmo SLAM.

Este algoritmo SLAM ejecutado es el que proporciona el paquete "*gmapping*" proporcionado por ROS, denominado "*slam_gmapping*".

Para poder utilizarlo se requiere que el robot proporcione datos de odometría y que tenga un telémetro láser fijo y montado horizontalmente, pudiendo ser la cámara ya que, como se ha explicado en el apartado "3.2.3. Cámara 3D", puede simular el escáner de un láser escribiendo en el topic */scan*.

El archivo launch "*gmapping_demo.launch*" del que se ha hablado antes, por defecto utiliza la configuración que se utilizaría si se usase la cámara. Como el sensor LiDAR es más preciso y tiene mayor campo de visión, se ha modificado este archivo para conseguir usar el sensor LiDAR y se le ha llamado "*gmapping_demo_hokuyo.launch*" (ilustración 36). Para ello, hay que cambiar el parámetro "*scan_processing*" a falso, lo que consigue apagar el proceso de escaneo de la cámara. Además, se modifican diversos parámetros del gmapping para optimizar el uso del LiDAR.

```
<launch>
  <!-- 3D sensor -->
  <arg name="3d_sensor" default="$(env TURTLEBOT_3D_SENSOR)"/>  <!-- r200, kinect, asus_xtion_pro -->
  <include file="$(find turtlebot_bringup)/launch/3dsensor.launch">
    <arg name="rgb_processing" value="false" />
    <arg name="depth_registration" value="false" />
    <arg name="depth_processing" value="false" />
    <arg name="scan_processing" value="false" />
    <!-- We must specify an absolute topic name because if not it will be prefixed by "$(arg camera)".
         Probably is a bug in the nodelet manager: https://github.com/ros/nodelet_core/issues/7 -->
    <arg name="scan_topic" value="/scan" />
  </include>

  <!-- Gmapping -->
  <arg name="custom_gmapping_launch_file" default="$(find turtlebot_navigation)/launch/includes/gmapping/$(arg 3d_sensor)_hokuyo_gma
pping.launch.xml"/>
  <include file="$(arg custom_gmapping_launch_file)"/>

  <!-- Move base -->
  <include file="$(find turtlebot_navigation)/launch/includes/move_base.launch.xml"/>

</launch>
```

Ilustración 36 gmapping_demo_hokuyo.launch

Se adjunta una comparación de un mapa realizado con los datos de la cámara y otro realizado por el LiDAR, en el mismo entorno, para dar una muestra de la diferencia que hay entre ambos (Ilustraciones 37 y 38). Se puede observar que el mapa generado con el LiDAR es mucho más limpio.

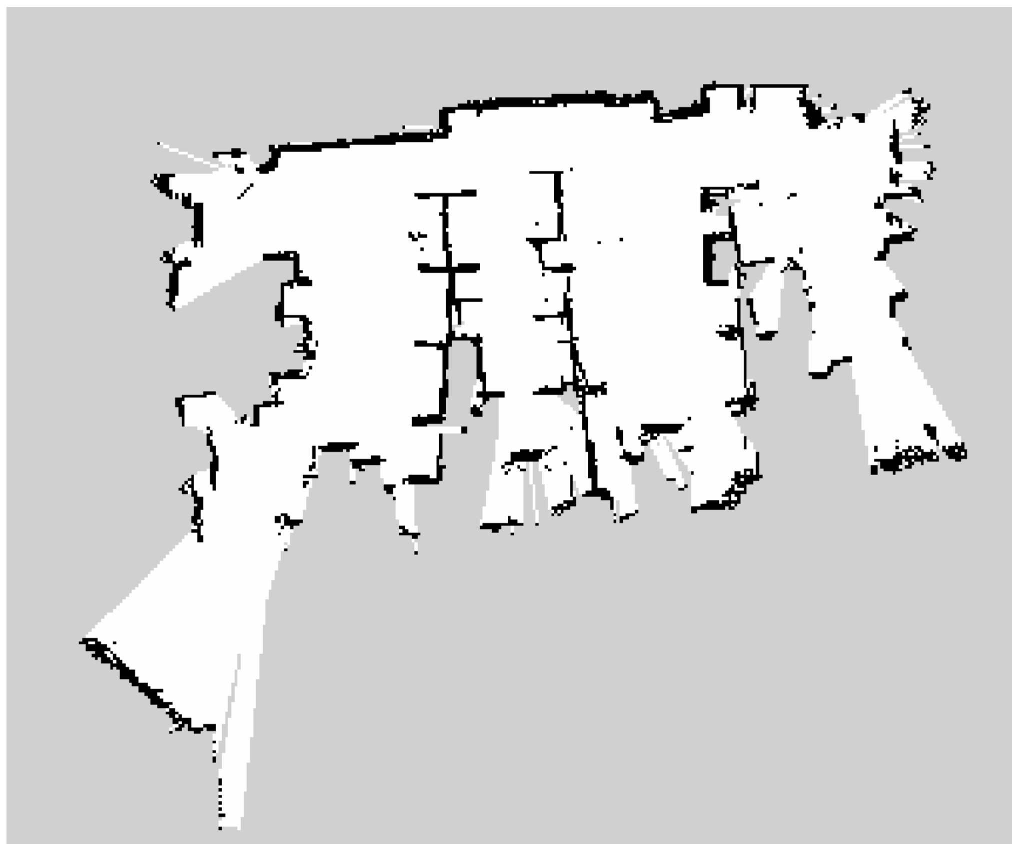

Ilustración 37 Mapa realizado por la cámara 3D

Ilustración 38 Mapa realizado por el sensor LiDAR "Hokuyo"

Para poder comenzar a mapear, lo primero de todo es arrancar la Turtlebot. Si se requiere utilizar el sensor LiDAR, se utilizarán los archivos Launch que se han modificado para ello. Luego, en diferentes terminales hay que ejecutar los siguientes comandos:

```
$ roslaunch turtlebot_bringup minimal_with_hokuyo.launch
$ roslaunch turtlebot_navigation gmapping_demo_hokuyo.launch
```

Para poder navegar con el robot mientras va mapeando, en otra terminal se puede ejecutar el nodo de teleoperación por teclado del robot que se ha explicado en el apartado "3.4.5. Teleoperación por teclado".

Para visualizar el mapa que se está creando, en el ordenador fijo (ya que es el que puede abrir RViz) se ejecuta el siguiente comando:

```
$ roslaunch turtlebot_rviz_launchers view_navigation.launch
```

Este archivo launch del paquete "*turtlebot_rviz_launchers*" se encarga de ejecutar RViz con la configuración ya realizada para poder visualizar el robot mientras crea el mapa o mientras navega gracias a la pila de navegación de ROS.

Además, también es posible navegar con el robot por medio de RViz, ya que proporciona una herramienta denominada "*2D Nav Goal*" con la cual se puede marcar la posición a la que se quiere llegar y la orientación, se publica en el topic "*move_base_simple/goal*", y el robot intenta crear una trayectoria para llegar hasta ella.

Esto se debe a que el Launch *gmapping* también ejecuta el nodo "*move_base*", encargado de la navegación autónoma, siendo el nodo principal de la pila de navegación de ROS

Cuando se haya obtenido el mapa que se requiere, para guardarlo se utiliza el siguiente comando en otra terminal (sin cerrar el Launch *gmapping*):

```
$ rosrun map_server map_saver -f /tmp/my_map
```

Siendo "*/tmp/my_map*" la ruta y el nombre donde se quiere guardar el mapa.

3.4.8. Navegación autónoma en un mapa conocido

Una vez generado el mapa, se podrá realizar una navegación autónoma por el entorno mapeado.

Para ello, tras arrancar la Turtlebot, hay que ejecutar el archivo Launch "*amcl_demo.launch*" del paquete "*turtlebot_navigation*", al cual se le ha de pasar el mapa creado:

```
$ roslaunch turtlebot_navigation amcl_demo.launch map_file:=/ruta/
my_map.yaml
```

Este archivo ejecuta los nodos encargados de la navegación autónoma y la localización en el mapa. Este paquete sigue los principios de la pila de navegación de ROS, explicada anteriormente. Además, como en el caso anterior, se ha modificado este archivo para apagar el proceso de escaneo de la cámara cambiando el valor del parámetro "*scan_processing*" a false. Consiguiendo así que el sensor LiDAR sea el único que escriba en el topic */scan*. A este nuevo archivo Launch se le ha denominado "*amcl_demo_hokuyo.launch*".

Para visualizar el mapa, la posición del robot y poder enviarle un destino y decirle donde está en un inicio, hay que lanzar el Launch "*view_navigation.launch*" del paquete "*turtlebot_rviz_launchers*" en el PC, como al mapear el entorno:

```
$ roslaunch turtlebot_rviz_launchers view_navigation.launch
```

Al iniciar los nodos, el robot no es capaz de saber dónde se encuentra. Hay que proporcionarle un punto aproximado de dónde se encuentra en el instante inicial, para que el algoritmo AMCL consiga situar el robot en el mapa más rápidamente. Esto se puede hacer con la herramienta "*2D Pose Estimate*" que ofrece RViz, la cual publica en el topic "*initialpose*".

Tras esto, se le podrá enviar el destino hasta el que se quiera navegar dentro del mapa. Esto se realizará con la herramienta de RViz mencionada en el apartado anterior "*2D Nav Goal*". Planificará la ruta para evitar obstáculos y empezará a moverse (si es posible llegar al destino). Si se encuentra con algún obstáculo imprevisto, lo conseguirá sortear si tiene hueco, como se ha explicado en el apartado "3.3.4.2. Planificadores".

3.4.9. Aparcamiento automático

Como se ha visto en el apartado "3.2.1. Base Móvil", el robot puede llegar automáticamente a la estación de carga gracias a rayos infrarrojos. La

estación de carga emite luces infrarrojas cubriendo tres regiones frente a ella, izquierda, centro y derecha, cada una dividida en dos campos: cerca y lejos (ilustración 39). Cada haz codifica esta información, luego el robot al recibirlo sabe en qué región se encuentra.

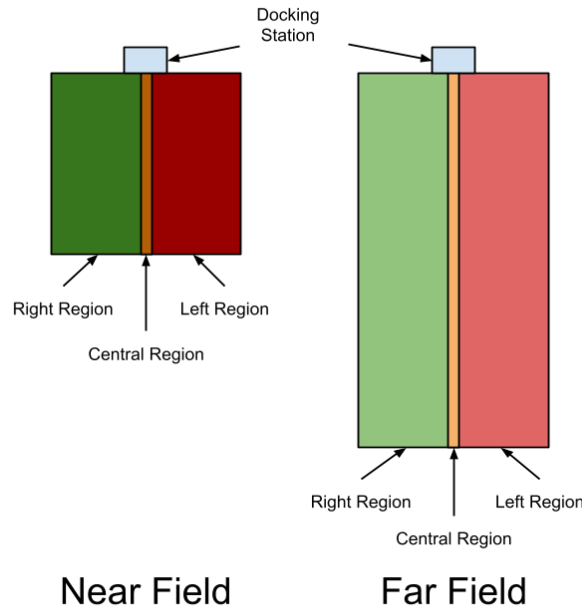

Ilustración 39 Regiones rayos infrarrojos emitidas por la estación [29]

La base móvil Kobuki tiene 3 receptores de rayos infrarrojos, uno delante, otro a la izquierda y otro a la derecha (Ilustración 40). Cuando el robot está dentro del área que cubre la base con los rayos emitidos y uno de los recibidores está orientado hacia el emisor, el robot es capaz de captar la señal y mandarla al controlador por el topic "*/mobile_base/sensors/dock_ir*", indicando dónde está.

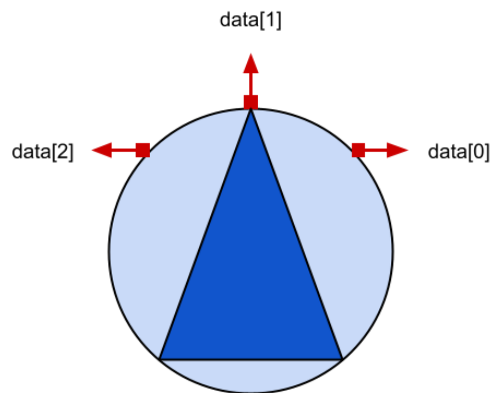

Ilustración 40 Recibidores infrarrojos [29]

La pila de paquetes “*kobuki*” ofrecida por el comerciante de la base móvil, contiene el paquete “*kobuki_auto_docking*”. Este paquete ofrece las herramientas para que la base pueda llegar a la estación de carga automáticamente.

La idea del algoritmo que ofrece este paquete es la siguiente. Si el robot está en la región central es fácil de aparcar, ya que solo necesitará seguir esta señal de frente.

Si el robot está en la región izquierda, el robot realizará un giro antihorario hasta detectar una señal de la región izquierda con el sensor derecho. Esto hace que el robot se encuentre de manera perpendicular a la región central, luego moviéndose hacia delante acabará encontrando la región central. Tras encontrar la región central, el robot seguirá esta señal de frente. Si el robot se hubiera encontrado en la región derecha, se haría algo similar, pero con las direcciones invertidas.

Para poder realizar esto, no tiene que haber obstáculos entre el robot y la estación de carga.

Primero se carga el algoritmo con el comando:

```
$ roslaunch kobuki_auto_docking minimal.launch --screen
```

Este comando solo carga el algoritmo, no lo activa. Solo se activa cuando un servidor le llama, luego para comenzar a aparcar se utilizará en otra terminal un nodo que actúe como cliente y llame a este servidor. El paquete ofrece el archivo “*activate.launch*” que ejecuta el nodo cliente:

```
$ roslaunch kobuki_auto_docking activate.launch --screen
```

Con esto, el robot comenzaría a intentar aparcar por su cuenta.

4. Capítulo 4. Sistema de mapeo y navegación.

En este capítulo se expone el sistema de navegación que requería este proyecto, junto con la explicación del código. Todo el código está escrito en lenguaje Python, bajo la versión 2.7.11.

4.1. Mapeo automático.

4.1.1. Descripción del problema

En primer lugar, se ha querido realizar un programa que consiguiera mapear el entorno sin necesidad de que el usuario estuviera controlando el robot.

Para una primera aproximación, se ha realizado un programa que, utilizando los sensores del parachoques, consiga que el robot pueda moverse por toda la habitación mapeándola sin quedarse atrapado. Esto se realizó pensando en un futuro realizar un programa con el que, en vez de llegar a chocar con las paredes, el robot pudiera reconocer que hay una pared cerca y girar antes, gracias a la lectura de sus sensores.

4.1.2. Diagrama de flujo del programa

Este programa, escrito en lenguaje *python*, se llama "*bumper.py*", se adjunta el diagrama de flujo para comenzar la explicación (Ilustración 41).

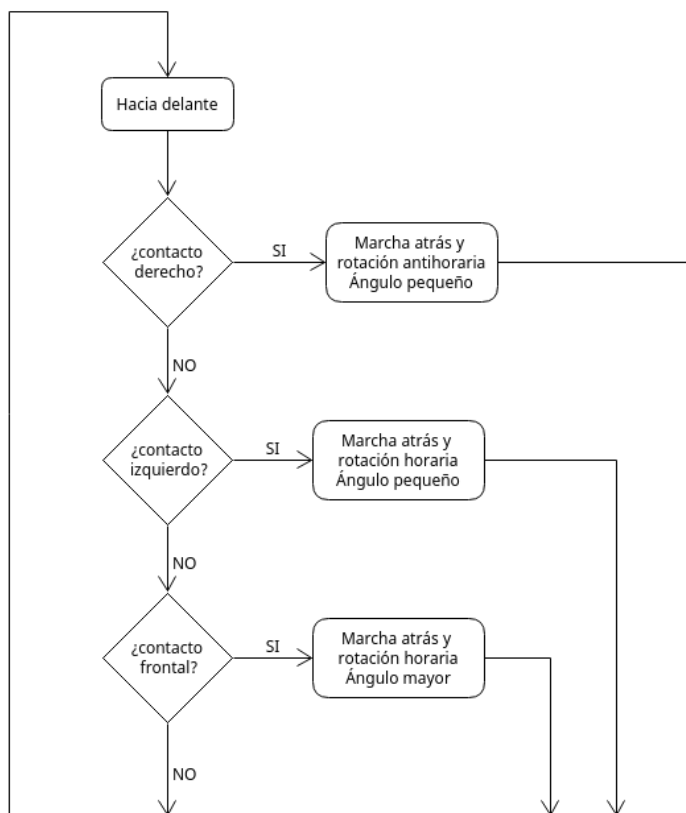

Ilustración 41 Diagrama de flujo del programa "bumper.py"

Este es un programa sencillo, para comenzar a tantear el manejo del robot mediante código.

El robot irá siempre hacia delante hasta que choque contra alguna pared. Si el choque proviene por la izquierda, el sensor del parachoques publicará en el topic */mobile_base/events/bumper* un mensaje del tipo *BumperEvent* (Ilustración 42). Dentro de este mensaje, el valor de la variable *state* indica si el parachoques está presionado o no y la variable *bumper* nos indica el lado por el que se ha chocado.

Entonces, si se choca por la derecha, para poder retomar el camino, el robot tendrá que dar marcha atrás y rotar en sentido antihorario un ángulo pequeño, ya que poco a poco conseguirá despegarse del obstáculo con el que está chocando.

Si se choca por la izquierda haría exactamente lo mismo, cambio el sentido de la rotación al de las agujas del reloj.

Si se choca de frente, necesitamos un ángulo de giro más grande para sortear este obstáculo, luego se ha impuesto que dé marcha atrás y rote en sentido horario un ángulo más grande (de casi unos 90º).

Además, notificará por pantalla cada vez que haya una colisión y por dónde.

kobuki_msgs/BumperEvent Message

File: kobuki_msgs/BumperEvent.msg

Raw Message Definition

```
# Provides a bumper event.
# This message is generated whenever a particular bumper is pressed or released.
# Note that, despite bumper field on SensorState messages, state field is not a
# bitmask, but the new state of a single sensor.

# bumper
uint8 LEFT   = 0
uint8 CENTER = 1
uint8 RIGHT  = 2

# state
uint8 RELEASED = 0
uint8 PRESSED  = 1

uint8 bumper
uint8 state
```

Compact Message Definition

```
uint8 LEFT=0
uint8 CENTER=1
uint8 RIGHT=2
uint8 RELEASED=0
uint8 PRESSED=1
uint8 bumper
uint8 state
```

Ilustración 42 kobuki_msgs/BumperEvent.msg

4.1.3. Explicación del código

El código entero se adjunta en los anexos (“9.1. Código”), en este apartado se explicará poco a poco este código.

```
def __init__(self): #self representa la instancia del objeto en si mismo

    #Mensaje que obtenemos por pantalla
    rospy.loginfo('Programa choques')

    #Funcion que llamamos si terminamos el programa (ctrl+c)
    rospy.on_shutdown(self.shutdown)

    #Suscriber que lee datos del parachoques
    #Se subscribe al topic '/mobile_base/events/bumper' usando el tipo de mensaje Bump
    #callback es la funcion a la que se llamara al recibir el mensaje en el topic
    rospy.Subscriber('/mobile_base/events/bumper', BumperEvent, self.callback)

    #Publisher que se encarga de los movimientos del robot
    #Va a publicar en el topic cmd_vel_mux/input/navi, encargado del movimiento de la
    #queue_size limita el numero de mensajes en cola si algun subscriptor no esta reci
    self.turtle_vel = rospy.Publisher('cmd_vel_mux/input/navi', Twist, queue_size=10)
```

Ilustración 43 Constructor de objetos de la clase bumpernode() del programa bumper.py

En la ilustración 43 se puede observar la función constructor de los objetos de la clase *bumpernode()*.

La función *rospy.on_shutdown(self.shutdown)* se utiliza para llamar a una función (en este caso la función “*shutdown*”) cuando se termina el programa por medio de recibir un Ctl+C en la terminal.

Tras esto, se crea un subscriptor que se subscribe al topic *‘/mobile_base/events/bumper’* y se le avisa que los mensajes serán del tipo BumperEvent, ya que, como hemos explicado antes, los sensores del parachoques publicarán aquí. Además, *self.callback()* es la función a la que se llamará cada vez que aparezca un mensaje en este topic.

Por último, se crea un publicador, indicándole que va a publicar mensajes del tipo ‘*Twist’* en el topic *‘cmd_vel_mux/input/navi’* y se limita el número de mensajes que puede haber en cola. El robot se mueve en función del contenido de los mensajes que se publiquen este topic, siempre y cuando se haya realizado el arranque de la Turlebot con el Launch *minimal* o *minimal_with_hokuyo* del paquete *Turtlebot_bringup*, el cual crea este topic.

```
#Funcion que se reproduce cada vez que hay una colision
#data es el mensaje que recibe BumperEvent
def callback(self,data):
    if data.state == 1: #Si es un 1, significa que hay colision
        self.colision = 1
        self.actual = time.time()
        if data.bumper == 0: #colision por la izquierda
            print "COLISION IZQUIERDA"
            self.angulo = -15
        elif data.bumper == 1: #colision por el centro
            print "COLISION CENTRAL"
            self.angulo = -45
        elif data.bumper == 2:
            print "COLISION DERECHA" #colision por la derecha
            self.angulo = 15
```

Ilustración 44 Función callback() del programa bumper.py

En la ilustración 44 se muestra la función que es llamada cada vez que se recibe un mensaje en el topic *‘/mobile_base/events/bumper’* de la que se ha hablado antes. Si la variable *state* recibida de este mensaje tiene el valor 1, significa que hay colisión, entonces se cambia la variable *colision* a 1 y se guarda el instante de tiempo en el que se produce esta colisión (variables que se utilizarán después). Tras esto, se comprueba el valor de la variable *bumper* para ver de dónde viene el impacto. En función del lado de la colisión, se cambia el valor de la variable *angulo*, la cual será utilizada para determinar el ángulo de giro del robot.

```
def navegar(self):
    self.move_cmd.linear.x = 0.2
    self.colision_cmd.linear.x = -0.1
    if self.colision == 0:
        self.turtle_vel.publish(self.move_cmd)
    else:
        self.colision_cmd.angular.z = radians(self.angulo)
        self.turtle_vel.publish(self.colision_cmd)
        if time.time()-self.actual>2:
            self.colision = 0
            self.actual = 0
```

Ilustración 45 Función navegar() del programa bumper.py

La ilustración 45 muestra el código de la función *navegar*() de este programa. Esta función es la encargada de hacer que el robot se mueva.

Las variables “move_cmd” y “colision_cmd” son mensajes del tipo Twist. Este tipo de mensaje suele ser encargado del control de la velocidad del robot, ya que tiene dos vectores de tres elementos tipo *float* cada uno. Un vector destinado a la velocidad lineal (desplazarse) y otro para la angular (girar). El contenido de estos vectores son las coordenadas x, y, z.

Entonces, se ha definido que la variable “*move_cmd*” sea la encargada de mover el robot hacia delante, mientras que la variable “*colision_cmd*” se encarga de dar marcha atrás y girar un número de grados en función del tipo de choque.

Si no ha detectado colisión (variable *colision* igual a 0), se publica el contenido de la variable “*move_cmd*” en el topic *‘cmd_vel_mux/input/navi’* del que se ha hablado antes. Esto hace que la Turtlebot vaya hacia delante, ya que el valor de este mensaje es 0.2 en la x de la velocidad lineal y en el resto es 0.

Si se detecta colisión o ha sido hace poco (variable *colision* igual a 1), se publica el contenido de la variable “colision_cmd”, el cual será -0.1 en la x de la velocidad lineal y un valor en la z de la velocidad angular que depende del lado por el que se ha producido la colisión. Esto hace que vaya para atrás y gire.

Cuando se produce una colisión se guarda el momento en el que se produce en la variable “*actual*” como se ha explicado antes. Así el programa comprueba cuanto tiempo ha pasado desde el momento del impacto hasta la situación actual con la diferencia entre este valor y el valor del instante actual. Si son más de dos segundos, se reinicia esta variable a 0 y se pone la variable “*colision*” a 0, la cual se había cambiado a 1 al recibir el impacto, ya que se supone que ya no habrá colisión. Esto se hace para que se publique el contenido de la variable “*colision_cmd*” durante 2 segundos, consiguiendo que

el robot retroceda y rote durante 2 segundos antes de comenzar a ir hacia delante otra vez.

```
def shutdown (self):
    rospy.loginfo("STOP TURTLEBOT")
    print "Saliendo..."
    self.turtle_vel.publish(Twist()) #Paramos el robot
    rospy.sleep(5) #Esperamos a que se detenga para salir.
```

Ilustración 46 Función shutdown() del programa bumper.py

La función *shutdown()*, mostrada en la ilustración 46, es llamada cuando se recibe CTL+C en la terminal, es decir, cuando se requiera terminar el programa.

Se muestra un mensaje por pantalla diciendo que la Turtlebot está parando y se publica en el topic *cmd_vel_mux/input/navi'* el mensaje *Twist*(), el cual es un mensaje tipo *Twist* con todas sus variables a 0, lo que hace que se pare el robot. Tras esto se esperan 5 segundos para salir, para que dé tiempo a que el robot se detenga.

```
if __name__ == '__main__':
    try:
        rospy.init_node('bumpernode',anonymous=True)
        explorer = bumpernode()
        while not rospy.is_shutdown():
            explorer.navegar()
    except:
        rospy.loginfo ("bumpernode node terminated")
```

Ilustración 47 Programa bumper.py

En la ilustración 47 se muestra lo que realiza el programa *bumper.py* al ser ejecutado en una terminal.

Si no hay ningún error (excepciones), con la función *rospy.init_node('bumpernode', anonymous=True)*, se inicia un nodo llamado *bumpernode*. Si no se sabe si ese nombre lo utiliza otro nodo, se pone la variable *anonymous* = *True*. Esto hace que se añada un número al azar detrás del nombre del nodo al crearlo.

Tras esto, se crea un objeto de la clase *bumpernode*() llamado *explorer* y, mientras no se cierre el programa, se va a estar ejecutando en bucle la función *navegar*().

Al iniciar en una terminal el mapeo con el paquete *turtlebot_navigation* que se ha explicado antes, y en otra terminal este programa, se conseguiría un mapeo

de la habitación automático, ya que el usuario puede mapear la habitación con la Turtlebot sin necesidad de controlarla. El usuario tendría que guardar el mapa cuando vea que ha conseguido el mapa objetivo que quería.

4.1.4. Mejoras para el programa bumper.py

El mayor problema que tiene este programa son las esquinas, ya que el robot puede quedarse atrapado en ellas al entrar en un bucle chocando primero con la pared derecha, luego con la pared izquierda y luego otra vez con la pared derecha (Ilustración 48).

El siguiente problema que tendría este programa, es que el robot puede pasar por el mismo sitio muchas veces, ya que el recorrido que hace es aleatorio. Por esto, el proceso de mapear se puede alargar.

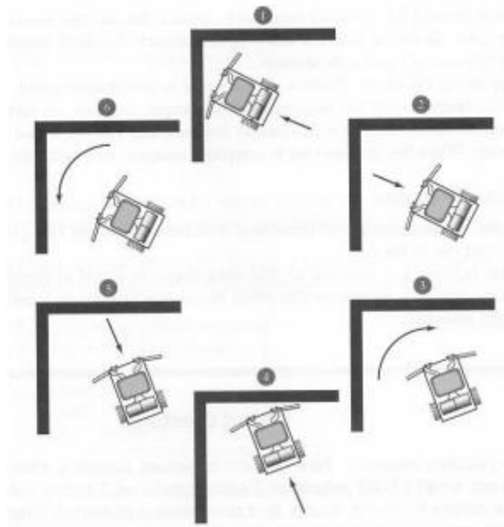

Ilustración 48 Robot atascado en una esquina

Para el problema de las esquinas, se planteó una mejora del programa añadiendo un contador de choques recientes. Se adjunta el diagrama de flujo del programa diseñado en la ilustración 49. En él podemos ver el funcionamiento de este programa. Si hay un choque, el sistema comprueba si ha habido otro choque recientemente con un temporizador. Si no lo ha habido, el programa haría lo mismo que el anterior y luego reiniciaría el temporizador (para calcular el tiempo hasta el siguiente choque) y pondría el contador de choques a 1.

Si hay un choque y ha ocurrido otro recientemente, el sistema comprobaría si han ocurrido 5 choques recientes entre sí, y, si no es así, haría lo mismo que

en el programa anterior añadiendo el reinicio del temporizador e incrementando el contador de choques en 1.

Si hay un choque y el programa detecta que ha habido 5 choques recientes, el sistema mandaría al robot una acción aleatoria de evasión, al darse cuenta de que se encuentra encajado. Esta sería dar marcha atrás y girar un número de grados aleatorio. Tras esto, se reiniciaría el temporizador y el contador de choques se pondría a 0.

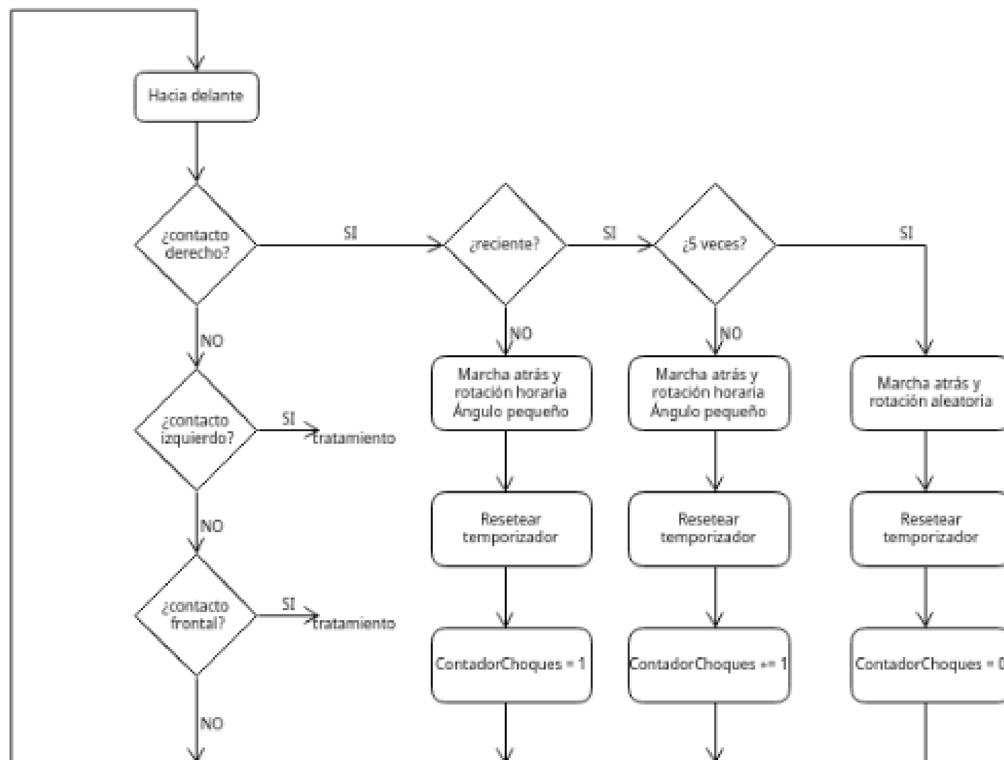

Ilustración 49 Diagrama de flujo del programa "bumper.py" mejorado

Con este programa se conseguía evitar el problema de las esquinas, pero no se llegó a implantar, ya que se encontró un paquete realizado por la comunidad con el que el robot es capaz de mapear el entorno de manera automática, evitando pasar por los lugares que ya ha pasado y con capacidad para no chocarse con las cosas, que era justo lo que se requería. El paquete en cuestión tiene el nombre de "*explore_lite*".

4.1.5. Paquete "explore_lite"

Este paquete realizado por Jiri Horner, permite al robot explorar el entorno sin necesidad de que el usuario esté manejando el robot. Además, evita los espacios cerrados que ya ha mapeado. [30]

Cuando el nodo se está ejecutando, el robot va a explorar el entorno buscando las fronteras desconocidas del mapa a las que pueda llegar. Cuando no haya más fronteras, se para.

Este paquete utiliza el nodo "move_base" para navegar, luego es imprescindible a la hora de utilizar *explore*_lite.

Para poder explorar, el paquete se subscribe al mapa de costo publicado por el nodo "*move_base*" o al mapa generado por algún algoritmo de mapeo (SLAM) y con la información obtenida construye un mapa en el que busca las fronteras (Ilustración 50). Dependiendo del entorno se obtienen mejores resultados si se usa el mapa del algoritmo de mapeo o el mapa de costo publicado por "*move_base*".

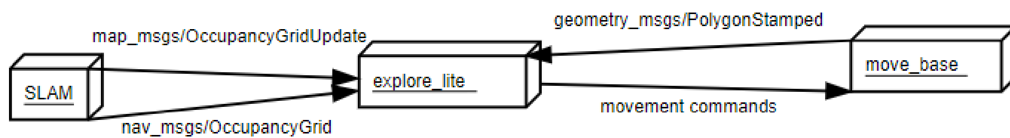

Ilustración 50 Grafo del paquete "explore_lite" [30]

Una de las ventajas que tiene utilizar el mapa de costo publicado por "move_base" es que es capaz de llegar a fronteras muy pequeñas.

Lo que hace este paquete es, una vez se ha lanzado el nodo "move_base", comienza a buscar fronteras en el mapa. Al encontrarlas, envía un punto de destino cercano para que el robot intente llegar a él trazando una trayectoria. Si no consigue llegar, vuelve a buscar fronteras.

Una de las ventajas de este paquete, es que también se puede navegar de forma manual a través de Rviz con "*move_base*", publicando el punto al que se quiere llegar.

Cuando el mapa obtenido sea el que se requiere, el usuario tendrá que guardar el mapa. Si el programa ha funcionado bien y el robot ha sido capaz de llegar a todos los rincones del entorno, el mapa estará completo cuando no detecte más fronteras.

El problema de este paquete es que, en espacios pequeños, el robot detecta colisiones cuando no las tiene. Esto se debe al parámetro "*inflation_radius*", el cual es la distancia mínima que puede haber entre el robot y los diferentes obstáculos. Luego para conseguir que funcione correctamente, habrá que minimizar el valor de este parámetro.

Además, tiene algún fallo, ya que a veces no detecta fronteras cuando sí que las hay. Esto haría que el programa dejase de funcionar porque se pensaría que ya ha terminado, luego lo único que habría que hacer es reiniciarlo. Lo

bueno es que este programa no se encarga de mapear, sino de explorar, así que el mapa que ya tuviera explorado, al reiniciar el programa, no se perdería, y empezaría desde donde se hubiera quedado.

Funciona bien en espacios amplios con los límites del entorno bien marcados y accesibles.

4.2. Sistema de navegación según agenda, llamada de usuario o decisiones del sistema de control.

4.2.1. Descripción del problema

Como objetivo principal de este proyecto, se requería realizar un sistema de navegación autónoma. En este sistema, el robot tiene que ser capaz de navegar a diferentes posiciones en función de una agenda establecida, una llamada del usuario o alguna decisión del sistema de control.

Esta agenda establecida será un fichero con las posiciones guardadas, así como a la hora que hay que ir a ese destino.

Para conseguir guardar en la agenda las posiciones a las que se requiere llegar, se ha escrito un programa a parte que tendrá que ser lanzado cada vez que se requiera guardar una posición del mapa, evitando que se tenga que poner a mano las coordenadas de estas posiciones.

4.2.2. Método de escritura de los destinos en el fichero

Como se ha dicho antes, para escribir los diferentes destinos en un fichero sin tener que meterlos a mano, se ha realizado un programa aparte.

RViz posee una herramienta llamada "*Publish Point*" que, al usarla, si se pulsa en cualquier punto del mapa, publica en el topic */clicked_point* las coordenas de dicho punto. Si este programa no existiera, para conseguir escribir los destinos en el fichero, se podría hacer un "*echo*" de este topic y, pulsando en la posición que queramos, nos aparecerían las coordenadas del punto pulsado. Estas coordenadas habría que meterlas a mano en el fichero. Esto resultaría muy tedioso si necesitamos escribir muchos puntos. Además, no se podría pasar la orientación del punto a no ser que se hallase.

El programa escrito, adjuntado en los anexos, escribe la posición del punto que se requiere alcanzar en el fichero, además de la orientación con la que se requiere llegar y la hora a la que se ha escrito.

Esto se hace debido a que no es posible implantar un cuadro de texto en el que le pases la hora a la que quieres llegar a ese punto. Se podría pedir este dato por la pantalla de la terminal cada vez que se quiera meter un punto en el fichero, pero se ha decidido realizar así para meter primero todos los puntos rápidamente y luego cambiar a mano las horas dentro del fichero. Al estar ya escrito el formato de la hora a la que se ha escrito este punto, solo sería cambiar la hora y los minutos.

Además, si se escribe la palabra "*home*" en el lugar de la posición, el robot vuelve a la estación de carga.

El fichero tendría un contenido como el de la ilustración 51, donde antes del primer punto y coma se escribe la hora a la que hay que ir a la posición indicada; el siguiente número es la coordenada x del punto, el siguiente la coordenada y, y los cuatro siguientes son los elementos del cuaternio que da la orientación, siendo el primer número de ellos la componente escalar del cuaternio y los tres siguientes la componente vectorial.

Si se escribe "home" en lugar de la posición y orientación, el robot irá a la estación de carga a la hora establecida.

Si se están escribiendo las posiciones en el fichero con el programa realizado, en el apartado de la hora saldría la hora a la que se ha escrito el punto. Para poder editarla hay que abrir el fichero y ponerla a mano.

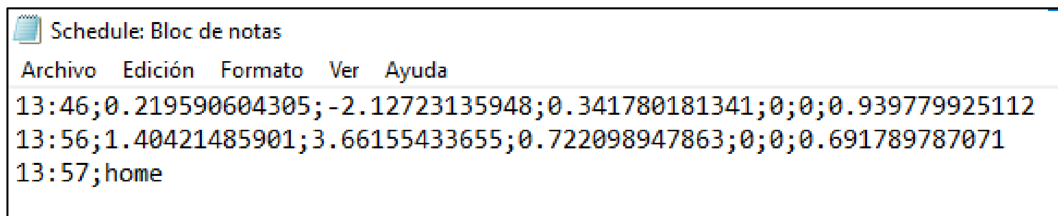

Ilustración 51 Fichero horario con destinos

Para poder escribir la orientación con la que se requiere llegar a ese punto, se ha realizado el programa de manera que, con el primer click estableces el punto al que hay que llegar y con un segundo click estableces la dirección hacia donde tiene que mirar el robot al llegar a ese punto.

4.2.2.1. Explicación del código

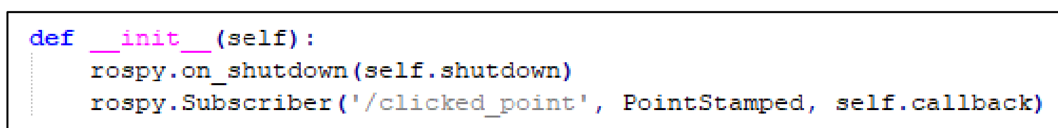
```
def __init__(self):
    rospy.on_shutdown(self.shutdown)
    rospy.Subscriber('/clicked_point', PointStamped, self.callback)
```

Ilustración 52 Constructor de objetos de la clase Meta() en el programa de escritura en el horario

En la función constructor, se define la función a la que se va a llamar al recibir *CTL+C* y se crea un subscriptor al topic *'/clicked_point'* (Ilustración 52). Se define también la función a la que se va a llamar cada vez que se publique un mensaje en ese topic.

Esta función (*self.callback()*), guardará la coordenada x y la coordenada y en la lista "x" y la lista "y" respectivamente. Estas listas son variables globales del objeto y en un inicio estarán vacías.

```
def main(self):
    if(len(self.x) == 2):
        posx = self.x[0]
        posy = self.y[0]
        cuaternio = self.orientacion(self.x[0],self.y[0],self.x[1],self.y[1])
        self.escribefichero(posx,posy,cuaternio)
```

Ilustración 53 main() del programa de escritura en el horario

En la ilustración 53 se expone la función *main*() de este programa, la cual se va a estar ejecutando una vez por segundo hasta que finalice el programa.

En ella se establece que, si el tamaño de la lista global "x" mencionada anteriormente es igual a 2 (lo que quiere decir que se han guardado dos puntos en esta lista, es decir, se ha clicado dos veces en el mapa), se iguala la primera posición de esta lista a la variable que marca la coordenada x del punto y se hace lo mismo con la primera posición de la lista y, igualándola a la variable que marca la coordenada y del punto al que se quiere llegar.

Tras esto se llama a la función que calcula el cuaternio ("*orientacion()*"), a la cual se le pasan las coordenadas de los dos puntos marcados y retorna una lista con las cuatro componentes del cuaternio, y se pasan los datos de la coordenada x, la coordenada y el cuaternio a la función que escribe en el fichero.

```
def orientacion(self,x1,y1,x2,y2):
    quat = []
    v = [x2-x1,y2-y1] #vector de direccion del eje x nuevo
    productoescalar = v[0]*1+v[1]*0 #producto escalar entre la direccion nueva del eje x y la direccion del eje x de la base (1,0)
    mod_v = sqrt(v[0]*v[0] + v[1]*v[1]) #modulo del vector
    ang = acos(productoescalar/mod_v) #angulo entre los dos vectores arccos(productoescalar/producto de modulos) en radianes
    quat = [cos(ang/2),0,0,sin(ang/2)] #cuaternio = [q0,q1,q2,q3] siendo q0 la componente escalar
    self.x[:] = []
    self.y[:] = []
    return quat
```

Ilustración 54 Función que calcula el cuaternio que marca la orientación del robot en el punto final

En la ilustración 54 se puede observar la función que calcula la orientación a partir de dos puntos dados.

Para conseguir realizarla, se ha analizado cómo funciona RViz a la hora de mandar las orientaciones a la Turtlebot. La Turtlebot posee un sistema de coordenadas en el que su eje x marca hacia dónde está mirando.

En la ilustración 55 se puede observar el sistema de coordenadas de la Turtlebot. La flecha negra indica la posición a la que está mirando.

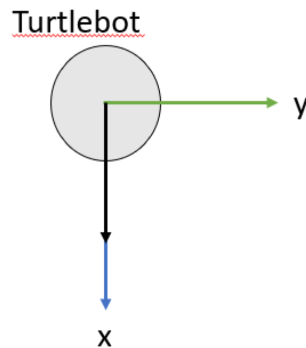

Ilustración 55 Sistema de coordenadas TurtleBot

En un inicio, este sistema de coordenadas coincide con el sistema de coordenadas global. Al rotar la TurtleBot de posición, este sistema ya no coincidiría con el sistema de coordenadas global.

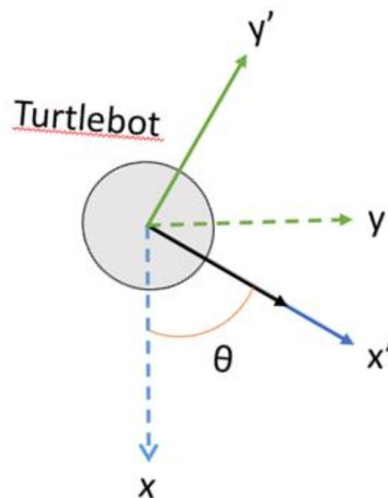

Ilustración 56 Sistema de coordenadas Turtlebot respecto sistema de coordenadas global

En la ilustración 56 se puede observar la rotación del robot Turtlebot respecto al eje z en un valor de θ. Las flechas discontinuas representan el sistema de coordenadas global.

El sistema de navegación calcula la orientación del robot en función del sistema de coordenadas global, con lo que hallando el ángulo que rota el robot con respecto a este sistema de coordenadas, se puede establecer hacia dónde mira el robot (eje x'). Los puntos recibidos por el topic están en la base del sistema de coordenadas global.

Entonces, lo único que hay que hacer es sacar con los dos puntos dados el eje x'. Para ello, se calcula el vector de dirección de este eje x' en la base del sistema de coordenadas global, donde el punto origen es el primer punto clickado y el punto final es el segundo punto. Esto se debe a que el primer click en el mapa establece la posición a la que se requiere ir y el segundo click establece la posición hacia donde se requiere mirar.

Tras esto se halla el ángulo entre el vector x del sistema de coordenadas global y el vector x' (vector x del sistema de coordenadas de la Turtlebot).

Para ello se utiliza el producto escalar entre dos vectores (Ecuación 1)

$$\vec{p} * \vec{q} = |\vec{p}| * |\vec{q}| * \cos\alpha \qquad (1)$$

Donde α es el ángulo que forman (Ilustración 57)

Ilustración 57 Producto escalar

Luego para hallar el ángulo que forman se realiza la ecuación 2:

$$\alpha = \cos^{-1}\left(\frac{\vec{p} * \vec{q}}{|\vec{p}| * |\vec{q}|}\right) \qquad (2)$$

En este caso, el vector $\vec{p}$ es el vector x' en la base del sistema de coordenadas global, hallado con los dos puntos, y el vector $\vec{q}$ es el vector x del sistema de coordenadas global, el cual siempre va a ser (1,0).

Obtenido el ángulo que forman estos dos vectores, faltaría hallar el cuaternio. Un cuaternio define la orientación con el giro de un ángulo θ sobre el vector K de la siguiente manera (ecuación 3):

$$Q = Rot(K,\theta) = \left(\cos\frac{\theta}{2}, k * \sin\frac{\theta}{2}\right) \qquad (3)$$

Siendo k el vector unitario del vector K.

Como en el caso del robot Turtlebot el giro siempre va a ser con respecto el eje Z (0,0,1), el cuaternio se hallará de la siguiente manera (ecuación 4):

$$cuaternio = \left(\cos\frac{\theta}{2}, 0, 0, \sin\frac{\theta}{2}\right) \qquad (4)$$

Siendo θ el ángulo que forma el vector x’ con el vector x.

Tras hallar el cuaternio, se borra la lista ‘*x*’ y la lista ‘*y*’ donde estaban guardadas las coordenadas de los puntos, ya que ya no se van a necesitar, dando paso a recibir otra posición con su orientación.

La función retornará este cuaternio en forma de lista, donde cada posición es una de sus componentes.

```
def escribefichero(self,x,y,cuaternio):
    f = open("puntosmapa.txt", "a")
    f.write(datetime.now().strftime('%H:%M'))
    f.write(";")
    f.write(str(x))
    f.write(";")
    f.write(str(y))
    f.write(";")
    f.write(str(cuaternio[0]))
    f.write(";")
    f.write(str(cuaternio[1]))
    f.write(";")
    f.write(str(cuaternio[2]))
    f.write(";")
    f.write(str(cuaternio[3]))
    f.write("\n")
    f.close()
```

Ilustración 58 Función encargada de escribir el horario en el fichero

Una vez hallada la orientación que se requiere, se escribe en el fichero la hora a la que se ha obtenido el punto y la coordenada ‘x’, la coordenada ‘y’ y el cuaternio de la posición que se requiere alcanzar (Ilustración 58)

4.2.3. Librería “Schedule”

Tras haber conseguido escribir el horario, ahora es necesario estudiar cómo se seguirá.

Para realizar acciones a hora establecidas se ha utilizado la librería “Schedule” de Python.

Esta librería es capaz de ejecutar funciones del programa a horas establecidas. Puede ejecutar funciones cada cierto tiempo de manera periódica, realizar una función cierto día a cierta hora todas las semanas, o marcar plazos para realizar una función (si se pasa el plazo no la realizaría).

Esta librería es perfecta para este proyecto, ya que se ha definido una función exclusiva para que el sistema planee e intente realizar una trayectoria a un punto dado.

Guarda en un objeto la planificación establecida para realizar las diversas funciones y funciona como interrupciones del sistema. Cuando se requiere hacer la tarea, parará cualquier cosa que esté haciendo el programa para realizarla (siempre y cuando el planificador esté activo).

4.2.4. Sistema de navegación

El robot tiene que ser capaz de seguir llegar a posiciones establecidas en función de una agenda dada. Esta agenda es el fichero del que se ha hablado antes, en el cual se escribe la posición y orientación que hay que alcanzar a una hora determinada. A la vez, tiene que ser capaz de moverse a los puntos que le indique el usuario vía RViz.

Además, como mejoras al proyecto se ha añadido que el Robot sea consciente de la batería que le queda, volviendo a la posición de carga si le queda poca y no saliendo de allí hasta que haya recargado al menos un pequeño porcentaje. También se ha utilizado uno de los botones de la Turtlebot de manera que, si se pulsa, el robot vuelve automáticamente a la estación de carga.

Siempre que se requiera volver automáticamente a la estación de carga, el robot va a realizar una trayectoria a un punto cercano de la estación (1 metro delante de ella, por ejemplo). Tras esto, intentará aparcar automáticamente.

Como manejo de excepciones se ha añadido que, si no consigue llegar al punto establecido, emita un pitido de error y vuelva a la estación de carga. Si no consigue volver a la estación, vuelve a emitir el pitido y lo vuelve a intentar. Si no lo consigue en 3 intentos, el programa se cerrará.

Para realizar este sistema de navegación, se requiere de los nodos que son lanzados por el Launch *amcl_demo.launch* del paquete *turtlebot_navigation* para la navegación autónoma del robot en un mapa conocido (Apartado 3.4.8. Navegación autónoma en un mapa conocido) y de los nodos lanzados por el *minimal.launch* (utilizando Hokuyo preferiblemente) del paquete *turtlebot_bringup* (Apartado 3.4.4. Preparar Turtlebot). Además, se necesitará el nodo "*dock_drive*" para el aparcamiento automático (Apartado 3.4.9. Aparcamiento Automático). Como se utiliza la navegación autónoma en un mapa conocido, al ejecutar el nodo *amcl* por primera vez, el robot no sabría en qué posición está (como se ha explicado en el Apartado 3.4.8. mencionado antes). Por lo cual, se ha creado un programa llamado "*posinicial.py*" que, si se ejecuta, indica al nodo *amcl* la posición inicial en la que está el robot (en lugar de tener que usar la flecha de RViz). Este programa solamente tiene sentido si siempre iniciamos el *amcl* desde la misma posición, sino habría que editarlo.

Se ha programado de tal forma que, si se modifica el fichero con el horario establecido, el programa, sin necesidad de reiniciarse, actualiza su horario con lo último escrito en el fichero.

4.2.4.1. Diagrama de flujo

Se van a presentar dos diagramas de flujo, uno para la actualización del horario y otro con los diversos estados del robot.

Para la actualización del horario, el sistema va a estar cada segundo revisando la última fecha de modificación del fichero (excepto cuando esté realizando una tarea, que esperará a que acabe para revisarlo). Si la fecha de modificación es diferente a la de la anterior pasada, el sistema detecta que es necesario actualizar el horario. Para esto, borra todos los trabajos que tenía que realizar con el anterior horario y crea el nuevo con los del fichero actualizado. Así se consigue que solo se tenga que abrir el fichero cuando se sabe de verdad que ha sido modificado (Ilustración 59).

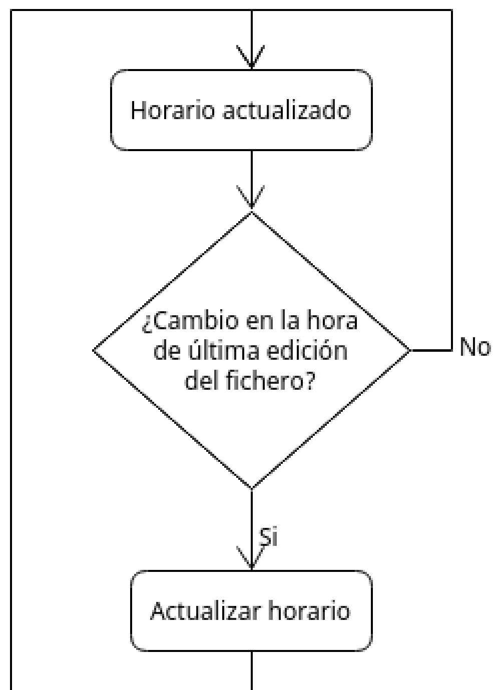

Ilustración 59 Diagrama de flujo para actualizar el horario

Para poder explicar mejor las acciones del robot, se adjunta el diagrama de flujo con sus diferentes estados (Ilustración 60).

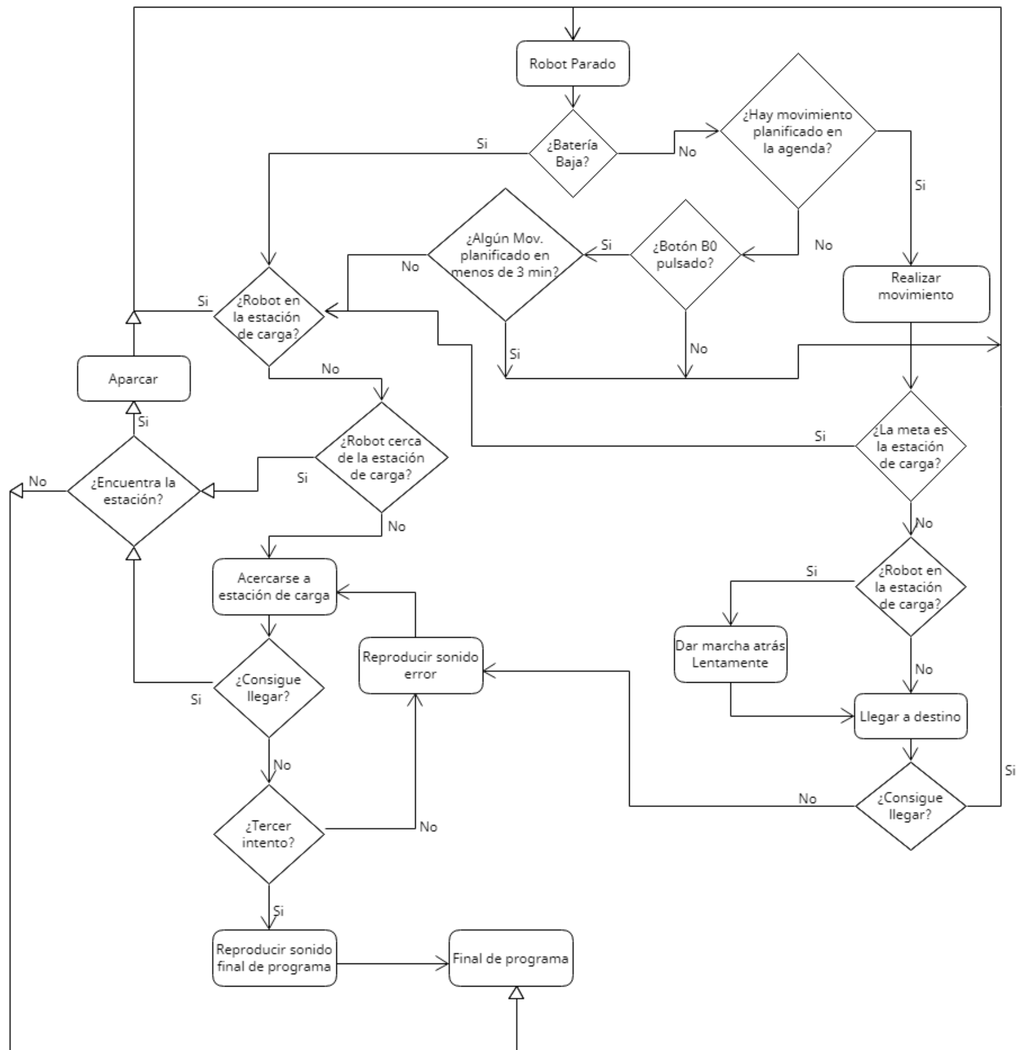

Ilustración 60 Diagrama de flujo de los estados del Robot

El robot cuando está parado se está preguntando todo el rato si tiene batería baja o no. Si la tiene y no está en la estación de carga, intenta volver hasta ella. Si se encuentra en ella, se mantiene en ella, sin permitir que haga algo hasta que la batería supere un porcentaje marcado.

Si no tiene batería baja, el robot puede salir de la estación de carga. Primero comprueba si tiene algún movimiento planificado en la agenda a esa hora. Si lo tiene intenta realizar el movimiento.

Si la posición que se requiere alcanzar es la estación de carga (‘home’ en el fichero, como se ha explicado antes), entonces comprueba si está en la estación de carga ya o no. Si lo está se queda parado, si no lo está intenta volver a la estación.

Si la posición que quiere alcanzar es otra cualquiera, primero comprueba si está en la estación de carga o no. Si lo está, da marcha atrás lentamente para despegarse de ella y tras esto, intenta llegar a la posición. Si no está en la estación de carga, intenta llegar al destino directamente.

Si no consigue llegar al destino, reproduce un sonido indicando un error y se vuelve a la estación de carga. Si lo consigue, se queda parado en el destino esperando la siguiente orden.

Si no tiene batería baja ni movimientos planificados en la agenda a esa hora, puede realizar otras acciones. Pulsar el botón B0 indica al sistema que se requiere la vuelta a la estación de carga.

Si se pulsa y no hay ningún movimiento planificado en la agenda en menos de 3 minutos, el robot comprueba si está en la estación de carga. Si no lo está, intenta volver a la estación de carga, si lo está se queda allí.

Si hay algún movimiento planificado en la agenda en menos de 3 minutos desde que se pulsa el botón, el robot se queda parado, indicando por pantalla que no hace nada porque hay una tarea que se va a realizar en los siguientes tres minutos. Esto se hace porque la librería Schedule trabaja como una interrupción del sistema. Entonces puede pasar que el robot esté volviendo a la base y, antes de conseguirlo, salte una tarea programada anteriormente en el fichero. Esto haría que el sistema pueda fallar, ya que, tras la interrupción, va a seguir por la parte del código donde estaba antes. Es mejor evitar esta situación, impidiendo volver a la base vía el botón B0 si hay una actividad programada en los 3 minutos siguientes a la pulsación del botón.

Cuando se requiera ir a la estación de carga, primero comprobará si está cerca de ella. Si lo está, intentará aparcar. Si lo consigue, el robot se quedará parado en la base. Si no lo consigue, se quedará parado en el sitio en el que esté cuando acabe el tiempo máximo para aparcar. Si no lo consigue debido a que hay obstáculos de por medio y no puede encontrar la estación vía infrarrojos, dará error de sistema y finalizará el programa.

Si no está cerca de la estación, se irá a un punto cercano a ella y después intentará aparcar.

Siempre que se intente volver a la estación de carga y no se pueda, se va a volver a intentar. Si tras tres intentos no se consigue, se reproducirá un sonido indicando el final de programa y lo finalizará.

Además, en todo momento el usuario puede acceder desde el ordenador a controlar el robot, ya sea vía teclado o mandando destinos vía RViz.

4.2.4.2. *Descripción de funciones*

En el apartado de código en los anexos (“9.1. Código”), se encuentran los códigos importantes escritos y comentados. Aquí se va a hacer una descripción de lo que se quiere hacer con cada función.

4.2.4.2.1. posinicial.py

El programa posinicial.py es el encargado de indicar al sistema de navegación dónde se encuentra el robot en un inicio. Este programa se basa en la llamada a la única función que tiene “*pub_posini()*” (ilustración 61)

```
def pub_posini():
    #VARIABLES
    posini_x = 1.7947359085083008
    posini_y = -0.5974310040473938
    orientacionini_z = -0.3667700591974526
    orientacionini_w = 0.9303116271853734

    rospy.init_node('PosicionInicial', anonymous=False)
    posinicial = rospy.Publisher('initialpose', PoseWithCovarianceStamped, queue_size=10)
    pose = PoseWithCovarianceStamped()
    pose.header.frame_id = "map"
    pose.pose.pose.position.x = posini_x
    pose.pose.pose.position.y = posini_y
    pose.pose.pose.position.z = 0.0
    pose.pose.covariance=[0.25, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.25, 0.0, 0.0, 0.0, 0.0, 0.0,
    pose.pose.pose.orientation.x = 0.0
    pose.pose.pose.orientation.y = 0.0
    pose.pose.pose.orientation.z = orientacionini_z
    pose.pose.pose.orientation.w = orientacionini_w
    rospy.sleep(1) #Para que de tiempo a arrancar el publicador
    posinicial.publish(pose)
    print("POSICION INICIAL PUBLICADA")
```

Ilustración 61 pub_posini() del programa posinicial.py

En esta función, se ha guardado como variables las coordenadas y orientación del robot cuando se encuentra en la base de carga en un mapa conocido. Si se utiliza otro mapa o se cambia la base de sitio habría que modificar estos parámetros. Estos valores se han obtenido escuchando el topic “*initialpose*” cuando se utilizaba la herramienta “*2D Pose Estimate*” de RViz para estimar la posición actual del robot cuando se encontraba en la base de carga.

Inicia el nodo “*PosicionInicial*” y crea un publicador capaz de publicar mensajes del tipo “*PoseWithCovarianceStamped*” en el topic “*initialpose*”.

Se crea una variable que guarda el mensaje del tipo “*PoseWithCovarianceStamped*” y se guardan los parámetros requeridos a publicar.

Tras esto espera 1 segundo ya que crear el publicador no es un proceso instantáneo y, si se intenta publicar cuando no está creado, ni publicará ni saldrá un error indicando que no se puede realizar.

Tras esperar este segundo, se publica el mensaje y se escribe por pantalla indicando que la posición inicial se ha publicado.

Esta función es llamada una vez cada vez que se inicia el programa.

4.2.4.2.2. sch_pos_with_battery_v4.py

El programa sch_pos_with_battery_v4.py es la base del sistema de navegación en función de una agenda establecida. Este programa se adjunta entero con comentarios en los anexos.

Para la realización de este programa, se ha intentado que la programación sea lo más modular posible, consiguiendo que, si hay algún fallo, se pueda encontrar rápidamente. Para ello se ha dividido en muchas funciones, las cuales se van a explicar ahora.

```
def __init__(self):

    self.goal_sent = False
    rospy.on_shutdown(self.shutdown)
    self.move_base = actionlib.SimpleActionClient("move_base", MoveBaseAction)
    rospy.loginfo("Esperando al servidor move_base action para empezar")

    if(self.move_base.wait_for_server()):
        rospy.loginfo("Servidor encontrado")

    self.sound = rospy.Publisher('/mobile_base/commands/sound', Sound, queue_size=10)
    rospy.Subscriber("/mobile_base/sensors/core",SensorState,self.SensorStateCallback)
    rospy.Subscriber("/mobile_base/events/button",ButtonEvent,self.ButtonEventCallback)
    rospy.sleep(1)
```

Ilustración 62 Constructor de la clase SchedulePosition() del programa sch_pos_with_battery_v4

En la ilustración 62 se expone la función constructor de objetos de la clase *SchedulePosition*(), los cuales tendrán como métodos todas las funciones de este programa. En el constructor se le indica que no hay ningún destino enviado y se le indica la función a llamar cuando se obtiene un CTL+C de la terminal, siendo esta la denominada *shutdown*().

Tras esto, se crea un cliente para el servidor "*move_base*" y se espera a que este servidor esté creado, ya que va a ser la base de la navegación autónoma.

También se crea un publicador de mensajes tipo "*Sound*" en el topic "*/mobile_base/commands/sounds*" para poder reproducir sonidos desde la Turtlebot.

Por último, se crea un subscriptor al topic "*/mobile_base/sensors/core*" donde se publican los datos de la batería y se le indica la función a la que tiene que llamar cada vez que se publique un mensaje, denominada *SensorStateCallback*(). Además, se crea otro subscriptor al topic "*/mobile_base/events/button*" donde se publican los mensajes relacionados

con los botones, y, al igual que antes, se le indica que la función "*ButtonEventCallback*" es a la que tiene que llamar cada vez que se publique un mensaje en este topic. Se espera un segundo para dar tiempo a que se inicialice todo.

```
def shutdown(self):
    if self.goal_sent:
            self.move_base.cancel_goal()
    rospy.loginfo("Ctrl-C caught. Quitting")
    self.ReproducirSonido(1)
    rospy.sleep(1)
```

Ilustración 63 Función shutdown() del programa sch_pos_with_battery_v4

La función *shutdown*() (ilustración 63) es llamada cuando se recibe un CTL+C desde la terminal.

Si hay un destino enviado, se cancela, se envía por pantalla un mensaje diciendo que se para el programa y llama a la función *ReproducirSonido()* con el argumento 1. Esta función reproduce sonidos dependiendo el argumento que se le pase. Con lo cual, la Turtlebot reproduce un sonido de apagado. Se espera un segundo a que se pare todo y se cierra el programa.

```
def LeerFichero(self):
    Schedule = []
    tarea = []
    f = open("/home/tb2/catkin_ws/src/scheduleposition/scripts/Schedule.txt") #Ruta donde va
    lineas = f.readlines() #lineas es una list con cada linea del fichero en cada posicion
    f.close()
    for line in lineas:
        frase = line.strip() #Borra espacios y \n al principio y al final del string
        if (frase != ""): #Si no es una linea en blanco, guarda la tarea. Para evitar errores
            frase = frase.replace(" ","") #Por si se ponen espacio entre medias, para evitar
            hora = frase.partition(";")[0] #Divide la frase en tres posiciones siendo la 0 la
            posicion = frase.partition(";")[2]
            if (posicion == "home"): #Si lee home, se vuelve a la estacion
                x = self.home_x
                y = self.home_y
                q0 = float("1")
                q1 = float("0")
                q2 = float("0")
                q3 = float("0")
            else:
                x = float(posicion.partition(";")[0])
                y = float(posicion.partition(";")[2].partition(";")[0])
                orientacion = posicion.partition(";")[2].partition(";")[2]
                q0 = float(orientacion.partition(";")[0])
                q1 = float(orientacion.partition(";")[2].partition(";")[0])
                q2 = float(orientacion.partition(";")[2].partition(";")[2].partition(";")[0])
                q3 = float(orientacion.partition(";")[2].partition(";")[2].partition(";")[2])
            tarea.append(hora)
            tarea.append(x)
            tarea.append(y)
            tarea.append(q0)
            tarea.append(q1)
            tarea.append(q2)
            tarea.append(q3)
            Schedule.append(tarea) #Matriz de tareas
            tarea = [] #Una vez guardada la tarea, se borra para meter una nueva
    return Schedule
```

Ilustración 64 Función LeerFichero() del programa sch_pos_with_battery_v4

La función *LeerFichero*() de la ilustración 64 es de las más importantes del programa. En esta, se abre el fichero con la agenda establecida, se guarda línea a línea en una lista de Python (vector), siendo cada línea una posición de esta lista, y se cierra el fichero.

Tras esto, para cada posición de la lista se borran los espacios del inicio y del final para evitar errores.

Entonces, si la posición no está vacía, lo que querría decir una línea en blanco en el fichero, se borran los espacios que pueda haber en el medio de las líneas por algún fallo del usuario al escribirlas, evitando errores a la hora de leerlas. Solo borra un espacio, si se han puesto más daría errores.

Tras esto, se empieza a descomponer la línea, la cual está dividida en puntos y comas. Lo que hay dentro de cada punto y coma se mete en un *string*, que en el caso de las coordenadas y las orientaciones se convierten en una variable *float*.

Con la línea ya dividida, se meten en una lista denominada tarea las variables obtenidas, siendo la primera posición la hora en formato *string*, seguida de la coordenada 'x', la coordenada 'y' y los elementos del cuaternio.

Esta lista, se mete en otra lista denominada *Schedule*, consiguiendo realizar una lista de listas en el cual cada línea es una tarea del horario. Esta tarea tiene dividido en diferentes posiciones las diferentes variables requeridas para navegar a una determinada hora.

Una vez metida la tarea en la lista *Schedule*, se borra la tarea y se pasa a hacer lo mismo con la siguiente línea del fichero, añadiendo la siguiente tarea en la siguiente posición de la lista *Schedule*.

Cuando termina con el fichero, retorna la lista Schedule, consiguiendo una matriz donde las filas son las tareas a realizar y las columnas son la hora a la que se requiere ir al destino, la coordenada 'x', la coordenada 'y' y los elementos del cuaternio del destino.

```
def ult_modificacion(self):
    estado = os.stat("/home/tb2/catkin_ws/src/scheduleposition/scripts/Schedule.txt")
    fecha = time.localtime(estado.st_mtime)
    fecha = datetime(fecha[0], fecha[1], fecha[2], fecha[3], fecha[4], fecha[5])
    return fecha
```

Ilustración 65 Función ult_modificacion() del programa sch_pos_with_battery_v4

Para conseguir saber si se ha realizado alguna modificación en el fichero sin tener que abrirlo todo el rato, se ha definido la función ult_*modificacion*() de la ilustración 65. Con esta, gracias a las librerías *os* y *time*, se puede retornar la fecha de la última modificación de un fichero.

```
def goto(self, pos, quat):
    if not pos['x'] == self.home_x and not pos['y'] == self.home_y:
        self.BackUp_DockStation()
    self.BorrarCostmaps() #Para borrar obstaculos que pueden haber desaparecido
    rospy.loginfo("Go to (%s, %s) pose", pos['x'], pos['y'])
    rospy.loginfo("Quaternion (%s, %s, %s, %s) ",quat['r1'], quat['r2'], quat['r3'], quat['r4'])
    self.goal_sent = True
    goal = MoveBaseGoal()
    goal.target_pose.header.frame_id = 'map'
    goal.target_pose.header.stamp = rospy.Time.now()
    goal.target_pose.pose = Pose(Point(pos['x'], pos['y'], 0.000), Quaternion(quat['r1'], quat['r2'], quat['r3'], quat['r4']))
    if pos['x'] == self.home_x and pos['y'] == self.home_y:
        result = self.GoToDockStation()
    else:
        self.move_base.send_goal(goal)
        success = self.move_base.wait_for_result(rospy.Duration(60))
        state = self.move_base.get_state()
        result = False
        if success and state == GoalStatus.SUCCEEDED:
            result = True
        else:
            self.move_base.cancel_goal()
            cmd_vel = rospy.Publisher('cmd_vel_mux/input/navi', Twist, queue_size=10)
            cmd_vel.publish(Twist())
            rospy.loginfo("Imposible llegar al punto deseado")
            self.ReproducirSonido(4)
            rospy.sleep(5)
            self.GoToDockStation()
    return result
```

Ilustración 66 Función goto() del programa sch_pos_with_battery_v4

La función *goto(pos, quat*) (ilustración 66) es otra de las más importantes del programa. Al ser llamada, primero, si la posición a la que quiere llegar no es la

base de carga, llama a la función *BackUp_DockStation()* que se encarga de salir de la estación de carga si el robot esta allí. Tras esto, se escribe el destino al que se quiere llegar, el cual ha sido pasado por argumentos.

Si la posición a la que quiere llegar es la estación de carga, llamará a la función *GoToDockStation*() que se explicará después. Si no, envía el destino al servidor "*move_base*", haciendo que intente ir hasta él. Si lo consigue, se devuelve un True, si no lo consigue, se cancela el destino, se para la Turtlebot, se envía mensaje de error por pantalla, se reproduce un sonido y se llama a la función *GoToDockStation*() para volver a la estación. Se devuelve un false.

```
def planificar(self, horario):
    i = 0
    while i<len(horario):
        position = {'x': horario[i][1], 'y': horario[i][2]}
        quaternion = {'r1' : horario[i][4], 'r2' : horario[i][5], 'r3' : horario[i][6], 'r4' : horario[i][3]}
        schedule.every().day.at(horario[i][0]).do(self.goto, position, quaternion)
        i = i+1
```

Ilustración 67 Función planificar() del programa sch_pos_with_battery_v4

La función *planificar*() (ilustración 67) es la encargada de añadir el horario que retorna la función *LeerFichero*() al objeto *schedule* (planificador) de la librería *schedule* explicada en el apartado "4.2.3. Librería "Schedule"". Aquí consigue que se cree un planificador en el que todos los días a la hora establecida en el horario se realice la función *goto*(), a la que se la pasan como argumentos la posición y el cuaternio que hay en el horario. Hace esto con todas las tareas que hay en el horario.

Esto es posible ya que la librería *schedule* va guardando las tareas sin sobreescribirlas. La única manera de eliminar las tareas guardadas del planificador es con la llamada de la función *schedule*.*clear*()

```
def SensorStateCallback(self, data):
    #data.battery esta variando continuamente entre dos valores, luego solo actualiza el valor si hay una diferencia de mas de 2
    if(math.fabs(int(data.battery) - self.nivel_bateria_anterior) > 2):
        rospy.loginfo("Nivel de bateria: " + str(round((float(data.battery)-float(self.battery_dangerous)) / (float(self.max_carga)-float
        self.nivel_bateria_anterior = int(data.battery)

    if(int(data.charger) == 0): #Si es un 0, significa que se esta descargando, luego no esta cargando.
        self.DockStation = False #lo que significa que no estaria en la estacion
    else:
        self.DockStation = True

    #Si la bateria disminuye por debajo del 10% de carga, significa que la bateria es baja
    if (round((float(data.battery)-float(self.battery_dangerous)) / (float(self.max_carga)-float(self.battery_dangerous)) * 100) < 10) :
        if(not self.LowBattery):
            rospy.loginfo("BATERIA BAJA")
            self.LowBattery = True #Bateria baja

    #Hasta que no este por encima del 20% no dejamos que salga de la estacion
    #Asi evitamos que se ponga a cargar y al instante salga
    if ((round(float(data.battery)-float(self.battery_dangerous)) / (float(self.max_carga)-float(self.battery_dangerous)) * 100) > 20) :
        if(self.LowBattery):
            rospy.loginfo("Puede salir de la estacion")
            self.LowBattery = False #Ya no hay bateria baja
```

Ilustración 68 Función SensorStateCallback() del programa sch_pos_with_battery_v4.py

En la ilustración 68 se muestra la función que es llamada cuando se reciben mensajes en el topic "*/mobile_base/sensors/core*" con los datos de la batería.

La batería (parámetro "*battery*" del mensaje) está variando continuamente entre dos valores, luego para actualizar el valor en porcentaje de la batería, solo se realizará cuando la variación entre estos dos valores sea mayor de dos.

Para calcular el valor de la batería en tanto por ciento, se realizará la siguiente cuenta (Ecuación 5):

$$\frac{carga\ de\ batería\ actual - carga\ de\ batería\ mínima}{carga\ de\ bateria\ máxima - carga\ de\ batería\ mínima} * 100 \qquad (5)$$

Donde la carga de batería máxima es la tensión que tiene cuando está el robot cargado al máximo (162) y la carga de batería mínima es la tensión a partir de la cual el robot se para (133).

Si el parámetro "*charger*" del mensaje es 0, significa que el robot se está descargando, lo que quiere decir que no está en la estación de carga.

Si el porcentaje de batería es menor que el 10% y no estaba en baja batería ya, se cambia la variable que marca que la batería es baja y se manda un mensaje por pantalla advirtiéndolo.

Si el porcentaje de batería es mayor del 20% y la variable marcaba que había poca batería, se cambia el valor de esta variable indicando que ya hay suficiente batería para realizar acciones.

```
def BackUp_DockStation(self):
    if(self.DockStation == True):
        rospy.loginfo("Robot en estacion. Dando marcha atras antes de empezar")
        self.DockStation = False
        self.home = False
        cmd_vel = rospy.Publisher('cmd_vel_mux/input/navi', Twist, queue_size=10)
        #Twist es el tipo de mensaje
        move_cmd = Twist()
        #Marcha atras a  0.1 m/s
        move_cmd.linear.x = -0.1
        move_cmd.angular.z = 0
        r = rospy.Rate(10);
        temp_count = 0
        #Marcha atras durante 30 segundos
        while (not rospy.is_shutdown() and temp_count < 300):
            cmd_vel.publish(move_cmd)
            #Espera hasta 0.1 s (10 HZ) y publica otra vez
            temp_count = temp_count + 1
            r.sleep()
            #Nos aseguramos de que para publicando un Twist()
        cmd_vel.publish(Twist())
        return True
```

Ilustración 69 Función BackUp_Dockstation() del programa sch_pos_with_battery_v4

Siempre que se requiera ir a algún destino y el robot esté en la base de carga, antes de nada, tendrá que dar marcha atrás lentamente para salir de la

estación y no llevársela consigo. Para ello se creado la función *BackUp_DockStation*() de la ilustración 69.

Esta función cuando es llamada comprueba si el robot está en la base de carga. Si está, crea un publicador para el topic "*cmd_vel_mux/input/navi*" encargado de los movimientos del robot. Publica la velocidad a la que quiere dar marcha atrás durante el tiempo que se imponga (en este caso 30 segundos). Tras esto, se para la Turtlebot.

Esto se hace para alejar el robot de la base de carga y no tener elementos cerca que puedan perjudicar al robot haciéndole creer que está atascado, luego la cantidad de tiempo que hay que dar marcha atrás dependerá de cada sala y de donde se ponga esta estación.

```
def GoToDockStation(self):
    #Primero nos aproximamos a la base
    if self.DockStation:
        rospy.loginfo("Ya estamos en la base")
        result = True
    else:
        while (self.home == False):
            if(self.intentos<3):
                rospy.loginfo("Going HOME")
                self.BorrarCostmaps()
                goal = MoveBaseGoal()
                goal.target_pose.header.frame_id = 'map'
                goal.target_pose.header.stamp = rospy.Time.now()
                goal.target_pose.pose = Pose(Point(self.home_x, self.home_y, 0.000), Quaternion(0.000,0.000, 0.000, 1.000))
                self.move_base.send_goal(goal)
                success = self.move_base.wait_for_result(rospy.Duration(60))
                state = self.move_base.get_state()
                result = False
                if success and state == GoalStatus.SUCCEEDED:
                    result = True
                    self.home = True
                    self.intentos = 0
                else:
                    self.move_base.cancel_goal()
                    cmd_vel = rospy.Publisher('cmd_vel_mux/input/navi', Twist, queue_size=10)
                    cmd_vel.publish(Twist())
                    self.intentos +=1
                    self.ReproducirSonido(4)
                    rospy.loginfo("Numero intentos: %d", self.intentos)
            else:
                self.move_base.cancel_goal()
                cmd_vel.publish(Twist())
                result = False
                rospy.signal_shutdown("ERROR. Imposible llegar a HOME")
```

Ilustración 70 Primera parte de la función GoToDockStation() del programa sch_pos_with_battery_v4

```
#Si ha llegado cerca de la base, aparcamos
self._client = actionlib.SimpleActionClient('/dock_drive_action', AutoDockingAction)
rospy.loginfo("Esperando al servidor auto_docking")
self._client.wait_for_server() #espera hasta encontrar el servidor por si no se ha e
rospy.loginfo("Servidor auto_docking encontrado")
goal = AutoDockingGoal()
rospy.loginfo("APARCANDO... (cancelando si pasa de 180s)")
self._client.send_goal(goal) #envia el destino que es la base de carga

#Le damos 180seg para aparcar
success = self._client.wait_for_result(rospy.Duration(180))
if success:
    rospy.loginfo("Aparcamiento conseguido")
    self.DockStation = True #Estamos en la base
    return True
else:
    self._client.cancel_goal() #cancela el aparcamiento
    rospy.loginfo("Fallo en el aparcamiento")
    return False
```

Ilustración 71 Segunda parte de la función GoToDockStation() del programa sch_pos_with_battery_v4

La función *GoToDockStation*() es la encargada de volver a la estación de carga. Se puede dividir en dos partes (se podrían haber hecho dos funciones diferentes), la primera (ilustración 70) en la cual el robot va a intentar llegar a una posición cercana a la estación de carga y la segunda (ilustración 71) en la cual el robot va a intentar aparcar automáticamente.

Nada más ser llamada, esta función comprueba si ya se encuentra en la base, y si ya está ahí, imprime un mensaje por pantalla indicándolo y no hace nada más. Si no está ahí, borra los mapas de costo e intenta llegar a un punto cercano a la estación de carga que se ha establecido como variable (para poder cambiarlo si se cambia de sitio la estación o se utiliza otro mapa). Para esto, utiliza el mismo método que la función *goto(pos,quat)* explicada antes. Lo único diferente es que, si no lo consigue, se reproduce otro sonido y se incremente el número de intentos en uno y se vuelve a intentar. Cuando el número de intentos supera los tres, el robot se da por vencido, cancelando todo y reproduciendo un sonido diferente al resto. Tras esto cierra el programa.

Si se ha conseguido llegar a la posición cercana a la base en alguno de esos intentos, se llama al servidor encargado de aparcar automáticamente (explicado en el apartado “3.4.9. Aparcamiento automático”).

Si el servidor está activo, comienza a aparcar automáticamente. Al conseguirlo, reinicia el número de intentos a 0 y retorna un True. Si no lo consigue en menos de 180 segundos, retorna un false e indica fallo de aparcamiento, quedándose en la posición en la que se encuentre.

```
def NeedCharge(self):
    #Si la bateria es baja pero estamos en la base, nos quedamos  hasta que este suficientemente cargado
    if(self.DockStation and self.LowBattery):
        rospy.loginfo("Robot en la base de carga")
        rospy.loginfo("Esperando a que este suficientemente cargado")
        time.sleep(30)
        return True

    #Si la bateria es baja y no estamos en la base, va a ella
    if(not self.DockStation and self.LowBattery):
        rospy.loginfo("Bateria baja. Yendo a la base de carga")
        self.GoToDockStation()
        return True

    return False
```

Ilustración 72 Función NeedCharge() del programa sch_pos_with_battery_v4

En la ilustración 72 se expone la función *NeedCharge*(), la cual es llamada cada segundo. Esta indica si el robot necesita volver a la estación de carga porque tiene batería baja o si puede salir de ella porque ya está suficientemente cargada.

```
def ButtonEventCallback(self, data):
    if (data.state == ButtonEvent.PRESSED):
        if (data.button == ButtonEvent.Button0):
            if (schedule.idle_seconds() <= 180): #Segundos hasta la siguiente tarea
                #Esto lo hago porque las tareas del schedule entran al programa parando cualquier cosa
                #que este haciendo.  Si esta yendo a la base y salta una tarea, parara y realizara la tarea
                rospy.loginfo("QUEDAN MENOS DE 3 MINUTOS PARA LA SIGUIENTE TAREA")
                rospy.loginfo("HASTA QUE NO LA REALICE, NO PUEDE IR A HOME POR SEGURIDAD")
            else:
                if(not self.DockStation):
                    rospy.loginfo("Yendo a la base...")
                    self.GoToDockStation()
                else:
                    rospy.loginfo("Ya estamos en la base")
```

Ilustración 73 Función ButtonEventCallBack() del programa sch_pos_with_battery_v4

La función *ButtonEventCallBack*() es la llamada cada vez que se presiona o se despulsa un botón (ilustración 73). En ella, si el botón ha sido presionado y este botón es el botón 0, va a intentar volver a la base de carga, excepto si faltan menos de 3 minutos para realizar una tarea previamente programada en el horario.

Si ya estuviera en la base de carga, mostraría un mensaje por pantalla diciendo esto y no haría nada.

```
def BorrarCostmaps(self):
    rospy.wait_for_service('/move_base/clear_costmaps')
    clear_costmaps = rospy.ServiceProxy('/move_base/clear_costmaps', Empty)
    if(clear_costmaps.call()):
        rospy.loginfo('GLOBAL COSTMAPS BORRADOS')
    else:
        rospy.loginfo('Error al llamar al servicio /move_base/clear_costmaps')
```

Ilustración 74 Función BorrarCostmaps() del programa sch_pos_with_battery_v4

La ilustración 74 muestra la función *BorrarCostmaps*(). Esta función borra los mapas de costo creados. Se ha tenido que implementar ya que, si ha habido un obstáculo que no le ha dejado pasar por un sitio, lo guarda como si hubiera una pared en el mapa de costo. Entonces, si le mandan ir a un destino en el

que es necesario traspasar ese obstáculo, y no consigue ver si el obstáculo sigue existiendo o no desde la posición en la que está, no intentaría ni ir, quedándose quieto. Por ello, a esta función se la llama cada vez que se requiere ir a algún destino.

```
def ReproducirSonido(self, num):
    self.sound.publish(num)
    rospy.sleep(5)
```

Ilustración 75 Función ReproducirSonido() del programa sch_pos_with_battery_v4

La función *ReproducirSonido(num),* como su propio nombre indica, es la encargada de que el robot reproduzca sonidos (ilustración 75). Para ello publica en el topic */mobile_base/commands/sounds* el valor del número que le pasan como argumento. Estos valores suponen un sonido diferente cada uno, siendo posibles del 0 al 6.

```
if __name__ == '__main__':
        try:
            rospy.init_node('SchedulePosition', anonymous=False) #inicia el nodo
            navigator = SchedulePosition() #crea objeto
            horario = navigator.LeerFichero() #guarda en horario lo que hay dentro d
            primeramod =  navigator.ult_modificacion() #guarda el valor de la ultima
            navigator.planificar(horario) #planifica
            while not rospy.is_shutdown(): #mientras no se cierre, entra en bucle
                ultmod = navigator.ult_modificacion()  #comprueba la fecha actual de
                if (ultmod != primeramod): #si es diferente con la guardada anterior
                    navigator.ficheromodificado = True #el fichero ha sido modficiad
                    primeramod = ultmod #guardamos la ultima fecha de modificacion
                    print("Fichero modificado")
                else:
                    navigator.ficheromodificado = False #sino, no ha sido modificado
                if (not navigator.NeedCharge() and not navigator.ficheromodificado):
                    schedule.run_pending() #comienza a realizar tareas en funcion de
                else:  #si el fichero ha sido modificado o necesita bateria
                    print("Actualizando horario")
                    schedule.clear() #borra el plan anterior
                    horario = navigator.LeerFichero() #lee el fichero
                    navigator.planificar(horario) #vuelve a planificarlo
                time.sleep(1)


        except rospy.ROSInterruptException:
                rospy.loginfo("Ctrl-C caught. Quitting")
```

Ilustración 76 Cuerpo del programa sch_pos_with_battery_v4

En la ilustración 76 se expone el comportamiento del programa al ser ejecutado como programa principal.

Si no hay excepciones, primero se crea el nodo "*SchedulePosition*".

Tras esto se crea un objeto de la clase *SchedulePosition*() y se ejecuta la función *LeerFichero*() con ese objeto. La matriz de tareas que retorna, explicada antes, se guarda en la variable horario.

Esta variable se le pasa como argumento a la función *planifica*() de ese objeto, creando el planificador.

Tras esto, mientas no se cierre el programa, el programa va a estar en bucle comprobando que la última modificación del fichero no ha cambiado. Si ha cambiado significa se ha modificado el fichero, por lo que hay que actualizar el horario.

Si el robot no necesita cargarse (comprobándolo con la función *NeedCharge*()) y el horario está actualizado (no ha habido cambios en el fichero), se ejecuta el planificador con todas las tareas que tenga guardadas.

Si el fichero no está actualizado o necesita carga, se borran todas las tareas del planificador, se vuelve a leer el fichero y se guardan otra vez.

5. Capítulo 5. Puesta en marcha y resultados.

En este capítulo se va a exponer la puesta en marcha del software desarrollado, así como un análisis de los resultados obtenidos.

5.1. Exploración

5.1.1. Puesta en marcha bumper.py

Suponiendo que la comunicación con el ROS Máster se ha configurado bien, en primer lugar, para poder mapear utilizando el programa *bumper.py* realizado, se necesita arrancar la Turtlebot. Para ello se utilizará el comando explicado en el apartado "3.4.4. Preparar Turtlebot" con el que se consigue inicializar el sensor LiDAR:

```
$ roslaunch turtlebot_bringup minimal_with_hokuyo.launch
```

Tras esto, en otro terminal se ejecuta el archivo launch con el que se consigue mapear con el LiDAR, explicado en el apartado "3.4.7. Construir un mapa (SLAM)":

```
$ roslaunch turtlebot_navigation gmapping_demo_hokuyo.launch
```

Por último, en otra terminal se ejecuta el script bumper.py (ya que no se implantó en ningún paquete) con el siguiente comando:

```
$ python /ruta/bumper.py
```

Con esto se conseguiría empezar a mapear la sala, consiguiendo que el robot se mueva de manera aleatoria sin necesidad de la presencia del usuario. Por pantalla aparecerían las diversas colisiones indicando el lado por el que se producen (Ilustración 77).

```
[INFO] [1656499964.488712]: Programa choques
COLISION IZQUIERDA
COLISION CENTRAL
COLISION IZQUIERDA
COLISION CENTRAL
COLISION CENTRAL
COLISION CENTRAL
^C[INFO] [1656499983.077265]: STOP TURTLEBOT
Saliendo...
```

Ilustración 77 Terminal programa bumper.py

Para poder visualizar el mapa que se va dibujando, habría que escribir en una terminal en el ordenador que se pueda ejecutar RViz lo siguiente:

```
$ roslaunch turtlebot_rviz_launchers view_navigation.launch
```

Y para guardar el mapa, en otra terminal, se debe ejecutar el siguiente comando, sin haber cerrado la terminal en la que se había lanzado el archivo launch de mapeo:

```
$ rosrun map_server map_saver -f /tmp/my_map
```

Estos dos últimos explicados en el apartado “3.4.7. Construir un mapa (SLAM)”.

5.1.2. Resultados bumper.py

Los resultados obtenidos con este programa fueron satisfactorios para el inicio del proyecto. Se consiguió una forma de que el robot mapeara la sala de manera automática.

Los principales problemas que tenía este programa eran las esquinas, quedarse encerrado en una habitación mucho tiempo hasta conseguir salir (ya que el movimiento era pseudo aleatorio) y que podía pasar por zonas ya mapeadas. Además, el robot se pone en marcha hacia delante y solo se da cuenta de que hay un objeto delante si toca el parachoques, pudiendo ser un problema si hay algo elevado.

Soluciones a estos problemas han sido expuestas en el apartado “4.1.4. Mejoras para el programa bumper.py”

5.1.3. Puesta en marcha “explore_lite”

Como se ha comentado en el apartado “4.1.5. Paquete “explore_lite””, se descubrió este paquete realizado por la comunidad capaz de explorar un entorno desconocido mientras realiza un mapa. Con este paquete se solucionaban todos los problemas del programa “*bumper.py*”. Tras modificar

este paquete para implementarlo en el robot del proyecto, se puede comenzar la puesta en marcha.

Para comenzar a mapear, al igual que antes, hay que abrir dos terminales, una para arrancar el robot y otra para arrancar los nodos encargados de mapear:

```
$ roslaunch turtlebot_bringup minimal_with_hokuyo.launch
$ roslaunch turtlebot_navigation gmapping_demo_hokuyo.launch
```

Tras esto, se ejecuta el Launch "*explore.launch*" del paquete "*explore_lite*":

```
$ roslaunch explore_lite explore.launch
```

Para poder visualizar el mapa creado y conseguir guardar el mapa se realizaría como con el programa "*bumper.py*".

5.1.4. Resultados "explore_lite"

Este paquete ha aportado justo lo que se requería para el proyecto. El robot es capaz de reconocer las partes del mapa que tiene sin cerrar, es decir, las fronteras abiertas, e intentar llegar hasta ellas. Además, utiliza el nodo "*move_base*" de la pila de navegación de ROS para conseguir moverse, lo cual hace que consiga sortear obstáculos sin necesidad de chocarse.

El principal problema que se encuentra a la hora de utilizar este paquete es la gran cantidad de tiempo que le toma encontrar las fronteras abiertas e intentar llegar hasta ellas, ya que tiene que intentar encontrar una trayectoria para llegar con cada frontera. Si existen fronteras que no son alcanzables, el programa se ralentiza ya que no es capaz de llegar y, al usar el nodo "*move_base*" para moverse, realizaría los mecanismos de recuperación mencionados en el apartado "3.3.4.3. Recovery behaviors" para intentar llegar a estos puntos.

Además, en espacios estrechos el robot se puede quedar atascado, al creer que, si se mueve, colisionaría contra algo.

A pesar de estos problemas, el resultado es bastante satisfactorio, ya que en un principio se quería realizar un programa por el cual un robot "tonto" mapease la sala moviéndose de manera aleatoria. Al final se ha conseguido que el robot sea inteligente, reconociendo las partes del mapa que le quedan por descubrir y llegando a ellas.

5.2. Sistema de navegación por agenda

5.2.1. Puesta en marcha

Para conseguir ejecutar el sistema de navegación autónoma a lugares establecidos en función de una agenda, es necesario haber arrancado la Turtlebot y los nodos encargados de la navegación autónoma en un mapa conocido. Luego lo primero de todo es realizar un mapa del entorno, por lo que el software implantado para la exploración y mapeo de lugares desconocidos viene muy a cuento.

Tras tener un mapa, se arranca la Turtlebot y el sensor LiDAR:

```
$ roslaunch turtlebot_bringup minimal_with_hokuyo.launch
```

Lo siguiente sería ejecutar los nodos encargados de la navegación autónoma en un entorno conocido. Para ello se utiliza el siguiente comando, explicado en el apartado "3.4.8. Navegación autónoma en un mapa conocido":

```
$ roslaunch turtlebot_navigation amcl_demo_hokuyo.launch map_file:
=/ruta/mymap.yaml
```

Para visualizar la posición en el mapa del robot, así como las trayectorias que va a realizar, habría que utilizar el siguiente comando en el ordenador que pueda ejecutar RViz:

```
$ roslaunch turtlebot_rviz_launchers view_navigation.launch
```

En un primer instante, habría que posicionar al robot en el sitio en el mapa donde se encuentra en ese momento. Para ello se podría utilizar la herramienta "2D Pose Estimate" de RViz, consiguiendo estimar la posición a base de pruebas, como se ha explicado en el apartado "3.4.8. Navegación autónoma en un mapa conocido".

Para no tener que hacer esto así, se creó el nodo "*posinicial.py*". En este se ha guardado la posición en el mapa realizado donde se encuentra la Turtlebot si está en la base de carga y al ejecutarse la publica indicando que el sitio donde se encuentra el robot es la base de carga. Luego para poder utilizarlo y que funcione bien, el robot tiene que estar en la base de carga y utilizar el mapa realizado del que se han sacado las coordenadas del robot cuando está en la base. Si la base de carga se mueve de sitio, habría que modificar estas coordenadas.

Para poder ejecutarlo se utiliza el siguiente comando:

```
$ rosrun scheduleposition posinicial.py
```

Tras publicar la posición inicial, se cierra el programa automáticamente.

El siguiente paso sería realizar el horario. Para ello se puede ejecutar el nodo "clickedpoint.py" con el comando:

```
$ rosrun scheduleposition clickedpoint.py
```

Una vez ejecutado, con RViz abierto con el mapa, se utiliza la herramienta "*Publish Point*" para guardar los puntos a los que se requiere ir en el fichero "Schedule.txt". Como se ha explicado en el apartado "4.2.2. Método de escritura de los destinos en el fichero", el primer click es el destino final y el segundo click será la posición hacia donde mirará el robot. Este nodo se seguirá ejecutando hasta que se pare manualmente con CTL+C o cerrando la terminal.

Tras esto, habría que modificar el fichero "Schedule.txt" cambiando la hora que pone por defecto por la hora que se requiera ir a cada posición.

También se podría realizar este fichero a mano, pero es más tedioso.

Una vez realizado todo esto, se podría comenzar a ejecutar el nodo realizado para la navegación en función de la agenda. Para ello se ha realizado un archivo Launch en el que se ejecuta el nodo que activa el servidor para el aparcamiento automático (ya que se le va a llamar cuando se requiera volver a la base de carga) y el nodo realizado. El contenido se muestra en la ilustración 78.

```
<launch>
  <!-- KOBUKI_AUTO_DOCKING minimal.launch -->

  <node pkg="nodelet" type="nodelet" name="dock_drive" args="load kobuki_auto_docking/AutoDockingNodelet mobile_base_nodelet_manager">
    <rosparam file="$(find kobuki_auto_docking)/param/auto_docking.yaml" command="load"/>
    <remap from="dock_drive/odom" to="odom"/>
    <remap from="dock_drive/core" to="mobile_base/sensors/core"/>
    <remap from="dock_drive/dock_ir" to="mobile_base/sensors/dock_ir"/>
    <remap from="dock_drive/motor_power" to="mobile_base/commands/motor_power"/>
    <remap from="dock_drive/velocity" to="mobile_base/commands/velocity"/>
  </node>

  <!-- SCHEDULEPOSITION -->

  <node pkg="scheduleposition" name="sch_pos_with_battery" type="sch_pos_with_battery_v4.py" output="screen">
  </node>
</launch>
```

Ilustración 78 minimalv2.launch del paquete scheduleposition

Para ejecutarlo, se escribiría el siguiente comando:

```
$ roslaunch scheduleposition minimalv2.launch
```

Con esto, el programa estaría iniciado. Es compatible con la teleoperación vía teclado o vía RViz, pero siempre da prioridad a las tareas de la agenda.

5.2.2. Resultados

Tras haber sometido a numerosas pruebas al programa, intentando hacer que falle para encontrar sus inconsistencias, se ha llegado a unos resultados altamente satisfactorios.

Gracias a este programa el robot es capaz de navegar a destinos previamente especificados en una agenda, además de por llamadas del usuario vía RViz o teleoperación y de por decisiones de sistema de control, como la vuelta a la estación si la batería es baja.

Además, se han analizado todos los posibles fallos que puede haber, creando una vía de escape si estos suceden.

A parte, la creación de los programas "*posinicial.py*" y "*clickedpoint.py*" consiguen que la preparación previa para utilizar este programa sea menos tediosa.

El mapa utilizado para estas pruebas se llama "*UltimateMap.yaml*", el cual se puede observar en la ilustración 79. Este mapa se ha creado con el software de exploración y mapeo de entornos desconocidos implementado en este proyecto.

Se ha cumplido con creces el objetivo propuesto para este proyecto.

Ilustración 79 Mapa utilizado en el proyecto

6. Capítulo 6. Gestión del trabajo.

En este apartado se va a realizar un estudio de la gestión del trabajo, es decir, la planificación que se ha seguido a la hora de llevar a cabo este proyecto, así como un estudio del coste económico que supone su elaboración.

6.1. Planificación del proyecto

Para poder exponer la planificación del proyecto para alcanzar los diferentes objetivos, se ha realizado un diagrama de Gantt (Ilustración 80). Este diagrama se ha agrupado en diferentes fases del proyecto que se van a explicar ahora, pero no se ha querido subdividir más porque quedaba un diagrama muy extenso y difícil de visualizar.

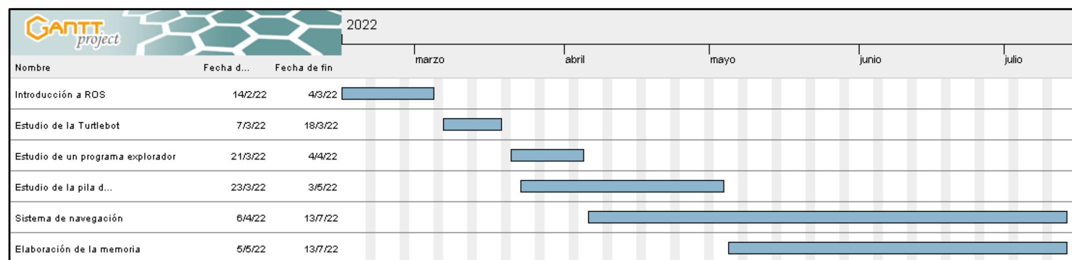

Ilustración 80 Planificación del proyecto

Las primeras semanas se comenzó con una preparación para hacer frente al proyecto. Se empezó con la introducción al entorno de programación ROS, con el aprendizaje de los conceptos y la realización de tutoriales.

Tras comprender bien esto, se comenzó el estudio de la Turtlebot2, donde se realizaron diferentes tutoriales y se analizó el Hardware y los diversos topics que conectaban el Hardware con ROS.

Teniendo conocimientos sobre ROS y el robot, la quinta semana se empezó a investigar la realización de un programa explorador que mapee de forma automática, lo que conllevó, poco después del inicio, a estudiar la pila de navegación de ROS. Tras realizar el programa bumper.py e intentar implementar diversos paquetes de la comunidad, se eligió el paquete *explore_lite*, ya explicado en el apartado 4.1.5. Paquete “explore_lite“.

Tras esto, se comenzó a realizar el sistema de navegación en base a una agenda, a la vez que se analizaba la pila de navegación de ROS cada vez que surgía algún problema.

Se inició realizando un programa que leyese una posición de un fichero e intentase llegar hasta ella. Luego, se hizo una segunda versión consiguiendo que leyese del fichero las posiciones y las horas a las que había que llegar a estos sitios, planificase un horario y realizara las acciones correspondientes. En la tercera versión se añadió que el robot fuera consciente de la batería que le quedaba y que si tenía poca volviera a la base de carga. Una vez finalizado esto, se añadieron las excepciones, controlando los movimientos del robot si no se consiguiera llegar a una posición establecida (se eligió que volviera a la base). Como último aporte se escribió el script que inicializa la posición inicial en el mapa.

También se realizó el programa para guardar los puntos del mapa en el fichero. En un principio solo tenía en cuenta las coordenadas del punto final y llegaba con cualquier orientación. Luego se escribió otra versión que conseguía tener en cuenta esta orientación.

Mientras tanto, ya se había comenzado a escribir la memoria.

La duración del proyecto total ha sido de 20 semanas. Sumando un total de unas 600 horas aproximadamente.

6.2. Estudio económico

6.2.1. Recursos empleados

A continuación, se muestra un resumen de los componentes hardware y software empleados en el desarrollo del proyecto:

- Software:
  - Sistema operativo: Ubuntu 20.04, Ubuntu 16.04 y Windows 10
  - Pack de Microsoft Office
  - Entorno de programación “Robot Operating System”
- Hardware:
  - Robot TurtleBot2
  - Brazo robótico WidowX
  - Sensor LiDAR Hokuyo URG-04LX
  - Controlador Intel NUC 7I5BNK
  - Ordenador fijo HP Compaq 8100 Elite i5.

- Material ofimático.

6.2.2. Costes directos

Los costes directos son aquellos que se asignan de manera clara a un proyecto, se asocian directamente con un producto terminado o su elaboración. Estos pueden ser el coste de material, costes amortizables de equipos y coste del personal.

6.2.2.1. Coste del personal

La realización del presente proyecto ha sido llevada a cabo por un estudiante de ingeniería, encargado de programar el sistema de navegación del robot y de escribir la memoria, bajo la supervisión de dos ingenieros.

Se estima el sueldo anual teórico del estudiante como si de un ingeniero júnior se tratara, teniendo en cuenta el sueldo medio anual de éste en España, para analizar el coste de su trabajo en función de las horas empleadas. Este coste incluye:

- Sueldo bruto anual, así como los posibles incentivos por su trabajo.
- Cotización a la Seguridad Social, que es un 35% del sueldo bruto.

Teniendo en cuenta esto, el coste anual del ingeniero será de 34.931,25 € (Tabla 6.1).

COSTE ANUAL	
Sueldo bruto más incentivos	25.875,00 €
Seguridad Social (35% sueldo bruto)	9.056,25 €
Coste total	34.931,25 €

Tabla 6.1. Coste anual del personal

Se calcula una estimación de los días efectivos trabajados en un año. (Tabla 6.2).

DÍAS EFECTIVOS POR AÑO	
Año medio	365,25 días
Sábados y Domingos	-104,36 días
Días de vacaciones efectivos	-20,00 días
Días festivos reconocidos	-15,00 días
Días perdidos estimados	-5,00 días
Total días efectivos estimados	**220,89 días**

Tabla 6.2. Días efectivos por año

Conociendo ya el número total de días efectivos de trabajo, y que la jornada laboral es de 8 horas, obtenemos el total de horas efectivas de trabajo:

$$220{,}89\ días/año\ x\ 8\ horas/día\ =\ \mathbf{1.767{,}12\ horas/año}$$

El coste por hora de un ingeniero se calcula como la división del sueldo anual entre las horas efectivas trabajadas al año:

$$\frac{Coste}{hora} = \frac{34.931{,}25\ \frac{€}{año}}{1.767{,}12\ \frac{horas}{año}} \approx \mathbf{19{,}77€/hora}$$

Sabiendo que la duración del proyecto ha sido de unas **600** horas, se puede calcular el coste personal directo como la multiplicación de las horas del proyecto y el coste efectivo de una hora de trabajo del ingeniero:

$$600\ horas\ x\ 19{,}77\ €/hora\ =\ \mathbf{11.860{,}40€}$$

COSTE PERSONAL DIRECTO	11.860,40€

6.2.2.2. Coste de amortización de equipos y programas

Para el cálculo de estos costes se debe realizar previamente la inversión total y calcular la amortización lineal correspondiente según los criterios aconsejados por la ley. En este apartado se estudiará los costes de amortización de los equipos de trabajo utilizados.

Se estima como tiempo de amortización un periodo de 4 años, ya que es el considerado como vida útil de dicho material. De forma que al calcular el coste hay que multiplicar por un factor del 0.25 los precios mostrados (Tabla 6.3).

MATERIAL	IMPORTE (aprox.)	AMORTIZACIÓN 25%
Sistema operativo Windows 10	145,00 €	36,25 €
Ordenador HP Compaq 8100 Elite i5	350,00 €	87,5 €
Controlador Intel NUC 7I5BNK	650,00 €	162.5 €
Total material	**1145,00 €**	**286,25 €**

Tabla 6.3. Amortización del material

Los programas utilizados de libre distribución no se han apuntado ya que su coste es 0.

Al precio de la amortización anual por los diversos equipos y sistema operativo, hay que añadir el precio anual del paquete Microsoft Office 365 que es de **69,00 €.**

Luego el coste anual de los equipos y programas asciende a **355,25 €**

El coste final por hora de utilización del material es calculado mediante la división de los costes anuales entre el número de horas de uso en dichos equipos.

$$\frac{Coste}{hora} = \frac{355{,}25€\text{ año}}{1.767{,}12\ horas\text{ año}} \approx \mathbf{0{,}21€/hora}$$

El tiempo de uso de estos materiales es igual al tiempo total necesario para la realización del proyecto, por ser necesario en las etapas de análisis, programación y documentación. Por tanto, el coste de amortización de material será:

$$600\ horas\ x\ 0{,}21\ €/hora\ =\ \mathbf{126{,}00}\ €$$

COSTE DE AMORTIZACIÓN DE MATERIAL DE OFICINA	126,00 €

6.2.2.3. Costes del material utilizado

En este apartado se tienen en cuenta los costes del material utilizado para el desarrollo del proyecto.

En primer lugar, se calculan los gastos producidos por la compra de los componentes Hardware del proyecto. Estos bienes no se han incluido como bienes amortizables suponiendo que se utilizarán solo para este proyecto (Tabla 6.4).

MATERIAL	COSTE
Robot TurtleBot2	1.450,00 €
Brazo robótico WidowX	1.769,00 €
Sensor Hokuyo URG-04LX	1.900,00 €
Total	**5.119,00 €**

Tabla 6.4. Costes del material utilizado (Hardware)

A estos gastos, hay que añadir costes derivados de otros materiales incluyendo cuadernos, bolígrafos, papel de impresora, cartuchos de tinta, etc.

El coste total estimado de este material es de **90€**

Lo que supone un coste total del material utilizado de **5.209,00€.**

COSTE TOTAL DEL MATERIAL UTILIZADO	**5.209,00 €**

6.2.2.4. Costes directos totales

Agregando todos los costes que se han ido obteniendo, los costes directos totales se obtienen como la suma de los costes de personal, la amortización de los equipos y programas, y el coste del material utilizado. Por tanto, será:

$$11.860{,}40 \; + \; 126{,}00 \; + \; 5.209{,}00 \; = \mathbf{17.195{,}40} \; €$$

COSTES DIRECTOS	**17.195,40 €**

6.2.3. Costes indirectos

Los costes indirectos son los gastos producidos por la actividad requerida para la elaboración del proyecto y que no se pueden incluir en ninguno de los gastos directos. No son directamente imputables a la producción de un bien. Comprenden gastos como el de la telefonía o la electricidad (Tabla 6.5).

COSTES INDIRECTOS PARCIALES	
Dirección y servicios administrativos	120,00 €
Consumo de electricidad	180,00 €
Consumo de telefonía	25,00 €
Total gastos indirectos	**325,00 €**

Tabla 6.5. Costes indirectos

Por lo tanto, los costes indirectos totales ascienden a:

COSTES INDIRECTOS	325,00 €

6.2.4. Costes totales

Los costes totales son el resultado de sumar los gastos directos y los indirectos, siendo el total de este proyecto los mostrados en la siguiente tabla 6.6.

COSTES TOTALES	
Costes directos	17.195,40 €
Costes indirectos	325,00 €
Coste total del proyecto	**17.520,40 €**

Tabla 6.6. Costes totales

En resumen, el coste total del proyecto asciende a la cantidad de:

COSTES TOTALES DEL PROYECTO	17.520,40 €

7. Capítulo 7. Conclusiones y líneas de trabajo futuras.

En este capítulo se realizará un juicio crítico sobre los resultados obtenidos, sobre el proyecto en general y sobre los objetivos planteados en el capítulo 1. Además, se indicarán propuestas de trabajo futuro que no han sido llevadas a cabo en este proyecto ya que se quedaban fuera del alcance de este.

7.1. Conclusiones generales

Lo primero y más importante, es que se han logrado superar con éxito todos los objetivos propuestos al inicio del proyecto. Esto resulta significativo debido a que se inició sin tener ningún conocimiento sobre robótica móvil y sobre ROS, el cual es la base de este proyecto.

Con este proyecto se ha logrado conseguir una gran formación en ROS, aprendiendo las bases de este entorno de programación para desarrollo de aplicaciones robóticas, además de conseguir gran soltura a trabajar sobre una terminal de Linux y sus comandos. A mayores, se ha conseguido ampliar los conocimientos en lenguajes de programación como son Python y C++.

También, el estudio del robot Turtlebot2 ha conllevado un aumento en los conocimientos sobre hardware común para los robots móviles (por ejemplo, el sensor LiDAR), además del aprendizaje de conceptos como la odometría. Este estudio ha hecho conocer el alcance de posibilidades de funcionamiento que puede otorgar este robot.

A lo largo del proyecto han ido surgiendo problemas, los cuales se han conseguido superar investigando la raíz de estos y consiguiendo solucionarlos.

Ha sido un proyecto interesante y enriquecedor a la hora del conocimiento adquirido, considerándolo una gran forma de complementar mi formación como ingeniero electrónico y abriendo un mundo de posibilidades con la robótica móvil.

7.2. Conclusiones sobre los objetivos planteados

El objetivo principal de este proyecto era realizar un sistema de navegación en el cual el robot consiguiera llegar a destinos establecidos en función de una agenda, de la llamada del usuario o de decisiones del sistema de control.

Este objetivo se ha cumplido con creces, ya que a mayores de esto se han añadido más funcionalidades, como es el manejo de excepciones si no consigue llegar al punto indicado o que esté atento a la batería que le queda.

Respecto a los objetivos intermedios que se impusieron al inicio del proyecto, se adjunta una tabla con los resultados obtenidos (Tabla 7.1).

Requerimiento	¿Objetivo obtenido?
Familiarización con el entorno de programación ROS	Si
Comprender la pila de navegación de ROS	Si
Aprender conceptos sobre el robot Turtlebot2	Si
Conseguir que el robot sea capaz de mapear el entorno	Si
Desarrollar un sistema de planificación para que el robot pueda dirigirse a ciertas ubicaciones a una hora determinada	Si

Tabla 7.1 Objetivos cumplidos

Ya no sólo se ha conseguido una familiarización con el entorno de programación de ROS, sino que se ha obtenido un gran conocimiento sobre este. Además de comprender de manera minuciosa su pila de navegación, tras haber hecho frente a una gran cantidad de problemas en la navegación autónoma del robot.

Se ha hecho un estudio amplio de los conceptos del robot Turtlebot2, tanto de su Hardware como de su Software.

Se ha conseguido que el robot sea capaz de mapear el entorno y, además, se ha conseguido que este mapeo sea automático, sin necesidad de que el usuario esté atento a él.

El desarrollo del sistema de navegación en función de una agenda planificada se ha cumplido perfectamente, estudiando todos los fallos que pueden cometerse y proponiendo una solución.

7.3. Propuestas de trabajo futuro

Al ser el inicio de un proyecto, este puede tener una gran cantidad de líneas de trabajo futuro. Por lo cual, se ha realizado una programación altamente escalable, con la mayor parte del código comentada para evitar dudas.

La principal sería conseguir una interfaz de usuario mejor a la hora de guardar los destinos y las horas en el horario, ya que la actual es bastante tediosa. Se estudió durante varios días la posibilidad de conseguir esto añadiendo alguna herramienta a RViz por la cual se pudiera añadir un display con la que se pasase la hora y el punto elegido o por medio de un servidor web que realice esto, pero se salía del alcance del proyecto. Se intentó realizar una interfaz con la librería de Python "*tkinter*", pero no fue convincente del todo al no poder añadir un mapa.

El primer paso hacia una interfaz mejor podría ser mejorar el programa que guarda los puntos en el fichero, haciendo que, el programa, una vez el usuario haya elegido el destino, pregunte al usuario la hora a la que quiere ir a ese sitio. Tras escribirla, el programa podría guardar en el fichero la hora y el destino.

Otra posible línea de trabajo futuro es la implementación de un servidor web para el manejo del robot Turtlebot, de manera que no haya que utilizar la terminal para realizar movimientos o mapear. Además, tras realizar este servidor, se podría estudiar lo de mejorar la interfaz que se ha propuesto antes.

Por otro lado, se podría hacer un estudio del brazo robótico, de su puesta en marcha y posibles funciones. Se podría plantear el cambio de la cámara Astra de sitio a estar encima de este brazo robótico, de manera que viese lo que está en frente de él. De esta forma, se podrían realizar programas basados en visión artificial para que la pinza pueda agarrar objetos que hay en frente. El cambio de posición de la cámara no sería un problema, ya que para mapear y navegar es capaz de utilizar el sensor LiDAR. El único fallo posible es la limitación de precisión que tendrá este brazo robótico, al estar formado por servomotores.

Además, otra propuesta de trabajo futuro sería implementar los programas en Python 3.

Todas estas posibles líneas de trabajo futuro podrían ser buenas opciones para trabajos de fin de grado de futuros estudiantes.

8. Bibliografía

[1] M. Ben-Ari, Elements of Robotics, ISBN 978-3-319-62532-4, Springer, 2017.

[2] E. B. Kuipers, «Shakey: From Conception to History,» 2017. [En línea]. Available: http://ai.stanford.edu/~nilsson/OnlinePubs-Nils/General%20Essays/Shakey-aimag-17.pdf. [Último acceso: Mayo 2022].

[3] «TurtleBot,» [En línea]. Available: https://www.turtlebot.com/turtlebot2/. [Último acceso: Junio 2022].

[4] «O'Reilly,» [En línea]. Available: https://www.oreilly.com/library/view/ros-programming-building/9781788627436/9ddba456-a610-4f81-a0ef-80ac7ccceee8.xhtml. [Último acceso: Junio 2022].

[5] A. Polanco Masa, «Tecnología Obsoleta,» 2015. [En línea]. Available: https://alpoma.net/tecob/?p=11359. [Último acceso: Junio 2022].

[6] «Kawasaki Robotics,» [En línea]. Available: https://kawasakirobotics.com/eu-africa/company/history/?wovn=es. [Último acceso: Junio 2022].

[7] «Museum of Transport and Technology,» [En línea]. Available: https://collection.motat.nz/objects/104032/industrial-robot-irb6-allmanna-svenska-elektriska-aktiebolaget-asea. [Último acceso: Junio 2022].

[8] «Proyecto Idis,» [En línea]. Available: https://proyectoidis.org/shakey/. [Último acceso: Mayo 2022].

[9] Oxford Languages, Diccionario Oxford Spanish, ISBN 978-0199543403, 2008.

[10] «International Federation of Robotics,» [En línea]. Available: https://ifr.org/service-robots. [Último acceso: Junio 2022].

[11] «WikiROS, Distribuciones,» [En línea]. Available: http://wiki.ros.org/Distributions. [Último acceso: Junio 2022].

[12] «TurtleBot,» [En línea]. Available: https://www.turtlebot.com/about/. [Último acceso: Junio 2022].

[13] «WikiROS, Introducción,» [En línea]. Available: http://wiki.ros.org/ROS/Introduction. [Último acceso: Junio 2022].

[14] «WikiRos, Conceptos,» [En línea]. Available: http://wiki.ros.org/es/ROS/Conceptos. [Último acceso: Junio 2022].

[15] «WikiROS, Mensajes,» [En línea]. Available: http://wiki.ros.org/msg. [Último acceso: Junio 2022].

[16] L. Joseph, Mastering ROS for Robotics Programming, ISBN 978-1788478953, Packt, 2015.

[17] M. S. H. Achmad, «Tele-Operated Mobile Robot for 3D Visual Inspection Utilizing Distributed Operating System Platform,» 2017.

[18] M. Ferber, «MARVINFERBER BLOG,» 2017. [En línea]. Available: https://marvinferber.net/?p=128. [Último acceso: Junio 2022].

[19] «Dabit Industries,» [En línea]. Available: https://dabit.industries/products/iclebo-kobuki. [Último acceso: Junio 2022].

[20] Yujin Robot, «Guía de Usuario Kobuki,» [En línea]. Available: http://kobuki.yujinrobot.com/wiki/online-user-guide/. [Último acceso: Junio 2022].

[21] «Intel NUC,» [En línea]. Available: https://www.intel.es/content/www/es/es/products/details/nuc.html. [Último acceso: Junio 2022].

[22] «ROS Components,» [En línea]. Available: https://www.roscomponents.com/es/camaras/76-orbbec.html. [Último acceso: Junio 2022].

[23] «HOKUYO,» [En línea]. Available: https://hokuyo-usa.com/products/lidar-obstacle-detection/urg-04lx. [Último acceso: Junio 2022].

[24] «WikiROS, PhantomX Reactor Arm,» [En línea]. Available: http://wiki.ros.org/phantomx_reactor_arm. [Último acceso: Junio 2022].

[25] «Trossen Robotics,» [En línea]. Available: https://www.trossenrobotics.com/p/phantomx-ax-12-reactor-robot-arm.aspx. [Último acceso: Junio 2022].

[26] «Pila de navegación ROS,» [En línea]. Available: http://wiki.ros.org/navigation. [Último acceso: Junio 2022].

[27] R. M. D. Silva, «Modelo del sistema de odometría de un robot,» [En línea]. Available: https://www.researchgate.net/figure/Figura-2-Modelo-do-sistema-de-odometria-do-robo_fig2_343646784. [Último acceso: Junio 2022].

[28] J. Zhang, 2020. [En línea]. Available: https://matheecs.tech/study/2020/01/08/move-base.html. [Último acceso: Junio 2022].

[29] «WikiROS, AutoAparcamiento,» [En línea]. Available: http://wiki.ros.org/kobuki/Tutorials/Automatic%20Docking. [Último acceso: Junio 2022].

[30] «Explore_lite,» [En línea]. Available: http://wiki.ros.org/explore_lite. [Último acceso: Junio 2022].

[31] M. Quigley, B. Gerkey y W. D. Smart, Programming Robots With ROS, ISBN 9781449323899, O'Reilly Media, 2015.

[32] J. M. O’Kane, A Gentle Introduction to ROS, ISBN 9781492143239, Columbia, South Carolina, 2016.

9. Anexos

9.1. Código

9.1.1. Scripts

9.1.1.1. bumper.py

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

import rospy #Para escribir un nodo de ROS en pythoni
import time
from std_msgs.msg import String #para usar el tipo de mensaje
String
from geometry_msgs.msg import Twist #para usar el tipo de mensaje
Twist (velocidad)
from kobuki_msgs.msg import BumperEvent #para usar el tipo de
mensaje BumperEvent (sensores de presion)
from math import radians

class bumpernode():

      colision = 0
      actual = 0
      angulo = 0
      move_cmd = Twist()
      colision_cmd = Twist()

    def __init__(self): #self representa la instancia del objeto
en si mismo

       #Mensaje que obtenemos por pantalla
        rospy.loginfo('Programa choques')

       #Funcion que llamamos si terminamos el programa (ctrl+c)
        rospy.on_shutdown(self.shutdown)

       #Suscriber que lee datos del parachoques
       #Se subscribe al topic '/mobile_base/events/bumper' usando
el tipo de mensaje BumperEvent
       #callback es la funcion a la que se llamara al recibir el
mensaje en el topic
        rospy.Subscriber('/mobile_base/events/bumper',
BumperEvent, self.callback)

       #Publisher que se encarga de los movimientos del robot
        #Va a publicar en el topic cmd_vel_mux/input/navi,
encargado del movimiento de la turtlebot, el tipo de mensaje Twist
       #queue_size limita el numero de mensajes en cola si algun
subscriptor no esta recibiendo los mensajes suficientemente rapido
        self.turtle_vel =
rospy.Publisher('cmd_vel_mux/input/navi', Twist, queue_size=10)
```

```
    def navegar(self):
        self.move_cmd.linear.x = 0.2 #velocidad publicada si no
hay choque
        self.colision_cmd.linear.x = -0.1 #velocidad publicada si
hay choque
        if self.colision == 0:
            self.turtle_vel.publish(self.move_cmd) #si no hay
choque va hacia delante
        else:
            self.colision_cmd.angular.z = radians(self.angulo)
            self.turtle_vel.publish(self.colision_cmd) #si hay
choque va hacia atras y gira
            if time.time()-self.actual>2: #esto lo hace durante
2seg desde el choque
                        self.colision = 0
                        self.actual = 0

      #Funcion que se reproduce cada vez que hay una colision
        #data es el mensaje que recibe BumperEvent
    def callback(self,data):
        if data.state == 1: #Si es un 1, significa que hay
colision
                  self.colision = 1
                  self.actual = time.time() #Guarda el instante
del choque
                  if data.bumper == 0: #colision por la izquierda
                        print "COLISION IZQUIERDA"
                        self.angulo = -15
                  elif data.bumper == 1: #colision por el centro
                        print "COLISION CENTRAL"
                        self.angulo = -45
                  elif data.bumper == 2:
                        print "COLISION DERECHA" #colision por la
derecha
                        self.angulo = 15

        def shutdown (self):
            rospy.loginfo("STOP TURTLEBOT")
            print "Saliendo..."
                self.turtle_vel.publish(Twist()) #Paramos el robot
                rospy.sleep(5) #Esperamos a que se detenga para
salir.

if __name__ == '__main__':
      try:
            rospy.init_node('bumpernode',anonymous=True)
            explorer = bumpernode()
            while not rospy.is_shutdown():
                  explorer.navegar() #navega en bucle mientras no
paremos el programa
      except:
            rospy.loginfo ("bumpernode node terminated")
```

9.1.1.2. clickedpoint.py

```
#!/usr/bin/env python
import rospy
from geometry_msgs.msg import PointStamped
from datetime import datetime
from math import *

class Meta():
	x = []
	y = []

#Constructor
	def __init__(self):
		rospy.on_shutdown(self.shutdown)
		rospy.Subscriber('/clicked_point', PointStamped,
self.callback) #subscriptor a los mensajes del topic clicked_point

#Funcion encargada de generar el cuaternio con la orientacion
	def orientacion(self,x1,y1,x2,y2):
		quat = []
		v = [x2-x1,y2-y1] #vector de direccion del eje x nuevo
		productoescalar = v[0]*1+v[1]*0 #producto escalar
entre la direccion nueva del eje x y la direccion del eje x de la
base (1,0)
		mod_v = sqrt(v[0]*v[0] + v[1]*v[1]) #modulo del vector
		ang = acos(productoescalar/mod_v) #angulo entre los
dos vectores arccos(productoescalar/producto de modulos) en
radianes
		quat = [cos(ang/2),0,0,sin(ang/2)] #cuaternio =
[q0,q1,q2,q3] siendo q0 la componente escalar
		self.x[:] = [] #borra el vector x para los siguientes
puntos
		self.y[:] = [] #borra el vector y para los siguientes
puntos
		return quat

#Funcion que es llamada cuando se recibe un mensaje del topic
clicked_point
	def callback(self,msg):
		rospy.loginfo("coordinates:x=%f y=%f"
%(msg.point.x,msg.point.y))
		self.x.append(msg.point.x) #Guarda la posicion x en el
vector x
		self.y.append(msg.point.y) #Guarda la posicion y en el
vector y

#Funcion encargada de escrbir en el fichero
	def escribefichero(self,x,y,cuaternio):
		f =
open("/home/tb2/catkin_ws/src/scheduleposition/scripts/Schedule.tx
t", "a") #Abre el fichero de la ruta en modo escritura
		f.write(datetime.now().strftime('%H:%M')) #Escribe
fecha actual en formato HORA:MINUTOS
		f.write(";")
		f.write(str(x))
		f.write(";")
		f.write(str(y))
```

```
            f.write(";")
            f.write(str(cuaternio[0]))
            f.write(";")
            f.write(str(cuaternio[1]))
            f.write(";")
            f.write(str(cuaternio[2]))
            f.write(";")
            f.write(str(cuaternio[3]))
            f.write("\n")
            f.close()

#Funcion que va a ser llamada en bucle
      def main(self):
            if(len(self.x) == 2): #Si el tamaño de la lista x es 2
significa que se han obtenido dos puntos
                  posx = self.x[0] #la posicion es el primer punto
                  posy = self.y[0]
                  cuaternio =
self.orientacion(self.x[0],self.y[0],self.x[1],self.y[1]) #con los
dos puntos se haya el cuaternio
                  print cuaternio
                  self.escribefichero(posx,posy,cuaternio) #lo
escribe en el fichero

#Funcion llamada cuando se cierra el programa
      def shutdown(self):
                rospy.loginfo("Ctrl-C caught. Quitting")

if __name__ == '__main__':
      try:
            escribe = Meta() #Crea objeto
            while not rospy.is_shutdown():
                  rospy.init_node('goal_publisher',
anonymous=True) #inicia nodo
                  escribe.main() #funcion main en bucle mientras
no se cierre el programa
                  rospy.sleep(1)
      except rospy.ROSInterruptException:
                rospy.loginfo("Ctrl-C caught. Quitting")
```

9.1.1.3. posinicial.py

```
#!/usr/bin/env python
import rospy
from geometry_msgs.msg import PoseWithCovarianceStamped

def pub_posini():
#VARIABLES cambian en funcion del mapa y de donde comience el
programa
    posini_x = 1.7947359085083008
    posini_y = -0.5974310040473938
    orientacionini_z = -0.3667700591974526
    orientacionini_w = 0.9303116271853734

    rospy.init_node('PosicionInicial', anonymous=False) #inicia el
nodo
    posinicial = rospy.Publisher('initialpose',
PoseWithCovarianceStamped, queue_size=10)#crea el publicador
    #le indico los valores
    pose = PoseWithCovarianceStamped()
    pose.header.frame_id = "map"
    pose.pose.pose.position.x = posini_x
    pose.pose.pose.position.y = posini_y
    pose.pose.pose.position.z = 0.0
    pose.pose.covariance=[0.25, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.25, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.06853891945200942]
    pose.pose.pose.orientation.x = 0.0
    pose.pose.pose.orientation.y = 0.0
    pose.pose.pose.orientation.z = orientacionini_z
    pose.pose.pose.orientation.w = orientacionini_w
    rospy.sleep(1) #Para que de tiempo a arrancar el publicador
    posinicial.publish(pose) #Publica la posicion inicial estimada
    print("POSICION INICIAL PUBLICADA")

pub_posini() #se llama a la funcion cada vez que se ejecute el
programa
```

9.1.1.4. sch_pos_with_battery_v4.py

```
#!/usr/bin/env python
import numpy as np
import time
import schedule
import actionlib
import rospy
import math
import os
from datetime import datetime
from std_srvs.srv import *
from move_base_msgs.msg import MoveBaseAction, MoveBaseGoal
from actionlib_msgs.msg import *
from geometry_msgs.msg import Pose, PoseWithCovarianceStamped,
Point, Quaternion, Twist
from kobuki_msgs.msg import SensorState, AutoDockingAction,
AutoDockingGoal, ButtonEvent, Sound

class SchedulePosition():
      #VARIABLES
    home_x = 0.671976923943 #coordenada x proxima a la estacion
    home_y = 0.73062390089 #coordenada y proxima a la estacion
    max_carga = 162 #Bateria al 100%
      battery_dangerous = 133 #A partir de este valor la bateria
es critica (5%)
    #Inicializar
    LowBattery = False #Parametro indicador de poca bateria
    DockStation = True #Parametro indicador de que el robot se
encuentra en la estacion
    nivel_bateria_anterior = 1000 #Bateria inalcanzable para
iniciar
      home = False #Parametro indicador de que el  robot se
encuentra cerca de la estacion para aparcar
      intentos = 0
      ficheromodificado = False

#Funcion constructor
      def __init__(self):
        self.goal_sent = False #Borra cualquier destino enviado al
iniciar
        rospy.on_shutdown(self.shutdown) #Funcion a la que se
llama al recibir CTL+C
        self.move_base = actionlib.SimpleActionClient("move_base",
MoveBaseAction) #Se crea el cliente al servidor MoveBaseAction
        rospy.loginfo("Esperando al servidor move_base action para
empezar")
        if(self.move_base.wait_for_server()): #Espera a encontrar
el servidor para continuar, por si no se ha ejecutado antes
            rospy.loginfo("Servidor encontrado")
            self.sound =
rospy.Publisher('/mobile_base/commands/sound', Sound,
queue_size=10) #Publicador encargado de sonidos

      rospy.Subscriber("/mobile_base/sensors/core",SensorState,sel
f.SensorStateCallback) #Subscriptor para sensores de bateria
```

```
      rospy.Subscriber("/mobile_base/events/button",ButtonEvent,se
lf.ButtonEventCallback) #Subscriptor para los botones del robot
            rospy.sleep(1) #Para que de tiempo a arrancar
publicadores y suscriptores

#Funcion llamada al recibir CTL+C
      def shutdown(self):
        if self.goal_sent:
            self.move_base.cancel_goal() #Si se ha enviado un
destino, lo cancela
        rospy.loginfo("Ctrl-C caught. Quitting")
            self.ReproducirSonido(1) #Reproduce sonido
        rospy.sleep(1) #Espera 1 segundo para que le de tiempo a
parar

#Funcion para leer del fichero
      def LeerFichero(self):
        Schedule = []
        tarea = []
        f =
open("/home/tb2/catkin_ws/src/scheduleposition/scripts/Schedule.tx
t") #Ruta donde va a estar el fichero
        lineas = f.readlines() #lineas es una list con cada linea
del fichero en cada posicion
            f.close()
        for line in lineas:
            frase = line.strip() #Borra espacios y \n al principio
y al final del string
            if (frase != ""): #Si no es una linea en blanco,
guarda la tarea. Para evitar errores si hay lineas en blanco en el
fichero
                  frase = frase.replace(" ","") #Por si se ponen
espacio entre medias, para evitar errores, los borra (si hay dos
espacios seguidos no los borra!!)
                  hora = frase.partition(";")[0] #Divide la frase
en tres posiciones siendo la 0 la hora, la 1 el ';' y la 2 la
posicion
                  posicion = frase.partition(";")[2]
                if (posicion == "home"): #Si lee home, se vuelve a
la estacion
                    x = self.home_x
                    y = self.home_y
                    q0 = float("1")
                    q1 = float("0")
                    q2 = float("0")
                    q3 = float("0")
                else:
                        x = float(posicion.partition(";")[0])
                        y =
float(posicion.partition(";")[2].partition(";")[0])
                    orientacion =
posicion.partition(";")[2].partition(";")[2]
                    q0 = float(orientacion.partition(";")[0])
                    q1 =
float(orientacion.partition(";")[2].partition(";")[0])
                    q2 =
float(orientacion.partition(";")[2].partition(";")[2].partition(";
")[0])
```

```
                    q3 =
float(orientacion.partition(";")[2].partition(";")[2].partition(";
")[2]) #Se va partiendo el string para dividir cada parte en una
variable
                  tarea.append(hora)
                tarea.append(x)
                tarea.append(y)
                tarea.append(q0)
                tarea.append(q1)
                tarea.append(q2)
                tarea.append(q3)
                        Schedule.append(tarea) #Matriz de tareas
                        tarea = [] #Una vez guardada la tarea, se
borra para meter una nueva
            return Schedule

      def ult_modificacion(self):
            estado =
os.stat("/home/tb2/catkin_ws/src/scheduleposition/scripts/Schedule
.txt")
            fecha = time.localtime(estado.st_mtime)
            fecha = datetime(fecha[0], fecha[1], fecha[2],
fecha[3], fecha[4], fecha[5])
            return fecha

      def goto(self, pos, quat):
            if not pos['x'] == self.home_x and not pos['y'] ==
self.home_y: #Si el destino no es la base, si esta en la base da
marcha atras
                  self.BackUp_DockStation()
            self.BorrarCostmaps() #Para borrar obstaculos que
pueden haber desaparecido
            rospy.loginfo("Go to (%s, %s) pose", pos['x'],
pos['y'])
            rospy.loginfo("Quaternion (%s, %s, %s, %s)
",quat['r1'], quat['r2'], quat['r3'], quat['r4'])
            self.goal_sent = True
        goal = MoveBaseGoal()
            goal.target_pose.header.frame_id = 'map'
        goal.target_pose.header.stamp = rospy.Time.now()
        goal.target_pose.pose = Pose(Point(pos['x'], pos['y'],
0.000), Quaternion(quat['r1'], quat['r2'], quat['r3'],
quat['r4'])) #destino al que hay que ir
        if pos['x'] == self.home_x and pos['y'] == self.home_y:
            result = self.GoToDockStation() #si el destino es la
base, se llama a la funcion encargada de llevar a la base
        else: #si el destino no es la base
                  self.home = False #sale de la base asi que se
cambia el valor de esta variable
            self.move_base.send_goal(goal) #Manda el destino
                  success =
self.move_base.wait_for_result(rospy.Duration(500)) #Da 500
segundos para llegar
            state = self.move_base.get_state() #Devuelve el estado
de la trayectoria (si esta atascado, lo ha conseguido...)
            result = False
            if success and state == GoalStatus.SUCCEEDED: #Si ha
llegado al destino y esta dentro del tiempo devuelve true
                result = True
```

```
            else:
                self.move_base.cancel_goal() #si no llega, cancela
el destino
                cmd_vel =
rospy.Publisher('cmd_vel_mux/input/navi', Twist, queue_size=10)
                cmd_vel.publish(Twist()) #publica velocidad 0 para
parar
                rospy.loginfo("Imposible llegar al punto deseado")
                self.ReproducirSonido(4) #reproduce sonido de
error
                rospy.sleep(5) #espera 5 segundos
                self.GoToDockStation() #vuelve a la estacion
            return result

#Funcion encargada de meter el horario en el planificador
      def planificar(self, horario):
            i = 0
            while i<len(horario): #mientras haya posiciones en la
lista (lineas en el fichero)
                  position = {'x': horario[i][1], 'y':
horario[i][2]}
                  quaternion = {'r1' : horario[i][4], 'r2' :
horario[i][5], 'r3' : horario[i][6], 'r4' : horario[i][3]}

      schedule.every().day.at(horario[i][0]).do(self.goto,
position, quaternion) #llamamos a la horas indicadas a la funcion
goto() que manda a las posiciones indicadas
                  i = i+1

#Funcion a la que se llama cuando se recibe un mensaje de la
bateria
    def SensorStateCallback(self, data):
        #data.battery esta variando continuamente entre dos
valores, luego solo actualiza el valor si hay una diferencia de
mas de 2
        if(math.fabs(int(data.battery) -
self.nivel_bateria_anterior) > 2):
                  rospy.loginfo("Nivel de bateria: " +
str(round((float(data.battery)-float(self.battery_dangerous)) /
(float(self.max_carga)-float(self.battery_dangerous)) * 100)) +
"%")
                  self.nivel_bateria_anterior = int(data.battery)

        if(int(data.charger) == 0): #Si es un 0, significa que se
esta descargando, luego no esta cargando.
            self.DockStation = False #lo que significa que no
estaria en la estacion
        else:
            self.DockStation = True

        #Si la bateria disminuye por debajo del 10% de carga,
significa que la bateria es baja
        if (round((float(data.battery)-
float(self.battery_dangerous)) / (float(self.max_carga)-
float(self.battery_dangerous)) * 100) < 10) :
            if(not self.LowBattery):
                rospy.loginfo("BATERIA BAJA")
                self.LowBattery = True #Bateria baja
```

```
        #Hasta que no este por encima del 20% no dejamos que salga
de la estacion
            #Asi evitamos que se ponga a cargar y al instante
salga
        if ((round(float(data.battery)-
float(self.battery_dangerous)) / (float(self.max_carga)-
float(self.battery_dangerous)) * 100) > 20) :
            if(self.LowBattery):
                rospy.loginfo("Puede salir de la estacion")
                self.LowBattery = False #Ya no hay bateria baja

#Funcion encargada de dar marcha atras si estamos en la base y
recibimos un destino
    def BackUp_DockStation(self):
        if(self.DockStation == True):
            rospy.loginfo("Robot en estacion. Dando marcha atras
antes de empezar")
            self.DockStation = False
            self.home = False
            cmd_vel = rospy.Publisher('cmd_vel_mux/input/navi',
Twist, queue_size=10)
            #Twist es el tipo de mensaje
            move_cmd = Twist()
            #Marcha atras a  0.1 m/s
            move_cmd.linear.x = -0.1
            move_cmd.angular.z = 0
            r = rospy.Rate(10); #10Hz
            temp_count = 0
            #Marcha atras durante 30 segundos
            while (not rospy.is_shutdown() and temp_count < 300):
                    cmd_vel.publish(move_cmd)
                    temp_count = temp_count + 1
                    r.sleep() #Espera hasta 0.1 s (10 HZ) y
publica otra vez
            #Nos aseguramos de que para publicando un Twist()
            cmd_vel.publish(Twist())
        return True

#Funcion para ir a la base de carga
    def GoToDockStation(self):
            #Primero nos aproximamos a la base
        if self.DockStation: #Si ya esta en la base no hace nada
            rospy.loginfo("Ya estamos en la base")
            result = True
        else:
            while (self.home == False): #mientras no estemos cerca
de la base, intentamos llegar a ese punto cercano
                if(self.intentos<3): #se intenta 3 veces
                    rospy.loginfo("Going HOME")
                    self.BorrarCostmaps() #se borran mapas de
costo
                    goal = MoveBaseGoal()
                    goal.target_pose.header.frame_id = 'map'
                    goal.target_pose.header.stamp =
rospy.Time.now()
                    goal.target_pose.pose =
Pose(Point(self.home_x, self.home_y, 0.000),
Quaternion(0.000,0.000, 0.000, 1.000))
```

```
                    self.move_base.send_goal(goal) #se le manda el
punto cercano a la base como destino

                    success =
self.move_base.wait_for_result(rospy.Duration(500)) #se le deja
500 segundos para llegar
                    state = self.move_base.get_state()
                    result = False
                    if success and state == GoalStatus.SUCCEEDED:
#si llega al punto
                            result = True
                        self.home = True #indica que estamos cerca
de la base
                        self.intentos = 0 #reinicia intentos
                    else: #si no llega al punto
                        rospy.loginfo("ERROR AL LLEGAR A LA BASE,
VOLVIENDO A INTENTARLO")
                        self.move_base.cancel_goal() #cancela el
destino
                        cmd_vel =
rospy.Publisher('cmd_vel_mux/input/navi', Twist, queue_size=10)
                        cmd_vel.publish(Twist()) #paramos el robot
                        self.intentos +=1 #incrementamos el valor
del intento
                        self.ReproducirSonido(4) #reproduce sonido
de error
                        rospy.loginfo("Numero intentos: %d",
self.intentos) #muestra el numero de intentos que lleva
                else: #si no lo consigue en tres intentos
                        self.move_base.cancel_goal() #cancela el
destino
                    cmd_vel.publish(Twist()) #para el robot
                    result = False
                    rospy.signal_shutdown("ERROR. Imposible llegar
a HOME") #sale del programa

                  #Si ha llegado cerca de la base, aparcamos
            self._client =
actionlib.SimpleActionClient('/dock_drive_action',
AutoDockingAction) #cliente del servidor AutoDockingAction
            rospy.loginfo("Esperando al servidor auto_docking")
            self._client.wait_for_server() #espera hasta encontrar
el servidor por si no se ha ejecutado
            rospy.loginfo("Servidor auto_docking encontrado")
            goal = AutoDockingGoal()
            rospy.loginfo("APARCANDO... (cancelando si pasa de
180s)")
            self._client.send_goal(goal) #envia el destino que es
la base de carga

            #Le damos 180seg para aparcar
            success =
self._client.wait_for_result(rospy.Duration(180))
            if success:
                  rospy.loginfo("Aparcamiento conseguido")
                  self.DockStation = True #Estamos en la base
                  return True
            else:
```

```
                        self._client.cancel_goal() #cancela el
aparcamiento
                rospy.loginfo("Fallo en el aparcamiento")
                return False

#Funcion que va a comprobar si se necesita ir a la base de carga
    def NeedCharge(self):
            #Si la bateria es baja pero estamos en la base, nos
quedamos  hasta que este suficientemente cargado
        if(self.DockStation and self.LowBattery):
            rospy.loginfo("Robot en la base de carga")
            rospy.loginfo("Esperando a que este suficientemente
cargado")
            time.sleep(30)
            return True

        #Si la bateria es baja y no estamos en la base, va a ella
        if(not self.DockStation and self.LowBattery):
            rospy.loginfo("Bateria baja. Yendo a la base de
carga")
            self.GoToDockStation()
            return True
        return False

#Funcion llamada tras mensaje de los botones
    def ButtonEventCallback(self, data):
            if (data.state == ButtonEvent.PRESSED): #Si un boton
ha sido presionado
                  if (data.button == ButtonEvent.Button0): #Y ha
sido el boton 0
                        if (schedule.idle_seconds() <= 180):
#Segundos hasta la siguiente tarea
                              #Esto lo hago porque las tareas del
schedule entran al programa parando cualquier cosa
                              #que este haciendo.  Si esta yendo a
la base y salta una tarea, parara y realizara la tarea
                              rospy.loginfo("QUEDAN MENOS DE 3
MINUTOS PARA LA SIGUIENTE TAREA")
                              rospy.loginfo("HASTA QUE NO LA
REALICE, NO PUEDE IR A HOME POR SEGURIDAD")
                        else: #si no hay tareas cerca
                              if(not self.DockStation):
                                    rospy.loginfo("Yendo a la
base...")
                                    self.GoToDockStation() #si no
esta en la base, vuelve a ella
                              else:
                                    rospy.loginfo("Ya estamos en
la base")

#Funcion encargada de borrar los mapas de costo
    def BorrarCostmaps(self):
            rospy.wait_for_service('/move_base/clear_costmaps')
            clear_costmaps =
rospy.ServiceProxy('/move_base/clear_costmaps', Empty)
            if(clear_costmaps.call()): #Borra los mapas
                  rospy.loginfo('GLOBAL COSTMAPS BORRADOS')
            else:
```

```
                  rospy.loginfo('Error al llamar al servicio
/move_base/clear_costmaps')

#Funcion encargada de reproducir sonidos
    def ReproducirSonido(self, num):
            self.sound.publish(num) #publica el sonido que se le
pase como argumento
            rospy.sleep(5)

if __name__ == '__main__':
        try:
            rospy.init_node('SchedulePosition', anonymous=False)
#inicia el nodo
            navigator = SchedulePosition() #crea objeto
            horario = navigator.LeerFichero() #guarda en horario
lo que hay dentro del fichero en una matriz de matrices
            primeramod =  navigator.ult_modificacion() #guarda el
valor de la ultima modificacion del fichero
            navigator.planificar(horario) #planifica
            while not rospy.is_shutdown(): #mientras no se cierre,
entra en bucle
                ultmod = navigator.ult_modificacion()  #comprueba
la fecha actual de la ultima modificacion del fichero
                if (ultmod != primeramod): #si es diferente con la
guardada anteriormente
                    navigator.ficheromodificado = True #el fichero
ha sido modficiado
                    primeramod = ultmod #guardamos la ultima fecha
de modificacion
                    print("Fichero modificado")
                else:
                    navigator.ficheromodificado = False #sino, no
ha sido modificado
                if (not navigator.NeedCharge() and not
navigator.ficheromodificado): #si no necesita carga y el fichero
no ha sido modificado.
                    schedule.run_pending() #comienza a realizar
tareas en funcion de la planificacion
                else:  #si el fichero ha sido modificado o
necesita bateria
                    print("Actualizando horario")
                    schedule.clear() #borra el plan anterior
                    horario = navigator.LeerFichero() #lee el
fichero
                    navigator.planificar(horario) #vuelve a
planificarlo
                time.sleep(1)


        except rospy.ROSInterruptException:
                rospy.loginfo("Ctrl-C caught. Quitting")
```

9.1.2. Archivos launch

9.1.2.1. minimalv2.launch del paquete scheduleposition

```
<launch>
  <!-- KOBUKI_AUTO_DOCKING minimal.launch -->

  <node pkg="nodelet" type="nodelet" name="dock_drive" args="load
kobuki_auto_docking/AutoDockingNodelet
mobile_base_nodelet_manager">
    <rosparam file="$(find
kobuki_auto_docking)/param/auto_docking.yaml" command="load"/>
    <remap from="dock_drive/odom" to="odom"/>
    <remap from="dock_drive/core" to="mobile_base/sensors/core"/>
    <remap from="dock_drive/dock_ir"
to="mobile_base/sensors/dock_ir"/>
    <remap from="dock_drive/motor_power"
to="mobile_base/commands/motor_power"/>
    <remap from="dock_drive/velocity"
to="mobile_base/commands/velocity"/>
  </node>

  <!-- SCHEDULEPOSITION -->

  <node pkg="scheduleposition" name="sch_pos_with_battery"
type="sch_pos_with_battery_v4.py" output="screen">
  </node>
</launch>
```

9.1.2.2. minimal_with_hokuyo.launch

```
<launch>
  <!-- Turtlebot -->
  <arg name="base"              default="$(env TURTLEBOT_BASE)"
doc="mobile base type [create, roomba]"/>
  <arg name="battery"           default="$(env TURTLEBOT_BATTERY)"
doc="kernel provided locatio for battery info, use
/proc/acpi/battery/BAT0 in 2.6 or earlier kernels." />
  <arg name="stacks"            default="$(env TURTLEBOT_STACKS)"
doc="stack type displayed in visualisation/simulation [circles,
hexagons]"/>
  <arg name="3d_sensor"         default="$(env
TURTLEBOT_3D_SENSOR)"    doc="3d sensor types [kinect,
asux_xtion_pro]"/>
  <arg name="simulation"        default="$(env
TURTLEBOT_SIMULATION)"   doc="set flags to indicate this turtle is
run in simulation mode."/>
  <arg name="serialport"        default="$(env
TURTLEBOT_SERIAL_PORT)"  doc="used by create to configure the port
it is connected on [/dev/ttyUSB0, /dev/ttyS0]"/>

  <param name="/use_sim_time" value="$(arg simulation)"/>

  <include file="$(find
turtlebot_bringup)/launch/includes/robot.launch.xml">
    <arg name="base" value="$(arg base)" />
```

```
    <arg name="stacks" value="$(arg stacks)" />
    <arg name="3d_sensor" value="$(arg 3d_sensor)" />
  </include>
  <include file="$(find
turtlebot_bringup)/launch/includes/mobile_base.launch.xml">
    <arg name="base" value="$(arg base)" />
    <arg name="serialport" value="$(arg serialport)" />
  </include>
  <include unless="$(eval arg('battery') == 'None')" file="$(find
turtlebot_bringup)/launch/includes/netbook.launch.xml">
    <arg name="battery" value="$(arg battery)" />
  </include>

  <node name="hokuyo" pkg="urg_node" type="urg_node"
respawn="false" output="screen">
    <param name="calibrate_time" type="bool" value="true"/>
    <param name="port" type="string" value="/dev/ttyACM0"/>
    <param name="intensity" type="bool" value="false"/>
    <param name="min_ang" value="-2.35619449615"/>
    <param name="max_ang" value="+2.09234976768"/>
    <param name="cluster" value="1"/>
    <param name="frame_id" value="hokuyo_laser_frame"/>
  </node>

</launch>
```

9.1.2.3. gmapping_demo_hokuyo.launch

```
<launch>
  <!-- 3D sensor -->
  <arg name="3d_sensor" default="$(env TURTLEBOT_3D_SENSOR)"/>
<!-- r200, kinect, asus_xtion_pro -->
  <include file="$(find
turtlebot_bringup)/launch/3dsensor.launch">
    <arg name="rgb_processing" value="false" />
    <arg name="depth_registration" value="false" />
    <arg name="depth_processing" value="false" />
    <arg name="scan_processing" value="false" />
    <!-- We must specify an absolute topic name because if not it
will be prefixed by "$(arg camera)".
         Probably is a bug in the nodelet manager:
https://github.com/ros/nodelet_core/issues/7 -->
    <arg name="scan_topic" value="/scan" />
  </include>

  <!-- Gmapping -->
  <arg name="custom_gmapping_launch_file" default="$(find
turtlebot_navigation)/launch/includes/gmapping/$(arg
3d_sensor)_hokuyo_gmapping.launch.xml"/>
  <include file="$(arg custom_gmapping_launch_file)"/>

  <!-- Move base -->
  <include file="$(find
turtlebot_navigation)/launch/includes/move_base.launch.xml"/>

</launch>
```

9.1.2.4. amcl_demo_hokuyo.launch

```
<launch>
  <!-- 3D sensor -->
  <arg name="3d_sensor" default="$(env TURTLEBOT_3D_SENSOR)"/>
<!-- r200, kinect, asus_xtion_pro -->
  <include file="$(find
turtlebot_bringup)/launch/3dsensor.launch">
    <arg name="rgb_processing" value="false" />
    <arg name="depth_registration" value="false" />
    <arg name="depth_processing" value="false" />
    <arg name="scan_processing" value="false" />

    <!-- We must specify an absolute topic name because if not it
will be prefixed by "$(arg camera)".
         Probably is a bug in the nodelet manager:
https://github.com/ros/nodelet_core/issues/7 -->
    <arg name="scan_topic" value="/scan" />
  </include>

  <!-- Map server -->
  <arg name="map_file" default="$(env TURTLEBOT_MAP_FILE)"/>
  <node name="map_server" pkg="map_server" type="map_server"
args="$(arg map_file)" />

  <!-- AMCL -->
  <arg name="custom_amcl_launch_file" default="$(find
turtlebot_navigation)/launch/includes/amcl/$(arg
3d_sensor)_amcl.launch.xml"/>
  <arg name="initial_pose_x" default="0.0"/> <!-- Use 17.0 for
willow's map in simulation -->
  <arg name="initial_pose_y" default="0.0"/> <!-- Use 17.0 for
willow's map in simulation -->
  <arg name="initial_pose_a" default="0.0"/>
  <include file="$(arg custom_amcl_launch_file)">
    <arg name="initial_pose_x" value="$(arg initial_pose_x)"/>
    <arg name="initial_pose_y" value="$(arg initial_pose_y)"/>
    <arg name="initial_pose_a" value="$(arg initial_pose_a)"/>
  </include>

  <!-- Move base -->
  <arg name="custom_param_file" default="$(find
turtlebot_navigation)/param/$(arg
3d_sensor)_costmap_params.yaml"/>
  <include file="$(find
turtlebot_navigation)/launch/includes/move_base.launch.xml">
    <arg name="custom_param_file" value="$(arg
custom_param_file)"/>
  </include>

</launch>
```