Escuela Técnica Superior de
INGENIERÍA DE SEVILLA

Robótica Avanzada - Ampliación de Robótica

Progreso del trabajo (29 abril)

Dron Repartidor

Autores:
Jorge Rodríguez Rubio (GITI)
Eduardo Sotelo Castillo (GITI)
Alejandro Rodríguez Armesto (GIERM)

Profesores:
José Ramiro Martínez de Dios (GIERM)
Guillermo Heredia Benot (GITI)

Sevilla
29 de abril de 2021

Índice

1. Introducción

En este trabajo, realizado en colaboración entre alumnos de Ampliación de Robótica (GIERM) y Robótica Avanzada (GITI), se tratará como objetivo hacer que un UAV con un paquete como carga sea capaz de dejarlo en una ubicación predefinida, de forma automática.
La inspiración para este trabajo ha sido la idea de crear una forma automatizada de entregar paquetes, siendo así una forma ideal de reducir la intervención humana en este proceso. Esta es una idea especialmente interesante de ser usada por empresas de mensajería o de compra por Internet, y de ahí su motivación para ser realizada en este trabajo.
Como aproximación inicial, planteamos crear un mapa donde el dron supiera localizarse, y a su vez fuera capaz de ir a una localización predefinida, soltar el paquete a una altura segura y volver a la posición de origen.
El entorno de simulación usado será ROS y el simulador Gazebo. Este documento ha sido generado usando LaTeX.

2. Objetivos

Para solucionar los retos iniciales que supone realizar este proyecto, hemos planteado los siguientes objetivos a realizar:

- Partiendo de la hipótesis de que siempre repartiremos en un mismo vecindario, dispondremos de un mapa conocido de la zona de operación que estará almacenado en el UAV. El mapa será creado por nosotros en el entorno de simulación Gazebo.
- El control del UAV se realizará con el paquete "grvc_ual", proporcionado por el Grupo de Robótica, Visión y Control de la Universidad de Sevilla. Se ha decidido implementar un controlador ya hecho de cara a simplificar la dificultad del trabajo y centrarnos en otras tareas.
- El dron deberá ser capaz de saber dónde se encuentra sin importar el punto del mapa donde nos encontremos. Asumiremos que se encuentra siempre en un entorno abierto. Para realizar la estimación de la posición inicial usará una señal GPS y las medidas de un Lidar instalado por nosotros, junto a un altímetro. Mediante filtros recursivos bayesianos se estimará tanto la posición inicial como las siguientes.
- El UAV será capaz de planificar el camino a seguir, esquivando los obstáculos que se encuentren en el camino y que, inicialmente, vendrán dados en el mapa.

Además de los objetivos iniciales, también contaremos con objetivos secundarios:

- El UAV será capaz de reaccionar a cambios en su entorno, modificando su trayectoria en caso de que eso ocurra.
- El UAV será capaz de usar técnicas de percepción.

3. Tareas realizadas

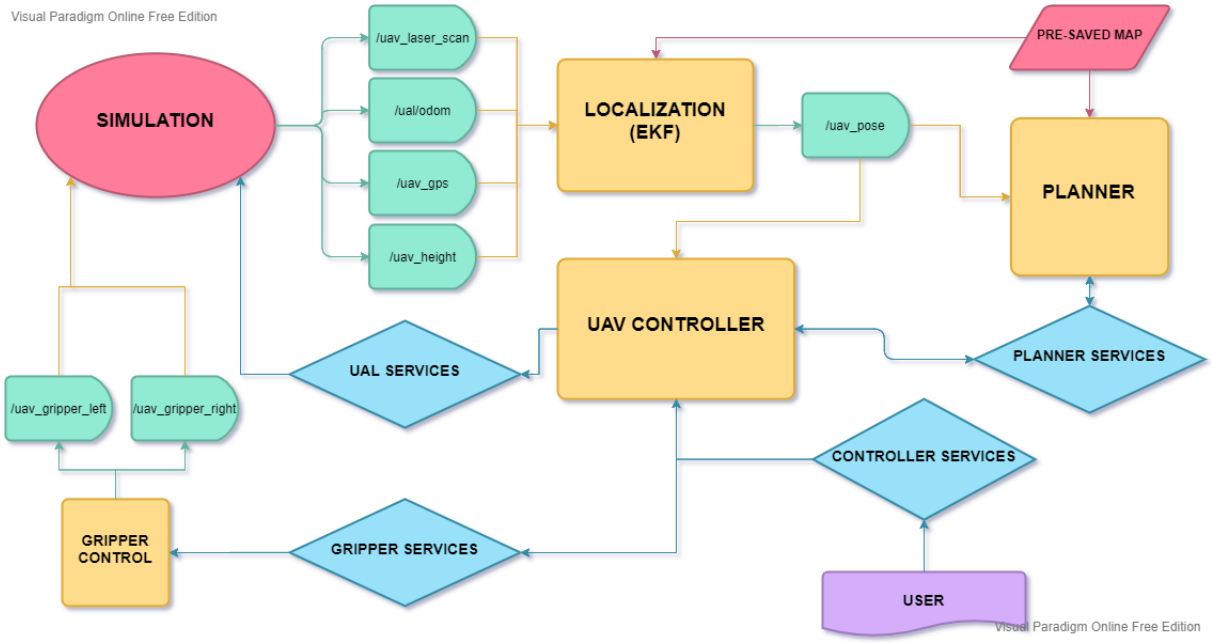
En este apartado se explicará con detalle las tareas realizadas hasta ahora, por todos los miembros del grupo.

3.1. Tarea previa: Aprendizaje de ROS

Dado que no todos los miembros del grupo parten de la misma experiencia con ROS, la tarea previa para estos alumnos ha sido realizar el tutorial de ROS correspondiente a la práctica 3 de la asignatura de Control y Programación de Robots (GIERM), y la práctica 4 para evaluar sus conocimientos con un ejercicio. Además, esta formación se ha complementado con la investigación de otros aspectos relacionados con ROS, como es la creación de robots usando el formato XACRO y URDF. La experiencia adquirida durante este trabajo también se usa de forma complementaria para mejorar los conocimientos de todos los alumnos.

Hecho esto, se procede a investigar cómo afrontar el problema desde el punto de vista de ROS.

3.2. Análisis del problema



Para llevar a cabo el comportamiento definido en los objetivos, implementaremos en ROS el sistema de nodos, servicios y topics que se expone en el diagrama. La funcionalidad de los nodos seria la siguiente:

- **Nodo de Localización:** Este nodo será el encargado de dar la posición concreta del UAV en cada momento. Para ello, podrá tomar la información de los sensores del UAV suscribiendose a los siguientes topic:
  - "/uav_laser_scan": En este topic se publica toda la información relacionada con el lidar.
  - "/uav_gps": En este topic se publican las medidas del sistema GPS.
  - "/uav_height": En este topic se publican las medidas del altimetro.
  - "/ual/odom": En este topic se publica la odometria del UAV, suministrada por el paquete "grvc_ual".

  El nodo utilizará la información de los sensores y, cotejándola con la información del mapa que tenemos guardado en memoria, estimará la posición del UAV en el mapa y la publicará a un topic "/uav_pose".

  Para lograr estimar esta posición, se utilizaran técnicas basadas en filtros recursivos bayesianos.

- **Nodo de Planificación:** Este nodo será el encargado de generar la trayectoria que ha de seguir el UAV. Para ello, obtendrá un punto inicial y un punto final por medio de un servicio y utilizará el mapa almacenado en memoria para encontrar el camino.

  Para obtener el camino de forma segura, se aplicará el algoritmo A* sobre el mapa ya acondicionado, dando prioridad a generar trayectorias seguras.

- **Nodo de Control:** Este nodo tendrá dos funciones:

  La primera es la de controlar el movimiento concreto del UAV procesando la trayectoria generada por el nodo de planificación y utilizando los servicios de control del paquete grvc_ual.

  La segunda es la de actuar como servidor para el resto de los servicios del sistema, así como interfaz para el usuario:

  - Servicios del planificador: Por medio de estos servicios el controlador enviará al nodo de planificación un punto inicial y un punto final y recibirá la trayectoria generada como respuesta.

  - Servicios del controlador: Estos servicios permitiran al usuario interactuar directamente con el sistema, ya sea para darle un punto final al que el UAV debe desplazarse como para interactuar directamente con el control del vehiculo (controlar la garra, acceder a los servicios de UAL o obtener información de la trayectoria).
  - Servicios del gripper: Estos servicios controlaran la garra, permitiendo al controlador decidir cuando hacer que esta se abra o se cierre.

- **Nodo del gripper:** Este pequeño nodo se encargará de publicar en los topic "/uav_gripper_left" y "/uav_gripper_right", que controlan directamente cuanta fuerza se ejerce sobre los dedos de la garra. El objetivo del nodo es que pueda conmutar entre dos estados de "abierto" y "cerrado" cuando se le ordene al nodo por medio de los Servicios del gripper.

3.3. Control con UAL

Tras crear nuestro paquete correspondiente al proyecto, lo siguiente fue investigar cómo usar "grvc_ual"

Este paquete nos proporciona una capa de abstracción sobre el UAV, y nos simplifica el problema del control para varios UAVs ya incorporados.

El primer paso fue instalar e investigar el paquete. Para ello seguimos las siguientes instrucciones (los pasos vienen especificados en la wiki de grvc_ual): https://github.com/grvcTeam/grvc-ual/wiki/How-to-build-and-install-grvc-ual y https://github.com/grvcTeam/grvc-ual/wiki/Setup-instructions:–PX4-SITL-(v1.7.3)-(OLD). En resumen,

- Descargamos el paquete de UAL, asegurándonos de usar la versión 3.0 (probada en Ubuntu 16.04)
- Configuramos los entornos MAVROS y Gazebo Light
- Instalamos el firmware del controlador PX4, configuramos que las simulaciones SITL (Software in-the-loop) se hagan con Gazebo y lo compilamos
- Probamos el funcionamiento:

```
$ roslaunch ual_backend_mavros simulation.launch
```

Como backend, hemos decidido usar MAVROS, dado que implementa un modelo más completo del UAV, a diferencia de Gazebo Light que implementa un modelo más simple. Aunque sea útil para probar cosas, hemos decidido trabajar directamente con un modelo más realista.

Además, el UAV escogido es el "mbzirc", que cuenta con 6 rotores. Nos parece más adecuado usar un mayor número de rotores dado que realizaremos una tarea de carga.

Tras seguir estos pasos, hemos comprobado el correcto funcionamiento del entorno. Para ello, podemos iniciar un vuelo de prueba con:

```
$ rosservice call /ual/take_off/ "height: 2.0 blocking: false" (revisar esto)
```

Esto hará que el UAV despegue, tras lo cual podremos ordenar que vaya a una posición con:

```
$ rosservice call /ual/go_to_waypoint ...
```

Con este comando se logra que el UAV vuele a la ubicación dada.

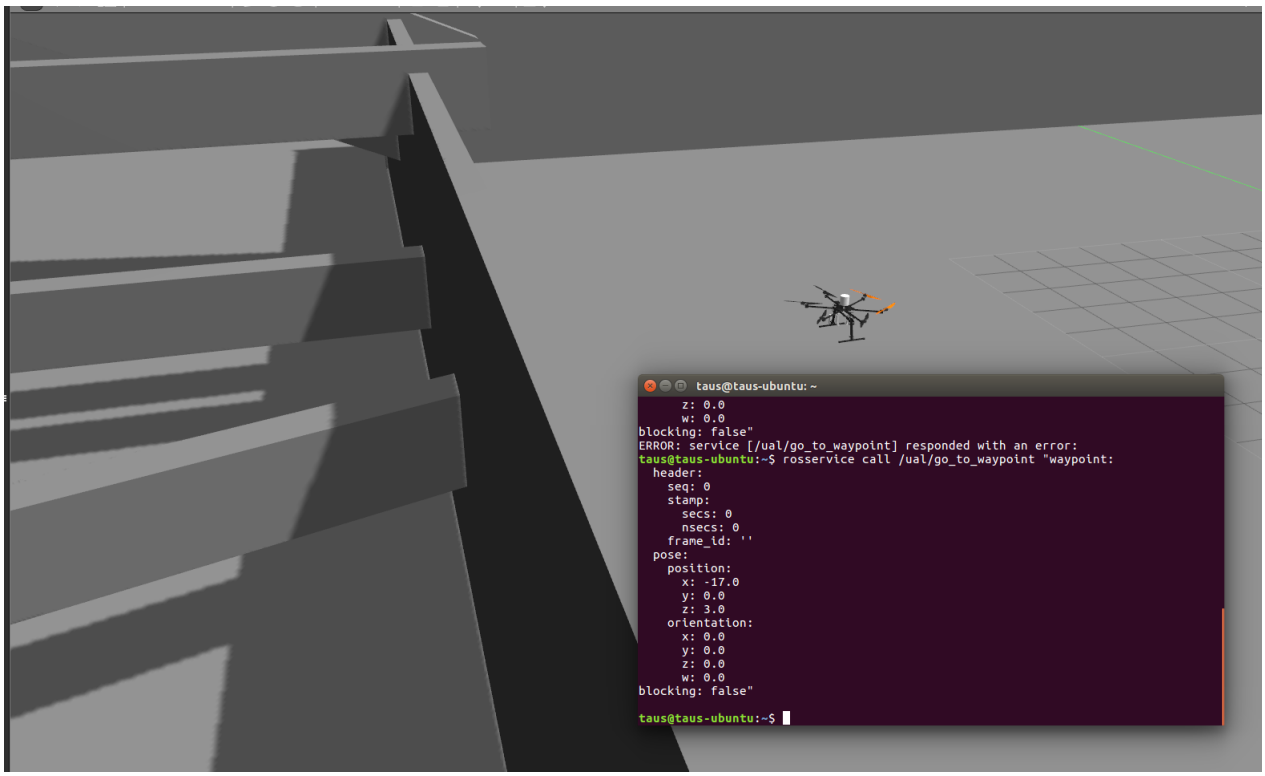


Figura 1: UAV volando en nuestro mapa de prueba con UAL

3.4. Creación y generación del mapa

Una de las características de nuestro problema es el hecho de tener un mapa cargado en memoria y que será usado para la localización del UAV. Inicialmente, consideraremos el problema de generación de mapas en 2 dimensiones, que posteriormente evolucionará al problema tridimensional. Este mapa deberá ser creado con anterioridad y, considerando este asunto un problema menor, usamos 2 paquetes ya preparados:

- "map_server": Es un paquete de ROS que nos proporciona un nodo donde se almacena el mapa. Nos permite tanto cargar un mapa en el nodo como obtener el mapa cargado en el nodo. Actualmente, sólo usamos la función "map_saver", que guarda la información en un archivo .pgm junto a un archivo .yaml que contiene preferencias para interpretar el mapa. Se puede obtener más información en http://wiki.ros.org/map_server
- "gazebo_ros_2dmap_plugin": Se trata de un plugin de Gazebo que nos permite generar un mapa 2D del mundo cargado. Cuenta con preferencias como el origen del mapa a generar, la altura a la que generarla, tamaño y resolución. Este plugin se usa en conjunto con "map_server", y mediante una llamada a un servicio se encarga de crear un mapa y guardarlo. Posteriormente con "map_saver" podremos guardar el mapa generado a un archivo local, donde podremos usarlo con otras herramientas. Más información en https://github.com/marinaKollmitz/gazebo_ros_2Dmap_plugin

Al final, se obtendrá una imagen que podrá ser visualizada por cualquier visor de imágenes, y que no es más que un array con valores de intensidad, dando lugar así a una imagen en blanco y negro. Esta imagen podrá ser procesada con módulos de Python para, por ejemplo, obtener una estimación de la localización.

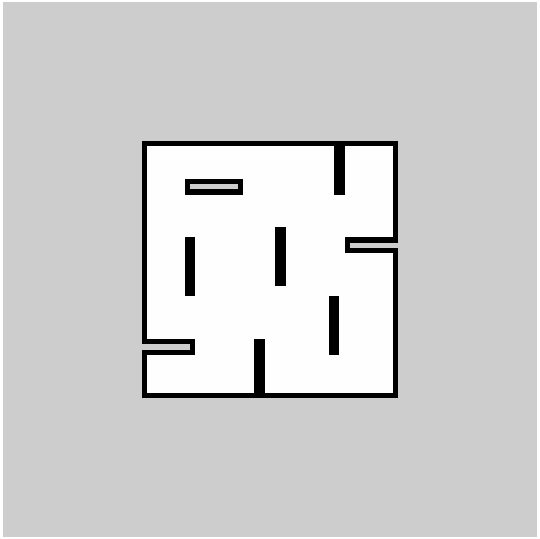


Figura 2: Mapa de prueba

3.5. Implementación de sensor LIDAR en el UAV

Un sensor Lidar (Light Detection and Ranging) es un sensor que nos permite hallar la distancia mediante el uso de un láser. Su funcionamiento se basa en pulsos que son enviados por el sensor, rebotan en la superficie y son detectados de vuelta. Mediante la diferencia de tiempo es posible hacer una estimación precisa de la distancia entre el sensor y el objeto, siempre que este tenga una buena reflectividad y no existan demasiadas interferencias. Una de estas interferencias es la propia radiación creada por el Sol, y que en nuestro problema no consideraremos. Los UAV incorporados con "grvc_ual" no incluyen un Lidar, por lo que se ha tomado la decisión de añadirlo nosotros.

Inicialmente, se ha hecho una copia de los modelos incluidos en el paquete "robots_description" en nuestro paquete, para realizar todas las modificaciones aquí. Tras un proceso de averiguación de cómo copiar y redirigir los scripts y ejecutables de UAL para que apunten a nuestro paquete (todavía en progreso de perfeccionar), se ha procedidido a la modificación en sí. El modelo del robot está definido en el lenguaje XACRO, que nos permite crear macros para el lenguaje URDF, que es el que realmente describe nuestro robot. Por lo tanto, se puede dividir el modelado en varios archivos y así facilitar la lectura. Los archivos (módulos) del modelo son:

- "model.xacro": Es el archivo principal, contiene algunos argumentos básicos y sirve para incluir e instanciar el resto de módulos.
- "component_snippets.xacro": Define las propiedades de cada sensor incluido en el UAV. Cada sensor cuenta con elementos del tipo link (enlaces), joint (articulaciones) y gazebo. A su vez, en el elemento gazebo podremos encontrar elementos del tipo "model", "visual.º "sensor". En este caso, todos los elementos serán del tipo "sensor", e incluyen propiedades. Junto a este elemento, se define el elemento plugin, que es un archivo compilado con las funciones que realiza el sensor. A su vez, también cuenta con preferencias.
- "geometry.xacro" y "multirotor_base.xacro": Relacionados con la construcción física del UAV y sus motores.

A estos archivos, nosotros añadimos los nuestros, y los incluimos en "model.xacro" con la siguiente línea:

```
<xacro:include filename="$(find delivery_uav)/models/mbzirc/nombrearchivo.xacro" />
```

Como decisión de diseño, se ha decidido usar un Lidar inspirado en un modelo comercial, que dará origen al Lidar 3D. Su implementación viene dada en http://gazebosim.org/tutorials?tut=guided_i1

Además del Lidar 3D, se ha creado una versión 2D que hace el análisis del mapa en un plano. Este modelo ha sido creado por nosotros, usando el plugin "libgazebo_ros_laser" ya incluido con Gazebo.

Ambos archivos incluyen una definición de joints, links y elementos gazebo necesarios para el correcto funcionamiento, similar a como funciona en "component_snippets.xacro".

```
<gazebo reference="lidar_top">
<sensor type="ray" name="laz">
   <always_on>true</always_on>
   <pose>0 0 0.117 0 0 0</pose>
   <visualize>true</visualize>
   <update_rate>10</update_rate>
   <ray>
     <scan>
       <horizontal>
         <samples>360</samples>
         <resolution>1</resolution>
         <min_angle>-3.1415</min_angle>
         <max_angle>3.12</max_angle>
       </horizontal>
     </scan>
      <range>
       <min>0.5</min>
       <max>30</max>
       <resolution>0.02</resolution>
     </range>
  <noise>
    <type>gaussian</type>
    <mean>0.0</mean>
    <stddev>0.01</stddev>
  </noise>
  </ray>
  <plugin name="gazebo_ros_head_hokuyo_controller" filename="libgazebo_ros_laser.so">
    <topicName>/uav_laser_scan</topicName>
    <frameName>lidar_top</frameName>
  </plugin>
</sensor>
</gazebo>
```

Figura 3: Fragmento de código del Lidar

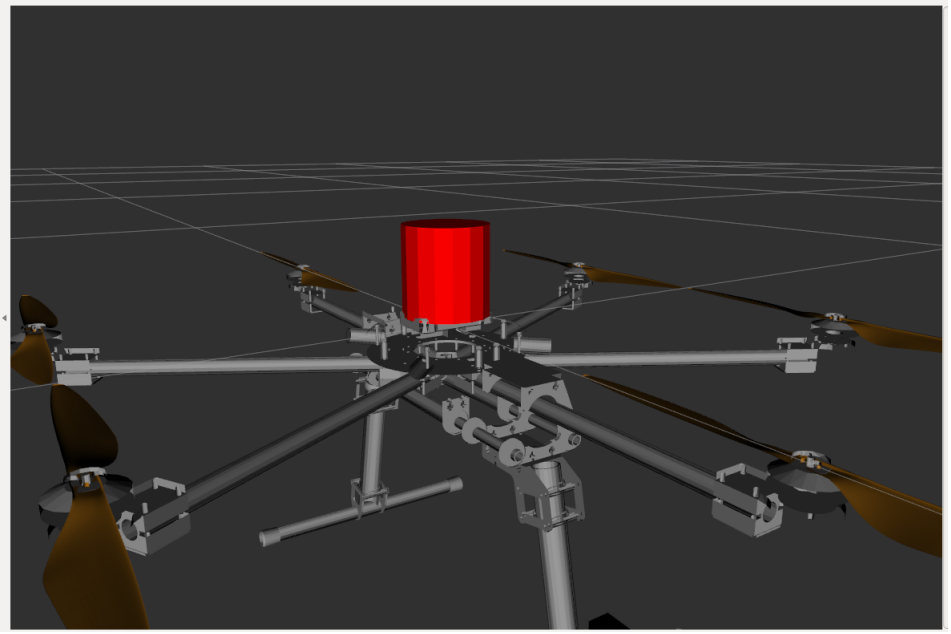


Figura 4: Visualización del Lidar en RViz

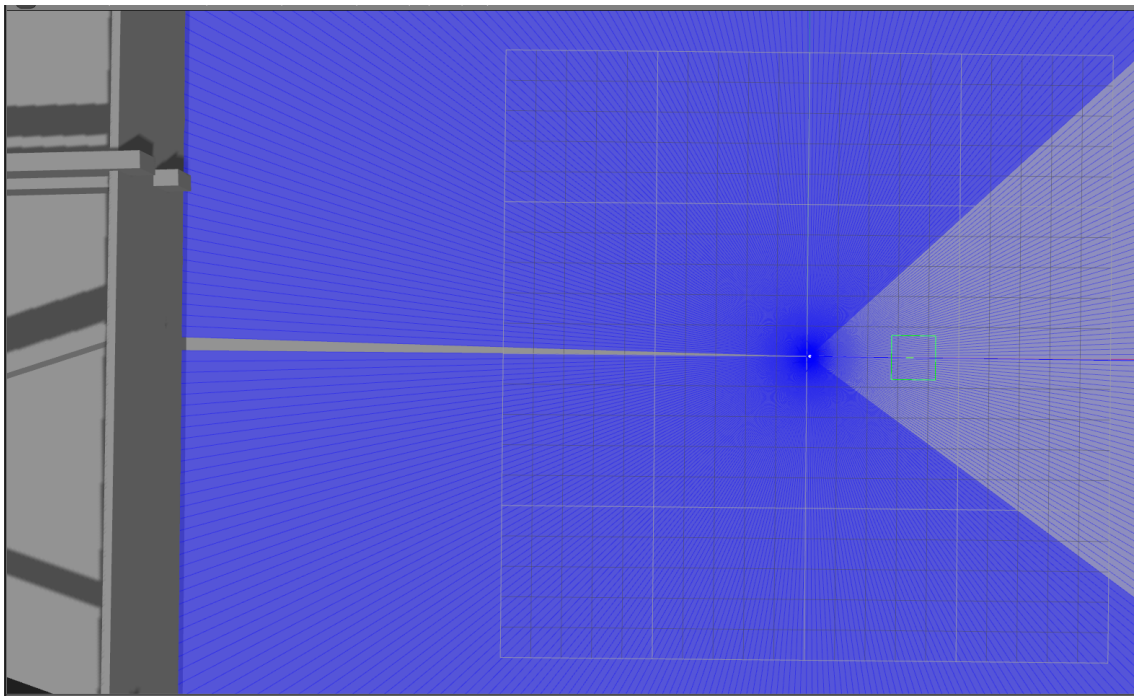


Figura 5: Visualización del Lidar en Gazebo

3.6. Implementación de garra para coger la caja

Para coger la caja, usaremos una garra que será creada por nosotros. Similarmente a como se crea el sensor, podremos crear un conjunto de links y joints que nos definan la garra. Junto a esto, será necesario realizar la implementación de algún sistema que nos permita simplificar el uso de la garra, y que sea capaz de coger el objeto sin realizar cosas raras.

En "simple_gripper.xacro" definimos los elementos de la garra, y con RViz podremos visualizarlos.

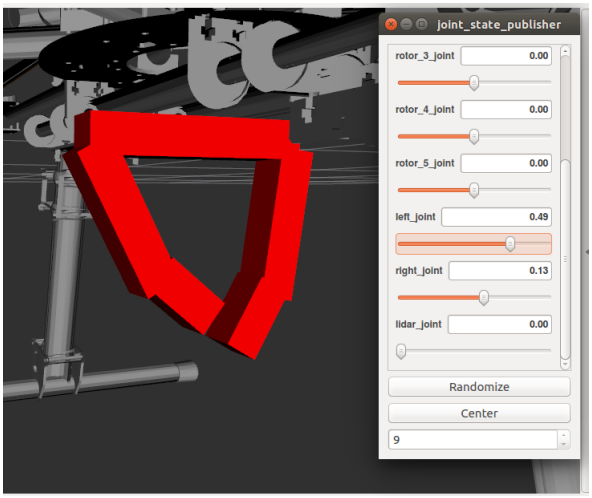


Figura 6: Visualización de la garra en Rviz

```xml
<?xml version="1.0"?>      Dian9914, 5 days ago • first aproach to making th
<robot xmlns:xacro="http://ros.org/wiki/xacro">
    <!-- propiedades del gripper -->
    <xacro:property name="basex" value="0" />
    <xacro:property name="basez" value="-0.09" />
    <xacro:property name="basey1" value="-0.0020" />
    <xacro:property name="basey2" value="-0.0050" />
    <xacro:property name="palm_length" value="0.12" />
    <xacro:property name="defwidth" value="0.020" />
    <xacro:property name="finger_length" value="0.08" />
    <xacro:property name="tip_length" value="0.04" />
    <xacro:property name="tip_rot" value="0.4" />
    <!-- descripcion de los links del gripper-->
    <link name="gripper_palm">
        <collision>
            <origin xyz="${basex} 0 ${basez}" rpy="0 0 0"/>
            <geometry>
                <box size="${defwidth} ${palm_length} ${defwidth}"/>
            </geometry>
        </collision>

        <visual>
            <origin xyz="${basex} 0 ${basez}" rpy="0 0 0"/>
            <geometry>
                <box size="${palm_length} ${defwidth} ${defwidth}"/>
            </geometry>
        </visual>

        <inertial>
            <origin xyz="0 0 0" rpy="0 0 0"/>
            <mass value="${palm_length}"/>
            <inertia
                ixx="0.0" ixy="0.0" ixz="0.0"
                iyy="${(0.015 * palm_length * palm_length)/12}" iyz="0.0"
                izz="${(0.015 * palm_length * palm_length)/12}"/>
        </inertial>
    </link>

    <link name="gripper_left_finger">
        <collision>
```

Figura 7: Fragmento de código para garra

4. Plan de trabajo

Nota importante: Este plan de trabajo está sujeto a ampliación y cambios a medida que el proyecto avance.

Tarea	Alumno	Progreso
Aprendizaje previo de ROS	Eduardo Jorge	Completado
Análisis detallado de tareas	Todos	Completado
Análisis de las herramientas a usar y su viabilidad (búsqueda de paquetes útiles, y cómo trabajar las cosas)	Alejandro	Completado
Búsqueda de forma para generar mapas	Alejandro	Completado
Creación de mapas para probar las simulaciones	Jorge	Completado
Generación experimental de mapas 2D	Jorge	Completado
Implementación de sensor Lidar 2D	Eduardo Alejandro	Completado
Adaptación de plugins y paquetes externos al nuestro	Alejandro	Parcialmente completado
Creación e implementación de garra para el robot	Eduardo	En progreso
Reorganización de archivos y adaptación de CMake	Alejandro	Pendiente
Implementación de filtro recursivo bayesiano para estimación de posición (Nodo ROS)	Alejandro	Pendiente
Creación de planificación (Nodo ROS)	Jorge	Pendiente
Creación de nodo central	Eduardo	Pendiente
Creación del control para el gripper	Eduardo	Pendiente
Realización de experimentos	Todos	Pendiente

5. Resultados de los experimentos

5.1. Generación de mapas

El primer problema con el que nos topamos fue que el plugin no compilaba bien. Sin embargo, tras investigación, conseguimos compilar el archivo .so que se usará como plugin. En la carpeta "plugins" se encuentra una copia comprimida de la carpeta con el código compilado. Más información en http://gazebosim.org/tutorials/?tut=ros_plugins y en la versión no-ROS: http://gazebosim.org/tutorials/?tut=plugins_hello_world

El plugin compilado se encuentra en la carpeta "lib" del proyecto, tras lo cual es importante modificar "package.xml" para incluir como directorio de plugins de Gazebo esta carpeta. Para ello, añadimos lo siguiente al final del archivo:

```
<export>
  <gazebo_ros plugin_path="${prefix}/lib" gazebo_media_path="${prefix}" />
</export>
```

Los problemas no acabaron aquí, y luego descubrimos que el plugin no establecía bien el mapa a distintas alturas. Tras analizar el código, todo lo necesario fue cambiar las preferencias del plugin en el archivo .world para hacer referencia a "map_z" en vez de a "map_height", que es lo que aparece erróneamente en la documentación.

Hecho esto, lo siguiente fue crear los mapas de prueba. Hemos creado dos mapas, que denominaremos "dron_test.world" y "Lab.world":

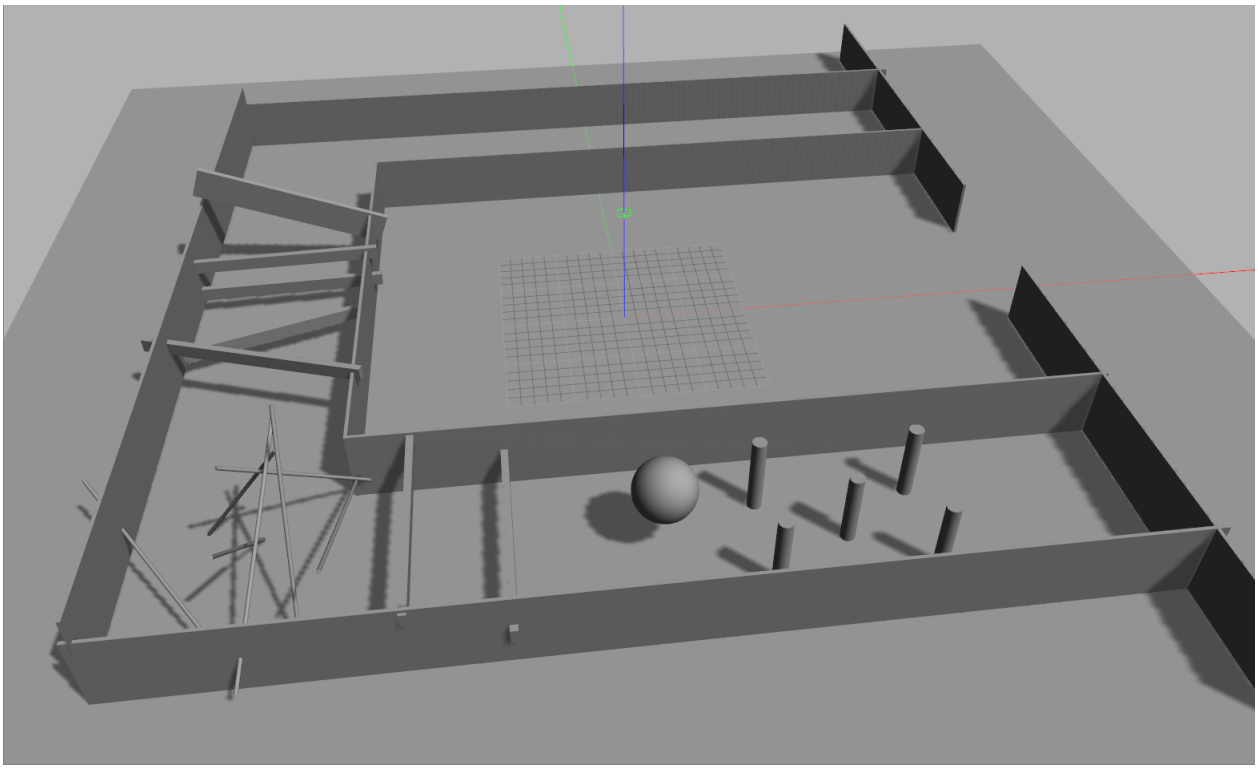


Figura 8: Mapa de prueba dron_test.world

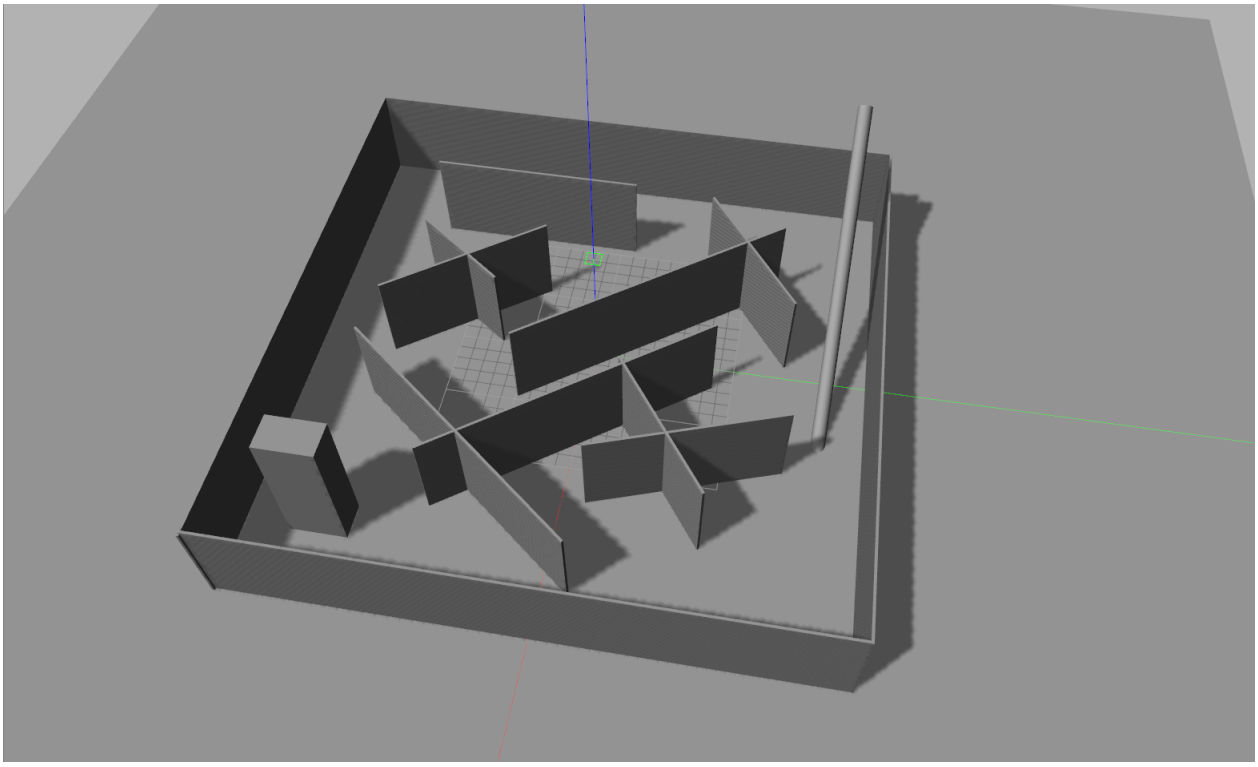


Figura 9: Mapa de prueba lab.world

Un ejemplo de imagen generada sería el siguiente:

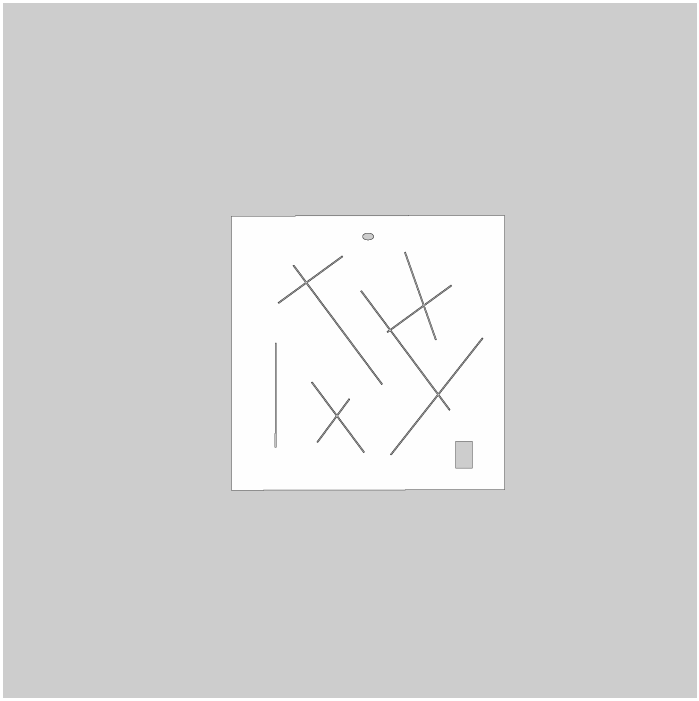


Figura 10: Mapa generado para z=2

Si modificamos la altura...

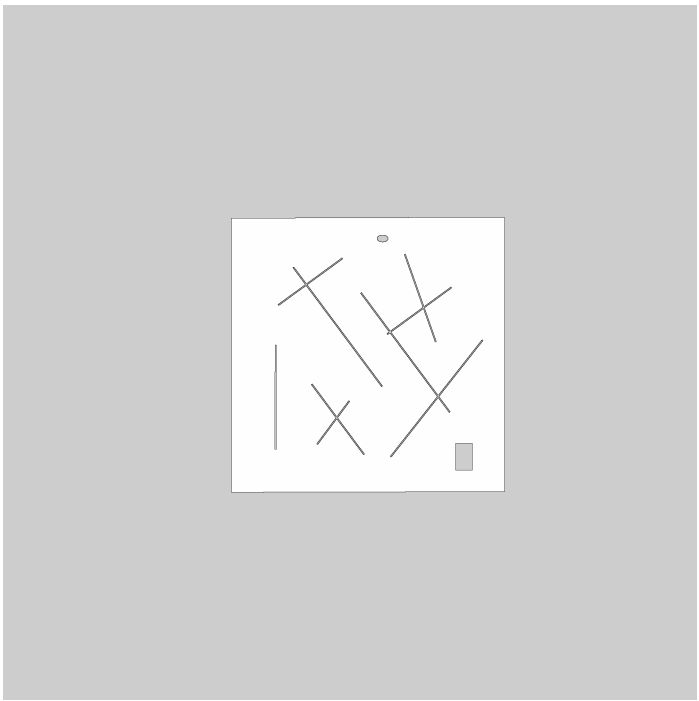


Figura 11: Mapa generado para z=0.3

Se puede apreciar que el palo inclinado se detecta más a la derecha.

5.2. Modificación de UAL

En este momento nuestro paquete depende de que UAL se encuentre instalado, junto al firmware PX4 para los UAVs. Sin embargo, se ha logrado hacer unos archivos .launch propios que cogen los modelos creados por nosotros. Sin embargo, hay que hacer una pequeña modificación en el paquete UAL para que funcione. Los cambios a realizar vienen descritos en el archivo "instrucciones.txt" incluido dentro del paquete. Se buscará implementar una forma menos intrusiva de realizar esto, a su vez de configurar correctamente las dependencias de nuestro paquete.

5.3. Implementación del Lidar

La implementación no fue trivial, dado que Gazebo se congelaba sin mostrar el motivo del error. Tras mucha investigación, encontramos que Gazebo no encontraba el plugin especificado y que necesitábamos añadir la ruta correspondiente, tal y como se ha explicado en el apartado correspondiente. Otro problema con el que nos encontramos es el lidar físicamente, cuyo peso puede desestabilizar al UAV. Además, el movimiento del dron causa que se incline hacia la dirección a la que se mueve, causando que el Lidar no tome una medida de un plano a una altura Z, sino de un plano inclinado. Se investigará cómo tratar este fenómeno.

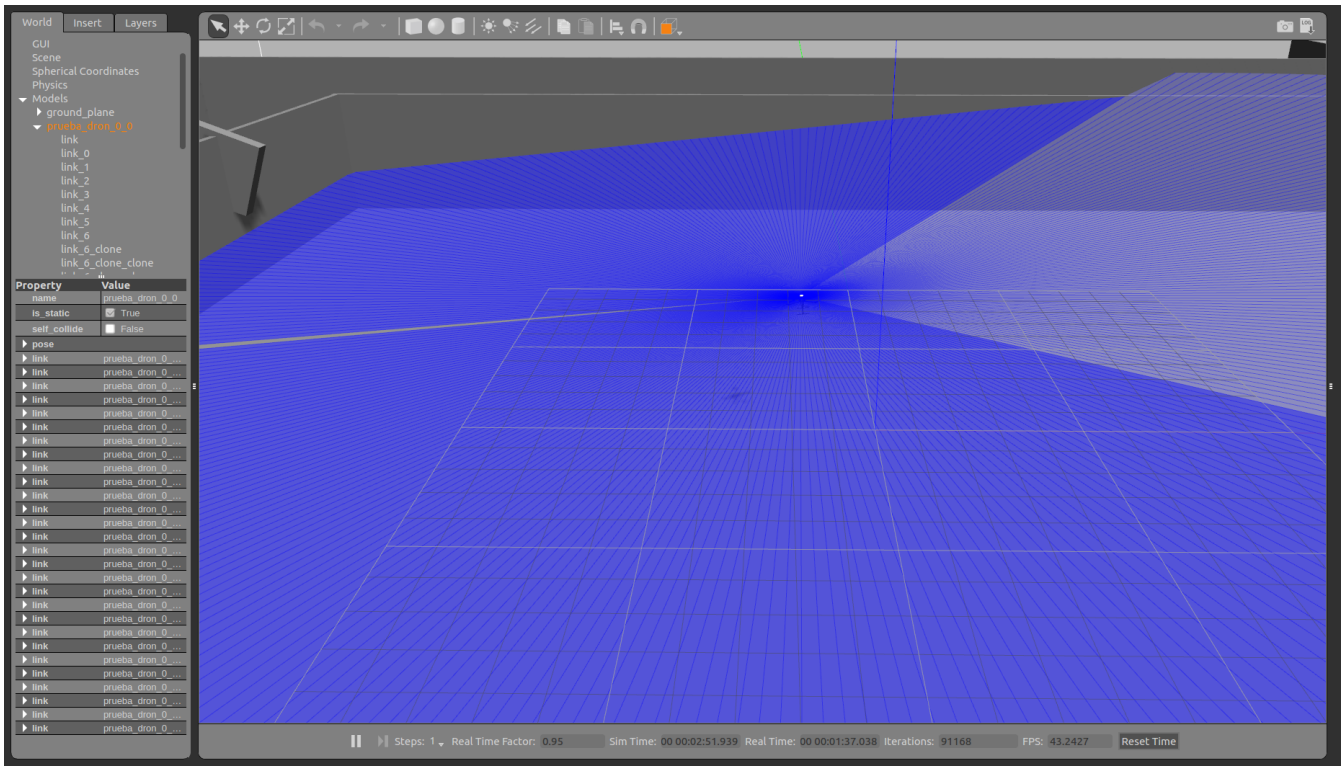


Figura 12: Efecto de la inclinación en el Lidar

5.4. Implementación de la garra

Todavía necesita más progreso hasta lograr un modelo funcional. Hasta el momento, los únicos problemas han surgido debido a que, originalmente, la caja de colisiones del dron era muy grande, y ha requerido modificación para poder implementar correctamente la garra.

6. Conclusiones

Para terminar, nos gustaría concluir diciendo que tenemos una buena base para trabajar y unas buenas herramientas, a su vez de una buena perspectiva de todos los miembros del grupo para realizar multitud de experimentos sobre el funcionamiento.