Trabajo de Fin de Grado
Grado en Ingienería Electrónica, Robótica y Mecatrónica

Comunicación de ROS con Arduino

Autor: Álvaro Pacheco Martinez
Tutor: Antonio Javier Gallego Len
Tutor externo: Ramón Andrés García Rodríguez

Dpto. Ingienería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2020

Trabajo de Fin de Grado
Grado en Ingienería Electrónica, Robótica y Mecatrónica

Comunicación de ROS con Arduino

Autor:
Álvaro Pacheco Martínez

Tutor:
Antonio Javier Gallego Len
Tutor externo:
Ramón Andrés García Rodríguez

Dpto. de Ingienería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2020

Autor:

Tutor:

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2020

El Secretario del Tribunal

A mi familia

A mis maestros

Agradecimientos

Quería mostrar mi agradecimiento a todas aquellas personas que, de una manera o de otra, me han apoyado y han contribuido en la realización de este trabajo.

A mis amigos, por todos esos buenos momentos que tan necesarios han sido durante todos estos años.

A mis padres, por darme la libertad para equivocarme.

A mi hermana, por dejarme simpre claro cuando no llevo la razón.

Resumen

La robótica ha experimentado un amplio avance en los últimos años convirtiéndose en un pilar fundamental para industrias como el automóvil o la aeronáutica entre otras muchas. Asi mismo la robótica es cada vez más utilizada en el sector asistencial y de servicio, formando así parte de la vida de las personas.

ROS, siglas de Robotic Operation System, se ha convertido en una herramienta fundamental en el desarrollo de la robótica desde su aparición en 2007. El ofrecer una gran cantidad de funcionalidades que facilitan el desarrollo de software y ser de código abierto ha disparado su popularidad.

El trabajo fin de grado está dirigido a explorar algunas de estas funcionalidades como la posibilidad de testeo de plataformas mediante simulaciones 3D o la comunicación con microcontroladores a través del puerto serie.

Abstract

Robotics has undergone a broad avance in recent years, becoming a fundamental pillar for industries such as the automobile or aeronautics, among many others. Likewise, robotics is increseangly used in care and service sector, thus becoming part of people's lives.

ROS, which stands for Robotic Operation System, has become a fundamental tool in the development of robotics since its appearance in 2007. Offering a large number of functionalities that facilitate software development and being open source has increased its popularity.

This final degree project is aimed at exploring some of these functionalities, such as the possibility of testing platforms through 3D simulations or communication with microcontrollers through the serial port.

Índice

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

Notación

sen	Función seno
tg	Función tangente
arctg	Función arco tangente
cos	Función coseno
lm	Lúmenes
R	Resistencia eléctrica
δ	Ángulo de declinación
λ	Latitud
ω	Ángulo horario
αs	Altitud
γs	Acimut

1 INTRODUCCIÓN

La filosofía DIY, Do It Yourself, se ha expandido rápidamente en la última década gracias al abaratamiento de los componentes electrónicos y a la aparición de plataformas de desarrollo potentes pero enfocadas al aprendendizaje como Arduino o Raspberry Pi.

1.1 Objetivos

El Objetivo de este trabajo es desarrollar un entorno de comunicación entre ROS y Arduino para medir la DNI, Direct Normal Irradiance, del Sol a través de cuatro sensores LDR. Se hará uso de una brújula para orientar los mencionados sensores. Además se simulará usando Gazebo el modelo del robot ROSBOT que se desplazará de forma autónoma hacia la ubicación seleccionada y una vez alcanzada empezará la comunicación con arduino.

1.2 Estructura del proyecto

El primer paso fue la simulación en Gazebo del modelo del robot ROSBOT junto con el mundo Willow Garage y conseguir que el robot se desplace por el mapa mediante control teledirigido.

Una vez realizado lo anterior se usó el paquete de ROS Navigation Stack para utilizar SLAM que nos permite tanto generar un mapa como desplazarnos de forma autónoma por la simulación. También se ha utilizado Rviz para poder visualizar tanto el mapa como la posición actual del robot en él y otros datos que puedan ser de interés.

Se ha utilizado el paquete Move_Base para la planificación de trayectoria y el control autónomo. Podremos seleccionar localizaciones de destino utilizando las herramientas disponibles en Rviz, pero también se ha implementado la forma de hacerlo mediante código.

Una vez se ha detectado que se ha alcanzado el objetivo se calcularán en ROS la altitud y acimut del Sol para ese dia y hora concretos usando las ecuaciones solares. La altitud y acimut serán enviados a Arduino que los ultilizará, junto a los datos de la brújula, para mover el pan and tilt donde estarán colocados los sensores LDR y hacer que aputen hacia el sol. Una vez capturados los datos serán devueltos a ROS.

2 SOFTWARE

En este capítulo se mostraran los diferentes programas y herramientas de software utilizados para la realización del trabajo.

2.1 ROS

ROS (Robotic Operating System) es un framework para el desarrollo de software para robots que provee la funcionalidad de un sistema operativo. ROS fue desarrollado originalmente en 2007 por el Laboratorio de Inteligencia Artificial de Stanford para dar soporte al proyecto del Robot con Inteligencia Artificial de Stanford. A partir de 2008 Willow Garage, un instituto de investigación robótica con más de veinte instituciones colaborando en un modelo de desarrollo federado, continúa con el desarrollo.

ROS provee diferentes servicion fundamentales para el desarrollo de software en robótica y que son los servicios estándar de un sistema operativo, abstracción del hardware, control de dispositivos de bajo nivel, paso de mensajes entre procesos y mantenimiento de paquetes. Utiliza una arquitectura de grafos donde el procesamiento se realiza en los nodos, que pueden recibir, mandar y multiplexar mensajes de sensores, actuadores y otros nodos. ROS está orientado para un sistema UNIX (Ubuntu) aunque también se está adaptando a otros sistemas operativos como Fedora, Mac OS X, Arch, Gentoo, OpenSUSE, Slackware, Debian o Microsoft Windows, considerados como 'experimentales'.

ROS tiene dos partes básicas: la parte del sistema operativo, que se ha descrito anteriormente y ros-pkg, una colección de paquetes aportados por la comunidad de usuarios, organizados en conjuntos llamados stacks que implementan la funcionalidades tales como localización y mapeo simultáneo, planificación, simulación, utilizadas en la realización de este trabajo.

En ROS hay implementados varios estilos de comunicación, como comunicación síncrona de tipo RPC (Remote Procedure Call) a través de servicios, comunicación asíncrona por medio de los tópicos y almacenamiento de datos en un servidor de parámetros que se pueden consultar desde cualquier nodo. Para mayor comprensión se describen a continuación los sistemas de ficheros de ROS.

En el sistema de ficheros de ROS, se pueden distinguir:

- Paquetes (packages): Los paquetes son la principal unidad para organizar el software en ROS. Un paquete puede contener nodos (la unidad de ejecución en ROS), librerías, data sets o ficheros de configuración, entre otros.
- Ficheros de manifiesto (manifest.xml): Los manifiestos proporcionan metadatos de un paquete, incluyendo información de licencia, dependencias con otros paquetes y opciones de compilación.

- Stacks: Un stack es un conjunto de paquetes que proporcionan una funcionalidad. Un ejemplo de stack es la ”navigation_stack” disponible en los repositorios de ROS.
- Ficheros de manifiesto de un stack (stack.xml): De forma similar al manifiesto de un paquete, este tipo de ficheros proporcionan información del stack, como la versión, la licencia y las dependencias.
- Descripción de mensajes: Estos ficheros describen los mensajes a utilizar por los nodos, definiendo la estructura de datos del mismo. Se suelen ubicar dentro de la carpeta msg de un paquete y tienen la extensión “.msg”.
- Descripción de servicio: Se utilizan para describir los servicios, definiendo la estructura de datos para la petición y la respuesta del servicio. Se suelen ubicar dentro de la carpeta srv de un paquete y tienen la extensión “.srv”.

A continuación se pasa a describir los distintos elementos que forman parte del proceso de comunicación entre nodos:

- Nodo: Los nodos son procesos que realizan la computación. Un nodo es un archivo ejecutable dentro de un paquete de ROS. Mediante los nodos se puede realizar un complejo modular en el que los nodos se comunican entre sí. Un sistema de control del robot comprende por lo general muchos nodos.
- Maestro: El ROS Master proporciona el registro de nombre y consulta el resto del grafo de computación. Almacena información de registro de topics y servicios de nodos ROS. El Master coordina los nodos realizando una función similar a la de un servidor DNS en internet.
- Servidor de parámetros: Este permite almacenar datos, de forma que se pueden actualizar y consultar por cualquier nodo. Actualmente es parte del Master.
- Mensajes: Es la forma de comunicación entre los nodos. Un mensaje es una estructura simple con campos tipados. Los tipos que se permiten son desde tipos básicos como enteros o reales a estructuras C más complejas y vectores.
- Tópico o topic: El topic es el canal de comunicación entre nodos. El intercambio de mensajes entre nodos lo hacen a través de los topics que son buses de datos. El topic es un nombre que se usa para identificar el contenido del mensaje. Un nodo publica un mensaje en un topic al que se ha suscrito un segundo nodo para leer los mensajes allí depositados. Pueden existir muchos publicadores concurrentemente para un solo topic y un nodo puede suscribirse y/o publicar en varios topics. El nodo suscriptor o publicador no tiene porque conocer si existen otros nodos utilizando el mismo topic.

- Servicios: Cuando no es posible la comunicación entre nodos mediante mensajes, ROS permite la comunicación a través del servicio. Este está definido por dos mensajes, uno para la petición y otro para la respuesta. De esta manera el nodo ofrece un servicio con un nombre específico y otro nodo pude solicitar dicho servicio mediante un mensaje de petición.
- Acciones: Se diferencian de los servicios, en que se envía una petición y se recibe una respuesta, pero en esta existe la posibilidad de cancelar el servicio, y por tanto no es necesario esperar hasta obtener la respuesta.
- Rosbags: Son archivos donde se guardan los datos de los mensajes publicados por los nodos publicadores que interesan para una posterior utilización. Los rosbags son muy útiles para almacenar datos de sensores para después probar los algoritmos de forma offline en nuestro ordenador.
- Rviz: Es un entorno gráfico de visualización en 3D para ROS.

2.2 Gazebo

Gazebo te permite crear un escenario 3D en tu ordenador con robots, obstáculos y otros tipos de objetos. Gazebo también contiene un motor de físicas para simular gravedad, inercias, iluminación, etc. Esto te permite evaluar tu robot sin necesidad de tener que construir un escenario de pruebas físico.

Algunos usos comunes de Gazebo son:

- Testeo de algoritmos
- Diseño de robots
- Realización de pruebas en escenarios realistas

Gazebo incluye las siguientes características:

- Múltiples motores de físicas
- Una amplia librería de modelos de robots y de entornos
- Una gran cantidad de sensores

2.3 ROSbot

ROSbot es una plataforma para el desarrollo de robots autónomos desarrollado por la empresa Husarion y basada en el controlador Core2 de la misma compañía. ROSbot puede ser útil para el desarrollo en diferentes campos como pueden ser la robótica de servicio, inspección o incluso robótica de enjambre.

Figura 2.1: ROSbot 2.0

ROSbot cuenta con las siguientes características:

- 4 ruedas móviles accionadas por motores DC con encoders.
- Cámara Orbbec Astra RGBD.
- Sensor inercial MPU 9250 (acelerómetro + giroscopio).
- Panel trasero que proporciona interfaz para distintos módulos.
- Procesador CORE2-ROS con Intel® ATOM™ x5-Z8350 procesador de 64 bits hasta 1.92GHz, 4GB DDR3L RAM y 32GB eMMC.
- Escaner RPLIDAR A3.

Husarion provee de forma gratuita un modelo de ROSbot para simular con Gazebo y así poder testear los algoritmos desarrollados.

2.4 Rosserial

Rosserial es un protocolo para enviar datos a través de puertos series. En una implementación cliente-servidor, el servidor es un ordenador en el que se está ejecutando ROS y el cliente es un microcontrolador que recibe o adquiere los datos y los envía al servidor a través de mensajes de ROS. En el caso concreto de este trabajo tanto el servidor como el cliente son suscriptores y publicadores.

El paquete Rosserial-client esta disponible para varios microcontroladores como Arduino, STM32, embeddedlinux entre otros. Por la otra parte el paquete Rosserial-

server esta disponible para Python y C++.

Arduino y el Arduino IDE son grandes herramientas para programar hardware de forma rápida y fácil. Usando el paquete Rosserial_arduino se puede usar ROS directamente en el Arduino IDE. Rosserial provee a ROS un protocolo de comunicación que funciona con la UART de Arduino. Esto permite a Arduino funcionar como un nodo más de ROS, lo que le permite subscribirse y publicar en los tópicos, publicar transformadas y acceder al tiempo del sistema de ROS.

2.5 Navigation Stack

En esta sección se mostrarán uno de los dos elementos principales utilizados en la navegación, el Navigation Stack [1][2]. En pocas palabras, la función del Navigatión Stack consiste en mover un robot desde una posición inicial hasta otra final sin que colisione con los elementos del entorno. Dentro vienen implementados varios algoritmos de navegación que pueden ser utilizados para implementar navegación autónoma en el robot.

Navigation Stack está diseñado para ser lo más genérico posible, sin embargo algunos requerimientos de software deben ser satisfechos por el robot.

- El robot debe llevar montado un sensor láser en algún lugar. Este sensor se usa para contruir el mapa del entorno.
- El paquete Navigation Stack tendrá mejor rendimientos en robots con forma de bases cuadradas y circulares. El paquete funcionará con formas arbitrarias pero el rendimiento no está asegurado.
- El paquete funcionará mejor en robots diferenciales y holónomos. El robot movil debe ser controlado enviándole comandos de velocidades v_x, v_y (velocidad lineal), Θ (velocidad angular).

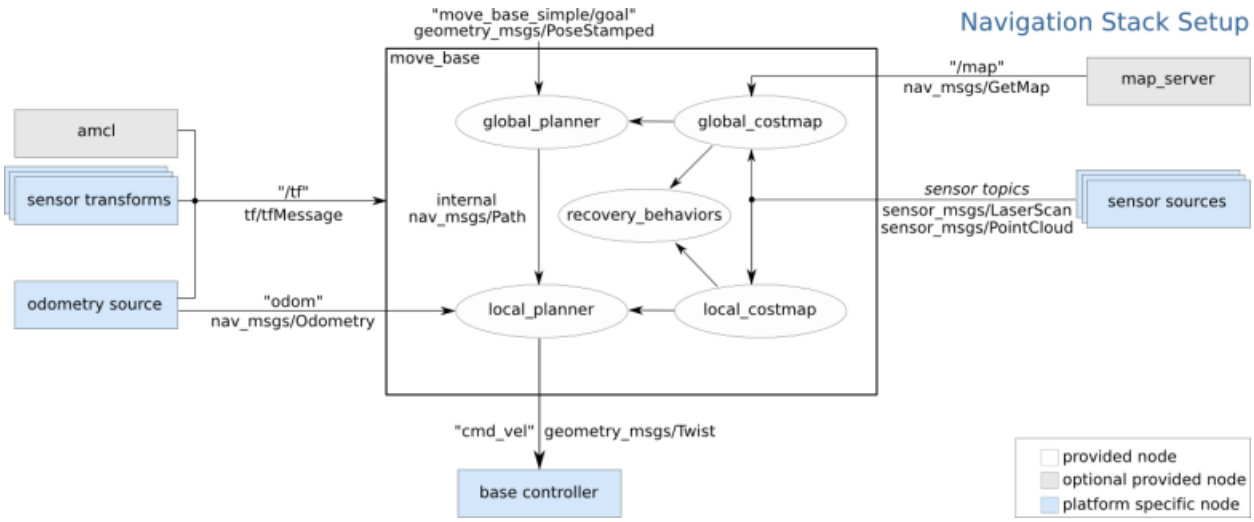

Figura 2.2: Esquema Navigation Stack

De acuerdo con el diagrama anterior, para configurar el paquete para un robot debemos proporcionar bloques funcionales que son interfaces para el paquete Navigation Stack. A continuación se detallan todos estos bloques que son las entradas para este paquete:

- **Odometría**: La odometría de un robot proporciona la posición actual de un robot con respecto a su posición inicial. La principal fuente de odometria son los encoders, IMU y cámaras 2D/3D (odometria visual). El valor de la odometría debe ser publicada con el formato propio del paquete nav_msgs/Odometry. El mensaje de la odometría puede contener la posición y la velocidad del robot. La odometría es fundamenteal para el Navigation Stack.
- **Sensores**: Como se ha comentado anteriormente, el paquete Navigation Stack necesita los datos de un sensor lidar o una nube de puntos para mapear el entorno del robot. Estos datos, junto con la odometría, se combinan para construir el global y local costmap del robot. Los datos del láser van dentro de un mensaje tipo sensor_msgs/LaserScan y la nube de puntos dentro de tipo sensor_msgs/PointCloud.
- **sensor transforms/tf**: el robot debe publicar la relación entre los diferentes ejes usando ROS.
- **base_controller**: La principal función del base_controller es convertir la salida del Navigation Stack, que es del tipo geometry_msgs/Twist, en las correspondientes velocidades del robot.

2.6 Move Base

El nodo Move base es el segundo de los elementos fundamentales utilizados en la navegación. El principal objetivo de este nodo es mover el robot desde su posición actual hacia la posición objetivo con la ayuda de los nodos de navegación. El nodo move_base une el global-planner y el local-planner, conectando el paquete rotate-recovery si el robot se queda estancado en algún obstáculo y conectando el global costmap y local costmap para obtener el mapa [3].

El nodo move_base es una implementación de SimpleActionServer, el cual toma la posición objetivo con tipo geometry_msgs/PoseStamped. Podemos enviar una posición objetivo usando un nodo SimpleActionClient.

El nodo move_base se subscribe al tópico move_base_simple/goal, el cual es la entrada del Navigation Stack, como se muestra en la Figura 2.2. Cuando este nodo recibe la posición objetivo une los diferentes componentes global_planner, local_planner, recovery_behavior, global_costmap, y local_costmap para generar la salida que es el comando de velocidad de tipo geometry_msgs/Twist y lo envía al base controller para mover el robot y alcazar el objetivo.

A continuación se explican una lista de los paquetes unidos al nodo move_base:

- **global-planner**: Este paquete provee librerías y nodos para planificar el camino óptimo desde la posición actual hasta la posición objetivo. Este paquete contiene la implementación de algoritmos de planificación como A*, Dijkstra y varios más para calcular el camino más corto hacia el objetivo.
- **local-planner**: La principal función de este paquete es dirigir al robot dentro de una sección del camino global planificado. El local planner tomará la odometría del robot y la lectura de los sensores y enviará un comando de velocidad apropiado al robot para completar una sección del camino.
- **rotate-recovery**: Este paquete ayuda al robot a recuperarse de un obstáculo local.
- **costmap-2D**: El principal objetivo de este paquete es mapear el entorno del robot. El robot solo puede planificar una ruta con respecto a un mapa. En ROS se generan mapas de cuadrículas de ocupación en 2D y 3D, los cuales representan el entorno. Cada casilla contiene una valor de probabilidad que indica si la casilla esta ocupada o no. El paquete costmap-2D puede construir la capa de cuadrículas subscribiendose a los tópicos donde se publican los datos del láser o de la nube de puntos y la odometría. Existen global costmap para navegaciones globales y local costmap para navegaciones locales.

3 HARDWARE

En esta sección se verán los diferentes elementos Hardware utilizados así como sus características principales.

3.1 Brújula digital HMC5883L

Esta brújula implementa en un solo chip tres sensores de campo magnéticos dipuestos perpendicularmente en cada uno de los ejes espaciales (X, Y, Z). Este módulo cuenta con la circuitería de control y comunición necesarias para obtener los datos a través del protocolo I2C de dos hilos. Todo esto nos da la posibilidad de implementar sensores de campo magnético, sensores de posicionamiento y brújulas tridimensionales.

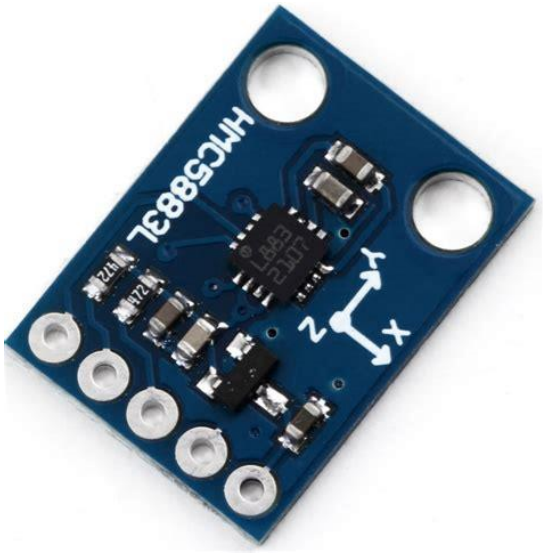

Figura 3.1: Brújula digital HMC5883L

Voltaje de alimentación	3V a 5V
Modelo interface I2C	GY-271
Chip	HMC58883L
Precisión de medida	±2º
Rango de medición	±1,3 – 8 Gauss
Resolución	5 mili Gauss
Medidas	14mm x 15mm

Tabla 3.1: características Brújula digital HMC5883L

3.2 LDR 7516

LDR corresponden a las siglas de Light Dependent Resistor, en español conocidas como fotoresistencias. Una fotoresistencia es un componente electrónico cuya resistencia disminuye con el aumento de la intensidad de la luz que incide en ella. Con ellas conseguimos capturar los datos de la cantidad de luz que nos llega del Sol.

Figura 3.2: LDR 7516

Resistencia a 10 Lux	10K a 20K Ohm
Tolerancia a 25ºC	500mW
Aislamiento	1000VDC
Dimensiones sensor	2 x 4 x 5mm
Largo de patillas	31mm

Tabla 3.2: Características de LDR 7516

Para calibrar los LDR se ha utilizado un método similar a [4]. Se han hecho incidir sobre ellos distintas intensidades de luz y se ha obtenido la resistencia de los LDR a través de las lecturas del Arduino. La intensidad de la luz se ha medido a través de una aplicación de móvil. La relación Resistencia-Lumen toma forma lineal en escala logáritmica. Una vez obtenido los puntos se ha realizado un ajuste de mínimos cuadrados.

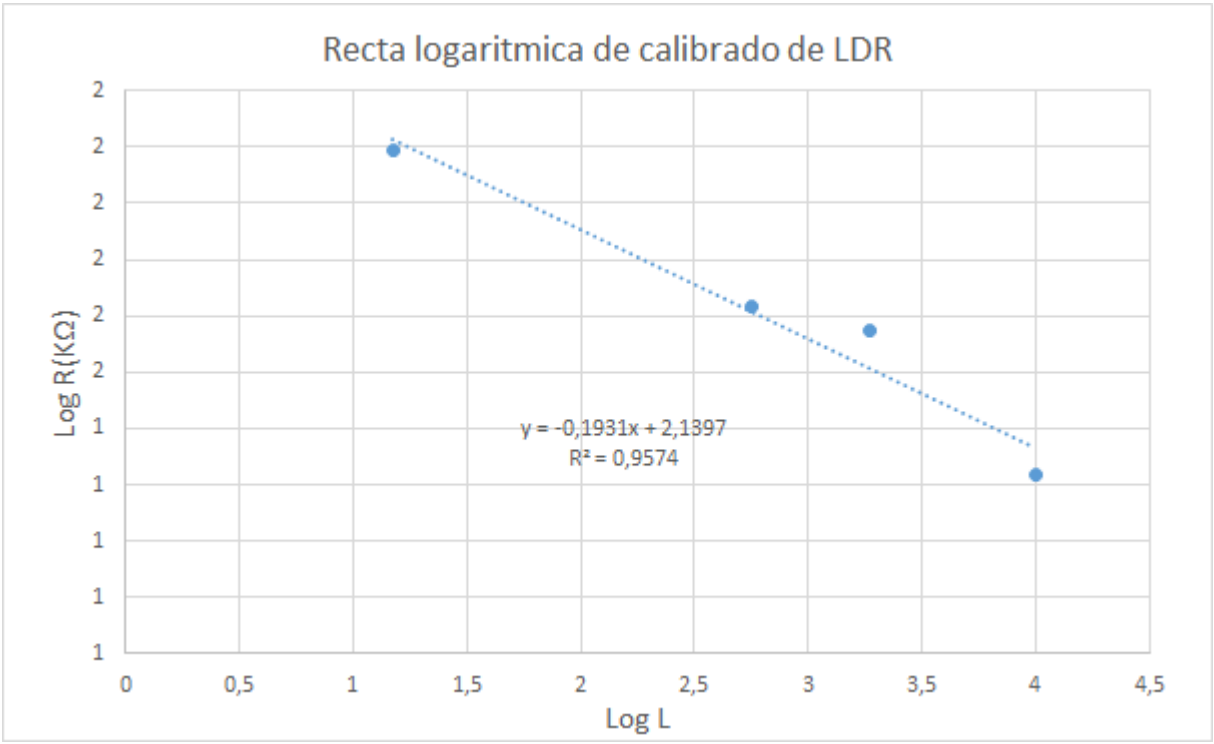

Figura 3.3: Recta logarítmica de calibrado de LDR

De la ecuación de la recta obtenida obtenemos que la relación de los lúmenes (lm) en función de la resistencia (R) es:

$$lm = 10^{(\frac{2.1397 - logR}{0.1931})}$$

3.3 Arduino Micro

Arduino es una placa electrónica de hardaware y software libre. Esto ha permitido que se hayan desarrollado una gran cantidad de componentes electrónicos y shields compatibles, lo que a su vez ha fomentado la creación de una gran comunidad entorno a las placas Arduino.

La mayoría de las placas Arduino pueden ser programadas a través del puerto serie que incorporan haciendo uso del Bootloader que traen programado por defecto. El software de Arduino consiste de dos elementos: un entorno de desarrollo (IDE) y en el cargador de arranque (*bootloader,* por su traducción al inglés) que es ejecutado de forma automática dentro del microcontrolador en cuanto este se enciende.

En este proyecto se ha usado la placa Arduino Micro, una de las placas más pequeñas de la familia.

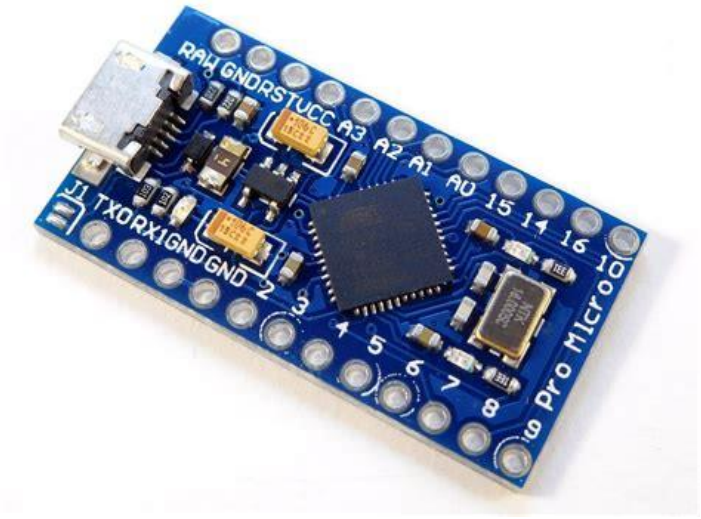

Figura 3.4: Arduino Micro

Entrada/Salidas digitales	12, de ellas 5 PWM
Entradas analógicas	4 (10 bit c/u)
Voltaje de entrada	5-12V (Posee regulador interno)
Voltaje	5v
Corriente máxima de salida	150mA
Microcontrolador	ATMega32U4
Conector	micro-USB
Dimensiones	3.31 x 1.78 cm
Velocidad de reloj	16Mhz

Tabla 3.3: Características Arduino Micro

3.4 Micro Servos

Un micro servo es un actuador rotatorio que nos permite controlar con precisión su posición, generalmente entre 0º y 180º. Su bajo precio y si capacidad para mover pequeñas cargas los han hecho muy populares en proyectos DIY.

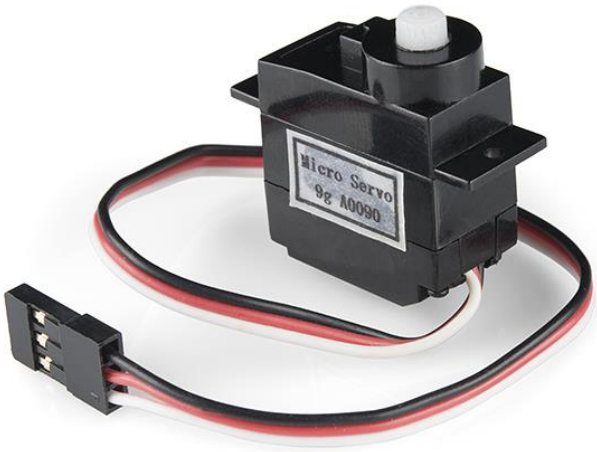

Figura 3.5: Micro servo

Voltaje	4.8-6.0 Voltios
Par	2.5 Kg-cm
Rotación	180º
Dimensiones	31.8 x 11.7 x 29mm

Tabla 3.4: Características Micro servo

3.5 Pan and Tilt

El Pan and Tilt utilizado en este trabajo es una estrucutura simple formada por dos piezas similares con forma de u. Junto a los micro servos es capaz de girar en el eje X y en el eje Z, lo que nos permite orientarlo hacia el Sol.

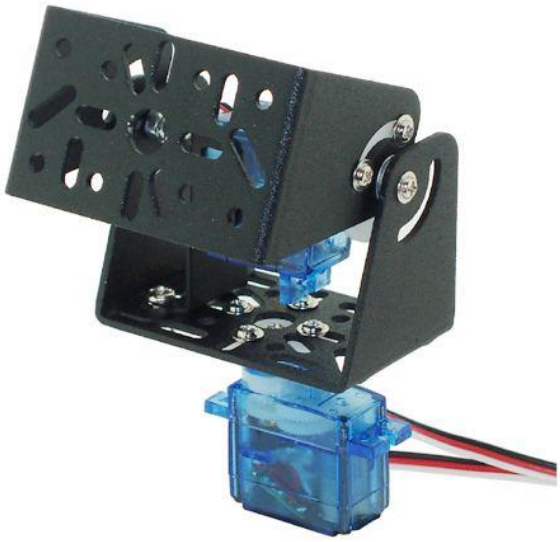

Figura 3.6: Pan and Tilt

3.6 Conexiones

En este apartado se detallan las conexiones entre el arduino y los diferentes componenetes electrónicos.

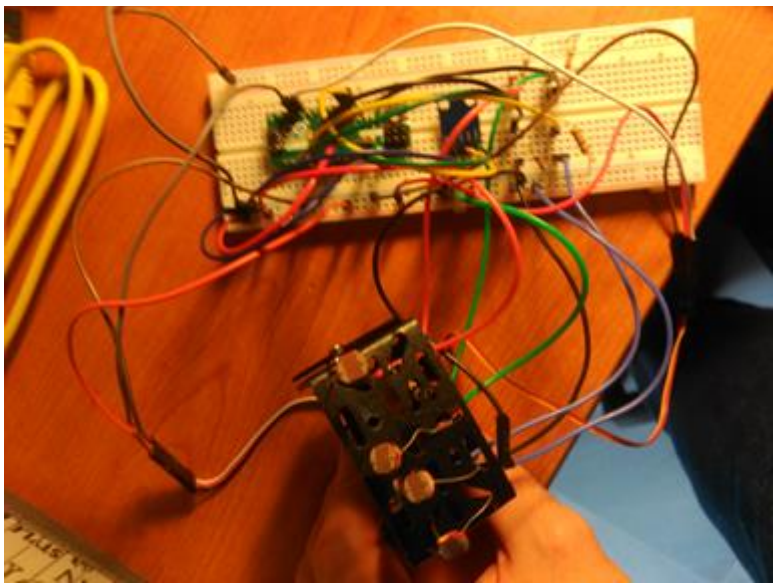

Figura 3.7: Conexiones

Brujula Digital	Arduino Micro
Vcc	5V
Gnd	Gnd
SDA	2
SCL	3

Servo Acimut	Arduino Micro
Vcc	5V
Gnd	Gnd
Control	9

Servo Altitud	Arduino Micro
Vcc	5V
Gnd	Gnd
Control	10

LDR	Arduino Micro
Lectura LDR 1	A0
Lectura LDR 2	A1
Lectura LDR 3	A2
Lectura LDR 4	A3

4 ECUACIONES SOLARES

4.1 Introducción

El objetivo de esta sección es explicar como se llega a las ecuaciones que calculan la altitud y acimut del Sol [5]. Para ello primero calcularemos las coordenadas del vector S, vector que apunta hacia el Sol desde el centro de la Tierra, en el sistema de referencia N, E, Z, situado en la superficie de la Tierra. El valor de estas coordenadas estará en función de los ángulos δ (declinación), λ (latitud) y ω (ángulo horario). El siguiente paso será hacer que estas coordenadas estén en función de αs (altitud) y γs (acimut).

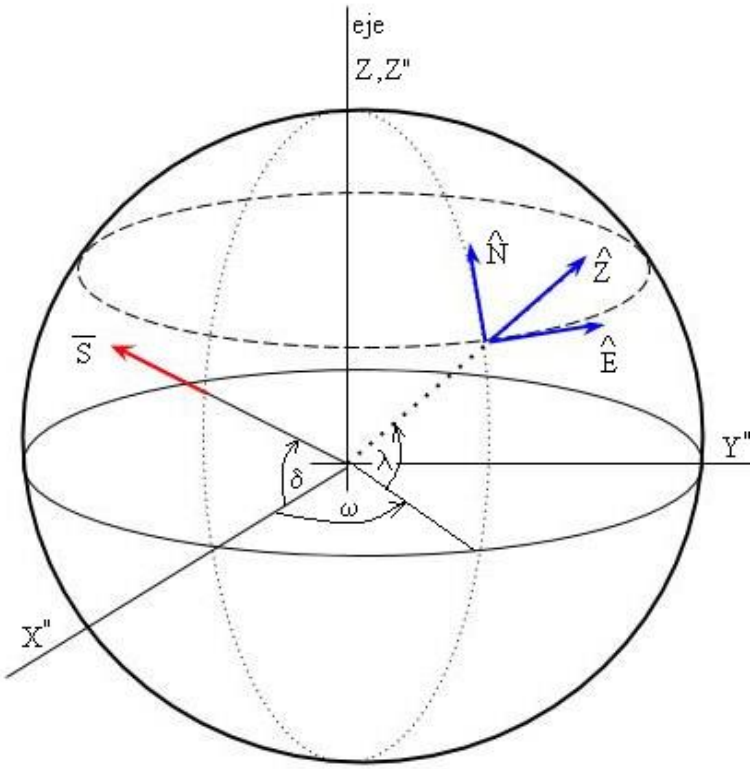

Figura 4.1: Vector S en el sistema de referencia centro de la Tierra

En el sistema de referencia de los ejes X", Y", Z" (vectores unitarios i" ,j", k" respectivamente) el plano formado por X" e Y" coinciden con el plano ecuatorial de la Tierra y el eje Z" con el eje de la Tierra. En este sistema de referencia el vector S, que va en la dirección del Sol, se encuentra siempre en el plano X", Z".

$$S = \cos(\delta) * i" + \sin(\delta) * k$$

El ángulo de declinación δ es aquel que forma el vector S con el eje X y se calcula con la siguiente fórmula:

$$\delta = (2 * pi/365) * (diajuliano - 1) \qquad (4.1)$$

Día Juliano es el número total de días transcurridos desde el inicio del año.

El sistema de referencia local formado por los vectores E, N, Z se situa en algún punto de la superficie terrestre. Las direcciones de dichos vectores son, la dirección Oeste-Este (tangente al paralelo en el punto), la dirección Sur-Norte (tangente al meridiano en el punto), y la dirección radial. El punto sobre la superficie terrestre esta definido por los los ángulos λ (latitud) y ω (ángulo horario). El ángulo ω se calcula mediante la siguiente fórmula:

$$\omega = ((\mathrm{Ts} - 12) * 15) * (\pi/180) \tag{4.2}$$

Donde Ts es la hora solar.

Como se dijo en la introducción, el objetivo es espresar S en el sistema de referencia N, E, Z. Para ello espresaremos los vectores unitarios E, N, Z en términos de los vectores unitarios i", j", k".

.

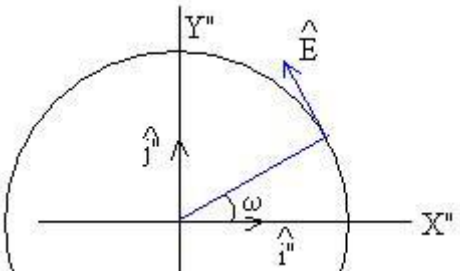

Figura 4.2: Vector E en el sistema de referencia ij

Como vemos en la figura, $E = -sin\omega \cdot i" + cos\omega \cdot j"$ (4.3)

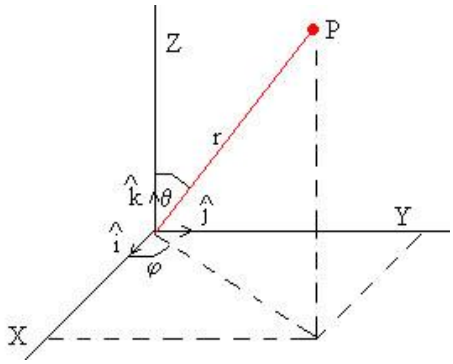

Figura 4.3: Representación de las coordenadas de un punto en el sistema XYZ

Sabiendo que Z se encuentra en la dirección radial y basándonos en la figura anterior podemos espresarlo de la siguiente forma.

$$Z = cos\lambda \cdot cos\omega \cdot i" + cos\lambda \cdot sin\omega \cdot j" + sin\lambda \cdot k" \tag{4.4}$$

Para sacar N calculamos el producto vectorial de $E\ x\ Z$.

$$N = \begin{vmatrix} i" & j" & k" \\ cos\lambda cos\omega & cos\lambda sin\omega & sin\lambda \\ -sin\omega & cos\omega & 0 \end{vmatrix} \tag{4.5}$$

$$N = -\cos\omega \cdot \sin\lambda \cdot \mathrm{i}" - \sin\omega \cdot \cos\lambda \cdot \mathrm{j}" + \cos\lambda \cdot \mathrm{k}" \tag{4.6}$$

Calculamos las componentes de S en los ejes N, E, Z mediante el producto escalar.

$$S_N = S \cdot N = -cos\omega \cdot sin\lambda \cdot cos\delta + cos\lambda \cdot sin\delta \quad (4.7)$$

$$S_E = S \cdot E = -sin\omega \cdot cos\delta \quad (4.8)$$

$$S_Z = S \cdot Z = cos\omega \cdot cos\lambda \cdot cos\delta + sin\lambda \cdot sin\delta \quad (4.9)$$

$$S = -sin\omega \cdot cos\delta \cdot E + (cos\lambda \cdot sin\delta - cos\omega \cdot sin\lambda \cdot cos\delta) \cdot N + (cos\omega \cdot cos\lambda \cdot cos\delta + sin\lambda \cdot sin\delta) \cdot Z \quad (4.10)$$

4.2 Altitud y Acimut

Como se puede ver en la siguiente figura, el vector S se puede espresar en función de los ángulos αs y γs que se corresponden con la altitud y el acimut del Sol respectivamente.

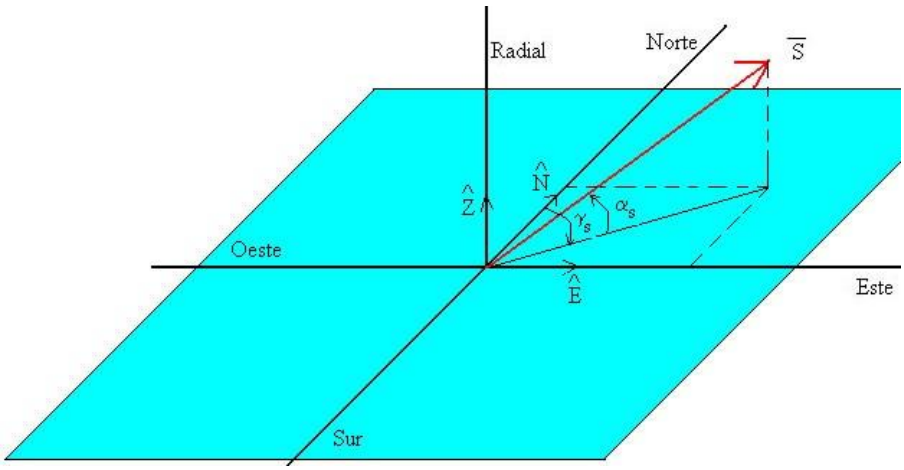

Figura 4.4: Vector S en el sistema de referencia local

Por lo tanto el vector S se podría escribir también de la siguiente manera:

$$S = cos\ \alpha s \cdot sin\ \gamma s \cdot E + cos\alpha s \cdot cos\ \gamma s \cdot N + sin\ \alpha s \cdot Z \quad (4.11)$$

Igualando las componentes, obtenemos tres relaciones:

$$cos\ \alpha s \cdot sin\ \gamma s = -sin\omega \cdot cos\delta \quad (4.12)$$

$$cos\ \alpha s \cdot cos\ \gamma s = cos\lambda \cdot sin\delta - sin\lambda \cdot cos\omega \cdot cos\delta \quad (4.13)$$

$$sin\ \alpha s = cos\lambda \cdot cos\omega \cdot cos\delta + sin\lambda \cdot sin\delta \quad (4.14)$$

Mediante la relación 4.14 obtenemos αs.

$$sin\ \alpha s = cos\lambda \cdot cos\omega \cdot cos\delta + sin\lambda \cdot sin\delta \quad (4.15)$$

Despejamos γs de la ecuación 4.13.

$$\cos\gamma s = \frac{\cos\lambda \cdot \sin\delta - \cos\omega \cdot \sin\lambda \cdot \cos\delta}{\cos\alpha s} \quad (4.16)$$

Debido a que el arcocoseno solo devuelve valores comprendidos entre 0º y 180º y el acimut comprende valores entre 0º y 360º es nesario seguir la siguiente regla.

- Si el ángulo horario $\omega<0$ entonces γs no varía
- Si el ángulo horario $\omega>0$ entonces $\gamma s= 360-\gamma s$

Si γs es menor de 180º significa que el Sol se encuentra al Este del observador. Si γs es mayor de 180º significa que el Sol se encuentra al Oeste.

5 EXPERIMENTOS

En esta sección se detallaran los diferentes experimentos realizados y los resultados obtenidos.

5.1 Simulación en Gazebo

La primera parte del trabajo consiste la simulación del robot Rosbot usando Gazebo y conseguir que se desplace de manera autónoma por el entorno. En este caso se ha simulado el laboratorio de investigación robótica Willow Garage con entorno.

5.1.1. Posiciones objetivo desde Rviz

Con Rviz podemos enviar posiciones objetivos al nodo move_base a través de la herramienta nav_goal. El código LecturaStatus.cpp recibe información del tópico move_base/status para saber cuando el robot ha alcanzado el objetivo.

En la Figura 5.1 se muestra el robot simulado en Gazebo, a la izquierda, y la visualización del robot y del mapa generado en Rviz, a la derecha.

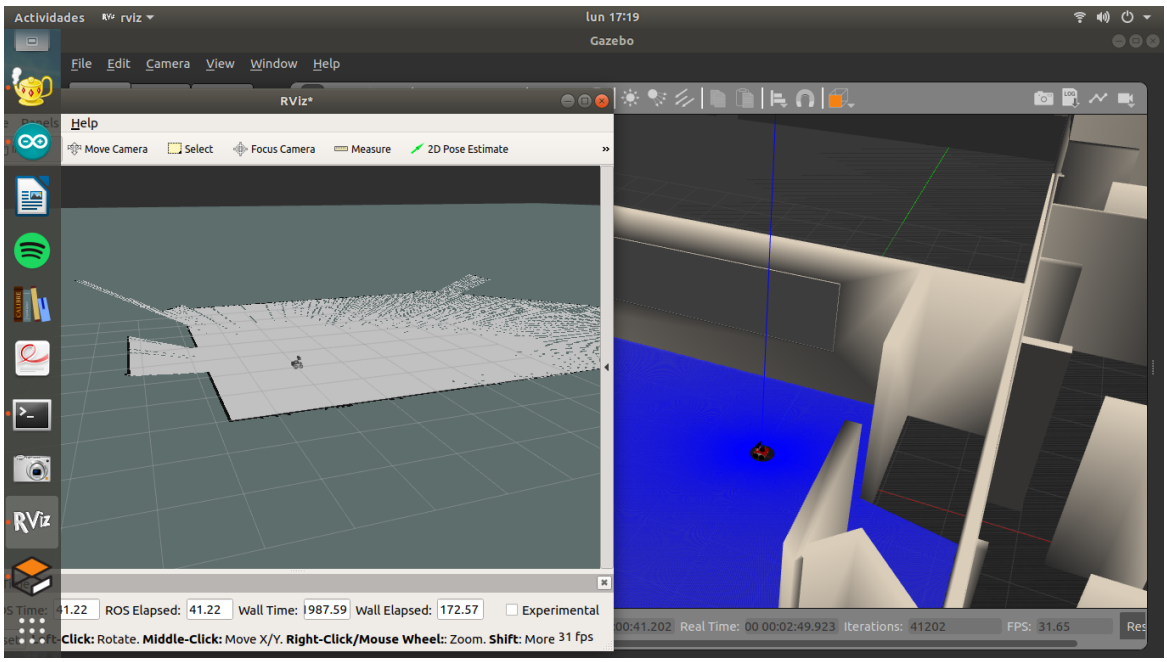

Figura 5.1: ROSbot simulado en Gazebo

En la Figura 5.2 se puede ver como se usan las herramientas de Rviz para enviar posiciones objetivo al sistema de navegación.

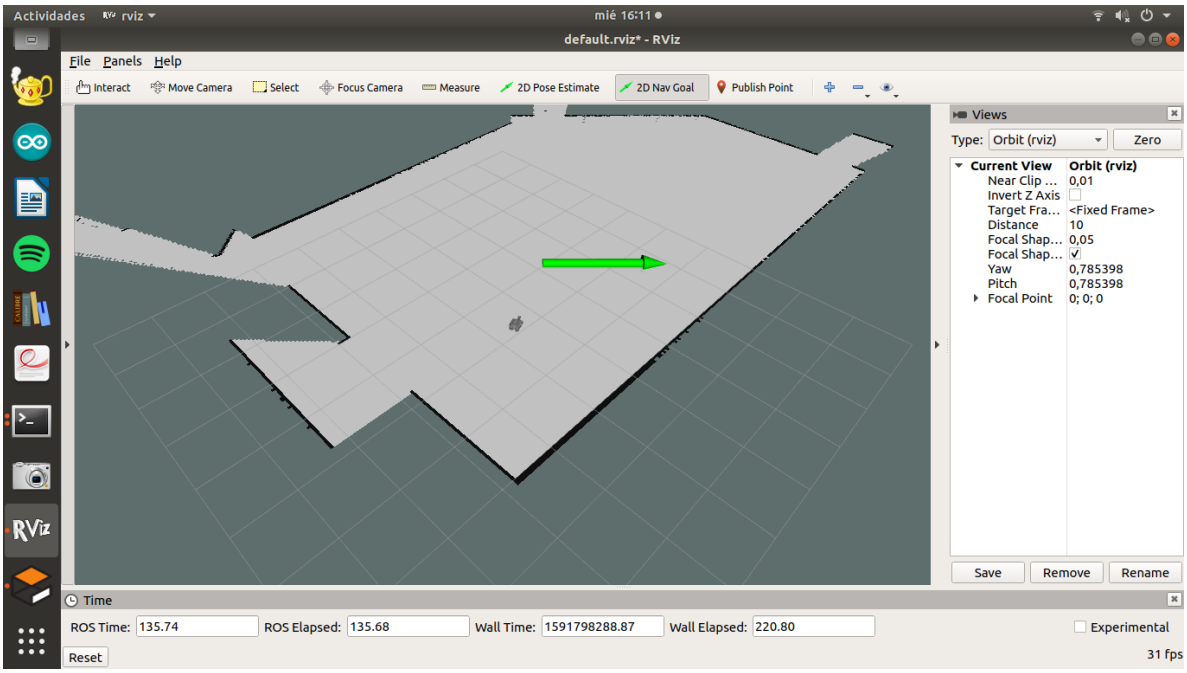

Figura 5.2: Envio posición objetivo desde Rviz

Finalmente en robot llega a la posición objetivo en la Figura 5.3.

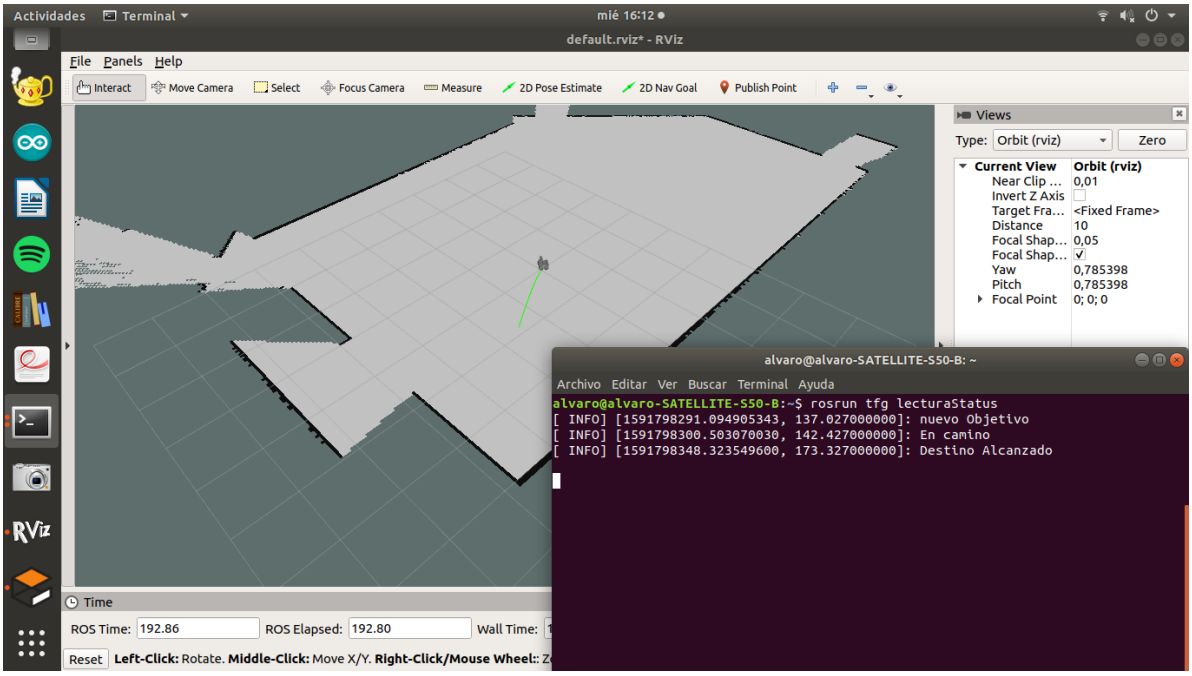

Figura 5.3: ROSbot llegando al destino

5.1.2. Posiciones objetivo desde código

Tambien es posible enviar la posición objetivo desde código utilizando el paquete actionlib. Esto esta programado en el código objetivo.cpp.

En la Figura 5.4 podemos ver como se ejecuta el código con nombre Objetivo y se muestra por pantalla que se ha enviado la posición de destino.

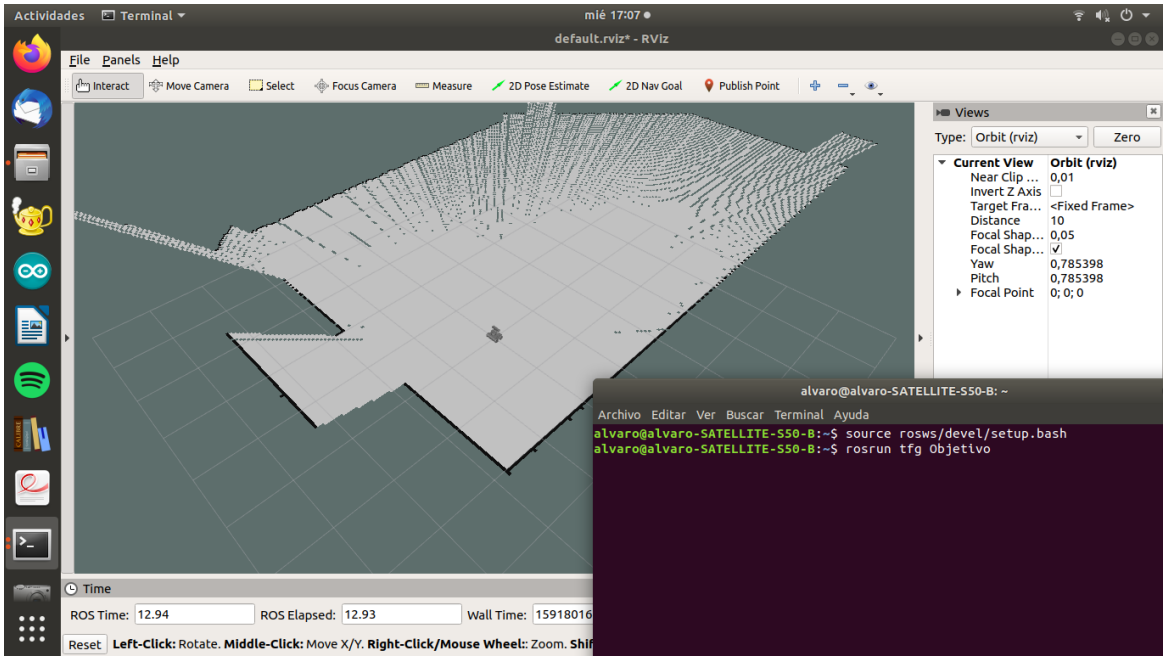

Figura 5.4: Envio de posición objetivo desde código

El robot llega al destino en la Figura 5.5.

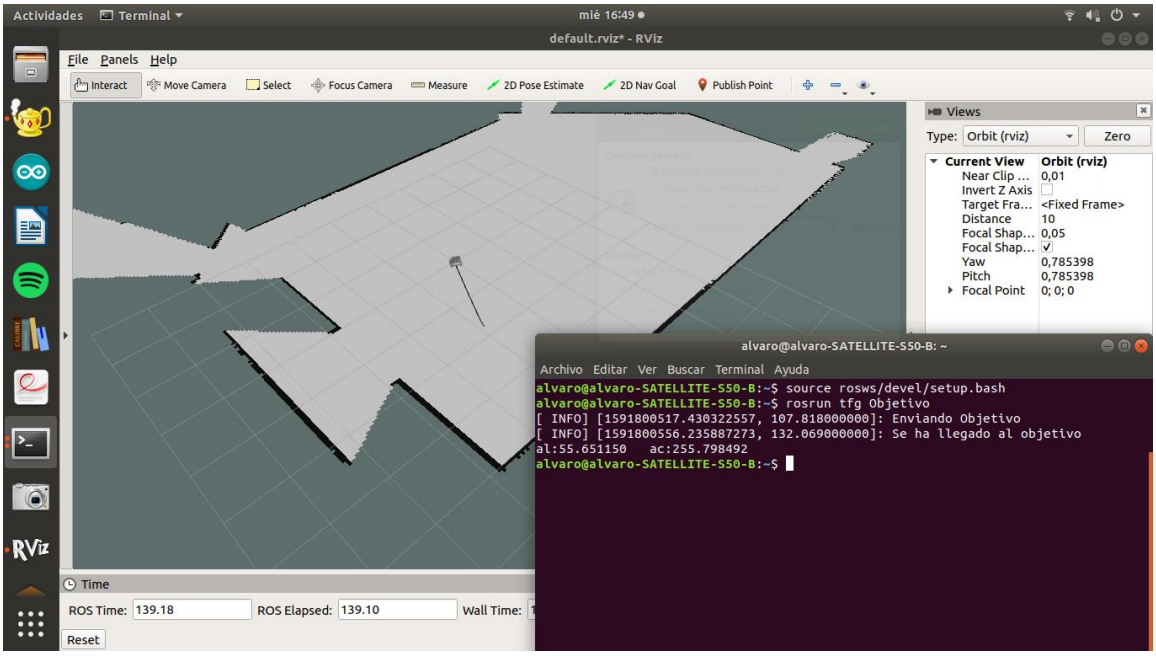

Figura 5.5: ROSbot llegando a la posición objetivo

5.1.3. Navegación por el entorno

Para probar la navegación autónoma de forma un poco mas exhaustiva se han simulado dos obstáculos con forma de cubo y cilindro, tal y como se puede ver en la siguiente imagen.

Se le enviarán tres posiciones distintas para poder ver como el robot es capaz de soltar los obstáculos adecuadamente. La línea verde es el camino que ha elegido el sistema de navegación hasta la posición objetivo.

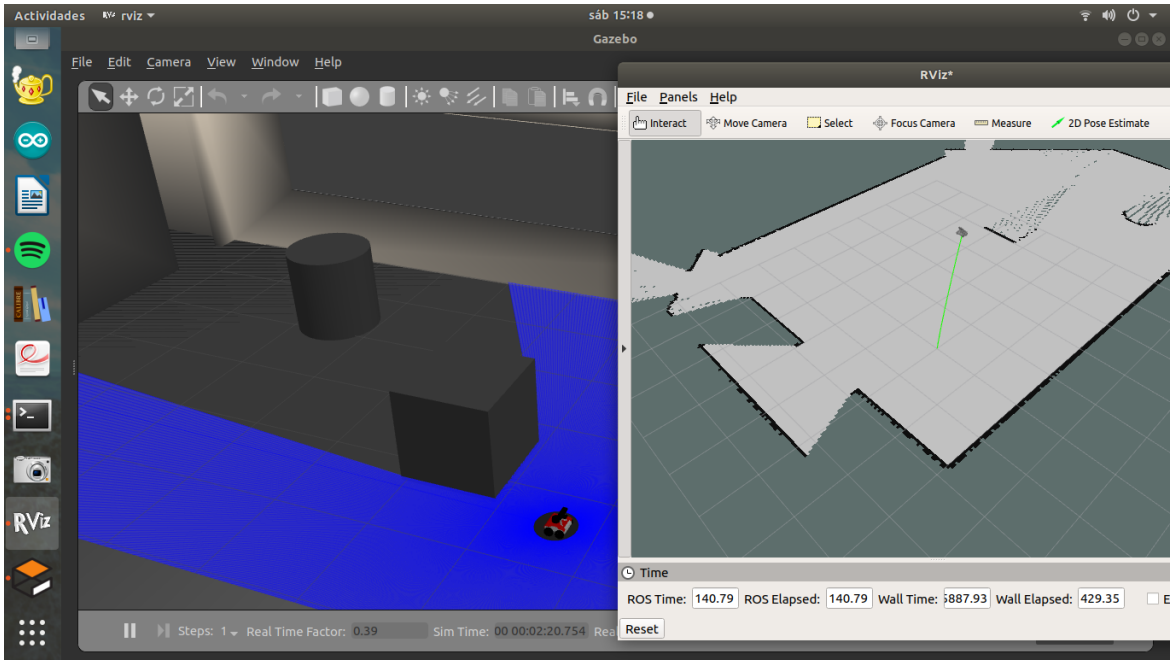

Figura 5.6: Primer tramo.

La siguiente gráfica muestra los valores, calculados por el sistema de navegación, que toman la velocidad lineal y la velocidad angular.

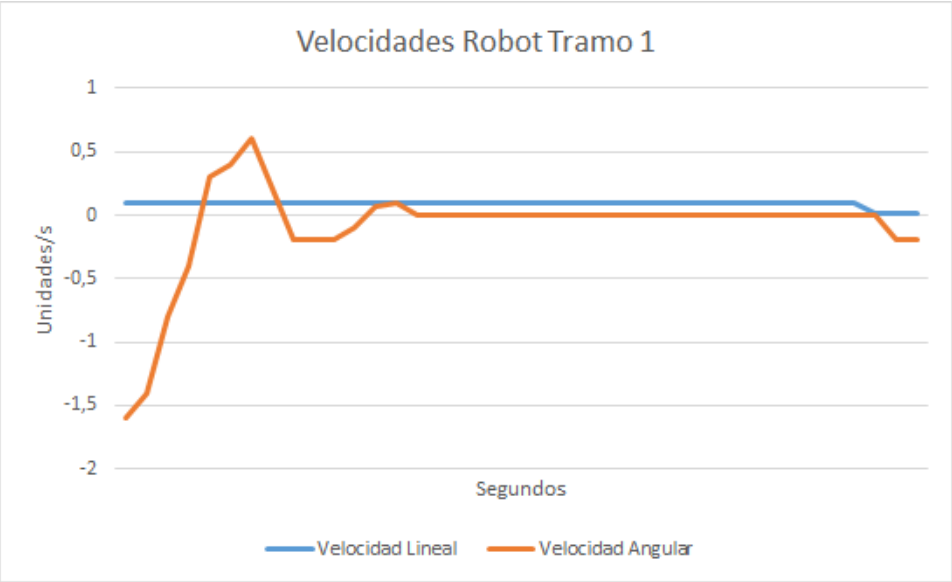

Figura 5.7: Velocidades primer tramo.

En las Figuras 5.8 y 5.9 se puede ver como el robot toma camino curvos para esquivar los obstáculos.

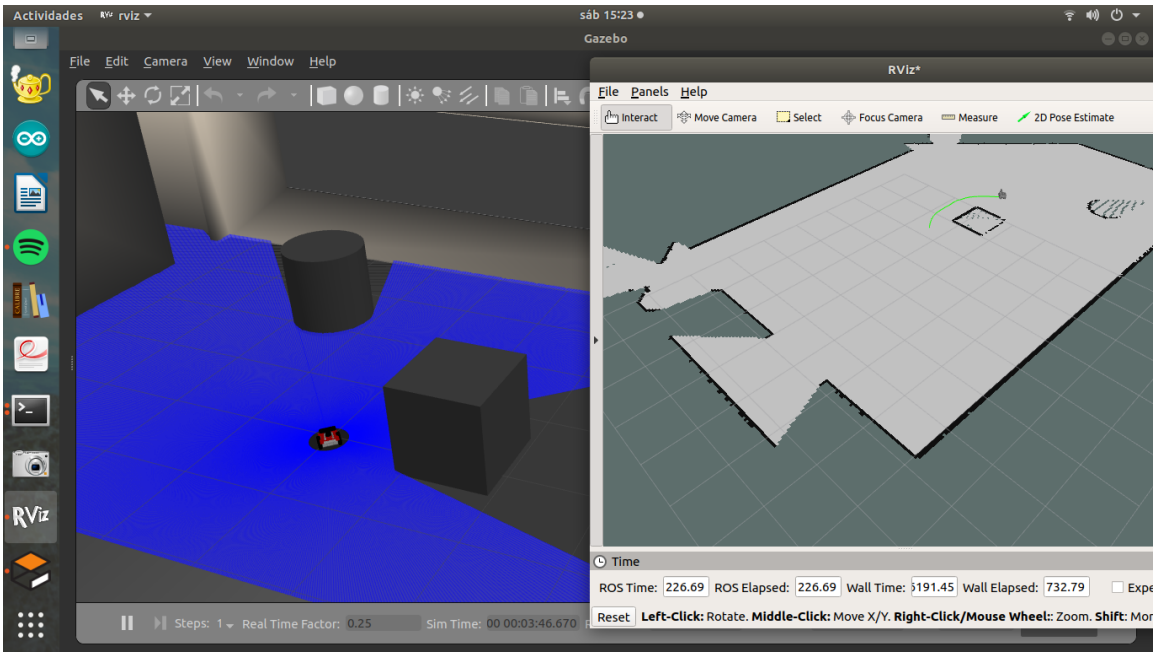

Figura 5.8: Segundo tramo.

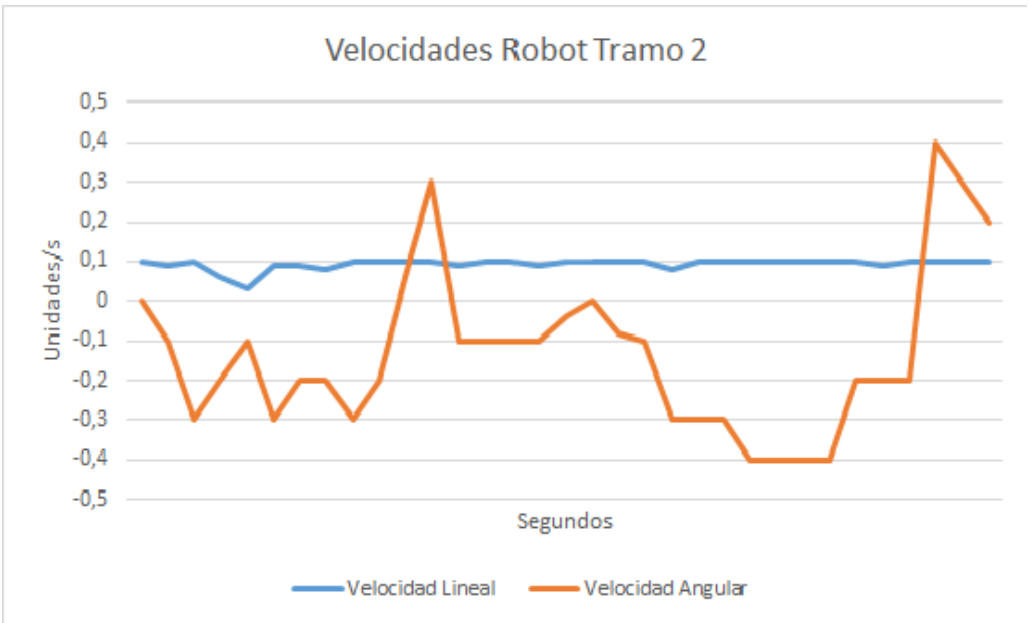

Figura 5.9: Velocidades segundo tramo.

Finalmente en las Figuras 5.10 y 5.11 se muestra el camino seguido en el último tramo así como las velocidades del robot para dicho tramo.

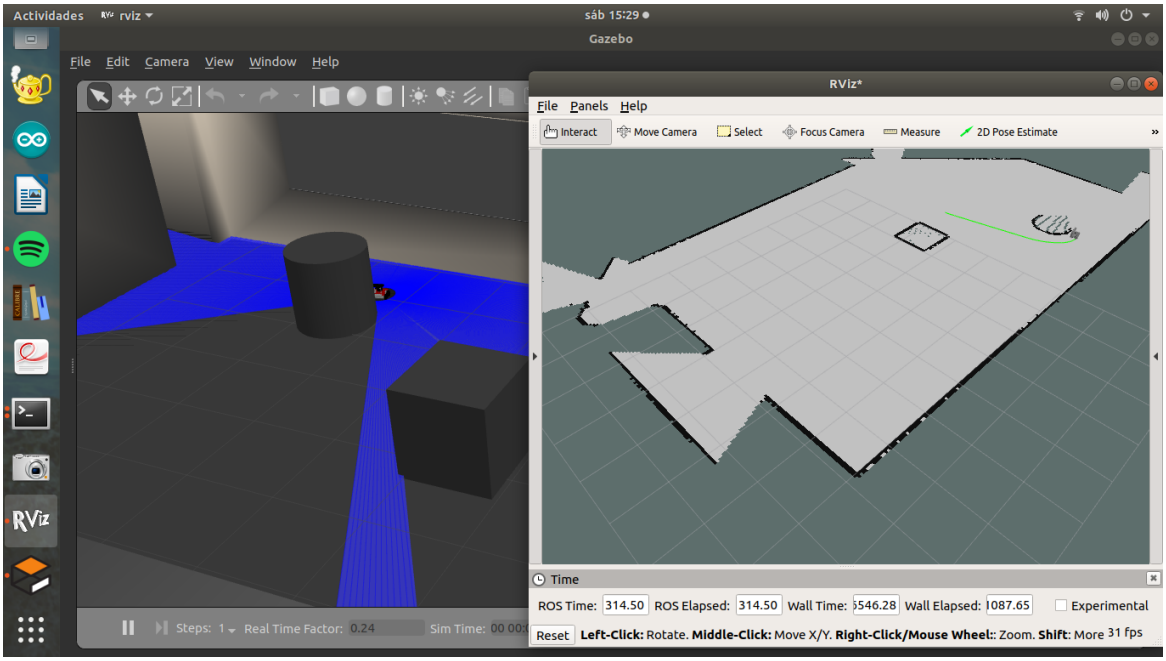

Figura 5.10: Tercer tramo.

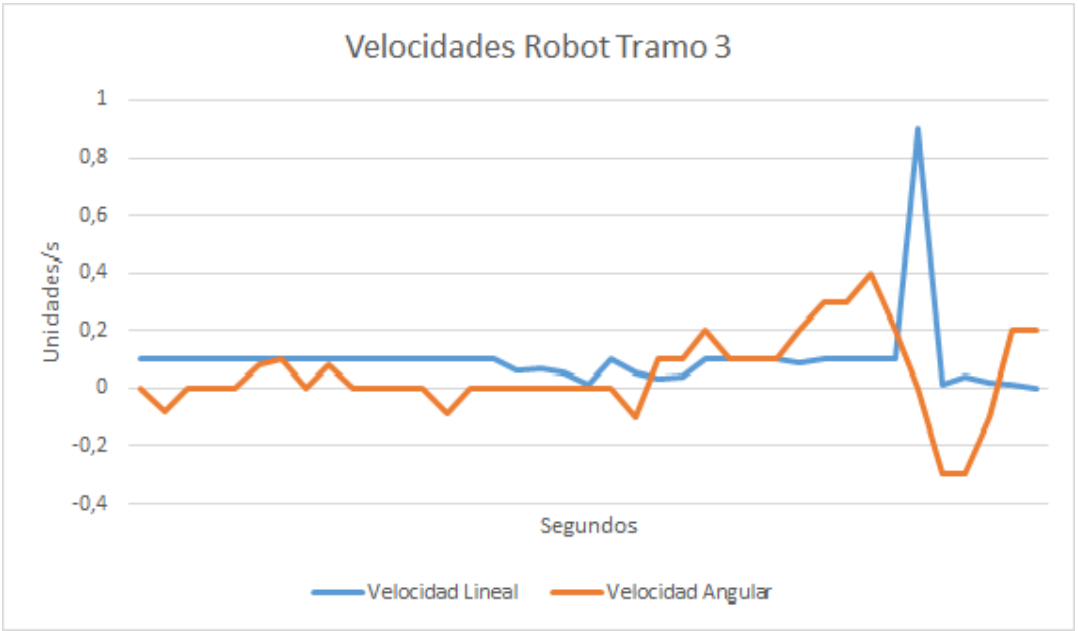

Figura 5.11: Velocidades tercer tramo.

5.2 Comunicación ROS-Arduino

Una vez realizada con éxito la simulación en Gazebo se añade la comunicación con Arduino. Para ello se crea un nodo de ROS en Arduino utilizando el paquete Rosserial. Tanto Leestatus.cpp como Objetivo.cpp calculan el acimut y la altitud del Sol utilizando la función CalAcAl, programada en el fichero CalAcAl.h, y envían los resultados al Arduino a través del tópico "alyac". Con estos datos y la orientación proporcionada por una brújula, Arduino calcula la referencia en ángulos entre 0º y 180º para los servos que mueven el pan and tilt. Una vez el pan and tilt está apuntando al Sol se leen los datos de los LDR, se pasan a lúmenes y se pasan de vuelta a través del tópico "datos_sensores"

En el siguiente experimento se desplazará el robot a cuatro posiciones diferentes y se mostrarán por pantalla los resultados. La Figura 5.12 muestra el inicio del pimer desplazamiento.

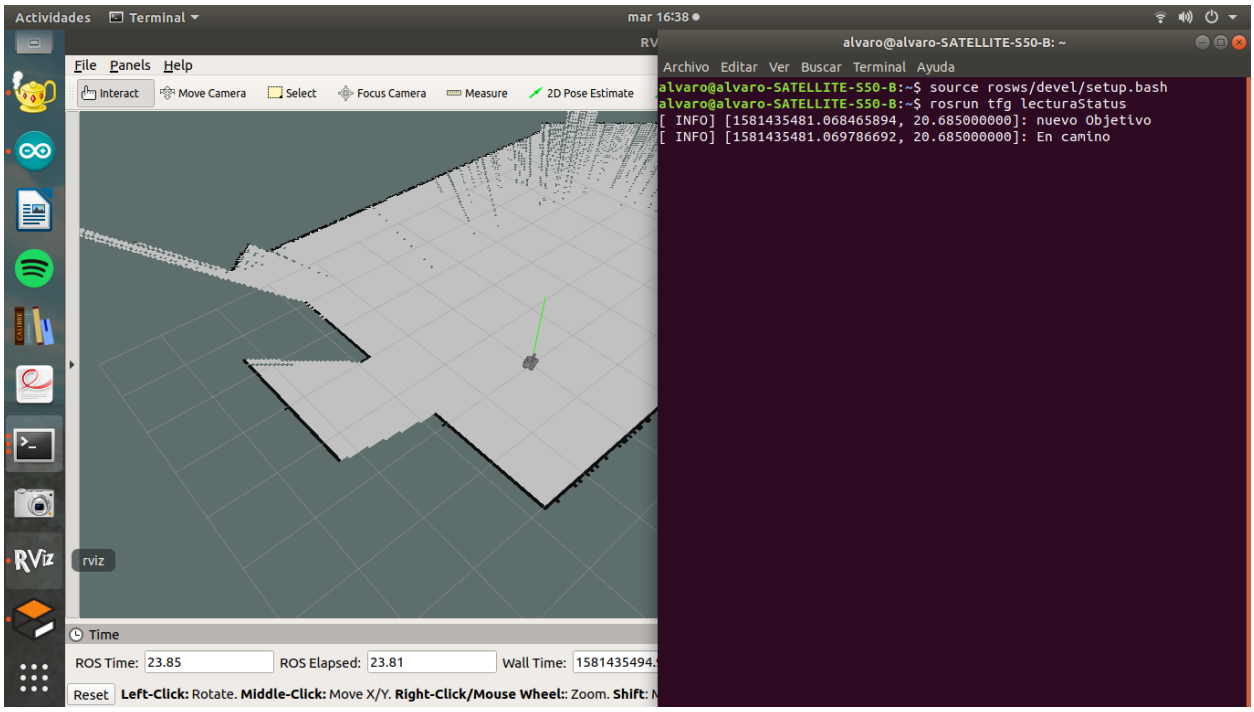

Figura 5.12: ROSbot iniciando el desplazamiento

En la Figura 5.13 se muestran los resultados obtenidos tras haber desplazado el robot a las cuatro posiciones diferentes. La brújula se ha movido manuelmente para que de diferentes datos en cada posición. De la misma manera se ha iluminado los LDR con una linterna cambiando la orientación de la misma para que la intensidad de la luz que incide en ellos varíe.

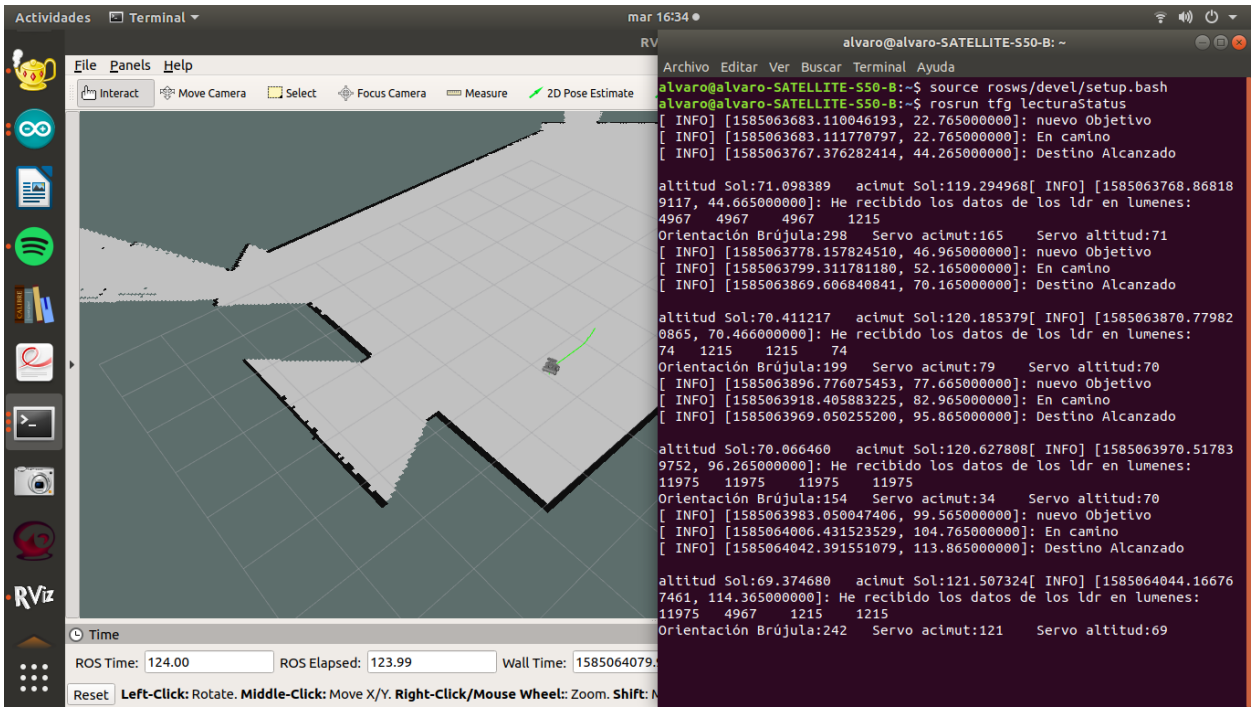

Figura 5.13: Comunicación ROS-Arduino

“Altitud Sol” y “Acimut Sol” se corresponden respectivamente con los cáculos de la altura y el acimut del Sol en grados. El valor “Orientación Brújula” en es el valor de la orientación obtenido por la brújula digital en grados, tomando como 0 el Norte e incrementandose en el sentido de las agujas del reloj. “Servo acimut” y “Servoaltitud” corresponden a la posición en grados a la que se deben mover los servos para apuntar hacia el Sol.

En las siguientes imágenes se muestran los servos en las diferentes posiciones del experimento.

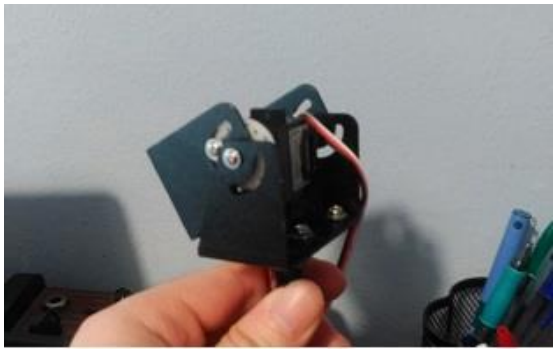

Figura 5.14: ser. acimut:15, ser. altitud: 0

Figura 5.15: ser. acimut:165, ser. altitud:71

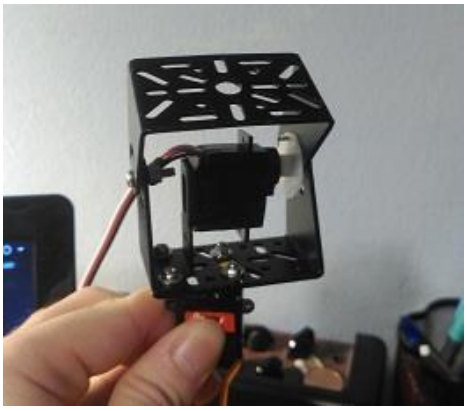

Figura 5.16: ser. acimut:79, ser. altitud:70

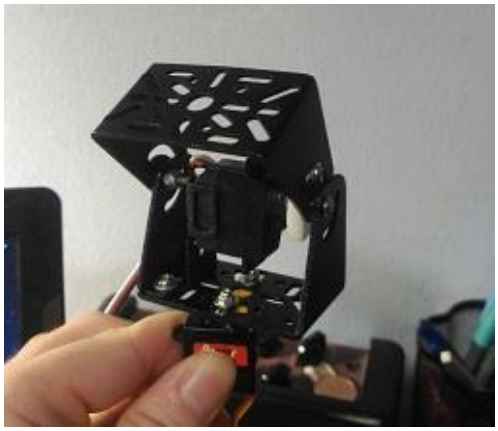

Figura 5.17: ser. acimut:34, ser. altitud:70

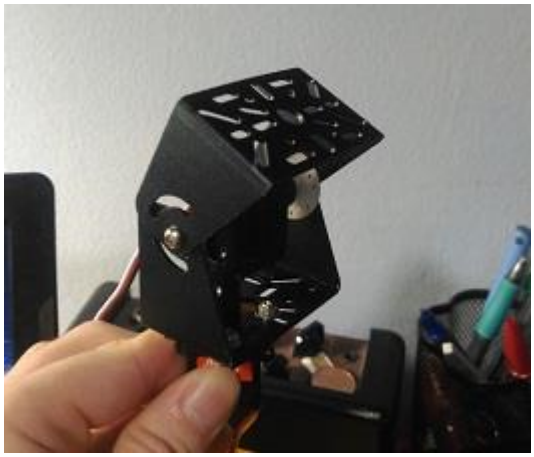

Figura 5.18: ser. acimut:121, ser. altitud: 69

5.3 Seguimiento

Cada 2 segundos Arduino lee el valor de la brújula y mueve los servos para que sigan apuntando al Sol, según el último dato de acimut recibido. La Figura 5.19 muestra los resultados de haber movido la brújula a diferentes posiciones, manteniendo el valor de acimut en 110. La brújula no puede aportar nuevos datos sobre la altitud del Sol, por ello el servo de la altitud se mantiene fijo.

"Acimut Sol", los puntos azules, se corresponden con el valor del acimut del Sol calculado. "Orientación Brújula", puntos naranjas, son los valores de la orientación obtenidos de la brújula. Por último "Servo Acimut", puntos grises, es al ángulo al que se mueve el servo para orientarse hacia el Sol.

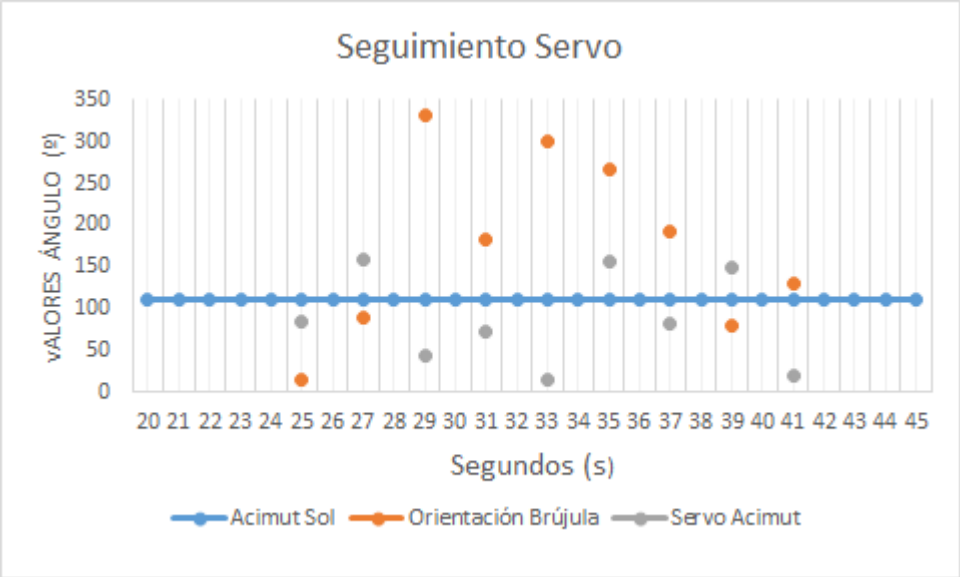

Figura 5.19: Seguimiento acimut del Sol

5.4 Cálculo de Servo Acimut y Servo Altitud

Acimut Sol y Orientación Brújula son ángulos que se miden en sentido horario siendo el Norte la dirección del ángulo 0. Servo Acimut sin embargo gira en sentido antihorario y la dirección de referencia es la del ángulo Orientación Brújula, ya que es el ángulo que indica la orientación del robot. Además Servo Acimut solo puede girar 180 grados. Debido a esto en algunas ocasiones el sentido de Acimut Sol y Servo Acimut será el mismo y en otros casos el contrario. Cuando el sentido de ámbos ángulos sea el mismo Altitud Sol y Servo Altitud serán iguales. Cuando el sentido sea el contrario Servo Altitud = 180º - Altitud Sol.

Si tomamos como ejemplo el caso del segundo 25 en la Figura 5.19 Acimut Sol es 110 y Orientación Brújula es 14. En este caso el sentido de Acimut Sol y Servo Acimut será el contrario y se calculará de la siguiente forma.

$$Servo\ Acimut = Orientación\ Brújula + (180 - Acimut\ Sol)$$

Que en este caso concreto da 84º.

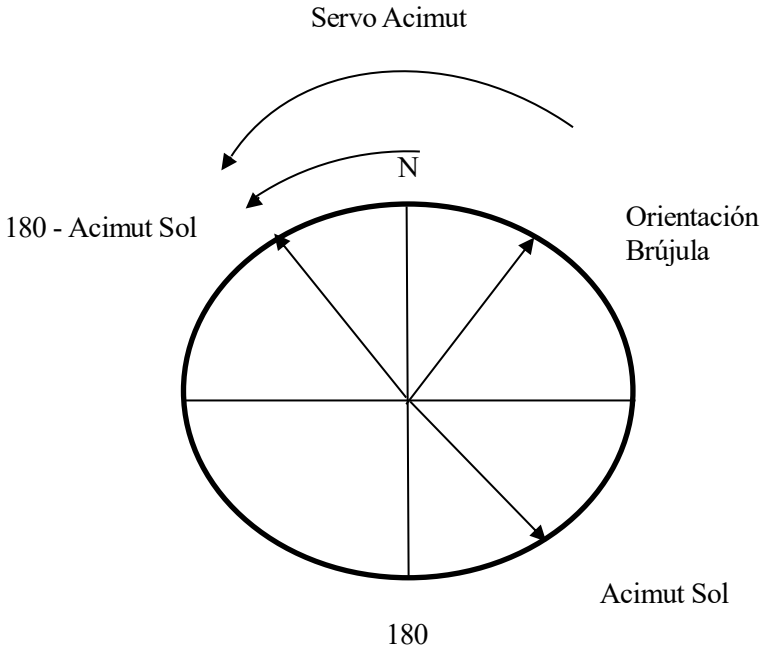

Figura 5.20: Cálculo Servo Acimut

6 CONCLUSIONES Y TRABAJOS FUTUROS

6.1 Conclusiones

Como conclusiones de este trabajo:

- Se ha simulado en Gazebo el robot ROSbot y el entorno Willow Garage.
- Se ha implementado SLAM para la navegación autónoma del robot.
- Se dan las posiciones objetivos desde Rviz y desde código.
- Se puede ver el robot, el mapa y el camino que va recorrer desde Rviz.
- Se han implementado las ecuaciones solares para calcular la altitud y el acimut del Sol.
- Se ha implementado un nodo de ROS en una placa Arduino.
- Se ha conseguido comunicación entre nodos de ROS ejecutados en un ordenador y el nodo en la placa Arduino mediante Rosserial.
- Se han creado tipos de mensajes de ROS propios para enviar datos desde el ordenador a la placa y viceversa.
- Se han controlado sensores y actuadores como la brújula digital, LDR y micro servos.
- Se ha caliabrado las resistencias LDR de forma experimental.

6.2 Futuros Trabajos

Como mejoras para un futuro trabajo se proponen:

1. Pasar del prototipo a un diseño definitivo para el hardware.
2. Pasar de la simulación a un robot real en un entorno físico.
3. Añadir más sensores como anemómetros o barómetros.
4. Añadir una cámara para hacer detención de personas.

Bibliografía

[1].“Navigation package summary”. [en línea] http://wiki.ros.org/navigation [capturado: 21 de marzo 2020].

[2].“Setup and configuration of the navigation stack in a robot”. [en linea] http://wiki.ros.org/navigation/Tutorials/RobotSetup [capturado: 21 de marzo 2020].

[3].Letin, J. (2015). Mastering ROS for robotics programming. Birminghan: Packt Publishing Ltd.

[4].”Calibrado de una LDR” [en linea] https://previa.uclm.es/ profesorado/ajbarbero/Practicas/Calibrado%20LDR%202003.pdf [capturado: 21 de marzo 2020].

[5].A. Franco Garcia (2016): “Posición del Sol en el sistema de referencia local”. [en linea] http://www.sc.ehu.es/sbweb/fisica3/celeste/sol/sol.html [capturado: 21 de marzo 2020].

A. Códigos ROS

Nombre	LecturaStatus.cpp
Descripción	Lee el estado de la trayectoria del robot en su camino hacia la ubicación objetivo. Una vez alcanzada calcula el acimut y la altitud del Sol y los envía al Arduino. Recibe de Arduino los datos de los sensores LDR.
Entradas	Tópico tipo *"move_base/status"* Tópico tipo *"move_base/goal"* Tópico tipo *"datos_sensores"*
Salidas	Tópico tipo *"alyac"*

```
23
24  #include "calAcAl.h"
25  #include "ros/ros.h"
26  #include <actionlib_msgs/GoalStatusArray.h>
27  #include <move_base_msgs/MoveBaseActionGoal.h>
28  #include <stdio.h>
29  #include <tfg/customsg.h>
30  #include <tfg/ldrdta.h>
31
32
33
34  int en_camino=0;
35  int primerObjetivo=0;
36
37  class pubclass {
38      ros::Publisher pub;
39      public:
40      pubclass(ros::Publisher p){pub=p;}
41      void leeStatus(const actionlib_msgs::GoalStatusArray::ConstPtr& status){
42          if(primerObjetivo==1){
43              if(status->status_list[0].status == 1  && en_camino==0){
44                  en_camino=1;
45                  ROS_INFO("En camino");
46              }
47              else if(status->status_list[0].status == 3  && en_camino==1){
48                  en_camino=0;
49                  ROS_INFO("Destino Alcanzado");
50                  pair<float,float> acal= calAcAl();
51                  float al = acal.first;
52                  float ac = acal.second;
53                  printf("al:%f   az:%f",al,ac);
54                  tfg::customsg datos;
```

```
55                  datos.num1=ac;
56                  datos.num2=al;
57                  pub.publish(datos);
58
59              }
60          }
61
62
63      }
64  };
65
66
67  void nuevoObjetivo(const move_base_msgs::MoveBaseActionGoal::ConstPtr& goal){
68
69      primerObjetivo=1;
70      ROS_INFO("nuevo Objetivo");
71  }
72
73  void fresp(const tfg::ldrdta::ConstPtr& val_sensores){
74
75      ROS_INFO("He recibido los datos de los ldr");
76      printf("%d   %d    %d    %d",val_sensores->val1,val_sensores->val2,val_sensores->val3,val_sensores->val4);
77      printf("angulo:%d   referencia acimut:%d    referencia altitud:%d",val_sensores->ang,val_sensores->mov,val_sensores->alt);
78      printf("\n");
79  }
80
81
82  int main(int argc, char **argv){
83      ros::init(argc,argv,"LecturaStatus");
84      ros::NodeHandle n;
85
86      ros::Publisher p=n.advertise<tfg::customsg>("alyac",1);
87      pubclass clasepub(p);
88      ros::Subscriber sub = n.subscribe("move_base/status", 1, &pubclass::leeStatus, &clasepub);
89      ros::Subscriber sub2 = n.subscribe("move_base/goal", 1, nuevoObjetivo);
90      ros::Subscriber sub3 = n.subscribe("datos_sensores", 1, fresp);
91
92      ros::Rate loop_rate(10);
93
94      while(ros::ok()){
95
96          ros::spinOnce();
97          loop_rate.sleep();
98      }
99  }
100
```

Nombre	Objetivo.cpp
Descripción	Determina la posición objetivo del robot. Una vez alcanzada calcula el acimut y la altitud del Sol y los envía al Arduino. Recibe de Arduino los datos de los sensores LDR.
Entradas	Tópico tipo *"datos_sensores"*
Salidas	Tópico tipo *"alyac"*

```
#include <ros/ros.h>
#include <move_base_msgs/MoveBaseAction.h>
#include <actionlib/client/simple_action_client.h>
#include "calAcAl.h"
#include <stdio.h>
#include <tfg/customsg.h>
#include <tfg/ldrdta.h>

typedef actionlib::SimpleActionClient<move_base_msgs::MoveBaseAction> MoveBaseClient;

void fresp(const tfg::ldrdta::ConstPtr& val_sensores){

    ROS_INFO("He recibido los datos de los ldr en lumenes:");
    printf("%d   %d    %d    %d",val_sensores->val1,val_sensores->val2,val_sensores->val3,val_sensores->val4);
    printf("\n");
    printf("Orientación Brújula:%d   Servo acimut:%d    Servo altitud:%d",val_sensores->ang,val_sensores->mov,val_sensores->alt);
    printf("\n");
}

int main(int argc, char** argv){
    ros::init(argc, argv, "Objetivos");
    MoveBaseClient ac("move_base", true);
    ros::NodeHandle n;
    ros::Publisher p = n.advertise<tfg::customsg>("alyac",1);
    ros::Subscriber sub = n.subscribe("datos_sensores", 1, fresp);

    while(!ac.waitForServer(ros::Duration(5.0))){
        ROS_INFO("Esperando move_base action server");
    }
    move_base_msgs::MoveBaseGoal goal[3];
```

```
    goal[0].target_pose.header.frame_id = "map";
    goal[0].target_pose.header.stamp = ros::Time::now();
    goal[0].target_pose.pose.position.x = -1.0;
    goal[0].target_pose.pose.position.y = -2.0;
    goal[0].target_pose.pose.position.z = 0.0;
    goal[0].target_pose.pose.orientation.x = 0.0;
    goal[0].target_pose.pose.orientation.y = 0.0;
    goal[0].target_pose.pose.orientation.z = 0.881636;
    goal[0].target_pose.pose.orientation.w = -0.471929;

    goal[1].target_pose.header.frame_id = "map";
    goal[1].target_pose.header.stamp = ros::Time::now();
    goal[1].target_pose.pose.position.x = -2.0;
    goal[1].target_pose.pose.position.y = -1.0;
    goal[1].target_pose.pose.position.z = 0.0;
    goal[1].target_pose.pose.orientation.x = 0.0;
    goal[1].target_pose.pose.orientation.y = 0.0;
    goal[1].target_pose.pose.orientation.z = 0.890484;
    goal[1].target_pose.pose.orientation.w = 0.455014;

    goal[2].target_pose.header.frame_id = "map";
    goal[2].target_pose.header.stamp = ros::Time::now();
    goal[2].target_pose.pose.position.x = -1.0;
    goal[2].target_pose.pose.position.y = 0.23;
    goal[2].target_pose.pose.position.z = 0.0;
    goal[2].target_pose.pose.orientation.x = 0.0;
    goal[2].target_pose.pose.orientation.y = 0.0;
    goal[2].target_pose.pose.orientation.z = 0.352570;
    goal[2].target_pose.pose.orientation.w = 0.935785;

    int contador = 0;

    ros::Rate loop_rate(10);


    while(contador < 3){

        ROS_INFO("Enviando Objetivo");
        ac.sendGoal(goal[contador]);
        ac.waitForResult();

        if(ac.getState() == actionlib::SimpleClientGoalState::SUCCEEDED){
            ROS_INFO("Se ha llegado al objetivo");
            pair<float,float> acal= calAcAl();
            float al = acal.first;
            float ac = acal.second;
            printf("al:%f   ac:%f",al,ac);
            printf("\n");
            tfg::customsg datos;
            datos.num1=ac;
            datos.num2=al;
            p.publish(datos);
            contador+=1;

        }
        else{
            ROS_INFO("Error");
        }
    ros::spinOnce();
    loop_rate.sleep();
    }
    return 0;
}
```

Nombre	CalAcAl.h
Descripción	Calcula el acimut y la altitud del Sol para una dia y hora concretos.
Salidas	Pair (Float acimut, Float altitud)

```
#include <time.h>
#include <iostream>
#include <stdlib.h>     /* abs */
#include <math.h>

using namespace std;
const float  PI=3.14159265358979f;

float calAngDia(int ano,int mes, int dia);
pair<float,float> calAcAl();

pair<float,float> calAcAl()
{
    time_t theTime = time(NULL);
    struct tm *aTime = localtime(&theTime);
    float Longitud=-5.9823;
    float Latitud=37.3881;
    Latitud= Latitud*PI/180;

    //Hora en decimal
    int hora=aTime->tm_hour;
    int min=aTime->tm_min;
    float horaactualoficial = (float)hora + (float)min/60;

    //Diferencia por posición minutos
    float diferenciaporposicion = abs(Longitud*4);

    //Diferencia por horario de Verano
    float diferenciapormes;
    int mes = aTime->tm_mon + 1;
    if(mes < 4 || mes > 10){
       diferenciapormes=60.0;
    }
    else {
        diferenciapormes=120.0;
    }

    //Ecuacion del tiempo
    int dia = aTime->tm_mday;
    int ano = aTime->tm_year + 1900;

    float angulodiario = calAngDia(ano,mes,dia);
    float declinacion = 0.006918-0.399912*cos(angulodiario)+0.070257*sin(angulodiario)-0.006758*cos(2*angulodiario)
    +0.000907*sin(2*angulodiario)-0.002697*cos(3*angulodiario)+0.001480*sin(3*angulodiario); //Angulo de declinación
    float Et=(0.000075+0.001868*cos(angulodiario)-0.032077*sin(angulodiario)-0.014615*cos(2*angulodiario)-0.04089*sin(2*angulodiario))*229.18;

    //Hora solar
    float diferenciatotal=diferenciapormes+diferenciaporposicion-Et;
    float Ts=horaactualoficial-(diferenciatotal/60);

    //Angulo diario
    float anguloHorario=((Ts-12)*15)*(PI/180);  // w

    //Altitud
    float al = asin(cos(Latitud)*cos(anguloHorario)*cos(declinacion)+sin(Latitud)*sin(declinacion));

    //Acimut
    float ac = acos((cos(Latitud)*sin(declinacion)-cos(anguloHorario)*sin(Latitud)*cos(declinacion))/cos(al));

    ac=ac*(180/PI);
    al=al*(180/PI);
```

```
64
65
66      if(anguloHorario>0){ac=360-ac;}
67
68
69      return make_pair(al,ac);
70  }
71
72  float calAngDia(int ano,int mes,int dia){
73
74      int tabla[12];
75      if(ano%4==0 && (ano%100>0 || ano%400==0)){
76          int tabla[] = {31, 29, 31,30, 31, 30, 31, 31, 30, 31, 30, 31};
77      }
78      else{
79          int tabla[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
80      }
81      int suma=0;
82      for (int i=0; i<mes-1; i++){
83          suma = suma + tabla[i];
84      }
85      int Juliano = suma + dia;
86
87      float angDiario = (2*PI/365)*((float)Juliano-1);
88      return angDiario;
89  }
90
```

Nombre	Gazebo_sim.launch
Descripción	Arranca la simulación en Gazebo del entorno Willow Garage junto al modelo del robot Rosbot. Tambien arranca el nodo SLAM y el move_base.

```
1   <?xml version="1.0" encoding="UTF-8"?>
2   <launch>
3
4     <include file="$(find rosbot_description)/launch/rosbot_gazebo.launch"></include>
5
6     <include file="$(find gazebo_ros)/launch/willowgarage_world.launch"></include>
7
8     <node name="rviz" pkg="rviz" type="rviz"/>
9
10    <node name="slam_gmapping" pkg="gmapping" type="slam_gmapping">
11      <param name="base_frame" value="base_link" />
12    </node>
13
14    <include file="$(find tfg)/launch/move_base.launch"></include>
15
16  </launch>
17
```

Nombre	Move_base.launch
Descripción	Fichero que carga todos los parámetros necesarios para el nodo move_base.

```
<?xml version="1.0"?>
<launch>
  <master auto="start"/>

  <node pkg="move_base" type="move_base" respawn="false" name="move_base" output="screen">
    <rosparam file="$(find tfg)/config/costmap_common_params.yaml" command="load" ns="global_costmap" />
    <rosparam file="$(find tfg)/config/costmap_common_params.yaml" command="load" ns="local_costmap" />
    <rosparam file="$(find tfg)/config/local_costmap_params.yaml" command="load" />
    <rosparam file="$(find tfg)/config/global_costmap_params.yaml" command="load" />
    <rosparam file="$(find tfg)/config/trajectory_planner.yaml" command="load" />

    <remap from="cmd_vel" to="cmd_vel"/>
    <remap from="odom" to="odom"/>
    <remap from="scan" to="/scan"/>
    <param name="move_base/DWAPlannerROS/yaw_goal_tolerance" value="1.0"/>
    <param name="move_base/DWAPlannerROS/xy_goal_tolerance" value="1.0"/>

  </node>
</launch>
```

Nombre	Costmap_common_params
Descripción	Fichero que carga todos los parámetros necesarios para el nodo move_base.

```
obstacle_range: 6.0
raytrace_range: 8.5
footprint: [[0.12, 0.14], [0.12, -0.14], [-0.12, -0.14], [-0.12, 0.14]]
map_topic: /map
subscribe_to_updates: true
observation_sources: laser_scan_sensor
laser_scan_sensor: {sensor_frame: laser, data_type: LaserScan, topic: scan, marking: true, clearing: true}
global_frame: map
robot_base_frame: base_link
always_send_full_costmap: true
```

Nombre	Global_costmap_params
Descripción	Fichero que carga todos los parámetros necesarios para el nodo move_base.

```
global_costmap:
  update_frequency: 2.5
  publish_frequency: 2.5
  transform_tolerance: 0.5
  width: 15
  height: 15
  origin_x: -7.5
  origin_y: -7.5
  static_map: false
  rolling_window: true
  inflation_radius: 2.5
  resolution: 0.01
```

Nombre	Local_costmal_params
Descripción	Fichero que carga todos los parámetros necesarios para el nodo move_base.

```
local_costmap:
  update_frequency: 5
  publish_frequency: 5
  transform_tolerance: 0.25
  static_map: false
  rolling_window: true
  width: 3
  height: 3
  origin_x: -1.5
  origin_y: -1.5
  resolution: 0.01
  inflation_radius: 1.0
```

Nombre	Trayectoria_planner
Descripción	Fichero que carga todos los parámetros necesarios para el nodo move_base.

```
TrajectoryPlannerROS:
  max_vel_x: 0.1
  min_vel_x: 0.01
  max_vel_theta: 2.5
  min_vel_theta: -2.5
  min_in_place_vel_theta: 0.25
  acc_lim_theta: 1.0
  acc_lim_x: 1.5
  acc_lim_Y: 1.5
  holonomic_robot: false
  meter_scoring: true
  xy_goal_tolerance: 0.1
  yaw_goal_tolerance: 0.1
  meter_scoring: true
  occdist_scale:  0.01
  pdist_scale: 0.4
  gdist_scale: 0.2
```

B. Código Arduino

Nombre	nodoArduino.ino
Descripción	Gira los servos del pan and tilt para que apunte hacia el Sol en función de los datos recibidos de Ros y la orientación obtenida de la brújula. Una vez leidos los datos de los LDR los envía a Ros.
Entradas	Tópico tipo *"alyac"*
Salidas	Tópico tipo *"datos_sensores"*

```
#include <ros.h>
#include <tfg/customsg.h>
#include <tfg/ldrdta.h>
#include <Wire.h>
#include <MechaQMC5883.h>
#include <Servo.h>
#include <math.h>

Servo s_acimut;
Servo s_altitud;

ros::NodeHandle  nh;

tfg::ldrdta val_sensores;

ros::Publisher chatter("datos_sensores", &val_sensores);

MechaQMC5883 qmc;

const float declinacion = -1;

const int sensor1 = A0;
const int sensor2 = A1;
const int sensor3 = A2;
const int sensor4 = A3;
```

```
int acimut = 15;
int altitud = 0;
int contador=0;

int valor1, valor2, valor3, valor4;

int movimiento,alt;

void moverServos(int angulo){
  if(acimut <= 180){
    if(angulo>acimut){
      if(angulo-acimut<180) {
        movimiento = angulo-acimut;
        if(movimiento>165) movimiento = 165;
        else if(movimiento<15) movimiento = 15;
        s_acimut.write(movimiento);
        alt = altitud;
        s_altitud.write(alt); }
      else{
        movimiento = angulo-(acimut+180);
        if(movimiento>165) movimiento = 165;
        else if(movimiento<15) movimiento = 15;
        s_acimut.write(movimiento);
        alt = 180-altitud;
        s_altitud.write(alt);}
      }
    else{
      movimiento =angulo-acimut+180;
      if(movimiento>165) movimiento = 165;
      else if(movimiento<15) movimiento = 15;
      s_acimut.write(movimiento);
      alt = 180-altitud;
      s_altitud.write(alt);}
      }
  else{
    if(angulo>acimut){
      movimiento = angulo-acimut;
      if(movimiento>165) movimiento = 165;
      else if(movimiento<15) movimiento = 15;
      s_acimut.write(movimiento);
      alt = altitud;
      s_altitud.write(alt); }
    else{
      if(acimut-angulo>180){
        movimiento = angulo-acimut+360;
        if(movimiento>165) movimiento = 165;
        else if(movimiento<15) movimiento = 15;
        s_acimut.write(movimiento);
        alt = altitud;
```

```
        s_altitud.write(alt);
        }
      else{
        movimiento = angulo-(acimut-180);
        if(movimiento>165) movimiento = 165;
        else if(movimiento<15) movimiento = 15;
        s_acimut.write(movimiento);
        alt = 180-altitud;
        s_altitud.write(alt);
        }
      }
  }
}

void AcimutyAltitud ( const tfg::customsg& datos){
  //Leer datos brujula
  int x,y,z;
  qmc.read(&x,&y,&z);
  //Calcular ángulo el ángulo del eje X respecto al norte
  float angulo = atan2(y, x);
  angulo = angulo * RAD_TO_DEG;
  angulo = angulo - declinacion;

  if(angulo < 0) angulo = angulo + 360;
  //mover servos
  acimut = (int)datos.num1;
  altitud = (int)datos.num2;

  moverServos(angulo);

  delay(1000);
  //enviar de vuelta los datos de los LCD
  valor1 = analogRead(sensor1);
  valor2 = analogRead(sensor2);
  valor3 = analogRead(sensor3);
  valor4 = analogRead(sensor4);

  valor1 = (int)calculoLumen(valor1);
  valor2 = (int)calculoLumen(valor2);
  valor3 = (int)calculoLumen(valor3);
  valor4 = (int)calculoLumen(valor4);

  val_sensores.val1=valor1;
  val_sensores.val2=valor2;
  val_sensores.val3=valor3;
  val_sensores.val4=valor4;

  val_sensores.ang=angulo;
  val_sensores.mov=movimiento;
  val_sensores.alt=alt;
```

```
  chatter.publish( &val_sensores);
}

void seguimiento(void){
  //Leer datos brujula
  int x,y,z;
  qmc.read(&x,&y,&z);
  //Calcular ángulo el ángulo del eje X respecto al norte
  float angulo = atan2(y, x);
  angulo = angulo * RAD_TO_DEG;
  angulo = angulo - declinacion;

  if(angulo < 0) angulo = angulo + 360;
  //mover servos

 moverServos(angulo);
 delay(500);
}

double calculoLumen(int lectura){
  double voltaje = lectura*5/1023;
  double resistencia = (10*5/voltaje) + 10;  //Resistencia en KO
  double lumen = pow(10,(2.1397-log10(resistencia))/0.1931);
  return lumen;
  }
```

```
ros::Subscriber<tfg::customsg> sub("alyac", AcimutyAltitud );

void setup()
{
  Wire.begin();
  Serial.begin(9600);
  qmc.init();
  nh.initNode();
  nh.advertise(chatter);
  nh.subscribe(sub);
  s_acimut.attach(9);
  s_acimut.write(15);
  s_altitud.attach(10);
  s_altitud.write(0);
}

void loop()
{
  nh.spinOnce();
  if(contador==10) {seguimiento(); contador=0;}
  else contador++;
  delay(500);

}
```