

Universidad Nacional de Córdoba
Facultad de Ciencias Exactas Físicas y Naturales
Ingeniería en Computación



Proyecto Integrador

Robot móvil autónomo capaz de producir mapas 3D del interior de una habitación

Agustín Caverzasi – Fernando Saravia Rajal
2013

Director: Ing. Orlando Micolini

Co-director: Ing. Ladislao Mathé

Robot móvil autónomo capaz de producir mapas 3D del interior de una habitación

Datos de los autores

Alumnos:

- Caverzasi, Agustín
Email: agucaverzasi@gmail.com
Teléfono de contacto: (0351) 153415019
- Saravia Rajal, Fernando
Email: fersarr@gmail.com
Teléfono de contacto: (0351) 152345378

Director:

- Ing. Micolini, Orlando
Email: omicolini@compuar.com

Codirector:

- Ing. Mathe, Ladislao
Email: ing.ladislao.mathe@gmail.com

1 Resumen

Los últimos dos años representaron los mejores años para la robótica industrial desde 1961. Las proyecciones muestran que esta tendencia va a mantenerse durante los próximos años y, posiblemente, durante décadas. De estar acotada exclusivamente al ámbito industrial, la robótica ha pasado a expandirse hacia áreas nunca antes pensadas, y aplicaciones de las más diversas. La realidad muestra que la robótica en sus diversos campos –industrial, médico, doméstico, militar, comercial, entre otros- ha dado pasos inmensos; el potencial es gigantesco.

Dichos avances, vienen alentados por la revolución de los celulares inteligentes, y su fabricación a gran escala, que produjeron un importante avance en la fabricación de nuevos y mejores sensores, además de una reducción en su tamaño y precio. Como si esto fuera poco, en los últimos 5 años el contexto se vio potenciado por la creación de un sistema operativo para robótica denominado Robot Operating System que alienta al desarrollo y la investigación, facilitando un desarrollo organizado de código reusable y abierto, con herramientas necesarias para cualquier proyecto de robótica. Adicionalmente, la aparición del sensor Kinect, de distribución masiva y bajo costo, desencadenó la construcción de una gran cantidad de robots capaces de “ver” en tres dimensiones, identificar personas y crear mapas.

En el presente trabajo, se realiza el diseño y la construcción de un robot móvil, con la capacidad de auto localizarse, navegar, y generar mapas del lugar por el que se desplaza. El diseño del robot cuenta con un desarrollo electrónico para el control de los motores y sensores de odometría, un microcontrolador Arduino sobre el cual se desarrolla toda la funcionalidad de bajo nivel para el control del robot, y con una computadora a bordo para la navegación, visualización y mapeo. Además, se utiliza un sensor Kinect, que es una cámara RGBD, con el fin de realizar odometría visual para la generación de mapas del entorno.

El robot desarrollado, además de la realización de mapas, puede ser utilizado como un robot para exploración, telepresencia o educación. En cuanto a esta última, los robots son una excelente herramienta para la enseñanza, ya que permiten acceder a una mezcla de experiencias tanto prácticas como teóricas, abarcando numerosas áreas como la electrónica, programación, la física y la matemática. La plataforma móvil quedará lista para ser utilizada en el futuro por otros grupos de desarrollo, para distintos trabajos de investigación, prácticas o trabajos de materias vinculadas a las carreras Ingeniería en Computación y Electrónica.

Abstract

The last two years have been the best for industrial robotics since 1961 and these trends are expected to continue during the next and, likely, for decades. From being almost exclusively used in the industrial scene, robotics has expanded to never-before-imagined areas and applications. Amongst the most popular sectors in which robotics has been increasing its role are the industrial, medical, military, commercial and domestic fields.

The revolution of smart phones that offer a wide variety of sensors has dropped both, the price and the size of the sensors, allowing people to build their own robots at home. The robotics expansion has been fueled, additionally, by the creation of the Robot Operating System (ROS), which is intended to facilitate the organized development of open source and reusable software for robotics. ROS also provides libraries and tools that are necessary in every robotics project. Besides, the release of the Kinect sensor as a massively distributed, low cost product has unchained the construction of a great amount of 3D-“seeing” robots which can also track people and create maps.

The goal of this project was to construct a mobile robot, capable of localizing itself, navigating, and generating maps of what it “sees” through an onboard camera. In order to control the wheels and provide movement, two motors were used. An Arduino board do the robot control, manages all the sensors needed for the odometry and movements, and on which was programmed all the low level functionality. On the other hand, there is an on board computer for navigation, visualization and mapping. Mapping and visual odometry are done based on the incoming RGBD stream of images from Kinect sensor.

The constructed robot offers a wide field of applications; besides mapping, other interesting alternatives are the exploration of dangerous zones, telepresence and education. Regarding education, robots can be an excellent tool: they represent an interesting mix of practical and theoretical experiences, involving different areas such as programming, electronics, physics and math. Furthermore, this robotic platform will be left available and ready to be used in the future by other groups for different projects, to do professional practices, or subject’s works associated with the careers Computer engineering and Electronic engineering.

2 Contenido

Datos de los autores.....	1
Resumen.....	3
Abstract.....	3
1 Contenido.....	6
2 Tabla de imágenes.....	11
3 Índice de tablas	17
4 Introducción	19
4.1 Objetivos	19
4.1.1 Principales	19
4.1.2 Secundarios	19
4.2 Motivación	19
4.3 Metodología	20
5 Contexto de trabajo.....	21
5.1 Situación del grupo cliente	21
5.2 Obtención de requerimientos	21
6 Estudio del problema	23
6.1 Requerimientos	39
6.1.1 Requerimientos funcionales.....	39
FSR1. Movimientos del robot a partir de comandos básicos.....	39
FSR2. Localización del robot.....	39
FSR3. Control de trayectoria y velocidad.....	40
FSR4. Integrar la computadora a bordo en el robot.....	40
FSR5. Medición de distancias con el sensor	41
FSR6. Interfaz gráfica para mostrar mapas.....	41
FSR7. Generar mapas 3D	41
FSR8. Navegación. Movimientos autónomos a partir de mapas	42
6.1.2 Requerimientos no-funcionales	43
6.2 Herramientas de control de versiones	44
6.3 Hardware y software considerado	44
6.3.1 Sensores	44
6.3.2 Placa de desarrollo – Microcontrolador	47
6.3.3 Sistema operativo y entorno de desarrollo	48
6.4 Análisis de riesgos	49

6.5	Especificaciones definitivas para el proyecto	50
6.5.1	Especificaciones de hardware	50
6.5.2	Especificaciones de software.....	51
7	Diseño de la arquitectura del sistema	53
7.1	Diseño modular	53
7.2	Diseño funcional.....	54
8	Proceso de desarrollo.....	57
9	Marco teórico.....	23
9.1	Robot.....	23
9.1.1	Clasificación de los robots	24
9.2	Microcontroladores y placas de desarrollo	28
9.3	ROS – Robot Operating System	29
9.4	Odometría	33
9.4.1	Odometria visual	34
9.5	Sensor Kinect.....	35
10	Desarrollo	69
10.1	Diseño del robot.....	69
10.2	Construcción del robot.....	73
10.2.1	Detalles constructivos	73
10.2.2	Motores y encoders	74
10.2.3	Sensores de orientación	78
10.2.4	Círcuito electrónico	84
10.2.5	Arduino.....	91
10.2.6	Baterías Li-PO	94
10.2.7	Sensor Kinect.....	102
10.3	Control.....	107
10.3.1	Introducción	107
10.3.2	Controlador proporcional (P-Controller)	108
10.3.3	Controlador proporcional derivador (PD-Controller)	108
10.3.4	Controlador proporcional integrador derivador (PID – Controller)	110
10.3.5	Sistema de control del Robot	110
10.4	Localización	129
10.4.1	Posición del robot.....	129
10.4.2	Implementación de la Localización.....	130
10.4.3	Test de localización	132
10.4.4	Conclusión	148

10.5	Mapeo y odometría visual.....	149
10.5.1	Introducción	149
10.5.2	Implementación de odometría visual y mapeo	152
10.5.3	Precisión de la odometría visual.....	162
10.5.4	Integración con el sistema.....	167
10.5.5	Conclusión	171
10.6	Navegación.....	173
10.6.1	Planificación de movimientos.....	173
10.6.2	Algoritmo de búsqueda de caminos.....	173
10.6.3	Elección del algoritmo	186
10.6.4	Test de navegación	187
10.6.5	Integración con el sistema.....	197
10.7	Sistema completo.....	199
11	Conclusión	203
12	Trabajos futuros y mejoras.....	205
13	Anexo	206
13.1	Anexo A: Propuesta del proyecto y estado del arte	207
13.2	Anexo B: Principios de funcionamiento del Magnetómetro.....	207
14	Bibliografía	211

3 Tabla de imágenes

Ilustración 1: Diseño modular de la arquitectura del sistema	53
Ilustración 2: Diseño funcional de la arquitectura del sistema y sus componentes.	55
Ilustración 3: Diagrama explicativo de la metodología de desarrollo iterativa.....	58
Ilustración 4: Muestra el proceso de testing que se realiza para el modelo de desarrollo incremental.	63
Ilustración 5: Relación entre sensores y actuadores.	23
Ilustración 6: Grados de libertad en un espacio en 3D.....	25
Ilustración 7: Robot holonómico. El robot es capaz de desplazarse en dos direcciones y, además, puede rotar sin desplazarse	25
Ilustración 8: Ejemplo de robot holonómico	25
Ilustración 9: Robot no holonómico. No es posible controlar todos los grados de libertad del robot.	26
Ilustración 10: Modelo cinemático diferencial con dos ruedas.....	26
Ilustración 11: Mecanismo de giro en el modelo cinemático de Ackerman.....	28
Ilustración 12: Microcontroladores más conocidos	28
Ilustración 13: Comunicación entre nodos en ROS a partir del tema (topic) llamado “command_velocity”. Obtenida con la herramienta de ROS rxgraph. Este ejemplo corresponde a uno de los tutoriales más básicos de ROS.	30
Ilustración 14: Composición de cada mensaje en ROS (Morgan Quigley, Brian Gerkey, Ken Conley, 2009).	30
Ilustración 15: Ejemplo de rqt_graph.....	31
Ilustración 16: Diferentes sistemas de coordenadas necesarios en un robot complejo	31
Ilustración 17: Diferentes tipos de display con los que cuenta RViz.....	32
Ilustración 18: Muestra el posicionamiento del robot en un Sistema de coordenadas.	34
Ilustración 19: El terreno produce deslizamientos de las ruedas, que resultan en una mala odometría por parte de los Encoders. En estos casos, la odometría visual produce mejores resultados.....	34
Ilustración 20: Sensores presentes en Kinect.....	36
Ilustración 21: Imagen de profundidad (esq. superior izquierda), seguimiento de humanos (esq. superior derecha) y imagen de color (esq. inferior derecha).	37
Ilustración 22: Bytes en la imagen de color proveniente de Kinect.....	37
Ilustración 23: Modos cercano y lejano del sensor Kinect for Windows	38
Ilustración 24: Ruedas utilizadas para el robot.....	71
Ilustración 25: Proceso de pintado de las plataformas de madera. Se pueden apreciar los agujeros a través de los cuales son pasados los cables que conectan los motores(debajo de la base) con el microcontrolador (arriba de la base).....	73
Ilustración 26: La pintura también se aplicó al tubo de plástico encargado de garantizar la alineación de los motores.....	73
Ilustración 27: Mecanismo de ajuste de los motores para evitar desplazamientos y rotaciones no deseadas durante el movimiento.....	74
Ilustración 28: Motor planetario. Sistema de engranajes.....	75
Ilustración 29: Se muestra el motorTT junto a la rueda, y en la parte inferior los cables 1 y 2 que son la alimentación del motor.	75
Ilustración 30: Especificaciones y dimensiones del motor TT.....	76
Ilustración 31: Mediciones de corriente motor derecho.....	77
Ilustración 32: Mediciones de corriente motor izquierdo	77
Ilustración 33: Se muestran las conexiones del encoder.....	78
Ilustración 34: Declinación magnética en Córdoba, Argentina	79

Ilustración 35: Sentido de la declinación magnética (Magnetic declination, Sentido de declinación magnética).....	79
Ilustración 36: Chip integrado del magnetómetro HMC5883L.....	80
Ilustración 37: Herramienta diseñada para la medición de orientaciones con el magnetómetro.....	80
Ilustración 38: Mediciones con el magnetómetro tomadas lejos de la facultad FCEFyN.	81
Ilustración 39: Mediciones con el magnetómetro tomadas en la facultad FCEFyN	81
Ilustración 40: Giróscopo, ejes de referencia.	82
Ilustración 41: Relación entre (x,y,z) y (yaw,pitch,roll)	82
Ilustración 42: Conexión del giróscopo con Arduino	83
Ilustración 43: Puente H. Principio de funcionamiento.....	84
Ilustración 44: Puente H. Señales de entrada y sus efectos	85
Ilustración 45: Circuito esquemático del L298 (Motor driver datasheet L298N).....	85
Ilustración 46: Circuito esquemático del optoacoplador. Modelo 6N137	86
Ilustración 47: Diodos flyback.....	87
Ilustración 48: Esquematico del circuito. Parte de control a la izquierda de los optoacopladores y parte de potencia a la derecha.	88
Ilustración 49: Primer prototipo de la placa electrónica para control de motores y encoders	89
Ilustración 50: Placa PCB del circuito electrónico diseñada para el robot.....	90
Ilustración 51: Foto del circuito implementado	90
Ilustración 52: Microcontrolador Arduino Atmega2560	91
Ilustración 53: Conexiones y pines del Arduino utilizadas para el presente trabajo	92
Ilustración 54: PWM y voltaje efectivo.....	92
Ilustración 55: PWM y su utilización en Arduino.....	93
Ilustración 56: Separación entre el circuito de potencia y el de control mediante un optoacoplador.....	94
Ilustración 57: Plataforma inferior terminada. Entre los componentes que van sobre esta, se distinguen la batería, el microcontrolador y el circuito de potencia y control.....	94
Ilustración 58: Batería Li-Po de 22.2 V y 3000mA (Zippy Flightmax 6S1P).	96
Ilustración 59: Curvas de descarga de una Li-Po con distintas capacidades.....	97
Ilustración 60: Cargador de las baterías	98
Ilustración 61: Cargador de baterías. Establecimiento del voltaje de carga.....	99
Ilustración 62: Cargador de baterías. Establecimiento del tiempo de carga	99
Ilustración 63 Cargador de baterías. Establecimiento de la corriente.....	99
Ilustración 64: Alimentación del cargador.....	100
Ilustración 65: Cargador de baterías. Establecimiento del modo para la batería utilizada	100
Ilustración 66: Cargador de baterías. Establecimiento de la cantidad de celdas.....	100
Ilustración 67: Cargador de baterías en modo balanceo. Voltaje.....	101
Ilustración 68: Cargador de baterías en modo balanceo. Celdas.	101
Ilustración 69: Cargador de baterías. Muestra de estado actual de la batería.....	101
Ilustración 70: Una vez colocada la base superior, se cuenta con varillas que permiten el ajuste de la altura de la misma.....	102
Ilustración 71: Altura de la base superior ajustada hasta el nivel definitivo. Adicionalmente, se observan las conexiones entre la PC y el microcontrolador	103
Ilustración 72: Cable de alimentación de Kinect fue cortado para introducir una batería como fuente de energía.....	103
Ilustración 73: Cables de Kinect que se conectan a los bornes de la batería.....	104
Ilustración 74: Batería de 12 V, 1A encargada de alimentar Kinect.....	104
Ilustración 75: Kinect alimentada por batería en funcionamiento. La computadora muestra la imagen de profundidad.....	105

Ilustración 76: Robot definitivo. Mirando en la dirección contraria al avance del robot.....	105
Ilustración 77: Robot definitivo, mirando en la dirección de avance del robot. Se puede apreciar la imagen de profundidad producida por Kinect.....	106
Ilustración 78: Robot definitivo con todos sus componentes conectados y funcionando	106
Ilustración 79: Control. crosstrack error.....	107
Ilustración 80: Corrección de ángulo y sobrepasamiento con el controlador proporcional.....	108
Ilustración 81: Corrección del desvío introducida por el controlador PD	109
Ilustración 82: Desviación sistemática. Necesidad de un controlador PID	110
Ilustración 83: Sistema de control del robot para el control de velocidad de cada rueda.	111
Ilustración 84: Valores de la función analogWrite de Arduino y sus efectos en el PWM	113
Ilustración 85: Diagrama de bloques general de un controlador PID	113
Ilustración 86: Análisis de varias respuestas del controlador PID para distintas constantes	115
Ilustración 87: Planilla donde se realizan los test del PID	118
Ilustración 88: Mediciones de la velocidad de las ruedas.....	118
Ilustración 89: Resultado de velocidad de motores para un dutyCycle del %100 (máxima velocidad)	119
Ilustración 90: Velocidad de los motores	119
Ilustración 91: Diferencias de velocidad entre ambos motores	120
Ilustración 92: Localización del robot.....	129
Ilustración 93: Posibles avances del robot según su orientación	130
Ilustración 94: Sentido de giro para los comandos right y left	131
Ilustración 95: Giros. Right 45. Posición inicial.	134
Ilustración 96: Giros. Right 45. Posición final.	134
Ilustración 97: Giros. Left 45. Posición inicial.	135
Ilustración 98: Giros. Left 45. Posición final.....	135
Ilustración 99: Giros. Right 90. Posición inicial.	136
Ilustración 100: Giros. Right 90. Posición final.	136
Ilustración 101: Giros. Left 90. Posición inicial.	137
Ilustración 102: Giros. Left 90. Posición final.....	137
Ilustración 103: Giros. Right 180. Posición inicial.	138
Ilustración 104: Giros. Right 180. Posición final.	138
Ilustración 105: Giros. Left 180. Posición inicial.	139
Ilustración 106: Giros. Left 180. Posición final.....	139
Ilustración 107: Movimiento lineal up. 10 cm. Inicial.....	141
Ilustración 108: Movimiento lineal up. 10 cm. Final.....	141
Ilustración 109: Movimiento lineal Down. 10 cm. Inicial.....	142
Ilustración 110: Movimiento lineal Down. 10 cm. Final.	142
Ilustración 111: Movimiento lineal up. 20 cm. Inicial.....	143
Ilustración 112: Movimiento lineal up. 20 cm. Final.....	143
Ilustración 113: Movimiento lineal Down. 20 cm. Inicial.....	144
Ilustración 114: Movimiento lineal Down. 20 cm. Final.	144
Ilustración 115: Movimiento lineal up. 100 cm. Inicial.....	145
Ilustración 116: Movimiento lineal up. 100 cm. Final.....	145
Ilustración 117: Movimiento lineal down. 100 cm. Inicial.....	146
Ilustración 118: Movimiento lineal down. 100 cm. Final.....	146
Ilustración 119: Detección de características y flujo óptico(Myung Hwangbo).....	149
Ilustración 120: Algoritmo de odometría visual (Scaramuzza Davide)	150
Ilustración 121: SLAM, a medida que el robot avanza, detecta puntos referencias a través de los cuales estima su localización.....	151

Ilustración 122: Corrección lograda mediante SLAM (Scaramuzza Davide)	152
Ilustración 123: Mapeo 3D de la habitación con el método basado en apariencias	153
Ilustración 124: Otra perspectiva del mapeo 3D de la habitación con el método basado en apariencias	154
Ilustración 125: Mapeo basado en apariencias. Habitación y ambiente vecino.....	154
Ilustración 126: Nodos constituyentes del trabajo ccny_rgbd (Dryanovski)	155
Ilustración 127: Mapeo 3D de la habitación con el método basado en extracción de características	156
Ilustración 128: La habitación según otra perspectiva. Mapeo con método basado en extracción de características.....	156
Ilustración 129: Tercera perspectiva de la habitación producida con el método basado en extracción de características. Se puede apreciar el ambiente vecino a la habitación.	157
Ilustración 130: Box de trabajo en el LAC, desde arriba	159
Ilustración 131: Otra perspectiva del box de trabajo en el LAC.....	159
Ilustración 132: GRSI desde arriba. Notar que la cámara fue posicionada en el centro del laboratorio. Punto desde el cual no eran visibles todas las paredes. Por esta razón, se ven “huecos” en el mapa.....	160
Ilustración 133: GRSI desde otro ángulo	160
Ilustración 134: GRSI desde adentro con el fin de mostrar los detalles.	161
Ilustración 135: Grafo de ROS durante la ejecución del mapeo	161
Ilustración 136: Recorrido a través de la odometría visual. Posición inicial	162
Ilustración 137: Recorrido a través de la odometría visual.	163
Ilustración 138: Mediciones de distancia mediante la odometría visual con Kinect	164
Ilustración 139: Regresión lineal de las mediciones de distancias.....	165
Ilustración 140: Medición de las rotaciones mediante la odometría visual	165
Ilustración 141: Medición de distancias sobre un mapa. Ancho de la puerta.	166
Ilustración 142 Medición sobre los mapas: foto real de la puerta	166
Ilustración 143: Medición sobre los mapas, pared de la oficina.....	167
Ilustración 144: Grafo de ROS, conexión del nodo encargado de generar el mapa 2d a partir del mapa 3d	168
Ilustración 145: Mapa 2d de la habitación. Los 1 representan paredes y obstáculos	168
Ilustración 146: Mapa 2d de la habitación. Cada 2 representa la inflación de los obstáculos y paredes ..	169
Ilustración 147: Generación de objetivos para la navegación	170
Ilustración 148: Proceso de mapeo en 2d	170
Ilustración 149: Continuación del proceso de mapeo en 2d y la generación simultánea de objetivos	171
Ilustración 151: Resumen del proceso de exploración	171
Ilustración 152: Navegación	173
Ilustración 153: Mapa de ejemplo y movimientos posibles para el robot.....	174
Ilustración 154: Representación del robot dentro de un mapa 2d.....	174
Ilustración 155: Árbol de exploración BFS (Wikipedia - BFS)	175
Ilustración 156: Trayectoria planeada	180
Ilustración 157: Avance del robot siguiendo la trayectoria	182
Ilustración 158: Mapeo 3d completo de una habitación.....	188
Ilustración 159: Exploración. Objetivo a alcanzar.....	189
Ilustración 160: Trayectoria generada.....	190
Ilustración 161: Mapeo 3D del box de trabajo en el LAC.....	190
Ilustración 162: Objetivos generados en el LAC	191
Ilustración 163: Trayectoria generada para salir del BOX hacia la primera salida	192
Ilustración 164: Trayectoria generada para salir del BOX hacia la segunda salida	193
Ilustración 165: Trayectoria hasta el tercer punto	194
Ilustración 166: Trayectoria hasta objetivo artificial	195

Ilustración 167: Nodos del sistema para los modos Debug y telepresencia.....	197
Ilustración 168: Librería “Gyroscope” de Arduino.....	199
Ilustración 169: Librería “Robot” de Arduino	199
Ilustración 170: Directorio del workspace de ROS donde se encuentra el Paquete ‘robot’ y su contenido	200
Ilustración 171: Carpetas con nodos en C++ del servidor y del cliente	200
Ilustración 172: Nodos de ROS del servidor en C++	201
Ilustración 173: Nodos de ROS del cliente en C++.....	201
Ilustración 174: Nodo de ROS en Python	201
Ilustración 175: Nodos de ROS en modo debugger.....	201
Ilustración 176: Nodos de ROS en modo telepresencia.	202
Ilustración 177: Nodos de ROS en modo de mapeo y exploración.....	202
Ilustración 178: Campo geomagnético alrededor de la Tierra (Wikimedia).....	208
Ilustración 179: Representación 3D del campo geomagnético	208
Ilustración 180: Modelo simplístico del campo geomagnético	209
Ilustración 181: Declinación magnética (Norte geográfico y norte magnético. Declinación).....	209
Ilustración 182: Declinación magnética sobre la superficie terrestre (Wikimedia)	210

4 Índice de tablas

Tabla 1: Análisis y comparación de para medición de posición y distancia, y sus características.....	45
Tabla 2: Análisis y comparación de sensores de orientación y sus características.	46
Tabla 3: Análisis y comparación de microcontroladores	48
Tabla 4: Análisis y comparación de los posibles Sistemas Operativos a elegir.	49
Tabla 5: Matriz de análisis de riesgos	50
Tabla 6: Modos de operación y funcionamiento del robot.	56
Tabla 7: Muestra cada iteración junto con sus incrementos, y a que requerimiento hace referencia.....	62
Tabla 8: Tabla de resumen de casos de testing mapeados con cada requerimiento y los módulos de la arquitectura que involucra.	67
Tabla 9: Muestra la relación entre cada sensor y la propiedad física que mide.	24
Tabla 10: Peso del robot.....	70
Tabla 11: Alimentación de cada componente	70
Tabla 12: Se muestran las especificaciones del motor. La fila señalada en amarillo muestra el rango de voltajes de operación utilizada por el robot (4-28V) (Motor gmp36 36mm especificaciones).	76
Tabla 13: Tarjeta de testing FSR3 ST3.1.....	120
Tabla 14: Tarjeta de testing FSR1 ST1.....	133
Tabla 15: Tarjeta de testing FSR2 ST2.1.....	133
Tabla 16: Tarjeta testing FSR2 ST2.2	140
Tabla 17: Resumen de tests de movimiento lineal.....	147
Tabla 17: Tarjeta de testing FSR6 ST6.1.....	158
Tabla 18: Tarjeta de testing FSR7 ST7.1.....	158
Tabla 19: Tarjeta de testing FSR7 ST7.2.....	158
Tabla 20: Tarjeta de testing FSR5 ST5.1.....	163
Tabla 21: Tarjeta de testing FSR5 ST5.1.....	164
Tabla 22: Mediciones de distancias con Kinect	165
Tabla 24: Tarjeta testing FSR8 ST8.1	187
Tabla 25: Tarjeta testing FSR8 ST8.2	187
Tabla 26: Tarjeta testing FSR8 ST8.3	187

5 Introducción

5.1 Objetivos

5.1.1 Principales

Explorar y construir un mapa aproximado de un espacio cerrado y acotado, el cual posee obstáculos que permiten la circulación.

5.1.2 Secundarios

- 1) Construir un dispositivo móvil que se desplace dentro de la habitación de forma autónoma, con el objetivo de obtener mediciones de la misma y evitar los obstáculos que se interpongan.
- 2) Seleccionar uno, o un conjunto de sensores que permitan la correcta medición del lugar.
- 3) Determinar sobre qué plataforma de hardware y software se va a montar el sistema.

5.2 Motivación

Tal como se mencionó anteriormente, la robótica y su campo de aplicación han demostrado un crecimiento extraordinario en los últimos años. Japón, Estados Unidos, Alemania y China se encuentran hoy en la vanguardia de la industria que tuvo, en 2011, su mejor año de ventas de robots industriales en los últimos 50 años (IFR International Federation of Robotics). En Estados Unidos y el Reino Unido los robots participan de los sistemas de salud como asistentes en todo tipo de intervenciones médicas. En China, existen restaurantes donde todos los camareros son robots. Pero el fenómeno es global. En América Latina la venta de robots se duplicó en México, tuvo un gran salto en Brasil, y se cuadruplicaron en Argentina (BBC Mundo, 2013). Las proyecciones que muestran que esta tendencia va a mantenerse durante los próximos años.

Este progreso viene acompañado por la revolución de los celulares inteligentes, que han provocado un importante avance en el desarrollo de sensores, motivando una reducción en tamaño para su integración y un bajo costo para poder ser insertados en el mercado. Adicionalmente, muchos nuevos sensores han sido lanzados en los últimos años. Por ejemplo, gracias al sensor Kinect de Microsoft o su equivalente de Asus, la cámara de profundidad se ha convertido en un elemento popular y accesible. A partir de este es posible obtener imágenes en 3D para creación de mapas, detección de personas, y reconocimiento de actividades de interés. Las principales Universidades del mundo, y los centros de desarrollo e investigación más prestigiosos cuentan con proyectos que lo utilizan para el desarrollo de robots móviles y aplicaciones de las más diversas.

Otro actor importante en la robótica es el Robot Operating System, o simplemente ROS, que con su nacimiento facilita el reúso de código relacionado a diversos tipos de robots, y brinda las herramientas necesarias para el desarrollo de este tipo de proyectos. Este nuevo Sistema Operativo (tiene sólo 5 años) se encuentra en constante desarrollo y muestra una comunidad de desarrolladores muy activa y en constante crecimiento.

Teniendo en cuenta los precios accesibles de los componentes y las tendencias indicadas, se puede decir que es el momento apropiado para introducirse a esta área de investigación en pleno auge y, gracias al diseño modular, por componentes, impuesto por el sistema operativo ROS, crear nuevas bases para el desarrollo de futuros proyectos en la Universidad.

La motivación personal de los alumnos viene dada principalmente por un curso dictado por Sebastian Thrun¹, en el cual se enseñaron los principios básicos, teóricos y prácticos, sobre programación relacionada a la mayor parte de los sistemas que componen a un auto robótico (o self-driving car). En la clase se presentaron los métodos básicos de Inteligencia Artificial para robotica como inferencia probabilística, planificación y búsqueda de caminos, localización, seguimiento (tracking) y control. Todos estos conceptos fueron aplicados sobre ejemplos prácticos y tareas a realizar. Ambos alumnos aprobaron el curso con la más alta distinción.

Además de las motivaciones personales, existen necesidades e intereses propios de los laboratorios por incluir estas nuevas tecnologías en sus proyectos de desarrollo e investigación, y además en la enseñanza de estas nuevas tecnologías. Éstos plantean como objetivos generales, lograr que el software para robots sea reusable, y buscan no solo mejorar el proceso de desarrollo, sino su tiempo y la fiabilidad de estos sistemas. Cabe mencionar que el prototipo desarrollado quedará disponible para ser usado en el Laboratorios de Arquitectura de computadoras y en el Grupo de Robótica y Sistemas Integrados de la Universidad Nacional de Córdoba.

5.3 Metodología

La información sobre el contexto y las necesidades de los clientes se obtuvieron a través de reuniones y entrevistas (ver 6.2).

Por otro lado, la metodología de desarrollo adoptada para el proyecto es Iterativo e Incremental (ver 10). La razón de la elección es la agilidad que brinda al proceso de desarrollo, dada la poca claridad que existe en la definición de requerimientos y la alta probabilidad de cambio que se tiene dado el grado de innovación del proyecto.

La idea básica de este método es el desarrollo del sistema a través de ciclos que se repiten (Iterativo) y en pequeñas porciones de tiempo (Incremental). Durante el desarrollo, más de una iteración del ciclo de desarrollo puede estar en progreso al mismo tiempo. Esto permite a los desarrolladores trabajar paralelamente e ir aprendiendo a medida que se desarrollan las partes, así como también de versiones anteriores. Además, en las primeras versiones se pueden tener prototipos que luego son desechados para volver a ser construidos en etapas posteriores considerando cambios y mejoras.

Dicho modelo permite satisfacer necesidades inmediatas del cliente mediante la entrega incremental del producto, y genera aprendizaje y entrenamiento continuo en el equipo de desarrollo.

Los requerimientos fueron validados mediante pruebas experimentales y prototipos, presentadas a lo largo de diversas reuniones con el equipo cliente (ver 6.1).

¹ Sebastian Thrun es un profesor e investigador de la Universidad de Stanford, un Google Fellow y es miembro de la National Academy of Engineering de Estados Unidos y de la German Academy of Sciences de Alemania. Thrun es más conocido por sus trabajos en robótica, especialmente por el auto capaz de manejarse solo. Además, fue el líder de los equipos de robots móviles autónomos de Stanford y de Google.

6 Contexto de trabajo

6.1 Situación del grupo cliente

El grupo cliente está compuesto por grupos de investigación de los laboratorios de Arquitectura de Computadoras, y el Grupo de Robótica y Sistemas Integrados de la Facultad de Ciencias Exactas Físicas y Naturales. El primero está vinculado a la investigación en arquitecturas de computadoras, seguridad informática, comunicaciones e ingeniería de software, mientras que el segundo se relaciona con las áreas de robots móviles autónomos, robótica en medicina, navegación por visión artificial, procesamiento de imágenes, sistemas de tiempo real, entre otros. Sus proyectos en algunas ocasiones surgen dada la libertad en la cual se desenvuelven, y este es uno de ellos, ya que el proyecto nació por una motivación personal de los alumnos, y luego fue planteado a los clientes con éxito.

6.2 Obtención de requerimientos

Durante la primera fase del proyecto, se delimitaron las responsabilidades de los desarrolladores y clientes, durante un tiempo en el cual ambos presentaron sus ideas y propuestas (ver Anexo 14.1), hasta llegar a un acuerdo sobre qué era conveniente realizar y cómo se pensaba que se iba a llevar a cabo el proyecto.

Las técnicas de obtención de requerimientos se describen a continuación:

- **Entrevistas:** Durante toda la vida del desarrollo del proyecto hubo una comunicación fluida con los clientes. Durante el proceso de la entrevista con los clientes (quienes toman las decisiones) se recopilaron datos para la obtención de requerimientos, escuchando metas, opiniones, y procedimientos informales. Como complemento, algunas veces se realizaron consultas y diálogos mediante correos electrónicos. Las entrevistas realizadas fueron de dos tipos: abiertas y cerradas.
Las primeras, les brindaron a los clientes la posibilidad de expresar sus objetivos y determinar los puntos que consideraban más importantes. Las segundas, fueron utilizadas durante el desarrollo para indagar al cliente sobre ciertas cuestiones que debían resolverse.
- **Releases²:** A medida que se fueron logrando avances en el proyecto, se hicieron entregas y demostraciones de cada incremento a los clientes. Dicha forma de entrega está muy relacionada a los procesos de desarrollo ágiles, y permite que el cliente esté todo el tiempo enterado del estado del desarrollo. Además provee una realimentación constante que arroja nuevas ideas y correcciones que el cliente considere necesarias, y valida todas las suposiciones hechas por el desarrollador.

El cambio de rumbo fue una constante durante las primeras fases del proyecto. Esto se debió a la falta de experiencia de los desarrolladores relacionada a la robótica y a las innovadoras y cambiantes tecnologías que se utilizaron en el desarrollo. Esto a veces produjo una falta de claridad en la definición de requerimientos funcionales.

² Software Release, (o entregas de software), es el proceso de entregas de software nuevo o de actualizaciones del software. También aplicable al hardware.

7 Marco teórico

7.1 Robot

Se denomina robot a un agente electro-mecánico controlado ya sea por un programa que se ejecuta en una computadora o por un circuito electrónico. Pueden ser autónomos o semi autónomos y sus aspectos varían según la función para la cual fueron diseñados. Van desde simples atornilladores hasta humanoides capaces de realizar tareas muy complejas.

La palabra robot fue utilizada con este significado por primera vez en 1920, en una novela del escritor Checo Karel Capek. En su lengua original la palabra “robot” significa servidumbre, y esclavitud. Justamente, las tendencias tecnológicas actuales, se utilizan para construir robots de servidumbre y que realicen tareas en las que se puede reemplazar el trabajo del hombre. Los primeros robots fueron construidos para realizar las tareas más simples y repetitivas, pero a medida que la tecnología disponible y la capacidad de procesamiento aumentaban, las tareas realizadas por robots se fueron haciendo cada vez más complejas.

Como se mencionó anteriormente, los robots generalmente son controlados por una computadora. Es común utilizar la analogía de la computadora como cerebro del robot, de los actuadores como sus músculos y de los sensores como sus sentidos (en referencia a los sentidos del olfato, vista, tacto, entre otros).

Haciendo una breve descripción de lo anterior, un actuador es un dispositivo mecánico que transforma un algún tipo de energía en movimiento (eléctrica, hidráulica, neumática, etc). En ingeniería electrónica se los clasifica como una subdivisión de los transductores que transforman una señal eléctrica de entrada en movimiento. Por otro lado, se denomina sensor a todo dispositivo capaz de medir cantidades físicas y transformarlas a una señal electrónica que puede ser leída por un observador o un instrumento.

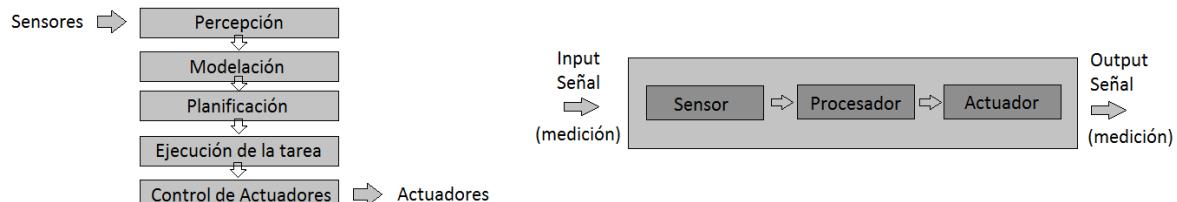


Ilustración 1: Relación entre sensores y actuadores.

La siguiente tabla muestra algunos de los más comunes:

Propiedad Física	Sensor
Contacto	Switch, sensor de contacto
Distancia	Ultrasonido, radar, infrarrojo, láser, cámara estéreo o de profundidad
Luz	Diodo infrarrojo, fotorresistencia
Nivel de luz	Cámara
Sonido	Micrófono
Temperatura	Termómetro, Termocupla, infrarrojo
Orientación	Magnetómetro, giróscopo
Altitud	Altímetro

Localización	GPS
Presión	Barómetro
Corriente Eléctrica	Amperímetro
Magnetismo	Magnetómetro, Sensor de efecto Hall
Voltaje	Voltímetro

Tabla 1: Muestra la relación entre cada sensor y la propiedad física que mide.

Teóricamente, un robot se compone de dos partes: la de control (procesamiento computacional, señales de bajo voltaje y corriente, etc.) y el circuito de potencia (altos voltajes y corrientes para el funcionamiento de los actuadores). Siempre es conveniente aislar ambas partes por razones de seguridad y de mantenimiento. Generalmente se utilizan opto-acopladores para el aislamiento de la parte electrónica de la de potencia, de tal manera que las altas intensidades de corriente de la parte de potencia no puedan dañar la computadora o el controlador del robot.

7.1.1 Clasificación de los robots

Los robots pueden ser móviles o estacionarios. Dentro de la clasificación de robots móviles se encuentran los que ruedan, los que se arrastran, los que nadan, entre otros. En cuanto a los estacionarios, algunos ejemplos son los brazos robóticos, la cara robótica, robots industriales, etc. Estos últimos, aunque son categorizados como estacionarios, no necesariamente están quietos, generalmente su operación está confinada a un área de trabajo delimitada.

Los robots móviles se clasifican según el movimiento que son capaces de realizar, existen dos tipos *holonómicos* y *no-holonómicos*. Para una mejor descripción sobre esta clasificación, es necesario hablar de grados de libertad:

“El grado de libertad de un sistema es un número que indica la cantidad de parámetros que pueden variar independientemente. En mecánica (y robótica), estos son los parámetros que definen el estado del sistema”.

Por ejemplo, considerando la posición de un auto sobre un plano 2D, éste tendría 3 grados de libertad: las coordenadas (x,y) que definen cualquier punto sobre dicho plano y la orientación³. Por otro lado, un cuerpo rígido en el espacio 3D, requiere de seis valores para representar su estado: tres traslaciones (x, y, z) y tres rotaciones (yaw, pitch, roll)⁴.

³ Grados de libertad en espacio 2D (5-5-13) [http://en.wikipedia.org/wiki/Degrees_of_freedom_\(mechanics\)](http://en.wikipedia.org/wiki/Degrees_of_freedom_(mechanics))

⁴ Grados de libertad en espacio 3D (5-5-13) <http://oxtool.blogspot.com.ar/2012/11/simple-six-axis-adjustment-system.html>

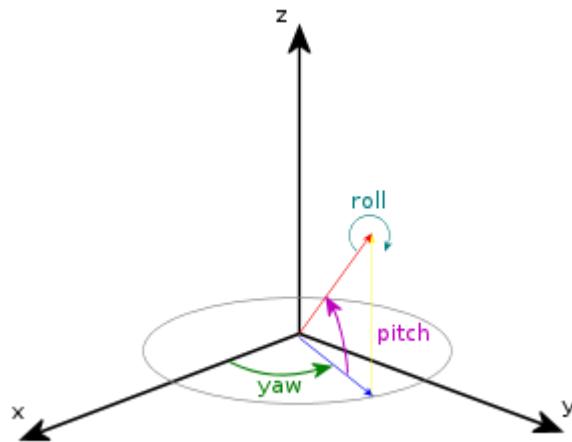


Ilustración 2: Grados de libertad en un espacio en 3D

7.1.1.1 Robots holonómicos y no-holonómicos

Estos términos hacen referencia a la relación entre la cantidad total de grados de movimiento de un robot y cuáles de ellos son controlables. En los casos en los que es posible controlar todos los grados de libertad, el robot se denomina holonómico. En caso contrario es no-holonómico.

1 Robot holonómico:

Este es el caso de un robot construido con todas sus ruedas esféricas u omnidireccionales. Este tipo de robots pueden moverse en cualquier dirección.

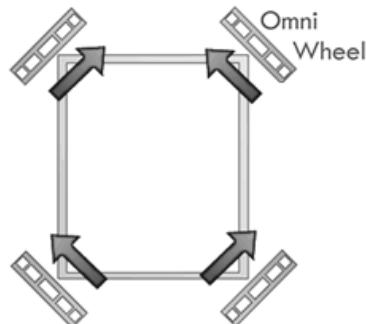


Ilustración 3: Robot holonómico. El robot es capaz de desplazarse en dos direcciones y, además, puede rotar sin desplazarse⁵

La siguiente figura muestra un ejemplo de dicho tipo de robots:



Ilustración 4: Ejemplo de robot holonómico⁵

5 Robot holonómico (1-5-13) <http://www.alpha-crucis.com/es/intermedio/2759-3wd-100mm-omni-wheel-mobile-robot-kit-3700386100130.html>

2 Robot no holonómico

En este caso, el robot no puede desplazarse según el eje Y sin hacerlo en el eje X. Tampoco es capaz de rotar sin desplazarse.

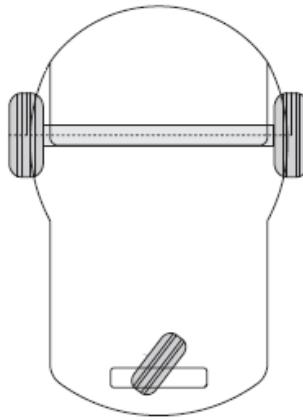


Ilustración 5: Robot no holonómico. No es posible controlar todos los grados de libertad del robot.

Dentro de este grupo, los modelos cinemáticos más comunes son: diferencial y Ackerman.

a) Modelo Diferencial

Los robots que utilizan este modelo cuentan con un mínimo de dos ruedas, cada una con su motor independiente. Para lograr desplazamientos, ambas ruedas giran en el mismo sentido, horario o anti horario, ya sea para avanzar o retroceder. Para rotaciones en el lugar, ambas ruedas deben rotar en sentidos opuestos y a la misma velocidad. En cuanto a las rotaciones con desplazamiento, cada rueda debe girar a una velocidad diferente para lograr un desvío. A continuación se muestra el modelo físico-matemático correspondiente.

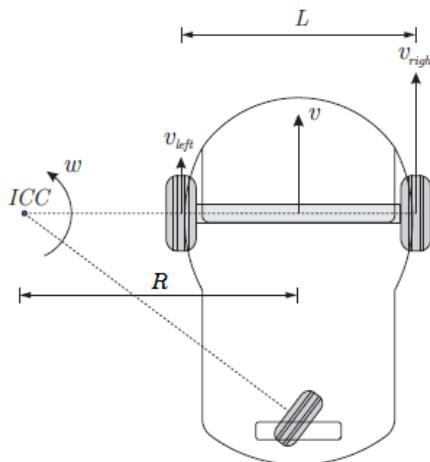


Ilustración 6: Modelo cinemático diferencial con dos ruedas.

En la imagen anterior se muestra un robot que utiliza el modelo cinemático diferencial con dos ruedas. Se denota con V_{right} y V_{left} a las velocidades de las ruedas derecha e izquierda, respectivamente. Se marca con ICC el centro de curvatura instantáneo a partir del cual se toma el radio de giro en cada momento. La velocidad angular se representa con W .

Continuando, las siguientes ecuaciones muestran la relación entre la velocidad angular del robot y la velocidad en cada rueda:

$$V_{Right} = w(R + \frac{L}{2})$$

$$V_{Left} = w(R - \frac{L}{2})$$

Ya que la velocidad angular w es la misma en ambas ecuaciones, se las puede igualar y obtener una ecuación que representa la relación entre el radio y las velocidades de cada rueda:

$$R = \frac{L}{2} * \frac{(V_{Right} + V_{Left})}{(V_{Right} - V_{Left})}$$

A partir de las ecuaciones anteriores, se pueden extraer tres casos particulares:

- **Movimiento rectilíneo:** para lograr un movimiento rectilíneo, sin desvíos, el radio de rotación debe ser infinito.

$$R = \infty \rightarrow V_{Right} = V_{Left}$$

Lo cual implica que la velocidad en ambas ruedas debe ser igual para lograr un movimiento rectilíneo.

- **Rotación sin desplazamiento:** para lograr una rotación sobre un eje fijo, el centro del robot, sin desplazamiento, el radio debe ser 0.

$$R = 0 \rightarrow V_{Right} = -V_{Left}$$

En consecuencia, para que el robot gire sobre su propio eje, la magnitud de la velocidad debe ser igual en ambas ruedas pero en sentidos contrarios.

- **Rotación respecto a una de las ruedas:** para desplazar el eje giro hacia una de las ruedas, esta debe tener velocidad nula y la otra rueda un valor de velocidad no nulo.

$$V_{Right} = 0 \text{ y } V_{Left} \neq 0$$

ó

$$V_{Left} = 0 \text{ y } V_{Right} \neq 0$$

b) Modelo Ackerman

Una alternativa al modelo diferencial, es el modelo de Ackerman. Se trata de una cinemática muy popular ya que es la utilizada en los automóviles comerciales. Como mínimo, se cuenta con un motor para tracción y otro para dirección. Las ruedas traseras se encargan de avanzar o retroceder, mientras que las delanteras determinan la dirección o el ángulo de curvatura. Este método requiere de mucho espacio físico para ciertas maniobras muy comunes en el campo de la robótica, como por ejemplo invertir la dirección de avance.

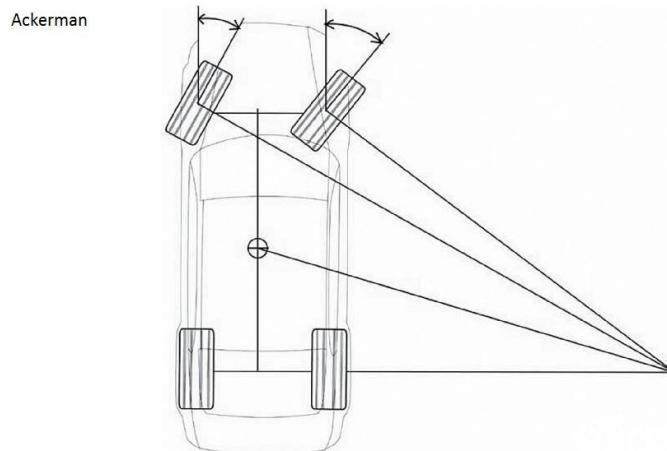


Ilustración 7: Mecanismo de giro en el modelo cinemático de Ackerman

7.2 Microcontroladores y placas de desarrollo

Usualmente abreviado como μ C, un microcontrolador es un circuito integrado programable, capaz de ejecutar una serie de instrucciones grabadas en su memoria. Está constituido por tres unidades funcionales: unidad central de procesamiento, o procesador, memoria y periféricos de entrada/salida. Dichos microcontroladores hoy en día se encuentran por todas partes en nuestra vida cotidiana, desde un auto hasta un celular o unos lentes inteligentes.

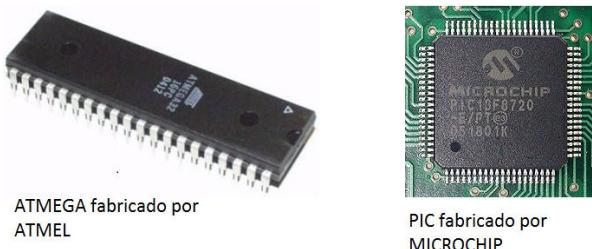


Ilustración 8: Microcontroladores más conocidos

Existen plataformas de hardware programables, en las cuales se integran los microcontroladores junto con otros periféricos y memorias que facilitan el uso de la electrónica para proyectos multidisciplinares.

Un ejemplo de ellos, y quizás la más apropiada para proyectos de robótica es la plataforma de hardware libre Arduino. Ésta se compone de una placa con un microcontrolador y un entorno de desarrollo diseñado para facilitar la interacción entre software y hardware a bajo nivel. Los microcontroladores más usados por Arduino son de 8 bits y arquitectura RISC, como el Atmel AVR, Atmega168, Atmega328, Atmega1280, ATmega8. Dichos microcontroladores fueron elegidos por su sencillez y bajo costo, y permitieron la fabricación de múltiples diseños de placas de desarrollo. Desde fines de 2012, Arduino ha comenzado a utilizar en sus placas otros microcontroladores de 32 bits como el CortexM3 de ARM.

Debido a su bajo costo, facilidad de uso y una comunidad activa que los utiliza, Arduino es la primera opción en diversos escenarios tales como los proyectos de robótica, educativos, domótica, etc.

7.3 ROS – Robot Operating System

Robot Operating System, o simplemente ROS, es un sistema operativo que provee librerías y herramientas que brinda a los desarrolladores de software todo lo necesario para crear aplicaciones relacionadas a la robótica. Este permite a los desarrolladores de software programar con un buen nivel de abstracción sobre el hardware, permitiendo controlar dispositivos mediante librerías, visualizadores, herramientas para el paso mensajes, manejo de paquetes, etc.

No es un sistema operativo en el sentido tradicional de administración y planificación de procesos, sino que provee una capa de comunicaciones estructuradas por encima del sistema operativo anfitrión.

El desarrollo de este framework para robótica comenzó en el año 2007 en el Laboratorio de Inteligencia Artificial de Stanford, pero a partir del año 2008, el desarrollo se lleva a cabo principalmente en Willow Garage, un instituto de investigación/incubadora de robótica, que tiene más de 20 instituciones que colaboran según un modelo de desarrollo federal.

Actualmente, Windows, Linux y Mac están entre los sistemas operativos soportados pero todos, excepto Ubuntu, figuran en estado “experimental”. Todo el software publicado en los repositorios se encuentra disponible bajo la licencia de código abierto BSD. El objetivo principal de este proyecto es agilizar el proceso de creación de aplicaciones robóticas utilizando una plataforma común.

ROS está basado en una arquitectura de grafo en la cual el procesamiento es llevado a cabo en nodos que se comunican mediante el paso mensajes. Estos mensajes pueden ser definidos para llevar cualquier tipo de información, desde simples números indicando valores de sensores hasta imágenes o cualquier tipo de estructura de datos. Para recibir mensajes, los nodos deben suscribirse a temas. Y para enviar mensajes a otros nodos, simplemente se publica hacia el tema deseado. Los nodos que forman parte de un sistema de ROS no necesariamente se ejecutan todos en una misma computadora, también funciona sobre sistemas distribuidos con varias máquinas comunicándose entre sí.

A continuación se definen algunos conceptos básicos de ROS:

- **Packages (rospack):** se crean colecciones mínimas de código para ser reusadas.
- **Stacks (rosstack):** los paquetes están organizados en otras estructuras de mayor jerarquía organizados en stacks que permiten compartir código fácilmente. Los stacks juntan paquetes que proveen funcionalidad en su conjunto.
Son simplemente un directorio con un archivo stack.xml, y todos los paquetes de ese directorio se consideran parte del stack.
Tanto los stacks como los paquetes, tienen un archivo llamado “manifest” que describe el paquete (manifest.xml), o el stack (stack.xml) en el cual se definen las dependencias entre stacks o paquetes.
- **Nodes:** un nodo es un programa ejecutable de un paquete, que usa la “ROS client library” para comunicarse con otros nodos.
 - Los nodos pueden suscribirse o publicar en un Topic.
 - Los nodos pueden proveer o usar un Service.

- **Topics (temas):** Para enviar o recibir mensajes, los nodos deben suscribirse a un tema. Por ejemplo si se crea un tema llamado “comandos”, tanto el nodo que los envía como el que los recibe se debe suscribir a “comandos”.



Ilustración 9: Comunicación entre nodos en ROS a partir del tema (topic) llamado “command_velocity”. Obtenida con la herramienta de ROS rxgraph. Este ejemplo corresponde a uno de los tutoriales más básicos de ROS.

La imagen mostrada fue generada por uno de los visualizadores de ROS llamada rxgraph que permite ver cada nodo del sistema y su interacción. En este caso se muestra el grafo actual formado por los nodos en ejecución. Vemos una arista conectando los nodos, esta representa la comunicación entre ellos mediante el paso de mensajes utilizando el tema “command_velocity”.

Por el momento, cada nodo de ROS puede ser programado con cualquiera de los siguientes lenguajes, que son muy distintos entre sí: C++, Python, Octave y LISP. Pero además se está trabajando para la inclusión de más lenguajes.

La especificación de ROS es a nivel de la capa de mensajes, en la cual se define el tipo de estructuras de datos que conformarán cada tipo de mensaje. Esto permite que varios nodos que fueron programados en distintos lenguajes puedan comunicarse. Para especificar la composición de los mensajes, se utiliza IDL (interface definition language) que tiene el siguiente aspecto:

```
Header header
Point32[] pts
ChannelFloat32[] chan
```

Ilustración 10: Composición de cada mensaje en ROS (Morgan Quigley, Brian Gerkeyy, Ken Conley, 2009).

A partir de esto, los generadores de código crean implementaciones nativas en el lenguaje usado para obtener un objeto que será serializado y deserializado por ROS a medida que se envían y reciben mensajes (Morgan Quigley, Brian Gerkeyy, Ken Conley, 2009).

7.3.1.1 Herramientas básicas de ROS

A continuación se presentan algunas herramientas de utilidad que provee el sistema operativo. Se describen aquellas que fueron de mayor utilidad para el proyecto.

7.3.1.1.1 Rqt_graph (sucesor de rxgraph)

Es un plugin para la GUI (Interfaz gráfica de usuario) que permite obtener una visualización del grafo computacional de ROS. Es decir, que muestra todos los nodos que se están ejecutando y su interacción a través de los temas que los conectan. A continuación se muestra un ejemplo de la ventana de visualización.

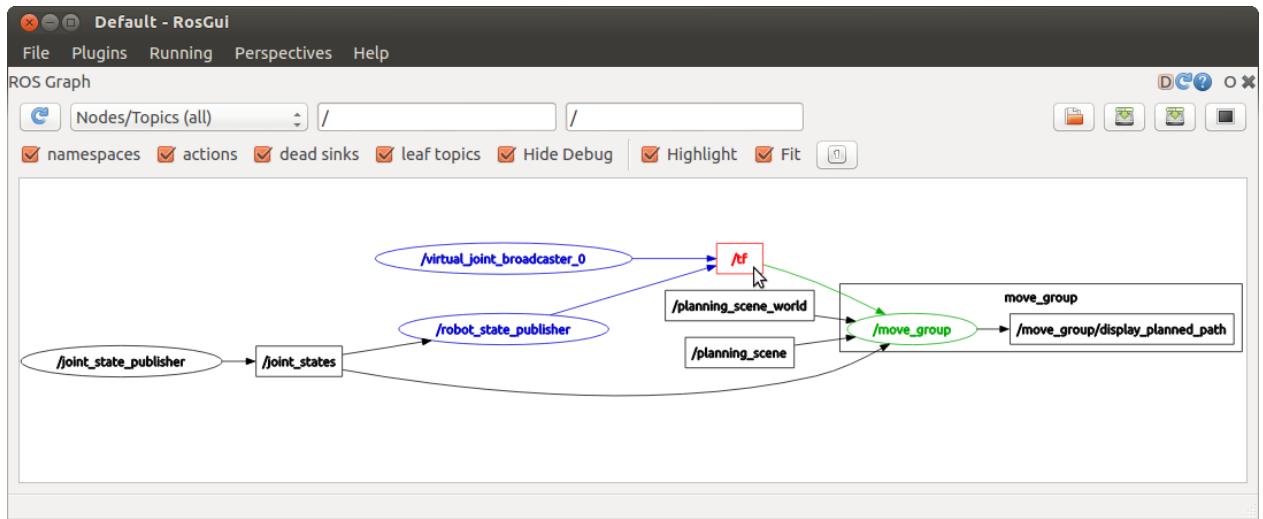


Ilustración 11: Ejemplo de rqt_graph⁶.

7.3.1.1.2 TF - librería para transformaciones

Es un paquete que permite la utilización de muchos sistemas de coordenadas y el seguimiento de cada uno de estos a lo largo del tiempo. La relación entre distintos sistemas de coordenadas se mantiene mediante una estructura de árbol que hace posible la transformación de puntos, vectores, etc. entre cualquier par de sistemas.

Un sistema robótico generalmente consiste en un conjunto de sistemas de coordenadas de tres dimensiones (3D) que cambian a través del tiempo. Por ejemplo, en un humanoide, se tiene un marco para el mundo, otro para los pies, brazos, cabeza, etc. En ese caso, TF se encarga del seguimiento de cada marco y de las transformaciones entre ellos.

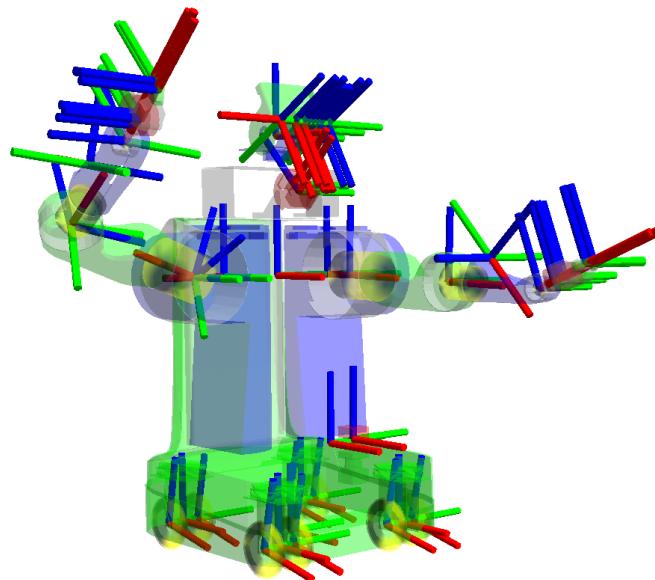


Ilustración 12: Diferentes sistemas de coordenadas necesarios en un robot complejo⁶

⁶ Imagen extraída de http://www.ros.org/wiki/rqt_graph (3-5-13)

7.3.1.1.3 RVIZ –visualización

Rviz es una herramienta de ROS dedicada a la visualización. Cuenta con muchos tipos de displays, cada uno capaz de mostrar distintos tipos de datos. A continuación se muestra una tabla con los displays relevantes para el presente proyecto:

Nombre	Descripción	Mensaje usado
Axes	Muestra un set de ejes	
Camera	Crea una nueva ventana de renderizaje desde la perspectiva de la cámara y superpone su imagen	sensor_msgs/Image sensor_msgs/CameraInfo
Grid	Muestra una red o cuadricula 2D o 3D	
Grid Cells	Dibuja celdas de un Grid que generalmente representan obstáculos a partir de un mapa de costos (proveniente del stack de ROS de navegación)	nav_msgs/GridCells
Image	Ventana con una imagen	sensor_msgs/Image
InteractiveMarker	Muestra objetos o marcadores en 3d	visualization_msgs/InteractiveMarker
Laser Scan	Muestra información proveniente de un laser	sensor_msgs/LaserScan
Map	Muestra un mapa en el plano	nav_msgs/OccupancyGrid
Markers	Permite a programadores mostrar primitivas	visualization_msgs/Marker visualization_msgs/MarkerArray
Path	Muestra un camino	nav_msgs/Path
Pose	Dibuja una pose en forma de flecha o eje	geometry_msgs/PoseStamped
Point Cloud	Visualización de una nube de puntos	sensor_msgs/PointCloud , sensor_msgs/PointCloud2
Polygon	Muestra el contorno de un polígono utilizando líneas	geometry_msgs/Polygon
Odometría	Acumula poses de odometría	nav_msgs/Odometry
TF	Muestra la jerarquía de las transformaciones	

Ilustración 13: Diferentes tipos de display con los que cuenta RViz

La herramienta Rviz tiene varios modos de visualización que se describen a continuación:

- **Modo orbital (modo por defecto)**

En el modo orbital la cámara simplemente rota alrededor de un punto focal, el cual es visible durante los movimientos. Los comandos de rotación o desplazamiento para interactuar con el visualizador a través del mouse son:

- Botón izquierdo*: rotación. Apretando y arrastrando el mouse se rota la cámara.
- Botón del medio*: Desplazamiento del punto focal. Apretando y arrastrando se desplaza sobre el plano formado por el vector “arriba” y “derecha” de la cámara.
- Botón derecho y rueda de scroll*: Permite hacer zoom respecto al punto focal

- **Modos FPS (primera persona)**

Las rotaciones se realizan como si la cámara fueran los ojos de una persona y ésta rotara su cabeza.

- Botón izquierdo*: Rotación. Con control-click se selecciona un objeto y se lo mira directamente.
- Botón del medio*: Desplazamiento del punto focal. Apretando y arrastrando se desplaza sobre el plano formado por el vector “arriba” y “derecha” de la cámara.
- Botón derecho y rueda de scroll*: Haciendo click y arrastrando se avanza o retrocede sobre el vector “forward” de la cámara.

- **Modo ortográfico (desde arriba hacia abajo)**

La cámara siempre mira hacia abajo sobre el eje Z y, al ser, ortográfico, los objetos no se hacen más pequeños a medida que se alejan.

- Botón izquierdo*: rotación alrededor del eje Z.
- Botón del medio*: haciendo click y arrastrando el mouse se desplaza sobre el plano XY.
- Botón derecho y rueda de scroll*: permite hacer zoom.

7.4 Odometría

Se denomina odometría al uso de datos provenientes de sensores en movimiento para estimar el cambio en la posición de un sistema a largo de un período de tiempo. La odometría es comúnmente utilizada para calcular la posición relativa de robots móviles.

En el presente trabajo, el estado del robot queda definido por tres variables: distancia en X, distancia en Y, y ángulo de orientación ϕ denominado “heading” (hacia donde mira el robot).

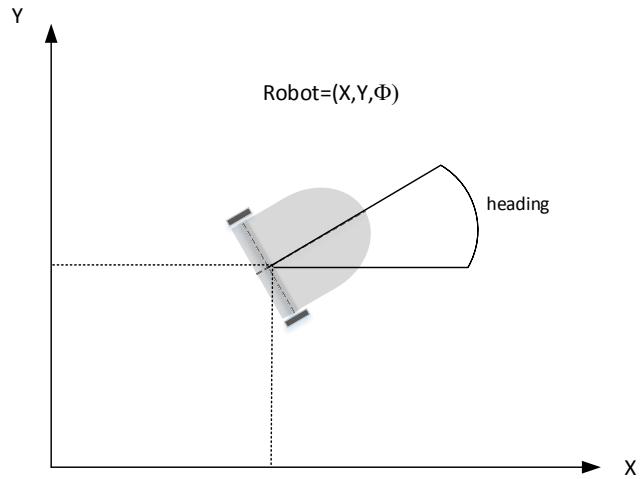


Ilustración 14: Muestra el posicionamiento del robot en un Sistema de coordenadas.

Con el fin de estimar estos valores que definen la posición del robot, luego de cada desplazamiento o rotación, se utiliza alguna combinación de los siguientes sensores: encoder, magnetómetro, o giróscopo. Bajo estos conceptos se hace posible la implementación de los requerimientos vinculados a la localización y el control del robot.

7.4.1 Odometría visual

En robótica y visión de computadoras, la odometría visual consiste en el proceso de estimación de la posición y la orientación del robot a través del análisis de imágenes tomadas con cámaras asociadas. En otras palabras, se estudia el efecto del movimiento sobre las imágenes producidas por las cámaras a bordo del robot.

Algunos factores que podrían dificultar el proceso son la falta de iluminación, escenas con poca textura y escenas con muchos objetos en movimiento. Además, para poder identificar características en dos imágenes consecutivas y correlacionarlas, es necesario que se superpongan los suficientemente bien como para no producir errores o deformaciones en la imagen.

A diferencia de las técnicas tradicionales de odometría que no pueden ser aplicadas a robots con piernas similares a las de un animal, cualquier tipo de robot con una cámara puede realizar odometría visual. En estos casos las estimaciones no se verán afectadas por un posible deslizamiento de las ruedas del robot en terrenos no parejos, en arena, o algún otro.



Ilustración 15: El terreno produce deslizamientos de las ruedas, que resultan en una mala odometría por parte de los Encoders. En estos casos, la odometría visual produce mejores resultados

Algunos robots complementan la odometría visual con el sistema de posicionamiento global GPS basado en localización a partir de satélites, odometría tradicional u odometría por láser. En ambientes bajos donde no es posible utilizar GPS, en interiores o bajo el agua, la odometría visual cumple un papel fundamental.

7.5 Sensor Kinect

Kinect es un sensor que fue originalmente diseñado para la industria de los videojuegos con el fin de captar e interpretar movimientos y gestos de los jugadores a través de la cámara.

Una vez en el mercado, a fines del año 2010⁷, rápidamente capto la atención de consumidores y de comunidades de desarrollo de software, principalmente las relacionadas con la robótica y la visión de computadoras. Las posibilidades brindadas por el sensor y su bajo costo (aproximadamente U\$S 150) lo convirtieron en un producto de consumo masivo y se vendieron alrededor de 8 millones de unidades en 60 días⁸. En cuestión de semanas aparecieron controladores para utilizar el dispositivo independientemente de los productos de Microsoft⁹.

PrimeSense, la empresa que diseño el sistema de información de profundidad de Kinect, liberó drivers que hoy se encuentran bajo responsabilidad de la organización OpenNI¹⁰. Esta organización, sin fines de lucro, se encarga de certificar y mejorar el funcionamiento de dispositivos de interfaces naturales. A partir de este controlador, el desarrollo basado en Kinect aumentó notablemente y esto es visible en los sitios web de las librerías abiertas OpenCV (visión de computadoras), PCL (Point Cloud Library), y en el sistema operativo de robótica ROS (Robot Operating System).

Actualmente, Microsoft extendió sus ramas de desarrollo basadas en Kinect y las tres principales son:

- **Kinect – Xbox:** videojuegos sin necesidad de controles. El juego responde a los movimientos de los jugadores y se comanda la consola a través de comandos de voz¹¹.
- **Kinect for Windows:** como dice el sitio oficial, Kinect for Windows le da a Windows ojos, oídos y la capacidad de usarlos. Se busca un salto en la forma que la gente interactúa con la tecnología¹². Con el objetivo de agilizar el desarrollo, se creó un SDK (Software Development Kit) disponible para el público, que incluye controladores y proyectos de ejemplo con código fuente. Los lenguajes de programación utilizados son C++, C# y Visual Basic.
- **Robotics Developer Studio:** se brindan herramientas para facilitar el desarrollo de software de robótica¹³.

7.5.1.1 Especificaciones del Kinect

El sensor no solo cuenta con varias cámaras, sino que también posee micrófonos para recibir comandos por voz y estimar la dirección desde la cual provienen. Más específicamente, el sensor cuenta con los siguientes componentes de hardware:

- Cámara RGB que almacena 3 canales de datos con una resolución de 1280x960 (color sensor en la figura).

7 (20-4-13) <http://en.wikipedia.org/wiki/Kinect>

8 (20-4-13) <http://www.microsoft-careers.com/content/rebrand/hardware/hardware-story-kinect/>

9 (20-4-13) <http://www.bbc.co.uk/news/technology-11742236>

10(20-4-13) <http://en.wikipedia.org/wiki/OpenNI>

11 (20-4-13) <http://www.microsoft.com/en-us/kinectforwindows/>

12 (20-4-13) <http://www.xbox.com/en-US/kinect>

13 (20-4-13) <http://www.microsoft.com/robotics/>

- Un emisor infrarrojo IR y un sensor de profundidad IR. El emisor emite rayos de luz infrarrojos, y el sensor de profundidad se encarga de leer los haces IR que se reflejan en el sensor. Los haces reflejados son convertidos a información de profundidad midiendo la distancia entre el objeto y el sensor.
- Arreglo múltiple de micrófonos que contiene 4 micrófonos para captura de sonidos.
- Acelerómetro de 3 ejes configurado para un rango de 2G, donde G es la aceleración debida a la gravedad. Se lo utiliza para determinar la orientación del sensor en todo momento.

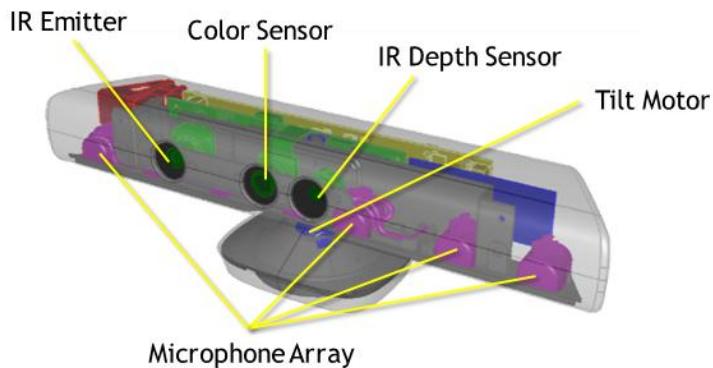


Ilustración 16: Sensores presentes en Kinect

La siguiente tabla muestra las especificaciones técnicas del sensor.

Kinect	Especificaciones
Angulo de visión	Campo de visión de 43° vertical y 57° horizontal
Rango de inclinación vertical	±27°
Frecuencia de cuadros (streams de color y de profundidad)	30 marcos por segundo (frames per second, en inglés, o FPS)
Formato de audio	16-kHz, 24-bit mono pulse code modulation (PCM)
Características de entrada de audio	Arreglo de 4 micrófonos con conversor analógico digital de 24 bits (ADC) y procesamiento de señal por hardware que incluye cancelación de eco y eliminación de ruido.
Características del acelerómetro	Acelerómetro de 2G/4G/8G configurado para un rango de 2G, con un límite superior de 1° de precisión.

El sensor genera tres distintos flujos de datos que son enviados hacia la computadora. Dos de estos son la secuencia de imágenes a color y la de profundidad y, el tercero, un flujo del audio y la dirección de la cual proviene:

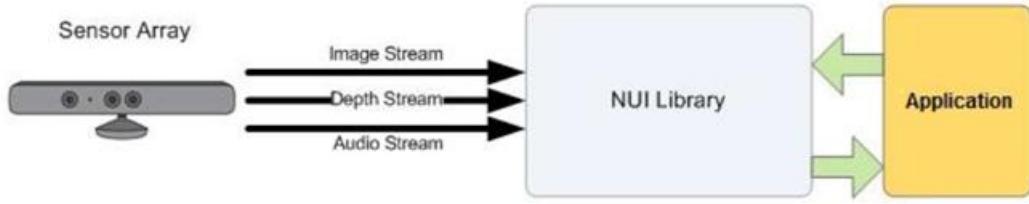


Figura 1 distintos flujos de datos que van desde el sensor Kinect hacia la computadora.

Teniendo en cuenta los objetivos de este proyecto, únicamente resultan relevantes los streams de color y de profundidad.

La siguiente es una impresión de pantalla durante la visualización de las cámaras del sensor hechas sobre Windows y a partir de SDK oficial:

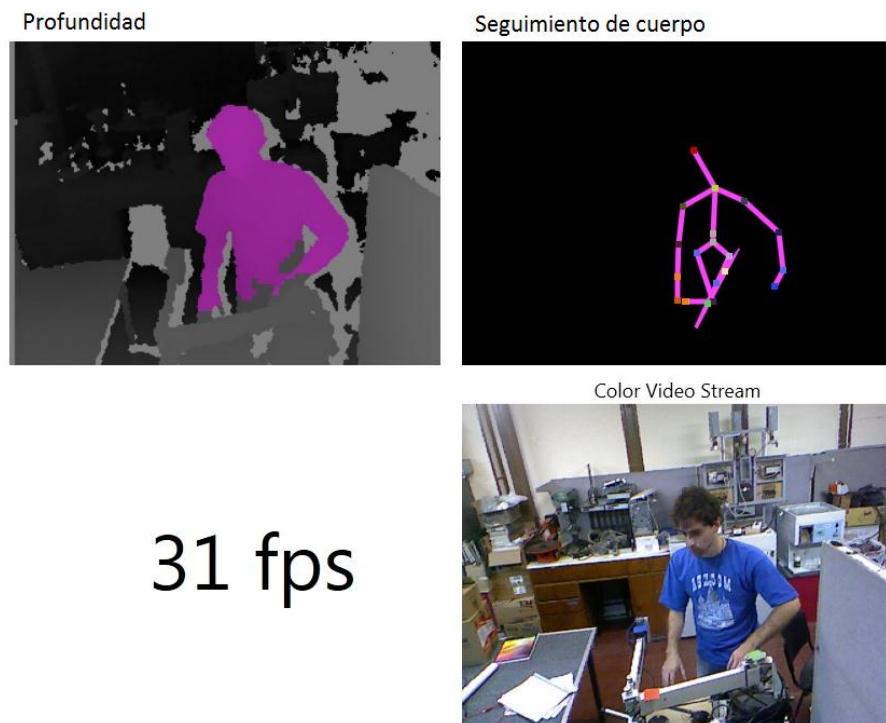


Ilustración 17: Imagen de profundidad (esq. superior izquierda), seguimiento de humanos (esq. superior derecha) y imagen de color (esq. inferior derecha).

7.5.1.1 Stream de color

Se recibe en la computadora una imagen a la vez. Cada una de estas imágenes es un arreglo de bytes en el cual cada pixel de la cámara está representado por 4 bytes. Se utiliza el sistema BGRA de 32 bits:

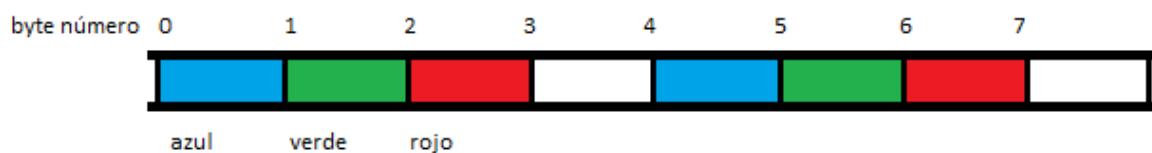


Ilustración 18: Bytes en la imagen de color proveniente de Kinect

Como se ve, cada 4 bytes se define un pixel. Tres de estos 4 indican el color del mismo mediante la combinación de componentes de rojo, verde y azul. La última componente establece la transparencia. Cada una de estas componentes es dada como 8 bits, otorgando un rango entre 0 y 255.

7.5.1.1.2 Stream de profundidad

Al igual que el stream de color, este consiste en una secuencia de imágenes. Pero, en este caso, las imágenes solo tienen colores en la escala del gris (desde el blanco hasta el negro). Cada pixel tiene un valor que indica la distancia a la cual se encuentra de la cámara. Visualmente, los pixeles más cercanos se verán más claros y los más lejanos se verán más oscuros.

Existen varias explicaciones no oficiales para el proceso de obtención de cada imagen de profundidad a partir de los sensores presentes en Kinect. Las empresas dueñas de la patente y de la propiedad intelectual no han revelado el algoritmo ni el proceso específico, pero generalmente, se habla de un escáner de luz estructurada que se usa ya sea para:

- 1) Detectar deformaciones sobre las superficies y así reconstruirlas o
- 2) Dara obtener la correspondencia entre los píxeles de dos imágenes distintas provenientes de un sistema de dos cámaras (cámara estéreo)

El stream de profundidad es una imagen en la cual cada pixel tiene un valor que indica la distancia a la que está ese pixel desde la cámara. Mientras más claro o blanco esté, más cerca de la cámara está el objeto. Los puntos más lejanos se ven en negro. Actualmente, existen dos versiones comerciales del sensor Kinect: Kinect para X-box y Kinect para Windows. En el primero, la distancia máxima capaz de ser detectada correctamente es 4 metros y la mínima 80 centímetros. En el segundo, existe el “modo cercano” en el cual es posible un rango de mediciones de 40 cm hasta 3 metros¹⁴.

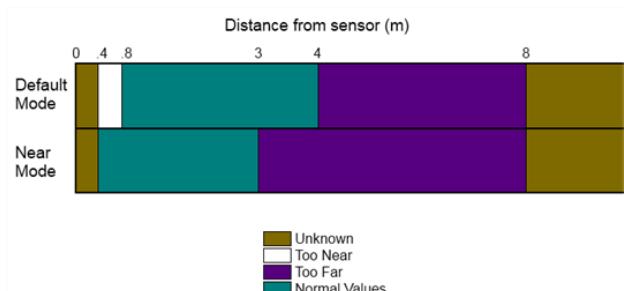


Ilustración 19: Modos cercano y lejano del sensor Kinect for Windows

14 (1-5-13) <http://msdn.microsoft.com/en-us/library/hh438998.aspx>

8 Estudio del problema

8.1 Requerimientos

Para la gestión de requerimientos se usó el software Enterprise Architect, a partir del cual se obtiene el siguiente documento de requerimientos:

8.1.1 Requerimientos funcionales

A continuación se describen cada uno de los requerimientos funcionales del sistema (FSR):

FSR1. Movimientos del robot a partir de comandos básicos

«Functional»

Status: Validated

Priority: High

Difficulty: Low

Phase: 1.0

Version: 1.0

El robot debe ser capaz de desplazarse a partir de 4 comandos básicos con una única velocidad:

- up()
- down()
- right()
- left()

1.a) ST. Prueba de cada comando por separado

«Testing»

Status: Validated

Priority: High

Difficulty: Low

Phase: 1.0

Version: 1.0

Probar para cada uno de los comandos, que el robot se desplace como es de esperarse.

Se deben hacer varios casos de testing que prueben que cada uno de los comandos responde bien por separado y que en la combinación de éstos no interfieran entre sí.

FSR2. Localización del robot

«Functional»

Status: Validated

Priority: High

Difficulty: High

Phase: 1.0

Version: 1.0

El robot debe ser capaz de auto-posicionarse dentro de un sistema de coordenadas, y saber dónde está.

2.a) ST. Comprobar posición del robot para movimientos angulares

«*Testing*» *Status:* Validated *Priority:* Medium *Difficulty:* Low

Phase: 1.0 *Version:* 1.0

Probar los comandos right() y left() para diferentes ángulos y medir el desplazamiento respecto del punto inicial.

2.b) ST. Comprobar posición del robot para movimientos lineales

«*Testing*» *Status:* Validated *Priority:* Medium *Difficulty:* Medium

Phase: 1.0 *Version:* 1.0

Situar el robot en una posición aleatoria dentro de la habitación y probar los comandos up() y down() para diferentes distancias de avance y retroceso. Luego comprobar que las medidas tomadas por el mismo sean correctas en relación a la distancia recorrida en la realidad.

FSR3. Control de trayectoria y velocidad

«*Functional*» *Status:* Validated *Priority:* Medium *Difficulty:* High

Phase: 1.0 *Version:* 1.0

Implementar un controlador al cual se le pueda decir que el robot se mueva a una cierta velocidad, y lo haga sin desviarse de la trayectoria.

3.a) ST. Ajuste de velocidad del robot para todos los movimientos posibles

«*Testing*» *Status:* Validated *Priority:* Medium *Difficulty:* High

Phase: 1.0 *Version:* 1.0

Se debe probar la reacción del robot a algunas velocidades incluyendo los casos extremos, y a su vez esto para cada uno de los comandos.

FSR4. Integrar la computadora a bordo en el robot

«*Functional*» *Status:* Validated *Priority:* High *Difficulty:* Medium

Phase: 1.0 *Version:* 1.0

El robot debe contar con una computadora a bordo para el procesamiento de datos y la generación de mapas. Esta se debe poder comunicar con el microcontrolador que controla el robot y al cual se le envían los comandos básicos.

4.a) ST. Comprobar interface microcontrolador-computadora y comunicación

«*Testing*» *Status:* Validated *Priority:* Medium *Difficulty:* Low

Phase: 1.0

Version: 1.0

Probar el correcto funcionamiento de la computadora, y sus interfaces con otros dispositivos.

FSR5. Medición de distancias con el sensor

«Functional»

Status: Validated

Priority: Low

Difficulty: Low

Phase: 1.0

Version: 1.0

El robot debe ser capaz de medir distancias con una cierta exactitud, mediante algun sensor elegido.

5.a) ST. Comprobar exactitud de mediciones a puntos conocidos

«Testing»

Status: Validated

Priority: Low

Difficulty: Low

Phase: 1.0

Version: 1.0

Situar el robot en varios puntos conocidos, y tomar mediciones a diferentes puntos del mapa, para luego comprobar la exactitud de las mediciones, en contraste con la realidad.

FSR6. Interfaz gráfica para mostrar mapas

«Functional»

Status: Validated

Priority: Low

Difficulty: Medium

Phase: 1.0

Version: 1.0

Interfaz gráfica capaz de mostrar los mapas a medida que son generados, en la computadora a bordo del robot.

6.a) ST. Probar la interfaz con mapas de prueba

«Testing»

Status: Validated

Priority: Low

Difficulty: Low

Phase: 1.0

Version: 1.0

Comprobar que ocurre la generación de mapas a medida que el robot es capaz de ver nuevos sectores.

FSR7. Generar mapas 3D

«Functional»

Status: Validated

Priority: Medium

Difficulty: High

Phase: 1.0

Version: 1.0

Generar mapas 3D del medio por el cual el robot se desplaza.

7.a) ST. Verificar coincidencia entre mapas y la habitación dinámicamente

«*Testing*» *Status:* Validated *Priority:* Medium *Difficulty:* Medium

Phase: 1.0 *Version:* 1.0

Comprobar a través de la interfaz gráfica que los mapas generados se asemejan a la realidad, con el robot en movimiento.

7.b) ST. Probar diferentes escenarios

«*Testing*» *Status:* Validated *Priority:* Low *Difficulty:* Low

Phase: 1.0 *Version:* 1.0

Repetir el proceso para escenarios diferentes.

FSR8. Navegación. Movimientos autónomos a partir de mapas

«*Functional*» *Status:* Validated *Priority:* Medium *Difficulty:* Medium

Phase: 1.0 *Version:* 1.0

A partir de los mapas que se van generando el robot debe ser capaz de planificar su movimiento para llegar a un objetivo dado (es decir, debe tomar decisiones autónomas).

8.a) ST. Comprobar exploración de la habitación

«*Testing*» *Status:* Validated *Priority:* Medium *Difficulty:* Medium

Phase: 1.0 *Version:* 1.0

Comprobar que el cuarto haya sido explorado.

8.b) ST. Evitar obstáculos que permitan circulación

«*Testing*» *Status:* Validated *Priority:* Medium *Difficulty:* Medium

Phase: 1.0 *Version:* 1.0

El robot debe ser capaz de evitar obstáculos dentro del lugar de desplazamiento. Siendo la posición de estos obstáculos estática.

8.c) ST. Evitar obstáculos que no permitan circulación

«*Testing*» *Status:* Validated

Phase: 1.0

Poner el robot ante una serie de obstáculos que no permitan la circulación, y ver cómo responde.

8.1.2 Requerimientos no-funcionales

Se necesita una definición clara y comprensiva de requerimientos que cubra todo los aspectos del sistema incluyendo usabilidad, reusabilidad, mantenimiento, entre otros. Todos estos factores pueden ser clasificados dentro de un grupo de factores de calidad. Para garantizar la calidad del sistema, se definen los requerimientos no funcionales según el modelo clásico de clasificación de factores de calidad de McCall (Galin, 2004). Dichos factores se agrupan en los siguientes 3 ejes: operación, transición, y revisión del sistema o producto.

- Transición:
 - **Portabilidad:** No aplica. El sistema sólo debe correr bajo Linux.
 - **Reusabilidad:** Los módulos de localización, planeamiento de caminos y mapeo deben ser módulos o componentes que podrán ser usados sin modificaciones en otras aplicaciones o proyectos.
 - **Interoperabilidad:** Cada módulo desarrollado tanto a alto como bajo nivel debe poder ser utilizado por el sistema operativo para robots ROS a partir de alguna interfaz que éste provea. Además todos los módulos o nodos deben ser escritos en alguno de los lenguajes de programación soportados por ROS: C++, Python o LISP.
- Operación:
 - **Correctitud:** En la computadora a bordo se debe poder visualizar la trayectoria del robot para comprobar si éste avanza de acuerdo al camino dado por el planificador de movimientos. Por otro lado el mapa de la habitación obtenida debe ser mostrado en la computadora a bordo en tiempo real, con el fin de poder determinar si tiene relación alguna con la misma.
 - **Confiabilidad:** El robot debe localizarse correctamente como mínimo en un lugar de 3 metros x 3 metros con un error inferior a 30cm.
 - **Eficiencia:** El robot debe ser capaz de moverse con el suficiente nivel de baterías y otros recursos necesarios, como para poder recorrer una habitación de 10x10 metros enteras. También debe ser capaz de soportar y desplazar su peso. Se toma un excedente del 50% del peso. Además la computadora a bordo deberá contar con alguna distribución de Linux y como mínimo un procesador multitarea para poder correr un sistema operativo para robots sobre ella.
 - **Integridad:** Sólo los desarrolladores o personas que formen parte de la investigación podrán hacer cambios en parámetros tanto de la parte electrónica como de localización, control, navegación y mapeo.
 - **Usabilidad:** El sistema sólo podrá ser usado por gente allegada al producto que tenga los conocimientos o el entrenamiento suficiente como para poder usarlo. Un nuevo desarrollador que se integre al equipo necesitará un entrenamiento no inferior a 1 semana para poder instalar y configurar el software del sistema.
- Revisión:
 - **Mantenibilidad:** para asegurar un fácil mantenimiento del sistema, se debe utilizar una arquitectura basada en componentes donde cada módulo sólo contenga una funcionalidad.
 - **Flexibilidad:** El sistema debe poseer un sistema operativo que permita una fácil y rápida extensión de funcionalidad.
 - **Facilidad de prueba:** La trayectoria y los mapas deben ser impresos en pantalla permitiendo un posterior análisis y verificación.

8.2 Herramientas de control de versiones

Para el almacenamiento de los archivos (de todo tipo, código, informe, paquetes de software, etc.) se utilizó el hosting de subversión SVN, Assembla que es libre y tiene espacio ilimitado.

Por otra parte la herramienta para el control de versiones usada depende del sistema operativo. El desarrollo se hizo en Linux pero muchas veces fue necesario utilizar Windows también, entonces para cada uno se utilizó:

- Linux: RapidSVN, SVN workbench
- Windows: TortoiseSVN

8.3 Hardware y software considerado

8.3.1 Sensores

A continuación se muestran comparaciones de los diferentes tipos de sensores considerados tanto para la posición y medición de distancias, como para la orientación.

8.3.1.1 Posición y distancias

En la siguiente tabla se presenta un análisis de los principales sensores para mediciones de posición, distancia y profundidad para el robot, de acuerdo con los requerimientos funcionales FSR2 y FSR5.

Los sensores utilizados para la comparación fueron: GMP36-528-Encoder, sensor de ultrasonido Parallax PING¹⁵, sensor de infrarrojo Sharp GP2D120¹⁶ y sensor Kinect para X-Box¹⁷ (Kinect para Windows tiene menos disponibilidad y un precio más elevado de U\$S 230¹⁸).

Características	Encoder	Kinect	Ultrasonido	Infrarrojo
Imagen				
Distancia recorrida	<p>Se calcula contando pulsos. Se pierde correctitud cuando el robot resbala o es levantado y</p>	<p>Se calcula con odometría visual más SLAM (ver 11.5.1.1). También es posible medir el avance en línea</p>	<p>Mide distancias solo en línea recta haciendo diferencias. Se debe complementar con otro sensor para medir la</p>	Ídem a ultrasonido

15 (2-5-13) <http://www.parallax.com/tid/768/productid/92/default.aspx>

16 (2-5-13) <http://www.farnell.com/datasheets/1496354.pdf>

17 (2-5-13) <http://www.amazon.com/Kinect-Sensor-Adventures-Xbox-360/dp/B002BSA298>

18 (2-5-13) <http://www.amazon.com/Microsoft-L6M-00001-Kinect-for-Windows/dp/B006UIS53K>

	reubicado.	recta calculando diferencias de valores de profundidad.	orientación.	
Rango de medidas	Contando pulsos se puede medir la cantidad de metros que se necesiten.	80 cm – 4 m y puede ajustarse al rango 40 cm- 3 m en Modo cercano ¹⁹ (solo en la versión Kinect para Windows) ²⁰ .	2cm – 3m	4 cm – 30 cm
Condición de uso	Terreno no resbaladizo	Solo en interiores o lugares no expuestos al rayo del sol ya que esto impide el reconocimiento del patrón infrarrojo.	Entorno sin objetos absorbentes de sonido.	Solo en interiores o lugares no expuestos al rayo del sol ya que esto impide el reconocimiento del patrón infrarrojo.
Uso como sensor de proximidad para detección de obstáculos	No	No, mínima ya que la distancia mínima medible es 80 cm.	Si, rayo ancho.	Si, rayo fino.
Costo	Viene junto con los motores TT	U\$S 100	U\$S 29	U\$S 14

Tabla 2: Análisis y comparación de para medición de posición y distancia, y sus características

Cabe mencionar que no se considera el uso de GPS para la localización ya que el robot operará principalmente en entornos cerrados, bajo techo, donde es difícil obtener un fix de GPS por la baja penetración de las señales.

8.3.1.1.1.1 Elección del sensor

Los encoders son necesarios para calcular y controlar la trayectoria del robot. Éstos vienen integrados junto con los motores TT. La información que proveen complementada con medidas de algún sensor de profundidad es considerada la mejor opción y más adecuada para el proyecto.

Ya que el objetivo final del proyecto es la generación de mapas, lo más conveniente es elegir un sensor capaz de brindar información de color y profundidad como el Kinect. Con este es posible realizar la odometría visual y SLAM (ver 11.5.1.1) sin necesidad de otros sensores.

El problema surge cuando el robot debe ser capaz de desplazarse autónomamente evitando chocar con obstáculos ya que Kinect no puede detectar objetos muy cercanos. Para esto, es necesario

19 (2-5-13) <http://blogs.msdn.com/b/kinectforwindows/archive/2012/01/20/near-mode-what-it-is-and-isn-t.aspx>

20 (2-5-13) <http://msdn.microsoft.com/en-us/library/hh438998.aspx>

agregar un sensor infrarrojo o de ultrasonido. Ya que los sensores de infrarrojo tienen un rayo muy fino, se considera más seguro utilizar un sensor de ultrasonido para este caso. De esta forma se tienen mejores posibilidades de detectar objetos con mayor precisión y aquellos objetos que no están directamente en frente del robot.

8.3.1.2 Orientación

En la siguiente tabla se consideran tres tipos de sensores de orientación: magnetómetro HMC5883L, giróscopo MPU 6050 (GY-521), sensor Kinect.

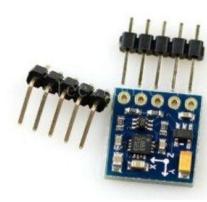
Características	Magnetómetro	Giróscopo	Kinect
Imágenes			
Funcionamiento	Basado en mediciones sobre el campo magnético terrestre.	Basado en el cálculo y la integración de la velocidad angular.	Tiene dos formas de dar información de orientación, con un giróscopo interno, y basado en técnicas de odometría visual.
Condición de uso	Zona libre de campos magnéticos que puedan interferir.	-	Solo en interiores o lugares no expuestos al rayo del sol ya que esto impide el reconocimiento del patrón infrarrojo.
Costo	U\$S 18 ²¹	U\$S 17 (incluye acelerómetro) ²²	U\$S 100

Tabla 3: Análisis y comparación de sensores de orientación y sus características.

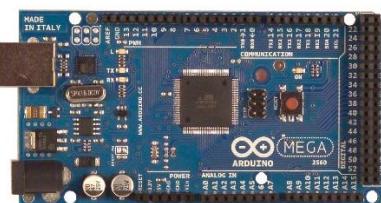
Elección de sensores

En el presente trabajo se realizaron pruebas con los tres sensores. Las realizadas con el magnetómetro pueden verse en la sección 14.2, mientras que el giróscopo se describe en la sección 11.2.3.2. Dado que con el magnetómetro se observaron demasiadas interferencias, el ítem de "Condición de uso" fue clave a la hora de elegir entre los sensores. Se terminó utilizando el giróscopo, y dado que el sensor Kinect fue elegido con anterioridad para el posicionamiento, se pueden complementar ambos. Con el sensor Kinect, hay casos en los que la odometría visual puede fallar

21 (3-5-13) [22 \(3-5-13\) <http://www.amazon.com/Arduino-GY-521-MPU-6050-gyroscope-accelerometer/dp/B008BOPN40>](http://www.amazon.com/GY-271-HMC5883L-Triple-Compass-Magnetometer/dp/B00CGY618I/ref=sr_1_11?s=electronics&ie=UTF8&qid=1367129597&sr=1-11&keywords=hmc5883</p>
</div>
<div data-bbox=)

(cuando la cámara es rotada muy rápidamente y el detector de características de imágenes no tienen tiempo para seguir el movimiento), y en tales casos se debería utilizar el giróscopo.

8.3.2 Placa de desarrollo – Microcontrolador

Características	NXP LPC1343 Board ²³	Arduino mega2560 ²⁴
Imagenes		
Procesador	ARM Cortex-M3 32-bit arquitectura RISC	AVR de Atmel de 8 bits
Lenguaje de programación	Ensamblador, C.	C, C++.
Curva de aprendizaje, tiempo de desarrollo	<p>En cada aspecto de la programación se debe tratar con rutinas de muy bajo nivel como la manipulación de bits para la activación pines de entrada/salida, interrupciones, modos de operación, etc.</p> <p>Antes de poder leer o escribir pines, se debe identificar la dirección (en hexadecimal) del registro asociado y setear sus bits con el valor binario necesario.</p> <p>Se necesitan muchas librerías para enviar un simple “Hello World” entre el chip y la computadora.</p>	<p>La programación se puede hacer a alto nivel en C++ y tiene muchas librerías muy probadas y fáciles de usar, con funciones de muy alto nivel. Basta con escribir Serial.println("Hello World") para empezar a utilizarlo.</p> <p>Cada pin simplemente se puede usar identificándolo con el mismo número que tiene en la placa.</p> <p>Leer o escribir un valor desde/hacia un pin:</p> <pre> 1. Int pin1=1,pin2=2,valor; 2. valor = analogRead(pin1); 3. analogWrite(pin2, valor); </pre>

23 (3-5-13) NXP LPC1343 Board <http://www.rds.com.ar/>

24 (3-5-13) Arduino Mega 2560 http://articulo.mercadolibre.com.ar/MLA-458198934-arduino-mega-2560-rev-3-_JM

	El uso de PWM requiere que se seteen distintos registros como contadores y es muy limitada la cantidad de moduladores independientes.	Posee 14 salidas PWM fácilmente configurables 4. <code>analogWrite(ledPin, dutyCycle);</code>
Comunidad y entorno de desarrollo, soporte	Es posible encontrar tutoriales sobre algunas funcionalidades, pero es prácticamente imposible contactarse con los desarrolladores o encargados del mantenimiento.	Comunidad muy activa y orientada al soporte mutuo y software libre. Foro oficial de soporte muy activo. Librerías listas para ser utilizadas con los sensores y shields disponibles comercialmente.
Costo	\$ 399	\$ 220

Tabla 4: Análisis y comparación de microcontroladores

Elección de la placa

Inicialmente se trabajó sobre la placa LPC1343 durante 3 semanas. Se logró establecer la comunicación serial con la computadora a bordo para el envío de comandos al robot, y luego se continuó con el desarrollo de 4 PWM que debieron ser programados a muy bajo nivel leyendo el manual de la placa y viendo que registros se debían involucrar para programar cada uno de los PWM. Se logró configurar 4 PWM, pero muchas veces se obtenían resultados inesperados en 2 de ellos, cuyos valores de frecuencia cambiaban inesperadamente. Probablemente esto esté relacionado a que la placa provee 2 contadores de 16-bits y dos de 32-bits que si bien fueron programados para funcionar a la misma frecuencia, pueden haber influido en el comportamiento. Dado que los resultados obtenidos no fueron los esperados, se decidió a cambiar la placa de desarrollo por un Arduino Mega2560. El tiempo de desarrollo que con la placa LPC1343 había llevado 3 semanas, con Arduino se pudo lograr en 2 días de trabajo. En muy poco tiempo se tuvo toda la funcionalidad necesaria implementada y probada. Es altamente recomendable utilizar esta tecnología para proyectos similares.

8.3.3 Sistema operativo y entorno de desarrollo

A continuación se muestra una comparación de los dos sistemas operativos considerados:

	Windows	Linux
Microsoft Kinect SDK	Si	No
OpenNI	Si	Si
Lenguajes de programación	El SDK oficial soporta los lenguajes C++, C#, Visual Basic. Pueden usarse otros en caso de usar OpenNI.	Mayormente C++ pero el diseño modular que se consigue utilizando ROS permite la independencia de partes posibilitando el uso de cualquier

		lenguaje.
OpenCV	Si	Si
PCL	Si	Si
ROS	Solo tiene funcionalidad básica y está muy poco desarrollado hasta el momento	Si
OpenGL	Si	Si
Arduino	Si	Si

Tabla 5: Análisis y comparación de los posibles Sistemas Operativos a elegir.

La mayor parte del desarrollo e investigación relacionado a la robótica en el mundo se está llevando a cabo con ROS bajo el sistema operativo Linux. Por ello, para el presente trabajo se eligió utilizar el sistema operativo ROS bajo la distribución de Linux Ubuntu 12.10.

8.4 Análisis de riesgos

Los riesgos del proyectos son las situaciones adversas que pueden ocurrir y amenazar el plan del proyecto, ya sea atrasándolo o aumentando los costos. En la siguiente tabla se listan los posibles riesgos junto a su probabilidad de ocurrencia y el impacto que tendrían sobre el desarrollo del proyecto.

Riesgo	Probabilidad	Impacto	Acciones
Motores, baterías y otros recursos necesarios no disponibles comercialmente en el mercado local.	alta	alto	Solo es posible comprar los que pueda encontrarse en el mercado nacional, o comprar en el exterior y esperar a que lleguen.
El microcontrolador elegido no tiene la velocidad de procesamiento necesaria para procesar los Encoders en tiempo real y efectuar la localización y el control.	baja	alto	Dos alternativas: <ul style="list-style-type: none"> - De alguna manera reducir la cantidad de pulsos por vuelta (resolución) del encoder. - Elegir el microcontrolador adecuado
No se encuentran herramientas que faciliten la visualización del mapeo	baja	alto	Deberá desarrollarse un visualizador.

Alguno de los componentes requiere una alimentación eléctrica mayor que la provista por la batería elegida.	baja	media	Dos alternativas: - Búsqueda de componentes alternativos - Adaptación del componente.
El tiempo requerido para desarrollar el sistema fue subestimado	alta	medio	Se buscará acotar el proyecto o se buscarán nuevas herramientas que permitan más productividad.
No es posible acceder a plataformas robóticas para el desarrollo	alto	alto	Se deberá construir toda la parte mecánica y electrónica.
Las computadoras utilizadas no tienen la capacidad de procesamiento necesario para la realización del mapeo y su visualización en tiempo real.	baja	alto	Se tendrá que evitar la visualización de los resultados en tiempo real.

Tabla 6: Matriz de análisis de riesgos

8.5 Especificaciones definitivas para el proyecto

8.5.1 Especificaciones de hardware

a) Base del robot y alimentación del sistema

- Plataformas de madera
- Batería Li-Po de 22.2V 3000mA/h para el robot y de 12V 1.3A/h para el sensor de mapeo

b) Sensores

- Microsoft Kinect (cámara color RGB y cámara de profundidad RGBD).
Ver Tabla 2: Análisis y comparación de para medición de posición y distancia, y sus características)
- Giróscopo
Ver Tabla 3: Análisis y comparación de sensores de orientación y sus características.
- Encoders Rotacionales (en motores TT)
Ver Tabla 2: Análisis y comparación de para medición de posición y distancia, y sus características)

c) Microcontrolador

- Arduino Mega2560
Ver Tabla 4: Análisis y comparación de microcontroladores

d) Computadora a bordo (laptop)

- Cualquier laptop que cumpla con los requerimientos mínimos para el sensor Kinect:
 - Linux (a través de los drivers de OpenNI).
 - Arquitectura de 32 bit (x86) o 64 bit (x64)
 - Dual-core 2.66-GHz o superior
 - Puertos USB 2.0 o 3.0
 - 2 GB RAM como mínimo.

Ver Tabla 5: Análisis y comparación de los posibles Sistemas Operativos a elegir.

e) Hardware y mecánica del robot

- Dos motores TT GMP36528 of 36mm dc brushless con encoder
- Dos ruedas traseras para tracción
- Una rueda delantera libre

Ver sección 0

8.5.2 Especificaciones de software

- Linux Ubuntu 12.10 Quantal
- ROS versión groovy
- Driver para dispositivos de interfaces naturales OpenNI.
- Lenguajes de programación C++ y Python.

9 Diseño de la arquitectura del sistema

En esta sección se muestra el diseño del sistema. La arquitectura del sistema se basa en la metodología basada en componentes en la cual un componente, se define como un elemento de software o hardware independiente, que puede interactuar con otros componentes mediante interfaces, para crear un sistema de completo (Sommerville, 2011).

La ventaja de este enfoque es que cada componente puede ser desarrollado independientemente de los otros, y puede ser reusado en otros sistemas. No se necesita ningún conocimiento en cuanto al código fuente para usarlo. Los servicios ofrecidos por el componente están disponibles a partir de su interfaz expresada en términos de operaciones parametrizadas.

A continuación se presenta, en primer lugar, el diseño modular de la arquitectura y, seguido de este, se muestra un diseño funcional.

9.1 Diseño modular

En la figura se muestra al más alto nivel, cada uno de los componentes del sistema y las conexiones o interfaces entre ellos. Cada módulo cuenta con una responsabilidad dada y su interfaz.

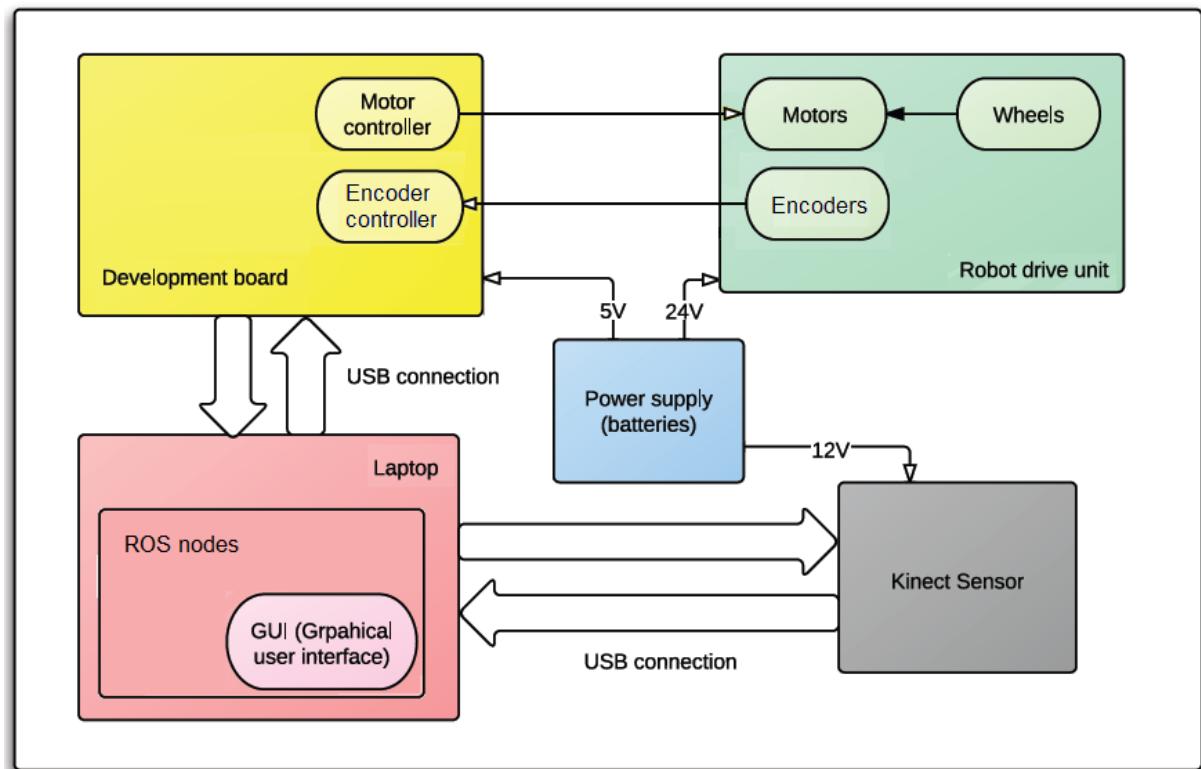


Ilustración 20: Diseño modular de la arquitectura del sistema

A partir de la imagen anterior, se pueden distinguir las responsabilidades de cada módulo:

- **Unidad de manejo del robot (robot drive unit):** este módulo es totalmente mecánico y de hardware, y contiene los motores junto con cada rueda para el movimiento de tracción del robot, una rueda adicional libre con posibilidad de giro para su estabilización, y encoders para cada motor usados para la localización. Además en este módulo pueden existir otros sensores necesarios para tomar medidas adicionales como orientación y proximidad.

- **Baterías (power supply):** todos los componentes necesitan de una fuente de energía. La unidad de mando del robot requiere el mayor voltaje, para la alimentación de los motores. La lógica del sistema necesita de una alimentación de 5V, mientras que el sensor Kinect será alimentado con una batería de 12V. En el caso de la computadora a bordo, se utilizará su batería interna.
- **Placa de desarrollo (Development board):** este módulo se compone del microcontrolador Arduino y los sensores asociados. Se conecta por hardware directamente a la unidad de manejo del robot a través de su interfaz de salida. Su responsabilidad es controlar por software dicha unidad, y realizar todo el control del robot y las tareas de más bajo nivel del sistema. Cabe mencionar que sobre este módulo se desarrolló el controlador PID del robot. Además a partir de su interfaz de entrada, tiene conexión con la computadora a bordo del robot, la cual le envía comandos de movimientos necesarios para el desplazamiento del robot.
- **Computadora a bordo (Laptop):** dentro de este módulo corre sistema operativo para robots ROS, sobre el cual se conectan internamente cada uno de los módulos de más alto nivel utilizados para la comunicación con el módulo de control del robot (microcontrolador), la planificación de caminos, y la visualización y generación de mapas a partir del procesamiento de las imágenes provistas por el módulo que contiene el sensor Kinect. Además de las responsabilidades mencionadas, dicho módulo contiene una interfaz gráfica sobre la cual se muestran los mapas generados, y permite obtener por consola todos los valores y parámetros del robot necesarios para el testing del sistema.
- **Sensor Kinect:** este módulo contiene el sensor utilizado para la generación de mapas, y su responsabilidad consiste en la captura de imágenes color y profundidad que son enviadas al módulo que contiene la laptop para su procesamiento y visualización.

9.2 Diseño funcional

Para lograr una explicación más detallada del sistema, se profundiza más reduciendo el nivel de abstracción y viendo el funcionamiento interno de cada componente descrito anteriormente.

En este caso, se muestran los diferentes nodos que componen el sistema completo. Los nodos ejecutados bajo el sistema operativo ROS, se muestran en azul.

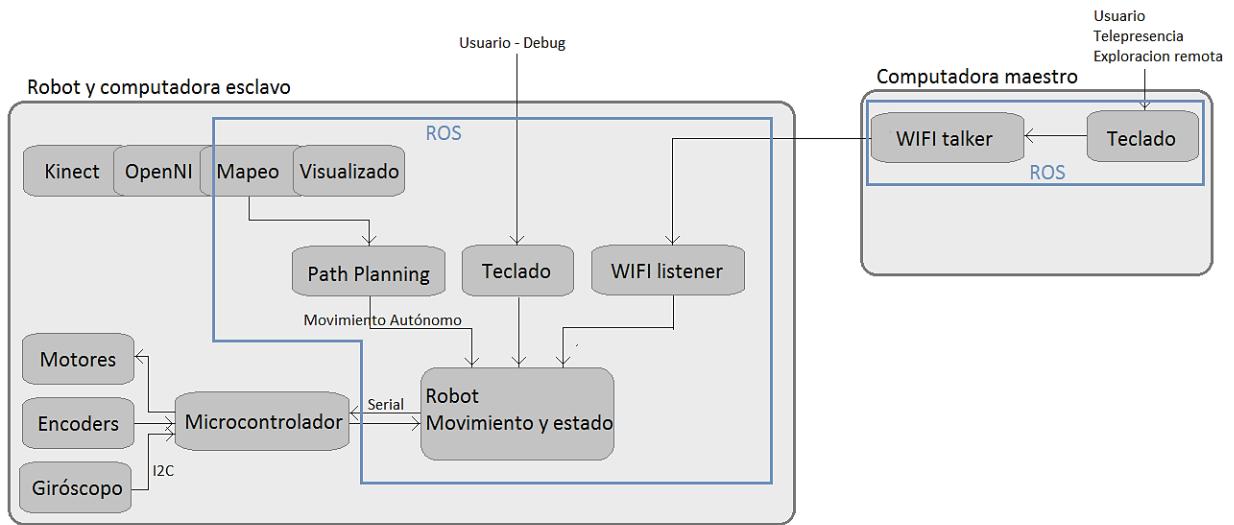


Ilustración 21: Diseño funcional de la arquitectura del sistema y sus componentes.

Se describirán uno por uno cada nodo y sus responsabilidades e interfaces. Luego, se explicarán los tres modos de uso posibles del sistema.

- **Teclado:** permite la interacción del usuario con el robot. El usuario puede escribir comandos a través de la consola o manejar el robot a partir de las flechas del teclado. Los movimientos que se puede introducir son up, down, right, left o stop y su puede especificar la distancia o ángulo deseado como parámetro. Este nodo debe ser ejecutado en la computadora a bordo. Existe otra forma de enviar comandos al robot remotamente con el modo de telepresencia, a partir del nodo WIFI Talker que será explicado luego.
- **Robot:** Representa la instancia del robot móvil en la computadora a bordo. Es un nodo de ROS. Está encargado de recibir comandos del usuario y de la comunicación entre el microcontrolador y la computadora.
- **Microcontrolador:** es la placa Arduino que se encuentra en la plataforma del robot y se comunica con el nodo de ROS Robot a partir de una interfaz USB-Serial. Dicho microcontrolador tiene acceso al hardware del robot y es el encargado de la lectura de los sensores. La información que proveen los encoders llegan al microcontrolador mediante interrupciones externas, mientras que la conexión con el giróscopo y el controlador es mediante IC2. Éste tiene una MPU (Motion processing unit) que también interrumpe al microcontrolador para enviarle la información. Además posee 4 PWM para el envío de pulsos a los motores según el comando que haya llegado desde el nodo Robot y el movimiento que se quiera realizar. Por otro lado, tanto el control del robot que se implementó con un algoritmo PID, como la localización del mismo se llevan a cabo sobre Arduino.
- **Kinect y OpenNI:** representa al sensor Kinect el cual, a través del driver de OpenNI que corre sobre el Robot Operating System, genera las imágenes de color mediante su cámara RGB, y profundidad mediante su cámara infrarroja, que luego se convertirá en la entrada del nodo de Mapeo y Visualización.
- **Mapeo y Visualización:** es un nodo de ROS que a partir de la entrada que provee el sensor Kinect, construye un mapa 3D del entorno explorado y hace posible su visualización a partir de una herramienta que provee ROS llamada Rviz. Luego hace una transformación generando

un mapa 2D que será la entrada del nodo de planeamiento de trayectorias que también corre sobre ROS.

- **Path Planning:** este nodo del Robot Operating System recibe del nodo de visualización y mapeo, el mapa 2D, el punto inicial del robot en el mapa, y el objetivo al cual quiere llegar, y se encarga de generar la mejor trayectoria hacia el objetivo dado, evitando obstáculos que se hayan visualizado en el mapa. Esta trayectoria es descompuesta en pequeños pasos o comandos que serán enviados al nodo Robot el cual tiene comunicación directa con el hardware.
- **WIFI Talker:** En el modo telepresencia, este nodo se encarga de enviar los comandos desde la computadora remota (maestro) hasta la computadora a bordo (esclavo) a través de una conexión TCP (Transfer control protocol). La entrada, es a partir de los comandos que escribe el usuario desde el teclado.
- **WIFI Listener:** Se utiliza en el modo telepresencia para recibir los comandos desde la computadora maestro. Es decir, que se conecta directamente al WIFI Talker a partir del protocolo de comunicación TCP.

Como se mencionó anteriormente, existen tres modos de uso o manejo del robot:

Modo	Movimiento	Descripción
Debug	Guiado por el usuario	Con fines de testeo, solo se utiliza una computadora, la que va a bordo del robot y el usuario introduce comandos de movimiento mediante el teclado
Mapeo y exploración	Autónomo	Una vez dada la orden de inicialización, el robot comienza a realizar un mapeo y decide hacia qué puntos desplazarse para realizar la exploración
Telepresencia	Guiado por el usuario remotamente	Se utilizan dos computadoras, una esclava (a bordo del robot) que recibe comandos inalámbricamente desde la computadora remota (maestro).

Tabla 7: Modos de operación y funcionamiento del robot.

10 Proceso de desarrollo

El desarrollo del proyecto fue llevado a cabo con un modelo de desarrollo iterativo e incremental (Turner, 1996). Dicha metodología permite satisfacer las necesidades de los clientes inmediatamente, esto genera entrenamiento del cliente y como realimentación, un constante aprendizaje en el equipo de desarrollo.

Al tener múltiples entregas o releases incrementales, se deben hacer pruebas en cada una de las etapas, y a su vez de las anteriores. Esto requiere un esfuerzo adicional, pero a cambio, se pueden detectar defectos en etapas muy tempranas del desarrollo (es importante considerar que el costo de remover un defecto en la fase de especificación de requerimientos es aproximadamente 110 veces menor que remover un defecto post-release (Galin, 2004)). Esto significa que si en cada reléase se tiene un producto con muy pocos defectos, y si se van eliminando en cada entrega, el efecto de los mismos no será acumulativo.

Este método requiere tener un buen conocimiento de la arquitectura del sistema en etapas muy tempranas del desarrollo, esto puede llegar a ser un problema si ésta no se conoce al inicio del proyecto. En el presente proyecto, el sistema operativo ROS permitió diseñar una arquitectura modular bastante clara desde el comienzo.

Además toma algunas características de otros modelos, como por ejemplo: el análisis de riesgos es basado en la metodología de desarrollo en Espiral, la construcción de prototipos se basa en el modelo de Prototipado, y el hecho de que hay que conocer la arquitectura al comienzo del desarrollo, también es necesario en el desarrollo en Cascada.

En la siguiente figura se muestra un diagrama explicativo del modelo en el cual los ciclos se repiten hasta obtener un prototipo del producto medianamente satisfactorio.

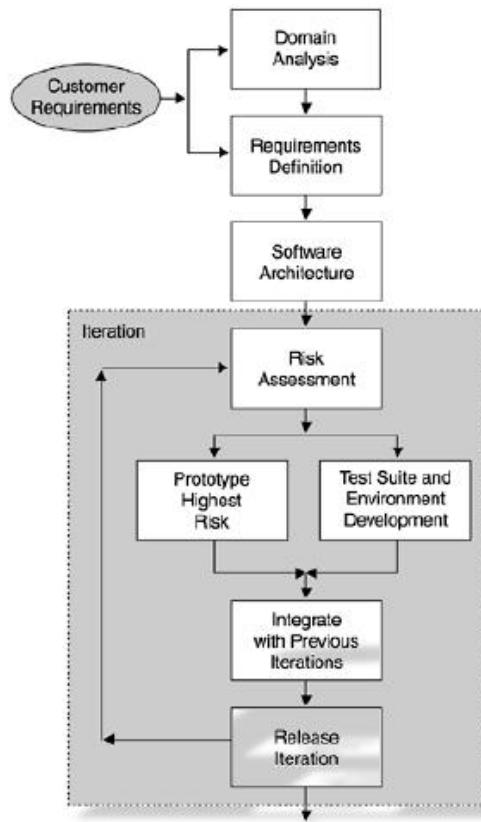


Ilustración 22: Diagrama explicativo de la metodología de desarrollo iterativa

El planeamiento de cada iteración implica describir metas, examinar riesgos, construir un prototipo o agregar funcionalidad a uno desarrollado en etapas anteriores, hacer test unitarios y luego integrarlo con las iteraciones previas antes del release. Para ello es necesario observar los resultados de iteraciones previas, y el estado de los riesgos que aplican. Para cada nueva iteración a ejecutar, el plan debe ser específico, es decir debe ser detallado, mientras que las demás iteraciones se detallan a medida que se hacen releases de las previas.

Para el planeamiento de cada iteración, se dividió el proyecto en grandes áreas y cada una de estas fue dividida en pequeños incrementos. Cabe mencionar, que cada incremento es la diferencia entre el reléase de una iteración y el de la próxima. Una de las grandes ventajas de esta metodología es que cada área puede ser desarrollada independientemente de las demás y solo será necesario unirlas al finalizar.

A continuación se describe cada iteración y sus incrementos:

ITERACIONES					
	Construcción del prototipo del Robot	Control del robot	Instalación de sistema operativo ROS, herramientas y librerías	Localización	Mapeo y visualización
INCREMENTOS	Obtención de los insumos necesarios: plataforma, ruedas, ejes para motores, motores, pintura, herramientas, etc.	Análisis y obtención del microcontrolador adecuado	Aprendizaje sobre conceptos básicos del sistema operativo a partir de tutoriales básicos	Tomar lecturas de los encoders con el microcontrolador mediante interrupciones externas	Investigación sobre la odometría visual y SLAM. Instalación de drivers, librerías, y herramientas de ROS.
	Responde a requerimientos: -	Responde a requerimientos: -	Responde a requerimientos: FSR4	Responde a requerimientos: FSR2	Responde a requerimientos: FSR6, FSR7
	Corte y pintura de la plataforma de madera	Establecimiento de una conexión USB-serial con la computadora y envío de palabras.	Creación de un workspace de trabajo e instalación de paquetes básicos	Cálculo de posición del robot a partir de los pulsos de encoder	Implementación o instalación de otros modulos existentes de ROS.
	Responde a requerimientos: -	Responde a requerimientos: FSR1	Responde a requerimientos: FSR4	Responde a requerimientos: FSR2	Responde a requerimientos: FSR5, FSR6, FSR7
	Colocación de los	Confección de	Creación de nodos	Estudio, análisis y	Elección de un
					Adaptación del

	<i>motores en las ruedas e integración en la plataforma</i>	<i>comandos a recibir y parseo</i>	<i>básicos de pasos de mensajes entre módulos. Envío de mensajes simples.</i>	<i>pruebas de sensores de orientación</i>	<i>módulo de mapeo definitivo. Mediciones y pruebas sobre el modulo.</i>	<i>algoritmo a mapas dados por el módulo de visualización y mapeo</i>
	<i>Responde a requerimientos: -</i>	<i>Responde a requerimientos: FSR1</i>	<i>Responde a requerimientos: FSR4</i>	<i>Responde a requerimientos: FSR2</i>	<i>Responde a requerimientos: FSR5, FSR7</i>	<i>Responde a requerimientos: FSR8</i>
6	<i>Primer prototipo del circuito electrónico para control de motores</i>	<i>Control de los movimientos del robot a partir de los comandos recibidos</i>	<i>Estudio sobre makefiles, creación de paquetes y nodos propios y comandos básicos de compilación</i>	<i>Obtención de mediciones del sensor de orientación</i>	<i>Proyección a 2D a partir de los mapas 3D.</i>	<i>Traducción de la trayectoria a comandos entendibles por el robot</i>
	<i>Responde a requerimientos: FSR1</i>	<i>Responde a requerimientos: FSR1</i>	<i>Responde a requerimientos: FSR4</i>	<i>Responde a requerimientos: FSR2</i>	<i>Responde a requerimientos: FSR8</i>	<i>Responde a requerimientos: FSR8</i>
	<i>Se analiza la salida de los encoders con instrumentos de laboratorio y se agrega la conexión de los mismos a la placa para su posterior lectura.</i>	<i>Diseño del control del robot (controlador PID) para ajuste de velocidad y movimientos en línea recta y de giro correctos</i>	<i>Interacción entre diferentes módulos programados sobre lenguajes diferentes (se probó interacción entre módulos desarrollados en Python y C++)</i>	<i>Cálculo de posición completa del robot (x, y, ϕ)</i>	<i>Identificación de objetivos a explorar.</i>	<i>Envío secuencial de los pasos necesarios hasta el objetivo.</i>
	<i>Responde a requerimientos: FSR2</i>	<i>Responde a requerimientos: FSR3</i>	<i>Responde a requerimientos: FSR4</i>	<i>Responde a requerimientos: FSR2</i>	<i>Responde a requerimientos: FSR8</i>	<i>Responde a requerimientos: FSR8</i>
	<i>Se construye el</i>	<i>Corrección de</i>	<i>Instalación de</i>	<i>Métodos de</i>	<i>Creación de</i>	<i>Creación de un</i>

	<p><i>prototipo final de la placa, haciendo el diseño sobre un circuito impreso</i></p>	<p><i>velocidad de los motores (rpm) y alineación del robot</i></p>	<p><i>herramientas varias (de visualización de nodos y topics del sistema, interfaces gráficas donde podrán ser mostrados los mapas, etc)</i></p>	<p><i>comprobación de posición del robot</i></p>	<p><i>mensajes para enviar al módulo planificador de caminos: el mapa 2D, punto inicial del robot, y objetivos a explorar.</i></p>	<p><i>módulo de ROS que se comunique con el módulo de ROS que tiene acceso al microcontrolador para enviar comandos</i></p>
	<p><i>Responde a requerimientos: FSR1, FSR2</i></p>	<p><i>Responde a requerimientos: FSR3</i></p>	<p><i>Responde a requerimientos: FSR4, FSR6</i></p>	<p><i>Responde a requerimientos: FSR2</i></p>	<p><i>Responde a requerimientos: FSR8</i></p>	<p><i>Responde a requerimientos: FSR8</i></p>
	<p><i>Cálculo de parámetros del controlador manual o automática (mediante algún algoritmo de configuración)</i></p>	<p><i>Creación del primero paquete con nodo de envío de comandos al módulo controlador del robot.</i></p>	<p><i>Nueva funcionalidad a comandos del robot especificando como parámetro la distancia u orientación máxima a avanzar</i></p>			
		<p><i>Responde a requerimientos: FSR3</i></p>	<p><i>Responde a requerimientos: FSR1, FSR4</i></p>	<p><i>Responde a requerimientos: FSR2</i></p>		<p><i>Estudio de alternativas disponibles en ROS para la comunicación entre computadora a bordo (maestro) a la computadora remota (esclavo). Este fue</i></p>

	<p><i>diseñado para soportar el procesamiento distribuido.</i></p>
	<p><i>Responde a requerimientos: FSR4</i></p>

Tabla 8: Muestra cada iteración junto con sus incrementos, y a que requerimiento hace referencia.

La estrategia de pruebas utilizada es incremental, y se hace dentro del ciclo de vida de desarrollo del proceso, dado que permite remover defectos en etapas tempranas tal como se mencionó anteriormente. Este tipo de test requiere pruebas unitarias, test de integración de módulos, y test del sistema completo.

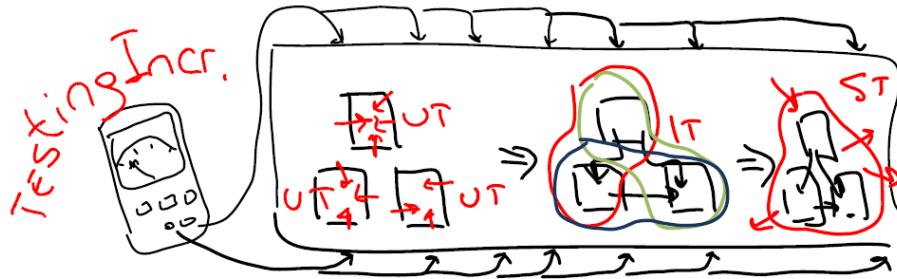


Ilustración 23: Muestra el proceso de testing que se realiza para el modelo de desarrollo incremental²⁵.

En cada incremento se fueron haciendo test unitarios, de integración y luego del sistema completo, cuyos resultados se muestran más adelante en el presente informe, en los desarrollos de cada capítulo.

A continuación se muestra una tabla con cada uno de los test unitarios mapeados con cada requerimiento funcional:

²⁵ Imagen extraída de Introducción a las pruebas del software estrategias y tipo de pruebas, Martin Miceli, 2011

#	Requerimiento funcional	Requerimiento de testing	Descripción del test	Módulos y componentes asociados
1	FSR1 Movimientos del robot a partir de comandos básicos	ST. Prueba de cada comando por separado	Probar para cada uno de los comandos, que el robot se desplace como es de esperarse. Se deben hacer varios casos de testing que prueben que cada uno de los comandos responde bien por separado y que en la combinación de éstos no interfieran entre sí. Se envían los comandos up, down, left y right y se comprueba que el robot realice el tipo de movimiento esperado.	-Unidad de manejo del robot <ul style="list-style-type: none"> • Motores -Microcontrolador (Arduino) <ul style="list-style-type: none"> • Motores • Comandos -Computadora a bordo <ul style="list-style-type: none"> • Nodo Robot (ROS)
2	FSR2 Localización del robot	ST. Comprobar posición del robot para movimientos lineales	Situar el robot en una posición aleatoria dentro de la habitación y probar los comandos up() y down() para diferentes distancias de avance y retroceso. Luego comprobar que las medidas tomadas por el mismo sean correctas en relación a la distancia recorrida en la realidad. Se envían los comandos up y down junto a una distancia. Se comprueba que se realice un avance o retroceso de la cantidad indicada.	-Unidad de manejo del robot <ul style="list-style-type: none"> • Motores • Encoders -Microcontrolador (Arduino) <ul style="list-style-type: none"> • Motores • Encoders • Comandos • Localización -Computadora a bordo <ul style="list-style-type: none"> • Nodo Robot (ROS)
		ST. Comprobar posición del robot para movimientos angulares	Probar los comandos right() y left() para diferentes ángulos y medir el desplazamiento respecto del punto inicial. Se comprueba que el robot realice el giro indicado en cuanto a sentido de giro y ángulo girado.	
3	FSR3 Control de trayectoria y velocidad	ST. Ajuste de velocidad del robot para todos los movimientos posibles	Se debe probar la reacción del robot algunas velocidades incluyendo los casos extremos, y a su vez esto para cada uno de los comandos.	-Unidad de manejo del robot <ul style="list-style-type: none"> • Motores • Encoders

			<p>Se envían todos los comandos de movimiento, uno a la vez. En el caso de up/down, se espera que se realice un avance o retroceso sin desvíos, dadas las condiciones ideales. En cuanto a left/right, se espera que el robot gire sobre su propio eje, situado en el centro de la recta que une las dos ruedas, sin desplazamientos.</p>	<ul style="list-style-type: none"> • Giróscopo -Microcontrolador (Arduino) <ul style="list-style-type: none"> • Motores • Encoders • Comandos • Localización • Giróscopo • Control -Computadora a bordo <ul style="list-style-type: none"> • Nodo Robot (ROS)
4	FSR4 Integrar la computadora a bordo en el robot	ST. Comprobar interface microcontrolador-computadora y comunicación	<p>Probar el correcto funcionamiento de la computadora, y sus interfaces con otros dispositivos.</p> <p>Utilizando la comunicación serial, se envían comandos desde la computadora hacia el controlador en Arduino. Este último debe ser capaz de interpretar los comandos y ejecutarlos.</p>	<ul style="list-style-type: none"> - Unidad de manejo del robot <ul style="list-style-type: none"> • Motores • Encoders • Giróscopo -Microcontrolador (Arduino) <ul style="list-style-type: none"> • Motores • Encoders • Comandos • Localización • Giróscopo • Control -Computadora a bordo <ul style="list-style-type: none"> • Nodo Robot (ROS) • Nodo Teclado (ROS) • Nodo WIFI Talker (ROS) • Nodo WIFI Listener (ROS)
5	FSR5 Medición de distancias con el sensor	ST. Comprobar exactitud de mediciones a puntos conocidos	<p>Situar el robot en varios puntos conocidos, y tomar mediciones a diferentes puntos del mapa, para luego comprobar la exactitud de las mediciones, en contraste con la realidad.</p> <p>Se utilizan las nubes de puntos (<code>pcl::PointCloud<pcl::PointXYZRGB></code>) generadas a partir de los drivers de Kinect</p>	<ul style="list-style-type: none"> -Sensor Kinect <ul style="list-style-type: none"> • Open NI -Computadora a bordo <ul style="list-style-type: none"> • Nodo Visualización y Mapeo (ROS)

			para medir la distancia euclíadiana entre dos puntos en el espacio tridimensional. Esto puede realizarse manualmente, a través de Rviz.	
6	FSR6 Interfaz gráfica para mostrar mapas	ST. Probar la interfaz con mapas de prueba	Comprobar que ocurre la generación de mapas a medida que el robot es capaz de ver nuevos sectores. Se subscribe al visualizador a la salida del nodo de mapeo. Rviz debe mostrar los marcos claves provenientes del nodo de mapeo. Además, se debe mostrar las coordenadas del robot, provistas por la odometría.	-Sensor Kinect <ul style="list-style-type: none">• Open NI -Computadora a bordo <ul style="list-style-type: none">• Nodo Visualización y Mapeo (ROS)• Visualizador Rviz
7	FSR7 Generar mapas 3D	ST. Verificar coincidencia entre mapas y la habitación dinámicamente	Comprobar a través de la interfaz gráfica que los mapas generados se asemejan a la realidad, con el robot en movimiento. Se posiciona el sensor Kinect en el centro de una habitación y, lentamente, se lo gira 360 grados. Se comprueba que se haya generado un mapa o modelo representativo del entorno.	-Sensor Kinect <ul style="list-style-type: none">• Open NI -Computadora a bordo <ul style="list-style-type: none">• Nodo Visualización y Mapeo (ROS)• Visualizador Rviz
		ST. Probar diferentes escenarios	Repetir el proceso para escenarios diferentes. Al igual que en el caso anterior, pero se realiza la prueba en distintos escenarios.	
8	FSR8 Navegación. Movimientos autónomos a partir de mapas	ST. Comprobar exploración de la habitación	Comprobar que el cuarto haya sido explorado. El nodo de navegación espera el punto inicial, las coordenadas de los puntos a visitar y, además, recibe una matriz con el mapa 2D. El robot debe desplazarse hasta el punto objetivo.	- Unidad de manejo del robot <ul style="list-style-type: none">• Motores• Encoders• Giróscopo -Microcontrolador (Arduino) <ul style="list-style-type: none">• Motores• Encoders• Comandos• Localización• Giróscopo• Control
		ST. Evitar obstáculos que permitan circulación	El robot debe ser capaz de evitar obstáculos dentro del lugar de desplazamiento. Siendo la posición de estos obstáculos estática. Igual a la prueba anterior, pero en este caso se colocan obstáculos entre el robot y el	

		<p>punto objetivo. Estos serán visibles en la matriz pasada por el nodo de mapeo. La disposición de los obstáculos es elegida de forma tal que sea posible la navegación hasta el objetivo.</p>	<ul style="list-style-type: none"> -Computadora a bordo <ul style="list-style-type: none"> • Nodo Robot (ROS) • Nodo Planner (ROS) -Sensor Kinect <ul style="list-style-type: none"> • Open NI -Computadora a bordo <ul style="list-style-type: none"> • Nodo Robot (ROS) • Nodo Teclado (ROS) • Nodo WIFI Talker (ROS) • Nodo WIFI Listener (ROS) • Nodo Planner (ROS) • Nodo Visualización y Mapeo (ROS) • Visualizador Rviz
	ST. Evitar obstáculos que no permitan circulación	<p>Poner el robot ante una serie de obstáculos que no permitan la circulación, y ver cómo responde. En este caso, los obstáculos impiden la circulación y el robot debe optar por no desplazarse, evitando colisiones.</p>	

Tabla 9: Tabla de resumen de casos de testing mapeados con cada requerimiento y los módulos de la arquitectura que involucra.

Robot móvil autónomo capaz de producir mapas 3D del interior de una habitación

11 Desarrollo

11.1 Diseño del robot

Teniendo en cuenta los distintos tipos de robots descriptos en 7.1.1 y los objetivos planteados para el presente trabajo en 5.1, a continuación se justifica cada decisión tomada en relación a varios aspectos de su estructura y composición:

Movimientos que va a ser capaz de realizar, holonómico o no.

Tal como se describió, el robot debe ser capaz seguir una trayectoria dada desde un punto inicial hasta un objetivo indicado por el módulo planificador. Para lograr un cambio de dirección basta con que el robot se detenga, gire el ángulo dado, y continúe con su trayectoria. Es decir, que es suficiente con que el robot pueda desplazarse hacia adelante, atrás y rotar en el lugar para cambiar de dirección.

Dentro de la clasificación de robots se encontraban holonómicos y no-holonómicos. Un robot holonómico no se lo considera un modelo simple, dado que requiere ruedas especiales y varios motores que permitan controlar todos los grados de libertad del robot. Dado que el principal objetivo de dicho trabajo no es la construcción del robot, se opta por una solución más sencilla. Por encima de eso, hay que sumar la falta de disponibilidad comercial de dichos elementos que excede las posibilidades del grupo de trabajo.

Por otro lado, dentro de la clasificación de robots no-holonómicos, queda descartado el modelo de Ackerman ya que en cada rotación el robot necesita desplazarse, requiriendo mucho espacio para realizar cambios de dirección. Este modelo es útil en situaciones como las carreteras, en las que gran parte del desplazamiento es en línea recta y los cambios son graduales.

Por simplicidad y conveniencia, se decidió utilizar un modelo diferencial. Para ello se utilizarán dos motores independientes para cada rueda trasera de tracción, y adelante como apoyo, se utilizará una rueda libre giratoria. Lo óptimo sería utilizar una rueda esférica que le dé una mayor libertad de giro y un menor rozamiento, pero dados los tiempos y circunstancias del mercado en la actualidad, es probable que esto quede planteado como una futura mejora para trabajos futuros.

En resumen, se construirá un robot móvil, no holonómico de cinemática diferencial.

Velocidad requerida para el robot

Para este primer prototipo del robot, no está contemplado dentro de los requerimientos un límite mínimo ni máximo de velocidad. Simplemente se utilizará una velocidad (que será ajustable y controlada) que permita realizar la construcción de mapas sin inconvenientes, tal como se plantea en los objetivos.

Carga o peso que debe soportar y trasladar

En este aspecto se deben tener en cuenta todos los elementos que irán encima de la plataforma del robot, para calcular el torque de los motores de tal forma que soporten el peso.

Laptop	2.5kg aprox.
Kinect	0.2kg ²⁶
Motores TT	0.8kg
Batería Li-Po	0.487kg ²⁷
Batería Gel	< 0.8kg
Microcontrolador Arduino	0.040g ²⁸
Circuito de potencia	< 0.2kg
Plataforma del robot	< 3kg
PESO TOTAL ROBOT	< 8kg

Tabla 10: Peso del robot

Considerando que el Grupo de Robótica y Sistemas Integrados cuenta con motores que han sido importados desde China para otros proyectos, y que se adaptan a esta carga ya que su torque es de 60kg.cm, se decidió utilizar dichos motores TT cuyo modelo es GMP 36-528 (36 mm de diámetro). Además, éstos vienen equipados con Encoders que son de utilidad para la odometría del robot, lo cual es una ventaja.

Diseño y disposición de elementos sobre el robot

Con el objetivo de agrupar elementos relacionados, se optó por diseñar un robot de dos “pisos” en el cual los elementos relacionados a la electrónica (motores, circuito de potencia, baterías, microcontrolador y sensores de odometría) fueron ubicados en la parte inferior. Mientras que la computadora a bordo y el sensor Kinect fueron ubicados en el superior. Esto diseño permite además, darle un mejor rango de visión a la cámara ya que se encuentra más elevada en el piso superior.

Fuente de energía

El robot cuenta con 3 fuentes de energía. Una batería Li-Po de 22.2V y 3A/h para alimentar los motores y el circuito de potencia, que se describe en 11.2.5. Una batería de Gel de 12V y 1.3A/h para alimentar el sensor Kinect. Y por último la batería de la computadora a bordo será alimentada con su propia batería y a su vez ésta alimenta al microcontrolador Arduino.

Computadora a bordo	Batería propia	-
Sensor Kinect	12V	1.1A
Motores TT	22.2V	3A/h
Sensores de odometria	5V	-

Tabla 11: Alimentación de cada componente

26 (29-4-13) <http://www.amazon.co.uk/Kinect-Sensor-Adventures-Xbox-360/dp/B0036DDW2G>

27 (29-4-13) http://www.hobbyking.com/hobbyking/store/_9262_turnigy_3000mah_6s_20c_lipo_pack_great_for_t_rex_500_.html

28 (29-4-13) <http://www.hwkitchen.com/products/arduino-mega-2560/>

Terreno esperado

El desarrollo del primer prototipo se espera que circule con normalidad en superficies regulares sin demasiadas cavidades que provoquen el deslizamiento del robot. Para evitar esto, se utilizaron ruedas de 10cm de diámetro con superficie de goma que permiten un mejor aferre al suelo.



Ilustración 24: Ruedas utilizadas para el robot.

Robot móvil autónomo capaz de producir mapas 3D del interior de una habitación

11.2 Construcción del robot

11.2.1 Detalles constructivos

Las plataformas o bases que forman el robot fueron construidas con madera MDF dado que es un material liviano, fueron pulidas, pintadas, protegidas. Para lograr el corte circular se utilizó una herramienta de corte de espejos solares circulares. Además se debió perforar cada lugar por donde deben pasar cables provenientes de los motores y encoders hacia el microcontrolador. La pintura se realizó con aerosol plateado y luego fue recubierto con un protector adecuado para dicha pintura.



Ilustración 25: Proceso de pintado de las plataformas de madera. Se pueden apreciar los agujeros a través de los cuales son pasados los cables que conectan los motores(debajo de la base) con el microcontrolador (arriba de la base).

Por otro lado, para mantener ambos motores alineados, se utilizó un tubo plástico del mismo diámetro de los motores (36 mm), que también fue pintado y se muestra a continuación:



Ilustración 26: La pintura también se aplicó al tubo de plástico encargado de garantizar la alineación de los motores.

Una vez listos, se instalaron los motores junto con cada rueda a la base del robot, según se muestra en la figura que sigue. Los motores se fijaron al eje (tubo plástico) y éste a la base para prevenir desplazamientos y rotaciones no deseadas que podrían desviar al robot. Hay 4 tornillos en las esquinas que fueron colocados a 90 grados uno del otro con el fin de eliminar el movimiento sobre ese plano. El tornillo central atraviesa el tubo de plástico y tiene como objetivo impedir el desplazamiento del tubo respecto a la base.

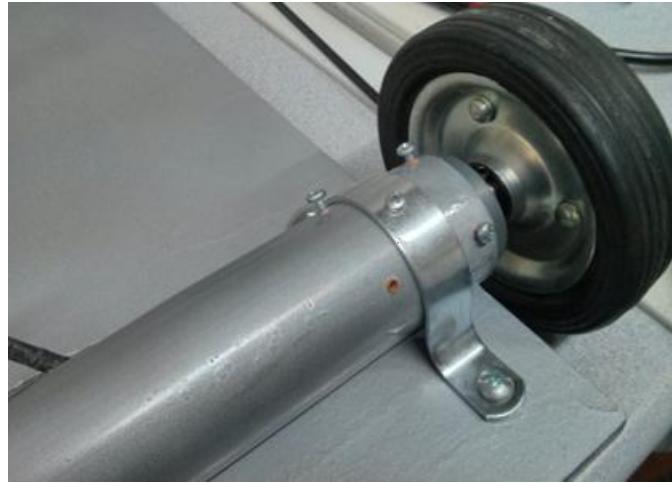


Ilustración 27: Mecanismo de ajuste de los motores para evitar desplazamientos y rotaciones no deseadas durante el movimiento.

Además se agrega en la parte frontal, una rueda giratoria suelta que estabiliza el robot y permite su rotación libre.

11.2.2 Motores y encoders

Como se vio en las fotos anteriores, el robot cuenta con 3 ruedas, y dos de ellas son manejadas con motores. Estas dos son las responsables del desplazamiento y la rotación del robot, mientras que la rueda delantera solo tiene función de soporte.

Cada una de las ruedas de tracción son independientes una de la otra, ya que tienen su motor propio. Esto es lo que permite no solo desplazamientos hacia adelante y atrás sino también rotaciones en sentido horario y anti horario. Por otro lado, esto trae desventajas ya que las ruedas pueden girar a distintas velocidades y producir desvíos durante el desplazamiento del robot en línea recta. En 11.3 se muestra un análisis realizado y el desarrollo de una unidad de control utilizada para garantizar el correcto movimiento del robot.

El modelo de los motores utilizados para las ruedas de tracción es MotorTT GMP 36-528-E.

11.2.2.1 Motores TT

Un motor TT es un motor de corriente continua con engranajes planetarios. Un sistema de engranajes planetarios permite una mayor reducción (menor velocidad de salida y más torque) ocupando menos espacio. Para lograr esto, los engranajes son colocados uno dentro de otro según se muestra en la siguiente figura:

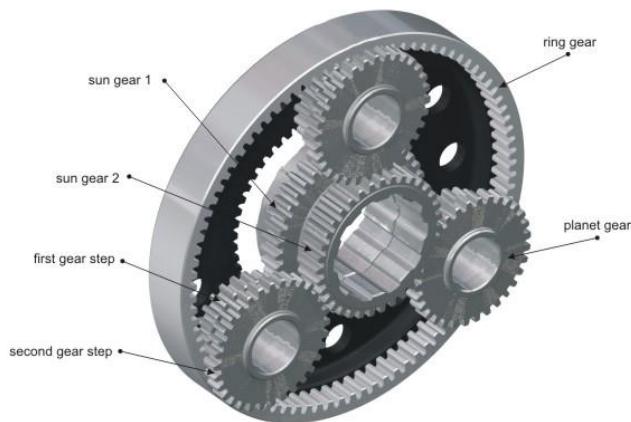


Ilustración 28: Motor planetario. Sistema de engranajes²⁹

La velocidad y el sentido de rotación del motor están determinados por la diferencia de tensión entre dos cables (rojo y negro). Es decir, los voltajes de entrada pueden ir desde 4V a 28V, y a más voltaje, mayor velocidad. Además, al invertir los cables, se invierte el sentido de giro.

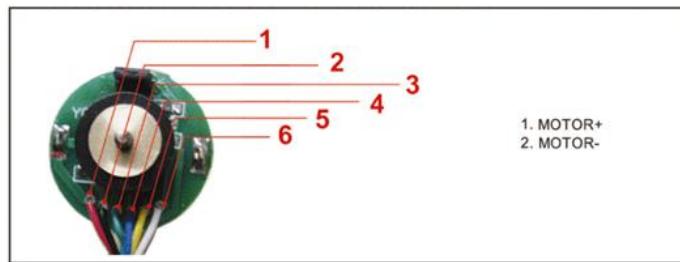
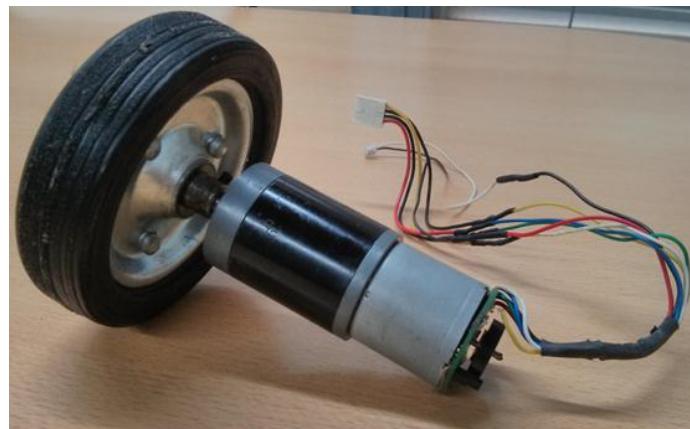


Ilustración 29: Se muestra el motor TT junto a la rueda, y en la parte inferior los cables 1 y 2 que son la alimentación del motor.

29 (25-4-13) Imagen extraída de http://www.rohloff.de/en/technology/speedhub/planetary_gear_system/

Especificaciones del motor

- Voltaje de entrada 4V a 28V
- Torque de salida 60 kg.cm
- Diámetro: 36 mm.
- Cuenta con un encoder rotacional.
- Sistema planetario de engranajes de 69.3 mm.
- Peso 400-550 g.

Cada motor tiene 2 cables de salida para la alimentación del motor, Vcc y GND.

Modelo	Voltaje		Sin carga		Con carga				Parado (stall)		
	Rango de operación	Nominal	Velocidad	Corriente	Velocidad	Corriente	Torque	Salida	Torque	Corriente	
		[V]	[rpm]	[A]	[rpm]	[A]	[N.m]	[Kg.cm]	[w]	[N.m]	[Kg.cm]
GMP36-528-	15330-264	4.0-28.0	24.0	20.0	0.05	18	0.15	0.78	8.00	1.50	5.88
Encoder	12560-369	4.0-17.0	12.0	100	0.09	82	0.40	0.29	3.00	2.50	1.76
										60.00	0.80
										18.00	1.90

Tabla 12: Se muestran las especificaciones del motor. La fila señalada en amarillo muestra el rango de voltajes de operación utilizada por el robot (4-28V) (Motor gmp36 36mm especificaciones).

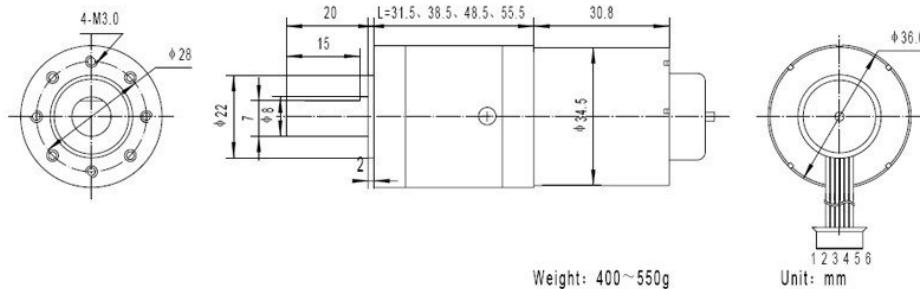


Ilustración 30: Especificaciones y dimensiones del motor TT

Se realizaron pruebas ambos motores a diferentes tensiones, primero a 11.1V y luego a 22.2V, en vacío (sin carga) y luego parándolos (en stall) y la corriente que demandaban es:

Motor derecho	
Sin carga	
Tensión	Corriente que demanda
11.1V	0.05 A
22.2V	0.05 A
Stall	
Tensión	Corriente que demanda
11.1V	0.23 A
22.2V	0.38 A



Ilustración 31: Mediciones de corriente motor derecho

Motor izquierdo	
Sin carga	
Tensión	Corriente que demanda
11.1V	0.04 A
22.2V	0.06 A
Stall	
Tensión	Corriente que demanda
11.1V	0.22 A
22.2V	0.36 A



Ilustración 32: Mediciones de corriente motor izquierdo

Comparando las mediciones realizadas, con la tabla de especificaciones de los motores, se puede ver que los valores de corriente obtenidos para el rango correspondiente en la medida tomada sin carga son muy similares, mientras que para el caso de stall la corriente obtenida es menor a la especificada en la tabla lo cual también es correcto.

11.2.2.2 Encoder

El motor TT utilizado trae incorporado un encoder rotacional cuyo funcionamiento se basa en el Efecto Hall. Sobre el eje tiene colocado un disco que cuenta con polos magnéticos positivos y negativos consecutivamente. Y, paralelo al eje de rotación, se ve un sensor de efecto hall que es capaz de detectar los cambios en los polos magnéticos y genera pulsos. De esta forma, a medida que el motor gira, se generan pulsos indicando la rotación de la rueda.

Como se ve en la figura, cada motor tiene 6 cables de salida, 4 de ellos constituyen las salidas del encoder que son: Vcc, GND, canal1, canal2. Ambos canales entregan la misma señal que son los pulsos de salida. Por otro lado la alimentación de los encoders es de 5V.

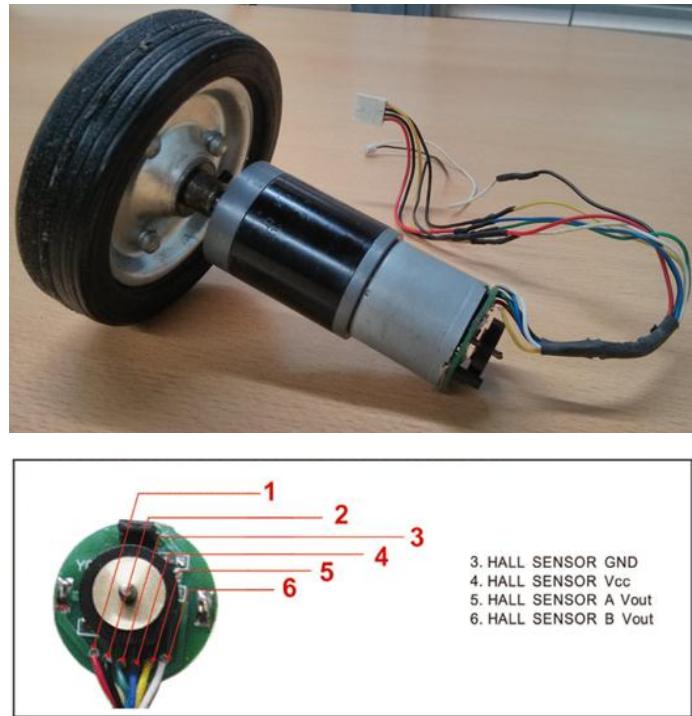


Ilustración 33: Se muestran las conexiones del encoder.

En el robot, los canales de salida de cada encoder se conectan al microcontrolador Arduino el cual lee los pulsos a través de interrupciones externas. Dicho funcionamiento se detalla más adelante.

Conociendo el diámetro de la rueda y la cantidad de pulsos que se generan por revolución, es posible calcular la distancia recorrida por cada pulso de encoder. La cantidad de pulsos por revolución que cada encoder entrega se muestra en la sección 11.3.5.5.

Este valor es usado por el microprocesador para la localización del robot, ya que al contar los pulsos del encoder, puede estimar la distancia avanzada. Utilizando esta información en conjunto con el giroscopio, es posible estimar los valores (x, y, heading) del robot.

11.2.3 Sensores de orientación

A continuación se describen dos sensores utilizados en el proyecto para medir la orientación del robot.

11.2.3.1 Magnetómetro HMC5883L

Si el robot contara con un magnetómetro a bordo (ver Anexo B: Principios de funcionamiento del Magnetómetro), sería capaz de medir las componentes vectoriales del campo magnético terrestre y a través de esto estimar su orientación, o ángulo respecto al norte terrestre (Tutorial magnetómetro HMC5883L).

```
5. float orientacion = atan2(medicion.EjeY, medicion.EjeX);
```

El módulo para Arduino HMC5883L llamado compás digital, cuenta con un magnetómetro que nos permite obtener dicha orientación. Para una mejor estimación, es necesario tener en cuenta la declinación magnética en la zona geográfica donde se desarrolla el proyecto, en este caso Córdoba, Argentina:

- Declinación magnética: -4° 42' WEST
- En radianes: -57.6 mrad (mili radianes)
- En grados: -3.3°



Ilustración 34: Declinación magnética en Córdoba, Argentina ³⁰

El valor obtenido es utilizado para corregir la orientación calculada previamente. Ya que la declinación es WEST, se la debe restar (giro en sentido anti horario) de la orientación. Visualmente:

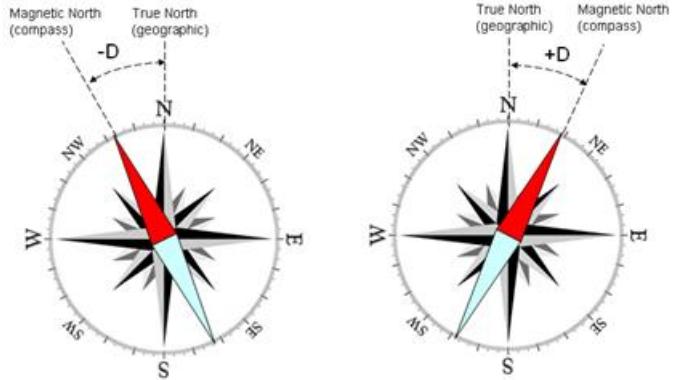


Ilustración 35: Sentido de la declinación magnética (Magnetic declination, Sentido de declinación magnética)

Antes de proceder a colocar el sensor en el robot, se realizaron varias pruebas para comprobar el correcto funcionamiento del sensor. Concretamente, el sensor es el HMC5883L producido por Honeywell y fue elegido dada su masividad y la gran cantidad de tutoriales y proyectos de código abierto que lo

30 (15-4-13) Datos extraídos de <http://magnetic-declination.com/>

utilizan (Tutorial magnetómetro HMC5883L). El integrado cuenta con un conversor ADC de 12 bits que permite una resolución de 1 grado. Sus dimensiones son 3.0x3.0x0.9mm, y se comunica con Arduino a través del protocolo de comunicación I2C (Datasheet HMC5883L).



Ilustración 36: Chip integrado del magnetómetro HMC5883L

Las pruebas que se realizaron consistieron en colocar el sensor inicialmente apuntando al norte geográfico y, a medida que se lo fue girando, se fueron tomando nota de los valores medidos en los ángulos correspondientes. Además se verificaron los casos extremos, es decir, luego de una vuelta completa la lectura debería pasar de 360° a 0° volviendo al ángulo inicial.

Para facilitar la rotación se colocó el sensor en la punta de una vara plástica. Esta vara es rotada y, con ella, el sensor toma las mediciones.

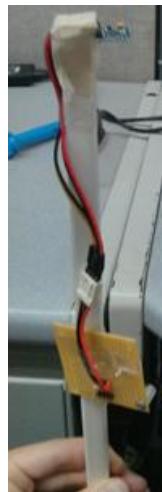


Ilustración 37: Herramienta diseñada para la medición de orientaciones con el magnetómetro

Se tomaron dos mediciones: una en el Laboratorio de Arquitectura de Computadoras de la Facultad, y otra fuera de ésta (a varias cuadras de distancia), para luego hacer una comparación

En las siguientes imágenes se muestran los resultados. En la primera, las mediciones realizadas fuera de ciudad universitaria produjeron los resultados esperados. Por el contrario, el otro grupo de mediciones, representado en la segunda imagen, muestra valores que indicarían la presencia de un fuerte campo magnético en los alrededores de la Facultad. Dicho campo distorsiona completamente los valores obtenidos y hace prácticamente imposible el uso de este sensor para obtener una orientación correcta.

Cabe destacar que se probaron dos magnetómetros distintos para eliminar la posibilidad de que el sensor sea el problema.

Los valores internos al círculo son los valores correctos o esperados y los externos son los medidos. Tener en cuenta que la declinación magnética en Córdoba es -3.3° y no está contemplada en la figura.

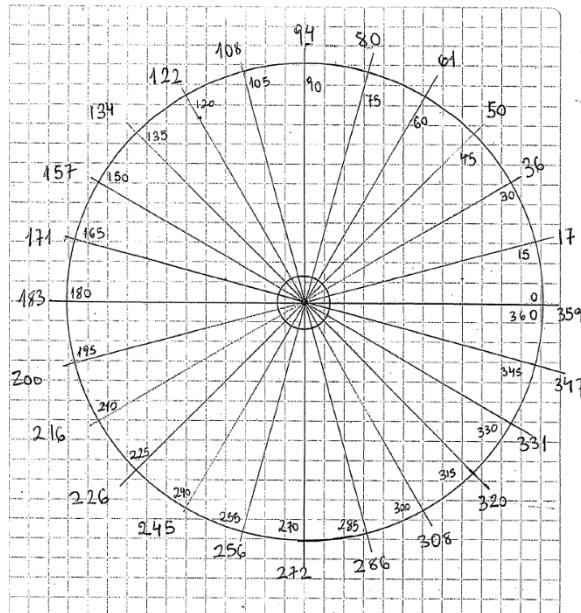


Ilustración 38: Mediciones con el magnetómetro tomadas lejos de la facultad FCEFyN.

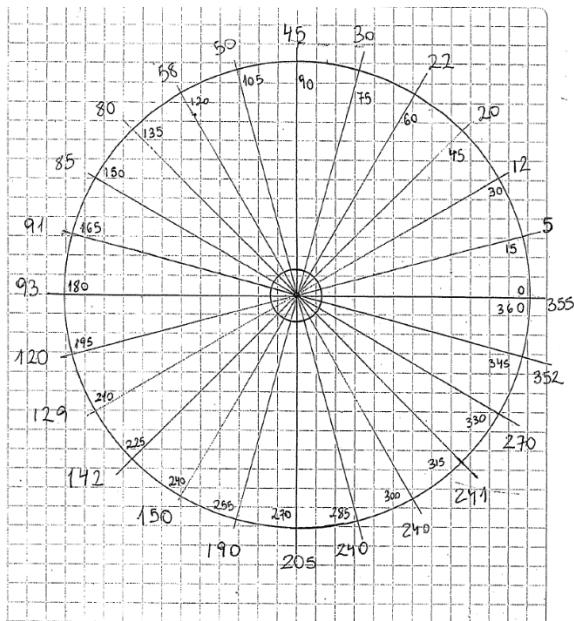


Ilustración 39: Mediciones con el magnetómetro tomadas en la facultad FCEFyN

Las mediciones fueron muy inconsistentes (tanto que fue necesario utilizar dos magnetómetros distintos para asegurarse de que era comportamiento normal).

Se concluye en que dado que el sensor está expuesto a factores externos, y su correcto funcionamiento depende del lugar en que se encuentre, se analiza otra alternativa: el giróscopo.

11.2.3.2 Acelerómetro, más giróscopo MPU6050

El módulo MPU6050 para Arduino, combina un giróscopo y un acelerómetro de 3 ejes, y brinda la posibilidad de hacer medidas de aceleración y velocidad angular en los ejes (x,y,z).

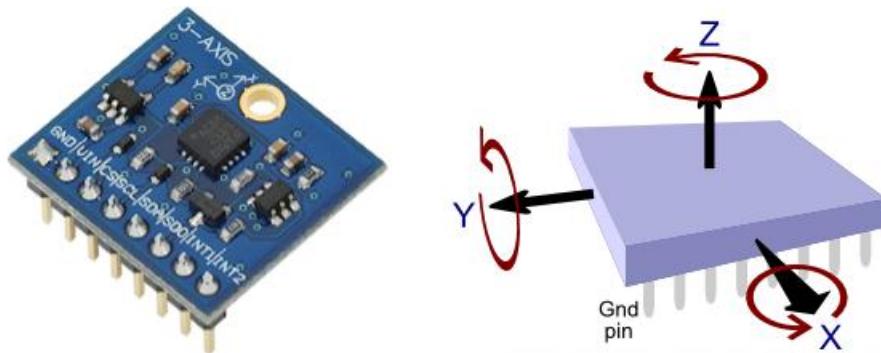


Ilustración 40: Giróscopo, ejes de referencia.

Integrando la velocidad angular respecto al tiempo, se obtienen los valores de orientación (yaw, pitch, roll). Estos son los valores que se utilizan para estimar la orientación del robot, es decir hacia donde está mirando.

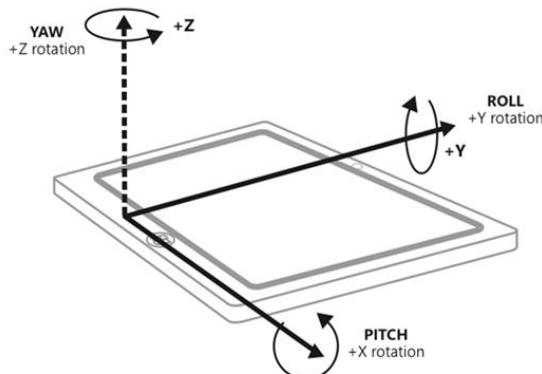


Ilustración 41: Relación entre (x,y,z) y (yaw,pitch,roll)

Ambas medidas, tanto aceleración como velocidad angular, son muy precisas y el chip cuenta con un conversor analógico digital de 16 bits para cada canal (Datasheet InvenSense) de tal forma que los valores entregados ya fueron digitalizados.

Ademas, sobre el mismo chip, hay una unidad de procesamiento llamada Digital Motion Processor (DMP) o procesador digital de movimiento, que puede ser programada para realizar todo el procesamiento independientemente del microprocesador central del robot (Arduino en este caso),

disminuyendo la carga sobre éste. Por otra parte, se cuenta con un sensor de temperatura que se puede utilizar para compensar la deriva térmica.

La comunicación entre Arduino y el sensor se logra mediante el protocolo I2C utilizando un buffer FIFO de 1024 bytes (MPU6050 Arduino Playground). Cada vez que se introduce un dato en la FIFO, se genera una señal de interrupción para avisar a Arduino que hay valores esperando ser leídos. Específicamente, nuestro proyecto utiliza la interrupción 0 (pin 2 de Arduino). Es posible modificar la frecuencia con la cual el DMP introduce datos a la pila para evitar la necesidad de revisarla tan frecuentemente sin que se desborde.

El firmware de la DMP y la librería utilizada para el MPU6050 son desarrollados y mantenidos por Jeff Rowberg. Se encuentran disponibles, en su repositorio personal (Rowberg). Para cumplir con los requerimientos de usabilidad y hacer el diseño más modular, se creó una librería para Arduino llamada “Gyroscope” de tal forma que simplemente incluyéndola, e inicializándola, luego llamando a los métodos `gyroGetData()`, y `getLastYaw()` se obtienen los datos del sensor, y luego el yaw que es el valor de interés:

```

6. #include <Gyroscope.h>
7. float headingBase;
8. Gyroscope g;
9.
10. //Initialization
11. g.mpuInit();
12. g.joinI2Cbus();
13.
14. //...Get data
15. g.gyroGetData();
16. headingBase=g.getLastYaw();
```

A continuación se muestra el tamaño del módulo a comparación del Arduino:

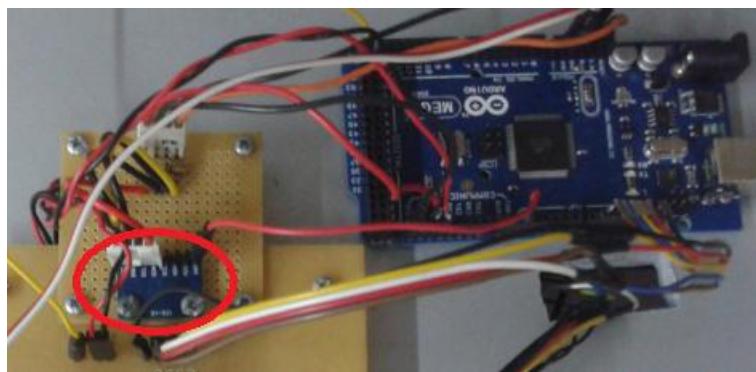


Ilustración 42: Conexión del giróscopo con Arduino

Al igual que con el magnetómetro, se realizaron pruebas para verificar el correcto funcionamiento del sensor. Para esto, se marcaron ángulos esperados sobre el perímetro de un círculo y se fue girando el giróscopo mientras se anotaban los valores medidos.

Las medidas obtenidas fueron satisfactorias, y dado que éste ha sido elegido como sensor de orientación para el robot, sus medidas se muestran en la sección de localización Test de localización.

11.2.4 Circuito electrónico

En esta sección se describe la electrónica necesaria para conectar gran parte de los componentes del robot. Se mostrarán las conexiones entre el microcontrolador Arduino, los motores, y los distintos sensores.

El robot cuenta con dos motores de corriente continua independientes controlados por Arduino. Cada motor cuenta con 2 cables, Vcc y GND, los cuales deben ser invertidos para cambiar el sentido de giro del motor. Esta tarea puede hacerse electrónicamente, sin tener que físicamente mover los cables, utilizando lo que se conoce como puente H.

- **Puente H:**

Se puede pensar un puente H como una configuración en la cual el estado de 4 llaves, abiertas o cerradas, determinan si va a circular corriente y en qué sentido. Esta corriente es la que pasará por el motor y lo hará girar en un sentido u otro. Una imagen puede demostrar la idea con más facilidad:

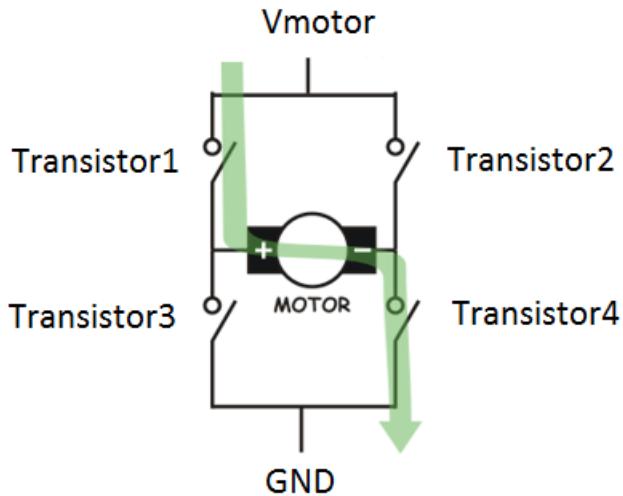


Ilustración 43: Puente H. Principio de funcionamiento.

La imagen muestra el sentido de la corriente en el caso que se cierren las llaves 1 y 4. Como efecto, el motor gira en un sentido. Para invertir el giro del robot, es necesario cerrar las llaves 2 y 3 y abrir la 1 y la 4.

Para evitar las llaves mecánicas y automatizar la operación, cada una puede ser sustituida por un transistor, también conocidos como llaves electrónicas. De esta forma, se puede cerrar o abrir cada compuerta con un “1” o “0” en la base del transistor.

Para frenar el motor, hay varias combinaciones posibles, una de ellas es abrir todas las llaves.

Pensando en el sistema binario, ya que $2^2=4$, es posible codificar cada posibilidad de giro de los motores con un circuito combinacional que tenga 2 bits de entrada. Es decir, el valor de dos entradas indican el estado deseado en cada motor:

IN1	IN2	Estado del motor
0	0	Frenado
0	1	Giro sentido 1
1	0	Giro sentido 2
1	1	Frenado

Ilustración 44: Puente H. Señales de entrada y sus efectos

Comercialmente, es posible encontrar chips integrados con el circuitario necesario para controlar el estado de 2 motores. Los más conocidos son el L293D y el L298. La mayor diferencia entre estos dos es la corriente total que pueden suministrar en pin de salida. En el primero, la corriente máxima de salida en cada pin es 600mA. En el segundo, cada pin puede suministrar hasta 2A. Teniendo en cuenta que en el perno de los casos el motor TT puede requerir cerca de 1A, para este proyecto se utilizó el L298. A continuación se muestra en esquemático del mismo:

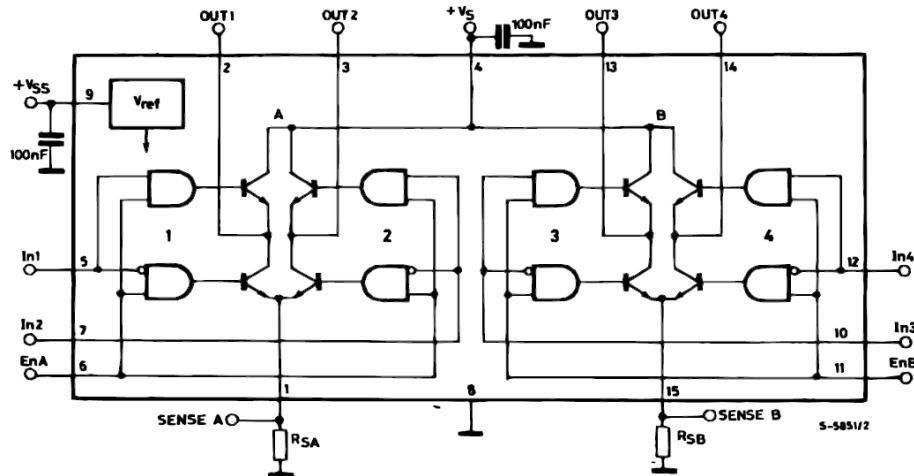


Ilustración 45: Circuito esquemático del L298 (Motor driver datasheet L298N)

Las entradas para controlar el puente H de cada motor son In1, In2 e In3, In4 respectivamente. Las señales de enable, EnA y EnB, se mantienen siempre en "1". Las salidas que van hacia los motores son Out1, Out2 y Out3,Out4 respectivamente.

El voltaje Vs, utilizado directamente por los motores, puede ser hasta 50V según (Motor driver datasheet L298N). El nivel de tensión para la lógica, Vss, hasta 7V. En este proyecto, la batería que alimenta a los motores es de 22,2V y el voltaje lógico es de 5V. Ya que no se desea medir la corriente en este proyecto, los cables en la sección inferior del esquemático son conectados a tierra.

Cada motor tiene dos estados: frenado o girando. En el último caso se debe controlar la velocidad de giro con el fin de controlar la velocidad de avance del robot. Dado que los motores son de corriente continua, su velocidad de giro está dada por la diferencia de potencial entre sus bornes. Se utiliza la técnica de modulación por ancho de pulso PWM para regular dicha velocidad. La salida de cada PWM se

aplica a las compuertas que controlan el estado de los motores. De esta forma, el voltaje Vs se mantiene constante en 22,2V pero este solo es aplicado a los motores cuando el pulso del PWM habilita el circuito.

- **Optoacopladores:**

Se utilizan optoacopladores de rápida respuesta con el fin de que la señal que provee el circuito de control se transfiera al circuito de potencia sin distorsiones. Además sirven para aislar ambos circuitos de tal forma de proteger el microcontrolador (entrada al circuito de control). Esto se logra con un fotoemisor y un fotoreceptor cuya conexión es óptica, es decir, que funciona como un interruptor activado por luz. Por este motivo, en este proyecto se utilizaron los optoacopladores de muy alta velocidad 6N137. Su velocidad es de 10MBit/s (Very high speed optocoupler 6N137 datasheet).

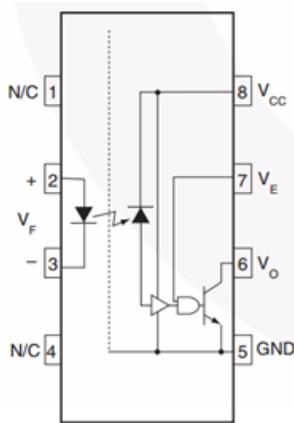


Ilustración 46: Circuito esquemático del optoacoplador. Modelo 6N137

- **Diodos flyback o fast recovery:**

Otro detalle importante a mencionar son los diodos presentes entre el puente H y los motores. Estos son muy importantes para evitar que el driver de los motores, en este caso el L298, no sea destruido por una gran corriente que se genera al invertir la polaridad del motor. Dado que un motor puede ser representado como una inductancia, se puede utilizar el siguiente modelo matemático:

$$V_L = -\frac{d\varphi_B}{dt} = -L \frac{dI}{dt}$$

Es decir, el voltaje generado en los bornes de una inductancia, es inversamente proporcional a la variación de la corriente. Cuando la corriente cambia rápidamente, para frenar el motor por ejemplo, se pueden producir voltajes inversos muy grandes capaces de quemar los componentes del driver (Wikipedia, Diodos flyback).

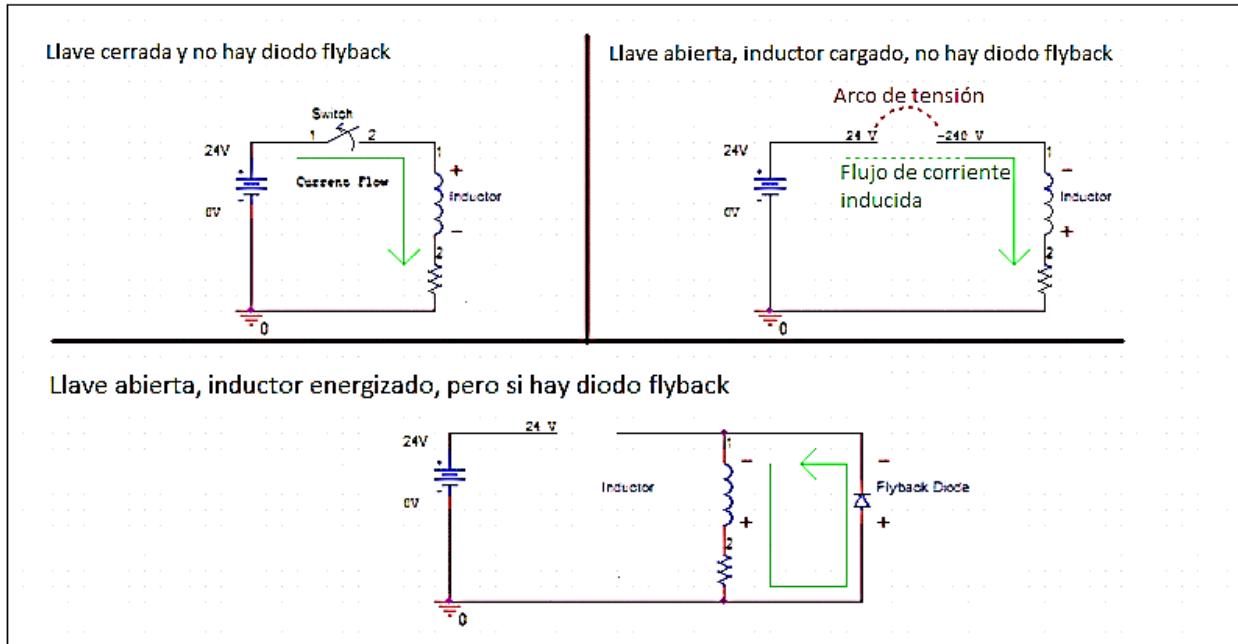


Ilustración 47: Diodos flyback

Esquemático

A continuación se muestra un esquemático general de todo el circuito que permite apreciar la separación entre el circuito de potencia y el de control. En él se incluyen las conexiones con Arduino, los motores, y encoders. Se omiten las conexiones con el giróscopó para mayor claridad de las partes importantes. Los cuatro integrados centrales son optoacopladores, y a su izquierda tenemos el circuito de control el cual incluye la placa Arduino, mientras que a su derecha el de potencia que se conecta a los motores. En la esquina superior de la imagen, se puede apreciar un regulador de tensión que obtiene 5V a partir de los 22,2V utilizados para alimentar la parte lógica en el circuito de potencia.

Cabe destacar que si bien los encoders se encuentran del lado derecho (de potencia) por estar cerca de los motores físicamente, pertenecen al circuito de control, por lo que no tienen ningún cable en común con la parte de potencia.

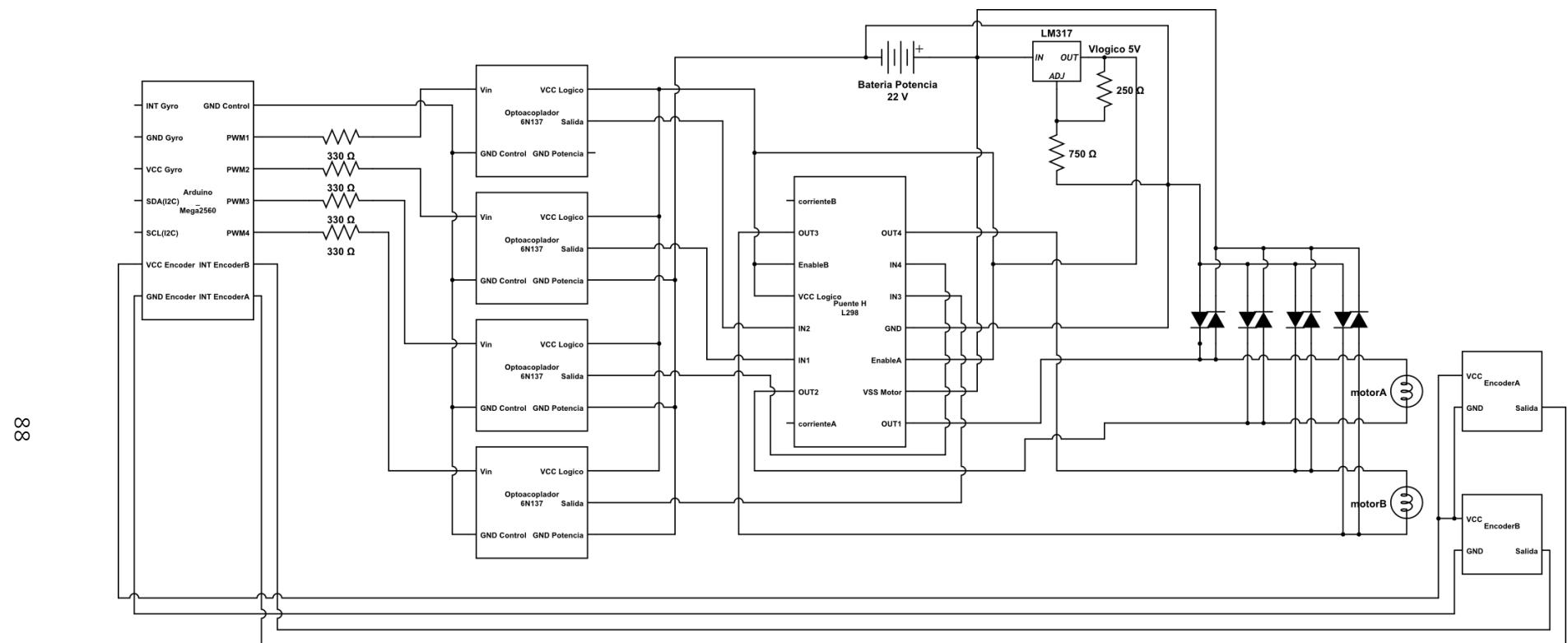


Ilustración 48: Esquemático del circuito. Parte de control a la izquierda de los optoacopladores y parte de potencia a la derecha

Inicialmente se construyó una primera versión de la placa como prototipo de prueba donde en lugar de usarse un L298 se había utilizado un L293, y no se tenían diodos de recuperación rápida.

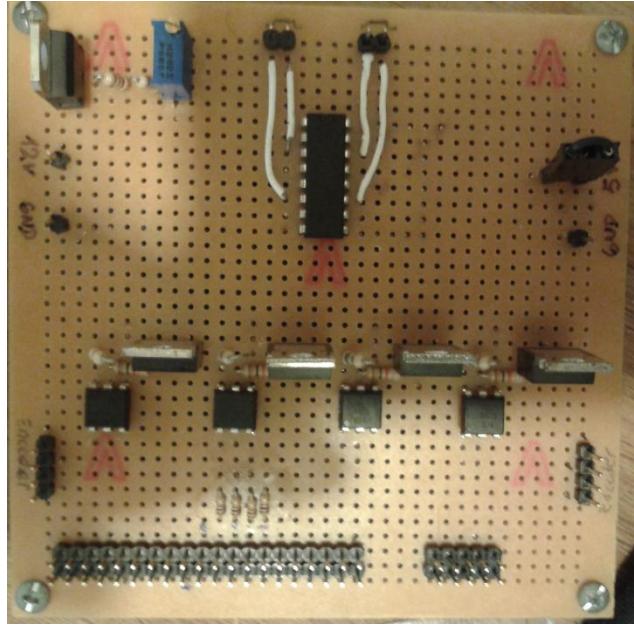


Ilustración 49: Primer prototipo de la placa electrónica para control de motores y encoders

Una vez comprobado su funcionamiento y luego de algunos cambios y mejoras se creó una placa PCB (del inglés Printed Circuit Board) con el fin de facilitar el proceso de conexión de componentes, hacer un diseño definitivo, y obtener un nuevo prototipo más prolíjo. Los componentes utilizados se listan a continuación:

- ✓ 2 Reguladores de tensión LM317
- ✓ 1 Puente H L298
- ✓ 2 Trimpot 5kΩ
- ✓ 4 Optocoupler 6N137
- ✓ 8 Diodos MUR460
- ✓ Resistencias

El diseño de la PCB se hizo con el software PCB Wizard y el resultado se muestra en la siguiente imagen:

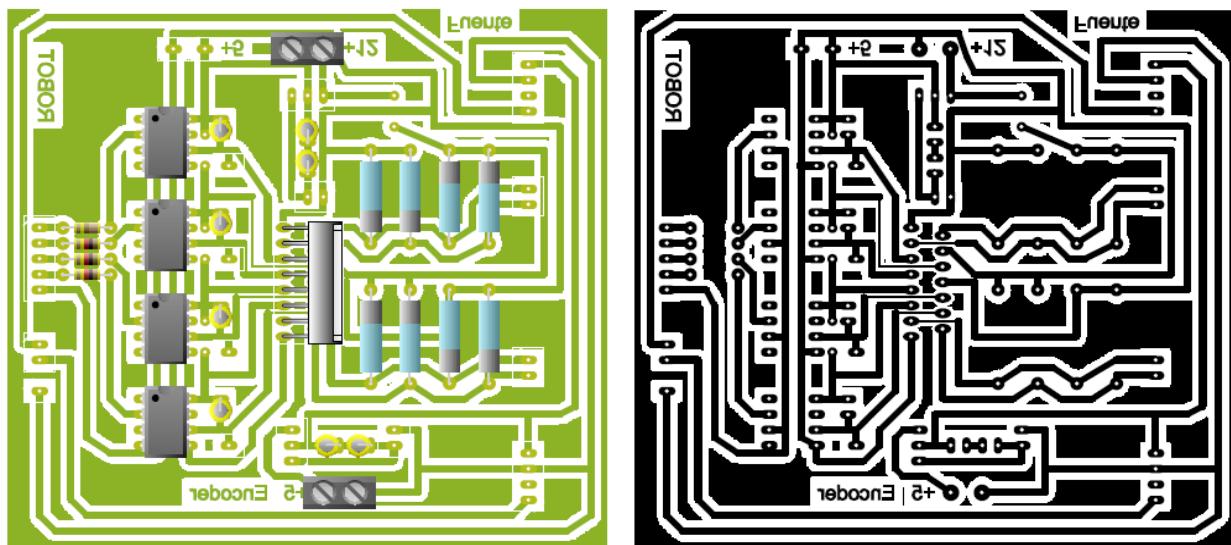


Ilustración 50: Placa PCB del circuito electrónico diseñada para el robot.

Por último, se muestra el resultado de todo el proceso descripto. En la imagen se puede ver el circuito ya conectado a Arduino. Además, se puede ver una pequeña placa aferrada a la placa grande, sobre la que se encuentra el giroscopio.

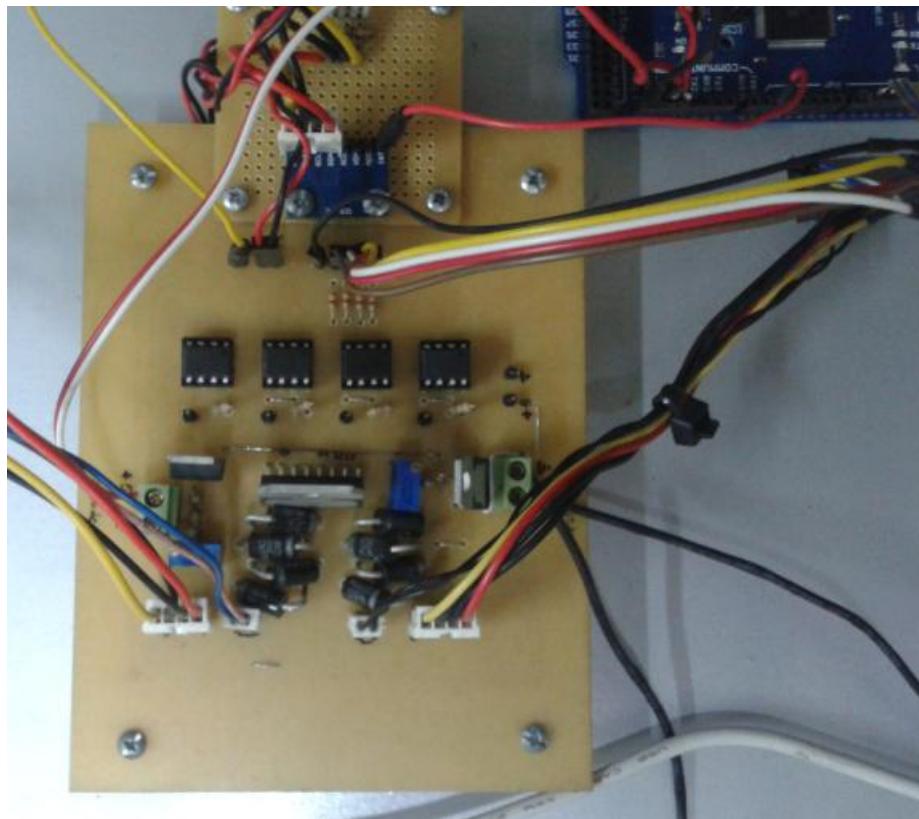


Ilustración 51: Foto del circuito implementado

11.2.5 Arduino

Arduino es un microcontrolador diseñado con el objetivo de facilitar el uso de la electrónica en proyectos multidisciplinarios. El hardware consiste en una placa cuya completa documentación se encuentra disponible como hardware libre. Todos los diseños del hardware (esquemáticos, PCB, código fuente HDL, etc.) y el software necesario están disponibles bajo el enfoque FOSS (software libre y gratuito, traducción de “Free and open software”) y pueden ser usados, copiados, estudiados y modificados.

Para el presente trabajo se utilizó una placa Arduino Mega 2560 que se basa en el microcontrolador ATmega2560 (AVR de bajo consumo de 8 bits de Atmel).

Especificaciones:

- Contiene 135 instrucciones (casi todas de ejecución en un ciclo)
- 32 registros de propósito general de 8 bits
- Hasta 16 MIPS (Millones de instrucciones por segundo) a 16 Mhz
- Multiplicador de dos ciclos
- 256 KB de memoria flash para código (de los cuales 8kb son del bootloader)
- 4Kbytes EEPROM (que puede ser leída y escrita con la librería de EEPROM)
- 8Kbytes SRAM interna
- 54 pines I/O (14 pueden ser usados como PWM)
- 16 entradas analógicas
- 4 UARTS
- Oscilador de cristal de 16 Mhz
- Conexión USB
- Operación en 5V, salidas de 5V y 3.3V
- Corriente DC en cada I/O pin 40 mA
- 6 pines de interrupciones externas: 2(int 0), 3, 21, 20, 19, 18(int 5).

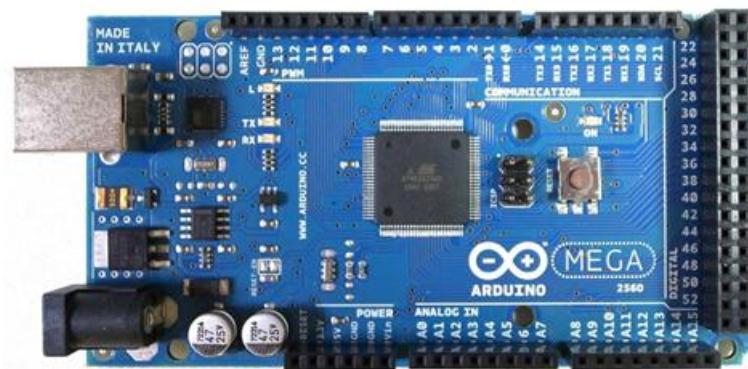


Ilustración 52: Microcontrolador Arduino Atmega2560

La placa, además, cuenta con un conector de alimentación cuyos voltajes pueden ir desde 6V a 20V pero se recomienda utilizar entre 7 y 12 volts (Arduino Mega2560).

En la siguiente imagen se señalan los pines utilizados para el Robot, y se escribe a que componente se conecta o que función cumple:

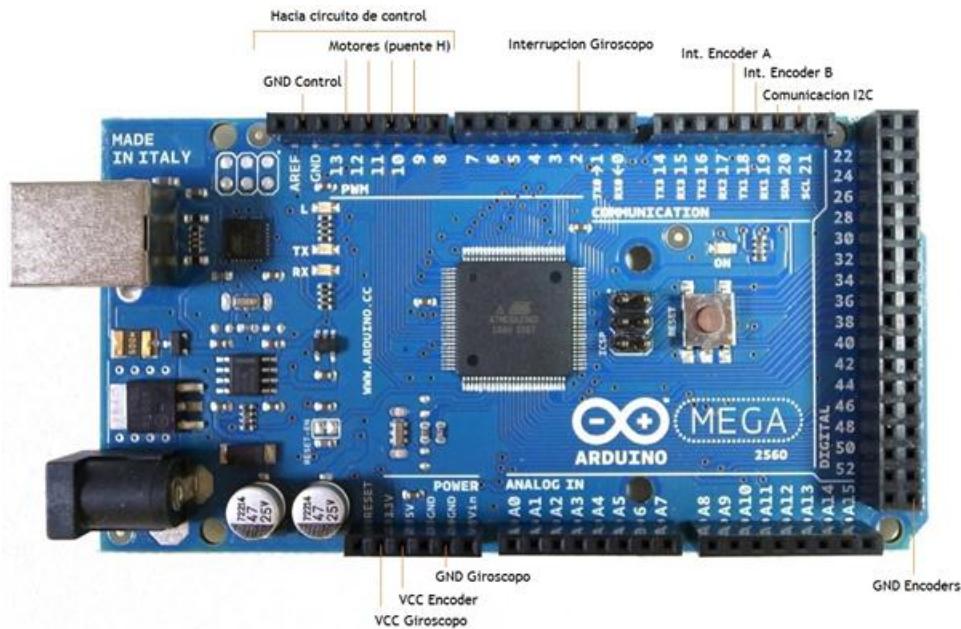


Ilustración 53: Conexiones y pines del Arduino utilizadas para el presente trabajo

Modulación de ancho de pulso PWM con Arduino

En las secciones anteriores se mostraron los motores utilizados. Al tratarse de motores de corriente continua, es posible regular la velocidad de revolución ajustando los voltajes de entrada. Específicamente, para los motores de este proyecto, el rango admisible de voltaje es 4-28V. Esto implica que la velocidad mínima posible se obtendrá al aplicar 4V y la velocidad máxima se obtendrá con 28V.

Una de las técnicas más comunes para variar el voltaje de salida del microcontrolador se llama modulación de ancho de pulso (o PWM por su nombre en inglés pulse-width modulation). En esta técnica, se utiliza una señal periódica de alta frecuencia con un ciclo de trabajo variable (duty cycle en inglés). De esta forma, el voltaje efectivo obtenido es función del duty cycle:

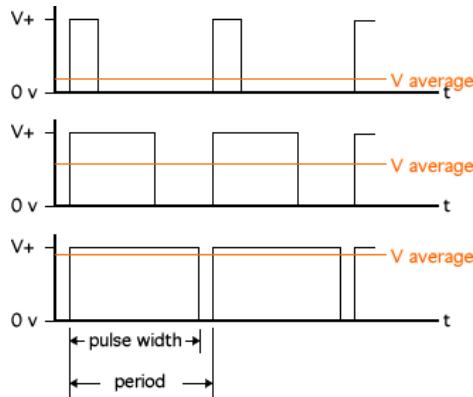


Ilustración 54: PWM y voltaje efectivo

La imagen muestra que a medida que el ciclo de trabajo de la señal aumenta, el voltaje promedio también lo hace. El voltaje mínimo se da cuando el ancho es cero ya que la señal está durante todo su período en cero. Por el contrario, el voltaje máximo se obtiene al tener la señal en alto durante todo el período.

Para el caso de señales con formas más complejas, el voltaje promedio o efectivo se obtiene mediante una integral de la señal respecto al tiempo con el fin de promediarla durante todo su período:

$$y = \frac{1}{T} \int_0^T f(t) dt$$

Pero para el caso de una onda cuadrada, la relación entre el voltaje de salida y el voltaje efectivo es una simple multiplicación:

$$V_{\text{efectivo}} = V_{\text{Salida}} * \text{dutyCycle}$$

Es común que los microcontroladores se encuentren equipados con la capacidad de modular el ancho de una salida. Para utilizar esta funcionalidad, algunos controladores requieren una configuración previa de registros de reloj, como el caso del LPC1343, y otros, como Arduino disponen de una función lista para usarse en la cual simplemente se especifica el ancho deseado.

En el caso particular de Arduino, la función `analogWrite(int dutyCycle)` es la encargada de regular el ancho. Como parámetro se especifica un valor de 8 bits (de 0 a 255) que representa el porcentaje de ciclo de trabajo. Como es de esperar, 0 es el máximo y 255 el mínimo. La imagen siguiente muestra la salida para distintos valores.

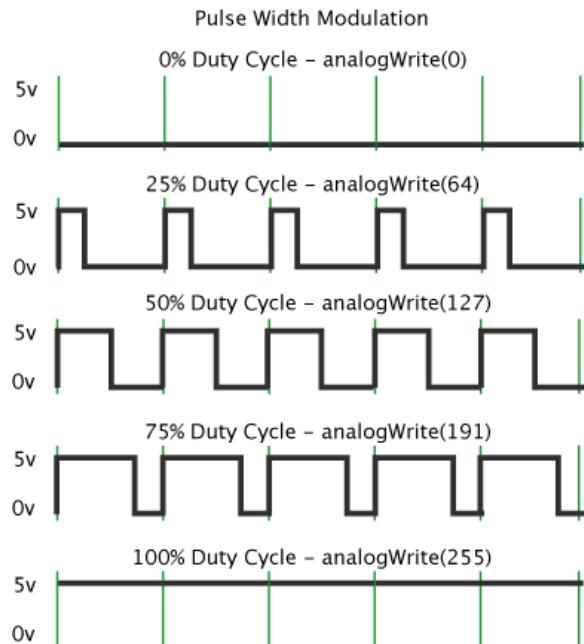


Ilustración 55: PWM y su utilización en Arduino.

Tal como se mencionó, por varias razones, es importante separar el circuito de control y potencia en dos partes. Se lo había logrado agregando optoacopladores para aislar ambos circuitos.

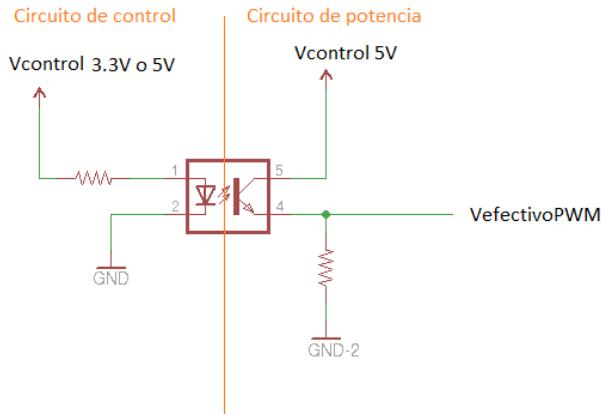


Ilustración 56: Separación entre el circuito de potencia y el de control mediante un optoacoplador

De esta forma, no hay contacto eléctrico entre ambos circuitos. Pero, para que el funcionamiento sea el esperado, es necesario que el optoacoplador pueda cambiar de estado tan rápido como lo hace el PWM. Es decir, necesitamos que el optoacoplador tenga tiempos de respuesta muy altos para cambiar su nivel lógico a medida que el PWM lo requiere, de lo contrario, se obtendría una señal distorsionada en el circuito de potencia. Por dicha razón, para el diseño del circuito electrónico se eligió usar optoacopladores de alta velocidad 6N137 (Very high speed optocoupler 6N137 datasheet).

A continuación se muestra la plataforma inferior del robot completa:

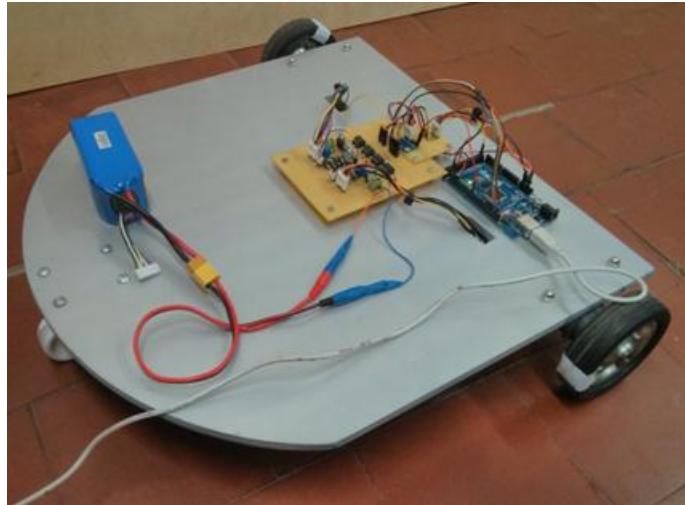


Ilustración 57: Plataforma inferior terminada. Entre los componentes que van sobre esta, se distinguen la batería, el microcontrolador y el circuito de potencia y control.

11.2.6 Baterías Li-PO

Se hace una breve descripción de cómo están compuestas las baterías, sus características y requisitos a tener en cuenta para la selección de la misma.

Estructura

Las baterías internamente están divididas en celdas. Cada celda tiene un voltaje nominal de 3,7V. Las mismas no pueden sobrepasar los 4,2 voltios en carga máxima y no pueden bajar de 2,8 ó 3 voltios en descarga (dicho valor depende del fabricante). Este es el límite. Las celdas pueden estar conectadas internamente en serie o paralelo. Esto permite obtener la nomenclatura de las mismas donde "S" define el voltaje final de la batería, cada "S" son 3,7V:

- Una Li-Po de 2S1P tiene dos celdas en serie
- Una Li-Po de 2S2P tiene dos celdas en serie y dos en paralelo.

Capacidad

Es la corriente que es capaz de suministrar durante una hora, medida en [mA]. Las "C" hacen referencia a las veces que un Li-PO es capaz de suministrar su capacidad, es decir, una Li-PO de 6S1P, 3000mA con 20C continuos y 30C de pico, tiene un voltaje nominal de 22,2V, una capacidad de 3000mA, y puede realizar una descarga continua de 60A y una descarga puntual de 90A. Obviamente, mientras mayor capacidad tenga la batería, más tiempo tardará en descargarse.

Elección de baterías

Para elegir qué tipo de batería se adecúa al robot, es necesario calcular la corriente máxima que consumen nuestros motores. Una buena práctica es elegir un valor que sea un 50% de la corriente de descarga continua de la Li-PO, es decir, si tenemos que dos motores consumen 3A, debería suministrar como mínimo 6A continuos. Para este caso necesitaríamos una Li-PO de 240mA con 25C de descarga continua.

En la Tabla 12 de la sección 11.2.2.1 se muestran las corrientes máximas de los motores utilizados para el robot. Para el rango de voltajes 4v-28V, en el peor de los casos será de 0.8A. Dado que se tienen 2 motores, la corriente máxima que deberá entregar la batería en el peor de los casos es de 1.6A. Se recomienda elegir una batería cuyo valor de descarga continua sea un 50% más de la corriente que se necesita, es decir, se debería elegir una batería que suministre 3.2A continuos.

La batería adquirida finalmente es:

Li-PO Zippy Flightmax 6s 22.2v 3000mah 20 / 30c.



Ilustración 58: Batería Li-Po de 22.2 V y 3000mA (Zippy Flightmax 6S1P).

Especificaciones:

- Capacidad: 3000mAh
- Voltaje: 6S1P / 6 Cell / 22.2v
- Descarga: 20C continua / 30C puntual
- Peso: 429g
- Dimensiones: 106x45x43mm
- Plug de balanceo: JST-XH
- Plug de descarga: XT60
- Tasa máxima de carga: 2C
- Costo: 25USD

Es decir que tiene 6 celdas en serie, “6S1P” que hacen un total de 22.2V. Con una capacidad de 20C continua y 30C de pico, lo que significa que tiene una capacidad de descarga continua de 60A, y una descarga puntual de 90A, o de 3A en el lapso de tiempo de 1 hora. El valor de descarga continua es muy superior al requerido que era 3.2A, pero esto se toma como una ventaja ya que permitirá que el robot circule durante 1 hora con el mayor de los consumos. Además, bajo condiciones normales el robot podrá moverse durante mucho más tiempo que 1 hora.

La siguiente curva de carga/descarga muestra la capacidad de descarga de una Li-Po de 3300mA y 21.0V muy similar a la adquirida. Es importante mencionar que se muestra dicha imagen, dado que en las especificaciones de la batería que se compró no se tenía esta información³¹. El análisis de esta figura es que mientras más capacidad C tiene una batería, el nivel de tensión se mantiene casi constante a lo largo de todo el proceso de descarga (la curva está más acostada), por el contrario mientras menor es la capacidad, la batería se descarga más rápidamente.

31 (2-5-13) <http://www.chargery.com/batteryPHE.asp>

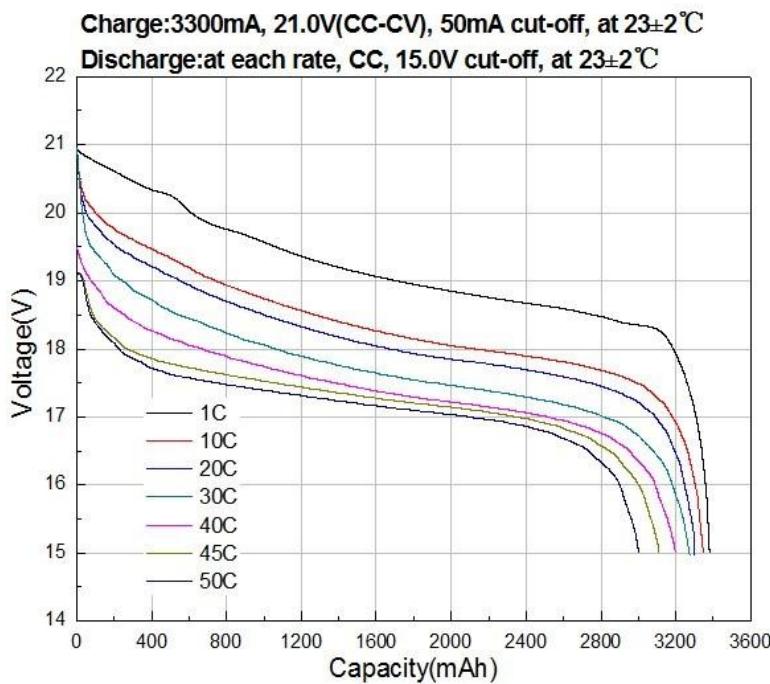


Ilustración 59: Curvas de descarga de una Li-Po con distintas capacidades

Proceso de carga de las baterías

Como se dijo anteriormente no se puede exceder los 4.2V por celda. Si esto ocurre la batería se deteriora, incluso son inflamables. Por dicha razón, un cargador de Li-PO tiene que tener carga con balanceo sí o sí. Otros aspectos a tener en cuenta son el número de celdas en serie que puede cargar, la potencia de carga y potencia de descarga.

- **Potencia de carga:** se refiere al valor de potencia necesario para cargar la batería. Para saber que potencia es la que se necesita, se multiplica el voltaje máximo de la batería por la intensidad a la que se quiere cargar la misma.

Ejemplo: Para cargar una Li-Po de 4S y 3000mA a la máxima intensidad que nos permite la batería, se necesitaría una potencia de carga de $4 \times 4,2V \times 3A = 50,4W$. Un cargador que entregue una potencia de carga aproximada de 50W permitiría cargarla en 1 hora.

En cambio si tuviera 6S y 3000mA se necesitan $6 \times 4,2V \times 3A = 75,6W$. Un cargador de 50W no permitiría cargarla en una hora, sino que tardaría una hora y media aproximadamente.

- **Potencia de descarga** es igual que el anterior pero se refiere a la potencia necesaria para descargar la batería.

Entonces para la carga se utiliza un cargador/balanceador, y es necesario conocer las características de la Li-PO para ver cuál es la máxima intensidad de carga (la gran mayoría tienen una intensidad máxima de carga de 1C, o en otros casos de 2C).

Ejemplo: una Lipo de 3000mA que se carga a 1000mA, tarda aproximadamente 3 horas en cargar, pero durará alguna carga más. En caso de querer bajar el tiempo de carga se podría cargar a 3000mA, es decir a 1C.

Se adquirió un cargador TURNIGY ACCUCELL-6 balance charger/discharger para baterías NiCd/NiMH/Litio/Pb.



Ilustración 60: Cargador de las baterías

Especificaciones:

- Voltaje de operación: 10.0-18.0V
- Potencia de carga: max.50W para carga
- Potencia de descarga: max.5W para descarga
- Corriente de carga: 0.1-5.0A
- Corriente de descarga: 0.1-1.0A
- Corriente de drain para balanceo de Li-po: 300mAh/celda
- Cantidad de celdas para baterías NiCd/NiMH: 1-15cells
- Cantidad de celdas para baterías de Litio: 1-6Series
- Voltaje para baterías Pb: 2-20V
- Peso: 400g
- Dimensión: 135×100×40mm

Lo primero que debe hacerse es conectar el cargador a una fuente de entre 11-18V.

Parámetros iniciales de carga

- 1) Establecer el voltaje de las celdas de la Li-PO a 3,7V.

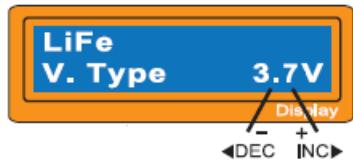


Ilustración 61: Cargador de baterías. Establecimiento del voltaje de carga

- 2) Se establece un tiempo de seguridad. Cuando comienza el proceso de carga, el timer de seguridad comienza a correr. Debe ser programado para prevenir una sobrecarga de la batería si está fallada, o si el mecanismo de terminación no puede detectar que la batería ya está totalmente cargada. Se debe programar con un valor que permita terminar el con el proceso de carga.



Ilustración 62: Cargador de baterías. Establecimiento del tiempo de carga

- 3) Establecer la máxima capacidad de carga que será suministrada a la batería durante el proceso de carga. Para nuestra batería es necesario configurarla a 3000mA.

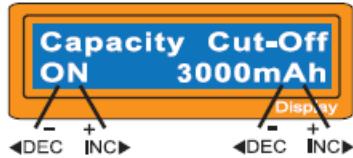


Ilustración 63 Cargador de baterías. Establecimiento de la corriente

Programacion para baterías Li-PO

Este tipo de baterías necesitan ser cargadas a voltaje constante y corriente constante. La corriente de carga varía en función de la capacidad y performance de la batería. El voltaje final del proceso de carga es muy importante, debe ser exactamente el valor para dicha batería 4,2V (en Li-PO).

Antes de seleccionar el modo se debe conectar la salida del cargador a la batería con los plugs negro y rojo en el orden que corresponden (según se muestra en la figura). Importante: si se conectan al revés estos últimos directamente se daña el cargador.



Ilustración 64: Alimentación del cargador

Carga de baterías Li-PO

- Seleccionar el modo Li-PO CHARGE. Luego establecer la corriente a 3.0A y el voltaje a 22.2V. La corriente puede ser menor a 3.0A si se quiere cargar en más tiempo. Esto hará que la batería dure algunas cargas más.

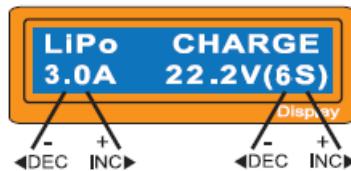


Ilustración 65: Cargador de baterías. Establecimiento del modo para la batería utilizada

- Ahora se establece la cantidad de celdas que se visualiza en S:, y en R: se muestra la cantidad de celdas detectadas por el cargador en la batería. Si ambos números son idénticos se puede comenzar a cargar presionando el botón Start durante 3 segundos.

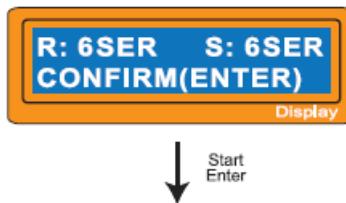


Ilustración 66: Cargador de baterías. Establecimiento de la cantidad de celdas

El display muestra los parámetros del proceso de carga. Para parar la carga se presiona Type/Stop.

Carga de baterías Li-PO en modo de balanceo

En el modo balanceado antes de conectar la salida del cargador a la batería es necesario conectar el puerto de balanceo de la batería (conector blanco) en el cargador.

En este modo, el procesador interno del cargador monitorea los voltajes de cada celda controlando la corriente de carga con que se alimenta cada una para normalizar su voltaje.

- Seleccionar el modo Li-PO BALANCE. Luego establecer la corriente a 3.0A o menor a ésta y el voltaje a 22.2V.

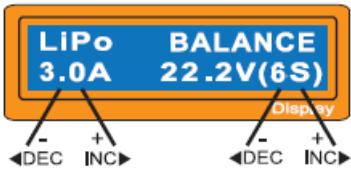


Ilustración 67: Cargador de baterías en modo balanceo. Voltaje.

- b. Ahora se establece la cantidad de celdas que se visualiza en S:, y en R: se muestra la cantidad de celdas detectadas por el cargador en la batería. Si ambos números son idénticos se puede comenzar a cargar presionando el botón Start durante 3 segundos.



Ilustración 68: Cargador de baterías en modo balanceo. Celdas.

El display muestra los parámetros del proceso de carga. Para parar la carga se presiona Type/Stop.

Además de los modos descriptos hay 3 modos más que pueden ser consultados en el manual: Carga rápida de baterías de Litio, Control de almacenamiento de baterías de Litio, y modo de Descarga de baterías de Litio.

Monitoreo durante el proceso de carga

Durante el proceso de carga, en el display se muestra información del proceso de carga actual.

Presionando el botón DEC el cargador muestra la configuración de parámetros iniciales elegida por el usuario. Además si se presiona el botón INC se puede ver el voltaje individual que tiene cada celda, si es que el conector blanco de la batería fue conectado a la entrada que le permite al cargador conectarse con cada una de ellas.

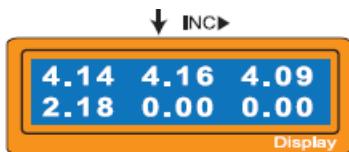


Ilustración 69: Cargador de baterías. Muestra de estado actual de la batería.

Si una celda de la batería está dañada, puede ser fácilmente comprobado presionando el botón INC para ver de qué celda se trata.

Forma de almacenarlas

Este tipo de baterías son delicadas y las altas temperaturas se deterioran. No es aconsejable almacenarlas con la carga máxima puesto que la actividad interna es alta y esta termina deteriorando la

batería. Tampoco es recomendable almacenarlas con carga mínima puesto que con el tiempo la carga baja y podría bajar de 3 voltios por celda produciendo el deterioro de la batería.

Lo recomendable es utilizar la carga para almacenamiento que tienen los cargadores de Li-PO, este tipo de carga deja la batería a unos 3.85V por celda, siendo este voltaje el recomendado para su almacenamiento.

El uso correcto de este tipo de baterías sobrepasa considerablemente el rendimiento de las baterías de NiMH (RC explosion).

11.2.7 Sensor Kinect

Por último se muestra la plataforma superior sobre la cual se monta la computadora a bordo y el sensor Kinect utilizado para la exploración.



Ilustración 70: Una vez colocada la base superior, se cuenta con varillas que permiten el ajuste de la altura de la misma.

La altura de la plataforma superior es ajustable. En la siguiente imagen se muestra la altura definitiva:



Ilustración 71: Altura de la base superior ajustada hasta el nivel definitivo. Adicionalmente, se observan las conexiones entre la PC y el microcontrolador

Comercialmente, Kinect viene preparada para ser utilizada ya sea con una computadora o con la Xbox. En consecuencia, viene equipada con un cable para enchufarla a la toma de corriente de la pared (110V-240V) y un adaptador que produce un voltaje de salida de 12V de corriente continua. Este último es el que utiliza el sensor.

Ya que el robot construido en este trabajo es un robot móvil, no es admisible un largo cable para alimentar a Kinect, el robot debe ser capaz de desplazarse libremente sin las limitaciones que generaría un cable. Para esto, se cortaron los cables justo antes del adaptador y se lo conectó a una batería de 12V. En cuanto a corriente, los requerimientos de Kinect son de 1Ah y la batería utilizada suministra 1.3 Ah. Como guía, se utilizó un documento que explica la adaptación de Kinect al iRobot Create (How to connect Kinect to 12V battery).

La siguiente imagen muestra el cable comercial y su uso convencional. Se puede ver donde fue cortado:

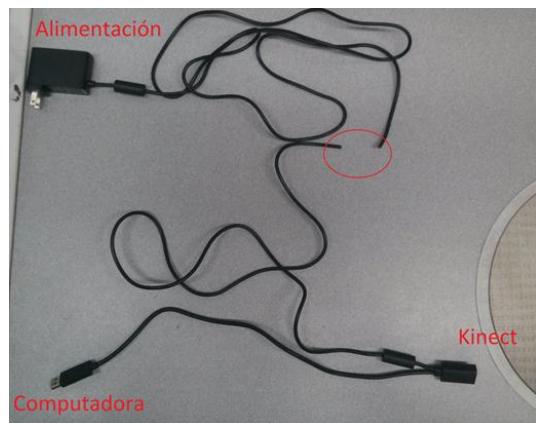


Ilustración 72: Cable de alimentación de Kinect fue cortado para introducir una batería como fuente de energía.

Robot móvil autónomo capaz de producir mapas 3D del interior de una habitación

Los cables para la alimentación del robot se identifican a continuación:

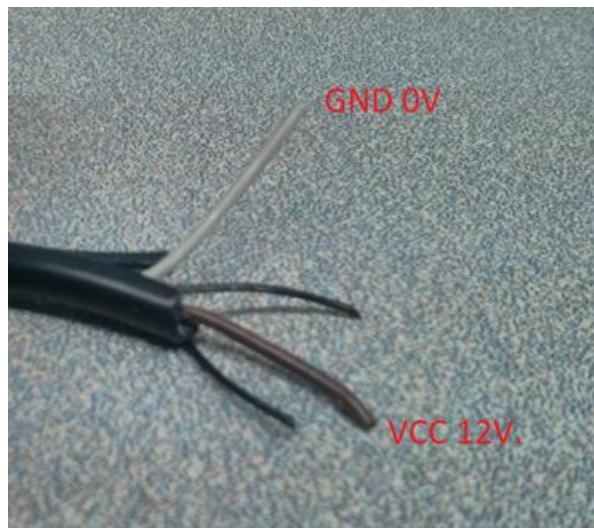


Ilustración 73: Cables de Kinect que se conectan a los bornes de la batería.

La batería utilizada para alimentar al Kinect una vez a bordo del robot es una batería de gel de 12V y 1.3Ah que estaba disponible para su uso en el laboratorio.



Ilustración 74: Batería de 12 V, 1A encargada de alimentar Kinect.

Por último, se muestra la conexión completa, y el funcionamiento de Kinect usando la batería. La computadora muestra la imagen de profundidad producida proveniente del sensor.

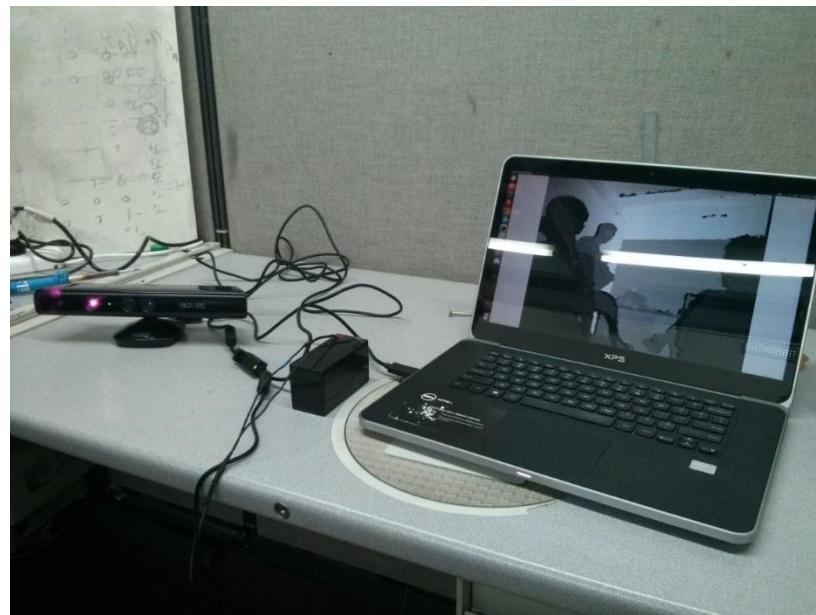


Ilustración 75: Kinect alimentada por batería en funcionamiento. La computadora muestra la imagen de profundidad.

Finalmente se muestran algunas imágenes del robot terminado, con sus dos plataformas, la computadora a bordo, más el sensor Kinect:



Ilustración 76: Robot definitivo. Mirando en la dirección contraria al avance del robot.

Robot móvil autónomo capaz de producir mapas 3D del interior de una habitación



Ilustración 77: Robot definitivo, mirando en la dirección de avance del robot. Se puede apreciar la imagen de profundidad producida por Kinect.



Ilustración 78: Robot definitivo con todos sus componentes conectados y funcionando

11.3 Control

11.3.1 Introducción

En condiciones ideales, ambos motores del robot deberían ser idénticos, y si se les proporciona la misma entrada, deberían avanzar a la misma velocidad. En la realidad, esto no sucede y las pequeñas diferencias entre los motores resultan en desvíos del robot. A continuación, se procederá a explicar la teoría e implementación de un controlador de trayectoria que fue utilizado para el robot.

Al desarrollar un controlador de trayectoria, se deben considerar dos direcciones en relación a la trayectoria: *along track* y *cross track* (Udacity CTE). La velocidad del robot, por ejemplo, puede ser descompuesta en dos componentes:

- 1) Una a lo largo de la trayectoria, paralela a la trayectoria (*along track*) y
- 2) Otra que atraviesa la trayectoria, perpendicular a la trayectoria (*cross track*).

En el siguiente caso se busca lograr que el robot se mueva en línea recta sobre el eje X. Entonces, el objetivo del controlador es que el CROSTRACK ERROR ó CTE disminuya hasta cero. Para ello, se implementa un controlador PID (Proporcional Integrador Derivador).

El principal objetivo del controlador es orientar el ángulo de giro del robot según la línea de trayectoria deseada para disminuir el CTE a cero. Por ende, mientras mayor sea el CTE, mayor será el ángulo de giro necesario que apunte hacia la referencia de trayectoria. A medida que el robot se acerque a la línea de trayectoria, el ángulo será cada vez menor, hasta que se alcance la línea de trayectoria y se avance en línea recta.

Como se mencionó anteriormente, el movimiento del robot está basado en el modelo cinemático diferencial, es decir el robot controla el ángulo de giro a través de la diferencia de velocidad de sus dos motores. Dado que el prototipo fue implementado con 4 comandos, up, down, right y left, se busca que ambos motores giren a la misma velocidad siempre, y con sentidos contrarios en el caso de left y right. Es decir, al darse el comando up, se desea que ambos motores giren a la misma velocidad, produciendo un avance sin desvíos. Cabe mencionar que la velocidad de las ruedas se mide en revoluciones por minuto (RPM).

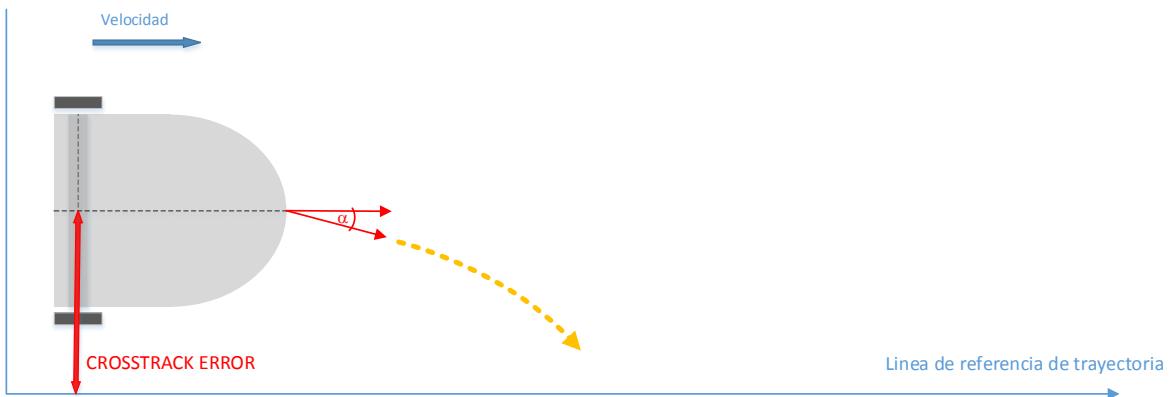


Ilustración 79: Control. crosstrack error

11.3.2 Controlador proporcional (P-Controller)

La letra P, en P-controller, proviene de la palabra “proporcional”, es decir, el ángulo de corrección indicado por el controlador es proporcional al error crosstrack. La constante de proporcionalidad es un valor denominado K_p . Se utilizará el símbolo α (alfa) para representar al ángulo de giro de corrección:

$$\alpha = K_p * CTE$$

Luego de sucesivas correcciones, el robot llegará a la línea de referencia y la sobrepasará. Por ello, el ángulo de corrección será en sentido inverso. Se producirán oscilaciones cada vez de menor amplitud hasta llegar a un cierto límite. Este efecto de oscilación y las amplitudes del mismo dependen del valor de la constante K_p . Un K_p mayor implica correcciones más significativas en cada paso, pero también un sobrepasamiento mayor y un error estacionario mayor. Dado que el error en estado estacionario es distinto de cero, el robot no dejará de oscilar.

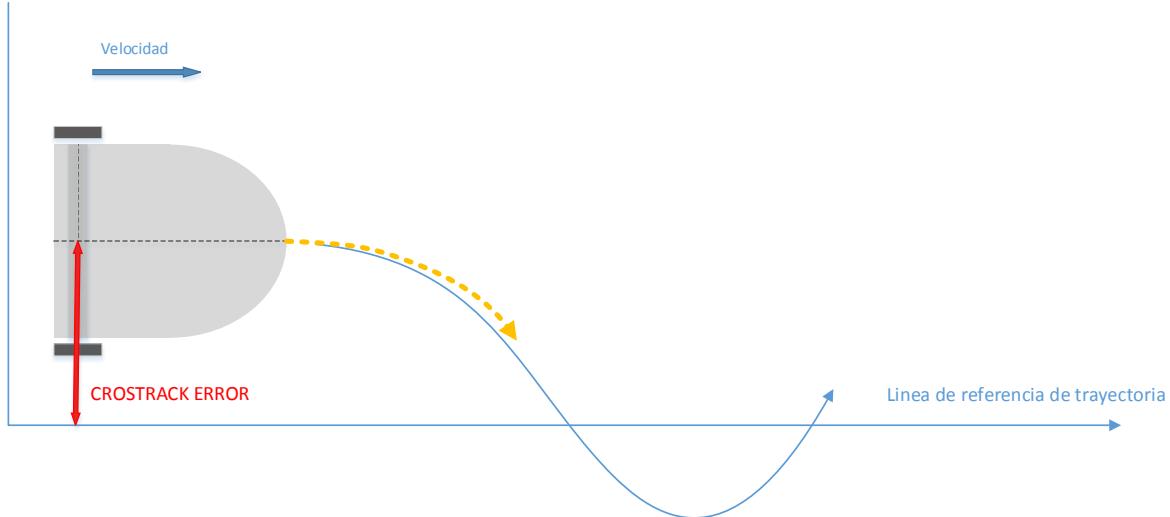


Ilustración 80: Corrección de ángulo y sobrepasamiento con el controlador proporcional

En la figura se intenta ilustrar que el robot tendrá un ligero sobrepasamiento y que luego va a mejorar. Dado que el sobrepasamiento es muy pequeño, desde el punto de vista de la orientación final, no tiene demasiada importancia. Pero, en realidad, el sistema nunca converge. Este problema se denomina “Marginally stable response”. Con el fin de mejorar las correcciones, hacerlas más adaptivas y atacar el sobrepasamiento, se introduce un nuevo término a la ecuación del controlador.

11.3.3 Controlador proporcional derivador (PD-Controller)

Este nuevo diseño proporciona una manera de evitar un sobrepasamiento pronunciado, evitando la oscilación del robot. Para lograrlo, se busca disminuir las correcciones a medida que el robot se acerca a la línea de referencia (a medida que el error CTE se hace más pequeño) y aumentar la corrección a

medida que se aleja. De esta forma, la corrección es mayor cuando el robot se encuentra más alejado de la orientación deseada y es menor cuando se está acercando.

Dado que la derivada indica pendientes instantáneas, puede utilizarse, derivando CTE respecto al tiempo, para medir los cambios en la orientación. Al comienzo, el cambio será mayor debido al controlador proporcional y luego irá disminuyendo por el efecto del derivador. Al llegar a la orientación deseada la trayectoria del robot es paralela a la línea de referencia y por ende la derivada es cero, ya que el CTE es constante. Pero, el hecho de que sean paralelos, no implica que sean coincidentes, esto será analizado luego en la siguiente sección.

Como se dijo, la corrección alfa no solo se mantiene proporcional al CTE, sino que se introduce la derivada del CTE respecto al tiempo. Computacionalmente, se la puede calcular instante tras instante como una diferencia de errores:

$$\frac{dCTE}{dt} = CTE_t - CTE_{t-1} / \Delta t$$

Reemplazando:

$$\alpha = K_p * CTE + K_d \frac{dCTE}{dt}$$

Haciendo una elección apropiada de las constantes K_p y K_d , el robot se aproximará de manera “elegante” a su trayectoria deseada.

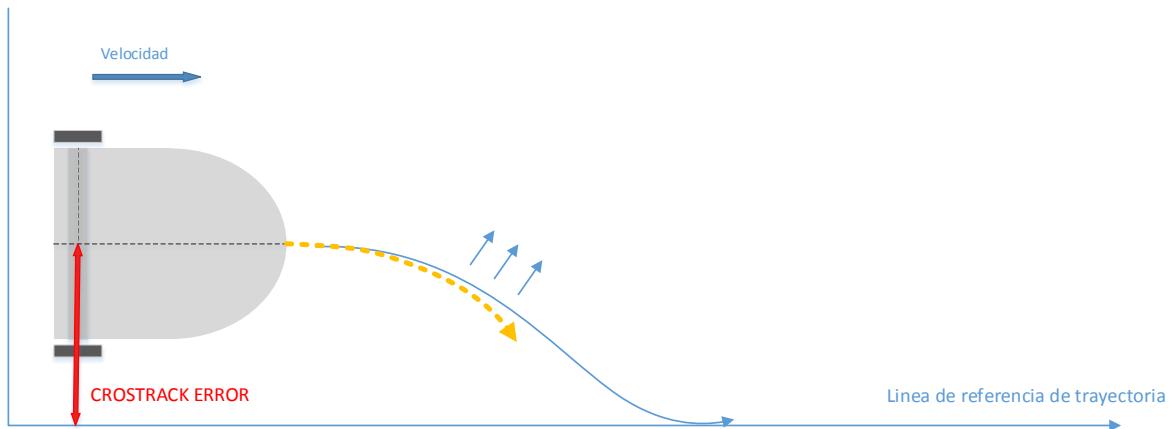


Ilustración 81: Corrección del desvío introducida por el controlador PD

Es decir, ahora, el control no solo se calcula en relación proporcional al CTE con K_p , sino que también se calcula en relación a la diferencia de errores usando una segunda constante K_d .

En la implementación del presente trabajo, el Δt está dado por una constante llamada LOOPTIME que fue establecida a 100ms. Mientras que el CTE es calculado como la diferencia entre las RPM medidas y las requeridas para cada motor. La medición de las RPM de cada motor se basa en los pulsos de los Encoders presentes en cada motor (se calcula la diferencia de pulsos).

Por último, como se dijo, el hecho de que sean paralelos, no implica que sean coincidentes (resultado en un error estacionario constante) y, por esto, es necesaria la introducción de un último término.

11.3.4 Controlador proporcional integrador derivador (PID – Controller)

Este último error presente se denomina desviación sistemática o “Systematic bias”. En el caso de un vehículo móvil (ya sea un auto, un robot, u otro) una posibles causas de desviaciones sistemáticas es el hecho que las ruedas podrían no estar perfectamente alineadas.

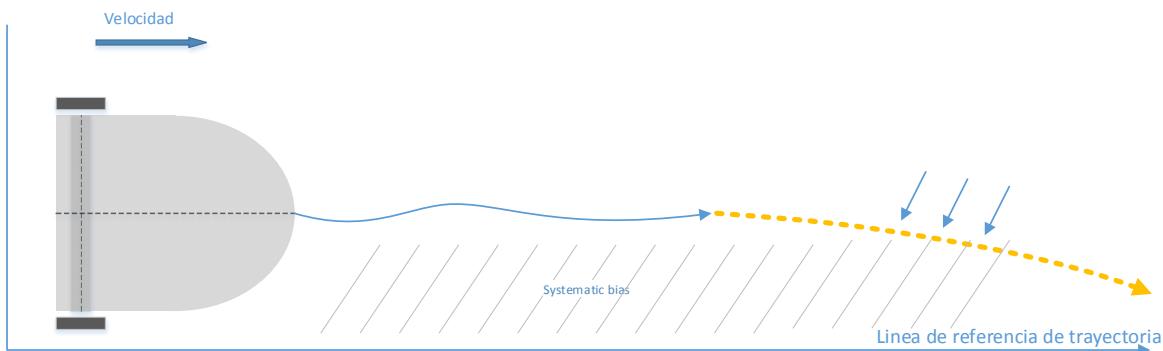


Ilustración 82: Desviación sistemática. Necesidad de un controlador PID

Lo mismo ocurre con el robot, cuando éste avanza, a medida que transcurre el tiempo se irá alejando de la trayectoria deseada producto de esta desviación sistemática. Para que esto no ocurra es necesario que el robot gire paulatinamente para que se acerque a la línea de referencia y avance como lo venía haciendo (derecho) pero por la línea deseada.

Para compensar la desviación se necesita tener en cuenta un error que sea apreciable a lo largo del tiempo. Se introduce un último término medido como una integral del error, que representa la suma de los CTE a lo largo del tiempo. El nuevo controlador considera todos los errores que se mencionaron hasta este punto:

$$\alpha = K_p * CTE + Kd * \frac{dCTE}{dt} + Ki * \text{sum}(CTE)$$

Con el fin de remarcar la importancia de este último término, suponiendo $CTE = 0.8$, la suma se incrementa 0.8 en cada intervalo, por lo que al transcurrir un determinado tiempo se convertirá en un valor importante que antes no se tenía en cuenta.

11.3.5 Sistema de control del Robot

El diseño del sistema de control se muestra a continuación a través de un esquemático:

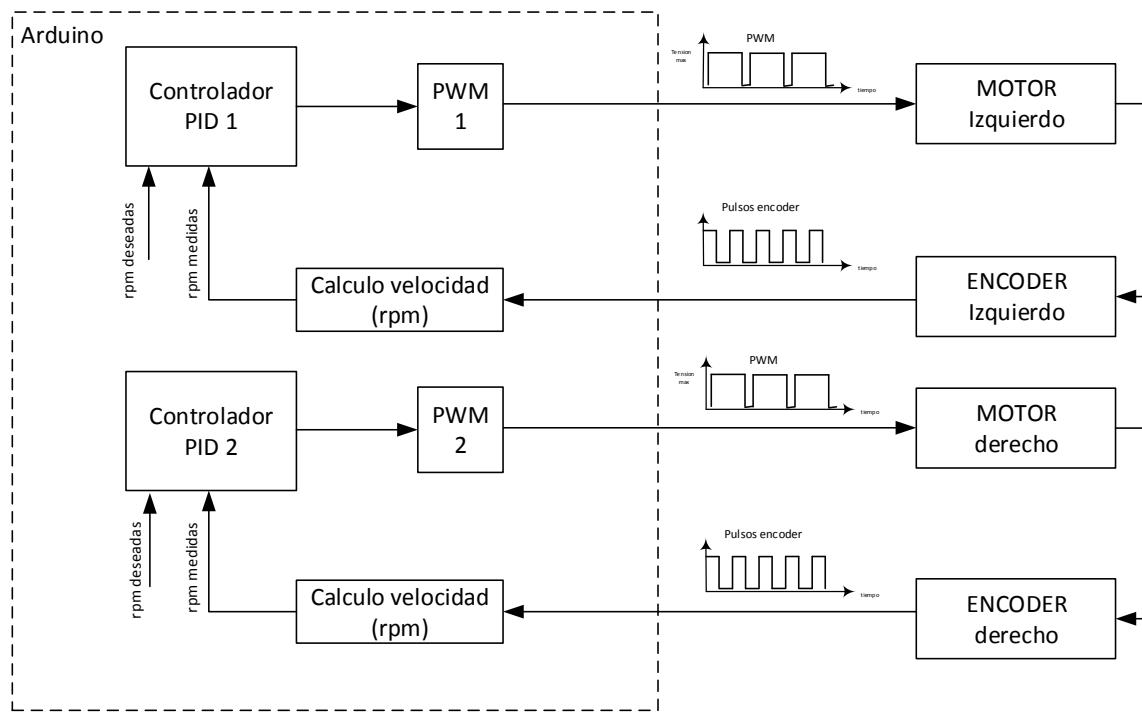


Ilustración 83: Sistema de control del robot para el control de velocidad de cada rueda.

En las siguientes secciones se describen cada uno de los módulos con un mayor detalle.

11.3.5.1 Calculo de velocidad

Este módulo fue implementado en Arduino. Los encoders se conectan a los pines 18 y 19 del Arduino Mega2560 que son establecidos como interrupciones externas. A partir de la función `attachInterrupt()`³² que provee la librería de Arduino, se especifica la función a invocar cuando llega una interrupción. Estas funciones simplemente llevan la cuenta, en variables globales, de la cantidad de pulsos recibidos de cada uno de los encoders.

A continuación se muestra la implementación:

```

1.  /* Las salidas de cada encoder se conectan a Los pines 18 y 19 del Arduino Mega2560 */
2.  #define encodPinA    18
3.  #define encodPinB    19
4.
5.  /* Direccion de Los pines 18 y 19 para LeerLos directamente desde el puerto D de Arduino */
6.  #define PIN18_HIGH  PORTD3
7.  #define PIN19_HIGH  PORTD2
8.
9.  void setup()
10. {
11.     /* Se definen Los pines como entrada */
12.     pinMode(encodPinA1, INPUT);
13.     pinMode(encodPinB1, INPUT);
14.
15.     /* Especifica La función a invocar cuando se produce una interrupción externa en Los
16.      * pines 18 y 19
17.      */
18.     digitalWrite(encodPinA1, HIGH);           // turn on pullup resistor
19.     digitalWrite(encodPinB1, HIGH);
20.     attachInterrupt(4, rencoderB, FALLING);
21.     attachInterrupt(5, rencoderA, FALLING);

```

32 (11-4-13) <http://arduino.cc/es/Reference/AttachInterrupt>

```

22. }
23.
24. /*
25. *   Cada vez que el encoder interrumpe se invoca a esta función que lee el puerto y si el pulso
26. *   es positivo incrementa un contador que lleva la cuenta de la cantidad de pulsos recibidos.
27. */
28. void rencoderA()
29. {
30.     if (PIN18_HIGH)
31.         countA++;
32.     else
33.         countA--;
34. }
35.
36. void rencoderB()
37. {
38.     if (PIN19_HIGH)
39.         countB++;
40.     else
41.         countB--;
42. }
```

La función calcSpeed() calcula la velocidad de cada motor en revoluciones por minuto (RPM) a partir de la cuenta de pulsos de la señal que entrega el encoder. La función que se utiliza para el cálculo se muestra a continuación:

$$\text{Speed[rpm]} = \frac{\Delta \text{encoder } x (60 \times \frac{1000}{\text{LOOPTIME}})}{\text{cantidad pulsos por vuelta}}$$

Siendo: $\text{LOOPTIME} \rightarrow 100\text{ms}$ (frecuencia de llamado a la función)
 Cantidad pulsos por vuelta \rightarrow Valor obtenido por mediciones

El código se muestra a continuación:

```

43. #define LOOPTIME 100      //time in milliseconds
44.
45. void calcSpeed()
46. {
47.     static long countAntA = 0;
48.     static long countAntB = 0;
49.
50.     if(countA!=countAntA)
51.         speed_actA = ((countA - countAntA)*(60*(1000/LOOPTIME)))/4234.378;
52.     countAntA = countA;
53.
54.     if(countB!=countAntB)
55.         speed_actB = ((countB - countAntB)*(60*(1000/LOOPTIME)))/4250.575;
56.     countAntB = countB;
57. }
```

11.3.5.2 Modulación por ancho de pulso PWM

La señal PWM es una onda digital cuadrada, cuya frecuencia es constante, pero la fracción de tiempo en que la señal está en "1" (ciclo de trabajo o duty cycle) puede variar entre 0 y 100%.

Para el caso del robot es necesario proveer a los motores una señal analógica que será la tensión de alimentación de los motores. El PWM permite proveer un voltaje entre el 0 y el 100% del voltaje de la fuente. Así, con esta señal, es posible variar la velocidad de los motores. Ver Ilustración 55³³.

³³(10-4-13) <http://arduino.cc/en/Tutorial/PWM>

Para una tensión máxima de 22.2V, la siguiente tabla muestra el voltaje de salida obtenido, según el valor del dutyCycle que se le pasado como argumento a la función analogWrite:

Valor	Duty cycle	Tensión [V]
64	25%	5.55V
127	50%	11.11V
191	75%	16.65V
255	100%	22.2V

Ilustración 84: Valores de la función analogWrite de Arduino y sus efectos en el PWM

11.3.5.2.1 Modulación de Ancho de Pulso simple con analogWrite³⁴

Arduino facilita el uso de la técnica PWM. La función analogWrite (pin, dutyCycle) es lo único necesario. El parámetro dutyCycle es un valor entre 0 y 255, y pin es uno de los pines que Arduino provee para PWM (3, 5, 6, 9, 10, o 11).

Dicha función es utilizada para darle el valor de tensión deseada a los motores.

11.3.5.3 Controlador PID

El módulo PID fue implementado en Arduino y tiene como entradas la velocidad requerida, set point (valor a alcanzar), y la velocidad estimada a partir de las mediciones mediante los encoders.

La siguiente figura muestra el funcionamiento interno del módulo:

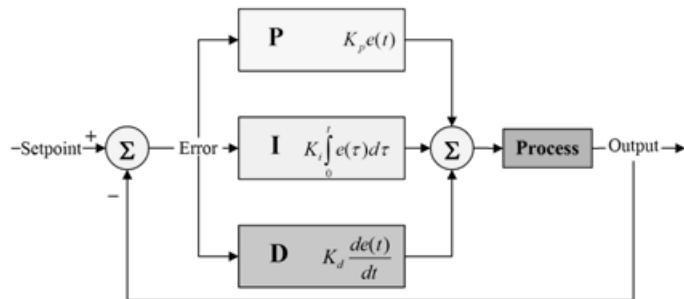


Ilustración 85: Diagrama de bloques general de un controlador PID³⁵

La salida del módulo será el nuevo valor del duty cycle (entre 0 y 255) para la señal PWM que se envía a los motores.

A continuación se muestra la función del controlador. Recibe como argumentos: el valor del duty cycle actual (command), la velocidad deseada (targetValue), la velocidad actual (currentValue), y las constantes del PID. Como salida se obtiene el nuevo valor del duty cycle como suma del actual, más el término calculado por la siguiente ecuación:

34 (10-4-13) <http://arduino.cc/es/Tutorial/SecretsOfArduinoPWM>

35 (10-4-13) <http://quantapublication.files.wordpress.com/2011/02/picture1.png>

$$pid = Kp * error + Kd * \frac{derror}{dt} + Ki * sum(error)$$

Proporcional Derivador Integrador

El “error” es la diferencia entre la velocidad actual y la deseada.

Se puede notar que la salida la devuelve la función constrain³⁶. Esta simplemente verifica que el valor devuelto esté dentro del rango 0-255.

```

58. double accum_error=0;
59. int pid(int command, double targetValue, double currentValue, float Kp, float Kd, float Ki)
60. {
61.     double pidTerm = 0;
62.     double error=0;
63.     static double last_error=0;
64.
65.     error = abs(targetValue) - abs(currentValue);
66.     pidTerm = (Kp * error) + (Kd * (error - last_error)) + (Ki * accum_error);
67.     last_error = error;
68.     accum_error+=error;
69.     return constrain(command + int(pidTerm), 0, 255);
70. }
```

11.3.5.4 Optimización de parámetros

Los parámetros del controlador Kp, Kd, y Ki son fuertemente dependientes del tipo de sistema. Entonces, es necesario conocer los valores adecuados para el robot. Existen dos formas de obtener dichos valores: *manual* y *automática*.

Inicialmente se eligieron los valores ***manualmente*** a partir de numerosas pruebas en las que se fueron variando los 3 parámetros del controlador. La forma de obtenerlos se describe a continuación:

1. Se estableció Kp=Kd=Ki=0.
2. Se incrementó la ganancia Kp hasta obtener una oscilación estable.
3. Se incrementó la ganancia Kd hasta eliminar la oscilación. Se repiten los pasos 2 y 3 hasta que el incremento de Kd ya no elimine las oscilaciones (si las oscilaciones son cada vez mayores, es necesario reducir la ganancia Kp)
4. Se deja en Kp y Kd el último valor estable.
5. Se incrementa la ganancia Ki hasta alcanzar el set point (velocidad deseada) con la menor cantidad de oscilaciones posible (por lo general se busca no tener oscilaciones, aunque se puede tener una respuesta más rápida si se admiten unos pocos sobrepasamientos).

La siguiente figura muestra cómo puede variar la respuesta del robot para diferentes ganancias:

36 (10-4-13) <http://arduino.cc/en/Reference/Constrain>

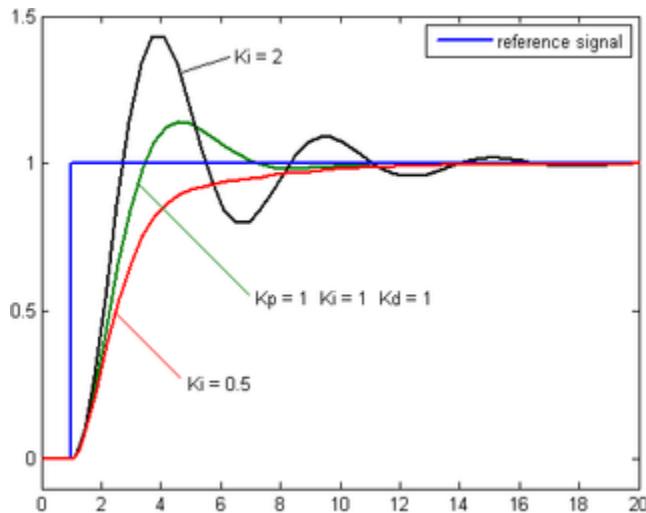


Ilustración 86: Análisis de varias respuestas del controlador PID para distintas constantes³⁷

Para obtener los valores de los parámetros **automáticamente** y optimizarlos uno de los algoritmos más usados se denomina Twiddle (Udacity - Twiddle algorithm, 2012). Este requiere conocer el error promedio (diferencia entre la velocidad deseada y la actual) y busca minimizarlo.

Para implementarlo se han agregado unas líneas de código a la función “pid” descripta en la sección anterior, para poder calcular el error promedio. Además, se debió agregar otro método “run_twiddle” que ejecuta la función pid una cantidad de veces dada por la variable “tiempo”, y al finalizar devuelve el error promedio que se busca minimizar. El algoritmo recibe 3 parámetros que son la “velocidad”, la “tolerancia”, y el “motor” para el que se calculan los parámetros. La tolerancia es un número que determina la condición de terminación del algoritmo. Por ejemplo si se quiere tener un error menor a 0,5 se debe pasar ese valor como tolerancia. Cuando el error promedio es menor a ese valor, el algoritmo termina y devuelve los parámetros calculados. El código se muestra a continuación:

```

71. void twiddle(float _speed, float tolerance, int motor)
72. {
73.     float p[3]={0,0,0};                                //pid params
74.     float dp[3]={1,1,1};
75.     int tiempo=300;
76.     best_error=run_twiddle(_speed, p[0], p[1], p[2], p[0], p[1], p[2], 0, tiempo, false, motor);
77.
78.     while ( ( (dp[0]+dp[1]+dp[2]) > tolerance ) )
79.     {
80.         for(int i=0; i<3; i++)
81.         {
82.             p[i]=p[i]+dp[i];
83.             error_prom=run_twiddle(_speed, p[0], p[1], p[2], p[0], p[1], p[2], 0, tiempo, false, motor);
84.
85.             if(error_prom < best_error && error_prom > 0)
86.             {
87.                 best_error=error_prom;
88.                 dp[i]=dp[i]*1.1;
89.             }
90.             else
91.             {
92.                 p[i]=p[i]-2*dp[i];
93.                 error_prom=run_twiddle(_speed, p[0], p[1], p[2], p[0], p[1], p[2], 0, tiempo, false, motor);
94.
95.                 if (error_prom < best_error && error_prom > 0)
96.                 {
97.                     best_error=error_prom;

```

³⁷ (12-4-13) Imagen extraída de http://upload.wikimedia.org/wikipedia/commons/thumb/c/c0/Change_with_Ki.png/320px-Change_with_Ki.png

```

98.           dp[i]=dp[i]*1.1;
99.       }
100.      else
101.      {
102.          p[i]=p[i]+dp[i];
103.          dp[i]=dp[i]*0.9;
104.      }
105.  }
106. }
107. }
108. }
```

En la sección 11.3.5.7 se describen los casos de test llevados a cabo para cada caso.

11.3.5.5 Cálculo de cantidad de pulsos por revolución

La cantidad de pulsos por revolución está dada por la siguiente ecuación:

$$velocidad[rpm] = \frac{\Delta pulsosEncoder * 60 * \frac{1000}{LOOPTIME[ms]}}{cantidadPulsosPorVuelta}$$

La variable $\Delta pulsosEncoder$ se obtiene por consola, el LOOPTIME es el período de muestreo, y las rpm se pueden medir con un tacómetro digital, por lo que despejando la variable cantidadPulsosPorVuelta se tiene el valor deseado:

$$cantidadPulsosPorVuelta = \frac{\Delta pulsosEncoder * 60 * \frac{1000}{LOOPTIME[ms]}}{velocidad[rpm]}$$

Se realizó un test con una batería Li-Po cuyo nivel de carga era 22.2V, donde a ambos motores se les dio la máxima velocidad posible (duty cycle de 255) y se obtuvieron muestras cada 1000ms de la cantidad de pulsos que devuelven los encoder. Al mismo tiempo se midieron las revoluciones por minuto de cada uno de los motores con un tacómetro digital. Los resultados del test son:

Mediciones tacómetro digital	
RPM izquierdo	22,2
RPM derecho	22,6

Estadísticas (Mediciones promedio tomadas)	
PWM promedio izquierdo	255
PWM promedio derecho	255
Promedio diferencias encoder izquierdo	1566,73 pulsos
Promedio diferencias encoder derecho	1601,06 pulsos

Entonces queda:

$$cantidadPulsosPorVuelta = \frac{\Delta pulsosEncoder * 60 * \frac{1000}{LOOPTIME[ms]}}{velocidad[rpm]}$$

Los resultados obtenidos son:

$$cantidadPulsosPorVuelta_{motorIzquierdo} = \frac{1566,73 * 60 * \frac{1000}{1000}}{22,2} = 4234,37$$

$$cantidadPulsosPorVuelta_{motorDerecho} = \frac{1601,06 * 60 * \frac{1000}{1000}}{22,6} = 4250,57$$

Como se puede apreciar, son valores muy grandes, lo que implica una resolución muy alta de los encoders. Como el diámetro de la rueda es:

- Diámetro de cada rueda: 10 cm.
- Circunferencia: $10 \times \pi = 31.415926$ cm.

Es decir, si la circunferencia es de 31,415926 cm, cada pulso implica una distancia recorrida de

$$cmPorPulso_{motorIzquierdo} = \frac{\text{Circunferencia}}{\text{cantidadPulsosPorVuelta}} = 0,007419\text{cm}$$

Este valor permite obtener la cantidad avanzada del robot por pulso de encoder del motor izquierdo. El mismo cálculo se realiza para el encoder derecho.

11.3.5.6 Test de velocidad máxima

Se confeccionó una planilla de test que permite obtener datos estadísticos del funcionamiento del robot y gráficas del control del robot simplemente importando los datos que se obtienen por consola en la ejecución.

En cada test se registran los siguientes datos:

- Número de test
- Fecha del test
- Valores de los parámetros Kp, Kd, Ki
- Tipo de batería usada durante el test
- Nivel de voltaje de la batería
- Cantidad de muestras tomadas
- Período de muestreo (LOOPTIME)

En la tabla se hace una estadística de:

- Velocidad [rpm] promedio para ambos motores
- Valor del duty cycle promedio
- Promedio de diferencia en cantidad de pulsos obtenidas en cada muestra de cada encoder

Se generan gráficos de:

- Respuesta del PID para cada motor medida en RPM y valores que toma el PWM.
- Velocidad en RPM para cada motor
- Superposición de las curvas de velocidad de ambos motores para comparar sus ajustes de velocidad.

Robot móvil autónomo capaz de producir mapas 3D del interior de una habitación

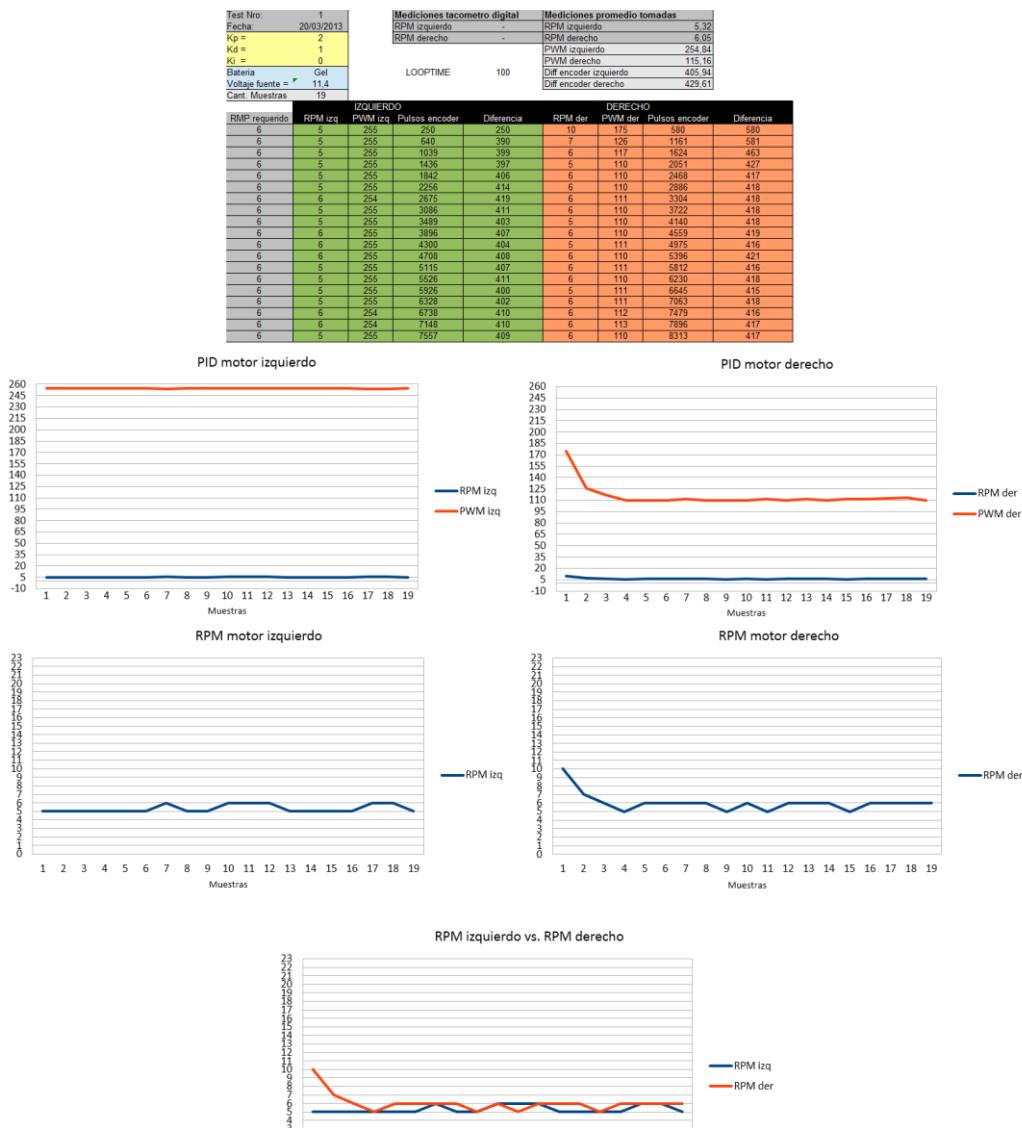


Ilustración 87: Planilla donde se realizan los test del PID

Inicialmente se realizó un **test de cálculo de velocidad máxima** a la que puede llegar el robot con una batería Li-Po con un nivel de tensión de 22,9V. Para ello se realizó una ejecución para el comando up, y se obtuvieron los siguientes resultados:

Estadísticas (Mediciones promedio tomadas)	
RPM promedio izquierdo	20,91 rpm
RPM promedio derecho	20,89 rpm
PWM promedio izquierdo	255
PWM promedio derecho	252
Promedio diferencias encoder izquierdo	740,31 pulsos
Promedio diferencias encoder derecho	741,85 pulsos
Mediciones tacómetro digital (en régimen)	
RPM izquierdo	21,0 rpm
RPM derecho	21,0 rpm

Ilustración 88: Mediciones de la velocidad de las ruedas

Del análisis estadístico, el dato más relevante obtenido es el promedio de velocidad de cada motor, medida en rpm. Ambos motores en promedio tuvieron prácticamente la misma velocidad lo que significa que el avance del robot fue el correcto y en linea recta. La velocidad máxima obtenida para 22,9V es de 20,9 rpm. Adicionalmente, se puede apreciar el valor del PWM promedio de cada motor para alcanzar dicha velocidad, y el promedio de diferencia de pulsos entre cada muestra de los encoders.

Por otro lado, se realizó una medición con un tacómetro digital para medir la velocidad “real” de cada motor y se obtuvieron valores muy similares a las calculadas (21 rpm).

Junto con el análisis estadístico se generan gráficos que muestran la respuesta del controlador para cada motor. En los dos graficos de abajo, se aprecia la velocidad de cada motor y el valor que fue tomando el PWM que en este caso como es el valor máximo es constante (recordemos que se está midiendo la velocidad máxima de cada motor).

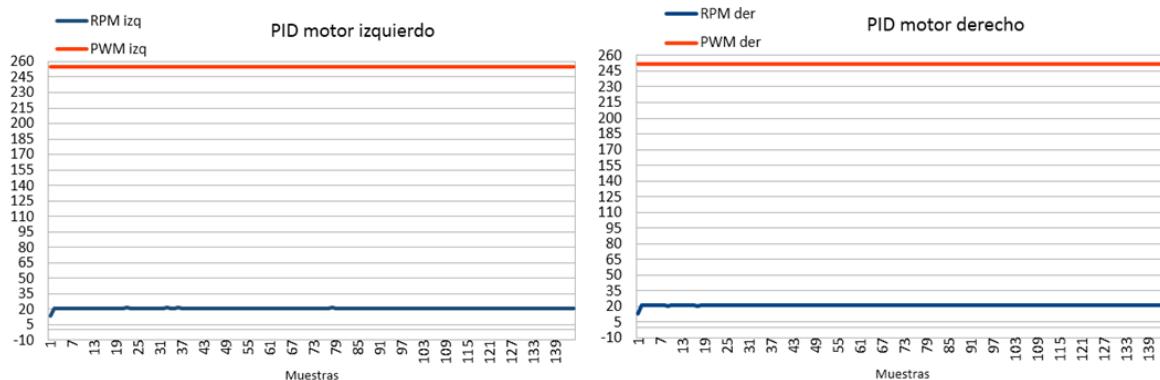


Ilustración 89: Resultado de velocidad de motores para un dutyCycle del %100 (máxima velocidad)

Estos gráficos son el mismo de arriba pero con otra escala que permite ver más de cerca y en detalle la respuesta de velocidad de cada motor. Se puede apreciar que, una vez que llega al régimen, se mantiene en un valor constante (velocidad máxima) con pequeñas oscilaciones sobre el valor de régimen.

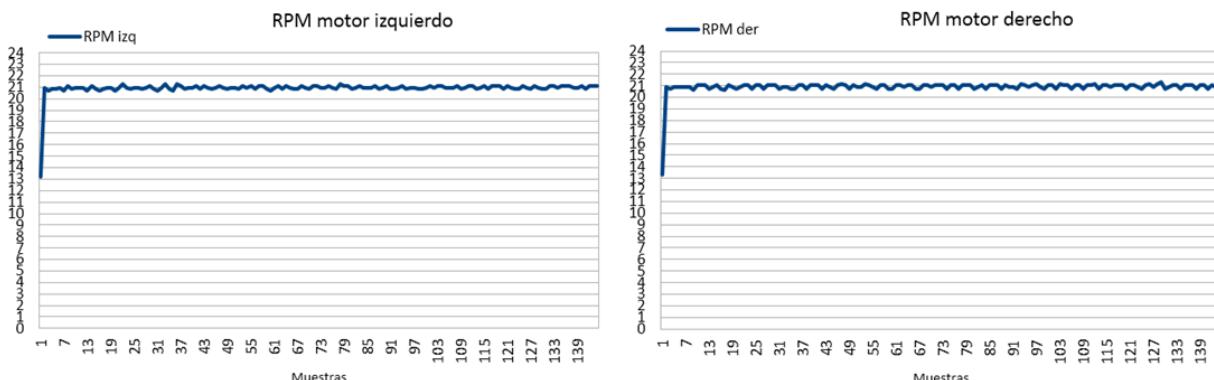


Ilustración 90: Velocidad de los motores

El siguiente gráfico muestra ambas respuestas superpuestas lo que deja en evidencia que ambos motores están ajustados a la misma velocidad por lo que el robot avanzará en línea recta.

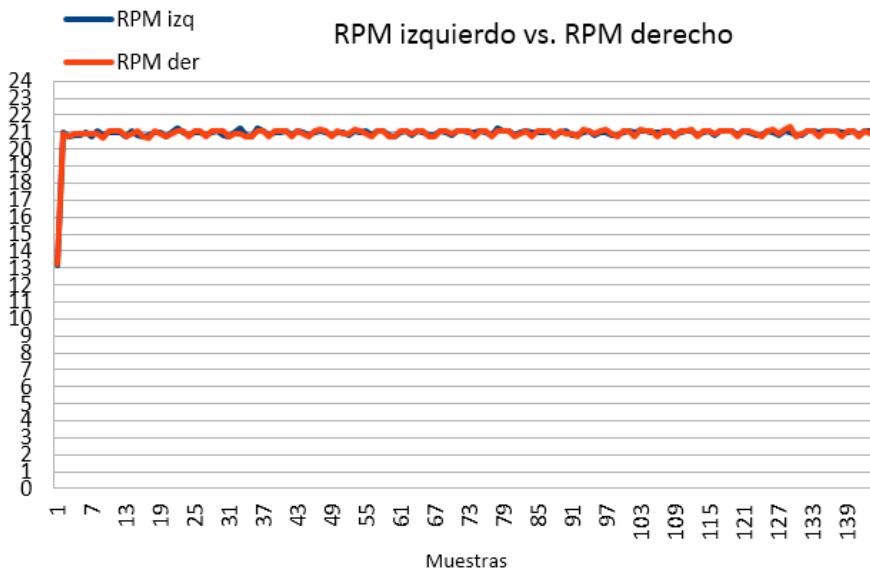


Ilustración 91: Diferencias de velocidad entre ambos motores

11.3.5.7 Test de control del robot

A continuación se muestran 7 test realizados para probar cómo se comporta el controlador para cada comando del robot (up, down, right, left) y para diferentes velocidades 5rpm, 7.5rpm, 10rpm, 12.5rpm, 15rpm, 17.5rpm, y 20rpm.

TEST	
Requerimiento	FSR3) Control de trayectoria y velocidad
Prueba	ST3.1) Ajuste de velocidad del robot para todos los movimientos posibles: Se debe probar la reacción del robot algunas velocidades incluyendo los casos extremos, y a su vez esto para cada uno de los comandos.
Precondición	Robot alimentado por batería cargada Aplicación ejecutándose en Arduino Aplicación ejecutándose en la computadora a bordo ROS ejecutándose
Poscondición	Mismas que las iniciales
Resultado obtenido	El robot se desplaza sin desvíos significativos (dentro del error aceptable)
Resultado esperado	El robot se desplaza sin desvíos significativos (dentro del error aceptable)
Conclusión	PASADO

Tabla 13: Tarjeta de testing FSR3 ST3.1

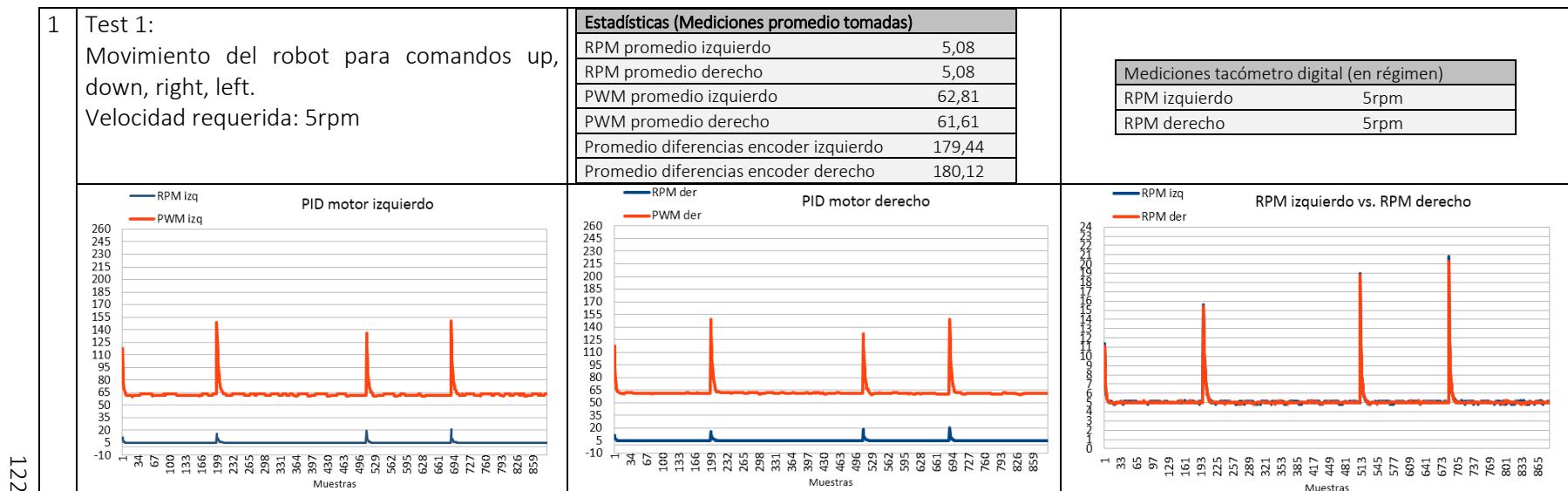
Las pruebas se realizaron con una batería Li-Po cuyo nivel de carga al momento de realizar la prueba fue de 22.9V. Además los parámetros del controlador Kp, Kd, Ki usados fueron 2.5, 3.5, 0.962 respectivamente. La función pid fue ejecutada con un período de 10ms (dado por la variable LOOPTIME), mientras que las muestras se tomaron cada 500ms (son valores que se obtienen desde la consola cuyo período está dado para la variable PRINTTIME).

Kp =	2.5
Kd =	3.5
Ki =	0.962
Bateria	Li-Po
Voltaje =	22.9

Período ajuste (LOOPTIME)	10ms
Periodo muestreo (PRINTTIME)	500ms

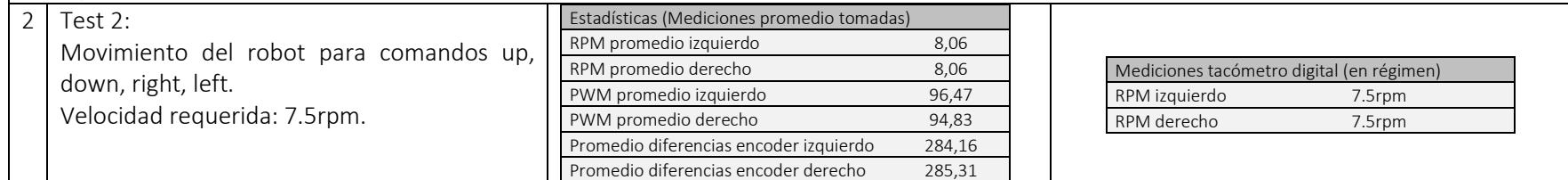
En cada gráfico se tienen las respuestas para los comandos up, down, right y left. En cada comando que se le envía al robot el PWM toma un valor inicial de 255 (valor máximo). Esto explica los picos para cada cambio de movimiento que se ven en cada uno de los 3 gráficos de cada test.

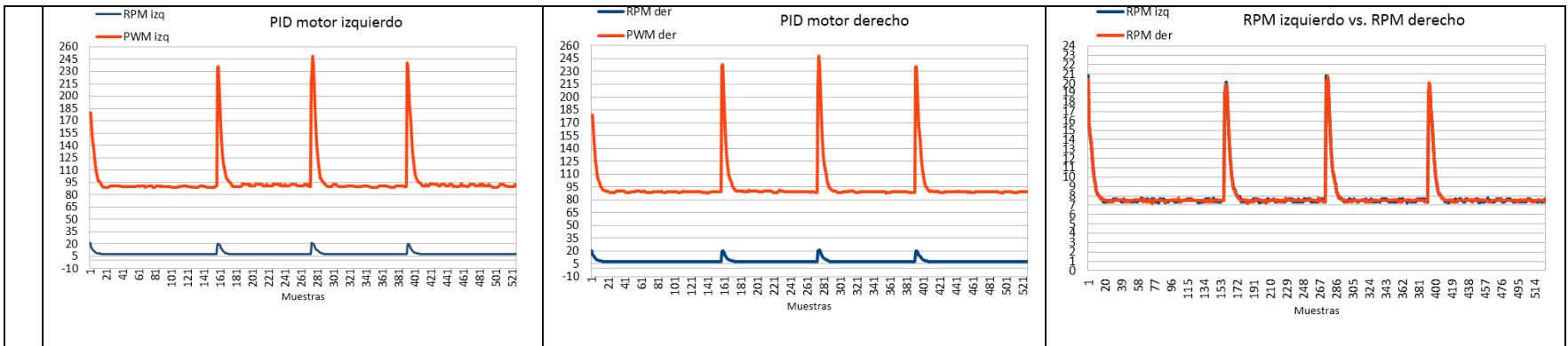
Los resultados de las pruebas se presentan en la siguiente tabla:



122

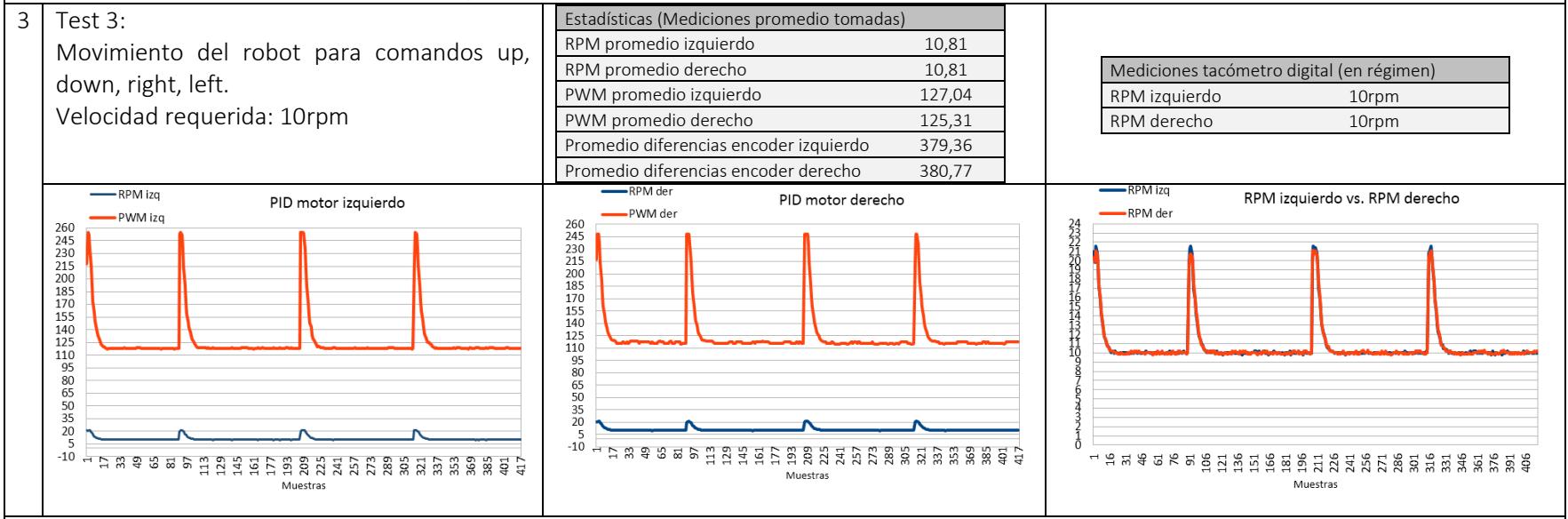
Análisis de resultados: Es importante destacar que la velocidad promedio es la misma en cada motor. Ésta es superior a la velocidad requerida en una cantidad muy pequeña 0,08rpm, como consecuencia de que en el cálculo del promedio se incluyen valores de velocidad de tramos donde el sistema todavía no llegó al régimen. Es importante destacar que el tiempo de respuesta es de unos pocos milisegundos (casi instantáneo). La medición tomada con el tacómetro demuestra que el robot avanza exactamente a la velocidad requerida (medida tomada cuando el sistema entra en régimen).



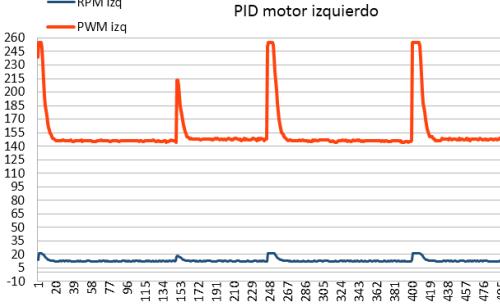
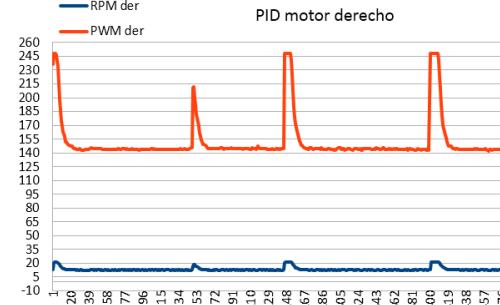
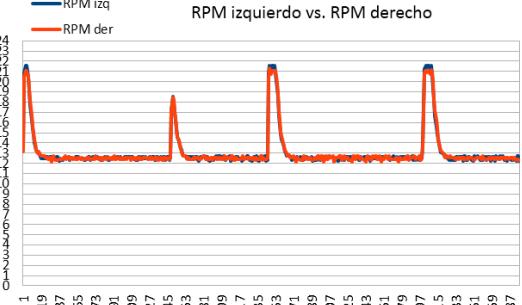
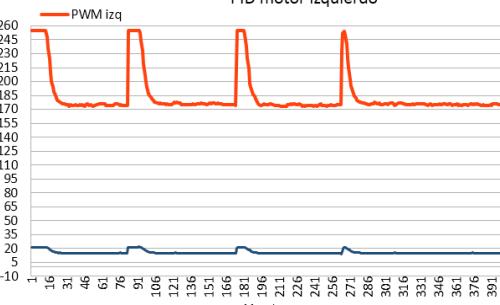
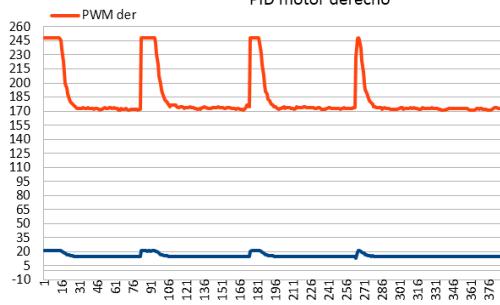
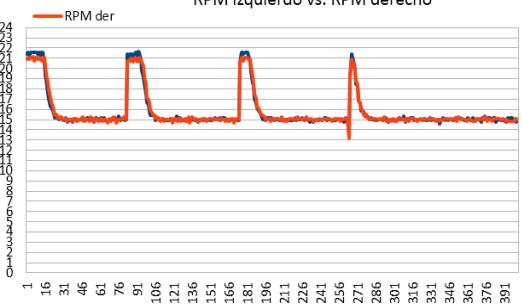


Análisis de resultados: Se tienen 4 respuestas en cada gráfico para cada uno de los comandos ejecutados. La velocidad promedio obtenida es 8.06 rpm y si bien tiene una diferencia de 0,56rpm respecto de la velocidad deseada, lo importante es que ambos motores tienen la misma velocidad. El tiempo que tarda el sistema en entrar en régimen sigue siendo muy pequeño aunque es ligeramente mayor respecto al del test 1. La velocidad medida con el tacómetro es exactamente igual a la requerida.

123

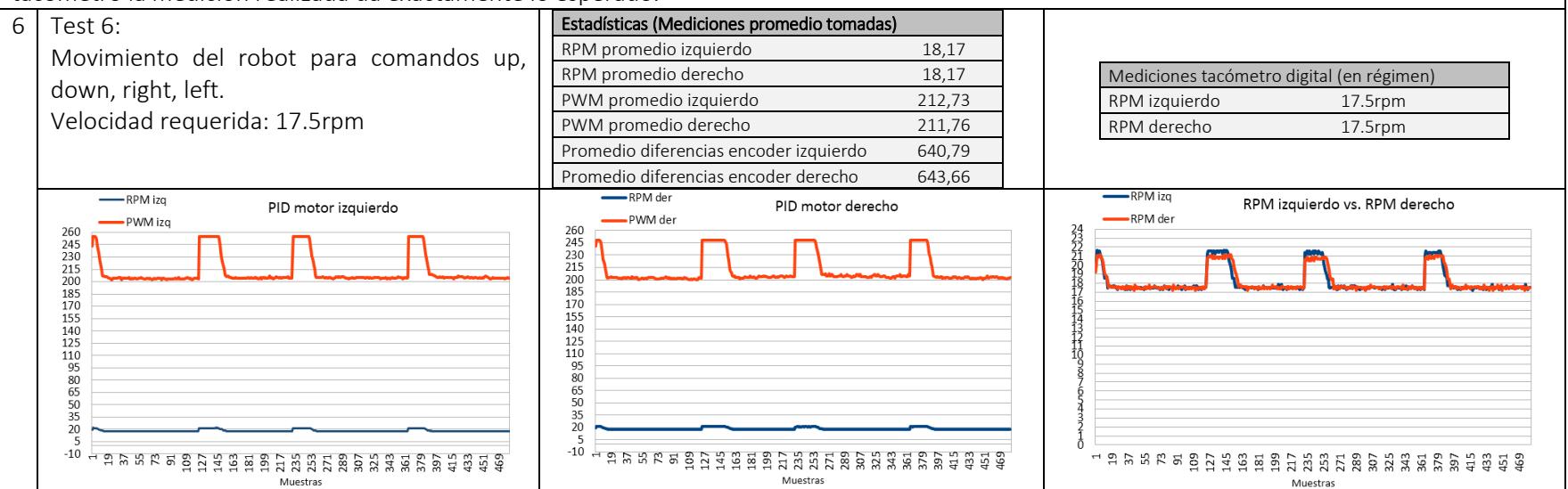


Análisis de resultados: Se pueden apreciar las respuestas para cada comando. Todas las respuestas son idénticas. Una vez que el sistema alcanza el régimen se establece en la velocidad deseada (10 rpm). Esto se comprobó con las medidas tomadas con el tacómetro para cada motor que dan exactamente 10 rpm. El promedio de velocidad es 0,81 rpm superior a la velocidad deseada. Esto se obtiene como consecuencia del tiempo que

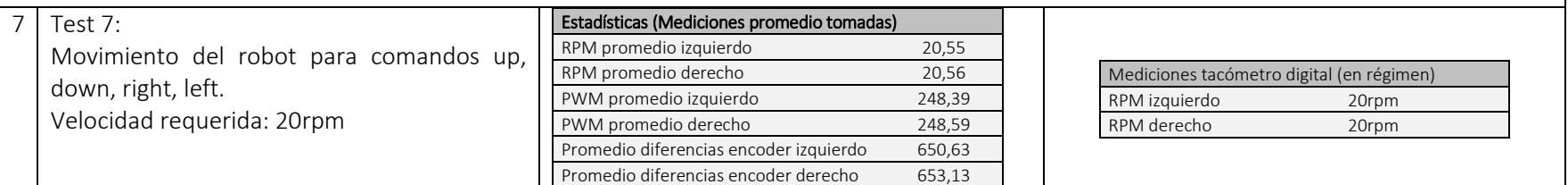
		tarda el sistema en estabilizarse a la velocidad deseada. Lo importante es que ambos motores tienen siempre la misma velocidad.																	
4	Test 4: Movimiento del robot para comandos up, down, right, left. Velocidad requerida: 12.5rpm	<p>Estadísticas (Mediciones promedio tomadas)</p> <table border="1"> <tbody> <tr><td>RPM promedio izquierdo</td><td>13,13</td></tr> <tr><td>RPM promedio derecho</td><td>13,13</td></tr> <tr><td>PWM promedio izquierdo</td><td>154,06</td></tr> <tr><td>PWM promedio derecho</td><td>152,02</td></tr> <tr><td>Promedio diferencias encoder izquierdo</td><td>462,41</td></tr> <tr><td>Promedio diferencias encoder derecho</td><td>464,19</td></tr> </tbody> </table> <p>Mediciones tacómetro digital (en régimen)</p> <table border="1"> <tbody> <tr><td>RPM izquierdo</td><td>12.5rpm</td></tr> <tr><td>RPM derecho</td><td>12.5rpm</td></tr> </tbody> </table>	RPM promedio izquierdo	13,13	RPM promedio derecho	13,13	PWM promedio izquierdo	154,06	PWM promedio derecho	152,02	Promedio diferencias encoder izquierdo	462,41	Promedio diferencias encoder derecho	464,19	RPM izquierdo	12.5rpm	RPM derecho	12.5rpm	
RPM promedio izquierdo	13,13																		
RPM promedio derecho	13,13																		
PWM promedio izquierdo	154,06																		
PWM promedio derecho	152,02																		
Promedio diferencias encoder izquierdo	462,41																		
Promedio diferencias encoder derecho	464,19																		
RPM izquierdo	12.5rpm																		
RPM derecho	12.5rpm																		
124		  																	
	<i>Análisis de resultados:</i> Para esta prueba el error de la velocidad promedio es de 0.63 rpm respecto a la velocidad requerida dado por el tiempo de respuesta. La velocidad en régimen es exactamente la esperada con pequeñas oscilaciones, y la velocidad promedio de ambos motores es exactamente la misma (ésta se puede apreciar en el tercer gráfico). Las medidas tomadas con el tacómetro dan exactamente 12,5rpm.																		
5	Test 5: Movimiento del robot para comandos up, down, right, left. Velocidad requerida: 15rpm	<p>Estadísticas (Mediciones promedio tomadas)</p> <table border="1"> <tbody> <tr><td>RPM promedio izquierdo</td><td>15,80</td></tr> <tr><td>RPM promedio derecho</td><td>15,80</td></tr> <tr><td>PWM promedio izquierdo</td><td>184,93</td></tr> <tr><td>PWM promedio derecho</td><td>183,06</td></tr> <tr><td>Promedio diferencias encoder izquierdo</td><td>558,41</td></tr> <tr><td>Promedio diferencias encoder derecho</td><td>559,76</td></tr> </tbody> </table> <p>Mediciones tacómetro digital (en régimen)</p> <table border="1"> <tbody> <tr><td>RPM izquierdo</td><td>15rpm</td></tr> <tr><td>RPM derecho</td><td>15rpm</td></tr> </tbody> </table>	RPM promedio izquierdo	15,80	RPM promedio derecho	15,80	PWM promedio izquierdo	184,93	PWM promedio derecho	183,06	Promedio diferencias encoder izquierdo	558,41	Promedio diferencias encoder derecho	559,76	RPM izquierdo	15rpm	RPM derecho	15rpm	
RPM promedio izquierdo	15,80																		
RPM promedio derecho	15,80																		
PWM promedio izquierdo	184,93																		
PWM promedio derecho	183,06																		
Promedio diferencias encoder izquierdo	558,41																		
Promedio diferencias encoder derecho	559,76																		
RPM izquierdo	15rpm																		
RPM derecho	15rpm																		
		  																	

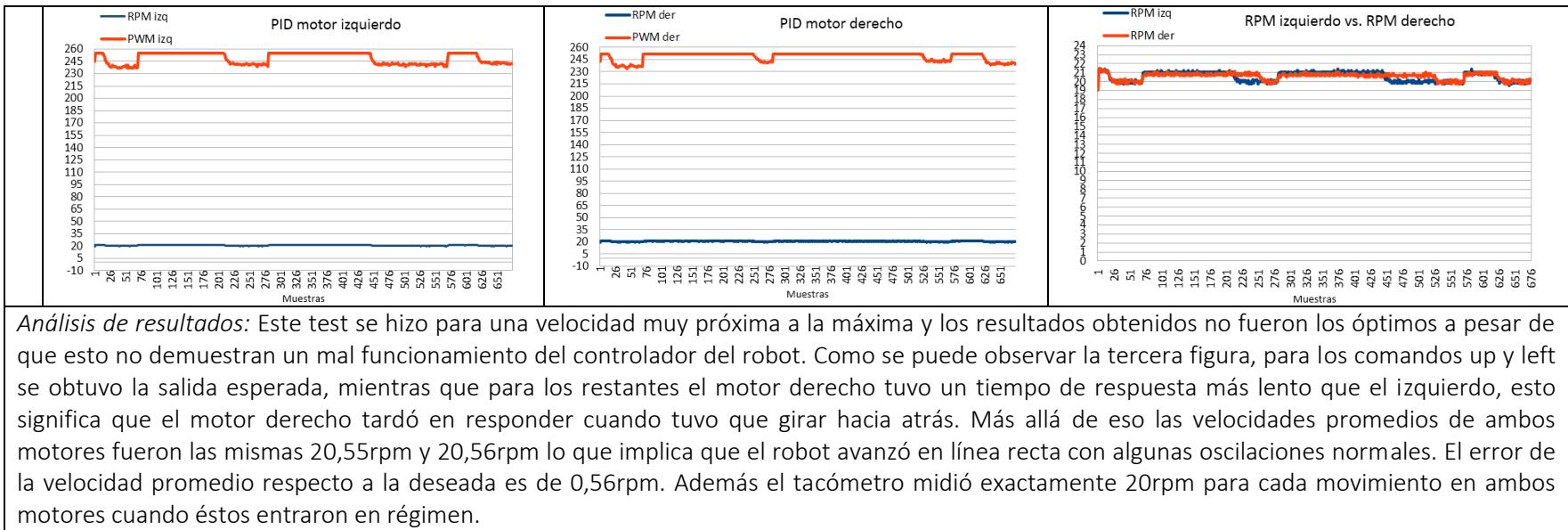
Análisis de resultados: Incrementando la velocidad se prueba que ambos motores siguen ajustándose al mismo tiempo lo que implica que el robot avanza en línea recta. Para este caso el promedio de velocidad de ambos motores es exactamente la misma 15,80rpm la cual difiere en 0,80rpm respecto de la velocidad deseada. Como se explicó anteriormente esto se debe a que se está promediando el tramo donde el sistema todavía no está en régimen. En este test ya se comienza a notar un menor tiempo de respuesta, es decir, el robot tarda más tiempo en alcanzar la velocidad deseada y entrar en régimen. Eso es simplemente porque la velocidad requerida es mayor y se tarda más tiempo en alcanzar el set point. Con el tacómetro la medición realizada da exactamente lo esperado.

125



Análisis de resultados: Este es el anteúltimo test y es en el que más tiempo se tarda en alcanzar el valor de régimen. En el tercer gráfico se puede apreciar también un leve desfasaje en la velocidad de los motores para los comandos down y right, pero son poco significativos y no influyen en el resultado obtenido. Al igual que todos los test anteriores hay un error de la velocidad promedio respecto de la deseada, de 0,67rpm pero ambos motores tienen la misma velocidad por lo cual el robot realiza sus movimientos correctamente. Las medidas tomadas con el tacómetro dan exactas.





11.3.5.8 Resultado de cálculo de Twiddle para optimización de parámetros

El algoritmo se ejecutó para calcular los valores de cada motor por separado con una velocidad dada de 20rpm y un nivel de tensión de la batería de 23,5V. El valor de tolerancia pasado como argumento fue de 0,2 lo que significa que se busca tener un error menor a esa cifra.

Los resultados obtenidos para cada motor fueron:

- Motor izquierdo
 - $K_p=9,88$
 - $K_d=5,84$
 - $K_i=1,43$
- Motor derecho
 - $K_p=2,70$
 - $K_d=3,31$
 - $K_i=4,64$

Si bien estos valores de ganancia permiten un buen ajuste comprobado y un error muy pequeño, al probarlos sobre los movimientos del robot, éste se aceleraba y desaceleraba constantemente dado a que los valores de K_i obtenidos son demasiado elevados (se podían observar los sobre pasamientos en el movimiento). Debido a este comportamiento, se decidió continuar con el método de optimización de parámetros manual que dio muy buenos resultados, y se plantea continuar trabajando sobre este algoritmo para la obtención de parámetros como un trabajo de mejora a futuro.

11.3.5.9 Conclusión

Como conclusión general del control del robot, en todos los test realizados para diferentes velocidades dadas como set point, ambos motores ajustaron exactamente igual y tuvieron la misma velocidad promedio. Con el tacómetro se comprobó que las velocidades obtenidas son reales y están correctamente calculadas. Algo que se puede apreciar es que a medida que se le exigen velocidades de avance más grandes al robot, el tiempo de respuesta empeora por lo que es importante destacar que el mejor funcionamiento se obtiene para velocidades más pequeñas. Veremos en capítulos posteriores que para la creación de mapas es necesario tener un muy buen control de robot a bajas velocidades.

Por otro lado, es importante mencionar que esta fue una de las tareas más complicadas del proyecto, esto se debió a que si bien ha quedado comprobado el correcto funcionamiento del controlador, el robot cuenta con algunos problemas que hacen que se desvíe por problemas puramente relacionados a la mecánica y totalmente ajenos al controlador. Uno de ellos es la rueda frontal elegida para la construcción.

Robot móvil autónomo capaz de producir mapas 3D del interior de una habitación

11.4 Localización

11.4.1 Posición del robot

Al introducir al robot en el mundo real, es necesario que este conozca su posición en cada momento. Para ello se debe desarrollar un sistema de localización con una precisión en centímetros.

Inicialmente se lo introduce en un sistema de coordenadas donde su estado queda definido por los valores (x, y, Φ) , que representan los desplazamientos en cada eje y la orientación del robot. Ver Ilustración 14: Muestra el posicionamiento del robot en un Sistema de coordenadas.

En el caso que se deseé el avance de cierta distancia, se calcula la distancia euclídea desde el punto inicial hasta el punto final:

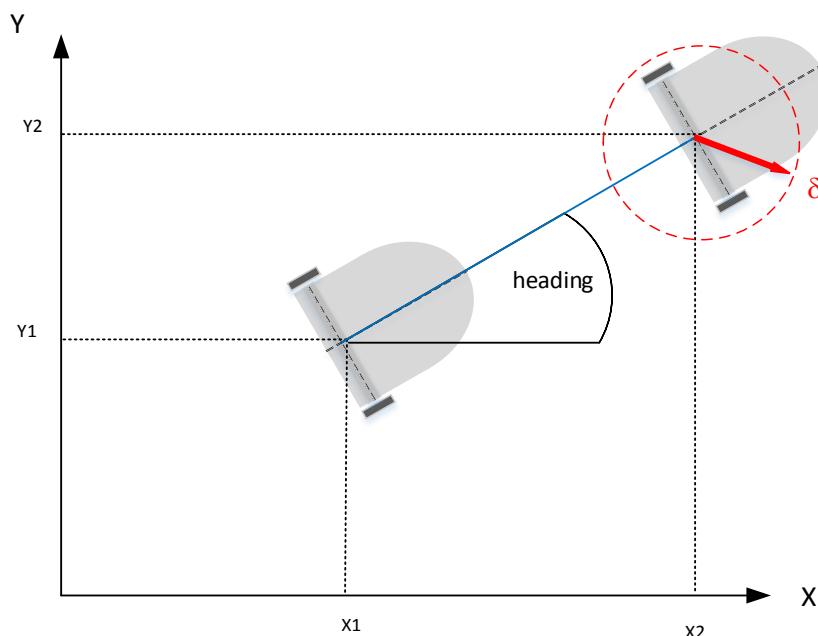


Ilustración 92: Localización del robot

La distancia entre el punto (x_1, y_1) y el punto (x_2, y_2) está dada por el teorema de Pitágoras:

$$ds^2 = dx^2 + dy^2$$

Debido a desviaciones, es posible que el robot no pase exactamente por el punto (x_2, y_2) ya, entonces se admite un error denominado delta δ cuyo valor se medirá luego. Para controlar si el robot ha llegado a destino, se utiliza la siguiente desigualdad:

$$dx^2 + dy^2 \leq \delta^2$$

La ecuación se puede visualizar como un círculo de radio delta, que delimita una zona alrededor del objetivo dentro la cual el error es admisible. En la imagen, esta zona se muestra

con una circunferencia roja. Si el robot se encuentra dentro de esta, se considera que ha alcanzado el objetivo.

11.4.2 Implementación de la Localización

La localización del robot se implementó en el micro controlador Arduino, es decir a bajo nivel, con el objetivo de que una vez enviado un comando de movimiento indicando X, Y ó ϕ , se verifica periódicamente si se ha alcanzado esa distancia o ángulo.

Los comandos que se le envían al controlador, pueden ser:

- Up()
- Down()
- Right()
- Left()

Tal como se muestra, los comandos de movimiento lineal producirán un avance o retroceso de 10cm por defecto. Pero para darle una mayor versatilidad, a dichos comandos se les puede enviar la cantidad (en cm) que se desea que el robot se desplace. Por ejemplo, se le puede enviar el comando up(25) y el robot avanzará una distancia de 25cm según la orientación actual.

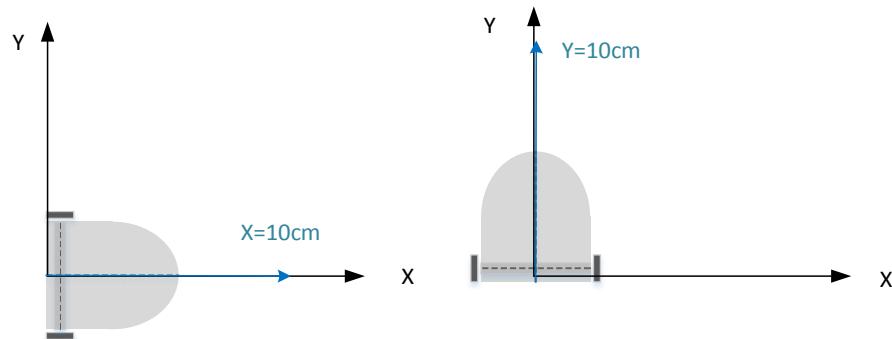


Ilustración 93: Posibles avances del robot según su orientación

Por otro lado, los comandos de movimiento angular, por defecto, harán que el robot gire un ángulo de 90 relativos a la orientación actual. También es posible pasar como argumento la cantidad en grados que se desea girar. Por ejemplo, si se envía el comando right(75), el robot girará 75 grados hacia la derecha.

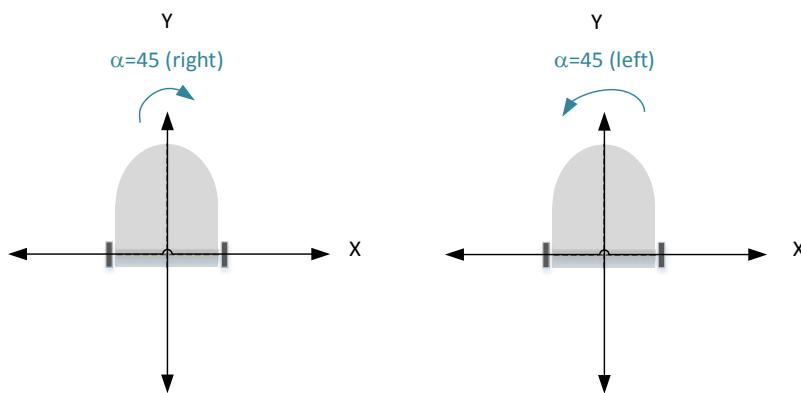


Ilustración 94: Sentido de giro para los comandos right y left

Cada vez que se ejecuta un comando, luego se llama periódicamente a la función position(), encargada de calcular el avance realizado por el robot desde la última vez que se ejecutó dicha función.

Para el movimiento lineal, se lee la diferencia de pulsos que entregan los encoders de ambos motores. Luego, se hace el producto de esa cantidad de pulsos, por una constante llamada “centímetros por pulso”. Dicha constante fue obtenida experimentalmente e indica la relación entre la cantidad de pulsos de encoder y los centímetros avanzados. Finalmente, se hace un promedio entre ambas lecturas para obtener la distancia avanzada.

Por otro lado, se mide la variación de la orientación respecto de la última llamada a la función. Dicha medida es proveída por el giróscopo.

A partir de la distancia y el ángulo, se calculan las componentes dx e dy, y dichos valores se suman a la posición actual (x,y) del robot, actualizando de esta forma su posición. Además, se actualiza el “heading” del robot con el ángulo actual.

El código implementado en C++ se muestra a continuación:

```

17. #define cm_pulsoA 0.007419      // diámetro * pi / pulsos por vuelta -> 10*pi/4234.378
18. #define cm_pulsoB 0.007390      // diámetro * pi / pulsos por vuelta -> 10*pi/4250.575
19.
20. void position()
21. {
22.     volatile double dist=( (countA-pulsosAntA) * cm_pulsoA + (countB-pulsosAntB) * cm_pulsoB ) / 2;
23.     pulsosAntA=countA;
24.     pulsosAntB=countB;
25.
26.     g.gyroGetData();
27.     heading = g.getLastYaw() - headingBase;
28.     if( heading < 0 ) heading=heading+360;
29.
30.     float headingRad=heading*2*PI/360;
31.
32.     volatile double y1=dist*sin(headingRad);
33.     volatile double x1=dist*cos(headingRad);
34.
35.     switch(current_state)
36.     {
37.         case UP:
38.             x+=x1;
39.             y+=y1;
40.             break;
41.         case DOWN:
42.             x-=x1;
43.             y-=y1;
44.             break;
45.         default:
46.             break;
47.     }
48. }
```

Dado que en cada comando que se envía al robot se pasa como argumento una cantidad máxima de avance [cm] o giro [grados], es necesario verificar periódicamente si el robot ya ha alcanzado dicha posición. Para ello, se implementó una función llamada checkPosition() la cual comprueba dos condiciones.

1)En primera instancia, se comprueba que tipo de comando se recibió; en caso de corresponder a un movimiento de giro, se chequea si ya se alcanzó el valor deseado a partir de la llamada a una función isInBounds() que comprueba si el valor actual está dentro de un rango deseado [ángulo deseado, ángulo deseado +1].

2)Por otro lado, si se trata de un comando de movimiento lineal, debe comprobar si se cumple la desigualdad $dx2+dy2 \leq \delta$.

Para ambos casos, si la comprobación resulta afirmativa, se procede a buscar el próximo movimiento, en una cola de comandos. En dicha cola, el programa implementado en Arduino almacena todos los movimientos que llegan desde la PC. Además, se imprime en consola, la posición alcanzada. En caso contrario, se espera a que alguna de las condiciones se cumpla.

La implementación de la función se muestra abajo:

```

49. void checkPosition(QueueList<String> &q)
50. {
51.     if(current_state==STOP)
52.         nextCommand(q);
53.     else
54.     {
55.         /*
56.          *      Check limits for steering movement
57.          */
58.         if( isInBounds(heading, max_heading, max_heading + 1) )
59.         {
60.             printPositionInfo();
61.             nextCommand(q);
62.         }
63.         else
64.         {
65.             /*
66.              *      Check limits for linear movement
67.              */
68.             float delta = 0.09;
69.             if( (((max_y-getY())*(max_y-getY()))+((max_x-getX())*(max_x-getX()))) <= delta)
70.             {
71.                 printPositionInfo();
72.                 nextCommand(q);
73.             }
74.         }
75.     }
76. }
```

11.4.3 Test de localización

Para medir la precisión del método checkPosition(), a la hora de detener el robot en una posición dada, se realizaron varios tests para medir el estado final (x, y, orientación).

TEST	
Requerimiento	FSR1) Movimientos del robot a partir de comandos básicos
Prueba	ST1) Prueba de cada comando por separado: para cada uno de los comandos, que el robot se desplace como es de esperarse.
Precondición	Robot alimentado por batería cargada Aplicación ejecutándose en Arduino Aplicación ejecutándose en la computadora a bordo ROS ejecutándose

	Robot en posición inicial X0,Y0
Poscondición	Nueva posición del robot X1,Y1 y todo lo demás sin cambios
Resultado obtenido	El robot se encuentra en una nueva posición X1,Y1
Resultado esperado	El robot debe encontrarse en una nueva posición X1,Y1
Conclusión	PASADO

Tabla 14: Tarjeta de testing FSR1 ST1

11.4.3.1 Movimientos de giro, right(), left()

Se realizaron pruebas para comprobar la precisión en cuanto a ángulos de giro en direcciones dadas por los comandos right() y left() y valores entre 0-360.

TEST	
Requerimiento	FSR2) Localización del robot
Prueba	ST2.1) Comprobar orientación del robot para movimientos angulares: probar los comandos right() y left() para diferentes ángulos y medir el desplazamiento respecto del punto inicial.
Precondición	Robot alimentado por batería cargada Aplicación ejecutándose en Arduino Aplicación ejecutándose en la computadora a bordo ROS ejecutándose Estado del robot X0,Y0,T0
Poscondición	Estado del robot X0,Y0,T1 y lo demás sin cambios
Resultado obtenido	Rotación del robot hasta en ángulo indicado
Resultado esperado	Rotación del robot hasta en ángulo indicado
Conclusión	PASADO

Tabla 15: Tarjeta de testing FSR2 ST2.1

11.4.3.1.1 Movimiento angular de 45 grados: Comando right(45)

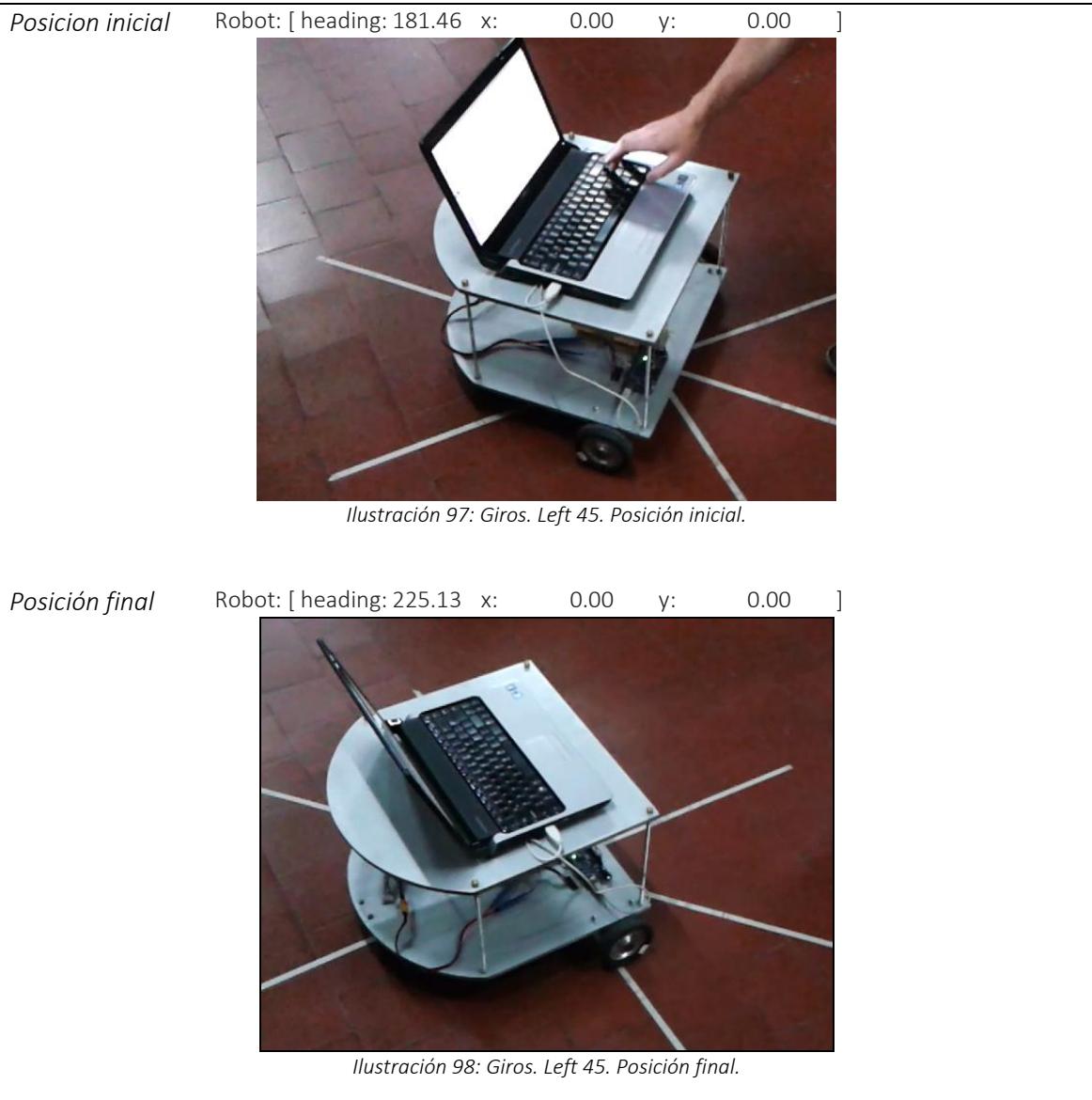
Giro	Comando	% error
45 grados	Right(45)	3,33%



Robot móvil autónomo capaz de producir mapas 3D del interior de una habitación

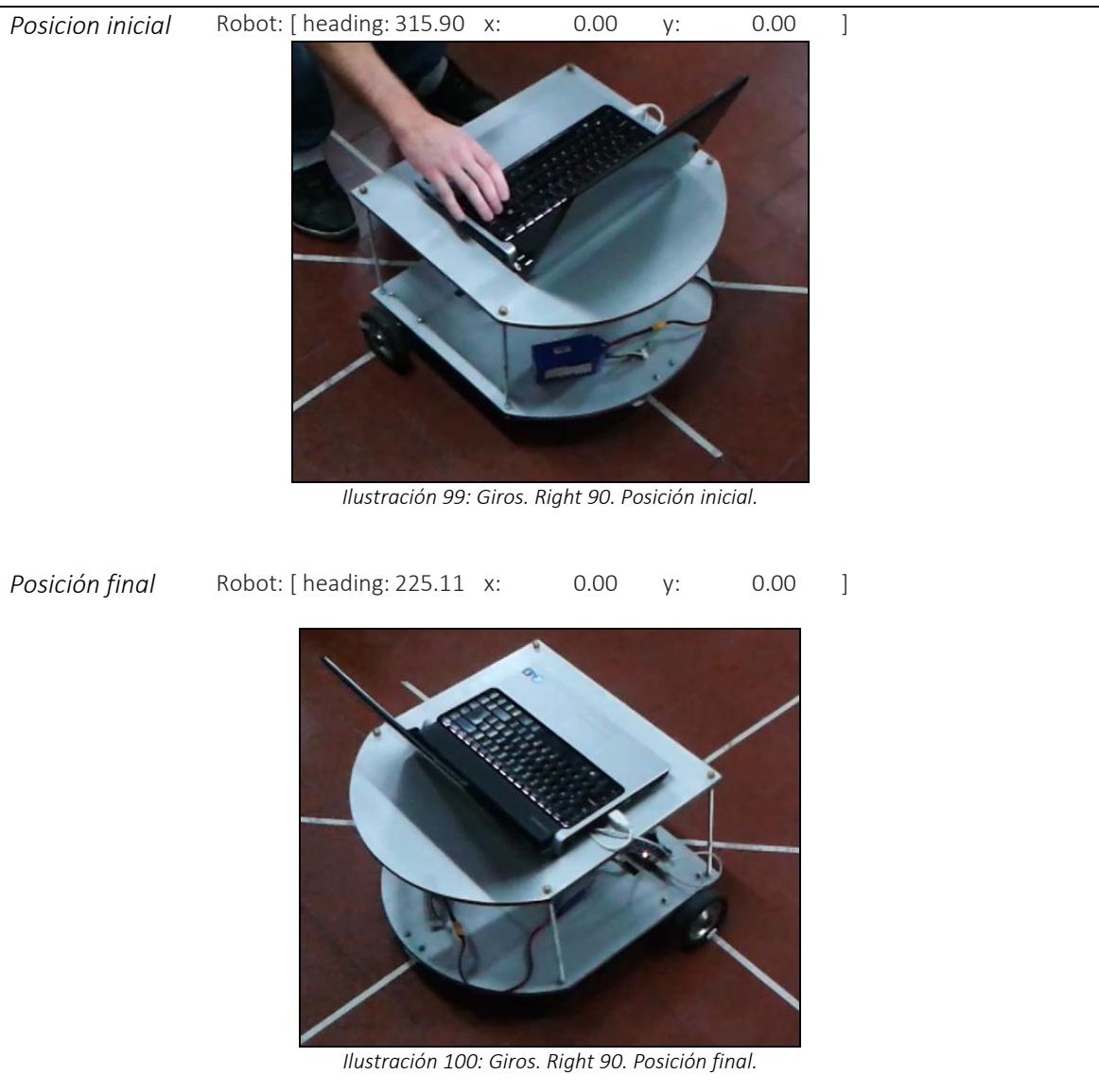
11.4.3.1.2 Movimiento angular de 45 grados: Comando left(45)

Giro	Comando	% error
45 grados	Left(45)	2,95%



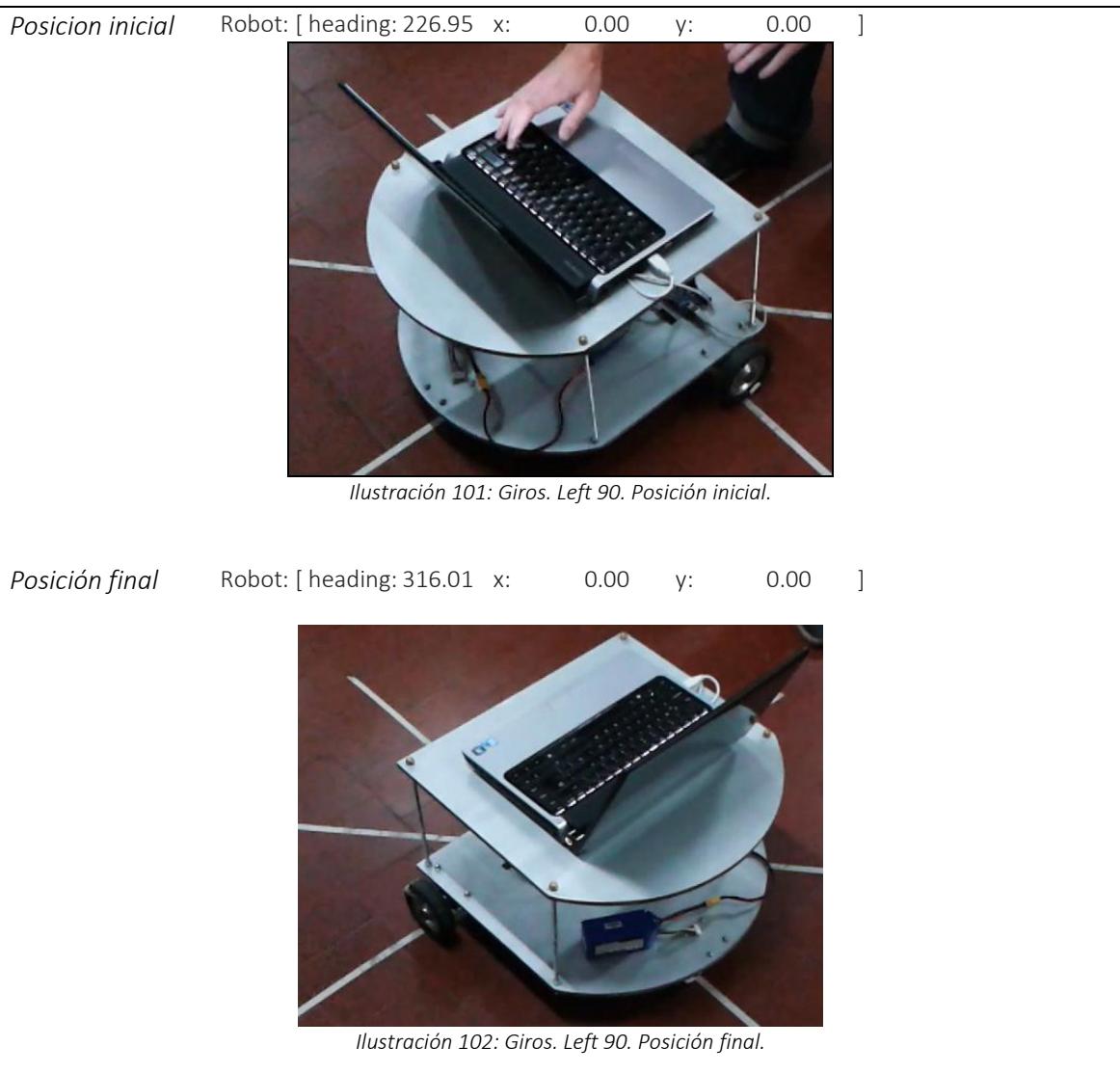
11.4.3.1.3 Movimiento angular de 90 grados: Comando right(90)

Giro	Comando	% error
90 grados	Right(90)	0,87%



11.4.3.1.4 Movimiento angular de 90 grados: Comando left(90)

Giro	Comando	% error
90 grados	Left(90)	1,04%



Robot móvil autónomo capaz de producir mapas 3D del interior de una habitación

11.4.3.1.5 Movimiento angular de 180grados: Comando right(180)

Giro	Comando	% error
180 grados	Right(180)	1,16%



Robot móvil autónomo capaz de producir mapas 3D del interior de una habitación

11.4.3.1.6 Movimiento angular de 180 grados: Comando left(180)

Giro	Comando	% error
180 grados	Left(180)	0,43%



11.4.3.1.7 Análisis de resultados y conclusión para movimientos de giro

Como se puede ver en las pruebas que fueron analizadas, en el peor de los casos, el error es representado el 3,33%. Esto muestra que el giróscopo tiene una muy buena precisión y permitió implementar una localización con un error menor al 4% para movimientos de giro

11.4.3.2 Movimientos lineales, comandos up(), down()

Se realizaron pruebas para avances y retrocesos de distancias de 10cm, 20cm, 50cm, y 100cm. En cada prueba se midió el valor de dx^2+dy^2 que debe ser menor al valor de delta, y se calculó el error en cada medición.

TEST	
Requerimiento	FSR2) Localización del robot
Prueba	ST2.2) Comprobar posición del robot para movimientos lineales: Situar el robot en una posición aleatoria dentro de la habitación y probar los comandos up() y down() para diferentes distancias de avance y retroceso. Luego comprobar que las medidas tomadas por el mismo sean correctas en relación a la distancia recorrida en la realidad.
Precondición	Robot alimentado por batería cargada Aplicación ejecutándose en Arduino Aplicación ejecutándose en la computadora a bordo ROS ejecutándose Estado del robot X0,Y0,T0
Poscondición	Estado del robot X1,Y1,T0, lo demás sin cambios
Resultado obtenido	Desplazamiento del robot hasta una distancia dada.
Resultado esperado	Desplazamiento del robot hasta una distancia dada.
Conclusión	PASADO

Tabla 16: Tarjeta testing FSR2 ST2.2

A continuación se muestran algunas pruebas en detalla y los valores obtenidos en la medición. Y además al final se incluye una tabla que resume las pruebas realizadas.

11.4.3.2.1 Movimiento lineal de 10cm: Comando up()

Distancia	Comando	Delta (error)	$(\Delta a^2)_{\text{máximo}}$	dx^2+dy^2	% error
10cm	Up()	0,3cm	0,09	0,08	2,3%

Posición inicial

Robot: [heading: 0 x: 0 y: 0]



Ilustración 107: Movimiento lineal up. 10 cm. Inicial.

Posición final

Robot: [heading: 359.43 x: 9.77 y: -0.04]



Ilustración 108: Movimiento lineal up. 10 cm. Final.

Análisis de resultados: Luego de avanzar 9,77cm el robot se detiene, por lo que se tiene un error del 2,3%.

11.4.3.2.2 Movimiento lineal de 10cm: Comando down()

Distancia	Comando	Delta (error)	(Delta2)máximo	dx2+dy2	% error
10cm	Down()	0,3cm	0,09	0,06	2,1%

Posición inicial Robot: [heading: 358.08 x: 68.58 y: -1.50]



Ilustración 109: Movimiento lineal Down. 10 cm. Inicial.

Posición final Robot: [heading: 358.09 x: 58.79 y: -1.21]



Ilustración 109: Movimiento lineal Down. 10 cm. Final.

Análisis de resultados: Para el retroceso se obtiene un error del 2,1%. Similar al obtenido para el avance.

11.4.3.2.3 Movimiento lineal de 20cm: Comando up(20)

Distancia	Comando	Delta (error)	$(\Delta^2)^{\text{máximo}}$	dx^2+dy^2	% error
20cm	Up()	0,5cm	0,25	0,01	0,45%



11.4.3.2.4 Movimiento lineal de 20cm: Comando down(20)

Distancia	Comando	Delta (error)	$(\Delta^2)^{\text{máximo}}$	dx^2+dy^2	% error
20cm	Down()	0,5cm	0,25	0,02	0,01%

Posicion inicial Robot: [heading: 359.33 x: 19.91 y: -0.06]



Ilustración 113: Movimiento lineal Down. 20 cm. Inicial.

Posición final Robot: [heading: 358.78 x: 0.10 y: 0.27]



Ilustración 114: Movimiento lineal Down. 20 cm. Final.

Análisis de resultados: Luego de avanzar 19,91cm el robot retrocede 20,01cm, por lo que se el error obtenido en esta medición es prácticamente nulo, 0,01%.

11.4.3.2.5 Movimiento lineal de 100cm: comando up(100)

Distancia	Comando	Delta (error)	(Delta2)máximo	dx2+dy2	% error
100cm	Up(100)	4,58cm	21	20,66	4,17%

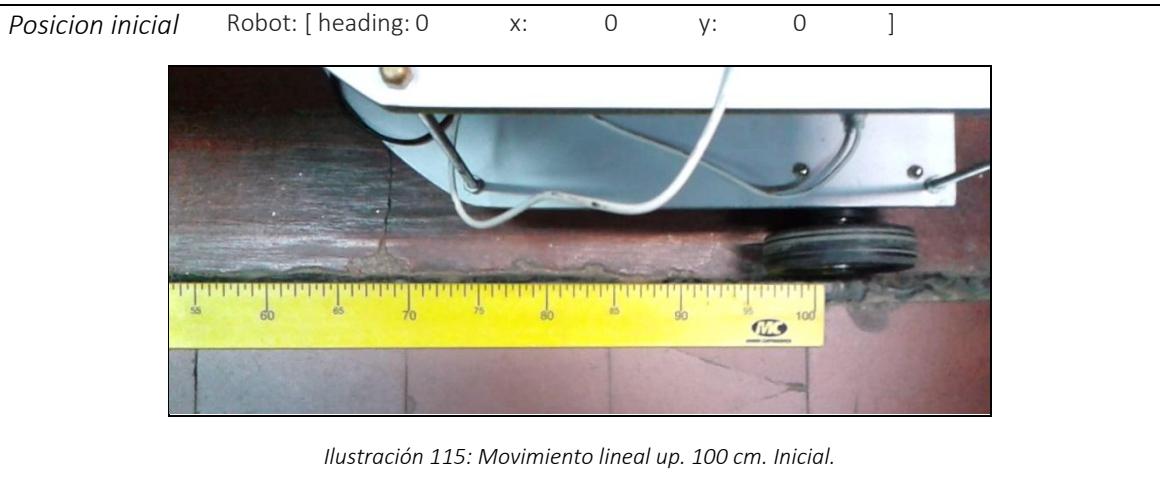


Ilustración 115: Movimiento lineal up. 100 cm. Inicial.



Ilustración 116: Movimiento lineal up. 100 cm. Final.

Análisis de resultados: Dado que la distancia de recorrido es de 1metro, el error obtenido es mayor al obtenido en test anteriores, pero es del 4,17% lo cual es muy satisfactorio.

11.4.3.2.6 Movimiento lineal de 100cm: comando down(100)

Distancia	Comando	Delta (error)	(Delta2)máximo	dx2+dy2	% error
100cm	Down(100)	4,58cm	21	20,86	4,41%

Posición inicial Robot: [heading: 359.88 x: 95.83 y: 1.91



Ilustración 117: Movimiento lineal down. 100 cm. Inicial.

Posición final Robot: [heading: 357.14 x: -0.24 y: 4.27]



Ilustración 118: Movimiento lineal down. 100 cm. Final.

Análisis de resultados: Al igual que en el avance, el error es cercano al 4%. Se lo considera un muy buen resultado.

11.4.3.2.7 Resumen de otros test

En la siguiente tabla se muestra un resumen de los valores de error obtenidos para cada una de las distancias probadas. Algunas fueron mostradas en detalle en secciones anteriores, y otras no fueron incluidas porque no aportan información adicional útil. Sí son incluidas en la siguiente tabla a modo de resumen.

Distancia		Delta (error)	$(\Delta^2)_{\text{máximo}}$	dx^2+dy^2	% error
10cm	Up()	0,3cm	0,09	0,08	2,3%
20cm	Up(20)	0,5cm	0,25	0,02	0,45%
50cm	Up(50)	1,41cm	2	1,89	2,67%
100cm	Up(100)	4,58cm	21	20,66	4,17%

Tabla 17: Resumen de tests de movimiento lineal

11.4.3.2.8 Análisis de resultados y conclusión para movimientos lineales

Tal como se observa en la tabla, el valor de $dx^2 + dy^2$ crece a medida que las distancias son mayores. Esto se debe a la desviación normal que tiene el robot, que si bien es pequeña, se percibe aún más cuando las distancias son mayores.

Si el delta del método que comprueba la posición del robot periódicamente, es menor que el menor valor de $dx^2 + dy^2$ alcanzado, el robot nunca se detendrá. Para evitar esto se plantean dos posibles soluciones:

- La primera consiste en hacer que el valor delta sea configurable en tiempo de ejecución. Esto es, a medida que el robot recibe comandos, el valor de delta cambiaría en función de la distancia que se quiera recorrer. Por ejemplo si el robot recibe el comando up(), la distancia a recorrer es de 10cm por que se debería establecer un valor delta = 0,3cm que es el error o la desviación esperada. Por otro lado, si el robot recibe up(100) o down(100), el valor de delta debería ser 4,58cm.
- La segunda solución consiste en limitar la distancia de los comandos que puedan enviarse al robot a que sea siempre la misma. Si el planificador de caminos (path planner) siempre envía comandos up() o down(), no sería necesario tener diferentes valores de delta, ya que siempre el robot avanzará o retrocederá la misma distancia, de 10cm. Entonces el valor de delta será siempre de 0,3cm y cuando el robot deba avanzar una cantidad mayor, lo hará como una sucesión de comandos up() o down(). Esto además asegura que el error por la desviación va a ser muy pequeño.

Para el robot si bien las dos implementaciones son posibles, se eligió la segunda, dado que el planner envía sólo comandos up() y down() para el movimiento lineal, y right(), left() para el movimiento angular. De todas formas se podría utilizar la otra implementación simplemente cambiando el valor de delta en tiempo de ejecución, dependiendo del comando recibido.

11.4.4 Conclusión

Según las pruebas realizadas para movimientos de giro el robot registra un error menor al 4%. Por otro lado para los movimientos lineales también se logra tener un error muy pequeño, dado que con cada comando que el planificador le envía al robot, le debe decir cuánto avanzar en cm. Si se toma una cantidad de avance baja, por ejemplo 10cm, se mantiene un error de localización muy pequeño. Si el robot necesita avanzar cantidades mayores lo hará como una secuencia de comandos.

11.5 Mapeo y odometría visual

11.5.1 Introducción

Continuando con la introducción sobre odometría visual presentada en el marco teórico (ver 7.4.1), se describe su principio de funcionamiento:

Actualmente, hay dos formas conocidas para realizar el cómputo del movimiento a partir de una secuencia de imágenes tomadas por cámaras desplazándose:

- **Basado en las apariencias (Appearance based):** se utilizan los valores de intensidad de los píxeles en cada imagen.
- **Basado en la extracción de características (Feature extraction):** se identifican regiones con características particulares y distintivas, y se las rastrea a lo largo de las imágenes. Este suele ser el más común, dado que requiere una menor complejidad computacional. Las características más comunes son las aristas, las esquinas (unión de dos aristas) y las burbujas (patrón que difiere de sus píxeles vecinos ya sea en color, intensidad o textura). Generalmente, se aplica un filtro Gaussiano para suavizar las imágenes antes de la detección de características.



Ilustración 119: Detección de características y flujo óptico (Myung Hwangbo)

En la imagen anterior se observa que incluso luego de rotar la cámara (o la escena), se siguen detectando las mismas características, lo cual brinda la posibilidad de detectar el movimiento o cambio de posición de las mismas. Esto es denominado flujo óptico (optical flow).

Generalmente, los métodos basados en extracción de características son más precisos y rápidos que los globales (basados en apariencia), siendo estos últimos muy caros computacionalmente. Esto es de esperarse ya que la extracción de características puede ser considerada como una reducción de la dimensionalidad.

La idea general del algoritmo es la siguiente:

- 1) Obtención de imágenes utilizando cámaras simples, cámaras estéreo o cámaras omnidireccionales.
- 2) Corrección de imágenes: aplicar técnicas de procesamiento de imágenes para eliminar posibles distorsiones causadas por la cámara, iluminación, etc. que afectan a la siguiente etapa.
- 3) En este paso se usa una de las dos alternativas mencionadas anteriormente:

- a) Detección y extracción de características (Feature Extraction) de la imagen y detectarlas o correlacionarlas a lo largo de los frames para construir el flujo óptico (optical flow).
 - b) Basado en apariencias
- 4) Remover los valores extremos o anómalos.
- 5) Estimación del movimiento de la cámara usando el flujo óptico. Existen dos alternativas:
- o Filtros de Kalman
 - o Minimizar error de la re proyección de las imágenes

A continuación se muestra un diagrama de flujo con el procedimiento:

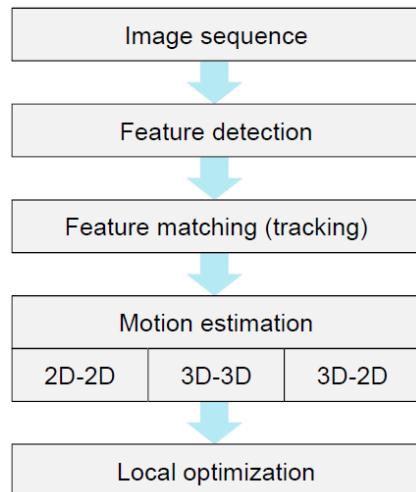


Ilustración 120: Algoritmo de odometría visual (Scaramuzza Davide)

11.5.1.1 Mapeo y localización simultánea SLAM

SLAM (del inglés Simultaneous Localization and Mapping), es una técnica utilizada por robots, especialmente vehículos autónomos, para construir un mapa de un entorno desconocido y, al mismo tiempo, utilizarlo para estimar su localización (posición y orientación).

Para una explicación más detallada, la siguiente imagen muestra cómo, a medida que avanza el robot, se registran características del entorno, denominadas puntos de referencia, y, a partir de estas, se estima la localización del robot y se puede detectar si se vuelve a visitar zonas previamente vistas (Hugh Durrant-Whyte).

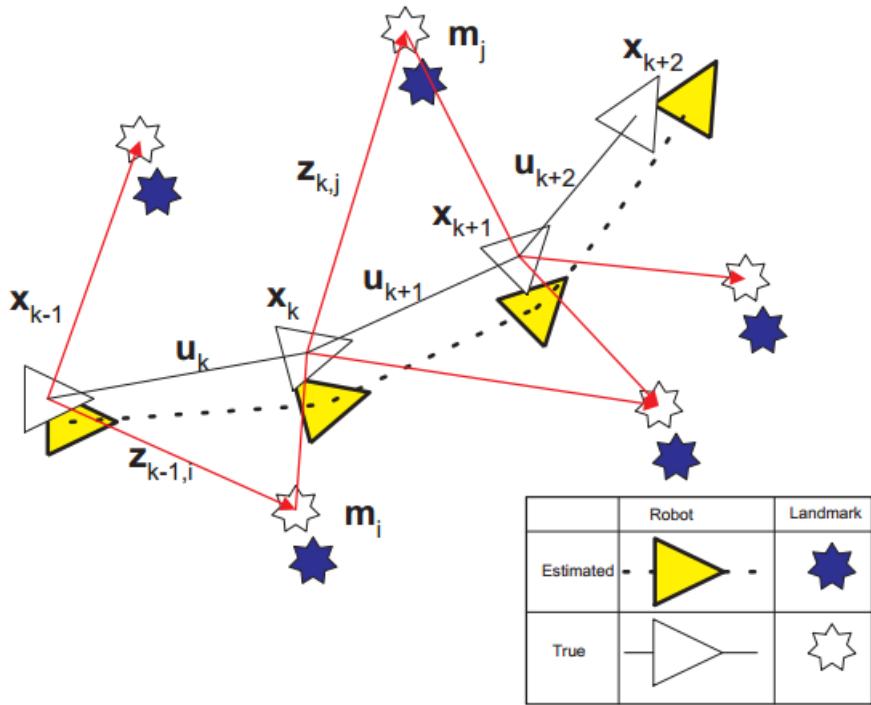


Ilustración 121: SLAM, a medida que el robot avanza, detecta puntos referencias a través de los cuales estima su localización.

La imagen muestra una representación de un robot que ejecuta SLAM a medida que se desplaza. Cada triángulo blanco simboliza la verdadera ubicación del robot, mientras que los amarillos indican donde éste cree estar. Además, las estrellas blancas indican la posición real de los puntos de referencia y las azules las posiciones estimadas. Los símbolos utilizados tienen el significado descripto a continuación:

x_k Vector estado. Define la posición y la orientación.

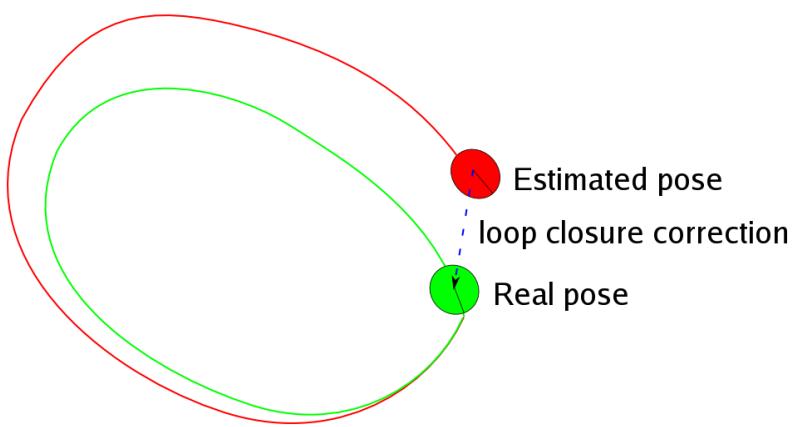
u_k Comando de movimiento aplicado en el tiempo k-1.

m_k Vector que describe la localización del punto de referencia k. Se asume invariante.

z_{ik} Observación del punto de referencia i en el tiempo k

Cada vez que se realiza un movimiento, el robot realiza una nueva observación a cada uno de los puntos de referencia. Teniendo en cuenta que en cada observación y en cada movimiento se tiene cierta incertidumbre, la estimación de la ubicación del robot consiste en una superficie probabilística en la que cada punto tiene asociada una probabilidad de que el robot se encuentre sobre dicho punto.

En caso de que el robot vuelva a una zona ya visitada, este debe ser capaz de identificarla y cerrar el ciclo (loop-closure). Una vez cerrado el ciclo, sabiendo que se ha vuelto a una posición conocida, se pueden reducir los errores del mapa y de la trayectoria obtenida. A continuación se muestra una imagen explicativa.



<http://cogrob.ensta-paristech.fr/loopclosure.html>

Ilustración 122: Corrección lograda mediante SLAM (Scaramuzza Davide)

La imagen muestra cómo, luego de cerrar el ciclo, se corrigen los errores y se obtiene una nueva estimación de posición y de trayectoria. La trayectoria antes del cierre es la roja y la posterior al cierre es la verde.

En los últimos años, el problema de SLAM ha sido intensamente estudiado por diferentes grupos de investigación de robótica en el mundo. Numerosas técnicas han sido implementadas pero no tantas están disponibles para la comunidad. Por esta razón, la organización openSLAM tiene como objetivo organizar y publicar todas las implementaciones robustas que les sean mandadas (openSLAM)

A medida que un robot realiza el mapeo de un entorno, este va documentando los puntos de referencia que observa (Hugh Durrant-Whyte). Estos serán utilizados durante el mapeo posterior para la localización (odometría visual) y para detectar si ya se ha visitado alguno de estos lugares. En este proyecto, los puntos de referencias consisten en características de las imágenes observadas.

11.5.2 Implementación de odometría visual y mapeo

Previamente se describieron los pasos necesarios para realizar odometría visual a partir de una secuencia de imágenes y se indicó que hay dos alternativas posibles a la hora de relacionar una imagen con la siguiente; basada en apariencias y basada en la extracción de características. Para el presente trabajo, se probaron ambos métodos para luego realizar una comparación.

Para que la comparación sea justa, en ambos casos se utilizó la misma computadora, la misma cámara RGBD (Xbox Kinect) y su driver (OpenNI). Además el mapeo fue realizado sobre la misma habitación. La prueba consistió en ubicar la cámara en el centro de la habitación y lentamente realizar un giro de 360° para que ésta sea capaz de observar toda la habitación. Los resultados fueron visualizados a partir de la herramienta Rviz que provee ROS.

11.5.2.1 Método basado en apariencias

Se utilizó la implementación del módulo *RTAB-Map: Real-Time Appearance-Based Mapping* de la Universidad de Sherbrooke, de Quebec, Canadá. El trabajo se encuentra publicado bajo el nombre Appearance-Based Loop Closure Detection for Online Large-Scale and Long-Term Operation (Mathieu Labbe).

En las siguientes imágenes, se podrá apreciar la reconstrucción en tres dimensiones de la habitación mencionada. Cabe destacar que es posible distinguir formas, colores y objetos con claridad. Además, la forma perimetral, rectangular, es representada correctamente en el mapa. Ya que se usa Rviz para la visualización, es posible rotar la imagen, hacer zoom, trasladarse por la escena, etc (ver 7.3.1.1.3).

Como se puede ver, el mapa es dibujado mediante una nube de puntos de colores (Point Cloud) a partir de la librería PCL (Point cloud library). Específicamente, es declarada como PointCloud<PointXYZRGB> ya que incluye el canal para los colores.

Los ejes de coordenadas representan la posición de la cámara. Existe una línea que conecta este sistema con el sistema fijo (donde estaba la cámara al inicializar el proceso) que representa la traslación de la cámara respecto al sistema fijo. Tanto la posición (x,y,z) como la orientación yaw, pitch y roll de la cámara son resultado de la odometría visual.

A continuación se muestran tres imágenes del mapa obtenido:

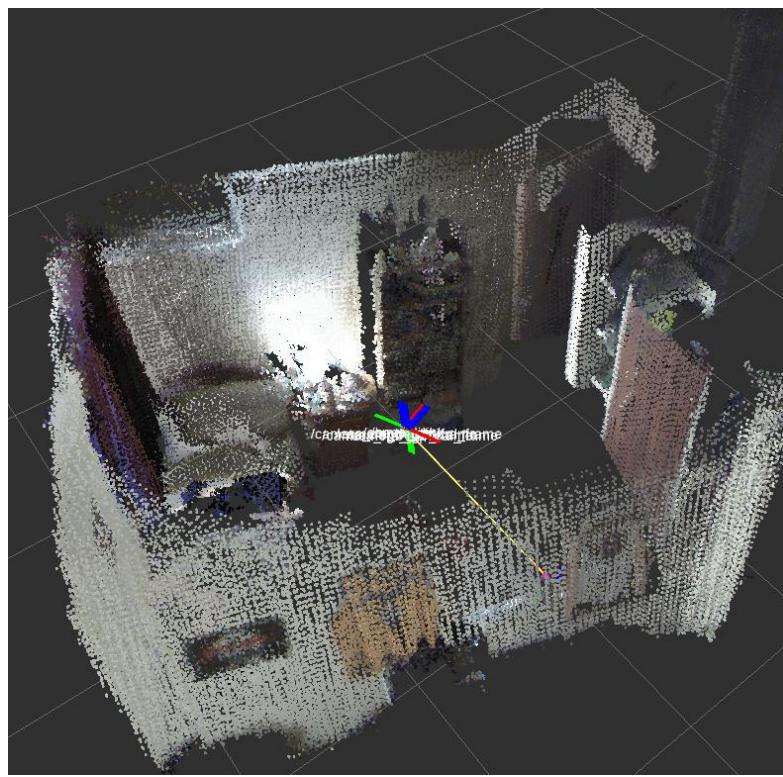


Ilustración 123: Mapeo 3D de la habitación con el método basado en apariencias

Robot móvil autónomo capaz de producir mapas 3D del interior de una habitación

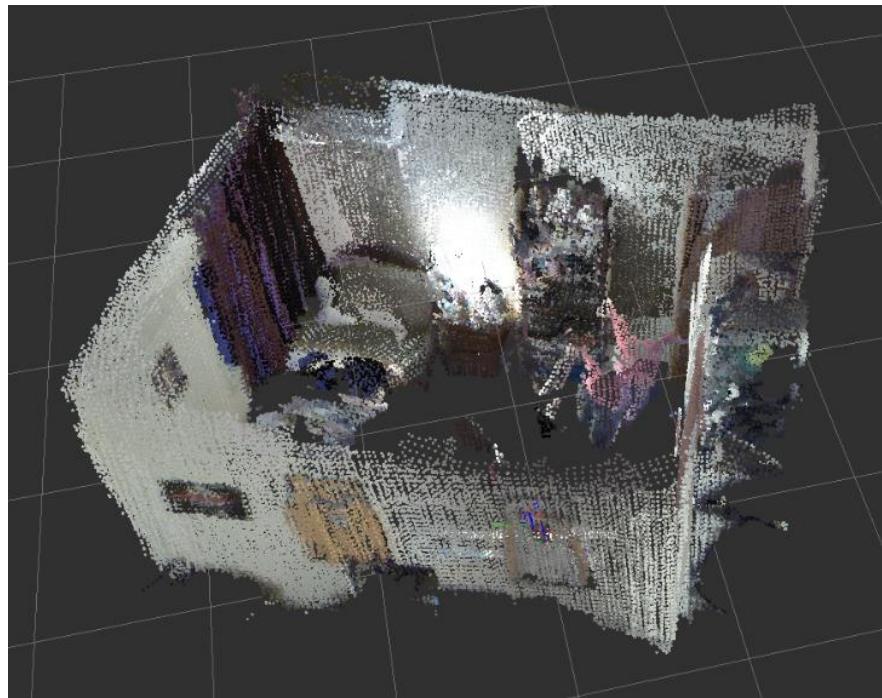


Ilustración 124: Otra perspectiva del mapeo 3D de la habitación con el método basado en apariencias

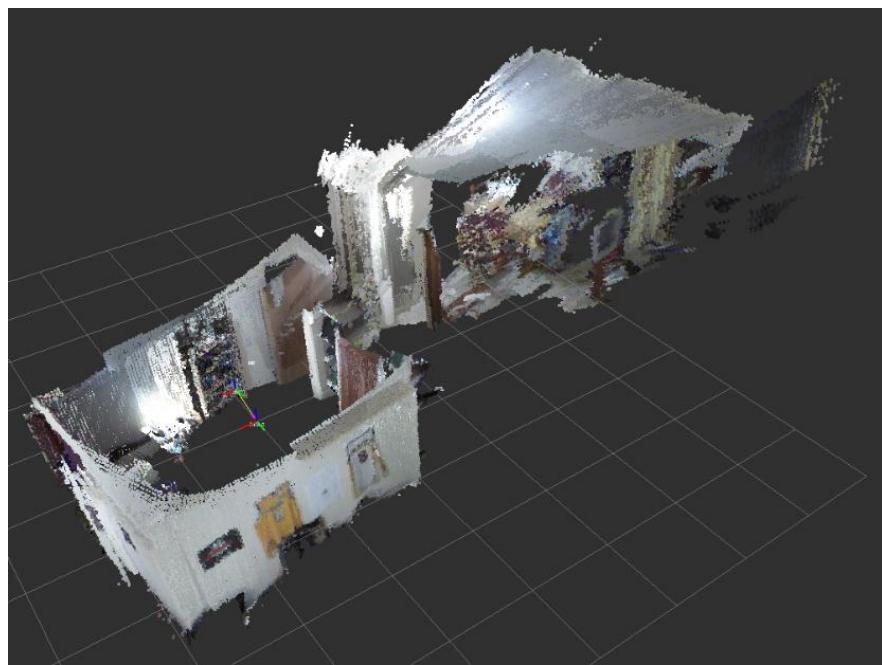


Ilustración 125: Mapeo basado en apariencias. Habitación y ambiente vecino.

En esta última imagen, la cámara captó también parte de la habitación contigua, ya que la luz de ésta se encontraba encendida, y no lo estaba en las pruebas anteriores.

Ya que el mapeo es observable a medida que se rota la cámara, también es observable el hecho de que al aumentar la velocidad de giro, la odometría empieza a tener problemas. La construcción del mapa se detiene, hasta que el módulo logra relocalizar la cámara, al hacerla coincidir con algunas de las tomas previas.

Se concluye en que para lograr un mapeo correcto, es necesario que la velocidad de giro sea muy baja y evitar movimientos bruscos y vibraciones.

11.5.2.2 Método basado en extracción de características

El módulo utilizado es desarrollado y mantenido por el grupo de robótica de la universidad City College of New York (CCNY) y se llama ccny_rgbd. La implementación está basada en el trabajo Fast visual odometry and Mapping from RGB-D Data (Ivan Dryanovski, 2013).

El siguiente diagrama funcional muestra un panorama de cómo está compuesto el módulo:

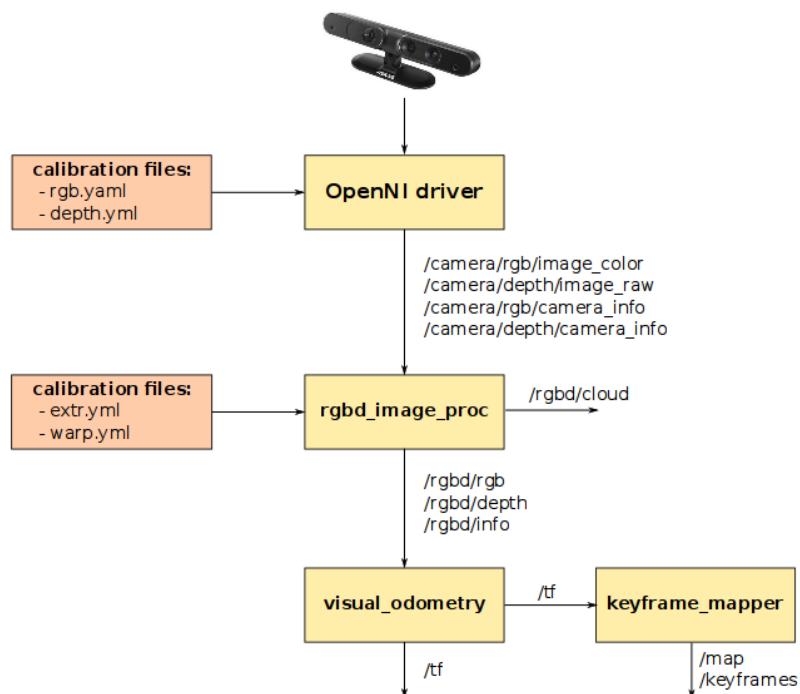


Ilustración 126: Nodos constituyentes del trabajo ccny_rgbd (Dryanovski)

En este caso, la salida consiste en keyframes o marcos claves. Es decir, el mapa obtenido, es una secuencia de marcos (que son nubes de puntos) y al mostrarlos a todos a la vez mediante Rviz, se obtiene el mapa deseado. Rviz, por defecto, solo muestra una cantidad dada de los últimos marcos recibidos (no todos), pero es posible alargar el tiempo de muestra (decay time). De esta forma, cada marco permanecerá durante todo el mapeo y será posible visualizar el mapa completo.

Una vez descriptos, se muestran algunas imágenes de pruebas hechas con dicho módulo sobre la misma habitación:

Robot móvil autónomo capaz de producir mapas 3D del interior de una habitación

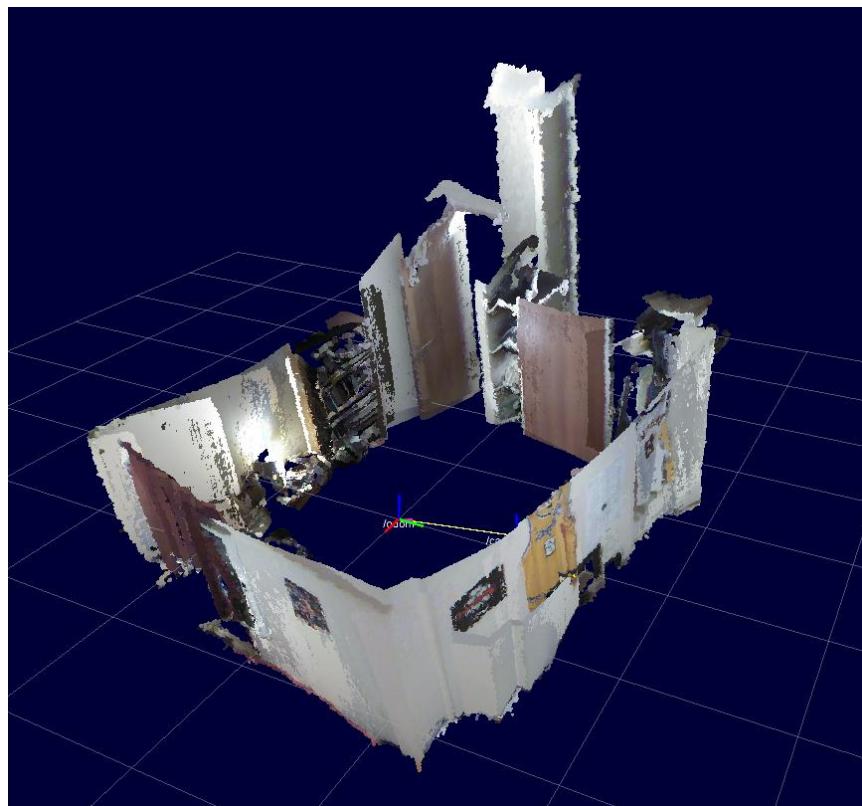


Ilustración 127: Mapeo 3D de la habitación con el método basado en extracción de características

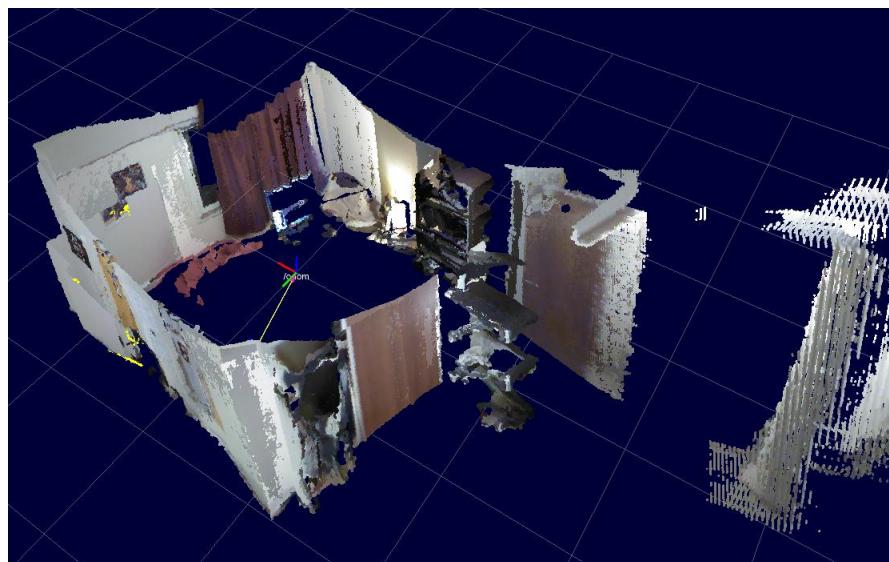


Ilustración 128: La habitación según otra perspectiva. Mapeo con método basado en extracción de características

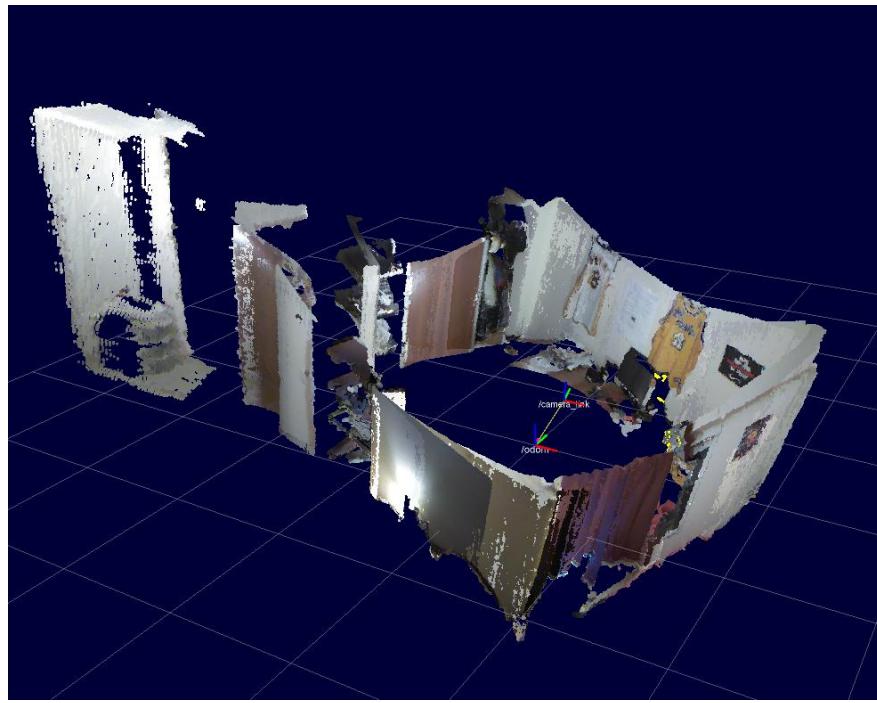


Ilustración 129: Tercera perspectiva de la habitación producida con el método basado en extracción de características.
Se puede apreciar el ambiente vecino a la habitación.

11.5.2.3 Elección del método más adecuado y testing

Luego de realizar varias pruebas, se observó que el último enfoque, el basado en características, es mucho más veloz que el primero. Las rotaciones y las traslaciones pueden ser realizadas más rápidamente. Debido a que en el presente proyecto, la cámara va situada sobre un robot móvil, el factor velocidad es muy importante. Durante el movimiento normal del robot se producen vibraciones y se pueden realizar rápidos giros y traslaciones que no darían tiempo suficiente para la correlación global de las imágenes.

En cuanto a la calidad y los detalles de los mapas, es evidente la diferencia entre los mapas generados por RTAB y ccny_rgbd. En el primero se muestran más detalles aunque con menor resolución (se pueden apreciar los puntos que forman la imagen) y no se producen las dobles paredes. En ccny_rgbd, al volver a una pared que ya había sido visitada, se vuelve a publicar el keyframe de lo visto y, por los errores acumulados durante la odometría, se pueden observar pequeñas imperfecciones como si la pared fuera doble. Además, cabe destacar que ccny_rgbd cuenta con una buena documentación y API, que permiten adentrarse en los detalles y utilizar sus resultados (ccny_rgbd documentation)

Por los motivos presentados, se decidió utilizar ccny_rgbd como método por defecto. En los casos que sea importante un mapa de alta calidad y no sea necesario un rápido movimiento, se podrá optar por RTAB.

A continuación se muestran las tarjetas de test, que corresponden a las pruebas realizadas en cada uno de los métodos analizados:

TEST	
Requerimiento	FSR6) Interfaz gráfica para mostrar mapas
Prueba	ST6.1) probar la interfaz con mapas de prueba
Precondición	Robot alimentado por batería cargada

	Kinect y OpenNI en funcionamiento Aplicación ejecutándose en Arduino Aplicación ejecutándose en la computadora a bordo ROS y Rviz ejecutándose
Poscondición	Mismas que precondiciones
Resultado obtenido	Visualización de la construcción del mapa a medida que es explorado
Resultado esperado	Visualización de la construcción del mapa a medida que es explorado
Conclusión	PASADO

Tabla 18: Tarjeta de testing FSR6 ST6.1

TEST	
Requerimiento	FSR7) Generar mapas 3D
Prueba	ST7.1) Verificar coincidencia entre mapas y la habitación dinámicamente
Precondición	Robot alimentado por batería cargada Kinect y OpenNI en funcionamiento Aplicación ejecutándose en Arduino Aplicación ejecutándose en la computadora a bordo ROS y Rviz ejecutándose
Poscondición	Mismas que precondiciones
Resultado obtenido	Construcción del mapa a medida que es explorado
Resultado esperado	Construcción del mapa a medida que es explorado
Conclusión	PASADO

Tabla 19: Tarjeta de testing FSR7 ST7.1

11.5.2.4 Test de mapeo para diferentes escenarios

Tal como se describe en las siguientes tarjetas, se realizaron algunos test en escenarios diferentes para corroborar la correcta construcción de los mapas.

TEST	
Requerimiento	FSR7) Generar mapas 3D
Prueba	ST7.2) Probar diferentes escenarios
Precondición	Robot alimentado por batería cargada Kinect y OpenNI en funcionamiento Aplicación ejecutándose en Arduino Aplicación ejecutándose en la computadora a bordo ROS y Rviz ejecutándose
Poscondición	Mismas que las precondiciones
Resultado obtenido	Se genera una reconstrucción 3D del entorno.
Resultado esperado	Se genera una reconstrucción 3D del entorno.
Conclusión	PASADO

Tabla 20: Tarjeta de testing FSR7 ST7.2

11.5.2.4.1 Mapeo en el LAC

Con fines demostrativos, se muestran mapas del laboratorio de arquitectura de computadoras (LAC), en particular de las zonas en las cuales se desarrolló este trabajo.

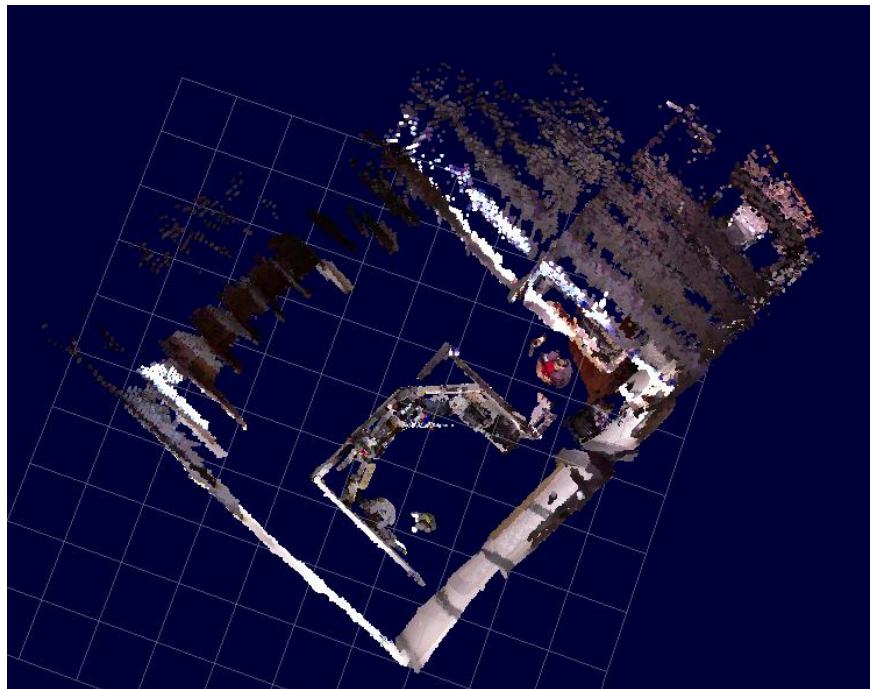


Ilustración 130: Box de trabajo en el LAC, desde arriba

Analizando el mapa desde otra perspectiva se tiene:

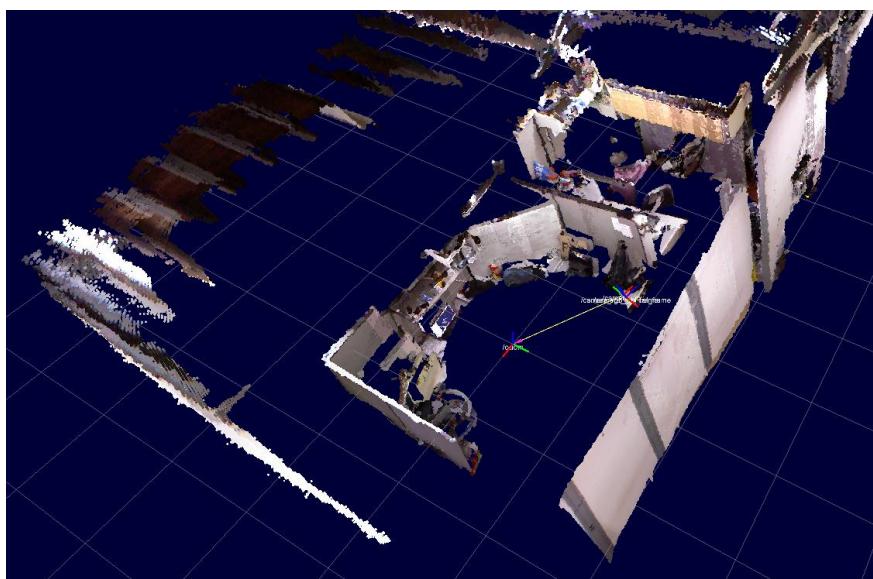


Ilustración 131: Otra perspectiva del box de trabajo en el LAC

11.5.2.4.2 Mapeo en el GRSI

También se muestran los resultados del mapeo realizados en el laboratorio del Grupo de Robótica y Sistemas Integrados (GRSI).

La primera imagen muestra el laboratorio desde arriba, luego desde los costados, y finalmente desde adentro. En la última imagen, es posible apreciar el detalle que se guarda en los mapas.

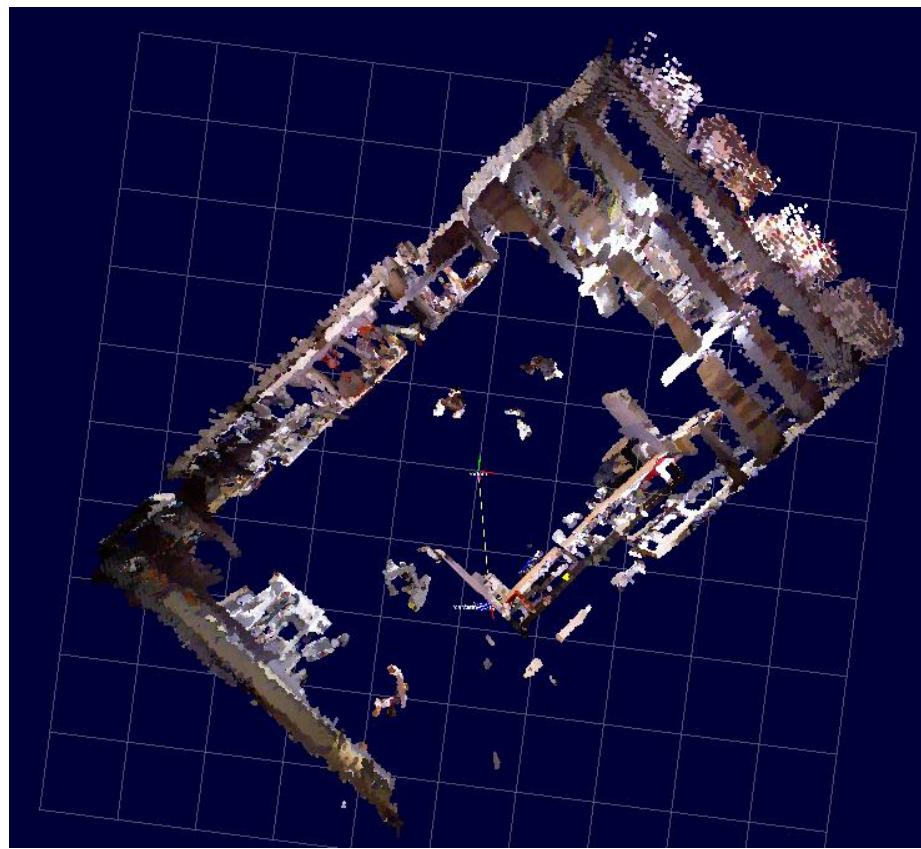


Ilustración 132: GRSI desde arriba. Notar que la cámara fue posicionada en el centro del laboratorio. Punto desde el cual no eran visibles todas las paredes. Por esta razón, se ven "huecos" en el mapa.

Las próximas imágenes muestran otras perspectivas del mismo laboratorio. Las impresiones de pantalla fueron tomadas con un mayor zoom para destacar el detalle con el que se representan los distintos objetos presentes en la escena.

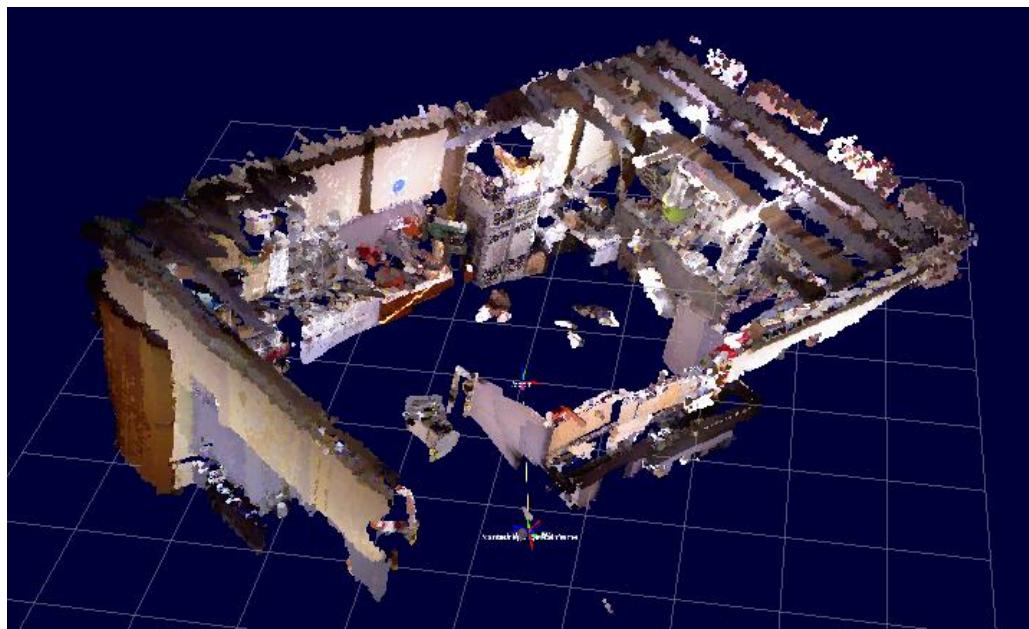


Ilustración 133: GRSI desde otro ángulo

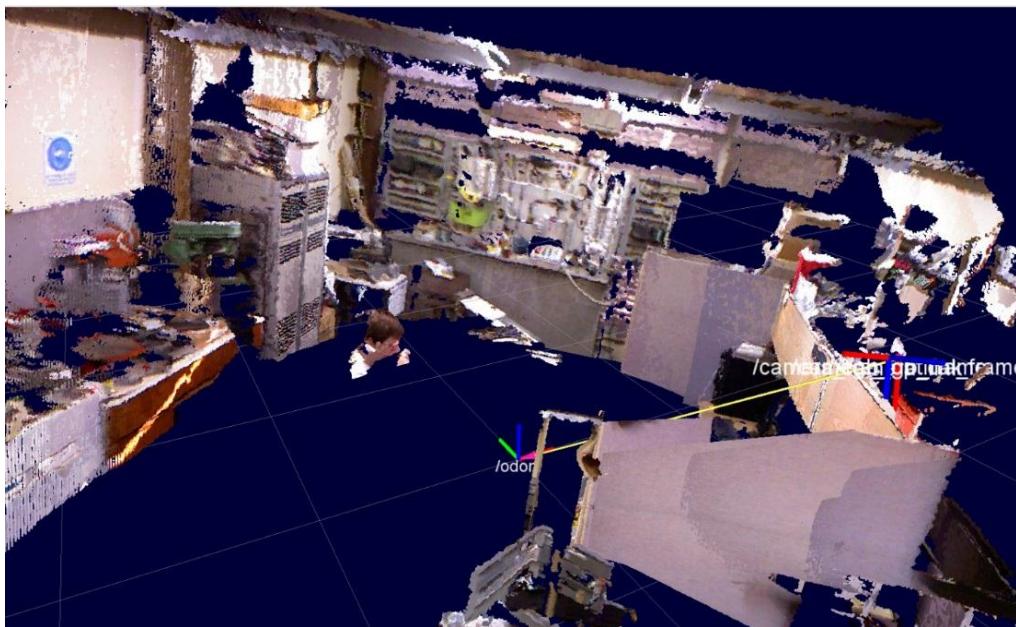


Ilustración 134: GRSI desde adentro con el fin de mostrar los detalles.

11.5.2.5 Arquitectura del paquete *ccny_rgbd* de ROS

A continuación se muestra el grafo que provee la herramienta *rqt_graph* de ROS construido durante la ejecución del módulo de mapeo *ccny_rgbd*. Se pueden observar, a un nivel muy detallado, los diferentes nodos que lo componen y las relaciones entre ellos.

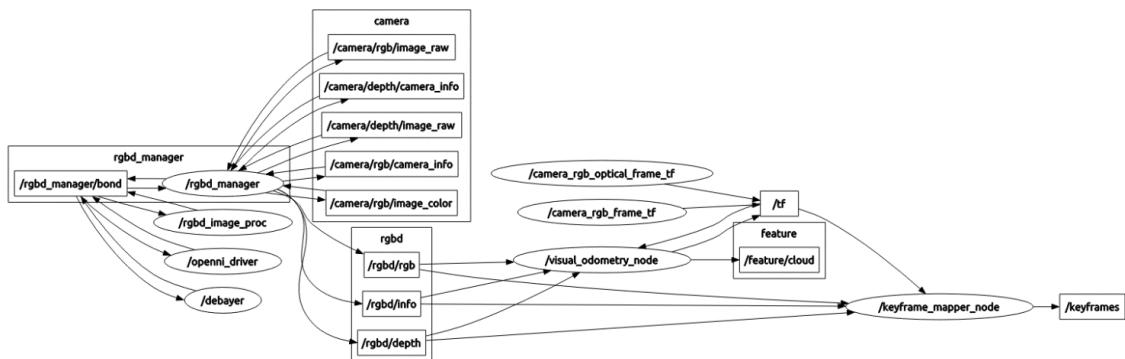


Ilustración 135: Grafo de ROS durante la ejecución del mapeo

Como muestra la imagen anterior, la cámara está constituida por varios nodos, que proveen diferentes streams de información. Dichos nodos son utilizados por el *rgbd_manager* que incluye al driver *openNI*, y el proceso de combinación de los puntos de profundidad con los de color. El resultado es el nodo *rgbd* que luego es utilizado por el nodo de la odometría visual que se encarga de generar las coordenadas de la cámara, las características de las imágenes y los keyframes que constituyen el mapeo. Como se puede ver en su totalidad el módulo es bastante complejo y está compuesto de muchos nodos.

11.5.3 Precisión de la odometría visual

En las imágenes anteriores se observaban ejes de coordenadas que representan el estado actual de la cámara. En esta sección, con un simple experimento, usando el módulo `ccny_rgbd`, se mostrará la precisión de la odometría visual.

11.5.3.1 *Trayectoria*

En este experimento analizará la trayectoria o recorrido de la cámara a partir de la odometría visual. Es decir, se mostrará con una línea, los puntos por los que pasó la cámara. Esta prueba se desarrolló en un lugar más espacioso.

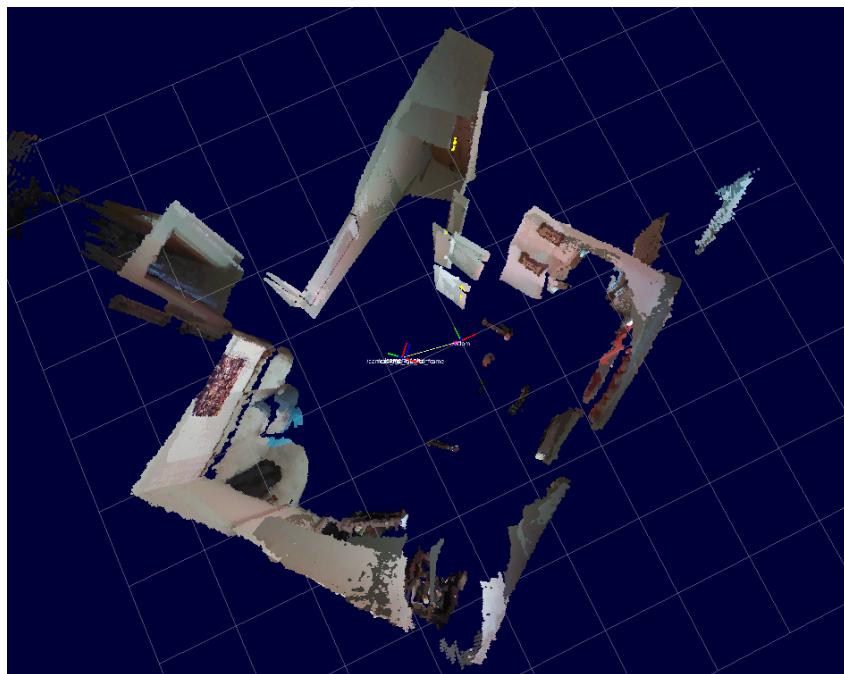


Ilustración 136: Recorrido a través de la odometría visual. Posición inicial

En el centro del lugar, hay una mesa. El recorrido consiste en dar una vuelta alrededor de ella, con la cámara en las manos. Dicho recorrido está representado con verde en la siguiente imagen.

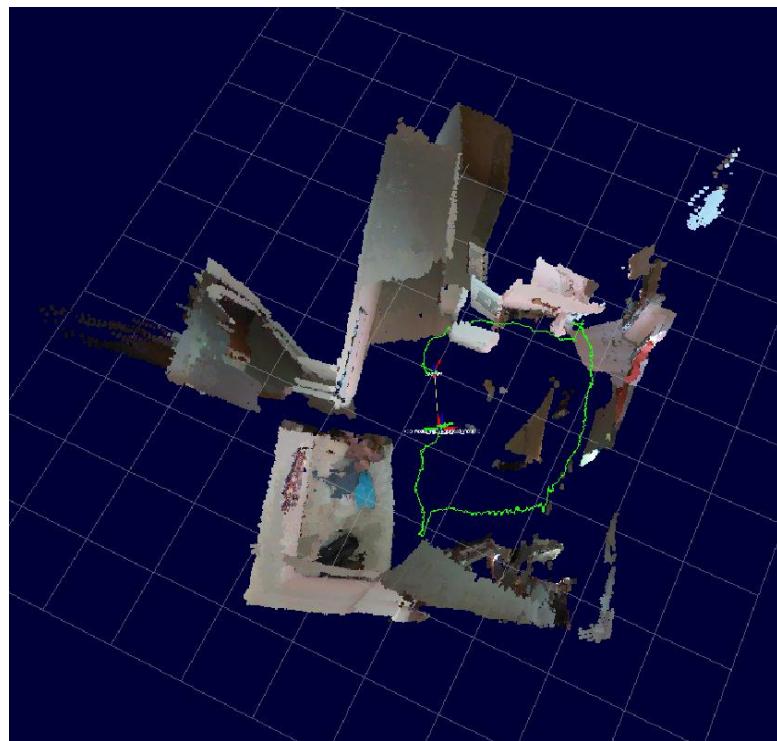


Ilustración 137: Recorrido a través de la odometría visual.

11.5.3.1.1 Test de mediciones de distancias con el sensor

Las presentes mediciones se realizaron trasladando y/o rotando la cámara manualmente. Se muestra la salida de la odometría visual a través de Rviz. Los valores de (x,y,z) son mostrados en metros y las rotaciones se representan con cuaterniones con ángulos en radianes.

TEST	
Requerimiento	FSR5) Medición de distancias con el sensor
Prueba	ST5.1) comprobar exactitud de mediciones a puntos conocidos
Precondición	Robot alimentado por batería cargada Aplicación ejecutándose en Arduino Aplicación ejecutándose en la computadora a bordo ROS y Rviz ejecutándose
Poscondición	Mismas que las iniciales
Resultado obtenido	Construcción del mapa a medida que es explorado
Resultado esperado	Construcción del mapa a medida que es explorado
Conclusión	PASADO

Tabla 21: Tarjeta de testing FSR5 ST5.1

A continuación se muestran los test de traslaciones y rotaciones por separado:

11.5.3.1.1.1 Traslaciones de la cámara

A partir de una posición inicial, se traslada el sensor Kinect primero 50 centímetros y luego 1 metro. Esto fue realizado repetidas veces ida y vuelta para obtener varios valores y comprobar si se genera algún tipo de error acumulativo. Cabe mencionar, que el sistema de

Robot móvil autónomo capaz de producir mapas 3D del interior de una habitación

coordenadas de Kinect se denomina `/camera_link` y las transformaciones mostradas son relativas al marco `/odom`.

TEST	
Requerimiento	FSR5) Medición de distancias con el sensor
Prueba	ST5.1) comprobar exactitud de mediciones a puntos conocidos
Precondición	Robot alimentado por batería cargada Aplicación ejecutándose en Arduino Aplicación ejecutándose en la computadora a bordo ROS y Rviz ejecutándose
Poscondición	Mismas que las iniciales
Resultado obtenido	Construcción del mapa a medida que es explorado
Resultado esperado	Construcción del mapa a medida que es explorado
Conclusión	PASADO

Tabla 22: Tarjeta de testing FSR5 ST5.1

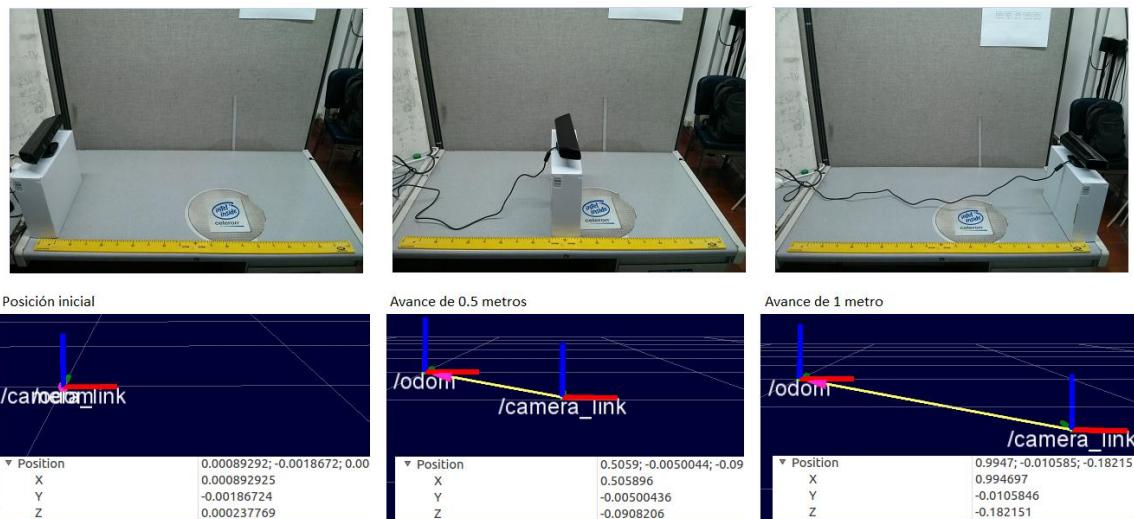


Ilustración 138: Mediciones de distancia mediante la odometría visual con Kinect

La siguiente tabla muestra las mediciones obtenidas a través de Kinect. Cada columna, de izquierda a derecha, contiene las sucesivas mediciones tomadas en las posiciones 0 metros, 0,5 metros y 1 m respectivamente. Luego, se realiza una regresión lineal para medir variaciones en la medición, y se muestra la pendiente obtenida para cada una de las posiciones.

Posición inicial (0 m)	50 cm	1 m
0,000	0,496	0,994
0,003	0,488	0,992
0,006	0,496	0,991
0,001	0,492	0,990
0,002	0,498	0,992
0,001	0,489	0,993
0,002	0,492	0,995

0,004	0,493	0,993
0,003	0,498	0,992
0,001	0,495	0,991
Regresión Lineal – Pendiente obtenida		
-0,000036	0,000224	-0,000060

Tabla 23: Mediciones de distancias con Kinect

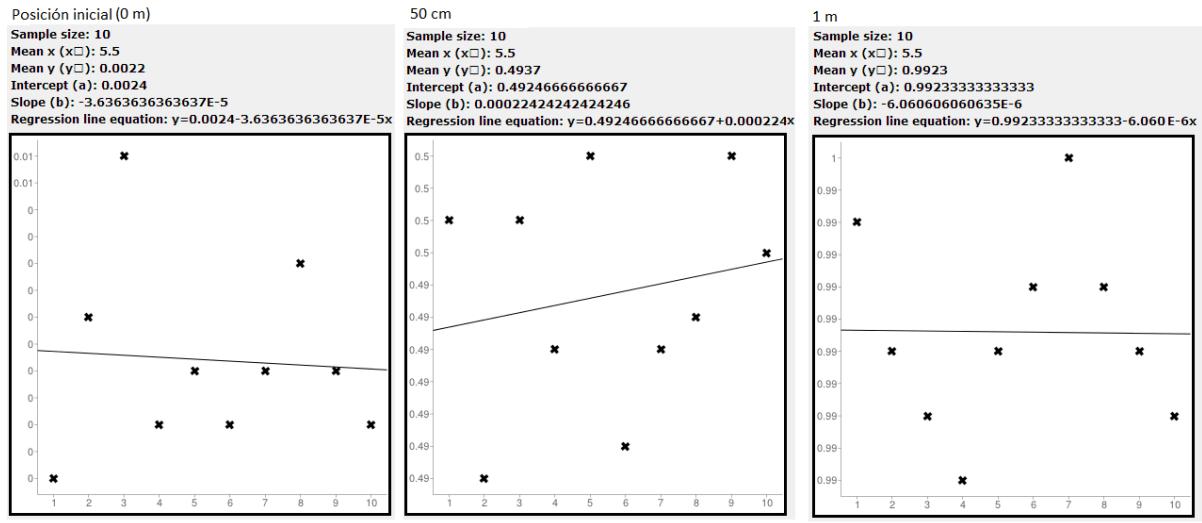


Ilustración 139: Regresión lineal de las mediciones de distancias

Teniendo en cuenta que las pendientes son del orden de las decimas de milímetro, las variaciones en las mediciones no representan un riesgo considerable a la hora de utilizar los valores obtenidos mediante la odometría visual.

11.5.3.1.1.2 Rotaciones de la cámara

De la misma forma que se haría con un transportador, se marcaron los ángulos sobre una superficie y sobre ésta se roto la cámara. A continuación se ve una secuencia de imágenes que muestran la salida de la odometría para dichas rotaciones.

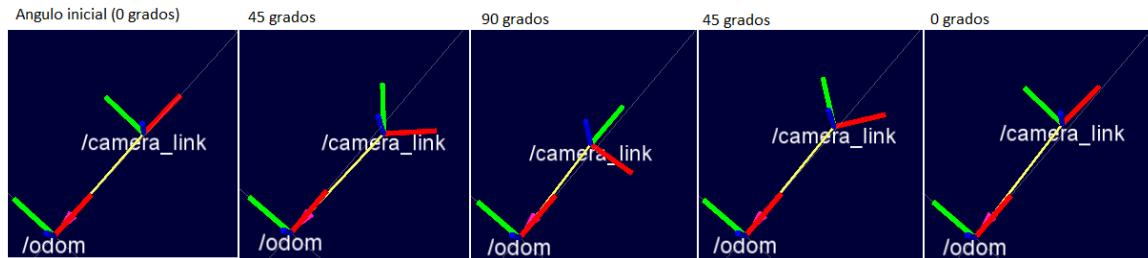


Ilustración 140: Medición de las rotaciones mediante la odometría visual

11.5.3.2 *Mediciones sobre el mapa*

Por último, se realizan mediciones sobre los mapas obtenidos. Gracias al visualizador de ROS, Rviz, es posible medir distancia entre los puntos de una nube de puntos. Para hacerlo, simplemente se elige la opción “measure” y se seleccionan los puntos en cuestión. Las siguientes pruebas se realizaron dentro de una oficina del LAC y, en ella, se midió la distancia entre los bordes de una puerta y, también, el largo de la pared.

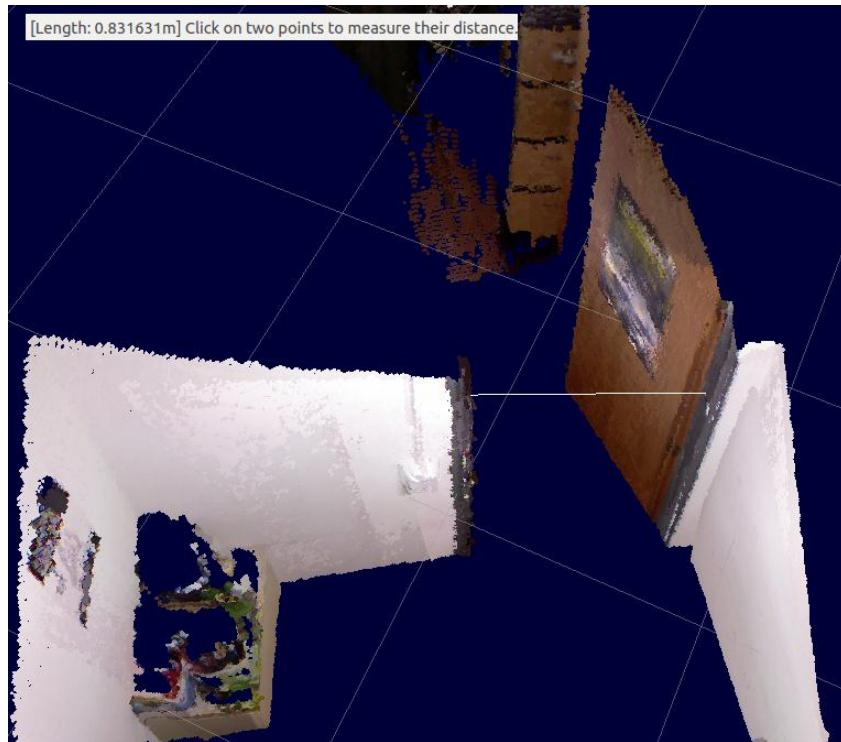


Ilustración 141: Medición de distancias sobre un mapa. Ancho de la puerta.

En la parte superior de la imagen, se ve la distancia medida, que, en este caso, es de 0.83 metros. La línea blanca que va desde un borde de la puerta hasta el otro, representa la distancia medida. Para comparar con la realidad, se muestra una foto de la puerta y se posicionó una regla sobre ella.



Ilustración 142 Medición sobre los mapas: foto real de la puerta

La imagen muestra que la apertura de la puerta es de, aproximadamente, 0.85 metros. Es decir, teniendo en cuenta que la medición sobre el mapa indica 0.83 metros, el error obtenido es del 2% aproximadamente.

Como se anticipó, también se realizó la medida del largo de la pared de la oficina:

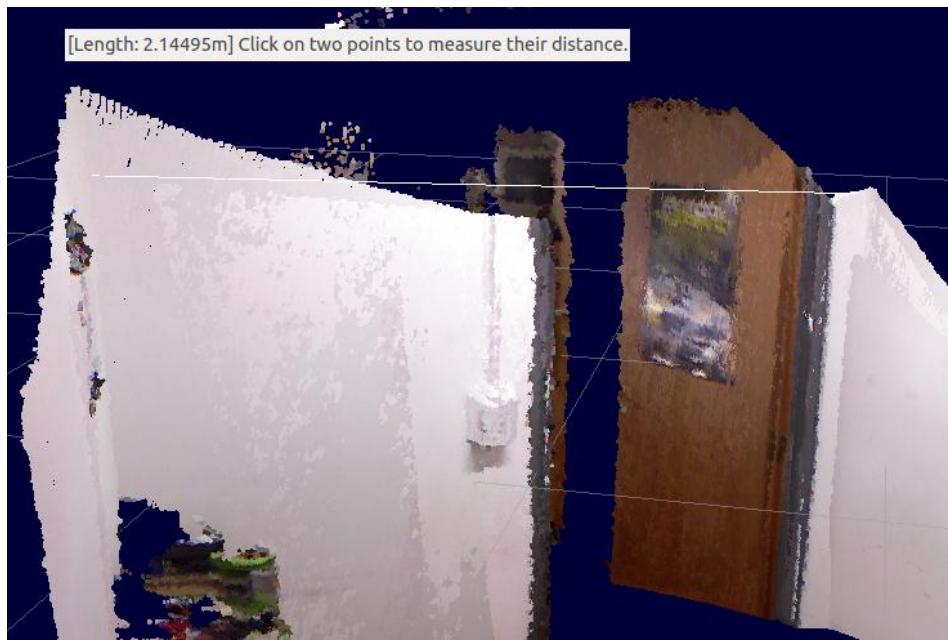


Ilustración 143: Medición sobre los mapas, pared de la oficina

La imagen nos muestra una medida de 2.14 metros. Mediante métodos más tradicionales, con una cinta métrica, se obtuvo un valor de 2.10 metros. En este caso el error es casi del %1,9.

Se concluye que el error es insignificante para los fines de este trabajo.

11.5.4 Integración con el sistema

Una vez obtenido el mapa, este es usado para guiar la navegación sobre el entorno y así continuar la exploración y el mapeo. Ya que el robot solamente se desplaza sobre un plano, para la navegación es preciso eliminar la dimensión “altura” presente en los mapas producidos anteriormente.

El proceso de mapeo y exploración inicia con la rotación de 360 grados del robot para obtener un primer mapa de la zona. De esta forma, a partir de los keyframes del mapa 3D, se proyecta un mapa 2D, basado en el plano a la altura de la cámara del robot, en el cual se marcan los obstáculos y paredes con el valor “1”. Los espacios libres se muestran con “0”. La ubicación de la cámara se muestra con un “9”. Cada elemento de la matriz representa un cuadrado de 10 cm x 10 cm. Recordar que Kinect es capaz de detectar profundidad hasta 4 metros.

A continuación se muestra una imagen del gráfico computacional de ROS que muestra la conexión entre el mapeo 3D y el mapeo 2D.

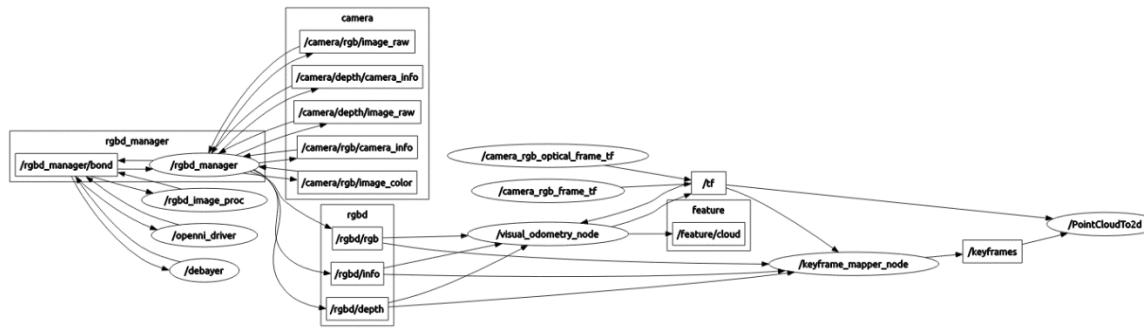


Ilustración 144: Grafo de ROS, conexión del nodo encargado de generar el mapa 2d a partir del mapa 3d

El nodo que se describirá a continuación es el denominado “PointCloudTo2d”. El nombre fue elegido ya que su tarea es transformar la nube de puntos 3D a una matriz 2D. Cabe destacar que dicho nodo recibe tanto el mapa 3D (/keyframes) como las coordenadas de la cámara (/tf) para lograr una correcta representación del mapa a medida que el robot se mueve.

Se muestra como ejemplo, el mapa 2D de la habitación usada anteriormente en la Ilustración 127: Mapeo 3D de la habitación con el método .

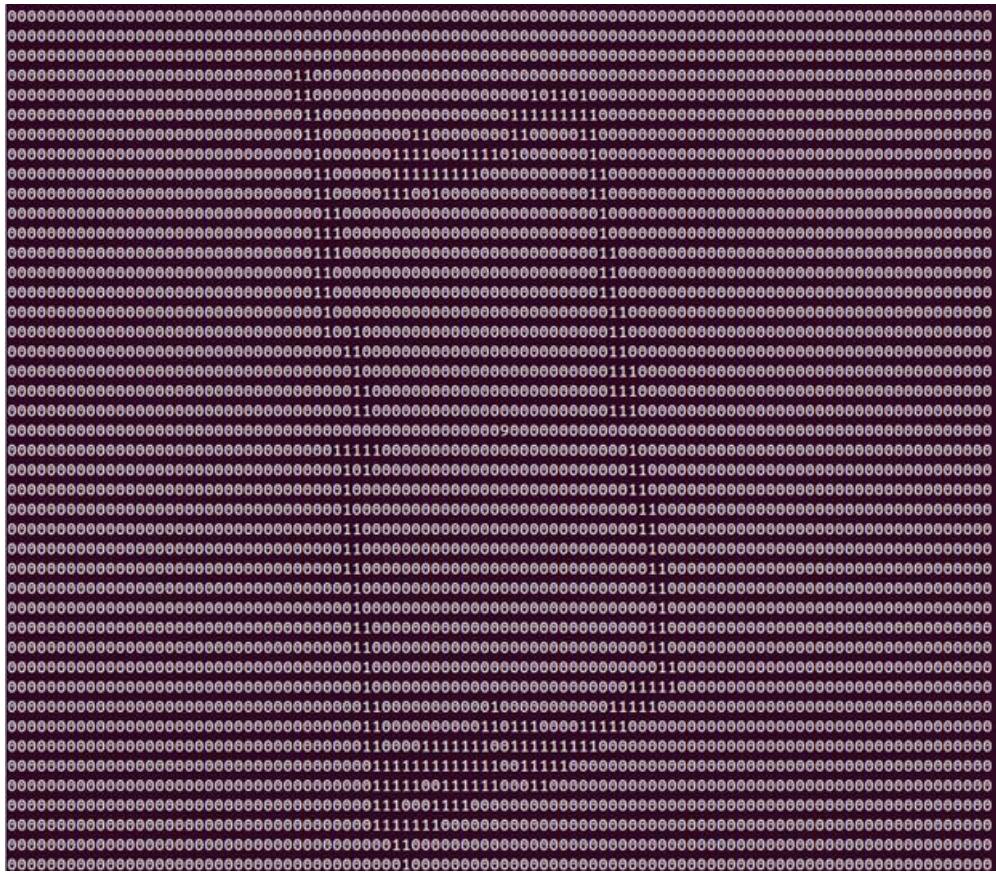


Ilustración 145: Mapa 2d de la habitación. Los 1 representan paredes y obstáculos

Es posible apreciar que mantiene la forma real de la habitación. También se distingue el espacio correspondiente a la puerta abierta. Para que el robot continúe la exploración, debe desplazarse hacia nuevas ubicaciones todavía sin mapear. Para esto, se deben detectar puertas, espacios abiertos hacia otros ambientes y zonas lejanas no visibles desde la cámara. En este

último caso, el robot simplemente debe acercarse hasta poder apreciar la zona lejana para agregarla al mapa.

Para evitar la colisión del robot con objetos y paredes, es necesario tener en cuenta las dimensiones del mismo. Es decir, las órdenes de desplazamiento indicarán la nueva posición a la que debe llegar el centro del robot, pero este último es un cuerpo de 40 cm x 40 cm (viéndolo desde arriba), lo cual implica una necesidad 20 cm de espacio vacío alrededor de su centro.

Para esto, se realiza un proceso de inflación de los obstáculos. Este consiste en envolver los obstáculos con zonas prohibidas marcadas con “2”. Al terminar, se puede observar que, adicionalmente, se han eliminado ciertos agujeros pequeños por los que el robot no era capaz de pasar.

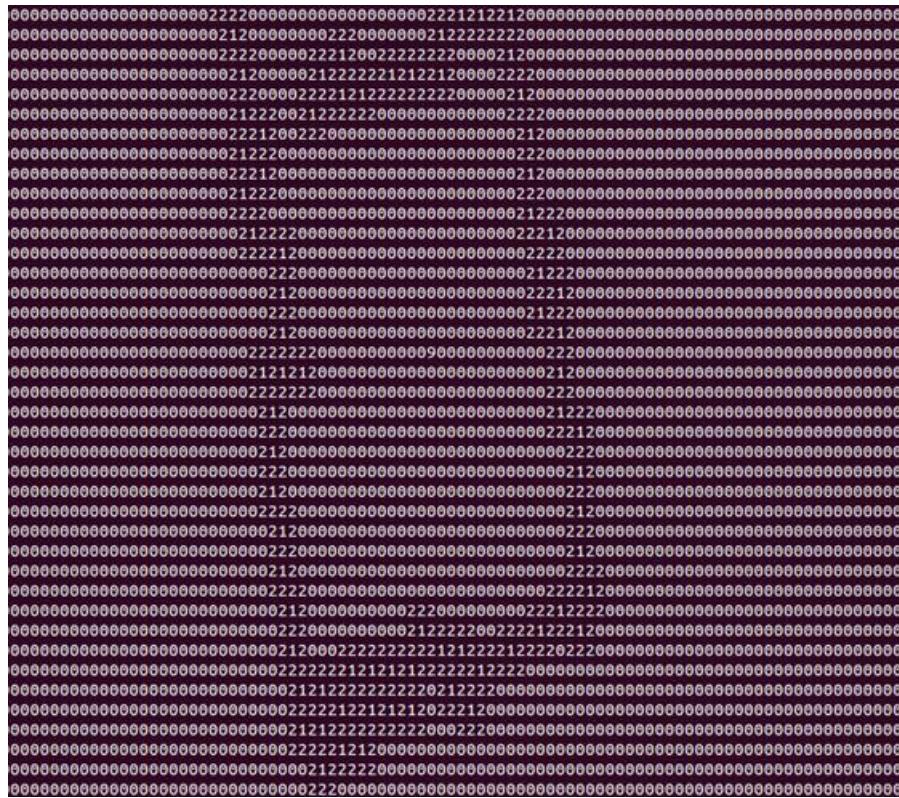


Ilustración 146: Mapa 2d de la habitación. Cada 2 representa la inflación de los obstáculos y paredes

Aun luego de inflar los obstáculos, existe un camino para salir del cuarto. El robot debe continuar su exploración empezando por allí. Para detectar las aberturas y zonas inexploradas, se utilizó una búsqueda por profundidad primero (del inglés Depth-First Search o DFS). Desde el “9”, posición inicial de la cámara, se buscan los caminos posibles que lleven a zonas no exploradas. Cada vez que el DFS encuentra un punto “objetivo”, se los inserta en una pila (stack). El robot irá uno por uno visitando estos objetivos y explorando a partir de ellos. Es importante que los objetivos sean introducidos en una pila ya que el siguiente objetivo a visitar será el último guardado, es decir el más cercano. De esta forma evitaremos ir y volver de manera innecesaria.

A continuación se muestra en la matriz con el número “7” el objetivo que el robot debe explorar a continuación:

Robot móvil autónomo capaz de producir mapas 3D del interior de una habitación

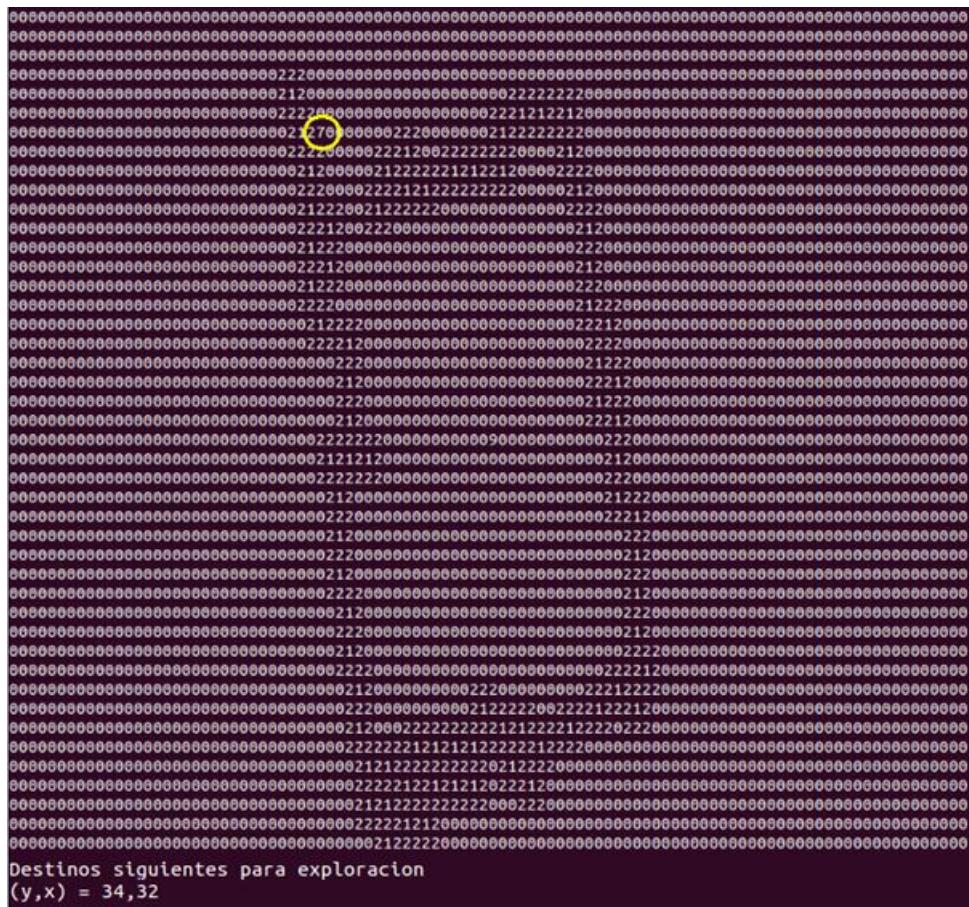


Ilustración 147: Generación de objetivos para la navegación

Con fines demostrativos, se mostrará una secuencia de imágenes que muestra cómo se va llenando la matriz durante una exploración. Se agregó color para lograr un mejor entendimiento. Como en los casos anteriores, 1 representa paredes y obstáculos (rojo), 9 representa la posición de la cámara (amarillo), y 7 los objetivos a explorar (verde) que solo aparecen en la quinta imagen de la secuencia. Cabe destacar que el desplazamiento de la cámara puede visualizarse siguiendo el 9 amarillo.

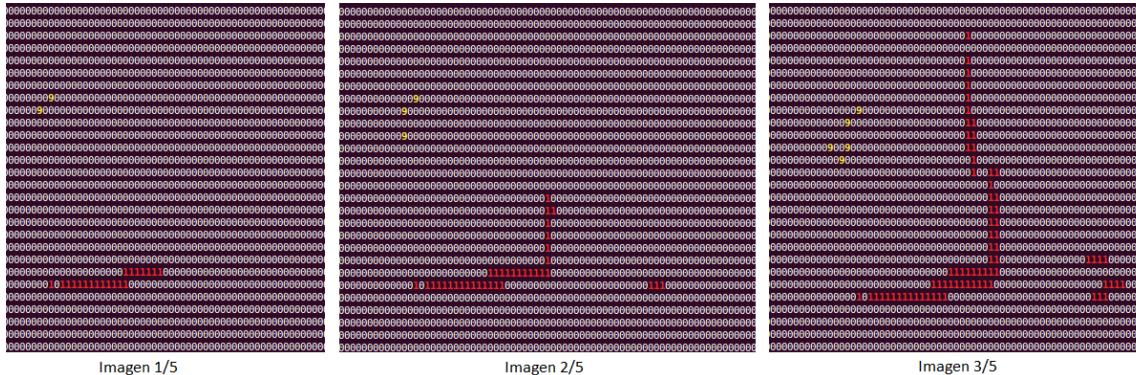


Ilustración 148: Proceso de mapeo en 2d

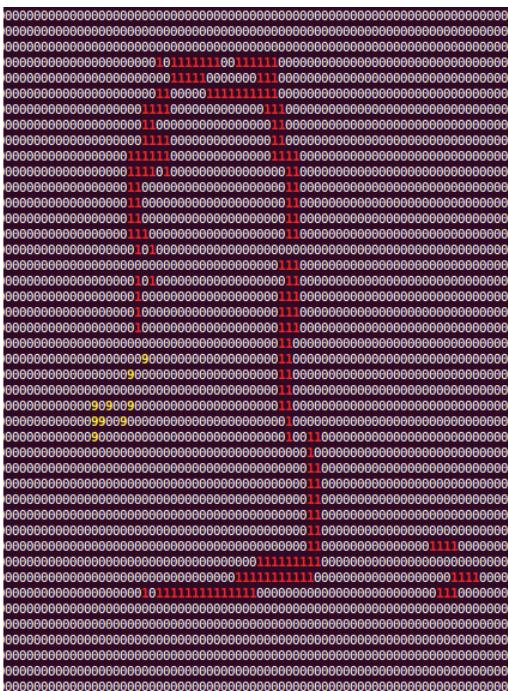


Imagen 4/5

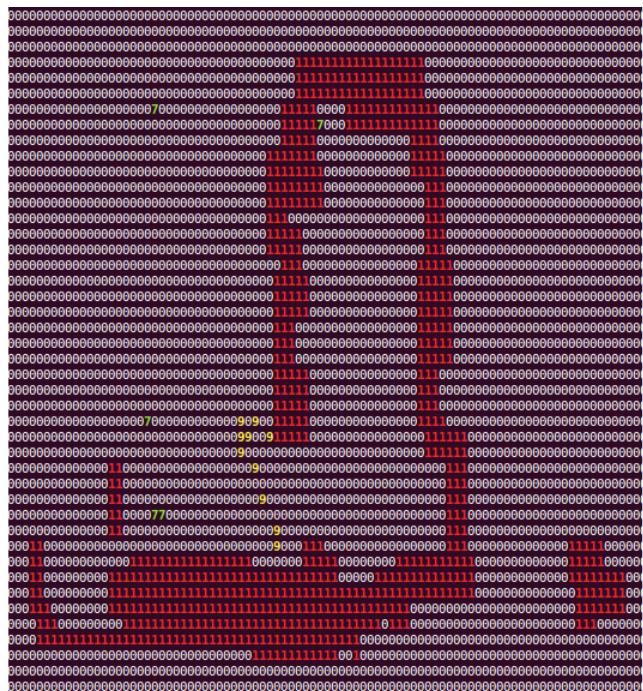


Imagen 5/5

Ilustración 149: Continuación del proceso de mapeo en 2d y la generación simultánea de objetivos

Resumiendo, el proceso de generación de mapas y obtención de objetivos es:

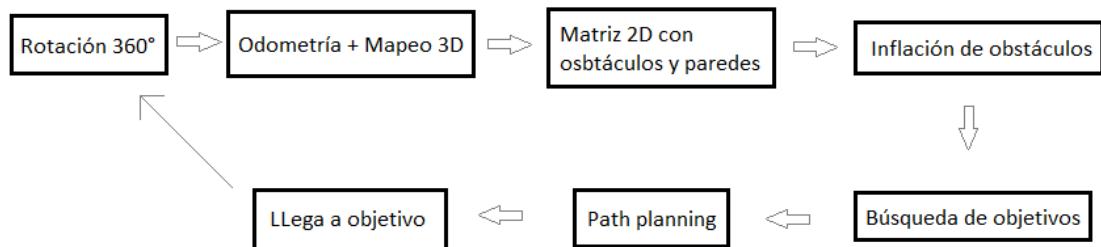


Ilustración 150: Resumen del proceso de exploración

11.5.5 Conclusión

Al inicio del proyecto, se pensaba utilizar directamente los valores de profundidad producidos por Kinect, es decir “en bruto”. Se planeaba realizar el mapeo en dos dimensiones y sin color, marcando en una matriz 2D las posiciones en las que se detectaban obstáculos. Luego de investigar y experimentar, se descubrieron mejores alternativas que permiten generar mapas a color y en tres dimensiones. Estas alternativas utilizan técnicas y procesos que fueron logrados en el contexto de grandes proyectos de investigación que no hubiesen sido posibles desarrollar en el marco de un proyecto integrador. Para aprovechar todo este conocimiento y lograr mapas de mejor calidad, se decidió explorar alternativas desarrolladas bajo licencias que permitan su utilización.

En cuanto a la precisión, las mediciones y experimentos realizados mostraron que el error, tanto durante las traslaciones, como en las rotaciones, es insignificante para los fines de este proyecto.

11.6 Navegación

11.6.1 Planificación de movimientos

El principal problema al que se enfrenta un robot móvil es llegar a su objetivo dentro de una habitación (denotado con el símbolo *). Este debe tener la inteligencia suficiente como para desarrollar un plan que le permita alcanzar el objetivo.

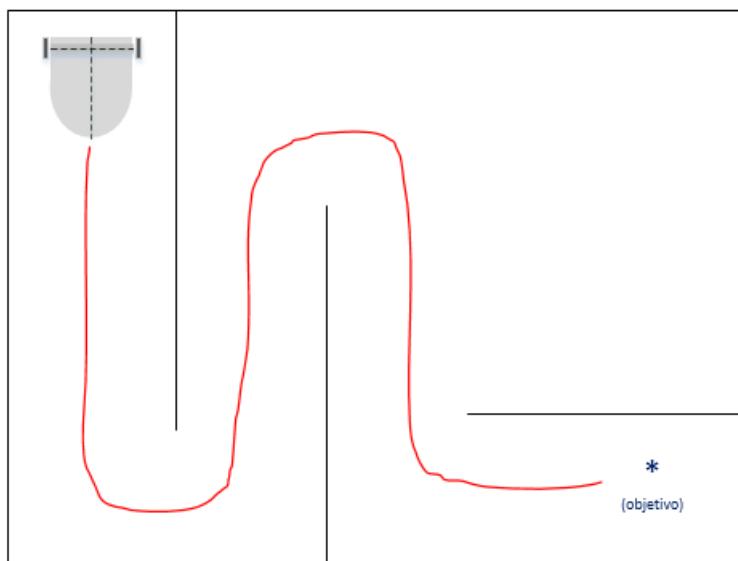


Ilustración 151: Navegación

Haciendo una analogía, el mismo problema se presenta en el caso de los autos robóticos (o self-driving cars en inglés) que son programables y transitan por sí solos, en el medio de una ciudad, que es parte una red de ciudades unidas por autopistas. En este caso, deben lograr una planificación que les permita alcanzar la ciudad objetivo.

En ambos casos, para encontrar el mejor camino hacia un objetivo, el problema se reduce a:

"Dado un mapa, una posición inicial, un objetivo, y un costo de transitar por cada camino alternativo, se debe encontrar el camino con el menor costo".

11.6.2 Algoritmo de búsqueda de caminos

El algoritmo consiste en encontrar el camino más corto desde el punto inicial hasta el objetivo.

El problema del planeamiento se reduce a:

Dados:

- Mapa del mundo donde se moverá el robot
- Posición inicial del robot
- Posición final u objetivo
- Costo de realizar cada movimiento

Objetivo:

- Encontrar el camino de menor costo

Tal como se mencionó anteriormente, el robot vive en un mapa, representado por una matriz, y los movimientos que este puede realizar son 4. Con fines demostrativos, se toma como ejemplo el mapa mostrado a continuación:



Ilustración 152: Mapa de ejemplo y movimientos posibles para el robot

El mapa arreglo bidimensional que puede ser interpretado como un grafo en el que cada elemento o coordenada representa un nodo del mismo. El estado inicial del robot es [0,0] y el estado final es el [4,5].

Los cuadrados azules indican lugares por donde el robot no puede transitar, como paredes u obstáculos, mientras que los cuadrados son transitables.

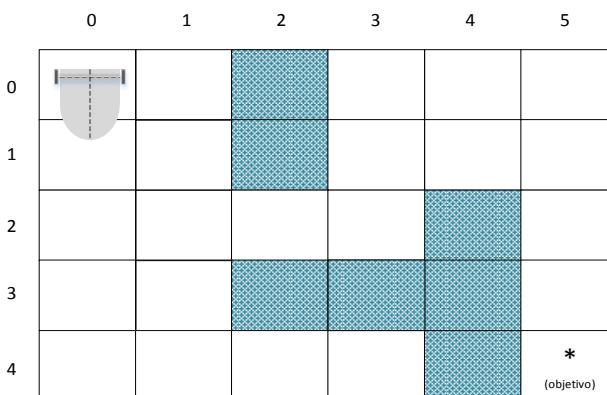


Ilustración 153: Representación del robot dentro de un mapa 2d

11.6.2.1 BFS (Breadth-first search)

Este algoritmo utiliza una lista que se llamará “open list” y en esta se guardarán los nodos visibles cada vez que se haga una expansión hacia un nuevo nodo. Dicha expansión se basa en una estrategia de búsqueda sobre grafos llamada BFS (Breadth-first Search) la cual permite calcular distancias desde un nodo en un grafo, hacia todos los demás. Para ello se elige un nodo, que será llamado raíz o estado inicial y, luego, se realiza una expansión hacia todos sus vecinos. En el siguiente paso se visita a los vecinos de los vecinos, y así sucesivamente hasta recorrer todos los nodos. La búsqueda se limita esencialmente a dos operaciones:

- Visitar un nodo del grafo.

- b. A partir de éste poder visitar sus nodos vecinos.

La siguiente imagen muestra el orden en el que los nodos son expandidos:

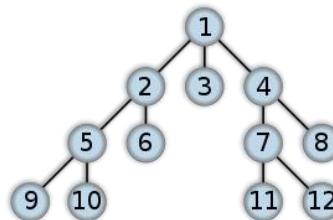


Ilustración 154: Árbol de exploración BFS (Wikipedia - BFS)

Además, se tiene un costo asociado a cada nodo denominado valor-g, que se incrementa en cada expansión. Este valor lleva la cuenta de cuántas veces es necesario expandirse desde el estado inicial para alcanzar el estado actual. Al final del algoritmo, el valor g será el largo del camino óptimo.

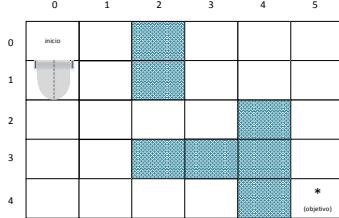
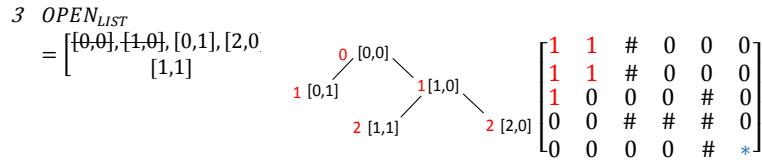
El siguiente cuadro muestra la ejecución paso a paso del algoritmo para el ejemplo mencionado. En la tabla se muestra la lista “open_list” que contiene los nodos de cada expansión, un árbol que muestra cómo se va recorriendo el grafo y lleva la cuenta del valor-g en cada iteración, una matriz que representa el estado de cada nodo (si fue visitado o no), y en la última columna, un gráfico que muestra el camino óptimo que deberá recorrer el robot para llegar a su objetivo.

OPEN LIST	ARBOL DE EXPANSION	MATRIZ	GRAFICO (indica estado actual)
1 $OPEN_{LIST} = [[0,0]]$	0 [0,0]	$\begin{bmatrix} 1 & 0 & \# & 0 & 0 & 0 \\ 0 & 0 & \# & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \# & 0 \\ 0 & 0 & \# & \# & \# & 0 \\ 0 & 0 & 0 & 0 & \# & * \end{bmatrix}$	<p>Este cuadro ilustra la ejecución del algoritmo BFS. La parte superior contiene tablas para la lista abierta (OPEN LIST), el árbol de expansión y la matriz de estados. La parte inferior es un gráfico de 5x5 que muestra el mapa y el camino óptimo. El mapa incluye un robot en la posición (0,0) y un objetivo en la posición (4,4). El camino óptimo es representado por cuadrados sombreados, comenzando en (0,0) y extendiéndose hasta el objetivo.</p>

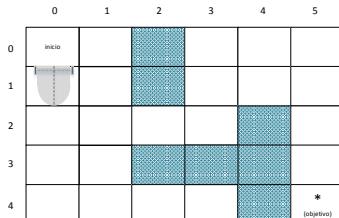
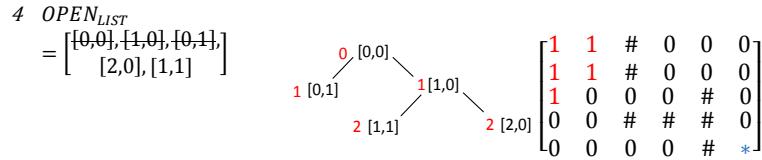
Al inicio solo se tiene el estado [0,0] en la lista. Para no volver a ese estado se marca el elemento en la matriz con un 1. Luego se comprueba si el estado [0,0] es el estado final. Como no lo es, hay que expandirse al próximo estado. El valor-g es 0 ya que para ir desde el nodo inicial hasta el nodo inicial se necesitan 0 expansiones.

2 $OPEN_{LIST}$ = $[[0,0], [1,0], [0,1]]$	<p>Este cuadro muestra el estado actualizado del algoritmo. La lista abierta ahora incluye los estados vecinos [1,0] y [0,1]. El árbol de expansión muestra las rutas desde el nodo inicial hasta estos estados. La matriz de estados muestra que los estados vecinos han sido marcados como visitados (valor-g 1). El gráfico muestra el mapa y el camino actualmente explorado.</p>	$\begin{bmatrix} 1 & 1 & \# & 0 & 0 & 0 \\ 1 & 0 & \# & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \# & 0 \\ 0 & 0 & \# & \# & \# & 0 \\ 0 & 0 & 0 & 0 & \# & * \end{bmatrix}$	
--	---	--	--

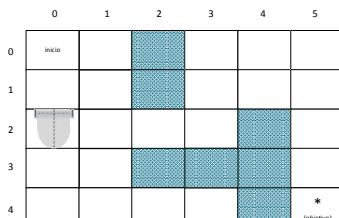
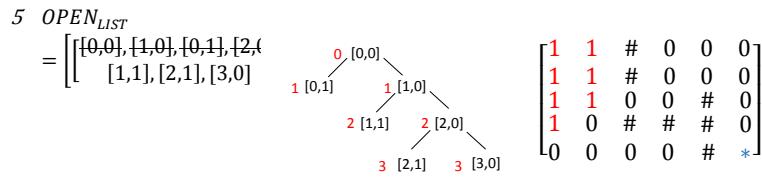
Se quita el [0,0] de la lista, se agregan los nodos vecinos [1,0] y [0,1], y se marcan en la matriz como visitados. El valor-g para llegar a cualquiera de los dos estados es 1, entonces hay dos estados posibles para expandir. Como política de implementación del algoritmo, siempre se expande el estado con el menor valor-g (camino más corto) pero en este caso son equivalentes, se podría expandir cualquiera de los dos. Se opta por expandir el estado [1,0].



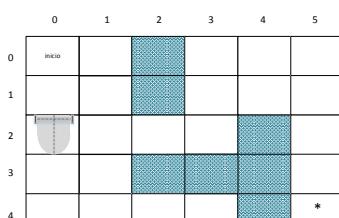
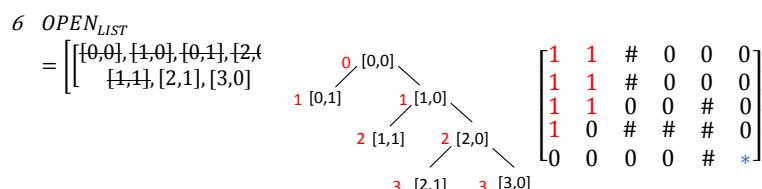
El estado $[1,0]$ se elimina de la lista, y se buscan sus vecinos, el $[0,0]$, $[1,1]$ y $[2,0]$. Pero solo se agregan el $[2,0]$ y $[1,1]$ ya que el $[0,0]$ ya fue visitado. Los que fueron agregados se marcan en la matriz como visitados. El valor-g para llegar a estos nodos es 2. Ahora se toma de la lista, el estado con el menor valor-g que es el $[0,1]$.



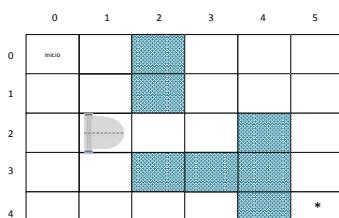
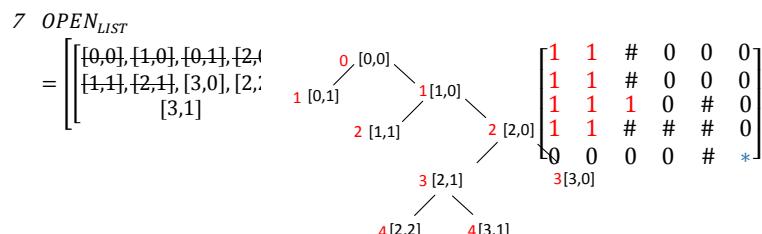
El estado $[0,1]$ no tiene posibilidad de expandirse hacia ninguno de sus vecinos ya los estados $[0,0]$ y $[1,1]$ están marcados como visitados, y el estado $[0,2]$ es la pared ($\#$). Entonces se quita de la open list y se busca el próximo estado con el menor valor-g para expandirse. Los nodos $[2,0]$ y $[1,1]$ tienen el mismo valor-g. Se elige el $[2,0]$ (aunque podría haberse elegido el restante también).



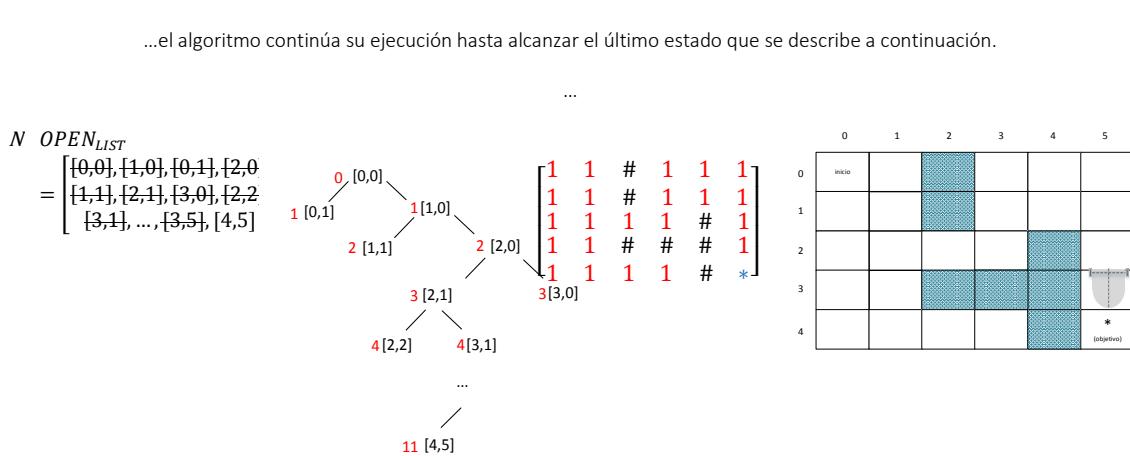
Al expandir el nodo $[2,0]$, se quita de la lista, y se agregan los nodos $[2,1]$ y $[3,0]$ con un valor-g de 3. En la matriz se marcan con 1 indicando que fueron visitados. El nodo con el menor valor-g en la lista es el $[1,1]$ por lo que se procede a expandirlo.



Los vecinos del estado $[1,1]$ ya fueron visitados o son paredes por lo que directamente se elimina de la lista y se marca como visitado en la matriz. Luego se procede a expandir el nodo $[2,1]$.



Los nodos vecinos del [2,1] son el [1,1] y [2,0] ya visitados, y el [2,2] y [3,0] que se agregan a la lista. A su vez se elimina el [2,1]. El valor-g para estos dos nuevos nodos es de 4.



Finalmente, al alcanzar el estado [3,5], el único nodo que queda por expandir es el [4,5] y al comprobar si es el estado final (esto debió haberse hecho en cada estado) la respuesta es afirmativa por lo que el robot ha alcanzado su objetivo. Como se puede observar en el árbol, el valor-g del estado final es 11 lo cual implica que el robot puede llegar a su objetivo en 11 pasos, y éste será el camino más corto que pueda tomar.

Según se mostró en el ejemplo anterior, el camino óptimo demanda 11 expansiones para alcanzar el objetivo deseado. Entonces, suponiendo que cada elemento de la matriz representa 10cm en un mapa real, el robot deberá recorrer $11 \times 10 \text{ cm} = 1,10 \text{ metros}$ hasta llegar al objetivo.

La implementación del algoritmo escrita en el lenguaje Python se muestra a continuación:

- Init[] → Estado inicial del robot
- Goal[] → Objetivo o estado final
- Grid[][] → Mapa por donde se mueve el robot
- Cost → Costo de realizar cada movimiento
- Delta[] → Lista de movimientos que puede realizar el robot
- Colsed[][] → Matriz de estados o nodos visitados
- Action[][] → Matriz donde se guarda el camino óptimo

```

1.     init=[0,0]
2.     goal=[4,5]
3.     grid = [[0,0,1,0,0,0],[0,0,1,0,0,0],[0,0,0,0,1,0],[0,0,1,1,1,0],[0,0,0,0,1,0]]
4.     cost=1
5.
6.     #-----
7.     #   movements
8.     #
9.     delta = [[-1, 0], # go up
10.                [0, -1], # go Left
11.                [1, 0], # go down
12.                [0, 1]] # go right
13.
14.    def search():
15.        #-----
16.        #   print map(grid) size
17.        #

```

```

18.     print "Grid rows: %d" % len(grid)
19.     print "Grid columns: %d" % len(grid[0])
20.
21.     #-----
22.     #    A* algorithm (compute shortest path)
23.     #
24.     closed = [[0 for row in range(len(grid))] for col in range(len(grid[0]))]
25.     action = [[-1 for row in range(len(grid))] for col in range(len(grid[0]))]
26.     grid_map = [[' ' for row in range(len(grid))] for col in range(len(grid[0]))]
27.     closed[init[0]][init[1]] = 1
28.     x = init[0]
29.     y = init[1]
30.     g = 0
31.     open = [[g, x, y]]
32.
33.     found = False # flag that is set when search is complet
34.     resign = False # flag set if we can't find expand
35.
36.     while not found and not resign:
37.         if len(open) == 0:
38.             resign = True
39.             return 'fail'
40.         else:
41.             open.sort()
42.             open.reverse()
43.             next = open.pop()
44.             x = next[1]
45.             y = next[2]
46.             g = next[0]
47.
48.             if ( (x == goal[0]) and (y == goal[1]) ):
49.                 found = True
50.             else:
51.                 for i in range(len(delta)):
52.                     x2 = x + delta[i][0]
53.                     y2 = y + delta[i][1]
54.
55.                     if ( (x2 >= 0) and (x2 < len(grid[0])) and (y2 >=0) and (y2 < len(grid)) ):
56.                         if closed[x2][y2] == 0 and grid[x2][y2] == 0:
57.                             g2 = g + cost
58.                             open.append([g2, x2, y2])
59.                             closed[x2][y2] = 1
60.                             action[x2][y2] = i

```

La complejidad temporal del algoritmo es aproximadamente $O(|V|)$ siendo V el conjunto de nodos del grafo y $|V|$ la cardinalidad del mismo, es decir, la cantidad de nodos.

Como resultado, el algoritmo entrega una matriz de expansión llamada “action” que muestra cómo se recorre el grafo mediante un contador que se incrementa en cada movimiento del robot. Se agrega -1 en los lugares en los que se encontró obstáculos. El resultado de esta matriz no es único, sino que dependiendo de cuantos nodos con el mismo valor-g se tengan, se expandirá de formas diferentes. Por lo tanto, no se espera la matriz sea siempre la misma, pero esto no implica que cambie el resultado final. Para el caso del ejemplo la matriz “action” será:

$$\begin{bmatrix} -1, & -1, & 0, & 3, & 3 \\ 2, & -1, & 0, & 3, & 3 \\ 2, & 3, & 3, & -1, & 2 \\ 2, & -1, & -1, & -1, & 2 \\ 2, & 3, & 3, & -1, & 2 \end{bmatrix}$$

Luego con un código de conversión, se recorre la matriz de expansión “action” y se obtiene el camino óptimo en una matriz llamada “grid_map”, sus coordenadas en un arreglo “path_coord”, y los movimientos del robot en otro arreglo “path_move”. El algoritmo de conversión se muestra a continuación:

```

61.     motions = ['^', '<', 'v', '>']
62.     move = ['up', 'left', 'down', 'right']
63.     #-----
64.     #    get the path in coordinates, robot movements and road map

```

```

65.      #
66.      path_coord = []
67.      path_move = []
68.      x = goal[0]
69.      y = goal[1]
70.      grid_map[x][y] = '*'
71.      path_coord.append([x, y])
72.      while x != init[0] or y != init[1]:
73.          x2 = x - delta[action[x][y]][0]
74.          y2 = y - delta[action[x][y]][1]
75.          x = x2
76.          y = y2
77.          grid_map[x2][y2] = motions[action[x2][y2]]
78.          path_move.append(move[action[x2][y2]])
79.          path_coord.append([x2, y2])
80.
81.      path_move=path_move[::-1]           #reverse List
82.      path_coord=path_coord[::-1]         #reverse List

```

Para el ejemplo se obtuvo:

```

['v', ' ', ' ', ' ', ' ', ' ']
['v', ' ', ' ', '^', '>', '>']
['v', '>', '>', ' ', ' ', 'v']
[' ', ' ', ' ', ' ', ' ', 'v']
[' ', ' ', ' ', ' ', ' ', '*']

path_coord = [ [0, 0],
               [1, 0],
               [2, 0],
               [2, 1],
               [2, 2],
               [1, 2],
               [1, 3],
               [1, 4],
               [2, 4],
               [3, 4],
               [4, 4] ]

path_move = [ 'down' ,
              'down' ,
              'down' ,
              'right' ,
              'right' ,
              'up' ,
              'right' ,
              'right' ,
              'down' ,
              'down' ]

```

El arreglo “path_move” contiene comandos que no se corresponden con la implementación de los movimientos del robot construido en este proyecto, es decir, en el mundo real. Dado que el robot no es holonómico, este no puede desplazarse en las direcciones X e Y independientemente. Es necesario introducir un giro, ya sea left o right, en el medio de un cambio de dirección. Por ejemplo, si el robot está moviéndose hacia la izquierda (left), y quiere comenzar a avanzar en dirección hacia arriba (up), se deberá introducir un “right” para el cambio de dirección. En la implementación se usan 4 comandos: up (avance del robot en la dirección actual), down (retroceso del robot en la dirección actual), right y left (giros de 90 grados en el lugar, en sentido horario y anti horario respectivamente).

Para la traducción se escribió una rutina de conversión de comandos cuya salida consiste directamente en los movimientos a darle al robot:

```
real_path(robot)= [      'init' ,
                      'up'   ,
                      'up'   ,
                      'left' ,
                      'up'   ,
                      'up'   ,
                      'left' ,
                      'up'   ,
                      'right' ,
                      'up'   ,
                      'up'   ,
                      'right' ,
                      'up'   ,
                      'up'   ,
                      'stop' ]
```

Además con la librería matplotlib de Python se crea un mapa en 2D muy simple para una mejor representación de la matriz “grid_map”:

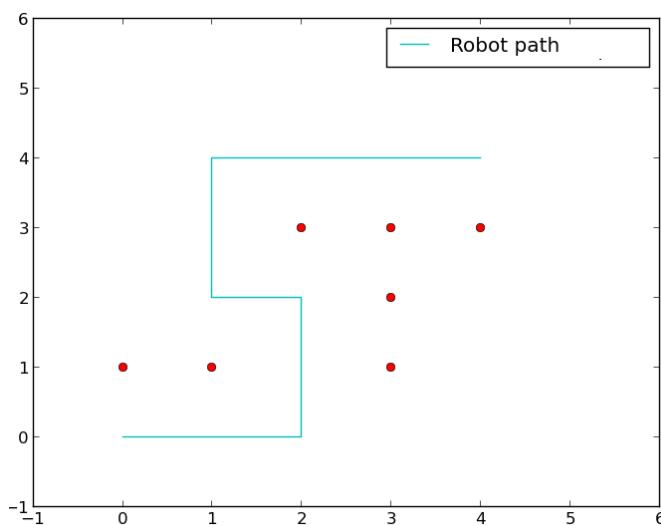


Ilustración 155: Trayectoria planeada

En la figura, los puntos rojos representan los nodos que no son navegables, mientras que en celeste se muestra la trayectoria a seguir hasta alcanzar su objetivo.

11.6.2.2 Algoritmo A*

Una variante más eficiente del algoritmo de búsqueda descripto anteriormente se denomina A* (se lee a estrella). Éste fue desarrollado Nils Nilsson en la Universidad de Stanford en 1968, como una extensión del algoritmo de Dijkstra, brindando una mejor performance a partir del uso de heurísticas que hacen que no sea necesaria la expansión hacia todos los nodos del grafo.

La única diferencia que tiene el A* con el algoritmo presentado anteriormente, es que para expandirse no sólo elige a que nodo hacerlo según el valor-g, sino que también usa una función heurística.

Supongamos el siguiente ejemplo, dado el mapa:

$$grid = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & * \end{bmatrix}$$

Y los nodos de inicio y fin:

$$init = [0,0]$$

$$goal = [4,5]$$

En el algoritmo BFS el resultado de la matriz de expansión es:

$$\begin{bmatrix} -1 & -1 & 0 & 3 & 3 & -1 \\ 2 & -1 & 0 & 3 & 3 & 3 \\ 2 & -1 & 0 & 3 & 3 & 3 \\ 2 & -1 & 0 & 3 & 3 & 3 \\ 2 & 3 & 3 & 3 & -1 & 2 \end{bmatrix}$$

Mientras que en el el A* el resultado es:

$$\begin{bmatrix} -1 & -1 & -1 & -1 & -1 & -1 \\ 2 & -1 & 0 & 0 & -1 & -1 \\ 2 & -1 & 0 & 0 & 0 & 0 \\ 2 & -1 & 0 & 0 & 3 & 3 \\ 2 & 3 & 3 & 3 & -1 & 2 \end{bmatrix}$$

Como se puede ver, mediante A* se llega al objetivo con muchas menos expansiones (todos los valores de la matriz marcados con el valor -1 o 0, son posiciones que no fueron visitadas).

La función heurística jugará un papel importante a la hora de hacer la expansión. Con fines demostrativos, se la implementa a partir de un arreglo bidimensional. Es decir, para cada elemento de la matriz "mapa" se tiene un valor de h.

$$heuristic = \begin{bmatrix} 9 & 8 & 7 & 6 & 5 & 4 \\ 8 & 7 & 6 & 5 & 4 & 3 \\ 7 & 6 & 5 & 4 & 3 & 2 \\ 6 & 5 & 4 & 3 & 2 & 1 \\ 5 & 4 & 3 & 2 & 1 & 0 \end{bmatrix}$$

El valor de h es una estimación de la cantidad de pasos que necesitará realizar el robot para alcanzar el objetivo. Claramente, los valores mostrados no son reales ya que no se están teniendo en cuenta los obstáculos o lugares no transitables. Es decir, la función heurística solo proveerá una estimación de la distancia, el único caso en el que será igual a la distancia euclíadiana hasta el objetivo, será en un mundo sin obstáculos.

Entonces se puede decir que cada elemento de la matriz provee una suposición optimista de cuantos pasos necesita el robot hasta alcanzar su objetivo. Es decir, la función elegida es admisible.

$$h(x, y) \leq distancia\ al\ objetivo\ desde\ x, y$$

Analizando rápidamente el mapa ejemplo, se puede apreciar que para llegar al objetivo desde el inicio, se necesitan más de 9 pasos debido a los obstáculos que se interponen entre medio (se necesitan 11 pasos).

Para añadir al cálculo la función heurística a la búsqueda, la modificación sobre el algoritmo BFS es muy simple. La nueva estrategia de decisión en la expansión, se basa en elegir el nodo con el menor valor-f que ya contiene el valor-g. Entonces, en cada nodo además de llevar la cuenta del valor-g, se usa un valor-f que es:

$$f = g + h(x, y)$$

Donde g =distancia desde el nodo inicial hasta el actual
 $h(x,y)$ = estimación de la distancia desde el nodo actual hasta el final

Para la implementación del algoritmo en Python simplemente es necesario agregar como entrada la matriz heurística, y cambiar la línea donde se hace el cálculo del costo:

57. `g2 = g + cost`

Por la siguiente que incluye la estimación:

57. `g2 = g + heuristic[x2][y2] + cost`

La complejidad temporal del A* depende de la heurística. En el peor caso, la cantidad de nodos expandidos es exponencial en el largo del camino más corto, es decir es $O(n*n)$ siendo n el largo del camino óptimo.

A continuación se muestra un ejemplo:

$$grid = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & * \end{bmatrix} \quad heuristic = \begin{bmatrix} 9 & 8 & 7 & 6 & 5 & 4 \\ 8 & 7 & 6 & 5 & 4 & 3 \\ 7 & 6 & 5 & 4 & 3 & 2 \\ 6 & 5 & 4 & 3 & 2 & 1 \\ 5 & 4 & 3 & 2 & 1 & 0 \end{bmatrix}$$

Desde el nodo inicial hasta el nodo [4,1] no hay opción de elegir ya que la expansión está limitada por la pared. Entonces supongamos que se está en el nodo [4,1] el cual podría expandirse a los vecinos [4,0] y [4,2].

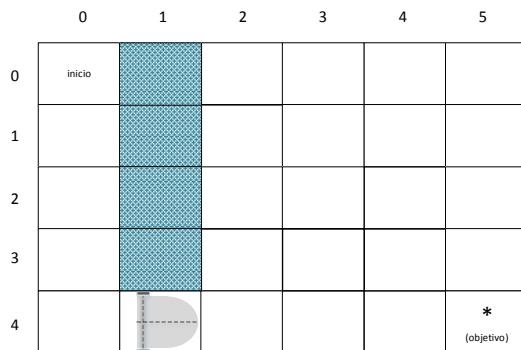
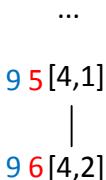
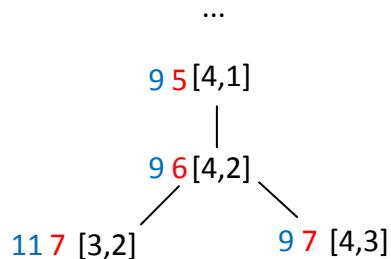


Ilustración 156: Avance del robot siguiendo la trayectoria

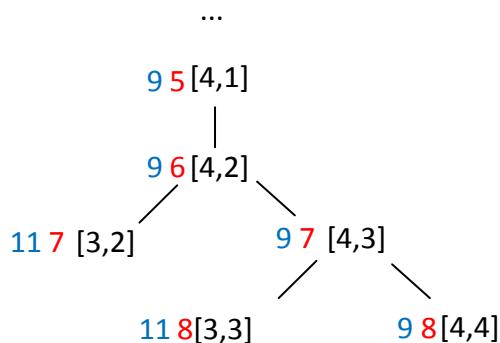
El nodo [4,0] ya fue visitado por lo que se expande al nodo [4,2]. El valor en azul muestra el valor-f que es la suma entre valor-g + heuristic[4][1]=5+4=9. Es decir, la función heurística estima que desde el punto [4,1] se llegará al objetivo en 4 pasos.



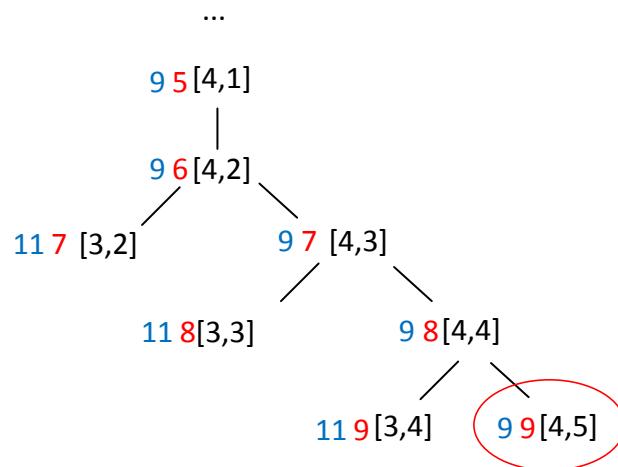
Ahora, en el nodo debe elegir si expandirse al [3,2] o al [4,3]. Aquí es donde el A* hace la diferencia respecto del BFS. Dado que ambos estados tienen el mismo valor-g, se necesita una nueva medida que permita diferenciar a cuál de los dos expandirse. El el estado [4,3] tiene un valor-f menor, ya que la heurística estima que está más cerca del objetivo, por lo que se expande hacia ese nodo.



Ahora en el nodo [4,3] se tiene la posibilidad de expansión hacia los nodos [3,3] y [4,4]. Se elige el [4,4] ya que el valor-f lo indica.



Y finalmente, los nodos vecinos del [4,4] son el [3,4] y el [4,5] y como éste último tiene el menor valor-f, se alcanza ese estado que es el objetivo que se quería alcanzar.



Según se observa en el árbol, se alcanza el objetivo con un valor-g de 9 que implica que ese es el costo de ir desde el estado inicial hasta el final. Y queda en evidencia que hay muchos otros nodos en la matriz que no fueron visitados ya que no forman parte del camino.

Entonces se concluye en que el A* es muy similar al BFS y se llega al mismo resultado, pero gracias a la información adicional (proveniente de una estimación) que guía la búsqueda, se obtiene una mejor performance. En caso de que la matriz heurística contenga todos 0, el algoritmo se comporta de la misma forma que el BFS.

11.6.2.3 Algoritmo basado en programación dinámica

Este algoritmo, al igual que el BFS y el A* permite encontrar el camino óptimo. La diferencia con los dos primeros es que dado un mapa y un objetivo, calcula el mejor camino hacia él desde todos los puntos del mapa.

En el mundo real, estocástico, las acciones son no deterministas, por ejemplo el robot podría sufrir un desliz, tomar un camino en el que hay otro robot, o una persona, u otro ente, etc. Hasta ahora, no se había considerado esto, pero en la realidad puede ocurrir. Entonces, no solo se debe hacer un plan para la posición más probable, sino para cualquier posición del mapa en la que el robot se pueda encontrar.

La programación dinámica es una técnica que permite resolver problemas complejos, a través de la división en subproblemas más simples (Wikipedia - Dynamic programming). A su vez estos problemas se dividen en otros subproblemas hasta alcanzar un caso base fácilmente calculable. Luego, a partir de este, se construyen las soluciones para los subproblemas más grandes. Además, se disminuye la cantidad de cálculos necesarios guardando el resultado de cada instancia del problema. Es decir, se cachean los valores para que no sea necesario recalcularlos.

Esta técnica solo es aplicable a problemas que tengan una subestructura óptima y subproblemas superpuestos. Permite disminuir considerablemente el tiempo de ejecución respecto a otros métodos que no toman ventaja de la superposición de subproblemas (como la búsqueda con BFS).

Por ejemplo para el siguiente mapa:

$$grid = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & * \end{bmatrix}$$

La salida que se debería obtener es:

```
[['v', '#', 'v', 'v', 'v', 'v']
 ['v', '>', '>', '>', '>', '*']]
```

Como se puede ver ya no solo se calcula el camino desde el nodo inicial, sino que tomando cualquiera de los como inicial se puede llegar al objetivo por el mejor camino.

La forma de implementar este algoritmo de manera eficiente es a partir de una función que asocie a cada elemento de la matriz el valor del camino más corto desde ese elemento hasta el objetivo.

Inicialmente, la matriz contiene el valor 99 (representa infinito) para los lugares no transitables, y 0 para los demás elementos. Luego cada valor (x,y) se calcula de forma recursiva tomando el valor óptimo (más pequeño) de los vecinos (x',y') y añadiendo el costo de llegar a ese elemento. Si el costo es igual a 1, la función de actualización de cada elemento es:

$$f(x, y) = \min(x', y') + 1$$

Aplicando esta función recursivamente comenzando desde el objetivo, se obtiene la matriz donde cada elemento contiene el valor de llegar desde ese punto al objetivo.

Tal como se dijo anteriormente, el cálculo de cada elemento se basa en sumarle 1 a otro elemento calculado previamente. Esto permite ahorrar tiempo de cálculo y mejora la performance del algoritmo.

La implementación del algoritmo en Python se muestra a continuación:

```
77. grid = [[0, 1, 0, 0, 0, 0],
78.           [0, 1, 0, 0, 0, 0],
79.           [0, 0, 0, 0, 1, 0],
80.           [0, 1, 1, 1, 1, 0],
81.           [0, 0, 0, 0, 1, 0]]
82.
83. init = [0, 0]
84. goal = [len(grid)-1, len(grid[0])-1]
85.
86. delta = [[-1, 0], # go up
87.           [0, -1], # go left
88.           [1, 0], # go down
89.           [0, 1]] # go right
90.
91. delta_name = ['^', '<', 'v', '>']
92.
93. cost_step = 1 # the cost associated with moving from a cell to an adjacent one.
94.
95. def optimum_policy():
96.     value = [[99 for row in range(len(grid[0]))] for col in range(len(grid))]
97.     policy = [[' ' for row in range(len(grid[0]))] for col in range(len(grid))]
98.
99.     change = True
100.    while change:
101.        change = False
102.
103.        for x in range(len(grid)):
104.            for y in range(len(grid[0])):
105.                if goal[0] == x and goal[1] == y:
106.                    if value[x][y] > 0:
```

```

107.         value[x][y] = 0
108.         policy[x][y] = '*'
109.         change = True
110.
111.     elif grid[x][y] == 0:
112.         for a in range(len(delta)):
113.             x2 = x + delta[a][0]
114.             y2 = y + delta[a][1]
115.
116.             if x2 >= 0 and x2 < len(grid) and y2 >= 0 and y2 < len(grid[0]) and grid[x2][
117.                 y2] == 0:
118.                 v2 = value[x2][y2] + cost_step
119.
120.                 if v2 < value[x][y]:
121.                     change = True
122.                     value[x][y] = v2
123.                     policy[x][y] = delta_name[a]
124.     for i in range(len(value)):
125.         print policy[i]
126.
127.     return policy

```

Para el mapa de ejemplo:

$$grid = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & * \end{bmatrix}$$

El resultado que se obtiene es:

```

['v', ' ', 'v', 'v', 'v', 'v']
['v', ' ', '>', '>', '>', 'v']
['>', '>', '^', '^', ' ', 'v']
['^', ' ', ' ', ' ', ' ', 'v']
['^', '<', '<', '<', ' ', '*']

```

Como se observa, cada elemento de la matriz tiene el camino óptimo hasta el objetivo.

Existe otra variante de este algoritmo con 3 valores diferentes de costo para cada movimiento, left, up y right. Entonces según el valor que se le dé el robot elegirá ir por uno u otro camino. Este algoritmo se usa en algunos sistemas logísticos donde el costo de realizar algunos movimientos es más caro en tiempo que otros. Por ejemplo se podría elegir que el robot doble siempre a la derecha evitando cruzar de carril. Esto es útil si se transita por un pasillo que es doble mano, es decir, en caso que otros vehículos transiten en sentido contrario.

11.6.3 Elección del algoritmo

Las implementaciones de los 3 algoritmos ya fueron presentadas. A la hora de elegir uno de ellos para implementar en el robot, se optó por BFS (Breadth-first search) debido a que por el momento el robot no tiene requerimientos de alto rendimiento, y este algoritmo permite ahorrar el cálculo de la función heurística para cada nuevo objetivo al que se quiera llegar. Si bien por ahora se utiliza este, para pasar a implementar el A* sólo bastaría con calcular la matriz heurística y cambiar una sola línea de código.

En relación al algoritmo basado en programación dinámica, se elige el BFS ya que la entrada al planificador contiene el punto inicial en el que el robot se encuentra, por lo que no es necesario conocer el camino para llegar al objetivo desde otras posiciones. De elegir el otro, se estaría ocupando capacidad de cálculo en algo que luego no se va a ser usado.

11.6.4 Test de navegación

Una vez obtenido el mapa 3D, este es usado para guiar la navegación sobre el entorno y así continuar la exploración y el mapeo. Ya que el robot solamente se desplaza sobre un plano, para la navegación se utiliza un mapa en 2D obtenido mediante una transformación del mapa 3D según se explicó en la sección 11.5.4.

A continuación se muestran las tarjetas de los test que se realizarán:

TEST	
Requerimiento	FSR8) Navegación. Movimientos autónomos a partir de mapas.
Prueba	ST8.1) comprobar la exploración de la habitación
Precondición	Robot alimentado por batería cargada Kinect y OpenNI en funcionamiento Aplicación ejecutándose en Arduino Aplicación ejecutándose en la computadora a bordo ROS y Rviz ejecutándose
Poscondición	Mismas que las precondiciones
Resultado obtenido	Se generan puntos a explorar y comandos el movimiento del robot hasta ellos
Resultado esperado	Se generan puntos a explorar y comandos el movimiento del robot hasta ellos
Conclusión	PASADO

Tabla 24: Tarjeta testing FSR8 ST8.1

TEST	
Requerimiento	FSR8) Navegación. Movimientos autónomos a partir de mapas.
Prueba	ST8.2) Evitar obstáculos que permitan circulación
Precondición	Robot alimentado por batería cargada Kinect y OpenNI en funcionamiento Aplicación ejecutándose en Arduino Aplicación ejecutándose en la computadora a bordo ROS y Rviz ejecutándose
Poscondición	Mismas que las precondiciones
Resultado obtenido	El robot llega hasta el objetivo sin producirse colisiones
Resultado esperado	El robot llega hasta el objetivo sin producirse colisiones
Conclusión	PASADO

Tabla 25: Tarjeta testing FSR8 ST8.2

TEST	
Requerimiento	FSR8) Navegación. Movimientos autónomos a partir de mapas.
Prueba	ST8.3) Evitar obstáculos que no permitan circulación
Precondición	Robot alimentado por batería cargada Kinect y OpenNI en funcionamiento Aplicación ejecutándose en Arduino Aplicación ejecutándose en la computadora a bordo ROS y Rviz ejecutándose
Poscondición	Mismas que las precondiciones
Resultado obtenido	El robot no intenta circular por espacios limitados
Resultado esperado	El robot no intenta circular por espacios limitados
Conclusión	PASADO

Tabla 26: Tarjeta testing FSR8 ST8.3

11.6.4.1 Test habitación cerrada

La primera prueba se llevó a cabo sobre el mapa 3D de una habitación cerrada. El mapa 3D se muestra a continuación:

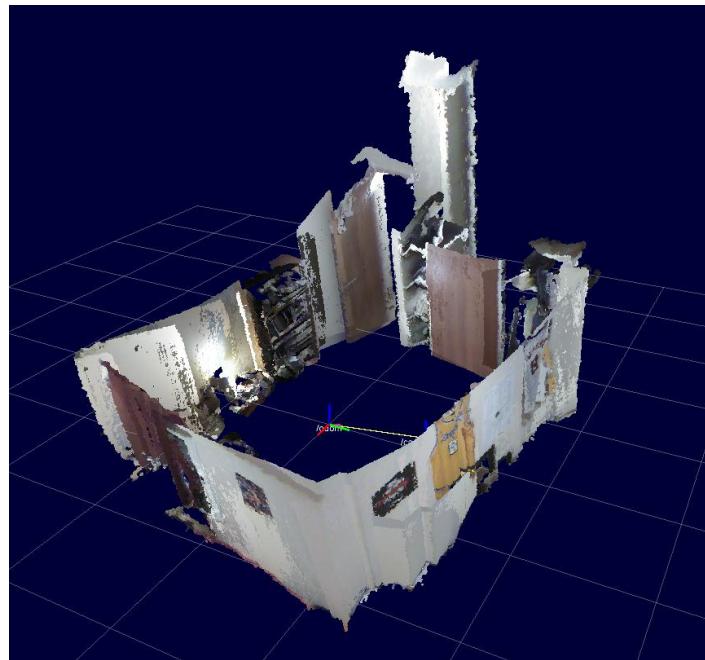


Ilustración 157: Mapeo 3d completo de una habitación

A partir de este se obtiene otro en 2D, representado mediante una matriz de 100x100 elementos en la cual cada elemento representa 10cm del lugar. Este es la entrada al módulo planificador, el cual calculará el camino óptimo, desde un punto inicial hasta un objetivo. Ambos, inicio y objetivo son dados como entrada al planificador. A continuación, se muestra un fragmento del mapa en 2D donde se tiene la información más relevante. El punto inicial es el elemento marcado con un círculo azul, mientras que el objetivo, puerta de salida hacia otro entorno a explorar, se marca en rojo:

Robot móvil autónomo capaz de producir mapas 3D del interior de una habitación

Ilustración 158: Exploración. Objetivo a alcanzar.

El resultado del test se muestra abajo:

```

#-----#
#    Test1
#
init=[50,50]
goal=[35,33]
map_file="map.dat"

Reading map...
Grid rows: 100
Grid columns: 100

path_coord = [[50, 50], [49, 50], [48, 50], [47, 50], [46, 50], [45, 50], [44, 50], [43, 50], [42, 50],
[41, 50], [40, 50], [39, 50], [39, 49], [39, 48], [39, 47], [39, 46], [39, 45], [39, 44], [39, 43], [39,
42], [39, 41], [39, 40], [39, 39], [39, 38], [39, 37], [38, 37], [37, 37], [36, 37], [35, 37], [35, 36],
[35, 35], [35, 34], [35, 33]]

path_move = ['up', 'up', 'left', 'left',
'left', 'left', 'left', 'left', 'left', 'left', 'left', 'left', 'left', 'left', 'up', 'up', 'up',
'up', 'left', 'left', 'left']

real_path(robot)= ['init', 'up', 'up',
'up', 'up', 'up', 'up', 'up', 'up', 'up', 'up', 'up', 'up', 'right', 'up', 'up', 'up', 'up',
'up', 'left', 'up', 'up', 'up', 'stop']

```

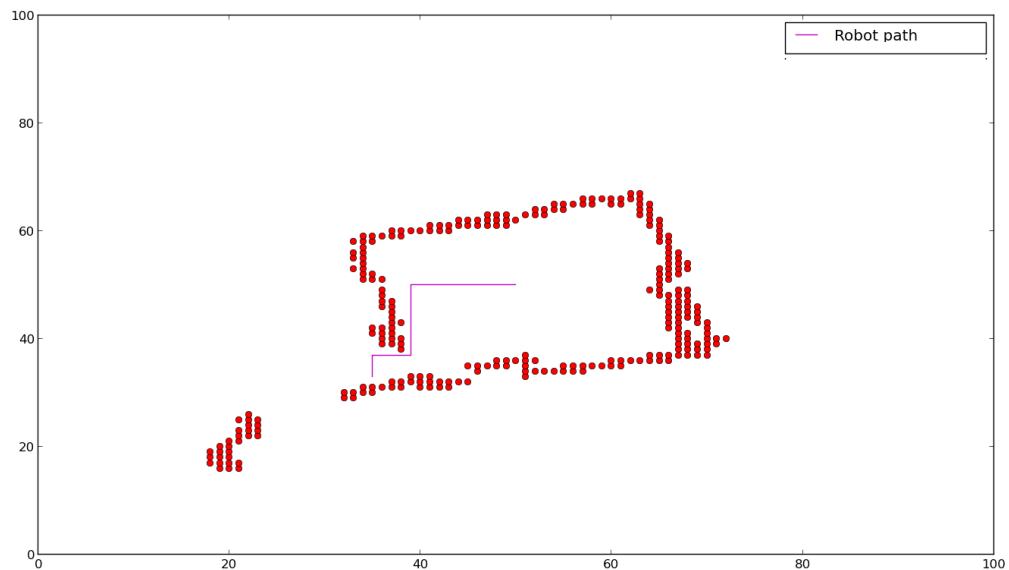


Ilustración 159: Trayectoria generada

El planificador obtiene como salida el camino óptimo en 4 formatos diferentes: las coordenadas del camino en el mapa, los comandos para recorrer el mapa, los comandos reales que se deben enviar al robot, y una construcción grafica del mapa en 2D donde se grafica la trayectoria que debe seguir el robot para alcanzar su objetivo.

Análisis de resultados: Según se observa en la salida, el robot deberá ejecutar 35 comandos para alcanzar el objetivo (la puerta de la habitación). Como cada elemento del mapa significa 10cm de la realidad, el robot deberá recorrer 3,5metros hasta la puerta.

11.6.4.2 Test en box de una habitación

El siguiente test se realizó dentro del Laboratorio de Arquitectura de Computadoras de la facultad el cual se encuentra dividido en 4 box de trabajo. Se obtuvo un mapa del primer box y se calcularon los caminos hacia los 3 objetivos generados en el paso anterior: el siguiente box que se encuentra al norte, la puerta de entrada que se encuentra al sur, y el final de un pasillo que se dirige hacia los box opuestos, que se encuentra al oeste del punto inicial.

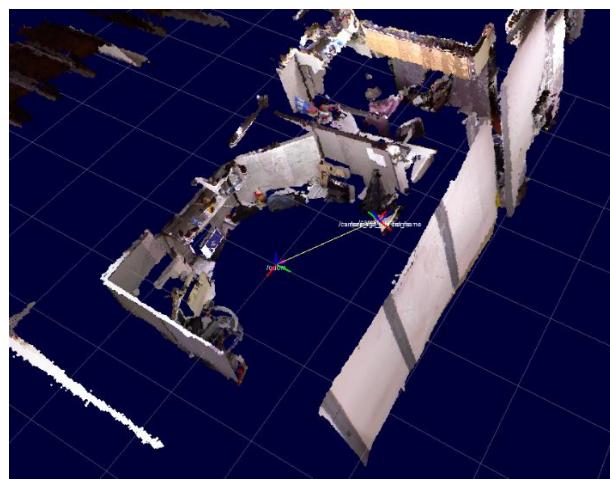


Ilustración 160: Mapeo 3D del box de trabajo en el LAC

Los objetivos mencionados se muestran en el mapa con círculos rojos, mientras que el punto inicial se marcó en azul.

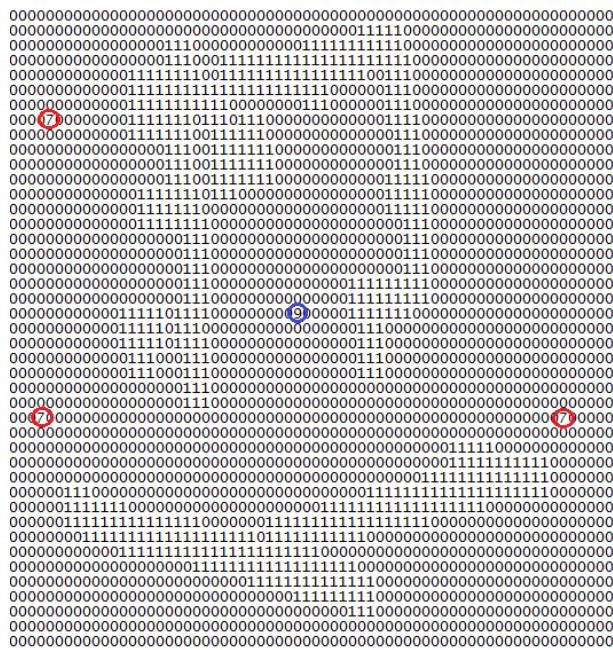


Ilustración 161: Objetivos generados en el LAC

11.6.4.2.1

Primer objetivo: Box Norte

El primer objetivo consiste en conducir el robot al siguiente box, que se ubica al norte del actual (desde donde se creó el mapa). Los resultados son:

```
#-----
#  Test1
#
init=[50,50] #(map1.dat)
goal=[57,79] #
map_file="map1.dat"

Reading map...
Grid rows: 100
Grid columns: 100

path_coord = [[50, 50], [50, 51], [50, 52], [50, 53], [50, 54], [50, 55], [51, 55], [51, 56], [52, 56],
[53, 56], [54, 56], [55, 56], [55, 57], [55, 58], [55, 59], [55, 60], [55, 61], [55, 62], [55, 63], [55,
64], [55, 65], [55, 66], [55, 67], [55, 68], [55, 69], [55, 70], [55, 71], [55, 72], [55, 73], [55, 74],
[55, 75], [55, 76], [55, 77], [55, 78], [55, 79], [56, 79], [57, 79]]

path_move = ['right', 'right', 'right', 'right', 'right', 'down', 'right', 'down', 'down',
'down', 'down', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right',
'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right',
'right', 'right', 'right']

real_path(robot)= ['init', 'up', 'up', 'up', 'up', 'right', 'up', 'left', 'up', 'right', 'up',
'up', 'up', 'up', 'left', 'up', 'up',
'up', 'up', 'up', 'up', 'up', 'up', 'up', 'up', 'right', 'up', 'stop']
```

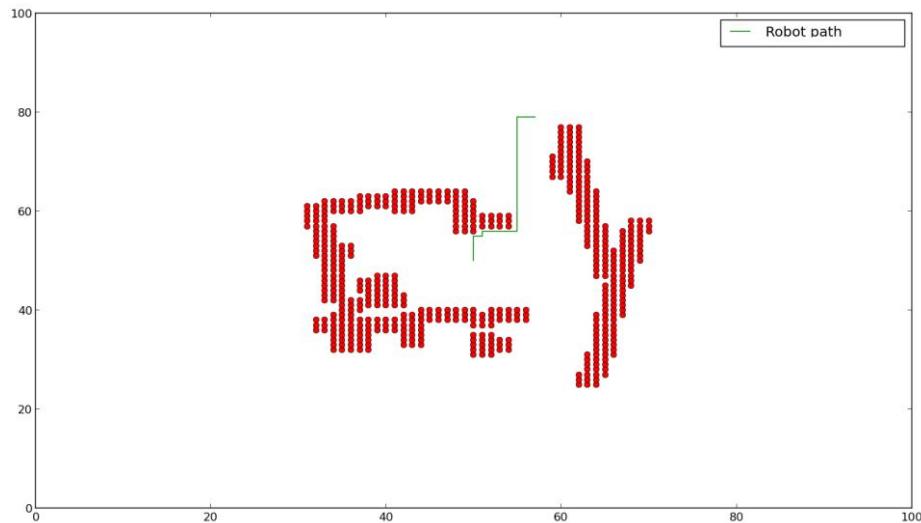


Ilustración 162: Trayectoria generada para salir del BOX hacia la primera salida

Análisis de resultados: Se puede ver en el arreglo `path_coord` que el robot alcanzará el objetivo deseado [57,79]. Además lo hará por el camino menos costoso según se muestra en la línea verde de la figura. Los comandos que se deben enviar al robot para que alcance el objetivo deseado están dados por el arreglo `real_path`.

11.6.4.2.2 Segundo objetivo: Puerta de entrada

El próximo objetivo consiste en hallar la puerta de entrada del laboratorio.

```
#-----
# Test2
#
init=[50,50] #(map1.dat)
goal=[57,22] #
map_file="map1.dat"

Reading map...
Grid rows: 100
Grid columns: 100

path_coord = [[50, 50], [50, 49], [50, 48], [50, 47], [50, 46], [50, 45], [50, 44], [50, 43], [50, 42],
[50, 41], [51, 41], [52, 41], [53, 41], [54, 41], [55, 41], [56, 41], [57, 41], [57, 40], [57, 39], [57,
38], [57, 37], [57, 36], [57, 35], [57, 34], [57, 33], [57, 32], [57, 31], [57, 30], [57, 29], [57, 28],
[57, 27], [57, 26], [57, 25], [57, 24], [57, 23], [57, 22]]

path_move = ['left', 'left', 'left', 'left', 'left', 'left', 'left', 'left', 'left', 'down',
'down', 'down', 'down', 'down', 'left', 'left', 'left', 'left', 'left', 'left', 'left', 'left',
'left', 'left', 'left', 'left', 'left', 'left', 'left', 'left', 'left', 'left', 'left']

real_path(robot)= ['init', 'up', 'up',
'up', 'up', 'up', 'up', 'right', 'up', 'up',
'up', 'up', 'up', 'up', 'up', 'up', 'stop']
```

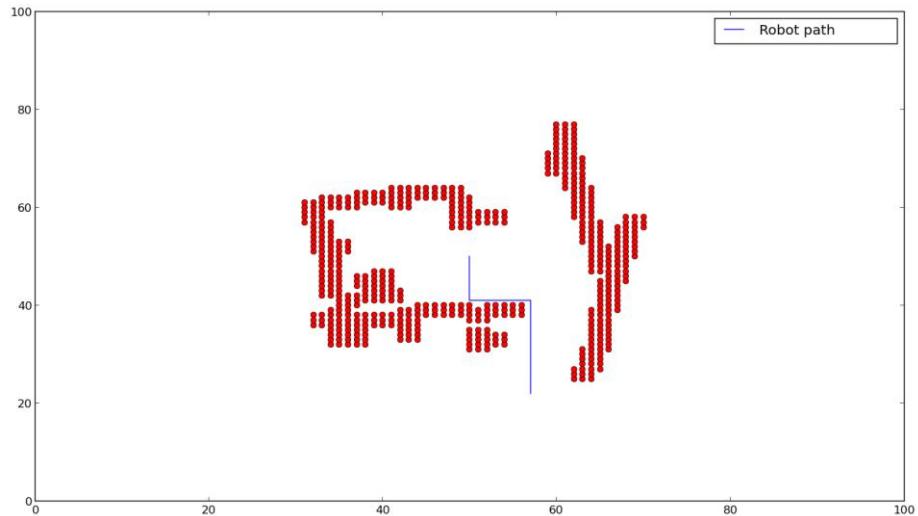


Ilustración 163: Trayectoria generada para salir del BOX hacia la segunda salida

Análisis de resultados: En la figura se ve el mejor camino que conduce hacia la puerta de entrada. El camino a recorrer son 36 pasos lo que representan 3,6 metros que deberá avanzar el robot hasta llegar a la puerta de entrada.

11.6.4.2.3 *Tercer objetivo: final del pasillo que se dirige a los demás box*

Este objetivo consiste en recorrer el pasillo que permite dirigirse a los demás boxes del laboratorio. Los resultados obtenidos son:

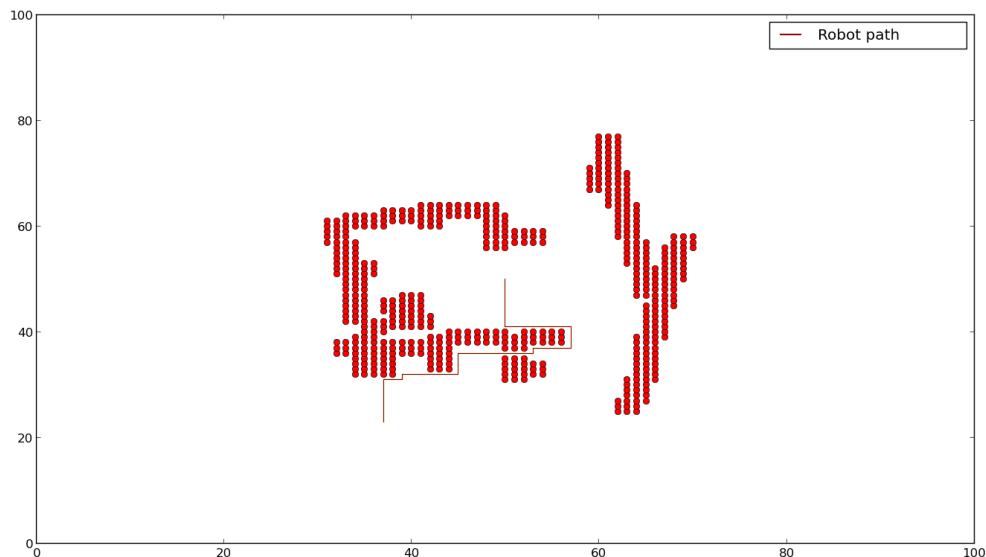


Ilustración 164: Trayectoria hasta el tercer punto

Análisis de resultados: Como se puede ver en la figura, el robot doblará para esquivar la pared, y luego pasará entre dos obstáculos (la pared y algún otro obstáculo por debajo). En la realidad ese obstáculo no existe. El mapa lo incluye debido a que desde la posición desde donde se generó no se podía ver que había detrás de la pared. Esto permite tomar conocimiento de que el mapa debe ser enviado de nuevo al planificador cada vez que tenga nuevas informaciones, y a medida que se va haciendo más preciso debido a nuevas zonas que explora el robot.

11.6.4.2.4 Otro objetivo: salir del laboratorio

Este es un test adicional que se realizó para probar la capacidad del algoritmo de evitar obstáculos. No es aplicable a la realidad, debido a que el mapa se generó desde adentro del laboratorio y se le está pidiendo al planificador que devuelva el mejor camino hacia un punto que se encuentra en el exterior del mismo (todavía no conocemos que hay afuera del laboratorio). Pero a pesar de eso, nos muestra la capacidad que tiene el planificador de calcular caminos más largos y rebuscados.

```
#-----
# Test4
#
init=[50,50] #(map1.dat)
goal=[77,37] #
map_file="map1.dat"

Reading map...
Grid rows: 100
Grid columns: 100

path_coord = [[50, 50], [50, 49], [50, 48], [50, 47], [50, 46], [50, 45], [50, 44], [50, 43], [50, 42],
[50, 41], [51, 41], [52, 41], [53, 41], [54, 41], [55, 41], [56, 41], [57, 41], [57, 40], [57, 39], [57,
38], [57, 37], [57, 36], [57, 35], [57, 34], [57, 33], [57, 32], [57, 31], [57, 30], [57, 29], [57, 28],
[57, 27], [57, 26], [57, 25], [57, 24], [58, 24], [59, 24], [60, 24], [61, 24], [62, 24], [63, 24], [64,
24], [65, 24], [65, 25], [65, 26], [66, 26], [66, 27], [66, 28], [66, 29], [66, 30], [67, 30], [67, 31],
[67, 32], [67, 33], [67, 34], [67, 35], [67, 36], [67, 37], [68, 37], [69, 37], [70, 37], [71, 37], [72,
37], [73, 37], [74, 37], [75, 37], [76, 37], [77, 37]]]

path_move = ['left', 'left', 'left', 'left', 'left', 'left', 'left', 'left', 'left', 'down',
'down', 'down', 'down', 'down', 'down', 'left', 'left', 'left', 'left', 'left', 'left', 'left',
'left', 'left', 'left', 'left', 'left', 'left', 'left', 'left', 'left', 'down', 'down',
'down', 'down', 'down', 'down', 'right', 'right', 'right', 'down', 'right', 'right', 'right',
'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'down', 'down',
'down', 'down', 'down', 'down']
```

```
real_path(robot)= ['init', 'up', 'up', 'up', 'up', 'up', 'up', 'up', 'up', 'up', 'left', 'up', 'up',  
'up', 'up', 'up', 'up', 'right', 'up',  
'up', 'up', 'up', 'up', 'up', 'left', 'up',  
'up', 'right', 'up', 'left', 'up', 'up', 'up', 'up', 'right', 'up', 'left', 'up', 'up', 'up', 'up', 'up',  
'up', 'up', 'right', 'up', 'stop']
```

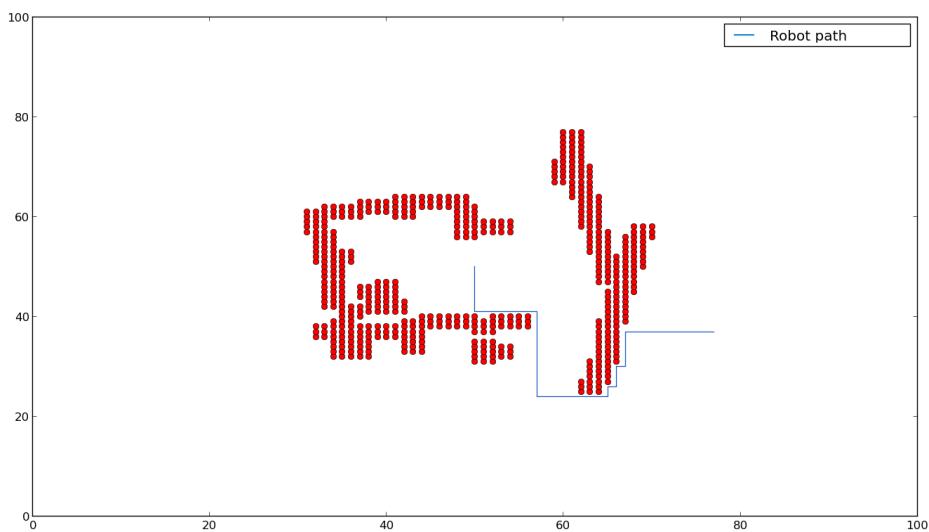


Ilustración 165: Trayectoria hasta objetivo artificial

Análisis de resultados: Como se ve en la figura y en los arreglos, se alcanza el objetivo deseado con el camino más corto posible. A pesar de que no se conozca que hay más allá de la puerta de entrada.

11.6.4.3 Test sobre el robot

Se hicieron numerosos test sobre el robot para diferentes caminos. A continuación se hace un análisis detallado de dos casos, uno muy simple y otro un poco más complejo.

11.6.4.3.1 Camino simple

Para la primera prueba se generó un camino muy simple con dos tramos y un giro para observar la respuesta del robot:

```
robot_path(transformed)=  
['init', 'up', 'left', 'up', 'up', 'up', 'up', 'up', 'stop']
```

La información de localización recibida desde Arduino se muestra a continuación:

```
Pop from buffer: init
Pop from buffer: up
Robot: [ heading: 359.79 x: 9.79 y: -0.01 ]
Robot: [ heading: 359.26 x: 19.57 y: -0.09 ]
Robot: [ heading: 358.64 x: 29.32 y: -0.27 ]
Robot: [ heading: 358.11 x: 39.09 y: -0.56 ]
Robot: [ heading: 357.37 x: 48.84 y: -0.95 ]
Robot: [ heading: 356.61 x: 58.57 y: -1.46 ]
Robot: [ heading: 355.85 x: 68.29 y: -2.10 ]
Robot: [ heading: 355.18 x: 78.00 y: -2.86 ]
Robot: [ heading: 354.43 x: 87.70 y: -3.75 ]
Pop from buffer: left
Robot: [ heading: 84.01 x: 87.75 y: -3.75 ]
Pop from buffer: up
Robot: [ heading: 84.05 x: 88.72 y: 5.92 ]
Robot: [ heading: 83.52 x: 89.77 y: 15.59 ]
Robot: [ heading: 83.08 x: 90.91 y: 25.26 ]
Robot: [ heading: 82.56 x: 92.12 y: 34.91 ]
```

```

Robot: [ heading: 82.11 x: 93.42 y: 44.54 ]
Robot: [ heading: 81.67 x: 94.80 y: 54.19 ]
Robot: [ heading: 81.26 x: 96.24 y: 63.79 ]
Robot: [ heading: 81.14 x: 97.73 y: 73.41 ]
Robot: [ heading: 81.09 x: 99.24 y: 83.01 ]
Robot: [ heading: 80.93 x: 100.75 y: 92.59 ]
Robot: [ heading: 80.83 x: 102.29 y: 102.18 ]
Robot: [ heading: 80.79 x: 103.85 y: 111.77 ]
Robot: [ heading: 80.79 x: 105.41 y: 121.38 ]
Robot: [ heading: 80.73 x: 106.98 y: 130.97 ]
Robot: [ heading: 89.20 x: 108.42 y: 140.60 ]
Pop from buffer: stop

```

Análisis de resultados: La información de localización muestra que se ejecutaron 9 comandos up que representan un avance de 10cm cada uno, es decir, se recorrieron 90cm en X, luego se hizo un giro a la izquierda, y finalmente se avanza en Y 1,5 metros más. Se analizan los errores por tramo:

- *Tramo 1 (avance de 90cm en X): se obtiene 2,5% de error ya que se llega a 84,7cm en lugar de 90cm. Luego en el giro de 90 grados en el sentido hacia la izquierda, el giro es de 89,57 grados por lo que el error mínimo, 0,47%.*
- *Tramo 2 (avance de 150cm en Y): se alcanzan los 140,6cm partiendo desde los -3,7cm lo que significa un error del 3,76%.*

En el movimiento lineal el error acumulado es de 6,26% en 2,4 metros recorridos, mientras que en el movimiento de giro es solo del 0,47% en 90 grados. El error representa el porcentaje de desvió respecto del objetivo deseado.

11.6.4.3.2 Camino con varios giros

Para el siguiente análisis se generó un nuevo camino en el que hay un tramo de avance en X de 70cm, un giro de 90 grados a la izquierda, avance de 10cm más, giro de 90 grados a la derecha, avance de 10cm más, giro de 90 a la izquierda, avance de 60cm en Y:

```

robot_path(transformed)=
['init', 'up', 'up', 'up', 'up', 'up', 'up', 'left', 'up', 'right', 'up', 'left', 'up', 'up', 'up',
'up', 'up', 'up', 'up', 'up', 'up', 'up', 'up', 'up', 'up', 'up', 'up', 'up', 'up', 'up', 'up',
'up', 'up', 'up', 'up', 'up', 'up']
Pop from buffer: init
Pop from buffer: up
Robot: [ heading: 359.50 x: 9.78 y: -0.03 ]
Robot: [ heading: 358.82 x: 19.54 y: -0.17 ]
Robot: [ heading: 358.04 x: 29.35 y: -0.44 ]
Robot: [ heading: 357.25 x: 39.16 y: -0.85 ]
Robot: [ heading: 356.50 x: 48.94 y: -1.38 ]
Robot: [ heading: 355.76 x: 58.65 y: -2.04 ]
Robot: [ heading: 355.02 x: 68.40 y: -2.83 ]
Pop from buffer: left
Robot: [ heading: 85.83 x: 68.46 y: -2.83 ]
Pop from buffer: up
Robot: [ heading: 86.86 x: 68.95 y: 7.02 ]
Pop from buffer: right
Robot: [ heading: 357.48 x: 68.95 y: 7.07 ]
Pop from buffer: up
Robot: [ heading: 356.53 x: 78.77 y: 6.44 ]
Pop from buffer: left
Robot: [ heading: 86.45 x: 78.82 y: 6.43 ]
Pop from buffer: up
Robot: [ heading: 85.13 x: 79.31 y: 16.17 ]
Robot: [ heading: 84.07 x: 80.70 y: 26.34 ]
Robot: [ heading: 83.56 x: 81.82 y: 36.27 ]
Robot: [ heading: 82.41 x: 83.14 y: 46.18 ]
Robot: [ heading: 81.61 x: 84.60 y: 56.08 ]
Robot: [ heading: 80.27 x: 86.29 y: 65.93 ]
Pop from buffer: stop

```

Análisis de resultados: La información de localización que provee Arduino se analiza por tramos.

- Tramo 1 (avance de 70cm en X): se obtiene 2,2% de error ya que se llega a 68,4cm en lugar de 70cm.
- Tramo 2 (giro a la izquierda y avance de 10cm): en el movimiento de giro se mide un giro de 90,81 grados, lo que implica un error del 0,9%. En el movimiento en Y se tiene un error del 0,15% ya que se avanza 9,85cm en lugar de 10cm.
- Tramo 3 (giro a la derecha y avance de 10cm): en el giro se mide un ángulo de 89,38 grados lo que significa un error del 0,68%. Luego en el avance en X el error es del 0,18%.
- Tramo 4 (giro a la izquierda y avance 60cm): en el giro el error es del 0,08% mientras que en el avance en Y, el robot avanzó 59,5cm en lugar de 60cm por lo que el error es de 0,83%.

Finalmente el error acumulado para el movimiento lineal es de 3,36% para un total de 1,5metros. Mientras que el error acumulado de giro es del 2,41% para un total de 270 grados recorridos.

11.6.5 Integración con el sistema

El planificador, implementado como nodo “planner”, debe recibir la siguiente información desde información desde el nodo de mapeo: el mapa en 2 dimensiones, el punto inicial o posición actual del robot, y los objetivos a alcanzar.

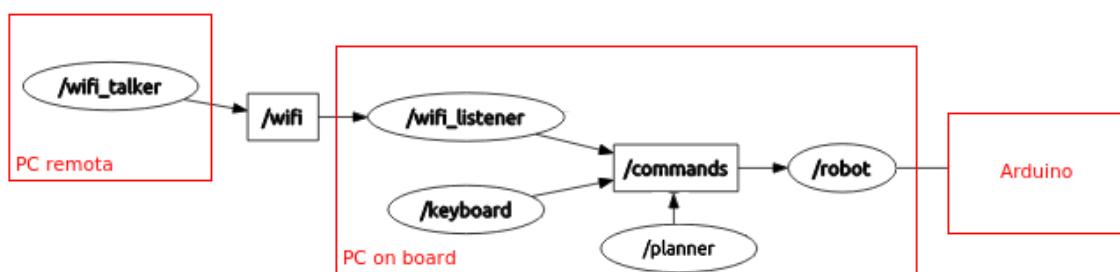


Ilustración 166: Nodos del sistema para los modos Debug y telepresencia

El Planner fue implementado en Python e internamente contiene el algoritmo de búsqueda de caminos óptimos. Una vez que encuentra la lista de comandos se los envía al nodo Robot mediante el paso de mensajes que se logra a través del topic “commands” en el que el Planner publica y Robot escucha. El nodo Robot se comunica a través del puerto serial con Arduino, por lo que una vez que le llegan los comandos, estos son enviados a hacia Arduino. En el controlador se implementó una cola para ir guardando los comandos que le llegan e ir ejecutándolos uno a la vez.

Robot móvil autónomo capaz de producir mapas 3D del interior de una habitación

11.7 Sistema completo

Tal como se describió en el diseño de la arquitectura (sección 0), el sistema está compuesto por módulos de hardware y paquetes de software.

A bajo nivel se crearon paquetes para el manejo del robot sobre el microcontrolador Arduino el cual tiene conexión con el hardware. El primero primero se denomina “Gyroscope” y todo su contenido está basado en el código para el MPU6050 desarrollado y mantenido por Jeff Rowberg (ver 11.2.3.2).

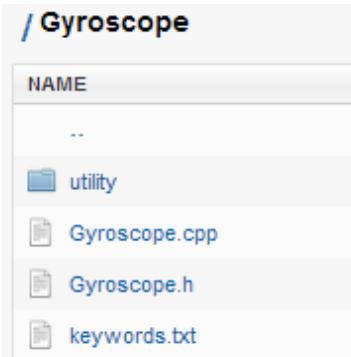


Ilustración 167: Librería “Gyroscope” de Arduino

El segundo paquete reúne varias librerías, y se denomina “Robot”. Cuenta con todo el código desarrollado para: controlar los motores en una librería “motores.cpp”, hacer lecturas de los encoders en “encoders.cpp”, la librería “control.cpp” tiene el controlador PID del robot, hay otra “localizacion.cpp” para el posicionamiento del robot, “commands.cpp” que es la interfaz de comunicación con Arduino donde se parsean todos los comandos, y finalmente se tiene el archivo “pin_out.h” donde se mapean los pines de Arduino con el hardware asociado como motores, encoders, giróscopo, etc.



Ilustración 168: Librería “Robot” de Arduino

Luego, desde el Sketchbook de Arduino (lugar de trabajo donde se guardan los archivos .ino) se debe tener un “main.ino” que incluya todas las librerías antes mencionadas. El sistema se pondrá en funcionamiento y se establecerá la conexión con ROS mediante inicializaciones en

setup(). Finalmente, se llama al método run() con los parámetros velocidad, ganancias del controlador PID, y habilitando o no la impresión por consola de información de los motores, e información de localización.

```

128. #include <control.h>
129. #include <motor.h>
130. #include <pin_out.h>
131. #include <encoder.h>
132. #include <commands.h>
133. #include <localization.h>
134. #include <Gyroscope.h>
135.
136. void setup()
137. {
138.   Serial.begin(115600);
139.   motorInit();
140.   encoderInit();
141.   gyroInit();
142.   positionInit();
143. }
144.
145. void loop()
146. {
147.   run(10, 2.5, 3.5, 0.0952, false, true);
148. }
```

En un nivel más arriba, sobre la computadora a bordo, se encuentra el sistema operativo ROS. En el directorio /home de la computadora a bordo, se creó un workspace donde se deben incluir todos los stacks y paquetes de ROS necesarios para el funcionamiento del robot. Se construyó un package robot el cual contiene los siguientes nodos, separados en dos carpetas, una para el cliente y otra para el servidor:

1. src/client
 - wifi_talker
2. src/server
 - wifi_listener
 - keyboard
 - robot

A continuación se muestra el contenido del paquete:



Ilustración 169: Directorio del workspace de ROS donde se encuentra el Paquete ‘robot’ y su contenido

Se hizo una subdivisión en dos carpetas, una que contiene los nodos de la computadora a bordo, se la llamó “server”, y otra para el manejo del robot remotamente, que se denomina “client”. Dentro del directorio /src se incluyen todos los nodos desarrollados en C++.

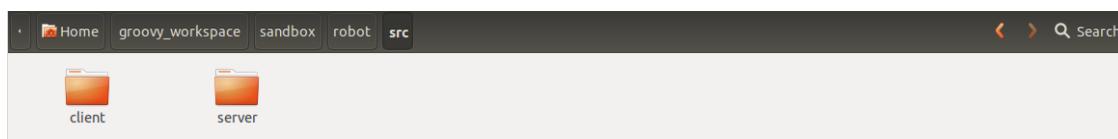


Ilustración 170: Carpetas con nodos en C++ del servidor y del cliente

Dentro del servidor se tienen los nodos “keyboard.cpp”, “robot.cpp” y “wifi_listener.cpp” cuyo funcionamiento ya ha sido descripto en (9.2).

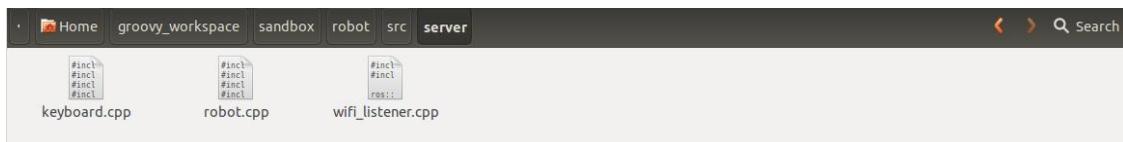


Ilustración 171: Nodos de ROS del servidor en C++

Dentro del cliente se tiene el nodo “wifi_talker.cpp” que envía comandos al servidor en modo de telepresencia (ver 9.2).

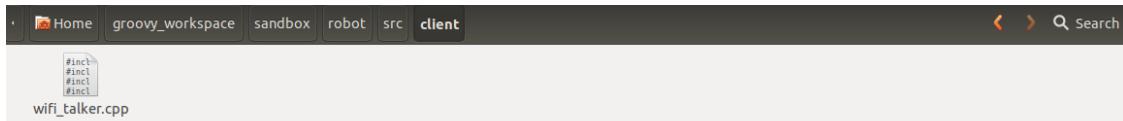


Ilustración 172: Nodos de ROS del cliente en C++

Por otro lado, en el directorio /scripts se incluyen los nodos en Python. En este se tiene el “planner.py” y los mapas que provee el módulo de visualización y mapeo sobre los cuales se calcula el camino óptimo.



Ilustración 173: Nodo de ROS en Python

Modo de operación

En **modo debugging** (ver 9.2), hay un nodo de ROS llamado keyboard que permite ingresar comandos directamente desde el teclado del servidor. Para el modo de ingreso, se puede optar por escribir el nombre de cada comando, o simplemente mover el robot con las flechas del teclado, donde cada se traduce a un movimiento del robot.

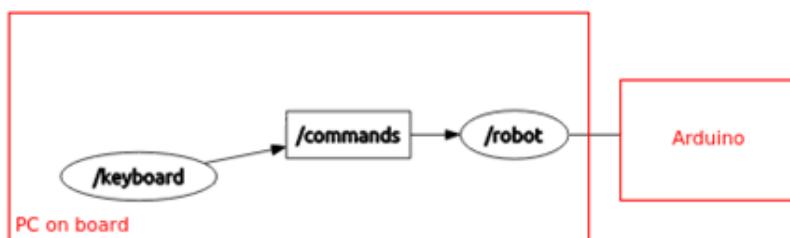


Ilustración 174: Nodos de ROS en modo debugger.

Por otro lado, en el **modo de telepresencia** desde la maquina cliente, se pueden enviar comandos mediante una conexión TCP al servidor (el cual se encontrará en el robot mismo). La entrada de los comandos es por teclado. El nodo wifi_talker (que pertenece al cliente) envía comandos “up”, “down”, “right”, “left”, o “stop” al nodo wifi_listener (que pertenece al servidor) que los retransmite al nodo robot. Este último tiene una conexión USB-Serial establecida con Arduino a partir de la cual envía todos los comandos al robot. La siguiente imagen muestra los nodos involucrados en el modo de operación descripto:

Robot móvil autónomo capaz de producir mapas 3D del interior de una habitación

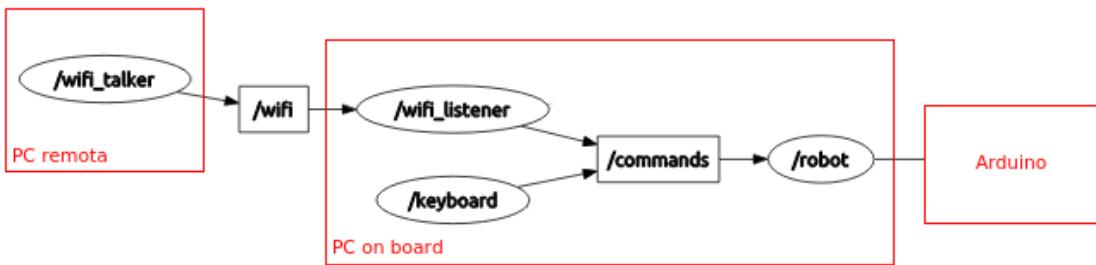


Ilustración 175: Nodos de ROS en modo telepresencia.

Finalmente, en el **modo de mapeo y exploración**, el robot planea sus movimientos autónomamente en base a un mapa, según se describió en 11.5. Como se puede ver, en este caso se agrega el nodo planner de ROS que es el encargado de enviar los comandos al robot. Cabe destacar que, por falta de tiempo, en este trabajo no se realizó la conexión entre el nodo de mapeo y el de navegación (ver 13). Sin embargo, se crearon las interfaces para la conexión de dichos nodos.

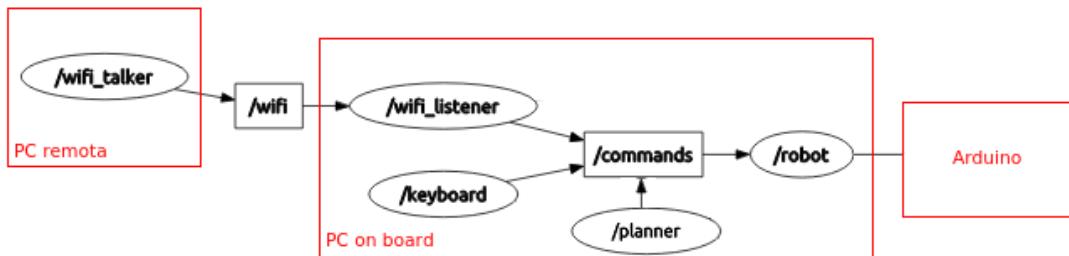


Ilustración 176: Nodos de ROS en modo de mapeo y exploración.

12 Conclusión

El presente proyecto involucró diferentes etapas tales como: prueba de motores y sensores, su comunicación con el microcontrolador Arduino, desarrollo de paquetes sobre ROS, comunicación entre la computadora (remota o local) y el robot para el envío de comandos, construcción del robot, control del robot, odometría visual y mapeo y, exploración y planificación de trayectorias. A medida que se analizaba y aprendía sobre nuevos temas y herramientas disponibles, las expectativas y requerimientos cambiaban debido al alto grado de innovación del proyecto. Un ejemplo de esto se puede ver en la sección 11.5.5.

Al utilizar un diseño modular, se tuvo la ventaja de que cada etapa pudo ser desarrollada por separado y en paralelo, y luego se integraron y probaron las partes en cada incremento. Resultó muy adecuada la elección de un desarrollo basado en componentes; lo que permitió definir responsabilidades e interfaces al inicio del proyecto, dejando en claro qué partes era necesario desarrollar, para luego ir integrándolas a medida que se necesitaba hacerlo.

Durante la etapa de construcción del robot, se plantearon varias posibilidades de diseño. Inicialmente, se construyó un prototipo rápido para iniciar las pruebas. Luego, basándose en el diseño original, se armó un nuevo prototipo de mayor tamaño para soportar una más variada cantidad de computadoras y mejor estética. Se utilizaron nuevos mecanismos para el ajuste de los motores, solucionando el problema del deslizamiento de estos. Por último, se optó por dos ruedas de tracción y un direccionamiento basado en el modelo diferencial.

En cuanto a los motores y sus encoders, las conexiones del robot se mostraron frágiles en distintas ocasiones durante el desarrollo del proyecto y fue necesario repararlas más de una vez. Esto causó funcionamientos indebidos del sistema, que en algunos casos no fue posible solucionar de forma inmediata. De todos modos, utilizar y verificar el correcto funcionamiento de motores y encoders no presentó ningún tipo de obstáculos para la ejecución del proyecto.

El control del robot fue una de las etapas con mayores dificultades. Inicialmente, se trabajó en la alineación de las ruedas traseras. Luego, se buscó conseguir un movimiento en línea recta hasta una distancia dada ajustando la velocidad de los motores. La estructura y tipo de rueda delantera utilizada, causó muchos problemas, ya que se generaba un desvío en el desplazamiento y un patinado en la rotación, que no eran producto de un mal funcionamiento del controlador, sino puramente mecánicos. Lamentablemente, no se pudieron conseguir ruedas esféricas en el país por lo que se las tuvo que comprar en el exterior. Al día de la fecha, todavía no contamos con ella por lo que se la propone como Trabajos futuros y mejoras (ver 13). Ésta debe ser colocada sustituyendo la rueda delantera.

Afortunadamente, con la utilización de todos los demás sensores (magnetómetro, giróscopo, acelerómetro, Kinect) no hubo inconvenientes. Hay disponibilidad de estos componentes en el mercado local, y se los puede adquirir rápidamente. Las comunidades de ROS y Arduino disponen de librerías y drivers con buenos tutoriales que permiten utilizar los sensores muy rápidamente. En cuanto al giróscopo, se observó que hay un período inicial durante el cual éste debe estabilizarse. La única forma de impedir que esto resulte en un mal funcionamiento, fue introduciendo una demora inicial en el software hasta que el giróscopo se encuentre listo.

Sobre el mapeo y la odometría visual, inicialmente solo se buscaba generar mapas, sin importar si eran en 2D o 3D, con o sin color. Se pensaba en utilizar directamente los valores de la cámara de profundidad para detectar paredes y obstáculos y usar esto para complementar la localización. Pero, luego de investigar, se decidió aplicar técnicas de odometría visual en lugar de usar los valores directos desde el sensor. Parte de esta etapa consistió en la lectura de material científico y documentación de diferentes proyectos, probándose implementaciones de distintas universidades y grupos de investigación. Finalmente, se concluyó esta etapa con dos modos para mapeo en 3D color, uno más rápido que otro. Ver sección 11.5.2.3.

La última etapa, de generación de trayectorias y navegación, depende de la etapa de control del robot. Logrado el correcto funcionamiento del control, el seguimiento de una trayectoria se basa en dividir el trayecto en pequeños comandos y enviarlos al robot. Dado que el requerimiento inicial indica que el robot no solo debe seguir una trayectoria dada y decidir a donde ir por sí solo (ser autónomo), se requirió la utilización de algoritmos de generación de objetivos y luego de búsqueda para encontrar el mejor camino hasta dichos puntos, definidos sobre el mapa. Debido a la modularidad del sistema, fue muy simple lograr utilizar el mapa (salida) como entrada del planeador de trayectorias y ésta, a su vez, como entrada al controlador del robot.

Tal como se mencionó en la etapa de análisis de riesgos (ver página 49), el no disponer de motores, baterías y otros insumos era un riesgo de alta probabilidad de ocurrencia. Esto lamentablemente no se pudo solucionar ya que escapa a las posibilidades del equipo, pero de todos modos no impactó en la ejecución del proyecto ya que fue posible reparar todos los componentes que mostraron un mal funcionamiento.

La ejecución del proyecto involucró componentes de hardware, software, comunicaciones, sensores, motores, etc.; lo cual requirió el dominio de las diferentes áreas de la carrera y, a su vez, una oportunidad para profundizar en la teoría y práctica de las mismas. Esto significó una experiencia altamente gratificante para todo el equipo de desarrollo, los que consideran cumplidos sus objetivos técnicos y personales.

Fue de vital importancia para el éxito del proyecto el entorno de trabajo brindado por el Laboratorio de Arquitectura de Computadoras y el Grupo de Robótica y Sistemas Integrados, que permitieron, cada vez que fue necesario, consultar a expertos de distintas áreas.

Este trabajo representa el primer prototipo de la plataforma robótica, el cual se puede mejorar en muchos aspectos y abre numerosos caminos para futuros estudios y desarrollos.

Se deja expresa la disposición del equipo de trabajo, para brindar apoyo, y todo el conocimiento adquirido a lo largo de estos 6 meses de desarrollo, a todo lo que surja en relación al proyecto, de aquí en adelante.

13 Trabajos futuros y mejoras

A continuación se plantean algunos trabajos a realizar a futuro, sobre trabajos pendientes y mejoras que no fue posible concretar durante el desarrollo del proyecto.

1. **Rueda delantera del robot:** Dado que muchas veces los movimientos del robot se vieron influenciados por la rueda delantera utilizada en el actual prototipo, se propone cambiarla por una rueda esférica. Dicha rueda ya fue encargada en el extranjero, por lo que probablemente en el corto plazo se disponga de ella. Se plantea como trabajo futuro colocarla, y realizar nuevos test para ver cómo reacciona el robot ante el cambio.
2. **Medición del nivel de batería y corriente de los motores:** En muchas ocasiones durante el proceso de testing que involucraba al robot, la batería se agotó y el robot comenzó a funcionar mal. Se debía controlar el nivel de carga de la batería en cada momento para que su nivel de tensión no sea inferior a la mínima permitida dado que esto puede dañar algunas celdas. Para evitar este problema, lo óptimo sería poder medir el nivel de tensión de la batería con Arduino, y enviar alarmas de bajo nivel de batería a la computadora a bordo, para que el usuario se entere de esto y proceda a cargarlas. Otra aplicación de esta mejora es el controlador PID. Para establecer la máxima velocidad a la que el robot puede desplazarse, el controlador debe conocer el nivel de batería actual del robot. De esta forma, la velocidad máxima podría establecerse de acuerdo al nivel de carga que tenga la batería.
Por otro lado sería bueno poder monitorear la corriente que los motores consumen por software, con el fin de alertar malos funcionamientos de los mismos.
3. **Automatización de test de control:** Para hacer test del controlador, se confeccionó una planilla de cálculo. Cada vez que se realiza un test es necesario cargar los datos obtenidos por consola en la planilla. Para evitar este trabajo, se podría hacer un script (por ejemplo en el lenguaje Perl) que permita importar los datos del robot obtenidos en la consola, directamente en la planilla de test que hace los gráficos automáticamente.
4. **Algoritmo Twiddle:** Si bien ya se tiene una implementación del algoritmo que se presenta en la sección 11.3.5.8, los resultados medidos sobre el robot no fueron del todo satisfactorio. Por lo tanto, se propone como trabajo futuro, seguir investigando dicho algoritmo para lograr tener una optimización del cálculo de parámetros correcta.
5. **Mejoras en la localización:** Dado que el paquete de ROS utilizado para mapeo permite obtener datos de localización desde la cámara, se plantea como mejora combinar datos de localización que provee la odometría visual, con la localización que cuenta el robot actualmente, hecha a partir de encoders rotacionales y giróscopos.

6. **Integración completa para autonomía en exploración:** En el presente, se crearon los módulos y las interfaces necesarias para la conexión entre el módulo de mapeo y de navegación o planeamiento de trayectorias. Se realizaron los test correspondientes sobre cada una por separado, pero, por falta de tiempo, no se llegó a conectarlos. Se plantea como trabajo futuro integrar ambos módulos y probarlos de manera conjunta, de forma tal que los mapas que envía el módulo de mapeo lleguen al navegador a partir de su interfaz.

14 Anexo

14.1 Anexo A: Propuesta del proyecto y estado del arte

Al momento de presentar el pre-proyecto de tesis, se creó una presentación en Prezi en la cual se muestran muchos proyectos similares que están siendo desarrollados tanto en otras Universidades del mundo, como en empresas. Se analizan las posibles tecnologías relacionadas al proyecto, empresas que fabrican robots similares, etc. La presentación se puede encontrar en:

http://prezi.com/13osqlrg7vve/present/?auth_key=fmeuwfe&follow=kqhvoyg7s53r&kw=present-13osqlrg7vve&rc=ref-5906028

14.2 Anexo B: Principios de funcionamiento del Magnetómetro

El magnetómetro es uno de los sensores con los que se intentó medir la orientación del robot. Para entender cómo un magnetómetro puede ser utilizado con tales fines, se presenta una breve introducción al tema.

Como el nombre lo indica, este dispositivo es utilizado para realizar mediciones de intensidad de campos magnéticos. También existen magnetómetros vectoriales, con los cuales se mide la intensidad del campo en la dirección de los tres ejes perpendiculares entre sí (x, y, z) para poder estimar la dirección del campo.

Como es sabido, alrededor de la Tierra se puede medir un campo magnético cuyos orígenes se atribuyen a la rotación de hierro fundido en el núcleo terrestre. Este campo es denominado campo magnético terrestre o campo geomagnético y los valores que se miden a lo largo y ancho de la superficie terrestre oscilan entre 25 y 65 μ T (0.25 – 0.65 G) (Wikipedia, Earth's magnetic field). Es un campo muy débil a comparación de los producidos por algunos de los electrodomésticos presentes en una casa promedio. Por ejemplo, una heladera genera un campo de intensidad de 100Gauss aproximadamente (Eric Palm, Deputy Lab Director). A continuación se muestra los valores de intensidad sobre la superficie terrestre. Todos los valores indicados se dan en la unidad nano Tesla (nT). La imagen muestra que la intensidad es mayor cerca de los polos y menor cerca del ecuador.

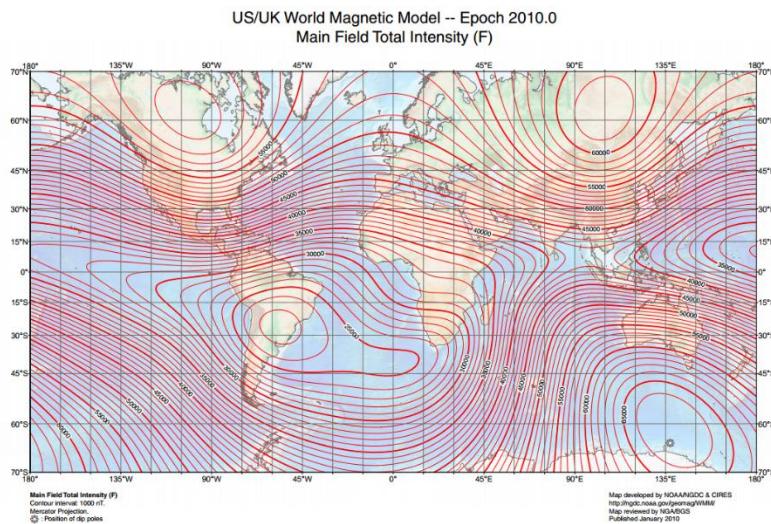


Ilustración 177: Campo geomagnético alrededor de la Tierra (Wikimedia)

A través de una simulación por computadora, investigadores obtuvieron una representación tridimensional del campo geomagnético (Wikipedia, Earth's magnetic field).

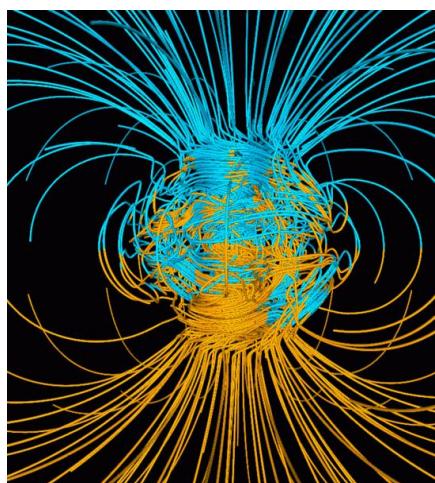


Ilustración 178: Representación 3D del campo geomagnético

De la imagen anterior se deduce que en cada punto de la superficie terrestre se puede representar el campo mediante un vector tridimensional con una dirección, sentido y amplitud. Además, se puede apreciar la similitud de las líneas de fuerza del campo terrestre con las líneas de campo producidas por una barra magnética. Por este motivo, es común pensar en un gigantesco imán cuyo polo sur se encuentra en el norte geográfico. De esta forma el polo norte de las brújulas apunta al polo norte geográfico, ya que es atraído hacia el polo sur del imán terrestre (norte geográfico).

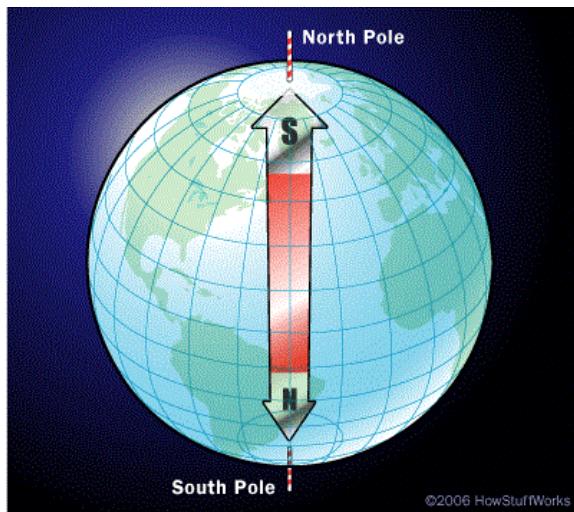


Ilustración 179: Modelo simplístico del campo geomagnético

Para ser más precisos, el imán terrestre no está completamente alineado con el norte geográfico o norte verdadero. De esta diferencia surge la denominada declinación magnética. Esto es, el ángulo entre la dirección indicada por una brújula y la dirección hacia el norte terrestre verdadero.

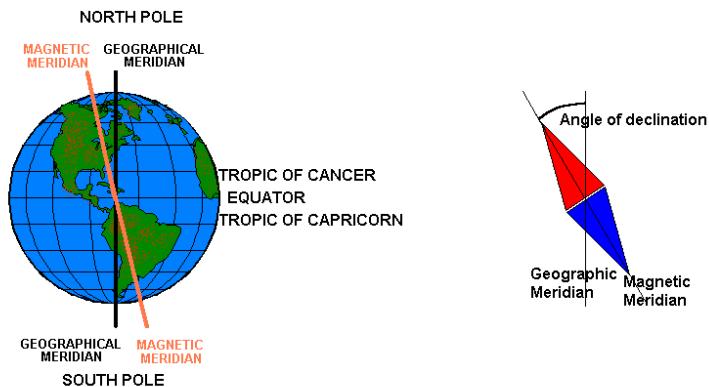


Ilustración 180: Declinación magnética (Norte geográfico y norte magnético. Declinación)

El siguiente mapa indica como varía la declinación en la superficie terrestre. La línea verde que va desde el sur hacia el norte sobre las Américas, marca el 0. Al Este de la línea, los valores son negativos y al Oeste positivos.

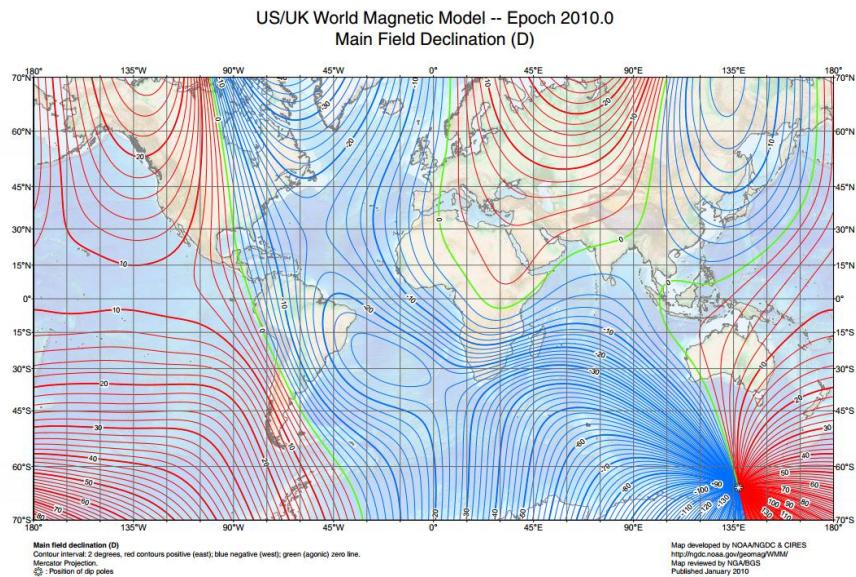


Ilustración 181: Declinación magnética sobre la superficie terrestre (Wikimedia)

15 Bibliografía

*Cada una de las siguientes URLs se encuentran en funcionamiento el día 24 de Mayo del 2013.

Arduino Mega2560. Obtenido de

<http://arduino.cc/en/Main/arduinoBoardMega2560>

BBC. Obtenido de

<http://www.bbc.co.uk/news/technology-11742236>

BBC Mundo. Obtenido de

http://www.bbc.co.uk/mundo/noticias/2013/02/130222_robots_avanzan_sobre_economia_mundial.mj.shtml

ccny_rgbd documentation. Obtenido de

http://ros.org/doc/fuerte/api/ccny_rgbd/html/

Datasheet HMC5883L. Obtenido de

http://www51.honeywell.com/aero/common/documents/myaerospacecatalog-documents/Defense_Brochures-documents/HMC5883L_3-Axis_Digital_Compass_IC.pdf

Datasheet InvenseSense. Obtenido de

<http://www.invensense.com/mems/gyro/documents/PS-MPU-6000A.pdf>

Dryanovski, I. ccny_rgbd. Obtenido de

http://www.ros.org/wiki/ccny_rgbd

Eric Palm, Deputy Lab Director. Magnet Lab. Obtenido de

<http://www.magnet.fsu.edu/education/tutorials/magnetminute/tesla-transcript.html>

Galin, D. (2004). *Software Quality Assurance, From theory to implementation*. PEARSON Addison Wesley.

How to connect Kinect to 12V battery. Obtenido de

<http://answers.ros.org/question/9647/how-to-connect-kinect-to-a-12v-battery/>

Hugh Durrant-Whyte, F. I. Simultaneous Localisation and Mapping (SLAM): Part I The Essential Algorithms. UC Berkeley CS. Obtenido de

http://www.cs.berkeley.edu/~pabbeel/cs287-fa09/readings/Durrant-Whyte_Bailey_SLAM-tutorial-I.pdf

IFR International Federation of Robotics. Obtenido de

<http://www.ifr.org/industrial-robots/statistics/>

Ivan Dryanovski, R. G. (2013). Fast Visual Odometry and Mapping from RGB-D Data. *International Conference on Robotics and Automation (ICRA2013)*.

Robot móvil autónomo capaz de producir mapas 3D del interior de una habitación

Magnetic declination, Sentido de declinación magnética. (s.f.). Obtenido de

<http://magnetic-declination.com/what-is-magnetic-declination.php>

Mathieu Labbe, F. M. (2013). Appearance-Based Loop Closure Detection for Online Large-Scale and Long Term Operation. Obtenido de

<https://introlab.3it.usherbrooke.ca/mediawiki-introlab/images/b/bc/TRO2013.pdf>

Miceli, M. (2011). *Imagen extraída de Introducción a las Pruebas del Software Estrategias y tipos de pruebas.*

Microsoft careers. Obtenido de

<http://www.microsoft-careers.com/content/rebrand/hardware/hardware-story-kinect/>

Morgan Quigley, Brian Gerkeyy, Ken Conley. (2009). *ROS: an open-source Robot Operating System.* Stanford, CA: Computer Science Department, Stanford University.

Motor driver datasheet L298N. Obtenido de

<https://www.sparkfun.com/datasheets/Components/General/L298N.pdf>

Motor gmp36 36mm especificaciones. Obtenido de

<http://spanish.alibaba.com/product-gs/gmp36-tec3650-36mm-brushless-planetary-ger-motor-325329178.html>

MPU6050 Arduino Playground. Obtenido de

<http://playground.arduino.cc/Main/MPU-6050>

Myung Hwangbo, J.-S. K. (s.f.). *IMU-Aided KLT Feature Tracking.* Obtenido de

http://www.cs.cmu.edu/~myung/IMU_KLT/

Norte geográfico y norte magnético. Declinación. Obtenido de

<http://www.cyberphysics.co.uk/topics/magnetsm/Earth.htm>

openSLAM, O. *openSLAM.* Obtenido de

<http://www.openslam.org>

RC explosion. Obtenido de

<http://rcexplosion.foroes.biz/t151-algo-sobre-lipos-y-cargadores>

Rowberg, J. Obtenido de

<https://github.com/jrowberg/i2cdevlib/tree/master/Arduino/MPU6050>

Scaramuzza Davide. *Visual Odometry.* Obtenido de

<sites.google.com/site/scarabotix/>

Sommerville, I. (2011). *Software engineering 9.* Addison-Wesley.

Turner, B. a. (1996). The Role of Experimentation in Software Engineering: Past, Current, and Future. *18th International Conference on Software Engineering, Berlin, Germany.*

Robot móvil autónomo capaz de producir mapas 3D del interior de una habitación

Tutorial magnetómetro HMC5883L . Obtenido de

<https://www.loveelectronics.co.uk/Tutorials/8/hmc5883l-tutorial-and-arduino-library>

Udacity CTE. Obtenido de

<http://forums.udacity.com/questions/1026725/unit-59-what-is-the-crosstrack-error>

Very high speed optocoupler 6N137 datasheet. (s.f.). Obtenido de

<http://www.fairchildsemi.com/ds/6N/6N137.pdf>

Wikimedia. Obtenido de

http://upload.wikimedia.org/wikipedia/commons/d/d4/World_Magnetic_Declination_2010.pdf

Wikimedia. *Intensidad campo magnético terrestre* . Obtenido de

http://upload.wikimedia.org/wikipedia/commons/c/c7/WMM2010_F_MERC.pdf

Wikipedia - Kinect. Obtenido de

<http://en.wikipedia.org/wiki/Kinect>

Wikipedia - *Iterative and Incremental development*. Obtenido de

http://en.wikipedia.org/wiki/Iterative_and_incremental_development

Wikipedia, Diodos flyback. Obtenido de

http://en.wikipedia.org/wiki/Flyback_diode

Wikipedia, Earth's magnetic field. Obtenido de

http://en.wikipedia.org/wiki/Earth's_magnetic_field

Zippy Flightmax 6S1P. Obtenido de

http://www.hobbyking.com/hobbyking/store/_8918_ZIPPY_Flightmax_3000mAh_6S1P_20C.html