UNIVERSIDAD SAN FRANCISCO DE QUITO USFQ

Colegio de Ciencias e Ingenierías

Desarrollo de un Vehículo Autónomo
Fase 1: Implementación de Vehículo Autónomo en entorno de simulación usando ROS

Marlon Alexis Aucancela Proaño
William Andrés López Pilatuña

Ingeniería Electrónica

Trabajo de fin de carrera presentado como requisito
para la obtención del título de
Ingeniero Electrónico

Quito, 08 de mayo de 2020

UNIVERSIDAD SAN FRANCISCO DE QUITO USFQ

Colegio de Ciencias e Ingenierías

HOJA DE CALIFICACIÓN
DE TRABAJO DE FIN DE CARRERA

Desarrollo de un Vehículo Autónomo
Fase 1: Implementación de Vehículo Autónomo en entorno de simulación usando ROS

Marlon Alexis Aucancela Proaño

William Andrés López Pilatuña

Nombre del profesor, Título académico **René Játiva Espinoza, PhD**

Quito, 08 de mayo de 2020

DERECHOS DE AUTOR

Por medio del presente documento certifico que he leído todas las Políticas y Manuales de la Universidad San Francisco de Quito USFQ, incluyendo la Política de Propiedad Intelectual USFQ, y estoy de acuerdo con su contenido, por lo que los derechos de propiedad intelectual del presente trabajo quedan sujetos a lo dispuesto en esas Políticas.

Asimismo, autorizo a la USFQ para que realice la digitalización y publicación de este trabajo en el repositorio virtual, de conformidad a lo dispuesto en el Art. 144 de la Ley Orgánica de Educación Superior.

Nombres y apellidos: Marlon Alexis Aucancela Proaño

Código: 00134633

Cédula de identidad: 0602060725

Nombres y apellidos: William Andrés López Pilatuña

Código: 00124116

Cédula de identidad: 1723346159

Lugar y fecha: Quito, mayo de 2020

ACLARACIÓN PARA PUBLICACIÓN

Nota: El presente trabajo, en su totalidad o cualquiera de sus partes, no debe ser considerado como una publicación, incluso a pesar de estar disponible sin restricciones a través de un repositorio institucional. Esta declaración se alinea con las prácticas y recomendaciones presentadas por el Committee on Publication Ethics COPE descritas por Barbour et al. (2017) Discussion document on best practice for issues around theses publishing, disponible en http://bit.ly/COPETheses.

UNPUBLISHED DOCUMENT

Note: The following capstone project is available through Universidad San Francisco de Quito USFQ institutional repository. Nonetheless, this project – in whole or in part – should not be considered a publication. This statement follows the recommendations presented by the Committee on Publication Ethics COPE described by Barbour et al. (2017) Discussion document on best practice for issues around theses publishing available on http://bit.ly/COPETheses.

RESUMEN

El presente proyecto, que es parte de un trabajo mayor financiado por un poligrant, se enfocó en dotar a una plataforma móvil; construida para el efecto, con capacidades de exploración cognitiva de entornos y de navegación autónoma, equipándola con sensores varios, entre los que destaca un dispositivo LIDAR. El sistema de procesamiento del prototipo de este vehículo autónomo es un RaspBerry PI 4b. En este documento se describe el desarrollo de la prueba de concepto del vehículo autónomo, usando herramientas de código abierto tales como el sistema operativo robótico ROS, y los entornos de simulación Gazebo y RViz. Se probó también las funcionalidades básicas en el prototipo físico.

Palabras clave: vehículo autónomo, exploración de entornos, sensor LIDAR, ROS, RaspBerry PI 4b, Gazebo, RViz.

ABSTRACT

This project, which is part of a larger work financed by a poligrant, focused on equipping a mobile platform; built for the purpose, with cognitive exploration of environments and autonomous navigation capabilities, equipping it with various sensors, among which a LIDAR device stands out. The prototype processing system for this autonomous vehicle is a RaspBerry PI 4b. This document describes the development of the autonomous vehicle proof of concept, using open source tools such as the robotic ROS operating system, and the Gazebo and RViz simulation environments. The basic functionalities in the physical prototype were also tested.

Key words: autonomous vehicle, environment scan, LIDAR sensor, ROS, RaspBerry PI 4b, Gazebo, RViz.

TABLA DE CONTENIDO

ÍNDICE DE FIGURAS

INTRODUCCIÓN

Fue en el año de 1925 en las calles de Manhattan , cuando se observó por primera vez, según lo reporta el New York Times,

> “ un automóvil controlado por radio, el cual se maneja por las calles de Manhattan sin nadie al volante. Este vehículo controlado por radio puede encender su motor, cambiar de marcha y hacer sonar el pito como si una mano fantasma estuviese al volante” (Matus, 2017).

En los años siguientes se han ido reportando miles de apariciones de robots móviles autónomos (ARM), que se caracterizan por ser capaces de moverse de una posición a otra de forma autónoma.

Para el presente proyecto se ha planteado como objetivo el dotar a una plataforma móvil construida para el efecto capacidades de exploración cognitiva de entornos y de navegación autónoma, sin la necesidad de que una persona lo guíe de manera remota. El éxito de este proyecto lo garantizará que el vehículo autónomo en cuestión se desplace sin problemas desde un punto A hasta otro punto B, mapeando con un sensor LIDAR el entrono recorrido para sortear todos los obstáculos que se le puedan presentar en su camino y a la vez ir almacenando la información del lugar recorrido. Es importante resaltar, que este proyecto tiene como objetivo secundario el permitir a otros estudiantes de la Universidad San Francisco de Quito aplicar mejoras u optimizar el vehículo con nuevas tecnologías, siempre y cuando se mantenga el mismo objetivo de este proyecto.

DESARROLLO DEL TEMA

Para el desarrollo de este proyecto se requirió de un software en específico que nos permitió acoplar al vehículo un GPS y lecturas de sensores a un sistema robótico que posibilitaron el desplazamiento por distintos entornos o localizaciones mapeando la ruta y evitando obstáculos de forma autónoma. Este desarrollo permite que el proyecto pueda ser experimentado tanto para vehículos de bajo costo como la posibilidad de experimentarlo en un vehículo autónomo profesional.

En el desarrollo del funcionamiento del vehículo autónomo se destacó el software de robótica y programación ROS. Este software al tener licencia Open Source nos permitió trabajar con una variedad de librerías orientadas a la robótica. Para el presente proyecto orientamos principalmente la utilización de este software en la construcción o mapeo de entornos utilizando el sensor incorporado en el vehículo, para de esta manera cumplir los objetivos planteados.

El principal responsable de conseguir que el vehículo al mapear su entorno evite todo obstáculo que se le presentó en su camino e el láser Light Detection And Ranging (LIDAR) 2D el cual cuenta con una observación inmediata al vehículo, la misma que se compara con la morfología generada y acumulada en el mapa, para de esta manera evaluar mediante estimaciones y observaciones simultaneas el trayecto recorrido. Lamentablemente el acceder a obtener un LIDAR es considerablemente costosa, lo cual genera limitaciones en cuanto a masificaciones

Habiendo explicado el desarrollo de este proyecto de carácter académico, procedemos a describir en lo que resta del documento, las secciones y etapas realizadas para conseguir el éxito obtenido experimentalmente. Es importante mencionar que se utilizó un reducido número de sensores y dispositivos de rango bidimensional, ya que fueron suficientes para la

construcción de entornos en tiempo real y permitieron el desplazamiento sin inconvenientes del ARM en cuestión.

Materiales.

Dado que el prototipo fue construido desde cero, es decir, fue ensamblado a partir de módulos independientes, a continuación se muestran dichos módulos y una descripción de su funcionamiento respectivamente.

Inicialmente, procedemos a la configuración del sensor principal de este proyecto el cual es el LIDAR, acrónimo que corresponde a las siglas de Laser Imaging Detection and Ranging. Este emite haces de rayos de luz láser infrarroja, no peligrosa en las condiciones de uso previstas, que "rebotan" en los objetos. Consiste de manera muy básica en un foco emisor de haces de rayos láser infrarrojos que no se ven por tanto no son peligrosos para la vista y de una lente receptora infrarroja capaz de ver esos haces láser.

Figura # 1. RPLIDAR A2

El sistema puede realizar escaneos 2D de 360 grados dentro de un rango de seis metros. Los datos de nubes de puntos 2D generados se pueden usar en mapeo, localización y modelado de objetos o entornos. El procesador del LIDAR obtiene una nube de puntos del entorno, con la que la computadora procesa una imagen tridimensional en tiempo real, que se actualiza. Lo

más importante de esta nube de puntos es que para cada uno de estos se conoce su posición precisa en el espacio y la distancia que hay hasta él. De esta manera se pueden prever las situaciones que se producirán (por ejemplo el movimiento de otros objetos cuya trayectoria se cruzará con la nuestra), o determinar si existe peligro de rozar o impactar contra algo.

Antes de salir de la fábrica, cada RPLIDAR A2 ha pasado pruebas estrictas para garantizar que la potencia de salida del láser cumpla con los estándares de la Clase I de la FDA. De estos estándares se destacan, la frecuencia de escaneo típica del RPLIDAR A2 360 ° Laser Scanner es 10hz (600rpm). En esta condición, la resolución será de 0.9 grados. La frecuencia de escaneo real se puede ajustar libremente dentro del rango de 5-15 [Hz] según los requisitos de los usuarios. Puede tomar hasta 4000 muestras de láser por segundo con alta velocidad de rotación y equipado con la tecnología patentada OPTMAG de SLAMTEC elimina la limitación de la vida útil del sistema tradicional LIDAR para trabajar de manera estable durante mucho tiempo.

Para la puesta en marcha se debe conectar el puerto UART que tiene el sensor al correspondiente lado del adaptador USB que trae de fábrica, así mismo mediante un cable que convierte de micro USB a USB se hace la conexión con el computador o RaspBerry que estemos usando.

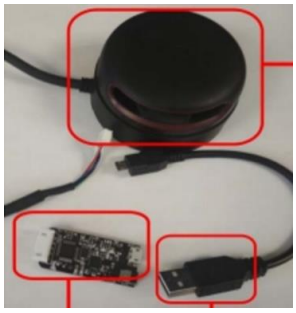

Figura # 2. Conexiones del RPLIDAR A2

Es necesario tener instalado cualquiera de los sistemas operativos basados en Linux, se nos recomienda ya sea Ubuntu o Raspbian, esta última de preferencia si se está utilizando la tarjeta RaspBerry Pi (Correa, 2018).

En la página de ROS.org se encuentra la librería rplidar la cual nos permite controlar el sensor RPLIDAR en ROS. Esta librería es descargable por código abierto ya que cuenta con licencia MIT que permite su libre distribución y uso.

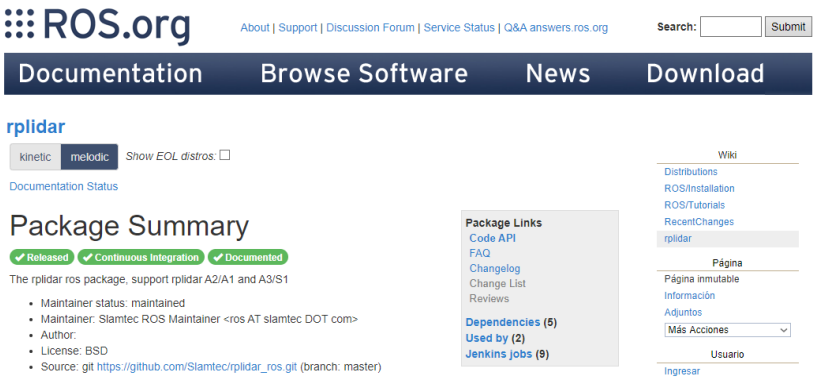

Figura # 3. Página Web de ROS

Entrando en una descripción mejor detallado del funcionamiento de este láser, el LIDAR durante cada proceso de medición emite una señal láser infrarroja modulada la misma que es reflejada por el objeto a detectar. Esta señal de retorno es luego muestreada por el sistema de adquisición de visión en el LIDAR y el DSP incorporado en este laser comienza a procesar los datos de muestra y genera el valor de distancia y el valor de ángulo entre el objeto y LIDAR a través de la interfaz de comunicación. Cuando es conducido por el motor del sistema, el núcleo del escáner de rango girará en sentido horario y realizará el escaneo de 360 grados para el entorno actual. El RPLIDAR A2 adopta el sistema de coordenadas de la mano izquierda como se muestra en las gráficas siguientes.

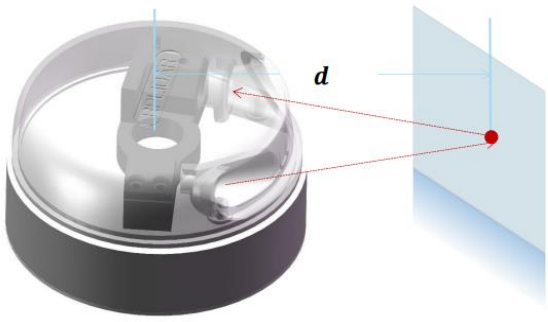

Figura # 4. Funcionamiento RPLIDAR A2

El punto muerto delante de los sensores es el eje x del sistema de coordenadas. El origen es el centro giratorio del núcleo del escáner de rango. El ángulo de rotación aumenta a medida que gira en sentido horario.

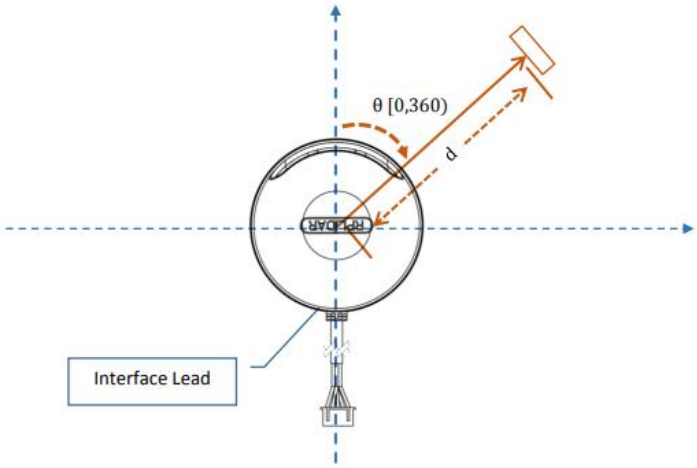

Figura # 5. Mediciones del RPLIDAR A2

Durante el proceso de trabajo, el LIDAR generará los datos de muestreo a través de la interfaz de comunicación. Cada dato de punto de muestra contiene la información en la siguiente gráfica.

● For Model A2M7/A2M8 Only

Item	Unit	Min	Typical	Max	Comments
Distance Range	Meter(m)	0.15	-	8	Based om white objects with 70% reflectivity
Angular Range	Degree	-	0-360	-	-
Distance Resolution	mm	-	<0.5	-	<1.5 meters
			<1% of the distance		All distance range*
Angular Resolution	Degree	0.45	0.9	1.35	10Hz scan rate
Sample Duration	Millisecond(ms)	-	0.25	-	-
Sample Frequency	Hz	2000	4000	4100	
Scan Rate	Hz	5	10	15	The rate is for a round of scan. The typical value is measured when RPLIDAR takes 400 samples per scan

- For Model A2M7/A2M8 Only

Item	Unit	Min	Typical	Max	Comments
Laser wavelength	Nanometer(nm)	775	785	795	Infrared Light Band
Laser power	Milliwatt (mW)	-	3	5	Peak power
Pulse length	Microsecond(us)	60	87	90	-
Laser Safety Class	-	-	FDA Class I	-	-

Figura # 6. Datos de muestreo del RPLIDAR A2

El módulo empleado en el control de los motores del prototipo es el driver L298n, implementando una configuración Puente-H. Este driver es sumamente versátil al momento de prototipar, pues su funcionamiento es bastante sencillo permitiendo controlar hasta dos motores DC, es compatible con la mayoría de microcontroladores y microprocesadores. Este dispositivo permite controlar a dirección de giro de cada motor a partir de dos pines de entrada digital, adicionalmente es posible controlar la velocidad de giro empleando un pin de entrada del tipo PWM. Este driver puede ser alimentado con un voltaje DC en el rango de [3-35] V y es capaz de soportar hasta una tensión de 2A.

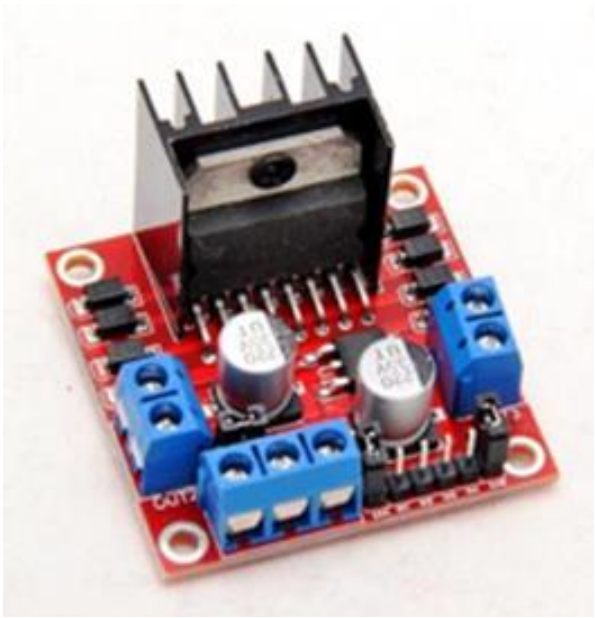

Figura # 7. Driver L298n

Otra gran ventaja de este dispositivo es que incluye un regulador de tensión constante de 5V, con el cual se puede alimentar otros dispositivos del prototipo. Para poder utilizar este regulador es necesario que este colocado el jumper que se observa en la siguiente figura.

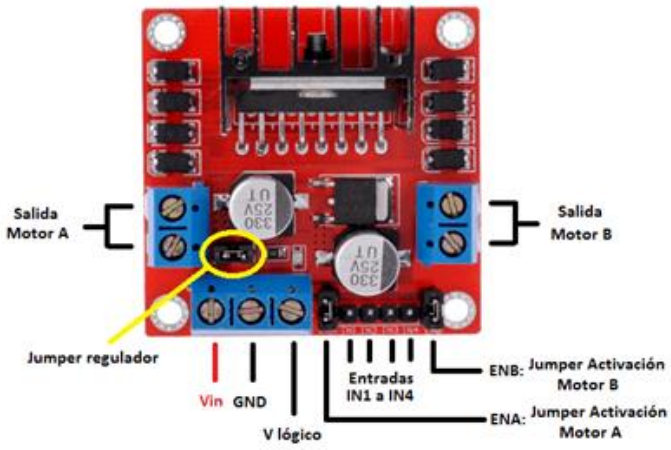

Figura # 8. Conexiones en el Driver L298n

El cerebro del vehículo autónomo se concentra en el microcontrolador RaspBerry Pi 4b, esta versión es la última y más potente de RaspBerry. Implementa el procesador Cortex A-72 de cuatro núcleos a 1-5 [GHz], es decir, tres veces más potente que las versiones previas, siendo comparable con un ordenador de gama baja, con una memora RAM de 4GB, esta nueva versión implementa conectores micro-usb y USB tipo C para mejorar su rendimiento siendo capaz de usar dispositivos USB a 1.2 [A]. Por otro lado, en cuanto al software, usa el sistema operativo basado en Debian 10 Buster (basadas en LINUX) con una interfaz más moderna y amigable.

Figura # 9. Microprocesador RaspBerry PI 4b

Se ha incorporado en el vehículo autónomo dos pares de motores DC de 5V con sus respectivas llantas para realizar su desplazamiento.

Figura # 10. Par Motores-llanta

Como fuente de energía para el vehículo autónomo se utiliza dos baterías de 11.1 [V] a 1000 [mA] de tres celdas.

Figura # 11. Baterías

Para la carga del vehículo autónomo se dispone como se muestra en la figura siguiente de un Notebook Power Adapter y en la figura trece de un cargador balanceado de baterías tipo LiPo de uno a cuatro celdas.

Figura # 12. Notebook Power Adapter

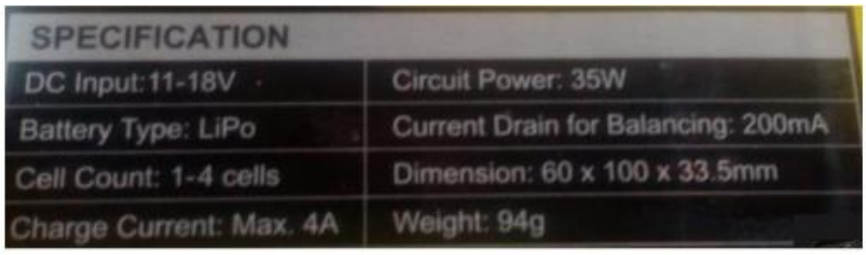

Figura # 13. Descripción de las baterías tipo LiPo de uno a cuatro celdas

El sistema de desplazamiento del vehículo autónomo se basa en un sistema de tracción en cuatro ruedas, totalmente independientes. En términos generales, el vehículo implementa cuatro motores, es decir, se utilizan dos driver L298n los mismos que son alimentados por una batería de LiPo de 11.1 [V] de tres celdas. Destacamos que los drivers están controlados por el hardware central que es la tarjeta RaspBerry PI 4b. A continuación, se presenta el esquema de conexiones y hardware empleado.

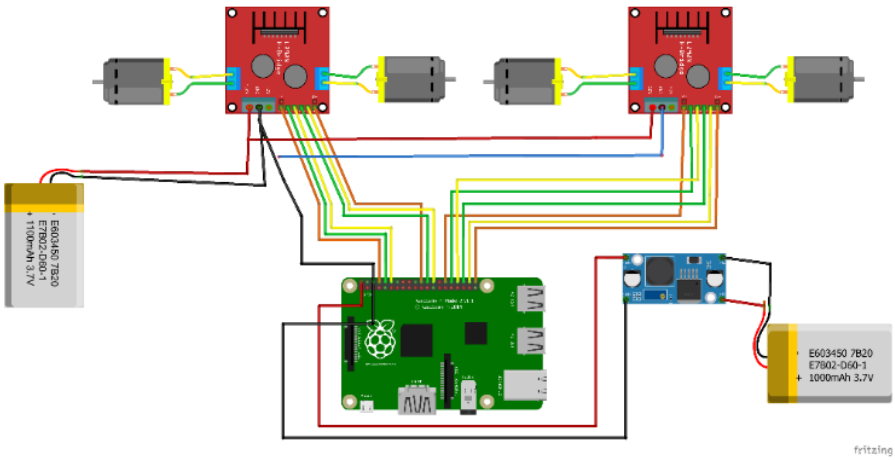

Figura # 14. Hardware Vehículo Autónomo

Entorno de simulación en RViz y Gazebo

1. Desarrollo y utilización del entorno de simulación.

1.1. Lo primero fue crear un nuevo entorno de trabajo (workspace). $ mkdir -p ~/catkin_ws/src $ cd ~/catkinp_ws/ * catkinp_ws es el nombre de nuestro nuevo entorno de trabajo $ catkin_make $ source devel/setup.bash

Para asegurarse de que nuestro espacio de trabajo esta superpuesto correctamente por el script de configuración, debemos asegurarnos de que la variable de entorno ROS_PACKAGE_PATH incluya el directorio en el que se encuentra $ echo $ROS_PACKAGE_PATH.

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

1.2. Para el desarrollo de este entorno de simulación se requiere instalar los paquetes de distribución libre del simulador Gazebo 3D y 3D RViz dependientes para el control del robot Turtlebot3. $ sudo apt-get install ros-kinetic-joy ros-kinetic-teleop-twist-joy ros-kinetic-teleop-twist-keyboard ros-kinetic-laser-proc ros-kinetic-rgbd-launch ros-kinetic-depthimage-to-laserscan ros-kinetic-rosserial-arduino ros-kinetic-rosserial-python ros-kinetic-rosserial-server ros-kinetic-rosserial-client ros-kinetic-rosserial-msgs ros-kinetic-amcl ros-kinetic-map-server ros-kinetic-move-base ros-kinetic-urdf ros-kinetic-xacro ros-kinetic-compressed-image-transport ros-kinetic-rqt-image-view ros-kinetic-gmapping ros-kinetic-navigation ros-kinetic-interactive-markers. $ cd ~/catkinp_ws/src/ $ git clone https://github.com/ROBOTIS GIT/turtlebot3_msgs.git $ git clone -b kinetic-devel https://github.com/ROBOTIS-GIT/turtlebot3.git

1.3. Al finalizar la instalación, se nos apareció dos paquetes iniciales necesarios para la simulación. Sin embargo fue necesario también instalar un tercer paquete de simulación que es exclusivo del robot TURTLEBOT3 waffle. $ cd ~/catkinp_ws/src/ $ git clone https://github.com/ROBOTIS-GIT/turtlebot3_simulations.git. Instalados

los paquetes, procedimos a construir estos paquetes en el Workspace. $ cd ~/catkinp_ws/ /* Esto nos dirige al directorio donde se encuentran nuestros paquetes. $ ls src /* Al escribir este comando en el directorio que se nos dirigió, nos muestra la lista de los paquetes que se encuentran en nuestro workspace. $ catkin_make /* Este comando nos construye los paquetes en nuestro workspace. Cada vez que se hace una actualización, se nos hizo necesario al final correr el siguiente comando para actualizar el workspace. $. ~/catkin_ws/devel/setup.bash /* Este comando actualiza el worksapce. Nota: para poder implementar y ejecutar los comandos siguientes es necesario que en cada terminal que abramos, corramos todos estos comandos descritos ya que al ser exportaciones de software libre, continuamente debemos actulizar nuestro workspace de catkin. Para facilidad de entendimiento de este paso, a continuación muestro los comandos sin comentarios a utilizar. $ cd ~/catkinp_ws/ $ git clone…. Paquete a instalar/* Escribir este comando solo cuando desea instalar un paquete, para correr nodos o archivos .launch no es necesario. $ ls src $ catkin_make $. ~/catkinp_ws/devel/setup.bash. TurtleBot3 admite un entorno de desarrollo que se puede programar y desarrollar con un robot virtual en la simulación. El primer entorno en el que exploramos, se usó un nodo falso y la herramienta de visualización 3D RViz.

1.4. El método de nodo falso nos fue útil para probar con el modelo de robot exportado y sus movimiento, pero no se puede usar sensores. Lo primero fue exportar nuestro robot a este primer entorno y ejecutar los archivos .launch para ejecutar este nodo falso. Nota: para ejecutar los archivos .launch y que funcionen, es importante haber seguido los pasos descritos en el punto 3. Hecho esto, escribimos $ export TURTLEBOT3_MODEL=waffle $ roslaunch turtlebot3_fake turtlebot3_fake.launch. Esto ejecutará el software RVIZ 3D, en el cual se observará en el centro de la navegación al robot waffle exportado. (Cuando se corre por primera vez, el software

es relativamente lento). El primer proceso de simulación programado y realizado fue mover el robot con las siguientes teclas. W (Hacia Adelante sin detenerse), S (Detener al robot), A (Girar a la Izquierda), D (Girar a la derecha), X (Retroceder sin detenerse). Para conseguir esto, en nuevo terminal realizamos el proceso del tercer literal de esta sección. $ export TURTLEBOT3_MODEL=waffle $ roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch. Nota: es importante tener activo este terminal (tener el cursor titilando en este terminal) para realizar los movimientos del robot con el teclado.

1.5. El segundo entorno en el que vamos a explorar se usa el simulador de robot 3D Gazebo. Si necesitamos probar SLAM (Construcción del entorno con sensores y laser) y Navegación, la mejor opción fue usar Gazebo ya que nos permitió utilizar nuestro LIDAR en tiempo real.En esta opción también funcionan sensores como IMU, LDS y cámara en la simulación. En este entorno existen dos formas de simular usando Gazebo.

- El primer método es usar con ROS a través del paquete turtlebot3_gazebo.
- El segundo método es usar solo Gazebo y el robot físico como tal sin usar ROS. (Este no corresponde a nuestro entorno de simulación virtual).

En este entorno de simulación utilizamos dos ambientes (exportados del código abierto). El mapa Turtlebot3 World se utilizó principalmente para realizar pruebas de SLAM y navegación en simulación. Con los siguientes comandos obtuvimos este mapa en Gazebo3D. En un terminal nuevo siguiendo el proceso descrito en el tercer literal, ejecutamos. $ export TURTLEBOT3_MODEL=waffle $ roslaunch turtlebot3_gazebo turtlebot3_world.launch. El mapa Casa TurtleBot3 se lo utilizó para pruebas relacionadas con el rendimiento de tareas más complejas. Con los siguientes comandos

obtuvimos este mapa en Gazebo3D. En un terminal nuevo siguiendo el proceso descrito en el tercer literal, ejecutamos. $ export TURTLEBOT3_MODEL=waffle $ roslaunch turtlebot3_gazebo turtlebot3_house.launch. En estos dos mapas de nuestro segundo entorno de simulación, al igual que lo hicimos antes, el primer proceso de simulación programado y realizado fue mover el robot con las siguientes teclas. W (Hacia Adelante sin detenerse), S (Detener al robot), A (Girar a la Izquierda), D (Girar a la derecha), X (Retroceder sin detenerse). Nota: antes de continuar con el desarrollo del entorno de simulación, es importante recalcar que el proceso para ejecutar los distintos nodos necesarios en cada uno de los procesos mostrados en este documento están descritos en el tercer literal. En muchos previos apartados se menciona que se deben seguir estos procesos en cada nuevo terminal que abramos para poder ejecutar los nodos que intervienen, sin embargo a este punto consideramos explicar que existe otro método más corto, donde no se vuelve a construir los paquetes del workspace, que al que se describe en el tercer literal. Este otro método simplemente se basa en actualizar o reabrir el workspace con los paquetes ya construidos. Recopilando el problema para una mejor explicación, cada vez que deseábamos llamar a ejecución un nodo en un terminal nuevo obteníamos como respuesta que los nodos no existían en el paquete, es por eso que en cada nuevo terminal se volvía a construir los paquetes y actualizar el workspace y de esta manera ya no volver a tener este error.

Continuando con el segundo método y utilizando como ejemplo el de habilitar el nodo que nos permite manejar el robot en el entorno de simulación Gazebo con el teclado, tan solo basta en un nuevo terminal actualizar el archivo devel, de la siguiente manera. $ cd ~/catkinp_ws $ source devel/setup.bash $ roscd turtlebot3_gazebo $ export TURTLEBOT3_MODEL=waffle $ roslaunch turtlebot3_teleop

turtlebot3_teleop_key.launch. Con estos comandos, a nuestro robot lo movilizamos por el mapa que escogimos utilizando el teclado de la PC en el entorno de simulación.

Conducción autónoma en el entorno de simulación Gazebo.

El familiarizarnos con la programación (nivel intermedio) en ROS, fue un proceso de muchas horas de investigación en la web, autoaprendizaje con videos en la web y recopilación de documentos (PDF) referentes a la construcción en ROS de un auto completamente autónomo, de los cuales muchos de estos documentos estaban escritos en idioma inglés, árabe y persa. Sin previo estudio alguno o mínimo aprendizaje en el entorno educativo sobre el tema, se convirtió en un gran reto el desarrollo de este proyecto. En esta sección, la cual es la más importante en cuanto a los objetivos de este proyecto, se muestra los procesos de programación y acoplamiento de paquetes, tópicos y nodos necesarios para la conducción autónoma de nuestro TURTLEBOT3 modelo waffle en el entorno de simulación Gazebo. Todos los nodos mostrados previamente se comunicaron entre sí a través de ROS Topics. Para demostrarlo, utilizamos los últimos dos nodos ejecutados en la sección anterior que son el nodo turtlebot3_gazebo y el nodo turtlebot3_teleop. Como se explicó, estos dos se comunican entre sí para hacer mover al robot waffle con las pulsaciones de la teclas en el mapa de Gazebo definido.

Usamos este ejemplo simple para dar una introducción a como fue nuestro proceso de programación de cada uno de los nodos y tópicos que requieren comunicarse entre sí al momento de correr los mismos para determinada ejecución que deseemos realizar en nuestro workspace. Utilizamos la herramienta rqt_graph para que nos muestre los nodos y los tópicos que se encuentran en ejecución. Al correr el comando $ rosrun rqt_graph rqt_graph en un nuevo terminal obtenemos una descripción gráfica de lo explicado.

Figura # 15. Herramienta rqt_graph

Obtuvimos muchas gráficas como estas al ejecutar el paquete del simulador Gazebo, el paquete de simulación y el paquete del robot modelo waffle. El análisis de estos paquetes nos permitió al principio entender el tipo de programación que se requiere para utilizar el robot waffle en los mapas de Gazebo disponibles para utilizar por código abierto.

En el proceso de aprendizaje sobre la programación en ROS, como herramienta para familiarizarnos con este entorno, se utilizó el paquete descargable de código abierto begginer_tutorials disponible en la página ROS.org. Con los tutoriales sobre la ejecución de este paquete se obtuvo un entorno de simulación simple con fondo de color único en el cual se nos presentaba como robot operable a una tortuga que se caracteriza por cambiar los colores de su aspecto despendiendo el tiempo de ejecución.

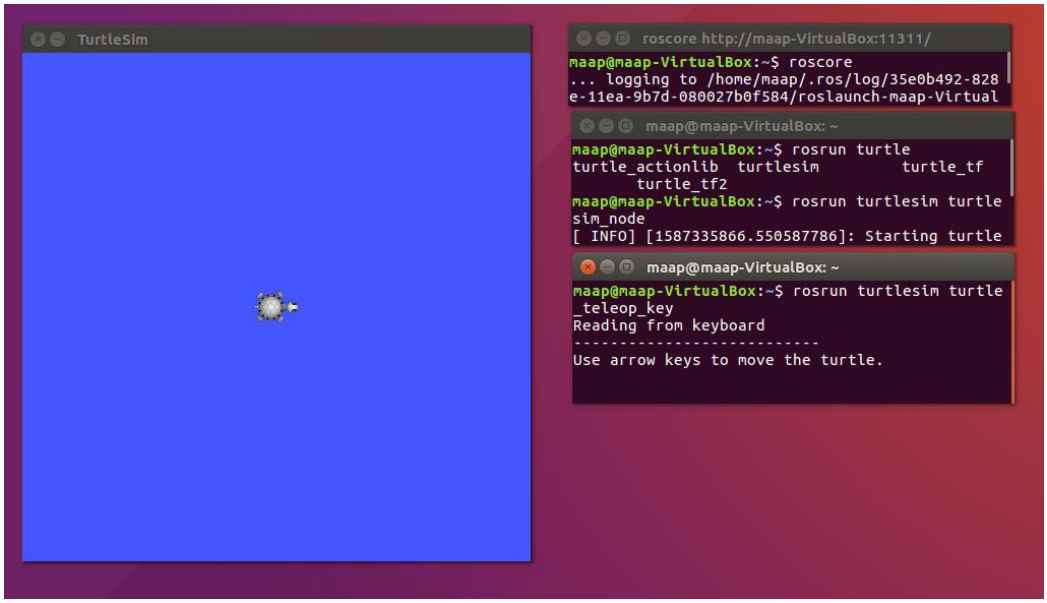

Figura # 16. Ejecución del paquete begginer_tutorials

En esta figura se muestra el resultado de uno de los tutoriales realizados para la comprensión del entorno ROS. En esta gráfica se muestra también el terminal en el cual se ejecuta el paquete donde se programó el movimiento de la tortuga con las fechas del teclado de nuestro computador. Como dato curiosos de diseño, el ya ocupar estas teclas para este tutorial nos motivó a definir las teclas W,A,D,X,S para mover nuestro robot waffle en el entorno de simulación. El procedimiento realizado en el workspace de este tutorial, lo reutilizamos y reproducimos en el nuestro para hacer que el robot waffle que hemos incorporado a nuestro proyecto se mueva en el entorno de simulación de Gazebo. Los créditos de la programación y resultados ejecutables que se muestra en la parte previa a esta sección, en parte corresponden a este tutorial mencionado ya que utilizamos una lógica y jerarquía de programación muy similar.

Figura # 17. Herramienta rqt_graph del paquete begginer_tutorials

Localización de los mapas disponibles en el entorno de simulación Gazebo.

Para continuar con lo que corresponde específicamente a los objetivos de este proyecto, presentamos a continuación todos los paquetes, nodos y tópicos que se crearon para hacer que nuestro robot TURTLEBOT3 modelo waffle se mueva de manera autónoma en los entornos de simulación. La siguiente gráfica nos muestra el nombre de nuestro Workspace que creamos para el proyecto.

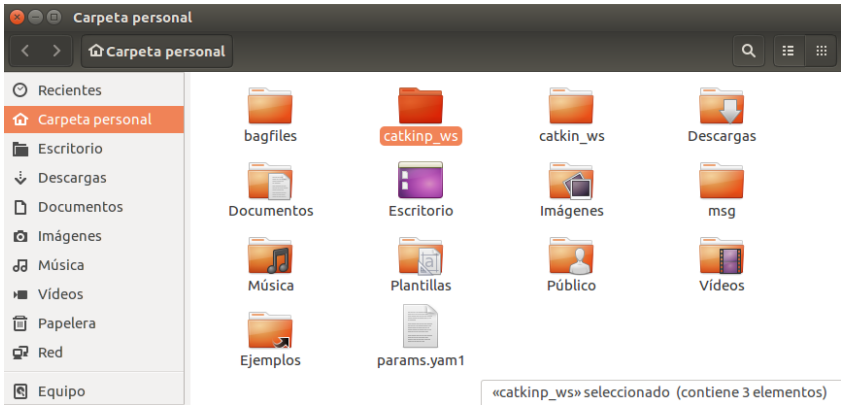

Figura # 18. Workspace catkinp_ws

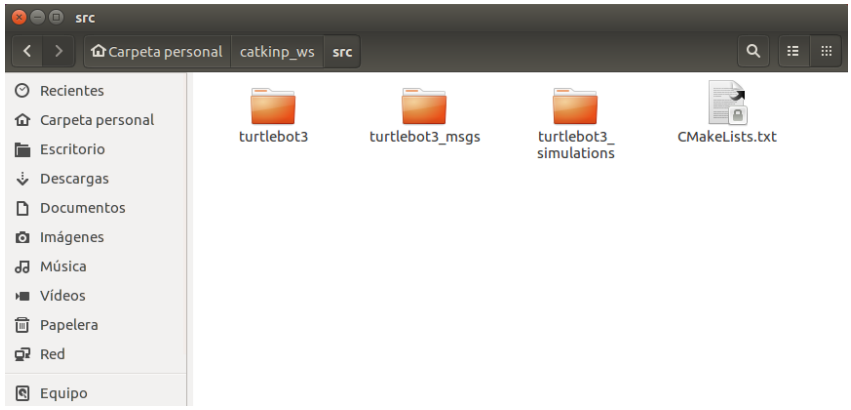

Figura # 19. Paquetes del workspace catkinp_ws

La gráfica de la figura nos muestra los tres paquetes creados y requeridos para ejecutar y hacer que el robot waffle se mueva de manera autónoma.

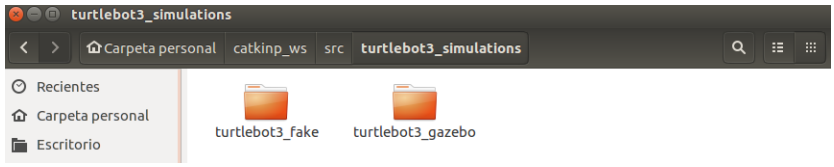

Figura # 20. Paquetes de simulación dentro del paquete turtlebot3_simulation

Dentro del paquete turtlebot3_simulation se crearon otros dos paquete que se muestran en la figura. El paquete tutlebot3_fake se refiere a la utilización del método nodo falso el cual se descargó pero ya no utilizamos, mientras que el paquete tutlebot3_gazebo contiene todos los mapas de simulación utilizados de Gazebo como se muestra en la siguiente figura.

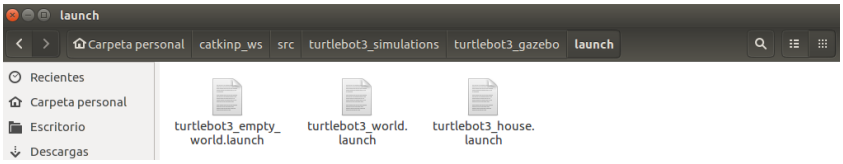

Figura # 21. Mapas de simulación disponibles del workspace

Programación, configuración e implemtación de laser LIDAR virtual al robot virtual waffle en entorno de simulación RViz y Gazebo.

Lo necesario para la conducción autónoma tanto virtual como real, como se ha plasmado con información en todo este documento, es que el cerebro principal de esta conducción sin colisiones sea un láser incorporado al robot. Este dispositivo es el que se encargó de identificar y recolectar datos de los objetos localizados alrededor del robot, para después esa información mediante una tarjeta de programación enviarla a los motores del robot y que los mismos ejecuten los movimientos correspondientes. En esta sección de entorno de simulación, para el reconocimiento de objetos se utilizó un láser virtual LIDAR. Este dispositivo se conecta y sincroniza con el mapa del entorno de simulación Gazebo mediante la creación de un topic denominado scan. Lo que sucede en este topic es que mientras el robot waffle se moviliza en el mapa de Gazebo, este laser va recopilando valores de mapeo y reconocimiento para después almacenarlos y ponerlos a disposición para ser usados por los motores del robot waffle.

Para apreciar el funcionamiento de este dispositivo LIDAR utilizamos el entorno de simulación RViz en el cual apreciamos series de puntos rojos producidos por el láser, los cuales son una representación simultanea de los objetos que se encuentran en el mapa de Gazebo. Los comandos que se describen a continuación al ejecutarlos dan fiabilidad de lo descrito. $ cd ~/catkinp_ws $ source devel/setup.bash $ roscd turtlebot3_gazebo $ export TURTLEBOT3_MODEL=waffle $ roslaunch turtlebot3_gazebo turtlebot3_gazebo_rviz.launch. Con la ejecución de estos comandos demostramos el el buen funcionamiento de la navegación completa del robot. Sin embargo, esta simulación

en RViz solo nos muestra parte del lugar por donde se moviliza el robot. Motivados por esta razón, hemos creado un nodo que nos permitió de igual manera en el software RViz guardar todos los valores obtenidos por el láser en el mapa de simulación Gazebo. Con esta recopilación y almacenamiento de datos, se nos fue posible generar el slam o mapeo de navegación virtual. Vale aclarar que este slam fue obtenida al movilizar al robot waffle por todo el mapa de simulación utilizando las teclas del computador ya descritas anteriormente. Para ejecutar el nodo de slam, es necesario ejecutar los siguiente comandos $ cd ~/catkinp_ws $ source devel/setup.bash $ roscd turtlebot3_gazebo $ export TURTLEBOT3_MODEL=waffle $ roslaunch turtlebot3_slam turtlebot3_slam.launch slam_methods:=gmapping Para guardar los datos de navegación es necesario en un terminal diferente ejecutar $ rosrun map_server map_saver -f ~/map.

Programación y configuración para la conducción autónoma del robot virtual waffle en el entorno de simulación Gazebo

En esta sección del documento explicamos lo que fue el proceso de programación para que nuestro robot virtual waffle se mueva por el entorno de simulación de un punto a otro esquivando los obstáculos. Esta programación se la realizó en el software de C++, es decir se creó un archivo .cpp para que el robot virtual realice esta operación. Para el desarrollo de este código programado se han recopilado líneas de código de diversos ejemplos ya desarrollados que se enfocaban indirectamente en objetivos similares a los nuestros. Estos códigos de programación se describen en los archivos adjuntos a este documento. En el workspace del proyecto estos códigos descritos se encuentran en las siguientes direcciones.

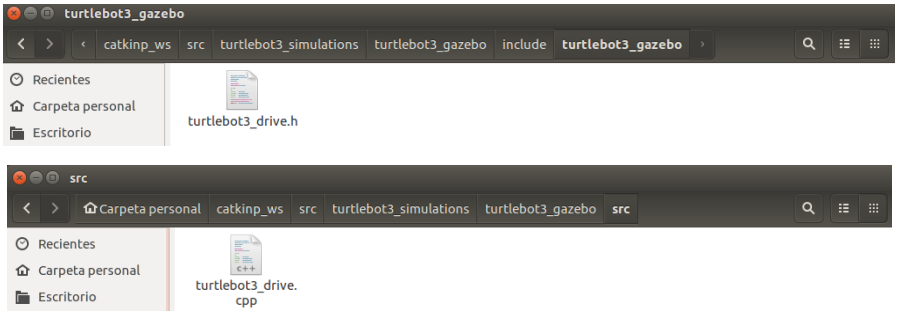

Figura # 22. Nodos de programación del workspace catkinp_ws

1.6. De vuelta a la ejecución del programa, en un nuevo terminal procedemos a probar el funcionamiento del paquete de simulación creado para que el programa funcione exitosamente en el entorno de simulación escogido ejecutando $ cd ~/catkinp_ws $ source devel/setup.bash $ roscd turtlebot3_gazebo $ export TURTLEBOT3_MODEL=waffle $ roslaunch turtlebot3_gazebo turtlebot3_house.launch. A la par en un segundo terminal para que el robot se mueva por el entorno de simulación de manera autónoma ejecutamos los comandos $ cd ~/catkinp_ws $ source devel/setup.bash $ roscd turtlebot3_gazebo $ export TURTLEBOT3_MODEL=waffle $ roslaunch turtlebot3_gazebo turtlebot3_simulation.launch. Despues de ejecutar estos dos terminales, se consiguió el objetivo del presente proyecto. Para verificación de procesos en un tercer terminal procedemos para obtener un diagrama de los topics y nodos que se encuentran en roscore a correr los siguientes comandos $ cd ~/catkinp_ws $ source devel/setup.bash $ roscd turtlebot3_gazebo $ rosrun rqt_graph rqt_graph.

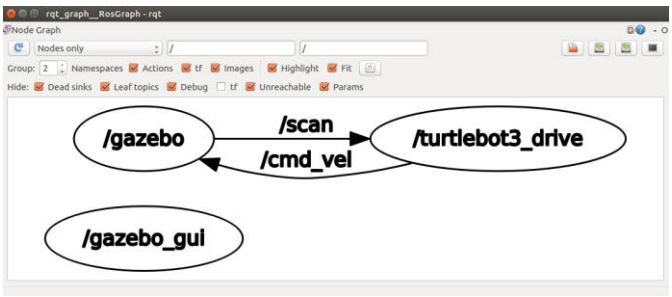

Figura # 23. Visualización de los paquetes del workspace catkinp_ws en rqt_graph

Entorno de montaje

Conducción autónoma en el entorno de montaje.

2. Instalación ROS y configuración del workspace.

El sistema operativo que se utilizó para instalar ROS fue Ubuntu 16.04 LTS. Esta versión de Ubuntu, soporta la distribución ROS KINETIC. Para descargarla se debe ingresar al siguiente enlace http://wiki.ros.org/kinetic/Installation/Ubuntu. Al entrar en el enlace, se procedió a abrir el terminal y se siguió el proceso de instalación del software. Este proceso contiene una serie de configuraciones para llevar a cabo la correcta instalación de la distribución de ROS especificada.

2.1. Como primer paso, en la interfaz de Ubuntu se procedió a ingresar "Configuración". Después se ingresó en la opción "Software y actualizaciones" y se seleccionó la opción Ubuntu Software. En esta opción se configuraron los repositorios de Ubuntu para permitir las opciones "restingido", "universo" y "multiverso".

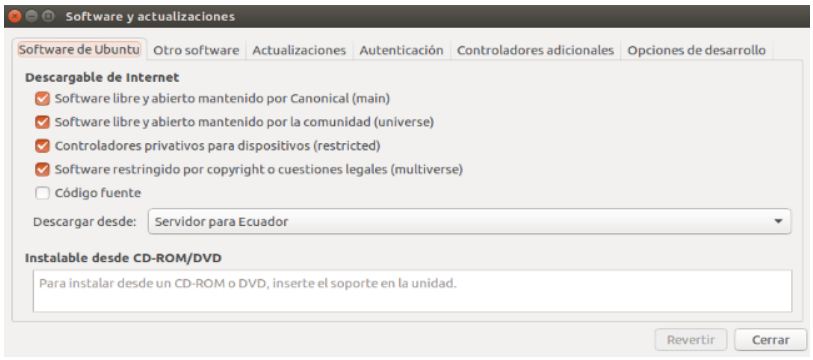

Figura # 24. Configuración de repositorio en Ubuntu

2.2. Al terminar de realizar esta configuración, se abrió un terminal en el sistema operativo Ubuntu y se procedió a configurar los archivos sources.list. En este caso se configuró al ordenador para aceptar los paquetes de tipo ros.org, el cual pertenece a los paquetes relacionados con la interfaz de ROS. Para hacer esto en el terminal se ejecutó sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main" > /etc/apt/sources.list.d/ros-latest.list'

2.3. Al terminar de configurar los archivos sources.list, se procedió a configurar las keys. Estas keys son listas de claves para autenticar paquetes. Es decir que los paquetes que sean autenticados en este caso, se consideraran de confianza en el sistema. Para ello se utilizó el comando sudo apt-key adv --keyserver 'hkp://keyserver.ubuntu.com:80' --recv-key C1CF6E31E6BADE8868B172B4F42ED6FBAB17C654. Con los keys configurados, se procedió a actualizar los programas y paquetes que existen en el sistema. Esto incluye a aquellos que están siendo instalados, para lograrlo se utilizó el comando sudo apt-get update.

2.4. Con toda la configuración previa realizada, se procedió a instalar una configuración para el entorno de ROS. La configuración que se utilizó para el desarrollo de ROS es "Desktop-Full Install". Esta configuración permite contar con varias herramientas de desarrollo como Rqt, Rviz, Robot-Generic Libraries etc. Se instaló esta configuración utilizando el comando sudo apt-get install ros-kinetic-desktop-full. En este punto, la distribución de ROS se ha instalado en nuestro sistema operativo. Para comprobar que la instalación fue un éxito, se verificó mediante el terminal. Utilizando los comandos $ cd /opt/ros/kinetic/ $ ls. Cuando estos comandos se ejecutaron, aparecieron los paquetes asociados a la distribución de ROS que ha sido descargada.

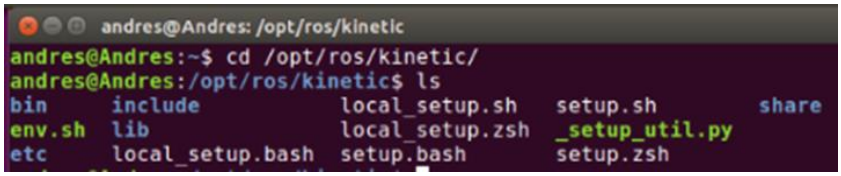

Figura # 25.Paquetes asociados a la distribución ROS-KINETIC

2.5. Teniendo el entorno de la distribución ROS-KINETIC descargada, se comenzó a configurar el espacio de trabajo (workspace). Como primer paso se editó el archivo bashrc. Este archivo entra en ejecución cada vez que se abre un nuevo terminal. Para editar este archivo se usó el editor de texto asociado a Linux denominado "gedit". En

el terminal se coloca $ gedit .bashrc. En este archivo se procedió a agregar las siguientes líneas de comando, las cuales nos permiten definir variables del entorno de trabajo y cargar las diferentes fuentes que apuntan a los ficheros asociados a la distribución instalada de ROS y al espacio de trabajo que se procederá a crear (catkin_ws).

```
source /opt/ros/kinetic/setup.bash
source /home/andres/catkin_ws/devel/setup.bash
export ROS_WORKSPACE=/home/andres/catkin_ws
export ROS_PACKAGE_PATH=$ROS_PACKAGE_PATH:$ROS_WORKSPACE
export ROSCONSOLE_FORMAT='[${severity}] [${time}]: ${message}'
```

Figura # 26. Definición de las fuentes, variables de entorno y enrutamiento de paquetes

2.6. Al realizar las modificaciones en el archivo .bashrc, se tuvieron que cargar las mismas. Para ello se procedió a usar el comando $ source .bashrc. Este comando se ejecutó en un nuevo terminal y permitió cargar los cambios realizados en el archivo .bahsrc y actualiza el mismo. Al realizar estos cambios, se tiene que verificar la correcta definición de las fuentes, variables de entorno y enrutamiento de paquetes. Para llevar a cabo este procedimiento, se ejecutó el comando $ export. Cuando se ejecutó este comando, se exportan las variables definidas en el sistema operativo. Después, se buscaron las variables que hemos definido en el archivo .bashrc. Si las variables se encuentran en este listado, significa que están bien definidas como se muestra a continuación.

```
declare -x ROS_DISTRO="kinetic"
declare -x ROS_ETC_DIR="/opt/ros/kinetic/etc/ros"
declare -x ROS_MASTER_URI="http://localhost:11311"
declare -x ROS_PACKAGE_PATH="/home/andres/catkin_ws/src:/opt/ros/kinetic/share:/
home/andres/catkin_ws"
declare -x ROS_PYTHON_VERSION="2"
declare -x ROS_ROOT="/opt/ros/kinetic/share/ros"
declare -x ROS_VERSION="1"
declare -x ROS_WORKSPACE="/home/andres/catkin_ws"
```

Figura # 27. Definición de variables y enrutamiento de paquetes

2.7. Después de haber realizado estas configuraciones, se procedió a crear un workspace. Se limpió el terminal con el comando clear y se creó una carpeta denominada

catkin_ws. Esta carpeta es la raíz principal de nuestro espacio de trabajo. Para crear esta carpeta, se definió el directorio en el que se pretende crear el Workspace y se procedió a ejecutar el comando $ mkdir catkin_ws. Después de crear este directorio, se procede a crear un nuevo directorio dentro del mismo. Este nuevo directorio se denominó src (source) utilizando los comandos $ cd catkin_ws $ mkdir src. Dentro del directorio src, se procedió a inicializar el espacio de trabajo. Para ello se utilizó el comando $ catkin_init_workspace. Este comando crea dentro del directorio src un archivo denominado CMakeLists.txt. Este archivo contiene instrucciones que describen los archivos de origen, el espacio de trabajo del paquete y otras directrices.

2.8. Por último, para terminar de construir nuestro espacio de trabajo, se compiló las configuraciones realizadas en nuestro workspace utilizando el comando $ catkin_make Este comando permite compilar los paquetes y las actualizaciones que se han realizado dentro del espacio de trabajo. En este punto se ha instalado de una forma correcta la distribución de ROS-KINETIC. Adicionalmente se configuró el espacio de trabajo donde tendrán cabida los diferentes procesos que se ejecutaran en el entorno ROS. Nuestro espacio de trabajo tiene la siguiente forma.

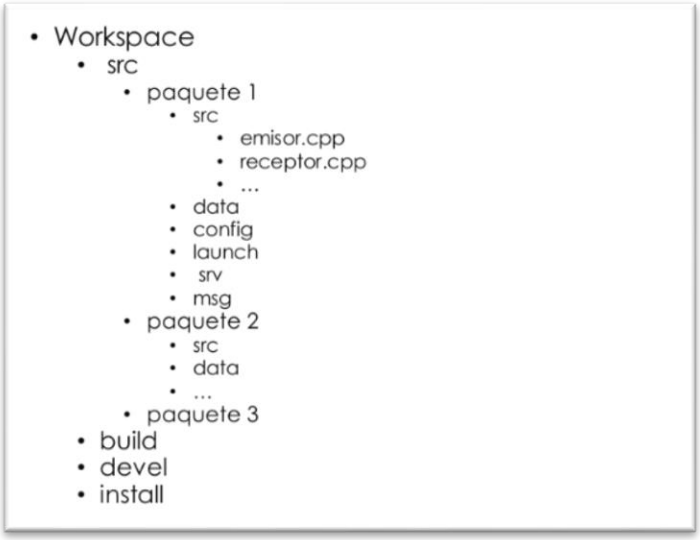

Figura # 28. Workspace catkin_ws

Podemos ver que la estructura de carpetas constas de varios directorios. Primeramente se encuentra uno de los directorios más importantes denominado src (source space). Este directorio contiene los códigos fuente que se encuentran en los paquetes catkin. Cada paquete dentro de este puede contener incluso otros directorios src. En los paquetes contenidos por este directorio se pueden definir una serie de procesos como la configuración de procesos, servicios, mensajes, datos, definición de variables etc.

Otra directorio que conforma el Workspace es ña carpeta build (build space). Este directorio hace alusión a CMake. Esto permitirá construir los paquetes catkin en el src. Cmake y catkin mantienen en este directorio su caché y archivos de configuración.

En el directorio Devel (development), se construirán los targets construidos antes de su intalacion. De esta manera se brindará un entorno útil para prueba y se procesaran los desarrollos que no necesitan solicitar el paso de instalación.

Install pro su parte, es un directorio que permite instalar un objetivo de instalación después de construir los objetivos especificados.

Instalación del paquete rplidar.

Después de crear nuestro espacio de trabajo, se procedió a configurar un sistema que permita observar las características del sensor RPLIDAR A2M8. Para ello, se implementó un desarrollo de entorno para ROS. Se procedió a ingresar en el siguiente enlace que contiene la información del paquete rplidar en la página oficial de ROS http://wiki.ros.org/rplidar.

2.9. Se abrió un terminal y se accedió al directorio src del workspace; el cual está destinado para la creación de paquetes, utilizando el comando $ cd /catkin_ws/src. Dentro del directorio src, se procedió a descargar el paquete rplidar. En la página oficial de ROS,

muestra un enlace donde podemos descargar el paquete en GitHub. Para ello usamos el comando $ git clone https://github.com/Slamtec/rplidar_ros.git

2.10. Después de haber descargado el paquete rplidar_ros, se procedío a compilar el mismo para que se encuentre dentro del entorno de ROS. Para ello nos ubicamos en el directorio catkin_ws y compilamos los paquetes utilizando el comando $ catkin_make. Antes de comprobar el funcionamiento del paquete y por tanto del sensor RPLIDAR A2M8, procedimos en el terminal a escribir el siguiente comando $ sudo chmod 666 /dev/ttyUSB0. Este comando le da un permiso para utilizar el puerto USB como conexión de dispositivo, en este caso el sensor. Es por ello que el LIDAR mediante la comunicación USB con el ordenador, tiene que encontrarse conectado al mismo para que se lo pueda reconocer.

2.11. Finalmente con el permiso verificado, ejecutamos el nodo que viene por defecto en el paquete rplidar_ros. Para ello, debemos ejecutar el workspace ROS con el comando $ roscore. Este comando activó los nodos que se encuentran dentro de los paquetes que forman parte de workspace. En este punto solo tenemos el paquete rplidar instalado por lo cual será el único paquete que será activado con el respectivo nodo que lo conforma. Para ejecutar el nodo denominado rplidar_node, el cual se encuentra en el paquete rplidar, se debió abrir otro terminal en el cual procedimos a ubicarnos en el directorio catkin_ws y correr el comando $ rosrun rplidar_ros rplidarNode. Este nodo que viene por defecto, se encarga de inicializar todas las variables y los procesos que construyen el sensor RPLIDAR A2M8. Este nodo es el encargado de construir las mediciones de ángulos y distancias. Se encarga de confirmar la conexión del sensor con el sistema operativo, muestra las diferentes características que trae consigo el sensor etc. Para esto utiliza una serie de variables inicializadas en diferentes topics que definen el paquete rplidar.

Creación, compilación y ejecución de nodos

Teniendo en cuenta que el paquete rplidar provee un nodo madre (rplidar:node), el cuál define las variables, procesos y mediciones del sensor RPLIDAR A2M8, se pretendió estructurar un sistema que permita controlar el prototipo del vehículo autónomo. Para esto fue necesario tener en cuenta varios procesos de por medio. Para entender de una mejor manera el proceso de montaje, debemos enteder dos conceptos importante que se encuentran en el entorno ROS. Primero están los mensajes son ficheros que contienen el nombre (topic) y tipos de variables (enteros, flotantes, vectores, etc). Por otro lado, están los mecanismos de comunicación en ROS los cuales nos permitieron transferir los mensajes de un nodo a otro. Los principales mecanismos de comunicación en ROS son paso de mensajes, servicios, actionLibs y servidor de parámetros.

Para nuestro proyecto se implementó el mecanismo de paso de mensajes o también denominado emisor-subscriptor. Este mecanismo consiste en un nodo publicador, el cual emite un topic de un determinado tipo. Por otro lado se encuentra un nodo subscriptor, el cual como su nombre lo indica, esta subscrito al nodo publicador mediante el topic que este emite. En realidad puede haber n números de nodos suscritos a un nodo publicador. La siguiente imagen conceptualiza este mecanismo de comunicación.

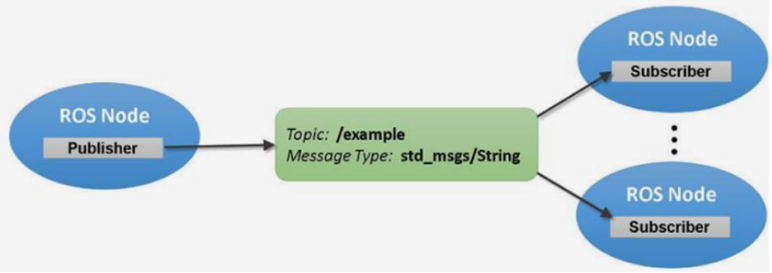

Figura # 29. Mecanismo paso de mensajes

Como se recalcó anteriormente, una de las características importantes que se destaca de ROS, es que los nodos (tanto suscriptor como publicador), pueden encontrarse estructurados en cualquier diferentes lenguajes de programación (C++, Python, Java, etc).

Cumpliendo así una de las características principales de ROS como es el soporte a varios lenguajes de programación.

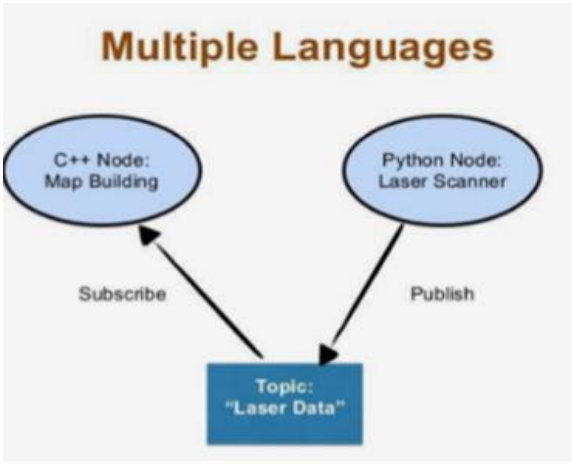

Figura # 30. Publicación y subscripción en ROS

Con los conceptos que se han mostrado brevemente, se procedió a la construcción de los nodos que nos permitieron brindar funcionalidades específicas utilizando el sensor RPLIDAR A2M8. Para ello, teniendo en cuenta que solo existe un paquete y dentro del mismo un nodo en ejecución, podemos observar los topics que este nodo emite y recibe utilizando el comando $ rostopic list. Como se puede ver en la imagen, este comando permite observar que los diferentes topics que existen para la configuración del nodo rplidar_node que viene por defecto. Se puede observar entonces los topics en ejecución.

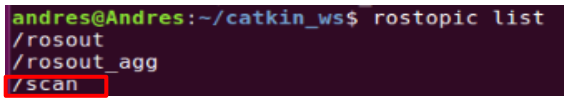

Figura # 31. Lista de nodos en ejecución

Como se puede observar en la imagen anterior existen tres topics interactuando con ese nodo. Para observar que variables están definidas en los mismo, utilizamos el comando rostopic echo seguido del topic que se desea analizar. Se conoce que las variables definidas para el topic/scan, definen las diferentes variables de medición (output data) que se encuentran definidas para el sensor RPLIDAR A2M8. Por lo tanto para observar las

variables que se encuentran en este topic, ejecutamos el comando $ rostopic echo scan. Observamos las variables que están definidas en este topic. Además, observamos que se encuentran definidas variables como rango mínimo , rango máximo, tiempos, distancias, ángulos, entre otras. Este topic es el más importante del sistema, puesto que este topic será emitido por el nodo madre; el cual será un nodo publicador, hacia nodos que se subscriban a este nodo publicador mediante el topic/scan. En la siguiente imagen se puede observar las diferentes variables contenidos en el topic scan.

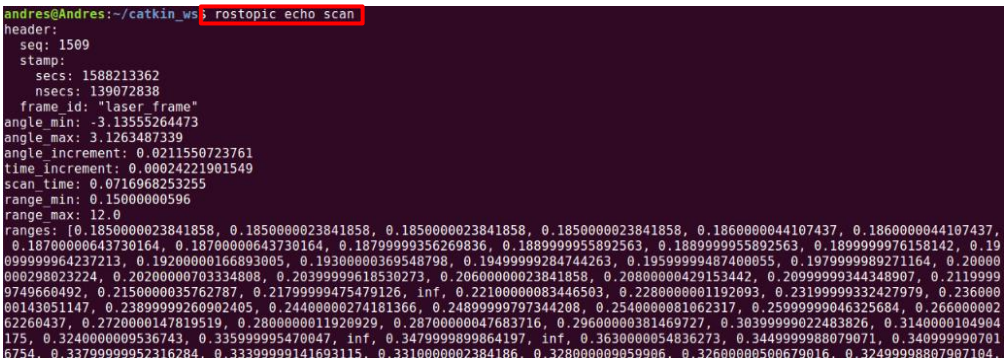

Figura # 32. Varibles en ejecución

Teniendo en cuenta que se conoce el nodo madre y su estructura, los topics que se encuentran definidos en el sistema y los mecanismos de comunicación que se pretende aplicar, se procede a la construcción de nodos que realizaran funciones especificas. El primer nodo se denominó /rplidar_node_client. Este nodo está desarrollado en C++ que usa el método paso de mensajes. Para estar subscrito al nodo “rplidar_node” mediante el topic /scan. Básicamente este nodo nos permitió observar en tiempo real las mediciones de ángulo-distancia que mide el sensor RPLIDAR A2M8. Después de crear el archivo .cpp , se procedió a compilar el mismo usando el comando $ catkin_make.

Con las mediciones observadas gracias al nodo /rplidar_node_client, podemos ver que el sensor RPLIDAR A2M8 está determinando de una manera correcta la medición de los puntos en tiempo real, por lo cual se puede pasar a la siguiente etapa que consiste un nodo que permita manejar los movimientos del prototipo de auto autónomo. Por consiguiente

pasamos a la creación del siguiente nodo. Este nodo se denominó /car_control y fue desarrollado en python el cual usa el método paso de mensajes. Para estar subscrito al nodo rplidar_node mediante el topic /scan. Este nodo procesa en tiempo real los datos de ángulos y distancias medidas por el sensor LIDAR y determina el manejo de los motores. El algoritmo consiste en el avance del carro en dirección norte, si se encuentra algún objeto en un rango [350°-10°], se detiene y compara inmediatamente los rangos [80°-100°] (Derecha) y [260°-290°] (Izquierda) y toma la decisión de seguir en la dirección que tenga mayor distancia de detección para circular.

Resultados entorno de montaje.

Nodo madre - /rplidar_node.

Como se mencionó anteriormente, este nodo define el sistema del RPLIDAR A2M8 en el entorno ROS. Para ejecutar el nodo, en otro terminal se ubicó el directorio catkin_ws y se ejecutó el comando $ rosrun rplidar_ros rplidarNode. Este nodo que viene por defecto, se encarga de inicializar todas las variables y los procesos que construyen el sensor RPLIDAR A2M8. Cuando se ejecuta este nodo podemos observar algunos datos del sensor como el firmware, status, distancia máxima, compensación de ángulo, etc.

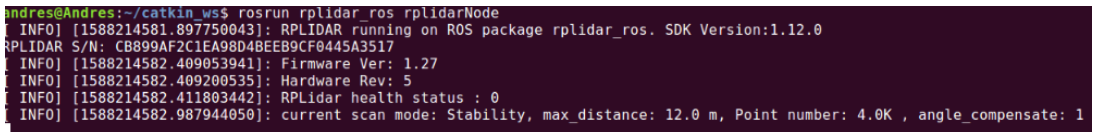

Figura # 33. Puesta en ejecución del nodo rplidar_node

Nodo publicador de datos en tiempo real- /rplidar_node_client

Este nodo publica los datos del sensor RPLIDAR A2M8 en tiempo real. Es un nodo construido en C++ que imprime en pantalla los diferentes valores de distancia y ángulos medidos por el sensor. Cabe recalcar que es un nodo subscriptor el cual está subscrito

mediante el topic/scan a el nodo publicador rplidar_node_client. Ejecutando $ rosrun rplidar_ros rplidarNodeClient obtenemos la siguiente gráfica.

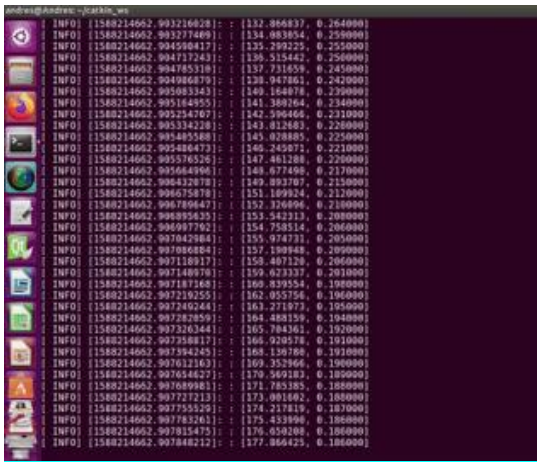

Figura # 34. Muestreo en tiempo real tras la ejecución del nodo rplidar_node_client

Nodo para el manejo autónomo - /car_control

Este nodo es el encargado del manejo de los motores y es el segundo nodo subscriptor al topic/scan por lo cual lee las variables que están definidas en el mismo, en este caso distancia y ángulos, para procesarlos y el determinar el movimiento de los motores. Esto se logra ejecutando $ rosrun rplidar_ros car_contol. Recalcamos que este nodo funcionó únicamente si el nodo madre se esta ejecutando, puesto que es el nodo que publica los valores de distancia y ángulos contenidos dentro del topic scan.

Figura # 35. Conducción autónoma del prototipo

Launchers

Se define al launcher como fichero que especifican como poner en ejecución los nodos de un determinado paquete. Con este sistema estructurado y con los datos obtenidos en tiempo real, se ejecutaron dos launchers.

El primer launcher, consiste en ejecutar los puntos en tiempo real ángulo-distancia en un plano 2D. Para ello se utilizó rviz, que es una interfaz gráfica asociada a ROS. Se crea un fichero asociando los resultados a la interfaz rviz y se obtiene el siguiente resultado.

```
<launch>
  <node name="rplidarNode"          pkg="rplidar_ros"  type="rplidarNode" output="screen">
  <param name="serial_port"         type="string" value="/dev/ttyUSB0"/>
  <param name="serial_baudrate"     type="int"    value="115200"/><!--A1/A2 -->
  <!--param name="serial_baudrate"     type="int"    value="256000"--><!--A3 -->
  <param name="frame_id"            type="string" value="laser"/>
  <param name="inverted"            type="bool"   value="false"/>
  <param name="angle_compensate"    type="bool"   value="true"/>
  </node>
</launch>
```

Figura # 36. Variables usadas en rplidar.laucnh

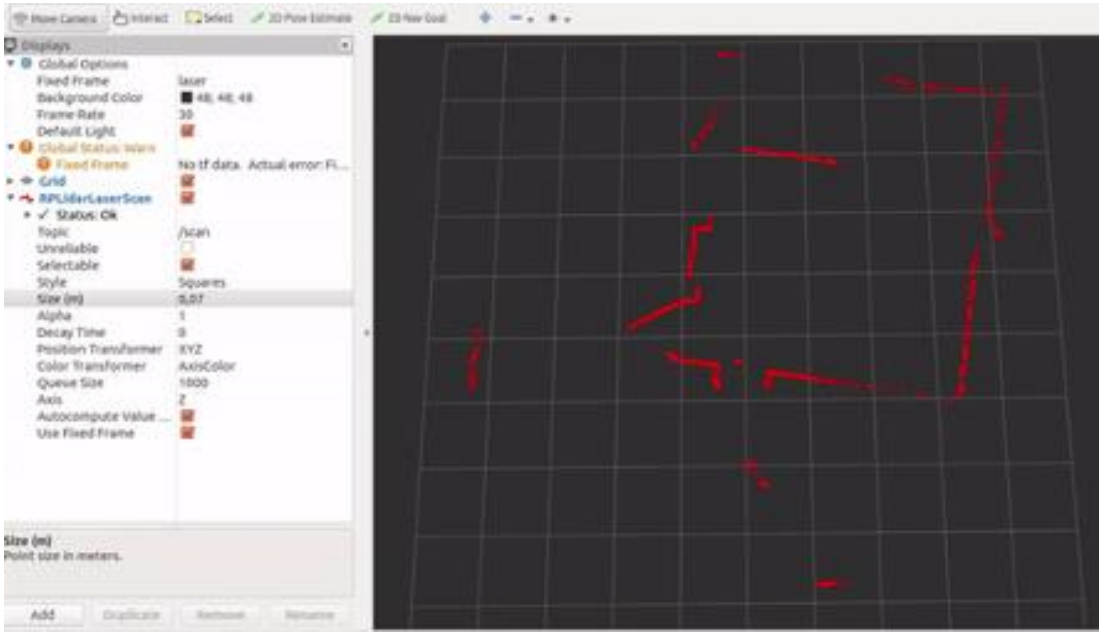

Figura # 37. Grafica del entorno en 2D utilizando la interfaz gráfica Rviz

El segundo launcher se denomina SLAM. Este launcher nos permite ver la ubicación en tiempo real y reconstruir en un mapa 2D el entorno por el cual circula nuestro carro. Para ello, necesitamos descargar un paquete denominado hector_:slam. Se procede a ingresar en el directorio catkin_ws y se ejecuta el comando $ git clone https://github.com/tu-darmstadt-

ros-pkg/hector_slam.git$ git clone https://github.com/Slamtec/rplidar_ros.git. Al descargar este paquete, podemos observar que tenemos varios directorios nuevos. Primeramente, configuramos un launcher denominado hector_mapping. En este launcher cambiaron las siguientes variables.

```
<?xml version="1.0"?>

<launch>
  <arg name="tf_map_scanmatch_transform_frame_name" default="scanmatcher_frame"/>
  <arg name="base_frame" default="base_link"/>
  <arg name="odom_frame" default="base_link"/>
  <arg name="pub_map_odom_transform" default="true"/>
  <arg name="scan_subscriber_queue_size" default="5"/>
  <arg name="scan_topic" default="scan"/>
  <arg name="map_size" default="2048"/>
```

Figura # 38. Variables de configuración del launcher hector_mapping

Despues de guardar todos los cambios relaizados, para compilar los nuevos nodos y launcher creados, ejecutamos el comando $ catkin_make. Finalmente se procede a ejecutar el launch ejecutando en el terminal $ roslaunch hector_slam tutorial.launch. Los resultados obtenidos son los siguientes.

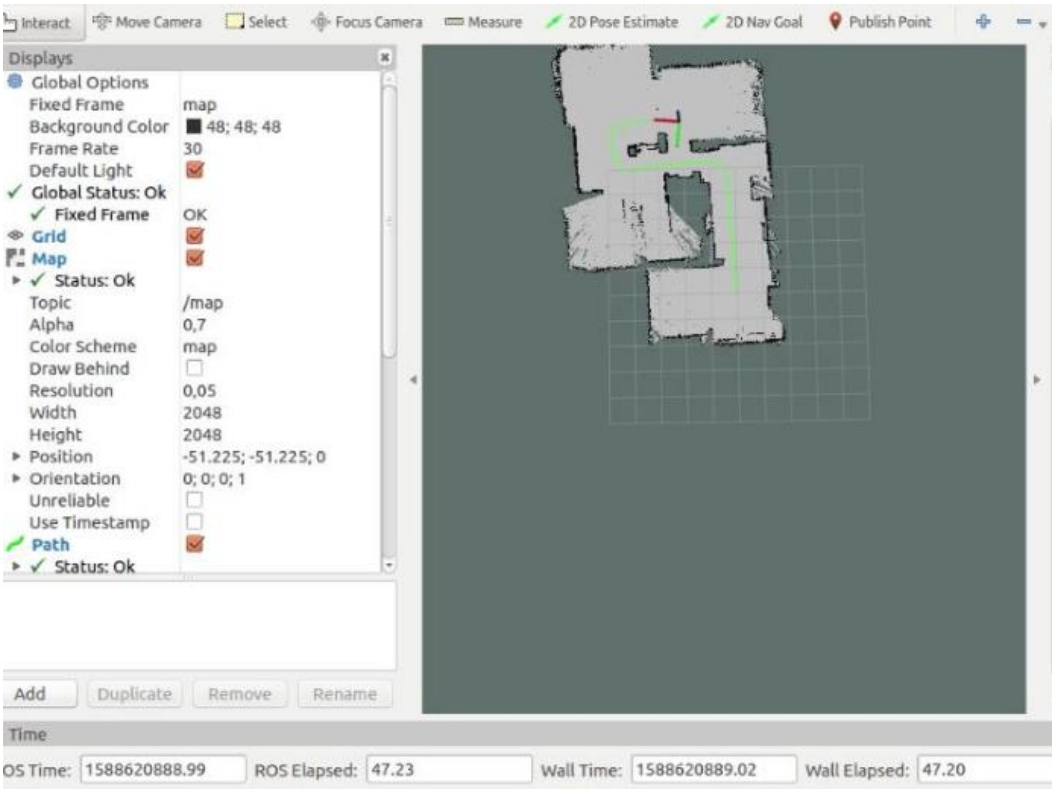

Figura # 39. Slam de navegación utilizando la interfaz gráfica Rviz

Proyectos Futuros

Este proyecto es altamente escalable y ya que futuras promociones trabajarán en él hemos realizado cada avance pensando en este aspecto, razón por lo cual ponemos a disposición:

- Esquemas del prototipo fáciles de dar seguimiento.
- Códigos de programación implementado para simulación como para el prototipo real.
- Guía o tutorial para la reproducción del entorno de simulación.

Y con estos materiales conseguir el vehículo autónomo completamente real y funcional usando ROS.

El reto para siguientes promociones es la incorporación de dispositivos como cámaras y otros sensores que garanticen un mejorado sistema dinámico de reconocimiento y mucho mas funcional para identificar en tiempo real objetos extraños o elementos no esperados en el trayecto predefinido.

CONCLUSIONES

Dada la crisis sanitaria ocasionada por la pandemiade COVID-19, fue necesario un cambio drástico de los primeros objetivos y un rediseño del entorno de trabajo. Por esta razón, el enfoque que hemos presentado en este documento se centró principalmente en poder solucionar los problemas que se nos presentaron con la poca disponibilidad de hardware accesible, vale mencionar como ejemplo el cambio que realizamos al reemplazar un Raspberry Pi 3B+ por un Raspberry Pi 4b para la implementación y funcionamiento final del proyecto.

Inicialmente nos habiamos planteado presentar un vehículo autónomo en una pista de obstáculos, sin embargo por lo sucedido a nivel nacional con el aislamiento, la presentación de los resultados en entorno de simulación tomaron un peso mucho mayor dentro del desarrollo del proyecto.

Todos los resultados que se han descrito en el presente documento estan disponibles como videos adjuntos tanto del entorno de simulación como el entorno de montaje. En estos videos y otros archivos adjuntos resultantes del proyecto se aprecia cada uno de los procesos que se han descrito en todo este documento.

Vale recalcar que la evidencia del éxito del proyecto en ejecución recae en:

- Esquemas del prototipo fáciles de dar seguimiento.
- Códigos empleados mencionando sus fortalezas y debilidades
- Tutorial de Entorno de simulación.
- Videos explicativos de los avances más significativos explicando lo que se realiza.

De esta manera, posibilitamos a futuras generaciones seguir avanzando e innovar en el desarrollo del vehículo autónomo.

REFERENCIAS BIBLIOGRÁFICAS

Matus, D. (2017). La historia de los carros autónomos contada en unos pocos hitos. 2020, marzo 3, de DIGITAL TRENDS ES Recuperado de https://es.digitaltrends.com/autos/historia-carros-autonomos/

Parra, S.(2014). Generación de mapa de entorno para navegación de vehículo terrestre autónomo. 2020, marzo 19, Recuperado de http://repositorio.uchile.cl/handle/2250/131903

Wurzburg, H, Rempel, S, Kancir, P, & Mackay, R.(2019). ROS and Hector SLAM for Non-GPS Navigation. 2020, marzo 26, de ARDUPILOT Recuperado de https://ardupilot.org/dev/docs/ros-slam.html

Son, W. (2020). TurtleBot3. 2020, abril 03, de ROBOTS e-Manual Recuperado de http://emanual.robotis.com/docs/en/platform/turtlebot3/pc_setup/

RPLIDAR A2. (2017, 15 de mayo). 2020, abril 15, Recuperado de https://www.slamtec.com/

Pérez, A.(2017, 09 de octubre). Desarrollo de sistema de navegación para vehículos autónomos terrestres utilizando ROS [Trabajo fin de máster]. Universidad Politécnica de Cartagena, Cartagena.

Gutiérrez, J.(s.f). Implementación de vehículo autónomo en entorno simulado [Trabajo fin de grado]. Universidad Carlos III de Madrid, España.

Vázquez, J.(2013, 29 de enero). Desarrollo de un vehículo controlado mediante GPS [Trabajo de fin de carrera]. Universidad Politécnica de Cataluña, Cataluña.

Parra, S.(2014). Generación de mapa de entorno para navegación de vehículo terrestre autónomo [Tesis de grado]. Universidad de Chile, Chile.

Correa, P, Barranco, A, Pérez, F, & Bautista, P.(2018). Puesta en funcionamiento del sensor RPLIDAR A2 M8 usando Linux y Python [Trabajo de fin de carrera]. Universidad de Guanajuato, México.