UNIVERSIDAD AUTÓNOMA DE YUCATÁN

FACULTAD DE MATEMÁTICAS

Proceso de ensamblado y desarrollo de un entorno de simulación de un dron

Tesis presentada por:
I.E. Carlos Enrique Acosta Montalvo

Para obtener el grado de:
Maestro en Ciencias de la Computación

Supervisada por:
Dr. Carlos Brito Loeza
Dr. Arturo Espinosa Romero

Mérida, Yucatán, México
Enero 2019

Declaración de Autoría

Yo, I.E. Carlos Enrique Acosta Montalvo, declaro que esta tesis titulada "Proceso de ensamblado y desarrollo de un entorno de simulación de un dron" y el trabajo aquí presentado son de mi autoría. Y confirmo que:

- Este trabajo fue realizado durante el período de maestría que cursé en la Facultad de Matemáticas de la Universidad Autónoma de Yucatán.
- Cualquier información utilizada en este trabajo perteneciente a otros autores se encuentra debidamente referenciada.
- Las herramientas y material de terceros incluyendo bibliotecas, código fuente, software e imágenes fueron utilizadas exclusivamente con fines educativos y se referenciaron adecuadamente.

Resumen

En este trabajo se abordan aspectos prácticos pero importantes para el uso y desarrollo de proyectos con vehículos aéreos no tripulados.

El trabajo comienza con una pequeña introducción a los drones, específicamente a los modelos con seis motores, detallando el proceso de ensamblado de la estructura, el montaje de hardware mencionando sus especificaciones, así como las configuraciones de software necesarias para que el vehículo pueda levantar vuelo utilizando la estación de control terrestre y el sistema de radio control.

Posteriormente se detalla el proceso de simulación de vuelo de un dron a través de Gazebo, el cual es un ambiente de simulación con alto grado de fidelidad, de libre acceso y con una comunidad muy amplia de desarrollo. Aparte, este ambiente tiene la capacidad de operar de forma coordinada con el sistema ROS.

En el trabajo se detalla el diseño de un entorno dentro del simulador, e igualmente el uso del sistema operativo ROS para trabajar con el vehículo y poder guiarlo automáticamente a partir de programas que se ejecutan en la computadora de vuelo. El uso de simuladores permite observar los diferentes desempeños del vehículo para diferentes algoritmos de control previo a la realización de pruebas físicas reales.

Agradecimientos

Agradezco y doy gracias a Dios por haberme dado vida y capacidad intelectual para acabar de manera satisfactoria este trabajo.

A mis padres, José María y Alicia, yo soy quien soy gracias a su esfuerzo y apoyo incondicional, gracias por todo el amor que siempre me han dado y gracias por que siempre me han guiado para ser una mejor persona y un buen estudiante.

A mi hermano Emmanuel, por todas esas veces que necesito ayuda y él está ahí para brindarla.

A mis tíos, Pedro y Maria, por ser siempre buenos consejeros, gracias por venir desde lejos a pasar tiempo en familia y estar siempre presentes en mis logros académicos.

A mis padrinos, Renan y Gregoria, por todo el apoyo que siempre nos brindan.

A mis asesores de tesis, Dr. Carlos Brito y Dr. Arturo Espinosa, por el conocimiento brindado de manera excelente en las materias que impartieron y por su apoyo al aclarar cualquier tipo de duda o situación que se presentó durante la realización del presente trabajo.

Al Dr. Fernando Curi por el conocimiento y apoyo brindado como coordinador durante todo el transcurso del posgrado, así como un agradecimiento a todos los demás profesores que me impartieron clases, siendo todos excelentes al compartir sus conocimientos.

Al CONACYT por la beca otorgada y a la Facultad de Matemáticas por ser ahora un egresado perteneciente a esta gran Universidad.

Índice general

Índice de figuras

Índice de cuadros

Dedicado a mis padres, los amo con todo mi corazón y gracias por estar siempre conmigo.

Capítulo 1

Introducción

1.1. Introducción

El desarrollo tecnológico en años recientes de los robots conocidos como vehículos aéreos no tripulados (VANT) y popularmente llamados drones, así como su utilización para realizar tareas que ayuden al desarrollo de proyectos en áreas diversas de investigación o área comercial ha tenido mucho auge.

Estos vehículos existen en diversos tipos y se pueden clasificar basados en su habilidad para desarrollar un trabajo particular. Los que utilizan 6 rotores o propelas se conocen como hexacópteros, tienen la característica de que su aterrizaje y despegue es más estable que los modelos con 4 propelas, así como su maniobrabilidad y poder de vuelo.

De la mano con el desarrollo de los VANT, la importancia de los simuladores tiene un papel prioritario en la investigación robótica, siendo eficaces y rápidos al dar la capacidad de observar el comportamiento que tendría el robot al probar estrategias, nuevos conceptos y algoritmos que se utilizarán en los robots reales, los cuales solo requerirán de pequeños ajustes necesarios por errores que se generan en la implementación real y que el simulador no es capaz de reproducir de forma precisa.

Existen diversos simuladores para robótica. Entre los más conocidos se encuentran V-REP, USARSim, jmeSim, Webots y Gazebo, entre otros , cada uno ofrece gran fidelidad en gráficos y buena exactitud al simular la física del robot [1]. En particular, siendo Gazebo un simulador de código abierto lo convierte en una buena elección para desarrollar simulaciones de proyectos en robótica.

El simulador Gazebo ofrece integración con el sistema operativo para robots (*ROS por sus siglas en inglés*), esta capacidad los hace ser muy utilizados en conjunto en el desarrollo de proyectos con drones. Debido a que ROS es un entorno flexible de código abierto para programar robots que cuenta con una comunidad muy amplia de desarrolladores, se vuelve conveniente de utilizar por su capa de abstracción de hardware la cual permite que se puedan construir aplicaciones sin preocuparse por el hardware subyacente [2], entre otras características.

1.2. Objetivos

A continuación se presentan los objetivos de la tesis, el objetivo general es la meta principal a lograr en el trabajo y los objetivos particulares son los pasos a lograr para alcanzar la meta principal durante el desarrollo de la metodología, de igual forma se presenta la estructura general de la tesis y la organización por capítulos.

Objetivo general

El objetivo general consiste en describir el proceso de ensamblamiento de la estructura, el montaje de hardware y la configuración de software de un dron hexarotor; de igual manera, el uso del simulador de robótica Gazebo para el diseño de un escenario real de operación de este dron donde se puedan realizar vuelos simulados; así también, la implementación de programas en la computadora de vuelo para el guiado automático del vehículo. Todo esto con el fin de sentar las bases para el desarrollo posterior de algoritmos de visión computacional que se utilicen en el mismo.

Objetivos específicos

- Llevar acabo el ensamblado de la estructura, montaje del hardware y configuración del software del vehículo aéreo.
- Establecer comunicación con la computadora de vuelo y desarrollar programas que controlen el movimiento del vehículo.
- Llevar acabo simulaciones en Gazebo para observar el funcionamiento del vehículo.
- Obtención de imágenes en el simulador utilizando la cámara del vehículo aéreo para la detección de marcadores.
- Diseñar un entorno de simulación en Gazebo similar al entorno donde el vehículo se desenvolverá.
- Implementar un controlador sencillo para el seguimiento de objetivos.

1.3. Estructura de la tesis

El trabajo se encuentra dividido por capítulos de la siguiente forma:

- El capítulo dos presenta el estado del arte que es una recopilación de la información actualmente disponible sobre el tema.
- El capítulo tres presenta el marco teórico, donde se exponen los argumentos que sustentan el proyecto y se utilizan durante su desarrollo.
- El capítulo cuatro presenta el desarrollo de la metodología, exponiendo la planeación y teoría específica para realizar el proyecto.
- El capítulo cinco presenta el diseño y los resultados experimentales llevados a cabo en la implementación del proyecto.
- El capítulo seis presenta las conclusiones del proyecto y el trabajo a futuro.

Capítulo 2

Estado del arte

2.1. Trabajos relacionados con Visual Servoing

En esta sección, se presentan trabajos que incluyen técnicas que involucran el uso de *Visual Servoing*. Este tema tiene como objetivo principal el minimizar la función de error:

$$e(t) = s(m(t), a) - s*$$

donde

- $m(t)$, vector que contiene medidas relacionadas con el objeto de interés.
- a incluye conocimiento adicional del sistema, típicamente parámetros intrínsecos de la cámara.
- s representa un vector dimensional k calculado de $m(t)$ y a.
- $s*$ es el estado deseado del vector s.

En el trabajo realizado por Oualid Araar y Nabil Aouf [3] se utiliza *Visual Servoing* basado en imágenes para hacer que un dron de cuatro motores sea capaz de realizar una inspección de manera autónoma a las líneas de transmisión de energía eléctrica ($p_L = 0$) con un rumbo constante ψ y una altura determinada h como se muestra en la figura 2.1.

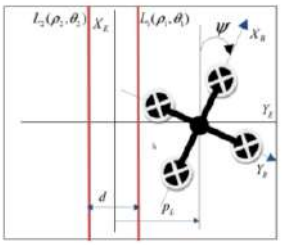


Figura 2.1: Modelo gráfico de la inspección autónoma.

La representación que adoptan en este trabajo se basa en coordenadas polares ρ, θ. La ecuación de una línea $2D$ usando esta representación está dada por

$$x \cos\theta + y \sin\theta = \rho$$

La elección de esta representación ayuda debido a que la mayoría de algoritmos de detección están basados en la transformada de Hough, la cual usa coordenadas polares. Esta representación resulta en un buen desacoplamiento en la matriz de iteración cuando se usa el enfoque de *Visual Servoing* basado en imágenes. Para simplificar el problema *Servoing* se considera que las líneas eléctricas sean aproximadas como rectilíneas y paralelas, y de que éstas pertenecen a un plano paralelo al suelo.

En el trabajo realizado por Marinela Georgieva Popova [4] se implementan las dos técnicas de *Visual Servoing*, basado en imágenes y basado en la pose, en un VANT. En el trabajo se asume que el objetivo está localizado en el piso y nunca cambia su altitud, sin ninguna restricción en su movimiento horizontal. La información sobre la posición deseada es obtenida de una cámara montada en el vehículo, la cual tiene una orientación fija con respecto al VANT, en el trabajo se asume que pueden medir la pose del vehículo. Finalmente, asumen que se dará una altura deseada y un ángulo *yaw* deseado (ambos constantes) los cuales quieren que el dron alcance. El objetivo es entonces definir un algoritmo de control, el cual dirija el vehículo hacia la posición horizontal, la altura y el ángulo *yaw* deseado.

Figura 2.2: Implementación de *Visual Servoing* en un AR.Drone para la detección de un marcador.

En lo presentado por Wang Chao [5] se motiva en el hecho de que los drones pueden volar de manera independiente en lugares donde los sistemas de comunicación con la base de telemetría fallen. En el trabajo se utiliza un Parrot AR.Drone. Para lograr el movimiento autónomo del vehículo se implementan controladores proporcionales sobre los movimientos de guiña y altitud, así como controladores PID en los movimientos de alabeo y cabeceo. El autor utiliza el enfoque de *Visual Servoing* basado en imágenes

para hacer que un vehículo detecte a través de su cámara un marcador visual con forma de círculo como se muestra en la figura 2.2, detallando el procesamiento de imagen realizado para la detección del marcador y el seguimiento del mismo, además del comportamiento global del dron para no perder de vista el marcador y tener un vuelo estable al seguir el objetivo cuando se mueva en el suelo.

En lo realizado por Lee *et al.* [6] se utiliza *Visual Servoing* basado en imágenes integrado con un control de deslizamiento adaptativo para operar un dron de cuatro motores. Para una integración perfecta con dinámica de cuadrorotor no actuada, los canales de balanceo y cabeceo se desacoplan de los otros canales usando características virtuales. Esto permite un algoritmo simple y preciso para estimar la información de profundidad y la aplicación exitosa del algoritmo de orientación y control propuesto.La configuración general permite que las características de la imagen se coloquen en la posición deseada en el plano de la imagen de una cámara montada en el dron. La estabilidad del sistema IBVS con el controlador se prueba usando el análisis de estabilidad de Lyapunov. El rendimiento del enfoque general se valida mediante la simulación numérica, la simulación de hardware integrado en la visión y los experimentos. Los resultados confirman que la imagen objetivo se coloca con éxito en la posición deseada del plano de la imagen y las variables de estado del cuadrorotor están debidamente reguladas, mostrando robustez en presencia de ruido del sensor, incertidumbre paramétrica y vibración de los motores.

2.2. Trabajos relacionados con ROS y Gazebo

El sistema ROS y el simulador Gazebo son ampliamente usados en el área de robótica, por un lado ROS es provechoso debido a su fácil abstracción de hardware y reutilización de código, mientras que gracias a sus motores de física y alta definición en gráficos Gazebo permite a los investigadores sustituir el sistema físico con el modelo simulado con el fin de estimar el comportamiento antes de aplicarlo al robot real.

En la literatura es posible encontrar diversas investigaciones que emplean estos dos sistemas en conjunto como marco de trabajo básico para su desarrollo. En lo realizado por Qian *et al.* [7] tienen la intención de crear una simulación de un manipulador e ilustrar los métodos de como implementar el controlador del robot de una manera rápida, eligen a ROS para organizar rápidamente la arquitectura de tareas y a Gazebo por su amplia compatibilidad con el mismo sistema. Uno de los objetivos principales de este trabajo es presentar a los investigadores no tan familiarizados la idea de entender fácilmente como trabaja el sistema y el simulador en conjunto.

En la investigación de Afanasyev *et al.* [8] los autores describen el enfoque de simulación utilizando Gazebo para localización y mapeo simultáneo (*SLAM por sus siglas en inglés*) basado en ROS usando un robot PR2. Consideran una simulación de SLAM basada en ROS utilizando el algoritmo *OpenSLAM GMapping* basado en un filtro de partículas. Utilizando dos láser LRF, uno montado en la base del PR2 y otro montado en el torso del robot. La tarea del algoritmo es interpretar los datos provenientes del sensor puesto en la base del robot en orden de producir un mapa de un entorno desconocido y realizar simultáneamente una autolocalización dentro de este mapa. Con esto se demuestra la factibilidad de usar SLAM basado en ROS utilizando un robot móvil en un entorno interior simulado en Gazebo.

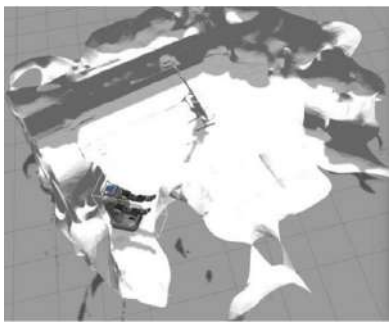


Figura 2.3: Simulación en Gazebo del robot PR2 implementando SLAM.

Los investigadores consideran adecuado el uso del simulador para realizar estudios en drones debido a todos riesgos que con lleva el utilizar el vehículo físicamente. Como se muestra en lo realizado por Meyer *et al.* [9] y por Zhang *et al.* [10] basar un proyecto en ROS y utilizar un entorno interior simulado en Gazebo similar al laboratorio real para realizar pruebas es favorable al implementar SLAM en drones, aplicaciones basadas en

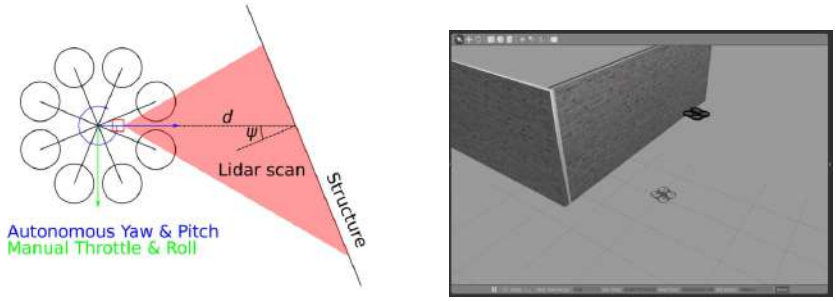
visión, entre otros algoritmos para definir la trayectoria del vehículo.

En el trabajo elaborado por Zamora *et al.* [11] se presenta una extensión del gimnasio *OpenAI* para robótica usando ROS y Gazebo. El gimnasio *OpenAI* es una herramienta utilizada en la investigación de aprendizaje por refuerzo. La publicación muestra una colección de seis entornos desarrollados en el simulador Gazebo para los robots: Turtlebot, el vehículo terrestre Erle-Rover y el dron Erle-Copter. Probando técnicas de aprendizaje por refuerzo y operando bajo las mismas condiciones virtuales, los vehículos recorren los circuitos utilizando la información del sensor LIDAR para realizar el entrenamiento. Las dos técnicas de aprendizaje por refuerzo utilizadas son Q-Learning y SARSA. En las pruebas se ejecuta la simulación 3000 episodios, en cada episodio se hace un máximo de 1500 iteraciones, lo que significa que el robot no se ha bloqueado. En cada iteración se elige una acción, se da un paso (se ejecuta una acción por un corto tiempo o distancia) y se reciben comentarios como retroalimentación para construir la siguiente acción que se tomará. En el caso particular del vehículo aéreo utilizan como piloto automático al software APM y proponen como trabajo a futuro la admisión de otros softwares de piloto automático como PX4 o Paparazzi. En la figura 2.4 se muestran las imágenes de los vehículos utilizados en el simulador recorriendo los distintos circuitos.

Figura 2.4: Vehículos Turtlebot, Erle-Rover y Erle-Copter en sus respectivos circuitos en el simulador.

En el estudio hecho por McAree *et al.* [12] se presenta al simulador Gazebo como el encargado de mostrar la implementación de un sistema de control en un dron para realizar la labor de inspeccionar estructuras. La operación de un dron cerca de una estructura puede llegar a ser una tarea desafiante debido a las condiciones ambientales que podrían poner a prueba la habilidad del piloto para dirigir el vehículo. La encomienda del sistema de control es encargarse de mantener una distancia establecida del objetivo a inspeccionar (ver figura 2.5) además de una pose relativa constante, esto permite al operador maniobrar el dron con mayor facilidad al rededor de la estructura, haciendo que el trabajo del operador se reduzca dejándolo libre para realizar otra tarea asignada para él como la recolección de datos.

El dron físico utilizado es un octorotor equipado con el sistema de control de vuelo *Pixhawk* corriendo el firmware *ArduCopter*, utiliza un escaner LIDAR para detectar la estructura, una Odroid XU3 corriendo Linux y el sistema ROS. En el trabajo la señal



(a) Diagrama representativo de la tarea de control.

(b) Simulación de la inspección de estructuras en Gazebo.

Figura 2.5: Inspección de estructuras utilizando drones con ROS y Gazebo.

del sistema de control sobreescribe los canales dos y cuatro del radio control correspondientes a los movimientos de cabeceo y guiñada del dron. En la simulación de Gazebo se utiliza un AR.Drone 2.0 y en los resultados se logra mantener una distancia mayor de 2 metros hacia la estructura. En la figura 2.5a se muestra el diagrama representativo de la tarea de control del sistema, el valor de d es controlado alternado el ángulo de cabeceo, ψ es controlada a cero alternando el ángulo de guiña del vehículo. El piloto mantiene control sobre el acelerador y el ángulo de alabeo. En la figura 2.5b se puede observar el comportamiento del dron al realizar la inspección de una estructura en el simulador Gazebo.

En lo realizado por Wang *et al.* [13] se desarrolla una plataforma de investigación de apoyo integrada con marcadores de referencia en 2D para la localización autónoma en interiores de un cuadrorotor parrot AR.Drone 2.0. El algoritmo utilizado para conocer la posición y postura del vehículo es el algoritmo *ArilTag*. ROS y Gazebo son considerados y seleccionados como marco básico debido a su generalidad y están simultáneamente integrados dentro de la plataforma. En la figura 2.6 se muestra un diagrama el flujo de trabajo del sistema desarrollado.

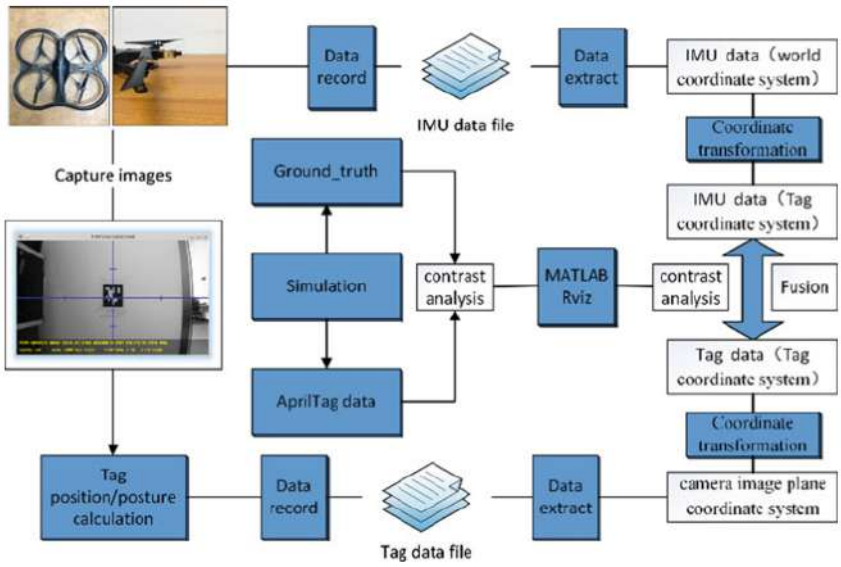


Figura 2.6: Flujo de trabajo del sistema desarrollado en las plataformas ROS y Gazebo.

Capítulo 3

Marco teórico

3.1. Vehículo Aéreo No Tripulado

3.1.1. Introducción a los VANT

Un vehículo aéreo no tripulado (VANT) es una aeronave que no lleva un operador humano, utiliza fuerzas aerodinámicas para proporcionar sustentación, puede volar de manera autónoma (a través de una computadora de vuelo abordo) y/o ser piloteado remotamente (por radio control) [14]. Los VANT son considerados robots aéreos y la tendencia en su desarrollo muestra que pasarán de ser robots operados a distancia a ser robots independientes y autosuficientes para realizar las tareas asignadas. Un hexacóptero o hexarotor es un tipo de VANT que está equipado de seis rotores distribuidos en los vértices de un armazón con forma de hexágono regular como muestra en la figura 3.1.

Figura 3.1: Modelo de un hexarotor.

Hoy en día el desarrollo de VANT de cuatro o más motores es llevado a cabo debido a que particularmente presentan ciertas ventajas sobre sus contra partes de menos de cuatro motores. Estas ventajas aparecen en términos de más potencia, más sustentación en el aire es decir más tiempo de vuelo, así también como más capacidad en el aumento de carga que puede levantar el vehículo [15]; sin embargo, presentan desventajas como lo es un mayor peso del vehículo y un mayor consumo de energía debido a la cantidad de motores.

Los VANT de cuatro hélices son de fácil manufactura, gran maniobrabilidad y

buena estabilidad, entre otra ventajas que ofrecen; no obstante, los modelos con seis hélices se vuelven una elección óptima al alcanzar mayores velocidades debido a los dos motores extras, llegando a alturas elevadas de manera más rápida, y al tener los seis motores separados 120 grados, uno o hasta dos motores pueden dañarse o dejar de funcionar completamente durante un vuelo y el vehículo puede ser capaz de aterrizar con seguridad por lo que son ideales cuando se trabaja con cámaras costosas. Al comparar los modelos de seis hélices contra los modelos de ocho hélices, ambos presentan las mismas ventajas [16].

Como se describe en [17] el movimiento vertical de estos vehículos se logra aumentando o disminuyendo el empuje total manteniendo por igual cada empuje indivual generado por cada motor. Los movimientos de avance, retroceso, izquierda, derecha y guiña se logran por medio de una estrategia de control diferencial en el empuje de cada motor. Para evitar la deriva de guiña que se genera en este tipo de vehículos por la reacción debido al torque de cada motor, el hexacóptero está configurado de tal manera que tres de sus propelas giran en sentido de las agujas del reloj y tres lo hacen en sentido antihorario como se muestra en la figura 3.2.

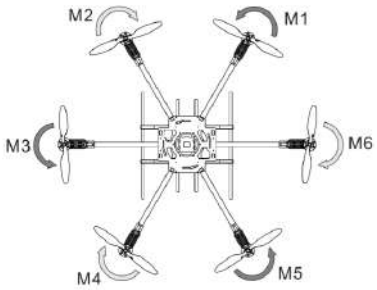


Figura 3.2: Sentido de giro de las hélices de un hexarotor[1].

En los países de habla inglesa alternativamente al término vehículo aéreo no tripulado (*UAV por sus siglas en inglés*) se utiliza el sustantivo *drone* [18], que en español significa literalmente zángano [18] y/o zumbido; no obstante, este mismo término se aludió también a aparatos de uso militar con aspecto similar al de un avión. En el año 2012 la Real Academia Española anexó el término dron[2] que se considera ahora una adaptación valida al español del sustantivo inglés *drone*.

[1]Fuente: https://www.helifreak.com/showthread.php?t=600192
[2]http://dle.rae.es/?id=ED2QqnQ

3.1.2. Ensamblado de la estructura

En el mercado actual las estructuras (cuerpo, marco, armazón o *frame* por su sustantivo en inglés) existen en una una gama muy amplia para los diferentes tipos de multirotores[3]. En esta sección se describirá la del hexacóptero modelo T960 de la línea Tarot que se muestra en la figura 3.3.

Figura 3.3: Estructura completa del Tarot T960.

Figura 3.4: Cubierta principal.

Este modelo se compone de una cubierta principal (figura 3.4) de seis ejes con un tamaño de 210x210x2.0mm elaborada de fibra de carbono, la cual tiene un peso ligero, resistencia a la fluencia, alta resistencia a la compresión y a los golpes. Los elementos que se muestran en la figuras 3.5a y 3.5b van colocados en la cubierta principal como se muestra en la figura 3.5c y sirven para sostener los brazos de la estructura.

(a) (b) (c)

Figura 3.5: Elementos de sostenimiento para los brazos de la estructura.

[3]https://www.fpvmodel.com/multi-rotors_fpv-frames_c5.html, https://www.aliexpress.com/popular/hexacopter-frame.html, http://ardupilot.org/copter/docs/choosing-a-frame.html, https://hobbyking.com/en_us/drones/frame-kits.html?___store=en_us, http://www.helipal.com/product_info.php?currency=SGD&products_id=10767

La estructura tiene seis brazos (nanotubos TL96010) de los cuales dos laterales son fijos y los cuatro restantes pueden ser plegados, esto con el fin de transportar el vehículo de manera más fácil. Los tubos son de carbono 3K y tienen un diámetro de 25mm, el diseño tiene una fuerza de sujeción fuerte y la fuerza de bloqueo es más alta en los tubos plegables que en los tubos fijos.

Figura 3.6: Brazos del hexacóptero.

Los asientos de los motores que se fijan en los extremos de cada brazo son de aluminio, cuentan con una base de fibra de carbono para la instalación de los ESC, así como con una placa adaptadora de motor de igual material que tiene un espaciado de agujeros para montaje de motor de 16mm, 19mm, 25mm y 27mm.

Figura 3.7: Asientos para motor.

El tren de aterrizaje va ensamblado en la parte inferior de la cubierta principal como se muestran en la figura 3.8, éste también es plegable para ser fácil de transportar.

Figura 3.8: Tren de aterrizaje.

En general el material con el que se encuentra elaborada la estructura del T960 es de aluminio y fibra de carbono (Toray, 3K), el peso de la estructura es de 1050 gramos; sin embargo, el vehículo tiene un peso de carga de vuelo de 3.5 kg. La configuración recomendada por el fabricante es:

- Motor sin escobillas multirrotor de la serie 5 TL100B08.
- 17-18 pulgadas multi-rotor Carbon Efficient pros paddle TL2812 / TL2815.
- 40A Multi Rotor ESC.
- Batería 6S 10000-15000MAH.
- Cuchillas multi-rotor 1855 Carbon Efficient TL2822 pros.

Los detalles técnicos se obtuvieron de [19] y algunas imágenes de páginas en internet[4]. El manual de ensamblado que proporciona el fabricante se puede encuentra en el apéndice A[5].

3.1.3. Hardware del VANT

Motores y Hélices

Los motores seleccionados para el dron son el modelo Quanum Serie MT 4012 480KV sin escobillas (figura 3.9) construidos por DYS, tiene como características principales[6] aquellas que se muestran en la tabla 3.1.

Figura 3.9: Motor Quanum Serie MT 4012 480KV.

Existen diferentes tipos y tamaños de hélices para drones, como ya se mencionó en la sección 3.1.1 la mitad del número de hélices de un dron giran en sentido horario, mientras que la otra mitad giran en sentido antihorario. El diámetro de las hélices se refiere al tamaño que tiene de punta a punta y se le conoce así por la circunferencia que se genera al girar. De la misma manera el grado con la que atacan el aire es importante

[4]Fuente: http://www.helipal.com/tarot-t960-hexacopter-frame-set.html
[5]Fuente: http://cyanscorpion.com/tarot-t810-t960–manual.html
[6]Fuente: https://hobbyking.com/es_es/quanum-mt-series-4012-480kv-brushless-multirotor-motor-built-by-dys.html

Quanum MT4012	
Peso (g)	266.00
Longitud	95.00
Ancho	50.00
Altura	70.00
Eje del Motor (mm)	4
C Diámetro (mm)	47.00
Longitud Total E (mm)	38.00
Células Lipo	4-6S
Kv (rpm / v)	50
Potencia máxima (W)	364.00
Max Amp (A)	16.00
Max Voltaje (V)	22.00
Imanes	42SH
Alambre	18AWG
Conector	3,5 mm de bala

Cuadro 3.1: Características del motor Quanum Serie MT 4012 480KV.

puesto que un aumento en el grado de inclinación supondrá un mayor empuje, aunque esto a su vez genere un mayor consumo de energía, siendo las hélices con un menor grado de inclinación capaces de girar más rápido [20].

Figura 3.10: Hélices Quanum 15x5.5L.

Para el vehículo se utilizaron las hélices Quanum elaboradas de fibra de carbono T-Style hélice 17x5.5 (CW / CCW), por ser del diámetro adecuado que menciona el fabricante del Tarot T960. En la tabla 3.2 se muestran las especificaciones del producto[7].

Figura 3.11: Montaje de motores MT4012 en el hexacóptero.

[7]Fuente: https://hobbyking.com/es_es/quanum-carbon-fiber-t-style-propeller-17x5-5-cw-ccw-2pcs-1.html

Hélices Quanum 17x5.5	
Longitud	17 in
Pitch	5.5 in
Diámetro del eje	4 mm
Placa de cubierta centros de orificios	12 mm
Hub Espesor	3.5 mm
Peso	27 g
Rotación	CW / CCW

Cuadro 3.2: Características de las hélices Quanum T-Style 17x5.5.

En la figura 3.11 se puede observar un motor ya instalado en el asiento de un brazo del hexacóptero.

Baterías

Para suministrar la energía necesaria a los motores, computadora de vuelo y otros dispositivos que requieran ser montados en un dron se utilizan las baterías LiPo (abreviatura de litio y polímero) que son baterías recargables y actualmente son las más utilizadas. Éstas difieren entre si por el número de celdas con las que cuentan, una celda puede entenderse como una pequeña batería que repetidas veces forma parte de una batería completa. El voltaje que suministran es de acuerdo al número de celdas que posean [21], las baterías LiPo tienen celdas de 3.7 voltios y 4.2 voltios cuando están totalmente cargadas. En el cuadro 3.3 se muestra cuanto voltaje suministra una batería de acuerdo al número del celdas que tenga.

Baterías LiPo			
Nombre	Celdas	V por celda	V total
1S	1	3.7V	3.7V
2S	2	3.7V	7.4V
3S	3	3.7V	11.1V
4S	4	3.7V	14.8V
5S	5	3.7V	18.5V
6S	6	3.7V	22.2V

Cuadro 3.3: Voltajes de las baterías LiPo.

El nombre de cada batería lo adquiere por el número de celdas que tenga y la letra **S** es solo un indicativo de que las baterías son conectadas en serie. Otro aspecto importante es la capacidad que tienen para brindar energía, la cual se maneja en miliamperios por hora (mAh). Por ejemplo una batería de 8000 mAh se descarga completamente en media hora si tuviera una carga de 16000 mA, en una hora si tuviera una carga de 8000 mA y en dos horas si tuviera una carga de 4000 mA. Un aspecto poco tomado en cuenta pero igual de importante es la tasa de descarga, que se indica mediante la letra

C y señala la rapidez con la que una batería puede descargarse.

Figura 3.12: Baterías LiPo montadas en el vehículo.

Para este trabajo se utilizaron las baterías zippy de 8000 mAh sus especificaciones[8] se muestran en el cuadro 3.4. En la figura 3.12 se muestra el montaje que por cuestiones de diseño se hizo en la parte inferior de la cubierta principal del vehículo.

Zippy 8000 mAh 3S	
Capacidad mínima	8000mAh
Configuración	3S1P 11.1v 3Cell
Constante de descarga	30C
Pico de descarga (10seg)	60C
Peso	565g
Tamaño	167 x 69 x 24mm
Tipo de enchufe	JST-XH
Tapón de descarga	XT90

Cuadro 3.4: Especificaciones de la batería Zippy 8000 mAh.

Controlador de velocidad electrónico

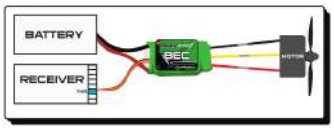
El controlador de velocidad electrónico (*ESC por sus siglas en inglés*) tiene la tarea de suministrar la corriente a un motor, aislando la señal PWM proveniente de la computadora de vuelo. En un VANT el controlador recibe la corriente para el motor directamente de la placa de distribución de potencia que se describirá más adelante. En este trabajo se utilizará el modelo Turnigy MultiStar de 40A que se muestra en la figura 3.13a, su diagrama de conexión se muestra en la figura 3.13b, las especificaciones técnicas[9] se muestran en la tabla 3.5.

Este tipo de controladores para drones son capaces de emitir cierto número de pitidos que sirven para configurar o indicar parámetros del mismo, las configuraciones e indicaciones en base a pitidos que emite cada controlador varia de acuerdo a su marca

[8]Fuente: https://hobbyking.com/es_es/zippy-flightmax-8000mah-3s1p-30c-lipo-pack-xt90.html?___store=es_es

[9]Fuente: https://hobbyking.com/es_es/turnigy-multistar-40a-v2-slim-blheli-multi-rotor-brushless-opto-esc-2-6s.html?___store=es_es

(a) ESC MultiStar 40A



(b) diagrama de conexión

Figura 3.13: Controlador de velocidad electrónico Turnigy MultiStar de 40A.

ESC Turnigy MultiStar 40A	
Corriente constante	40A
Voltaje de entrada	2-6 células LiPoly
BEC	Ninguno (OPTO)
PWM	8 KHz Max
Frecuencia	20-500Hz
RPM	240,000rpm para 2 Pole motores sin escobillas
PCB Tamaño	45 mm x 18 mm
Los tapones de descarga	Hombre Bala 3,5 mm conector
Motor Tapones	conector hembra de 3,5 mm de bala
Peso	35 g

Cuadro 3.5: Especificaciones del ESC MultiStar 40A.

y modelo por eso es recomendable siempre revisar las especificaciones del producto. Por ejemplo para entrar en modo de programación del modelo Turnigy mencionado se acopla el conector JR en el canal de control del acelerador del receptor, luego se enciende el transmisor y se debe mover la palanca del acelerador a la posición más alta. En este momento se conecta la fuente de alimentación principal a ESC. Se deben esperar 5 segundos y es entonces que se escucharán 4 pitidos, eso significa que ya se entró en el modo de programación.

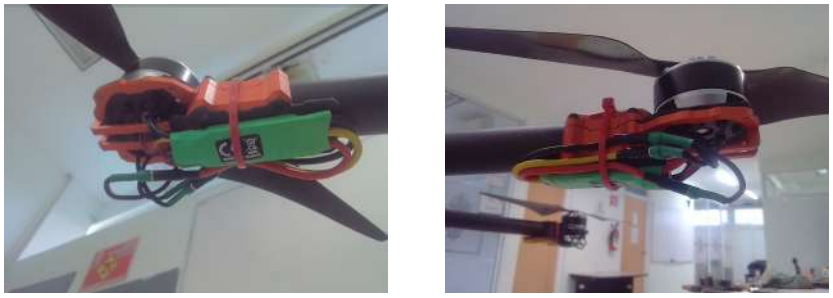


Figura 3.14: Montaje de motores MT4012 en el hexacóptero.

Placa de distribución de potencia

La placa de distribución de potencia (*PDB por sus siglas en inglés*) es la encargada de distribuir la energía a todos los ESC del dron facilitando un cableado ordenado. Para este trabajo se utilizó la placa que se muestra en la figura 3.15.

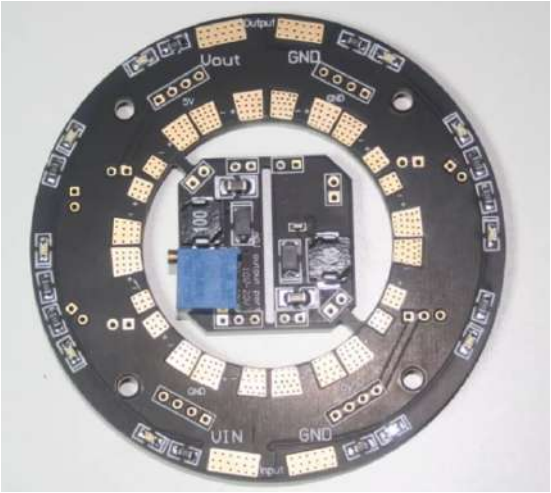


Figura 3.15: Placa de distribución de potencia.

El número total de motores en el hexacóptero son seis y de acuerdo a la información obtenida, cada motor Quanum MT4012 puede alcanzar un pico máximo de corriente de 16A, por lo que en total en el vehículo se puede alcanzar hasta una corriente de 96A, la placa disponible es capaz de soportar hasta 120A por lo que resulta adecuada para el diseño. El voltaje de entrada proveniente de las baterías LiPo de 3S es de 11.1V; sin embargo, la placa cuenta con dos reguladores voltaje uno ajustable y otro de 5V por si se necesita montar en el dron algún dispositivo que requiera de un voltaje menor al que proporcionan las baterías.

Placa distribución de potencia 120A	
Corriente constante	40A
Salida máxima	120A
Potencia máxima de entrada	hasta 6S (22.2V)
Salida BEC 1	5V 3A fija (15Watts max)
Salida BEC 2	10 20V ajustable (36Watts max)
Diámetro exterior	77mm
Diámetro interior	40mm
Orificios de montaje	45 x 45mm
Peso	23g (incluidos los pines del conector)

Cuadro 3.6: Especificaciones de la placa PDB.

En la tabla 3.6 se muestran las especificaciones[10] de la placa de distribución de potencia. Como se puede observar la placa puede utilizarse con baterías de hasta 6S y es capaz de soportar un máximo de corriente de 120A. En la figura 3.16 se muestra la placa de distribución montada en la parte inferior de la cubierta principal del dron, con las conexiones ya soldadas hacia los seis ESC de cada motor.

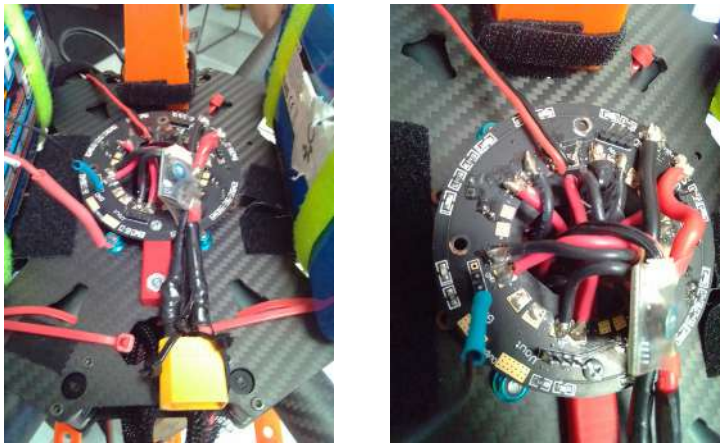


Figura 3.16: Montaje de la PDB en el cubierta.

3.1.4. Computadora de vuelo

El piloto automático es un sistema utilizado para guiar aeronaves sin la necesidad de una asistencia constante por parte de operadores humanos. Este sistema no remplaza completamente al operador, más bien, lo ayuda a automatizar el proceso de guiar y controlar la aeronave [22].

En los drones el sistema del piloto automático que también se conoce como controladora de vuelo, ésta incluye sensores como: giroscopio, acelerómetro, magnetómetro, barómetro, GPS para la determinación del estado; procesadores a bordo para usos de estimación y control; además de circuitos periféricos para servo y comunicaciones de módem [23]. Algunas de las tareas principales de la controladora sobre el vehículo son mantener la actitud, altitud, velocidad, el despegue y aterrizaje automático, mantener el ángulo de alabeo y cabeceo. El sistema establece comunicación con la estación de control terrestre con el fin de cambiar el modo de vuelo, enviar la información de posición actualizada proveniente del satélite GPS, recibir entradas de control a los motores, enviar de información de sensores, etc.

Para este trabajo se utilizó la computadora de vuelo *HK32Pilot* que se muestra en la figura 3.17a, la cual tiene un tamaño de 81x44x15mm y un peso de 33.1g, utiliza modulación PPM para el radio control, incluye ademas un buzzer, un botón de seguri-

[10]Fuente: https://hobbyking.com/es_es/universal-12-way-120a-multirotor-power-distribution-hub-w-leds-dual-becs.html?___store=es_es

(a) Computadora de vuelo HK32Pilot.

(b) Dispositivos y conexiones de la computadora de vuelo.

Figura 3.17: Computadora de vuelo HK32Pilot.

dad, un módulo de poder 10s con conectores XT60 y conectores extras para GPS y el módulo de la brújula.

Especificaciones:

- Procesador ARM Cortex de 32 bits de 168MHz.
- 14 salidas PWM/servo.
- Memoria RAM 256 KB.
- Memoria Flash 2 MB.
- NuttX RTOS.
- 5 x UART.
- I2C.
- SPI.
- 2 x CAN.
- Lector de tarjeta SD.
- Entradas ADC de 3.3 y 6.6V
- Puerto externo microUSB.

Sensores:

- Giroscopio ST Micro L3GD20 3-axis 16-bit.
- Acelerómetro y magnetómetro ST Micro LSM303D 3-axis 14-bit.
- Acelerómetro y giroscopio Invensense MPU 6000 3-ejes
- Barómetro MEAS MS5611.

En la figura 3.17b se muestra un esquema general de los dispositivos que se conectan a la computadora de vuelo, desde los motores, baterías, módulo de poder, telemetría, módulo receptor de radio control, módulo PPM, GPS, buzzer y botón de seguridad. En la figura 3.18 se muestra la computadora ya montada en el vehículo en la parte superior de la cubierta.

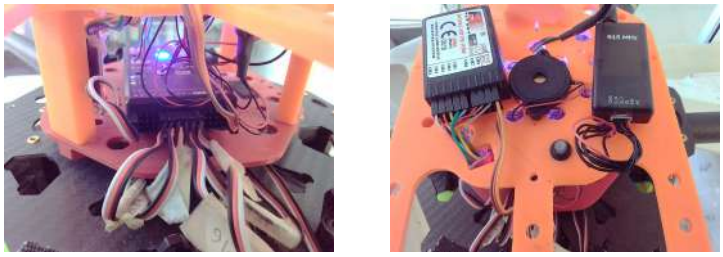
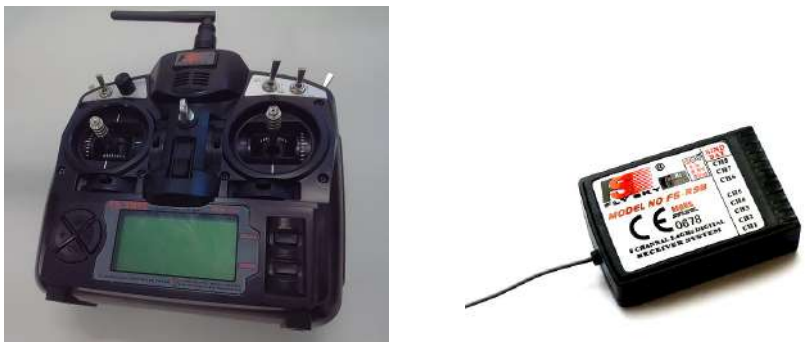


Figura 3.18: Montaje final de la computadora HK32Pilot en el hexacóptero.

3.2. Configuraciones y calibraciones del vehículo

3.2.1. Radio control y módulo receptor

El sistema de control remoto o radio control RC sirve para manejar el vehículo de manera manual. Para este trabajo se utilizó el radio control modelo FS-TH9X que se muestra en la figura 3.19a. El módulo receptor modelo FS-R9B que se instala en el vehículo se muestra la figura 3.19b.



(a) Control remoto FS-TH9X. (b) Módulo receptor FS-R9B.

Figura 3.19: Sistema de control remoto.

El radio control tiene una salida de 9 canales; sin embargo, para este proyecto se utilizaron solo seis canales de salida, cuatro para enviar las señales referentes al acelerador, alabeo, cabeceo y guiñada, y dos canales más para seleccionar el modo de vuelo del vehículo. En el apéndice B se muestran los nombres de los controles del transmisor. El control debe ser configurado de acuerdo a las necesidades del vehículo que controla, algunas de las configuraciones que se establecieron fueron:

- Selección de modo: el control puede guardar en memoria 8 configuraciones generales, que pueden ser seleccionadas y configuradas independientemente una de la otra por el usuario.
- Selección del tipo: establece el tipo de programación utilizada para este modelo. Los tipos de modelos son helicóptero, avión y planeador.

- Selección de la modulación: selecciona entre el tipo de modulación PPM o PCM. Para este trabajo se utiliza la modulación PPM.
- Selección de la configuración de las palancas: mapea las palancas del control con las señales referentes al acelerador, alabeo, cabeceo y guiñada. Existen cuatro configuraciones distintas que se seleccionan por comodidad del usuario, el modo 1 y modo 2 son las más utilizadas por diversos usuarios.

Como el tipo de vehículo a utilizar es un hexacóptero se seleccionó la configuración de helicóptero para el radio control, de la cual se realizaron las configuraciones siguientes:

- Inversión de la señal de los canales: las señales de las palancas del control como el acelerador, alabeo, cabeceo y guiñada pueden invertirse para conveniencia del usuario. La inversión de las señales se puede realizar para todos los nueve canales.
- Configuración de la curva del acelerador: se puede configurar la respuesta de la palanca referente a la acelación del vehículo, para que los cambios de velocidad no sean lineales y bruscos. Esta configuración puede ser establecidad de diferente manera para cada modo de vuelo del vehículo.
- Configuración de la curva del cabeceo: se puede configurar la respuesta de la palanca referente al cabeceo del vehículo, para que los cambios de velocidad no sean lineales y bruscos. Esta configuración puede ser establecidad de diferente manera para cada modo de vuelo del vehículo.

(a) Selección del tipo.

(b) Selección de la modulación.

(c) Configuración de las palancas.

(d) Inversión de la señal de los canales.

(e) Configuración de la curva de aceleración.

(f) Configuración de la curva de cabeceo.

Figura 3.20: Configuraciones realizadas en el control remoto FS-TH9X.

En la figura 3.20 se muestran los menus disponibles por el control para realizar las configuraciones. El módulo receptor se encarga de enviar la información recibida de los 9 canales del radio control al módulo PPM, el cual va conectado a la computadora de vuelo como se muestra en la figura 3.21b.

(a) Módulo PPM.

(b) Conexión del módulo PPM con la computadora de vuelo.

Figura 3.21: Módulo PPM.

Se debe realizar un emparejamiento entre el módulo receptor y el control remoto para que estos dos dispositivos se transmitan información, este emparejamiento se logra mediante los pasos siguientes:

- Conectar el cable negro que forma un puente en la posición BIND del módulo receptor y conectar a BAT la alimentación (la alimentación puede ser conectada a cualquier otro canal).
- El módulo emitirá una luz de color rojo en su interior que parpadeará constantemente, este parpadeo indica que el módulo se encuentra disponible para establecer conexión con cualquier radio control
- Para establecer conexión con el control remoto se debe presionar el boton que se encuentra en la parte posterior del mismo y luego prenderlo.
- Se debe esperar hasta que la luz del módulo receptor se quede estable y no parpadeé. Cuando la luz deja de parpadear y se queda fija indica que ya se establecio la conexión (no se debe dejar de presionar el boton del control hasta que la luz del receptor esté fija).
- Se procede a desconectar la alimentación del módulo receptor, quitar el cable negro del BIND y se procede a apagar el radio control.

El emparejamiento entre los dispositivos solo se debe realiza una vez, debido a que el módulo receptor debe ser siempre capaz de reconocer al radio control cada vez que se encienda el vehículo.

3.2.2. Estación de control terrestre.

La estación de control terrestre (*GCS por sus siglas en inglés*) es un programa que se ejecuta en una computadora y se comunica inalambricamente con el vehículo mediante telemetría. La estación en tierra muestra datos en tiempo real sobre el rendimiento y la posición de dron con instrumentos muy parecidos a los de una cabina de un avión real, sirve para planear el vuelo autónomo del vehículo controlando su despegue, aterrizaje y misión, esto con ayuda del GPS y de la interfaz que vizualiza un mapa el cual muestra la posición del vehículo, la trayectoria de vuelo, y los puntos del camino [24]. De igual modo, la estación sirve también para establecer parámetros principales de configuración en la computadora de vuelo que son indispensables para que el vehículo pueda realizar los vuelos y se garantice la seguridad del mismo, por mencionar algunas configuraciones:

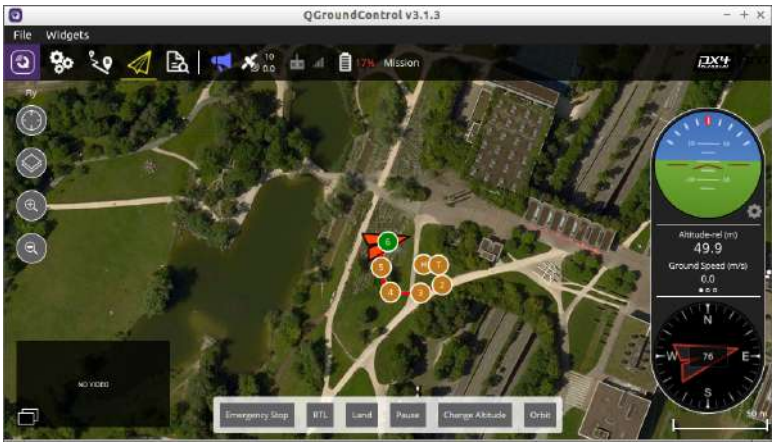


Figura 3.22: Estación de control *QGroundControl*.

- Descarga del firmware en la compuadora de vuelo.
- Establecimiento del tipo de estructura del vehículo.
- Calibración de la radio.
- Calibración de sensores.
- Modos de vuelo en el control RC.
- Configuración de la energía.
- Calibración de ESC.
- Configuración de seguridad.

Para este trabajo se utilizó la estación de control terrestre *QGroundControl* que es capaz de ejecutarse en Windows y en plataformas Linux. El firmware que se cargó al

Figura 3.23: Descarga del firmware en *QGroundControl*.

HK32Pilot fue el *PX4* debido a que el *ArduPilot* presentó algunos problemas al realizar ciertas configuraciones.

La computadora de vuelo está diseñada para poder trabajar con diferentes tipos de vehículos aéreos, desde helicópteros, aviones de una sola ala, hasta drones con distinto número de motores, incluso es capaz de trabajar con vehículos terrestres. Por este motivo a la computadora de vuelo se le debe de configurar que tipo de estructura va a manejar, que para este trabajo es un hexacóptero. En la figura 3.24 se muestra la pantalla de la interfaz para realizar esta configuración.

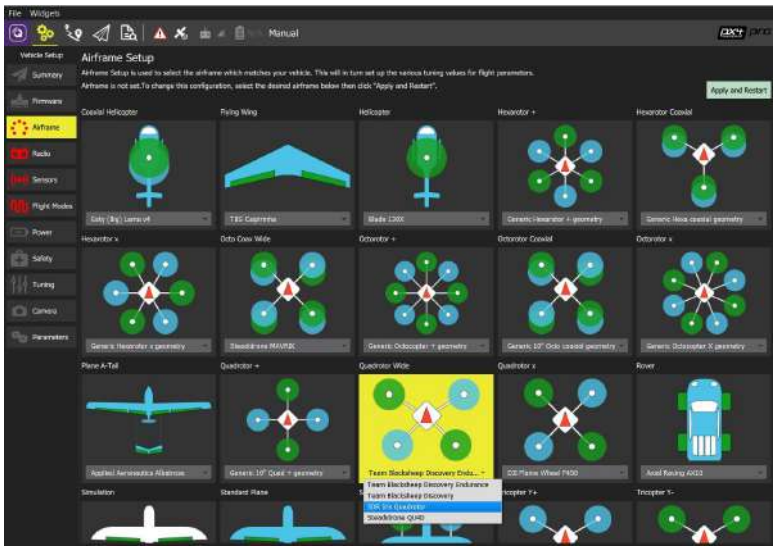


Figura 3.24: Configuración del tipo de estructura en *QGroundControl*.

Con el fin de que la computadora de vuelo reconozca adecuadamente los valores referente a los movimientos de balanceo, cabeceo, guiñada, acelerador que provienen de las palancas del control RC, se debe hacer una calibración pertinente. Para este momento ya se debe de haber establecido una conexión entre el modulo receptor de telemetría con el control RC. En la figura 3.25 se muestra la pantalla de calibración de *QGroundControl*. Una vez iniciada la configuración, el programa guía paso a paso la serie de movimientos que se deben hacer manual mente en el control RC, una vez terminada la configuración el la computadora de vuelo ya reconoce acertadamente los comandos que envía el control. Cabe mencionar que para hacer adecuadamente esta

calibración es importante indicar en que modo se tiene configuradas las palancas del radio control (modo 1 o modo 2).

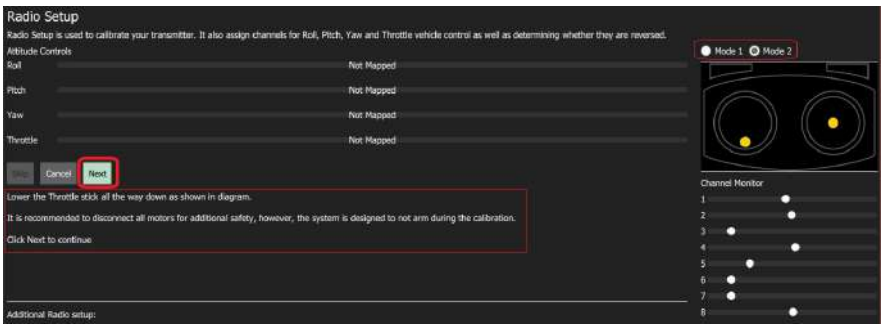
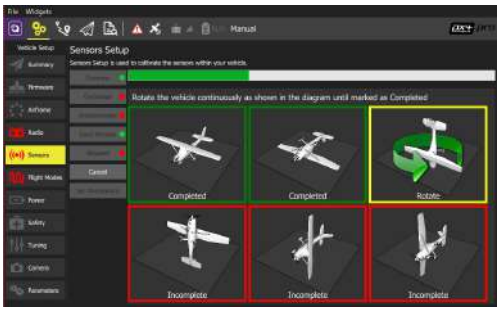
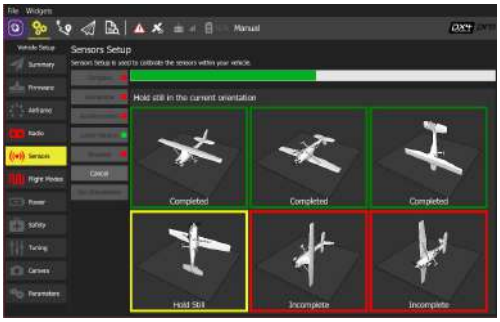


Figura 3.25: Calibración del radio control en *QGroundControl*.

Inicialmente la computadora de vuelo aun no reconoce adecuadamente la información proveniente de los sensores como el acelerómetro, giroscopio y brújula para que esta pueda ejercer un control adecuado y poder estabilizar el vehículo cuando se comience el vuelo, es por este motivo que se debe hacer una calibración pertinente a cada sensor, considerando que la placa ya se encuentra montada en el vehículo y ésta no va a ser cambiada de posición, un cambio de lugar de la placa requeriría nuevamente de una calibración de sensores.



(a) Calibración de la brújula en *QGroundControl*.



(b) Calibración del acelerómetro en *QGroundControl*.

Figura 3.26: Calibración de sensores en *QGroundControl*.

El programa *QGroundControl* realiza la calibración de cada sensor de manera individual. La calibración de la brújula se realiza mediante una serie de movimientos o giros del vehículo que se indican en la pantalla del programa como se muestra en la figura 3.26a. La calibración del acelerómetro requiere de poner al dron en diversas posiciones que se indican igualmente en la pantalla del programa como se muestra en la figura 3.26b. Una vez realizadas todas las rotaciones y posiciones indicadas el vehículo ya debe tener un nivel de horizonte bien definido, en caso contrario, se puede realizar la calibración de horizonte, dejando el vehículo en una posición normal.

La configuración de los modos de vuelo permiten que utilizando los interruptores del control de RC se pueda cambiar de modo de vuelo en el vehículo. Esto puede configurarse de acuerdo a las necesidades del usuario, para este proyecto solo se utilizó el modo de vuelo estabilizado y el modo aterrizaje. En la configuración de energía que se muestra en la figura 3.27 se deben indicar parámetros de la batería como el número de celdas, el voltaje máximo y mínimo por celda. Un parámetro importante de esta sección es la calibración de los controladores de velocidad electrónicos del vehículo, sin esta calibración los valores máximos y mínimos de *pwm* son distintos en cada motor lo que ocasiona que al prender el vehículo se observe una diferencia en las velocidades de giro de cada uno. Esta calibración también puede realizarse individualmente por cada controlador de velocidad; sin embargo, es mucho más rápido realizar la que ofrece el programa *QGroundControl*.

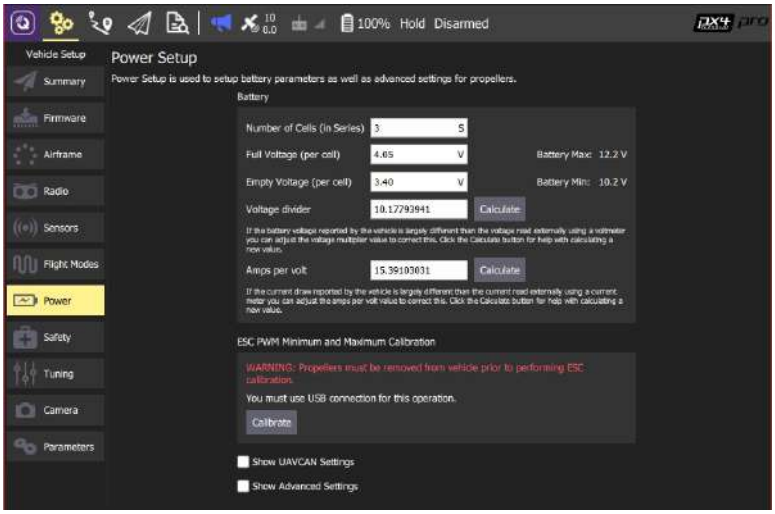


Figura 3.27: Configuración de energía en *QGroundControl*.

Todas las calibraciones y configuraciones mencionadas anteriormente son necesarias para que el vehículo pueda realizar los vuelos de manera adecuada. Las calibración de los canales del radio control y la calibración de los controladores de velocidad de los motores se deben realizar cada vez que el vehículo muestre problemas en el despegue, como por ejemplo que no pueda levantar el vuelo o que el despegue no lo pueda realizar verticalmente. Si el vehículo sufre una caída es recomendable que se realice nuevamente una calibración de sensores.

QGroundControl ofrece configuraciones de seguridad para que las misiones del vehículo se puedan realizar de manera más segura, como por ejemplo se puede seleccionar un nivel de batería bajo para que alerte al usuario de que seguir realizando el vuelo puede ser peligroso. Si el vehículo pierde su conexión con el control de telemetría después de cierto tiempo se puede configurar para que realice un regreso a casa, este regreso a casa tiene como objetivo la posición GPS del despegue. Se puede establecer una

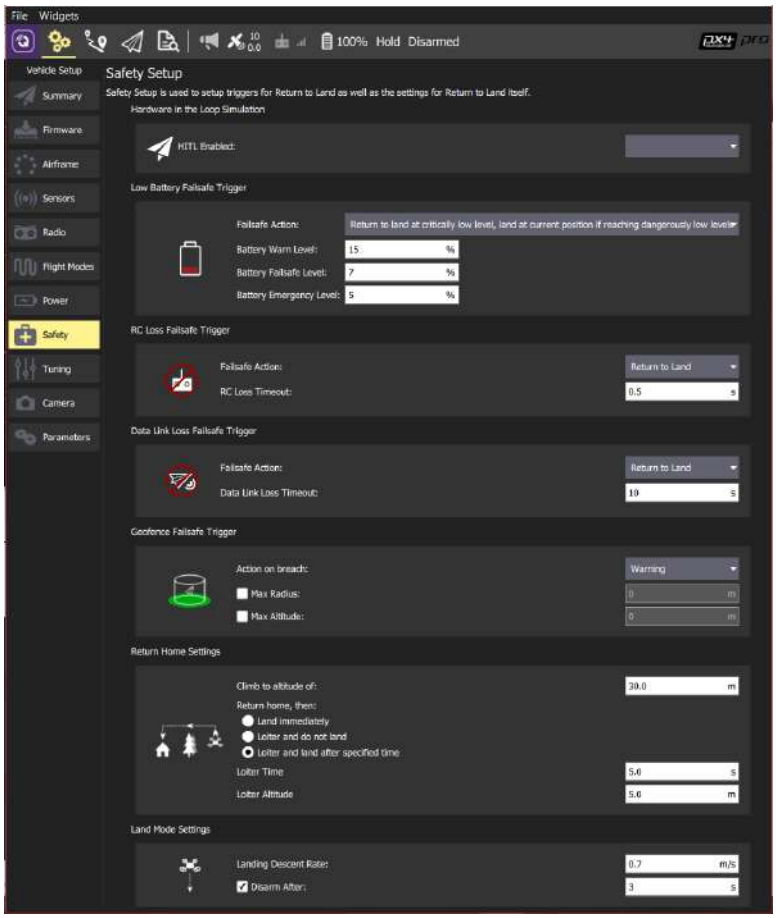


Figura 3.28: Configuración de seguridad en *QGroundControl*.

geocerca para que el radio de vuelo del vehículo no sobre pase lo estipulado y se mande una advertencia en caso de que suceda. El regreso a casa se puede configurar para que el vehículo una vez alcanzada la posición de casa realice un aterrizaje, o simplemente para que regrese y se quede volando en la ubicación. De igual forma, con motivo de seguridad al regresar a casa el vehículo puede ascender a una altura determinada con el fin de evitar obstáculos en su regreso. Una configuración de seguridad principal es la de aterrizaje, se debe establecer una velocidad de aterrizaje adecuada para que el vehículo no sufra algún daño, esta velocidad se muestra en m/s, de la misma manera se puede establecer que el vehículo se desarme o no cuando ya haya aterrizado. En la figura 3.28 se muestra la pantalla para la configuración de seguridad del *QGroundControl*.

Para realizar las configuraciones y calibraciones mencionadas anteriormente la computadora de vuelo se conecta mediante USB a la estación de control terrestre. La comunicación inalámbrica entre la estación de control terrestre y la computadora de vuelo utiliza el sistema de telemetría que se muestra en la figura 3.29. Este mismo sistema de comunicación es el que se utiliza posteriormente para cargar programas en la computadora de vuelo utilizando ROS, como se explicará más adelante.

(a) Conexíon de telemetría al HK32pilot.

(b) Conexión de telemetría a la estación de control.

Figura 3.29: Sistema de telemetría.

3.2.3. Armado del vehículo

El armado es un término que se refiere a prender o activar el vehículo, realizando un movimiento de las palancas del control, para que se prendan los motores y pueda comenzar el despegue. Este movimiento varía de acuerdo al dron que se utilice, para realizar el armado del vehículo utilizado se requirió de una configuración previa del radio control.

Existen 4 modos de configurar las palancas del control remoto, por lo regular los más utilizados por gente que se dedica al vuelo de drones son los modos 1 y 2. Cada modo configura de una manera distinta las dos palancas del control para realizar los movimientos de aceleración, alabeo, cabeceo y guiñada del dron, como se muestra en la figura 3.30.

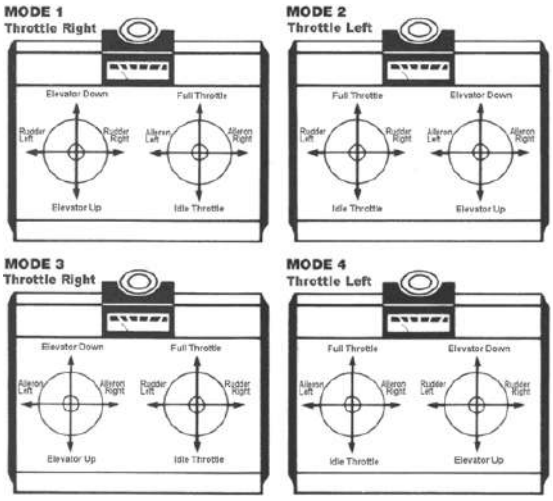


Figura 3.30: Tipos de configuraciones de las palancas en el radio control.

Para lograr el armado se estableció en modo 2 la configuración de las palancas del control y se invirtió la señal de guiña (aunque el modo y la inversión de la señales son opcionales), a partir de esto se debe realizar lo siguiente:

- Conectar a alimentación el vehículo, y esperar a que el GPS establezca una cone-

xión, lo cual se observa cuando el *HK32Pilot* emite una luz de color verde.

- El movimiento de armado en el control es haciendo una L con la palanca izquierda, (movimiento de palanca de acelerador hacia abajo y movimiento de palanca de guiña hacia la derecha) para este momento la computadora de vuelo emitirá una serie de pitidos hasta emitir uno constante y cuando este último pitido suene se debe presionar el botón de seguridad.
- El vehículo se arma, y los motores empiezan a girar a su velocidad mínima.

Para desarmar el vehículo se debe de realizar el movimiento de L pero a la izquierda (movimiento de palanca de acelerador hacia abajo y movimiento de palanca de guiña hacia la izquierda).

3.3. Fundamentos de Gazebo

El desarrollo de Gazebo comenzó en el 2002 en la Universidad del Sur de California. Los creadores originales fueron Dr. Andrew Howard y su estudiante Nate Koenig. Nate continuó el desarrollo del simulador mientras completaba su PhD. En 2009, John Hsu, integró ROS y el robot PR2 a Gazebo, el cual se convirtió en una de las herramientas importantes usadas en la comunidad ROS. Desde 2012 a la fecha, la Open Source Robotics Foundation (*OSRF por sus siglas en inglés*) es la administradora del proyecto Gazebo y continua su desarrollo con el apoyo de una comunidad activa diversa.

3.3.1. Descripción del simulador

Gazebo es un simulador dinámico 3D con la habilidad de simular precisa y eficientemente poblaciones de robots en entornos interiores y exteriores ofreciendo simulaciones físicas con alto grado de fidelidad [25]. En este simulador es posible desarrollar diseños propios de modelos describiéndolos en el formato de descripción de simulación (*SDF por sus siglas en inglés*). De igual manera es posible tener en el simulador robots comerciales muy utilizados en robótica como lo son el PR2, Youbot, y Turtlebot (ver figura 3.31) y drones como el Quadcopter, Parrot Bepod 2 y ARdrone.

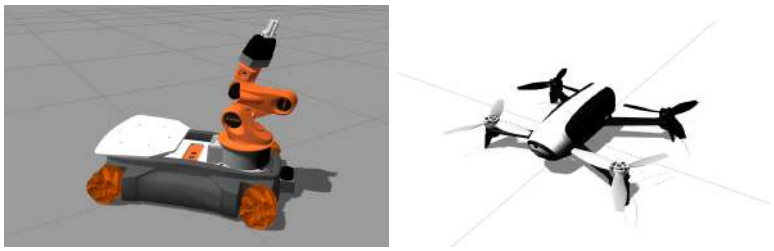


Figura 3.31: Modelos de robots en Gazebo.

Para proporcionar a la simulación más realismo la biblioteca de física de Gazebo se ha integrado con cuatro motores de física de código abierto: Open Dynamics Engine (*ODE por sus siglas en inglés*), Bullet, Symbody y Dynamic Animation and Robotics Toolkit (*DART por sus siglas en inglés*).

A continuación se presenta un listado de información esencial donde se obtuvieron datos porcentuales acerca del simulador [26].

- API principal: 80 % C++.
- Razón príncipal para adopción: 53 % mejor herramienta sobre evaluación, 20 % software actualmente usado en el laboratorio, 20 % herramienta oficial para trabajo, 7 % libre acceso.
- Mayormente usado en Estados Unidos (33 %).

- Usado principalmente para: 33 % robótica móvil, 27 % robótica de servicio, 20 % robótica humanoide.
- Robots simulados principalmente: 40 % Atlas, 33 % plataforma personalizada, 27 % vehículo de ruedas, 27 % multirotores, 27 % turtlebot, 20 % PR2.
- Middleware usado principalmente: 93 % ROS.

3.3.2. Instalación del simulador

Gazebo está basado en Linux y es ampliamente usado en Ubuntu, en la figura 3.32 se muestra la compatibilidad para realizar la instalación de las diferentes versiones del simulador en las diferentes versiones de Ubuntu.

Figura 3.32: Compatibilidad de Ubuntu y Gazebo[11].

Los requerimientos del simulador son:

- Contar con una Unidad de Procesamiento Gráfico (*GPU por sus siglas en inglés*).
- Una Unidad de Procesamiento Central (*CPU por sus siglas en inglés*) de al menos 5 núcleos.
- Al menos 500MB de espacio libre en memoria.
- Ubuntu Trusty o posteriores instalados.

Si se cuenta con Ubuntu Trusty el simulador puede ser instalado desde su versión 3 hasta su versión 7, el mismo puede ser descargado directamente desde su página

[11]Fuente: http://gazebosim.org/

oficial[12], o haciendo una instalación alternativa paso a paso[13] tipeando lo siguiente en una terminal:

```
$ sudo sh -c 'echo "deb http://packages.osrfoundation.org/
gazebo/ubuntu-stable `lsb_release -cs` main" > /etc/apt/
sources.list.d/gazebo-stable.list'

$ wget http://packages.osrfoundation.org/gazebo.key -O
- | sudo apt-key add -

$ sudo apt-get update

$ sudo apt-get install gazebo7
```

Si se genera algún tipo de error en la instalación es muy probable que la versión de Ubuntu con la que dispone el equipo no sea compatible con la versión de Gazebo que se desea instalar como se mencionó anteriormente.

Para realizar la instalación en otros sistemas operativos diferentes de Ubuntu, visitar la página[14].

3.3.3. Descripción del mundo

Gazebo puede ser ejecutado abriendo una termial del sistema y tipeando:

$ gazebo

Este comando ejecutará a Gazebo como servidor y como cliente por lo que se abrirá su interfaz gráfica, en donde se puede observar el mundo que el simulador lanza por defecto.

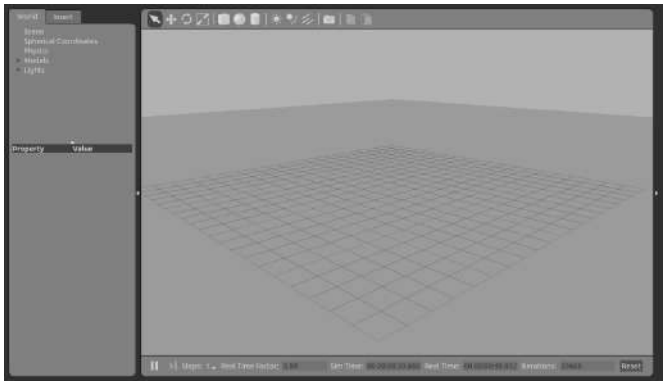


Figura 3.33: Mundo perteneciente al archivo empty_world.world

La interfaz gráfica muestra un mundo que contiene solamente un plano de piso y luz, el mundo puede ser modificado añadiendo todos los elementos con que dispone el simulador como modelos de robots, estructuras, u objetos que formarán parte del escenario de trabajo para realizar pruebas.

[12]Fuente: http://gazebosim.org/download
[13]Fuente: http://gazebosim.org/tutorials?tut=install_ubuntu&cat=install
[14]Fuente:http://gazebosim.org/tutorials?cat=install

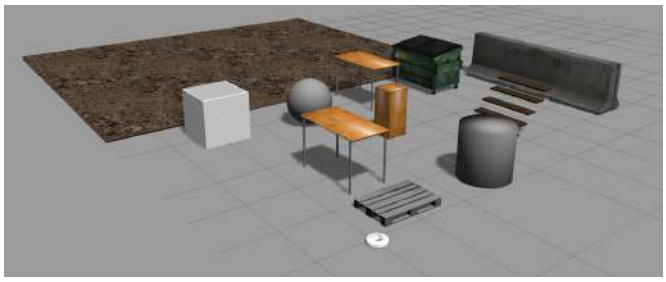


Figura 3.34: Objetos disponibles en Gazebo.

El mundo por defecto que lanza el simulador pertenece al archivo *empty_world.world* a este tipo de archivos se les conoce como archivos mundo y su extensión siempre es *.world*. Los archivos mundo contienen la descripción de todos los elementos de una simulación incluidos robots, luces, sensores y objetos. Así mismo pueden ser usados para controlar algunos aspectos del motor de simulación como la gravedad o el tiempo de simulación.

El archivo *.world* se encuentra escrito en formato SDF y puede ser creado y modificado usando cualquier editor de texto. Gazebo dispone de mundos ejemplo que se encuentran en la instalación por defecto: *usr/share/gazebo(version)/worlds*. Estos archivos pueden ser ejecutados tipeando en una terminal:

```
$ gazebo archivo.world
```

3.3.4. Descripción de un modelo

Dentro del mundo de Gazebo se encontrarán los modelos los cuales tienen la tarea de describir a cualquier tipo de objeto físico que se quiera simular, especificando sus propiedades dinámicas, cinemáticas y visuales. El modelo puede representar desde suelos y estructuras, hasta objetos muy simples y robots muy complejos.

Estructura de archivos de un modelo

Gazebo provee modelos en su base de datos en línea que pueden ser insertados en cualquier simulación, sin ningún problema. Cualquier modelo ya creado o por desarrollar cuentan con la siguiente estructura de archivos.

- Modelo1
  - Archivo *model.config* y contiene información básica sobre el modelo como su nombre, la versión SDF, autor y descripción del modelo, etc.
  - Archivo *model.sdf* descripción del modelo escrito en formado SDF.
  - Directorio *meshes* para todo los archivos COLLADA y STL.

- Directorio *plugins* es un directorio opcional contiene todos los ejecutables que le sirven al modelo.
- Directorio *materials* es un directorio opcional que contiene a su vez dos directorios
  - Directorio *textures* contiene archivos de imagen (jpg,png,etc.)
  - Directorio *scripts* contiene scripts de material OGRE.

Tanto el archivo *.world* como el archivo *model.sdf* se encuentran escritos en formato SDF por esta razón se explicará con más detalle este tipo de formato.

3.3.5. El formato SDF

Gazebo utiliza el formato SDF para describir entornos y objetos a simular, el cual está basado en XML; por lo tanto, los archivos *.world* como los archivos *model.sdf* utilizan este formato.

Utilizando el lenguaje SDF se pueden describir con precisión todos los aspectos de los modelos, como por ejemplo un robot; que puede ser un chasis simple con ruedas, hasta un humanoide. Además de los atributos cinemáticos y dinámicos, se pueden definir sensores, propiedades de superficie, texturas, fricción y muchas más propiedades [27]. Todas estas características permiten usar SDF para simulación, visualización, planificación de movimiento y control del robot.

3.3.6. El formato SDF para el archivo mundo

Como ya se mencionó con anterioridad, el archivo *.world* contendrá la descripción de todos los elementos de la simulación, es decir, todos los modelos utilizados, luces, sensores u objetos.

El SDF es un lenguaje que utiliza etiquetas personalizadas para descripción y organización de los datos, de forma que estos mantienen una estructura para que sean interpretados de una manera mejor.

```
<?xml version="1.0" ?>
<sdf version="1.5">
  <world name="default">
    ..
  </world>
</sdf>
```

El diseño estándar para el archivo *.world* se puede ver en los ejemplos incluidos en el directorio de mundos. El mundo comienza a ser descrito dentro de la etiqueta $< world >$, ésta tiene solamente como padre al elemento $< sdf >$ y contiene solo al atributo *name* para especificar el nombre del mundo.

De forma básica un mundo solo podría incluir un plano de piso, un sol y modelos.

```
<?xml version="1.0" ?>
<sdf version="1.5">
  <world name="default">
    <!-- A global light source -->
    <include>
      <uri>model://sun</uri>
    </include>
    <!-- Pioneer2dx model -->
    <include>
      <uri>model://pioneer2dx</uri>
      <pose>0 0 0 0 0 0</pose>
    </include>
    <!-- A ground plane -->
    <include>
      <uri>model://ground_plane</uri>
    </include>
  </world>
</sdf>
```

Al mundo se le pueden describir muchos más elementos, como por ejemplo, añadiendo la etiqueta escena se le pueden añadir nubes con cierta velocidad, sombras, luz ambiental o color de fondo.

```
<?xml version="1.0" ?>
<sdf version="1.5">
  <world name="default">
  ...
    <scene>
      <ambient>0.5 0.5 0.5 1</ambient>
      <background>0.5 0.5 0.5 1</background>
      <shadows>0</shadows>
      <sky>
        <clouds>
          <speed>12</speed>
        </clouds>
      </sky>
    </scene>
  ...
  </world>
</sdf>
```

A continuación se muestra una lista de etiquetas que pueden describir el mundo:

- $<$ *physics* $>$ Este elemento especifica el tipo y las propiedades del motor de dinámica.
- $<$ *scene* $>$ Este elemento especifica el aspecto del entorno como el color de las luz ambiental, el color del fondo, propiedades del cielo como la hora el amanecer, la velocidad y dirección de las nubes, entre otras.

- $< light >$ Este elemento describe la fuente de luz, las sombras, luz difusa, atenuación de luz, dirección, pose, entre otras.
- $< model >$ Este elemento define un robot completo o cualquier objeto físico, su elemento padre puede ser $< sdf >$ o $< world >$.
- $< actor >$ Un mundo puede contener muchos elementos actores.
- $< plugin >$ Este elemento enlaza con los códigos que se conocen como plugins.
- $< road >$ Este elemento define una carretera, se le puede definir el camino o forma, su ancho y el material.
- $< spherical\_coordinates >$ Este elemento se usa en la implementación del GPS en la simulación.

```
<?xml version="1.0" ?>
<sdf version="1.5">
  <world name="default">
    <physics type="ode">
      ...
    </physics>

    <scene>
      ...
    </scene>

    <model name="box">
      ...
    </model>

    <model name="sphere">
      ...
    </model>

    <light name="spotlight">
      ...
    </light>

  </world>
</sdf>
```

3.3.7. El formato SDF para el archivo del modelo

Dentro del mundo se describen los objetos y tal como se muestra en la figura 3.35, un modelo en SDF se compone principalmente de enlaces (*links*), articulaciones (*joints*) y ejecutables (*plugins*). Las enlaces y articulaciones descritas en SDF son la abstracción de las partes que conforman a un robot, se les conoce como enlaces debido a que generalizan secciones de la estructura que unen a las diferentes articulaciones, en un

robot los enlaces representan los brazos, el cuerpo o la mano del robot; las articulaciones representarían a la cintura, el codo, un hombro o la muñeca del robot; por último los ejecutables son fragmentos de código que se utilizan para manipular a los modelos.

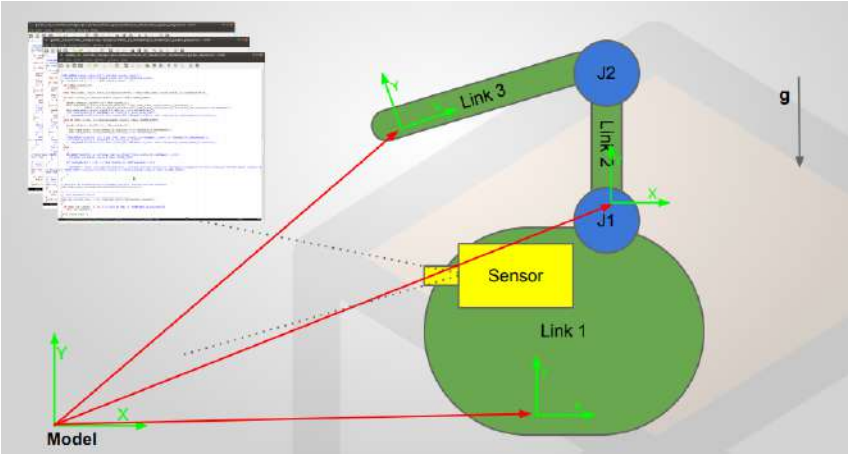


Figura 3.35: Modelo gráfico de un robot descrito en SDF.

Los enlaces deben ser descritos desde su aspecto visual y geométrico (una caja, un cilindro, etc) considerando su masa, la matriz de inercia, sus colisiones, hasta aspectos como su pose o sí éste puede ser afectado por la gravedad, entre otras características que se les pueden definir en el formato SDF. Las articulaciones al ser el elemento de unión entre dos enlaces deben ser descritas con sus propiedades cinemáticas y dinámicas. Los diversos tipos de articulaciones son revolución, caja de cambios, prismático, bola, tornillo, universal y fijo. A los modelos se les puede añadir sensores como cámaras a las cuales se les pueden describir aspectos como la pose, el tamaño en ancho y largo de la imagen que proporcionan, el ruido y si se está utilizando el sistema ROS el tópico donde publica su información.

```
<?xml version="1.0" ?>
<sdf version="1.5">
  <model name="box">
    <pose>0 0 0.5 0 0 0</pose>
    <static>false</static>

    <link name="link">
      ...
    </link>

    <joint type="revolute" name="my_joint">
      ...
    </joint>

    <plugin filename="libMyPlugin.so" name="my_plugin"/>

  </model>
</sdf>
```

3.4. Fundamentos del sistema para robots ROS

3.4.1. Descripción de ROS

Robot Operating System mejor conocido como ROS es una plataforma para desarrollo de aplicaciones en robots muy utilizada hoy en día que provee de varias características como el pase de mensajería, cómputo distribuido, reutilización de código, etc. El proyecto ROS comenzó en el año 2007 con el nombre de *Switchyard* por Morgan Quigley, como parte del proyecto *Stanford STAIR robot project*. En la actualidad ROS cuenta con una comunidad de usuarios y desarrolladores a lo largo del mundo que crece rápidamente, y muchas de las grandes compañías de robots están portando sus softwares a ROS.

ROS ofrece ventajas sobre otras plataformas como Player, YARP, Oroscos, etc; se puede decir que el sistema está listo para trabajar con algoritmos de localización y mapeo simultáneos (*SLAM por sus siglas en inglés*), localización adaptativa de Monte Carlo (*AMCL por sus siglas en inglés*), navegación autónoma y planeación de movimiento para robots manipuladores; está repleto de herramientas para depurar, visualizar y realizar simulaciones, algunas de código abierto como rqt_gui, RViz y Gazebo; cuenta con dispositivos controladores y paquetes de interfaz de varios sensores y actuadores en robótica, los sensores de gama alta incluyen Velodyne LIDAR, escáneres láser, Kinect, y actuadores tales como servos Dynamixel; cuenta con interoperabilidad entre plataformas permitiendo comunicación entre diferentes nodos, lo cuales pueden ser programados en cualquier lenguaje que tenga bibliotecas cliente ROS, como C++ o C, Python o Java; el uso de nodos y tópicos permite utilizar de forma concurrente recursos de hardware, lo que reduce la complejidad del cómputo e incrementa la capacidad de depuración del sistema.

3.4.2. Instalación de la distribución Indigo

Uno de los primeros conocimientos que se deben adquirir para trabajar con ROS es conocer sus distribuciones y su compatibilidad con los sistemas operativos que hay disponibles en el mercado, en los cuales es posible instalar ROS.

Se conoce que ROS es un meta-sistema operativo que es completamente soportado por sistemas Linux, como Debian y especialmente por Ubuntu; sin embargo, es posible

que trabaje en otros sistemas como Mac y versiones recientes de ROS experimentalmente mencionan ser instaladas en Windows. En el cuadro de la figura 3.36 se muestran las distribuciones disponibles por ROS y su compatibilidad con el sistema preferente Ubuntu.

Figura 3.36: Distribuciones de ROS[15].

La versión más actualizada de Ubuntu es Xenial y por ende la distribución de ROS más adecuada sería Kinetic; sin embargo, en este proyecto se trabajó con la distribución Indigo debido a que se cuenta con el sistema Ubuntu 14.04 LT. ROS Indigo es una de las distribuciones más utilizadas y recomendadas por diversos desarrolladores a lo largo del mundo y por muchos autores de libros sobre ROS, por lo que brinda acceso sin inconvenientes a diversos proyectos basados en esta plataforma que se encuentran disponibles por la comunidad.

Es importante tener en cuenta que un paquete ROS puede presentar errores de compilación en una distribución diferente a la de su desarrollo. De igual manera es recomendable realizar primero la instalación del sistema operativo para robots antes de realizar la instalación del simulador Gazebo, ya que cada distribución de ROS cuenta

[15]Fuente: http://wiki.ros.org/Distributions

con una versión por defecto del simulador, ROS Melodic trabaja por defecto con la versión 9 de Gazebo, ROS Lunar y Kinetic trabajan con la versión 7 del simulador, y ROS Indigo con la versión 2.

A continuación se muestran los pasos de instalación [28] para la distribución Indigo de ROS que es solamente soportada por Ubuntu Saucy 13.10 y Trusty 14.04. Se comienza configurando los repositorios del sistema abriendo el centro de Software y Actualizaciones de Ubuntu permitiendo *restricted*, *universe* y *multiverse*.

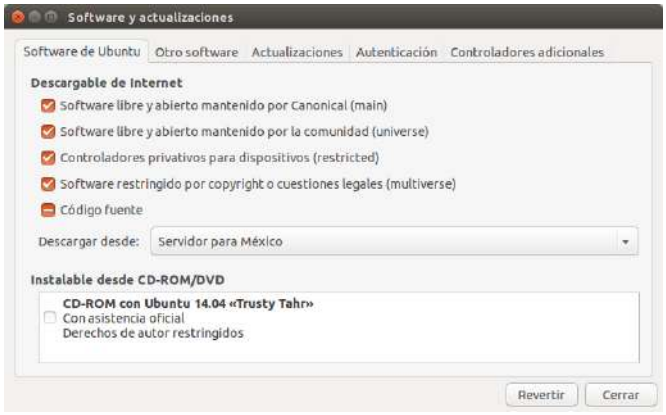


Figura 3.37: Centro de Software y Actualizaciones de Ubuntu.

Abriendo una terminal, el primer comando sirve para que la computadora acepte el software del repositorio de software de ROS *packages.ros.org* que es el sitio oficial.

```
sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu
$(lsb_release -sc) main" > /etc/apt/sources.list.d/ros-latest.list'
```

El comando siguiente es para agregar *apt-keys* administrando la lista de claves utilizadas por *apt* para autenticar los paquetes. Los paquetes que tengan esta clave serán considerados de confianza.

```
sudo apt-key adv --keyserver hkp://ha.pool.sks-keyservers.net:80
--recv-key \\ 421C365BD9FF1F717815A3895523BAEEB01FA116
```

Se deben actualizar la lista de paquetes disponibles para Ubuntu.

```
sudo apt-get update
```

Después de actualizar la lista de paquetes se procede a la instalación de ROS. La instalación recomendada es *Desktop-Full* que incluye al sistema operativo, rqt, rviz, bibliotecas robot-generic, navegación, simuladores 2D/3D y percepción 2D/3D. El simulador por defecto que se instala es Gazebo en su versión 2, siendo el más compatible con ROS Indigo; no obstante, es posible utilizar versiones más actuales del simulador junto con la distribución, como Gazebo 5, 6 y 7.

```
sudo apt-get install ros-indigo-desktop-full
```

No debe resultar extraño seguir realizando instalaciones de paquetes, ya que a pesar de instalar *desktop-full* siempre se necesitarán paquetes adicionales de ROS que se tienen que instalar de acuerdo a los proyectos y paquetes con los que se trabaje. El siguiente paso es inicializar *rosdep*, que permite descargar e instalar fácilmente dependencias del sistema para los paquetes fuente de ROS que se desean compilar, también es necesario para ejecutar algunos componentes principales de ROS.

```
sudo rosdep init

rosdep update
```

Cada vez que se abra una terminal se deben configurar las variables de entorno de ROS para que la terminal pueda ejecutar los comandos propios del sistema, para lograrlo se debe tipear el comando:

```
source /opt/ros/indigo/setup.bash
```

Alternativamente se pueden configurar las variables de entorno de ROS automáticamente cada vez que se abra una nueva terminal. Si se está usando *bash* se puede tipear el comando:

```
echo "source /opt/ros/indigo/setup.bash" >> ~/.bashrc

source ~/.bashrc
```

Se instala por último *rosinstall* que es una herramienta de línea de comandos útil que permite fácilmente descargar paquetes de ROS con un solo comando:

```
sudo apt-get install python-rosinstall
```

En este momento ya se cuenta con la instalación más completa de ROS Indigo y con la versión 2 del simulador Gazebo. Si se desea desintalar la versión por defecto del sistema para trabajar con una versión más actual del simulador bastaría con tipear el comando:

```
sudo apt-get remove gazebo2
```

si se tiene alguna otra versión instalada, solo basta con indicar el número de la versión de Gazebo que se desea desinstalar.

3.4.3. Nivel de sistema de archivos

El nivel de sistema de archivos se refiere a la organización en el disco de carpetas y archivos ROS, de forma similar a un sistema operativo. Su objetivo principal es el de centralizar el proceso de construcción de un proyecto, mientras que al mismo tiempo proporciona suficiente flexibilidad y herramientas para descentralizar sus dependencias [29]. En este apartado se explica: que es un paquete en ROS, a que se conoce como pilas (*stacks*) o meta-paquetes, que son los archivos de manifiesto, así como una descripción de mensajes, servicios y sus lenguajes simplificados. De forma gráfica el nivel tiene la forma que se muestra en la figura 3.38.

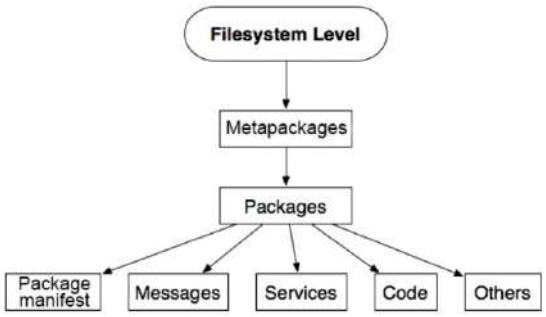


Figura 3.38: ROS File system level.

- Los paquetes de ROS son la unidad más básica del software de ROS. Estos contienen los procesos de ROS en tiempo de ejecución (nodos), bibliotecas, archivos de configuración, y más, encontrándose organizados juntos como una sola unidad.
- El archivo de manifiesto de un paquete se encuentra dentro del mismo y contiene información sobre el paquete como autor, licencia, dependencias, banderas de compilación, y más (*package.xml*).
- Cuando se juntan paquetes con alguna funcionalidad se obtiene un meta paquete también llamados pilas. En ROS existen muchas pilas con diferentes usos, una de ellas es la pila de navegación.
- Los meta paquetes tienen un archivo de manifiesto al igual que los paquetes, a diferencia del archivo de manifiesto de los paquetes, éste podría incluir dentro paquetes como dependencias de tiempo de ejecución.
- Los mensajes son el tipo de información que es enviada de un proceso a otro en ROS. Dentro de un paquete en la carpeta *msg* se pueden definir mensajes personalizados (my_package/msg/MyMessageType.msg).
- Los servicios son un tipo de respuestas o solicitudes que sirven de interacción entre los procesos. Los datos de respuesta y solicitudes pueden ser definidos dentro de un paquete en la carpeta *srv* (my_package/srv/MyServiceType.srv).

Paquetes y Pilas

Para entender la organización de los paquetes que se utilizarán en el desarrollo del proyecto es conveniente explicar las carpetas y archivos que puede contener [30].

- Archivo *package.xml*: contiene información de manifiesto del paquete.
- Archivo *CMakeLists.txt*: describe como compilar el código del paquete y las dependencias que utilizará.
- Carpeta *config*: contiene todos los archivos de configuración del paquete. Esta carpeta es creada por el usuario y nombrarla *config* es de práctica común.
- Carpeta *include*: contiene los archivos *headers* (.h)y bibliotecas usadas por el paquete.
- Carpeta *scripts*: almacena archivos ejecutables escritos en Python (.py).
- Carpeta *src*: almacena códigos fuente escritos en C++ (.cpp).
- Carpeta *launch*: almacena los archivos de lanzamiento (.launch) que son usados para lanzar uno o más nodos.
- Carpeta *msg*: contiene definiciones de mensajes (.msg) personalizados.
- Carpeta *srv*: contiene definiciones de servicios.
- Carpeta *action*: contiene definiciones de acciones.

Figura 3.39: Archivos dentro del paquete.

Cuando nos referimos a una pila hablamos de un conjunto de paquetes organizados. La meta de una pila es simplificar el proceso de compartir código. Una pila se puede crear manualmente o puede ser creada con el comando *roscreate-stack*.

Mensajes

Los mensajes son un tipo de comunicación entre los procesos de ROS, cada tipo de mensaje utiliza una descripción particular que se almacena en archivos *.msg* y estos a su vez se encuentran en el subdirectorio *msg* de cada paquete. Los mensajes tienen un lenguaje de descripción que facilita que las herramientas de ROS generen código fuente para el tipo de mensaje en diferentes lenguajes destino. Un mensaje se describe mediante una lista de campos, cada campo tiene un tipo y un nombre separados por un espacio.

```
fieldtype1 fieldname1
fieldtype2 fieldname2
fieldtype3 fieldname3
```

Figura 3.40: Estructura de un mensaje.

La figura 3.41 es un ejemplo de un mensaje tomado del paquete *erlecopeter* de la carpeta *mavros_msg*. y en la figura 3.42 se observan los tipos de campo comunes para los mensajes de ROS.

RCIn.msg ×

```
# RAW RC input state

std_msgs/Header header
uint8 rssi
uint16[] channels
```

Figura 3.41: Mensaje RCIn.msg

Primitive Type	Serialization	C++	Python2	Python3
bool (1)	unsigned 8-bit int	uint8_t (2)	bool	
int8	signed 8-bit int	int8_t	int	
uint8	unsigned 8-bit int	uint8_t	int (3)	
int16	signed 16-bit int	int16_t	int	
uint16	unsigned 16-bit int	uint16_t	int	
int32	signed 32-bit int	int32_t	int	
uint32	unsigned 32-bit int	uint32_t	int	
int64	signed 64-bit int	int64_t	long	int
uint64	unsigned 64-bit int	uint64_t	long	int
float32	32-bit IEEE float	float	float	
float64	64-bit IEEE float	double	float	
string	ascii string (4)	std::string	str	bytes
time	secs/nsecs unsigned 32-bit ints	ros::Time	rospy.Time	
duration	secs/nsecs signed 32-bit ints	ros::Duration	rospy.Duration	

Figura 3.42: Tipos de campo en ROS[16].

[16]Fuente: http://wiki.ros.org/msg

Servicios

La comunicación por servicios utiliza un lenguaje de descripción para cada tipo de servicio, los archivos *srv* de cada descripción se encuentran en el subdirectorio *srv* de cada paquete. Un modo de definirlos es considerarlos como dos mensajes, por lo que en el servicio se definen tipos de datos de solicitud y tipos de datos de respuesta, dos secciones separadas por tres guiones altos, de manera que ambas son un tipo de mensaje.

Un ejemplo del formato para la descripción de servicios se observa en la figura 3.43.

```
#request constants
int8 FOO=1
int8 BAR=2
#request fields
int8 foobar
another_pkg/AnotherMessage msg
---
#response constants
uint32 SECRET=123456
#response fields
another_pkg/YetAnotherMessage val
CustomMessageDefinedInThisPackage value
uint32 an_integer
```

Figura 3.43: Ejemplo de un servicio.

3.4.4. Nivel gráfico de computo

Al trabajar con ROS se utilizan frecuentemente términos propios del sistema, conceptos como nodo, maestro, servidor de parámetros, mensajes, tópicos, servicios y bolsas (*bags*) estos nuevos términos forman parte del nivel gráfico de ROS [31]. En la figura 3.44 se observa la representación gráfica de este nivel mostrando la interacción de elementos del nivel gráfico de ROS en un robot.

La pila *ros_comm* contiene los paquetes *middleware* de comunicación de ROS y estos paquetes en conjunto son llamados Capa de Grafo de ROS.

Nodos

Se le conoce como nodos a todos los procesos que realizan computo en ROS, programas escritos C++ ó Python que utilizan bibliotecas de cliente, como:

- *roscpp*: biblioteca de C++.
- *rospy*: biblioteca de pyhton.

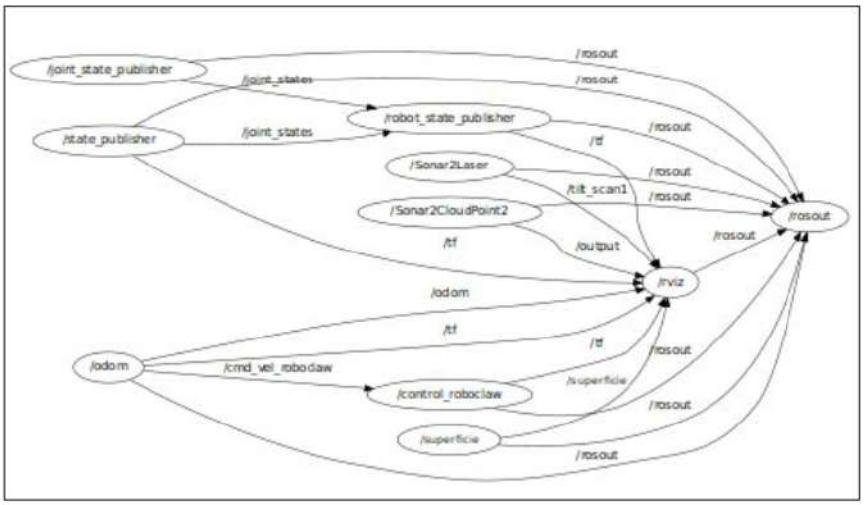


Figura 3.44: Grafo de un robot usando nodos y tópicos con la herramienta *rqt*.

En un robot que utiliza ROS pueden haber muchos nodos ejecutándose al mismo tiempo realizando diferentes tipos de tareas, de manera que un nodo puede estar procesando imágenes de una cámara, mientras que otro realiza odometría y al mismo tiempo otro nodo ejecuta algún control de movimiento. Aunque alguno de estos nodos llegue a fallar, los procesos son independientes por lo que el sistema entero del robot no se detendrá.

La comunicación entre los nodos se realiza usando métodos propios de ROS enviando o recibiendo datos utilizando tópicos, servicios y parámetros. Cada nodo posee un nombre único y para que dos nodos se comuniquen no es necesario que estén escritos en el mismo lenguaje, solo basta con que la interfaz entre estos se cumpla. Podemos ejemplificar la comunicación entre nodos en la figura 3.45, el nodo hablante publica en un tópico denominado charla y el nodo oyente está subscrito a él.

Figura 3.45: Comunicación entre Nodos.

Mensajes

Como se mencionó en la sección anterior, los nodos se comunican entre ellos publicando y recibiendo mensajes a través de tópicos generando una comunicación unidireccional a diferencia de los servicios. Un mensaje es una estructura de datos que contiene tipos de campos, los mensajes están definidos en la carpeta *msg* de cada paquete, cada mensaje tiene la extensión *.msg*. Se puede acceder a la definición del mensaje refiriéndose a él por el nombre del paquete que lo contiene y el nombre del mensaje por ejemplo para el mensaje String.msg que se localiza en *std_msgs/msg/String.msg* se le refiere como *std_msgs/String*.

La herramienta de línea de comandos para mensajes es *rosmsg*, ésta imprime información sobre el mensaje y puede encontrar archivos fuente que usan un tipo de mensaje.

Servicios

Al igual que los mensajes, los servicios cumplen con la tarea de establecer comunicación entre los nodos; no obstante, el uso de servicios establece un tipo de comunicación no unidereccional de solicitud (*request*) y respuesta (*response*). Todos los servicios de un paquete se pueden encontrar almacenados en la carpeta *srv*.

En los servicios de ROS, un nodo ofrece un servicio (*server*) y otro nodo (*client*) puede hacer uso de éste, haciendo una llamada mediante un mensaje y esperando su respuesta. Al igual que en los tópicos, los servicios cuenta con un nombre único que los identifica. Para llamar a un servicio se necesita referirse a él por el nombre del paquete que lo contiene y el nombre del servicio, por ejemplo para el archivo de servicio *sample1.srv* que se localiza en *sample_package1/srv/sample1.srv* se le refiere como *sample_package1/sample1*.

Existen dos herramientas de línea de comandos que sirven para obtener información sobre los servicios de ROS. La primera es *rossrv*, que es muy parecida a *rosmsg*, y sirve para acceder a la información sobre los tipos de servicios. El segundo comando es *rosservice* que proporciona información variada o específica sobre los servicios

```
rosservice list
```

proporciona información sobre los servicios que se encuentran disponibles al momento

```
rosservice find rospy_tutorials/AddTwoInts
```

muestra en la línea de comandos todos los los servicios de un tipo particular

```
rosservice info /rosout
```

provee información sobre un servicio específico.

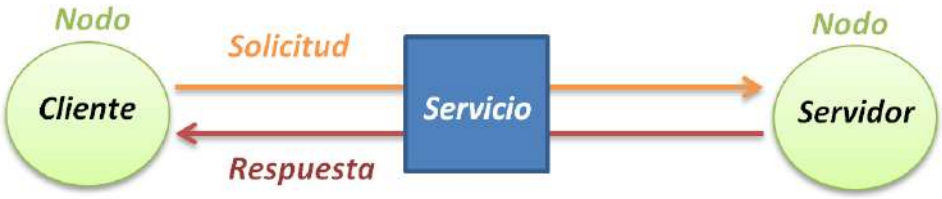


Figura 3.46: Servicios en ROS.

Tópicos

Al sistema de intercambio de información entre nodos se le conoce como *publisher and suscriber*. Esta forma de intercambio obliga a que dos procesos independientes puedan únicamente comunicarse publicando o suscribiéndose a un tópico, el cual solo es un *bus* de información por el que viaja el mensaje y que cuenta con un nombre exclusivo asignado para identificar su contenido. El número de nodos que se suscribe a un tópico es irrelevante, permitiendo que el mensaje llegue a cualquier nodo suscrito a él, donde lo único que debe coincidir entre el nodo que publica en un tópico y el nodo que se suscribe al mismo, es el tipo de mensaje.

La información que se envía en cada tópico es unidireccional, por lo que los nodos que publican no se enteran si cero, uno o más nodos reciben la información, esto genera un desacoplo entre la producción y el consumo de información.

rostopic es una herramienta de línea de comandos para interaccionar con los tópicos, es muy común utilizar el comando:

```
rostopic list
```

el cual permite al usuario observar directamente en la línea de comandos una lista de todos los tópicos disponibles. De igual manera:

```
rostopic echo /topic
```

permite observar la información que publica el tópico indicado mediante el comando.

Nodo Maestro

La primera acción que realiza un nodo al ejecutarse en el sistema es buscar al nodo maestro para registrar su nombre en él. La función del maestro consiste en permitir a los nodos localizarse entre ellos, teniendo los detalles útiles para establecer la comunicación, como el nombre de los nodos y el tipo de datos que están publicando en un tópico. El maestro conserva esta información de todos los procesos que se encuentran corriendo en el sistema y actualiza cada cambio que generen los mismos. Después de que dos nodos se encuentran conectados el maestro no juega ningún papel en su comunicación y se comporta de la misma manera para establecer comunicación mediante el uso mensajes como el de servicios. Este nodo se ejecuta usando el comando *roscore*. En la figura 3.47 se muestra como el maestro interactua con los nodos que publican y suscriben a tópicos.

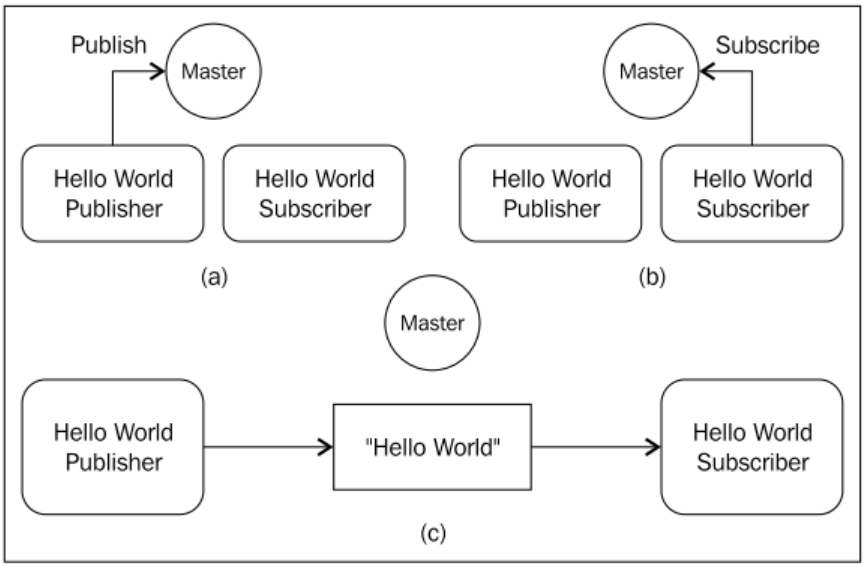


Figura 3.47: Interacción del ROS Master.

Servidor de parámetros

Mientras se trabaja con ROS es posible que se requiera del uso de un número considerable de parámetros por lo que ROS proporciona el servidor de parámetros, que es utilizado por los nodos para almacenar o leer parámetros en tiempo de ejecución. La herramienta *rosparam* es usada para obtenerlos y modificarlos desde la línea de comandos.

3.4.5. Creación del catkin workspace y paquetes

El espacio de trabajo en ROS se conoce como *catking workspace*, el cual es una carpeta que puede contener, modificar, compilar e instalar múltiples paquetes que son

interdependientes.

Para crear y compilar [32] este espacio de trabajo se procede a tipear:

```
mkdir -p ~/catkin_ws/src

cd ~/catkin_ws/

catkin_make
```

El comando *catkin_make* se encarga construir paquetes en ROS, engloba las instrucciones de CMake.

En un proyecto CMake

```
mkdir build

cd build

cmake ..

make

make install # (opcionalmente)
```

Dentro del *workspace* se crearán tres carpetas *build*, *devel* y *src*. La carpeta de compilación (*build*) es la ubicación predeterminada del espacio de construcción y es donde se llama a *cmake* y *make* para configurar y crear sus paquetes. La carpeta de desarrollo (*devel*) es la ubicación predeterminada del espacio de desarrollo, que es donde se encuentran sus ejecutables y bibliotecas antes de instalar sus paquetes. Dentro del espacio source (*src*) se creará automáticamente un archivo *CMakeLists.txt* este archivo es invocado por el *cmake* durante la configuración de los proyectos en el espacio de trabajo, esta misma carpeta es la que contendrá los códigos fuente de los paquetes. La estructura del *workspace*será parecida a lo que se observa en la figura 3.48.

```
workspace_folder/        -- WORKSPACE
  src/                   -- SOURCE SPACE
    CMakeLists.txt       -- 'Toplevel' CMake file, provided by catkin
    package_1/
      CMakeLists.txt     -- CMakeLists.txt file for package_1
      package.xml        -- Package manifest for package_1
    ...
    package_n/
      CMakeLists.txt     -- CMakeLists.txt file for package_n
      package.xml        -- Package manifest for package_n
```

Figura 3.48: Diversos paquetes en el *workspace*.

Una de las primeras herramientas con las que se trabajará en ROS es con la creación y el uso de paquetes, ya sea de aquellos que se encuentran disponibles gracias a

la comunidad de ROS y son constantemente utilizados en diversos proyectos, así como para paquetes que deseen crear usuarios comunes. ROS proporciona el comando *catking_create_pkg* para crear paquetes, la forma correcta para crearlos y compilarlos es tipeando lo siguiente:

```
cd ~/catkin_ws/src

catking_create_pkg beginner_tutorials std_msgs rospy roscpp

cd ~/catkin_ws

catkin_make

.~/catkin_ws/devel/setup.bash
```

El paquete creado llamado *beginner_tutorials* tendrá como dependencias de primer orden std_msgs, rospy y roscpp, automáticamente se crearán los archivos *package.xml* y *CMakeLists.txt* que se han completado parcialmente con la información que propocionó *catking_create_pkg*.

En la figura 3.49 se puede observar el archivo .xml que se genera.

```
[des]activar nros. de línea
 1 <?xml version="1.0"?>
 2 <package format="2">
 3   <name>beginner_tutorials</name>
 4   <version>0.1.0</version>
 5   <description>The beginner_tutorials package</description>
 6
 7   <maintainer email="you@yourdomain.tld">Your Name</maintainer>
 8   <license>BSD</license>
 9   <url type="website">http://wiki.ros.org/beginner_tutorials</url>
10   <author email="you@yourdomain.tld">Jane Doe</author>
11
12   <buildtool_depend>catkin</buildtool_depend>
13
14   <build_depend>roscpp</build_depend>
15   <build_depend>rospy</build_depend>
16   <build_depend>std_msgs</build_depend>
17
18   <exec_depend>roscpp</exec_depend>
19   <exec_depend>rospy</exec_depend>
20   <exec_depend>std_msgs</exec_depend>
21
22 </package>
```

Figura 3.49: Estructura del archivo .xml.

Dentro de la etiqueta $<$ *build_depend* $><$ /*build_depend* $>$ se incluyen los paquetes que son necesarios para la compilación del código fuente del paquete. Los paquetes dentro de la etiqueta $<$ *run_depend* $><$ /*run_depend* $>$ son necesarios durante el corrimiento (*runtime*) de los nodos.

3.5. MavLink y Mavros

3.5.1. MavLink

Para el desarrollo de proyectos en drones es importante conocer como se lleva a cabo la comunicación abordo de ellos, por este motivo se explica el protocolo MAVLink, el cual es un protocolo de mensajería para comunicarse con drones y entre los componentes abordo del mismo [33]. Básicamente es un flujo de bytes codificados y enviados a través de USB serial, frecuencias RC, WiFi, GPRS, etc. La codificación se refiere a que el paquete de datos se estructura agregando sumas de comprobación, números de secuencia que se envían a través del canal en bytes.

La estructura del mensaje o paquete de MAVLink principalmente tiene 6 bytes de encabezamiento, bytes de carga útil (*payload*) y 2 bytes de suma de comprobación (*checksum*). La longitud mínima del paquete es de 8 bytes cuando los paquetes no tiene carga útil, y la longitud máxima es de 263 bytes para la carga útil completa [34].

En la figura 3.50 se muestra la estructura de un mensaje de MAVLink.

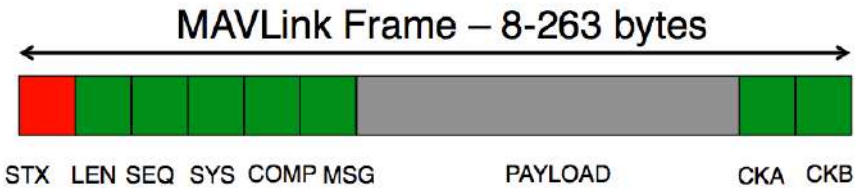


Figura 3.50: Estructura de los mensajes de MAVLink

El protocolo se orientó principalmente a la velocidad de transmisión y a la seguridad, admite datos enteros de tamaño fijo, número de punto flotante de precisión simple, matrices de estos tipos de datos (char, int8, uint8, int16, uint16, int32, uint32, int64, uint64, float, double).

Las estaciones de control terrestre son un mediador que utilizan este protocolo para comunicarse con el dron. Por ejemplo, el mensaje MAVLINK_MSG_ID_HEARTBEAT es el mensaje más importante ya que cada segundo la estación de control envía el mensaje para asegurar que el MP está sincronizado con el APM cuando actualice algunos parámetros. En caso de que la conexión falle el vehículo activa ciertas rutinas preprogramadas como regresar al punto de lanzamiento o aterrizar [35].

3.5.2. Mavros

La computadora de vuelo *Pixhack* funciona utilizando el protocolo MAVLink; no obstante, existe un puente que une este protocolo con el sistema ROS. El paquete Mavros de ROS permite una comunicación amplia de pilotos automáticos que utilizan MAVLink entre computadoras que ejecutan ROS. Prácticamente Mavros es el puente "oficial" soportado entre ROS y el protocolo MAVLink. De la misma manera que ROS,

Mavros utiliza los conceptos de nodos, tópicos, suscriptores, publicadores, servicios, mensajes, etc. En la figura 3.51 se muestran algunos tópicos de Mavros utilizando la herramienta *rqt* de ROS.

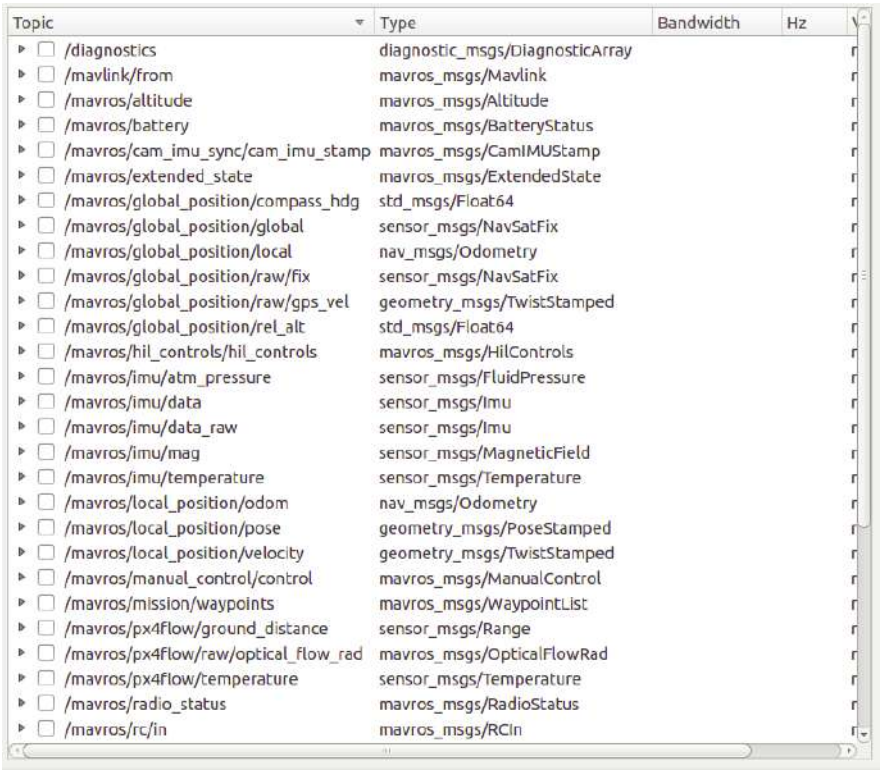


Figura 3.51: Tópicos de Mavros utilizando la herramienta rqt.

La instalación de MAVLink y Mavros que se muestra a continuación se realizó en un sistema Ubuntu 14.04 con ROS Indigo instalado. Utilizando los comandos siguientes:

```
$ sudo apt-get install ros-indigo-mavlink
```

```
$ sudo apt-get install ros-kinetic-mavros
ros-kinetic-mavros-extras
```

En [36] es posible encontrar información más detallada sobre el paquete Mavros.

3.6. Fundamentos de Visual Servoing

En los últimos años tanto el desarrollo tecnológico de los vehículos aéreos no tripulados como su utilización para realizar tareas que ayuden al desarrollo de proyectos en áreas diversas ha tenido mucho auge. Para muchas de sus aplicaciones es muy común que a los drones se le establezcan trayectorias de vuelo predefinidas, las cuales dependen principalmente de datos del Geoposicionamiento Satelital (GPS) en donde el dron se dirige en línea recta a cada punto establecido (*View points*); sin embargo, una forma diferente de hacer que el vehículo siga una trayectoria definida, podría ser por medio de un marcador visual, este marcador puede ser una línea en el suelo la cual pueda ser observada directamente por una cámara montada sobre el vehículo, haciendo que éste siga el marcador teniendo un control preciso de movimientos de alabeo y cabeceo. Esto puede ser logrado con un tema conocido como *Visual Servoing*.

Se conoce como *Visual Servoing* a la utilización de técnicas de visión computacional combinada con teoría de control para manipular los movimientos de un robot. De manera práctica sirve incrementar la precisión de los sistemas usando un bucle de control con retroalimentación visual para controlar la pose del efector final de un robot relativo a su objetivo o un conjunto de objetivos [37]. El problema se expresa como una función de error $\mathbf{e}(t)$ que relaciona al vector de características $\mathbf{s}$, la descripción del estado actual del robot, con el vector de características $\mathbf{s}^*$, la descripción del estado objetivo [38], es decir:

$$\mathbf{e}(t) = \mathbf{s}(\mathbf{m}(t), \mathbf{a}) - \mathbf{s}^* \tag{3.1}$$

donde $\mathbf{m}(t)$ son mediciones realizadas en la imagen (momentos, contornos, etc.) y $\mathbf{a}$ información adicional conocida del sistema (parámetros intrínsecos de la cámara, modelo 3D del objetivo, etc.).

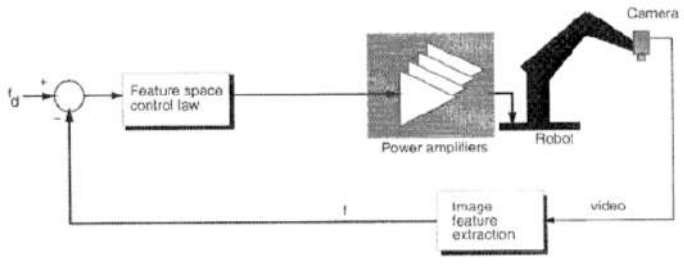


Figura 3.52: Sistema de *Visual Servoing* basado en imágenes.

Dependiendo del tipo de características utilizadas para describir el error entre el estado actual y el objetivo, se clasifica el problema de *Visual Servoing* en dos principales enfoques: basado en imagen y basado en posición. En el enfoque basado en imagen (*IBVS por sus siglas en inglés*), los estados del robot se definen a partir de

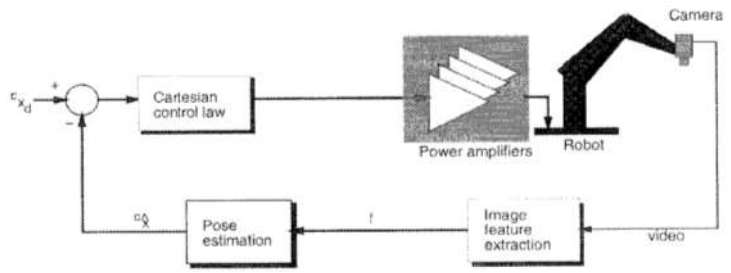


Figura 3.53: Sistema de *Visual Servoing* basado en posición.

características extraídas directamente de la imagen, como las coordenadas (en pixeles) de un grupo de puntos en la imagen, el error consiste entonces en la posición actual de estos puntos, respecto a la ubicación objetivo. En el enfoque basado en posición (*PBVS por sus siglas en inglés*), la ley de control se establece utilizando la pose en tres dimensiones del objetivo como referencia para determinar los movimientos del robot. La pose está formada por la rotación y traslación del objetivo respecto a la cámara y es determinada utilizando información procedente de mediciones en la imagen, realizadas en tiempo real e información del objetivo y la cámara. Las figuras 3.52 y 3.53 muestran los esquemas de los dos tipos de enfoques mencionados anteriormente.

Capítulo 4

Metodología

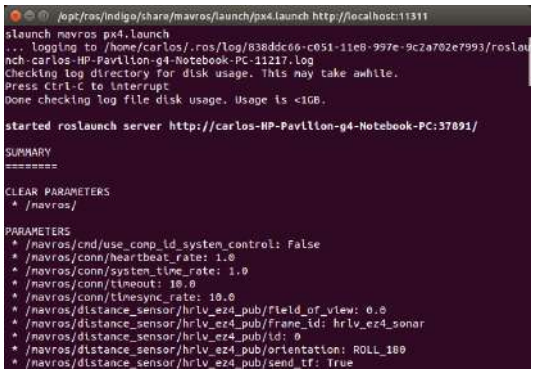
4.1. Conexión a la computadora computadora de vuelo del dron

En la sección 3.2.2 se habla acerca como utilizar la estación de control terrestre para realizar las calibraciones del vehículo. Aunque utilizando la estación de control se puede controlar el dron y planear vuelos autónomos de manera muy eficiente, el objetivo que se tiene en este trabajo es que sea posible crear programas que sean capaces de controlar al vehículo. Utilizando los conocimientos de la sección 3.4 estos programas serán nodos de ROS y como también se mencionó en 3.5.2 el paquete utilizado para controlar el vehículo será Mavros.

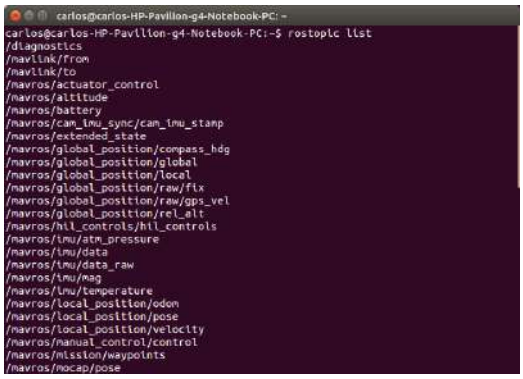
Como se indicó en 3.4.4 la función del nodo maestro es permitir la comunicación entre los nodos, es por esto que la primera acción para establecer comunicación con la computadora de vuelo *Pixhawk* es lanzar el nodo maestro. El archivo de inicio *px4.launch* se encuentra ubicado en la carpeta *mavros* y se ejecuta mediante los comandos siguientes:

```
$ roscd mavros

$ cd lauch

$ roslaunch mavros px4.launch
```

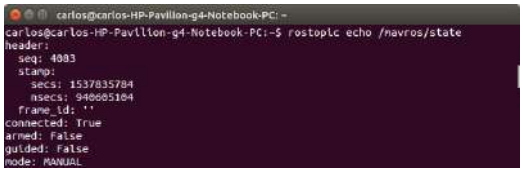
En la figura 4.1a se muestra el lanzamiento del nodo maestro. La comunicación con la computadora de vuelo se puede establecer via cable USB o usando la telemetría del sistema como se muestra en la figura 3.29 (*roslaunch mavros px4.launch* inicia la comunicación mediante cable USB). A partir de este momento ya se puede ejecutar el comando *$ rostopic list* el cual enlista todos los tópicos que se encuentren activos en el sistema como se muestra en la figura 4.1b. Se puede ejecutar un *$ rostopic echo* para imprimir el contenido de los tópicos como se muestra en la figura 4.1c.



(a) Lanzamiento del archivo *px4.launch*.



(b) Ejecución del comando *rostopic list*.



(c) Ejecución del comando *rostopic echo /mavros/state*.

Figura 4.1: Comunicación con la computadora de vuelo mediante línea de comandos.

Para poder lanzar el nodo maestro utilizando la telemetría se utiliza el siguiente comando:

```
$ roslaunch mavros px4.launch fcu_url:=/dev/ttyUSB0:57600
```

Ahora es posible ejecutar nodos los cuales se encarguen de proporcionar la información de la computadora de vuelo; por ejemplo, para leer los datos de la Unidad de Medida Inercial (*IMU por sus siglas en inglés*) se debe de crear y compilar un espacio de trabajo con los pasos descritos en 3.4.5, y ejecutar el nodo.

```
#include "ros/ros.h"
#include "sensor_msgs/Imu.h"

void chatterCallback(const sensor_msgs::Imu::ConstPtr& msg){
    ROS_INFO("\nlinear_acceleration\\nx:[%f]\ny:[%f]\nz:[%f]", msg->
        linear_acceleration.x, msg->linear_acceleration.y, msg->
        linear_acceleration.z);
}

int main(int argc, char **argv){
    ros::init(argc, argv, "imu_data");
    ros::NodeHandle n;
    ros::Subscriber sub = n.subscribe("/mavros/imu/data", 1000,
        chatterCallback);
    ros::spin();
    return 0;
}
```

Figura 4.2: Código para mostrar datos del *IMU* en la computadora de vuelo.

El código que se muestra en la figura 4.2 se encarga de imprimir en consola (ver figura 4.3) la información que proporciona la *IMU* una vez que ya se estableció la conexión con la computadora de vuelo del vehículo. Inicialmente en el código se agrega la librería *Imu.h*, dentro de *main* se comienza inicializando el nodo ROS con *init()* asignando el nombre de *imu_data* al nodo y declarando el manejador *n* del nodo con *NodeHandle*. El objeto de tipo *Subscriber* se utiliza para suscribirse al tópico */mavros/imu/data* y cada vez un nuevo mensaje haya llegado por el tópico la función *chatterCallback* será llamada. La función *spin()* genera un bucle, generando devoluciones de llamada de mensajes tan rápido como sea posible. Dentro de la función *chatterCallback* podemos ver el acceso a la información de cada eje (*x,y,z*) que proporciona la *IMU* para su impresión en la consola.

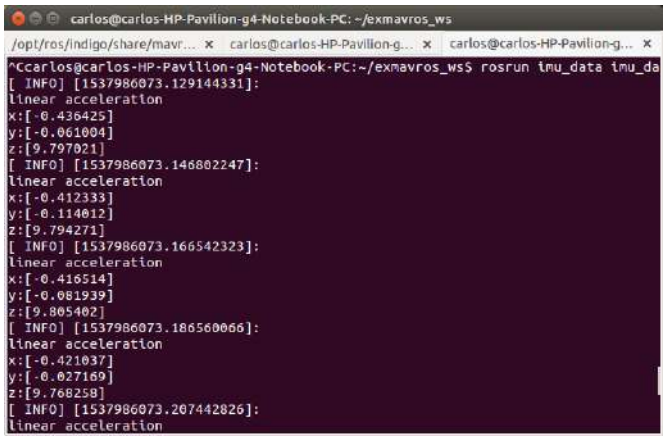


Figura 4.3: Información en consola de los datos provenientes de la *IMU*.

Como ya se ha mencionado con anterioridad el uso del vehículo físico para efectuar pruebas siempre es un poco riesgoso debido a alguna falla que pueda presentarse durante el proceso, por este motivo se utiliza el apoyo del simulador Gazebo y la simulación del vehículo *Erle-Copter* ya que es posible implementar los mismos códigos en el vehículo físico que en el utilizado en la simulación.

4.2. Simulación utilizando el vehículo Erle-Copter

El *Erle-Copter* es un dron desarrollado por la empresa Erle Robotics[1]. Usa la computadora *Erle-brain* como controladora de vuelo la cual es capaz de trabajar con ROS y software para piloto automático APM y PX4.

Figura 4.4: Imagen del vehículo *Erle-Copter*

Erle Robotics es una empresa dedicada al campo de la robótica, particularmente en el área de drones. Cuyo objetivo es el brindar condiciones para el aprendizaje y la investigación en esta área. Inspirados en la placa de desarrollo BeagleBone, desarrollaron *Erle-brain* que es una pequeña computadora con más de 36 sensores, muchas E/S y potencia de procesamiento para el análisis en tiempo real, capaz de ejecutarse sistemas operativos de propósito general tales como Windows Embedded CE, los sistemas de archivos basados en kernel de Linux (Android, Ångström, Debian, Ubuntu, Fedora, ArchLinux, Gentoo, etc). Sus características son:

- Sistema operativo Linux (Ubuntu, Debian, Android).
- ROS.
- BeaglePilot (trabajo en progreso).
- Más de 36 sensores.
- Almacenamiento de tarjeta microSD.
- GPS.
- Host USB (dongles WiFi, dongles Bluetooth, monitores, teclados, etc).
- 92 pines para conectar otros dispositivos.
- Carga inalámbrica (trabajo en progreso).

[1]Fuente: http://docs.erlerobotics.com/

Figura 4.5: *Erle-brain*.

4.2.1. Simulación en Gazebo

Erle Robotics se ha encargado de desarrollar un entorno de simulación del *Erle-Copter* en el simulador Gazebo, compatible con la versión Indigo de ROS. La simulación es provista de características como:

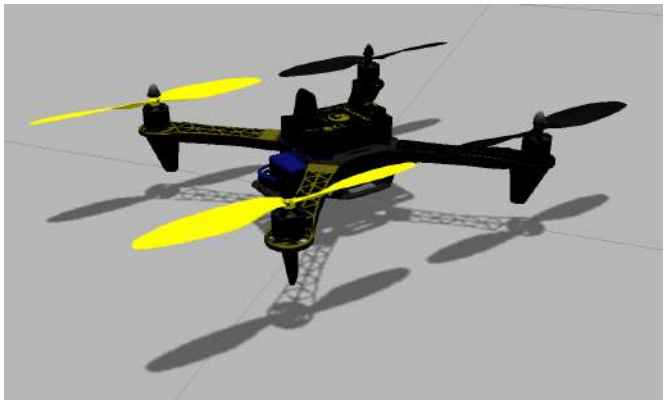


Figura 4.6: Modelo del *Erle-Copter* en el simulador.

- Estabilidad de simulación y respuesta determinante de UAV (mecanismo de bloqueo por pasos).
- Simplificación del lanzamiento de la simulación de ROS / Gazebo, completamente configurable a través de argumentos para la secuencia de comandos de lanzamiento SITL.
- Integración del sensor de GPS proporcionado por el complemento de Héctor.
- Imagen de mapa de superposición georeferenciada en MavProxy, para evaluar mejor la posición del UAV en relación con su entorno simulado.
- Unidad de medición inercial (IMU) que proporciona aceleración lineal, velocidad angular, presión atmosférica y altitud.
- Brújula que proporciona el título.

- GPS que proporciona longitud, latitud y altitud.
- Erle-Copter está equipado con 2 cámaras, una frontal y una inferior, 2 sensores de sonda, uno frontal y un botón y también es posible incluir un láser 2D o cámaras de profundidad.
- Permite la manipulación del joystick.
- Esta simulación incluye el paquete Mavros (*Micro Air Vehicle ROS*).
- Los comandos de navegación se pueden enviar directamente al controlador de vuelo producido por un algoritmo de robótica a través de Mavros.

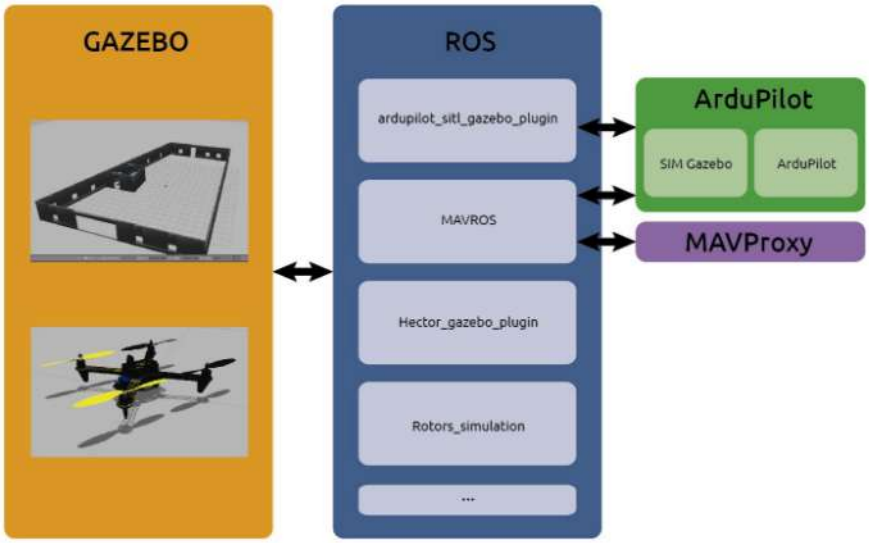


Figura 4.7: Esquema general de funcionamiento[2].

4.2.2. Configuración del entorno

Para realizar la instalación del vehículo en un sistema Ubuntu, primero se procede a instalar los paquetes bases:

```
sudo apt-get update

sudo apt-get install gawk make git curl cmake -y
```

Se instalan las dependencias para MAVProxy:

```
sudo apt-get install g++ python-pip python-matplotlib
python-serial python-wxgtk2.8 python-scipy python-opencv
python-numpy python-pyparsing ccache realpath libopencv-dev -y
```

[2]http://docs.erlerobotics.com/simulation/intro

Se instala MAVProxy:

```
sudo pip install future
sudo apt-get install libxml2-dev libxslt1-dev -y
sudo pip2 install pymavlink catkin\_pkg --upgrade
sudo pip install MAVProxy==1.5.2
```

Descargando ArUco 1.3.0 se procede a instalar:

```
cd ~/Downloads # Replace this with your Download directory
tar -xvzf aruco-1.3.0.tgz
cd aruco-1.3.0/
mkdir build && cd build
cmake ..
make
sudo make install
```

Se procede a descargar el proyecto ArduPilot que es un piloto automático de fuente abierta y del cual se utiliza su código para simular los vehículos aéreos no tripulados.

```
mkdir -p ~/simulation; cd ~/simulation
git clone https://github.com/erlerobot/ardupilot -b gazebo
```

Ahora se procederá a crear el espacio de trabajo de ROS, dentro de la carpeta *simulation*:

```
mkdir -p ~/simulation/ros_catkin_ws/src
cd ~/simulation/ros_catkin_ws/src
catkin_init_workspace
cd ~/simulation/ros_catkin_ws
catkin_make
source devel/setup.bash
```

Como ya se mencionó antes, cada proyecto en ROS necesitará de la instalación de paquetes específicos, por lo que dentro de la carpeta *src* de nuestro espacio de trabajo se procede a descargar lo siguiente:

```
git clone https://github.com/erlerobot/ardupilot_sitl_gazebo_plugin
git clone https://github.com/tu-darmstadt-ros-pkg/hector_gazebo/
git clone https://github.com/erlerobot/rotors_simulator -b sonar_plugin
git clone https://github.com/PX4/mav_comm.git
```

```
git clone https://github.com/ethz-asl/glog_catkin.git
git clone https://github.com/catkin/catkin_simple.git
git clone https://github.com/erlerobot/mavros.git
git clone https://github.com/ros-simulation/gazebo\_ros\_pkgs.\item     \$

# Añadir ejemplos los ejemplos de Python y C++.
git clone https://github.com/erlerobot/gazebo_cpp_examples
git clone https://github.com/erlerobot/gazebo_python_examples
```

Una vez realizadas las descargar de los paquetes se procede a realizar la compilación completa de todo el espacio de trabajo.

```
cd ~/simulation/ros_catkin_ws
catkin_make --pkg mav_msgs mavros_msgs gazebo_msgs
source devel/setup.bash
catkin_make -j 4
```

Dentro de la carpeta *.gazebo/models* se debe realizar la descarga del modelo del vehículo.

```
git clone https://github.com/erlerobot/erle_gazebo_models
mv erle_gazebo_models/* ~/.gazebo/models -f Gazebo
```

4.2.3. Lanzando la simulación

Para lanzar la simulación que se encuentra en línea como ejemplo disponible, se utilizan tres terminales distintas. En la primera se tipea:

```
source ~/simulation/ros_catkin_ws/devel/setup.bash

cd ~/simulation/ardupilot/ArduCopter

../Tools/autotest/sim_vehicle.sh -j 4 -f Gazebo
```

en la segunda terminal tipear:

```
source ~/simulation/ros_catkin_ws/devel/setup.bash

roslaunch ardupilot_sitl_gazebo_plugin erlecopter_spawn.launch
```

y en la tercera se ejecuta el nodo con el programa deseado:

```
source ~/simulation/ros_catkin_ws/devel/setup.bash

rosrun  ros_erle_pattern_follower ros_erle_pattern_follower
```

```
cmake_minimum_required(VERSION 2.8.3)
project(ros_erle_seguilineas)
```

Figura 4.8: Modificación al archivo CMakeLists.txt

```
<?xml version="1.0"?>
<package>
  <name>ros_erle_seguilineas</name>
  <version>0.0.0</version>
  <description>The ros_erle_seguilineas package</description>
```

Figura 4.9: Modificación al archivo package.xml

A partir de paquetes creados es posible crear otros paquetes modificando los archivos *CMakeLists.txt* y *package.xml*. Como ejemplo se puede crear un paquete llamado *ros_erle_seguilineas* y se compila utilizando:

```
catkin_make --pkg ros_erle_seguilineas
```

De acuerdo los pasos mencionados anteriormente para lanzar la simulación, en la tercera terminal se debería tipear el *rosrun* con el nombre nuevo del paquete.

```
source ~/simulation/ros_catkin_ws/devel/setup.bash

rosrun ros_erle_seguilineas ros_erle_seguilineas
```

4.3. Movimiento del vehículo

En la presente sección se explica como es posible controlar el vehículo aéreo, primeramente explicando como establecer los diversos modos de vuelo que se encuentran disponibles por el piloto automático, para luego hacer mención de las dos maniobras o movimientos automáticos para realizar el despegue y el aterrizaje, y por último la sobre escritura de los canales RC que sirven para dirigir al dron.

4.3.1. Establecimiento de los modos de vuelo

Dependiendo de que software se instale en la controladora de vuelo, si es APM o PX4, estarán disponibles diversos modos de vuelo. Para establecer los modos de vuelo se necesita incluir la librería de Mavros *SetMode.h* y para poder conocer que modo vuelo tiene el vehículo se debe añadir también *State.h*

```
#include <mavros_msgs/SetMode.h>
#include <mavros_msgs/State.h>
```

En el nodo para establecer los modos de vuelo se utiliza comunicación por servicios por lo que se debe crear el objeto de tipo de tipo cliente *ServiceClient* para hacer la llamada al servicio */mavros/set_mode*. Se crea un objeto de tipo *SetMode* para acceder y asignar valores a los miembros de solicitud de la clase *base_mode* y *custom_mode* en la cual vamos a asignar el modo de vuelo. En ROS una clase servicio contiene dos miembros, solicitud y respuesta (*request* and *response*). También contiene dos definiciones de clase, solicitud y respuesta.

```
ros::ServiceClient cl = n.serviceClient<mavros_msgs::SetMode>
("/mavros/set_mode");
mavros_msgs::SetMode srv_setMode;
srv_setMode.request.base_mode = 0;
srv_setMode.request.custom_mode = "GUIDED";
if(cl.call(srv_setMode)){
    ROS_ERROR("setmode send ok %d value:", srv_setMode.
    response.success);
}else{
    ROS_ERROR("Failed SetMode");
    return -1;
}
```

4.3.2. Maniobras automáticas

La controladora de vuelo o piloto automático ofrece la capacidad de realizar maniobras de vuelo preprogramadas como el despegue vertical del vehículo (*Takeoff*) para mantenerlo en vuelo a una altura determinada, así como el aterrizaje automático (*Land*).

La librería de mavros que incluye estos movimientos es *CommandTOL.h*. Para hacer estas maniobras se utiliza la comunicación bidireccional de ROS que son los servicios.

```
#include <mavros_msgs/CommandTOL.h>
```

Para realizar el despegue automático (*Takeoff*) se debe crear un cliente para el servicio */mavros/cmd/takeoff*, el objeto cliente del tipo *ServiceClient* se usa para hacer la llamada al servicio. De la misma manera se crea un objeto de tipo *CommandTOL* el cual sirve para acceder y asignar valores a los miembros de solicitud de la clase como la altitud, latitud y longitud. Esto hace que el vehículo despegue y permanezca a una altura deseada.

```
ros::ServiceClient takeoff_cl = n.serviceClient<mavros_msgs::
CommandTOL>("/mavros/cmd/takeoff");
mavros_msgs::CommandTOL srv_takeoff;
srv_takeoff.request.altitude = 3;
srv_takeoff.request.latitude = 0;
srv_takeoff.request.longitude = 0;
srv_takeoff.request.min_pitch = 0;
srv_takeoff.request.yaw = 0;
if(takeoff_cl.call(srv_takeoff))
{
   ROS_ERROR("srv_takeoff send ok %d", srv_takeoff.response.
   success);
}else
{
   ROS_ERROR("Failed Takeoff");
}
```

Para realizar la maniobra de aterrizaje (*Land*) se realiza el mismo procedimiento, pero ahora utilizando el servicio*/mavros/cmd/land*.

```
ros::ServiceClient land_cl = n.serviceClient<mavros_msgs::
CommandTOL>("/mavros/cmd/land");
mavros_msgs::CommandTOL srv_land;
srv_land.request.altitude = 3;
srv_land.request.latitude = 0;
srv_land.request.longitude = 0;
srv_land.request.min_pitch = 0;
srv_land.request.yaw = 0;
if(land_cl.call(srv_land))
{
   ROS_INFO("srv_land send ok %d", srv_land.response.success);
}else
{
   ROS_ERROR("Failed Land");
}
```

4.3.3. Manejo de los canales RC

Un piloto automático es el sistema usado para controlar la trayectoria de una aeronave sin la necesidad de que un operador humano sea requerido para establecer control por propia mano. En los drones, como ya se mencionó antes, el software del piloto automático establece un control sobre la velocidad de los motores del vehículo para que el operador pueda realizar maniobras de aceleración del vehículo y giros de guiñada, cabeceo y alabeo, sin que el vehículo pierda su estabilidad. En la sección 1.1.1 del marco teórico se explicó a detalle los movimientos mencionados.

Para controlar los movimientos de aceleración, guiñada, cabeceo y alabeo del vehículo Mavros utiliza la librería *OverrideRCIn.h*.

```
#include <mavros_msgs/OverrideRCIn.h>
```

En esencia Mavros realiza el manejo del vehículo de la misma manera y utilizando los mismos canales que utiliza la telemetría o radio control. Existen 7 canales, los canales 0, 1, 2 y 4 corresponden a los movimientos de alabeo, cabeceo, acelerador y guiñada. Cada canal puede ser cargado con un valor de entre 1000 y 2000, siendo 1500 el valor medio del canal, por lo que si un canal recibe un valor menor a 1500 el vehículo realiza movimiento en un sentido y si recibe un valor mayor a 1500 se realiza el movimiento en sentido opuesto.

La siguiente tabla muestra una lista de la función que realiza cada canal así como los valores que pueden ser cargados en ellos.

Canales RC			
Canal	Función	Valor Min	Valor Max
0	Alabeo	1000	2000
1	Cabeceo	1000	2000
2	Acelerador	1000	2000
3	Guiñada	1000	2000

Cuadro 4.1: Funciones de los canales RC.

Para utilizar los mensajes de control RC se debe crear inicialmente un objeto de tipo *Publisher* esto es debido a que el método *advertise* regresa un objeto de este tipo. El método *advertise()* sirve para dos propósitos 1) contiene un método *publish()* que le permite publicar mensajes sobre el tema con el que fue creado, y 2) cuando sale de alcance, se desadvertirá automáticamente [3]. Utilizando el objeto *Publisher* se le dirá al nodo maestro que se va a estar publicando un mensaje de tipo *mavros_msgs::OverrideRCIn* en el tópico */mavros/rc/override*. El segundo argumento es el tamaño de la cola. En este caso, si se publica demasiado rápido, almacenará en un *buffer* hasta 10 mensajes

[3]Fuente: http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28c%2B%2B%29

como máximo antes de comenzar a deshacerse de los viejos. Se crea un objeto de la clase *OverrideRCIn* para acceder a los canales y poder asignarles valores. Por último esta información se publica en el tópico utilizando el método *publish*.

```
ros::Publisher pub;
pub = nh.advertise<mavros_msgs::OverrideRCIn>("/mavros/rc/
override", 10);
mavros_msgs::OverrideRCIn msg;
msg.channels[0] = 0;
msg.channels[1] = 0;
msg.channels[2] = 0;
msg.channels[3] = 0;
msg.channels[4] = 0;
msg.channels[5] = 0;
msg.channels[6] = 0;
msg.channels[7] = 0;
pub.publish(msg);
```

4.4. Imágenes ROS con OpenCV

El *Erle-Copter* cuenta con dos cámaras, una frontal y otra inferior, las cuales están descritas en el archivo *erlecopter.xacro* que se encuentra en la dirección *ros_catkin_ws/src/ardupilot_sitl_gazebo_plugin/ardupilot_sitl_gazebo_plugin/urdf/erlecopter.xacro* y es posible acceder a ellas desde la línea de comandos para visualizar en el simulador Gazebo. El comando

```
rosrun image_view image_view image:=/erlecopter/front/
image_front_raw
```

realiza un acceso a la cámara frontal del vehículo, mientras que el comando

```
rosrun image_view image_view image:=/erlecopter/bottom/image_raw
```

se realiza un acceso a la cámara inferior. En las figuras 4.10 y 4.11 se observa el fragmento de código referente a las cámaras que se describen en el archivo *.xacro*.

```
<!-- Front facing camera -->
<xacro:include filename="$(find ardupilot_sitl_gazebo_plugin)/urdf/sensors/generic_camera.urdf.xacro" />
<xacro:generic_camera
  name="erlecopter/front"
  parent="base_link"
  ros_topic="image_front_raw"
  cam_info_topic="camera_front_info"
  update_rate="25"
  res_x="640"
  res_y="360"
  image_format="R8G8B8"
  hfov="81"
  framename="erlecopter_frontcam">
  <origin xyz="0.17 0.0 0.0" rpy="0 0 0"/>
</xacro:generic_camera>
```

Figura 4.10: Archivo *erlecopter.xacro* cámara posterior.

```
<!-- Downward facing camera -->
<xacro:include filename="$(find ardupilot_sitl_gazebo_plugin)/urdf/sensors/generic_camera.urdf.xacro" />
<xacro:generic_camera
  name="erlecopter/bottom"
  parent="base_link"
  ros_topic="image_raw"
  cam_info_topic="camera_info"
  update_rate="60"
  res_x="640"
  res_y="360"
  image_format="R8G8B8"
  hfov="81"
  framename="erlecopter_bottomcam">
  <origin xyz="0.14 0.0 -0.02" rpy="0 ${M_PI/2} 0"/>
</xacro:generic_camera>
```

Figura 4.11: Archivo *erlecopter.xacro* cámara inferior.

A continuación se describe como establecer una conexión entre ROS y OpenCV, convirtiendo imágenes de ROS generadas en el simulador a imágenes capaces de ser trabajadas con OpenCV, y viceversa, utilizando la biblioteca *CvBridge*[4] perteneciente a ROS.

[4]Fuente: http://wiki.ros.org/cv_bridge/Tutorials/ConvertingBetweenROSImagesAndOpenCVImagesPython

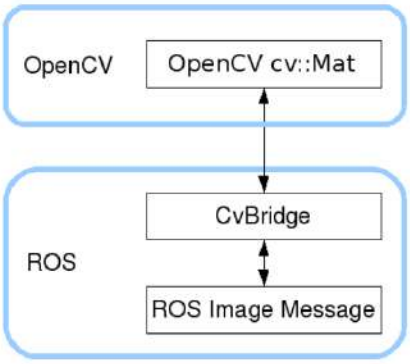


Figura 4.12: Interfaz *CvBridge*.

CvBridge define un tipo de *CvImage* que contiene una imagen de OpenCV, su codificación y un encabezado ROS.

```
1  namespace cv_bridge {
2
3  class CvImage
4  {
5  public:
6    std_msgs::Header header;
7    std::string encoding;
8    cv::Mat image;
9  };
10
11 typedef boost::shared_ptr<CvImage> CvImagePtr;
12 typedef boost::shared_ptr<CvImage const> CvImageConstPtr;
13
14 }
```

Figura 4.13: Constructor de *cv_bridge*.

El puente proporciona las funciones *toCvCopy* y *toCvShare* como se muestra en la figura 4.14.

```
1  // Case 1: Always copy, returning a mutable CvImage
2  CvImagePtr toCvCopy(const sensor_msgs::ImageConstPtr& source,
3                      const std::string& encoding = std::string());
4  CvImagePtr toCvCopy(const sensor_msgs::Image& source,
5                      const std::string& encoding = std::string());
6
7  // Case 2: Share if possible, returning a const CvImage
8  CvImageConstPtr toCvShare(const sensor_msgs::ImageConstPtr& source,
9                            const std::string& encoding = std::string());
10 CvImageConstPtr toCvShare(const sensor_msgs::Image& source,
11                           const boost::shared_ptr<void const>& tracked_object,
12                           const std::string& encoding = std::string());
```

Figura 4.14: Funciones pertenecientes al puente *CvBridge*.

Para las codificaciones de imagen populares, *CvBridge* opcionalmente realizará conversiones de profundidad de color o píxeles según sea necesario. Para usar esta función, se especifica la codificación como una de las siguientes cadenas:

mono8 : CV_8UC1, imagen en escala de grises.

mono16 : CV_16UC1, imagen en escala de grises de 16 bits.

bgr8 : CV_8UC3, imagen en color con orden de color azul-verde-rojo.

rgb8 : CV_8UC3, imagen en color con orden de color rojo-verde-azul.

bgra8 : CV_8UC4, imagen en color BGR con un canal alfa.

rgba8 : CV_8UC4, imagen en color RGB con un canal alfa.

En el archivo *package.xml* y *CMakeLists.xml* se deben agregar las siguientes dependencias:

sensor_msgs
cv_bridge
roscpp
std_msgs
image_transport

En la figura 4.15 se muestra un ejemplo de como luciría un nodo de ROS que trabaje con imágenes provenientes del *Erle-Copter*.

```
#include <ros/ros.h>
#include <image_transport/image_transport.h>
#include <cv_bridge/cv_bridge.h>
#include <opencv2/imgproc/imgproc.hpp>
#include <opencv2/highgui/highgui.hpp>

image_transport::ImageTransport it;
image_transport::Publisher image_pub;

void imageCallback(const sensor_msgs::ImageConstPtr& msg)
{
    try
    {
        cv::Mat InImage;
        InImage = cv_bridge::toCvShare(msg, "bgr8")->image;
                ...
    }
    catch (cv_bridge::Exception& e)
    {
        ROS_ERROR("Could not convert from '%s' to 'bgr8'.", msg->encoding.
            c_str());
    }
}

int main(int argc, char **argv)
{
    ros::init(argc, argv, "image_listener");
    ros::NodeHandle nh;
    image_transport::Subscriber image_sub(nh);
    image_sub = it.subscribe("/erlecopter/bottom/image_raw", 1,
        imageCallback);
    image_pub = it.advertise("/erlecopter/bottom/image_raw", 1);
    ros::spin();
}
```

Figura 4.15: Propuesta de código para suscribirse a imágenes en ROS.

```
# include <image_transport / image_transport.h>
# include <cv_bridge / cv_bridge.h>
# include <opencv2 / imgproc / imgproc.hpp>
# include <opencv2 / highgui / highgui.hpp>
```

El nodo debe incluir bibliotecas que funcionan para la suscripción y publicación a imágenes en ROS, trabajar con el puente *CvBridge* y realizar procesamiento de imágenes con OpenCV. Los objetos suscriptores y publicadores al tópico de la imagen se declaran utilizando *image_transport*.

```
image_transport :: ImageTransport it ;
image_transport :: Suscriptor image_sub (nh);
image_transport :: Publisher image_pub ;
```

Se debe suscribir al tópico de la cámara inferior del vehículo */erlecopter/bottom/image_raw*. ROS llamará a la función *imageCallback* cada vez que un nuevo mensaje sea recibido.

```
image\_sub = it.subscribe("/erlecopter/bottom/image_raw", 1,
imageCallback);
image\_pub = it.advertise("/erlecopter/bottom/image_raw", 1);
```

imageCallback es la función para la devolución de la llamada que se llama cada vez que llega un mensaje nuevo del tópico */erlecopter/bottom/image_raw*. Dentro de esta devolución de llamada se convierte el mensaje de la imagen de ROS a una imagen *CvImage* para trabajar con OpenCv.

```
cv::Mat InImage;
InImage = cv_bridge::toCvShare(msg, "bgr8");
```

4.5. Detección de marcadores

Los marcadores de Aruco [39] son cuadrados de color negro y blanco con un identificador numérico codificado, estos marcadores muestran principalmente robustez para su detección, dando la capacidad de estimar la pose de la cámara a partir de ellos [40] y la disponibilidad de bibliotecas compatibles con OpenCV [41]. La referencia a la clase *aruco::MarkerDetector*[5] tiene dos funciones miembro públicas *detect*.

```
detect (cv::Mat                &input,
        std::vector< Marker > &detectedMarkers,
        CameraParameters       camParams = CameraParameters(),
        float                  markerSizeMeters=-1
       ) throw (cv::Exception)

detect (cv::Mat                &input,
        std::vector< Marker > &detectedMarkers,
        cv::Mat                camMatrix=cv::Mat(),
        cv::Mat                distCoeff=cv::Mat(),
        float                  markerSizeMeters=-1
       ) throw (cv::Exception)
```

El vector de salida *&detectedMarkers* contiene los marcadores detectados por la función. Este vector se declara de la clase *aruco::Marker*, esta clase representa un marcador, siendo un vector que contiene las cuatro esquinas del marcador.

```
draw (cv::Mat &     in,
      cv::Scalar    color,
      int           lineWidth=1,
      bool          writeId=true
     )
```

El método *draw* dibujará los marcadores encontrados en la imagen. Para obtener el centro del marcador se usa la función *getCenter* que devuelve el centroide del marcador.

```
cv::Point2f Marker::getCenter()const
{
   cv::Point2f cent(0,0);
   for(size_t i=0;i<size();i++){
     cent.x+=(*this)[i].x;
     cent.y+=(*this)[i].y;
   }
   cent.x/=float(size());
   cent.y/=float(size());
   return cent;
}
```

[5]Fuente: http://docs.ros.org/electric/api/aruco_pose/html/classaruco_1_1MarkerDetector.html

4.6. Diseño del entorno en Gazebo

De los comandos de lanzamiento mostrados con anterioridad, se lanza la simulación utilizando

```
roslaunch ardupilot_sitl_gazebo_plugin erlecopter_spawn.launch
```

el cual indica que el archivo de lanzamiento que se está utilizando es el *erlecopter_spawn.launch*. Dentro de este archivo se especifica que el archivo mundo utiliza el archivo de lanzamiento, el cual es *empty.world* que se encuentra en la dirección

```
/home/carlos/simulation/ros_catkin_ws/src/ardupilot_sitl_
gazebo_plugin/ardupilot_sitl_gazebo_plugin/worlds/empty_world
```

Se procedió a crear otro archivo de lanzamiento *.launch* junto con otro archivo mundo *.world* para desarrollar un entorno de simulación exterior que se asemeje al entorno universitario para realizar las pruebas pertinentes.

4.6.1. Elaboración de modelos de caminos

Los archivos *road.world* y *road_texture.world* que se encuentran en la carpeta *worlds*; que se instala con gazebo, se pueden lanzar tipeando en una terminal:

```
gazebo road.world

gazebo worlds/road_textures.world
```

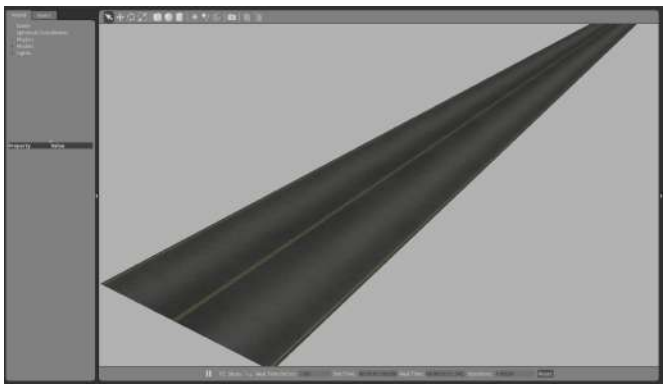


Figura 4.16: archivo *road.world*.

Estos archivos muestran diferentes tipos de materiales para caminos que se diseñados utilizando el elemento $< road >< /road >$ tiene solo como padre al elemento $< world >< /world >$. Se define el ancho del camino con $< width >< /width >$ y su trayectoria mediante una serie de puntos $< point >< /point >$ que se entrelazan, a cada punto se le deben especificar sus coordenadas x, y, z, las unidades que utiliza el simulador son metros. Para añadir un material a los modelos se utiliza $< material >< /material >$, y funciona de la misma manera que en los modelos. En

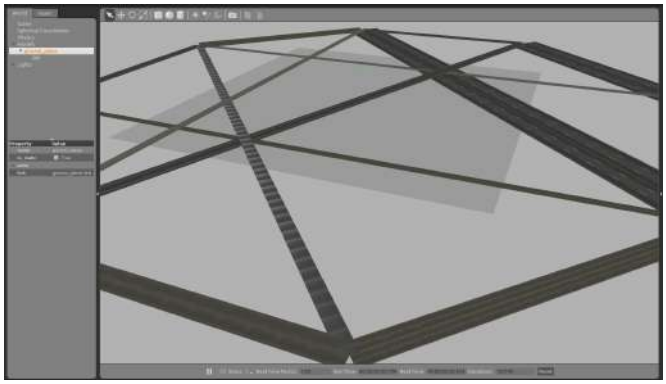


Figura 4.17: archivo *road_textures.world*.

la carpeta */usr/share/gazebo-4.1/media/materials/textures* están disponibles diversas texturas en diversos formatos para los caminos, para seleccionar el material adecuado dentro del elemento $< script >< /script >$ se indica la URI $< uri >< /uri >$ del archivo *script* del material, este es un archivo *.material* que se encuentra en la carpeta */usr/share/gazebo-4.1/media/materials/scripts*. De la misma manera se debe de indicar el nombre $< name >< /name >$ del *script* a utilizar dentro del archivo *.material*, este señala que textura se utilizará de la carpeta. En la imagen 4.18 se muestra el diagrama del formato SDF para lo ya mencionado para el elemento $< road >< /road >$, el cual se encuentra disponible en la pagina oficial *http://sdformat.org*.

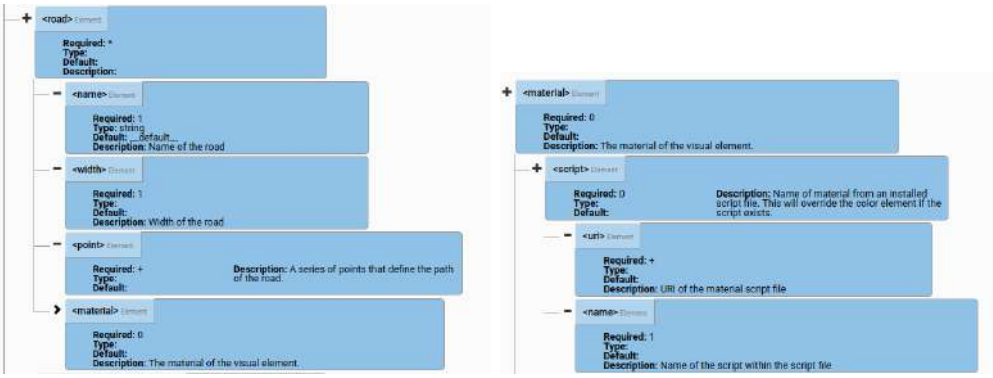


Figura 4.18: *Formato SDF para los caminos.*

El fragmento de código que se muestra a continuación genera un camino de dos metros de ancho empezando en el centro de la simulación con coordenadas $(0, 0, 0{,}005)$ dirigiéndose a la coordenada $(10, 0, 0{,}005)$, siguiendo a $(10, 5, 0{,}005)$ y terminando en $(0, 5, 0{,}005)$. Es importante tomar en cuenta que el valor de z representa la altura a la que estará el modelo del camino, por lo que ésta no debe ser igual a la altura de cualquier otro modelo de piso, como el *ground_plane*, debido a que si están a la misma altura la imagen se observa borrosa en la simulación. En el código se indica que se utilizará el archivo de material *gazebo.material* y dentro de éste $Gazebo/Trunk$ para indicar la textura.

```
<?xml version="1.0" ?>
<sdf version="1.5">
 <world name="default">

  <scene>
    <grid>false</grid>
  </scene>
  <include>
    <uri>model://sun</uri>
  </include>
  <include>
    <uri>model://ground_plane</uri>
  </include>

  <road name="my_road">
    <width>2</width>
    <point>0 0 .005</point>
    <point>10 0 .005</point>
    <point>10 5 .005</point>
    <point>0 5 .005</point>
    <material>
      <script>
        <uri>file://media/materials/scripts/gazebo.material</uri>
        <name>Gazebo/Trunk</name>
      </script>
    </material>
  </road>
 </world>
</sdf>
```

Figura 4.19: Archivo *model.sdf* para un camino.

La imagen que se muestra en la figura 4.20 es el camino que se desarrolló a partir del archivo *model.sdf*.

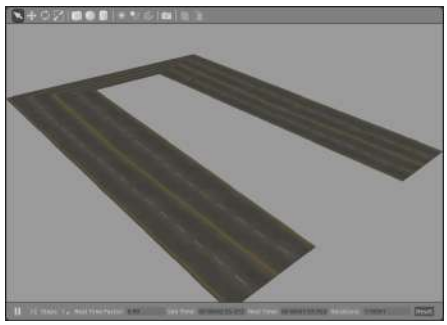


Figura 4.20: archivo *road2.world*.

4.6.2. Elaboración de modelos de superficies

El modelo de superficie disponible por Gazebo *mud_box* se puede descargar desde el menú insertar directamente en el simulador.

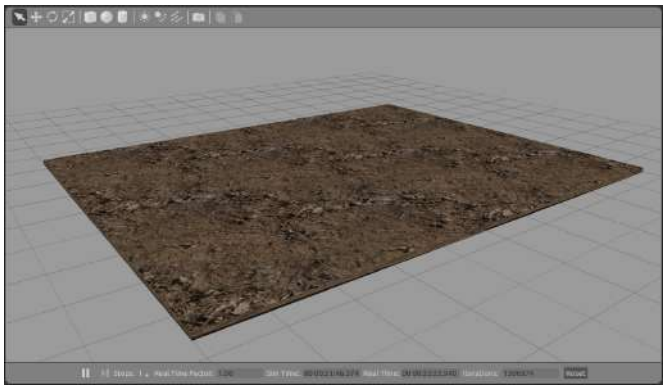


Figura 4.21: Modelo *Mud Box*.

Los modelos que se descargan se añaden directamente a la carpeta *models* ubicada en la dirección */home/carlos/.gazebo/models* . Como ya se mencionó en la sección 3.3.4 un modelo cuenta una estructura de predeterminada de archivos y carpetas, es por esto que el modelo que se desarrolla contendrá lo siguiente:

- archivo *model.config*
- archivo *model.sdf*
- carpeta *materials*
    - carpeta *textures*
    - carpeta *scripts*

En el archivo *model.config* se indica el nombre del modelo, la versión SDF, el autor y una breve descripción. En el archivo *model.sdf* se describe la superficie en formato SDF, diseñándola desde el punto de vista de este formato una superficie puede ser descrita como un enlace $< link >< /link >$ que será una figura rectangular de ciertas dimensiones de largo y ancho con un grosor, con elementos visuales para que se observe en ella la imagen que será su textura, tal y como se distingue en el modelo de la figura 4.21. Dentro del enlace, el elemento de colisión $< collision >< /collision >$ describirá un elemento geométrico $< box >< /box >$ con valores x, y, z para indicar su dimensión.

```
<collision name="collision">
  <geometry>
    <box>
      <size>22 20 0.003</size>
    </box>
  </geometry>
</collision>
```

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

<model>
  <name>Grass Box</name>
  <version>1.0</version>
  <sdf version="1.4">model-1_4.sdf</sdf>
  <sdf version="1.5">model.sdf</sdf>

  <author>
    <name>Enrique Montalvo</name>
    <email>cenrique789@gmail.com</email>
  </author>

  <description>
    A grass texture plane.
  </description>
</model>
```

Figura 4.22: Archivo *model.config* para superficie.

El elemento visual $<$ *visual* $><$ */visual* $>$ del enlace se declara en una pose específica $<$ *pose* $><$ */pose* $>$ indicado 3 valores para su posición (*x,y,z*) y 3 para su orientación (*roll, pitch, yaw*), un elemento geométrico $<$ *box* $><$ */box* $>$ para definir su forma y un material $<$ *material* $><$ */material* $>$ para seleccionar la imagen que desea visualizar en la superficie. Es importante señalar que un enlace puede tener un número de elementos visuales indefinido por lo que si se desea crear un modelo de superficie que tenga diferentes texturas, basta con añadir al código más elementos visuales que contengan otras texturas.

```
      <visual name="visual_1">
        <pose>-2 2.5 0 0 0 0</pose>
        <cast_shadows>false</cast_shadows>
        <geometry>
          <box>
            <size>4 5 0.003</size>
          </box>
        </geometry>
        <material>
          <script>
            <uri>model://grass_box/materials/scripts</uri>
            <uri>model://grass_box/materials/textures</uri>
            <name>vrc/mud</name>
          </script>
        </material>
      </visual>
```

También es importante mencionar que la imagen que se carga como textura se alinea al tamaño del elemento geométrico $<$ *box* $><$ */box* $>$ señalado dentro de $<$ *visual* $><$ */visual* $>$ y no al elemento $<$ *box* $><$ */box* $>$ señalado dentro de $<$ *collision* $><$

/collision > . Para que la imagen no se observe pixeleada es recomendable hacer elementos visuales pequeños y añadirlos en una posición adecuada para que no se traslapen. El archivo *model.sdf* se muestra en la figura 4.23.

```
<?xml version="1.0" ?>
<sdf version="1.5">
  <model name="grass_box">
    <static>true</static>
    <link name="link">

      <collision name="collision">
        <geometry>
          <box>
            <size>22 20 0.003</size>
          </box>
        </geometry>
      </collision>

      <visual name="visual_1">
        <pose>-2 2.5 0 0 0 0</pose>
        ...
      </visual>

      <visual name="visual_2">
        <pose>2 2.5 0 0 0 0</pose>
       ...
        </material>
      </visual>

      <visual name="visual_3">
        <pose>2 -2.5 0 0 0 0</pose>
        ...
      </visual>

      ...

    </link>
  </model>
</sdf>
```

Figura 4.23: Archivo *model.sdf* para superficie.

Dentro de la carpeta *materials* se encontrarán las dos carpetas *scripts* y *textures*, dentro de la carpeta *textures* se deben almacenar las imágenes en formatos jpg, png, etc. que utiliza el modelo. El archivo *grass.material* que se muestra en la figura 4.24 indica que imagen se utilizará como textura, este archivo se encuentra dentro de la carpeta *scripts*, a éste mismo se refiere el elemento visual $< name > vrc/mud < /name >$ de *model.sdf*.

```
material vrc/mud
{
        technique
        {
                pass
                {
                        ambient 0.5 0.5 0.5 1.0
                        diffuse 0.5 0.5 0.5 1.0
                        specular 0.2 0.2 0.2 1.0 12.5

                        texture_unit
                        {
                                texture grass2.jpg
                                filtering anistropic
                                max_anisotropy 16
                        }
                }
        }
}
```

Figura 4.24: Archivo *grass.material* para superficie.

La figura 4.25 muestra el modelo de superficie desarrollado que tiene dimensiones de *22* por *20* y se encuentra a una altura sobre el eje z de *0.003*.

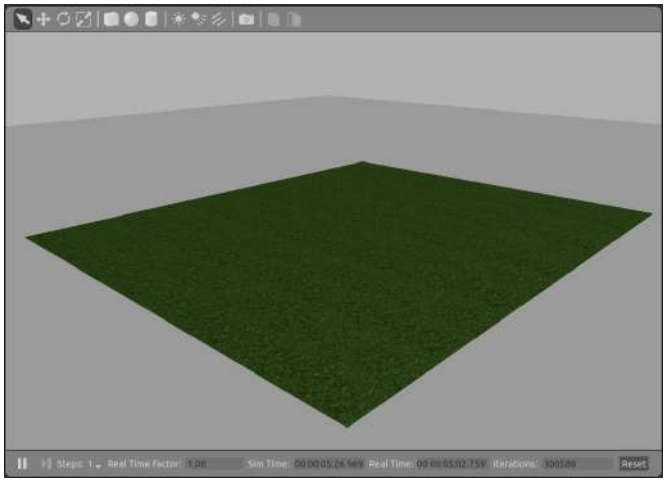


Figura 4.25: Modelo *Grass box*.

4.6.3. Elaboración de modelos de edificios

El desarrollo de entornos interiores y exteriores ha llevado a Gazebo a ser un simulador muy utilizado para probar toda clase de robots. La herramienta *Building Editor*[6] que proporciona Gazebo permite de forma muy rápida construir un edificio directamente dentro de la simulación. Utilizando este editor de edificios es posible importar una imagen que tenga los planos de una estructura y sobre estos construir el edificio, seleccionando los iconos correspondientes para levantar paredes, añadir ventanas y puertas o agregar escaleras, los cuales se encuentran en el menú del editor. A la estructura se le pueden añadir la cantidad de pisos que se requieran, así como texturas y pintura en las paredes. Para detallar las medidas de los componentes añadidos a la estructura se pueden utilizar los inspectores que ofrece el *Building Editor*.

Una vez terminado el edificio se debe guardar como un archivo *model.sdf* y debe ser puesto dentro de la carpeta de modelos */home/carlos/.gazebo/models* . Para que el modelo pueda ser añadido en cualquier otra simulación, la carpeta que contendrá al modelo, debe contener el archivo *model.config* que se muestra en la figura 4.26 y el archivo del modelo *model.sdf*, al igual que como se mencionó en la creación de otros modelos.

```
<?xml version="1.0"?>
<model>
  <name>CLIR</name>
  <version>1.0</version>
  <sdf version="1.5">model.sdf</sdf>

  <author>
    <name>Carlos EACM</name>
    <email>cerinque789@gmail.com</email>
  </author>

  <description>
    Un laboratorio.
  </description>
</model>
```

Figura 4.26: Archivo *model.config*

Con el fin de generar un entorno interior de simulación, se tomó como modelo al laboratorio escolar como se muestra en la figura 4.27.

[6]Fuente: http://gazebosim.org/tutorials?tut=building_editor

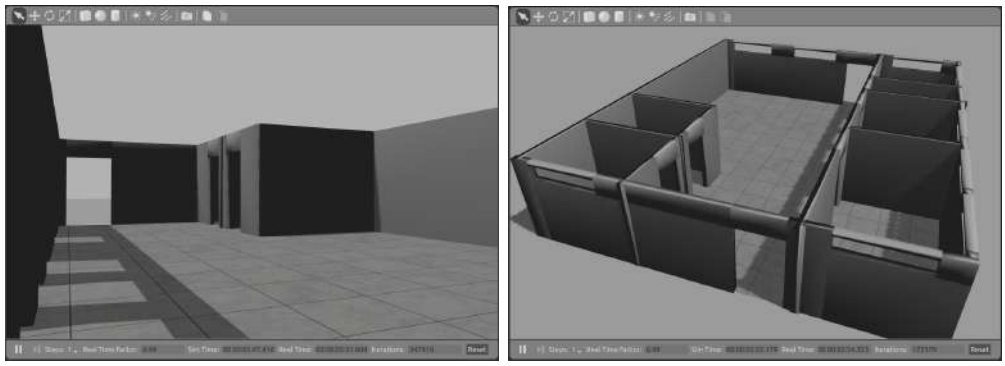


Figura 4.27: Diseño del laboratorio CLIR en Gazebo.

4.6.4. Elaboración del mundo

Como ya se mencionó en la sección 3.3.6 el archivo *mundo.world* contiene todos los elementos que se incluyen en la simulación.

```
<?xml version="1.0" ?>
<sdf version="1.5">
<world name="default">
    <include>{\tiny }
      <uri>model://ground_plane</uri>
    </include>

    <include>
      <uri>model://grass_box</uri>
      <pose>0 0 0 0 0 0</pose>
    </include>

          <include>
      <uri>model://pine_tree</uri>
      <pose>9 5 0 0 0 0</pose>
    </include>

     <include>
      <uri>model://CLIR</uri>
      <pose> 4 -14 0 0 0 0</pose>
    </include>

     ...

</world>
</sdf>
```

Figura 4.28: Fragmento del archivo *.world*

Para cargar un modelo al archivo mundo se debe ubicar en la carpeta */home/-carlos/.gazebo/models* del sistema. Cada modelo se añade utilizando el elemento < *include* >< */include* > especificando el URI *model://nombre_modelo* del modelo < *uri* >< */uri* > . Para posicionar cada modelo se establece su pose < *pose* >< */pose* > que es respecto al marco de referencia del mundo. En la figura 4.28 se muestra un fragmento de código del archivo mundo y en la figura 4.29 se observa el entorno final.

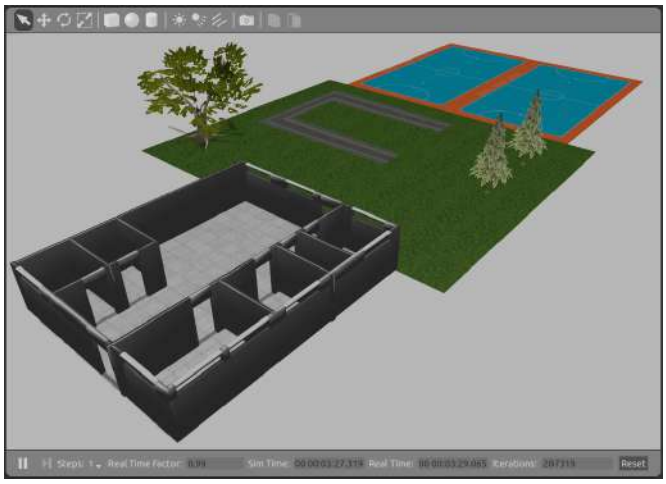


Figura 4.29: Entorno de simulación desarrollado.

Capítulo 5

Experimentos

En la siguiente sección se exponen los resultados que se obtuvieron a partir de tres experimentos que se realizaron con el vehículo aéreo *Erle-Copter* simulado en Gazebo. El control de posición fue el primer experimento realizado, este control hace que el vehículo se dirija a una posición deseada en base a la tripleta de coordenadas *x, y, z*. El segundo experimento se realizó de igual manera utilizando el control de posición pero haciendo que el punto meta cambie en cada iteración del programa describiendo una trayectoria circular. Debido a que este trabajo de tesis se realizó simultáneamente con la tesis realizada por Manuel Poot [42], el tercer experimento realizado incluye resultados obtenidos en éste, en dicho experimento se utiliza la cámara inferior del vehículo para detectar los marcadores de Aruco y realizar un aterrizaje en el mismo marcador implementando un controlador PID para el control de los movimientos del dron.

5.1. Control de posición

En la etapa de experimentación se realizó la implementación del control fuera de borda (*Offboard*) que se utiliza para hacer que el vehículo se dirija y mantenga en una posición específica hasta que se le asigne alguna otra posición. El tópico de Mavros que se utiliza para publicar la información de posicionamiento es *mavros/setpoint_position/local*. En la figura 5.1 se muestra el diagrama a bloques del grafo de nodos y tópicos del programa ejecutado.

El control mencionado dirige al dron a diversas posiciones basadas en un plano *x, y, z* en metros, donde *0, 0, 0* es la posición donde inicia el despeque. El experimento se realizó con 4 posiciones, todas inician en (0, 0, 0); es decir, que el vehículo se enciende y enseguida realiza el movimiento hacia una sola posición y la simulación se da por terminada. Las posiciones implementadas fueron (0, -9, 5), (4, 3, 6), (8, 10, 7) y (15, 20, 18). Para realizar la gráfica de las posiciones que adquiere el vehículo se utiliza la herramienta *rqt* adquiriendo la información del tópico */mavros/local_position/local/pose/pose/position*.

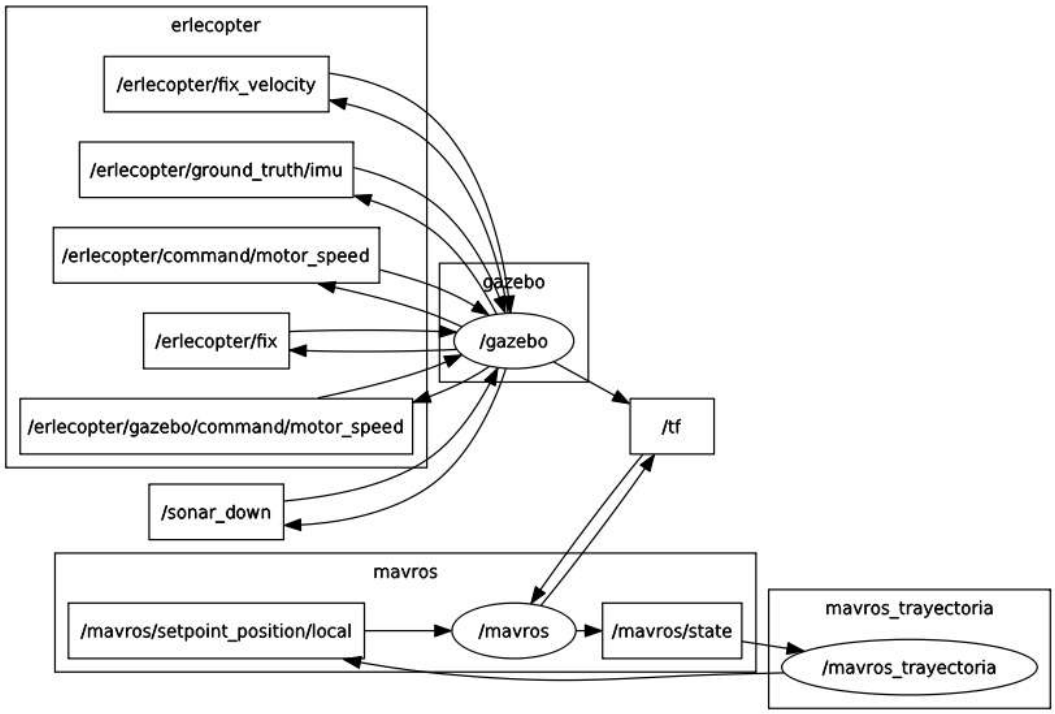


Figura 5.1: Grafo de ROS para el control de posición usando la herramienta rqt_graph.

En las figuras 5.2, 5.3, 5.4 y 5.5 se muestra la información de las posiciones que se establecen en el dron. Como se puede observar las gráficas comparan la posición de las tres coordenadas del vehículo contra el tiempo que tardar en realizar adecuadamente la traslación hacia la posición que se le indica. Se puede observar que el vehículo se establece en la posición indicada para las tres coordenadas en un tiempo muy similar, realizando los movimientos entre los 10 y 20 segundos. De igual manera, en las tres coordenadas el punto establecido es alcanzado adecuadamente por el vehículo llegando a la posición sin ningún tipo de sobretiro en su respuesta y permaneciendo en la posición hasta terminar la simulación. Todas las gráficas muestran un comportamiento bastante similar sin importar las distintas posiciones que se establezcan en el dron.

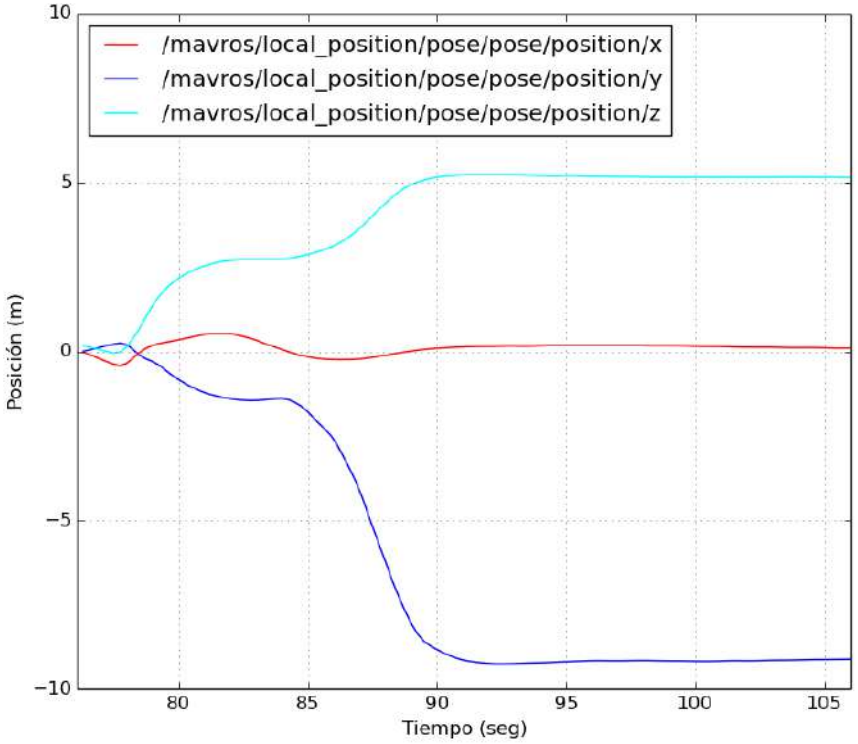


Figura 5.2: Gráfica para la posición (0, -9, 5) utilizando el tópico *local_position*.

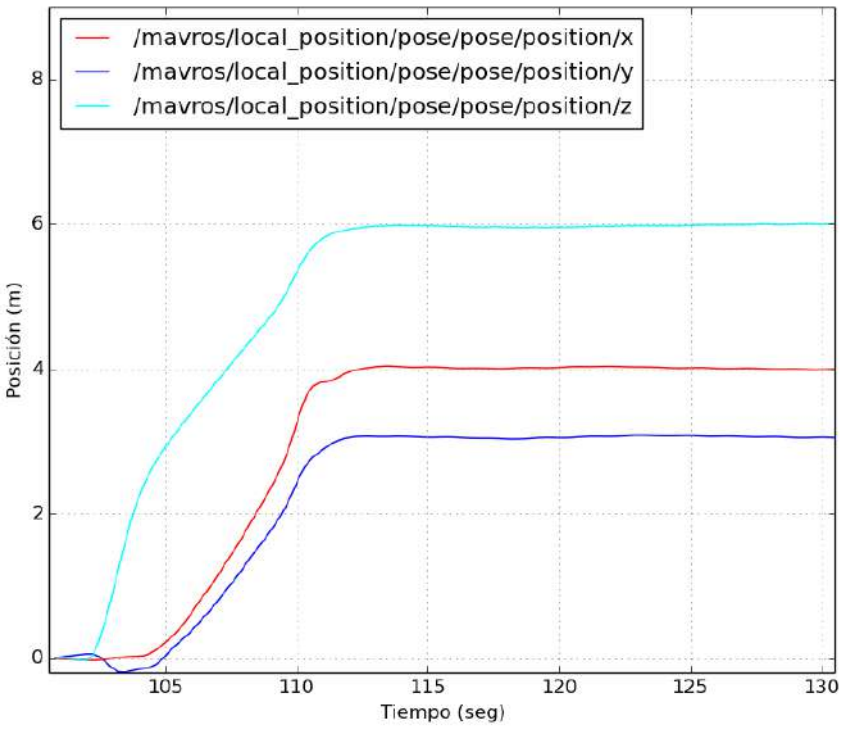


Figura 5.3: Gráfica para la posición (4, 3, 6) utilizando el tópico *local_position*.

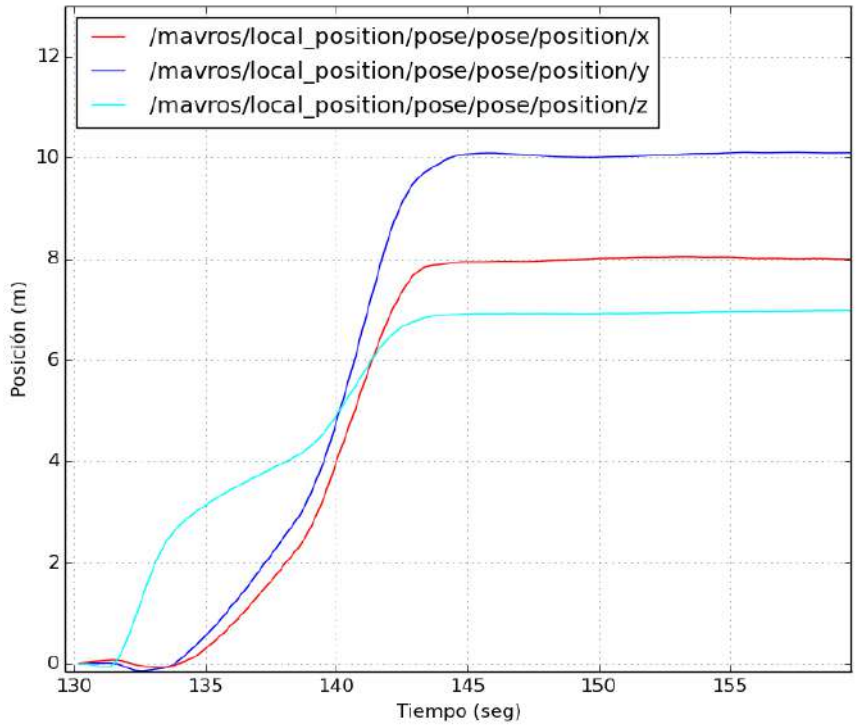


Figura 5.4: Gráfica para la posición (8, 10, 7) utilizando el tópico *local_position*.

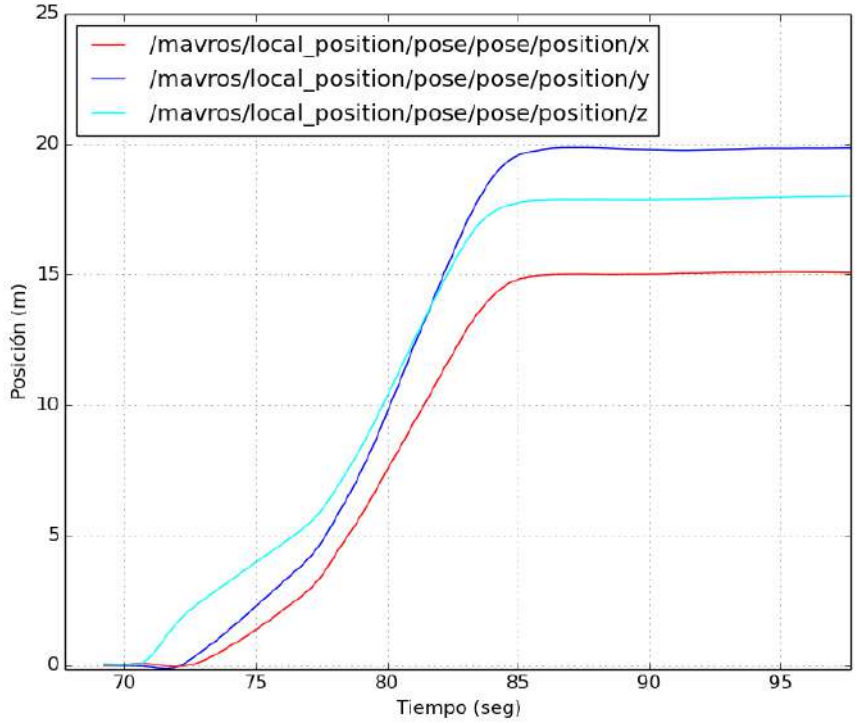


Figura 5.5: Gráfica para la posición (15, 20, 18) utilizando el tópico *local_position*.

5.1.1. Error del control de posición

Para demostrar el error que el dron tiene al realizar los movimientos hacia las posiciones que se le programan se utiliza la ecuación de error:

$$error = \sqrt{(x - x_o)^2 + (y - y_o)^2 + (z - z_o)^2}$$

donde x, y y z son las posiciones reales y x_o, y_o y z_o son las posiciones deseadas.

Las figuras 5.6 y 5.8 muestran el error por coordenada cuando se manda al vehículo a las posiciones (0, -9, 5) y (4, 3, 6) mencionadas anteriormente. Utilizando las mismas posiciones el error total se muestra en las figuras 5.7 y 5.9. En las gráficas se puede observar como el error disminuye en un lapso de tiempo de 15 segundos. Para las dos posiciones se observa correctamente como tanto los errores por coordenadas como el error total disminuyen a cero lo que indica que el vehículo se posiciona adecuadamente en las coordenadas indicadas.

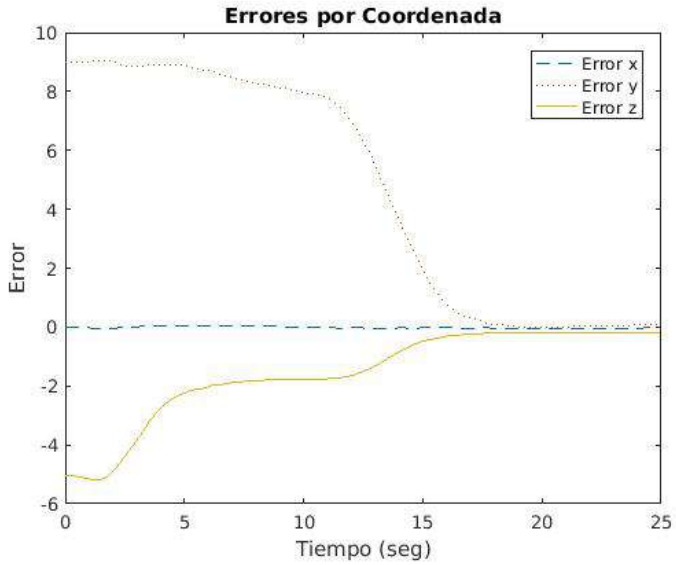


Figura 5.6: Errores por coordenada para la posición (0, -9, 5).

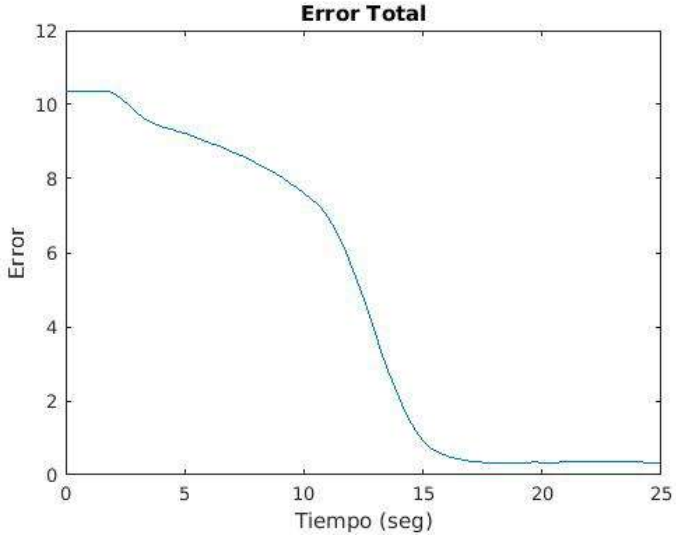


Figura 5.7: Error total para la posición (0, -9, 5).

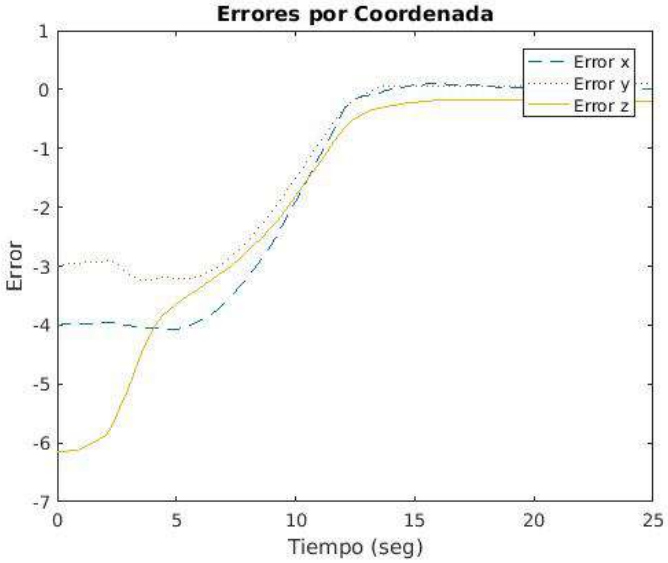


Figura 5.8: Errores por coordenada para la posición (4, 3, 6).

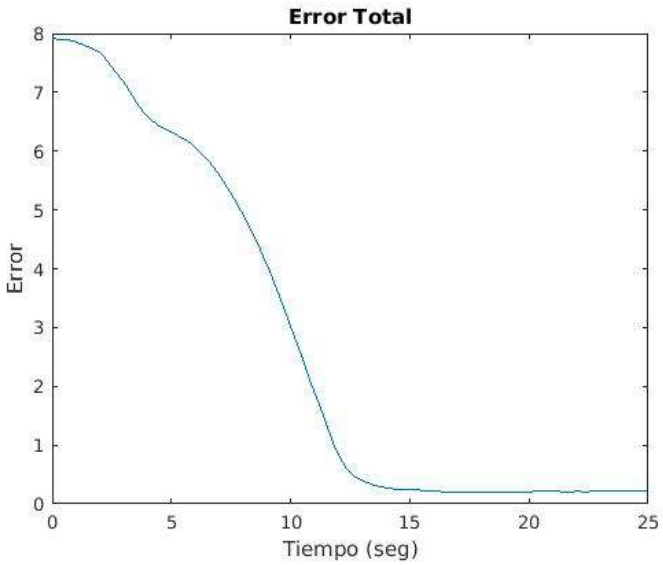


Figura 5.9: Error total para la posición (4, 3, 6).

5.2. Seguimiento de una trayectoria

El control de posición también es utilizado para que el dron realice el seguimiento de una trayectoria. La trayectoria definida para que el vehículo realice es un circulo con un radio de $3m$ a una altura determinada. De igual manera se utiliza la herramienta *rqt* para mostrar las gráficas del posicionamiento obteniendo la información del tópico */mavros/local_position/local/pose/pose/position*. La figura 5.10 muestra la información correspondiente al seguimiento de la trayectoria, que a diferencia de las anteriores gráficas, en ésta solo se grafican los datos el eje x y el eje y ya que el eje z se refiere a la altura donde el vehículo realizará el movimiento, la cual no es indispensable de graficar.

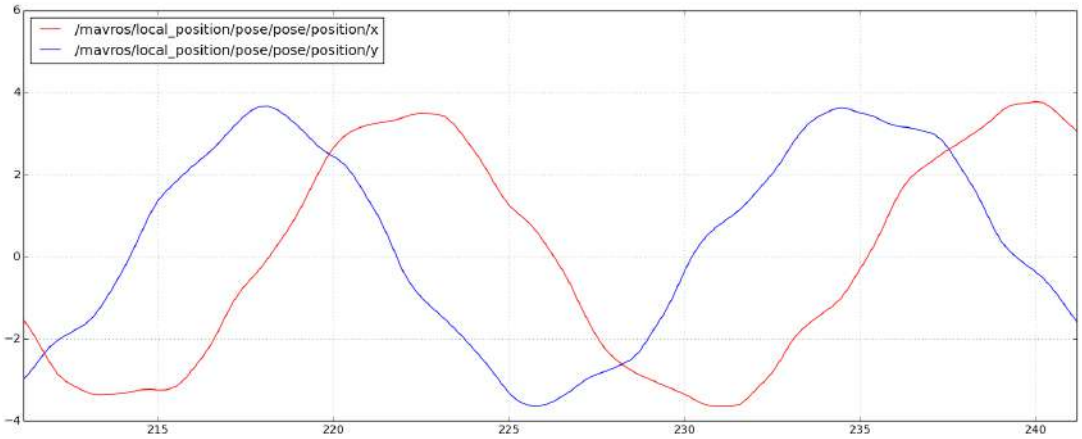


Figura 5.10: Gráfica del movimiento circular del dron con un radio de $3m$ utilizando el tópico *local_position*.

En la gráfica se observa que el tiempo que tarda el vehículo en realizar una vuelta completa es de 15 segundos, de igual manera se puede observar que el radio de $3m$ de la circunferencia establecida si es alcanzado; sin embargo, la senoidal que se obtiene de la posición del dron no se observa uniforme, esto indica que realizar el seguimiento de la trayectoria circular de esta manera no es muy adecuado, este problema podría ser mejorado implementando un método de control, ya sea con compensadores o utilizando un modelo dinámico integrado a la misma ley de control.

5.2.1. Error del seguimiento de una trayectoria

De la misma manera se utilizó la ecuación de error:

$$error = \sqrt{(x - x_o)^2 + (y - y_o)^2 + (z - z_o)^2)}$$

para la realización de la trayectoria circular que se implementa en el vehículo, en la gráfica 5.11 se muestra el error por coordenada y en la gráfica 5.12 el error total. Una observación que se puede hacer a partir de las gráficas, es que el error al realizar la

trayectoria va aumentando con el paso del tiempo debido a la falta de control del sistema.

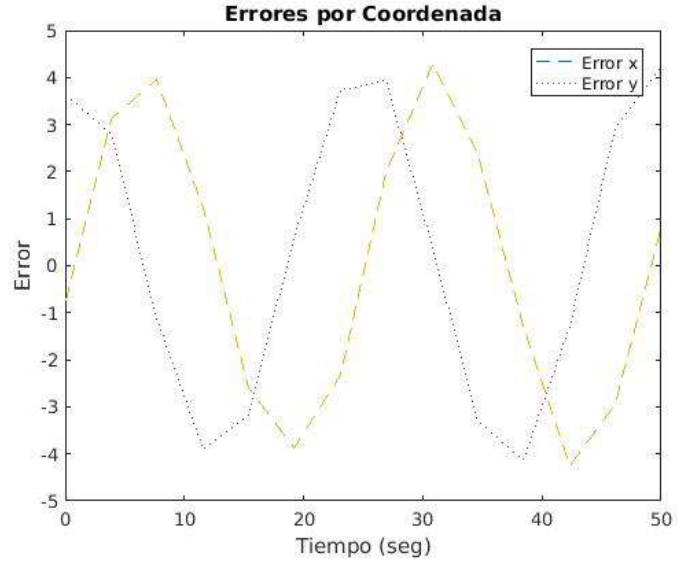


Figura 5.11: Errores por coordenada para la trayectoria circular con radio de $3m$.

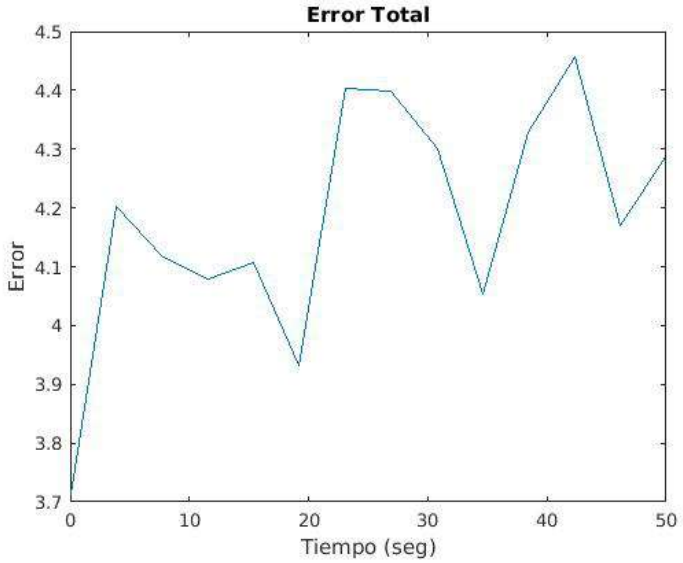


Figura 5.12: Error total para la trayectoria circular con radio de $3m$.

5.3. Aterrizaje con detección de marcadores

En lo hecho por [42] se implementó en el vehículo simulado *Erle-Copter* una técnica de *Visual Servoing* con un algoritmo capaz de detectar los marcadores de Aruco y realizar una maniobra de aterrizaje sobre el mismo marcador. El vehículo visualiza el marcador utilizando su cámara inferior. El proceso de aterrizaje es controlado por un PID para realizar las correcciones de pose (alabeo, cabeceo y guiña) de forma automática sobreescribiendo los canales RC.

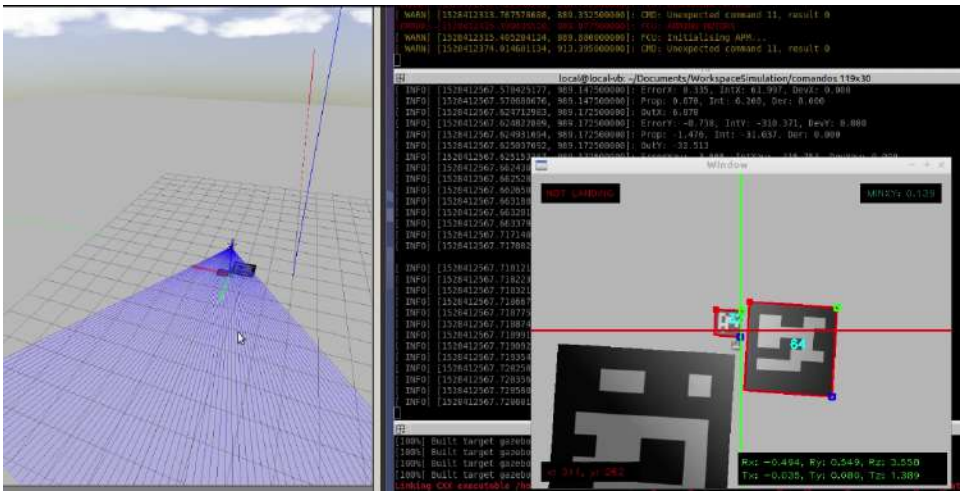


Figura 5.13: Vista de simulación para la detección de marcadores de Aruco.

Las pruebas se realizaron en el simulador Gazebo, elevando el vehículo a una altura 10m y activando el nodo del algoritmo para que comenzará la detección del marcador y con elló el descenso del dron con el fin de comparar el rendimiento de estimación de la posición del algoritmo propuesto contra el obtenido a través del GPS. Los datos comparados son los generados por el GPS, el algoritmo de visión y el tópico */mavros/local_position/pose/*. En la figura 5.14 se resume el comportamiento de la traslación en los ejes x, y y z que sufrió el dron con el paso del tiempo, se puede observar el rendimiento del algoritmo de visión computacional propuesto tuvo un rendimiento aceptable, muy apegado a lo informado por el servicio de *mavros*. El eje x y y tiene ciertas oscilaciones como consecuencia del desplazamiento del vehículo y el funcionamiento del algoritmo en tiempo real que intenta compensar el empuje de los motores. De igual manera se puede apreciar que cuando el vehículo se ubica por encima del marcador, el algoritmo inicia de forma automática el proceso el proceso de aterrizaje disminuyendo la altura de forma gradual hasta aproximarse a tierra.

De igual manera se realizó un análisis de comparación entre los datos obtenidos del GPS del dron y el algoritmo de visión propuesto. La figura 5.14 muestra el resultado de la comparación.

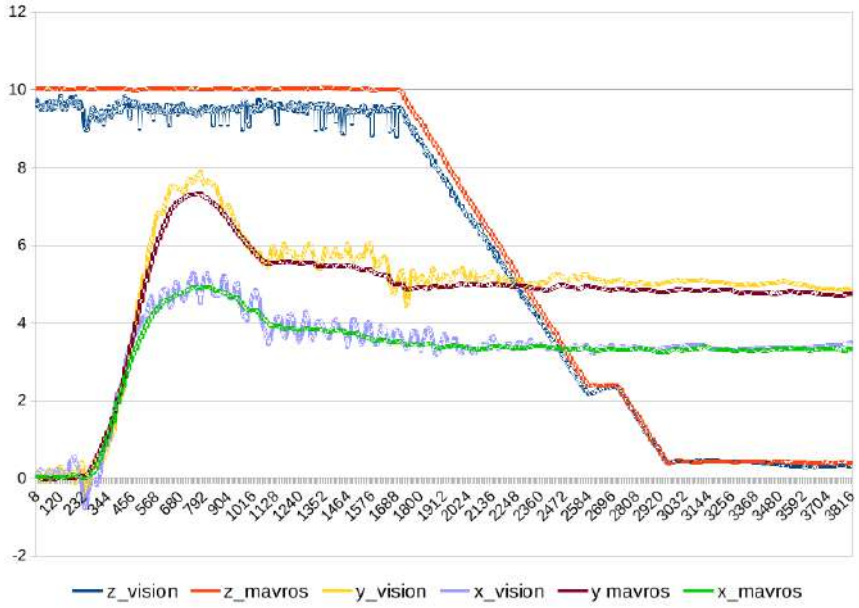


Figura 5.14: Evolución de la traslación. Algoritmo de visión contra servicio *mavros*.

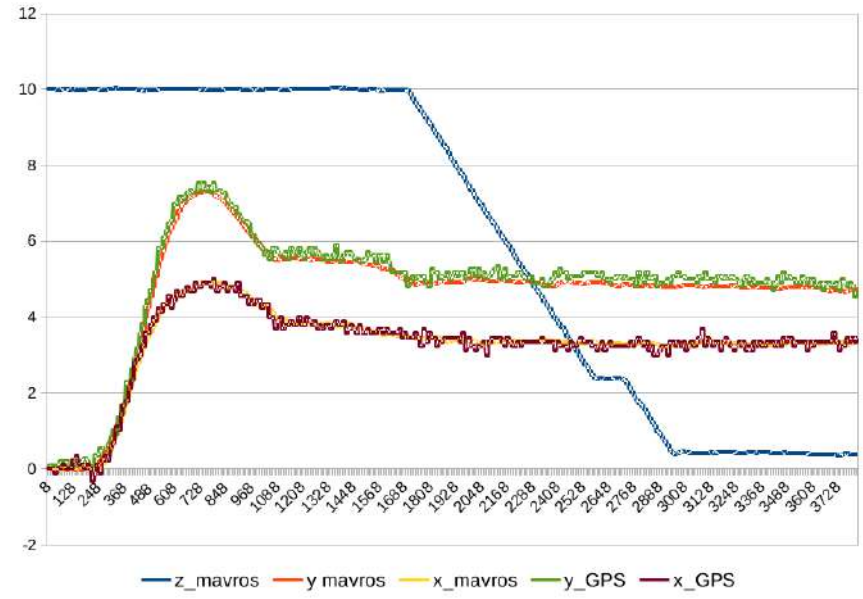


Figura 5.15: Evolución de la traslación. GPS contra servicio *mavros*.

5.3.1. Error del aterrizaje con detección de marcadores

Realizando la comparación entre el GPS y el algoritmo propuesto se puede observar que proveen un comportamiento bastante similar que se apega al tópico */mavros/-local_position/pose/*. Como se puede observar en la tabla 5.1, el GPS ofrece mejores resultados en general; sin embargo, el algoritmo de visión tiene un mejor comportamiento a bajas altitudes, lo que es muy adecuado debido a que el aterrizaje es lo más cercano posible al centro del marcador.

Módulo	Error	x	y
GPS	Mínimo	0.062 cm	0.1 cm
	Máximo	41.87 cm	73.33 cm
	Promedio	14.324 %	12.717 %
Visión	Mínimo	0.011 cm	0.014 cm
	Máximo	77.586 cm	81.4789 cm
	Promedio	16.124 %	14.001 %

Cuadro 5.1: Estadísticas de error.

Si se analizan las gráficas de error absoluto, se puede observar como el error disminuye a medida que la aeronave se aproximaba al centro del marcador. El error observado es el resultado de las correcciones realizadas para ajustar la posición, lo que ocasionaba que el dron se desplazara horizontal y verticalmente cambiando su distancia respecto al marcador. Como en un principio el error era significativo, la etapa de control del algoritmo propuesto generaba correcciones bruscas, dado a que su comportamiento es lineal pero la dinámica de vuelo de la aeronave no lo es.

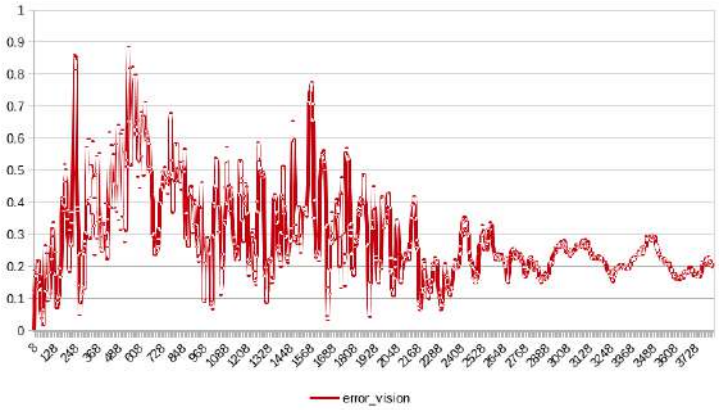


Figura 5.16: Error absoluto de traslación. Algoritmo de visión contra servicio *mavros*.

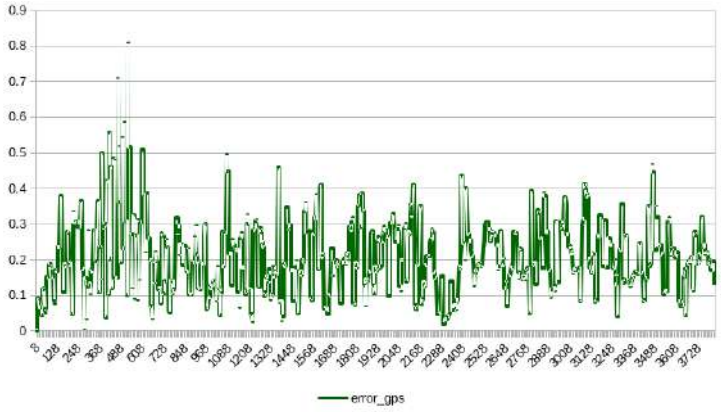


Figura 5.17: Error absoluto de traslación. GPS contra servicio *mavros*.

Capítulo 6

Conclusiones y Trabajo a futuro

En este trabajo se presentó el proceso de ensamblado de un vehículo aéreo modelo Tarot T960 detallando el montaje del hardware, en el cual no se presentó mayor problema excepto por lo laborioso que es llevar acabo las conexiones de los motores a los *ESC* y a la placa de distribución de potencia, además de las conexiones que tiene la computadora de vuelo. Como estación de control terrestre se utilizó el programa *QGroundControl* que es compatible con la computadora de vuelo *HK32Pilot* y sirve adecuadamente para cada una de las configuraciones y calibraciones que se le tienen que realizar al vehículo para que el dron sea capaz de realizar vuelos.

El desarrollo de un entorno exterior de simulación robótica se llevo a cabo usando el programa Gazebo diseñando un escenario virtual que muestra al laboratorio CLIR y lo que es su patio trasero, mismo escenario donde es posible realizar pruebas de vuelo del vehículo físico, demostrando que este simulador es un programa totalmente adecuado para la realización de proyectos en robótica que involucren el desarrollo de entornos exteriores. Un tema igual de importante para la tesis fue el uso del sistema ROS, ya que éste es necesario para llevar a cabo el desarrollo de programas en la computadora de vuelo del vehículo. Debido a que la realización de pruebas con un dron físico siempre es un poco riesgoso por los problemas que se pueden presentar durante la realización de las mismas se mostró el uso de un dron simulado, el cual es capaz de emular correctamente algoritmos que igualmente puedan ser implementados en el vehículo físico ya que ambos utilizan el puente mavros.

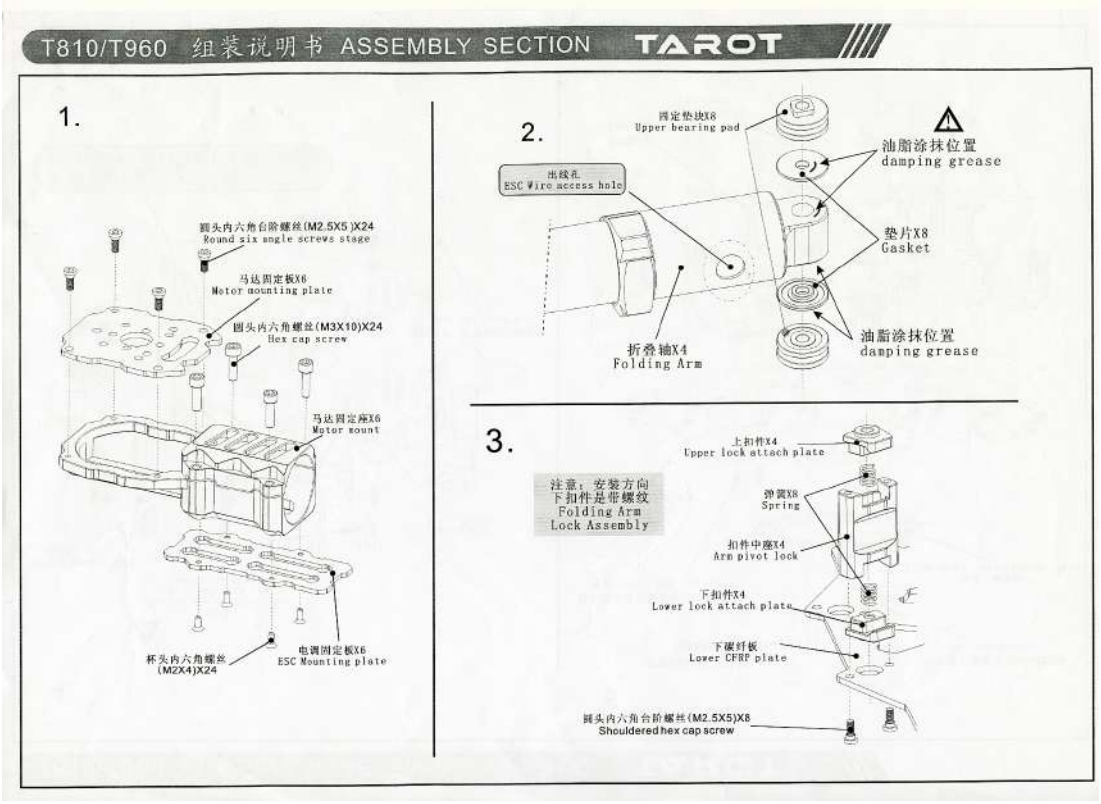
En la etapa experimental de este trabajo se presentaron algoritmos basados en el sistema ROS que sirven para programar la computadora de vuelo del vehículo aéreo, utilizando el vehículo simulado se realizó el control de posición que sirve para llevar al vehículo a una posición determinada en base a la tripleta de coordenadas x, y, z mostrando que el error que se genera es muy poco, concluyendo que el control es adecuado para realizar estos movimientos; sin embargo, al realizar el seguimiento de la trayectoria circular el vehículo mostró un error mayor realizando el seguimiento inadecuadamente mostrando en la gráfica de posición una senoidal no muy uniforme.

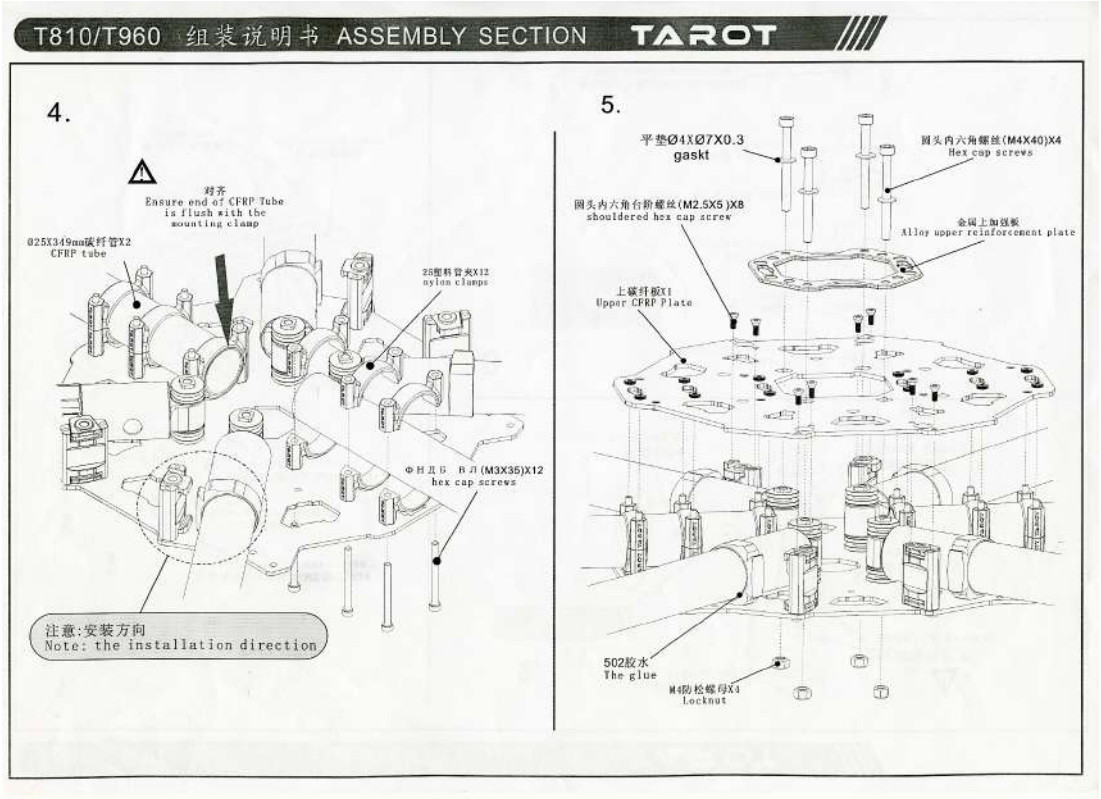
En este trabajo se presentó a detalle como llevar a cabo la programación de la computadora de vuelo para maniobrar el vehículo, así como un tema importante para el desarrollo de futuros trabajos que es el uso de imágenes con ROS y OpenCV y la función para la detección de los marcadores de Aruco. En el trabajo de tesis realizado simultáneamente por el Ing. Manuel Poot se muestra el uso de las imágenes obtenidas en el simulador desde el dron, en el trabajo se realiza la implementación de la técnica de *Visual Servoing* para que una vez que el algoritmo detecte un marcador de Aruco se realice una maniobra de aterrizaje sobre el mismo.

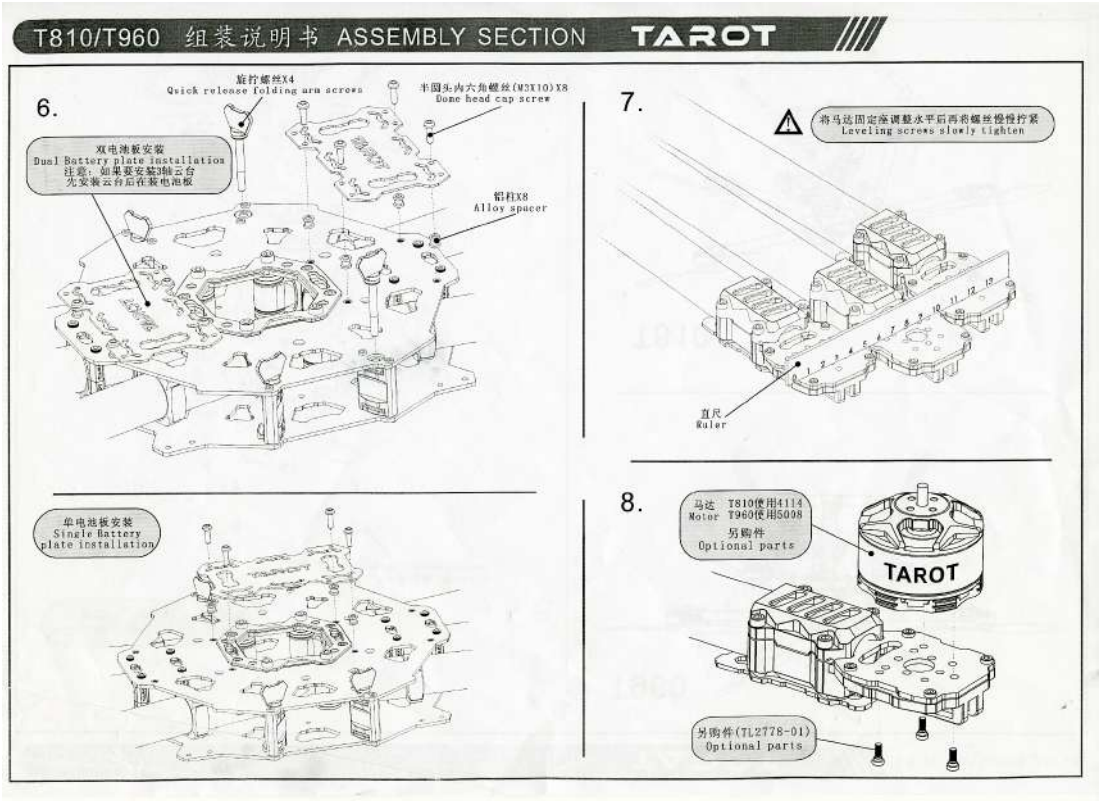
Una de las principales finalidades de este trabajo es que sirva de guía a posteriores alumnos de la maestría a realizar proyectos con drones de una manera más rápida dejando las bases suficientes para que se implementen algoritmos de visión computacional en el vehículo, que es una de las metas principales que se tienen en el laboratorio. Por consiguiente, no perdiendo tanto tiempo en el armado del dron, en las calibraciones que se requieren y con el conocimiento básico necesario en el sistema ROS para poder programar la computadora del vehículo.

Apéndice A

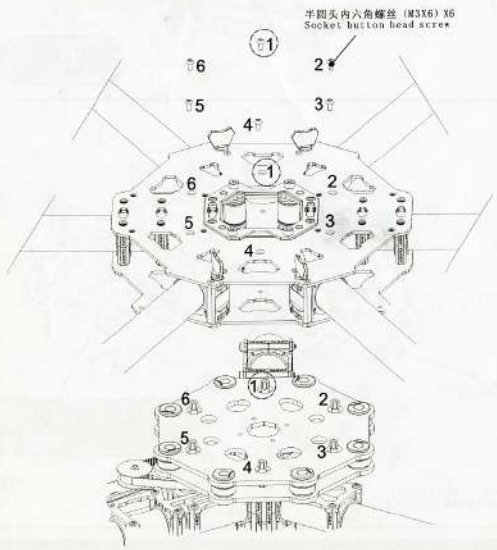
Manual de Ensamblado Tarot T960



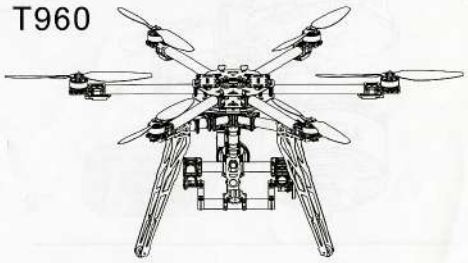

T810/T960 组装说明书 ASSEMBLY SECTION TAROT
4.
对齐
Ensure end of CFRP Tube is flush with the mounting clamp
Ø25X349mm碳纤管X2
CFRP tube
25型料管夹X12
nylon clamps
M3X35)X12
hex cap screws
注意:安装方向
Note: the installation direction
5.
平垫Ø4XØ7X0.3
gaskt
圆头内六角螺丝(M4X40)X4
Hex cap screws
圆头内六角台阶螺丝(M2.5X5)X8
shouldered hex cap screw
金属上加强板
Alloy upper reinforcement plate
上碳纤板X1
Upper CFRP Plate
502胶水
The glue
M4防松螺母X4
Locknut


T810/T960 组装说明书 ASSEMBLY SECTION TAROT
6.
旋拧螺丝X4
Qsick release folding arm screws
半圆头内六角螺丝(M3X10)X8
Dome head cap screw
双电池板安装
Dual Battery plate installation
注意：如果要安装3轴云台
先安装云台后在装电池板
铝柱X8
Alloy spacer
单电池板安装
Single Battery plate installation
7.
将马达固定座调整水平后再将螺丝慢慢拧紧
Leveling screws slowly tighten
直尺
Ruler
8.
马达 T810使用4114
Moter T960使用5008
另购件
Optional parts
TAROT
另购件(TL2778-01)
Optional parts

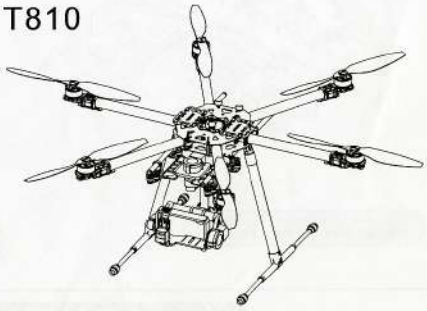
T810安装3轴云台孔位置说明
注意:先装3轴云台再装电池板



T960



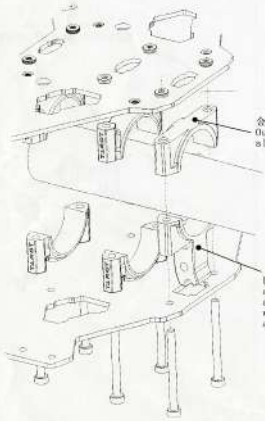
T810



（脚架版本）脚架组装
Tricycle landing gear assembly

1.

脚架版本金属管夹与
脚架连接头安装
Metal pipe clamps and
triped version Tripod
connector installation

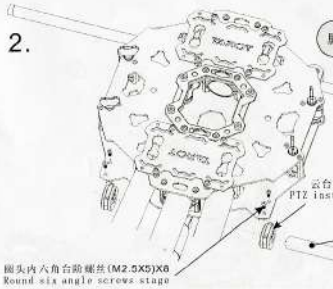


Replace the outboard Arm clamps of3
equally spaced arms(2,4,6)with the provided
alloy clamps

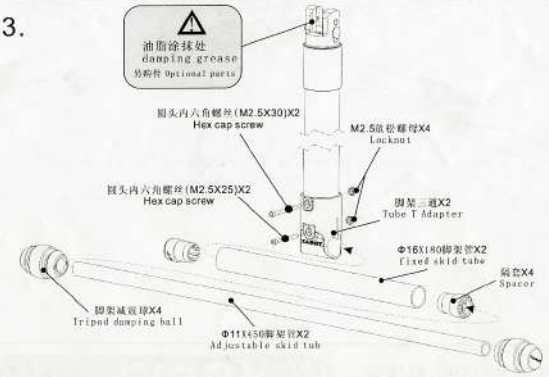
另购件 Optional parts

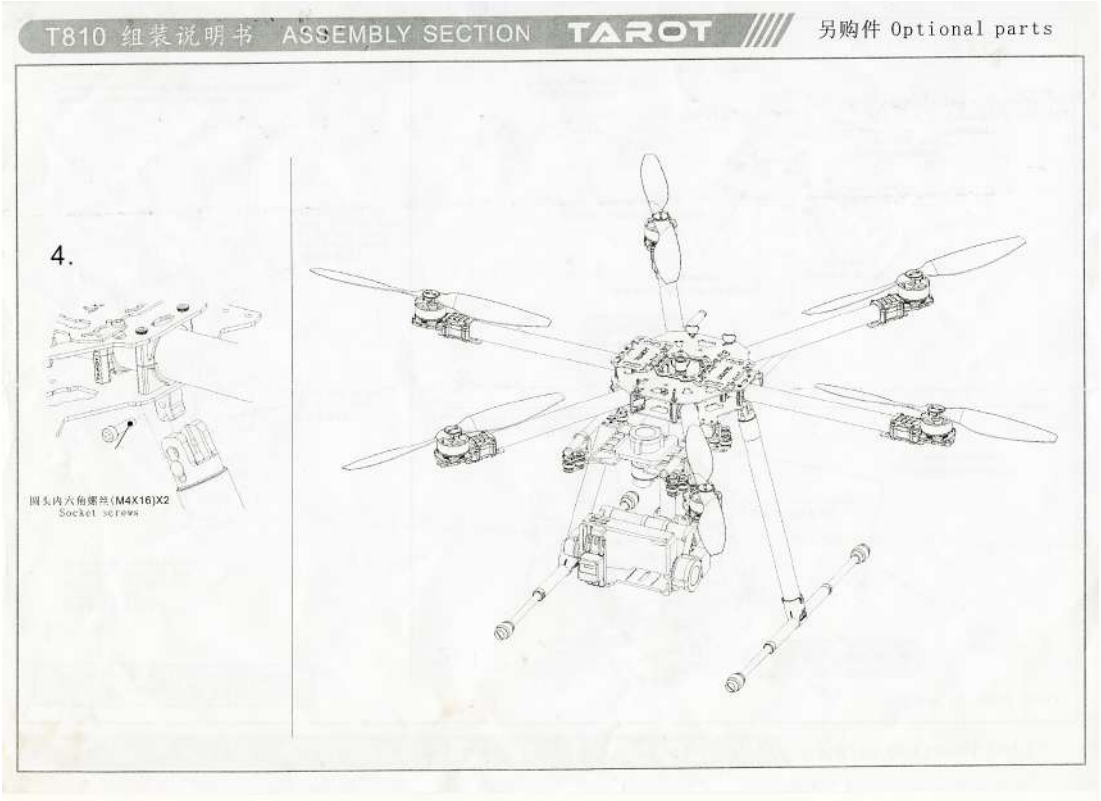
2.

脚架版本云台安装管安装



3.





Este manual se encuentra disponible en [43].

Apéndice B

Controles del transmisor RC

Apéndice C

Comandos ROS

C.1. Paquetes

En este apartado se presentan algunos comandos de terminal específicos para trabajar con ROS.

Para crear, modificar y trabajar con los paquetes de ROS, algunos comandos son:

catkin_create_pkg: comando usado para crear un paquete nuevo.

rospack: comando usado para obtener información del paquete en el sistema de archivos.

catkin_make: comando usado para compilar paquetes en el espacio de trabajo.

rosdep: comando usado para instalar dependencias del sistema requeridas por el paquete.

Para navegar y manipular los paquetes

roscd: comando usado para navegar entre carpetas de ROS.

roscp: comando usado para copiar un archivo de un paquete.

rosed: comando usado para editar un archivo.

rosrun: comando usado para correr un archivo ejecutable dentro de un paquete.

C.2. Nodos

El comando *rosnode* es usado para obtener información de los nodos

$ *rosnode info [node_name]*: imprimirá información sobre el nodo.

$ *rosnode kill [node_name]*: detiene a un nodo que se está ejecutando.

$ *rosnode list*: genera una lista de los nodos que se están ejecutando.

$ *rosnode machine [machine_name]*: genera una lista de los nodos corriendo en una máquina particular o lista de máquinas.

$ *rosnode ping*: revisa la conectividad del nodo.

$ *rosnode cleanup*: purga el registro de los nodos no alcanzables.

C.3. Mensajes

El comando *rosmsg* puede ser usado para inspeccionar el encabezado de los mensajes. El siguiente comando ayuda a ver los encabezados de un mensaje particular.

$ *rosmsg show std_msgs/Header*

Algunos parámetros que son utilizados comúnmente con *rosmsg* se muestran continuación:

$ *rosmsg show [message]*: muestra la descripción del mensaje.

$ *rosmsg list*: genera una lista todos los mensajes.

$ *rosmsg md5 [message]*: muestra la suma *md5* de cada mensaje.

$ *rosmsg package [package_name]*: enlista los mensajes en un paquete.

$ *rosmsg packages [package_1] [package_2]*: enlista paquetes que contienen mensajes.

C.4. Tópicos

La herramienta para tópicos que brinda información de los mismos y puede ser usada de la siguiente manera

$ *rostopic bw /topic*: muestra el ancho de banda usado por el tópico.

$ *rostopic echo /topic*: imprime el contenido del tópico.

$ *rostopic find /message_type*: busca tópicos que usen el mensaje que se indica.

$ *rostopic hz /topic0*: muestra la taza de publicación del tópico que se indica.

$ *rostopic info /topic*: imprime información sobre un tópico que se encuentra activo.

$ *rostopic list*: enlista todos los tópicos que se encuentren activos en el sistema

$ *rostopic pub /topic message_type args*: sirve para publicar un valor a un tópico con un tipo de mensaje.

$ *rostopic type /topic*: muestra el tipo de mensaje del tópico.

C.5. Servicios

Los siguientes comandos muestran como obtener información de los servicios de ROS.

$ *rosservice call /service args*: llama al servicio usando los argumentos dados.

$ *rosservice find service_type*: encuentra los servicios que tengan el tipo de servicio que se indica.

$ *rosservice info /services*: imprime información sobre el servicio indicado.

$ *rosservice list*: imprime una lista de los servicios que están activos en el sistema.

$ *rosservice type /service*: imprime el tipo de servicio que ofrece el servicio dado.

$ *rosservice uri /service*: imprime el servicios ROSRPC URI.

C.6. Bags

El comando *rosbag* es usado para trabajar con archivos tipo *bag*.

$ *rosbag record [topic_1] [topic_2] -o [bag_name]*: almacena el tópico dado dentro de un archivo *bag* que es dado en el comando. De la misma manera se pueden almacenar todos los tópicos usando el argumento -a.

$ *rosbag play [bag_name]*: reproduce el archivo *bag* existente.

C.7. Servidor de Parámetros

La herramienta *rosparam* es usada para obtener y configurar parámetros de ROS desde la línea de comandos.

$ *rosparam set [parameter_name] [value]*: establece un valor el parámetro dado.

$ *rosparam get [parameter_name]*: recupera un valor del parámetro dado.

$ *rosparam load [YAML file]*: guarda en un archivo YAML parámetros de ROS y cargarlos al servidor de parámetros.

$ *rosparam dump [YAML file*]: vuelca parámetros existentes a un archivo YAML

$ *rosparam delete [parameter_name]*: elimina el parámetro dado.

$ *rosparam list*: genera una lista con nombres de los parámetros existentes.

Apéndice D

Códigos ejemplo

D.1. Código para el manejo de los canales RC

```
#include <cstdlib>
#include <ros/ros.h>
#include <mavros_msgs/CommandBool.h>
#include <mavros_msgs/CommandTOL.h>
#include <mavros_msgs/SetMode.h>

#include <iostream>
#include <mavros_msgs/OverrideRCIn.h>
#include <mavros_msgs/State.h>

#define MINRC   1100
#define BASERC  1500
#define MAXRC   1900

// RC publisher
ros::Publisher pub;

double Roll, Pitch;

int main(int argc, char **argv)
{
    int rate = 10;
    ros::init(argc, argv, "mavros_takeoff");
    ros::NodeHandle n;
    ros::Rate r(rate);

    //Modo Guiado
    printf("Cambiando a modo Guiado \n");
    ros::ServiceClient cl = n.serviceClient<mavros_msgs::SetMode>
    ("/mavros/set_mode");
    mavros_msgs::SetMode srv_setMode;
    srv_setMode.request.base_mode = 0;
    srv_setMode.request.custom_mode = "GUIDED";
```

```
    if(cl.call(srv_setMode)){
        ROS_ERROR("setmode send ok %d value:",
        srv_setMode.response.success);
    }else{
        ROS_ERROR("Failed SetMode");
        return -1;
    }
    //Armado
    ros::ServiceClient arming_cl = n.serviceClient<mavros_msgs::
    CommandBool>("/mavros/cmd/arming");
    mavros_msgs::CommandBool srv;
    srv.request.value = true;
    if(arming_cl.call(srv)){
        ROS_ERROR("ARM send ok %d", srv.response.success);
    }else{
        ROS_ERROR("Failed arming or disarming");
    }
    //Despegue
    printf("Realizando Despeque Vertical \n");
    ros::ServiceClient takeoff_cl = n.serviceClient<mavros_msgs::
    CommandTOL>("/mavros/cmd/takeoff");
    mavros_msgs::CommandTOL srv_takeoff;
    srv_takeoff.request.altitude = 3;
    srv_takeoff.request.latitude = 0;
    srv_takeoff.request.longitude = 0;
    srv_takeoff.request.min_pitch = 0;
    srv_takeoff.request.yaw = 0;
    if(takeoff_cl.call(srv_takeoff)){
        ROS_ERROR("srv_takeoff send ok %d",
        srv_takeoff.response.success);
    }else{
        ROS_ERROR("Failed Takeoff");
    }


    //Espera
    sleep(10);
    //ros::ServiceClient cl = n.serviceClient<mavros_msgs::
    SetMode>("/mavros/set_mode");
    //mavros_msgs::SetMode srv_setMode;
    printf("Cambiando a modo LOITER \n");
    srv_setMode.request.base_mode = 0;
    srv_setMode.request.custom_mode = "LOITER";
    if(cl.call(srv_setMode)){
        ROS_ERROR("setmode send ok %d value:",
        srv_setMode.response.success);
    }else{
        ROS_ERROR("Failed SetMode");
        return -1;
    }
```

```
        pub = n.advertise<mavros_msgs::OverrideRCIn>
        ("/mavros/rc/override", 10);
        //Create RC msg
        mavros_msgs::OverrideRCIn msg;

        int i=0;
    do
        {
        if(i==0)
        {
        printf("Movimiento 0 \n");  //ESTABLE
        msg.channels[0] = BASERC;   //Roll
        msg.channels[1] = BASERC;   //Pitch
        msg.channels[2] = BASERC;   //Throttle
        i+=1;
        }
        else if(i==1)
        {
        printf("Movimiento 1 \n");  //ADELANTE
        msg.channels[0] = BASERC;   //Roll
        msg.channels[1] = 1400;     //Pitch
        msg.channels[2] = BASERC;   //Throttle
        i+=1;
        }
        else if(i==2)
        {
        printf("Movimiento 2 \n");  //IZQUIERDA
        msg.channels[0] = 1400;     //Roll
        msg.channels[1] = BASERC;   //Pitch
        msg.channels[2] = BASERC;   //Throttle

        i+=1;
        }
        else if(i==3)
        {
        printf("Movimiento 3 \n");  //ATRAS
        msg.channels[0] = BASERC;   //Roll
        msg.channels[1] = 1600;     //Pitch
        msg.channels[2] = BASERC;   //Throttle
        i+=1;
        }
        else if(i==4)
        {
        printf("Movimiento 4 \n");  //DERECHA
        msg.channels[0] = 1600;     //Roll
        msg.channels[1] = BASERC;   //Pitch
        msg.channels[2] = BASERC;   //Throttle
        i+=1;
        }

        else
        {
```

```
        printf("Movimiento 5 \n");  //ESTABLE
        msg.channels[0] = BASERC;   //Roll
        msg.channels[1] = BASERC;   //Pitch
        msg.channels[2] = BASERC;   //Throttle
        i+=1;
        }
        pub.publish(msg);
        sleep(7);

        }
        while(i<=5);
    //LAND
    printf("Comenzando Aterrizaje \n");
    ros::ServiceClient land_cl = n.serviceClient<mavros_msgs::
    CommandTOL>("/mavros/cmd/land");
    mavros_msgs::CommandTOL srv_land;
    srv_land.request.altitude = 3;
    srv_land.request.latitude = 0;
    srv_land.request.longitude = 0;
    srv_land.request.min_pitch = 0;
    srv_land.request.yaw = 0;
    if(land_cl.call(srv_land)){
        ROS_INFO("srv_land send ok %d",
        srv_land.response.success);
    }else{
        ROS_ERROR("Failed Land");
    }
        while (ros::ok())
        {
                ros::spinOnce();
                r.sleep();
        }
    return 0;
}
```

D.2. Código para el control de posición

```
#include <cstdlib>
#include <ros/ros.h>
#include <mavros_msgs/CommandBool.h>
#include <mavros_msgs/CommandTOL.h>
#include <mavros_msgs/SetMode.h>
#include <iostream>
#include <mavros_msgs/OverrideRCIn.h>
#include <mavros_msgs/State.h>
#include "math.h"
#include <geometry_msgs/PoseStamped.h>
#include "std_msgs/Float64.h"

using namespace std;
mavros_msgs::State current_state;
void state_cb(const mavros_msgs::State::ConstPtr& msg){
    current_state = *msg;
}

int main(int argc, char **argv)
{
    ros::init(argc, argv, "mavros_trayectoria");
    ros::NodeHandle n;
        ros::Rate rate(20);

    ros::Subscriber state_sub = n.subscribe<mavros_msgs::State>
    ("mavros/state", 10,state_cb);

    ros::Publisher local_pos_pub = n.advertise <geometry_msgs::
    PoseStamped >("mavros/setpoint_position/local", 10);
        geometry_msgs::PoseStamped pose;

    ros::ServiceClient arming_cl = n.serviceClient<mavros_msgs::
    CommandBool>("/mavros/cmd/arming");
    mavros_msgs::CommandBool srv;

    ros::ServiceClient cl = n.serviceClient<mavros_msgs::
    SetMode>("/mavros/set_mode");
    mavros_msgs::SetMode srv_setMode;

    ros::ServiceClient takeoff_cl = n.serviceClient<mavros_msgs::
    CommandTOL>("/mavros/cmd/takeoff");
    mavros_msgs::CommandTOL srv_takeoff;
    //FCU
    while(ros::ok() && !current_state.connected)
    {
         ros::spinOnce();
         rate.sleep();
    }
```

```
    // Armado
    srv.request.value = true;
    if(arming_cl.call(srv))
    {
        ROS_ERROR("ARM send ok %d", srv.response.success);
    }else{
        ROS_ERROR("Failed arming or disarming");
    }
    // Modo
    srv_setMode.request.base_mode = 0;
    srv_setMode.request.custom_mode = "GUIDED";
    if(cl.call(srv_setMode))
    {
        ROS_ERROR("setmode send ok %d value:",
        srv_setMode.response.success);
    }else{
    ROS_ERROR("Failed SetMode");
        return -1;
    }
    // Takeoff
    srv_takeoff.request.altitude = 3;
    srv_takeoff.request.latitude = 0;
    srv_takeoff.request.longitude = 0;
    srv_takeoff.request.min_pitch = 0;
    srv_takeoff.request.yaw = 0;
    if(takeoff_cl.call(srv_takeoff))
    {
        ROS_ERROR("srv_takeoff send ok %d",
        srv_takeoff.response.success);
    }else{
        ROS_ERROR("Failed Takeoff");
    }
    sleep(5);
    // Posicion
    pose.pose.position.x = 0;
    pose.pose.position.y = -9;
    pose.pose.position.z =  5;
    //send a few setpoints before starting
    for(int i = 100; ros::ok() && i > 0; --i)
    {
        local_pos_pub.publish(pose);
        ros::spinOnce();
        rate.sleep();
    }

    ros::Time last_request = ros::Time::now();
    while (ros::ok())
    {
        ros::spinOnce();
        rate.sleep();
    }
    return 0;
}
```

D.3. Código para obtener la posición local

```
#include "ros/ros.h"
#include "sensor_msgs/Imu.h"
#include <geometry_msgs/PoseStamped.h>


void local_pos(const geometry_msgs::PoseStamped::ConstPtr&  msg)
{
        ROS_INFO("\n Posicion local\
                \nx:[%f]  y:[%f]  z:[%f]",
                msg->pose.position.x,
                msg->pose.position.y,
                msg->pose.position.z);
}


int main(int argc, char **argv)
{
   ros::init(argc, argv, "local_position");
   ros::NodeHandle n;
   ros::Subscriber sub = n.subscribe("/mavros/local_position/
   pose",1000,local_pos);
   ros::spin();
   return 0;
}
```

Bibliografía

[1] Adam Haber, Matthew McGill, and Claude Sammut. Jmesim: An open source, multi platform robotics simulator. In *Proceedings of Australasian Conference on Robotics and Automation (December 2012)*, 2012.

[2] Lentin Joseph. *ROS Robotics Projects*. Packt Publishing Ltd., 2017.

[3] Oualid Araar and Nabil Aouf. Visual servoing of a quadrotor uav for autonomous power lines inspection. In *Control and Automation (MED), 2014 22nd Mediterranean Conference of*, pages 1418–1424. IEEE, 2014.

[4] Marinela Georgieva Popova. *Visual Servoing for a Quadrotor UAV in Target Tracking Applications*. PhD thesis, 2015.

[5] Wang Chao. Vision-based autonomous control and navigation of a uav. 2014.

[6] Daewon Lee, Hyon Lim, H Jin Kim, Youdan Kim, and Kie Jeong Seong. Adaptive image-based visual servoing for an underactuated quadrotor system. *Journal of Guidance, Control, and Dynamics*, 35(4):1335–1353, 2012.

[7] Wei Qian, Zeyang Xia, Jing Xiong, Yangzhou Gan, Yangchao Guo, Shaokui Weng, Hao Deng, Ying Hu, and Jianwei Zhang. Manipulation task simulation using ros and gazebo. In *Robotics and Biomimetics (ROBIO), 2014 IEEE International Conference on*, pages 2594–2598. IEEE, 2014.

[8] Ilya Afanasyev, Artur Sagitov, and Evgeni Magid. Ros-based slam for a gazebo-simulated mobile robot in image-based 3d model of indoor environment. In *International Conference on Advanced Concepts for Intelligent Vision Systems*, pages 273–283. Springer, 2015.

[9] Johannes Meyer, Alexander Sendobry, Stefan Kohlbrecher, Uwe Klingauf, and Oskar Von Stryk. Comprehensive simulation of quadrotor uavs using ros and gazebo. In *International conference on simulation, modeling, and programming for autonomous robots*, pages 400–411. Springer, 2012.

[10] Mengmi Zhang, Hailong Qin, Menglu Lan, Jiaxin Lin, Shuai Wang, Kaijun Liu, Feng Lin, and Ben M Chen. A high fidelity simulator for a quadrotor uav using

ros and gazebo. In *Industrial Electronics Society, IECON 2015-41st Annual Conference of the IEEE*, pages 002846–002851. IEEE, 2015.

[11] Iker Zamora, Nestor Gonzalez Lopez, Victor Mayoral Vilches, and Alejandro Hernandez Cordero. Extending the openai gym for robotics: a toolkit for reinforcement learning using ros and gazebo. *arXiv preprint arXiv:1608.05742*, 2016.

[12] Owen McAree, Jonathan M Aitken, and Sandor M Veres. A model based design framework for safety verification of a semi-autonomous inspection drone. In *Control (CONTROL), 2016 UKACC 11th International Conference on*, pages 1–6. IEEE, 2016.

[13] Shuyuan Wang and Tianjiang Hu. Ros-gazebo supported platform for tag-in-loop indoor localization of quadrocopter. In *International Conference on Intelligent Autonomous Systems*, pages 185–197. Springer, 2016.

[14] Sergio Salazar, Hugo Romero, Rogelio Lozano, and Pedro Castillo. Modeling and real-time stabilization of an aircraft having eight rotors. In *Unmanned Aircraft Systems*, pages 455–470. Springer, 2008.

[15] Pedro Castillo, Rogelio Lozano, and Alejandro E Dzul. *Modelling and control of mini-flying machines*. Physica-Verlag, 2006.

[16] DRONEBLY. Quadcopter vs hexacopter vs octocopter: The pros and cons. `http://dronebly.com/quadcopter-vs-hexacopter-vs-octocopter-the-pros-and-cons`. [Accessed: 2018-06-26].

[17] H Bouadi, M Bouchoucha, and M Tadjine. Modelling and stabilizing control laws design based on backstepping for an uav type-quadrotor. *IFAC Proceedings Volumes*, 40(15):245–250, 2007.

[18] fundéu. dron, adaptación al español de drone. `https://www.fundeu.es/recomendacion/dron-adpatacion-al-espanol-de-drone/`. [Accessed: 2018-06-26].

[19] Tarot. Partes del tarot t810 / t960. `http://www.tarot-rc.com/index.php?main_page=index&cPath=65_97`. [Accessed: 2018-07-04].

[20] FPVMAX. Hélices para drones: Tipos y tamaños. `http://fpvmax.com/2017/02/10/helices-drones-tipos-tamanos/`. [Accessed: 2018-07-11].

[21] Erle Robotics. Lipo batteries. `https://erlerobotics.gitbooks.io/erle-robotics-erle-copter/es/safety/lipo.html`. [Accessed: 2018-07-10].

[22] Federal Aviation Administration. *Advanced Avionics Handbook*. Federal Aviation Administration, 2009.

[23] HaiYang Chao, YongCan Cao, and YangQuan Chen. Autopilots for small unmanned aerial vehicles: a survey. *International Journal of Control, Automation and Systems*, 8(1):36–44, 2010.

[24] Dronecode. Qgroundcontrol user guide. `https://docs.qgroundcontrol.com/en/`. [Accessed: 2018-11-02].

[25] Open Source Robotics Foundation. Gazebo. `http://gazebosim.org/`. [Accessed: 2017-11-24].

[26] Serena Ivaldi, Vincent Padois, and Francesco Nori. Tools for dynamics simulation of robots: a survey based on user feedback. *arXiv preprint arXiv:1402.7050*, 2014.

[27] Open Source Robotics Foundation. Sdf. `http://sdformat.org/`. [Accessed: 2017-11-24].

[28] Carol Fairchild and Thomas L Harman. *ROS Robotics By Example: Learning to control wheeled, limbed, and flying robots using ROS Kinetic Kame*. Packt Publishing Ltd, 2017.

[29] Anil Mahtani, Luis Sanchez, Enrique Fernandez, and Aaron Martinez. *Effective Robotics Programming with ROS*. Packt Publishing Ltd, 2016.

[30] Aaron Martinez and Enrique Fernández. *Learning ROS for robotics programming*. Packt Publishing Ltd, 2013.

[31] Lentin Joseph. *Mastering ROS for robotics programming*. Packt Publishing Ltd, 2015.

[32] ROS.org. Creating a workspace for catkin. `http://wiki.ros.org/catkin/Tutorials/create_a_workspace`. [Accessed: 2018-11-25].

[33] MAVLINK. Mavlink. `https://mavlink.io/en/`. [Accessed: 2018-11-03].

[34] Erle Robotics. Mavlink. `https://erlerobotics.gitbooks.io/erlerobot/en/mavlink/mavlink.html`. [Accessed: 2018-11-03].

[35] Shyam Balasubramanian. Mavlink tutorial for absolute dummies. `http://api.ning.com/files/i*tFWQTF2R*7Mmw7hksAU-u9IABKNDO9apguOiSOCfvi2znk1tXhur0Bt00jTOldFvob-Sczg3*lDcgChG2MAVLINK_FOR_DUMMIESPart1_v.1.1.pdf`. [Accessed: 2018-11-03].

[36] ROS.org. mavros. `http://wiki.ros.org/mavros`. [Accessed: 2018-11-11].

[37] Seth Hutchinson, Gregory D Hager, and Peter I Corke. A tutorial on visual servo control. *IEEE transactions on robotics and automation*, 12(5):651–670, 1996.

[38] Francois Chaumette and Seth Hutchinson. Visual servo control. I. Basic approaches [Tutorial]. *IEEE Robotics & Automation Magazine*, 13(4):82–90, 2006.

[39] Aruco: a minimal library for augmented reality applications based on opencv. `https://www.uco.es/investiga/grupos/ava/node/26`. [Accessed: 2018-03-07].

[40] S. Garrido-Jurado, R. Muñoz-Salinas, F.J. Madrid-Cuevas, and M.J. Marín-Jiménez. Automatic generation and detection of highly reliable fiducial markers under occlusion. *Pattern Recognition*, 47(6):2280–2292, 2014.

[41] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[42] Manuel Esteban Poot Chin. Desarrollo de un sistema de visión computacional para el vuelo autónomo de un vant. Master's thesis, Universidad Autónoma de Yucatán, 2018.

[43] Mtarot t810 - t960 assembly manual. `http://cyanscorpion.com/tarot-t810-t960--manual.html`. [Accessed: 2018-07-02].