UNIVERSIDAD DE MÁLAGA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INDUSTRIAL

Departamento de Ingeniería de Sistemas y Automática

Área de Conocimiento de Ingeniería de Sistemas y Automática

Proyecto final de carrera

Construcción de mapas de exteriores mediante *octrees* para un robot móvil equipado con un telémetro láser 3D

Autor **Manuel Zafra Granados**

Directores **Dr. Jesús Morales Rodríguez**
Dr. Jorge Luis Martínez Rodríguez

Titulación **Ingeniero en Automática y Electrónica Industrial**

Málaga, Abril de 2015

Dedicado a
mi familia y amigos

Resumen

La navegación autónoma de vehículos en exteriores requiere de mapas precisos de su entorno. Los árboles octales (*octrees*) son una estructura de datos que proveen una eficiente representación volumétrica del espacio tridimensional (3D). En este proyecto fin de carrera se propone incorporar la construcción de mapas 3D mediante *octrees* en el robot móvil de exteriores Andábata.

La aplicación *Octomap* presenta un sistema de construcción de mapas con *octrees* que es capaz de fusionar de forma probabilística medidas 3D tomadas desde un robot móvil. En este proyecto, Octomap se utilizará desde el sistema operativo ROS (*Robotic Operating System*) con las medidas proporcionadas por un telémetro láser 3D de 360º de campo de visión.

Acrónimos

2D	Bidimensional
3D	Tridimensional
BSD	Berkeley Software Distribution (distribución de software berkeley)
CIR	Centro Instantaneo de Rotación
DSP	Digital Signal Processor (Procesador digital de señales)
GPS	Global Positioning System (Sistema de posicionamiento global)
HAL	Hardware Abstraction Layer (Capa de abstracción del hardware)
I+D	Investigación y Desarrollo
IP	Internet Protocol (Protocolo de internet)
IMU	Inertial Measurement Unit (Unidad de medida inercial)
OpenCV	Open Source Computer Vision (Visión por computador de código abierto)
P2P	Peer to Peer (De igual a igual)
PCL	Point Cloud Library (Libreria de la nube de puntos)
PWM	Pulse Width Modulation (Modulación por ancho de pulsos)
ROS	Robotic Operating System (Sistema operativo para robots)
RPC	Remote Procedure Call (Llamada a procedimiento remoto)
RSF	Robotic Software Framework (Marco de trabajo para software para robots)
SLAM	Simultaneous Localization And Mapping (Localización y mapeado simultáneos)
SSD	Solid State Drive (Unidad de estado sólido)
SSH	Secure SHell (intérprete de órdenes segura)
URDF	Unified Robot Description Format (Formato unificado de descripción de robots)
USB	Universal Serial Bus (Bus serie universal)

Índice general

Capítulo 1

INTRODUCCIÓN

1.1. Motivación personal

Considero el desarrollo del proyecto fin de carrera como un reto y una oportunidad para aprender sobre un ámbito de la ingeniería de tu propia elección. Es por ello que busqué un proyecto en el departamento de Ingeniería de Sistemas y Automática y, en concreto, decidí realizar el presente porque los trabajos que encuentras sobre ROS resultan muy interesantes. Además, supone un reto personal enfrentarse a un nuevo robot y al uso de herramientas de desarrollo de las que no tenía conocimientos previos.

Este proyecto representa una buena oportunidad para trabajar con un robot equipado con un telémetro láser 3D y con ROS. Este sistema operativo ha sido una herramienta muy importante, ya que esta muy extendido y, al tratarse de software libre, existe gran cantidad de información, paquetes, tutoriales y ayuda por parte de la comunidad que lo usa. Considero, además, muy atractivo el objeto final de este proyecto, ya que la construcción de mapas es una parte muy importante de la robótica.

Finalmente pienso que si en un futuro quiero centrar mi carrera profesional en proyectos de I+D, es un gran acierto hacer un proyecto final de carrera de este tipo.

1.2. Objetivos

Este proyecto se enmarca dentro del Proyecto de Investigación de Excelencia de la Junta de Andalucía P10-TEP-6101-R, "Navegación autónoma de un robot móvil 4x4 en entornos naturales mediante GPS diferencial y telémetro láser tridimensional" [1].

El objetivo final de este proyecto final de carrera es conseguir que el robot móvil Andábata realice mapas 3D. Dado que el robot está pensado para moverse por exteriores no es suficiente un mapa 2D y es necesario conocer la ocupación del espacio tridimensional. La mayoría de mapas 3D ocupan mucho espacio en memoria, por eso se ha decidido utilizar el paquete Octomap para ROS.

Los objetivos concretos de este proyecto final de carrera son:

1. Descargar e instalar el sistema operativo ROS en Andábata, así como el paquete Octomap, comprender su funcionamiento y ponerlo en marcha.
2. Integrar un telémetro láser 3D en ROS y conseguir la toma de medidas del mismo.
3. Incorporar Octomap y el escáner láser al sistema de navegación de Andábata.
4. Programar el sistema para poder realizar mapas 3D mientras el robot está en movimiento.
5. Probar la construcción de mapas con el robot móvil mientras se desplaza.

1.3. Fases del trabajo

Las tareas realizadas durante el proyecto han sido las siguientes:

1. Familiarización con el robot móvil Andábata.
2. Estudio de la aplicación Octomap y del sistema operativo para robots ROS.
3. Instalar los sistemas operativos Ubuntu y ROS en Andábata.
4. Descargar el código abierto de Octomap y los repositorios de datos disponibles.
5. Puesta en marcha de Octomap en ROS con datos de repositorio de entornos exteriores.
6. Integración del escáner láser 3D en ROS.
7. Incorporar Octomap y el telémetro láser en el sistema de navegación del robot móvil Andábata para la realización de mapas 3D.
8. Realizar pruebas de construcción de mapas con Andábata.
9. Elaboración de la memoria escrita del trabajo realizado.
10. Preparar la presentación ante el tribunal examinador.

1.4. Estructura del proyecto

El presente proyecto consta de siete capítulos y tres anexos. Los primeros capítulos introducen las herramientas y los recursos que se van a utilizar. Después se explica la estructura software creada. Finalmente se encuentran los anexos.

En el segundo capítulo se introduce el sistema operativo ROS junto con algunas de sus herramientas más destacables y comúnmente utilizadas. Se describe su historia y el

porqué de su elección.

El tercer capítulo explica la aplicación Octomap, centrándose en su funcionamiento y las posibilidades que ofrece. También se muestra cómo está integrado Octomap en ROS.

En el cuarto y quinto capítulos se describe el hardware del que se dispone, comenzando en el capítulo cuarto por el robot Andábata y continuando en el capítulo quinto con el telémetro láser 3D.

En el capítulo sexto se explica el software diseñado y se muestra la estructura final alcanzada. Además se describe con detalle cada uno de los programas que estarán en ejecución cuando el robot Andábata se encuentre en funcionamiento creando mapas.

El capítulo final contiene experimentos, las conclusiones alcanzadas en el proyecto y los posibles trabajos futuros.

El anexo A enumera los pasos que se deben llevar a cabo para la correcta utilización del robot Andábata.

El anexo B muestra cómo se ha configurado la computadora del robot para poder acceder a algunas aplicaciones necesarias y definir ciertos comportamientos.

El anexo C contiene el código final implementado en el robot Andábata.

Capítulo 2

EL SISTEMA OPERATIVO ROS

2.1. Introducción

El desarrollo de software para robots se está convirtiendo en un proceso cada vez más complicado, en parte debido a que los usos de la robótica no paran de crecer, cada vez en ámbitos más dispares. Para cada uso diferente se requiere un tipo de robot diferente, requiriendo distintos tipos de hardware, lo que origina que se necesite un software nuevo para cada robot, dificultando en gran medida la reutilización del código. Otro problema aparece en la cantidad de niveles distintos en los que el desarrollador de robots tiene que trabajar, a menudo teniendo que programar desde los *drivers* de los actuadores, hasta la percepción, o el razonamiento de alto nivel del robot. Para superar estos problemas, la industria robótica ha creado diferentes entornos de trabajo para facilitar el desarrollo de robots. Cada uno de estos entornos intenta adaptar el desarrollo de los robots a las necesidades del programador.

El nombre de ROS viene de **Robot Operating System**. ROS puede clasificarse como un entorno de desarrollo en robótica (*RSF*) y ofrece herramientas y librerías para el desarrollo de sistemas robóticos. A pesar de su nombre, ROS no es un sistema operativo propiamente dicho (de hecho funciona sobre Linux), sino una infraestructura de desarrollo, despliegue y ejecución de sistemas robóticos, así como una gran federación de repositorios que ofrecen paquetes con software de todo tipo. A diferencia de la mayoría de los *RSFs* existentes, ROS pretende dar una solución integral al problema de desarrollo de robots aportando soluciones a las áreas de:

- **Capa de abstracción de Hardware (*HAL*)** y reutilización e integración de robots y dispositivos, encapsulandolos tras interfaces estables y manteniendo las diferencias en archivos de configuración.
- **Algoritmos de robótica** con bajo acoplamiento: desde algoritmos de bajo nivel de control, cinemática, *SLAM*, hasta algoritmos de alto nivel como planificación o aprendizaje. También otros muchos más específicos como lograr que el robot X coja un objeto o el robot Y se enchufe a la red eléctrica.
- ROS proporciona un mecanismo de comunicaciones distribuido entre nodos del sistema robótico. Un nodo es cualquier pieza de software del sistema (desde un algoritmo

de aprendizaje hasta un *driver* para el manejo de un motor). El objetivo de este sistema es doble: encapsulado/abstracción/reutilización de software; y ubicuidad, es decir, independencia de dónde el nodo esté localizado (un sistema robótico puede tener muchos procesadores). Estos nodos se comunican entre ellos mediante mecanismos de paso de mensajes *RPC* o *Publish/Subscribe*, *Service lookup*, etc. Permite crear arquitecturas *P2P* de componentes robóticos distribuidos.

- **Herramientas de desarrollo**, despliegue y monitorización de sistemas robóticos.
- **Administración de paquetes** de forma autonoma. En la actualidad existen mas de 2000 paquetes con diversas utilidades.

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

ROS está basado en una arquitectura de grafos donde el procesamiento toma lugar en los nodos que pueden recibir, mandar y multiplexar mensajes de sensores, control, estados, planificaciones y actuadores, entre otros.

A pesar de la importancia de la baja latencia y la rápida reacción en control de robots, ROS, por si mismo, no es un sistema operativo en tiempo real, aunque es posible integrar ROS con codigo de tiempo real.

2.1.1. Historia

ROS fue inicialmente desarrollado en 2007 bajo el nombre de *switchyard* por *Stanford Artificial Intelligence Laboratory* en apoyo al proyecto *STAIR* (*Stanford AI Robot*). Desde 2008 hasta 2013, el desarrollo se realizó principalmente en *Willow Garage*, una institución de investigación en robotica. Durante ese tiempo, investigadores y mas de veinte instituciones colaboraron con los ingenieros de Willow Garage. En febrero del 2013, la administración de ROS se trasladó al *Open Source Robotics Foundation*.

Las versiones de ROS suelen ser referidas por un sobrenombre, en vez de por un numero, cuya inicial sigue el alfabeto. Algunas de estas no son compatibles con otras. Las que se han lanzado hasta la fecha, desde la más actual a la primera versión son:

- Indigo Igloo - 22/Julio/2014 (ver figura 2.1)
- Hydro Medusa - 04/Septiembre/2013
- Groovy Galapagos - 31/Diciembre/2012
- Fuerte - 23/Abril/2012
- Electric Emys - 30/Agosto/2011

- Diamondback - 02/Marzo/2011
- C Turtle - 03/Agosto/2010
- Box Turtle - 01/Marzo/2010
- ROS 1.0 - 22/Enero/2010

Figura 2.1: Logo de ROS Indigo Igloo.

Desde 2012 se realiza anualmente un congreso dedicado a ROS (ROSCon), donde se exponen los avances mas significativos en el entorno de ROS, y una escuela de verano dirigida a alumnos e investigadores que quieran introducirse en dicho entorno.

2.1.2. ¿Por qué ROS?

Un motivo para emplear ROS es su gran modularidad, ya que permite usar las partes que se quieran e implementar aquellas que se necesiten. Por ejemplo, si existe un paquete de teleoperación que se quiere utilizar, pero con un mando diferente, se puede usar el resto del paquete y crear un nodo que lea y publique las instrucciones de ese mando.

También es fácil encontrar software dedicado a hacer las tareas que se puedan necesitar, ya que ROS dispone de una comunidad muy activa y participativa que crean y mantienen gran cantidad de paquetes.

El corazón de ROS esta publicado bajo licencia BSD, que es una licencia muy abierta que permite un uso personal, comercial y en productos de código cerrado. No obstante, cada paquete consta de un tipo de licencia propio, con lo que es fácil encontrar los paquetes que se ajustan a diferentes necesidades.

2.2. Estructura

ROS está diseñado para ser lo más modular posible en todos sus ambitos, lo que facilita la reutilización, reimplantación en un nuevo robot y organización del software.

2.2.1. Estructura del software

El principio básico para un sistema operativo de robots es poder ejecutar un gran número de procesos en paralelo que tienen que ser capaces de intercambiar datos de forma síncrona y asíncrona. Por otra parte, necesita administrar y asegurar un acceso eficaz a los recursos del robot. Esto se consigue con la modularización que se explica a continuación:

- **Nodos**: es la instancia de un ejecutable, un proceso que realiza un cálculo o cómputo. Los nodos pueden combinarse entre sí utilizando *topics*, *services* y el *parameter server*. Estos nodos están diseñados para operar a una escala mínima, donde un sistema robótico normalmente estará compuesto por varios nodos. El uso de nodos proporciona diferentes ventajas a la arquitectura de ROS.
  Un caso especial de nodo es el ***master***, que actúa de servidor central de la arquitectura y permite que los nodos se localicen entre sí para que puedan comunicarse mediante *topics* y *services*. También incluye el *parameter server*.

- **Mensajes**: los nodos se comunican entre sí mediante la publicación de mensajes que transmiten mediante *topics* o *services*. Un mensaje es una estructura de diferentes tipos de datos. Los tipos básicos son enteros, decimales, booleanos, etc. También soportan vectores y estructuras de datos.

- ***Topics***: son las vías de intercambio de mensajes entre nodos de forma asíncrona. Estos *topics* utilizan un sistema de publicador/subscriptor de forma anónima. En general, los nodos no saben con qué otro nodo se están comunicando (vease figura 2.2). En vez de eso, los nodos que quieren acceder a unos datos concretos se suscriben al *topic* relevante, y los nodos que quieren generar datos publican en el *topic* relevante. Como muestra la figura 2.3 puede haber varios nodos publicadores o subscritos para un mismo *topic*.

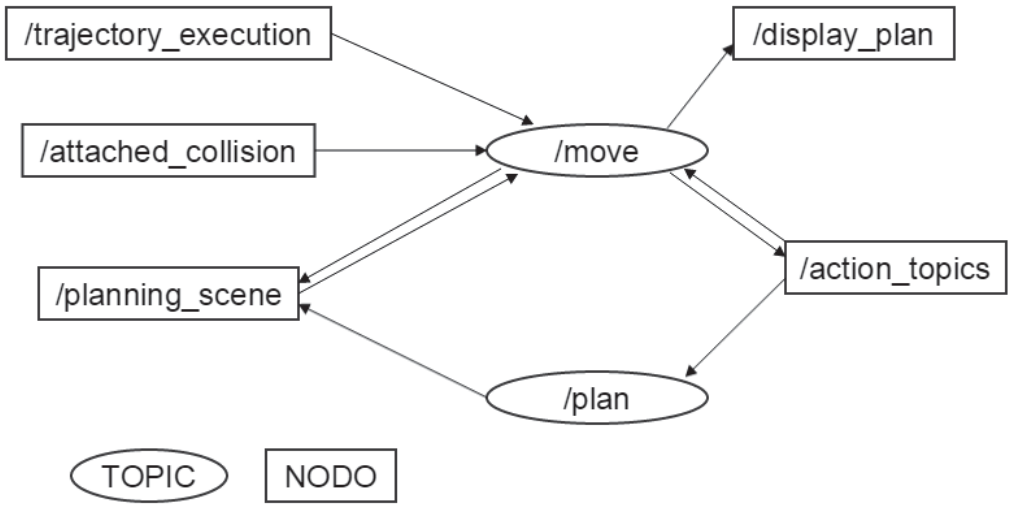

Figura 2.2: Ejemplo de un diagrama de grafos en ROS.

- ***Services***: son el medio de intercambio de mensajes de forma síncrona. Se definen mediante un mensaje de petición y otro de respuesta, y utiliza el modelo de comunicación cliente/servidor (ver figura 2.3). El nodo que funciona como servidor es el que ofrece el servicio, y el nodo cliente, el que llama al servicio enviando un mensaje de petición y esperando la respuesta.

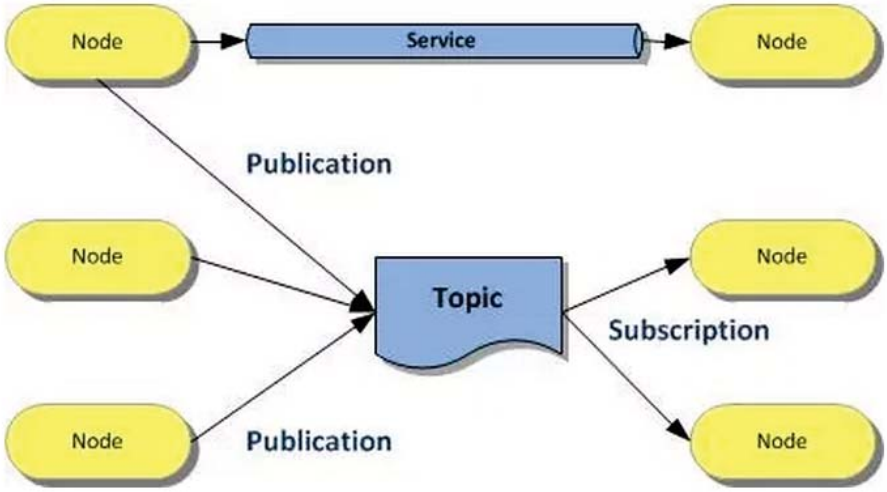

Figura 2.3: Diagrama de funcionamiento de *topics* y *services*.

- ***Parameter server***: es un diccionario nombre-valor compartido que puede ser accedido en toda la red de ROS. Los nodos pueden utilizar este servidor para guardar u obtener parámetros en tiempo de ejecución. Está pensado para almacenar datos estáticos y no demasiado complejos, como por ejemplo parámetros de configuración.

2.2.2. Organización de archivos

Para facilitar su reutilización y difusión, el código generado para ROS se agrupa en *Packages* y *Stacks*. A continuación se describen ambas estructuras de organización:

- ***Packages***: El software de ROS está organizado en *packages* o paquetes. Cada paquete posee una función propia permitiendo una arquitectura modular que facilita el uso del software implementado. Un paquete puede contener diferentes nodos ROS, librerías, archivos de configuración, software de terceras partes, etc. Esta organización facilita la reutilización del código. En caso de que se quiera diseñar otro robot con diferente funcionalidad podrían reutilizarse los paquetes que fueran útiles e implementar sólo los que faltan.

- ***Stacks***: Los paquetes ROS están organizados en *Stacks* para facilitar el proceso de distribución. Un *Stack* engloba *packages* que juntos suministran una funcionalidad. Por ejemplo, puede existir el *stack* Automoción que suministra todos los paquetes que permiten que el robot se mueva.

2.3. Herramientas

ROS contiene una colección de herramientas y algoritmos para la programación, simulación y ejecución de tareas de robots. A continuación se presentan algunos de los mas usados.

2.3.1. Stage

Stage es un entorno de simulación 2D multi-robot. Es capaz de simular una gran población de robots moviéndose y percibiendo un entorno 2D con un costo computacional relativamente bajo, debido a que utiliza modelos simples de los robots. Tiene incorporados varios modelos de sensores, incluyendo sonar, escáneres laser, codificadores, etc. La mayoría de los sistemas no detectan si están siendo ejecutados en Stage o en un sistema físico real (a menos que ese sea su propósito).

2.3.2. Gazebo

Gazebo es un simulador 3D cinemático, dinámico y multi-robot, que permite realizar simulaciones de robots articulados en entornos complejos, interiores o exteriores, realistas y tridimensionales (ver figura 2.4). Los robots pueden interactuar con el mundo (pueden coger y empujar cosas, rodar y deslizarse por el suelo) y viceversa (les afecta la gravedad y puede colisionar con obstáculos del mundo).

Figura 2.4: Ejemplo de simulación en Gazebo del robot Baxter.

Además, permite desarrollar y simular modelos de robots propios (*URDF*) e incluso cargarlos en tiempo de ejecución. También tiene la posibilidad de crear escenarios (mundos) de simulación, variando gran variedad de características de éstos. Contiene diversos *plugins* para añadir sensores al modelo del robot y simularlos, como sensores de odometría, de fuerza, de contacto, telémetros láser y cámaras estéreo.

2.3.3. Rviz

Rviz es un entorno de visualización 3D que permite combinar en una misma pantalla modelos de robots, datos de sensores (cámara, láser, etc) y otros datos 3D.

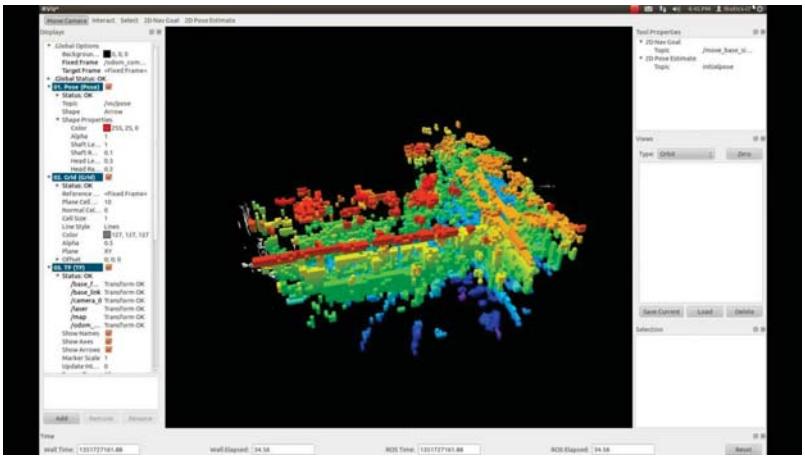

Figura 2.5: Panel principal de Rviz.

Posibilita que, prácticamente, cualquier plataforma robótica pueda ser representada en 3D, respondiendo rapidamente a lo que le ocurre en el mundo real, y, como se puede ver en la figura 2.5, mostrando lecturas de sensores.

2.3.4. tf

El paquete tf permite al usuario hacer un seguimiento de múltiples ejes de coordenadas en el tiempo (ver figura 2.6). tf mantiene la relación entre dichos ejes en una estructura de árbol almacenada en el tiempo, y permite al usuario transformar puntos, vectores, etc. entre dos sistemas de coordenadas en cualquier instante deseado.

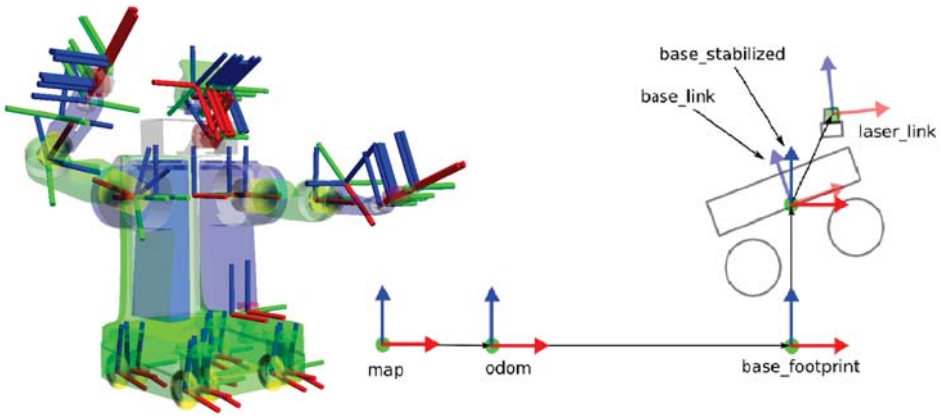

Figura 2.6: Ejemplos de uso de ejes de coordenadas.

2.3.5. OpenCV

OpenCV es software de código abierto de visión por computador y una libreria de software sobre aprendizaje. Fue construido para proporcionar una infraestructura común para aplicaciones de visión por computador y para acelerar el uso de la percepción en los productos comerciales.

2.3.6. Point Cloud Library

PCL es un entorno para el procesado de nubes de puntos 3D. Contiene algoritmos para gran número de tareas, como el filtrado, la estimación de características, la reconstrucción de superficies, el emparejamiento de nube de puntos o la segmentación.

Capítulo 3

LA APLICACIÓN OCTOMAP

3.1. Introducción

Octomap es una librería que implementa la creación de mapas 3D mediante un planteamiento en rejilla, proporcionando las estructuras de datos y los algoritmos en C++ necesarios. La implementación está basada en *octrees* o árboles octales [2].

El modelado 3D del entorno es básico para muchos campos de la robótica, por lo que existe gran variedad de métodos para afrontarlo. Las características deseadas serían:

- **Representación probabilística**, ya que para crear un mapa se usan sensores que tienen un error implícito. Además existen otras fuentes de fallo en la medida, como errores aleatorios u objetos dinámicos. Para crear un mapa con medidas ruidosas, el mejor medio es tomarlas de forma probabilística, donde múltiples medidas ruidosas se pueden fusionar para crear una estimación robusta del entorno. También es importante poder fusionar medidas de diferentes sensores y desde diferentes robots.

- **Áreas desconocidas**, la representación de estas áreas es necesaria, ya que las zonas sin explorar deben ser evadidas durante la navegación y visitadas durante la exploración.

- **Eficiente**, se necesita un mapa para navegar y planificar una trayectoria, por esta razón, un mapa debe de ser rápido en el acceso a datos y eficiente en el uso de memoria.

Las opciones más conocidas se muestran en la figura 3.1, y son: la nube de puntos, la cual carece de áreas desconocidas y además utiliza una gran cantidad de memoria, y mapas de elevación y multinivel, que son eficientes en el uso de la memoria, pero que no son capaces de representar formas aleatorias, ya que toman solo la máxima altura de cada punto.

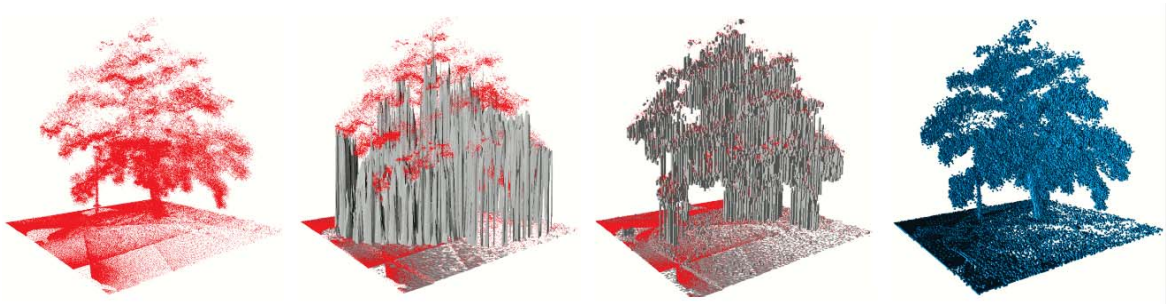

Figura 3.1: Ejemplo de entorno modelado con: a) *Nube de puntos* b) *Mapa de elevación* c) *Mapa multinivel* d) *Octrees*.

3.2. Sistema de creación de mapas Octomap

Para dar flexibilidad a la resolución y a la zona de interés asignadas Octomap utiliza una estructura en árbol. Se realiza una estimación probabilística de la ocupación para garantizar la capacidad de actualización y de hacer frente a ruidos del sensor y el entorno. Además, los métodos de compresión aseguran la compacidad de los modelos resultantes.

3.2.1. Octrees

Un *octree*, como se ve en la figura 3.2, es una estructura de datos jerárquica que se usa para la subdivisión del espacio en 3D. Cada nodo en un octree representa el espacio contenido en un volumen cúbico, generalmente llamado *voxel*. Este volumen se subdivide repetidamente en ocho sub-volúmenes hasta que se alcance un tamaño mínimo dado. El tamaño mínimo de *voxel* determina la resolución del *octree*. Puesto que un *octree* es una estructura de datos jerárquica, se puede cortar en cualquier nivel para obtener una subdivisión con menos resolución.

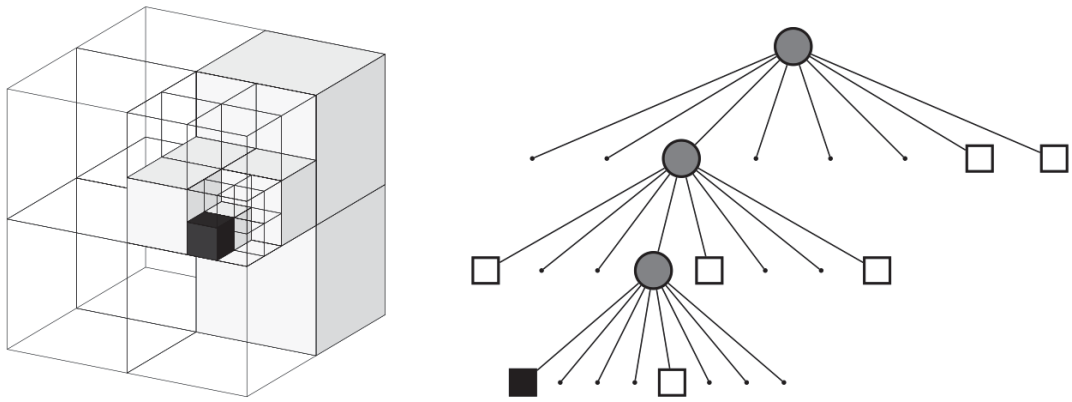

Figura 3.2: Ilustración de *octree* y su estructura en arbol.

En el modo más básico, los *octrees* son usados para almacenar propiedades booleanas, que en el caso del mapeado en robótica suele ser la ocupación del espacio. Con esta configuración cada nodo del *octree* puede estar inicializado a ocupado, espacio libre o área desconocida. Para resolver este problema, las zonas de espacio vacío se representan explícitamente, son las que se encuentran entre el sensor y el punto final captado por el mismo.

Para disminuir el espacio utilizado en memoria por el mapa se podan los hijos de un nodo si todos ellos tienen el mismo estado. Pero las medidas suelen tener ruido y el entorno puede estar cambiando continuamente, por lo que normalmente, un sistema booleano de ocupación no es suficiente, la ocupación debe ser modelada de forma probabilística. Sin embargo, esto choca con la idea de podar hijos con igual estado. Octomap combina de forma compacta ambos procedimientos, eliminando la información estadística cuando ésta es cercana a cero o uno.

En términos de acceso a los datos, los octrees tienen mayor complejidad que una cuadricula 3D corriente, sin llegar a ser un problema, ya que con una profundidad máxima de 16 nodos se puede cubrir un cubo de 655.36 m de lado con una resolución de 1 cm.

3.2.2. Fusión probabilística de sensores

Las lecturas de los sensores son integradas de forma probabilística. La probabilidad de que un nodo esté ocupado dada una nueva medida de un sensor depende de la estimación anterior del estado del nodo, de la probabilidad a priori de que dicho nodo este ocupado (normalmente se toma 0,5) y de la probabilidad de que el nodo este ocupado dada una medida positiva (este parámetro depende del sensor). De este modo se guarda la probabilidad de ocupación en cada *voxel* en lugar de un dato booleano. Cuando un mapa 3D es usado para navegación se aplica un umbral en la probabilidad de que un nodo este ocupado para determinar su ocupación o no, de este modo, si supera el umbral se tomara como espacio ocupado, y si no lo supera, como espacio libre.

Se definen un límite superior e inferior de actualización del estado de un nodo, con lo que se evita que para cambiar el estado de un nodo se necesiten tantas medidas como lo definieron (es decir, si 10 medidas hicieron que un nodo se definiera como ocupado, sin este mecanismo, harían falta más de 10 medidas de voxel libre para determinarlo como espacio libre). Debido a esta política de actualización se consigue que el modelo se adapte rápidamente a cambios en el entorno.

3.2.3. Consultas multi-resolución

Cuando se integra una medida en la estructura de un mapa, la actualización se realiza solo en los nodos finales, o nodos hoja, pero como el *octree* es una estructura jerárquica se puede hacer uso de los nodos interiores para habilitar las consultas multi-resolución, como muestra la figura 3.3.

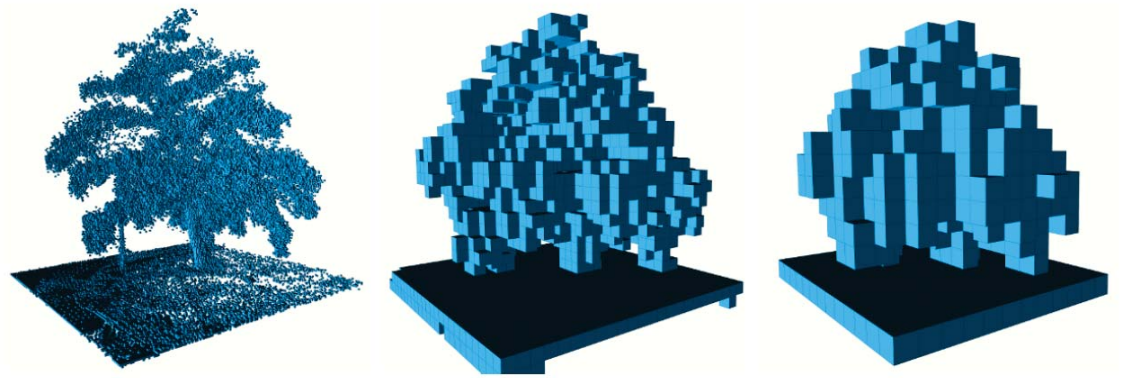

Figura 3.3: Entorno representado a diferentes profundidades de la estructura para mostrar la multi-resolución.

Existen varias políticas para decidir el estado de un nodo padre dados los estados de sus ocho nodos hijos. Se podría escoger la media aritmética de las probabilidades de ocupación de los hijos, o la mayor probabilidad de los nodos hijo. Esta última política de decisiones es más conservadora y establece una seguridad mayor para la navegación. Asímismo establece que la probabilidad de que un nodo padre este ocupado es igual a la mayor de las probabilidades de sus hijos, como consecuencia de lo cual, se pueden dar casos en los que se determine una zona desconocida como ocupada.

En Octomap, tras varias medidas con el mismo resultado (ocupado o libre) el nodo se vuelve estable. Si todos los hijos de un nodo son estables y con el mismo estado (la probabilidad de que estén ocupados es cercana a 1 ó 0), estos son eliminados y solo serán regenerados si una medida contradice su estado. Con este método se logra un gran ahorro de memoria.

3.3. Octomap en ROS

Octomap está totalmente integrado en ROS. Se puede instalar el paquete Octomap con sólo la librería Octomap, para utilizarla implementando algoritmos propios. Esto daría acceso a todas las clases y funciones que ésta tiene para utilizarlas, modificarlas y adaptarlas a las necesidades de cada cual.

Otra opción es instalar el paquete *octomap_server* con el que se dispone de un servidor ya implementado para crear mapas 3D. El paquete tiene varios nodos implementados:

- *octomap_server_node* que lee datos del tipo PointCloud2 procedentes de sensores, simulaciones, grabaciones, etc. y crea con ellos, de forma incremental, el mapa 3D.
- *octomap_saver*, cuya función es la de guardar en un archivo el mapa que se ha creado.

- *octomap_server_multilayer*, es una extensión de octomap_server que publica mapas 2D multicapa que se utiliza para la navegación.

La ultima opción es instalar el stack *octomap_mapping* que incluye los dos paquetes anteriores además de los siguientes:

- *octomap_msgs* que se encarga de manejar los mensajes de la librería.
- *octomap_ros* que convierte entre los tipos nativos de ROS y Octomap.
- *octomap_rviz_plugins* que añade a *rviz* funciones de visualizacíon para los tipos creados por la librería Octomap.
- *octovis*, que es una herramienta de visualización para Octomap.

Capítulo 4

EL ROBOT MÓVIL ANDÁBATA

4.1. Introducción

En este capítulo se introduce el hardware del robot móvil Andábata, enumerándose las características más relevantes para este proyecto fin de carrera. También se realiza el modelado cinemático para poder obtener la odometría.

Figura 4.1: Fotografía del robot Andábata.

Como se puede observar en la figura 4.1, Andábata es un robot todoterreno para la navegación en exteriores de dimensiones reducidas (49x64x76 cm) y cuyo peso es de 41 kg [1]. El robot tiene tres niveles:

1. El nivel inferior, en el que se alberga una batería de iones de litio de 8 celdas con una capacidad total de 7.2 Ah, que proporciona una tensión continua de hasta 29.6 V. Cada celda es doble, montadas en paralelo, proporcionando cada una de ellas 3.7 V y 0.9 Ah. El peso de la batería es de 7 kg y sus medidas son 59 x 16 x 9 cm.

2. El nivel intermedio contiene la controladora de los motores de las ruedas (ver sección 4.2) y la computadora. Las características de la computadora se resumen en: placa ASRock Z87M Extreme4; microprocesador Intel Core i7 4771 (3.5 GHz, 8 MB cache); 16 GB de memoria RAM que opera a una frecuencia de 1333 MHz; y disco duro SSD Samsung 840 Evo con 250 GB de capacidad.

3. El nivel superior es utilizado para colocar sensores. Se ha instalado un telémetro láser 3D UNO-Motion (ver capítulo 5) y un teléfono móvil Samsung Galaxy S4 Active I9295. El teléfono envía a la computadora datos de sus sensores: GPS, inclinómetros, giróscopos y magnetómetros.

Andábata dispone de una tableta Samsung Galaxy Tab 3 P5200 para la teleoperación, donde se ejecutarán las aplicaciones Android *"Andábata"* (proveniente de modificar la aplicación *"Mover-bot"* [3]) y *"ROS Android Sensors Driver"* [4].

4.2. Sistema motriz

Andábata se desplaza con tracción por deslizamiento (*skid-steering*). Cada rueda consta de un motor, una caja reductora, un sensor Hall y una suspensión independiente de las demás ruedas.

- Las ruedas son de 20 cm de diámetro y contienen cámara de aire y llanta. Están conectadas a los motores mediante una reductora epicicloidal de 1:36 como relación de transmisión (ver figura 4.2).

Figura 4.2: La rueda y la tapa de la reductora a la izquierda y la reductora epicicloidal a la derecha.

- Los motores son de conmutación electrónica (*brushless*) de 100 W con una velocidad nominal de 3580 rpm. Cada motor consta de un sensor Hall de 14 pulsos por vuelta

y 2 canales, que están instalados en el eje del motor, antes de la reductora, por lo que por cada vuelta de la rueda se obtienen 504 pulsos. Dado que el radio de las ruedas es de 10 cm, cada pulso se corresponde con 1.25 mm.

- Cada rueda consta de un sistema de suspensión pasiva con una guía lineal de 6,5 cm de recorrido, regulables mediante la apropiada elección de dos muelles.

- La velocidad lineal de la rueda, como se puede observar en la figura 4.3, puede ser calculada, en la mayor parte de su rango, con la formula:

$$vel\_lineal = 1{,}2414 * PWM - 8{,}00(mm/s)$$

  Se puede observar una saturación a partir de una señal PWM de valor 500, de esta forma, la velocidad máxima del vehículo en línea recta es 600 mm/s.

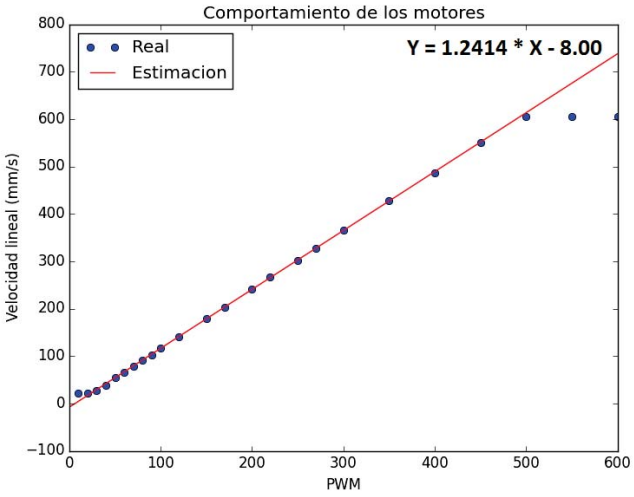

Figura 4.3: Estimación de la velocidad lineal de las ruedas en función del PWM aplicado.

- A la controladora de Andábata se accede por medio del puerto de comunicación RS232 configurado con los siguientes parámetros: 38400 baudios, 8 bits, sin paridad, 1 bit de parada y sin control de flujo.

  Todos los comandos comienzan con dos letras mayúsculas y deben terminar con el carácter ASCII 10, 11 ó 13. Los comandos disponibles están explicados en la tabla 4.1. Si el comando es correcto la controladora devolverá el carácter ASCII 13 y no contestará en caso contrario.

4.3. Sistemas de comunicación

La figura 4.4 muestra un esquema de las conexiones entre los distintos componentes del robot. El enrutador Wifi (router) conecta la tableta y el móvil mediante Wifi. El telémetro

Comando	Descripción
EN	Habilitar la controladora.
DI	Deshabilitar la controladora.
RB	Leer el estado de la batería. La respuesta es de la forma +XXXXXXmV, donde XXXXXX es el voltaje de la batería en milivoltios.
SV+XX+YY	Establecer la velocidad deseada. Los caracteres XX e YY codifican el PWM deseado de las ruedas derechas e izquierdas respectivamente. Si se desea una velocidad negativa el signo "+" se cambia por el signo "-". Las comandas de PWM están codificadas según la formula: PWM deseado = 49*(X1 – 48) + (X2 - 48) siendo X1 y X2 los valores ASCII del carácter de mayor y menor peso respectivamente. Por lo que si se quiere un PWM determinado, la formula para obtener X1 o Y1 y X2 o Y2 sería: X1 o Y1 = PWM / 49 + 48 (entero sin decimales) X2 o Y2 = PWM – (PWM / 49) * 49 + 48
RP	Leer la posición de los sensores Hall del lado derecho e izquierdo. La controladora responderá enviando los pulsos de las ruedas en el formato +XXXXXX+YYYYYY. Además se ponen a cero estos contadores.

Cuadro 4.1: Comandos para la controladora de los motores.

láser se comunica con el *router* mediante Ethernet. La computadora de a bordo también está conectada mediante Ethernet con el enrutador, lo que, aparte de permitirle recibir datos del láser, abre la posibilidad de conectar la computadora con otra computadora para manejarla mediante escritorio remoto. Para ello el telémetro láser, la computadora y el enrutador deben de estar en la misma red local, es decir, su IP debe variar solo en el último número. Así, el telémetro láser tiene la IP 192.168.0.10, el enrutador la 192.168.0.1 y a la computadora se le ha asignado la IP 192.168.0.2.

Además, el teléfono móvil está conectado a la computadora mediante un puerto USB. En otros dos puertos USB están conectados las controladoras de las ruedas y de la base del telémetro láser.

4.4. Modelado cinemático aproximado

En el sistema de guiado por deslizamiento (*skid-steer*), se controla la velocidad de las ruedas derecha e izquierda de forma independiente, al igual que los de tracción diferencial, por ello se puede obtener un modelo cinemático aproximado basado en la correspondencia con estos [5]. Dicho modelo proporcionará una aproximación de la velocidad lineal y angular del vehículo, que se podrá integrar para obtener los incrementos de posición del robot.

Cuando la conducción diferencial se aplica a un robot con un solo par de ruedas, la asunción de no deslizamiento no difiere mucho de la realidad. Sin embargo, para el tipo de desplazamientos que se realizan con un vehículo *skid-steer* donde las ruedas están alineadas con la longitudinal del vehículo, se necesita que las ruedas deslicen para girar.

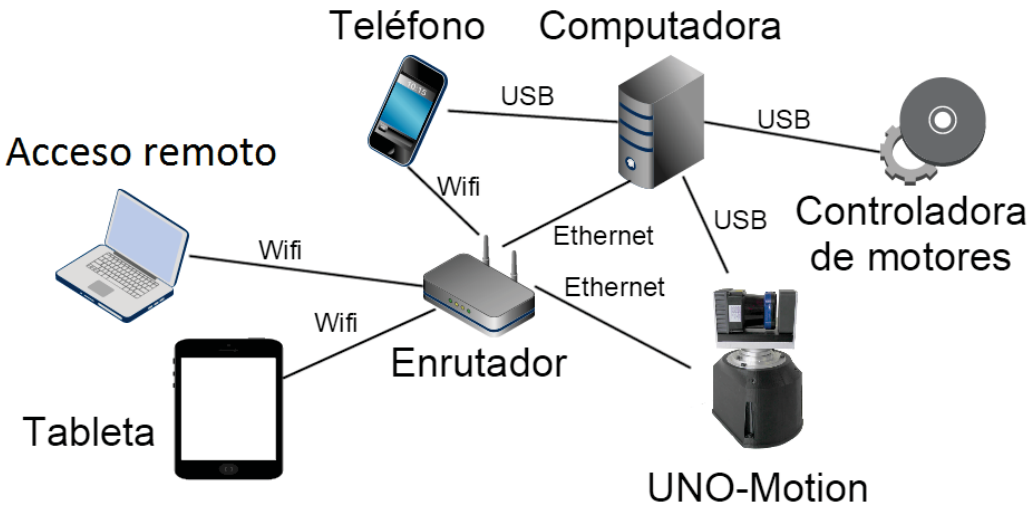

Figura 4.4: Esquema simplificado de las conexiones de comunicaciones en el robot Andábata.

El sistema de referencia del robot, que se puede ver en la imagen 4.5, se posiciona en el centro geométrico de las ruedas del robot a una altura nula sobre el suelo. El eje Y se establece paralelo al eje longitudinal del vehículo y en sentido del avance positivo del mismo, el eje Z se posiciona verticalmente con sentido positivo hacia arriba y el eje X paralelo al eje transversal del vehículo.

Los vehículos del tipo *skid-steer* se controlan mediante dos señales de entrada, que son las velocidades lineales de las ruedas derechas e izquierdas (V_d, V_i). Entonces la cinemática directa sobre el plano horizontal se puede representar con:

$$\begin{pmatrix} v_x \\ v_y \\ \omega_z \end{pmatrix} = A \begin{pmatrix} V_i \\ V_d \end{pmatrix} \tag{4.1}$$

donde $v = (v_x, v_y)$ es la velocidad de traslación del vehículo expresada en los ejes propios del robot (ver figura 4.5) y ω_z es la velocidad angular en el eje Z. La ecuación 4.2 expresa la matriz A en el caso de que se tome un modelo simétrico aproximado:

$$A = \frac{\mu}{2x_{CIR}} \begin{pmatrix} 0 & 0 \\ x_{CIR} & x_{CIR} \\ -1 & 1 \end{pmatrix} \tag{4.2}$$

donde, como se puede ver en la figura 4.5, x_{CIR} es la posición de las ruedas de un vehículo de solo dos ruedas cuyo comportamiento sea similar, y μ es un factor de corrección lineal.

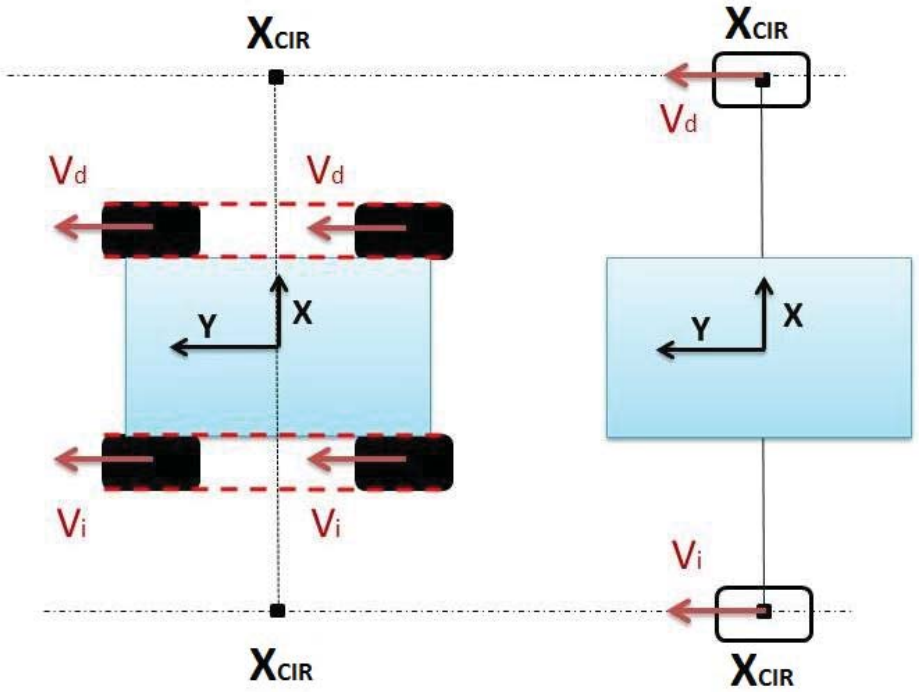

Figura 4.5: Equivalencias de *skid-steer* con la tracción diferencial.

Para realizar la identificación de los parámetros del modelo se llevan a cabo dos experimentos, uno para determinar x_{CIR} y otro para calcular μ.

4.4.1. Coeficiente rotacional (x_{CIR})

El experimento consiste en hacer girar el robot sobre si mismo, dándoles a las ruedas derechas e izquierdas velocidades iguales y distinto sentido. Se trata de medir el ángulo girado ϕ y aplicar la ecuación:

$$x_{CIR} \approx \frac{\int V_d dt - \int V_i dt}{2\phi} \tag{4.3}$$

En la práctica no se han integrado las velocidades, en cambio, se ha tomado la medida de los sensores Hall, y con las dimensiones de las ruedas, se ha calculado la distancia total recorrida por cada rueda. El ángulo girado se ha intentado que fuera aproximadamente 360º, pero de todas formas se ha medido en todos los casos. El experimento se ha realizado en las dos direcciones de giro y a dos velocidades de giro diferentes. Los datos del experimento se ven reflejados en la tabla 4.2 y llevan a determinar el valor de x_{CIR} como

404.33 mm.

Velocidad (PWM)		**Pulsos**		**Distancia lineal (mm)**		$\boldsymbol{\phi}$		$\boldsymbol{x_{CIR}}$
Der	**Izq**	**Der**	**Izq**	$\int \boldsymbol{V_d dt}$	$\int \boldsymbol{V_i dt}$	**º**	**Rad**	**(mm)**
250	-250	2002	-2054	2495,83	-2560,65	356	6,2134	406,9
-250	250	-2066	2118	-2575,61	2640,44	-365	-6,3705	409,4
125	-125	1982	-2028	2470,90	-2528,24	359	6,2657	398,9
-125	125	-2014	2039	-2510,79	2541,95	-360	-6,2832	402,1

Cuadro 4.2: Resultados del experimento realizado para determinar x_{CIR}.

4.4.2. Coeficiente lineal (μ)

De forma similar, para obtener μ se necesita medir la distancia lineal recorrida por el robot, d, cuando se mueve en línea recta, es decir, cuando las ruedas se mueven a la misma velocidad y mismo sentido. La siguiente ecuación proporciona dicho parámetro a partir de las medidas del experimento:

$$\mu \approx \frac{2d}{\int V_d dt - \int V_i dt} \tag{4.4}$$

El experimento se ha realizado tanto hacia adelante como hacia atrás y con velocidades diferentes. Se han obtenido los datos que se muestran en la tabla 4.3 y a partir de ellos se considerará un μ con un valor de 1,0111.

Velocidad (PWM)		**Pulsos**		**Distancia lineal (mm)**		**d**	$\boldsymbol{\mu}$
Der	**Izq**	**Der**	**Izq**	$\int \boldsymbol{V_d dt}$	$\int \boldsymbol{V_i dt}$	**(mm)**	
250	250	3170	3243	3951,93	4042,94	3900	1,0114
-250	-250	-3124	-3193	-3894,59	-3980,61	3770	1,0109
375	375	3572	3642	4453,09	4540,36	4310	1,0097
-375	-375	-3592	-3681	-4478,03	-4588,98	4330	1,0122

Cuadro 4.3: Resultados del experimento realizado para determinar μ.

Capítulo 5

EL TELÉMETRO LÁSER 3D UNO-MOTION

5.1. Introducción

UNO-Motion es un escáner láser 3D de bajo costo que se basa en añadir un grado de libertad adicional a un escáner láser 2D comercial (ver figura 5.1). El diseño persigue los siguientes objetivos: el centro óptico del escáner 3D coincide con el del dispositivo 2D, el mecanismo no interfiere con los planos de medida y se puede obtener un tiempo de barrido eficiente mediante el giro continuo [6].

Figura 5.1: Sensor UNO-Motion.

5.2. Construcción del sensor

El sensor láser 3D está basado en hacer girar un Hokuyo UTM-30LX-EX (ver figura 5.2), por lo que sus características de medidas se heredan de este último. Las dimensiones del sensor 2D son 62 x 62 x 87 mm y pesa 400 g. Tiene un rango de medida de 0.1-30 m y en 25 ms se puede obtener un barrido 2D con un campo de visión de 270º y una resolución angular de 0.25º. Se alimenta a 12 V de corriente continua con una corriente nominal de 0.7 A con picos de hasta 1 A. El dispositivo tiene protección IP67, función multieco, un interface mediante Ethernet y también provee medidas de intensidad.

Figura 5.2: Escáner láser 2D Hokuyo UTM-30LX-EX.

Al alinear el eje extra de rotación con el centro óptico del dispositivo 2D, el escáner 3D mantiene el mismo rango mínimo y evita emplear desplazamiento para el cálculo de las coordenadas. La frecuencia de barrido 3D es el doble que la de giro, ya que con media vuelta se puede obtener un barrido 3D completo. La resolución horizontal depende de la velocidad de giro.

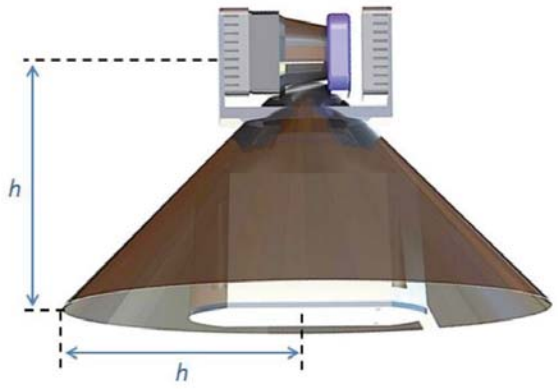

Figura 5.3: Sensor láser con el cono ciego en su base.

La zona ciega del sensor 2D, que es de 90º, se sitúa debajo para evitar interferencias con la base. De esta forma la zona ciega del escáner 3D es un cono cuyo radio en la base es igual a la altura del centro óptico del láser (ver la figura 5.3).

Las partes principales del escáner láser 3D, que se muestran en la figura 5.4, son la cabeza y la base. La cabeza es soportada por la base mediante un cojinete de bolas. La rotación continua se implementa mediante un anillo rozante que permite transmitir energía y señales entre la base y la cabeza giratoria.

Un servomotor de corriente continua proporciona movimiento al mecanismo, que está equipado con una reductora de 3.75:1 de relación de transmisión y un codificador óptico incremental de 4000 pulsos por vuelta. El motor se acciona mediante un puente H completo con señal PWM. La transmisión del movimiento desde el motor hasta el cojinete de bolas se realiza con poleas conectadas mediante correas, que concretamente tienen 20 y 60 dientes, respectivamente, por lo que, una vuelta completa del sensor 2D comprende 45000 pulsos del codificador.

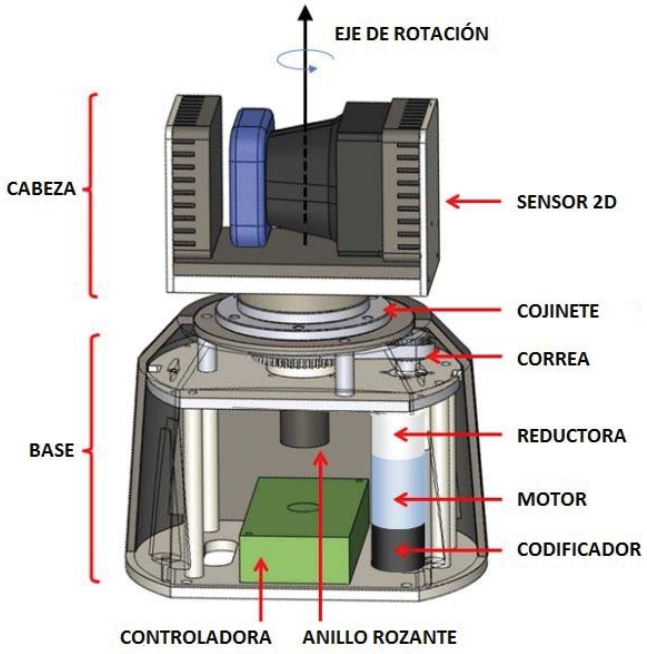

Figura 5.4: Componentes principales del escáner láser 3D.

El controlador del sistema de rotación consiste en un DSP de 16 bits equipado con un generador de señales PWM, decodificador de cuadratura, interface UART y puertos de entrada/salida. Un sensor óptico sirve para inicializar el ángulo cero del telémetro láser cada vez que la base es conectada a la corriente. El propósito del controlador es mantener constante la velocidad de giro y sincronizar la rotación con la adquisición de datos 2D. La computadora se comunica con el controlador mediante un cable convertidor de USB a UART.

Como se puede comprobar, respecto al diseño anterior, basado en hacer cabecear a un *Hokuyo* UTM-30LX alrededor de su centro óptico [7] (ver figura 5.5), se han mejorado los

siguientes aspectos:

- Un nuevo sensor 2D, que incorpora nuevas características tales como la medida de la intensidad.
- El sistema mecánico ha sido simplificado y los cables ahora no son visibles.
- Se puede acceder a los datos del sensor 2D y del controlador de movimiento por separado. Esto confiere una mayor flexibilidad de uso al nuevo modelo.
- El campo de visión ha sido ampliado a 360º, a la vez que el tiempo de barrido se ha disminuido.

Figura 5.5: Telémetro láser 3D UNO-Laser.

La tabla 5.1 muestra la comparación de UNO-Motion con dos modelos comerciales ampliamente utilizados en robótica de la marca Velodyne (ver figura 5.6). Se puede observar que las resoluciones verticales y horizontales son mejores en el UNO-Motion, sin embargo, el rango y la frecuencia de adquisición son menores.

Especificaciones	**HDL-64E**	**HDL-32E**	**UNO-Motion**
Resolución vertical	0.42º	1.33º	0.25º
Campo de visión vertical	26.8º	41.34º	67.5º
Rango máximo	120 m	70 m	30 m
Frecuencia de barrido	5 - 20 Hz	10 Hz	0 - 1.48 Hz
Resolución horizontal	0.09 - 0.35º	0.16º	0 - 6.67º

Cuadro 5.1: Comparación de UNO-Motion con los modelos de Velodyne.

Figura 5.6: Modelos Velodyne HDL-64E y HDL-32E.

5.2.1. Sistemas de comunicación

Se puede acceder al sensor 2D y a la controladora del giro de forma independiente mediante diferentes puertos como se ve en la figura 5.7. El sensor 2D, al que se accede mediante Ethernet, envía continuamente barridos a la computadora y eventualmente puede recibir comandos de configuración desde la misma. La base del sensor 3D se comunica con la computadora por medio de un puerto USB. La controladora de la base captura la señal de sincronización del sensor 2D (que se activa cada vez que se completa un barrido 2D) y envía el ángulo del codificador y su tiempo interno a la computadora. Esta le específica a la controladora de la base del sensor 3D la velocidad de giro.

El puerto de de la controladora USB está configurado con los siguientes parámetros: 500000 baudios, 8 bits, sin paridad, 1 bit de parada y sin control de flujo. Todos los comandos deben terminar con el carácter ASCII 10, 11 ó 13. Los comandos disponibles se muestran en la tabla 5.2.

5.3. Calibración de parámetros

Los parámetros intrínsecos son aquellos relacionados con el mecanismo de adquisición del sensor 3D. Para obtener coordenadas cartesianas exactas se necesita una buena calibración temporal (sincronización) y de los parámetros geométricos [6].

5.3.1. Cálculo de coordenadas cartesianas

Se sitúa el centro O_2 del telémetro láser 2D en el espejo rotatorio, Z_2 en el eje de rotación del mismo y el eje Y_2 estará alineado con la línea central del plano de medida. Un punto dado por el dispositivo 2D tendría unas coordenadas polares definidas por: ángulo α, que se supone nulo en la dirección del eje X_2, y el rango ρ.

Figura 5.7: Diagrama funcional del sensor láser 3D.

Comando	Descripción
EN	Habilitar envió de datos (ángulo y tiempo).
DI	Deshabilitar el envío de datos.
SV±XXXX	Establecer la velocidad de giro, donde +XXXX es un numero con signo comprendido entre -5000 y +5000. Su velocidad de rotación máxima es de 4.65 rad/s, con un comportamiento lineal en todo su rango.
RP	Pedir la posición del codificador. La información recibida tiene el formato +XXXXXX+YYYYYY, donde +YYYYYY es el tiempo interno de la controladora en milisegundos y +XXXXXX es la posición de la cabeza del sensor láser, comprendida entre -45000 y +45000.
RT	Pedir el tiempo actual. La respuesta tiene la forma +TTTTTT.
RET	Poner a cero el tiempo interno de la controladora.

Cuadro 5.2: Comandos para la controladora de la rotación del telémetro láser.

Los ejes de referencia $OXYZ$ del sensor 3D se definen coincidentes con los del sensor 2D cuando la rotación del mismo sobre su eje Y_2 es 0^o, es decir, cuando $\theta = 0^o$. Entonces, las coordenadas cartesianas de la nube de puntos pueden ser calculadas dados ρ, α y θ como:

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \rho \begin{pmatrix} C(\alpha)\,C(\theta) \\ S(\alpha) \\ C(\alpha)\,S(\theta) \end{pmatrix} \tag{5.1}$$

donde $C(\cdot)$ y $S(\cdot)$ representan las funciones coseno y seno respectivamente.

5.3.2. Calibración geométrica

Pequeños errores en la fijación del dispositivo 2D con el mecanismo de rotación pueden provocar que Y_2 no esté perfectamente alineado con Y. Esta desalineación provoca distorsiones en la nube de puntos calculada mediante la ecuación 5.1.

Para telémetros láser 3D de bajo coste construidos haciendo girar un sensor 2D sobre su centro óptico el método de calibración presentado en [8] puede ser utilizado para obtener las desalineaciones del láser UNO-Motion. Debido a las limitaciones en las medidas del dispositivo 2D, con errores del orden de centímetros, los desplazamientos pequeños del orden de milímetros no pueden ser detectados, por lo que la calibración se reduce a obtener los parámetros de rotación sobre los ejes.

Los ángulos óptimos β_0 y α_0, definidos en la figura 5.8, se calculan mediante la maximización iterativa de las áreas planas detectadas a partir de un solo barrido 3D [8].

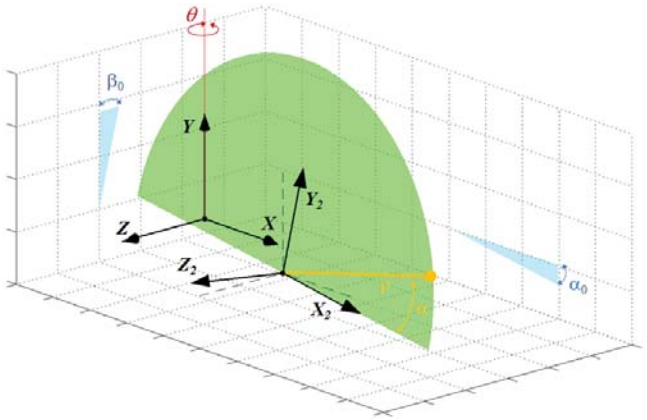

Figura 5.8: Parametros de la calibración.

Los parámetros obtenidos mediante la calibración son: $\beta_0 = 0{,}5^{\circ}$ y $\alpha_0 = -0{,}02^{\circ}$. Después de la calibración la siguiente ecuación puede ser usada para obtener las coordenadas cartesianas de los puntos:

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} C(\alpha_0)\,C(\theta) + S(\beta_0)\,S(\alpha_0)\,S(\theta) & -S(\alpha_0)\,C(\theta) + S(\beta_0)\,C(\alpha_0)\,S(\theta) \\ C(\beta_0)\,S(\alpha_0) & C(\beta_0)\,C(\alpha_0) \\ C(\alpha_0)\,S(\theta) - S(\beta_0)\,S(\alpha_0)\,C(\theta) & -S(\alpha_0)\,S(\theta) - S(\beta_0)\,C(\alpha_0)\,C(\theta) \end{pmatrix} \begin{pmatrix} \rho\,C(\alpha) \\ \rho\,S(\alpha) \end{pmatrix}, \tag{5.2}$$

5.4. Calibración en el robot móvil Andábata

UNO-Motion se ha colocado centrado encima del robot Andábata, en una posición elevada, utilizando un soporte de 20 cm de altura, de forma que la zona ciega del láser es, al menos, igual de grande que el robot. De esta forma se consigue que ninguna medida del láser alcance el cuerpo del robot.

Sea h la altura del centro óptico del sensor 3D, como se vió en el apartado 5.2, y sea θ_0 el ángulo que el sensor láser 2D está rotado, medido desde el eje longitudinal del robot, cuando el mecanismo marca un ángulo de 0^o. Ambos, h y θ_0, son parámetros extrínsecos que depende de la instalación del sensor sobre el robot.

El método utilizado para realizar la calibración es el siguiente:

1. Se coloca el robot Andábata paralelo y pegado a una de las paredes de un pasillo y se toma un barrido completo.
2. Se extraen los planos principales [9], el resultado se puede ver en la figura 5.9.
3. θ_0 puede ser calculado haciendo uso de las normales a las paredes paralelas. El proceso de calibración da como resultado un ángulo $\theta_0 = -150{,}5^o$.
4. El parámetro h es la distancia al plano del suelo e igual al radio del círculo ciego sin medidas que se encuentra en el suelo. Se ha obtenido para h un valor de $72{,}3cm$.

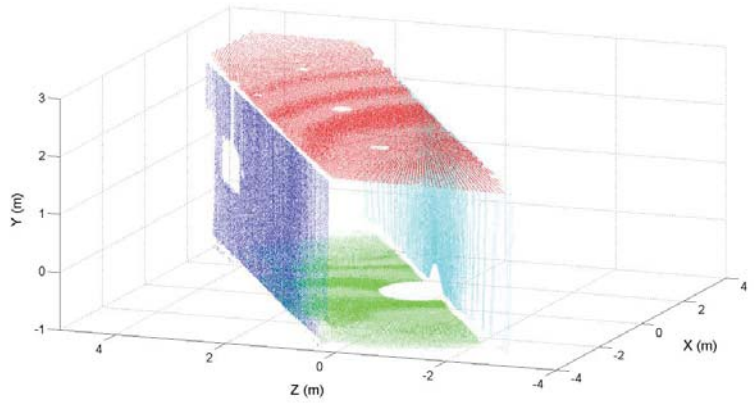

Figura 5.9: Principales planos extraidos de la nube de puntos 3D de un pasillo.

Después de realizar la calibración se puede calcular las coordenadas de un punto capturado por el telémetro láser 3D en referencia al robot Andábata. Sea (x_l, y_l, z_l) un punto del espacio visto desde el sistema de referencias del sensor. Se pueden calcular sus coordenadas respecto el sistema de referencia del robot (x_r, y_r, z_r) mediante la fórmula:

$$\begin{pmatrix} x_r \\ y_r \\ z_r \\ 1 \end{pmatrix} = \begin{pmatrix} C(\theta_0) & 0 & S(\theta_0) & 0 \\ S(\theta_0) & 0 & -C(\theta_0) & 0 \\ 0 & 1 & 0 & h \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_l \\ y_l \\ z_l \\ 1 \end{pmatrix}, \tag{5.3}$$

Capítulo 6

PROGRAMACIÓN DE ANDÁBATA

6.1. Introducción

El objetivo principal del proyecto es la creación de mapas 3D mientras el robot Andábata se encuentra en movimiento. La computadora de a bordo tiene instalada la distribución de Ubuntu 14.04 basada en Linux. También se ha instalado la versión Indigo Igloo de ROS (ver apartado 2.1.1) con una colección de paquetes que incluye todos los paquetes derivados de la aplicación Octomap (ver apartado 3.3).

Como muestra la figura 6.1, es necesario realizar una serie de tareas adicionales para poder crear mapas, las cuales se describen a continuación:

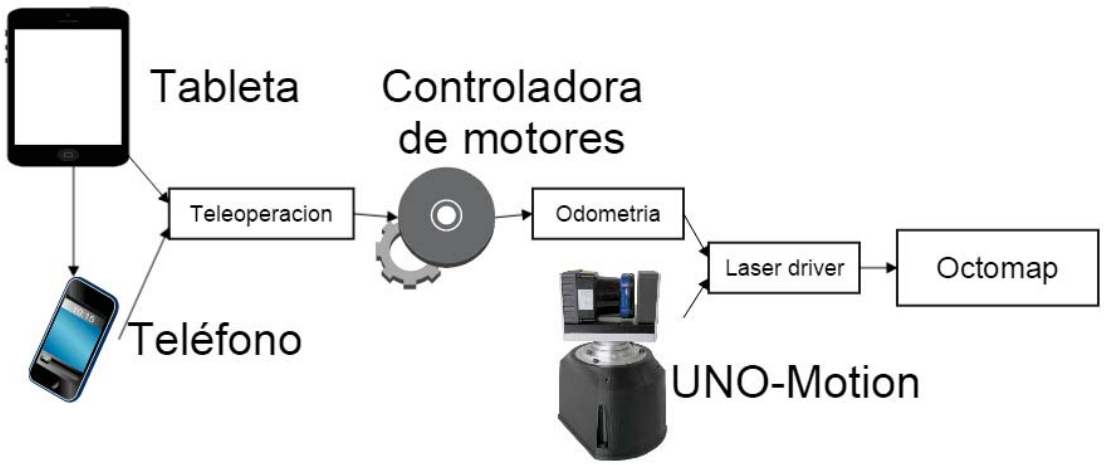

Figura 6.1: Diagrama simplificado de tareas.

1. Mover el robot por las zonas que se quieran explorar. Para ello se ha utilizado un sistema de teleoperación con el cual se puede controlar el robot desde diferentes mandos. Se ha implementado un control mediante deslizaderas en la aplicación de la tableta y otro que usa los datos de su IMU.

2. Conocer la posición y orientación del robot con respecto al mundo, ya que la información de los puntos adquiridos por el telémetro láser vienen expresados en su propio sistema de referencia, y el láser se encuentra instalado en el robot. Se ha creado una aplicación que, mediante el modelo cinamático del robot, estima la posición del mismo mediante odometría.

3. Expresar los puntos obtenidos en el sistema de referencia global. Se necesita conocer la posición del láser 2D en su movimiento de rotación y asociarlo correctamente a los puntos del barrido 2D. La posición en el espacio de estos puntos depende de la posición del sensor láser respecto del robot y de la posición del robot respecto del mundo.

4. Traducir los puntos 3D al formato que entiende Octomap y proporcionarselos, para que los vaya añadiendo al mapa según estén disponibles.

El software desarrollado se ha organizado en tres paquetes. El paquete *"andabata_teleoperation"* contiene los programas pertenecientes a la teleoperación del robot y a la odometría. *"laser_uno"* contiene los *drives* y programas necesarios para controlar el telémetro láser 3D y la creación de mapas. Y *"andabata_msgs"*, que contiene las definiciones de los mensajes personalizados utilizados durante el proyecto.

6.2. Teleoperación

En la figura 6.2 se muestra un esquema de las conexiones del subsistema de teleoperación, que utiliza los siguientes componentes: tableta, móvil, enrutador, ordenador y controladora de los motores.

La transmisión de las órdenes siempre comienza en la tableta y acaba con el envío de órdenes a la controladora de los motores. Consta de varias partes; mandos y *drivers* de los mandos, un selector que escoge el mando que está activo (en este caso se encuentra en uno de los mandos, la tableta), control de la creación de mapas y el control de la velocidad de las ruedas.

La tableta hace de interfaz entre el robot y el usuario, y se comunica tanto con el teléfono móvil como con el ordenador del robot directamente, ambos mediante Wifi utilizando el enrutador como elemento de interconexión.

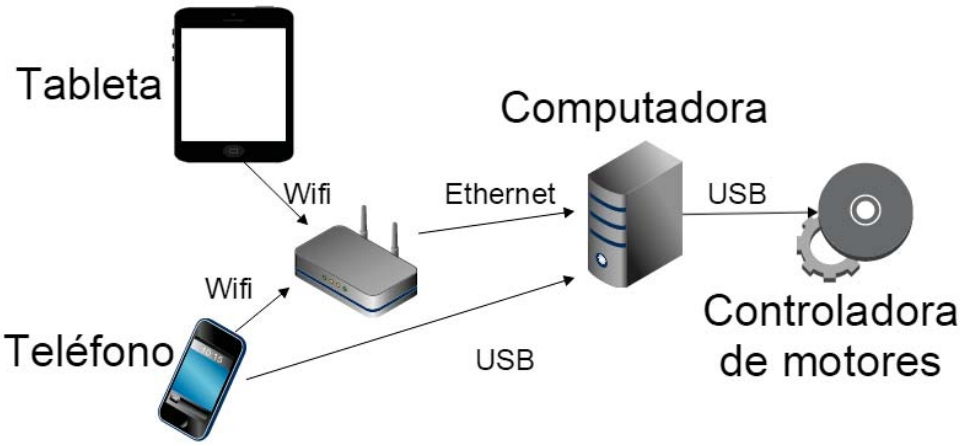

Figura 6.2: Diagrama simplificado del subsistema de teleoperación.

La tableta consta de dos aplicaciones. La primera, *"Andabata"* es una modificación de la aplicación *"Mover-bot"* y permite a la tableta enviar órdenes al móvil y recibir información de los sensores del mismo. Como se ve en la figura 6.3, consta de deslizaderas y botones de control de varias funciones, una brújula esférica, video de las imágenes captadas por la cámara del móvil e indicadores de estado. La segunda aplicación, *"ROS Android Sensors Driver"*, se conecta, a través de la conexión Wifi, con el ROS *master* de la computadora y publica en *topics* la información de los sensores de la tableta.

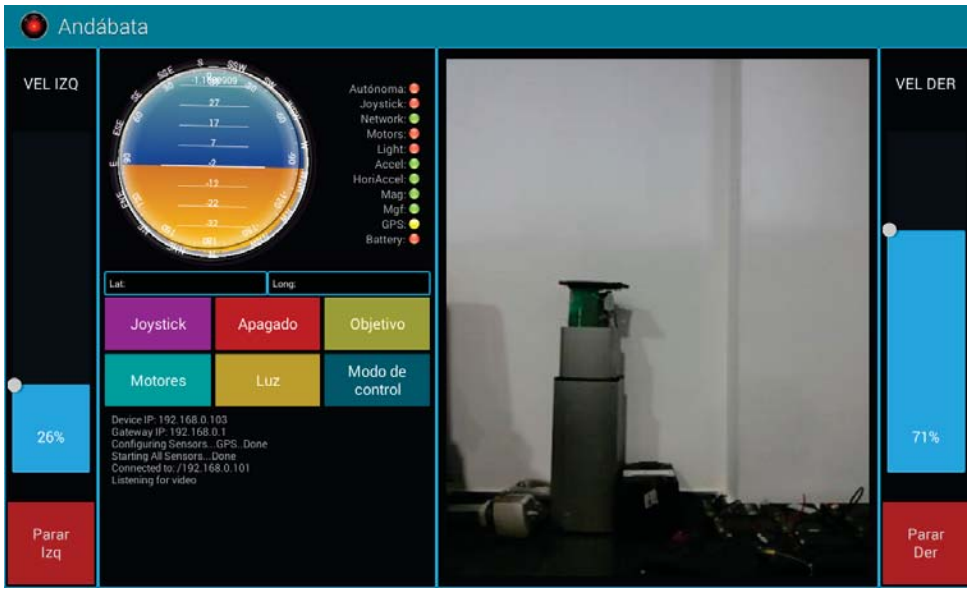

Figura 6.3: Captura de pantalla de la tableta ejecutando la aplicación Andábata.

El teléfono móvil envía las medidas de sus sensores a la tableta y a la computadora, además, hace de intermediario entre la tableta y la computadora del robot. Las imágenes

de su cámara serán útiles para el usuario que dirija el movimiento del robot ya que se pueden visualizar en la tableta, y las medidas de su IMU y el GPS serán utilizadas para la estimación de la posición del robot.

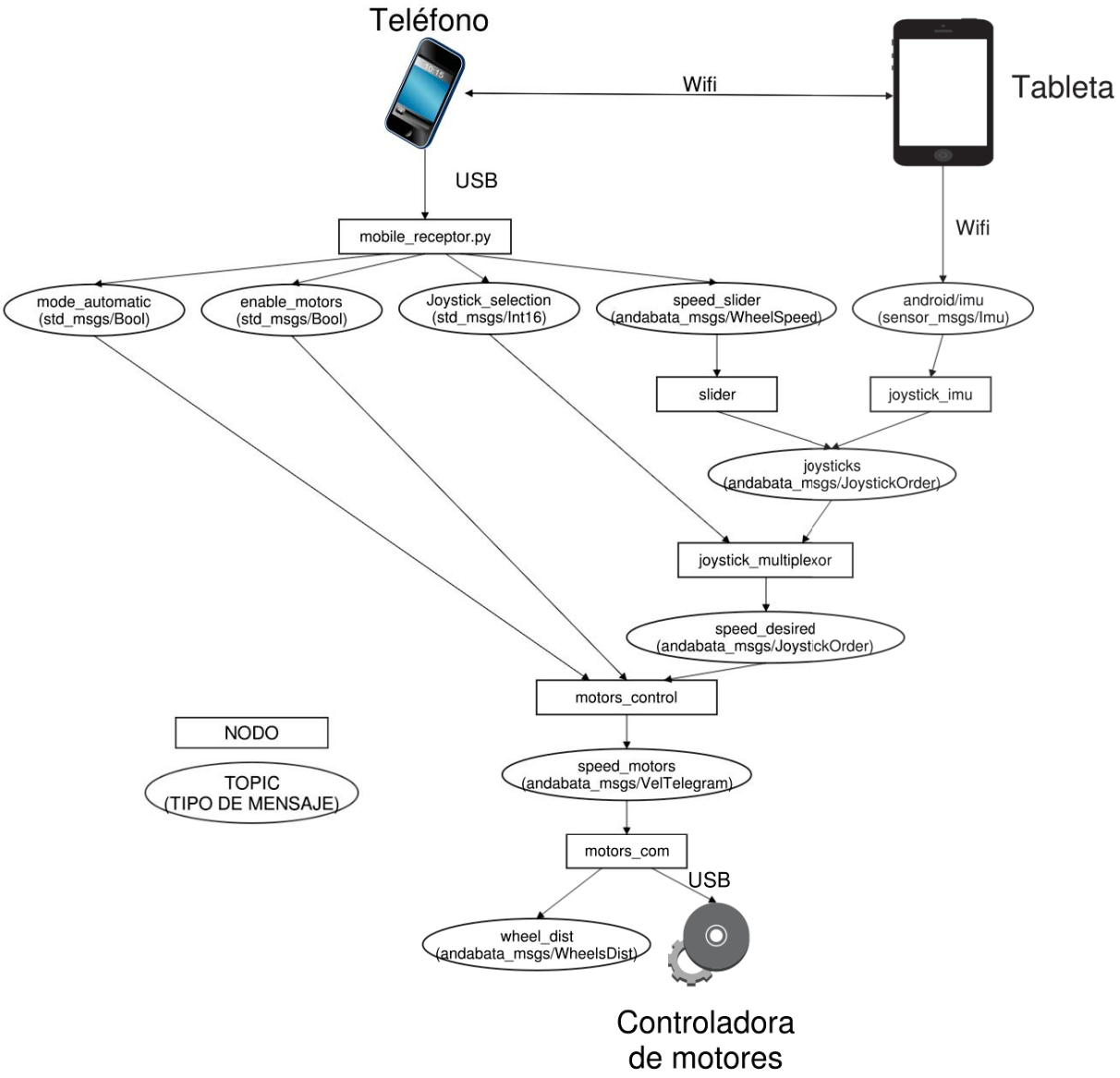

Figura 6.4: Diagrama general de teleoperación.

La estructura del software para la teleoperación se muestra en la figura 6.4, donde se indica el nombre de los nodos y *topics*, así como el tipo de mensaje publicado en cada *topic*. Al esquema hay que añadirle un nodo, *system_control*, que controla cuándo comienza y cuándo se para la creación de mapas. En general lo que se hace es filtrar los datos de los

sensores, seleccionar los datos válidos, procesarlos y enviar instrucciones a la controladora de los motores. Se pueden tener diferentes mandos y seleccionar el que más se adapte a las necesidades del momento. Esto se hace asignando a cada mando un número de identificación, procesando los datos de todos los mandos, y mediante un selector controlar un multiplexor cuya entrada son los datos de todos los mandos y cuya salida son los datos del mando seleccionado. Concretamente las tareas de los nodos son:

- **mobile_receptor** se encarga de leer y reconstruir los telegramas recibidos desde el teléfono móvil y publicarlos en los correspondientes topics. Se ha implementado para que publique la información que se va a utilizar: estado de las deslizaderas, habilitación de los motores, activación del modo automático y de la creación de mapas, y selección del mando.

- **slider** recibe los telegramas que el móvil envía a la computadora cuando se producen cambios en el estado de alguna deslizadera. El telegrama sólo contiene el nuevo estado de dicha deslizadera, y dado que a los motores hay que enviarles las velocidades deseadas de las ruedas de ambos lados, hay que tener un nodo que recuerde el estado de ambas deslizaderas y modifique el correspondiente según los nuevos datos, manteniendo el estado del otro.

- **joystick_imu** recibe los datos del sensor IMU de la tableta, los filtra y los usa para generar y publicar consignas de velocidad para el robot. La implementación de este nodo contempla una zona muerta alrededor del origen de coordenadas (ver figura 6.5) que facilita detener el robot y teleoperarlo en línea recta.

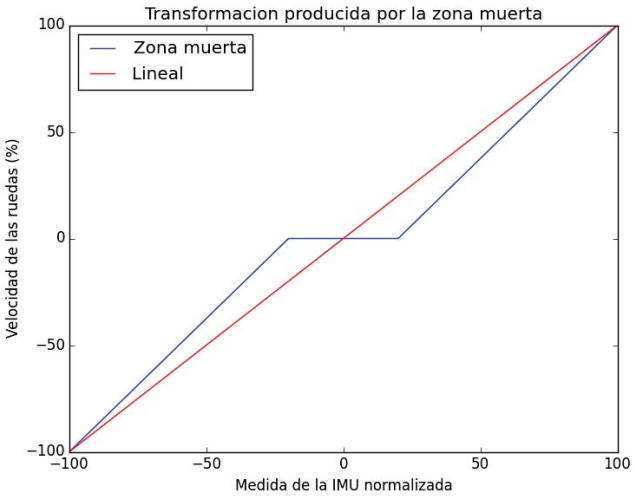

Figura 6.5: Representación del funcionamiento de la zona muerta del sensor.

- **joystick_multiplexor** no es más que la implementación de un multiplexor. A su entrada llegan las órdenes de todos los mandos del sistema y solo deja pasar las

órdenes que provengan del mando cuyo número de identificación coincida con el que se encuentra actualmente seleccionado y, como medida de seguridad, cuando se produce un cambio de mando de control se envía al robot una orden de parada.

- **motors_control** decide las ordenes que se les van a enviar a los motores teniendo en cuenta las órdenes recibidas y el estado de los controles. Para que se envíe la velocidad que el usuario decide los motores tienen que estar activados y el modo automático desactivado. En el caso de que el modo automático esté activado, se enviarán las ordenes procedentes del modulo de navegación (esta función no está actualmente implementada y por seguridad se envía una señal de parada). Si los motores están desactivados se enviará, en todo caso, la orden de parada.

- **motors_com** se comunica directamente con la controladora de los motores. Este recibe la velocidad a la que deben ir los motores, forma el correspondiente telegrama y lo envía. También recibe las lecturas de los codificadores de las ruedas, calcula la distancia recorrida por éstas y las publica para su posterior uso.

- Finalmente **system_control** que lanza y para todo el subsistema de creación de mapas respondiendo al botón *"Apagado"* de la tableta.

6.3. Odometría

Para implementar la odometría en Andábata se ha utilizado el modelo cinemático aproximado desarrollado en la sección 4.4, que permite estimar la velocidad lineal y angular del robot a partir de la velocidad de las ruedas en cada instante.

Como se ve en la figura 6.6, el nodo *motors_com* publica las distancias que las ruedas han recorrido en el *topic wheel_dist* utilizando un tipo de dato personalizado, *WheelsDist*. Este tipo de mensaje consiste en una estructura de tres datos simples, el tiempo en el que fue tomada la medida y las distancias recorridas por las ruedas del lado derecho e izquierdo desde la última medida.

El nodo **odometry** utiliza el modelo cinemático aproximado y los mensajes que recibe a través del *topic wheel_dist* para estimar la posición y orientación. En primer lugar calcula las velocidades del robot respecto al sistema de referencia del mismo (ver figura 4.5), y a continuación las integra de forma aproximada para calcular la posición del robot. Finalmente la publica en formato de mensaje de odometría y de transformación tf.

6.4. Creación de mapas

Para poder crear mapas mientras el robot se mueve, además de lo obtenido en las secciones anteriores, se necesita adquirir los puntos mediante el sensor 2D y conocer el ángulo de giro del mismo.

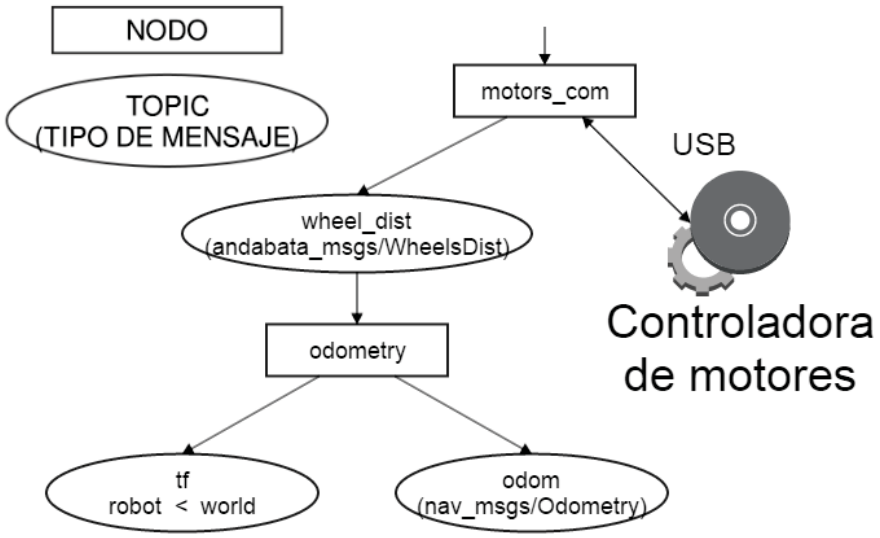

Figura 6.6: Esquema de comunicaciones entre los nodos para la odometría.

Se ha diseñado un subsistema (ver figura 6.7) que, cuando el nodo *system_control* lo lanza, realiza las siguientes tareas:

1. Envío de la velocidad de giro deseada a la controladora de la base del láser.
2. Recepción de la posición del láser y del barrido 2D.
3. Construcción de la nube de puntos 3D.
4. Traducción del tipo de dato.
5. Creación de mapas mediante Octomap.

Las tareas concretas que realiza cada nodo son las siguientes:

- **rotation_speed_input** lee una orden del teclado y la envía al nodo de control del sensor láser. Cuando el robot está en funcionamiento se utiliza escritorio remoto o SSH para introducir las ordenes.

- **laser_com** envía a través de USB las órdenes que se publican en el *topic rotation_speed_laser*. Cuando se lanza inicia la rotación a una velocidad predeterminada, después queda a la espera de nuevas ordenes y, antes de cerrarse, detiene la rotación.

- **laser_position** lee el ángulo en el que se encuentra la cabeza del sensor 3D y la publica en forma de mensaje *LaserEvent* y en forma de transformada tf. En este

punto se tiene el árbol de sistemas de referencia completo (ver figura 6.8), que permite conocer la posición de los puntos adquiridos por el telémetro láser respecto a los ejes coordenados fijos (*world*). Al iniciar el sistema el sistema de referencia *robot* coincide con el sistema de referencia *world*.

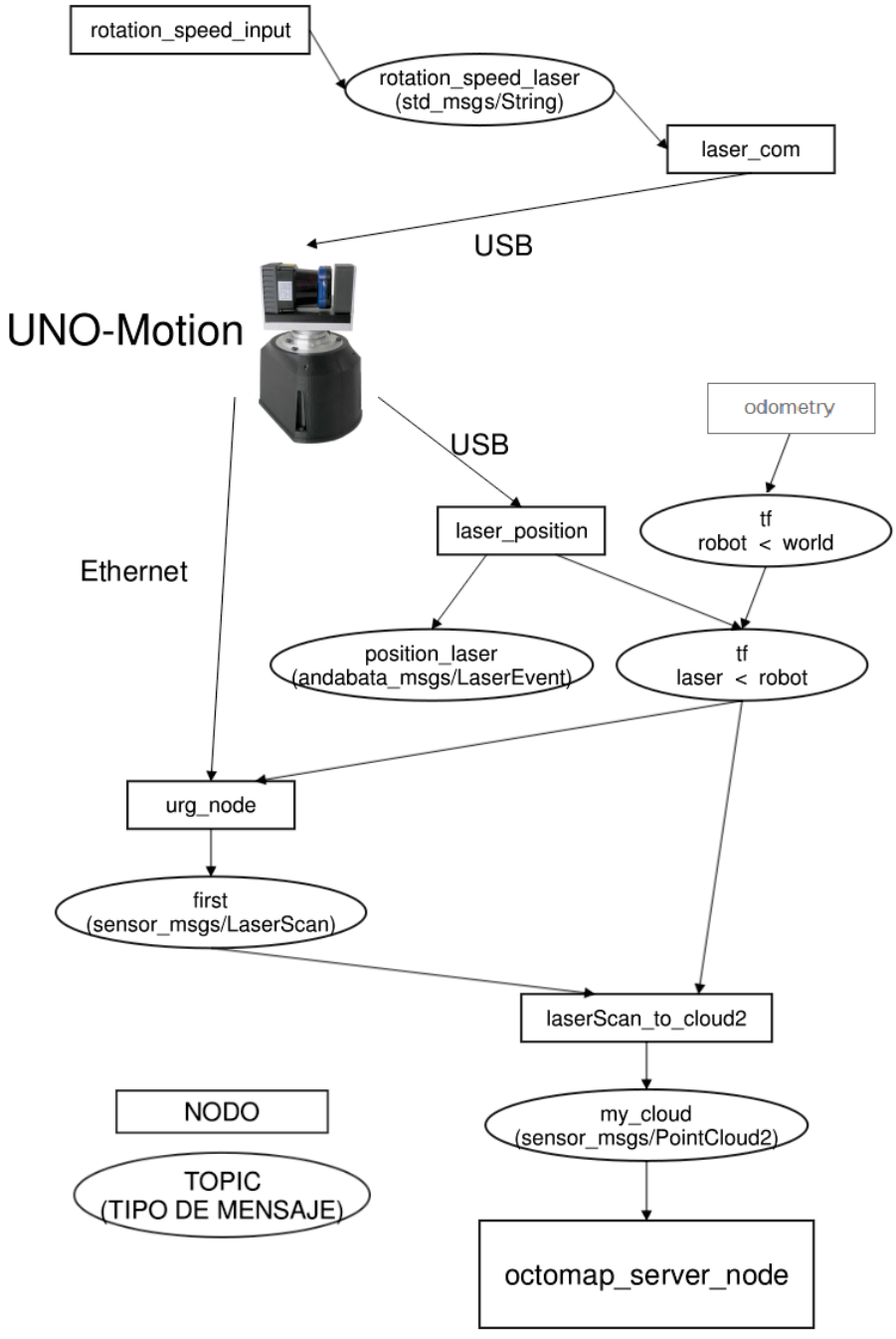

Figura 6.7: Esquema de los ejes coordenados del robot.

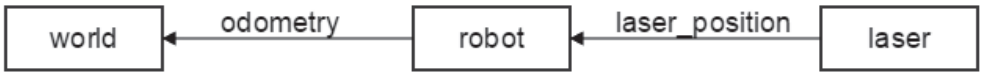

Figura 6.8: Árbol de ejes coordenados del sistema.

- **urg_node** es un nodo que procede de un paquete con el mismo nombre y que contiene los *drivers* para los sensores láser 2D *UTM-30LX* de la marca *Hokuyo*. Se ha usado para la adquisición de escáneres 2D. Este nodo necesita como entrada la posición del láser mediante su transformada tf, esto hace posible que, proporcionándole los ejes del láser en rotación, sitúe los puntos adquiridos en forma de puntos 3D. Los puntos los publica en el formato *sensor_msgs/LaserScan*.

- **laserScan_to_cloud2** transforma la nube de puntos expresada en forma de *sensor_msgs/LaserScan* en datos de la forma *sensor_msgs/PointCloud2*, que son el tipo de datos con el que *octomap_server_node* crea los mapas.

- **octomap_server_node** recibe nubes de puntos y crea mapas basados en octrees. Los mapas permanecen en el sistema mientras ROS y Octomap estén activos.

Después de todo esto, el robot es capaz de crear mapas mientras se desplaza, pero los mapas se borran en el momento en el que Octomap o ROS se paren. Para solventar este problema el paquete *octomap_server* dispone, como se comentó en la sección 3.3, de un nodo llamado **octomap_saver** cuya función es guardar el mapa creado hasta el momento. En el *script*, *start_andabata.sh*, que se ha programado para comenzar todas las tareas del robot se ha incluido la tarea de guardar, cada cierto tiempo, el mapa creado lanzando el nodo *octomap_saver*. Los mapas se guardan con la fecha y la hora en el nombre, creandose uno diferente cada vez.

Capítulo 7

CONCLUSIONES Y TRABAJOS FUTUROS

7.1. Resultados experimentales

A continuación se presenta uno de los experimentos realizados para comprobar la correcta construcción de mapas 3D con Andábata. El entorno experimental se sitúa en las cercanías del túnel mostrado en la figura 7.1. El robot Andábata comienza su movimiento en el interior del tunel y se desplaza hacia el exterior creando un mapa de la zona (ver figura 7.1). Durante el trayecto se realizan tres paradas: una al comienzo (A), en mitad del recorrido (B) y al final del mismo (C).

Figura 7.1: Fotografía del entorno y puntos de paso.

Los mapas creados se han guardado cada 5 segundos, un resumen de los mismos se puede ver en la figura 7.2, junto con el tiempo de construcción desde el comienzo del experimento (minutos:segundos).

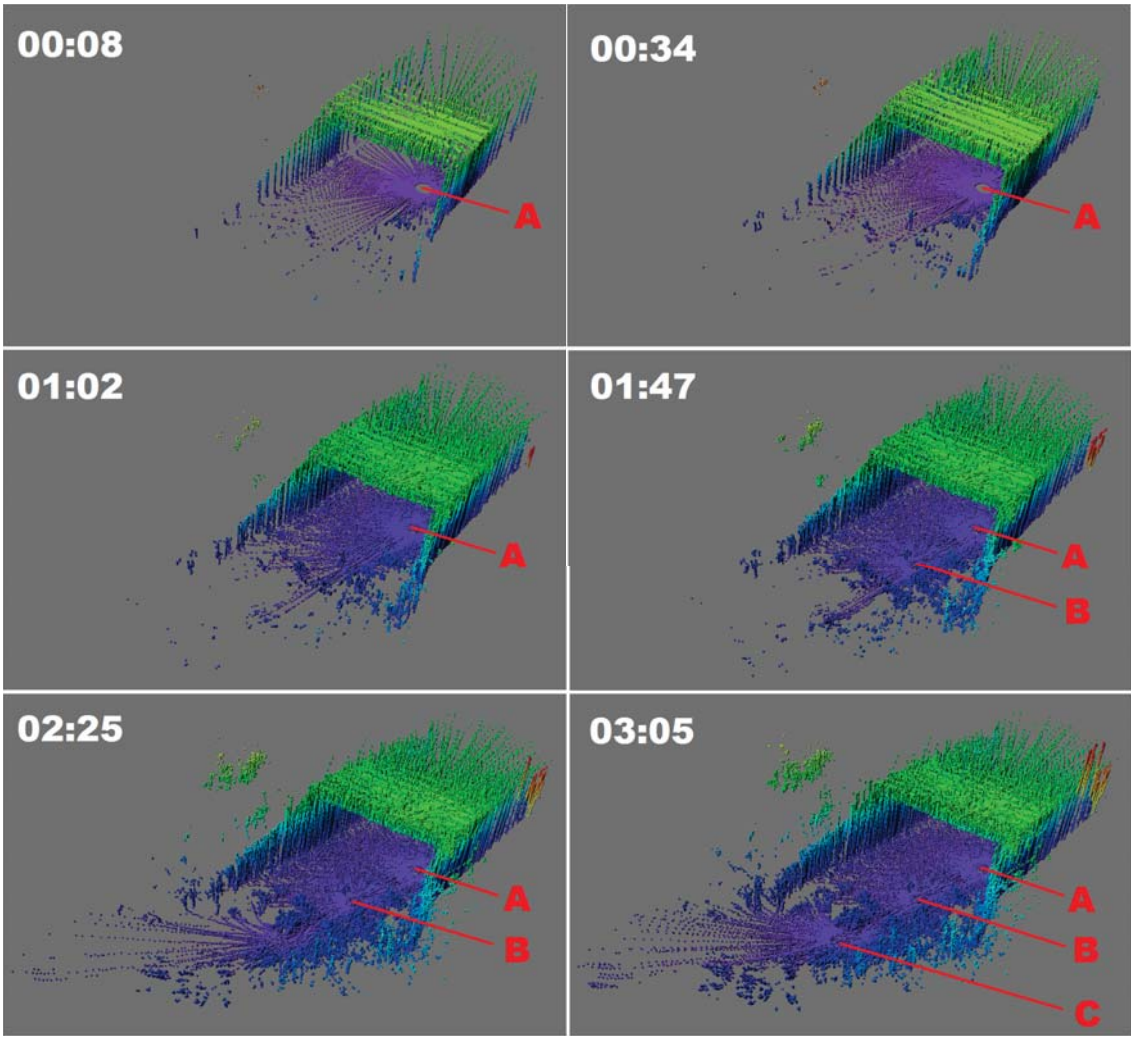

Figura 7.2: Progresión del mapa 3D durante el experimento.

El resultado final ocupa 7.65 MB y contiene un total de 1.604.427 cubos. En la figura 7.3 se muestra, para el mapa final, el espacio libre a) y la estructura final de los *octrees* b).

Los resultados obtenidos en el experimento son satisfactorios, ya que el entorno se distingue con claridad. Sin embargo, se ha comprobado que para que el mapa 3D se cree de forma correcta es necesaria una buena estimación de la posición del robot. Además, es recomendable hacer paradas para tomar datos sin que el robot esté en movimiento.

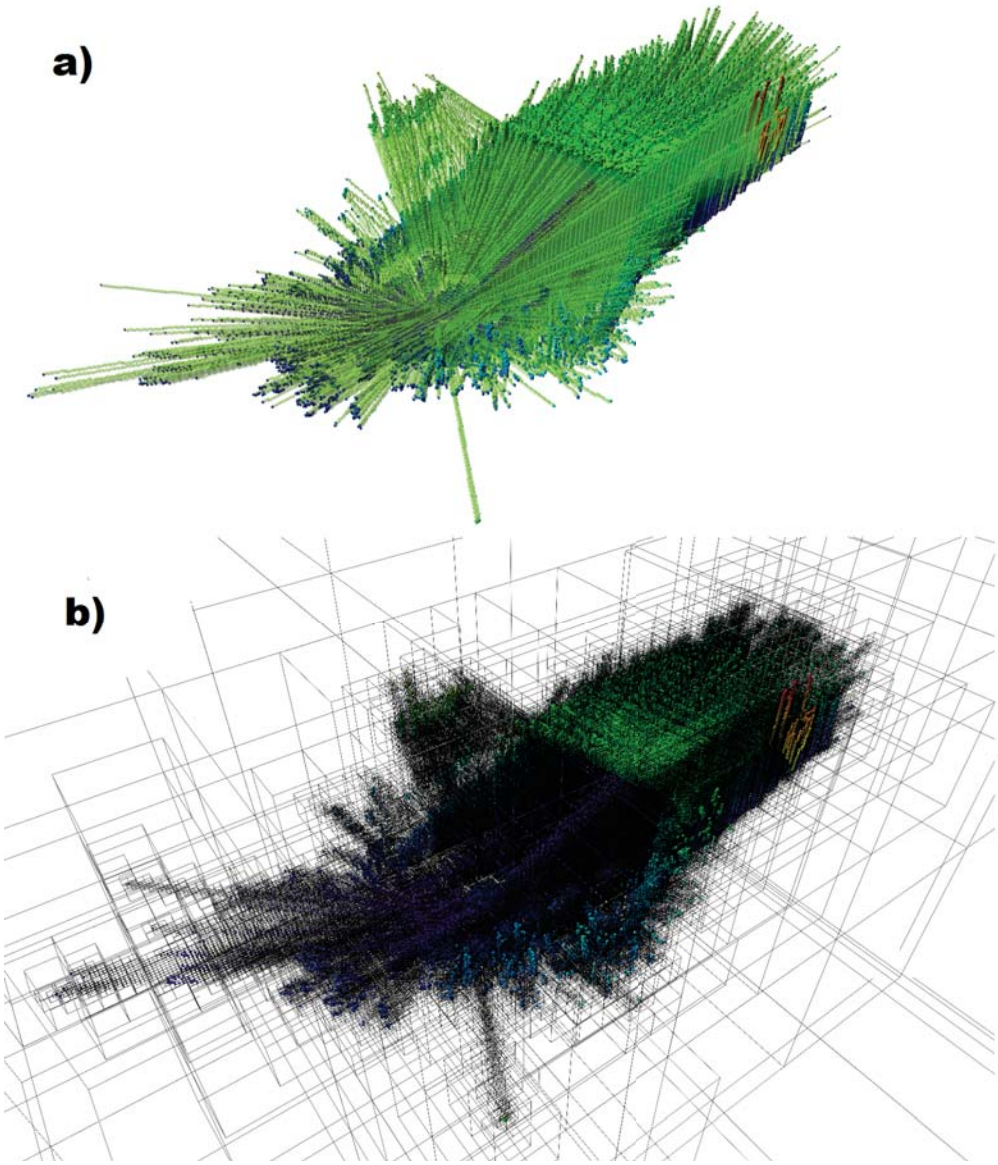

Figura 7.3: El espacio detectado como libre a), la estructura de los *octrees* b).

7.2. Conclusiones

El principal objetivo, que es la creación de mapas en exteriores cuando el robot Andábata se encuentra en movimiento, se ha alcanzado satisfactoriamente. Para ello ha sido necesaria la puesta en marcha del telémetro láser 3D y del robot móvil Andábata. Respecto del robot móvil Andábata:

- Se ha obtenido un modelo cinemático aproximado en 2D que puede ser empleado para obtener la odometría.
- Se ha desarrollado un modulo de teleoperación que se integra en ROS.

En cuanto al telémetro láser:

- Se han incorporado las medidas 3D a ROS.
- Se ha hecho una interfase con la controladora de la base.

Cabe decir que Octomap solo formará bien el mapa 3D cuando la estimación de la posición del robot sea correcta. Al realizar giros bruscos con el robot, los mapas creados presentan desviaciones visibles en las superficies rectas.

Como conclusión personal, este proyecto ha sido un trabajo interesante y estimulante. Realizarlo me ha ayudado a consolidar conceptos nuevos sobre la robótica, sobre todo robótica móvil, y también me ha ayudado a conocer más sobre la comunidad Open Source de robótica y aprender de ella. Me ha permitido aprender a usar el entorno de trabajo ROS, que actualmente es el estándar de código abierto de la comunidad robótica. También he tenido que estudiar el funcionamiento de la librería Octomap, y he afianzado mis conocimientos de programación y de Linux.

7.3. Trabajos futuros

Algunas de las mejoras que se podrían implantar en el robot Andábata en el futuro como continuación del trabajo realizado en este proyecto fin de carrera son:

- Se debe programar un método de estimación de la posición del robot en 3D añadiendo medidas de otros sensores como la IMU o el GPS del teléfono.

- Al iniciar la marcha y al parar el robot cabecea mucho, esto se ve empeorado por el hecho de que el sensor láser se encuentra en una posición elevada. Las medidas del sensor 3D durante esos estados transitorios no son totalmente correctas y se deberían de corregir.

Apéndice A

GUÍA DE INICIO RÁPIDO

Para comenzar a utilizar el robot Andábata, y que todo funcione correctamente, hay que seguir una serie de pasos que se describen es este apéndice.

A.1. Puesta en marcha

Para la puesta en marcha del robot Andábata es necesario:

1. Conectar mediante USB el teléfono móvil encendido al robot Andábata.
2. Encender la computadora del robot. A los pocos segundos en el móvil se habrá lanzado la aplicación Andábata que se verá en la pantalla del mismo. Para que la comunicación no se detenga es necesario que la aplicación no pase a segundo plano (el teléfono móvil se puede bloquear, pero siempre con la aplicación Andábata en primer plano), en el caso de que esto ocurra habrá que reiniciar el sistema.
3. Encender la tableta, iniciar en primer lugar la aplicación *"ROS Android Sensors Drive"* donde se debe introducir la dirección IP del ROS master (192.168.0.102), y a continuación minimizar la aplicación pulsando el botón de inicio.

   Iniciar la aplicación *Andabata*, pulsar el botón *"conect"* e introducir la dirección IP del teléfono móvil, que aparece en este bajo la frase *"listening on 192.168.0.100"*.

A.2. Manejo

Tras finalizar los pasos anteriores el robot estará listo para ser controlado. Al iniciarlo se podrá controlar con las deslizaderas de la aplicación Andábata, uno para las ruedas de la izquierda y el otro para las de la derecha. Para cambiar de mando de control sólo hay que pulsar el botón *"Joystick"* en la tableta.

El manejo mediante la IMU de la tableta es muy intuitivo, como se observa en la figura A.1, donde se definen dos ejes de giro, uno para controlar el avance del robot y otro para el direccionamiento. La posición que se muestra en la figura es la de reposo, 45º respecto a la horizontal en el eje de avance y 0º en el de giro. Para un avance hacia delante hay

que girar la tableta en el sentido negativo del eje *"avance"*, lo contrario para un avance hacia atrás. Para girar a la izquierda se debe de girar la tableta en el sentido positivo del eje *"giro"*, lo contrario para girar a la derecha.

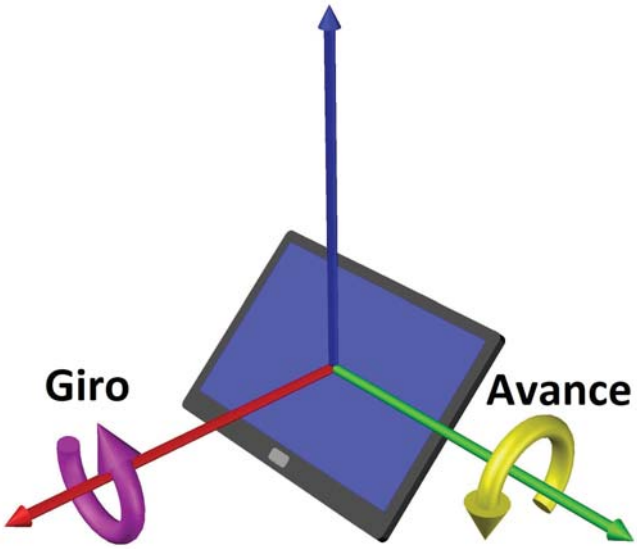

Figura A.1: Posición de reposo y giros para el control del robot Andábata mediante la IMU de la tableta.

Además, se han anulado los movimientos alrededor de los valores de reposo para evitar el movimiento con pequeños giros de la tableta.

Para comenzar y parar la creación de mapas hay que pulsar el botón *"Apagado"* en la tableta. Cuando esta función este activada el telémetro láser 3D comenzará a girar y se empezará a crear y guardar el mapa 3D.

A.3. Registro de mapas 3D y apagado del robot Andábata

Los mapas se guardan, cuando la creación de mapas está activa, con una frecuencia predeterminada. Todos los mapas 3D guardados estarán disponibles en la carpeta */home/am/maps"*.

Para apagar el robot móvil Andábata basta con pulsar el botón de encendido del mismo, así se apaga el sistema de forma segura. El teléfono móvil y la tableta se pueden apagar de forma normal.

Apéndice B

CONFIGURACIÓN DEL SISTEMA

B.1. Dispositivo USB con nombre propio

Cuando se desconecta y se vuelve a conectar un dispositivo USB, podría cambiar de nombre en el sistema, por lo que ponerle nombre a los dispositivos puede ser muy útil. Si se necesitan unos permisos especiales para un dispositivo, habría que dárselos cada vez que se conecta. Sin embargo, al identificarlo con un nombre propio, cada vez que se conecte el dispositivo tendrá el nombre y los permisos que se le otorguen. Para ello, hay que seguir los siguientes pasos:

1. Conectar el dispositivo a la computadora.
2. Encontrar el actual nombre del dispositivo. Ubuntu monta los dispositivos serie-USB en /dev/ttyUSB*, donde * es un numero. Teclear en el terminal:

   ```
   ls /dev/ttyUSB*
   ```

   A partir de ahora se asume que el dispositivo es /dev/ttyUSB0.
3. Encontrar las propiedades udev que son únicas en el dispositivo USB. Teclear en el terminal:

   ```
   udevadm info -a -p $(udevadm info -q path -n ttyUSB0) | egrep
   -i "ATTRS{serial}|ATTRS{idVendor}|ATTRS{idProduct}" -m 3
   ```

   Anotar la información que devuelve, será necesaria para los siguientes pasos.
4. Crear un archivo udev. En el terminal crear un archivo llamado ftdi.rules en /etc/udev/rules.d con 644 como permisos, para ello teclear:

   ```
   sudo touch /etc/udev/rules.d/ftdi.rules && sudo chmod 644
   /etc/udev/rules.d/ftdi.rules
   ```

Abrir el archivo con un editor de textos, por ejemplo gedit:

```
sudo gedit /etc/udev/rules.d/ftdi.rules
```

y copiar la siguiente linea:

```
SUBSYSTEM=="tty", ATTRS{idProduct}=="IDPRODUCT",
ATTRS{idVendor}=="IDVENDOR", ATTRS{serial}=="SERIAL",
SYMLINK+="DEVICENAME", MODE="0666"
```

Substituir IDPRODUCT, IDVENDOR, y SERIAL por los valores obtenidos en el paso 3. DEVICENAME será el nombre que tendrá el dispositivo y MODE los permisos que tendrá.
Ejemplo:

```
SUBSYSTEM=="tty", ATTRS{idProduct}=="6001",
ATTRS{idVendor}=="0403", ATTRS{serial}=="FTCWYO5H",
SYMLINK+="ttyTilt", MODE="0666"
```

Guardar y cerrar el documento.

5. Desconectar y volver a conectar el dispositivo para empezar a usar la nueva regla creada. Si se desea crear más de una regla solo se tiene que añadir al mismo archivo, no es necesario crear otro nuevo.

B.2. Programas de ejecución al arrancar

Lo deseable en un robot es que al arrancar se inicien una serie de tareas, programas y aplicaciones. Para ello hay que iniciar la herramienta "startup applications" (ver figura B.1).

Para añadir aplicaciones que se inicien al arrancar la computadora se hace clic en el botón de añadir. Aparecerá la ventana de la figura B.2 en la que habrá que rellenar los campos. En el campo de nombre y el de comentario se escriben el nombre y la descripción. Para el campo de comando existen más posibilidades:

- Escribir un comando que se ejecutará al arrancar la computadora.
- Seleccionar una aplicación que se lanzará al iniciar la computadora.
- Seleccionar un *script* que se ejecutará cuando se arranque la computadora.

En este proyecto se utilizá la tercera opción y se ejecutará un *script* que iniciará todos los procesos necesarios y aplicaciones desarrolladas para el robot.

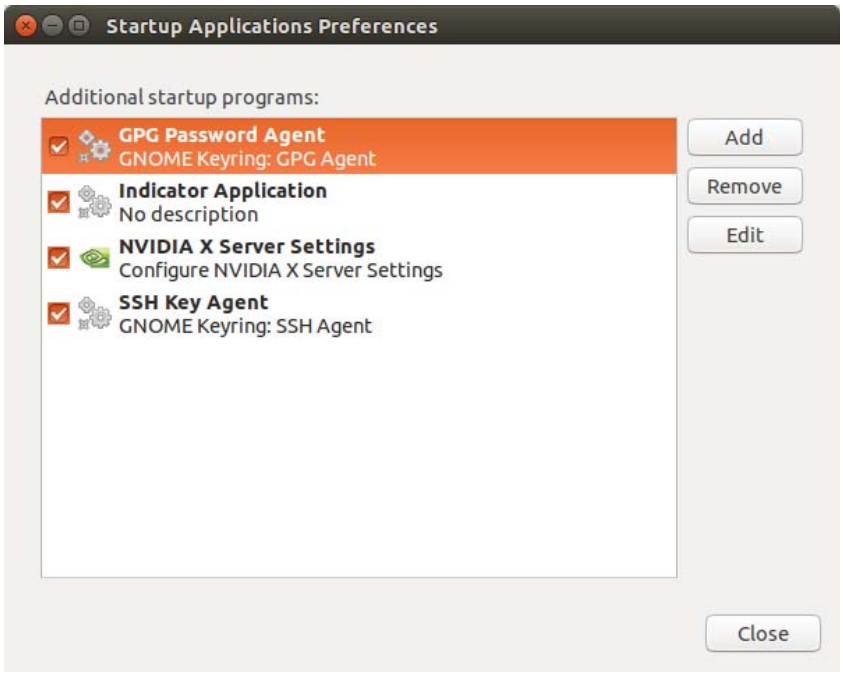

Figura B.1: Ventana de *startup application* en Ubuntu.

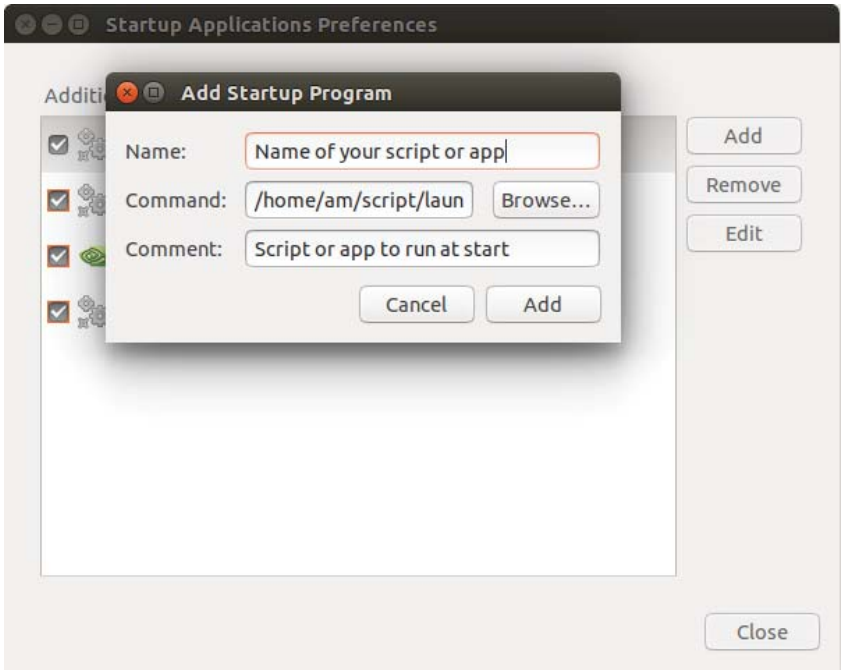

Figura B.2: Ventana para añadir una nueva aplicación.

B.3. Iniciar el sistema sin pantalla

Por defecto Ubuntu no arranca con todas sus funciones si no detecta una pantalla conectada. Se necesita que Ubuntu esté configurado para no necesitar pantalla, ya que, cuando el robot esté operativo no dispondrá de ella. Para ello se utiliza la herramienta xdiagnose, que sirve para gestionar errores y problemas con el sistema gráfico de Ubuntu. Para acceder a su menú teclear:

```
sudo xdiagnose
```

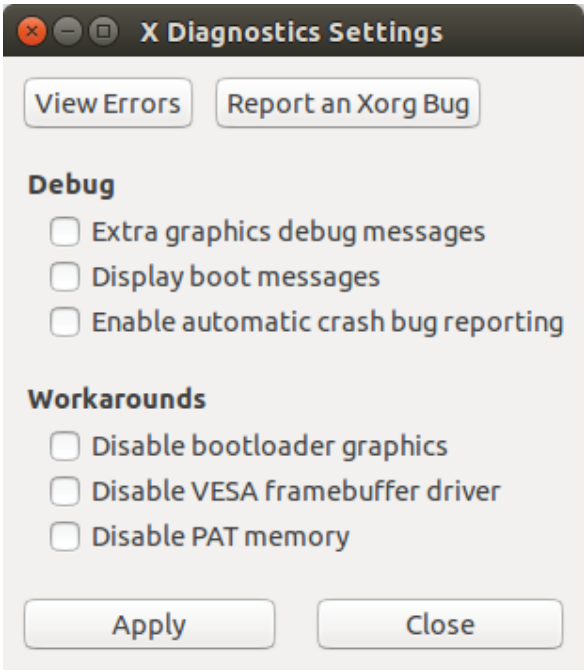

Figura B.3: Ventana de la aplicación *xdiagnose*.

Después de introducir la contraseña aparecerá una nueva ventana. Para que el sistema operativo no muestre problemas al no tener pantalla al arrancar hay que deshabilitar todas las casillas del apartado "Debug", quedará como en la imagen B.3, aceptar y cerrar la ventana.

B.4. Terminal remoto SSH

SSH no sólo sirve para usar comandos en máquinas remotas, sino para transferencias de ficheros de forma segura ya sea por SCP o FTP y servicios de terminal remoto. Se usará OpenSSH. Para instalarlo teclear:

```
sudo apt-get install openssh-server
```

Se puede acceder y modificar su configuracion:

```
sudo gedit /etc/ssh/sshd_config
```

Para arrancar, parar y reiniciar el servidor teclear respectivamente:

```
sudo /etc/init.d/ssh start
sudo /etc/init.d/ssh stop
sudo /etc/init.d/ssh restart
```

Para acceder desde otra computadora se puede usar alguno de los programas que provean de este servicio. Ubuntu tiene preinstalado este servicio, basta con teclear:

```
ssh host-name
```

donde host-name es la IP y el puerto del servidor. Para salir basta con teclear *"exit"*.

También se puede utilizar, por ejemplo, PuTTY, que ya se instaló en la computadora para algunas pruebas y es un poco más interectivo:

```
sudo apt-get install putty
```

Dado que se va a usar para introducir comandos de forma remota sólo hay que introducir la IP y el puerto que se esté usando en el servidor.

B.5. Escritorio remoto

El servicio de escritorio remoto permite ver y controlar el escritorio de una computadora desde otra distinta. Esto se puede hacer si se tienen los dos computadoras en una red local y también por internet, pero eso sí, la computadora que se quiere ver y controlar debe tener una IP estática. Para este último caso se debe de acceder a la configuración del enrutador desde el navegador web y modificar el tipo de IP a estática, también se debe abrir el puerto 5900, que es el que se usará, ya que los enrutadores también suelen tener un cortafuegos.

Se denominará "servidor" a la computadora que se quiere controlar y "cliente" desde la que se quiere acceder.

B.5.1. Servidor

Lo primero será configurar el servidor, para darle los correspondientes permisos, para que se pueda acceder a él. Accedemos a la herramienta "Desktop Sharing Preferences" y se abrirá la ventana de la imagen B.4.

Para el presente proyecto final de carrera se deben activar todas las casillas del apartado de compartir (las dos primeras casillas). Así se permite que los usuarios que se conecten al escritorio remoto puedan ver y controlar la computadora.

En el apartado de seguridad la ultima casilla debe de estar activada, con esta casilla activa el equipo se configura corectamente de forma automatica y se abre el puerto (5900). Y, además, se deja desactivada la casilla "confirmar cada acceso a esta maquina", ya que

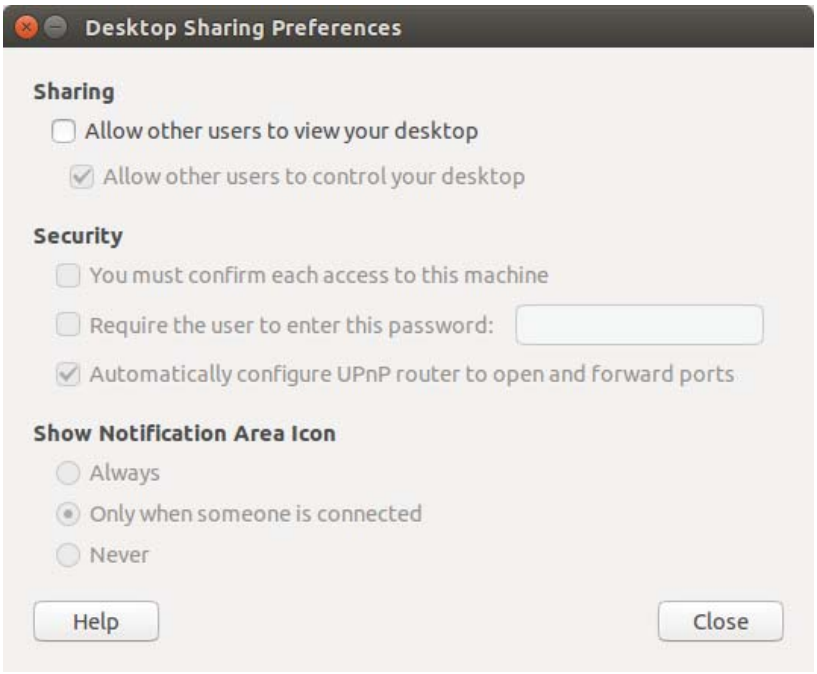

Figura B.4: Ventana de configuración para permitir el acceso al escritorio remoto.

si esta casilla esta activada siempre se tiene que aceptar usuarios entrantes y esto no se puede hacer cuando el robot esté sin perifericos.

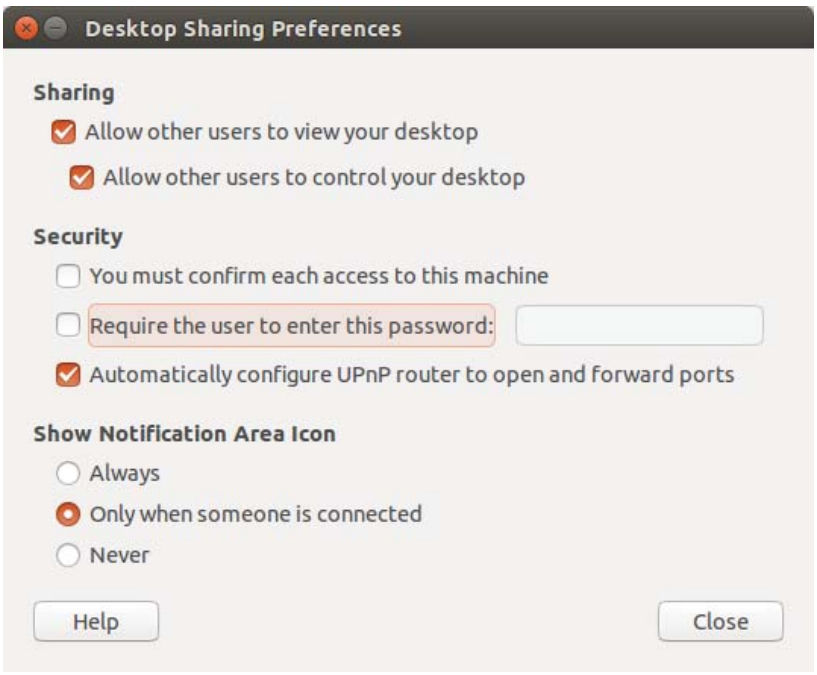

Figura B.5: Configuración para permitir el acceso al escritorio remoto sin contraseña.

El resto de casillas es opcional. Si se quiere que no todo el mundo pueda acceder se pondrá una contraseña que será necesario conocer para acceder. En este caso, como el robot carecerá de acceso a internet, solo los equipos de la red local podrán acceder al escritorio remoto, por lo que se prefiere dejar la configuración sin contraseña.

En el último apartado se decide cuándo se quiere que aparezca el icono en la barra de notificación. Si se escoge que solo aparezca el icono cuando se conecte una computadora, la configuración quedaría como en la figura B.5.

B.5.2. Cliente

Para acceder al servidor desde el cliente se debe iniciar la aplicación "Remmina Remote Desktop Client". Se abrirá la ventana del visor, como la de la figura B.6, en la que hay que pulsar el botón de añadir conexión.

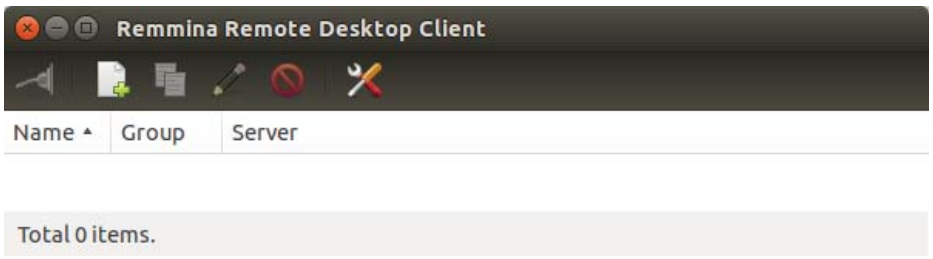

Figura B.6: Ventana principal de la aplicación del cliente.

Se abrirá una nueva ventana. En esta ventana se da nombre a la conexión (Andabata, por ejemplo), y como protocolo se selecciona VNC. En el apartado servidor debemos escribir la dirección IP del servidor y en calidad seleccionar calidad pobre, esto hará que la comunicación sea mas fluida en detrimento de la calidad de imagen. La configuración quedará como se muestra en la figura B.7. En el caso de que el servidor necesitara de algún tipo de identificación o contraseña, también habría que rellenarlo aquí.

Le damos a guardar y cerramos. Ahora en la ventana principal del programa aparecerá la conexión que se ha configurado, como se puede ver en la figura B.8, y para conectarse bastará con hacer doble clic en ella o seleccionarla y hacer clic en el botón conectar. Al conectarnos se abrirá una ventana con el escritorio del servidor, el cual se puede controlar como si fuera la computadora cliente.

Existen limitaciones gráficas con el escritorio remoto, ya que en este modo no se ejecutarán tareas con alto trabajo de gráficos tales como simulaciones. Esto hace que, por ejemplo, no se pueda usar el escritorio remoto para ver en tiempo real el mapa que el robot esta construyendo.

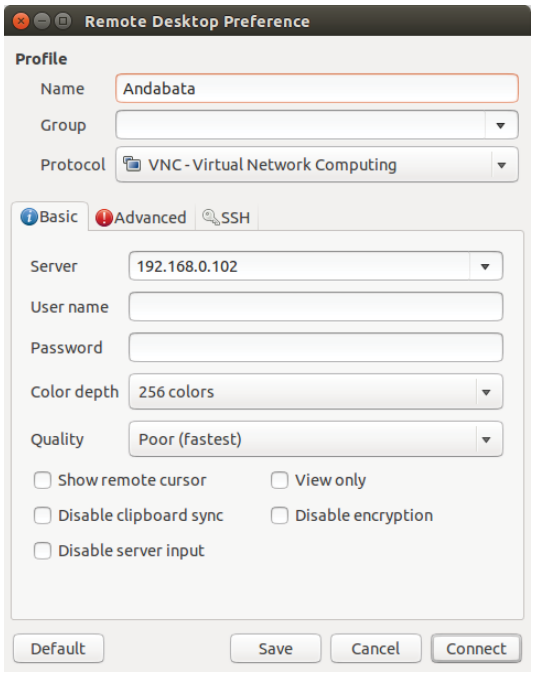

Figura B.7: Configuración para acceder al servidor de escritorio remoto.

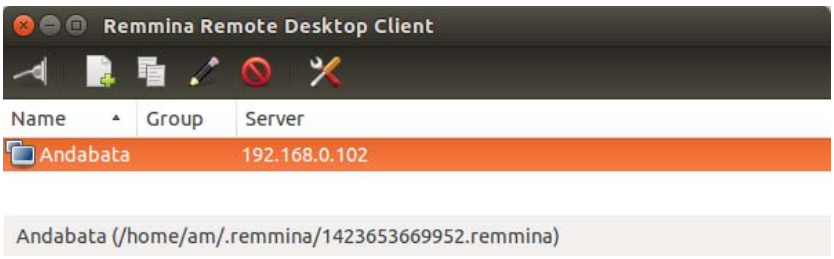

Figura B.8: Nueva conexión añadida.

B.6. Cambiar el comportamiento del botón de encendido

Ya que en el uso del robot no se contará con pantalla y ratón, no hay forma de apagar el robot mas que con el botón de encendido.

En computadoras como la que hay instalada en el robot Andábata, por defecto el comportamiento de este botón es, cuando la computadora está apagada, iniciarla, y cuando esta está encendida, mostrar un menú interactivo en el que se puede elegir lo que se quiere hacer (ver figura B.9). Esto resulta inútil, ya que dicho menu ni se puede ver, ni se puede hacer clic en ninguna de las opciones.

Lo que se quiere hacer es escoger directamente lo que haga la computadora al pulsar el botón de encendido, que será apagarse. Para ello primeramente se comprueba cual es la configuración que hay en este momento introduciendo en el terminal:

Figura B.9: Menu interactivo que aparece al pulsar el botón de encendido.

```
gsettings list-recursively org.gnome.settings-daemon.plugins.power
```

Aquí se puede ver el comportamiento de la computadora al pulsar el botón de encendido:

```
org.gnome.settings-daemon.plugins.power button-power 'interactive'
```

Lo que hay que hacer es cambiar *ínteractive'* por *'shutdown'*. Escribir en el terminal:

```
gsettings set org.gnome.settings-daemon.plugins.power button-power 'shutdown'
```

A partir de ahora al pulsar el botón de encendido la computadora se apagará. De la misma forma se puede escoger diferentes comportamientos como *'hibernate'* o *'suspend'*.

Apéndice C

CÓDIGO DE LOS PROGRAMAS

C.1. Nodos del paquete andabata_teleoperation

slider

codigo/andabata_teleoperation/slider.cpp

```
// Convierte la informacion que llega de los sliders de la
   tableta en mensajes entendibles por el sistema

#include "ros/ros.h"
#include "std_msgs/Int16.h"
#include "andabata_msgs/WheelSpeed.h"
#include "andabata_msgs/JoystickOrder.h"

#define joystick_n 1

// Necesitamos almacenar las velocidades de las ruedas, ya que
   solo recibimos los cambios y estos se produciran en una sola
   de las ruedas. El mensaje debe de tener la velocidad deseada
   para ambas ruedas
int vel_l=0, vel_r=0;

void change_speed(const andabata_msgs::WheelSpeed msg)
{
   static ros::NodeHandle n;
   static ros::Publisher pub_vel = n.advertise<andabata_msgs::
      JoystickOrder>("joysticks", 1);

   if(!msg.wheel){       // Rueda izquierda
      vel_l = msg.speed;          // Actualiza la velocidad de la
         rueda
   }else{          // Rueda derecha
      vel_r = msg.speed;
```

```
    }

    andabata_msgs::JoystickOrder message;
    message.id=joystick_n;
    message.wheel_l=vel_l;
    message.wheel_r=vel_r;

    pub_vel.publish(message);
}


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

    ros::Subscriber sub_vel = n.subscribe("speed_slider", 10,
        change_speed);

    ros::spin();

    return 0;
}
```

joystick_imu

codigo/andabata_teleoperation/joystick_imu.cpp

```
// decide, segun la posicion de la tableta, las velocidades de
   las ruedas, utilizando los datos de la IMU.

#include "ros/ros.h"
#include "sensor_msgs/Imu.h"
#include "andabata_msgs/JoystickOrder.h"
#include <math.h>
#include <ros/callback_queue.h>


#define joystick_n 2     // id del joystick en el sistema
#define dead_zone 10     // Zona muesta del sensor, tanto en giro
    como en desplazamiento lineal

// Los comandos de velocidad se envian en porcentaje, por lo
   tanto no se puede superar el 100
int limit(int num){
   if (num > 100){
      return 100;
   }else if (num < -100){
      return -100;
   }else
      return num;
}

// Esta funcion adapta el valor recibido al nuevo rango creado
   por la zona muerta
int adapt_range(int num){
   if (abs(num) < dead_zone){
      return 0;
   }else{
      return (int)round((num-dead_zone)*100.0/(100.0-dead_zone))
         ;
   }
}


void vel(const sensor_msgs::Imu msg){

   static ros::NodeHandle n;
   static ros::Publisher pub_vel = n.advertise<andabata_msgs::
      JoystickOrder>("joysticks", 10);
```

```
// Se guardan 10 valores para hacer un filtro mediante la
   media
static int pos[3][10]={{0,0,0,0,0,0,0,0,0,0},
   {70,70,70,70,70,70,70,70,70,70},
   {70,70,70,70,70,70,70,70,70,70}};
static int i=0;
int sumx=0;
int sumy=0;
int sumz=0;

int vel_r, vel_l, diff;

// Adaptamos las medidas al rango de 0-100 y eliminamos los
   decimales
int x = (int)round(msg.linear_acceleration.x * 10);
int y = (int)round(msg.linear_acceleration.y * 10);
int z = (int)round(msg.linear_acceleration.z * 10);
x = limit(x);
y = limit(y);
z = limit(z);
printf("-------------------------------------------------------\n");

// Actualizamos el estado de los filtros
pos[0][i] = x;
pos[1][i] = y;
pos[2][i] = z;

for(int j=0; j<10; j++){
   sumx+=pos[0][j];
   sumy+=pos[1][j];
   sumz+=pos[2][j];
}

printf("x: %d      \ty: %d      \tz: %d\n", x, y, z);
printf("sumx: %d    \tsumy: %d    \tsumz: %d\n", sumx, sumy,
   sumz);

i++;
if(i>10)
   i = 0;

// El avance y el giro estan desacoplados. El avance lineal
   depende de los ejes z e y y el giro del eje x
int lineal = (sumz - sumy)/10;
int angular = abs(sumx)/10;
```

```
// creamos la zona muerta
lineal = adapt_range(lineal);
angular = adapt_range(angular);

// Calculamos las velocidades de cada rueda para que el robot
    mantenga la velocidad lineal y gire la velocidad angular.
    El giro se hace de forma diferencial, sumando a unas
   ruedas y restando a las otras
if ((abs(lineal) + angular) > 100){
   diff = abs(lineal) + angular -100;
}else{
   diff = 0;
}
printf("lineal: %d \tangular: %d \tdiff. %d\n", lineal,
   angular, diff);

if (lineal > 0){  // adelante
   printf("adelante - ");
   if (sumx > 0){ // izquierda
      printf("izquierda\n");
      vel_l = lineal - angular - diff;
      vel_r = limit(lineal + angular);
   }else{            // derecha
      printf("derecha\n");
      vel_l = limit(lineal + angular);
      vel_r = lineal - angular - diff;
   }
}else{            // atras
   printf("atras - ");
   if (sumx > 0){ // izquierda
      printf("izquierda\n");
      vel_l = lineal + angular + diff;
      vel_r = limit(lineal - angular);
   }else{            // derecha
      printf("derecha\n");
      vel_l = limit(lineal - angular);
      vel_r = lineal + angular + diff;
   }
}

printf("vel_i: %d\tvel_d: %d\n", vel_l, vel_r);
andabata_msgs::JoystickOrder message;
message.id=joystick_n;
message.wheel_l=vel_l;
message.wheel_r=vel_r;
```

```
    pub_vel.publish(message);
}



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

    ros::Subscriber sub_sel = n.subscribe("/android/imu",1,vel);

    ros::Rate r(30);

    while(ros::ok()){
        ros::getGlobalCallbackQueue()->callAvailable(ros::
            WallDuration(0.01));
        r.sleep();
    }

    return 0;
}
```

joystick_multiplexor

codigo/andabata_teleoperation/joystick_multiplexor.cpp

```
// Actua como un multiplexor, tiene muchas entradas y una sola
   salida. Dependiendo de la segnal de control, a la salida,
   habra una u otra de las entradas

#include "ros/ros.h"
#include "std_msgs/Int16.h"
#include "andabata_msgs/JoystickOrder.h"
#include <ros/callback_queue.h>

// Empezamos por el joystick con id 1
int joystick=1;

// Cuando se recibe una nueva id se realiza el cambio y se para
   el robot
void change_joystick(const std_msgs::Int16 msg){
   static ros::NodeHandle n;
   static ros::Publisher pub_vel = n.advertise<andabata_msgs::
      JoystickOrder>("speed_desired", 5);

   joystick = msg.data;

   // Aqui se puede implementar algun protocolo de actuacion o
      de aviso de que el joystick ha cambiado. Actualmente lo
      que hace es parar el robot
   andabata_msgs::JoystickOrder message;
   message.id=msg.data;
   message.wheel_l=0;
   message.wheel_r=0;
   pub_vel.publish(message);
}


// Solo deja pasar mensajes de uno de los joysticks
void change_speed(const andabata_msgs::JoystickOrder msg)
{
   static ros::NodeHandle n;
   static ros::Publisher pub_vel = n.advertise<andabata_msgs::
      JoystickOrder>("speed_desired", 5);


   if (msg.id==joystick)
      pub_vel.publish(msg);
}
```

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

    ros::Subscriber sub_vel = n.subscribe("joysticks", 10,
        change_speed);
    ros::Subscriber sub_sel = n.subscribe("joysticks_selection",
        1, change_joystick);

    ros::spin();

    return 0;
}
```

motors_control

codigo/andabata_teleoperation/motors_control.cpp

```
// Este programa decide que velocidades se les envian a los
   motores en funcion de varios parametros como que los motores
   esten activados, el modo automatico este activado o la
   velocidad decidida por el usuario

#include "ros/ros.h"
#include "std_msgs/Bool.h"
#include "andabata_msgs/JoystickOrder.h"
#include "andabata_msgs/VelTelegram.h"
#include <ros/callback_queue.h>

using std::string;
using std::exception;

bool mode_auto=0, enabled_motors=1;

// Esta funcion es la encargada de comunicarse con el nodo de
   comunicacion con los motores
void send_speed(int vel_l, int vel_r){
   static ros::NodeHandle n;
   static ros::Publisher pub_vel = n.advertise<andabata_msgs::
      VelTelegram>("speed_motors", 1);

   andabata_msgs::VelTelegram msg_motors;

   msg_motors.vel_l = vel_l;
   msg_motors.vel_r = vel_r;
   pub_vel.publish(msg_motors);
   return;
}


// Decide el comando a enviar a los motores
void change_speed(const andabata_msgs::JoystickOrder msg)
{
      try {
         if(enabled_motors){  // Motores habilitados
            // mode_auto=0 teleoperacion; mode_auto=1 autonomo
            if (mode_auto){
               //                 //
               //                 //
               // rellenar con el programa del modo automatico
               //                 //
```

```
                //                    //
                send_speed(0,0); // Por ahora por seguridad

            }else{
                // Modo teleoperacion
                send_speed(msg.wheel_l, msg.wheel_r);
            }
        }else{                          //motores deshabilitados
            send_speed(0,0);
        }
    } catch (exception &e) {
        printf("Unhandled Exception: %s", e.what());
    }
}


void change_mode(const std_msgs::Bool msg)
{
    // Cambia el modo automatico
    mode_auto=msg.data;
    send_speed(0,0);
}

void change_motors_state(const std_msgs::Bool msg)
{
    // Cambia el estado de la habilitacion de los motores
    enabled_motors=msg.data;
    send_speed(0,0);
}

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

    ros::Subscriber sub_vel = n.subscribe("speed_desired", 1,
        change_speed);
    ros::Subscriber sub_motor = n.subscribe("enable_motors", 1,
        change_motors_state);
    ros::Subscriber sub_mode = n.subscribe("mode_automatic", 1,
        change_mode);

    ros::spin();

    return 0;
}
```

motors_com

codigo/andabata_teleoperation/motors_com.cpp

```
// Este programa es el encargado de comunicarse con los motores,
    envia una velocidad y lee la posicion de los encoders en
   cada ciclo

#include "ros/ros.h"
#include "serial/serial.h"
#include <ros/callback_queue.h>
#include "andabata_msgs/VelTelegram.h"
#include "andabata_msgs/WheelsDist.h"

using std::string;
using std::exception;

#define WHEELS_RADIUS 0.1
#define PULSES_TURN 504


serial::Serial motors;


std::string translate_vel(int num){
   // Conversion al lenguaje de la controladora de los motores
   num = 5 * num; // para ponerlo en el rango [-500, 500] poner
      5
   if(num<0)
      num=-num;
   std::string str;
   str[0] = (num/49+48);
   str[1] = (num - 49*(num/49) +48);
   return str;
}

// Cuando se recive una velocidad se envia
void send_velocity(const andabata_msgs::VelTelegram msg)
{
   try {
      std::string order = "SV";

      if (msg.vel_r<0){
         order += "-";
      }else{
         order += "+";
      }
```

```
        order += translate_vel(msg.vel_r).c_str();

        if (msg.vel_l<0){
            order += "-";
        }else{
            order += "+";
        }

        order += translate_vel(msg.vel_l).c_str();

        order += "\n";

        motors.write(order);

        motors.readline(20,"\r");

    } catch (exception &e) {
        printf("Unhandled Exception: %s", e.what());
    }
}



float pulses_to_dist(int pulses){
    return pulses*(2*M_PI*WHEELS_RADIUS)/PULSES_TURN;
}

// Lee la posicion de los encoders del robot, calcula la
    distancia recorrida por cada rueda y las publica
void read_encoders(void){
    static ros::NodeHandle n;
    static ros::Publisher pub_pos = n.advertise<andabata_msgs::
        WheelsDist>("wheel_dist", 20);

    //leer posiciones y descifrar
    motors.write("\n");
    motors.write("RP\n");
        // Leemos una linea, hasta el final de carro, que
            contendra angulo
    usleep(50);

    std::string message = motors.readline(20,"\r");
    printf("El mensaje es: %s\n", message.c_str());
    if (message.size() >= 14){
        if ((message[1]=='R') && (message[2]=='P')){
```

```
            printf("
               ----------------------------------------------------------\n")
               ;
            printf("Posicion:  %s",message.substr(3).c_str());

            // Separamos los dos campos de la trama y los
               convertimos a enteros
            std::string wheel_r = message.substr(4,7);
            int pulses_r = atoi(wheel_r.c_str());
            std::string wheel_l = message.substr(11);
            int pulses_l = atoi(wheel_l.c_str());
            printf("pulses_r: %s\t pulses_l: %s\n", wheel_r.c_str()
               , wheel_l.c_str());
            printf("pulses_r: %i\t pulses_l: %i\n", pulses_r,
               pulses_l);

            //calculamos la distancia que recorre cada rueda
            float dist_r = pulses_to_dist(pulses_r);
            float dist_l = pulses_to_dist(pulses_l);
            printf("dist_d: %f\t dist_i: %f\n", dist_r, dist_l);

            // construimos el message
            andabata_msgs::WheelsDist msg;
            msg.ros_time = ros::Time::now();
            msg.dist_r = dist_r;
            msg.dist_l = dist_l;

            // publicamos el mensaje
            pub_pos.publish(msg);

        }else{
            printf("El message es corrupto1.\n");
        }
    }else{
        printf("El message es corrupto2.\n");
    }
}


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

    ros::Subscriber sub_vel = n.subscribe("speed_motors", 1,
        send_velocity);
```

```
    // Capturamos los parametros, si no es posible se les asigna
       un valor por defecto
    int baudrate;
    std:string device;

    n.param("motors_baudrate", baudrate, 38400);
    n.param<std::string>("motors_id", device, "/dev/
       control_ruedas");

    // Iniciamos la comunicacion serie
    motors.setPort(device);
    motors.setBaudrate(baudrate);
    serial::Timeout to=serial::Timeout::simpleTimeout(1000);
    motors.setTimeout(to);
    motors.open();



    ros::Rate r(10); // ajustar a la frecuencia de muestreo


    while(ros::ok()){
       // Se comprueva si hay que enviar alguna velocidad, en
          caso de que si, se envia
       ros::getGlobalCallbackQueue()->callAvailable(ros::
          WallDuration(0.01)); //spinOnce con tiempo

       // Se lee la posicion de los encoders
       read_encoders();


       r.sleep();

    }
    return 0;
}
```

system_control

codigo/andabata_teleoperation/system_control.py

```
#!/usr/bin/python
import rospy
import roslaunch
import time

from std_msgs.msg import Bool

actives = False
changes = False


def callback(data):
    global actives
    global changes
    print 'actives0 = ' + str(actives)
    print 'changes0 = ' + str(changes)
    actives = not(actives)
    changes = True
    print 'actives1 = ' + str(actives)
    print 'changes1 = ' + str(changes)



if __name__ == '__main__':
    rospy.init_node('system_control')
    global actives
    global changes

    rospy.Subscriber("request_off", Bool, callback)


    node1 = roslaunch.core.Node('laser_uno', 'laser_com', name='
       laser_com')
    node2 = roslaunch.core.Node('laser_uno', 'laser_position',
       name='laser_position')
    node3 = roslaunch.core.Node('laser_uno', '
       laserScan_to_cloud2', name='laserScan_to_cloud2')
    node4 = roslaunch.core.Node('urg_node', 'urg_node', name='
       urg_node')
    node5 = roslaunch.core.Node('octomap_server', '
       octomap_server_node', name='octomap_server_node',
       remap_args=[('cloud_in', '/my_cloud')])
```

```
launch = roslaunch.scriptapi.ROSLaunch()
launch.start()

r = rospy.Rate(10)
while not rospy.is_shutdown():
    r.sleep()
    if changes:
        if actives:
            process1 = launch.launch(node1)
            process2 = launch.launch(node2)
            process3 = launch.launch(node3)
            process4 = launch.launch(node4)
            process5 = launch.launch(node5)
        else:
            process1.stop()
            process2.stop()
            process3.stop()
            process4.stop()
            process5.stop()
        changes = False
```

odometry

codigo/andabata_teleoperation/odometry.cpp

```
// Recive la distancia recorrida por las ruedas, calcula la
   odometria y publica los datos

#include <ros/ros.h>
#include <tf/transform_broadcaster.h>
#include <nav_msgs/Odometry.h>
#include <andabata_msgs/WheelsDist.h>

// Parametros del modelo del robot
#define xIRC 0.4043261
#define alpha 1.0111


void odometry(const andabata_msgs::WheelsDist& msg)
{
   static ros::NodeHandle n;
   static ros::Publisher odom_pub = n.advertise<nav_msgs::
      Odometry>("odom", 50);

   static tf::TransformBroadcaster odom_broadcaster;

   // Posicion inicial
   static double x = 0.0;
   static double y = 0.0;
   static double th = 0.0;

   double vx, vy, vth;

   static ros::Time last_time = msg.ros_time - ros::Duration
      (0.01);
   ros::Time current_time = msg.ros_time;

   // Leemos la distancia que recorre cada rueda
   float dist_r = msg.dist_r;
   float dist_l = msg.dist_l;

   // Calculamos el tiempo entre las dos ultimas medidas
   double dt = (current_time - last_time).toSec();

   // Calculamos la velocidades de cada rueda
   float vel_r = dist_r/dt;
   float vel_l = dist_l/dt;
```

```
// Calculamos las velocidades segun los ejes coordenados
vx = 0.0;
vy = alpha / 2 * (vel_r + vel_l);
vth = alpha / (2 * xIRC) * (vel_r - vel_l);

// Calculamos la odometria del robot integrando las
   velocidades
double delta_x = (vx * cos(th) - vy * sin(th)) * dt;
double delta_y = (vx * sin(th) + vy * cos(th)) * dt;
double delta_th = vth * dt;
x += delta_x;
y += delta_y;
th += delta_th;
// Dado que la odometria es 6DOF necesitaremos un quaternion
   creado por Yaw
geometry_msgs::Quaternion odom_quat = tf::
   createQuaternionMsgFromYaw(th);

// Publicamos la informacion de los ejes coordenados del
   robot respecto a world
geometry_msgs::TransformStamped odom_trans;
odom_trans.header.stamp = current_time;
odom_trans.header.frame_id = "world";
odom_trans.child_frame_id = "robot";

odom_trans.transform.translation.x = x;
odom_trans.transform.translation.y = y;
odom_trans.transform.translation.z = 0.0;
odom_trans.transform.rotation = odom_quat;

odom_broadcaster.sendTransform(odom_trans);

// Publicamos el mensaje de odometria
nav_msgs::Odometry odom;
odom.header.stamp = current_time;
odom.header.frame_id = "world";

odom.pose.pose.position.x = x;
odom.pose.pose.position.y = y;
odom.pose.pose.position.z = 0.0;
odom.pose.pose.orientation = odom_quat;

odom.child_frame_id = "robot";
odom.twist.twist.linear.x = vx;
odom.twist.twist.linear.y = vy;
odom.twist.twist.angular.z = vth;
```

```
    odom_pub.publish(odom);

    // Actualizamos el tiempo
    last_time = current_time;
}


int main(int argc, char** argv){
    ros::init(argc, argv, "odometry");

    ros::NodeHandle n;
    ros::Subscriber sub_dist = n.subscribe("wheel_dist", 20,
        odometry);

    ros::spin();

}
```

C.2. Nodos del paquete laser_uno

rotation_speed_input

codigo/laser_uno/rotation_speed_input.cpp

```
// Este programa pide al usuario que escriba una orden, este la
   lee del teclado, la almacena en un string y la publica en el
   topic "rotation_speed_laser".

#include "ros/ros.h"
#include "std_msgs/String.h"

int main(int argc, char **argv)
{
   char speed[20];

   ros::init(argc, argv, "rotation_speed_input");

   ros::NodeHandle n;
   // Donde se van a publicar las ordenes
   ros::Publisher pub_speed = n.advertise<std_msgs::String>("
      rotation_speed_laser", 2);
   // La frecuencia con la que se va a ejecutar el while
   ros::Rate r(5);

   while (ros::ok())
   {
      std_msgs::String msg;

      printf("Write an order: ");
      scanf("%s", speed);

      msg.data = speed;

      pub_speed.publish(msg);

      r.sleep();
   }
  return 0;
}
```

laser_com

codigo/laser_uno/laser_com.cpp

```
// Inicia la velocidad de rotacion de la base del laser a un
   valor predeterminado, queda a la espera de ordenes que enviar
    al laser, antes de salir para la rotacion del laser.

#include "ros/ros.h"
#include "std_msgs/String.h"
#include "serial/serial.h"

using std::string;
using std::exception;



serial::Serial laser;

// Cuando se recibe una orden se manda al laser
void send_order(const std_msgs::String::ConstPtr& msg)
{
   try {

      laser.write(msg->data);
      laser.write("\n");

   } catch (exception &e) {
      printf("Unhandled Exception: %s", e.what());
  }
}

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

   ros::Subscriber sub_order = n.subscribe("rotation_speed_laser
      ", 1000, send_order);


   // Capturamos los parametros, si no se encuentran en el
      parameter server se utilizan los valores predefinidos
   int baudrate;
   std:string device;

   n.param("base_laser_baudrate", baudrate, 500000);
```

```
    n.param<std::string>("base_laser_id", device,"/dev/laser_uno"
        );

    // Se inicializa la comunicacion serie
    laser.setPort(device);
    laser.setBaudrate(baudrate);
    serial::Timeout to=serial::Timeout::simpleTimeout(1000);
    laser.setTimeout(to);
    laser.open();

    // El laser comienza a girar
    laser.write("SV+1500\n");

    // Quedamos a la espera de nuevas ordenes que enviar al laser
    ros::spin();

    // Antes de salir del programa paramos la rotacion del laser
    laser.write("SV+0000\n");

    return 0;
}
```

laser_position

codigo/laser_uno/laser_position.cpp

```
// Leemos la posicion del laser respecto de la base, publicamos
   la informacion y actualizamos sus ejes coordenados "laser"
   respecto a la del  robot "robot"


#include "ros/ros.h"
#include <tf/transform_broadcaster.h>
#include "serial/serial.h"
#include "andabata_msgs/LaserEvent.h"

using std::string;

#define theta_0 -0.5148723  //calibracion extrinseca theta_0 =
   -29.5  (150.5-180)
#define beta_0 0.008726646        //calibracion intrinseca
   beta_0 = 0.5     alpha_0 = -0.02
#define alpha_0 -0.000349066


// El encoder del laser tiene 45000 puntos por vuelta
float pulses_to_rad(float pulses){
   return (pulses*2*M_PI)/45000 + theta_0;
}

float deg_to_rad(float degrees){
   return (degrees*M_PI)/180;
}


int main(int argc, char **argv)
{

   std::string pos_str, time_str;

   ros::init(argc, argv, "laser_position");

   ros::NodeHandle n;

   ros::Publisher pub_event = n.advertise<andabata_msgs::
      LaserEvent>("position_laser", 1000);

   tf::TransformBroadcaster broadcaster;
```

```
ros::Rate r(40);

// Capturamos los parametros, en caso de que no esten
   disponibles se les asigna el valor por defecto
int baudrate;
std::string device;

n.param("base_laser_baudrate", baudrate, 500000);
n.param<std::string>("base_laser_id", device,"/dev/laser_uno"
   );

// Iniciamos la comunicacion serie
serial::Serial laser(device, baudrate, serial::Timeout::
   simpleTimeout(1000));

// Creacion de la transformacion para la position inicial del
    laser
// El laser se encuentra, aproximadamente, a 72 cm del suelo,
    donde se encuentran los ejes coordenados del robot
tf::Transform laserTransform;
laserTransform.setOrigin(tf::Vector3(0.0, 0.0, 0.72));
tf::Quaternion q, calibration;
// Los parametros de calibracion no estan en los mismos ejes
   que en el trabajo ni coinciden los ejes, porque en el
   trabajo se tomaron ejes diferentes.
calibration.setEuler(deg_to_rad(-90)+beta_0, 0, -alpha_0);
laserTransform.setRotation(calibration);

// Inicializamos el envio de datos de la posicion por parte
   del laser
laser.write("EN\n");

while (ros::ok())
{
   // Leemos una linea, hasta el final de carro, que
      contrndra angulo y tiempo en el que se encuentr el
      laser
   std::string message = laser.readline(20,"\r");
   if (message.size() >= 14){
      // Separamos los dos campos de la trama y los
         convertimos a enteros
      std::size_t pos_plus=message.rfind("+");     // Empieza
          buscando desde el final hasta encontrar el + que va
          delante del tiempo
      pos_str=message.substr(0,pos_plus);
      int position=atoi(pos_str.c_str());
```

```
            time_str=message.substr(pos_plus);
            int time=atoi(time_str.c_str());
            printf("pos: %i\n", position);

            // Construimos y publicamos el message "LaserEvent"
            andabata_msgs::LaserEvent msg;
            msg.position = position;
            msg.laser_time = time;
            msg.ros_time = ros::Time::now();
            pub_event.publish(msg); //envio del dato

            // Actualizamos la position del laser
            q.setRPY(0, 0, pulses_to_rad(position));
            laserTransform.setRotation(q*calibration);

            // Enviamos la transformacion
            broadcaster.sendTransform(tf::StampedTransform(
               laserTransform, ros::Time::now(),"robot", "laser"));

         }
         r.sleep();
      }

      // Paramos el envio de datos por parte del laser antes de
         salir
      laser.write("DI\n");

      return 0;
   }
```

laserScan_to_cloud2

codigo/laser_uno/laserScan_to_cloud2.cpp

```
// Transforma los datos del tipo laserScan a datos del tipo
   pointCloud2

#include "ros/ros.h"
#include "tf/transform_listener.h"
#include "sensor_msgs/PointCloud2.h"
#include "tf/message_filter.h"
#include "message_filters/subscriber.h"
#include "laser_geometry/laser_geometry.h"

// first es el topic donde publicar los laserScan
// my_cloud es el topic donde se publican los pointCloud2
// laser es los ejes del laser

class LaserScanToPointCloud{

public:

  ros::NodeHandle n_;
  laser_geometry::LaserProjection projector_;
  tf::TransformListener listener_;
  message_filters::Subscriber<sensor_msgs::LaserScan> laser_sub_
     ;
  tf::MessageFilter<sensor_msgs::LaserScan> laser_notifier_;
  ros::Publisher scan_pub_;

  LaserScanToPointCloud(ros::NodeHandle n) :
    n_(n),
    laser_sub_(n_, "first", 10),
    laser_notifier_(laser_sub_,listener_, "laser", 10)
  {
    laser_notifier_.registerCallback(
      boost::bind(&LaserScanToPointCloud::scanCallback, this, _1
         ));
    laser_notifier_.setTolerance(ros::Duration(0.01));
    scan_pub_ = n_.advertise<sensor_msgs::PointCloud2>("/
       my_cloud",1);
  }

  void scanCallback (const sensor_msgs::LaserScan::ConstPtr&
     scan_in)
  {
    sensor_msgs::PointCloud2 cloud;
```

```
    try
    {
        projector_.transformLaserScanToPointCloud(
          "laser",*scan_in, cloud,listener_);
    }
    catch (tf::TransformException& e)
    {
        std::cout << e.what();
        return;
    }

    scan_pub_.publish(cloud);

  }
};

int main(int argc, char** argv)
{

  ros::init(argc, argv, "laserScan_to_cloud2");
  ros::NodeHandle n;
  LaserScanToPointCloud lstopc(n);

  ros::spin();

  return 0;
}
```

C.3. Mensajes personalizados del paquete andabata_msgs

JoystickOrder

codigo/andabata_msgs/JoystickOrder.msg

```
int16 id
int16 wheel_r
int16 wheel_l
```

LaserEvent

codigo/andabata_msgs/LaserEvent.msg

```
int32 position
int32 laser_time
time ros_time
```

VelTelegram

codigo/andabata_msgs/VelTelegram.msg

```
int16 vel_r
int16 vel_l
```

WheelsDist

codigo/andabata_msgs/WheelsDist.msg

```
time ros_time
float32 dist_r
float32 dist_l
```

WheelSpeed

codigo/andabata_msgs/WheelSpeed.msg

```
bool wheel
int16 speed
```

C.4. *Launchs* y *scripts*

Launch andabata_teleoperation

codigo/launch/teleoperation.launch

```
<!--
Lanza todos los programas necesarios para controlar el robot
    andabata y obtener la odometria
Se pueden remapear los nombre de los topics. Para mas detalle
    consulta el codigo de los programas
-->

<launch>
      <param name="motors_id" value="/dev/control_ruedas" />
      <param name="motors_baudrate" value="38400" />

   <node pkg="andabata_teleoperation" type="mobile_receptor.py"
       name="mobile_receptor" respawn="true">
      <!-- <remap from="baseLaser_ordenes" to="hello"/>--> </
          node>

   <node pkg="andabata_teleoperation" type="slider" name="slider
       " respawn="true">
      <!-- <remap from="baseLaser_ordenes" to="hello"/>--> </
          node>

   <node pkg="andabata_teleoperation" type="joystick_imu" name="
       joystick_imu" respawn="true">
      <!-- <remap from="baseLaser_ordenes" to="hello"/>--> </
          node>

   <node pkg="andabata_teleoperation" type="joystick_multiplexor
       " name="joystick_multiplexor" respawn="true">
      <!-- <remap from="baseLaser_ordenes" to="hello"/>--> </
          node>

   <node pkg="andabata_teleoperation" type="motors_control" name
       ="motors_control" respawn="true">
      <!-- <remap from="baseLaser_ordenes" to="hello"/>--> </
          node>

   <node pkg="andabata_teleoperation" type="motors_com" name="
       motors_com" respawn="true">
      <!-- <remap from="baseLaser_ordenes" to="hello"/>--> </
          node>
```

```
    <node pkg="andabata_teleoperation" type="odometry" name="
        odometry" respawn="true">
        <!-- <remap from="baseLaser_ordenes" to="hello"/>--> </
            node>
</launch>
```

***Launch* laser_uno**

codigo/launch/laser_uno.launch

```
<!--
No tiene en cuenta el control ejercido por system_control
-->


<!--
Inicia todo lo necesario para que el laser comience a funcionar.
Es necesario tener la odometria funcionando, ya que necesita
    tener la cadena de ejescoordenados hasta uno estatico "world"
-->

<launch>

    <group>
        <param name="base_laser_id" value="/dev/laser_uno" />
        <param name="base_laser_baudrate" value="500000" />

        <node pkg="laser_uno" type="rotation_speed_input" name="
            rotation_speed_input" output="screen" respawn="true"> <
            /node>

        <node pkg="laser_uno" type="laser_com" name="laser_com"
            respawn="true"> </node>

        <node pkg="laser_uno" type="laser_position" name="
            laser_position" respawn="true"> </node>
    </group>

    <node name="urg_node" pkg="urg_node" type="urg_node" respawn=
        "true">
        <param name="ip_address" value="192.168.0.10" />
        <param name="frame_id" value="/laser" />
        <param name="calibrate_time" value="true" />
        <param name="publish_intensity" value="false" />
    </node>


    <node pkg="laser_uno" type="laserScan_to_cloud2" name="
        laserScan_to_cloud2" respawn="true"> </node>


    <node pkg="octomap_server" type="octomap_server_node" name="
        octomap_server_node" respawn="true">
```

```
        <param name="resolution" value="0.1" />

        <!-- fixed map frame (set to 'map' if SLAM or localization
             running!) -->
        <param name="frame_id" type="string" value="world" />

        <!-- maximum range to integrate (speedup!) -->
        <param name="sensor_model/max_range" value="30.0" />

        <!-- data source to integrate (PointCloud2) -->
        <remap from="cloud_in" to="/my_cloud" />

    </node>


</launch>
```

***Launch* laser_parametre**

codigo/launch/laser_parametres.launch

```
<!--
Es el que se debe lanzar si se quiere tener control de cuando se
    crean mapas 3D
-->

<!--
Inicia todo lo necesario para que el laser comience a funcionar.
Es necesario tener la odometria funcionando, ya que necesita
   tener la cadena de ejescoordenados hasta uno estatico "world"
-->

<launch>


   <param name="base_laser_id" value="/dev/laser_uno" />
   <param name="base_laser_baudrate" value="500000" />

   <param name="/urg_node/ip_address" value="192.168.0.10" />
   <param name="/urg_node/frame_id" value="/laser" />
   <param name="/urg_node/calibrate_time" value="true" />
   <param name="/urg_node/publish_intensity" value="false" />

   <param name="/octomap_server_node/resolution" value="0.1" />
   <param name="/octomap_server_node/frame_id" type="string"
      value="world" />
   <param name="/octomap_server_node/sensor_model/max_range"
      value="30.0" />

   <node pkg="andabata_teleoperation" type="system_control.py"
      name="system_control" respawn="true"> </node>


</launch>
```

***Script* start_andabata**

codigo/script/tstart_andabata.sh

```
#! /bin/bash -x


# inicializamos nuestro workspace de ROS
source /home/am/ros_workspace/devel/setup.bash

# esperamos y lanzamos las aplicaciones (el tiempo de espera no
    es necesario)
sleep 10
roscore &
sleep 1
rosrun rviz rviz -d ~/.rviz/andabata_rviz.rviz &
sleep 5
roslaunch andabata_teleoperation teleoperation.launch &
sleep 2
roslaunch laser_uno laser_parametres.launch &



# comprobar si la carpeta maps esta, si no crearla
if [ ! -d "/home/am/maps" ]; then
    mkdir /home/am/maps
fi


# cada 10 segundo guardar el mapa
while true; do
    sleep 10
    NOW=`date +"%y-%m-%d_%T"`
    # si octomap esta funcionando
    OCTOMAP_OK=`rostopic list | grep /octomap | wc -l`
    if [ $OCTOMAP_OK -gt "0" ]; then
        rosrun octomap_server octomap_saver -f /home/am/maps/
            map_$NOW.ot
    fi

done
```

Bibliografía

[1] J. L. Martínez, "Proyecto de Investigación de Excelencia de la Junta de Andalucía P10-TEP-6101-R," http://www.uma.es/cms/base/ver/section/document/73618/navegacion-autonoma-de-un-robot-movil-4x4/, Mar. 2013.

[2] A. Hornung, K. M. Wurm, M. Bennewitz, C. Stachniss, and W. Burgard, "OctoMap: An efficient probabilistic 3D mapping framework based on octrees," *Autonomous Robots*, vol. 34, pp. 189–206, 2013, software available at http://octomap.github.com. [Online]. Available: http://octomap.github.com

[3] P. Bovbel and F. Andargie, "Mover-bot," https://code.google.com/p/mover-bot/.

[4] C. Rockey and A. Furlan, "ROS Android Sensors Driver," https://github.com/ros-android/android_sensors_driver.

[5] A. Mandow, J. L. Martínez, J. Morales, J. L. Blanco, A. García-Cerezo, and J. González, "Experimental kinematics for wheeled skid-steer mobile robots," in *Proc. of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, San Diego, CA, 2007, pp. 1222 – 1227.

[6] J. L. Martínez, J. Morales, A. J. Reina, A. Mandow, A. Pequeño-Boter, and A. García-Cerezo, "Construction and Calibration of a Low-cost 3D Laser Scanner with Full Field-of-view for Mobile Robots," in *Proc. of the IEEE International Conference on Industrial Technology*, Sevilla, España, 2015, pp. 149–154.

[7] J. Morales, J. L. Martínez, A. Mandow, A. Pequeño-Boter, and A. García-Cerezo, "Design and Development of a Fast and Precise Low-Cost 3D Laser Rangefinder," in *Proc. of the IEEE International Conference on Mechatronics*, Estambul, Turquía, 2011, pp. 621 – 626.

[8] J. Morales, J. L. Martínez, A. Mandow, A. J. Reina, A. Pequeño-Boter, and A. García-Cerezo, "Boresight calibration of construction misalignments for 3D scanners built with a 2D laser rangefinder rotating on its optical center," *Sensors*, vol. 14, no. 11, pp. 20 025 – 20 040, 2014.

[9] R. B. Rusu and S. Cousins, "3D is here: Point Cloud Library (PCL)," in *Proc. of the IEEE International Conference on Robotics and Automation*, Shanghai, China, May 2011, pp. 1–4.

[10] J. Valbuena, "Remodelación de los sistemas de alimentación y control en el robot Auriga-α," Proyecto Final de Carrera (E.T.S.I. Industrial), Universidad de Málaga, 2014.

[11] W. Foundation, "Wikipedia: the free encyclopedia," http://en.wikipedia.org/, Jan. 2001.

[12] A. ROS, "ROS," http://www.ros.org/.

[13] W. Garage, "Willow garage," https://www.willowgarage.com/, 2006.

[14] J. M. O'Kane, "A gentle introduction to ROS," http://www.cse.sc.edu/~jokane/agitr/, University of South Carolina, 2014.

[15] "Ubuntu guia," http://www.ubuntu-guia.com/.

[16] "askubuntu," http://askubuntu.com/.

[17] R. Association, "robohub," http://robohub.org/.

[18] "stackoverflow," http://stackoverflow.com/.

[19] E. T. S. de Ingenieros de Sevilla, "e-reding," http://bibing.us.es/proyectos/.