



UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA Y
DISEÑO INDUSTRIAL

Grado en Ingeniería Electrónica Industrial y Automática

TRABAJO FIN DE GRADO

GUIADO DE UN ROBOT MÓVIL BASADO EN ROS Y KINECT

Autor: Daniel Manzaneque Amo

Cotutor: Dr. Miguel Hernando

Gutiérrez

Departamento: Electricidad,
Electrónica, Automática y Física
aplicada.

Tutor: Dr. Alberto Brunete

González

Departamento: Electricidad,
Electrónica, Automática y Física
aplicada.

Madrid, Febrero 2016



UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA Y
DISEÑO INDUSTRIAL

Grado en Ingeniería Electrónica y Automática Industrial

TRABAJO FIN DE GRADO

GUIADO DE UN ROBOT MÓVIL
BASADO EN ROS Y KINECT

Firma Autor

Firma Cotutor

Firma Tutor

Copyright ©2016. Daniel Manzaneque Amo

Esta obra está licenciada bajo la licencia Creative Commons

Atribución-NoComercial-SinDerivadas 3.0 Unported (CC BY-NC-ND 3.0). Para ver una copia de esta licencia, visite <http://creativecommons.org/licenses/by-nc-nd/3.0/deed.es> o envíe una carta a Creative Commons, 444 Castro Street, Suite 900, Mountain View, California, 94041, EE.UU. Todas las opiniones aquí expresadas son del autor, y no reflejan necesariamente las opiniones de la Universidad Politécnica de Madrid.

Título: Guiado de un robot móvil basado en ROS y kinect

Autor: Daniel Manzaneque Amo

Tutor: Dr. Alberto Brunete González

Cotutor: Dr. Miguel Hernando Gutiérrez

EL TRIBUNAL

Presidente:

Vocal:

Secretario:

Realizado el acto de defensa y lectura del Trabajo Fin de Grado el día de de ... en, en la Escuela Técnica Superior de Ingeniería y Diseño Industrial de la Universidad Politécnica de Madrid, acuerda otorgarle la CALIFICACIÓN de:

VOCAL

SECRETARIO

PRESIDENTE

*“Lo sé porque muchos ya se fueron
y hoy sigo sus pasos al caminar.
Y aquí tú y yo, solo quedamos los buenos,
nadie nos enseña donde parar.”*

— Vetusta Morla, *Los Buenos*

Agradecimientos

Cuando una etapa en la vida termina normalmente uno se para y reflexiona sobre las metas y objetivos alcanzados y muchas veces descuida todo aquel esfuerzo y los momentos difíciles que se sortearon hasta poder cumplirlos. Quiero que estos agradecimientos sirvan para reconocer el apoyo de todas aquellas personas que han contribuido en mi vida llegado este punto y que me han ayudado de una u otra manera a superar todos los retos.

En primer lugar quiero agradecer el apoyo incondicional de mi familia durante todos estos años, en especial a mis padres por haberme dado la oportunidad de estudiar algo que me entusiasma para que en un futuro pueda dedicarme a ello. Gracias a ellos y a mi hermana por el apoyo en cada uno de los momentos complicados y por disfrutar conmigo de los logros y buenos momentos.

Quiero agradecer el apoyo de todos mis amigos pero especialmente a Samu, Manu y Sas por sus ánimos, los buenos ratos que hemos pasado y porque sé que apuestan por mí más de lo que hago yo a veces.

No quiero dejar pasar la oportunidad de agradecer a Andrea todo el esfuerzo y la paciencia que ha tenido conmigo desde los primeros años de universidad, su inestimable apoyo en los momentos de bajón y todas las experiencias que he vivido junto a ella y que me han servido para creer como persona, afrontar nuevos retos y creer más en mí cada día. Creo que podemos decir que los dos hemos hecho un gran doble grado después de todo.

Durante mis años de universidad he tenido la oportunidad de compartir espacio y tiempo con grandes compañeros. Daniela, Isma, Luis y Gema, habéis sido grandes compañeros y sois unos grandes amigos. Gracias por los buenos momentos en clase y sobre todo fuera de ella, ojalá nuestras inquietudes nos lleven a trabajar juntos en futuros proyectos.

También deseo dar las gracias a los que han sido mis compañeros de laboratorio durante el proyecto, Mario, Irene, Sofía, Koldo y Chechu, por todas las tardes que hemos pasado juntos y por lo mucho que nos hemos reído. Sé que me acordaré de esos momentos más de una vez.

Agradecer a mis profesores de instituto, en especial a mis profesores de tecnología, por hacer que su labor de enseñanza me llevase a estudiar este grado. También a mis tutores de proyecto Miguel y Alberto por haberme dado la oportunidad de desarrol-

lar este proyecto, por haber contribuido a mi enseñanza como ingeniero y por todos sus buenos consejos.

Por último, agradecer a todos los que me han aguantado con mi robotito andando y haciendo experimentos de aquí para allá. ¡Me lo he pasado en grande!

Resumen

Muchos robots autónomos surgen como herramienta para acceder a lugares donde el ser humano no puede o no conviene que acceda porque se encontraría en riesgo.

Un robot autónomo es por tanto una pieza fundamental en tareas de rescate, salvamento, inspección, exploración de entornos peligrosos o inaccesibles, como la exploración en la superficie de otros planetas. Además, tareas sociales como la asistencia a humanos en entornos públicos, la interacción con el entorno o una navegación más segura, como es el caso de los coches autónomos, cada vez están tomando más relevancia en nuestro día a día.

Este proyecto de fin de grado trata sobre el guiado y control de un robot móvil de cuatro ruedas, con un sistema motriz en configuración skid-steer, equipado con una serie de sensores que permiten su orientación y posicionado en el entorno así como un sensor capaz de captar este en tres dimensiones y un sensor adicional que lo hace tan solo en dos dimensiones.

Los datos de los sensores sirven tanto para construir mapas en dos dimensiones del entorno del robot como para navegar por él evitando obstáculos de manera dinámica. El robot es capaz de generar mapas de celdas en los que situar tanto los objetos estáticos como los móviles, calcular una trayectoria adecuada y dirigirse hasta un punto indicado evitando obstáculos interpuestos en su camino.

Todo esta información, procesado de datos, cálculo de trayectorias y ejecución de movimientos se realiza en un ordenador de abordo integrado en el propio robot utilizando el software Robot Operating System (conocido en robótica por sus siglas ROS), que nos ofrece una interfaz común para interconectar nuestro robot con los sensores y con los algoritmos de navegación.

A parte de la navegación autónoma, también se ha incluido un sistema de telecontrol del robot mediante otro ordenador externo y de un algoritmo de detección frontal de objetos en 3 dimensiones (nubes de puntos) que puedan servirle como guía. De esta forma, el robot es capaz de navegar siguiendo el movimiento de una persona o de un robot que le preceda.

El robot Pioneer 3 AT es el robot móvil que se ha empleado en este proyecto (Figura 1) y sobre el que se ha trabajado de manera específica para realizar las pruebas reales de este proyecto. A este robot se le incorporan un sensor láser de dos dimensiones (sensor Sick LMS100) y un sensor de tres dimensiones (sensor Kinect). El cómputo de la navegación se realiza en un ordenador compacto incorporado en el robot (Intel NUC NUC5i7RYH).

Las consignas de navegación se realizan mediante un ordenador externo cualquiera conectado a una red inalámbrica o mediante consignas de voz, en las que se indica al robot las tareas de navegación a realizar (avanzar, girar, seguir a una persona...) o un punto del entorno al que dirigirse.

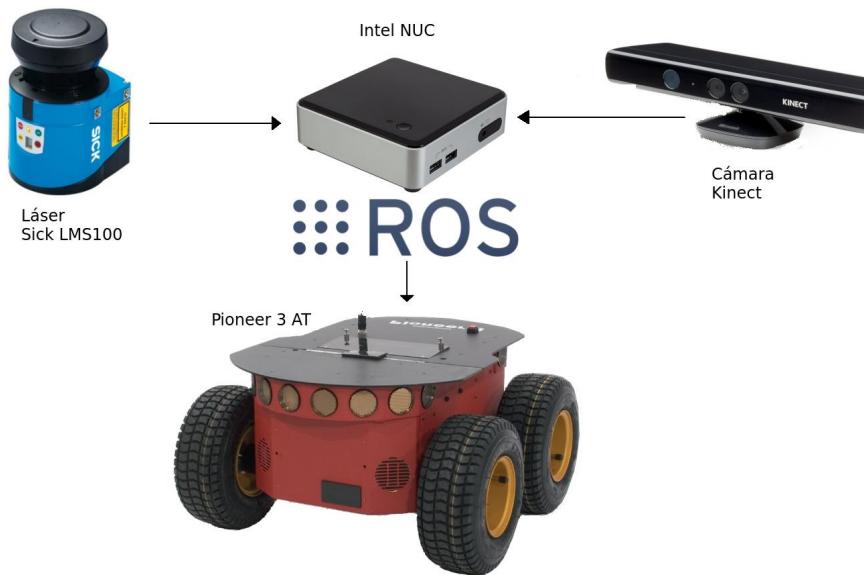


Figura 1: Esquema del sistema robótico utilizado en el proyecto

Todas estas implementaciones están desarrolladas bajo el entorno ROS, lo cual permite añadir funcionalidades de manera más rápida y menos laboriosa, como es el caso del control mediante comandos de voz o la interacción mediante sonidos. Es el caso también del simulador de robótica Gazebo, que se integra como funcionalidad en ROS y que ha servido para testar el sistema y aportar las pruebas teóricas pertinentes para luego aplicarlas en el robot real.

Para concluir, podemos decir que este proyecto se encarga de integrar ROS como sistema en un ordenador de abordo incorporado en el robot que permita conectarse con los sensores y realizar la construcción de mapas y navegación autónoma mediante el cálculo de mapas y trayectorias globales y locales, realizar los movimientos del robot, así como reconocer consignas de voz o de teleoperación.

Palabras clave: robot móvil, ROS, navegación reactiva, cálculo de trayectorias.

Abstract

Achieving navigation and guidance of mobile robot comes up as a tool for rescue purposes in places where humans can't access or that involve a high risk for life. Many of those repetitive and fatigating tasks could be done with a robust and capable mobile robot.

An autonomous robot is, an essential part in rescue, inspection and exploration tasks developed in dangerous or non-reachable places, such as the surface of other planets. Moreover, social tasks such as assistance for humans in public places, interaction with the environment or a safer navigation in the cities. Autonomous cars are a good example of this.

This final degree project is about guidance and control of a four-wheel mobile robot with a skid-steer configuration. It is equipped with a sort of sensors, allowing it to make positioning and orientation in the environment. There is also a main sensor capturing the environment in three dimensions and an additional one doing it in two dimensions.

Sensor data is used to build two dimensional maps of the exploration place as well as to take care of dynamical obstacles. The robot can build maps formed by cells where to incorporate or raytrace static and dynamic obstacles, calculate the proper trajectory plan and head for a destination point avoiding obstacles in its way.

All this information, data processing, trajectory calculation and movement execution is done in an onboard computer inside the robot. It uses the Robot Operating System software (known as ROS), which offers a common interface to communicate the robot with sensors and navigation algorithms.

Apart from autonomous navigation, the robot also has a telecontrol system from an outside computer and an algorithm to detect frontal objects in three dimensions (pointclouds) that can guide the robot. This is how it can navigate following a person when it is walking or another robot in front of it.

Pioneer 3 AT robot is the one used in this project (Figure 2). It has been the specific platform for all real tests. This robot is equipped with a two dimension laser scanner (Sick LMS100 sensor) and a three dimensional sensor (Kinect sensor). The navigation computation is done in an onboard compact computer (Intel NUC NUC5i7RYH computer).

The navigation commands are sent from an outside computer connected to the same wireless network or from voice navigation commands speaking directly to the robot (go forward, backward, turn right...) or pointing a goal in the map.

All those implementations are developed under ROS framework. This is why additional features can be added in a faster and effortless way. That is the case of the robot simulator Gazebo, which integrates as an add-on in ROS. Gazebo has been used to perform tests in navigation and to check theoretical concepts to lately

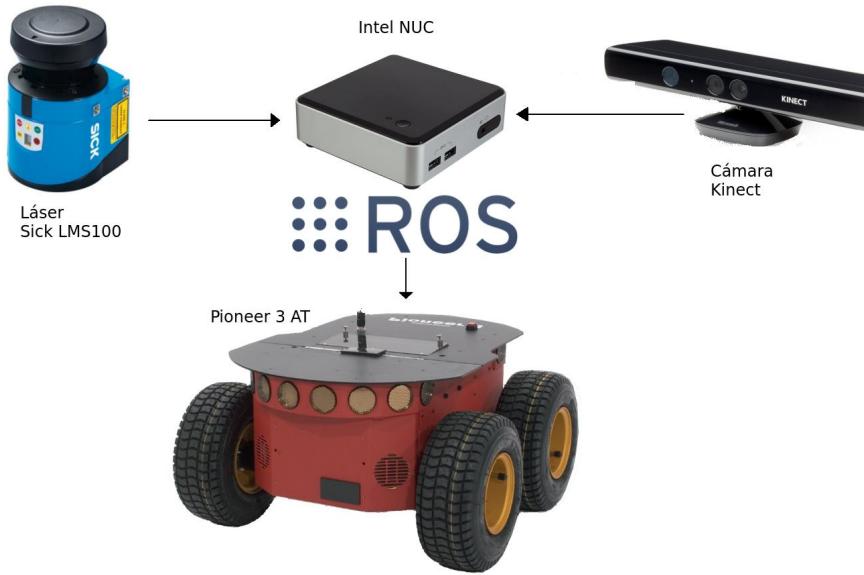


Figure 2: Diagram of the robotic system used for this project

incorporate them in the real robotic system.

To conclude, it can be said that this project integrates ROS as a robotic system in an onboard computer and connects to sensors to perform tasks such as building maps or navigation from one point to another. The system calculates local and global maps and trajectories, makes movements according to them, as well as recognises voice or teleoperation commands.

Keywords: mobile robot, ROS, reactive navigation, trajectory calculation.

Índice general

Agradecimientos	ix
Resumen	xi
Abstract	xiii
1 Introducción	1
1.1 Motivación del proyecto	1
1.2 Objetivos del proyecto	2
1.3 Estructura del documento	3
2 Estado del arte	5
2.1 Robótica Móvil	5
2.2 Hardware en robótica móvil	6
2.3 Sensores en robótica móvil	9
2.3.1 Sensores propioceptivos	9
2.3.2 Sensores exteroceptivos	10
2.4 Aplicaciones actuales de la robótica móvil	12
3 Planteamiento del proyecto	19
3.1 Planteamiento preliminar	19
3.2 Planificación del proyecto	20
3.3 Tecnologías y herramientas empleadas en el proyecto	24
3.3.1 Robot Operating System	24
3.3.2 Lenguaje de programación C++	25
3.3.3 Lenguaje de programación Python	25
3.3.4 Controlador de versiones git y repositorios GitHub	25
3.3.5 Simulador de robótica Gazebo	26
3.3.6 RViz: Herramienta de visualización robótica	26
3.3.7 Impresión 3D	27
3.4 Hardware	28
3.4.1 Pioneer 3 AT	28
3.4.2 Sensor Kinect	29
3.4.3 Láser SICK LMS100	31
3.4.4 Intel NUC NUC5i7RYH	32

4 Conceptos previos	35
4.1 Entorno ROS	35
4.1.1 Funcionamiento de ROS	35
4.1.2 Configuración de ROS	38
4.1.3 Configuración de los paquetes ROS	39
4.2 Arquitectura	40
4.2.1 Arquitectura del sistema	40
4.2.2 Estructura software del proyecto	40
5 Navegación	43
5.1 Navigation Stack	43
5.1.1 Funcionamiento	44
5.1.2 Requisitos para la navegación	45
5.1.3 Configuración de la navegación	46
5.2 SLAM	49
5.2.1 slam_gmapping en ROS	49
5.3 AMCL	50
5.4 Mapas de coste: Costmaps	52
5.5 Planificador de trayectoria global	55
5.5.1 Algoritmo de Dijkstra	55
5.5.2 Algoritmo de A estrella	58
5.6 Planificador de trayectoria local	59
6 Control a bajo nivel	61
6.1 Nodos Hardware	61
6.1.1 Control del robot: Rosaria y p2os	61
6.1.2 Sensor Kinect	62
6.1.3 Sensor Láser Sick LMS100	63
6.1.4 Integración del hardware	64
6.2 Nodo de teleoperación	66
6.2.1 Características distribuidas de ROS	66
6.2.2 Implementación del nodo	66
6.3 Nodo de navegación estimada	67
7 Implementación del sistema	69
7.1 Configuraciones hardware	69
7.1.1 Pioneer 3 AT	69
7.1.2 Sensor Láser	70
7.1.3 Sensor Kinect	70
7.1.4 Primera configuración hardware	71
7.1.5 Segunda configuración hardware	74
7.2 Navegación	78
7.2.1 Configuración de los costmaps y los sensores	78
7.2.2 Configuración de los planificadores de trayectoria	83
7.2.3 Navegación con mapa	85
7.2.4 Navegación reactiva	86
7.3 Nodo de navegación por puntos	87
7.3.1 Endurance test	89

7.4	Nodo de guiado (follower)	89
7.5	Feedback mediante text-to-speech	91
7.6	Reconocimiento de comandos de voz	91
7.6.1	Crear una lista de vocabulario	93
7.7	Nodo de ejecución automática de nodos	94
7.7.1	Ejecución de nodos	94
7.7.2	Funcionamiento	95
8	Simulación del sistema	97
8.1	Simulación con MobileSim	97
8.2	Simulación con Gazebo	98
8.2.1	Modelado del robot en el simulador	98
8.2.2	Funcionamiento de Gazebo	100
9	Pruebas del sistema	103
9.1	Pruebas simuladas	103
9.1.1	SLAM	103
9.1.2	Navegación con mapa	103
9.1.3	Navegación reactiva	105
9.1.4	Pruebas de resistencia	106
9.2	Pruebas reales	107
9.2.1	SLAM	108
9.2.2	Pruebas de resistencia	109
9.2.3	Aspectos de la navegación	112
10	Conclusiones	113
10.1	Conclusión sobre la metodología	113
10.2	Conclusión sobre los resultados	113
10.3	Desarrollos futuros	113
Apéndice		114
A	Configuración del sistema	117
A.1	Configuración del espacio de trabajo	117
A.1.1	Instalación de las librerías	118
A.1.2	Gestión de las dependencias	118
A.2	Configuración del hardware	120
A.2.1	Calibración de la odometría	120
A.2.2	Ordenador de abordo Intel NUC	121
A.2.3	Sensor Kinect	122
A.2.4	Láser SICK LMS100	122
B	Manual de uso del robot	125
B.1	Encendido del robot	125
B.2	Panel de control y parada de emergencia	125
B.3	Conexión mediante un ordenador externo vía Wifi	127
B.4	Acceder a la placa de alimentación	128
B.5	Cargar las baterías del robot	128
B.6	Ordenador interno del Pioneer 3 AT	129

C Información y documentos ONLINE	131
C.1 Repositorio software	131
C.1.1 Readme	132
C.2 Preguntas en ROS Answers y Github	136
C.3 Multimedia	136
C.4 Memoria del trabajo	137
Bibliografía	139

Índice de figuras

1	Esquema del sistema robótico utilizado en el proyecto	xii
2	Diagram of the robotic system used for this project	xiv
2.1	Motor de corriente continua con encoder.	6
2.2	Configuraciones hardware. Basado en [SOGSBS ⁺ 10].	7
2.3	Robot de configuración diferencial Pioneer 3 DX.	7
2.4	Ejemplos de configuración skid-steer: Cargador frontal, Robotnik Guardian, Pioneer 3 AT.	8
2.5	Configuración síncrona.	8
2.6	Configuración síncrona y rueda omnidireccional.	9
2.7	Robot Uranus con ruedas tipo Mecanum.	9
2.8	Esquema de un encoder absoluto	10
2.9	Unidad de medida inercial, IMU	10
2.10	Cámara estereoscópica del robot PR2.	11
2.11	Sensor de 3 dimensiones Kinect para Xbox 360.	12
2.12	Imagen tomada a sí mismo por el robot Curiosity en la superficie marciana.	13
2.13	Robots de desactivación de explosivos: iRobot 510 Packbot y TALON.	13
2.14	Robot Quince (izq.) y robot Raccoon (dcha.).	14
2.15	Robot para limpieza del hogar Roomba, de iRobot.	14
2.16	Robot pulverizador de aplicación agrícola, AgriRobot.	14
2.17	Robot de inspección de viñedos, VinBot.	15
2.18	Robot social Maggie y robot de asistencia ROSA.	15
2.19	Robot PR2 desarrollado por Willow Garage.	16
2.20	Sensores del coche autónomo de Google.	16
3.1	Diagrama de Gantt de la evolución del proyecto.	23
3.2	Logo de ROS.	24
3.3	Entorno de simulación Gazebo.	26
3.4	Entorno gráfico RViz.	27
3.5	Robot Pioneer 3-AT.	28
3.6	Panel de control del robot Pioneer 3-AT.	28
3.7	Sensor Kinect.	29
3.8	Proyección de infrarrojos y obtención de la nube de puntos.	30
3.9	Sensor escáner láser Sick LMS100.	31
3.10	ordenador compacto Intel NUC.	32
4.1	Grafo de ejemplo con nodos conectados	36

4.2	Esquema del funcionamiento de ROS	36
4.3	Frames utilizados en el robot PR2.	38
4.4	Estructura del proyecto	41
4.5	Estructura de carpetas del paquete pioneer_utils	42
5.1	Ejemplo de mapas de coste en navegación.	44
5.2	Diagrama de funcionamiento del Navigation Stack [ME10b]	44
5.3	Visualización de costmaps, sensor láser y modelo del robot en RViz. .	48
5.4	slam_gmapping visualizado en RViz	50
5.5	Esquema de la labor del nodo AMCL entre los <i>frames</i> map y base. .	51
5.6	Esquema sobre el cálculo del coste de cada celda en el mapa.	54
5.7	Clasificación de los diferentes métodos de planificación de trayectorias. Basado en [PB15].	55
5.8	Pseudo-código del algoritmo de Dijkstra. Basado en [Tom14].	56
5.9	Implementación del algoritmo de Dijkstra en un mapa sin obstáculos. Basado en [Pat10].	56
5.10	Implementación del algoritmo Best-First-Search en un mapa sin obstáculos. Basado en [Pat10].	57
5.11	Algoritmo de Dijkstra (Izq.) y algoritmo Best-First-Search (Dcha.) en un mapa con obstáculo cóncavo. Basado en [Pat10].	57
5.12	Pseudo-código del algoritmo A*. Basado en [Tom14].	58
5.13	Algoritmo A* sin obstáculo (Izq.) y con obstáculo cóncavo (Dcha.). Basado en [Pat10].	59
5.14	Esquema del funcionamiento de los planificadores de trayectoria local.	59
6.1	Referencias <i>frames</i> de la configuración del robot.	65
7.1	Estado del robot al comienzo del proyecto [HdC13].	70
7.2	Conexión usual del sensor Kinect a la consola Xbox.	71
7.3	Adaptación de cables para la alimentación del sensor Kinect (izq.) y cable a 12V de la placa de alimentación del robot (dcha.).	71
7.4	Primera configuración hardware del robot: Kinect y láser en la parte frontal.	72
7.5	Esquema de la segunda configuración del sistema. Basado en [HdC13].	73
7.6	Robot con portátil incorporado realizando navegación y soporte realizado con la impresora 3D.	74
7.7	Esquema de la nueva configuración hardware.	75
7.8	Posición retrasada del láser y nuevas varillas de soporte.	75
7.9	Pieza 3D para el soporte del sensor Kinect y su posición final.	76
7.10	Panelado del robot.	76
7.11	Imágenes del diseño final del robot "Petrois".	77
7.12	Esquema del sistema robótico final utilizado en el proyecto.	78
7.13	Visualizado del costmap global.	82
7.14	Planificador de trayectoria A* visualizado en RViz.	83
7.15	Planificador de trayectoria Dijkstra visualizado en RViz.	84
7.16	Planificador de trayectoria local <i>Trajectory Rollout</i> en RViz.	85
7.17	Creación de un mapa mediante SLAM.	86
7.18	Herramienta web Sphinx Knowledge Base Tool.	94

8.1	MobileSim junto con RViz funcionando con teleoperación.	97
8.2	Modelo URDF visualizado en <i>Gazebo</i>	99
8.3	Simulación en Gazebo y visualizado de datos en RViz.	101
8.4	Pioneer 3 AT simulado en el mapa Willow Garage.	101
9.1	Prueba de SLAM en el simulador Gazebo.	104
9.2	Trayectoria global erronea.	104
9.3	Navegación con mapa final.	105
9.4	Navegación con mapa final.	105
9.5	Mapa con la primera prueba de SLAM.	108
9.6	Mapa tras la segunda prueba de SLAM.	109
9.7	Distribución de puntos de meta en la segunda prueba de resistencia. .	111
A.1	Vista del repositorio de GitHub de este proyecto.	118
A.2	Software Sick SOPAS con el sensor detectado.	123
A.3	Configurando la dirección IP del dispositivo.	123
A.4	Gráfico de la información captada por el sensor en el software SOPAS.	124
B.1	Panel de control del robot Petrois.	125
B.2	Botón de parada de emergencia del robot Petrois.	127
B.3	Trampilla de acceso a las baterías.	128
B.4	Puerto de carga y cargador del robot Petrois.	129
B.5	Panel del ordenador interno (lateral izquierdo) del robot Petrois. . .	129
C.1	Repositorio software <i>pioneer3at_ETSIDI</i>	131

Índice de tablas

3.1	Especificaciones del robot Pioneer 3 AT.	29
3.2	Características del sensor Kinect.	30
3.3	Características del sensor láser Sick LMS100. Basado en [SIC09].	31
3.4	Características del ordenador Intel NUC NUC5i7RYH.	33
6.1	API de <i>rosaria</i> utilizada. Basado en [ROS15].	62
6.2	API de freenect_stack utilizada.	63
6.3	API de LMS1xx utilizada.	64
6.4	API de teleop_p3at	67
6.5	API del nodo <i>moving_alone</i>	68
7.1	API de endurance_test	89
7.2	API de <i>turtlebot_follower</i>	90
7.3	API del nodo <i>soundplay_node</i>	91
7.4	API del nodo <i>recognizer</i>	92
7.5	API del nodo <i>voice_cmd_vel</i>	95
9.1	Primera prueba de resistencia simulada.	106
9.2	Segunda prueba de resistencia simulada.	107
9.3	Tercera prueba de resistencia simulada.	107
9.4	Primera prueba de resistencia real ¹	110
9.5	Segunda prueba de resistencia real.	110
9.6	Tercera prueba de resistencia real.	111

Índice de códigos

4.1	Mandato de consola para instalar la versión completa de ROS Indigo.	38
4.2	Source al setup de ROS Indigo	39
4.3	Instalación y workspace de Catkin	39
4.4	Source al setup de nuestro entorno Catkin	39
4.5	Clonado del repositorio <i>pioneer3at_ETSIDI</i>	39
5.1	Ejemplo de <i>costmap_common_params.yaml</i>	46
5.2	Ejemplo de <i>global_costmap.yaml</i>	46
5.3	Ejemplo de <i>local_costmap.yaml</i>	47
5.4	Ejemplo de <i>local_costmap.yaml</i>	47
5.5	Ejemplo de <i>global_planner_params.yaml</i>	47
5.6	Ejemplo de <i>robot_navigation.launch</i>	48
5.7	Launchfile <i>slam_gmapping</i>	50
5.8	Launchfile para visualizar slam_gmapping en RViz	50
5.9	Ejecución del nodo <i>map_saver</i> para guardar el mapa	50
5.10	Launchfile del nodo <i>amcl</i> utilizado.	52
6.1	Launchfile para RosAria.	62
6.2	Launchfile para Kinect en el paquete <i>freenect_launch</i>	63
6.3	Launchfile para el sensor Láser Sick LMS100.	64
6.4	Launchfile creado para robot Pioneer 3 AT.	65
6.5	Líneas del archivo <i>.bashrc</i> en el ordenador de abordo Intel NUC.	66
6.6	Ejemplo <i>.bashrc</i> en un ordenador externo para realizar comunicación con el máster.	66
6.7	Fragmento de código del nodo <i>teleop_p3at</i>	67
6.8	Fragmento de código del nodo <i>moving_alone</i>	68
7.1	<i>Launchfile</i> del nodo <i>DepthImage_to_LaserScan</i> para obstáculos bajos.	80
7.2	Configuración del <i>global_costmap</i>	81
7.3	Configuración del <i>local_costmap</i>	82
7.4	Configuración del <i>global_planner</i>	84
7.5	Configuración de <i>base_local_planner</i>	85
7.6	Configuración de navegación global.	86
7.7	Configuración de <i>global_costmap</i> para navegación reactiva.	87
7.8	Fragmento de código del nodo <i>nav-waypoints</i>	88
7.9	Archivo <i>launchfile</i> para el nodo <i>endurance_test</i>	89
7.10	Launchfile para <i>turtlebot_follower</i> en el robot Pioneer 3 AT.	90
7.11	Nodo <i>soundplay_node</i> para reproducir sonidos.	91

7.12	Fragmento del nodo <i>voice_cmd</i> utilizando el cliente de sonido de <i>soundplay_node</i>	92
7.13	Paquete necesario para reconocer el micrófono del sensor Kinect.	92
7.14	Archivo <i>launchfile</i> para el nodo <i>recognizer</i>	93
7.15	Fragmento del nodo <i>voide_cmd</i> utilizando las palabras reconocidas por el nodo <i>recognizer</i>	93
7.16	Ejemplo de uso de <i>subprocess</i> en Python lanzando y parando el nodo <i>turtlebot_follower</i>	95
7.17	Archivo <i>voice_cmd.launch</i>	96
8.1	Fragmento de la configuración URDF del robot.	99
8.2	Launchfile para lanzar Gazebo con el modelo del robot y sus sensores.	100
A.1	Instalación y workspace de Catkin	117
A.2	Source al setup de nuestro entorno Catkin	117
A.3	Clonado del repositorio <i>pioneer3at.ETSIDI</i>	118
A.4	Inicializando la herramienta <i>rosdep</i>	119
A.5	Instalando los paquetes de navegación mapeo.	119
A.6	Instalando las dependencias de rosaria.	119
A.7	Instalando las dependencias de <i>pocketsphinx</i> y <i>gstreamer</i>	119
A.8	Instalando las dependencias de <i>audio_capture</i>	120
A.9	Instalando las dependencias de <i>sound_play</i>	120
A.10	Compilando los paquetes del espacio de trabajo Catkin.	120
A.11	Abriendo el <i>.bashrc</i>	122
A.12	Añadidendo las direcciones IP al <i>.bashrc</i>	122
B.1	Abriendo el <i>.bashrc</i>	127
B.2	Añadidendo las direcciones IP al <i>.bashrc</i>	127

Capítulo 1

Introducción

Este proyecto surge como una manera de actualizar y poner en funcionamiento el robot móvil de investigación Pioneer 3 AT, integrarlo en una nueva plataforma robótica como es ROS e implementar la navegación autónoma del robot basada en sensores de percepción del entorno como el sensor Kinect y un sensor láser tipo LIDAR.

En este proyecto se desarrolla e implementa un sistema robótico tomando como base al robot Pioneer y sobre la plataforma ROS desde su inicio de forma que todos los sensores queden integrados dentro del robot y este se encuentre operativo para resolver tareas de navegación. Todo ello con el propósito de servir como plataforma de navegación en futuros proyectos relacionados con la robótica móvil o con el entorno ROS en concreto.

1.1 Motivación del proyecto

Dotar a un robot de la capacidad de navegar autónomamente puede ser una alternativa imprescindible en el caso de que se necesite explorar un entorno que no sea fácilmente accesible para el ser humano o que conlleve cierto riesgo.

Cualquier proyecto que desarrolle la automatización de un proceso es ya una motivación, puesto que se va a diseñar una máquina que sea capaz de realizar una tarea que antes solo podía realizarse por un ser humano. Además, dichas tareas realizadas por un robot pueden realizarse, en principio, con una mayor precisión y con mayor repetibilidad debido a que se elimina el factor del cansancio.

Este proyecto viene motivado por la integración de ROS dentro de una plataforma móvil. Con esta plataforma de desarrollo software podemos explorar un concepto diferente de programación en robótica, que ofrece características como:

- Abstraerse de la programación a bajo nivel.
- Reutilizar software ya desarrollado.
- Interfaz de comunicación común.
- Escalabilidad del sistema.
- Simulación mediante Gazebo.
- Visualización gráfica de la información aportada por sensores.

- Transformación entre los diferentes sistemas de coordenadas.

Estas características hacen que adoptar ROS como plataforma de desarrollo robótico sea una ventaja en cuanto a la facilidad para integrar diferentes dispositivos de un robot, comunicarnos con él a través de TCP/IP, utilizar desarrollos ya existentes... Al ser ROS una plataforma muy viva y dinámica, con muchas personas utilizándolo en universidades y empresas de muchos lugares del mundo, permite que el sistema evolucione e incorpore nuevas características que pueden ser fácilmente integrables en futuras actualizaciones del robot.

Utilizar el sensor Kinect es otra de las motivaciones de este proyecto debido al bajo coste del mismo frente al gran aporte de información que supone percibir el entorno en tres dimensiones. Se puede realizar una navegación basada solamente en este sensor además de reconocer objetos por su forma y realizar el guiado del robot esquivando estos.

La aplicación que más ha servido como motivación para el desarrollo de este proyecto ha sido la automatización de las tareas de conducción de automóviles [Xat11], un sector que se encuentra en auge y que comienza a dar sus primeros pasos en el mundo real [Nev12].

También los robots de exploración espacial de la NASA, en especial a su último rover en Marte, Curiosity [NAS12], que permite explorar el entorno árido de la superficie marciana con un alto grado de autonomía en las labores de inspección y análisis de elementos.

1.2 Objetivos del proyecto

El objetivo de este proyecto es realizar el control de un robot móvil para que sea un robot autónomo, basándose en la información que da la odometría, en la nube de puntos que proporciona un sensor que captura el entorno en 3 dimensiones como el sensor Kinect y en el sensor láser LIDAR. Este control debe realizarse con la ayuda del software ROS, integrándolo como parte del sistema robótico para que sirva de soporte al desarrollo del proyecto.

El robot debe ser capaz de localizarse y situarse en el entorno, el sensor Kinect ofrecerá información sobre los objetos alrededor del robot, y el software desarrollado para ROS deberá ser capaz de hacer un control reactivo sobre los movimientos para permitir al robot moverse por interiores y guiarlo hacia un punto indicado.

A la finalización del proyecto, el sistema robótico debe quedar integrado de tal forma que en futuros desarrollos pueda utilizarse como plataforma móvil y permita abstraerse de las tareas de orientación y navegación.

El alcance del proyecto requiere múltiples fases de trabajo:

En primer lugar, se necesita un conocimiento previo del sistema hardware, como es el robot Pioneer 3 AT así como el sensor Kinect y el sensor láser. Cómo integrar estos elementos y acceder a la información que aportan sus sensores y comandar al robot para que realice movimientos.

En segundo lugar, requiere un conocimiento del entorno de desarrollo ROS. Las herramientas software de las que dispone, el funcionamiento interno y la manera de programar e interaccionar con los diferentes elementos, el aprendizaje y comprensión.

En tercer lugar, incorporar los sensores pertinentes para obtener la información que permita al robot posicionarse en el entorno.

En cuarto lugar, implementar los ajustes necesarios para que el robot pueda operar utilizando el entorno de navegación ofrecido por ROS. Realizar una configuración óptima de los sensores y realizar las pruebas reales para el cálculo de trayectorias y el control reactivo del robot.

Por último, realizar la integración del sistema dentro de la plataforma robótica. Disponer de todo lo necesario para que el robot quede totalmente adaptado al sistema ROS e integrado con el sensor Kinect y el sensor láser.

Objetivos específicos

A continuación se citan los puntos más importantes para cumplir con el objetivo de este proyecto:

- a) Familiarizarse con el control de los robots móviles y más concretamente en el control del robot Pioneer 3 AT. Familiarizarse con el sensor Kinect y con el software necesario para su uso. Y familiarizarse con el framework ROS y sus herramientas para el desarrollo de sistemas robóticos.
- b) Detección de obstáculos simples con el sensor Kinect para su inclusión posterior en el sistema de navegación del robot.
- c) Telecontrol del robot Pioneer 3 AT a partir del software ROS para realizar un controlador manual desde un ordenador externo.
- d) Realizar un sistema de navegación autónomo que sea capaz de dirigir el robot basándose en la información aportada por el sensor Kinect y otro tipo de sensores embebidos.

De los puntos anteriores podemos desgranar algunas fases intermedias como son:

- a) Comprender el funcionamiento de ROS y la integración e interacción del software desarrollado bajo este entorno.
- b) Control del movimiento del robot Pioneer 3 AT a través de ROS, así como la obtención de la información de la odometría, estado de la batería, encendido de motores...
- c) Puesta en marcha el sistema de navegación para robots de ROS conocido como "Navigation Stack" y exploración de las capacidades del sistema.
- d) Valorar el uso de sensores adicionales y buscar una disposición óptima de los mismos para integrarlos en el sistema de navegación y en la arquitectura hardware del propio robot.

1.3 Estructura del documento

A continuación y para facilitar la lectura del documento se detalla el contenido que consta en cada uno de los capítulos.

- En el capítulo 1 se realiza una introducción del proyecto, un análisis de las motivaciones y objetivos del mismo.
- En el capítulo 2 se hace un repaso del estado del arte de la robótica móvil, los tipos de robots y principales sensores utilizados así como las aplicaciones de la robótica móvil.
- En el capítulo 3 se expone el planteamiento del proyecto, los pasos seguidos junto con un análisis de tiempos y las tecnologías utilizadas.
- En el capítulo 4 se habla de los conceptos relacionados con ROS, su filosofía de funcionamiento y sus herramientas así como la estructura del desarrollo software llevado a cabo en este proyecto.
- En el capítulo 5 se abordan los conceptos relacionados con la navegación, su enfoque teórico y las particularidades de funcionamiento de la navegación en ROS.
- En el capítulo 6 se exponen los primeros nodos de control a bajo nivel, el nodo de teleoperación y el de navegación estimada.
- En el capítulo 7 se expone la implementación del sistema pasando por todas sus fases de desarrollo y las soluciones que se han llevado a cabo.
- En el capítulo 8 se exponen las configuraciones y el uso de los simuladores robóticos utilizados para dar apoyo al desarrollo del proyecto.
- En el capítulo 9 se exponen las pruebas del sistema en navegación real y simulada, se exponen los resultados y se discuten los mismos.
- En el capítulo 10 se exponen las conclusiones y se plantean los futuros desarrollos a implementar en el robot móvil.

Adicionalmente existen 3 anexos que sirven de apoyo a las explicaciones del proyecto y donde se realiza un manual aclaratorio del funcionamiento del sistema robótico objeto de este proyecto.

Capítulo 2

Estado del arte

La robótica móvil vive actualmente un momento de gran desarrollo para multitud de aplicaciones en entornos diversos, desde espacios abiertos con orografía accidentada y condiciones climáticas adversas [Gui13] [PBR06], entornos controlados y espacios interiores conocidos como la automatización de tareas de almacenaje de productos [Reu12], hasta orientación y exploración de espacios interiores desconocidos con robots usados para la creación de mapas de edificios.

De la misma forma, el interés en robots que sea capaces de reproducir las capacidades de un ser humano e incluso que pueda dar asistencia ya sea en entornos conocidos o no abre un área de posibilidades en las que los robots móviles cobran importancia.

Los avances tanto en las características hardware como software son notables aunque estas suelen variar dependiendo de la aplicación a la que un robot esté destinado.

Este capítulo trata de hacer un recorrido por la historia de la robótica y un resumen del estado actual de la robótica móvil.

2.1 Robótica Móvil

Prácticamente cualquier robot consta de alguna parte móvil que le permite realizar algún tipo de tarea, sin embargo nos referimos a la «robótica móvil» como el área de la robótica que estudia los robots con capacidad para trasladarse.

Los robots móviles son aquellos que tienen la capacidad de desplazarse utilizando algún sistema locomoción como pueden ser ruedas, patas, orugas... Estos robots se diferencian respecto a los robots fijos que permanecen anclados a una superficie, como un brazo robótico industrial. Tampoco debe confundirse con los robots destinados a desplazarse por otros medios como agua o aire, ya que estaríamos entrando en el área de la robótica acuática/submarina o robótica aérea respectivamente. Podemos decir en este proyecto que la robótica móvil se refiere a robots que se mueven en el entorno terrestre.

Las aplicaciones dentro de la robótica móvil pueden ser múltiples: exploración de entornos peligrosos, exploración espacial o minera, misiones e búsqueda y rescate de personas, telepresencia, automatización de procesos, transporte autónomo, vigilancia, inspección y reconocimiento del terreno o utilizados como plataformas móviles que incorporan otros sistemas robóticos como podrían ser un brazo manipulador.

2.2 Hardware en robótica móvil

Como hemos indicado previamente, la configuración hardware de un robot móvil varía dependiendo de la aplicación a la que vaya destinado. Es cierto que lo ideal para un robot sería disponer de una configuración hardware común que fuera polivalente en los diferentes terrenos y situaciones, sin embargo, debido a la variedad de aplicaciones y dado que un robot suele destinarse a tareas específicas, la elección del hardware que mejor se adapta es una tendencia común en robótica.

Para seleccionar el hardware debemos valorar el tipo de actuador que se requiere, entendiéndose por actuador al dispositivo que genera el movimiento de los elementos que hacen que el robot móvil se desplace. En robótica móvil suelen utilizarse los actuadores de tipo eléctrico, ya que ofrecen unas prestaciones de potencia, controlabilidad y coste adecuados. Además, ofrecen la posibilidad de que la alimentación esté integrada en el robot, haciéndolo independiente de una fuente de energía accesoria.

Los actuadores eléctricos son, por tanto, los más utilizados. En concreto, los motores de corriente continua (Figura 2.1) ofrecen un fácil control y acoplamiento a un encoder. Los encoder son sensores de posición que permiten conocer el giro de un eje de rotación. Estos sensores son muy importantes en robótica móvil, ya que a partir de la información que arrojan el robot tiene conciencia de su posición relativa, en el caso de los encoders incrementales, o su posición absoluta, en el caso de los encoders absolutos (Más detalladamente en el apartado 2.3).



Figura 2.1: Motor de corriente continua con encoder.

Existen otros tipos de actuadores eléctricos que se utilizan en robótica, como puede ser el caso de los motores paso a paso, sin embargo su baja velocidad de giro no los hacen adecuados para robots móviles.

La disposición de los actuadores determina la configuración del robot. Centrándonos en robots que se desplazan mediante ruedas y descartando los robots con patas, podemos distinguir las siguientes configuraciones: Ackerman, triciclo clásico, tracción diferencial, skid-steer, síncrona y omnidireccional.

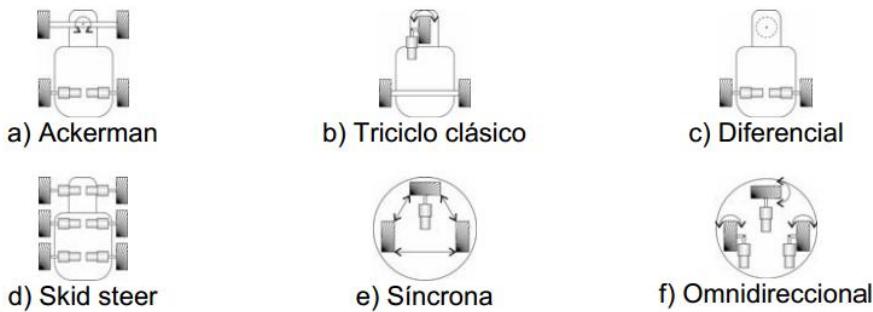


Figura 2.2: Configuraciones hardware. Basado en [SOGSBS⁺¹⁰].

a. Configuración Ackerman

Consta de cuatro ruedas. Las ruedas motrices son las traseras o delanteras, y éstas últimas se encargan además de la dirección. Permite un desplazamiento a altas velocidades y la posibilidad de realizar giros con estabilidad. Esta configuración es la que se utiliza en la industria del automóvil.

b. Triciclo clásico

Consta de tres ruedas. Las ruedas motrices pueden ser las dos ruedas traseras o solo la delantera, que se encarga de la dirección. Este es el caso de los triciclos y de algunas bicicletas. Esta configuración ofrece alto grado de maniobrabilidad penalizando la estabilidad del conjunto y realizar giros de 90°.

c. Configuración diferencial

Consta de dos ruedas colocadas en el eje perpendicular a la dirección de desplazamiento del robot. Cada rueda es controlada por un motor, de tal forma que la diferencia de velocidad de giro de una rueda respecto a otra determina el giro, avance o retroceso del robot. Los robots que presentan esta configuración suelen utilizar una tercera rueda que gira libremente que sirve como apoyo (rueda loca). Es la configuración típica de las sillas de ruedas y su característica principal es que permite realizar giros completos sobre sí mismo.



Figura 2.3: Robot de configuración diferencial Pioneer 3 DX.

d. Skid steer

Consta de cuatro o más ruedas, todas ellas motrices, y su principio de funcionamiento es el mismo que el utilizado en la configuración diferencial. Esta configuración

presenta las ventajas de la configuración diferencial, pudiendo realizar giros sobre el eje del robot, pero presenta la desventaja de que las ruedas deben deslizarse lateralmente, por tanto existe un rozamiento que varía en función de la inclinación el tipo de terreno.

Proporciona mucha tracción y estabilidad y suele encontrarse en aplicaciones relacionadas con la exploración, vehículos obra o vehículos todo terreno (Figura 2.4).



Figura 2.4: Ejemplos de configuración skid-steer: Cargador frontal, Robotnik Guardian, Pioneer 3 AT.

Este sistema es el que se utiliza también en los tanques de guerra, aunque en vez de neumáticos se utilizan orugas, denominado configuración por deslizamiento de cintas [dS07].

e. Configuración síncrona

Conformado por tres o más ruedas acopladas mecánicamente y dotadas de tracción, este sistema permite que todas las ruedas roten en la misma dirección y giren a la misma velocidad (Figura 2.5) manteniendo al cuerpo del robot con una orientación fija. Es utilizada ampliamente en robots móviles de interior, aunque está siendo desplazada por la configuración omnidiireccional.

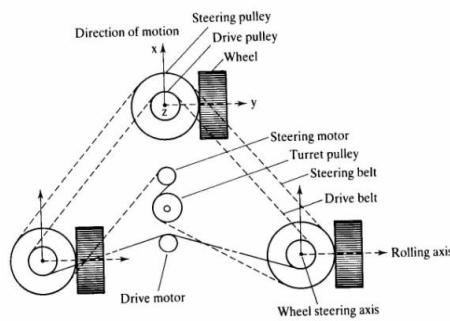


Figura 2.5: Configuración síncrona.

f. Configuración omnidiireccional

Consta de 3 ruedas cada una con un motor independiente, que permiten el desplazamiento en cualquier dirección (Figura 2.6). Las ruedas omnidiireccionales constan de una serie de rodillos con el eje de rotación perpendicular a la dirección de avance.

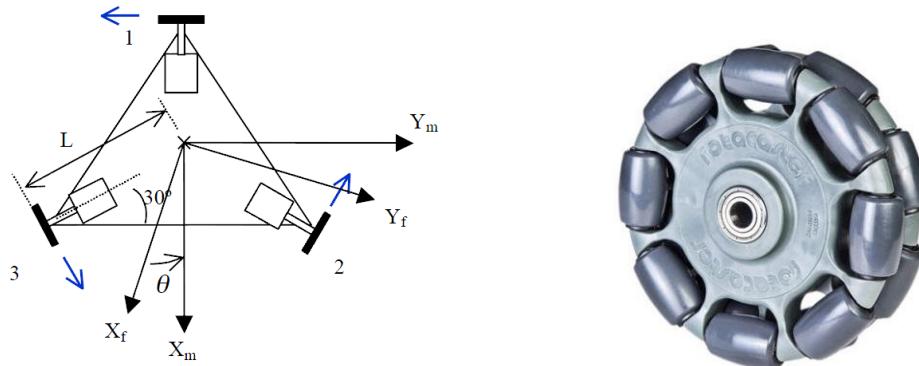


Figura 2.6: Configuración síncrona y rueda omnidireccional.

Esta configuración diferencial empieza a utilizarse en sistemas de 4 ruedas con las denominadas "Mecanum Wheels" [DL91], que son ruedas similares a las omnidireccionales pero con los rodillos colocados en cierto ángulo (Figura 2.7). La combinación de los giros de cada una permiten al robot moverse en cualquier dirección.



Figura 2.7: Robot Uranus con ruedas tipo Mecanum.

2.3 Sensores en robótica móvil

Para que un robot pueda realizar tareas con una determinada precisión y velocidad debe conocer el entorno del sistema en el que se quiera actuar así como el estado del robot en ese sistema.

Existen dos tipos de sensores, los sensores propioceptivos, que aportan información sobre el estado del robot, y los exteroceptivos, que aportan información del entorno en el que se encuentra el robot.

2.3.1 Sensores propioceptivos

Dentro de los sensores propioceptivos, los sensores de posición primordiales son los encoders, tanto los de tipo incremental como los de tipo absoluto (Figura 2.8). Su funcionamiento se basa en un foto-emisor y un foto-receptor que detectan el paso o no de luz a través de un disco con ciertas marcas acoplado al eje de giro del actuador.

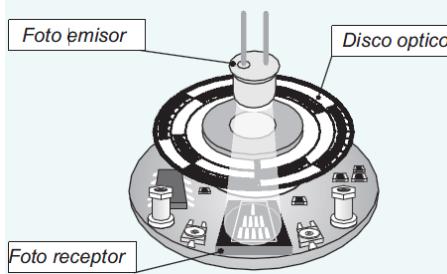


Figura 2.8: Esquema de un encoder absoluto

Los sensores de velocidad son similares a los encoders pero miden la velocidad de giro del eje del actuador. La tacogeneratriz proporciona una tensión proporcional a la velocidad de giro.

Los sensores acelerómetros o inclinómetros, permiten conocer la inclinación del robot en cada uno de sus ejes, así como las aceleraciones producidas por su propio desplazamiento.

Existen otros sensores más sofisticados como las Unidades de medida inercial (IMU) (Figura 2.9). Son dispositivos que combinan las medidas de un giróscopo y varios acelerómetros para determinar la posición relativa (x , y , z) y la orientación (roll, pitch, yaw), velocidad y aceleración respecto a un sistema de referencia.

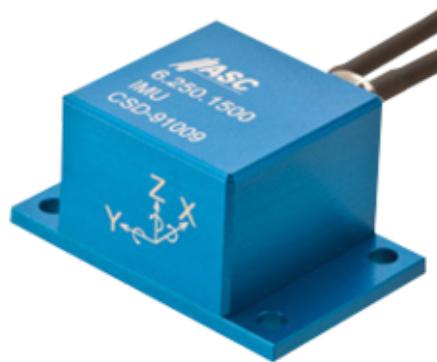


Figura 2.9: Unidad de medida inercial, IMU

Debido a que la aceleración se ha de integrar dos veces para obtener la posición, el error crece de forma cuadrática. Luego para largos períodos de operación las unidades IMU se deben de resetear con sensores de posicionamiento absoluto.

2.3.2 Sensores exteroceptivos

Los sensores externos son aquellos que nos aportan información sobre el estado del robot respecto al entorno o que nos dan información sobre lo que ocurre alrededor de este.

- **Sensores de presencia:** Son sensores de tipo inductivo, capacitivo, óptico o mecánico. Sea cual sea la naturaleza del sensor, su función es la de detectar presencia. Un ejemplo de aplicación a un robot móvil sería una serie de sensores

de presencia mecánicos, denominados "fin de carrera", colocados en la parte delantera, de modo que al tocar algún obstáculo se tuviera conocimiento de su presencia.

- **Sensores de posicionamiento absoluto:** El sensor más conocido de este tipo es el GPS (Global Positioning System) que permiten determinar la posición de un objeto en todo el mundo, normalmente con una precisión de metros. El GPS funciona con una red de satélites con trayectorias sincronizadas que cubren toda la superficie de la tierra. El GPS recibe señales de los satélites y calcula el intervalo de tiempo entre cada una de ellas, de esta forma obtiene la posición por triangulación.

Los sensores GPS se utilizan en robots móviles que operan en el exterior y suelen combinarse con otros sensores que ofrezcan una mayor precisión.

Existen otros sensores de posicionamiento absoluto como los sistemas IPS (Indoor Positioning System) que sirven para la localización en interiores y puede utilizar ondas de radio, campos magnéticos, señales acústicas... Dentro de esta categoría no existe un estándar y coexisten diferentes sistemas como el sistema Bat, basado en ultrasonidos o el sistema Symeo, basado en radiofrecuencia [CR10] por citar algunos de ellos.

- **Sensores de distancia:** Son aquellos que nos dan una referencia de la longitud que existe a los objetos cercanos. Es el caso de los sensores de ultrasonidos, donde un emisor emite una onda ultrasónica y cuando es reflejada por un objeto se puede determinar la distancia a la que se encuentra midiendo el tiempo que tarda el sonido en ir y volver. Los sensores de distancia también pueden ser infrarrojos, funcionando de manera análoga. En robótica móvil es común colocar varios sensores de este tipo alrededor del robot.

Existen sensores de distancia que utilizan tecnología láser para determinar la longitud de un punto a otro, se denominan Scanners láser o LIDAR. Estos sensores emiten rayos láser en un plano de 2 dimensiones y en un rango determinado, y midiendo el tiempo de vuelo del haz láser son capaces de obtener una medida muy precisa de la distancia.

- **Sensores de distancia basados en visión:** Existen otro tipo de sensores de distancia que permiten obtener distancias a puntos de manera tridimensional. Los pasivos, como las cámaras estereoscópicas, que utilizan un sistema de doble cámara (Figura 2.10).



Figura 2.10: Cámara estereoscópica del robot PR2.

Estos sensores son capaces de obtener imágenes 3D con la información de dos imágenes tomadas a cierta distancia una de otra. Es el sensor más parecido a la visión humana.

Por otro lado están los sensores activos, como las cámaras por tiempo de vuelo (Time-of-flight) que obtienen distancias a los puntos de una imagen basándose en el tiempo que tarda la luz en alcanzar los objetos, o los sensores de tipo infrarrojo, como es el sensor Kinect, de gran interés debido a su bajo coste y su buena respuesta.

Kinect es un dispositivo desarrollado por PrimeSense y distribuido por Microsoft para la videoconsola Xbox 360 (Figura 2.11).

Inicialmente permitía controlar e interactuar con la consola XBOX sin necesidad de tener contacto físico con un controlador. Este sensor permite reconocer gestos, comandos de voz, objetos e imágenes; esto hace que tenga mucho interés en el mundo de la robótica.



Figura 2.11: Sensor de 3 dimensiones Kinect para Xbox 360.

Para captar el entorno en 3 dimensiones, Kinect incluye una cámara de vídeo RGB, un emisor de haz infrarrojo y una cámara infrarroja.

2.4 Aplicaciones actuales de la robótica móvil

Algunas de las aplicaciones del área de la robótica móvil ya han sido mencionadas con anterioridad en este documento, estas van enfocadas a sustituir la labor que realiza el ser humano en situaciones de riesgo o en tareas repetitivas que aportan poco valor.

Una de las aplicaciones más famosas sobre robótica móvil son los Rovers de exploración espacial de la NASA "Spirit" y "Opportunity" dentro de la misión "Mars Exploration Rover" lanzada en 2003. Estos robots disponen de sistemas de navegación y exploración ideados para sus misiones en la superficie del planeta Marte, con el objetivo de analizar el entorno y los materiales de sus rocas y cráteres y enviar la información de vuelta a la Tierra [NAS03].

Un tercer robot no tripulado fue enviado con posterioridad a la superficie del planeta rojo. El robot "Curiosity" forma parte de la segunda generación de robots de exploración espacial y aterrizó en Marte en el año 2012. La misión "Mars Science Laboratory" ha permitido descubrir la existencia de antiguos lagos en la superficie del planeta [NAS12].



Figura 2.12: Imagen tomada a sí mismo por el robot Curiosity en la superficie marciana.

Otras aplicaciones conocidas de la robótica móvil son los robots de búsqueda, reconocimiento y desactivación de explosivos. Robots como el iRobot 510 PackBot (Figura 2.13), desarrollado por iRobot Corporation, o TALON (Figura 2.13), desarrollado por QinetiQ North America, son ejemplos de cómo la robótica móvil es una herramienta muy valiosa en situaciones peligrosas o en sectores como seguridad y defensa.

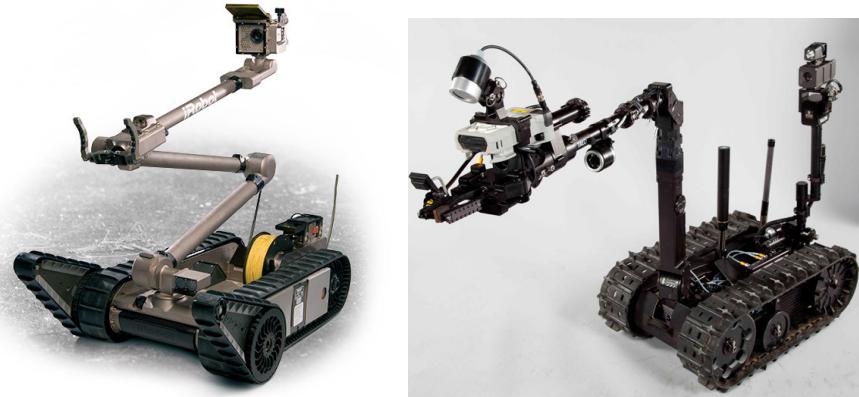


Figura 2.13: Robots de desactivación de explosivos: iRobot 510 Packbot y TALON.

También existen robots ideados para realizar tareas en entornos peligrosos o en situaciones de desastre. Como ejemplo podemos citar la catástrofe que sufrió Japón el 11 de Marzo de 2011, en la que un terremoto causó daños catastróficos en la central nuclear de Fukushima [Paí11]. Los niveles de radiación fueron tan altos que solo los robots eran capaces de entrar a valorar la situación de la central.

Estos robots, preparados para aguantar la radiación y realizar tareas de exploración y limpieza fueron Quince (Figura 2.14), desarrollado por Chiba Institute of Technology [NKO+11], un robot móvil capaz de subir y bajar escaleras y operar en las plantas superiores de la central nuclear.

Y Raccoon (Figura 2.14), desarrollado por Tepco, equipado con dos cabezales móviles preparados para aspirar y limpiar, encargado de recuperar el polvo contaminado del edificio del reactor 2.

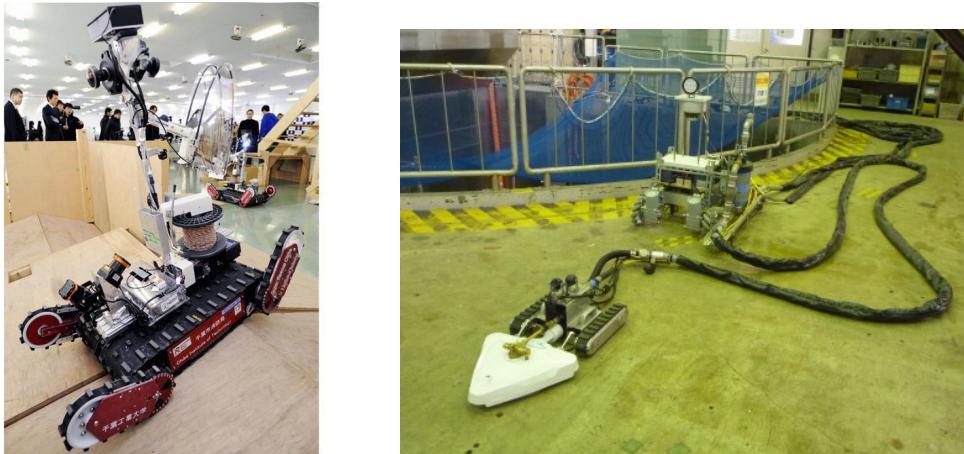


Figura 2.14: Robot Quince (izq.) y robot Raccoon (dcha.).

Siguiendo con la aplicación de robots móviles en tareas de limpieza, podemos destacar la popularización de los robots domésticos de tipo aspiradora, que se encargan de las tareas repetitivas del entorno del hogar, como el iRobot Roomba (Figura 2.15).



Figura 2.15: Robot para limpieza del hogar Roomba, de iRobot.

También existen aplicaciones de robots móviles en el ámbito de la agricultura. La empresa Robotnik [Rob02] tiene en marcha diferentes programas para incorporar sus robots en tareas de recolección o inspección de la cosecha. Su proyecto AgriRobot (Figura 2.16), investiga el aspecto de la interacción humano-robot (Human-Robot Interface, HRI, en inglés). Para ello se sirven del robot Summit X Lincorporado con 4 ruedas motoras de alta potencia. El robot contiene un pulverizador eléctrico de 10 litros, tiene un sistema de visión, navegación y localización, y utiliza el software ROS.



Figura 2.16: Robot pulverizador de aplicación agrícola, AgriRobot.

Al igual que el proyecto VinBot (Figura 2.17), de la misma empresa. Un robot móvil autónomo todo terreno dotado con un conjunto de sensores capaces de capturar y analizar imágenes de viñedos y datos en 3D mediante el uso de aplicaciones de *cloud computing*. Su finalidad es determinar el rendimiento de los viñedos y compartir esta información con los viticultores.



Figura 2.17: Robot de inspección de viñedos, VinBot.

En el apartado de la robótica social, podemos hablar de robots móviles con ruedas destinados a la asistencia de personas en lugares públicos como centros comerciales, aeropuertos u hospitales, o también como asistentes domésticos en hogares de personas con movilidad reducida. Es el caso del robot Maggie [AMRS11], desarrollado por la Universidad Carlos III de Madrid, o el robot de asistencia social ROSA, desarrollado por la Universidad Politécnica de Madrid (Figura 2.18).



Figura 2.18: Robot social Maggie y robot de asistencia ROSA.

Existen otro tipo de robots con altas capacidades que realizan la función de un robot móvil, como el robot de investigación PR2 (Figura 2.19), dotado con un sistema de desplazamiento omnidireccional.

Este robot, desarrollado por los investigadores de *Willow Garage*, fue creado para proporcionar una plataforma común junto con ROS (Robot Operating System), sobre la que realizar investigaciones que ayuden al desarrollo software de aplicaciones

robóticas.



Figura 2.19: Robot PR2 desarrollado por Willow Garage.

Finalmente, podemos destacar una de las aplicaciones que más han llamado la atención en la sociedad, los coches sin conductor. Potenciados por los desarrollos de la empresa Google, cuyo proyecto consiste en combinar la información obtenida del servicio de mapas de la compañía con la inteligencia artificial, estos coches (Figura 2.20) suponen un cambio notable a nivel de seguridad en el transporte urbano de nuestra sociedad.

Los vehículos van equipados con cámaras de vídeo, un sensor láser de 360° colocado en la parte superior del vehículo, sensores radar, sensores de odometría en las ruedas y de posición con localización GPS [Gui11].



Figura 2.20: Sensores del coche autónomo de Google.

Estos coches disponen actualmente de permiso para circular en algunos estados de Estados Unidos tras haber superado 1.800.000 millas desde que el programa (Google Self-Driving Car Project) comenzase en 2009 [Goo09].

En conclusión, podemos decir que los campos de aplicación de la robótica móvil son amplios, variados y con grandes perspectivas de futuro. Sin embargo, el mercado de la robótica civil a penas acaba de comenzar con la aparición de los primeros robots domésticos y los robots de carácter social. En otros campos, vemos que existe un auge de la robótica móvil y que a medida que los sistemas robóticos avanzan surgen nuevas posibilidades donde aplicarla.

Capítulo 3

Planteamiento del proyecto

En esta capítulo se expone cuál ha sido el planteamiento incial del proyecto, un análisis de tiempos, las tecnologías y equipos utilizados y el hardware que forma parte del proyecto en sí mismo.

3.1 Planteamiento preliminar

El robot sobre el que se ha trabajado es el Pioneer 3 AT, de la empresa Adept Mobile Robots, cuyas características se detallarán más adelante. La configuración del sistema motriz es de tipo skid-steer y será determinante a la hora de realizar el control del desplazamiento.

Para realizar la teleoperación del robot, se han utilizado las herramientas de comunicación de ROS, que hacen que la ejecución de los diferentes nodos de forma distribuida entre equipos se realice de forma transparente para el usuario. Con esta característica es posible desarrollar con facilidad un sistema de telecontrol sin preocuparnos en exceso por la implementación de la comunicación entre equipos.

Para la navegación se pretende que el robot base sus movimientos en un sistema reactivo, es decir, que el robot base su navegación principalmente en la información captada por sus sensores y no en un mapa preestablecido. El control en navegación del robot se basa en la funcionalidad "Navigation Stack" de ROS, que también será explicada en detalle más delante.

El desarrollo principal para la navegación utiliza la información aportada por el sensor Kinect y el sensor láser, ya que ambos proporcionan información muy valiosa debido a sus diferentes características.

Seguidamente, se han realizado los ajustes pertinentes en la navegación del robot, para la cual se ha seguido el concepto de mapas de coste y descomposición en mapas de celdillas. También se ha valorado la disposición de ambos sensores para capturar el entorno, así como el tratamiento dispar de los datos capturados por cada uno de ellos. De esta forma logramos que no se produzcan detecciones de objetos de manera duplicada y que no haya discrepancias entre los obstáculos que detecta un sensor respecto al otro¹.

Finalmente, se han incorporado características adicionales que aportan valor al desarrollo del proyecto, como el uso del simulador Gazebo o la interacción con el robot mediante comandos de voz y sintetizado de voz.

¹De especial interés en la incorporación de obstáculos al mapa mediante el uso de Costmaps en el sistema de navegación de ROS

3.2 Planificación del proyecto

En este apartado se desarrollan las fases por las que ha pasado este proyecto y realizaremos un análisis de tiempos.

Fases de desarrollo:

- Fase inicial: Familiarización con el entorno ROS y elección de herramientas.
 - i. Herramientas proporcionadas por ROS para evaluar los datos que pueda manejar el robot: *RViz*, *rqt_graph*, *map_server*, *Topics*...
 - ii. Para el control del movimiento del robot se ha utilizado el nodo RoSaria debido a sus posibilidades.
 - iii. Para acceder a la información de la Kinect se utilizan los drivers *libfreenect* por ser librerías de código libre integradas en ROS y de las cuales se ha hecho uso extenso en otros proyectos.
- Segunda fase: Realización del nodo de teleoperación y comunicación entre equipos conectados a la misma red.
 - i. Configuración de equipos en red para acceder a la información publicada por nodos que se ejecuten en varias máquinas [ROS14].
 - ii. Partiendo del nodo de teleoperación de *turtlesim* [Fau11], se ha realizado un nodo similar para nuestro robot.
- Tercera fase: Incorporación de los sensores al robot y acceso a los datos.
 - i. Para el sensor Kinect, se ha realizado un adaptador para conectarlo a la alimentación del robot. Utilizando el nodo *freenect_stack* [Kha11], accedemos a la nube de puntos y la imagen.
 - ii. Se ha utilizado el nodo *LMS1xx* [Ban11] para la puesta en marcha del láser y el acceso a los datos.
 - iii. Incorporación de sistemas de referencia "*base_link*", "*laser*", "*camera_link*" y sus transformadas mediante el paquete *tf* [FMEM11].
 - iv. Visualización del conjunto de datos junto con los ejes de referencia en *RViz*.
- Cuarta fase: Incorporación del sistema de navegación y ajuste de los parámetros
 - i. Calibrado de los encoders de las ruedas del robot y ajuste de la odometría mediante *RosAria*.
 - ii. Incorporación del sistema de navegación ROS de forma básica.
 - iii. Navegación utilizando el sensor Kinect y el sensor Sick y generando de mapas mediante SLAM.
 - iv. Ajuste de los planificadores de trayectoria del robot y parámetros de giro y control.
- Quinta fase: Ajuste de la navegación y simulación mediante *Gazebo*.
 - i. Navegación en modo global (utilizando un mapa guardado) y en modo local (completamente reactivo).

- ii. Puesta en marcha del simulador Gazebo y configuración del robot en el entorno.
- Sexta fase: Remodelado de la estructura del robot
 - i. Disposición de los sensores de manera óptima e integración de los mismos.
 - ii. Remodelado de la estructura física del robot.
 - iii. Incorporación del ordenador compacto Intel NUC.
- Séptima fase: Nuevas funcionalidades y toma de datos.
 - i. Incorporación de la funcionalidad de guiado (follower), adaptada a partir del robot *Turtlebot*.
 - ii. Interfaz de comandos por voz y sintetizador de texto a voz.
 - iii. Pruebas reales, recogida y análisis de los datos.

Análisis de tiempos:

Este proyecto fin de grado comenzó en Noviembre de 2014 y terminó en Febrero de 2015.

Durante el primer mes de Noviembre se estuvo recopilando información sobre ROS y su funcionamiento, los desarrollos existentes aplicados a robots reales y la filosofía del sistema.

En el mes de Diciembre se comenzó a trabajar con el robot, comprobando que todos los elementos se encontraban en correcto funcionamiento y se instaló el sistema operativo en su ordenador de abordo

Durante el mes de Enero se pudo avanzar menos debido a los exámenes y trabajos de las últimas asignaturas.

En el mes de Febrero se retomó el trabajo, empezando por una primera toma de contacto con la librería Aria y la ejecución de movimientos desde un ordenador externo conectado vía puerto serie.

Durante el mes de Marzo, el robot comenzó a funcionar con ROS, realizando los primeros movimientos con control por teclado. Seguidamente se realizó el nodo de telecontrol y un nodo para realizar movimientos basados tan solo en la odometría.

En el mes de Abril, se comenzaron a probar la compatibilidad con ROS de la cámara Kinect y el sensor Láser. Acto seguido, comenzaron las primeras pruebas de navegación autónoma.

En el mes de Mayo se realizaron las pruebas de navegación con el portátil incorporado en el cuerpo del robot. Durante ese mes se realizaron diferentes ajustes de navegación así como mapas mediante SLAM.

En los meses de Junio y Julio, siguieron los ajustes en la navegación, tanto en el planificador de trayectoria como en los mapas de coste, así como en el sensor Kinect para la detección de obstáculos a diferente altura. Además se incorporó la funcionalidad de seguimiento.

En Julio también comenzaron las primeras pruebas de comandos de voz y la sintetización de voz.

Durante ese mes y el mes de Agosto, se comenzó a redactar gran parte del trabajo en esta memoria, donde se organizó la estructura del proyecto y la información a incluir.

En el mes de Septiembre se decidió incorporar un ordenador más potente al robot y reestructurar su chasis para dejar el sistema desarrollado integrado de manera permanente. También se utilizó el array de micrófonos del sensor Kinect para los comandos de voz.

Durante el mes de Octubre se organizó la estructura del proyecto y se puso en marcha el simulador Gazebo. A continuación se realizó el ajuste de los sensores y la optimización del sistema de navegación. En paralelo se realizaron las modificaciones mecánicas y estructurales para la integración de los sensores y el ordenador en el robot.

Durante el mes de Noviembre se realizó un pequeño parón a nivel de software, se continuó con la parte mecánica y con la redacción de la memoria de este proyecto.

A continuación, comenzaron a realizarse las pruebas reales con la nueva configuración en el robot y la redacción de la memoria.

A continuación se muestra un diagrama de Gantt (Figura 3.1) con el análisis de tiempos de las diferentes tareas.

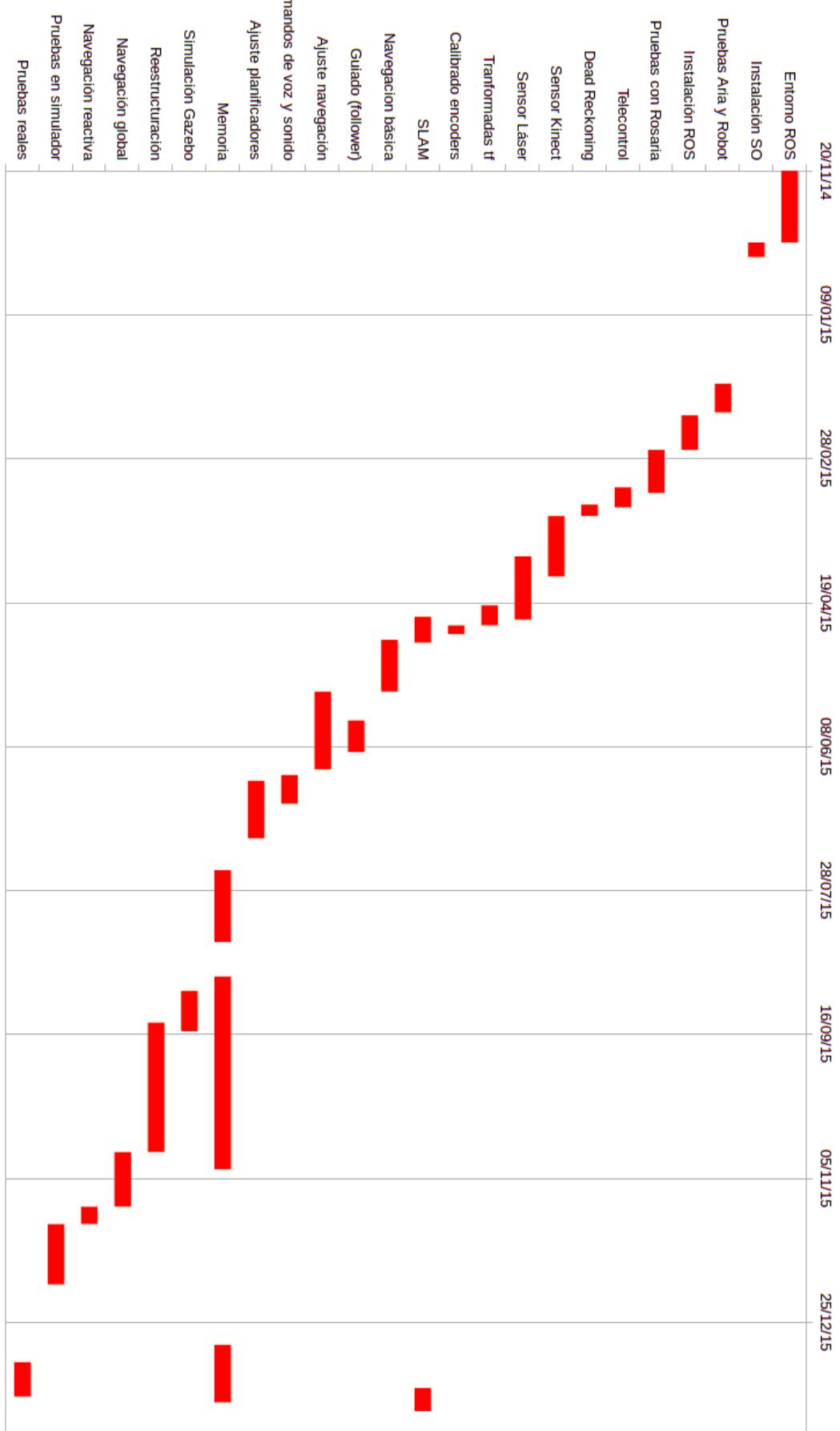


Figura 3.1: Diagrama de Gantt de la evolución del proyecto.

3.3 Tecnologías y herramientas empleadas en el proyecto

En esta sección se describen tanto las tecnologías como las herramientas utilizadas en el desarrollo del proyecto.

3.3.1 Robot Operating System

El Sistema Operativo Robótico [ROSa] (conocido en inglés como Robot Operating System o ROS) es un framework para el desarrollo de software para robots que provee la funcionalidad de un sistema operativo [QCG⁺09]. ROS fue desarrollado originalmente en 2007 por el Laboratorio de Inteligencia Artificial de Stanford para dar soporte al proyecto del Robot con Inteligencia Artificial de Stanford [NQG⁺08]. Desde 2008, el desarrollo continua principalmente en Willow Garage, un instituto de investigación robótico con más de veinte instituciones que colaboran conjuntamente.

ROS provee los servicios estándar de un sistema operativo como abstracción del hardware, control de dispositivos de bajo nivel, implementación de funcionalidad de uso común, paso de mensajes entre procesos y mantenimiento de paquetes. Está basado en una arquitectura de nodos interconectados que pueden mandar, recibir y multiplexar mensajes de sensores, control, estados, planificaciones y actuadores, entre otros. La librería está orientada para un sistema UNIX (Ubuntu (Linux)) aunque también se está adaptando a otros sistemas operativos como Fedora, Mac OS X, Arch, Gentoo, OpenSUSE, Slackware, Debian o Microsoft Windows, considerados como 'experimentales'.



Figura 3.2: Logo de ROS.

ROS consta de dos partes básicas: la parte del sistema operativo, ros, como se ha descrito anteriormente y ros-pkg, una suite de paquetes aportados por la contribución de usuarios (organizados en conjuntos llamado en inglés "stacks") que implementan las funcionalidades tales como localización y mapeo simultáneo, planificación, percepción, simulación, etc. Este tipo de paquetes favorecen el desarrollo rápido de otros robots, consiguiendo que el código pueda reutilizarse gracias a su sistema de nodos, que mantienen cada funcionalidad desacoplada.

ROS ofrece principalmente dos lenguajes de programación para acceder a su API (Application Programming Interface) completa. Esos lenguajes son C++ y Python [ROS13].

ROS es software libre bajo términos de licencia BSD. Esta licencia permite libertad para uso comercial e investigador. Las contribuciones de los paquetes en ros-pkg están bajo una gran variedad de licencias diferentes.

Actualmente ROS es mantenido y desarrollado por Open Source robotics Foundation [OSF], una organización independiente sin ánimo de lucro fundada por miembros de la comunidad robótica a nivel global.

3.3.2 Lenguaje de programación C++

El lenguaje C++ es un lenguaje orientado a objetos, y como tal, tiene como objetivo la reducción del tiempo de desarrollo aumentando la eficacia del proceso de generación de los programas gracias a la reutilización de código.

Como consecuencia, los programas tienden a tener menos líneas de código y con más facilidad de introducir elementos nuevos escritos por otras personas.

Al tratarse de un lenguaje compilado, presenta una buena eficiencia en tiempo de ejecución frente a los lenguajes interpretados.

En sistemas operativos basados en Linux, el lenguaje C++ se compila bajo el compilador GCC (GNU Compiler Collection).

Dentro del desarrollo software en C++ para ROS (roscpp [ROSb]), existe una amplia interfaz para acceder a las diferentes funcionalidades y comunicarse con nodos desarrollados tanto en C++ como en Python.

3.3.3 Lenguaje de programación Python

Python es un lenguaje de programación interpretado cuya principal característica es que utiliza una sintaxis que favorece el código legible.

Se trata de un lenguaje de programación multiparadigma, ya que soporta orientación a objetos, programación imperativa y, en menor medida, programación funcional. Es un lenguaje interpretado, usa tipado dinámico y es multiplataforma.

Gracias a sus características, su uso es totalmente flexible y permite un tiempo de desarrollo menor principalmente por su tipado dinámico y su sintaxis. Sin embargo, al tratarse de un lenguaje interpretado, el tiempo de ejecución es más alto lo cual no lo hace adecuado para tareas que requieran altos niveles de eficiencia.

Dentro del desarrollo en Python para ROS (rospy [ROSc]), existe una interfaz completa para comunicarse con los nodos y otras funcionalidades de ROS desarrolladas en Python o C++.

3.3.4 Controlador de versiones git y repositorios GitHub

Git es un software de control de versiones creado por Linus Torvalds. Git gestiona los archivos y directorios y los cambios hechos en ellos a lo largo del tiempo. Esto permite recuperar antiguas revisiones del proyecto o ver el historial de cambios.

Git fue creado pensando en la eficiencia y la confiabilidad del mantenimiento e versiones cuando estas tienen un gran número de archivos de código fuente. Tiene la capacidad de poder trabajar varias personas con el mismo paquete siempre que no modifiquen el mismo archivo. Además en ese caso, sería posible ver las diferencias entre ambas versiones, y unirlas o crear una rama del proyecto principal si fuera necesario tener las dos versiones.

GitHub² es un sistema de almacenamiento público de código fuente (de cualquier tipo) o un servicio de repositorios. Su principal característica es la de ofrecer una plataforma de interacción social [DSTH12] en la que distintas personas pueden trabajar conjuntamente. Esto permite que varios desarrolladores contribuyan a un proyecto y trabajen de manera coordinada.

²www.github.com

Tanto para el desarrollo software de este proyecto como para la redacción de esta memoria se han utilizado estas herramientas, y el acceso a los repositorios se encuentran en las siguientes direcciones:

- Desarrollo software del proyecto https://github.com/danimtb/pioneer3at_ETSIDI
- Memoria del proyecto https://github.com/danimtb/TFG_pioneer3at

3.3.5 Simulador de robótica Gazebo

Gazebo [Ope] es un simulador de robótica en tres dimensiones que ofrece la simulación de complejos entornos de diversas características, así como robots de todo tipo, su interacción con el entorno y la representación visual de datos obtenidos por diversos sensores como cámaras, láseres, ultrasonidos...

Un buen simulador de robótica es esencial para cualquier tipo de desarrollo robótico, ya que podemos realizar las pruebas software o la viabilidad de un sistema antes de construirlo. Gazebo cuenta con un potente motor de física simulada, interacción con objetos y dinámica de los mismos [KH04].

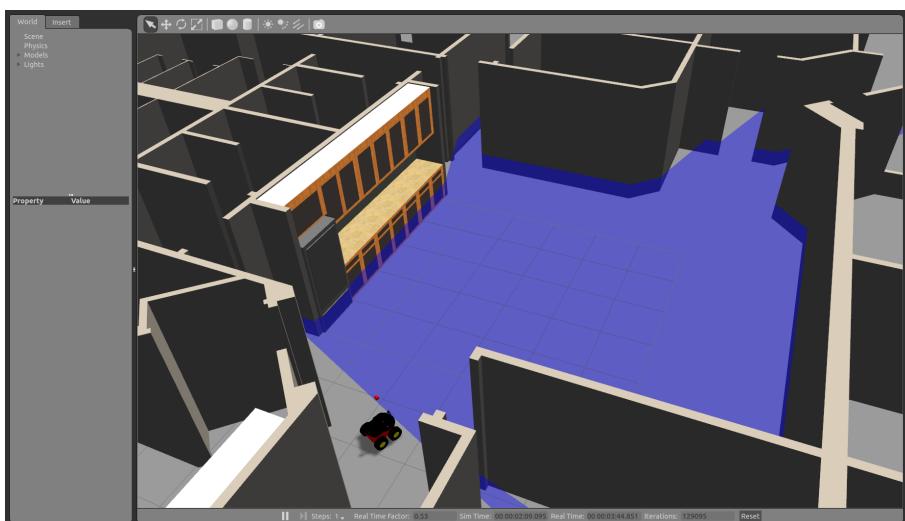


Figura 3.3: Entorno de simulación Gazebo.

Gazebo permite una integración completa con ROS, gestiona modelos físicos de robots utilizando el formato URDF (Unified Robot Description Format) [Gar11] y añade características específicas como el tipo de material, los momentos de inercia o el modelo de colisión. Además, incorpora plug-ins (funcionalidades añadidas) que permiten la simulación de robots de tipo diferencial, simulación de sensores y el cálculo de transformadas entre los distintos sistemas de referencia.

Gazebo es mantenido y desarrollado actualmente por la Open Source Robotics Foundation [OSF].

3.3.6 RViz: Herramienta de visualización robótica

RViz es una herramienta para la visualización de datos en 3 dimensiones de forma gráfica que trabaja dentro del entorno ROS (Figura 3.4). Esta aplicación nos permite ver lo que está ocurriendo en nuestra plataforma robótica a tiempo real.

RViz puede usarse para mostrar lecturas de sensores, datos devueltos por sensores de percepción en 3 dimensiones (nubes de puntos), visualizar mapas, visualizar un modelo de nuestro robot, su posición, etc. También puede utilizarse para interactuar con nuestro robot, utilizando marcadores interactivos o su interfaz de usuario.

RViz no es un simulador de robótica, si no una herramienta que nos muestra la información que maneja nuestro sistema robótico ROS de manera visual.

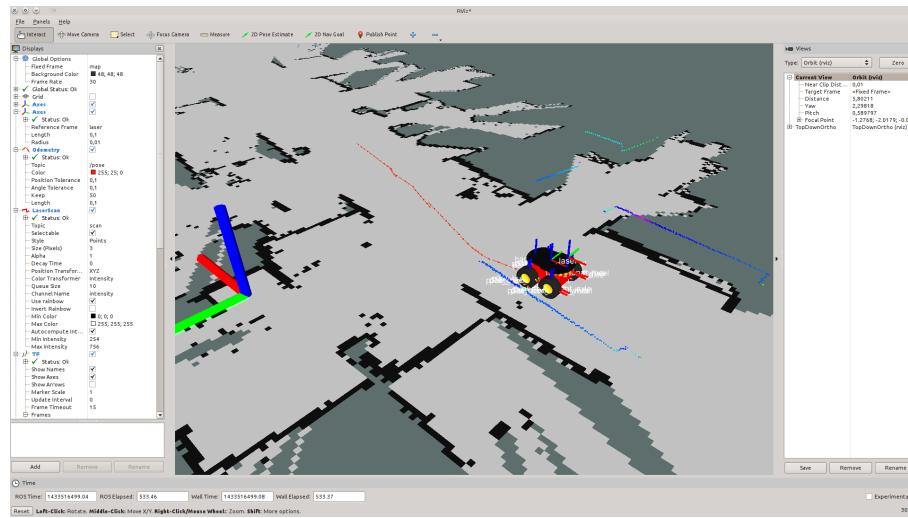


Figura 3.4: Entorno gráfico RViz.

Al uso, RViz es un nodo más dentro de ROS que se suscribe o publica mensajes a otros nodos.

Su uso está muy extendido en ROS ya que nos permite entender lo que ocurre alrededor del robot y la información que manejan los nodos dentro del sistema de una manera intuitiva.

3.3.7 Impresión 3D

La impresión 3D es un grupo de tecnologías de fabricación por adición donde un objeto tridimensional es creado mediante la superposición de capas sucesivas de materia.

Las impresoras 3D de uso popular utilizan un extrusor preparado para plásticos procesados en forma de filamento de pocos milímetros de grosor que deposita material capa a capa hasta que el objeto deseado alcanza su forma final.

EL diseño de los objetos se realiza mediante una herramienta de diseño asistido por ordenador y los objetos deseados se exportan en formato de mallas.

En este proyecto la impresión 3D ha servido de ayuda para crear piezas específicas para colocar los sensores del robot, anclar elementos de la estructura del robot o servir de soporte a equipos provisionales.

El software utilizado para el modelado 3D ha sido la herramienta FreeCAD y la impresora 3D Makerbot thing-o-matic con su software ReplicatorG para generar las capas y controlar la impresora.

3.4 Hardware

En esta parte se explica detalladamente el hardware empleado en el desarrollo del proyecto.

3.4.1 Pioneer 3 AT

El robot Pioneer 3 AT (Figura 3.5), perteneciente a la empresa Adept MobileRobots, es un robot de cuatro ruedas en configuración skid-steer y todo terreno (AT, All Terrain) de operación e investigación en laboratorio.



Figura 3.5: Robot Pioneer 3-AT.

Su configuración en skid-steer permite un control relativamente simple utilizando el modo diferencial para poder realizar giros con gran maniobrabilidad, sin embargo, esta configuración depende mucho del tipo de suelo, con lo que se pierde precisión.

Este robot dispone de baterías, interruptor con parada de emergencia, dos motores de corriente continua para cada par de ruedas con transmisión mediante correa, encoders para leer la odometría y un microcontrolador con firmware ARCOS.

Ademas cuenta con un pequeño computador interno conectado al microcontrolador que puede utilizarse para realizar operaciones de manera autónoma.

El cuerpo del robot es de aluminio y su parte delantera así como superior es fácilmente desmontable para realizar las conexiones pertinentes y acceder al ordenador de a bordo y la placa microcontroladora. En la plataforma superior se sitúa el panel de control (Figura 3.6)para acceder al ordenador de abordo conectando un monitor, teclado y ratón, puerto serial RS-232, botones de encendido y reset varios leds indicadores de estado y de envío y recepción de datos.

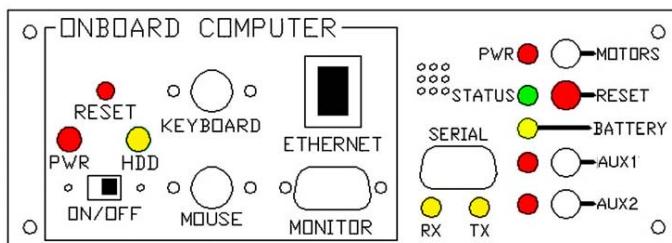


Figura 3.6: Panel de control del robot Pioneer 3-AT.

En la siguiente tabla (Tabla 3.1) se describen las principales características del robot.

Especificaciones	Pioneer 3 AT
Largo	508 mm
Ancho	497 mm
Alto	277 mm
Distancia al suelo	80 mm
Peso	12 kg
Carga útil	32 kg
Cuerpo	Aluminio de 1.6 mm
Baterías	3 de 12 V, estancas, plomo-ácido
Autonomía	4-8 horas
Sistema motriz	4 ruedas motrices
Ruedas	Neumáticos de Nylon
Diámetro de rueda	222 mm (Ruedas todoterreno) / 190 mm (Ruedas interior)
Ancho de rueda	88 mm
Sistema de giro	Diferencial
radio máxima curvatura	40 cm
Radio de giro	0 cm
Máxima velocidad de avance	1.2 m/s
Máximo escalón	10 cm
Máximo hueco	15.2 cm
Terreno	Asfalto, Tierra, Césped, etc.
Encoders	500 pulsos
Procesador	Hitachi H8S

Tabla 3.1: Especificaciones del robot Pioneer 3 AT.

3.4.2 Sensor Kinect

Kinect es un conjunto de sensores de bajo coste que lo convierte en una herramienta excepcional (Figura 3.7). Este dispositivo incluye una cámara de vídeo RGB, una cámara infrarroja de profundidad, un array de micrófonos y altavoces, un acelerómetro y un pequeño motor que le permite hacer movimientos de inclinación.



Figura 3.7: Sensor Kinect.

Su función principal es la de percibir el entorno captando una serie de puntos que se ubican en las tres dimensiones. Su funcionamiento a grandes rasgos se basa

en un emisor de infrarrojos a 830 nm que interactúa con los objetos y una cámara infrarroja que detecta la diferencia entre la proyección anterior y la actual, obteniendo la distancia a cada objeto.

En primer lugar, el laser infrarrojo es emitido por Kinect con un patrón determinado (Projected textures), el cual no es simétrico sino que tiene puntos aleatorios que se dispersa gracias a unas lentes de proyección. Estos puntos aleatorios se reflejan en los objetos, los cuales sería posible verlos con una cámara externa.

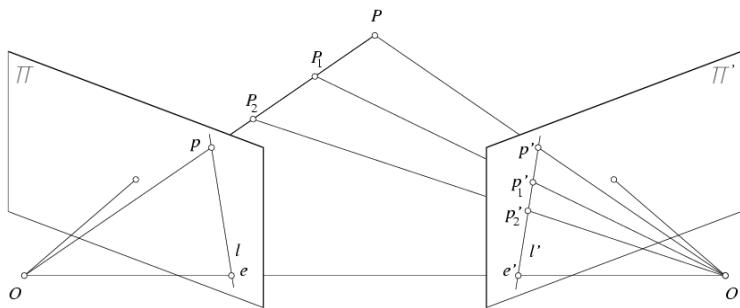


Figura 3.8: Proyección de infrarrojos y obtención de la nube de puntos.

A continuación, al sensor de Kinect MT9M001C12STM, que no es más que el sensor CMOS de una cámara en la que se le trata para que observe solo el infrarrojo, obteniendo los puntos infrarrojos en el plano 2D. El motivo por el que podemos medir la profundidad de los objetos (su distancia) es porque sabemos el patrón de cómo emite el laser emisor [Kon10], por tanto sabremos que si un punto no está en el sitio que corresponde, se ha trasladado respecto al punto inicial y se le aplica la correspondiente transformación (Figura 3.8), obteniendo finalmente los puntos de toda la nube en coordenadas cartesianas XYZ.

La siguiente tabla (Tabla 3.2) muestra las especificaciones del sensor Kinect.

Especificaciones	Sensor Kinect
Dimensiones del conjunto	270mm x 50mm x 70mm
Fuente infrarroja	830nm
Potencia	60 mW
Cámara Infrarroja	MT9M001C12STM
Resolución cámara infrarroja	1200x960 pixeles
Frecuencia	30 Hz
Tamaño pixel	5.2um x 5.2um
Pixeles activos	1280H x 1024V
Campo de visión	58° H, 45° V, 70° D
Resolución espacial	3mm (a 2 metros de distancia)
Resolución de profundidad	1cm (a 2 metros de distancia)
Distancia de operación	0.45m ? 6.5m
Cámara RGB	MT9M112
Resolución cámara RGB	640 x 480
Audio	TAS1020B (Controlador de Audio)
Formato	16kHz, 16-bit mono, modulación por codificación de pulso (PCM)
Entrada de audio	4 micrófonos con conversión analógico digital de 24bits
Acelerómetro	KXSD9-2050

Tabla 3.2: Características del sensor Kinect.

3.4.3 Láser SICK LMS100

El sensor Sick LMS100 es un sensor láser por infrarrojos de clase I (Inofensivo para el ojo humano), que obtiene la medida de distancias con gran precisión y rapidez en un solo plano y realizando un barrido de 270° (Figura 3.9).

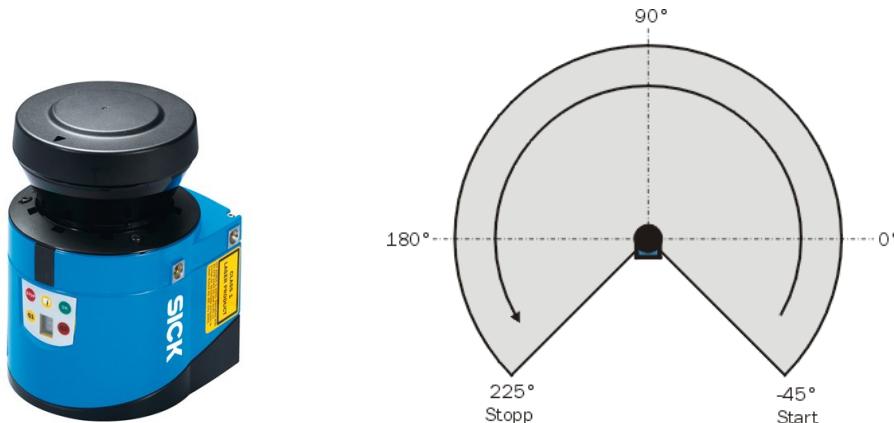


Figura 3.9: Sensor escaner láser Sick LMS100.

Tras un análisis realizado durante este proyecto este sensor se ha situado en la parte trasera del robot, enfocando hacia atrás para cubrir un mayor rango y conocer todo el entorno alrededor del robot.

Destaca por su amplio rango, alcance y precisión. En la siguiente tabla (Tabla 3.3) se recogen sus características principales.

Especificaciones	Sick LMS100
Campo de aplicación	Interiores
Fuente infrarroja	905 nm
Clase Láser	1 (IEC 60825-1)
Campo de visión	270°
Frecuencia de escaneo	25Hz/50Hz
Resolución angular	0.25°/0.5°
Distancia de operación	0.05 - 20 m
Tiempo de respuesta	20 ms
Error	30 mm
Interfaz de datos	Ethernet
Tensión de operación	10.8V - 20V DC
Consumo	20 W
Peso	1.1 Kg
Dimensiones	105mm x 102mm x 152mm

Tabla 3.3: Características del sensor láser Sick LMS100. Basado en [SIC09].

3.4.4 Intel NUC NUC5i7RYH

El ordenador Intel NUC NUC5i7RYH, es un ordenador compacto de altas prestaciones y de tamaño compacto que ofrece unas buenas características para procesar datos y realizar la algoritmia adecuada para tareas de robótica.

Está equipado con un procesador Intel i7-5557U de quinta generación que ofrece una frecuencia de reloj de 3.1 GHz. Dispone de un disco duro de estado sólido que permite una alta velocidad de lectura y escritura en disco, así como una tarjeta RAM de tipo DDR3L de 8GB que permitirá el intercambio de información entre los nodos ROS de una manera fluida.



Figura 3.10: ordenador compacto Intel NUC.

Su cometido será el de procesar la información de los sensores, generar los mapas incorporando los obstáculos, generar las trayectorias de navegación, comandar los motores del robot para realizar movimientos y realizar las funciones de servidor de datos.

Dispone de tamaño compacto y un consumo bajo, junto con una alimentación a partir de los 12 voltios, lo que lo hace ideal para incorporarlo en robots móviles que requieran realizar tareas con alto procesamiento sin depender de una infraestructura.

En la tabla 3.4 pueden consultarse sus características principales.

Especificaciones	Intel NUC NUC5i7RYH
Procesador	Intel Core i7-5557U, dual-core
Frecuencia de reloj	3.1 GHz hasta 3.4 GHz
Memoria RAM	DDR3L1 8 GB
Disco duro	M.2 SSD 120 GB
Gráficos	Iris Graphics 6100
Conectividad de periféricos	2 x USB 3.0 en el panel posterior 2 x USB 3.0 en el panel frontal 2 x USB 2.0 internos vía colector Intel 10/100/1000 Mbps
Conectividad de red	Intel® Wireless-AC 7265 M.2 Antenas inalámbricas (IEEE 802.11ac)
Alimentación	12-19V DC
Consumo	65 W
Dimensiones	115mm x 111mm x 48.7mm

Tabla 3.4: Características del ordenador Intel NUC NUC5i7RYH.

Capítulo 4

Conceptos previos

En este capítulo se exponen las particularidades de ROS y su funcionamiento así como la estructura del proyecto software desarrollado en este trabajo.

4.1 Entorno ROS

La versión del software ROS que se ha utilizado para el desarrollo del proyecto ha tratado de ser siempre la más actual posible, ya que eso nos asegura mantener la compatibilidad en futuros trabajos y que el software esté actualizado.

El sistema se ha desarrollado bajo la versión ROS Indigo Igloo ,en el sistema operativo Ubuntu 14.04. A fecha de entrega del proyecto existe una versión más actualizada de ROS, sin embargo se desestimó su uso debido a que aún no era una versión estable y algunos paquetes no se encontraban disponibles para dicha versión.

ROS realiza funciones similares a un sistema operativo, como la comunicación e interacción entre diferentes procesos, la distribución en hilos, comunicación distribuida entre máquinas, abstracción del hardware, control a bajo nivel, etc.

ROS también proporciona herramientas y librerías para crear código, compilarlo y ejecutarlo en diferentes máquinas.

El concepto más importante dentro de ROS es el nodo, que no son más que un proceso que se ejecuta y conecta al proceso principal, llamado MASTER, para comunicarse con otros nodos. Existen diferentes modos de interacción entre ellos: rostopics, rosservices, nodelets, etc. Los más usuales se explican con más detalle a continuación..

4.1.1 Funcionamiento de ROS

Como hemos dicho, en ROS las acciones del robot se llevan a cabo mediante la interacción de diferentes nodos que se conectan entre sí en forma de grafo. Cada nodo se conecta a un nodo principal llamado MASTER (roscore) que se encarga de abrir un canal de comunicación entre cada nodo o proceso.

En la figura 4.1 vemos un ejemplo para el robot simulado *turtlesim*, al cual le envía comandos el nodo de teleoperación *teleop_turtle* y ambos publican información a través del nodo *rosout*.

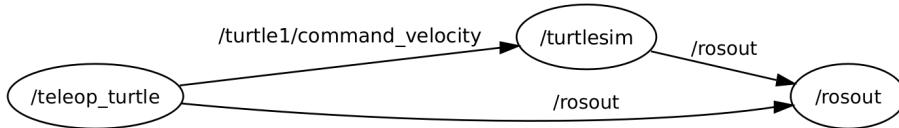


Figura 4.1: Grafo de ejemplo con nodos conectados

Teniendo esta estructura en mente, es conveniente definir algunos conceptos sobre los que trabaja ROS:

- Nodos (nodes): Los nodos son procesos que realizan algún tipo de cómputo. Un robot basado en ROS precisará de varios nodos ejecutándose al mismo tiempo e intercambiándose información a través de Topics, Servicios y Parámetros.
- Máster (master): El nodo ROS master proporciona una vía comunicación para el intercambio de mensajes y un registro de cada nodo. También dispone de un registro de parámetros a los que pueden tener acceso cualquiera de los nodos.
- Mensajes (messages): Los nodos se comunican unos con otros a través de mensajes. La estructura de estos mensajes puede definirse por el usuario y contener diferentes tipos de datos.
- Topics: Los mensajes son transmitidos a través de un sistema de publicación / suscripción. Un nodo envía un mensaje publicándolo en cierto topic y otro nodo puede leer el mensaje si se suscribe.
- Servicios (services): Los servicios son similares a los topics pero son mucho más apropiados para realizar comunicaciones del tipo solicitud / respuesta.
- Bags: El concepto de "bag" puede entenderse como un almacén de mensajes. Los mensajes de cierto topic pueden guardarse para analizar los datos más tarde o reproducir cierta situación.

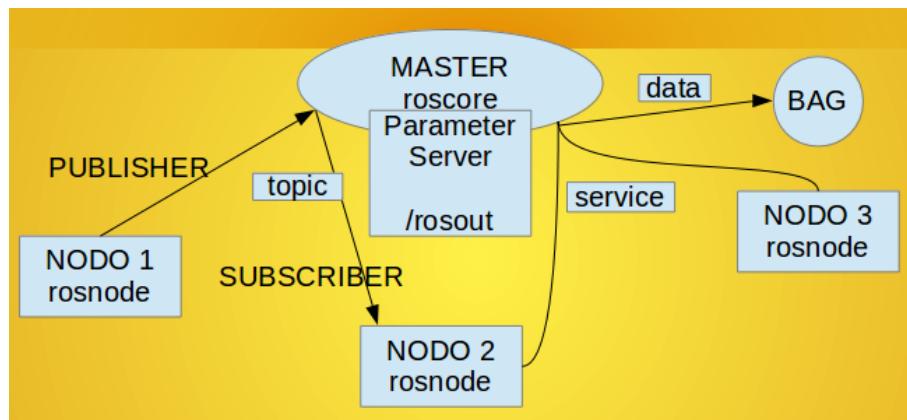


Figura 4.2: Esquema del funcionamiento de ROS

A nivel de sistema de ficheros ROS se estructura de la siguiente forma:

- Paquetes (packages): Es la unidad principal de organización de software en ROS. Un paquete puede contener nodos, tipos de datos, archivos de configuración.
- Metapaquetes (metapackages): Sirven para representar un conjunto de paquetes que tienen relación entre sí.
- Package manifests: Son archivos de tipo xml donde se indican los metadatos de un paquete, como su nombre, su versión, una descripción, licencia, dependencias de otras librerías o paquetes...
- Repositorios: Normalmente, para distribuir los paquetes en ROS se utilizan repositorios bajo un sistema de control de versiones.
- Tipos de mensajes: Son archivos con la estructura de un tipo de mensaje (MiMensaje.msg)
- Tipos de servicios: Son archivos con la estructura de un tipo de servicio tanto para los datos de solicitud como los de respuesta (MiServicio.srv)

También es necesario conocer algunos tipos de archivos y su uso dentro de ROS para comprender el trabajo de este proyecto:

- Launchfiles: ROS ofrece la herramienta *"roslaunch"* para ejecutar varios nodos con un solo comando y configurar sus parámetros. Es una forma cómoda de organizar la puesta en marcha de todos los nodos y enlazar unos *launchfiles* con otros. Los archivos suelen tener la extensión *.launch* y su estructura es similar a la sintaxis xml.
- yaml: Es un formato de serialización de datos legible para los humanos de tal modo que se enfoca más en los datos que en el marcado de los archivos. Su formato es *.yaml* y se utiliza para definir parámetros en cada nodo.
- xacro: Son archivos que combinan el lenguaje xml con macros de tal modo que podemos mantener los archivos más legibles y ordenados. Estos archivos se utilizan principalmente para definir un modelo de nuestro robot en URDF (Unified Robot Description Format) y su extensión es *.xacro*.
- URDF (Unified Robot Description Format): Es un tipo de formato utilizado para describir la estructura de un objeto en forma de eslabones y articulaciones de distinto tipo [Gar11]. En ROS se utilizan para generar modelos del robot en Gazebo y en RViz y conocer de manera gráfica el estado de los eslabones del robot. La extensión de estos archivos es *.urdf*.
- gazebo: Los archivos *.gazebo* definen características concretas para utilizar robots definidos mediante URDF. Estos archivos permiten desacoplar las características utilizadas en Gazebo del modelo original.
- world: Los archivos *.world* definen en sintaxis xml mundos virtuales para cargar en el simulador Gazebo. Pueden definirse los objetos a incorporar, las luces y la posición del punto de vista.

Por último, cabe destacar la importancia de uno de los paquetes integrados en ROS que más beneficios nos aporta, el paquete de transformadas *tf*.

tf [FMEM11] es uno de los paquetes de geometría dentro de ROS que nos permite realizar transformadas de los datos de un sistema de coordenadas concreto a otro. estos sistemas de coordenadas son denominados como *frames* y realizan la transformación de los datos de manera periódica. Muchas funcionalidades de ROS se basan en este tipo de transformadas para realizar los cálculos matemáticos oportunos así como para determinar la posición de nuestro robot y sus articulaciones.

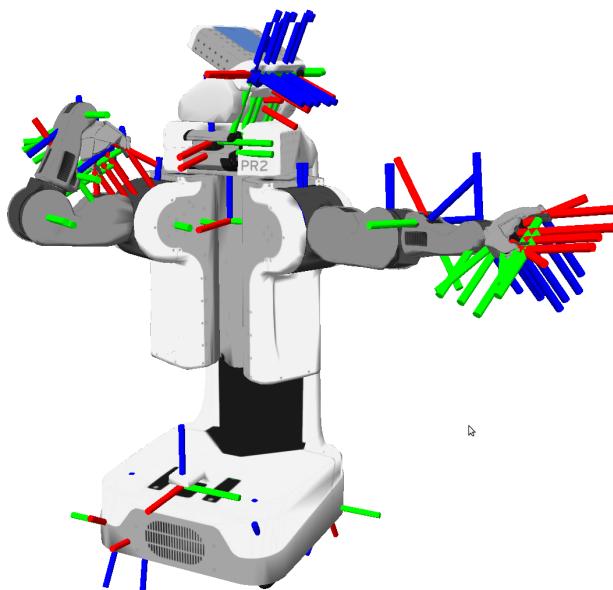


Figura 4.3: Frames utilizados en el robot PR2.

4.1.2 Configuración de ROS

Para la instalación de ROS es necesario seguir ciertos pasos bien explicados en la wiki de su página¹. Al ser un sistema fuertemente basado en sistemas UNIX y con mayor soporte para Ubuntu, las instalaciones se realizan principalmente a través de herramientas como apt (Advanced Packaging Tool).

Existen diferentes instalaciones de ROS dependiendo de si precisamos de todas sus herramientas y utilidades o no, ya que por ejemplo no tendría sentido instalar herramientas de interfaz gráfica en un robot que dispone de entorno gráfico. Para el ordenador de abordo del robot utilizamos la versión completa del software.

```
1 $ sudo apt-get install ros-indigo-desktop-full
```

Código 4.1: Mandato de consola para instalar la versión completa de ROS Indigo.

Añadimos nuestra variable de entorno para que el sistema reconozca las herramientas de ROS:

¹<http://wiki.ros.org/indigo/Installation/Ubuntu>

```
1 $ source /opt/ros/indigo/setup.bash
```

Código 4.2: Source al setup de ROS Indigo

Para el desarrollo dentro de ROS se utiliza el entorno de desarrollo Catkin **referencia**, que facilita el enlazado y compilación de los paquetes. para ello es necesario tenerlo instalado y crear un entorno de desarrollo:

```
1 $ sudo apt-get install ros-indigo-catkin
2 $ mkdir -p ~/catkin_ws/src
3 $ cd ~/catkin_ws/src
4 $ catkin_init_workspace
```

Código 4.3: Instalación y workspace de Catkin

Y a continuación es necesario incluir nuestro directorio de desarrollo para que sea reconocido:

```
1 $ cd ~/catkin_ws
2 $ source devel/setup.bash
```

Código 4.4: Source al setup de nuestro entorno Catkin

A partir de este punto podríamos crear nuestros propios paquetes utilizando las funcionalidades de ROS ².

4.1.3 Configuración de los paquetes ROS

Los paquetes ROS son funcionalidades desarrolladas por terceros que se integran en el sistema ROS y que son transferibles de un robot a otro.

Los paquetes ROS incorporan un archivo *CMakeLists.txt* para la compilación de los nodos que se hayan desarrollado así como sus mensajes y un archivo Package Manifest (*package.xml*), como hemos indicado anteriormente.

Para el desarrollo de este proyecto es necesario clonar el repositorio GitHub *pioneer3at_ETSIDI* en nuestra carpeta *~/catkin_ws/src*. Este repositorio contiene todos los paquetes necesarios en nuestro entorno:

```
1 $ cd ~/catkin_ws/src
2 $ git clone --recursive https://github.com/danimtb/pioneer3at_ETSIDI.git .
```

Fuente: https://github.com/danimtb/pioneer3at_ETSIDI

Código 4.5: Clonado del repositorio *pioneer3at_ETSIDI*.

Junto con el repositorio de desarrollo también es preciso instalar funcionalidades adicionales de ROS. Toda esta configuración puede consultarse con más detalle en el apéndice A.

²Crear un paquete Catkin <http://wiki.ros.org/catkin/Tutorials/CreatingPackage>

4.2 Arquitectura

Esta sección tiene como objetivo plantear la arquitectura general utilizada en el robot, las comunicaciones con el resto del hardware y con los nodos que proporcionan la información necesaria para que el robot sea totalmente autónomo.

4.2.1 Arquitectura del sistema

A nivel de hardware utilizado en el proyecto, como es el propio robot, los sensores el ordenador de abordo el sistema se estructura de a siguiente manera:

- i) Robot Pioneer 3 AT: Este es el robot mencionado anteriormente, el cual debe ser configurado para acceder al puerto serie RS-232 de su placa controladora. Esto nos permite conectarnos con el firmware ARCOS y comunicarnos a través de la librería ARIA. De esta forma se controlan los motores y se obtiene el valor de los encoders de la odometría.
- ii) Sensores: Tanto el sensor Kinect como el sensor láser sea alimentan mediante las baterías del robot y se comunican con el ordenador de abordo a través del puerto USB y ethernet respectivamente.
- iii) Ordenador Intel NUC: Es el ordenador de abordo encargado de ejecutar ROS y realizar todo el procesamiento necesario. Está equipado con el sistema operativo Ubuntu 14.04 por ser la última versión estable disponible a fecha de la entrega del proyecto. Se conecta al microcontrolador del robot mediante un convertidor RS-232 a USB, al sensor láser y al sensor Kinect. Además el sistema contiene unos altavoces externos que se conectan por su salida específica.
- iv) Ordenador externo: Como se ha mencionado anteriormente, un ordenador externo opcional equipado con ROS podrá utilizarse para realizar tareas de supervisión a través de RViz y para realizar la teleoperación del robot vía TCP/IP en ROS.

4.2.2 Estructura software del proyecto

El proyecto está estructurado siguiendo la filosofía de "paquetes" desarrollados en ROS. Los paquetes se compilan dentro de un entorno de trabajo tipo Catkin [SKGT12] que se encarga de compilar correctamente a través de CMake todos los ejecutables y de realizar el enlazado correctamente.

En el directorio raíz del proyecto por tanto, encontraremos los paquetes necesarios para que el sistema funcione:

Las carpetas de color azul son paquetes desarrollados por terceros que no vienen integrados por defecto en ROS, mientras que los paquetes ROS de desarrollo propio en este proyecto son los indicados en color naranja. El procedimiento para utilizarlos es clonar su repositorio en GitHub e incluirlos como submodulos dentro de nuestro proyecto.

A continuación se realiza una breve descripción de cada uno:

- *audio_common*: Agrupa todas las funcionalidades para reproducir sonidos y voz sintetizada.

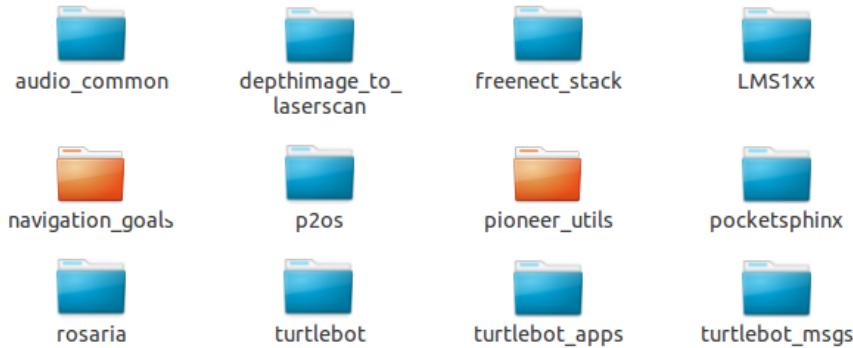


Figura 4.4: Estructura del proyecto

- *depthimage_to_laserscan*: Nodo que realiza la conversión del tipo de dato Pointcloud2 a LaserScan mediante el análisis de imagen y la profundidad de la nube de puntos. Configurable mediante parámetros.
- *freenect_stack*: Agrupa los nodos controladores del sensor Kinect basados en libfreenect y transfiere la información para que sea accesible mediante ROS.
- *LMS1xx*: Nodo para la conexión con los sensores Láser Sick De la familia LMS100 a través de puerto ethernet.
- *p2os*: Agrupa utilidades y nodos para conectarse con los robots de la familia Pioneer, en especial Pioneer 3 AT y 3 DX. Ofrece modelos 3D de cada robot y algunos parámetros de configuración de los robots.
- *pocketsphinx*: Utilidad para el reconocimiento de voz mediante cualquier tipo de micrófono.
- *rosaria*: Interfaz de comunicación ROS con la librería Aria para el control de los motores del robot y la lectura de los encoders. Ofrece parámetros de calibración de los encoders y acceso al array de ultrasonidos del robot (funcionalidad no incorporada en el robot utilizado para este proyecto).
- *turtlebot*, *turtlebot_apps* y *turtlebot_msgs*: Paquetes que agrupan funcionalidades para el robot *Turtlebot*, usadas en este caso en nuestro desarrollo.

Los paquetes ROS de desarrollo propio son los siguientes:

- El paquete *navigation_goals* es un nodo que realiza navegación a través de puntos establecidos de modo que se puedan programar rutas a seguir por el robot.
- El paquete *pioneer_utils* es donde se encuentra el desarrollo principal de este proyecto y se describirá en detalle a continuación.

Como cualquier paquete en ROS, disponemos de un archivo *CMakeLists.txt* y un *package.xml*, donde se indican los objetivos a compilar y las librerías adicionales para el enlazado así como las dependencias respectivamente.

En el directorio del paquete *pioneer_utils* encontramos:

- *follower*: Archivos de configuración para utilizar la funcionalidad follower del robot Turtlebot en el Pioneer 3 AT.

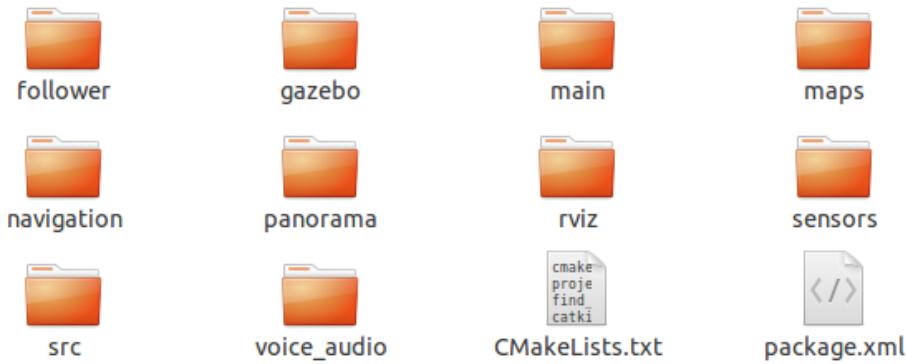


Figura 4.5: Estructura de carpetas del paquete `pioneer_utils`

- *gazebo*: Especificaciones necesarias para ejecutar Gazebo. Incluye un modelo en tres dimensiones del robot Pioneer 3 AT y los sensores del mismo tipo que el utilizado en la realidad. También incluye *launchfiles* para realizar la navegación con diferentes mapas.
- *main*: Launchfiles de las aplicaciones principales que incorpora el robot como el nodo *endurance_test*.
- *maps*: Información sobre mapas guardados realizados con la funcionalidad de *slam_gmapping*.
- *navigation*: Archivos y ajustes necesarios para realizar navegación del robot con y sin mapa (global navigation o local navigation).
- *panorama*: Archivos de configuración para utilizar la funcionalidad *panorama* del robot *Turtlebot* en el nuestro.
- *rviz*: Configuraciones preguardadas para rviz.
- *sensors*: Archivos de configuración para tener acceso a toda la información del hardware y de los sensores que se utilizan.
- *src*: Carpeta donde se incluye el código fuente en C++ o Python de los nodos de teleoperación, test de navegación y navegación estimada (Dead reckoning).
- *voice_audio*: Archivos de configuración para el nodo de reconocimiento de voz y sonido, diccionarios de palabras y pronunciación.

Capítulo 5

Navegación

En este capítulo se exponen los conceptos referentes a la navegación dentro del ecosistema ROS. En él se describen las características de navegación del paquete específico y sus posibilidades.

Este capítulo trata de abordar el aspecto de la navegación desde el punto de vista de la utilidad, pasando por su fundamento teórico y sin dejar de lado su aplicación en el robot Pioneer 3 AT.

Por tanto, se exponen algunos ejemplos de configuraciones para la navegación, aunque la configuración final del robot y la discusión sobre la misma se abordará en el capítulo 7, donde se hablará de su implementación más en detalle.

5.1 Navigation Stack

Dentro de las funcionalidades de ROS ya hemos comentado que existen los metapquetes o ”Stacks”, que son grupos de paquetes de software que todos juntos ofrece una funcionalidad.

El paquete de navegación de ROS [ME10a] se define como un paquete de navegación en dos dimensiones que toma información de la odometría, de los sensores y de un punto de meta y dirige el robot mediante comandos de velocidad seguros.

La navegación se basa en el uso de ”Mapas de coste”¹ y planificación global y local de trayectorias.

Por un lado, el sistema hace un mapa de coste global que tiene encuentra información de los sensores y la posibilidad de cargar un mapa previo. A este mapa se incorporan los obstáculos que permanecen estáticos durante más tiempo y en base a este se realizan los cálculos de trayectoria global.

Por otro lado, el sistema realiza un mapa de coste local, que analiza los obstáculos más cercanos al robot en cada momento. A este mapa se incorpora la información de los sensores sobre cualquier tipo de obstáculo detectado. En base a este mapa y de forma reactiva se realizan los cálculos de trayectoria local que llevará el control de la navegación reactiva del robot.

¹También denominados mapas de potencial, en esta memoria se hace referencia como mapa de coste o *costmap*

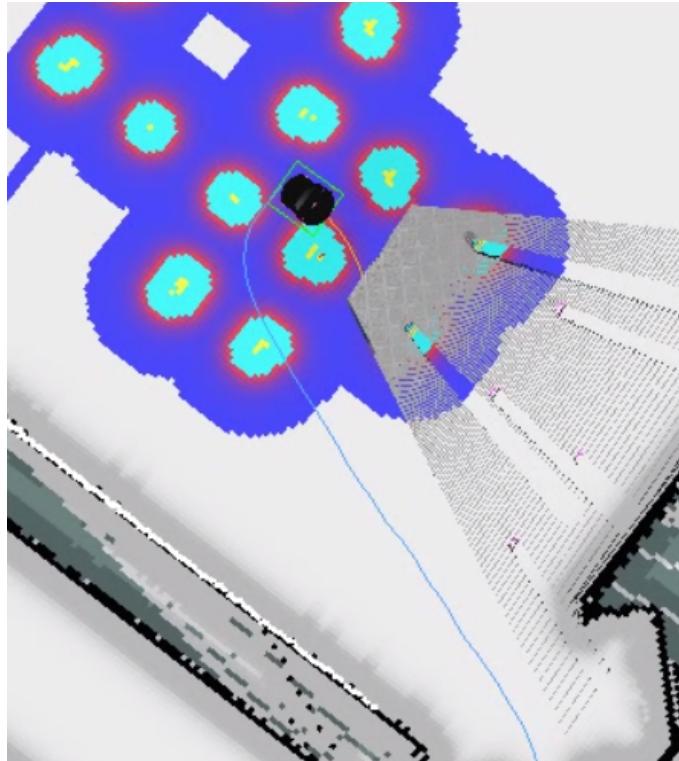


Figura 5.1: Ejemplo de mapas de coste en navegación.

5.1.1 Funcionamiento

El paquete de navegación se basa en diferentes nodos que interactúan para dirigir el robot hasta un punto en el mapa indicado como meta.

Este sistema utiliza un mapa de obstáculos estáticos (*global_costmap*), un mapa de obstáculo locales (*local_costmap*), un nodo de cálculo de trayectoria global (*global_planner*), un nodo de cálculo de trayectoria local (*local_planner*) y mecanismos de recuperación de trayectoria (*recovery_behaviors*).

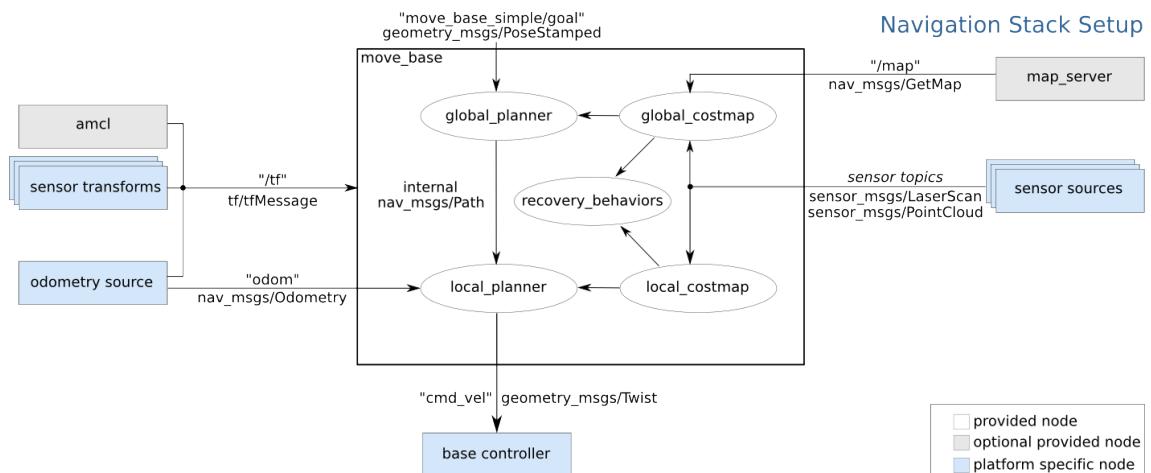


Figura 5.2: Diagrama de funcionamiento del Navigation Stack [ME10b]

En el diagrama mostrado en la figura 5.2 podemos ver una representación de los nodos y mensajes utilizados por el paquete de navegación así como la información

externa y opcional que se precisa.

Como puede apreciarse, los sensores, la odometría y el ajuste de las transformadas entre cada uno de los sistemas de coordenadas son específicos de cada plataforma robótica. La ejecución principal se centra en el nodo *move_base*, que es el encargado de realizar los movimientos acordes a la información que aportan los mapas y los planificadores de trayectoria.

Por último indicar que el paquete de navegación no precisa de un mapa pregrabado para funcionar. Puede utilizarse un mapa si se precisa y cargarlo mediante un nodo externo como *map_server* o generararlo simultáneamente a la navegación mediante *slam_gmappig* (Subsección 5.2.1).

El uso de un nodo de localización del robot en el mapa, como AMCL (Adaptive Monte Carlo Localization), será preciso cuando se utilice un mapa pregrabado para conocer la posición inicial del robot y posicionarlo correctamente dentro del mismo (Sección 5.3).

5.1.2 Requisitos para la navegación

El sistema de navegación de ROS está diseñado para ser flexible, altamente configurable y adaptable a muchos tipos de robots. Sin embargo requiere de ciertos requisitos para poder ser integrado. A continuación se describe cada uno de los requisitos aplicados al robot Pioneer 3 AT utilizado en este trabajo.

Odometría

El sistema de navegación requiere de un robot que publique información sobre la odometría del robot en mensajes de tipo *nav_msgs/Odometry*.

En el robot Pioneer 3 AT y gracias al paquete *RosAria* (Subsección 6.1.1) obtenemos la información de los encoders del robot convertida en datos de odometría a través del topic *pose*.

Movimientos

El sistema de navegación requiere que el robot sea controlable a través de comandos de velocidad de tipo *geometry_msgs/Twist*.

En el robot Pioneer 3 AT a través de *RosAria* (Subsección 6.1.1) podemos mover nuestro robot con comandos de velocidad publicando los al topic *cmd_vel*.

Sensores

Los tipos de sensores que pueden utilizarse en la navegación son variados, el requisito que deben cumplir para poder integrarse en el sistema de navegación es que publiquen información de tipo *sensor_msgs/LaserScan* o *sensor_msgs/PointCloud2*.

Con los nodos descritos anteriormente, podemos obtener información de este tipo para el sensor Kinect (Subsección 6.1.2) y para el sensor Láser (Subsección 6.1.3). El nodo *freenect_stack* publica información de la nube de puntos a través del topic *camera/depth/points* y el nodo *LMS1xx* lo hace a través del topic *scan*.

Transformadas

Es necesario que toda la información esté estructurada geométricamente para que el sistema de navegación pueda realizar los cálculos pertinentes. Se requiere información de los sistemas de coordenadas de cada sensor, de la base del robot y de la odometría.

Cada uno de los nodos descritos con anterioridad publican diferentes *frames* que sirven para este propósito. Estos *frames* (*odom*, *base_link*, *laser*, *camera_link*) relacionados mediante transformadas estáticas como se describió en la subsección 6.1.4,

proporcionan la información necesaria.

5.1.3 Configuración de la navegación

Aunque se explicará más adelante, una vez presentados los requisitos para utilizar el paquete de navegación es necesario entrar en detalle sobre la configuración necesaria para hacer que el sistema funcione.

En primer lugar, para facilitar la comprensión, suponemos que disponemos de un mapa del entorno en el que se va a desplazar nuestro robot y que disponemos de tan solo un sensor láser para realizar la navegación.

Comenzamos por la configuración de los llamados *Costmaps*, esto es, mapas que almacenan los obstáculos del entorno del robot. Disponemos de *global_costmap* y *local_costmap* como dijimos antes y parte de la configuración de estos será compartida, por tanto debemos crear un archivo de parámetros comunes *costmap_common_params.yaml*.

```

1  obstacle_range: 6.0
2  raytrace_range: 6.5
3  footprint: [ [0.254, -0.230], [-0.254, -0.230], [-0.254, 0.230], [0.254, 0.230] ]
4  inflation_radius: 0.5
5
6  observation_sources: laser_scan_sensor
7
8  laser_scan_sensor: {sensor_frame: laser, data_type: LaserScan, topic: /scan, marking: true,
    clearing: true}

```

Código 5.1: Ejemplo de *costmap_common_params.yaml*

Como vemos, configuramos parámetros como la forma de nuestro robot, el radio de seguridad de los obstáculos (*inflation_radius*), el rango de incorporación de obstáculos y los sensores.

A continuación creamos el archivo de configuración de nuestro mapa global *global_costmap.yaml*.

```

1  global_costmap:
2    global_frame: /map
3    robot_base_frame: base_link
4    update_frequency: 5.0
5    static_map: true

```

Código 5.2: Ejemplo de *global_costmap.yaml*

En este archivo vemos como este mapa de coste permanece estático (*static_map: true*) y anclado al eje de coordenadas del mapa.

El mapa de navegación local o reactiva es similar, aunque en este caso el mapa tendrá unas dimensiones preestablecidas y se moverá con el robot.

```

1 local_costmap:
2   global_frame: odom
3   robot_base_frame: base_link
4   update_frequency: 5.0
5   publish_frequency: 2.0
6   static_map: false
7   rolling_window: true
8   width: 6.0
9   height: 6.0
10  resolution: 0.05

```

Código 5.3: Ejemplo de *local_costmap.yaml*

La configuración básica del planificador local se guardará en un archivo *base_local_planner_params.yaml*.

```

1 TrajectoryPlannerROS:
2   max_vel_x: 0.45
3   min_vel_x: 0.1
4   max_vel_theta: 1.0
5   min_in_place_vel_theta: 0.4
6
7   acc_lim_theta: 3.2
8   acc_lim_x: 2.5
9   acc_lim_y: 2.5
10
11  holonomic_robot: true

```

Código 5.4: Ejemplo de *local_costmap.yaml*

También debemos indicar la configuración del planificador global en *global_planner_params.yaml*

```

1 GlobalPlanner:
2   old_navfn_behavior: false
3   use_quadratic: true
4   use_dijkstra: flase
5   use_grid_path: false

```

Código 5.5: Ejemplo de *global_planner_params.yaml*

Aquí podemos definir el cálculo de trayectorias siguiendo aproximaciones cuadráticas, basadas en ocupación de celdas, tipo A* o Dijkstra (Se hablará de cada una en la sección 5.5).

Para finalizar, debemos ejecutar el nodo *move_base* con las configuraciones descritas anteriormente, así como el mapa y el nodo de localización AMCL.

La representación visual de ambos costmaps, el área del robot (footprint), el haz láser y el modelo de robot pueden observarse mediante RViz, tal y como muestra la figura 5.3.

Este primer ajuste no tiene en cuenta muchos factores concretos de la navegación en el robot Pioneer 3 AT, como pueda ser la disposición correcta de los sensores, los ajustes de actualización de los mapas, el rango de los sensores o las interferencias de información que se incorporan o borran en los costmaps. Estos detalles se presentarán en la implementación del sistema (Capítulo 7).

```

1 <launch>
2   <!-- Run the map server -->
3   <node name="map_server" pkg="map_server" type="map_server" args="$(find pioneer_utils)/maps
4     /floor_zero-map.yaml"/>
5
6   <!-- Run AMCL -->
7   <include file="$(find pioneer_utils)/navigation/common/amcl.launch"/>
8
9   <node pkg="move_base" type="move_base" respawn="false" name="move_base" output="screen
10    ">
11     <rosparam file="$(find pioneer_utils)/navigation/common/costmap_common_params_p3at.yaml
12       " command="load" ns="global_costmap" />
13     <rosparam file="$(find pioneer_utils)/navigation/common/costmap_common_params_p3at.yaml
14       " command="load" ns="local_costmap" />
15     <rosparam file="$(find pioneer_utils)/navigation/common/local_costmap_params.yaml"
16       command="load" />
17     <rosparam file="$(find pioneer_utils)/navigation/global_navigation/
18       global_costmap_params.yaml" command="load" />
19     <rosparam file="$(find pioneer_utils)/navigation/common/base_local_planner_params.yaml"
20       command="load"/>
21     <rosparam file="$(find pioneer_utils)/navigation/common/global_planner_params.yaml"
22       command="load" />
23   </node>
24 </launch>false

```

Código 5.6: Ejemplo de *robot_navigation.launch*

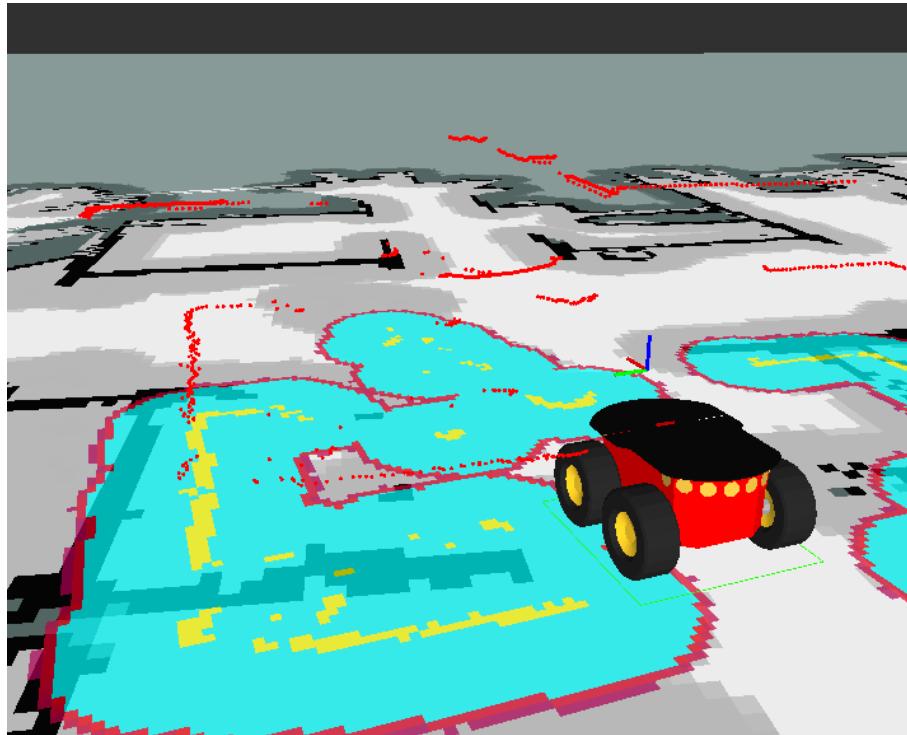


Figura 5.3: Visualización de costmaps, sensor láser y modelo del robot en RViz.

5.2 SLAM

La técnica de SLAM (Simultaneous Localization And Mapping) es una técnica muy utilizada en robótica para construir un mapa de un entorno que es desconocido al mismo tiempo que se estima la posición del robot en el mismo.

Esta técnica presenta dificultades como la incertidumbre de los sensores, la imperfección del sistema de locomoción del robot, la repetitividad de las medidas... Además de que la localización y el mapeo simultáneo de un entorno son dos problemas que están intrínsecamente acoplados [PY12].

Principales algoritmos

Los principales algoritmos para realizar los cálculos probabilísticos en técnicas de SLAM son los siguientes:

- Filtro extendido de Kalman: Es uno de los métodos más extendidos en la técnica del SLAM por ofrecer resultados satisfactorios a pesar de sus problemas de convergencia [RLMJG06]. Esta ha sido la técnica utilizada de facto hasta a aparición del FastSLAM [MTK⁺02].
- Mapas de ocupación de celdillas: Se basa en discretizar el espacio dividiéndolo en unidades de tamaño predefinido que se clasifican como ocupadas o vacías con un determinado nivel de confianza o probabilidad. La precisión (mayor cuanto más fina es la división del espacio), permite que el algoritmo de localización empleado acumule errores reducidos a lo largo de intervalos prolongados de tiempo. Su mayor desventaja de estos métodos es la pérdida de potencia que se deriva de no tener en cuenta la incertidumbre asociada a la posición del robot, lo cual origina que su capacidad para cerrar bucles correctamente se vea mermada [CCR07].

5.2.1 slam_gmapping en ROS

GMapping es una librería perteneciente al proyecto OpenSLAM [SFG07] que utiliza la técnica SLAM Grid Mapping, basada en la generación de mapas mediante la ocupación de celdillas utilizando un filtro de partículas [GSB07].

En ROS se integra bajo el paquete gmapping² que no es más que un *wrapper* de OpenSLAM adaptando su interfaz.

Para crear mapas utilizando *slam_gmapping* se necesita un robot configurado para leer su odometría y un sensor capaz de ofrecernos información de tipo *sensor_msgs/LaserScan*, así como la relación geométrica entre un sistema de referencia y otro.

Existen muchos parámetros para ajustar el proceso de SLAM en este nodo, sin embargo en este caso nos servimos de la configuración utilizada por el paquete *p2os* por ofrecer unos resultados más que aceptables:

²Gmapping <http://wiki.ros.org/gmapping>

```

1 <launch>
2   <node pkg="gmapping" type="slam_gmapping" name="slam_gmapping" args="/scan">
3     <param name="delta" type="double" value="0.05" />
4     <param name="temporalUpdate" type="double" value="2.5" />
5     <param name="xmin" type="double" value="-2" />
6     <param name="xmax" type="double" value="2" />
7     <param name="ymin" type="double" value="-2" />
8     <param name="ymax" type="double" value="2" />
9   </node>
10 </launch>

```

Fuente: *p2os/p2os_launch/launch/gmapping.launch*

Código 5.7: Launchfile *slam_gmaping*.

Para visualizar la creación del mapa en RViz se ha creado un archivo específico:

```
1 $ roslaunch pioneer_utils rviz-gmapping.launch
```

Fuente: *pioneer_utils/rviz/rviz-gmapping.launch*

Código 5.8: Launchfile para visualizar *slam_gmaping* en RViz

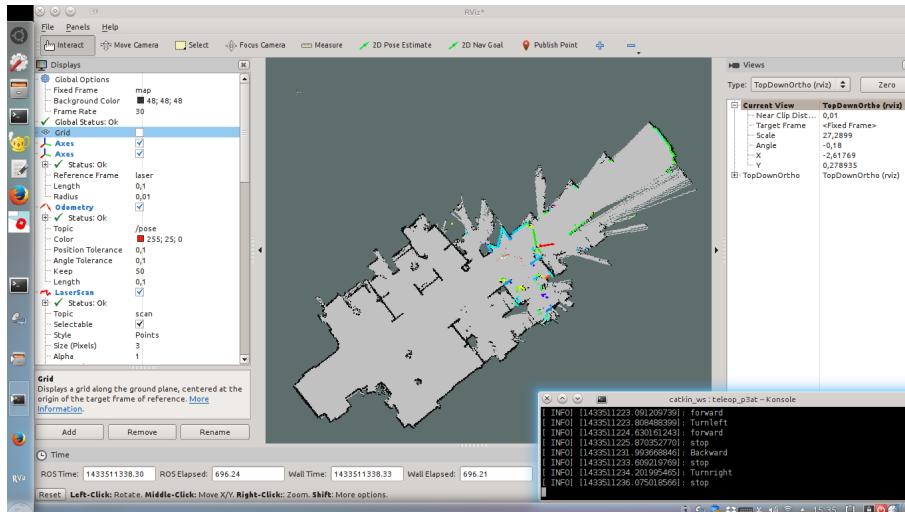


Figura 5.4: *slam_gmapping* visualizado en RViz

Finalmente, para guardar los mapas generados nos servimos de la herramienta *map_server* que nos ofrece guardar mapas que se estén publicando en el topic *map* así como publicar estos mapas para utilizarlos en la navegación.

```
1 $ rosrun map_server map_saver
```

Código 5.9: Ejecución del nodo *map_saver* para guardar el mapa

5.3 AMCL

”Adaptative Monte Carlo Localization”, también conocida por localización mediante filtro de partículas, es un algoritmo utilizado en robótica para determinar la posición

de un robot en un mapa [FBDT99].

El algoritmo emplea un filtro de partículas para representar una distribución de posibles estados dentro del mapa. A medida que los sensores detectan el entorno y el robot se desplaza dentro de este se van otorgando mayor peso a aquellas partículas que se encuentran en una posición más coherente con la información sensorial que se va obteniendo y se van descartando otras.

La idea de esta técnica es conseguir que todas las partículas converjan en una sola (a efectos prácticos, unas pocas partículas) para determinar la posición del robot en el mapa.

Esta técnica se encuentra en ROS bajo un paquete llamado *amcl* [Ger09] y utiliza las técnicas descritas en el libro Probabilistic Robotics [TBF05].

Trasladado a la filosofía de funcionamiento de ROS, esta técnica nos ofrece la posibilidad de situar al robot en un mapa guardado previamente, aportándonos la transformada entre el sistema de referencia (*frame*) del mapa y el sistema de la odometría (Figura 5.5).

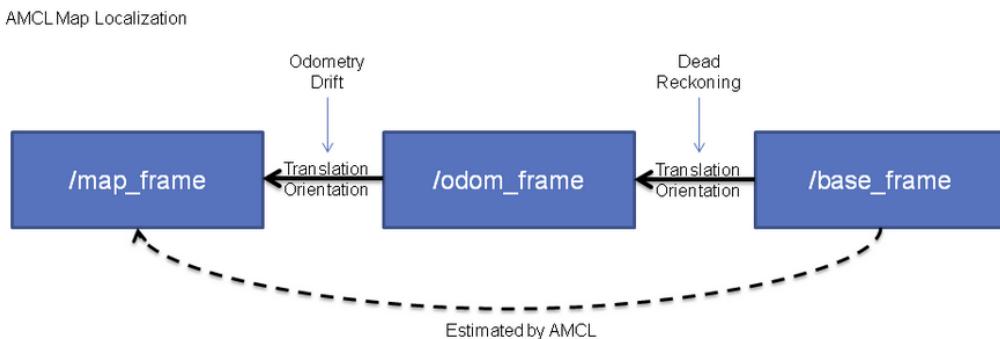


Figura 5.5: Esquema de la labor del nodo AMCL entre los *frames* map y base.

Configuración

Como es habitual en ROS, esta funcionalidad nos aparece empaquetada en forma de nodo configurable a través de parámetros.

Para funcionar requiere el uso de un mapa, el árbol de transformadas de nuestro robot e información de tipo *sensor-msgs/LaserScan*.

La lista de parámetros configurables es amplia ya que debemos encontrar un modelo que se comporte bien con nuestro robot y sensor láser. Son especialmente importantes los parámetros del modelo de la odometría y los referentes al haz láser de nuestro sensor.

```

1 <launch>
2   <node pkg="amcl" type="amcl" name="amcl">
3     <param name="odom_model_type" value="diff"/>
4     <param name="odom_alpha5" value="0.1"/>
5     <param name="transform_tolerance" value="0.2" />
6     <param name="gui_publish_rate" value="10.0"/>
7     <param name="laser_max_beams" value="30"/>
8     <param name="min_particles" value="500"/>
9     <param name="max_particles" value="5000"/>
10    <param name="kld_err" value="0.05"/>
11    <param name="kld_z" value="0.99"/>
12    <param name="odom_alpha1" value="0.2"/>
13    <param name="odom_alpha2" value="0.2"/>
14    <param name="odom_alpha3" value="0.8"/>
15    <param name="odom_alpha4" value="0.2"/>
16    <param name="laser_z_hit" value="0.5"/>
17    <param name="laser_z_short" value="0.05"/>
18    <param name="laser_z_max" value="0.05"/>
19    <param name="laser_z_rand" value="0.5"/>
20    <param name="laser_sigma_hit" value="0.2"/>
21    <param name="laser_lambda_short" value="0.1"/>
22    <param name="laser_lambda_short" value="0.1"/>
23    <param name="laser_model_type" value="likelihood_field"/>
24    <param name="laser_likelihood_max_dist" value="2.0"/>
25    <param name="update_min_d" value="0.2"/>
26    <param name="update_min_a" value="0.5"/>
27    <param name="odom_frame_id" value="odom"/>
28    <param name="resample_interval" value="1"/>
29    <param name="transform_tolerance" value="0.1"/>
30    <param name="recovery_alpha_slow" value="0.0"/>
31    <param name="recovery_alpha_fast" value="0.0"/>
32  </node>
33 </launch>

```

Fuente: *pioneer_utils/navigation/common/amcl.launch*

Código 5.10: Launchfile del nodo amcl utilizado.

5.4 Mapas de coste: Costmaps

Como ya se ha referido con anterioridad, el modelo de navegación que sigue el Navigation Stack de ROS se basa en el concepto de Costmaps.

Un costmap es un mapa generado a partir de los obstáculos detectados por sensores que son representados en forma de mapa de celdas y sobre los que se calcula un gradiente de "coste" desde los obstáculos hasta las zonas despejadas. Este tipo de mapas pretenden disponer de la información necesaria para que el robot pueda navegar.

Generalmente, este enfoque solo tiene en cuenta navegación en un solo plano o navegación en dos dimensiones, ya que todos los objetos son incorporados al mapa sin importar su altura. Existe una variante que pseudo-3D que tiene en cuenta la altura a la que se sitúan los sensores, aunque no es el modelo típico de aplicación.

El funcionamiento de este tipo de mapas se basa en la incorporación o el borrado de obstáculos al mapa. Cada sensor puede utilizarse de manera independiente para incorporar o borrar obstáculos en estos mapas. Esto permite a nuestro sistema robótico abstraerse de la información directa de los sensores y obtener información más completa de todos los obstáculos su alrededor.

La principal ventaja de utilizar este tipo de mapas en navegación es la de crear mapas dinámicos locales en el que guardar información de los obstáculos alrededor del robot sin que necesariamente dispongamos de un sensor captando el mismo continuamente. Esto nos permite, por ejemplo, realizar navegación con un sensor con un rango horizontal de pocos grados, como el sensor Kinect, y aún así permitir

al robot conocer información de obstáculos que han quedado fuera del rango del sensor.

El nodo encargado de la creación de estos costmaps es *costmap_2d* y su funcionamiento está basado en diferentes capas que manejan la información de los sensores y generan cálculos a partir de los mismos.

Las denominadas capas de las que hace uso *costmap_2d* son las siguientes:

- Static Map: Esta capa es la encargada de tener el cuenta los obstáculos que aporta un mapa pregrabado en caso de utilizarse en la navegación. Está ligado al parámetro *static_map* y normalmente se utiliza con nodos de localización en un mapa como *amcl*.
- Obstacle Map: Esta capa es la encargada de incorporar o borrar obstáculos obteniendo la información de los sensores declarados en su archivo de configuración. Destaca la facilidad para incorporar varios sensores a la vez.
- Inflation: En esta capa se realizan los cálculos de coste de cada celda a partir de la información que aportan las capas anteriores. Este gradiente de valores clasifica cada celda del mapa en diferentes tipos:
 1. Letal: Existe un obstáculo real en esa celda del mapa
 2. Inscrito: Las celdas con este valor se encuentran a una distancia de un obstáculo que es menor o igual al radio de la circunferencia inscrita en el área que ocupa el robot.
 3. Posiblemente circunscrito: Celdas que se encuentran a una distancia de un obstáculo que es mayor que el radio de la circunferencia inscrita en el área que ocupa el robot pero menor que el radio de la circunferencia circunscrita. Esta información como "posible" debido que la orientación del robot puede no ser exacta.
 4. Espacio libre: El coste de este tipo de celdas es cero e indica que no existe ningún obstáculo cercano que impida o limite el movimiento del robot.
 5. Desconocido: No existe información sobre el estado de esa celda.

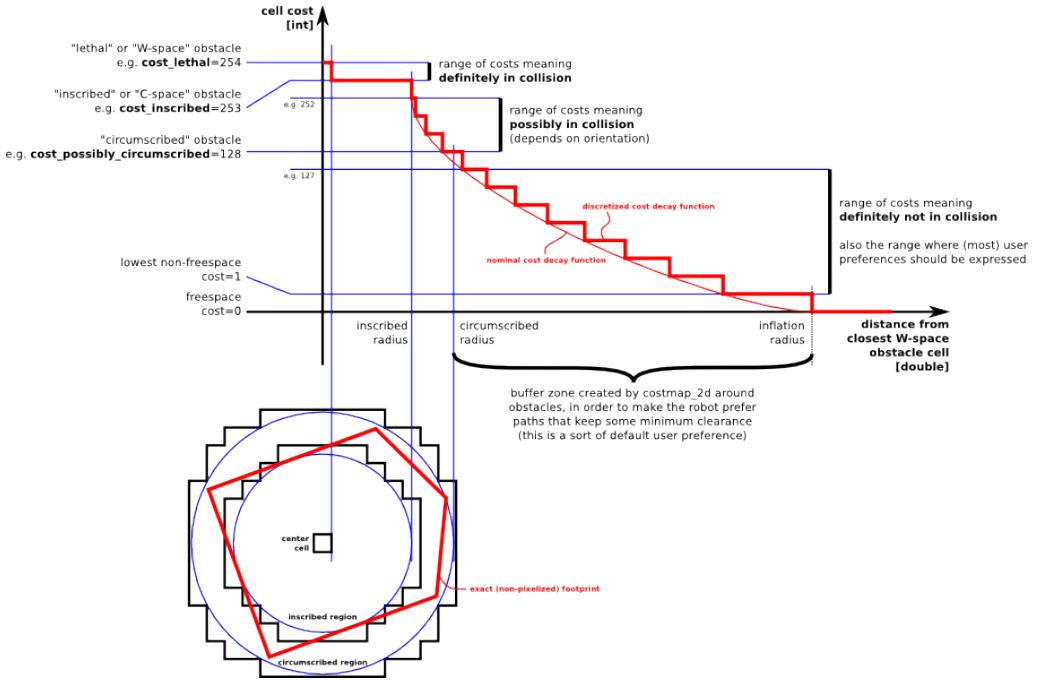


Figura 5.6: Esquema sobre el cálculo del coste de cada celda en el mapa.

Esta capa indica a los planificadores las zonas óptimas por las que trazar su ruta, es la más importante y donde reside gran parte de la funcionalidad de los costmaps. Existen parámetros configurables por el usuario como el radio de inflado (*inflation_radius*) o el valor de escala (*cost_scaling_factor*).

A parte de estas capas, existe una API que nos permite crear nuestras propias capas para el nodo *costmap_2d* para darles la funcionalidad que se desee.

Existe una configuración por defecto que de estos costmaps que genera una sola capa de obstáculos, una capa de inflado y adicionalmente una capa de mapa estático siempre y cuando pongamos el parámetro *static_map* a TRUE. Sin embargo, existe la posibilidad de crear costmaps con las capas que consideremos oportunas, siendo de especial interés poder incorporar diferentes capas de obstáculos.

Este último enfoque es el que ha sido adoptado en la implementación final de navegación y será descrito en la sección 7.2.1.

5.5 Planificador de trayectoria global

La planificación de trayectoria es una de las áreas que más interés suscita entre los investigadores relacionados con la robótica móvil o la navegación autónoma, no en vano, existe una gran variedad de algoritmos que realizan cálculos apoyándose en distintos enfoques matemáticos.

Los planificadores de trayectoria tratan de construir la ruta que lleve al robot desde su posición a un punto en un mapa o en un entorno que pudiera ser desconocido. Esta planificación es calculada en un ámbito que denominamos "global" ya que se realiza en función al mapa global (*global_costmap* en nuestro caso) y no se consideran los detalles del entorno local al robot. Es, por tanto, una aproximación al camino final que el robot va a seguir.

Por tanto, los planificadores de trayectoria global deben basarse en información almacenada del entorno como un mapa estático y el cálculo se realiza mediante el análisis de sus características.

En la figura 5.7 se muestra una clasificación de los principales métodos de planificación de trayectorias

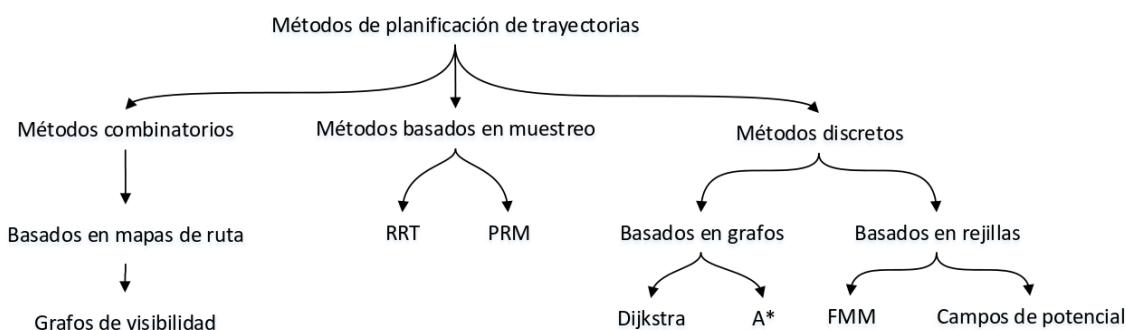


Figura 5.7: Clasificación de los diferentes métodos de planificación de trayectorias. Basado en [PB15].

Los algoritmos implementados en ROS se basan en métodos discretos mediante grafos, debido a que los mapas utilizados son mapas de celdillas que pueden transformarse en un grafo por medio de la propiedad de vecindad. A continuación se describen los planificadores de trayectoria disponibles en ROS, su fundamento teórico y su comportamiento en el cálculo de la trayectoria.

5.5.1 Algoritmo de Dijkstra

El algoritmo de Dijkstra es un algoritmo para la determinación del camino más corto dado un vértice origen al resto de vértices de un grafo con costes en cada arista. Este algoritmo debe su nombre a Edsger Dijkstra, quien lo describió por primera vez en el año 1959.

El algoritmo de Dijkstra funciona visitando celdas partiendo desde una celda inicial. Examina repetidamente las celdas vecinas que aún no han sido examinadas expandiéndose desde el punto de partida hasta el punto de meta. De esta forma el algoritmo garantiza que se encuentre uno de los caminos más cortos (pueden existir varios caminos igual de cortos).

```

1 Function Dijkstra (Graph, source):
2     for each vertex v in Graph:           // Initializations
3         dist[v]:= infinity;               // Unknown distance function from
4         previous[v]:= undefined;          // Previous node in optimal path
5     end for                                // from source
6
7
8     dist [source]:= 0;                      // Distance from source to source
9     Q:= the set of all nodes in Graph;      // All nodes in the graph are
10    // unoptimized - thus are in Q
11    while Q is not empty:                  // the main loop
12        u:= vertex in Q with smallest distance in dist []; // Source node in
first case
13        remove u from Q ;
14        if dist[u] = infinity:
15            break;                         // all remaining vertices are
16        end if                            // inaccessible from source
17
18        for each neighbor v of u:          // where v has not yet been
19            // removed from Q.
20            alt:= dist[u] + dist_between (u, v);
21            if alt < dist[v]:              // Relax (u, v, a)
22                dist[v]:= alt;
23                previous[v]:= u;
24                decrease-key v in Q;       // Reorder v in the Queue
25            end if
26        end for
27    end while
28    return dist;
29 end function

```

Figura 5.8: Pseudo-código del algoritmo de Dijkstra. Basado en [Tom14].

En la figura 5.9 se pueden observar la implementación del algoritmo de Dijkstra. El punto de color rosa es el punto inicial y el azul oscuro el punto de meta.

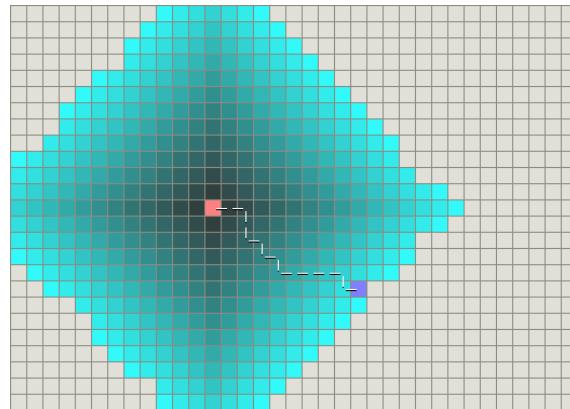


Figura 5.9: Implementación del algoritmo de Dijkstra en un mapa sin obstáculos. Basado en [Pat10].

Comparándolo con el algoritmo heurístico "Greedy Best-First-Search algorithm", cuyo funcionamiento es similar al algoritmo de Dijkstra aunque explora primero las celdas que se encuentran en dirección al punto de meta, vemos en la figura 5.10 que el camino encontrado es similar y que el número de nodos analizados se reduce considerablemente.

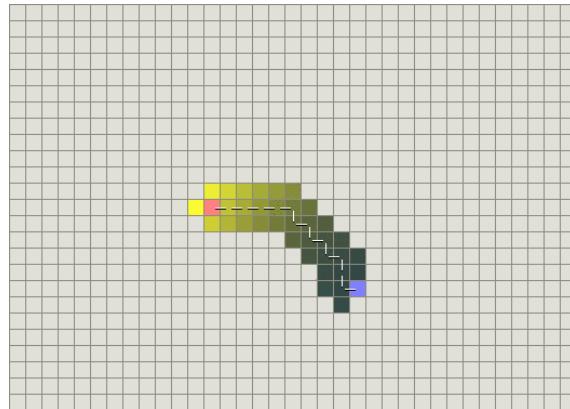


Figura 5.10: Implementación del algoritmo Best-First-Search en un mapa sin obstáculos. Basado en [Pat10].

Sin embargo, estamos analizando el mejor de los casos, cuando el mapa se encuentra libre de obstáculos. Situando un obstáculo cóncavo delante del punto inicial obtenemos el resultado de la figura 5.11.

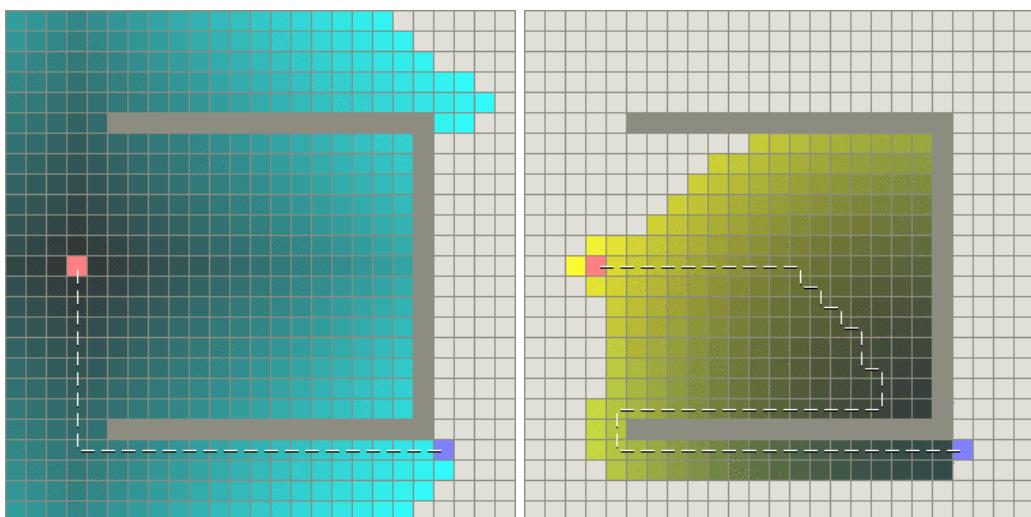


Figura 5.11: Algoritmo de Dijkstra (Izq.) y algoritmo Best-First-Search (Dcha.) en un mapa con obstáculo cóncavo. Basado en [Pat10].

El algoritmo Best-First-Search realiza un análisis de celdas menor hasta llegar al punto de meta comparado con el algoritmo de Dijkstra, lo cual se significa un menor coste computacional, sin embargo el camino obtenido no es el más corto, ya que Best-First-Search prioriza aquellas celdas que se encuentran más cerca del punto de meta.

El algoritmo de Dijkstra garantiza que se encuentre el camino más corto entre los dos puntos a costa de realizar un análisis de celdas mayor. Como vemos, este algoritmo es más inmersivo y se expande por una cantidad mayor de nodos, sin embargo su resultado es el óptimo.

5.5.2 Algoritmo de A estrella

El algoritmo A estrella (denominado comúnmente A*) es un algoritmo para la búsqueda del camino más corto entre las celdas de origen y objetivo dentro de un grafo con costes en cada arista. Fue presentado por primera vez en 1968 por Peter E. Hart, Nils J. Nilsson y Bertram Raphael.

```

1 Function A*(start, goal)
2   closedset: = the empty set      // the set of nodes already evaluated.
3   openset: = {start}            // the set of tentative nodes to be evaluated,
                                //initially containing the start node
4   came_from:= the empty map    // the map of navigated nodes.
5   g_score [start]:= 0          // Cost from start along best known path.
6   // Estimated total cost from start to goal through y.
7   f_score [start]:= g_score [start] + heuristic_cost_estimate (start, goal)
8   while openset is not empty
9     current: = the node in openset having the lowest f_score [] value
10    if current = goal
11      return reconstruct_path (came_from, goal)
12      remove current from openset
13      add current to closedset
14      for each neighbor in neighbor_nodes (current)
15        if neighbor in closedset
16          continue
17        tentative_g_score:= g_score [current]+ dist_between (current,neighbor)
18        if neighbor not in openset or tentative_g_score < g_score [neighbor]
19          came_from [neighbor]:= current
20          g_score [neighbor]:= tentative_g_score
21          f_score [neighbor]:= g_score [neighbor]+ heuristic_cost_estimate
                                (neighbor, goal)
22        if neighbor not in openset
23          add neighbor to openset
24        return failure
25  Function reconstruct_path (came_from, current_node)
26  if current_node in came_from
27    p:= reconstruct_path (came_from, came_from[current_node])
28    return (p + current_node)
29  else
30    return current_node

```

Figura 5.12: Pseudo-código del algoritmo A*. Basado en [Tom14].

El algoritmo A* combina el enfoque de algoritmos como Best-First-Search y Dijkstra. Este algoritmo favorece la creación de un camino por celdas que se encuentren cerca del punto inicial así como aquellas que se encuentren en dirección al punto de meta. El cálculo del camino se basa en considerar tanto el coste de llegada hasta un nodo como en la estimación de la distancia que falta para alcanzarlo. Si la estimación es menor que el coste real, el resultado es óptimo.

En la figura 5.13 podemos observar el comportamiento del algoritmo en un mapa despejado y en otro con un obstáculo cóncavo. Las celdas de color amarillo muestran la transición partiendo desde el punto inicial y las de color azulado muestran la transición desde el punto final.

El algoritmo A estrella ofrece por tanto unos resultados similares al algoritmo de Dijkstra en cuanto al camino calculado. Sin embargo análisis de los vértices es similar e incluso menor Best-First-Search y por tanto menor que en el de Dijkstra.

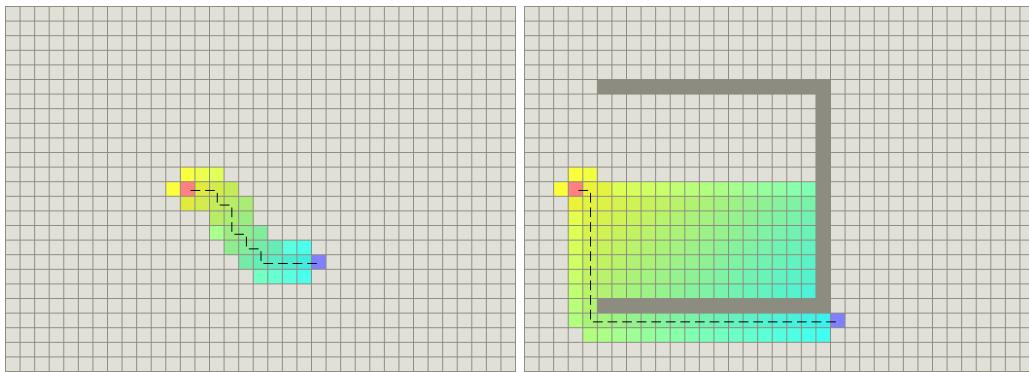


Figura 5.13: Algoritmo A* sin obstáculo (Izq.) y con obstáculo cóncavo (Dcha.). Basado en [Pat10].

5.6 Planificador de trayectoria local

El planificador de trayectoria local es el encargado de guiar al robot, según sus posibilidades de movimiento, hacia la ruta marcada por el planificador global esquivando obstáculos de manera reactiva. Este planificador de trayectoria se basa en el mapa de coste local, el cuál determina por la información de los sensores los obstáculos más próximos al robot.

La misión del planificador de trayectoria local es resolver las obstrucciones sobre la ruta global en el entorno local al robot para determinar la ruta real que será seguida. El modelo del entorno local se construye mediante la fusión de la información proporcionada por los sensores exteroceptivos del robot móvil.

EL planificador de trayectoria local ya implementado en ROS utiliza los algoritmos *Trajectory Rollout* y *Dynamic Window* para realizar la navegación reactiva basada en un solo plano. Los pasos que realizan estos algoritmos son los siguientes:

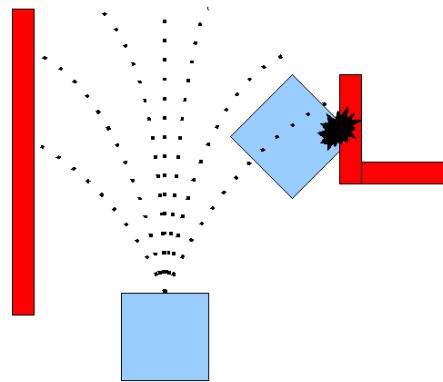


Figura 5.14: Esquema del funcionamiento de los planificadores de trayectoria local.

1. Muestreo discreto del espacio de control del robot según sus velocidades en cada eje.
2. Para cada velocidad, realizar una simulación desde el estado actual del robot para predecir la posición del robot durante un periodo corto de tiempo.
3. Evaluar cada una de las trayectorias resultantes del paso anterior utilizando características como la proximidad a los obstáculos, proximidad a la meta, proximidad a la ruta global y la velocidad.

4. Repetir los pasos anteriores hasta alcanzar el punto final de meta.

La diferencia entre los algoritmos *Trajectory Rollout* [GK08] y *Dynamic Window* [FBT⁺97] es el modo en el que se realiza el muestreo del espacio de control del robot.

Trajectory Rollout muestrea todas las trayectorias alcanzables variando la velocidad en todo el periodo de simulación, mientras que *Dynamic Window* realiza la evaluación de las trayectorias en cada uno de los pasos, evaluando la velocidad comparándola con las capacidades de aceleración del robot.

Dynamic Window es un algoritmo más eficiente pero que se comporta peor en robots con bajos límites de aceleración por lo que en ocasiones el algoritmo *Trajectory Rollout* realiza trayectorias locales más adecuadas.

Es importante que la configuración de los parámetros de velocidad y aceleración del robot sean los adecuados, de tal forma que la simulación de posibles trayectorias sea realizada de acuerdo a las capacidades específicas del robot.

Capítulo 6

Control a bajo nivel

En este capítulo se realiza una primera aproximación al control del robot y a los nodos básicos que deben ejecutarse para controlarlo y acceder a la información de los sensores.

6.1 Nodos Hardware

Los nodos necesarios para el control del robot requieren el acceso a los motores y a la lectura de la odometría. También es necesario disponer de controladores para los dos sensores exteroceptivos utilizados y que su información se publique en tipos de datos reconocibles por ROS. Los nodos utilizados para estos dispositivos hardware se describen a continuación.

6.1.1 Control del robot: Rosaria y p2os

Como hemos indicado anteriormente, la librería que nos permite el acceso a la placa controladora de nuestro robot Pioneer es Aria. Esta librería, proporcionada por Adept Mobile Robots, permite realizar el control completo del robot y acceder a sus parámetros configurables.

Los paquetes disponibles en ROS para el control de los robots de la familia Pioneer son dos, por un lado tenemos *rosaria* y por otro *p2os*.

- **p2os** [All11] es un paquete que agrupa un conjunto de utilidades y nodos desarrollados para controlar el robot. Su característica principal es que accede de manera nativa a la placa controladora del robot por lo que no depende de la librería Aria. Además incorpora funcionalidades adicionales como modelos 3D de robot, simulación con Gazebo o la configuración de la navegación.

Sin embargo, *p2os* no integra todas las funcionalidades a las que tiene acceso Aria como es la reconfiguración de los parámetros de la odometría.

- **rosaria** [ROS15] es un nodo de interfaz entre ROS y Aria, por tanto incluye todas prácticamente todas las funcionalidades de esta. Podemos acceder a la calibración de los encoders de la odometría así como conectar con el simulador *MobileSim* (ver sección 8.1).

El paquete que se ha utilizado para controlar el robot a bajo nivel es *rosaria* debido a que integra funcionalidades adicionales importantes para el ajuste odométrico y a que el paquete *p2os* se encuentra desactualizado en la versión ROS que utiliza.

A continuación se muestra el *launchfile* para ejecutar el nodo *RosAria*:

```

1 <launch>
2   <!-- Starting rosaria driver for motors and encoders -->
3   <node name="rosaria" pkg="rosaria" type="RosAria" args="_port:=/dev/ttyUSB0">
4     <rosparam>
5       TicksMM: 166
6       RevCount: 37350
7       DriftFactor: 0
8     </rosparam>
9     <remap from="~cmd_vel" to="cmd_vel"/>
10    </node>
11  </launch>

```

Fuente: *pioneer_utils/sensors/rosaria.launch*

Código 6.1: Launchfile para RosAria.

Como puede verse, los valores de la odometría son configurables (más información la subsección A.2.1 del apéndice).

Como cualquier nodo en ROS, el nodo *RosAria* publica una serie de Topics y se suscribe a otros para poder intercambiar información entre otros nodos. En la tabla 6.1 se muestra parte de la API utilizada de *RosAria*.

RosAria API			
Topics suscritos	Mensaje	Descripción	
cmd_vel	geometry_msgs/Twist	Recibe los comandos de velocidad	
Topics publicados	Mensaje	Descripción	
pose	nav_msgs/Odometry	Publica la odometría	
Parámetros	Tipo	Descripción	
port	string	Puerto serie del robot	
TicksMM	float	Calibración de la odometría	
DriftFactor	float	Rozamiento de la odometría	
RevCount	float	Calibración de los encoders	
Frames publicados		Descripción	
base_link		Referencia base del robot	
odom		Referencia odométrica	

Tabla 6.1: API de *rosaria* utilizada. Basado en [ROS15].

6.1.2 Sensor Kinect

Para la puesta en marcha del sensor Kinect existen en ROS diferentes paquetes que utilizan una u otra librería de código en función de quién lo haya desarrollado.

Existen dos paquetes destinados al control del sensor Kinect:

- Por un lado tenemos *openni_kinect* [RF11], que utiliza los drivers de la librería OpenNI. Este paquete y en concreto los drivers del dispositivo han sido utilizados ampliamente tanto en desarrollos realizados con ROS como fuera de este entorno. Sus características principales son la total funcionalidad, aprovechamiento de toda la tecnología de este sensor y capacidad para monitorear la posición del esqueleto de una persona. Sin embargo, el soporte del paquete *openni_kinect* solo se mantuvo activo hasta a versión ROS Fuerte debido a la compra de PrimeSense, empresa creadora del sensor y miembro fundador del proyecto OpenNI, por la conocida marca de informática Apple [Pre13].

- Por otro lado, gracias al gran desarrollo software llevado a cabo por la comunidad OpenSource, disponemos de los divers *libfreenect* desarrollados por el proyecto OpenKinect [Com10] que trata de ofrecer una vía alternativa para controlar el sensor de Microsoft. Estos drivers se encapsulan y adaptan su interfaz a ROS a través del paquete *freenect_stack* [Kha11] el cual nos ofrece acceso tan solo a la imagen y la nube de puntos del sensor. Su integración no es completa, no dispone de características adicionales como el monitoreo de la posición de una persona, sin embargo su funcionamiento es correcto y está adaptado a ROS en su versión Indigo por lo que ofrece la posibilidad de integrarlo en nuestro sistema.

La comparación anterior denota que el software más adecuado para acceder a la nube de puntos del sensor Kinect es el paquete *freenect_stack*, por lo que éste ha sido el utilizado en este proyecto.

Al ser un paquete de terceros, debe clonarse desde su repositorio de código fuente e incorporarlo al entorno Catkin. La puesta en marcha del mismo es bastante inmediata haciendo uso de los *launchfiles* que ofrece *freenect_launch* desde consola de la siguiente forma:

```
1 rosrun freenect_launch freenect.launch
```

Fuente: *freenect_stack/freenect.launch/launch/freenect.launch*

Código 6.2: Launchfile para Kinect en el paquete *freenect_launch*.

En la tabla 6.2 se muestra parte utilizada de su API.

libfreenect_stack API		
Topics publicados	Mensaje	Descripción
camera/depth/points	sensor_msgs/PointCloud2	Publica la nube de puntos
camera/depth/image_raw	sensor_msgs/Image	Imagen captada
camera/depth/camera_info	sensor_msgs/CameraInfo	Información de la cámara
Frames		
camera_link		Referencia base de Kinect
camera_rgb_frame		Referencia cámara RGB
camera_depth_frame		Referencia cámara IR

Tabla 6.2: API de *freenect_stack* utilizada.

6.1.3 Sensor Láser Sick LMS100

Existe un amplio soporte en ROS para sensores láser de la marca Sick, entre los más populares se encuentra la familia Sick LMS200, que se utiliza en muchos desarrollos relacionados con la robótica móvil. La familia de sensores LMS200 utiliza una interfaz de comunicación en serie a través de puerto RS-232, sin embargo, la familia de dispositivos Sick LMS100 utiliza interfaz ethernet y requiere un tratamiento de datos diferente.

El láser Sick LMS100 ha sido integrado en ROS y utilizado en este proyecto ya que se había dado uso en proyectos anteriores [HdC13] y se consideró conveniente incorporarlo debido a su precisión y rango.

Para acceder al sensor Sick LMS100 se utiliza el paquete *LMS1xx* desarrollado por Clearpath Robotics [Cle10] que se basa en los drivers desarrollados en la librería *libLMS1xx*¹.

El paquete *LMS1xx* consta de un solo nodo que se conecta a través de una IP indicada como parámetro. Para conectarse al sensor láser es necesario configurar el puerto del ordenador con una IP fija dentro del mismo rango que la IP del sensor, la puerta de enlace queda vacía y utilizamos la máscara de subred por defecto.

Seguidamente basta con indicar en el archivo *launchfile* la IP del sensor.

```

1 <launch>
2   <arg name="host" default="192.168.1.14" />
3   <node pkg="lms1xx" name="lms1xx" type="LMS1xx_node">
4     <param name="host" value="$(arg host)" />
5   </node>
6 </launch>
```

Fuente: *LMS1xx/launch/LMS1xx.launch*

Código 6.3: Launchfile para el sensor Láser Sick LMS100.

Pueden precisarse algunos ajustes previos para ajustar la dirección IP del sensor y algunos de sus parámetros. Esto puede hacerse mediante la herramienta "SO-PAS Engineering tool." ofrecida el fabricante. Estos ajustes pueden encontrarse en el apéndice de este trabajo (Sección A.2.4).

La API de este nodo se muestra en la tabla 6.3.

LMS1xx API		
Topics publicados	Mensaje	Descripción
/scan	sensor_msgs/LaserScan	Puntos láser
Parámetros	Tipo	Descripción
host	string	Dirección IP del láser
Frames		Descripción
laser		Centro del haz láser

Tabla 6.3: API de LMS1xx utilizada.

6.1.4 Integración del hardware

Una vez se dispuso de todos los paquetes necesarios para poner en funcionamiento todo el hardware en el robot, era necesario integrar todos los nodos bajo una misma configuración y definir mediante transformadas la posición de los sensores en el robot.

A partir de un nuevo archivo *launchfile* (Código 6.4) se enlanzan cada nodo con las configuraciones del hardware y se definen las transformadas estáticas entre cada uno de los *frames*.

Como resultado, se obtiene una relación entre cada sistema de referencia (*frame*), lo cual permite realizar transformaciones de los datos entre cada uno de ellos. Esto también sirve para que el robot "sea consciente" de su configuración (Figura 6.1).

¹Repositorio software *libLMS1xx*: <https://github.com/konradb3/libLMS1xx>

```

1 <launch>
2
3 <!-- Launching p2os RobotModel --> OJO!!!!!!!!!!!!!!!!!!!!!!!
4   <include file="$(find p2os_urdf)/launch/pioneer3at_urdf.launch"/>
5
6 <!-- Launching LMSixx_node for laser Sick LMS100 via ethernet -->
7   <include file="$(find pioneer_utils)/sensors/LMSixx.launch"/>
8
9 <!-- start sensor-->
10 <include file="$(find freenect_launch)/launch/freenect.launch"/>
11
12 <!-- Launch kinect and depthimage_to_laser node -->
13   <include file="$(find pioneer_utils)/sensors/kinect_to_laser_low.launch"/>
14
15 <!-- Launch kinect and depthimage_to_laser node -->
16   <include file="$(find pioneer_utils)/sensors/kinect_to_laser.launch"/>
17
18 <!-- Starting rosaria driver for motors and encoders -->
19   <include file="$(find pioneer_utils)/sensors/rosaria.launch"/>
20
21   <node pkg="tf" type="static_transform_publisher" name="base_to_laser_broadcaster" args="-0.2
22     0 0.390 3.141592 0 0 base_link laser 1" />
23   <node pkg="tf" type="static_transform_publisher" name="base_to_camera_broadcaster" args
24     ="0.020 0 0.375 0 0 0 base_link camera_link 1" />
25
26 </launch>

```

Fuente: *pioneer_utils/sensors/pioneer3at-rosaria.launch*

Código 6.4: Launchfile creado para robot Pioneer 3 AT.

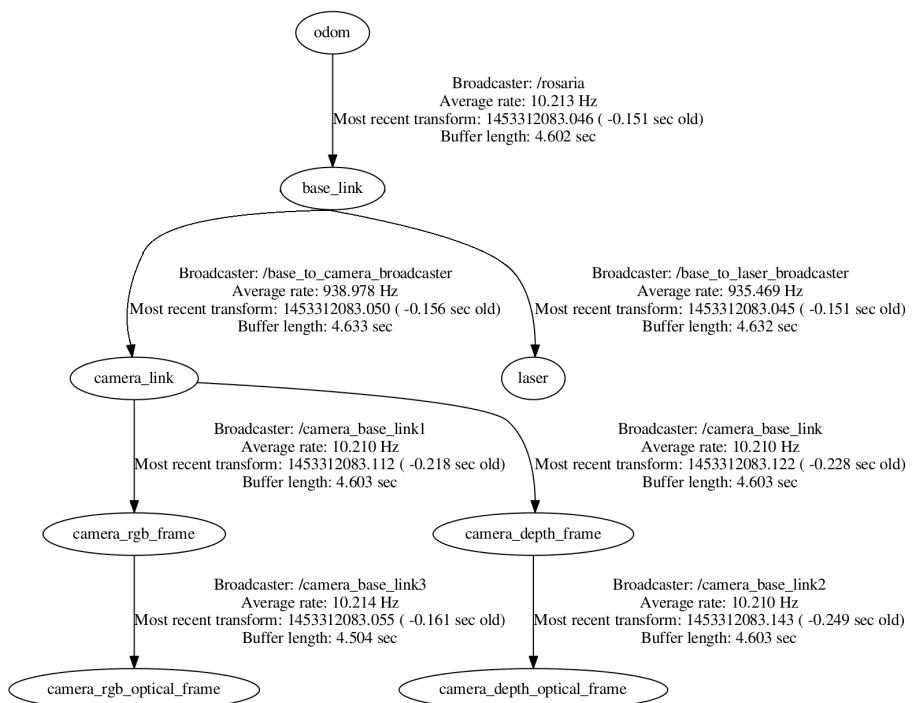


Figura 6.1: Referencias *frames* de la configuración del robot.

6.2 Nodo de teleoperación

Uno de los primeros objetivos de este proyecto consiste realizar el control teleoperado del robot. Utilizando ROS y sus características para operar de manera distribuida en diferentes máquinas, esta tarea se vuelve inmediata para el usuario.

6.2.1 Características distribuidas de ROS

ROS trabaja en forma de procesos que se ejecutan de manera independiente y se comunican a través del nodo principal MASTER mediante el paso de mensajes. Ya que el nodo MASTER dispone de una dirección IP en la máquina que lo ejecuta, basta con indicar en el entorno ROS de cada máquina la dirección de este para que los nodos abran una comunicación con esa dirección IP.

El nodo MASTER en este caso se ejecutará en el ordenador Intel NUC del robot, por lo que realizará la función de servidor de datos de cada nodo.

Los pasos para configurar las máquinas bajo la misma red se describen con detalle en la guía *ROS Network Setup* [ROS14] y consisten en indicar en el script *.bashrc*² los parámetros ROS_IP y ROS_MASTER_URI.

ROS_IP debe contener la IP de nuestra máquina en la red que esté operando y ROS_MASTER_URI la dirección *http* correspondiente de la máquina donde se ejecute el nodo principal, especificando el puerto.

```

1 | export ROS_IP=10.42.0.1
2 | export ROS_MASTER_URI=http://10.42.0.1:11311

```

Código 6.5: Líneas del archivo *.bashrc* en el ordenador de abordo Intel NUC.

El procedimiento es el mismo en el caso del ordenador que va a servir como control remoto. En la sección del apéndice se encuentra más información sobre la conexión de un ordenador externo.

```

1 | export ROS_IP=10.42.0.77
2 | export ROS_MASTER_URI=http://10.42.0.1:11311

```

Código 6.6: Ejemplo *.bashrc* en un ordenador externo para realizar comunicación con el máster.

6.2.2 Implementación del nodo

Gracias a las características de sistema distribuido, podemos desarrollar un nodo ROS que se conecte al Topic *cmd_vel* de *RosAria* y publicar diferentes valores de velocidad en función de las teclas que se pulsen sin tener que preocuparnos por implementar un socket de comunicación.

El nodo que se ha desarrollado realiza una teleoperación básica del robot, publicando mensajes de velocidad fija. Se ha partido del nodo de teleoperación desarrollado en el paquete *turtlesim* [Fau11] ya que este reacciona ante pulsaciones instantáneas de teclado y favorece el control del robot.

²El script *.bashrc* se ejecuta cada vez que se abre una nueva terminal en el ordenador.

La API ROS de este nodo solo consta de un parámetro (6.4), el Topic de comandos de velocidad al que se conecta, por lo que esto lo hace reutilizable en cualquier otro robot que opere con ROS.

teleop_p3at API		
Topics publicados	Mensaje	Descripción
cmd_vel	geometry_msgs/Twist	Publica comandos de velocidad

Tabla 6.4: API de teleop_p3at

La implementación del nodo se ha realizado en C++ y su principal característica es que toma las pulsaciones del teclado en modo *raw* de tal modo que es detectada inmediatamente.

```

1 ...
2     // get the console in raw mode
3     tcgetattr(kfd, &cooked);
4     memcpy(&raw, &cooked, sizeof(struct termios));
5     raw.c_lflag &=~ (ICANON | ECHO);
6     // Setting a new line, then end of file
7     raw.c_cc[VEOL] = 1;
8     raw.c_cc[VEOF] = 2;
9     tcsetattr(kfd, TCSANOW, &raw);
10    puts("-----Reading from keyboard-----");
11    puts("-----");
12    puts("Use arrow keys to move Pioneer and SPACE to stop");
13
14
15    while(!g_request_shutdown && ros::ok())
16    {
17
18        // get the next event from the keyboard
19
20        if(read(kfd, &c, 1) < 0)
21        {
22            perror("read():");
23            exit(-1);
24        }
25
26        switch(c)
27        {
28            case KEYCODE_L:
29                vel.linear.x = 0;
30                vel.linear.y=0;
31 ...

```

Fuente: *pioneer_utils/src/teleop_p3at.cpp*

Código 6.7: Fragmento de código del nodo *teleop_p3at*.

6.3 Nodo de navegación estimada

La navegación estimada, más conocida en inglés como Dead Reckoning [Wik15], es la capacidad para realizar navegación en un entorno basándose solamente en la información que aportan los sensores de la odometría.

Es un método estimado de localización que se basa en la información de los encoders y no tiene en cuenta aspectos como el tipo de superficie, la inclinación, el rozamiento o incluso obstáculos que puedan frenar o modificar el desplazamiento del robot (a pesar de que sus ruedas giren).

Este nodo de navegación puede utilizarse para indicar al robot que avance cierta cantidad de metros y que realice giros a derecha o izquierda en un determinado ángulo siempre tomando como referencia la información aportada por la odometría.

Este nodo toma los valores de la odometría y envía comandos de velocidad para que realice el movimiento indicado. A continuación se muestra su API (Tabla 6.5).

moving_alone API		
Topics suscritos	Mensaje	Descripción
pose	nav_msgs/Odometry	Recibe la odmetría
Topics publicados		
cmd_vel	geometry_msgs/Twist	Publica comandos de velocidad
Frames suscritos		
base_link		Referencia base del robot
odom		Referencia odométrica

Tabla 6.5: API del nodo *moving_alone*.

El nodo *moving_alone* está desarrollado en la interfaz C++ y su uso se muestra función *main*.

```

1 ...
2 int main(int argc, char** argv)
3 {
4     ros::init(argc, argv, "moving_alone");
5     ROS_INFO("Moving alone started");
6     ros::NodeHandle nh;
7     MoveAlone moving_alone(nh);
8     moving_alone.avanza(1.0f, 0.5f);
9     moving_alone.gira(true, 90.0f, 0.2f);
10    moving_alone.avanza(-0.5f, 0.1f);
11    return 0;
12 }
```

Fuente: *pioneer_utils/src/moving_alone.cpp*

Código 6.8: Fragmento de código del nodo *moving_alone*.

Capítulo 7

Implementación del sistema

Este capítulo describe los cambios, ajustes y modificaciones que, basados en la información anterior expuesta, las características de ROS y el hardware del que disponemos, se han realizado para alcanzar los objetivos del proyecto.

7.1 Configuraciones hardware

Como ya se ha descrito, la navegación se basa en el sensor Kinect y en el sensor láser Sick. Estos sensores han sido incorporados al robot de manera que disponen de alimentación desde las baterías del mismo y su interfaz de conexión es conexión la correspondiente en cada caso.

En esta sección se describen las implementaciones que se han llevado a cabo en el hardware utilizado en este proyecto.

7.1.1 Pioneer 3 AT

El robot Pioneer 3 AT de Adept Mobile Robots es la base de la plataforma robótica. El modelo que ha sido utilizado en el laboratorio de la Escuela Técnica Superior de Ingeniería y Diseño Industrial llevaba incorporado un ordenador de tipo dual-core¹. Al comienzo de este proyecto ya existían algunas adaptaciones como la incorporación de un altavoz frontal, acceso a los puertos USB del ordenador interno y conexión para el sensor láser (Figura 7.1).

El robot había sido utilizado mediante el software MRCore [RLHVdlP13] en el proyecto anterior [HdC13] y el sistema operativo del ordenador interno era Ubuntu Server 10.

Para integrar la versión Indigo de ROS lo más recomendable era partir de la versión estable más actualizada de Ubuntu, por lo que se sustituyó el sistema operativo por Ubuntu 14.04 LTS en su versión de escritorio.

Una vez integrado el sistema operativo, la primera toma de contacto con el robot fue a partir de la librería Aria para controlar el movimiento de los motores y comprobar que el robot se encontraba en buen estado.

A continuación, tras instalar ROS Indigo, se procedió a las pruebas mediante el paquete Rosaria de ROS. La conexión con el microcontrolador de la placa de motores fue exitosa y se comprobó que los valores de la odometría también funcionaban.

¹Ordenadores integrados en los robots de Mobile Robots: <http://www.mobilerobots.com/Accessories/EmbeddedComputers.aspx>



Figura 7.1: Estado del robot al comienzo del proyecto [HdC13].

Llegados a este punto, el robot se encontraba en disposición para realizar las primeras pruebas.

7.1.2 Sensor Láser

El sensor láser Sick también había sido integrado en un proyecto anterior y sus conexiones de alimentación y datos vía Ethernet ya estaban preparadas para utilizarlo.

Para conectarlo a través del puerto Ethernet fue necesario ajustar su dirección IP a través del software del fabricante y ajustar la IP del ordenador del robot Pioneer (más información en el apéndice A.2.4).

El agarre mecánico del sensor se dejó tal y como había sido utilizado en ocasiones anteriores, situado en la parte frontal agarrado mediante un par de tornillos al chasis con tuercas de palometa para su fácil manipulación.

El sensor láser se conecta a la interfaz ROS mediante el paquete *LMS1xx* tal y como se describió en el apartado 6.1.3.

7.1.3 Sensor Kinect

La integración del sensor Kinect fue relativamente sencilla debido a que las entradas de los puestos USB del ordenador habían sido cableadas previamente. La adaptación a realizar era sobre la parte de alimentación, ya que este sensor trabaja a una tensión de 12 voltios.

En el manual del robot se encuentra una descripción detallada de la placa de alimentación a la cual pueden conectarse diferentes periféricos. Esta placa ofrece tomas de conexión de 5 voltios controlados por unos botones auxiliares y tomas de 12 voltios (ver apéndice).

El sensor Kinect dispone de un adaptador USB, preparado para trabajar con la videoconsola XBOX 360, el cual suministra 12 voltios mediante un transformador conectado a una toma de corriente alterna de 220v e incorpora los cables de datos del propio sensor Kinect.

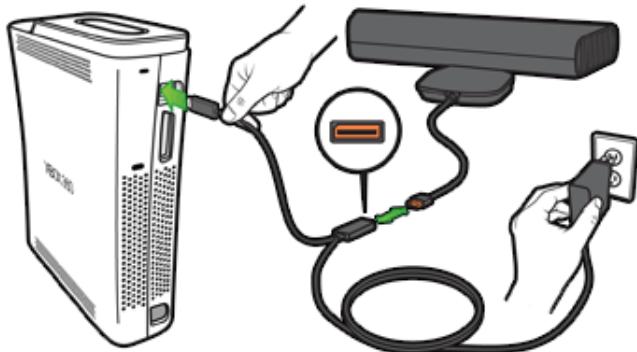


Figura 7.2: Conexión usual del sensor Kinect a la consola Xbox.

Para integrar el sensor Kinect en el robot, se adaptó el cable de alimentación soldando unas clavijas de conexión para obtener directamente alimentación a 12 voltios de la placa del robot. También se realizó lo oportuno en el adaptador de corriente, para poder usar el sensor Kinect de la manera habitual (Figura 7.3).



Figura 7.3: Adaptación de cables para la alimentación del sensor Kinect (izq.) y cable a 12V de la placa de alimentación del robot (dcha.).

Para anclar el sensor Kinect al robot se optó por situarlo en la parte superior del sensor Láser, para lo cual se diseñó una pieza que encajase en la base de la Kinect y en el sensor láser (Figura 7.4).

El sensor Kinect se conecta a la interfaz ROS mediante el paquete `freenect_stack` tal y como se describió en el apartado 6.1.2.

7.1.4 Primera configuración hardware

La primera configuración del robot consistió en ambos sensores situados en la parte frontal del mismo (Figura 7.4). Los sensores se encontraban colocados de manera

vertical otro, de tal forma que no existieran interferencias en su rango de detección.



Figura 7.4: Primera configuración hardware del robot: Kinect y láser en la parte frontal.

De esta forma se consigue una vista despejada y contamos con la información del láser para detectar obstáculos laterales.

Primera configuración del sistema

El ordenador interno corría todos los nodos de ROS, de modo que se disponía de la información de los sensores, el control sobre los motores y la lectura de la odometría para realizar las primeras pruebas con el paquete de navegación de ROS (Sección 5.1).

Sin embargo, la primera implementación con los primeros ajustes de los sensores no fue posible debido a la sobrecarga de la CPU del ordenador interno del robot Pioneer y a problemas de memoria en la ejecución de nodos como *amcl*.

Segunda configuración del sistema

La siguiente opción ha consistido en utilizar un ordenador externo para realizar los cálculos de navegación y enviar al robot las consignas de movimiento a través de una red inalámbrica (Figura 7.5). Esta idea no era la solución más ideal, ya que desde el principio la idea era que el robot fuese lo más autónomo posible sin depender de una infraestructura, sin embargo esta configuración no suponía mucho esfuerzo debido a que los nodos pueden ejecutarse de manera distribuida.

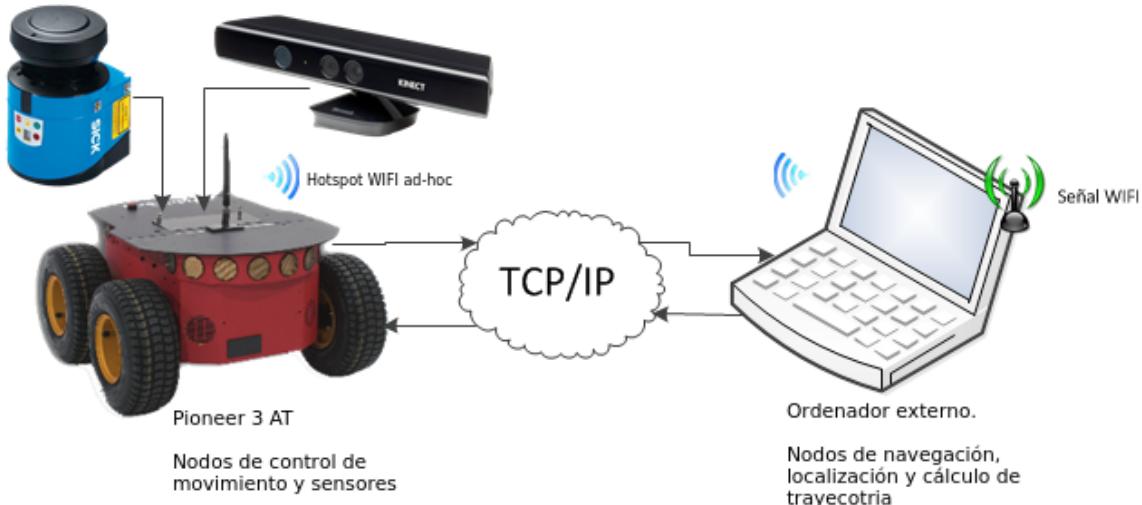


Figura 7.5: Esquema de la segunda configuración del sistema. Basado en [HdC13].

Esta configuración fue probada utilizando un ordenador portátil con suficiente capacidad de procesamiento y memoria como para ejecutar la navegación, sin embargo aparecieron algunos inconvenientes relacionados con la comunicación. Estos problemas se detallan a continuación.

- **Problemática del sensor Láser: LMS1xx**

Debido a que el sensor láser se conecta vía Ethernet al ordenador del robot, el nodo *LMS1xx* debe obtener información a través de la IP del láser y enviarla a través del adaptador Wifi a la IP del nodo *MASTER*. El problema reside en que el nodo se saturaba al tener que lidiar con ambas interfaces de conexión y provocaba su detención.

Tras varias consultas a *Clearpath Robotics* a través de su repositorio de *Github* y preguntas en el foro *ROS Answers* (ver sección C.2), la solución no estaba implementada en código y lo más inmediato era hacer un *bridge* en el ordenador del Pioneer 3 AT entre la interfaz Ethernet y la Wifi.

Los resultados de esta solución no fueron satisfactorios ya que el comportamiento era el mismo: el nodo *LMS1xx* se saturaba e interrumpía a los pocos minutos de su ejecución.

Tratando de resolver este problema, se hicieron pruebas generando una red Wifi Ad-hoc desde el ordenador del robot, a la cual se conectaba el ordenador externo. Con esta configuración resultados son satisfactorios siempre y cuando las IPs del nodo *MASTER* y del sensor Láser se encuentren en el mismo subrango.

Esta solución es la implementada en el sistema final.

Una vez conectado el ordenador externo, la ejecución del nodo de navegación es correcta y las consignas de movimiento se envían correctamente al robot, pudiendo realizar las primeras pruebas de navegación autónoma.

Sin embargo en ocasiones la recepción y el envío de datos era demasiado alta y esto provocaba que existiese mucho retraso en la comunicación, haciendo que el robot reaccionase tarde para esquivar los obstáculos y el control del robot fuera impracticable.

Tercera configuración del sistema

Finalmente se optó por montar un ordenador más potente en el robot, para lo cual se utilizó un portátil externo al que se conectaba tanto el sensor Kinect como el sensor láser. Para controlar el movimiento del robot y leer la odometría se utilizaba un convertidor de puerto USB a puerto serie RS-232. El ordenador del robot quedó por tanto apartado, sustituido por la incorporación del ordenador portátil mediante unos soportes realizados a medida en impresión 3D. Con esta configuración se realizaron las primeras pruebas de navegación autónoma del robot (Figura 7.6).



Figura 7.6: Robot con portátil incorporado realizando navegación y soporte realizado con la impresora 3D.

7.1.5 Segunda configuración hardware

La segunda configuración hardware vino dada tras las pruebas satisfactorias con el ordenador portátil como encargado de la ejecución del sistema ROS. Debido a ello se optó por utilizar el ordenador compacto Intel NUC de manera dedicada en el robot, sustituyendo al portátil, dejando más espacio para colocar los sensores y hacer un sistema más integrado.

Una de las principales desventajas con las que contaba la primera configuración hardware era la posición tan adelantada del sensor Kinect, ya que debido a sus características, si un objeto se situaba a medio metro o menos delante, las proyecciones de su emisor infrarrojo no pueden ser captadas por la cámara por lo que obtenemos una nube de puntos vacía (se dice que el sensor Kinect se queda "ciego"). La nueva configuración hardware del robot vino dada por el hecho de retrasar la posición de del sensor y obtener cierta distancia de margen para evitar el efecto anterior.

Tras varias pruebas en el simulador Gazebo (Sección 8.2) cambiando la posición del sensor Kinect, el ordenador Intel NUC y del sensor láser Sick, se aprovechó todo el área del robot de manera que ningún elemento estorbase a los haces de infrarrojos de ambos sensores.

En esta nueva configuración el sensor Kinect se ha situado más retrasado, hacia la mitad del robot, dejando tan solo sitio en la parte trasera para incorporar el resto de elementos. El sensor láser se ha situado en la parte trasera mirando hacia atrás

ya que gracias a su rango de 270° se obtienen lecturas de prácticamente todo el perímetro del robot (Figura 7.7).

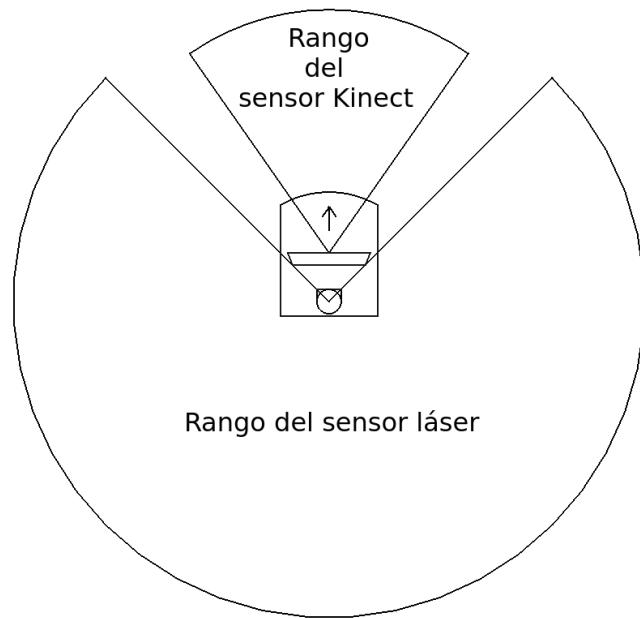


Figura 7.7: Esquema de la nueva configuración hardware.

El ordenador Intel NUC se ha situado al lado izquierdo del sensor láser dejando el lado derecho para situar el cableado de los sensores y del ordenador.

Finalmente, para dejar todo el sistema integrado, se han diseñado unos paneles laterales, un sistema de varillas roscadas atornilladas a la base del robot y un panel superior para esconder el cableado interno. Previamente se diseñó un modelo en 3D en el simulador *Gazebo* para comprobar su correcta compatibilidad con la disposición de los sensores (Ver figura 8.2).

Tras comprobar que el diseño funcionaba y cumplía las características necesarias se mecanizó la parte superior del robot para anclar el sensor láser y las varillas roscadas, como se ve en la figura 7.8.

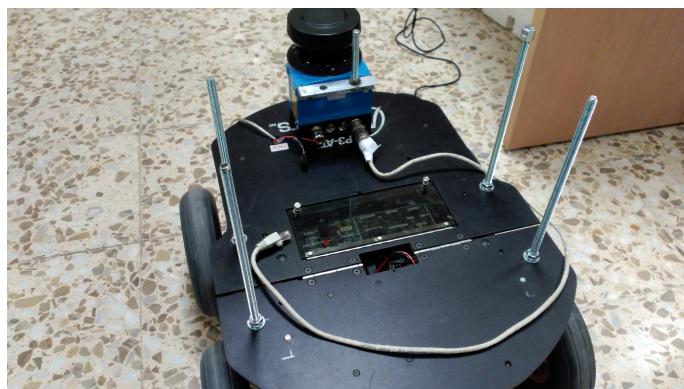


Figura 7.8: Posición retrasada del láser y nuevas varillas de soporte.

Además, para evitar vibraciones, anclar el sensor Kinect a la base del robot y situarlo a la altura necesaria, se diseñaron e imprimieron en 3D unas piezas de soporte (Figura 7.9).

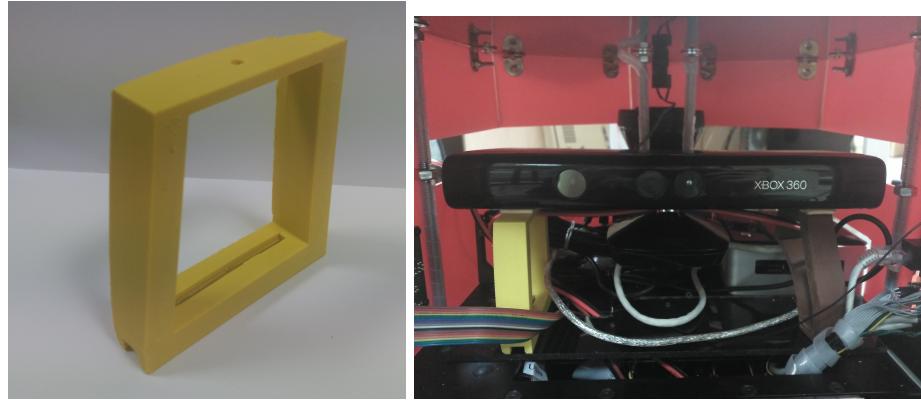


Figura 7.9: Pieza 3D para el soporte del sensor Kinect y su posición final.

Los paneles laterales y el panel superior se mecanizaron en metacrilato de 3 y 4 milímetros respectivamente con una fresadora y se sujetaron al robot mediante remaches, tornillos y escuadras, algunas de las cuales fueron diseñadas también en 3D.

El panel de control del robot quedó dividido en dos partes, la destinada al ordenador de abordo y la destinada al microcontrolador del robot. Ambas partes fueron recolocadas en los laterales del robot para facilitar su acceso.



Figura 7.10: Panelado del robot.

A continuación se muestran algunas imágenes del rediseño del robot.



Figura 7.11: Imágenes del diseño final del robot "Petrois".

Intel NUC

El ordenador compacto Intel NUC se eligió como unidad de procesamiento y ejecución del sistema ROS debido a sus altas capacidades de procesamiento y memoria. Además su bajo consumo y la posibilidad de alimentarlo a tensión de 12 voltios hacen que sea el ordenador ideal para este tipo de aplicación frente a otros ordenadores similares. La alimentación se realiza de la misma forma que en el caso de los sensores y su anclaje al robot se ha realizado mediante tiras del velcro adhesivo para su fácil manipulación en caso de ser extraído.

Sus características son superiores a las del portátil utilizado en la configuración previa del sistema, por lo que su desempeño es mayor y maneja el sistema ROS en navegación con adecuadamente.

Configuración del sistema final

La configuración final del sistema quedó definida como se indica en la figura 7.12. El ordenador Intel NUC se configuró para que generase en su arranque una red Wifi ad-hoc propia (que permite conectarse a ROS mediante ordenadores externos) dentro del mismo subrango que las direcciones IP del sensor láser para evitar el malfuncionamiento del mismo tal y como se ha indicado anteriormente.

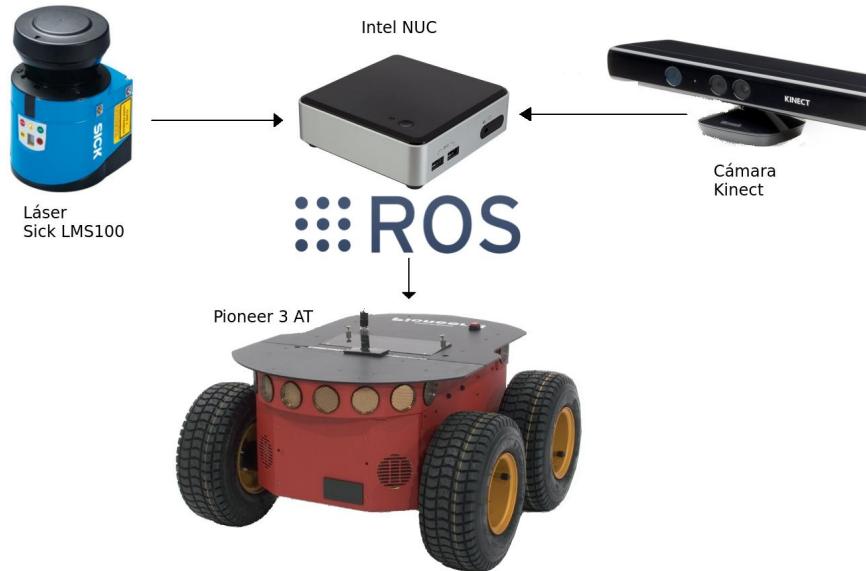


Figura 7.12: Esquema del sistema robótico final utilizado en el proyecto.

El suministro de energía a todos los elementos que incorpora el robot se realiza a través de la placa de alimentación del robot, la cual se conecta al pack de 3 baterías alojado en su interior.

7.2 Navegación

La navegación es el propósito central de este proyecto. Como ya se ha indicado anteriormente la navegación se centra en el Navigation Stack de ROS.

En el capítulo 5 se vió una descripción de cada una de sus partes y unos ajustes generales para nuestro robot. En este apartado se describen los ajustes específicos llevados implementados en el sistema robótico final.

Cabe decir que el robot ha sido adaptado y configurado de manera óptima basándonos en las características de la navegación en dos dimensiones que ofrece ROS tal y como se ha mostrado en el apartado anterior.

7.2.1 Configuración de los costmaps y los sensores

La navegación del robot se basa en la información recogida en los llamados *costmaps*.

Para realizar la navegación disponemos de dos costmaps, el llamado *global_costmap* y el llamado *local_costmap*. El primero sirve de base al planificador de trayectoria global y el segundo lo hace para el planificador de trayectoria local.

Su comportamiento y posibilidades de configuración son las mismas con la excepción de que el mapa global toma información del mapa que se cargue para realizar navegación (en caso de utilizar uno) además de la información aportada por los sensores.

La misión de los mapas de coste es la tomar la información de los sensores e incorporarla a un mapa de celdillas y marcar o borrar los obstáculos pertinentes. A partir de esa información se calcula un gradiente de coste que asigna un valor a cada una de ellas.

Los mapas de coste tal y como está implementados en ROS integran la información de los sensores en una misma capa, de tal forma que no existe distinción entre el tipo de información que está tomando un sensor u otro. Esto supone un problema añadido en el caso de este proyecto, ya que se pretende utilizar la información de los dos sensores: Kinect y láser.

• Problemática de la fusión sensorial

El problema consiste en que el sensor Kinect es capaz de tomar información de la posición de los obstáculos a diferente altura pero con un alcance más reducido. Sin embargo, el sensor láser dispone de un alcance mayor, pero no puede detectar obstáculos que queden por encima o por debajo de su haz.

Esto hace que, con la configuración de los mapas de coste con una sola capa de obstáculos, si el sensor Kinect incorpora un obstáculo al mapa que queda por debajo del haz láser y ese obstáculo deja de ser visto por el sensor Kinect pero permanece dentro del rango del láser, provoca que el obstáculo se borre del mapa.

Esta problemática se produce tanto para el mapa de coste global como para el local.

Esta problemática es bien conocida dentro del mundo de la robótica y se denomina "fusión sensorial" donde una de las técnicas más conocidas es la de fusión mediante Filtro de Kalman [LO03].

La solución más adecuada a esta problemática es realizar una composición de los puntos obtenidos por el sensor láser y el sensor Kinect. Sin embargo, el coste computacional de crear una nueva nube de puntos a partir de dos tipos de datos diferentes a una frecuencia adecuada se antoja elevado, por lo que la opción óptima consistió en utilizar capas de obstáculos diferentes (*costmap_2d::VoxelLayer*) para cada uno de los sensores.

De este modo, cada sensor es capaz de incorporar o borrar obstáculos del mapa solo si son detectados o no por ese mismo sensor y no por el otro. Si bien es cierto que de esta forma existen duplicidades de los obstáculos al tener que ser incorporados o borrados del mapa por cada sensor de manera independiente, esto nos permite salvar el caso en el que exista un obstáculo y este no se tenga en cuenta por interferencias de los sensores, que es la situación crucial a evitar.

• Problemática de la nube de puntos

Otro de los problemas a solventar fue la manera en la que gestionar los obstáculos del sensor Kinect.

El gran número de puntos disponibles incrementa mucho el cálculo de los obstáculos si se analiza toda la nube directamente, por lo que es mejor recurrir a analizarla por partes.

Para hacer esta operación se probaron nodos de ROS que realizaban la conversión del dato tipo *PointCloud2* a tipo *LaserScan* definiendo parámetros como la altura y distancia., con el objetivo de el procesado de puntos sea más rápido.

El nodo ***PointCloud_to_LaserScan***² realiza el filtrado y la conversión de tipo de datos a partir del análisis de la nube de puntos basándose en la librería PCL [poi10]. Su funcionamiento es correcto y el coste computacional se reduce, sin embargo no se consigue la frecuencia adecuada para que los datos se actualicen a medida que el robot navega.

Otro de los nodos utilizados para este propósito es ***DepthImage_to_LaserScan***³.

En este caso su enfoque es diferente, ya que utilizan los píxeles de la imagen junto con la componente de profundidad de los puntos para analizar la nube, de tal modo que solo analizan los puntos correspondientes a un determinado rango de píxeles. Este nodo es mucho más eficiente en el cálculo pero no tiene en cuenta aspectos como la inclinación del sensor o la detección del suelo como obstáculo.

Las pruebas realizadas utilizando ofrecieron un resultado correcto, ya que el procesado de la nube de puntos es más rápido y permite realizar los cálculos a una frecuencia adecuada. Sin embargo esta estrategia requiere más de una instancia de estos nodos para poder detectar obstáculos bajos a corta (hasta 1.5 metros) y media distancia (hasta unos 3 metros) para que no se produzcan interferencias con el suelo.

De este modo, se crearon 3 instancias diferentes de este nodo: uno para obstáculos situados a una distancia mayor, un segundo para obstáculos a distancias medias, y un tercero para obstáculos a distancias cortas.

```

1 <launch>
2   <node pkg="depthimage_to_laserscan" type="depthimage_to_laserscan" name="depthimage_to_laserscan_low">
3     <remap from="image" to="/camera/depth/image_raw"/>
4     <remap from="camera_info" to="/camera/depth/camera_info"/>
5     <remap from="scan" to="camera/scan_depth_low"/>
6     <rosparam>
7       scan_height: 220
8       scan_time: 0.167
9       range_min: 0.45
10      range_max: 1.4
11    </rosparam>
12  </node>
13 </launch>
```

Código 7.1: *Launchfile* del nodo *DepthImage_to_LaserScan* para obstáculos bajos.
4

Una vez solventados los problemas anteriores se realizaron los ajustes de ambos costmaps mediante ensayos prueba error con el propio robot y con su modelo creado en el simulador *Gazebo*.

La configuración de los mismos se expone a continuación.

²http://wiki.ros.org/pointcloud_to_laserscan

³http://wiki.ros.org/depthimage_to_laserscan

⁴ Fuente: *pioneer_utils/sensors/kinect_to_laser_low.launch*

⁵ Fuente: *pioneer_utils/navigation/global_navigation/global_costmap_params.yaml*

```

1 global_costmap:
2   global_frame: /map
3   robot_base_frame: /base_link
4   update_frequency: 2.0
5   publish_frequency: 2.0
6   static_map: true
7   rolling_window: false
8   track_unknown_space: true
9   plugins:
10     - {name: static_layer,           type: "costmap_2d::StaticLayer"}
11     - {name: obstacle_layer_kinect, type: "costmap_2d::VoxelLayer"}
12     - {name: obstacle_layer_laser,  type: "costmap_2d::VoxelLayer"}
13     - {name: inflation_layer,      type: "costmap_2d::InflationLayer"}
14
15 obstacle_layer_kinect:
16   observation_sources: kinect_laser kinect_laser_low kinect_laser_long
17   kinect_laser: {sensor_frame: camera_link, data_type: LaserScan, topic: camera/scan_depth,
18                 marking: true, clearing: true, obstacle_range: 3.0, raytrace_range: 6.5, inf_is_valid:
19                 true}
20   kinect_laser_low: {sensor_frame: camera_link, data_type: LaserScan, topic: camera/
21                      scan_depth_low, marking: true, clearing: true, obstacle_range: 3.0, raytrace_range:
22                      6.5, inf_is_valid: true}
23   kinect_laser_long: {sensor_frame: camera_link, data_type: LaserScan, topic: camera/
24                      scan_depth_long, marking: true, clearing: true, obstacle_range: 3.0, raytrace_range:
25                      6.5, inf_is_valid: true}
26
27 obstacle_layer_laser:
28   observation_sources: sick_lms1xx
29   sick_lms1xx: {sensor_frame: laser, data_type: LaserScan, topic: scan, marking: true,
30                  clearing: true, obstacle_range: 5.0, raytrace_range: 10.5, inf_is_valid: true}
31
32 inflation_layer:
33   inflation_radius: 0.55
34   cost_scaling_factor: 4.0

```

Código 7.2: Configuración del *global_costmap*.

5

De especial interés la configuración de los parámetros de la capa *costmap_2d::InflationLayer* donde se ajusta el radio de "inflado" (*inflation_radius*) de los obstáculos así como un valor de escala en el cálculo del coste de cada celda (*cost_scaling_factor*). Esto determina en gran medida el cálculo de trayectoria global, permitiendo trayectorias más suaves y alejadas de los obstáculos 7.13.

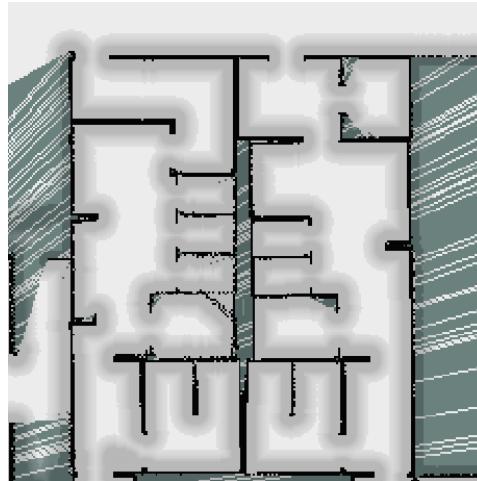


Figura 7.13: Visualizado del costmap global.

Para el mapa local la configuración es muy similar y lo más importante es una frecuencia de actualización del mapa mayor y la ausencia de la capa estática.

```

1 local_costmap:
2   global_frame: /odom
3   robot_base_frame: /base_link
4   update_frequency: 10.0
5   publish_frequency: 10.0
6   static_map: false
7   rolling_window: true
8   width: 6.0
9   height: 6.0
10  resolution: 0.05
11  max_obstacle_height: 0.5
12  plugins:
13    - {name: obstacle_layer_laser, type: "costmap_2d::VoxelLayer"}
14    - {name: obstacle_layer_kinect, type: "costmap_2d::VoxelLayer"}
15    - {name: inflation_layer, type: "costmap_2d::InflationLayer"}
16
17 obstacle_layer_kinect:
18   observation_sources: kinect_laser kinect_laser_low kinect_laser_long
19   kinect_laser: {sensor_frame: camera_link, data_type: LaserScan, topic: camera/scan_depth,
20     marking: true, clearing: true, obstacle_range: 9.0, raytrace_range: 9.5, inf_is_valid:
21       false}
22   kinect_laser_low: {sensor_frame: camera_link, data_type: LaserScan, topic: camera/
23     scan_depth_low, marking: true, clearing: true, obstacle_range: 9.0, raytrace_range:
24       9.5, inf_is_valid: false}
25   kinect_laser_long: {sensor_frame: camera_link, data_type: LaserScan, topic: camera/
26     scan_depth_long, marking: true, clearing: true, obstacle_range: 9.0, raytrace_range:
27       9.5, inf_is_valid: false}
28   kinect: {sensor_frame: camera_link, data_type: PointCloud2, topic: camera/depth/points,
29     marking: true, clearing: true, inf_is_valid: true}
30
31 obstacle_layer_laser:
32   observation_sources: sick_lms1xx
33   sick_lms1xx: {sensor_frame: laser, data_type: LaserScan, topic: scan, marking: true,
34     clearing: true, obstacle_range: 10.0, raytrace_range: 12.0, inf_is_valid: true}
35
36 inflation_layer:
37   inflation_radius: 0.55
38   cost_scaling_factor: 4.0

```

Código 7.3: Configuración del *local_costmap*.

⁶ Fuente: *pioneer_utils/navigation/common/local_costmap_params.yaml*

7.2.2 Configuración de los planificadores de trayectoria

Los planificadores de trayectoria por defecto en ROS utilizan algoritmos como Dijkstra o A* además de algunos ajustes para el cálculo y parámetros especiales en el caso del planificador de trayectoria local.

Parámetros del global_planner

En ROS, el nodo encargado de realizar el cálculo de la trayectoria global es el denominado `global_planner`. Este nodo dispone de los algoritmos de planificación ya implementados que realizan los cálculos de trayectoria.

La configuración del planificador global se realiza a través de parámetros que podemos configurar, distinguiendo entre el uso del algoritmo de Dijkstra o el de A*, utilizar un camino definido por rejilla, etc.

A continuación podemos ver el comportamiento del planificador con diferente configuración en sus parámetros.

- **Algoritmo A*:**

El planificador de trayectoria A* ofrece los siguientes resultados.



Figura 7.14: Planificador de trayectoria A* visualizado en RViz.

El espacio analizado es menor que en el caso de Dijkstra, sin embargo el camino final no es el adecuado, ya que es sinuoso y se acerca demasiado a obstáculos y paredes. Eso a simple vista parece un mal funcionamiento del algoritmo, ya que el comportamiento es exagerado.

- **Algoritmo de Dijkstra:**

La configuración por defecto del planeador realiza el cálculo de trayectoria mediante el algoritmo de Dijkstra.

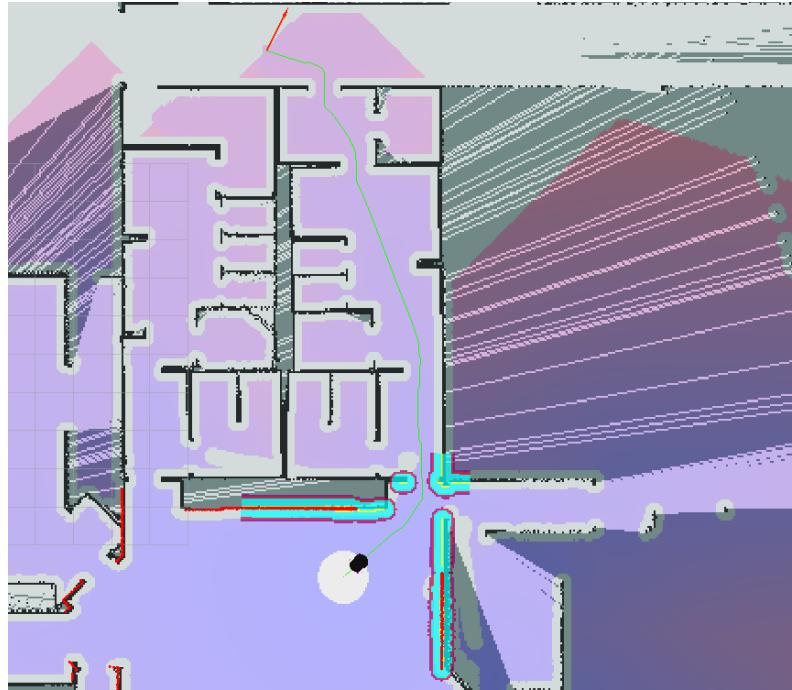


Figura 7.15: Planificador de trayectoria Dijkstra visualizado en RViz.

Como se ve en la imagen, el comportamiento de este planificador es más adecuado, trazando una trayectoria que se mantiene equidistante a los obstáculos y mucho más suavizada respecto a A* a pesar de su exploración mucho más masiva del espacio.

El algoritmo que se ha utilizado finalmente es el de Dijkstra debido a las pruebas anteriores y a su buen comportamiento en pruebas con el robot.

```

1 GlobalPlanner:
2   old_navfn_behavior: false
3   use_quadratic: true
4   use_dijkstra: true
5   use_grid_path: false
6
7   allow_unknown: true
8
9   planner_window_x: 0.0
10  planner_window_y: 0.0
11  default_tolerance: 0.1
12
13  publish_scale: 100
14  planner_costmap_publish_frequency: 0.0

```

Código 7.4: Configuración del *global_planner*.

7

Parámetros del local_planner

El planificador de trayectoria local que se ha utilizado es *Trajectory Rollout* por sus buenos resultados en robot con bajas capacidades de aceleración.

⁷ Fuente: *pioneer_utils/navigation/common/global_planner_params.yaml*

Las pruebas realizadas tanto en el simulador como en el propio robot han servido para ajustar los parámetros de velocidad y aceleración así como para mantener un compromiso entre la correcta reacción ante obstáculos locales y el ajuste a la trayectoria global.

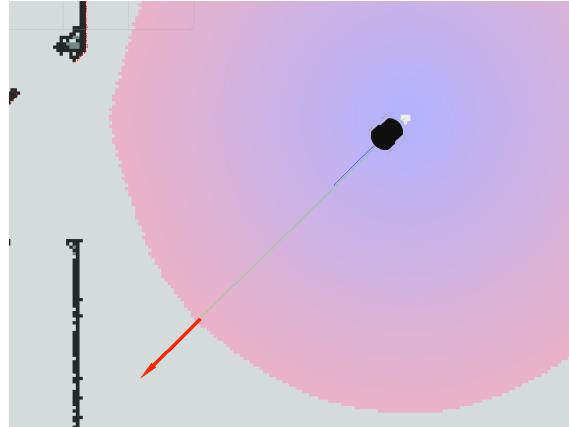


Figura 7.16: Planificador de trayectoria local *Trajectory Rollout* en RViz.

Los parámetros utilizados son los que se presentan a continuación.

```

1 TrajectoryPlannerROS:
2   max_vel_x: 0.6
3   min_vel_x: 0.1
4   max_vel_theta: 0.8
5   min_in_place_vel_theta: 0.4
6
7   acc_lim_theta: 3.2
8   acc_lim_x: 2.5
9   acc_lim_y: 2.5
10
11  holonomic_robot: false
12
13  yaw_goal_tolerance: 3.1415
14
15  sim_granularity: 0.025
16  sim_time: 2.0
17  meter_scoring: true
18  pdist_scale: 0.9
19  gdist_scale: 0.6

```

Código 7.5: Configuración de *base_local_planner*.

8

7.2.3 Navegación con mapa

Usualmente cuando hablamos de navegación nos referimos a una navegación basada en un mapa previo que se carga en la memoria del robot.

Los mapas utilizados para la navegación han sido todos creados utilizando el paquete *gmapping* de ROS, utilizando el sensor láser del robot para obtener un rango y precisión mayor (Figura 7.17).

⁸ Fuente: *pioneer_utils/navigation/common/base_local_planner_params.yaml*

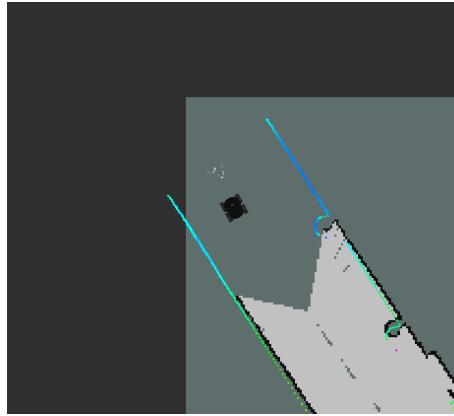


Figura 7.17: Creación de un mapa mediante SLAM.

Para realizar una navegación con mapa se utiliza un mapa del tipo anterior cargado en memoria acompañado del ya mencionado *global_costmap* de manera estática.

Esta es la configuración que se ha venido utilizando de manera general y el uso de todos sus elementos queda reflejado en el *launchfile* de navegación global.

```

1 <launch>
2   <!-- Run the map server -->
3   <node name="map_server" pkg="map_server" type="map_server" args="$(find pioneer_utils)/maps
4     /floor_zero-map.yaml"/>
5
6   <!-- Run AMCL -->
7   <include file="$(find pioneer_utils)/navigation/common/amcl.launch"/>
8
9     <node pkg="move_base" type="move_base" respawn="false" name="move_base" output="screen
10    ">
11      <rosparam file="$(find pioneer_utils)/navigation/common/costmap_common_params_p3at.yaml
12        " command="load" ns="global_costmap" />
13      <rosparam file="$(find pioneer_utils)/navigation/common/costmap_common_params_p3at.yaml
14        " command="load" ns="local_costmap" />
15      <rosparam file="$(find pioneer_utils)/navigation/common/local_costmap_params.yaml"
16        " command="load" />
17      <rosparam file="$(find pioneer_utils)/navigation/global_navigation/
18        global_costmap_params.yaml" command="load" />
19      <rosparam file="$(find pioneer_utils)/navigation/common/base_local_planner_params.yaml"
20        " command="load" />
21      <rosparam file="$(find pioneer_utils)/navigation/common/global_planner_params.yaml"
22        " command="load" />
23      <rosparam file="$(find pioneer_utils)/navigation/common/recovery_behaviors.yaml"
24        " command="load" />
25      <rosparam>
26        planner_frequency: 1.0
27      </rosparam>
28      <param name="base_global_planner" value="global_planner/GlobalPlanner"/>
29    </node>
30  </launch>

```

Código 7.6: Configuración de navegación global.

9

7.2.4 Navegación reactiva

La navegación reactiva es la que se conoce por carecer de un mapa previo cargado en la memoria del robot. En su caso el robot percibe el entorno a medida que navega construyendo un mapa global de manera dinámica al igual que sucede con la configuración del mapa local. En este caso la configuración del mapa global carece

⁹ Fuente: *pioneer_utils/navigation/global_navigation/global_navigation_p3at.launch*

de capa estática, por lo que el mapa se desplaza junto con el robot. Tampoco es preciso el nodo *amcl* para situar al robot en el mismo por lo que la orientación y posición del robot queda determinada por su odometría.

```

1 global_costmap:
2   global_frame: /odom
3   robot_base_frame: /base_link
4   update_frequency: 2.0
5   publish_frequency: 2.0
6   static_map: false
7   rolling_window: true
8   track_unknown_space: true
9   width: 15.0
10  height: 15.0
11  origin_x: 0.0
12  origin_y: 0.0
13
14  plugins:
15    - {name: obstacle_layer_kinect,           type: "costmap_2d::VoxelLayer"}
16    - {name: obstacle_layer_laser,            type: "costmap_2d::VoxelLayer"}
17    - {name: inflation_layer,                type: "costmap_2d::InflationLayer"}
18
19  obstacle_layer_kinect:
20    observation_sources: kinect_laser kinect_laser_low kinect_laser_long
21    kinect_laser: {sensor_frame: camera_link, data_type: LaserScan, topic: camera/scan_depth,
22      marking: true, clearing: true, obstacle_range: 3.0, raytrace_range: 6.5, inf_is_valid:
23      true}
24    kinect_laser_low: {sensor_frame: camera_link, data_type: LaserScan, topic: camera/
25      scan_depth_low, marking: true, clearing: true, obstacle_range: 3.0, raytrace_range:
26      6.5, inf_is_valid: true}
27    kinect_laser_long: {sensor_frame: camera_link, data_type: LaserScan, topic: camera/
28      scan_depth_long, marking: true, clearing: true, obstacle_range: 3.0, raytrace_range:
29      6.5, inf_is_valid: true}
30
31  obstacle_layer_laser:
32    observation_sources: sick_lms1xx
33    sick_lms1xx: {sensor_frame: laser, data_type: LaserScan, topic: scan, marking: true,
34      clearing: true, obstacle_range: 5.0, raytrace_range: 12.5, inf_is_valid: true}
35
36  inflation_layer:
37    inflation_radius: 0.55
38    cost_scaling_factor: 4.0

```

Código 7.7: Configuración de *global_costmap* para navegación reactiva.

10

7.3 Nodo de navegación por puntos

El nodo de navegación por puntos ofrece a posibilidad de enviar diferentes puntos de meta al robot tanto en el modo global como en el modo local. esto quiere decir, que ofrece unas funciones capaces de manda al robot a un punto deseado del mapa o hacer que este avance cierta distancia determinada.

El nodo de navegación por puntos *nav-waypoints* está desarrollado en C++ y ofrece un ejemplo de las posibilidades de uso de la navegación del robot. Este nodo se subscribe al estado del nodo de navegación del robot y publica puntos de meta en el *frame* apropiado dependiendo de si es una meta global o local.

Este nodo se encuentra separado del directorio habitual *pioneer_utils* debido a dependencias adicionales que pueden causar conflicto. Por ello, se aloja en el paquete *navigation_goals*.

¹⁰ Fuente: *pioneer_utils/navigation/local_navigation/global_costmap_params.yaml*

```

1 ...
2     bool setGlobalGoal(const float &x, const float &y, const float &angle)
3 {
4     //tell the action client that we want to spin a thread by default
5     MoveBaseClient ac("move_base", true);
6
7     //wait for the action server to come up
8     while(!ac.waitForServer(ros::Duration(5.0)))
9     {
10         ROS_INFO("Waiting for the move_base action server to come
11             up");
12     }
13
14     move_base_msgs::MoveBaseGoal goal;
15     goal.target_pose.header.frame_id = "/map";
16     goal.target_pose.header.stamp = ros::Time::now();
17
18     goal.target_pose.pose.position.x = x;
19     goal.target_pose.pose.position.y = y;
20     goal.target_pose.pose.orientation = tf::createQuaternionMsgFromYaw(
21         angle);
22     ROS_INFO("Sending GLOBAL goal");
23     ac.sendGoal(goal);
24
25     ac.waitForResult();
26
27     if(ac.getState() == actionlib::SimpleClientGoalState::SUCCEEDED)
28     {
29         ROS_INFO("Hooray, the base moved 1 meter forward");
30         return true;
31     }
32     else
33     {
34         ROS_INFO("The base failed to move forward 1 meter for some
35             reason");
36         return false;
37     }
38
39     int main(int argc, char** argv){
40         ros::init(argc, argv, "simple_navigation_goals");
41         setGlobalGoal(-0.671, 1.938, 1.0); //rosa
42         setGlobalGoal(0.193, -1.520, 1.0); //invernadero
43         setGlobalGoal(2.118, -8.223, 1.0); //comau
44         setGlobalGoal(-1.073, -9.271, 1.0); //puerta principal
45
46         return 0;
47     }

```

Código 7.8: Fragmento de código del nodo *nav-waypoints*.

7.3.1 Endurance test

El nodo *endurance_test* es un nodo de navegación similar al descrito en el apartado anterior pero desarrollado con la API de Python.

Este nodo sirve para realizar un test de resistencia en la navegación del robot con un mapa, mandando al robot diferentes puntos de meta y llevando un registro de las metas alcanzadas, el tiempo transcurrido y los metros recorridos.

Los puntos de meta son leídos desde un archivo de texto plano donde se indica el nombre del punto de meta y las coordenadas X, Y del mismo en el mapa utilizado. Adicionalmente y para dotarlo de utilidad a otros robots es posible configurar el Topic de la odometría, el Topic de velocidad y el tiempo de esperas entre metas.

endurance_test API		
Topics suscritos	Mensaje	Descripción
odom	nav_msgs/Odometry	Odometría del robot
Topics publicados		
cmd_vel	geometry_msgs/Twist	Publica comandos de velocidad
Parámetros		
rest_time	int	Tiempo de espera (segundos)
map_locations	file	Archivo con los puntos de meta
odometry_topic	string	Nombre del Topic de odometría
cmd_vel_topic	string	Nombre del Topic de velocidad

Tabla 7.1: API de endurance-test

Para lanzar este nodo se utiliza un archivo *launchfile* en el que se indican los parámetros.

```

1 <launch>
2   <node name="endurance_test" pkg="pioneer_utils" type="endurance_test.py"
3     output="screen">
4     <param name="map_locations" value="$(find pioneer_utils)/main/
5       locations_lab.txt"/>
6     <rosparam>
7       odometry_topic: rosaria/pose
8       cmd_vel_topic: cmd_vel
9       rest_time: 1
10      </rosparam>
11    </node>
12  </launch>

```

Código 7.9: Archivo *launchfile* para el nodo *endurance_test*.

7.4 Nodo de guiado (follower)

El nodo de guiado consiste en el análisis de la nube de puntos que capta el sensor Kinect para detectar un objeto delante y dirigir al robot ajustando sus velocidades para que mantenga la posición hacia ese objeto. Su funcionamiento se basa en el procesamiento de la nube de puntos obtenida a través de los nodos del paquete *freenect_stack*.

El nodo está originalmente desarrollado para el robot Turtlebot pero es fácil adaptable a otros robots con el propósito de que el robot siga a personas, a otros robot o a objetos en movimiento.

Requiere un Topic de tipo *sensor_msgs/PointCloud2* al que suscribirse para leer la nube de puntos y un Topic de tipo *geometry_msgs/Twist* al que publicar los movimientos de giro, avance y retroceso.

turtlebot_follower API		
Topics suscritos	Mensaje	Descripción
camera/depth/points	sensor_msgs/PointCloud2	Recibe la nube de puntos
Topics publicados	Mensaje	Descripción
cmd_vel	geometry_msgs/Twist	Publica comandos de velocidad
Parámetros	Tipo	Descripción
min_y	double	Posición mínima de puntos en Y
max_y	double	Posición máxima de puntos en Y
min_x	double	Posición mínima de puntos en X
max_x	double	Posición máxima de puntos en X
max_z	double	Posición máxima de puntos en Y
goal_z	double	Distancia mantenida en el seguimiento
z_scale	double	Factor de escala velocidad trans.
x_scale	double	Factor de escala en velocidad de rot.
enabled	bool	Habilita los movimientos

Tabla 7.2: API de *turtlebot_follower*

El tratamiento de la nube de puntos se realiza con la librería PCL (PointCloud Library [poi10]) y su funcionamiento es el siguiente:

- 1.- Busca puntos dentro de los límites establecidos.
- 2.- Calcula las dimensiones de los puntos encontrados.
- 3.- Calcula el centroide del la zona destacada.
- 4.- Mueve el robot de manera acorde hasta que alcanza la distancia establecida.

```

1 <launch>
2
3 <!-- Load turtlebot follower into the 3d sensors nodelet manager to avoid pointcloud
4   serializing -->
5 <node pkg="nodelet" type="nodelet" name="turtlebot_follower" args="load turtlebot_follower/
6   TurtlebotFollower camera/camera_nodelet_manager">
7   <remap from="turtlebot_follower/cmd_vel" to="/cmd_vel"/>
8   <remap from="depth/points" to="camera/depth/points"/>
9   <param name="enabled" value="true" />
10  <param name="x_scale" value="10.0" />
11  <param name="z_scale" value="10.0" />
12  <param name="min_x" value="-0.35" />
13  <param name="max_x" value="0.35" />
14  <param name="min_y" value="0.1" />
15  <param name="max_y" value="0.5" />
16  <param name="max_z" value="1.2" />
17  <param name="goal_z" value="0.6" />
</node>
</launch>
```

Código 7.10: Launchfile para *turtlebot_follower* en el robot Pioneer 3 AT.

7.5 Feedback mediante text-to-speech

Como ya se ha visto, la variedad de paquetes de software robótico en ROS es notable y ofrece una amplia variedad de características gracias a los aportes de la comunidad. A medida que evolucionaba este proyecto y debido a que el robot lleva incorporado su propio altavoz, apareció la idea de dotar de sonidos al robot de tal manera que existiera un feedback hacia las personas que se encuentren en su entorno.

Existe un nodo en ROS llamado *sound_play* [GH11] que permite, mediante la publicación de mensajes, reproducir sonidos preincorporados, archivos de sonido OGG/WAV o incluso realizar síntesis de voz a partir de un texto, conocido como *text-to-speech* (TTS), utilizando voces del *Festival Speech Synthesis System* desarrollado por *The Centre for Speech Technology Research* de la Universidad de Edinburgo [The14].

En este caso se ha utilizado la API de Python para interactuar con el nodo de tal modo que tan solo es necesario intercambiar mensajes con el nodo *soundplay_node*.

<i>soundplay_node</i> API		
Topics suscritos	Mensaje	Descripción
robotsound	sound_play/SoundRequest	Sonido a reproducir

Tabla 7.3: API del nodo *soundplay_node*.

Para ponerlo en marcha se inicia el nodo desde el archivo de *launch*.

```
1 | <node name="sound_play" pkg="sound_play" type="soundplay_node.py"/>
```

Código 7.11: Nodo *soundplay_node* para reproducir sonidos.

Para reproducir sonidos se utiliza un "handle" de sonido que facilita el paso de mensajes en caso de reproducir sonidos o voz con las funciones *.play()* y *.say()* respectivamente. A continuación se presenta un fragmento del nodo *voice_cmd_vel* de desarrollo propio y que se describirá más adelante

El uso y aplicación del nodo de sonido se expone con más detalle en la sección 7.7.

7.6 Reconocimiento de comandos de voz

Siguiendo con la idea de utilizar todos los elementos que incorpora el robot y de manera adicional a los objetivos del proyecto, se exploró la idea de la interacción con el robot mediante comandos de voz, gracias una vez más a los paquetes de ROS.

pocketsphinx [Fer12] es un paquete que actúa como *wrapper* del motor de reconocimiento de voz del mismo nombre, que utiliza el framework multimedia *GStreamer* [Wal01]. La implementación de este nodo está basada en los estudios e investigaciones sobre reconocimiento de voz y patrones en el habla de la Universidad de Carnegie Mellon dentro del proyecto CMU Sphinx [Car90].

EL objetivo del paquete *pocketsphinx*, mediante el nodo *recognizer*, es realizar el reconocimiento y procesado de voz, comparar la voz con un diccionario de palabras

```

1   from sound_play.libsoundplay import SoundClient
2
3   # Create the sound client object
4   self.soundhandle = SoundClient()
5
6   rospy.sleep(1)
7   self.soundhandle.stopAll()
8
9   # Subscribe to the move_base action server
10  self.move_base = actionlib.SimpleActionClient("move_base", MoveBaseAction)
11
12  rospy.loginfo("Waiting for move_base action server...")
13  self.soundhandle.play(1)
14
15  # Wait 60 seconds for the action server to become available
16  self.move_base.wait_for_server(rospy.Duration(60))
17
18  rospy.loginfo("Connected to move base server")
19
20  # Announce that we are ready for input
21  rospy.sleep(1)
22  self.soundhandle.say('Hi, my name is Petrois')
23  rospy.sleep(2)
24  self.soundhandle.say("Say one of the navigation commands")

```

Código 7.12: Fragmento del nodo *voice_cmd* utilizando el cliente de sonido de *soundplay_node*.

y pronunciación y devolver una cadena de texto con las palabras que han sido detectadas [Goe15].

recognizer API			
Topics publicados	Mensaje	Descripción	
Parámetros	std_msgs/String	Palabras detectadas	
	Tipo	Descripción	
output	string	Archivo de pronunciación	
lm	string	Diccionario de palabras	
dict	string		

Tabla 7.4: API del nodo *recognizer*.

La entrada de audio se realiza a través de las entradas de micrófono configuradas en Ubuntu. En este caso, para sacar mayor partido al sensor Kinect, se pretende utilizar el array de micrófonos que incorpora. La entrada de audio del sensor Kinect no se reconoce por defecto en Ubuntu, ni siquiera a través de los drivers *libfreenect* por lo que es necesario instalar el paquete *kinect-audio-setup*.

```

1 sudo apt-get install kinect-audio-setup

```

Código 7.13: Paquete necesario para reconocer el micrófono del sensor Kinect.

Una vez instalado se selecciona el micrófono del sensor en el panel de ajustes de audio de Ubuntu. El resto de instalaciones necesarias para el paquete *pocketsphinx* se detallan en el apéndice A.1.2.

Para utilizar el nodo *recognizer* deben indicarse los parámetros del siguiente modo:

Para realizar acciones con los comandos de voz se utiliza un nodo alternativo que se suscribe al Topic *recognizer/output*, el cual devuelve las palabras reconocidas. A continuación se expone un fragmento de su uso en Python:

```

1 <launch>
2   <node name="recognizer" pkg="pocketsphinx" type="recognizer.py">
3     <param name="lm" value="$(find pioneer_utils)/voice_audio/dic/commands.lm"/>
4     <param name="dict" value="$(find pioneer_utils)/voice_audio/dic/commands.dic"/>
5   </node>
6 <\launch>

```

Código 7.14: Archivo *launchfile* para el nodo *recognizer*.

```

1 ...
2   rospy.Subscriber('recognizer/output', String, self.speechCb)
3
4   def speechCb(self, msg):
5     rospy.loginfo(msg.data)
6
7     if msg.data.find("fast") > -1:
8       if self.speed != 0.3:
9         self.soundhandle.say('Speeding up')
10        self.set_speed(0.3)
11 ...

```

Código 7.15: Fragmento del nodo *voide_cmd* utilizando las palabras reconocidas por el nodo *recognizer*.

El uso y aplicación del nodo de reconocimiento de voz se expone con más detalle en la sección 7.7.

7.6.1 Crear una lista de vocabulario

Como ya se ha mostrado, este nodo precisa de un archivo de pronunciación y un archivo de diccionario. Estos archivos actúan como base de datos de vocabulario del motor de reconocimiento de voz.

Para crear estos archivos de vocabulario y pronunciación se utiliza la herramienta *Sphinx Knowledge Base Tool*¹² que genera archivos de pronunciación gracias a la herramienta *lmtool*¹³. Una de las limitaciones de la herramienta es que su desarrollo está basado principalmente en el reconocimiento de palabras en Inglés, y aunque se pueden incluir palabras de otros lenguajes su pronunciación es probable que no sea la correcta.

Accediendo a la herramienta en web (Figura 7.18), tenemos la posibilidad de subir un archivo con las palabras que queramos que nuestro sistema reconozca. En este caso hemos utilizado palabras en Inglés. La herramienta nos genera los archivos necesarios *.lm* y *.dic*.

¹²<http://www.speech.cs.cmu.edu/tools/lmtool-new.html>

¹³<http://www.speech.cs.cmu.edu/tools/lmtool.html>



Figura 7.18: Herramienta web Sphinx Knowledge Base Tool.

7.7 Nodo de ejecución automática de nodos

Para dar integración a todas las funcionalidades de ROS incorporadas en el robot de este proyecto se ha implementado un nodo que ejecuta o detiene cada uno de los nodos dependiendo de las tareas que se envíen al robot mediante comandos de voz.

El nombre de este nodo es *voice_cmd_vel*. Es un nodo desarrollado en Python debido a su fácil utilización de los nodos *recognizer* (para reconocer comandos de voz) y del nodo *soundplay_node* (para reproducir sonidos).

Las funcionalidades que integra son las siguientes:

- Reconocimiento de comandos de voz con el micrófono del sensor Kinect.
- Respuesta con voz sintetizada a través del altavoz.
- Capacidad para realizar movimientos de desplazamiento básico: forward, backward, right, left, stop.
- Indicar al robot que active el nodo de guiado (follower).
- Generar mapas mediante SLAM y guardarlos en memoria.
- Navegar hacia puntos del mapa preestablecidos (navegación con mapa).

Este nodo es un desarrollo específico y por tanto se encuentra muy acoplado a las configuraciones que se han realizado durante este trabajo.

La API de *voice_cmd_vel* (Tabla 7.5) sirve para conectarse a los nodos de reconocimiento de voz, al nodo de navegación y al nodo de sonido. Además recibe los mismos parámetros que el nodo *endurance_test* ya que incorpora su funcionalidad para leer puntos guardados del mapa para después utilizarlos como objetivos de meta.

7.7.1 Ejecución de nodos

Para la ejecución y para automática de los nodos existe el paquete *roslaunch*¹⁴, sin embargo la API del mismo ofrece una funcionalidad limitada ya que no se pueden ejecutar archivos *launchfile* completos con todos los nodos y parámetros necesarios.

¹⁴<http://wiki.ros.org/roslaunch/API%20Usage>

<i>voice_cmd_vel</i> API		
Topics suscritos	Mensaje	Descripción
recognizer/output	std_msgs/String	Comandos de voz detectados
Topics publicados	Mensaje	Descripción
cmd_vel	geometry_msgs/Twist	Publica comandos de velocidad
move_base/goal	move_base_msgs/MoveBaseActionGoal	Punto de meta
robotsound	sound_play/SoundRequest	Sonido a reproducir
Parámetros	Tipo	Descripción
map_locations	file	Archivo con los puntos de meta
cmd_vel_topic	string	Nombre del Topic de velocidad

Tabla 7.5: API del nodo *voice_cmd_vel*

Por tanto, se ha recurrido a los la funcionalidad *Subprocess* de Python, que permite ejecutar subprocessos con determinados comandos de consolas. Con esta forma de proceder podemos ejecutar archivos *launch* completos y después para los utilizando directamente los comandos de ROS `rosnode kill`".

```

1 subprocess.Popen(['roslaunch', 'pioneer_utils', 'simple-follower.launch'])
2 ...
3 subprocess.Popen(['rosnode', 'kill', 'turtlebot_follower'])

```

Código 7.16: Ejemplo de uso de *subprocess* en Python lanzando y parando el nodo *turtlebot_follower*.

La funcionalidad subprocess genera procesos por detrás, que sería el equivalente a abrir manualmente una terminal y lanzar el *launchfile*. Este proceso se mantiene abierto hasta que se ejecuta `rosnode kill`", que abre un proceso que se encarga de parar el proceso anterior y después de hacerlo muere automáticamente.

Para que este nodo funcione correctamente requiere que los siguientes nodos se ejecuten al inicio del mismo:

- Nodo de reconocimiento de voz: *recognizer*.
- Nodo de reproducción de sonidos: *soundplay_node*.
- Nodos de navegación: *move_base*, *amcl* y *map_server*.
- Nodos de bajo nivel: *RosAria*, *LMS1xx*, *freenect_launch* y *depthimage_to_laserscan*.

Adicionalmente, controla algunos nodos que no se ejecutan a inicio:

- *turtlebot_follower*: Para seguir a personas.
- *slam_gmapping*: Para capturar mapas mediante SLAM.
- *map_saver*: Para guardar mapas en memoria.

7.7.2 Funcionamiento

El funcionamiento del nodo *voice_cmd_vel* es el siguiente:

1. Se conecta al Topic de velocidad (*cmd_vel*) y lee los puntos de meta del archivo indicado en *map_locations*.
2. Se conecta al Topic *move_base/goal* para poder publicar puntos de meta.

3. Se conecta al Topic *robotsound* para enviar comandos de voz y sonidos.
4. Se suscribe al Topic *recognizer/output* para leer las palabras del nodo de reconocimiento de voz.
5. Entra en un bucle esperando algún comando de voz reconocido.
6. En caso de reconocer alguno de los comandos, ejecuta la acción y responde mediante voz o un sonido.

Los comandos de voz que puede reconocer el robot son los siguientes:

- Comandos de velocidad "fast", "half", "slow": Cambia la velocidad de desplazamiento del robot.
- Comandos de movimiento básico "forward", "back", "right", "left", "stop": El robot se mueve de la forma indicada.
- Comandos de guiado "follow me", "stop follower": Inicia o detiene el nodo de guiado (follower).
- Comandos de mapa "build map", "save map", "stop map": Construye un mapa mediante SLAM, guarda el mapa o detiene el SLAM.
- Comandos de navegación: "Navigate to..." seguido del nombre de un punto de meta guardado en el archivo *map_locations*: Indica al robot que navegue hasta el punto deseado.

El archivo *voice_cmd.launch* ejecuta el inicio del nodo de reconocimiento de voz, el nodo de reproducción de sonido y el nodo *voice_cmd_vel* con sus respectivos parámetros.

```

1 <launch>
2   <node name="recognizer" pkg="pocketsphinx" type="recognizer.py">
3     <param name="lm" value="$(find pioneer_utils)/voice_audio/dic/commands.lm"/>
4     <param name="dict" value="$(find pioneer_utils)/voice_audio/dic/commands.dic"/>
5   </node>
6
7   <node name="sound_play" pkg="sound_play" type="soundplay_node.py"/>
8
9   <node name="voice_cmd_vel" pkg="pioneer_utils" type="voice_cmd_vel.py" output="screen">
10    <param name="map_locations" value="$(find pioneer_utils)/main/locations.txt"/>
11    <rosparam>
12      cmd_vel_topic: cmd_vel
13    </rosparam>
14  </node>
15 </launch>
```

Código 7.17: Archivo *voice_cmd.launch*.

15

El código fuente de este nodo se ha omitido por no alargar la explicación, sin embargo puede consultarse de forma externa¹⁶.

¹⁵ Fuente: *pioneer_utils/voice_audio/launch/voice_cmd.launch*

¹⁶ Fuente: *pioneer_utils/voice_audio/node/voice_cmd_vel.py*

Capítulo 8

Simulación del sistema

En este capítulo se recoge el apartado de simulación de este proyecto, para el que se han utilizado los simuladores MobileSim en las pruebas preliminares con la plataforma Pioneer 3 AT y Gazebo para realizar pruebas y optimizaciones del sistema robótico completo gracias a sus amplias capacidades y su integración con ROS.

8.1 Simulación con MobileSim

Una de las ventajas de utilizar el nodo rosaria para controlar el robot y obtener los datos de odometría es que las herramientas ofrecidas por el fabricante siguen pudiendo utilizarse. Este es el caso del simulador MobileSim.

MobileSim es un simulador robótico en dos dimensiones creado para los robots de Adpet Mobile Robots que puede utilizarse con robots controlados mediante la librería Aria.

Su funcionamiento es el siguiente: El simulador abre un puerto de comunicación local en el ordenador y al ejecutar la conexión con el robot de Aria, si el robot no se encuentra se procede a conectarse a dicho puerto de comunicación. Esto nos permite utilizar MobileSim con RosAria de la misma manera y sin cambiar nuestra configuración.

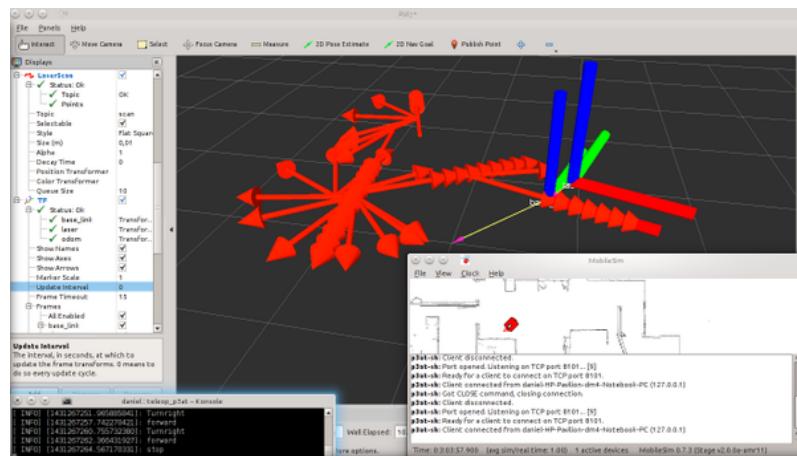


Figura 8.1: MobileSim junto con RViz funcionando con teleoperación.

Las pruebas con este simulador sirvieron para comprobar que el nodo rosaria disponía de la funcionalidad adecuada, así como para realizar pruebas con el nodo

de teleoperación y dead reckoning (Figura 8.1).

Por otro lado, las limitaciones de este simulador son evidentes. No existe posibilidad de simular otros sensores incorporados al robot, la integración de ROS se realiza con una librería intermedia y la más importante, no puede simularse un entorno en tres dimensiones.

8.2 Simulación con Gazebo

El simulador Gazebo permite realizar configuraciones más elaboradas y su integración con ROS es completa a pesar de que existen ciertas dificultades en su configuración.

En este entorno podemos simular el robot Pioneer 3 AT y dotarle de los sensores necesarios mediante plug-ins desarrollados que simulan los diferentes sensores y actuadores.

8.2.1 Modelado del robot en el simulador

Gazebo es capaz de simular robots definidos mediante archivos URDF, donde se indica cada tipo articulación y eslabón del robot. Además existen archivos de propiedades de los materiales y características específicas de gazebo que se definen a parte.

A continuación se hace un breve descripción de los archivos utilizados y su función¹:

- **pioneer3at.xacro:** Es el archivo principal donde se define el modelo del robot mediante URDF y se llama a los plug-ins de los sensores y actuadores y los demás archivos xacro.
- **materials.xacro:** Se definen propiedades de los materiales de cada parte del robot.
- **pioneer3at_wheels.xacro:** Macro para definir las ruedas del robot.
- **pioneer3at_pilars.xacro:** Macro para definir las barras de sujeción del robot.
- **pioneer3at_gazebo:** Define funcionalidades adicionales de gazebo. Aquí se definen los plug-ins a utilizar como *skid_steer_drive_controller* para el movimiento del robot o *kinect_camera_controller* para simular el sensor Kinect.
- **pioneer3at.world:** Es un archivo específico del simulador que guarda una descripción del mundo virtual. En este caso se trata de un mundo sin objetos.
- **pioneer3at_gazebo_world.launch:** Archivo principal para lanzar el simulador. En él se lanza el mundo virtual (configurado para utilizar *willowgarage_world*), el modelo del robot creado y el nodo *robot_state_publisher* que realiza la publicación de transformadas entre los diferentes ejes de coordenadas en base al modelo de nuestro robot.

¹ La configuración de Gazebo puede encontrarse en: *pioneer_utils/gazebo*

La definición URDF es la más importante ya que configura los parámetros del robot como el peso de cada parte, el material, el momento de inercia...

A continuación se muestra un fragmento de *pioneer3at.xacro* donde se describe el elemento que conforma la parte superior del robot y su “articulación estática” (fixed) con el chasis:

```

1      <!-- Top -->
2      <link name="top_plate">
3          <inertial>
4              <mass value="0.1"/>
5              <origin xyz="-0.025 0 -0.223"/>
6              <inertia ixx="1.0" ixy="0.0" ixz="0.0"
7                  iyy="1.0" iyz="0.0"
8                  izz="1.0"/>
9          </inertial>
10         <visual name="top_visual">
11             <origin xyz="0 0 0" rpy="0 0 0"/>
12             <geometry name="pioneer_geom">
13                 <mesh filename="package://p2os_urdf/meshes/p3at_meshes/top.stl
14                     "/>
15             </geometry>
16             <material name="TopBlack">
17                 <color rgba="0.038 0.038 0.038 1.0"/>
18             </material>
19         </visual>
20         <collision>
21             <origin xyz="0 0 0" rpy="0 0 0"/>
22             <geometry>
23                 <box size="0 0 0"/>
24             </geometry>
25         </collision>
26     </link>
27
28     <joint name="base_top_joint" type="fixed">
29         <origin xyz="0.003 0 0.274" rpy="0 0 0"/>
30         <parent link="base_link"/>
31         <child link="top_plate"/>
32     </joint>

```

Código 8.1: Fragmento de la configuración URDF del robot.

²

La parte gráfica recibe en los elementos ”mesh”³: Objetos definidos por mallas en formato .stl generados con un programa de modelado en tres dimensiones. Con las dimensiones de esos archivos también podemos simplificar el modelado de colisión para cada pieza.

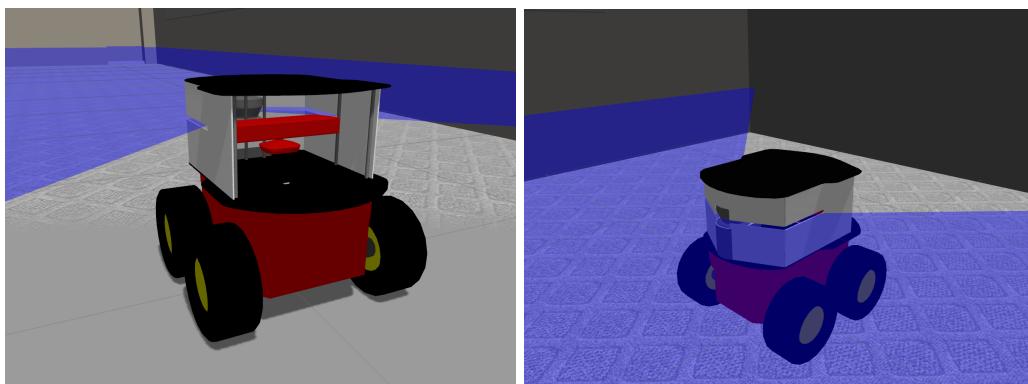


Figura 8.2: Modelo URDF visualizado en *Gazebo*.

² Fuente: *pioneer_utils/gazebo/pioneer3at.xacro*

³ Los diseños 3D se encuentran en: *pioneer_utils/gazebo/meshes*

8.2.2 Funcionamiento de Gazebo

Como se ha descrito anteriormente, Gazebo permite simular el sistema robótico completo de tal manera que podemos utilizarlo para probar desarrollos de nodos ROS para luego incorporarlos al robot real.

Para lanzar los nodos de Gazebo es necesario disponer de un modelo del robot configurado mediante URDF así como los plug-ins necesarios para simular el comportamiento del mismo y generar sensores virtuales que captarán el entorno del mundo simulado.

La prueba en marcha de Gazebo se realiza a partir de un nuevo archivo *launchfile*. En él se carga la definición URDF del robot junto con el mundo virtual y nodos auxiliares para simular el robot e iniciar Gazebo. Esto se recoge en el archivo *pioneer3at_gazebo_world.launch* mostrado a continuación.

```

1 <launch>
2
3     <!-- these are the arguments you can pass this launch file, for example paused:=true -->
4     <arg name="paused" default="false"/>
5     <arg name="use_sim_time" default="true"/>
6     <arg name="gui" default="true"/>
7     <arg name="headless" default="false"/>
8     <arg name="debug" default="false"/>
9
10    <!-- We resume the logic in empty_world.launch, changing only the name of the world to be
11        launched -->
12    <include file="$(find gazebo_ros)/launch/willowgarage_world.launch">
13    </include>
14
15    <!-- Load the URDF into the ROS Parameter Server -->
16    <param name="robot_description"
17           command="$(find xacro)/xacro.py $(find pioneer_utils)/gazebo/pioneer3at.xacro" />
18
19    <!-- Run a python script to send a service call to gazebo_ros to spawn a URDF robot -->
20    <node name="urdf_spawner" pkg="gazebo_ros" type="spawn_model" respawn="false" output="screen"
21        args="-urdf -model pioneer3at -param robot_description"/>
22
23    <!-- publish all the frames to TF -->
24    <node name="robot_state_publisher" pkg="robot_state_publisher" type="state_publisher">
25        <param name="publish_frequency" value="50"/> <!-- Hz -->
26    </node>
27
28    <!-- Launch kinect and depthimage_to_laser node -->
29    <include file="$(find pioneer_utils)/sensors/kinect_to_laser_low.launch"/>
30
31    <!-- Launch kinect and depthimage_to_laser node -->
32    <include file="$(find pioneer_utils)/sensors/kinect_to_laser_long.launch"/>
33
34    <!-- Launch kinect and depthimage_to_laser node -->
35    <include file="$(find pioneer_utils)/sensors/kinect_to_laser.launch"/>
36
37 </launch>
```

Código 8.2: Launchfile para lanzar Gazebo con el modelo del robot y sus sensores.

4

Como se ve, en el *launchfile* no es necesario indicar las transformadas entre los diferentes sistemas de referencia debido a que esta información ya la aporta el modelo URDF del robot. Tan solo es necesario llamar al nodo *state_publisher* e indicar la frecuencia de conversión.

También puede apreciarse los nodos intermedios *depthimage_to_laser* que se incluyen y funcionan de la misma manera que en el robot real. Ejemplo de ello es la figura 8.3, donde se observa la nube de puntos virtual junto con el haz láser generado por el nodo *depthimage_to_laserscan*.

⁴ Fuente: *pioneer_utils/gazebo/pioneer3at_gazebo_world.launch*

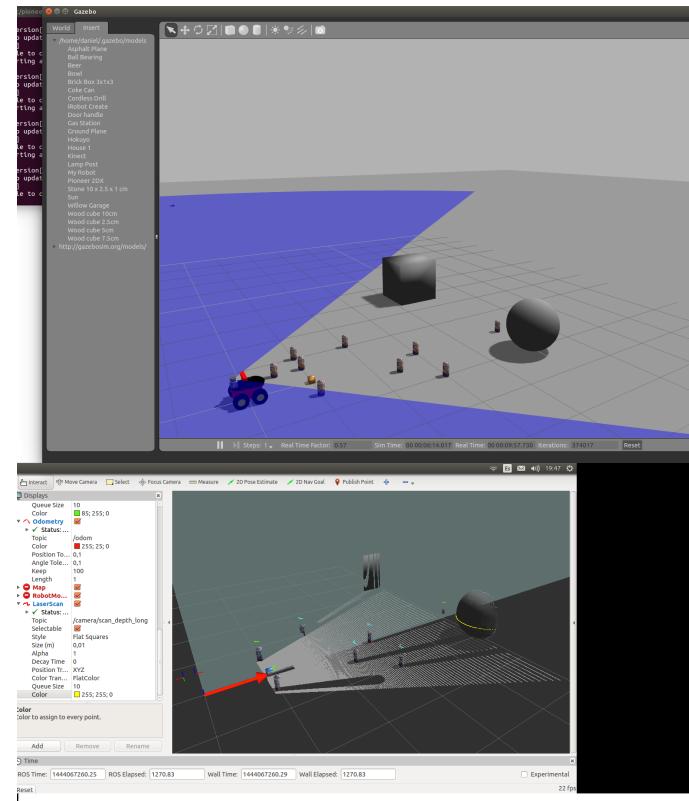


Figura 8.3: Simulación en Gazebo y visualizado de datos en RViz.

Para iniciar Gazebo con el modelo del robot tan solo es necesario lanzar el archivo *launch* anterior, tras lo cual se carga el modelo y se abre la interfaz gráfica del simulador (Figura 8.4).

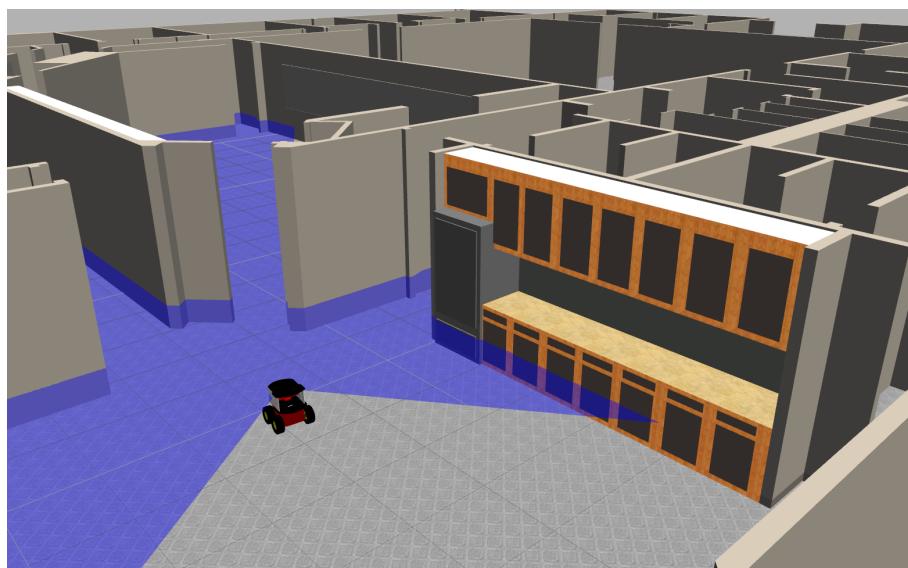


Figura 8.4: Pioneer 3 AT simulado en el mapa Willow Garage.

A partir de este punto pueden lanzarse cualquiera de los nodos que se han desarrollado en este proyecto (navegación global, teleopeación, comandos de voz, etc.),

de tal forma que se puede simular el sistema robótico completo.

En el caso de navegación global, se ha creado por comodidad un *launchfile* específico⁵ cuya única diferencia respecto de la navegación global real es que carga el mapa utilizado en el simulador "Willow Garage".

La ventaja de disponer del simulador Gazebo funcionando con el modelo de nuestro robot ha permitido que el robot evolucione más rápido ya que los pequeños ajustes se probaban de manera inmediata para validarlos y finalmente implementarlo en el robot real o no. Con la ayuda de Gazebo se han podido probar todas las configuraciones de navegación, el funcionamiento de cada nodo y el intercambio de datos entre ellos.

Por último indicar que la puesta en marcha del simulador Gazebo a supuesto cierto esfuerzo, ya que no existe una documentación unificada entre ROS y Gazebo debido al salto entre las versiones de uno y otro.

Se han seguido parte de los tutoriales de la web de Gazebo⁶ aunque algunas partes se encuentran incompletas, sobre todo en lo referente a los plug-ins de simulación de sensores y del robot.

También el modelo modelo URDF del robot se ha creado mediante la recopilación de información variada y en gran medida reutiliza el modelo incluido en el paquete *p2os* [All11] incorporando macros nuevas para los nuevos componentes como los paneles laterales.

⁵ Fuente: *pioneer_utils/gazebo/avigation/gazebo-global-navigation_p3at.launch*

⁶<http://gazebosim.org/tutorials>

Capítulo 9

Pruebas del sistema

En este capítulo se describen las principales pruebas realizadas con el robot de tal modo que sirva para validar el trabajo realizado y exponer a los resultados obtenidos.

Dividimos este capítulo en dos partes, pruebas del robot en simulación y pruebas en el entorno real, centrándonos exclusivamente en la creación de mapas mediante SLAM y los tests de navegación por ser el objeto principal de este proyecto.

9.1 Pruebas simuladas

En este apartado se describen las pruebas llevadas a cabo en el simulador Gazebo con el modelo del robot descrito anteriormente. Cabe indicar que todas las pruebas se realizan con la misma configuración que en el caso del robot real a excepción de los nodos de bajo nivel.

9.1.1 SLAM

Las pruebas de SLAM se realizaron con el ya mencionado nodo *slam_gmapping*, funcionando de la manera habitual.

Para comprobar su validez, se dispuso al robot en el mundo "Willow Garage" que simula un entorno de oficinas, similar al entorno real del laboratorio. El objetivo principal fue comprobar que el algoritmo de mapeado funcionaba correctamente y que se conseguía cerrar el mapa realizando un bucle completo.

Para esta prueba se utilizó el nodo de mapeado y el de teleoperación y se guió al robot por una parte del entorno realizando diversos bucles (Figura 9.1), en los que se observa que no existe distorsión en paredes paralelas y que el mapa no se encuentra solapado.

Indicar que este fue el mapa base utilizado para la navegación de sucesivas pruebas con Gazebo en las cuales el nodo AMCL situó correctamente al robot en el entorno virtual, por lo que validación del mapa generado es correcta.

9.1.2 Navegación con mapa

Las navegaciones con mapa en el simulador fueron determinantes para realizar una configuración más elaborada de ambos costmaps debido a los problemas con el borrado de los obstáculos (ver Subsección 7.2.1).

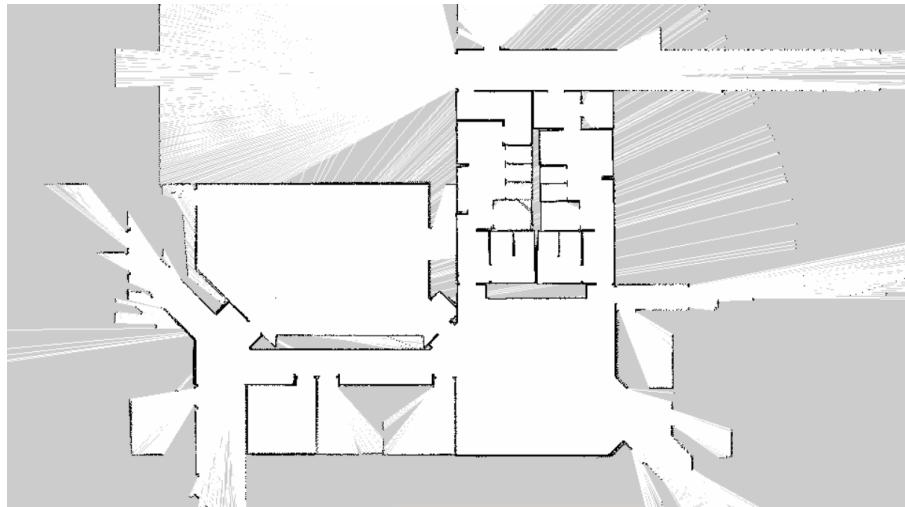


Figura 9.1: Prueba de SLAM en el simulador Gazebo.

En un principio se valoró la posibilidad de utilizar el sensor Kinect tan solo para obstáculos locales y el sensor láser para obstáculos globales, sin embargo esta idea se descartó ya que el robot trataba de seguir una trayectoria global incorrecta que no contemplaba los obstáculos bajos (Figura), con las complicaciones de control que eso supone para el planificador de trayectoria local.

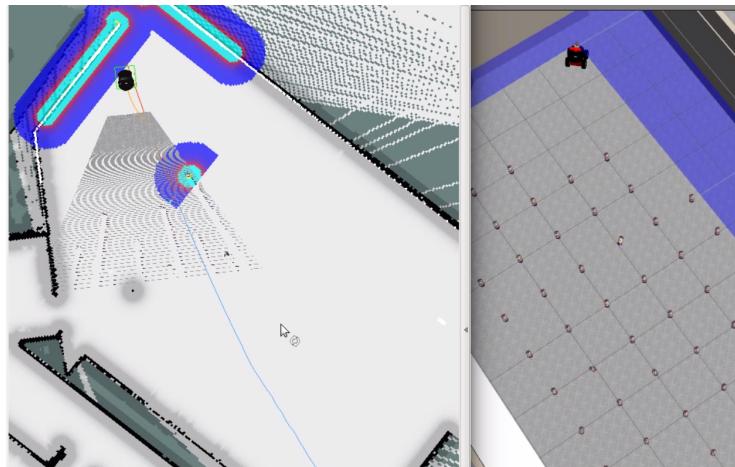


Figura 9.2: Trayectoria global erronea.

Esto determinó, junto con las características especiales del sensor Kinect, que los sensores fueran recolocados para la configuración final del robot y que se añadiesen capas de obstáculos diferenciadas para cada sensor.

Pruebas posteriores con la nueva configuración y obstáculos bajos determinaron la configuración correcta. Además se configuró un cálculo de trayectoria global repetitivo (parámetro *planner-frequency: 1.0*), de tal modo que el planificador global genera trayectorias actualizadas a medida que el robot se desplaza e incorpora nuevos obstáculos al mapa. Esto ayuda al planificador local de trayectoria y en definitiva a que el robot realice menos maniobras.



Figura 9.3: Navegación con mapa final.

9.1.3 Navegación reactiva

La navegación reactiva es una configuración menor de la anterior, por lo que no supuso un desarrollo más elaborado en las pruebas realizadas. Las pruebas consistieron en realizar navegación hacia varios puntos de meta y comprobar si el robot se quedaba atascado en algún momento.

Los resultados determinaron que el robot mostraba el mismo comportamiento en la navegación pero existían limitaciones como la distancia a la que pueden encontrarse los puntos de meta (puntos más cercanos al robot debido a la ausencia de mapa) o solapamientos y giros en el mapa global si este era demasiado grande.

Como es normal en una navegación reactiva, el robot se desplaza siguiendo la trayectoria más corta en base a la información que sus sensores captan en ese momento, por lo que suceden casos, como en la figura , en los que la trayectoria global calculada no es la correcta en un principio. A pesar de ello, gracias a la información de los obstáculos que queda retenida en el mapa global, el planificador recalcula encontrando la trayectoria adecuada.

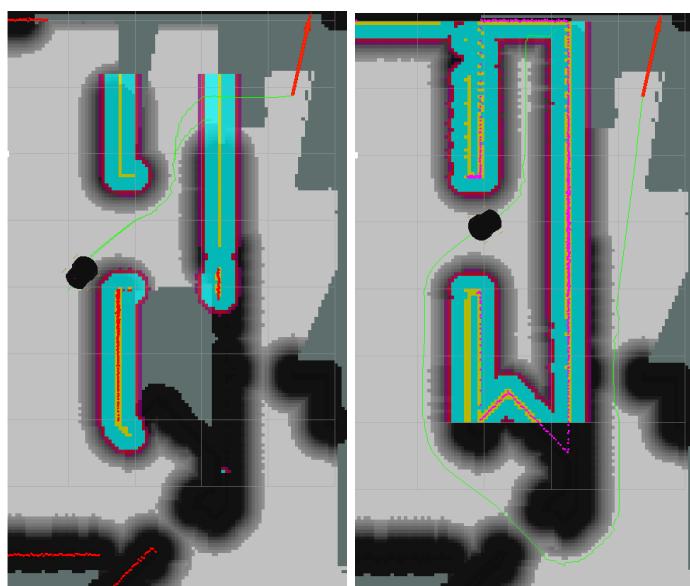


Figura 9.4: Navegación con mapa final.

9.1.4 Pruebas de resistencia

Para las pruebas de resistencia utilizamos un nodo *endurance_test*, específicamente creado para el caso.

En estas pruebas se consideraron las siguientes variables:

- Puntos de meta: Puntos totales repartidos por el mapa que servirán como puntos objetivo a alcanzar por el robot.
- Tiempo total: Tiempo total de la prueba.
- Metas enviadas: Número total de puntos de meta que se han enviado como objetivo al robot durante la prueba.
- Metas alcanzadas: Número de metas a las que ha conseguido llegar el robot.
- Tasa de metas alcanzadas: Tanto por ciento de las metas alcanzadas respecto de las enviadas.
- Distancia total: Distancia total recorrida, considerando obstáculos intermedios y maniobras o replanificación de trayectorias del robot.
- N° de choques: Número de obstáculos con los que el robot ha chocado (si los hubiera).

Las pruebas de resistencia se realizaron en el modo de navegación con mapa, el entorno virtual fue el mundo "Willow Garagez se utilizaron un total de 9 puntos de meta como objetivo, distribuidos de manera aleatoria (dentro de habitaciones, pasillos, salas grandes...) a distancias largas. Cabe señalar que no existe ningún tipo de obstáculo móvil implicado en estas pruebas.

Primera prueba

Prueba inicial sin obstáculos adicionales en el entorno.

Puntos de meta	9
Tiempo total	20 minutos
Metas enviadas	18
Metas alcanzadas	18
Tasa metas alcanzadas	100 %
Distancia total	275.2 metros
N° de choques	0

Tabla 9.1: Primera prueba de resistencia simulada.

La navegación del robot fue correcta durante toda la prueba.

Segunda prueba

Segunda prueba más larga y con obstáculos adicionales en el entorno.

Puntos de meta	9
Tiempo total	56 minutos
Metas enviadas	29
Metas alcanzadas	22
Tasa metas alcanzadas	75 %
Distancia total	354.6 metros
Nº de choques	0

Tabla 9.2: Segunda prueba de resistencia simulada.

La prueba finalizó debido a que el robot quedó inmovilizado por encontrarse demasiado cerca de una pared frontal y otra lateral. El incremento de tiempo a pesar de no haber recorrido muchos más metros respecto de la prueba anterior se debe a las esperas por reintento en alcanzar las metas fallidas.

A pesar de todo el robot no colisiona con ningún obstáculo.

Tercera prueba

Tercera prueba con las mismas características que la anterior. El objetivo de esta prueba era recorrer una distancia total más larga sin colisión.

Puntos de meta	9
Tiempo total	23 minutos
Metas enviadas	18
Metas alcanzadas	15
Tasa metas alcanzadas	83 %
Distancia total	488.8 metros
Nº de choques	0

Tabla 9.3: Tercera prueba de resistencia simulada.

Las metas no alcanzadas se produjeron por un exceso de tiempo en alcanzarlas, saltando el *timeout* y abortando la meta objetivo.

Los resultados de esta prueba indican que los planificadores responden bien en el cálculo de trayectorias hacia puntos de meta alejados.

9.2 Pruebas reales

Las pruebas reales se llevaron a cabo en el robot con su configuración hardware y configuración del sistema final. Todos los nodos se ejecutan en el ordenador integrado Intel NUC y se utiliza un ordenador cliente para visualizar datos o teleoperar el robot en caso necesario.

Las pruebas de navegación con y sin mapa se han omitido debido a que los resultados fueron correctos y no se observaron grandes diferencias respecto a los resultados obtenidos en el simulador.

9.2.1 SLAM

Las pruebas reales de SLAM se realizaron en la planta baja y el aparcamiento de la universidad, ya que era de los pocos lugares donde podía realizar un bucle completo.

Para realizar estas pruebas se utilizó al robot en modo de guiado y al mismo tiempo se realizó el mapeado. Indicar que no existen interferencias al realizar SLAM guiando de este modo al robot, ya que la persona que sirve de guía se encuentra en frente y solo es detectada por el sensor Kinect, de tal modo que el sensor láser se encarga de mapear el entorno sin perturbaciones.

En la figura 9.5 se muestra el resultado de la primera prueba.

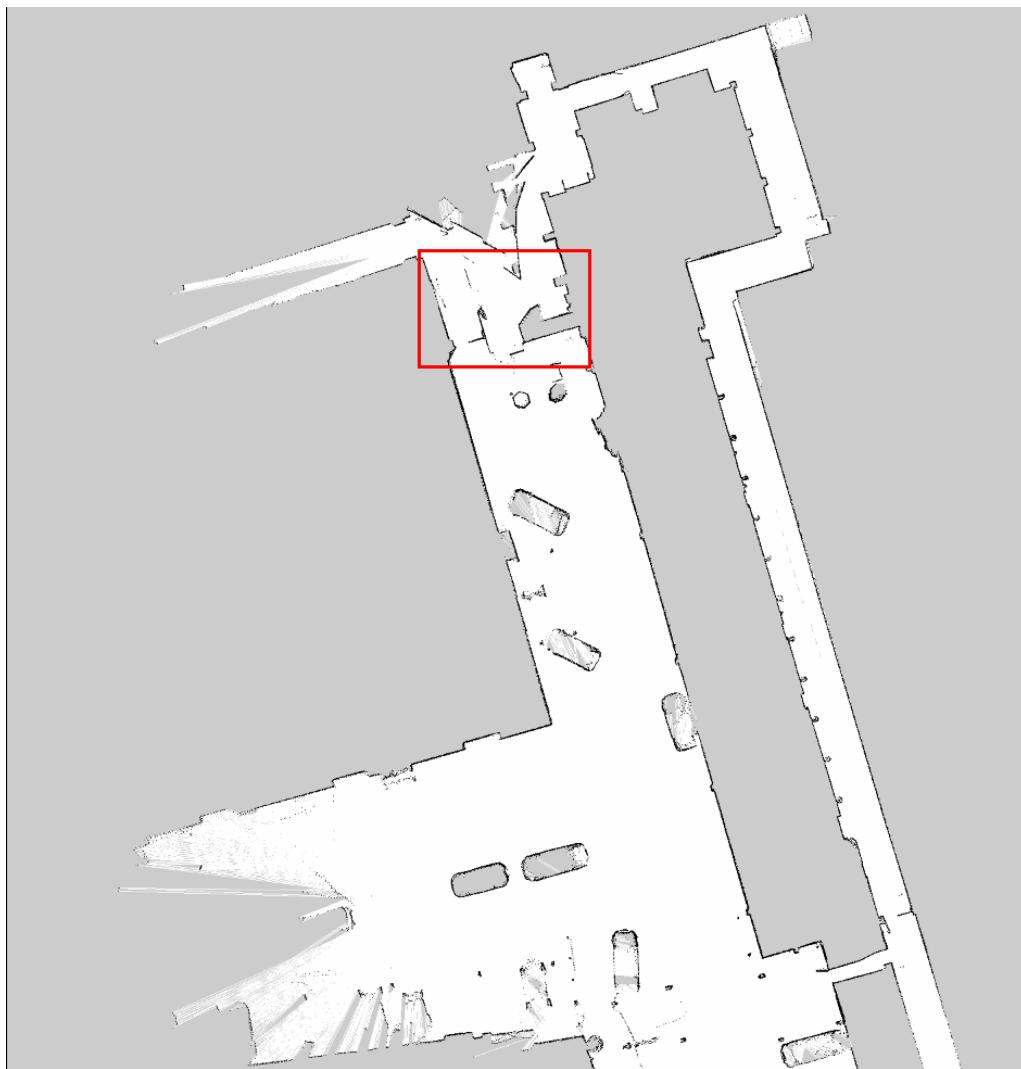


Figura 9.5: Mapa con la primera prueba de SLAM.

1

El área recuadrada en color rojo indica la zona de cierre del bucle, donde se aprecian ciertas discrepancias en las paredes y un ligero desvío hacia la derecha. Parte de este efecto es debido a que el bucle a cerrar es demasiado grande para esta prueba y a que el robot se desplazó realizando un recorrido sinuoso por el área del

¹ Fuente: *pioneer_utils/maps/slam_real1.pgm*

parking (se aprecia que se cubrió gran parte del mismo sin ser algo necesario para la prueba), con lo que el error de la odometría pudo haberse visto incrementado.

Debido a estos resultados se decidió realizar una segunda prueba de SLAM en las mismas condiciones pero guiando al robot de una manera más directa a lo largo del bucle (sin giros adicionales).

En la figura 9.6 puede apreciarse el resultado de la segunda prueba con el cierre del bucle en la zona recuadrada en rojo. Esta vez el resultado es mejor, ya que no existen grandes discrepancias y el robot es capaz de situarse sin problemas. Tan solo se aprecia una pared exterior duplicada en ángulo, fruto ligero desvío en el mapa.

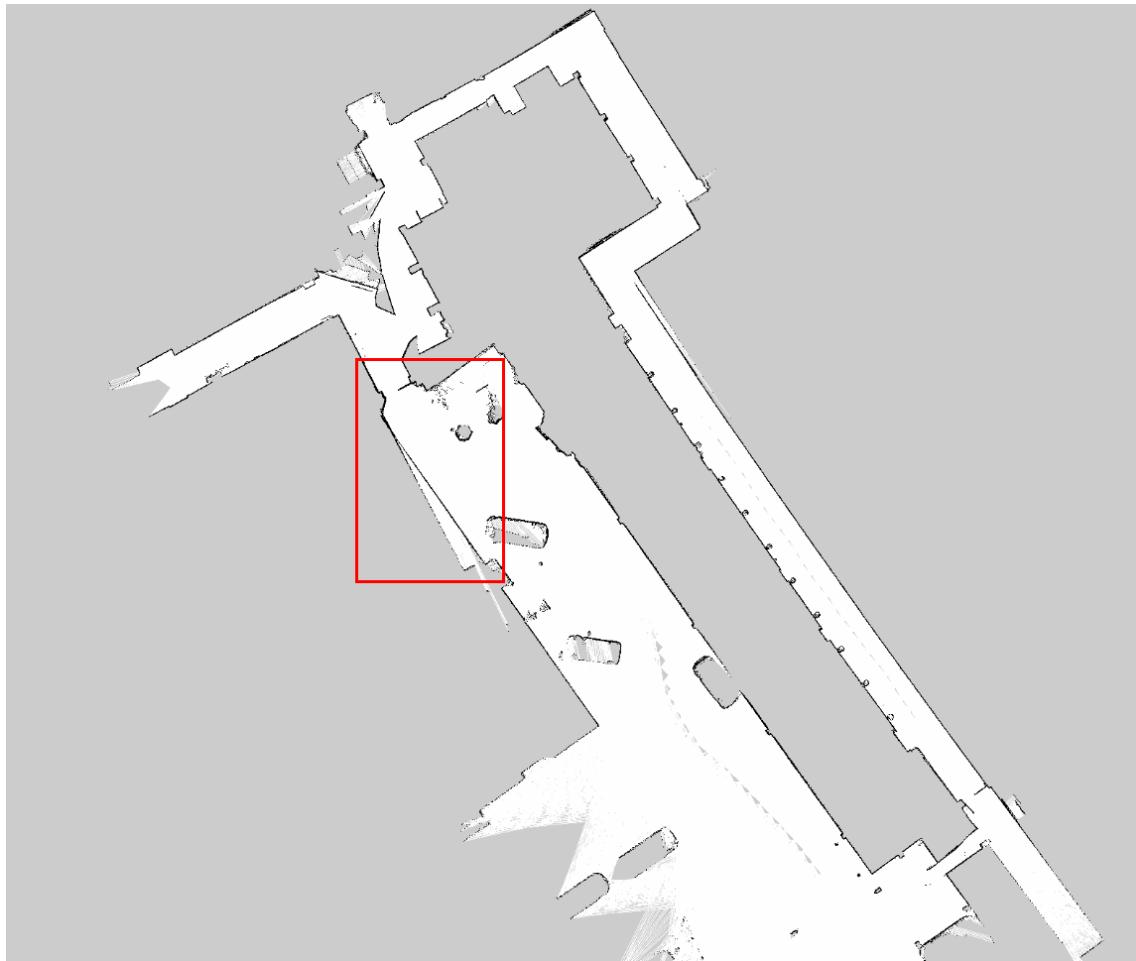


Figura 9.6: Mapa tras la segunda prueba de SLAM.

2

9.2.2 Pruebas de resistencia

Las pruebas de resistencia se llevaron a cabo en las instalaciones de la planta baja de la universidad mediante el nodo *endurance_test* creado para el caso y utilizando diversos puntos dentro del mapa "floor_zero-map"³ que abarca las zonas indicadas.

El objetivo primordial de las pruebas es que el robot sea capaz de navegar sin

² Fuente: *pioneer_utils/maps/slam_real1.pgm*

³ Fuente: *pioneer_utils/maps/floor_zero-map.pgm*

ningún tipo de intervención humana, ya que invalidaría el concepto de robot autónomo.

Primera prueba

Primera prueba real desarrollada en el laboratorio con obstáculos y personas en movimiento.

Puntos de meta	7
Tiempo total	44 minutos
Metas enviadas	91
Metas alcanzadas	56
Tasa metas alcanzadas	61 %
Distancia total	-
Nº de choques	0

Tabla 9.4: Primera prueba de resistencia real⁴.

Los resultados de la primera prueba no reflejan una navegación buena del robot, sin embargo esto es debido a la existencia de obstáculos prominentes no incluidos en el mapa del laboratorio que generaban distorsión en AMCL provocando que el robot quedase mal situado en el mapa. Estos obstáculos generaban a su vez un pasillo estrecho a través del cuál el planificador en ocasiones no era capaz de trazar una trayectoria correcta hasta la meta.

A pesar de ello el robot no colisionó con ningún tipo de obstáculo.

Segunda prueba

Segunda prueba de mayor recorrido, con obstáculos adicionales en el entorno y personas en movimiento. El objetivo de la prueba era medir la total movilidad del robot en el laboratorio.

Puntos de meta	7
Tiempo total	57 minutos
Metas enviadas	141
Metas alcanzadas	114
Tasa metas alcanzadas	80 %
Distancia total	1119.5 metros
Nº de choques	4

Tabla 9.5: Segunda prueba de resistencia real.

La prueba finalizó por falta de batería en el robot, sin embargo no puede considerarse que ese tiempo sea la duración de la misma en navegación con el robot ya que no se encontraba al 100 %.

Las metas no alcanzadas se produjeron por obstáculos móviles (personas en movimiento) bloqueando el paso del robot (imposible llegar a la meta físicamente). También se produjeron metas abordadas por riesgo de colisión (el robot había quedado situado muy cerca de las patas de una mesa).

Los choques fueron roces leves de las ruedas con obstáculos bajos. Esto es debido a que el planificador ajusta mucho la trayectoria para que el robot pueda navegar por lugares estrechos.

⁴La distancia total recorrida no pudo contabilizarse debido a un error en la configuración del nodo.

Cabe destacar la distancia total recorrida, que asciende a más de un kilómetro realizando maniobras con todo tipo de obstáculos.

Tercera prueba

La tercera prueba de resistencia se planteó como la prueba absoluta, midiendo parámetros como la autonomía de las baterías, con distancias entre puntos más largas, navegación por diferentes salas y pequeños desniveles.

Los puntos de meta preestablecidos se distribuyen en el mapa tal y como se muestra en la figura 9.7 y el robot navega aleatoriamente de uno a otro.



Figura 9.7: Distribución de puntos de meta en la segunda prueba de resistencia.

El entorno de la prueba presentó obstáculos altos y bajos (no detectables por el sensor láser), dos pequeñas rampas, personas en continuo movimiento e incluso elementos del entorno cambiados de posición. A continuación se muestran los resultados de la prueba.

Puntos de meta	17
Tiempo total	2 horas y 38 minutos
Metas enviadas	164
Metas alcanzadas	106
Tasa metas alcanzadas	65 %
Distancia total	4000.5 metros
Nº de choques	2

Tabla 9.6: Tercera prueba de resistencia real.

Sobre estos resultados cabe decir que el número de metas alcanzadas se ve redu-

cido debido a que en ocasiones el planificador global precisaba de algún reintento para trazar la trayectoria al el punto de meta y a que el robot a veces encontraba dificultades para conducir su trayectoria a través de pasos estrechos.

La duración de la batería en navegación continua durante esta prueba superó las dos horas y media. Al finalizar la prueba el voltaje de las baterías alcanzó un valor 10.5 voltios.

En cuanto al número de choques, estos fueron un roce a un obstáculo bajo al tratar de esquivarlo y un choque con las ruedas de una silla.

Destaca el alto valor de la distancia recorrida, que alcanza los 4 kilómetros sin la intervención humana, sin que se ponga en riesgo la integridad de ninguna persona involucrada en la prueba ni la del propio robot.

9.2.3 Aspectos de la navegación

Tras las pruebas realizadas, analizando los resultados y observando el comportamiento general de todo el sistema robótico podemos decir que existen algunas limitaciones en la navegación y aspectos que no se tienen en cuenta por las características del robot.

- **En cuanto a la navegación:**

Existen limitaciones en cuanto a rampas, ya que este modo de navegación no tiene en cuenta la inclinación del robot para saber si este se encuentra desplazándose por un plano inclinado. Por otro lado tampoco es posible distinguir cambios por debajo del plano de navegación como agujeros, escalones... Estos dos aspectos se deben principalmente a que la navegación se realiza puramente en dos dimensiones.

No obstante, el sistema sí es capaz de absorber pequeñas rampas en las que los sensores no reciban excesivas interferencias con el cambio de plano y siempre que el robot no realice muchos giros para no comprometer la localización debido al cambio del rozamiento de las ruedas del robot con el suelo.

- **En cuanto a las características del robot:**

Ciertos obstáculos bajos no son detectados por los sensores debido a que el robot es capaz de superarlos sin problemas, es por ello que no se ha incidido en detectarlos.

Obstáculos de materiales transparentes, como puertas de cristal, no son detectadas por ninguno de los dos sensores debido a su naturaleza infrarroja, por tanto se definen este tipo de obstáculos como caso límite para el sistema robótico.

Capítulo 10

Conclusiones

Se presentan a continuación las conclusiones...

10.1 Conclusión sobre la metodología

10.2 Conclusión sobre los resultados

Una vez finalizado el proyecto...

10.3 Desarrollos futuros

Un posible desarrollo...

Apéndice

Apéndice A

Configuración del sistema

En este apéndice se resume de manera simplificada y muy enfocada la configuración completa del sistema robótico objeto de este trabajo, tanto la parte referente al software de robot como la parte destinada a la configuración hardware de todos los equipos que incorpora.

A.1 Configuración del espacio de trabajo

En ROS el espacio de trabajo es el lugar donde se realiza el desarrollo software de los paquetes ROS. Este en torno de trabajo es capaz de gestionar la correcta compilación de los nodos.

El espacio de trabajo es el determinado por la herramienta Catkin. A partir de la versión Indigo de ROS, casi todos los paquetes se encuentran adaptados al entorno Catkin y funcionan de manera casi inmediata.

A continuación se describen los pasos para crear un espacio de trabajo Catkin (extraídos del tutorial http://wiki.ros.org/catkin/Tutorials/create_a_workspace) y los pasos para clonar el repositorio de GitHub que aloja el código.

Tras instalar ROS en el equipo deseado, instalamos Catkin:

```
1 $ sudo apt-get install ros-indigo-catkin
2 $ mkdir -p ~/catkin_ws/src
3 $ cd ~/catkin_ws/src
4 $ catkin_init_workspace
```

Código A.1: Instalación y workspace de Catkin

A continuación incluimos el directorio de desarrollo para que sea reconocido por ROS a la hora de buscar dependencias:

```
1 $ cd ~/catkin_ws
2 $ source devel/setup.bash
```

Código A.2: Source al setup de nuestro entorno Catkin

A.1.1 Instalación de las librerías

Para clonar el desarrollo software de este proyecto bastaría con clonar el repositorio de GitHub en el que se ha trabajado en este proyecto https://github.com/danimtb/pioneer3at_ETSIDI en nuestra carpeta `~/catkin_ws/src`.

```
1 $ cd ~/catkin_ws/src
2 $ git clone --recursive https://github.com/danimtb/pioneer3at_ETSIDI.git .
```

Fuente: https://github.com/danimtb/pioneer3at_ETSIDI

Código A.3: Clonado del repositorio *pioneer3at_ETSIDI*

Sin embargo, es recomendable que si se desea realizar algún desarrollo posterior, se realice un fork en GitHub de este proyecto (Figura A.1) y se clone el repositorio propio.

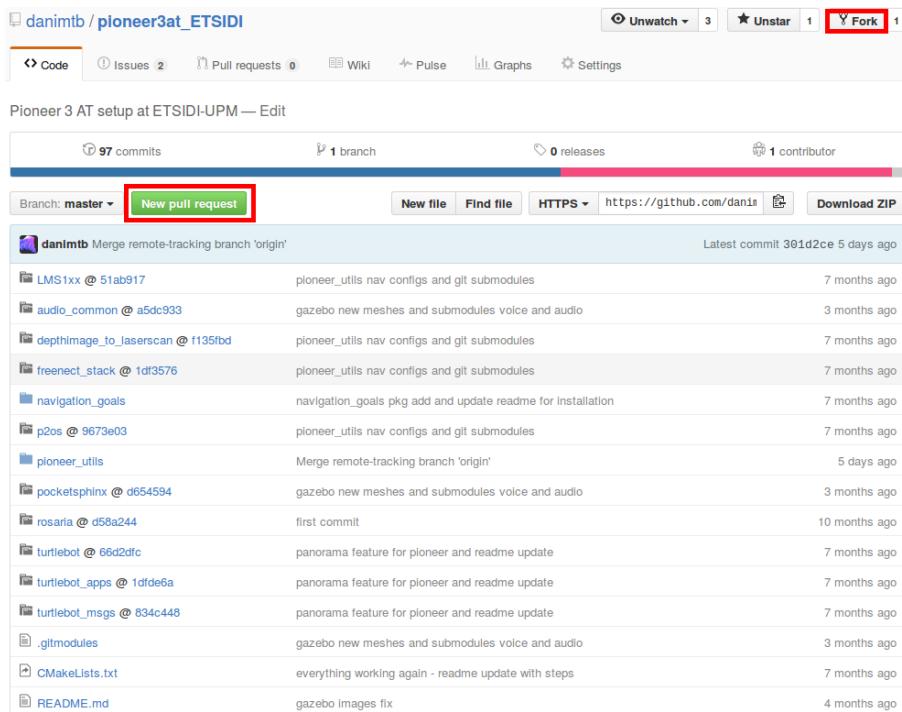


Figura A.1: Vista del repositorio de GitHub de este proyecto.

De esta forma podemos guardar los cambios realizados en el repositorio de la persona que hace la modificación, con la intención de incorporar los cambios más tarde al repositorio de desarrollo principal (https://github.com/danimtb/pioneer3at_ETSIDI.git) mediante Pull Request.

A.1.2 Gestión de las dependencias

ROS se vale de la información almacenada en la definición de cada uno de sus paquetes en *package.xml* y gestionar algunas dependencias de librerías, para lo cual se sirve de la herramienta rosdep **referencia** creada por los desarrolladores de ROS para este propósito.

Para comenzar a utilizarla inicializamos rosdep y actualizamos las dependencias:

```
1 $ sudo rosdep init  
2 $ rosdep update
```

Código A.4: Inicializando la herramienta *rosdep*.

A continuación se describen los pasos necesarios para instalar las dependencias de cada sección:

- Navegación: Instalamos los paquetes adicionales para realizar la navegación y mapeo (Capítulo 5).

```
1 $ sudo apt-get install ros-indigo-navigation  
2 $ sudo apt-get install ros-indigo-gmapping
```

Código A.5: Instalando los paquetes de navegación mapeo.

- Rosaria: Instalamos las dependencias de RosAria, principalmente la librería Aria.

```
1 $ rosdep install rosaria
```

Código A.6: Instalando las dependencias de rosaria.

Para el reconocimiento de los comandos de voz utilizamos el paquete *pocketsphinx*:

- pocketsphinx: Instalamos los paquetes de conversión a texto y buscamos las dependencias del paquete.

```
1 $ sudo apt-get install gstreamer0.10-gconf  
2 $ rosdep install pocketsphinx
```

Código A.7: Instalando las dependencias de *pocketsphinx* y *gstreamer*.

Es posible que necesitemos las dependencias añadir de *audio_capture*.

```
1 | $ rosdep install audio_capture
```

Código A.8: Instalando las dependencias de *audio_capture*.

Para el audio (sintetizado de voz) utilizamos el nodo *sound_play*, por lo que debemos instalar sus dependencias.

- *sound_play*: Nodo para el sintetizado de voz y reproducir sonidos.

```
1 | $ rosdep install sound_play
```

Código A.9: Instalando las dependencias de *sound_play*.

Una vez disponemos de todos los paquetes y sus respectivas dependencias no debemos olvidarnos de que es necesario compilarlos:

```
1 | $ cd ~/catkin_ws
2 | $ catkin_make
```

Código A.10: Compilando los paquetes del espacio de trabajo Catkin.

A.2 Configuración del hardware

En este apéndice se describen algunos ajustes útiles del hardware que se ha utilizado en el proyecto. Esta información es importante para que el sistema robótico desarrollado funcione pero no ha sido incluida dentro del cuerpo de la memoria para no alargar las explicaciones.

En cualquier caso, los procedimientos que a continuación se describen es probable que no sea necesario volver a realizarlos en futuros usos del robot si se mantiene la configuración descrita en este trabajo.

A.2.1 Calibración de la odometría

La lectura de la posición de los motores del robot se realiza mediante unos encoders situados en el eje de cada motor.

El firmware ARCOS de la placa controladora del robot es actualizable y gestiona los parámetros de calibración de la odometría. Este firmware es actualizable aunque para ello es necesario ponerse en contacto con el fabricante Adept.

Estos parámetros referentes a la odometría son configurables a través de la librería Aria en caso de que no se desee acceder a modificar el firmware del robot.

El nodo de ROS RosAria es capaz de pasar estos parámetros a la placa controladora del robot a través de Aria. Es por ello que podemos configurar dinámicamente estos parámetros del robot cada vez que ejecutamos el nodo. De esta forma evitamos tener que recurrir a Adept y a software específico para modificar los parámetros de ARCOS.

El problema principal es que el robot viene configurado para utilizar los neumáticos de exterior, que tienen un diámetro mayor, mientras que los neumáticos de interior son más pequeños y por tanto hay que ajustar los parámetros de la odometría.

Los parámetros a ajustar son los siguientes:

- TicksMM: Pulsos por cada milímetro.
- DriftFactor: Valor de pulsos para compensar las diferencias entre las ruedas de un lado y del otro.
- RevCount: Número de pulsos en los encoders para que el robot realice un giro de 180°.

Del propio manual del robot extraemos la siguiente información:

- Pulsos por revolución (ticks/rev): 500.
- Relación de transmisión (gear ratio): 49:8.

Para ajustar *TicksMM* utilizamos la fórmula del manual:

$$TicksMM = \frac{4 \cdot \text{ticks/rev} \cdot \text{gearingratio}}{\text{wheeldiam}(mm) \cdot \pi} == \frac{4 \cdot 500 \cdot 49,8}{190 \cdot \pi} = 166,86 \quad (\text{A.1})$$

Para ajustar *RevCount* situamos al robot paralelo a una referencia en el suelo (una línea recta en el suelo), marcamos el punto en el que las ruedas tocan el suelo y manualmente, agarrando al robot por las ruedas, giramos las mismas hasta que el robot quede girado 180°. Contamos las revoluciones de una de las ruedas gracias a las marcas que habíamos hecho y hacemos el siguiente cálculo:

$$RevCount = \text{ticks/rev} \cdot \text{gearingratio} \cdot \text{revolucionesdeunaruenda} \quad (\text{A.2})$$

En el caso de la última calibración:

$$RevCount = 500 \cdot 49,8 \cdot 1,5 = 37350 \quad (\text{A.3})$$

Por último, para ajustar *DriftFactor* procedemos de manera experimental. Situamos al robot paralelo a una linea recta marcada en el suelo y lo teleoperamos hacia adelante unos 4 metros. Si observamos que el robot se separa de la linea recta, ajustamos el valor de este parámetro en consecuencia.

Para más información, se recomienda leer el manual del fabricante [Mob06].

A.2.2 Ordenador de abordo Intel NUC

Los datos de inicio de sesión del ordenador son:

- Usuario: pioneer3at.
- Contraseña: pioneer3at.
- Nombre del equipo: P3AT.

```
1 $ gedit ~/.bashrc
```

Código A.11: Abriendo el *.bashrc*.

EL ordenador Intel NUC está configurado para que genere una red Wifi Ad-hoc llamada "P3AT" de manera automática al inicio. Esta red presenta direcciones IP dentro del rango 10.42.0.X y su contraseña de acceso es "pioneer3atROS".

El nodo máster de ROS se ejecuta en este ordenador, por lo que es preciso indicarlo en el archivo *.bashrc*.

Para editararlo, abrimos la terminal y escribimos:

Una vez abierto, al final del archivo añadimos las siguientes dos líneas indicando la dirección y el puerto del Master y la dirección IP del ordenador:

```
1 export ROS_IP=10.42.0.1
2 export ROS_MASTER_URI=http://10.42.0.1:11311
```

Código A.12: Añadiendo las direcciones IP al *.bashrc*.

El resto de configuraciones de software se han indicado a lo largo del documento.

El ordenador está alimentado a 12 voltios directamente desde la placa de alimentación del robot y se encuentra anclado al mismo mediante tiras adhesivas de velcro, por lo que es posible retirarlo del cuerpo del robot en caso de que sea necesario.

A.2.3 Sensor Kinect

El sensor Kinect utilizado en el proyecto no dispone de ninguna moficiación específica o modo de funcionamiento especial. Lo referente a su conexiónado ha sido descrito anteriormente en la sección 7.1.3.

Tan solo indicar que puede realizarse una primera toma de contacto a través de la librería *libfreenect*.

Una manera rápida de hacerlo es utilizando el gestor de dependencias en C++ biicode (www.biicode.com), en un sistema operativo Ubuntu siguiendo esta guía <http://blog.biicode.com/kinect-xbox360-biicode/>.

A.2.4 Láser SICK LMS100

El sensor láser LMS100 de la marca Sick es un sensor de ampli rango y largo alcance de tipo industrial.

El fabricante Sick ofrece un software propietario llamado SOPAS Ingeineering Tool que permite acceder a la configuración interna del sensor. Este software tan solo puede utilizarse bajo el sistema operativo Microsoft Windows, en concreto ha sido utilizado con Windows 7.

Para instalarlo hay que recurrir a la web del fabricante: <https://www.mysick.com/eCat.aspx?go=FinderSearch&Cat=Row&At=Fa&Cult=English&FamilyID=235&Category=Software&Selections=54195,54293>

Para configurar el dispositivo debemos conectarlo a un puerto Ethernet de un ordenador con el software instalado y proporcionarle alimentación.

El puerto ethernet del ordenador debe estar configurado para obtener una IP automática. Una vez aparezca la red como conectada procedemos a abrir el software SOPAS.

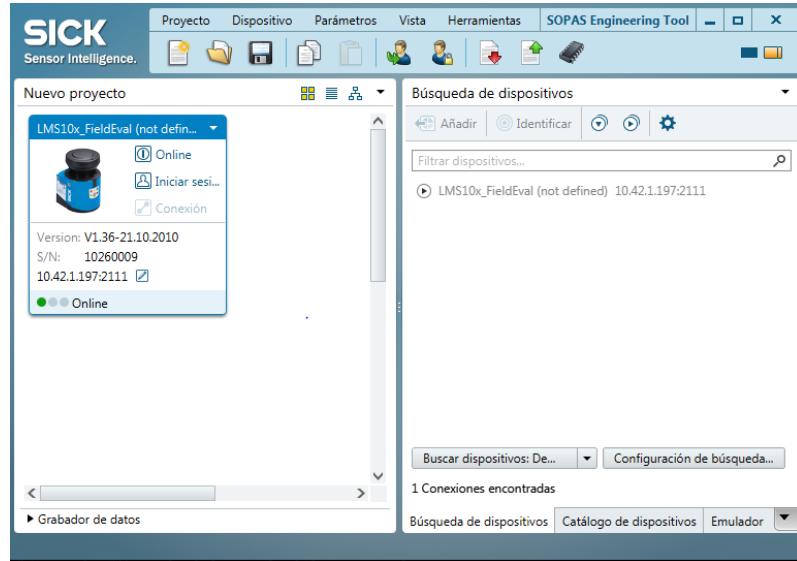


Figura A.2: Software Sick SOPAS con el sensor detectado.

Automáticamente detectará el dispositivo pero seguramente no podamos acceder a él debido a una dirección IP errónea.

Para ello abriremos el apartado de configuración y seleccionamos "Asignar IP automáticamente" para que nos asigne el rango adecuado.

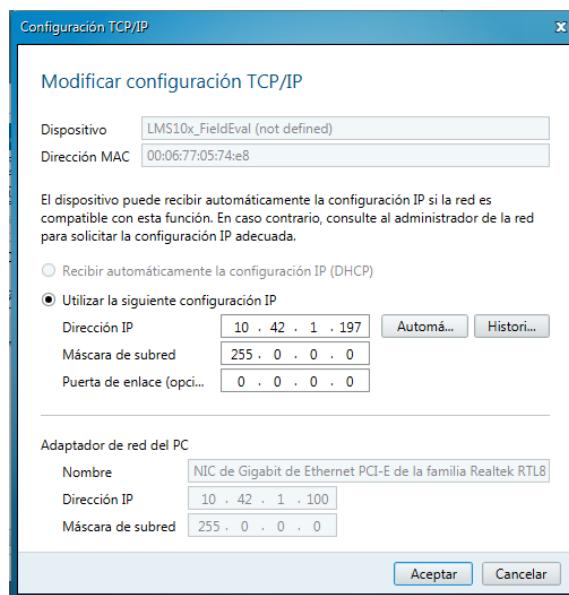


Figura A.3: Configurando la dirección IP del dispositivo.

Si queremos una dirección IP de determinado rango, la mejor manera de proceder es asignar al adaptador de red del ordenador una IP manual del rango deseado y realizar el procedimiento anterior.

También podemos acceder a los parámetros del sensor y visualizar en un gráfico

la información que está captando en el momento.

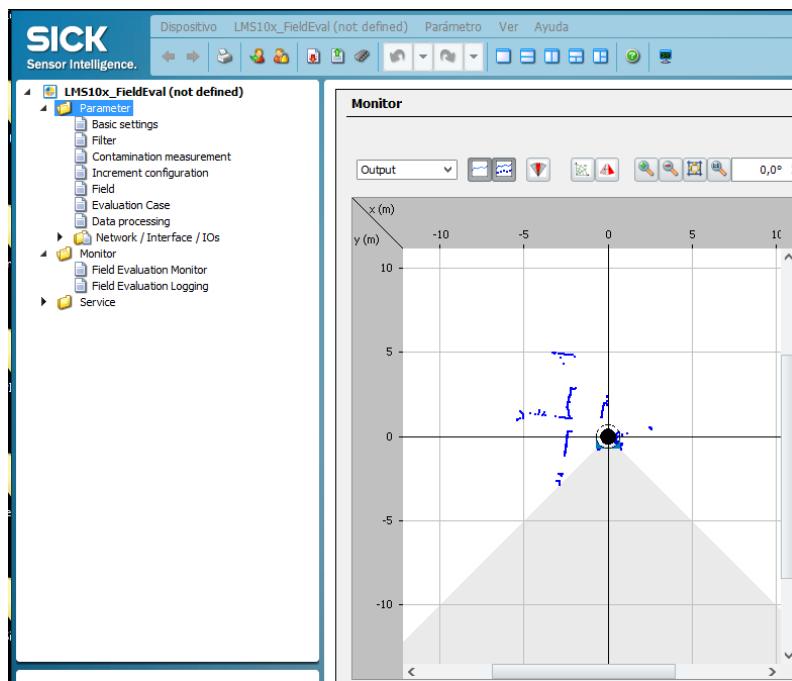


Figura A.4: Gráfico de la información captada por el sensor en el software SOPAS.

A partir de aquí podremos conectarnos al láser con el nodo ROS LMS1xx, configurando una IP fija en el adaptador de red ethernet del ordenador al que lo conectemos, en este caso el ordenador Intel NUC del robot.

- Dirección IP del láser Sick LMS100: 10.42.1.197
- Dirección IP del adaptador ethernet del ordenador Intel NUC: 10.42.1.1

Para más información conviene leer el manual que ofrece Sick [SIC09].

Apéndice B

Manual de uso del robot

Este apéndice es una guía de funcionamiento del robot Pioneer 3 AT (Petrois) utilizado e implementado al acabar este proyecto fin de grado.

B.1 Encendido del robot

El robot se enciende mediante un interruptor situado en su parte trasera, el cual proporciona alimentación general a todos los sistemas.

Adicionalmente es necesario encender el ordenador de abordo Intel NUC mediante su correspondiente botón y el altavoz frontal del robot, para lo cual será necesario tener activada la alimentación auxiliar 2 (AUX 2) en el panel de control.

B.2 Panel de control y parada de emergencia

El panel de control del robot se sitúa en su lateral derecho. Aquí disponemos de una serie de botones, indicadores luminosos, acústicos y conexiones.



Figura B.1: Panel de control del robot Petrois.

A continuación se describe cada uno de los elementos de la figura B.1:

- Pulsador MOTORS: Encargado de habilitar y deshabilitar los motores del robot. La activación de los mismos puede realizarse a través de software. Es necesario pulsar este botón para rearmar el robot cuando se pulsa la seta de emergencia. Si se pulsa una vez más el robot realiza una secuencia de movimientos para comprobar que los motores funcionan correctamente.
- Pulsador RESET: Resetea la placa controladora del robot. El robot queda su estado inicial, tal y como si lo acabásemos de encender. Al pulsar este botón se interrumpe cualquier comunicación con la placa de control.
- Pulsador AUX 1: Habilita la alimentación 1 de la placa de alimentación. No se encuentra en uso.
- Pulsador AUX 2: Habilita la alimentación 2 de la placa de alimentación. Es necesario tenerlo habilitado para que el altavoz frontal del robot funcione.
- LED PWR: Indica el estado de los motores.
- LED STATUS: Indica el estado del robot.
- LED BATERY: Indica el estado de carga de la batería.
- LEDs AUX 1 y AUX 2: Indica si las alimentaciones auxiliares se encuentran activas.
- LEDs RX y TX: Muestran el estado de la comunicación con la placa de control.
- SERIAL: Puerto RS-232 que comunica con la placa de control. Puede ser utilizado para conectar un ordeandor externo directamente la placa de control del robot.

Muchos de los estados anteriores se indican mediante una señal acústica. El resedeo de la placa tiene un sonido característico al igual que cuando se pierde la conexión con la placa de control.

El pulsador de emergencia se encuentra en la parte superior del robot y es de vital importancia cuando el robot se mueve de manera descontrolada, choca o está en peligro la integridad de una persona o del propio robot. Es por ello que se recomienda su uso siempre que se encuentre en una situación de las anteriores.

Al pulsar la seta de emergencia ésta quedará enclavada y el robot dará una señal acústica continuada indicando que se encuentra en parada. En este punto los motores quedan deshabilitados y las ruedas del robot pueden moverse libremente.

Para rearmar el robot basta con devolver la seta de emergencia a su posición DE reposo girando levemente la misma en el sentido de las flechas. Despues es necesario pulsar el botón MOTORS para volver a habilitar los motores.

Cabe indicar que la comunicación de cualquier ordenador con la placa de control del robot no se interrumpe, por lo que al rearmar el robot es posible que éste continúe con los movimientos previos a la parada de emergencia. Asegúrese de que las consignas de movimiento son las correctas y ponga especial precaución cuando realice el rearmando del robot.

Para información más extensa y detallada se recomienda leer la guía de usuario ofrecida por el fabricante [Mob06].



Figura B.2: Botón de parada de emergencia del robot Petrois.

B.3 Conexión mediante un ordenador externo vía Wifi

Al encender el ordenador Intel NUC, éste está configurado para generar directamente un HotSpot Wifi o red Ad-hoc con el nombre "P3AT". Esta red Wifi maneja direcciones IP en el rango 10.42.0.X por lo que es recomendable conectarse a ella con una IP fija que esté dentro de ese rango. Su contraseña es "pioneer3atROS".

Si se está ejecutando el nodo MASTER en el ordenador Intel NUC, éste estará configurado con la IP 10.42.0.1. Para indicar que queremos conectarnos a un nodo MASTER que se ejecuta en otra máquina debemos editar el script *.bashrc* del ordenador externo, que se ejecuta siempre que abrimos una terminal.

Para editarlo, abrimos la terminal y escribimos:

```
1 $ gedit ~/.bashrc
```

Código B.1: Abriendo el *.bashrc*.

Una vez abierto, al final del archivo añadimos las siguientes dos líneas:

```
1 export ROS_IP=10.42.0.77
2 export ROS_MASTER_URI=http://10.42.0.1:11311
```

Código B.2: Añadiendo las direcciones IP al *.bashrc*.

La primera línea indica la IP fija con la que el ordenador externo se conecta a la red Ad-hoc del Intel NUC. Modificar la IP y escribir la IP del ordenador externo. La dirección IP del MASTER es la dirección que está configurada en el Intel NUC y a la que tratarán de conectarse los nodos cuando se lancen desde el ordenador externo.

Este procedimiento está hecho igual en el Intel NUC pero indicando tan solo que la IP de ese equipo es **ROS_IP=10.42.0.1**.

Para que sea más cómo controlar este ordenador podemos conectarnos vía VNC con el cliente por defecto de Ubuntu 14.04 Cliente de escritorio remoto Remmina” siempre y cuando estemos conectados a la red Wifi ”P3AT”. De este modo podremos visualizar el escritorio y lanzar los nodos pertinentes.

B.4 Acceder a la placa de alimentación

La placa de alimentación se encuentra bajo la tapa negra principal del robot, justo debajo del sensor Kinect.

Su posición no es muy accesible y se recomienda encarecidamente leer el manual para tener acceso a la misma.

Por otro lado, siempre puede acderse a la alimetación de 12 voltios a través del cable de alimentación del sensor Kinect y a 5 voltios a través de los cables de alimentación del altavoz delantero.

B.5 Cargar las baterías del robot

El robot Pioneer 3 AT dispone de un pack de 3 baterías estásicas de plomo-ácido a 12 voltios. Estas baterías suministran alimentación a todos los sistemas del robot y proporcionan una autonomía aproximada de 2 horas de funcionamiento.

El acceso a las baterías se encuentra en el interior del chasis del robot y son accesibles mediante una trampilla trasera.



Figura B.3: Trampilla de acceso a las baterías.

La carga de las baterías se realiza con el adaptador de la figura B.4 a través del conector de carga situado junto al interruptor de alimentación general.

No ha podido estimarse el tiempo de carga completo, sin embargo el propio cargador incorpora un indicador luminoso que muestra el estado de carga de las baterías.



Figura B.4: Puerto de carga y cargador del robot Petrois.

B.6 Ordenador interno del Pioneer 3 AT

EL ordenador interno del robot Pioneer 3 AT se encuentra en desuso.

Todos los controles necesarios para acceder al ordenador están disponibles a través del panel situado en el lateral izquierdo del robot.



Figura B.5: Panel del ordenador interno (lateral izquierdo) del robot Petrois.

En este lateral encontramos elementos adicionales como la antena de conexión

wifi del ordenador interno y el acceso a los puertos USB y Jacks de micrófono y auriculares.

IMPORTANTE:

Originalmente este ordenador se encontraba conectado a la placa controladora del robot de manera interna a través el puerto COM1 (Accesible en el interior del robot). En su lugar se conectó el convertidor de RS-232 a USB utilizado con el ordenador Intel NUC, por lo que el ordenador interno se encuentra desconectado del robot.

IMPORTANTE:

Uno de los ventiladores internos situado en la parte frontal derecha del robot fue desconectado debido a su elevado ruido y su encendido permanente. En caso de utilizar el ordenador interno del robot es posible que sea necesario volver a conectarlo. Sin embargo no existe riesgo de sobrecalentamiento ya que el propio ordenador dispone de un ventilador adicional que se pone en marcha al encenderlo.

Apéndice C

Información y documentos ONLINE

En este apéndice se muestra información adicional de todo tipo referente al proyecto y a los materiales utilizados.

C.1 Repositorio software

El repositorio del software desarrollado en ente proyecto se encuentra almacenado en GitHub.

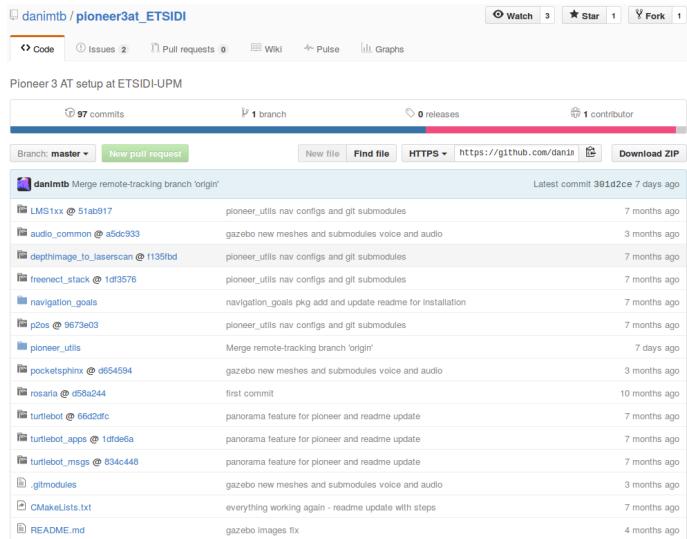


Figura C.1: Repositorio software *pioneer3at_ETSIDI*.

https://github.com/danimtb/pioneer3at_ETSIDI

En él se encuentra una pequeña guía *Readme* actualizada con información sobre las utilidades que ofrece el repositorio y en concreto el desarrollo realizado en el paquete *pioneer_utils*.

C.1.1 Readme

Copia del documento *Readme* del repositorio.

pioneer3at_ETSIDI: Pioneer 3 AT setup at ETSIDI-UPM

This repo hosts ROS packages working on Indigo version. These packages are needed to setup a CATKIN workspace and include all files needed for Pioneer 3 AT robot at ETSIDI-UPM university.

Submodules

Pioneer 3 AT uses the following additional ROS packages showed up as git submodules:

- [rosaria](#): Interface with Aria library to control motors, battery and encoders. (See [rosaria docs](#))
- [p2os\(indigo-stable\)](#): package with some useful configurations for navigation and pioneer urdf models. (See [p2os docs](#)).
- [LMS1xx](#): Sick ROS drivers from ClearPath Robotics to use Sick LMS100 ethernet laser scanner. (See [LMS1xx docs](#)).
- [freenect_stack](#): For Kinect 1 XBOX 360, (See [freenect_stack docs](#) and [freenect_launch docs](#)).
- [depthimage_to_laserscan](#): Creates LaserScan data from depthimage devices such as kinect. (See [depthimage_to_laserscan docs](#)).
- [turtlebot_apps](#): Interactive implementations reused for P3AT robot such as "follower" demo. (See [turtlebot docs](#)).

Catkin workspace

Please, refer to de ROS Indigo installation page and follow the steps to install and set your ROS environment as well as updating rosdep tool.

- [ROS Indigo installation wiki](#).

Steps may change for each ROS version:

1.- \$ sudo apt-get install ros-indigo-desktop-full

2.- \$ sudo rosdep init

3.- \$ rosdep update

All thesee files and directories should be placed at src/ directory in a [catkin workspace](#). Follow steps in a terminal:

1.- \$ catkin_init_workspace catkin_ws

2.- \$ cd catkin_ws/src

3.- \$ git clone --recursive https://github.com/danimtb/pioneer3at_ETSIDI.git . (NOTE THE . AT THE END OF THE LINE)

```
4.- $ cd ~/catkin_ws  
5.- $ rosdep install rosaria  
6.- $ rosdep install freenect_launch  
7.- $ catkin_make This will compile all targets placed in you catkin src directory
```

You'll may also need ros navigation stack and gmapping:

```
$ sudo apt-get install ros-indigo-navigation  
$ sudo apt-get install ros-indigo-gmapping
```

For turtlebot applications to compile and run:

```
$ rosdep install turtlebot  
$ rosdep install turtlebot_teleop
```

Content: pioneer_utils

This is the core of my work. **pioneer_utils** mainly adds some configuration specific parameters to keep all things working.

- Odometry params calibrations used in rosaria.
- Laser IP address.
- Pioneer URDF model with Sick Laser and Kinect.
- Navigation tweaks in costmaps, base and planners.
- depthimage to scan configs (for low, medium and long range obstacles).
- Gazebo settings and launch files with gazebo plugins and urdf model.
- Maps used at ETSIDI-UPM Lab and in gazebo Willow Garage world.
- RViz launch files with specific visualization configs.

And implements easy to use nodes:

- Teleoperation node. `rosrun pioneer_utils teleop_p3at`
- Dead Reckoning node: Let robot move alone and making turns. `rosrun pioneer_utils moving_alone`
- nav-waypoints node (navigation_goals): Send global or local goals to navigation stack. `rosrun pioneer_utils nav-waypoints`
- endurance_test node: implements randomly navigation to a list of pointsSee launch file template:
` ` `roslaunch pioneer_utils endurance_test.launch"List of points as *map_locations.txt* rosparam.

Usage

This repository is intended to be a simple method to setup and run everything needed to use

Pioneer 3 AT. After cloning this github repo in your catkin_ws src/ directory do the following: \$ cd catkin_ws \$ catkin_make

Navigation Stack

Now run "roscore" and the nodes needed with the .launch file you'll find in src/ directory. In your terminal run "roscore": \$ roscore

In another terminal, we'll bring up all drivers for hardware using kinect, laser Sick, and Rosaria with calibration config setup: \$ roslaunch pioneer_utils pioneer3at-rosaria.launch

Now, you can start navigation stack with amcl like this: \$ roslaunch pioneer_utils global_navigation_p3at.launch

Pioneer 3 AT Follower (from turtlebot)

Open a terminal and launch the follower: \$ roslaunch pioneer_utils simple_follower.launch

If you want to guide your robot following you to build a map, run also: \$ roslaunch p2os gmapping.launch

Pioneer 3 AT Panorama (from turtlebot)

Open a terminal and launch the panorama: \$ roslaunch pioneer_utils panorama-pioneer-3at.launch

Follow [turtlebot's panorama wiki](#) to know how to use this and take nice panorama pics. Also see [turtlebot_panorama API](#).

Gazebo Simulation

Open a terminal and launch the follower: \$ roslaunch pioneer_utils pioneer3at_gazebo_world.launch

If you want to do some navigation with Willow Garage's map type in other terminal: \$ roslaunch pioneer_utils gazebo-global_navigation_p3at.launch

C.2 Preguntas en ROS Answers y Github

Preguntas realizadas en el foro ROS Answers y mediante issues en repositorios GitHub

ROS Answers:

- What compact PC to buy to run ROS in a mobile robot

<http://answers.ros.org/question/218043/what-compact-pc-to-buy-to-run-ros-in-a-mobi>

- Connecting a Sick laserscanner via ethernet

<http://answers.ros.org/question/207220/connecting-a-sick-laserscanner-via-ethernet/#209347>

Github:

- clearpathrobotics/LMS1xx: Measurement stopped with LMS100
<https://github.com/clearpathrobotics/LMS1xx/issues/14>

C.3 Multimedia

Archivos multimedia obtenidos durante este proyecto en relación.

Colección de vídeos:

- Algunos vídeos de este proyecto:

- Pioneer 3 AT navigation using ROS + Laser + Kinect
<https://www.youtube.com/watch?v=vXFqmWmqZSs>
- Pioneer 3 AT first steps with ROS navigation
<https://www.youtube.com/watch?v=w9qAdscY48k>
- Pioneer 3 AT long navigation runs at ETSIDI-UPM
<https://www.youtube.com/watch?v=DeSSNJWEcaE>
- Pioneer 3 AT follower mode - Timelapse
<https://www.youtube.com/watch?v=925kjDg3v5A>
- Pioneer 3 AT follower mode - test
<https://www.youtube.com/watch?v=R58UV5R6UjM>

- Vídeos de antiguos proyectos:

- Pioneer 3at testing a square path
<https://www.youtube.com/watch?v=ignPU8WrxjI>
- Robot Mobile Control (Pioneer3at). PFC developed in Euiti, UPM.
<https://www.youtube.com/watch?v=cTNxp5lORAE>

Imágenes:

- De robots e impresoras 3D - TFG Pioneer 3 AT
<https://goo.gl/photos/NLyeXUGAGYdYiTcy9>

C.4 Memoria del trabajo

La memoria del trabajo ha sido desarrollada en L^AT_EX, bajo el editor *TexStudio* en un ordenador con sistema operativo GNU/Linux Ubuntu 14.04 LTS.

Su desarrollo ha sido puesto bajo un controlador de versiones y alojado en el siguiente repositorio de github:

https://github.com/danimtb/TFG_pioneer3at

En el propio repositorio se encuentran las figuras utilizadas y los archivos .tex, donde se ha escrito el mismo, y la bibliografía empleada.

El documento PDF compilado de L^AT_EXpuede consultarse en el siguiente enlace:

https://github.com/danimtb/TFG_pioneer3at/TFG.pdf

Bibliografía

- [All11] Hunter Allen. p2os ROS Package. <http://wiki.ros.org/p2os>, 2011. [Online, consultado 10 de Marzo de 2015].
- [AMRS11] F Alonso-Martin, Arnaud A Ramey, and Miguel A Salichs. Maggie: el robot traductor. In *Proceedings of the 9th Workshop RoboCity2030-II, Madrid, Spain*, pages 57–73, 2011.
- [Ban11] Konrad Banachowicz. LMS1xx ROS Package. <http://wiki.ros.org/LMS1xx>, 2011. [Online, consultado 1 de Marzo de 2015].
- [Car90] Carnegie Mellon University. CMU Sphinx, Open source speech recognition toolkit. <http://cmusphinx.sourceforge.net/>, 1990. [Online, consultado 26 de Julio de 2015].
- [CCR07] Thomas Collins, JJ Collins, and Conor Ryan. Occupancy grid mapping: An empirical evaluation. In *Control & Automation, 2007. MED'07. Mediterranean Conference on*, pages 1–6. IEEE, 2007.
- [Cle10] Clearpath Robotics. Clearpath Robotics: Autonomous Mobile Robots. www.clearpathrobotics.com, 2010. [Online, consultado 23 de Marzo de 2015].
- [Com10] Comunidad Open Source. Proyecto OpenKinect. http://openkinect.org/wiki/Main_Page, 2010. [Online, consultado 16 de Marzo de 2015].
- [CR10] Javier A. Cuenca Retana. *Sistema de posicionamiento en interiores mediante balizas bluetooth*. 2010.
- [DL91] Stephen L Dickerson and Brett D Lapin. Control of an omnidirectional robotic vehicle with mecanum wheels. In *Telesystems Conference, 1991. Proceedings. Vol. 1., NTC'91., National*, pages 323–328. IEEE, 1991.
- [dS07] Universidad de Sevilla. Sistemas de locomoción de robots móviles. http://www.esi2.us.es/~vivas/ayr2iae/Loc_MOV.pdf, 2007. [Online, consultado 15 de Junio de 2015].
- [DSTH12] Laura Dabbish, Colleen Stuart, Jason Tsay, and Jim Herbsleb. Social coding in github: transparency and collaboration in an open software repository. In *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work*, pages 1277–1286. ACM, 2012.

- [Fau11] Josh Faust. turtlesim ROS Package. <http://wiki.ros.org/turtlesim>, 2011. [Online, consultado 10 de Febrero de 2015].
- [FBDT99] Dieter Fox, Wolfram Burgard, Frank Dellaert, and Sebastian Thrun. Monte carlo localization: Efficient position estimation for mobile robots. *AAAI/IAAI*, 1999:343–349, 1999.
- [FBT⁺97] Dieter Fox, Wolfram Burgard, Sebastian Thrun, et al. The dynamic window approach to collision avoidance. *IEEE Robotics & Automation Magazine*, 4(1):23–33, 1997.
- [Fer12] Michael Ferguson. pocketsphinx ROS Package. <http://wiki.ros.org/pocketsphinx>, 2012. [Online, consultado 25 de Julio de 2015].
- [FMEM11] Tully Foote, Eitan Marder-Eppstein, and Wim Meeussen. tf ROS Package. <http://wiki.ros.org/tf>, 2011. [Online, consultado 15 de Febrero de 2015].
- [Gar11] Willow Garage. Unified Robot Description Format (URDF). <http://wiki.ros.org/urdf>, 2011. [Online, consultado 28 de Marzo de 2015].
- [Ger09] Brian P Gerkey. amcl. <http://wiki.ros.org/amcl>, 2009. [Online, consultado 16 de Abril de 2015].
- [GH11] Blaise Gassend and Austin Hendrix. SoundPlay ROS Package. http://wiki.ros.org/sound_play, 2011. [Online, consultado 15 de Julio de 2015].
- [GK08] Brian P Gerkey and Kurt Konolige. Planning and control in unstructured terrain. In *ICRA Workshop on Path Planning on Costmaps*, 2008.
- [Goe15] R. Patrick Goebel. *ROS By Example Volume 2 - INDIGO*. Lulu Enterprises, Inc., 2015.
- [Goo09] Google. Google self-driving car project. <https://www.google.com/selfdrivingcar/>, 2009. [Online, consultado 20 de Octubre de 2015].
- [GSB07] Giorgio Grisetti, Cyrill Stachniss, and Wolfram Burgard. Improved techniques for grid mapping with rao-blackwellized particle filters. *Robotics, IEEE Transactions on*, 23(1):34–46, 2007.
- [Gui11] Erico Guizzo. How google’s self-driving car works. *IEEE Spectrum Online, October*, 18, 2011.
- [Gui13] Erico Guizzo. DARPA’s Rescue-Robot Showdown. <http://spectrum.ieee.org/robotics/humanoids/darpas-rescuerobot-showdown>, 2013. [Online, consultado 11 de Junio de 2015].
- [HdC13] Alejandro Herrero de Campos. Desarrollo de un sistema de control para robots móviles usando información en dos y tres dimensiones. Proyecto fin de carrera, EUITI-UPM, 2013.

- [KH04] Nathan Koenig and Andrew Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *Intelligent Robots and Systems, 2004.(IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on*, volume 3, pages 2149–2154. IEEE, 2004.
- [Kha11] Piyush Khandelwal. freenect stack ROS Package. http://wiki.ros.org/freenect_stack, 2011. [Online, consultado 27 de Febrero de 2015].
- [Kon10] Kurt Konolige. Projected texture stereo. In *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, pages 148–155. IEEE, 2010.
- [LO03] José Antonio López Orozco. *Integración y fusión multisensorial en robots móviles autónomos*. Universidad Complutense de Madrid, Servicio de Publicaciones, 2003.
- [ME10a] Eitan Marder-Eppstein. Navigation Stack. <http://wiki.ros.org/navigation>, 2010.
- [ME10b] Eitan Marder-Eppstein. Navigation Stack Setup. <http://wiki.ros.org/navigation/Tutorials/RobotSetup>, 2010. [Online, consultado 13 de Abril de 2015].
- [Mob06] Mobile Robots Inc. *Pioneer 3 Operations Manual*. 2006.
- [MTK⁺02] Michael Montemerlo, Sebastian Thrun, Daphne Koller, Ben Wegbreit, et al. Fastslam: A factored solution to the simultaneous localization and mapping problem. In *AAAI/IAAI*, pages 593–598, 2002.
- [NAS03] NASA. Mars Exploration Rover Mission: Overview. <http://mars.nasa.gov/mer/overview/>, 2003. [Online, consultado 20 de Junio de 2015].
- [NAS12] NASA. Curiosity Rover, Mars Science Laboratory. <http://www.nasa.gov/feature/jpl/nasas-curiosity-rover-team-confirms-ancient-lakes-on-mars/>, 2012. [Online; consultado 20 de Agosto de 2013].
- [Nev12] Nevada Department of Motor Vehicles. Nevada dmv issues first autonomous vehicle testing license to google. <http://www.dmvnv.com/news/12005-autonomous-vehicle-licensed.htm>, 2012. [Online, consultado 10 de Junio de 2015].
- [NKO⁺11] Keiji Nagatani, Seiga Kiribayashi, Yoshito Okada, Satoshi Tadokoro, Takeshi Nishimura, Tomoaki Yoshida, Eiji Koyanagi, and Yasushi Hada. Redesign of rescue mobile robot quince. In *Safety, Security, and Rescue Robotics (SSRR), 2011 IEEE International Symposium on*, pages 13–18. IEEE, 2011.
- [NQG⁺08] Andrew Y Ng, Morgan Quigley, Stephen Gould, Ashutosh Saxena, and Eric Berger. STAIR: STanford Artificial Intelligence Robot. <http://stair.stanford.edu/>, 2008. [Online, consultado 12 de Junio de 2015].

- [Ope] Open Source Robotics Foundation. Gazebo. <http://gazebosim.org/>. [Online, consultado 10 de Septiembre de 2015].
- [OSF] OSFR. Open Source Robotics Foundation. <http://www.osrfoundation.org/>. [Online; consultado 12 de Noviembre de 2014].
- [Pat10] Amit Patel. Introduction to A*. <http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html>, 2010. [Online, consultado 10 de Octubre de 2015].
- [Paí11] El País. Fukushima vive el peor accidente nuclear desde Chernóbil. http://internacional.elpais.com/internacional/2011/03/12/actualidad/1299884402_850215.html, 2011. [Online, consultado 21 de Junio de 2015].
- [PB15] Jose Pardeiro Blanco. Algoritmos de planificación de trayectorias basados en Fast Marching Square. Master's thesis, UC3M, 2015.
- [PBR06] R Playter, M Buehler, and M Raibert. Bigdog. In *Defense and Security Symposium*, pages 623020–623020. International Society for Optics and Photonics, 2006.
- [poi10] pointclouds.org. Point cloud library. <http://pointclouds.org>, 2010. [Online, consultado 22 de Marzo de 2015].
- [Pre13] Europa Press. Apple adquiere PrimeSense, desarrolladora del sensor Kinect de Xbox. <http://www.abc.es/tecnologia/informatica-hardware/20131125/abci-apple-primesense-compra-kinect-201311251257.html>, 2013. [Online, consultado 20 de Enero de 2016].
- [PY12] Paloma de la Puente Yusty. *Probabilistic mapping with mobile robots in structured environments*. PhD thesis, Industriales, 2012.
- [QCG⁺09] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5, 2009.
- [Reu12] Reuters. Amazon's Kiva Systems Acquisition To Put Robots To Work. http://www.huffingtonpost.com/2012/03/19/amazoncom-kiva-acquisition_n_1365512.html, 2012. [Online, consultado 11 de Junio de 2015].
- [RF11] Vincent Rabaud and Tully Foote. openni kinect ROS Package. http://wiki.ros.org/openni_kinect, 2011. [Online, consultado 16 de Marzo de 2015].
- [RLHVdlP13] Diego Rodriguez-Losada, Miguel Hernando, Alberto Valero, and Paloma de la Puente. Mobile Robots Core. <https://github.com/drodrri/MRCore>, 2013. [Online, consultado 15 de Junio de 2015].
- [RLMJG06] Diego Rodriguez-Losada, Fernando Matia, Agustin Jimenez, and Ramón Galán. Consistency improvement for slam-ekf for indoor environments. In *ICRA*, pages 418–423, 2006.

- [Rob02] Robotnik. Servicios de ingeniería robótica. <http://www.robotnik.es/>, 2002.
- [ROSa] ROS. Robot Operating System. <http://www.ros.org/>. [Online, consultado 10 de Diciembre de 2014].
- [ROSc] ROS.org. roscpp C++ API. <http://wiki.ros.org/roscpp>. [Online, consultado 10 de Noviembre de 2014].
- [ROSc] ROS.org. rospy Python API. <http://wiki.ros.org/rospy>. [Online, consultado 10 de Noviembre de 2014].
- [ROS13] ROS.org. ROS APIs. <http://wiki.ros.org/APIs>, 2013. [Online; consultado 10 de Noviembre de 2014].
- [ROS14] ROS.org. ROS Network Setup. <http://wiki.ros.org/ROS/NetworkSetup>, 2014. [Online, consultado 10 de Marzo de 2015].
- [ROS15] ROS.org. Rosaria. <http://wiki.ros.org/ROSRosaria>, 2015. [Online, consultado 8 de Febrero de 2015].
- [SFG07] Cyrill Stachniss, Udo Frese, and Giorgio Grisetti. Openslam. <http://openslam.org>, 2007. [Online, consultado 20 de Febrero de 2015].
- [SIC09] SICK. Laser scanners, LMS100-10000. <https://www.mysick.com/ecat.aspx?go=FinderSearch&Cat=Row&At=Fa&Cult=English&FamilyID=344&Category=Produktfinder&Selections=34242>, 2009. [Online; consultado 22 de Agosto de 2013].
- [SKGT12] Troy Straszheim, Morten Kjaergaard, Brian Gerkey, and Dirk Thomas. catkin ROS Package. <http://wiki.ros.org/catkin>, 2012. [Online, consultado 17 de Febrero de 2015].
- [SOGSBS⁺¹⁰] Ramón Silva Ortigoza, Rafael García Sánchez, Ricardo Barriontos Sotelo, María Aurora Molina Vilchis, Víctor Manuel Hernández Guzmán, and Gilberto Silva Ortigoza. Una panorámica de los robots móviles. *Télématique*, 6(3):1–14, 2010.
- [TBF05] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic robotics*. MIT press, 2005.
- [The14] The Centre for Speech Technology Research. The Festival Speech Synthesis System. <http://www.cstr.ed.ac.uk/projects/festival/>, 2014. [Online, consultado 18 de Julio de 2015].
- [Tom14] Aleksandar Tomović. Path planning algorithms for the robot operating system. 2014.
- [Wal01] Erik Walthinsen. GStreamer, Open source multimedia framework. <http://gstreamer.freedesktop.org/>, 2001. [Online, consultado 25 de Julio de 2015].
- [Wik15] Wikipedia. Dead Reckoning. https://en.wikipedia.org/wiki/Dead_reckoning, 2015. [Online, consultado 14 de Marzo de 2015].

- [Xat11] Xataka. Conducción autónoma y futuro de los coches. <http://www.xataka.com/automovil/conduccion-autonoma-y-futuro-de-los-coches>, 2011. [Online, consultado 10 de Junio de 2015].