




**Universidad Nacional de Educación a Distancia**
**Universidad Complutense de Madrid**

## Proyecto Final de Máster

Máster en Ingeniería de Sistemas y de Control

# Navegación autónoma de un robot diferencial usando un dispositivo Kinect

Alberto Morales Guerrero

---

Directores:
Dr. Dictino Chaos García
Dr. Luis De La Torre Cubillo

---

Curso académico 2016/2017 - Convocatoria Junio






## Proyecto Final de Máster

Máster en Ingeniería de Sistemas y de Control

# Navegación autónoma de un robot diferencial usando un dispositivo Kinect

**Proyecto tipo A**

Alberto Morales Guerrero

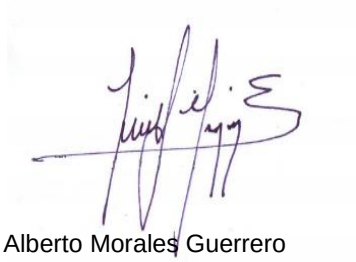
---

Directores:
Dr. Dictino Chaos García
Dr. Luis De La Torre Cubillo

# Autorización

Autorizamos a la Universidad Complutense y a la UNED a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la memoria de este Trabajo Fin de Máster, como el código, la documentación y/o el prototipo desarrollado.

Firmado:



Alberto Morales Guerrero

DECLARACIÓN JURADA DE AUTORÍA DEL TRABAJO CIENTÍFICO,

PARA LA DEFENFA DEL TRABAJO FIN DE MASTER

Fecha: 30 Mayo de 2017

Quien se suscribe:


**Autor:** Alberto Morales Guerrero

**DNI:** AM8875577

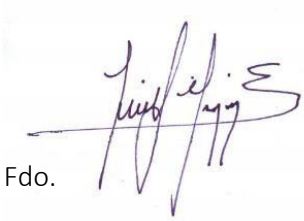

Hace Constar que es el autor del trabajo:

# Navegación Autónoma de un robot diferencial usando un dispositivo Kinect

En tal sentido, manifiesto la originalidad de la conceptualización del trabajo, interpretación de datos y la elaboración de las conclusiones, dejando establecido que aquellos aportes intelectuales de otros autores, se han referenciado debidamente en el texto de dicho trabajo.

DECLARACIÓN:

- ✓ Garantizo que el trabajo que remito es un documento original y no ha sido publicado, total ni parcialmente, en otra revista.
- ✓ Certifico que he contribuido directamente al contenido intelectual de este manuscrito, a la génesis y análisis de sus datos, por lo cual estoy en condiciones de hacerme públicamente responsable de él.
- ✓ No he incurrido en fraude científico, plagio o vicios de autoría; en caso contrario, aceptaré las medidas disciplinarias sancionadoras que correspondan.

Fdo. 

# Resumen

La robótica móvil es un campo de amplia investigación, debido a la expansión y utilización por parte de los sectores de producción (Industria) y servicios, en donde se incluyen sistemas robotizados para realizar diferentes tipos de tareas, como por ejemplo: desplazarse de un punto a otro, evadir obstáculos e identificar el medio que le rodea.

En este trabajo se presenta el diseño y construcción de un robot tipo diferencial, compuesto por un set de sensores - actuadores y una tarjeta Raspberry Pi, que permita la realización de pruebas experimentales en torno a estrategias y algoritmos relacionados con robótica móvil. A su vez, se realiza el modelo computacional del robot en el formato urdf *(Unified Robot Description Format)* el cual puede ser leído desde diferentes paquetes de software de robótica, permitiendo la simulación del sistema, y posteriormente la ejecución de las pruebas experimentales con el robot físico.

El robot construido integra el framework *Robot Operating System*, con el objeto de proporcionar flexibilidad al robot, permitiendo la programación distribuida y modular del sistema. En esta memoria se presentan los algoritmos empleados para la comunicación y control de los actuadores y sensores empleados.

Una de las tareas comunes en robótica móvil es la construcción de mapas. Por tanto, se implementa una estrategia basada en un filtro de partículas con el objetivo de realizar el mismo. Para su construcción, el usuario controla los desplazamientos del robot remotamente, mientras el algoritmo implementado en el robot almacena y procesa la formación para realizar el mapa de tipo rejilla. Una de las dificultades en la construcción del mapa se presenta en la utilización de recursos físicos del sistema, lo cual es solventado disminuyendo la cantidad de partículas.

Finalizada la construcción del mapa se implementa una estrategia de navegación, que emplea el algoritmo *Dynamic Window Approach,* y un nodo *Adaptive Monte Carlo Localization* para la ubicación del robot. Así, el robot realiza la planificación de trayectorias. La trayectoria es seguida por el robot y puede ser actualizada cuando se encuentran obstáculos en su recorrido. Por último se realizan las pruebas tanto en simulación como en el robot desarrollado.

**Palabras clave:** *Robot diferencial, ROS, Navegación, Mapeo, Filtro de partículas,SLAM, Kinect*

# Contenido


**1 Preámbulo** **8**
- 1.1 Introducción . . . 8
- 1.2 Estado del Arte . . . 8

**2 Diseño y Construcción del Prototipo** **11**
- 2.1 Requerimiento de diseño . . . 11
- 2.2 Diseño del chasis y componentes mecánicos . . . 11
- 2.3 Modelo matemático del robot . . . 13
- 2.4 Especificaciones actuadores, sensores y etapa de potencia. . . . 15
  - 2.4.1 Motores, encoder y driver . . . 15
  - 2.4.2 Sensor de ultrasonidos . . . 17
  - 2.4.3 Compás . . . 17
  - 2.4.4 Kinect como sensor LIDAR . . . 18
  - 2.4.5 Etapa de Potencia . . . 20

**3 Diseño del Software** **22**
- 3.1 Arquitectura de ROS . . . 22
  - 3.1.1 Sistema de Archivos en ROS . . . 22
  - 3.1.2 Computación gráfica . . . 23
- 3.2 Package descripción del Robot . . . 23
- 3.3 Diseño firmware para la comunicación con el hardware . . . 26
- 3.4 Cálculo de la odometría . . . 30
- 3.5 Lectura del Kinect . . . 30
- 3.6 Construcción del Mapa . . . 33
- 3.7 Navegación . . . 36

**4 Resultados** **41**
- 4.1 Simulaciones en rviz y gazebo . . . 41
- 4.2 Pruebas de funcionamiento . . . 44
  - 4.2.1 Control en Remoto del Robot . . . 44
  - 4.2.2 Construcción de un mapa . . . 45
  - 4.2.3 Navegación Autónoma . . . 46
  - 4.2.4 Evasión de Obstáculos . . . 48

**5 Conclusiones** **53**
- 5.1 Trabajo Futuro . . . 54

**A Instalación y Configuración** **56**
A.1 Configuración tarjeta . . . . . 56
A.2 Instalación de ROS . . . . . 57
A.3 Instalación de packages . . . . . 57
A.3.1 Descripción de los *packages* . . . . . 58
A.4 Simulador y pruebas . . . . . 61

**B Estimación coste del robot** **63**

**C Listado de siglas** **64**

**Bibliografía** **65**

# Lista de Figuras


1.1 Pi Robot . . . 9
1.2 Robot monocular con Raspberry Pi . . . 9
1.3 Turtlebot 2 . . . 10
1.4 Robot Pioneer 3 DX . . . 10

2.1 Diseño base del Robot . . . 12
2.2 Diseño de soportes para impresión 3D. . . . 12
2.3 Modelo en Bleder del robot. . . . 13
2.4 Prototipo del robot finalizado . . . 13
2.5 Configuración de un robot diferencial . . . 14
2.6 Modelo del Robot diferencial . . . 14
2.7 Motor 37D. . . . 16
2.8 Enconder motores 37D mm. . . . 16
2.9 Conexión Motores y etapa de potencia . . . 17
2.10 Compás CMPS03 . . . 18
2.11 Microcontrolador A-Star 32U4 (ATmega32U4 32U4) . . . 19
2.12 Sistema de alimentación a los diferentes módulos. . . . 21

3.1 Interconexión de nodos para el modelo del Robot . . . 26
3.2 Representación del modelo creado en el archivo .urdf . . . 26
3.3 Sensor ultrasonido . . . 28
3.4 Visualización del encoder en el osciloscopio . . . 29
3.5 Elementos relevantes del nodo Odometría . . . 31
3.6 Listado de Topics generados por el Kinect empleando Freenect. . . . 32
3.7 Nodo *camera* y nodo *depth_to_laserscan* . . . 34
3.8 Identificación de celdas ocupadas y vacías en el mapa . . . 35
3.9 Módulo de construcción del mapa . . . 37
3.10 Modulo de navegación. Tomado de `http://wiki.ros.org/navigation` Modificado de acuerdo al desarrollo presentado en este documento. . . . 38
3.11 Parámetros nodo AMCL . . . 38

4.1 Interfaz de simulación del robot interactuando en un ambiente. . . . 41
4.2 Interfaz de rviz con lectura de diferentes Topic . . . 43
4.3 Carga de Topic al ambiente rviz . . . 43
4.4 Cámara de profundidad en rviz . . . 44
4.5 Control remoto del Robot . . . 44
4.6 Lectura del sensor Kinect. . . . 45
4.7 Construcción del Mapa . . . 45

4.8 Mapa construido por el robot . . . . . 46
4.9 Robot realizando la construcción del Mapa . . . . . 46
4.10 Interfaz para verificar la navegación del robot . . . . . 47
4.11 Identificación de los Topic en rviz . . . . . 48
4.12 Navegación del robot . . . . . 49
4.13 Secuencia de navegación con obstáculos . . . . . 49
4.14 Experimentación con el robot . . . . . 49
4.15 Experimentación con el robot sin obstáculos agregados . . . . . 51
4.16 Experimentación del robot con obstáculos agregados . . . . . 52

A.1 Instalación Ubuntu Mate . . . . . 56
A.2 Organización del workspace en ROS . . . . . 58
A.3 Simulación del robot . . . . . 61
A.4 Modificación archivo .bashrc . . . . . 62

# Capítulo 1

# Preámbulo

## 1.1 Introducción

El objetivo de este proyecto es construir un robot autónomo dotado de un dispositivo Kinect y realizar mapas de su entorno utilizando diferentes técnicas. El procesamiento del robot está bajo el sistema embebido Raspberry Pi; administrado por el sistema operativo *Ubuntu Mate* en el que se instala *Robot Operating System* **(ROS)** para integrar las diversas partes del robot.

En particular la realización del proyecto se compone de las siguientes etapas:

1. Diseño del robot diferencial y su correspondiente simulación.
2. Puesta a punto del sistema donde se incluye el montaje del robot, integración de sensores, actuadores y procesamiento de las imágenes obtenidas con el Kinect.
3. Captura de datos del robot usando control manual: En esta fase se desplaza el robot empleando comandos generados manualmente por el operador y se registra toda la información odométrica así como los datos capturados por el Kinect.
4. Creación de mapas: A partir de la información registrada en las pruebas de la fase anterior se ensayan diversos parámetros para la de creación de mapas y determinar los que resultan más adecuados.
5. Implementación de la navegación autónoma y validación del sistema de navegación: En esta fase se implementa la estrategia de creación de mapas y navegación empleadas en la simulación, validando su funcionamiento en el robot real.

## 1.2 Estado del Arte

El desarrollo de la robótica móvil, en particular el estudio del control y diseño de sistemas autónomos inicio en la década de los 60's. En Stanford University se construyeron prototipos como *Stanford Car* desarrollado por diversos investigadores como Adams (1961), Braisted (1963), Goldberg and Siegwart (2002). Desde entonces la rama de la robótica móvil se ha expandido y son numerosos los robots y estudios que se han realizado. En este apartado se realiza una revisión bibliográfica de algunos con mayor relevancia que se han construido en los últimos años, y en

particular los que se relacionan con el desarrollo de esta propuesta, en torno a la implementación de algoritmos de navegación y mapeo, así como la implementación de ROS.

Robot Pi es un proyecto desarrollado por Goebel (2014), quien es autor de varios libros orientados al desarrollo en *Robot Operating System* **(ROS)**. El proyecto fue diseñado para el estudio de diferentes tipos de algoritmos de navegación y el diseño de mapas; aunque su estructura es diferencial, ha tenido diversas modificaciones tanto holónomicos como no holónomicos. En su evolución se han agregado sensores de mayor calidad y un par de brazos mecánicos. En la figura 1.1 se observa el Robot Pi acompañado del Turtlebot, el cual es un reconocido robot en el campo de la robótica móvil.

Aunque el sistema de control inicial no estaba desarrollado bajo una arquitectura especifica, este

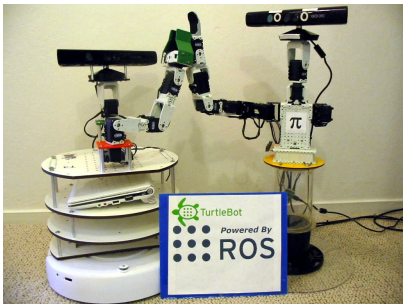



Figura 1.1: Pi Robot

desarrollo se hace significativo en relación a los resultados obtenidos en la utilización de ROS como el sistema operativo del robot.

Por otro lado, en Cornell Unversity, Gautham Ponnu (2016) ha desarrollado un robot de tipo diferencial, diseñado con una cámara monocular para realizar *Simultaneous Localization And Mapping* **(SLAM)**; el robot es construido empleando un nodo cliente-servidor. La unidad de procesamiento en el robot es una tarjeta Raspberry Pi empleada para ejecutar ROS como cliente y adicionalmente se conecta con un computador portátil como servidor para realizar el procesamiento relacionado con SLAM. Dicho trabajo se focaliza en el desarrollo de un algoritmo para realizar odometría visual con un sistema mono-cámara.

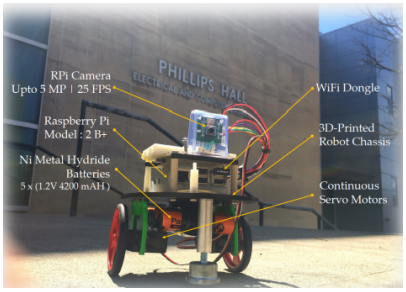



Figura 1.2: Robot monocular con Raspberry Pi

La potencialidad de ROS para el desarrollo de algoritmos y prototipos en robótica es amplia como se observa en Gautham Ponnu (2016), Goebel (2014), Cousins (2012) y en particular en el trabajo de (Omara and Sahari, 2015) para el que se implementa un robot diferencial en donde el objeto de estudio está centrado en el análisis de la efectividad del Kinect como sensor láser y su

comparación con otros posibles sensores de mayor coste. Para este desarrollo se utilizó un robot como el mostrado en la figura 1.3, en donde el nivel de control se centra un computador portátil, el cual es trasportado sobre su estructura.



Figura 1.3: Turtlebot 2

En el robot Pioneer 3-DX en Zaman et al. (2011) se realiza un estudio sobre el desarrollo de mapas en diferentes ambientes; así como algunos factores asociados a cada uno de los ambientes que pueden llegar a afectar la construcción del mapa, la localización y la navegación autónoma. Para ello se realizan tres experimentos: dos sobre un ambiente artificial y uno en un entorno real. Estos



Figura 1.4: Robot Pioneer 3 DX

robots han sido desarrollados con el objetivo de estudiar técnicas para realizar SLAM; empleando diferentes frameworks para el procesamiento de la información, así como diferentes sensores. Esta memoria presenta el desarrollo de un robot con un sistema embebido que integra ROS, un sensor Kinect y la opción de implementar diferentes tipos de sensores.

En los siguientes capítulos se presenta el diseño mecánico del robot, el diseño electrónico y el software que permite el mapeo y navegación del prototipo desarrollado en este trabajo.

# Capítulo 2

# Diseño y Construcción del Prototipo

El desarrollo del sistema móvil empleado para realizar las pruebas de mapping, navegación y planeación, se focaliza en la implementación de un sistema embebido para realizar el procesamiento de datos, compuesto por una plataforma circular con una configuración de tipo diferencial. A continuación se realiza la descripción de cada uno de los aspectos que componen el hardware del robot.

## 2.1 Requerimiento de diseño

El diseño propuesto para la realización de este trabajo, tiene como principio el desarrollo de un sistema de bajo coste comparado con robots comerciales (Apéndice B); en donde se implementa una arquitectura de hardware - software abierta y con la posibilidad de ser replicable por la comunidad académica. La construcción del robot sobre el cual se realizan las pruebas de los diferentes algoritmos de control, navegación, mapping, entre otros, se convierte en una etapa fundamental para alcanzar los objetivos propuestos; sumado a ello, el robot construido se convierte en herramienta para futuros trabajos de investigación y desarrollo en la linea de robótica móvil.

En este sentido se plantean los requerimientos del robot, incluyendo los componentes más relevantes en el desarrollo de robótica móvil:

- Sistema de locomoción: Robot diferencial no holónomico.
- Instrumentación: Sensores internos y externos, que permitan determinar obstáculos así como el cálculo la posición del robot en un ambiente determinado.
- Sistema de procesamiento: Sistema embebido de arquitectura abierta.

A continuación se realiza la descripción de diseño del robot, sus características y correspondiente modelo matemático del sistema desarrollado. De igual manera, se realizará la descripción del diseño electrónico empleado.

## 2.2 Diseño del chasis y componentes mecánicos

Uno de los diseños más simples y económicos de implementar en robótica móvil, son los sistemas no holónomicos. Un robot móvil *holónomico* se puede mover de manera instantánea en cualquier

dirección; por el contrario un robot *no holónomico* para moverse en una dirección determinada, deberá ejecutar una serie de rutinas hasta alcanzar la dirección deseada (Goris, 2005). El robot *diferencial* se caracteriza por tener dos ruedas alineadas sobre un mismo eje y controladas a través de dos actuadores eléctricos diferentes. A su vez, se emplean una o dos ruedas sin motor cuya función es mantener el robot en equilibrio. El robot diseñado tiene como base una circunferencia dado que su control y modelo para realizar mapping es mas eficiente (Joseph, 2015). En la figura 2.1 se observa la plataforma diseñada.

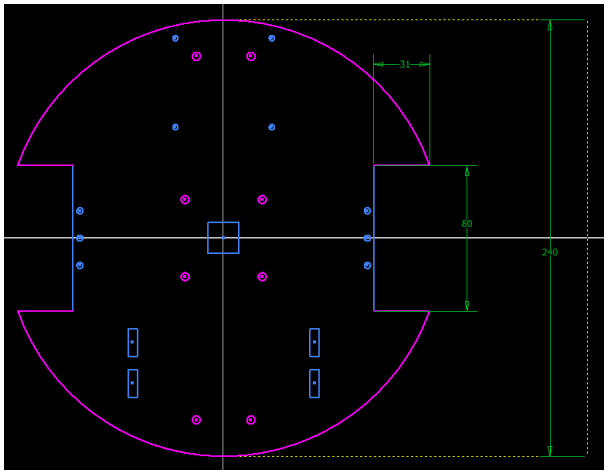
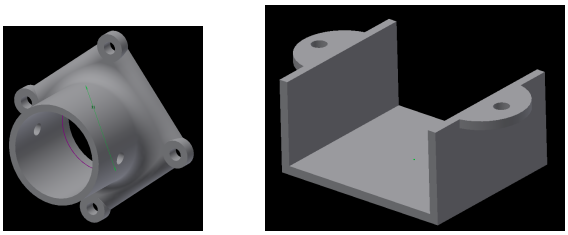



Figura 2.1: Diseño base del Robot

Esta plataforma se realiza en polimetilmetacrilato que es un material resistente y muy liviano. El diseño se desarrolló en Autocad® y posteriormente se procede a su corte en láser. El diámetro de la circunferencia se calculo acorde al tamaño de los motores que se ubican alineados sobre un mismo eje; por lo tanto el tamaño de la base es de 240 mm. Adicionalmente se incluyen un espacio para las llantas de 31 mm x 80 mm. La base contiene orificios para sujetar los motores, baterías y las ruedas de apoyo frontal y trasera.

Para la sujeción de las baterías y del sensor Kinect se diseñan los respectivos soportes, y posteriormente son impresos en 3D. La impresión 3D facilita la construcción de un diseño mecánico con precisión. En la figura 2.2(a) se observa el diseño para la estructura que soporta el Kinect.

Para soportar la baterías y permitir el fácil acceso para su recargar, se diseña un soporte que se sujeta en la parte inferior del robot, como se observa en la figura 2.2(b).



(a) Soporte Kinect (b) Soporte baterías

Figura 2.2: Diseño de soportes para impresión 3D.

Es importante destacar que antes de realizar el montaje real del robot, se realizaron diferentes simulaciones que concluyeron en un último modelo, como se observa en la figura 2.3

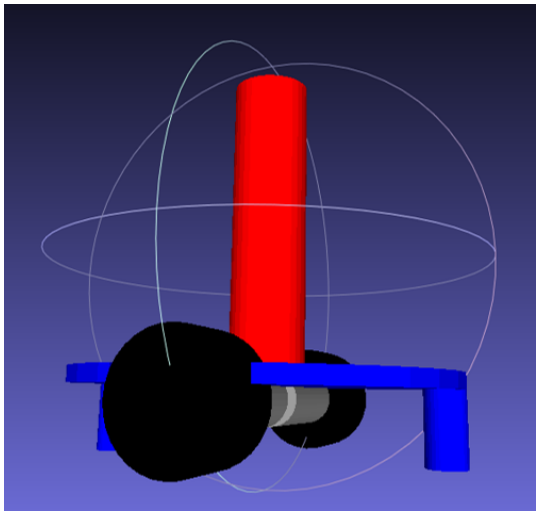


Figura 2.3: Modelo en Bleder del robot.

Todos los diseños en 3D fueron realizados en *Blender*, y las piezas fueron impresas en un termoplastico de tipo ABS. Finalmente, el robot construido se observa en la figura 2.4

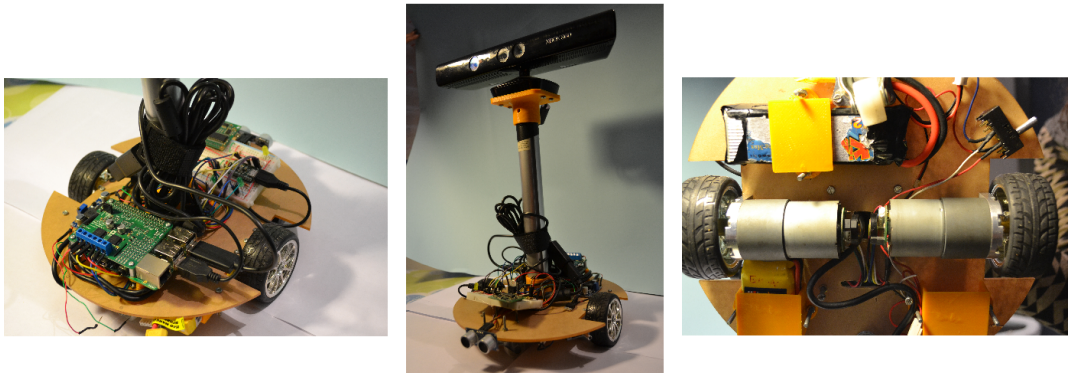


Figura 2.4: Prototipo del robot finalizado

## 2.3 Modelo matemático del robot

A continuación se realizará una representación de las ecuaciones cinemáticas que gobiernan el robot. Siguiendo el modelo de Hellström (2011), el diseño empleado para un robot diferencial se caracteriza por ubicar los dos actuadores sobre un mismo eje. En la figura 2.5 se observa la configuración para un robot diferencial con un centro instantáneo de rotación (CIR).

Para el calculo de las ecuaciones cinemáticas, tomamos como variables la velocidad angular $w$ del robot. Cada una de las ruedas del robot gira alrededor del CIR con una velocidad $V_{L,R} = 2\pi r_{(L,R)}/T$ donde $r_{(L,R)}$ es la distancia de cada rueda al CIR y $T$ es el tiempo que tarda en realizar una vuelta completa alrededor del CIR. La velocidad angular es definida como $w = \frac{2\pi}{T}$,

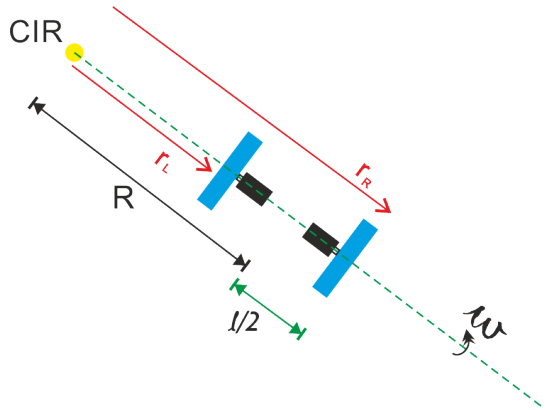



Figura 2.5: Configuración de un robot diferencial

por lo tanto podemos concluir que $V_{L,R} = wr_{(L,R)}$.

En la figura 2.6 se realiza una descripción mas detallada, con la que se facilita realizar un análisis mas completo.

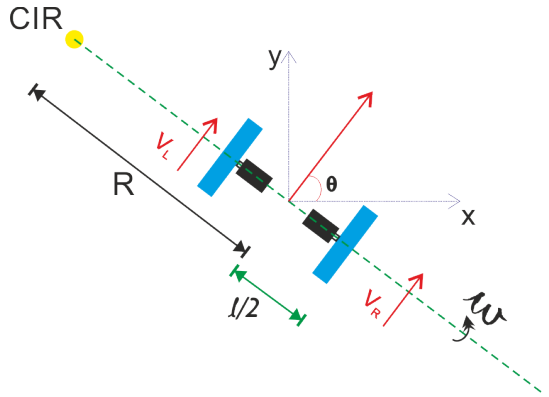



Figura 2.6: Modelo del Robot diferencial

Para este caso, tenemos que la velocidad en la rueda derecha es $V_R = w(R + l/2)$ y para la rueda izquierda será $V_L = w(R - l/2)$. Despejando para los valores de $R$ y $l$ tenemos:

$$R = \frac{l}{2}\frac{V_L + V_R}{V_L - V_R}$$

$$w = \frac{V_R - V_L}{l}$$

Ahora, si asumimos que el robot se desplaza con una velocidad angular $w$ durante un $\delta t$, la orientación del robot cambiará,de la forma:

$$\theta' = w\delta t + \theta$$

Donde el centro de rotación se describe en forma trigonométrica como:

$$CIR = [CIR_x, CIR_y] = [x - Rsen\theta, y + Rcos\theta]$$

Es posible calcular una nueva posición conociendo la ubicación inicial del robot y calculando la matriz de rotación en 2D. La rotación alrededor de un punto CIR con una velocidad de rotación $w$ para un $t + \delta t$ estará dada por:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} cos(w\delta t) & -sin(w\delta t) \\ sin(w\delta t) & cos(w\delta t) \end{pmatrix} \begin{pmatrix} x - CIR_x \\ y - CIR_y \end{pmatrix} + \begin{pmatrix} CIR_x \\ CIR_y \end{pmatrix} \tag{2.1}$$

En la matriz la velocidad angular es calculada indirectamente a través de $w = \frac{V_R - V_L}{l}$; para el cálculo de la velocidad en cada una de las ruedas, se realiza mediante la lectura del encoder asociado a cada motor.

Por lo tanto podemos definir que a velocidad para un tiempo $t$ a $t + \delta t$ se describe como:

$$V = \frac{N * distanciaTic}{\delta t} \tag{2.2}$$

En donde $N$ corresponde a la cantidad de pulsos leídos (En el intervalo $t + \delta t$); y $distanciaTic$ corresponde a la distancia que recorre el robot por cada pulso emitido.

Para el prototipo desarrollado en el encoder entrega 1610 pulsos por revolución; con una rueda de 63 mm de diámetro, el robot se desplaza aproximadamente 0.12 mm por pulso.

Reemplazando con las ecuaciones descritas previamente, tenemos:

$$R = \frac{l}{2} \frac{V_L + V_R}{V_L - V_R} = \frac{l}{2} \frac{N_L + N_R}{N_L - N_R}$$

$$w\delta t = \frac{(N_R - N_L)\delta t}{l} = \frac{distanciaTic * N_R - N_L}{l}$$

En donde $N_R$ y $NL$ son los pulsos emitidos por el encoder para cada una de las ruedas. Por lo tanto si l robot genera pulsos a través de los encoder indica que se ha realizado un movimiento y su posición final puede ser calculada a través de la ecuación 2.3:

$$\begin{pmatrix} x' \\ y' \\ \theta' \end{pmatrix} = \begin{pmatrix} cos(w\delta t) & -sin(w\delta t) & 0 \\ sin(w\delta t) & cos(w\delta t & 0 \\ 0 & 0 & 1) \end{pmatrix} \begin{pmatrix} x - CIR_x \\ y - CIR_y \\ \theta \end{pmatrix} + \begin{pmatrix} CIR_x \\ CIR_y \\ w\delta t \end{pmatrix} \tag{2.3}$$

## 2.4 Especificaciones actuadores, sensores y etapa de potencia.

La selección de los componentes eléctricos y electrónicos, están orientados al diseño de un robot que permita la navegación autónoma, la construcción de un mapa y la planificación de trayectorias. A continuación se realiza una descripción de los componentes empleados en el desarrollo del prototipo.

### 2.4.1 Motores, encoder y driver

Se emplea un motor con escobillas (*brushed*) que trabaja a una tensión nominal de 12v, este motor funciona apropiadamente para valores de tensión inferiores. Para el diseño de nuestro robot se

alimenta a 7.7v. Cada motor presenta un consumo de 100 mA sin carga y alcanza has 330mA con carga. Una de las características mas sobresalientes es el torque, que para este motor es de 1.76 $N \cdot m$. En la figura 2.7 se observa el motor empleado.

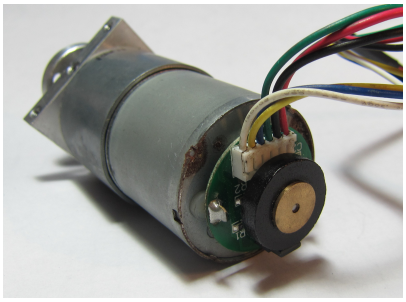


Figura 2.7: Motor 37D.

Por otro lado, el motor incluye un eje extendido en donde se implementa un encoder de dos canales que emplea dos sensores efecto *hall* con los que es posible determinar la velocidad de rotación y realizar el cálculo para identificar el sentido de giro. Para determinar el sentido de giro, basta con realizar la comparación entre las señales de cada canal del encoder, al estar desfasadas, el ejercicio se reduce a identificar cual es el primer canal en detectar un pulso. Esto se realiza mediante un disco como se muestra en la figura 2.8.

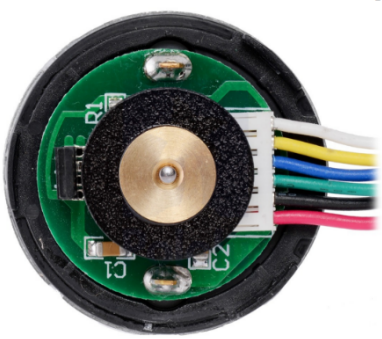


Figura 2.8: Enconder motores 37D mm.

El encoder tiene una resolución de 1610 pulsos por revolución sobre el eje que acopla la rueda, por tanto se considera apropiado para realizar la estimación odometríca.

Para el control de potencia y giro de los dos motores se emplea el driver $MC33926$ del fabricante Freescale e integrada por la empresa Pololu y es diseñado para acoplarse a la placa *Raspberry Pi*. Cada uno de los puente H permite una alimentación entre 5 y 28v y hasta 3A por canal en funcionamiento continuo. Soporta corrientes pico de hasta 5A. Es un driver ideal por la integración en una sola tarjeta, la capacidad de corriente nominal así como la corriente pico por canal. En la figura 2.9 se observa la conexión con la tarjeta de desarrollo. El driver trabaja con frecuencias de hasta 20KHz; también es posible realizar la alimentación de la *Raspberry pi* a través de un regulador externo. Esta tarjeta emplea tres lineas para cada uno de los canales: señal PWM y dos lineas de control para el giro.

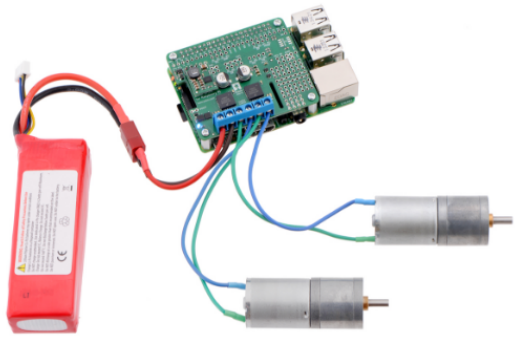


Figura 2.9: Conexión Motores y etapa de potencia

### 2.4.2 Sensor de ultrasonidos

El sensor de ultrasonidos se encarga de detectar objetos en un rango determinado; sin embargo, no es posible determinar con exactitud en que lugar se encuentra el objeto detectado. Si el objetivo fuera determinar la distancia a la que se encuentra con exactitud un obstáculo, se propondría el uso de un sensor óptico. Dado que para determinar un obstáculo y su distancia se emplea el Kinect, el sensor de ultrasonidos no tendría ninguna relevancia. Sin embargo, puesto que el rango lectura del Kinect, en ocasiones no permite detectar objetos o elementos que se encuentren a distancias muy cercanas, así como la alturas menores 5 cms, o existan casos en los que el Kinect pueda presentar fallos; se decide emplear un sistema auxiliar que evite cualquier choque accidental ocasionado por una falla en el procesamiento de los datos en el robot.

Así, este sensor enviará la información y será capaz de interrumpir cualquier proceso para detenerse de inmediato cuando considere que se encuentra cerca de una posible colisión con un objeto a su alrededor. La lectura de este sensor es realizada en un microcontrolador, que envía la información de forma serial y es publicada como un *Topic* (en la sección 3.1.2 se amplia esta información) en ROS.

### 2.4.3 Compás

Una de las tareas mas importantes para el robot es determinar su posición local relacionada a un mapa. Para ello, mediante el sentido y numero de rotaciones del robot es posible obtener una estimación del posición del robot. Sin embargo, este método no es de gran precisión por los errores acumulativos en el cálculo, así como errores causados por problemas mecánicos (Deslizamiento de las llantas en trayectorias rectas y en giros, superficies no uniformes entre otros). Con el objetivo de minimizar el error causado por el cambio de dirección del robot se plantea el uso de un brújula electromagnética que permita identificar la posición del robot.

Para ello se ha optado por el compás CMPS03, que emplea el protocolo de comunicación I2C para su comunicación con el microcontrolador. Esta lectura entrega un valor único correspondiente a la dirección que apunta el robot respecto al Norte magnético de la tierra. Presenta un consumo de corriente aproximadamente de 20mA. El módulo emplea un sensor de campo magnético Philips KMZ51 que es procesado por un microcontrolador de Microchip que entrega la respectiva señal I2C. En la figura 2.10 se observa la descripción del módulo.

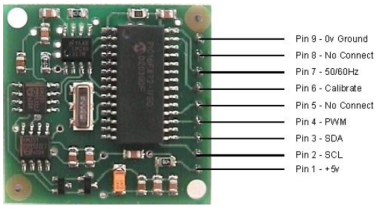



Figura 2.10: Compás CMPS03

### 2.4.4 Kinect como sensor LIDAR

El uso del Kinect en robótica móvil, ha tenido un alto impacto (Fankhauser et al., 2015). Esto es debido a su bajo coste, que permite acceder fácilmente a este dispositivo. El Kinect fue un desarrollo creado por Microsoft para la consola de video juegos Xbox 360. Sus aplicaciones se extendieron rápidamente y la compañía empezó a distribuir su versión Kinect para PC, con el que se incluye un entorno de desarrollo así como los drivers para facilitar su utilización. En el desarrollo de esté proyecto se emplea una *Kinect For Xbox 360* versión 1.0. Este dispositivo esta compuesto por:

1. Cámara de profundidad con resolución de 640x480 píxeles con una apertura de 58.5x46.6 grados en horizontal y vertical respectivamente. El rango de la cámara de profundidad esta entre los 0.30 m y hasta 8 m. Velocidad 30fps. (Se construye con un emisor y receptor de infrarrojos.)

2. Cámara RGB de 640x480 píxeles, con una apertura de 58x46.6 grados en horizontal y vertical respectivamente, con una velocidad de captura de 30fps

3. Cuatro micrófonos con una frecuencia de muestreo 16KHz.

4. Un servomotor y acelerómetro.

Un sensor LIDAR (Ligh Detection and Ranging) es normalmente empleado en robots de alta precisión, debido a su alta resolución, aunque es uno de los sensores mas apropiados para realizar un mapeo en un robot móvil, su coste es elevado; por ejemplo, los sensores empleados en los robots comerciales son fabricados por *Hokuyo*, y su valor oscila entre los USD $1000 y los USD $5000. Por lo tanto, una de las estrategias más empleadas por investigadores en robótica móvil es emplear dispositivos como el Kinect (con un coste alrededor de USD $150), del cual es posible obtener señales de profundidad similares a las que entregaría un sensor LIDAR. Aunque su resolución en menor, son muy empleados en robótica móvil.

#### Sistemas embebidos

Una de las metas propuestas en el desarrollo de este trabajo es emplear un sistema embebido de bajo coste para realizar el procesamiento relacionado con el mapeo, planeación y navegación. Es común encontrar robots comerciales como TurttleBot en los cuales el procesamiento se realiza sobre un PC dedicado y diseñado exclusivamente para estos robots. También encontramos desarrollos en donde se emplea un computador portátil, incrementando el peso del robot, como se describe en Gautham Ponnu (2016), Goebel (2014), Cousins (2012).

Actualmente los sistemas embebidos de bajo coste han incrementado notoriamente. Encontramos tarjetas como la Galileo de Intel, La BeagleBone, Ordroid-C2 o la Raspberry Pi entre muchas otras, todas ellas con procesadores entre 700MHz y 2GHz, memoria RAM de hasta 2GB con puertos de comunicación (Bluetooth, Wifi, Eternet), Conexión VGA y HDMI para video, salida de audio, entre otras características que varían en las diferentes versiones.

El sistema empleado en este desarrollo es una tarjeta *Raspberry Pi 3*. A continuación se describen las principales características de esta tarjeta:

- Procesador a 1.2GHz, 64 bits de cuatro núcleos ARMv8.
- Wireless LAN 802.11n
- Bluetooth 4.1
- Bluetooth Low Energy
- 1 GB de memoria RAM
- Cuatro puertos USB
- 40 pines de propósito general
- Puerto HDMI
- Expansión para memoria microSD

Sobre esta tarjeta es instalado el sistema operativo *Ubuntu Mate Xenial* y ROS (Robot Operating System); con la tarjeta se realiza el control del hardware (Control de Motores y lectura del Kinect, así como la comunicación inalámbrica), para la lectura de los sensores se emplea como estrategia incluir un microcontrolador.

Puesto que la Raspberry Pi no cuenta con convertidores análogo/digital, el uso del microcontrolador que se encargue de esta tarea facilita la implementación. Para ello se ha empleado un microcontrolador ATmega32U4, ensamblado por Pololu en una board denominada A-Star 32U4, y que provee una interfaz USB con 18 pines de entrada salida, de los cuales 8 son análogos, un reloj de 16 MHz y necesita una alimentación de 5v.Figura 2.11

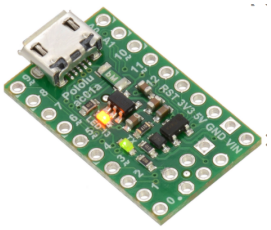


Figura 2.11: Microcontrolador A-Star 32U4 (ATmega32U4 32U4)

Este microcontrolador se encarga de realizar la lectura I2C del Compás, procesar la información y enviar la orientación en términos de cuaterniones. La descripción del programa se realiza en la sección 3.3. También se codifica la lectura y procesamiento del sensor de ultrasonidos, para entregar la información de distancia en centímetros; Adicionalmente se realiza la monitorización del nivel de tensión en las baterías. Toda la información procesada es enviada mediante comunicación serial a la Raspberry Pi, eliminando carga computacional al procesador principal.

### 2.4.5 Etapa de Potencia

La alimentación de los diferentes módulos de instrumentación y control juega un papel esencial en el tiempo de autonomía del robot, pues dependiendo de ella podremos realizar la experimentación asociada a los algoritmos e interacción del robot en el medio ambiente. A continuación se describen los dispositivos y estrategias empleados para alimentar los diferentes componentes del robot:

1. Baterías: un sistema de almacenamiento recargable y que permita un buen suministro de corriente a cada uno de los componentes según su requerimiento. Para ello se propone el uso de baterías Lipo 12 y 7.4 voltios, con una capacidad de corriente de 1000 mAh. Son empleadas tres baterías: Dos de 12v conectadas en paralelo para alimentar el sensor Kinect y una para suministrar energía a los motores y el sistema electrónico.

2. Convertidor DC-DC: se emplea un elevador tipo Boost con el fin de asegurar 12.2v en la entrada del sensor Kinect. Es normal que durante el funcionamiento en las baterías el voltaje empiece a descender, por lo tanto sin importar que ello pase, el elevador mantendrá la salida constante en la entrada del sensor. Por lo tanto se emplea el modulo XL6009, que permite niveles de entrada entre 3-32v y un voltaje de salida de 5-32v. Tienen una capacidad de corriente hasta de 4A, con una eficiencia de 90% según datos del fabricante.

3. Regulador de 5v: La batería encargada de suministrar energía a la Raspberry Pi y los circuitos de instrumentación entrega un voltaje entre 7.2v y 8.0 dependiendo la carga de la batería; por lo tanto se emplea un regulador *Step-Down*, que reduce eficientemente a 5v con una salida de corriente constante hasta 2.5A.

Por último, en la figura 2.12 se observa la conexión correspondiente a cada uno de los sistemas eléctricos y electrónicos que componen el robot.

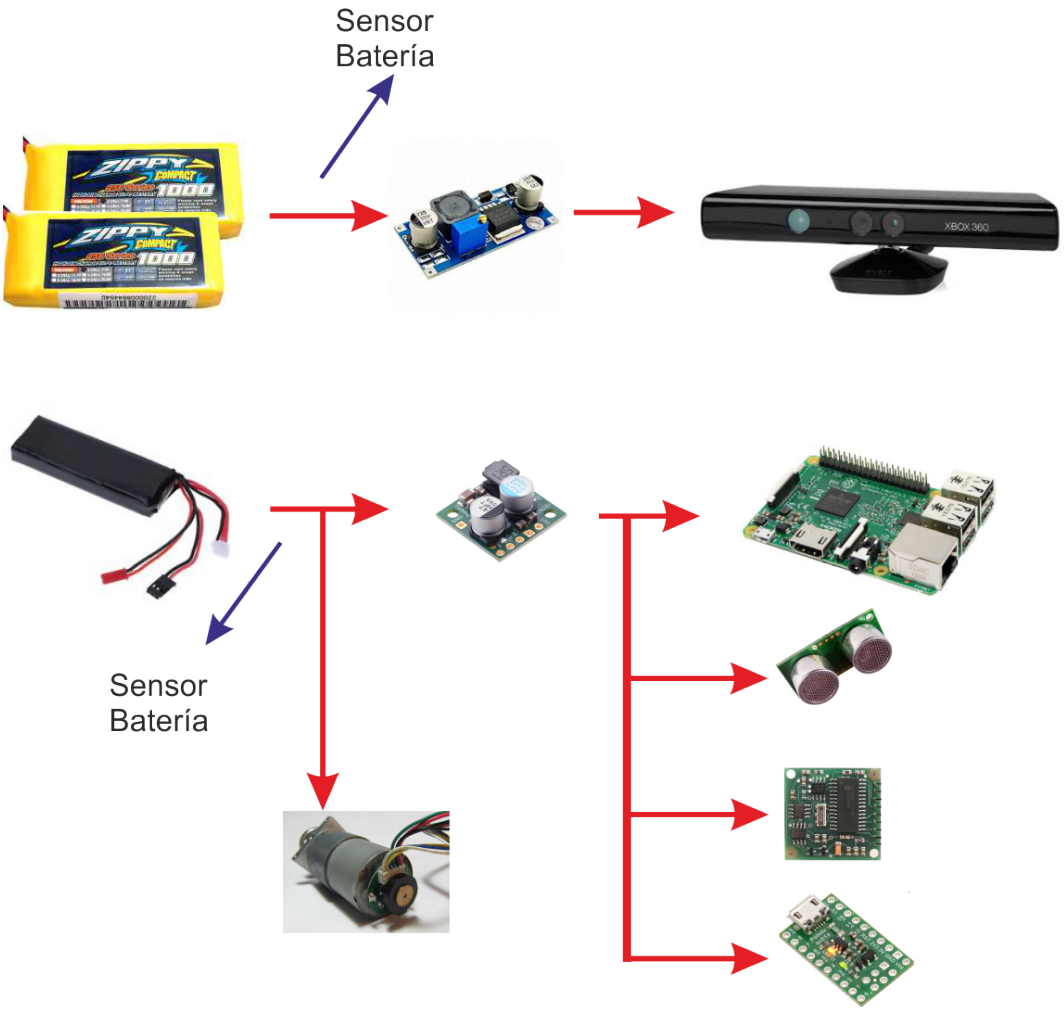



Figura 2.12: Sistema de alimentación a los diferentes módulos.

# Capítulo 3

# Diseño del Software

El segundo eje central de este trabajo, es el desarrollo de los algoritmos que se encargan de realizar el control, mapeo y planificación de trayectorias en un entorno. Por lo tanto se realiza la descripción del framework empleado, así como los algoritmos orientados al control del hardware y las estrategias de navegación.

## 3.1 Arquitectura de ROS

**Robot Operating System** (ROS) es un framework desarrollado en 2007 por **Stanford Artificial Intelligence Laboratory**. Desde el 2008, Willow Garage ha continuado su desarrollo y recientemente se ha creado *Open Source Robotics Foundation* que se encarga del sostenimiento de ROS. Este framework es usado ampliamente por la comunidad académica que realiza investigación en las diferentes lineas de robótica. Varias compañias han empezado a utilizar ROS en sus productos; por ejemplo robots como Baxter de la empresa *Rethink Rrobotics*, PR2 de la compañia *Willow Garage* o Husky producido por *Clearpath Robotics*, entre otros. En el sitio *www.robots.ros.org*, se aprecian varios ejemplos de robots empleados tanto en educación como en la industria, utilizando ROS como base para el desarrollo de sus algoritmos (Cousins, 2012).

El uso de ROS está bajo los términos de licencia (BSD) y es de tipo *open source* y puede ser empleado de forma comercial y educativa. ROS permite el diseño de una arquitectura centralizada y distribuida, donde el procesamiento toma lugar en cada uno de los nodos, permitiendo recibir y enviar información para comunicarse con otros nodos. Un nodo es cualquier proceso que permita leer un sensor, controlar un actuador o ejecutar algoritmos para el cálculo de odometría, navegación, entre otros.

Una de las características es la independencia de lenguajes. Es posible desarrollar scripts empleando los lenguajes de programación más comunes como C, C++, Lisp, Python entre otros. A continuación se realiza una síntesis de los conceptos mas relevantes de ROS.

### 3.1.1 Sistema de Archivos en ROS

La organización de los archivos empleada por ROS, permite compartir, modificar, desarrollar e integrar funciones.

- **Packages:** Son la principal unidad del framework. Los paquetes contienen archivos ejecutables, librerias, archivos de descripción entre otros. La idea principal, es que estos paquetes puedan ser re-utilizables.
- **Packages Manifest:** Es un solo archivo que se encuentra en cada paquete y contiene todos los detalles de un paquete, incluyendo licencias, descripciones y dependencias.
- **Mensajes:** Son estructuras de datos que se envían a través de ROS. Estas son guardadas en una carpeta llamada *msg*.

### 3.1.2 Computación gráfica

En ROS todos los procesos, mensajes, programas, servicios, entre otros, se encuentran contados mediante un red. Cada uno de los elementos del sistema pueden acceder a la red e interactuar entre ellos. Los conceptos básicos de *Computation Graph* son:

- **Nodes:** Son los procesos que se realizan. Por ejemplo: el calculo de la odometria, el control de los motores, la lectura del sensor, etc.
- **ROS Master:** Es el elemento principal que se encarga de realizar la comunicación entre nodos. Permite localizar todos los nodos entre si.
- **Mensajes:** Es una estructura que describe las características de los datos que envían y reciben los nodos. Por ejemplo, datos de tipo flotante, enteros, estructuras etc.
- **Topics:** Es el nombre usado para identificar el contenido de cada uno de los mensajes enviados y recibidos por los nodos. Un nodo se puede publicar un Topic o suscribirse a un Topic. Un Topic puede ser leído por cualquier nodo sin ninguna restricción, es decir, esta disponible para todos los nodos.
- **Servicios:** Son procesos sincrónicos entre nodos. A diferencia de un Topic, un servicio emplea una arquitectura cliente/servidor, es decir, a un requerimiento una respuesta.
- **Bags:** Son un mecanismo para guardar los datos obtenidos en pruebas (simulación o en tiempo real) con los que posteriormente podremos realizar análisis.

## 3.2 Package descripción del Robot

Una de las herramientas más importantes al momento de diseñar sistemas robóticos, son los simuladores. En ellos podemos realizar pruebas emulando sensores como ultrasonidos, cámaras 3D, encoder, infrarojos, así como diferentes tipos de actuadores.

A través ROS es posible realizar el modelo y simulación de un robot. Para ello se emplean diferentes lenguajes de marcaje orientados a la descripción del robot a simular. Uno de los principales formatos para la descripción del robot es URDF (Unified Robot Description Format), que es un archivo de etiquetas tipo XML orientado a describir todas las partes del robot; sus uniones, dimensiones, características mecánicas, entre otras. A continuación se realizará la descripción del robot empleado en este trabajo.

Para la creación del modelo se realiza un *package* el cual ROS posteriormente identifica y con le que podremos realizar las simulaciones. El objetivo final del modelo será simularlo en GAZEBO donde el robot podrá interactuar con el entorno, verificar el funcionamiento de los sensores, actuadores y probar los algoritmos diseñados; por tanto el modelo deberá permitir la lectura de sus sensores internos y externos desde rviz.

Para la creación del modelo, se crea un package denominado *ibot_description*; en él se crea una carpeta donde se almacenarán todos los archivos URDF asociados al robot. Inicialmente creamos un archivo base (*ibot_base.urdf*) en donde se encuentra la estructura del robot. A continuación se describe parte del código empleado:

```
<link name="base_link">
<visual>
<origin xyz= "0 0 0.01" rpy="0 0 0" />
<geometry>
<box size="0.24 0.16 0.03"/>     <!-- largo ancho alto -->.
</geometry>
<material name="yellow">
<color rgba="0 0.3 1 1"/>
</material>
</visual>
<!-- Base para colision -->
<collision>
<origin xyz="0 0 0.01" rpy="0 0 0" />
<geometry>
<box size="0.24 0.16 0.03"/>
</geometry>
</collision>
<inertial>
<mass value="2"/>        <!-- calculo aprox de la inercia del robot-->
<inertia ixx="0.019995" ixy="0.0" ixz="0.0"
iyy="0.019995" iyz="0.0"
izz="0.03675" />
</inertial>
```

En las etiquetas anteriores se describe la base del robot; tamaño de visualización así como el área que se describirá para la colisión que corresponde con el mismo tamaño del robot. Cada una de las partes del robot se describen de forma independiente, y se relacionan entre si a través de uniones. Por ejemplo, la etiqueta *joint* nos permite realizar las diferentes uniones entre elementos del robot. En el sito oficial de ROS, `www.ros.org/urdf` se encuentra la documentación detallada del formato URDF para la descripción de robots. A continuación se muestra a modo de ejemplo el script para una rueda.

```
<joint name="joint_left_wheel" type="continuous">
<parent link="base_link"/>
<child link="left_wheel"/>
<origin xyz="0 0.1 0" rpy="0 0 0" />
```

```
<axis xyz="0 1 0" />
</joint>
```

De manera similar se realiza la descripción para los sensores, como por ejemplo el Kinect, el sensor ultrasonidos, el compás y los encoder. Por ejemplo, para la cámara de profundidad se incluye como se muestra en la siguiente sección del archivo URDF:

```
<joint name="camera_depth_joint" type="fixed">
 <origin xyz="0 ${2 * -cam_py} 0" rpy="0 0 0" />
 <parent link="camera_rgb_frame" />
 <child link="camera_depth_frame" />
 </joint>
 <link name="camera_depth_frame">
 <inertial>
 <mass value="0.001" />
 <origin xyz="0 0 0" />
 <inertia ixx="0.0001" ixy="0.0" ixz="0.0"
 iyy="0.0001" iyz="0.0"
 izz="0.0001" />
 </inertial>
 </link>
 <joint name="camera_depth_optical_joint" type="fixed">
 <origin xyz="0 0 0" rpy="${-M_PI/2} 0 ${-M_PI/2}" />
 <parent link="camera_depth_frame" />
 <child link="camera_depth_optical_frame" />
 </joint>
 <link name="camera_depth_optical_frame">
 <inertial>
 <mass value="0.001" />
 <origin xyz="0 0 0" />
 <inertia ixx="0.0001" ixy="0.0" ixz="0.0"
 iyy="0.0001" iyz="0.0"
 izz="0.0001" />
 </inertial>
 </link>

 <! -- Esta librería nos permite cargar el modelo del Kinect -->
 <!-- Kinect sensor for simulation -->
 <turtlebot_sim_3dsensor/>
 </robot>
```

Una vez realizado el diseño, se procede a realizar la comprobación de sintaxis y relación entre los nodos creados. Para ello, ROS cuenta con un framework que permite cargar el archivo a evaluar de la siguiente manera: *urdf_to_graphiz ibot_base.urdf*. Cuando el archivo .urdf no contiene ningún error se crea un documento que muestra la relación entre los nodos creados. En la figura 3.1 se observa la relación entre los nodos y las ruedas creadas en el robot.

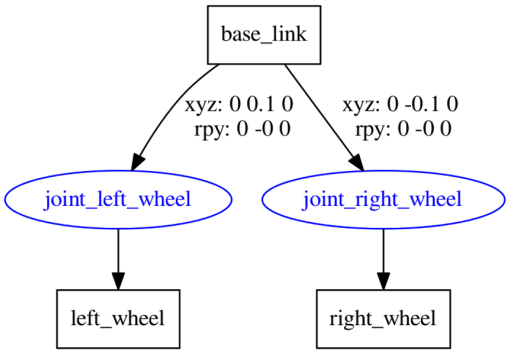



Figura 3.1: Interconexión de nodos para el modelo del Robot

El nodo principal es llamado base_link, que corresponde al eje de referencia para todas nuestras simulaciones. En la figura 3.1 se observan la posición $xyz$ y la orientación $rpy$. En azul, aparecen los elementos de unión, los cuales son necesarios para establecer relación entre elementos del robot; para este caso en particular se realiza la unión entre las ruedas y la base del robot.

Una vez finalizada la descripción del robot, existen diferentes simuladores que permiten visualizar y evaluar su comportamiento. Para el desarrollo de este trabajo se empleo rviz y gazebo, de los cuales se realizará la descripción mas adelante. En la figura 3.2 se observa la apariencia del robot en el simulador gazebo.

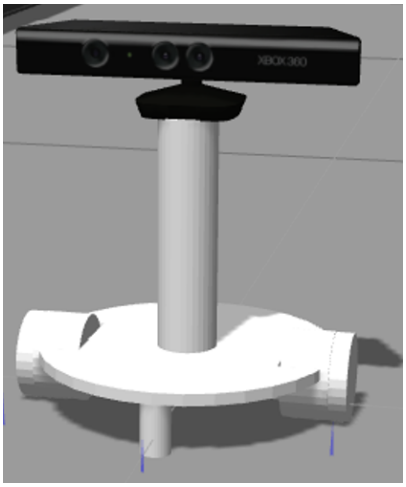


Figura 3.2: Representación del modelo creado en el archivo .urdf

## 3.3 Diseño firmware para la comunicación con el hardware

Una de las ventajas mas relevantes de emplear ROS, como el sistema operativo sobre el que se desarrolla el prototipo que se plantea en este documento, es la comunicación mediante Topics, Mensajes y Servicios; pues la información puede ser procesada por diferentes paquetes facilitando la interacción entre algoritmos. Por lo tanto, la comunicación con el hardware se realiza manteniendo la misma estructura. Como se mencionó en la sección 2.4.4, se emplea un microcontrolador

orientado a procesar la información relacionada con los sensores del robot.

Por otro lado, para el control de los actuadores, se crea un nodo que envía directamente los datos sobre los puertos de la *Raspberry Pi*. A continuación se realiza un síntesis de los algoritmos para los sensores y actuadores.

### Control del driver para los motores

Para realizar el control de los motores empleando el *shield* dedicado a la etapa de potencia, se realiza un script en *python* para acceder a los puertos GPIO's de la Raspberry Pi, cuando es requerido por ROS. En el script se parametrizan los pines de entrada y salida, así como las variables para la generación del PWM. Este archivo es un *nodo* encargado de leer el Topic asociado al control de velocidad y generar la salida en los GPIO: el sentido de giro y la potencia correspondiente.

Para ello fue necesario realizar en un script la configuración de los puertos de la Raspberry Pi; posteriormente se asignan los valores máximo y mínimo de velocidad. A continuación se muestran algunos fragmentos del código:

```
def io_init():
  wiringpi.wiringPiSetupGpio()
  wiringpi.pinMode(12, wiringpi.GPIO.PWM_OUTPUT)
  wiringpi.pinMode(13, wiringpi.GPIO.PWM_OUTPUT)

  wiringpi.pwmSetMode(wiringpi.GPIO.PWM_MODE_MS)
  wiringpi.pwmSetRange(MAX_SPEED)
  wiringpi.pwmSetClock(2)

  wiringpi.pinMode(22, wiringpi.GPIO.OUTPUT)
  wiringpi.pinMode(23, wiringpi.GPIO.OUTPUT)
  wiringpi.pinMode(24, wiringpi.GPIO.OUTPUT)
  wiringpi.pinMode(25, wiringpi.GPIO.OUTPUT)

class Motor(object):
    MAX_SPEED = _max_speed
    def __init__(self, pwm_pin, dir_pin, en_pin):
        self.pwm_pin = pwm_pin
        self.dir_pin = dir_pin
        self.en_pin = en_pin

    def enable(self):
        io_init()
        wiringpi.digitalWrite(self.en_pin, 1)

    def disable(self):
        io_init()
        wiringpi.digitalWrite(self.en_pin, 0)
```

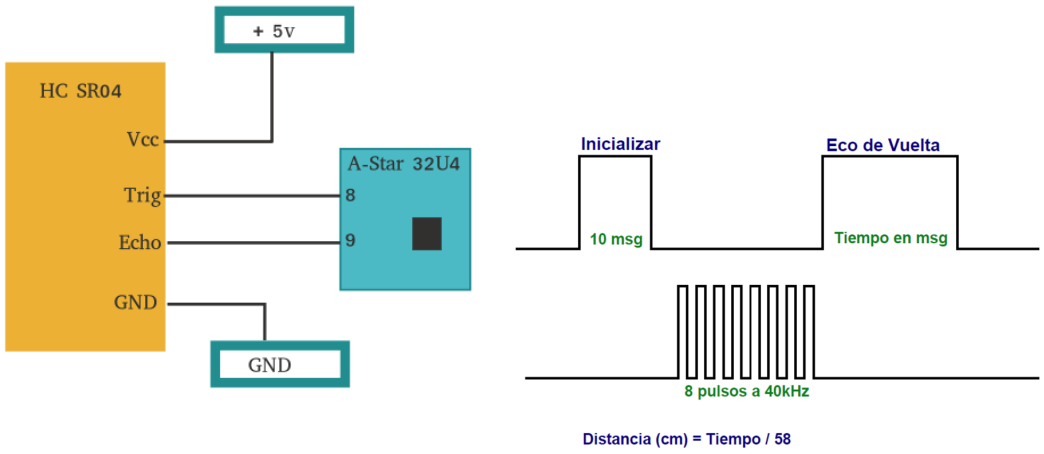
```
    def setSpeed(self, speed):
        if speed < 0:
            speed = -speed
            dir_value = 1
        else:
            dir_value = 0

        if speed > MAX_SPEED:
            speed = MAX_SPEED

        io_init()
        wiringpi.digitalWrite(self.dir_pin, dir_value)
        wiringpi.pwmWrite(self.pwm_pin, speed)
```

### Lectura del sensor ultrasonidos

Para la lectura del sensor ultrasonidos se requiere un canal encargado de emitir un pulso y otro canal que se encargará de la lectura del tiempo de vuelo de la señal. En la figura 3.3 se observan los pines de conexión con el microcontrolador. La emisión del pulso es generada durante $10\mu S$ posteriormente se espera la lectura de la señal reflejada. La distancia medida es proporcional al tiempo de vuelo de la señal emitida.




(a) Conexión del sensor

(b) Diagrama de tiempos activación del ultrasonidos

Figura 3.3: Sensor ultrasonido

Una vez, se ha realizado la lectura, basta con realizar la conversión de micro-segundos a centímetros.

### Lectura de los encoder

Cada uno de los encoder asociados a cada motor entrega dos señales: Canal A y Canal B. Para la lectura se establece una interrupción en el microcontrolador para el Canal A. Cada vez que

el programa ingresa a la interrupción se realiza la lectura del Canal B para determinar si se ha emitido un pulso. De ser así, se indicaría que el motor gira en un sentido (aumentamos el acumulador *Encoder_derecha_Ticks*) de lo contrario indicará el giro en contrasentido (disminuimos el acumulador *Encoder_derecha_Ticks*). La figura 3.4 corresponde a la lectura de los pulsos generados por el canal A y B del encoder vistos en el osciloscopio. En azul se representa el canal B y en amarillo el canal A.

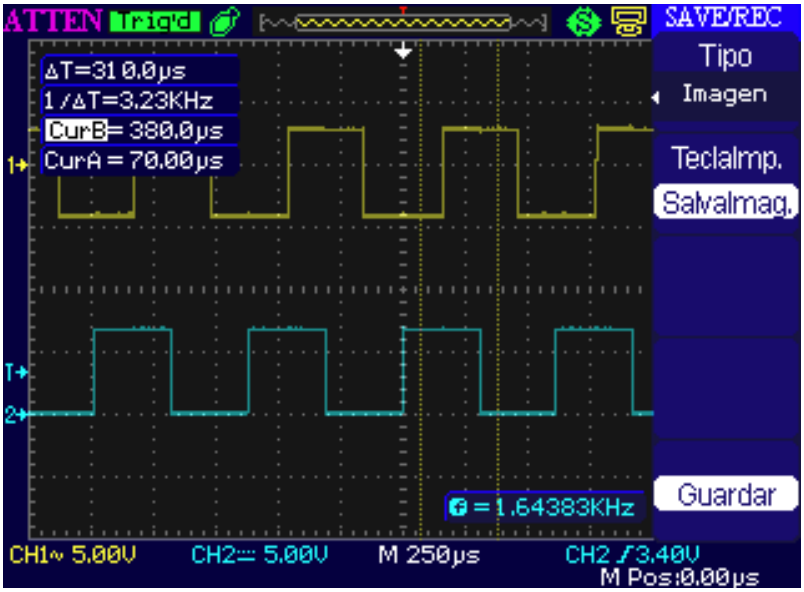



Figura 3.4: Visualización del encoder en el osciloscopio

A continuación se muestra el código empleado para determinar la dirección del robot.

```
//Interrupción Canal A
void Leer_Encoder_Der(){
   if (digitalRead(Encoder_CanalA)) {
    Encoder_derecha_Ticks++;
  }else{
    Encoder_derecha_Ticks--;
  }
}
```

### Lectura de la batería

La lectura de las baterías se realiza a través de un divisor de tensión con dos resistencias de $1M\Omega$; por lo tanto el valor máximo que leerá el sensor corresponde a $\frac{V_{cc}}{2}$. Para ello se utilizan dos de los convertidores análogo/digital del microcontrolador (Uno por cada batería). Se realiza la parametrización para calcular el votaje en la batería y es enviado a ROS. Adicionalmente, si la carga es inferior al limite de la batería se activa un indicador luminoso para alertar sobre el estado de la carga en las baterías.

**Lectura y procesamiento del compás**

El compás utilizado (CMP03) se comunica mediante el protocolo I2C, a través del cual envía un dato correspondiente a la ubicación del sensor respecto al polo norte magnético de la tierra, en el eje *yaw*. Este dato es procesado para calcular el valor del angulo *yaw*, y posteriormente realizar la representación en términos de cuaterniones (*qw qx qy qz*) los cuales son empleados para determinar la dirección que tiene el robot. A continuación se describe un apartado del código empleado para desarrollar esta función:

```
 float yaw = (medida*(3.1416))/180;
//La variable "medida" corresponde al dato entregado por el compás//
   float roll = 0;
   float pitch = 0;
   t0 = cos(yaw * 0.5);
   t1 = sin(yaw * 0.5);
   t2 = cos(roll * 0.5);
   t3 = sin(roll * 0.5);
   t4 = cos(pitch * 0.5);
   t5 = sin(pitch * 0.5);

   qw = t0 * t2 * t4 + t1 * t3 * t5;  //w
   qx = t0 * t3 * t4 - t1 * t2 * t5;  //x
   qy = t0 * t2 * t5 + t1 * t3 * t4;  //y
   qz = t1 * t2 * t4 - t0 * t3 * t5;  //z
```

## 3.4 Cálculo de la odometría

Algunos aspectos de la odometría fueron tratados en la sección 2.3 en donde se describen los cálculos necesarios para determinar la posición en un tiempo $(t + \delta t)$ conocida una posición inicial. El cálculo de la odometría es fundamental para la construcción del mapa, así como para la navegación en él.

Se realiza un *nodo* que se subscribe a los Topic *lwheel* y *rwheel* que corresponde a los datos entregados por el encoder. En la figura 3.5, se observan los elementos mas relevantes de este nodo.

## 3.5 Lectura del Kinect

ROS presenta una gran cantidad de alternativas y soporte para la implementación de cámaras con el objeto de realizar procesamiento de imágenes. OpenCV permite una integración con ROS facilitando el procesamiento visión por computador. Para el desarrollo de este proyecto se ha empleado un sensor Kinect; tal como se mencionó en la sección 2.4.4, el sistema está integrado por varios sensores.

Para poder usar el Kinect desde un dispositivo (PC, Raspberry, etc) es necesario cargar el driver que nos permita realizar la lectura de correspondiente a los sensores del Kinect.

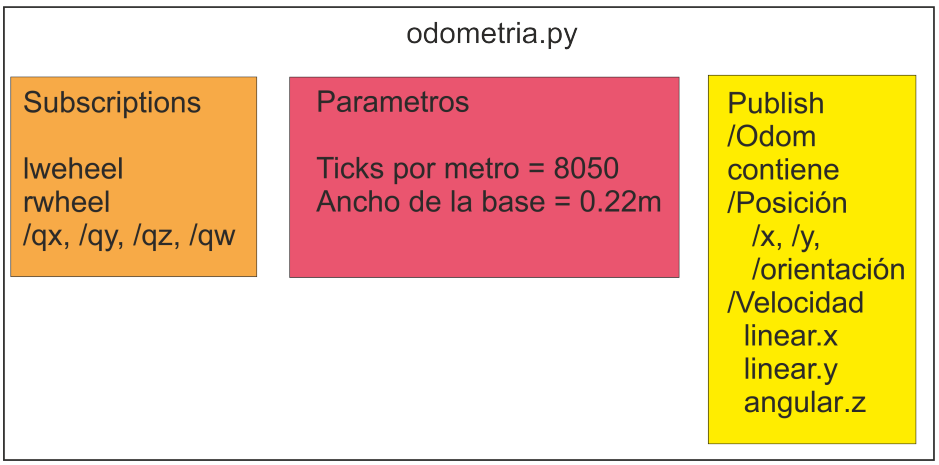



Figura 3.5: Elementos relevantes del nodo Odometría

OpenNI es un framework que define API's para escribir aplicaciones usando **Natural Interaction** (NI), la cual permite la interacción de forma natural a través de gestos, expresiones, movimiento, entre otros. Por tanto, se convierte en una herramienta fundamental para el desarrollo de aplicaciones que están relacionadas con la captura y procesamiento de datos provenientes del Kinect.

Sin embargo, el driver esta limitado a determinadas plataformas y sistemas de desarrollo. En particular, para el desarrollo de este proyecto el driver no funciona apropiadamente en la *Rasberry Pi*. En la revisión de la literatura, se encontró el framework *freenect* que a diferencia de OpenNI, no integra las características de pre-procesamiento de las imágenes; podría mencionarse que su funcionamiento es como un driver orientado a la lectura de los sensores del Kinect.

De acuerdo con Beno et al. (2014), Freenect es compatible con el sistema operativo y la *Raspberry Pi*, por lo tanto es el empleado en desarrollo de este proyecto.

Para realizar la lectura desde ROS, se crea un archivo *sensor.launch* desde el que se realiza la configuración para la lectura de los datos del Kinect. A continuación se realiza una síntesis de script:

```
<launch>
  <arg name="camera"          default="camera"/>
  <arg name="publish_tf"      default="false"/>
  <arg name="depth_registration"                  default="true"/>
  <arg if="$(arg depth_registration)"
  name="depth" value="depth_registered" />
  <arg unless="$(arg depth_registration)" name="depth" value="depth" />

  <!-- Processing Modules -->
  <arg name="rgb_processing"                      default="false"/>
  <arg name="ir_processing"                       default="true"/>
  <arg name="depth_processing"                    default="false"/>
  <arg name="depth_registered_processing"         default="true"/>
```

```
  <arg name="disparity_processing"                default="true"/>
  <arg name="disparity_registered_processing" default="true"/>
  <arg name="scan_processing"                     default="true"/>


  <!-- Laser topic -->
  <arg name="scan_topic" default="scan"/>

  <include file="$(find freenect_launch)/launch/freenect.launch">
    <arg name="camera"                value="$(arg camera)"/>
    <arg name="publish_tf"            value="$(arg publish_tf)"/>
    <arg name="depth_registration"
      value="$(arg depth_registration)"/>
    <arg name="num_worker_threads"
      value="$(arg num_worker_threads)" />

    <!-- Parametros para procesar -->
    <arg name="rgb_processing"     value="$(arg rgb_processing)"/>
    <arg name="ir_processing"      value="$(arg ir_processing)"/>
    <arg name="depth_processing"
      value="$(arg depth_processing)"/>
    <arg name="depth_registered_processing"
     value="$(arg depth_registered_processing)"/>
    <arg name="disparity_processing"
     value="$(arg disparity_processing)"/>
    <arg name="disparity_registered_processing"
    value="$(arg disparity_registered_processing)"/>
  </include>
</launch>
```

Como se observa el código, se incluye el archivo freenect.launch que contiene los parámetros necesarios para realizar la respectiva lectura. Una vez se ejecuta el archivo .launch, es posible revisar los Topic producidos obteniendo alrededor de 100 mensajes. En la figura 3.6 se observan algunos de los Topic.

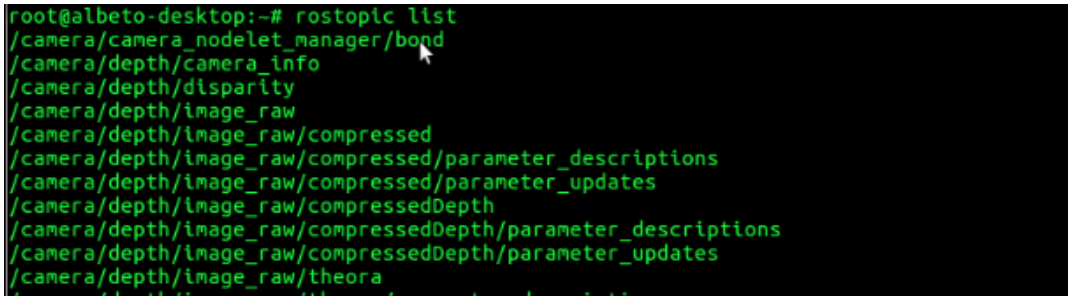



Figura 3.6: Listado de Topics generados por el Kinect empleando Freenect.

Por ejemplo el Topic relacionado con la cámara de profundidad */camera/depth/image_raw*

entrega una imagen de profundidad. A partir de allí es posible iniciar el procesamiento de imagen de acuerdo a las necesidades del diseño.

Finalizada la lectura de los correspondientes sensores, y con el objeto de adecuar nuestro sensor para que permita realizar la lectura de distancia a la que se encuentran los objetos al nivel del sensor, se realiza la conversión de imagen de profundidad a una señal de tipo sensor láser *virtual*, que nos permitirá realizar el mapa del entorno con el robot.

Para ello se emplea el nodo *depthimage_to_laserscan* que se encuentra en los repositorios de ROS. El nodo toma una imagen de profundidad y genera una mensaje correspondiente a un sensor láser en 2D, en la figura 4.2 se observa un ejemplo de su resultado. El nodo se suscribe a los topics: */camera/depth/image_raw*, */camera_info*; y publica el Topic */Scan*. Para realizar la configuración se agrega al archivo correspondiente a *sensor.lauch*:

```
<node pkg="nodelet" type="nodelet" name="depthimage_to_laserscan"
args= "load depthimage_to_laserscan/DepthImageToLaserScanNodelet
$(arg camera)/$(arg camera)_nodelet_manager">

    <param name="scan_height" value="15"/>
    <param name="output_frame_id" value="/$(arg camera)_depth_frame"/>
    <param name="range_min" value="0.45"/>

    <remap from="image" to="$(arg camera)/$(arg depth)/image_raw"/>
    <remap from="scan" to="$(arg scan_topic)"/>

    <remap from="$(arg camera)/image"
          to="$(arg camera)/$(arg depth)/image_raw"/>
    <remap from="$(arg camera)/scan" to="$(arg scan_topic)"/>
</node>
```

Estos parámetros permiten la configuración de acuerdo a los requerimiento del diseño. Por ejemplo,

```
<param name="scan_height" value="15"/>
```

permite la configuración de la altura a la que se realizará la conversión a una salida tipo láser; para este ejemplo serán 15 píxeles. Todos los parámetros de configuración del nodo se encuentran en `http://wiki.ros.org/depthimage_to_laserscan`

Utilizando la herramienta *rqt_graph* empleada para analizar gráficamente la relación entre los diferentes nodos, se evidencia la interconexión entre el nodo de la cámara y *depthimage_to_laserscan*, los cuales son representados en la figura 3.7.

## 3.6 Construcción del Mapa

Una de los retos para los investigadores en SLAM es el desarrollo de algoritmos que requieran un mínimo esfuerzo computacional, un consumo de memoria razonable, con el objeto de hacer robots de menor tamaño y por supuesto de menor coste (Abdelrasoul et al., 2016).

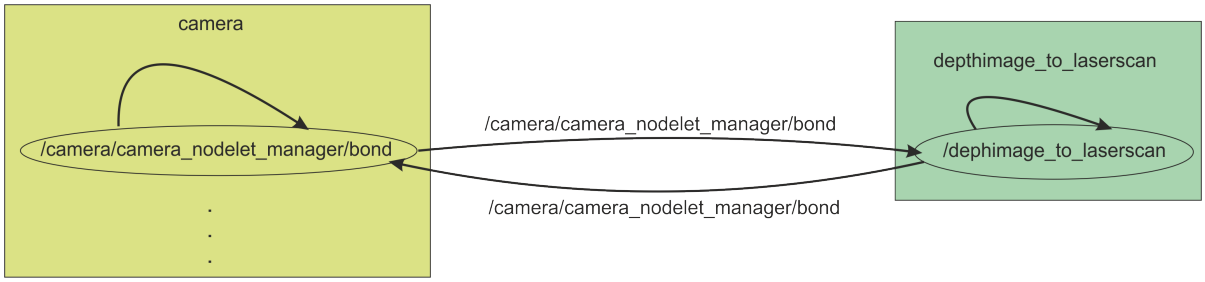



Figura 3.7: Nodo *camera* y nodo *depth_to_laserscan*

En la literatura se encuentran varias estrategias para la creación de mapas. Una de las primeras que se destacan siguiendo la filosofía de bajo coste computacional, es el algoritmo Fast-Slam (Montemerlo and Thrun, 2003), en donde se realiza la estimación de la posición del robot, empleando un filtro de partículas Rao-Backwellized (FPRB). Cada una de las partículas mantienen un mapa y cada posición de la partícula es conocida por lo tanto la creación del mapa tiende a ser mas precisa (Monjazeb et al., 2011).

Posteriormente, Grisetti et al. (2007) proponen el algoritmo Fast-Slam 2.0 que es empleando con aceptación en modelos de robótica móvil.

Para la gran mayoría de robots móviles, orientados a la realización de mapas perpendiculares al robot, Nguyen et al. (2007) se introduce un algoritmo basado en (OrthoSLAM) que se fundamenta en la extracción de segmentos de línea ortogonales del ambiente y así reducir el consumo de memoria. Este algoritmo ha sido probado satisfactoriamente en ambientes cerrados; en ambientes abiertos presenta errores. Esto es debido a que el procesamiento de la imagen se ve afectada por los cambios de luz.

### Filtro de Partículas Rao-Backwellized

El algoritmo Fast-SLAM 2.0 usa el concepto de Filtro de Partículas Rao-Backwellized (FPRB) se fundamenta en la estrategia de división en rejillas, con algunas modificaciones en cuanto a la estimación y el momento en que se realiza el muestreo. Los elementos principales del algoritmo establecidos en Grisettiyz et al. (2005) son:

- Muestreo
- Asignación de peso a cada partícula la es considerada una buena representación del mapa
- Muestreo Adaptativo
- Estimación del Mapa

ROS ofrece varios tipos de algoritmos para la construcción de mapas: $HectorSLAM$; $Gmapping$; $KartoSLAM$; $CoreSLAM$; $LagoSLAM$. De acuerdo con Santos et al. (2013), los algoritmos $HectorSLAM$, $Gmapping$ y $KartoSLAM$ son los de mayor precisión en la construcción del mapa. $Gmapping$ y $KartoSLAM$ son algoritmos basados en FPRB, a diferencia de $HectorSLAM$ que se basa en OrthoSLAM.

Según Grisetti et al. (2007), *Gmapping* es apropiado para sistemas que tienen una limitación en el procesamiento, característica que lo hace apropiado para el desarrollo de este trabajo ya que la implementación se realiza sobre una *Raspberry Pi*. El algoritmo emplea un numero bajo de partículas (Entre 5 y 30 dependiendo de la precisión que se desee obtener) entonces, permite disminuir el procesamiento y el uso de la memoria RAM. Por lo tanto, se selecciona como estrategia para ser implementada en el robot.

El nodo *gmapping* permite modificar los parámetros como por ejemplo el número de partículas, el umbral de muestreo entre otros. En consecuencia, permite ajustarse al robot que se desarrolla.

### Configuración del nodo gmapping

La creación de un mapa en ROS se realiza a través de una imagen en algunos de los formatos como png, jpg o pgm que se encuentran en escala de grises. Asociado a cada uno de los mapas se crea un archivo de extensión YAML en donde se almacena la información relacionada con la resolución, el tamaño de cada celda en metros, el origen del mapa y los umbrales que determinan si una celda esta ocupada o vacía. Por ejemplo, para uno de los mapas realizados con el robot, fue generado el archivo *mapa_01.yaml* que se observa a continuación:

```
image: /root/albeto/mirobot/maps/mapa_01.pgm
resolution: 0.050000
origin: [-12.200000, -12.200000, 0.000000]
negate: 0
occupied_thresh: 0.65
free_thresh: 0.20
```

En este mapa las celdas están representadas cada 5 cm, y se estableció una celda vacía cuando el valor es menor al 20% del rango permitido. Las celdas vacías se aprecian en color blanco; cuando el valor de lectura por el sensor es mayor al 65% se asignará como celda ocupada en color negro; las demás que se encuentren en valores intermedios serán definidas como celdas desconocidas. En la figura 3.8 se aprecia una representación respecto a un mapa. Esta clasificación es muy útil cuando se realiza el nodo de planificación de trayectorias.

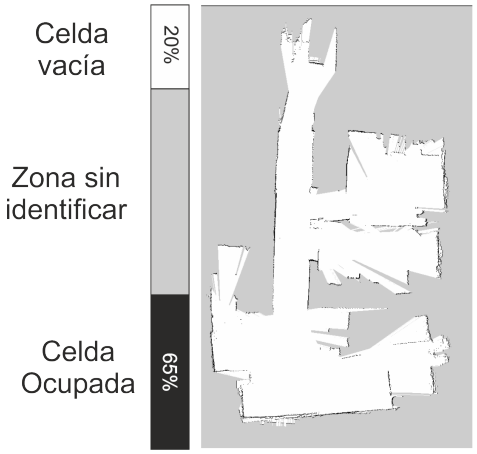



Figura 3.8: Identificación de celdas ocupadas y vacías en el mapa

Para realizar la configuración del nodo, se crea un archivo mapa.launch el cual carga el nodo y se realiza la configuración de la frecuencia de actualización del mapa, la cantidad de iteraciones, así como en numero de partículas, entre otros. A continuación se muestra un fragmento de las definiciones empleadas.

```
<node pkg="gmapping" type="slam_gmapping"
 name="slam_gmapping" output="screen">
   <param name="base_frame" value="base_footprint"/>
   <param name="map_update_interval" value="5.0"/>
   <param name="iterations" value="5"/>
   <param name="particles" value="80"/>

   <!-- Resolucion de muestreo -->
   <param name="delta" value="0.01"/>
   <param name="llsamplerange" value="0.01"/>
   <param name="llsamplestep" value="0.01"/>
   <param name="lasamplerange" value="0.005"/>
   <param name="lasamplestep" value="0.005"/>
   <remap from="scan" to="$(arg scan_topic)"/>
 </node>
```

En la figura 3.9 se observa un diagrama de la arquitectura empleada para la construcción del mapa y su visualización en una computadora externa. Para ello se emplea la interfaz gráfica *rviz*, que nos permite visualizar el comportamiento del robot. En la sección 4.1 se expondrá en detalle su funcionamiento.

## 3.7 Navegación

La navegación autónoma requiere la configuración de varios parámetros que permiten realizar la tarea de ubicar el robot en el mapa y planificar una trayectoria para desplazarse de un punto a otro. La arquitectura de navegación se presenta en la figura 3.10, en donde se configuran parámetros para los módulos AMCL, base_loca_planner, Costmap y Local_Planner.

#### Adaptative Monte Carlo Localización

El nodo Adaptive Monte Carlo Localization **(AMCL)**, implementa un conjunto de algoritmos probabilísticos para la localización, incluyendo *sample_motion_model_odometry*, *beam_range_finder_model*, *likelihood_fiel_range_finder_model*, *Augmente_MCL*, entre otros.

La localización del robot o ubicación es representada por medio una posición y una orientación en el mapa. Para ello el nodo AMCL entrega un conjunto de posibles ubicaciones donde es posible que el robot se encuentre (filtro de partículas). Cada uno de los posibles lugares donde se encuentra el robot es asociado a una probabilidad, que tiende a ser alta cuando hay correlación con la ubicación en el mapa. Para calcular la probabilidad, el nodo compara la lectura de los sensores con los datos que esperaría estar leyendo de acuerdo al mapa; si la lectura es consistente, entonces la probabilidad aumentará.

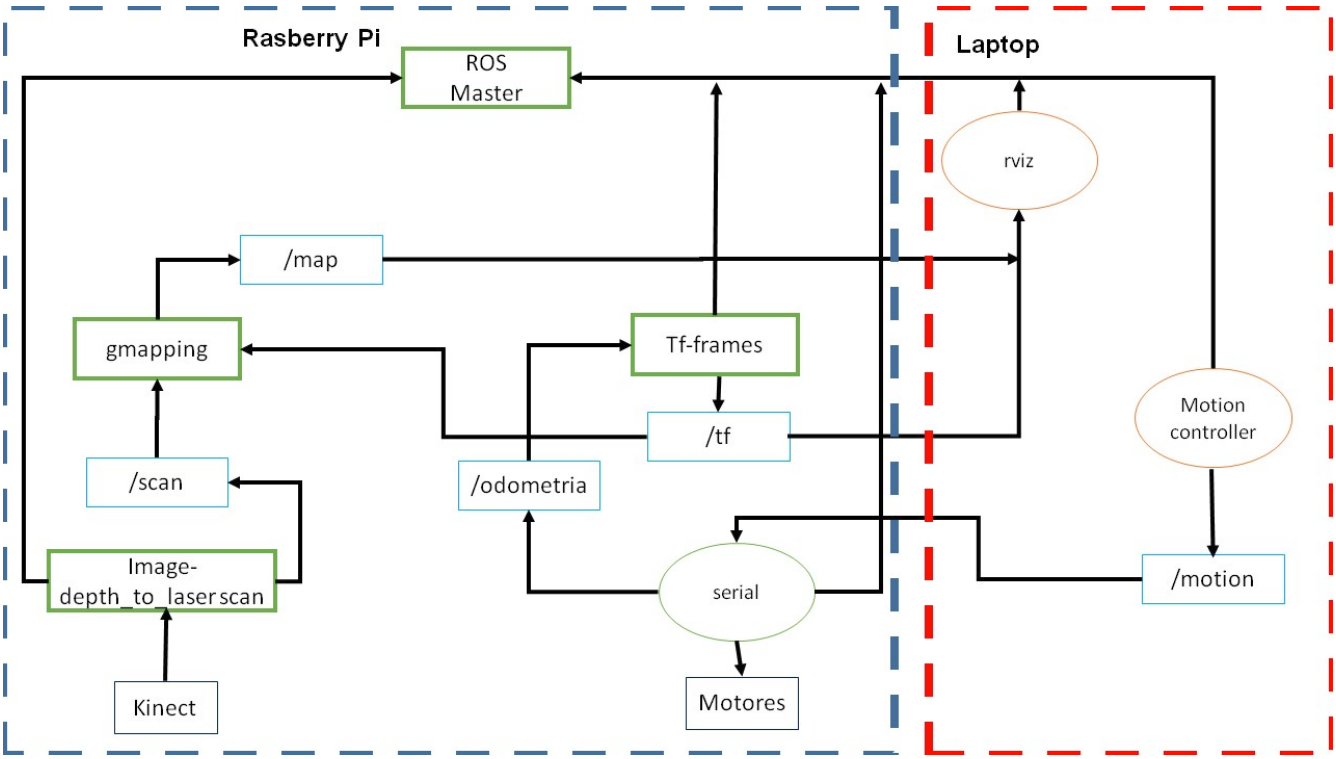



Figura 3.9: Módulo de construcción del mapa

El nodo se inicia con un conjunto de ubicaciones posibles donde se cree que el robot se encuentra; para ello, durante el desarrollo de este trabajo se trata de hacer coincidir la lectura del sensor frente a una esquina y su representación en el mapa. Una vez que el robot inicia su movimiento, los conjuntos de posiciones convergen hasta la posición del robot. Para el cálculo de la trayectoria, la posición con la que inicia es aquella donde la probabilidad es la mas alta.

Este nodo se subscribe a los Topics:

- */Scan* : corresponde la señal que entregada por el Kinect y una vez convertida en una señal tipo laser, como se menciono en al sección 3.5.
- */tf* : permite realizar la transformación en las coordenadas de referencia.
- */initialpose* : Es la posición actual, y con la que se realimenta el filtro de partículas.
- */map* : El nodo AMCL se subscribe a este Topic para realizar el emparejamiento de la lectura de los sensores y el valor que se esperaría obtener de ellos para cada una de las partículas.

Por otro lado, este nodo publica los siguientes Topic:

- */amcl_pose* : entrega la ubicación (posición y orientación) que ha tenido la mas alta probabilidad.
- */particlecloud* : este Topic contiene el conjunto de posiciones estimadas.
- */tf* : Publica la transformación para el Topic */Odom/* encargado de re-ubicar al robot en el mapa.

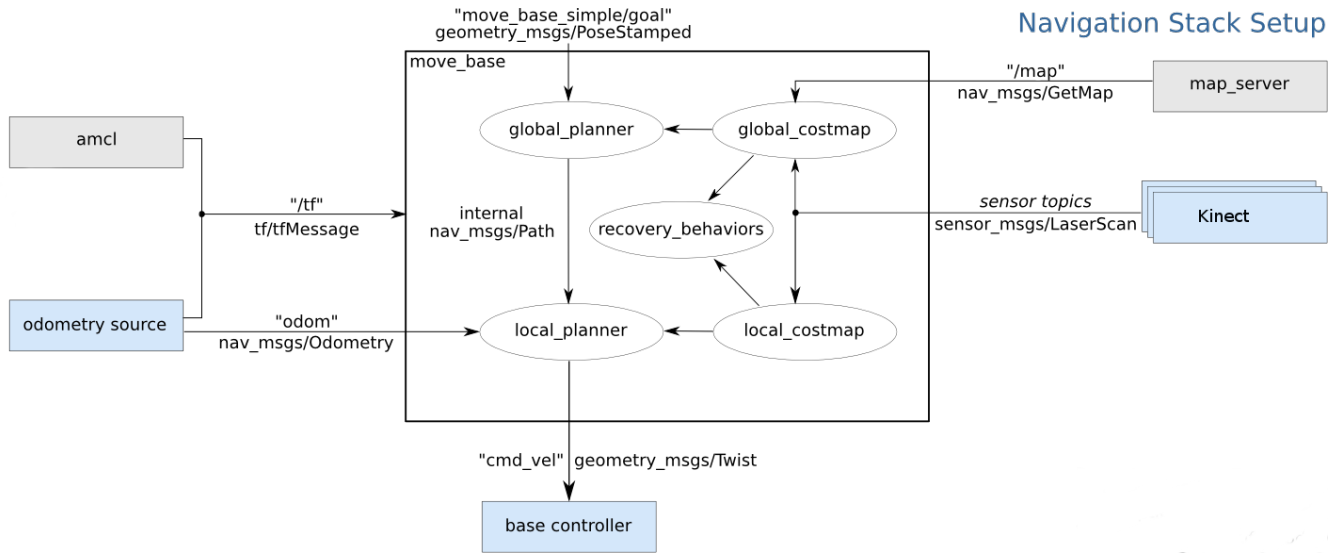



Figura 3.10: Modulo de navegación. Tomado de `http://wiki.ros.org/navigation` Modificado de acuerdo al desarrollo presentado en este documento.

Los parámetros configurables para el nodo ACML se pueden dividir en tres categorías: *Modelo_laser*, *generalidades_del_filtro* y *modelo_de_odometria*. En la figura 3.11 se describen cada una de las categorías. La configuración de los parámetros se realiza en el archivo miRobot_acml.launch.




Figura 3.11: Parámetros nodo AMCL

### Configuración *Costmap*

El nodo de navegación utiliza dos archivos en los cuales se almacena la información asociada a la identificación y evasión de obstáculos y la creación de trayectorias. Para ello, se crean dos archivos en donde se guardarán las configuración para el mapa local y el mapa global. La documentación en `http://wiki.ros.org/costmap_2d` presenta una descripción detallada sobre cada uno de los parámetros. A continuación se describe la configuración del *costmap_local*

```
local_costmaps:
    global_frame: odom
    robot_base_frame: /base_footprint
    update_frequency: 5.0
    publish_frequency: 2.0
    static_map: false
    rolling_window: true
    width: 4.0
    height: 4.0
    resolution: 0.05
    transform_tolerance: 0.5
```

En él se observa la configuración relacionada con la frecuencia de actualización, resolución, sistema de referencia entre otros. Así mismo, se realiza la configuración del los parámetros para el *costmap_global*:

```
global_costmap:
    global_frame: /map
    robot_base_frame: /base_footprint
    update_frequency: 1.0
    publish_frequency: 0.5
    static_map: true
    transform_tolerance: 0.5
```

**Configuración *Local Planner***

Actualmente no existe un método analítico estandarizado para determinar las trayectorias de un robot. Existen diferentes estrategias que permiten brindar una solución al problema como las propuestas en Helguero (2016) y Kelly (1994) entre otras investigaciones. Para este trabajo se ha implementado un planificador local que es el responsable de realizar los cálculos necesarios para enviar al robot la información cinemática para realizar el desplazamiento.

Este nodo se fundamenta en el algoritmo *Dynamic Window Approach* (DWA). La base de este algoritmo es considerar una trayectoria como la suma de arcos en intervalos de tiempo pequeños.

El algoritmo se inicia leyendo la velocidad actual del robot, y calculando todas las posibles velocidades discretas que el robot puede tomar; cada una de las velocidades generan una trayectoria en forma de arco. Posteriormente se evalúan las velocidades que no generaran colisión en el siguiente estado, descartando aquellas que puedan presentar un posible choque.

Paso seguido, se determinan las velocidades que el robot puede alcanzar en el siguiente lapso de tiempo, descartando aquellas que son imposibles de implementar. Por último queda un conjunto de velocidades que puede ser implementada. Para determinar cual seleccionar se realiza un función de coste en donde se analiza sobre tres criterios: alineación con el objetivo; espacio libre y máxima velocidad alcanzable.

Para realizar la evaluación sobre el nodo, se realiza la modificación de los parámetros relacionados con el planificador de trayectorias. Para ello se carga la configuración a través de `ros master`, que permite al nodo `base_local_planner` realizar la actualización para el cálculo de la trayectoria. A continuación se muestran algunos de los parámetros configurados para el planificador:

```
# Configuración velocidades
  max_vel_x: 0.3
  min_vel_x: 0.1
  max_vel_theta:  1.0
  min_vel_theta: -1.0
  min_in_place_vel_theta: 0.6

  acc_lim_x: 0.5
  acc_lim_theta: 1.0

# Tolerancia para el objetivo
  yaw_goal_tolerance: 0.3
  xy_goal_tolerance: 0.15

# prevencion de oscilación
  oscillation_reset_dist: 0.05
```

# Capítulo 4

# Resultados

Finalizada la construcción del hardware, así como el diseño del software empleado para realizar la navegación autónoma, se presentan los resultados obtenidos mediante las simulaciones, así como las pruebas sobre el robot final.

## 4.1 Simulaciones en rviz y gazebo

La flexibilidad en la arquitectura de ROS permite la integración de diferentes herramientas, como por ejemplo **rviz** y **gazebo**. A través de estos frameworks es posible simular el modelo del robot, analizar el comportamiento de los sensores y los diferentes algoritmos que se han desarrollado.

**Gazebo** es empleado para realizar la simulación de nuestro modelo descrito en la sección 3.2, en donde se muestra el diseño del robot mediante un archivo .urdf. Gazebo nos permite configurar el ambiente en donde el robot se desplazará. A continuación se realiza una explicación de las características funcionales del simulador:

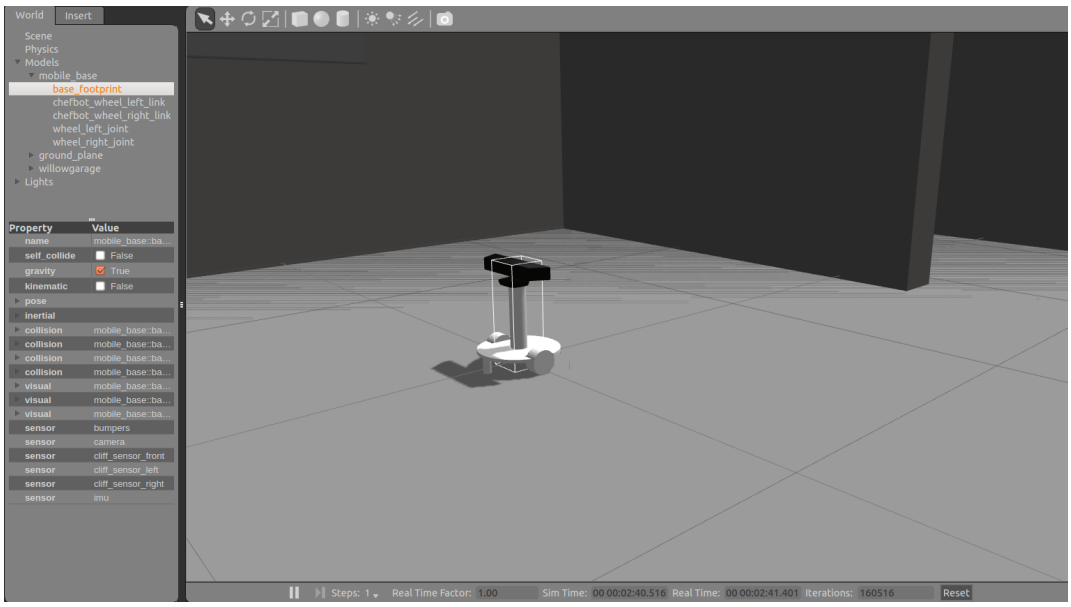



Figura 4.1: Interfaz de simulación del robot interactuando en un ambiente.

En la figura 4.1, en la parte central se observa el robot en interacción con un ambiente. En la

parte derecha se observan los elementos relacionados con el modelo que fueron escritos en el archivo *ibot_description.urdf* desde allí es posible modificar los parámetros que se consideren necesarios. En la parte inferior de la imagen, se aprecia el tiempo transcurrido de la simulación.

La lectura de los sensores es publicada a través de los correspondientes Topic asociados al robot que se encuentran en el ambiente. En gazebo, no es posible visualizar lo que ocurre internamente en el robot; para ello empleamos rviz.

Para realizar el desplazamiento del robot es necesario enviar el mensaje a través del Topic /cmd_vel, el cual es generado desde un nodo en donde se asocia la lectura del teclado a un desplazamiento.

La configuración del ambiente donde el robot interactúa, se realiza mediante el archivo *ibot_world.launch* en donde se configura el Kinect como un sensor laser *"virtual"*. Para la simulación se ha cargado un modelo de un ambiente 3D en el que se realiza la simulación de la construcción del mapa, así como la interacción de los sensores con el entorno. A continuación se muestra un fragmento de la configuración realizada:

```
<?xml version="1.0"?>
<launch>
  <remap from="/cmd_vel_mux/input/teleop" to="/cmd_vel" />
<!-- Ambiente a utilizar -->
        <include file="$(find gazebo_ros)
                        /launch/willowgarage_world.launch" >
                        <param name="/use_sim_time" value="true" />
        </include>

  <!-- Laser -->
  <remap from="image" to="/camera/depth/image_raw"/>
  <remap from="scan" to="/scan"/>
  </node>
</launch>
```

Rviz nos permite realizar la verificación de cada uno de los topics que se han implementado, algunos de ellos pueden ser visualizados, de tal manera que logramos identificar el comportamiento que presentan cuando el robot ejecuta determinadas tareas. Por lo tanto, rviz requiere la ejecución del ROS Máster y a su vez la generación de los Topics; éstos pueden ser generados por el simulador gazebo o directamente por el robot en tiempo real. Para ilustrar un poco mejor este concepto, en la figura 4.2 se observa la interfaz de rviz con el robot.

Rviz se compone principalmente de dos ventanas: En la parte central se encuentra la visualización del robot y los topics como el sensor láser, la cámara, y otros topics que sean generados por algún algoritmo como por ejemplo los mapas. En la parte derecha se encuentran los Topic con los que actualmente se han cargado para el análisis. En caso de ser necesario, se podrán añadir los Topic que fueran necesarios; por ejemplo en la figura 4.3 se muestran diferentes tipos de mensajes que pueden ser cargados desde rviz.

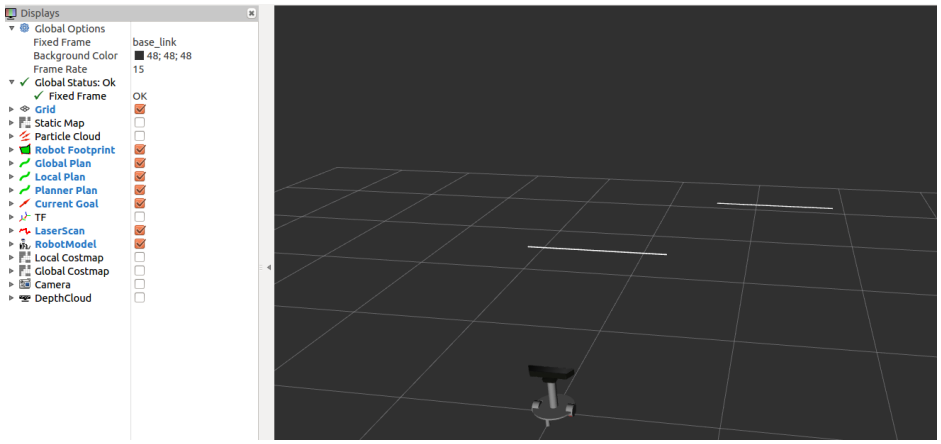



Figura 4.2: Interfaz de rviz con lectura de diferentes Topic

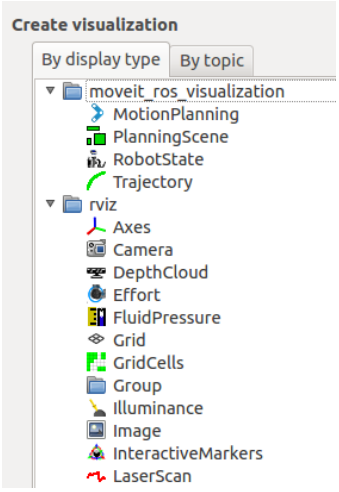



Figura 4.3: Carga de Topic al ambiente rviz

En la figura 4.2, se encuentra activo el Topic *LaserScan* que corresponde al resultado de emplear el nodo *ImageDepth_to_laserScan* en donde, con una línea blanca representan los obstáculos. En este caso, las líneas corresponden a los dos muros que se encuentran a distancias diferentes, estos son obtenidos de los Topic enviados por la simulación en gazebo que se muestra en la figura 4.1. Por ejemplo, en la imagen se observa la activación de la lectura para el Topic asociado con la cámara de profundidad.

Esta imagen corresponde al mismo punto de simulación representado en la figuras 4.1 y 4.2 en donde se obtiene una imagen en profundidad proyectada sobre el espacio de simulación. Toda la configuración de rviz se ha realizado de forma manual; es decir, se realiza la lectura y visualización de cada uno de los Topic según se determine en cada caso de estudio.

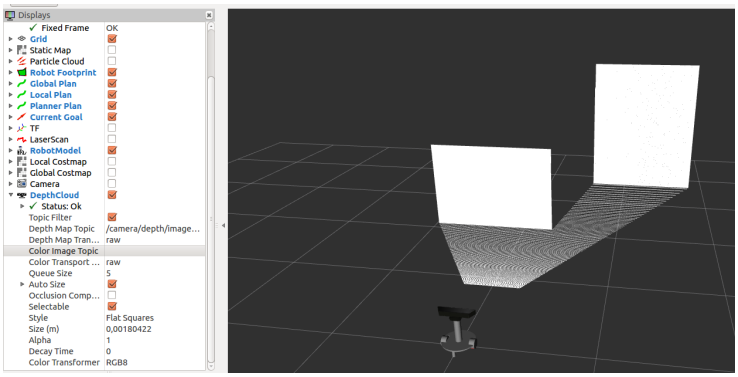



Figura 4.4: Cámara de profundidad en rviz

## 4.2 Pruebas de funcionamiento

A continuación se realiza la descripción de las pruebas en el robot, relacionadas con el control en remoto y la respectiva visualización, la construcción de un mapa, la navegación en el mapa y algunas pruebas relacionadas con la evasión de obstáculos.

### 4.2.1 Control en Remoto del Robot

Los algoritmos planteados se ejecutan en dos fases: Controlado por un usuario y Navegación en modo autónomo. La primera fase esta relacionada con la construcción de un mapa en donde el usuario controla remotamente el desplazamiento del robot. En la segunda fase, el robot esta en modo autónomo y de forma remota se le envía una posición final, para que el robot realice el recorrido. Por lo tanto, en ambas situaciones se requiere emplear alguna estrategia para comunicarse con el robot.

Para ello, se opta por usar la comunicación vía WiFi puesto que la Raspberry Pi tiene embebido el puerto para esta conexión. Con el propósito de visualizar los Topic en el PC en remoto, es necesario configurar en el entorno de ROS las respectivas direcciones IP de `ros master`. Para el control del movimiento del robot, se ejecuta un script que se encarga de leer los comandos por teclado y enviar el respectivo Topic para su procesamiento y determinación de velocidad sobre cada una de las ruedas. En la figura 4.5 se observa la interfaz de control.

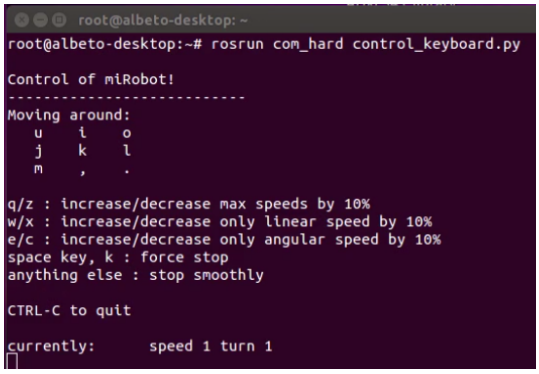



Figura 4.5: Control remoto del Robot

Por otro lado, una vez se realiza la comunicación con el robot, es necesario visualizar los Topic generados; para ello se carga rviz en el equipo remoto y se leen los mensajes enviados por el robot.



(a) Cámara de profundidad (b) Imagen del robot (c) Cámara RBG B-N

Figura 4.6: Lectura del sensor Kinect.

En la figura 4.6, en la parte izquierda se presenta una toma de la cámara de profundidad; en la parte central se encuentra la imagen del robot desde donde fue tomada la fotografía y en la parte derecha se observa la imagen en blanco y negro. Estas imágenes fueron tomadas desde el computador en remoto.

### 4.2.2 Construcción de un mapa

La construcción inicia con la carga de los nodos gmapping, ImageDepth_to_LaserScan y Odometría como se describió en la sección 3.6. Una vez se activa el módulo, se realiza la conexión desde un computador remoto, desde el cual se controla el robot y se realiza la navegación manual; de está manera se inicia la construcción del mapa como se observa en la figura 4.7.

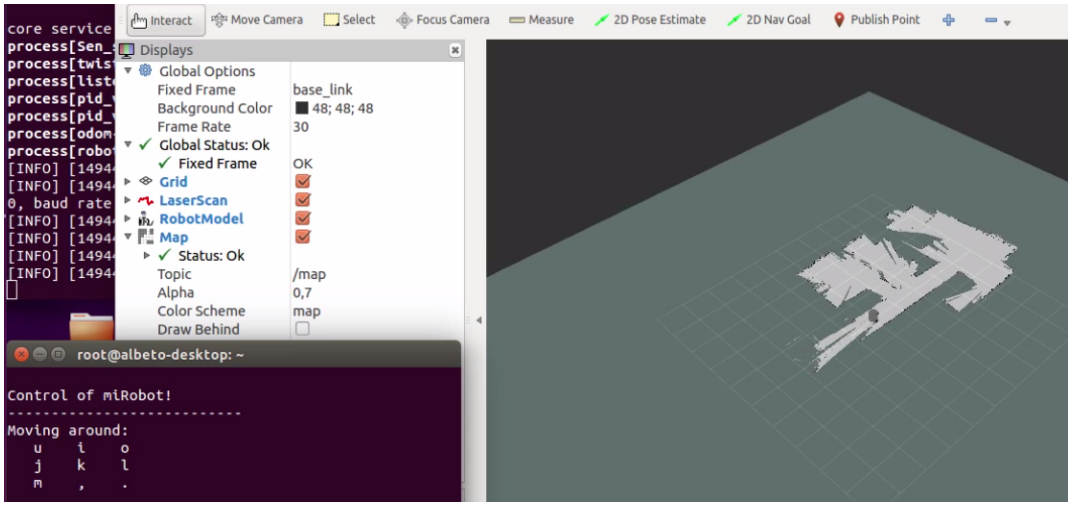
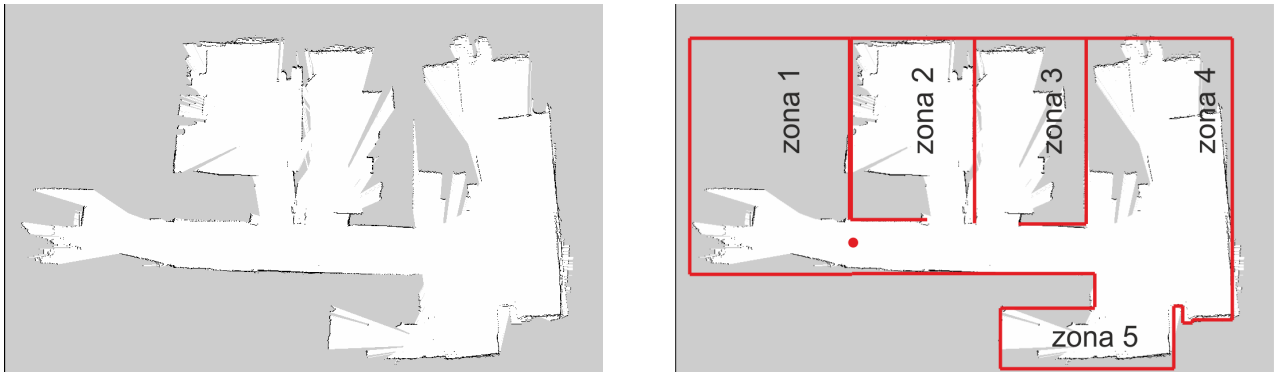



Figura 4.7: Construcción del Mapa

En la figura anterior vemos como el nodo va almacenando para cada una de las rejillas configuradas en el mapa, los obstáculos que va encontrando. Al terminar el recorrido controlado por el

ambiente, se obtienen los archivos map.yam y map.png.




(a) Mapa del robot (b) En el mapa real

Figura 4.8: Mapa construido por el robot

En la figura 4.8(a) se observa el mapa construido por el robot realizando la navegación controlada por el usuario. En la figura 4.8(b), en rojo, se aprecia el mapa real de la vivienda superpuesto al mapa realizado por el robot. El punto rojo indica el lugar donde se inicio el desplazamiento del robot con destino hacia la zona 4 y 5; una vez se realizó la identificación de estas zonas, el robot retorna ingresando a la zona 3 y 2 respectivamente. Finaliza en el mismo punto de inicio pero con orientación opuesta, por lo tanto se alcanzan a tomar muestras de la zona 1.

En el mapa se observan zonas en gris que corresponden al mobiliario del lugar. Las zonas en negro están asociadas a bordes de los obstáculos encontrados. Algunas zonas por donde el robot exploró para la construcción del mapa se muestran en la figura 4.9

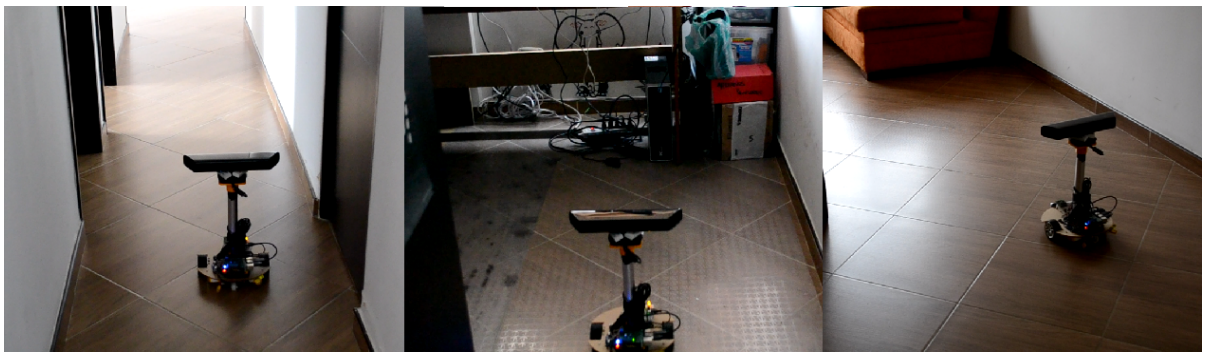


Figura 4.9: Robot realizando la construcción del Mapa

Los videos relacionados con la construcción del mapa y navegación se encuentran en:
`www.albertomorales.com.co/ibot`

### 4.2.3 Navegación Autónoma

La construcción del mapa nos permite identificar el lugar en el que se encuentra el robot, y los posibles espacios de navegación. La siguiente fase tiene como objeto emplear el mapa para realizar la navegación autónoma.

Como se describió en la sección 3.7 para la ejecución de la navegación es necesaria la configuración de varios nodos. A continuación se describirán los Topic utilizados:

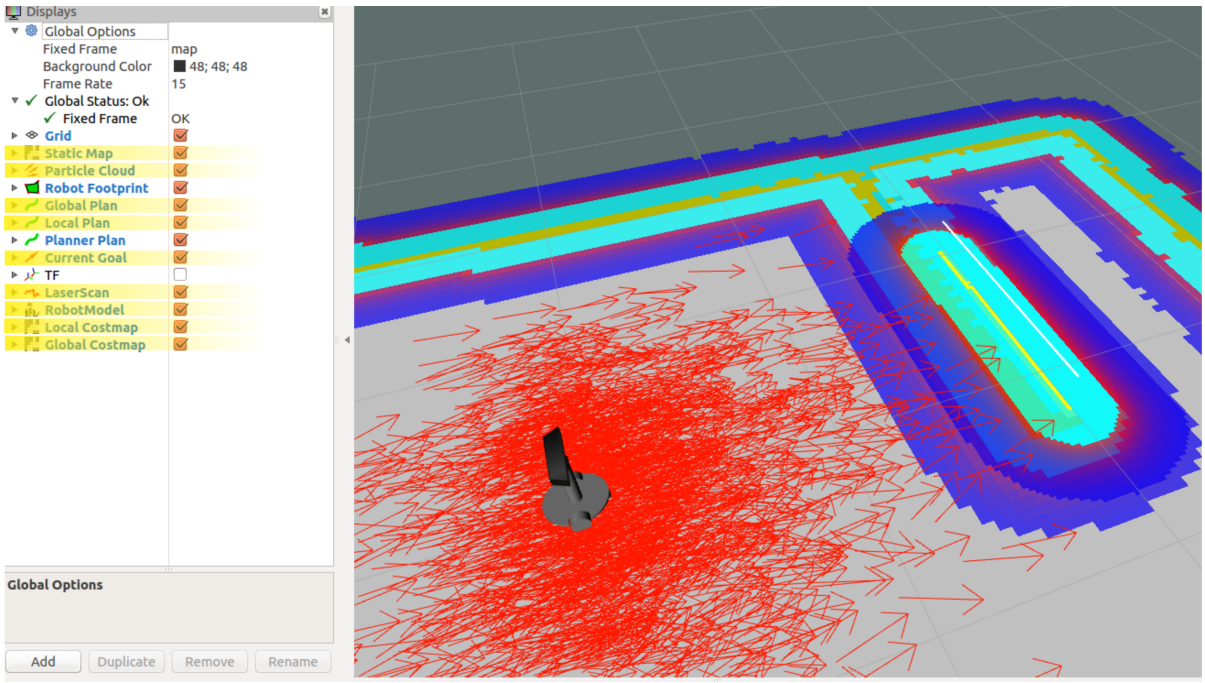
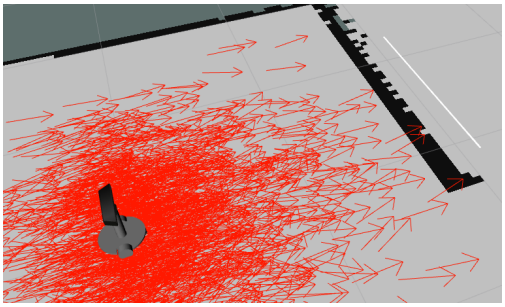


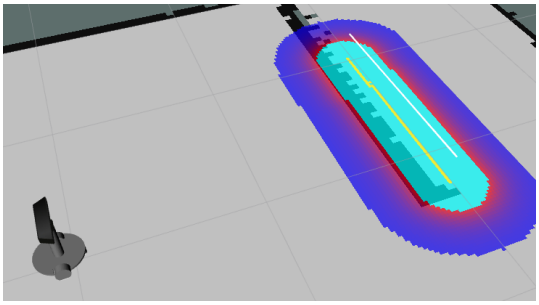

Figura 4.10: Interfaz para verificar la navegación del robot

En la figura 4.10 se observa la interfaz en rviz donde se pueden apreciar los parámetros relacionados con la navegación:

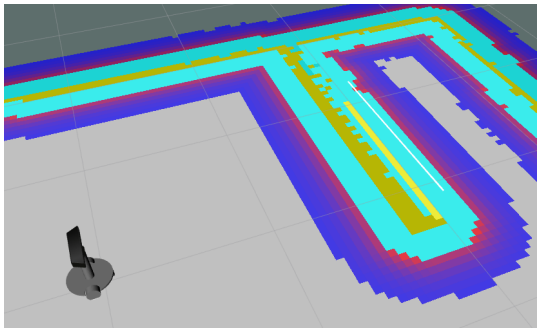
- **Static Map:** El mapa creado previamente por el robot es cargado leyendo el Topic /Map. El color negro representa muros/obstáculos para el robot.
- **Particle Cloud:** Corresponde al filtro de partículas, como fue descrito e la sección 3.10, cada una de las flechas rojas en la figura 4.11(a) corresponden a la probabilidad de que el robot se encuentre en esa posición. Para este caso la gran mayoría de partículas se encuentra cerca al robot.
- **Laser Scan:** Corresponde a la representación del láser. Cuando es detectado un objeto se dibuja una linea blanca sobre el objeto. En las imágenes correspondientes a esta sección se puede apreciar la lectura del láser; por ejemplo en la figura 4.11(a). En este caso pareciera que el objeto se encuentra distante del muro (linea negra); sin embargo, está representación es en 3D, en donde la transformación a una señal de láser se realiza a una altura de 15 cm del suelo.
- **Robot Model:** Corresponde al modelo (urdf) creado para el robot.
- **Local Costmap:** Este Topic carga las configuraciones previas para incrementar el tamaño del objeto encontrado. En la figura 3.10(b) se observa el robot y el Topic Local Costmap. El color amarillo corresponde al muro real, el color azul identifica la expansión que se ha realizado del muro para evitar choques.



(a) Filtro de partículas



(b) Local Costmap



(c) Global Costmap

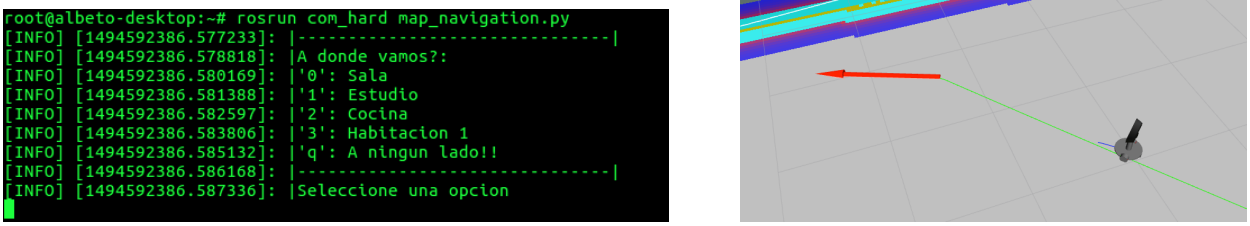
Figura 4.11: Identificación de los Topic en rviz

- **Global Costmap** A diferencia del Local Costmap, este Topic carga la configuración para todo el mapa aunque no sea leída por el sensor. Este método es empleado para calcular trayectorias y determinar los posibles caminos que puede transitar sin chocar con obstáculos.4.11(c)
- **Current Goal:** A través de este Topic el usuario publica la ubicación objetivo a donde se desea llevar el robot. Para el desarrollo de este prototipo se emplea un script con algunas posiciones fijas relacionadas con diferentes ubicaciones el en mapa. En la figura 4.12(a) se observa la interfaz empleada. El objetivo aparece marcado con una flecha en color rojo 4.12(b)
- **Global Plan:** Corresponde a la trayectoria que seguirá el robot. En la 4.12(b) la trayectoria aparece marcada con un línea verde, la cual el robot deberá seguir para alcanzar el objetivo propuesto.
- **Local Plan:** Corresponde a la trayectoria que esta llevando el robot. En la figura 4.12(b) se representa en color azul el *LocalPlan* y corresponde al objetivo local de trayectoria del robot.

### 4.2.4 Evasión de Obstáculos

Una vez la navegación se ha realizado satisfactoriamente, se realizan varias pruebas agregando obstáculos que no se encontraban en el mapa original. Por lo tanto el robot tiene que re-calcular la trayectoria para llegar al objetivo planteado.

En la figura 4.13 se presentan tres tomas correspondientes al experimento en donde el robot se encuentra con un obstáculo durante su trayectoria. La primera imagen corresponde a la trayectoria




(a) Selección destino (b) Trayectoria del Robot

Figura 4.12: Navegación del robot

calculada para alcanzar el objetivo; en la imagen central, el robot se encuentra con un obstáculo; para representarlo se incrementa su tamaño con el objeto de evitar colisiones y se re-calcula la ruta. 4.13.

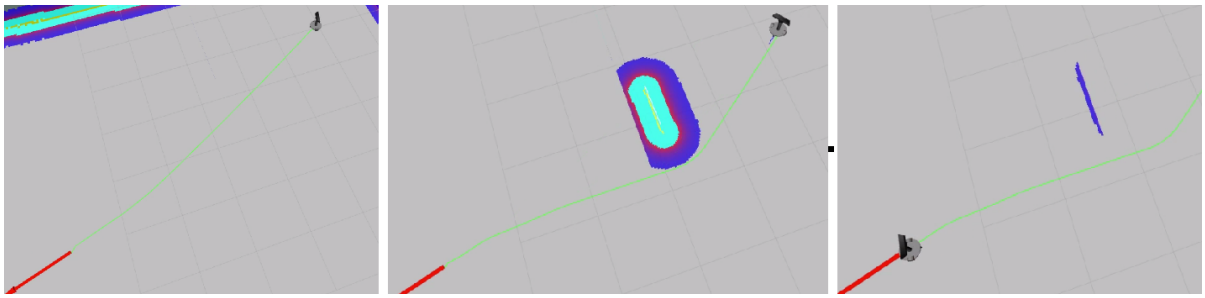
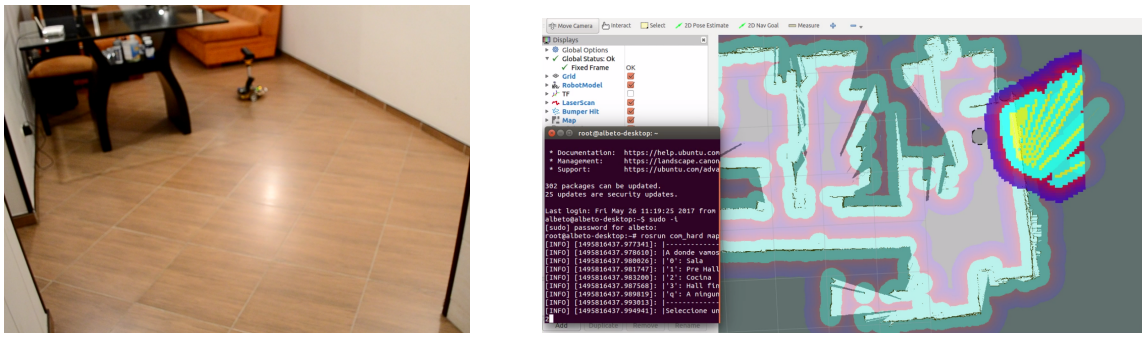


Figura 4.13: Secuencia de navegación con obstáculos

Una vez el robot ha superado el obstáculo éste empieza a desaparecer del mapa. Ésto es debido a que el nodo correspondiente al mapa es estático, por lo tanto no se actualiza, como se observa en el fotograma derecho de la imagen.

Finalizada la simulación, se realizan pruebas experimentales con el robot real. En las siguientes imágenes se presenta la secuencia del robot para ir de un punto a otro, y evadir un obstáculo.



(a) Inicio del Robot (b) Visualización en rviz

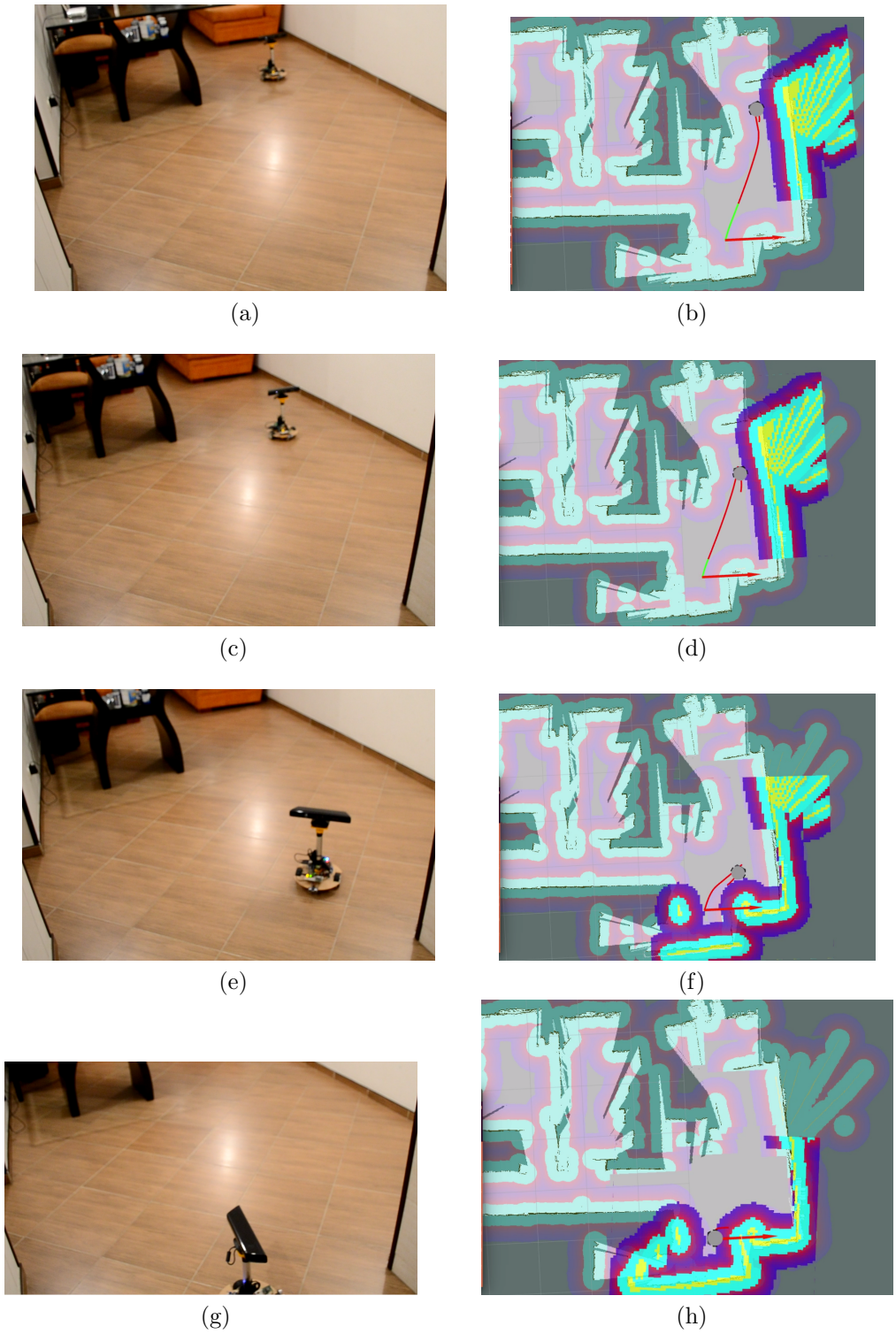
Figura 4.14: Experimentación con el robot

En la figura 4.14, en la parte derecha se observa la posición real del robot; en la parte izquierda se aprecia la información capturada por los sensores, la ubicación en el mapa y la trayectoria

propuesta para el robot.

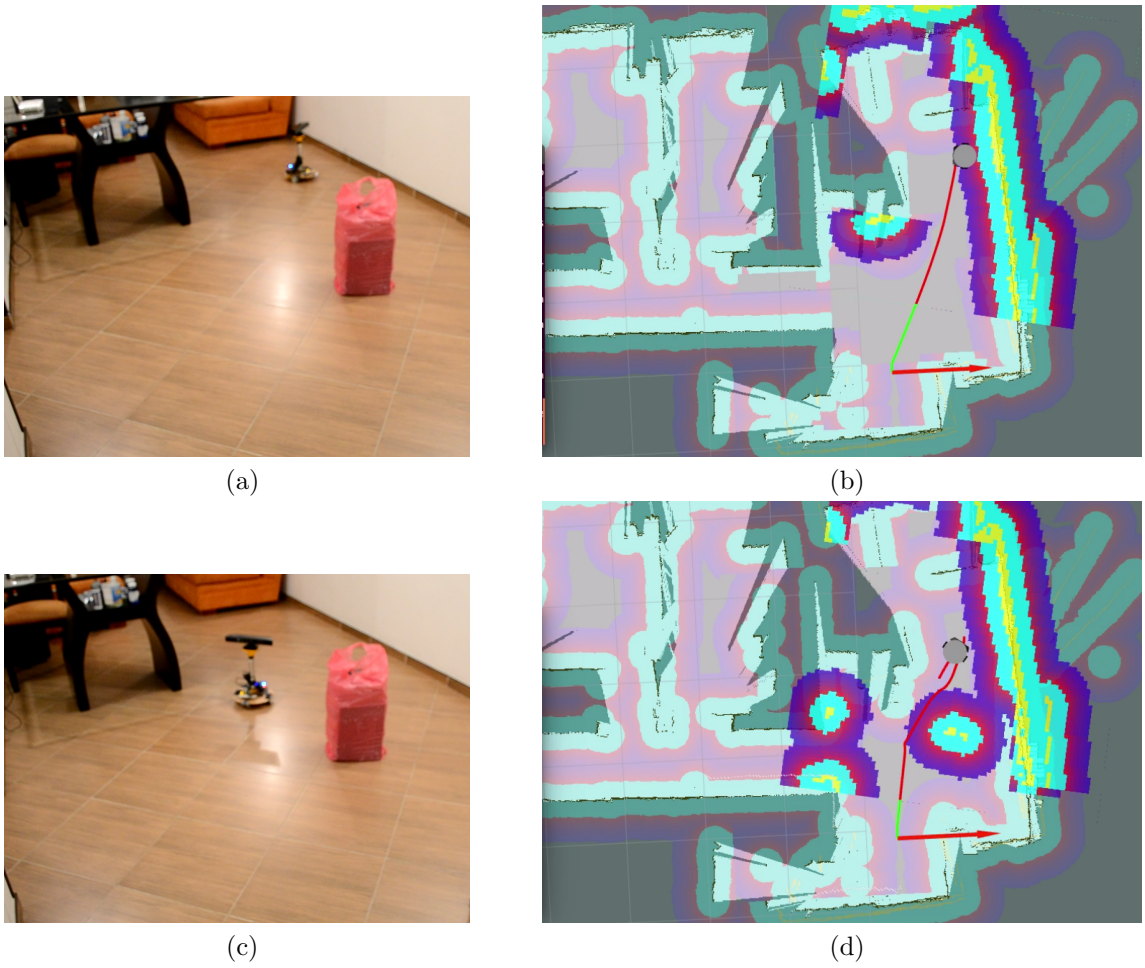
En la figura 4.15 se realiza el experimento de navegación para llevar el robot de una posición a otra; en la columna izquierda, las imágenes corresponden a la posición y desplazamiento real del robot; mientras en la columna derecha se observa la ubicación del robot en el mapa previamente creado. Adicionalmente es posible observar en rojo, la trayectoria propuesta para alcanzar el objetivo.

En la figura 4.16 se propone el mismo recorrido, sin embargo esta vez se agrega un obstáculo, que no estaba cargado en el mapa original como se observa en esta misma figura en el literal b. Allí el robot calcula la ruta a seguir, e inicia su recorrido. Tan pronto como el robot detecta un obstáculo, se re-calcula la trayectoria como se observa en la imagen d. De esta manera el robot hace evasión de obstáculos. En el link `www.albertomorales.com.co/ibot` se encuentran los videos relacionados con este experimento.



(a) (b) (c) (d) (e) (f) (g) (h)

Figura 4.15: Experimentación con el robot sin obstáculos agregados



(a) (b)

(c) (d)

Figura 4.16: Experimentación del robot con obstáculos agregados

# Capítulo 5

# Conclusiones

El producto final de este trabajo se focaliza en el desarrollo de una plataforma móvil de tipo diferencial, sobre el cual es posible realizar diferentes tipos de pruebas experimentales en torno a los problemas asociados a la robótica móvil. Como resultado, se ha diseñado y fabricado un robot móvil de tipo diferencial, procurando el bajo coste para su implementación, que incluye dos actuadores, cuatro sensores (Kinect, ultrasonidos, compás y encoders) y, como unidad de procesamiento, el sistema embebido *Raspberry Pi*.

El diseño en hardware electrónico, es realizado para permitir su expansión agregando otros sensores y actuadores si el usuario lo considera necesario. Esto es posible dado que: 1) el sistema de alimentación se realiza de forma modular, 2) los puertos *GPIO* de la *Raspberry Pi* se encuentran libres, excepto por cuatro pines empleados para el control de los actuadores; por tanto se pueden agregar otros motores, por ejemplo servomotores Dynamixel para añadir un brazo, 3) se permite la comunicación serial con el sistema de procesamiento principal, por lo tanto es posible cambiar el microcontrolador auxiliar para procesar una mayor cantidad de sensores.

Uno de los problemas comunes en un robot móvil se centra en la odometría, que es causado por alteraciones en la superficie o derrapes de las ruedas. Este tipo de error aumenta cuando el robot gira sobre su propio eje, dado que se presenta deslizamiento, causando que el cálculo de la orientación sea erróneo. Con el fin de mitigar este error en el robot presentado en esta memoria, se incorpora un compás encargado de medir la orientación del robot respecto al polo norte magnético de la tierra; estos datos en conjunto con el encoder son procesados para obtener una orientación con un menor error, lo que se concluye como satisfactorio en las pruebas finales del robot.

El robot se ha integrado satisfactoriamente con *Robot Operating System*, y al igual que el hardware diseñado, se permite la modularidad en el diseño de algoritmos y su posible expansión; brindando a la comunidad académica una plataforma flexible, para realizar pruebas experimentales de algoritmos relacionados con robótica, en el campo de visión por computador, técnicas de mapeo, técnicas de navegación, robótica social, entre otros.

La simulación es uno de los elementos mas importantes durante el desarrollo de algoritmos y estrategias para la interacción con el robot. Por lo tanto, se desarrolla el modelo que incluye los actuadores y sensores para la simulación del prototipo en formato *urdf*, que puede ser empleado por diferentes paquetes de software.

A partir del modelo se realiza la simulación empleando rviz y gazebo, en donde se prueban los algoritmos de mapeo y navegación. En concordancia con el hardware y el software planteados anteriormente, el modelo de simulación planteado permite ser modificado y expandido según los requerimientos del usuario.

El sistema propuesto, mediante un mando manual remoto, es capaz de realizar el mapa del lugar en donde se esta desplazando el robot. Para ello se implementa como estrategia el algoritmo de Grisetti et al. (2007), que se fundamenta en un filtro de partículas. Este algoritmo funciona muy bien sobre equipos de computo con buenas capacidades de procesamiento y memoria RAM; sin embargo, en el sistema embebido empleado, los recursos físicos no son tan amplios. Para solventar este problema se realizan diferentes pruebas disminuyendo el numero de partículas en el algoritmo con el objetivo de reducir el consumo de memoria. Esta experimentación finaliza satisfactoriamente, permitiendo al robot construir sus propios mapas.

Una vez finalizada la construcción del mapa, el robot es capaz de realizar la navegación autónoma. Para ello se ha implementado el algoritmo *Dynamic Window Approach* y el emplea un modulo AMCL que permite realizar la localización del robot para corregir su ubicación con cada desplazamiento. Al igual que en el mapeo, se emplea un filtro de partículas para determinar la posición del robot. Una vez se a determinado la posición, se realiza al sintonización de la función de coste con la que se determinan las trayectorias.

La construcción del mapa, así como la navegación autónoma del robot fueron validados a través de las simulaciones y posteriormente en la implementación sobre robot diseñado, mostrando resultados satisfactorios como se aprecia en los vídeos citados en este documento.
`www.albertomorales.com.co/ibot`

## 5.1 Trabajo Futuro

El desarrollo de este proyecto ha permitido explorar a profundidad diferentes campos de acción en robótica móvil. Por tanto, se plantean varias líneas de estudio que pretenden dar continuidad a este trabajo, así como profundizar en nuevas áreas. Como secuencia de este trabajo se proponen:

- Profundizar en nuevas estrategias para la creación de mapas en tiempo real empleando, técnicas basadas en Deep Learnig, con el objetivo de reducir el consumo de recursos físicos en el procesamiento de la información.
- Modificar la estrategia de posicionamiento, realizando odometría visual, en donde solo se emplee una cámara; es decir, realizar odometría visual monocular. Así, se potenciaría el desarrollo de robots móviles mas pequeños y de menor consumo de energía.
- Continuando con este trabajo, sería interesante generar una estrategia para almacenar los mapas en 3D. En el desarrollo de este prototipo se almacenan los obstáculos en un mapa de rejilla en 2D; sin embargo el Kinect también entrega la información en 3D; por lo tanto sería interesante crear un mapa de rejilla en el cual se almacenan las características de profundidad de la imagen. También sería apropiado un estudio del consumo de recursos para ser implementado en un único procesador.

- La robótica colaborativa permite estudiar diversas estrategias de interacción para resolver una problemática en particular. Continuando con el estudio de SLAM, la construcción de mapas de forma colaborativa, así como la planeación de trayectorias para nubes de robots se plantean como nuevas lineas de profundización.

# Apéndice A

# Instalación y Configuración

A continuación se realizará la descripción general de la instalación del sistema operativo para la Raspberry Pi, ROS y su correspondiente configuración, así como la instalación de los *packages* necesarios para la simulación y pruebas experimentales en el robot.

## A.1 Configuración tarjeta

En la tarjeta Raspberry Pi es común emplear *Raspbian* como el sistema operativo. Sin embargo, para el desarrollo de este trabajo se emplea *Ubuntu Mate xenial* el cual es una distribución basada en Ubuntu **armhf**. El sistema es instalado sobre una microSD de 32GB, aunque según las especificaciones se requiere un mínimo 6GB de espacio en memoria.

La instalación paso a paso del sistema operativo, asi como el link de descarga se encuentra descrita en `https://ubuntu-mate.org/raspberry-pi/` Una vez finalizada la instalación, al iniciar el sistema aparecerá una pantalla como la que se presenta en la figura A.1.

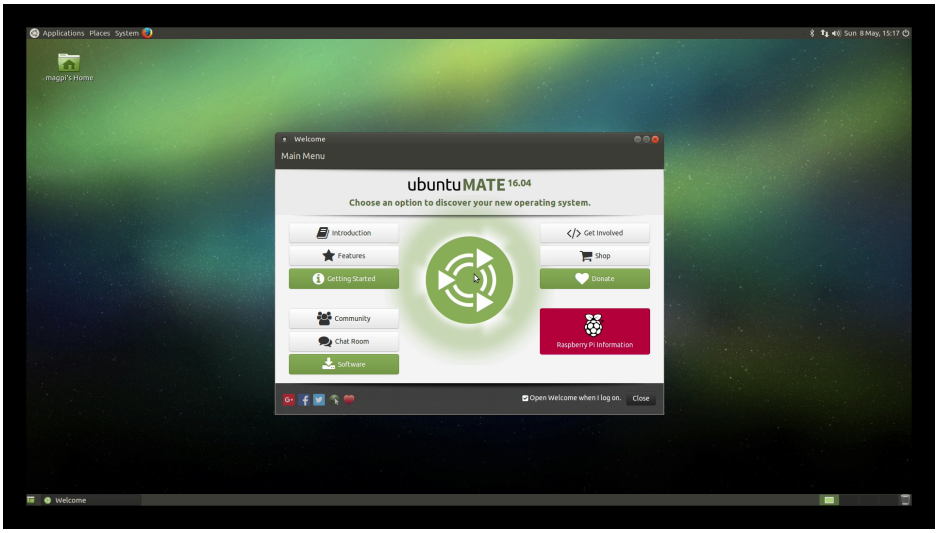



Figura A.1: Instalación Ubuntu Mate

Antes de continuar con la configuración básica, es necesario realizar la actualización del sistema operativo. Para ello, en una nueva terminal escribimos `apt-get update` con el objetivo de

actualizar las bases de datos de los *packages*; posteriormente las instalamos empleando el comando `apt-get upgrade` para realizar la instalación.

## A.2 Instalación de ROS

La versión empleada para el desarrollo de este trabajo es *ROS Kinectic Kame* que corresponde a la décima versión de ROS. Fue lanzada en mayo de 2016 y soporta Ubunto Wily y Xenial; este sistema operativo permite trabajar con procesadores armhf, siendo una de la razones para la elección del sistema operativo instalado en la Raspberry Pi.

En `http://wiki.ros.org/kinetic/Installation/Ubuntu`, se encuentra la descripción de instalación de ROS. Dado que se implementa sobre la tarjeta Raspberry Pi, no se hace una instalación del sistema completa. Por lo tanto, se instala la versión básica que no incluye herramientas de interfaz gráfica. Para ello en una terminal escribimos: `sudo apt-get install ros-kinetic-ros-base`. Posteriormente se instalan los *packages* gmapping y freenectic:

```
sudo apt-get install ros-kinetic-slam-gmapping
sudo apt-get install libfreenect-dev
sudo apt-get install ros-kinetic-freenect-launch
sudo apt-get install ros-kinetic-move-base
```

La instalación de ROS también debe realizarse en el equipo remoto. Para el desarrollo de este trabajo se empleo un computador portátil, bajo el sistema operativo Ubuntu TrusTy Tahr. Sobre el cual se instalo la versión *ROS Indigo*. Para este caso la distribución empleada no tiene relevancia, puesto que se utiliza para visualizar a través de las herramientas gráficas (rviz y gazebo) las simulaciones y la construcción del mapa desarrollado por el robot. El proceso de instalación se encuentra en `http://wiki.ros.org/indigo/Installation/Ubuntu`

## A.3 Instalación de packages

Para el desarrollo de este trabajo se crearon dos *packages* en los que se encuentran contenido los scripts relacionados con el control del robot, el mapeo y la navegación, así como los archivos necesarios para la simulación.

Antes de ser instalados y cargados, es necesario crear un *Workspace* en donde se cargarán todos los *packages*. Para ello se pueden seguir las orientaciones planteadas en: `http://wiki.ros.org/catkin/Tutorials/create_a_workspace`. Para la creación del workspace, en una terminal nueva se ejecuta la siguiente orden:

```
source /opt/ros/kinetic/setup.bash
```

Ahora se crea la carpeta del workspace, para este trabajo es *miRobot*. Es importante asignarle este nombre ya que algunos archivos se direccionan a esta carpeta.

```
mkdir -p ~/miRobot/src
cd ~/miRobot/
```

```
catkin\_make\_init\_workspace
catkin\_make
```

Con *catkin_make* se compila el espacio de trabajo y se crean los archivos setup.bash y las carpetas *build* y *devel*. Continuamos con la configuración del workspace:

```
source devel/setup.bash
```

Una vez configurado el espacio de trabajo, podemos agregar los *packages* creados. Por tanto, se guardan las carpetas com_hard e ibot_description que son proporcionadas como anexo a este documento. Por tanto la organización quedará como se plantea en la figura A.2.

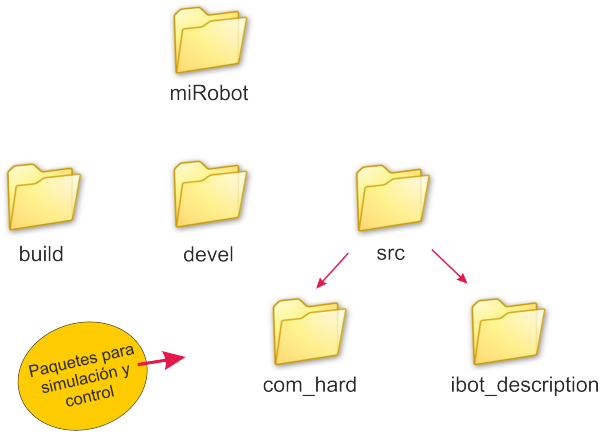



Figura A.2: Organización del workspace en ROS

### A.3.1 Descripción de los *packages*

Como se describió en la sección anterior se han realizado dos *packages*: Simulación y Control. A continuación se realiza la descripción del contenido y la organización en cada modulo.

#### Package com_hard

El modulo com_hard contiene los archivos relacionados con la comunicación serial, el mapeo y la navegación. Se organizan en carpetas según el tipo de información. Los archivos en cada carpeta corresponden a:

- **launch:** En esta carpeta se almacenan los archivos con extensión launch. Estos ficheros son creados para cargar varios nodos (programas y/o configuraciones) y contiene los archivos:
  - control.launch
  - 3dsensor.launch
  - base_local_planner_params.yaml
  - costmap_common_params.yaml
  - global_costmap_params.yaml
  - local_costmap_params.yaml

  - miRobot_acml.launch
  - miRobot_gmapping.launch
  - move_base.launch
  - view_navegacion.launch
  - view_robot.launch
  - include
    * amcl.launch.xml
    * gmapping.launch.xml
    * move_base.launch.xml

- **maps:** En este fichero se almacenan los mapas creados por el robot.
- **param:** Contiene los archivos de configuración del nodo move_base:
  - base_local_planner_params.yaml
  - costmap_common_params.yaml
  - dwa_local_planner_params.yaml
  - global_costmap_params.yaml
  - local_costmap_params.yaml
  - move_base_params.yaml
- **rviz:** Esta carpeta contiene los archivos que guardan la configuración empleada en rviz.
  - model.rviz
  - navigation.rviz
  - robot.rviz
- **scripts:** En esta carpeta se guardan todos los nodos del package:
  - ComunicacionHardware.py
  - control_keyboard.py
  - map_navigation.py
  - odometria.py
  - pid_velocidad_Der.py
  - pid_velocidad_Iz.py
  - sensores.py
  - SerialDataGateway.py
  - SerialDataGateway.pyc
  - talker.py
  - twist_to_motors.py

## Package ibot_description

El modulo ibot_description contiene los archivos relacionados que permiten cargar el modelo en el simulador. Se organizan en carpetas según el tipo de información. Los archivos en cada carpeta corresponden a:

- **launch:** En esta carpeta se almacenan los archivos con extensión launch. Estos ficheros son creados para cargar varios nodos (programas y/o configuraciones) y contiene los archivos:
  - base_local_planner_params.yaml
  - configuration_gazebo.launch
  - costmap_common_params.yaml
  - display.vcg
  - global_costmap_params.yaml
  - ibot_hotel_world.launch
  - local_costmap_params.yaml
  - modelo.launch
  - model_robot.launch
  - move_base.launch
  - navigation.rviz
  - include
    - ibot_base.launch.xml
- **maps:** En este fichero se almacenan los mapas creados por el robot.
- **meshes:** Contiene los archivos que cargan la mascara en 3D del robot.
- **robots:** En este fichero se emplea para guardar los modelos de robots prediseñados de donde se cargan los sensores del robot (Kinect).
- **rviz:** Esta carpeta contiene los archivos que guardan la configuración empleada en rviz.
  - model.rviz
  - navigation.rviz
  - robot.rviz
- **scripts:** En esta carpeta se guardan todos los nodos del package:
  - map_navigation.py
- **urdf:** En esta carpeta se encuentran los archivos que realizan la descripción del robot.
  - common_properties.urdf.xacro
  - ibot_base.urdf
  - ibot_base.urdf.xacro

- ibot_base.xacro
- ibot_base_gazebo.urdf.xacro
- ibot_gazebo.urdf.xacro
- ibot_library.urdf.xacro
- ibot_properties.urdf.xacro
- ibot_rviz_base.urdf.xacro
- ibot_rviz_library.urdf.xacro

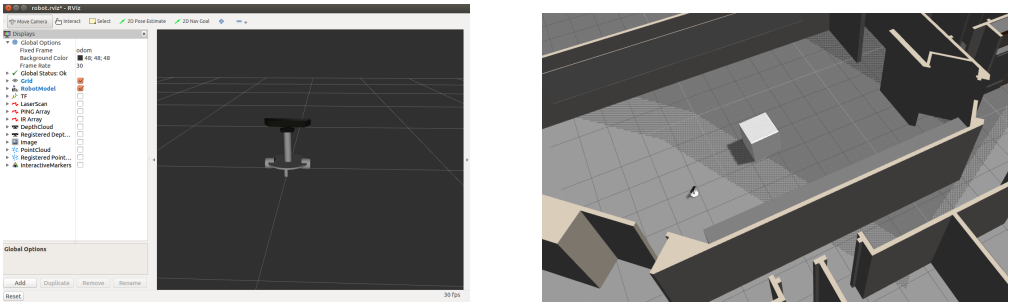
## A.4 Simulador y pruebas

Una vez finalizada la instalación del sistema operativo; de la configuración e instalación ROS y la configuración de los paquetes, el sistema esta listo para realizar prueba de los nodos programados. Para la experimentación con el simulador se debe ejecutar en el computador remoto y no se requiere tener encendido el robot físico; pero si es necesario haber realizado la configuración de los paquetes en el computador.

Iniciamos el sistema cargando el ROS Máster, para ello en una terminal nueva escribimos `ros_core`, que es el encargado de realizar la comunicación de todos los nodos y servicios implementados.
Posteriormente, en una nueva terminal podremos iniciar pruebas con cualquiera de los archivos de extensión launch. Cada archivo .launch debe cargarse en una nueva terminal de Ubuntu. Por ejemplo para cargar el robot en un ambiente predefinido se emplea:

```
roslaunch ibot\_description ibot\_hotel\_world.launch
```

Esta instrucción indica que se lanzará el archivo ibot_hotel_world.launch que se encuentra en el package ibot_description. Una vez ejecutada, se cargará el simulador gazebo con el robot y el ambiente en donde el robot puede interactuar. A su vez, se cargará rviz sobre el cual podremos ver la información capturada por los sensores. En la figura A.3 se observa la interfaz para gazebo y rviz.



(a) Robot en rviz (b) Robot en gazebo

Figura A.3: Simulación del robot

Para realizar el control manual del robot es necesario cargar en una nueva terminal:

```
rosrun com\_hard control\_keyboard.py
```

En este caso se ejecuta un archivo .py (python) por tanto empleamos la instrucción `rosrun`. Adicionalmente se indica que el archivo se encuentra en el package com_hard. Una vez cargado el programa se puede realizar el desplazamiento emplenado las teclas `u,i,o,j,k,l,n,m`, las cuales permiten el desplazamiento hacia adelante, atrás, parar y rotación sobre su eje. En la figura 4.5 se aprecia la interfaz de usuario.

Para realizar la pruebas directamente con el robot, es necesario establecer la comunicación entre el robot y el computador en remoto, configurando el sistema para que ROS Máster se ejecute sobre la Raspberry Pi y envíe Topics para visualizar la construcción del mapa, sensores y en general, la ejecución de los programas a través de rviz. Para ello, se modifica el archivo .bashrc de la Raspberry pi y del computador remoto como se observa en la figura A.4. Terminada la configuración, se realiza la conexión vía ssh entre el robot y el computador en remoto.

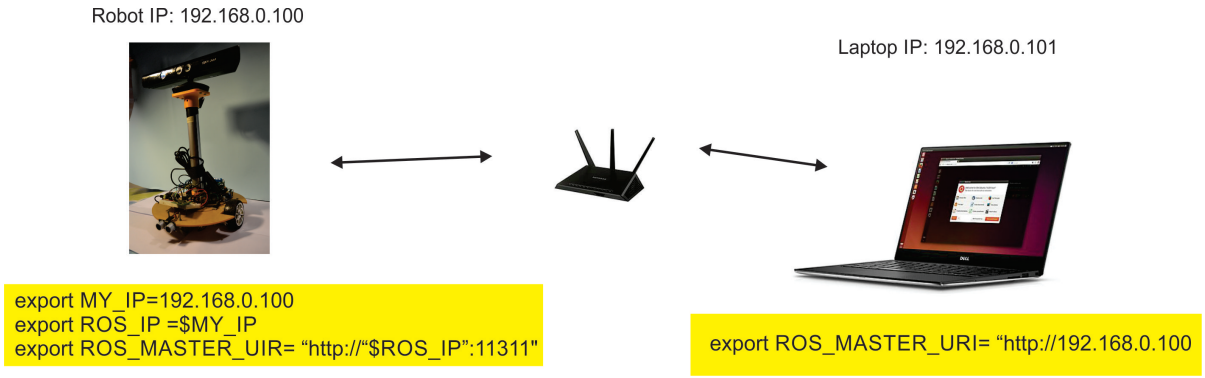



Figura A.4: Modificación archivo .bashrc

Por último, se carga `ros master` en el robot: `ros_core` y desde el computador en remoto se cargarán archivos .launch para la realizar la tarea deseada. Para realizar el mapa del robot se requiere cargar los siguientes archivos:

```
roslaunch com\_hard control.launch\\
roslaunch com\_hard miRobot\_gmapping.launch\\
rosrun com\_hard control\_keyboard.py\\
```

y en una nueva terminal del equipo en remoto, sin conexión con el ROS MASTER, cargamos rviz:

```
rosrun rviz rviz\\
```

# Apéndice B

# Estimación coste del robot

A continuación se presenta una estimación del coste de los dispositivos empleados para la construcción del robot.

| **Cantidad** | **Dispositivo** | **coste total en Euros** |
|---|---|---|
| 1 | Raspberry Pi con microSD 16 GB | 38.65 |
| 1 | Microcontrolador Astar32U4 | 11.70 |
| 1 | Kinect | 150 |
| 1 | Compás CMPS03 | 26.17 |
| 1 | Sensor Ultrasonido | 2.58 |
| 1 | Batería Lipo 7.4v 2000mAh | 13.19 |
| 1 | Batería Lipo 11.1v 2500mAh | 26.39 |
| 1 | Convertidor DC-DC | 3.70 |
| 1 | Regulador StepDown | 8.75 |
| 1 | driver MC33926 | 26.60 |
| 2 | Motoreductor 37D con encoder | 77.2 |
| 2 | llantas | 10 |
| 2 | Soportes e impresiones 3D | 5 |
| 1 | Tornillos y varios | 10 |
| | **Total** | 409.93 |

Tabla B.1: Estimación coste del robot

En la tabla B.1, se calcula el coste aproximado para el desarrollo de este prototipo. En la tabla B.2 se muestra el costo de algunos robot comerciales diseñados para realizar funciones similares a las propuestas en este trabajo.

| **Robot** | **coste en Euros** |
|---|---|
| Dr. Robot X80SV | 3,185.00 |
| TurtleBot 2 | 1,821.31 |
| Pioneer P3-Dx | 4000 |

Tabla B.2: Coste robots comerciales

# Apéndice C

# Listado de siglas

**ABS:** Acrilonitrilo Butadieno Stireno
**AMCL:** Adaptive Monte Carlo Localization
**ARMHF:** Procesador ARM tipo "Hard Float"
**BASH:** Bourne again shell
**DWA:** Dynamic Window Approach
**FPRB:** filtro de partículas Rao-Backwellized
**ROS:** Robot Operating System
**SSH:** Secure SHell
**URDF:** Unified Robot Description Format
**YAML** Ain't Another Markup Language

# Bibliografía


Abdelrasoul, Y., Saman, A. B. S. H., and Sebastian, P. (2016). A quantitative study of tuning ros gmapping parameters and their effect on performing indoor 2d slam. In *Robotics and Manufacturing Automation (ROMA), 2016 2nd IEEE International Symposium on*, pages 1–6. IEEE.

Adams, J. L. (1961). *Remote control with long transmission delays*. PhD thesis, PhD in Mechanical Engineering.

Beno, P., Duchon, F., Tolgyessy, M., Hubinsky, P., and Kajan, M. (2014). 3d map reconstruction with sensor kinect: Searching for solution applicable to small mobile robots. In *2014 23rd International Conference on Robotics in Alpe-Adria-Danube Region (RAAD)*, pages 1–6.

Braisted, P. W. (1963). *Study of a Predictor for Remote Control Systems Operating with Signal Transmission Delays*. Department of Mechanical Engineering, Stanford University.

Cousins, S. (2012). Is ros good for robotics?[ros topics]. *IEEE Robotics & Automation Magazine*, 19(2):13–14.

Fankhauser, P., Bloesch, M., Rodriguez, D., Kaestner, R., Hutter, M., and Siegwart, R. (2015). Kinect v2 for mobile robot navigation: Evaluation and modeling. In *Advanced Robotics (ICAR), 2015 International Conference on*, pages 388–394. IEEE.

Gautham Ponnu, J. G. (2016). Real-Time ROSBerryPi SLAM Robot. A design project report, School of Electrical Computer Engineering of Cornell University.

Goebel, P. (2014). Pi robot. *Pi Robot*.

Goldberg, K. and Siegwart, R. (2002). *Beyond Webcams: an introduction to online robots*. MIT Press.

Goris, K. (2005). Autonomous mobile robot mechanical design. *VrijeUniversiteitBrussel, Engineering Degree Thesis, Brussels, Belgium*.

Grisetti, G., Stachniss, C., and Burgard, W. (2007). Improved techniques for grid mapping with rao-blackwellized particle filters. *IEEE transactions on Robotics*, 23(1):34–46.

Grisettiyz, G., Stachniss, C., and Burgard, W. (2005). Improving grid-based slam with rao-blackwellized particle filters by adaptive proposals and selective resampling. In *Robotics and Automation, 2005. ICRA 2005. Proceedings of the 2005 IEEE International Conference on*, pages 2432–2437. IEEE.

Helguero, F. P. (2016). Planificador de trayectorias de vehiculos Ã¡ereos no tripulados basado en algoritmos evolutivos multiobjetivo. Master's thesis, Universidad Complutense de Madrid - Universidad Nacional Abierta y a Distancia UNAD.

Hellström, T. (2011). *Kinematics equations for differential drive and articulated steering*. Department of Computing Science, Umeå University.

Joseph, L. (2015). *Mastering ROS for robotics programming*. Packt Publishing Ltd.

Kelly, A. (1994). An intelligent predictive controller for autonomous vehicles. Technical report, DTIC Document.

Monjazeb, A., Sasiadek, J. Z., and Necsulescu, D. (2011). Autonomous navigation among large number of nearby landmarks using fastslam and ekf-slam-a comparative study. In *Methods and Models in Automation and Robotics (MMAR), 2011 16th International Conference on*, pages 369–374. IEEE.

Montemerlo, M. and Thrun, S. (2003). Simultaneous localization and mapping with unknown data association using fastslam. In *Robotics and Automation, 2003. Proceedings. ICRA'03. IEEE International Conference on*, volume 2, pages 1985–1991. IEEE.

Nguyen, V., Harati, A., and Siegwart, R. (2007). A lightweight slam algorithm using orthogonal planes for indoor mobile robotics. In *Intelligent Robots and Systems, 2007. IROS 2007. IEEE/RSJ International Conference on*, pages 658–663. IEEE.

Omara, H. I. M. A. and Sahari, K. S. M. (2015). Indoor mapping using kinect and ros. In *Agents, Multi-Agent Systems and Robotics (ISAMSR), 2015 International Symposium on*, pages 110–116. IEEE.

Santos, J. M., Portugal, D., and Rocha, R. P. (2013). An evaluation of 2d slam techniques available in robot operating system. In *Safety, Security, and Rescue Robotics (SSRR), 2013 IEEE International Symposium on*, pages 1–6. IEEE.

Zaman, S., Slany, W., and Steinbauer, G. (2011). Ros-based mapping, localization and autonomous navigation using a pioneer 3-dx robot and their relevant issues. In *Electronics, Communications and Photonics Conference (SIECPC), 2011 Saudi International*, pages 1–5. IEEE.