# Proyecto Realidad Virtual
# Simulación de un robot diferencial

Jeremías Pino Demichelis
Universidad Nacional de Cuyo, Facultad de Ingeniería, Mendoza
20 de Junio del 2020

ÍNDICE



ÍNDICE DE FIGURAS

**Resumen**

Lejos de ser un tutorial de ROS el siguiente trabajo intenta explicar los pasos básicos para correr y entender la aplicación desarrollada para el proyecto final de la materia Realidad Virtual. Este consistió en obtener la posición de un marcador de realidad aumentada y enviarla para que un robot se dirija hacia esa posición en el mundo virtual

## I. Introducción

En la actualidad se utilizan robots móviles para un gran número de tareas, ya sea por la facilidad que tienen para movilizarse en algunos terrenos o por los riesgos y dificultades que tiene que enfrentar el ser humano (Richard C. Dorf [3]). Ejemplos de ello son los robots Pathfinder y Nomad, ambos diseñados para la exploración espacial. También existen robots encargados de explorar en nuestro planeta, al interior de volcanes, en las profundidades del océano o en los laberintos de las pirámides, logrando llegar a lugares inaccesibles para el hombre. En los últimos años ha surgido también una gran gama de robots destinados al entretenimiento, todos los cuales requieren de sofisticados sistemas de control para desarrollar sus variadas funciones.

En el siguiente trabajo se pretende simular un robot diferencial que sea capaz de navegar autónomamente por su entorno, para esto se hará uso de herramientas tales como GAZEBO y ROS. Una vez realizada la simulación se le enviara una posición objetivo mediante la detección de un objetivo por una cámara usb.

## II. Herramientas

A continuación se detallaran las herramientas utilizadas durante este proyecto

### *II-A. ROS*

Sistema Operativo Robótico (en inglés Robot Operating System, ROS) es un framework para el desarrollo de software para robots que provee la funcionalidad de un sistema operativo en un clúster heterogéneo.

ROS provee los servicios estándar de un sistema operativo tales como abstracción del hardware, control de dispositivos de bajo nivel, implementación de funcionalidad de uso común, paso de mensajes entre procesos y mantenimiento de paquetes.

ROS tiene dos partes básicas: la parte del sistema operativo, ros, como se ha descrito anteriormente y ros-pkg, una suite de paquetes aportados por la contribución de usuarios (organizados en conjuntos llamados pilas o en inglés stacks) que implementan la funcionalidades tales como localización y mapeo simultáneo, planificación, percepción, simulación, etc.

ROS es software libre bajo términos de licencia BSD. Esta licencia permite libertad para uso comercial e investigador. Las contribuciones de los paquetes en ros-pkg están bajo una gran variedad de licencias diferentes. (*Sistema Operativo Robótico* [4])

### *II-B. Gazebo*

La simulación de robots es una herramienta esencial en la caja de herramientas de cada robotista. Un simulador bien diseñado permite probar rápidamente algoritmos, diseñar robots, realizar pruebas de regresión y entrenar el sistema de inteligencia artificial utilizando escenarios realistas. Gazebo ofrece la capacidad de simular con precisión y eficiencia poblaciones de robots en entornos complejos interiores y exteriores. Al alcance de su mano hay un motor de física robusto, gráficos de alta calidad e interfaces programáticas y gráficas convenientes. Lo mejor de todo, Gazebo es gratis con una comunidad vibrante.(*Gazebo* [2])

### *II-C. ALVAR*

ALVAR (a library for virtual and augmented reality) es una biblioteca para la realidad virtual y aumentada. El SDK de ALVAR le permite crear aplicaciones AR con la implementación más precisa, eficiente y robusta para el seguimiento basado en la nube de puntos, marcadores múltiples, imágenes 2D y puntos 3D. ALVAR proporciona una API C ++ de bajo nivel e incluye varias herramientas que ayudan en la creación de aplicaciones AR. Su interfaz de bajo nivel permite desarrollar soluciones personalizadas que se pueden integrar en productos y servicios existentes.

ALVAR Tracker es un rastreador 3D avanzado que proporciona nube de puntos, imagen plana y seguimiento de orientación para aplicaciones de realidad aumentada en dispositivos móviles. Ha sido desarrollado por VTT Technical Research Center of Finland Ltd.(*ALVAR* [1])

## III. Descripción de la aplicación

En la figura 1 se puede observar la estructura general de la aplicación. Esta consta de una cámara usb que obtiene la imagen cruda del entorno. Mediante procesamiento de imágenes se puede obtener la posición y la orientación del objetivo en el ambiente. Una vez obtenidas, se envían al navegador para luego enviar al robot al objetivo. Cabe aclarar que las flechas en la imagen simbolizan la dirección de la información.

La aplicación se encuentra dividida en 4 paquetes ros:

- raspimouse_description: Este paquete contiene la descripción del robot y los archivos launch para lanzar la visualización y el nodo ar_tack_alvar.
- raspimouse_gazebo: Este paquete contiene la descripción del mundo a simular.
- raspimouse_navigation: Este paquete contiene las configuraciones del paquete navigation stack y el archivo launch para generar el mapa del mundo.
- simple_navigation_goal: Este paquete toma la posición del marcador captada por la cámara y se le en vía una posición de objetivo al paquete de navegación.


Figura 1. Estructura de la aplicación

## IV. Desarrollo de la aplicación

Ahora a modo de tutorial se procederá a detallar los pasos realizados en el desarrollo de la aplicación.

### *IV-A. Instalación*

Primero que nada vamos a instalar el software necesario para poder desarrollar nuestra aplicación. Para esto es necesario tener instalado el sistema operativo Ubuntu, si no sabes como hacerlo a continuación dejo un link donde explica como hacerlo.

Instalación de Ubuntu: https://ubuntu.com/tutorials/tutorial-install-ubuntu-desktop#1-overview

Una vez instalado el sistema operativo (SO) procedemos a instalar ROS and GAZEBO, para esto puedes seguir el siguiente tutorial e instalar Desktop-Full Install, si usas esta instalación, ya estarás instalando GAZEBO tambien.

Instalación ROS: http://wiki.ros.org/melodic/Installation/Ubuntu

Ya instalado ROS, procedemos a instalar los paquetes necesarios para la ejecución de nuestra aplicación.

*IV-A1. Cámara:* Para que nuestra cámara pueda comunicarse con ROS se necesitan un par de paquetes *libuvc_camera* y *usb-cam*. Para instalarlos en una terminal copia el siguiente código.

```
sudo apt-get install ros-melodic-libuvc-camera
sudo apt-get install ros-melodic-usb-cam
```

Luego para probar la instalación:

```
roslaunch usb_cam usb_cam-test.launch
```

Deberías ver algo como la imagen 2.

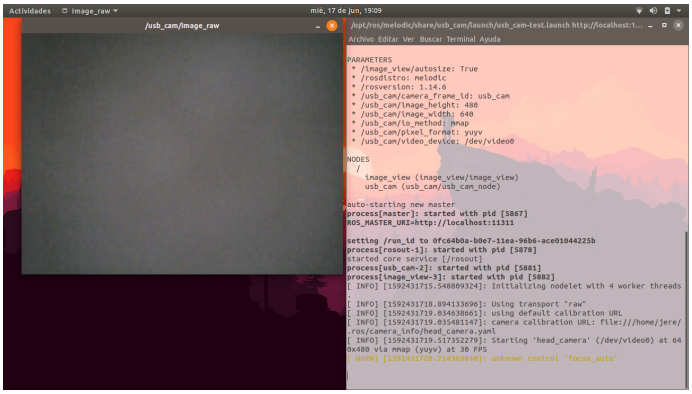

Figura 2. Prueba Cámara

*IV-A2. ALVAR:* Para la detección y el seguimiento de marcadores utilizaremos el paquete de *ar_track_alvar*, para instalarlo copiamos la siguiente linea de código en la terminal:

```
sudo apt-get install ros-melodic-ar-track-alvar
```

*IV-A3. Clonar el repositorio git:* Finalmente clonamos el repositorio que contiene el espacio de trabajo y los paquetes con nuestra aplicación.

```
git clone https://github.com/jerepino/raspimouse_ws.git
```

*IV-A4. Catkin Work Space:* Ahora vamos a inicializar nuestro espacio de trabajo. Para esto vamos a iniciar un *catkin_workspace*. Abrimos una terminal y copiamos linea por linea los siguientes comandos.

```
cd ~/raspimouse_ws          # Navigate to ros workspace root
catkin init                 # Initialize workspace
```

Básicamente un catkin work space[1] es un directorio donde colocamos los paquetes que están siendo editados por el usuario. Si todo salio bien a este paso deberías ver un árbol como el de la figura 3 y la ningún error en la inicialización.

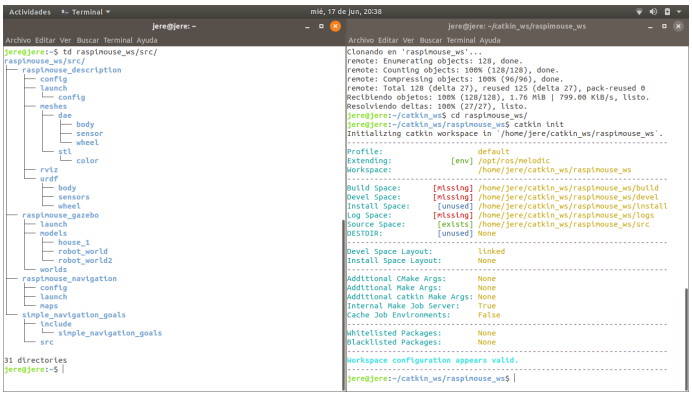

Figura 3. A la izquierda el árbol de directorios y a la derecha el resultado de catkin init

## *IV-B. Descripción del robot*

Para describir nuestro robot usamos Xacro (XML Macros). Este es un XML macro lenguaje utilizado para simplificar archivos URDF (Unified Robot Description Format) utilizados para representar un modelo de robot. No es el objetivo de este tutorial explicar en profundidad estos lenguajes. Pero puedes ver más de estos en el siguiente link *tutoriales URDF*.

A continuación se detalla brevemente el archivo raspimouse_urg.urdf.xacro ubicado en el directorio /raspimouse_ws/src/raspimouse_descri

En la primera parte del archivo se pueden observar los include de nuestro xacro. Estos son otros archivos xacro que contiene marcos (elementos de código que aceptan parámetros) que facilitan el mantenimiento y la reutilización de código en los modelos de robots. Los archivos incluidos son los sensores, las ruedas y el cuerpo del robot.

[1]Se recomienda agregar en el archivo .bashrc la siguiente linea para ahorrar tiempo **source /home/your_user_name/raspimouse_ws/devel/setup.bash**

```
<?xml version="1.0"?>
<robot name="raspimouse_on_gazebo"
       xmlns:sensor="http://playerstage.sourceforge.net/gazebo/xmlschema/#sensor"
       xmlns:controller="http://playerstage.sourceforge.net/gazebo/xmlschema/#controller"
       xmlns:interface="http://playerstage.sourceforge.net/gazebo/xmlschema/#interface"
       xmlns:xacro="http://ros.org/wiki/xacro">

<!-- =============== Include MACROS =============== -->
  <xacro:include filename="$(find raspimouse_description)/urdf/body/body_urg.urdf.xacro"/>
  <xacro:include filename="$(find raspimouse_description)/urdf/wheel/wheel.urdf.xacro"/>
  <xacro:include filename="$(find raspimouse_description)/urdf/sensors/lightsens.urdf.xacro"/>
  <xacro:include filename="$(find raspimouse_description)/urdf/sensors/lrf.urdf.xacro"/>
```

Listing 1. Includes

En la siguiente parte se puede ver como se crean los eslabones y las articulaciones del robot llamando a los macros previamente incluidos. Una cosa a destacar es el eslabón base_footprint que es el eslabón de referencia del robot. Este sera usado para conectarlo con el mundo virtual.

```
  <!-- =============== Link & Joint =============== -->
  <!-- Base -->
  <link name="base_footprint"/>

  <xacro:base parent="base_footprint">
    <origin xyz="0 0 0.00185"/>
  </xacro:base>

  <!-- Wheel -->
  <xacro:wheel prefix="right" parent="base_link">
    <origin xyz="0 -0.0425 0.02215" rpy="1.57 0 0"/>
    <axis xyz="0 0 -1"/>
  </xacro:wheel>
  <xacro:wheel prefix="left" parent="base_link">
    <origin xyz="0 0.0425 0.02215" rpy="-1.57 0 0"/>
    <axis xyz="0 0 1"/>
  </xacro:wheel>

  <!-- Sensors -->
  <xacro:lrf_sensor prefix="urg_lrf_link" parent="base_link">
    <origin xyz="0.0 0.0 0.14060" rpy="0 0 0"/>
  </xacro:lrf_sensor>
```

Listing 2. Eslabones y articulaciones

En la ultima parte se pueden ver los *Gazebo plugins*. Los complementos de Gazebo brindan al modelo URDF una mayor funcionalidad y pueden vincular mensajes ROS y llamadas de servicio. Estos son los encargados de generar los datos de salida de los sensores y habilitan a Gazebo a manipular el robot. Para mayor entendimiento se recomienda ver el tutorial.

```
  <!-- ===============  Transmission =============== -->
  <xacro:wheel_trans prefix="right" interface="hardware_interface/VelocityJointInterface"/>
  <xacro:wheel_trans prefix="left" interface="hardware_interface/VelocityJointInterface"/>

  <!-- =============== Gazebo =============== -->
  <gazebo>
    <plugin name="gazebo_ros_control" filename="libgazebo_ros_control.so">
      <robotNamespace>raspimouse_on_gazebo</robotNamespace>
    </plugin>
  </gazebo>

  <!-- Base -->
  <xacro:base_gazebo/>

  <!-- Wheel -->
  <xacro:wheel_gazebo prefix="right"/>
  <xacro:wheel_gazebo prefix="left"/>

  <!-- Sensors -->
  <xacro:lrf_gazebo prefix="urg" base_rad="0" rad_range="4.71" min_range="0.10" max_range="5.6"/>

<raspimouse_on_gazebo/>
</robot>
```

Listing 3. Gazebo plugins

Ahora estamos en condiciones de ver nuestro robot para esto abrimos una terminal y colocamos una por una las siguientes lineas:

```
cd raspimouse_ws
catkin build raspimouse_description  # Build description packages
source devel/setup.bash
roslaunch raspimouse_description display_xacro.launch model:='$(find raspimouse_description)/urdf/
    raspimouse_urg.urdf.xacro'
```

Luego de ejecutar podrás ver a nuestro robot en el entorno RVIZ figura 4

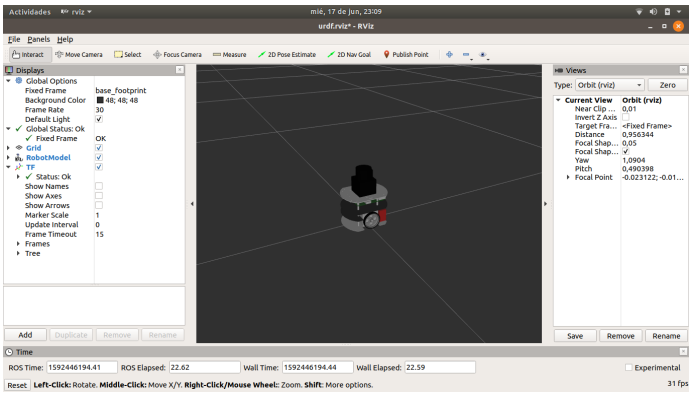

Figura 4. Robot visualizado en RVIZ

Ahora para ver que nuestras ruedas definitivamente se mueven. Finalizamos el proceso mediante Ctrl + C en la terminal y copiamos la siguiente linea.

```
roslaunch raspimouse_description display_xacro.launch model:='$(find raspimouse_description)/urdf/
    raspimouse_urg.urdf.xacro' gui:=true
```

Esto hará que aparezca una ventana en la cual podremos darles valores a nuestras articulaciones de rotación (las ruedas). Para ver mas claro el movimiento se recomienda tildar en el apartado TF de al izquierda la opción show axes (figura 5).

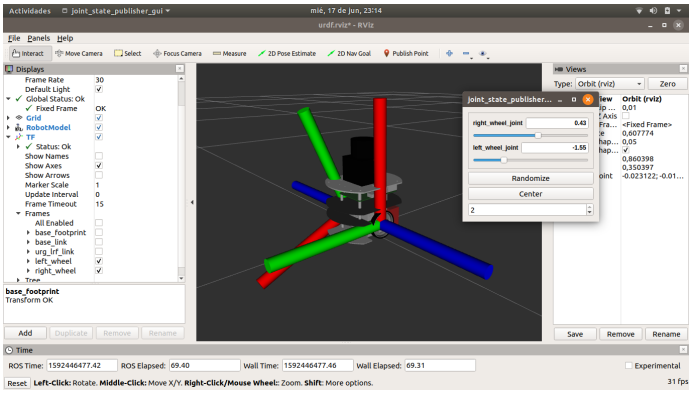

Figura 5. Robot visualizado en RVIZ + articulaciones

### *IV-C. Mundo en Gazebo*

Ahora veamos un poco de Gazebo. Para esto podemos iniciarlo directamente desde el menú o desde un launch file. A continuación explicaremos un poco esta forma.

```
<launch>

  <!-- these are the arguments you can pass this launch file, for example paused:=true -->
  <arg name="paused" default="false"/>
  <arg name="use_sim_time" default="true"/>
  <arg name="gui" default="true"/>
  <arg name="headless" default="false"/>
  <arg name="debug" default="false"/>
  <arg name="model" default="$(find raspimouse_description)/urdf/raspimouse.urdf.xacro"/>

  <!-- We resume the logic in empty_world.launch, changing only the name of the world to be launched -->
  <include file="$(find gazebo_ros)/launch/empty_world.launch">
    <arg name="world_name" value="$(find raspimouse_gazebo)/worlds/camera_world_1.world"/>
```

```
    <arg name="debug" value="$(arg debug)" />
    <arg name="gui" value="$(arg gui)" />
    <arg name="paused" value="$(arg paused)"/>
    <arg name="use_sim_time" value="$(arg use_sim_time)"/>
    <arg name="headless" value="$(arg headless)"/>
  </include>

  <param name="robot_description" command="$(find xacro)/xacro $(arg model)" />

  <!-- push robot_description to factory and spawn robot in gazebo -->
  <node name="urdf_spawner" pkg="gazebo_ros" type="spawn_model"
        args="-x 0.5 -y 0.5 -z 0.1 -unpause -urdf -model robot -param robot_description" respawn="false"
    output="screen" />
<!--
  <node pkg="robot_state_publisher" type="robot_state_publisher"  name="robot_state_publisher">
    <param name="publish_frequency" type="double" value="30.0" />
  </node>
-->
</launch>
```

Listing 4. Launch Gazebo

Para lanzar un nuevo mundo lo que se hace es tomar el launch file de gazebo con mundo vació y pasarle como argumento el mundo que queremos lanzar. En nuestro caso camera_world_1.world. Luego para ver nuestro robot se usa un nodo provisto por el paquete gazebo_ros y se le pasan como argumentos la posición y el modelo del robot que queremos poner en el mundo. Para ver nuestro robot en gazebo colocar el siguiente comando en una terminal.

```
roslaunch raspimouse_gazebo gazebo.launch model:='$(find raspimouse_description)/urdf/raspimouse_urg.urdf.
    xacro'
```

Deberías ver algo como en la figura 6

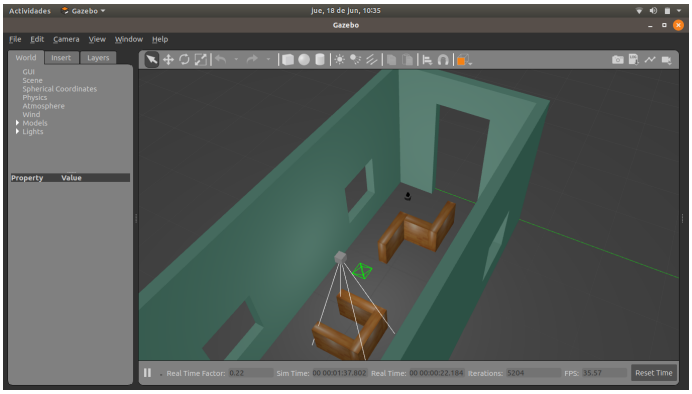


Figura 6. Gazebo

### *IV-D. Mover el robot al rededor del mundo*

Para esto se necesita 2 cosas. La primera es tener nuestros gazebo plugin en nuestra descripción del robot y la segundo es un archivo de configuración YAML que contendrá los parámetros de configuración de nuestros controladores. Este archivo se muestra a continuación.

```
# Publisher
joint_state_controller:
  type:                         "joint_state_controller/JointStateController"
  publish_rate:                 10

mobile_base_controller:         #Name of controller
  type        :                 "diff_drive_controller/DiffDriveController"

  left_wheel  :                 'left_wheel_joint'
  right_wheel :                 'right_wheel_joint'
  publish_rate: 10                # default: 50
  pose_covariance_diagonal :    [0.001, 0.001, 1000000.0, 1000000.0, 1000000.0, 1000.0]
  twist_covariance_diagonal:    [0.001, 0.001, 1000000.0, 1000000.0, 1000000.0, 1000.0]

  # Wheel separation and diameter. These are both optional.
  # diff_drive_controller will attempt to read either one or both from the
```

```
  # URDF if not specified as a parameter
  wheel_separation :           0.085
  wheel_radius :               0.024

  # Wheel separation and radius multipliers
  wheel_separation_multiplier: 1.0 # default: 1.0
  wheel_radius_multiplier    : 1.0 # default: 1.0

  # Velocity commands timeout [s], default 0.5
  #cmd_vel_timeout:            0.25

  # Base frame_id
  base_frame_id: base_footprint #default: base_link
  enable_odom_tf:              true
```

En el archivo se pueden ver definido dos tipos de controles el primero joint_state_controller es el encargado de publicar el estado de todas las articulaciones de nuestro robot. Y el segundo mobile_base_controller es el encargado de controlar el movimiento del robot. Este controlador es del tipo diferencial. Lo que hace es enviar velocidad lineal y de giro y luego dividir esto en la velocidad de cada una de las ruedas. Además este controlador publica la odometría.

Ahora para poder controlar nuestro robot es necesario lanzar varios nodos. Estos están en el archivo teleop.launch y a continuación solo se muestran los que tienen que ver con el control del robot.

```
<group ns="/raspimouse_on_gazebo">
    <!-- load the robot model -->
    <param name="/raspimouse_on_gazebo/robot_description" command="$(find xacro)/xacro $(arg model)" />
    <!-- Load the controllers -->
    <!-- Load joint controller configurations from YAML file to parameter server -->
    <rosparam file="$(find raspimouse_description)/config/controller.yaml" command="load"/>

    <node name="raspimouse_controller_spawner" pkg="controller_manager" type="spawner"
       args="mobile_base_controller joint_state_controller">
    </node>
     <!-- convert joint states to TF transforms for rviz, etc -->
    <node pkg="robot_state_publisher" type="robot_state_publisher"  name="raspimouse_state_publisher">
      <param name="publish_frequency" type="double" value="50.0" />
    </node>
  </group>

   <!-- push robot_description to factory and spawn robot in gazebo -->
    <node name="raspimouse_spawner" pkg="gazebo_ros" type="spawn_model"
          args="-x 0.5 -y 0.5 -z 0.1 -unpause -urdf -model robot -param /raspimouse_on_gazebo/
    robot_description" respawn="false" output="screen" launch-prefix="bash -c 'sleep 2.0; $0 $@' "/> <!--
    agregue  .0 al 5-->


  <node name="rqt_robot_steering" pkg="rqt_robot_steering" type="rqt_robot_steering">
    <param name="default_topic" value="/raspimouse_on_gazebo/mobile_base_controller/cmd_vel"/>
  </node>
```

Lo que el código hace es, cargar los parámetros de los controladores en ros, se los envía a gezebo, crea un nodo para obtener los marcos de referencia y finalmente envía el robot a gazebo y abre un publicador para enviar consigna de velocidad al robot.

Para ejecutarlo copia lo siguiente en una terminal.

```
roslaunch raspimouse_description teleope.launch
```

Ahora puedes mover el robot y recorrer el mundo (figura 7).

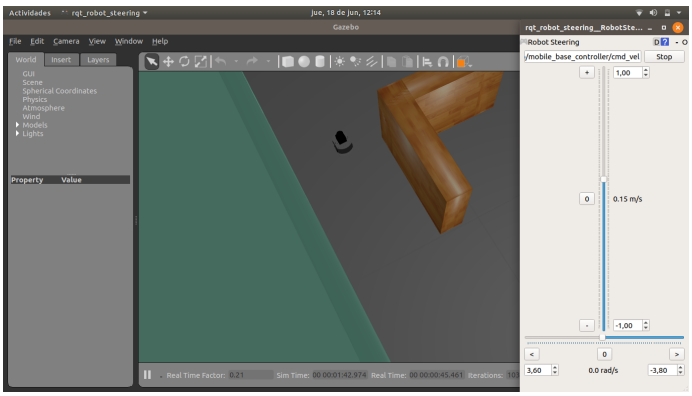


Figura 7. Robot en movimiento

### *IV-E. Navegación*

En esta sección se verá el paquete raspimouse_navigation. En este se encuentran los archivos responsables de la navegación autónoma del robot, para más información ver *navigation package*.

Primero vamos a crear un mapa global para utilizar en la navegación. Para esto abrimos una terminal.

```
cd raspimouse_ws
catkin build raspimouse_navigation
source devel/setup.bash
roslaunch raspimouse_description teleope.launch gui:=false
```

Luego en otra terminal.

```
roslaunch raspimouse_navigation gmapping_demo.launch
```

Ahora seleccionar en MAP topic /map. Deberá verse como en la figura 8.

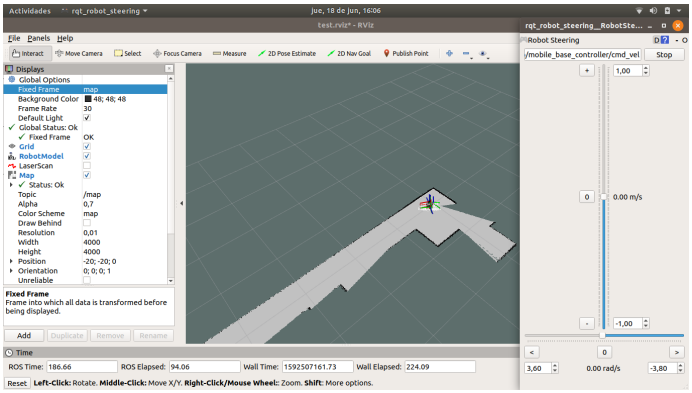


Figura 8. Mapeando

Ahora tendrás que navegar el robot al rededor de todo el mundo hasta que puedas ver el mapa como en la figura 9.

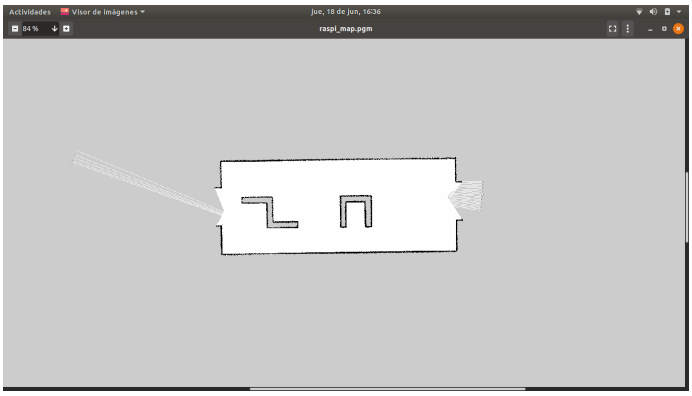


Figura 9. Mapa generado

Ahora para generar el mapa abrimos otra terminal y copiamos lo siguiente.

```
rosrun map_server map_saver -f /home/<user_name>/raspimouse_ws/raspimouse_navigation/maps/tutorial
```

Esto generara los archivos tutorial.yaml y tutorial.pgm. Que utilizaremos para cargar nuestro mapa para navegar.

A continuación procederemos a explicar los archivos de configuraciones del paquete navegación. Este requiere 4 archivos. Cada uno de ellos es responsable de una configuración en particular.

Como la pila de navegación guarda información en mapas de costo, necesitamos configurar de donde obtiene los datos para actualizar los mapas, esto es configurado en Costmap_common_params.yaml.

Las configuraciones de los mapas de costo locales y globales estan albergados en local_costmap_params.yaml y global_costmap_params.yaml respectivamente.

Finalmente el base_local_planner es responsable de computar los comando de velocidades para ser enviados a la base móvil dado el plan de alto nivel y el archivo base_local_planner.yaml guardar los valores limites de velocidad y aceleración.

Más información de como poner a punto el stack de navegación y de los parámetros en *RobotSetup*.

Ahora para mover nuestro robot primero en una terminal colocamos lo siguiente.

```
roslaunch raspimouse_description simulation.launch gui:=false
```

Y en otra terminal.

```
roslaunch raspimouse_navigation navigation.launch
```

Se abrirá la interfaz de RVIZ y con el botón 2D Nav Goal podrás hacer clic en el mundo (figura 10) y ver como el robot se dirige a la posición indicada (figura 11).

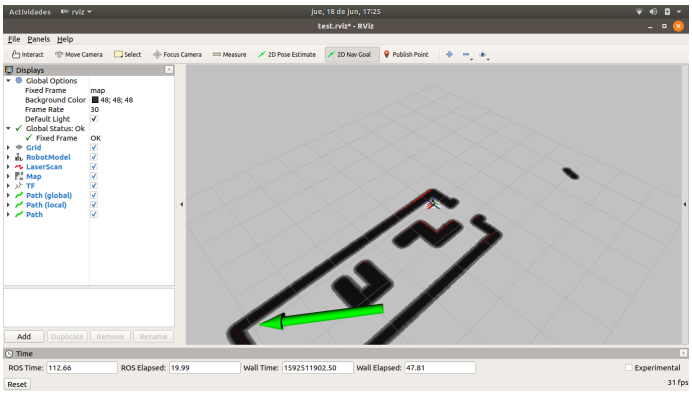


Figura 10. 2D Nav Goal

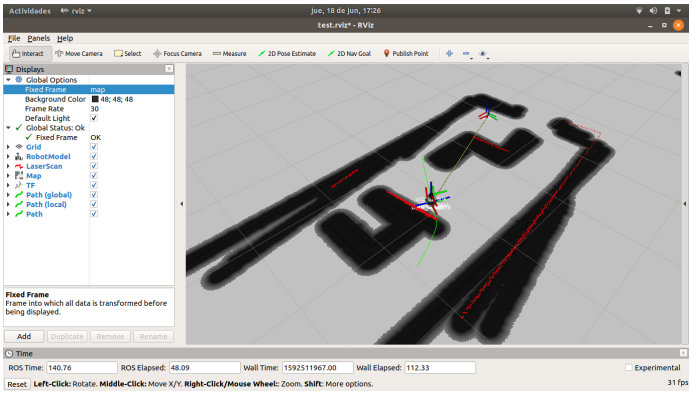


Figura 11. Robot Navegando

### *IV-F. Obtención de la posición de los marcadores*

Lo primero que vamos a hacer es ver los parámetros y en que consiste el paquete *ar_track_alvar*. Este paquete básicamente identifica y sigue la posición de marcadores individuales. Para su uso se requiere especificar los siguientes parámetros:

- marker_size (double): Medida en centímetros de uno de los lados del cuadrado negro del marcador. En nuestro caso 5.4 cm
- max_new_marker_error (double): Umbral que determina cuando un nuevo marcador puede ser detectado bajo incertidumbre. En nuestro caso se deja el valor por defaul.
- max_track_error (double): Umbral que determina que tanto error al seguir puede ser observado antes que un marcador sea considerado que ha desaparecido. En nuestro caso se deja el valor por defaul.
- camera_image (string): Nombre del tópico que provee los cuadros de la cámara para detectar los marcadores. En nuestro caso /usb_cam/image_raw
- camera_info (string): Nombre del tópico que provee los parámetros de la calibración de la cámara así la imagen puede ser rectificada. En nuestro caso /usb_cam/camera_info
- output_frame (string): Nombre del marco de referencia respecto al cual los marcadores serán relativos. En nuestro caso usb_cam

Antes de lanzar nuestro nodo hay que calibrar los *parámetros internos de la cámara*. Para esto utilizamos el paquete *camera_calibration*. Para esto puedes seguir el turorial *Monocular Calibration*. Los prerrequisitos son tener un tablero de ajedrez que puedes descargar del tutorial y tener una cámara publicando las imágenes sobre ros. Para esto bastara con lanzar nuestra cámara.

```
roslaunch usb_cam usb_cam-test.launch
```

Luego de esto puedes seguir los pasos del tutorial que resumidamente es lanzar el siguiente comando y seguir las instrucciones de la interfase.

```
rosrun camera_calibration cameracalibrator.py --size 8x6 --square 0.011 image:=/usb_cam/image_raw camera:=/
    usb_cam
```

Una vez finalizada la calibración, se recomienda generar el archivo **head_camera.yaml** y colocarlo en el directorio **/home/-your_user_name/.ros/camera_info** para que nuestro nodo usb_cam pueda utilizar nuestros parámetros.

Ahora lanzaremos nuestra cámara y el nodo ar_track_alvar y lo visualizaremos en RVIZ. En una terminal lanzamos el nodo de nuestra cámara nuevamente y en otra lo siguiente.

```
cd raspimouse_ws
source devel/setup.bash
roslaunch ar_track_alvar pr2_indiv_no_kinect.launch marker_size:=5.4 cam_image_topic:=/usb_cam/image_raw
    cam_info_topic:=/usb_cam/camera_info output_frame:=usb_cam
```

Ahora en una tercer terminal lanzaremos rviz.

```
rosrun rviz rviz
```

Antes de poder visualizar deberemos colocar cambiar el nombre del parámetro Fixed Frame a usb_cam y agregar una cámara, un Marker y TF, para ello tocamos en el botón add y los marcamos. Ahora una vez agregada la cámara en el parámetro image topic colocar el tópico de nuestra cámara. Ahora puedes jugar con los marcadores y ver como el programa es capaz de seguirlo (figura 12 y 13).

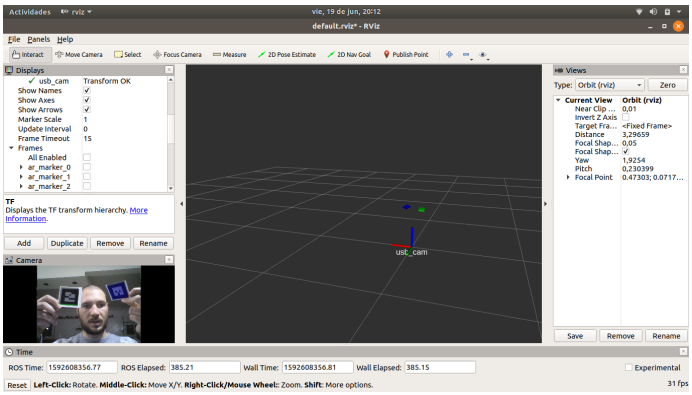

Figura 12. Marcadores

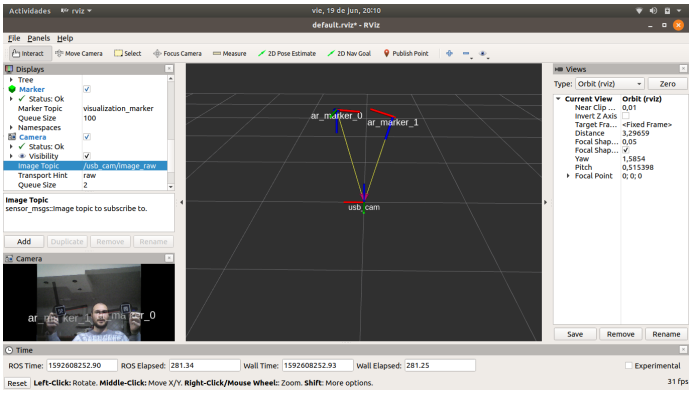

Figura 13. TF y Marcadores

## *IV-G. Integración de todos los paquetes juntos*

Una vez entendido el funcionamiento de cada parte de la aplicación es hora de ponerlas todas juntas.

Primero vamos a compilar el paquete simple_navigation_goal. Abrimos una terminal y copiamos las siguientes lineas.

```
cd raspimouse_ws
catkin build simple_navigation_goal
souce devel/setup.bash
```

Ahora en 4 terminales lanzamos 1 por 1 los siguientes nodos.

```
roslaunch raspimouse_description simulation.launch
```

Listing 5. Terminal 1

Para lanzar la simulación sin la interfaz grafica de gazebo. Esto reduce la carga computacional pero no podrás ver el robot en el entorno virtual gazebo.

```
roslaunch raspimouse_description simulation.launch gui:=false
```

Listing 6. Terminal 1

```
roslaunch raspimouse_description ras_ar_track.launch
```

Listing 7. Terminal 2

```
roslaunch raspimouse_navigation navigation.launch
```

Listing 8. Terminal 3

```
 rosrun simple_navigation_goals pose_listener
```

Listing 9. Terminal 4

A continuación se muestra el camino recorrido por el robot (figura 14), el árbol de transformaciones (figura 15) y la lista de nodos (figura 16) y tópicos (figura 17). Estas ultimas figuras muestran las comunicaciones que se realizan en la aplicacion.

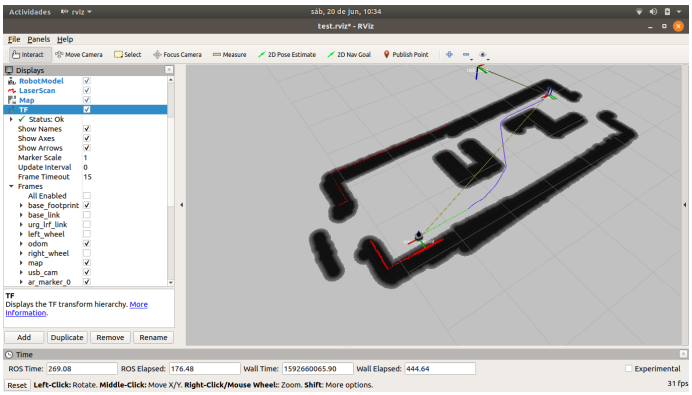


Figura 14. Camino seguido por el robot hasta llegar al objetivo

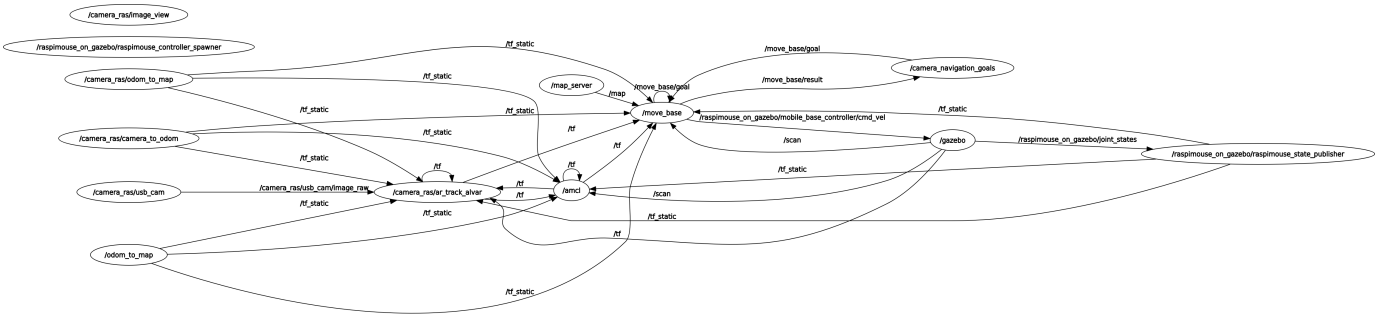


Figura 16. Grafo de nodos en ROS

## V. Conclusión

En este trabajo se pudieron aplicar conceptos aprendidos durante el desarrollo de la materia de realidad virtual. Además se logro profundizar en la utilización de herramientas de simulación como Gazebo.

Finalmente se pretende resaltar la utilidad de dicha aplicación, ya que esta es independiente del entorno de simulación y con los cambios adecuados, fácilmente se podría integrar a cualquier robot diferencial que haya sido rosificado, ejemplos de estos robot son los de la familia *TurtleBot*.

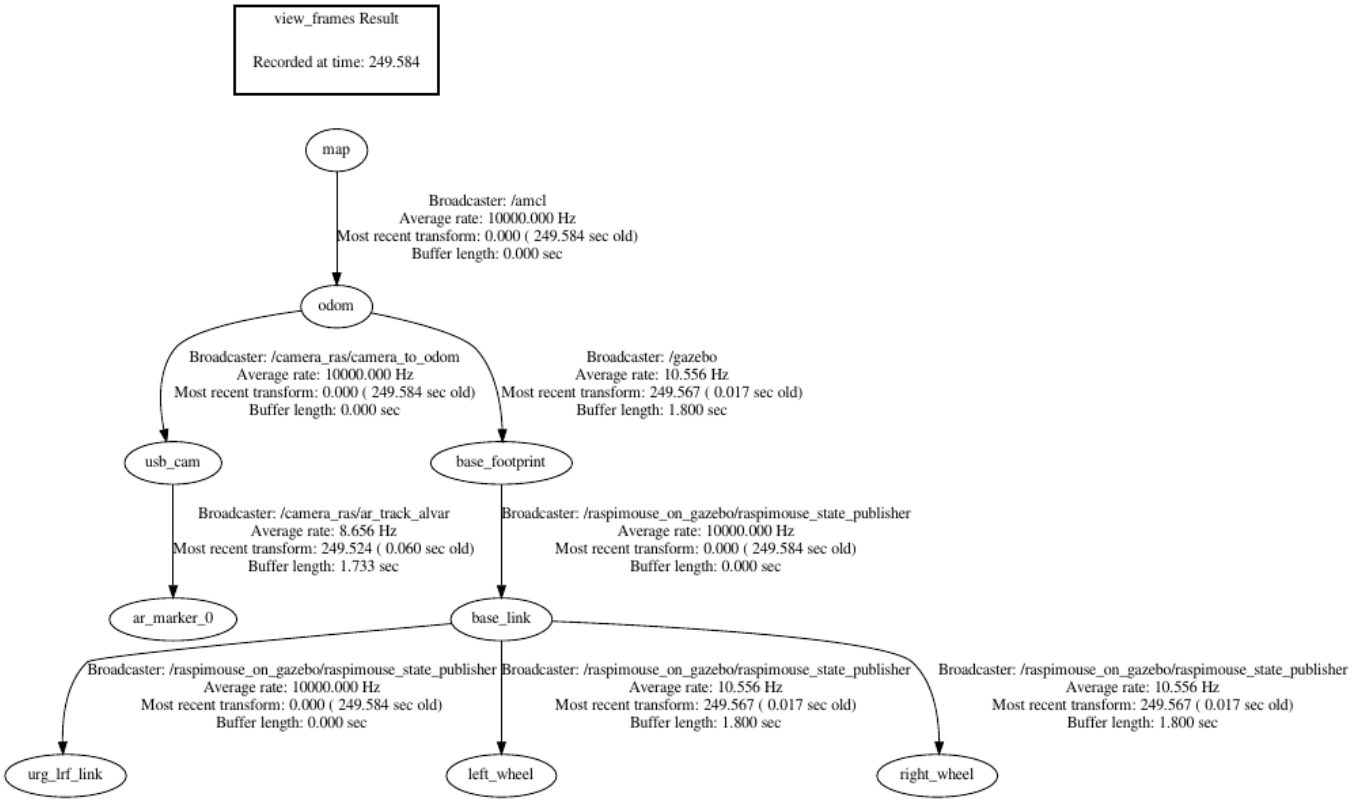


Figura 15. Árbol de transformadas

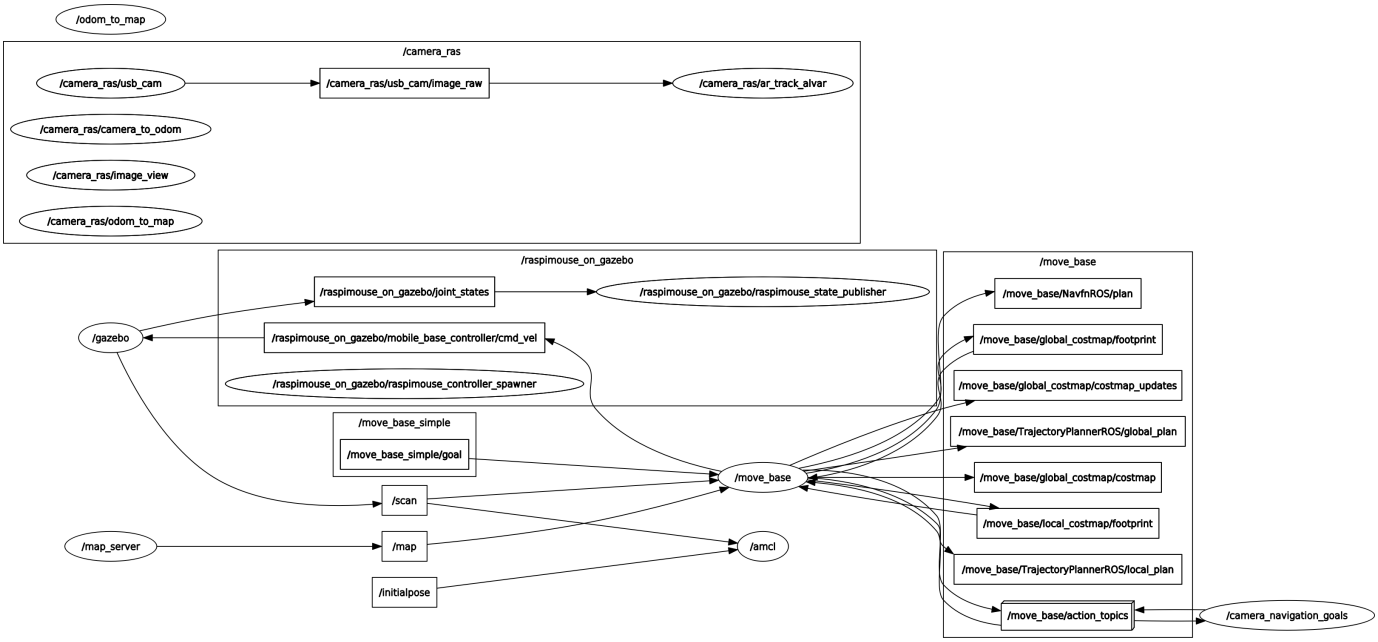


Figura 17. Grafo de tópicos y nodos activos

## VI. Trabajos Futuros

Queda como trabajo futuro construir un robot diferencial y con una mejor cámara probar la aplicación en el mundo real. También se podría diseñar una interfaz gráfica para el usuario, facilitando el ingreso de la posición objetivo.
Finalmente se podría utilizar la librería ALVAR para hacer una aplicación móvil y enviar la posición al robot desde esta.

## Referencias

[1] *ALVAR*. URL: https://docs.ros.org/api/ar_track_alvar/html/.
[2] *Gazebo*. URL: http://gazebosim.org/.
[3] Robert H. Bishop Richard C. Dorf. *Modern control systems*. Prentice Hall, 2000.
[4] *Sistema Operativo Robótico*. URL: https://es.wikipedia.org/wiki/Sistema_Operativo_Rob%C3%B3tico.