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

Simulación de un quadrotor en el entorno ROS con la librería Hector

Autor: Francisco Jiménez García

Tutor: Jesús Iván Maza Alcañiz

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

Sevilla, 2020

Trabajo Fin de Grado

Grado en Ingeniería Electrónica, Robótica y Mecatrónica

Simulación de un quadrotor en el entorno ROS con la librería Hector

Autor:

Francisco Jiménez García

Tutor:

Jesús Iván Maza Alcañiz

Profesor Titular

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

Sevilla, 2020

Trabajo Fin de Grado: Simulación de un quadrotor en el entorno ROS con la librería Hector

Autor: Francisco Jiménez García
Tutor: Jesús Iván Maza Alcañiz

El tribunal nombrado para juzgar el trabajo arriba indicado, compuesto por los siguientes profesores:

Presidente:

Vocal/es:

Secretario:

acuerdan otorgarle la calificación de:

El Secretario del Tribunal

Fecha:

Agradecimientos

En el apartado sentimental, me gustaría dar las gracias principalmente a mis padres, por educarme y guiarme en la cultura del esfuerzo y trabajo, además de hacer posible que haya estudiado esta carrera, sin contar con todo el apoyo y cariño diario, gracias. También agradecer a mi familia el trato excepcional durante toda mi vida, gracias. Y por último, pero no menos importante, a mis amigos, gracias por los mil y un favores, charlas y risas durante estos años.
Gracias también a todos aquellos profesores que creyeron en mi, durante mi peor momento, y en especial gracias a mi tutor Iván y a Alejandro, por toda la ayuda, incluso fuera del horario laboral, durante este trabajo de fin de grado.
Y por último, gracias Cristina por conseguir que siguiera en la rama de ciencias y alentarme en el mundo de las matemáticas junto a la enseñanza, mi nota de selectividad es tuya.

Francisco Jiménez García
Sevilla, 2020.

Resumen

En el presente trabajo se explica como integrar los paquetes de ROS; hector_quadrotor y rviz_-satellite con el fin de obtener un entorno de simulación, en Gazebo y Rviz, para un drone de tipo quadrotor, manejado por coordenadas GPS via consola. Durante la memoria se explica como adaptar esta idea, a modo de primer paso de un proyecto real de aplicación agrícola, mediante drones con software propio, en lugar de recurrir a empresas como DJI, con el fin de reducir la inversión inicial de dicho proyecto.

Abstract

This work explains how to integrate ROS packages; hector_quadrotor and rviz_satellite in order to obtain a simulation environment, in Gazebo and Rviz, for a quadrotor drone, managed by GPS coordinates via console. During the report, it is explained how to adapt this idea, as a first step of a real project of aerial application, using drones with its own software, instead of resorting to companies like DJI, in order to reduce the initial investment of said project.

Índice

1 Introducción

En esta primera sección de la memoria se presentará el proyecto en cuestión, asi como una serie de razones por las cuales se ha llegado a este punto. Además se enumeraran los objetivos que se esperan cumplir al finalizar el proyecto, junto con una breve exposición acerca de la estructura que van a seguir las explicaciones.

1.1 Motivación

La robótica se creó, entre otras cosas, para facilitar la vida del ser humano mediante el desarrollo de máquinas con una cierta inteligencia que realizan diversas labores en varios campos, en lugar del propio ser humano, ya sea por peligrosidad, eficiencia, coste, etc. Sin emgargo este proyecto no esta destinado sólo a la mejora cualitativa de la actividad en cuestión; la motivación mayoritaria para desarrollar este proyecto nace en la necesidad de establecer un modelo de negocio que genere unos ingresos considerables mediante una inversión relativamente baja con un riesgo mínimo, si se satisfacen las especificaciones del consumidor, ya que el cumplimiento de dichas espectativas se traduce en un importante ahorro de tiempo y capital por parte del comprador.
En resumen, se busca un proyecto cuyos beneficios sean directamente proporcionales al correcto trabajo e investigación desarrollado a lo largo del mismo, ya que cuenta con un nicho de mercado importante basado en la necesidad de optimizar una actividad que, por si misma, es obligatoria. Se ha elegido para dicho modelo de negocio la idea del fumigado de campos de cultivo, también conocido tecnicamente como aplicación aérea, mediante un dron de tipo quadrotor controlado de manera autónoma por autopiloto y que usa guiado por GPS para llegar a los puntos objetivos que introduce el usuario por consola a modo de misión de vuelo. Dada la flexibilidad de esta solución, se espera que esta metodología sustituya el fumigado tradicional a mano por personas, o por avionetas manejadas por pilotos.
Importante destacar que un proyecto de ese tipo requiere un desarrollo a nivel de hardware y software, por tanto, la inversión de capital y tiempo sobrepasaría los límites razonables de un trabajo de fin de grado. Por esa razón, este proyecto se centra unicamente en el software de control, dejando el desarrollo de hardware como trabajo futuro junto al desarrollo a nivel conceptual de una aplicación Android para el control del drone, a nivel de usuario, bajo una intuitiva interfaz de tipo mapa, donde el usuario solo tendría que marcar en el mapa el punto objetivo el dron.

1.2 Objetivos

Los objetivos marcados para este proyecto son los siguientes:

- Estudio de mercado para ver viabilidad económica del proyecto, así como el precio de los componentes y dispositivos a usar.
- Adquirir conocimientos en Ubuntu, ROS y python.
- Desarrollo de software que permita el vuelo del drone a puntos GPS marcados por el usuario.
- Elaboración de un entorno de simulación donde ver el resultado de lo que se desarrolla a nivel de software.
- Desarrollo conceptual de una aplicación Android que permita el control del drone mediante una interfaz a nivel usuario.

1.3 Estructura del documento

A continuación se explica como se desarrolla la explicación del proyecto:

- Capítulo 2: Se detalla la situación actual del proyecto en relación al mercado actual y las opciones existentes. También se añade una breve explicación de la localización GPS.
- Capítulo 3: Se explica que tipo de software se ha usado, así como los paquetes, librerias, o scripts empleados en la elaboración del mismo, junto al entorno de simulación desarrollado para verificar que se cumplen las exigencias.
- Capítulo 4: Se detalla todos los pasos a seguir en la correcta integración de los paquetes usados para el correcto funcionamiento del sistema.
- Capítulo 5: Conclusiones y trabajo futuro.
- Apénices: Se detalla los aspectos fundamentales de los codigos principales desarrollados en el proyecto.

2 Estado del Arte

2.1 Contexto histórico

La aplicación aérea, o como se conoce coloquialmente, fumigación de cultivos; se basa en la aplicación de pesticidas agroquímicos y otros productos fitosanitarios desde una avioneta agrícola. Esta técnica tiene sus orígenes en Dayton, Ohio, cuando a mediados de 1921 el ingeniero agrícola McCook Étienne Dormoy modificó un avión Curtiss JN4 Jenny, pilotado por John A. Macready, para extender arseniato de plomo para combatir una plaga de orugas (figura 2.1).[1]

Figura 2.1 Instantánea de McCook junto a Jonh A.

El éxito de los resultados en dicha operación, así como la disminución notable de tiempo de aplicación e inversión en dicha tarea, ya que un avión es mas rentable y rápido que contratar a personas, produjeron que las empresas aeronáuticas desarrollasen modelos de aeronaves especializadas en tareas de aplicación de productos fitosanitarios a principios de 1950, estabeciendo las bases de un importante mercado que se extendería por todo el globo. En 1951 el ingeniero aeronáutico Leland Snow diseña el primer avion especializado en la aplicación aérea, su nombre fue S-1 (figura 2.2).[2]

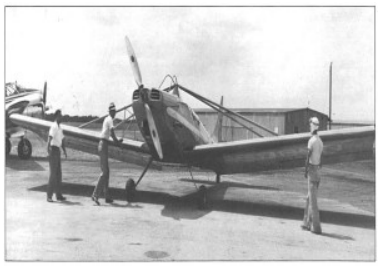


Figura 2.2 Leland S. el primer día de pruebas.

[1] Fuente: www.wikipedia.org
[2] Fuente: www.wikipedia.org

Durante las décadas venideras el sector se expande y evoluciona considerablemente llegando a normalizar su uso, incluso a principios de 1990 en Japón y Korea del Sur se comienza con la investigación de realizar esta tarea mediante aviones no tripulados por necesidades topográficas del terreno ya que ponian en peligro la seguridad del proceso. Se hace uso, a finales de 1991, del Yahama R-MAX (figura 2.3)[3], uno de los primeros mini-helicopteros con autopiloto, mediante el cual, se llevo con gran exito la tarea en cuestión.

Figura 2.3 UAV-Yamaha R-MAX.

Interesante destacar que gran parte del exito de ese proyecto fue gracias a la aparición del autopiloto moderno (figura 2.4)[4], el cual tiene como predecesor al primer autopiloto, inventado en 1914, por el estadounidense Ambrose Sperry y el físico alemán Hermann Anschütz-Kaempfe, los cuales propusieron un sistema que permitía a cualquier vehículo en movimiento mantener un rumbo determinado. Su funcionamiento se basaba en la fusión de la información de varios giroscopos y una brújula en coexión con los elementos de control del vehículo en cuestión.
Con el desarrollo de la electrónica y los diferentes algoritmos de procesamiento de datos a tiempo real, así como tecnicas de fusión sensorial como el Filtro de Kalman, se permite el desarrollo de un dispositivo que procese la información de sensores como acelerómetros, giróscopos, altímetros, GPS ... que en adición al modelo dinámico del dron, permite tener una correcta lectura (cuanta más información mas precisa será la lectura) del comportamiento cinemático y dinámico del dron, pudiendo así abordar el problema desde una óptica de control automático del dispositivo en cuestión.

Figura 2.4 Autopiloto moderno PixHawk-PX4.

[3] Fuente: www.wikipedia.org
[4] Fuente: https://pixhawk.org

A partir de 1970 los paises comenzaron a legislar en andas de regular esta práctica ya que en ocasiones por motivos meteorológicos, el producto llegaba a los núcleos de población. Incluso por mala praxis o falta de normas de seguridad ocurrian desafortunados incidentes que hoy día serian facilmente evitables. El aumento de burocracia y limitaciones en adición al crecimiento de población en nucleos urbanos colindantes a zonas de cultivo, provocó que se buscasen alternativas como la aplicación mediante tractores o vehiculos motorizados, los cuales en ocasiones tampoco son viables por problemas del terreno, y por tanto, se recurría en última instancia al fumigado a mano. Gracias a los avances en robótica de estos últimos 15 años, se ha conseguido perfeccionar el diseño y control de drones de tipo quadrotor, los cuales, por motivos de morfologia, versatilidad y potencia son idóneos para esta tarea por los siguientes motivos:

- Coste menor en dos ordenes de magnitud a una avioneta, sin contar con que apenas precisa de mantenimiento ni intervención humana salvo para trazar el camino o en caso de emergencia.
- Es más preciso, ya que se puede ajustar la altura del dron en el caso de fuerte viento que produzca derivas en la aplicación del producto.
- Tiene mejor movilidad que una avioneta, dado que esta necesita unos 800 metros en dar la vuelta y posicionarse en el punto preciso de aplicación.
- Al usar motores electricos es menos dañino para el medio ambiente.
- No precisa de los mismos certificados ni permisos que una avioneta y, por tanto, su uso es más flexible aunque actualmente haya algunas trabas legales en zonas de tráfico aéreo. Detalle incomprensible dado la baja altura de trabajo del dron.

Debido a esta serie de ventajas, muchos fabricantes vieron la oportunidad de mercado y empezaron el desarrollo de drones de tipo quadrotor (o morfología similar) especializados en la aplicación aérea. Destaca principalmente la compañía DJI, la cual desarrolla tanto software como hardware (figura 2.5)[5], teniendo así en el stock varios modelos de drones especializados en este área. De este modo ofrecen una solución completa del proyecto ya que proporciona un drone que cumple las espectativas del consumidor, en adición a una interfaz a nivel de usuario que cubre todo tipo de necesidades.

(a) (b)

Figura 2.5 a) Dron AGRAS MG-1 DJI b) Software usuario DJI .

Recurrir a DJI es la opción mas segura y rápida, ya que parten de años de investigación y perfeccionamiento, aunque costosa al mismo tiempo, ya que todo el equipo cuesta del orden de 15000 euros, sin contar accesorios extras como baterías, protecciones... Esta solución sería rapida en ejecucuión, ya que solo habría que invertir, pero tiene mayor riesgo al tener menos margen de beneficio.

[5] Fuente: https://www.dji.com/es

Existe la opción de que el usuario desarrolle o integre soluciones de código abierto hasta llegar a la solución global requerida. Para ello se necesitaría el desarrollo de los siguientes aspectos partiendo de un drone fabricado:

- Integración del autopiloto con el drone. Importante tener en cuenta la compatibilidad de ambos elementos. Generalmente se recurre a dispositivos del mismo fabricante para evitar problemas.
- Desarrollo del software de control y supervisión. Como podrían ser el software que controla el proceso a nivel de tarea, o la interfaz que proporciona el estado actual del proceso al usuario.
- Adición de sensores (si el autopiloto no los tuviese), para aumentar así la versatilidad y seguridad del proyecto. Como es el caso de un leader, mediante el cual se coexiona su información con un algoritmo de evitación de obstaculos para dotar de una seguridad necesaria al drone ya que no se sabe a priori la morfología del terreno.

Evidentemente en este proyecto se opta por la opción de código abierto, ya que supondría una inversión inicial mucho menor que si se elige la opción de DJI, sin contar la adquisición del gran número de competencias que supone el desarrollo de un proyecto de este tipo

2.2 Paradigma actual

Abarcar todo lo desarrollado anteriormente en un único Trabajo de Fin de Grado es complejo, ya que habría que desarrollar aspectos a nivel de hardware y para ello se necesita tiempo y capital extra dado el elevado número de contratiempos, tanto a nivel logístico como conceptual, que supone trabajar con hardware.
Por esta serie de razones el estado actual del proyecto se centra en el desarrollo de un entorno de simulación que indique la viabilidad de integrar, mediante software, diversos algoritmos de control para el drone a nivel de tarea, ya que el propio control del dron lo realizaría el autopiloto en condiciones reales. Al final de proyecto se abordará el diseño conceptual de una aplicación Android que dote de una sencilla interfaz al farragoso proceso de controlar un robot mediante una ventana de comandos, teniendo así una solución muy completa y versátil a nivel de software.
A continuación , se muestra un esquema que representa los diversos aspectos que entran en juego durante el proyecto (figura 2.6)[6], ya sea a modo de trabajo actual (Software), o incluso trabajo futuro (Hardware y App), mostrando así al lector una óptica completa del proyecto en sí.

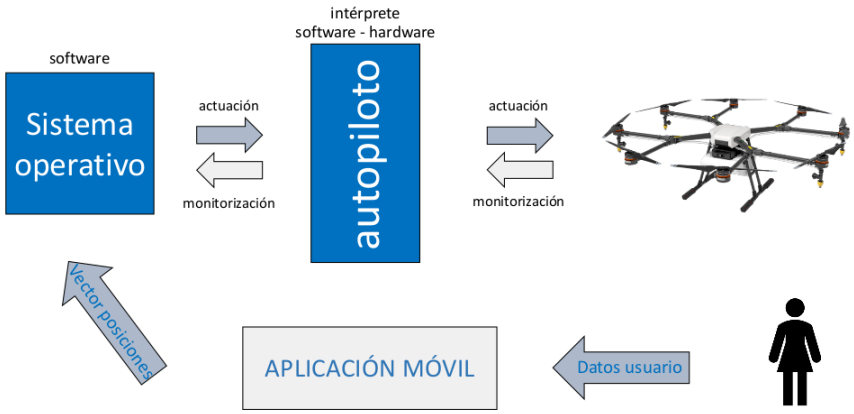


Figura 2.6 Esquema conceptual del proyecto.

[6] Fuente: Elaboración propia

2.3 Introducción a GPS

Es importante detallar los fundamentos básicos del GPS ya que es la forma que en la que se van a introducir las posiciones objetivo del dron, y por tanto, a nivel de software sería necesaria una función, en el lenguaje en que se esté programando, que permita transformar de coordenadas GPS a locales y viceversa. El desarrollo de dicha función se deja para futuros apartados, en esta sección se intenta acotar la explicación desde una perspectiva conceptual y matemática.
Desarrollado por el departamento de defensa de los Estados Unidos a finales del siglo pasado, el sistema GPS, como indican sus siglas en inglés (Global System Position), se usa como método para saber la posición de un objeto situado en la superficie terrestre. Para ello usa una flota de aproximadamente 24 satélites, originalmente NAVSTAR, los cuales envian una señal al receptor en tierra que acto seguido la devuelve al satélite. Si se mide el tiempo de transmisión y considerando que la velocidad de transmisión es la velocidad de la luz, se tienen suficientes datos para determinar una región equidistante al satélite la cual sería una esfera. El hecho que cualquier medida tenga error en adición a que el objeto a localizar esté en movimiento provoca que se necesite repetir este proceso hasta 4 veces para disminuir el error, ya que si cada medida se traduce a una esfera, cuatro medidas se traducen a 4 esferas cuya región de intersección correspondería con el objeto a localizar. Este método se conoce como trilateración, y es una forma de usar la geometría de triangulos para determinar la posición relativa de objetos, de forma análoga a la triagulación. A continuación se explica un ejemplo con 3 esferas ilustrado en la siguiente imagen (figura 2.7)[7]:

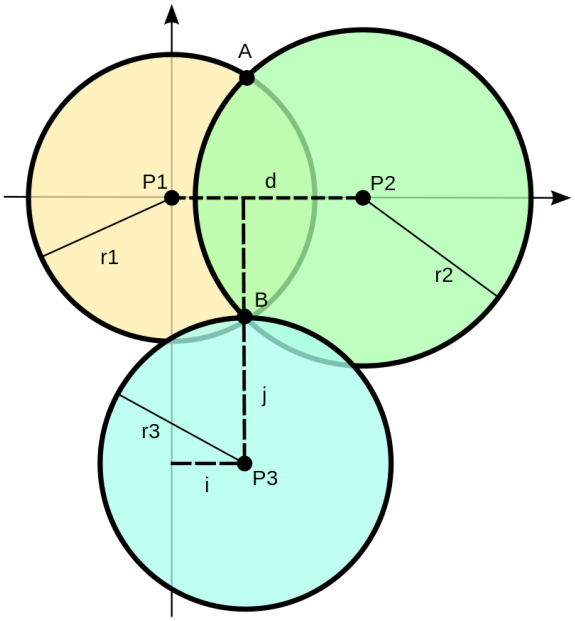


Figura 2.7 Ejemplo de trilateración 2D.

Partiendo del punto B, se quiere determinar la posción relativa respecto los puntos P1, P2 y P3. Para ello se procede de la siguiente forma:

$$r_1^2 = x^2 + y^2 + z^2 \tag{2.1}$$
$$r_2^2 = (x-d)^2 + y^2 + z^2 \tag{2.2}$$
$$r_3^2 = (x-i)^2 + (y-j)^2 + z^2 \tag{2.3}$$

[7] Fuente: www.wikipedia.org

Despejamos x después de restar 2.1 con 2.2

$$x = \frac{r_1^2 - r_2^2 + d^2}{2d} \tag{2.4}$$

Al sustituir en 2.1 se obtiene la formula de un circulo, la cual es la solución a la intersección de las dos primeras esferas

$$y^2 + z^2 = r_1^2 - \frac{(r_1^2 - r_2^2 + d^2)^2}{4d^2} \tag{2.5}$$

Igualando con 2.3

$$y = \frac{r_1^2 - r_3^2 - x^2 + (x-i)^2 + j^2}{2j} = \frac{r_1^2 - r_3^2 + i^2 + j^2}{2j} - \frac{i}{j}x \tag{2.6}$$

Con las coordenadas x e y solo queda despejar z de 2.1 y sustituir

$$z = \sqrt{r_1^2 - x^2 - y^2} \tag{2.7}$$

De esta forma se obtiene las coordenadas tridimensionales del punto B respecto la esfera de referencia, en este caso r_1. A continuación se muestra una imagen tridimensional del problema para mejor entendimiento (figura 2.8)[8].

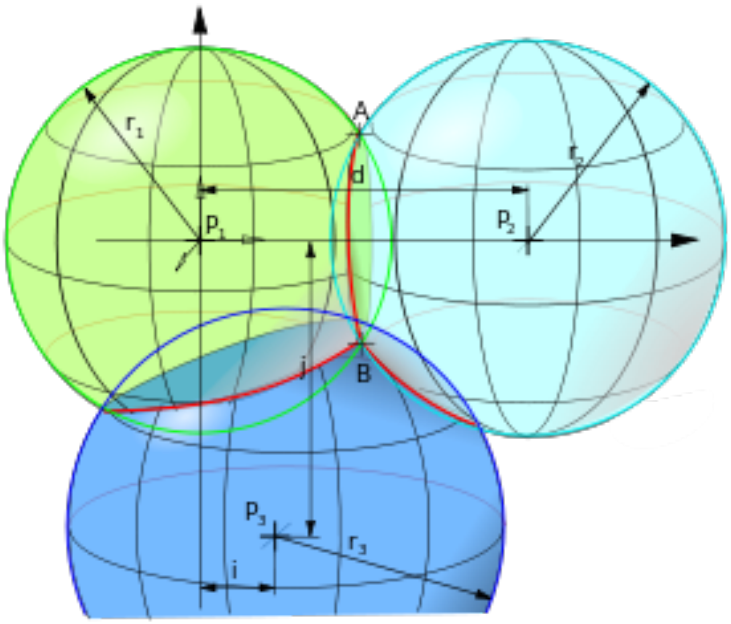


Figura 2.8 Ejemplo de trilateración 3D.

Importante destacar que la precisión real de un GPS es del orden de metros, ya que por motivos de seguridad, se imposibilita la capacidad de localizar algo con precisión de centímetros a nivel de usuario. Por ello, en el proyecto real habría que recurrir a un GPS diferencial, el cual es un sistema que determina la posición de un objeto mediante la correlación de la medida de un GPS normal, y la de un dispositivo con coordenadas GPS con error despreciable. En la simulación no se usa esta técnica ya que se parte de un entorno ideal y por tanto, sin errores aparentes en la medida.

[8] Fuente: www.wikipedia.org

3 Descripción del Software

3.1 Introduccion

No hay ingeniería sin abstracción. Para este proyecto se puede diseñar y controlar un drone desde cero o bien se pueden usar harramientas existentes y modificarlas hasta conseguir los resultados esperados. En este caso, donde la tarea no requiere de demasiada precisón, se tiene un cierto margen de maniobra como para permitirse el uso de herramientas que abstraen al ingeniero con un coste aceptable en pérdida en flexibilidad y entendimiento del entorno. Hacer incapié en esto es importante ya que a lo largo de este apartado se va a explicar solamente aquellos aspectos susceptibles de modificación para conseguir una correcta integración de todos los elementos, por tanto, se pasará por alto algunos archivos secundarios, de configuración, github ... dado que su explicación no aportaría un conocimiento crucial a la hora de comprender la funcionalidad del software. En dicho caso, la herramienta usada es ROS [1], cuyo logo se muestra en la figura 3.1[1].

Figura 3.1 Logo corporativo de ROS.

A groso modo, el software a desarrollar, se encarga de transformar coordenadas GPS dadas por el usuario, en coordenadas locales en el sistema de referencia del dron, asi como una evaluación continua de la autonomía, control de la posicion y comunicación con el usuario para evitar accidentes.

3.2 ¿Que es ROS?

ROS (Robot Operating System) es un meta-sistema operativo de código abierto para robots, el cual proporciona al usuario muchos de los servicios que se esperan de un sistema operativo común como; abstracción hardware, control de dispositivos a bajo nivel, comunicación entre procesos, manejo de paquetes, etc. De esta forma, ROS ofrece al desarrollador un marco de trabajo perfecto para descargar, organizar, escribir y ejecutar códigos que implementen funcionalidades específicas al robot.
En este caso, se parte de la versión Kinetic Kame, propia de Ubuntu 16.04 LTS, además de paquetes donde ya están desarrollados el dron y GPS, por tanto, solo hay que integrarlos de tal manera que se cumplan las exigencias propias del proyecto.

[1] Fuente: www.ros.org

A lo largo de la memoria se va a detallar solo los aspectos relevantes en el mismo, por tanto, todo aquello que no se explique a nivel de código (instalación, ejecución de archivos, comandos específicos ...) es porque esta detallado en la página oficial de ROS (http://wiki.ros.org/es).

3.2.1 Sistema de archivos

Al ser ROS un meta-sistema operativo cuenta con su propio sistema de archivos, los cuales son:

- Paquetes: Unidad de organización principal de software en ROS. Dispone de procesos ejecutables, bibliotecas y archivos de configuración. En esencia un paquete organiza toda aquella serie de archivos necesarios para el control del robot.
- Meta-Paquetes: Colecciones de paquetes destinados a un proceso común. Basicamente es una forma de abstraernos y conglomerar la actuaciones de varios paquetes bajo una misma tarea.
- Manifiestos de paquetes: Proporciona la información esencial de un paquete como el nombre, autor, descripción ...
- Repositorios: Colección de paquetes accesibles mediante internet
- Mensajes: Archivo que define la estructura de un mensaje en ROS
- Servicios: Archivo donde se detalla el tipo de petición o respuesta en una comunicación cliente-servidor en ROS

3.2.2 Red de computación

ROS basicamente trata de sincronizar diversas actuaciones a partir del contenido de los menajes propios del robot mediante el protocolo de comunicación peer-to peer (P2P). A continuación se detallan los elementos involucrados en este proceso:

- Nodo: Proceso ejecutado en un paquete. Dicho proceso se corresponde generalmente con una actuación del robot programada en python o c++. Normalmente el control de un robot se basa en la ejecución de varios nodos comunicados entre si.
- Master: Proceso principal encargado de organizar la ejecución de los nodos. Para ello usa servicios mediante los cuales los nodos se comunican entre si.
- Servidor de parámetros: Pila que contiene todos los parámetros accesibles por los nodos.
- Mensajes: Estructura de datos definida en los archivos de tipo de mensajes. Pueden ser de varios tipos como enteros, booleanos ...
- Temas (Topics): Los mensajes se enrutan mediante un sistema de publicación/subscripción bajo el nombre de un tema. A groso modo, el tema dota de significado semántico a un mensaje que simplemente es una estructura de datos.
- Bolsas (Bags): Archivos encargados de guardar y reproducir mensajes enviados
- Servicios: Modelo de mensaje petición-respuesta definido en el tipo de archivo servicio. Se usa cuando un nodo necesita una respuesta concreta para una publicación determinada.

Se puede concluir que un paquete o meta-paquete ROS es simplemente una red de comunicación entre nodos, los cuales, desempeñan principalmente dos funciones; suscribirse a un tópico, o publicarlo. Si se mira esta afirmación con óptica, realmente un sensor envía las lecturas al microcontrolador para control o supervisión de un proceso, por tanto, se podría modelar como un nodo que publica un cierto tópico. De la misma forma, un actuador genera una salida en función de ciertas especificaciones de control, siendo la información sensorial una de ellas, por ello, se podría modelar como un nodo que se suscribe a uno o varios tópicos y así realizar la correcta actuación.

En la figura 3.2[2] se representa un ejemplo simple de comunicación entre dos nodos. Un nodo publica un tópico llamado /talker, el cual es un tipo de mensaje de tipo std_msgs/String, mientras el otro nodo se encarga de recibirlo (subcribir) para hacer una actuación en base a dicho mensaje, que en este caso podría ser imprimir por pantalla lo que recibe por ejemplo.

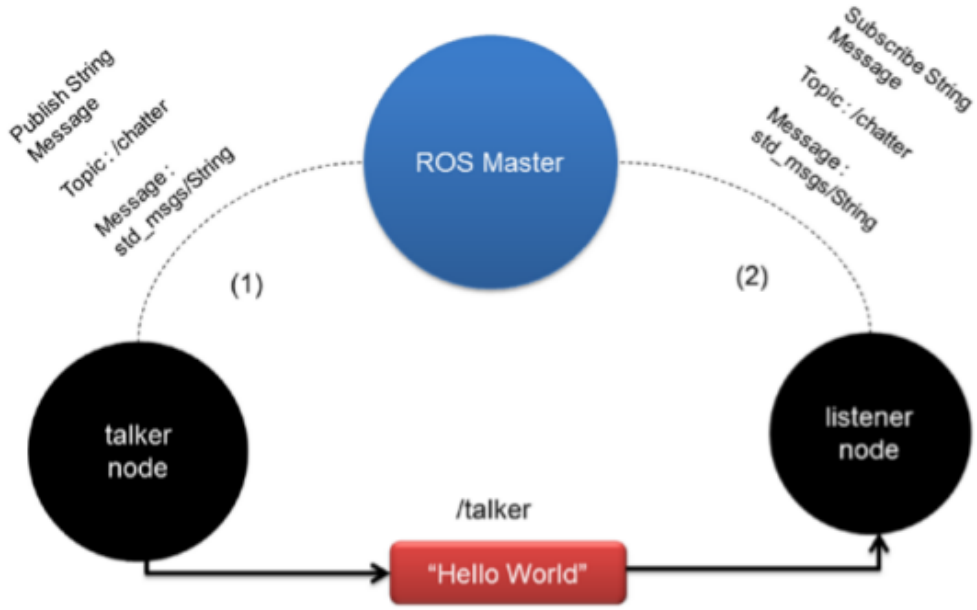


Figura 3.2 Esquema de comunicación entre nodos.

3.3 Tutorial ROS

Con el fin de que el lector entienda todo lo que se expone, se ha decidido hacer una pequeña introducción práctica a modo de complemento a la teoría explicada. Como se dijo anteriormente en la página oficial de ROS (http://wiki.ros.org/es) está todo lo que se va a detallar a continuación, por tanto, solo se detallan aquellos elementos que se han usado unicamente en la ejecución del proyecto, ya que hay otros destinados a depurar entre otras muchas cosas, y por brevedad en la memoria no se ahonda en ellos.
Se comienza este "mini" tutorial dando por hecho que se ha completado la descarga e instalación de ROS en su versión Kinetic Kame, además de la correcta configuración del sistema para poder compilar y ejecutar códigos de este framework. De este modo se procede a ejecutar varios nodos de un paquete llamdo "Turtlesim", el cual consta de la visualización de una tortuga y diversos tópicos y nodos que definen su movimiento, entre otros detalles menores.
En primer lugar, para ejecutar cualquier archivo en ROS es necesario activar el nodo Máster:

```
mrhyde@francisco:~/catkin_ws/$ roscore
```

Después se procede a ejecutar, en otro teminal, el nodo de visualización de la tortuga mediante el comando rosrun, el cual le sigue el nombre del paquete y el nodo a ejecutar:

```
mrhyde@francisco:~/catkin_ws/$ rosrun turtlesim turtlesim_node
```

[2] Fuente: www.ros.org

Trás la ejecución de estos comandos se obtiene lo mostrado en la figura 3.3[3]:

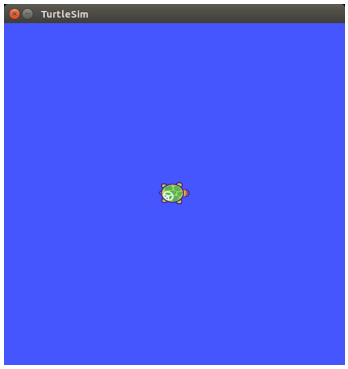


Figura 3.3 Visualización tortuga ROS demo.

Como se explicó anteriormente, los tópicos son los encargados de enviar o recibir cierto tipos de mensajes que contienen información del robot en cuestión. Mediante el comando rostopic list se accede a ellos:

```
mrhyde@francisco:~/catkin_ws/$ rostopic list
/rosout
/rosout_agg
/turtle1/cmd_vel
/turtle1/color_sensor
/turtle1/pose
```

A priori no se sabe la función de cada tópico, aunque a veces se puede intuir por su nombre. Para ello se usa el comando rostopic type el cual nos dice el tipo de mensaje que envía o recibe, y de esa forma si se busca dicho mensaje en la API de ROS (http://docs.ros.org/kinetic/api/) se obtiene una información clara y concisa de la función de dicho tópico, así como los parámetros que lo definen.

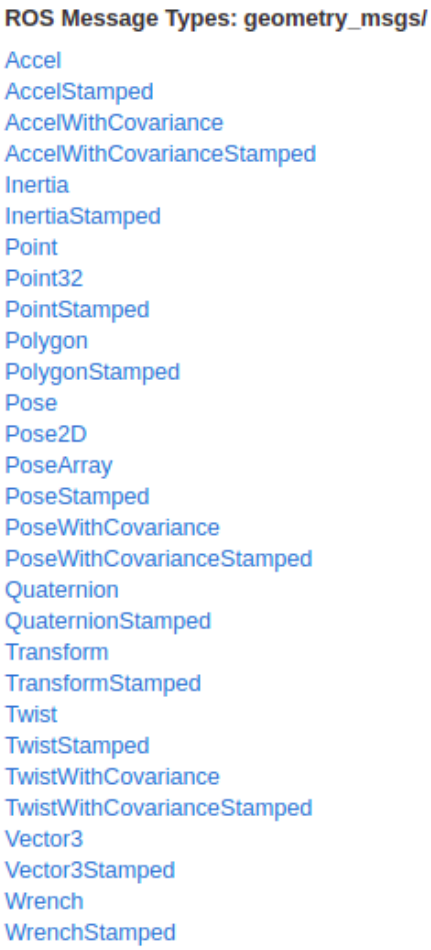
```
mrhyde@francisco:~/catkin_ws/$ rostopic type /turtle1/cmd_vel
geometry_msgs/Twist
```

Al buscar "geometry_msgs" en la API se obtiene la lista de subtipos de mensajes de tipo geometry_msgs (http://wiki.ros.org/geometry_msgs), entre los cuales se encuentra Twist (figura 3.4)[4].

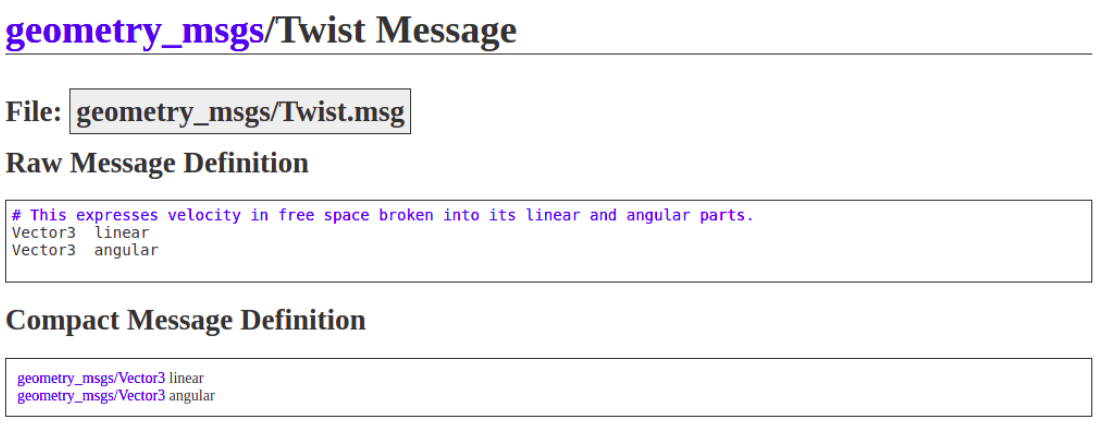
Figura 3.4 geometry_msgs/ en la API.

[3] Fuente: Simulación propia
[4] Fuente: www.ros.org

Del mismo modo que antes, se obtiene la siguiente lista, donde al elegir el mensaje Twist se especifica sus funciones y parámetros (figura 3.5)[5]:



(a)



(b)

Figura 3.5 a) Tipos de mensage geometry_msgs b) Información mensaje Twist.

[5] a) Fuente: www.ros.org
b) Fuente: www.ros.org

Esta metodología sirve generalmente para todo tipo de mensajes, y por tanto, una vez acotado como identificarlos, es necesario detallar como publicarlo o suscribirse. En este caso se va ha realizar mediante el terminal por simplicidad, aunque también se podría hacer en un archivo .py o .cpp, aspecto que se explicara después de acotar dichas funciones por el terminal.
Para hacer que la tortuga se mueva basta con localizar el tópico necesario, el cual es /turtle1/cmd_vel, y publicarlo mediante el comando rostopic pub:

```
mrhyde@francisco:~/catkin_ws/$ rostopic pub /turtle1/cmd_vel geometry_
    msgs/Twist "linear:
  x: 1.0
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 0.0"
```

Ejecutando este comando se consigue que se publique una velocidad lineal en el eje x, lo que se traduce a un movimiento de 1 cm en dicho eje. Si se hubiera publicado algún valor positivo en angular.z, la tortuga giraría en sentido antihorario. Del mismo modo que se puede publicar, se puede imprimir por pantalla el valor de un cierto tópico mediante el comando rostopic echo. A continuación se muestra por pantalla el valor del topico /turtle1/pose el cual muestra la posición actual:

```
mrhyde@francisco:~/catkin_ws/$ rostopic echo /turtle1/pose
---
x: 6.98779678345
y: 6.95192146301
theta: 0.772814691067
linear_velocity: 0.0
angular_velocity: 0.0
---
```

De la misma forma se puede acceder a los servicios:

```
mrhyde@francisco:~/catkin_ws/$ rosservice list
/clear
/kill
/reset
/rosout/get_loggers
/rosout/set_logger_level
/spawn
/turtle1/set_pen
/turtle1/teleport_absolute
/turtle1/teleport_relative
/turtlesim/get_loggers
/turtlesim/set_logger_level
```

Nodos:

```
mrhyde@francisco:~/catkin_ws/$ rosnode list
/rosout
/turtlesim
```

Parámetros:

```
mrhyde@francisco:~/catkin_ws/$ rosparam list
/background_b
/background_g
/background_r
/rosdistro
/roslaunch/uris/host_localhost__42051
/rosversion
/run_id
```

Con estos comandos se tiene una buena base para entender la semántica de un paquete ROS, ya que si se puede identificar, acceder y modificar cualquier parámetro funcional de un paquete, es fácil trabajar con el. A continuación se muestra los pasos básicos para crear un script, en este caso en python, que controle el movimiento de la tortuga definido por el tópico /turtle1/cmd_vel:

- 1.- En primer lugar, para publicar o subscribirse a un determinado tipo de mensaje hace falta importar su librería, junto con la del lenguaje en cuestión (rospy). Para ello es necesario saber el tipo de mensaje, que con rostopic type se obtiene facilmente "geometry_msgs/Twist". Se tendría por tanto:

```
**********Importar librerías**********
import rospy
import geometry_msgs.msg from Twist
```

- 2.- En segundo lugar es necesario acotar las funciones de publicación/suscripción de tópicos:

```
**********Funcion para publicar**********
def muevo_tortuga()
  #Defino nodo inicio
  rospy.init_node('muevo_tortuga', anonymous=False)
  #Defino publicador
  pub=rospy.Publisher('turtle1/cmd_vel',Twist, queue_size=10)
  #Pido velocidad lineal en x
  print("Introduce velocidad en x:")
  valor=raw_input()
  #Defino tipo de mensaje
  vel=Twist()
  #Doy valor
  vel.linear.x=valor
  #Publico
  pub.publish(vel)
```

Esta parte es sencilla, excepto por la forma en la que se da valor a la velocidad lineal en x. ¿Porqué se pone "vel.linear.x"?. Sencillo, para ello solo hay que usar el comando rosmsg show en adición al tipo de mensaje:

```
mrhyde@francisco:~/catkin_ws/$ rosmsg show geometry_msgs/Twist
geometry_msgs/Vector3 linear
  float64 x
  float64 y
  float64 z
geometry_msgs/Vector3 angular
  float64 x
  float64 y
  float64 z
```

Para acceder a un parámetro de un determinado mensaje solo hay que ir escalando por cabeceras, es decir, si se quiere publicar un valor de velocidad angular en el eje y, bastaría con poner:

```
vel.angular.y=valor
```

- 3.- Por último habría que instanciar dicha función en el "main":

```
if __name__==’__main__’:
    try:
        muevo_tortuga()
    except rospy.ROSInterruptException:
        pass
```

De la misma forma que se ha realizado un nodo que publique la velocidad, se puede hacer otro que se suscriba al tópico de la posición definido por el tópico /turtle1/pose:

- 1.- Se procede de igual forma que en el ejemplo anterior. En este caso el mensaje es de tipo "turtlesim/Pose". Se tendría por tanto:

```
**********Importar librerías**********
import rospy
import turtlesim.msg from Pose
```

- 2.- En segundo lugar se detalla la función callback:

```
**********Funcion para suscribirse**********
def callback(pose)
  rospy.loginfo("Posicion XY=[%f, %f]",pose.x,pose.y)

def print_pose()
  #Defino el nodo de inicio
  rospy.init_node(’print_pose’, anonymous=True)
  #Me suscribo al topico
  rospy.Subscriber(’/turtle1/pose’,Pose, callback)
  #Espero a que se pare el nodo
  rospy.spin()
```

- 3.- Por último habría que instanciar dicha función en el "main":

```
if __name__==’__main__’:
    try:
        print_pose()
    except rospy.ROSInterruptException:
        pass
```

De esta forma se crean los script muevo_trotuga.py y posicion_tortuga.py respectivamente, los cuales serían nodos dentro del paquete correspondiente, el cual ha sido creado a priori siguiendo la documentación oficial disponible. Para ejecutarlos se procede de la misma forma que antes:
Ejecuto el nodo máster

```
mrhyde@francisco:~/catkin_ws/$ roscore
```

Ejecuto el nodo muevo_tortuga.py dentro del paquete tfg

```
mrhyde@francisco:~/catkin_ws/$ rosrun tfg muevo_tortuga.py
Introduce velocidad en x:
```

Dado que no se ha puesto condición de espera, el print se ejecutará constantemente, echo que se pasa por alto ya que el objetivo del ejemplo no es la funcionalidad, sino como el acotar conceptos prácticos de ROS. De la misma forma ejecuto el archivo print_pose.py dentro del mismo paquete.

```
mrhyde@francisco:~/catkin_ws/$ rosrun tfg print_pose.py
Posicion:[5.544 5.544]
```

Llegados a este punto se intuye la limitación de lanzar nodos de forma manual. Si se tiene la necesidad de controlar un sistema gobernado por varios nodos, los cuales procesan información de otros recursivamente, es evidente que habría que detallar una estrategia de lanzamiento de nodos, ya que sería absurdo lanzar un nodo que controlase algo en base a la salida de un nodo que no se ha ejecutado aún.
La solución a este problema se llama roslaunch, y es un comando para lanzar archivos .launch situados en la carpeta launch dentro del mismo directorio de trabajo. La sintaxis es muy simple:

```
<launch>
  <node name="move_node" pkg="tfg" type="muevo_tortuga.py"
output="screen"/>
  <node name="callback_node" pkg="tfg" type="posicion_tortuga.py"
output="screen"/>
</launch>
```

De esta forma se lanzan simultáneamente los dos nodos en cuestión. Este detalle es muy importante durante el proyecto ya que se tiene una infinidad de nodos tanto para dron como para GPS. Los archivos .launch tienen muchas mas opciones de configuración de la que se muestra en este ejemplo, de echo algunas de ellas se mostrará en futuros apartados donde haya la necesidad de realizar una ejecución más completa.

3.4 Entorno de simulacion: Gazebo y Rviz

3.4.1 Gazebo

Gazebo, cuyo logo se muestra en la figura 3.6[6], es un simulador de entornos 3D que posibilita evaluar el comportamiento de un robot en un mundo virtual. Permite, entre muchas otras opciones, diseñar robots de forma personalizada, crear mundos virtuales usando sencillas herramientas CAD, importar modelos ya creados, comprobar algoritmos de control, así como modelos fisicos entre una infinidad de opciones.
En este proyecto se usa Gazebo para la simulacion de la estructura del drone unicamente ya que el entorno de trabajo, en este caso una parcela, es invariable y plano, y por tanto no es relevamte la morfología del terreno. Por este motivo, se ejecuta un mundo vacío, donde solo hay un terreno plano donde se posa el drone cuando es necesario, ya que si no hubiera suelo, la simulación obtenida sería la de un drone cayendo al vacío.

Figura 3.6 Logo de Gazebo.

3.4.2 Rviz

Rviz, cuyo logo se muestra en la figura 3.7[7] es un visualizador de datos ejecutados en un nodo o archivo .launch, comunmente se visualiza los diversos topicos publicados por nodos que procesan la lectura de sensores. También se pueden añadir elementos simulados en Gazebo, como el drone en este caso.
Según estas consideraciones es fácil entender que si se dispone de un paquete, que simule la física y estructura de un drone, ademas de un mapa GPS centrado en cierto punto, es facil conectar ambos paquetes en una visualización en Rviz. Solo habria que añadir los tópicos que juegen un papel relevante en dicha tarea.

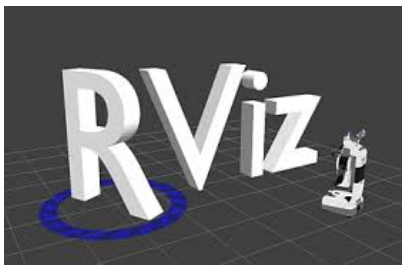


Figura 3.7 Logo de Rviz.

[6] Fuente: www.ros.org
[7] Fuente: www.ros.org

3.5 Entorno de trabajo

En este apartado se explica que paquetes se han usado, además de detallar sus elementos mas importantes, ya que en la siguiente sección de la memoria se comienza a explicar como se ha llevado a cabo la integración de ambos paquetes, así como los elementos que han sufrido modificación. Destacar que no se añade nada de código ya que, como se explicó anteriormente, se adjuntan en el apéndice.

3.5.1 Rviz_satellite

Este paquete se encarga de 'traducirúna URL de mosaico (URI objeto) a un tipo de dato entendible por Rviz, como es el caso de ÁerialMapDisplay', con el fin de obtener una nube de datos que el propio programa los transforma en un mapa 2D de resolución variable según las exigencias del usuario. El árbol de directorio del paquete es el mostrado en la figura 3.8[8].

Figura 3.8 Arbol directorio rviz_satellite.

Los archivos que dotan de funcionalidad al paquete se ubican en la carpeta 'launch'y 'src', los archivos omitidos son solamente librerias y demás archivos de configuración propios de cada paquete y que vienen detallados en sus respectivas documentaciones. A continuación se detalla la función de cada elemento ubicado en las carpetas mencionadas:

- demo.gps: En este archivo se define pricipalmente la posición inicial del GPS, a modo de sistema de referencia. Tambien se puede modificar el frame y la covarianza de la posición a modo de ruido del sensor.
- demo.rviz: En este fichero se define la configuración del visualizador; frames, ejes de referencias, puntos asociados a topicos... Como su programación no es trivial, se ha optado por partir de una instancia vacía en Rviz y se ha ido añadiendo elementos de visualización, con el fin de guardar la configuración en el formato dado y así poder usarlo a posteriori.
- demo.launch: En este apartado se establece la configuración de ejecución de los siguientes nodos:
  - fake_gps_fix: Se encarga de delimitar la posición inicial del GPS que recibe como argumentos.

[8] Fuente: Elaboración propia

- rviz_satellite: Ejecuta Rviz con la configuracion del archivo demo.rviz mencionada anteriormente.
- static_tf_fake: Transforma las coordenadas de translación y orientación de los frames que recibe como argumentos.

Una vez detallado la estructura de archivos del paquete, así como la importancia y utilidad de cada uno de sus nodos, se pasa a detallar que tópicos se van a usar de este paquete:

- /gps/fix: Indica la posición del eje de referencia en coordenadas GPS.
- /move_base_simple/goal: Indica la posición de un punto relativo al eje de referencia en coordenadas absolutas.

A continuación se procede a mostrar el funcionamiento de este paquete dentro del proyecto. Primero hay que ejecutar el master:

```
mrhyde@francisco:~/catikin_ws/$ roscore
```

Ejecutamos el paquete:

```
mrhyde@francisco:~/catkin_ws/$ roslaunch rviz_satellite demo.launch
```

En la figura 3.9 [9], se pueden ver los resultados de ejecutar los comandos anteriores:

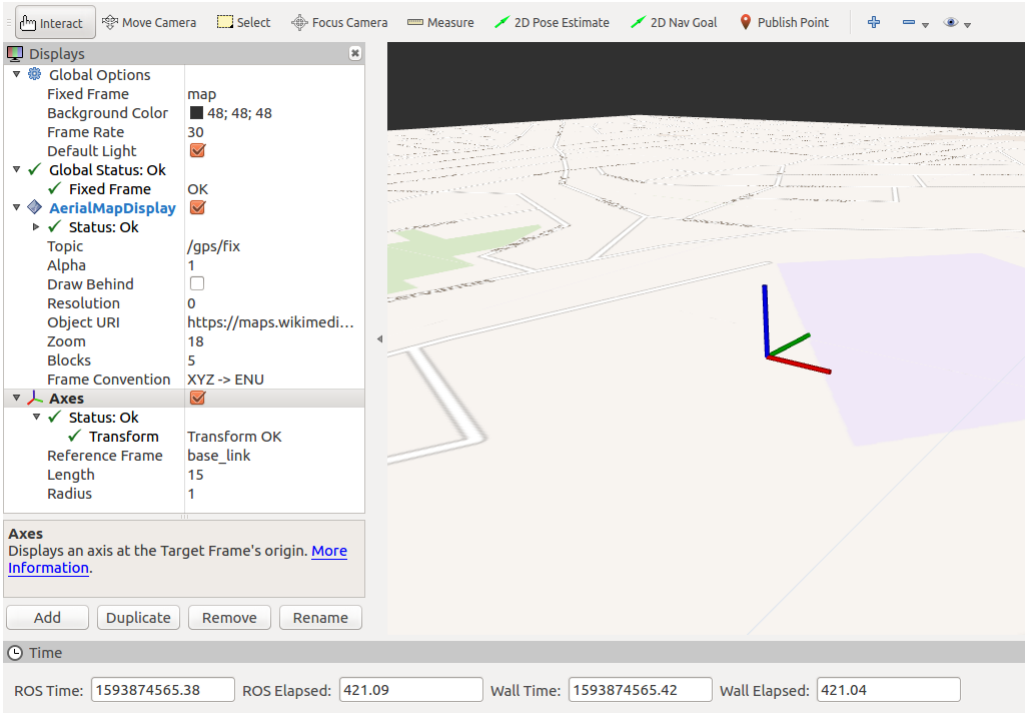


Figura 3.9 rviz_satellite demo punto inicial.

[9] Fuente: Elaboración propia

A continuación se mueve la base a otro punto en coordenadas locales relativo al punto inicial (figura 3.10)[10]:

```
mrhyde@francisco:~/catikin_ws/$ rostopic pub /move_base_simple/goal
    geomey_msgs/PoseStamped "header:
  seq: 0
  stamp:
    secs: 0
    nsecs: 0
  frame_id: 'map'
pose:
  position:
    x: 10.0
    y: 2.0
    z: 8.0
  orientation:
    x: 0.0
    y: 0.0
    z: 0.0
    w: 0.0"
```

Con todo lo mostrado anteriormente es facil acotar la función de este paquete. El punto de referencia de GPS se corresponde con el spawn del dron, y los puntos objetivos del usuario se marcan en el mapa publicando el anterior tópico con dicha posición, asi de esa forma si se superpone el frame del dron al mapa debería quedarse estático en dichos puntos y así se puede comprobar que el dron llega a los puntos objetivos correctamente.

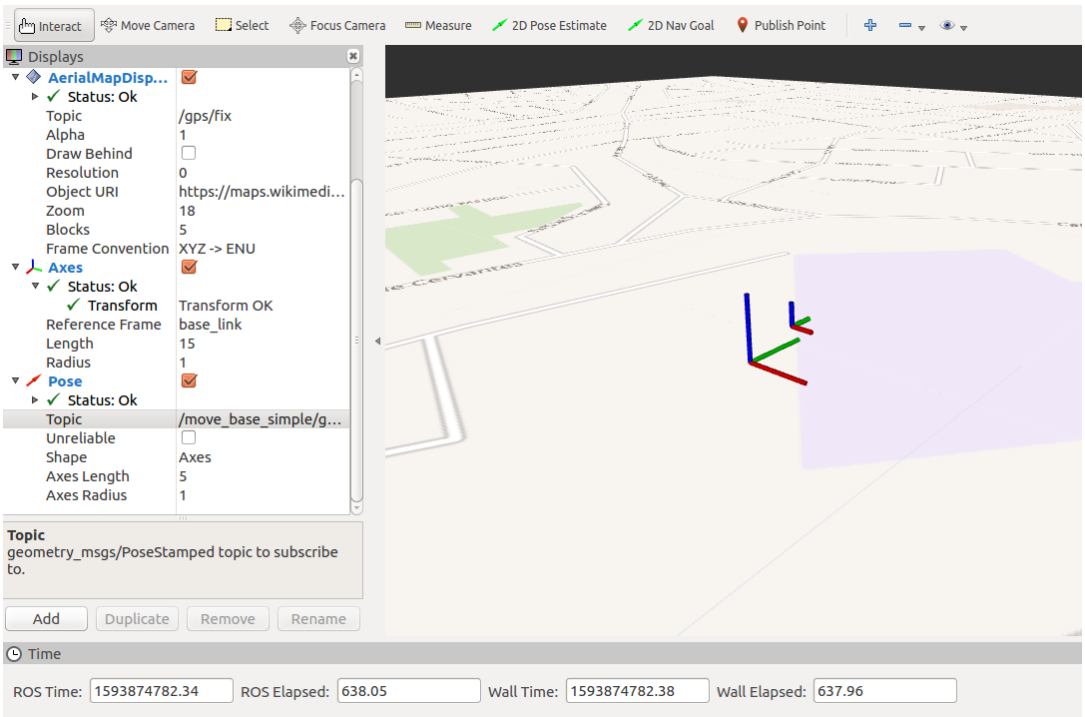


Figura 3.10 rviz_satellite demo translación.

[10]Fuente: Elaboración propia

3.5.2 Hector_quadrotor_tutorial

Es un meta-paquete donde principalmente se intenta simular el comportamiento físico de un dron en entornos predefinidos en el mismo paquete. Además cuenta con numerosos nodos que modelan tanto la lectura de diversos sensores; IMU, giróscopo, acelerómetro, altímetro, leader ... como la posterior fusión sensorial y procesado de los mismos, obteniendo asi numerosas mediciones para llevar a cabo un fiable control del dron.
En este proyecto se usa principalmente el modelo físico del dron y los tópicos que definen su comportamiento cinemático, ya que el entorno de trabajo, como es el caso de una parcela agrícola, es plano y con apenas obstáculos. Por esa razón, solo se acude a los entornos predefinidos en el meta-paquete para probar el algoritmo de evitación de obstáculos, el cual se aleja de los conocidos, ya que se ha optado por uno donde simplemente detiene el dron al detectar un obstáculo a una cierta distancia de seguridad, informando de la posición del obstáculo al usuario y pidiendo una nueva posición objetivo.
De igual forma con el apartado anterior, en la figura 3.11[11] se procede a detallar brevemente el contenido de las carpetas principales, así como aquellas que se han modificado durante el desarrollo del proyecto: De forma breve y concisa, la función de estos subpaquetes es la siguiente:

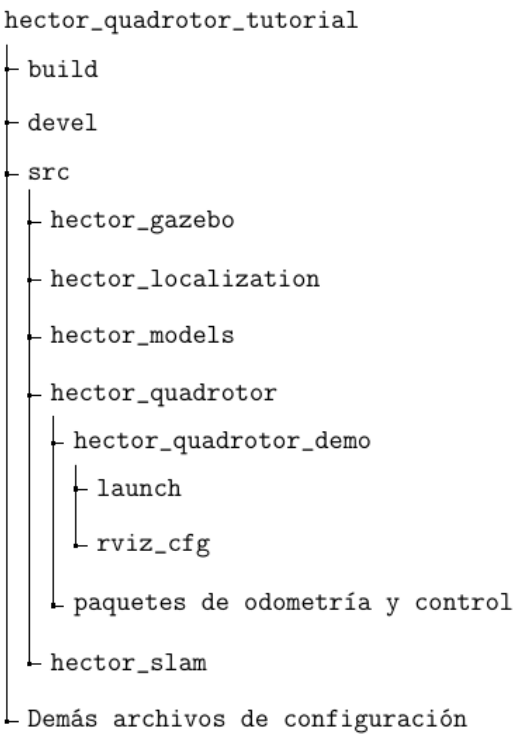


Figura 3.11 Arbol directorio hector_quadrotor_demo.

- hector_gazebo: Contiene la información y modelado de los sensores, plugins y mundos en Gazebo, por tanto, no se ha modificado ningún fichero, ya que solo usamos Gazebo para poder añadir el modelo del drone a Rviz y así superponerlo en el mapa GPS.

[11]Fuente: Elaboración propia

- hector_model: Este paquete proporciona los archivos URDF a hector_gazebo. Importante destacar que los archivos URDF son ficheros .xacro donde se definen, mediante etiquetas, los diversos parametros de los elementos que vamos a visualizar en Gazebo, como su longitud, masa, inercia ... Al igual que el punto anterior, no se modifica nada, solo se usa cuando instanciamos el modelo del drone en el .launch correspondiente.
- hector_localization: Este paquete se encarga del procesado y fusión de los datos de los sensores, además de la publicación, mediante topicos, de dichos resultados. Como en el caso anterior no se ha modificado nada, solo se ha usado la información de la posición del drone.
- hector_slam: Paquete que realiza SLAM. No tiene sentido su uso en este proyecto ya que el mapa lo proporciona el usuario mediante coordenadas GPS, y el algoritmo de evitación de obstáculos se encarga del resto.
- hector_quadrotor: Este es el paquete principal, en el se incluye toda la lógica de control del drone, además del desarrollo de todos los sensores, actuadores, frames ... que se instancian en los dos paquetes primeros para su posterior visualización en Gazebo. Solo se ha modificado los archivos .launch y .rviz de la subcarpeta hector_quadrotor_demo, ya que son los encargados de; inicializar el mundo en gazebo y la configuración de visialización en rviz.

Una vez detallados los paquetes que conforman el dron, es importante delimitar que tópicos se han usado, ya que por el elevado número no se considera óptimo definirlos todos si no van a ser de utilidad en el proyecto.

- /action/pose/goal: Indica la posición objetivo del dron dada por el usuario en coordenadas absolutas al sistema de referencia.
- /ground_truth_to_tf/pose: Indica la posición actual de dron. Es el resultado de la fusión sensorial de todas las demás lecturas de sensores.

Como se ha dicho anteriormente, para este paquete hay un gran número de tópicos que arrojan información útil del estado del dron, además de la posibilidad de modificar parámetros de los controladores del dron entre otras muchas cosas. Para la apliación que se trata es suficiente tener la posibilidad de enviar el dron donde se requiera, además de saber su ubicación.
A continuación se procede a mostrar el funcionamiento de este paquete, dónde primero hay que ejecutar el master:

```
mrhyde@francisco:~/catikin_ws/$ roscore
```

Ejecutamos el paquete:

```
mrhyde@francisco:~/catkin_ws/$ roslaunch hector_quadrotor_demo outdoor_
    flight_gazebo.launch
```

Tras ejecutar los comandos anteriores, se obtiene lo mostrado en las figuras 3.12[12] y 3.13[13]:

[12]Fuente: Simulación propia
[13]Fuente: Simulación propia

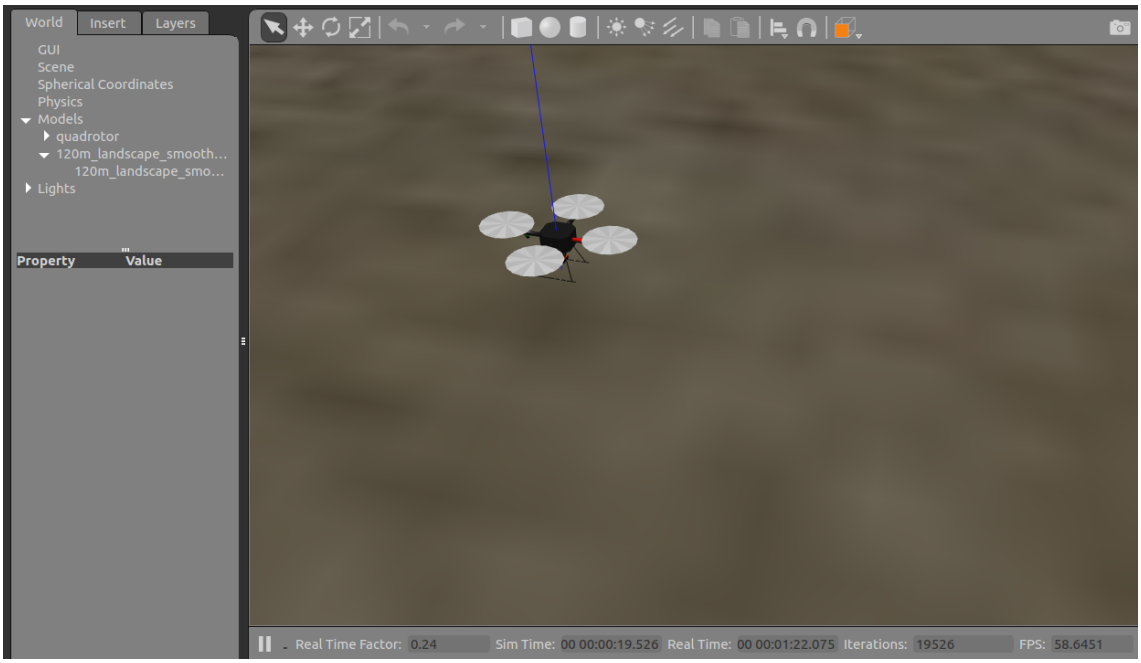


Figura 3.12 Modelado dron.

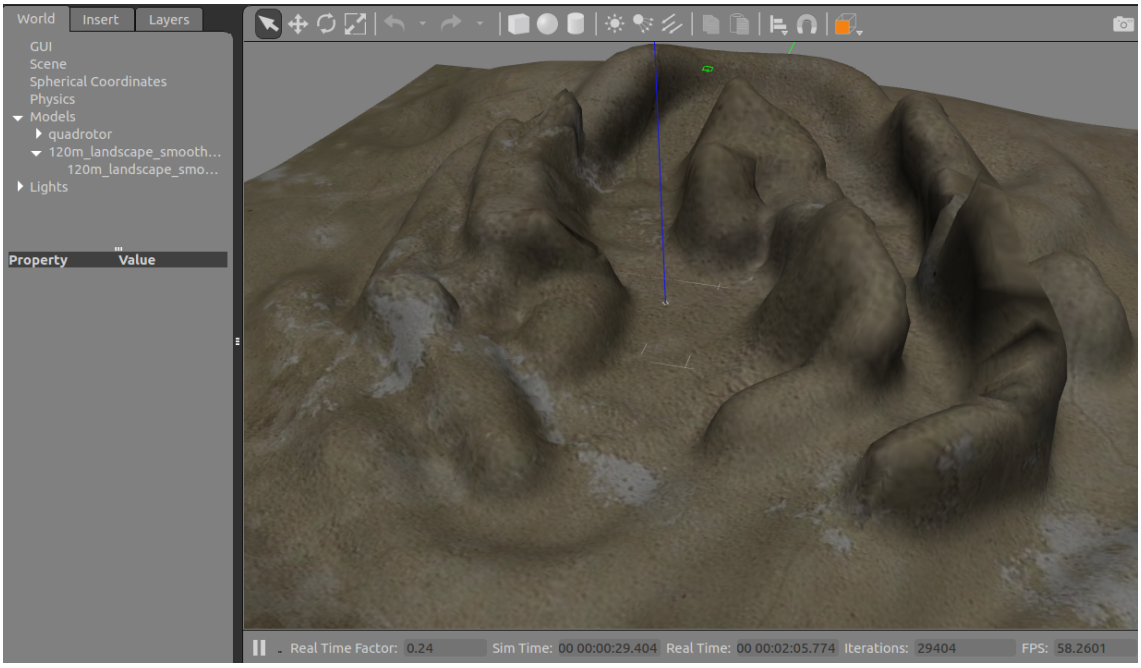


Figura 3.13 Panorámica del entorno.

Del mismo modo que el paquete anterior, es facil acotar la funcionalidad de este paquete dentro del proyecto, ya que solo se usa el modelo físico del dron para superponerlo al mapa GPS dejando a un lado el entorno ya que el espacio de trabajo es plano y a priori sin obstáculos. De forma breve y concisa, de este paquete solo se usa el modelo de dron en un mundo "vacío", lo cual se corresponde simpemente con un plano XY a modo de suelo, para poder instanciarlo en rviz y que no simule un drone cayendo al vacío.

3.5.3 geo.py

En este subapartado se va a detallar el script de python encargado de hacer la conversión entre coordenadas GPS y coordenadas cartesianas. Antes de eso, se va a hacer una breve introducción del tipo de coordenadas así como algunos de los fundamentos que las definen:

- Sistema de coordenadas geográficas o geodésicas: Permite definir la ubicación terrestre mediante un conjunto de números, letras o símbolos. Su sistema de referencia se encuentra en el centro geométrico de un elipsoide de revolución como modelo matemático de la tierra. Normalmente los parámetros usados son los descritos en la figura 3.14[14]:

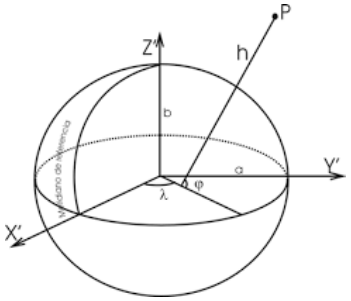


Figura 3.14 Coordenadas geodésicas.

  - Latitud geodésica (φ): Ángulo medido en el plano meridiano que forma la normal al elipsoide en P y el plano del ecuador.
  - Longitud geodésica (λ): Ángulo medido en el plano del ecuador, en sentido horario, que forma el plano meridiano que contiene P y el plano meridiano de Greenwich.
  - Altura elipsoidal (h): Distancia entre P y el elipsoide, medida a lo largo de la normal al elipsoide que pasa por dicho punto.

- Sistemas de coordenadas ECEF: Representa puntos en coordenadas cartesianas con referencia el centro de la tierra mediante transformaciones geométricas en base a la figura 3.15[15]:

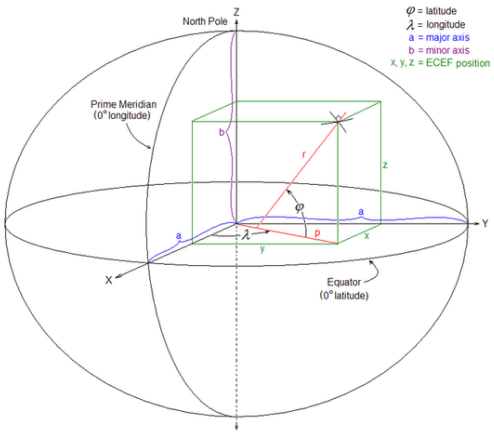


Figura 3.15 Coordenadas ECEF.

[14]Fuente: https://www.ign.es/web/ign/portal/gds-teoria-geodesia
[15]Fuente: www.wikipedia.org

Proyectando el punto en cuestión, en cada plano XYZ con los parámetros dados, se obtiene la posición en cartesianas de dicho punto:

$$X = (N(\lambda) + h) * \cos\lambda * \cos\varphi \tag{3.1}$$

$$Y = (N(\lambda) + h) * \cos\lambda * \sin\varphi \tag{3.2}$$

$$Z = (\frac{b^2}{a^2} * N(\lambda) + h) * \sin\lambda \tag{3.3}$$

siendo N:

$$N(\lambda) = \frac{a^2}{\sqrt{a^2\ cos^2\lambda + b^2\ sin^2\lambda}} = \frac{a}{\sqrt{1 - e^2\ sin^2\varphi}} \tag{3.4}$$

- Sistemas de coordenadas ENU (figura 3.16 a))[16]: Su nombre es un acrónimo de East, North y Up, los cuales son los elementos que definen la posición terrestre de un objeto, y equivalen a los ejes xyz si se considera las siguientes transformaciones plamadas en la figura 3.16 b)[17]:
  - El eje x apunta hacia el norte elipsoidal (meridiano).
  - El eje y apunta hacia el este y forma así un sistema de mano izquierda.
  - El eje z apunta en la misma dirección que el cenit, perpendicular al plano xy.

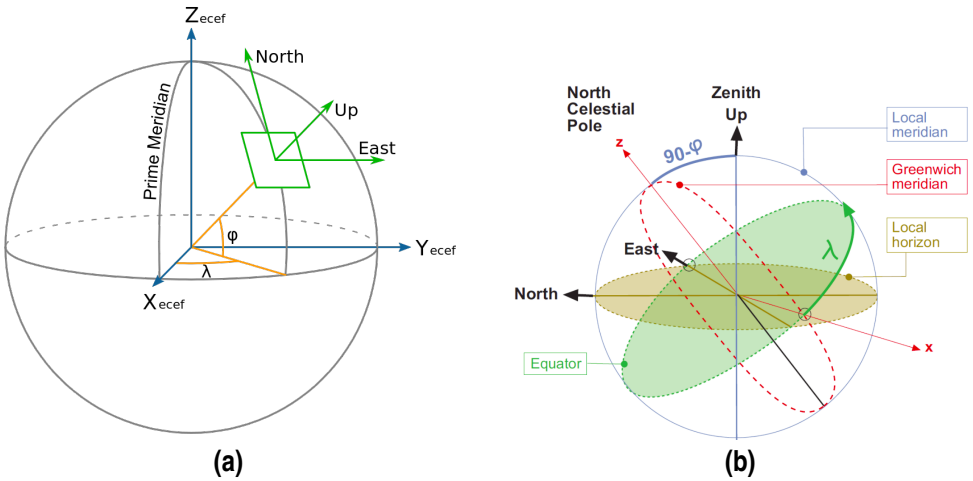


Figura 3.16 a) Coordenadas ENU b) Transformación de ECEF a ENU .

Con las anteriores consideraciones plasmadas en ambas figuras, se procede a detallar las transformaciones necesarias:

$$\begin{bmatrix} E \\ N \\ U \end{bmatrix} = R_x[\frac{\pi}{2} - \lambda]R_z[\frac{\pi}{2} + \lambda] \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta z \end{bmatrix} \tag{3.5}$$

[16] Fuente: www.wikipedia.org

[17] Fuente: https://es.qwe.wiki/wiki/Geographic$_c$oordinate$_c$onversion

Donde la matriz de rotación resultante de $R_x[\frac{\pi}{2}-\lambda]R_z[\frac{\pi}{2}+\lambda]$, seria:

$$\begin{bmatrix} -\sin\lambda & \cos\varphi & 0 \\ -\cos\lambda\sin\varphi & -\sin\lambda\sin\varphi & \cos\varphi \\ \cos\lambda\cos\varphi & \sin\lambda\cos\varphi & \sin\varphi \end{bmatrix} \tag{3.6}$$

Además es importante destacar que Δx, Δy y Δz, son los incrementos de posición teniendo como referencia una latitud, longitud y alturas determinadas, las cuales se pueden expresar en los planos XYZ proyectando como se explicó en el anterior punto.

Tras hacer una breve introducción conceptual de los tipos de coordenadas queda claro que el GPS entiende coordenadas geodésicas, las cuales habría que pasar a ECEF y posteriormente transformarlas a ENU. A continuación se procede a detallar la estructura en pseudocódigo del script geo.py:

Código 3.1 geo.py.

```
{Definicion de constantes teóricas}
a = 6378137
b = 6356752.3142
f = (a - b) / a
e_sq = f * (2-f)

def geodetic_to_ecef(Lat, Long, Alt)
  {Caracterizo los parametros del modelo}
  Lambda=Lat en radianes
  Phi=Long en radianes
  N=1/(sqrt(1-e_sq*sin(Lambda)^2))

  {Se obtiene XYZ proyectando con Lambda y Phi}
  x=(Alt+N)*cos(Lambda)*cos(phi)
  y=(Alt+N)*cos(Lambda)*sin(phi)
  z=(Alt+(1-e_sq)*N)*sin(Lambda)

  return x y z

def ecef_to_enu(x, y, z, Lat_ref, Long_ref, Alt_ref)
  {Caracterizo los parametros del modelo}
  Lambda=Lat_ref en radianes
  Phi=Long_ref en radianes
  N=1/(sqrt(1-e_sq*sin(Lambda)^2))

  {Proyectando como antes se obtiene x0 y0 z0}
  x0=(Alt_ref+N)*cos(Lambda)*cos(phi)
  y0=(Alt_ref+N)*cos(Lambda)*sin(phi)
  z0=(Alt_ref+(1-e_sq)*N)*sin(Lambda)

  {Incrementos de la posición}
  xd=x-x0
  yd=y-y0
  zd=z-z0
```

```
  {Transforma el vector incremento posición a ENU mediantela matriz de
     rotación}
  xEast=-sin(phi)*xd+cos(phi)*yd
  yNorth=-cos(phi)*sin(Lambda)*xd-sin(Lambda)*sin(phi)*yd+cos(Lambda)*zd
  Zup=cos(Lambda)*cos(phi)*xd+cos(Lambda)*sin(phi)*yd+sin(Lambda)*zd

  return xEast, yNorth y Zup

def geodetic_to_enu(lat, long, alt, Lat_ref, Long_ref, Alt_ref)
  x, y, z=geodetic_to_ecef(x, y, z, Lat_ref, Long_ref, Alt_ref)
  return ecef_to_enu(x, y, z, Lat_ref, Long_ref, Alt_ref)

{El resto del codigo está destinado a tratar el problema de puntos muy
    próximos <1e-4}
def are_close(a, b):
  return abs(a-b) < 1e-4

    latLA = 34.00000048
    lonLA = -117.3335693
    hLA = 251.702

    x0, y0, z0 = geodetic_to_ecef(latLA, lonLA, hLA)
    x = x0 + 1
    y = y0
    z = z0
    xEast, yNorth, zUp = ecef_to_enu(x, y, z, latLA, lonLA, hLA)
    assert are_close(0.88834836, xEast)
    assert are_close(0.25676467, yNorth)
    assert are_close(-0.38066927, zUp)

    x = x0
    y = y0 + 1
    z = z0
    xEast, yNorth, zUp = ecef_to_enu(x, y, z, latLA, lonLA, hLA)
    assert are_close(-0.45917011, xEast)
    assert are_close(0.49675810, yNorth)
    assert are_close(-0.73647416, zUp)

    x = x0
    y = y0
    z = z0 + 1
    xEast, yNorth, zUp = ecef_to_enu(x, y, z, latLA, lonLA, hLA)
    assert are_close(0.00000000, xEast)
    assert are_close(0.82903757, yNorth)
    assert are_close(0.55919291, zUp)
```

4 Desarrollo del Software

4.1 Introduccion

En este capítulo se detalla los pasos a seguir en la elaboración del proyecto en su totalidad, teniendo como base los fundamentos teóricos y prácticos explicados en el anterior capítulo.

4.2 Configuracion rviz_satellite

Si se recuerda la explicación de la sección 3.5.1, se partía del arbol directorio principal (figura 4.1)[1], donde se detallaba la estructura del paquete, además de los archivos de interés en el funcionamiento del mismo.

Figura 4.1 Arbol directorio rviz_satellite.

Por ello se modificcaban los siguientes archivos:

- demo.gps: En este archivo se guarda los valores iniciales del tópico /gps/fix, el cual se corresponde con la posición de referencia del GPS. A priori es razonable que no se entienda porque se instancia de esa forma teniendo herramientas como rostopic pub, echo que quedará subsanado al final de la sección. A continuación se detalla el contenido del archivo:

[1] Fuente: Elaboración propia

```
header:
  seq: 999
  stamp:
    secs: 1435607489
    nsecs: 924811809
  frame_id: base_link
status:
  status: 0
  service: 1
latitude: 37.287050 //Latitud de referencia
longitude: -6.049846 //Longitud de referencia
altitude: 0 //Altitud de referencia
position_covariance: [3.9561210000000004, 0.0, 0.0, 0.0,
   3.9561210000000004, 0.0, 0.0, 0.0, 7.650756]
position_covariance_type: 2
```

- demo.rviz: El contenido de este archivo consta de las instancias de configuración del visualizador rviz unicamente. En el se detalla el objeto URI, tamaños de los marcadores, etc. A continuación se añaden sólo los aspectos mas destacables ya que cuenta con unas 200 líneas de código:

```
*********Defino el objeto URI, tópico a visualizar y sistema de
   referencia*********
Visualization Manager:
  Class: ""
  Displays:
    - Alpha: 1
      Blocks: 5
      Class: rviz_plugins/AerialMapDisplay
      Draw Behind: false
      Enabled: true
      Frame Convention: XYZ -> ENU
      Name: AerialMapDisplay
      Object URI: https://maps.wikimedia.org/osm-intl/{z}/{x}/{y}.
         png
      Topic: /gps/fix
      Value: true
      Zoom: 18
    - Class: rviz/Axes
      Enabled: true
      Length: 50
      Name: Axes
      Radius: 3
      Reference Frame: base_link
      Value: true
  Enabled: true
```

```
*********Defino las opciones de visualización*********
  Global Options:
    Background Color: 48; 48; 48
    Fixed Frame: map
    Frame Rate: 30
  Name: root
  Tools:
    - Class: rviz/Interact
      Hide Inactive Objects: true
    - Class: rviz/MoveCamera
    - Class: rviz/Select
    - Class: rviz/FocusCamera
    - Class: rviz/Measure
    - Class: rviz/SetInitialPose
      Topic: /initialpose
    - Class: rviz/SetGoal
      Topic: /move_base_simple/goal
    - Class: rviz/PublishPoint
      Single click: true
      Topic: /clicked_point
  Value: true
```

- demo.lauch: Como se explicó en el anterior capítulo, los archivos .launch se usan para lanzar varios nodos simultáneamente. A continuación se detalla el contenido de dicho archivo destacando que las explicaciones se añaden a modo de comentario:

```
<launch>
    <!-- Aquí se usa el archivo demo.gps (que ahora si se entiende
        su función) para inicializar /gps/fix mediante esa sintaxis
        -->
     <node pkg="rostopic" type="rostopic" name="fake_gps_fix" args
         ="pub /gps/fix sensor_msgs/NavSatFix --latch --file=$(find
         rviz_satellite)/launch/demo.gps" />


    <!-- Aquí se usa el archivo demo.rviz para el mismo fin que el
        anterior, solo que cambia el tópico -->
    <node pkg="rviz" type="rviz" name="rviz" args="-d $(find rviz_
        satellite)/launch/demo.rviz"/>

    <!-- El paquete tf2_ros se encarga de transformar los sistemas
        de referencias de map a base_link -->
    <node pkg="tf2_ros" type="static_transform_publisher" name="
        static_tf_fake" args="0 0 0 0 0 0 map base_link" />
</launch>
```

Realmente el elemento funcional mas importante es el .launch el cual en un futuro se añadirá al .launch del dron para que ambos compartan la misma instancia en el visualizador y así poder verlos de forma superpuesta.

4.3 Configuración de hector_quadrotor

Como ya se explicó en la sección 3.5.2, de este paquete solo se usa el modelo del dron para superponerlo en rviz junto al mapa GPS, por tanto, no se modifica absolutamente nada salvo los archivos referentes a las carpetas launch y rviz_cgf. A continuación se detalla de nuevo el arbol directorio de dicho paquete (figura 4.2)[2], además de las modificaciones llevadas a cabo:

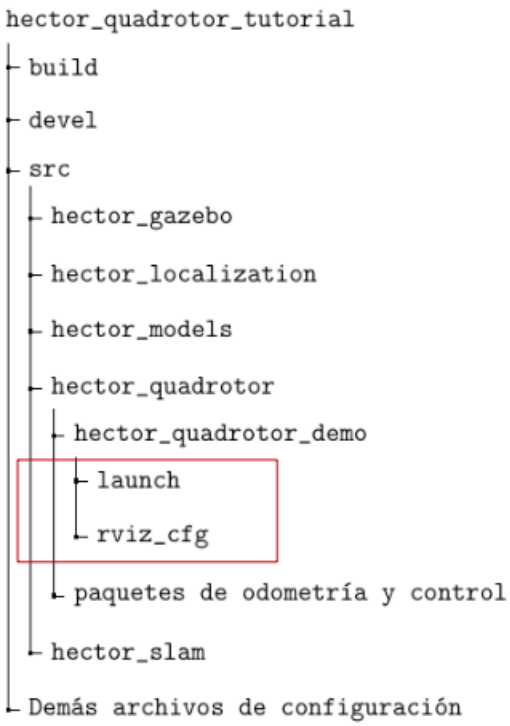


Figura 4.2 Arbol directorio hector_quadrotor_demo.

Los archivos modificados son los siguientes:

- rviz_cfg/rviz_coria.rviz: En este caso no se ha modificado un archivo .rviz, si no que se ha creado uno en base al de prueba. Basicamente lo que se ha hecho es un copia y pega del .rviz del GPS añdiendo la instancia del dron, la cual se muestra a continuación:

```
- Alpha: 1
    Class: rviz/RobotModel
    Collision Enabled: false
    Enabled: true
    Links:
      All Links Enabled: true
      Expand Joint Details: false
      Expand Link Details: false
      Expand Tree: false
      Link Tree Style: Links in Alphabetic Order
      ---
```

[2] Fuente: Elaboración propia

```
      ---
        base_link:
          Alpha: 1
          Show Axes: true
          Show Trail: false
          Value: true
        front_cam_link:
          Alpha: 1
          Show Axes: false
          Show Trail: false
        front_cam_optical_frame:
          Alpha: 1
          Show Axes: false
          Show Trail: false
        laser0_frame:
          Alpha: 1
          Show Axes: false
          Show Trail: false
          Value: true
        sonar_link:
          Alpha: 1
          Show Axes: false
          Show Trail: false
          Value: true
      Name: RobotModel
      Robot Description: robot_description
      TF Prefix: ""
      Update Interval: 0
      Value: true
      Visual Enabled: true
```

- launch/outdoor_coria.launch: Este archivo .launch es el pilar fundamental, ya que en el se ejecutan todos los nodos que dotan de funcionalidad al proyecto. Una parte de este archivo es un copia y pega del .launch del GPS como se explicó anteriormente, con el fin de visualizar ambos paquetes de forma simultánea. La otra parte restante del archivo en cuestión consta de estos elementos:
  - Instancia de un mundo vacío en gazebo: Como se ha ido haciendo hincapié a lo largo de la memoria, gazebo es el motor que genera un modelado físico en 3D del dron. Para ello se encarga de simular, entre otras cosas, los elementos físicos terrestres, como la gravedad, viento, etc. De esta forma si se instancia un dron en gazebo sin un mundo, se simularía un dron cayendo al vacío, por tanto es necesario definir un mundo vacío de la siguiente forma.

```
<!-- Inicializo parametros funcionales de gazebo en tiempo
    real (max) -->
  <arg name="paused" default="false"/>
  <arg name="use_sim_time" default="true"/>
  <arg name="extra_gazebo_args" default=""/>
  <arg name="gui" default="true"/>
  <arg name="recording" default="false"/>
  <arg name="headless" default="false"/>
  <arg name="debug" default="false"/>
  <arg name="physics" default="ode"/>
  <arg name="verbose" default="false"/>
  <arg name="output" default="screen"/>
  <arg name="world_name" default="worlds/empty.world"/>
  <arg name="respawn_gazebo" default="false"/>
  <arg name="use_clock_frequency" default="false"/>
  <arg name="pub_clock_frequency" default="100"/>

  <!-- Uso tiempo de simulacion-->
  <param name="/use_sim_time" value="$(arg use_sim_time)"/>

  <!-- Inicializo parametros auxiliares-->
  <arg unless="$(arg paused)" name="command_arg1" value=""/>
  <arg    if="$(arg paused)" name="command_arg1" value="-u"/>
  <arg unless="$(arg recording)" name="command_arg2" value
     =""/>
  <arg    if="$(arg recording)" name="command_arg2" value="-r
     "/>
  <arg unless="$(arg verbose)" name="command_arg3" value=""/>
  <arg    if="$(arg verbose)" name="command_arg3" value="--
     verbose"/>
  <arg unless="$(arg debug)" name="script_type" value="
     gzserver"/>
  <arg    if="$(arg debug)" name="script_type" value="debug
     "/>
  ---
```

Hasta este punto se inicializan todos aquellos parámetros para arrancar una simulación en tiempo real. Destacar que todo lo realiado ha sido en base a la documentación oficial de ROS referido a varios ejemplos, por esa razón aunque funcione, existe la posibilidad que inicialice algun parámetro que a priori no haga falta.

A continuación, se muestra como se inicializa la comunicación servidor-cliente de gazebo:

```
  <!-- Inicializo el servidor-->
  <group if="$(arg use_clock_frequency)">
  <param name="gazebo/pub_clock_frequency" value="$(arg pub_
      clock_frequency)" />
 </group>
  <node name="gazebo" pkg="gazebo_ros" type="$(arg script_
      type)" respawn="$(arg respawn_gazebo)" output="$(arg
      output)" args="$(arg command_arg1) $(arg command_arg2)
      $(arg command_arg3) -e $(arg physics) $(arg extra_gazebo
      _args) $(arg world_name)" />

  <!-- Inicializo el cliente -->
  <group if="$(arg gui)">
  <node name="gazebo_gui" pkg="gazebo_ros" type="gzclient"
      respawn="false" output="$(arg output)"   args="$(arg
      command_arg3)"/>
  </group>
```

– Spawn del dron: Una vez creado un mundo vacío es necesario indicar que el dron aparezca en ese mundo, a modo de referirlo de alguna forma a dicho mundo y que su comportamiento esté referido al mismo.

```
<!-- Spawn simulated quadrotor uav -->
<include file="$(find hector_quadrotor_gazebo)/launch/spawn_
    quadrotor.launch" >
  <arg name="model" value="$(find hector_quadrotor_
      description)/urdf/quadrotor_hokuyo_utm30lx.gazebo.xacro
      "/>
  <arg name="controllers" value="
      controller/attitude
      controller/velocity
      controller/position
      "/>
</include>
```

– Instanciado del GPS y la configuración rviz: Una vez creado un mundo con un punto delimitado de spawn, solo queda instanciar la configuración de rviz del dron con el GPS, además de inicializar el tópico /gps/fix como se ha explicado en la anterior sección.

```
<!-- Inicializo rviz con la configuracion explicada -->
  <node pkg="rviz" type="rviz" name="rviz" args="-d $(find
      hector_quadrotor_demo)/rviz_cfg/rviz_coria.rviz"/>
---
```

```
  ---
<!-- Inicializo gps/fix -->
  <node pkg="rostopic" type="rostopic" name="fake_gps_fix"
      args="pub /gps/fix sensor_msgs/NavSatFix --latch --file
      =$(find rviz_satellite)/launch/demo.gps" />
```

– Transformación de frames: En este punto se tienen 3 sistemas de referencias; dron, GPS y mundo, por tanto para que todo esté referido al mismo se realiza el mismo proceso con el paquete tf2_ros. Este apartado se puede abordar de distintas formas, tal y como está, se considera que el sistema de referencia principal es el del mnundo y por tanto hay que referir los demás a el.

```
<!-- Static fake TF transform -->
  <node pkg="tf2_ros" type="static_transform_publisher" name
      ="static_tf_fake" args="0 0 0 0 0 0 map base_link" />
  <node pkg="tf2_ros" type="static_transform_publisher" name
      ="$(anon ground_truth_to_tf)" args="0 0 0 0 0 0 map
      world" />
```

4.4 Scripts .py de control del dron

En esta sección se expone a nivel de pseudocódigo los archivos.py encargados de controlar en dron. Como se explicó anteriormente, el objetivo de este proyecto se basa en controlar la posición objetivo del dron y evitar obstáculos, por ese motivo se ha decidido separar ambos archivos ya que en la simulación del dron junto al GPS no había obstáculos, y por ello se hace uso del mapa de demostración, el cual constaba de un terreno montañoso.

4.4.1 muevo_dron.py

Este script esta destinado a corroborar que el dron alcanza los puntos GPS dados por el usuario. Para evaluar su funcionamiento habría que seguir los siguientes pasos:

```
mrhyde@francisco-PC:~$ roslaunch hector_quadrotor_demo outdoor_coria.
    launch
```

A continuación se ejecuta el nodo de control en otro terminal

```
mrhyde@francisco-PC:~$ rosrun tfg muevo_dron.py
Introduzca la latitud
37.28705
Introduzca la longitud
-6.049846
Introduzca la Altura
10
```

Como se explicó anteriormente, el punto de spawn [0, 0, 0] corresponde con el punto en coordenadas GPS [37.28705, -6.049846, 0], por tanto, solo se estaría aumentando la altura del dron. Cuando se introduce nuevos puntos objetivos se publica dichos puntos en el tópico /move_base_simple/goal, ya que dicho tópico está visualizado en rviz, y así poder ver mediante simulación que el dron llega a puntos dados por un usuario (figura 4.3)[3].

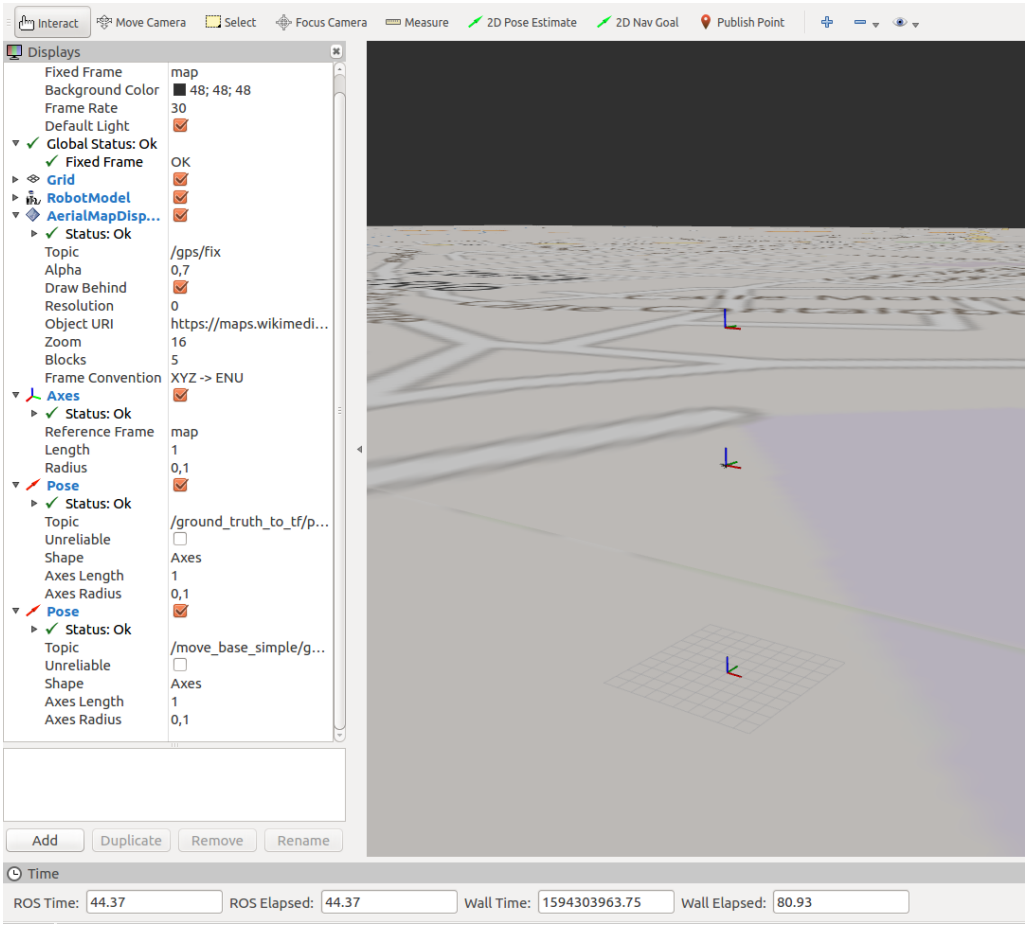


Figura 4.3 Demo dron GPS.

A continuación se explica el código de forma breve mediante pseudocódigo, ya que el objetivo de esta sección es entender el funcionamiento principalmente ya que la sintaxis se aborda en los apéndices como se ha ido comentado a lo largo de la memoria:

Código 4.1 muevo_dron.py->Pseudocódigo.

```
Importo librerías asociadad a tópicos:
  -/ground_truth_to_tf/pose
    -action/pose/goal
    -move_base_simple/goal

{Inicializo las variables de posicion y posicion realimentada}

{Instancio la funcion de suscripcion a /ground_truth_to_tf/pose}
```

[3] Fuente: Simulación propia

```
{Instancio funcion principal}
  Declaro todas las variables de suscripcion y publicacion
  {Pido posicion de referencia}
    publico_posicion() -> action/pose/goal
    publico_marcador() -> move_base_simple/goal

  while no este rospy apagado
    if si llega a esa posicion
      {Pido posicion de referencia}
      publico_posicion() -> action/pose/goal
      publico_marcador() -> move_base_simple/goal
```

4.4.2 muevo_posicion.py

Este script se encarga de evitar que el dron colisione con algún obstáculo del entorno de trabajo a una distancia predefinida por el programador. Es sabido que existen numerosos algoritmos de detección de obstáculos y cálculo de caminos en base a dichas detecciones, pero no se consideran funcionales para este proyecto, ya que los entornos relacionados con la aplicación aérea no cuentan con obstáculos a priori, y si hay alguno, es de forma muy excepcional, y por tanto, necesitaría de intervención humana para trazar un nuevo punto objetivo en base a que el anterior no es posible dado que hay un obstáculo. De este modo se ha decidio que cuando el dron detecte un obstáculo, vaya al punto seguro anterior, y llegado a ese punto pida una nueva posición al usuario.
De la misma forma que el apartado anterior, los pasos a seguir en la verificación de este programa es la siguiente:

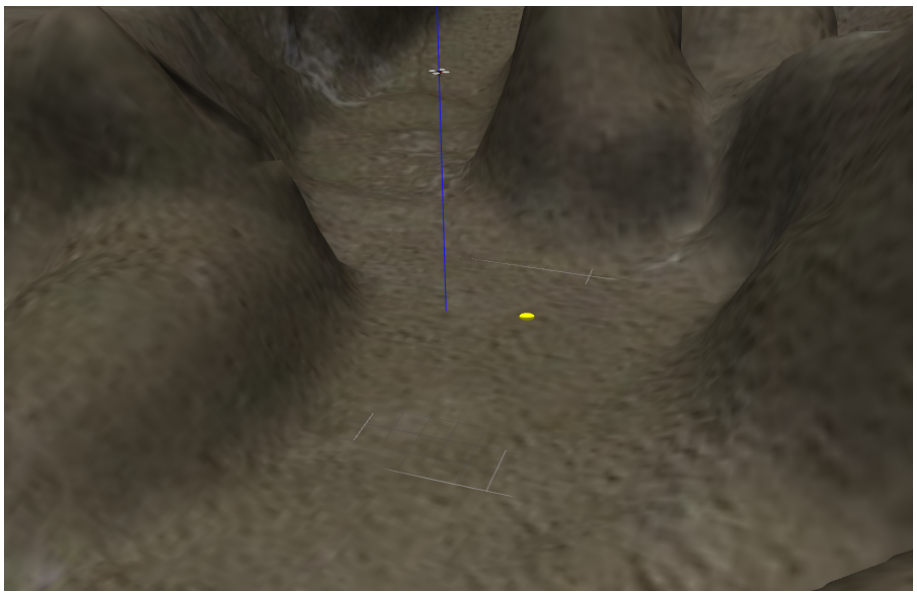
```
mrhyde@francisco-PC:~$ roslaunch hector_quadrotor_demo outdoor_flight_
    gazebo.launch
```

En un nuevo terminal se ejecuta el archivo de control

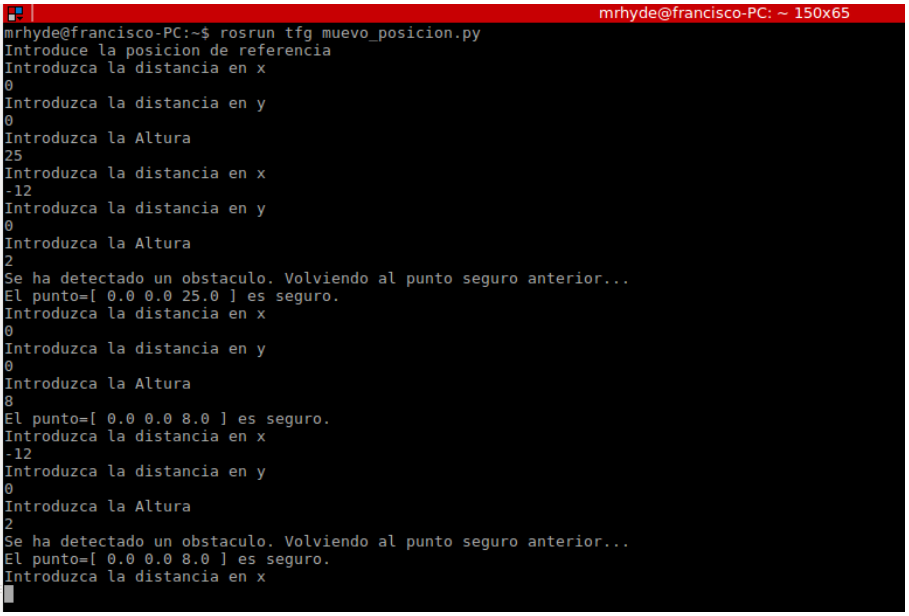
```
mrhyde@francisco-PC:~$ rosrun tfg muevo_posicion.py
Introduce la posicion de referencia
Introduzca la distancia en x
0
Introduzca la distancia en y
0
Introduzca la Altura
25
```

Es importante que la primera posición objetivo (posición de referencia) sea una posición libre de obstáculos (como la del ejemplo), debido a que cuando se detecta un obstáculo el dron vuelve a la posición objetivo anterior, y por tanto si estamos en el comienzo del programa, la posición anterior simplemente no existe.

A continuación, en la figura 4.4 a),[4] se muestra el dron en dicho entorno en la posición inicial definida anteriormente, junto con el instante donde se proporciona por consola (figura 4.4 b)[5]) un punto localizado en un obstáculo:



(a)



(b)

Figura 4.4 a) Dron en entorno obstáculos b) Consola.

Esta metodología para evitar obstáculos no es autónoma, ya que el dron detecta un obstáculo, vuelve a la anterior posición segura y espera a la internvención humana que arroje un nuevo punto, pero para la aplicación en cuestión es la opción mas segura ya que se asegura que no va haber colisión si el primer punto dado por el usuario es libre de obstáculos y además se aprovecha el echo de que haya una persona supervisando.

[4] Fuente: Simulación propia
[5] Fuente: Elaboración propia

A continuación se explica el algoritmo mediante pseudocódigo:

Código 4.2 muevo_posicion.py ->Pseudocódigo.

```
Importo librerías asociadad a tópicos:
  -/ground_truth_to_tf/pose
    -/action/pose/goal
    -/move_base_simple/goal
  -/scan
{Inicializo las variables de posicion referencia, posicion objetivo y
    obstaculo}

{Instancio funcion que suscribe /ground_truth_to_tf/pose}

{Instancio funcion que suscribe /scan}
  if distancia_obstaculo<distancia_minima
    obstaculo=1
  else
    obstaculo=0

{Instancio funcion principal}
  Declaro todas las variables de suscripcion y publicacion de topicos
      usados
  Pido posicion de referencia
  Publico posicion referencia
  flag=0
  while flag==0
    if dron llega a la referencia
      Pido posicion objetivo
      Publico posicion objetivo
      Posicion anterior=Posicion referencia
    flag=1
  while rospy no este apagado
    if obstaculo==1
      Publico -100*posicion objetivo ->El dron se aleja rapidamente del
          obstaculo
      Publico posicion anterior
    if llega a posicion objetivo
      Posicion anterior=Posicion objetivo
      Pido nueva posicion objetivo
      Publico posicion objetivo
```

5 Conclusiones y trabajo futuro

A lo largo de este último apartado se expone las conclusiones obtenidas tras finalizar el trabajo, así como una breve explicación de los siguientes pasos en el desarrollo del proyecto principal en cuestión.

5.1 Conclusiones

Para hacer más facil el entendimiento se va a separar las conclusiones por funcionalidad, es decir, se detallarán las conclusiones obtenidas en la simulación en Rviz del GPS, y en la siguiete subsección se abordarán las conclusiones del algoritmo de evitación de obstáculos.

5.1.1 Simulación en Rviz del GPS

Los resultados obtenidos de esta parte han sido muy positivos ya que la potencial fuente de error, la cual es la precisión del GPS y errores en componentes hardware, son despreciables en simulaciones de este tipo ya que se tienen condiciones cuasi ideales, en comparación con la cantidad de problemas que causaría esta misma prueba implantada en un dron con GPS real. De este modo, como se expuso en el apartado 4.2 se verifica visualementemediante mediante marcadores en Rviz, con posición igual a la posición objetivo dada por el usuario, que el dron se ajusta perfectamente a cada punto en cuestión. De esta forma el usuario puede comprobar de forma empírica e inmediata los resultados.

5.1.2 Evitación de obstáculos

Como se ha ido explicando, el entorno de trabajo consta de una superficie plana y diáfana, en lo que área de aplicación se refiere, por tanto, las exigencias cara a un algoritmo de evitación de obstáculos se antoja poco exigente ya que el dron se limita a alcanzar puntos trazados a priori por un humano, por tanto, como es poco probable que alguien traze como punto objetivo un obstáculo se presupone que los obstáculos van a ser de naturaleza aleatoria, y por tanto, al ser un fenómeno unico y aletorio no tendría sentido hacer un algoritmo complejo, basado en cálculo de caminos, etc. Se presupone mas funcional un algoritmo que simplemente, cuando el dron detecte un obstáculo, vuelva al punto anterior seguro más cercano, y espere que el usuario planifique una nueva trayectoria con el fin de continuar con el trabajo y asegurar la seguridad del proyecto.
Del mismo modo que en el apartado 4.3, se verifica mediante la simulación en Gazebo, que el dron no colisiona con ningún elemento del mundo simulado. Importante destacar que la efectividad de esta metodología depende de la distancia mínima de detección, ya que a priori no existe ningún tópico que regule la velocidad del dron sin variar los parámetros funcionales de los PID de control usados en cada motor, por tanto, si tiene un tiempo de reacción estimado de t segundos y el dron va a una velocidad x metros/s, la distancia mínima estimada sería del orden de x*t metros.

5.2 Trabajo futuro

Actualmente se está ideando junto a la empresa Bluumi, la posibilidad de realizar una inversión inicial, con el fin de integrar un autopiloto en el frame de un dron capaz de soportar una infraestructura que integre el hardware de aplicación de pesticidas, además de los primeros pasos en el desarrollo de una aplicación Android que envía, mediante conexión a internet o bluetooth, coordenadas GPS obtenidas a partir de pulsar un cierto punto en un mapa, a otro módulo con bluetooth o conexión a internet. Dicha inversión dependerá del resultado del estudio de mercado que se lleva realizando unos meses, donde se está intentando estimar el número de agricultores o cooperativas interesadas en este proyecto.
En resumen el coste de aplicación de pesticidas es de unos 40 euros la hectárea, por tanto, asumiendo una inversión de 20.000 euros mas 10.000 euros en fondos de emergencia, bastaría con captar del orden de 750 hectáreas para recuperar la inversión, que en adición al dato de que en Andalucía hay del orden de 1.5 millones de hectáreas sólo de olivo, se presupone que hace falta un porcentaje muy bajo de aplicación en relación a superficie total, dato que asume que hay un gran mercado, y por tanto, una necesidad real si se consigue obtener un precio de aplicación menor que el tradicional.

Apéndice A
Se adjuntan los códigos de control del dron

Código A.1 muevo_posicion.py.

```
#!/usr/bin/python
#Importo el tipo de mensajes
import rospy
from geometry_msgs.msg import PoseStamped #/ground_truth_to_tf/pose
from hector_uav_msgs.msg import PoseActionGoal #/action/pose/goal
from sensor_msgs.msg import LaserScan #/scan
#Librerias para operaciones
import numpy as np
import sys

#Posiciones actuales del dron
x=0
y=0
z=0
#Vector de poisicones dada por usuario
pos_r=[0, 0, 0]
pos_obj=[0, 0, 0]
#Indica si hay obstaculo
obstaculo=0

#funcion que se suscribe a /ground_truth_to_tf/pose
def callback(msg):
    global x, y, z
    x=msg.pose.position.x
    y=msg.pose.position.y
    z=msg.pose.position.z

#funcion que se suscribe a /scan
#Se usa esta funcion para poner el vector en valor absoluto
def abs_lista(lista):
```

```
  for i in range(len(lista)):
    lista[i] = abs(lista[i])
def scan(msg):
    global obstaculo
    dist_min=8
    arr = np.array(msg.ranges)
    abs_lista(arr)

    if(np.amin(arr)<dist_min):
        obstaculo=1
    else:
        obstaculo=0


#funcion principal
def muevo_dron():
    #Defino variables globales
    global obstaculo
    global pos_r, pos_obj
    global x_r, y_r, z_r
    global estado, datos
    global pos, pub
    rospy.init_node('muevo_dron', anonymous=False)
    pub=rospy.Publisher('action/pose/goal', PoseActionGoal, queue_size
        =10)
    rospy.Subscriber('/ground_truth_to_tf/pose',PoseStamped,callback)


    #Defino tipo de mensaje
    pos=PoseActionGoal()

    #Defino referencia
    print "Introduce la posicion de referencia"
    datos=('distancia en x', 'distancia en y', 'Altura')
    for i in range(3):
        print "Introduzca la",datos[i]
        valor=raw_input()
        pos_r[i]=float(valor)

    #Publico referencia
    pos.goal.target_pose.pose.position.x=pos_r[0]
    pos.goal.target_pose.pose.position.y=pos_r[1]
    pos.goal.target_pose.pose.position.z=pos_r[2]
    pos.goal.target_pose.header.frame_id='world'
    pub.publish(pos)

    #LLego a la referencia inicial
    flag=0
    while flag==0:
        #Calculo errores de diferencia
```

```
        e_x=x-pos_r[0]
        e_y=y-pos_r[1]
        e_z=z-pos_r[2]

        if ((e_x<0.01 and e_y<0.01) and e_z<0.01):
            rospy.sleep(2.)
            #Calculo errores de diferencia
            e_x=x-pos_r[0]
            e_y=y-pos_r[1]
            e_z=z-pos_r[2]
            if ((e_x<0.01 and e_y<0.01) and e_z<0.01):
                for i in range(3):
                    print "Introduzca la",datos[i]
                    valor=raw_input()
                    pos_obj[i]=float(valor)
                pos.goal.target_pose.pose.position.x=pos_obj[0]
                pos.goal.target_pose.pose.position.y=pos_obj[1]
                pos.goal.target_pose.pose.position.z=pos_obj[2]
                pos.goal.target_pose.header.frame_id='world'
                pub.publish(pos)
                x_ant=pos_r[0]
                y_ant=pos_r[1]
                z_ant=pos_r[2]
                flag=1

    #Espero a que llegue a la referencia
    sub=rospy.Subscriber('/scan',LaserScan,scan)
    while not rospy.is_shutdown():
        if obstaculo==1:
            #Se aleja
            pos.goal.target_pose.pose.position.x=-10000*pos_obj[0]
            pos.goal.target_pose.pose.position.y=-10000*pos_obj[1]
            pos.goal.target_pose.pose.position.z=-10000*pos_obj[2]
            pos.goal.target_pose.header.frame_id='world'
            pub.publish(pos)
            print "Se ha detectado un obstaculo. Volviendo al punto
                seguro anterior..."
            sub.unregister()
            obstaculo=0
            rospy.sleep(1.5)
            #Publico referencia
            pos.goal.target_pose.pose.position.x=x_ant
            pos.goal.target_pose.pose.position.y=y_ant
            pos.goal.target_pose.pose.position.z=z_ant
            pos.goal.target_pose.header.frame_id='world'
            pub.publish(pos)
            pos_obj[0]=x_ant
            pos_obj[1]=y_ant
            pos_obj[2]=z_ant
```

```
        e_x=x-pos_obj[0]
        e_y=y-pos_obj[1]
        e_z=z-pos_obj[2]


        if ((e_x<0.01 and e_y<0.01) and e_z<0.01):
            rospy.sleep(2.)
            e_x=x-pos_obj[0]
            e_y=y-pos_obj[1]
            e_z=z-pos_obj[2]
            if ((e_x<0.01 and e_y<0.01) and e_z<0.01):
                print "El punto=[",pos_obj[0], pos_obj[1], pos_obj[2],"]
                    es seguro."
                x_ant=pos_obj[0]
                y_ant=pos_obj[1]
                z_ant=pos_obj[2]
                for i in range(3):
                    print "Introduzca la",datos[i]
                    valor=raw_input()
                    pos_obj[i]=float(valor)
                sub=rospy.Subscriber('/scan',LaserScan,scan)
                pos.goal.target_pose.pose.position.x=pos_obj[0]
                pos.goal.target_pose.pose.position.y=pos_obj[1]
                pos.goal.target_pose.pose.position.z=pos_obj[2]
                pos.goal.target_pose.header.frame_id='world'
                pub.publish(pos)


if __name__=='__main__':
    try:
       muevo_dron()
    except rospy.ROSInterruptException:
        pos.goal.target_pose.pose.position.x=0
        pos.goal.target_pose.pose.position.y=0
        pos.goal.target_pose.pose.position.z=0
        pos.goal.target_pose.header.frame_id='world'
        pub.publish(pos)
        pass
```

Código A.2 muevo_dron.py.

```
#!/usr/bin/python
#Importo el tipo de mensajes
import rospy
from geometry_msgs.msg import PoseStamped #/ground_truth_to_tf/pose
from hector_uav_msgs.msg import PoseActionGoal #/action/pose/goal
from geometry_msgs.msg import PoseStamped #/move_base_simple/goal
#Librerias para operaciones
import numpy as np
```

```
import sys
#Libreria conversion GEO_ECEF_ENU
import geo

#Posiciones actuales del dron
x=0
y=0
z=0
#Vector de poisicones dada por usuario
pos_r=[0, 0, 0]

#funcion que se suscribe a /ground_truth_to_tf/pose
def callback(msg):
    global x, y, z
    x=msg.pose.position.x
    y=msg.pose.position.y
    z=msg.pose.position.z

#funcion principal
def muevo_dron():
    #Variables
    global pos_r #Pos de referencia en GPS
    global x_r, y_r, z_r #Pos de referencia cartesiana
    #Inicio nodo
    rospy.init_node('muevo_dron', anonymous=False)
    #Defino pub parapublicar posicion
    pub=rospy.Publisher('action/pose/goal', PoseActionGoal, queue_size
        =10)
    #Me suscribo a la posicion actual del dron
    rospy.Subscriber('/ground_truth_to_tf/pose',PoseStamped,callback)
    #Defino pub1 para publicar la siguiente posicion a modo de
        referencia visual
    pub1=rospy.Publisher('/move_base_simple/goal', PoseStamped, queue_
        size=10 )

    pos=PoseActionGoal()
    sig_pos=PoseStamped()

    #Pido posicion de referencia
    datos=('latitud', 'longitud', 'Altura')
    for i in range(3):
        print "Introduzca la",datos[i]
        valor=raw_input()
        pos_r[i]=float(valor)

    #Conversion de GPS a ENU
    x_r,y_r,z_r=geo.geodetic_to_enu(pos_r[0],pos_r[1],pos_r
        [2],37.28705,-6.049846,0)

    #Publico las coordenadas ENU
```

```
    pos.goal.target_pose.pose.position.x=x_r
    pos.goal.target_pose.pose.position.y=y_r
    pos.goal.target_pose.pose.position.z=z_r
    pos.goal.target_pose.header.frame_id='world'
    pub.publish(pos)

    #Publico el siguiente punto
    sig_pos.pose.position.x=x_r
    sig_pos.pose.position.y=y_r
    sig_pos.pose.position.z=z_r
    sig_pos.header.frame_id='world'
    pub1.publish(sig_pos)

    while not rospy.is_shutdown():
        #calculo los errores de pos
        e_x=x_r-x
        e_y=y_r-y
        e_z=z_r-z
        if ((e_x<0.1 and e_y<0.1) and e_z<0.1):
            e_x=x_r-x
            e_y=y_r-y
            e_z=z_r-z
            if ((e_x<0.1 and e_y<0.1) and e_z<0.1):
                for i in range(3):
                    print "Introduzca la",datos[i]
                    valor=raw_input()
                    pos_r[i]=float(valor)
                x_r,y_r,z_r=geo.geodetic_to_enu(pos_r[0],pos_r[1],pos_r
                    [2],37.28705,-6.049846,0)
                pos.goal.target_pose.pose.position.x=x_r
                pos.goal.target_pose.pose.position.y=y_r
                pos.goal.target_pose.pose.position.z=z_r
                pos.goal.target_pose.header.frame_id='world'
                pub.publish(pos)

                #Publico el siguiente punto
                sig_pos.pose.position.x=x_r
                sig_pos.pose.position.y=y_r
                sig_pos.pose.position.z=z_r
                sig_pos.header.frame_id='world'
                pub1.publish(sig_pos)




if __name__=='__main__':
    try:
        muevo_dron()
    except rospy.ROSInterruptException:
        pass
```

Índice de Figuras

Índice de Códigos

Bibliografía

[1] Morgan Quigley, Brian Gerkey & William D. Smart.(2015) *Programming Robots with ROS*. EEUU: O'Reilly.

[2] Lentin Joseph.(2018) *Robot Operating System for Absolute Beginners*. New York: Apress.

[3] Carol Fairchild & Dr. Thomas L. Harman.(2016) *ROS Robotics By Example*. Birmingham, UK: [PACKT]Open source*.

[4] Documentación y API's oficiales de ROS.
`http://wiki.ros.org/es`

[5] Fundamentos del GPS.
`https://es.wikipedia.org/wiki/GPS`

[6] Fundamentos de coordenadas geodesicas.
`https://www.ign.es/web/ign/portal/gds-teoria-geodesia`

[7] Historia aplicación aérea.
`https://es.wikipedia.org/wiki/Aplicacion_aerea_plaguicidas`