Proceso de simulación e implementación física para el robot diferencial Turtlebot 2

Correa, Jose Miguel. Pinto, Ana Maria y Saavedra, Miguel Ángel
{jose.correa, ana.pinto, miguel.saavedra}@uao.edu.co
Universidad Autónoma de Occidente

***Resumen*—En el siguiente informe se explica el proceso de simulación en Rviz y Gazebo para el turtlebot 2 usando ROS y la configuración de la tarjeta Jetson TK1 para su funcionamiento en el modelo real. Inicialmente, se diseñó el modelo del robot el Solidworks y se realizó la simulación del robot por medio de ROS, Rviz y Gazebo. Adicionalmente, se implementaron, en la simulación, los métodos de SLAM y AMCL para mapear y planear trayectorias en un ambiente determinado. Finalmente, se explica el proceso de configuración de la tarjeta TK1, los problemas que se nos presentaron durante este proceso y el procedimiento a realizar para aplicar el proceso de mapeo y localización en el robot.**

***Palabras clave*—turtlebot 2, Jetson TK1, mapear, planear trayectorias.**

I. INTRODUCCIÓN

El turtlebot 2 es un robot móvil diferencial usado para áreas de investigación y educación que puede ser programado por medio de ROS. Al ser un robot diferencial implica que tiene dos llantas motorizadas independientes a las que se les controla su velocidad para permitir que se mueva en línea recta o caminos curvos. Adicionalmente, el turtlebot cuenta con dos ruedas adicionales que sirven únicamente de apoyo para la estructura.

Previo a la programación del robot diferencial no será necesario realizar algún tipo de cálculo para su funcionamiento, sin embargo es importante resaltar que los datos como el radio de las llantas y la distancia desde su centro hasta el centro de la estructura del robot deben conocerse para añadirlos, como parámetros, a los archivos que describen su funcionamiento y cinemática. Además se debe tener en cuenta que este tipo de robot móvil tiene una restricción de movimiento para cada llana motorizada y una única restricción de deslizamiento.

II. MODELO 3D DEL ROBOT

El Turtlebot es un robot diferencial, lo que implica que tiene dos ruedas motorizadas que permiten modificar su velocidad de manera independiente, para seguir caminos rectilíneos, curvos o circulares. El robot está conformado por una base (kobuki), tres bandejas para posicionar diferentes sensores u objetos, dos ruedas motorizadas o de tracción y dos ruedas adicionales, tambien conocidas como ruedas locas, que sirven únicamente de apoyo para el robot.

Para el diseño del modelo 3D se usó el software Solidworks (ver Fig. 1). Cada una de las partes del robot hace referencia a un link diferente, siendo en total ocho links -incluyendo sensor láser, cámara y un pequeño bloque en ubicado en el centro de lavase del robot- y siete joints. En el anexo 1 se muestra la estructura del robot, en donde se puede apreciar la relación entre links y joints.

A cada uno de los eslabones (links) se les asignó su propio marco de referencia en Solidworks, además con ayuda del software fue posible obtener datos físicos como masas, colisiones e inercias de cada parte del robot.

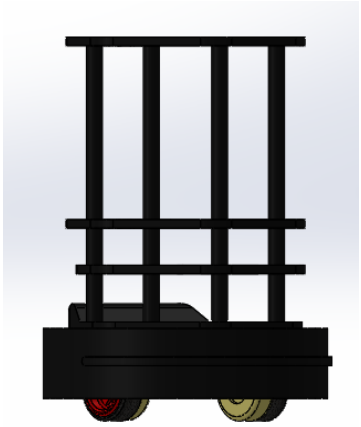

Fig. 1. Modelo del robot el Solidworks

III. SIMULACIÓN USANDO ROS

Antes de iniciar el proceso de simulación con ROS fue necesario exportar cada uno de los eslabones o partes del robot a archivos con formato *.stl*, que permite crear una maya del objeto 3D sin tener en cuenta características físicas como color y texturas.

Se creó un archivo URDF con la descripción del robot y cada una de sus partes teniendo en cuenta información como propiedades físicas, marco de referencia, tipo de parte ("link" o "joint"), tipo de junta o "joint" y relación entre eslabones (padre o hijo). En este archivo también se definen parámetros como el uso de láser y cámara para reconocer el ambiente, transmisión de cada llanta y los controladores PID para la velocidad de las ruedas motorizadas. Además, se realiza la descripción del bloque en el centro de la base, que permite crear un frame global asociado a la estructura del robot. En las Fig. 2 y 3 se muestran fragmentos del código URDF usado, inicialmente, para la simulación.

```
<!-- Joint between the block and chasis-->
<joint name="bloque_to_armazon" type="fixed">
  <parent link="bloque"/>
  <child link="armazon_link"/>
  <origin xyz="0 0 0"/>
  <axis xyz="0 0 0" />
```

Fig. 2. Joint del chasis del robot definida en el archive URDF.

```
<!-- Chasis of the robot -->
<link name="armazon_link">
  <visual>
    <origin rpy="0 0 0" xyz="0 0 0"/>
    <geometry>
      <mesh filename="package://proyecto/
meshes/armazon.STL"/>
    </geometry>
    <material name="black_metal">
      <color rgba="0.1 0.1 0.1 1"/>
    </material>
  </visual>

  <inertial>
    <origin
      xyz="-0.00020681 -5.8482E-09 0.04814"
      rpy="0 0 0" />
    <mass
      value="9.9908" />
    <inertia
      ixx="0.18286"
      ixy="-8.1444E-09"
      ixz="-0.00095615"
      iyy="0.18195"
      iyz="-4.5751E-09"
      izz="0.14187" />
  </inertial>

  <collision>
    <origin
      xyz="0 0 0"
      rpy="0 0 0" />
    <geometry>
      <mesh
        filename="package://proyecto/meshes/
armazon.STL" />
    </geometry>
  </collision>

</link>
```

Fig. 3. Construcción de un link usando URDF.

Para simular en Gazebo es necesario referir cada parte del robot al nodo de Gazebo e inicializar propiedades como el uso de la gravedad (ver Fig. 4). Las ruedas motorizadas necesitan información adicional en su descripción como restricción de velocidad, fricción, máximo esfuerzo y controlador PID. Adicionalmente se usó el nodo teleop_node que permite controlar el modelo del robot mediante el teclado.

```
<!-- Gazebo reference of the chasis -->
<gazebo reference="armazon_link">
  <material>Gazebo/Grey</material>
  <turnGravityOff>false</turnGravityOff>
</gazebo>
```

Fig. 4. Inicialización de un link para Gazebo.

```
<!-- Gazebo reference of the left wheel -->
<gazebo reference="left_wheel">
  <mu1 value="1.0"/>
  <mu2 value="1.0"/>
  <kp value="10000000.0"/>
  <kd value="1.0"/>
  <fdir1 value="1 0 0"/>
  <material>Gazebo/Black</material>
  <turnGravityOff>false</turnGravityOff>
</gazebo>
```

Fig. 5. Controlador PID para una de las llantas.

Posteriormente, se creó un archivo *.xacro* con el fin de optimizar el código generado en el urdf. Los archivos tipo xacro contienen la misma información de un urdf,

sin embargo son modulares y permiten la reutilización de código al describir un modelo.

Para ejecutar el modelo del robot se creó un archivo *.launch* que permite correr todas las dependencias y nodos necesarios para la visualización en RViz, Gazebo y su control. El archivo *proyecto.launch* contiene los nodos robot_description, empty_world y robot_state_publisher, que permiten, respectivamente, generar a visualización del robot en Rviz, crear un ambiente vacío en Gazebo y mostrar todas las partes del robot. Para ejecutar el código se usa el comando $*roslaunch proyecto proyecto.launch.*

```
        <launch>
        <!-- Including Empty world files from
gazebo -->
        <include file="$(find gazebo_ros)/
launch/empty_world.launch" />
        <arg name="model" />
        <!-- Parsing xacro and setting
robot_description parameter -->
        <param name="robot_description"
textfile="$(find proyecto)/urdf/proyecto.urdf" /
>
        <!-- Setting gui parameter to true for
display joint slider -->
        <param name="use_gui" value="true"/>
        <!-- Starting Joint state publisher
node which will publish the joint values -->
        <node name="joint_state_publisher"
pkg="joint_state_publisher"
type="joint_state_publisher" />
        <!-- Starting robot state publish which
will publish tf -->
        <node name="robot_state_publisher"
pkg="robot_state_publisher"
type="robot_state_publisher"/>
        <!-- Launch visualization in Gazebo -->
        <node name="spawn_model"
pkg="gazebo_ros" type="spawn_model" args="-file
$(find proyecto)/urdf/proyecto.urdf -urdf -
model proyecto" output="screen" />
    <param name="publish_frequency"
type="double" value="50.0" />
        <!-- Launch visualization in rviz -->
        <node name="rviz" pkg="rviz"
type="rviz" args="-d $(find proyecto)/
urdf.rviz" required="true" />
</launch>
```

Fig 6. Código de *proyecto.launch*.

En las Fig. 7 y 8 se muestran los modelos del robot en Rviz y Gazebo.

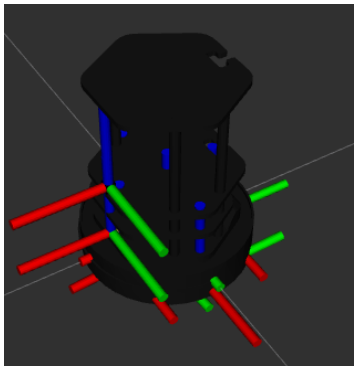

Fig. 7. Modelo en Rviz.

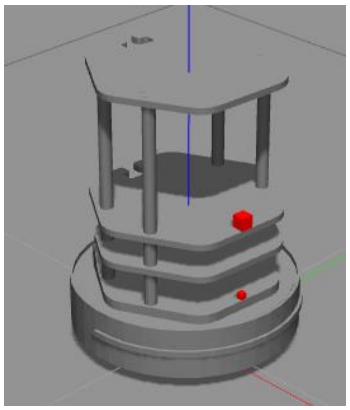

Figure 8. Modelo en Gazebo.

Adicionalmente se creó el paquete teleoperation package que permite controlar los movimientos del robot mediante el teclado. El complemento libgazebo_ros_diff_drive.so controla el movimiento del robot, la comunicación entre el script de control del paquete proyecto_key y el robot realizado mediante el cmd_vel que finalmente permite al usuario tener un control exitoso del robot con el teclado.

El uso de sensores implica crearlos como links del robot ubicándolos de manera estratégica en su estructura para tener un buen campo de visión. Este sensor debe definirse con parámetros como resolución, ángulo de visión, cantidad de muestras tomadas, entre otras. El sensor laser hoyuko utiliza el plugin "libgazebo_ros_laser.so", este comunica la medición del sensor y su visualización por medio del asunto o “topic” *scan*. Para la cámara se debe realizar un proceso similar.

IV. MAPEO Y PLANEACIÓN DE TRAYECTORIAS USANDO SLAM Y AMCL

Algunos robots autónomos usan los métodos de SLAM y AMCL para mapear un ambiente determinado y navegar de manera autónoma en él una vez ha sido guardado. SLAM significa Localización Y Mapeo Simultaneo, esta técnica permite construir mapas del entorno en que se encuentra al reconocer los obstáculos. El AMCL, Localización Apatativa Monte Carlo, es un algoritmo que permite localizar un robot respecto a un mapa, aquí la información del mapa y los datos del sensor le permiten al robot planear una trayectoria para alcanzar un punto deseado siempre y cuando se encuentre en el mapa.

La implementación de la navegación autónoma en la simulación es realizada mediante el paquete de ROS *ros-kinetic-navigation.* El primer paso para realizar el proceso de navegación autónoma es construir el mapa del entorno usando el nodo *slam_gmapping*, que se encarga de crear representaciones 2D del mapa con ayuda del sensor laser.

Se creó un archivo *.launch* para el *gmapping*, este contiene el nodo *slam_gmapping* que suscribe los datos del sensor y publica los datos del mapa. Aquí tambien se debe configurar el nodo *move_base* que permite configurar los planos *global* y *local costmap* además del *local planner.*

```
<launch>
  <arg name="scan_topic" default="scan" />

  <node pkg="gmapping" type="slam_gmapping" name="slam_gmapping"
output="screen">
    <param name="base_frame" value="bloque"/>
    <param name="odom_frame" value="odom"/>
    <param name="map_update_interval" value="5.0"/>
    <param name="maxUrange" value="6.0"/>
    <param name="maxRange" value="8.0"/>
    <param name="sigma" value="0.05"/>
    <param name="kernelSize" value="1"/>
    <param name="lstep" value="0.05"/>
    <param name="astep" value="0.05"/>
```

Figure 9. Parámetros en archivo *gmapping.launch*.

Los controladores del robto y los parámetros de los planos local y global se guardan como archivos *.YAML.* Estos contienen información sobre los controladores de las ruedas, velocidades mínimas y máximas, exactitud al planear trayectorias y la mínima distancia que puede tener el robot respecto a los obstáculos.

La construcción del mapa se realiza corriendo el archivo *proyecto_xacro.launch*, en otro terminal se corre *gmapping.launch* y finalmente, en un tercer terminal, se corre el archivo *keyboard_teleop.launch.* En las Fig. 10, 11 y 12 se muestran el entorno que será mapeado, el proceso de mapeo usando el teclado y el mapa generado. El mapa se genera moviendo el robot por todo el espacio de trabajo y se guarda utilizando el comando *$ rosrun map_server map_server -f nombreMapa.* Este último guarda dos archivos un *.YAML* que contiene información como resolución y datos del mapa y una imagen del plano 2D.

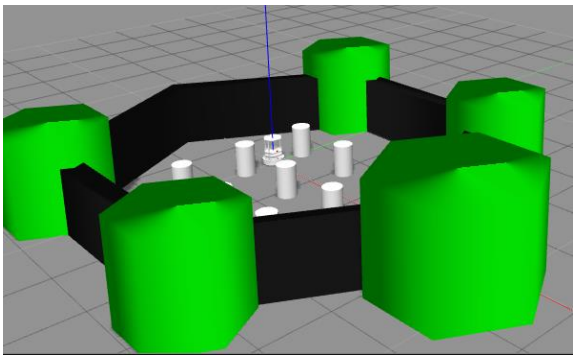

Figure 10. Entorno usado para el mapeo.

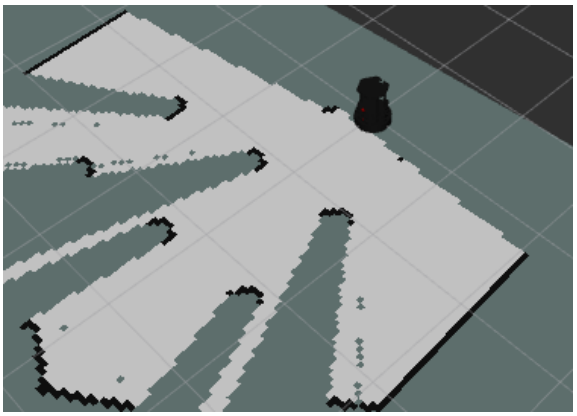

Figure 11. Proceso de mapeo.

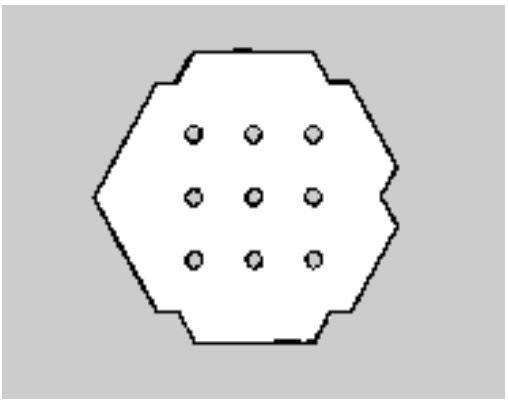

Figure 12. Mapa guardado.

La navegación autónoma se realiza mediante le paquete AMCL, este se encarga de suscribir información del sensor laser, el mapa e información de tf y publica la ubicació estimada del robot respecto al mapa y su posición. Para usar SLAM en ROS fue necesario crear el archivo *amcl.launch* que contiene información del mapa guardado. Tambien se crearon tres archivos *amcl.launch.xml*, *gmapping.launch.xml* y *move_base.launch.xml* que contienen los parámetros del nodo amcl.

```
  <!-- Map server -->
  <arg name="map_file" default="$(find proyecto)/maps/test.yaml"/>
  <node name="map_server" pkg="map_server" type="map_server"
rgs="$(arg map_file)" />

  <arg name="initial_pose_x" default="0.0"/> <!-- Use 17.0 for
illow's map in simulation -->
  <arg name="initial_pose_y" default="0.0"/> <!-- Use 17.0 for
illow's map in simulation -->
  <arg name="initial_pose_a" default="0.0"/>

  <include file="$(find proyecto)/launch/includes/amcl.launch.xml">

    <arg name="initial_pose_x" value="0"/>
    <arg name="initial_pose_y" value="0"/>
    <arg name="initial_pose_a" value="0"/>

<!--
    <arg name="initial_pose_x" value="$(arg initial_pose_x)"/>
    <arg name="initial_pose_y" value="$(arg initial_pose_y)"/>
    <arg name="initial_pose_a" value="$(arg initial_pose_a)"/>
-->
  </include>

  <include file="$(find proyecto)/launch/includes/
ove_base.launch.xml"/>
```

Figure 13. *amcl.launch*

Para ejecutar la navegación autónoma del turtlebot se debe correr la simulación y posteriormente el archivo *amcl.launch* que contiene los parámetros necesarios para planear trayectorias y permitir el movimiento siguiendo la mejor opción. En Rviz deben agregarse plugins como el modelo del robot, sensor laser, mapa, camino y posición. Es importante recordar que las trayectorias son caminos que el robot puede seguir, pero tienen en cuenta el tiempo que puede tardar en recorrerlos.

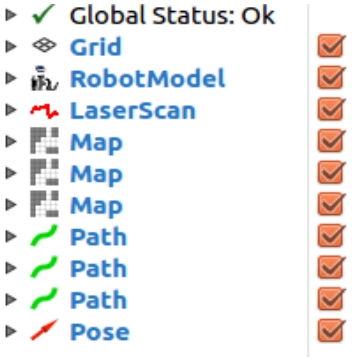

Figure 14. Plugins en Rviz

En Rviz deberá seleccionar el punto deseado para la posición final del robot usando *2D navigation goal* El punto deseado será indicado con una flecha roja y apuntará al lugar donde deberá quedar la parte frontal del robot. Las trayectorias serán mostradas con líneas verdes, o del color que hayan sido configuradas en su respectivo plugin, y teniendo en cuenta el tiempo aproximado en su recorrido el turtlebot escogerá un camino para alcanzar el punto deseado.

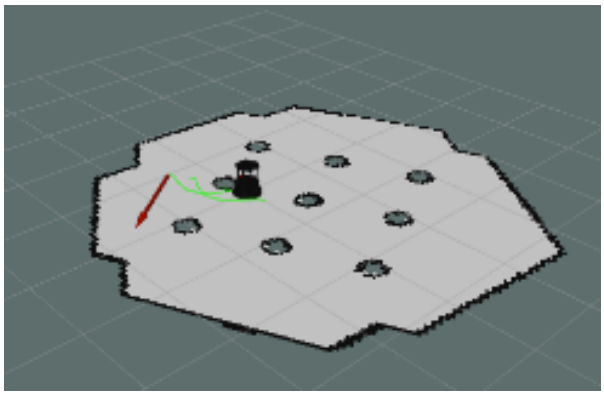

Figure 10. Navegación autónoma en Rviz

En la primera prueba realizada para la navegación autónoma se presentaron inconvenientes como que el robot giraba sobre sí mismo buscando un camino hasta dar una vuelta completa sin escoger un camino. Para solucionar este problema se modificó el archivo *dwa_local_planner_params.yaml.* Se cambió la velocidad mínima por un valor negativo permitiendo el movimiento hacia atrás, se cambió también la distancia mínima que el robot debe tener con los obstáculos y finalmente se modificó el la precisión del controlador para alcanzar el punto deseado en el mapa.

```
# Trajectory Scoring Parameters
  path_distance_bias: 0.8      # 32.0    -
  goal_distance_bias: 0.6       # 24.0
  occdist_scale: 0.5           # 0.01    -
  forward_point_distance: 0.325 # 0.325
  stop_time_buffer: 0.2         # 0.2
  scaling_speed: 0.25           # 0.25
  max_scaling_factor: 0.2       # 0.2
```

Figure 15. Parámetros para la trayectoria.

V. CONFIGURACIÓN DE LA TARJETA JETSON TK1

La tarjeta NVIDIA Jetson TK1 es una tarjeta que permite desarrollar e implementar sistemas de cálculo y visualización en áreas como la robótica usando como sistema operativo Ubuntu 14.04. Esta tarjeta permite la instalación de ROS, lo que a su vez permite aplicaciones robóticas como el mapeo de una ambiente determinado y la navegación autónoma a partir del mapa creado. Es importante resaltar que para algunas aplicaciones es necesario el uso de sensores láser o cámaras.

El proceso de instalación y configuración fue realizado para una de las tarjetas del laboratorio de electrónica de la universidad. El primer paso para la

configuración es descargar e instalar la última versión de Jetpack TK1 (21.3), por ello se revisó si la tarjeta ya tenía una versión previamente instalada. Ya que la tarjeta tenía una versión anterior que no contenía paquetes como CUDA o grinch kernel, se intentó instalar la última versión por medio de un computador host, sin embargo no fue posible porque el computador no reconocía la dirección IP de la tarjeta, impidiendo la instalación.

Posteriormente, se cambió la tarjeta por una que ya tenía instalada la última versión del Jetpack, se reinstaló el sistema de la tarjeta para instalar la versión 21.3 de Linus for Tegra y el grinch kernel 21.3.4. El siguiente paso fue instalar ROS indigo. Sin embargo después de realizar la instalación, al usar el comando *catkin_make* generaba un error, lo que impedía la construcción de los códigos de la carpeta *catkin_ws*. No se pudo corregir el error para la construcción de códigos en el espacio de trabajo *catkin*, por ello se decidió iniciar todo el proceso de instalación, incluyendo la última versión de Jetpack.

Cabe resaltar que esta última vez el proceso de configuración de la tarjeta fue exitoso, sin embargo no se ha realizado una prueba con el robot real, ya que la tarjeta no permite tener conexión wi-fi (es necesario tener una tarjeta adicional para ello) y la conexión entre la tarjeta para el control del robot y el computador "host" se realiza vía wi-fi. A continuación se describe el proceso realizado, nuevamente, para la instalación del sistema y de ROS Indigo.

1. Descargar e instalar la última versión de Jetpack TK1, en este caso, la 21.3.
2. Reinstalar el sistema con Ubuntu 14.04.
3. Actualizar los repositorios del sistema con los comandos *$ sudo apt-get update* y *$sudo apt-get upgrade*.
4. Instalar el custom kernel, en este caso *grinch kernel*, para ello ejecutar las siguientes líneas en el terminal

$ sudo apt-get install git
$ git clone
https://github.com/jetsonhacks/installGrinch.git
$ cd installGrinch
$ ./installGrinch.sh

5. Seguir el tutorial de https://github.com/jetsonhacks/postFlash para mejorar el rendimiento de la tarjeta.
6. Instalar ROS Indigo

$ git clone
https://github.com/jetsonhacks/installROS.git
$ cd installROS
$ ./installROS.sh

7. Instalar essentials

$ sudo apt-get install build-essential

8. Instalar g++

$ sudo apt-get install g++

VI. PROCESO DE MAPEO USANDO SLAM CON EL TURTLEBOT

Para generar un mapa utilizando SLAM con el turtlebot, es necesario haber configurado el turtlebot para su puesta en marcha. Inicialmente para generar un mapa se debe de ejecutar la aplicación gmapping, desde el turtlebot y desde la estación de trabajo.

Desde el turtlebot se inicializa los paquetes de la aplicación y posteriormente el demo pre instalado utilizando

roslaunch turtlebot_bringup minimal.launch

roslaunch turtlebot_navigation gmapping_demo.launch

Posteriormente se ejecuta el archivo launch con la información necesaria del mapa. Desde terminal se inicializa y ejecuta rviz mediante

 # Pre-Groovy
rosrun rviz rviz -d `rospack find turtlebot_navigation`/nav_rviz.vcg
 # Groovy or later
roslaunch turtlebot_rviz_launchers view_navigation.launch

Para finalizar desde el turtlebot se inicializa y corren el paquete de *teleop* utilizando el comando

$ roslaunch turtlebot_teleop keyboard_teleop.launch

Para realizar el desplazamiento del turtlebot en el área y así generar la imagen correspondiente, la cual debe de ser guardada mediante el comando

$ rosrun map_server map_saver -f /tmp/my_map

Cabe resaltar no se debe de cancelar el proceso de gmapping hasta verificar que se guardó adecuadamente el mapa.

VII. NAVEGACIÓN AUTÓNOMA EN UN MAPA CONOCIDO CON EL TURTLEBOT

Teniendo en cuenta que ya se tiene un mapa del área de trabajo y teniendo el turtlebot está configurado para su puesta en marcha, se debe tener cuidado si se utiliza un mapa creado de por el usuario, de manera que sea calibrado basándose en la odometría del turtlebot.

Inicialmente se debe ejecutar el amcl app, desde el turtlebot se debe de ejecutar la amcl app sobre el archivo de mapa a utilizar mediante el comando

$ roslaunch turtlebot_navigation amcl_demo.launch map_file:=/tmp/my_map.yaml

Desde el pc de trabajo, asumiendo se tiene todo lo necesario como ROS y ROS_MASTER_URI para el trabajo con el turtlebot, se debe de ejecutar rviz

 # Pre-Groovy
rosrun rviz rviz -d `rospack find turtlebot_navigation`/nav_rviz.vcg
 # Groovy or later
roslaunch turtlebot_rviz_launchers view_navigation.launch --screen

Posteriormente habiendo ejecutado Rviz es necesario localizar el turtlebot, puesto que en un inicio el trtlebot no reconoce su ubicación referida al mapa, para ello se necesita presionar en pose estimada 2D y seguido indicar mediante un click donde está ubicado el turtlebot y en qué sentido de orientación para que se localice.

Usando Teleoperation, el cual puede ejecutarse simultáneamente con el paquete de navegación, el cual enviará comandos de posición para desplazarse si recibe señales de movimiento.

Es recomendado realizar inicialmente realizar su localización para evitar conflictos antes de enviar comandos de movimiento.

Para finalizar ya posicionado el turtlebot es posible que este realice trayectorias de manera autónoma a través del mapa para ello, se envía una posición de llegada mediante el botón de "2D nav goal" y desde el mapa se indica cual es la posición deseada y la orientación que el turtlebot debería de tener.

VIII. CONCLUSIONES

Gazebo y Rviz son softwares de simulación que permiten crear ambientes para el robot de manera similar a un entorno real, esto permite apreciar el comportamiento que el robot real pueda tener, permitiendo a su vez la corrección de errores antes de la implementación en el modelo real.

Los archivos URDF permiten describir las propiedades de cada parte del robot incluyendo controladores y sensores; esta descripción puede ser optimizada cambiando a archivos tipo xacro, en los que el código es modular y puede ser reutilizado.

El paquete gmapping es una herramienta que facilita el proceso de navegación autónoma puesto que permite realizar y guardar mapas de trabajo utilizando la información de medio.

El entorno de simulación en ROS posee grandes herramientas para la generación de mapas y trayectorias para la navegación de robots móviles aplicando diferentes métodos como SLAM y AMCL.

Los paquetes de SLAM y AMCL permiten realizar procesos de reconocimiento de entornos, mapeo y la posterior navegación del robot móvil en el entorno mapeado; estos procesos pueden ser implementados tanto en modelos simulados como en procesos físicos, siempre y cuando el espacio de trabajo siempre sea el mismo.

El robot diferencial es un robot muy útil, usado gracias a su gran maniobrabilidad y costo de desarrollo, lo cual le permite ser usado en diferentes aplicaciones como agricultura, inspección y mantenimiento, limpieza, seguridad y defensa, entre otras aplicaciones.

El robot diferencial es un robot muy versátil puesto que cuenta con 2 grados de maniobrabilidad, permitiéndole alcanzar diferentes posiciones y orientaciones con tan solo variaciones de velocidad en sus llantas, convirtiéndolo en un robot muy rápido con la máxima cantidad de grados de libertad admisibles.

Ros es un middleware muy útil puesto que nos permite no solo hacer la simulación del robot para estudiar su comportamiento, sino que a la hora de ser implementado físico, solo es necesario modificar algunos datos en el urdf y controladores del robot para

que empiece a trabajar como middleware y así permitir el funcionamiento de nuestro robot físico.

Se pudo ver a lo largo de todo el proceso, la utilidad de los conceptos vistos en clase como la cinemática directa e inversa de robots móviles para entender el funcionamiento de alguno de los controladores de ROS que trabajaban bajo estos conceptos, los cuales al no ser entendidos adecuadamente, el robot no habría funcionado a totalidad.

Para el funcionamiento del turtlebot, es necesario controlarlo y enviarle las tareas a realizar vía wi-fi por medio de un computador, puesto que la tarjeta jetson tk1, cuenta con una capacidad limitada de computo, haciendo que esta se encargue solo de la tarea a realizar, y habilitando el PC host como visualizador de lo que ve y hace el robot, además de control de tareas a realizar.

IX. REFERENCES

[1] J. Lentin, *Mastering ROS for robotics programming*. Birmingham: CPackt Publishing, 2015.

[2] R. Siegwart, I. Nourbakhsh and D. Scaramuzza, Introduction to autonomous mobile robots. Cambridge, MA: MIT, 2011.

[3] Solaque Guzmán, L., Molina Villa, M. and Rodríguez Vásquez, E. (2014). SEGUIMIENTO DE TRAYECTORIAS CON UN ROBOT MÓVIL DE CONFIGURACIÓN DIFERENCIAL. [online] Revistas.usb.edu.co. Available at: http://revistas.usb.edu.co/index.php/IngUSBmed/article/viewFile/298/211 [Accessed 15 Nov. 2017].

[4] Wiki.ros.org. (n.d.). Autonomous Navigation of a Known Map with TurtleBot. [online] Available at: http://wiki.ros.org/turtlebot_navigation/Tutorials/Autonomously%20navigate%20in%20a%20known%20map [Accessed 14 Nov. 2017].

[5] Wiki.ros.org. (n.d.). SLAM Map Building with TurtleBot. [online] Available at: http://wiki.ros.org/turtlebot_navigation/Tutorials/Build%20a%20map%20with%20SLAM [Accessed 18 Nov. 2017].

[6] Nvidia.com. (2017). Jetson TK1 Embedded Developer Kit from NVIDIA. [online] Available at: http://www.nvidia.com/object/jetson-tk1-embedded-dev-kit.html [Accessed 20 Oct. 2017].