UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

PROGRAMA DE MAESTRÍA Y DOCTORADO EN INGENIERÍA
MECÁNICA – MECATRÓNICA

ROBOT MÓVIL OMNIDIRECCIONAL PARA SLAM Y NAVEGACIÓN EN ROS

TESIS
QUE PARA OPTAR POR EL GRADO DE:
MAESTRO EN INGENIERÍA

PRESENTA:
JUAN DE DIOS FLORES MÉNDEZ

TUTOR PRINCIPAL
DR. VÍCTOR JAVIER GONZÁLEZ VILLELA
FACULTAD DE INGENIERÍA

MÉXICO, D. F. JUNIO 2015

JURADO ASIGNADO:

Presidente:	Dr. Cuenca Jiménez Francisco
Secretario:	Dr. Díaz Hernández Octavio
Vocal:	Dr. González Villela Víctor Javier
1 er. Suplente:	Dra. Corona Lira María Del Pilar
2 d o. Suplente:	Dr. Martínez Zamudio Patricio

Lugar o lugares donde se realizó la tesis: Universidad de Aalborg en Copenhague; Facultad de Ingeniería, UNAM.

TUTOR DE TESIS:

DR. VÍCTOR JAVIER GONZÁLEZ VILLELA

FIRMA

Agradecimientos

Al apoyo brindado por la DGAPA, UNAM, a través del proyecto PAPIIT IN117614: “Robótica intuitiva, adaptable, reactiva, hibrida y móvil aplicada al servicio, el rescate y la medicina”

A CONACYT por su apoyo al programa de Maestría.

Resumen

La robótica móvil es la rama de la robótica que estudia a los robots que tienen la capacidad de moverse en su entorno. En años recientes ha crecido el interés en el estudio de la robótica móvil por las ventajas que significa tener un robot que pueda desplazarse y desarrollar sus funciones, que no se encuentre necesariamente en un lugar fijo.

En el presente trabajo se muestra:

- El desarrollo e integración de un robot móvil omnidireccional en el Sistema Operativo Robótico. El Robot Móvil Omnidireccional está conformado por tres ruedas *mecanum,* dos al frente y una atrás.

- Se realiza su análisis cinemático directo e inverso para poder controlar cinemáticamente el robot y obtener una estimación de su posición relativa mediante un sistema de odometría basado en la velocidad medida de sus ruedas.
- Se analiza su maniobrabilidad y direccionabilidad a modo de comprobación de su omnidireccionalidad.
- Además se integra el control de los motores, la cinemática directa, interfaz de sensores y odometría con el microcontrolador Arduino al Sistema Operativo Robótico.
- Se realiza un sistema de tele operación con el mando Sixaxis de Playstation.
- Se implementan algoritmos de Mapeo y Localización Simultáneos y de Navegación y se realizan varias pruebas con el sistema.
- Finalmente se muestran los resultados obtenidos al realizar *SLAM* y Navegación con el robot.

Índice General

Índice de Tablas y Figuras

Capítulo 1. Introducción

1.1 Robótica

La robótica es la actividad intelectual y práctica que abarca el estudio sistemático de la aplicación del conocimiento científico asociado a la concepción, producción, teoría y aplicación de los robots.

La palabra robot proviene de la obra *Rossum's Univeral Robots* de Karel Capek, la cual se deriva de "*robota"* que significa trabajo duro en el idioma checo y lenguas eslavas.

Un robot es una máquina compuesta de sensores, una unidad de procesamiento y actuadores. Un robot capturará información de su ambiente, usará esta información para procesarla y realizará un trabajo. El instituto de Robótica de América (RIA por sus siglas en inglés) define a un robot como *"Un manipulador multifuncional reprogramable, diseñado para mover materiales, partes, herramientas o dispositivos, por medio de varios movimientos programados con el propósito de completar diferentes tareas. Un robot es un equipo que opera automáticamente, adaptable a las condiciones complejas del ambiente en el que opera, por medio de reprogramación, administración para prolongar, amplificar y reemplazar una o más funciones humanas en su interacción con el ambiente"*.

Los robots surgieron desde la antigüedad como entes mecánicos que pretendían imitar los movimientos humanos. Posteriormente los robots fueron diseñados para sustituir al hombre en la industria en tareas que son repetitivas. Al principio los robots realizaban tareas repetitivas e invariables porque su reprogramación representa un alto costo dado que solo un experto lo puede hacer.

La introducción de los robots en la industria representó un incremento en la producción y mejoría en la precisión y exactitud de los procesos donde se implementaron los sistemas robóticos. Más adelante fueron necesarios sistemas de manufactura flexible, es decir sistemas en la línea de producción que pudieran ser reprogramados fácil y rápidamente para adaptarse a la producción de productos con diferentes características o de baja producción. Las nuevas necesidades de producción han resultado en un incremento en el estudio de

robots que puedan ser reprogramados fácilmente y, últimamente, en el estudio de robots que puedan desplazarse a través de las líneas de producción.

1.1.1 Clasificación general de los robots

De acuerdo a su grado de movilidad los robots se clasifican en:

- Robots manipuladores cuya base posee un enlace mecánico con la base de referencia fija.
- Robots móviles. Son los robots que no poseen un enlace mecánico con la base de referencia fija.

a) b)

Figura 1. A) Robot Manipulador fijo KR6, imagen tomada de www.kuka-robotics.com, b) Robot Móvil Summit, imagen tomada de www.robotnik.es

1.2 Robótica Móvil

En años recientes el estudio de la robótica móvil se ha incrementado debido a la necesidad de crear máquinas capaces de moverse en su entorno y porque su invención es más reciente que los robots manipuladores [1]. Las aplicaciones de los robots móviles incluyen misiones de búsqueda, misiones de exploración, rescate de personas, automatización de la industria, vigilancia, etc.

Los robots móviles se pueden clasificar de acuerdo a diferentes criterios [2]:

- Por el ambiente en el que se desplazan:
  - Robots terrestres
  - Robots aéreos
  - Robots acuáticos

a) b) c)

Figura 2. a) Robot terrestre Tlaloc 2, imagen tomada de CNN México. b) Robot aéreo AR Drone II, imagen tomada de Parrot ®. c) Robot Acuático AquaJelly, imagen tomada de Festo Automation.

- Por su sistema de locomoción:
  - Robots con patas
  - Robots con llantas
  - Robots con orugas.
  - Etc.

a) b) c)

Figura 3. a) Robot SandFlea con ruedas, imagen tomada de Boston Dynamics. b) Legged Squad Support Systems, robot con patas, imagen tomada de Boston Dynamics, c) Robot Tanky con orugas, imagen tomada de Robotics Community.

- Por su nivel de autonomía:
  - Autónomo
  - Semi-autónomo
- Por su aplicación:
  - Robots educativos
  - Robots de servicio
  - Robots militares
  - Etc.

En particular los robots móviles con llantas se pueden clasificar como holonómicos o no holonómicos. Los robots móviles no holonómicos son los que poseen restricciones que limitan su movimiento, como por ejemplo un carro que no puede moverse lateralmente. Por otro lado, un robot holonómico es aquel que

no posee restricciones de movimiento, también conocido como robot omnidireccional porque puede moverse en cualquier dirección sin ninguna restricción, además de la que el medio supone.

Un robot móvil está conformado por cuatro grandes sistemas [3]:

1. El sistema mecánico
2. El sistema sensorial
3. El sistema de control
4. Fuente de energía.

Es por esto que la Mecatrónica es la rama de la ingeniería encargada de la construcción de estos robots, siendo la Mecatrónica definida como la sinergia de la ingeniería de precisión, control electrónico y sistemas mecánicos [4].

El sistema mecánico de un robot móvil es aquel que comprende el conjunto de elementos o dispositivos cuya función es la transmisión de movimiento desde su fuente de origen a través de transformación de energía [5]. Es el sistema encargado de la locomoción del robot, ya sea a través de llantas, patas, orugas, etc. Es el mecanismo que da movimiento al robot.

El sistema sensorial es el conjunto de sensores que permiten al robot percibir su ambiente. El sistema sensorial puede dividirse en sensores internos y sensores externos [6]. Los sensores internos, como su nombre lo indica, sirven para estimar el estado interno del robot; los sensores externos sirven para percibir el ambiente en el que el robot realiza su tarea.

Por otra parte, el sistema de control es el que se encarga de gobernar los movimientos y comportamientos del robot. A su vez, el sistema de control se puede separar en diferentes capas. La capa de control de bajo nivel es la que se encarga del control de cada uno de los actuadores; la capa intermedia de la coordinación de los actuadores y la de más alto nivel se encarga del control de comportamientos complejos del robot. Es importante mencionar el sistema de control depende directamente del sistema sensorial del robot.

Por último, la fuente de energía es el sistema que suministra la energía necesaria a los demás sistemas para que el robot pueda funcionar correctamente. Por lo general, los robots dependen de una fuente de poder eléctrica la cual

consiste en una batería y circuitos electrónicos de protección para la correcta alimentación eléctrica de los demás sistemas.

1.2.1 Aportación

El presente trabajo muestra el desarrollo de un robot móvil omnidireccional de tres ruedas *mecanum,* usadas como medio de tracción en una nueva configuración que no había sido usada antes.

Se desarrollan las ecuaciones cinemáticas que describen el movimiento del robot. Además, se realiza un diseño electrónico de control para el sistema sensorial y de locomoción del robot. Posteriormente, se hace su integración en el Sistema Operativo Robótico y se realizan pruebas de las paqueterías de SLAM y Navegación. El robot será utilizado para clases de robótica móvil y experimentación en robótica móvil en la Universidad de Aalborg-Copenhague, en Dinamarca.

1.3 Trabajos previos

En 2006, Jae-Bok y su colaborador realizaron el diseño de un robot móvil con cuatro llantas direccionables omnidireccionales. La capacidad de poder controlar la orientación de las llantas omnidireccionales le hace posible tener una mayor eficiencia energética.

Figura 4. Foto del Robot Móvil Omnidireccional con ruedas direccionables. Tomada de Jae-Bok Song 2006.

Giovanni Indiveri y colaboradores, presentaron en 2006 el control de un robot, de tres llantas suecas en una configuración simétrica, considerando la

saturación de los motores para poder concretar trayectorias viables tomando en cuenta las propiedades físicas de sus actuadores.

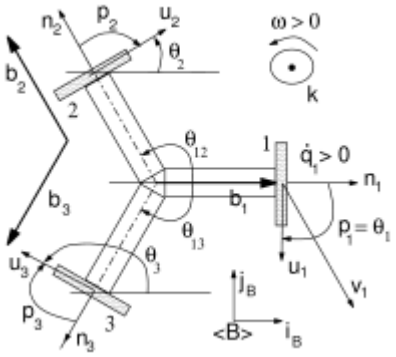


Figura 5. Esquema del robot móvil omnidireccional. Esquema tomado de Indivieri, 2006.

González-Villela, a través de su Tesis Doctoral en 2006, publica una serie de ecuaciones en una teoría unificada para el análisis cinemático y dinámico de robots móviles. Estas ecuaciones permiten definir por completo la matriz Jacobiana, que es la relación matemática entre el movimiento de las ruedas del robot y el movimiento del robot.

En 2009, el grupo de Kyung-Lyong Han desarrolla un robot móvil omnidireccional de cuatro llantas *mecanum* con un control con ganancias basadas en la lógica difusa.

En 2011, Safdar Zaman y colaboradores estudian los algoritmos de Mapeo y Localización y Navegación Autónoma en el Sistema Operativo Robótico con un robot Pioneer 3-DX, un robot de tracción diferencial. Encontraron que diferentes factores como la forma de los objetos y la diferencia de superficies afecta en el proceso de mapeo.

En 2012, Ayrton Oliver y colaboradores usan el Kinect como un sensor de navegación para robots móviles, en comparación con un escáner laser que se usa generalmente para este fin. Su sistema se basa en el Sistema Operativo Robótico. El resultado de su estudio fue determinar que es posible usar el Kinect como un sensor de navegación y mapeo.

Figura 6. Robot con sensor Kinect para navegación. Imagen tomada de A. Oliver, 2012.

André Gonçalves Araújo presenta, en 2012, la integración de un robot móvil de tracción diferencial en el Sistema Operativo Robótico. El robot es de bajo costo y, como describe en su trabajo, es un sistema listo para usarse rápidamente (*off-the-shelf solution* en inglés).

1.4 Planteamiento del problema

En el laboratorio de Robótica, Visión e Inteligencia Artificial de la Universidad de Aalborg-Copenhague se ha construido un robot móvil omnidireccional de llantas suecas. El robot cuenta con una batería, una motherboard equipada con un procesador Intel Atom y 4 GB de memoria RAM, una tarjeta con microcontrolador Arduino para controlar los motores y leer los encoders de los motores. Este robot no está programado ni configurado. El uso final del robot será para impartir una clase sobre robótica móvil en la Universidad de Aalborg. Su ambiente será un ambiente estructurado, de piso uniforme sin pendientes. El sistema del robot deberá funcionar en el Sistema Operativo Robótico debido a que es la plataforma de software en el que funcionan y se desarrollan las demás paqueterías de software en el laboratorio antes mencionado.

En este trabajo se presenta

1.4.1 Hipótesis

La nueva configuración es omnidireccional y es capaz de ser implementado en el Sistema Operativo Robótico para tareas de SLAM y navegación. La hipótesis será corroborada mediante experimentos en el robot.

1.5 Objetivos

1.5.1 Objetivo General

Desarrollo del robot móvil omnidireccional en el Sistema Operativo Robótico.

1.5.2 Objetivos Específicos

1. Modelado de las ecuaciones cinemáticas del robot móvil omnidireccional.
2. Integración del sistema electrónico y mecánico del robot.
3. Implementación de las ecuaciones cinemáticas en el robot.
4. Implementación de los algoritmos de SLAM y Navegación Autónoma en el robot.
5. Análisis de resultados.

1.6 Delineamiento de la tesis

Este **Capítulo I** describe el objetivo general de la tesis y da una pequeña introducción a la robótica móvil, robots móviles omnidireccionales y una visión general al estado del arte de los robos móviles omnidireccionales, se define el problema, la hipótesis y los objetivos generales y particulares del proyecto.

La descripción a detalle del robot móvil omnidireccional se lleva a cabo en el **Capítulo II.** El diseño mecánico, selección de componentes, diseño electrónico son descritos a detalles y se ofrece un presupuesto preliminar para la construcción del robot a un bajo costo.

En el **Capítulo III** se desarrolla el análisis cinemático del robot. Los grados de libertad, maniobrabilidad y direccionabilidad son calculados. Se obtiene la cinemática inversa y directa, los cuales son necesarios para el desarrollo de un control de lazo abierto y cerrado y de la estimación de su posición, respectivamente.

El Sistema Operativo Robótico se presenta en el **Capítulo IV.** En este capítulo también se presenta la programación del micro controlador basado en el sistema *Arduino* que realiza un control de los motores, interfaz con los *encoders*, telémetro ultrasónico y el Sistema de Medición Inercial (*IMU – Inertial Measurement Unit*) y

estimación de la posición del robot basado en su odómetro. Además, se presenta la integración de una cámara de color y profundidad como sensor de mapeo y navegación. Se configura una descripción de los eslabones del robot y un emisor de transformación del estado del robot. Se describe también un sistema de teleoperación con el mando Dual Shock 3 de Playstation. Finalmente, se configura un acceso WiFi Hotspot y un acceso remoto SSH.

En el **Capítulo V** se hace una breve descripción de la Localización y Mapeo Simultaneo (*SLAM* por sus siglas en inglés). También se detalla la determinación de los parámetros del algoritmo de *SLAM* para el robot omnidireccional.

Se explican brevemente algoritmos de navegación basados en mapas de cuadriculas en el **Capítulo VI.** Se especifican también los parámetros usados para la navegación para el robot omnidireccional.

Se realizan pruebas y se recaban los resultados en el **Capítulo VII.** Se crea un mapa de uno de los pasillos de la Universidad de Aalborg en Copenhague. Posteriormente se usa el mapa para navegar y evadir obstáculos.

En el **Capítulo VIII** se presentan las conclusiones y el trabajo a futuro. Finalmente, se presentan los apéndices con el código fuente del programa del robot y se presentan las referencias consultadas.

Capítulo 2. Robot Móvil Omnidireccional

2.1 Introducción

En este capítulo se describe al lector el diseño y construcción del robot móvil omnidireccional. El robot fue diseñado y construido mecánicamente, pero estaba incompleta su adecuada programación y su desarrollo cinemático, los cuales serán descritos en capítulos posteriores. El robot fue diseñado con el fin de usar el Sistema Operativo Robótico como plataforma de programación y funcionamiento. Dentro de su diseño también se considera un bajo costo para su construcción. Aunque el robot estaba ya construido, se señalan los criterios de su diseño, además de que se diseñaron soportes para componentes adicionales, los cuales fueron manufacturados en una impresora 3D. Es importante saber los criterios de selección de componentes porque es información importante que se tiene que considerar para realizar una programación adecuada.

2.2 Hardware del Robot Móvil Omnidireccional

El Robot Móvil Omnidireccional fue diseñado para poder ser utilizado en el Sistema Operativo Robótico y con la intencionalidad de que además tenga suficiente poder de procesamiento para poder realizar diversas tareas de manera autónoma sin tener que depender estrictamente de una estación de trabajo que realice todo el procesamiento.

El fin último del Robot Móvil Omnidireccional es ser una plataforma para la enseñanza sobre robótica móvil, es por eso que no necesita ser de grandes dimensiones. Se eligieron las ruedas *mecanum* de la empresa Vex Robotics porque es una empresa que se especializa en fabricar componentes para robots educativos, provee de las conexiones necesarias para acoplarlo al motor así como los motores para mover las ruedas, además de suplir los elementos mecánicos necesarios para montar los motores sobre el chasis del robot.

Figura 7. Rueda Mecanum de la marca Vex Robotics, imagen tomada de la web oficial de Vex Robotics, 2015.

Los actuadores elegidos fueron los motores de dos cables modelo 269 de la marca Vex Robotics. Estos motores tienen la característica de tener una caja de engranes de aleación de metal que hace que no sea necesario el reemplazo de los engranes. El fabricante especifica que si se quiere controlar los motores con algún micro controlador diferente al que ellos proveen es necesario conectarlo a un módulo adicional.

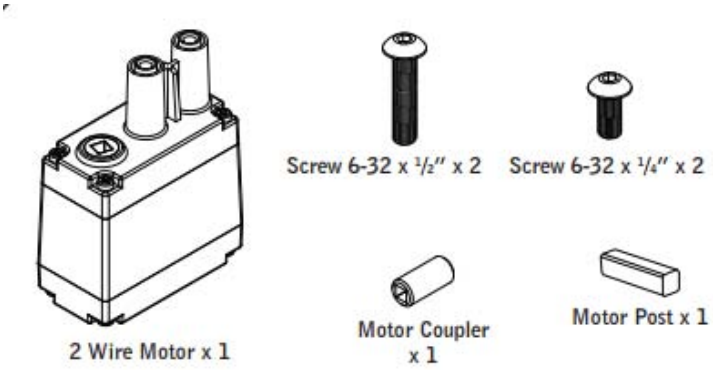


Figura 8. Contenido del paquete de motor 269, imagen tomada de la web oficial de Vex Robotics, 2015.

Las especificaciones, como las marca el proveedor, a 7.2 volts son:

Descripción	Especificación
Torque a rotor bloqueado	8.7 in-lb [0.97 N-m]
Velocidad a rotor libre	100 RPM
Corriente a rotor bloqueado	2.6 Amps
Corriente a rotor libre	0.18 Amps

Tabla 1. Especificaciones del motor 269 VEX Robotics, 2015.

Para poder tener un mayor control sobre la velocidad de los motores el proveedor también ofrece un sensor de velocidad. El sensor de velocidad es un *encoder* óptico diseñado para el motor 269 de Vex Robotics. El *encoder* tiene integrada la electrónica de acondicionamiento de señal y tiene como salida una comunicación I2C para comunicarse con el micro controlador.

Figura 9. Modulo Integrado de Encoder para el Motor 269, imagen tomada de la web oficial de Vex Robotics, 2015.

La resolución del *encoder* es de 240.448 pulsos por revolución, por lo que se puede calcular que:

$$\left(\frac{240.488\ pulsos}{1\ rev} * \frac{1\ rev}{360\ grados}\right)^{-1} = 1.49\ {}^{\circ}/_{pulso}$$

Por lo que tiene una resolución de 1.49 grados geométricos.

El módulo adicional para poder controlar el motor con un micro controlador es el *Motor Controller 29.* La característica principal de este controlador es que se conecta a un motor de dos cables de alimentación y tiene como salida tres cables: Voltaje, Tierra y Señal de control por Modulación de Ancho de Pulso (PWM por sus siglas en inglés).

Figura 10. Motor controller 29. Imagen tomada de la web oficial de Vex Robotics.

Las especificaciones, como la web oficial de Vex Robotics lo indica, son las siguientes:

- Cada *Vex Motor Controller 29* puede manejar un motor de dos cables.
- Corriente máxima: 4 A
- Voltaje máximo de alimentación: 8.5V
- Entrada de PWM: 1-2 ms ejecutará máxima reversa a máximo avance, 1.5 ms es neutral.

- Cables de entrada: Cable Negro – Tierra, Cable Naranja – Voltaje, Cable Blanco – Señal de PWM.
- Cables de salida: Cables Rojo y Negro – Conexión a Motor.

(Vex Robotics, 2015)

Se puede observar en las especificaciones que este controlador de motores es un controlador de lazo abierto porque no tiene una señal de retroalimentación. Si se usa en conjunto con el encoder es posible tener un controlador de lazo cerrado, por eso el proveedor ofrece un micro controlador para realizar esa tarea pero también se puede implementar un controlador usando un micro controlador diferente.

2.2.1 Soportes, Ejes y conexiones

Los siguientes componentes se adquirieron también con el proveedor Vex Robotics. Estos elementos son parte de los soportes, ejes, cojinetes y conexiones necesarias para sujetar los motores al chasis del robot y las ruedas omnidireccionales.

Elemento mecánico	Imagen	Material	Uso
Ángulo de acero		Acero galvanizado	Usado para realizar marcos estructurales. Puede ser cortado en incrementos de 0.5”.
Soporte plano		Acetal	Es usado como elemento de conexión entre el motor y el ángulo de acero. Ofrece una baja fricción con el eje que pasa perpendicularmente por su estructura.
Soporte del eje con prisionero (Collarín del eje)		Acero	Este elemento es usado para evitar deslizamientos longitudinales del eje.
Eje		Acero y placas de Zinc	Es el eje que conecta el motor con la rueda. Su forma con bordes redondeados sirve para que gire fácilmente en un agujero redondo, mientras esta aprisionado en un agujero cuadrado.

Tabla 2. Elementos mecánicos de montaje de la marca VEX Robotics, imágenes y especificaciones tomadas de la web oficial de VEX Robotics, 2015.

2.2.2 CPU

El robot cuenta con una computadora a bordo con el fin de ejecutar el Sistema Operativo Robótico. Se elige una tarjeta madre con CPU integrado y de formato ITX. El formato ITX es un formato más pequeño que el formato estándar ATX, por lo que es más práctico para usarse en un robot en el que se busca mantener las dimensiones más reducidas posibles. En el formato ITX se definen las dimensiones de 170 mm x 170 mm y compatibilidad con las especificaciones eléctricas y de componentes del formato estándar ATX cuyas dimensiones son 305 mm x 244 mm.

El modelo elegido es ZOTAC D2550-ITX Supreme B-series ya que cuenta con una tarjeta gráfica integrada que es útil para las aplicaciones que requieren el manejo de gráficos en el robot.

Las especificaciones de ZOTAC D2550-ITX son:

Product Name		
	Product Name	D2550-ITX WiFi Supreme B-series
Model		
	Model	D2550ITXS-B-E
Chipset		
	Manufacturer	Intel
	Chipset	NM10 Express
	GPU	NVIDIA GeForce GT 610 w/512MB DDR3
CPU Compatibility		
	Name	
	Socket	NA (integrated)
	Bus	N/A
Memory		
	Memory Type	DDR3
	Memory Speed	1066 MHz
	Slots	2 204-pin SO-DIMM
	Capacity	Up to 4GB
CPU and Memory Compatibility List		
	CPU and Memory Compatibility List	Intel Atom D2550 (1.86 GHz dual-core)
3D API		
	DirectX	DirectX11
	OpenGL	OpenGL 4.1
	Slots	
Expansion		
	Slots	

Networking		
	Ethernet	10/100/1000Mbps
	WiFi	802.11n & Bluetooth 3.0
Audio		
	Analog	8-ch HD
	Digital	S/PDIF + HDMI
Ports		
	DVI	1
	HDMI	1 (HDMI 1.4a)
	DisplayPort	1
	SATA	2 (SATA 3.0 Gb/s)
	eSATA	0
	PATA	
	PS/2	
	Serial Port	Serial port header
	USB Ports	2 USB 3.0 (back panel) 6 USB 2.0 (4 on back panel 2 on pin header)
	FireWire Ports	
Cooler		
	Cooler	Active (with fan)
Form Factor		
	Form Factor	
OS Compatibility		
	Windows	Windows 7 ready
General		
	SLI Supported	
	Maximum Resolution	
Other		
	Wake-On Support	
	Overclocking features	

Figura 11. Especificaciones ZOTAC D2550-ITX. ZOTAC Product Description, 2014.

La tarjeta madre fue complementada con un disco duro de estado sólido y dos memorias RAM de 2 GB cada una. El disco duro es marca Kingston de 60 GB, el cual contendrá el sistema operativo y almacenará los programas desarrollados para el robot.

Figura 12. Disco duro de estado sólido marca Kingston de 60GB. Imagen tomada de http://www.cyberpuerta.mx/

2.2.3 Plataforma Arduino

Arduino es una plataforma de fuente abierta (*open-source* en inglés) de electrónica conformada por una tarjeta de desarrollo con un micro controlador y un software libre de programación C++. Arduino surge como una solución de bajo costo y de fácil acceso para hobbyistas, artistas y personas interesadas en aprender sobre electrónica y robótica. Arduino se ha convertido en una solución rápida y eficaz para proyectos relacionados con electrónica, control de robots e interfaz con otros dispositivos y la computadora.

Se eligió la tarjeta Arduino DUE para el control de los motores e interfaz con los sensores que no tienen conexión directa con la computadora. El Arduino DUE es la única tarjeta Arduino que posee un procesador de 32 bits de nucleo ARM, lo que lo convierte en una solución apropiada para un robot móvil que debe mantener seguimiento de su posición basado en el cálculo de su cinemática inversa en un intervalo de tiempo regular, sin sufrir de letargos debido a los cálculos de punto flotante. La tarjeta de desarrollo Arduino DUE cuenta con 54 salidas y entradas digitales con 12 PWM, 12 entradas analógicas, 2 DAC, 2 TWI, un adaptador Jack de alimentación, un header JTAG y botones de reinicio y borrado.

Las principales ventajas que ofrece el Arduino DUE, con respecto a las otras tarjetas desarrollo Arduino como el Arduino UNO o el Arduino Mega de 8 bits, es el procesador de 32 bits que permite operaciones de datos de cuatro bytes en un solo ciclo de reloj, una velocidad de CPU de 84MHz, 96 KBytes de SRAM disponible en el tiempo de ejecución, 512Kbytes de memoria flash para el programa y control de acceso directo a la memoria (*Direct Memory Access*) para aligerar el procesamiento del CPU en tareas intensivas.

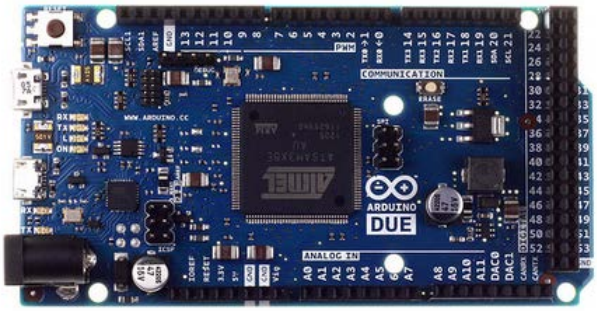


Figura 13. Arduino DUE, imagen tomada de www.arduino.cc

2.2.4 Sensores

El sensor principal del robot es una cámara Asus Xtion Pro Live. La cámara Asus Xtion Pro Live es una cámara RGB y de profundidad con características similares al sensor Kinect de Microsoft.

Las características, como se listan en la página oficial del producto, son:

Figura 14. Cámara Asus Xtion Pro Live. Imagen tomada de la web oficial de ASUS, 2015.

Descripción	Especificación
Consumo de energía	Inferior a 2.5 W
Distancia de uso	Entre 0.8 m y 3.5 m
Campo de visión	58° H, 45° V, 70° D (Horizontal, Vertical, Diagonal)
Sensor	RGB y Profundidad
Tamaño de imagen de profundidad:	VGA (640x480): 30 fps; QVGA (320x240): 60 fps
Resolución	SXGA (1280*1024)
Plataforma	Intel X86 & AMD
Soporte OS	Win 32/64: XP, Win7; Linux Ubuntu 10.10: X86,32/64 bit; Android (bajo petición)
Interfaz	USB2.0
Software	Kit de desarrollo de software (OPEN NI SDK bundled)
Lenguaje de programación	C++/C# (Windows); C++ (Linux); Java
Ambiente de Operación	Interiores
Dimensiones	18 x 3.5 x 5 pulgadas

Tabla 3. Especificaciones de la cámara Asus Xtion Pro Live, tomadas de la web oficial de ASUS, 2014.

Los sensores secundarios son un sensor ultrasónico, el cual es colocado en la parte frontal del robot y una Unidad de Medición Inercial colocado en el interior del robot.

El sensor ultrasónico es el modelo HC-SR04 para proyectos básicos de electrónica. Se coloca en la parte frontal del robot, donde la cámara no puede registrar el ambiente y el sensor puede encontrar obstáculos u objetos.

El sensor ultrasónico es capaz de medir la presencia de un obstáculo en un rango de 2 cm – 400 cm sin contacto, con una precisión de hasta 3 mm. El principio básico de funcionamiento es como sigue:

- Se activa con una señal de gatillo de 10us en estado alto.
- El módulo activa automáticamente una señal ultrasónica de 40KHz y mide si hay una señal de regreso.
- Si hay una señal de regreso, a través de una señal de alto nivel, el tiempo del pulso alto de salida es directamente proporcional a la distancia que se encuentra el obstáculo u objeto.

(ELEC Freaks)

Figura 15.. Sensor HC-SR04. Imagen tomada de ELEC Freaks.

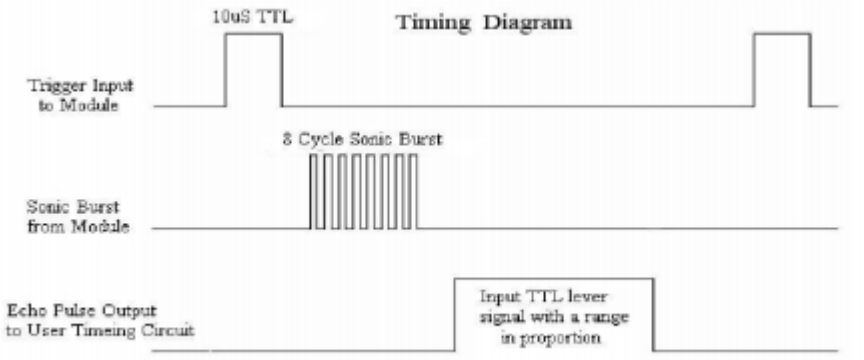


Figura 16. Esquema de tiempos requeridos para la medición de presencia de obstáculos. Imagen tomada de ELEC Freaks.

La Unidad de Medición Inercial es el modelo MPU6050 que está compuesto por un giroscopio y un acelerómetro.

Las características del giroscopio son:

- Salida Digital de los ejes X, Y, y Z de velocidad angular con programación de escala completa ±250, ±500, ±1000, y ±2000°/seg.
- Sincronización externa en el pin FSYNC.
- ADC integrado de 16 bits que permite un muestreo simultáneo de giros.
- Estabilidad mejorada con sensibilidad de temperatura.
- Desempeño mejorado de ruido a bajas frecuencias.
- Filtro pasa bajas digital programable.
- Corriente de operación del giroscopio: 3.6mA.
- Corriente en espera: 5µA.
- Calibración de fábrica.
- User self-test

Las características del acelerómetro son:

- Salida digital de tres ejes con escala completa programable de ±2g, ±4g, ±8g y ±16g.
- ADC integrado de 16 bits que permite un muestreo simultáneo de los acelerómetros.
- Corriente de Operación: 500µA.
- Detección de orientación.
- Detección de golpe.
- Interrupciones programables.
- Interrupción de alto G.
- User self-test.

Para poder obtener las mediciones del IMU es necesario realizar una conexión I2C con el dispositivo, el cual ya tiene el acondicionamiento de señal integrado. El MPU6050 también posee un módulo de Procesamiento Digital de Movimiento (Digital Motion Processing) para relevar de procesamiento al micro controlador maestro. El MPU6050 es usado para mejorar la estimación de la posición angular del robot móvil omnidireccional.

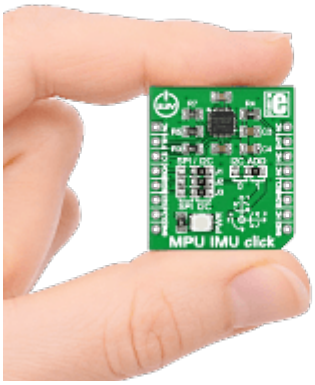


Figura 17. MPU6050 en una placa desarrollada por Mikroelektronika, imagen tomada de www.mikroe.com

2.2.5 Fuente de Energía

Una de las partes más importantes del robot es la fuente de energía o fuente de poder. La fuente de poder, como se mencionó anteriormente, es la que se encarga de distribuir la energía que necesitan los demás sistemas para funcionar correctamente. En el caso de robótica móvil se necesita una fuente de energía transportable, que por facilidad se decidió que sería una batería eléctrica.

Se selecciona un convertidor de corriente directa a corriente directa para reducir el voltaje de alimentación para los motores, con un voltaje de salida de 7V. El convertidor es de la marca Castle Creations'. Las especificaciones del convertidor son las siguientes:

Figura 18. CC BEC convertidor DC-DC, imagen tomada de www.robotmarketplace.com

Descripción	Especificación
Corriente pico de salida	10 Amps
Corriente continua de salida	@ 12V entrada – 7 Amps

	@ 24V entrada – 5 Amps
Voltaje de Salida	5.2V, se puede regular con un adaptador Castle Link
Voltaje de Entrada	5V a 25.2V
Longitud	30mm
Ancho	15mm
Alto	10mm
Peso	11 gramos

Tabla 4. Especificaciones del convertido DC-DC BEC 10 Amp peak, tomados de www.robotmarketplace.com

El robot requiere de tener una autonomía de por lo menos una hora y media porque es un prototipo para experimentos en robótica móvil.

La selección de la batería se realiza mediante un cálculo de la potencia total que necesita el robot para funcionar y se fija un tiempo de uso de por lo menos una hora y media para el análisis. A continuación se presenta el cálculo de la potencia necesaria que necesita el robot para funcionar.

	A-factor	V	I	W
Computadora	1.0	12.0	5.000	60.0
Motores	3.0	7.2	1.500	32.4
Arduino	1.0	5.0	0.500	2.5
Asus Xtion	1.0	5.0	0.500	2.5
DC-DC conv.	1.0	12.0	0.200	2.4
Perdidas Conv.	1.0	19.0	0.1	1.9
Total				101.7

Tabla 5. Tabla de cálculo de Potencia total que necesita el robot para funcionar en un instante dado.

Por lo que se calcula una potencia de 101.7 W. El siguiente parámetro necesario para determinar la batería que se usará es el tiempo necesario de uso del robot, que es de 1.5 horas, por lo que la energía necesaria que necesita tener la pila es:

$$101.7\ [w] * 1.5\ [h] = 152.55\ [Wh]$$

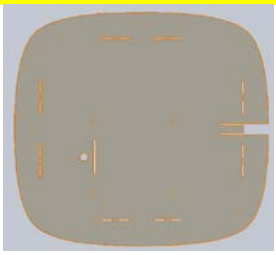
Una vez calculada la energía mínima necesaria que necesita la batería se selecciona una batería de 153 Wh. La batería tiene dos voltajes de salida

seleccionables y cuenta con un puerto de entrada para cargar la batería, uno de salida con voltaje seleccionable de 16 o 19 V y otro de salida para cargar dispositivos por medio de un puerto USB a 5 V. La batería también cuenta con cuatro luces LED para indicar el estado de carga de la batería. Sus dimensiones son 270.0 x 145.0 x 32.55 mm y un peso de 1340 g.

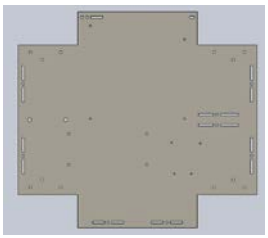
Figura 19. Batería externa de 153 Wh, imagen tomada de www.amazon.es

2.2.6 Chasis del Robot Móvil

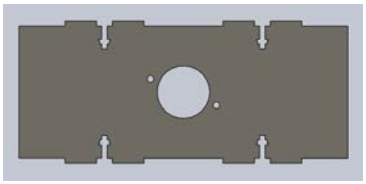
El chasis del Robot Móvil fue diseñado para tener una estructura de prisma rectangular, con una pala en la parte frontal para tareas de manipulación de objetos mediante la acción de empujar y una base en la parte posterior para sostener la cámara que funciona como sensor principal del robot. Las placas laterales fueron diseñadas para mostrar las conexiones de la placa madre de la computadora y el otro lado para contener el interruptor de encendido general, el botón de encendido de la computadora y los LEDs indicadores del robot.

Parte	Imagen
Placa superior	

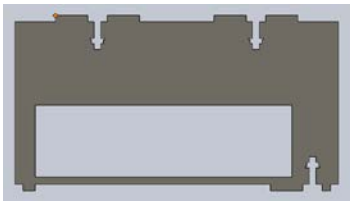
Placa inferior



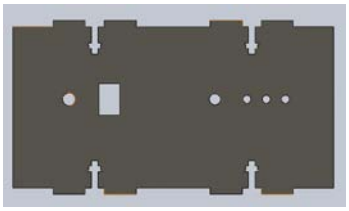
Placa frontal



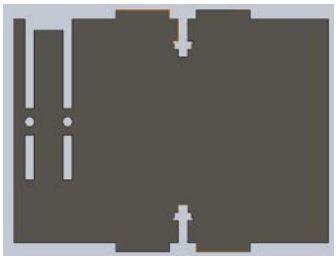
Placa lateral derecha



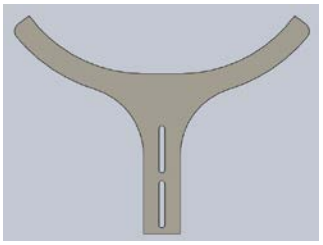
Placa lateral izquierda



Placa posterior



Pala



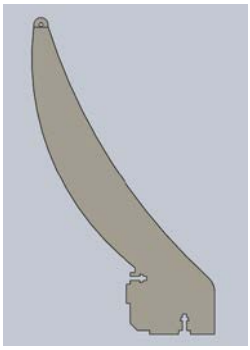
Placa de soporte de la cámara	

Tabla 6. Partes que conforman el chasis del robot.

Las placas del chasis del robot fueron manufacturadas por medio de corte laser en placas de madera de 4 mm de ancho. Se usaron tornillos de cabeza hexagonal ISO 4762 M4 20 y tuercas ISO 4035 M4 N.

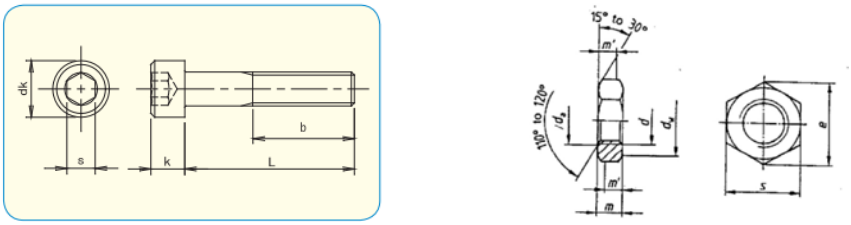


Figura 20. Descripción de las dimensiones de los tornillos y tuercas usadas en el chasis del robot, imágenes tomadas de la documentación ISO.

2.2.7 Soportes adicionales

Después del diseño inicial del robot se necesitaron soportes adicionales para mantener los cables en su lugar, instalar el sensor ultrasónico y dar rigidez al soporte de la cámara. Para fabricar o manufacturar los soportes adicionales se dispone de una impresora 3D de la compañía Makerbot. Para el diseño para la manufactura de partes en una impresora 3D de extrusión de plástico se tienen que tomar en consideración tres factores importantes [25]:

- Los objetos que tienen salientes de más de 45° necesitan material de soporte. Estos pueden ser añadidos por el diseñador o por el software de impresión de la impresora.
- Diseñar evitando usar material de soporte. El material de soporte puede dejar marcas en los objetos impresos cuando son removidos.
- Acomodar los objetos con la mejor orientación para la impresión.

2.2.7.1 Soporte para cables

Se diseñó un soporte adicional para sujetar los cables de la cámara y el Arduino, los cuales son conectados al costado del robot donde se encuentra expuesta la parte lateral de la placa madre de la computadora interna. El soporte de cables se empotra a presión en la placa superior del chasis del robot.

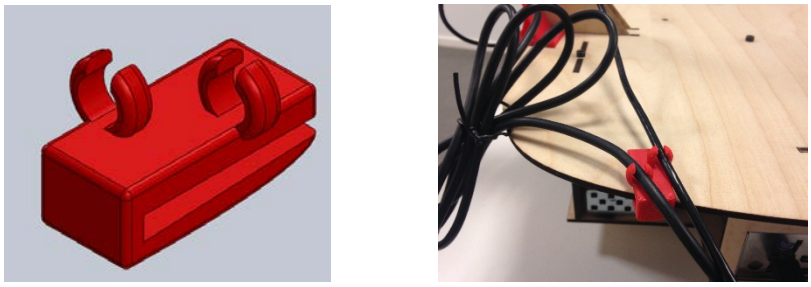


Figura 21. Soporte de cables, izquierda diseño modelo sólido, derecha impresión en 3D instalado en el robot.

2.2.7.2 Soporte para el sensor ultrasónico

Para poder añadir el sensor ultrasónico en la parte frontal del robot fue necesario diseñar un soporte para poderlo sujetar apropiadamente al robot. El soporte se ensambla a la pala del robot con el uso de dos tornillos y dos tuercas.

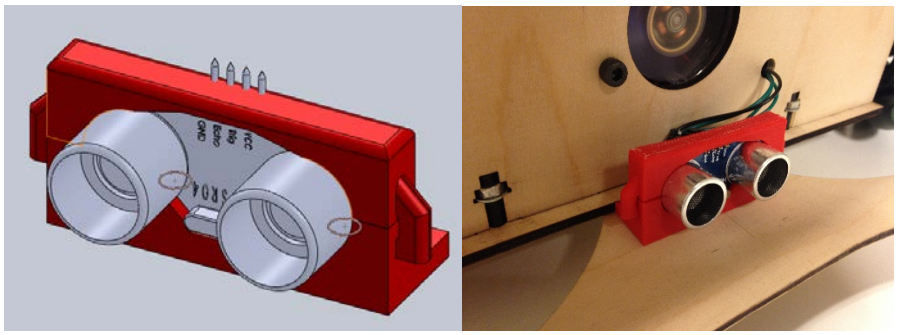


Figura 22. Soporte del sensor ultrasónico, izquierda diseño en modelo sólido, derecha impresión 3D e instalado en el robot.

2.2.7.3 Soporte para las placas de sujeción de la cámara

Posteriormente, cuando se movió el robot manualmente, se notó que la parte que sostenía la cámara era muy inestable por una carencia de soporte con las otras piezas del chasis. Se diseñó entonces un soporte, de cuatro piezas, para sostener adecuadamente el soporte de la cámara con el resto del chasis.

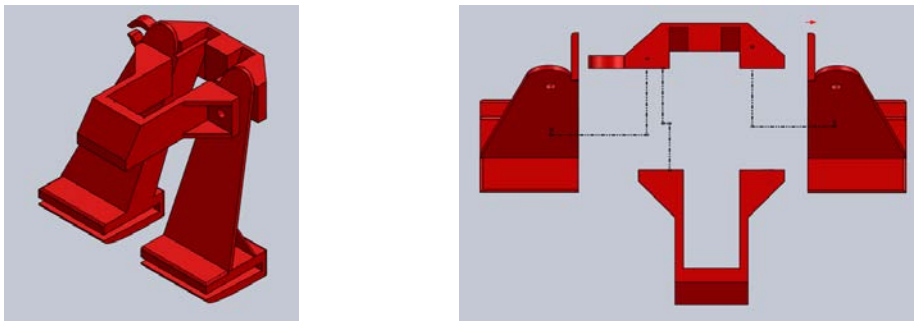


Figura 23. Soporte para el soporte de la cámara. Ensamble y explosivo vista superior.

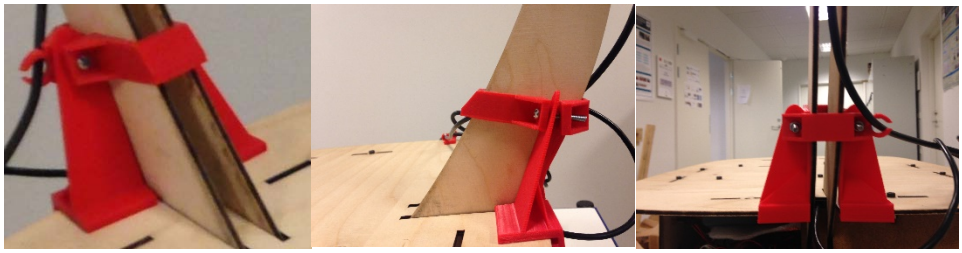


Figura 24. Impresión en 3D del soporte, instalado en el robot.

2.3 Ensamble

A continuación se muestra el ensamble final del robot móvil omnidireccional. El ensamble es una modificación del ensamble original realizado por Ph.D. Mikkel Rath Pedersen, en la Universidad de Aalborg en Copenhague.

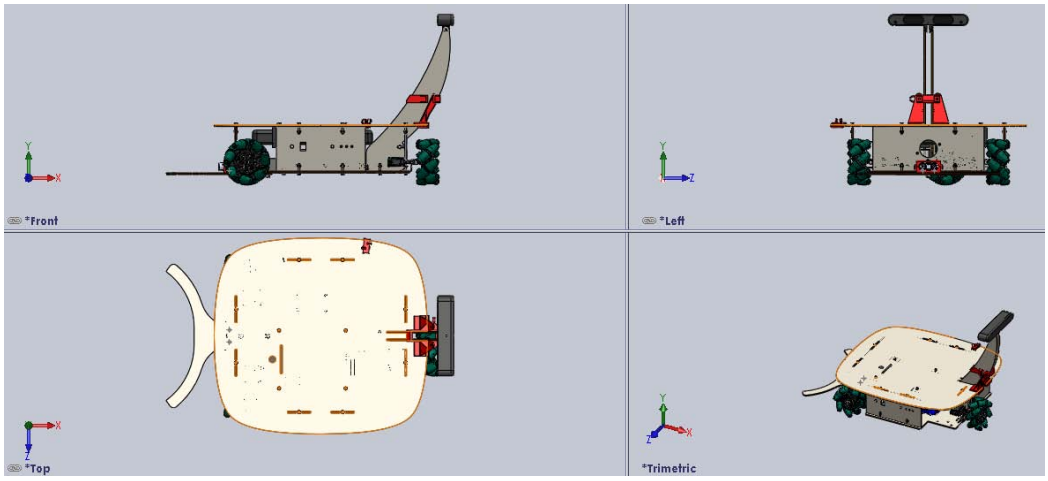


Figura 25. Ensamble del Robot Móvil Omnidireccional.

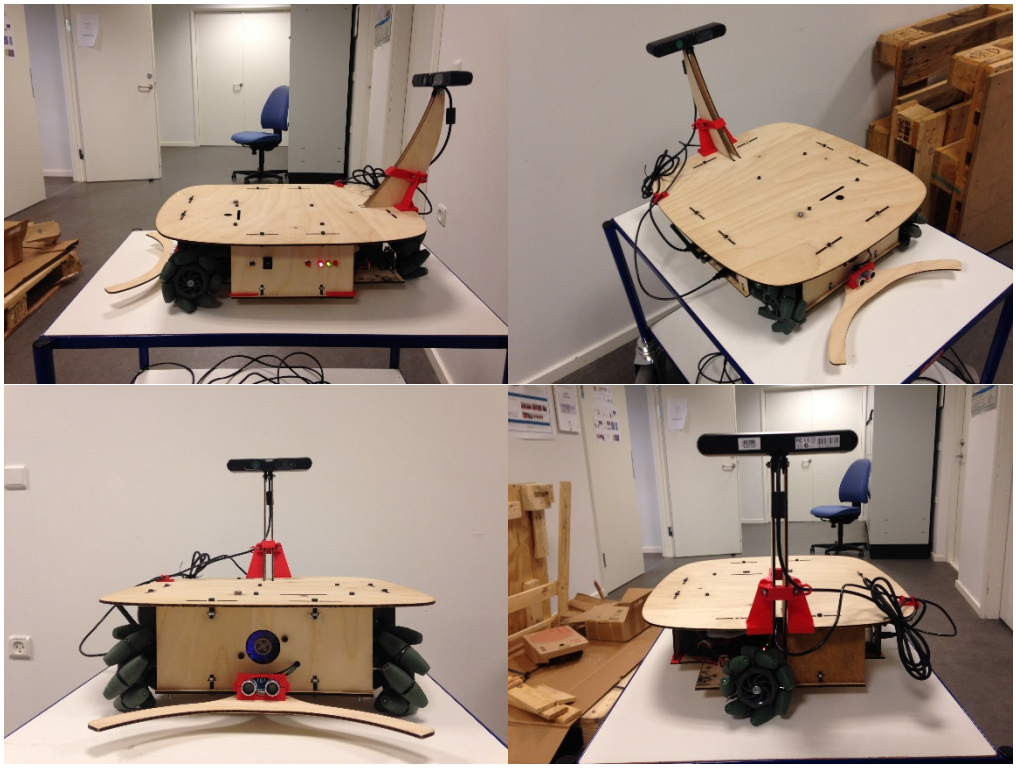


Figura 26. Fotos del Robot Móvil Omnidireccional

Se muestra la visión en explosivo del Robot Móvil Omnidireccional.

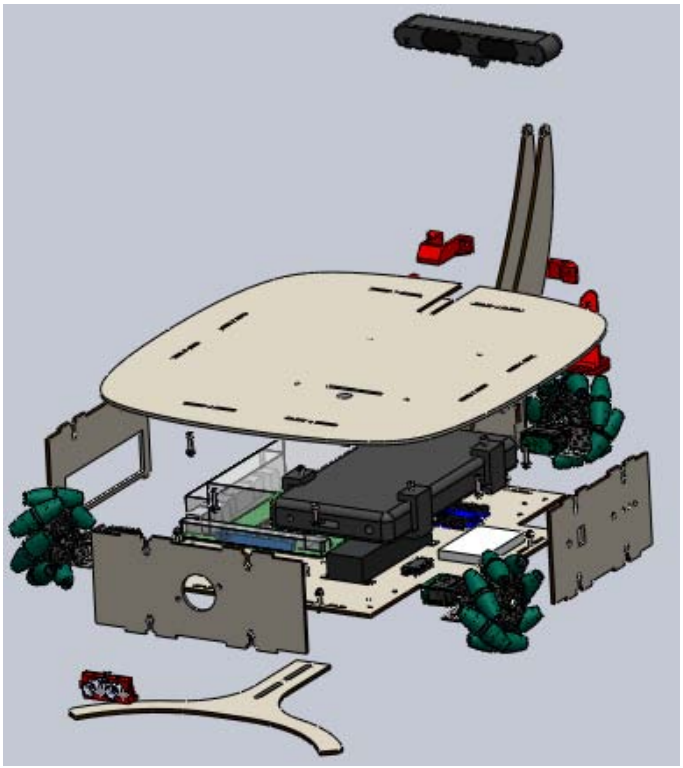


Figura 27. Explosivo del Robot Móvil Omnidireccional

Figura 28. Foto del Robot Móvil Omnidireccional ensamblado.

2.4 Diseño electrónico

El diseño electrónico estaba incompleto dado que solo consideraba el control y lectura de *encoders,* pero no estaba considerado el uso de un sensor ultrasónico ni de una Unidad de Medición Inercial. El Arduino DUE, en comparación con las demás tarjetas de desarrollo Arduino, se maneja con un voltaje de alimentación de 3.3 V y los pines de entrada y salida no soportan un voltaje mayor a 3.3 V por riesgo de que la tarjeta electrónica se dañe. Los encoders integrados en los motores se requieren un voltaje de alimentación de 5 V por lo que se asume que sus líneas de comunicación I2C requieren que sean conectadas por medio de una resistencia *"pull up"* a una línea de 5 V, como lo marca la documentación de comunicación I2C [19]. Para poder conectar una línea de comunicación I2C de 5 V a una línea de comunicación I2C de 3.3 V se requiere de un circuito integrado de cambio de nivel. El circuito integrado PCA9512AD de montaje superficial es un componente electrónico diseñado para conmutar simultáneamente los diferentes voltajes de entrada y de salida, bidireccionalmente, de una comunicación I2C [20] y soporta una velocidad de hasta 400 kHz. Este circuito integrado es usado para

conectar la línea de comunicación I2C de los *encoders* a la línea de comunicación I2C de la tarjeta Arduino DUE.

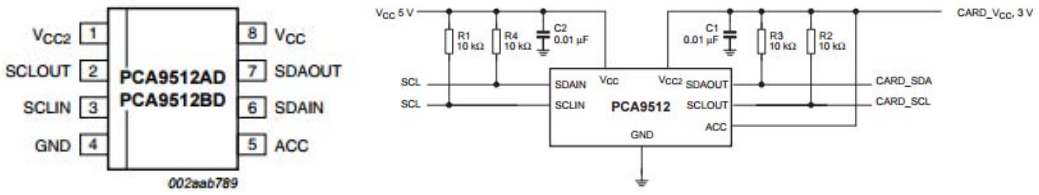


Figura 29. Diagrama de pines del circuito integrado PCA9512AD la izquierda y circuito de aplicación común a la derecha, tomado de la hoja de especificaciones de NXP.

La Unidad de Medición Inercial requiere un voltaje de alimentación de 3.3 V, y su hoja de especificaciones [21] marca un voltaje de operación de 3.3 V para la comunicación I2C, por lo que no requiere de ningún elemento adicional para poder conectarse a la tarjeta de desarrollo Arduino DUE.

El sensor ultrasónico HCSR04 requiere de un voltaje de operación de 5 V. Se ha demostrado en diferentes eventos que una señal de 3.3 V es suficiente para poder activar el sensor ultrasónico por lo que no se requiere ninguna interconexión entre el pin de activación del sensor con el Arduino DUE. La señal de retorno del eco es de 5 V, por lo que es necesario realizar cambio de nivel de esta señal a 3.3 V para mantener la integridad física de la tarjeta de desarrollo Arduino DUE. Para realizar un cambio de nivel en un solo sentido de 5 V a 3.3 V se usa un transistor cuyo voltaje base de activación sea menor o igual a 5 V. Finalmente se decide usar un transistor BC547 para esta tarea, como lo propone T. Carpenter [22].

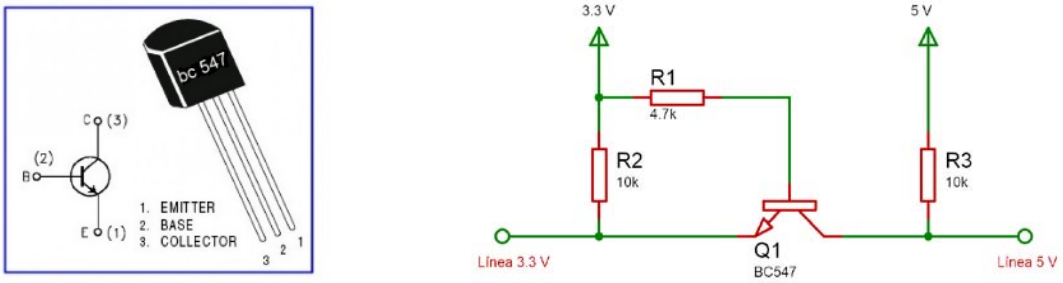


Figura 30. Diagrama de pines del transistor BC547, imagen tomada de la hoja de especificaciones y circuito de aplicación de conversor de nivel.

La señal de PWM que necesitan los motores para poder funcionar puede ser de 5 o 3.3 V por lo que tampoco es necesario realizar una interconexión en este caso.

A continuación se muestra el diagrama electrónico completo del robot móvil omnidireccional realizado en el software de diseño electrónico Proteus.

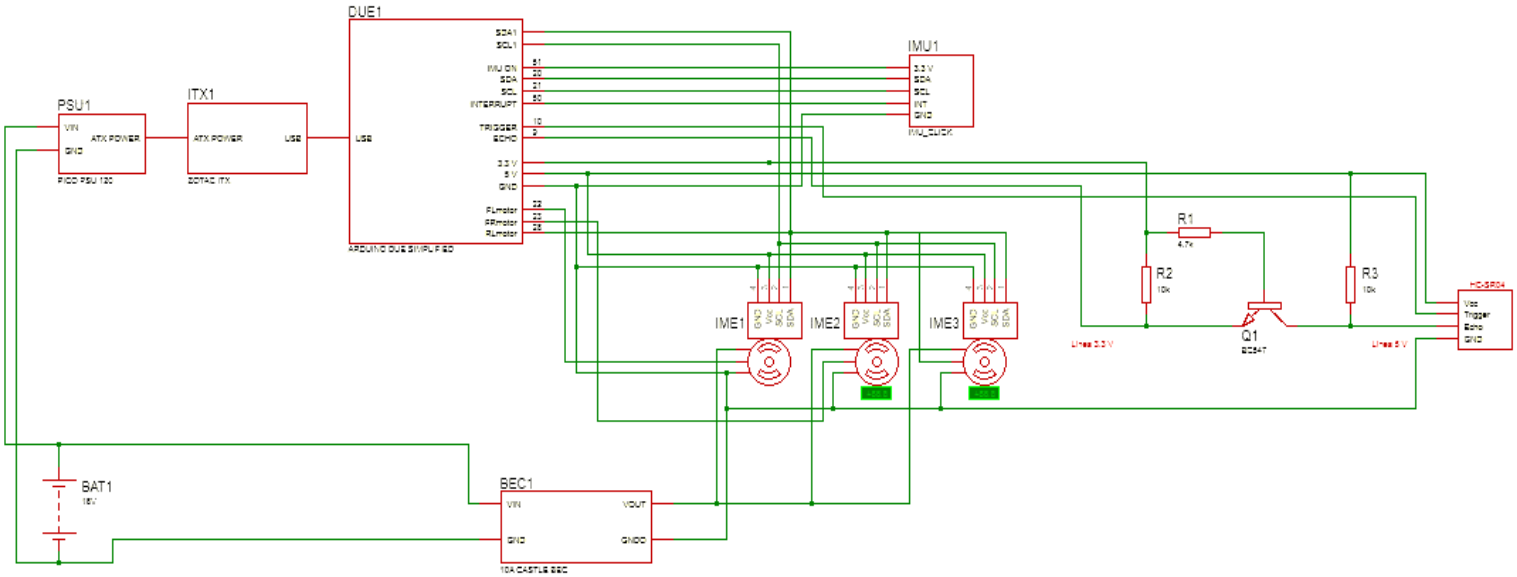


Figura 31. Diseño electrónico final del Robot Móvil Omnidireccional.

Capítulo 3. Análisis Cinemático del Robot Móvil Omnidireccional

3.1 Introducción

En este capítulo se presenta el análisis cinemático del Robot Móvil Omnidireccional. El análisis cinemático se refiere al desarrollo de las ecuaciones matemáticas, que basadas en las restricciones físicas de sus elementos, describen el movimiento del Robot Móvil Omnidireccional.

3.2 Postura del Robot

Para el análisis de la cinemática se consideran a los robots móviles compuestos por un marco no deformable y que se mueven en un plano horizontal. Se ubica una base inercial ortonormal de manera arbitraria $\{\overrightarrow{I_x}, \overrightarrow{I_y}, 0\}$ en el plano de movimiento. La posición del robot es definida por las variables x, y, θ como se muestra en la figura (Campion, 1993). También se define un sistema de referencia $\{\overrightarrow{x_1}, \overrightarrow{x_2}\}$ en el marco del robot móvil.

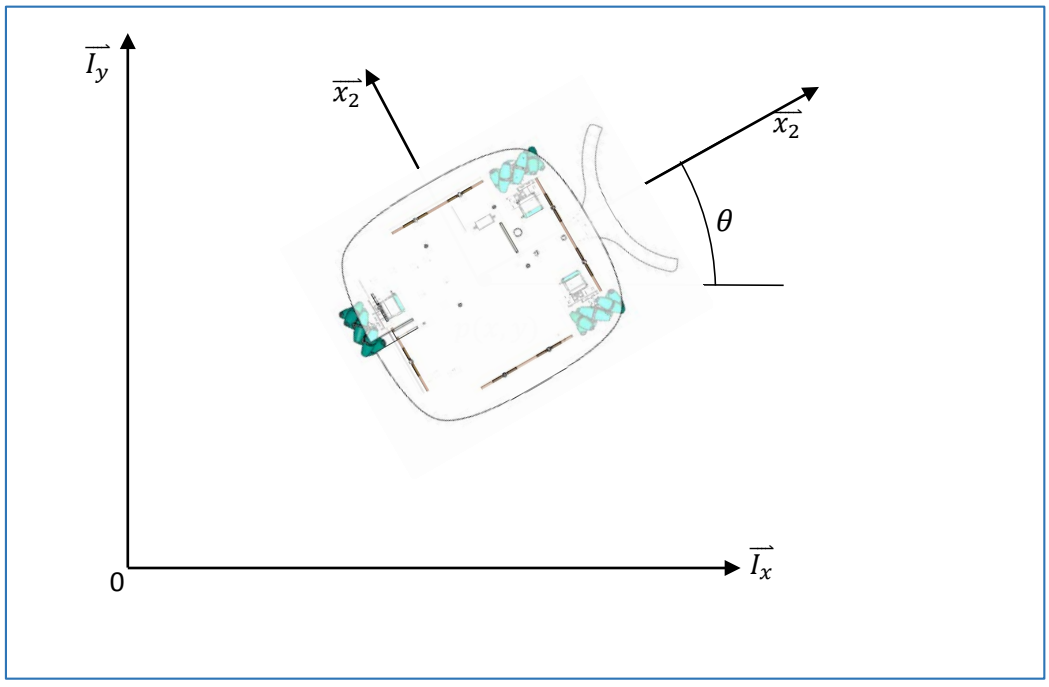


Figura 32. Sistema de referencia fijo y sistema de referencia móvil.

Las ruedas se consideran no deformables, y que siempre mantendrán un punto de contacto con el suelo para que pueda rodar sin deslizamiento. Se considera un solo punto de contacto con el plano de rodamiento. Para la llanta Mecanum (o sueca) solo un componente de la velocidad del punto de contacto de la rueda con el suelo se supone cero en todo el movimiento. Se considera arbitraria la dirección del componente cero de la velocidad pero su orientación es fija con respecto a la rueda.

La posición de la rueda *Mecanum* queda definida por los parámetros: b, d y γ, donde γ representa la orientación del componente cero de la velocidad durante el movimiento del robot. La descripción se basa en el sistema de referencia móvil, cuyo origen es el punto P.

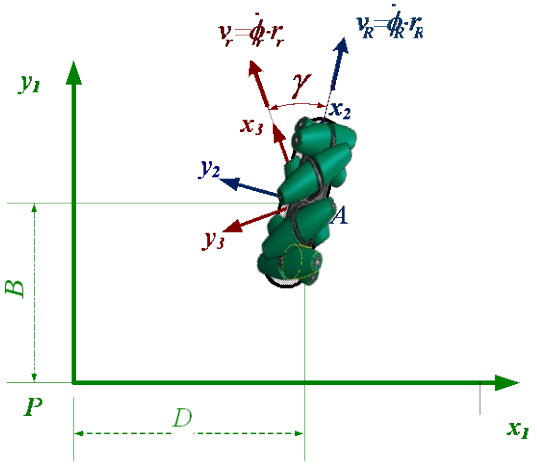


Figura 33. Definición de parámetros de modelado matemático de una rueda mecanum, adaptado de González-Villela.

Por lo tanto las ecuaciones cinemáticas de restricción para una llanta son:

$$\begin{bmatrix} \cos(\gamma+\theta) & sin(\gamma+\theta) & dsin(\gamma)-bcos(\gamma) & -r_R cos(\gamma) & -1 \\ -sin(\gamma+\theta) & cos(\gamma+\theta) & bsin(\gamma)+dcos(\gamma) & r_R sin(\gamma) & 0 \end{bmatrix} \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \\ \dot{\Phi}_R \\ v_r \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Ecuación 1

Donde:

$\dot{\theta}$ - Velocidad angular del ángulo de orientación, del sistema de referencia móvil con respecto al sistema de referencia fijo.
γ - Ángulo de orientación del punto de contacto de la rueda con la superficie de rodamiento.
b, d - Distancias asociadas a la ubicación de la rueda con respecto al punto P.

r_R - Radio de la rueda.
$\dot{\Phi}_R$ - Velocidad anular de la rueda.
v_R - Velocidad lineal del punto de contacto de la rueda con la superficie de rodamiento.
$\dot{x}$ - Velocidad lineal sobre el eje x del punto P.
$\dot{y}$ - Velocidad lineal sobre el eje y del punto P.

(González-Villela, 2014).

A continuación se muestra la definición de las dimensiones, sistemas de referencia fijo y móvil y la configuración completa del robot tomado como referencia para el modelado del robot móvil omnidireccional.

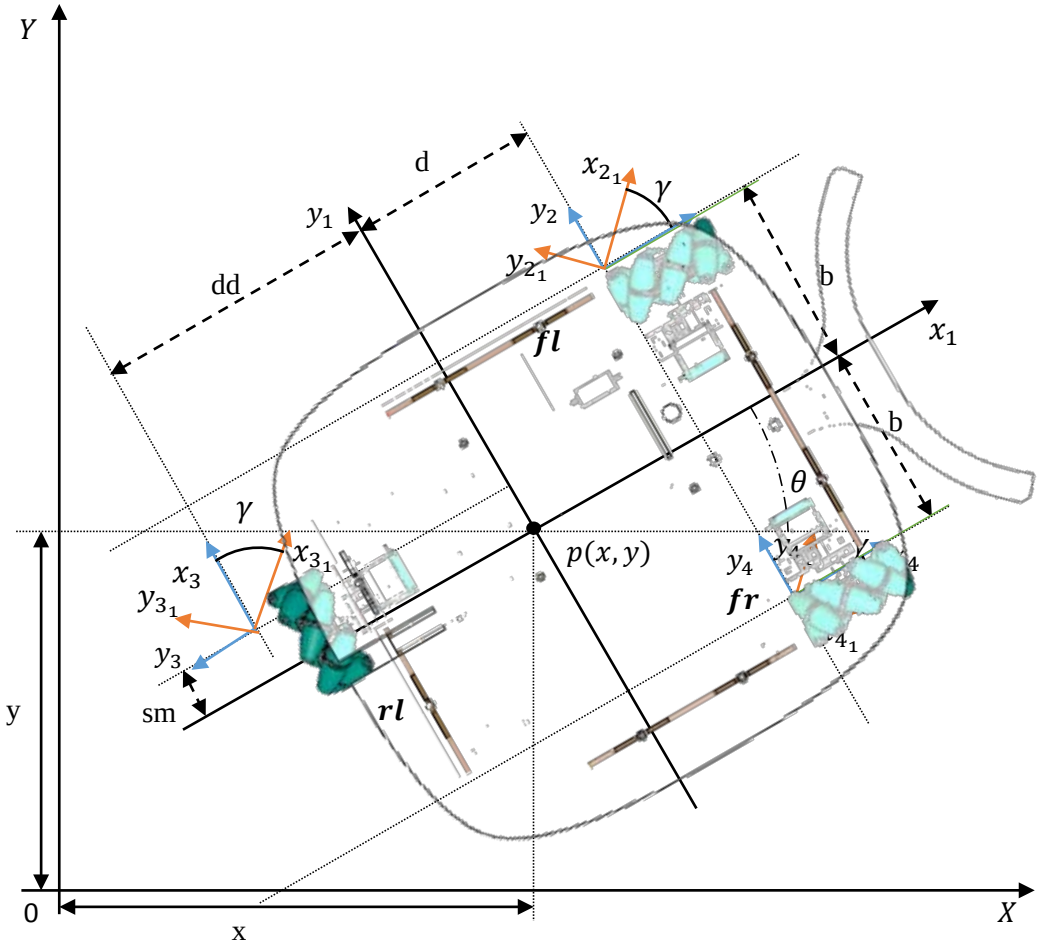


Figura 34. Definición de las dimensiones y configuración de las ruedas del robot.

El punto p define el origen del sistema de referencia móvil que se encuentra en el centro del robot y esta referencia por el sistema de referencia fijo. Se muestran además los subsistemas de referencia en color naranja que definen la

dirección de la velocidad del punto de contacto de la rueda con la superficie de rodamiento, el robot solamente está apoyado en sus tres ruedas *mecanum.* Se nombran cada una de las tres ruedas para su fácil identificación. La rueda fl es la rueda frontal izquierda, la rueda fr es la rueda frontal derecha y la rueda rl es la rueda posterior.

3.3 Coordenadas de configuración

En [24] se define que un robot de ruedas convencionales, con un total de:

$$N_T = N_f + N_c + N_{oc} + N_{sw}$$

Ecuación 2

Donde:

N_T – Total de ruedas convencionales.

N_c – Total de ruedas centradas giratorias.

N_{oc} – Total de ruedas descentradas.

N_{sw} – Total de ruedas *mecanum (suecas)*

Tiene un total de $n_T = 3 + N_T + N_c + N_{oc} + N_{sw}$ coordenadas de configuración $q = [q_{1,\cdots,}q_n]^T$, comprendidas de las coordenadas de postura ξ, coordenadas de las ruedas centradas giratorias $\propto_c$, coordenadas de las ruedas descentradas $\propto_{oc}$ y las coordenadas de rotación de las ruedas Φ.

Entonces, el número de coordenadas de configuración para el caso particular del robot móvil omnidireccional de tres ruedas *mecanum* son:

$$n_T = 3 + 3 = 6$$

Ecuación 3

Por lo tanto, las coordenadas generalizadas de configuración son:

$$q = [x, y, \theta, \Phi_{fl}, \Phi_{rl}, \Phi_{fr}]^T$$

Por lo que las velocidades generalizadas son:

$$\dot{q} = [\dot{x}, \dot{y}, \dot{\theta}, \dot{\Phi_{fl}}, \dot{\Phi_{rl}}, \dot{\Phi_{fr}}]^T$$

Una vez definidas las coordenadas de configuración se puede escribir la cinemática del robot con tres ruedas *mecanum* dentro de una sola expresión matricial de la forma $A_T(q)\dot{q} = 0$, donde $A_T(q)$ es la matriz asociada a las restricciones cinemáticas externas y esta parametrizada por las coordenadas de

configuración. Por lo que escribiendo la ecuación *1* para las tres ruedas con los parámetros descritos anteriormente y agrupándola en una sola ecuación matricial entonces queda de la siguiente forma:

$$\begin{bmatrix} -\sin(\gamma+\theta) & \cos(\gamma+\theta) & b\sin(\gamma)+d\cos(\gamma) & r\sin(\gamma) & 0 & 0 \\ -\sin(\theta-\gamma+90°) & \cos(\theta+\gamma+90°) & sm\sin(90°-\gamma)-dd\cos(90°'\gamma) & 0 & -r\sin(\gamma) & 0 \\ -\sin(-\gamma+\theta) & \cos(-\gamma+\theta) & d\cos(-\gamma)-b\sin(-\gamma) & 0 & 0 & -r\sin(\gamma) \end{bmatrix} \dot{q} = 0$$

Ecuación 4

Tomando en cuenta el ángulo de dirección de la velocidad $\gamma = \frac{\pi}{4}$ (definido por la forma física de las ruedas) se obtiene lo siguiente:

$$\begin{bmatrix} -\sin\left(\frac{1}{4}\cdot\pi+\theta\right) & \cos\left(\frac{\pi}{4}+\theta\right) & \sqrt{2}\cdot\left(\frac{b}{2}+\frac{d}{2}\right) & \frac{\sqrt{2}\cdot r}{2} & 0 & 0 \\ -\sin\left(\frac{1}{4}\cdot\pi+\theta\right) & \cos\left(\frac{\pi}{4}+\theta\right) & -\sqrt{2}\cdot\left(\frac{dd}{2}-\frac{sm}{2}\right) & 0 & -\frac{\sqrt{2}\cdot r}{2} & 0 \\ \cos\left(\frac{\pi}{4}+\theta\right) & \sin\left(\frac{\pi}{4}+\theta\right) & \sqrt{2}\cdot\left(\frac{b}{2}+\frac{d}{2}\right) & 0 & 0 & -\frac{\sqrt{2}\cdot r}{2} \end{bmatrix} \dot{q} = 0$$

Ecuación 5

3.4 Grado de movilidad, direccionabilidad y maniobrabilidad

En 1996 G. Campion introduce los términos de grado de movilidad, grado de direccionabilidad y grado de maniobrabilidad en robots móviles con ruedas. Estos términos también determinan el tipo de robot del que se trata. El grado de movilidad corresponde a los grados de libertad, es decir el número de variables que es posible controlar a partir de las velocidades de entrada del robot, sin tener que girar ninguna de sus ruedas, entendido como los grados de libertad instantáneos.

El grado de movilidad se puede calcular a partir de la siguiente ecuación:

$$r_m = \dim\left[Null\left(O_{fc}\right)\right] = 3 - rank(O_{fc})^6$$

Ecuación 6

En el caso de un robot móvil omnidireccional de ruedas *mecanum*, que no tiene ruedas fijas centradas, resulta:

$$r_m = 3$$

El grado de direccionabilidad r_s define cuantas ruedas centradas giratorias se pueden controlar. Por lo que en este caso, como no se tienen ese tipo de ruedas, da como resultado $r_s = 0$.

El grado de maniobrabilidad r_M define el número de grados de libertad totales que el robot puede controlar. El grado de maniobrabilidad queda definido por la siguiente ecuación:

$$r_M = r_m + r_s = 3$$

Ecuación 7

Dado el resultado anterior se puede observar una de las ventajas de los robots omnidireccionales, cuando su grado de movilidad es igual a su grado de maniobrabilidad, es decir que el robot no necesita realizar movimientos previos para poder moverse en la dirección deseada.

Además, G. Campion (1996) describe una clasificación de robots basado en su grado de movilidad y grado de direccionabilidad de la forma (r_m, r_s), por lo que en esa clasificación el robot móvil omnidireccional analizado queda dentro de la categoría (3,0), lo cual concuerda con la clasificación propuesta.

3.5 Cinemática inversa

Una vez que se obtiene la ecuación 4 y se definen como variables de salida las velocidades deseadas del robot $(\dot{x}, \dot{y}, \dot{\theta})$ y como variables de entrada las velocidades de las ruedas $(\dot{\Phi}_{fl}, \dot{\Phi}_{rl}, \dot{\Phi}_{fr})$. Por lo tanto se resuelve la ecuación 4 usando las variables $(\dot{x}, \dot{y}, \dot{\theta})$ y se tiene:

$$\begin{pmatrix} x' \\ y' \\ \theta' \\ \Phi'_{fl} \\ \Phi'_{rl} \\ \Phi'_{fr} \end{pmatrix} = \begin{pmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \\ \frac{1}{r} & \frac{1}{r} & \frac{b+d}{r} \\ \frac{1}{r} & \frac{1}{r} & \frac{dd-sm}{r} \\ \frac{1}{r} & \frac{1}{r} & \frac{b+d}{r} \end{pmatrix} \cdot \begin{pmatrix} v_{x1} \\ v_{y1} \\ w \end{pmatrix}$$

Ecuación 8

Es conveniente el cálculo de la cinemática inversa para poder determinar las velocidades que se deben ejecutar en las ruedas del robot a partir de la velocidad de salida requerida del robot. Por lo tanto si se resuelve la ecuación 8, que es una ecuación matricial, para las velocidades de las ruedas y tomando en consideración que $\theta = 0°$, siendo un caso particular de $A_T(q)\dot{q} = 0$ el cual da como resultado

$A_{T1}(q)\dot{q}_1 = 0$ que corresponde a su cinemática interna, es decir la asociada al sistema de referencia móvil. Por lo que se tiene:

$$\begin{bmatrix} \phi'_{fl} \\ \phi'_{rl} \\ \phi'_{rf} \end{bmatrix} = \begin{bmatrix} \frac{1}{r} & -\frac{1}{r} & -\frac{b+d}{r} \\ -\frac{1}{r} & \frac{1}{r} & -\frac{dd-sm}{r} \\ \frac{1}{r} & \frac{1}{r} & \frac{b+d}{r} \end{bmatrix} \cdot \begin{pmatrix} v_{.x1} \\ v_{.y1} \\ w \end{pmatrix}$$

Ecuación 9

3.6 Odometría

La odometría se encarga del estudio de la estimación de la posición de un vehículo. La palabra odometría, por las palabras de origen griego que la componen, significa la medida del trayecto.

Las principales técnicas de odometría en los vehículos con ruedas se basan en la información de la rotación de las ruedas para poder estimar la posición del vehículo a través del desplazamiento lineal relativo a la superficie de rodamiento. También se usa la palabra para nombrar a la distancia que el vehículo ha recorrido.

En la robótica móvil se utiliza para estimar la posición relativa del robot a partir de su posición inicial. La odometría tiene buenos resultados a corto plazo y es relativamente fácil de implementar si se conoce la cinemática del robot.

La posición de un objeto, conocida la velocidad en cada instante y la posición inicial, puede calcularse dado que la velocidad es la tasa de variación de la posición con respecto al tiempo, es decir que la velocidad es la derivada matemática de la posición.

$$\frac{d}{dt}p = v(t)$$

La velocidad puede ser variable en el intervalo de tiempo en que se mueve de un punto a otro. Si se divide el intervalo de tiempo que le toma moverse de una posición a otra en intervalos de tiempo de muy corta duración, que llamaremos el tiempo de muestreo, y se supone una velocidad constante entre cada intervalo entonces, se puede hacer una aproximación de la posición instantánea del objeto en cada instante.

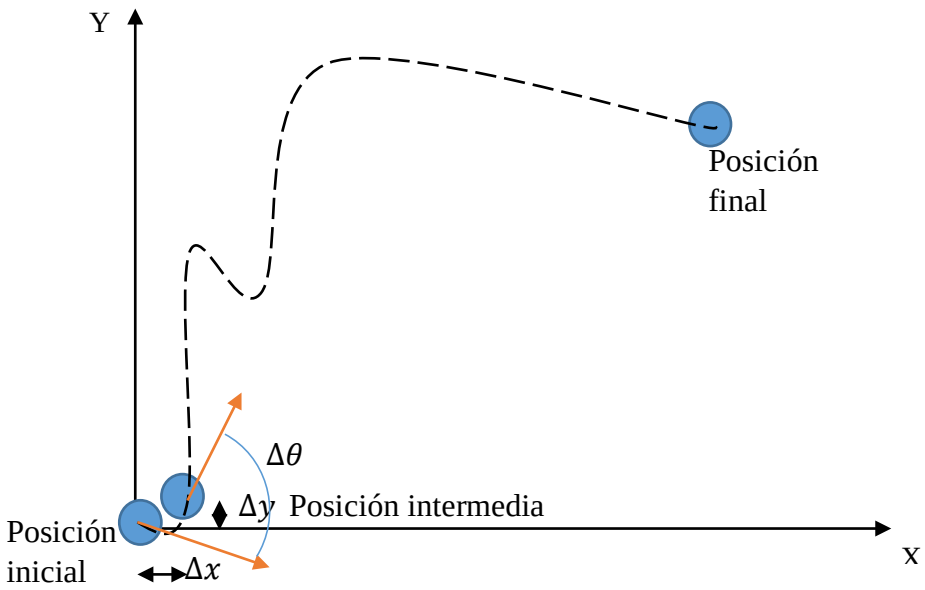


Figura 35. Gráfica que muestra el cálculo de la posición instantánea.

En la figura 35 se puede apreciar la posición inicial de un objeto que se mueve en el plano {x,y} con una orientación Θ a lo largo de su trayectoria. Se muestra una posición intermedia, la cual se encuentra con una diferencia de tiempo muy pequeña y se conoce su velocidad. Además, la posición intermedia queda definida por la posición inicial más un incremento en la posición sobre el eje *x,* un incremento en el eje *y* y un incremento en la orientación. Estos pequeños incrementos se calculan como el producto del intervalo de tiempo entre la posición inicial y la posición intermedia por la velocidad instantánea $(\dot{x}, \dot{y}, \dot{\theta})$. Entonces se tiene que:

$$x_i = x_{i-1} + \Delta x$$

Ecuación 10

$$y_i = y_{i-1} + \Delta y$$

Ecuación 11

$$\theta_i = \theta_{i-1} + \Delta\theta$$

Ecuación 12

Donde:

x_i – Es la posición actual en el eje x.

y_i – Es la posición actual en el eje y.

θ – Es la orientación relativa.

x_{i-1} – Es la posición anterior en x.

y_{i-1} – Es la posición anterior en y.

θ_{i-1} – Es la orientación anterior.

Δx – Incremento en la posición en el eje x.

Δy – Incremento en la posición en el eje y.

$\Delta\theta$ – Incremento en la orientación.

Donde los incrementos se calculan como:

$$\Delta x = \dot{x}\,\Delta t$$
$$\Delta y = \dot{y}\,\Delta t$$
$$\Delta\theta = \dot{\theta}\,\Delta t$$

Donde Δt es el tiempo de muestreo entre las dos posiciones.

En el caso de robótica móvil la posición inicial se considera cero para el inicio del cálculo de la estimación odométrica ($x = y = 0, \theta = 0$). En el caso particular del robot móvil omnidireccional descrito en el presente trabajo se puede conocer la velocidad de las ruedas en cada momento. Por lo tanto, a partir de la cinemática directa del robot se puede conocer las velocidades del robot en cada momento. La cinemática directa del robot móvil omnidireccional se puede calcular a partir de la ecuación 9, porque contiene una matriz cuadrada de rango 3 (por su omnidireccionabilidad) que es invertible, entonces se tiene que:

$$\begin{pmatrix} v_{.x1} \\ v_{.y1} \\ w \end{pmatrix} = \begin{pmatrix} \frac{b\cdot r + d\cdot r - dd\cdot r + r\cdot sm}{2\cdot b + 2\cdot d + 2\cdot dd - 2\cdot sm} & \frac{b\cdot r + d\cdot r}{b + d + dd - sm} & \frac{r}{2} \\ \frac{r}{2} & 0 & \frac{r}{2} \\ \frac{r}{b + d + dd - sm} & \frac{r}{b + d + dd - sm} & 0 \end{pmatrix} \begin{bmatrix} \phi'_{fl} \\ \phi'_{rl} \\ \phi'_{rf} \end{bmatrix}$$

Ecuación 13

Por último, para calcular la velocidad que necesita la ecuación de estimación odométrica, se calcula primero la velocidad del robot móvil con la Ecuación 13 y después se calcula su velocidad en el sistema de referencia fijo con la Ecuación 8. Después se hace el cálculo recurrente de la estimación odométrica usando las ecuaciones 10, 11 y 12.

Capítulo 4. Sistema Operativo Robótico

4.1 Introducción

El sistema de control de un robot móvil es el que se encarga de controlar los comportamientos de un robot móvil basado en su percepción del mundo. El sistema de control se encarga del procesamiento de las señales que percibe a través de su sistema sensorial. El sistema de control se ejecuta sobre una computadora, es decir que es un software. Existen diferentes plataformas para la programación de robots y usan diferentes lenguajes de programación como c++, c#, java, Python, etc.

En el año 2009 surge el Sistema Operativo Robótico (ROS) como una solución al desarrollo de software para robots y para alentar la creación de soluciones robóticas de manera colaborativa. ROS es una plataforma de desarrollo y un conjunto de librerías y paqueterías de software. ROS es un software de código abierto y cuenta con herramientas comunes para la creación de software robótico así como de una comunidad internacional de desarrolladores y entusiastas en robótica. El Sistema Operativo Robótico pretende también ser una plataforma robótica transparente donde fácilmente se tenga acceso a los datos de sensores, procesos y tareas.

ROS es un meta sistema operativo, es decir que necesita de un sistema operativo anfitrión para poder ejecutarse y que esté basado en Ubix. La distribución Ubuntu basada en Unix es uno de los Sistemas Operativos con mejor soporte para la correcta ejecución de ROS. Como sistema operativo, ROS tiene una estructura distribuida y modular. Esto permite poder instalar los componentes mínimos necesarios para el proyecto que se desarrolle.

4.1.1 Niveles de conceptos de ROS

ROS funciona bajo tres niveles de concepto (ROS Wikipedia, 2014).

1. Nivel de sistema de archivos de ROS

   Los conceptos en el sistema de archivos cubren los recursos computacionales de los que hace uso. Estos conceptos son paqueterías, meta paqueterías, manifiestos de paquetería, repositorios, tipos de

mensaje y tipos de servicio. Las paqueterías son el conjunto de programas destinados a realizar una tarea en específico como por ejemplo obtener la lectura de un sensor y procesarla para que pueda ser leída en una escala común. Las meta paqueterías agrupan a paqueterías que trabajan en conjunto para realizar una tarea más compleja. Los repositorios son colecciones de paqueterías con un sistema común. Los tipos de mensaje y tipos de servicio contienen la descripción de los mensajes y de los servicios respectivamente.

2. Nivel de computación de grafos en ROS

   La computación de grafos es una estructura de datos abstractos que se compone de un conjunto de nodos y un conjunto de aristas que establecen las relaciones entre los nodos. Los conceptos que maneja este nivel son:

   - *Nodes*: Los nodos son todos los procesos que realizan algún tipo de cómputo. Es decir son cada uno de los programas individuales.
   - *Master*: El *ROS Master* es el encargado de hacer un registro del nombre y buscar todos los nodos. Es el encargado de realizar la comunicación entre los nodos.
   - *Parameter Server*: Es el servidor donde se registran los parámetros de los programas.
   - *Messages*: Los mensajes son los datos que son usados para la comunicación entre los diferentes nodos.
   - *Topics*: Los tópicos es el medio de transporte de los mensajes. Los nodos pueden *publicar* o *suscribirse* a estos tópicos.
   - *Services*: Los servicios son usados cuando se requiere una estructura de *solicitud – respuesta* en el procesamiento de datos.
   - *Bags*: Es el formato que usa ROS para guardar y reproducir información, independientemente del tipo de mensaje que almacene.

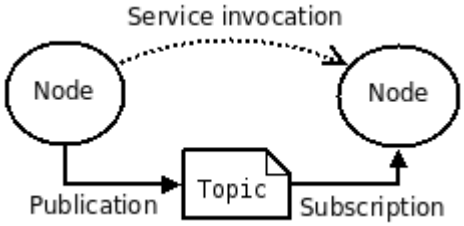


Figura 36. Esta imagen muestra la relación entre un nodo que publica y un nodo subscriptor así como del llamado de un servicio (ROS wikipedia, 2014).

3. Nivel de comunidad de ROS

   Este nivel de conceptos se refiere al desarrollo colaborativo de software en robótica. Se compone de las distribuciones de ROS, repositorios, el wiki, del sistema de reporte de fallas, la lista de correos, ROS respuestas y el blog. Sirve como un sistema de comunicación entre los creadores de ROS, desarrolladores y entusiastas para resolver dudas y problemas en común relacionadas con el desarrollo de software para los robots.

4.2 Controlador de la base móvil – Software de Arduino

4.2.1 Controlador de motores

En el Capítulo 2 se mencionaron las partes que conforman el Robot Móvil Omnidireccional. Su sistema de locomoción está conformado por un microcontrolador, sensores de posición y velocidad de las ruedas, motores con controlador de motores y tres ruedas *mecanum.* En las especificaciones del controlador de motores se describe que alcanzará su velocidad máxima nominal (100 rpm) al aplicar una señal de Modulación por Ancho de Pulso de 2 ms, su velocidad nominal en reversa al aplicar la señal de 1 ms y sin velocidad con una señal de 1.5 ms. El controlador es un controlador de lazo abierto, es decir que no existe una retroalimentación de la señal.

Los motores cuentan con un encoder digital con comunicación I2C[1] para la lectura de la velocidad y posición de las ruedas. La comunicación I2C se usa cuando se requiere comunicar un dispositivo maestro con un dispositivo esclavo a corta distancia. El protocolo I2C se basa en el uso de dos cables: uno para datos y uno para una señal de sincronización de reloj. Se usa la librería desarrollada por

[1] I2C – Inter Circuitos Integrados. Formato de comunicación inter circuitos.

A. Henning [26] para la comunicación I2C entre un microcontrolador Arduino y los *encoders* integrados en los motores.

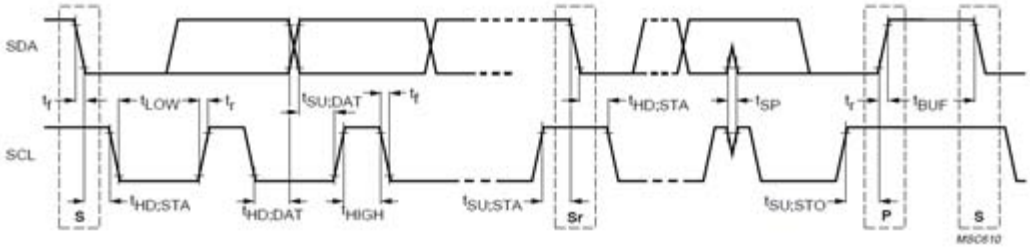


Figura 37. Descripción de tiempos para una comunicación I2C.

Se realizaron pruebas con el controlador y se observó que no existe una relación lineal entre la señal de entrada, de modulación PWM, y la velocidad del motor.

Se decide usar un controlador Proporcional, Integral, Derivativo en cada uno de los motores para garantizar que se alcance la velocidad comandada por el microcontrolador y un cambio suave de las velocidades de las ruedas.

Un controlador PID está conformado por un controlador Proporcional cuya salida es proporcional al error de la variable a controlar, un controlador Integral que es proporcional al error acumulado y un controlador Derivativo que es proporcional al cambio del error por unidad de tiempo, las salidas de los tres controladores se suman para dar resultado a una sola salida.

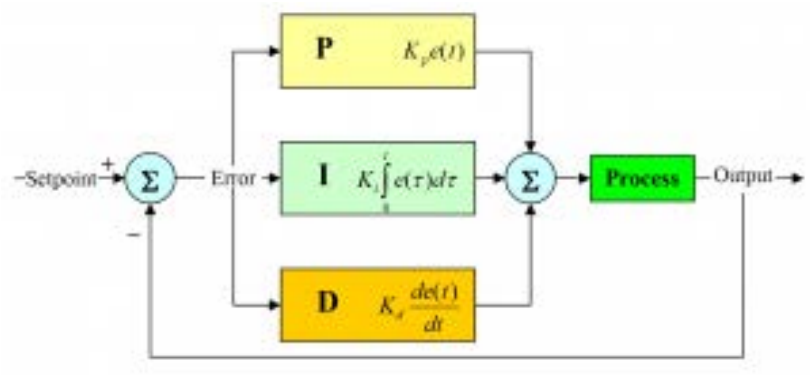


Figura 38. Esquema de control de un controlador PID, imagen tomada de STM32F4 Discovery Libraries and tutorials, 2014.

Se hace uso de la librería de Arduino de PID desarrollada por B. Beauregard [27] para la implementación de un control PID digital con la tarjeta Arduino. La

librería de PID permite la implementación de diferentes controladores PID en una sola tarjeta Arduino, además de que está basado en los controladores PID industriales.

En situaciones en las que se necesita implementar un control PID pero no se conoce el modelo del sistema, es posible determinar experimentalmente valores aceptables del controlador PID [28]. Se determinaron los valores de las ganancias del sistema de control para el control de velocidad de un motor a partir del siguiente método (T. Braunl, Williams, 2006):

1. Selecciona un valor típico de operación para la velocidad deseada, apaga las partes derivativa e integral, después incrementa la ganancia Proporcional K_P hasta un máximo o hasta que el motor entre en oscilación.
2. Si el sistema oscila, divide K_P entre dos.
3. Incrementa la ganancia derivativa K_D y observa el comportamiento cuando se incrementa/reduce la velocidad deseada un 5%. Elige el valor de K_D que resulte en una respuesta amortiguada.
4. Lentamente incrementa el valor de la ganancia Integral K_I hasta que la oscilación inicie. Luego, divide K_I entre dos o entre 3.
5. Revisa que el desempeño general es satisfactorio en condiciones típicas de operación.

Posteriormente, se desarrolla una interfaz en Simulink® de MATLAB® para modificar manualmente las ganancias Proporcional, Integral y Derivativo y observar en una gráfica la respuesta del motor. Se realiza una conexión serial bajo el protocolo RS-232 para el envío y recibimiento de datos del microcontrolador Arduino y la computadora.

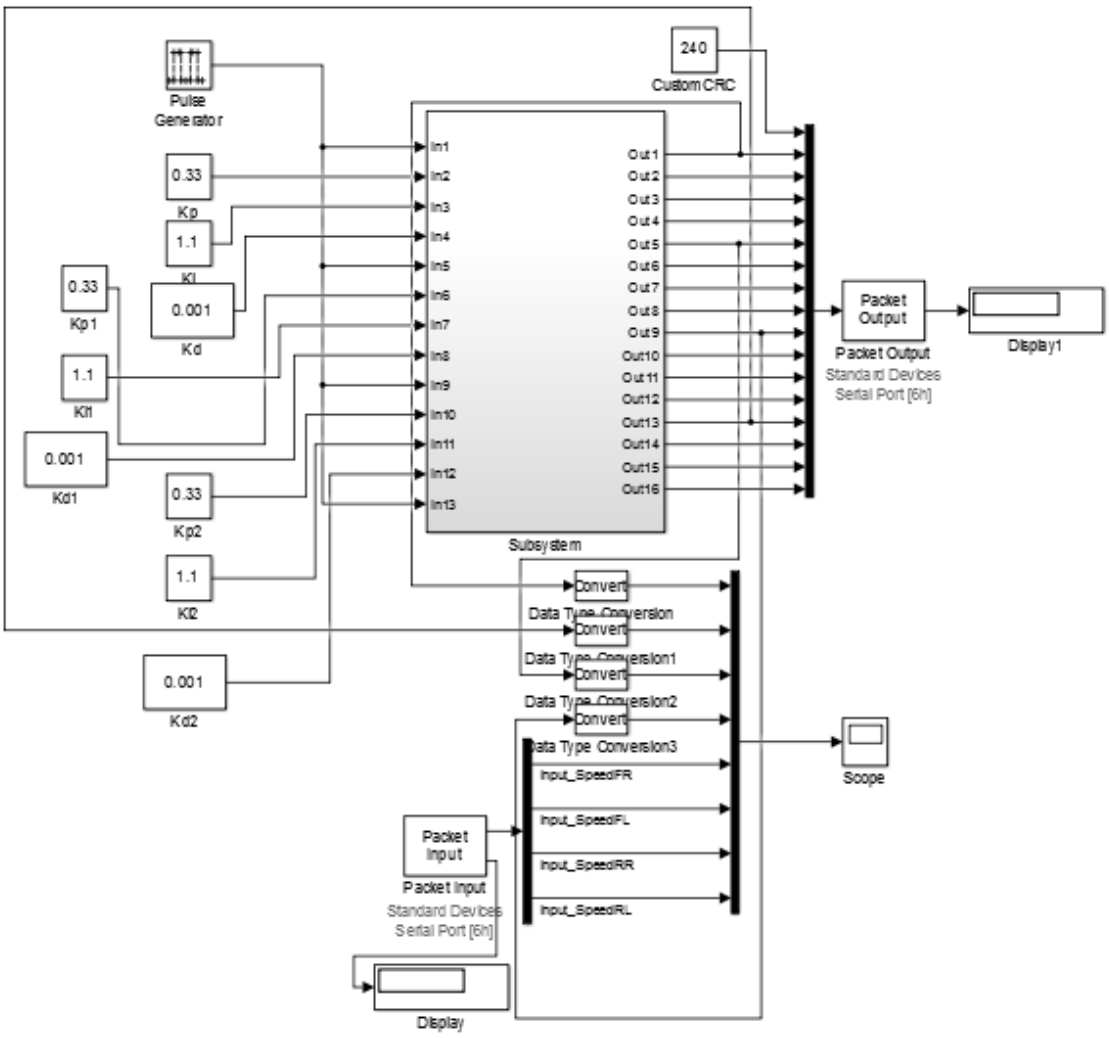


Figura 39. Interfaz programada en Simulink® para modificar manualmente los valores de las ganancias de cada uno de los motores del Robot Móvil Omnidireccional.

Se aplica el método descrito anteriormente y se obtienen los siguientes valores de K_P, K_I y K_D para cada uno de los motores:

Motor	K_P	K_I	K_D
FR	0.46	1.6	0.001
RL	0.454	1.65	0.0012
FL	0.465	1.54	0.001

Tabla 7. Valores de las ganancias de los controladores PID de cada una de las ruedas del Robot Móvil Omnidireccional

En la tabla anterior se puede observar que los valores de las ganancias para cada uno de los motores son muy similares entre ellas y es porque los motores son del mismo modelo y se encuentran en condiciones similares de carga.

A continuación se muestran graficas de la respuesta en velocidad de los motores a una entrada escalón y a entradas de pulsos con el sistema de control PID implementado.

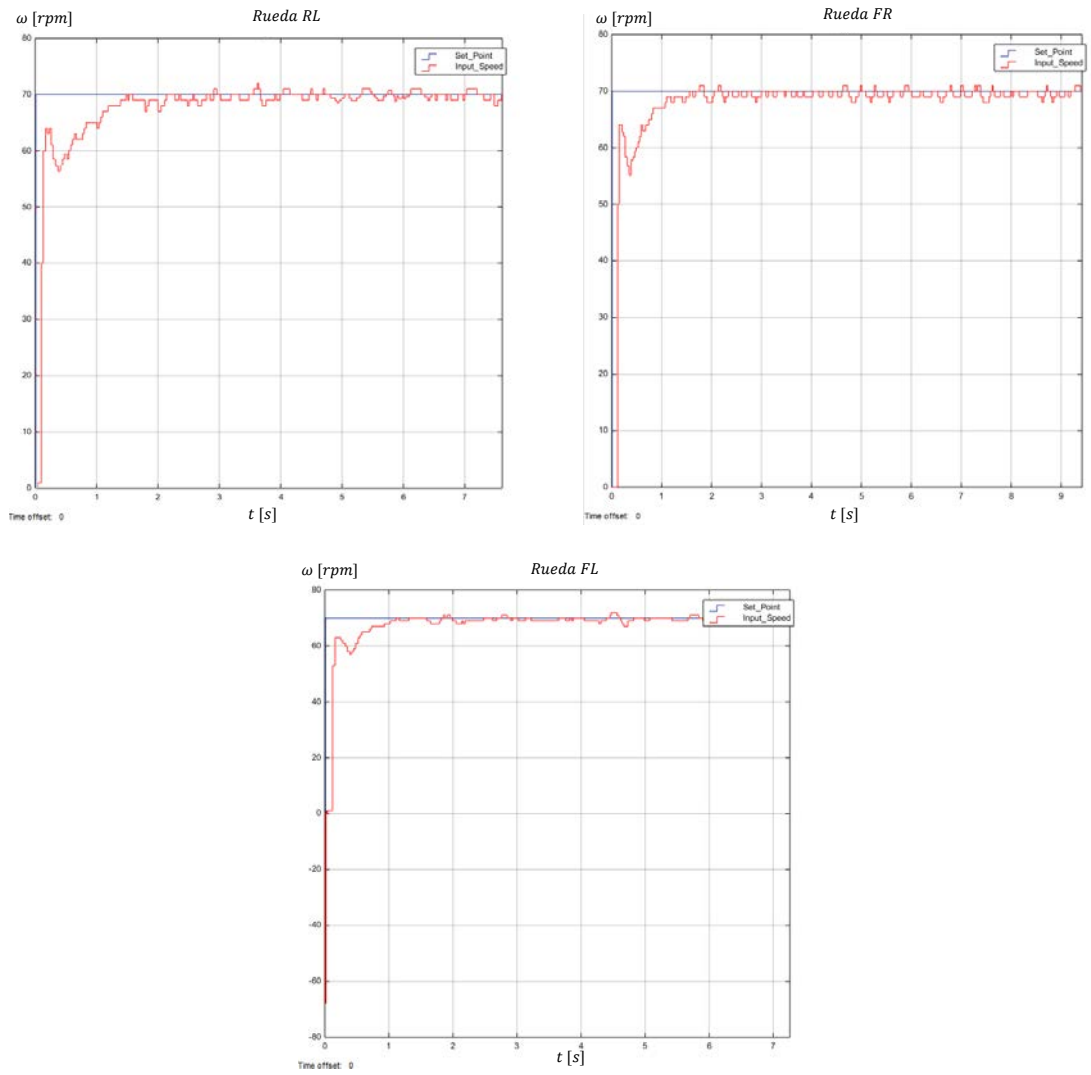


Figura 40. Respuestas de los motores en velocidad a una entrada escalón con control PID digital.

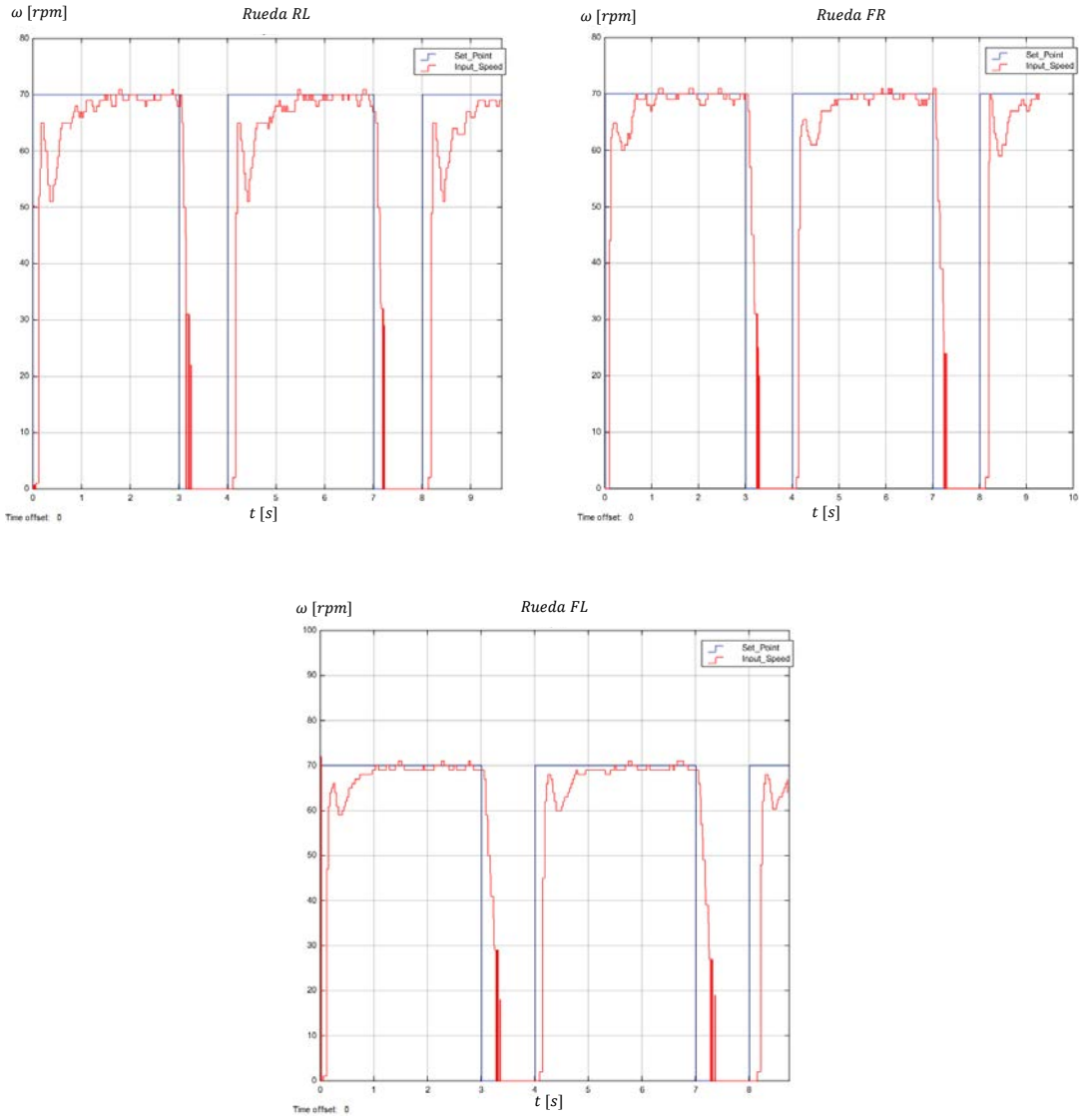


Figura 41. Respuesta en velocidad de los motores a una entrada de pulsos cuadrados.

4.2.2 Unidad de Medición Inercial

Una Unidad de Medición Inercial (IMU – Inertial Measurement Unit) es un dispositivo usado en los sistemas de navegación que mide la velocidad, orientación y aceleración. Su sistema de medición se basa en el uso de acelerómetros y giroscopios, que en conjunto con un sistema de procesamiento puede determinar la aceleración, velocidad, y orientación del sistema.

Un IMU determina la orientación del sistema midiendo los cambios de cabeceo (pitch), alabeo (roll) y guiñada (yaw) del sistema. El acelerómetro del IMU mide la aceleración en uno, dos o tres ejes, dependiendo del modelo de acelerómetro,

mientras que el giroscopio mide la velocidad angular, es decir la velocidad de rotación.

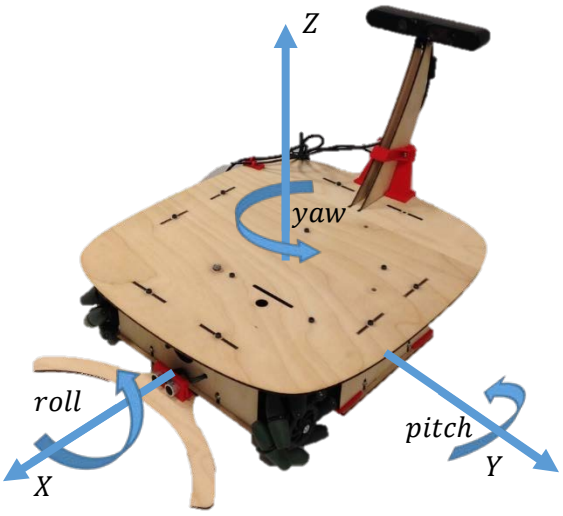


Figura 42. Definición de las rotaciones yaw, pitch, roll de un sistema. En específico del Robot Móvil Omnidireccional.

El dispositivo MPU6050 es un IMU que tiene un sensor acelerómetro de tres ejes con un sensor giroscopio de tres ejes. Además, posee un Procesador Digital de Movimiento (Digital Motion Processor™ - DMP™) capaz de procesar algoritmos de fusión de movimiento y se encarga de el alineamiento de los ejes. Este dispositivo puede conseguirse en diferentes presentaciones de placa de circuito impreso. Como se mencionó en el Capítulo 2 la placa de circuito impreso del IMU es de la marca mikroelektronika.

La implementación del sensor se hace a través del uso de la librería para Arduino desarrollada por J. Rowberg [29] para la configuración y uso del IMU MPU6050. La librería hace uso de las funciones del Procesador Digital de Movimiento, la cual releva de procesamiento al microcontrolador Arduino y envía los datos de aceleración y orientación ya procesados para hacer uso de ellos. Se requiere un tiempo de espera de aproximadamente treinta segundos antes de poder hacer uso del IMU porque es el tiempo que requiere para su auto calibración. Debido a que el robot se mueve sobre el plano $\{x, y\}$ el único valor del IMU que es de interés es el de la guiñada (yaw).

4.2.3 Sensor de distancia ultrasónico

En la parte frontal del robot existe una zona que el robot no puede percibir con su sensor principal, la cámara Asus Xtion Pro Live, por la posición del sensor y por la forma del robot. En esta zona se ubica el sensor ultrasónico HC SR04, el cual puede medir la distancia a la que se encuentra un objeto enfrente de este.

El principio de funcionamiento del sensor HC SR04, como se mencionó en el Capítulo 2, se basa en la medición del tiempo que le toma regresar a una señal ultrasónica al ser reflejada por un objeto sólido. El sensor tiene dos pines para controlarlo: *Trigger* y *Echo.* El pin *Trigger*, como su nombre lo indica, es el disparador de la señal, cuando se le aplica un voltaje lógico en nivel alto. El pin *Echo* cambiará su nivel lógico de voltaje en un tiempo directamente proporcional a la distancia que se encuentra el objeto. El sensor ultrasónico funcionará mejor en un ángulo de detección de 30° (arcrobotics, 2014).

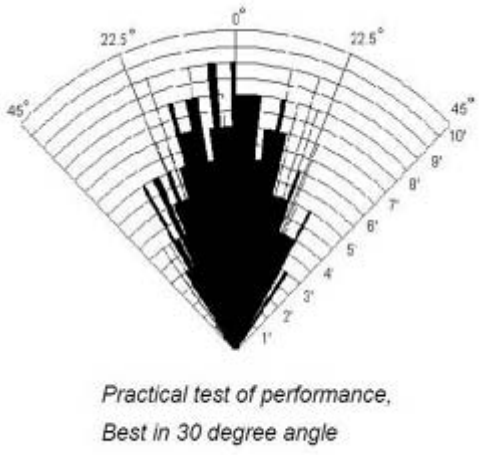


Figura 43. Ángulo de mejor detección para un sensor ultrasónico HC SR04. Imagen tomada de arcrobotics, 2014.

Se programa la lectura del sensor ultrasónico con la librería del sensor ultrasónico desarrollada por J. Rodrigo [30] para Arduino. La función del sensor es determinar si un objeto se encuentra cerca del robot o no, como por ejemplo cuando el robot empuja algún objeto.

4.2.4 Nodo Arduino en ROS y comunicación Serial

4.2.4.1 Librerías de cinemática y odometría en Arduino

El Arduino DUE actúa como interfaz entre el sensor ultrasónico, los *encoders* y la Unidad de Medición Inercial. También es la interfaz que sirve como controlador de la base móvil, la cual recibe los datos de velocidad de la base móvil referenciados al sistema de referencia móvil y reporta su estimación Odométrica. El Arduino DUE es capaz de procesar esta información porque tiene un procesador de 32 bits. Para programar de una manera eficiente se crearon dos librerías para el Arduino: *kinematics_arduino3* y *Odometer3*, las cuales contienen los objetos[2] y métodos[3] de la cinemática y la odometría.

La librería *kinematics_arduino3* define una clase[4] del tipo *kinematics_arduino3* con elementos que contienen apuntadores, es decir que no crean un espacio dedicado en memoria sino que crea una referencia a otra variable en memoria, a las variables de interés que son las variables de velocidad de las ruedas y velocidad del robot. Se usan apuntadores para no utilizar instrucciones y espacios de memoria extra en el microcontrolador.

Los métodos incluidos en la clase *kinematics_arduino3* son:

- *compute:* Calcula la cinemática inversa del Robot Móvil Omnidireccional a partir de las velocidades del robot requeridas. El resultado se ve reflejado en las variables de velocidad de las ruedas.
- *stop_speed:* Fija en cero las variables de las ruedas. Al fijar en cero las variables de las ruedas se ejecuta automáticamente la instrucción para llevar a velocidad cero las ruedas del robot.

Por otro lado, la librería *Odometer3* contiene la clase *Odometer3* con apuntadores a las variables de las velocidades de las ruedas y del *yaw* (guiñada) del IMU. Las velocidades del robot no las asocia a las velocidades comandadas puesto que existe un pequeño retraso entre la velocidad comandada y la velocidad real, siendo de interés también la velocidad real. Los métodos de la clase *Odometer3* son:

[2] Programa de computadora que se compone de un estado y un comportamiento.
[3] Subrutina en donde su código se define en una clase.
[4] Modelo sobre el cual se construirá un objeto.

- *update:* Realiza el cálculo de la cinemática inversa para calcular la velocidad instantánea del robot y también realiza el cálculo de odometría descrito anteriormente.
- *useImu:* Habilita el uso del IMU para el cálculo de la orientación del robot.
- *noImu:* Deshabilita el uso del IMU para el cálculo de la orientación del robot. La orientación del robot se calcula con base en la velocidad de las ruedas del robot.

4.2.4.2 Comunicación Serial Arduino

Para comunicar a la computadora en el interior del robot con el Arduino se utiliza el protocolo de comunicación Serial RS-232. El protocolo de comunicación RS-232 es un protocolo de comunicación punto a punto, es decir que solo puede haber dos puntos de comunicación en este protocolo a diferencia del protocolo de comunicación I2C que puede ser multipunto. Para hacer la implementación de un protocolo de comunicación con el Arduino se utiliza la librería *Messenger* para Arduino de T. O. Fredericks [31] la cual facilita el análisis de los mensajes en ascii. Esta librería solo se usa para recibir los datos de la computadora en el Arduino, para enviarlos se usa el método común de Arduino para enviarlos por el puerto Serial.

4.2.4.3 Programa principal de Arduino

Programa principal de Arduino
1. Inicio
2. Declaración de librerías *Messenger.h, kinematics_arduino3.h, odometer3.h, PID_v1.h, Servo.h, Wire.h, I2CEncoder.h, Ultrasonic.h, I2Cdev.h, MPU6050_6Axis_MotionApps20.h*
3. Declaración de variables y objetos globales de apoyo.
4. Configuración de protocolos de comunicación I2C y RS-232, objetos y de todos los dispositivos.
5. Leer puerto serial. **Revisar** Instrucciones de velocidad y de uso del IMU.
6. Sí inicializado **Y** time_update > 20 ms.

7. **Haz** pidcompute(); motorwrite(); odometer.update(); printodometry(); printultrasonic().
8. **De otro modo** Pide inicialización por puerto serial.
9. **Sí** *tiempo_inicio > 30 [s]* **Y** imu_active = *true*.
10. **Haz** imu_use = *true*.
11. **Sí** mpuInterrupt **O** data = *true*.
12. **Haz** getIMUdata()
13. Fin

Tabla 8. Pseudocódigo del programa principal en Arduino.

4.2.4.4 Nodo de ROS

Se programa un Nodo (*Node*) en ROS para recibir los datos de Arduino y publicarlos en diferentes tópicos (*Topics)* dependiendo del tipo de información que contengan. En ROS existen tipos de mensajes predefinidos de uso común en robótica como por ejemplo un mensaje tipo *Twist* que define instrucciones de velocidad lineal y angular de un robot móvil.

El Arduino recibirá las instrucciones de la velocidad de la base móvil y si se usará el IMU o no, y transmitirá los datos de odometría, del sensor ultrasónico y algunos datos de aviso del estado del microcontrolador. Se hace uso de la clase *SerialDataGateway* desarrollada por R. Hessmer [32] para recibir y enviar datos del Arduino por medio del puerto serial de la computadora.

En ROS está definido un tipo de mensaje que contiene los comandos de velocidad para cualquier robot móvil, cualquiera sea su medio y forma de locomoción. Este mensaje es *Twist* y lo componen dos vectores de tres componentes xyz: el vector de velocidad lineal y el vector de velocidad angular. En la documentación de ROS se define que las dimensiones para la velocidad lineal son $^{m}/_{s}$ y para la velocidad angular son $^{rad}/_{s}$.

Twist	*Vector3 linear*	*float64 x*
		float64 y
		float64 z
	Vector3 angular	*float64 x*
		float64 y

float64 z

Figura 44. Definición del tipo de mensaje Twist, tomado de la Documentación de ROS.

Para el envío de datos de sensores de distancia está definido el tipo de mensaje *Range.* El mensaje *Range* se compone de un encabezado que contiene la información de su sistema de referencia y el tiempo en que envía el mensaje, definición del sensor que se usa, definición del tipo de señal que usa, definición de su campo de visión (un arco donde se acepta la lectura), mínima distancia de detección, máxima distancia de detección y la lectura del sensor cuyas dimensiones son en metros.

Range
uint8 ULTRASOUND=0
uint8 INFRARED=1
Header header
uint32 seq
time stamp
string frame_id
uint8 radiation_type
float32 field_of_view
float32 min_range
float32 max_range
float32 range

Figura 45. Definición del mensaje tipo Range, tomado de la documentación de ROS.

También está definido el mensaje para la Odometría, el cual es del tipo *Odometry.* Este mensaje se compone de un encabezado, el sistema de referencia anterior, una pose y un mensaje tipo *Twist.* En la Pose se especifica la posición y orientación del robot. En el mensaje tipo *Twist* se reporta la velocidad del robot.

Odometry
Header header
string child_frame_id
PoseWithCovariance pose
Point position
Quaternion orientation
PoseWithCovariance twist

Figura 46. Definición del tipo de mensaje Odometry, tomado de la documentación de ROS.

Se define un tipo de mensaje llamado *speed_wheel* para poder tener acceso a las velocidades de las llantas medidas por los *encoders* con unidades en rpm. El mensaje *speed_wheel* esta definido como sigue:

speed_wheel

Header header
float32 FL
float32 RL
float32 FR

Figura 47. Definición del tipo de mensaje speed_wheel.

El nodo se compone por una clase *Arduino* que hace uso de la clase *SerialDataGateway* para la comunicación con el Arduino. La clase *Arduino* tiene definido los métodos para recibir los datos del Arduino y procesarlos para enviar mensajes de tipo *Odometry* y *Range* y recibir mensajes de tipo *Twist.* Entonces en la clase se definen dos publicadores y un subscriptor. Además, es necesario definir un servicio para la activación o desactivación del IMU. Se define un servicio porque es una acción que no requiere estar todo el tiempo procesando sino hasta que se le requiera [33]. Por último, se define un método cuya función es desplegar en pantalla información del Arduino como mensajes de alerta o información.

Para ejecutar el programa (Nodo) en ROS se sigue la siguiente sintaxis en una Terminal:

rosrun [nombre_paqutería] [nombre_nodo]

Por lo tanto para ejecutar el nodo de Arduino se ejecuta en Terminal:

rosrun om_bot Arduino.py

4.3 Cámara de Profundidad y Calibración

En el capítulo 2 se describe una cámara de profundidad que funciona como el sensor principal del robot. La función principal de la cámara será para navegación, lo que significa que percibirá el ambiente y los obstáculos en el camino del robot. La paquetería openni2_launch para ROS contiene los drivers necesarios para que la cámara Asus Xtion Pro Live funcione en conjunto con ROS y publique la imagen en color, de profundidad y en nube de puntos. También tiene soporte para leer los archivos de configuración y corrección lenticular de la cámara.

Una cámara de profundidad puede percibir el ambiente en tres dimensiones. Está compuesta por un proyector, una cámara de color y una cámara infrarroja. Su principio de funcionamiento se basa en la proyección de un patrón en luz infrarroja que la cámara de luz infrarroja detecta y compara con el patrón para determinar e inferir la distancia a la que se encuentra cada uno de los pixeles, después junta la información de distancia de los pixeles con el color de los pixeles

que detecta con la cámara de color. Debido a que su principio de funcionamiento se basa en la detección de luz infrarroja no es aconsejable su uso en exteriores por la interferencia con la luz solar, la cual también emite luz infrarroja.

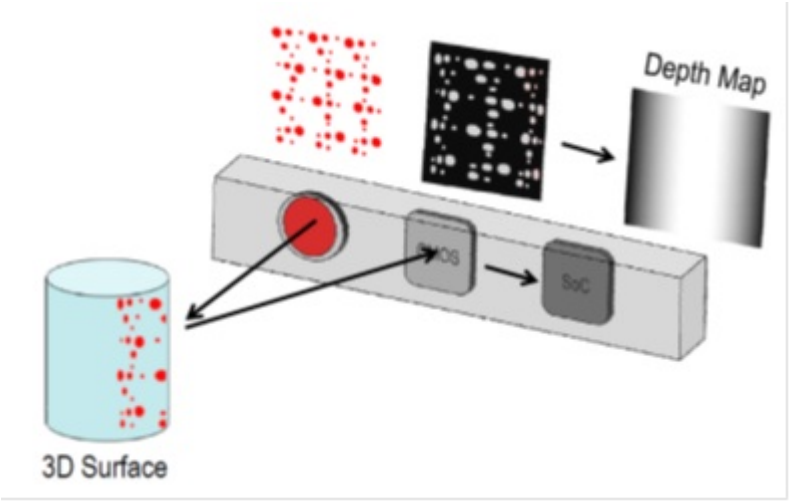


Figura 48. Principio de funcionamiento de las cámaras tipo Kinect, tomada de Depth Biomechanics

La paquetería *Openni2_launch* de ROS [37] viene con una calibración de fábrica que para aplicaciones sencillas es más que suficiente. Pero es recomendable aplicar una calibración adicional para reducir los errores provocados por la distancia focal y distorsión del lente. La calibración se hace por medio del nodo *cameracalibrator.py* [36] la cual se basa en la detección de un patrón de ajedrez con dimensiones conocidas. El primer paso de la calibración es ejecutar *cameracalibrator.py* con la imagen en color y mover el patrón de ajedrez por la imagen. Una vez que el programa informe que tiene información suficiente entonces se detiene la calibración. El segundo paso consiste en tapar el proyector de luz infrarroja y exponer el patrón de ajedrez, con *cameracalibrator.py,* a otra fuente de luz infrarroja, con el fin de no afectar la imagen infrarroja por el patrón de proyección. Se repite la misma acción de mover el patrón de ajedrez a través de la imagen hasta que el programa tenga la información suficiente para determinar una calibración apropiada.

Se hace uso del nodo *depthimage_to_laserscan* de ROS [35] para convertir una imagen de profundidad a una lectura de escáner de láser. El programa *depthimage_to_laserscan* se subscribe a un tópico de la imagen de profundidad y publica un mensaje del tipo *LaserScan,* el cual contiene información de un escáner

láser. Esto quiere decir que crea una lectura de sensor escáner láser falsa basada en una imagen de profundidad. Se pueden definir los parámetros de la dimensión de la franja de la imagen de profundidad que usará para realizar la conversión, el tiempo de muestreo, el rango mínimo, el rango máximo y el sistema de referencia asociado. Se usaron los siguientes parámetros, basados en las especificaciones de la cámara:

Parámetro	Valor
scan_height	10 píxeles.
output_frame_id	camera_depth_frame
range_min	0.45 m
range_max	6.0 m
scan_time	0.07 s

Tabla 9. Parámetros usados en el nodo depthimage_to_laserscan

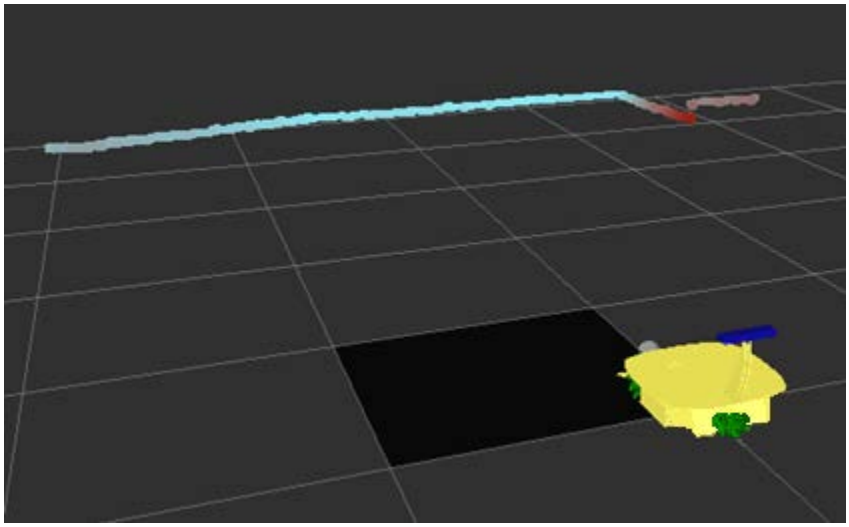


Figura 49. Visualización de la información del sensor Asus Xtion Pro Live en RVIZ después de ser convertido de imagen de profundad a escáner láser.

Un archivo *launch* en ROS brinda la posibilidad de ejecutar varios Nodos al mismo tiempo, pero los Nodos tienen que ser lo suficientemente robustos para que no importe el orden de ejecución siempre se ejecuten correctamente porque el archivo *launch* no tiene un orden de ejecución definido. El archivo *launch* también puede incluir otros archivos *launch* y ejecutarlos junto con los otros nodos.

Se escribe un archivo *launch* que contiene el archivo *launch* para la cámara (*openni2.launch*) y el nodo *depthimage_to_laserscan,* el cual se ejecuta con la siguiente instrucción en la Terminal:

roslaunch om_bot camera_laser.launch

4.4 Teleoperación

La teleoperación es el conjunto de tecnologías que comprenden la operación a distancia de un dispositivo por un ser humano. Un sistema de teleoperación es aquel que permite a un ser humano operar un dispositivo remoto.

En el caso del Robot Móvil Omnidireccional se cuenta con un mando *joystick* de PlayStation, modelo DualShock 3, el cual puede conectarse inalámbricamente por medio de bluetooth. El *joystick* cuenta con un sensor de movimiento de seis ejes, dos palancas analógicas de 10 bits de precisión, dos botones analógicos, seis botones de selección sensibles a la presión, cuatro botones de dirección sensibles a la presión y cinco botones digitales.

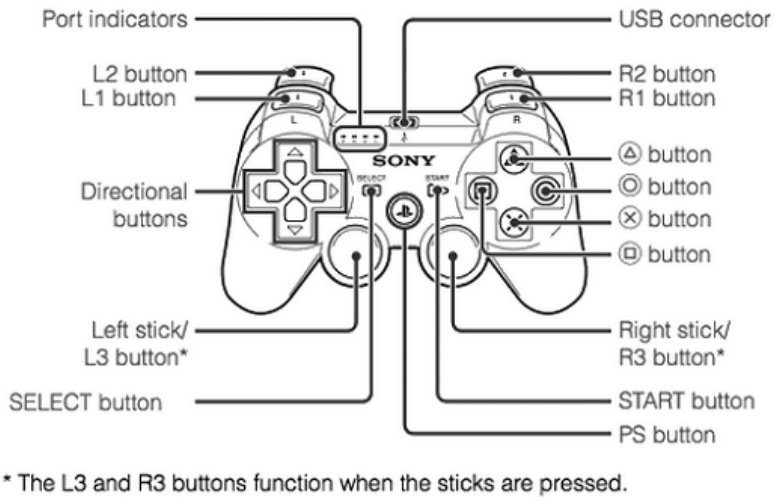


Figura 50. Mapeo de los botones del joystick Dual Shock 3. (kompyuteran, 2014)

Se decide usar la paquetería de *teleop_twist_joy* [38], la cual hace uso de algún mando *joystick* para republicarlos como mensajes tipo *Twist* para el control del robot. En sus parámetros es posible definir un botón para la habilitación de la teleoperación, el eje de las palancas para controlar cada uno de los ejes de velocidad lineal y la velocidad angular del robot así como definir las velocidades

lineales máximas y la velocidad angular máxima. Se realiza además una modificación al código fuente del programa para que pueda publicar velocidades lineales laterales porque la paquetería solo permite publicar velocidades para una base móvil de dos grados de libertad.

Se define a la palanca izquierda para definir la velocidad de traslación hacia adelante, atrás y lateralmente. Debido a la versatilidad de la palanca analógica del mando fácilmente se pueden combinar las velocidades en los diferentes ejes para lograr movimientos en diagonal. La palanca derecha se define para los movimientos de rotación pura.

El uso combinado de las palancas, derecha e izquierda, permiten tener movimientos de traslación y rotación combinados; mientras que el uso independiente de cada una de ellas solo permite la traslación pura (hacia adelante, hacia atrás, lateral y diagonal) o rotación pura.

Las velocidades para la teleoperación se definen a través de la cinemática directa aplicando las velocidades máximas de operación permitidas y manteniendo una velocidad moderada. Por lo cual se definen las siguientes velocidades para la teleoperación:

Velocidad lineal máxima: 0.2 $^{m}/_{s}$

Velocidad lineal turbo: 0.5 $^{m}/_{s}$

Velocidad angular máxima: 1.0 $^{rad}/_{s}$

Para ejecutar el modo de teleoperación se ejecuta la siguiente instrucción en la terminal:

roslaunch teleop_twist_joy teleop.launch

4.5 Formato de Descripción Unificado de los Robots

ROS cuenta con una paquetería intérprete del formato URDF. URDF, por sus siglas en inglés, es un Formato Unificado de Descripción del Robot (Unified Robot Description Format). Este formato se escribe en formato xml y representa el modelo del robot, que en otras palabras se interpreta como la descripción física del robot, dimensiones y relaciones mecánicas entre cada una de sus partes.

El archivo URDF contiene la información de los eslabones del robot, masas, inercias, así como su representación en modelo sólido para análisis de colisiones.

En un archivo URDF también es importante definir el tipo de articulación o junta que une cada uno de los eslabones. La articulación puede ser:

- *Revolute* – Gira en torno a un eje con un rango limitado especificado.
- *Continuous* – Similar a la de revolución con la diferencia de que no está limitada en cuanto puede girar.
- *Prismatic* – Se desliza a lo largo de un eje y tiene un rango de movimiento limitado especificado.
- *Fixed* – No puede moverse.
- *Floating* – Tiene 6 grados de libertad.
- *Planar* – Permite el movimiento en un plano perpendicular al eje.

(ROS Wikipedia, 2013)

En el caso del Robot Móvil Omnidireccional se definió un eslabón base, el cual es el cuerpo del robot. El eslabón base tiene como eslabones *hijos* cada una de las llantas, la base de la cámara RGB-D y la pala en la parte frontal.

Se define como juntas continuas a las uniones entre el eslabón base y las llantas del robot por el movimiento natural de una llanta la cual no tiene límites en su movimiento rotacional. En el caso de las juntas para la pala y la cámara, con respecto al eslabón base, se definen como fijas, lo que quiere decir que solo tendrá una transformación fija entre estas.

La ventaja de definir el formato de descripción del robot es que nos permite simular el robot en un ambiente virtual con las herramientas que proporciona ROS. Las herramientas con las que se puede simular el robot son RVIZ y Gazebo.

RVIZ y Gazebo nos permiten visualizar en un mundo virtual el robot y su interacción con objetos virtuales así como la lectura de sus sensores, ya sean estos simulados o lecturas reales.

Figura 51. Iconos del software RViz y GAZEBO para simulación de robots

Otra ventaja de realizar la descripción URDF es que se puede utilizar un nodo que lee el formato URDF para transmitir las transformaciones entre cada una de las articulaciones del robot. La paquetería *robot_state_publisher* permite publicar el estado de las articulaciones de un robot a través de la paquetería *tf* [39]. La paquetería *tf* permite llevar un control de las transformaciones homogéneas de cada uno de los eslabones respecto a los otros, si es que se encuentra su transformación definida. De esta manera se evita tener que programar recurrentemente las matrices de transformación homogénea. A continuación se muestra el diagrama de relaciones de eslabones del Robot Móvil Omnidireccional.

Se escribe un archivo *launch* que incluye la descripción del robot, el controlador de la base Arduino y el *launch camera_laser.launch* para que al ejecutarse la instrucción se inicien todos los sistemas principales del robot. Entonces, la instrucción en terminal para iniciar todos los sistemas del robot es:

roslaunch om_bot om_botlobster.launch

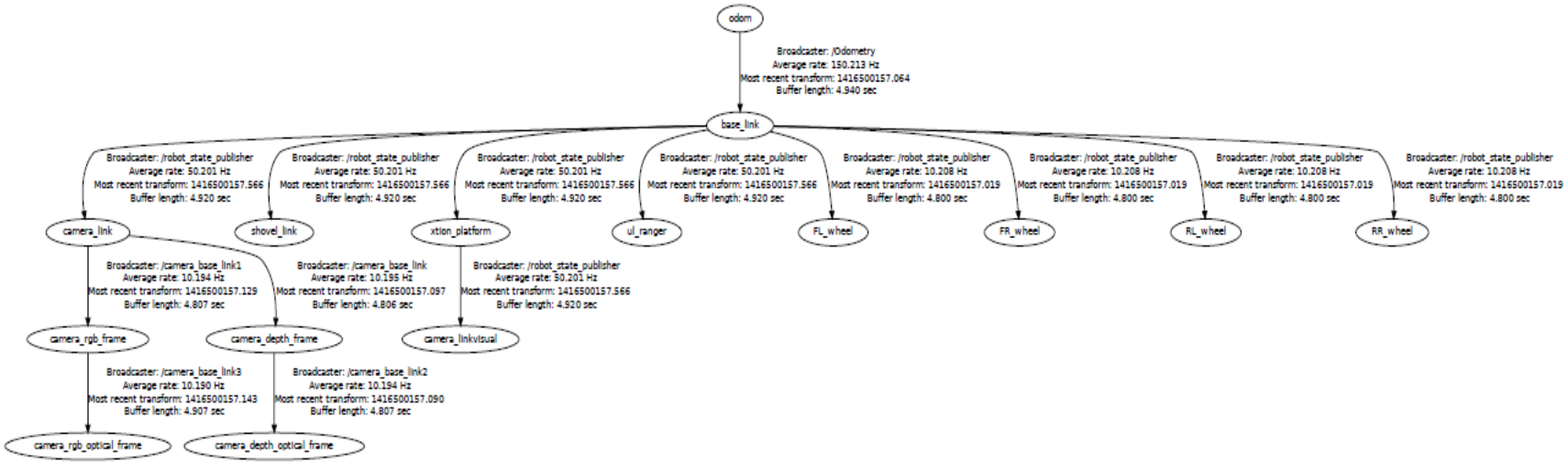


Figura 52. Diagrama de relaciones de los sistemas de referencia de los eslabones del Robot Móvil Omnidireccional.

4.6 Esquema de Nodos y tópicos

Una vez que se tienen todos los Nodos programados y ejecutándose es posible observarlos mediante una herramienta implementada en ROS. Se muestra a continuación el esquemático de los Nodos principales de operación y su relación entre ellos a través de los diferentes tópicos del Robot Móvil Omnidireccional.

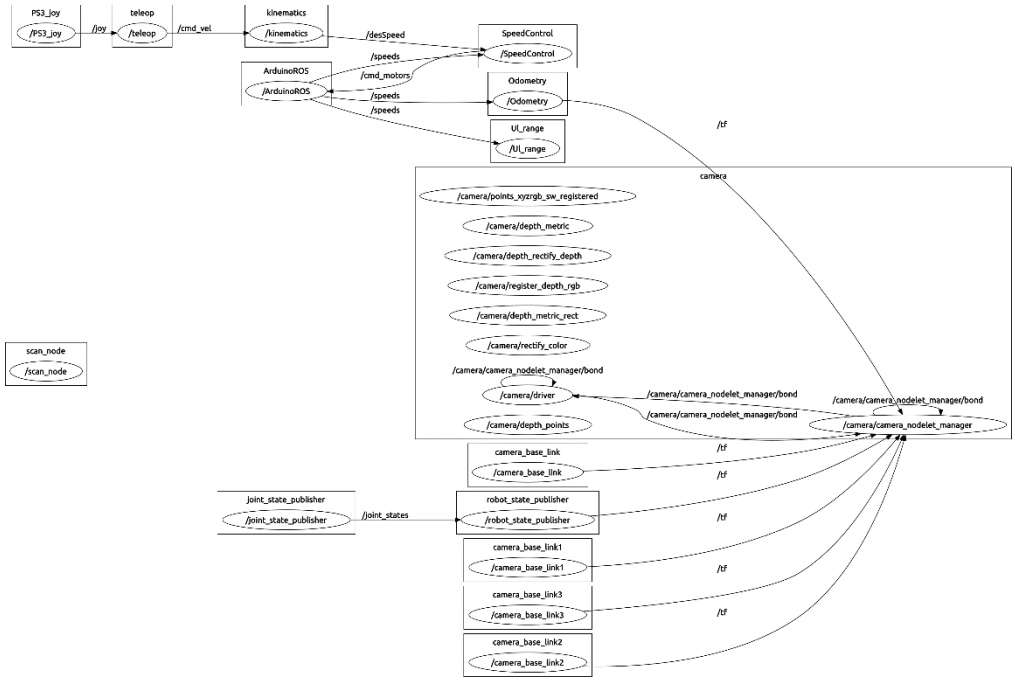


Figura 53. Esquema de nodos y tópicos principales del Robot Móvil Omniireccional.

4.7 Acceso Remoto SSH

El Robot Móvil Omnidireccional cuenta con una computadora en su interior con el fin de poder ejecutar algoritmos complejos de navegación, análisis de imágenes, etc. Esta computadora puede conectarse a un monitor pero cuando se usa el Robot Móvil Omnidireccional no es posible porque el Robot estaría atado a un lugar fijo. El Robot Móvil Omnidireccional no tiene una pantalla externa donde se pueda ver su estado, solo cuenta con tres LEDs indicadores para ver si el interruptor general esta encendido, si la computadora se encuentra encendida y si se encuentra trabajando el disco duro. La computadora cuenta con conexión WiFi por lo que es posible crear una conexión remota a la computadora del robot

para ejecutar los programas que se deseen y para cargar los programas fuente que se requieran.

El protocolo *Secure Shell* (SSH) es una interfaz de comandos y protocolo para acceder remotamente a una computadora. Se puede utilizar para iniciar sesión en alguna plataforma (*slogin),* interfaz remota de comandos (*ssh*) o transferir archivos con seguridad (*scp*). La conexión se realiza mediante TCP, lo cual garantiza que los datos llegarán correctamente a su destino. El cifrado de SSH utiliza llaves para la autentificación de la máquina remota.

En el caso del Robot Móvil Omnidireccional se usa OpenSSH. OpenSSH es un cliente SSH incluido en Ubuntu por default. Después, se crean las llaves en la computadora desde la cual se ejecutarán de manera remota, que llamaremos la estación de trabajo*,* los comandos con el fin de dejar de introducir el usuario y contraseña cada vez que se requiera conectarse remotamente.

Se configura el *Ros Master* para ejecutarse en la computadora del Robot Móvil Omnidireccional, como se mencionó anteriormente *Ros Master* coordinará a los nodos en un mismo registro de nombre para publicar y subscribirse a diferentes tópicos*.* Entonces, si además se ejecutan nodos en la estación de trabajo, será posible comunicarse con los nodos que se ejecutan en la computadora del Robot Móvil Omnidireccional. Como regla general se ejecutan los nodos que tienen una interfaz gráfica en la estación de trabajo y los nodos que tienen

Capítulo 5. Localización y Mapeado Simultáneo

5.1 Introducción

La Localización y Mapeado Simultáneo (*SLAM – Simultaneous Localization And Mapping)* se refiere a la acción que tiene que llevar a cabo un robot móvil para la creación de un mapa de un ambiente desconocido al mismo tiempo que se localiza en este mapa [40]. El problema de *SLAM* también tiene que ver con la interpretación de los sensores y la representación del mapa. Para construir un mapa es necesario que el robot disponga de por lo menos un sensor de distancia, el cual puede ser desde un sensor ultrasónico hasta un escáner láser.

El problema del *SLAM* también se hace presente en el aspecto computacional y se debe a que conforme el robot avanza, va recibiendo mucha información a través de sus sensores. Esa información la tiene que interpretar y analizar para reconstruir un mapa a su vez que se localiza en él. Es por esto que la manera en que se ha abordado últimamente el problema del *SLAM* es con técnicas probabilísticas [41]. Las técnicas probabilísticas aplicadas a la robótica pueden lidiar con las fuentes de incertidumbre del proceso.

Los factores más importantes para el mapeado son [42]:

- El tamaño: Entre más grande sea en comparación con el rango del sensor de distancia más difícil será construir un mapa.
- El ruido en la percepción y actuación: Es un problema más difícil entre más ruido exista.
- Ambigüedad perceptual: Es difícil establecer relaciones espaciales si lugares diferentes se ven iguales.
- Ciclos: Cuando se cierra un circuito en un mapa es difícil cerrar sin errores debido a que el error acumulado en la odometría puede ser muy grande.

Suponiendo que se tenga una medida exacta de odometría, sin errores sistemáticos[5] o no sistemáticos[6], entonces al recibir la lectura del sensor de distancia es posible construir un mapa sin muchas complicaciones. Debido a que en la práctica existen errores odométricos de diferentes fuentes se resuelven los

[5] Errores provocados por usar diferentes tamaños de ruedas, desalineación de las ruedas, muestreo limitado en el *encoder* o resolución limitada del *encoder* [43]

[6] Los errores no sistemáticos se deben a navegar en suelos irregulares o deslizamiento de las ruedas [43].

problemas del *SLAM* basándose en el principio básico probabilístico del teorema de Bayes de la probabilidad condicional [42].

Descrito brevemente el teorema de Bayes permite el cálculo de la probabilidad posteriori, es decir de la probabilidad de un evento basado en la probabilidad conocida de un evento anterior (priori).

El teorema de Bayes describe que para dos variables aleatorias A y B cuyas probabilidades son $p(A)$ y $p(B)$ entonces para la probabilidad condicional de A dado B se tiene:

$$p(A|B) = \frac{p(B|A)p(A)}{p(B)}$$

Ecuación 14

Donde:

$p(A|B)$ – es la probabilidad condicional de A dado B.

$p(B|A)$ – es la probabilidad condicional de B dado A.

$p(A)$ – es la probabilidad de A.

$P(B)$ – es la probabilidad de B.

(T. Bayes, 1763)

Por lo anterior se puede aplicar este teorema a la información que se recibe de los sensores y estimar la probabilidad de que el robot se encuentre en la posición que se estima basado en su odometría.

En desarrollos recientes en algoritmos de *SLAM* para robótica móvil se han usado mapas de ocupación de celdas [45]. Los mapas de ocupación de celdas se basan en la discretización del espacio en unidades de tamaño fijo. Cada una de las celdas que componen el mapa tienen asignado un valor dependiendo de si se encuentran ocupadas o vacías con un nivel de probabilidad de confianza. La precisión de estos mapas depende de que tan finas (pequeñas) son las unidades que describen este espacio.

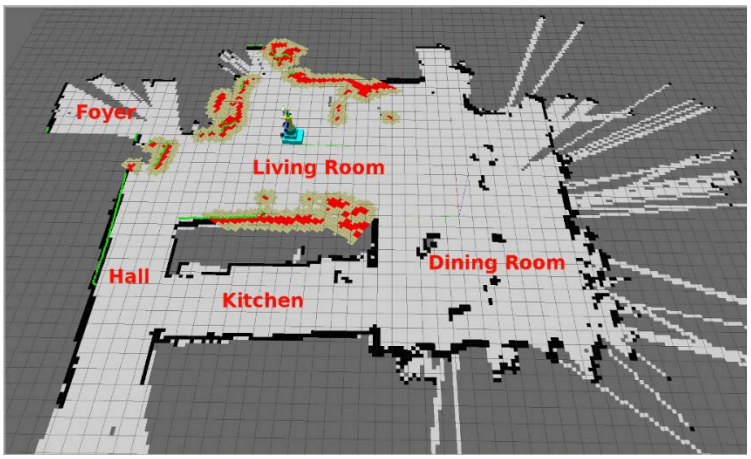


Figura 54. Mapa de ocupación de celdas creado con el robot Pi robot, imagen tomada de www.pirobot.org.

5.2 GMapping

Uno de los algoritmos más populares para la tarea de *SLAM* es el algoritmo llamado *GMapping* [46]. Este algoritmo es muy usado en robots para exploración en edificios y oficinas, de hecho es el algoritmo que utiliza el robot PR-2 para realizar *SLAM* [47]. Este algoritmo fue desarrollado con el fin de mejorar los algoritmos actuales de *SLAM* basados en el filtro de partículas, siendo las partículas cada una de las muestras (hipótesis) que realiza el robot con su sensor, *Rao-Blackwellized.*

El algoritmo *GMapping* es un filtro de partículas *Rao-Blackwellized* para crear mapas de celdas a partir de un sensor de distancia de láser.

El filtro de partículas *Rao-Blackwellized* aplicado al *SLAM* estima un estado posterior $p(x_{1:t}|z_{1:t}, u_{0:t})$ de las potenciales trayectorias $x_{1:t}$ del robot dadas sus observaciones $z_{1:t}$ y su odometría $u_{0:t}$ y después usarlo para calcular un estado posterior sobre mapas y trayectorias $p(x_{1:t}, m|z_{1:t}, u_{\cdot 0:t}) = p(m|x_{1:t}, z_{1:t})p(x_{1:t}|z_{1:t}, u_{0:t})$ [48]. Este filtro de partículas se asocia un mapa a cada partícula y se le asigna un peso para decidir su distribución. Las partículas con poco peso son reemplazadas, en un nuevo muestreo, por partículas con mayor peso. Posteriormente se estima un mapa.

El algoritmo *GMapping* utiliza un filtro de partículas *Rao-Blackwellized* aplicando una aproximación de la distribución alrededor del máximo de la función de probabilidad usando un Gaussiano, mejorando así el algoritmo original. Además, usan técnicas de re-muestreo selectivas.

La paquetería *GMapping* de ROS [49] es un *wrapper* de la librería *GMapping* de Open*SLAM,* que significa que crea un vínculo entre la librería de *GMapping* y ROS para poder usarlo con todas sus funciones.

Los requerimientos de la paquetería para poder funcionar es que exista un sensor de distancia y mediciones odométricas; como datos de salida tiene el mapa creado, y la transformación entre el mapa y el sistema de referencia odométrico.

Los parámetros importantes que se tienen que definir, dependiendo de la estructura del robot móvil son los siguientes:

Sensor láser

- maxUrange: El máximo rango del sensor.

Modelo Odométrico

- srr: Error Odométrico en la traslación como función de traslación.
- srt: Error Odométrico en la traslación como función de rotación.
- str: Error Odométrico en la rotación como función de traslación.
- stt: Error Odométrico en la rotación como función de rotación.

Modelo Probabilístico

- linearUpdate: Cada cuanta distancia se procesan nuevos datos.
- angularUpdate: Cada cuanta rotación se procesan nuevos datos.
- particles: Numero de partículas.

Dimensiones del mapa

- xmin: mínima posición x en el mapa [m]
- ymin: mínima posición y en el mapa [m]
- xmax: máxima posición x en el mapa [m]
- ymax: máxima posición y en el mapa [m]
- delta: tamaño de un pixel [m]

Se define un tamaño del mapa de 10x10 [m], lo cual no limita a que el tamaño total del mapa sea de 10x10 [m] sino que solo es el tamaño inicial. Debido a la potencia de la computadora interna del robot, que es donde se lleva a cabo el proceso de mapeado se define un tamaño de 0.05 [m] por pixel, es decir que cada celda tiene un tamaño de 0.05x0.05 [m]. El rango del sensor de delimita a 6 [m] por sus delimitaciones de hardware.

Se realizan pruebas sencillas con el robot para determinar los errores del modelo odométrico del algoritmo de mapeado. Lo que se hace es comandar al robot para que se mueva 1 [m] linealmente hacia el frente sin girar acorde a su odometría. Después, se mide la distancia que avanzó realmente y si hubo un giro. Hecha esta prueba se define su error odométrico en rotación y traslación como función de traslación. Posteriormente, se realiza una prueba donde se hizo girar una vuelta completa, sin avanzar de acuerdo a su odometría. Se mide su orientación real final y, en caso de existir, su traslación. Terminada esta segunda prueba se puede determinar su error odométrico en traslación y rotación en función de su rotación. El resultado de hacer estas pruebas es el siguiente:

$$srr = 0.13$$
$$srt = 0.05$$
$$str = 0.12$$
$$stt = 0.1$$

Se usan los valores de 0.12 para *linearUpdate* y de 0.4 para *angularUpdate* para el proceso de nuevos datos para mejorar la calidad de los mapas creados. Se eligen 30 partículas, las cuales son suficientes para una calidad de un mapa aceptable [48].

Capítulo 6. Navegación

6.1 Introducción

La navegación en robótica móvil se refiere a la planeación de movimientos para moverse de una posición a otra. Esta planeación de movimientos tiene que ser capaz de calcular las velocidades durante su movimiento y tener la capacidad de evadir obstáculos. El problema de la navegación en robótica móvil se ha abordado de diferentes maneras como la planificación basada en grafos de visibilidad [50], planificación basada en diagramas de Voronoi [51], planificación basada en el espacio libre [52], entre otros. En este capítulo se abordará la navegación basada en un mapa de celdas y su función de costo.

6.2 Algoritmo de localización *AMCL*

Una parte esencial para la navegación del robot es tener el conocimiento de su localización en el ambiente. En el capítulo anterior se abordó el tema del *SLAM* para la creación de mapas. Una vez que se obtiene el mapa es necesario tener la habilidad de ubicarse en éste. Una vez más la probabilidad en robótica aborda este tema debido a los errores en los sistemas odométricos.

El algoritmo de localización *AMCL* viene de las siglas de *Adaptative Monte Carlo Localization* que significa Localización Adaptativa Monte Carlo, el cual es usado para localizar un robot en un mapa. Entonces es necesario tener un mapa del ambiente cuando se pone en práctica este algoritmo.

AMCL se basa en los algoritmos *sample_motion_model_odometry, beam_range_finder_model, likelihood_field_range_finder_model, Augmented_MCL y KLD_Sampling_MCL* [53]. El algoritmo *simple_motion_model_odometry* se encarga del modelo de probabilidad de odometría, el algoritmo *beam_range_finder_model* se encarga del modelo probabílistico del sensor, el *likelihood_field_range_finder_model* calculará la correspondencia de la percepción del sensor y las observaciones previas, *Augmented_MCL* realiza las estimaciones de la posición del robot y *KLD_Sampling_MCL* mejora el algoritmo anterior al realizar una ponderación de la probabilidad de posición estimada mediante el uso de la distribución normal. Su desarrollo matemático específico puede ser consultado en [42].

AMCL toma en consideración el modelo odométrico del robot ya sea diferencial u omnidireccional. *AMCL* se basa en lecturas de un escáner láser y su función probabilística de estimación de localización. Utiliza una función de probabilidad basada en partículas, las cuales esparce a su alrededor para hacer predicciones que serán evaluadas en su precisión y el peso calculado.

La paquetería de ROS *AMCL* permite usar el algoritmo cumpliendo con los requerimientos de información que necesita para funcionar correctamente. Para poder usar esta paquetería es necesario tener una fuente de información odométrica, un sensor escáner láser, y un mapa.

Los parámetros que se pueden modificar, dependiendo de las características del robot en uso, son:

- Parámetros del filtro: *min_particles, max_particles, kld_error, kld_z, update_min_d, update_min_a, resample_interval, transform_tolerance, recovery_alpha_slow, recovery_alpha_fast, initial_pose_x, initial_pose_a, initial_cov_xx, initial_cov_yy, initial_cov_aa, gui_publish_rate, save_pose_rate.*
- Parámetros del modelo del láser: *laser_min_range, laser_max_range, lase_max_beams, laser_z_hit, laser_z_short, laser_z_max, laser_z_rand, laser_sigma_hit, laser_lambda_short, laser_likelihood_max_dist* y *laser_model_type.*
- Parámetros del modelo odométrico: *odom_model_type, odom_alpha1, odom_alpha2, odom_alpha3, odom_alpha4, odom_alpha5, odom_frame_id, base_frame_id* y *global_frame_id.*

Los parámetros que se modificaron de los parámetros por defecto, por las características del Robot Móvil Omnidireccional, son:

- Parámetros del filtro:
    - *min_particles*: número mínimo de partículas permitidas.
    - *max_particles*: número máximo de partículas permitidas.
    - *update_min_d*: movimiento traslacional requerido antes de actualizar las partículas*,*

- *update_min_a*: movimiento rotacional requerido antes de actualizar las partículas
- *resample_interval*: número de actualizaciones del filtro antes de hacer otro muestreo.
- *transform_tolerance*: tiempo de espera para una nueva transformación posfechada.

- Parámetros del modelo del láser:
  - *laser_min_range:* distancia mínima que puede medir el sensor.
  - *laser_max_range:* distancia máxima que puede medir el sensor.
- Parámetros del modelo odométrico:
  - *odom_model_type:* define si es un robot diferencial u omnidireccional.
  - *odom_alpha1*: es el ruido estimado en la rotación causada por el movimiento rotacional.
  - *odom_alpha2*: el ruido estimado de rotación causado por un movimiento traslacional.
  - *odom_alpha3*: el ruido estimado de traslación causado por un movimiento traslacional.
  - *odom_alpha4*: el ruido estimado de traslación causado por un movimiento rotacional.
  - *odom_alpha5*: el ruido estimado de la tendencia del robot móvil a moverse perpendicularmente a su dirección de movimiento. Este parámetro solo aplica si es un robot omnidireccional.
  - *odom_frame_id:* nombre del sistema de referencia odométrico.
  - *base_frame_id*: nombre del sistema de referencia del robot móvil.
  - *global_frame_id*: nombre del sistema de referencia global.

Después de realizar pruebas con los valores por defecto de partículas y otros valores, se determinó un mínimo de 80 partículas (*min_particles*) y un máximo de 2000 partículas (*max_particles*) son suficientes y se puede mantener un procesamiento rápido en el robot. Se eligieron valores de desplazamiento para

actualización de las partículas de movimiento traslacional cada 0.25 m (*update_min_d*) y en rotación cada 0.2 rad (*update_min_a*), con un remuestreo de partículas cada actualización (*resample_interval*). Se permite una tolerancia de 0.2 segundos (*transform_tolerance*) de validez de transformación posfechada.

En cuanto al láser, como se describe en capítulos anteriores, se especifica un rango mínimo de 0.45 m (*laser_min_range*) y un rango máximo de 6.0 m (*laser_max_range*).

Los valores de estimación de ruido en el modelo odométrico son los mismos que se utilizaron para el *SLAM* en el capítulo anterior. Los nombres de los sistemas de referencia odométricos, robot móvil y sistema global son *odom, base_link* y *map* respectivamente.

6.3 Paquetería de Navegación en ROS

La navegación es la acción que se lleva a cabo para ir de un punto de partida a un punto de llegada, comprendiendo todos los aspectos del sistema requeridos para esta acción así como la superación de obstáculos durante el camino. En el caso de robótica móvil comprende la acción de localización y planeación de trayectorias para llegar a la meta establecida y requiere de una representación del ambiente como un mapa y la interpretación de éste. Las aplicaciones de la navegación en robótica móvil abarcan los robots de servicio [54], robots para el traslado de material en una fábrica [55], manipulador móvil para la casa [56], e inclusive robots de limpieza [57].

La navegación en robótica móvil se divide en cuatro etapas [58]:

- Percepción del mundo: Se refiere a la creación de mapas a partir de lectura de sensores.
- Planificación de la ruta: Es el cálculo de la ruta a seguir por el robot móvil. Esta planificación está conformada por una serie de metas.
- Generación del camino: Se discretiza la ruta generada para generar un camino.
- Seguimiento del camino: Realiza el movimiento del vehículo según el camino generado.

ROS cuenta con una paquetería de navegación que se basa en los mapas de celdas. Su algoritmo de navegación se divide en un Planificador Global y un Planificador Local [59].

El Planificador Global hace uso de un mapa de celdas, la posición inicial del robot en el mapa y la meta del robot. El Planificador Global encontrará una ruta basado en el algoritmo A*. El algoritmo A* es la combinación del algoritmo *Best-First-Search* y *Dijkstra* [60].

El algoritmo *Dijkstra* buscará el camino más corto dado un vértice origen (posición inicial del robot) a otro vértice con pesos en cada arista. Este algoritmo no es óptimo debido a que explora todas las opciones posibles para encontrar el camino más corto. El cálculo del camino más corto inicia desde el vértice origen hasta la meta.

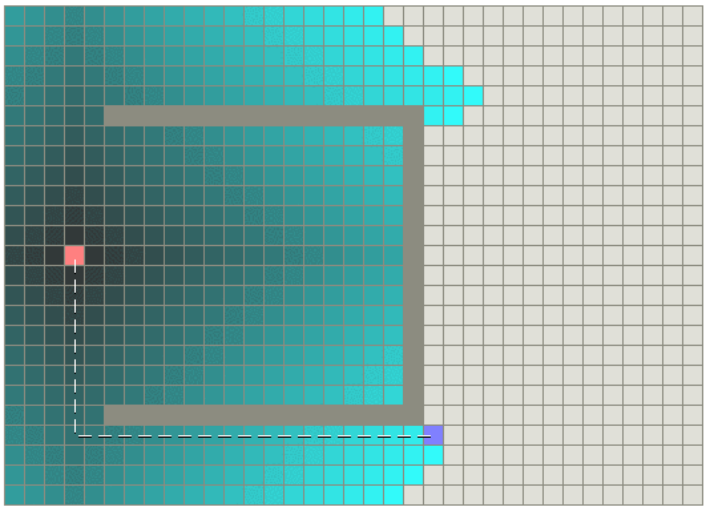


Figura 55. Ruta determinada por el algoritmo Dijsktra. Imagen tomada de http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html

El algoritmo *Best-First-Search* buscará el camino más rápido hacia la meta inclusive si este no es el más corto. Es un algoritmo que usa una función de estimación para determinar qué tan lejos está el vértice de la meta pero no garantiza el camino más corto.

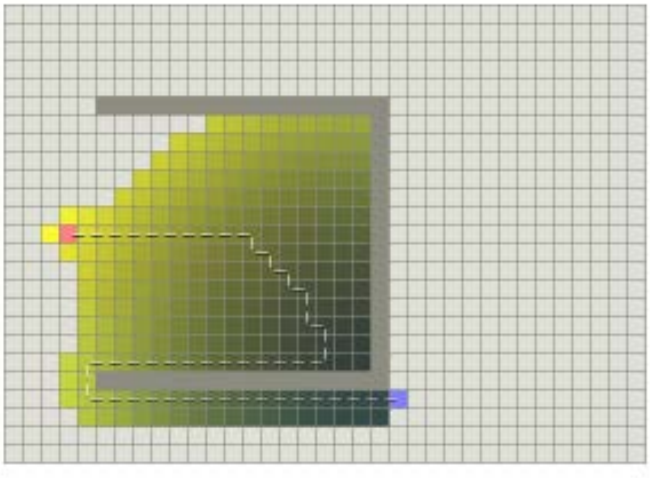


Figura 56. Ruta calculada por el algoritmo Best-First-Search, imagen tomada de http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html

El algoritmo usado por el Planificador Global, el algoritmo A*, usa la combinación de la búsqueda del camino más corto desde el vértice origen hasta la meta y la búsqueda del camino más rápido por un método heurístico, hasta la meta desde el vértice origen, lo que da por entendido que hace uso del algoritmo *Dijkstra* y *Best-First-Search* conjuntamente. El uso de la función heurística hace que el algoritmo sea más rápido que el algoritmo *Dijkstra* y se obtenga un buen resultado [61].

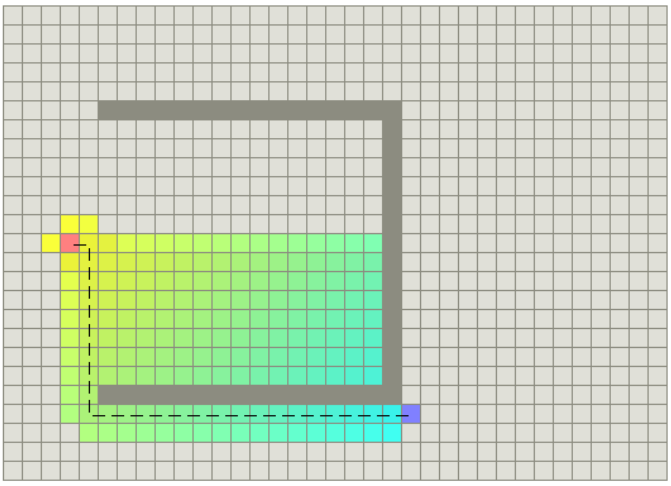


Figura 57. Ruta calculada por el algoritmo A. Imagen tomada de http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html.*

El Planificador Global cuenta con un mapa de costos, es decir que crea zonas donde define aquellas que son de más difícil acceso para el robot que otras. El

mapa de costos global tiene un “radio de inflación” el cual es usado para “inflar” los obstáculos del mapa y calcular la ruta considerando la “inflación”.

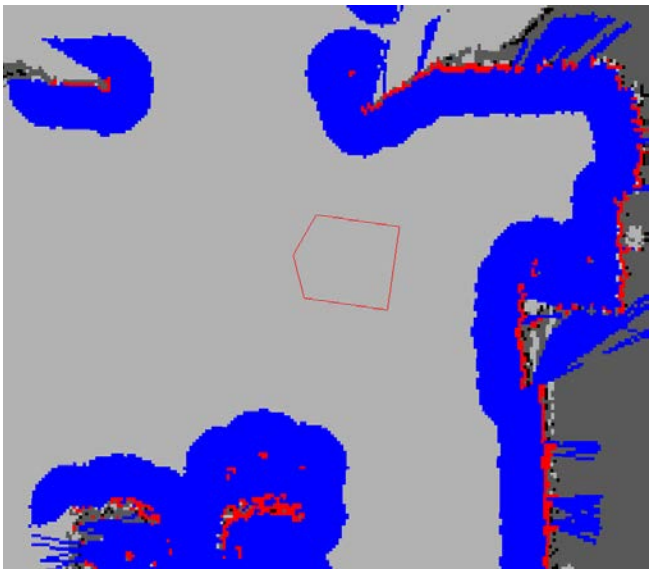


Figura 58. En la imagen se muestra el radio de inflación en color azul mientras un robot, de forma pentagonal, navega en el ambiente. Imagen tomada de http://wiki.ros.org/costmap_2d.

El Planificador Global no toma en cuenta si los pasillos son angostos o la dinámica del robot. Es por esto que la paquetería de navegación contiene un Planificador Local.

El planificador local se encargará del cálculo de las velocidades del robot para seguir una ruta generada por el Planificador Global. La ruta que seguirá no será exactamente la ruta generada, pero la seguirá lo más cercana posible, evitando los obstáculos del mapa y los obstáculos detectados por su sensor. También considera el tamaño del robot y los parámetros dinámicos de velocidades y aceleraciones del robot.

El Planificador Local cuenta con un mapa local dinámico, lo que da a entender que es un mapa que se mueve con el robot, que puede añadir y quitar obstáculos a partir de las lecturas del sensor. Este mapa dinámico es un mapa de costos, el cual considera el mapa de costos Global y los costos añadidos por los obstáculos detectados.

La idea básica del Planificador Local es la siguiente:

1. Discretiza muestras en el espacio de control del robot (dx,dy,dtheta).

2. Para cada muestra de velocidad se realiza una simulación adelantada al estado actual del robot para predecir el comportamiento al aplicar la muestra de velocidad por un corto periodo de tiempo.

3. Evaluar cada trayectoria como resultado de la simulación usando una métrica que le incorpora características como: proximidad a los obstáculos, proximidad a la meta, proximidad al Plan Global y velocidad. Descarta trayectorias ilegales (las que colisionan con los obstáculos).

4. Se escoge la trayectoria con la evaluación más alta y se manda el comando de velocidad al robot móvil.

5. Termina el cálculo y repite.

(Base local Planner documentation, ROS Wikipedia 05/10/2014)

La paquetería de navegación cuenta con comportamientos de recuperación cuando se encuentra varado en algún lugar los cuales se describen en el siguiente diagrama:

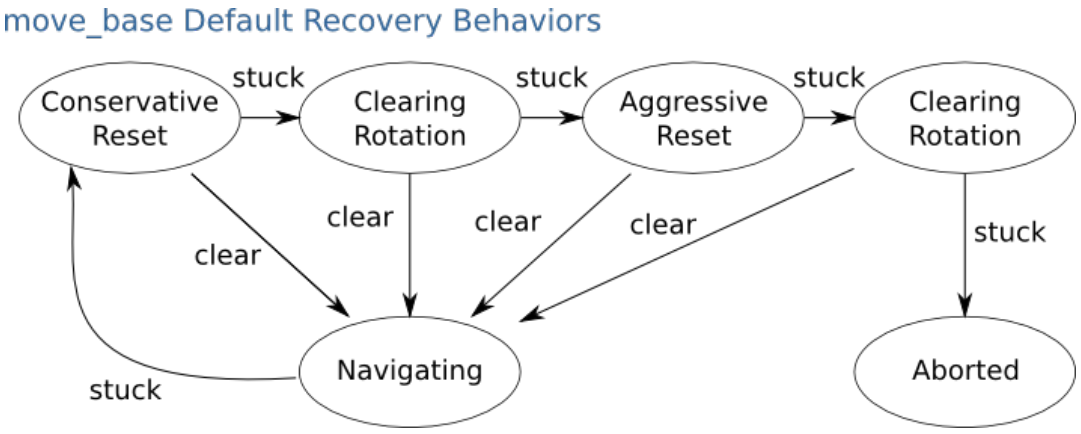


Figura 59. Comportamientos de recuperación. Imagen tomada de http://wiki.ros.org/move_base

Se implementa la paquetería de navegación en el Robot Móvil Omnidireccional. Se definen los siguientes parámetros para los mapas de costos:

Rango de obstáculos (*obstacle_range):* 3.0 m

Rango de eliminación de obstáculos (*raytrace_range):* 3.0 m

Huella del robot (*footprint): [[0.35, 0.2], [0.35, -0.2], [-0.3, -0.2], [-0.3, 0.2]]*

Radio de inflación (*Inflation_radius):* 0.1 m

Se define un tamaño del mapa local de costos de 4x4 m, para no ocupar mucho espacio en memoria activa del robot.

Se definen velocidades relativamente bajas en comparación con la velocidad máxima que puede producir el robot para dar tiempo al procesamiento de trayectorias y evasión de obstáculos al momento de experimentar con el robot. La potencia de la computadora central del robot es similar a las computadoras *netbook.*

Las velocidades y aceleraciones son:

Máxima velocidad x: 0.13 m/s

Mínima velocidad x: 0.11 m/s

Límite de aceleración x: 1.0 m/s^2

Mínima velocidad theta: 0.43 rad/s

Mínima velocidad theta pura: 0.5 rad/s

Máxima velocidad theta: 0.5 rad/s

Máxima velocidad y: 0.13 m/s

Mínima velocidad y: 0.1 m/s

Los parámetros definidos para el alcance de una meta son:

Tolerancia de orientación: 0,3 rad

Tolerancia xy: 0.2 m

Capítulo 7. Pruebas y Resultados

7.1 Descripción

En los capítulos anteriores se ha descrito el análisis matemático de movimiento del robot, sus partes mecánicas, sus circuitos electrónicos de control, la programación del robot en la estructura de software de ROS y su integración usando un Arduino como el controlador de la base móvil[7]. También se ofrece una pequeña descripción del principio básico de funcionamiento de los algoritmos de SLAM y Navegación en robótica móvil, además de la paqueterías usadas para estas tareas y la determinación de sus parámetros correspondientes.

En este capítulo se describen las pruebas realizadas con el robot y un análisis de los resultados obtenidos.

Las pruebas se realizaron en el pasillo que conecta un salón de clases con el laboratorio de robótica y la salida a otras áreas de la universidad de Aalborg. Las pruebas se realizaron sin tener obstáculos móviles, lo que incluye que no hubo interferencia por el paso de personas.

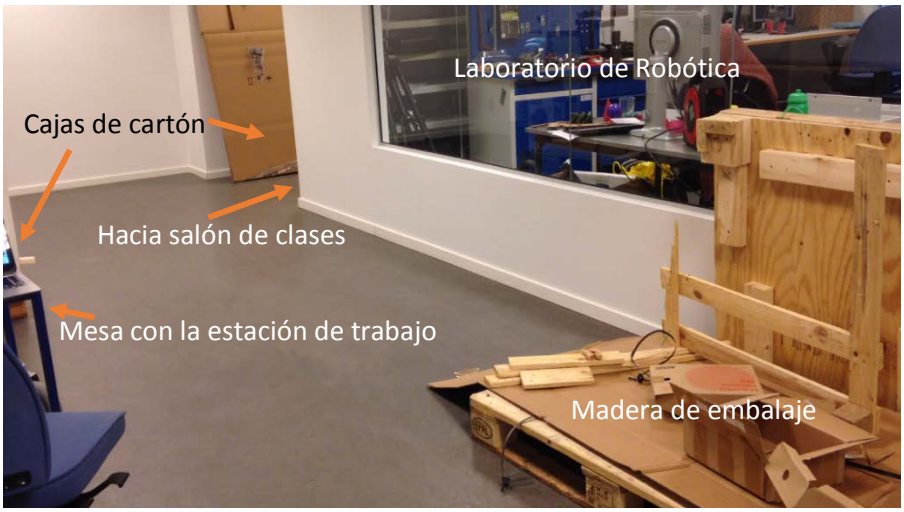


Figura 60. Pasillo que conecta el salón de clases con el laboratorio de robótica.

[7] La base móvil se considera a cualquier robot móvil que recibe comandos de velocidad para desplazarse en su ambiente.

Figura 61. Parte posterior de la imagen anterior. Muestra la entrada al laboratorio de robótica y la salida general.

7.2 Pruebas y Resultados de SLAM

Se realizó la prueba de Mapeado con la paquetería descrita en los capítulos anteriores y con los parámetros calculados. La prueba de SLAM se llevó a cabo en la computadora a bordo del robot y también se guardaron los datos para realizar el mapeado en la estación de trabajo. Para la creación del mapa se utilizó el control de teleoperación con el mando Sixaxis™ de Playstation®, manteniendo una velocidad menor a 0.16 m/s. La imagen siguiente muestra la ruta que siguió el robot durante el mapeado.

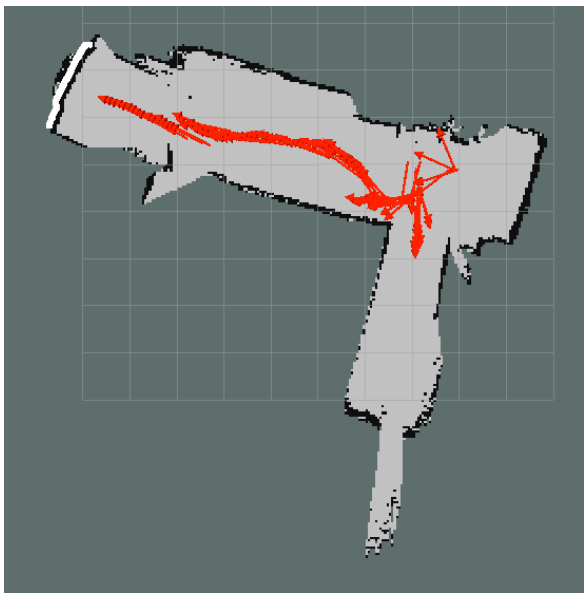


Figura 62. Ruta del robot durante el mapeado. Las flechas rojas indican el recorrido del robot.

A continuación se muestra el mapa creado durante la prueba con la computadora a bordo usando 30 partículas.

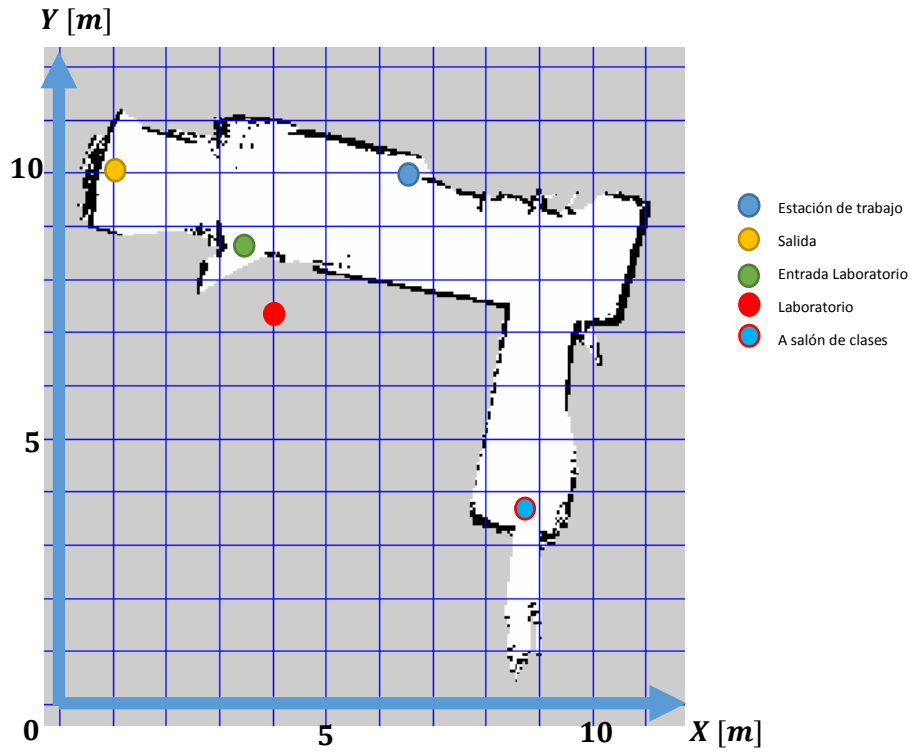


Figura 63. Mapa creado con 30 partículas.

El siguiente mapa se creó con 20 partículas a partir de los datos guardados con el robot. Este mapeado se hizo en la estación de trabajo que cuenta con un procesador Intel i5 y 6gb de memoria RAM lo que permite un procesamiento más rápido.

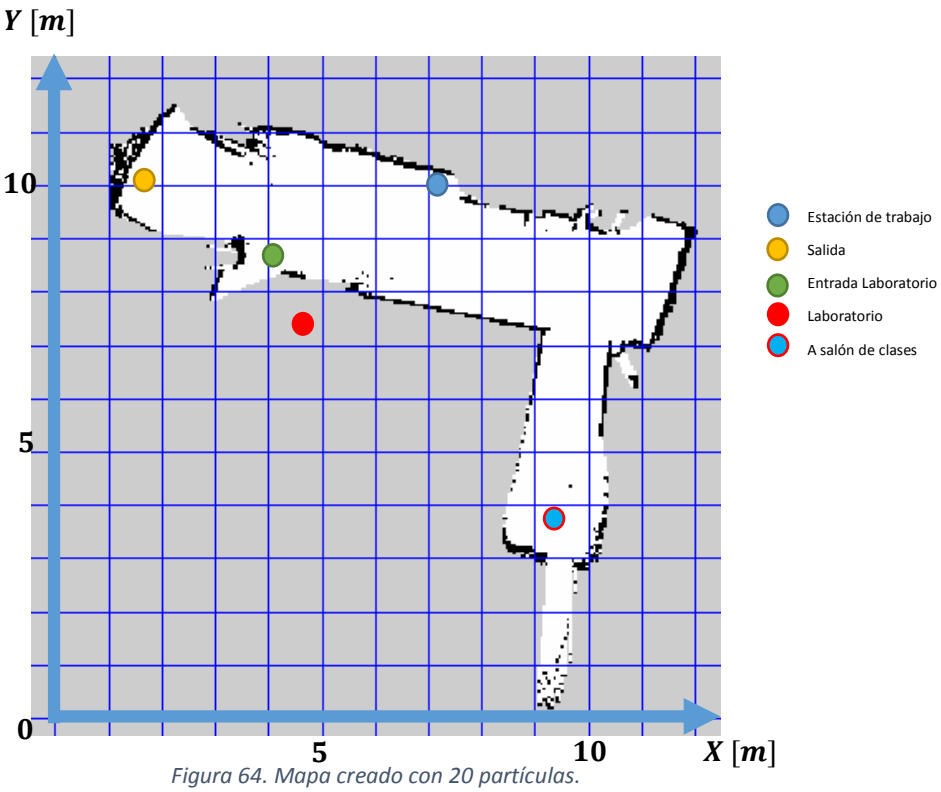


Figura 64. Mapa creado con 20 partículas.

Por último, se crea un mapa usando 200 partículas, es decir que se utilizarán más conjeturas para la estimación del mapa. Este mapa también se crea en la estación de trabajo, cuando el robot no se encuentra en marcha.

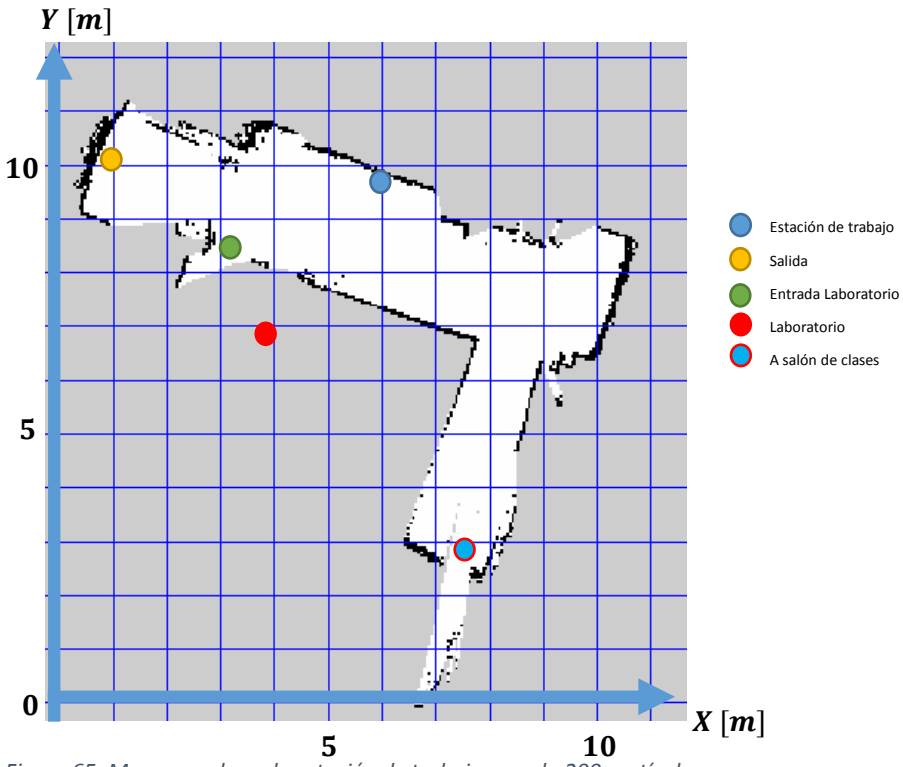


Figura 65. Mapa creado en la estación de trabajo usando 200 partículas.

7.2.1 Discusión

Se crearon mapas con la computadora a bordo y en la estación de trabajo. El robot puede crear mapas con su computadora a bordo mientras es comandado por medio de teleoperación.

El proceso de mapeo se realizó a una velocidad máxima de 0.17 m/s y velocidad angular máxima de 0.5 rad/s para evitar que se incremente el error odométrico debido a una mayor velocidad de movimiento del robot y dar tiempo para que se lleve a cabo el procesamiento correctamente.

El mapa creado con 20 partículas en la estación de trabajo tiene algunas paredes descuadradas, que deberían ser completamente paralelas. En el mapa creado con 30 partículas se puede observar con mejor claridad como la orientación entra las paredes es más concordante. Mientras que en el mapa creado usando 200 partículas se puede ver un mapa más fino.

Los colores de los pixeles que conforman el mapa corresponden a la probabilidad de ocupación que tiene el pixel. Si es de color negro significa que existe una probabilidad de seguridad de que está ocupado, si el color es blanco significa que tiene una probabilidad de seguridad de que está libre y si es de color gris significa que no tiene información de ocupación en esa parte del mapa.

7.3 Pruebas y resultados de navegación

Las pruebas se realizaron con el mapa creado en el apartado anterior con 30 partículas. Cuando se termina el proceso de mapeado se puede guardar el mapa usando el servidor de mapas de ROS [65]. Una vez que se tiene el mapa guardado se puede ejecutar y mandar por medio de mensajes de tipo *map,* que contienen la información del mapa, para ser usado por el programa de localización mencionado en los capítulos anteriores. La paquetería de localización *AMCL* permite una localización global por medio de la dispersión de las partículas en el mapa, en la cual va actualizando su conjetura sobre su posición actual hasta que se localiza. En el caso en que se use una interfaz visual, en este caso *RViz*[8], para indicar cuál es la posición actual del robot.

El procedimiento que se llevó a cabo para la prueba de navegación fue el siguiente:

- Se accedió por medio de *SSH* a la computadora interna del robot.
- Se inicia el controlador de motores y de la cámara, y todos sus programas asociados para la publicación de sus respectivos mensajes.
- Se inician los programas de localización y navegación.
- Se inicia en la estación de trabajo la interfaz visual *RViz.*
- Se indica la posición inicial del robot.
- Se manda por medio de la interfaz visual una meta para que el robot navegue desde su posición inicial hasta la meta.

En el ambiente de operación de robot se colocó una caja de cartón, la cual no estaba modelada en el mapeado. Esta caja representa un obstáculo que el robot debe superar.

[8] Interfaz visual integrada en ROS para visualización de datos de sensores.

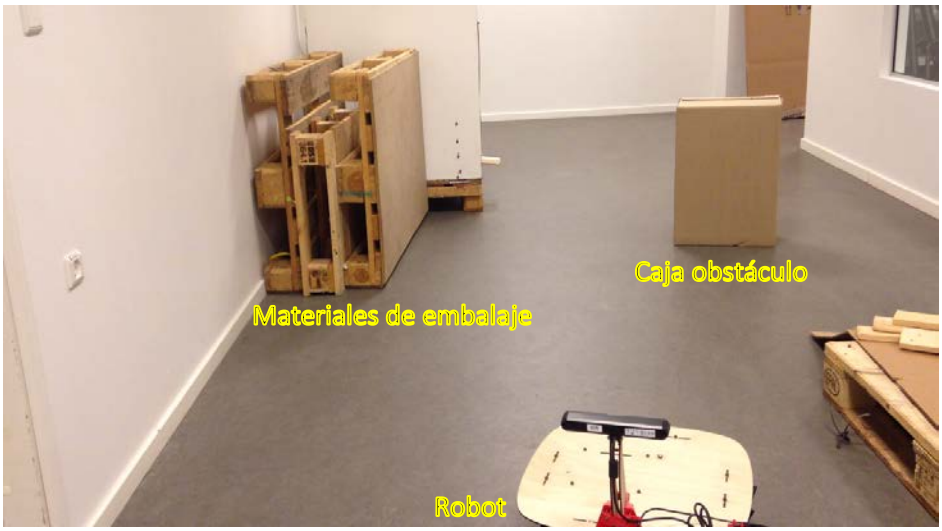


Figura 66. En la imagen se muestra la caja obstáculo en el ambiente de operación del robot.

La ruta que siguió el robot en las dos metas mandadas fue la siguiente:

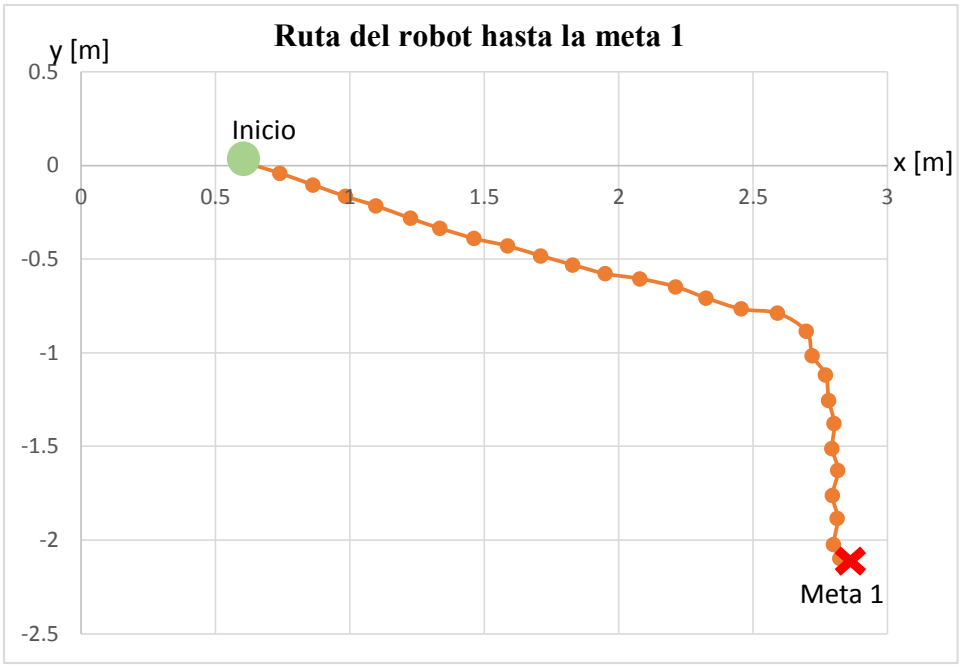


Figura 67. Ruta seguida por el robot hasta la Meta 1.

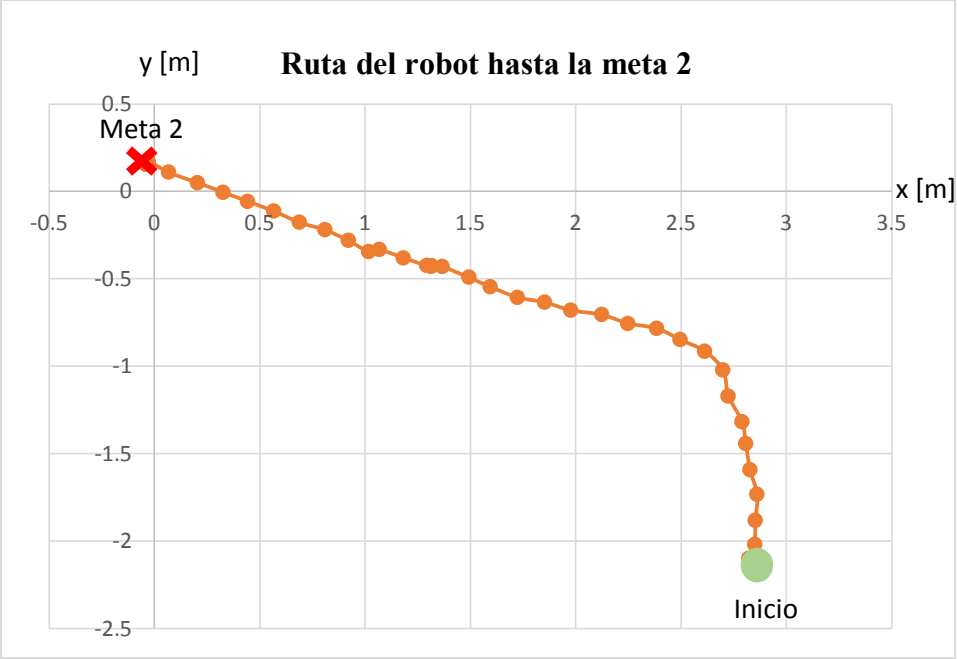


Figura 68. Ruta seguida por el robot hasta la Meta 2.

Durante la navegación se observó como el robot percibió y evadió el obstáculo en su camino.

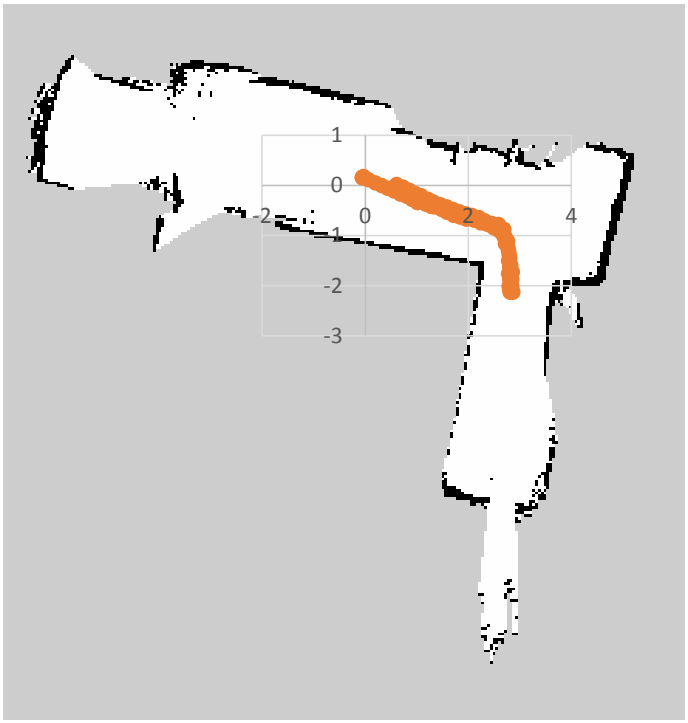


Figura 69. Ruta total seguida por el robot sobrepuesta sobre el mapa que uso como referencia.

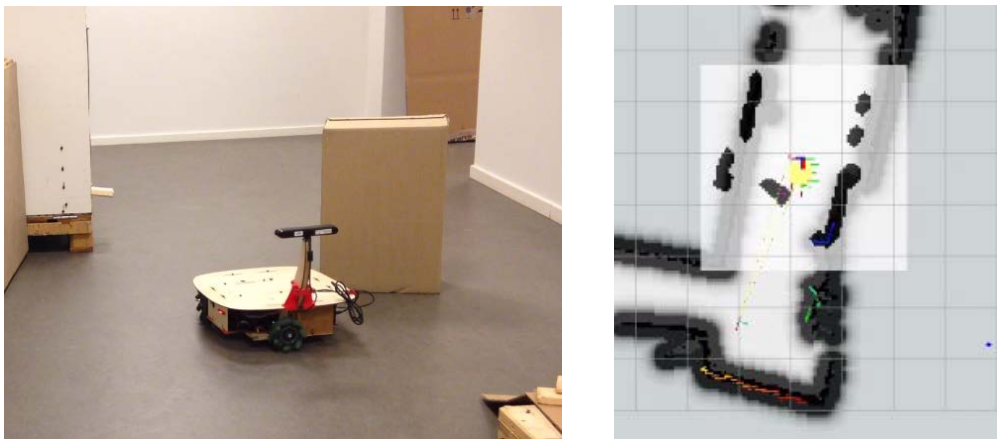


Figura 70. En estas imágenes se muestra al robot evadiendo el obstáculo, a la izquierda en el ambiente real, a la derecha en la interfaz visual RViz.

Durante la ejecución de la navegación hubo errores en la estimación odométrica, los cuales eran considerados por el programa de localización. A continuación se muestran los errores odométricos acumulados durante la navegación.

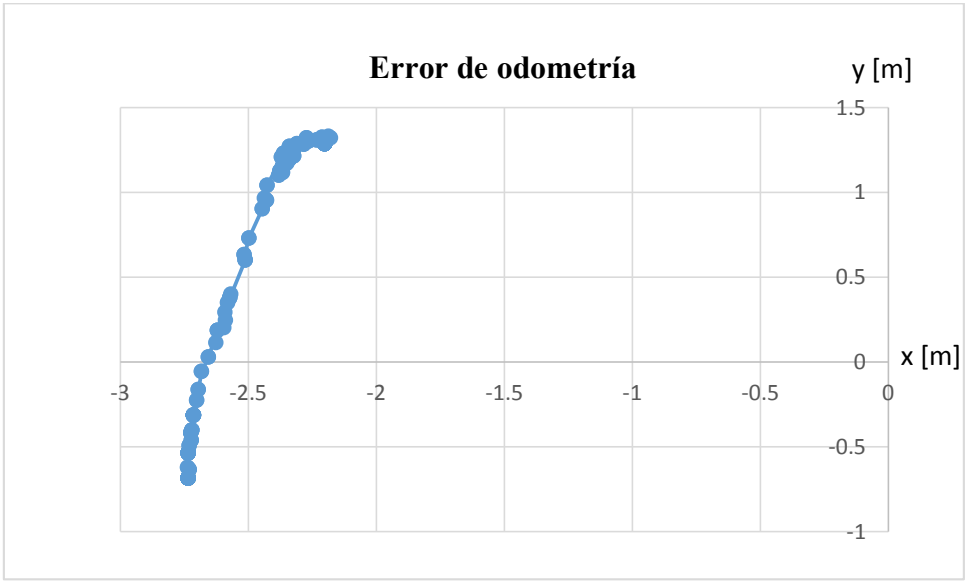


Figura 71. Error acumulado de odometría durante la navegación.

La grafica muestra la transformación del sistema de referencia global al sistema de referencia odométrico que provee el programa de localización. Esta transformación surge como resultado de la localización que hace del sistema de referencia global al sistema de referencia móvil pero para realizar esto tiene que pasar a través de la transformación intermedia de la base del sistema de odometría al sistema de referencia móvil.

7.3.1 Discusión

Se pudo llevar a cabo la navegación usando la paquetería de navegación de ROS. Se mantuvo una velocidad relativamente baja comparada con la velocidad máxima que el robot puede desarrollar para dar tiempo al procesamiento que tiene que llevar a cabo para la localización, navegación y evasión de obstáculos.

La velocidad con la que procesa los datos de navegación y localización dependen directamente de la capacidad de procesamiento del robot. El poder de procesamiento del robot es similar a la capacidad de procesamiento de las computadoras portátiles conocidas como *netbook,* por lo que su velocidad se ve limitada por las características de su computadora interna. A pesar de la velocidad de procesamiento se pudo llevar a cabo la tarea de navegación en una velocidad aceptable.

Capítulo 8. Conclusiones y Trabajo a Futuro

8.1 Conclusiones

La robótica móvil, siendo una rama de la robótica, tiene muchas áreas de desarrollo. Estas áreas incluyen el estudio de la mecánica, de electrónica, de sensores y de control. Últimamente se han hecho muchos estudios y avances en el área de cognición robótica para la ejecución de tareas complejas con menos intervención humana durante su ejecución, esto se refiere a la tendencia en su desarrollo para que cada vez sean más autónomos los robots que como lo son en el presente.

El desarrollo de un robot móvil omnidireccional para enseñanza y experimentación permite el tener un robot con el cual se pueden hacer experimentaciones didácticas y, dependiendo del nivel de procesamiento que se requiera, otro tipo de experimentos como la prueba de conceptos de algoritmos de visión, navegación o exploración en ambientes desconocidos.

Se han presentado diferentes configuraciones de ruedas *mecanum* en robots omnidireccionales, pero con la distinción de que se encuentran equidistantes y en particular en robots con ruedas *mecanum* en un arreglo circular. En este trabajo se presenta una configuración diferente que muestra mediante un análisis matemático que no es necesario que se encuentren equidistantes para poder realizar movimientos propios de un robot móvil omnidireccional con ruedas.

Se muestra que es posible integrar una tarjeta de microcontrolador Arduino con el Sistema Operativo Robótico para controlar motores e interfaz de sensores. Es recomendable usar un Arduino basado en un microcontrolador de 32 bits para que sea lo suficientemente rápido para realizar todas las funciones y procesamiento relacionado con el control de la base móvil.

Basado en el Sistema Operativo Robótico se logró realizar las tareas de mapeado y navegación usando las paqueterías de software, cumpliendo con los requisitos mínimos para usar estos sistemas, para realizar el mapeado de un pasillo de la Universidad de Aalborg Copenhague en Dinamarca. Se creó un mapa, el cual es usado posteriormente para la navegación usando el robot móvil omnidireccional. En el caso de que al robot se le comandara para intentar ir a una

de las zonas en color gris, éste podría percibir obstáculos por medio de su sensor de profundidad y no podría llevar a cabo la navegación por esa zona si fuera el caso.

El costo del robot fue aproximadamente de $ 16,000.00 m.x.n. (aproximadamente € 10,000.00). El costo del robot es menor que el robot Turtlebot-2 el cual tiene un costo de $ 19,995.00 U.S.D (aproximadamente $ 30,000.00 m.x.n.).

El robot Turtlebot-2 está equipado con una tracción diferencial, una cámara Kinect, una batería Li-Po y una laptop tipo *Netbook* de la marca Asus. El costo de este robot es considerablemente menor en comparación con el costo del Turtlebot-2 y además posee la característica de ser omnidireccional y tiene una Unidad de Medición Inercial. El menor costo de este robot implica que más personas interesadas en el aprendizaje sobre robótica móvil tengan acceso a este tipo de tecnologías con un costo más accesible.

8.2 Trabajo a Futuro

El robot ya es capaz de crear mapas y navegar en ellos. Con estas funciones es posible comandar al robot para moverse autónomamente de un lugar a otro con un mapa dado. Dados los sensores con los que cuenta y el controlador de la base móvil completamente definido se pueden describir posibles experimentaciones con este robot móvil:

- Se empezó a desarrollar un algoritmo para manipulación de objetos por medio de la acción de empujar. Dada la paquetería de navegación se implementó un mapa de costos personalizado que consideró el objeto a empujar como un obstáculo con el fin de navegar, evadiendo el objeto, hasta la parte posterior del objeto. Después, desactivando el mapa de costos personalizado, comandar al robot a navegar hasta el punto donde tiene que empujar el objeto. Durante toda la segunda acción de navegación se sensará que el objeto siempre se encuentre enfrente del robot con el sensor ultrasónico.
- Se pueden crear tareas que incluyan la identificación de objetos en el ambiente para ubicarlos dentro del mapa. La ubicación de objetos puede ser a través de procesamiento de imágenes 2D o percepción 3D.

- Un algoritmo de exploración es posible de implementar. Con el sensor y con los comandos de velocidad se puede implementar un algoritmo de exploración como por ejemplo el algoritmo *frontier-based exploration* el cual se basa en explorar las fronteras de sus observaciones para determinar si es accesible o inaccesible y crear un mapa de todos los lugares accesibles de su ambiente.

Son muchos los algoritmos y tareas que se pueden implementar en este Robot Móvil Omnidireccional, desde algoritmos de comportamiento general para robots móviles como para robots móviles omnidireccionales.

Bibliografía

1. J. Ruiz de Garibay Pascual. Robótica: Estado del arte. Universidad de Deusto.
2. XHANGM, Houxiang. Mobile Robotics Lecture slides. University of Hamburg/ https://tams.informatik.uni-hamburg.de/lehre/2010ss/seminar/ir/PDF/MobilerobotLecture3_Review%20on%20mobile%20robot.pdf
3. G. Bermúdez. Robots móviles. TEORIA, APLICACIONES Y EXPERIENCIAS. TECNURA 10, 2002
4. J.A. Rietdijk."Ten propositions on mechatronics". Mechatronics in Products and Manufacturing Conference, Inglaterra. 1989.
5. Budynas, Richard; Nisbett Keith. Shigley's Mechanical Engineering Design. Mc Graw Hill 2014.
6. B.O. Nnaji. Theory of Automatic Robot Assembly and Programming. CHAPMAN & HALL. 1993.
7. J.-B. Song and K.-S. Byun, "Design and Control of a Four-Wheeled Omnidirectional Mobile Robot with Steerable Omnidirectional Wheels", Journal of Robotic Systems, 2006.
8. G. Indivieri, J. Paulus, P. G. Ploger. "Motion Control of Swedish Wheeled Mobile Robots in the Presence of Actuator Saturation". Lectures in Computer Science, 2006.
9. K.-L. Han, O.-K. Choi, J. Kim, J. S. Lee. "Design and Control of Mobile Robot with Mecanum Wheel". ICCAS-SICE, 2009.
10. S. Zaman, W. Slany, G. Steinbauer. "ROS-based Mapping, Localization and Autonomous Navigation using a Pioneer 3-DX Robot and their Relevant Issues". Electronics, Communications and Photonics Conference, 2011.
11. A. Oliver, S. Kang, B. C. Wunsche, B. MacDonald. "Using the Kinect as a Navigation Sensor for Mobile Robotics". Proceedings of the 27th Conference on Image and Vision Computing, 2012.
12. A. G. Araújo. ROSint – Integration of a mobile robot in ROS architecture. University of Coimbra, 2012.
13. L.-C. Lin, H.-Y. Shih. "Modeling and Adaptive Control of an Omni-Mecanum-Wheeled Robot". Intelligent Control and Automation, 2013.
14. J. A. Sustaeta. Desarrollo de un robot de servicio para la asistencia a personas de la tercera edad. Universitat Politécnica de Valéncia, 2014.
15. Vex Robotics. http://www.vexrobotics.com.mx/. Fecha de consulta: 7/05/2015.
16. ZOTAC D2550-ITX Product Description. http://www.zotac.com/products/mainboards/integrated-intel-cpu/zotac-d2550/product/zotac-d2550/detail/d2550-itx-wifi-supreme-b-series/sort/starttime/order/DESC/amount/10.html. Fecha de consulta: 7/05/2015
17. Asus Xtion Pro Live Descripción del Producto. http://www.asus.com/mx/Multimedia/Xtion_PRO_LIVE/. Fecha de consulta: 7/05/2015
18. Sensor Ultrasónico HC-SR04, Hoja de especificaciones. http://www.micropik.com/PDF/HCSR04.pdf Fecha de consulta: 7/05/2015
19. I2C Specification and User Manual, Phillips. http://www.nxp.com/documents/user_manual/UM10204.pdf, Rev. 6, April 2014.
20. PCA9512 Datasheet. http://docs-europe.electrocomponents.com/webdocs/0f74/0900766b80f74537.pdf
21. Product information MPU6050, http://www.invensense.com/mems/gyro/mpu6050.html, Fecha de consulta: 04/11/2014
22. Thomas Carpenter, Bidirectional Shifter using a BC547 Transistor, http://forum.arduino.cc/index.php?topic=119340.0
23. Campion Guy, Bastin Georges, *"Structural properties and classification of kinematic and dynamic models of wheeled mobile robots IEEE Conf. on Robotics and Automation, Atlanta May 1993, pp. 462-469"*, 1993.
24. González-Villela, V.J. (2006) *IV A Unifying Theory on Conventional Wheeled Mobile Robots.* Reino Unido
25. Anna Kaziunas France. Top Ten Tips: Designing Models For 3D Printing. http://makezine.com/2013/12/11/top-ten-tips-designing-models-for-3d-printing/. Fecha de consulta: 10/11/2014
26. A. Henning, I2C Encoder Vex Integrated Encoder. https://github.com/alexhenning/I2CEncoder. Fecha de consulta: 04/11/2014

27. B. Beauregard, PID Library for Arduino. http://playground.arduino.cc/Code/PIDLibrary. Fecha de consulta: 04/11/2014
28. T. Braunl. Embedded Robotics: Mobile Robot Deign and Applications with Embedded Systems. Chapter 4.2.4. Springer Science and Bussiness Media. 2006.
29. J. Rowberg. MPU6050 Arduino Library. https://github.com/jrowberg/i2cdevlib/tree/master/Arduino/MPU6050. Fecha de consulta: 01/11/2014
30. J. Rodrigo. Librería para arduino del módulo Ultrasonico Ranging HC-SR04. http://www.ardublog.com/library-for-arduino-ultrasonic-ranging-hc-sr04/. Fecha de consulta: 01/11/2014
31. T. O. Fredericks. Messenger Library for Arduino. http://playground.arduino.cc/code/messenger. Fecha de consulta: 01/11/2014
32. R. Hessmer. ROS Arduino Node. http://www.hessmer.org/blog/2010/11/21/sending-data-from-arduino-to-ros/. Fecha de consulta: 03/11/2014
33. L. Mosenlechner. ROS Best Practices. http://robohow.eu/_media/meetings/first-integration-workshop/ros-best-practices.pdf. Fecha de consulta: 06/11/2014
34. How the Kinect works. http://www.depthbiomechanics.co.uk/?p=100. Fecha de consulta: 16/05/2014
35. Depthimage_to_laserscan. http://wiki.ros.org/depthimage_to_laserscan. Fecha de consulta: 04/11/2014.
36. Ros Tutorials Camera Calibration. http://wiki.ros.org/camera_calibration/Tutorials/MonocularCalibration. Fecha de consulta: 06/12/2014.
37. Drivers Openni2_launch. http://wiki.ros.org/openni2_launch. Fecha de consulta: 06/12/2014.
38. Teleop_twist_joy package. http://wiki.ros.org/teleop_twist_joy. Fecha de consulta: 13/03/2015
39. T. Foote. “Tf: The transform library”. Technologies for Practical Robot Applications, 2013 IEEE Conference. April 2013.
40. S. Riisgaard, M. R. Blas. SLAM for Dummies, A Tutorial Approach to Simultaneous Localization and Mapping. 2005.
41. M. Montemerlo. FastSLAM: A Factored Solution to the Simultaneous Localization and Mapping Problem with Unknown Data Association. Robotics Institute, Carnegie Mellon University, July, 2003.
42. S. Thrun, W. Burgard, D. Fox. Probabilistic Robotics. The MIT Press. 2005.
43. J. Borenstein y L. Feng. Measurement and Correction of Systematic Odometry Errors in Mobile Robots. IEEE Transactions on Robotics and Automation, 2006.
44. T. Bayes. An Essay towards solving a Problem in the Doctrine of Chances. Philosophical Transactions of the Royal Society of London.
45. A. Elfes. Using Occupancy Grids for Mobile Robot Perception and Navigation. Carnegie Mellon University. IEEE, 1989.
46. G. Grisetti, C. Stachniss y W. Burgard. GMapping, OpenSLAM. https://www.openslam.org/gmapping.html. Fecha de consulta: 04/11/2014.
47. A. Hendrix. PR2 Simulator. http://wiki.ros.org/pr2_simulator. Fecha de consulta: 05/05/2015.
48. G. Grisetti, C. Stachniss y W. Burgard. Improving Grid-based SLAM with Rao-Blackwellized Particle Filters by Adaptive Proposals and Selective Resampling. Robotics and Automation, 2005.
49. B. Gerkey. Gmapping package ROS. http://wiki.ros.org/gmapping. Fecha de consulta: 01/11/2014.
50. N. J. Nilsson. A mobile automaton: An application of artificial intelligence techniques. IJCAI 1969.
51. S. Garrido, L. Moreno, M. Abderrahim y F. Martin. Path Planning for Mobile Robot Navigation using Voronoi Diagram and Fast Marching. International Conference on Intelligent Robots and Systems, 2006.
52. R. A. Brooks. Planning Collision-Free Motions for Pick-And-Place Operations. International Journal of Robotics Research. 1983.
53. D. V. Lu y M. Ferguson. AMCL ROS Package. http://wiki.ros.org/amcl. Fecha de consulta: 04/11/2014.
54. R. Li, M. A. Oskoei y H. Hu. Towards ROS Based Multi-robot Architecture for Ambient Assisted Living. IEEE International Conference on: Systems, Man, and Cybernetics, 2013.
55. M. R. Pedersen, D. L. Herzog y V. Kruger. Intuitive skill-level programming of industrial handling tasks on a mobile manipulator. Intelligent Robots and Systems, 2014.
56. S. S. Srinivasa, D. Berenson, M. Cakmak, A. Collet, M. R. Dogar, A. D. Dragan, R. A. Knepper, T. Niemueller, K. Strabala, M. V. Weghe y J. Ziegler. HERB 2.0: Lessons Learned from Developing a Mobile Manipulator for the Home. PROCEEDINGS OF THE IEEE, VOL. 100, NO. 8, JULY 2012.

57. J. Cheng. Scheduling and Motion Planning of iRobot Roomba. Department of Information and Computer Sciences, University of Hawaii Manoa.
58. M. M. Victor Fernando. Planificación de Trayectorias para Robots Móviles. Departamento de Ingeniería de Sistemas y Automática, Universidad de Malaga, 1995.
59. D. V. Lu y M. Ferguson. ROS Navigation Stack. http://wiki.ros.org/navigation. Fecha de consulta: 04/11/2014.
60. R. Hessmer. 2d Navigation. http://www.hessmer.org/blog/wp-content/uploads/2011/06/2d_Navigation_ROS.pdf. June 2011.
61. Amit. Introduction to A*. http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html. Fecha de consulta: 18/05/2014.
62. D. V. Lu y M. Ferguson. Costmap_2d Package ROS. http://wiki.ros.org/costmap_2d. Fecha de consulta: 01/11/2014.
63. D. V. Lu y M. Ferguson. Move_base Package ROS. http://wiki.ros.org/move_base. Fecha de consulta: 02/12/2014.
64. D. V. Lu y M. Ferguson. Base local Planner ROS. http://wiki.ros.org/base_local_planner. Fecha de consulta: 02/12/2014.
65. B. Gerkey y T. Pratkanis. Map Server Package. http://wiki.ros.org/map_server. Fecha de consulta: 05/12/2014.
66. Turtlebot-2 Product Description. ClearPath Robotics. http://store.clearpathrobotics.com/products/turtlebot-2. Fecha de consulta: 25/05/2015.

APENDICES

Apéndice 1. Librería Odometría

Odometer.h

```
#ifndef odometer3_H
#define odometer3_H

class Odometer3
{
public:
    Odometer3(double*, double*, double*, double*, float*, double*, double*);

    void update();
    void useImu();
    void noImu();
    double x;
    double y;
    double th;
    double vx;
    double vy;
    double w;
    bool use_imu;
    double yfl;
    double yrl;
    double wsupport;
private:
    double *rFL;
    double *rFR;
    double *rRR;
    double *rRL;
    float  *yaw; //Is reversed in my particular case
    double offset;
    unsigned long lastTime;
    double secs;
    double l1;// = 0.16;   //m
    double l2;// = 0.1665; // 0.16637; //m
    double radius_wheel;// = 0.052; //0.05146; //m
    double dd;
    double sm;
    double quarter_radiuswheel;
    double half_radiuswheel;
    double support_constant1;
    double support_constant2;
    double lsum;
    double _th; // last th value
    double alpha; //low pass filter value
    double *linearscale;
    double *angularscale;
};

#endif // ODOMETER_H
```

Odometer.cpp

```
#include "odometer3.h"
#include "Arduino.h"
```

```
#define PI 3.14159265
#define TwoPI 6.28318531

Odometer3::Odometer3(double* wFL, double* wFR, double* wRR, double* wRL, float* yyaw, double*
linscale, double* angscale)
{
    //Attach wheel speeds
    rFL = wFL;
    rFR = wFR;
    rRR = wRR;
    rRL = wRL;
    yaw = yyaw;
    use_imu = false;
    l1 = 0.16;
    l2 = 0.1665;
    lsum = l1 + l2;
    dd = 0.245;
    sm = 0.035;
    radius_wheel = 0.0468;
    quarter_radiuswheel = radius_wheel / 4.0;
    half_radiuswheel = radius_wheel / 2.0;
    lastTime = millis();
    support_constant1 = lsum + dd - sm;
    support_constant2 = lsum - dd + sm;
    yfl = half_radiuswheel * support_constant2 / support_constant1;
    yrl = radius_wheel * lsum / support_constant1;
    wsupport = radius_wheel / support_constant1;
    //Initialize variables
    x = 0.0;
    y = 0.0;
    th = 0.0;
    _th = 0.0;
    vx = 0.0;
    vy = 0.0;
    w = 0.0;
    alpha = 0.05; //alpha muy baja
    //attach linear and angular scales
    linearscale = linscale;
    angularscale = angscale;
}
void Odometer3::useImu()
{
     use_imu = true;
     offset = (-*yaw) - (th * 180 / PI);
}
void Odometer3::noImu()
{
     use_imu = false;
}
void Odometer3::update()
{
    //Conversion from rpm to rad/sec with the factor 2 pi / 60 = 0.104717
    double RRinput = *rRR * 0.104717;
    double RLinput = *rRL * 0.104717;
    double FRinput = *rFR * 0.104717;
    double FLinput = *rFL * 0.104717;
    //linear and angular scale only applies for odometry using wheels
    /*
    vx = (quarter_radiuswheel) * (FLinput + FRinput + RLinput + RRinput) * *linearscale;
    vy = (quarter_radiuswheel) * (-FLinput + FRinput +RLinput - RRinput) * *linearscale;
```

```
    w = (quarter_radiuswheel) * ((1/(-lsum))*FLinput + (1/(lsum))*FRinput + (1/(-
lsum))*RLinput + (1/(lsum))*RRinput ) * *angularscale;
    */
    vx = (half_radiuswheel) * (FLinput + FRinput) * *linearscale;
    vy = (yfl * *linearscale * FLinput) + (*linearscale * half_radiuswheel * FRinput) +
(RLinput * yrl * *linearscale) ;
    w = ( (-wsupport * FLinput) + (-wsupport * RLinput) ) * *angularscale;

    unsigned long dt = (millis() - lastTime);
    secs = dt / 1000.0;

    _th = th; //Last value of th

    if(use_imu) //get the yaw from the IMU with the correspondant offset
    {
        th = ((-*yaw) - offset) * PI / 180;
        if (th > PI)
        {
            th -= TwoPI;
        }
        else
        {
            if (th <= -PI)
            {
                th += TwoPI;
            }
        }
    }

    double delta_x = (vx * (double)cos(th) - vy * (double)sin(th)) * secs;
    double delta_y = (vx * (double)sin(th) + vy * (double)cos(th)) * secs;
    double delta_th = w * secs;
    if(!use_imu) //if not using IMU do the calculations using wheels
         th += delta_th;
    if (th > PI)
    {
        th -= TwoPI;
    }
    else
    {
        if (th <= -PI)
        {
            th += TwoPI;
        }
    }
    x += delta_x;
    y += delta_y;
    lastTime = millis();
}
```

Apéndice 2. Librería de la Cinemática

Kinematics_arduino3.h

```
#ifndef kinematics_arduino3_H
#define kinematics_arduino3_H

#include "Arduino.h"
```

```
class kinematics_arduino3
{
public:
    double timemsec();
    void stop_speed();
    kinematics_arduino3(double*, double*, double*,double*, double*, double*, double*, double*,
double*);
    bool compute();
private:
    double secs();
    double *vx;
    double *vy;
    double *vth;
    double *FL;
    double *FR;
    double *RR;
    double *RL;
    double lastvx;
    double lastvy;
    double lastvth;
    unsigned long lastTime;
    //robot dimensions
    double l1;// = 0.16;   //m
    double l2;// = 0.1665; // 0.16637; //m
    double dd; //For the third wheel
    double sm; //For the third wheel
    double radius_wheel;// = 0.052; //0.05146; //m
    double *linearscale;
    double *angularscale;

};


#endif // KINEMATICS_ARDUINO_H
```

Kinematics_arduino.cpp

```
#include "Arduino.h"
#include "kinematics_arduino3.h"

//Constructor to set the pointers to the variables
kinematics_arduino3::kinematics_arduino3(double* evx, double* evy, double* evth,
                                         double* eFL, double* eFR, double* eRR,
                                         double* eRL, double* linscale, double* angscale)
{
    vx = evx;
    vy = evy;
    vth = evth;
    FL = eFL;
    FR = eFR;
    RR = eRR;
    RL = eRL;
    lastvx = 0.0;
    lastvy = 0.0;
    lastvth = 0.0;
    lastTime = millis();
    l1 = 0.16;
    l2 = 0.1665;
    dd = 0.24;
    sm = 0.035;
    radius_wheel = 0.052;
```

```
    linearscale = linscale;
    angularscale = angscale;
}

double kinematics_arduino3::timemsec()
{
    return lastTime;
}
void kinematics_arduino3::stop_speed()
{
        *FL = 0.0;
        *FR = 0.0;
        *RL = 0.0;
        *RR = 0.0;
}
bool kinematics_arduino3::compute()
{
    lastTime = millis();


        *FL = ((*vx / *linearscale) - (*vy / *linearscale) - ((l1+l2) * (*vth /
*angularscale))) / radius_wheel ;
        *FR = ((*vx / *linearscale) + (*vy / *linearscale) + ((l1+l2) * (*vth /
*angularscale))) / radius_wheel ;
        *RL = ((*vy / *linearscale) - (*vx / *linearscale) - ((dd-sm) * (*vth /*/
*angularscale*/))) / radius_wheel ;

        //The control node will receive the desired speeds in RPM , so it is neccesary to
        //do a units conversion, from rad/sec to RPM , multiplying by a factor of 60 / 2 pi =
9.5496
        *FL = *FL * 9.5496;
        *FR = *FR * 9.5496;
        *RL = *RL * 9.5496;
        *RR = *RR * 9.5496;
        lastvx = *vx;
        lastvy = *vy;
        lastvth = *vth;
        return true;
    //}
   // else return false;
}
```

Apéndice 3. Programa Arduino

```
//////////////////////////////////////////
/*This program was modified to include
the kinematics of the new configuration
using three wheels*/
This program was created based on the one created by Dr. Hessmer and the example of the
library for MPU6050
//////////////////////////////////////////
#include <Messenger.h> //Library for communicating with the computer
#include <kinematics_arduino3.h> //Library by Juan for the kinematics
#include <odometer3.h> //Library by Juan for the Odometry
#include <PID_v1.h>  //PID library for Arduino, Modified Integral term reset when in setpoint
and output =0
#include <Servo.h>  //Arduino servo library
#include <Wire.h>  //Arduino I2C Library
```

```
#include <I2CEncoder.h>  //Library from alexhenning on github for driving the 269 Vex Encoders
with arduino
#include <Ultrasonic.h> //Library to drive the Ultrasonic sensor on the shovel of the robot
#include "I2Cdev.h" //Necessary library for using MPU
#include "MPU6050_6Axis_MotionApps20.h" //Library for interfacing the MPU

#define encoders_on 53 //Encoders switch on
#define IMU_ON 51 //Imu switch on

//Declaring the MPU object
MPU6050 mpu(0x69); //With the I2C address

// MPU control/status vars
bool dmpReady = false;  // set true if DMP init was successful
uint8_t mpuIntStatus;   // holds actual interrupt status byte from MPU
uint8_t devStatus;      // return status after each device operation (0 = success, !0 = error)
uint16_t packetSize;    // expected DMP packet size (default is 42 bytes)
uint16_t fifoCount;     // count of all bytes currently in FIFO
uint8_t fifoBuffer[64]; // FIFO storage buffer

// orientation/motion vars
Quaternion q;           // [w, x, y, z]         quaternion container
VectorInt16 aa;         // [x, y, z]            accel sensor measurements
VectorInt16 aaReal;     // [x, y, z]            gravity-free accel sensor measurements
VectorInt16 aaWorld;    // [x, y, z]            world-frame accel sensor measurements
VectorFloat gravity;    // [x, y, z]            gravity vector
float euler[3];         // [psi, theta, phi]    Euler angle container
float ypr[3];           // [yaw, pitch, roll]   yaw/pitch/roll container and gravity vector
float yaw = 0.0, pitch = 0.0, roll = 0.0;
bool imu_use = false;
//Odometer send values
int sx, sy, sth, svx, svy, sw;
//Declaring the Ultrasonic object
Ultrasonic ultrasonic(10, 9); // (Trig PIN, Echo PIN)
//Declaring the servos
Servo FR;
Servo FL;
Servo RR;
Servo RL;
//Declaring the Encoders
I2CEncoder FRencoder;
I2CEncoder FLencoder;
I2CEncoder RRencoder;
I2CEncoder RLencoder;
//Declaring servo pins
const int servoFL = 22; //correct
const int servoFR = 24; //correct
const int servoRR = 30; //correct
const int servoRL = 28; //correct
//Messenger declaration
Messenger omMessenger = Messenger();
bool initialized = false;
unsigned long updte = 0;
//Declaring PID objects an dvariables
double FLsetpoint, FLinput, FLoutput, FLlastp;
double FRsetpoint, FRinput, FRoutput, FRlastp;
double RRsetpoint, RRinput, RRoutput, RRlastp;
double RLsetpoint, RLinput, RLoutput, RLlastp;
double flKp = 0.46, flKi = 1.6, flKd = 0.001;
double frKp = 0.46, frKi = 1.6, frKd = 0.001;
double rrKp = 0.46, rrKi = 1.6, rrKd = 0.001;
```

```
double rlKp = 0.46, rlKi = 1.6, rlKd = 0.001;
//Declaring the PID
PID FLpid(&FLinput, &FLoutput, &FLsetpoint, flKp, flKi, flKd, DIRECT);
PID FRpid(&FRinput, &FRoutput, &FRsetpoint, frKp, frKi, frKd, REVERSE);
PID RRpid(&RRinput, &RRoutput, &RRsetpoint, rrKp, rrKi, rrKd, REVERSE);
PID RLpid(&RLinput, &RLoutput, &RLsetpoint, rlKp, rlKi, rlKd, DIRECT);
//kinematics variables
double vx, vy, vth;
//Correction values
double linearscale = 1;
double angularscale = 0.984;
//Kinematics declaration
kinematics_arduino3 kinematics(&vx, &vy, &vth, &FLsetpoint, &FRsetpoint, &RRsetpoint,
&RLsetpoint, &linearscale, &angularscale);
//Odometer declaration
Odometer3 odometer(&FLinput, &FRinput, &RRinput, &RLinput, &yaw, &linearscale, &angularscale);
//Support variables
int ultra_delay = 0;
unsigned long timesincelastcommand = 0;
boolean flagzero = true;
boolean connection_mpu;
boolean using_imu = false ;   //Flag for indicating use of Imu
boolean auto_imu = false; //Flag for activating the imu after 30 seconds
boolean no_imu = false;


//
// ================================================================
// ===               INTERRUPT DETECTION ROUTINE  MPU             ===
// ================================================================

volatile bool mpuInterrupt = false;     // indicates whether MPU interrupt pin has gone high
void dmpDataReady()
{
  mpuInterrupt = true;
}
// ================================================================
// ===                      INITIAL SETUP                       ===
// ================================================================

void setup()
{
  Wire.begin();
  imu_on();
  Wire1.begin();
  Serial.begin(57600);
  omMessenger.attach(OnReceived);
  mpu.initialize(); //Initialize Imu
  connection_mpu = mpu.testConnection(); //CHeck if connection was succesful
  devStatus = mpu.dmpInitialize(); //Initialize the Digital Motion Proccesing
  // supply your own gyro offsets here, scaled for min sensitivity
  mpu.setXGyroOffset(220);
  mpu.setYGyroOffset(76);
  mpu.setZGyroOffset(-85);
  mpu.setZAccelOffset(1788); // 1688 factory default for my test chip
  if ( devStatus == 0)
  {
    Serial.print("p");
    Serial.print("\t");
    Serial.print("IMU initiated");
    Serial.print("\n");
```

```
    mpu.setDMPEnabled(true);
    attachInterrupt(50, dmpDataReady, RISING);
    mpuIntStatus = mpu.getIntStatus();
    dmpReady = true;
    packetSize = mpu.dmpGetFIFOPacketSize();
  }
  else
  {
    Serial.print("p");
    Serial.print("\t");
    Serial.print("IMU failed");
    Serial.print("\n");
  }
  //encodersPowerOn();
  encoderSetup();
  motorsSetup();
  pidSetup();
  delay(100);
}

// ================================================================
// ===                      MAIN PROGRAM LOOP                   ===
// ================================================================

void loop()
{
  ReadSerial();

  if ( millis() - updte >= 20) //Was 30,  update each 20 ms
  {
    updte = millis();
    if (initialized)
    {
      doOm();
      Serial.flush();
    }
    else
    {
      req_init();
    }
  }
  if (millis() > 30000 && !auto_imu) //30 seconds for IMU stabilisation
  {
    odometer.useImu();
    auto_imu = true;
  }
  if (mpuInterrupt || fifoCount >= packetSize)
  {
    //if (mpuInterrupt)// || fifoCount >= packetSize) //Retrieve IMU data if interrupt or new
packet available
    getIMUdata();
  }
}

// ================================================================
// ===                      FUNCTIONS                           ===
// ================================================================
```

```
void getIMUdata()
{
  /*
  Serial.print("p");
    Serial.print("\t");
    Serial.print(" Here I am ");
    Serial.print(millis());
    Serial.print("\n");
    */
  // reset interrupt flag and get INT_STATUS byte
  mpuInterrupt = false;
  mpuIntStatus = mpu.getIntStatus();

  // get current FIFO count
  fifoCount = mpu.getFIFOCount();

  // check for overflow (this should never happen unless our code is too inefficient)
  if ((mpuIntStatus & 0x10) || fifoCount == 1024)
  {
    // reset so we can continue cleanly
    mpu.resetFIFO();
    //Serial.println(F("FIFO overflow!"));

    // otherwise, check for DMP data ready interrupt (this should happen frequently)
  }
  else if (mpuIntStatus & 0x02)
  {
    // wait for correct available data length, should be a VERY MAIN PROGRAM LOOPMAIN PROGRAM
LOOPshort wait
    while (fifoCount < packetSize) fifoCount = mpu.getFIFOCount();

    // read a packet from FIFO
    mpu.getFIFOBytes(fifoBuffer, packetSize);

    // track FIFO count here in case there is > 1 packet available
    // (this lets us immediately read more without waiting for an interrupt)
    fifoCount -= packetSize;

    mpu.dmpGetQuaternion(&q, fifoBuffer);
    mpu.dmpGetGravity(&gravity, &q);
    mpu.dmpGetYawPitchRoll(ypr, &q, &gravity);

    yaw = ypr[0] * 180 / M_PI;
    pitch = ypr[1] * 180 / M_PI;
    roll = ypr[2] * 180 / M_PI;
  }
}
void imu_on()  //Turns on the IMU
{
  pinMode(IMU_ON, OUTPUT);
  digitalWrite(IMU_ON, LOW);
  delay(100);
  digitalWrite(IMU_ON, HIGH);
  delay(30);
}
void doOm() //Will work as robot
{
  if ( odometer.use_imu && !using_imu)
  {
    Serial.print("p");
    Serial.print("\t");
```

```
    Serial.print("Now using IMU");
    Serial.print("\n");
    using_imu = true;
    no_imu = false;
  }
  if ( !odometer.use_imu && !no_imu)
  {
    Serial.print("p");
    Serial.print("\t");
    Serial.print("Not using IMU");
    Serial.print("\n");
    no_imu = true;
    using_imu = false;
  }

  if (millis() - timesincelastcommand > 1500 && !flagzero) //Stop motors after 1.5 seconds of
no command
  {
    setpointzero();
    //encodersPowerOn();
    //encoderSetup();
    flagzero = true;
    Serial.print("p");
    Serial.print("\t");
    Serial.print("Set to Zero. No command received in 1.5 second");
    Serial.print("\n");
  }

  inputsetpoint(); //Get the inputs and setpoints
  pidcompute(); //Compute the speeds
  motorwrite(); //output for the motors
  odometer.update(); //Update the Odometry
  printodometry(); //Sends the Odometry information to the computer
  printultrasonic(); //Sends the ultrsonic information to the computer
  //*Just debugging*/printypr();

}
void printypr()
{

  Serial.print("p");
  Serial.print("\t");
  Serial.print("Yaw ");
  Serial.print(yaw);
  Serial.print("\t");
  Serial.print("Pitch ");
  Serial.print(pitch);
  Serial.print("\t");
  Serial.print("Roll ");
  Serial.print(roll);
  Serial.print("\n");

}
void encodersPowerOn() //Will start the encoders
{
  pinMode(encoders_on, OUTPUT);
  digitalWrite(encoders_on, LOW);
  delay(50);
  digitalWrite(encoders_on, HIGH);
  delay(50);
}
```

```
void encoderSetup() //Will setup the encoders
{
  //Wire.setClock(10000);
  RRencoder.init(MOTOR_269_ROTATIONS, MOTOR_269_TIME_DELTA);  //For these settings the output
will be given in RPM
  RLencoder.init(MOTOR_269_ROTATIONS, MOTOR_269_TIME_DELTA);
  FLencoder.init(MOTOR_269_ROTATIONS, MOTOR_269_TIME_DELTA);
  FRencoder.init(MOTOR_269_ROTATIONS, MOTOR_269_TIME_DELTA);
  RLencoder.setReversed(true); //Function in case the encoders are not counting in the right
direction
  FLencoder.setReversed(true);
  FLlastp = FLencoder.getPosition();
  FRlastp = FRencoder.getPosition();
  RRlastp = RRencoder.getPosition();
  RLlastp = RLencoder.getPosition();
}
//Setting up the motors
void motorsSetup()
{
  pinMode(24, OUTPUT);
  pinMode(22, OUTPUT);
  pinMode(30, OUTPUT);
  pinMode(28, OUTPUT);
  FR.attach(servoFR, 1000, 2000);
  FL.attach(servoFL, 1000, 2000);
  RR.attach(servoRR, 1000, 2000);
  RL.attach(servoRL, 1000, 2000);
}
void pidSetup() //Setting up the PID control
{

  //Test PID
  FLsetpoint = 0;
  FRsetpoint = 0;
  RRsetpoint = 0;
  RLsetpoint = 0;
  FLinput = InputPID(FLlastp, 4);
  FRinput = InputPID(FRlastp, 3);
  RRinput = InputPID(RRlastp, 1);
  RLinput = InputPID(RLlastp, 2);

  FLpid.SetMode(AUTOMATIC);
  FRpid.SetMode(AUTOMATIC);
  RRpid.SetMode(AUTOMATIC);
  RLpid.SetMode(AUTOMATIC);
  FLpid.SetOutputLimits(-75.0, 75.0);
  FRpid.SetOutputLimits(-75.0, 75.0);
  RRpid.SetOutputLimits(-75.0, 75.0);
  RLpid.SetOutputLimits(-75.0, 75.0);
}
void ReadSerial() //Read the serial for the Messenger
{
  while (Serial.available())
  {
    omMessenger.process(Serial.read());
  }
}

void OnReceived() //Will execute this everytime a message is received
{
  if (omMessenger.checkString("Startimu"))  //Initiate communication
```

```
  {
    initialized = true;

    Serial.print("Active");
    Serial.print("\n");
    return;
  }
  if (omMessenger.checkString("Startnoimu"))
  {
    initialized = true;
    auto_imu = true;
    Serial.print("Active");
    Serial.print("\n");
    return;
  }
  if (omMessenger.checkString("lscale"))
  {
    linearscaleR();
    Serial.print("p");
    Serial.print("\t");
    Serial.print("Received linear correction");
    Serial.print("\n");
    return;
  }
  if (omMessenger.checkString("ascale"))
  {
    angularscaleR();
    Serial.print("p");
    Serial.print("\t");
    Serial.print("Received angular correction");
    Serial.print("\n");
    return;
  }
  if (omMessenger.checkString("s")) // Get speed commands
  {
    SpeedReceived();
    return;
  }
  if (omMessenger.checkString("noimu"))
  {
    odometer.noImu();
    return;
  }
  if (omMessenger.checkString("imu"))
  {
    odometer.useImu();
    return;
  }
  // clear out unrecognized content
  while (omMessenger.available())
  {
    omMessenger.readInt();
  }
}

void req_init() //Requesting the initial message
{
  Serial.print("InitializeBaseController");
  Serial.print("\n");

}
```

```
void setpointzero() //Set the inputs to zero
{
  FLsetpoint = 0;
  FRsetpoint = 0;
  RLsetpoint = 0;
  RRsetpoint = 0;
}
void printultrasonic() //Send ultrasonic information to the computer
{
  //ultra_delay++;
  //if (ultra_delay > 5)
  //{
  Serial.print("u");
  Serial.print("\t");
  Serial.print(ultrasonic.Ranging(CM));
  Serial.print("\n");
  //  ultra_delay = 0;
  //}
}
void printodometry() //Sends odometry information to the computer
{

  Serial.print("o"); //Sends the Odometry values
  Serial.print("\t");
  Serial.print(odometer.x, 4);
  Serial.print("\t");
  Serial.print(odometer.y, 4);
  Serial.print("\t");
  Serial.print(odometer.th, 4);
  Serial.print("\t");
  Serial.print(odometer.vx, 4);
  Serial.print("\t");
  Serial.print(odometer.vy, 4);
  Serial.print("\t");
  Serial.print(odometer.w, 4);
  Serial.print("\n");

  Serial.print("w"); //Sends the Odometry values
  Serial.print("\t");
  Serial.print(FLinput,4);
  Serial.print("\t");
  Serial.print(RLinput,4);
  Serial.print("\t");
  Serial.print(FRinput,4);
  Serial.print("\n");
}
void inputsetpoint() //Get the current speed of the motors
{
  FLinput = InputPID(FLlastp, 4);
  FRinput = InputPID(FRlastp, 3);
  RRinput = InputPID(RRlastp, 1);
  RLinput = InputPID(RLlastp, 2);

  if (RLsetpoint > 85)
    RLsetpoint = 85;
  if (RLsetpoint < -85)
    RLsetpoint = -85;
  if (RRsetpoint > 85)
    RRsetpoint = 85;
  if (RRsetpoint < -85)
```

```
    RRsetpoint = -85;
  if (FRsetpoint > 85)
    FRsetpoint = 85;
  if (FRsetpoint < -85)
    FRsetpoint = -85;
  if (FLsetpoint > 85)
    FLsetpoint = 85;
  if (FLsetpoint < -85)
    FLsetpoint = -85;
}
void motorwrite() //Write the current values to the motors
{
  FL.write(90 + FLoutput);
  FR.write(90 + FRoutput);
  //RR.write(90 + RRoutput);
  RR.write(90);
  RL.write(90 + RLoutput);
}
void motorzero() //Function to stop the motors
{
  FL.write(90);
  FR.write(90);
  RR.write(90);
  RL.write(90);
}
void pidcompute() //Does the control computation
{
  FLpid.Compute();
  FRpid.Compute();
  RRpid.Compute();
  RLpid.Compute();
  if ( abs(FLsetpoint) < 5)
    FLoutput = 0;
  if ( abs(FRsetpoint) < 5)
    FRoutput = 0;
  if ( abs(RRsetpoint) < 5)
    RRoutput = 0;
  if ( abs(RLsetpoint) < 5)
    RLoutput = 0;
}
void SpeedReceived() //Manage the speed received
{
  timesincelastcommand = millis();
  flagzero = false;
  float cmd_vx = GetFloatFromBaseAndExponent(omMessenger.readInt(), omMessenger.readInt());
  float cmd_vy = GetFloatFromBaseAndExponent(omMessenger.readInt(), omMessenger.readInt());
  float cmd_w = GetFloatFromBaseAndExponent(omMessenger.readInt(), omMessenger.readInt());
  setSpeedWheels(cmd_vx, cmd_vy, cmd_w);
}
void linearscaleR()
{
  float lscale = GetFloatFromBaseAndExponent(omMessenger.readInt(), omMessenger.readInt());
  if (lscale > 0)
  {
    linearscale = lscale;
  }
}
void angularscaleR()
{
  float ascale = GetFloatFromBaseAndExponent(omMessenger.readInt(), omMessenger.readInt());
  if (ascale > 0)
```

```
  {
    angularscale = ascale;
  }
}

float GetFloatFromBaseAndExponent(int base, int exponent) //Get the actual value of number
received
{
  return base * pow(10, exponent);
}

void setSpeedWheels(float _vx, float _vy, float _w) //This function will set the new speed
through the kinematics class
{
  vx = _vx;
  vy = _vy;
  vth = _w;
  kinematics.compute(); //Calculates the wheel speeds through the kinematics class

}
double InputPID(double lpos, int slct) //Get the correct value of the speed based on last and
current position of the wheels
{
  double currentPosition = 0;
  double InPID = 0;
  double diffPosition = 0;
  int signo;
  switch (slct)
  {
    case 1:
      currentPosition = RRencoder.getPosition();
      RRlastp = currentPosition;
      break;
    case 2:
      currentPosition = RLencoder.getPosition();
      RLlastp = currentPosition;
      break;
    case 3:
      currentPosition = FRencoder.getPosition();
      FRlastp = currentPosition;
      break;
    case 4:
      currentPosition = FLencoder.getPosition();
      FLlastp = currentPosition;
      break;
  }
  diffPosition = currentPosition - lpos;
  if (diffPosition < 0)
  {
    signo = -1;
  }
  else if (diffPosition == 0)
  {
    signo = 0;
  }
  else    {
    signo = 1;
  }
  switch (slct)
  {
    case 1:
```

```
      InPID = RRencoder.getSpeed() * signo ;
      break;
    case 2:
      InPID = RLencoder.getSpeed() * signo ;
      break;
    case 3:
      InPID = FRencoder.getSpeed() * signo ;
      break;
    case 4:
      InPID = FLencoder.getSpeed() * signo ;
      break;
  }
  return InPID;
}
```

Apéndice 4. Nodo de Arduino

Arduino.py

```
#!/usr/bin/env python
'''
Heavily borrowed from Dr. Hessmer
Modified by Juan de Dios
'''

#import roslib

import rospy
import tf
import math
from math import sin, cos, pi
import sys

from geometry_msgs.msg import Quaternion
from geometry_msgs.msg import Twist
from nav_msgs.msg import Odometry
from std_msgs.msg import String
from sensor_msgs.msg import Range
from my_repository.msg import speed_wheel
from my_repository.srv import *

from SerialDataGateway import SerialDataGateway

class Arduino(object):
        '''
        Helper class for communicating with an Arduino board over serial port
        '''
        def _HandleReceivedLine(self,  line):
                self._Counter = self._Counter + 1
                self._speedsimupub.publish(String(str(self._Counter) + " " + line))

                if (len(line) > 0):
                        lineParts = line.split('\t')
                        if (lineParts[0] == 'u'):
                                self._ulsensor(lineParts)
                                return
                        if (lineParts[0] == 'o'):
                                self._BroadcastOdometry(lineParts)
                                return
```

```
                if (lineParts[0] == "InitializeBaseController"):
                        # controller requesting initialization
                        self._InitializeBase()
                        return
                if (lineParts[0] == "Active"):
                        # controller requesting initialization
                        self._IsActive()
                        return
                if (lineParts[0] == 'p'):
                        self._printThis(lineParts)
                        return
                if (lineParts[0] == 'w'):
                        self._sendspeeds(lineParts)
                        return

        def _ulsensor(self, lineParts):
                partsCount = len(lineParts)
                if (partsCount < 2):
                        pass
                try:
                    drange = float(lineParts[1])

                    rangesensor = Range()
                    drange = drange / 100;
                    rangesensor.range = drange
                    rangesensor.header.frame_id = "ul_range"
                    rangesensor.field_of_view = 0.4
                    rangesensor.min_range = 0.01
                    rangesensor.max_range = 1
                    rosNow = rospy.Time.now()
                    rangesensor.header.stamp = rosNow
                    self._RangePublisher.publish(rangesensor)
                except:
                    rospy.logwarn("Unexpected error:" + str(sys.exc_info()[0]))

        def _printThis(self, lineParts): #Function to receive information from Arduino
                partsCount = len(lineParts)
                for x in range(1, partsCount):
                        rospy.loginfo("Printing: " + lineParts[x])


        def _BroadcastOdometry(self, lineParts):
                partsCount = len(lineParts)
                #rospy.logwarn(partsCount)
                if (partsCount  < 6):
                        pass

                try:
                        x = float(lineParts[1])

                        y = float(lineParts[2])
                        theta = float(lineParts[3])

                        vx = float(lineParts[4])
                        vy = float(lineParts[5])
                        omega = float(lineParts[6])

                        #quaternion = tf.transformations.quaternion_from_euler(0, 0, theta)
                        quaternion = Quaternion()
                        quaternion.x = 0.0
                        quaternion.y = 0.0
```

```
                        quaternion.z = sin(theta/2.0)
                        quaternion.w = cos(theta/2.0)


                        rosNow = rospy.Time.now()

                        # First, we'll publish the transform from frame odom to frame
base_link over tf
                        # Note that sendTransform requires that 'to' is passed in before
'from' while
                        # the TransformListener' lookupTransform function expects 'from' first
followed by 'to'.
                        self._OdometryTransformBroadcaster.sendTransform(
                                (x, y, 0),
                                (quaternion.x, quaternion.y, quaternion.z, quaternion.w),
                                rosNow,
                                "base_link",
                                "odom"
                                )

                        # next, we'll publish the odometry message over ROS
                        odometry = Odometry()
                        odometry.header.frame_id = "odom"
                        odometry.header.stamp = rosNow
                        odometry.pose.pose.position.x = x
                        odometry.pose.pose.position.y = y
                        odometry.pose.pose.position.z = 0.0812779 #just for debugging
                        odometry.pose.pose.orientation = quaternion

                        odometry.child_frame_id = "base_link"
                        odometry.twist.twist.linear.x = vx
                        odometry.twist.twist.linear.y = vy
                        odometry.twist.twist.angular.z = omega

                        self._OdometryPublisher.publish(odometry)

                        #rospy.loginfo(odometry)

                except:
                        rospy.logwarn("Unexpected error:" + str(sys.exc_info()[0]))



        def _sendspeeds(self, lineParts):
                partsCount = len(lineParts)
                #rospy.logwarn(partsCount)
                if (partsCount  < 3):
                        pass

                try:

                            FL = float(lineParts[1])
                            #rospy.loginfo("Todo bien")
                        RL = float(lineParts[2])
                        FR = float(lineParts[3])
                        rosNow = rospy.Time.now()
                        wheelspeds = speed_wheel()

                            wheelspeds.header.frame_id = "base_link"
                        wheelspeds.header.stamp = rosNow
```

```
                        wheelspeds.FL = FL
                        wheelspeds.RL = RL
                        wheelspeds.FR = FR




                        self._SpeedsPublisher.publish(wheelspeds)

                        #rospy.loginfo(odometry)

                except:
                        rospy.logwarn("Unexpected error:" + str(sys.exc_info()[0]))


        def _WriteSerial(self, message):
                #self._SerialPublisher.publish(String(str(self._Counter) + ", out: " +
message))
                self._SerialDataGateway.Write(message)

        def __init__(self, port="/dev/ttyACM0", baudrate=57600):
                '''
                Initializes the receiver class.
                port: The serial port to listen to.
                baudrate: Baud rate for the serial communication
                '''

                self._Counter = 0

                rospy.init_node('arduino')

                port = rospy.get_param("~port", "/dev/ttyACM0")
                baudRate = int(rospy.get_param("~baudRate", 57600))


                rospy.loginfo("Starting with serial port: " + port + ", baud rate: " +
str(baudRate))
                self._SerialDataGateway = SerialDataGateway(port, baudRate,
self._HandleReceivedLine)


                # Service for IMU use
                self._ImuService = rospy.Service('Use_Imu', imuactive, self._sendImu)
                # Service for angular and linear correction values
                self._angularservice = rospy.Service('Angular_float_value', angularscale,
self._sendang)
                self._linearservice = rospy.Service('Linear_float_value', linearscale,
self._sendlin)
                # subscriptions
                rospy.Subscriber("cmd_vel", Twist, self._GetSpeeds, queue_size=10)

                self._OdometryPublisher = rospy.Publisher("odom", Odometry, queue_size=10)
                self._SpeedsPublisher = rospy.Publisher("wheel_speeds", speed_wheel,
queue_size=10)
                self._OdometryTransformBroadcaster = tf.TransformBroadcaster()
                self._RangePublisher = rospy.Publisher("ulrange", Range, queue_size=10)
                self._SerialPublisher = rospy.Publisher('serial', String, queue_size=10)
                self._speedsimupub = rospy.Publisher('speeds', String, queue_size=10)
```

```
        def Start(self):
                rospy.logdebug("Starting")
                self._SerialDataGateway.Start()

        def Stop(self):
                rospy.logdebug("Stopping")
                self._SerialDataGateway.Stop()

        def _GetSpeeds(self, twist_command):
                """ Handle movement requests. """
                vx = twist_command.linear.x        # m/s
                vy = twist_command.linear.y        # m/s
                omega = twist_command.angular.z      # rad/s
                #rospy.loginfo("Sending twist command: " + str(vx) + "," + str(vy) + "," +
str(omega))

                message = 's %d %d %d %d %d %d\r' % self._GetBaseAndExponents((vx, vy, omega))
                rospy.logdebug("Sending speed command message: " + message)
                self._WriteSerial(message)
                #rospy.loginfo("Sent: " + message)
        def _sendImu(self, request):
                """ Service to enable or disable imu for odometry"""

                rospy.set_param("~imuenable", request.imu_active)
                if request.imu_active:
                   message = 'imu\r'
                   rospy.loginfo("Imu enabled")
                else:
                   message = 'noimu\r'
                   rospy.loginfo("Imu disabled")

                self._WriteSerial(message)
                return imuactiveResponse()

        def _sendang(self, request):
                rospy.set_param("~angular_correction", request.avaluescale)
                message = 'ascale %d %d\r' % self._GetBaseAndExponent(request.avaluescale)
                rospy.loginfo("Sending correction value: " + message)
                self._WriteSerial(message)
                return angularscaleResponse()

        def _sendlin(self, request):
                rospy.set_param("~linear_correction", request.lvaluescale)
                message = 'lscale %d %d\r' % self._GetBaseAndExponent(request.lvaluescale)
                rospy.loginfo("Sending correction value: " + message)
                self._WriteSerial(message)
                return linearscaleResponse()

        def _InitializeBase(self):
                """ Writes an initializing string to start the Base """
                imuen = rospy.get_param("~imuenable", "True")
                if imuen:
                     message = 'Startimu\r'
                else:
                     message = 'Startnoimu\r'

                rospy.loginfo("Initializing Base " + message)
                self._WriteSerial(message)
```

```
                lincorrection = rospy.get_param("~linear_correction", 1.0)
                angcorrection = rospy.get_param("~angular_correction", 0.984)
                message = 'ascale %d %d\r' % self._GetBaseAndExponent(angcorrection)
                rospy.loginfo("Sending correction value: " + message)
                self._WriteSerial(message)
                message = 'lscale %d %d\r' % self._GetBaseAndExponent(lincorrection)
                rospy.loginfo("Sending correction value: " + message)
                self._WriteSerial(message)

        def _GetBaseAndExponent(self, floatValue, resolution=4):
                '''
                Converts a float into a tuple holding two integers:
                The base, an integer with the number of digits equaling resolution.
                The exponent indicating what the base needs to multiplied with to get
                back the original float value with the specified resolution.
                '''

                if (floatValue == 0.0):
                        return (0, 0)
                else:
                        exponent = int(1.0 + math.log10(abs(floatValue)))
                        multiplier = math.pow(10, resolution - exponent)
                        base = int(floatValue * multiplier)

                        return(base, exponent - resolution)

        def _GetBaseAndExponents(self, floatValues, resolution=4):
                '''
                Converts a list or tuple of floats into a tuple holding two integers for each
float:
                The base, an integer with the number of digits equaling resolution.
                The exponent indicating what the base needs to multiplied with to get
                back the original float value with the specified resolution.
                '''

                baseAndExponents = []
                for floatValue in floatValues:
                      baseAndExponent = self._GetBaseAndExponent(floatValue)
                      baseAndExponents.append(baseAndExponent[0])
                      baseAndExponents.append(baseAndExponent[1])

                return tuple(baseAndExponents)

        def _IsActive(self):
                rospy.loginfo("Is Active")


if __name__ == '__main__':
        arduino = Arduino()
        try:
                arduino.Start()
                rospy.spin()

        except rospy.ROSInterruptException:
                arduino.Stop()
```

Apéndice 5. Paquetería de teleoperación Modificada

Teleop_twist_joy.cpp

```
/**
Software License Agreement (BSD)

\authors   Mike Purvis <mpurvis@clearpathrobotics.com>
\copyright Copyright (c) 2014, Clearpath Robotics, Inc., All rights reserved.

*/
/**
Modified by Juan de Dios to accept commands for the Y axis of movement

*/

#include "geometry_msgs/Twist.h"
#include "ros/ros.h"
#include "sensor_msgs/Joy.h"
#include "teleop_twist_joy/teleop_twist_joy.h"


namespace teleop_twist_joy
{

/**
 * Internal members of class. This is the pimpl idiom, and allows more flexibility in adding
 * parameters later without breaking ABI compatibility, for robots which link TeleopTwistJoy
 * directly into base nodes.
 */
struct TeleopTwistJoy::Impl
{
  void joyCallback(const sensor_msgs::Joy::ConstPtr& joy);

  ros::Subscriber joy_sub;
  ros::Publisher cmd_vel_pub;

  int enable_button;
  /**/int change_button;
  int enable_turbo_button;
  int axis_linear;
  int axis_angular;
  double scale_linear;
  double scale_linear_turbo;
  double scale_angular;

  bool sent_disable_msg;
};

/**
 * Constructs TeleopTwistJoy.
 * \param nh NodeHandle to use for setting up the publisher and subscriber.
 * \param nh_param NodeHandle to use for searching for configuration parameters.
 */
TeleopTwistJoy::TeleopTwistJoy(ros::NodeHandle* nh, ros::NodeHandle* nh_param)
{
  pimpl_ = new Impl;
```

```
  pimpl_->cmd_vel_pub = nh->advertise<geometry_msgs::Twist>("cmd_vel", 1, true);
  pimpl_->joy_sub = nh->subscribe<sensor_msgs::Joy>("joy", 1,
&TeleopTwistJoy::Impl::joyCallback, pimpl_);

  nh_param->param<int>("enable_button", pimpl_->enable_button, 10);
  nh_param->param<int>("enable_turbo_button", pimpl_->enable_turbo_button, -1);

  nh_param->param<int>("axis_linear", pimpl_->axis_linear, 1);
  nh_param->param<double>("scale_linear", pimpl_->scale_linear, 0.5);
  nh_param->param<double>("scale_linear_turbo", pimpl_->scale_linear_turbo, 1.0);

  nh_param->param<int>("axis_angular", pimpl_->axis_angular, 0);
  nh_param->param<double>("scale_angular", pimpl_->scale_angular, 1.0);

  nh_param->param<int>("change_button", pimpl_->change_button, 11);

  ROS_INFO_NAMED("TeleopTwistJoy", "Using axis %i for linear and axis %i for angular.",
      pimpl_->axis_linear, pimpl_->axis_angular);
  ROS_INFO_NAMED("TeleopTwistJoy", "Teleop on button %i at scale %f linear, scale %f
angular.",
      pimpl_->enable_button, pimpl_->scale_linear, pimpl_->scale_angular);
  ROS_INFO_COND_NAMED(pimpl_->enable_turbo_button >= 0, "TeleopTwistJoy",
      "Turbo on button %i at scale %f linear.", pimpl_->enable_turbo_button, pimpl_-
>scale_linear_turbo);

  pimpl_->sent_disable_msg = false;
}

void TeleopTwistJoy::Impl::joyCallback(const sensor_msgs::Joy::ConstPtr& joy_msg)
{
  // Initializes with zeros by default.
  geometry_msgs::Twist cmd_vel_msg;

  if (enable_turbo_button >= 0 && joy_msg->buttons[enable_turbo_button] && !joy_msg-
>buttons[change_button])
  {
    cmd_vel_msg.linear.x = joy_msg->axes[axis_linear] * scale_linear_turbo;
    cmd_vel_msg.linear.y = joy_msg->axes[2] * scale_linear_turbo;
    cmd_vel_msg.angular.z = joy_msg->axes[axis_angular] * scale_angular;
    cmd_vel_pub.publish(cmd_vel_msg);
    sent_disable_msg = false;
  }
  else if (joy_msg->buttons[enable_button] && !joy_msg->buttons[change_button])
  {
    cmd_vel_msg.linear.x = joy_msg->axes[axis_linear] * scale_linear;
    cmd_vel_msg.linear.y = joy_msg->axes[2] * scale_linear;
    cmd_vel_msg.angular.z = joy_msg->axes[axis_angular] * scale_angular;
    cmd_vel_pub.publish(cmd_vel_msg);
    sent_disable_msg = false;
  }
  else if (enable_turbo_button >= 0 && joy_msg->buttons[enable_turbo_button] && joy_msg-
>buttons[change_button])
  {
    cmd_vel_msg.linear.x = joy_msg->axes[axis_linear] * scale_linear_turbo;
    cmd_vel_msg.linear.y = joy_msg->axes[axis_angular] * scale_linear_turbo;
    cmd_vel_msg.angular.z = joy_msg->axes[2] * scale_angular;
    cmd_vel_pub.publish(cmd_vel_msg);
    sent_disable_msg = false;
  }
  else if (joy_msg->buttons[enable_button] && joy_msg->buttons[change_button])
```

```
  {
    cmd_vel_msg.linear.x = joy_msg->axes[axis_linear] * scale_linear;
    cmd_vel_msg.linear.y = joy_msg->axes[axis_angular] * scale_linear;
    cmd_vel_msg.angular.z = joy_msg->axes[2] * scale_angular;
    cmd_vel_pub.publish(cmd_vel_msg);
    sent_disable_msg = false;
  }
  else
  {
    // When enable button is released, immediately send a single no-motion command
    // in order to stop the robot.
    if (!sent_disable_msg)
    {
      cmd_vel_pub.publish(cmd_vel_msg);
      sent_disable_msg = true;
    }
  }
}

}  // namespace teleop_twist_joy
```