Universidad de Valladolid

ESCUELA DE INGENIERÍAS INDUSTRIALES

UNIVERSIDAD DE VALLADOLID

ESCUELA DE INGENIERIAS INDUSTRIALES

Grado en Ingeniería Electrónica Industrial y Automática

Desarrollo de un sistema de control de un robot móvil

Autor:

Jiménez Jiménez, Carlos

Tutor:

Zalama Casanova, Eduardo

Departamento de Ingeniería de Sistemas y Automática

Valladolid, Junio 2017.

Universidad de Valladolid

ESCUELA DE INGENIERÍAS INDUSTRIALES

Carlos Jiménez Jiménez

EDUBOT

CARLOS JIMÉNEZ JIMÉNEZ

Agradecimientos:

Eduardo Zalama Casanova

Roberto Pinillos Herrero

Jaime Gómez García Bermejo

ÍNDICE

CAPÍTULO 1. INTRODUCCIÓN

1.1 Resumen

El presente proyecto consiste en el desarrollo de un sistema de control de bajo y alto nivel de un robot móvil con tracción diferencial. Se llevará a cabo la implementación de los sistemas de comunicación entre el control de bajo nivel (microcontrolador) y el de alto nivel (raspberry PI) así como el protocolo de comunicación entre ambos. Además, se realizará la programación del módulo de abstracción de actuadores y sensores que permita integrar el robot en el entorno de programación "Robot Operating System" (ROS).

La plataforma a automatizar se compone de un chasis, dos motores de corriente continua con realimentación basada en codificador incremental, un conjunto de sensores ultrasónicos dispuestos en la periferia del robot, sensores de contacto en la parte frontal y trasera del robot, cámara frontal para la función sigue líneas y un láser "Hokuyo" para la detección de obstáculos y localización mediante mapas.

Palabras clave: Robótica, Control, Raspberry PI, ROS, Electrónica.

1.2 Justificación

El presente trabajo se enmarca dentro del objetivo de desarrollar una plataforma plenamente operativa que pueda ser usada con fines docentes y de investigación.

En estos campos permitirá, por ejemplo, el desarrollo de nuevas estrategias de control, planificación, localización, modelado del entorno, análisis cinemático y dinámico, etc.

Como puede verse, las posibilidades que ofrecen tanto el chasis como la Raspberry son muy variadas, dando lugar a un sistema muy abierto que permitirá incorporar y probar nuevas tecnologías al robot de una forma muy dinámica y sencilla.

Finalmente, una vez desarrollado, algunas de las aplicaciones que puede tener la robótica móvil son las siguientes:

- El rastreo de zonas peligrosas para los seres humanos de forma remota y la capacidad de recibir datos, imágenes, etc.
- La ejecución de tareas programadas que faciliten el trabajo de las personas.
- La navegación autónoma en entornos de trabajo, con la capacidad de aprendizaje y creación de mapas en tiempo real.

- Relacionado con lo anterior, transporte de objetos de un punto a otro por aprendizaje, con la capacidad de esquivar en tiempo real personas u obstáculos que puedan interponerse en su camino.

1.3 Antecedentes

Sobre esta misma plataforma se han desarrollado varios proyectos final de carrera anteriormente, todos ellos dirigidos al control y manejo de un robot diferencial.

No obstante, a lo largo de este camino se han utilizado tecnologías diferentes:

- David Verdejo Garrido, en 2011, desarrolló su proyecto "Diseño y construcción de un robot didáctico controlado mediante plataforma Player Stage". Utilizó una Raspberry 1 para el control de alto nivel, lo cual se vió que no era suficiente. Además, utilizó dos "puentes H" para el control de los motores y una placa "GPMRC" (General Purpose Mobile Robot Controller), modelo "GPRMC6LC" diseñada en el centro tecnológico CARTIF, que ocupaba mucho espacio en el interior del robot. Para el control del resto de sensores también precisaba de otras placas, todas ellas diseñadas en CARTIF.

- Cristina Blanco Abia, en su proyecto "Desarrollo del sistema de navegación para un robot móvil, en 2014, implementó un sistema de control de alto nivel que se realizaba mediante un ordenador portátil colocado en su superficie, mientras que para el control de bajo nivel utilizaba varias placas controladoras PIC. Utilizó la misma placa controladora "GPMRC6LC" que su predecesor, diseñada en el centro tecnológico Cartif. El principal inconveniente era acarrear un ordenador portátil sobre la parte superior del robot, el gran espacio que ocupaban las placas dentro del robot, además del coste y la dificultad en el manejo.

- Posteriormente, Ana Misiego López, con su proyecto "Control de robots autónomos mediante microcontrolador Arduino" en 2015, desarrolló su plataforma implementando microcontroladores Arduino para el control de motores, ultrasonidos y bumpers, sustituyendo así las placas controladoras PIC que utilizaba su predecesor. Para el control de alto nivel, continuó utilizando un ordenador portátil que iba colocado en la parte superior del robot. Finalmente, se vió que no era posible el control de la plataforma utilizando 2 arduinos, si no que se necesitarían 3, o incluso, 4.

La novedad que supone este proyecto reside en la capacidad de procesamiento de la "Raspberry Pi 3", su bajo coste y tamaño reducido, pudiendo sustituir a un ordenador convencional para el control de esta plataforma. Las ventajas son múltiples:

- ✓ La concentración de todas las tareas de control en un mismo dispositivo.
- ✓ No es necesario, como en los anteriores proyectos, colocar un ordenador portátil sobre la superficie del propio robot para procesar la información.
- ✓ Liberación de espacio y peso, al necesitar únicamente la Raspberry que supone un tamaño muy reducido.
- ✓ La capacidad de acceder remotamente no solo a la información de todos los sensores sino también al control del robot en tiempo real, gracias a la conexión WIFI que nos proporciona la Raspberry.
- ✓ Gracias a la implementación de ROS (Robot Operating System) y a la conectividad WIFI ya citada, cabe la posibilidad de descentralizar alguna tarea que requiera un procesamiento pesado como, por ejemplo, el tratamiento de imágenes o la creación de mapas, y llevarlo a cabo en un ordenador más potente que puede estar situado en cualquier otro lugar de forma remota.

1.4 Objetivos

El objetivo principal es lograr desarrollar el control de alto y bajo nivel de un robot móvil para obtener como resultado una plataforma plenamente operativa, que pueda dar lugar a múltiples opciones de uso.

Como objetivos intermedios se marcan los siguientes:

- La primera fase del proyecto consistirá en la implementación y cableado de la circuitería necesaria para el futuro correcto desarrollo del robot.
- La segunda fase, se basará en el control de la velocidad de los motores de tracción, que se realizará mediante un lazo cerrado y gracias a las referencias que se obtengan de los codificadores incremental.
- La tercera fase se centrará en la implementación de ROS (Robot Operating System) en el control de alto nivel.
- La cuarta fase irá destinada a la incorporación de una cámara y la implementación de un nodo capaz de seguir una línea.
- La quinta y última fase se enfocará hacia el "Stack de Navegación", desarrollando todas las configuraciones necesarias para lograr modelar el entorno y crear un mapa por el que pueda navegarse de forma autónoma.

1.5 Estructura de la memoria

La memoria se estructura en 7 capítulos bien diferenciados, como son:

- Capítulo 1. Introducción: En él se adjunta un pequeño resumen de lo que va a ser el proyecto y se justifica el por qué de su existencia. Se describen de forma general cuales han sido sus antecedentes y se marcan unos pequeños objetivos que se deben ir alcanzando para lograr el objetivo final.

- Capítulo 2. Diseño Previo: En este capítulo no solo se realiza un estudio mecánico de las características previas de la plataforma a automatizar sino también se lleva a cabo un estudio y desarrollo eléctrico y electrónico con el objetivo de que al final de este capítulo, la plataforma sea totalmente operativa y se pueda empezar a programar.

- Capítulo 3. Desarrollo del sistema bajo "*Robot Operating System"*: Se lleva a cabo la implementación del sistema de control tanto de nivel bajo como de nivel alto. Se realiza una pequeña introducción a ROS y se adopta toda la configuración necesaria para poder utilizar el robot bajo este Sistema Operativo. Se describen los nodos principales y, por último, se explicará el "Stack de Navegación".

- Capítulo 4. Resultados: En este capítulo se llevarán a cabo todas las pruebas y experimentos que hayan sido necesarios para ajustar los parámetros del robot. Además, se testearán todas las aplicaciones desarrolladas, principalmente el sigue líneas y la navegación autónoma, para corroborar que se hayan alcanzado los objetivos iniciales.

- Capítulo 5. Manual de Usuario. Pequeña guía de información para que cualquier usuario pueda ser capaz de ejecutar las diferentes aplicaciones del robot.

- Capítulo 6. Estudio económico. Informe detallado del coste que ha supuesto o supondrá la realización de este proyecto.

- Capítulo 7. Conclusiones: En este último capítulo se extraen las conclusiones obtenidas tras la realización del proyecto. Además, se introducirán pequeñas líneas de mejora que puedan ser adoptadas en proyectos futuros.

Además, se incluyen la Bibliografía y los Anexos.

CAPÍTULO 2. DISEÑO PREVIO

2.1 Introducción

Aunque se parte de una plataforma operativa con un sistema de control previo que se quiere mejorar, se precisa de un amplio estudio de las características de la plataforma, no solo para el correcto desarrollo del futuro proyecto, sino también para la evaluación de las posibilidades que la plataforma puede ofrecer. Gracias a la versátil estructura del chasis y también a los orificios de los que dispone su tapa superior, se podrán utilizar multitud de sensores y actuadores, de los cuales habrá que estudiar su óptima localización, tanto desde el punto de vista mecánico y cinemático, como desde el punto de vista eléctrico, para realizar un correcto y ordenado esquema.

En primer lugar, se hablará del diseño mecánico y a continuación se pasará al diseño eléctrico y electrónico.

2.2 Diseño mecánico

Cuando se habla del diseño, cabe destacar su versátil y simétrico chasis, con numerosos y variados orificios para poder implantar nuevas mejoras, integrar nuevas aplicaciones y adaptar todo tipo de sensores y actuadores, tales como ultrasonidos, seta de emergencia, bumpers (sensores de choque), indicadores de batería, etc. Incluso, su tapa superior también cuenta con numerosos orificios que nos pueden permitir colocar también sensores más grandes como láseres o antenas.

2.2.1 Chasis

El chasis, que se muestra en la Figura 1, es la parte principal del cuerpo del robot. Se encarga de soportar todo el peso del robot, de alojar todos los componentes y de darle forma y rigidez. Sin embargo, a la vez que robusto debe ser ligero para que los motores utilizados soporten el movimiento correctamente.

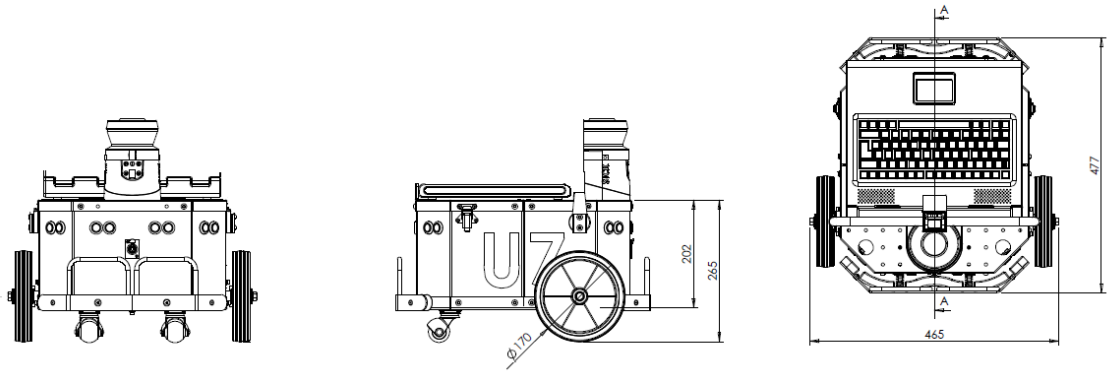


Figura 1 Planos del chasis

Como se puede apreciar, se trata de una estructura robusta con forma hexagonal. Está fabricada en aluminio ya que es un material resistente, no muy pesado y barato. Al mismo tiempo, este material ofrece unas características muy interesantes como, por ejemplo, la resistencia a la corrosión y la capacidad de ser soldado. No obstante, una de las principales ventajas que ofrece es, sin duda, su fácil mecanizado, lo cual puede permitir en cualquier momento, perforar nuevos orificios o modificar alguna parte que interese.

La forma hexagonal de la que se hablaba anteriormente permite colocar sensores en diferentes ángulos para poder disponer de un mayor control del medio que nos rodea.

En su interior, se aprecian claramente dos zonas diferenciadas: la parte delantera se aprovechará para la colocación de los motores como se muestra en la Figura 2, y la parte trasera para la alimentación, aunque esto se explicará con más detalle más adelante. La tapa superior también está dividida a la mitad, gracias a unas bisagras que nos permitirán acceder a ambas divisiones internas fácilmente, sin necesidad de tener que desmontar nada.

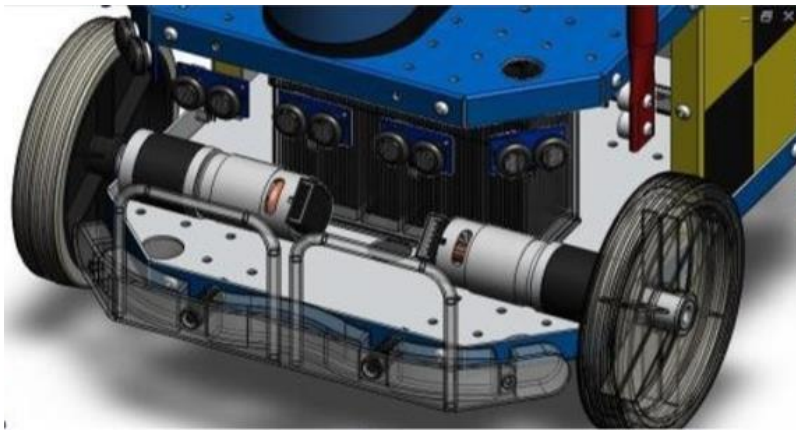


Figura 2 Distribución del robot

2.2.2 Configuración cinemática

Se entiende por configuración cinemática la distribución de las ruedas motrices, en este caso de un robot móvil, que determinará el movimiento y sus limitaciones. Este modelo cinemático será el que permita al robot moverse dentro de un determinado entorno y para ellos contamos con diferentes posibilidades que se detallan brevemente a continuación. [3]

- Configuración diferencial (Figura 3):

  Se caracteriza por ser uno de los modelos más sencillos. En este tipo de configuración no se cuenta con ruedas directrices, solamente dos ruedas diametralmente opuestas en un eje perpendicular a la dirección del robot.

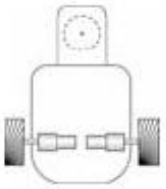


Figura 3 Esquema conf. diferencial

El giro se consigue con la variación de velocidad de cada una de las ruedas, ya que cada cual lleva acoplada un motor diferente.

- Configuración en triciclo (Figura 4):

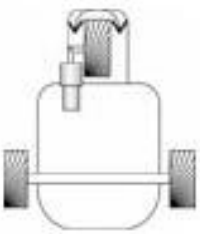


Figura 4 Esquema conf. triciclo

Como su propio nombre indica, este tipo de configuración se caracteriza por constar de tres ruedas, dos de las cuales serán pasivas y una tercera motriz y directriz. Esta última podrá ir acoplada en la parte delantera o trasera del vehículo a mover, pero normalmente las ruedas pasivas van en la parte trasera y la directriz en la delantera.
Como desventaja, este tipo de configuración dota al robot de ciertas restricciones a la hora de realizar múltiples movimientos y de una cierta inestabilidad ya que el centro de gravedad tiende a alejarse de la rueda de tracción en terrenos inclinados.

- Configuración Ackerman (Figura 5):

Es la más frecuente en la industria del automóvil. Consta de cuatro ruedas, dos traseras y dos delanteras. El motor va acoplado a uno de los dos ejes, de ahí que podamos distinguir entre tracción trasera o delantera. Normalmente se adopta esta última configuración debido a que ofrece un mayor control en los giros y puede evitar deslizamientos traseros.

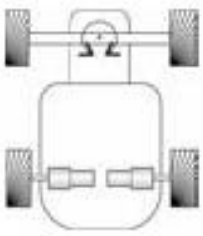


Figura 5 Esquema conf. Ackerman

- Configuración síncrona (Figura 6):

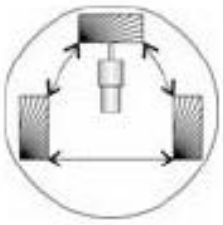


Figura 6 Esquema conf síncrona

Se trata de una configuración poco usual e innovadora. En este modelo todas las ruedas actúan de forma síncrona propulsadas por un mismo motor. De esta forma, todas giran en la misma dirección y a la misma velocidad, por lo que se necesita una gran sincronización a la hora de pivotar para lograr un correcto y deseado movimiento.

- Configuración omnidireccional (Figura 7):

Se dota al robot de ruedas omnidireccionales lo cual radica en la gran libertad de movilidad que puede adoptar en sus movimientos. No obstante, el movimiento en línea recta no está garantizado, se necesita un control previo para lograrlo. Es un diseño algo complejo y caro de realizar.

Figura 7 Esquema conf. omnidireccional

- Configuración Skid-steer (Figura 8):

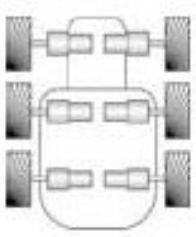


Figura 8 Esquema conf Skid-steer

Finalmente, hablamos de esta última configuración en la cual el robot dispone de dos o más ruedas en cada lado. El giro de las ruedas no está permitido y, por lo tanto, los giros se llevan a cabo de manera similar a como se hace en la configuración diferencial. Para hacernos una idea, un caso concreto es la transmisión por orugas.

En este caso, para el proyecto que se desarrolla, se ha adoptado una configuración diferencial.

Debido a su sencillez, es uno de los sistemas más usuales en cuanto a la configuración de movimiento. Es adecuado para la navegación en entornos de desarrollo típicos de actividades humanas como, por ejemplo, en este caso, entornos de interior como universidades y oficinas.

La tracción diferencial es muy popular y, además, permite calcular la posición exacta del robot mediante unas ecuaciones geométricas sencillas (Figura 9), las cuales se basan no solo en los sistemas de propulsión (en este caso los motores), sino también en aspectos y medidas como los diámetros de las ruedas o las distancias entre ambas. Todos estos temas los engloba el estudio odométrico, que se tratará más adelante. [8]

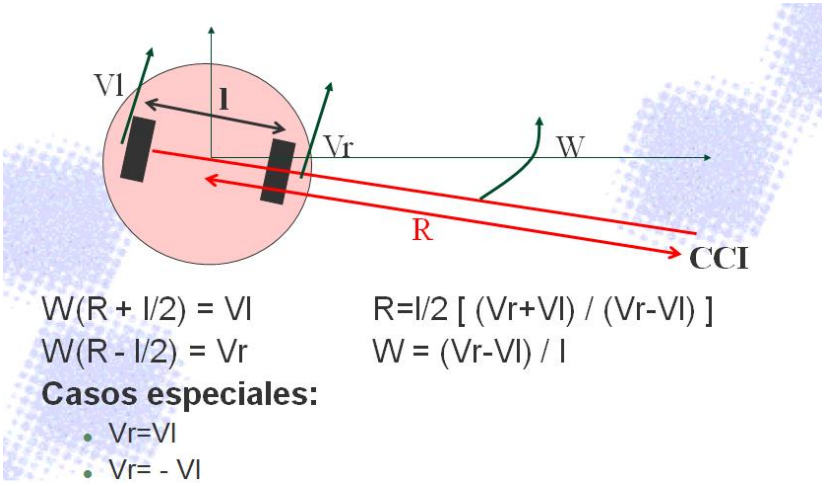


Figura 9 Cinemática de la configuración diferencial

Cabe destacar, que se ha decidido invertir el sentido de la marcha del robot heredado, pasando ahora a ser la parte delantera del robot donde se encuentran las ruedas motrices y la parte trasera donde se encuentran las ruedas locas. Se tomó esta decisión por razones obvias de mejora del movimiento, ya que las ruedas motrices no solo darán propulsión al robot, si no que serán las encargadas del giro, el cual será más preciso si estas se encuentran en la parte delantera. Además, en

tareas como seguir líneas o caminos, el resultado es mucho más satisfactorio si las ruedas motrices se encuentran en la parte delantera.

Por último, se pasa a detallar la disposición de las ruedas, que se muestra en la Figura 10, y sus medidas (Tabla 1):

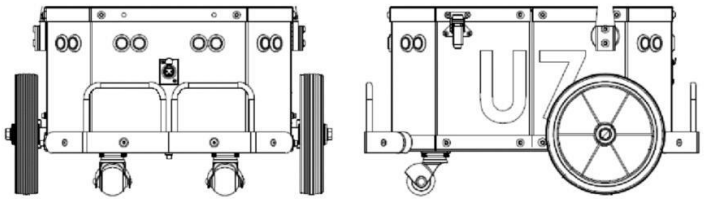


Figura 10 Configuración cinemática

Diámetro de las ruedas motrices:	**14 cm**
Separación entre ruedas motrices:	42 cm
Separación entre las ruedas locas:	6 cm
Distancia entre ejes:	22,5 cm
Altura con respecto al suelo:	24,5 cm

Tabla 1 Medidas de las ruedas y su localización

2.2.3 Motores

A la hora de escoger los motores que accionarían este robot, en su momento se eligió dos motores de corriente continua de los cuales nos ocuparemos un poco más adelante.

Entre otras opciones con las que se contaban, se encontraban los motores de alterna, los motores paso a paso y los servomotores.

Los primeros se descartan debido al coste adicional que supondría instalar circuitos inversores y también debido a que estos motores no se controlan muy bien a bajas velocidades.

La segunda opción, a pesar de la exactitud en cuanto a posición que ofrecen los motores paso a paso, no se puede llevar a cabo debido al poco par que tienen, lo cual llevaría a acoplarles grandes reductoras que elevarían el precio considerablemente.

Finalmente, los servomotores son motores de corriente continua que disponen de sensores que permiten su control en posición. Por tanto, pueden moverse hasta una determinada posición dentro de un rango y permanecer en esa posición de manera estable. No obstante, estos motores no son muy adecuados para las características que se estaban buscando.

Por tanto, entre tantas opciones se tenía que llegar a un compromiso entre coste, dimensiones, peso, potencia, eficiencia y cierto control dentro de los rangos de velocidades en los que se iba a mover el robot, de ahí que se eligiesen los motores de corriente continua.

En la Tabla 2 se puede ver una breve comparación entre las características principales de los motores anteriormente detallados.

Tipo	Control	Potencia	Precisión	Coste
Motor DC	Medio	Alta	Media	Bajo
Motor AC	Fácil	Alta	Media	Alto
Motor Paso a Paso	Fácil	Baja	Alta	Bajo
Servomotor	Fácil	Alta	Alta	Alto

Tabla 2 Comparación principales características de motores

En cuanto a los motores instalados, se tratan de dos motores de corriente continua (DC) que ofrecen las prestaciones suficientes para este tipo de robot. Para superar sus limitaciones, como son las bajas velocidades y el desconocimiento del valor de las mismas, llevan acoplados a los mismos unos pequeños engranajes reductores a la salida, lo cual nos ofrecerá un mayor par y un mejor control a bajas velocidades. Además, también cuentan con codificadores incrementales o encoders, que nos ayudaran a conocer la velocidad de cada una de las ruedas y en un futuro, la posición del robot gracias a los cálculos odométricos.

Como breve introducción a este tipo de motores, aquí van unas pequeñas pinceladas teóricas de los motores DC:

Como todo motor, es un dispositivo encargado de transformar la energía eléctrica en energía mecánica. Este tipo de motores están formados por dos partes principales: el rotor y el estator. Un esquema básico de este tipo de motor se muestra en la Figura 11.

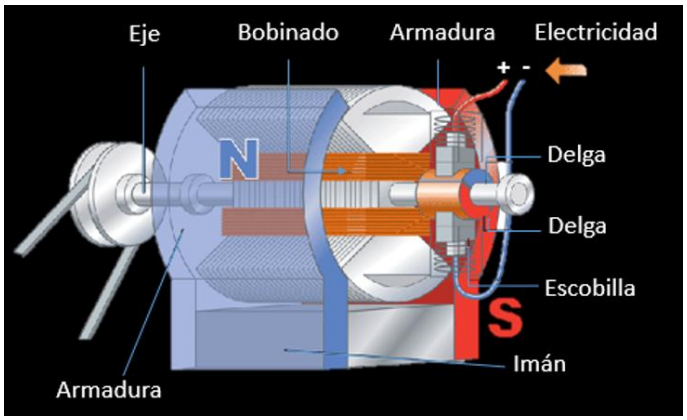


Figura 11 Esquema Motor DC

El primero constituye la parte móvil del motor, como su propio nombre indica. Está formado por un eje con un núcleo laminado sobre el que se enrollan las bobinas del devanado conectadas a un extremo del colector. El giro de este, conmuta los terminales con los del motor para conectarlas en sentido adecuado.

El estator es la parte fija también llamado imán permanente. Se encarga de suministrar el flujo magnético que será usado por el bobinado del rotor para realizar su movimiento giratorio (Figura 12). Las escobillas se encuentran fijadas en el lateral del armazón donde se encuentre el colector y su función es transmitir la tensión de los terminales del motor a las láminas del colector para conmutar la tensión del devanado.

Por tanto, la alimentación del motor es a través de dos bornes y para cambiar el sentido de giro basta con invertir la polaridad de la alimentación.

Figura 12 Funcionamiento Motor DC

Se basa simplemente en que el campo magnético generado por la bobina tenderá a alinearse con el del estator haciendo que el rotor se mueva generando un par en el eje del motor. Cuando los campos estén casi alineados el colector invertirá la polaridad de la bobina y el giro continuará.

Así, cuando hablamos de par motor, nos referimos a la fuerza con que se atraigan o se repelan los campos magnéticos del rotor y del estator. Se debe tener en cuenta que el campo generado por el estator es proporcional a la intensidad que circula por él, por lo tanto, a más intensidad, más fuerza se desarrollará. La curva típica de los motores es la de Par frente a intensidad del inducido, como se muestra en la Figura 13.

Figura 13 Par motor frente a intensidad del inducido

2.2.4 Cálculo de los motores

A continuación, se pasará a explicar y realizar los cálculos pertinentes para saber que par debe suministrar cada motor.

Especificaciones técnicas:

- Velocidad esperada (V): se ha decidido establecerla en $1\,{}^{m}/_{s}$
- Aceleración esperada (a): fijada en $0.6\,{}^{m}/_{s^2}$
- Posible pendiente a superar (p): de unos $5º$

Características del robot:

- Masa total (M):

$$M = m_{robot} + m_{carga} = 10kg + 10kg = 20kg$$

- Eficiencia de las reductoras o rendimiento:

$$\eta = 0.8$$

- Número de ruedas motrices

$$n = 2$$

- Radio de ruedas motrices:

$$r = 0.07m$$

- Rozamiento del cojinete:

$$\mu = 0.01$$

Cuando se habla de la elección de un motor, hay tres aspectos fundamentales a tener en cuenta, como son: la velocidad, el par y la eficiencia energética.

La velocidad de un motor siempre se refiere a la velocidad angular de su eje de giro y en el Sistema Internacional (SI) viene dada en m/s aunque hay otra unidad de uso muy frecuente en los motores que es en revoluciones por minuto (rpm).

Como se ha explicado anteriormente, el par de un motor está relacionado con la fuerza que se atraen los campos magnéticos del estator y del rotor. Este concepto llevado a una aplicación se refiere a la fuerza que hay que ejercer para mover una carga colocada a una cierta distancia del eje del motor. En este caso, el par motor viene dado por la expresión 1:

$$Par\ motor\ (T) = Fuerza\ (F) \cdot Distancia\ (d) \qquad (\mathbf{1})$$

En el SI el par viene dado en N/m.

Además, es importante distinguir entre dos tipos de par:

- *Stall Torque* o Par de arranque o de bloqueo, es aquel que el motor puede ejercer cuando está parado, es decir, máximo par capaz de ejercer para comenzar el movimiento.
- *Rated Torque* o Par nominal, es aquel que puede proporcionar el motor a la velocidad nominal, es decir, a la velocidad para la que ha sido diseñado.

La última consideración que se debe tener antes de comenzar los cálculos, es la relación que existe entre el par motor y el diámetro de las ruedas. Cuanto mayor es el radio de las ruedas mayor será la velocidad lineal y menor la fuerza ejercida por el motor. Esto se relaciona con la ecuación de la velocidad angular (2) y la ecuación del par (1):

$$Velocidad\ lineal\ (V) = Velocidad\ angular\ (w) * Radio\ rueda\ (r) \qquad (\mathbf{2})$$

<u>Cálculos:</u>

Con los datos que se disponen, lo primero será hallar la velocidad angular que tendrán que proporcionar los motores.

$$w\ (rpm) = \frac{V}{2 \cdot \pi \cdot r} = \frac{1\ m/s}{2 \cdot \pi \cdot 0{,}07m} \cdot 60\frac{s}{min} = 136.41\ rpm$$

A continuación, se debe relacionar la fuerza que deben ejercer los motores en función del peso del robot y la velocidad que queremos conseguir. Para ello, se pasará a calcular el par motor o torque necesario que viene dado por la expresión 3:

$$M_{in} = \frac{d}{2} \cdot \frac{F_L}{\eta} \qquad (\mathbf{3})$$

Siendo F_L la fuerza que deberán desarrolar los motores y que viene dada por la expresión 4:

$$F_L = m \cdot g \cdot sen(\alpha) + \mu \cdot m \cdot g \cdot \cos(\alpha) \quad (\mathbf{4})$$

$$F_L = 20 \cdot 9.81 \cdot sen(5) + 0.01 \cdot 20 \cdot 9.81 \cdot \cos(5) + m * a = 31.05\ N$$

Por lo que el par será:

$$M_{in} = \frac{0.14m}{2} \cdot \frac{31.05\ N}{0.8} = 2.7172\ N \cdot m$$

Sin embargo, al disponer de una configuración cinemática de tracción diferencial, el robot cuenta con 2 ruedas motrices cada cual lleva acoplada un motor independiente, de forma que el par que deberá desarrollar cada uno será la mitad del anteriormente calculado:

$$M_{m1} = M_{m2} = \frac{2.7172\ N \cdot m}{2} = 1.3586\ \approx 1.36\ N \cdot m$$

En cuanto a la potencia que deben desarrollar los motores (Pm) que debe ser mayor que la potencia mecánica del robot (Pr):

$$Pm > Pr$$

La ecuación 5 liga todos estos aspectos:

$$Pr = \frac{Masa(M) \cdot Aceleración(a) \cdot Velocidad(v)}{2 \cdot \pi \cdot Eficiencia(e)} \quad (\mathbf{5})$$

$$Pr = \frac{20kg \cdot 0.6m/s^2 \cdot 1m/s}{2 \cdot \pi \cdot 0.8} = 2{,}38\ W$$

Pero, además, si se quiere tener en cuenta una posible inclinación de 5º como se nos indica en las especificaciones, se deberá introducir este factor en la ecuación:

$$Pr = \frac{Masa(M) \cdot (Aceleración(a) + gravedad(g) \cdot sen(i) \cdot Velocidad(v)}{2 \cdot \pi \cdot Eficiencia(e)}$$

$$Pr = \frac{20kg \cdot (0{,}6\ \frac{m}{s^2} + 9.81 \cdot sen(5)) \cdot 1m/s}{2 \cdot \pi \cdot 0.8} = 5{,}78\ W$$

Por lo tanto, de este cálculo deducimos que para cumplir las especificaciones indicadas en llano nos bastará con una potencia superior a 2.38 W mientras que si queremos que las cumpla también con una inclinación de 5º deberemos colocar motores con potencia superior a 2.78 W.

2.2.5 Elección de los motores

Para este robot se decidió utilizar la familia de motores IGM. Para seleccionar el motor que mejor se adapta, se recurre al catálogo del fabricante donde se pueden observar las diferentes características de cada motor y, en función de los cálculos realizados anteriormente, escoger el motor correcto.

Figura 15 Motor escogido

Más concretamente, se seleccionó el motor IG-42GM. Este motor presenta dos modelos comerciales, uno para 12V y otro para 24V. En este caso se eligió el de 12V (Figura 15) por cuestiones eléctricas que se explicarán más adelante.

En el Anexo A puede verse la hoja de características de este motor. La Figura 14 muestra, a modo de resumen, las características principales de este motor:

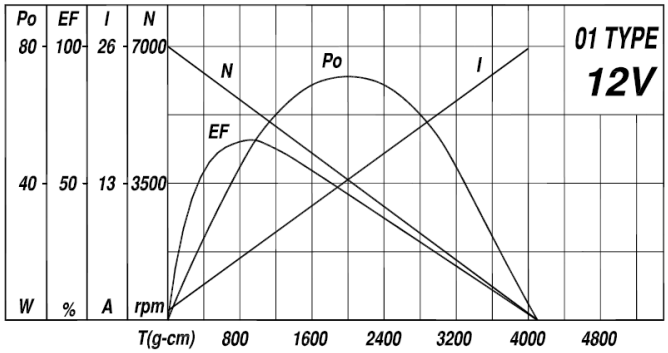


Figura 14 Principales curvas del Motor IG-42GM 01

Por tanto, con los cálculos realizados y la elección de los motores, queda demostrado que los motores que lleva el robot son aptos para su correcto funcionamiento.

2.3 Diseño eléctrico

A continuación, se describirán las características de alimentación para los diferentes elementos del robot.

Edubot, que es el nombre que recibe este robot, cuenta con una batería de 12 Voltios y 17 Ah que va colocada en la semiparte trasera del robot (Figura 16), intentando siempre que quede lo más centrada posible respecto al eje longitudinal del robot para compensar así, los posibles pesos y contrapesos, de forma que ninguno de los dos motores tenga que realizar más trabajo que el otro, ya que la batería es, sin duda, el elemento más pesado del robot.

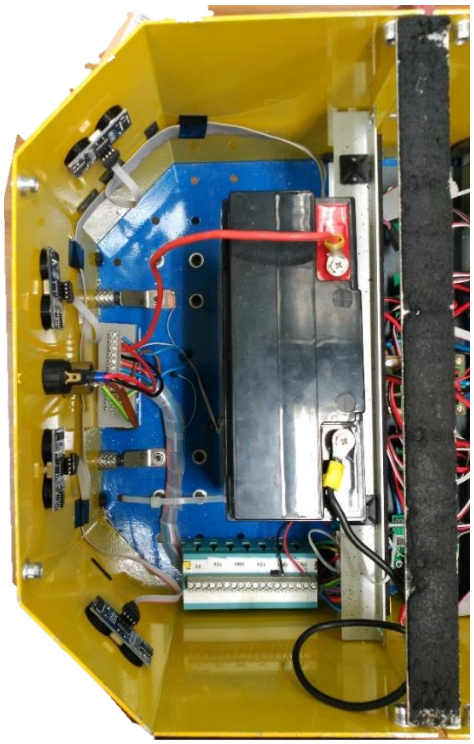


Figura 16 Distribución trasera del robot

Será la encargada de alimentar todo el robot, es decir, todos los sensores, placas controladoras, Raspberry y, como no, los motores. Para ello contamos además con un regulador de tensión (Figura 17) que será el encargado de convertir los 12 Voltios a 5V, para poder alimentar así la mayoría de componentes que exigen esta tensión. Ofrece 1 Amperio de intensidad lo cual se considera suficiente ya que los mayores consumos residen en la Raspberry Pi 3 (350mA máx) y el láser Hokuyo (500mA máx).

Figura 17 Regulador de tensión

En términos generales, información que quedará perfectamente documentada con el esquema eléctrico y los documentos gráficos, el robot se ha dividido en 2 grandes zonas, aprovechando la separación intermedia con la que cuenta:

- En la parte trasera se localiza la etapa de potencia, donde se encuentra en la propia batería, la clavija para el cargador de la misma (Figura 18) y dos borneras para facilitar las conexiones eléctricas. La bornera posterior, inaccesible para el usuario, localiza las conexiones principales, tanto de tierra como de 12 Voltios. Conecta de esta manera, la batería con la toma del cargador y distribuye varias conexiones de 12V que serán utilizadas en otras partes del circuito.

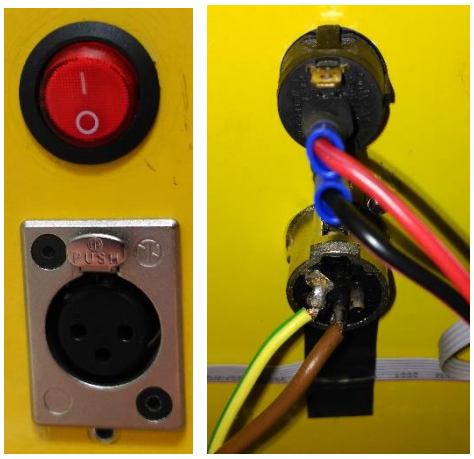


Figura 18 Interruptor general y cargador, parte exterior (imagen izquierda), parte interior (imagen derecha).

Todo este circuito de potencia cuenta con la protección de unos fusibles por si se diese el caso en algún momento de una sobre-intensidad, estos se fundirían desconectando el circuito y protegiendo, de esta manera, el resto de elementos del robot.

Figura 19 Fusibles

Finalmente, la parte trasera cuenta con otra bornera, esta sí, totalmente accesible, en la que el usuario contará con diferentes conexiones tanto de GND como de 12V e, incluso, de 5V. Esta última, como se puede sobreentender y que también queda reflejado en el esquema, viene, como no podía ser de otra manera, de una de las salidas del regulador de tensión.

- La parte delantera es donde se localiza toda la parte de la electrónica propiamente dicha, a excepción de los motores.

Se decidió dividir este espacio en 2 niveles, un piso inferior y otro superior. Están separados por una lámina de metacrilato perfectamente encuadrada y atornillada como se puede apreciar en la Figura 20.

En el piso inferior se encuentra todo el nivel bajo de la electrónica, consiguiendo de esta manera que sea inaccesible para el usuario. Ahí se localizan los motores con sus respectivas reductoras y codificadores incrementales, el regulador de tensión, las placas controladoras de los motores y las placas controladoras de ultrasonidos.

La parte superior queda reservada para la Raspberry (Figura 20), con la comodidad que eso supone, pudiendo acceder a ella de manera muy simple para realizar cualquier ajuste.

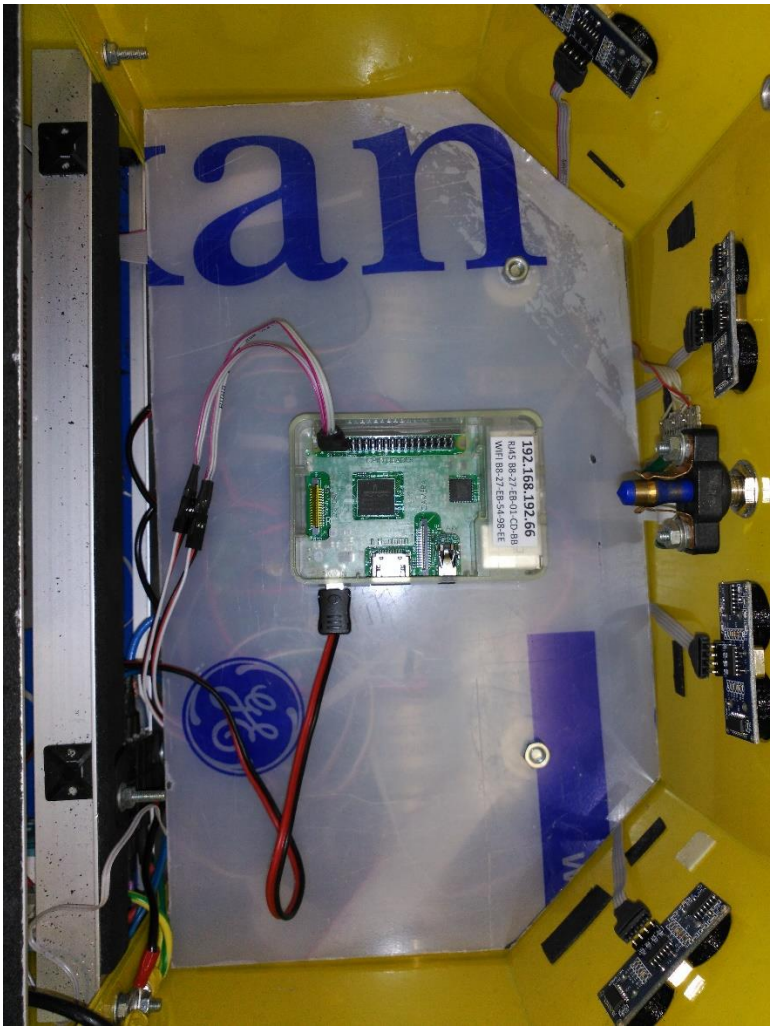


Figura 20 Piso superior

2.4 Diseño electrónico

En este apartado se desarrollará toda la parte electrónica del proyecto, desde los sensores que lleva incorporado el robot, pasando por las controladoras de los motores y ultrasonidos hasta el controlador principal, la Raspberry.

2.4.1 Sensores

Sonar

El sensor ultrasónico (Figura 21) se enmarca dentro de los sensores para medir distancias o detectar obstáculos. En este proyecto, como se utiliza un láser Hokuyo para el mapeo de zonas, utilizaremos los sonar para para prevenir posibles choques.

Figura 21 Ultrasonídos

El funcionamiento es simple (Figura 22): el sensor dispone de dos pequeños cilindros, uno de los cuales trabaja como emisor y el otro como receptor. El primero emite ondas ultrasónicas, inaudibles para el ser humano debido a su alta frecuencia, mientras que el segundo estará esperando a que dicho sonido rebote sobre objetos y vuelva. Como se conoce la velocidad del sonido y es posible conocer el tiempo que ha tardado en rebotar la onda, se puede saber la distancia a la que se encuentra el objeto u obstáculo. Estos tipos de sensores son capaces de medir distancias desde unos pocos centímetros hasta unos pocos metros. Los sensores con los que contamos trabajan en un rango de 3 cm a 3 m con una precisión de 3 mm.

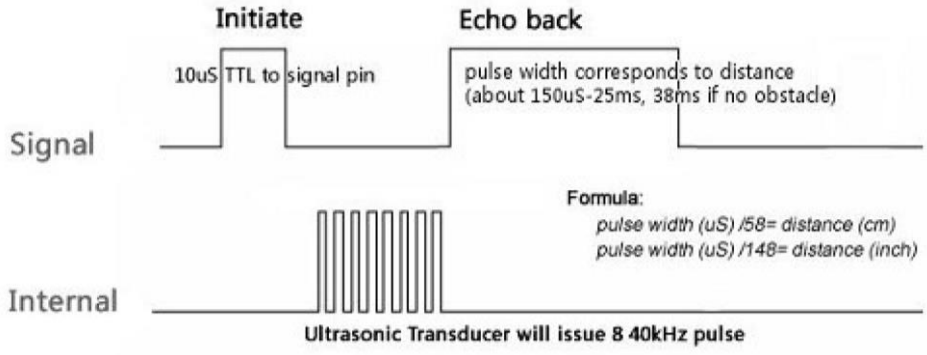


Figura 22 Funcionamiento de un sensor ultrasónico

Los ultrasonidos utilizados cuentan con 4 pines, 2 de ellos para alimentación 5 V (Vcc) y tierra (GND), mientras que los otros 2 quedan reservados para enviar una onda (trig) y para recibir el rebote (echo). [2]

El robot cuenta con 8 ultrasonidos en su periferia, 4 en la parte delantera y 4 en la parte trasera, como se puede ver en la Figura 23.

Figura 23 Localización de los ultrasonidos

Bumper

Se trata de un sensor de choque, ya que se activan cuando son presionados. El bumper es un conmutador de dos posiciones con muelle de retorno a la posición de reposo. En estado de reposo la patilla común y la de reposo están en contacto, mientras que cuando se activa debido a una presión externa, el bumper cambia de estado y pasa a la posición de contacto, cerrando un pequeño circuito que se alimenta a 5V para mandar una señal al controlador correspondiente. Entiéndase que este tipo de sensor es un todo o nada, o hay choque o no lo hay.

En el robot que se está desarrollando, este tipo de sensor lo utilizamos tanto en la parte trasera como en la parte delantera a modo de seguridad, ya que en caso de que falle el láser o los ultrasonidos y el robot llegue a impactar con algún objeto, los bumpers se activarán y mandaran una pequeña señal eléctrica al controlador que mandará para al robot.

Codificador incremental

Un codificador incremental (Figura 24) o también llamado encoder, es un dispositivo usado para convertir la posición angular de un eje a un código digital. Sus aplicaciones son muy diversas, se utilizan en robótica, en lentes fotográficas, en periféricos del ordenador como el ratón, etc. Pero una de las principales y que se aplica en este proyecto es la de su acoplamiento a la salida del eje de un motor, en este caso de un motor de corriente continua, para conocer no solo su velocidad si no también su posición.

Figura 24 Encoder

Hay dos tipos de codificadores: incrementales y absolutos. Estos últimos producen un código digital único para cada ángulo distinto del eje y, por tanto, no necesitan ninguna referencia de partida. No obstante, nos centraremos en los incrementales que son los que se emplean.

En el caso de los incrementales, la posición del disco acoplado al eje se determina por el número de pulsos producidos desde una determinada posición, es decir, en este caso partimos de una referencia. También dentro de este tipo de codificadores podemos encontrar variedades en la forma de generar los pulsos como, por ejemplo, ópticos o magnéticos. Los que se utilizan para este proyecto son codificadores incrementales magnéticos.

Los codificadores incrementales cuentan con dos o tres canales, dos de ellos para la lectura de posición y sentido de giro y otro para ubicar la posición origen en el disco. Los 2 canales principales, que llamaremos A y B, están desfasados 90º de forma que basta con contar los flancos de subida (o bajada) de uno de los dos canales y fijarnos en el estado del otro canal para determinar la dirección de giro. Se puede duplicar la resolución leyendo los flancos de subida (o bajada) de los dos canales e, incluso, cuadruplicarla, leyendo tanto los flancos de subida como de bajada.

A continuación, se muestra un pequeño ejemplo para aclararlo:

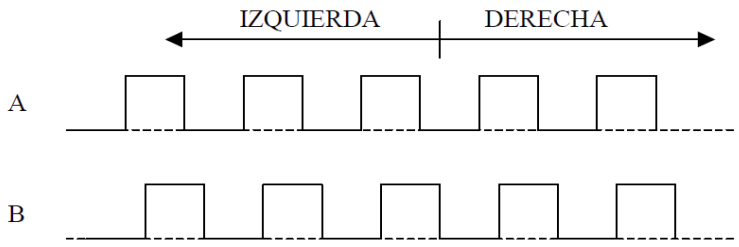


Figura 25 Lectura del encoder

- Si tenemos flanco de subida en A
  - Si B=0, desplazamiento de un paso a la derecha
  - Si B=1, desplazamiento de un paso a la izquierda
- Si tenemos flanco de bajada en A
  - Si B=0, desplazamiento de un paso a la izquierda
  - Si B=1, desplazamiento de un paso a la derecha
- Si tenemos flanco de subida en B
  - Si A=0, desplazamiento de un paso a la izquierda
  - Si A=1, desplazamiento de un paso a la derecha
- Si tenemos flanco de bajada en B
  - Si A=0, desplazamiento de un paso a la derecha
  - Si A=1, desplazamiento de un paso a la izquierda

Cámara

También se ha decidido incorporar a este robot una pequeña cámara, como la que se muestra en la Figura 26, que se conecta directamente a la Raspberry. Con ella no solo se pretende conseguir utilidades como el seguimiento de líneas o circuitos, sino que también se podrá utilizar para la visión remota de zonas o lugares que pueden ser peligrosos para las personas. Y no solo eso, ya que gracias a la Visión Artificial se abre un gran abanico de posibilidades, con aplicaciones como detección de códigos, logotipos, señales, etc.

Figura 26 Cámara de la Raspberry

Hokuyo

Se trata de un escáner láser 2D (Figura 27) ideal para aplicaciones de robótica, con un área de trabajo de hasta 240º de visión y una distancia de hasta 4 metros.

La principal aplicación del láser es crear un mapa del entorno, conociendo en todo momento las distancias a los objetos. Gracias a ROS resulta muy fácil crear y guardar un mapa por el que después se podrá navegar.

Figura 27 Hokuyo

2.4.2 Controladores

Dentro de este apartado se va a distinguir entre dos niveles muy diferenciados.

El control de nivel bajo será llevado a cabo por 4 plaquitas: 2 para los motores (PicoBorg Reverse) y otras 2 para los sonar (Picoborg Ultraborg). El controlador de nivel alto será la Raspberry.

Picoborg Reverse

Se trata de un controlador para motor avanzado que puede utilizarse con la Raspberry Pi. Permite la posibilidad de manejar incluso 2 motores, desde pequeños motores de continua de poca potencia hasta motores de 350W con altos torques. En la Figura 28 puede verse un esquema del conexionado.

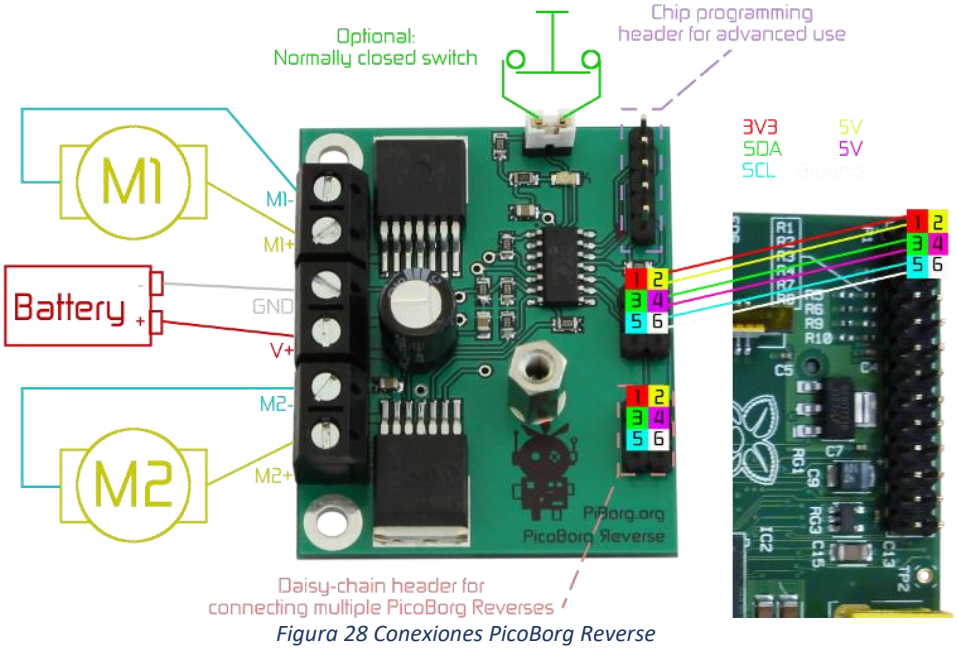


Figura 28 Conexiones PicoBorg Reverse

No obstante, en el proyecto se utilizan 2 placas de este tipo, ya que se emplean una de ellas para cada motor, de forma que se conecta la batería a la toma de alimentación (GND y V+) y en paralelo los terminales del motor a las 4 entradas M1-, M1+, M2- y M2+ para obtener la corriente suficiente. Ha sido necesario reprogramar el software de las controladoras, como se explicará un poco más adelante.

En la parte superior, esta placa cuenta con un jumper, que desarrolla la funcionalidad equivalente a la seta de emergencia. Por esta razón, los jumpers de ambas tarjetas están conectados a la seta de emergencia física del robot, la cual está situada en la parte exterior del robot. De esta forma, cuando pulsemos la seta, ambas controladoras darán la señal de emergencia y detendrán los motores instantáneamente.

En la parte superior derecha, se encuentran los pines no solo de programación de las controladoras si no también donde se conectarán los codificadores incrementales. Esto es, las placas llevan preinstalado un software para controlar los motores como se ha explicado, aunque nos permite la reprogramación de ese software para poder añadir, eliminar o modificar partes de código según nuestras necesidades. En este caso, se han reprogramado entre otras razones, para obtener diferentes alarmas, para controlar un motor con cada placa en lugar de dos y para cambiar su comportamiento en caso de que no reciban referencia alguna. El código aparece en el Anexo C.

De esta forma y una vez reprogramadas, la conexión de los codificadores incrementales se muestra en la Figura 29:

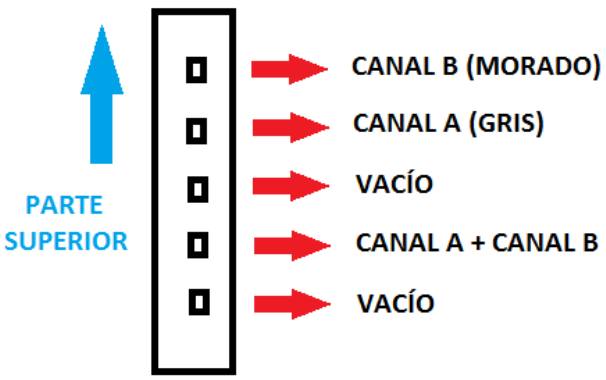


Figura 29 Conexión encoders en PicoBorg

Las conexiones restantes se indican en la Figura 28. Se corresponden con la alimentación y la conexión I2C con la Raspberry. Se distinguen 2 grupos de conexiones con los pines numerados del 1 al 6. El primer grupo es la conexión con la propia Raspbery y el segundo grupo, justo debajo, se utiliza para el conexionado de placas sucesivas.

Picoborg Ultraborg

Al igual que las controladoras de los motores, estas placas (Figura 30) también se comunican mediante I2C con la Raspberry, permitiendo conectar 4 sensores de ultrasonidos y/o 4 servos. Al ser del mismo fabricante que las "*PicoBorg Reverese*" cuetan con la ventaja de poder conectarse una tras otra. Por tanto, será muy cómodo conectar a la Raspberry las cuatro placas de bajo nivel, las dos "*Ultraborg*" y las dos de los motores.

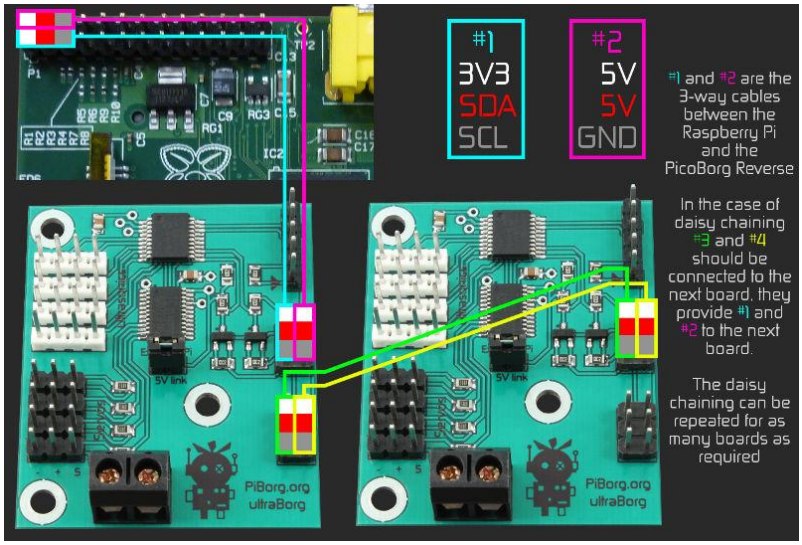


Figura 30 Descripción y conexiones Ultraborg

RASPBERRY PI 3

Se trata del controlador principal (Figura 31). Será la encargada de procesar toda la información y dar órdenes a los motores.

Figura 31 Raspberry Pi 3

La Raspberry es un ordenador de placa única, reducida, simple y de bajo coste desarrollado en Reino Unido. No queda claro si es de hardware libre pero sí de

software, y gracias a la gran comunidad que hay detrás se pueden llegar a desarrollar multitud de proyectos.

Como computador que es cuenta con todo lo necesario para comportarse como tal, así por ejemplo la Raspberry Pi 3 que se está utilizando cuenta con: 1 GB de memoria RAM, procesador de 64 bits de 1.2 GHz de cuatro núcleos, 4 puertos USB 2.0, puerto HDMI, puerto Ethernet, WIFI, Bluetooth, salida de audio jack de 3.5mm y ranura de tarjeta de memoria MicroSD.

Debido a sus limitaciones de potencia, es ideal para controlar pequeños proyectos o automatizar algunas tareas como en este caso nuestro robot.

Cuentan con software libre y soportan sistemas operativos como "Raspbian" "Linux", que son los más populares y conocidos, aunque también, "OSMC", "Ubuntu Mate", etc.

La Raspberry se encargará no solo del control de las placas controladoras de motores y ultrasonidos, sino también de todos aquellos sensores que se van a ir incluyendo, como pueden ser la cámara, el láser Hokuyo, etc….

La comunicación de ésta con las 4 placas controladoras será mediante la conexión I2C [12], con la comodidad que ello supone. A modo de resumen, la comunicación I2C se basa en 2 canales, uno de transmisión de datos (SDA) y otro que marca los ciclos de reloj (SCL) para la perfecta sincronización. Consta de un maestro y varios esclavos. A cada dispositivo o esclavo que se conecta al I2C se le asigna una dirección, para que posteriormente, utilizando el protocolo de transmisión de datos adecuado, el maestro pueda comunicarse con cualquier esclavo y, de esta forma, solicitarle o enviarle información. En la Figura 32 puede verse un esquema de este tipo de comunicación.

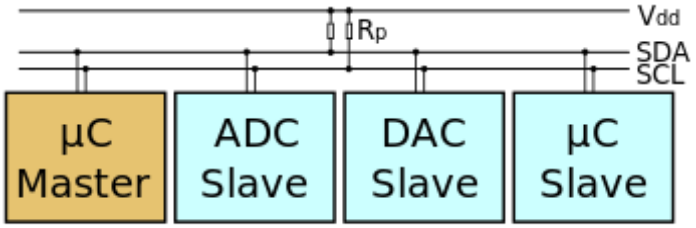


Figura 32 Esquema de la comunicación I2C

Finalmente hablaremos de ROS (Robot Operating System) que, a pesar del significado de sus siglas, no se trata de un sistema operativo como tal, sino de un conjunto de paquetes y un espacio de trabajo que junto con cualquier distribución de Linux ofrece una experiencia muy cómoda y adecuada para robots. Textualmente los creadores lo definen como: "*Un marco flexible para el desarrollo de software de Robots. Es una colección de herramientas, librerías y convenciones que tienen como objetivo simplificar las tareas de creación de un sistema robótico complejo y robusto, a través de una amplia variedad de plataformas robóticas.*" [10]

CAPÍTULO 3. DESARROLLO DEL SISTEMA BAJO *ROBOT OPERATING SYSTEM*

En primer lugar, como ya indicamos anteriormente, ROS es un framework utilizado para desarrollar aplicaciones robóticas. Entre sus principales características destacan que es de código abierto, se encarga de realizar la abstracción del hardware y permite desarrollar una computación distribuida. [10]

Además, proporciona librerías y herramientas para el desarrollo de programas, con multitud de recursos disponibles en la red.

Desde su lanzamiento inicial en 2007, en el que inicialmente recibió el nombre de "switchyard" por el Laboratorio de Inteligencia Artificial de Stanford, ROS ha ido evolucionando de una forma rápida y dinámica a través de sus diferentes distribuciones más o menos estables con una media de un año por lanzamiento. Actualmente, "*Kinetic Kame*" es la versión más reciente y estable con la que se puede trabajar y, por consiguiente, la que llevará instalada la Raspberry. No obstante, está a punto de salir una nueva versión "*Lunar Loggerhead*" con las ventajas y desventajas que eso supone al inicio.

Como dato curioso, destaca que la primera letra del nombre que se va dando a las distribuciones de ROS sigue un orden alfabético. Así, comenzaron con la excepción de "*Ros 1.0*" a la que siguieron "**B**ox Turtle", "**C** Turtle", "**D**iamondback", etc. hasta llegar a las últimas versiones "**I***ndigo", "**J***ade" y "**K***inetic".*

3.1 INSTALACIÓN Y CONFIGURACIÓN DEL ENTORNO ROS

A pesar del significado de sus siglas, ROS no es un sistema operativo como tal y necesita una plataforma para poder instalarse y desarrollarse. Actualmente las plataformas más estables que soportan ROS son "*Ubuntu" y "Debian"*, aunque se encuentran en fase experimental *"OS x", "Gento" y "OpenEmbedded".*

En el caso de este proyecto se va a utilizar "*Ubuntu*" pero, además, si se quiere que funcione todo correctamente para soportar "*Kinetic*" se necesita la versión 16.04 de "*Ubuntu*".

Gracias a la magnífica web del propio ROS [10][1], se puede encontrar multitud de información y tutoriales [9], además de una sencilla guía de instalación, que se resume a continuación:

1) Configuración de la Raspberry para aceptar software de "*packages.ros.org*".

```
sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) mai
n" > /etc/apt/sources.list.d/ros-latest.list'
```

2) Asegurarse de que están todos los paquetes actualizados:

```
sudo apt-get update
```

3) *"Desktop-Full Install"*, que incluye "*ROS", "rviz", "rqt", "2D/3D simulators", "navigation",* etc.

```
sudo apt-get install ros-kinetic-desktop-full
```

A continuación, toca configurar el entorno para una correcta posterior utilización. Recién acabada la instalación de ROS, los paquetes se encuentran en el directorio "*/opt/ros/kinetic/"*. Para referirse a ellos de una manera rápida y sencilla debemos ubicarlos de alguna manera para que el terminal donde estemos programando sepa dónde están esos paquetes. Esto se consigue ejecutando el "*setup.bash":*

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

No obstante, para no tener que realizar esta operación cada vez que se abra un terminal nuevo, la solución más sencilla es añadir esta línea al fichero "*. bashrc"*, que se ejecuta automáticamente cada vez que se abre un nuevo terminal.

Para poder trabajar correctamente, se debe crear un espacio de trabajo, un entorno ROS en el que colocar los paquetes. Se denomina "*catkin_workspace"* y se trata de una carpeta donde modificar, construir e instalar paquetes. Realmente, los paquetes pueden ser compilados como proyectos independientes pero la ventaja que realmente ofrece el "*catkin"* es el concepto de "espacio de trabajo", que permite compilar múltiples e independientes paquetes juntos y a la vez.

Para crear el "catkin_workspace"*:*

1) Se crea el directorio:

```
mkdir -p ~/catkin_ws/src

cd ~/catkin_ws/src
```

2) Aunque el directorio este vacío, compilamos el espacio de trabajo con el comando:

```
cd ~/catkin_ws/

catkin_make
```

De esta forma, se crean en el directorio las carpetas "*build" y "devel"*.

3) Dentro de la última carpeta citada, se pueden ver varios ficheros "*setup.*sh*". Ejecutando cualquiera de estos ficheros se superpondrá este espacio de trabajo en la parte superior del entorno. De esta forma:

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

4) Para asegurarse de que el espacio de trabajo está correctamente configurado y superpuesto se debe ejecutar el siguiente comando:

```
echo $ROS_PACKAGE_PATH
```

y comprobar que la variable incluye el directorio en el que estamos, algo como lo siguiente:

```
/home/youruser/catkin_ws/src:/opt/ros/kinetic/share
```

3.2 ARQUITECTURA DE ROS. FUNCIONAMIENTO GENERAL.

Una vez instalado ROS, en este apartado se explicará la estructura de ROS y qué partes tiene [1].

La arquitectura de ROS ha sido diseñada y dividida en 3 secciones o niveles de conceptos:

- El nivel "*Filesystem*"
- El nivel *"Computation Graph"*
- Y, el nivel *"Community"*

3.2.1 Filesystem Level

En este nivel, se usa un grupo de conceptos para explicar cómo ROS está formado internamente, la estructura de carpetas y los ficheros mínimos que se necesitan para funcionar, como puede verse en la Figura 33.

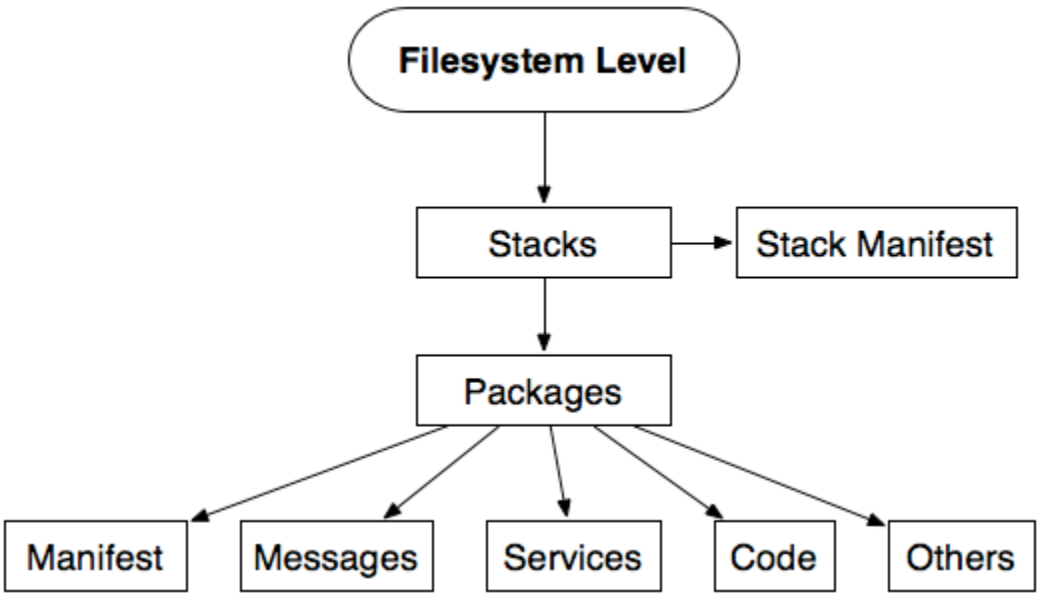


Figura 33 Estructura del nivel "Filesystem"

De forma similar a un sistema operativo, un programa de ROS está dividido en carpetas y estas carpetas contiene algunos ficheros que describen su funcionamiento, que son las siguientes:

- **Packages:** los paquetes forman el nivel atómico de ROS. Un paquete tiene la mínima estructura y contenido para crear un programa dentro de ROS. Pueden tener procesos de tiempo de ejecución (nodos), archivos de configuración, etc.
- **Manifests:** proporcionan información sobre un paquete, información de licencia, dependencias, "flags" del compilador, y mucho más. Se gestionan con un fichero llamado "manifests.xml".
- **Stacks:** Cuando se reúnen varios paquetes con alguna funcionalidad, se obtiene una pila. En ROS existe una gran cantidad de estas pilas con diferentes usos como, por ejemplo, la pila de navegación, muy importante.
- **Stacks manifests:** al igual que antes, proporcionan información, pero en este caso sobre una pila, incluyendo información de licencias y sus dependencias de otras pilas. Son gestionados en el "stack.xml"
- **Messages (msg) types:** Un mensaje es la información que un proceso manda a otro u otros procesos. Son estructuras de datos simples que se pasan entre los nodos. Es el contenido de los "topics". ROS tiene muchos tipos estándar de mensajes. La descripción de los mensajes se encuentra en "*my_package/msg/MyMessageType.msg".*
- **Services (srv) types**: Los servicios siguen una estructura de tipo cliente/servidor entre los nodos. Realmente utilizan dos mensajes, uno en la solicitud y otro en la respuesta. El nodo "servidor" se mantiene a la espera de las solicitudes hasta que el nodo "cliente" realiza una solicitud. Es entonces cuando el "servidor" realiza un procesamiento y responde al cliente.
  Las descripciones de los servicios se encuentran almacenadas en "*my_package/srv/MyServiceTypes.srv*". En ellas se definen las estructuras de datos tanto para las peticiones como para las respuestas de los servicios en ROS.

A continuación, se van a desarrollar con más profundidad algunos de los conceptos numerados anteriormente.

3.2.1.1 Packages

Normalmente cuando se habla de paquetes, se refiere a una estructura típica de ficheros y carpetas (Figura 34, Figura 35). La estructura tiene la siguiente apariencia:

- *Bin/:* Esta es la carpeta donde los programas compilados y enlazados son almacenados después de haber sido compilados.

Figura 34 Estructura de un paquete

- *Include/package_name/:* Este directorio incluye los encabezados "headers" de las librerías que se necesitarían. No hay que olvidar exportar el "manifest" ya que están destinados a ser proporcionados para otros paquetes.
- Msg/: Si se van a desarrollar mensajes no estándar, hay que definirlos aquí.
- *Scripts/:* Estos son los "scripts" ejecutables que pueden ser de Python, Bash o cualquier otro tipo.
- *Src/:* Aquí es donde están presentes los archivos "source" de los programas.
- *Srv/:* Donde se representan los tipos de servicios
- *CmakeLists.txt:* Este es el fichero de compilación del "CMake"
- *Manifest.xml:* Este es el archivo del "manifest" del paquete.

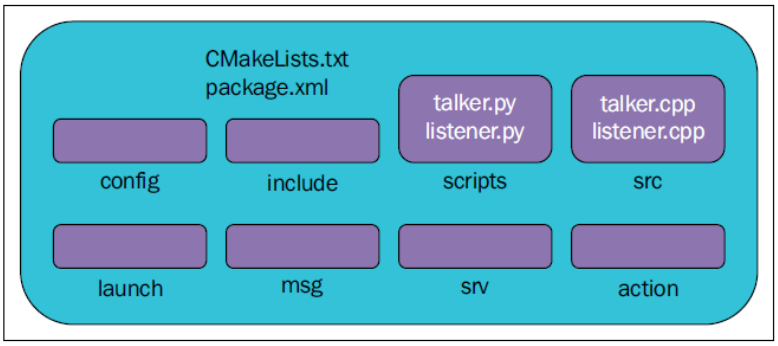


Figura 35 Estructura típica de un paquete en ROS

Para crear, modificar o trabajar con paquetes, ROS nos ofrece varias herramientas de asistencia:

- *rospack:* Este comando se usa para obtener información o encontrar paquetes en el sistema.
- *roscreate-pkg:* Se utiliza para crear un paquete nuevo, de cero.
- *rosmake:* Cuando se quiera compilar un paquete, se ejecuta este comando.
- *rosdep:* Con esto se instala las dependencias del sistema de un paquete.
- *rxdeps:* Muestra un gráfico con las dependencias del paquete.

Para moverse a través de los diferentes paquetes y sus carpetas y ficheros, ROS proporciona un paquete muy útil llamado *"rosbash",* que proporciona varios

comandos muy similares a los de Linux. A continuación, se muestran algunos ejemplos:

- *roscd:* Este comando nos ayuda a cambiar de directorio; es similar al comando "cd" de Linux.
- *rosed:* Se utiliza para editar un fichero.
- *roscp:* Este comando se utilizar para copiar un fichero de algún paquete.
- *rosd:* lista los directorios de un paquete.
- *rosls:* lista los ficheros de un paquete; es similar al comando "ls" en Linux.
- *rosrun*: este comando es utilizado para arrancar un ejecutable dentro de un paquete.

```
<?xml version="1.0"?>
<package>
  <name>hello_world</name>
  <version>0.0.0</version>
  <description>The hello_world package</description>

  <maintainer email="qboticslabs@gmail.com">Lentin Joseph</maintainer>
  <license>BSD</license>
  <url type="website">http://wiki.ros.org/hello_world</url>
 <author email="qboticslabs@gmail.com">Lentin Joseph</author>

  <buildtool_depend>catkin</buildtool_depend>
  <build_depend>roscpp</build_depend>
  <build_depend>rospy</build_depend>
  <build_depend>std_msgs</build_depend>

  <run_depend>roscpp</run_depend>
  <run_depend>rospy</run_depend>
  <run_depend>std_msgs</run_depend>

  <export>
  </export>
</package>
```

Figura 36 Estructura de un package.xml

Finalmente, el fichero "manifest.xml", debe estar en cada paquete y es utilizado para especificar información sobre este. Si se encuentra este fichero en una carpeta, probablemente esta carpeta sea un paquete.

El contenido del fichero "manifest.xml", incluye el nombre del paquete, las dependencias de este, etc. Todo esto es para hacer más fácil la instalación y la distribución del paquete.

Dos etiquetas típicas que se usan en este tipo de archivo son "<*depend*>" y "<*export*>". La etiqueta "<*depend*>" muestra que paquetes deben estar instalados antes de la instalación del paquete actual. Esto es porque el nuevo paquete usa algunas funcionalidades de otro u otros paquetes. La etiqueta "<*export*>" dice al sistema que "flags" necesita para ser usadas para compilar el paquete.

3.2.1.2 Stack

Los paquetes en ROS son organizados en pilas o "*Stacks" (*Figura 36*).* Mientras que el objetivo de los paquetes es crear colecciones mínimas de código para su

sencilla distribución y reutilización, el objetivo de las pilas es simplificar el proceso de compartir código.

Una pila necesita una estructura básica de ficheros y carpetas. Se puede crear manualmente, pero ROS proporciona el comando *"roscreate-stack"* para este proceso.

Las siguientes 3 fichero son necesarias para una pila: *"CmakeList.txt", "Makefile" y "stack.xml".* Si se ve el fichero "stack.xml" en una carpeta, se puede estar seguro de que se trata de una pila.

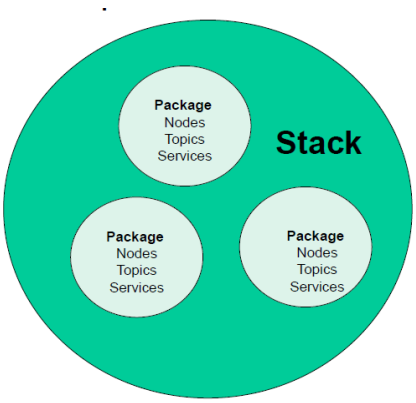


Figura 37 Estructura de un Stack

3.2.1.3 Messages

ROS utiliza un lenguaje de descripción de mensajes simplificado para describir los valores de datos que los nodos de ROS publican. Con esta descripción, ROS puede generar el código fuente adecuado para este tipo de mensajes en varios lenguajes de programación.

ROS tiene numerosos tipos de mensajes predefinidos, pero si es necesario, se pueden crear nuevos tipos que estarán en la carpeta /msg del paquete correspondiente. Dentro de esta carpeta, varios archivos con la extensión *".msg"* definen los mensajes.

Un mensaje debe tener dos partes principales: campos (fields) y constantes (constants). Los campos definen el tipo de dato que va a ser transmitido en el mensaje, por ejemplo, *"int32"*, "float*32*", and *"string"*, o nuevos tipos que hayan sido creados anteriormente, tales como, *"tipo1", "tipo2".* Las constantes definen el nombre de los campos. Un ejemplo de mensaje se muestra a continuación.

```
int32 id
float32 vel
string name
```

En ROS podemos encontrar multitud de tipos de mensajes predefinidos como los que se muestran en la Tabla 3:

Primitive type	Serialization	C++	Python
bool	Unsigned 8-bit int	uint8_t	bool
int8	Signed 8-bit int	int8_t	int
uint8	Unsigned 8-bit int	uint8_t	int
int16	Signed 16-bit int	int16_t	int
uint16	Unsigned 16-bit int	uint16_t	int
int32	Signed 32-bit int	int32_t	int
uint32	Unsigned 32-bit int	uint32_t	int
int64	Signed 64-bit int	int64_t	long
uint64	Unsigned 64-bit int	uint64_t	long
float32	32-bit IEEE float	float	float
float64	64-bit IEEE float	double	float
string	ASCII string (4-bit)	std::string	string
time	Secs/nsecs signed 32-bit ints	ros::Time	rospy.Time
duration	Secs/nsecs signed 32-bit ints	ros::Duration	rospy.Duration

Tabla 3 Resumen de los tipos de mensaje estándar en ROS

3.2.1.4 Services

Al igual que en los mensajes, ROS utiliza un lenguaje de descripción de servicios simplificado. Esto se basa directamente en el formato de los mensajes de ROS para habilitar la comunicación de petición y respuesta entre nodos. Las descripciones de estos servicios son almacenadas en archivos con extensión "*.srv*" en el subdirectorio "*/srv*" del paquete.

Para hacer una llamada a un servicio se necesita usar el nombre del paquete seguido del nombre del servicio, por ejemplo, "*sample_package1/srv/sample1.srv*".

Existen algunas herramientas que realizan algunas funciones con servicios. El comando "*rossrv*" muestra las descripciones de los servicios, los paquetes que contienen ficheros de tipo *.srv* y pueden encontrar fichero fuente de tipo "source" que usan ese tipo de servicio.

Si se quiere crear un servicio nuevo, ROS incorpora herramientas generadoras de servicios. Estas herramientas generan código a partir de las especificaciones iniciales del servicio. Solamente se necesita añadir la línea "*gensrv()" al fichero* "CmakeLists.txt".

3.2.1.5 "ROSRUN" Y "ROSLAUNCH"

Para terminar con este nivel, es importante explicar los diferentes modos que tiene ROS para ejecutar los nodos. Principalmente, nos centraremos en "rosrun" y "roslaunch".

- "Rosrun": comando que permite ejecutar un archivo ejecutable en un paquete arbitrario cualquiera que sea su ubicación, sin necesidad de indicarla, como por ejemplo:

```
rosrun <package> <executable>
```

  Incluso es posible pasar un parámetro utilizando la siguiente sintaxis:

```
rosrun package node _parameter:=value
```

- "Roslaunch": Es una herramienta para ejecutar de forma sencilla varios nodos a la vez, que pueden estar ubicados en diferentes directorios y que incluso puede ser ejecutado de forma local o remota (a través de SSH). Otra de las características importantes es que, al igual que en el "rosrun", se pueden establecer los valores de diferentes parámetros. Incluso, incluye opciones para decidir que acciones realizar cuando un proceso muere. Además, cabe la posibilidad de realizar un cambio de nombres de los topics, con el comando "remap", por ejemplo, si se está publicando información de un láser con el topic *"/scan1"* y el nodo que vamos a ejecutar se suscribe a *"/scan",* no es necesario cambiar el código de los programas si no simplemente escribir en el "launch" lo siguiente:

  <remap from="scan1" to scan"/>

  El launch se desarrolla en archivos de configuración de tipo XML con la extensión *".launch".* La mayoría de los nodos que vienen con ROS o que se descargan de la comunidad vienen con ficheros de este tipo, que al ejecutarlos lanzan todo lo necesario para que funcione correctamente.

  La forma de ejecutarlos es la siguiente:

```
roslaunch  package_name  file.launch
```

  En cuanto a su sintaxis, se muestra un ejemplo en la Figura 38:

```
<launch>
    <node
        pkg="camara"
        type="camara_siguelinea"
        name="camara_siguelinea"
        required="true"
        output="screen"
    >
    <remap from="camera/image" to="/mybot/camera1/image_raw"/>
    </node>

    <param name="Kd" value="0.005" />
    <param name="Kg" value="0.3" />

</launch>
```

Figura 38 Ejemplo "roslaunch"

3.2.2 Computation Graph Level

ROS crea una red donde todos los procesos están conectados. Cualquier nodo en el sistema puede acceder a esta red, interactuar con otros nodos, ver la información que están enviando y transmitir datos a la red. En la Figura 39 se puede ver un esquema del nivel de computación.

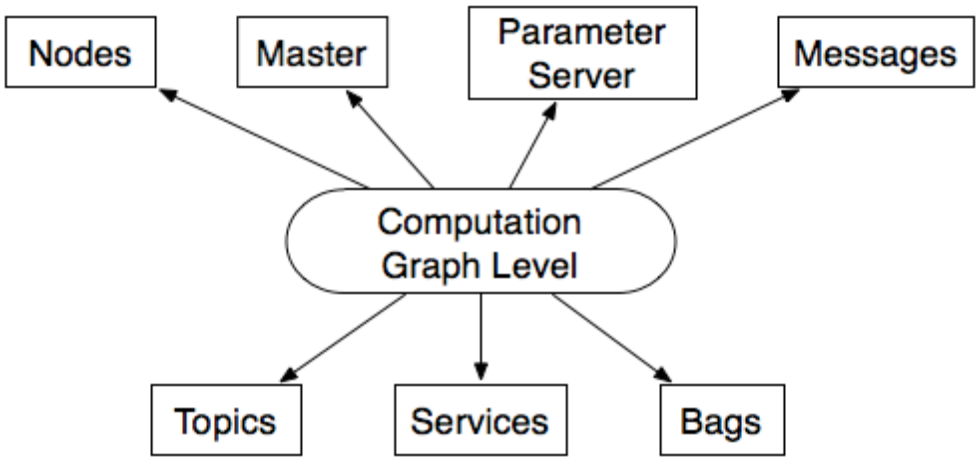


Figura 39 Esquema del nivel de computación

Los conceptos básicos de este nivel son:

- Nodos: los nodos son procesos donde se realiza el proceso computacional o cálculo. Si se quiere tener un proceso que interactúe con otros nodos, se necesita crear un nodo con este proceso para conectarlo a la red que crea ROS (ROS network). Normalmente, un sistema tiene muchos nodos para controlar diferentes funciones. Es preferible tener varios nodos que proporcionen una determinada funcionalidad, a tener un gran nodo que haga

todo el trabajo del sistema. Los nodos se escriben con una biblioteca cliente de ROS como, por ejemplo, *"roscpp" o "rospy".*

- Master: El master proporciona el registro de nombres y la búsqueda del resto de los nodos. Si no se tiene ejecutado el master en el sistema será imposible la comunicación entre nodos, servicios, mensajes, y demás. Aunque cabe la posibilidad de tenerlo en un ordenador mientras los nodos se ejecutan en otro, como ya se explicará más adelante.
- Servidor de Parámetros: ofrece la posibilidad de tener datos almacenados usando claves en una localización central. Con este parámetro, es posible configurar nodos mientras están siendo ejecutados o cambiar el funcionamiento de los nodos.
- Mensajes: los nodos se comunican unos con otros a través de estos mensajes. Un mensaje, contiene la información que se envía a otros nodos.
- Topics: cada mensaje debe tener un nombre para ser enrutado a través de la red de ROS (ROS network). Cuando un nodo está enviado datos, se dice que está publicando un "topic". Los nodos pueden recibir "topics" de otros nodos simplemente suscribiéndose al "topic". Un nodo puede suscribirse a un "topic" y no es necesario que el nodo que está publicando este "topic" deba existir. Esto permite desacoplar el consumo de recursos. Es importante que el nombre del "topic" sea único para evitar problemas y confusiones entre "topics" con el mismo nombre.
- Servicios: Cuando se están publicando "topics", se está enviando información de "muchos a muchos", pero cuando se necesita realizar una petición o una respuesta de un nodo no se puede hacer con topics. Los servicios nos ofrecen la posibilidad de interactuar con nodos. También, los servicios deben tener un nombre único. Cuando un nodo tiene un servicio, todos los nodos pueden comunicarse con él, gracias a las bibliotecas cliente de ROS.
- Bags: Son ficheros con formato que permiten almacenar mensajes de ROS. Los "bags" son un importante mecanismo para almacenar información como, por ejemplo, los datos de un sensor, que pueden ser difíciles de recolectar, pero es necesario para desarrollar y probar algoritmos.

A continuación, se desarrollarán con algo más de profundidad los nodos, los "topics" y el master debido a su gran importancia.

3.2.2.1 Nodes

Los nodos son ejecutables que se pueden comunicar con otros procesos usando topics, servicios o el servidor de parámetros. Usar nodos en ROS proporciona una tolerancia a fallos y separa el código y las funcionalidades haciendo el sistema mucho más simple.

Un nodo debe tener un nombre único en el sistema. Este nombre se usa para permitir que un nodo se comunique con otro nodo sin ambigüedades. Un nodo puede

haber sido escrito en un lenguaje diferente a otro y, sin embargo, gracias a los topics comunicarse sin ningún problema. Se suelen usar dos librerías principales: *"roscpp"* que es para C++ y "*rospy"* que es para "Python".

ROS tiene herramientas para manejar estos nodos y dar información sobre ellos como, por ejemplo, "*rosnode"*. Esta herramienta es una línea de comando para mostrar por pantalla la información sobre los nodos, es decir, un listado de los nodos que se están ejecutando. Los siguientes comandos son soportados:

- `rosnode info node`: Imprime información por pantalla sobre cierto nodo.
- `rosnode kill node`: Este comando mata a un nodo que este siendo ejecutado o enviando señal.
- `rosnode list`: lista los nodos activos.
- `rosnode machine hostname`: Muestra los nodos que se están ejecutando en una determinada máquina.
- `rosnode ping node`: Prueba la conectividad del nodo.
- `rosnode cleanup`: Limpia la información de registro de nodos inaccesibles.

Una de las características más potentes de los nodos en ROS es la posibilidad de cambiar parámetros mientras se está arrancando un nodo. Esta característica ofrece el poder de cambiar el nombre del nodo, el nombre o nombres de los "topics" y los nombres de los parámetros. Se usa esto para reconfigurar el nodo sin necesidad de recompilar todo el código de forma que podamos usar el nodo en diferentes escenarios.

A continuación, se muestra un ejemplo para cambiar el nombre de un "topic", en este caso pasa de llamarse "topic1" a llamarse "/level1/topic1":

```
rosrun book_tutorials tutorialX topic1:=/level1/topic1
```

Y de la misma forma se puede cambiar el valor de un parámetro:

```
rosrun book_tutorials tutorialX _param:=9.0
```

3.2.2.2 Topics

Los "*topics*" son canales de comunicación utilizados por los nodos para transmitir información. Puede ser transmitida sin una conexión directa entre nodos, lo que significa que la producción y el consumo de datos está desacoplados. Un "*topic*" puede tener varios suscriptores.

Cada topic está fuertemente ligado al tipo de mensaje de ROS utilizado para su publicación, y los nodos solamente pueden recibir "*topics*" que coincidan exactamente con este, es decir, un nodo puede suscribirse a un topic única y exclusivamente si utiliza el mismo tipo de mensaje.

Los *topics* en ROS pueden ser transmitidos a través de *TCP/IP* y *UDP*. El primer medio se conoce como *TCPROS* y usa la arquitectura *TCP/IP* persistente. Es el transporte por defecto usado en *ROS*.

El transporte basado en *UDP* es conocido como *UDPROS* aunque es un transporte de baja latencia y con pérdidas, por lo que es más adecuado para tareas como la teleoperación.

ROS proporciona una herramienta para trabajar con los topics que se llama "*rostopic"*. Es una herramienta de línea de comandos que proporciona información o bien sobre un topic o bien de los datos publicados directamente en la red *ROS*.

Esta herramienta cuenta con los siguientes parámetros:

- `rostopic bw /topic`: muestra el ancho de banda utilizado por dicho topic.
- `rostopic echo /topic`: muestra el contenido del topic
- `rostopic find message_type`: encuentra topics que utilicen el tipo de mensaje dado
- `rostopic hz /topic`: muestran la frecuencia de publicación de dicho topic
- `rostopic info /topic`: da información sobre un topic activo.
- `rostopic list`: lista por pantalla todos los topic activos en el sistema ROS
- `rostopic pub /topic type args`: Este comando se utiliza para publicar un valor a un topic con el tipo de mensaje que utilice
- `rostopic type /topic`: Muestra por pantalla el tipo de mensaje que utiliza el topic indicado.

3.2.2.3 Master

El ROS Master es muy similar a un servidor DNS (Sistema de Nombres de Dominio). Cuando un nodo arranca en el sistema ROS, este comienza a buscar al ROS Master y registra el nombre del nodo en él. De esta forma, el master tiene los detalles de todos los nodos que están corriendo en el sistema actualmente (Figura 40). Además, cuando algún detalle de los nodos cambia, se realiza una llamada a la función "*call-back*" y se actualiza con la última información.

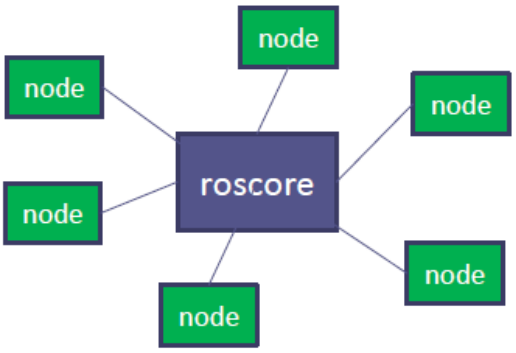


Figura 40 Diagrama conexión master con nodos

Cuando un nodo empieza a publicar un topic, el nodo se encarga de buscar al master y proporcionarle la información necesaria, tal como, el nombre y el tipo de mensaje que utiliza. Así, el máster será capaz de comprobar si hay o no algún otro nodo suscrito al mismo topic y que, por lo tanto, sea el mismo tipo de mensaje. Si varios nodos están suscritos al mismo tipo de mensaje, el master se encargará de interconectar a estos nodos y compartir la información del que publica a los suscriptores (Figura 41). Una vez obtenidos los detalles de cada nodo, el máster interconecta a estos mediante el protocolo TCPROS que se basa en los sockets de la arquitectura TCP/IP. Luego de conectarlos, el master no tiene ninguna función de control sobre ellos, será el usuario el que tenga la posibilidad parar cualesquiera de ellos, ya sea el publicador o el suscriptor. Este mismo método es utilizado con los "Servicios de ROS".

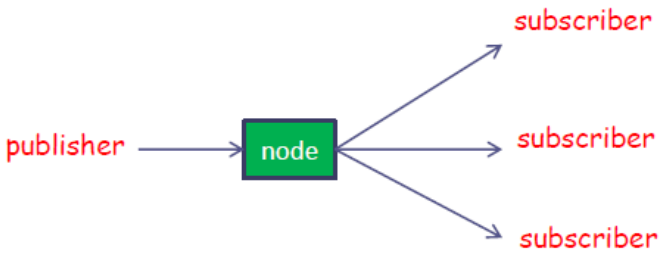


Figura 41 Diagrama de conexión entre publicador y suscriptores

Otra de las características que destacan, y que es de gran importancia, es el entorno del "ROS_MASTER_URI". Esta variable de entorno contiene la IP y el puerto del ROS Master, de modo que, utilizando esta variable, los nodos de ROS puedan localizar al máster independientemente de que se encuentre ejecutándose en otra máquina. Si esta variable no está correctamente definida, la comunicación entre nodos no tendrá lugar. Cuando se utiliza ROS solamente en una máquina se utiliza como IP el "localhost" o bien el nombre de éste. Pero en una red distribuida, en la que se están ejecutando varios nodos en diferentes máquinas, se debe definir correctamente la variable "ROS_MASTER_URI", ya que solo de esta manera, los nodos remotos serán capaces de encontrar el máster y, así, poder comunicarse.

En el diagrama que se muestra en la Figura 42 se puede ver como el ROS Master interactúa con el nodo publicador y suscriptor, en este caso el primero publicando un string “hello world” y el segundo suscribiéndose a este.

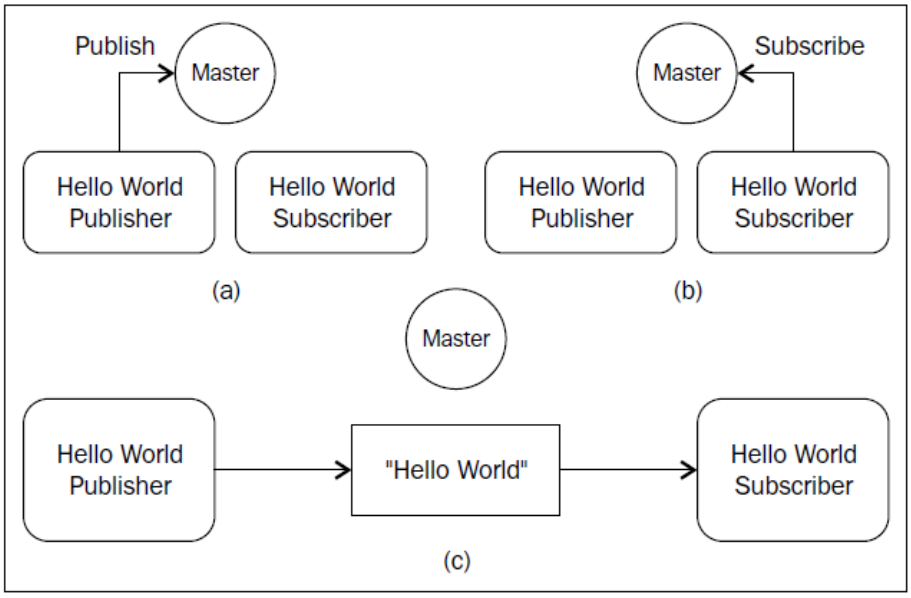


Figura 42 Comunicación entre nodos a través del ROS Master

Finalmente, ROS dispone de otra herramienta que permite la visualización del gráfico de comunicación y se llama “*rqt_graph*” (Figura 43). En este caso los óvalos representan los nodos y las flechas las relaciones de comunicación publicación-suscripción. Las etiquetas de estas son los topics.

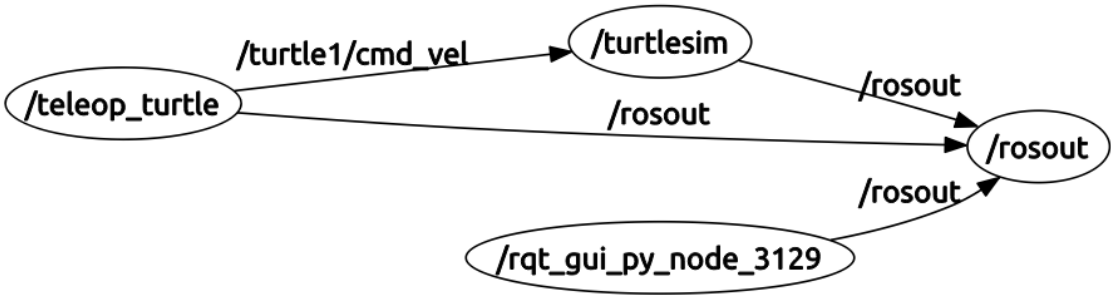


Figura 43 rqt_graph publicación-suscripción

3.2.3 Community Level

Por último, aunque no menos importante, se hablará brevemente sobre el nivel de la comunidad como concepto de ROS. Se basa en el hecho de compartir e intercambiar software, conocimientos y recursos entre los miembros de la comunidad. Entre muchos otros se incluyen recursos como:

- **Distributions:** Son colecciones de pilas versionadas que se pueden instalar. Las distribuciones de ROS juegan un papel muy similar a las distribuciones de

Linux. Facilitan la instalación de una colección de software y también mantiene versiones estables de este.

- **Repositories:** ROS se basa en una red federada de repositorios de código, donde diferentes instituciones pueden desarrollar y liberar sus propios componentes de software de robots.
- **The ROS Wiki**: Es el principal foro para documentar información sobre ROS. Cualquiera puede crearse una cuenta y contribuir con su propia documentación, proporcionar correcciones o actualizaciones, incluso escribir tutoriales y mucho más.
- **Mailing lists**: Las listas de correo de usuario de ROS son el principal canal de comunicación sobre las nuevas actualizaciones de ROS, así como un foro para hacer preguntas sobre el software ROS.

3.3 ROBOT CONTROL

Se trata del nodo principal, encargado del control del robot, no solo de su movimiento sino también de todos los sensores de los que dispone. Es el nodo más extenso e importante del proyecto, ya que sin el no funciona ninguno de los nodos restantes. Todas las referencias a códigos y archivos de los que se hablará a continuación se pueden ver en los Anexos, más concretamente en el Anexo D.

El script principal es "*edubot_control_main.py*" en el cual se lanzan 2 hilos:

1) *Thread_update*: es el encargado del control, envía referencias de velocidad a los motores, actualiza el estado del robot y publica el topic *"/robot_control/state*". El bucle se ejecuta según la constante de tiempo "*stepDelay=0.3*".
2) *Thread_sonar*: se encarga de la lectura de los ultrasonidos y de publicar esta información en el topic *"/robot_control/sonar_state*". El bucle se ejecuta según la constante de tiempo "*stepDelay_sonar=0.3*".

Además, contiene las siguientes funciones:

- Simulation(): realiza llamadas a todas las funciones necesarias para realizar una simulación del robot, por ejemplo, solicita el estado del robot, sus sistemas de referencia, odometría, alarmas, etc.

- Sonar_simulation(): en este caso, llama a las funciones necesarias para saber el estado de los sonar y poder simularlos.

- Callback_command(msg): muy importante, está a la espera de un comando para resetear las placas controladoras de motores. Cuando recibe el mensaje "RESET" llama a las funciones "*PBR_R.ResetEpo()" y "PBR_L.ResetEpo()"* que se encargan de resetear las placas controladoras, por ejemplo, en el caso de que haya saltado alguna alarma para que puedan volver a funcionar.

- Callback_vel_command(msg): El nodo está suscrito al topic de velocidad "*/base_control/vel_command"* de modo que cuando hay algún mensaje nuevo de este tipo esta función es llamada. Se encargar de recoger la referencia de velocidad que debe tener cada rueda y establecerlas en las correspondientes placas controladoras de los motores, mediante las funciones "*PBR_L-SetSpeedMotor()" y* "*PBR_L-SetSpeedMotor()".*

- Callback_test_vel_command_pulses(msg): Se suscribe al topic *'test/vel_command_pulses'* para hacer pruebas de envío de pulsos/seg y poder graficar los resultados en *"rqt_plot".*

- Pub_odometry(pose): publica la odometría y todo lo que ella conlleva: "frame_id", "child_frame_id", "position", "orientation",

- Pub_tf(pose): publica el "tf", un paquete que permite al usuario realizar un seguimiento de múltiples sistemas de coordenadas a lo largo del tiempo. Mantiene la relación entre los diferentes sistemas en una estructura de árbol almacenada en el tiempo. Se desarrollará más adelante.

- Read_control(): llama a todas las funciones necesarias para conocer el estado general del robot, es decir, adquiere datos de la velocidad de los encoders, de la odometría, el estado de los ejes, el estado de las alarmas y el estado de la batería

- Read_alarms(): Se trata de una función cuyo objetivo es publicar un vector de alarmas en el que quedan recogidos todos los posibles fallos o avisos de las placas controladoras que manejan los motores del robot, para que pueda ser conocido por la función "*Read Control".* Para ello, esta función se encarga de llamar a "*PBR_R.GetAlarms()" y* "*PBR_L.GetAlarms()"* las cuales devuelven un valor decimal, que tiene una correspondencia con cada alarma.

  El vector que devuelve tiene la estructura que se muestra en la Tabla 4:

Pos	Alarma	Descripción
[1]	com_R	Fallo de comunicación del motor derecho
[2]	com_L	Fallo de comunicación del motor derecho
[3]	driver_R	Fallo en el driver de la controladora motor derecho
[4]	driver_L	Fallo en el driver de la controladora motor izquierdo
[5]	emergency_R	Se ha pulsado la seta de emergencia
[6]	emergency_L	Se ha pulsado la seta de emergencia
[7]	disabled_emg_R	Se ha desactivado la seta de emergencia tras haber sido pulsada previamente
[8]	disabled_emg_L	Se ha desactivado la seta de emergencia tras haber sido pulsada previamente
[9]	encoder_R	Fallo del encoder del motor derecho
[10]	encoder_L	Fallo del encoder del motor izquierdo

Tabla 4 Descripción del vector de alarmas

Sin embargo, para reconocer estos errores o avisos, se han tenido que reprogramar las placas controladoras "PicoBorgReverse" como se puede ver en el ANEXO C. Lo que se hace es que cuando se produzca uno de estos avisos, las funciones "*PBR_R.GetAlarms()" y "PBR_L.GetAlarms()"* devuelvan un número que será interpretado por la función "*ReadAlarms()". En la* Tabla 5 se muestran los valores que han sido establecidos:

Valor binario	Valor decimal	Significado
100000000	256	disabled_emg_R, disabled_emg_L
1000000000	512	driver_R, driver_L
1100000000	768	emergency_R, emergency_L
10000000000	1024	encoder_R, encoder_L
10100000000	1280	Cuando desactivamos la seta tras haber estado activa un largo periodo
11100000000	1792	Cuando la seta lleva un largo rato pulsada

Tabla 5 Valores de las alarmas

- Read_digout_state(): Se trata de un vector parecido al de alarmas, pero que complementa a este. Ya que el anterior únicamente mostraba avisos o errores de las placas controladoras de los motores, en este caso, el *"digout_state"* se utiliza para rellenar un vector con información relevante del estado actual del robot en cuanto a la predisposición de este a ser utilizado, es decir, no tiene nada que ver con la función *"ReadControl()"* explicada anteriormente, si no que se centra en alarmas y avisos generales del robot para saber de una manera rápida si todo está correcto y puede ser utilizado.

Se trata de un vector de enteros, aunque 3 de los 4 campos se utilizarán como booleanos, es decir, que mostrarán "0" y "1" como si de "flags" se tratase. El único que no se trata como tal es el tercero, que indicará el nivel de batería. Los campos de los que se componen se muestran en la Tabla 6.

Pos	Alarma	Descripción
[1]	ready	Indicará con un "1" si todo está listo para que el robot pueda ejecutar cualquier tarea.
[2]	alarms	Mostrará un "1" si alguna de las 10 alarmas de los motores vistas anteriormente está activa. En caso de que esto ocurra se encargará de enviar la orden de paro a los motores y de que se dejen de publicar referencias a los encoders.
[3]	battery	Indicador de batería: 0. Agotada 1. Nivel crítico 2. Nivel medio 3. Nivel óptimo
[4]	busy	Indicará con un "1" si el robot está ocupado realizando alguna tarea.

Tabla 6 Estructura del vector "digout_state"

- Read_sonar(): Función que se encarga de la lectura de los ocho ultrasonidos, gracias a las placas controladoras "*Ultraborg*" que manejan cuatro cada una. Se comunican con ellas mediante I2C al igual que ocurría con las controladoras de los motores "*PicoBorgReverse*" .
  Esta función devuelve un vector con los valores de los ocho ultrasonidos.

- Init(): Su función es inicializar todos los parámetros y vectores y, para ello, llama a "*Control_state*" e inicializa vectores como "alarms", "digout_state", "axes". También establece las referencias de velocidad de los motores a cero.

- Update(): Una de las funciones más importantes, no solo se encarga de actualizar cada valor para conocer el estado actual del robot si no que si detecta alguna anomalía se encarga de parar los motores.
  Para mantener actualizado el estado del robot llama a las funciones "read_control", "digout_state", "currente_state", "diff_ctrl", etc. Y publica la odometría y el "tf".

- Sonar_update(): Como su propio nombre indica actualiza el estado de los sonar.

Para terminar, cabe destacar a modo de resumen que el módulo robot_control utiliza otros ficheros para la realización de sus funciones, como son:

- Las funciones que se comunican con las placas de lectura de ultrasonidos usan “UtraBorg.py”.

- Las funciones que se comunican con las placas de control de motores usan “PicoBorgReverse.py”. Ya se ha dicho en varias ocasiones que no es la librería original de las placas Picoborg, ya que se han modificado y/o añadido algunas funciones, como también se ha modificado el programa del PIC de las placas. Todo ello se puede ver en el Anexo D.

- Las funciones que dependen de la cinemática del robot (diferencial) están en “diffdrive_controller.py”.

- Y, por último, las funciones que necesitan GPIO en “gpio_interface.py”

3.4 CAMARA

En este apartado es importante distinguir entre los dos nodos principales que se utilizan, ya que uno es el que publica y el otro el que se suscribe:

- Picamera: Nodo que publica la imagen de la cámara, con ciertos parámetros.
- Camera: Nodo que se suscribe a la imagen, con diferentes funcionalidades.

Esto es destacable porque es donde reside la verdadera potencia de esta descentralización, pudiendo ejecutar de esta forma el nodo suscriptor en cualquier máquina remota.

3.4.1 Picamera_node

Todo lo que se explicará a continuación puede verse en el código “picamera_main” que se encuentra en el Anexo D.

El nodo cuenta con las siguientes funciones:

- Callback_camera_start(): esta función es llamada cuando hay una petición para comenzar la captura y, por tanto, se encarga de establecer la variable “b_capture” en “true”.

- Callback_camera_stop(): al igual que la anterior, esta función es llamada cuando hay una petición de finalizar y, por tanto, se encarga de establecer la variable “b_capture” en “false”, terminando así la captura de imagen.

- Capture(): comprueba la variable “b_capture” y en función de su valor (“true” o “false”) comienza a capturar la imagen de la cámara o deja de hacerlo. Para ello, se encarga de tomar el frame de la cámara y guardarlo en una variable que posteriormente será utilizado por la función “main”.

- Main(): Cuenta con un hilo principal que es el encargado de llamar a "capture()". Una vez obtenido el frame de la cámara, "main" se encarga de establecer todos los parámetros de la imagen: brillo, contraste, resolución, estabilización, color, rotación, etc. Finalmente, publica con el tipo de mensaje adecuado en un topic el resultado, que lleva el siguiente nombre *"/picamera_node/image_raw/compressed"*

3.4.2 Camera_node

Este nodo contiene numerosos scripts, sin embargo, se centrará la atención en 2 de ellos:

- "Camera_subscriber": Como su propio nombre indica, es el encargado de suscribirse al topic donde está publicada la imagen y mostrarla por pantalla en una ventana emergente al usuario. Este nodo es muy útil para cuando se quiera hacer un control remoto del robot, pudiendo dirigirle a través de la imagen que muestra en tiempo real. Es importante para aplicaciones como el rastreo de zonas peligrosas o lugares a los que no pueda acceder el ser humano.

- "Camara_siguelinea" junto con "seguir_linea": Estos dos scripts funcionan en conjunto para lograr el objetivo final de que el robot sea capaz de seguir una línea. Para ello se lleva a cabo un algoritmo de identificación y de control que se explica a continuación.

  Lo primero que se hace es tomar el frame de la imagen suscribiéndose al topic correspondiente que lo contiene (Figura 45). Una vez se dispone de la imagen se realiza un proceso de binarización pura, pasando así a una imagen en blanco y negro con la que es mucho más fácil detectar la línea (Figura 46).

  A continuación, se trazan un conjunto de líneas horizontales separadas mínimamente y se busca el punto de corte de cada una de ellas con la línea negra objetivo a seguir. Al unir esos puntos de intersección, se obtiene una pequeña trayectoria a seguir (Figura 44). Además, se publica un topic con el nombre *"/distancialinea"* que será utilizado por el otro script "seguir_linea". Este topic tiene 2 campos: el primero de ellos es la distancia que hay del centro de la imagen a la línea, que servirá para indicar el desvío que lleva el robot de la línea respecto a su eje longitudinal; el segundo de ellos es el ángulo, valor que indica la desviación en radianes de la línea respecto a una vertical. De todo esto se encarga el fichero "camara_siguelinea".

Figura 45 Imagen captada

Figura 46 Imagen binarizada

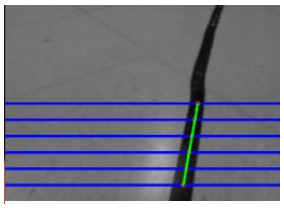


Figura 44 Calculo de trayectoria

Cabe destacar que estas líneas horizontales azules que se muestran en la figura 40, tienen un parámetro que se puede modificar para ajustar a que altura de la imagen queremos que estén, de forma que si se colocan en la parte más inferior tendríamos una trayectoria inmediata mientras que si se colocan en la parte superior se estaría hablando de una trayectoria de carácter anticipativo, algo que habrá que tener en cuenta a la hora de mandar referencias a los motores.

De las órdenes a los motores se encarga el fichero "seguir_linea" el cual realiza un control de tipo P (proporcional). Como ya se ha dicho anteriormente, este módulo se suscribe al topic *"/distancialinea"* para recibir los valores de las variables "distancia" y "ángulo" que serán utilizados para realizar el control proporcional junto con las constates correspondientes. A la hora de lanzar este módulo, hay tres parámetros importantes a tener en cuenta que se pueden establecer en el "roslaunch" (más tarde se hablará de esto, pero se trata de un lanzador). Los parámetros son los siguientes:

- Velocidad_lineal: Como su propio nombre indica, establecerá el valor de la velocidad con la que el robot recorrerá la trayectoria pertinente. Cabe destacar que una velocidad elevada puede hacer perder la referencia de la línea, así que es aconsejable establecer una velocidad reducida.
- Kd: es la constante proporcional de distancia. Este será el valor por el que se multiplique la variable distancia y cuyo resultado influirá en el valor del giro.
- Kg: del mismo modo, esta es la constante de giro, cuyo valor será multiplicado por la variable ángulo y que influirá en un incremento o decremento del valor del giro.

Para que quede más claro, se muestra a continuación la asignación de velocidad y giro:

$$msg.linear.x = Velocidad\ lineal$$

$$msg.angular.z = Kd * distancia + Kg * ángulo$$

Hay que tener en cuenta que, obviamente el valor de la velocidad lineal del robot influirá en los parámetros "Kd" y "Kg", es decir, una vez que

se haya ajustado estas constantes proporcionales correctamente para una velocidad de 0.1 m/s, por ejemplo, no serán igual de válidos si se decide cambiar la velocidad. En el capítulo 4 se mostrarán diferentes experimentos y el ajuste de parámetros del robot siguiendo líneas.

Por último, pero no menos importante, se ha implementado en este módulo un método para cuando el robot pierda la línea de la imagen. Se basa en que, en lugar de dejar de mandar referencias de velocidad a los motores en caso de no encontrar línea, el nodo sigue mandando durante un pequeño periodo de tiempo las mismas referencias de velocidad que estaba mandando instantes antes de perder la línea, de modo que sea capaz de intentar encontrarla de nuevo. Esto es así debido a que la línea suele ser perdida en las curvas y, por tanto, si se sigue girando como se estaba haciendo justo antes de perderla, es probable que la cámara vuelva a captar la línea.

3.5 JOYSTICK

Se trata de una pequeña aplicación web que implementa un "Joystick" para el control remoto del robot, como puede verse en la Figura 47. No se entrará en detalles de esta aplicación, ya que no se tiene conocimientos suficientes acerca de la programación web.

Figura 47 vista preliminar de la aplicación web Joystick

De lo que podemos hablar es de que la aplicación se conecta a una dirección IP que deberá ser asignada por el usuario. Se debe indicar la dirección IP del "rosmaster" para la correcta comunicación con el robot. Además, se especificará el puerto de conexión que será el 9090.

```
"ros.connect('ws://192.168.192.66:9090')"
```

Para que todo funcione correctamente, debe estar implementado el nodo de ros "WebSocket server", que será el que interprete el comando "ros.connect" y realice la conexión con el máster.

Además, son requeridos una serie de permisos web adicionales que son:

- ✓ `sudo chgrp -R www-data estado_mobile/`
- ✓ `sudo chmod -R 2755  estado_mobile/`
- ✓ `sudo chown porrob /var/www/padel/`
- ✓ `sudo chmod -R 0775 /var/www/padel/`

Finalmente, para que el robot interprete correctamente las órdenes, esta aplicación web deberá publicar un topic con el contenido de sus comandos. Para el caso de comandos de movimiento, los datos serán publicados en un topic con el nombre de *"/base_control/vel_command"* y con el tipo correcto de mensaje de velocidad, que es *"std_msgs/Twist"*. Sin embargo, esta aplicación también cuenta con un botón de reset que permite enviar este comando a las controladoras de los motores en caso de que haya saltado alguna alarma. Para ello, el tipo de mensaje que utiliza es *"std_msgs/String"* y el topic en el que publica lleva por nombre *"/base_control/command"*.

3.6 ODOMETRÍA

La odometría es el estudio de la estimación de la posición de vehículos con ruedas durante la navegación. Para realizar esta estimación se usa información sobre la rotación de las ruedas para estimar cambios en la posición a lo largo del tiempo.

En robótica, los robots móviles usan la odometría para estimar su posición relativa a su localización inicial. La odometría que proporciona una buena precisión a corto plazo es barata de implantar, sin embargo, la idea fundamental es la integración a largo plazo, lo cual puede llevar a una acumulación de errores inevitable. Estos errores en la estimación de la posición van aumentando proporcionalmente con la distancia recorrida por el robot. No obstante, la odometría es una parte importante del sistema de navegación de un robot y debe usarse.

La odometría se basa en ecuaciones simples, que utilizan datos de los encoders situados en las ruedas del robot. Además, también está basada en la suposición de que las revoluciones de las ruedas pueden ser traducidas en un

desplazamiento lineal relativo al suelo. Esto no tiene una validez absoluta debido a que se pueden producir diferentes errores, como, por ejemplo [5]:

- Que alguna de las ruedas patine en algún momento.
- Los diámetros de las ruedas no son exactamente iguales.
- Un mal alineamiento de las ruedas.
- Que la tasa de muestreo del encoder sea discreta, o algún fallo en el mismo.
- Suelos desnivelados.
- Desplazamiento sobre objetos inesperados que se encuentren en el suelo.

Para desarrollar la odometría en el robot "Edubot" se tendrá en cuenta la información que proporcionan los codificadores incrementales que llevan acoplados ambos motores. Estos, a través de las controladoras "PicoBorg Reverese" muestran el número de pulsos acumulados. Por tanto, para estimar la posición del robot habrá que basarse en el modelo cinemático diferencial que ya se pudo ver en el apartado 2.2.2.

Para este modelo la posición puede ser estimada mediante la información de los codificadores y las ecuaciones geométricas que surgen de la disposición de los componentes del sistema de propulsión, que se expresaron en la Figura 9.

La posición del robot siempre estará referida al punto medio del eje entre las ruedas motrices, como se expresa en la Figura 48.

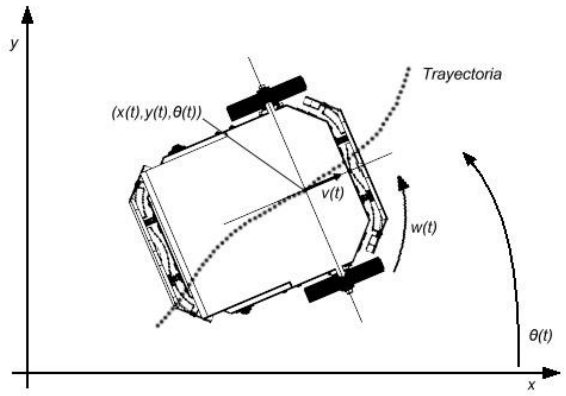


Figura 48 Localización del robot en el plano cartesiano

Los parámetros para determinar la posición en este modelo son la distancia entre ejes, el diámetro de las ruedas y el ángulo de giro de cada uno de los ejes. Partiendo, entonces, de cuantos pulsos han contabilizado los codificadores incrementales y, por tanto, cuantos grados a girado cada eje, se pude deducir la distancia que ha recorrido cada rueda [6]. Matemáticamente, queda expresado como:

$$\dot{x} = v(t)\cos\big(\theta(t)\big)$$

$$\dot{y} = v(t)\,\mathrm{sen}\big(\theta(t)\big) \qquad (\mathbf{6})$$

$$\dot{\theta} = w(t)$$

De modo que la posición y orientación del robot se obtendrán integrando las velocidades en un periodo de tiempo:

$$x(t) = x(t_o) + \int_{\Delta t} v(t)cos\big(\theta(t)\big)dt$$

$$y(t) = y(t_o) + \int_{\Delta t} v(t)sen\big(\theta(t)\big)dt \qquad (\mathbf{7})$$

$$\theta(t) = \theta(t_o) + \int_{\Delta t} w(t)dt$$

No obstante, si el periodo de observación "Δt" tiende a cero, podemos considerar desplazamientos diferenciales ($\Delta x\ \Delta y\ \Delta\theta$) para que desaparezcan las integrales. Dicho de otro modo, se puede considerar que la velocidad angular de cada una de las ruedas es constante si la frecuencia de muestreo se mantiene constante y elevada sobre la odometría del robot, y estimar la posición y orientación mediante las siguientes ecuaciones en diferencias:

$$x_k = x_{k-1} + \Delta x_k$$

$$y_k = y_{k-1} + \Delta y_k \qquad (\mathbf{8})$$

$$\theta_k = \theta_{k-1} + \Delta\theta_k$$

Para calcular la distancia del robot, sabiendo la distancia recorrida por cada una de las ruedas, se pueden deducir las siguientes relaciones de la Figura 49

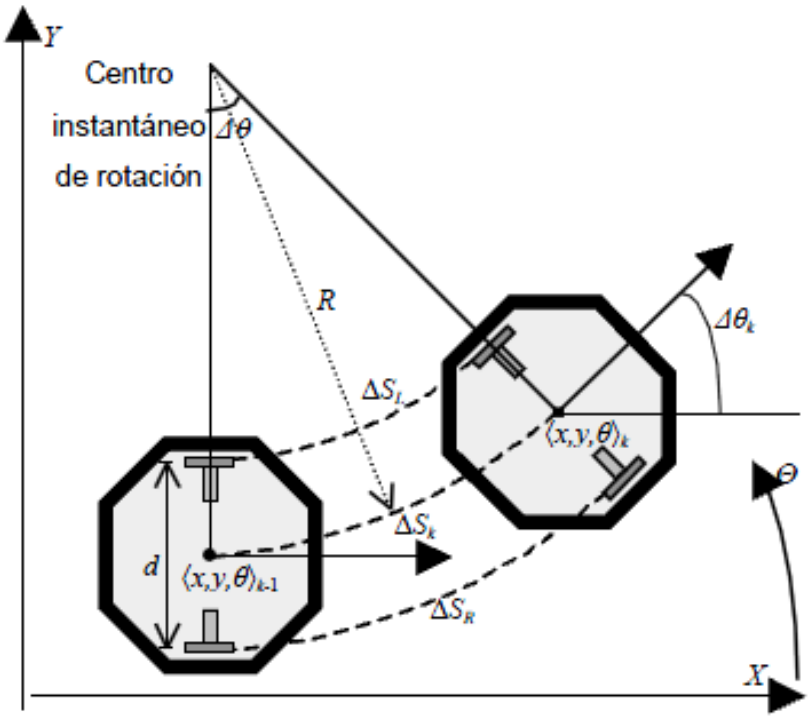


Figura 49 Desplazamiento relativo entre instantes de muestreo

$$\Delta S_k = (\Delta S_R + \Delta S_L)/2$$
$$\Delta\theta = (\Delta S_R - \Delta S_L)/d \quad \textbf{(9)}$$

Entonces, la posición del robot utilizando las ecuaciones en diferencias obtenidas y la distancia recorrida por cada rueda vendrá determinada por las siguientes ecuaciones [8]:

$$x_k = x_{k-1} + \Delta S_k \cos\left(\theta_{k-1} + \frac{\Delta\theta_k}{2}\right)$$
$$y_k = y_{k-1} + \Delta S_k \sin\left(\theta_{k-1} + \frac{\Delta\theta_k}{2}\right) \quad (\mathbf{10})$$
$$\theta_k = \theta_{k-1} + \Delta\theta_k$$

Para crear la odometría del robot en ROS, se desarrollará un pequeño código en un fichero ".cpp" cuyas partes más importantes se explican a continuación.

Lo primero de todo será definir la variable de transformación "tf" y rellenarla con los valores de "frame_id" y "child_frame_id" para saber cómo deben moverse los sistemas de referencia. En este caso, la base será "base_frame" que se moverá respecto del sistema fijo "odom".

```
geometry_msgs::TransformStamped odom_trans;
      odom_trans.header.frame_id = "odom";
      odom_trans.child_frame_id = "base_frame";
```

A continución, se genera la posicion y orientación del robot. Con la velocidad lineal y la angular, se puede calcular de forma teórica la posición del robot pasado un tiempo:

```
double dt = (current_time - last_time).toSec();
double delta_x = (vx * cos(th) - vy * sin(th)) * dt;
double delta_y = (vx * sin(th) + vy * cos(th)) * dt;
double delta_th = vth * dt;
x += delta_x;
y += delta_y;
th += delta_th;
geometry_msgs::Quaternion odom_quat;
odom_quat =
tf::createQuaternionMsgFromRollPitchYaw(0,0,th);
```

En las transformaciones, solamente se rellenarán los campos de posición en “x” y rotación, debido a que el robot solo es capaz de moverse hacia delante, hacia atrás y girar.

```
odom_trans.header.stamp = current_time;
odom_trans.transform.translation.x = x;
odom_trans.transform.translation.y = 0.0;
odom_trans.transform.translation.z = 0.0;
odom_trans.transform.rotation =
tf::createQuaternionMsgFromYaw(th);
```

Con la odometría se hará lo mismo. Se rellenarán los campos “frame_id” y “child_frame_id” con “odom” y “base_frame”.

Como la odometría tiene dos estructuras, posición y velocidad, se rellenarán los campos “x”, “y” y “orientation” de la estructura “pose” y los campos “linear.x” y “angular.z” en la estructura “twist”, como se muestra:

```
// position
odom.pose.pose.position.x = x;
odom.pose.pose.position.y = y;
odom.pose.pose.orientation = odom_quat;
// velocity
odom.twist.twist.linear.x = vx;
odom.twist.twist.angular.z = vth;
```

Una vez que todos los campos han sido completados correctamente, se pasará a publicar la información:

```
// publishing the odometry and the new tf
broadcaster.sendTransform(odom_trans);
odom_pub.publish(odom);
```

Es importante escribir la siguiente línea en el archivo “CmakeLists.txt” antes de compilar todo lo anterior.

```
rosbuild_add_executable(odometry src/odometry.cpp)
```

Se puede ver el código completo referente a la odometría del robot en Anexo D.

3.7 MODELADO 3D DEL ROBOT EN ROS

El modelado 3D del robot no solo será útil para posteriores posibles simulaciones antes de probar el real, sino que también será necesario para el siguiente capítulo en el que se va a utilizar el "Stack de Navegación".

La forma en que ROS utiliza el modelo 3D de un robot y sus partes para simularlos o simplemente para ayudar a los desarrolladores en el trabajo del día a día, es a través de los archivos "URDF" o "Xacro".

URDF ("*Unified Robot Description Format*") es un fichero de formato XML que describe un robot, sus partes, sus articulaciones, dimensiones, etc. Cada robot existente en ROS tiene un fichero URDF asociado a él.

Para empezar con la descripción del robot se debe crear una nueva carpeta en el espacio de trabajo que se denominará "*/edubot_description/URDF"* donde se localizará el fichero en nuestro caso con la extensión ".xacro" aunque también se puede hacer con la extensión ".urdf". Se comenzará el archivo describiendo la base del robot con las 2 ruedas, como se puede ver en la Figura 50. Se puede ver una descripción completa en Anexo D.

```
<?xml version="1.0"?>
   <robot name="edubot">
       <link name="base_link">
              <visual>
                     <geometry>
                            <box size="0.41 0.35 0.202"/>
                     </geometry>
                     <origin rpy="0 0 0" xyz="-0.11 0 0"/>
                     <material name="white">
                            <color rgba="1 1 1 1"/>
                     </material>
              </visual>
       </link>

       <link name="left_wheel">
              <visual>
                     <geometry>
                            <cylinder length="0.038" radius="0.085"/>
                     </geometry>
                     <origin rpy="0 1.5707 1.5707" xyz="0 0 0"/>
                     <material name="black">
                            <color rgba="0 0 0 1"/>
                     </material>
              </visual>

       </link>

       <link name="right_wheel">
              <visual>
                     <geometry>
                            <cylinder length="0.038" radius="0.085"/>
                     </geometry>
                     <origin rpy="0 1.5707 1.5707" xyz="0 0 0"/>
                     <material name="black"/>
                            <color rgba="0 0 0 1"/>
                     </material>
              </visual>
       </link>

       <joint type="continuous" name="left_wheel_hinge">
           <origin xyz="0.0 0.205 -0.091" rpy="0 0 0"/>
           <child link="left_wheel"/>
           <parent link="base_link"/>
           <axis xyz="0 1 0" rpy="0 0 0"/>
           <limit effort="10000" velocity="1000"/>
           <joint_properties damping="1.0" friction="1.0"/>
        </joint>

        <joint type="continuous" name="right_wheel_hinge">
           <origin xyz="0.0 -0.2025 -0.091" rpy="0 0 0"/>
           <child link="right_wheel"/>
           <parent link="base_link"/>
           <axis xyz="0 1 0" rpy="0 0 0"/>
           <limit effort="10000" velocity="1000"/>
           <joint_properties damping="1.0" friction="1.0"/>
        </joint>
         </robot>
```

Figura 50 Base link y ruedas URDF (1)

Como se puede ver en el código de la Figura 50 , hay dos campos principales que describen la geometría del robot: *"links"* y *"joints".*

El primer “link” lleva el nombre de “base_link”; este nombre debe ser único en el fichero. Dentro de este código se pueden definir todos los aspectos visuales que más tarde se apreciarán en las simulaciones tales como la geometría (cilindro, caja, esfera), el material (color y textura) y el origen.

En cuanto al código de los “joints”, al igual que los “links” deben llevar un nombre único. Además, se debe definir el tipo de unión (fija, revolución, continua, flotante o planar), el padre y el hijo (se refieren al “TF” o árbol de transformaciones). En este caso, “right wheel” es el hijo de “base_link”.

Para comprobar si hay errores en el código, se puede utilizar la herramienta “check_urdf” en cualquier terminal. También, si se quiere ver la jerarquía y relación entre los sistemas de referencia del URDF gráficamente se puede emplear la herramienta “urdf_to_graphiz” Figura 51. [4]

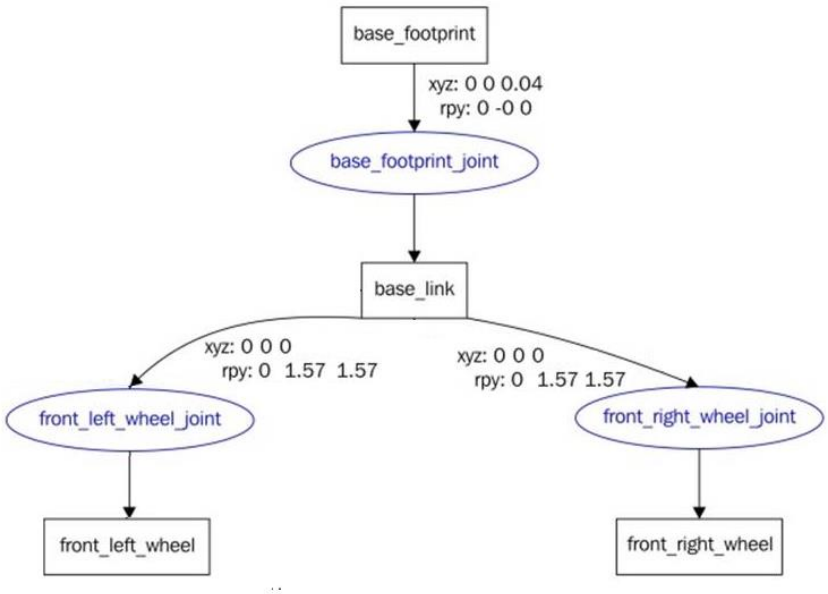


Figura 51 gráfico URDF

Una vez definido el robot y las ruedas en el archivo URDF se deben definir también los sensores que vamos a utilizar, en este caso la cámara y el láser “Hokuyo”. Del mismo modo que se acaba de explicar, constarán de un “link” y un “joint”. Se deberán definir la posición de cada uno de ellos respecto al origen y su geometría, además del padre y el hijo según el “TF” (“Transform Frames” o árbol de transformaciones) Figura 52.

```
<link name="camera">
    <collision>
      <origin xyz="0 0 0" rpy="0 0 0"/>
      <geometry>
        <box size="${cameraSize} ${cameraSize} ${cameraSize}"/>
      </geometry>
    </collision>
</link>

<joint name="camera_joint" type="fixed">
    <axis xyz="0 1 0" />
    <origin xyz=".07 0 0.06" rpy="0 0 0"/>
    <parent link="base_link"/>
    <child link="camera"/>
</joint>

<link name="laser">
    <collision>
      <origin xyz="0 0 0" rpy="0 0 0"/>
      <geometry>
    <box size="0.05 0.05 0.07"/>
      </geometry>
    </collision>
</link>

<joint name="hokuyo_joint" type="fixed">
    <origin xyz=".0675 0 0.12" rpy="0 0 0"/>
    <axis xyz="0 1 0" />
    <parent link="base_link"/>
    <child link="laser"/>
</joint>
```

Figura 52 URDF cámara y hokuyo

3.8 STACK DE NAVEGACIÓN

Posiblemente, el "Stack de Navegación" sea una de las herramientas más potentes en ROS, la cual va a permitir mover el robot de forma autónoma. Y, lo que es más, gracias a la comunidad y al código abierto, ROS dispone de una importante cantidad de algoritmos que pueden ser usados para la navegación.

Lo primero de todo, en este capítulo se va a aprender todas las formas necesarias para configurar el "Stack de Navegación" con el robot, dándole metas y configurando algunos parámetros para obtener los mejores resultados. En particular, se cubrirán los siguientes aspectos:

- Introducción al "Stack de navegación" y sus potentes capacidades.
- Explicación del "tf" con el objetivo de mostrar como transformar el "frame" o sistema de coordenadas de un punto a otro, por ejemplo, los datos obtenidos por un determinado sensor como puede ser el hokuyo o los comandos que deben ser enviados a un actuador para alcanzar una posición deseada.
- Se verá cómo controlar y configurar el láser hokuyo.
- Se presentará el "base controller" y se creará uno de nuestro robot.

- Ejecución del paquete SLAM y cómo utilizarlo para crear un mapa del entorno por el que pueda navegar el robot posteriormente.
- Y finalmente, se aprenderá a localizar el robot en el mapa utilizando los algoritmos de localización.

En la Figura 53 se puede observar cómo está organizado el "Stack de Navegación". Se diferencian tres grupos de recuadros: uno en color blanco, otro en color gris y finalmente, otro en azul. El primer grupo indica las pilas que son proporcionadas por ROS, el segundo los nodos que son opcionales y el tercero, los nodos específicos [11].

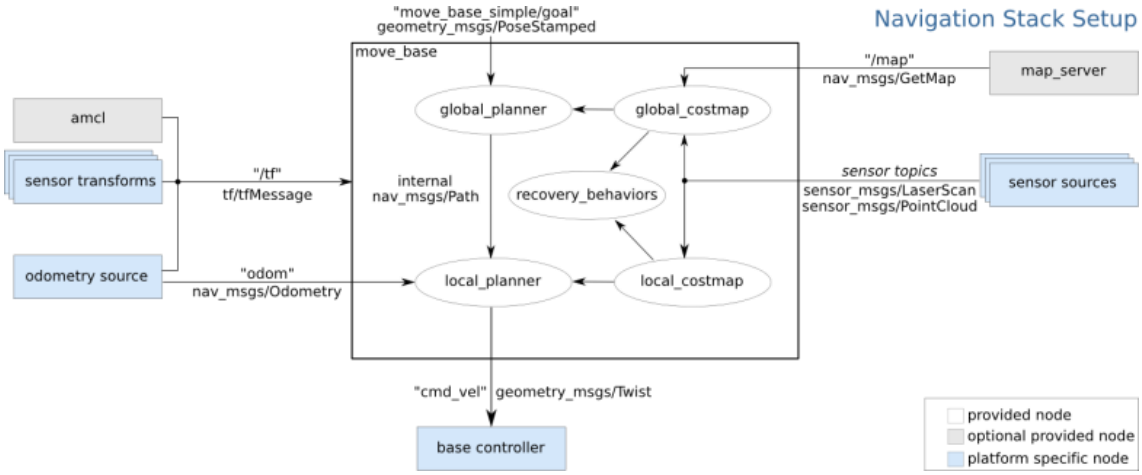


Figura 53 Diagrama de organización del Stack de Navegación

En los siguientes apartados se explicará cómo los nodos necesarios para hacer operativo el stack de navegación. Estas dependen de la plataforma que se utilice como robot móvil, por lo que será necesario escribir código para adaptarla y que pueda ser usada por ROS y por el "Stack de Navegación".

3.8.1 TRANSFORMACIONES ("TF")

El "Stack de Navegación" necesita conocer la posición de los sensores, las ruedas y las articulaciones [1] [7].

Para hacer esto utilizamos el paquete ROS de "TF" (que son las siglas de "Transform Frames"). Esta gestiona el árbol de transformación. Todo esto se podría hacer con procedimientos matemáticos, pero si tienes numerosos "frames" que calcular, puede ser muy complicado y desastroso.

Gracias al "TF" se pueden añadir más partes y sensores al robot, que serán manejados automáticamente por esta librería. Por ejemplo, si se quiere añadir un láser y la posición de este va a ser 15cm hacia delante y 10 hacia arriba respecto del origen (donde se haya establecido en el robot), se necesitará un nuevo frame con estos valores que ligue esta posición al origen.

Una vez creado e insertado, es fácil conocer la posición del láser con respecto al valor del "base_link" o las ruedas. Lo único que se necesita es realizar una llamada a la biblioteca "TF" y obtener la transformación.

Además, ROS cuenta con una herramienta para poder visualizar un árbol de transformaciones ("trasnform tree"), utilizando el siguiente comando

```
$ rosrun tf view_frames
```

El resultado se muestra en la Figura 54 :

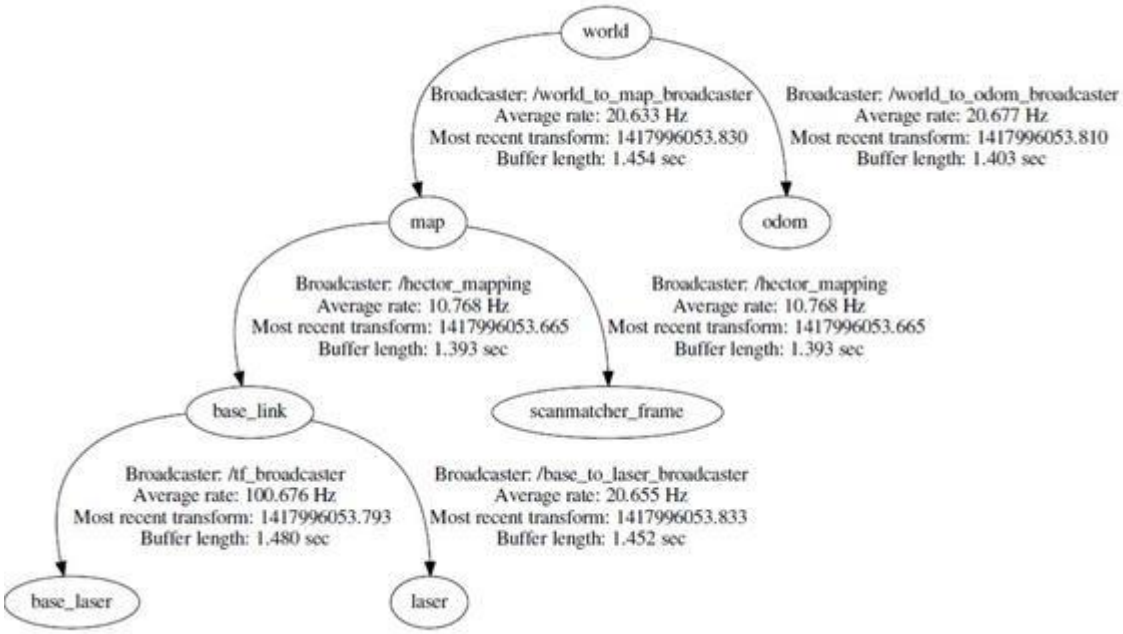


Figura 54 Árbol de transformaciones

3.8.2 PUBLICAR INFORMACIÓN DE LOS SENSORES

El robot puede tener multitud de sensores para captar todo lo que le rodea al igual que numerosos nodos programados para tomar esos datos y realizar alguna acción al respecto. No obstante, el "Stack de Navegación" está preparado única y exclusivamente para utilizar los datos de un sensor laser planar, como puede ser el "Hokuyo". Más concretamente, el sensor debe publicar sus datos con alguno de estos dos tipos de mensaje: "sensor_msgs/LaserScan" o "sensor_msgs/PointCloud".

En caso de que no se disponga de un láser de este tipo, cabe la posibilidad de colocar numerosos sensores de ultrasonidos en la periferia del robot y mediante un pequeño código transformar esos datos en "PointClouds". Sin embargo, no es el caso, ya que se dispone de un láser "Hokuyo" que publica su información con el tipo de mensaje "sensor_msgs/LaserScan". Para este tipo de láser no es necesario configurar nada ya que, gracias a la comunidad, se dispone de un paquete "Hokuyo_node" que al ejecutarlo con el comando "rosrun" abre el puerto donde se

encuentra conectado el láser y comienza a publicar la información en el topic "/scan". En el manual de usuario podrá verse como dar permisos a los puertos para poder ejecutarlo correctamente.

Con las herramientas de visualización que dispone ROS como, por ejemplo, "rviz", será muy sencillo visualizar la información que publican estos sensores. En la Figura 55se muestra una captura de visualización de los datos del láser.

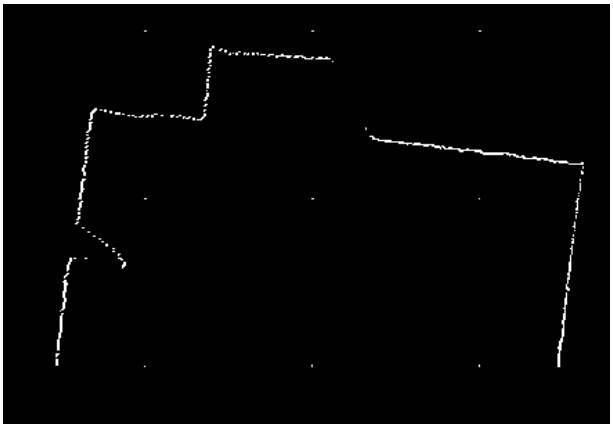


Figura 55 Visualización láser Hokuyo

3.8.3 PUBLICAR INFORMACIÓN DE ODOMETRÍA

El "Stack de Navegación" también necesita recibir información sobre la odometría del robot. La odometría es la distancia del origen del robot respecto a un punto de referencia. En este caso, es la distancia entre el "base_link" y un punto fijo que se establece en el frame de "odom".

El tipo de mensaje que es usado para el "Stack de Navegación" es "nav_msgs/Odometry". Se puede observar la estructura de este mensaje ejecutando el siguiente comando:

```
$ rosmsg show nav_msgs/Odometry
```

El resultado se puede ver en la Figura 56

```
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
string child_frame_id
geometry_msgs/PoseWithCovariance pose
  geometry_msgs/Pose pose
    geometry_msgs/Point position
      float64 x
      float64 y
      float64 z
    geometry_msgs/Quaternion orientation
      float64 x
      float64 y
      float64 z
      float64 w
  float64[36] covariance
geometry_msgs/TwistWithCovariance twist
  geometry_msgs/Twist twist
    geometry_msgs/Vector3 linear
      float64 x
      float64 y
      float64 z
    geometry_msgs/Vector3 angular
      float64 x
      float64 y
      float64 z
  float64[36] covariance
```

Figura 56 Estructura del mensaje "nav_msgs/Odometry"

Como se puede apreciar en la Figura 56, la odometría da la posición del robot entre “frame_id” y “child_frame_id”. También proporciona la posición y orientación (pose) del robot utilizando el tipo de mensaje “*geometry_msgs/Pose*” y la velocidad con “*geometry_msgs/Twist*”.

El campo “pose” tiene dos estructuras que muestran la posición en coordenadas de Euler y la orientación del robot utilizando un cuaternio. La orientación es el desplazamiento angular del robot.

El campo de velocidad tiene también dos estructuras que indican la velocidad linear y angular. En este caso, solamente se utiliza la velocidad linear en “x”, para moverlo hacia delante o hacia atrás en función de si es positivo o negativo; y la velocidad angular en “z” para girar a izquierda o derecha.

Finalmente, indicar que como la odometría realmente es un desplazamiento entre dos frames, es necesario publicar también el “TF”.

3.8.4 CREAR EL “BASE_CONTROLLER”

Es uno de los elementos más importantes en el “Stack de Navegación” porque es la única manera para controlar el robot de manera efectiva. Se comunica directamente con la electrónica del robot [1].

ROS no proporciona un estándar de "base_controller", por lo que se debe escribir el código necesario para crear la propia base de la plataforma móvil.

El robot será controlado con el tipo de mensaje *"geometry_msgs/Twist"*. Por lo tanto, el "base_controller" debe suscribirse al topic con el nombre "*cmd_vel*" y generar los comandos correctos para mover la plataforma con las velocidades lineales y angulares correctas. En la Figura 56 se puede ver la estructura de este mensaje.

```
def callback_vel_command(msg):

    global ref_vel_right,ref_vel_left

    print
'callback_vel_command::msg(',msg.linear.x,'',msg.angular.z,')'
    ref_vel_right,ref_vel_left=diff_ctrl.command_velocity(msg)

    PBR_L.SetSpeedMotor(ref_vel_left)
    PBR_R.SetSpeedMotor(ref_vel_right)

    print 'vel INT(left=',ref_vel_left,',right=',ref_vel_right,')'

def command_velocity(self, vel_twist):

        target_v = vel_twist.linear.x
        target_w = vel_twist.angular.z

        vr = (2*target_v + target_w*self.L) / (2)
        vl = (2*target_v - target_w*self.L) / (2)

        print 'command_velocity::(',vl,'',vr,')'
        vr_tangent=self.tangentvel_2_angularvel(vr)
        vl_tangent=self.tangentvel_2_angularvel(vl)
        print 'command_velocity::(',vl,'',vr,')'

        # Mapping angular velocity targets to motor commands
        vr_tangent=vr_tangent*self.rad_2_pulses*self.k_rad_2_pulses
        vl_tangent=vl_tangent*self.rad_2_pulses*self.k_rad_2_pulses

        # Transfom meters/s to pulse/s
        print
'command_velocity::vl_tangent(',vl_tangent,'',vl_tangent,')'

        vr_tangent=min(int(vr_tangent),self.lim_max_pulses)
        vr_tangent=max(int(vr_tangent),self.lim_min_pulses)

        vl_tangent=min(int(vl_tangent),self.lim_max_pulses)
        vl_tangent=max(int(vl_tangent),self.lim_min_pulses)

        return int(vr_tangent),int(vl_tangent)
```

Figura 57 fragmento de código que muestra el "base_controller"

En cuanto al código, puede verse completo en el Anexo D, con las funciones *"callback_vel_command(msg)"* en el fichero *"edubot_control_main.py"* que llama a la función *"command_velocity(vel_twist)"* del fichero *"diffdrive_controller.py"*. En la Figura 57 se muestra un pequeño fragmento del código, que desempeña la función de "base_controller" en el robot.

3.8.5 MAPAS: CREAR, GUARDAR Y CARGAR

La creación de un mapa del entorno puede ser una tarea, en ocasiones, ardua y tediosa si no se tienen las herramientas adecuadas para ello. No obstante, ROS cuenta con una serie de ellas que hacen de esta tarea algo fácil y sencillo. ROS es capaz de crear un mapa utilizando la información de la odometría y del láser [7]. Entre estas herramientas destacan "map_server" y "gmapping". Permiten crear un mapa, guardarlo y cargarlo de nuevo.

Para utilizarlo de forma cómoda y sencilla se ha creado un fichero *".launch"*, que lanza el nodo "slam_gmapping" con todos los parámetros necesarios y, además, abre el "rviz" para visualizarlo. Se puede ver el código completo en el Anexo D.

Además, en otra ventana es aconsejable lanzar el "launch" "edubot_teleop" (puede verse en el Anexo D). Este lanza el nodo de teleoperación publicando velocidades con el topic "/base_control/cmd_vel" para comunicarse correctamente con el robot. De esta forma, es posible manejar el robot desde el teclado del ordenador de forma remota.

Una vez que el robot comienza a moverse, en la herramienta de visualización (Figura 58 visualización del proceso de creación del mapa) "rviz" se puede apreciar de diferentes colores los tipos de espacios que hay: en color gris claro se aprecian las zonas libres para la navegación; en gris oscuro las zonas aún desconocidas; y, por último, de color negro las líneas que se corresponden con los obstáculos.

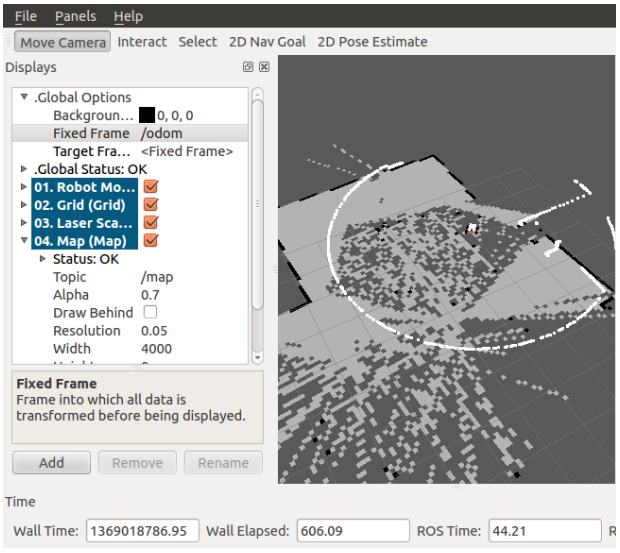


Figura 58 visualización del proceso de creación del mapa

El nodo “slam_mapping” va actualizando esta información gracias a los datos recibidos tanto del láser “hokuyo” como de la odometría. De esta forma, va localizando de manera precisa al robot sobre el mapa y construyendo el “OGM” (Occupancy Grid Map) o mapa gráfico de ocupación.

Una vez finalizado el rastreo de una zona y obtenido el mapa correspondiente, es hora de guardarlo para su posterior utilización. Para ello, se utiliza el siguiente comando:

```
rosrun map_server map_saver -f map
```

Este comando creará dos archivos, “map.pgm” y “map.yaml”. El primero de ellos es el mapa en formato “.pgm”, que significa, “formato portable de mapa gris”. El otro, es un fichero de configuración para el mapa, en el que se encuentran parámetros como la resolución, el origen, etc.

Finalmente, cuando se quiera volver a utilizar el mapa para la navegación, es necesario cargarlo con el paquete “map_server”. Para ello, se utiliza el siguiente comando:

```
rosrun map_server map_server map.yaml
```

Sin embargo, para que sea más sencillo, se ha creado un fichero “.launch” que además de cargar el mapa, lanza el nodo “amcl” (“adaptive Monte Carlo localization”) nodo de localización y abre el rviz. Puede verse en el Anexo D archivo navigation.launch.

3.8.6 COSTMAPS

Una vez desarrollado todo lo anterior, necesario para el correcto funcionamiento del “Stack de Navegación”, se pasará a la configuración del mismo.

Una de las configuraciones más importantes reside en los “costmaps”. El robot se moverá a través del mapa utilizando dos tipos de navegación, global y local.

- La navegación global es utilizada para crear rutas o caminos hacía un punto objetivo en el mapa, a una distancia considerable.
- La navegación local es utilizada para crear rutas en distancias cortas y para evitar obstáculos, por ejemplo, un cuadrado de 4x4 alrededor del robot.

Estos módulos usan los “costmaps” para guardar toda la información de los mapas. El “global costmap” es utilizado por la navegación global y el “local costmap” es utilizado por la navegación local.

Estos “costmaps” cuentan con numerosos parámetros para configurar su comportamiento [7][1]. Consiste básicamente en tres ficheros:

- Costmap_common_params.yaml

- Global_costmap_params.yaml
- Local_costmap_params.yaml

En estos ficheros se pueden encontrar multitud de parámetros entre los que cabe destacar: giros y velocidades máximas y mínimas, aceleraciones, dimensiones del perímetro de seguridad del robot para evitar choques, precisión y tolerancia al punto objetivo, tanto en la posición (en metros) como en la orientación (en radianes), etc.

Pueden verse estos archivos de configuración en el Anexo D.

3.8.7 CONFIGURACIÓN "RVIZ"

Para cualquier aplicación en tiempo real, es conveniente visualizar la mayor cantidad de datos posibles, siempre que estos sean relevantes. En el caso de la navegación, y gracias a la herramienta de visualización "rviz", se tiene la capacidad de observar una gran cantidad de datos para supervisar que todo funcione correctamente.

Se enumera, a continuación, una serie de "topics" de visualización que se recomiendan añadir a la herramienta "rviz" [1]:

- **2D pose estimate**: más que un "topic", se trata de una herramienta que permite al usuario inicializar la localización del robot en el mapa cargado, tanto su posición como su orientación, como se puede ver en Figura 59.

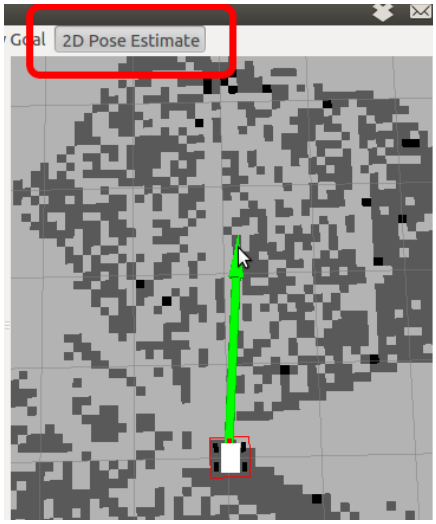


Figura 59 Herramienta "2D pose"

Esta herramienta publica un topic con el nombre "/initialpose" que contiene la información de la posición y la orientación en un mensaje con el tipo de formato "geometry_msgs/PoseWithCovarianceStamped" el cual contiene dos campos, un "header" o cabecera con información y un "pose" el cual contiene los campos de "position" y "orientation".

El programa principal de "edubot_control_main" (Anexo D) debe suscribirse al topic "/initialpose" y cuando llegue un mensaje de este tipo, forzar a la propia odometría a adoptar esta posición y orientación, ya que, si no, el "stack de navegación" estaría posicionando al robot según "/initialpose" pero estaría recibiendo datos de la odometría del robot indicándole que está en otro punto.

- 2D nav goal: esta herramienta permite al usuario marcar un punto objetivo para la navegación, estableciendo una posición deseada para que el robot alcance.
  Al igual que la anterior, publica un "topic" con el nombre "/move_base_simple/goal" al que el stack de navegación estará suscrito y esperando que llegue información.
  Con el mismo método que utilizaba la herramienta anterior, se puede establecer la posición y orientación del punto objetivo, como se muestra en la Figura 60.

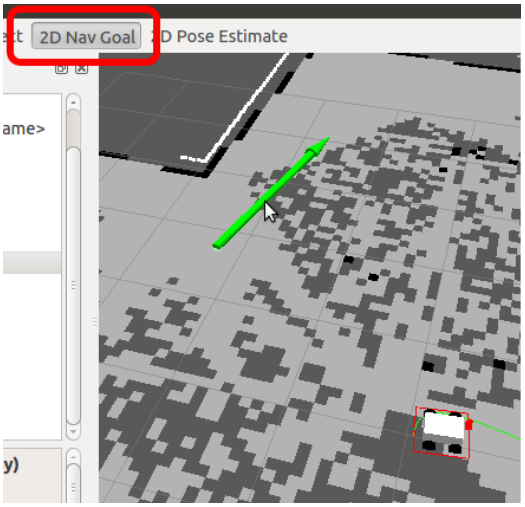


Figura 60 Herramienta 2D Nav Goal

- Static Map: Muestra el mapa ofrecido por el nodo "map_server", si es que anteriormente ha sido creado, como se vió en el apartado 3.8.5.

- Particle Cloud: Saca por pantalla la nube de puntos utilizada por el sistema de localización del robot. La propagación de la nube representa la incertidumbre del sistema de localización sobre la posición del robot. Una nube de puntos que se extiende mucho refleja una alta incertidumbre, mientras que una nube condensada representa una baja incertidumbre. Un ejemplo puede verse en la Figura 61.

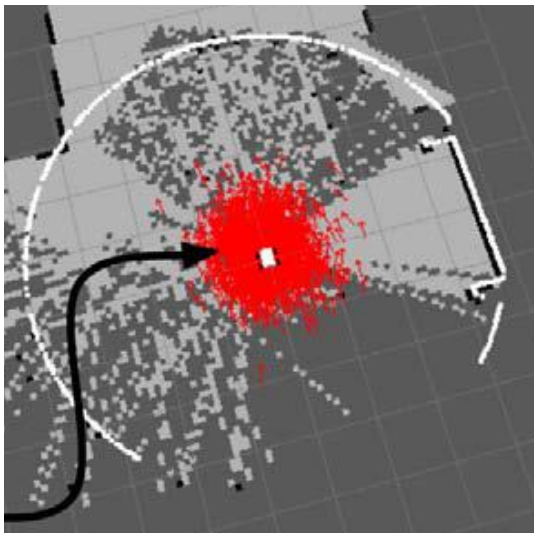


Figura 61 Nube de puntos

La información se publica en el topic "/particlecloud" que tiene la estructura del tipo de mensaje "geometry_msgs/PoseArray".

- **Inflated obstacles:** Muestra los obstáculos en el mapa de costes o "costmaps" del "stack de navegación" inflados por el radio inscrito del robot, otro de los parámetros que puede ser establecido en los archivos de configuración del "costmaps" como se vió en COSTMAPS. Para evitar colisiones, el punto central del robot nunca debe estar ocupando una de las celdas azules que contienen los "obstáculos inflados" (Figura 62).
  La información se publica en el topic "/local_costmap/inflated_obstacles" con el tipo de mensaje "nav_msgs/GridCells".

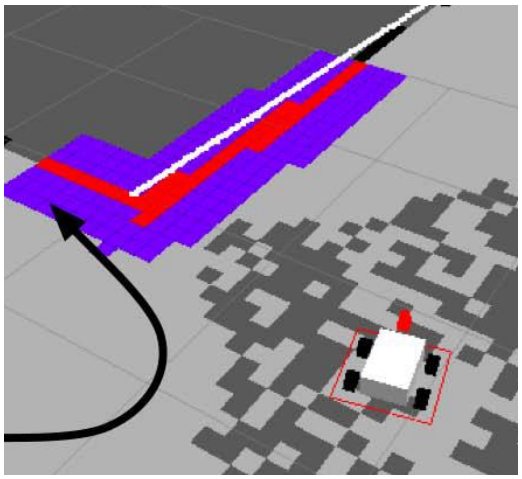


Figura 62 Objetos inflados

- **Global plan:** Representa la parte del plan globlal ("global plan") que el planificador local ("local planner") está siguiendo en ese momento. La trayectoria se muestra con una línea de color verde (Figura 63. Cabe la

posibilidad de que, en su recorrido, el robot encuentre algún obstáculo durante el movimiento, y el plan local ("local plan") recalcule la ruta a seguir para evitar dichos obstáculos e intentar seguir el plan global ("global plan").

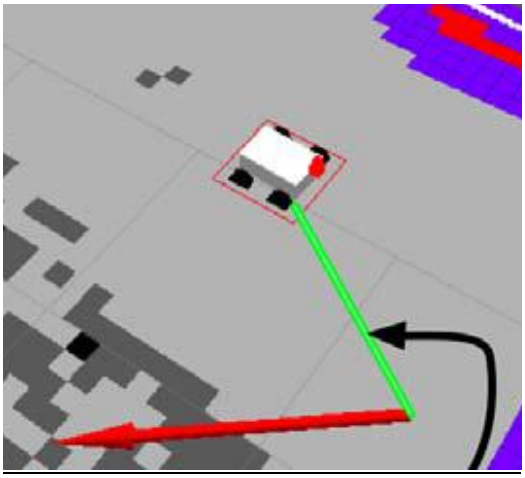


Figura 63 Trayectoria del "global plan"

La información se publica en el topic "/TrajectoryPlannerROS/global_plan" con el tipo de mensaje "nav_msgs/Path".

- **Local plan:** Muestra la trayectoria asociada con los comandos de velocidad que actualmente están siendo enviados a la base por el planificador local (local planner). Se puede ver la trayectoria en una línea de color azul (Figura 64).

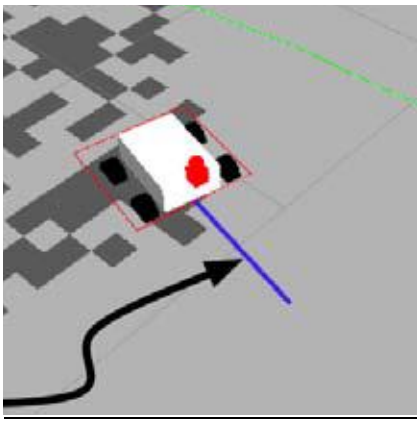


Figura 64 Trayectoria del "local plan"

La información se publica en el topic "/TrajectoryPlannerROS/local_plan" con el tipo de mensaje "nav_msgs/Path".

- **Planner Plan:** Representa el plan completo para el robot calculado por el planificador global . Es muy similar al plan global.
  La información se publica en el topic "/NavfnROS/plan" con el tipo de mensaje "nav_msgs/Path".

Con esto, se da por finalizado el capítulo 3.

CAPÍTULO 4. RESULTADOS

En este capítulo se van a seguir tres líneas de actuación, a saber: pruebas y ajuste de la odometría, establecimiento de parámetros adecuados para el algoritmo sigue líneas y, por último, configuración y ajuste del stack de navegación.

Es un capítulo muy importante, ya que deja constancia de que se han alcanzado los objetivos iniciales.

4.1 ODOMETRÍA

Como ya se explicó en el apartado 3.6, son necesarios dos parámetros para la estimación de la posición, que son: el diámetro de las ruedas y la distancia entre ejes. Con el primero, y con ayuda de los codificadores incrementales, se consigue calibrar la distancia recorrida por cada una de las ruedas, mientras que con el segundo parámetro se calibran los giros.

El proceso de calibración a seguir consta de tres etapas: primero se deben calibrar las distancias, posteriormente las desviaciones y, por último, los giros. Cuando se calibra un parámetro se toma como bueno para el siguiente experimento y así sucesivamente hasta que se complete el proceso de calibración, momento en que se evaluará si los resultados obtenidos son adecuados.

4.1.1 Calibrar distancia

El primer paso es ajustar el diámetro de las ruedas para que en concordancia con la información de los pulsos que llegan de los codificadores incrementales, se obtenga un dato preciso de la distancia que realmente ha recorrido el robot. Por tanto, en esta prueba no tendrá ninguna influencia el parámetro de separación entre ejes.

Para realizar el experimento, se precisa de una pequeña pista de longitud conocida. En este caso, se ha dibujado una pequeña pista en el suelo con cinta y junto a ella se ha acoplado un metro como se puede ver en la Figura 65 y Figura 66.

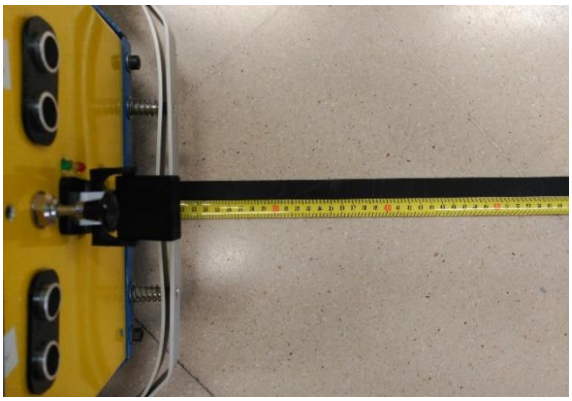


Figura 65 Medidas de odometría (1)

Figura 66 Medidas de odometría (2)

Además, para obtener mayor precisión, se mueve el robot de forma manual, lenta y empujándole desde arriba, de forma que se trate de evitar el deslizamiento de alguna de las ruedas.

El método que se llevará a cabo es el siguiente: Se parte de un valor del radio de $r = 8\,cm$ tras llevar a cabo una medida aproximada del díametro de la rueda. A continuación, se moverá al robot, como se ha indicado, una distancia de dos metros y se llevará a cabo una comprobación tanto gráfica como escalar de las medidas obtenidas según la odometría. En caso de que el valor supere los dos metros, se deberá reducir el radio, mientras que, en caso contrario, se aumentará.

Los experimentos que se llevaron a cabo pueden verse en la Tabla 7

Nº del experimento	Radio (mm)	Distancia real (m)	Distancia medida (m)	Figura
1	80	2.00	2.27	Figura 67
2	70	2.00	1.97	Figura 68
3	75	2.00	2.11	Figura 69
4	72	2.00	2.3	Figura 70
5	71	2.00	2.01	Figura 71

Tabla 7 Calibración distancia (1)

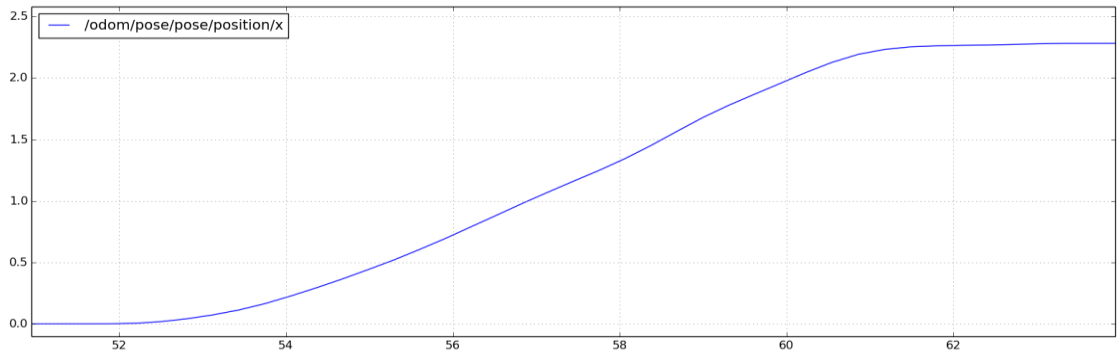


Figura 67 Distancia-tiempo (1)

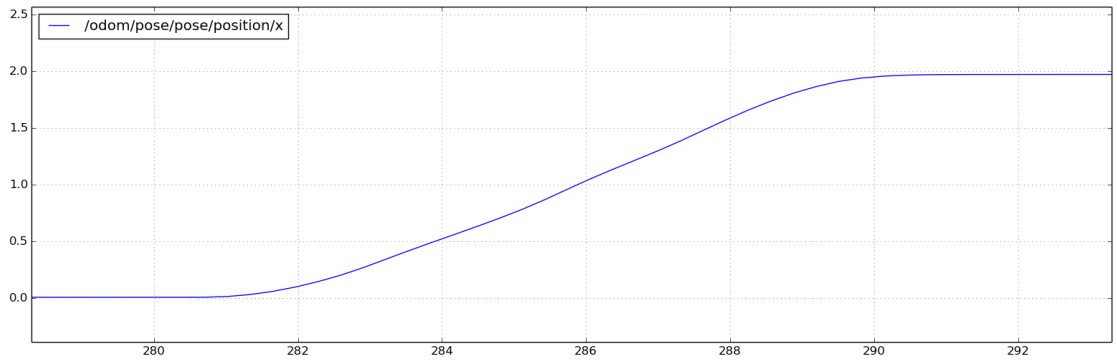


Figura 68 Distancia-tiempo (2)

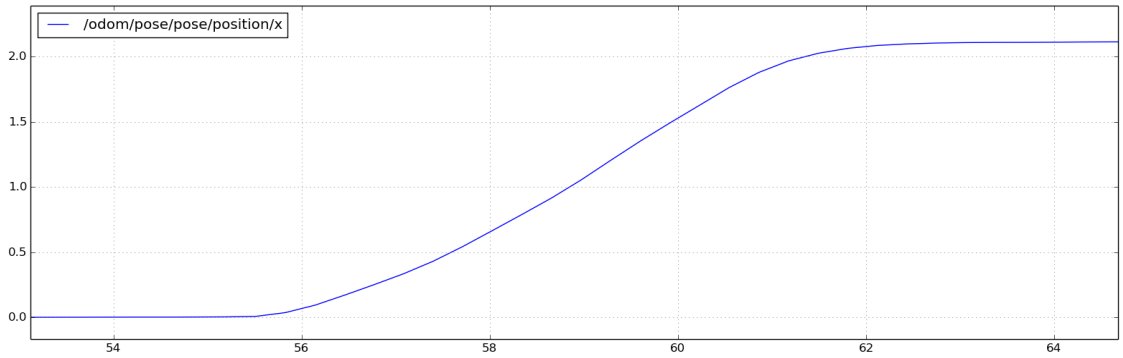


Figura 69 Distancia-tiempo (3)

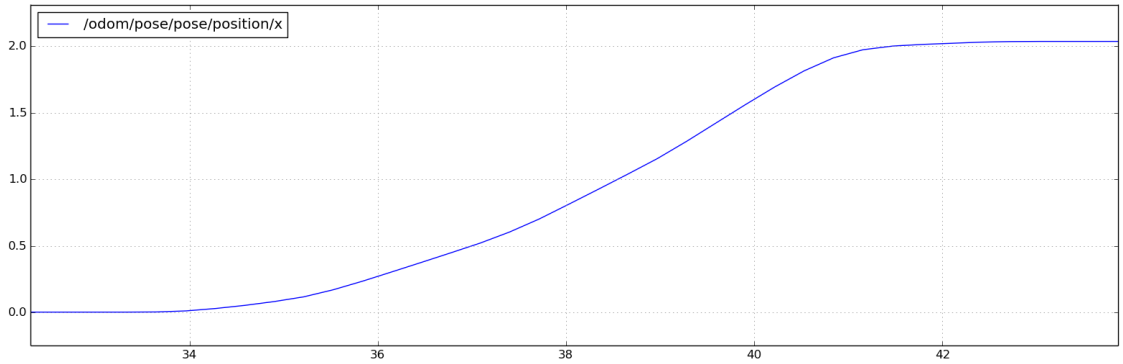


Figura 70 Distancia-tiempo (4)

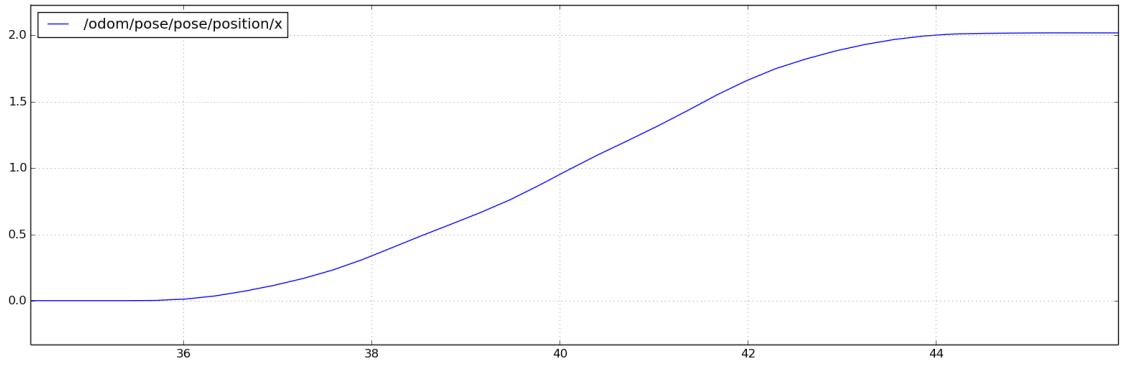


Figura 71 Distancia-tiempo(5)

Se detiene aquí el experimento porque con la prueba Nº5 se llega a una solución bastante aceptable. De nuevo para corroborar los resultados, se realiza otra prueba más con este valor de radio, pero esta vez se tomarán tres medidas, una cada metro recorrido. El resultado puede verse en la Tabla 8.

Nº del experimento	Radio (mm)	Distancia real (m)			Distancia medida (m)			Figura
7	71	1.00	2.00	3.00	1.00	2.04	3.00	Figura 72

Tabla 8 Calibración distancias (2)

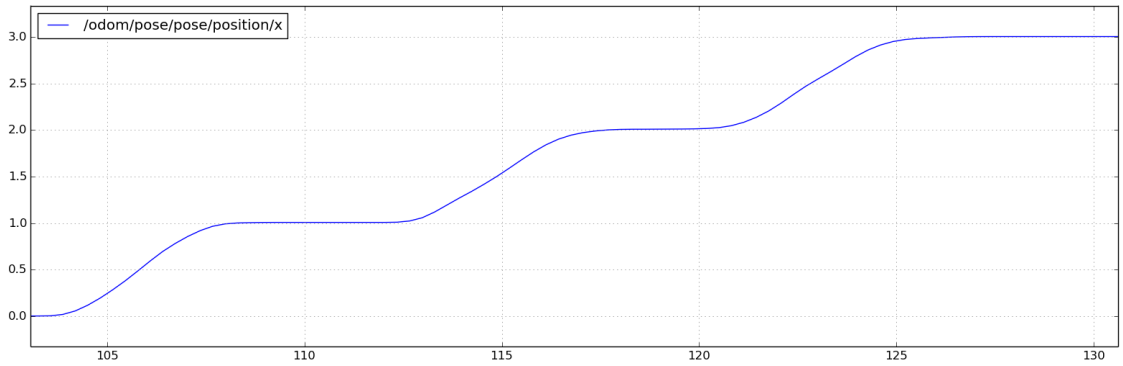


Figura 72 Distancia-tiempo (6)

Se da por bueno el valor del radio escogido

4.1.2 Calibrar desviaciones

Suele ocurrir que cuando se le ordena al robot moverse una cierta distancia en línea recta, este no lo haga perfectamente. Esto es debido a que las ruedas no son exactamente iguales, aunque solo difieran en unos cuantos milímetros. Por lo tanto, el siguiente paso será ajustar los radios de cada una de las ruedas, para que,

en concordancia con los pulsos medidos por el codificador incremental, avancen exactamente lo mismo.

El procedimiento a seguir es el siguiente: Según la Figura 73Figura 73 Trayectoria ideal (rojo); desviación a la derecha (azul); desviación a la izquierda (verde), la trayectoria deseable sería la roja. Si toma la trayectoria verde querrá decir que la rueda izquierda es más pequeña que la derecha y, al contrario, si la rueda derecha es más pequeña tomará la trayectoria azul [6].

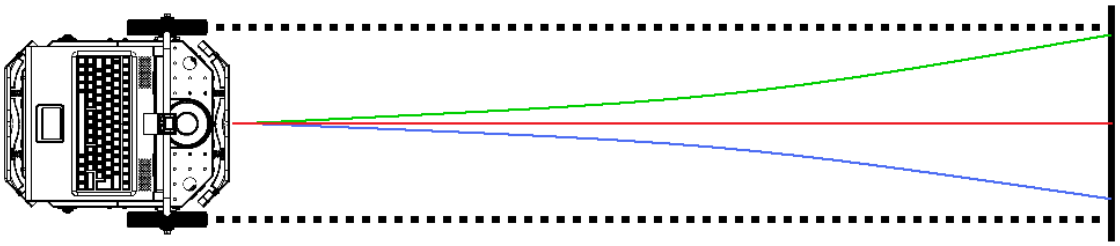


Figura 73 Trayectoria ideal (rojo); desviación a la derecha (azul); desviación a la izquierda (verde)

4.1.3 Calibrar giros

Una vez completadas con éxito las dos etapas anteriores, se pasará a calibrar los giros del robot. Para ello, el único parámetro que se debe ajustar es la distancia entre ejes. Si el valor es mayor de lo debido, el robot recorrerá una circunferencia respecto a un eje instantáneo de rotación mayor que la comandada, y al contrario, dibujará una circunferencia menor si la distancia entre ejes no es suficiente.

Se llevarán a cabo dos procesos de calibración: Primero se ajustará correctamente el giro de 90º y una vez que éste sea adecuado se comprobará que el robot hace correctamente el cuadrado de calibración.

Al igual que en calibraciones anteriores, el robot se moverá manualmente y con una velocidad lenta, de modo que se evite cualquier posible deslizamiento de las ruedas. El procedimiento a seguir para la calibración del giro a 90º será el siguiente: El robot se moverá dos metros en línea recta y una vez se encuentre en ese punto, se realizará un giro de 90º y se moverá otros dos metros en línea recta. De este modo, el robot habrá recorrido una distancia de dos metros en el eje “x” y otros 2 metros en el eje “y”.

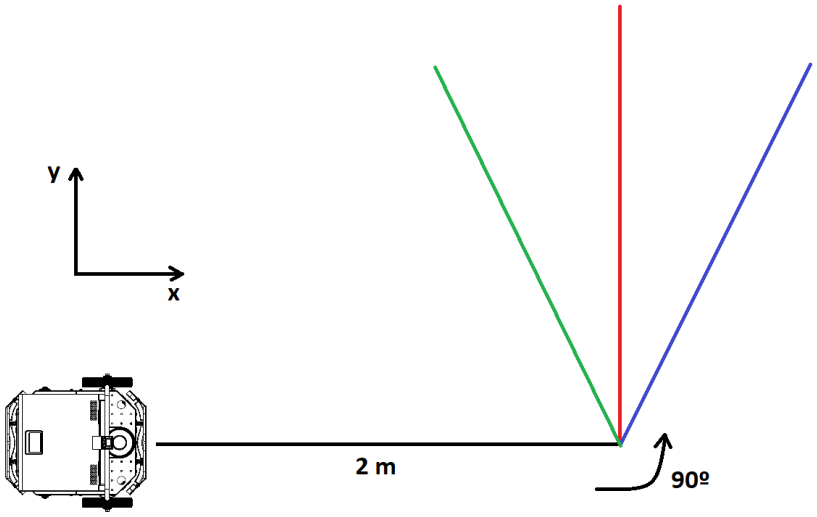


Figura 74 Calibración giros: trayectoria ideal (roja), giro insuficiente (azul), giro sobrepasado (verde)

Como se ve en la Figura 74 la trayectoria ideal a seguir por el robot sería la de color rojo. En caso de obtener una trayectoria similar a la azul será porque el giro es insuficiente, y esto se da porque el parámetro de distancia entre ejes es mayor de lo que debe ser. Al contrario, si se obtiene una trayectoria similar a la verde será que el robot ha girado más de lo debido, y esto se produce porque la distancia entre ejes es más pequeña de lo debido.

Una forma muy sencilla de probarlo es comprobar que antes de que el robot realice el giro la lectura del eje "x" indique dos metros. Si así es, cuando el robot esté en el punto final de la trayectoria solo deberá haber aumentado el valor del eje "y" mientras que el "x" permanecería constante. Esto es lo que ocurre con la trayectoria roja de la Figura 74. En caso de que la "x" haya disminuido significará que ha seguido una trayectoria similar a la verde, mientras que si la "x" aumenta habrá seguido una trayectoria como la azul (Ejemplo real en la Figura 75).

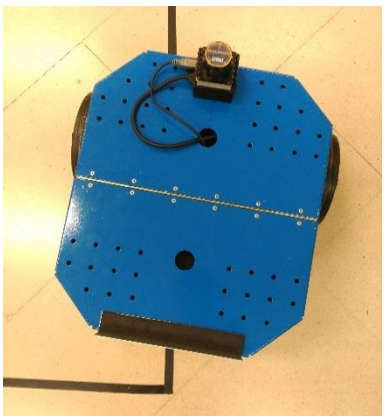


Figura 75 Calibración, giro insuficiente

Por lo tanto, las pruebas de calibración de giros (Figura 76), buscarán ajustar el parámetro de distancia entre ejes para que la medida en "x" ni aumente ni disminuya. Estos experimentos pueden verse en la Tabla 9.

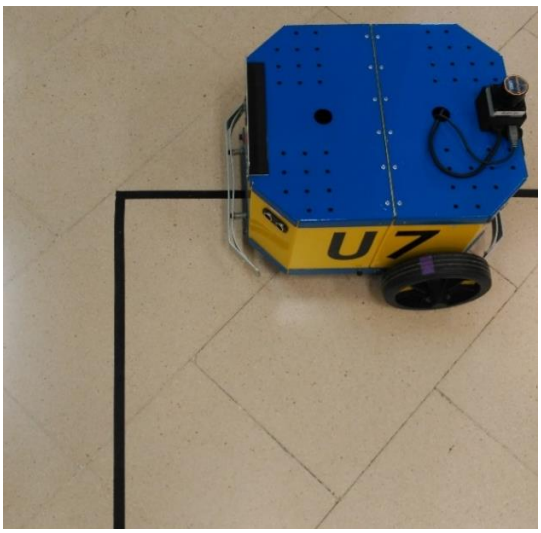


Figura 76 Prueba de calibración de giros

Nº del experimento	Separación entre ejes (cm)	Distancia real (m)		Distancia medida (m)		Figura
		Eje X	Eje Y	Eje X	Eje Y	
1	45	1.00	1.00	1.23	0.87	Figura 77
2	40	1.00	1.00	1.18	0.84	Figura 78
3	35	1.00	1.00	0.95	0.88	Figura 79
4	37	1.00	1.00	1.06	0.84	Figura 80
5	36	1.00	1.00	0.97	0.98	Figura 81

Tabla 9 Pruebas odometría giros

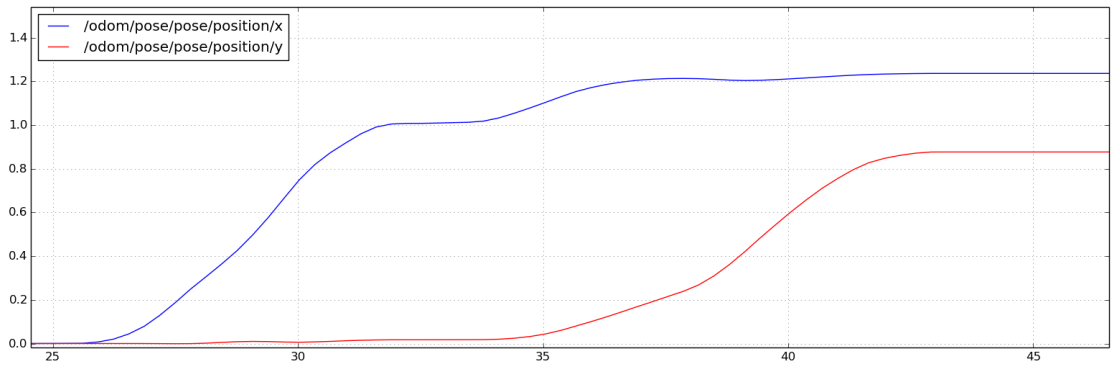


Figura 77 Distancia XY - tiempo (1)

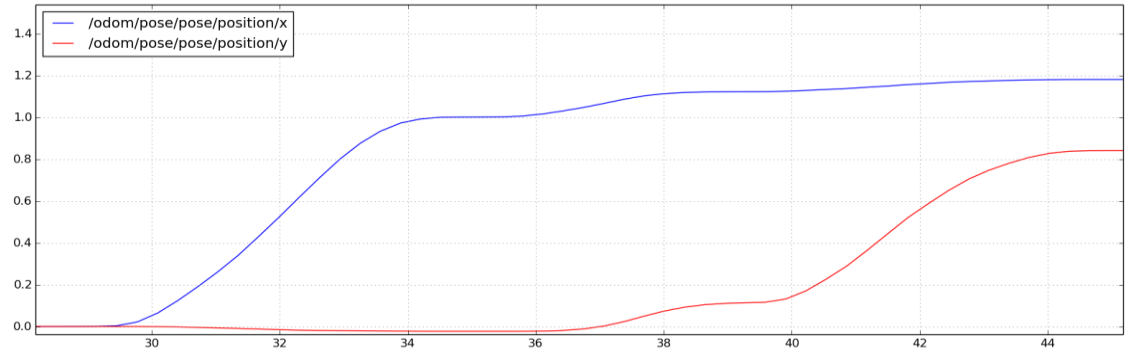


Figura 78 Distancia XY - tiempo (2)

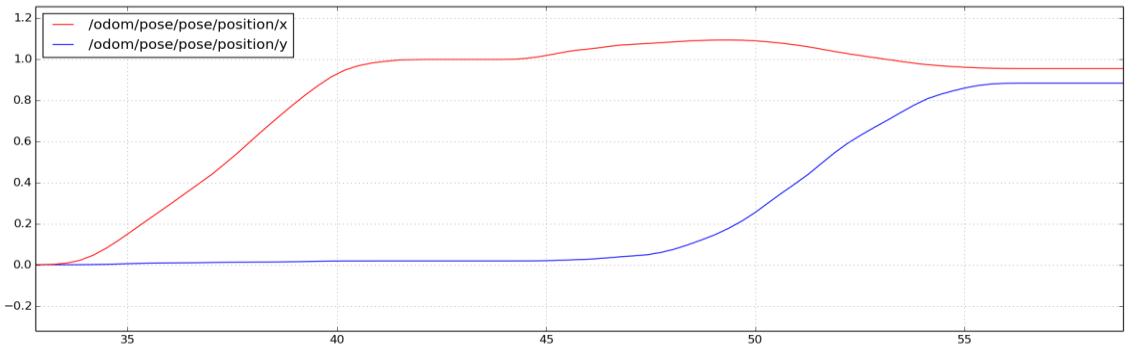


Figura 79 Distancia XY - tiempo (3)

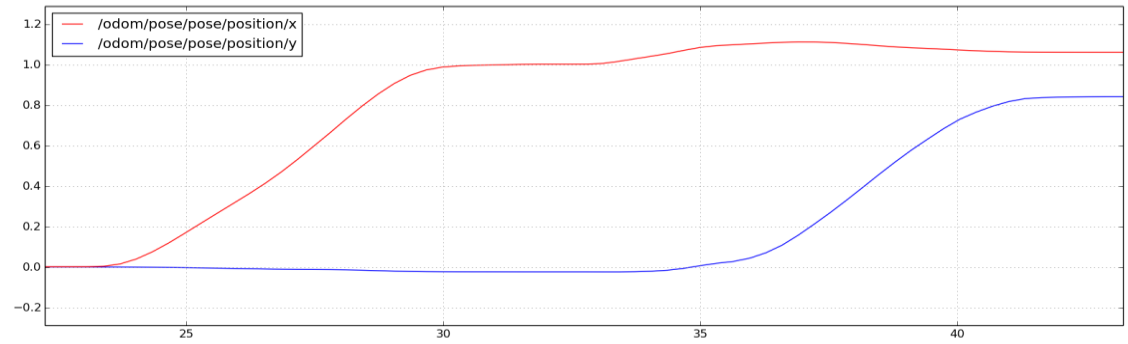


Figura 80 Distancia XY - tiempo (4)

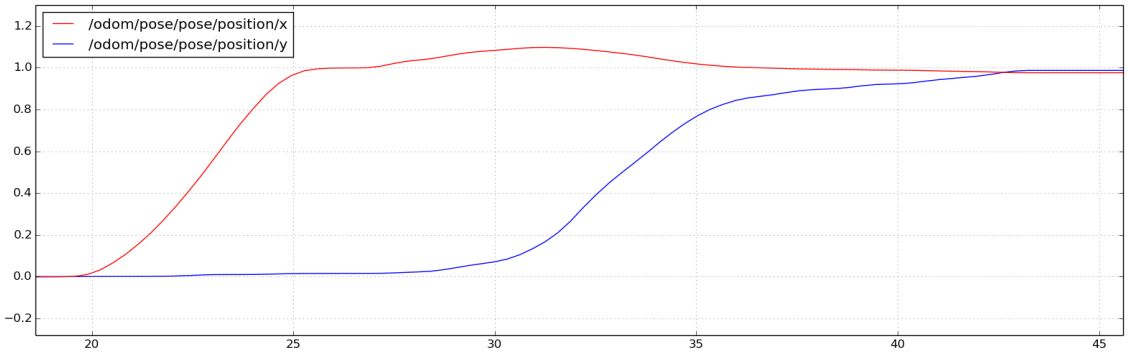


Figura 81 Distancia XY - tiempo(5)

Nuevamente, se detiene el experimento al encontrar unos resultados razonables. Para corroborarlo, se llevará a cabo la prueba del cuadrado de calibración. Esta consiste en hacer que el robot realice un recorrido con forma de cuadrado como se muestra en la Figura 82 [6]. El robot en este desplazamiento hará cuatro giros por vuelta, y dependiendo de cuál sea la posición final con respecto a la inicial se podrá saber si se ha ajustado correctamente el giro y si no es así corregirlo.

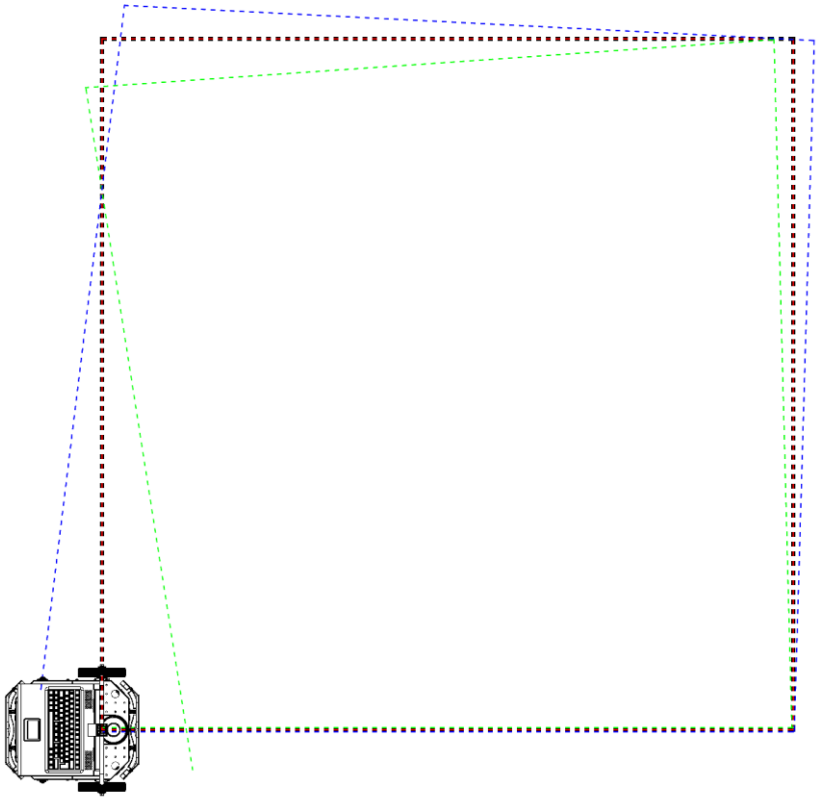


Figura 82 Cuadrado para calibración de giros. Rojo: trayectoria ideal. Azul: trayectoria abierta. Verde: trayectoria cerrada.

En la Figura 83 se puede ver el desplazamiento del robot en los ejes “X” e “Y”. Las franjas de color verde marcan los vértices del cuadrado.

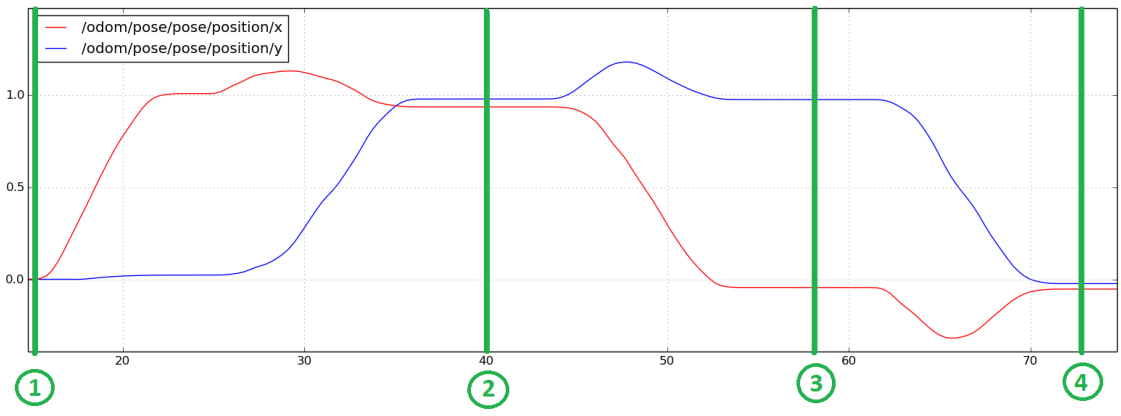


Figura 83 Distancia XY para el cuadrado de calibración

Por realizar una última prueba se ha decidido llevar al robot a una posición de 2 metros en “X” y 2 metros en “Y” de forma libre, es decir sin tener por qué seguir una trayectoria cuadrada. En este caso se ha elegido una trayectoria prácticamente diagonal y los resultados son los que se muestran en la Figura 84.

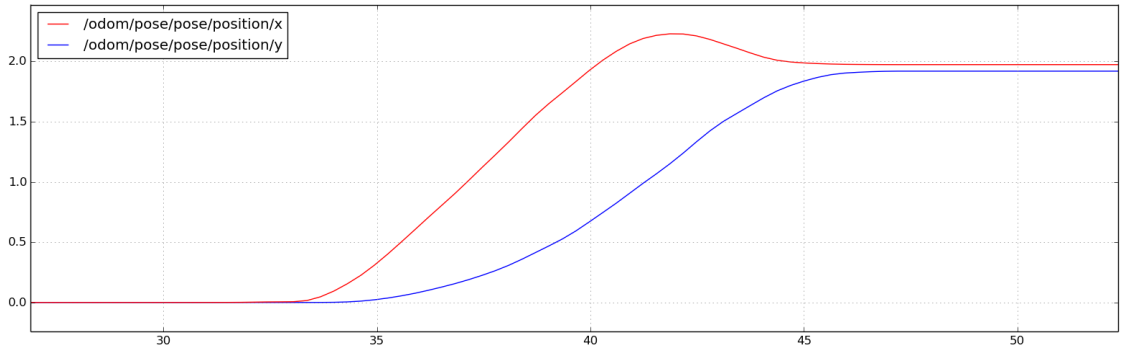


Figura 84 Distancia XY- tiempo (6)

Con estos resultados, la calibración de la odometría se da por completada.

4.2 SIGUE-LÍNEAS

Como se explicó en el apartado 3.4, es necesario ajustar 3 parámetros fundamentales para el seguimiento de la línea. El primero de ellos es la velocidad lineal del robot, un parámetro arbitrario que decide el usuario. Este parámetro se ha establecido en $0.15\ m/s$ una velocidad reducida para que el robot no pierda la referencia de la línea. A partir de este parámetro se ajustarán los otros dos, que son, las constantes proporcionales tanto de giro como de distancia. Cabe recordar que estos parámetros son únicos y exclusivos para cada velocidad, ya que si, por ejemplo, se decide establecer una velocidad más elevada, se necesitarían proporcionales más grandes, para que el sistema fuese más rápido.

Para ajustar estos controles proporcionales se han llevado a cabo una serie de experimentos repetitivos, introduciendo saltos de tipo escalón y observando la respuesta en el tiempo para ver como se alcanzaba el estacionario.

ROS cuenta con una herramienta muy adecuada para este tipo de experimentos. Se denomina “rosbag” y se encarga de guardar la información de un topic a lo largo del tiempo en un fichero de tipo “.bag”. La potencia de esta herramienta reside en que, posteriormente, se puede volver a reproducir el topic desde el archivo “.bag” correspondiente e incluso, graficarlo mediante la herramienta de visualización “rqt_bag”.

4.2.1 Ajuste del "Kd"

Este parámetro es la constante proporcional de distancia. Este será el valor por el que se multiplique la variable distancia y cuyo resultado influirá en el valor del giro.

Para el ajuste de este parámetro se ha llevado a cabo el siguiente experimento. Se coloca al robot descentrado de la línea respecto de su eje longitudinal y le se le da la orden de seguir la línea (Figura 86 y Figura 85). De esta forma, el sistema parte con un salto de tipo escalón e intentará centrarse con la línea.

En la Tabla 10 se pueden ver los experimentos realizados.

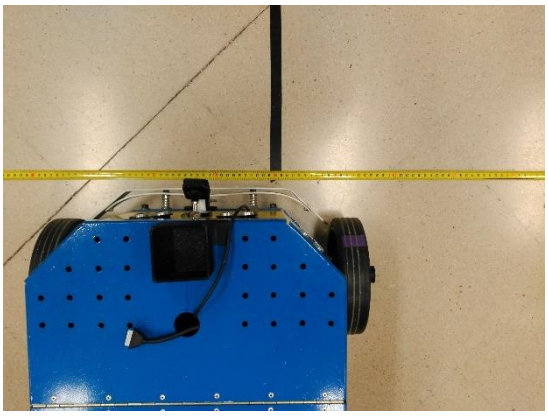


Figura 85 Robot descentrado de la línea

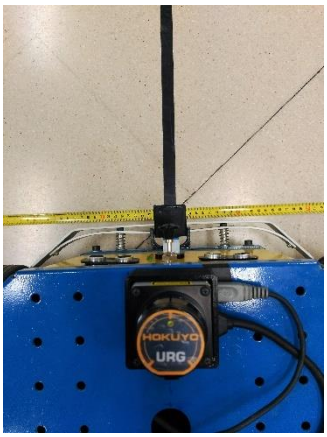


Figura 86 Robot centrado

Tabla 10 Pruebas sigue líneas para establecer "Kd"

Nº del experimento	Escalón (mm)	Valor del "Kd"	Observaciones
1	100	0.1	El proporcional es demasiado alto, gira demasiado y pierde la línea
2	100	0.01	Sigue siendo alto, gira demasiado y pierde la referencia. Figura 87
3	100	0.001	Consigue seguir la línea, pero el comportamiento es muy lento. Figura 88
4	100	0.002	Respuesta algo más rápida que la anterior. Buen resultado. Figura 89
5	100	0.003	Hasta ahora el mejor resultado obtenido. La respuesta es rápida y alcanza muy bien el estacionario. Figura 90
6	100	0.004	Valor un poco elevado, que provoca un gran sobrepico en la respuesta temporal. No alcanza bien el estacionario. El robot oscila mucho cuando encara la recta. Figura 91
7	100	0.005	Demasiado alto. Sobrepico muy grande que hace al sistema inestable. No alcanza el estacionario. Figura 92

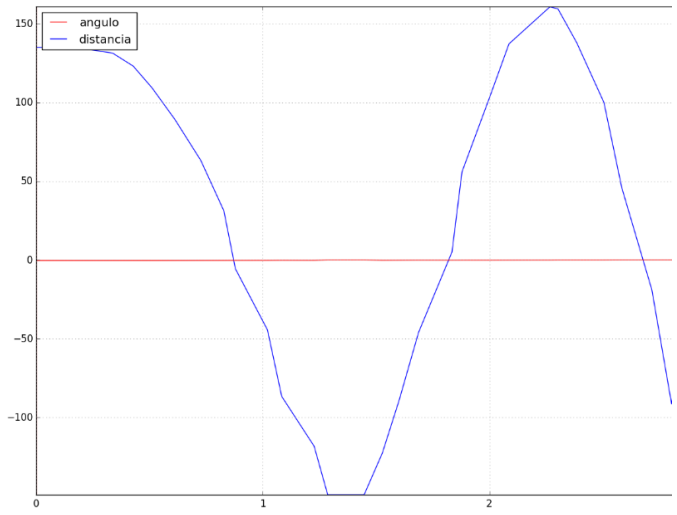


Figura 87 Comportamiento de la distancia frente al tiempo "Kd"=0.01

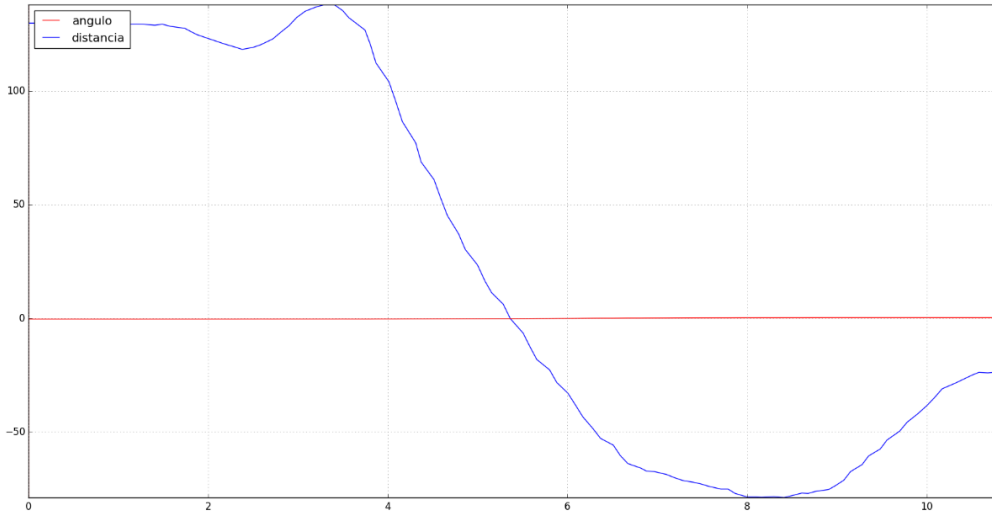


Figura 88 Comportamiento de la distancia frente al tiempo "Kd"=0.001

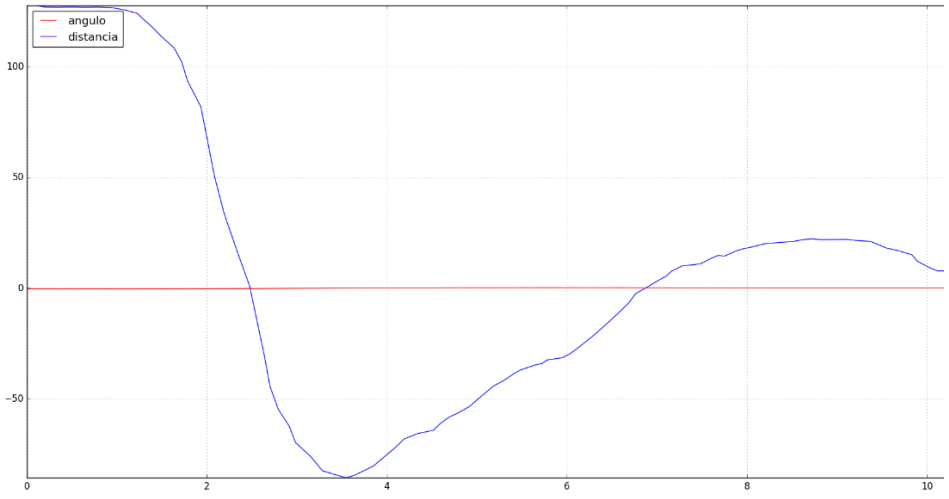


Figura 89 Comportamiento de la distancia frente al tiempo "Kd"=0.002

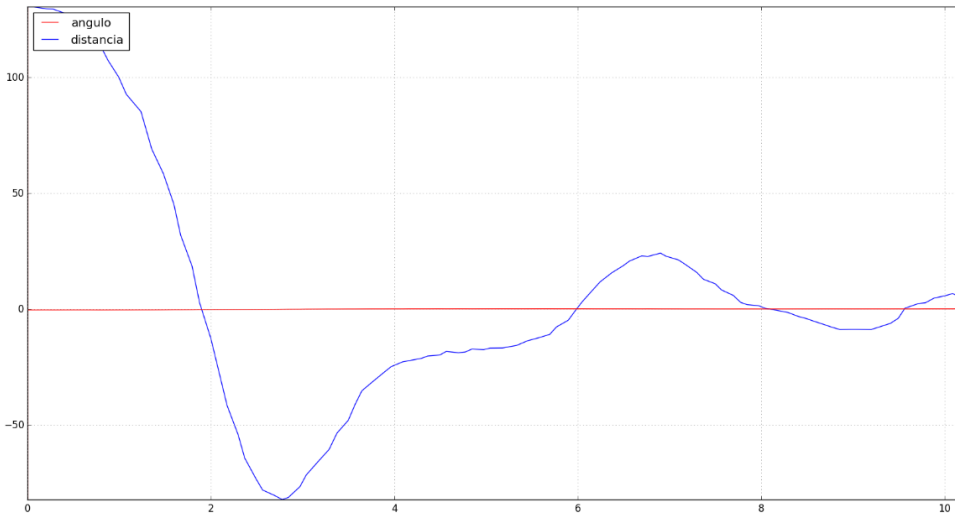


Figura 90 Comportamiento de la distancia frente al tiempo "Kd"=0.003

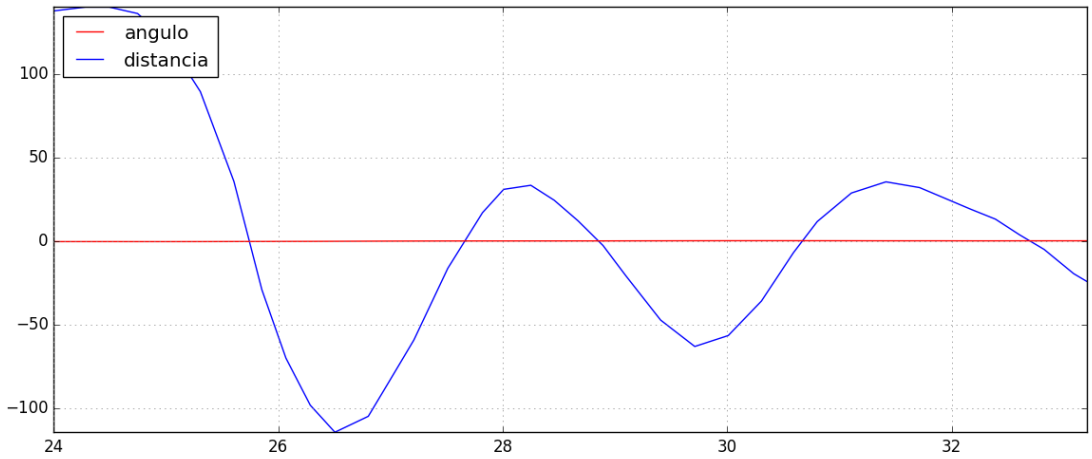


Figura 91 Comportamiento de la distancia frente al tiempo "Kd"=0.004

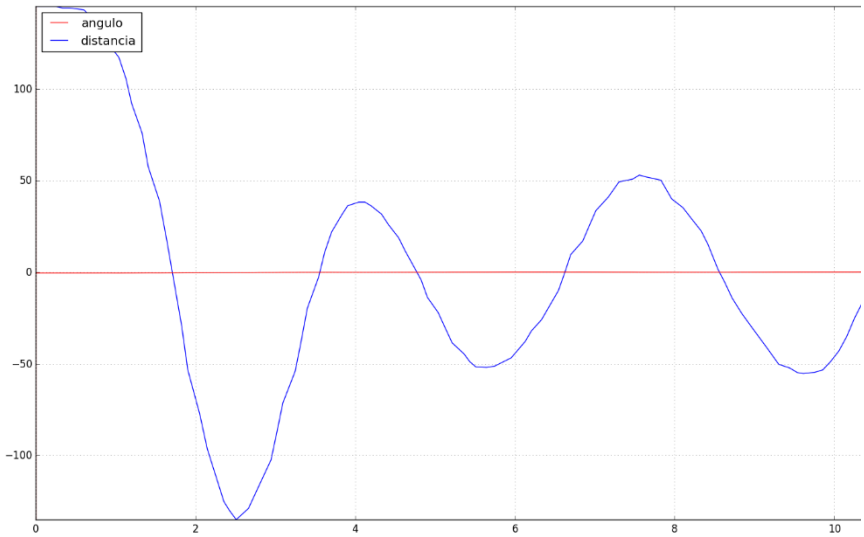


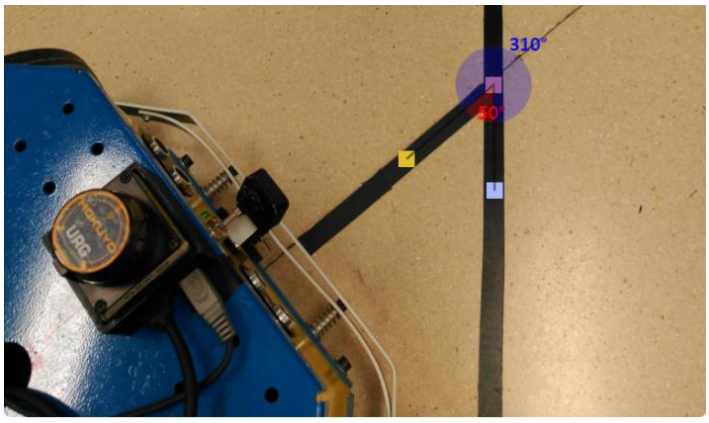
Figura 92 Comportamiento de la distancia frente al tiempo "Kd"=0.005

Tras la realización de estas pruebas, se determina que el valor más adecuado para “Kd” es de 0.003.

4.2.2 Ajuste del “Kg”

Este parámetro es la constante proporcional de ángulo, cuyo valor será multiplicado por el desvío de ángulo que sufra la línea respecto de la vertical y que influirá en un incremento o decremento del valor del giro.

El experimento, al igual que el anterior, consiste en introducir un escalón pero en este caso de ángulo (Figura 94 y Figura 93) en lugar de distancia y comprobar cómo es la respuesta para diferentes valores de “Kg”.



Ángulos: 50 ° - 310 °

Figura 94 Ángulo escalón

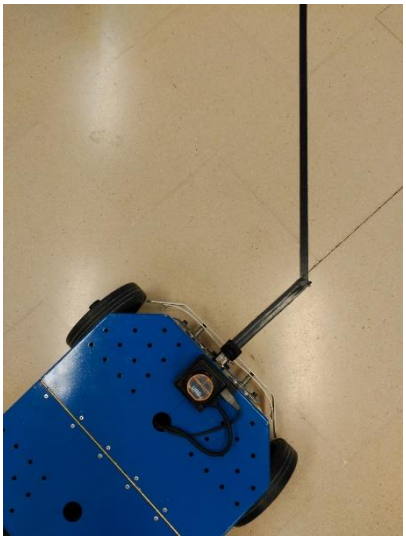


Figura 93 Trayectoria a seguir pruebas "Kg"

En la Tabla 11 se muestran los experimentos realizados.

Tabla 11 Pruebas sigue líneas para establecer "Kg"

Nº del experimento	Escalón (º)	Valor del "Kg"	Observaciones
1	50	-0.1	Gira demasiado y pierde la referencia
2	50	-0.6	Giro insuficiente. No logra seguir la trayectoria.
3	50	-0.5	Resultado bastante aceptable. Sin embargo, no alcanza bien el estacionario. Figura 95 Comportamiento del ángulo frente al tiempo "Kg"=-0.5Figura 95
4	50	-0.4	Respuesta más rápida que la anterior. Comportamiento muy fino. Figura 96
5	50	-0.3	Respuesta aún más rápida. Comportamiento excelente. Figura 97
6	50	-0.2	Se produce un ligero sobrepico, aunque alcanza bien el estacionario. Al encarar la recta el robot oscila un poco. Figura 98

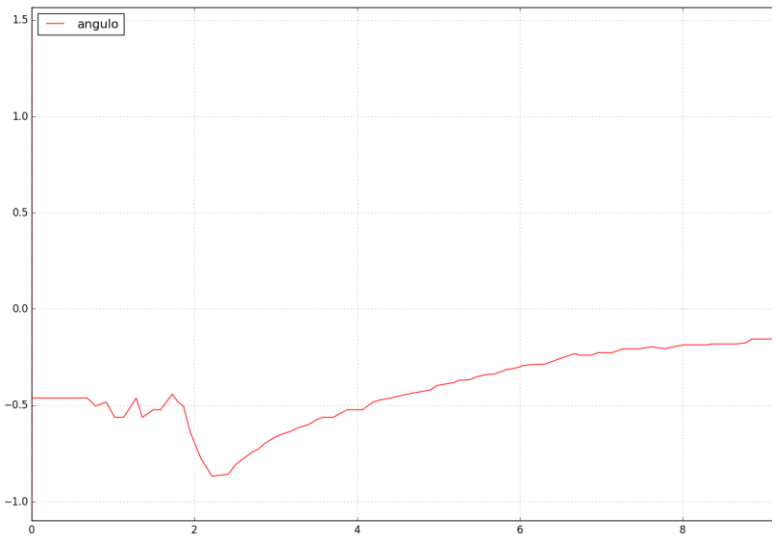


Figura 95 Comportamiento del ángulo frente al tiempo "Kg"=-0.5

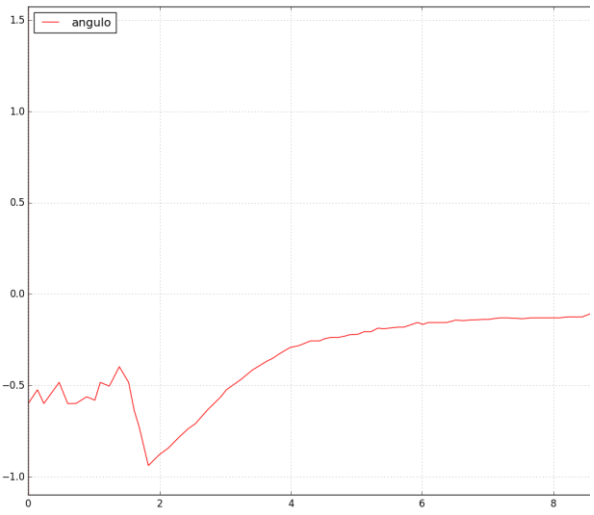


Figura 96 Comportamiento del ángulo frente al tiempo "Kg"=-0.4

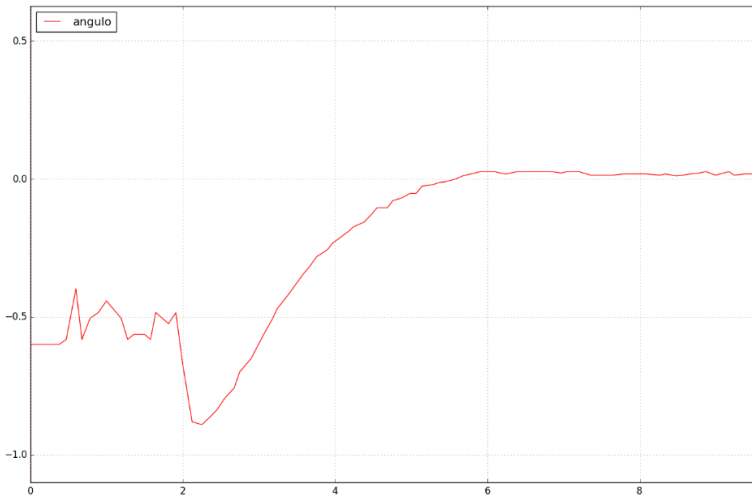


Figura 97 Comportamiento del ángulo frente al tiempo "Kg"=-0.3

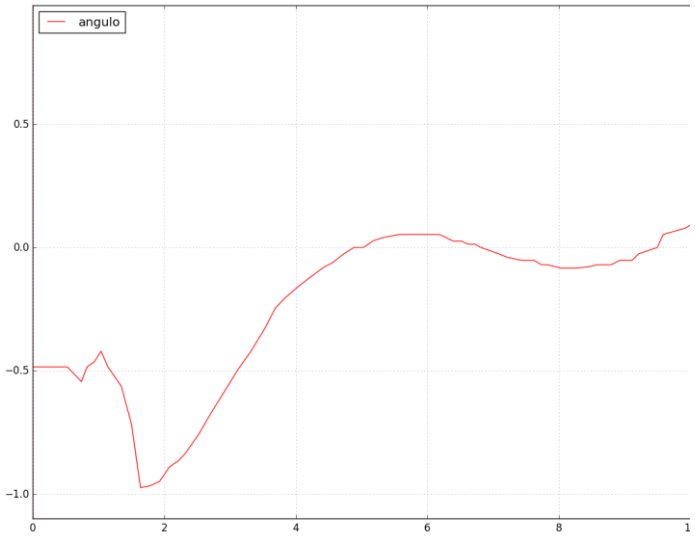


Figura 98 Comportamiento del ángulo frente al tiempo "Kg"=-0.2

Tras la realización de estas pruebas, se determina que el valor más adecuado para "Kg" es de -0.3.

4.3 STACK DE NAVEGACIÓN

Para probar los resultados de la navegación, lo primero que se debe hacer es crear un mapa. Para ello, se utiliza el nodo "gmapping", que se lanza por terminal utilizando el siguiente "roslaunch":

```
roslaunch edubot_navigation gmapping_demo.launch
```

Este ".launch" se encargará no solo de lanzar el paquete "gmapping" sino también de abrir la herramienta de visualización "rviz" para ir observando el proceso de creación. Su código puede verse en el Anexo D, en el apartado gmapping_demo.launch.

Para ir moviendo al robot por todo el entorno, se ha utilizado el módulo de teleoperación, que precisa de un teclado para enviar los comandos de velocidad lineal y angular. Su lanzamiento es muy sencillo, basta con ejecutar por terminal el siguiente comando:

```
roslaunch edubot_navigation edubot_teleop.launch
```

El código puede verse en el Anexo D.

En la Figura 99, Figura 100, Figura 101 y Figura 102 puede verse el proceso de creación del mapa.

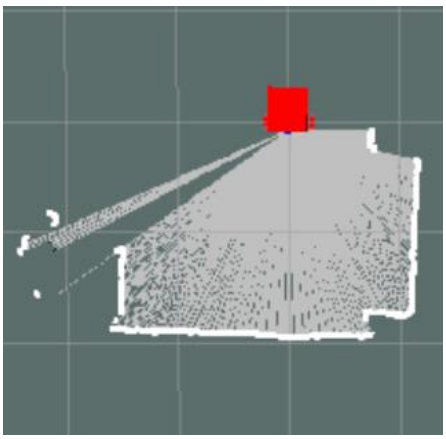


Figura 99 Proceso de creación de mapa (1)

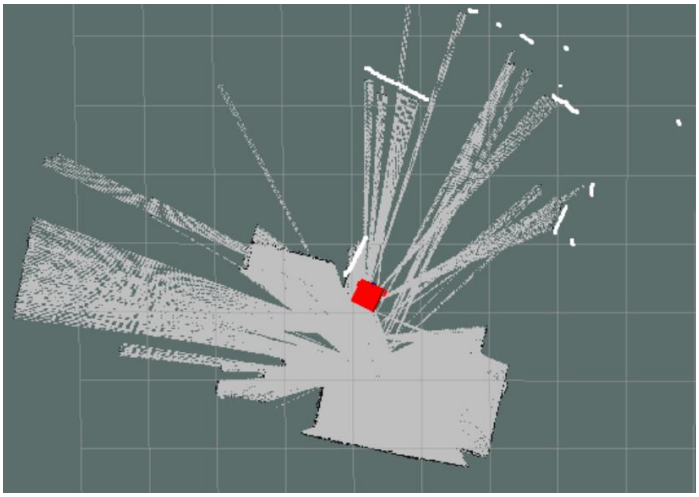


Figura 100 Proceso de creación de mapa (2)

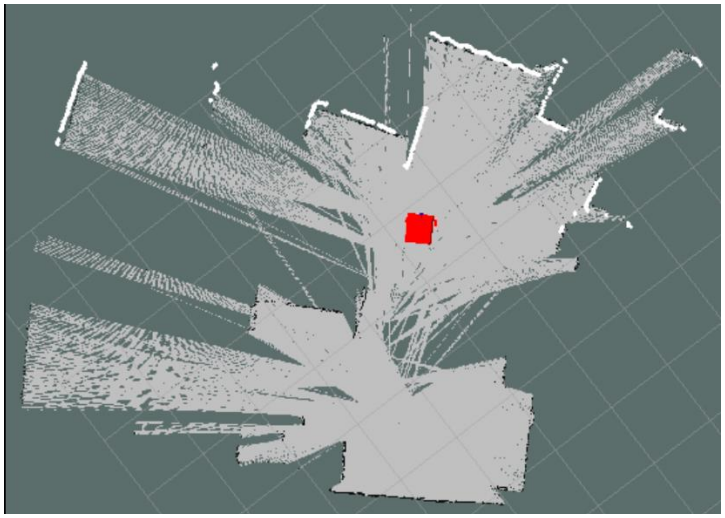


Figura 101 Proceso de creación de mapa (3)

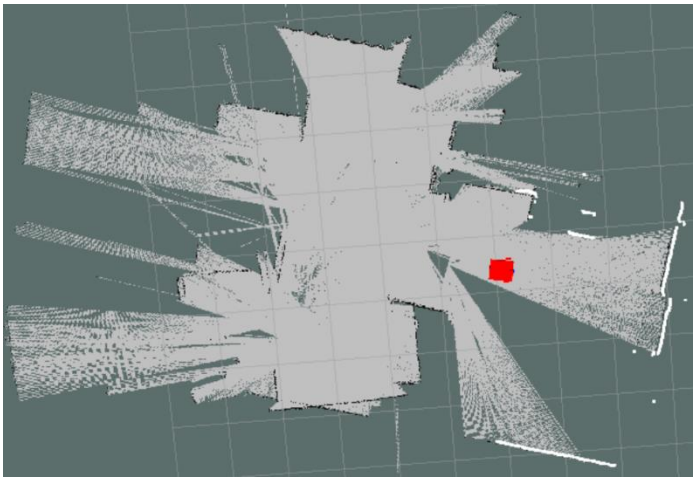


Figura 102 Proceso de creación de mapa (4)

Una vez cubierta toda la zona deseada, es el momento de guardar el mapa, y para ello, se utiliza el siguiente comando:

```
rosrun map_server map_saver -f
~/catkin_ws/src/edubot_navigation/maps/test_map
```

Indicándole la ruta donde se quiere guardar.

Este comando genera dos archivos: uno es la propia imagen del mapa en formato ".pgm" y el otro es un archivo de configuración de tipo ".yaml", donde se pueden establecer algunos parámetros como la resolución, el origen, la ruta donde ese encuentra la imagen, etc. El resultado final puede verse en la Figura 103.

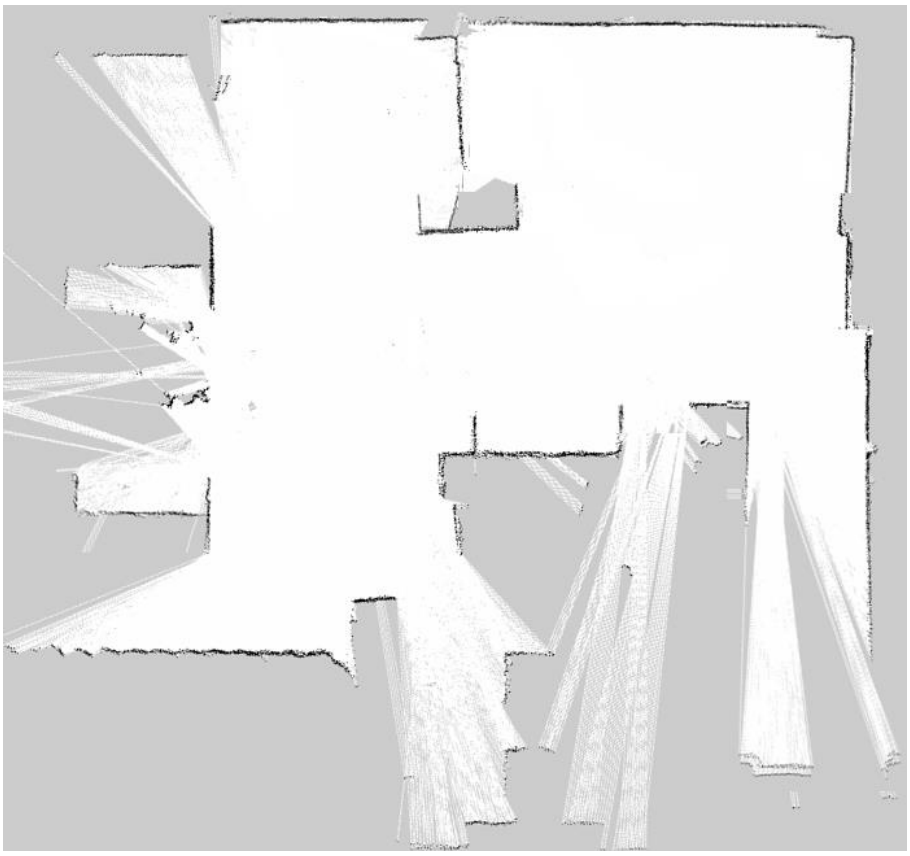


Figura 103 Plano definitivo

Una vez construido el mapa, ya se puede comenzar a configurar el stack de navegación para navegar.

Es necesaria la configuración de cuatro ficheros para el correcto funcionamiento. Tres de ellos correspondientes a los "Costmaps" y el otro referente al "Move base" [7].

- Costmap_common_params.yaml: se establecen parámetros como el rango de obstáculos, las medidas del robot, el radio imaginario libre de obstáculos, el frame del láser, etc.

- Global_costmap_params.yaml: Contiene parámetros como la resolución, la frecuencia de publicación y de refresco y, lo más importante, el frame de la odometría y del robot.
- Local_costmap_params.yaml: Ajusta los mismos parámetros que el anterior pero esta vez para el mapa de costes local.
- Base_local_planner_params.yaml: Finalmente, este archivo de configuración contiene parámetros referentes a las velocidades y aceleraciones del robot a la hora de generar las trayectorias.

Estos archivos pueden verse en el Anexo D.

El funcionamiento general del "Stack de Navegación" es el que se muestra en la Figura 104.

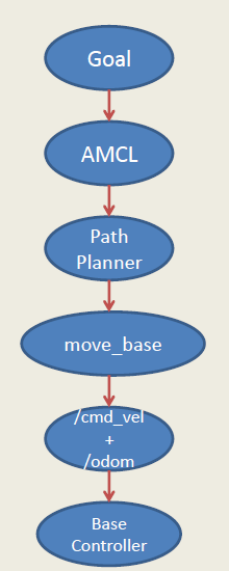


Figura 104 Principales pasos de Navegación

Como se explicó en el apartado 3.8, son necesarias dos configuraciones imprescindibles para el correcto funcionamiento del "Stack de Navegación". Por una parte, una estructura muy concreta en cuanto a los sistemas de referencia ("TF") y, por otra parte, la organización del módulo "move_base" (Figura 53), que es el encargado de publicar las velocidades para seguir las trayectorias.

En la Figura 105 se muestra el resultado conseguido de la relación de los sistemas de referencia del robot.

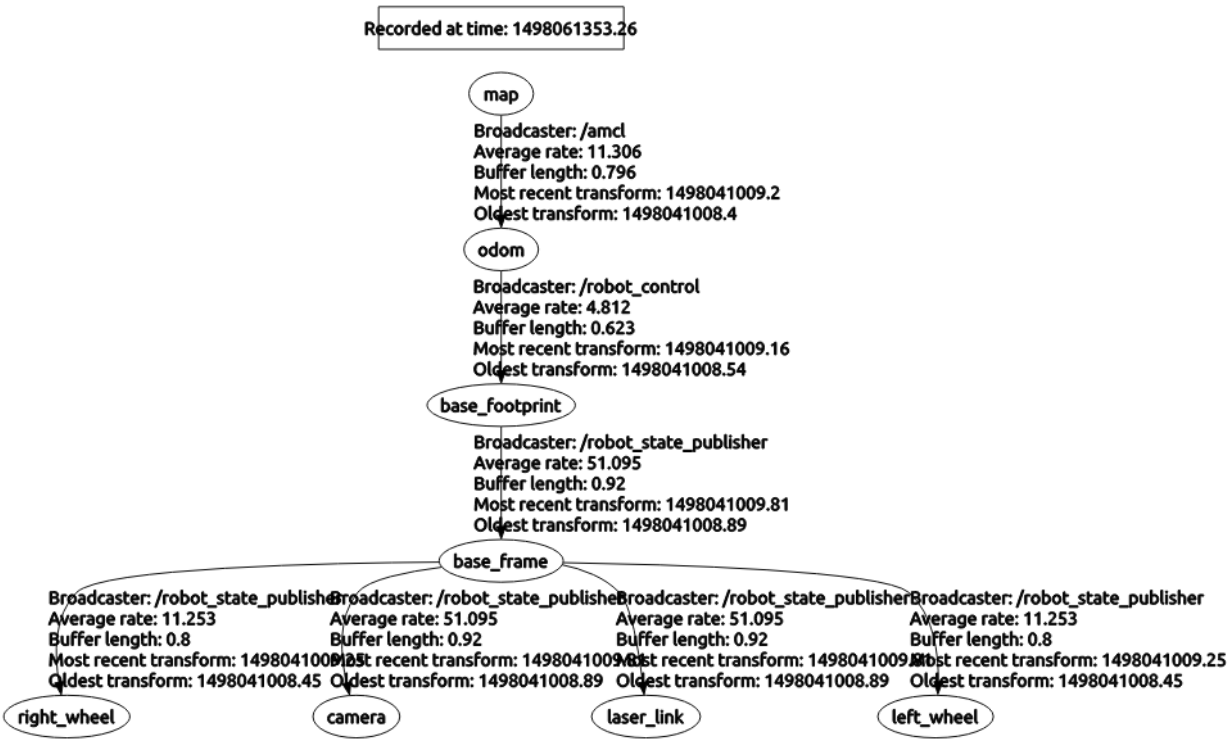


Figura 105 Estructura Árbol de transformaciones

Y, además, en la Figura 106 se muestra la configuración del "move base" conseguida, de modo que se suscriba a todos los "topics" que necesita y publique las velocidades correspondientes:

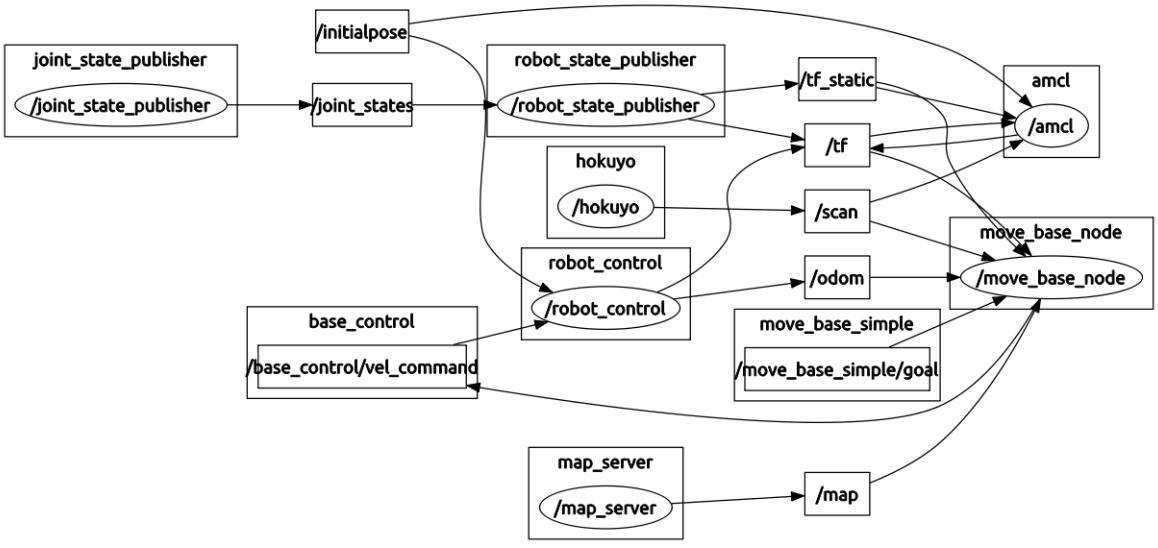


Figura 106 Organización del "move_base"

Después de configurar todo lo referente al “Stack de Navegación”, se puede comenzar la navegación. Se debe lanzar el nodo de localización “ACML”. Para ello se ha creado el lanzador siguiente:

```
roslaunch edubot_navigation amcl_demo.launch
```

A continuación, se abrirá la herramienta “rviz” con la que se podrá indicar la posición inicial del robot estimada sobre el mapa y enviar puntos objetivos de destino. Para más información, ver apartado 4.3.

Finalmente, en la Figura 108 y Figura 107 se muestra una prueba de los resultados conseguidos, indicándole al robot una posición y orientación objetivos y observando como la alcanza.

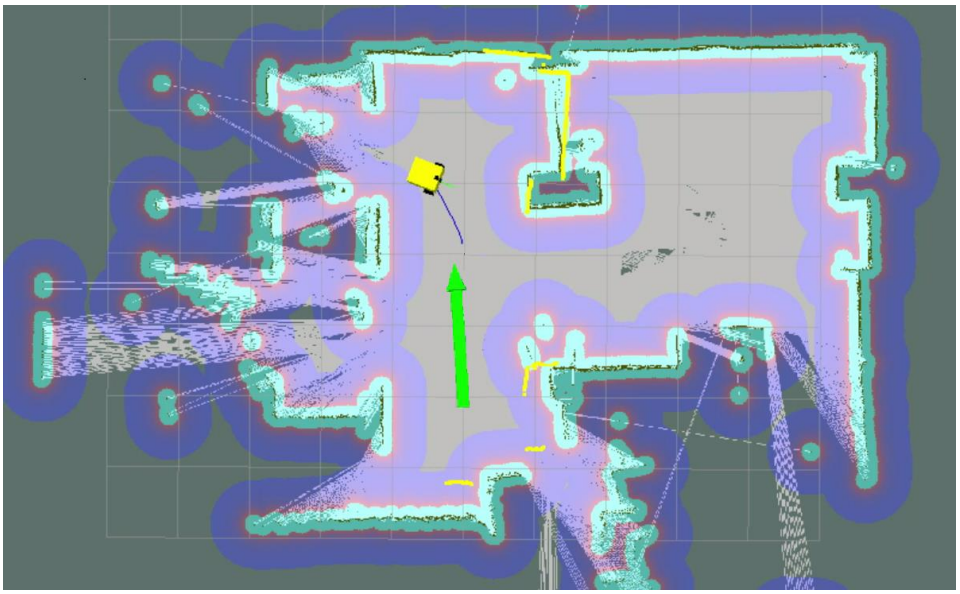


Figura 108 Envío de posición y orientación objetivo

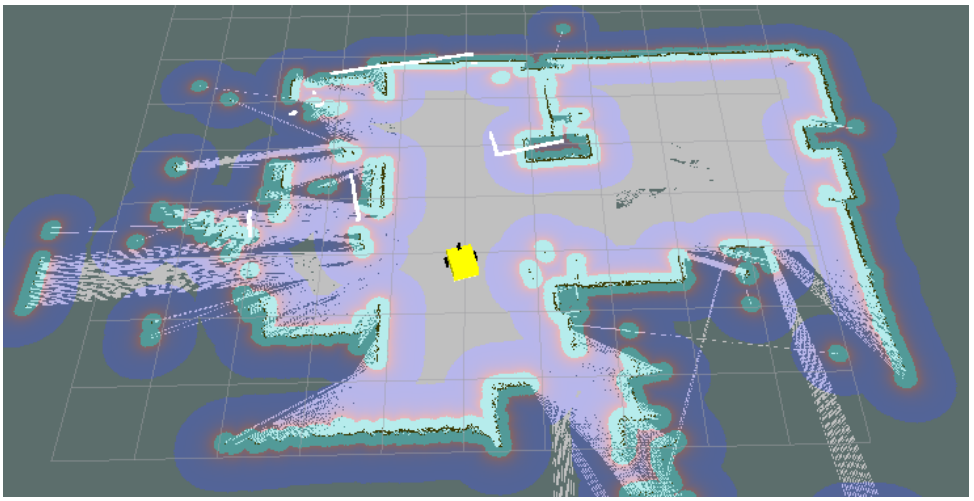


Figura 107 Alcanzando posición y orientación objetivo

CAPÍTULO 5. MANUAL DE USUARIO

En este capítulo se pretende desarrollar un pequeño manual de usuario, una guía básica para que cualquier usuario sea capaz de ejecutar las aplicaciones del robot sin ninguna dificultad.

5.1 ARRANQUE Y CONEXIÓN REMOTA

Lo primero de todo será arrancar la Raspberry y el ordenador que se vaya a utilizar para su manejo. Para encoder la Raspberry basta con accionar el interruptor del robot y automáticamente comenzará a iniciarse.

La única condición que deben cumplir ambos dispositivos es estar conectados a la misma red WIFI. La Raspberry está predefinida para conectarse a "DISA-PC", red del laboratorio de automática diseñada para estas aplicaciones. No obstante, podría cambiarse fácilmente esta red a través del interfaz gráfico de Ubuntu utilizando la salida HDMI y conectándola a un monitor.

Una vez ambas máquinas están encendidas y conectadas a la misma red, desde el ordenador se establecerá una conexión de tipo SSH con la Raspberry introduciendo el siguiente comando:

```
ssh edubot@192.168.19.66
```

Y, a continuación, nos pedirá la contraseña de usuario.

Una vez establecida la conexión, el siguiente paso será ejecutar el nodo principal "robot_control". Este módulo contiene toda la información del estado del robot, se encarga de calcular su odometría y se suscribe a los topics necesarios de velocidad y comandos. Además, lanza el "roscore" o máster, que como ya se explicó en el apartado 3.2.2.3, será el encargado de establecer la comunicación entre los diferentes nodos que se ejecuten en el sistema. Para su lanzamiento, se ejecuta el siguiente comando:

```
roslaunch robot_control edubot.launch
```

En un nuevo terminal, se realizará la conexión del ordenador con el máster que ha ejecutado la Raspberry. Para ello, se le debe indicar en que dirección y puerto se encuentra el máster y, además, exportar la dirección IP del ordenador para que la reconozca la Raspberry. Los comandos a ejecutar son los que siguen:

```
export ROS_MASTER_URI=http://192.168.192.66:11311

export ROS_IP=192.168.192.178
```

Para comprobar si la conexión se ha realizado correctamente, es conveniente ejecutar el comando “rostopic list” que mostrará por pantalla los topics del módulo “robot_control” si la conexión ha sido establecida.

5.2 MOVIMIENTO EDUBOT

Para este apartado es necesario haber realizado los pasos del epígrafe 5.1 ARRANQUE Y CONEXIÓN REMOTA.

Existen multitud de formas de hacer mover al robot, basta con pensar en las posibilidades que hay de publicar un topic. No obstante, en este apartado se explicarán 3 métodos muy sencillos y fáciles de manejar.

5.2.1 Joystick ordenador

Como se explicó en el apartado 3.5, se implementó un código web que proporciona una aplicación joystick, la cual publica comandos de velocidad con el topic *“/base_control/vel_command”* adecuado para la comunicación con el robot.

Es necesario establecer la dirección IP donde se ubica el máster. Para ello, se ha de editar el archivo “index.html” y en la línea 194 se ubica el comando “*ros.connect*”. Se deberá especificar:

```
ros.connect('ws://192.168.192.66:9090');
```

Para ejecutarlo, solamente es necesario acceder a la ruta que contenga el archivo, en este caso “~/catkin_ws/src/www_joy” y hacer doble click sobre el archivo “index”. Entonces se abrirá una página web mostrando un “joystick” y aparecerá un mensaje diciendo si hay o no conexión con el robot, como se muestra en la Figura 109.

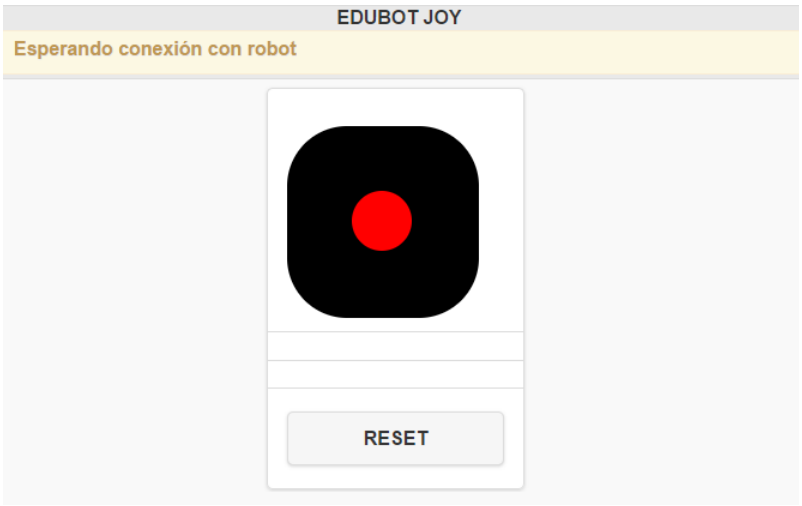


Figura 109 Joystick html

5.2.2 Android app

En la tienda de aplicaciones de Android "PlayStore" existen numerosas aplicaciones para trabajar con ROS, si bien es cierto que no hay ninguna que esté perfectamente desarrollada ni que sea oficial.

Tras trabajar con varias de ellas, se recomienda instalar la que lleva por nombre "ROS Control", que tiene el aspecto que muestra la Figura 110.

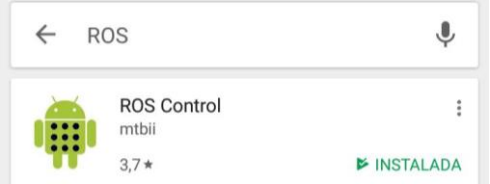


Figura 110 ROS App

Esta aplicación cuenta con numerosas funciones (Figura 111) como visualización de cámara, de mapas, las nubes de puntos de un láser, incluso permite gestionar varios robots. No obstante, la que interesa para este apartado es la del "joystick" que permitirá mover el robot de manera cómoda y sencilla.

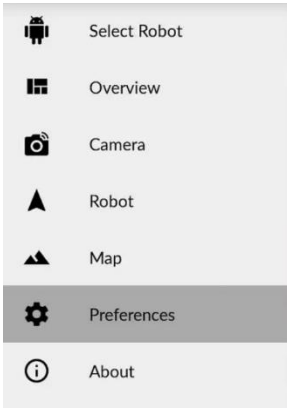


Figura 111 Funciones de la aplicación

Sin embargo, es necesario hacer unas pequeñas configuraciones en la pestaña "Preferences", para establecer la dirección del Máster y el nombre de los "topics" con los que debe publicar y suscribirse, para la correcta comunicación con Edubot (Figura 112).

Figura 112 Configuración de los topics

En la Figura 113 puede verse un ejemplo de vista previa tanto del láser y la nube de puntos como del joystick en la parte inferior derecha.

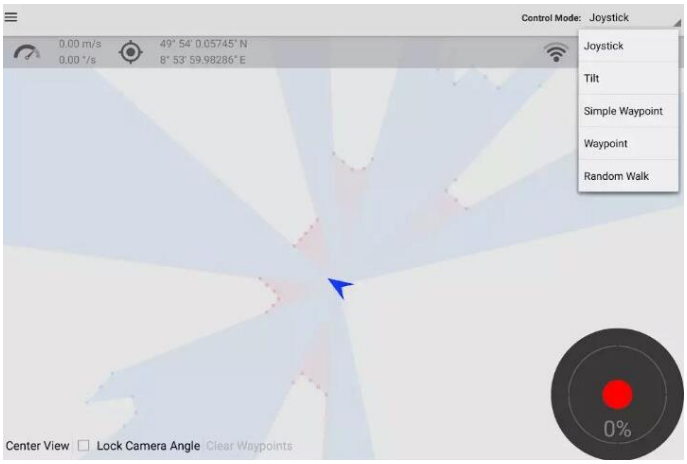


Figura 113 Vista previa de la aplicación

5.2.3 Edubot_teleop

Por último, se ha desarrollado un nodo de ROS que permite el manejo y control del robot desde el teclado del ordenador. Para ello, se ha aprovechado la herramienta de ROS "turtlebot", que se trata de un módulo que desarrolla un pequeño robot con forma de tortuga en el cual se encuentran desarrolladas numerosas herramientas, entre ellas, "turtlebot_teleop".

A partir de ella, se han modificado algunos parámetros para adaptarla a Edubot, en especial, el topic con el que publicaba la velocidad que ha pasado a llamarse "/base_control/vel_command".

Para su lanzamiento, solamente se necesita ejecutar el comando:

```
roslaunch edubot_navigation edubot_teleop.launch
```

Las instrucciones de uso son las siguientes:

- Teclas “u”/ ”i”/ “o” → avanza izquierda / recto / derecha
- Teclas “n”/ “m”/ “,” → retrocede izquierda / recto / derecha
- Teclas “w”/ “x” → aumenta/disminuye la velocidad lineal
- Teclas “e” / ”c” → aumenta/disminuye la velocidad angular

5.3 LANZAMIENTO DEL NODO SIGUE-LÍNEAS

Para la ejecución de este nodo es necesario el previo lanzamiento del nodo “robot_control” que se realizó en el apartado 5.1 pero de una forma algo diferente. Lo único que cambia es que en lugar de lanzar el “launch” con el nombre de “edubot.launch” hay que lanzar otro que se llama “edubot_camara.launch”. La diferencia es que este además de lanzar todo el control de Edubot, toma el frame de la cámara y lo publica en un topic, para que pueda ser tomado por el nodo sigue-líneas. De forma que el comando previo a ejecutar será:

```
roslaunch robot_control edubot_camara.launch
```

De este modo, Edubot ya estará publicando la imagen de la cámara y suscrito al topic de velocidad esperando comandos.

A continuación, hay que ejecutar el nodo que contiene el algoritmo de seguir la línea. Para ello, se debe escribir el siguiente comando:

```
roslaunch camara camara_linea.launch
```

Entonces comenzará a recibir la imagen, a tratarla y a publicar los comandos de velocidad adecuados para el seguimiento de la línea. Además, se abrirán tres pestañas en las que se podrá visualizar la imagen original, la imagen binarizada y la imagen con el algoritmo de detección de línea, como mostraban la Figura 44, Figura 45 y Figura 46 del apartado 3.4.

5.4 LANZAMIENTO DEL STACK DE NAVEGACIÓN

Para este apartado es necesario haber realizado los pasos del epígrafe 5.1 ARRANQUE Y CONEXIÓN REMOTA.

Se distinguen dos casos prácticos, como pueden ser, la creación del mapa o la navegación por el mismo. Si no se tiene creado ningún mapa realice primero el apartado 5.4.1, y si ya lo tiene creado, puede pasar directamente al apartado 5.4.2.

No obstante, para ambos casos, es necesario la conexión del láser y su configuración, ya que se necesitan dar permisos de puertos. Para ello, se debe ejecutar por terminal el siguiente comando:

```
sudo chmod a+rw /dev/ttyACM0
```

De esta forma se habilita el puerto para que pueda empezar a publicar información el láser. A continuación, se ejecuta el nodo del "Hokuyo", introduciendo el siguiente comando por terminal:

```
rosrun hokuyo_node hokuyo_node
```

Si todo está correcto, se deberá ver por terminal la información "Streaming data".

5.4.1 Crear mapa

Una vez configurado lo anterior, para comenzar a crear un mapa se debe ejecutar el módulo "gmapping". Para ello, se introducirá por terminal el siguiente comando:

```
roslaunch edubot_navigation gmapping_demo.launch
```

Este ".launch" se encargará no solo de lanzar el paquete "gmapping" sino también de abrir la herramienta de visualización "rviz" para ir observando el proceso de creación.

Para ir moviendo al robot por todo el entorno, se puede hacer con cualquiera de las posibilidades vistas en el apartado 5.2, o bien, manualmente. Se aconseja utilizar la opción de 5.2.3 Edubot_teleop, para moverlo de forma más precisa.

No obstante, se implementó una opción muy interesante. Se trata de la creación de un mapa de forma autónoma, bien utilizando los ultrasonidos para la navegación, o bien, el láser. Con esta filosofía, se ha desarrollado un algoritmo de navegación muy sencillo a partir de la nube de puntos que ofrece el láser o los ultrasonidos, de modo que el robot se va moviendo y a medida que detecta objetos gira y continúa moviéndose, similar al de los aspiradores modernos. De esta forma, el robot sería capaz de ir generando un mapa de forma autónoma, sin necesidad de un movimiento manual por el entorno.

Una vez barrida toda la zona deseada, es el momento de guardar el mapa, y para ello, se utiliza el siguiente comando:

```
rosrun map_server map_saver -f
~/catkin_ws/src/edubot_navigation/maps/test_map
```

Indicándole la ruta donde se quiere guardar (es aconsejable poner la que está arriba de ejemplo).

5.4.2 Cargar mapa y navegar

Finalmente, para navegar por el mapa, lo primero que se debe hacer es cargarlo y abrir la herramienta de visualización "rviz". Para ello, se ha creado un fichero ".launch" encargado de ello. Para ejecutarlo, se debe escribir el siguiente comando:

```
Roslaunch edubot_navigation amcl_demo.launch
```

A continuación, se abrirá la herramienta "rviz" con la que se podrá indicar la posición inicial del robot estimada sobre el mapa y enviar puntos objetivos de destino. Para más información, ver apartado 4.3.

CAPÍTULO 6. ESTUDIO ECONÓMICO

En este capítulo se llevará a cabo una estimación del coste del proyecto al mismo tiempo que una enumeración de los materiales empleados. Se agruparán los costes según el origen y serán divididos en costes directos e indirectos. Finalmente, se detallará el presupuesto final.

6.1 Costes directos

Los costes directos son aquellos relacionados con un producto o un servicio, por lo que en este apartado se incluyen los costes de material, construcción, recursos y herramientas empleadas.

6.1.1 Costes asociados al equipo

Dentro de este apartado se distinguen: costes de material, de herramientas y equipo y de software

Costes de Material

Los costes del material necesario para la construcción del robot pueden verse en la Tabla 12, Tabla 13 y Tabla 14. Finalmente, en la Tabla 15 se realiza un resumen de todos los costes de material necesario.

Tabla 12 Coste de los elementos mecánicos del robot

Elementos Mecánicos				
Concepto	Procedencia	Coste unitario (€)	Unidades	Coste total (€)
Base del chasis	Talleres Tejedor	20.00	1	20.00
Tapa abisagrada	Talleres Tejedor	15.00	2	30.00
Paredes del chasis	Talleres Tejedor	20.00	2	40.00
Buje de rueda	Talleres Tejedor	45.00	2	90.00
Rodamientos rueda SFK 6003 12,DE,NC	Rodamientos Palencia	13.33	2	26.66
Rueda 170x32	Ruedas ALEX	22.60	2	45.20
Ruedas locas Caster esfera 40	Ruedas ALEX	4.50	2	9.00
Bumpers	Service Robotics	35.80	2	71.60
Mecanismo paragolpes	Service Robotics	2.50	4	10.00
Ángulo protección	Service Robotics	3.00	1	3.00
Cierres tapa	Ferretería Pablo Llamas	1.80	2	3.60
ISO 7380 -M6 X 16	Ferretería Pablo Llamas	0.15	34	5.10
DIN 912 M4 x 20	Ferretería Pablo Llamas	0.16	6	0.96
DIN 912 M5 x 10	Ferretería Pablo Llamas	0.15	4	0.60
DIN 7991 M5 x 20	Ferretería Pablo Llamas	0.15	4	0.60
DIN EN ISO 10511 M6	Ferretería Pablo Llamas	0.10	1	0.10
Autorroscante DIN 7982 - 9 x 9.5	Ferretería Pablo Llamas	0.05	2	0.10
Arandela DIN 433 - 5.3	Ferretería Pablo Llamas	0.02	4	0.08
Arandela ISO 7090 - 8	Ferretería Pablo Llamas	0.02	8	0.16
Remacha M5 x 10	Ferretería Pablo Llamas	0.02	4	0.08
			Total	356.84

Tabla 13 Coste de los elementos eléctricos del robot

Elementos Eléctricos				
Concepto	Procedencia	Coste unitario (€)	Unidades	Coste total (€)
Batería Yasa 14AHR 12V PB	Solo Recambios	42.00	1	42.00
Conector de carga XLR Base	Electrosón Castilla	7.50	1	7.50
Portafusibles 4x4	Soto Recambios	19.91	1	19.91
Canaleta 25x25 Bornes	Electrosón Castilla	0.80	1	0.80
Borna 3 conectores	Electrosón Castilla	0.50	6	3.00
Interruptor general	Electrosón Castilla	1.85	1	1.85
Cableado	Electrosón Castilla	0.71	4	2.84
			Total	77.90

Tabla 14 Coste de los elementos electrónicos del robot

Elementos Electrónicos				
Concepto	Procedencia	Coste unitario (€)	Unidades	Coste total (€)
Raspberry Pi 3	Farnell España	32,59	1	32.59
PicoBorg Reverse	PiBorg	32.00	2	64.00
Ultraborg	PiBorg	15.99	2	31.98
Motor DC-x42x550	Superdroid Robots	54,00	2	108,00
Encoder 12PPR 42 x 15	Superdroid Robots	32,60	2	65,20
Regulador tensión	Centro Tecnológico CARTIF	15.00	1	15.00
Sonar TS-601Audiowell	Audiowell Electronic	7,00	8	56,00
Interruptor TactSMD Bumper	Electrosón Catilla	0,50	4	2,00
Cable plano múltiples vías	Electrosón Catilla	2,90	3	8,70
Cámara Raspberry	Farnell España	16.31	1	16.31
Láser	Robotshop	1680,00	1	1.680,00
			Total	1.919,90

Tabla 15 Resumen costes del material

Coste Material del Robot		
Concepto		Coste (€)
Elementos mecánicos		356.84
Elementos eléctricos		77.90
Elementos electrónicos		1.919,90
	Suma	2.354,64
	IVA 21%	494,47
	Total	2.849,11

Costes de herramientas y equipo

En este apartado se incluyen los costes derivados del uso de herramientas y el equipo necesario para la construcción física del robot y la implementación del sistema de control. Se detallan en la Tabla 16.

Tabla 16 Costes de herramientas y equipo

Coste de herramientas y equipo utilizado			
Concepto	Coste unitario (€/h)	Unidades (horas)	Coste total (€)
Banco de trabajo (herramientas varias)	2,32	10	23,20
Material de laboratorio (polímetro, soldador, etc)	0,75	20	15,00
Dispositivos electrónicos (fuente de alimentación, osciloscopio, generador de señal)	0,93	20	18,60
Ordenador de sobremesa	0,29	600	174,00
		Total	230,80

Costes de software

En este apartado se detallan los software utilizados a lo largo del proceso. No obstante, cabe destacar que todos ellos son de libre distribución y que, por tanto, el coste total será cero. Se puede ver una descripción más detallada en Tabla 17, además de un número aproximado de horas de utilización de cada uno de ellos.

Tabla 17 Costes de software

Coste del software empleado			
Concepto	Coste unitario (€/h)	Unidades(h)	Coste total (€)
Ubuntu	0,00	600	0,00
Raspbian	0,00	200	0,00
ROS	0,00	180	0,00
		Total	0,00

Resumen de costes asociados al equipo

En la Tabla 18 pueden verse los costes generales asociados a cada apartado y su suma total.

Tabla 18 Suma de costes asociados al equipo

Resumen de costes asociados al equipo		
Concepto		Importe (€)
Coste material		2.849,11
Herramientas y equipo utilizado		230,80
Software empleado		0,00
	Total	3.079,91

6.1.2 Coste de la mano de obra

En este apartado se van a estimar los costes asociados a la mano de obra necesarios para la construcción del robot y la puesta en marcha del mismo. Los costes del montaje de los diferentes componentes mecánicos, eléctricos y electrónicos pueden ser abaratados si este trabajo lo realiza un titulado en formación profesional, cuya mano de obra es más barata que la de un ingeniero.

En primer lugar, se calcularán las horas de trabajo que hay en un año y, de esta manera se podrá estimar el suelo por horas y ajustar el pago al tiempo que ha sido necesario para realizar cada parte del trabajo (Tabla 19).

Tabla 19 Cálculo de las horas laborables en un año

Coste de la mano de obra por tiempo empleado		
Concepto		Tiempo (días)
Año medio		365
Vacaciones		22
Días festivos		14
Días perdidos (aprox.)		5
Fines de semana		104
	Total	220
	x 8 horas/día	1760 h laborables/año

Una vez calculadas las horas laborables por un año se tasará el sueldo que cobran un ingeniero y un técnico superior de formación profesional a la hora, teniendo en cuenta el sueldo medio de estos profesionales en España con un nivel medio de experiencia. Este cálculo puede verse en la Tabla 20 y Tabla 21.

Tabla 20 Cálculo del sueldo por horas de un ingeniero

Cálculo del sueldo de un ingeniero por hora		
Concepto		Importe (€)
Sueldo bruto e incentivos		27.600,00
Sueldo neto e incentivos		25.806,00
S.S. a cargo de la empresa		8.197,00
	Sueldo Total	35.797,00
	Sueldo por hora	20,34

Tabla 21 Cálculo del sueldo por hora de un técnico

Cálculo del sueldo de un técnico por hora		
Concepto		Importe (€)
Sueldo bruto e incentivos		17500,00
Sueldo neto e incentivos		16362,50
S.S. a cargo de la empresa		5197,50
	Sueldo Total	22697,50
	Sueldo por hora	12.90

Una vez conocido el sueldo por horas se estimarán las horas necesarias que ha de realizar cada trabajador y con ello se estimarán los costes totales de personal.

El tiempo de desarrollo del proyecto ha sido de cinco meses. Antes se ha calculado que en un año las horas laborables son 1760, por lo que las horas laborables en 5 meses trabajando 8 horas diarias serán 740.

Por otro lado, el tiempo de montaje del robot se estima que son unas 16 horas, por lo que el coste total de personal se refleja en la Tabla 22.

Tabla 22 Costes de personal

Costes de personal			
Concepto	Coste unitario (€)	Unidades	Coste Total (€)
Construcción del robot	12.90	16.00	206.40
Desarrollo del proyecto	20.34	740.00	15.254.40
		Total	15.460.80

6.1.3 Otros costes directos

Aquí se incluyen aquellos costes que no pueden atribuirse a elementos o herramientas ni a costes de personal, pero que también son costes directos. Se refiere, por ejemplo, a los gastos del material de oficina como folios, fotocopias, etc. y de material de laboratorio, como cinta adhesiva, estaño, etc. También se incluirán los costes para la documentación de este proyecto. Todos estos costes pueden verse en la Tabla 23.

Tabla 23 Otros costes directos

Otros costes directos	
Concepto	Importe (€)
Material de laboratorio	50
Material de oficina	30
Documentación del proyecto	60
Total	140

6.1.4 Total costes directos

Puede verse un resumen de los gastos que suponen los costes totales en la Tabla 24.

Tabla 24 Total de costes directos

Total de costes directos	
Concepto	Importe (€)
Costes asociados al equipo	3.079,91
Coste de personal directo	15.460.80
Otros costes	140.00
Total	18.680,71

6.2 Costes indirectos

Los costes indirectos son aquellos que corresponden a recursos consumidos durante la realización del proyecto pero que no pueden asociarse directamente a la realización de este. Se incluyen, por ejemplo, los gastos de electricidad, agua, climatización, etc. En la Tabla 25 se muestra un cálculo estimado.

Tabla 25 Total de costes indirectos

Costes Indirectos		
Concepto		Importe (€)
Consumo eléctrico de herramientas y equipos		45.00
Otros consumos eléctricos (calefacción, iluminación, etc.)		75.00
	Total	120.00

6.3 Coste total del proyecto

Finalmente, en la Tabla 26 se realiza un resumen del total de costes que supone la realización del proyecto, tanto directos como indirectos.

Tabla 26 Coste total del proyecto

Coste total del proyecto		
Concepto		Importe (€)
Costes directos		18.680,71
Costes indirectos		120.00
	Total	18.800,71

CAPÍTULO 7. CONCLUSIONES

Este capítulo contiene las conclusiones que se han obtenido tras la realización del proyecto, además de una serie de líneas de mejora, que puedan ser útiles en proyectos futuros sobre la plataforma de Edubot.

Lo primero y más importante, es que se han logrado superar con éxito todos y cada uno de los objetivos que se marcaron inicialmente. Esto es algo que resulta significativo, teniendo en cuenta la amplitud de los objetivos y la gran cantidad de técnicas y conocimientos que ha sido necesario aplicar.

Se ha realizado la integración del hardware, y el conexionado para el control de un robot móvil. Una de las especificaciones iniciales era desarrolloar un nuevo robot que permitiera el control de bajo nivel con elementos comerciales, que en este caso fueron, las “PicoBorg” y “Ultraborg” e integrarlo en una arquitectura de alto nivel basada en ROS (Robot Operating System). En el control de alto nivel, se introdujo una “Raspberry Pi 3” pequeño y potente controlador, que ha dotado al robot de una consistencia y robustez de las que anteriormente carecía.

En cuanto al software, se ha realizado el control de bajo nivel tanto en los microcontroladores encargados de los motores como en la Raspberry. En los primeros se ha llevado a cabo un control en lazo cerrado para el ajuste de la velocidad de los motores, además de sistemas de seguridad en caso de perder las referencias de los codificadores incrementales. Además, la Raspberry se encarga de monitorizar el estado del robot, velocidad, posición, nivel de carga, lecturas de los ultrasonidos y gestión de las alarmas.

El proceso de calibración de la odometría se ha basado en varios experimentos y pruebas iterativas, mediante el ajuste de ciertos parámetros como el radio de las ruedas y la separación entre ellas, además de la lectura de valores como los pulsos de los codificaodres incrementales y las distancias recorridas tanto teóricas como experimentales.

En la Raspberry también se ha llevado a cabo el control de alto nivel. Uno de los principales objetivos se centraba en implementar el sistema robótico ROS en la plataforma. Ha sido una de las tareas que, sin duda, más tiempo ha llevado, pero que, ha dotado a la plataforma de una serie de características que hacen mucho más simple su automatización.

Se ha realizado un modelado URDF del robot y una descripción del mismo. Se han ajustado y localizado cada uno de los sistemas de referencia del robot como, por ejemplo, el sistema de referencia global o mundo, la odometría, el propio robot, las ruedas, el láser y la cámara. Con las herramientas gráficas que propociona ROS, se ha podido simular la plataforma.

Gracias no solo a ROS sino también a la claridad del hardware implementado, ha resultado muy cómodo y sencillo añadir diferentes sensores a la plataforma, en especial, los ultrasonidos, la cámara y el láser. Gracias a ellos, se han podido desarrollar aplicaciones como la de seguir líneas, navegar de forma autónoma o, incluso, la teleoperación del robot desde diferentes dispositivos, como pueden ser, un joystick, un ordenador o un smartphone.

La funcionalidad de seguir líneas ha sido desarrollada gracias a la imagen captada de la cámara, al tratamiento de la misma, a los algoritmos implementados basados en visión artificial y al ajuste óptimo de los parámetros del controlador proporcional mediante diferentes pruebas que estudiaban el comportamiento en el tiempo de variables como el ángulo y la distancia a la línea.

En cuanto la navegación, ha resultado una de las tareas más arduas y tediosas, con multitud de ficheros y parámetros que configurar, que van desde las velocidades y aceleraciones del robot, hasta la jerarquía y organización de los sistemas de referencia del mismo, pasando por el modelado y la descripción de la plataforma. Han sido necesarias numerosas pruebas de ajuste en el entorno de trabajo, como la creación de mapas y la navegación por ellos.

Finalmente, la unión de todas estas etapas ha dado como resultado una plataforma totalmente operativa que puede ser utilizada como herramienta docente.

Durante la realización del proyecto, se han empleado una gran cantidad de conocimientos aprendidos a lo largo del grado y, lo que es más, se han adquirido multitud de conocimientos, no solo de los temas conocidos, sino de campos completamente nuevos para el autor como la Raspberry o el sistema operativo ROS.

Para concluir, se introducen, a continuación, una serie de líneas de mejora para que se tengan en cuenta en futuros proyectos de este tipo.

En primer lugar, a lo largo del desarrollo del proyecto se han tenido problemas relacionados con la potencia que ofrece el regulador lineal, de modo que, en ocasiones cuando se conectaban numerosos sensores o se utilizaba el puerto HDMI de la Raspberry, el regulador no daba la suficiente potencia y se reiniciaba la Raspberry o se apagaban sensores como el láser. Para ello se recomienda utilizar un regulador comercial más potente, que ofrezca un mayor amperaje para evitar este tipo de problemas.

Relacionado con lo anterior, otra de las líneas de actuación podría ir encaminada a la sustitución de la Raspberry por otro tipo de controladores similares más potentes como, por ejemplo, “ODROID-XU4” que dispone de hasta 8 núcleos y su precio no supera los 60€. Esto ofrecería una mayor capacidad de cálculo y velocidad para tareas como el tratamiento de imágenes o la navegación.

Otra de las principales líneas de mejora es el láser. Se podría sustituir el láser "Hokuyo" por el "RPLIDAR", que ofrece una nube de puntos de 360º frente a los 240º del "Hokuyo". Y no solo eso, sino que además la diferencia económica es muy llamativa. De los 1700€ que cuesta el "Hokuyo" se pasaría a unos 300€ o 400€ que vale el "RPLIDAR".

En cuanto a la navegación, las herramientas que ofrece ROS son muy amplias y potentes, por lo que, en futuros proyectos, se aconsejaría dedicarle mucho más tiempo a este campo, ya que se pueden llegar a conseguir aplicaciones espectaculares.

Otras líneas de mejora menos importantes, pero que pueden resultar de cierto interés para futuros sucesores de este proyecto son las siguientes:

- ✓ Inclusión de giróscopos para que sean estos sensores los que se encarguen de actualizar los giros en la odometría y no a través de estimaciones con las medidas de los codificadores incrementales, ya que ha sido uno de los aspectos más complicados de calibrar.
- ✓ Acoplar al controlador, ya sea la Raspberry o el "ODROID-XU4", una antena WIFI de mayor potencia que sobresalga por la parte superior, ya que el chasis del robot al ser de acero supone ciertas limitaciones con la comunicación inalámbrica del mismo.
- ✓ Finalmente, sería una característica de interés, la capacidad de desarrollar en el robot una aplicación que al detectar un nivel bajo de batería vaya automáticamente a un punto de carga establecido.

CAPÍTULO 8. BIBLIOGRAFÍA

[1] Aaron Martinez, Enrique Fernandez. (2013). Learning ROS for Robotics Programming.

[2] Barbus. (2014). El Cajón de Arduino. Available: http://elcajondeardu.blogspot.com.es/2014/03/tutorial-sensor-ultrasonidos-hc-sr04.html

[3] Rafael Belloso. (2010). URBE. Available: http://publicaciones.urbe.edu/index.php/telematique/article/viewFile/833/2037/6191

[4] Cmumrsdproject. (2017). Available: https://cmumrsdproject.wikispaces.com/ROS+navigation+stack+for+your+custom+robot

[5] Contribution of several authors. (2016). Odometría en robótica. Wikipedia. Available: https://es.wikipedia.org/wiki/Odometr%C3%ADa . Accessed 27/01/17.

[6] David Verdejo Garrido. (2011). Diseño y construcción de un robot didáctico controlado mediante plataforma Player Stage.

[7] MooreRobots. (2016). MooreRobots. Available: http://moorerobots.com/ . Accessed 20/05/17.

[8] Universidad Tecnológica de Pereira. (2009). Modelo cinemático de un robot móvil. Available: https://dialnet.unirioja.es/descarga/articulo/4729008.pdf

[9] ROS. (2017). ROS Tutorials. Available: http://wiki.ros.org/ROS/Tutorials

[10] ROS. (2017). ROS org. Available: http://www.ros.org/

[11] ROS. (2017). ROS navigation. Available: http://wiki.ros.org/navigation

[12] Contribution of several authors. (2017). Protocolo de comunicación I2C. Available: https://es.wikipedia.org/wiki/I%C2%B2C

CAPÍTULO 9. ANEXOS

Aquí se incluye información que, por su extensión o especificidad, no se ha incluido en los otros capítulos.

A continuación, se pueden ver los siguientes Anexos:

- Anexo A: Datasheets de los componentes.
- Anexo B: Planos y piezas de diseño 3D.
- Anexo C: *Drivers* controladores de los motores.
- Anexo D: *ROS WORKING SPACE*

ANEXO A: DATASHEETS DE LOS COMPONENTES

Raspberry pi 3

Raspberry Pi

Raspberry Pi 3 Model B

Product Name	**Raspberry Pi 3**
Product Description	The Raspberry Pi 3 Model B is the third generation Raspberry Pi. This powerful credit-card sized single board computer can be used for many applications and supersedes the original Raspberry Pi Model B+ and Raspberry Pi 2 Model B. Whilst maintaining the popular board format the Raspberry Pi 3 Model B brings you a more powerful processor, 10x faster than the first generation Raspberry Pi. Additionally it adds wireless LAN & Bluetooth connectivity making it the ideal solution for powerful connected designs.
RS Part Number	**896-8660**

RS

www.rs-components.com/raspberrypi

Raspberry Pi

Raspberry Pi 3 Model B

Specifications

Processor	Broadcom BCM2387 chipset. 1.2GHz Quad-Core ARM Cortex-A53 802.11 b/g/n Wireless LAN and Bluetooth 4.1 (Bluetooth Classic and LE)
GPU	Dual Core VideoCore IV® Multimedia Co-Processor. Provides Open GL ES 2.0, hardware-accelerated OpenVG, and 1080p30 H.264 high-profile decode. Capable of 1Gpixel/s, 1.5Gtexel/s or 24GFLOPs with texture filtering and DMA infrastructure
Memory	1GB LPDDR2
Operating System	Boots from Micro SD card, running a version of the Linux operating system or Windows 10 IoT
Dimensions	85 x 56 x 17mm
Power	Micro USB socket 5V1, 2.5A

Connectors:

Ethernet	10/100 BaseT Ethernet socket
Video Output	HDMI (rev 1.3 & 1.4 Composite RCA (PAL and NTSC)
Audio Output	Audio Output 3.5mm jack, HDMI USB 4 x USB 2.0 Connector
GPIO Connector	40-pin 2.54 mm (100 mil) expansion header: 2x20 strip Providing 27 GPIO pins as well as +3.3 V, +5 V and GND supply lines
Camera Connector	15-pin MIPI Camera Serial Interface (CSI-2)
Display Connector	Display Serial Interface (DSI) 15 way flat flex cable connector with two data lanes and a clock lane
Memory Card Slot	Push/pull Micro SDIO

Key Benefits

- Low cost
- 10x faster processing
- Consistent board format
- Added connectivity

Key Applications

- Low cost PC/tablet/laptop
- Media centre
- Industrial/Home automation
- Print server
- Web camera
- Wireless access point
- IoT applications
- Robotics
- Server/cloud server
- Security monitoring
- Gaming
- Environmental sensing/monitoring (e.g. weather station)

RS

www.rs-components.com/raspberrypi

Hokuyo

Date: 2005.10.26

Scanning Laser Range Finder
URG-04LX

Specifications

⑤ x 2	Revision history of firmware added	5	2008.4.25	Yamamoto	PR-5451
④ x 1	Scanning area	5	2007.4.16	Maeda	PR-5269
③ x 2	Com. Protocol added, revision history of firmware added	4,5	2007.1.18	Maeda	PR-5225
② x 3	Changes in resolution, revision history of firmware added	3,5	2006.9.21	Maeda	PR-5160
① x 5	Changes in cable color	4	2006.6.14	Maeda	PR-5111
Symbol	Amended Reason	Pages	Date	Corrector	Amendment No

Approved by	Checked by	Drawn by	Designed by	Title	Scanning Laser Range Finder URG-04LX Specifications	
MORI	MAEJIMA	SANTOSH	MAEDA	Drawing No.	C-42-3389	1/5

HOKUYO AUTOMATIC CO.,LTD.

1. General

URG-04LX is a laser sensor for area scanning. The light source of the sensor is infrared laser of wavelength 785nm with laser class 1 safety. Scan area is 240° semicircle with maximum radius 4000mm. Pitch angle is 0.36° and sensor outputs the distance measured at every point (683 steps). Laser beam diameter is less than 20mm at 2000mm with maximum divergence 40mm at 4000mm.

Principle of distance measurement is based on calculation of the phase difference, due to which it is possible to obtain stable measurement with minimum influence from object's color and reflectance.

URG-04LX is designed under JISC8201-5-2 and IEC60947-5-2 standards for industrial applications.

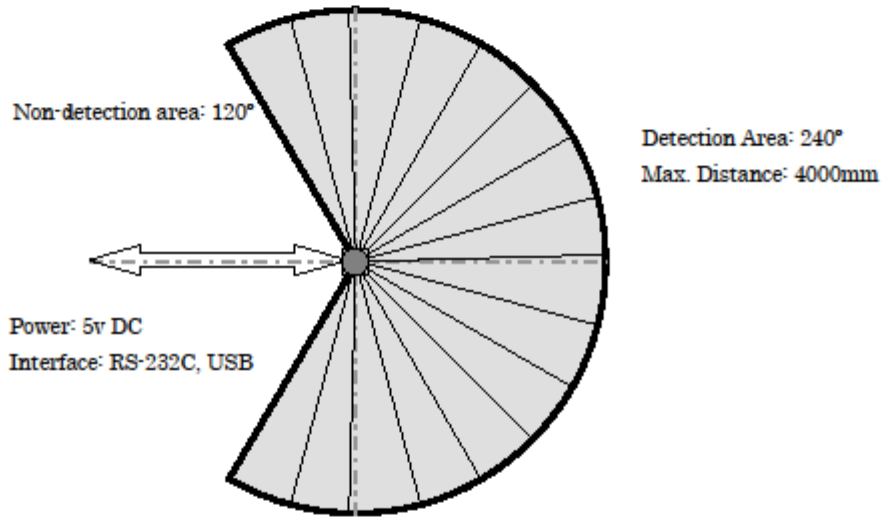


Figure 1

Note
Figure 1 shows the detectable area for white Kent sheet (70mm×70mm). Detection distance may vary with size and object.

2. Important Notice
This sensor is designed for indoor use only.
This sensor is not a safety device/tool
This sensor is not for use in military applications
Read specifications carefully before use.

Title	URG-04LX Specification	Drawing No	C-42-3389	2/5

HOKUYO AUTOMATIC CO.,LTD.

3. Specifications

Product name	Scanning Laser Range Finder
Model	URG-04LX
Light source	Semiconductor laser diode (λ=785nm), Laser power : less than 0.8mW Laser safety Class 1 (IEC60825-1)
Power voltage	5VDC ±5%
Power consumption	500mA or less (Start-up current 800mA)
Detection	60 mm ~ 4,095 mm (Guaranteed accuracy distance) 20mm ~ 5,600mm (Distance)* ⚠
Accuracy	Distance 20 ~ 1000mm: ±10mm*⚠ Distance 1000 ~ 4000mm: ±1% of measurement*⚠
Resolution	1 mm
Scan angle	240°
Angular resolution	0.36° (360° /1024)
Scanning speed	100msec/scan
Interface	RS-232C (19.2, 57.6, 115.2 ,500 ,750 kbps) USB Version 2.0 FS mode (12Mbps)
Ambient (Temperature/Humidity)	-10 ~ 50°C / 85%RH or less (without dew and frost)
Storage temperature	-25 ~ 75°C
Ambient light resistance	10000Lx or less
Vibration resistance	1.5mm double amplitude, 10 ~ 55Hz, X, Y and Z direction (2 hours), 98m/s² 55Hz ~ 150Hz in 2 minutes sweep, 1 hour each in X, Y and Z direction
Shock resistance	196 m/s², 10 times each in X, Y and Z direction
Protective structure	Optics : IP64 Case : IP40
Insulation	10MΩ for DC 500Vmegger
Weight	Approx. 160 g
Casing	Polycarbonate
Dimension (W×D×H)	50×50×70mm (Refer to design sheet No. C-40-3362)

*Under standard test conditions with white Kent sheet 70mm×70mm

4. Quality reference value

Operating vibration resistance	19.6m/s², 10Hz ~ 150Hz with 2 minutes sweep, 0.5 hours each in X, Y and Z direction
Operating impact resistance	49 m/s², 10 times each in X, Y and Z direction
Angular speed	360 deg/s
Angular acceleration	π/2 rad/s²
Lifespan	5 years (Vary on the operating conditions)
Noise level	25db or less (at 300mm)
FDA	This product complies with 21 CFR parts 1040.10 and 1040.11. (Registration Number 0521258)

Title	URG-04LX Specification	Drawing No	C-42-3389	3/5

HOKUYO AUTOMATIC CO.,LTD.

5. Interface

- CN1 (8 Pins)

	URG-04LX	Lead Color
1	N.C.	RED ⚠
2	N.C.	WHITE ⚠
3	OUTPUT (SYNCHRONOUS)	BLACK
4	GND (9pin Dsub 5p)	PURPLE ⚠
5	RxD (9pin Dsub 3p)	YELLOW ⚠
6	TxD (9pin Dsub 2p)	GREEN ⚠
7	0V	BLUE
8	DC 5V	BROWN

Note
GND and 0V are internally connected
A standard unit consists of power supply cable and 9-pin D-sub communication connector

- CN2 USB-mini (5 Pin)
  Cable is not included. Use commercially available compatible unit.

- Communication protocol ⚠
  Please refer to the respective document for details on communication protocol
  a) SCIP 1.0 : C-42-3320A
  b) SCIP 2.0 : C-42-3320B

- 1 Sync signal (approx. 12.5 ms) is outputted at each scan. Figure 2 shows the timing of the sync signal.

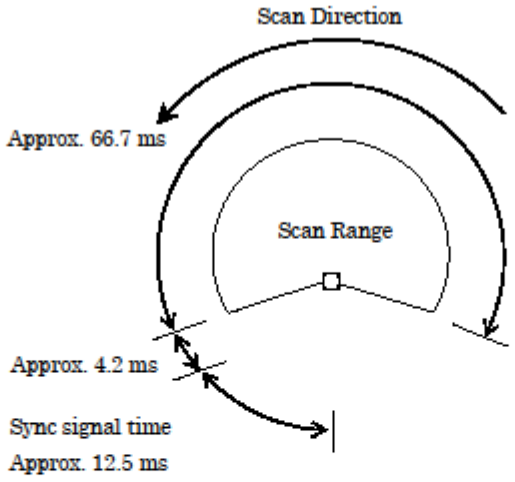


Figure 2

Title	URG-04LX Specification	Drawing No	C-42-3389	4/5

HOKUYO AUTOMATIC CO.,LTD.

6. Output circuit:

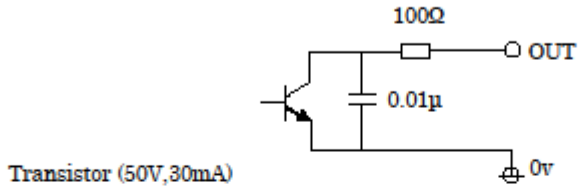


Figure 3

7. Additional notes:

- This sensor only needs 5VDC as power supply. Excessive power supply could cause damage to the sensor.
- Maximum data step is 683 steps. Since the angular resolution is 0.3515625° (360° /1024 steps), angular range is 239.765625° ((683-1)×360/1024).
- Angular resolution is configurable by the host. Read communication protocol specification (No C-42-3320) for details.
- Scan direction of the sensor is counterclockwise.
- In RS-232C communication, communication problems could happen if baud rate above 500 Kbps is set.
- USB driver used is the communication device class (CDC) supported by standard operating system. Sensor will be treated as COM port under the same utility when connected to the standard operating system.
- Plug and play function of the USB driver is not supported.

8. Firmware revision history ⚠2

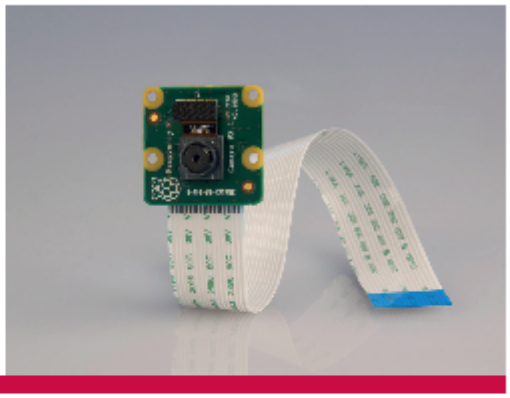
Firmware version	Changes
Ver. 2.91a	Laser is not radiated and LED will continue to blink until the connection is established.
Ver. 3.1.00 ⚠3	Fixed for SCIP 2.0. Function in ver. 2.91a is disabled. LED indicating the power supply will turn ON before communication is establish and start laser radiation.
Ver. 3.1.04 ⚠5	Corrections on MD/MS of SCIP
Ver. 3.3.00 ⚠5	HS command is added. Corrections on MD/MS of SCIP 2.0 Enhancement on error handling

Title	URG-04LX Specification	Drawing No	C-42-3389	5/5

HOKUYO AUTOMATIC CO.,LTD.

Cámara raspberry

Raspberry Pi



Camera Module

Product Name	**Raspberry Pi Camera Module**
Product Description	High Definition camera module compatible with all Raspberry Pi models. Provides high sensitivity, low crosstalk and low noise image capture in an ultra small and lightweight design. The camera module connects to the Raspberry Pi board via the CSI connector designed specifically for interfacing to cameras. The CSI bus is capable of extremely high data rates, and it exclusively carries pixel data to the processor.
RS Part Numer	**913-2664**
Specifications	
Image Sensor	Sony IMX 219 PQ CMOS image sensor in a fixed-focus module.
Resolution	8-megapixel
Still picture resolution	3280 x 2464
Max image transfer rate	1080p: 30fps (encode and decode) 720p: 60fps
Connection to Raspberry Pi	15-pin ribbon cable, to the dedicated 15-pin MIPI Camera Serial Interface (CSI-2).
Image control functions	Automatic exposure control Automatic white balance Automatic band filter Automatic 50/60 Hz luminance detection Automatic black level calibration
Temp range	Operating: -20° to 60° Stable image: -20° to 60°
Lens size	1/4"
Dimensions	23.86 x 25 x 9mm
Weight	3g

www.rs-online.com/raspberrypi

Ultrasonidos

ELEC Freaks Tech Support: services@elecfreaks.com

Ultrasonic Ranging Module HC - SR04

Product features:

Ultrasonic ranging module HC - SR04 provides 2cm - 400cm non-contact measurement function, the ranging accuracy can reach to 3mm. The modules includes ultrasonic transmitters, receiver and control circuit. The basic principle of work:
(1) Using IO trigger for at least 10us high level signal,
(2) The Module automatically sends eight 40 kHz and detect whether there is a pulse signal back.
(3) IF the signal back, through high level , time of high output IO duration is the time from sending ultrasonic to returning.
Test distance = (high level time×velocity of sound (340M/S) / 2,

Wire connecting direct as following:

- 5V Supply
- Trigger Pulse Input
- Echo Pulse Output
- 0V Ground

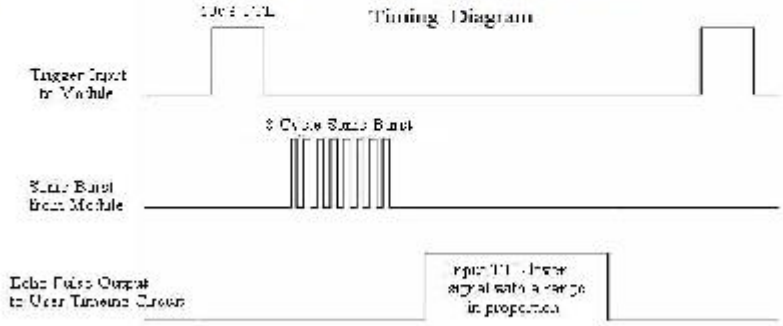
Electric Parameter

Working Voltage	DC 5 V
Working Current	15mA
Working Frequency	40Hz
Max Range	4m
Min Range	2cm
MeasuringAngle	15 degree
Trigger Input Signal	10uS TTL pulse
Echo Output Signal	Input TTL lever signal and the range in proportion
Dimension	45*20*15mm

Vcc Trig Echo GND

Timing diagram

The Timing diagram is shown below. You only need to supply a short 10uS pulse to the trigger input to start the ranging, and then the module will send out an 8 cycle burst of ultrasound at 40 kHz and raise its echo. The Echo is a distance object that is pulse width and the range in proportion .You can calculate the range through the time interval between sending trigger signal and receiving echo signal. Formula: uS / 58 = centimeters or uS / 148 =inch; or: the range = high level time * velocity (340M/S) / 2; we suggest to use over 60ms measurement cycle, in order to prevent trigger signal to the echo signal.



Motores

IG-42GM

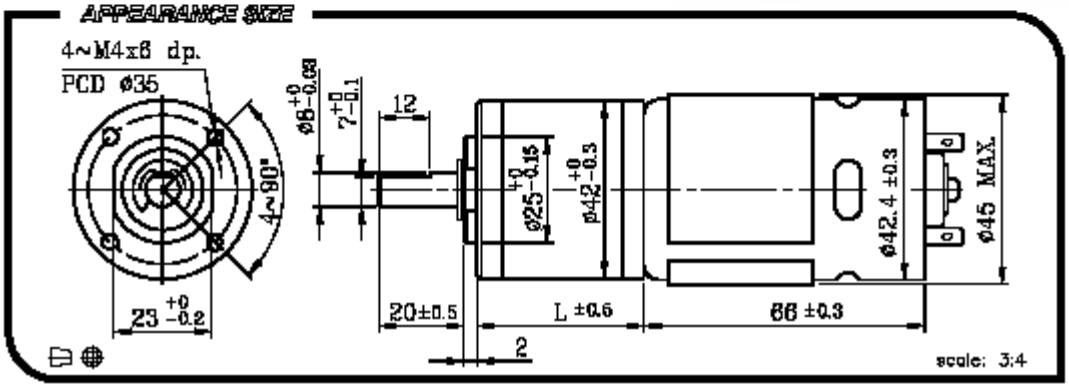
01&02 TYPE

(DC Carbon-brush motors)

IG-42 GEARED MOTOR SERIES

REDUCTION RATIO	L	REDUCTION RATIO	L
1/4	32.5	1/212~1/864	52.6
1/14~1/24	39.2	1/1062~1/3600	59.6
1/49~1/144	45.9		

APPEARANCE SIZE



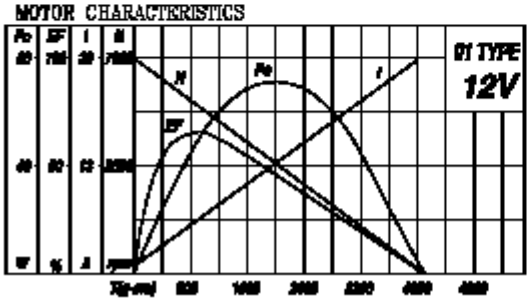
GEARED MOTOR TORQUE/SPEED

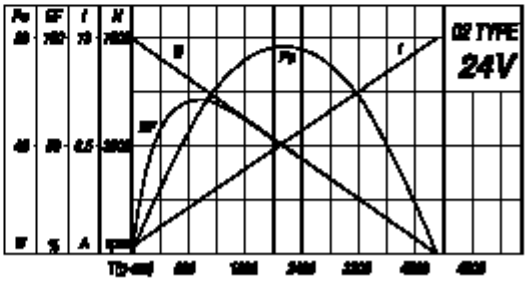
	Reduction ratio	1/4	1/14	1/17	1/24	1/49	1/61	1/84	1/104	1/144	1/212	1/294	1/504	1/624	1/720	1/864	1/1062	1/1470	1/2500	1/3000	1/3600
12V	Rated torque (Kg-cm)	2.2	6.5	8.1	10	18	18	18	20	20	25	25	30	30	30	30	30	30	30	30	30
	Rated speed (rpm)	1400	405	325	248	120	98	76	63	45	31	24	13.5	10.9	9.5	8.0	6.5	4.6	2.7	2.3	1.9
24V	Rated torque (Kg-cm)	1.8	5.4	6.6	8.5	16	18	18	20	20	25	25	30	30	30	30	30	30	30	30	30
	Rated speed (rpm)	1445	420	340	240	122	102	77.5	63	47	31	23.8	13.5	10.9	9.5	8.0	6.5	4.6	2.7	2.3	1.9

MOTOR DATA

Rated volt (V)	Rated torque (g-cm)	Rated speed (rpm)	Rated current (mA)	No load speed (rpm)	No load current (mA)	Rated output (W)	Weight (g)
12	700	5700	≤ 5500	7000	≤ 900	41.3	360
24	570	5900	≤ 2100	7000	≤ 500	34.7	360

MOTOR CHARACTERISTICS



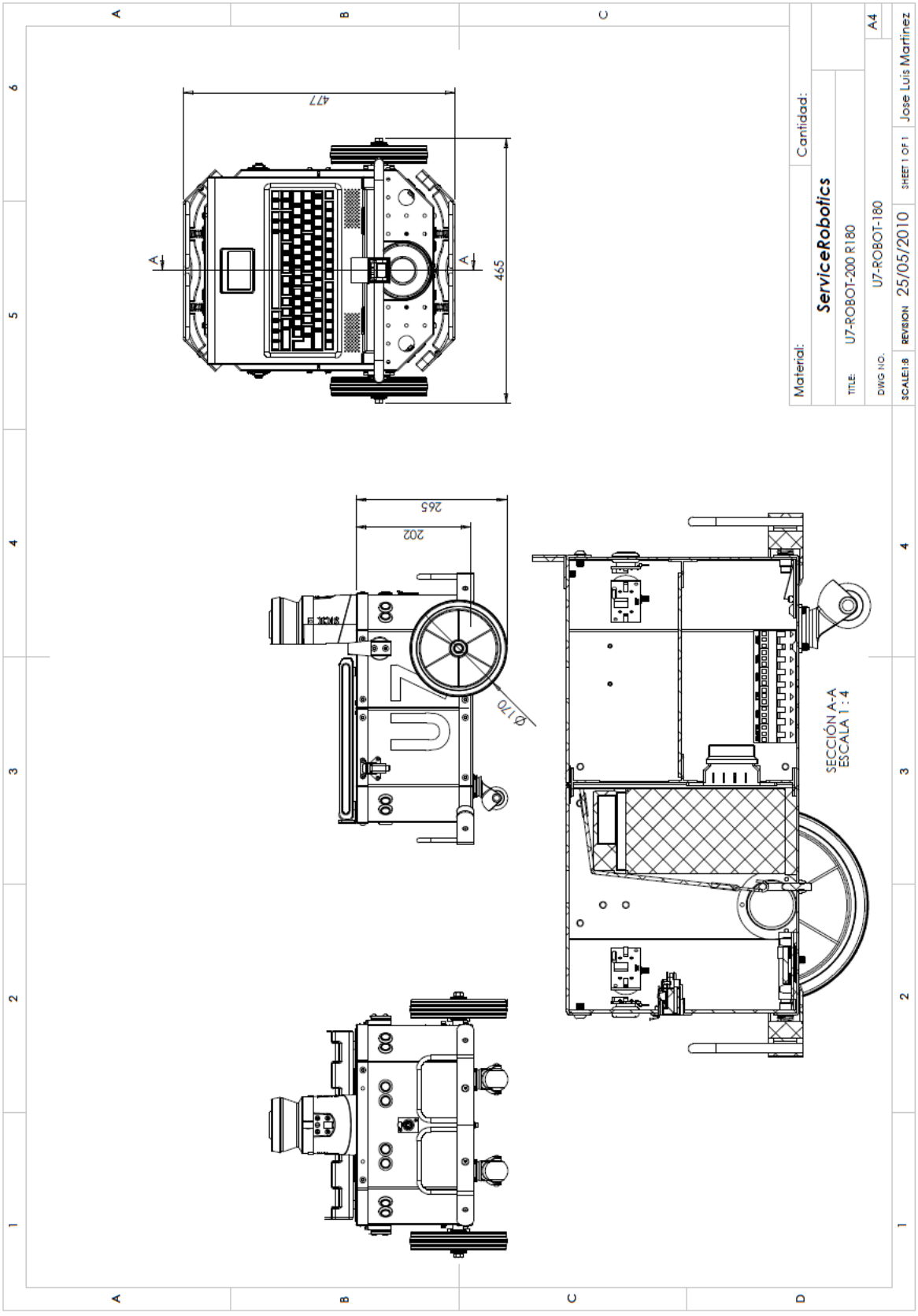


ISL PRODUCTS, INC.

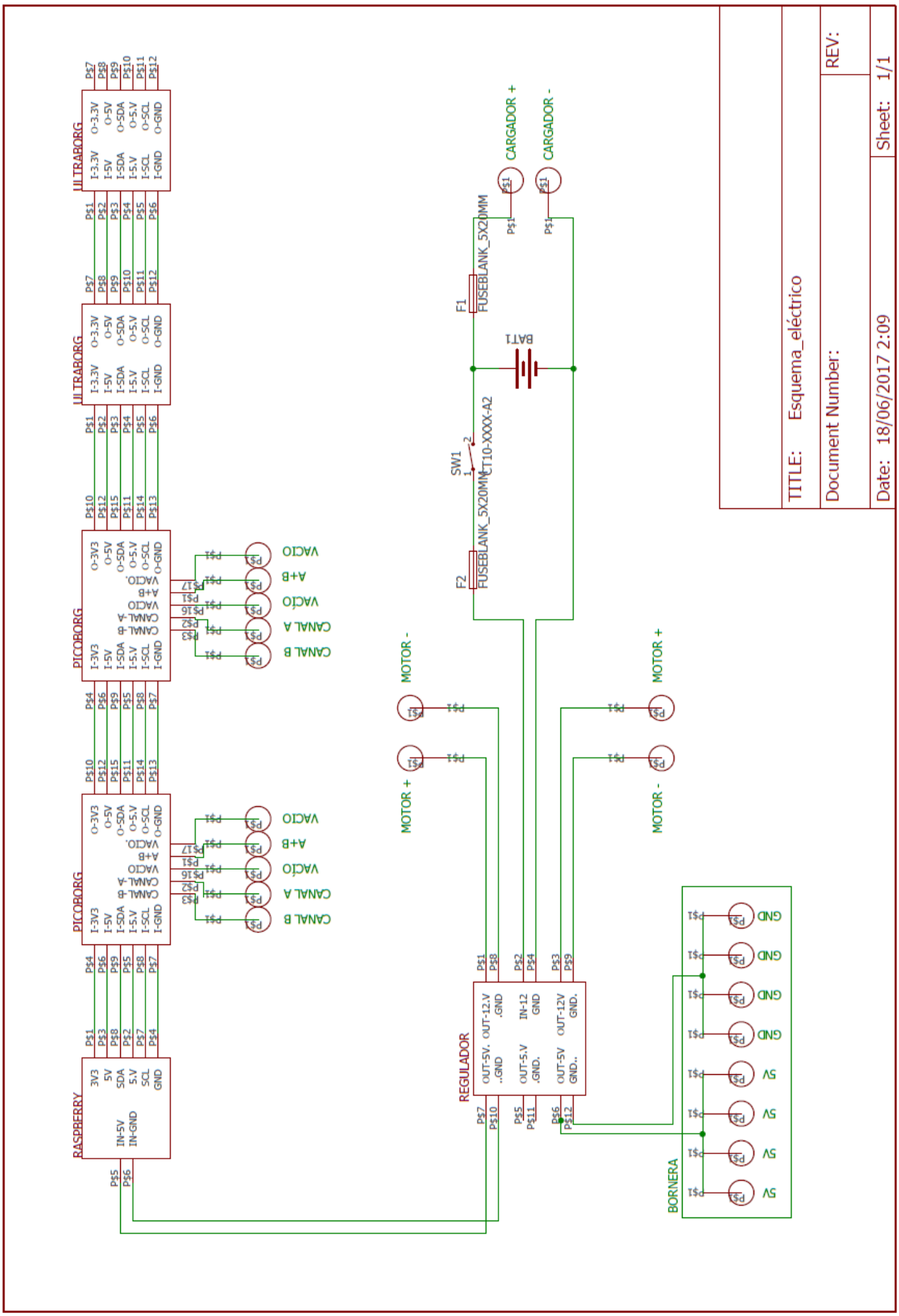
September 2006

ANEXO B: PLANOS Y PIEZAS DE DISEÑO 3D

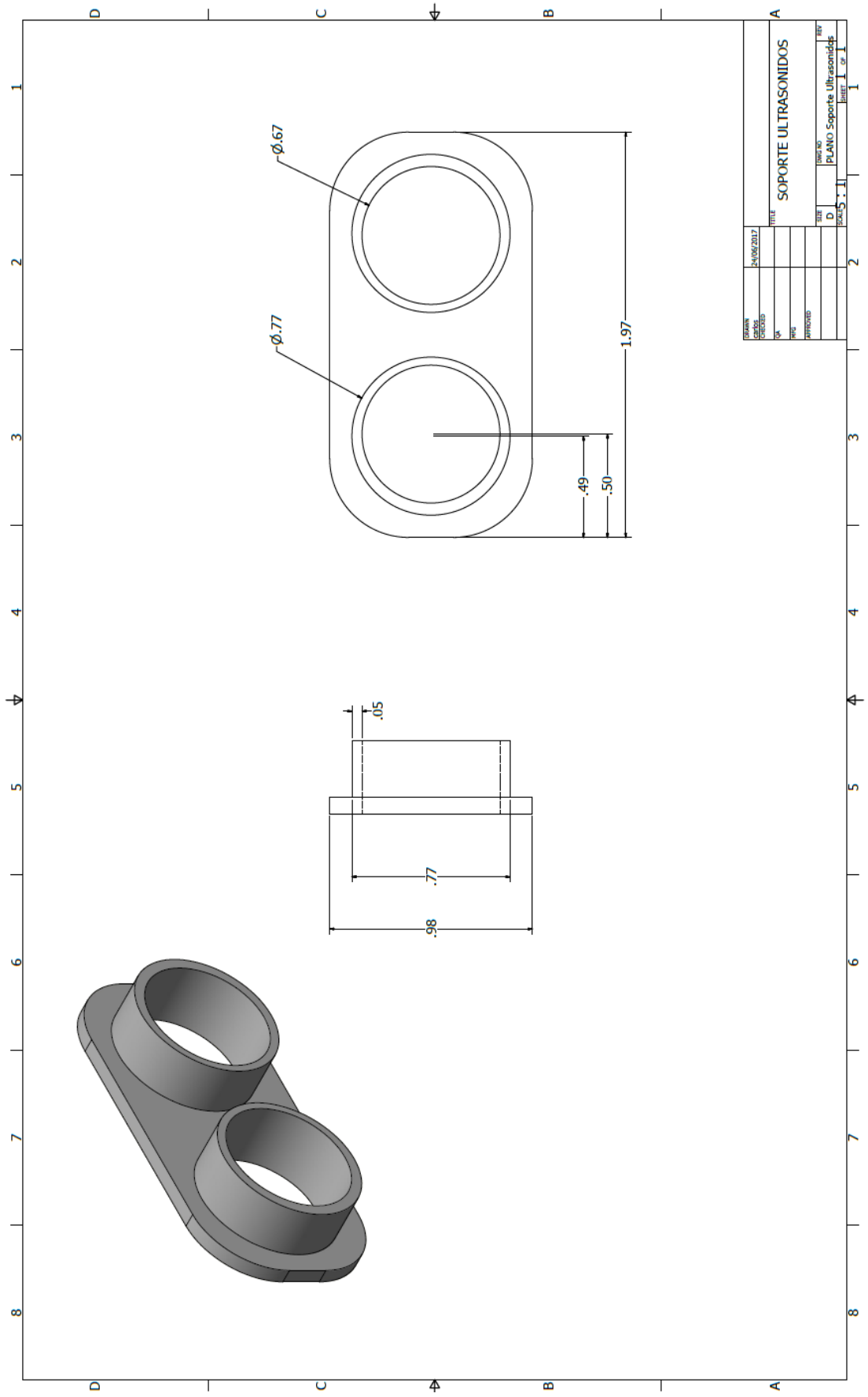
Plano del chasis de Edubot



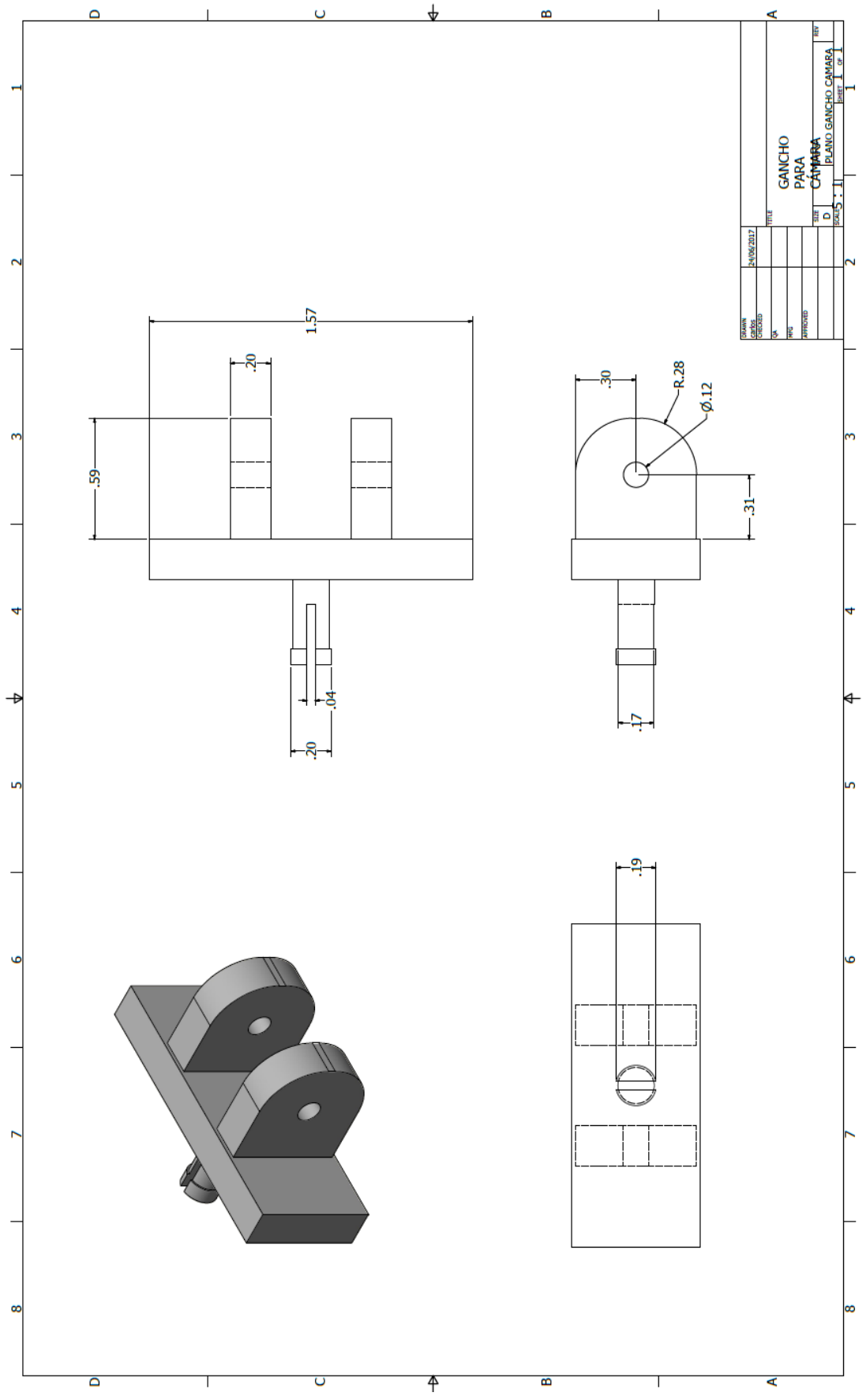
Plano eléctrico



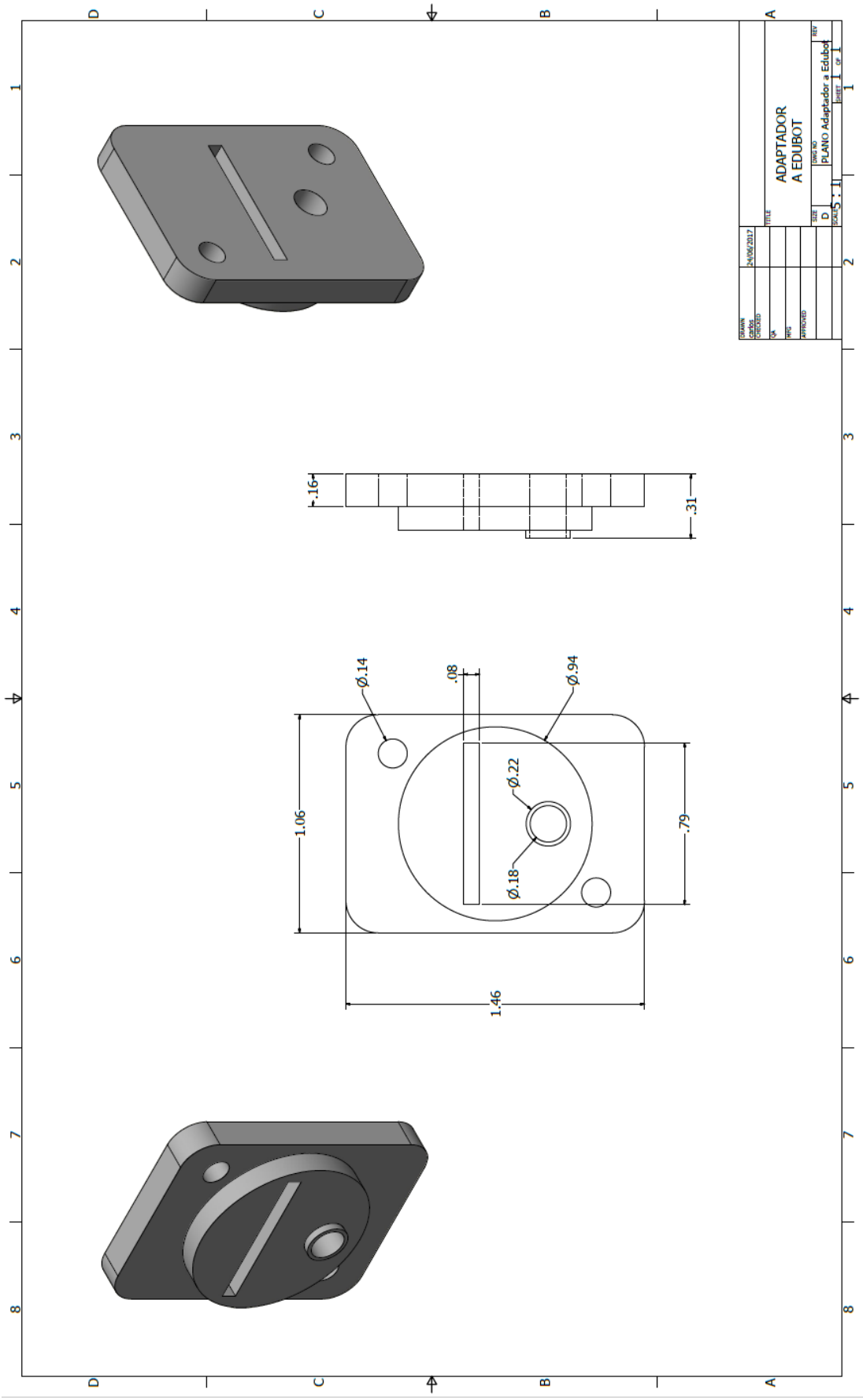
Soporte ultrasonidos



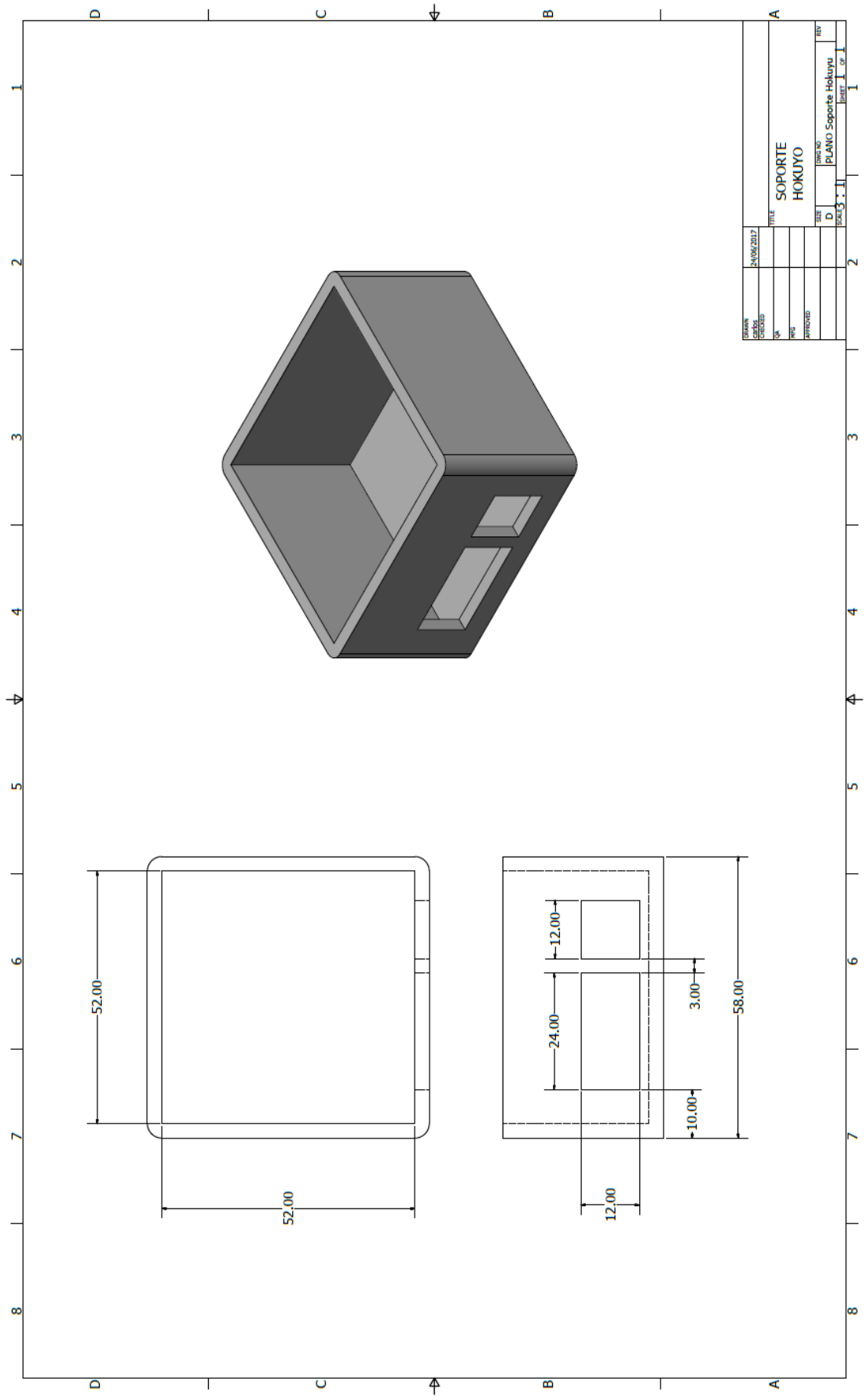
Gancho cámara



Adaptador a Edubot



Soporte hokuyo



ANEXO C: DRIVERS CONTROLADORAS DE LOS MOTORES (PICOBORG REVERSE)

System

```
/******************************************************************************/
/* System Level #define Macros                                                */
/******************************************************************************/

/* TODO Define system operating frequency */

/* Microcontroller MIPs (FCY) */
#define SYS_FREQ        32000000L
#define _XTAL_FREQ      SYS_FREQ
#define FCY             SYS_FREQ/4

/******************************************************************************/
/* System Function Prototypes                                                 */
/******************************************************************************/

/* Custom oscillator configuration funtions, reset source evaluation
functions, and other non-peripheral microcontroller initialization functions
go here. */

#define Delay_us(x)         __delay_us(x)

void ConfigureOscillator(void); /* Handles clock switching/osc initialization */
//void Delay_ms(int x);
//unsigned int set_ms_match_value16(float tms,unsigned int preescaler);
```

User

```
/************************************************************************/
/* User Level #define Macros                                           */
/************************************************************************/

#define PWM_MAX                 (255)
#define I2C_MAX_LEN             (4)
#define FAILSAFE_LIMIT          (93)    //(31) Approximately 0.25 seconds at a
122 Hz timer

#define KMOT                    0.1
#define KP                      0.4
#define KI                      0.1
#define UMBRAL_INTEGRAL         3000
#define LIMVMMAX                1000
#define MIN_MOV                 110
#define I2C_ID_PICOBORG_REV     (0x15)

#define COMMAND_SET_LED         (1)     // Set the LED status
#define COMMAND_GET_LED         (2)     // Get the LED status
#define COMMAND_SET_A_FWD       (3)     // Set motor A PWM rate in a forwards
direction
#define COMMAND_SET_A_REV       (4)     // Set motor A PWM rate in a reverse
direction
#define COMMAND_GET_A           (5)     // Get motor A direction and PWM rate
#define COMMAND_SET_B_FWD       (6)     // Set motor B PWM rate in a forwards
direction
#define COMMAND_SET_B_REV       (7)     // Set motor B PWM rate in a reverse
direction
#define COMMAND_GET_B           (8)     // Get motor B direction and PWM rate
#define COMMAND_ALL_OFF         (9)     // Switch everything off
#define COMMAND_RESET_EPO       (10)    // Resets the EPO flag, use after EPO
has been tripped and switch is now clear
#define COMMAND_GET_EPO         (11)    // Get the EPO latched flag
//#define COMMAND_SET_EPO_IGNORE    (12)    // Set the EPO ignored flag, allows
the system to run without an EPO
//#define COMMAND_GET_EPO_IGNORE    (13)    // Get the EPO ignored flag
#define COMMAND_GET_DRIVE_FAULT (14)    // Get the drive fault flag, indicates
faults such as short-circuits and under voltage
#define COMMAND_SET_ALL_FWD     (15)    // Set all motors PWM rate in a forwards
direction
#define COMMAND_SET_ALL_REV     (16)    // Set all motors PWM rate in a reverse
direction
#define COMMAND_SET_FAILSAFE    (17)    // Set the failsafe flag, turns the
motors off if communication is interrupted
#define COMMAND_GET_FAILSAFE    (18)    // Get the failsafe flag
#define COMMAND_SET_ENC_MODE    (19)    // Set the board into encoder or speed
mode
#define COMMAND_GET_ENC_MODE    (20)    // Get the boards current mode, encoder
or speed
#define COMMAND_SET_KMOT        (21)    // Set Kmot
#define COMMAND_SET_KP          (22)    // Set Kp
#define COMMAND_SET_KI          (23)    // Set Ki
#define COMMAND_GET_KMOT        (24)    // Get Kmot
#define COMMAND_GET_KP          (25)    // Get Kp
#define COMMAND_GET_KI          (26)    // Get Ki
#define COMMAND_GET_ENC_MOVING  (27)    // Get the status of encoders moving
#define COMMAND_SET_ENC_SPEED   (28)    // Set the maximum PWM rate in encoder
mode
```

```
#define COMMAND_SET_SPEED_FBCK  (30)    // Set speed in pulses/seg
#define COMMAND_GET_SPEED_FBCK  (31)    // Get speed in pulses/seg
#define COMMAND_GET_EPO_PRESSED (32)    // Get EPO state
#define COMMAND_GET_ALARMS      (33)    // Get all alarms 0000
#define COMMAND_GET_ID          (0x99)  // Get the board identifier
#define COMMAND_SET_I2C_ADD     (0xAA)  // Set a new I2C address
#define COMMAND_VALUE_FWD       (1)     // I2C value representing forward
#define COMMAND_VALUE_REV       (2)     // I2C value representing reverse
#define COMMAND_VALUE_ON        (1)     // I2C value representing on
#define COMMAND_VALUE_OFF       (0)     // I2C value representing off
#define EEPROM_I2C_ADDRESS      (0x00)  // Saved I2C address
/*************************************************************************/
/* Global variables                                                      */
/*************************************************************************/
extern unsigned char i2cSend[I2C_MAX_LEN];
extern unsigned char i2cRecv[I2C_MAX_LEN];
extern unsigned char i2cByte;
extern bool epoTripped;
//extern bool epoIgnored;
extern bool lowVoltage;
extern bool encoderFail;
extern char junk;
extern int failsafeCounter;
extern bool encMode;
extern bool movingA;
extern bool movingB;
extern int encLimit;
//extern int remainingCountsA;
//extern int remainingCountsB;
extern long int CountEnc;
extern signed int sCountEnc;
extern bool flagTim;
extern int CountTim;
extern long int Pulsos_seg;
//extern long int ContaPulsos_env;
extern signed int Pulsos_seg_10[10];
extern float Pulsos_seg_sum;
extern int Referencia;
extern float Ref_ant;
extern float Error1;
extern int vm;
extern int vm_ant;
//extern int pwm2;
extern int ContaParo;
extern int ContaEnc;
extern bool sentidoprueba;
extern float Kmot;
extern float Kp;
extern float Ki;
extern bool Control;
/*************************************************************************/
/* User Function Prototypes                                              */
/*************************************************************************/
void InitApp(void);         /* I/O and Peripheral Initialization */
void Conf_Timer0();
//void SetMotorA(bool reverse, int pwm);
//void SetMotorB(bool reverse, int pwm);
void SetAllMotors(bool reverse, int pwm);
void SetEncoderMode(bool enabled);
void ProcessI2C(int len);
void ControlMotor();
```

Main

```
/*****************************************************************************/
/* Files to Include                                                          */
/*****************************************************************************/

#if defined(__XC)
    #include <xc.h>         /* XC8 General Include File */
#elif defined(HI_TECH_C)
    #include <htc.h>        /* HiTech General Include File */
#endif

#include <stdint.h>        /* For uint8_t definition */
#include <stdbool.h>       /* For true/false definition */

#include "system.h"        /* System funct/params, like osc/peripheral config */
#include "user.h"          /* User funct/params, such as InitApp */

/*****************************************************************************/
/* Private Prototypes                                                        */
/*****************************************************************************/
void LedTimingTest(void);
void MotorTest(void);
void MotorTestPwm(void);
void MotorTestPwm2(void);

/*****************************************************************************/
/* User Global Variable Declaration                                          */
/*****************************************************************************/

#define SEQ_DELAY_MS        (5000)
#define PWM_SEQ_DELAY_MS    (50)
#define PWM2_SEQ_DELAY_MS   (100)
#define PWM2_FULL_DELAY_MS  (500)

unsigned char i2cSend[I2C_MAX_LEN] = {0};
unsigned char i2cRecv[I2C_MAX_LEN] = {0};
unsigned char i2cByte = 0;
bool epoTripped = false;
//bool epoIgnored = false;
bool lowVoltage = false;
bool encoderFail= false;
char junk = 0x00;
int failsafeCounter = 0;
bool encMode = false;
bool movingA = false;
bool movingB = false;
int encLimit = 255;
//int remainingCountsA = 0;
//int remainingCountsB = 0;
long int CountEnc = 0;
signed int sCountEnc=0;
//bool flag_led=false;
bool flagTim=false;
int CountTim=0;
long int Pulsos_seg=0;
signed int Pulsos_seg_10[10];
float Pulsos_seg_sum;
```

```
//long int ContaPulsos_env;
int Referencia=0;
float Ref_ant=0;
float Error1=0;
int vm;
int vm_ant;
//int pwm2;
int ContaParo=0;
bool ContaSeta=false;
bool sentidoprueba=false;
float Kmot=KMOT;
float Kp=KP;
float Ki=KI;
bool Control=false;

/*****************************************************************************/
/* Main Program                                                              */
/*****************************************************************************/
void main(void) {
    short IteracionEncoder=9;
//  int i,j=0,h;
//    unsigned int dato_array=0;
//    bool lleno=false;
    // Configure the oscillator for the device
    ConfigureOscillator();

    // Initialize I/O and Peripherals for application
    InitApp();

    // Test sequences
    LATAbits.LATA4 = 1;     // LED Off
    //LedTimingTest();
    //MotorTest();
    //MotorTestPwm();
    //MotorTestPwm2();

    // Brief LED pulse
//  LATAbits.LATA4 = 0;     // LED On
//  Delay_ms(500);
//  LATAbits.LATA4 = 1;     // LED Off

    // Clear the I2C buffer
    do {
        PIR1bits.SSP1IF = 0;            // Clear the interrupt
        junk = SSP1BUF;                 // Discard the transmitted byte
        SSP1CON1bits.CKP = 1;           // Release clock line
    } while (SSP1STATbits.BF);

    INTCONbits.T0IE=1;
    INTCONbits.T0IF=0;
    // Main loop
    INTCONbits.GIE = 1;                 // Enable interrupts
    failsafeCounter = 0;
```

```
while (1) {
        // Check the communications failsafe

        if (PIR3bits.TMR4IF) {
            PIR3bits.TMR4IF = 0;            // Clear the interrupt

            ++failsafeCounter;              // Increment the counter
            if (failsafeCounter > FAILSAFE_LIMIT) {
                // Failsafe timeout exceeded, presume that communications have
failed

//                SetAllMotors(false, 0);
                Referencia=0;
                // Invert the LED
                if (PORTAbits.RA4 == 1) {
                    LATAbits.LATA4 = 0;
                } else {
                    LATAbits.LATA4 = 1;
                }
                // Reset the counter
                failsafeCounter = 0;
            }
        }


        if(flagTim==1){
            //Check if voltage low
            if (PORTAbits.RA2 == 0)
                lowVoltage=true;
            else
                lowVoltage=false;

            // Check if the EPO has been tripped
            if (!epoTripped && (PORTAbits.RA5 == 1)) {
                if(ContaSeta){
                    epoTripped = true;
                    SetAllMotors(false,0);
                    movingA = false;
                    movingB = false;
                    Control=false;
                    Error1=0.0;
                }
                ContaSeta=true;
            }

            if(CountEnc>32767) sCountEnc=CountEnc-65535;
                        else  sCountEnc=CountEnc;
                        CountEnc=0;
                        Pulsos_seg_10[9]=sCountEnc*10;
                        Pulsos_seg_sum=0;
                        for(int h=9;h>=(IteracionEncoder-1);h--){
                            Pulsos_seg_sum+=Pulsos_seg_10[h];
                            if(IteracionEncoder == 0)
                                Pulsos_seg_10[9-h]=Pulsos_seg_10[10-h];
                            else
                                Pulsos_seg_10[IteracionEncoder+(8-
            h)]=Pulsos_seg_10[IteracionEncoder+(9-h)];
            }
```

```
Pulsos_seg=(int)(Pulsos_seg_sum/(10-IteracionEncoder));

            if(IteracionEncoder>0){
                IteracionEncoder--;
            }

            if(Control==true){
                ControlMotor();
            }

            if(Pulsos_seg_sum==0 && Referencia!=0){         //Comprobacion para
saber si el encoder cuenta pulsos

                if(ContaParo>30){
                    ContaParo=30;
                    encoderFail = true;
                    SetAllMotors(false,0);
                    movingA = false;
                    movingB = false;
                    Control=false;
                    Error1=0.0;
                    LATAbits.LATA4 = 0;
                }
                ContaParo++;
            }else{
                ContaParo=0;
                encoderFail = false;
                LATAbits.LATA4 = 1;
                Control = true;
            }

            flagTim=0;
        }
        // Kick the watchdog
        CLRWDT();
    }
}


/*
void LedTimingTest(void) {
    // Pulse LED at a 1 second interval
    while (1) {
        Delay_ms(500);
        CLRWDT();
        LATAbits.LATA4 = 1;     // LED Off
        Delay_ms(500);
        CLRWDT();
        LATAbits.LATA4 = 0;     // LED On
    }
}
*/
```

System

```
/*********************************************************************************/
/*Files to Include                                                               */
/*********************************************************************************/

#if defined(__XC)
    #include <xc.h>         /* XC8 General Include File */
#elif defined(HI_TECH_C)
    #include <htc.h>        /* HiTech General Include File */
#endif

#include <stdint.h>        /* For uint8_t definition */
#include <stdbool.h>       /* For true/false definition */

#include "system.h"

/* Refer to the device datasheet for information about available
oscillator configurations. */
void ConfigureOscillator(void) {
    OSCCONbits.SCS = 0b0;       // Set the clock to the CONFIG1 setting
(internal oscillator)
    OSCCONbits.IRCF = 0b1110;       // Select the 8 MHz postscaler
    OSCCONbits.SPLLEN = 1;      // Enable the 4x PLL (overriden in CONFIG2 by
PLLEN = ON)
}

//void Delay_ms(int x) {
//    while (x > 0) {
//        Delay_us(1000);
//        --x;
//    }
//}
void Conf_Timer0()
{
    unsigned int match_value;

    //match_value=set_ms_match_value16(100.0, 64);
    match_value=0x05;

    if(match_value>0)
    {
        OPTION_REG= 0xC6;//C5;
        TMR0=0x10;
    }else{
        LATAbits.LATA4 = 0;     // LED On

    }
    //TMR0IE=0;
}
//unsigned int set_ms_match_value16(float tms,unsigned int preescaler)
//{
//
//  float mv;
//
//  mv=(tms*FCY)/(((float)preescaler)*1000.0);
//
//  if(mv>((float)0xFFFF))  return 0;
//  else                    return (unsigned int)mv;
//}
```

User

```
/******************************************************************************/
/* Files to Include                                                           */
/******************************************************************************/

#if defined(__XC)
    #include <xc.h>         /* XC8 General Include File */
#elif defined(HI_TECH_C)
    #include <htc.h>        /* HiTech General Include File */
#endif

#include <stdint.h>         /* For uint8_t definition */
#include <stdbool.h>        /* For true/false definition */

#include "user.h"

/******************************************************************************/
/* User Functions                                                             */
/******************************************************************************/

void InitApp(void) {
    unsigned char value;

    /* Setup analog functionality and port direction */
    ANSELA = 0;                     // All digital IO, port A
    ANSELC = 0;                     // All digital IO, port C
    TRISAbits.TRISA0 = 1;           // Encoder line for B
    TRISAbits.TRISA1 = 1;           // Encoder line for A
    TRISAbits.TRISA2 = 1;           // Error flag
    TRISAbits.TRISA4 = 0;           // LED
    TRISAbits.TRISA5 = 1;           // EPO
    TRISCbits.TRISC1 = 1;           // I2C Data
    TRISCbits.TRISC0 = 1;           // I2C Clock
    TRISCbits.TRISC3 = 0;           // Motor A:1
    TRISCbits.TRISC2 = 0;           // Motor A:2
    TRISCbits.TRISC5 = 0;           // Motor B:1
    TRISCbits.TRISC4 = 0;           // Motor B:2

    /* Initialize peripherals */

    // Ports
    PORTAbits.RA4 = 0;              // LED On
    PORTCbits.RC3 = 0;              // Motor A:1 Off
    PORTCbits.RC2 = 0;              // Motor A:2 Off
    PORTCbits.RC5 = 0;              // Motor B:1 Off
    PORTCbits.RC4 = 0;              // Motor B:2 Off
// MSSP (I2C)
    SSP1CON1bits.SSPM = 0b0110;     // I2C slave mode, 7b address (interrupts
enabled manually)
    SSP1CON1bits.CKP = 1;           // Release clock line
    SSP1CON1bits.SSPOV = 1;         // Clear the overflow flag
    SSP1CON1bits.WCOL = 1;          // Clear the collision flag
    SSP1STATbits.SMP = 1;           // Slew rate control disabled (100 KHz I2C)
    SSP1STATbits.CKE = 0;           // Transmission on idle to active clock
    SSP1CON2bits.GCEN = 1;          // Enable listening for broadcasts
    SSP1CON2bits.SEN = 1;           // Clock stretching enabled
    SSP1CON3bits.PCIE = 1;          // Stop condition interrupts enabled
```

```
SSP1CON3bits.SCIE = 0;           // Start condition interrupts disabled
    SSP1CON3bits.BOEN = 0;           // Buffer is only overwritten if SSPOV is
clear
    SSP1CON3bits.SDAHT = 1;          // Minimum of 300ns SDA hold time after
clock edge
    SSP1CON3bits.SBCDE = 0;          // Bus collision detection disabled
    SSP1CON3bits.AHEN = 0;           // Automatic holding of the clock after
address disabled
    SSP1CON3bits.DHEN = 0;           // Automatic holding of the clock after data
disabled
    SSP1MSK = 0xFE;                  // Address mask against full 7 bits
    SSP1ADD = 0x88;                  // Default I2C address
    SSP1CON1bits.SSPEN = 1;          // Enable the SDA and SCL pins as the port
inputs (RC0 and RC1)

    // PWM
    CCP1CONbits.CCP1M = 0b0000;      // PWM 1 disabled
    CCP2CONbits.CCP2M = 0b0000;      // PWM 2 disabled
    CCP3CONbits.CCP3M = 0b0000;      // PWM 3 disabled
    CCP4CONbits.CCP4M = 0b0000;      // PWM 4 disabled
    TRISCbits.TRISC5 = 1;            // Disable PWM 1 output
    TRISCbits.TRISC3 = 1;            // Disable PWM 2 output
    PR2 = 0x3F;                      // 8-bit resolution, 125 KHz at 1:1 prescale
    CCP1CONbits.P1M = 0b00;          // PWM 1 single output mode (P1A/RC5 only)
    CCP2CONbits.P2M = 0b00;          // PWM 2 single output mode (P2A/RC3 only)
    CCP1CONbits.CCP1M = 0b1100;      // PWM 1 set to PWM mode (B/C/D active high)
    CCP2CONbits.CCP2M = 0b1100;      // PWM 2 set to PWM mode (B/C/D active high)
    CCPR1L = 0x0;                    // PWM 1 pulse width set to 0 (MSBs)
    CCP1CONbits.DC1B = 0b00;         // PWM 1 pulse width set to 0 (LSBs)
    CCPR2L = 0x0;                    // PWM 2 pulse width set to 0 (MSBs)
    CCP2CONbits.DC2B = 0b00;         // PWM 2 pulse width set to 0 (LSBs)
    CCPTMRS0bits.C1TSEL = 0b00;      // PWM 1 set to period from Timer 2
    CCPTMRS0bits.C2TSEL = 0b00;      // PWM 2 set to period from Timer 2
    T2CONbits.T2OUTPS = 0b0000;      // Timer 2 1:1 postscaler
    T2CONbits.T2CKPS = 0b10;         // Timer 2 1:16 prescaler, 7.8125 KHz
    T2CONbits.TMR2ON = 1;            // Timer 2 enabled
    PIR1bits.TMR2IF = 0;             // Clear Timer 2 interrupt flag
    while (PIR1bits.TMR2IF == 0) ;   // Wait for Timer 2 to loop
    TRISCbits.TRISC5 = 0;            // Enable PWM 1 output
    TRISCbits.TRISC3 = 0;            // Enable PWM 2 output

    // Failsafe timer
    PR4 = 0x3F;                      // 8-bit resolution, 125 KHz at 1:1 prescale
    T4CONbits.T4CKPS = 0b11;         // Timer 4 1:64 prescaler, 1,953.125 Hz
    T4CONbits.T4OUTPS = 0b1111;      // Timer 4 1:16 postscaler, ~122 Hz
    T4CONbits.TMR4ON = 0;            // Timer 4 disabled
    PIR3bits.TMR4IF = 0;             // Clear Timer 4 interrupt flag
// OTHERS
    MDCONbits.MDEN = 0;              // Modulator disabled
    MDCONbits.MDOE = 0;              // Modulator ouput disabled
    SRCON0bits.SRLEN = 0;            // SR latch disabled
    SRCON0bits.SRQEN = 0;            // SR latch output disabled
    SRCON0bits.SRNQEN = 0;           // SR latch #output disabled
    CM1CON0bits.C1ON = 0;            // Comparitor 1 disabled
    CM1CON0bits.C1OE = 0;            // Comparitor 1 output disabled
    CM2CON0bits.C2ON = 0;            // Comparitor 2 disabled
    CM2CON0bits.C2OE = 0;            // Comparitor 2 output disabled
    RCSTAbits.SPEN = 0;              // UART disabled
```

```
// Interrupts (global enable elsewhere)
    INTCONbits.PEIE = 1;            // Peripheral interrupts enabled
    PIE1bits.SSP1IE = 1;            // MSSP (I2C) interrupt enabled
    PIR1bits.SSP1IF = 0;            // Clear interrupt flag
    INTCONbits.IOCIE = 0;           // Interrupts on change disabled (enable
when using encoder mode)
    IOCAN = 0;                      // Disable all interrupts on falling pin
changes
    IOCANbits.IOCAN0 = 1;           // Enable interrupts on falling A0 (Encoder
line for B)
//  IOCANbits.IOCAN1 = 1;           // Enable interrupts on falling A1 (Encoder
line for A)
    IOCAP = 0;                      // Disable all interrupts on rising pin
changes
    IOCAPbits.IOCAP0 = 1;           // Enable interrupts on rising A0 (Encoder
line for B)
//  IOCAPbits.IOCAP1 = 1;           // Enable interrupts on rising A1 (Encoder
line for A)
    INTCONbits.IOCIF = 0;           // Clear interrupt flag

    Conf_Timer0();  //Configuramos timer0 como temporizador

    /* Read EEPROM settings */
    // Read the I2C address
    EEADRL = EEPROM_I2C_ADDRESS;
    EECON1bits.EEPGD = 0;
    EECON1bits.CFGS = 0;
    EECON1bits.RD = 1;
    value = EEDATL;
    // Check against limits before setting, if out of range keep with the
default
    if ((value > 0x02) && (value < 0x78)) {
        SSP1ADD = value << 1;
    }
}
void SetAllMotors(bool reverse, int pwm) {
    // Check if the EPO has been tripped
    if (epoTripped) {
        reverse = false;
        pwm = 0;
    }
    // Work out the required settings
    if (pwm > PWM_MAX) pwm = PWM_MAX;
    if (reverse) {
        pwm = PWM_MAX - pwm;            // PWM pulse width inverted for reverse
        LATC = PORTC | (_LATC_LATC2_MASK | _LATC_LATC4_MASK);   // Motor A:2 and
B:2 On
    } else {
        // PWM pulse width is not inverted for forward
        LATC = PORTC & ~(_LATC_LATC2_MASK | _LATC_LATC4_MASK);  // Motor A:2 and
B:2 On
    }
    // Wait until it is safe to change the pulse widths
    PIR1bits.TMR2IF = 0;            // Clear Timer 2 interrupt flag
    while (PIR1bits.TMR2IF == 0) ;  // Wait for Timer 2 to loop
    // Set Motor A:1 and B:1 to the pulse width
    CCPR2L = pwm >> 2;              // Set PWM 2 MSBs
    CCPR1L = pwm >> 2;              // Set PWM 1 MSBs
    CCP2CONbits.DC2B = pwm & 0b11;  // Set PWM 2 LSBs
    CCP1CONbits.DC1B = pwm & 0b11;  // Set PWM 1 LSBs
}
```

```
void SetEncoderMode(bool enabled) {
    // Disable any automatic routines
    encMode = false;
    movingA = false;
    movingB = false;

    // Turn off both motors
    SetAllMotors(false, 0);

    // Set the correct mode
    encMode = enabled;
    if (encMode) {
        INTCONbits.IOCIE = 1;           // Interrupts on change enabled
    } else {
        INTCONbits.IOCIE = 0;           // Interrupts on change disabled
    }
}


void ControlMotor(){
    float Error0;
    int pwm2;
//    float Ref_act;

//    float mor;
//    float pwm1;
    bool sentidom;

    Error0=(float)(Referencia-Pulsos_seg);

    if(Ref_ant<0.0 && Referencia>0.0) Error1=0.0;
    if(Ref_ant>0.0 && Referencia<0.0) Error1=0.0;

    Error1+=Error0*Ki;

    if(Error1>UMBRAL_INTEGRAL)  Error1=UMBRAL_INTEGRAL;         //satura
errorv[1] a umbral_integral si esta por encima
    else if(Error1<-UMBRAL_INTEGRAL) Error1=-UMBRAL_INTEGRAL;   //y a -
umbral_integral si esta por debajo (5000)

    vm=(int)((Pulsos_seg*Kmot)+(Kp*Error0)+Error1);     //calculo del
proporcional integral

//    vm=(int)(mor);
    //Recorte del valor del pwm
    if(vm>LIMVMMAX) vm=LIMVMMAX;                                //satura vm a los
valores de limite velocidad motor maxima
    else if(vm<-LIMVMMAX)   vm=-LIMVMMAX;
    if((vm>0)&&(vm<MIN_MOV)) vm=MIN_MOV;                //satura vm a los
valores del minimo movimiento del motor
    else if ((vm<0)&&(vm>-MIN_MOV)) vm=-MIN_MOV;

    vm_ant=vm;
    if(vm<0) sentidom=false;
    else sentidom=true;

    if(vm<0) vm=-vm;   //cogemos los valores en positivo
```

```
    pwm2=(int)(1/1180.0*((float)vm)*254.0); //tranformamos a valores de pwm
//    pwm2=(int)pwm1;

    if(Referencia==0){
        pwm2=0;
        Error1=0.0;
    }
    SetAllMotors(sentidom, pwm2);

    Ref_ant=(float)Referencia;

}

void ProcessI2C(int len) {
    int i;
    bool echo = false;
    float datof;

    // Clear the reply buffer
    for (i = 0; i < I2C_MAX_LEN; ++i) {
        i2cSend[i] = 0;
    }

    if (len < 2) return;

    if (i2cRecv[0] == 0x00) {
        // Broadcast message, only used for setting up the I2C address
        if ((i2cRecv[1] != COMMAND_SET_I2C_ADD) &&
            (i2cRecv[1] != COMMAND_GET_ID)) {
            return;
        }
    }
      case COMMAND_ALL_OFF:
            SetAllMotors(false, 0);
            LATAbits.LATA4 = 1;     // LED Off
            movingA = false;
            movingB = false;
            Referencia=0;
            Control=false;
            echo = true; len = 3;
            break;
        case COMMAND_RESET_EPO:
            epoTripped = false;
            echo = true;
            break;
        case COMMAND_GET_EPO:
            i2cSend[0] = COMMAND_GET_EPO;
            if (epoTripped) {
                i2cSend[1] = COMMAND_VALUE_ON;
            } else {
                i2cSend[1] = COMMAND_VALUE_OFF;
            }
            break;
```

```
        case COMMAND_GET_EPO_PRESSED:
            i2cSend[0] = COMMAND_GET_EPO_PRESSED;
            if (PORTAbits.RA5 == 1) {
                i2cSend[1] = COMMAND_VALUE_ON;
            } else {
                i2cSend[1] = COMMAND_VALUE_OFF;
            }
            break;

        case COMMAND_GET_DRIVE_FAULT:
            i2cSend[0] = COMMAND_GET_DRIVE_FAULT;
            if (PORTAbits.RA2 == 1) {
                i2cSend[1] = COMMAND_VALUE_OFF;
            } else {
                i2cSend[1] = COMMAND_VALUE_ON;
            }
            break;
        case COMMAND_SET_ALL_FWD:
            if (len < 3) break;
            SetAllMotors(false, (int)i2cRecv[2]);
            echo = true; len = 3;
            break;
        case COMMAND_SET_ALL_REV:
            if (len < 3) break;
            SetAllMotors(true, (int)i2cRecv[2]);
            echo = true; len = 3;
            break;
        case COMMAND_SET_FAILSAFE:
            if (len < 3) break;
            if (i2cRecv[2] == COMMAND_VALUE_OFF) {
                T4CONbits.TMR4ON = 0;           // Timer 4 disabled
            } else {
                T4CONbits.TMR4ON = 1;           // Timer 4 enabled
            }
            PIR3bits.TMR4IF = 0;            // Clear Timer 4 interrupt flag
            echo = true; len = 3;
            break;
        case COMMAND_GET_FAILSAFE:
            i2cSend[0] = COMMAND_GET_FAILSAFE;
            if (T4CONbits.TMR4ON == 0) {
                i2cSend[1] = COMMAND_VALUE_OFF;
            } else {
                i2cSend[1] = COMMAND_VALUE_ON;
            }
            break;
        case COMMAND_SET_ENC_MODE:
            if (len < 3) break;
            if (i2cRecv[2] == COMMAND_VALUE_OFF) {
                SetEncoderMode(false);
            } else {
                SetEncoderMode(true);
            }
            echo = true; len = 3;
            break;
```

```
        case COMMAND_GET_ENC_MODE:
            i2cSend[0] = COMMAND_GET_ENC_MODE;
            if (encMode) {
                i2cSend[1] = COMMAND_VALUE_ON;
            } else {
                i2cSend[1] = COMMAND_VALUE_OFF;
            }
            break;
        case COMMAND_SET_KMOT:
            echo = true; len = 3;
            Kmot=(float)i2cRecv[2]/10.0;
            break;
        case COMMAND_SET_KP:
            echo = true; len = 3;
            Kp=(float)i2cRecv[2]/10.0;
            break;
        case COMMAND_SET_KI:
            echo = true; len = 3;
            Ki=(float)i2cRecv[2]/10.0;
            break;
        case COMMAND_GET_KMOT:
            i2cSend[0] = COMMAND_GET_KMOT;
            datof=Kmot*10.0;

            i2cSend[1] = (int)datof;
            break;
        case COMMAND_GET_KP:
            i2cSend[0] = COMMAND_GET_KP;
            datof=Kp*10.0;
            i2cSend[1] = (int)datof;
            break;
        case COMMAND_GET_KI:
            i2cSend[0] = COMMAND_GET_KI;
          datof=Ki*10.0;
            i2cSend[1] = (int)datof;
            break;
        case COMMAND_GET_ENC_MOVING:
            i2cSend[0] = COMMAND_GET_ENC_MOVING;
            if (movingA || movingB) {
                i2cSend[1] = COMMAND_VALUE_ON;
            } else {
                i2cSend[1] = COMMAND_VALUE_OFF;
            }
            break;
        case COMMAND_SET_ENC_SPEED:
            if (len < 3) break;
            if (i2cRecv[2] > PWM_MAX) {
                encLimit = PWM_MAX;
            } else {
                encLimit = i2cRecv[2];
            }
            echo = true; len = 3;
            break;
        case COMMAND_GET_ENC_SPEED:
            i2cSend[0] = COMMAND_GET_ENC_SPEED;
              break;
        case COMMAND_GET_ID:
            i2cSend[0] = COMMAND_GET_ID;
            i2cSend[1] = I2C_ID_PICOBORG_REV;
            break;
```

```
        case COMMAND_SET_I2C_ADD:
            // Set the live I2C address
            SSP1ADD = i2cRecv[2] << 1;
            // Save the I2C address to EEPROM
            PIR2bits.EEIF = 0;
            EECON1bits.WREN = 1;
            EEADRL = EEPROM_I2C_ADDRESS;
            EEDATL = i2cRecv[2];
            EECON2 = 0x55;
            EECON2 = 0xAA;
            EECON1bits.WR = 1;
            while (PIR2bits.EEIF == 0) ;
            PIR2bits.EEIF = 0;
            EECON1bits.WREN = 0;
            echo = true; len = 3;
            break;
        case COMMAND_SET_SPEED_FBCK:
            echo = true; len = 3;
            Referencia=((int)i2cRecv[2] << 8) + (int)i2cRecv[3];
            Control=true;
            movingB = true;
            movingA = true;

            failsafeCounter = 0;
           // sCountEnc=0;    //Prueba para contar pulsos
           // CountEnc=0;

            break;
        case COMMAND_GET_SPEED_FBCK:
            i2cSend[0]=COMMAND_GET_SPEED_FBCK;

          // ContaPulsos_env=sCountEnc;
//           ContaPulsos_env=Pulsos_seg;
           if(Pulsos_seg<0){
                Pulsos_seg=65535+Pulsos_seg;
            }
            i2cSend[2] = Pulsos_seg & 0xFF;
            i2cSend[1] = (Pulsos_seg>>8) & 0xFF;
            Pulsos_seg=0;

            break;
        case COMMAND_GET_ALARMS:
            i2cSend[0] = COMMAND_GET_ALARMS;
            i2cSend[1] = ((epoTripped * 0x01) + (PORTAbits.RA5 * 0x02) +
(encoderFail * 0x04) + (lowVoltage * 0x08)) & 0xFF;
            break;
        default:
            // Do nothing
            break;
    }

    // Echo sent command if required
    if (echo) {
        for (i = 1; i < len; ++i) {
            i2cSend[i - 1] = i2cRecv[i];
        }
    }
}
```

Configuration_bits

```
 // PIC16F1824 Configuration Bit Settings


// 'C' source line config statements

#include <xc.h>

// #pragma config statements should precede project file includes.
// Use project enums instead of #define for ON and OFF.

// CONFIG1
#pragma config FOSC = INTOSC    // Oscillator Selection (INTOSC oscillator: I/O
function on CLKIN pin)
//#pragma config WDTE = ON        // Watchdog Timer Enable (WDT enabled)
#pragma config WDTE = OFF
#pragma config PWRTE = ON       // Power-up Timer Enable (PWRT enabled)
#pragma config MCLRE = ON       // MCLR Pin Function Select (MCLR/VPP pin
function is MCLR)
#pragma config CP = OFF         // Flash Program Memory Code Protection (Program
memory code protection is disabled)
#pragma config CPD = OFF        // Data Memory Code Protection (Data memory code
protection is disabled)
#pragma config BOREN = ON       // Brown-out Reset Enable (Brown-out Reset
enabled)
#pragma config CLKOUTEN = OFF   // Clock Out Enable (CLKOUT function is
disabled. I/O or oscillator function on the CLKOUT pin)
#pragma config IESO = OFF       // Internal/External Switchover
(Internal/External Switchover mode is disabled)
#pragma config FCMEN = ON       // Fail-Safe Clock Monitor Enable (Fail-Safe
Clock Monitor is enabled)

// CONFIG2
#pragma config WRT = OFF        // Flash Memory Self-Write Protection (Write
protection off)
#pragma config PLLEN = ON       // PLL Enable (4x PLL enabled)
#pragma config STVREN = ON      // Stack Overflow/Underflow Reset Enable (Stack
Overflow or Underflow will cause a Reset)
#pragma config BORV = LO        // Brown-out Reset Voltage Selection (Brown-out
Reset Voltage (Vbor), low trip point selected.)
//#pragma config LVP = ON         // Low-Voltage Programming Enable (Low-voltage
programming enabled)
#pragma config LVP = OFF
```

Interrupts

```
/*****************************************************************************/
/*Files to Include                                                           */
/*****************************************************************************/

#if defined(__XC)
    #include <xc.h>         /* XC8 General Include File */
#elif defined(HI_TECH_C)
    #include <htc.h>        /* HiTech General Include File */
#endif

#include <stdint.h>         /* For uint8_t definition */
#include <stdbool.h>        /* For true/false definition */

#include "user.h"

/*****************************************************************************/
/* Interrupt Routines                                                        */
/*****************************************************************************/

/* Baseline devices don't have interrupts. Note that some PIC16's
 * are baseline devices.  Unfortunately the baseline detection macro is
 * _PIC12 */
#ifndef _PIC12

void interrupt isr(void) {
    /* This code stub shows general interrupt handling.  Note that these
    conditional statements are not handled within 3 seperate if blocks.
    Do not use a seperate if block for each interrupt flag to avoid run
    time errors. */

    // Check for I2C events
    if (PIR1bits.SSP1IF) {
        PIR1bits.SSP1IF = 0;            // Clear the interrupt

        // Analyse interrupt
        if (SSP1STATbits.P) {
            // Stop condition, process command
            if (SSP1STATbits.R_nW) {
                // Master reading mode, do nothing
            } else {
                // Master writing mode, analyse the command
                if (i2cByte > I2C_MAX_LEN) i2cByte = I2C_MAX_LEN;
                ProcessI2C(i2cByte);
            }
            i2cByte = 0;
            //failsafeCounter = 0;
        } else if (i2cByte < I2C_MAX_LEN) {
            // Within length limit
            if (SSP1STATbits.R_nW) {
                // Master reading mode, forward the data bytes
                SSP1BUF = i2cSend[i2cByte];
            } else {
                // Master writing mode, read the transmitted byte
                i2cRecv[i2cByte] = SSP1BUF;
            }
            ++i2cByte;
        }
```

```
else {
                // Master writing mode, discard the transmitted byte
                junk = SSP1BUF;
            }
        }

        SSP1CON1bits.CKP = 1;           // Release clock line
    }

    // Check for pin change events
    if (INTCONbits.IOCIF) {
       // LATAbits.LATA4 = 0;
        INTCONbits.IOCIF = 0;           // Clear the interrupt
         //lectura del encoder
         if (IOCAFbits.IOCAF0) {
            if(PORTAbits.RA0 == 0){
               if(PORTAbits.RA1 == 0) CountEnc++; //--Si el canalB esta a 0
               else CountEnc--;
            }else{
                if(PORTAbits.RA1 == 1) CountEnc++;
                else CountEnc--;

            }
            IOCAFbits.IOCAF0 = 0;
         //   LATAbits.LATA4 = 1;
         }

    }

    if(INTCONbits.T0IF==1){       //interrupcion del timer0 para calcular
100mseg
        CountTim++;
        if(CountTim>=25){
            flagTim=1;
            CountTim=0;
        }
        INTCONbits.T0IF=0;
    }

}
#endif
```

ANEXO D: ROS WORKING SPACE

Edubot_control_main

```
#!/usr/bin/env python

import rospy

from std_msgs.msg import String
from nav_msgs.msg import Odometry
from geometry_msgs.msg import Point, Quaternion, Twist
import tf
from base_control_msgs.msg import *

import time
import os
import numpy as np;
import json
from array import *
from datetime import timedelta
from threading import Thread

# Import library PicoBorgRev
import PicoBorgRevEnc as PicoBorgRev
import UltraBorg

from diffdrive_controller import *

from gpio_interface import *

#ALARMS

firmware_version='raspi_controller'

Address_Right_Motor=11
Address_Left_Motor=10
Address_Forward_sonars=12
Address_Backward_sonars=13
stepDelay = 0.3                     # Number of seconds between each sequence
step
stepDelay_sonar = 0.3

b_simulation =False

global enable_UB_F    #Enabled/disabled ultrasonic reading
global enable_UB_B

SONAR_DIST_MIN= 10
SONAR_DIST_MAX= 5000

global ref_vel_right,ref_vel_left
global current_state
global diff_ctrl
```

```
def simulation():


    sim_state=Control_state()
    sim_state.contr_name=firmware_version
    sim_state.volt_values=24.5
    sim_state.axes=[]
    #axes[0] motor right, axes[1] motor left
    temp_axes=AxisData()
    temp_axes.vel=ref_vel_right
    temp_axes.odo=0
    temp_axes.cal=True
    sim_state.axes.append(temp_axes)
    temp_axes.vel=ref_vel_left
    sim_state.axes.append(temp_axes)
    sim_state.digout_state=current_digout_state
    sim_state.alarms=current_alarms
    return sim_state


def sonar_simulation():

    sim_sonar_state=Sonar_state()

    sim_sonar_state.son_num=8
    sim_sonar_state.son_values=[]
    for i in range(0, sim_sonar_state.son_num):
        sim_sonar_state.son_values.append(50)
    sim_sonar_state.bmp_num=0
    sim_sonar_state.bmp_state=[]
    return sim_sonar_state

def callback_command(msg):
    print 'callback_command:: msg',msg
    if msg.data=='RESET':
        PBR_R.ResetEpo()
        PBR_L.ResetEpo()
        print 'callback_command:: RESET'


def callback_test_vel_command_pulses(msg):

    global ref_vel_right,ref_vel_left


    print 'callback_vel_command::msg(',msg.linear.x,'',msg.angular.z,')'
    ref_vel_right=int(msg.linear.x)
    ref_vel_left=int(msg.linear.x)

    PBR_L.SetSpeedMotor(ref_vel_left)
    PBR_R.SetSpeedMotor(ref_vel_right)

    print 'vel INT(left=',ref_vel_left,',right=',ref_vel_right,')'
```

```
def callback_vel_command(msg):

    global ref_vel_right,ref_vel_left


    print 'callback_vel_command::msg(',msg.linear.x,'',msg.angular.z,')'


    ref_vel_right,ref_vel_left=diff_ctrl.command_velocity(msg)

    PBR_L.SetSpeedMotor(ref_vel_left)
    PBR_R.SetSpeedMotor(ref_vel_right)


    print 'vel INT(left=',ref_vel_left,',right=',ref_vel_right,')'


def pub_odometry(pose):


    # Construct odometry message
    odom_msg = Odometry()

    odom_msg.header.stamp = rospy.Time.now()
    odom_msg.header.frame_id = frame_id
    odom_msg.child_frame_id = child_frame_id
    odom_msg.pose.pose.position = Point(pose['x'], pose['y'], 0)
    odom_msg.pose.pose.orientation =
Quaternion(*tf.transformations.quaternion_from_euler(0,0,pose['th']))
    #    P = numpy.mat(numpy.diag([0.0]*3)) # Dummy covariance
    #    odom_msg.pose.covariance = tuple(P.ravel().tolist())
    pub_odom.publish(odom_msg)

def pub_tf(pose):
    tf_broadcaster = tf.TransformBroadcaster()
    tf_broadcaster.sendTransform( \
                               (pose['x'], pose['y'], 0), \

tf.transformations.quaternion_from_euler(0,0,pose['th']), \
                               rospy.Time.now(), \
                               child_frame_id, \
                               frame_id \
                               )



def command_control():

    global ref_vel_right,ref_vel_left
    #print 'command_control'

    #PBR_L.SetSpeedMotor(ref_vel_left)
    #PBR_R.SetSpeedMotor(ref_vel_right)

    #ref_vel_right = 0
    #ref_vel_left = 0
```

```
def read_control():

    print 'read_control'

    encoder_R=PBR_R.GetEncoderSpeed()
    print "encoder right=", encoder_R

    encoder_L=PBR_L.GetEncoderSpeed()
    print "encoder left=", encoder_L
    if encoder_R==None:
        encoder_R=0

    if encoder_L==None:
        encoder_L=0

#     PBR_R.SetSpeedMotor(150)
#     PBR_L.SetSpeedMotor(100)
    global current_state

    current_state.header.stamp = rospy.Time.now()
    current_state.header.frame_id = 'base_link'
    current_state.contr_name=firmware_version
#     current_state.volt_values=gpio.battery_level()
    current_state.volt_values=12.0
    #axes[0] motor right, axes[1] motor left
    temp_axesR=AxisData()
    temp_axesR.vel=encoder_R
    temp_axesR.odo=0
    temp_axesR.cal=True

    #current_state.axes.append(temp_axesR)
    current_state.axes[0]=temp_axesR
    temp_axesL=AxisData()

    temp_axesL.vel=encoder_L
    temp_axesL.cal=True
    temp_axesL.odo=0

    #current_state.axes.append(temp_axesL)
    current_state.axes[1]=temp_axesL
    current_alarms=read_alarms()
    current_state.alarms=current_alarms
    current_state.digout_state=read_digout_state()



def read_digout_state():
    #[ready, alarms, battery, empty]
    temp_digout=[0,0,0,0]

    for i in range(0, 9):
        if current_state.alarms[i]==True:
            temp_digout[1]=1
            PBR_R.SetSpeedMotor(0)
            PBR_L.SetSpeedMotor(0)



    return temp digout
```

```
def read_alarms():
   #[com_R, com_L, driver_R, driver_L, emergency_R, emergency_L, disabled_emg_R,
disabled_emg_L, encoder_R, encoder_L]

   #[com_R, com_L, driver_R, driver_L, preset_emg_R, preset_emg_L, Stop_R,
Stop_L, encoder_R, encoder_L]
    temp_alarms=[True,True,False,False,False,False,False,False,False,False]
    temp_alarms_r=PBR_R.GetAlarms()
    temp_alarms_l=PBR_L.GetAlarms()

#    bin_alarm=int(bin(temp_alarms_r)[2:])
#    print 'alarms_r int=',bin_alarm

#    str_alarm=list(str(bin(temp_alarms_r)[2:]))
#    print 'str_alarm=',str_alarm

    if temp_alarms_r!=-1:
        temp_alarms[0]=False
    if temp_alarms_r==768:
        temp_alarms[4]=True
        temp_alarms[6]=True
    elif temp_alarms_r==256:
        temp_alarms[4]=False
        temp_alarms[6]=True
    elif temp_alarms_r==1792:
        temp_alarms[2]=True
        temp_alarms[4]=True
        temp_alarms[6]=True
    elif temp_alarms_r==1280:
        temp_alarms[2]=True
        temp_alarms[4]=False
        temp_alarms[6]=True
    elif temp_alarms_r==1024:
        temp_alarms[8]=True
        temp_alarms[6]=True

    if temp_alarms_l!=-1:
        temp_alarms[1]=False
    if temp_alarms_l==768:
        temp_alarms[5]=True
        temp_alarms[7]=True
    elif temp_alarms_l==256:
        temp_alarms[5]=False
        temp_alarms[7]=True
    elif temp_alarms_l==1792:
        temp_alarms[3]=True
        temp_alarms[5]=True
        temp_alarms[7]=True
    elif temp_alarms_l==1280:
        temp_alarms[3]=True
        temp_alarms[5]=False
        temp_alarms[7]=True
    elif temp_alarms_l==1024:
        temp_alarms[9]=True
        temp_alarms[7]=True

print 'alarms_r=',temp_alarms_r
    print 'alarms_l=',temp_alarms_l
    return temp_alarms
```

```
def read_sonar():



    sim_sonar_state=Sonar_state()

    sim_sonar_state.son_num=8
    sim_sonar_state.son_values=[]

    # Read all four ultrasonic values
    # Convert to the nearest millimeter
    #sim_sonar_state.son_values.append(int(UB_F.GetDistance1()))
    if enable_UB_F:


sim_sonar_state.son_values.append(normalized_read_sonar(UB_F.GetDistance1()))

sim_sonar_state.son_values.append(normalized_read_sonar(UB_F.GetDistance2()))

sim_sonar_state.son_values.append(normalized_read_sonar(UB_F.GetDistance3()))

sim_sonar_state.son_values.append(normalized_read_sonar(UB_F.GetDistance4()))


    else:

        sim_sonar_state.son_values.append(SONAR_DIST_MAX)
        sim_sonar_state.son_values.append(SONAR_DIST_MAX)
        sim_sonar_state.son_values.append(SONAR_DIST_MAX)
        sim_sonar_state.son_values.append(SONAR_DIST_MAX)

    if enable_UB_B:

sim_sonar_state.son_values.append(normalized_read_sonar(UB_B.GetDistance1()))

sim_sonar_state.son_values.append(normalized_read_sonar(UB_B.GetDistance2()))

sim_sonar_state.son_values.append(normalized_read_sonar(UB_B.GetDistance3()))

sim_sonar_state.son_values.append(normalized_read_sonar(UB_B.GetDistance4()))
    else:
        sim_sonar_state.son_values.append(SONAR_DIST_MAX)
        sim_sonar_state.son_values.append(SONAR_DIST_MAX)
        sim_sonar_state.son_values.append(SONAR_DIST_MAX)
        sim_sonar_state.son_values.append(SONAR_DIST_MAX)


#    for i in range(len(sim_sonar_state.son_values)):
#
#sim_sonar_state.son_values[i]=max(sim_sonar_state.son_values[i],SONAR_DIST_MIN)
#        print 'sonar_update
sim_sonar_state.son_values=',sim_sonar_state.son_values[i]

    sim_sonar_state.bmp_num=0
    sim_sonar_state.bmp_state=[]
    return sim_sonar_state
```

```
def normalized_read_sonar(dist):

    norm_dist=int(min(dist,SONAR_DIST_MAX))
    if norm_dist<SONAR_DIST_MIN:
        norm_dist=SONAR_DIST_MAX

    return norm_dist

def init():

    global current_state
    current_state=Control_state()

    current_state.alarms=[False,False,False,False,False,False]
    current_state.digout_state=[0,0,0,0]
    current_state.axes=[None,None]


    global ref_vel_right,ref_vel_left

    ref_vel_right=0
    ref_vel_left=0

    print "init"

def update():


    while not rospy.is_shutdown():



        if b_simulation:
            #current_state=simulation()
            pass

        else:
            read_control()
            if current_state.digout_state[1]==0:
                command_control()

        pub_msg_control_state.publish(current_state)


pose=diff_ctrl.pose_update(current_state.axes[0].vel,current_state.axes[1].vel)
        #print 'pose=',pose
        pub_odometry(pose)
        pub_tf(pose)
        time.sleep(stepDelay)                    # Wait between steps


    #Turn both motors off
    PBR_R.MotorsOff()
    PBR_L.MotorsOff()
    print 'Shutdown'
```

```
def sonar_update():

    while not rospy.is_shutdown():

        current_sonar_state=Sonar_state()

        if b_simulation:
            current_sonar_state=sonar_simulation()

        else:

            current_sonar_state=read_sonar()


        pub_msg_sonar_state.publish(current_sonar_state)
        time.sleep(stepDelay_sonar)                    # Wait between steps



if __name__ == "__main__":

    rospy.init_node('robot_control')

    rospy.Subscriber('base_control/vel_command',Twist,callback_vel_command)
    rospy.Subscriber('base_control/command',String,callback_command)
    pub_msg_control_state=rospy.Publisher('robot_control/state',
Control_state,queue_size=1)
    pub_msg_sonar_state=rospy.Publisher('robot_control/sonar_state',
Sonar_state,queue_size=1)

    pub_odom = rospy.Publisher('odom', Odometry, queue_size=10)

rospy.Subscriber('test/vel_command_pulses',Twist,callback_test_vel_command_pulse
s)

    global frame_id
    global child_frame_id


    frame_id = rospy.get_param('~frame_id','/odom')
    child_frame_id = rospy.get_param('~child_frame_id','/base_link')

    L = rospy.get_param('~robot_wheel_separation_distance', 0.42)
    R = rospy.get_param('~robot_wheel_radius', 0.071)
    k_rad_2_pulses = rospy.get_param('~k_rad_2_pulses', 1.1)     # variable to
map m/s to pulses/seg
    cte_rad_2_pulses = rospy.get_param('~cte_rad_2_pulses', 66) # 1 Complete
turn of wheel = 66 pulses
    v_max_meters = rospy.get_param('~v_max_meters', 1.5)
    w_max_rad = rospy.get_param('~w_max_rad', 1.57)
    lim_max_pulses = rospy.get_param('~lim_max_pulses', 300)
    lim_min_pulses = rospy.get_param('~lim_min_pulses', -100)


    global enable_UB_F
    global enable_UB_B

    enable_UB_F = rospy.get_param('~enable_ultrasonics_F', True)
    enable_UB_B = rospy.get_param('~enable_ultrasonics_B', True)
```

```
    if not b_simulation:

        # INIT
            # Setup the PicoBorg Reverse
        PBR_R = PicoBorgRev.PicoBorgRev()     # Create a new PicoBorg Reverse
object
        PBR_R.i2cAddress = Address_Right_Motor
        PBR_R.Init()                          # Set the board up (checks the
board is connected)
        PBR_R.ResetEpo()                      # Reset the stop switch (EPO)
state
                                             # if you do not have a switch across
the two pin header then fit the jumper

        PBR_R.SetCommsFailsafe(True)
        PBR_R.SetEncoderMoveMode(True)
        PBR_R.SetEpoIgnore(False)

        kmot=PBR_R.GetPIDVariables('KMOT')
        kp=PBR_R.GetPIDVariables('KP')
        ki=PBR_R.GetPIDVariables('KI')
        print 'PBR_R::Variables PID:: KMOT=',kmot,' KP=',kp,' KI=',ki

        PBR_R.SetPIDVariables('KMOT', 0.1)
        PBR_R.SetPIDVariables('KP', 0.8)
        PBR_R.SetPIDVariables('KI', 0.1)

        # Setup the PicoBorg Reverse
        PBR_L = PicoBorgRev.PicoBorgRev()     # Create a new PicoBorg Reverse
object
        PBR_L.i2cAddress = Address_Left_Motor
        PBR_L.Init()                          # Set the board up (checks the
board is connected)
        PBR_L.ResetEpo()                      # Reset the stop switch (EPO)
state
                                             # if you do not have a switch across
the two pin header then fit the jumper

        PBR_L.SetCommsFailsafe(True)
        PBR_L.SetEncoderMoveMode(True)
        PBR_L.SetEpoIgnore(False)

        kmot=PBR_L.GetPIDVariables('KMOT')
        kp=PBR_L.GetPIDVariables('KP')
        ki=PBR_L.GetPIDVariables('KI')
        print 'PBR_L::Variables PID:: KMOT=',kmot,' KP=',kp,' KI=',ki

#        PBR_L.SetPIDVariables('KMOT', 0.1)
#        PBR_L.SetPIDVariables('KP', 0.8)
#        PBR_L.SetPIDVariables('KI', 0.4)

        PBR_L.SetPIDVariables('KMOT', 0.1)
        PBR_L.SetPIDVariables('KP', 0.8)
        PBR_L.SetPIDVariables('KI', 0.1)
```

Diffdrive_controller

```
#!/usr/bin/python

import rospy
import roslib
import math
import numpy

# EDUBOT 2pi*RAD= 496 PULSES
# SACARINO 2pi*RAD= 416 PULSES

class CmdVelToDiffDriveMotors:
    def __init__(self, L, R,cte_rad_2_pulses, k_rad_2_pulses, v_max_meters,
w_max_rad, lim_max_pulses, lim_min_pulses):

        self.L = L     # robot_wheel_separation_distance
        self.R = R     # robot_wheel_radius
        self.rad_2_pulses =cte_rad_2_pulses  #cte map rad/s to pulses/seg
        self.k_rad_2_pulses =k_rad_2_pulses
        self.v_max_meters =v_max_meters
        self.w_max_rad =w_max_rad
        self.lim_max_pulses =lim_max_pulses
        self.lim_min_pulses =lim_min_pulses

        self.pose = {'x':0, 'y': 0, 'th': 0}
        self.time_prev_update = rospy.Time.now()

    # Compute angular velocity target
    def angularvel_2_tangentvel(self,angular_vel):
        tangent_vel = angular_vel * self.R
        return tangent_vel

    # Compute tangent velocity target
    def tangentvel_2_angularvel(self,tangentvel):
        # v = wr
        # v - tangential velocity (m/s)
        # w - angular velocity (rad/s)
        # r - radius of wheel (m)
        angular_vel = tangentvel / self.R
        return angular_vel


    def command_velocity(self, vel_twist):

        # Suppose we have a target velocity v and angular velocity w
        # Suppose we have a robot with wheel radius R and distance between
wheels L
        # Let vr and vl be angular wheel velocity for right and left wheels,
respectively
        # Relate 2v = R (vr +vl) because the forward speed is the sum of the
combined wheel velocities
        # Relate Lw = R (vr - vl) because rotation is a function of counter-
clockwise wheel speeds
        # Compute vr = (2v + wL) / 2R
        # Compute vl = (2v - wL) / 2R


        target_v = vel_twist.linear.x
        target_w = vel_twist.angular.z
```

```
#         vr = (2*target_v + target_w*self.L) / (2)
#         vl = (2*target_v - target_w*self.L) / (2)

        vr = (2*target_v + target_w*self.L) / (2)
        vl = (2*target_v - target_w*self.L) / (2)


        print 'command_velocity::(',vl,'',vr,')'
        vr_tangent=self.tangentvel_2_angularvel(vr)
        vl_tangent=self.tangentvel_2_angularvel(vl)
        print 'command_velocity::(',vl,'',vr,')'

        # Mapping angular velocity targets to motor commands
        vr_tangent=vr_tangent*self.rad_2_pulses*self.k_rad_2_pulses
        vl_tangent=vl_tangent*self.rad_2_pulses*self.k_rad_2_pulses


    #
    #     if target_v>0.0:
    #         vr=min(vr,self.v_max_meters)
    #         vr=max(vr,0.0)
    #
    #         vl=min(vl,self.v_max_meters)
    #         vl=max(vl,0.0)
    #
    #     else:
    #         vr=max(vr,-self.v_max_meters)
    #         vr=min(vr,0.0)
    #
    #         vl=max(vl,-self.v_max_meters)
    #         vl=min(vl,0.0)

        # Transfom meters/s to pulse/s

        print 'command_velocity::vl_tangent(',vl_tangent,'',vl_tangent,')'


        vr_tangent=min(int(vr_tangent),self.lim_max_pulses)
        vr_tangent=max(int(vr_tangent),self.lim_min_pulses)

        vl_tangent=min(int(vl_tangent),self.lim_max_pulses)
        vl_tangent=max(int(vl_tangent),self.lim_min_pulses)

        return int(vr_tangent),int(vl_tangent)



    def pose_next(self, lwheel_tangent_vel_enc, rwheel_tangent_vel_enc):
        x = self.pose['x']; y = self.pose['y']; th = self.pose['th']
        time_curr_update = rospy.Time.now()
        dt = (time_curr_update - self.time_prev_update).to_sec()
        self.time_prev_update = time_curr_update

        # Special case where just moving straight
        if abs(rwheel_tangent_vel_enc - lwheel_tangent_vel_enc)< 0.001:
            v = (lwheel_tangent_vel_enc + rwheel_tangent_vel_enc) / 2.0
            w = 0
            x = x + v*dt*numpy.cos(th)
            y = y + v*dt*numpy.sin(th)
            print 'IF'
```

```
        else:
                    # Compute the instantaneous center of curvature
            v = (lwheel_tangent_vel_enc + rwheel_tangent_vel_enc) / 2.0
            w = (rwheel_tangent_vel_enc - lwheel_tangent_vel_enc) / self.L
            R = ( self.L / 2.0 ) * (lwheel_tangent_vel_enc +
rwheel_tangent_vel_enc) / (rwheel_tangent_vel_enc - lwheel_tangent_vel_enc)

            # Update robot pose
            translation = numpy.matrix([[x - R*numpy.sin(th)], [y +
R*numpy.cos(th)], [w*dt]])
            icc_pt = numpy.matrix([[R*numpy.sin(th)],[-R*numpy.cos(th)],[th]])
            rotation = numpy.matrix([[numpy.cos(w*dt), -numpy.sin(w*dt),
0],[numpy.sin(w*dt), numpy.cos(w*dt), 0],[0,0,1]])
            pose_next = rotation * icc_pt + translation
            print 'pose_next=',pose_next

            x = pose_next[0,0]
            y = pose_next[1,0]
            th = pose_next[2,0]

        self.pose['x']=x
        self.pose['y']=y
        self.pose['th'] = math.atan2(math.sin(th),math.cos(th)) # squash the
orientation to between -pi,pi
        print 'x=',x,'y=',y,'th=',th
        print 'self.pose[th]=',self.pose['th']
       # self.pose['x']=x; self.pose['y']=y;  self.pose['th']=th
        return {'x':x, 'y':y, 'th':th,'v':v,'w':w}

    def pose_update(self,rwheel_angular_vel_enc,lwheel_angular_vel_enc):




        lwheel_tangent_vel_enc =
self.angularvel_2_tangentvel(lwheel_angular_vel_enc)
        rwheel_tangent_vel_enc =
self.angularvel_2_tangentvel(rwheel_angular_vel_enc)

        l_vel_enc_rad= float(lwheel_tangent_vel_enc/float(self.rad_2_pulses))
        r_vel_enc_rad= float(rwheel_tangent_vel_enc/float(self.rad_2_pulses))

#        print 'pose_update::
(lwheel_tangent_vel_enc=',lwheel_tangent_vel_enc,',rwheel_tangent_vel_enc=',rwhe
el_tangent_vel_enc,')'
#
#        print 'pose_update::
(l_vel_enc_rad=',l_vel_enc_rad,',r_vel_enc_rad=',r_vel_enc_rad,')'

        pose_next = self.pose_next(l_vel_enc_rad, r_vel_enc_rad)


        return pose_next
```

Set_Adress

```
#!/usr/bin/env python
# coding: latin-1
import PicoBorgRevEnc as PicoBorgRev
import UltraBorg

#SetNewAddress(newAddress, [oldAddress], [busNumber])


#PicoBorgRev.SetNewAddress(11,68)
# Setup the PicoBorg Reverse

#scanp=PicoBorgRev.ScanForPicoBorgReverse()
#print 'scan PicoBorgRev=',scanp


scanu=UltraBorg.ScanForUltraBorg()
print 'scan UltraBorg=',scanu
#UltraBorg.SetNewAddress(13,12)
```

Edubot.launch

```
<launch>


  <node pkg="robot_control" type="edubot_control_main.py" name="robot_control"
respawn="true" cwd="node" output="screen">
   <param name="frame_id" type="string" value="/odom" />
    <param name="child_frame_id" type="string" value="/base_link"/>
   <param name="robot_wheel_separation_distance" value="0.41" />
   <param name="robot_wheel_radius" value="0.071" />
   <param name="v_max_meters"  value="1.5" />
   <param name="w_max_rad"  value="1.57" />
   <param name="k_rad_2_pulses" value="1.1" />
   <param name="cte_rad_2_pulses" type="int" value="66" />
  <param name="lim_max_pulses" type="int" value="500" />
  <param name="lim_min_pulses" type="int" value="-500" />
  <param name="enable_ultrasonics_F" type="bool" value="True" />
  <param name="enable_ultrasonics_B" type="bool" value="True" />
</node>

  <include file="$(find rosbridge_server)/launch/rosbridge_websocket.launch"/>


  <param name="robot_description" command="$(find xacro)/xacro.py '$(find
edubot_description)/urdf/edubot.xacro'"/>

  <!-- send fake joint values -->
  <node name="joint_state_publisher" pkg="joint_state_publisher"
type="joint_state_publisher">
    <param name="use_gui" value="False"/>
  </node>

  <!-- Combine joint values -->
  <node name="robot_state_publisher" pkg="robot_state_publisher"
type="state_publisher"/>

  <!-- Show in Rviz   -->
  <!--node name="rviz" pkg="rviz" type="rviz"/-->
  <!--node name="rviz" pkg="rviz" type="rviz" args="-d $(find
mybot_description)/launch/edubot.rviz"/-->


</launch>
```

Edubot_camara.launch

```
<launch>


  <node pkg="robot_control" type="edubot_control_main.py" name="robot_control"
respawn="true" cwd="node" output="screen">
   <param name="frame_id" type="string" value="/odom" />
      <!--param name="child_frame_id" type="string" value="/base_link"/-->
      <param name="child_frame_id" type="string" value="/base_frame"/>
   <param name="robot_wheel_separation_distance" value="0.41" />
   <param name="robot_wheel_radius" value="0.071" />
   <param name="v_max_meters"  value="1.5" />
   <param name="w_max_rad"  value="1.57" />
   <param name="k_rad_2_pulses" value="1.1" />
   <param name="cte_rad_2_pulses" type="int" value="66" />
  <param name="lim_max_pulses" type="int" value="500" />
  <param name="lim_min_pulses" type="int" value="-500" />
  <param name="enable_ultrasonics_F" type="bool" value="True" />
  <param name="enable_ultrasonics_B" type="bool" value="True" />
</node>

  <include file="$(find rosbridge_server)/launch/rosbridge_websocket.launch"/>


  <param name="robot_description" command="$(find xacro)/xacro.py '$(find
edubot_description)/urdf/edubot.xacro'"/>

  <!-- send fake joint values -->
  <node name="joint_state_publisher" pkg="joint_state_publisher"
type="joint_state_publisher">
    <param name="use_gui" value="False"/>
  </node>

  <!-- Combine joint values -->
  <node name="robot_state_publisher" pkg="robot_state_publisher"
type="state_publisher"/>

 <node pkg="picamera_node" type="picamera_main.py" name="picamera_node"
cwd="node" output="screen">
   <param name="pixel_format" type="string" value="yuyv" />
   <param name="image_width" type="int" value="320" />
   <param name="image_height" type="int" value="240" />
   <param name="framerate" type="int" value="10" />
  <param name="brightness" type="int" value="50" />
  <param name="enable_capture" type="bool" value="True" />
  <param name="show_img" type="bool" value="False" />
  </node>



  <!-- Show in Rviz   -->
  <!--node name="rviz" pkg="rviz" type="rviz"/-->
  <!--node name="rviz" pkg="rviz" type="rviz" args="-d $(find
mybot_description)/launch/edubot.rviz"/-->


</launch>
```

EDUBOT_DESCRIPTION(URDF)

```
<?xml version='1.0'?>

<robot name="edubot" xmlns:xacro="http://www.ros.org/wiki/xacro">

  <xacro:property name="cameraSize" value="0.05"/>
  <xacro:property name="cameraMass" value="0.1"/>

  <!--xacro:include filename="$(find edubot_description)/urdf/edubot.gazebo" /--
>
  <xacro:include filename="$(find edubot_description)/urdf/materials.xacro" />
  <xacro:include filename="$(find edubot_description)/urdf/macros.xacro" />

  <link name='chassis'>
    <pose>0 0 0.1 0 0 0</pose>

    <inertial>
      <mass value="15.0"/>
      <origin xyz="0.0 0 0.1" rpy=" 0 0 0"/>
      <inertia
          ixx="0.1" ixy="0" ixz="0"
          iyy="0.1" iyz="0"
          izz="0.1"
      />
    </inertial>

    <collision name='collision'>
      <geometry>
        <box size=".41 .35 .202"/>
      </geometry>
    </collision>

    <visual name='chassis_visual'>
      <origin xyz="0 0 0" rpy=" 0 0 0"/>
      <geometry>
    <mesh filename="package://edubot_description/meshes/CuerpoEdubot.stl"
scale="0.001 0.001 0.001">
        <!--box size=".41 .35 .202"/-->
      </geometry>
    </visual>

    <collision name='caster_collision'>
      <origin xyz="-0.14 0 -0.05" rpy=" 0 0 0"/>
      <geometry>
        <sphere radius="0.05"/>
      </geometry>
      <surface>
        <friction>
          <ode>
            <mu>0</mu>
            <mu2>0</mu2>
            <slip1>1.0</slip1>
            <slip2>1.0</slip2>
          </ode>
        </friction>
      </surface>
    </collision>
```

```
    <visual name='caster_visual'>
      <origin xyz="-0.14 0 -0.05" rpy=" 0 0 0"/>
      <geometry>
        <sphere radius="0.05"/>
      </geometry>
    </visual>

    <collision name='caster_front_collision'>
      <origin xyz="0.14 0 -0.05" rpy=" 0 0 0"/>
      <geometry>
        <sphere radius="0.05"/>
      </geometry>
      <surface>
        <friction>
          <ode>
            <mu>0</mu>
            <mu2>0</mu2>
            <slip1>1.0</slip1>
            <slip2>1.0</slip2>
          </ode>
        </friction>
      </surface>
    </collision>

    <visual name='caster_front_visual'>
      <origin xyz="0.14 0 -0.05" rpy=" 0 0 0"/>
      <geometry>
        <sphere radius="0.05"/>
      </geometry>
    </visual>

  </link>


  <link name="left_wheel">
    <!--origin xyz="0.1 0.13 0.1"      rpy="0 1.5707 1.5707"/-->
    <!--origin xyz="0.042 0.2125 -0.091" rpy="0 1.5707 1.5707"/-->
    <collision name="collision">
      <origin xyz="0 0 0" rpy="0 1.5707 1.5707"/>
      <geometry>
        <cylinder radius="0.085" length="0.038"/>
      </geometry>
    </collision>
    <visual name="left_wheel_visual">
      <origin xyz="0 0 0" rpy="0 1.5707 1.5707"/>
      <geometry>
        <cylinder radius="0.085" length="0.038"/>
      </geometry>
    </visual>
    <inertial>
      <origin xyz="0 0 0" rpy="0 1.5707 1.5707"/>
      <mass value="5"/>
      <inertia
        ixx=".1" ixy="0.0" ixz="0.0"
        iyy=".1" iyz="0.0"
        izz=".1"/>
    </inertial>
  </link>
```

```
<link name="right_wheel">
  <!--origin xyz="0.1 -0.13 0.1" rpy="0 1.5707 1.5707"/-->
  <!--origin xyz="0.042 -0.2125 -0.091" rpy="0 1.5707 1.5707"/-->
  <collision name="collision">
    <origin xyz="0 0 0" rpy="0 1.5707 1.5707"/>
    <geometry>
      <cylinder radius="0.085" length="0.038"/>
    </geometry>
  </collision>
  <visual name="right_wheel_visual">
    <origin xyz="0 0 0" rpy="0 1.5707 1.5707"/>
    <geometry>
      <cylinder radius="0.085" length="0.038"/>
    </geometry>
  </visual>
  <inertial>
    <origin xyz="0 0 0" rpy="0 1.5707 1.5707"/>
    <mass value="5"/>
    <inertia
      ixx=".1" ixy="0.0" ixz="0.0"
      iyy=".1" iyz="0.0"
      izz=".1"/>
  </inertial>
</link>


<joint type="continuous" name="left_wheel_hinge">
  <origin xyz="0.042 0.2125 -0.091" rpy="0 0 0"/>
  <!--origin xyz="0.1 0.13 0" rpy="0 1.5707 1.5707"/-->
  <child link="left_wheel"/>
  <parent link="chassis"/>
  <axis xyz="0 1 0" rpy="0 0 0"/>
  <limit effort="10000" velocity="1000"/>
  <joint_properties damping="1.0" friction="1.0"/>
</joint>

<joint type="continuous" name="right_wheel_hinge">
  <origin xyz="0.042 -0.2125 -0.091" rpy="0 0 0"/>
  <!--origin xyz="0.1 -0.13 0" rpy="0 1.5707 1.5707"/-->
  <child link="right_wheel"/>
  <parent link="chassis"/>
  <axis xyz="0 1 0" rpy="0 0 0"/>
  <limit effort="10000" velocity="1000"/>
  <joint_properties damping="1.0" friction="1.0"/>
</joint>

<link name="camera">
  <collision>
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <geometry>
      <box size="${cameraSize} ${cameraSize} ${cameraSize}"/>
    </geometry>
  </collision>

  <visual>
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <geometry>
      <box size="${cameraSize} ${cameraSize} ${cameraSize}"/>
    </geometry>
    <material name="blue"/>
  </visual>
```

```
    <inertial>
      <mass value="${cameraMass}" />
      <origin xyz="0 0 0" rpy="0 0 0"/>
      <box_inertia m="${cameraMass}" x="${cameraSize}" y="${cameraSize}"
z="${cameraSize}" />
      <inertia ixx="1e-6" ixy="0" ixz="0" iyy="1e-6" iyz="0" izz="1e-6" />
    </inertial>
  </link>

  <joint name="camera_joint" type="fixed">
    <axis xyz="0 1 0" />
    <origin xyz=".2 0 0" rpy="0 0 0"/>
    <parent link="chassis"/>
    <child link="camera"/>
  </joint>


  <joint name="hokuyo_joint" type="fixed">
    <axis xyz="0 1 0" />
    <origin xyz=".15 0 .1" rpy="0 0 0"/>
    <parent link="chassis"/>
    <child link="hokuyo"/>
  </joint>


  <link name="hokuyo">
    <collision>
      <origin xyz="0 0 0" rpy="0 0 0"/>
      <geometry>
    <box size="0.1 0.1 0.1"/>
      </geometry>
    </collision>

    <visual>
      <origin xyz="0 0 0" rpy="0 0 0"/>
      <geometry>
        <mesh filename="package://edubot_description/meshes/hokuyo.dae"/>
      </geometry>
    </visual>

    <inertial>
      <mass value="1e-5" />
      <origin xyz="0 0 0" rpy="0 0 0"/>
      <inertia ixx="1e-6" ixy="0" ixz="0" iyy="1e-6" iyz="0" izz="1e-6" />
    </inertial>
  </link>


</robot>
```

EDUBOT_NAVIGATION

base_local_planner_params.yaml

```
TrajectoryPlannerROS:

 max_vel_x: 0.5
 min_vel_x: 0.2
 max_vel_theta: 1.5
 min_in_place_vel_theta: 0.5

 acc_lim_theta: 1.5
 acc_lim_x: 3.5
 acc_lim_y: 3.5


 holonomic_robot: false
```

costmap_common_params.yaml

```
obstacle_range: 2.5
raytrace_range: 3.0
footprint: [[0.12, 0.23], [0.12, -0.23],[0.35, 0.23], [0.35, -0.23]]
#robot_radius: ir_of_robot
robot_radius: 0.5 # distance a circular robot should be clear of the obstacle
inflation_radius: 3.0

observation_sources: laser_scan_sensor #point_cloud_sensor

# marking - add obstacle information to cost map
# clearing - clear obstacle information to cost map
laser_scan_sensor: {sensor_frame: hokuyo, data_type: LaserScan, topic:
/mybot/laser/scan, marking: true, clearing: true}

#point_cloud_sensor: {sensor_frame: frame_name, data_type: PointCloud, topic:
topic_name, marking: true, clearing: true}
```

global_costmap_params.yaml

```
# static_map - True if using existing map

global_costmap:
  global_frame: odom
  robot_base_frame: chassis
  update_frequency: 10.0
  publish_frequency: 1.0
  resolution: 0.05
  static_map: true
  width: 10.0
  height: 10.0
```

local_costmap_params.yaml

```
local_costmap:
  global_frame: odom
  robot_base_frame: chassis
  update_frequency: 10.0
  publish_frequency: 1.0
  static_map: false
  rolling_window: true
  width: 6.0
  height: 6.0
  resolution: 0.05
```

gmapping_demo.launch

```
<?xml version="1.0"?>
<launch>
  <master auto="start"/>
  <param name="/use_sim_time" value="true"/>
  <!--- Run gmapping -->
  <node pkg="gmapping" name="slam_gmapping" type="slam_gmapping"
output="screen">
    <param name="delta" value="0.01"/>
    <param name="xmin" value="-20"/>
    <param name="xmax" value="20"/>
    <param name="ymin" value="-20"/>
    <param name="ymax" value="20"/>
    <!--remap from="scan" to="mybot/laser/scan"/-->
    <param name="base_frame" value="base_link" />

    <param name="linearUpdate" value="0.5"/>
    <param name="angularUpdate" value="0.436"/>
    <param name="temporalUpdate" value="-1.0"/>
    <param name="resampleThreshold" value="0.5"/>
    <param name="particles" value="80"/>

  </node>
 <!-- Show in Rviz   -->
  <node name="rviz" pkg="rviz" type="rviz" args="-d $(find
mybot_navigation)/rviz/mapping.rviz"/>
  <!--node name="rviz" pkg="rviz" type="rviz" args="-d $(find
mybot_navigation)/launch/myrobot.rviz"/-->

</launch>
```

navigation.launch

```
<?xml version="1.0"?>
<launch>


  <!-- Map server -->
  <arg name="map_file" default="$(find edubot_navigation)/maps/test_map.yaml"/>
  <node name="map_server" pkg="map_server" type="map_server" args="$(arg
map_file)" />

  <!-- Place map frame at odometry frame
  <node pkg="tf" type="static_transform_publisher" name="map_odom_broadcaster"
      args="0 0 0 0 0 0 map odom 100"/>
-->

  <!-- Localization -->
    <node pkg="amcl" type="amcl" name="amcl" output="screen">

          <!-- see param description on http://www.ros.org/wiki/amcl -->

        <!--overall filter -->
        <param name="min_particles" value="100"/>
        <param name="max_particles" value="5000"/>
        <param name="kld_err" value="0.01"/>
        <param name="kld_z" value="0.99"/>
        <param name="update_min_d" value="0.2"/>
        <param name="update_min_a" value="0.52"/>
        <param name="resample_interval" value="2"/>
        <param name="transform_tolerance" value="0.1"/>
        <param name="recovery_alpha_slow" value="0.0"/>
        <param name="recovery_alpha_fast" value="0.0"/>
        <param name="gui_publish_rate" value="10.0"/>

        <!-- laser model -->
        <param name="laser_max_beams" value="30"/>
        <param name="laser_z_hit" value="0.95"/>
        <param name="laser_z_short" value="0.1"/>
        <param name="laser_z_max" value="0.05"/>
        <param name="laser_z_rand" value="0.05"/>
        <param name="laser_sigma_hit" value="0.2"/>
        <param name="laser_lambda_short" value="0.1"/>
        <param name="laser_likelihood_max_dist" value="2.0"/>
        <param name="laser_model_type" value="likelihood_field"/>


        <!-- odometry model -->
        <param name="odom_model_type" value="diff"/>
        <param name="odom_alpha1" value="0.2"/>
        <param name="odom_alpha2" value="0.2"/>
        <param name="odom_alpha3" value="0.2"/>
        <param name="odom_alpha4" value="0.2"/>
        <param name="odom_frame_id" value="odom"/>
        <param name="base_frame_id" value="base_frame"/>
        <param name="global_frame_id" value="map" />

    </node>
```

```
<!-- Move base -->
  <node pkg="move_base" type="move_base" respawn="false" name="move_base_node"
output="screen">
     <rosparam file="$(find
edubot_navigation)/config_move_base_params/move_base_params.yaml"
command="load"/>
    <rosparam file="$(find
edubot_navigation)/config_move_base_params/costmap_common_params.yaml"
command="load" ns="global_costmap" />
    <rosparam file="$(find
edubot_navigation)/config_move_base_params/costmap_common_params.yaml"
command="load" ns="local_costmap" />
    <rosparam file="$(find
edubot_navigation)/config_move_base_params/local_costmap_params.yaml"
command="load" />
    <rosparam file="$(find
edubot_navigation)/config_move_base_params/global_costmap_params.yaml"
command="load" />
<rosparam file="$(find
edubot_navigation)/config_move_base_params/dwa_planner_ros.yaml" command="load"
/>

    <remap from="cmd_vel" to="base_control/vel_command"/>



  </node>

<node name="rviz" pkg="rviz" type="rviz" args="-d $(find
edubot_description)/rviz/amcl.rviz"/>

</launch>
```

edubot_teleop.launch

```
<?xml version="1.0"?>
<launch>
  <!-- turtlebot_teleop_key already has its own built in velocity smoother -->
  <node pkg="turtlebot_teleop" type="turtlebot_teleop_key"
name="turtlebot_teleop_keyboard"  output="screen">
    <param name="scale_linear" value="0.5" type="double"/>
    <param name="scale_angular" value="1.5" type="double"/>
    <remap from="turtlebot_teleop_keyboard/cmd_vel"
to="base_control/vel_command"/>
  </node>
</launch>
```

Picamera_main

```
#!/usr/bin/env python

import rospy
import std_srvs.srv
from std_msgs.msg import String
from sensor_msgs.msg import Image
# Ros Messages
from sensor_msgs.msg import CompressedImage
# We do not use cv_bridge it does not support CompressedImage in python
# from cv_bridge import CvBridge, CvBridgeError


import time
from picamera.array import PiRGBArray
from picamera import PiCamera

import sys

import cv2
import numpy as np;
#from matplotlib import pyplot as plt

from array import *

from dateutil import parser
from dateutil import tz
from datetime import datetime
from datetime import timedelta

from threading import Thread

global b_capture
global b_img_show


def callback_camera_start(req):

    print "callback_camera_start Response"

    global b_capture
    b_capture=True


    return std_srvs.srv.EmptyResponse();

    #return std_srvs.srv.TriggerResponse(True,'waypoint 1');

def callback_camera_stop(req):

    print "callback_camera_stop Response"
    #thread_capture.exit()
    global b_capture
    b_capture=False
    #return std_srvs.srv.TriggerResponse(True,'waypoint 1');
    return std_srvs.srv.EmptyResponse();
```

```
def capture():

    global b_capture

    print b_capture
    while not rospy.is_shutdown():

        if not b_capture:
            rospy.sleep(1)

        else:

            # capture frames from the camera
            for frame in camera.capture_continuous(rawCapture, format="bgr",
use_video_port=True):
                imggray = cv2.cvtColor(frame.array, cv2.COLOR_BGR2GRAY)
                if b_img_show:
                    # show the frame
                    cv2.imshow("Frame", imggray)
                    key = cv2.waitKey(1) & 0xFF

                msg = CompressedImage()
                msg.header.stamp = rospy.Time.now()
                msg.format = "jpeg"
                msg.data = np.array(cv2.imencode('.jpg', imggray)[1]).tostring()
                pub_img_compressed.publish(msg)
                #msg_frame = CvBridge()
                #pub_videoRaw.publish(msg_frame.cv2_to_imgmsg(imggray), "bgr8")

                # clear the stream in preparation for the next frame
                rawCapture.truncate(0)

                # if the `q` key was pressed, break from the loop
                #if key == ord("q"):
                #    break

                if rospy.is_shutdown():
                    break

                if not b_capture:
                    rawCapture.truncate(0)
                    break

    print "shutdown"


if __name__ == "__main__":
    rospy.init_node('picamera_node')


rospy.Service('picamera_node/camera_start',std_srvs.srv.Empty,callback_camera_st
art)

rospy.Service('picamera_node/camera_stop',std_srvs.srv.Empty,callback_camera_sto
p)
    pub_videoRaw = rospy.Publisher('picamera_node/image_raw', Image,
queue_size=1)
    pub_img_compressed =
rospy.Publisher("picamera_node/image_raw/compressed",CompressedImage,
queue_size=1)
    #pub_msg_array=rospy.Publisher('picamera_node/ir_array', IrArray)
```

```
    camera = PiCamera()
    #camera.resolution = (640,480)
    camera.sharpness = 0
    camera.contrast = 10
    #camera.brightness = 20
    camera.saturation = 0
    camera.ISO = 0
    camera.video_stabilization = False
    camera.exposure_compensation = 0
    camera.exposure_mode = 'auto'
    camera.meter_mode = 'average'
    camera.awb_mode = 'auto'
    camera.image_effect = 'none'
    camera.color_effects = None
    camera.rotation = 0
    camera.hflip = False
    camera.vflip = False
    camera.crop = (0.0, 0.0, 1.0, 1.0)
    camera.framerate = 32

    camera.resolution=(rospy.get_param('~image_width',
320),rospy.get_param('~image_height', 240))
    camera.framerate=rospy.get_param('~framerate', 30)
    camera.brightness=rospy.get_param('~brightness', 20)


    print "camera.brightness" ,camera.brightness
    rawCapture = PiRGBArray(camera, size=camera.resolution)

    global b_capture
    b_capture=rospy.get_param('~enable_capture', True)

    global b_img_show
    b_img_show=rospy.get_param('~show_img', True)

    thread_capture = Thread(target=capture)
    thread_capture.start()



    print "Started"
    rospy.spin()
```

Camara_publisher

```
#include <ros/ros.h>
#include <image_transport/image_transport.h>
#include <opencv2/highgui/highgui.hpp>
#include <cv_bridge/cv_bridge.h>
#include <sstream> // for converting the command line parameter to integer

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


  ros::init(argc, argv, "image_publisher");
  ros::NodeHandle nh;
  image_transport::ImageTransport it(nh);
  image_transport::Publisher pub = it.advertise("camera/image", 1);

  // Convert the passed as command line parameter index for the video device to
an integer

  // Check if video source has been passed as a parameter
  if(argv[1] == NULL)video_sourceCmd(0);
  else std::istringstream video_sourceCmd(argv[1]);
  int video_source;
  // Check if it is indeed a number
  if(!(video_sourceCmd >> video_source)) return 1;

  cv::VideoCapture cap(video_source);
  // Check if video device can be opened with the given index
  if(!cap.isOpened()) return 1;
  cv::Mat frame;
  sensor_msgs::ImagePtr msg;

  ros::Rate loop_rate(5);
  while (nh.ok()) {
    cap >> frame;
    // Check if grabbed frame is actually full with some content
    if(!frame.empty()) {
      msg = cv_bridge::CvImage(std_msgs::Header(), "bgr8", frame).toImageMsg();
      pub.publish(msg);
      cv::waitKey(1);
    }

    ros::spinOnce();
    loop_rate.sleep();
  }
}
```

Camara_siguelinea

```
#include <ros/ros.h>
#include <image_transport/image_transport.h>
#include <opencv2/highgui/highgui.hpp>
#include <cv_bridge/cv_bridge.h>
#include <robotsim/siguelinea.h>

#define FILA_SUP 0.7  //Fila superior a analizar, en % sobre el total
#define FILA_INF 0.80  //Fila inferior a analizar, en % sobre el total

using namespace std;
using namespace cv;

ros::Publisher pub; // Create a publisher object.

int calcCdg(uchar* ptrFila, int fila, int nc, Mat& colorFrame)
{
    float suma=0;
    float total=0;
    float cdg=0;
    for (int i=0; i<nc; i++)
    {
        suma += i*ptrFila[i];
        total += ptrFila[i];
        circle(colorFrame, Point(i, fila), 1, Scalar(255, 0, 0));
    }
    if(total>0)
        cdg=suma/total;
    else
        cdg=-1;

    return cdg;
}

void imageCallback(const sensor_msgs::ImageConstPtr& msg)
{
  robotsim::siguelinea linea_msg;
  try
  {
    //cv::imshow("view", cv_bridge::toCvShare(msg, "bgr8")->image);

    //cv::waitKey(30);

    Mat colorFrame=cv_bridge::toCvShare(msg, "bgr8")->image;
    Scalar color(0,0,255);

    int nf=colorFrame.rows;
    int nc=colorFrame.cols;

    Mat grisFrame;
    cvtColor(colorFrame, grisFrame, COLOR_BGR2GRAY); //transforma a niveles de
gris

    Mat binFrame;
    threshold(grisFrame, binFrame, 0, 255, THRESH_BINARY_INV | CV_THRESH_OTSU);
    //imshow("Bin", binFrame);
```

```
    uchar* ptrFila;
    float cdg_sup, cdg_inf;

    int filaSup= (int) (FILA_SUP * nf);
    ptrFila= binFrame.ptr<uchar>(filaSup); //apunta a la filaInf
    cdg_sup = calcCdg(ptrFila, filaSup, nc, colorFrame);



    int filaInf= (int) (FILA_INF * nf);
    ptrFila= binFrame.ptr<uchar>(filaInf); //apunta a la filaInf
    cdg_inf = calcCdg(ptrFila, filaInf, nc, colorFrame);

    if(cdg_sup> -1) circle(colorFrame, Point(cdg_sup, filaSup), 3, color);
    if(cdg_inf> -1) circle(colorFrame, Point(cdg_inf, filaInf), 3, color);

    //ROS_INFO_STREAM( "Superior: " << "(fila " << filaSup << ", col " <<
cdg_sup << ")");
    //ROS_INFO_STREAM( "Inferior: " << "(fila " << filaInf << ", col " <<
cdg_inf << ")" );

    //imshow("Resultado", colorFrame);


    float x1, x2, y1, y2, A, B, C, dist, ang;

    if( cdg_sup> -1 && cdg_inf> -1 && filaSup!=filaInf )
    {
        //Traslada al punto central de la última línea.
        y2 = nf-filaSup;
        y1 = nf-filaInf;
        x2 = cdg_sup-nc/2.;
        x1 = cdg_inf-nc/2.;
    //ROS_INFO_STREAM( "(x1, y1)= ( " << x1 << " , " << y1 << " )" );
    //ROS_INFO_STREAM( "(x2, y2)= ( " << x2 << " , " << y2 << " )" );

        //Recta; y distancia y ángulo
        A= y1-y2;
        B= x2-x1;
        C= x1*y2 + x1*y1 - x1*y1 - x2*y1;
        dist=fabs(C)/sqrt(A*A+B*B);
        ang =atan( (x2-x1)/(y2-y1) );
    }
    else
    {
        dist= -1;
        ang = -1;
    }

    ROS_INFO_STREAM("Distancia: " << dist << " Angulo: " << ang << " rad  (" <<
ang*180./3.141592 << " grad)");

    linea_msg.distancia=dist;
    linea_msg.angulo=ang;
    pub.publish(linea_msg);

  }
```

```
  catch (cv_bridge::Exception& e)
  {
    ROS_ERROR("Could not convert from '%s' to 'bgr8'.", msg->encoding.c_str());
  }
}

int main(int argc, char **argv)
{
  ros::init(argc, argv, "image_listener");
  ros::NodeHandle nh;
  cv::namedWindow("view");
  cv::startWindowThread();
  image_transport::ImageTransport it(nh);
  image_transport::Subscriber sub = it.subscribe("camera/image", 1,
imageCallback);

  pub = nh.advertise<robotsim::siguelinea>("distancialinea", 1000);// Asign
publisher object.
  ros::spin();
  cv::destroyWindow("view");
}
```

Seguir_linea

```
// Este programa da la distancia y angulo de la tortuga a una linea ax²+bx+c=0
#include <ros/ros.h>
#include <robotsim/siguelinea.h>  // For geometry_msgs::Twist
#include "geometry_msgs/Twist.h"
#include <stdlib.h> // For rand() and RAND_MAX
#include <std_srvs/Empty.h>


double Kd=0.15; // Ganancia controlador distancia
double Kg=4;    // Ganancia giro angular

ros::Publisher vel_pub;




void lineaCallback(const robotsim::siguelinea::ConstPtr & linea){
    double distancia,angulo;
    distancia=linea->distancia;
    angulo=linea->angulo;
    geometry_msgs::Twist msg;
    if(distancia==-1 && angulo==-1){
        ROS_INFO_STREAM_ONCE("Linea no encontrada");
        msg.linear.x=0;
        msg.linear.y=0;
        msg.linear.z=0;
        msg.angular.x=0;
        msg.angular.y=0;
        msg.angular.z=0;
    }else{

        msg.linear.x=0.5;
        msg.linear.y=0;
        msg.linear.z=0;
        msg.angular.x=0;
        msg.angular.y=0;
        msg.angular.z=Kd*distancia+Kg*angulo;

        ROS_INFO_STREAM("Velocidad:"<< " lineal=" << msg.linear.x << " angular="
<< msg.angular.z << "Kg*angulo" << Kg*angulo);
    }
        vel_pub.publish(msg);



}

int main(int argc, char **argv) {
  // Initialize the ROS system and become a node.
  ros::init(argc, argv, "sensorlinea");
  ros::NodeHandle nh;

  // Create a publisher object.
  ros::Subscriber sub = nh.subscribe("distancialinea", 100,lineaCallback);
  vel_pub =nh.advertise<geometry_msgs::Twist>("cmd_vel",100);
```

```
  //ros::service::waitForService("clear");
  //Obtiene valores de los parámetros Kd y Kg.
  const std::string PARAM_NAME1 = "~Kd";
  bool ok = ros::param::get("~Kd",Kd);
  if(!ok) {
    ROS_FATAL_STREAM("No puedo obtener parametro " << PARAM_NAME1);
    exit(1);
  }
   const std::string PARAM_NAME2 = "~Kg";
   ok = ros::param::get(PARAM_NAME2,Kg);
  if(!ok) {
    ROS_FATAL_STREAM("No puedo obtener parametro " << PARAM_NAME2);
    exit(1);
  }

  while(ros::ok()) {
    ros::spinOnce();

  }
  ROS_INFO_STREAM_ONCE("Nodo finalizado");
        geometry_msgs::Twist msg;
        msg.linear.x=0;
        msg.linear.y=0;
        msg.linear.z=0;
        msg.angular.x=0;
        msg.angular.y=0;
        msg.angular.z=0;
        vel_pub.publish(msg);


  return 0;
}
```

Camara_linea.launch

```
<launch>

    <node
        pkg="camara"
        type="camara_siguelinea"
        name="camara_siguelinea"
        required="true"
        output="screen"

    >
    <remap from="camera/image" to="/mybot/camera1/image_raw"/>
    </node>

    <node
        pkg="camara"
        type="seguir_linea"
        name="seguir_linea"
        required="false"
        output="screen"
    >
    <param name="Kd" value="0.005" />
    <param name="Kg" value="0.3" />
    </node>



</launch>
```

Joystick

```
<!DOCTYPE html>
<html>
<head>

    <title>EDUBOT</title>
    <meta charset="utf-8"></meta>
<!--    <meta name="viewport" content="width=device-width, initial-scale=1"> -->
  <meta name="viewport" content="initial-scale=1.0, user-scalable=no"></meta>
  <meta name="apple-mobile-web-app-capable" content="yes"></meta>
  <meta name="apple-mobile-web-app-status-bar-style" content="black"></meta>

<script src="js/jquery-1.7.2.min.js"></script>
<link rel="stylesheet" href="css/jquery.mobile-1.4.5.min.css" />

<script src="js/jquery.mobile-1.4.5.min.js"></script>

<script src="./js/jquery-ui-1.10.3.custom.js"></script>
<script src="./js/jquery-ui-1.10.3.js"></script>


<!-- JoyStick -->
<script type="text/javascript" src="./joystick/JoyStick.js"></script>
<link rel="stylesheet" href="./joystick/JoyStick.css" />

<!-- rosbridge -->

  <script type="text/javascript" src="js/WebSocketTest.js"></script>
    <script src="js/eventemitter2.js"></script>
    <script type="text/javascript" src="./js/mjpegcanvas.js"></script>
    <script src="js/roslib.js"></script>

<style type="text/css">
.ui-content{
padding-top: 5px;
padding-left: 10px;

}
.alert {
padding-top: 5px;
padding-right: 35px;
padding-bottom: 0px;
padding-left: 14px;

  margin-bottom: 2px;
  color: #c09853;
  background-color: #fcf8e3;
  border: 1px solid #fbeed5;
  border-radius: 4px;
}
```

```
#popupPanel {
    width: 0;
    height: 0;
    right: 0 !important;
    left: auto !important;

    position:absolute;
    border-right: none;
    background: black;
    margin: 2em 15px;
}

#panel_content {
    width: 400px;
    height: 340px;
    right: 0 !important;
    left: auto !important;
    position:absolute;
    top: 3%;
    border-right: none;
    background: black;
-webkit-border-radius: 25px;
-moz-border-radius: 25px;
border-radius: 25px;
}

#mjpeg{
-webkit-border-radius: 25px;
-moz-border-radius: 25px;
border-radius: 25px;
}

.ui-header .ui-title{
 padding:0px;
 min-height: 0px;
}

ui-listview > .ui-li-static {
    padding: 2px;
}

.ui-li-count{

min-width: 50px;
}


.ui-content .ui-listview-inset, .ui-panel-inner > .ui-listview-inset {
    margin: 2px 0;
}

.img_prev{

width:50px;
height:50px;
}
```

```
</style>

</head>

<body onload="start_ros();">
<div data-role="page" id="page_status">



<div data-role="header" class="ui-body-b " style="height:60px;">
  <h1>EDUBOT JOY</h1>
    <div id="console_html" class="alert" style="height:30px;">
        Esperando conexión con robot
    </div>
</div><!-- /header -->



<div data-role="content" class="ui-content">

<div class="ui-grid-b">
    <div class="ui-block-a">
    </div>
    <div class="ui-block-b">

        <ul id="list_status" data-role="listview" data-split-theme="a"
class="ui-field-contain" data-inset="true" >
            <li>
                <div id="joystick_nav" class="JoyStick" ></div>
            </li>
            <li>
                <div id="log_joy_nav" class="" ></div>
            </li>
            <li>
                <div id="alarm_status" class="" ></div>
            </li>
            <li>
                <button onclick="Command('RESET')">RESET</button>
            </li>
        </ul>
    </div>
    <div class="ui-block-c">
    </div>
</div>

</div><!-- /content -->

</div><!-- /page -->

<script>

var ros = null;

var pub_joy=null;
var pub_command=null;
var array_state=null;
var listener_task_status=null;
```

```
function start_ros(){


  // Connecting to ROS
  // -----------------
  ros = new ROSLIB.Ros();

  // If there is an error on the backend, an 'error' emit will be emitted.
  ros.on('error', function(error) {
    //console.log(error);
    $('#console_html').html('No hay conexión con el robot');
  });

// Find out exactly when we made a connection.
  ros.on('connection', function() {

    $('#console_html').html('Conectado: Recibiendo datos');

  });
  // Create a connection to the rosbridge WebSocket server.
   ros.connect('ws://localhost:9090');
   //ros.connect('ws://192.168.192.67:9090');
  // Publishing a Topic
  // ------------------

  pub_joy = new ROSLIB.Topic({
    ros : ros,
    name : '/base_control/vel_command',
    messageType : 'geometry_msgs/Twist'
  });

  pub_command = new ROSLIB.Topic({
    ros : ros,
    name : '/base_control/command',
    messageType : 'std_msgs/String'
  });

Subscribe()

}
function Subscribe(){
  //Subscribing to a Topic
  //----------------------
listener_status_alarms = new ROSLIB.Topic({
    ros : ros,
    name : 'robot_control/state',
    messageType : 'robot_control/Control_state'
  });

listener_status_alarms.subscribe(function(message) {

$('#console_html').html('[driver_R='+ message.alarms[0]+',driver_L='+
message.alarms[1]+',disabled_emg_R='+ message.alarms[2]+',disabled_emg_L='+
message.alarms[3]+',emergency_R='+ message.alarms[4]+',emergency_L='+
message.alarms[5]+']' );
//$('#alarm_status').html('[driver_R='+message.alarms[0]+']);
//console.log('alarms=[driver_R='+message.alarms[0]+',driver_L='+
message.alarms[1]+',disabled_emg_R='+ message.alarms[2]+',disabled_emg_L='+
message.alarms[3]+',emergency_R='+ message.alarms[4]+',emergency_L='+
message.alarms[5]+']');
});
```

```
function Joy_command(id,ref_vel,ref_ang){

$('#log_joy_nav').html('('+ ref_vel+','+ref_ang+')' );

var vec_lin = Object.seal({
    x: ref_vel,
    y: 0.0,
    z: 0.0
});

var vec_ang = Object.seal({
    x: 0.0,
    y: 0.0,
    z: ref_ang
});

 joy = new ROSLIB.Message({
   linear: vec_lin,
   angular: vec_ang
  });
 // Then we create the payload to be published. The object we pass in to
ros.Message matches the

  // And finally, publish.
  pub_joy.publish(joy);
}

function Command(txt){
//console.log('Command::txt'+txt);
var str_tmp=txt;
 goal = new ROSLIB.Message({
   data: str_tmp

  });
 // Then we create the payload to be published. The object we pass in to
ros.Message matches the

  // And finally, publish.
  pub_command.publish(goal);

}

$(document).on("pageshow","#page_status",function(){

vec = Object.seal({
    x: 0,
    theta: 0
});

JoyStick('#joystick_nav', 100, function(magnitude, theta, ximpulse, yimpulse) {
    //console.log(magnitude, theta, ximpulse, yimpulse);

    vec.x = (ximpulse/100 );
   if (vec.x<0.0 && vec.x>-0.1)vec.x=0.0;
    vec.theta = theta;
});

});

</script>
</body>
</html>
```